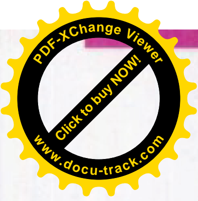
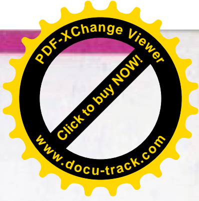



Python for UNIX and Linux System Administration

Python

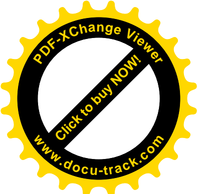
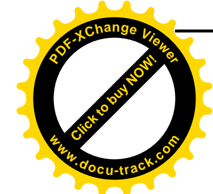
UNIX和Linux系统管理指南

O'REILLY®

机械工业出版社
China Machine Press

Noah Gift & Jeremy M. Jones 著

杨明华 谭励 等译

Python UNIX和Linux系统管理指南

Noah Gift & Jeremy M. Jones 著

杨明华 谭励 等译

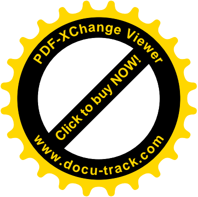
O'REILLY®

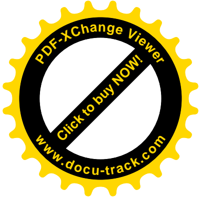
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc.授权机械工业出版社出版

机械工业出版社

www.TopSage.com





图书在版编目（CIP）数据

Python UNIX和Linux系统管理指南/（美）基弗特（Gift, N.）等著；杨明华等译.
－北京：机械工业出版社，2009.9
（O'Reilly精品图书系列）
书名原文：Python for UNIX and Linux System Administration
ISBN 978-7-111-26663-1

I. P… II. ①基… ②杨… III. ①软件工具－程序设计 ②Unix操作系统－程序设计
③Linux操作系统－程序设计 IV. TP311.56 TP316.8
中国版本图书馆CIP数据核字（2009）第043665号

北京市版权局著作权合同登记
图字：01-2009-1576号

©2008 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Machine Press, 2008. Authorized translation of the English edition, 2009 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出版权和销售权的所有者 —— O'Reilly Media, Inc.的许可。

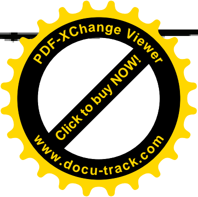
版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

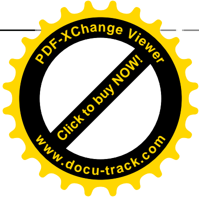
本书法律顾问
北京市展达律师事务所

书　　名/ Python UNIX和Linux系统管理指南
书　　号/ ISBN 978-7-111-26663-1
责任编辑/ 盛东亮
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（邮政编码100037）
印　　刷/ 北京京北印刷有限公司印刷
开　　本/ 178毫米×233毫米　16开本　27印张
版　　次/ 2009年9月第1版　2009年9月第1次印刷
定　　价/ 65.00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

www.TopSage.com





O'Reilly Media, Inc.介绍

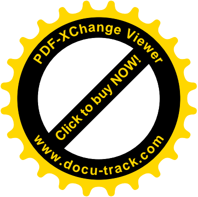
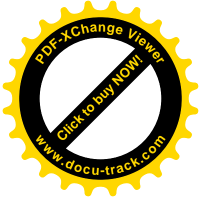
为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc.授权机械工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

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' 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方针。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联系着，所以O'Reilly Media, Inc.知道市场上真正需要什么图书。

www.TopSage.com

译者序

系统管理员每天都会面临许多繁杂而琐碎的工作，这些工作往往需要耗费大量的时间和精力，令许多系统管理员疲于应对。Python的出现为系统管理员带来了希望，如资深系统管理员Æleen Frisch在使用其他语言从事了多年的编程工作之后，第一次使用Python时所体会到的："它就像冬日过后一缕清新的空气，一束温暖的阳光。"

Python是一种简单易学、功能强大的编程语言，也是世界上发展速度最快的语言之一。在最近的计算机语言热度排名中，Python已跃至第七位，仅排在Java、C、C++、VB、PHP和C#之后。Python在大多数平台上的各种应用中都是理想的脚本语言，特别适用于快速的应用程序开发。著名的搜索引擎Google也大量使用了Python脚本，而在Nokia智能手机所采用的Symbian操作系统上，Python也成为继C++和Java之后的第三种编程语言。Python拥有一个强大的基本类库和数量众多的第三方扩展，其丰富程度可以与Java的JDK相媲美。将Python应用于系统管理，无疑会让系统管理员如虎添翼。

本书的作者有多年学习和使用Python的实践经验，并且该书经过多位评审专家的严格审核，集成了众多Python程序员、爱好者的智慧，仅从长长的致谢列表中就可以看到作者对本书所倾注的心血。

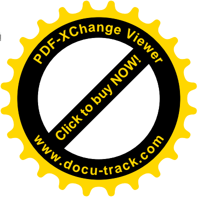
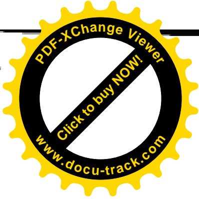
本书构思精巧，在知识点的组织和示例的选择上精心安排，每一章节都提出了具体的管理问题，并通过示例逐一给出了完整的解决方案。书中提供了大量的示例代码。这些精心构建的示例可以帮助读者由浅入深地领悟Python的精髓。以书中的示例为参照，读者完全可以开发出一套适用于自己的工具来解决遇到的实际问题。而这也是本书的一大特色。

本书内容浅显易懂，非常适合于初、中级Python程序员，也无疑会成为系统管理员手中的必备手册。

参与本书翻译工作的人员还包括张西广、成保栋、王振海、关志涛。于炯和张常有教授审阅了全书，并提出了宝贵意见。

由于时间仓促，译者水平有限，在翻译过程中难免会出现一些错误，恳请读者批评指正。

www.TopSage.com

作者简介

Noah Gift是加州州立大学洛杉矶分校的CIS硕士、加州理工学院圣路易斯奥比斯波营养学学士、Apple和LPI认证系统管理员，曾就职于加州理工学院、迪斯尼动画公司、索尼图像和Turner工作室。

在闲暇时间里，他喜欢与妻子Leah、儿子Liam一起弹钢琴和做运动。

Jeremy M. Jones是一名软件工程师，现任职于Predictix。他选择的开发工具是Python，而他对shell、Perl也有一定研究，了解Java的相关知识，当前在学习C#。他对函数式编程语言（尤其是OCaml）非常感兴趣。

他是开放源码项目Munkware的开发者，Munkware是一个多生产者/多消费者、事务性、持久队列机制的项目；他还是ediplex的开发者，ediplex是一个EDI（电子数据交换）解析引擎。此外，他也是podgrabber的开发者，podgrabber是一个podcast下载器。以上三个项目都是由Python语言编写。

Jeremy将他的空闲时间花在家庭生活和写作上。他和他的妻子Debra以及两个孩子Zane和Justus住在Georgia（Atlanta的东部）的Conyers，那里有一个名为Genevieve的实验室。

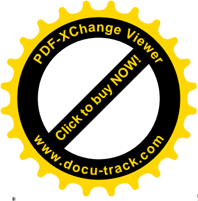
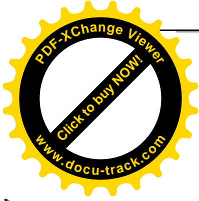
Jeremy所表达的想法和观点仅代表他个人，不代表Predictix的观点。

封面介绍

本书封面上的图片是一个红尾蚺（boa constrictor）。在整个南美洲和中美洲一些岛屿以及加勒比地区都发现有它们的踪迹，红尾蚺不是毒蛇，它们可以生活在各种各样的环境中，从沙漠到热带草原或是湿热的热带森林，但它们更喜欢生活在干旱地区中因地形而形成的潮湿环境中。它们大都依赖陆地和乔木生活，但是当渐渐地长大，它们往往花费更多的时间在地面上。

红尾蚺有非常独特的标记，包括钻石和椭圆状花纹。表皮的颜色和花纹取决于它们的栖息地，帮助它们能够更好地隐藏，以便狩猎森林中的各种动物。

www.TopSage.com

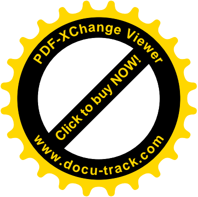
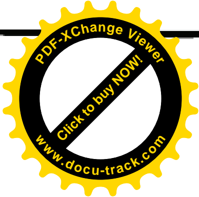



在野生环境中，红尾蚺能够扑食中小啮齿动物、蜥蜴、蝙蝠、鸟类、猫鼬、松鼠，甚至也可以扑食豹猫等一些其他较大的哺乳动物。红尾蚺冷血而且移动缓慢，它们可以在捕获大型猎物之后一个星期都不再进食。它们习惯于独行并且夜间狩猎，在它们的头上有热感装置，帮助它们寻找猎物。红尾蚺特别喜欢扑食蝙蝠，它们挂在树木或洞穴的入口等着，蝙蝠一旦飞过它们就可以一口咬住蝙蝠。不足为奇的是，红尾蚺依靠收缩使猎物致命。 蛇身就像包裹在猎物身体周围的线圈，紧缩地控制每次猎物的呼吸，最终使猎物窒息死亡。

红尾蚺在动物园中十分常见，它们也是相对常见的宠物。事实上，每年都要花费大量的美元进口它们到美国。在南非，它们被尊为“啮齿动物中的驱逐舰”，而且人们也往往出于这个原因去驯化它们。红尾蚺在洞穴中生活相当温和，可以静静的生活在那里20~30年。然而它们因为宠物贸易和装饰市场的需要而惨遭猎杀，一些红尾蚺属于濒危动物，应当受到保护。

红尾蚺季节性育种。为了吸引雄性，雌性从泄殖腔发出气味，泄殖腔是其肠道和泌尿生殖道出口。受精发生在体内，雌性红尾蚺可以同时生育多达60个婴儿。新生的红尾蚺平均在2英尺长，明显小于其堂兄弟水蟒。出生后红尾蚺可以长到13英尺长，体重超过100磅。在南美洲发现的最大红尾蚺纪录是18英尺！

www.TopSage.com

目录

序 1

前言 3

第1章 Python简介 11

为什么要选Python 11

学习的动力 17

一些基础知识 18

在Python中执行命令 19

在Python中使用函数 23

通过Import语句实现代码复用 26

第2章 IPython 31

安装IPython 32

基础知识 33

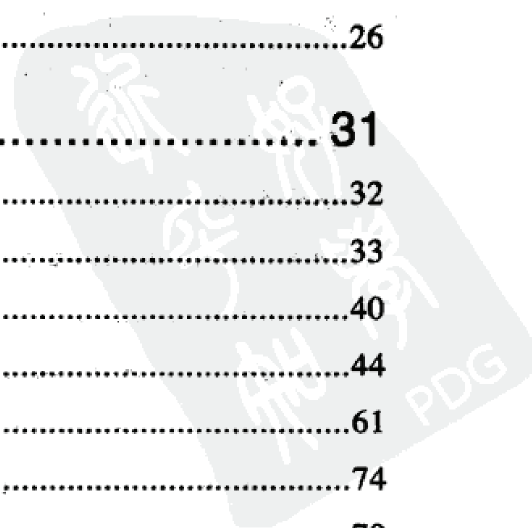
从功能强大的函数获得帮助 40

UNIX Shell 44

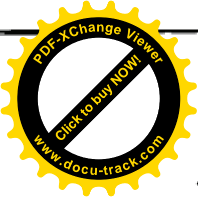
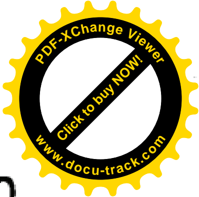
信息搜集 61

自动和快捷方式 74

本章小结 79



www.TopSage.com

第3章 文本 80

Python的内建功能及模块 80

ElementTree 124

本章小结 127

第4章 文档与报告 129

自动信息收集 129

手工信息收集 132

信息格式化 141

信息发布 147

本章小结 151

第5章 网络 152

网络客户端 152

远程过程调用 163

SSH 169

Twisted 171

Scapy 177

使用Scapy创建脚本 179

第6章 数据 182

引言 182

使用 OS 模块与Data进行交互 183

拷贝、移动、重命名和删除数据 184

使用路径、目录和文件 186

数据比较 189

合并数据 192

对文件和目录的模式匹配 197

包装rsync 199

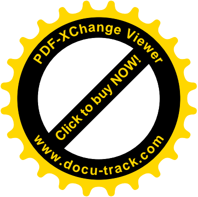
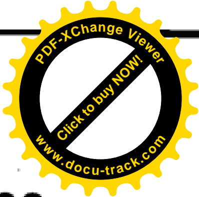
元数据: 关于数据的数据 200

存档、压缩、映像和恢复 202

使用tarfile模块创建TAR归档 203

使用tarfile模块检查TAR文件内容 205

www.TopSage.com

第7章 SNMP 208

引言 208

对SNMP的简要介绍 208

IPython与Net-SNMP 211

查找数据中心 214

使用Net-SNMP获取多个值 217

创建混合的SNMP工具 222

Net-SNMP扩展 224

SNMP设备控制 227

整合Zenoss的企业级SNMP 228

第8章 操作系统什锦 229

引言 229

Python中跨平台的UNIX编辑 230

PyInotify 240

OS X 241

Red Hat Linux系统管理 246

Ubuntu管理 246

Solaris系统管理 247

虚拟化 247

云计算 248

使用Zenoss从Linux上管理Windows服务器 255

第9章 包管理 258

引言 258

Setuptools和Python Egg 259

使用easy_install 259

easy_install的高级特征 261

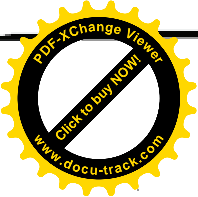
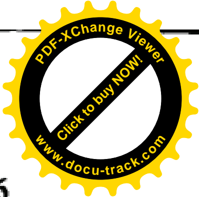
创建egg 267

进入点及控制台脚本 271

使用Python包索引注册一个包 272

Distutils 274

www.TopSage.com

Buildout 276
使用Buildout 277
使用Buildout进行开发 280
virtualenv 280
EPM包管理 285
EPM总结：真的非常简单 289

第10章 进程与并发 290

引言 290
子进程 290
使用Supervisor来管理进程 299
使用Screen来管理进程 301
Python中的线程 302
进程 313
Processing模块 314
调度Python进程 317
daemonizer 318
本章小结 321

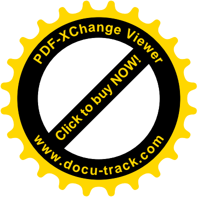
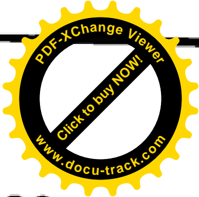
第11章 创建GUI 322

GUI创建理论 322
生成一个简单的PyGTK应用 323
使用PyGTK创建Apache日志浏览器 325
使用Curses创建Apache日志浏览器 329
Web应用 332
Django 333
本章小结 351

第12章 数据持久性 353

简单序列化 353
关系序列化 372
本章小结 381

www.TopSage.com

第13章 命令行 382

引言 382

基本标准输入的使用 383

Optparse简介 384

简单的Optparse使用模式 385

Unix Mashups：整合Shell命令到Python命令行工具中 392

整合配置文件 397

本章小结 399

第14章 实例 400

使用Python管理DNS 400

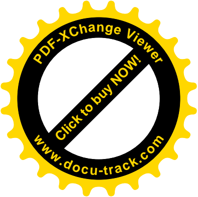
使用OpenLDAP、Active Directory以及其他Python工具实现LDAP 402

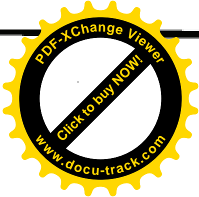
Apache日志报告 404

FTP镜像 410

附录 回调 415

www.TopSage.com





序

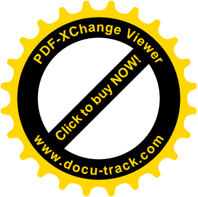
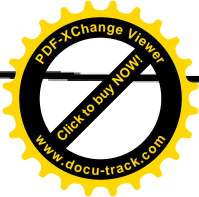
在预先阅读了这本有关使用Python进行系统管理的书后，我非常兴奋。至今，我仍然记得当我在使用其他语言编写了多年程序之后终于发现并使用Python时的感觉：它就像冬日过后一缕清新的空气，一束温暖的阳光。使用Python，让编写代码突然变得简单而有趣，让我可以更快地完成程序。

做为一名系统管理员，我更多地在系统和网络管理任务中使用Python。我深知，一本专注于讲解如何使用Python进行系统管理的书对于系统管理员而言是多么实用。我非常高兴地说，本书正是这样一本难能可贵的书。总地来说，Noah和Jeremy针对在系统管理领域使用Python完成了一系列充满趣味和富有才智的工作。我发现这本书非常有用，阅读起来令人愉快。

对于首次使用Python的系统管理员（也包括其他读者），本书的最初两章中关于Python的介绍非常不错。我想我应该算是一名Python中级程序员了，而我同样从这本书中学到了许多知识。我特别推荐网络和管理网络服务、SNMP以及异构系统管理等几章，因为这几章尤其有用，真正解决了一些系统管理员每天都会遇到的、非常普通的实际任务。

—— Æleen Frisch，2008年7月

www.TopSage.com

前言

本书的体例

本书使用如下的排印约定：

斜体（*Itailc*）
  表示新的条目，网址、电子邮件地址、文件名和文件扩展名。

等宽字体（`Constant width`）
  用于程序列表，文本中涉及的程序元素，如变量或函数名、数据库、数据类型、环境变量、语句、应用、关键词以及模块。

等宽粗体（**`Consrant width bold`**）
  显示命令或其他应该由用户逐个输入的文本。

等宽斜体（*`Consrant width italic`*）
  显示应该由用户提供的值或上下文确定的值进行替换的文本。

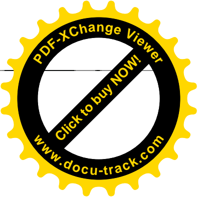
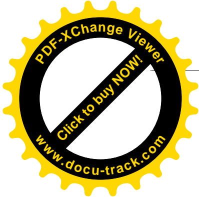
注意： 表示一个技巧、建议或一般性说明。

警告： 表示一个警告或注意。

使用代码示例

这本书能够帮助你更好地完成工作。通常，你或许会在程序或文档中使用本书中的代码。这不需要联系我们以获得批准，除非你正再次重写代码中的关键部分。例如，在编写程序中你使用了本书中的多段代码，不需要经我们许可，但通过CD-ROM销售或发布的程序，如果来自O'Reilly出版的书，则确实是需要获得许可的。引用这本书或引证书

www.TopSage.com

中的示例来回答问题不需要许可，但将这本书中的大量示例代码合并到你的产品文档中则需要许可。

引用时不必指明出处，但我们会对标明出处表示感谢。一个引用出处说明通常包括标题、作者、出版商和ISBN，例如 “*Python for UNIX and Linux System Administration* by Noah Gift and Jeremy M. Jones. Copyright 2008 Noah Gift and Jeremy M. Jones, 978-0-596-51582-9.”

如果你对代码示例的使用超出了合理范围，或超出了上述许可范围，随时联系我们：*permissions@oreilly.com*。

如何联系我们

关于本书的批评建议和相关问题请使用如下地址与出版社联系：

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

我们为本书提供了一个网页，在此页面中我们列出了勘误表、示例和其他的信息。读者可以登录以下网址访问该页面：

http://www.oreilly.com/catalog/9780596515829（英文版）
http://www.oreilly.com.cn/book.php?bn=978-7-111-26663-1（中文版）

我们为这本书设立了网页，在其中列出了勘误表、示例和其他的信息。网页地址是：

http://www.oreilly.com/

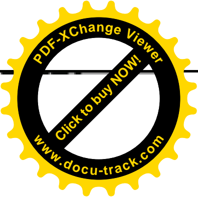
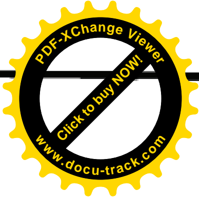
如果你要对这本书发表评论或回答技术问题，请把邮件发到以下地址：

bookquestions@oreilly.com

关于我们出版的书，会议、资源中心以及O'Reilly Network的更多内容，可以查看我们的网站：

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

www.TopSage.com

致谢

来自Noah的致谢

当坐下来写这本书的致谢时，我必须首先感谢Joseph E. Bogen博士，因为他是这段时间里对我影响最大的人。当我在加州理工学院工作时遇见了Bogen博士，他开拓了我的视野，在生活上，以及在心理学、神经科学、数学、意识科学研究等多方面给予我许多建议。他是我遇到的最为聪慧的人，是我敬仰的对象。我正计划找时间写一本关于这些经历的书。由于他已辞逝而不能坐下来读读我写的书，令我感到非常遗憾。

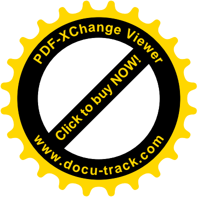
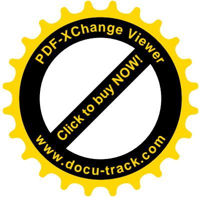
我需要感谢我的妻子Leah，娶她为妻是我曾经感到最美好的事情之一。没有她的爱和支持，我根本无法完成这本书。她有圣女一样的耐心。我期待着我们能一起走过今后的旅程，我爱你Leah。我也十分感谢我的孩子Liam能够耐心地等待我写这本书，他的确十分有耐心。为了写书，我不得不缩短孩子的吉它、钢琴、体能课程，因此我亏欠了孩子很多很多。

我爱您，我的妈妈，感谢您一直以来对我的鼓励。当然我也要感谢我的合作者Jeremy M. Jones，感谢他愿意与我共同完成这本书。我想我们是具有差异的团队，但是也是风格互补的团队，我们在一起写了一本非常有意义的书。你教了我许多有关Python的知识，是一个好搭档、好朋友，感谢你！

Titus Brown，我想现在必须称他为Brown博士了。当我在加州理工学院遇见他时，是他让我对Python产生了浓厚的兴趣。他是另一个可以很好地诠释“人活着应该有所作为”的榜样，我很高兴可以把他视为老朋友，这是金钱难买的。他一直问我“为什么你不使用Python？”，于是，有一天我就尝试着使用了Python。如果不是Titus，现在我肯定还在继续使用Java和Perl。你可以阅读他的博客：*http://ivory.idyll.org/blog*。

Shannon Behrens有金子一样的心，思维敏锐，并且对Python有着惊人的见解。我第一次遇见Shannon是通过Titus，但是他和我很快成为了朋友。Shannon是一个文字能力很强的人，而且教了我许多Python知识。他在Python方面以及对书的编辑方面给了我太多的帮助。如果没有他的帮助，我都难以想象将是什么样子。我不能想象一个公司会愚蠢地让他离职，我希望帮助他完成他的第一本书。最后要说的是，他也是一个非常难得的技术评审专家，你可以在这里阅读到他的博客：*http://jjinux.blogspot.com/*。

Doug Hellmann是我们另一个星级技术评审专家，他效率很高而且对我很有帮助。Jeremy和我非常幸运请到他来审阅此书。这已经超出了他的职责。他的工作非常高效，当我们一起在Racemi工作时，他是促进我们工作的巨大动力源泉。你可以在这里阅览他的博客：*http://blog.doughellmann.com/*。

感谢Scott Leerseen对本书的评阅，并给出了好的建议。他对代码的审阅让我受益匪浅。

感谢Alfredo Deza为这本书创建了Ubuntu虚拟机，他的经验非常值得称赞。

非常感谢Liza Daly为此书的草稿提出了非常好的反馈意见。这对我们非常有帮助。

特别感谢Rush提出的建议以及Buildout、Eggs和Virtualenv提供的参考资料。

感谢Aaron Hillegass，他给了我一些很棒的建议和帮助。他有一个Big Nerd Ranch的测试公司，他是一个专业人员，我很幸运遇到他。感谢Mark Lutz，他很高兴地完成了Python的测试过程，并写了关于Python的一些很有名的书。

感谢亚特兰大Python社区以及PytAtl（*http://pyatl.org*）的成员，他们教会了我许多东西。他们是Rick Copeland、Rick Thomas、Brandon Rhodes、Derek Richardson、Jonathan La Cour、a.k.a Mr. Metaclass、DrewSmathers、Cary Hull、Bernard Matthews、Michael Langford，还有许多人我无法一一提及。Brandon和Rick Copeland是非常可敬的Python程序员，他们给了我尤其多的帮助。你可以阅读Brandon的博客：*http://rhodesmill.org/brandon/*。

感谢Grig Gheorghiu给了我们专业的系统管理的测试建议，感谢他们在我们需要的时候所给予的帮助。

感谢我的前任老板Racemi，以及创建人兼首席技术官Charles Watt。我从他们那里学到了许多。

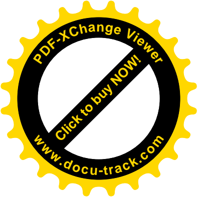
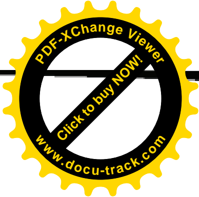
感谢Nada Ganesan博士，他是一个CSULA研究院的资深导师。教会了我许多信息技术方面的知识，并鼓励我有所作为。

感谢Cindy Heiss博士，他是我读营养科学本科时的教授。他让我注意Web的发展，鼓励我要自信，并最终影响我的人生，感谢您！

感谢Sheldon Blockburger，他让我在加州进行尝试，尽管我没有创建团队，他向我展示了如何成为一名更积极的竞争者和战士，教会我自我约束。我相信每周勤奋的工作能够使自己成为一名更好的软件工程师。

在这条路上有许许多多的人都曾帮助过我，包括Jennifer Davis，一个来自加州工学院的朋友，他给了我一些重要的反馈；我在Turner的一些朋友和同事，Doug Wake、Wayne Blanchard、Sam Allgood、Don Voravong；在Disney Feature Animation工作期间的朋友和同事，包括Sean Someroff、Greg Neagle和Bobby Lea。尤其是Greg Neagle，他还教了我许多OS X的知识。也感谢J.F. Panisset，他是我在Sony Imageworks遇到的朋友，教了我许多通用的工程知识。他现在是一名技术总监，可以说是任何公司都稀缺的人才。

www.TopSage.com

我还要感谢其他一些人，他们也作出了重要的贡献：Mike Wagner、Chris McDowell和Shaun Smoot。

感谢Bruce J. Bell，我与他在加州理工学院一同工作。他连续多年教我UNIX和编程，我欠他太多了。你可以在这里看到他的相关资料：*http://www.ugcs.caltech.edu/~bruce/*。

也感谢我在Sony Imageworks的老板Alberto Valez，他是我曾经遇到的最好的老板，给我宽松的工作环境。感谢电影编辑Ed Fuller，他对这本书提出了许多建议，并且在这个过程中我们成为了好朋友。

感谢在Python社区中的所有人。首先，感谢Guido van Rossum，他编写了一种伟大的语言，是一个伟大的领导，当我询问他对这本书的建议时非常有耐心。在Python社区中有许多耀眼的明星，他们编写了非常有用的工具，这些工具我每天都在使用。他们是Ian Bicking、Fernando Perez、Villi Vainio、Mike Bayer、Gustavo Niemeyer等。感谢你们！感谢David Beazely编写的书，感谢他在PyCon 2008的精彩教学。感谢其他Python和系统管理方面书籍的作者。可以通过下面的链接看到他们的工作：*http://wiki.python.org/moin/systems_administration*。也感谢Repoze全体人员：Tres Seaver和Chris McDonough（*http://repoze.org/index.html*）。

对于setuptools一节特别感谢Phillip J. Eby的丰常棒的工具、建议以及对我的包容。也感谢Jim Fulton，他包容了我许多关于ZODB和buildout的问题。另外感谢Martijn Fassen，他教我学习ZODB和Grok。如果你希望看到Python在Web开发的未来，参阅Grok：*http://grok.zope.org/*。

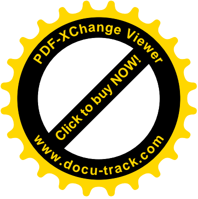
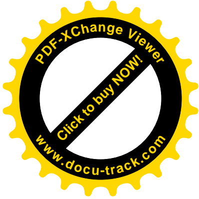
感谢《Red Hat Magazine》的员工：Julie Bryce、Jessica Gerber、Bascha Harris和Ruth Suehle，他们让我尝试以文章的形式表达我的思想。也要感谢IBM Developerworks的Mike McCrary。

我还要感谢那些在我人生中曾告诉我我干不了某件事的人。几乎在每一步，都会遇到令我气馁的人，从告诉我无法进入我想去的大学，到告诉我根本学不会编程。感谢这些人给我的额外的动力，让我成就了梦想。

如果能够真正地相信自己，人们可以开创自己的现实生活。我会鼓励每一个人，给自己实现人生梦想的机会。

最后，感谢O Reilly和Tatiana Apandi，感谢他们对我的这本有关Python和系统管理书的信任。希望你能把握机会阅读此书，相信我和Jeremy，我对此表示感谢。另外，Tatiana在这本书完成的时候离开了O'Reilly，去追逐梦想，她给我留下了深刻的印象。我也要感谢我们的新编辑Julie Steele，她给予了我们有力的支持和帮助。我极为欣赏她沉稳的个性。我希望将来能听到Julie的好消息，与她一起工作让人干劲十足。

www.TopSage.com

来自Jeremy的致谢

在读了Noah的致谢名单之后，我感到有得有失。糟糕的方面是我的名单没有Noah的这么长，而好的方面是，Noah的名单已经涵盖了几乎所有我想要感谢的人。

首先，我必须感谢上帝，是他让我成就了所有的事情，没有他我将一无所成。

最先要感谢的人是我的爱人Debra。在我写这本书时，她让孩子们做别的活动而不来打扰我。她一次次地向孩子们重复规定：“在爸爸写书的时候，不要烦他”。当我需要帮助时，她鼓励我，给我许多空间，这是我最需要的。感谢你，我的爱人。没有你，我无法完成此书。

我也要感谢我的孩子Zane和Justus。在我写书的过程中，他们能够如此耐心。我错过了许多与他们去Stone Mountain旅行的机会。虽然大多数晚上我仍把他们抱上床，但是我却不能像从前那样在床前哄他们安然入睡。我错过了几周前那个星期三晚上的Kid s Rock。我错过了许多，但是你们都没有抱怨过，感谢你们对我的耐心。当你们听到我马上要完成这本书时，是如此兴奋，这让我很感动。我爱你们。

我需要感谢我的父母，Charles和Lynda Jones。在我写这本书时，他们一直支持我。但是我更要感谢的是他们为我树立了榜样，让我有了强烈的工作观念，学会了努力争取，努力工作来改变生活和经济条件。我希望Zane和Justus也能够继承这些好的传统。

感谢Noah Gift，我的合著者。完成这本书所遇到的困难超出了我的想象，可以说是我一生中曾经做过的最为艰辛的事情之一。感谢Noah，我的朋友。如果没有他，这本书根本就不会开始。

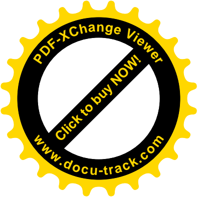
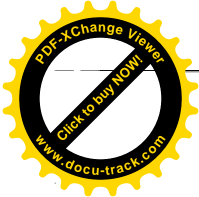
我还要感谢评审组。我想Noah已经感谢过你们所有人，但是我仍要再次感谢：Doug Hellman、Jennifer Davis、Shannon JJ Behrens、Chris McDowell、Titus Brown以及 Scott Leerseen。你们是可敬的。好多次，我认为已经做得恰到好处了，但你们再次调整了我的想法，或者从全新的视角审视这本书，帮助我能够从另外的角度进行思考（其中给予指导最多的是Jennifer。系统管理中的文字处理一章，要尤其感谢你）。感谢你们所有人。

感谢我们的编辑，Tatiana Apandi和Julie Steele。你们处理了其他事务，消减了我们的负担，让我们能够专心完成这本书。感谢你们。

感谢Fernando Perez和Ville Vainio的积极反馈。

感谢Duncan McGreggor帮助我完成了Twisted代码。他的注释非常有帮助。感谢他在Twisted所做的工作。这是一个非常好的框架，我希望更多地使用它。

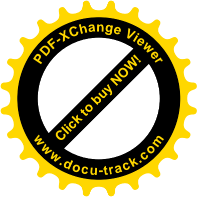
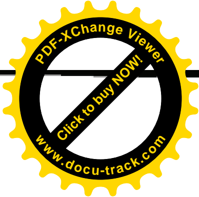
www.TopSage.com

感谢Bram Moolenaar和所有对Vim编辑器做出贡献的人。几乎每一个我写的单词和XML标签都是通过Vim写的。

我也要感谢Linus Torvalds、Debian的开发人员、Ubuntu的开发人员，以及所有工作在Linux平台的人。我键入的每一个单词都是在Linux下完成的。你们将工作过程中构建新的环境，测试不同的事务变得非常简便。感谢你们。

最后要感谢的绝不是分量最轻的人，我要感谢Guido van Rossum和每一个曾经使用过Python的朋友。我从你们的工作中获得了许多经验。我前两次被雇用都是因为熟悉Python。我大约从2001年到2002年开始使用Python，Pyhon语言和Python社区给我带来了许多快乐。感谢你们。Python对我来说一直是非常棒的工具。

第1章

Python简介

为什么要选Python

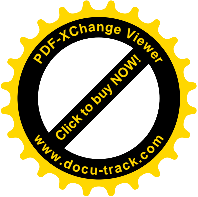
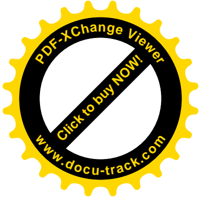
如果你是一位系统管理员，那么可能已对Perl、Bash、ksh或其他一些脚本语言有所了解，甚至已经使用了其中的一种或几种。我们通常利用脚本语言来完成一些重复、乏味的工作，使用后，工作完成的速度和准确性都远远高于不使用它们时的情况。实际上，所有的语言都是工具，它们为完成工作提供了便捷的手段，而这些工具的价值也正是体现在它们能够帮助人们把工作做得更好。我们有理由相信Python是一个非常有价值的工具，它会帮助你更为高效地完成工作。

选择Python是因为它比Perl、Bash、Ruby或其他语言更好么？事实上，我们很难对各种编程语言的优劣进行排序，因为这些工具与使用它们的程序员的思维习惯有着紧密联系。编程是一种主观性很强的活动，与程序员直接相关。一门优秀的编程语言，必须适合于使用它的人。因此，在这里我们不会争论Python是否比其他语言更好，但我们将给出理由说明为什么Python是个不错的选择，也将说明为什么Python尤其适合完成系统管理任务。

Python之所以是一门优秀的编程语言，第一个原因就在于Python十分容易学习。如果一门语言无法在很短的时间内帮助人们迅速提高工作效率，那么它作为语言的魅力也就消失了。想一想在使用一种程序语言去完成一些工作之前，需要花费几周甚至几个月的时间去学习这门语言，值得吗？答案恐怕是否定的，对于系统管理员尤其如此。如果你是一位系统管理员，就会感觉到总有处理不完的工作，花费大量时间学习一门语言，简直难以想象。而使用Python则不同，在几个小时内，你就可以写出规范的脚本程序，完全不用花上几天或是几周时间。的确如此，对于一门语言，如果人们不能很容易就掌握它，并很快地学会编写该语言的代码，那么就应该问一问是否有学习这门语言的必要了。

然而，如果一门语言过于简单，不具备完成复杂工作的能力，它同样也没有太高的价

www.TopSage.com

值。因此，Python之所以被称为一门非常优秀的编程语言，第二个原因就在于Python既简单易学，又可以完成你能够想象到的任何复杂任务。当需要一行一行地读取某个日志文件，然后输出其中一些基本信息时，Python可以胜任。或是当需要解析一个日志文件，提取出其中的信息，将其中的IP地址与过去三个月中每个日志文件里的IP地址相比较（这些日志文件都存储在关系数据库中），然后将比较的结果存储在关系数据库中时，Python同样也可以处理。实际上，Python正用于处理一些非常复杂的问题，如基因序列分析、多线程Web服务和繁重的统计分析任务。也许你并不会遇到上述那样复杂的工作任务，但是当我们需要去完成一些比较复杂的任务时，有Python这门语言相助终归是一件好事情。

此外，虽然你能够使用一门语言编写一些复杂的代码，但这些代码的维护却让你伤透脑筋，这同样不是一件好事。尽管Python也有代码维护问题，但是Python能够帮助我们用简单的语法结构来实现复杂的编程思想。在编写代码过程中，代码简化是一个极为重要的因素，因为简化的代码能够使后续的代码维护工作变得简单、轻松。Python在代码简化方面做得非常出色，即使我们需要维护几个月都没再看过的那些代码，也并非难事。甚至当我们使用之前从未见过的陌生Python代码时，也同样简单方便。由于其语法和通用词汇十分简洁、精练，Python语言很容易让人一经学习就能够长时间使用。

Python之所以是一门非常优秀的编程语言的另一个原因是其可读性非常好。Python使用空格（whitespace）来分隔代码块的开始和结尾。代码的缩进格式帮助人们很容易理解程序的流程。此外，Python可以说是“基于词”（word-based）的语言，也就意味着，Python语言在使用一些特殊字符的同时，特征经常是以关键词或词库的形式实现的。强调词而不是强调特殊字符，使得Python代码具有很好的可读性，并且十分易于理解。

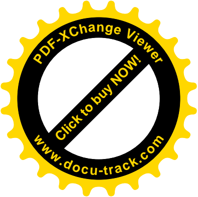
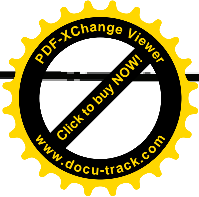
以上，我们总结了Python的一些优点，接下来，我们将分别对Python、Perl和Bash编写的代码示例进行对比分析。通过对比，我们将可以更清楚地看到Python的优势。以下是一个简单的Bash示例程序，显示数字1、2与字母a、b的所有组合，代码如下所示：

```
#!/bin/bash

for a in 1 2; do
    for b in a b; do
        echo "$a $b"
    done
done
```

对比下面使用Perl完成的代码：

```
#!/usr/bin/perl

foreach $a ('1', '2') {
    foreach $b ('a', 'b') {
        print "$a $b\n";
```


www.TopSage.com

```
    }
}
```

我们可以看到，这是一个非常简单的嵌套循环。以下是使用Python中的for循环完成同样循环嵌套机制的代码：

```
#!/usr/bin/env python

for a in [1, 2]:
    for b in ['a', 'b']:
        print a, b
```

接下来，我们再对Bash、Perl和Python中条件语句的使用进行对比。以下代码是Bash中一个简单的if/else条件结构，用于检测指定的文件路径是否为一个目录。

```
#!/bin/bash

if [ -d "/tmp" ] ; then
    echo "/tmp is a directory"
else
    echo "/tmp is not a directory"
fi
```

下面是使用Perl编写的具有相同功能的代码：

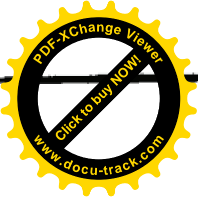
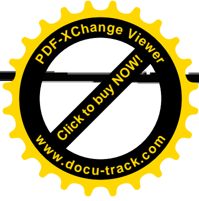
```
#!/usr/bin/perl

if (-d "/tmp") {
    print "/tmp is a directory\n";
}
else {
    print "/tmp is not a directory\n";
}
```

最后是使用Python编写的代码，实现相同的检测功能：

```
#!/usr/bin/env python

import os

if os.path.isdir("/tmp"):
    print "/tmp is a directory"
else:
    print "/tmp is not a directory"
```

Python语言的另一个优势是它对面向对象编程（OOP）的支持。事实上，如果你不想使用OOP来编写程序，完全可以不用，而如果你想使用OOP，就会发现，在Python中使用OOP编程是相当容易的。利用OOP技术，可以简单、清晰地将问题进行分解，将一系列功能模块组成独立的“事物”或“对象”。Bash不支持面向对象技术，Perl和Python则完全支持。下面的代码是采用Perl语言定义的一个类：

www.TopSage.com

```
package Server;
use strict;

sub new {
    my $class = shift;
    my $self = {};
    $self->{IP} = shift;
    $self->{HOSTNAME} = shift;
    bless($self);
    return $self;
}

sub set_ip {
    my $self = shift;
    $self->{IP} = shift;
    return $self->{IP};
}

sub set_hostname {
    my $self = shift;
    $self->{HOSTNAME} = shift;
    return $self->{HOSTNAME};
}

sub ping {
    my $self = shift;
    my $external_ip = shift;
    my $self_ip = $self->{IP};
    my $self_host = $self->{HOSTNAME};
    print "Pinging $external_ip from $self_ip ($self_host)\n";
    return 0;
}

1;
```

在下面的Perl代码中，使用了上述定义的类：

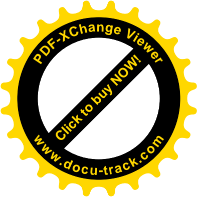
```
#!/usr/bin/perl

use Server;

$server = Server->new('192.168.1.15', 'grumbly');
$server->ping('192.168.1.20');
```

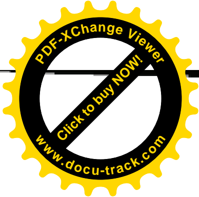
上述代码直接使用了OO模块，看起来十分简单。但如果对Perl如何使用OOP技术进行编程不太熟悉，那么你仍需要花费一些工夫来弄清Perl是如何处理OOP问题的。

接下来，我们看看在Python中定义并使用类的代码：

```
#!/usr/bin/env python

class Server(object):
```

www.TopSage.com





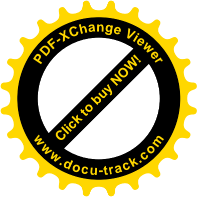
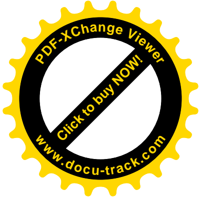
```
    def __init__(self, ip, hostname):
        self.ip = ip
        self.hostname = hostname
    def set_ip(self, ip):
        self.ip = ip
    def set_hostname(self, hostname):
        self.hostname = hostname
    def ping(self, ip_addr):
        print "Pinging %s from %s (%s)" % (ip_addr, self.ip, self.hostname)

if __name__ == '__main__':
    server = Server('192.168.1.20', 'bumbly')
    server.ping('192.168.1.15')
```

在上述Perl与Python的示例中，分别演示了两种语言实现面向对象设计的基本方法。这两个示例程序说明，这两种语言在实现各自的目标时具有不一样的优势。它们可以完成同样的工作，但完成的方式各不相同。因此，如果你希望使用OOP，那么Python正好提供了对OOP的支持，可以简单方便地将OOP技术应用到你所编写的程序中。

Python的另外一个优势并不是来自语言本身，而是来自Python社区。在Python社区中，你可以找到许多完成各类工作的成功经验，还可以学到许多能够选择使用（或是不应使用的）的语言特性。Python语言自身支持的一些句式（phrasing）虽然能够实现某些功能，但在社区中已形成的共识会帮助你远离一些不好的句式。例如，在Python模块的顶端使用“from module import *”是有效的，但是在社区中并不赞同这种写法，而是建议使用“:import module”或“:from module import resource”。因为，将一个模块的全部内容导入到另一个模块的命名空间，有时会引起很多麻烦。例如当你试图了解模块是如何工作的、调用了什么函数和从哪里获得这些函数时，问题就会变得十分复杂。社区中这些特殊的约定会帮助你写出更为清晰的代码，有助于其他人对你的代码进行维护。在编写代码时遵循这些公共约定，会使你在编程过程中少走许多弯路。应该说这是一件非常值得重视的事情。

Python标准库（Python Standard Library）是Python另一个非常值得称道的方面。你可能曾经听说过有人在形容Python时，说它是“连电池都包括在内的”（“batteries included”）。之所以这样评价Python，正是因为Python标准库可以帮助你完成几乎所有的工作，无须再到别的地方去寻找支持模块。Python标准库并不是直接内置（built-in）在语言中的，但并不影响其提供各种各样丰富的功能支持，例如正则表达式功能、套接字、线程、日期/时间功能，XML解析、配置文件解析，文件和目录功能、数据持久性、单元测试功能，还包括http、ftp、smpt和nntp客户端库文件等。因此，一旦Python安装完毕，支持所有这些功能的模块就可以根据需要加载到脚本中。你将获得以上列出的所有功能，而这些功能都是来自Python自身的，无须再从其他地方获取。所有这些功能将在使用Python完成工作的过程中，为你提供有力的帮助。

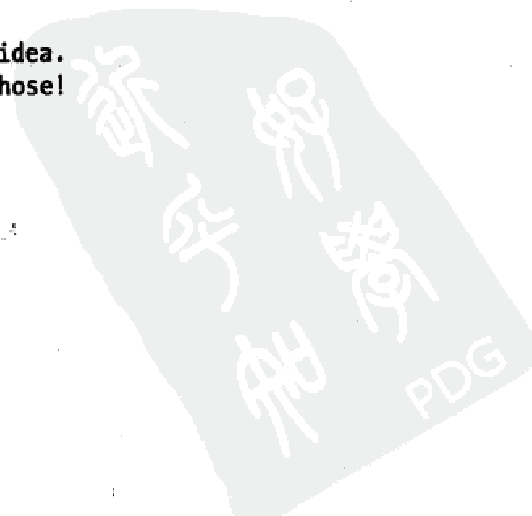
www.TopSage.com

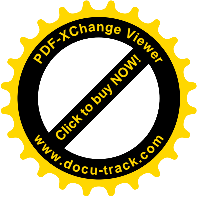
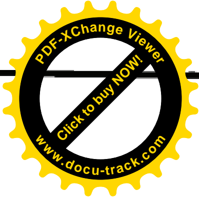
Python另一个真正的优势在于能够方便地获取各种各样的第三方软件包。除了Python标准库中提供的大量库文件外，在互联网上也有大量的库和实用工具可以使用，通过简单的Shell命令行就可以进行安装。在Python包索引（Python Package Index，PyPI，*http://pypi.python.org*）上，任何人都可以将编写的Python包上传到该网站，分享给其他人使用。在我们写这本书的时候，已经有超过3800个包文件可供下载使用。其中包括将在接下来的几章中讲述的IPython， Storm（一个对象-关系映射器，将在第12章中讲述）和Twisted（一个网络框架，将在第5章中讲述），这仅仅是3800多个包中的其中三个。一旦开始使用PyPI，你会发现它是查找和安装有用软件包几乎不可或缺的途径。

我们所看到Python的许多优点，都是源于Python的中心哲学。当我们在Python提示符下输入“import this”时，将会看到Tim Peters所写的“The Zen of Python”，内容如下：

```
In [1]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

译文如下：
Python的禅宗，作者Tim Peters
美丽优于丑陋。
清楚优于含糊。
简单优于复杂。
复杂优于繁琐。
平坦优于曲折。
宽松优于密集。
重要的是可读性。
特殊的案例不足以特殊到破坏规则。
尽管实践可以打破真理。
错误却不可置之不理。
除非另有明确要求。
面对模棱两可，拒绝猜测。
总会有一个 —— 最好是只有一个 —— 显而易见的方式来明辨。
```



www.TopSage.com

PDF-XChange Viewer Click to buy NOW! www.docu-track.com

哪怕这种方式在开始的时候可能并不明显 —— 除非你是荷兰人（译注1）。
现在有比没有好。
尽管没有经常好于现在。
如果如何实现很难被解释清楚，那么这个想法就是一个坏想法。
如果如何实现可以被很好的解释，那么这是一个好想法。
命名空间就是一个非常好的想法 —— 让我们在这方面多做些工作吧！

尽管这些描述并不是在各个层次的语言开发中都能教条式地强制贯彻的，但其思想精髓却可以渗透到语言的方方面面。我们发现这些思想非常有用，这或许也是我们日复一日选择使用Python的理由，而这些思想中的哲学，也会在我们使用语言的过程中不断显现。如果你也能体会到这些思想精髓，Python对你来说或许就是一个非常不错的选择。

学习的动力

如果你刚从书店拿起这本书，或是正在网上在线阅读本书的简介，你或许会问自己学习Python会有多难，甚至会问Python是否值得一学。目前，尽管Python发展得如火如荼，已经吸引了相当一部分人的眼球，但许多系统管理员正在使用的仍是Bash或Perl。如果你就是这些已经掌握了Bash或Perl的系统管理员中的一分子，那么在得知Python非常好学易懂时，也就完全没有必要再担心Python难学了。事实上，Python被许多人认为是最容易讲授和学习的一门语言，尽管这只是来自个人的体会，但的确就是这样！

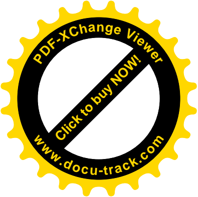
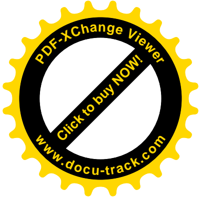
如果你已经对Python有所了解，或者是使用另一种编程语言的大师级人物，那么你可以跳过下面的简介，直接阅读以后各章，通过使用我们的示例，立即开始工作。经过努力，我们编写了一些示例程序，也许可以帮助你完成一些工作。这些示例程序包括：如何使用SNMP自动发现和监测子网，如何转换到交互式Python的Shell（即IPython），如何构建数据处理管道，如何使用对象-关系映射器编写自定义元数据管理工具，如何实现网络编程，如何编写命令行工具等等。

如果你有Shell编程或脚本编程背景，那么学习Python时就一点不用担心学不会了。你可以相当轻松地完成Python的学习，需要的仅是学习的动力、好奇心和毅力，这些因素也会促成你拿起这本书，开始阅读简介部分。

你可能还会有一些顾虑，也许会受到一些与编程相关的负面消息的影响。例如，一个常见的巨大误解就是，仅有一小部分思维奇特且属于精英类型的人才适合学习编程。而事实的真相是，任何人都可以学习编程。另一个同样巨大的误解是，只有攻读并获得计算机专业学位才是一个人成长为真正的软件工程师的必经之路。实际上，一些非常有成就的软件开发人员并没有获得任何工程方面的学位。他们中的许多人获得的是哲学、新闻

译注1： 荷兰人被认为具有非常外向和直接的性格。

www.TopSage.com

学、营养学和英语专业的学位，却成了非常优秀的Python程序员。因此，具有计算机专业的学位并不是学习Python的必要条件，当然，有计算机专业的学位也并不坏。

另外一个有趣的、也是经常被误解的事情是认为学习编程必须从十几岁开始，否则就永远错过了学习的时机。这一误解使得一些人倍感欣慰，因为他们有幸在人生中遇见了鼓励他们从小就开始学习编程的人。然而这简直是完全没有事实根据的观点，尽管在年青时就开始学习编程对以后进一步学习确实是有帮助，但是，年龄并不是学习Python的必要条件。学习Python确实不仅仅是年青人的事，正如我们所听到的，实际上有着数不清的在年近30岁、年近40岁、年近50岁，甚至在更大龄段的人也同样能够成功学习编程的案例。

至此，如果你已经有了学习Python的动力，那么可以说，作为读者，你已经具备了许多人没有的优势。如果你已经决定拿起这本书并开始学习Python，那么你最想知道的可能是如何通过Shell执行命令。掌握了如何在控制终端执行命令之后，本章对Python做简要介绍的任务也就完成了。如果你已经非常确信需要学习Python编程了，那么可以立即开始阅读下面的章节。如你还不是很确信，那么请重新阅读本节，它会让你相信自己有能力学会Python编程。Python编程确实很简单，如果你决定学习Python，它将会改变你的生活。

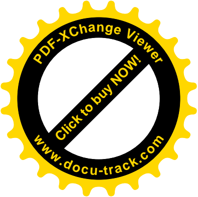
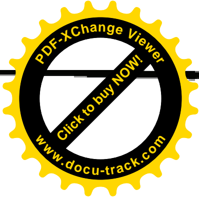
一些基础知识

接下来我们将学习一些基础知识，我们将要对Python所做的简要介绍，与之前我们所能看到的其他介绍不同，我们将同时使用被称为IPython的交互式Shell和常规的Bash shell。你需要打开两个终端窗口，一个是IPython，另一个是Bash。在每一个示例中，我们都将会对Python和Bash程序进行对比。首先需要根据你的平台，下载并安装IPython的正确版本。下载的地址为*http://ipython.scipy.org/moin/Download*。如果由于一些原因，无法下载并完成安装，你也可以使用普通的Python Shell。你可以下载一个包括本书所需软件的虚拟机，虚拟机中包括了一个预先配置并可以直接使用的IPython。只需输入“ipython”，就会看到一个命令提示符。

一旦安装了IPython，并且出现了IPython shell提示符，将会看到下面的内容：

```
[ngift@Macintosh-7][H:10679][J:0]# ipython
Python 2.5.1 (r251:54863, Jan 17 2008, 19:35:17)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.
```


www.TopSage.com

```
In [1]:
```

IPython shell与普通的Bash shell有些相似，可以执行ls、cd、pwd这样的命令。通过阅读接下来的章节，你会学习到更多的命令。本章主要讲述如何学习Python，因此不会对命令做更多讲解。

如果在Python终端上输入下面的内容，可以看到如下结果：

```
In [1]: print "I can program in Python"
I can program in Python
```

如果使用的是Bash终端，输入下面的内容，可以看到如下结果：

```
[ngift@Macintosh-7][H:10688][J:0]# echo "I can program in Bash"
I can program in Bash
```

在这两个示例中，看不出Python与Bash有什么太大的区别，而这正是Python的神奇之处。

在Python中执行命令

如果一天中会花费很多时间在终端里输入各种命令，那么你可能需要学会执行一些语句，例如将处理结果重定向，输出到文件或输出给另一个UNIX命令。接下来我们进行一些比较，从而了解一些Bash下执行的命令在Python中是如何执行的。在Bash终端中，键入如下内容：

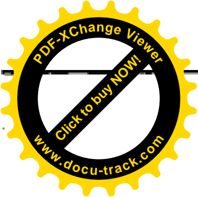
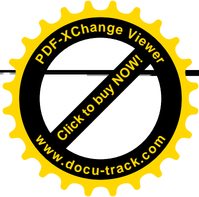
```
[ngift@Macintosh-7][H:10701][J:0]# ls -l /tmp/
total 0
-rw-r--r-- 1 ngift wheel 0 Apr 7 00:26 file.txt
```

而在Python终端中，需要键入的内容如下：

```
In [2]: import subprocess

In [3]: subprocess.call(["ls","-l ","/tmp/"])
total 0
-rw-r--r-- 1 ngift wheel 0 Apr 7 00:26 file.txt
Out[3]: 0
```

在上述Bash示例中，可以很清晰地看到，仅使用了一个非常简单的ls命令。但对于Python示例则不同，如果之前从没有见过Python代码，Python示例看起来可能会令人觉得有些怪异。你可能会想问“真见鬼，import subprocess究竟是什么意思”？Python之所以强大，原因之一就是Python可以载入其他模块，或包含其他文件，并在新的程序中做到代码复用。如果你比较熟悉Bash中的“sourcing”一个文件，就会发现相似之

www.TopSage.com

处。在上例所示的特定情况下，你只需要知道加载了模块subprocess，并且知道使用该模块的语法。我们之后会具体解释subprocess和import是如何工作的，现在直接复制下面的代码即可：

```
subprocess.call(["some_command", "some_argument", "another_argument_or_path"])
```

在Python中，可以就像在Bash中一样使用shell命令。给一点提示：你可以创建Python版本的ls命令。在另一个终端或另一个终端窗口中打开你经常使用的文本编辑器，将上述代码写入到该文件中，且将该文件命名为*pyls.py*，最后使用命令“chmod +x pyls.py”将该文件修改为可执行文件。参见例1-1。

例1-1：Python包装ls命令

```
#!/usr/bin/env python
#Python wrapper for the ls command

import subprocess

subprocess.call(["ls","-l"])
```

如果现在运行该脚本，将获得与在命令行使用ls -l命令完全相同的结果，如下所示：

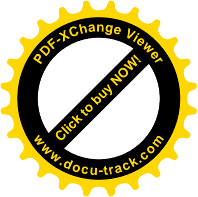
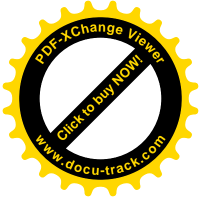
```
[ngift@Macintosh-7][H:10746][J:0]# ./pyls.py
total 8
-rwxr-xr-x 1 ngift staff 115 Apr 7 12:57 pyls.py
```

虽然这个示例看起来十分简单（事实上也确实简单），却可以给出一个在系统编程中使用Python的通用思路。我们经常需要使用Python对脚本或Unix命令进行“包装”（wrap）。实际上，如果在文件中一行接一行地写下命令，然后运行该文件，就可以说你已经开始在写一些基本的脚本了。接下来让我们看一个简单示例。编写例1-2所示的代码，或是直接将以下代码进行剪切和粘贴，然后运行脚本文件*pysysinfo.py*和*bashsysinfo.sh*。这些脚本文件都可以在本章的源代码中找到。例1-2和1-3如下所示：

例1-2：显示系统信息脚本——Python

```
#!/usr/bin/env python
#A System Information Gathering Script
import subprocess

#Command 1
uname = "uname"
uname_arg = "-a"
print "Gathering system information with %s command:\n" % uname
subprocess.call([uname, uname_arg])

#Command 2
diskspace = "df"
diskspace_arg = "-h"
print "Gathering diskspace information %s command:\n" % diskspace
subprocess.call([diskspace, diskspace_arg])
```


www.TopSage.com

例1-3：显示系统信息脚本——Bash

```
#!/usr/bin/env bash
#A System Information Gathering Script

#Command 1
UNAME="uname -a"
printf "Gathering system information with the $UNAME command: \n\n"
$UNAME

#Command 2
DISKSPACE="df -h"
printf "Gathering diskspace information with the $DISKSPACE command: \n\n"
$DISKSPACE
```

如果把这两个脚本都读一遍，会发现它们看起来非常相似。如果分别运行这两个脚本，所看到的输出结果也是完全相同的。需要注意的是，在使用subprocess.call时，将命令与参数完全分开的写法并不是必需的，也可以像下面这样写：

```
subprocess.call("df -h", shell=True)
```

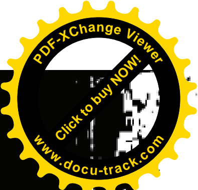
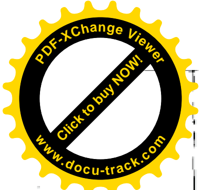
到目前为止，我们已经学习了很多，但仍没能将import和subprocess完全解释清楚。在Python脚本程序中我们加载了subprocess模块，subprocess模块已经包含了使用Python实现系统调用的代码。

正如之前提到的，加载模块（例如加载subprocess）仅仅是将可以使用的代码文件加载进来，也可以创建自己的模块或文件，供以后重复使用，这与加载subprocess模块的方式相同。模块并没什么神奇，只不过是一个写有代码的文件罢了。IPython shell的一个非常好的优点就是可以对模块或文件进行检查，查看其内部可用的属性。对于Unix，这非常类似于在*/usr/bin*目录下运行ls命令。如果你恰好刚开始使用Ubuntu或是Solaris系统，而已经用惯了Red Hat系统，那么你可能会在*/usr/bin*下使用ls命令去查看wget、curl或是lynx命令是否可用。现在如果你想使用在*/usr/bin*目录下找到的某个工具，只要简单地输入命令名称即可，例如**/usr/bin/wget**。

像subprocess这样的模块是非常简单的。使用IPython，你可以通过tab自动完成功能来查看模块中是否有可用的工具。让我们使用tab自动完成功能来查看subprocess中所有可用的属性。注意，一个模块仅仅是包含了代码的文件。下面是在IPython中，使用tab自动完成功能对subprocess进行查询的示例。

```
In [12]: subprocess.
subprocess.CalledProcessError    subprocess.__hash__          subprocess.call
subprocess.MAXFD                 subprocess.__init__          subprocess.check_call
subprocess.PIPE                  subprocess.__name__          subprocess.errno
subprocess.Popen                 subprocess.__new__           subprocess.fcntl
subprocess.STDOUT                subprocess.__reduce__        subprocess.list2cmdline
subprocess.__all__               subprocess.__reduce_ex__     subprocess.mswindows
```

www.TopSage.com

```
subprocess.__builtins__        subprocess.__repr__         subprocess.os
subprocess.__class__           subprocess.__setattr__      subprocess.pickle
subprocess.__delattr__         subprocess.__str__          subprocess.select
subprocess.__dict__            subprocess._active          subprocess.sys
subprocess.__doc__             subprocess._cleanup         subprocess.traceback
subprocess.__file__            subprocess._demo_posix      subprocess.types
subprocess.__getattribute__    subprocess._demo_windows
```

如果需要执行相同的命令，只需要输入：

```
import subprocess
```

然后输入：

```
subprocess.
```

接下来按Tab键使用其自动完成功能来查看可用属性。在示例中的第三列出现了subprocess.call。现在如果想查看更多如何使用subprocess.call的信息，输入如下内容：

```
In [13]: subprocess.call?

Type:           function
Base Class:     <type 'function'>
String Form:    <function call at 0x561370>
Namespace:      Interactive
File:           /System/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/
        subprocess.py
Definition: subprocess.call(*popenargs, **kwargs)
Docstring:
    Run command with arguments. Wait for command to complete, then
    return the returncode attribute.

    The arguments are the same as for the Popen constructor. Example:
    retcode = call(["ls", "-l"])
```

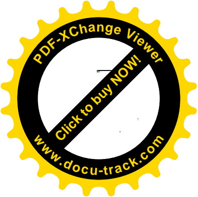
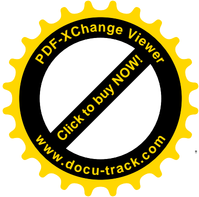
上例中，在属性的后面使用问号来调用manpage手册页查询相关信息。如果想查询UNIX下某个工具如何使用，也可以直接简单地输入：

```
man name_of_tool
```

如果希望查询模块中的其他属性，也可以使用与上例中查询subprocess.call类似的方法。在IPython中，在一个属性之后输入问号，就会找到该属性的相关信息，而且包含该属性的文档也会被打印出来。如果在标准库中使用该方法查询更多的属性，你会找到正确使用这些属性的十分有用的帮助信息。记往，也可以使用Python标准库文档来完成查询。

查看文档时，"Docstring"是正式的官方词汇，我们通过示例演示了如何查询subprocess.call，以及subprocess.call的功能描述。

www.TopSage.com

小结

现在你已经掌握了足够的知识，可以称自己是一名Python程序员了。你知道了如何编写一个简单的Python脚本，如何将Bash脚本翻译成Python脚本并执行它，最后，你知道了如何查询不熟悉的模块或属性的文档，从而获得帮助。在下面一节中，你将看到如何将一系列连续的命令组织到函数中。

在Python中使用函数

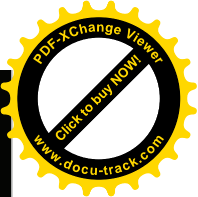
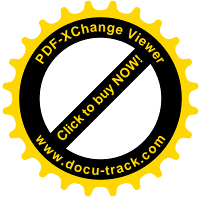
在前面一节中，我们学会了一条接一条地连续执行多个命令，这一点非常有用，因为这意味着可以让一些以前必须手工完成的事情变得可以自动完成。进一步来讲，如果要自动执行这些代码，就需要创建函数。如果你对Bash或其他语言中的函数还不太了解，那么可把函数想象成一个小脚本。函数允许创建一个代码块，代码块中的每一行代码属于该函数，并且在被调用时，代码块中的所有代码是被一起调用的。这与我们写的封装了两行命令的Bash脚本有点相似。Bash脚本与函数的不同之处在于，你可以包含许多函数脚本。最后你可以在脚本中编写多个函数，每个函数包含一组代码，那么这组代码就会在适合的时间被调用执行。

现在我们需要进一步讨论一下空格（whitespace）的问题。在Python中，嵌套代码是通过统一的缩进格式来维护的。在其他的语言中，例如Bash，当定义了一个函数，需要使用括号将函数代码包括起来。在Python中，必须在括号内缩进你的代码。如果你是一个初学者，这可能会造成一点点混乱，但过不了很久，你就会熟悉这种写法，而且还会意识到这对于提高代码的可读性多么有帮助。如果在使用这些交互式示例程序过程中遇到麻烦，仔细查看一下这些代码是否正确地缩进了。最普通的经验就是使用tab来实现缩进，一个tab可以缩进4个空格。

让我们看看在Python和Bash中都是如何处理的。如果还有一个打开的IPython shell，则无须再创建一个Python脚本文件，当然如果你愿意的话，也可以创建一个。在IPython提示符下，交互地输入下面的内容：

```
In [1]: def pyfunc():
...: print "Hello function"
...:
...:

In [2]: pyfunc
Out[2]: <function pyfunc at 0x2d5070>

In [3]: pyfunc()
Hello function

In [4]: for i in range(5):
...: pyfunc()
```

www.TopSage.com

```
...:
...:
Hello function
Hello function
Hello function
Hello function
Hello function
```

在这个示例中，可以看到在函数中放了一个打印语句。之后不仅调用了该函数，而且根据需要还进行了多次调用。在第[4]行中，我们使用了一个编程习惯或技巧，实现了对函数的5次执行。如果之前没看到过这种用法，只需要明白它让函数执行了5次即可。

在Bash shell中也可以完成类似的工作。下面是其中的一种实现方式：

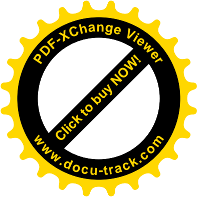
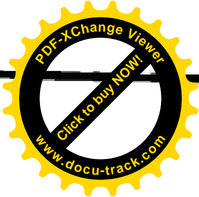
```
bash-3.2$ function shfunc()
> {
>     printf "Hello function\n"
> }
bash-3.2$ for (( i=0 ; i < 5 ; i++))
> do
>     shfunc
> done
Hello function
Hello function
Hello function
Hello function
Hello function
```

在这个Bash示例中，正如之前在Python代码示例中所做的一样，我们创建了一个简单的函数shfunc，然后调用它连续执行了5次，值得注意的是，与Python相比，Bash示例更为繁琐。注意一下Bash的for循环与Python的for循环之间的差异。如果是第一次接触Bash或IPython函数，你应该在IPthon窗口中多做一些函数的练习，然后再开始之后的学习。

函数没有什么神奇的，如果是第一次接触函数，练习交互式地编写多个函数是理解函数，去掉其神秘面纱最好的办法。下面是一些函数的简单示例：

```
In [1]: def print_many():
...:     print "Hello function"
...:     print "Hi again function"
...: print "Sick of me yet"
...:
...:

In [2]: print_many()
Hello function
Hi again function
Sick of me yet

In [3]: def addition():
...: sum = 1+1
```

www.TopSage.com

```
    ...: print "1 + 1 = %s" % sum
    ...:
    ...:

    In [4]: addition()
    1 + 1 = 2
```

除了上面的几个示例外，这里还有一些简单示例，可以在你的机器上一起运行试试。现在，我们回到显示系统信息的脚本，并将其转变成函数。见例1-4。

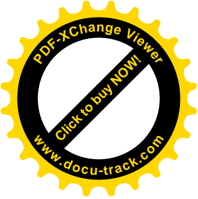
例1-4：转换Python显示系统信息脚本：pysysinfo_func.py

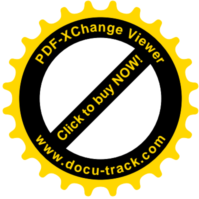
```
#!/usr/bin/env python
#A System Information Gathering Script
import subprocess

#Command 1
def uname_func():

    uname = "uname"
    uname_arg = "-a"
    print "Gathering system information with %s command:\n" % uname
    subprocess.call([uname, uname_arg])

#Command 2
def disk_func():

    diskspace = "df"
    diskspace_arg = "-h"
    print "Gathering diskspace information %s command:\n" % diskspace
    subprocess.call([diskspace, diskspace_arg])

#Main function that call other functions
def main():
    uname_func()
    disk_func()

main()
```

鉴于我们对函数已有的经验，这个转换示例将我们之前脚本中的代码简单地放到函数中，然后使用main函数进行一次性调用。如果不太熟悉这种风格，你可能不知道，这种在脚本中定义多个函数然后由一个main函数进行调用的方法十分普遍。主要原因之一就是如果你想在其他程序中再次使用这个脚本，可以独立地调用该函数，也可以使用main方法进行联合调用。关键取决于你加载模块之后的决定。

如果没有流程控制或main函数，所有代码在被加载时就会被立即执行。这对于一次性执行的脚本非常适用，但是如果你计划创建一个可重复使用的工具，则应该创建函数，封装指定的功能操作，然后通过main函数来执行整个脚本。

为了进行对比，我们将之前的Bash显示系统信息脚本也转化为函数，参见例1-5。





例1-5：转换Bash显示系统信息脚本——bashsysinfo_func.sh

```
#!/usr/bin/env bash
#A System Information Gathering Script

#Command 1
function uname_func ()
{
    UNAME="uname -a"
    printf "Gathering system information with the $UNAME command: \n\n"
    $UNAME
}
#Command 2
function disk_func ()
{
    DISKSPACE="df -h"
    printf "Gathering diskspace information with the $DISKSPACE command: \n\n"
    $DISKSPACE
}

function main ()
{
    uname_func
    disk_func
}

Main
```

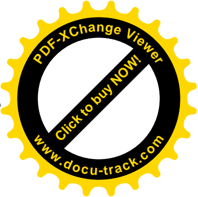
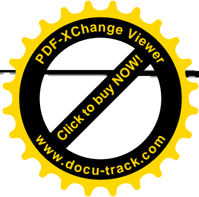
看完Bash示例之后，你可能会觉得它与Python有相当多的相似之处。我们创建了两个函数，然后通过main函数进行调用。如果这是你第一次使用函数，我们强烈建议你将Bash和Python脚本中的main方法注释掉（在Bash或Python脚本的每一行行首加上一个#），然后再运行一次。当再次运行时，你会发现输出结果彻底没有了，这是因为程序在执行时没有调用这两个函数。

学习到现在，你已经是一名能够使用Bash或Python编写简单函数的程序员了。作为程序员，编写程序的过程就是学习的过程。所以，此时我们强烈建议你修改这两个Bash和Python代码中的系统调用程序，将其变成自己的内容。如果可以在脚本中添加一些新的函数，并成功通过main函数实现了调用，那么应该说你干得已经相当不错了，可以好好犒劳一下自己。

通过Import语句实现代码复用

在学习新东西的过程中，总会有这样一个问题，那就是如果它十分抽象，类似微积分，那么我们很难有正当的理由去留意它。恐怕你压根不会记得什么时候曾在便利店中使用了高中时期所学的数学知识。在之前的示例中，我们向你演示了如何在脚本中创建函数，从而取代一行接一行的shell命令。我们也曾提到，模块实际上就是一段脚本，或者

www.TopSage.com

说是文件中的一段代码。这没有什么特别，但是确实需要按特定的方式进行组织，以便于将来在别的程序中进行复用。这也正是为什么你应该特别留意的原因。接下来，我们载入之前的显示系统信息的Bash和Python脚本，并且执行。

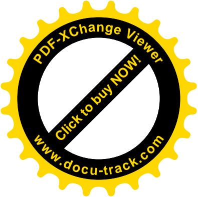
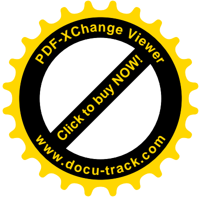
如果你已经关闭了IPython或Bash窗口，那么重新打开，我们会非常快地向你展示为什么函数对于代码复用如此重要。我们采用Python创建的第一个脚本是由一系列命令组成的文件，文件名为*pysysinfo.py*。在Python中，一个模块对应一个文件，反之亦然，我们将脚本文件载入到IPython中。记住你不需要特别指定载入文件的扩展名为*.py*。事实上，如果这样做了，载入反倒不会成功。下面是我们在Noah的Macbook Pro笔记本上所做的操作：

```
In [1]: import pysysinfo
Gathering system information with uname command:

Darwin Macintosh-8.local 9.2.2 Darwin Kernel Version 9.2.2: /
  Tue Mar 4 21:17:34 PST 2008; root:xnu-1228.4.31~1/RELEASE_I386 i386
Gathering diskspace information df command:

Filesystem        Size    Used    Avail   Capacity   Mounted on
/dev/disk0s2      93Gi    88Gi    4.2Gi      96%     /
devfs            110Ki   110Ki     0Bi      100%     /dev
fdesc            1.0Ki   1.0Ki     0Bi      100%     /dev
map -hosts         0Bi     0Bi     0Bi      100%     /net
map auto_home      0Bi     0Bi     0Bi      100%     /home
/dev/disk1s2     298Gi   105Gi   193Gi       36%     /Volumes/Backup
/dev/disk2s3     466Gi   240Gi   225Gi       52%     /Volumes/EditingDrive
```

哇，结果非常棒，是吧？如果你加载了一个完全是Python代码的文件，看起来运行得很好。但是，事实上这里面也存在一些问题。如果你计划执行Python代码，通常是作为脚本或程序的一部分从命令行来执行。使用“import”进行载入操作可以帮助我们实现代码复用。那么问题就来了：如果只想执行一部份脚本，打印输出有关磁盘容量部分代码执行结果，应该怎么做呢？你的回答也许是这不可能。实际并非如此，这个问题完全可以解决，这也正是我们使用函数的原因。如本例所示，函数允许我们对在什么时间执行代码以及执行脚本中哪个部分的代码进行控制，而不是一次将所有代码都执行一遍。不要仅仅听我说，自己也应亲自去试试。只要你加载一个包含多个命令的函数脚本，就会明白我的意思。

下面是IPython终端的输出结果：

```
In [3]: import pysysinfo_func
Gathering system information with uname command:

Darwin Macintosh-8.local 9.2.2 Darwin Kernel Version 9.2.2:
Tue Mar 4 21:17:34 PST 2008; root:xnu-1228.4.31~1/RELEASE_I386 i386
Gathering diskspace information df command:
```

```
Filesystem        Size   Used   Avail  Capacity   Mounted on
/dev/disk0s2      93Gi   88Gi   4.1Gi     96%     /
devfs            110Ki  110Ki    0Bi    100%     /dev
fdesc            1.0Ki  1.0Ki    0Bi    100%     /dev
map -hosts         0Bi    0Bi    0Bi    100%     /net
map auto_home      0Bi    0Bi    0Bi    100%     /home
/dev/disk1s2     298Gi  105Gi  193Gi     36%     /Volumes/Backup
/dev/disk2s3     466Gi  240Gi  225Gi     52%     /Volumes/EditingDrive
```

现在我们使用不包含函数的脚本来获得相同的输出结果。你或许会有些迷惑，这是一个好现象。想知道为什么会获得相同的输出结果，只需要查看一下源码即可。如果你是在主目录下，打开另一个终端标签或窗口，查看脚本pysysinfo_func：

```
#Main function that call other functions
def main():
    uname_func()
    disk_func()

main()
```

问题是之前创建的main函数再次出现。一方面我们希望在命令行运行脚本来获得输出结果，但是另一方面，在加载时，我们不希望一次获得所有的输出，而是希望模块既可以作为一个脚本从命令行直接执行，也可以当成一个可复用的模块使用。这种需求在Python中非常普遍。解决方法就是通过像下面这样修改脚本的最后几行从而改变main函数的调用方式，如下所示：

```
#Main function that call other functions
def main():
    uname_func()
    disk_func()

if __name__ == "__main__":
    main()
```

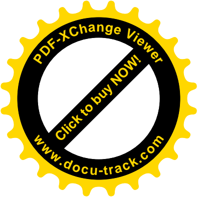
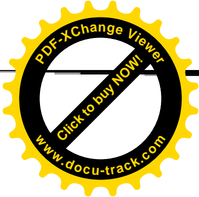
这是一个解决上述问题的比较常用的技巧。任何缩进在这个表达式之后的代码都可以从命令行执行。通过替换你脚本中相应位置的代码，或者载入脚本pysysinfo_func_2.py，可以看到这一应用的效果。

现在，如果回到IPython解释器，并且载入这一新脚本，我们可以看到以下内容：

```
In [1]: import pysysinfo_func_2
```

这时，修改之后的main方法没有被调用。因此，在代码中利用了这一技巧，我们就得到了三个可以在其他程序中使用，或是可以在IPython shell中交互使用的函数。还记得之前我们提到的，仅调用打印磁盘容量的函数，而不用调用执行其他命令的函数的想法吗？首先，我们需要重新回顾一下之前已经介绍的技术。还记得可以使用Tab键的自动完成功能吧，它会显示出所有可用的属所，如下所示：

www.TopSage.com

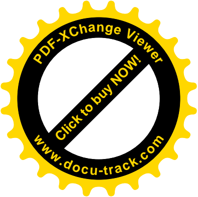
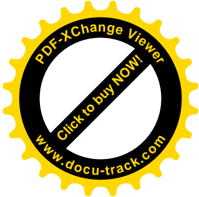
```
In [2]: pysysinfo_func_2.
pysysinfo_func_2.__builtins__       pysysinfo_func_2.disk_func
pysysinfo_func_2.__class__          pysysinfo_func_2.main
pysysinfo_func_2.__delattr__        pysysinfo_func_2.py
pysysinfo_func_2.__dict__           pysysinfo_func_2.pyc
pysysinfo_func_2.__doc__            pysysinfo_func_2.subproces
pysysinfo_func_2.__file__           pysysinfo_func_2.uname_fun
pysysinfo_func_2.__getattribute__
pysysinfo_func_2.__hash__
```

在这个例子中，我们将忽略有双下划线的内容，因为这些是特殊方法，已经超出了本章简介所涉及的内容。由于IPython也是一个常规的shell，所示它识别文件名，以及以*.psc*为扩展名的字节编译的（byte-compiled）Python文件。一旦忽略了这些有双下划线的名字，我们就会看到pysysinfo_func_2.disk_func。让我们继续下一步，调用函数：

```
In [2]: pysysinfo_func_2.disk_func()
Gathering diskspace information df command:

Filesystem          Size    Used    Avail  Capacity  Mounted on
/dev/disk0s2        93Gi    89Gi    4.1Gi     96%    /
devfs              111Ki   111Ki     0Bi    100%    /dev
fdesc              1.0Ki   1.0Ki     0Bi    100%    /dev
map -hosts           0Bi     0Bi     0Bi    100%    /net
map auto_home        0Bi     0Bi     0Bi    100%    /home
/dev/disk1s2       298Gi   105Gi   193Gi     36%    /Volumes/Backup
/dev/disk2s3       466Gi   240Gi   225Gi     52%    /Volumes/EditingDrive
```

或许你现在已经意识到了，函数总是通过被调用，或是在函数名后加“()”来执行的。在这个示例中，仅运行了文件中三个函数中的一个：调用的函数是disk_func。最终实现了代码复用。我们可以载入以前写的代码，并且交互式地运行需要的部分。当然，我们也运行了其他两个函数uname_func和main，这两个函数也是之前分别编写的。让我们看一下：

```
In [3]: pysysinfo_func_2.uname_func()
Gathering system information with uname command:

Darwin Macintosh-8.local 9.2.2 Darwin Kernel Version 9.2.2:
  Tue Mar 4 21:17:34 PST 2008; root:xnu-1228.4.31~1/RELEASE_I386 i386

In [4]: pysysinfo_func_2.main()
Gathering system information with uname command:

Darwin Macintosh-8.local 9.2.2 Darwin Kernel Version 9.2.2:
  Tue Mar 4 21:17:34 PST 2008; root:xnu-1228.4.31~1/RELEASE_I386 i386
Gathering diskspace information df command:

Filesystem          Size    Used   Avail  Capacity  Mounted on
/dev/disk0s2        93Gi    89Gi   4.1Gi     96%    /
devfs              111Ki   111Ki    0Bi    100%    /dev
fdesc              1.0Ki   1.0Ki    0Bi    100%    /dev
map -hosts           0Bi     0Bi    0Bi    100%    /net
```

```
map auto_home       0Bi     0Bi    0Bi    100%    /home
/dev/disk1s2       298Gi   105Gi  193Gi    36%    /Volumes/Backup
/dev/disk2s3       466Gi   240Gi  225Gi    52%    /Volumes/EditingDrive
```

如果仔细观察，你会发现我们同时运行了其他两个函数。记住，main函数会立即执行全部代码。

通常，编写可复用模块就是为了将来在一个新的脚本中可以一遍又一遍地使用该部分代码。例1-6在另一个脚本中展示了如何使用其中一个函数disk_func。

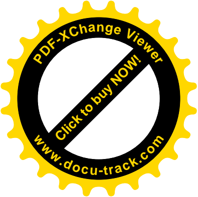
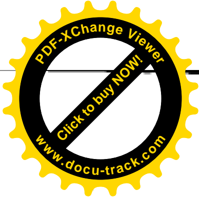
例1-6：使用import进行代码复用：new_pysysinfo

```
#Very short script that reuses pysysinfo_func_2 code
from pysysinfo_func_2 import disk_func
import subprocess

def tmp_space():
    tmp_usage = "du"
    tmp_arg = "-h"
    path = "/tmp"
    print "Space used in /tmp directory"
    subprocess.call([tmp_usage, tmp_arg, path])

def main():
    disk_func()
    tmp_space()

if __name__ == "__main__":
    main()
```

这个例子不仅演示了对之前编写的代码的复用，也展示了如何使用Python特定的语法加载我们需要的函数。代码复用的有趣之处也在于，通过载入之前所编写程序中的函数，可以实现一个完全不同的程序。注意，在main方法中包含其他模块中的函数disk_func()，以及我们创建的新函数。

在这一节，我们学习了代码复用的知识，了解了代码复用的强大之处，同时也看到，代码复用实现起来又是多么简单。简而言之，我们在文件中加入了一个或两个函数，如果还希望能以脚本方式运行，则需要加入特定的if __name__ == "__main__"语法。之后，我们就可以加载这些函数到IPython中，或是简单地在另一个脚本中进行复用。学习上述内容之后，你就可以写一些非常复杂的Python模块，通过多次复用来创建一个新的工具了。这时的你，应该已经成长为一名高手了。

www.TopSage.com

第2章

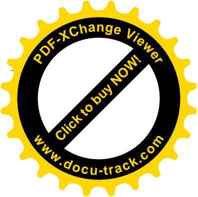
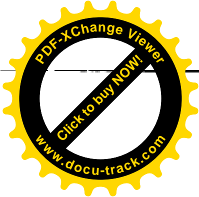
IPython

Python的优点之一是其交互式解释器，也称为shell。shell提供了一种能够快速实现灵感、检验特性的方法，以及交互式的模块界面，能够将一些需要两三行脚本才能完成的任务一次性完成。通常我们编写代码时，会采用同时运行文本编辑器和Python的方式（稍后会有介绍，这实际上运行的就是IPython），通过交互地使用编辑器和shell，也就是在两者之间切换来完成程序的编写。我们经常需要将代码从编辑器复制到shell或从shell复制到编辑器。这种方式使得我们可以即时看到代码在Python中的处理结果，并且可以快速地在文本编辑器中编写需要的代码。

事实上，IPython集成了交互式Python的诸多优点。IPython具有卓越的Python shell，其性能远远优于标准Python的shell。IPython同时提供了基于控制台命令环境的定制功能，可以十分轻松地将交互式Python shell包含在各种Python应用中，甚至可以当作系统级shell来使用。本章主要介绍如何使用IPython提高*nix-shell以及Python相关任务的执行效率。

与Python相同，IPython也有着非常活跃的社区支持。可以在 *http://lists.ipython.scipy.org/mailman/listinfo/ipython-user*注册邮件列表；此外，在*http://ipython.scipy.org/moin*有一个极好的wiki。作为wiki的一部分，在*http://ipython.scipy.org/moin/Cookbook*还有一个菜单列表。因此，可以根据需要进行阅读或向其中添加资源。也可以在IPython的开发领域贡献力量。最近，IPython的开发已经转变为分布式代码控制方式，因此可以将代码分段下载后再进行处理。如果做了一些有益的工作，还可以向其提交你所做的修改。

www.TopSage.com



PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

名人简介：IPython

Fernando Perez

Fernando Perez在获得物理学博士学位之后，在科罗拉多大学的应用数学系从事数值算法研究。目前，他在加州大学伯克利分校Helen Wills神经科学研究所，主要致力于脑成像问题的分析方法和高级科学计算工具方面的研究。在研究生期间，Fernando Perez就参与了Python工具的开发工作，这些工具都被用于科学计算领域。2001年，为了寻找能够更为高效地处理每天科研任务的交互式工作流程，Fernando Perez发起了IPython开源项目。该项目得到了社区众多参与者的关注，支持者队伍日渐壮大。经过数年发展，IPython已经不仅局限于科研领域的应用，而且也让并非从事科研工作的程序员受益匪浅。

名人简介：IPython

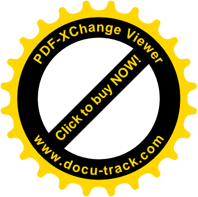
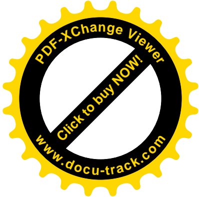
Ville Vainio

2003年，Ville Vainio在芬兰Satakunta应用科学大学（Satakunta University of Applied Sciences）Pori技术学院获得了软件工程学士学位。在撰写本书时，他是Digia Plc公司智能手机部的软件专家，主要在诺基亚和UIQ的Symbian操作系统平台上从事C++程序开发工作。此前，Ville Vainio曾就职于Cimcorp Oy公司，致力于使用Python语言开发工业机器人通信软件。Ville一直热衷于IPython，自2006年1月起，他就一直在维护0.x系列的稳定版本。Ville最初所做的工作是为IPython实现一系列补丁程序，使其具有比Windows系统shell更优越的性能。至今系统shell用例仍然是Ville关注的重点。Ville和未婚妻现在住在芬兰，在Pori的坦佩雷理工大学完成硕士论文。他的论文是关于ILeo项目的，ILeo试图在IPython和Leo之间架设一座桥梁，使Leo能够成为IPython的full-fledged记事本。

安装IPython

安装IPython可以有几种选择，其中最常见也是最常用的方法，是通过IPython发布的源码进行安装。IPython的源码可以在*http://ipython.scipy.org/dist/*下载。编写本书时，IPython的最新发布版本是0.8.2。0.8.3版本也即将完成。安装时需要下载*tar.gz*文件，例如*http://ipython.scipy.org/dist/ipython-0.8.2.tar.gz*。通过`tar zxvf ipython-0.8.2.tar.gz`命令解压软件包后，能够看到一个*setup.py*文件。通过调用带`install`参数的*setup.py*文件

www.TopSage.com



PDF-XChange Viewer Click to buy NOW! www.docu-track.com

（例如，python setup.py install）开始安装Python。该操作将在*site-packages*目录中安装IPython的库文件，并在*scripts*目录中创建一个ipython脚本。在UNIX系统中，该目录与python的二进制文件目录相同。如果系统中已经安装了python包，则IPython将会安装到*/usr/bin*目录下。本书中，我们安装的是IPython最新的开发版源码，因此你可能会在一些例子中看到“0.8.3”。

第二种选择是通过系统的软件包管理器安装IPython软件包。*.deb*安装包可在Debian和Ubuntu获取，运行*apt-get install ipython*命令即可。Ubuntu将IPython的库文件安装到*/usr/share/python-support/ipython*目录下，包括一系列*.pth*文件和符号链接。而IPython的二进制文件则安装在*/usr/bin/ipython*目录下。

第三种选择是通过Python包进行安装。也许你从没有注意到在Python包中包含了IPython。实际上，Python包是一个ZIP文件，解压后包含一个扩展名为*.egg*的文件。Egg文件可以通过easy_install工具安装。easy_install工具的突出特点之一，是能够检查egg文件的配置，从而选择需要安装的内容。大多数时候，easy_install工具被人们忽略了，而事实上，它非常简单易用。easy_install工具通过Python包索引（Python Package Index，简称PyPI，又被称作Python CheeseShop）确定包的安装。使用easy_install工具安装IPython，只需要用户对*site-package*目录具有写权限，直接运行*easy_install ipython*即可。

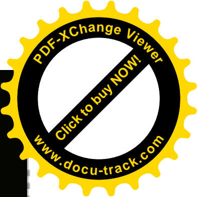
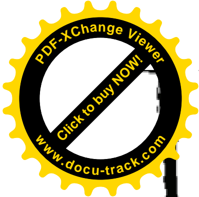
第四种选择可能会令你感到万分惊讶，那就是IPython不必安装即可使用。当下载了IPython发布的源码，并运行了ipython.py安装命令之后，就可以使用该下载版本中的IPython实例了。这种方法能够使*site-packages*目录保持简明，但同时也会带来一些问题，那就是如果没有解压IPython，也没有修改PYTHONPATH环境变量，IPython将不能作为一个库文件直接使用。

基础知识

IPython安装之后，第一次运行ipython命令，将看到如下内容：

```
jmjones@dink:~$ ipython
**********************************************************************
Welcome to IPython. I will try to create a personal configuration directory
where you can customize many aspects of IPython's functionality in:

/home/jmjones/.ipython

Successful installation!

Please read the sections 'Initial Configuration' and 'Quick Tips' in the
IPython manual (there are both HTML and PDF versions supplied with the
distribution) to make sure that your system environment is properly configured
to take advantage of IPython's features.
```

www.TopSage.com

```
Important note: the configuration system has changed! The old system is
still in place, but its setting may be partly overridden by the settings in
"~/.ipython/ipy_user_conf.py" config file. Please take a look at the file
if some of the new settings bother you.


Please press <RETURN> to start IPython.
```

此时光标会停留在原处，等待输入。若点击Return键，IPython将显示如下内容：

```
jmjones@dinkgutsy:stable-dev$ python ipython.py
Python 2.5.1 (r251:54863, Mar 7 2008, 03:39:23)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.3.bzr.r96 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]:
```

与IPython进行交互

当我们第一次接触到一个新的shell提示符，通常有些不知所措，甚至根本就不知道该做些什么。还记得第一次登录到UNIX，看到出现的(ba|k|c|z)sh提示符时的情形吗？既然正在阅读本书，我们假设你已经对Unix shell有一定的了解。如果的确如此，那么对你而言，掌握IPython将变得十分容易。

当你第一次看到IPython提示符，也许同样不知道该做些什么。出现这种情况的原因极有可能是在IPython提示符下，可以做到的事几乎没有任何限制。因此，关键是应该明确想要做些什么。通过IPython提示符，Python语言所有的一切都可以使用。而且，还有许多IPython魔术般神奇的函数可以利用。通过IPython，可以方便地执行任何Unix shell命令，并将执行结果保存到Python变量中。接下来的几个示例，将展示在IPython的默认配置下，可以从IPython中获得什么。

下面是一些简单的输入输出操作：

```
In [1]: a = 1

In [2]: b = 2

In [3]: c = 3
```

这与在标准Python提示符下输入相同的内容看起来没什么不同。我们简单地为a、b、c分别赋值1、2、3。

www.TopSage.com

PDF-XChange Viewer Click to buy NOW! www.docu-track.com

IPython与标准Python的最大区别在于，Ipython会对命令提示符的每一行进行编号。

现在我们已经将一些数值（1、2和3）分别保存到一些变量中（变量a、b和c），可以查看这些变量所保存的数值：

```
In [4]: print a
1

In [5]: print b
2

In [6]: print c
3
```

这是一个设计好的例子。仅需要简单地键入打印输出变量语句（print），就可以直接查看每个变量的赋值，最坏的情况不过是需要向上回滚屏幕。每一个显示的变量会占6个字符，多于显示它们的值所需要的实际长度。下面是显示变量值的另一种更为简洁的方式：

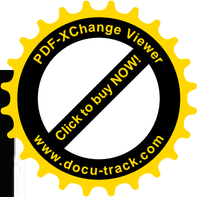
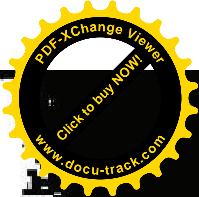
```
In [7]: a
Out[7]: 1

In [8]: b
Out[8]: 2

In [9]: c
Out[9]: 3
```

对比两个例子，其输出变量值似乎相同，其实仍有差别。print语句使用非正式的（unofficial）字符串表达式，而简单变量名（bare variable name）使用了正式的（official）字符串表达式。在处理自定义类而不是内置类时，这种差异会体现得非常明显。下面是使用不同字符串表示方法的示例：

```
In [10]: class DoubleRep(object):
....: def __str__(self):
....: return "Hi, I'm a __str__"
....: def __repr__(self):
....: return "Hi, I'm a __repr__"
....:
....:

In [11]: dr = DoubleRep()

In [12]: print dr
Hi, I'm a __str__

In [13]: dr
Out[13]: Hi, I'm a __repr__
```

我们创建了一个名为DoubleRep的类，类中有两个方法，一个是__str__，另一个是

www.TopSage.com

__repr__，用来演示使用print输出对象与使用正式字符串表达式的差异。在实例化DoubleRep对象后，指定变量dr保存该对象。接下来，使用print输出对象dr的值，可以看到__str__方法被调用。之后，简单地输入变量的名称dr，则__repr__方法被调用。产生其中差异的原因是，当输入变量名时，IPython以正式字符串表达式显示结果，而使用print输出变量时，IPython采用非正式字符串表达式。总之，在Python中，当调用str(obj)或是使用格式化字符串"%s" % obj时，__str__方法被调用，当调用repr(obj)或是使用格式化字符串"%r" % obj时，__repr__方法被调用。

实际上，这并非IPython的特例，标准Python shell也是如此。下面是使用标准Python shell的相同的DoubleRep示例：

```
>>> class DoubleRep(object):
... def __str__(self):
... return "Hi, I'm a __str__"
... def __repr__(self):
... return "Hi, I'm a __repr__"
...
>>>
>>> dr = DoubleRep()
>>> print dr
Hi, I'm a __str__
>>> dr
Hi, I'm a __repr__
```

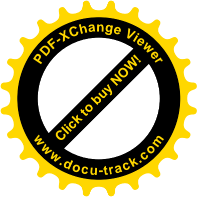
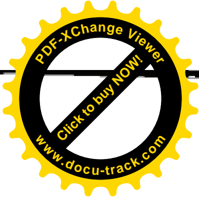
你可能已经注意到了，标准Python的提示符和IPython提示符是不一样的。标准Python的提示符由三个大于号（>>>）组成，而IPython的提示符由单词“In”，之后是方括号中的数字，最后是冒号组成（例如In [1]:）。采用这样的提示符，原因可能是IPython具有命令跟踪功能，所有输入的命令都被保存在一个名为In的列表中。在前一个示例中，当变量a、b、c赋值为1、2、3之后，In列表中的内容如下所示：

```
In [4]: print In
['\n', u'a = 1\n', u'b = 2\n', u'c = 3\n', u'print In\n']
```

IPython的输出提示符与标准Python输出提示符也是不同的。IPython的输出提示符看上去能够区分两种输出：写输出（written output）与求值输出（evaluated output）。而事实上，IPython并非真正能够区分这两种类型。print调用会引起计算的副作用，因此IPython忽略print，不会对其进行捕获。print的副作用会在标准输出stdout中反映出来，这是在调用过程中发送的。而在IPython执行用户代码时，会检测返回值，如果返回值不是空（None），会在提示符“Out [number]:”后将返回值打印输出。

标准Python提示符同样不会区分这两种输出类型。如果在IPython提示符后输入了一个语句来对一些数值进行求值，且值不为空，则IPython将求得的值输出到新的一行中，该行以Out开始，后跟方括号和行号，然后是冒号，最后是表达式求值的结果（例如

www.TopSage.com

Out[1]:1）。下面是一个示例，演示了在IPython中如何将一个整数赋值给变量，然后对变量进行求值运算，最后打印输出变量值。注意这三者之间的差异：为变量赋值、显示对变量求值的结果，以及打印输出变量。首先是IPython的提示符：

```
In [1]: a = 1

In [2]: a
Out[2]: 1

In [3]: print a
1

In [4]:
```

接下来是标准Python的提示符：

```
>>> a = 1
>>> a
1
>>> print a
1
>>>
```

在IPython和Python中，对整数进行赋值的方式没有什么差异。一个是IPython提示符，一个是标准Python提示符，两者都能够快速地向用户返回变量的赋值。但是，在显示正式字符串表示的变量时，IPython和标准Python有所不同。IPython显示一个Out提示符，而Python则直接显示值。对于打印输出变量，两者没有什么区别，都是以无提示符方式显示输出。

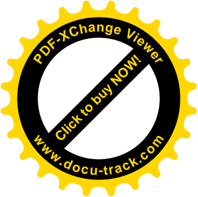
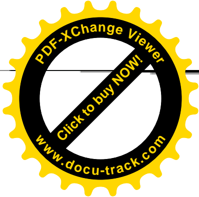
这种In [some number]:和Out [some number]:的方式或许会让人迷惑，是否在IPython与标准的Python之间存在着更深层的差异，还是这种差异纯粹就是表面上的？这种差异毫无疑问是有深层根源的。事实上，这种差异反映了IPython的功能区不同于Python，从而使得IPython这种不同类型的交互式shell与标准Python shell能够相互区分。

在这里，有两个内置变量应该引起注意，它们是In和Out。前者是IPython输入列表（list）对象，后者是一个字典（dict）对象。以下是type对In和Out的说明：

```
In [1]: type(In)
Out[1]: <class 'IPython.iplib.InputList'>

In [2]: type(Out)
Out[2]: <type 'dict'>
```

在开始使用In和Out之后，你会对此习以为常。

接下来，我们看看这两个数据类型保存了什么内容。

```
In [3]: print In
['\n', u'type(In)\n', u'type(Out)\n', u'print In\n']

In [4]: print Out
{1: <class 'IPython.iplib.InputList'>, 2: <type 'dict'>}
```

正如所期望的，In和Out分别保存了输入，以及非空语句和表达式求值运算的输出。由于每一行必须有输入，这对于跟踪类列表结构的输入非常有效。但是，跟踪类列表（list-like）结构的输出却可能导致一些空字段或所包含的内容为空的字段。因此，并不是每一行都会有可求值的非空输出，采用类字典（dictionary-like）的数据结构或是一个纯粹的字典（dict）对象对输出进行跟踪就显得十分有意义了。

Tab自动完成

另外一个极为有用的有关IPython数据输入（data-entry）的特征是Tab自动完成功能，该功能在默认状态下是开启的。标准Python shell如果编译时增加了readline支持特性，将具有tab自动完成功能，但需要做如下处理：

```
>>> import rlcompleter, readline
>>> readline.parse_and_bind('tab: complete')
```

经过上述设置，我们可以使用如下功能：

```
>>> import os
>>> os.lis<TAB>
>>> os.listdir
>>> os.li<TAB><TAB>
os.linesep os.link os.listdir
```

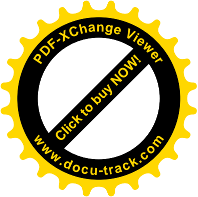
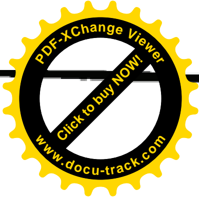
在加载了rlcompleter和readline，并设置了readline的Tab自动完成选项后，可以载入os，输入os.lis，按Tab键一次，让自动完成功能将其匹配成os.listdir。也可以输入los.li，然后按Tab键两次，将会出现一个所有可能匹配的列表。

在IPython中，可以实现相同的功能而无须进行任何额外的配置。对于Python，该项功能是可选的，而对于IPython，该功能则为默认开启的。下面是在IPython中运行与之前相同的示例：

```
In [1]: import os

In [2]: os.lis<TAB>
In [2]: os.listdir
In [2]: os.li<TAB>
os.linesep os.link os.listdir
```

注意，在本示例的最后一行，只需按Tab键一次即可。

www.TopSage.com

这个os.TAB示例仅仅演示了IPython的属性查找和自动完成功能，而另一个更不错的自动完成功能，则体现在模块导入方面。打开一个新的IPython shell，这样可以看到IPython将如何帮助我们找到需要载入的模块：

```
In [1]: import o
opcode       operator     optparse     os       os2emxpath   ossaudiodev

In [1]: import xm
xml       xmllib     xmlrpclib
```

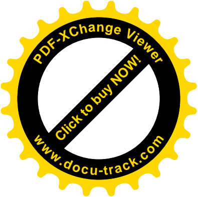
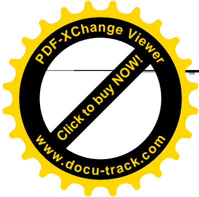
注意：所有通过import自动完成功能所列出的项都是模块，不需要为此感到意外，这就是IPython的特点。

IPython提供了两类自动完成功能：完成（complete）与菜单完成（menu-complete）。两者的差别在于“完成”尽可能扩展当前的主题词，并提供一个可能的替换列表，而“菜单完成”会扩展主题词，直接匹配可以替换列表中的一个，并且如果连续按Tab键时，每一次都会切换到下一个可能的替换。IPython的默认自动完成是“完成”。也可以通过简单的设置，轻松修改IPython的原有配置。

魔力编辑

最后一个将要涉及的基本输入输出主题是关于魔力编辑（magic edit）功能的。（在接下来的章节中将进一步介绍魔力编辑）。严格地说，使用shell这种面向命令行（line-oriented）的用户交互方式，尽管具有相当大的优势，但是也存在一定限制。这句话听起来有些矛盾，接下来我们一点一点分析解释。在shell中，采用一次输入一行命令的方式。每次输入一行命令，接下来shell会对命令进行处理，有时候你会坐下来等待命令执行的返回结果，然后再输入下一条命令。这个过程就是一个循环。事实上，也是非常有效的。但是，有时候如果一次能够处理多行命令，那将是非常不错的选择。为实现这一功能，使用文本编辑器对多行命令进行编辑是不错的做法。虽然IPython的readline支持在这方面有所改进，如可以使用文本编辑器编写Python模块，但这不是我们想在这里要谈论的内容。我们将要讨论的是在面向命令行的输入方式与文本编辑器输入方式之间进行整合，然后向shell提供需要执行的命令。可以说，有了对多行代码处理的支持，严格面向命令行的输入方式就显得功能有些受限了。这就是为什么我们说面向命令行的方式尽管具有相当大的优势，但是也存在一定限制的原因。

魔力编辑功能类似于上面提到的在Python shell的纯命令行交互方式与使用文本编辑器方式之间的折中。其好处是可以利用手边的资源，尽情享用你选择的编辑器的全部优点。可以简便地编辑代码块，对代码块中的循环或是函数的方法进行修改。而且，还具有与shell直接交互所带来的便捷和灵敏。当这两种来编写代码的方法可以整合时，其各自的

优点也被整合了。你可以保留shell环境，在编辑器中暂停、编辑和执行代码。当你继续使用熟悉的shell工作时，可以看到在编辑器中刚刚做的修改所带来的变化。

配置IPython

最后需要学习的基础知识是如何配置IPython。如果第一次运行IPython时没有指定其他信息，它会在home目录下创建一个*.ipython*目录。在*.ipython*目录中的是一个名为*ipy_user_conf.py*的文件。这就是使用Python语法的简单用户配置文件。为了让你可以随心所欲地使用IPython，配置文件中包含了大量的配置项，用户可以自行定制。例如，可以选择所使用shell的颜色，选择用于shell提示的组件，设置用%edit编辑文件时所使用的默认文件编辑器。这里，我们不再进一步描述其中的细节了。你只需要知道一点，那就是IPython有一个配置文件，并且它值得你仔细阅读一下，因为这有利于确定它是否包含了你所需要的配置项或希望设置的配置项。

从功能强大的函数获得帮助

正如已经提到的，IPython有着强大的功能。原因之一是它具有非常多的、内建的（built-in）魔力函数。什么是魔力函数？在IPython的文档中是这样描述的：

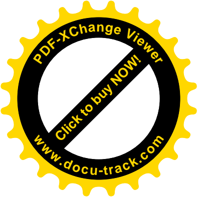
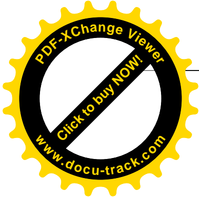
> *IPython会将任何第一个字母为%的行，视为对魔力函数的特殊调用。这样你就可以控制IPython，为其增加许多系统级的特征。魔力函数都是以%为前缀，并且参数中不包含括号或者引号。*
>
> *例如：输入“%cd mydir”（不包括引号），表示如果mydir存在的话，将当前工作目录修改到mydir。*

有两个魔力函数可以帮助你查看所有的函数，并且排序输出那些有用的函数。第一个魔力函数是lsmagic。lsmagic可以列出所有的魔力函数。下面是运行lsmagic的结果示例：

```
In [1]: lsmagic
Available magic functions:
%Exit  %Pprint  %Quit  %alias  %autocall  %autoindent  %automagic  %bg
%bookmark  %cd  %clear  %color_info  %colors  %cpaste  %debug  %dhist  %dirs
%doctest_mode  %ed  %edit  %env  %exit  %hist  %history  %logoff  %logon
%logstart  %logstate  %logstop  %lsmagic  %macro  %magic  %p  %page  %pdb
%pdef  %pdoc  %pfile  %pinfo  %popd  %profile  %prun  %psearch  %psource
%pushd  %pwd  %pycat  %quickref  %quit  %r  %rehash  %rehashx  %rep  %reset
%run  %runlog  %save  %sc  %store  %sx  %system_verbose  %time  %timeit
%unalia s %upgrade  %who  %who_ls  %whos  %xmode

Automagic is ON, % prefix NOT needed for magic functions.
```

正如你所看到的，这里有非常多的函数可供使用。事实上，在写这本书时，已经有69个魔力函数可供使用。也可以像下面这样列出所有的魔力函数：

www.TopSage.com

```
In [2]: %<TAB>
%Exit           %debug          %logstop        %psearch        %save
%Pprint         %dhist          %lsmagic        %psource        %sc
%Quit           %dirs           %macro          %pushd          %store
%alias          %doctest_mode   %magic          %pwd            %sx
%autocall       %ed             %p              %pycat          %system_verbose
%autoindent     %edit           %page           %quickref       %time
%automagic      %env            %pdb            %quit           %timeit
%bg             %exit           %pdef           %r              %unalias
%bookmark       %hist           %pdoc           %rehash         %upgrade
%cd             %history        %pfile          %rehashx        %who
%clear          %logoff         %pinfo          %rep            %who_ls
%color_info     %logon          %popd           %reset          %whos
%colors         %logstart       %profile        %run            %xmode
%cpaste         %logstate       %prun           %runlog
```

输入%然后按Tab键，可以看到69个magic函数的列表。使用lsmagic函数或输入%然后按Tab，都是为了快速查看所有可用的函数，当你正在寻找某一特定的函数时就可以这样做。或者你可以使用上述方法快速浏览所有的函数，看看都有哪些函数可用。但是除非看到具体的函数说明，否则仅靠列表，不足以帮助你了解每个函数的具体功能。

而这正是神奇的魔力函数发挥作用的另一个地方。魔力函数的名字magic本身就具有魔力。运行magic可以打开一个分页的帮助文档，其中记录了所有IPython内建函数的用法。这个帮助文档包括函数名，函数的用法（适用于何处），以及函数工作方式的描述。以下是魔力page函数的帮助说明：

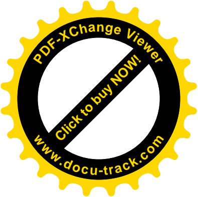
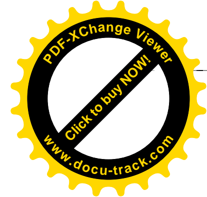
```
%page:
    Pretty print the object and display it through a pager.

    %page [options] OBJECT

    If no object is given, use _ (last output).

    Options:

      -r: page str(object), don't pretty-print it.
```

你可以在执行magic函数之后搜索或前后滚动，找到需要的内容。如果知道需要查找的具体函数，并且想直接跳到该函数所在位置而不是上下滚动进行查找，这么做非常有效。所有的函数按照字母顺序排列，因此无论是搜索还是滚动，都非常方便。

在本章的后面将介绍另外一种使用帮助的方法。在键入希望获得帮助信息的魔力函数名字之后，在其后输入问号（?），能够得到与使用%magic几乎相同的帮助信息。下面是使用%page?之后的结果：

```
In [1]: %page ?
Type:           Magic function
Base Class:     <type 'instancemethod'>
```

www.TopSage.com

```
String Form:    <bound method InteractiveShell.magic_page of
                <IPython.iplib.InteractiveShell object at 0x2ac5429b8a10>>
Namespace:      IPython internal
File:           /home/jmjones/local/python/psa/lib/python2.5/site-packages/IPython/
                Magic.py
Definition:     %page(self, parameter_s='')
Docstring:
    Pretty print the object and display it through a pager.

    %page [options] OBJECT

    If no object is given, use _ (last output).

    Options:

      -r: page str(object), don't pretty-print it.
```

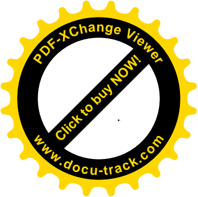
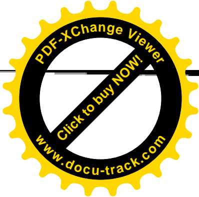
这是IPython帮助文档中的一部分，用它非常适于生成一个总结，也可以用于生成魔力函数自身的总结。在IPython提示符下输入%quickref后，可以看到一个分页的参考文档信息，如下所示：

```
IPython -- An enhanced Interactive Python - Quick Reference Card
================================================================

obj?, obj??      : Get help, or more help for object (also works as
                   ?obj, ??obj).
?foo.*abc*       : List names in 'foo' containing 'abc' in them.
%magic           : Information about IPython's 'magic' % functions.

Magic functions are prefixed by %, and typically take their arguments without
parentheses, quotes or even commas for convenience.

Example magic function calls:

%alias d ls -F   : 'd' is now an alias for 'ls -F'
alias d ls -F    : Works if 'alias' not a python name
alist = %alias   : Get list of aliases to 'alist'
cd /usr/share    : Obvious. cd -<tab> to choose from visited dirs.
%cd??            : See help AND source for magic %cd

System commands:

!cp a.txt b/     : System command escape, calls os.system()
cp a.txt b/      : after %rehashx, most system commands work without !
cp ${f}.txt $bar : Variable expansion in magics and system commands
files = !ls /usr  : Capture sytem command output
files.s, files.l, files.n: "a b c", ['a','b','c'], 'a\nb\nc'
```

其结束部分内容如下所示：

```
%time:
        Time execution of a Python statement or expression.
%timeit:
        Time execution of a Python statement or expression.
```


www.TopSage.com

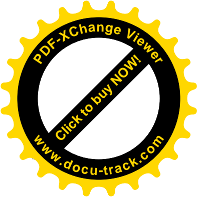
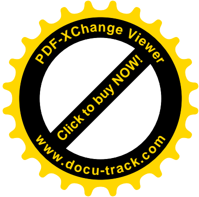



```
%unalias:
        Remove an alias
%upgrade:
        Upgrade your IPython installation
%who:
        Print all interactive variables, with some minimal formatting.
%who_ls:
        Return a sorted list of all interactive variables.
%whos:
        Like %who, but gives some extra information about each variable.
%xmode:
        Switch modes for the exception handlers.
```

%quickref的起始部分是一个对IPython各种用法的引用。%quickref的其余部分是对%magic函数的迷你总结，包括全部帮助信息的首行。例如，下面是一个对%who的全部帮助信息：

```
In [1]: %who ?
Type:           Magic function
Base Class:     <type 'instancemethod'>
String Form:    <bound method InteractiveShell.magic_who of
                <IPython.iplib.InteractiveShell object at 0x2ac9f449da10>>
Namespace:      IPython internal
File:           /home/jmjones/local/python/psa/lib/python2.5/site-packages/IPython/
                Magic.py
Definition:     who(self, parameter_s='')
Docstring:
    Print all interactive variables, with some minimal formatting.

    If any arguments are given, only variables whose type matches one of
    these are printed. For example:

      %who function str

    will only list functions and strings, excluding all other types of
    variables. To find the proper type names, simply use type(var) at a
    command line to see how python prints type names. For example:

      In [1]: type('hello')
      Out[1]: <type 'str'>

    indicates that the type name for strings is 'str'.

    %who always excludes executed names loaded through your configuration
    file and things which are internal to IPython.

    This is deliberate, as typically you may load many modules and the
    purpose of %who is to show you only what you've manually defined.
```

%wquickref中的%ho帮助行同使用标准的%who?返回的Docstring部分的第一行信息是一样的。

UNIX Shell

使用UNIX shell，毫无疑问有它的优点。UNIX shell提供了一个处理问题的统一方法，具有丰富的工具集，相当简炼容易的语法、标准I/O流、管道、以及重定向等功能。如果能够在我们所熟悉的UNIX中增加Python的功能，将会非常有用。IPython就具有一些兼顾两者优点的特征。

alias

起到衔接Python与UNIX shell功能的第一个特征是alias魔力函数。通过ailas，可以创建一个IPython的快速方式，用以执行系统命令。定义别名，只需要简单地输入alias，后跟系统命令（也可以附加该命令的参数）。例如：

```
In [1]: alias nss netstat -lptn

In [2]: nss
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address             State
tcp        0      0 0.0.0.0:80              0.0.0.0:*                   LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*                   LISTEN
```

别名还有一些不同的输入方式，其中之一是do-nothing方法。如果传递给命令的所有附加参数都能够组织在一起，就可以采用do-nothing方法。例如，如果想查找netstat命令中有关80的执行结果，可以这样输入：

```
In [3]: nss | grep 80
(Not all processes could be identified, non-owned process info
will not be shown, you would have to be root to see it all.)
tcp        0      0 0.0.0.0:80              0.0.0.0:*                   LISTEN -
```

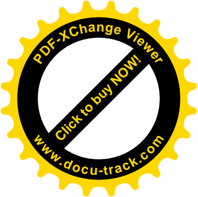
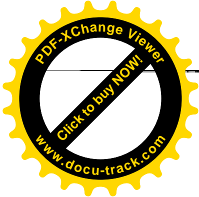
这样做没有传递附加的选项，但也无法显示参数是如何进行处理的。

还有一个do-everything方法。该方法与do-nothing方法相似，但采用的是隐含参数传递的方式。需要显示地操作所有后续参数。下面是一个示例，显示了如何以一个组的方式处理后续参数。

```
In [1]: alias achoo echo "|%l|"

In [2]: achoo
||
In [3]: achoo these are args
|these are args|
```

这里演示了%l（百分号后跟字母l）的语法，该方法用于将行的其他部分插入到alias中。

www.TopSage.com

事实上，使用该方法几乎可以在别名之后（在别名所代表的实现命令的中间位置）插入任何内容。

下面是一个do-nothing示例，重新处理全部参数：

```
In [1]: alias nss netstat -lptn %l

In [2]: nss
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN

In [3]: nss | grep 80
(Not all processes could be identified, non-owned process info
will not be shown, you would have to be root to see it all.)
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
```

在这个示例中，事实上不需要加入%l。即使没有加入，也会得到相同的结果。

还可以通过命令字符串插入不同的参数，这里使用%s表示字符串。下面示例显示参数如何插入：

```
In [1]: alias achoo echo first: "|%s|", second: "|%s|"

In [2]: achoo foo bar
first: |foo|, second: |bar|
```

然而，这存在一点问题。如果仅提供了一个参数，而需要的是两个参数，则会得到错误信息。

```
In [3]: achoo foo
ERROR: Alias <achoo> requires 2 arguments, 1 given.
-------------------------------------------------------------------------
AttributeError                         Traceback (most recent call last)
```

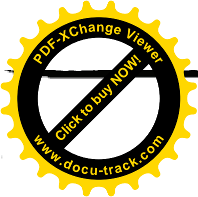
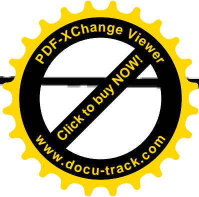
而另一方面，如果提供的参数个数多于需要的个数，那么执行起来结果是正确的。

```
In [4]: achoo foo bar bam
first: |foo|, second: |bar| bam
```

可以看到，foo和bar分别插入到各自的位置，而bam被附加到尾部，这正是它应当被放置的位置。

可以用%store魔力函数保留所使用的别名，本章后面会介绍如何实现。继续前面的例子，保留achoo别名，使得下一次打开IPython时，能够继续使用它，如下所示：

www.TopSage.com

```
In [5]: store achoo
Alias stored: achoo (2, 'echo first: "|%s|", second: "|%s|"')

In [6]:
Do you really want to exit ([y]/n)?
(psa)jmjones@dinkgutsy:code$ ipython -nobanner

In [1]: achoo one two
first: |one|, second: |two|
```

Shell的执行

另一个可以简易执行shell命令的方法，是在命令前加一个感叹号（!）：

```
In [1]: !netstat -lptn
(Not all processes could be identified, non-owned process info
will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
```

通过美元符（$）前缀，可以将变量传递到shell命令中，例如：

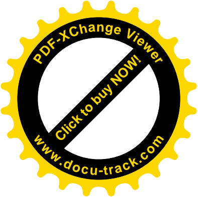
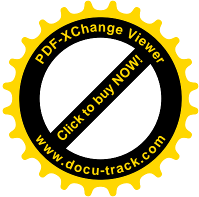
```
In [1]: user = 'jmjones'

In [2]: process = 'bash'

In [3]: !ps aux | grep $user | grep $process
jmjones    5967  0.0  0.4  21368  4344 pts/0    Ss+  Apr11   0:01 bash
jmjones    6008  0.0  0.4  21340  4304 pts/1    Ss   Apr11   0:02 bash
jmjones    8298  0.0  0.4  21296  4280 pts/2    Ss+  Apr11   0:04 bash
jmjones   10184  0.0  0.5  22644  5608 pts/3    Ss+  Apr11   0:01 bash
jmjones   12035  0.0  0.4  21260  4168 pts/15   Ss   Apr15   0:00 bash
jmjones   12943  0.0  0.4  21288  4268 pts/5    Ss   Apr11   0:01 bash
jmjones   15720  0.0  0.4  21360  4268 pts/17   Ss   02:37   0:00 bash
jmjones   18589  0.1  0.4  21356  4260 pts/4    Ss+  07:04   0:00 bash
jmjones   18661  0.0  0.0    320    16 pts/15   R+   07:06   0:00 grep bash
jmjones   27705  0.0  0.4  21384  4312 pts/7    Ss+  Apr12   0:01 bash
jmjones   32010  0.0  0.4  21252  4172 pts/6    Ss+  Apr12   0:00 bash
```

可以看到，上例中列出了所有属于jmjones的bash会话。

下面是一个示例，展示了如何保存使用感叹号执行的命令结果：

```
In [4]: l = !ps aux | grep $user | grep $process

In [5]: l

Out[5]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: jmjones    5967  0.0  0.4  21368  4344 pts/0   Ss+  Apr11   0:01 bash
1: jmjones    6008  0.0  0.4  21340  4304 pts/1   Ss   Apr11   0:02 bash
2: jmjones    8298  0.0  0.4  21296  4280 pts/2   Ss+  Apr11   0:04 bash
```

www.TopSage.com

```
3: jmjones  10184  0.0  0.5  22644  5608 pts/3   Ss+  Apr11   0:01 bash
4: jmjones  12035  0.0  0.4  21260  4168 pts/15  Ss   Apr15   0:00 bash
5: jmjones  12943  0.0  0.4  21288  4268 pts/5   Ss   Apr11   0:01 bash
6: jmjones  15720  0.0  0.4  21360  4268 pts/17  Ss   02:37   0:00 bash
7: jmjones  18589  0.0  0.4  21356  4260 pts/4   Ss+  07:04   0:00 bash
8: jmjones  27705  0.0  0.4  21384  4312 pts/7   Ss+  Apr12   0:01 bash
9: jmjones  32010  0.0  0.4  21252  4172 pts/6   Ss+  Apr12   0:00 bash
```

你或许注意到了，输出结果保存到了变量l中，这与之前示例中的输出不同。变量l包括了一个类列表（list-like）对象，而之前演示的示例中显示的是命令的原始输出。在之后的字符串处理章节中将进一步讨论类列表对象。

!!可以替换!，除了使用!!无法保存结果到变量之外，两者完全一致。可以使用_或_[0-9]*符号访问命令输出的结果，这将在之后的“历史结果”部分进行讨论。

在一个shell命令之前使用!或!!，无疑要比创建一个别名更为便捷。但是在使用过程中还需要视情况而定，一些情况下应该创建别名，而另一些情况下应该使用!或!!。例如，如果需要输入的命令时常用到，那么为其创建一个别名或宏比较好。如是仅仅使用一次，或偶尔用一下，那么最好使用!或!!。

rehash

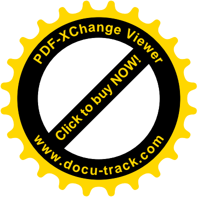
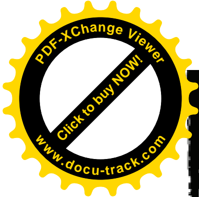
IPython中还有另一个使用别名或是执行shell命令的方法：重哈希（rehashing）。从技术上讲，它是为shell命令创建一个别名，但实际上并非如此。rehash魔力函数更新了PATH路径中的别名表（alias table）。你或许会问“什么是别名表”？当创建一个别名时，IPython映射别名到你希望关联的shell命令。别名表就是映射发生的地方。

注意： 比重哈希别名表更好的方式是使用rehashx魔力函数而不是rehash。我们将同时说明这两种方法，并描述两者的不同。

IPython提供了大量运行IPython时可以访问的变量，例如In和Out，这两个变量我们已经介绍了。IPython还提供了一个变量__IP。__IP实际上是一个交互式shell对象，拥有一个叫做alias_table的属性。这就是实现映射别名到shell命令的地方。可以采用与查看任何变量相同的方式来查看别名映射：

```
In [1]: __IP.alias_table

Out[1]:
{'cat': (0, 'cat'),
'clear': (0, 'clear'),
'cp': (0, 'cp -i'),
'lc': (0, 'ls -F -o --color'),
'ldir': (0, 'ls -F -o --color %l | grep /$'),
'less': (0, 'less'),
```

www.TopSage.com

```
'lf': (0, 'ls -F -o --color %l | grep ^-'),
'lk': (0, 'ls -F -o --color %l | grep ^l'),
'll': (0, 'ls -lF'),
'lrt': (0, 'ls -lart'),
'ls': (0, 'ls -F'),
'lx': (0, 'ls -F -o --color %l | grep ^-..x'),
'mkdir': (0, 'mkdir'),
'mv': (0, 'mv -i'),
'rm': (0, 'rm -i'),
'rmdir': (0, 'rmdir')}
```

这看起来十分像字典：

```
In [2]: type(__IP.alias_table)

Out[2]: <type 'dict'>
```

的确如此。当前，该字典具有16个记录项：

```
In [3]: len(__IP.alias_table)

Out[3]: 16
```

在执行重哈希之后，映射变得更大：

```
In [4]: rehash

In [5]: len(__IP.alias_table)

Out[5]: 2314
```

让我们查找一些之前没有，而现在应该存在的记录项，例如transcode工具应该出现在别名表中：

```
In [6]: __IP.alias_table['transcode']

Out[6]: (0, 'transcode')
```

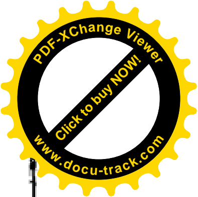
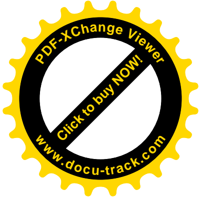
注意： 当看到一个变量或属性以双下划线（__）开头时，这实际上表示代码的作者不希望你对其进行修改。我们可以访问变量__IP，但这只能展示它的内部结构。如果想去访问IPython的正式API，应该使用_ip对象，该对象在IPython提示符下可以被访问。

---

rehashx

rehashx与rehash十分相似，只是rehashx在PATH中进行查找，并认为可以将其添加到别名表里。因此，当打开一个新的IPython shel并执行rehashx时，我们期望别名表与rehash的结果大小相同甚至更小。

www.TopSage.com

```
In [1]: rehashx

In [2]: len(_IP.alias_table)

Out[2]: 2307
```

有趣的是，rehashx产生的别名表内容比rehash的或还要少， 7个。以下是这7个不同之处：

```
In [3]: from sets import Set

In [4]: rehashx_set = Set(_IP.alias_table.keys())

In [5]: rehash

In [6]: rehash_set = Set(_IP.alias_table.keys())

In [7]: rehash_set - rehashx_set

Out[7]: Set(['fusermount', 'rmmod.modutils', 'modprobe.modutils', 'kallsyms', 'ksyms', /
        'lsmod.modutils', 'X11'])
```

如果想看看为什么运行rehashx时，rmmod.modutils没有出现在别名表中，而在运行rehash时却确实出现了，可以执行如下操作：

```
jmjones@dinkgutsy:Music$ slocate rmmod.modutils
/sbin/rmmod.modutils
jmjones@dinkgutsy:Music$ ls -l /sbin/rmmod.modutils
lrwxrwxrwx 1 root root 15 2007-12-07 10:34 /sbin/rmmod.modutils -> insmod.modutils
jmjones@dinkgutsy:Music$ ls -l /sbin/insmod.modutils
ls: /sbin/insmod.modutils: No such file or directory
```

可以看到，rmmod.modutils是insmod.modutils的链接，而insmod.modutils并不存在。

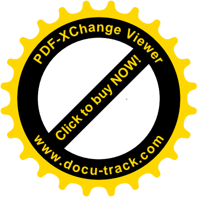
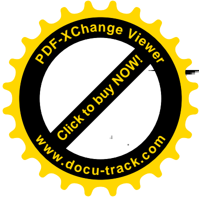
cd

如果使用标准的Python shell，你可能注意到妄想判断当前进入的是哪一个目录非常困难。尽管可以使用os.chdir()来更改目录，但这并不十分方便。也可以使用os.getcwd()来获得当前的目录，但也不是非常方便。如果执行的是Python命令而不是标准Python shell下的shell命令，这或许不是一个大问题。但是，如果正在使用IPython并且需要经常访问系统shell，那么找到一种能够更为方便地访问目录并进行目录浏览的方法，将变得非常重要。

这就是将要介绍的cd。输入cd不是一个创举，做到这一点也毫无困难。但是想象一下，如果错过，问题将十分严重。

在IPython中，cd的作用与Bash中cd的作用相同。主要用法是“cd directory_name”。如果你有Bash的经验，就知道这样用是可以的。如果没有参数，cd命令会让你回到主目

www.TopSage.com

录。如果使用空格加连字符作为参数“cd -”，能够让你回到前一个目录。以下三个附加的选项，是Bash中的cd所不具备的。

第一个是“-q”或quiet选项。不使用该选项，IPython会输出你刚刚改变的目录名。下面是一个示例，演示了使用了这一选项的不同：

```
In [1]: cd /tmp
/tmp

In [2]: pwd

Out[2]: '/tmp'

In [3]: cd -
/home/jmjones

In [4]: cd -q /tmp

In [5]: pwd

Out[5]: '/tmp'
```

使用-q会阻止IPython输出曾经进入过的*/tmp*目录。

另一个IPython的cd命令所包含的选项，能够切换当前目录到已定义标签所在的目录。（之后会解释如何创建一个标签）。以下示例演示了如何改变目录到创建标签的位置：

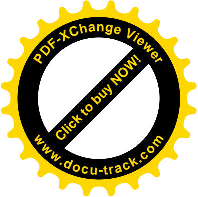
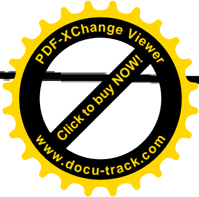
```
In [1]: cd -b t
(bookmark:t) -> /tmp
/tmp
```

示例中，假设对目录*/tmp*设置了标签t。切换到标签所示目录的正式语法是cd -b bookmak_name。但是，如果名为“bookmak_name”的标签已被定义，或是在当前目录下没有bookmark_name目录，则-b标志就是可选的；IPython能够找到你想要进入的用标签标识的目录。

IPython的cd命令所提供的最后一个特征选项，是能够切换当前目录到指定的目录，而指定的目录来自于由曾经访问过的目录组成的历史列表。下面是一个示例，其中就使用了目录历史列表：

```
0: /home/jmjones
1: /home/jmjones/local/Videos
2: /home/jmjones/local/Music
3: /home/jmjones/local/downloads
4: /home/jmjones/local/Pictures
5: /home/jmjones/local/Projects
6: /home/jmjones/local/tmp
7: /tmp
8: /home/jmjones
```

www.TopSage.com

```
In [2]: cd -6
/home/jmjones/local/tmp
```

首先，你会看到在目录历史列表中列出的所有目录。使用该功能可以立即进入到以前访问过的目录。接下来，传递数字参数-6，该参数告诉IPython我们希望进入的目录是目录历史列表中标识为6的目录，即*/home/jmjones/local/tmp*。最后，我们看到当前目录已经成功切换到了*/home/jmjones/local/tmp*。

bookmark

我们已经展示了如何使用cd选项进入到被标签（bookmark）标识的目录。现在介绍如何创建和管理标签。需要格外注意的是，标签在整个IPython会话过程中都是持久有效的。如果退出IPython，之后再次启动，你的标签仍将存在。有两种方式可以创建标签。下面是第一种方式：

```
In [1]: cd /tmp
/tmp

In [2]: bookmark t
```

在*/tmp*目录时，输入“bookmark t”，一个名为t的标签就创建了，且该标签指向*/tmp*目录。另一种创建标签的方法需要输入更多的参数，例如：

```
In [3]: bookmark muzak /home/jmjones/local/Music
```

这里创建了一个名为muzak的标签，该标签指向一个本地存放音乐的目录。第一个参数是标签的名称，第二个参数是标签指向的目录名。

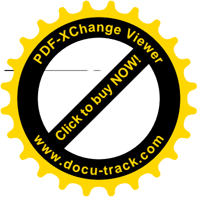
-l选项可以让IPython显示标签列表。我们已经定义了两个标签。现在看一下全部的标签：

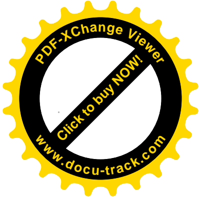
```
In [4]: bookmark -l
Current bookmarks:
muzak -> /home/jmjones/local/Music
t     -> /tmp
```

有两个选项可以删除标签：删除所有标签和一次删除一个标签。在以下示例中，我们创建一个标签，随后删除该标签，最后删除全部的标签，执行过程如下：

```
In [5]: bookmark ulb /usr/local/bin

In [6]: bookmark -l
Current bookmarks:
muzak -> /home/jmjones/local/Music
t -> /tmp
ulb -> /usr/local/bin
```

www.TopSage.com





```
In [7]: bookmark -d ulb

In [8]: bookmark -l
Current bookmarks:
muzak -> /home/jmjones/local/Music
t -> /tmp
```

-l还有一个可替换的选项-b，可以使用“cd -b”：

```
In [9]: cd -b<TAB>
muzak t        txt
```

继续后面的操作：

```
In [9]: bookmark -r

In [10]: bookmark -l
Current bookmarks:
```

可以看到，上述示例中创建的标签名为ulb，且指向*/usr/local/bin*目录。之后，使用选项“-d bookmark_name”删除了该标签。最后使用-r选项删除了全部标签。

dhist

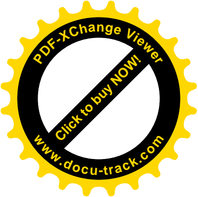
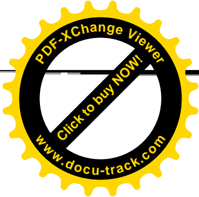
在前面有关cd的示例中，用到了一个由曾经访问过的目录所组成的历史目录列表。现在向你演示如何查看该列表。使用的命令为dhist。该命令不仅可以保存会话列表，而且可以保存IPython会话过程中使用的目录。下面是不带参数使用dhist命令得到的的结果：

```
In [1]: dhist
Directory history (kept in _dh)
0: /home/jmjones
1: /home/jmjones/local/Videos
2: /home/jmjones/local/Music
3: /home/jmjones/local/downloads
4: /home/jmjones/local/Pictures
5: /home/jmjones/local/Projects
6: /home/jmjones/local/tmp
7: /tmp
8: /home/jmjones
9: /home/jmjones/local/tmp
10: /tmp
```

一种访问目录历史的快捷方式是使用cd -<TAB>，如下所示：

```
In [1]: cd -
-00 [/home/jmjones]                      -06 [/home/jmjones/local/tmp]
-01 [/home/jmjones/local/Videos]         -07 [/tmp]
-02 [/home/jmjones/local/Music]          -08 [/home/jmjones]
-03 [/home/jmjones/local/downloads]      -09 [/home/jmjones/local/tmp]
-04 [/home/jmjones/local/Pictures]       -10 [/tmp]
-05 [/home/jmjones/local/Projects]
```


www.TopSage.com

dhist命令有两个选项，可以让使用该命令比cd-<TAB>更为灵活。第一个选项允许提供一个数字来定义显示多少个目录。例如，只想查看最近访问过的5个目录，可以输入如下内容：

```
In [2]: dhist 5
Directory history (kept in _dh)
6: /home/jmjones/local/tmp
7: /tmp
8: /home/jmjones
9: /home/jmjones/local/tmp
10: /tmp
```

第二个选项允许指定一个目录范围。例如，要查看第3个到第6个之间的所有目录，可以输入如下命令：

```
In [3]: dhist 3 7
Directory history (kept in _dh)
3: /home/jmjones/local/downloads
4: /home/jmjones/local/Pictures
5: /home/jmjones/local/Projects
6: /home/jmjones/local/tmp
```

注意，结束边界是非包含的，因此必须在设置结束位置时，将其指定为你想要查看的最后一个目录的下一个目录。

pwd

在目录操作中，一个简单但几乎是必须的函数就是pwd，pwd能够告诉你当前所在的目录。下面是一个示例：

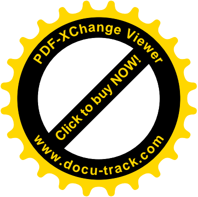
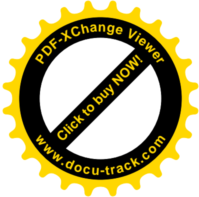
```
In [1]: cd /tmp
/tmp

In [2]: pwd

Out[2]: '/tmp'
```

可变扩展

前面介绍了八个IPython的特征，它们是非常有用的，也是必需的。接下来将要介绍的三个特征会让高级用户感到非常高兴。其中，第一个是可变扩展（Variable Expansion）。到目前为止，我们几乎一直保持着shell是shell，Python是Python。但是现在，我们要跨越边界，将两者进行合并。也就是说，从Python取得一个值，然后把值传递给shell。

```
In [1]: for i in range(10):
   ...:     !date > ${i}.txt
   ...:
```

www.TopSage.com

```
   ...:
In [2]: ls
0.txt 1.txt 2.txt 3.txt 4.txt 5.txt 6.txt 7.txt 8.txt 9.txt

In [3]: !cat 0.txt
Sat Mar 8 07:40:05 EST 2008
```

这个示例可能并不那么实用，因为很难有这样的需要：一下子创建10个文本文件，而且每个文本文件都包含日期。但这个示例却显示了如何将Python代码与shell代码结合。我们通过重复调用range()函数来创建一个列表，并且保存当前的项到变量i中。在每一次循环中，使用shell字符!来执行date命令。注意，这里调用date的语法，等同于已经定义了一个shell变量i，然后调用它。因此，date被调用，并且输出结果被重定向到文件*{current list item}.txt*中。在创建之后，我们使用ls命令列出了所有的文件，并且使用cat命令显示输出了其中一个文件。从文件的内容可以看到，这是一个日期。

可以将在Python中取得的任何值，传递到系统shell中。如果数值来自通过计算产生的数据库或是一个数据文件、一个XMLRPC服务，或者从文本文件中提取出的数据，可以先将其放到Python中，然后再使用!将其传递给系统shell。

字符串处理

IPython另一个强有力的特征是提供了采用字符串方式处理系统shell命令执行结果的功能。如果想查看属于用户jmjones的所有进程的PID值，可以通过输入以下命令实现：

```
ps aux | awk '{if ($1 == "jmjones") print $2}'
```

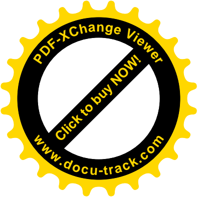
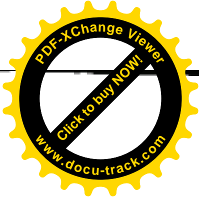
该命令十分紧凑、简练，而且可读性也很强。接下来，让我们看看如何使用IPython来处理相同的任务。首先，提取非过滤命令ps aux的输出结果：

```
In [1]: ps = !ps aux

In [2]:
```

ps aux的执行结果是一种类列表的结构，保存在变量ps中，其数据项是从系统shell调用返回的结果。这里所说的类列表结构继承了内建的列表类型，所以能够支持这种类型的各种方法。因此，如果有一个函数或是方法的输入是一个列表，你可将这些结果对象传递给它。另外，除了能够对标准的列表方法提供支持外，ps也支持一些非常有趣的方法和方便使用的属性。为了能说明都有哪些有趣的方法，我们将偏离要找到全部属于jmjones所有的进程这个任务一小会儿。第一个十分有趣的方法是指grep()方法。这是一个基本的、非常简单的过滤器，可以决定输出中保留哪些行，删除哪些行。例如，要查看是否在输出中存在一些可以匹配lighttpd的行，可以输入下面的内容：

```
In [2]: ps.grep('lighttpd')
```


www.TopSage.com

```
Out[2]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: www-data 4905 0.0 0.1........0:00 /usr/sbin/lighttpd -f /etc/lighttpd/l
```

我们调用了grep()方法，并向其传递了一个正则表达式lighttpd。记住，传递给grep()的正则表达式是大小写敏感的。grep()的调用结果是一行输出，该行输出表示正则表达式“lightpd”有一个正向匹配。我们可以像下面这样查看除匹配特定正则表达式外的所有记录：

```
In [3]: ps.grep('Mar07', prune=True)

Out[3]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: USER       PID %CPU %MEM    VSZ   RSS TTY    STAT START   TIME COMMAND
1: jmjones  19301  0.0  0.4  21364  4272 pts/2  Ss+  03:58   0:00 bash
2: jmjones  21340  0.0  0.9 202484 10184 pts/3  Sl+  07:00   0:06 vim ipytho
3: jmjones  23024  0.0  1.1  81480 11600 pts/4  S+   08:58   0:00 /home/jmjo
4: jmjones  23025  0.0  0.0      0     0 pts/4  Z+   08:59   0:00 [sh] <defu
5: jmjones  23373  5.4  1.0  81160 11196 pts/0  R+   09:20   0:00 /home/jmjo
6: jmjones  23374  0.0  0.0   3908   532 pts/0  R+   09:20   0:00 /bin/sh -c
7: jmjones  23375  0.0  0.1  15024  1056 pts/0  R+   09:20   0:00 ps aux
```

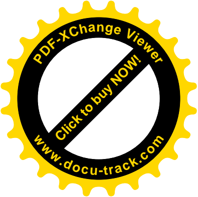
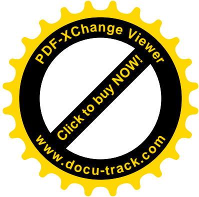
在将正则表达式“Mar07”传递给grep()方法后，可以看到大多数系统进程都是从Mar07（3月7日）开始运行的，因此我们决定查看所有不是在Mar07创建的进程。为了排除所有包含Mar07的记录项，传递了另一个参数给grep()，这个关键参数是：prune=True。这个关键参数告诉IPython “将匹配正则表达式的任何记录都删除掉”。正如你所看到的，输出结果中没有匹配Mar07的记录。

调用返回的结果也可以用于grep()。这表示grep()可以将函数作为一个参数来调用。它将函数传递给正在工作的列表项记录。如果函数返回值为真，则该项记录包括在过滤集中。例如，要创建一个目录列表，过滤掉文件或是目录：

```
In [1]: import os

In [2]: file_list = !ls

In [3]: file_list

Out[3]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: ch01.xml
1: code
2: ipython.pdf
3: ipython.xml
```

目录列表显示了四个文件。我们无法分辨列表中哪些是文件哪些是目录。但是如果使用os.path.isfile()进行过滤检测，就可以分辨出哪些是文件：

```
In [4]: file_list.grep(os.path.isfile)

Out[4]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: ch01.xml
```



PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

```
1: ipython.pdf
2: ipython.xml
```

这次，名为*code*的文件被过滤掉了，因此*code*根本就不是文件。接下来对目录进行过滤：

```
In [5]: file_list.grep(os.path.isdir)

Out[5]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: code
```

现在我们看到，code事实上就是一个目录。另一个有趣的方法是`fields()`。`fields()`可以在过滤完结果集并获得想要的数据之后（或许是之前），准确地列出希望显示的字段。看一下non-Mar07这个示例，在其中，我们输出了包含用户名、pid和登录时间的列：

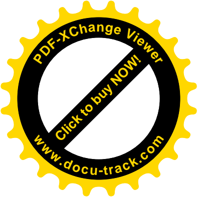
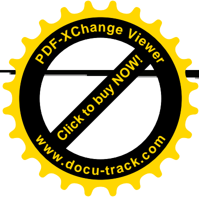
```
In [4]: ps.grep('Mar07', prune=True).fields(0, 1, 8)

Out[4]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: USER PID START
1: jmjones 19301 03:58
2: jmjones 21340 07:00
3: jmjones 23024 08:58
4: jmjones 23025 08:59
5: jmjones 23373 09:20
6: jmjones 23374 09:20
7: jmjones 23375 09:20
```

首先需要注意的是，不管使用`fields()`方法做什么，都是使用该方法对`grep()`方法的执行结果进行处理。之所以可以如此操作，是因为`grep()`返回一个与`ps`对象相同类型的对象，而`fields`本身返回与`grep()`相同类型的对象。基于此，可以将`grep()`与`fields()`联合在一起进行调用。现在，让我们看看这是如何工作的。`fields()`方法使用了多个数字作为参数，这些参数是我们希望输出的列，而且输出行中各列应当是被空白字符分隔的。你可能会想到这非常像awk对文本进行处理时所采用的默认分隔。本例调用了`fields()`方法来查看第0、1、8列，也就是USERNAME、PID和STARTTIME字段。

现在，回到显示所有属于`jmjones`的进程的PID值这个示例：

```
In [5]: ps.fields(0, 1).grep('jmjones').fields(1)

Out[5]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: 5385
1: 5388
2: 5423
3: 5425
4: 5429
5: 5431
6: 5437
7: 5440
```


www.TopSage.com

```
8: 5444
<continues on...>
```

这个示例首先将结果集进行削减，只余下第0列和第1列，即用户名字段和PID字段。然后，用grep()方法从削减后的结果集中提取出包含jmjones的记录。最后通过调filed(1)将过滤结果集中的第2个字段提取出来（注意，列的字段是从0开始编号的）。

关于字符串处理，我们最后想说一说能够直接访问进程列表的对象的s属性。这个对象或许不能给出需要寻找的结果。为了使系统shell与输出协同工作，可以在进程列表对象中使用s属性。

```
In [6]: ps.fields(0, 1).grep('jmjones').fields(1).s

Out[6]: '5385 5388 5423 5425 5429 5431 5437 5440 5444 5452 5454 5457 5458 5468
5470 5478 5480 5483 5489 5562 5568 5593 5595 5597 5598 5618 5621 5623 5628 5632
5640 5740 5742 5808 5838 12707 12913 14391 14785 19301 21340 23024 23025 23373
23374 23375'
```

s属性给了我们一个空格分隔的PID字符串，这些PID标明了在系统调用时可以使用的进程。我们希望能够将这个字符串字段列保存到一个名为pids的变量中，并且可以在IPython中执行类似kill $pids的操作。但是这会给用户jmjones的所有进程都发送一个SIGTERM信号，杀死它的文本编辑器和IPython会话。

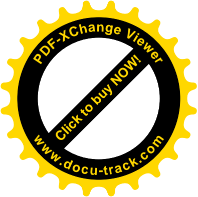
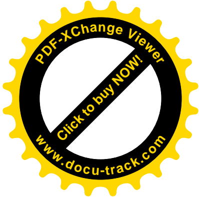
在IPython脚本中使用下面的awk单行命令，就可以成功地实现之前所说的目标：

```
ps aux | awk '{if ($1 == "jmjones") print $2}'
```

介绍上述这些概念之后，我们同样可以通过一系列命令成功完成这一目标。grep()方法采用了称为字段（field）的选项参数。如果我们指定了字段参数，那么为了将所需要的内容收集到结果集中，搜索将按顺序匹配字段：

```
In [1]: ps = !ps aux

In [2]: ps.grep('jmjones', field=0)

Out[2]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: jmjones 5361 0.0 0.1 46412 1828 ?           SL    Apr11
  0:00 /usr/bin/gnome-keyring-daemon -d
1: jmjones 5364 0.0 1.4 214948 14552 ?         Ssl   Apr11
  0:03 x-session-manager
....
53: jmjones 32425 0.0 0.0 3908 584 ?           S     Apr15
  0:00 /bin/sh /usr/lib/firefox/run-mozilla.
54: jmjones 32429 0.1 8.6 603780 88656 ?       Sl    Apr15
  2:38 /usr/lib/firefox/firefox-bin
```

www.TopSage.com

尽管这准确匹配了想要的行，但却输出了每行所有的列。为了取得指定PID值，应该作如下操作：

```
In [3]: ps.grep('jmjones', field=0).fields(1)

Out[3]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: 5361
1: 5364
....
53: 32425
54: 32429
```

这样，就实现了与awk过滤器相同的目标。

sh profile

一个我们没有讲到的概念是profile。一个profile是一个简单的配置集，在启动IPython时被加载。你可以自定义一些profile配置文件，让IPython可以根据会话的需要按不同的方式运行。要激活一个特定的profile配置文件，需要使用-p命令行选项并指定所使用的profile文件。

sh profile或者是shell profile是IPython内建的配置文件之一。sh profile可以使IPython在使用系统调用时更为友好。sh profile有两个配置项与标准IPython不同，sh不但显示当前目录而且rehash你的PATH，这样就可以与在Bash中一样，立刻访问所有的可执行程序。

除了设置一些配置值，sh profile也可以启动一些有助于shell的扩展。例如，启用环境持久性（envpersist）扩展。环境持久性扩展可以帮助你简单、持续地修改IPython sh profile中各种各样的环境变量，而无须升级.bash_profile或.bashrc。

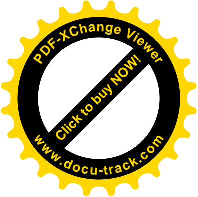
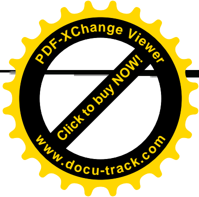
以下是PATH的内容：

```
jmjones@dinkgutsy:tmp$ ipython -p sh
IPython 0.8.3.bzr.r96 [on Py 2.5.1]
[~/tmp]|2> import os
[~/tmp]|3> os.environ['PATH']
       <3> '/home/jmjones/local/python/psa/bin:
       /home/jmjones/apps/lb/bin:/home/jmjones/bin:
       /usr/local/sbin:/usr/local/bin:/usr/sbin:
       /usr/bin:/sbin:/bin:/usr/games'
```

现在添加:/appended到当前PATH变量的后面：

```
[~/tmp]|4> env PATH+=:/appended
PATH after append = /home/jmjones/local/python/psa/bin:
/home/jmjones/apps/lb/bin:/home/jmjones/bin:
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
/sbin:/bin:/usr/games:/appended
```


www.TopSage.com

在当前PATH变量开始位置添加/prepended：

```
[~/tmp]|5> env PATH-=/prepended:
PATH after prepend = /prepended:/home/jmjones/local/python/psa/bin:
/home/jmjones/apps/lb/bin:/home/jmjones/bin:/usr/local/sbin:
/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/appended
```

下面显示了使用os.environ的PATH环境变量：

```
[~/tmp]|6> os.environ['PATH']
<6> '/prepended:/home/jmjones/local/python/psa/bin:
/home/jmjones/apps/lb/bin:/home/jmjones/bin:
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:
/bin:/usr/games:/appended'
```

现在退出IPython shell：

```
[~/tmp]|7>
Do you really want to exit ([y]/n)?
jmjones@dinkgutsy:tmp$
```

最后，打开一个新的IPython shell，查看PATH环境变量的值：

```
jmjones@dinkgutsy:tmp$ ipython -p sh
IPython 0.8.3.bzr.r96 [on Py 2.5.1]
[~/tmp]|2> import os
[~/tmp]|3> os.environ['PATH']
        <3> '/prepended:/home/jmjones/local/python/psa/bin:
        /home/jmjones/apps/lb/bin:/home/jmjones/bin:/usr/local/sbin:
        /usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/appended'
```

有趣的是，尽管没有更新配置脚本，但在PATH之前和之后所添加的值都被显示了出来。IPython可以保证对PATH的修改即时生效，并且无须再做额外的工作。现在显示一下所有能够即时生效的环境变量。

```
[~/tmp]|4> env -p
        <4> {'add': [('PATH', ':/appended')], 'pre': [('PATH', '/prepended:')], 'set': {}}
```

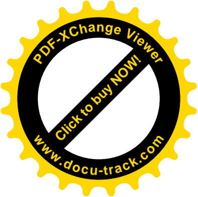
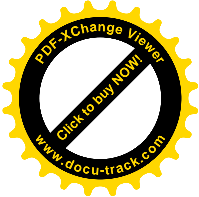
可以删除对PATH即时生效的设置：

```
[~/tmp]|5> env -d PATH
Forgot 'PATH' (for next session)
```

可以检查PATH的值：

```
[~/tmp]|6> os.environ['PATH']
        <6> '/prepended:/home/jmjones/local/python/psa/bin:/home/jmjones/apps/lb/bin:
        /home/jmjones/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
        /sbin:/bin:/usr/games:/appended'
```

www.TopSage.com

可以看到，在告诉IPython删除对PATH的即时生效的设置之后，之前设置的PATH值仍然保留着。实际上，删除的设置已经起作用了，IPython将会删除对这些项的即时生效指令。注意，一些以某些环境变量开始的进程会保留这些值，直到发生了修改。也就是说，当下次IPython shell启动时，PATH的值就不一样了：

```
[~/tmp]|7>
Do you really want to exit ([y]/n)?
jmjones@dinkgutsy:tmp$ ipython -p sh
IPython 0.8.3.bzr.r96 [on Py 2.5.1]
[~/tmp]|2> import os
[~/tmp]|3> os.environ['PATH']
       <3> '/home/jmjones/local/python/psa/bin:/home/jmjones/apps/lb/bin:
       /home/jmjones/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
       /sbin:/bin:/usr/games'
```

正如所期望的，PATH的值已经恢复到开始修改之前的情况了。

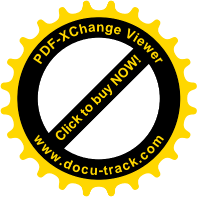
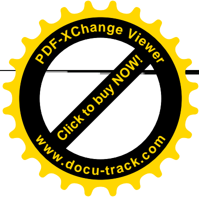
另一个在sh profile中非常有用的特征是mglob。mglob的许多一般性设置语法非常简单。例如，要查找所有的Django目录中的.*py*文件，可以这样操作：

```
[django/trunk]|3> mglob rec:*py
              <3> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: ./setup.py
1: ./examples/urls.py
2: ./examples/manage.py
3: ./examples/settings.py
4: ./examples/views.py
...
1103: ./django/conf/project_template/urls.py
1104: ./django/conf/project_template/manage.py
1105: ./django/conf/project_template/settings.py
1106: ./django/conf/project_template/__init__.py
1107: ./docs/conf.py
[django/trunk]|4>
```

rec指令简单地对它后面的模式执行递归查找。在本例中，**py*就是所谓的“模式”。为了显示Django目录中所有的目录，可以使用下面的命令：

```
[django/trunk]|3> mglob dir:*
              <3> SList (.p, .n, .l, .s, .grep(), .fields() available).
              Value:
              0: examples
              1: tests
              2: extras
              3: build
              4: django
              5: docs
              6: scripts
              </3>
```


www.TopSage.com

mglob命令返回一个Python列表对象。因此，在Python中可以执行的操作，同样也可以在这个返回文件或目录的列表中执行。

我们以上所介绍的只是sh profile中的几个部分。sh profile还有一些特征和特征选项，但未能在这里作一一介绍。

信息搜集

IPython不仅是一个能够帮助你完成工作的shell，它也可以像工具一样，搜集各种类型的、与正在使用的代码和对象相关的信息。它可以执行信息挖掘，感觉就像是一个调查或侦测工具。本节将简要介绍IPython中能够帮助搜集信息的一些特性。

page

如果正在处理的对象表示起来太过复杂，无法在一屏中完全显示，可以试试页（page）函数。page可以用来打印对象并且可以通过一个pager来运行。在许多系统中，默认的pager是less，但也可以使用其他的pager。标准用法如下：

```
In [1]: p = !ps aux
==
['USER        PID  %CPU %MEM    VSZ    RSS  TTY       STAT START   TIME COMMAND',
 'root          1   0.0  0.1   5116   1964  ?         Ss Mar07     0:00 /sbin/init',
< ... trimmed result ... >
In [2]: page p
['USER        PID  %CPU %MEM    VSZ    RSS  TTY       STAT START   TIME COMMAND',
 'root        1 0.0   0.1 5116   1964     ?  Ss       Mar07        0:00 /sbin/init',
< ... trimmed result ... >
```

这里，将系统shell命令ps aux的执行结果保存到变量p中。之后调用page，并且将处理结果对象传递给它。随后，page函数启动less。

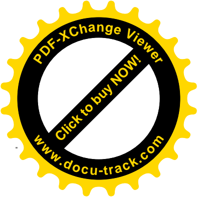
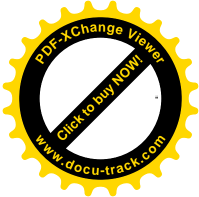
page有一个选项-r。该选项告诉page不要美化打印（译注1）（pretty print）对象，而是通过pager运行它的字符串表示（str()的执行结果）。结果看起来类似这样：

```
In [3]: page -r p
ilus-cd-burner/mapping-d', 'jmjones 5568 0.0 1.0 232004 10608 ? S
Mar07    0:00 /usr/lib/gnome-applets/trashapplet --', 'jmjones 5593 0.0 0.9
188996 10076 ?         S     Mar07    0:00 /usr/lib/gnome-applets/battstat-apple',
'jmjones    5595  0.0 2.8  402148 29412 ?          S     Mar07    0:01 p
< ... trimmed result ... >
```

这个非美化打印（non-pretty-print）的结果确实是不够完美。我们建议还是从美化打印开始，并在此基础上进行工作。

译注1：　美化打印是指自动格式化输出产生统一的缩进格式。

www.TopSage.com

pdef

魔力pdef函数能打印输出任何可被调用对象的定义名或是函数声明。这个示例创建了一个函数，并且该函数有注释和返回语句：

```
In [1]: def myfunc(a, b, c, d):
   ...:     '''return something by using a, b, c, d to do something'''
   ...:     return a, b, c, d
   ...:

In [2]: pdef myfunc
 myfunc(a, b, c, d)
```

pdef函数忽略了注释和返回语句，而输出了函数的声明部分。可以在任何可调用函数中这样使用。即使函数的源代码不可用，只要能够访问*.pyc*文件或egg文件，pdef函数就依然可以使用。

pdoc

pdoc函数可以打印传递给它的函数的注释信息。这里使用pdoc处理在pdef示例中使用的myfunc()函数：

```
In [3]: pdoc myfunc
Class Docstring:
    return something by using a, b, c, d to do something
Calling Docstring:
    x.__call__(...) <==> x(...)
```

这是一个相当完美的自解释（self-explanatory）。

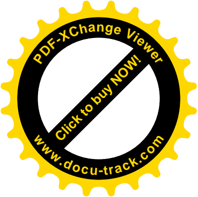
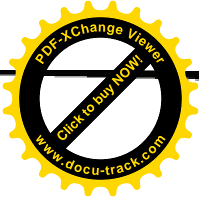
pfile

pfile函数能够运行对象的文件，但前提是对象所包含的文件能够找得到。例如：

```
In [1]: import os

In [2]: pfile os

r"""OS routines for Mac, NT, or Posix depending on what system we're on.

This exports:
  - all functions from posix, nt, os2, mac, or ce, e.g. unlink, stat, etc.

< ... trimmed result ... >
```

本例引入了os模块，并且通过less运行。这个示例能够帮助你理解一个代码段是如何开始运行的。显然，如果所包含的文件仅仅是egg或*.pyc*文件，pfile函数将不起作用。

www.TopSage.com

注意：从??操作符可以看到与使用魔力函数%pdef、%pdoc和%pfile可以看到的相同的信息。优先选择的方法是??。

pinfo

pinfo函数以及相关的工具使用起来非常方便。很难想象如果没有它们会怎么样。pinfo函数提供了诸如类型、基础类、命名空间和注释等信息。例如，有一个模块如下所示：

```
#!/usr/bin/env python

class Foo:
    """my Foo class"""
    def __init__(self):
        pass

class Bar:
    """my Bar class"""
    def __init__(self):
        pass

class Bam:
    """my Bam class"""
    def __init__(self):
        pass
```

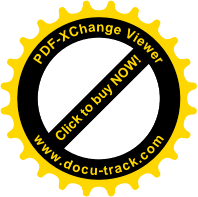
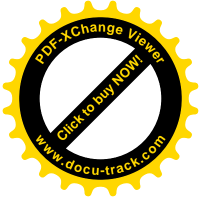
我们可以从模块自身获得相应的信息：

```
In [1]: import some_module

In [2]: pinfo some_module
Type:           module
Base Class:     <type 'module'>
String Form:    <module 'some_module' from 'some_module.py'>
Namespace:      Interactive
File:           /home/jmjones/code/some_module.py
Docstring:
    <no docstring>
```

也可以获得模块中所包含的类的相关信息：

```
In [3]: pinfo some_module.Foo
Type:              classobj
String Form:       some_module.Foo
Namespace:         Interactive
File:              /home/jmjones/code/some_module.py
Docstring:
    my Foo class

Constructor information:
Definition:        some_module.Foo(self)
```

www.TopSage.com

此外，还可以获得类的实例的相关信息：

```
In [4]: f = some_module.Foo()

In [5]:         pinfo f
Type:           instance
Base Class:     some_module.Foo
String Form:    <some_module.Foo instance at 0x86e9e0>
Namespace:      Interactive
Docstring:
    my Foo class
```

在对象名之前或之后的问号（?）提供了与pinfo相同的功能：

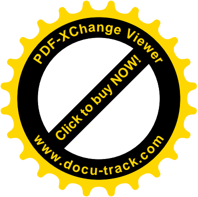
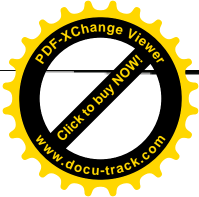
```
In [6]: ? f
Type:           instance
Base Class:     some_module.Foo
String Form:    <some_module.Foo instance at 0x86e9e0>
Namespace:      Interactive
Docstring:
    my Foo class


In [7]: f ?
Type:           instance
Base Class:     some_module.Foo
String Form:    <some_module.Foo instance at 0x86e9e0>
Namespace:      Interactive
Docstring:
    my Foo class
```

但是，在对象名前面或后面的两个问号（??）可以提供更多的信息：

```
In [8]: some_module.Foo ??
Type:           classobj
String Form:    some_module.Foo
Namespace:      Interactive
File:           /home/jmjones/code/some_module.py
Source:
class Foo:
    """my Foo class"""
    def __init__(self):
        pass
Constructor information:
Definition:      some_module.Foo(self)
```

可以看到，??不但能够提供pinfo提供给我们的全部信息，而且还能够提供所请求对象的源代码。由于仅仅是对类进行查询，因此??提供的结果是类的源码，而不是整个文件。这正是pinfo函数的特点。也正因为如此，它比其他函数更为常用。

www.TopSage.com

psource

psource函数显示定义的元素的源代码，不论该元素是一个模块或是模块中的类或函数。为将其显示出来，psource通过运行page显示源代码。以下是一个psource针对模块的应用示例：

```
In [1]: import some_other_module

In [2]: psource some_other_module
#!/usr/bin/env python

class Foo:
    """my Foo class"""
    def __init__(self):
        pass

class Bar:
    """my Bar class"""
    def __init__(self):
        pass

class Bam:
    """my Bam class"""
    def __init__(self):
        pass

def baz():
    """my baz function"""
    return None
```

这是一个psource针对模块中的一个类的应用示例：

```
In [3]: psource some_other_module.Foo
class Foo:
    """my Foo class"""
    def __init__(self):
        pass
```

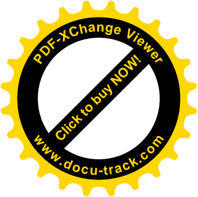
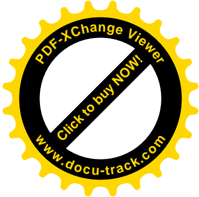
接下来是psource针对模块中的一个函数的应用示例：

```
In [4]: psource some_other_module.baz
def baz():
    """my baz function"""
    return None
```

psearch

psearch魔力函数不但能够依据名称查找Python对象，还可以使用通配符协助查找。我们这里只简略地描述psearch函数，如果你想要知道更多的信息，可以通过在IPython提示符下输入magic来查看帮助文档。

www.TopSage.com

让我们从声明下面的对象开始：

```
In [1]: a = 1

In [2]: aa = "one"

In [3]: b = 2

In [4]: bb = "two"

In [5]: c = 3

In [6]: cc = "three"
```

我们可以查找所有以a、b、c开头的对象，例如：

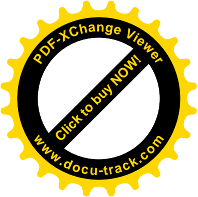
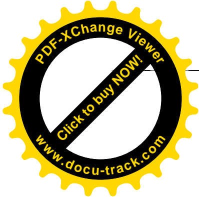
```
In [7]: psearch a*
a
aa
abs
all
any
apply

In [8]: psearch b*
b
basestring
bb
bool
buffer

In [9]: psearch c*
c
callable
cc
chr
classmethod
cmp
coerce
compile
complex
copyright
credits
```

注意，这里能够查到所有的对象，而不仅是a、aa、b、bb、c、cc，并且都是内建对象。

能够快速替换psearch函数的方法就是使用问号（?）操作符，下面是一个示例：

```
In [2]: import os

In [3]: psearch os.li*
os.linesep
os.link
os.listdir
```


www.TopSage.com

```
In [4]: os.li*?
os.linesep
os.link
os.listdir
```

除了psearch，还可以使用*?。

psearch在执行搜索操作时可以使用-s选项，排除搜索时可以使用-e选项，搜索的范围是内建的命名空间。命名空间包括builtin、user、user_global、internal和alias。默认情况下，psearch搜索builtin和user空间。如果只是明确地对用户进行搜索，传递-e builtin选项给psearch能够排除对内建空间的搜索。这似乎有点违反常规，但非常有意义。psearch的默认搜索路径是builtin和user，所以如果我们指定-s user，搜索builtin和user仍会得到我们想要的结果。在这个示例中，搜索又执行了一次。注意，这些结果不包括内建的命名空间：

```
In [10]: psearch -e builtin a*
a
aa

In [11]: psearch -e builtin b*
b
bb

In [12]: psearch -e builtin c*
c
cc
```

psearch函数允许搜索指定类型的对象。这里搜索user命名空间中的整数类型对象。

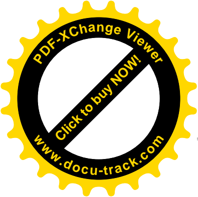
```
In [13]: psearch -e builtin * int
a
b
c
```

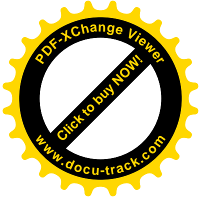
接下来是对字符串的搜索：

```
In [14]: psearch -e builtin * string
_
__
__name__
aa
bb
cc
```

这里出现的_和__objects是IPython的缩略表示，表示之前的返回结果。__name__object是一个指定的变量，表示模块的名称。如果__name__是“__main__”，这表示模块是从解释器运行而不是从另外一个模块引入的。

www.TopSage.com





who

IPython还提供了一些能够列出所有交互式对象的方法。第一个就是who函数。下面是之前的一个示例，其中通过who函数显示变量a、aa、b、bb、c、cc：

```
In [15]: who
a        aa        b        bb        c        cc
```

who函数的使用非常直接明了，我们得到了一个返回的简单列表，包括了所有交互定义的对象。可以使用who函数对类型进行过滤，例如：

```
In [16]: who int
a        b        c

In [17]: who str
aa       bb       cc
```

who_ls

who_ls函数与who函数十分相似，但who_ls函数返回的是一个列表而不是所匹配变量的名称。下面是一个示例，演示了没有参数的who_ls函数：

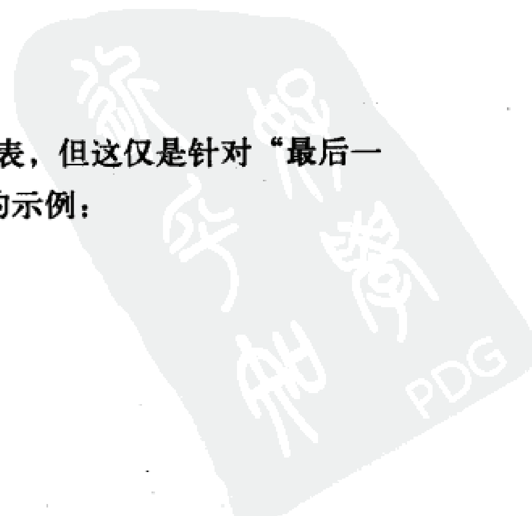
```
In [18]: who_ls

Out[18]: ['a', 'aa', 'b', 'bb', 'c', 'cc']
```

接下来是一个基于对象类型进行过滤的示例：

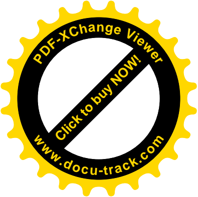
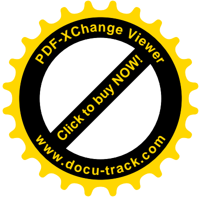
```
In [19]: who_ls int

Out[19]: ['a', 'b', 'c']

In [20]: who_ls str

Out[20]: ['aa', 'bb', 'cc']
```

由于who_ls返回名称列表，可以使用_variable访问名称列表，但这仅是针对“最后一次的输出”。下面是循环显示上次返回的匹配变量名称列表的示例：

```
In [21]: for n in _:
   ....:     print n
   ....:
   ....:
aa
bb
cc
```



www.TopSage.com

whos

whos函数与who函数非常相似，只是whos打印输出详细信息，而who不打印输出。下面是whos函数的一个示例，其中使用了非命令行参数：

```
In [22]: whos
Variable   Type    Data/Info
----------------------------
a          int     1
aa         str     one
b          int     2
bb         str     two
c          int     3
cc         str     three
n          str     cc
```

接下来，正如在who示例中所做的那样，使用whos根据类型进行过滤：

```
In [23]: whos int
Variable   Type    Data/Info
----------------------------
a          int     1
b          int     2
c          int     3

In [24]: whos str
Variable   Type    Data/Info
----------------------------
aa         str     one
bb         str     two
cc         str     three
n          str     cc
```

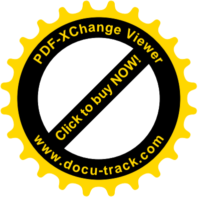
历史

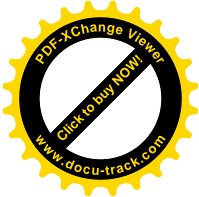
在IPython中，有两种方式可以访问输入的命令历史（History）。第一种是基于行（readline-based）的方式，第二种是基于hist函数的方式。

行支持（readline support）

到目前为止，你已经了解了IPython许多非常酷的特性，这些特性是在面向行的应用中常常会用得上的。如果习惯于使用Ctrl-s搜索Bash历史，在IPython中使用相同的功能时就不会有任何麻烦。下面，我们定义了一些变量，然后向后搜索整个历史。

```
In [1]: foo = 1

In [2]: bar = 2

In [3]: bam = 3

In [4]: d = dict(foo=foo, bar=bar, bam=bam)
```






```
In [5]: dict2 = dict(d=d, foo=foo)

In [6]: <CTRL-s>

(reverse-i-search)`fo': dict2 = dict(d=d, foo=foo)

<CTRL-r>

(reverse-i-search)`fo': d = dict(foo=foo, bar=bar, bam=bam)
```

首先输入Ctrl-r来启动搜索，然后输入fo作为搜索标准。它返回输入的行，如IPython中In[5]所示。使用行搜索功能，按Ctrl-r，它返回匹配输入的行，如IPython中In[4]所示。

可以通过行（readline）操作来完成更多的内容，但这里我们只能简单地做一个介绍。Ctrl-a让你回到行的开始位置，Ctrl-e让光标跳到行的结尾处。Ctrl-f用于删除字符，Ctrl-h能够向后删除一个字符（相当于backspace）。Ctrl-p将历史记录中的行向后移动一行，Ctrl-nt则是向前移动一行。如果想了解更多的行操作，可以在*nix系统中输入man readline进行查看。

hist命令（hist command）

除了具有访问行操作历史的各项功能外，IPython也提供了称为history的历史函数。可以使用hist代替history。如果不带参数，hist会连续打印用户输入命令的列表。默认情况下，该列表会被编号。例如设置一些变量，切换目录，然后运行hist命令：

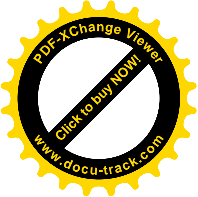
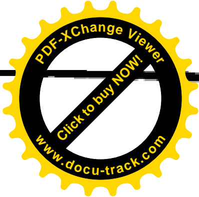
```
In [1]: foo = 1

In [2]: bar = 2

In [3]: bam = 3

In [4]: cd /tmp
/tmp

In [5]: hist
1: foo = 1
2: bar = 2
3: bam = 3
4: _ip.magic("cd /tmp")
5: _ip.magic("hist ")
```

在历史列表的第4和第5项是magic函数。注意，它们已经被IPython修改过。你可以看到如何通过Ipython调用magic()函数的过程。

如果希望去掉行号，可以使用-n选项。下面是hist命令使用-n选项的示例：

```
kIn [6]: hist -n
foo = 1
bar = 2
bam = 3
```


www.TopSage.com

```
_ip.magic("cd /tmp")
_ip.magic("hist ")
_ip.magic("hist -n")
```

如果在IPython中工作时想往文本编辑器中粘贴一段IPython的代码，这将非常有帮助。

-t选项返回一个被翻译的命令历史视图，历史命令记录了IPython看到的用户输入的命令。这是默认设置。在下面的示例中使用-t选项输出了到目前为止建立起来的命令历史：

```
In [7]: hist -t
1: foo = 1
2: bar = 2
3: bam = 3
4: _ip.magic("cd /tmp")
5: _ip.magic("hist ")
6: _ip.magic("hist -n")
7: _ip.magic("hist -t")
```

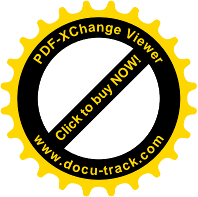
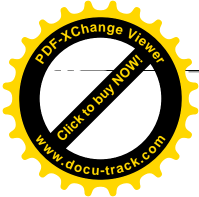
"raw history"或是选项-r能够准确显示输入了什么。下面的示例显示了在之前的示例中添加了"raw history"标志后的输出结果：

```
In [8]: hist -r
1: foo = 1
2: bar = 2
3: bam = 3
4: cd /tmp
5: hist
6: hist -n
7: hist -t
8: hist -r
```

IPython的-g标志也提供了一种搜索历史中指定模式的方式。下面的示例使用前面的示例配合-g标志对命令历史进行搜索：

```
In [9]: hist -g hist
0187: hist
0188: hist -n
0189: hist -g import
0190: hist -h
0191: hist -t
0192: hist -r
0193: hist -d
0213: hist -g foo
0219: hist -g hist
===
^shadow history ends, fetch by %rep <number> (must start with 0)
=== start of normal history ===
5 : _ip.magic("hist ")
6 : _ip.magic("hist -n")
7 : _ip.magic("hist -t")
```

www.TopSage.com

```
8 : _ip.magic("hist -r")
9 : _ip.magic("hist -g hist")
```

注意，之前的示例中返回了“shadow history”一词。shadow history是包括你输入的每一个命令的历史。“shadow history”从0开始显示在结果集的起始部分。来自会话的历史结果被保存在结果集的最后，但不以0开始。

历史结果（History results）

在Python和IPython中，不仅可以访问曾经输入的命令历史列表，而且可以访问结果的历史。第一种实现此用途的方法是使用“_”标志，这表示“上次输出”。下面的示例展示了_函数在IPython中是如何工作的：

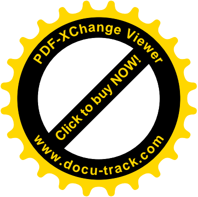
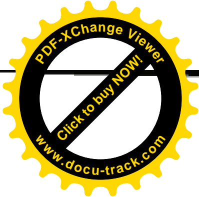
```
In [1]: foo = "foo_string"

In [2]: _

Out[2]: ''

In [3]: foo

Out[3]: 'foo_string'

In [4]: _

Out[4]: 'foo_string'

In [5]: a = _

In [6]: a

Out[6]: 'foo_string'
```

当我们在In[1]中定义了foo，在In[2]中的“_”返回了一个空字符串。当我们在In[3]中输出了foo，便可以使用“_”在In[4]中获得结果。在In[5]中，能够将结果保存到变量a中。

下面是使用标准Python shell运行相同示例的情形：

```
>>> foo = "foo_string"
>>> _
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_' is not defined
>>> foo
'foo_string'
>>> _
'foo_string'
>>> a = _
>>> a
'foo_string'
```


www.TopSage.com

可以看到，除了在试图访问“_”时输出了名字错误（NameError）异常，在IPython中的结果与标准Python shell的结果是非常相似的。

IPython在使用“上一次输出”的概念时更进一步：在本书“Shell的执行”部分中，我们描述了如何使用!和!!操作符，并且说明了不能将!!的结果保存到变量中，但是可以随后使用!!的结果。简言之，可以访问任何使用下划线之后跟随任一数字_[0-9]*的语法方式输出的结果。数字必须与想看到的Out[0-9]*结果相对应。

为了说明这一点，我们首先列出文件，但不对输出做任何处理：

```
In [1]: !!ls apa*py

Out[1]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: apache_conf_docroot_replace.py
1: apache_log_parser_regex.py
2: apache_log_parser_split.py

In [2]: !!ls e*py

Out[2]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: elementtree_system_profile.py
1: elementtree_tomcat_users.py

In [3]: !!ls t*py

Out[3]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: test_apache_log_parser_regex.py
1: test_apache_log_parser_split.py
```

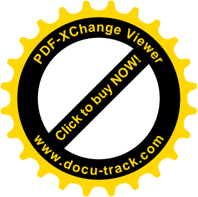
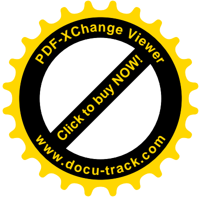
首先使用_1，_2和_3访问了Out[1-3]。接下来，为每一项添加一个更为明确的名字：

```
In [4]: apache_list = _1

In [5]: element_tree_list = _2

In [6]: tests = _3
```

现在，apache_list，element_tree_list和tests包括了相同的元素，分别对应输出中的Out[1]，Out [2]和Out [3]。

```
In [7]: apache_list

Out[7]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: apache_conf_docroot_replace.py
1: apache_log_parser_regex.py
2: apache_log_parser_split.py

In [8]: element_tree_list

Out[8]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: elementtree_system_profile.py
1: elementtree_tomcat_users.py
```

www.TopSage.com

```
In [9]: tests

Out[9]: SList(.p, .n, .l, .s, .grep(), .fields() available). Value:
0: test_apache_log_parser_regex.py
1: test_apache_log_parser_split.py
```

但是，所有这些的关键全部在于，在IPython中，可以使用_加指定的变量或是使用_加数字的方式，来访问之前输出的结果。

自动和快捷方式

即使IPython没能提高你的工作效率，它还是提供了一系列函数和特征来帮助你实现IPython任务和使用的自动化。

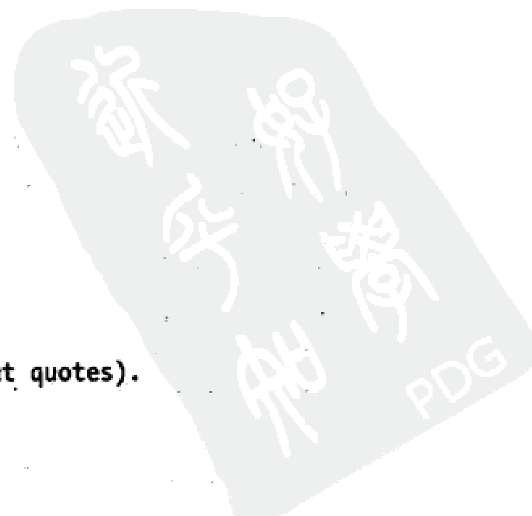
alias

首先说一说alias（别名）。在这一章的前面已经对别名进行了介绍，因此这里不再老调重谈。但是在这里特别指出的是，别名不仅能帮助你直接在IPython中使用*nix shell命令，而且可以帮助你将任务自动化。

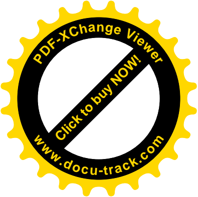
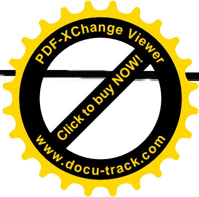
macro

macro（宏）函数允许你定义一个代码块，这段代码块可以在之后编写的程序中被内联（inline）执行，无论你正在编写什么代码都可以使用。这不同于创建函数或方法。宏，在某种意义上说，与当前代码运行的环境密切相关。如果有一个频繁在所有文件中执行的公共处理步骤，你就可以在文件中创建一个宏。为了更好地理解宏是如何在一系列文件中发挥作用的，请看下面的示例：

```
In [1]: dirlist = []

In [2]: for f in dirlist:
   ...:     print "working on", f
   ...:     print "done with", f
   ...:     print "moving %s to %s.done" % (f, f)
   ...:     print "*" * 40
   ...:
   ...:
In [3]: macro procdir 2
Macro `procdir` created. To execute, type its name (without quotes).
Macro contents:
for f in dirlist:
    print "working on", f
    print "done with", f
    print "moving %s to %s.done" % (f, f)
    print "*" * 40
```



www.TopSage.com

在In[2]中创建了一个循环，而在变量dirlist中没有循环需要的宏列表项，这是因为我们希望将来使用循环时再将循环列表项加入到变量dirlist中。我们创建了一个名为procdir的宏来遍历列表。创建宏的语法是macro macro_name range_of_lines。这里的range of lines是行的列表，来自想要合并到宏中的历史。宏列表的行应该用以空格为分隔的数字序列或表示数字序列的数字范围来标识（例如1-4）。

在这个示例中，我们创建一个系统文件名，并将其保存到dirlist变量中。通过执行procdir宏，遍历dirlist中的所有文件名：

```
In [4]: dirlist = ['a.txt', 'b.txt', 'c.txt']

In [5]: procdir
------> procdir()
working on a.txt
done with a.txt
moving a.txt to a.txt.done
****************************************
working on b.txt
done with b.txt
moving b.txt to b.txt.done
****************************************
working on c.txt
done with c.txt
moving c.txt to c.txt.done
****************************************
```

一旦定义了一个宏，就可以打开文本编辑器编辑它。在继续使用它之前，对其进行调试并确定它是否正确是非常必要的。

store

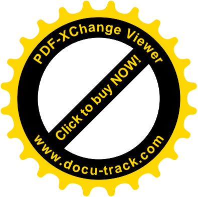
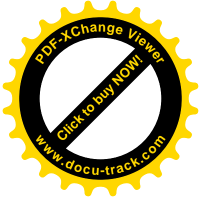
通过store魔力函数可以一直使用你的宏和一些普通的Python变量。store函数的标准用法就是store variable。然而，store函数也可以包含一些非常有用的参数：-d variable函数能够从持久存储包（persistence store）中删除指定的变量，-z函数能够删除所有存储的变量，-r函数能够从持久存储包中重新加载所有变量。

reset

reset函数用来从交互命名空间中删除所有变量。在下面的示例中，我们先定义了3个变量，再使用whos来检测它们的设置，之后reset命名空间，最后再次使用whos来验证它们已经被删除：

```
In [1]: a = 1

In [2]: b = 2
```

www.TopSage.com

```
In [3]: c = 3

In [4]: whos
Variable   Type    Data/Info
----------------------------
a          int     1
b          int     2
c          int     3

In [5]: reset
Once deleted, variables cannot be recovered. Proceed (y/[n])? y

In [6]: whos
Interactive namespace is empty.
```

run

run函数可以在IPython中执行指定的文件。在其他应用中，它允许使用外部文本编辑器修改一个Python模块，并在Ipython中交互式地测试这些修改。在执行指定的程序之后，将会返回到IPython shell。使用run的语法是run options specified_file args。

-n选项使得模块的__name__变量设置成它自己的名称，而不是'__main__'。这使得模块运行与简单地载入十分相像。

-i选项在IPython的当前命名空间中运行模块，因此，使用运行的模块可以访问所有定义的变量。

-e选项使得IPython忽略对sys.exit()的调用和SystemExit异常。如果两者都没有发生，IPython继续执行。

-t选项使IPython输出模块运行的时间信息。

-d选项使得指定的模块运行在Python调试器（pdb）中。

-p选项在Python配置下运行指定的模块。

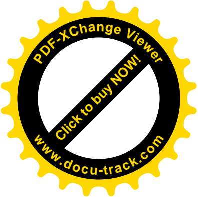
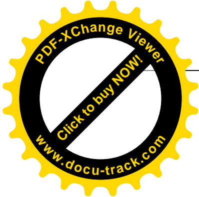
save

save函数会保存指定的输入行到指定的输出文件中。使用save的语法为save options filename lines。行可以使用与宏相同的范围格式。save仅有的选项是-r。-r表示将原始的输入而不是经过转换的内容进行保存。在标准Python中，对输入进行转换是默认的。

rep

rep是自动启用函数。rep函数有一些你或许觉得会非常有用的参数。使用不带参数的rep可以取回最近处理的结果，并在下一行输出时设置一个字符串进行表示。例如：

www.TopSage.com

```
In [1]: def format_str(s):
...: return "str(%s)" % s
...:

In [2]: format_str(1)

Out[2]: 'str(1)'

In [3]: rep

In [4]: str(1)
```

rep在In[3]被调用，这样你看到的文本被放到了In[4]中。这使得通过编程能够产生IPython需要处理的输入。尤其是当你混合使用generators和宏的时候，这非常方便。

普通的不带参数使用rep的方法是比较懒散、没有鼠标支持的编辑方式。如果你有一个包含一些值的变量，可以直接编辑这个值。作为一个示例，假设正在使用的函数对于指定的安装包返回到bin目录。我们将*bin*目录保存将在变量a中：

```
In [2]: a = some_blackbox_function('squiggly')

In [3]: a

Out[3]: '/opt/local/squiggly/bin'
```

如果输入rep，可以看到在新输入行中/opt/local/squiggly/bin之后出现闪烁的光标等待我们进行编辑：

```
In [4]: rep

In [5]: /opt/local/squiggly/bin<blinking cursor>
```

如果想去保存包的基目录，而不是bin目录，只需要删除路径最后的bin，在路径前加一个新变量名，之后是一个等号和前引号，最后加一个后引号即可：

```
In [5]: new_a = '/opt/local/squiggly'
```

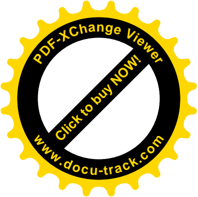
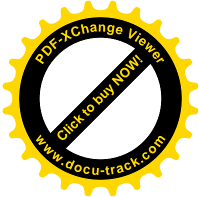
现在已经有一个包含字符串的变量，该字符串是包的基目录名。

的确，我们可以用鼠标进行复制和粘贴，但是使用起来比较麻烦。为什么不使用舒适的键盘，而去使用鼠标？现在可以根据包来使用一个新的标识_a，作为任何需要进行操作的基目录。

当把一个数字作为参数传递给rep时，IPython获得从历史记录得来的指定行，并且放到下一行中，然后将光标在放置行的末尾。这对于执行、编辑和再执行单行甚至一小段代码来说非常有帮助。例如：

```
In [1]: map = (('a', '1'), ('b', '2'), ('c', '3'))
```

www.TopSage.com

```
In [2]: for alph, num in map:
   ...:     print alph, num
   ...:
   ...:
a 1
b 2
c 3
```

这里，我们编辑In[2]，并打印输出数字值乘以2的结果，而不是一个非计算的值。我们既可以再次输入for循环，也可以使用rep：

```
In [3]: rep 2

In [4]: for alph, num in map:
    print alph, int(num) * 2
   ...:
   ...:
a 2
b 4
c 6
```

rep函数也可以使用数字范围作为参数。数字范围的语法与宏中的数字范围语法相同，这在本章的前面已经讨论过了。当你为rep指定一个范围时，行被立即执行。下面是rep的一个示例：

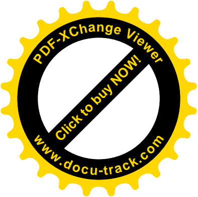
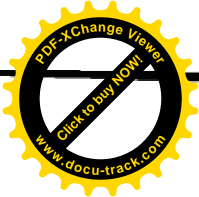
```
In [1]: i = 1

In [2]: i += 1

In [3]: print i
2

In [4]: rep 2-3
lines [u'i += 1\nprint i\n']
3

In [7]: rep 2-3
lines [u'i += 1\nprint i\n']
4
```

我们定义了一个递增计数器和在In[1]到In[3]之间打印输出当前计数值的代码，在In[4]和In[7]中告诉rep重复第2行和第3行。注意，由于第5和第6两行在In[4]之后执行，因此被错过了。

rep的最后一个选项是传递一个字符串。这与“将一个词传递给rep”或是“传递一个非引用搜索字符串到rep”非常相似。下面是一个示例：

```
In [1]: a = 1

In [2]: b = 2
```


www.TopSage.com

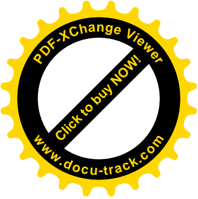
```
In [3]: c = 3

In [4]: rep a

In [5]: a = 1
```

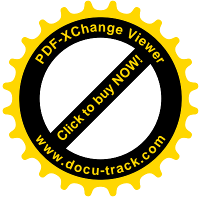
我们定义了一些变量，并且告诉rep重复包含字母“a”的行。它将In[1]返回给我们来编辑和重新执行。

本章小结

IPython是工具包中最常使用的一个工具。掌握了这个神奇的shell，就相当于掌握了一个神奇的文本编辑器：你越精通它，就可以越快速地完成单调乏味的工作。我们在几年前开始使用IPython的时候，就发现了这个工具了不起的强大功能。现在，Ipython得到进一步发展，变得更为强大了。grep函数和对字符串的处理功能是学习IPython时首先需要了解的特性，也是IPython最重要的优点。我们强烈建议你进一步深入学习IPython，你绝不会因为花费时间学习IPython而感到后悔的。

www.TopSage.com





第3章

文本

几乎所有的系统管理员都需要处理文本，无论其形式是日志、应用程序数据、XML、HTML、配置文件或是某些命令的输出结果。通常，你会使用诸如grep、awk这样的工具，但有时候可能需要一个更富表现力、更完美的工具来处理更为复杂的问题。在你需要利用从其他文件中提取的数据来创建新文件时，经常使用重定向文本处理工具（grep或awk）的输出到一个文件方法。实际上，一个易于扩展的工具可能更适合此项工作。

经验表明，相对Perl、bash或是其他语言而言，Python具有更富表现力，更完美，且更易于扩展的特点。关于为什么我们对Python的评价要比Perl或Bash（你同样可以使用sed或awk创建应用程序）更高，请参见第1章的内容。Python的标准库，语言特征和内建类型，对于读取文本文件、操作文本或是从文本文件中提取信息而言，都是非常强大的。Python及其标准库包含了大量灵活、功能强大的函数，适用于利用字符串类型、文件类型和正则表达式模块进行文本处理。标准库中最近新增了一个功能，即ElementTree，在需要处理XML时，ElementTree非常有用。在这一章中，我们将学习如何有效地使用标准库和内建组件来实现文本处理。

Python的内建功能及模块

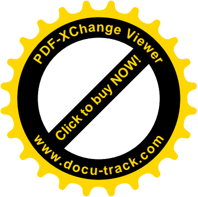
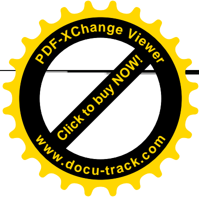
str

字符串由一系列字符组成。如果需要处理文本数据，则很可能需要使用一个字符串对象或是一系列字符串对象。字符串类型（str）是一个强大而灵活的、能够对字符串数据进行操作处理的数据类型。本节将演示如创建一个字符串，以及创建之后如何使用。

创建字符串

最普通的创建字符串的方法是在文本前后加上引号：

www.TopSage.com

```
In [1]: string1 = 'This is a string'

In [2]: string2 = "This is another string"

In [3]: string3 = '''This is still another string'''

In [4]: string4 = """And one more string"""

In [5]: type(string1), type(string2), type(string3), type(string4)
Out[5]: (<type 'str'>, <type 'str'>, <type 'str'>, <type 'str'>)
```

单引号、双引号、三引号可以完成相同的事：去掉创建一个str类型的对象。单引号和双引号在创建字符串时是相同的，可以替换使用。这与UNIX shell中引号的使用略有不同。在UNIX中两者是不可以替换使用的，例如：

```
jmjones@dink:~$ FOO=sometext
jmjones@dink:~$ echo "Here is $FOO"
Here is sometext
jmjones@dink:~$ echo 'Here is $FOO'
Here is $FOO
```

Perl在创建字符串时也使用单、双引号。下面是一个Perl脚本写的对比示例：

```
#!/usr/bin/perl

$FOO = "some_text";
print "-- $FOO --\n";
print '-- $FOO --\n';
```

下面是一个简单的Perl脚本的输出结果：

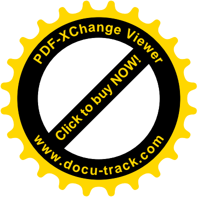
```
jmjones@dinkgutsy:code$ ./quotes.pl
-- some_text --
-- $FOO --\njmjones@dinkgutsy:code$
```

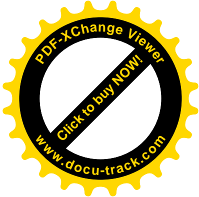
在Python中则不存在这样的差别。Python将差别留给了编程人员进行处理。例如，可以在单引号的字符串中嵌入双引号，并且不使用反斜线（“\”）进行字符转义。相反地，如果需要在字符串中嵌入一个单引号，并且不想对其进行转义，则该字符串应使用双引号，如例3-1。

例3-1：Python中单/双引号的比较

```
In [1]: s = "This is a string with 'quotes' in it"

In [2]: s
Out[2]: "This is a string with 'quotes' in it"
In [3]: s = 'This is a string with \'quotes\' in it'

In [4]: s
Out[4]: "This is a string with 'quotes' in it"

In [5]: s = 'This is a string with "quotes" in it'
```

www.TopSage.com


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

```
In [6]: s
Out[6]: 'This is a string with "quotes" in it'

In [7]: s = "This is a string with \"quotes\" in it"

In [8]: s
Out[8]: 'This is a string with "quotes" in it'
```

注意，在In [3]和In [7]中分别嵌入了一对相同类型的转义引号。

在希望一个字符串能够跨越多行时，可以在字符串中你希望分行的地方嵌入“\n”来解决，但是这种方法略显笨拙。另外一种更为简洁的替换方法是使用三引号。三引号允许创建多行字符串。例3-2中先演示了在多行字符串中单引号的使用，之后演示了三引号的使用：

例3-2：三引号

```
In [6]: s = 'this is
------------------------------------------------------------
  File "<ipython console>", line 1
    s = 'this is
               ^
SyntaxError: EOL while scanning single-quoted string

In [7]: s = '''this is a
   ...: multiline string'''

In [8]: s
Out[8]: 'this is a\nmultiline string'
```

对于复杂的处理，在Python中有另一种被称为“原始”字符串的字符串表示方法。在创建一个字符串时，通过在引号之前放置字母r，可以创建一个原始字符串。从根本上讲，创建一个原始字符串与创建一个非原始字符串的区别在于，Python不对原始字符串中的转义字符进行解析，而在处理普通字符串时，则对其进行解析。Python遵循类似于标准C语言对转义序列的一系列规则。例如，在普通字符串中，“\t”被解析成tab字符，“\n”被解析成换行，“\r”被解析成回车。表3-1展示了Python中使用的转义序列。

表3-1：Python的转义序列

转义字符	解析为
\	newline Ignored忽略换行符
\\	Backslash反斜杠
\'	Single quote单引号
\"	Double quote双引号
\a	ASCII Bell响铃

www.TopSage.com

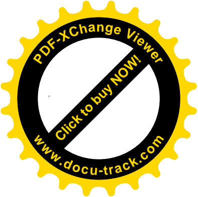
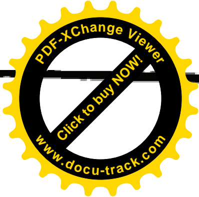



表3-1：Python的转义序列（续）

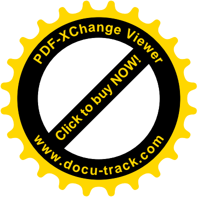
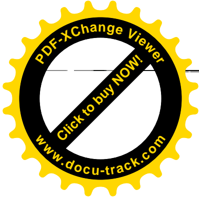
转义字符	解析为
\b	ASCII backspace退格
\f	ASCII form feed表格换行
\n	ASCII line feed换行
\N{name}	Named character in Unicode database (Unicode strings only)Unicode数据库中命名的字符（仅Unicode字符串）
\r	ASCII carriage return回车
\t	ASCII horizontal tab水平制表符
\uxxxx	Character with 16-bit hex value xxxx (Unicode only)16位十六进制值表示的字符（仅Unicode）
\Uxxxxxxxx	Character with 32-bit hex value xxxx (Unicode only)32位十六进制值表示的字符（仅Unicode）
\v	ASCII vertical tab垂直制表符
\ooo	Character with octal value oo八进制值表示的字符
\xhh	Character with hex value hh十六进制值表示的字符

转义序列的原始字符串方便记忆，尤其是在处理正则表达式的时候，这在本章的后面将进行介绍。例3-3显示了在原始字符串中转义序列是如何使用的。

例3-3：转义序列与原始字符串

```
In [1]: s = '\t'

In [2]: s
Out[2]: '\t'

In [3]: print s


In [4]: s = r'\t'

In [5]: s
Out[5]: '\\t'

In [6]: print s
\t

In [7]: s = '''\t'''

In [8]: s
Out[8]: '\t'

In [9]: print s

In [10]: s = r'''\t'''
```

www.TopSage.com

```
In [11]: s
Out[11]: '\\t'

In [12]: print s
\t

In [13]: s = r'\''

In [14]: s
Out[14]: "\\'"

In [15]: print s
\'
```

当转义序列被解析时，“\t”表示tab。当转义序列不被解析时，“\t”只是简单的一个由两个字符“\”和“t”组成的字符串。对于使用单引号、双引号或三引号创建的字符串，可以将“\t”解析为tab字符。而同样的字符串如果以r开头，可以将“\t”解析为“\”和“t”两个字符。

从这个示例中可以看到的另外一个有趣的事情就是__repr__与__str__之间的差别。如果在IPython提示符下输入一个变量名，然后按回车键，它的__repr__表示将显示出来。如果输入print加变量名，再按回车后，它的__str__表示将被显示出来。print函数解析字符串中的转义序列，并合理显示其内容。如果想看到对__repr__和__str__更多的介绍，请查阅第2章中的相关内容。

对于str进行数据提取的内建方法

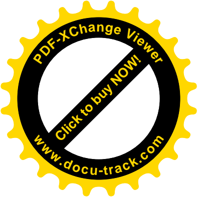
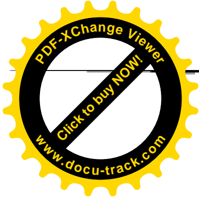
字符串是对象，提供了可以被调用以执行操作的方法。但是提及方法，不能仅限于str对象类型提供给我们使用的方法，还应包括所有可以用于从字符串对象中提取数据的可用的方法。这包括所有的str方法，也包括前一个示例中使用的in和not in文本操作。

从技术上讲，在例3-1中，in和not in调用了str对象的一个方法__contains__()。要了解更多的详细信息，请参见附录。可以使用in和not in来检查一个字符串是否是另一个字符串的一部分。参见例3-4。

例3-4：In和not in

```
In [1]: import subprocess

In [2]: res = subprocess.Popen(['uname', '-sv'], stdout=subprocess.PIPE)

In [3]: uname = res.stdout.read().strip()

In [4]: uname

Out[4]: 'Linux #1 SMP Tue Feb 12 02:46:46 UTC 2008'

In [5]: 'Linux' in uname

Out[5]: True
```


www.TopSage.com

```
In [6]: 'Darwin' in uname
Out[6]: False
In [7]: 'Linux' not in uname
Out[7]: False
In [8]: 'Darwin' not in uname
Out[8]: True
```

如果string2包含string1，则“string1 in string2”返回的值为真，否则返回的值为假。因此在检查"Linux"是否包含在uname所表示的字符串中时，返回值为真，而当检查"Darwin"是否包含在uname所表示的字符串中时，返回值为假。接下来我们演示了not in的有趣之处。

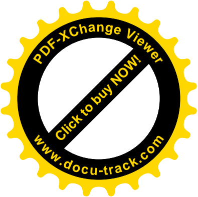
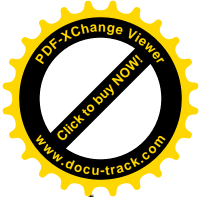
有时仅仅需要知道一个字符串是否是另一个字符串的子串。但有时还需要知道子字符串出现的具体位置。使用find()和index()可以实现这一目的。参见例3-5。

例3-5：find()和index()

```
In [9]: uname.index('Linux')
Out[9]: 0
In [10]: uname.find('Linux')
Out[10]: 0
In [11]: uname.index('Darwin')
---------------------------------------------------------------------------
<type 'exceptions.ValueError'>            Traceback (most recent call last)
/home/jmjones/code/<ipython console> in <module>()
<type 'exceptions.ValueError'>: substring not found
In [12]: uname.find('Darwin')
Out[12]: -1
```

如果string1在string2中（正如先前看到的示例），string2.find(string1)将返回string1第一个字符的索引，否则，返回-1，（别着急，我们马上会介绍有关索引的内容。）如果string1包括在string2中，string2.index(string1) 将返回string1的第一个字符的索引，否则它会抛出一个ValueError异常。在示例中，find()方法在字符串开始处找到字符串"Linux"，所示返回值为0，表示"Linux"的第一个字符的索引为0。然而，find()方法无法找到"Darwin"，因此返回值为-1。当Python搜索"Linux"时，index()方法与find()方法表现类似。但是，当查找"Darwin"时，index()抛出一个ValueError异常，表示它无法找到那个字符串。

www.TopSage.com

如何处理索引号呢？使用它们有什么好处呢？字符串被理解为一系列字符。find()和index()返回的索引表明从较长字符串中的哪一个字符开始匹配子字符串。参见例3-6。

例3-6：字符串切分

```
In [13]: smp_index = uname.index('SMP')

In [14]: smp_index

Out[14]: 9

In [15]: uname[smp_index:]

Out[15]: 'SMP Tue Feb 12 02:46:46 UTC 2008'

In [16]: uname[:smp_index]

Out[16]: 'Linux #1 '

In [17]: uname

Out[17]: 'Linux #1 SMP Tue Feb 12 02:46:46 UTC 2008'
```

使用字符串切分语法“string[index:]”，可以查找到从SMP开始到字符串结尾的每一个字符。使用语法string[:index]，我们可以看到从uname字符串起始位置到找到SMP所在位置的所有字符。两者之间仅有的差别在于冒号（:）在索引的左边或是右边。

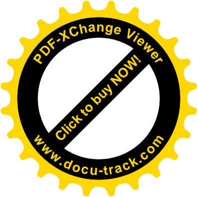
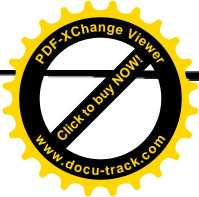
上述字符串切分示例，以及in与not in检测示例主要是为了向你展示字符串是一个序列，因此可以像处理列表这样的序列结构一样进行处理。如果希望查看更多对序列处理相关的讨论，参见由alex Martelli编著的《Python in a Nutshell (O'Reilly)》第4章序列操作，也可以在线访问*http://safari.oreilly.com/0596100469/pythonian-CHP-4-SECT-6*。

另外两个或许会用到的方法是startswith()和endswith()。正如方法名所表示的，这两个方法可以帮助你判断字符串是否以某一特定子串开始，或是以某一特定子串结束。参见例3-7：

例3-7：startswith()和endswith()

```
In [1]: some_string = "Raymond Luxury-Yacht"

In [2]: some_string.startswith("Raymond")
Out[2]: True

In [3]: some_string.startswith("Throatwarbler")
Out[3]: False

In [4]: some_string.endswith("Luxury-Yacht")
Out[4]: True

In [5]: some_string.endswith("Mangrove")
Out[5]: False
```

www.TopSage.com



PDF-XChange Viewer Click to buy NOW! www.docu-track.com

可以看到Python返回的信息：字符串“Raymond Luxury-Yacht”以“Raymond”开始，以“Luxury-Yacht.”结束。而不是以“Throatwarbler,”开始，也不是以“Mangrove.”结束。如果使用上文介绍的切分方法，也可以简单地获得相同的结果，但是切分方法使用起来有一些麻烦和枯燥。参见例3-8。

例3-8：使用切分技术实现Startswith()和endswith()功能

```
In [6]: some_string[:len("Raymond")] == "Raymond"
Out[6]: True
In [7]: some_string[:len("Throatwarbler")] == "Throatwarbler"
Out[7]: False
In [8]: some_string[-len("Luxury-Yacht"):] == "Luxury-Yacht"
Out[8]: True
In [9]: some_string[-len("Mangrove"):] == "Mangrove"
Out[9]: False
```

注意： 切分操作可以创建并返回一个新的字符串对象，而不是在行内对字符串进行修改。脚本中切分一个字符串的频率，会对内存和性能有明显的影响。即使没有明显的性能影响，在使用startwith()及endswith()可以满足需要的情况下，应避免使用切分操作。

可以看到，通过切分，尽管其中有多个字符，字符串“Paymond”还是出现在了some_string的开始。换句话说，我们可以看到some_string以字符串“Raymond”开始，而没有使用startswith()方法。在结尾的“Luxury-Yacht.”也是如此。

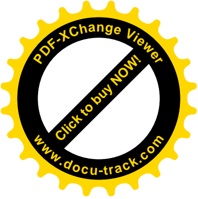
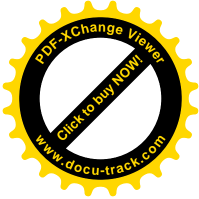
如果不带任何参数，lstrip()、rstrip()和strip()分别是用来删除前导空白，结尾空白，和前后空白的方法。空白可以包括tab、空格、回车和换行。不带参数使用lstrip()可以删除在字符串开始处出现的空白，并把去除空白的字符串作为一个新的字符串返回。使用不带参数的rstrip()可以删除出现在字符串结尾的空白，并把去除空白的字符串作为一个新的字符串返回。不带参数使用strip()可以删除在字符串开始及结尾的所有空白，并返回一个新字符串。参见例3-9。

注意： 所有的strip()方法创建并返回新的字符串对象，而不是对字符串进行行内修改。这或许根本不是什么问题，但是却是值得引起注意的地方。

例3-9：lstrip()、rstrip()和strip()

```
In [1]: spacious_string = "\n\t Some Non-Spacious Text\n \t\r"

In [2]: spacious_string
Out[2]: '\n\t Some Non-Spacious Text\n \t\r'

In [3]: print spacious_string

        Some Non-Spacious Text
```

www.TopSage.com

```
In [4]: spacious_string.lstrip()
Out[4]: 'Some Non-Spacious Text\n \t\r'

In [5]: print spacious_string.lstrip()
Some Non-Spacious Text


In [6]: spacious_string.rstrip()
Out[6]: '\n\t Some Non-Spacious Text'

In [7]: print spacious_string.rstrip()

        Some Non-Spacious Text

In [8]: spacious_string.strip()
Out[8]: 'Some Non-Spacious Text'

In [9]: print spacious_string.strip()
Some Non-Spacious Text
```

但是，strip()、rstrip()以及lstrip()都有一个可选参数：待除去的字符所组成的字符串。这意味着strip()等方法不只删除空白，还可以根据需要删除任何内容。

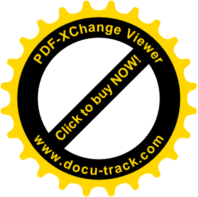
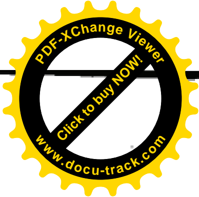
```
In [1]: xml_tag = "<some_tag>"

In [2]: xml_tag.lstrip("<")

Out[2]: 'some_tag>'

In [3]: xml_tag.lstrip(">")

Out[3]: '<some_tag>'

In [4]: xml_tag.rstrip(">")

Out[4]: '<some_tag'

In [5]: xml_tag.rstrip("<")

Out[5]: '<some_tag>'
```

以下示例演示如何依次去除XML标签的左尖括号和右尖括号。但是如果想同时删除左右尖括号时，该如何解决呢？可以这样来操作：

```
In [6]: xml_tag.strip("<").strip(">")

Out[6]: 'some_tag'
```

既然strip()等方法可以返回一个字符串，就可以在strip()调用之后直接调用另一个字符串操作。这里，我们将strip()调用连续使用。第一个strip()调用去除起始字符（左尖括号）并且返回一个字符串，第二个strip()函数删除末尾字符（右尖括号）并且返回字符串"some_tag"。但是这里有一个更为简单的方法：

www.TopSage.com

```
In [7]: xml_tag.strip("<>")
Out[7]: 'some_tag'
```

你可能设想strip()能够按照你所输入的字符进行准确删除，但是实际上，strip()将从字符串的适当一侧开始删除这些指定字符的任意顺序组合。在上面的示例中，我们告诉strip()删除“<>”。这不意味着将删除准确匹配的“<>”，也就是左尖括号在前右尖括号在后的组合，而是可以删除任何“<”和“>”的组合。

下面是一个比较清楚的示例：

```
In [8]: gt_lt_str = "<><><>gt lt str<><><>"
In [9]: gt_lt_str.strip("<>")
Out[9]: 'gt lt str'
In [10]: gt_lt_str.strip("><")
Out[10]: 'gt lt str'
```

我们去除了在字符串两边出现的“<”或“>”，只保留了字母和空格。

下面或许仍不是你希望的结果。例如：

```
In [11]: foo_str = "<fooooooo>blah<foo>"
In [12]: foo_str.strip("<foo>")
Out[12]: 'blah'
```

你可能希望strip()可以按照输入字符的顺序进行匹配删除，但是它并不是按照“<”、“f”、“o”和“>”的顺序进行匹配的。它将删除字符串中包含的全部4个字符，绝不会漏下一个“o”。下面是另一个strip()示例，可以清晰的看到这一点：

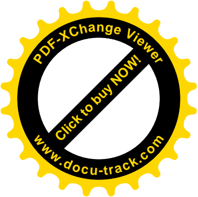
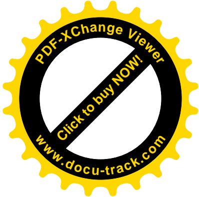
```
In [13]: foo_str.strip("><of")
Out[13]: 'blah'
```

在这个示例中，strip()同样也删除了“<”、“f”和“o”，尽管字符根本不是按这个顺排列的。

upper()方法和lower()方法非常有用，尤其是在需要对两个字符串进行比较，并且不考虑字符是大写或是小写时。upper()方法返回一个字符串，该字符串是大写的原始字符串的大写。lower()方法返回一个字符串，该字符串是小写的原始字符串（参见例3-10）。

例3-10：upper()和lower()

```
In [1]: mixed_case_string = "VOrpal BUnny"
```

```
In [2]: mixed_case_string == "vorpal bunny"
Out[2]: False

In [3]: mixed_case_string.lower() == "vorpal bunny"
Out[3]: True

In [4]: mixed_case_string == "VORPAL BUNNY"
Out[4]: False

In [5]: mixed_case_string.upper() == "VORPAL BUNNY"
Out[5]: True

In [6]: mixed_case_string.upper()
Out[6]: 'VORPAL BUNNY'

In [7]: mixed_case_string.lower()
Out[7]: 'vorpal bunny'
```

如果需要根据某个指定的分隔符对一个字符串进行提取，split()方法正好可以完成这类任务。参见例3-11。

例3-11：split()

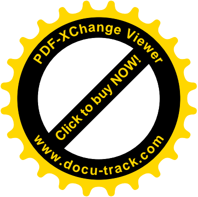
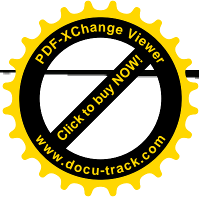
```
In [1]: comma_delim_string = "pos1,pos2,pos3"

In [2]: pipe_delim_string = "pipepos1|pipepos2|pipepos3"

In [3]: comma_delim_string.split(',')
Out[3]: ['pos1', 'pos2', 'pos3']

In [4]: pipe_delim_string.split('|')
Out[4]: ['pipepos1', 'pipepos2', 'pipepos3']
```

split()方法的典型用法是把希望作为分割符的分割的字符串作为参数传递给它。通常，分隔符是单个字管道符，例如逗号或符，但是也可以是多于一个字符的字符串。我们能够通过split()函数，以逗号分割comma_delim_string，以管道符（|）分割pipe_delim_ string，只需要将逗号和管道符分别传递给split()函数即可。split()的返回值是一系列字符串，每一个都是一组位于两个指定的分隔符之间的连续字符。当你需要的分割符是多个连续的字符，而不是单个字符时，split()方法也提供了支持。在我们写这本书的时候，Python中还没有字符类型，因此，虽然在两种情况下我们传递给split()方法的都只是单个字符，但实际上仍是一个字符串。因此在传递给split()多个字符时，它仍会正常工作。参见例3-12。

例3-12：split()多定界符（delimiter）示例

```
In [1]: multi_delim_string = "pos1XXXpos2XXXpos3"

In [2]: multi_delim_string.split("XXX")
Out[2]: ['pos1', 'pos2', 'pos3']

In [3]: multi_delim_string.split("XX")
Out[3]: ['pos1', 'Xpos2', 'Xpos3']
```


www.TopSage.com

```
In [4]: multi_delim_string.split("X")
Out[4]: ['pos1', '', '', 'pos2', '', '', 'pos3']
```

注意，首先为multi_delim_string定义"XXX"作为定界字符串。正如我们所期望的，返回结果为['pos1', 'pos2', 'pos3']。接下来定义"XX"作为定界字符串，split()返回['pos1', 'Xpos2', 'Xpos3']。split()会对出现在每对定界字符串"XX"之间的字符进行查找。"Pos1"从字符串起始位置，到第一个"XX"定界符；"Xpos2"出现在从第一次出现"XX"到第二次出现之间。最后一个split()使用了单个字符"X"作为定界字符串。值得注意的是，针对有两个紧临的"X"字符的位置，在返回列表里有一个空字符串""。这意味着在字符"X"之间没有字符。

但是，如果只是想在指定定界符第一次出现字符"n"的位置对字符串进行分割，那该怎么办呢？split()使用称为max_split的第2个参数。当一个整数值作为max_split被传递进来，split()仅对字符串分割由max_split指定的次数。

```
In [1]: two_field_string = "8675309,This is a freeform, plain text, string"

In [2]: two_field_string.split(',', 1)
Out[2]: ['8675309', 'This is a freeform, plain text, string']
```

我们根据逗号对字符串进行分割，并且仅对第一次出现的逗号定界符进行分割。虽然在这个示例中有许多逗号出现，字符串仅依据第一个进行分割。

如果想顺序去除诗歌等文本中的空格，split()是一个非常好的工具：

```
In [1]: prosaic_string = "Insert your clever little piece of text here."

In [2]: prosaic_string.split()
Out[2]: ['Insert', 'your', 'clever', 'little', 'piece', 'of', 'text', 'here.']
```

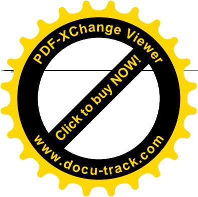
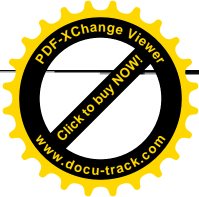
因为没有传递参数，split()将空格默认为分隔符。

绝大多数时候，使用split会看到期望的结果。然而，如果遇到多行文本，或许会得不到所期望的结果。通常，在处理多行文本时，需要一次处理一行。但是你或许会发现split会对字符串中的每一个词进行分割：

```
In [1]: multiline_string = """This
   ...: is
   ...: a multiline
   ...: piece of
   ...: text"""

In [2]: multiline_string.split()
Out[2]: ['This', 'is', 'a', 'multiline', 'piece', 'of', 'text']
```

在这个示例中，使用splitlines()会获得你更想得到的结果：

www.TopSage.com

```
In [3]: lines = multiline_string.splitlines()

In [4]: lines

Out[4]: ['This', 'is', 'a multiline', 'piece of', 'text']
```

splitlines()返回一个由字符串中的每一行所组成的列表，并且保存为一组。从这里可以看到，你可以迭代每一行，并将每一行内容分割为单词：

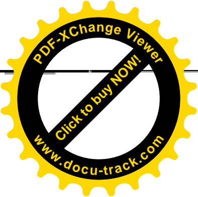
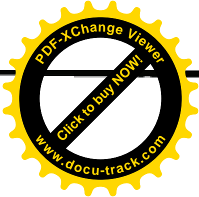
```
In [5]: for line in lines:
   ...:     print "START LINE::"
   ...:     print line.split()
   ...:     print "::END LINE"
   ...:
START LINE::
['This']
::END LINE
START LINE::
['is']
::END LINE
START LINE::
['a', 'multiline']
::END LINE
START LINE::
['piece', 'of']
::END LINE
START LINE::
['text']
::END LINE
```

有时，并不想分割字符串或从字符串中提取信息，而是希望将多个字符串连接到一起。在这种情况下，join()可以帮助完成该工作：

```
In [1]: some_list = ['one', 'two', 'three', 'four']

In [2]: ','.join(some_list)
Out[2]: 'one,two,three,four'

In [3]: ', '.join(some_list)
Out[3]: 'one, two, three, four'

In [4]: '\t'.join(some_list)
Out[4]: 'one\ttwo\tthree\tfour'

In [5]: ''.join(some_list)
Out[5]: 'onetwothreefour'
```

例如指定一个列表some_list，就可以将字符串“one”、“two”、“three”和“four”组合到一些变量中。我们使用一个逗号，再一个逗号，然后是一个空格和一个tab将列表some_list进行连接。join()是字符串方法，因此对于一个类似“,”的字符串，调用join()是有效的。join()会采用一连串的多个字符串作为参数。它将多个字符串

www.TopSage.com

串压缩成单个字符串，这样列表中的每个字符串将按顺序排列，但是调用join()的字符串会在序列中每一项前出现。

我们对join()和参数的使用有一些建议。注意，join()需要一个字符串序列。如果你给join()传递的是一系列整数，那会出现什么情况呢？

```
In [1]: some_list = range(10)

In [2]: some_list
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: ",".join(some_list)
---------------------------------------------------------------------------
exceptions.TypeError                      Traceback (most recent call last)

/Users/jmjones/<ipython console>

TypeError: sequence item 0: expected string, int found
```

join()抛出的异常追踪是自解释的，但是由于这是一个常见错误，值得引起注意。只需对列表稍加理解就可以很容易地避免这种问题的发生。这里列出了帮助信息，来帮助理解如何将some_list中所有整数元素转换为字符串：

```
In [4]: ",".join([str(i) for i in some_list])
Out[4]: '0,1,2,3,4,5,6,7,8,9'
```

也可以使用一个表达式：

```
In [5]: ",".join(str(i) for i in some_list)

Out[5]: '0,1,2,3,4,5,6,7,8,9'
```

关于列表的更多内容，可以参见《Python in a Nutshell》（也可以在线访问*http://safari.oreilly.com/0596100469/pythonian-CHP-4-SECT-10*）第4章“流程控制语句”。

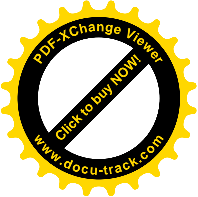
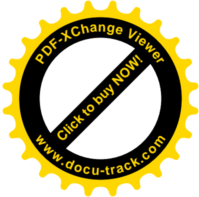
最后要介绍的是用于创建和修改文本字符的replace()方法。replace()有两个参数，分别是被替换的字符串以及替换字符串。下面是一个简单的replace()方法示例：

```
In [1]: replacable_string = "trancendental hibernational nation"

In [2]: replacable_string.replace("nation", "natty")
Out[2]: 'trancendental hibernattyal natty'
```

值得注意的是，replace()不关心替换字符串是在一个词的中间或者就是一个完整的单词。因此，在需要使用指定的字符序列去替换另一个字符序列时，replace()是一个非常好的工具。

然而，当需要用一个字符序列去替换另一个字符序列，但需要进行更为精确的控制时，这是不够的。有时需要指定一个字符模式来实现查找和替换。模式可以帮助实现对文本

www.TopSage.com

的搜索，从而实现数据的提取。在一些更适于使用模式的情况下，使用正则表达式是非常有帮助的。接下来我们会介绍正则表达式。

注意： 与切分操作以及strip()方法一样，replace()会创建一个新字符串，而不是对字符串进行行内修改。

Unicode字符串

到目前为止，所有查找字符串的示例全部都使用了内建的字符串类型（str），但是Python还有另一种你可能想要熟悉的字符串类型：Unicode。当你看到在计算机屏幕上显示的字符时，计算机正将其作为数字在内部正进行处理。根据语言和平台的不同，有许多种不同的数字-字符映射集。Unicode是一个标准，提供了数字-字符的单一映射，而无须考虑语言、平台或者是对文本进行处理的程序。在这一章，我们引入Unicode的概念并介绍Python如何对Unicode进行处理。要对Unicode作更深一步的了解，可以参阅A.M.Kuchling的非常不错的Unicode教程，网址是：*http://www.amk.ca/python/howto/unicode*。

创建一个Unicode字符串与创建一个普通的字符串一样简单：

```
In [1]: unicode_string = u'this is a unicode string'

In [2]: unicode_string
Out[2]: u'this is a unicode string'

In [3]: print unicode_string
this is a unicode string
```

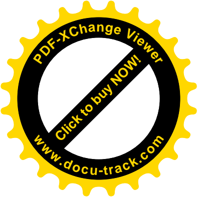
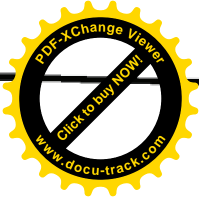
你也可以使用内建的unicode()函数：

```
In [4]: unicode('this is a unicode string')
Out[4]: u'this is a unicode string'
```

使用Unicode字符串看起来并不会给我们增加许多麻烦，尤其是仅对来自一种语言的字符进行处理的时候。但是当字符串包含多种语言的字符时该如何处理呢？Unicode会帮助你处理这种情况。可以使用指定的数值创建一个Unicode字符，你可以使用“\uXXX”或者是“\uXXXXXXXX”来表示。例如，下面是一个Unicode字符串，其中包括了拉丁、希腊和俄文字符：

```
In [1]: unicode_string = u'abc_\u03a0\u03a3\u03a9_\u0414\u0424\u042F'

In [2]: unicode_string
Out[2]: u'abc_\u03a0\u03a3\u03a9_\u0414\u0424\u042f'
```


www.TopSage.com

Python依据使用的编码产生字符串（str）。对于与Mac标准一致的Python，如果试图从之前的示例中打印字符串，将会返回一个错误，打印内容如下：

```
In [3]: print unicode_string
---------------------------------------------------------------------------
UnicodeEncodeError                        Traceback (most recent call last)

/Users/jmjones/<ipython console> in <module>()

UnicodeEncodeError: 'ascii' codec can't encode characters in position 4-6:
  ordinal not in range(128)
```

我们不得不告诉它所使用的编码，这样它就知道如何处理我们使用的字符：

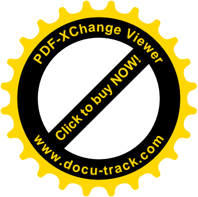
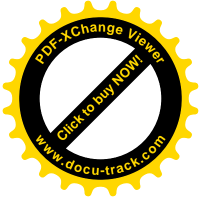
```
In [4]: print unicode_string.encode('utf-8')
abc_ΠΣΩ_ДФЯ
```

这里，我们采用UTF-8格式对包含拉丁、希腊以及俄文字符的字符串进行编码，这是Unicode数据的常用编码。

Unicode字符串包含了许多功能，其中包括在in的示例中已经提到的正则表达式的用法。

```
In [5]: u'abc' in unicode_string
Out[5]: True

In [6]: u'foo' in unicode_string
Out[6]: False

In [7]: unicode_string.split()
Out[7]: [u'abc_\u03a0\u03a3\u03a9_\u0414\u0424\u042f']

In [8]: unicode_string.
unicode_string.__add__              unicode_string.expandtabs
unicode_string.__class__            unicode_string.find
unicode_string.__contains__         unicode_string.index
unicode_string.__delattr__          unicode_string.isalnum
unicode_string.__doc__              unicode_string.isalpha
unicode_string.__eq__               unicode_string.isdecimal
unicode_string.__ge__               unicode_string.isdigit
unicode_string.__getattribute__     unicode_string.islower
unicode_string.__getitem__          unicode_string.isnumeric
unicode_string.__getnewargs__       unicode_string.isspace
unicode_string.__getslice__         unicode_string.istitle
unicode_string.__gt__               unicode_string.isupper
unicode_string.__hash__             unicode_string.join
unicode_string.__init__             unicode_string.ljust
unicode_string.__le__               unicode_string.lower
unicode_string.__len__              unicode_string.lstrip
unicode_string.__lt__               unicode_string.partition
unicode_string.__mod__              unicode_string.replace
unicode_string.__mul__              unicode_string.rfind
unicode_string.__ne__               unicode_string.rindex
unicode_string.__new__              unicode_string.rjust
unicode_string.__reduce__           unicode_string.rpartition
```

www.TopSage.com

```
unicode_string.__reduce_ex__      unicode_string.rsplit
unicode_string.__repr__           unicode_string.rstrip
unicode_string.__rmod__           unicode_string.split
unicode_string.__rmul__           unicode_string.splitlines
unicode_string.__setattr__        unicode_string.startswith
unicode_string.__str__            unicode_string.strip
unicode_string.capitalize         unicode_string.swapcase
unicode_string.center             unicode_string.title
unicode_string.count              unicode_string.translate
unicode_string.decode             unicode_string.upper
unicode_string.encode             unicode_string.zfill
unicode_string.endswith
```

你或许现在不需要Unicode。但是如果你希望能够一直使用Python编程，熟悉Unicode是非常必要的。

re

既然Python是一个连“电池都包括在内”的语言，你或许会希望Python也应包括一个正则表达式库。这一点不会让你失望的。本节讲述的重点是如何在Python中使用正则表达式，而不是正则表达式的in和out语法。因此，如果对正则表达式不熟悉，我们建议你阅读由Jeffrey E. F. Friedl编著的《Mastering Regular Expressions》（O'Reilly出版，你也可以访问*http://safari.oreilly.com/0596528124*）。本节假定你已经掌握了正则表达式，但是如果还没有掌握，不妨在手边准备一本Friedl的这本著作，这将是非常有帮助的。

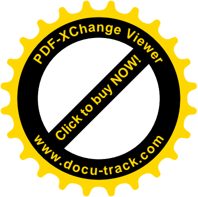
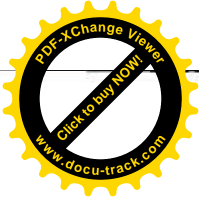
如果对Perl比较熟悉，你很可能习惯于通过“=~”来使用正则表达式。Python所包括的正则表达式来自于库，而不是语言自身的语法特征。因此，为了使用正则表达式，必须首先载入正则表达式模块re。下面是一个基本的示例，展示了正则表达式的创建和使用。参见例3-13。

例3-13：基本正则表达式的使用

```
In [1]: import re

In [2]: re_string = "{{(.*?)}}"

In [3]: some_string = "this is a string with {{words}} embedded in\
   ...: {{curly brackets}} to show an {{example}} of {{regular expressions}}"

In [4]: for match in re.findall(re_string, some_string):
   ...:     print "MATCH->", match
   ...:
MATCH-> words
MATCH-> curly brackets
MATCH-> example
MATCH-> regular expressions
```

我们做的第一件事情是加载re模块。或许正如你猜想的那样，re代表了“正则表达式”。接下来创建一个字符串re_string，这将是我们在示例中进行查找操作所使用的

www.TopSage.com

模式。该模式匹配两个连续的左大括号（{{），然后是任意文本（也可以为空），最后是两个连续的右大括号（}}）。然后，我们创建了一个字符串some_string，该字符串包括由大括号包括起来的一组单词以及没有被大括号包括起来的单词。最后，我们循环使用re模块中的findall()函数对匹配结果进行处理，findall()将根据模式re_string搜索some_string字符串。正如你所看到的，输出结果中包括words、curly brackets，example以及regular expressions，这正是双大括号中的所包含的内容。

在Python中有两种使用正则表达式的方式。第一种是直接使用re模块中的函数，正如上例中所演示的那样。第二种是创建一个它编译的正则表达式对象，然后使用对象中的方法。

什么是已编译的正则表达式呢？它是一个简单的对象，通过传递一个模式给re.compile()来创建；它包括一些正则表达式方法，也通过传递模式给re.compile()来创建。在使用编译与非编译的示例中存在两个主要的差别。首先，没有继续引用正则表达式模式"{{(.*?)}}"，而是创建了一个编译的正则表达式对象，并且是使用模式来创建。第二，没有在re模块上调用findall()，而是在编译后的正则表达式对象上调用findall()。

关于re模块内容的更多信息，包括可用函数等，参见《Python Library Reference》的Module Contents相关内容，*http://docs.python.org/lib/node46.html*。更多的关于编译后正则表达式对象的内容，可参见《Python Library Reference》的Regular Expression Objects相关内容，网络链接地址是：*http://docs.python.org/lib/re-objects.html*。

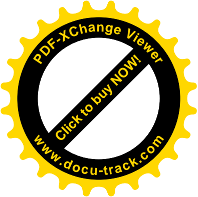
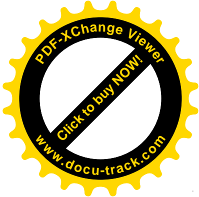
例3-14演示了双大括号示例，并展示了如何使用一个编译后的正则表达式对象。

例3-14：简单正则表达式，已编译模式

```
In [1]: import re

In [2]: re_obj = re.compile("{{(.*?)}}")

In [3]: some_string = "this is a string with {{words}} embedded in\
   ...:  {{curly brackets}} to show an {{example}} of {{regular expressions}}"

In [4]: for match in re_obj.findall(some_string):
   ...:     print "MATCH->", match
   ...:
MATCH-> words
MATCH-> curly brackets
MATCH-> example
MATCH-> regular expressions
```

在Python中，你可以根据个人喜好选择使用正则表达式的方法。然而，选用不同的方法会对执行的性能产生影响如果你在某些循环中多次重复某一操作，例如在一个循环中将正则表达式应用于一个有数百万行文本的文件的每一行，性能问题便会变得明显起来。

www.TopSage.com

在以下的示例中，我们运行了一个简单的包括正则表达式的脚本，脚本中使用了编译和非编译的正则表达式，处理的文件包括50万行文本。我们运行Unix下的timeit工具对脚本执行情况进行检测，使你能够看到性能的差异。参见例3-15。

例3-15：re非编译代码性能检测

```
#!/usr/bin/env python

import re

def run_re():
    pattern = 'pDq'

    infile = open('large_re_file.txt', 'r')
    match_count = 0
    lines = 0
    for line in infile:
        match = re.search(pattern, line)
        if match:
            match_count += 1
        lines += 1
    return (lines, match_count)

if __name__ == "__main__":
    lines, match_count = run_re()
    print 'LINES::', lines
    print 'MATCHES::', match_count
```

timeit工具执行一段代码数次，然后报告最佳运行所花费的时间。下面是在IPytnon中运行Python的timeit工具来执行这段代码结果：

```
In [1]: import re_loop_nocompile

In [2]: timeit -n 5 re_loop_nocompile.run_re()
5 loops, best of 3: 1.93 s per loop
```

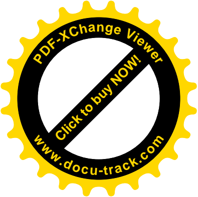
该示例执行run_re()函数5次，并且报告最佳运行花费了1.93秒/次。timeit在一段时间内重复运行一段相同的代码的原因，是为了减少其他在同一运行时间运行的进程对测试结果的影响。

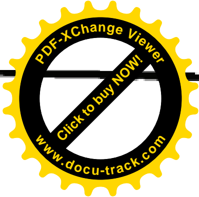
以下是使用Unix的time工具对相同代码的测试结果：

```
jmjones@dink:~/code$ time python re_loop_nocompile.py
LINES:: 500000 MATCHES:: 242

real 0m2.113s
user 0m1.888s
sys 0m0.163s
```

例3-16是相同的正则表达式示例，只是这里使用re.compile()来创建一个编译的模式对象。

www.TopSage.com





例3-16：re编译代码性能检测

```
#!/usr/bin/env python

import re

def run_re():
    pattern = 'pDq'
    re_obj = re.compile(pattern)

    infile = open('large_re_file.txt', 'r')
    match_count = 0
    lines = 0
    for line in infile:
        match = re_obj.search(line)
        if match:
            match_count += 1
        lines += 1
    return (lines, match_count)

if __name__ == "__main__":
    lines, match_count = run_re()
    print 'LINES::', lines
    print 'MATCHES::', match_count
```

在IPython中使用Python的timeit工具运行该脚本，所产生的结果如下：

```
In [3]: import re_loop_compile

In [4]: timeit -n 5 re_loop_compile.run_re()
5 loops, best of 3: 860 ms per loop
```

使用Unix的time工具运行相同的脚本，所产生的结果如下：

```
jmjones@dink:~/code$ time python
re_loop_compile.py LINES:: 500000 MATCHES:: 242

real 0m0.996s
user 0m0.836s
sys 0m0.154s
```

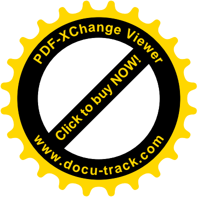
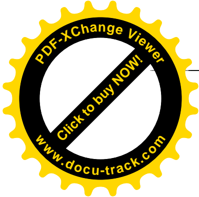
很明显，编译版本更为优越。正如由Unix的time和Python的timeit工具所测量的结果所表明的，它只花费了一半的运行时间。因此，我们强烈建议养成创建编译后的正则表达式的习惯。

正如在这一章前面部分所讨论的，原始字符串可以用于表示不对转义序列进行解析的字符串。例3-17显示了在正则表达式中使用的原始字符串。

例3-17：原始字符串与正则表达式

```
In [1]: import re

In [2]: raw_pattern = r'\b[a-z]+\b'
```

www.TopSage.com

```
In [3]: non_raw_pattern = '\b[a-z]+\b'

In [4]: some_string = 'a few little words'

In [5]: re.findall(raw_pattern, some_string)

Out[5]: ['a', 'few', 'little', 'words']

In [6]: re.findall(non_raw_pattern, some_string)

Out[6]: []
```

正则表达式模式“\b”匹配词边界。因此在原始（raw）及普通字符串中，我们寻找到了单个的小写单词。值得注意的是，raw_patern匹配了在some_string中合适的单词边界，而non_raw_pattern根本没有任何匹配。raw_pattern将“\b”识别为两个字符，而不是解析为转义字符中的退格字符。non_raw_patern则将“\b”解析为转义字符中的退格字符。正则表达式函数findall()可以使用原始字符串模式来查找单词。但是，当findall()使用非原始字符串模式进行搜索时，不会找到任何退格字符。

对于使用non_raw_pattern匹配字符串的问题，正如下面对单词“little”所做的操作一样，我们在其前后放置退格字符：

```
In [7]: some_other_string = 'a few \blittle\b words'

In [8]: re.findall(non_raw_pattern, some_other_string)
Out[8]: ['\x08little\x08']
```

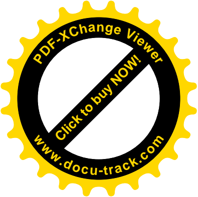
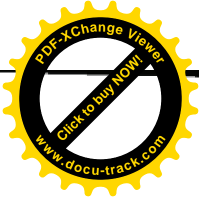
值得注意的是findall()函数在单词“little”之前和之后匹配了十六进制符号“\x08”。该十六进制符号对应于退格字符，即使用转义字符"\b"插入的字符。

因此，正如你所看到的，如果希望使用这种反斜杠加指定字符的方法（例如"\b"可以作为单词的边界，"\d"表示数字，"\w"表示希腊数字字符），原始字符串是非常有用的。如果想查看完整的由反斜杠定义的转义字符列表，参见《Python Library Reference》（*http://docs.python.org/lib/re-syntax.html*）中的正则表达式语法一节。

从例3-14到例3-17，无论是正则表达式，还是我们应用的不同方法都相当简单。有时，对正则表达式强大功能的这种有限使用就能满足我们的需要了。其他时候，你需要更多地利用包含在正则表达式库中的强大功能。

四个主要的也是最经常使用的正则表达式方法（或者称为函数）有findall()，finditer()，match()和search()。你或许也经常使用split()或是sub()，但很可能还是比不上使用其他那几个函数那样频繁。findall()可以找到搜索字符串中指定模式的所有匹配。findall()匹配模式返回的数据结构类型将取决于模式是否定义了一个组。

www.TopSage.com

注意： 关于正则表达式的一个快捷提示：分组允许你在一个正则表达式中定义需要提取的文本。要获得更多信息，请参见Friedl编著的《Mastering RegularExpressions》中"通用元字符与域"的相关内容，或在线访问以下网址：*http://safari.oreilly.com/0596528124/regex3-CHP-3-SECT-5?imagepage=137*。

如果没有在正则表达式模式中定义组，但是却找到了匹配，findall()将返回一个字符串列表。例如：

```
In [1]: import re

In [2]: re_obj = re.compile(r'\bt.*?e\b')

In [3]: re_obj.findall("time tame tune tint tire")
Out[3]: ['time', 'tame', 'tune', 'tint tire']
```

模式没有定义任何组，因此findall()返回一个字符串列表。有趣的是返回列表的最后一个元素包含两个单词：tint和tire。正则表达式希望匹配以"t"开始的单词和以"e"结尾的单词。但是命令".*?"匹配所有字符包括空白。findall()搜索所有可能的匹配。它找到一个以"t"开始的单词（tint），然后继续查找字符串，直到找到一个以"e"结尾的单词（tire）。因此，所匹配的"tint tire"是正确的。如果希望去除空白，你可以使用"r'\bt\w*e\b'"：

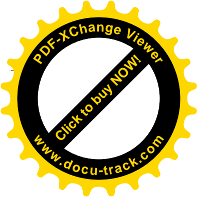
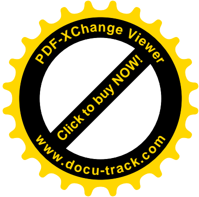
```
In [4]: re_obj = re.compile(r'\bt\w*e\b')

In [5]: re_obj.findall("time tame tune tint tire")
Out[5]: ['time', 'tame', 'tune', 'tire']
```

第二个可能被返回的数据结构类型是一个多元列表。如果定义了一个组，并且其中有一个匹配，那么findall()返回一个多元列表。例如，例3-18是一个使用这种模式和字符串的简单示例。

例3-18：findall()简单分组

```
In [1]: import re

In [2]: re_obj = re.compile(r"""(A\W+\b(big|small)\b\W+\b
   ...: (brown|purple)\b\W+\b(cow|dog)\b\W+\b(ran|jumped)\b\W+\b
   ...: (to|down)\b\W+\b(the)\b\W+\b(street|moon).*?\.)""",
   ...: re.VERBOSE)

In [3]: re_obj.findall('A big brown dog ran down the street. \
   ...: A small purple cow jumped to the moon.')

Out[3]:
[('A big brown dog ran down the street.',
  'big',
  'brown',
  'dog',
  'ran',
```

www.TopSage.com

```
  'down',
  'the',
  'street'),
 ('A small purple cow jumped to the moon.',
  'small',
  'purple',
  'cow',
  'jumped',
  'to',
  'the',
  'moon')]
```

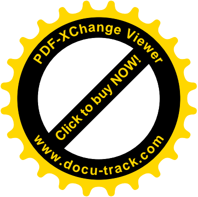
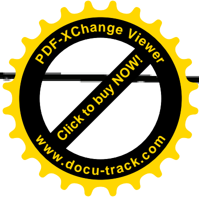
尽管非常简单，该示例还是展示了一些重要内容。首先值得注意的是，这个简单模式比较长，而且包括了许多非数字字符（如果你长时间盯着这些长字符串，眼睛都会充血）。这似乎是许多正则表达式的常用格式。其次，模式包括明确的组嵌套。外层组匹配以字母“A”开始到结尾的所有字符。在A和结束字符之间的所有字符组成了内层组，内层组匹配“big or small”、“brown or purple”等。接下来，findall()的返回值是一个多元列表。这个多元列表的每一个元素是我们在正则表达式中定义的组之一。整个语句是多元组的第一个元素，因此它是最大、最外层的组。每一个子组由多元元素序列组成。值得注意的是re.compile()的最后一个参数re.VERBOSE。它表示允许以冗余模式编写正则表达式字符串。这意味着可以非常简单地在整个行上分割正则表达式，而模式匹配不受分割的影响。在组之外的空白将被忽略掉。verbose允许在正则表达式每一行的结尾处插入注释，以记录每一个特殊的正则表达式都完成了什么，但是我们经常选择不去这样做。正则表达式的难点之一是模式的描述，你想要匹配的模式通常会变得巨大且很难阅读。re.VERBOSE函数允许写一个简单的正则表达式，这对于改善代码（包括正则表达式）的可维护性是一个非常不错的工具。

Finditer()是对findall()的略微修改，不同于findall()之处在于findall()所返回的是一个多元列表，而正如其名称所表明的，finditer()返回一个迭代。每一次迭代的元素都是一个正则表达式匹配的对象，这将在本章后面进行介绍。例3-19是一个使用finditer()而不是findall()的同样非常简单的示例。

例3-19：finditer() 示例

```
In [4]: re_iter = re_obj.finditer('A big brown dog ran down the street. \
   ...: A small purple cow jumped to the moon.')

In [5]: re_iter

Out[5]: <callable-iterator object at 0xa17ad0>

In [6]: for item in re_iter:
   ...: print item
   ...: print item.groups()
   ...:
<_sre.SRE_Match object at 0x9ff858>
```


www.TopSage.com

```
('A big brown dog ran down the street.', 'big', 'brown', 'dog', 'ran',
    'down', 'the', 'street')
<_sre.SRE_Match object at 0x9ff940>
('A small purple cow jumped to the moon.', 'small', 'purple', 'cow',
    'jumped', 'to', 'the', 'moon')
```

如果之前曾经接触过迭代器，你或许会把它想象成一个根据需要创建的列表。但这样认为是错误的。原因之一是在迭代器中不能通过索引来引用某一特定元素，但对于一个列表，却可以这样做，如some_list[3]。这一限制的后果是你不具备将迭代器进行分割的能力，而这在列表中是可以的，如some_list[2:6]。如果不考虑这个限制，迭代器可以说是轻量且功能强大的，尤其是仅需要迭代一些序列时。因为不需要将整个序列加载到内存，而是根据需要进行获取。因此，一个迭代器与其对应的列表相比，需要更小的内存。这也意味着迭代器在访问一个序列中的元素时，在很短的时间内就可以启动。

使用finditer()而不是findall()的另一个原因是finditer()的每一项都是一个匹配对象，而不是对应于匹配文本的简单的字符串列表或是多元列表。

match()与search()提供了相似的功能。它们对字符串应用一个正则表达式，定义开始搜索和结束搜索的位置，并且都返回一个首次匹配指定模式的匹配对象。两者之间的差异是match()从指定位置开始进行匹配，而不会移动到字符串的任何随机位置。但search()从指定的位置开始搜索，在指定的位置结束搜索。参见例3-20。

例3-20：比较match() 与search()

```
In [1]: import re

In [2]: re_obj = re.compile('FOO')

In [3]: search_string = ' FOO'

In [4]: re_obj.search(search_string)

Out[4]: <_sre.SRE_Match object at 0xa22f38>

In [5]: re_obj.match(search_string)

In [6]:
```

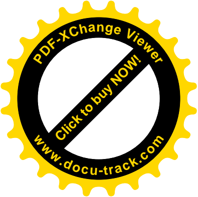
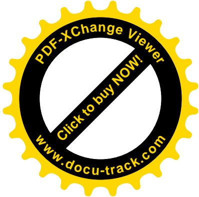
尽管search_string包括match()搜索的模式，但不能找到匹配。因为存在匹配的search_string子字符串无法在search_string起始位置开始查询。search()调用会创建一个匹配对象。

search()和match()调用接受开始和结束参数，这两个参数定义Python针对某一模式开始搜索的起始位置与结束位置。参见例3-21。

例3-21：search() 和match()的开始和结束

```
In [6]: re_obj.search(search_string, pos=1)
```

www.TopSage.com

```
Out[6]: <_sre.SRE_Match object at 0xabe030>

In [7]: re_obj.match(search_string, pos=1)

Out[7]: <_sre.SRE_Match object at 0xabe098>

In [8]: re_obj.search(search_string, pos=1, endpos=3)

In [9]: re_obj.match(search_string, pos=1, endpos=3)

In [10]:
```

参数pos是一个索引，该索引定义Python在字符串中寻找某一模式的位置。为search()定义起始参数pos不会改变任何事情，但是为match()定义pos参数会让其匹配在不使用pos时无法匹配模式。设置结束参数endpos为3将导致search()和match()都无法匹配模式，因为模式在第三个字符位置之后出现。

findall()和finditer()需要回答这样的问题："我的模式匹配什么？"。而search()和match()面临的主要的问题是"我的模式匹配吗？"。search()和match()还需要回答这个问题："我的模式第一次匹配的内容是什么？"。但是最经常的，也是你真正想知道的问题是"我的模式匹配吗？"。例如，假设你正在写一段代码来读取logfiles，并且为了显示效果更好，以HTML格式显示且在每一行自动换行。你想要所有的"ERROR"行以红色显示，因此你可以循环检查文件中的每一行，看它是否匹配正则表达式，并且如果search()在搜索中出现命中 "ERROR"的情况，你需要对该行进行格式化，用红色进行显示。

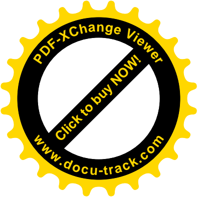
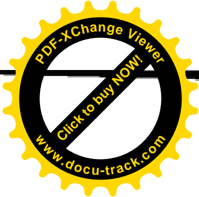
search()和match()是非常好的工具，因为它们不仅可以显示一个模式是否匹配一段文本，而且可以返回一个match()对象。match()对象包含当你搜索文本时各种各样的数据片段。start()、end()、span()、group()和groupdict()都是特别有趣的match()方法。

start()、end()和span()定义搜索字符串时模式匹配的开始和结束位置。start()返回一个整数，该整数代表字符串中模式开始进行匹配操作的位置。end()也返回一个整数，该整数代表模式匹配结束的位置。span()返回一个多元组，包括匹配的开始和结尾。

groups()返回一个匹配的多元组，每一元素都是模式所指定的组。这个元组与findall()返回的元组相似。groupdict()返回一个命名组字典，组的名字通过正则表达式自身的"?P<group_name>pattern"语法确定。

总之，为了有效地使用正则表达式，养成使用编译后的正则表达式对象的习惯是非常重要的。在希望查看模式匹配的文本是什么时，可以使用findall()和finditer()。记住，finditer()比findall()更具灵活性，因为它返回一个匹配对象的迭代。如果希望更详细地了解正则表达式库，参见由Alex Martelli编著的《Pythonin a Nutshell》

www.TopSage.com

（O'Reilly）第9章。如果希望查看一些正则表达式的实际应用，参见Greg Wilson编著的《DataCrunching》（The Pragmatic Bookshelf）。

Apache配置文件详解

在学习了Python的正则表达式之后，让我们进一步了解Apache的配置文件：

```
NameVirtualHost 127.0.0.1:80
<VirtualHost localhost:80>
    DocumentRoot /var/www/
    <Directory />
        Options FollowSymLinks
        AllowOverride None
    </Directory>
    ErrorLog /var/log/apache2/error.log
    LogLevel warn
    CustomLog /var/log/apache2/access.log combined
    ServerSignature On
</VirtualHost>
<VirtualHost local2:80>
    DocumentRoot /var/www2/
    <Directory />
        Options FollowSymLinks
        AllowOverride None
    </Directory>
    ErrorLog /var/log/apache2/error2.log
    LogLevel warn
    CustomLog /var/log/apache2/access2.log combined
    ServerSignature On
</VirtualHost>
```

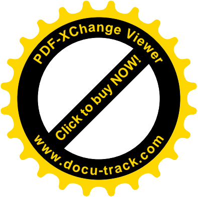
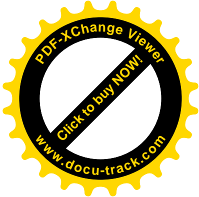
该文件是将安装在Ubuntu上的Apache2的配置文件略作修改得到的。我们创建了虚拟用户进行配置。修改*/etc/hosts*文件，加入如下所示的行：

```
127.0.0.1      local2
```

这允许在浏览器中输入local2，并且将其解析为127.0.0.1，这是一个本地主机地址。那么这么做的有什么作用呢？如果访问*http://local2*，浏览器会将主机名通过HTTP请求进行传递。下面是一个HTTP对local2的请求：

```
GET / HTTP/1.1
Host: local2
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.8.1.13)
Gecko/20080325 Ubuntu/7.10 (gutsy) Firefox/2.0.0.13
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

www.TopSage.com

```
If-Modified-Since: Tue, 15 Apr 2008 17:25:24 GMT
If-None-Match: "ac5ea-53-44aecaf804900"
Cache-Control: max-age=0
```

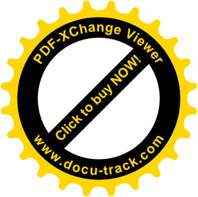
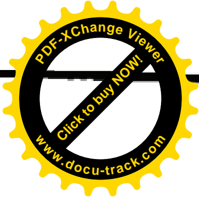
请注意以“Host:”开始的行。当Apache获得一个请求，它将其传递给匹配local2的虚拟主机。

因此，我们需要写一个脚本，解析Apache配置文件（类似于刚刚所演示的），找到VirtualHost部分，替换DocumentRoot。脚本如下所示：

```
#!/usr/bin/env python

from cStringIO import StringIO
import re

vhost_start = re.compile(r'<VirtualHost\s+(.*?)>')
vhost_end = re.compile(r'</VirtualHost')
docroot_re = re.compile(r'(DocumentRoot\s+)(\S+)')

def replace_docroot(conf_string, vhost, new_docroot):
    '''yield new lines of an httpd.conf file where docroot lines matching
        the specified vhost are replaced with the new_docroot
    '''
    conf_file = StringIO(conf_string)
    in_vhost = False
    curr_vhost = None
    for line in conf_file:
        vhost_start_match = vhost_start.search(line)
        if vhost_start_match:
            curr_vhost = vhost_start_match.groups()[0]
            in_vhost = True
        if in_vhost and (curr_vhost == vhost):
            docroot_match = docroot_re.search(line)
            if docroot_match:
                sub_line = docroot_re.sub(r'\1%s' % new_docroot, line)
                line = sub_line
        vhost_end_match = vhost_end.search(line)
        if vhost_end_match:
            in_vhost = False
        yield line
if __name__ == '__main__':
    import sys
    conf_file = sys.argv[1]
    vhost = sys.argv[2]
    docroot = sys.argv[3]
    conf_string = open(conf_file).read()
    for line in replace_docroot(conf_string, vhost, docroot):
        print line,
```

这个脚本先建立三个编译后的正则表达式对象：一个匹配VirtualHost的开始，一个匹配VirtualHost的结尾，一个匹配DocumentRoot这一行。我们创建了一个函数来完成繁杂的工作。函数被命名为replace_docroot()，其参数包括配置文件名，匹配的VirtualHost

www.TopSage.com

名和指向virtualHost的目录名DocumentRoot。该函数建立一个状态机，检测我们是否处于一个virtualHost部分，并保持对virtualHost的追踪。当处于调用代码定义的virtualHost中时，函数寻找DocumentRoot指令出现的位置，并且将指令所指向的目录更改为为调用代码定义的目录。当replace_docroot()迭代配置文件中的每一行时，将产生没有修改的输入行或修改过的DocumentRoot行。

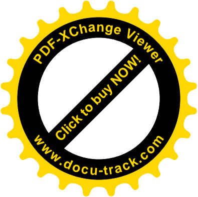
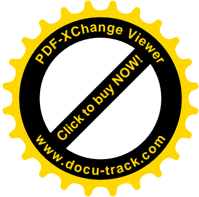
我们为此函数创建了一个简单的命令行接口。使用optparse进行实现没什么可奇怪的，对给定的参数数目进行错误检查也同样没什么异样，这些都是功能性的操作。现在在之前展示的Apache配置文件上运行脚本，修改“VirtualHost local2:80”，使用“/tmp”作为VirtualHost。该命令行接口打印输出从函数replace_docroot()得到的结果，而不是将其写入一个文件：

```
jmjones@dinkgutsy:code$ python apache_conf_docroot_replace.py
  /etc/apache2/sites-available/psa
  local2:80 /tmp
NameVirtualHost 127.0.0.1:80
<VirtualHost localhost:80>
        DocumentRoot /var/www/
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>
        ErrorLog /var/log/apache2/error.log
        LogLevel warn
        CustomLog /var/log/apache2/access.log combined
        ServerSignature On
</VirtualHost>
<VirtualHost local2:80>
        DocumentRoot /tmp
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>
        ErrorLog /var/log/apache2/error2.log
        LogLevel warn
        CustomLog /var/log/apache2/access2.log combined
        ServerSignature On
</VirtualHost>
```

与local2:80 virtualHost仅有的不相同的是DocumentRoot这一行。在重新定向脚本的输出到一个文件时，两者之间的不同之处在于：

```
jmjones@dinkgutsy:code$ diff apache_conf.diff /etc/apache2/sites-available/psa
20c20
<         DocumentRoot /tmp
---
>         DocumentRoot /var/www2/
```

通过修改Apache配置文件来改变DocumentRoot是非常简单的工作，但是如果不得不经

常改变文档的根，或者有多个虚拟主机需要你切换使用，那么写一个与刚才写的类似的脚本将是非常值得的。但是，这是一个相当简单的脚本。对脚本进行修改以注释掉VirtualHost部分、修改LogLevel指令或是改变VirtualHost存放日志的位置，这些都非常简单。

处理文件

学习处理文件的关键是学会如何处理文本数据。经常地，需要处理的文件包含在诸如日志文件、配置文件或应用数据文件这样的文本文件中。当需要对正在分析的数据做进一步深入处理时，通常要创建一个指定类型的报告文件，或是将数据放入一个文本文件以便日后进一步处理。幸运的是，Python包括一个容易使用的称为file的内建类型，该类型可以协助完成所有这些事情。

创建文件

这看起来似乎不太合理，但是为了读入一个现有文件，不得不创建一个新的文件对象。但是不要把创建一个文件对象和创建一个文件搞混淆了。对文件进行写入操作需要创建一个新对象，并且需要在磁盘上创建一个新文件，因此与读入文件时创建一个文件对象相比，这似乎更合情合理。创建一个文件对象的目的是让你可以与磁盘上的这个文件进行交互。

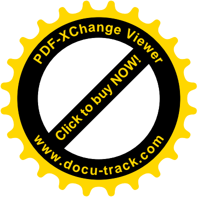
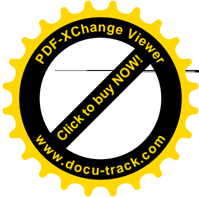
要创建一个文件对象，可以使用内建函数open()。下面是一个打开文件进行读取操作的代码示例：

```
In [1]: infile = open("foo.txt", "r")

In [2]: print infile.read()
Some Random
    Lines
Of
    Text.
```

由于open是内建函数，不需要使用import进行载入。open()具有三个参数：文件名，文件打开模式，以及缓冲区大小。只有第一个参数（文件名）是必须的。最普通的模式值为“r”（读模式，该值为默认值），“w”（写模式）和“a”（附加模式）。一个补充模式“b”可以被加到其他模式中，该模式表示二进制模式。第三个参数是缓存大小，表示缓存文件的操作方式。

在前一个示例中，我们指定以读模式使用open()打开文件“*filefoo.txt*”，并且使用变量infile引用新的可读文件对象。一旦创建了infile，就可以自由调用read()方法，读取整个文件的内容。

www.TopSage.com

创建一个用于写入的文件与创建一个用于读取的文件方法类似。写入文件使用"w"标志，而不是"r"标志。

```
In [1]: outputfile = open("foo_out.txt", "w")

In [2]: outputfile.write("This is\nSome\nRandom\nOutput Text\n")

In [3]: outputfile.close()
```

在这个示例中，我们使用open()函数以写入方式打开*foo_out.txt*文件，并且使用变量outputfile引用新创建的可写文件对象。一旦我们创建了outputfile，我们可以使用write()将文本写入到文件中，并使用close()关闭文件。

由于这是创建文件的非常简单的方式，你或许想有一个容错性更好的创建文件的方式。使用“try/finally”代码块将文件进行封闭是一个非常有用的方法，尤其是在使用write()调用时。下面的示例展示了一个在“try/findally”块中被封闭的可写文件：

```
In [1]: try:
   ...:     f = open('writeable.txt', 'w')
   ...:     f.write('quick line here\n')
   ...: finally:
   ...:     f.close()
```

使用这种写文件的方法，在异常发生时会引起close()方法被调用。事实上，即使没有异常发生，close()方法也会被包括进去。在try块执行完毕之后，不管异常是否发生，finally块都会被执行。

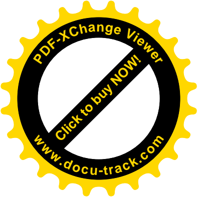
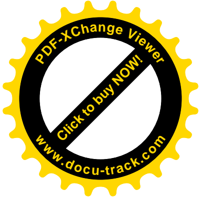
在Python2.5中一个新的关键词是with语句，它允许使用上下文管理器。一个上下文管理器是一个简单的对象，具有__enter__()和__exit__()方法。当一个对象在表达式中被创建时，上下文件管理器的__enter__()方法被调用。当with块结束后，即使发生异常，上下文管理器的__exit__()方法也会被调用。File对象定义了__enter__()和__exit__()方法。对于__exit__()，File对象close()方法被调用。下面的示例中演示了with语句：

```
In [1]: from __future__ import with_statement
In [2]: with open('writeable.txt', 'w') as f:
   ...:     f.write('this is a writeable file\n')
   ...:
   ...:
```

尽管我们没有调用文件对象f的close()方法，但上下文管理器在退出with块时将其关闭。

```
In [3]: f
Out[3]: <closed file 'writeable.txt', mode 'w' at 0x1382770>
```

www.TopSage.com

```
In [4]: f.write("this won't work")
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

/Users/jmjones/<ipython console> in <module>()

ValueError: I/O operation on closed file
```

正如我们所期望的，文件对象被关闭。这是一种处理可能的异常，并确保file对象被关闭的非常有效的实用方法。出于简化和清晰的目的，我们不会在所有示例中都这么做。

关于file对象可用方法的完整列表，参见《Python Library Reference》中文件对象一节的相关内容（*http://docs.python.org/lib/bltin-file-objects.html*）。

读取文件

一旦有了用r标志可以打开的可读文件对象，就会得到有三个常用的file方法来取得文件中包含的数据：read()、readline()和readlines()。毫无疑问read()方法可以从一个打开的文件对象读取数据，返回读取的字节数，并返回一个由这些字节组成的字符串对象。read()有一个可选参数，该参数定义读取的字节数。如果没有指定字节数，read()会尽可能读到文件的结尾。如果定义的字节数多于文件的实际字节数，read()会读到文件的结尾，并且返回已经读取的字符数。

假定有下面这样的文件：

```
jmjones@dink:~/some_random_directory$ cat foo.txt Some Random
    Lines
Of
    Text.
```

read()对文件进行处理，如下所示：

```
In [1]: f = open("foo.txt", "r")

In [2]: f.read()
Out[2]: 'Some Random\n Lines\nOf \n Text.\n'
```

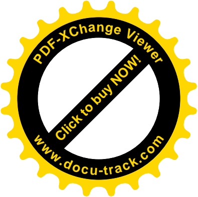
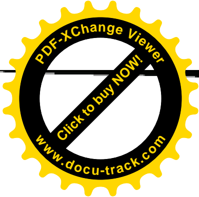
注意换行被表示成\n，这是创建一个新行的标准方法。

如果仅需要文件的前5个字节，可以这样进行处理：

```
In [1]: f = open("foo.txt", "r")

In [2]: f.read(5)
Out[2]: 'Some '
```

接下来介绍的从一个文件获取文本的方法readline()。readline()方法的目的是一次读取文件的一行文本。readline()有一个可选参数：size。size定义readline()在返回一

www.TopSage.com

个字符串对象之前读取的最大字节数，而不论是否达到了行的结尾。因此，在下面的示例中，程序会从文件foo.txt读入文本的第一行，然后从第2行读入前7个字节，后面跟着的是第二行的其余文本：

```
In [1]: f = open("foo.txt", "r")

In [2]: f.readline()
Out[2]: 'Some Random\n'

In [3]: f.readline(7)
Out[3]: '     Lin'

In [4]: f.readline()
Out[4]: 'es\n'
```

最后一个要介绍的从文件读取文本的方法是readlines()。readlines()不是排版错误，也不是之前示例中一个因剪切/粘贴留下的错误。readlines()读入文件中的所有行。是的，这就是事实。readlines()有一个sizehint选项，用于指定读入字符的大约总数。在接下来的示例中，我们创建了一个文件*biglines.txt*，该文件包含了10 000行，每一行包括80个字符。我们打开文件，希望读取文件的前N行（这总共大约1 024个字节），之后读入文件中的其他行：

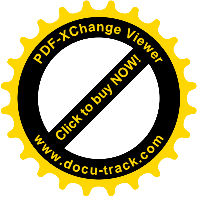
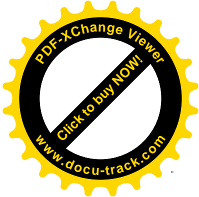
```
In [1]: f = open("biglines.txt", "r")

In [2]: lines = f.readlines(1024)

In [3]: len(lines)
Out[3]: 102

In [4]: len("".join(lines))
Out[4]: 8262

In [5]: lines = f.readlines()

In [6]: len(lines)
Out[6]: 9898

In [7]: len("".join(lines))
Out[7]: 801738
```

In [3]显示我们读取了102行，In [4]显示这些行总共8262字节。如果实际读取的字节数是8262，那么大约读取的字节数1024是怎么回事？原来内部缓存大约会占8KB。关键问题是sizehint并不是总是按着我们认为的方式工作，因此需要时刻注意。

写文件

有时需要对文件进行进一步的处理，而不仅是从文件中读取数据。这就需要创建自己的文件并将数据写到里面。为了将数据写入文件，需要掌握两个常用的file方法。第一个

www.TopSage.com

方法之前已经演示过，是write()。write()有一个参数：写入文件的字符串。下面将数据写入文件的示例中使用了write()方法：

```
In [1]: f = open("some_writable_file.txt", "w")

In [2]: f.write("Test\nFile\n")

In [3]: f.close()

In [4]: g = open("some_writable_file.txt", "r")

In [5]: g.read()
Out[5]: 'Test\nFile\n'
```

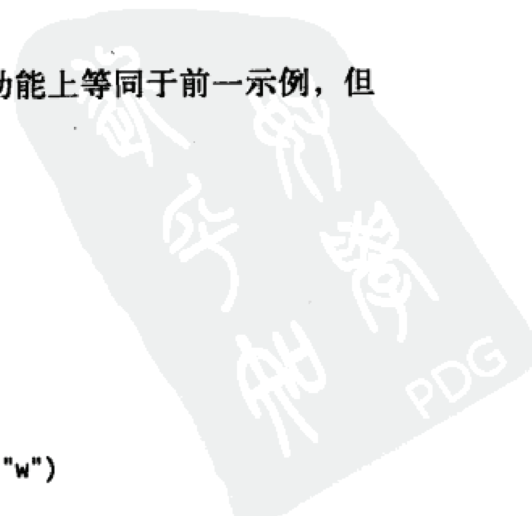
在In [1]，我们使用"w"标志（即可写入方式）打开了一个文件。In [2]向文件内写入了两行数据。In [4]为了避免与之前使用的f混淆，使用变量g作为文件对象。从In [5]可以看到写入文件的内容与我们使用read()读取出来的内容是相同的。

接下来介绍的常用数据写入方法是writelines()。writelines()必须有一个参数：writelines()要写入打开文件的序列。该序列可以是任何迭代对象类型，如列表，元组，组合列表（也是列表）或者是发生器。下面是一个发生器表达式writelines()的例子，使用writelines将数据写入文件：

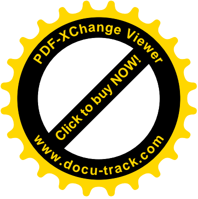
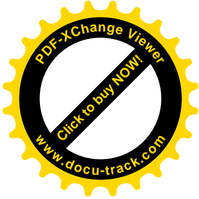
```
In [1]: f = open("writelines_outfile.txt", "w")

In [2]: f.writelines("%s\n" % i for i in range(10))

In [3]: f.close()

In [4]: g = open("writelines_outfile.txt", "r")

In [5]: g.read()

Out[5]: '0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n'
```

以下是一个使用发生器函数向文件中写入数据的示例（这在功能上等同于前一示例，但是使用了更多的代码）：

```
In [1]: def myRange(r):
   ...:     i = 0
   ...:     while i < r:
   ...:         yield "%s\n" % i
   ...:         i += 1
   ...:
   ...:

In [2]: f = open("writelines_generator_function_outfile", "w")

In [3]: f.writelines(myRange(10))

In [4]: f.close()
```



www.TopSage.com

```
In [5]: g = open("writelines_generator_function_outfile", "r")

In [6]: g.read()
Out[6]: '0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n'
```

值得注意的是writelines()无法写入一个新行（\n），你需要在写入的序列中使用“\n”。还有一点需要注意的是使用writelines()并非仅能向文件中写入基于行的信息，或许一个更准确的名字应该叫做writeiter()。在前面的示例中，我们在写入的文本中加入了换行，但是其实没有必要这么做。

额外资源

如果需要了解file对象更多的信息，请查阅DavidAscher 和Mark Lutz (O'Reilly)编著的《Learning Python》第7章（也可以在线访问*http://safari.oreilly.com/0596002815/lpython2-chp-7-sect-2*）或者查阅《Python LibraryReference》的文件对象一章（也可在线访问*http://docs.python.org/lib/bltin-file-objects.html*）。

要了解更多的generator表达式的相关内容可以参考《Python Reference Manual》的“generator expression”部分（*http://docs.python.org/ref/genexpr.html*）。为了获得更多有关yield语句的相关内容，可以查阅《Python Reference Manual》的yield语句一节（可在线访问*http://docs.python.org/ref/yield.html*）。

标准输入和输出

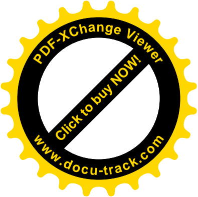
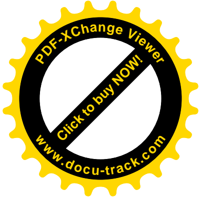
在进程的标准输入中读取文本，或是写入到进程的标准输出，这是绝大多数系统管理员熟悉的工作。标准输入只是简单地将数据送入程序，使程序在运行时可以读取。标准输出是程序的输出，由程序在运行时执行写入操作。使用标准输入和标准输出的好处是允许命令令与其他工具连接使用。

Python标准库包括一个内建的称为sys的模块，该模块提供对标准输入输出的支持。标准库提供对标准输入和输出的访问时，尽管它们没有直接连接到磁盘上的文件，却将它们当作类文件对象对待。既然它们是类文件对象，就可以使用在文件中使用的方法。可以将其视为磁盘上的文件，并且使用合适的方法。

标准输入需要通过加载sys模块和引用stdin属性（sys.stdin）进行访问。sys.stdin是一个可读的文件对象。让我们看一下如果先打开一个磁盘上的名为foo.txt的文件来创建一个文件对象，然后对比sys.stdin与这个打开的文件对象会发生什么。

```
In [1]: import sys

In [2]: f = open("foo.txt", "r")
```

www.TopSage.com

```
In [3]: sys.stdin
Out[3]: <open file '<stdin>', mode 'r' at 0x14020>

In [4]: f
Out[4]: <open file 'foo.txt', mode 'r' at 0x12179b0>

In [5]: type(sys.stdin) == type(f)
Out[5]: True
```

Python解释器将它们视为相同的类型，因此它们使用相同的方法。虽然技术上他们是相同的类型并且使用相同的方法，但类文件对象的一些方法却有不同的行为。例如，sys.stdin.seek()和sys.stdin.tell()是可用的，但是当你调用时，他们会抛出一个异常（IOError）。这里主要强调的是对于类文件对象，你可以像使用基于磁盘的文件一样进行使用。

在Python（或IPython）提示符下使用sys.stdin是毫无意义的。加载sys或执行sys.stdin.read()被永远禁止。为了展示sys.stdin如何工作的，我们创建了一个脚本，该脚本从sys.stdin()读取数据，然后打印输出带有行号的每一行。参见例3-22。

例3-22：枚举sys.stdin.readline

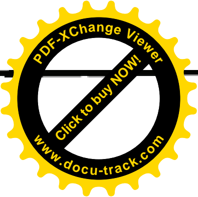
```
#!/usr/bin/env python

import sys

counter = 1
while True:
    line = sys.stdin.readline()
    if not line:
        break
    print "%s: %s" % (counter, line)
    counter += 1
```

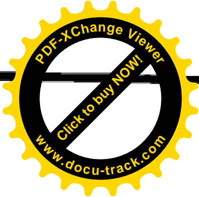
在这个示例中，我们创建变量counter来对行数进行追踪记录。脚本中有一个while循环，从标准输入设备读入行。对于每一行，打印行号和行数。在程序循环执行时，该脚本处理所有输入的行，即使其是空行。当然空行也不是彻底为空，因为包括换行符“\n”。当脚本执行到文件结尾时从循环中跳出。

下面是who与之前的脚本通过管道连接后输出的结果：

```
jmjones@dink:~/psabook/code$ who | ./sys_stdin_readline.py
1: jmjones console Jul 9 11:01

2: jmjones ttyp1 Jul 9 19:58

3: jmjones ttyp2 Jul 10 05:10

4: jmjones ttyp3 Jul 11 11:51

5: jmjones ttyp4 Jul 13 06:48
```


www.TopSage.com






```
6: jmjones ttyp5 Jul 11 21:49

7: jmjones ttyp6 Jul 15 04:38
```

有趣的是，可以通过使用enumerate函数将先前的示例写得非常简单短小。参见例3-23。

例3-23：sys.stdin readline 示例

```
#!/usr/bin/env python

import sys

for i, line in enumerate(sys.stdin):
    print "%s: %s" % (i, line)
```

在加载sys模块并且使用stdin属性时，可以使用标准输入。通过加载sys模块并引用stdout属性，可以使用标准输出。sys.stdin是一个可读的文件对象，sys.stdout是一个可写的文件对象。sys.stdin与可读文件对象具有相同的类型，sys.stdout与可写文件对象具有相同的类型。

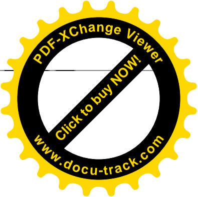
```
In [1]: import sys

In [2]: f = open('foo.txt', 'w')

In [3]: sys.stdout
Out[3]: <open file '<stdout>', mode 'w' at 0x14068>

In [4]: f
Out[4]: <open file 'foo.txt', mode 'w' at 0x1217968>

In [5]: type(sys.stdout) == type(f)
Out[5]: True
```

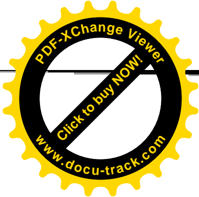
作为一个相关的方面，既然可读文件与可写文件共享相同的类型，最后一点也是理所应当的。

```
In [1]: readable_file = open('foo.txt', 'r')

In [2]: writable_file = open('foo_writable.txt', 'w')

In [3]: readable_file
Out[3]: <open file 'foo.txt', mode 'r' at 0x1243530>

In [4]: writable_file
Out[4]: <open file 'foo_writable.txt', mode 'w' at 0x1217968>

In [5]: type(readable_file) == type(writable_file)
Out[5]: True
```

要了解sys.stdout类型，首先要弄明白的一件重要事情是，它可以采用与处理可写文件相同的方法进行处理，就像sys.stdin可以被当作可读文件进行处理一样。

www.TopSage.com





StringIO

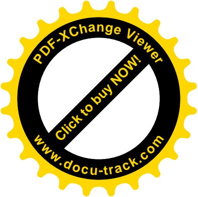
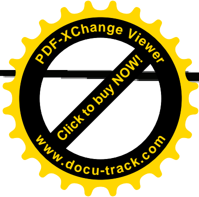
如果写了一个可以处理文件对象的函数，但需要处理的数据是文本字符串而不是文件，遇到这种情况你会不会有些不知所措？针对这个问题一个简单容易的解决方案是加载StringIO：

```
In [1]: from StringIO import StringIO

In [2]: file_like_string = StringIO("This is a\nmultiline string.\n
readline() should see\nmultiple lines of\ninput")

In [3]: file_like_string.readline()
Out[3]: 'This is a\n'

In [4]: file_like_string.readline()
Out[4]: 'multiline string.\n'

In [5]: file_like_string.readline()
Out[5]: 'readline() should see\n'

In [6]: file_like_string.readline()
Out[6]: 'multiple lines of\n'

In [7]: file_like_string.readline()
Out[7]: 'input'
```

在这个示例中，我们创建了一个StringIO对象，并将字符串“This is a\nmultiline string。\nnreadline()should see \nmultiple lines of\ninput”传递到构造器。我们能够通过StringIO对象调用readline()方法。这里，虽然readline()是我们调用的唯一方法时，但这绝不意味着这是仅有的可以使用的file方法。

```
In [8]: dir(file_like_string)
Out[8]:
['__doc__',
'__init__',
'__iter__',
'__module__',
'buf',
'buflist',
'close',
'closed',
'flush',
'getvalue',
'isatty',
'len',
'next',
'pos',
'read',
'readline',
'readlines',
'seek',
'softspace',
'tell',
```

www.TopSage.com

```
'truncate',
'write',
'writelines']
```

需要明确的是，虽然两者之间存在差异，但接口允许在文件与字符串之间进行一个简单的转换。下面是file的方法和属性与StringIO对象的方法与属性的对比。

```
In [9]: f = open("foo.txt", "r")

In [10]: from sets import Set

In [11]: sio_set = Set(dir(file_like_string))

In [12]: file_set = Set(dir(f))

In [13]: sio_set.difference(file_set)
Out[13]: Set(['__module__', 'buflist', 'pos', 'len', 'getvalue', 'buf'])

In [14]: file_set.difference(sio_set)
Out[14]: Set(['fileno', '__setattr__', '__reduce_ex__', '__new__', 'encoding',
'__getattribute__', '__str__', '__reduce__', '__class__', 'name',
'__delattr__', 'mode', '__repr__', 'xreadlines', '__hash__', 'readinto',
'newlines'])
```

正如你所看到的，如果需要将一个字符串作为文件来处理，StringIO会很有帮助。

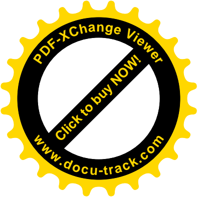
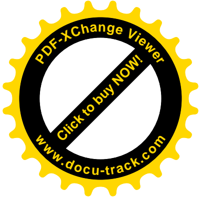
urllib

如果你有兴趣读入的文件碰巧是在互联网上该怎么办？或者是你希望复用一段代码，而该代码需要一个文件对象？内建的文件类型不知道互联网，但是urllib模块可以提供帮助。

如果想从一个web服务器读取文件，urllib.urlopen()提供了一个简单的解决方法。下面是一个简单的示例：

```
In [1]: import urllib

In [2]: url_file = urllib.urlopen("http://docs.python.org/lib/module-urllib.html")

In [3]: urllib_docs = url_file.read()

In [4]: url_file.close()

In [5]: len(urllib_docs)
Out[5]: 28486

In [6]: urllib_docs[:80]
Out[6]: '<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">\
  n<html>\n<head>\n<li'

In [7]: urllib_docs[-80:]
Out[7]: 'nt...</a></i> for information on suggesting changes.\
  n</address>\n</body>\n</html>\n'
```

www.TopSage.com

首先加载urllib。接下来使用urllib创建一个类文件对象，并命名为url_file。然后，将url_file的内容读入到称为urllib_docs的字符串中。为表明获取的数据实际上来自互联网，我们对获得的文档分割前后80个字符。注意，urllib文件对象支持read()和close()方法，也支持readline()，readlines()，fileno()和geturl()。

如果需要了解更多强大的功能，例如使用代理服务器，你可以在*http://docs.python.org/lib/module-urllib.html*找到更多的关于urllib的相关信息。如果你还需要进一步了解更多内容，例如摘要认证和cookies，可以查阅urllib2：*http://docs.python.org/lib/module-urllib2.html*。

日志解析

如果不涉及日志解析，那么从一名系统管理员的角度来讲，对文件处理的介绍是不完整的，所以接下来要对日志解析进行介绍。此时，你已经能够打开一个日志文件，读取其中每一行，并以能够以便捷的方式读取数据。在开始编写这个示例之前，我们不得不问自己"我们希望如何读取日志文件？"。答案相当简单：读取一个Apache访问日志，查看每一个独立客户端连接获得的字节数。

根据*http://httpd.apache.org/docs/1.3/logs.html*，这种"整合"的日志格式如下所示：

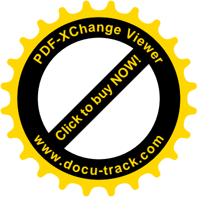
```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0"
200 2326 "http://www.example.com/start.html" "Mozilla/4.08 [en] (Win98; I;
Nav)"
```

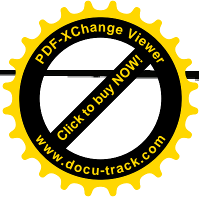
这同Apache日志数据相匹配。从日志文件中的每一行可以获得的两个感兴趣的信息：客户端的IP地址和转输的字节数。IP地址是日志文件中的第一个字段，在本例中是127.0.0.1。传输的字节数是IP地址字段之后的两个字段，在引用的前面。在本例中传输了2326字节。关于如何获得该字段，参见例3-24。

例3-24：Apache日志文件解析-以空格字符分隔

```
#!/usr/bin/env python
"""
USAGE:

apache_log_parser_split.py some_log_file

This script takes one command line argument: the name of a log file
to parse. It then parses the log file and generates a report which
associates remote hosts with number of bytes transferred to them.
"""

import sys

def dictify_logline(line):
    '''return a dictionary of the pertinent pieces of an apache combined log file
```


www.TopSage.com



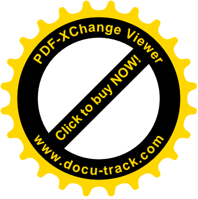
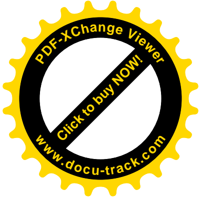


```
    Currently, the only fields we are interested in are remote host and bytes sent,
    but we are putting status in there just for good measure.
    '''
    split_line = line.split()
    return {'remote_host': split_line[0],
            'status': split_line[8],
            'bytes_sent': split_line[9],
    }

def generate_log_report(logfile):
    '''return a dictionary of format remote_host=>[list of bytes sent]

    This function takes a file object, iterates through all the lines in the file,
    and generates a report of the number of bytes transferred to each remote host
    for each hit on the webserver.
    '''
    report_dict = {}
    for line in logfile:
        line_dict = dictify_logline(line)
        print line_dict
        try:
            bytes_sent = int(line_dict['bytes_sent'])
        except ValueError:
            ##totally disregard anything we don't understand
            continue
        report_dict.setdefault(line_dict['remote_host'], []).append(bytes_sent)
    return report_dict
if __name__ == "__main__":
    if not len(sys.argv) > 1:
        print __doc__
        sys.exit(1)
    infile_name = sys.argv[1]
    try:
        infile = open(infile_name, 'r')
    except IOError:
        print "You must specify a valid file to parse"
        print __doc__
        sys.exit(1)
log_report = generate_log_report(infile)
print log_report
infile.close()
```

该示例非常简单。__main__部分仅执行了少量处理。首先，对命令行参数进行最少检测，以确保至少传递了一个参数。如果用户没有在命令行传递参数，脚本将打印一个用法信息并退出。关于如何更好地使用命令行及参数，可以参阅第13章，那里有详细的介绍。接下来，__main__试图打开指定的日志文件。如果不能成功打开，将显示用法信息并退出。然后将日志文件传递给generate_log_report()函数，并打印结果。

generate_log_report()创建一个字典，起到报告的作用。它会迭代日志文件中所有的行并将每一行传递给ditify_logline()，而该方法返回一个包含所需信息的字典。接下来，它检查bytes_sent的值是否是一个整数。如果是，则继续。如果不是，处理下一

www.TopSage.com

行。之后，利用dictfy_logline()返回的数据升级报告字典。最后，将报告字典返回到__main__。

dictify_logline()简单地以空格分割较长的行，从结果列表中提取某些项，并返回一个由分割行数据组成的字典。

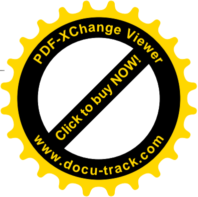
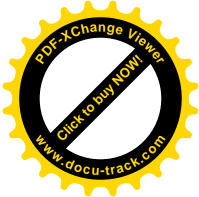
那么，它起作用了么？基本上起到了作用。请查看单元测试例3-25。

例3-25：Apache日志文件的单元测试——以空格字符分隔

```
#!/usr/bin/env python

import unittest
import apache_log_parser_split

class TestApacheLogParser(unittest.TestCase):

    def setUp(self):
        pass

    def testCombinedExample(self):
        # test the combined example from apache.org
        combined_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] '\
        '"GET /apache_pb.gif HTTP/1.0" 200 2326 "http://www.example.com/start.html" '\
        '"Mozilla/4.08 [en] (Win98; I ;Nav)"'
        self.assertEqual(apache_log_parser_split.dictify_logline(combined_log_entry),
            {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

    def testCommonExample(self):
        # test the common example from apache.org
        common_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] '\
        '"GET /apache_pb.gif HTTP/1.0" 200 2326'
        self.assertEqual(apache_log_parser_split.dictify_logline(common_log_entry),
            {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

    def testExtraWhitespace(self):
        # test for extra whitespace between fields
        common_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] '\
        '"GET /apache_pb.gif HTTP/1.0" 200 2326'
        self.assertEqual(apache_log_parser_split.dictify_logline(common_log_entry),
            {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

    def testMalformed(self):
        # test for extra whitespace between fields
        common_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] '\
        '"GET /some/url/with white space.html HTTP/1.0" 200 2326'
        self.assertEqual(apache_log_parser_split.dictify_logline(common_log_entry),
            {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

if __name__ == '__main__':
    unittest.main()
```

代码对于复合及通用的日志格式都是可以处理的，但是对于请求字段的略微修改则可能导致单员测试的失败。下面是一个测试的结果示例：

www.TopSage.com

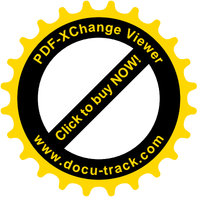
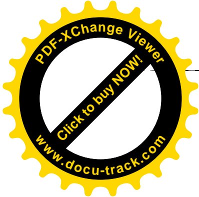
```
jmjones@dinkgutsy:code$ python test_apache_log_parser_split.py
...F
======================================================================
FAIL: testMalformed (__main__.TestApacheLogParser)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_apache_log_parser_split.py", line 38, in testMalformed
    {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})
AssertionError: {'status': 'space.html', 'bytes_sent': 'HTTP/1.0"',
'remote_host': '127.0.0.1'} != {'status': '200', 'bytes_sent': '2326',
'remote_host': '127.0.0.1'}

----------------------------------------------------------------------
Ran 4 tests in 0.001s

FAILED (failures=1)
```

因为数据域中的冒号可以转化为空格，日志文件中的所有域都被向右侧移动一位。严格根据日志格式的说明，提取远端主机名和字节数是相当安全的。这些字节数就是基于以空格为分隔符的字段。例3-26是使用正则表达式的相同示例。

例3-26：Apache日志文件解析——regex

```
#!/usr/bin/env python
"""
USAGE:

apache_log_parser_regex.py some_log_file

This script takes one command line argument: the name of a log file
to parse. It then parses the log file and generates a report which
associates remote hosts with number of bytes transferred to them.
"""

import sys
import re
log_line_re = re.compile(r'''(?P<remote_host>\S+) #IP ADDRESS
                             \s+ #whitespace
                             \S+ #remote logname
                             \s+ #whitespace
                             \S+ #remote user
                             \s+ #whitespace
                             \[[^\[\]]+\] #time
                             \s+ #whitespace
                             "[^"]+" #first line of request
                             \s+ #whitespace
                             (?P<status>\d+)
                             \s+ #whitespace
                             (?P<bytes_sent>-|\d+)
                             \s* #whitespace
                             ''', re.VERBOSE)

def dictify_logline(line):
    '''return a dictionary of the pertinent pieces of an apache combined log file
```

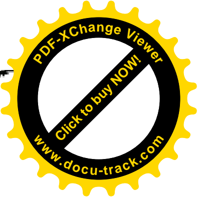
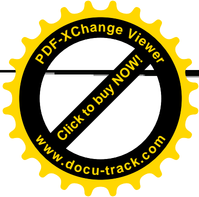
www.TopSage.com

```
    Currently, the only fields we are interested in are remote host and bytes sent,
    but we are putting status in there just for good measure.
    '''
    m = log_line_re.match(line)
    if m:
        groupdict = m.groupdict()
        if groupdict['bytes_sent'] == '-':
            groupdict['bytes_sent'] = '0'
        return groupdict
    else:
        return {'remote_host': None,
                'status': None,
                'bytes_sent': "0",
        }

def generate_log_report(logfile):
    '''return a dictionary of format remote_host=>[list of bytes sent]

    This function takes a file object, iterates through all the lines in the file,
    and generates a report of the number of bytes transferred to each remote host
    for each hit on the webserver.
    '''
    report_dict = {}
    for line in logfile:
        line_dict = dictify_logline(line)
        print line_dict
        try:
            bytes_sent = int(line_dict['bytes_sent'])
        except ValueError:
            ##totally disregard anything we don't understand
            continue
        report_dict.setdefault(line_dict['remote_host'], []).append(bytes_sent)
    return report_dict


if __name__ == "__main__":
    if not len(sys.argv) > 1:
        print __doc__
        sys.exit(1)
    infile_name = sys.argv[1]
    try:
        infile = open(infile_name, 'r')
    except IOError:
        print "You must specify a valid file to parse"
        print __doc__
        sys.exit(1)
    log_report = generate_log_report(infile)
    print log_report
    infile.close()
```

从regex示例到以空格为分隔符的示例，我们仅修改了函数dictify_logline()。这表明在regex示例中，我们放弃了函数的准确返回类型。没有对日志中的每一行以空格进行分割，而是使用了编译后的正则表达式对象log_line_re来匹配日志行。如果匹配成

www.TopSage.com

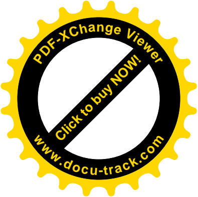
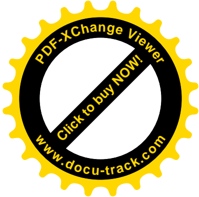
功，返回一个略有修改的groupdict()方法。该方法中当域中包含“-”时bytes_sent被设置为0，（因为“-”表示空）。在没有任何匹配的情况下，返回一个具有相同关键字的字典，但是其值为空和0。

那么，正则表达式比字符串分割更有优势么？事实上，确实如此。下面的示例是对Apache解析脚本regex的最新版本的一次单元测试。

```
#!/usr/bin/env python

import unittest
import apache_log_parser_regex

class TestApacheLogParser(unittest.TestCase):

    def setUp(self):
        pass

    def testCombinedExample(self):
        # test the combined example from apache.org
        combined_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] '\
        '"GET /apache_pb.gif HTTP/1.0" 200 2326 '\
        '"http://www.example.com/start.html" "Mozilla/4.08 [en] (Win98; I ;Nav)"'
        self.assertEqual(apache_log_parser_regex.dictify_logline(combined_log_entry),
            {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

    def testCommonExample(self):
        # test the common example from apache.org
        common_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] '\
        '"GET /apache_pb.gif HTTP/1.0" 200 2326'
        self.assertEqual(apache_log_parser_regex.dictify_logline(common_log_entry),
            {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

    def testMalformedEntry(self):
        # test a malformed modification dereived from the example at apache.org
        #malformed_log_entry = '127.0.0.1 - frank [10/Oct/2000 13:55:36 -0700] '\
        #'"GET /apache_pb.gif HTTP/1.0" 200 2326 '\
        #'"http://www.example.com/start.html" "Mozilla/4.08 [en] (Win98; I ;Nav)"'

        malformed_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] '\
        '"GET /some/url/with white space.html HTTP/1.0" 200 2326'
        self.assertEqual(apache_log_parser_regex.dictify_logline(malformed_log_entry),
            {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

if __name__ == '__main__':
    unittest.main()
```

下面是单元测试的结果：

```
jmjones@dinkgutsy:code$ python test_apache_log_parser_regex.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

ElementTree

如果需要解析的文本是XML，那么你需要一种不同于处理面向行的日志文件的处理方式。你可能不想一行接一行地读入文件、查找模式，也不想太多地依赖正则表达式。XML使用一种树结构，因此你并不希望对其按行读取。而使用正则表达式来建立一个树状数据结构对于处理任何略微大一点的文件来说，都是一件十分令人头痛的事。

这样的话，还能使用什么呢？有两种典型方法可以处理XML。一种是“simple API for XML,”或者称为 SAX。Python标准库中包括了SAX解析器。SAX是一种典型的极为快速的工具，在解析XML时，不会自动占用大量内存。但是这是基于回调机制的，因此在某些数据中，当它命中XML文档的某些部分时（例如开始和结束标签），它会调用某些方法并进行传递。这意味着必须为数据指定句柄，以维持自己的状态，而这是非常困难的。这两件事情使得“simple”在“simpleAPI for XML”中看起来有些牵强。另一个操作XML的方法是使用Document Object Model，或者称为DOM。Python标准库也包括一个DOM XML库。与SAX相比，DOM是典型的比较慢，消耗更多内存的方法，因为DOM会将整个XML树读入到内存中，并为树中的第一个节点建立一个对象。使用DOM的好处是你不需要对状态进行追踪，因为每一个节点都知道谁是它的父节点，谁是它的子节点。但是DOM API使用起来多少有些麻烦。

第三种选择是使用ElementTree（元素树）。ElementTree是XML解析库，已经在Python2.5之后被包括在标准库中。ElementTree感觉就像一个轻量级的DOM，具有方便使用、十分友好的API。除了代码可复用之外，它运行速度快，消耗内存较少。这里我们重点推荐使用ElementTree。如果需要使用XML解析器，不妨先试一试ElementTree。

使用ElementTree开始解析XML文件，只须简单地加载库和使用parse()对文件进行处理：

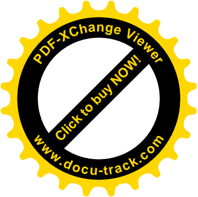
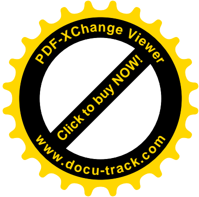
```
In [1]: from xml.etree import ElementTree as ET

In [2]: tcusers = ET.parse('/etc/tomcat5.5/tomcat-users.xml')

In [3]: tcusers

Out[3]: <xml.etree.ElementTree.ElementTree instance at 0xabb4d0></xml>
```

为了可以在使用库时省去不必要的键盘输入，我们使用名字ET来加载ElementTree模块，这样在使用该库时就可以使用ET，而不用输入全称。接下来，我们告诉ElementTree解析用户的XML文件，该文件来自一个已成功安装的Tomcat servlet引擎。我们称ElementTree对象为tcusers。tcusers的类型是xml.etree.Elementtree.ElementTree。

在删除了授权许可和用法提示之后，可以看到解析后的Tomcat用户文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<tomcat-users>
```


www.TopSage.com

```
    <user name="tomcat" password="tomcat" roles="tomcat" />
    <user name="role1" password="tomcat" roles="role1" />
    <user name="both" password="tomcat" roles="tomcat,role1" />
</tomcat-users>
```

ElementTree在解析Tomcat XML文件时创建了一个树对象，我们使用tcusers表示，这样就可以获得XML文件中的任何节点。在树对象中，两个最有意义的方法是find()和findall()。find()根据传递给它的查询内容查找匹配查询的第一个节点，并返回一个基于该节点的element对象。findall()会对匹配查询的所有节点进行查找，并返回一个element对象列表，该列表由找到的所有匹配的节点组成。

find()和findall()寻找的模式的类型都是XPath表达式的有限子集。ElementTree的有效搜索格式包括标签名*（匹配所有子元素）、.（匹配当前元素）和//（匹配搜索起启点的所有子孙节点）。斜杠（/）可以用来分隔匹配格式。利用Tomcat 用户文件，我们可以使用find()提取第一个用户节点的标签名：

```
In [4]: first_user = tcusers.find('/user')

In [5]: first_user

Out[5]: <Element user at abdd88>
```

给find()指定搜索格式为“/user”。最前面的斜杠定义了绝对路径，即从根节点开始搜索。文本“user”定义了要查找的标签。因此，find()返回标签是“user”的第一个节点。可以看到，引用的对象first_user是element类型。

一些更有趣的element方法和属性包括Attrib、find()、findall()、get()、tag和text。attrib是一个元素属性字典。find()和findall()与ElementTree对象采用相同的处理方式。get()是一个字典方法，可以获取指定的属性，如果没有定义属性，返回为空。attrib和get()都可以为当前的XML标签使用相同的属性字典。Tag是包含当前元素标签名的属性。text是当前元素作为文本节点时所具有的文本的属性。

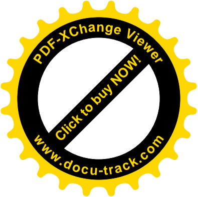
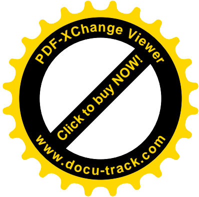
以下是一个XML元素ElementTree，是为fist_user元素对象创建的：

```
<user name="tomcat" password="tomcat" roles="tomcat" />
```

现在调用方法并且引用tcusers对象的属性：

```
In [6]: first_user.attrib

Out[6]: {'name': 'tomcat', 'password': 'tomcat', 'roles': 'tomcat'}

In [7]: first_user.get('name')

Out[7]: 'tomcat'

In [8]: first_user.get('foo')
```

www.TopSage.com

```
In [9]: first_user.tag

Out[9]: 'user'

In [10]: first_user.text
```

至此，你已经看到了一些ElementTree如何使用的基本示例。接下来看一个稍微深入且更通用的示例。我们解析Tomcat用户文件并且搜索name属性匹配我们指定内容（在本例中为'tomcat'）的用户节点（参见例3-27）。

例3-27：ElementTree解析Tomcat用户文件

```
#!/usr/bin/env python

from xml.etree import ElementTree as ET

if __name__ == '__main__':
    infile = '/etc/tomcat5.5/tomcat-users.xml'
    tomcat_users = ET.parse(infile)
    for user in [e for e in tomcat_users.findall('/user') if
        e.get('name') == 'tomcat']:
        print user.attrib
```

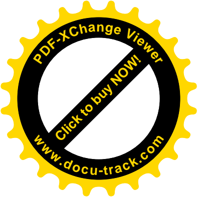
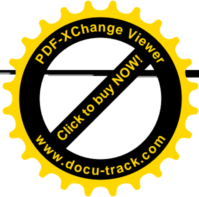
示例中，我们只是使用了一个复合列表来匹配name属性。运行这个示例将返回下面的结果：

```
jmjones@dinkgutsy:code$ python elementtree_tomcat_users.py
{'password': 'tomcat', 'name': 'tomcat', 'roles': 'tomcat'}
```

最后是一个ElementTree示例，该示例用来从一个写得不是很好的XML中提取信息。Mac OS X有一个称为system_profiler的工具，可以显示系统的大量信息。XML是system_profiler支持的两种输出格式之一，但是对XML的支持似乎来得晚一些。希望被提取出来的信息内容是操作系统的版本号，包括在类似下面这样的XML文件中：

```
<dict>
    <key>_dataType</key>
    <string>SPSoftwareDataType</string>
    <key>_detailLevel</key>
    <integer>-2</integer>
    <key>_items</key>
    <array>
        <dict>
        <key>_name</key>
        <string>os_overview</string>
        <key>kernel_version</key>
        <string>Darwin 8.11.1</string>
        <key>os_version</key>
        <string>Mac OS X 10.4.11 (8S2167)</string>
    </dict>
</array>
```

www.TopSage.com

为什么我们认为这个XML格式写得不规范呢？因为在任何一个XML标签中都没有属性。标签类型是主要的数据类型。一些元素，如可替换的key、string标签，位于相同的父节点下（参见例3-28）。

例3-28：Mac OS X system_profiler输出解析

```
#!/usr/bin/env python
import sys

from xml.etree import ElementTree as ET
e = ET.parse('system_profiler.xml')
if __name__ == '__main__':
    for d in e.findall('/array/dict'):
        if d.find('string').text == 'SPSoftwareDataType':
            sp_data = d.find('array').find('dict')
            break
    else:
        print "SPSoftwareDataType NOT FOUND"
        sys.exit(1)

record = []
for child in sp_data.getchildren():
    record.append(child.text)
    if child.tag == 'string':
        print "%-15s -> %s" % tuple(record)
        record = []
```

基本上，脚本搜索dict标签，该标签有一个字符串子元素，它的文本值为"SPSoftwareDataType"。脚本搜索的信息在该节点下。在这个示例中，我们之前没有介绍过的唯一内容是getchildren()方法。该方法可以简单地返回一个指定元素的子节点列表。另外，该示例条理非常清晰，XML也写得更好一些。下面是该脚本运行在Mac OS X Tiger笔记本上产生的输出结果：

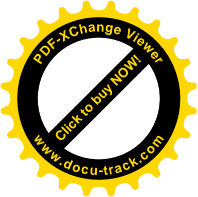
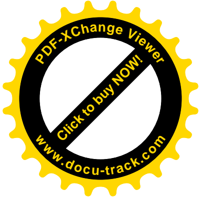
```
dink:~/code jmjones$ python elementtree_system_profile.py
_name           -> os_overview
kernel_version  -> Darwin 8.11.1
os_version      -> Mac OS X 10.4.11 (8S2167)
```

ElementTree是Python标准库的有力补充。到目前为止，我们已经多次使用，并且对使用它所带来的好处非常满意。可以尝试一下Python标准库中的SAX和DOM库，但是我们认为在你尝试之后仍会回来选择使用ElementTree。

本章小结

本章讲述了在Python中操作文本的一些基本技术。我们使用了来自标准库的内建string类型、正则表达式、标准输入输出、StringIO以及urllib模块。然后对其中的一些技术

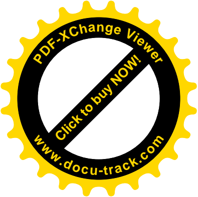
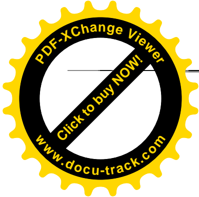
www.TopSage.com

进行联合，应用到两个解析Apache日志文件的示例中。最后介绍了ElementTree库的一些经典用法，并通过两个示例演示了实际应用效果。

一些UNIX使用者认为当复杂的文本处理已经超出了使用grep或awk所能应对的程度时，这种情况下，他们只考虑选用Perl作为更高级的替代工具。Perl是一个功能极为强大的语言，特别是在文本处理方面，然而我们认为Python具有与Perl一样优秀的性能。事实上，如果看到Python简单的语法，以及可以轻松实现由面向过程的编码方式到面向对象的编码方式的转变，你会认为Python比Perl更为优秀，甚至在文本处理方面同样如此。因此，我们希望你下次从事文本处理工作时，首先选择Python。

www.TopSage.com

第4章

文档与报告

你可以能会发现，工作中最单调乏味的事情就是根据用户的需要，对各种各样的信息进行归档。这可能是为了那些希望阅读文档的用户的直接利益，也可能是为了用户的间接利益，你或是你的继任者都或许会在将来对程序实施改进时用到这些文档。因此，无论哪种情况，创建文档都是工作中一个十分重要的内容。但是，如果发现这不是自己喜欢做的工作，就会让人十分烦恼。现在，Python可以帮助我们完成与文档相关的一些工作。尽管Python无法写文档，但是它可以帮你搜集、格式化这些文档，并将信息发送到需要它们的人手中。

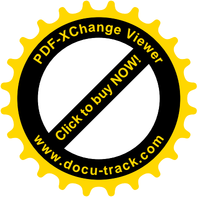
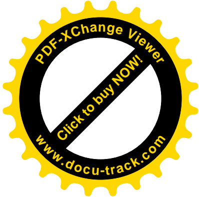
在本章中，我们将集中介绍如何对写的程序进行信息采集、格式化和发布。你感兴趣的共享信息会被保存到某些地方，或许是日志文件的某个位置，或许是头脑中，或许是执行某个shell命令的结果中，也有可能是在某处的数据库中。首先，要对信息进行采集。接下来，为了能够有效地共享信息，需要按照某种方式对数据进行格式化，使其更有意义。数据的格式可以是PDF、PNG、JPG、HTML或纯文本。最后，需要将这些信息送到感兴趣的人手中。感兴趣的人将会通过收取电子邮件，访问一个网站，或者直接通过共享驱动器来查找文件，但哪种方式最方便呢？

自动信息收集

信息共享的第一步就是收集信息。本书已在另外两章对数据采集进行了介绍，分别是：文本（第3章）和SNMP（第7章）。文本一章中包括了一些示例，这些示例演示了从一个大文本文件中解析和提取各种数据的方法。第3章中一个典型的示例，就是对Apache web服务器日志中的每一行进行解析，包括解析客户端IP地址、传输字节数以及HTTP状态代码。而第7章，包含了系统信息查询的示例，包括查询RAM容量以及网络接口的速度等。

比较而言，信息收集所涉及的内容，比定位和提取某些数据更为深入一些。大多数情况下，信息收集都包含了从某种格式中提取信息，例如Apache logfile，并以一个中间格式

www.TopSage.com

进行保存以备将来使用。例如，如果想创建一个图表，显示一个月内每一个唯一IP地址从指定的Apache web服务器下载信息的字节数。这一过程中，信息收集阶段可能包括每晚解析Apache日志，提取必要的信息（在本例中，就是IP地址信息和每一个请求所发送的字节数），并将这些数据保存起来，这样以后就可以打开使用。这些需要保存的数据可以存储在关系数据库、对象数据库、pickle文件、CSV文件和纯文本文件中。

这一节的剩余部分试图将文本处理和数据持久性中的多个概念进行整合，并特别阐明这种整合是如何构建在第3章的数据提取以及第12章的数据存储之上。我们将使用与文本处理中相同的库，也将使用shelve模块来保存每个来自HTTP客户端的HTTP请求。在第12章中会对shelve模块进行介绍。

下面是一个简单的模块示例，使用了在前一章中创建的Apache日志文件解析模块，以及shelve模块：

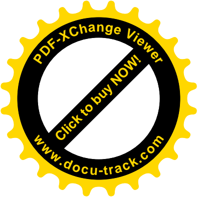
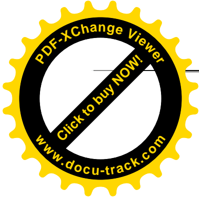
```
#!/usr/bin/env python

import shelve
import apache_log_parser_regex

logfile = open('access.log', 'r')
shelve_file = shelve.open('access.s')

for line in logfile:
    d_line = apache_log_parser_regex.dictify_logline(line)
    shelve_file[d_line['remote_host']] = \
        shelve_file.setdefault(d_line['remote_host'], 0) + \
        int(d_line['bytes_sent'])

logfile.close()
shelve_file.close()
```

该示例首先加载了shelve和apache_log_parser_regex。shelve是一个来自Python标准库的模块。apache_log_parser_regex是我们在第3章中编写的模块。之后，打开apache日志文件，access.log和一个框架文件access.s。对日志文件中的每一行进行迭代处理，使用Apache日志文件解析模块为每一行创建一个字典。该字典包括HTTP请求的状态代码，客户端IP地址，以及传输到客户端的字节数。将特定请求的字节数加到总字节数中，总字节数已经在shelve对象中为每一个客户端IP地址都进行了计算。如果在shelve对象中没有该客户端IP地址的记录，则总字节数自动归0。在对日志文件中的所有行都进行了迭代之后，关闭日志文件和shelve对象。本章后面介绍信息格式化的部分将继续使用这个示例。

收取邮件

你或许从未考虑过将收取邮件作为信息收集的一种方式，而事实上，收取邮件确实可以

www.TopSage.com

实现信息收集。想象一下你有一些服务器，这些服务器很难彼此直接连接在一起，但是每一台服务器都支持邮件功能。如果有一个监测这些服务器上web应用的脚本，该脚本可以每隔几分钟就写入或读取日志，那么，就可以利用email作为信息传递的机制。不管写入或读取日志成功与否，都发送一个email，email中记录了成功或失败的信息。可以收集这些email信息来生成报告，或者在出现服务器故障时随时进行替换。

IMAP和POP3是两个最常见的收取邮件协议。在Python“连电池都包括在内”的风格下，Python标准库提供了对这两个协议的支持。

POP3也许保存在指定的服务器上更常见些，使用poplib访问POP3的email非常容易。例4-1显示了如何使用poplib来收取所有的email并将它们写入一系列磁盘文件。

例4-1：通过POP3收取email

```
#!/usr/bin/env python

import poplib

username = 'someuser'
password = 'S3Cr37'

mail_server = 'mail.somedomain.com'

p = poplib.POP3(mail_server)
p.user(username)
p.pass_(password)
for msg_id in p.list()[1]:
    print msg_id
    outf = open('%s.eml' % msg_id, 'w')
    outf.write('\n'.join(p.retr(msg_id)[1]))
    outf.close()
p.quit()
```

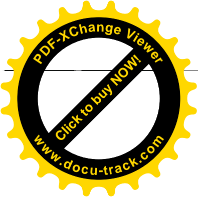
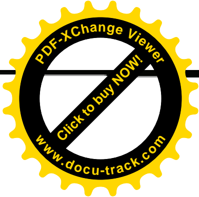
正如你所看到的，我们首先定义了username，password和mail_server，之后连接到邮件服务器，并给出了已定义的用户名和密码。假设用户名和密码正确，那么就获得了在该账户下查看email的权限。迭代处理邮件文件列表，依次读取邮件并将其写入磁盘。该脚本没有完成在收取邮件之后对邮件的删除操作。删除邮件只需要retr()之后调用dele()。

IMAP几乎与POP3一样简单，但是在Python标准库文档中没能充分地介绍。例4-2显示了IMAP代码，该代码能够完成POP3示例中相同的功能。

例4-2：通过IMAP收取email

```
#!/usr/bin/env python

import imaplib

username = 'some_user'
password = '70P53Cr37'
```

www.TopSage.com

```
mail_server = 'mail_server'

i = imaplib.IMAP4_SSL(mail_server)
print i.login(username, password)
print i.select('INBOX')
for msg_id in i.search(None, 'ALL')[1][0].split():
    print msg_id
    outf = open('%s.eml' % msg_id, 'w')
    outf.write(i.fetch(msg_id, '(RFC822)')[1][0][1])
    outf.close()
i.logout()
```

正如在POP3示例中所做的操作，我们在脚本开始处定义了`username`，`password`和`mail_server`，之后通过SSL连接到IMAP服务器，接下来登录并设置email的目录为INBOX，然后对整个目录进行迭代搜索。`search()`方法在Python标准库文件中只有少量的描述。字符集和搜索准则是`Search()`的两个必要参数。什么是一个有效的字符集？需要什么格式？我们怀疑阅读IMAP RFC是否会有帮助，但幸运的是，有大量关于IMAP示例的文档可供参考。对于每一次循环迭代，我们将邮件的内容写入到磁盘。这里会出现一个警告信息：该操作要求目录下的邮件标识为“可读”。这对你来说或许根本不是什么问题，如果删除这一信息也没什么大不了，但是还是注意一下这些信息为好。

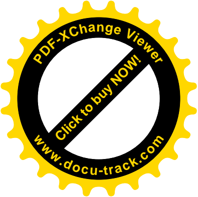
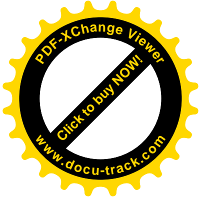
手工信息收集

接下来，让我们看一下更为复杂的手工收集信息的方法。手工收集信息意味着对我们依靠眼睛和手中的按键对内容进行收集。例如服务器的列表，其中包含每台服务器所对应的IP地址和功能；联系人的列表，其中包含email地址，电话号码，IM屏幕名；或是你的团队成员计划休假的日期等。实际上，已经有一些工具能够管理这类信息。尽管不能实现完全管理，但也可以管理大多数这类信息。例如，Excel或OpenOffice的电子表格（Spreadsheet）就能够用于管理服务器列表。Outlook或Address Book.app就能够用于管理联系人。无论是Excel/OpenOffice电子表格还是Outlook，都可以对人员休假日期进行管理。如果是使用纯文本编辑数据，并具有可配置的、支持HTML的输出（或XHTML），这或许就是解决这一问题的方案。

虽然有很多选择，但是我们将在这里推荐一种可替换的、专门的纯文本格式，即reStructuredText（也称作reST）。下面是reStructuredText网站对其进行的描述：

> reStructuredText是一个易于读取，所见即所得的纯文本标记语法和解析系统。对于行内程序归档非常有用（例如Python中的docstrings），可以快速地创建简单的web页面和独立的文档。reStructuredText为扩展特定应用领域而设计。reStructuredText解析器是一个Docutils组件。reStructuredText是对StructuredText和Setext这样的轻量级标记系统的修订和再解析。

www.TopSage.com

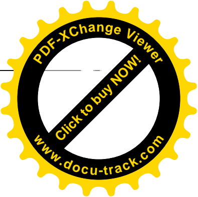
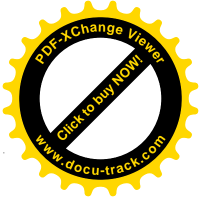
名人简介：RESTLESS

Aaron Hillegass

Aaron Hillegass，曾在NeXT和Apple任职，是Mac系统下的应用开发专家。他是*Cocoa Programming for Mac OS X* (Big Nerd Ranch)一书的作者。在Big Nerd Ranch教授Cocoa编程。

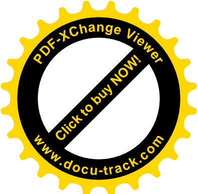
请从该书的代码资源库*http://www.oreilly.com/9780596515829*下载ReSTless的全部代码。下面是如何在Cocoa应用中调用Python脚本的示例：

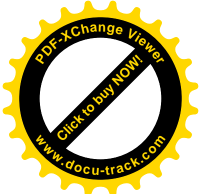
```
#import "MyDocument.h"

@implementation MyDocument

- (id)init
{
    if (![super init]) {
        return nil;
    }

    // What you see for a new document
    textStorage = [[NSTextStorage alloc] init];
    return self;
}

- (NSString *)windowNibName
{
    return @"MyDocument";
}

- (void)prepareEditView
{
    // The layout manager monitors the text storage and
    // layout the text in the text view
    NSLayoutManager *lm = [editView layoutManager];

    // Detach the old text storage
    [[editView textStorage] removeLayoutManager:lm];

    // Attach the new text storage
    [textStorage addLayoutManager:lm];
}

- (void)windowControllerDidLoadNib:(NSWindowController *) aController
{
    [super windowControllerDidLoadNib:aController];

    // Show the text storage in the text view
    [self prepareEditView];
}
```

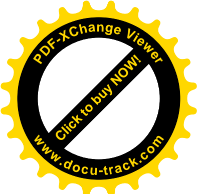
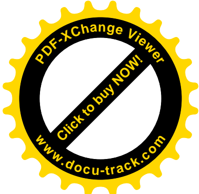
```
#pragma mark Saving and Loading

// Saves (the URL is always a file:)
- (BOOL)writeToURL:(NSURL *)absoluteURL
            ofType:(NSString *)typeName
             error:(NSError **)outError;
{
    return [[textStorage string] writeToURL:absoluteURL
                                  atomically:NO
                                    encoding:NSUTF8StringEncoding
                                       error:outError];
}

// Reading (the URL is always a file:)
- (BOOL)readFromURL:(NSURL *)absoluteURL
             ofType:(NSString *)typeName
              error:(NSError **)outError
{
    NSString *string = [NSString stringWithContentsOfURL:absoluteURL
                                                 encoding:NSUTF8StringEncoding
                                                    error:outError];

    // Read failed?
    if (!string) {
        return NO;
    }
    [textStorage release];
    textStorage = [[NSTextStorage alloc] initWithString:string
                                             attributes:nil];

    // Is this a revert?
    if (editView) {
        [self prepareEditView];
    }

    return YES;
}

#pragma mark Generating and Saving HTML

- (NSData *)dataForHTML
{
    // Create a task to run rst2html.py
    NSTask *task = [[NSTask alloc] init];

    // Guess the location of the executable
    NSString *path = @"/usr/local/bin/rst2html.py";

    // Is that file missing? Try inside the python framework
    if (![[NSFileManager defaultManager] fileExistsAtPath:path]) {
        path = @"/Library/Frameworks/Python.framework/Versions/Current/bin/rst2html.py";
    }
    [task setLaunchPath:path];

    // Connect a pipe where the ReST will go in
    NSPipe *inPipe = [[NSPipe alloc] init];
    [task setStandardInput:inPipe];
    [inPipe release];
```


www.TopSage.com



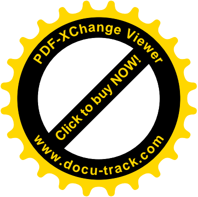
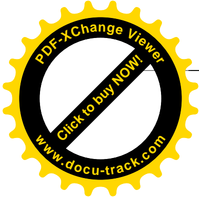


```
    // Connect a pipe where the HMTL will come out
    NSPipe *outPipe = [[NSPipe alloc] init];
    [task setStandardOutput:outPipe];
    [outPipe release];

    // Start the process
    [task launch];

    // Get the data from the text view
    NSData *inData = [[textStorage string] dataUsingEncoding:NSUTF8StringEncoding];

    // Put the data in the pipe and close it
    [[inPipe fileHandleForWriting] writeData:inData];
    [[inPipe fileHandleForWriting] closeFile];

    // Read the data out of the pipe
    NSData *outData = [[outPipe fileHandleForReading] readDataToEndOfFile];

    // All done with the task
    [task release];

    return outData;
}

- (IBAction)renderRest:(id)sender
{
    // Start the spinning so the user feels like waiting
    [progressIndicator startAnimation:nil];

    // Get the html as an NSData
    NSData *htmlData = [self dataForHTML];

    // Put the html in the main WebFrame
    WebFrame *wf = [webView mainFrame];
    [wf loadData:htmlData
        MIMEType:@"text/html"
textEncodingName:@"utf-8"
        baseURL:nil];


    // Stop the spinning so the user feels done
    [progressIndicator stopAnimation:nil];
}


// Triggered by menu item
- (IBAction)startSavePanelForHTML:(id)sender
{
    // Where does it save by default?
    NSString *restPath = [self fileName];
    NSString *directory = [restPath stringByDeletingLastPathComponent];
    NSString *filename = [[[restPath lastPathComponent]
                          stringByDeletingPathExtension]
                           stringByAppendingPathExtension:@"html"];

    // Start the save panel
    NSSavePanel *sp = [NSSavePanel savePanel];
```

```
    [sp setRequiredFileType:@"html"];
    [sp setCanSelectHiddenExtension:YES];
    [sp beginSheetForDirectory:directory
                          file:filename
                modalForWindow:[editView window]
                 modalDelegate:self
                didEndSelector:@selector(htmlSavePanel:endedWithCode:context:)
                   contextInfo:NULL];
}

// Called when the save panel is dismissed
- (void)htmlSavePanel:(NSSavePanel *)sp
        endedWithCode:(int)returnCode
              context:(void *)context
{
    // Did the user hit Cancel?
    if (returnCode != NSOKButton) {
        return;
    }

    // Get the chosen filename
    NSString *savePath = [sp filename];

    // Get the HTML data
    NSData *htmlData = [self dataForHTML];

    // Write it to the file
    NSError *writeError;
    BOOL success = [htmlData writeToFile:savePath
                                 options:NSAtomicWrite
                                   error:&writeError];

    // Did the write fail?
    if (!success) {

        // Show the user why
        NSAlert *alert = [NSAlert alertWithError:writeError];
        [alert beginSheetModalForWindow:[editView window]
                          modalDelegate:nil
                         didEndSelector:NULL
                            contextInfo:NULL];
        return;
    }
}

#pragma mark Printing Support

- (NSPrintOperation *)printOperationWithSettings:(NSDictionary *)printSettings
                                           error:(NSError **)outError
{
    // Get the information from Page Setup
    NSPrintInfo *printInfo = [self printInfo];

    // Get the view that displays the whole HTML document
    NSView *docView = [[[webView mainFrame] frameView] documentView];
```


www.TopSage.com

```
        // Create a print operation
        return [NSPrintOperation printOperationWithView:docView
                                               printInfo:printInfo];
    }

    @end
```

ReST是Python文档优先使用的格式。如果创建了一个包含了代码的Python包，并打算上传到PyPI，最好用reStructuredText作为文档的格式。由于归档的需要，许多独立的Python项目也使用ReST作为主要的文档格式。

那么，我们为什么要用ReST作为文档格式呢？首先，这种格式并不复杂。其次，标记几乎可以立即掌握。当你看到文档的结构，会很快理解作者的意图。下面是一个非常简单的ReST文件示例：

```
=======
Heading
=======
SubHeading
----------
This is just a simple
little subsection. Now,
we'll show a bulleted list:

- item one
- item two
- item three
```

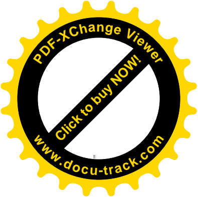
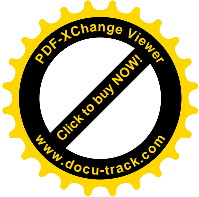
你或许已经理解了一些基本结构，不需要再去阅读构成一个有效的reStructuredText文件需要些什么之类的内容了。但是你还不能写ReST文件，只是已经可以一行接一行地进行阅读了。

第三，从ReST转换到HTML非常简单。这也是我们将在本节集中介绍的第三个方面。这里不会给出一个reStructuredText的培训指导。如果想要快速地浏览标记语法，可以访问*http://docutils.sourceforge.net/docs/user/rst/quickref.html*。

使用刚刚演示的ReST文档，下面的示例展示了将ReST转换为HTML的步骤：

```
In [2]: import docutils.core

In [3]: rest = '''=======
   ...: Heading
   ...: =======
   ...: SubHeading
   ...: ----------
   ...: This is just a simple
```

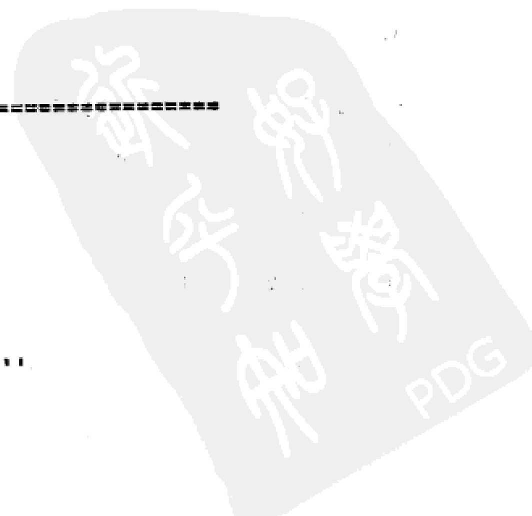
www.TopSage.com

```
   ...: little subsection. Now,
   ...: we'll show a bulleted list:
   ...:
   ...: - item one
   ...: - item two
   ...: - item three
   ...: '''

In [4]: html = docutils.core.publish_string(source=rest, writer_name='html')

In [5]: print html[html.find('<body>') + 6:html.find('</body>')]

<div class="document" id="heading">
<h1 class="title">Heading</h1>
<h2 class="subtitle" id="subheading">SubHeading</h2>
<p>This is just a simple
little subsection. Now,
we'll show a bulleted list:</p>
<ul class="simple">
<li>item one</li>
<li>item two</li>
<li>item three</li>
</ul>
</div>
```

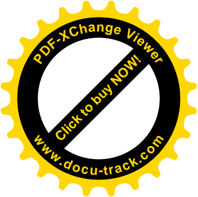
整个过程十分简单。首先加载docutils.core。然后定义了一个包含reStructuredText的字符串，再通过docutils.core.publish_string()运行字符串，并将它格式化为HTML。最后，我们做了一个字符串分割，提取在<body>和</body>标记之间的文本。我们分割div区域是因为docutils（进行HTML转换用到的库）在产生的HTML页面中嵌入样式表（stylesheet），以使转换的HTML页面看起来不是太平淡。

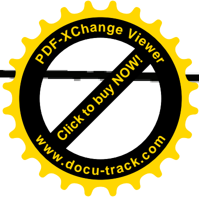
现在，举一个与系统管理员更为相关的例子。每一个好的系统管理需要对服务器以及服务器上的任务进行追踪。下面是一个示例，演示了创建一个纯文本（plain-text）服务列器表，并将其转换为HTML。



```
In [6]: server_list = '''============== ============ ================
   ...: Server Name    IP Address   Function
   ...: ============== ============ ================
   ...: card           192.168.1.2  mail server
   ...: vinge          192.168.1.4  web server
   ...: asimov         192.168.1.8  database server
   ...: stephenson     192.168.1.16 file server
   ...: gibson         192.168.1.32 print server
   ...: ============== ============ ================'''
In [7]: print server_list
============== ============ ================
  Server Name   IP Address   Function
============== ============ ================
card           192.168.1.2  mail server
vinge          192.168.1.4  web server
asimov         192.168.1.8  database server
```


www.TopSage.com

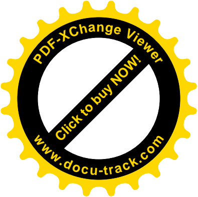
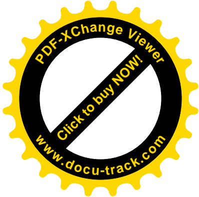




```
stephenson      192.168.1.16  file server
gibson          192.168.1.32  print server
==============  ============  =================

In [8]: html = docutils.core.publish_string(source=server_list,
writer_name='html')

In [9]: print html[html.find('<body>') + 6:html.find('</body>')]

<div class="document">
<table border="1" class="docutils">
<colgroup>
<col width="33%" />
<col width="29%" />
<col width="38%" />
</colgroup>
<thead valign="bottom">
<tr><th class="head">Server Name</th>
<th class="head">IP Address</th>
<th class="head">Function</th>
</tr>
</thead>
<tbody valign="top">
<tr><td>card</td>
<td>192.168.1.2</td>
<td>mail server</td>
</tr>
<tr><td>vinge</td>
<td>192.168.1.4</td>
<td>web server</td>
</tr>
<tr><td>asimov</td>
<td>192.168.1.8</td>
<td>database server</td>
</tr>
<tr><td>stephenson</td>
<td>192.168.1.16</td>
<td>file server</td>
</tr>
<tr><td>gibson</td>
<td>192.168.1.32</td>
<td>print server</td>
</tr>
</tbody>
</table>
</div>
```

另外一个非常好的纯文本标记模式是Textile。根据其网站的说明，Textile将纯文本替换为*simple*标记，产生有效的XHTML。在一些web应用中，例如内容管理系统、博客以及在线论坛中Texttile被广泛使用。那么，如果Textile是一种标记语言，为什么我们在这本关于Python的书中介绍它呢？原因是Python库允许处理Textile标记，并将其转换为XHTML。可以编写命令行工具来调用Python库，并转换Textile文件，然后重定向输

www.TopSage.com



PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

出到XHTML文件中。或者可以在一些脚本中调用Textile转换模块，并编程处理返回的XHTML。无论怎样做，Textile标记和Textile处理模块都可以根据归档的需求为你带来巨大的帮助。

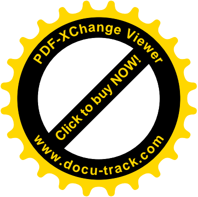
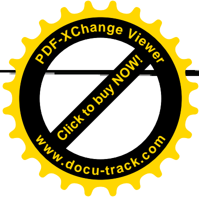
可以使用easy_install textile安装Textile Python模块。也可以使用系统中的打包系统（如果已经安装了）来完成安装。对于Ubuntu，包名是python-textile，可以使用apt-get install python-textile命令安装。一旦被安装，就可以通过简单地加载创建一个Textiler对象，并且在该对象上调用一个简单的方法来开始使用Textile。下面示例代码演示了如何将一个Textile的符号列表转换为XHTML：

```
In [1]: import textile

In [2]: t = textile.Textiler('''* item one
   ...: * item two
   ...: * item three''')

In [3]: print t.process()
<ul>
<li>item one</li>
<li>item two</li>
<li>item three</li>
</ul>
```

我们在这里不会对Textile进行教学。在web上有大量这方面的资源。例如*http://hobix.com/textile/*提供了许多与使用Textile相关的不错的参考资料。由于不会过多地对Textile的ins和outs进行介绍，我们会演示Textile的一个手工收集数据的示例。示例中收集的信息之前也说过，是一个包括IP地址和功能的服务器列表：

```
In [1]: import textile

In [2]: server_list = '''|_. Server Name|_. IP Address|_. Function|
   ...: |card|192.168.1.2|mail server|
   ...: |vinge|192.168.1.4|web server|
   ...: |asimov|192.168.1.8|database server|
   ...: |stephenson|192.168.1.16|file server|
   ...: |gibson|192.168.1.32|print server|'''

In [3]: print server_list
|_. Server Name|_. IP Address|_. Function|
|card|192.168.1.2|mail server|
|vinge|192.168.1.4|web server|
|asimov|192.168.1.8|database server|
|stephenson|192.168.1.16|file server|
|gibson|192.168.1.32|print server|

In [4]: t = textile.Textiler(server_list)

In [5]: print t.process()
<table>
<tr>
<th>Server Name</th>
```

www.TopSage.com

```
<th>IP Address</th>
<th>Function</th>
</tr>
<tr>
<td>card</td>
<td>192.168.1.2</td>
<td>mail server</td>
</tr>
<tr>
<td>vinge</td>
<td>192.168.1.4</td>
<td>web server</td>
</tr>
<tr>
<td>asimov</td>
<td>192.168.1.8</td>
<td>database server</td>
</tr>
<tr>
<td>stephenson</td>
<td>192.168.1.16</td>
<td>file server</td>
</tr>
<tr>
<td>gibson</td>
<td>192.168.1.32</td>
<td>print server</td>
</tr>
</table>
```

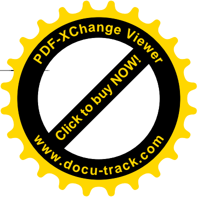
可以看到，使用ReST和Textile都可以有效地整合将纯文本数据转换为Python脚本的功能。如果确实有一些数据需要转换为HTML，例如服务器列表，联系人列表，然后在这些数据之上完成一些操作（如将HTML依据接收者列表通过email进行发送，或是将HTML通过FTP传输到某处的web服务器），那么docutils或者Textile库都是非常有用的工具。

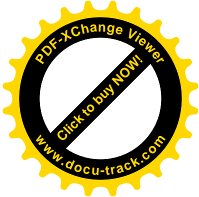
信息格式化

将信息交到用户手中之前，需要对信息进行格式化，将其转换为一种更为容易读取和识别的格式。这些格式应具有容易被用户理解的特点，如果还同时具有很强的吸引力，那就更好了。从技术上说，ReST和Textile包括对共享数据的收集和格式化两个步骤，但是，接下来的示例主要集中在对已经采集的数据进行转换方面，即如何将这些数据转换为一种更具表达力的格式。

Graphical Images

接下来的两个示例将继续前面的内容，即解析Apache日志文件的客户端IP地址和传输


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

的字节数。前一章中的示例产生了一个shelve文件，其中包括了一些我们希望与其他用户共享的信息。现在，我们根据shelve文件创建一个图表对象，实现更方便地阅读这些数据：

```
#!/usr/bin/env python

import gdchart
import shelve

shelve_file = shelve.open('access.s')
items_list = [(i[1], i[0]) for i in shelve_file.items()]
items_list.sort()
bytes_sent = [i[0] for i in items_list]
#ip_addresses = [i[1] for i in items_list]
ip_addresses = ['XXX.XXX.XXX.XXX' for i in items_list]

chart = gdchart.Bar()
chart.width = 400
chart.height = 400
chart.bg_color = 'white'
chart.plot_color = 'black'
chart.xtitle = "IP Address"
chart.ytitle = "Bytes Sent"
chart.title = "Usage By IP Address"
chart.setData(bytes_sent)
chart.setLabels(ip_addresses)
chart.draw("bytes_ip_bar.png")

shelve_file.close()
```

在这个示例中，首先加载了两个模块，gdchart和shelve。之后打开了在前一个示例中创建的shelve文件。shelve对象类似内建的字典对象，我们能够调用其中的Items()方法。items()返回一个元组列表，这个元组中的第一个元素就是字典关键字，第二个元素是关键字的值。items()方法能够按着使数据更有意义的方式来对数据进行排序。使用一个列表对之前的元组进行反向排序。现在元组的内容由（ip_address，bytes_sent）变成了（bytes_sent，ip_addresses）。然后对这个列表进行排序，由于bytes_sent是第一个元素，list.sort()方法会按该字段进行排序。再次使用复合列表填充bytes_sent和ip_addresses字段。你或许已经注意到了，我们插入XXX.XXX.XXX.XXX代替具体IP地址，因为这是从一个实际的服务器上获取的日志文件。

在获得用于生成图表的数据之后，使用gdchart来制作一个数据的图形表示。首先创建一个图表对象gdchart.Bar，为其设置一些属性，并生成一个PNG文件。之后，以像素为单位定义图表的大小。使用冒号来设置背景、前景并创建标题。为图表设置数据和标签，这两项都可以从Apache日志文件解析模块中获得。最后，使用draw()来绘制图表，并输出到文件，然后关闭shelve对象。图4-1显示了生成的图表。

www.TopSage.com

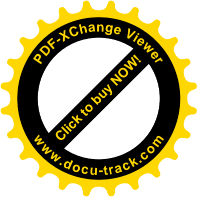
PDF-XChange Viewer Click to buy NOW! www.docu-track.com

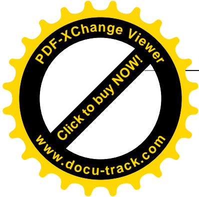
图4-1：每个IP地址请求字节数的柱状图

下面是另一个可视的格式化shelve数据脚本示例，但是这次没有使用柱状图，而是使用了饼图：

```
#!/usr/bin/env python

import gdchart
import shelve
import itertools

shelve_file = shelve.open('access.s')
items_list = [(i[1], i[0]) for i in shelve_file.items() if i[1] > 0]
items_list.sort()
bytes_sent = [i[0] for i in items_list]
#ip_addresses = [i[1] for i in items_list]
ip_addresses = ['XXX.XXX.XXX.XXX' for i in items_list]

chart = gdchart.Pie()
chart.width = 800
chart.height = 800
chart.bg_color = 'white'
color_cycle = itertools.cycle([0xDDDDDD, 0x111111, 0x777777])
color_list = []
for i in bytes_sent:
    color_list.append(color_cycle.next())
chart.color = color_list
```

www.TopSage.com



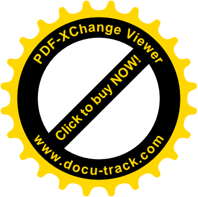
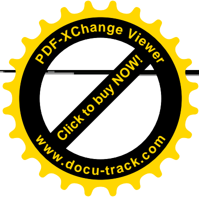



```
chart.plot_color = 'black'
chart.title = "Usage By IP Address"
chart.setData(*bytes_sent)
chart.setLabels(ip_addresses)
chart.draw("bytes_ip_pie.png")

shelve_file.close()
```

这个脚本与柱状图示例几乎相同，只是稍微做了一些修改。首先，脚本创建了一个gdchart.Pie对象实例，而不是gdchart.Bar。接下来，为单独的数据点设置颜色，而不是都使用黑色。因为这是一个饼图，若将所有数据饼块以黑色表示将无法阅读，因此轮流使用三种灰度色。这里使用itertools模块中的cycle()函数来轮换所选择的三种颜色。我们建议学习一下itertools模块。这里有许多有趣的函数可以帮助我们处理迭代对象（例如列表）。图4-2是生成饼图脚本的运行结果。

图4-2：每个IP地址请求字节数的饼图

www.TopSage.com

在饼图中，唯一的问题是IP地址有可能会与具体的字节数值叠在一起显示。柱状图和饼图使得保存在shelve文件中的数据十分易于阅读，两种图也都很容易创建，同时，向其中添加信息也极其简单。

PDF

另外一种对数据文件中的信息进行格式化的方法，是将信息保存在PDF文件中。现在PDF已经成为主流，我们总是希望所有的文档都可以转化为PDF格式。作为系统管理员，知道如何创建易读的PDF文档可以使自己的生活变得轻松。在阅读本节之后，你将能够创建各种PDF报告，例如网络应用情况、用户账号等。我们还将描述如何使用Python在多部件（multipart） MIME email中自动嵌入PDF。

在PDF库中，最重量级的库是ReportLab。ReportLab有自由版本和商业版本两种。你可以在ReportLab PDF库中查看一些示例，地址为*http://www.reportlab.com/docs/userguide.pdf*。除了阅读本章之外，我们强烈建议你阅读ReportLab的官方文档。为了在Ubuntu上安装ReportLab，可以简单地使用`apt-get install python-reportlab`命令。如果没有使用Ubuntu，可以搜索适合于你所使用操作系统的安装包。无论如何，总有一个源码发布版本可供使用。

例4-3演示了如何使用ReportLab创建一个“Hello World” PDF文件。

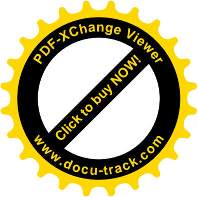
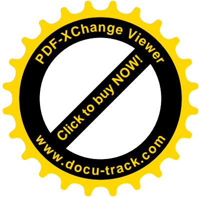
例4-3："Hello World" PDF

```
#!/usr/bin/env python
from reportlab.pdfgen import canvas

def hello():
    c = canvas.Canvas("helloworld.pdf")
    c.drawString(100,100,"Hello World")
    c.showPage()
    c.save()
hello()
```

在创建“Hello World” PDF文件的过程中，有一些需要注意的地方。首先，我们创建了一个canvas对象。接下来使用`drawString()`方法，该方法等同于处理文本文件时使用的`file_obj.write()`方法。最后，由`showPage()`方法停止绘制，使用`save()`方法创建实际的PDF文件。如果运行这段代码，会得到一个空白的PDF文件，在文件底部有“Hello World”字样。

如果已经下载了ReportLab源码版本，就可以使用其中包含的测试作为示例驱动文档。也就是说，运行测试时，会生成一系列PDF文件，你可以将这些PDF与测试代码进行比较，来查看通过ReportLab库如何成功获得各种各样的可视化效果。

www.TopSage.com

现在已经演示了如何使用ReportLab来创建PDF，接下来，让我们看看如何使用ReportLab来创建一个自定义磁盘使用情况报告。这样的报告是非常有用的。参见例4-4。

例4-4：PDF磁盘报告

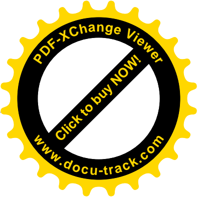
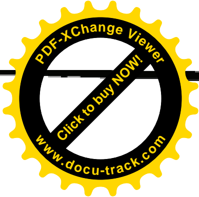
```
#!/usr/bin/env python
import subprocess
import datetime
from reportlab.pdfgen import canvas
from reportlab.lib.units import inch

def disk_report():
    p = subprocess.Popen("df -h", shell=True,
                         stdout=subprocess.PIPE)
    return p.stdout.readlines()


def create_pdf(input,output="disk_report.pdf"):
    now = datetime.datetime.today()
    date = now.strftime("%h %d %Y %H:%M:%S")
    c = canvas.Canvas(output)
    textobject = c.beginText()
    textobject.setTextOrigin(inch, 11*inch)
    textobject.textLines('''
    Disk Capacity Report: %s
    ''' % date)
    for line in input:
        textobject.textLine(line.strip())
    c.drawText(textobject)
    c.showPage()
    c.save()
report = disk_report()
create_pdf(report)
```

这段代码会产生一个报告来显示当前磁盘的使用情况，包括时间戳和“Disk Capacity Report”（磁盘容量报告）字样。完成这样一个功能仅用了如此少的几行代码，肯定会给你留下深刻印象。接下来看一下本示例中的一些亮点。首先，disk_report()函数只是简单地获得df -h命令的输出，并将其作为列表返回。接下来，在create_pdf()函数中创建了一个格式化的时间戳。而本示例中最重要的部分是textobject函数。

textobject函数用于创建一个放在PDF中的对象。我们先通过调用beginText()创建了一个textobject对象。然后，定义了数据打包的方法。这里使用的PDF大约为8.5×11英寸的文档，因此为了能从页面的最顶端进行打包，我们告诉text对象在11英寸处设置文本边界。之后，通过将字符串写入文本对象来创建标题，最后对df命令的执行结果的每一行进行迭代处理。值得注意的是这里使用line.strip()来删除换行字符。如果没有这样做，在换行符处会看到黑色方块。

通过添加颜色和图片，可以创建更为复杂的PDF文件。至于如何实现，可以通过阅读

www.TopSage.com

ReportLab PDF库中非常不错的用户指南得到答案。通过这些示例可以看到其中最重要的部分是文本对象，文本对象可以收集和保留数据并最终输出数据。

信息发布

在获得并格式化数据之后，需要将其分发给对其感兴趣的人。这一节将主要讲述如何被文档通过email发送给其接收者。如果需要将文档发送到web服务器供用户查看，可以使用FTP。下一章将讨论Python标准FTP模块的使用。

发送email

处理email是系统管理的一个重要内容。不仅要管理email服务器，还需要通过email来产生警告和报警信息。Python标准库对于发送email提供了强有力的支持，只是我们之前提及的比较少。由于所有的系统管理员都对自动发送email感兴趣，本节将向你展示如何使用Python来完成各种各样的email任务。

发送基本信息

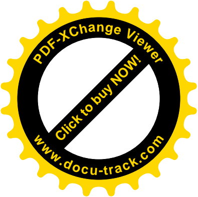
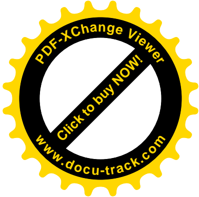
在Python中有两个不同的包允许你发送电子邮件。一个是低级别的包smtplib，是针对RFC中关于SMTP协议的各种描述的接口。它可以发送email。另一个包是email，帮助解析和产生email。例4-5中使用smtplib建立一个包含邮件信息的字符串，然后使用email包将邮件发送至邮件服务器。

例4-5：使用SMTP发送消息

```
#!/usr/bin/env python

import smtplib
mail_server = 'localhost'
mail_server_port = 25
from_addr = 'sender@example.com'
to_addr = 'receiver@example.com'

from_header = 'From: %s\r\n' % from_addr
to_header = 'To: %s\r\n\r\n' % to_addr
subject_header = 'Subject: nothing interesting'

body = 'This is a not-very-interesting email.'

email_message = '%s\n%s\n%s\n\n%s' % (from_header, to_header, subject_header, body)

s = smtplib.SMTP(mail_server, mail_server_port)
s.sendmail(from_addr, to_addr, email_message)
s.quit()
```

首先，我们定义了email服务器的主机和端口号，也定义了“to”和“from”地址。然后，通过连接邮件头与邮件体建立了email邮件。最后，连接到SMTP服务器，并从

www.TopSage.com

from_addr发送到to_addr。应该注意到的是，为了与RFC规定相兼容，这里使用“\r\n”专门对From:和To:进行了格式化。

在第10章的“调度Python进程”部分有一个创建cron作业的代码示例，该示例使用Python实现了邮件的自动发送。现在，让我们从这一简单的基本示例出发，转到使用Python中的电子邮件可以完成的更有趣的一些事情上。

使用SMTP认证

上一个示例非常简单，对于使用Python发送email来说简单得有些微不足道。但不幸的是，有一些SMTP服务器会强迫你使用认证，因此上面的示例在许多情况下无法完成相应工作。例4-6是一个包含了SMTP认证的示例。

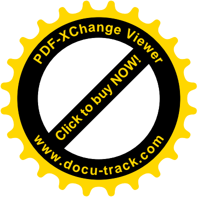
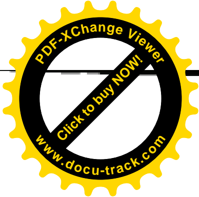
例4-6：SMTP认证

```
#!/usr/bin/env python
import smtplib
mail_server = 'smtp.example.com'
mail_server_port = 465

from_addr = 'foo@example.com'
to_addr = 'bar@exmaple.com'

from_header = 'From: %s\r\n' % from_addr
to_header = 'To: %s\r\n\r\n' % to_addr
subject_header = 'Subject: Testing SMTP Authentication'

body = 'This mail tests SMTP Authentication'

email_message = '%s\n%s\n%s\n\n%s' % (from_header, to_header, subject_header, body)

s = smtplib.SMTP(mail_server, mail_server_port)
s.set_debuglevel(1)
s.starttls()
s.login("fatalbert", "mysecretpassword")
s.sendmail(from_addr, to_addr, email_message)
s.quit()
```

其中，主要的差别在于定义了用户名和密码，启动了debuglevel，然后通过使用starttls()方法启动了SSL。使用认证时启动debugging是一个非常好的做法。如果我们查看一下失败的调试会话，会看到如下内容：

```
$ python2.5 mail.py
send: 'ehlo example.com\r\n'
reply: '250-example.com Hello example.com [127.0.0.1], pleased to meet you\r\n'
reply: '250-ENHANCEDSTATUSCODES\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-8BITMIME\r\n'
reply: '250-SIZE\r\n'
reply: '250-DSN\r\n'
reply: '250-ETRN\r\n'
```


www.TopSage.com


```
reply: '250-DELIVERBY\r\n'
reply: '250 HELP\r\n'
reply: retcode (250); Msg: example.com example.com [127.0.0.1], pleased to meet you
ENHANCEDSTATUSCODES
PIPELINING
8BITMIME
SIZE
DSN
ETRN
DELIVERBY
HELP
send: 'STARTTLS\r\n'
reply: '454 4.3.3 TLS not available after start\r\n'
reply: retcode (454); Msg: 4.3.3 TLS not available after start
```

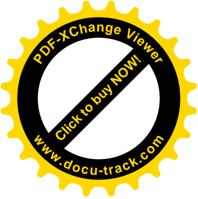
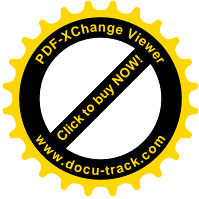
在这个示例中，我们试图用来初始化SSL的服务器没有提供对SSL的支持，因此将我们直接拒绝。类似这种问题的处理方法十分简单，在试图使用级联服务器系统来发送电子邮件时，通过命令localhost attempt to send mail，许多其他潜在的问题通过编写包含错误处理代码的脚本可以简单地解决。

使用Python发送附件

仅发送包含文本的邮件已经跟不上潮流了。使用Python我们可以使用MIME标准来发送信息，它允许对邮件的附件进行编码。在本章前面部分，我们介绍了如何创建PDF报表。由于系统管理是非常需要耐心的，这里跳过了令人乏味的对MIME起源的介绍，直接来到发送带附件的邮件部分。参见例4-7。

例4-7：发送带PDF附件的email

```
import email
from email.MIMEText import MIMEText
from email.MIMEMultipart import MIMEMultipart
from email.MIMEBase import MIMEBase
from email import encoders
import smtplib
import mimetypes

from_addr = 'noah.gift@gmail.com'
to_addr = 'jjinux@gmail.com'
subject_header = 'Subject: Sending PDF Attachemt'
attachment = 'disk_usage.pdf'
body = '''
This message sends a PDF attachment created with Report
Lab.
'''

m = MIMEMultipart()
m["To"] = to_addr
m["From"] = from_addr
m["Subject"] = subject_header
```

```
ctype, encoding = mimetypes.guess_type(attachment)
print ctype, encoding
maintype, subtype = ctype.split('/', 1)
print maintype, subtype

m.attach(MIMEText(body))
fp = open(attachment, 'rb')
msg = MIMEBase(maintype, subtype)
msg.set_payload(fp.read())
fp.close()
encoders.encode_base64(msg)
msg.add_header("Content-Disposition", "attachment", filename=attachment)
m.attach(msg)

s = smtplib.SMTP("localhost")
s.set_debuglevel(1)
s.sendmail(from_addr, to_addr, m.as_string())
s.quit()
```

在这里，我们使用了一些小技巧，对之前创建的磁盘报告PDF文件进行编码，然后通过电子邮件发送出去。

Trac

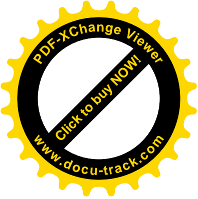
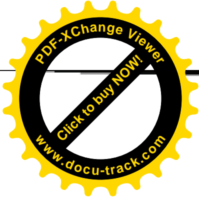
Trac是一个wiki和问题追踪系统，其典型应用是软件开发。但是只要你想使用wiki或是订票系统，就可以使用Trac。Trac在Python下编写。你可以在*http://trac.edgewall.org/*找到Trac的文档及安装包。这里不打算涉及Trac的更多细节，因为这超出了本书的范围。但是它对解决订票系统中的一般性故障，是非常好的工具。Trac的另一个非常有趣的地方是它可以通过插件来扩展。

之所以在此提到Trac，是因为它非常适合我们所讨论的所有三个类别：信息收集，格式化和发布。wiki允许用户通过浏览器创建web页面。他们添加的信息以HTML格式显示，这有利于其他用户通过浏览器查看。这是在这一章中多次讨论过的。

类似地，订票系统允许用户根据工作的需要提出请求，或是报告他们遇到的问题。你可以通过web界面报告输入的票数，甚至产生一个CSV报告。再说一次，Trac涵盖了这一章讨论的所有内容。

我们建议你对Trac是否可以满足自己的需要做进一步的尝试。或许你需要一些具有更多特征和性能的工具，或许你需要一些更简单的工具，不管哪种情况这确实是值得做更多尝试的事情。

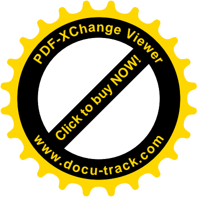
www.TopSage.com

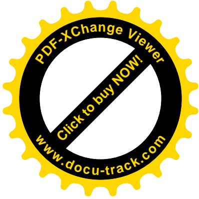



本章小结

在本章中，我们介绍了自动和手工进行数据收集的方法。也介绍了如何将数据整合为各种不同的更适合发布的格式，如HTML、PDF和PNG。最后，我们查看了将信息传送到对其感兴趣的用户手中的方法。正如我们在本章开始时所说的，处理文档或许不是你所有工作中最吸引眼球的一项，甚至在记录文件时，你可能还没有意识到是在对文档进行处理。但是简单明了的文档是系统管理的一个重要内容。我们希望本章中的一些技巧可以使枯燥的文档处理工作变得更有生趣一些。

www.TopSage.com





第5章

网络

说到网络，通常会涉及对多台计算机进行连接，保证它们之间可以相互通信。但是我们更感兴趣的不是计算机之间的相互通信，而是进程之间的相互通信。对于我们将要介绍的技术而言，进程是在同一台计算机上还是在不同的计算机上无关紧要。

本章集中介绍如何使用标准的socket库（也可能是建立在socket基础之上的其他库）编写Python程序，连接其他进程，并与其他进程进行交互。

网络客户端

在客户端连接到服务器之前，服务器一直处于等待状态，因此，连接是由客户端发起的。Python标准库中包括了许多已经实现的网络客户端代码。这一章，我们将讨论一些更为通用的、经常被使用的客户端。

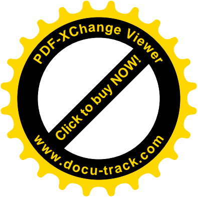
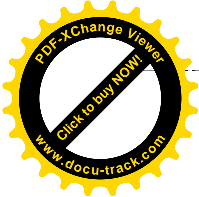
socket

socket模块为操作系统的socket连接提供了一个Python接口。这表示使用Python，可以完成任何使用socket或对socket进行处理的操作。如果之前没有做过任何网络编程工作，本章将会给出一个网络编程的简要说明。这会给你一个大体印象，即使用Python的networking库都可以完成哪些操作。

socket模块提供了工厂函数socket()。socket()函数会返回一个socket对象。为了定义socket的类型，需要传递给socket()一些参数。如果不带参数调用socket()工厂函数，其返回的socket对象默认使用TCP/IP协议：

```
In [1]: import socket

In [2]: s = socket.socket()

In [3]: s.connect(('192.168.1.15', 80))
```


www.TopSage.com

```
In [4]: s.send("GET / HTTP/1.0\n\n")
Out[4]: 16

In [5]: s.recv(200)
Out[5]: 'HTTP/1.1 200 OK\r\n\
Date: Mon, 03 Sep 2007 18:25:45 GMT\r\n\
Server: Apache/2.0.55 (Ubuntu) DAV/2 PHP/5.1.6\r\n\
Content-Length: 691\r\n\
Connection: close\r\n\
Content-Type: text/html; charset=UTF-8\r\n\
\r\n\
<!DOCTYPE HTML P'
In [6]: s.close()
```

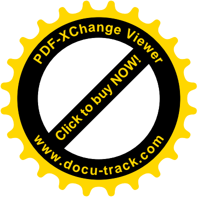
该示例通过socket()工厂函数创建一个名为s的socket对象。它连接到本地默认的web服务器，指定端口号为HTTP默认的端口号80。接下来，它将文本字符串"GET / HTTP/1.0\n\n"（一个简单的HTTP请求）发送到服务器。发送完毕之后，它会收到服务器响应的前200个字节，这200字节包含状态信息和HTTP头信息。最后，关闭连接。

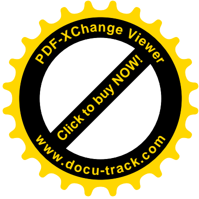
本例中演示的socket方法可能是你最常使用的方法。connect()用于在socket对象与远程对象（即“不是当前的socket对象”）之间建立通信连接。send()用于从socket对象向远端发送数据。recv()用于接收远端发送的任何数据。close()则用于关闭两个socket之间的连接。这一简单的示例表明了创建socket对象，并通过该对象发送和接收数据是非常容易的。

现在，让我们看一个更有用的示例。假定有一个运行某些网络应用程序的服务器，例如一个web服务器。现在希望查看该服务器，以确认在经历了一整天运行之后，仍可以创建一个连接到该web服务器上的socket连接。这几乎是一个最小规模的监测程序，但是却可以证明这个web服务器仍在工作，并且它仍旧在侦听某些端口。参见例5-1。

例5-1：TCP端口检查

```
#!/usr/bin/env python

import socket
import re
import sys

def check_server(address, port):
    #create a TCP socket
    s = socket.socket()
    print "Attempting to connect to %s on port %s" % (address, port)
    try:
        s.connect((address, port))
        print "Connected to %s on port %s" % (address, port)
        return True
    except socket.error, e:
        print "Connection to %s on port %s failed: %s" % (address, port, e)
        return False
```





```
if __name__ == '__main__':
    from optparse import OptionParser
    parser = OptionParser()

    parser.add_option("-a", "--address", dest="address", default='localhost',
                      help="ADDRESS for server", metavar="ADDRESS")

    parser.add_option("-p", "--port", dest="port", type="int", default=80,
                      help="PORT for server", metavar="PORT")

(options, args) = parser.parse_args()
print 'options: %s, args: %s' % (options, args)
check = check_server(options.address, options.port)
print 'check_server returned %s' % check
sys.exit(not check)
```

所有的工作都由check_server()函数完成的。check_server()先创建了一个socket对象，之后试图连接到指定的地址和端口号。如果成功，则返回值为真。如果失败，socket.connect()调用会抛出一个可操作的异常，函数返回值为假。代码中的main部分调用check_server()。main部分解析用户的参数，并将用户请求的参数转化为适当的格式传递给check_server()。整个脚本代码在执行过程中都能够输出状态信息。输出的最后一项数据是check_server()的返回值。最后，脚本将与check_server()返回值相反的值返回给shell。返回与check_server()返回值相反的值使得该脚本成为一个极为有用的工具。成功时返回0值到shell，失败时返回非0值（正数）到shell，这是一种非常典型的用法。以下是一个成功连接到web服务器的代码示例：

```
jmjones@dinkgutsy:code$ python port_checker_tcp.py -a 192.168.1.15 -p 80
options: {'port': 80, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 80
Connected to 192.168.1.15 on port 80
check_server returned True
```

最后一行输出check_server returned True，表示成功建立连接。

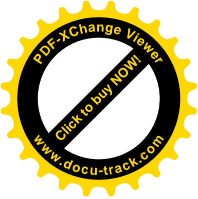
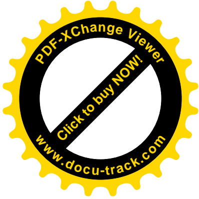
下面是一个失败的连接示例：

```
jmjones@dinkgutsy:code$ python port_checker_tcp.py -a 192.168.1.15 -p 81
options: {'port': 81, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 81
Connection to 192.168.1.15 on port 81 failed: (111, 'Connection refused')
check_server returned False
```

日志的最后一行是check_server returned False，这表明连接失败。倒数第二行的输出为 Connection to192.168.1.15 on port 81 failed，是因为“Connection refused”（连接被拒绝）。大胆地猜测一下，可能与这台服务器中没有在81端口守护的进程有关。

现在已经创建了三个示例来展示如何在shell脚本中使用socket工具。首先，给出运行

www.TopSage.com

脚本的shell命令，如果脚本成功执行，则输出SUCCESS。这里用&操作符来代替if-then语句：

```
$ python port_checker_tcp.py -a 192.168.1.15 -p 80 && echo "SUCCESS"
options: {'port': 80, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 80
Connected to 192.168.1.15 on port 80
check_server returned True
SUCCESS
```

可以看到，该脚本成功执行，因此，在执行并输出状态结果后，shell输出SUCCESS。

```
$ python port_checker_tcp.py -a 192.168.1.15 -p 81 && echo "FAILURE"
options: {'port': 81, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 81
Connection to 192.168.1.15 on port 81 failed: (111, 'Connection refused')
check_server returned False
```

该脚本执行失败，但不会输出FAILURE：

```
$ python port_checker_tcp.py -a 192.168.1.15 -p 81 && echo "FAILURE"
options: {'port': 81, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 81
Connection to 192.168.1.15 on port 81 failed: (111, 'Connection refused')
check_server returned False
FAILURE
```

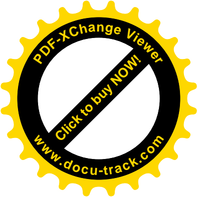
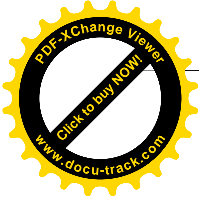
这个脚本执行失败，但是我们将&&变为||，这意味着如果脚本返回结果为假，将打印输出FAILURE。可以看到，它确实是这样执行的。

事实上，web服务器允许连接80端口，并不意味着存在可供连接使用的HTTP服务器。可以通过一个测试帮助我们准确地判断web服务器的状态，该测试可以检测是否产生HTTP头以及一些特定的URL状态代码。例5-2就实现了这样一个测试。代码如下所示：

例5-2：基于Socket的web服务器检测

```
#!/usr/bin/env python

import socket
import re
import sys


def check_webserver(address, port, resource):
    #build up HTTP request string
    if not resource.startswith('/'):
    resource = '/' + resource
request_string = "GET %s HTTP/1.1\r\nHost: %s\r\n\r\n" % (resource, address)
print 'HTTP request:'
print '|||%s|||' % request_string
```

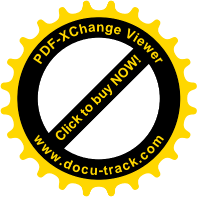
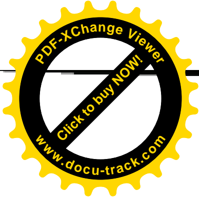
www.TopSage.com

```
#create a TCP socket
s = socket.socket()
print "Attempting to connect to %s on port %s" % (address, port)
try:
    s.connect((address, port))
    print "Connected to %s on port %s" % (address, port)
    s.send(request_string)
    #we should only need the first 100 bytes or so
    rsp = s.recv(100)
    print 'Received 100 bytes of HTTP response'
    print '|||%s|||' % rsp
except socket.error, e:
    print "Connection to %s on port %s failed: %s" % (address, port, e)
    return False
finally:
    #be a good citizen and close your connection
    print "Closing the connection"
    s.close()
lines = rsp.splitlines()
print 'First line of HTTP response: %s' % lines[0]
try:
    version, status, message = re.split(r'\s+', lines[0], 2)
    print 'Version: %s, Status: %s, Message: %s' % (version, status, message)
except ValueError:
    print 'Failed to split status line'
    return False
if status in ['200', '301']:
    print 'Success - status was %s' % status
    return True
else:
    print 'Status was %s' % status
    return False

if __name__ == '__main__':
    from optparse import OptionParser
    parser = OptionParser()
    parser.add_option("-a", "--address", dest="address", default='localhost',
                      help="ADDRESS for webserver", metavar="ADDRESS")

    parser.add_option("-p", "--port", dest="port", type="int", default=80,
                      help="PORT for webserver", metavar="PORT")

    parser.add_option("-r", "--resource", dest="resource", default='index.html',
                      help="RESOURCE to check", metavar="RESOURCE")

    (options, args) = parser.parse_args()
    print 'options: %s, args: %s' % (options, args)
    check = check_webserver(options.address, options.port, options.resource)
    print 'check_webserver returned %s' % check
    sys.exit(not check)
```

与之前使用check_server()完成所有工作的示例类似，本示例中使用check_webserver()来完成所有工作。首先，check_webserver()建立HTTP请求字符串。如果不了解HTTP，可以把HTTP协议理解为一种已定义的HTTP客户端与服务器进行通信的方法。check_

www.TopSage.com

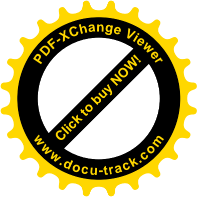
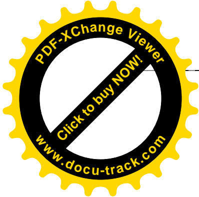
webserver()建立的HTTP请求几乎是最简单的HTTP请求。接下来，check_webserver()创建一个socket对象，连接到服务器，并向服务器发送HTTP请求。之后，它读取从服务器返回的响应并关闭连接。出现socket错误时，check_webserver()返回False，表示检测失败。它会取出从服务器读取的信息，并从中提出状态代码。如果状态码是表示OK的200，或是表示永久移动的301，check_webserver()都会返回True，否则，返回False。脚本中的main部分解析用户输入，并调用 check_webserver()。在从check_webserver()取得结果后，它向shell返回与check_webserver()返回值相反的值，这个与之前使用普通socket的检测代码相似。我们希望能够从shell脚本调用该方法，并且查看是否成功。下面是执行代码的示例：

```
$ python web_server_checker_tcp.py -a 192.168.1.15 -p 80 -r apache2-default
options: {'resource': 'apache2-default', 'port': 80, 'address':
'192.168.1.15'}, args: []
HTTP request:
|||GET /apache2-default HTTP/1.1
Host: 192.168.1.15

|||
Attempting to connect to 192.168.1.15 on port 80
Connected to 192.168.1.15 on port 80
Received 100 bytes of HTTP response
|||HTTP/1.1 301 Moved Permanently
Date: Wed, 16 Apr 2008 23:31:24 GMT
Server: Apache/2.0.55 (Ubuntu) |||
Closing the connection
First line of HTTP response: HTTP/1.1 301 Moved Permanently
Version: HTTP/1.1, Status: 301, Message: Moved Permanently
Success - status was 301
check_webserver returned True
```

最后四行输出表示在这个Web服务器上，HTTP/apache2-default的默认状态码为301，这说明运行是成功的。

下面是另一次运行。这一次，我们特意制定一个并不存在的资源，以查看HTTP调用失败时的显示结果：

```
$ python web_server_checker_tcp.py -a 192.168.1.15 -p 80 -r foo
options: {'resource': 'foo', 'port': 80, 'address': '192.168.1.15'}, args: []
HTTP request:
|||GET /foo HTTP/1.1
Host: 192.168.1.15

|||
Attempting to connect to 192.168.1.15 on port 80
Connected to 192.168.1.15 on port 80
Received 100 bytes of HTTP response
|||HTTP/1.1 404 Not Found
Date: Wed, 16 Apr 2008 23:58:55 GMT
Server: Apache/2.0.55 (Ubuntu) DAV/2 PH|||
```

```
Closing the connection
First line of HTTP response: HTTP/1.1 404 Not Found
Version: HTTP/1.1, Status: 404, Message: Not Found
Status was 404
check_webserver returned False
```

之前示例代码中，最后四行显示代码被成功执行，而这个示例代码的最后四行却表明它没有成功执行。由于在web服务器上不存在/foo，检测程序返回False。

本节主要介绍如何构建底层网络服务器连接，并实现基本的检测功能。通过一系列示例，展示了客户端与服务器进行通讯时有可能出现的不同场景。如果你有机会编写网络组件，应该使用高于socket模块的其他库。实际上，在真正编写网络组件的时候，并不需要在类似socket这样的底层库上花费太多时间。

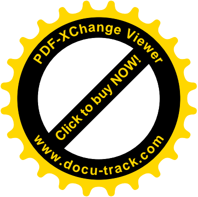
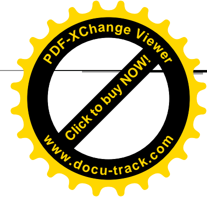
httplib

先前的示例演示了如何直接使用socket模块创建一个HTTP请求。接下来的示例将演示如何使用httplib模块。我们首先考虑的是，什么时候应该使用httplib模块而不是socket模块呢？或者说，什么时候应该使用更高层的库而不是较低层的库呢？真正的经验是根据具体情况而定。有时候，我们需要使用较低层的库。比如当我们需要完成一些在可用的库中找不到的任务时，或是需要对库中的任务进行细粒度的控制时，或是需要有更出色的性能时。但是在本示例中，我们没有任何理由不使用httplib这样的高级库而使用socket这样的低级库。例5-3实现了与之前示例相同的功能，只是这里使用httplib模块完成。

例5-3：基于httplib的web服务器检测

```
#!/usr/bin/env python

import httplib
import sys


def check_webserver(address, port, resource):
    #create connection
    if not resource.startswith('/'):
        resource = '/' + resource
    try:
        conn = httplib.HTTPConnection(address, port)
        print 'HTTP connection created successfully'
        #make request
        req = conn.request('GET', resource)
        print 'request for %s successful' % resource
        #get response
        response = conn.getresponse()
        print 'response status: %s' % response.status
```

www.TopSage.com

```
    except sock.error, e:
        print 'HTTP connection failed: %s' % e
        return False
    finally:
        conn.close()
        print 'HTTP connection closed successfully'
    if response.status in [200, 301]:
        return True
    else:
        return False

if __name__ == '__main__':
    from optparse import OptionParser
    parser = OptionParser()
    parser.add_option("-a", "--address", dest="address", default='localhost',
                      help="ADDRESS for webserver", metavar="ADDRESS")
    parser.add_option("-p", "--port", dest="port", type="int", default=80,
                      help="PORT for webserver", metavar="PORT")
    parser.add_option("-r", "--resource", dest="resource", default='index.html',
                      help="RESOURCE to check", metavar="RESOURCE")
    (options, args) = parser.parse_args()
    print 'options: %s, args: %s' % (options, args)
    check = check_webserver(options.address, options.port, options.resource)
    print 'check_webserver returned %s' % check
    sys.exit(not check)
```

本示例与socket示例非常相似。两者最大的差异是你没必要手动创建HTTP请求，也不必手动解析HTTP响应。httplib连接对象具有request()方法，该方法能够建立和发送HTTP请求。connection对象也有一个getresponse()方法，该方法可以创建一个响应对象。可以通过引用响应对象的状态（status）属性访问HTTP的状态。虽然这并没有少写代码，但无须我们手动创建、发送和接收HTTP请求和响应，减少了不必要的麻烦。而且，这段代码看起来也让人觉得更整洁一些。

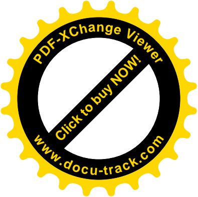
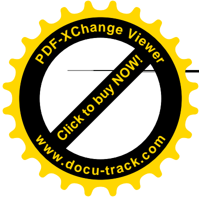
下面的示例中，我们使用了与之前示例成功执行时所使用的相同的命令行参数在web服务器上寻找/，并且找到了：

```
$ python web_server_checker_httplib.py -a 192.168.1.15 -r /
options: {'resource': '/', 'port': 80, 'address': '192.168.1.15'}, args: []
HTTP connection created successfully
request for / successful
response status: 200
HTTP connection closed successfully
check_webserver returned True
```

下面的示例中，我们使用了与之前示例执行失败时所使用的相同的命令行参数寻找/foo目录，但是没有找到。

```
$ python web_server_checker_httplib.py -a 192.168.1.15 -r /foo
options: {'resource': '/foo', 'port': 80, 'address': '192.168.1.15'}, args: []
HTTP connection created successfully
```

www.TopSage.com

```
request for /foo successful
response status: 404
HTTP connection closed successfully
check_webserver returned False
```

正如之前所说，如果有机会使用高级库，一定要使用高级库。使用httplib而不是单独使用socket模块会使代码更简洁、清晰。而代码越简洁，其中的bug就越少。

ftplib

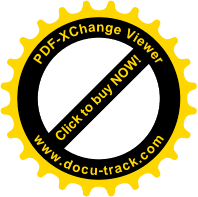
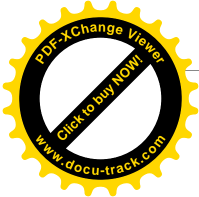
除了socket和httplib模块之外，Python标准库还包含了一个名为ftplib的FTP客户端模块。ftplib是一个全功能（full-featured）的FTP客户端库，它可让你以编程方式执行任何通常会使用FTP客户端应用程序来执行的任务。例如，可以登录FTP服务器，列出指定目录中的文件，下载文件、上传文件、更改目录、退出，所有这一切都可以通过Python脚本来完成。你甚至可以使用许多在Python中可用的GUI框架，建立属于自己的GUI FTP应用程序。

在这里，我们没有对该库进行全面的介绍，仅展示了例5-4，然后对该示例进行解析。

例5-4：使用ftplib实现FTP URL retriever

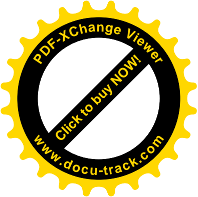
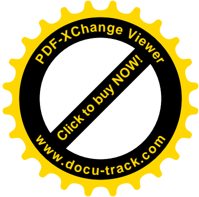
```
#!/usr/bin/env python

from ftplib import FTP
import ftplib
import sys
from optparse import OptionParser

parser = OptionParser()

parser.add_option("-a", "--remote_host_address", dest="remote_host_address",
    help="REMOTE FTP HOST.",
    metavar="REMOTE FTP HOST")

parser.add_option("-r", "--remote_file", dest="remote_file",
    help="REMOTE FILE NAME to download.",
    metavar="REMOTE FILE NAME")

parser.add_option("-l", "--local_file", dest="local_file",
    help="LOCAL FILE NAME to save remote file to", metavar="LOCAL FILE NAME")

parser.add_option("-u", "--username", dest="username",
    help="USERNAME for ftp server", metavar="USERNAME")

parser.add_option("-p", "--password", dest="password",
    help="PASSWORD for ftp server", metavar="PASSWORD")

(options, args) = parser.parse_args()

if not (options.remote_file and
        options.local_file and
        options.remote_host_address):
```


www.TopSage.com

```
    parser.error('REMOTE HOST, LOCAL FILE NAME, ' \
            'and REMOTE FILE NAME are mandatory')

if options.username and not options.password:
    parser.error('PASSWORD is mandatory if USERNAME is present')

ftp = FTP(options.remote_host_address)
if options.username:
    try:
        ftp.login(options.username, options.password)
    except ftplib.error_perm, e:
        print "Login failed: %s" % e
        sys.exit(1)
else:
    try:
        ftp.login()
    except ftplib.error_perm, e:
        print "Anonymous login failed: %s" % e
        sys.exit(1)
try:
    local_file = open(options.local_file, 'wb')
    ftp.retrbinary('RETR %s' % options.remote_file, local_file.write)
finally:
    local_file.close()
    ftp.close()
```

代码的起始部分（跳过所有的命令行解析）创建了一个FTP对象，该对象通过将FTP服务器的地址传递给FTP构造器（constructor）来实现。另一种可选择的方法是，创建FTP对象，但不向构造器传递参数，创建之后再调用connect()方法，而connect()方法需要指定FTP服务器地址。之后，登录到FTP服务器。如果提供了用户名和密码，则以该用户名和密码登录。如果没有提供，则使用匿名登录。接下来，创建了一个文件对象保存FTP服务器上的文件数据。之后，调用FTP对象的retrbinary()方法。retrbinary()方法，正如其名称所表达的含义，表示从FTP服务器上获得一个二进制文件。该方法需要两个参数：FTP的retrieve命令和一个回调（callback）函数。你或许注意到，这里的回调函数是在前一步骤中创建的文件对象的write方法。值得注意的是，在本示例中没有调用write()方法。我们将write方法传入retrbinary()方法，这样retrbinary()就可以调用write()。retribinary()会连同传递给它的从FTP服务器上获得的数据块调用传递给它的任何回调函数。回调函数可以对数据进行任何处理。这里，该回调函数仅对它从FTP服务器收到的字节数进行记录。传递一个file对象的write方法能够将脚本从FTP服务器上获得的文件内容写入到file对象中。最后，关闭文件对象和FTP连接。这一处理过程存在着一些疏漏：我们在FTP服务器上获取文件的代码中建立了一个try块，在关闭本地文件和FTP连接的代码中使用了finally块。这样，如果有错误发生，就会在脚本结束之前试图清除文件。附录提供了一个对回调函数的简明介绍，仅供参考。

urllib

urllib位于标准库模块的更高层。看到urllib时，我们很容易会想到HTTP库，而忘记了FTP资源也是可以通过URL来识别的。因此，或许你从没有想过使用urllib来获取FTP资源，但它的确具备这一功能。例5-5与之前的ftplib示例实现的功能相同，只是这里使用了urllib。

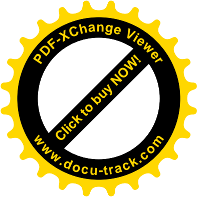
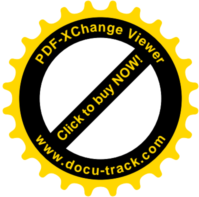
例5-5：使用urllib实现FTP URL retriever

```
#!/usr/bin/env python
"""
url retriever

Usage:

url_retrieve_urllib.py URL FILENAME

URL:
If the URL is an FTP URL the format should be:
ftp://[username[:password]@]hostname/filename
If you want to use absolute paths to the file to download,
you should make the URL look something like this:
ftp://user:password@host/%2Fpath/to/myfile.txt
Notice the '%2F' at the beginning of the path to the file.

FILENAME:
absolute or relative path to the filename to save downloaded file as
"""

import urllib
import sys

if '-h' in sys.argv or '--help' in sys.argv:
    print __doc__
    sys.exit(1)

if not len(sys.argv) == 3:
    print 'URL and FILENAME are mandatory'
    print __doc__
    sys.exit(1)
url = sys.argv[1]
filename = sys.argv[2]
urllib.urlretrieve(url, filename)
```

这一段脚本简短而且亲切，它展示了urllib的强大之处。事实上，其间有很多是使用文档而并非代码，甚至注释都比代码多。我们使用该脚本完成了一个非常简单的参数解析过程。由于两个选项是必须的，因此我们需要指定具体的参数。在该示例中，仅有的有效代码行如下所示：

```
    urllib.urlretrieve(url, filename)
```


www.TopSage.com

在获得sys.argv选项之后，代码下载指定的URL，并保存为指定的本地文件名。使用HTTP URL和FTP URL都可以，甚至在URL中包含用户名和密码也可以。

需要强调的一点是，你有可能认为应该有比使用其他语言完成这些功能更容易一些的方法，的确如此。在Python标准中应该会有一些更高级的库，可以完成经常要做的工作。而在本示例中，urllib已经能够准确地执行我们想做的事，因此，无须使用更多的库。有时候，Python标准库可能不能满足需要，但可以找到其他的Python资源，如*http://pypi.python.org/pypi*提供的Python包索引（PyPI）。

urllib2

另外一个高级库是urllib2。urllib2包含了很多与urllib相似的功能，是对urllib的扩展。例如，urllib2能够更好地支持认证和cookie。因此，当你发现urllib无法完成一些任务时，应该查看一下urllib2，看看它是否能够满足需要。

远程过程调用

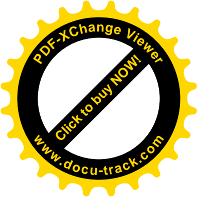
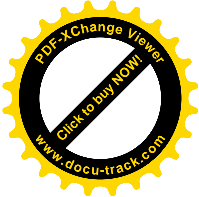
网络编程的典型目的就是实现进程间的通信（IPC）。通常情况下，简单的IPC使用HTTP或是socket就足够用了。但是，有些时候，需要在不同进程或者是不同计算机之间执行代码，这时，IPC能够让你有在同一进程中执行代码的感觉。事实上，如果可以在Python程序中远程执行一些代码，你或许会希望远程调用可以返回Python对象，这样你就可以更容易地进行处理，而不是返回一大堆文本后再进行手工解析。值得高兴的是，有许多能够执行RPC（远程过程调用）功能的工具可供使用。

XML-RPC

XML-RPC在两个进程之间交换指定格式的XML文档，以实现远程过程调用。但是这里不需要考虑XML的问题，很可能你根本就不需要知道两个进程间所交换的文档的具体格式。要使用XML-RPC，只需要知道在Python标准库中已经有了客户端和服务器端的实现。此外，XML-RPC在大多数编程语言中都可以使用，而且使用起来非常简单。例5-6是一个简单的XML-RPC服务器。

例5-6：简单 XML-RPC 服务器

```
#!/usr/bin/env python

import SimpleXMLRPCServer
import os

def ls(directory):
    try:
        return os.listdir(directory)
```

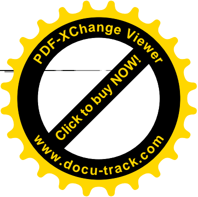
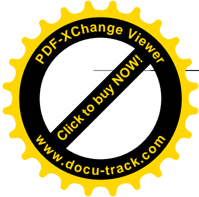



```
    except OSError:
        return []

def ls_boom(directory):
    return os.listdir(directory)

def cb(obj):
    print "OBJECT::", obj
    print "OBJECT.__class__::", obj.__class__
    return obj.cb()

if __name__ == '__main__':
    s = SimpleXMLRPCServer.SimpleXMLRPCServer(('127.0.0.1', 8765))
    s.register_function(ls)
    s.register_function(ls_boom)
    s.register_function(cb)
    s.serve_forever()
```

该代码创建了一个新的SimpleXMLRPCServer对象，并将其绑定到回环地址127.0.0.1的端口8765上，这使得该对象只能访问指定机器上的进程。然后对已定义的函数ls()、ls_boom()和cb()进行注册。稍后我们会对cb()函数进行介绍。ls()函数会列出使用os.listdir()传递给它的目录中的所有内容，并且以列表的方式返回结果。ls()会屏蔽任何OSError异常。ls_boom()可以将异常返回到XML-RPC客户端。接下来，程序进入serve_forever()循环，该循环等待可以处理的连接。下面是上述代码在IPython shell中使用的示例。

```
In [1]: import xmlrpclib

In [2]: x = xmlrpclib.ServerProxy('http://localhost:8765')

In [3]: x.ls('.')
Out[3]:
['.svn',
 'web_server_checker_httplib.py',
....
 'subprocess_arp.py',
 'web_server_checker_tcp.py']

In [4]: x.ls_boom('.')
Out[4]:
['.svn',
 'web_server_checker_httplib.py',
....
 'subprocess_arp.py',
 'web_server_checker_tcp.py']

In [5]: x.ls('/foo')
Out[5]: []

In [6]: x.ls_boom('/foo')
---------------------------------------------------------------------------
<class 'xmlrpclib.Fault'>                 Traceback (most recent call last)
...
.
```

www.TopSage.com

```
.
<<big nasty traceback>>
.
.
...
    786         if self._type == "fault":
--> 787             raise Fault(**self._stack[0])
    788         return tuple(self._stack)
    789

<class 'xmlrpclib.Fault'>: <Fault 1: "<type 'exceptions.OSError'>
    :[Errno 2] No such file or directory: '/foo'">
```

首先，通过传递XML-RPC服务器地址来创建一个ServerProxy()对象。然后，调用.ls('.')来查看服务器当前工作目录中有哪些文件。服务器在保存有示例代码的目录下运行，因此，你可以从该目录列表中看到这些文件。真正有趣的事是在客户端这一边，x.ls('.')返回一个Python列表。如果服务器是由Java、Perl、Ruby或是C#来实现的，也会有相同情况。实现服务器的语言可以执行列目录，创建列表，数组或是文件名集合；XML-RPC服务器代码可以用XML格式来表示创建的列表或数组，并通过连接客户端的线程将数据返回。这里也对ls_boom()进行了测试。由于ls_boom()缺少ls()的异常处理，可以看到异常从服务器传回了客户端。我们甚至在客户端也可以进行追踪。

XML-RPC带来了交互操作的可能性，非常有意义。但更有意义的是，可以写一个能够在多台机器上运行的代码，并根据需要远程执行。

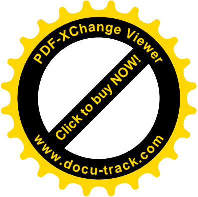
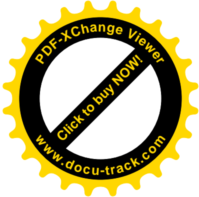
但是，XML-RPC也不是完全没有限制的，关键在于你是否把这些限制看作麻烦。例如，如果传入了一个自定义的Python对象，XML-RPC库会将该对象转换为Python字典，再将其串行化为XML，然后进行传递。这样，你需要写一些从字典的XML版本中提取数据的代码。实际上也可以直接使用RPC服务器上的对象。因此，是否使用XML-RPC，同样需要根据需要进行选择。

Pyro

Pyro是能够避免XML-RPC缺点的框架。Pyro代表Python Remote Objects（Python远程对象，首字母缩写）。Pyro能完成任何XML-RPC能够实现的功能，而不需将对象字典化。在传递数据时，它能够保持其原有类型。如果确定要使用Pyro，必需独立安装它。Python中不包含Pyro。值得注意的是，Pyro仅能够与Python一起使用，而XML-RPC既可以与Python一起使用，也可以与其他语言一起使用。例5-7实现了与XML-RPC示例相同的ls()功能。

例5-7：简单的Pyro服务器

```
#!/usr/bin/env python
```

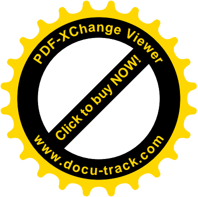
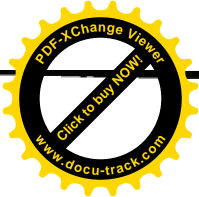
```
import Pyro.core
import os
from xmlrpc_pyro_diff import PSACB

class PSAExample(Pyro.core.ObjBase):

    def ls(self, directory):
        try:
            return os.listdir(directory)
        except OSError:
            return []

    def ls_boom(self, directory):
        return os.listdir(directory)

    def cb(self, obj):
        print "OBJECT:", obj
        print "OBJECT.__class__:", obj.__class__
        return obj.cb()

if __name__ == '__main__':
    Pyro.core.initServer()
    daemon=Pyro.core.Daemon()
    uri=daemon.connect(PSAExample(),"psaexample")

    print "The daemon runs on port:",daemon.port
    print "The object's uri is:",uri
    daemon.requestLoop()
```

Pyro示例与XML-RPC示例相似。首先，创建了PSAExample类，该类具有ls()、ls_boom()和cb()方法。然后通过Pyro的内部管道（internal plumbing）创建了daemon。接下来，将PSAExample与daemon关联。最后，通知daemon开始服务请求。

下面的示例演示如何在IPython提示符下访问Pyro服务器：

```
In [1]: import Pyro.core
/usr/lib/python2.5/site-packages/Pyro/core.py:11: DeprecationWarning:
The sre module is deprecated, please import re.
  import sys, time, sre, os, weakref

In [2]: psa = Pyro.core.getProxyForURI("PYROLOC://localhost:7766/psaexample")
Pyro Client Initialized. Using Pyro V3.5

In [3]: psa.ls(".")
Out[3]:
['pyro_server.py',
....
  'subprocess_arp.py',
  'web_server_checker_tcp.py']

In [4]: psa.ls_boom('.')
Out[4]:
['pyro_server.py',
....
  'subprocess_arp.py',
  'web_server_checker_tcp.py']
```

www.TopSage.com

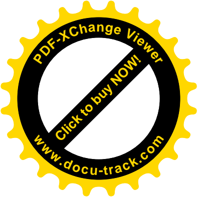
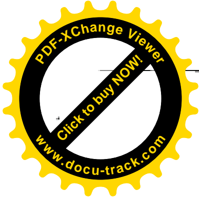
```
In [5]: psa.ls("/foo")
Out[5]: []

In [6]: psa.ls_boom("/foo")
---------------------------------------------------------------------------
<type 'exceptions.OSError'>                Traceback (most recent call last)

/home/jmjones/local/Projects/psabook/oreilly/<ipython console> in <module>()
.
.
...
<<big nasty traceback>>
...
.
.
--> 115                 raise self.excObj
    116             def __str__(self):
    117                 s=self.excObj.__class__.__name__

<type 'exceptions.OSError'>: [Errno 2] No such file or directory: '/foo'
```

非常不错！该示例返回了与XML-RPC示例相同的输出结果。这正是我们所期望的结果。但是，当传递一个自定义对象时，会是什么情况呢？下面，我们将要定义一个新的类，创建一个属于该类的对象，然后传递给XML-RPC的cb()函数以及Pyro示例中的cb()方法。例5-8显示了我们将要执行的代码段。

例5-8：XML-RPC 与Pyro的区别

```
import Pyro.core
import xmlrpclib

class PSACB:
    def __init__(self):
        self.some_attribute = 1

    def cb(self):
        return "PSA callback"

if __name__ == '__main__':
    cb = PSACB()

    print "PYRO SECTION"
    print "*" * 20
    psapyro = Pyro.core.getProxyForURI("PYROLOC://localhost:7766/psaexample")
    print "-->>", psapyro.cb(cb)
    print "*" * 20

    print "XML-RPC SECTION"
    print "*" * 20
    psaxmlrpc = xmlrpclib.ServerProxy('http://localhost:8765')
    print "-->>", psaxmlrpc.cb(cb)
    print "*" * 20
```

www.TopSage.com

调用Pyro和XML-RPC的cb()函数都需要在传递给它们的对象上调用cb()。在这两个示例中，都返回PSA回调字符串。下面是运行该代码时返回的内容：

```
jmjones@dinkgutsy:code$ python xmlrpc_pyro_diff.py
/usr/lib/python2.5/site-packages/Pyro/core.py:11: DeprecationWarning:
The sre module is deprecated, please import re.
  import sys, time, sre, os, weakref
PYRO SECTION
********************
Pyro Client Initialized. Using Pyro V3.5
-->> PSA callback
********************
XML-RPC SECTION
********************
-->>
Traceback (most recent call last):
  File "xmlrpc_pyro_diff.py", line 23, in <module>
    print "-->>", psaxmlrpc.cb(cb)
  File "/usr/lib/python2.5/xmlrpclib.py", line 1147, in __call__
    return self.__send(self.__name, args)
  File "/usr/lib/python2.5/xmlrpclib.py", line 1437, in __request
    verbose=self.__verbose
  File "/usr/lib/python2.5/xmlrpclib.py", line 1201, in request
    return self._parse_response(h.getfile(), sock)
  File "/usr/lib/python2.5/xmlrpclib.py", line 1340, in _parse_response
    return u.close()
  File "/usr/lib/python2.5/xmlrpclib.py", line 787, in close
    raise Fault(**self._stack[0])
xmlrpclib.Fault: <Fault 1: "<type 'exceptions.AttributeError'>:'dict' object
    has no attribute 'cb'">
```

Pyro的实现方法能够成功执行，但是XML-RPC的实现方法执行失败并返回一个追踪（traceback）。追踪返回的最后一行表明字典对象没有cb属性。当我们展示XML-RPC服务器的输出结果时，这会更容易理解。记住，cb()函数包含一些print语句，能够输出运行情况的信息。下面是XML-RPC服务器的输出结果：

```
OBJECT:: {'some_attribute': 1}
OBJECT.__class__:: <type 'dict'>
localhost - - [17/Apr/2008 16:39:02] "POST /RPC2 HTTP/1.0" 200 -
```

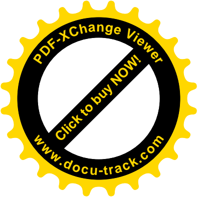
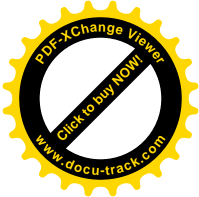
当我们字典化XML-RPC客户端创建的对象时，some_attribute被转换为一个字典关键字。当这个属性被保留时，cb()方法不保留。

下面是Pyro服务器的输出结果：

```
OBJECT: <xmlrpc_pyro_diff.PSACB instance at 0x9595a8>
OBJECT.__class__: xmlrpc_pyro_diff.PSACB
```

值得注意的是，对象所属类为PSACB。PSACB定义了对象如何被创建。在Pyro服务器端，

www.TopSage.com

我们不得不包括客户端使用的相同的代码。Pyro服务器需要导入客户端代码，这一点非常重要。Pyro使用Python标准的pickle来序列化对象，因此，它与Pyro非常相似。

总之，如果想要一个简单的RPC解决方案，而不希望有外部依赖，并且能够容忍XML-RPC的一些限制，就可以选择XML-RPC。如果再考虑到它能够与其他语言方便地实现互操作，那么或许就会认为XML-RPC是一个不错的选择了。另一方面，如果嫌XML-RPC限制太多，也不介意安装外部库，而且不介意仅使用Python一种语言，那么对你来说Pyro或许才是更好的选择。

SSH

SSH是一个极其强大并被广泛使用的协议。由于大多数协议的实现都会与协议有相同的名字，因此也可以将SSH视为一个工具。SSH允许你安全地连接到远程服务器，执行shell命令，传输文件，并在连接双方进行端口转发。

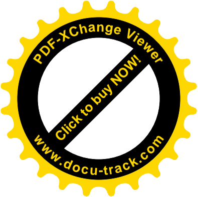
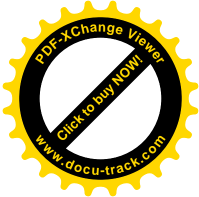
如果有一个命令行的SSH工具，为什么还要通过编写脚本来使用SSH协议呢？主要原因是这样做除了能够使用SSH协议的全部功能外，还能够使用Python的全部功能。

SSH2协议就是通过名为paramkio的Python库实现的。通过Python代码，可以连接到SSH服务器，并完成一些SSH任务。例5-9是一个连接到SSH服务器并执行简单命令的示例。

例5-9：连接到SSH服务器并远程执行命令

```
#!/usr/bin/env python

import paramiko

hostname = '192.168.1.15'
port = 22
username = 'jmjones'
password = 'xxxYYYxxx'

if __name__ == "__main__":
    paramiko.util.log_to_file('paramiko.log')
    s = paramiko.SSHClient()
    s.load_system_host_keys()
    s.connect(hostname, port, username, password)
    stdin, stdout, stderr = s.exec_command('ifconfig')
    print stdout.read()
    s.close()
```

从上面的代码可以看到，我们先加载了paramiko模块，定义了三个变量。接下来，创建了一个SSHClient对象。然后告诉它加载host keys。对于Linux系统，host keys来自文件known_host。之后就连接到SSH服务器上了。接下来就没有什么特别复杂的步骤了，尤其在熟悉了SSH之后。

现在，我们准备好远程执行命令了。对exec_command()的调用会执行传递给它的命令并返回三个与执行命令相关的文件句柄：标准输入、标准输出和标准错误。为了演示的需要，执行命令的机器IP地址与使用SSH调用连接的IP地址是相同的。下面是远程服务器上ifconfig命令的执行结果。

```
jmjones@dinkbuntu:~/code$ python paramiko_exec.py
eth0      Link encap:Ethernet HWaddr XX:XX:XX:XX:XX:XX
          inet addr:192.168.1.15 Bcast:192.168.1.255 Mask:255.255.255.0
          inet6 addr: xx00::000:x0xx:xx0x:0x00/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:9667336 errors:0 dropped:0 overruns:0 frame:0
          TX packets:11643909 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1427939179 (1.3 GiB) TX bytes:2940899219 (2.7 GiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:16436 Metric:1
          RX packets:123571 errors:0 dropped:0 overruns:0 frame:0
          TX packets:123571 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:94585734 (90.2 MiB) TX bytes:94585734 (90.2 MiB)
```

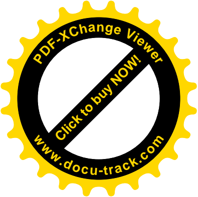
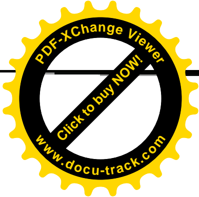
这看起来与在本地机器上执行ifconfig命令的结果非常相似，只是IP地址不同而已。

例5-10演示了如何使用paramiko的SFTP来实现远程主机与本地主机之前的文件传输。这个特殊示例演示的仅是使用get()方法从远程主机下载文件。如果想向远程主机上传文件，应使用put()方法。

例5-10：从SSH服务器上检索文件

```
#!/usr/bin/env python

import paramiko
import os

hostname = '192.168.1.15'
port = 22
username = 'jmjones'
password = 'xxxYYYxxx'
dir_path = '/home/jmjones/logs'

if __name__ == "__main__":
    t = paramiko.Transport((hostname, port))
    t.connect(username=username, password=password)
    sftp = paramiko.SFTPClient.from_transport(t)
    files = sftp.listdir(dir_path)
    for f in files:
        print 'Retrieving', f
        sftp.get(os.path.join(dir_path, f), f)
    t.close()
```

www.TopSage.com

如果需要使用公钥/私钥而不是密码该怎么做呢？例5-11是对上述远程执行命令程序的修改，其中使用了RSA加密。

例5-11：连接SSH服务器并远程执行命令-使用私钥

```
#!/usr/bin/env python

import paramiko

hostname = '192.168.1.15'
port = 22
username = 'jmjones'
pkey_file = '/home/jmjones/.ssh/id_rsa'

if __name__ == "__main__":
    key = paramiko.RSAKey.from_private_key_file(pkey_file)
    s = paramiko.SSHClient()
    s.load_system_host_keys()
    s.connect(hostname, port, pkey=key)
    stdin, stdout, stderr = s.exec_command('ifconfig')
    print stdout.read()
    s.close()
```

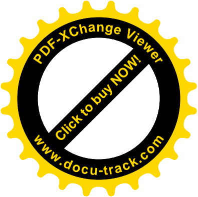
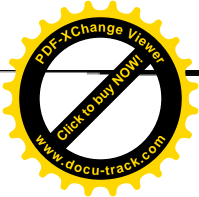
例5-12是一个sftp脚本的修改版本，其中也使用了RSA加密。

例5-12：从SSH服务器上检索文件

```
#!/usr/bin/env python

import paramiko
import os

hostname = '192.168.1.15'
port = 22
username = 'jmjones'
dir_path = '/home/jmjones/logs'
pkey_file = '/home/jmjones/.ssh/id_rsa'

if __name__ == "__main__":
    key = paramiko.RSAKey.from_private_key_file(pkey_file)
    t = paramiko.Transport((hostname, port))
    t.connect(username=username, pkey=key)
    sftp = paramiko.SFTPClient.from_transport(t)
    files = sftp.listdir(dir_path)
    for f in files:
        print 'Retrieving', f
        sftp.get(os.path.join(dir_path, f), f)
t.close()
```

Twisted

Twisted是一个事件驱动的Python网络框架，可以处理大量与网络相关的任务类型。一个全面独立的解决方案需要以复杂性为代价。在使用Twisted若干次之后，你会觉得它非常

好用，但是刚开始理解它确定存在困难。进一步说，学习Twisted是一个如此庞大的工程，以至于要找到一个能解决某一特定问题的切入点都有些困难。

尽管如此，我们还是强烈建议你去熟悉它，并且考虑一下它是否适合你。如果可以很容易地将固有思维转换到Twisted方式，那么学习Twisted会是一笔十分有价值的投资。由Abe Fettig (O'Reilly)编著的《Twisted Network Programming Essentials》是一本入门级的非常不错的书，可以帮助我们解决学习过程中的许多问题。

Twisted是一个事件驱动的网络，这意味着代码的编写不是集中在实现初始连接、断开连接以及数据处理等较低级的细节问题上，而是集中在编写任务的处理程序。

通过Twisted可以获得什么优势呢？Twisted框架能够帮助你将要处理的问题分成若干个小问题。而Twisted的网络连接能够帮助你无须考虑连接时需要的功能。这两个优势都能够使代码更易于复用。此外，使用Twisted，不需要考虑对一些低层的连接和错误操作进行处理。你的工作将集中在编写实现对各类事件处理的代码。

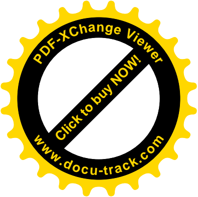
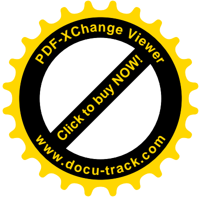
例5-13是用Twisted实现的一个端口检测程序。这是一个很基本的事例，但是在我们学习该代码后，会理解Twisted事件驱动的本质。在开始看示例之前，应该了解一些基本概念。最基本的概念包括反应器（reactor），工厂（factory），协议和延迟（deferred）。reactor是Twisted应用的主事件循环。reactor操纵事件分发，网络通信和线程。factory负责创建新的协议实例。每一个factory实例可以产生一种类型的协议。协议定义了对指定连接如何进行操作。运行时，每一个连接都会创建一个协议实例。deferred是一种链接协同行为方式。

> **Twisted**
>
> 大多数写代码的人对一个程序或脚本的逻辑流有非常强烈的直觉：就像水在山间流淌，自上而下。这样的代码非常容易理解，编写和调试也十分容易。Twisted代码则完全不同。作为一种异步方式，或许有人会说它更像是在低重力环境下的水滴，而不是一条沿着山坡流淌的河流。Twisted引入了新的组件：事件反应器（reactor）和friend。使用Twisted创建和调试代码，必须放弃之前的程序逻辑，建立起不同的逻辑流。

例5-13：Twisted实现端口检测

```
#!/usr/bin/env python

from twisted.internet import reactor, protocol
import sys

class PortCheckerProtocol(protocol.Protocol):
    def __init__(self):
```


www.TopSage.com

```
        print "Created a new protocol"
    def connectionMade(self):
        print "Connection made"
        reactor.stop()

class PortCheckerClientFactory(protocol.ClientFactory):
    protocol = PortCheckerProtocol
    def clientConnectionFailed(self, connector, reason):
        print "Connection failed because", reason
        reactor.stop()

if __name__ == '__main__':
    host, port = sys.argv[1].split(':')
    factory = PortCheckerClientFactory()
    print "Testing %s" % sys.argv[1]
    reactor.connectTCP(host, int(port), factory)
    reactor.run()
```

值得注意的是我们定义了Twisted类的两个子类PortCheckerProtocol和PortChecker-ClientFactory。我们通过将PortCheckerProtocol指定到PortCheckerClientFactory的protocol类属性，将PortCheckerClientFactory绑定到PortCheckerProtocol。如果factory试图创建一个连接，但是失败了，factory的clientConnectionFailed()方法将被调用。ClientConnectionFailed()是一个对所有Twisted 工厂来说都十分常见的方法，也是我们为factory定义的唯一方法。通过定义与factory类相关的方法，可以重载类的默认行为。在一个客户端连接失败之后，我们希望输出相关信息并停止reactor。

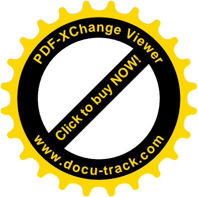
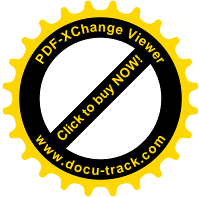
PortcheckerProtocol是之前讨论过的一个协议。一旦建立了一个到服务器的连接，且该服务器端口正是我们所检测的，该类的一个实例就会被创建。这里仅定义了PortCheckerProtocol类的一个方法：connectionMadw()。这是一个对所有的Twisted协议类都通用的方法。通过自己定义该方法，我们重载了默认行为。一旦成功建立了一个连接，Twisted会调用该协议的connectionMade()方法。可以看到，它输出了一个简单的信息并且停止了reactor（一会儿我们会介绍reactor）。

在这个示例中，connectionMade()和clientConnectionFailed()展示了Twisted的事件驱动本质。创建连接即为一个事件。同样，创建连接失败也是一个事件。当这些事件发生时，Twisted调用适合的方法来处理事件，这被称为事件处理函数。

在这个示例的主要部分，我们先创建了一个PortCheckerClientFactory实例，然后告诉Twisted 的reactor对指定的主机和端口进行连接。主机名和端口号是使用指定的工厂从命令行获得的。在告诉reactor连接某一服务器的某一端口之后，再告诉reactor运行起来。如果没有告诉reactor运行，什么也不会发生。

总结流程顺序，我们在给出一个指令之后启动reactor。在这个示例中，指令就是连接到一台服务器的指定端口，然后使用PortCheckerClientFactory来帮助分发事

www.TopSage.com

件。如果连接到指定主机和端口失败，事件循环会调用PortCheckerClientFactory的clientConnectionFailed()方法。如果连接成功，工厂会创建一个协议实例，PortcheckerProtocol，然后在该实例上调用connectionMade()方法。不管连接成功或失败，相应事件的处理函数都会关闭reactor，并且程序会停止执行。

这是一个非常基本的示例，但是它显示了Twisted事件处理的本质。有一个Twisted编程的关键概念在本例中没有涉及，那就是deferred和callback。一个deferred表示执行请求动作的承诺。callback指定了一个定义成功执行动作的方法。deferred可以连续使用，并可以将从一次使用得来的结果传递到下一次。在Twisted中这一点往往比较难理解。（例5-14将详细介绍deferred）

例5-14是一个使用Perspective Broker的示例，Perspective Broker是Twisted独有的RPC机制。这个示例是对远程ls服务器的实现，在这一章的前面部分已经在XML-RPC和Pyro中实现过了。首先，我们看服务器端实现。

例5-14：Twisted 实现Perspective Broker服务器

```
import os
from twisted.spread import pb
from twisted.internet import reactor

class PBDirLister(pb.Root):
    def remote_ls(self, directory):
        try:
            return os.listdir(directory)
        except OSError:
            return []

    def remote_ls_boom(self, directory):
        return os.listdir(directory)

if __name__ == '__main__':
    reactor.listenTCP(9876, pb.PBServerFactory(PBDirLister()))
    reactor.run()
```

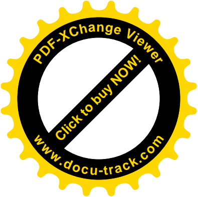
这个示例定义了一个PBDirLister类。这是一个当客户端接到服务器时，作为远端对象的Perspective Broker (PB)类。这个示例仅在这个类中定义了两个方法：remote_ls()和remote_ls_boom()。remote_ls()方法会简单地返回一个指定目录的列表。remote_ls_boom()能够实现与remote_ls()相同功能，但不会理会异常操作。在示例的主要部分，我们使用Perspective Broker绑定端口9876并返回reactor。

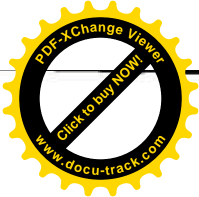
例5-15 不是顺序的，它调用了remote_ls()。

例5-15：Twisted实现 Perspective Broker客户端

```
#!/usr/bin/python
```


www.TopSage.com






```
from twisted.spread import pb
from twisted.internet import reactor

def handle_err(reason):
    print "an error occurred", reason
    reactor.stop()

def call_ls(def_call_obj):
    return def_call_obj.callRemote('ls', '/home/jmjones/logs')

def print_ls(print_result):
    print print_result
    reactor.stop()

if __name__ == '__main__':
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 9876, factory)
    d = factory.getRootObject()
    d.addCallback(call_ls)
    d.addCallback(print_ls)
    d.addErrback(handle_err)
    reactor.run()
```

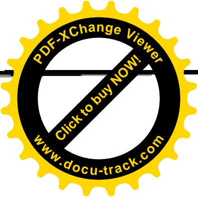
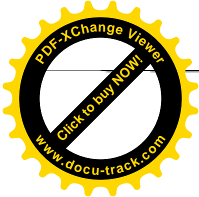
这个客户端示例定义了三个函数，handle_err()，call_ls()和print_ls()。handle_err()将处理这一过程中出现的任何错误。call_ls()会初始化调用远端的ls方法。print_ls()会输出ls调用的结果。也就是说代码中有一个初始化远端调用的函数和一个打印调用结果的函数，这似乎有点怪异。正是因为Twisted是一个异步的事件驱动网络框架，所以才会有这样的情况产生。这一框架鼓励我们在写代码的时候，将工作分解为许多小的部分。

这个示例的主要部分展示了reactor如何知道什么时候调用callback函数。首先，创建一个客户端Perspective Broker工厂，然后告诉reactor连接到localhost:9876，并使用PB客户端工厂来处理请求。接下来，通过调用factory.getRootObject()，我们获得了对远端对象的控制。这实际上是一个deferred，因此，可以通过调用addCallback()来进行管道连接。

我们添加的第一个回调函数是call_ls()。call_ls()在前一步创建的deferred对象上调用callRemote()方法。callRemote()也会返回一个deferred。在过程链中第二个回调函数是print_ls()。当reactor调用了print_ls()，print_ls()将远端调用的结果打印到前一步中创建的remote_ls()中。事实上，是reactor将远端调用的结果传递到print_ls()的。第三个在过程链中用到的回调函数是handle_err()，这是一个简单的错误处理函数，它让我们知道在这一过程中是否出现了错误。一旦有错误发生或是管道达到了print_ls()，respective方法将关闭reactor。

下面是客户端代码运行示例：

www.TopSage.com

```
jmjones@dinkgutsy:code$ python twisted_perspective_broker_client.py
['test.log']
```

输出结果为指定目录的所有文件的列表，这与我们期望的结果一致。

相对于这里列出的简单RPC示例，这一示例看起来有些复杂。服务器端看起来相当漂亮。创建客户端的工作看起来就是一些将callbacks，deferreds，reactors和工厂通过管道进行连接处理。但是这只是一个非常简单的示例。Twisted结构的真正闪光之处只有在处理非常复杂的任务时才能充分体现。

例5-16是对之前展示的Perspective Broker客户端代码进行略微修改的版本。没有在远端服务器上调用ls，而是调用了ls_boom。下面会展示客户端与服务器如何处理异常。

例5-16：Twisted 实现Perspective Broker 客户端－异常处理

```
#!/usr/bin/python

from twisted.spread import pb
from twisted.internet import reactor

def handle_err(reason):
    print "an error occurred", reason
    reactor.stop()

def call_ls(def_call_obj):
    return def_call_obj.callRemote('ls_boom', '/foo')

def print_ls(print_result):
    print print_result
    reactor.stop()

if __name__ == '__main__':
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 9876, factory)
    d = factory.getRootObject()
    d.addCallback(call_ls)
    d.addCallback(print_ls)
    d.addErrback(handle_err)
    reactor.run()
```

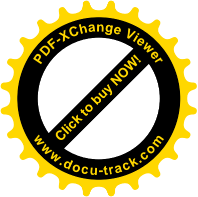
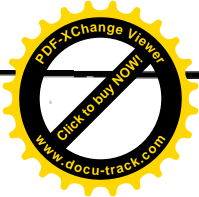
下面是代码运行后的结果：

```
jmjones@dinkgutsy:code$ python twisted_perspective_broker_client_boom.py an
error occurred [Failure instance: Traceback from remote host -- Traceback
unavailable
]
```

在服务器端：

```
Peer will receive following PB traceback:
Traceback (most recent call last):
...
```

www.TopSage.com

```
<more traceback>
...
    state = method(*args, **kw)
  File "twisted_perspective_broker_server.py", line 13, in remote_ls_boom
    return os.listdir(directory)
exceptions.OSError: [Errno 2] No such file or directory: '/foo'
```

错误位于服务器端代码中，而不是客户端代码中。在客户端，我们仅知道一个错误发生了。如果Pyro或是XML-RPC这样运行将是一件糟糕的事情。然而，在Twisted客户端代码中，错误处理函数被调用了。由于这是一个与Pyro和XML-RPC（基于事件）不同的模型，我们期望能够有差别地处理错误，而Perspective Broker代码正好实现了这些功能。

这里对Twisted的介绍仅是冰山一角。在刚开始使用Twisted时会觉得有些困难，因为它是非常复杂的项目和任务，是与我们通常习惯的方法不同的方法。Twisted非常值得进一步花些时间学习，也值得你把它放入工具包。

Scapy

如果喜欢网络编程，你会喜欢上Scapy的。Scapy是一个非常便捷的交互式包操作程序和库。Scapy可以发现网络、执行扫描、跟踪路由并进行探测。Scapy同样有非常好的文档和资料可供使用。如果希望深入学习，可以购买一本有关Scapy的书来进一步了解Scapy的细节。

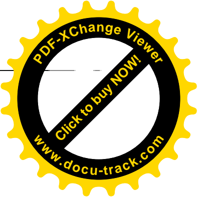
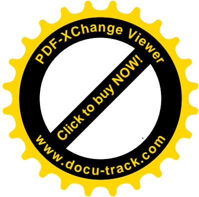
在写这本书时，Scapy还只是一个文件。可以在*http://hg.secdev.org/scapy/raw-file/tip/scapy.py*下载最新的版本。一旦下载了Scapy，就可以将其作为一个工具独立运行或是作为库文件加载使用。这里从交互式工具的角度来开始介绍Scapy。请记住，在运行Scapy时需要具有root权限，因为这样才能够对网络接口进行控制。

一旦下载并安装了Scapy，会看到这样的信息：

```
Welcome to Scapy (1.2.0.2)
>>>
```

你可以实现任何通常在Python解释器中所做的操作，同时，Scapy也有其特有的命令。我们要做的第一件事就是运行Scapy的`ls()`函数，它列出了所有可用的层次：

```
>>> ls()
ARP        : ARP
ASN1_Packet : None
BOOTP      : BOOTP
CookedLinux : cooked linux
DHCP       : DHCP options
DNS        : DNS
DNSQR      : DNS Question Record
DNSRR      : DNS Resource Record
```

```
Dot11       : 802.11
Dot11ATIM   : 802.11 ATIM
Dot11AssoReq : 802.11 Association Request
Dot11AssoResp : 802.11 Association Response
Dot11Auth   : 802.11 Authentication
[snip]
```

由于输出比较长，这里只截取了一部分。现在，执行一个递归DNS查询，使用加利福尼亚理工学院的公共DNS服务器对*www.oreilly.com*进行查询：

```
>>> sr1(IP(dst="131.215.9.49")/UDP()/DNS(rd=1,qd=DNSQR(qname="www.oreilly.com")))
Begin emission:
Finished to send 1 packets.
...*
Received 4 packets, got 1 answers, remaining 0 packets
IP  version=4L ihl=5L tos=0x0 len=223 id=59364 flags=DF
  frag=0L ttl=239 proto=udp chksum=0xb1e src=131.215.9.49 dst=10.0.1.3 options=''
|UDP sport=domain dport=domain len=203 chksum=0x843 |
DNS id=0 qr=1L opcode=QUERY aa=0L tc=0L rd=1L ra=1L z=0L
  rcode=ok qdcount=1 ancount=2 nscount=4 arcount=3 qd=
DNSQR qname='www.oreilly.com.' qtype=A qclass=IN |>
  an=DNSRR rrname='www.oreilly.com.' type=A rclass=IN ttl=21600 rdata='208.201.239.36'
[snip]
```

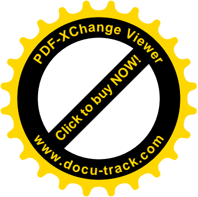
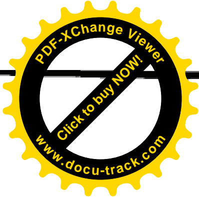
接下来，执行一个traceroute：

```
>>> ans,unans=sr(IP(dst="oreilly.com",
>>> ttl=(4,25),id=RandShort())/TCP(flags=0x2))
Begin emission:
..............*Finished to send 22 packets.
*...........*********.***.***.*.*.*.*.*
Received 54 packets, got 22 answers, remaining 0 packets
>>> for snd, rcv in ans:
...     print snd.ttl, rcv.src, isinstance(rcv.payload, TCP)
...
[snip]
20 208.201.239.37 True
21 208.201.239.37 True
22 208.201.239.37 True
23 208.201.239.37 True
24 208.201.239.37 True
25 208.201.239.37 True
```

Scapy甚至可以实现纯粹的包复制，类似于tcpdump：

```
>>> sniff(iface="en0", prn=lambda x: x.show())
###[ Ethernet ]###
dst= ff:ff:ff:ff:ff:ff
src= 00:16:cb:07:e4:58
type= IPv4
###[ IP ]###
version= 4L
ihl= 5L
```


www.TopSage.com

```
        tos= 0x0
        len= 78
        id= 27957
        flags=
        frag= 0L
        ttl= 64
        proto= udp
        chksum= 0xf668
        src= 10.0.1.3
        dst= 10.0.1.255
        options= ''
      [snip]
```

如果安装了graphviz和imagemagic，还可以将网络路由的追踪过程可视化。下面的示例来自Scapy官方文档：

```
>>> res,unans = traceroute(["www.microsoft.com","www.cisco.com","www.yahoo.com",
  "www.wanadoo.fr","www.pacsec.com"],dport=[80,443],maxttl=20,retry=-2)
Begin emission:
*************************************************************************
 Finished to send 200 packets.
******************Begin emission:
*******************************************Finished to send 110 packets.
**************************************************************Begin emission:
Finished to send 5 packets.
Begin emission:
Finished to send 5 packets.

Received 195 packets, got 195 answers, remaining 5 packets
193.252.122.103:tcp443 193.252.122.103:tcp80 198.133.219.25:tcp443 198.133.219.25:tcp80
  207.46.193.254:tcp443 207.46.193.254:tcp80 69.147.114.210:tcp443 69.147.114.210:tcp80
  72.9.236.58:tcp443 72.9.236.58:tcp80
```

现在，可以利用这些结果创建一幅有趣的图画：

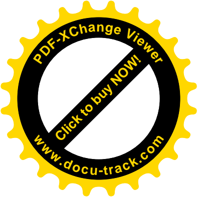
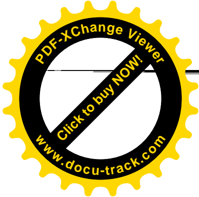
```
>>> res.graph()
>>> res.graph(type="ps",target="| lp")
>>> res.graph(target="> /tmp/graph.svg")
```

如果已经安装了graphviz和imagemagic，网络可视化将会给你留下深刻印象。

在使用Scapy的过程中，真正有趣的是创建自定义的命令行工具和脚本。在接下来的一节中，我们将进一步介绍Scapy的库。

使用Scapy创建脚本

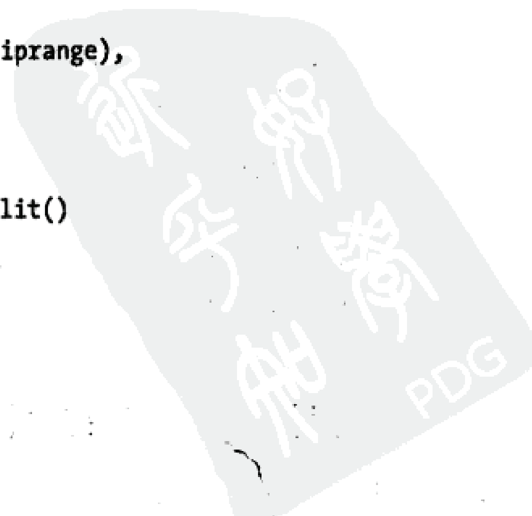
现在试着使用Scapy，模仿bat创建一个arping工具。首先看一下特定平台（platform-specific）的arping工具。

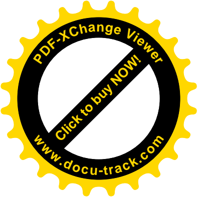
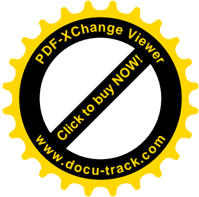
```
#!/usr/bin/env python
import subprocess
import re
import sys

def arping(ipaddress="10.0.1.1"):
    """Arping function takes IP Address or Network, returns nested mac/ip list"""

    #Assuming use of arping on Red Hat Linux
    p = subprocess.Popen("/usr/sbin/arping -c 2 %s" % ipaddress, shell=True,
                        stdout=subprocess.PIPE)
    out = p.stdout.read()
    result = out.split()
    #pattern = re.compile(":")
    for item in result:
        if ':' in item:
            print item

if __name__ == '__main__':
    if len(sys.argv) > 1:
        for ip in sys.argv[1:]:
            print "arping", ip
            arping(ip)
    else:
        arping()
```

现在，让我们看看如何以平台中立（platform-neutral）的方式使用Scapy完成同样的工作。

```
#!/usr/bin/env python
from scapy import srp,Ether,ARP,conf
import sys

def arping(iprange="10.0.1.0/24"):
    """Arping function takes IP Address or Network, returns nested mac/ip list"""

    conf.verb=0
    ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=iprange),
              timeout=2)

    collection = []
    for snd, rcv in ans:
        result = rcv.sprintf(r"%ARP.psrc% %Ether.src%").split()
        collection.append(result)
    return collection

if __name__ == '__main__':
    if len(sys.argv) > 1:
        for ip in sys.argv[1:]:
            print "arping", ip
            print arping(ip)
    else:
        print arping()
```



www.TopSage.com

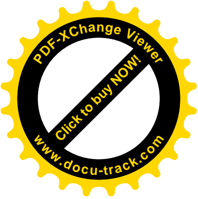
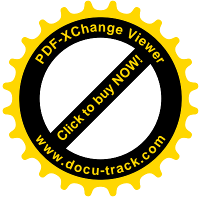



可以看到，输出中所包含的信息相当清晰，是子网中所有的Mac地址和IP地址：

```
# sudo python scapy_arp.py
[['10.0.1.1', '00:00:00:00:00:10'], ['10.0.1.7', '00:00:00:00:00:12'],
['10.0.1.30', '00:00:00:00:00:11'], ['10.0.1.200', '00:00:00:00:00:13']]
```

从这些示例中，你会对Scapy的简单和易用留下深刻印象。

www.TopSage.com

第6章

数据

引言

IT组织需要系统管理员的原因之一，是需要有专门人员能够对数据、文件及目录的处理进行控制。什么样的系统管理员不需要对目录树中的文件进行处理，不需要对文本进行解析和替换呢？如果不曾编写过一个对目录树中的所有文件进行重新命名的脚本，你或许在将来的某些时候还是需要这样做。这些能力是一名系统管理员应该具备的能力，或者至少是一名真正好的系统管理员必须具备的能力。本章接下来的内容将集中在数据、文件和目录上。

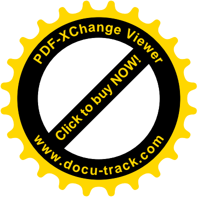
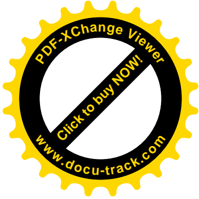
系统管理员需要不断地将数据从一个地方迁移到另一个地方。数据的迁移对于系统管理员来说是极为普通的工作。在动画业中，需要不断地将数据从一个位置迁移到另一个位置，因为数字电影产品需要数以千兆的存储空间。而且基于某一时刻观看图像时所需的不同分辨率和画面质量，不同数据有着不同的磁盘I/O。如果数据需要被迁移到HD预览房间进行数字化检测，则新被解压的或略微压缩的HD图像文件需要被移走。文件之所以常常需要被移动，是因为动画的存储设备通常有两种。既有廉价的、大的、慢的、安全的存储设备，也有快速的、昂贵的存储设备，例如JBOD或者是高速磁盘阵列RAID0。在电影工业中，主要进行数据处理的系统管理员被称为“数据牧马人”。

数据牧马人需要不断地将数据从一个地方移动和整合到另一个地方。通常，使用最为频繁的命令是rsync，scp，cp和mv。将这些简单而功能强大的工具用在Python脚本中，能够完成一些不可思议的工作。

使用标准库，可以在shell中完成许多不可思议的工作。使用标准库的好处是可以在任何地方执行数据移动脚本，而不需要依赖特定的平台。

当然，我们也不应忘记备份。只需要编写少量Python代码，就可以实现大多数自定义的备份脚本和应用程序。需要注意的是，为备份代码编写额外的测试程序不仅明智，而

www.TopSage.com

且必要。应该确保在需要执行自己编写的备份脚本时，已经对该脚本进行了单元及功能测试。

此外，我们经常需要在一次移动之前，之后或移动过程中对数据进行处理。当然，Python同样完全可以胜任这一工作。创建一个删除重复工具来查找并删除重复文件，是非常有意义的。接下来我们会展示如何实现这一功能。此外，下面还将例举一个有关系统管理员经常会遇到的数据流处理的示例。

使用 OS 模块与Data进行交互

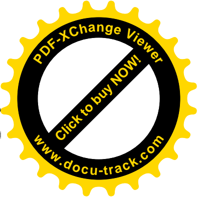
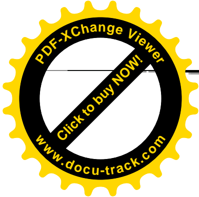
如果曾为编写跨平台的shell脚本而费尽心机，你会非常感谢OS模块，它是一个便携的系统服务应用程序接口（API）。在Python2.5中，OS模块包含超过200个方法，并且这些方法中的大部分都能够进行数据处理。本节会介绍该模块中的一些方法。在处理数据时，系统管理员应该更多地关注这个模块。

无论何时，如果发现自己需要去了解一个模块，IPython通常是完成这一工作的恰当工具。因此这里从使用IPython来执行一系列非常普通的操作起步，开始我们的OS模块学习之旅。

例6-1：浏览普通OS模块数据的方法

```
In [1]: import os

In [2]: os.getcwd()
Out[2]: '/private/tmp'

In [3]: os.mkdir("/tmp/os_mod_explore")

In [4]: os.listdir("/tmp/os_mod_explore")
Out[4]: []

In [5]: os.mkdir("/tmp/os_mod_explore/test_dir1")

In [6]: os.listdir("/tmp/os_mod_explore")
Out[6]: ['test_dir1']

In [7]: os.stat("/tmp/os_mod_explore")
Out[7]: (16877, 6029306L, 234881026L, 3, 501, 0, 102L,
1207014425, 1207014398, 1207014398)

In [8]: os.rename("/tmp/os_mod_explore/test_dir1",
"/tmp/os_mod_explore/test_dir1_renamed")

In [9]: os.listdir("/tmp/os_mod_explore")
Out[9]: ['test_dir1_renamed']

In [10]: os.rmdir("/tmp/os_mod_explore/test_dir1_renamed")

In [11]: os.rmdir("/tmp/os_mod_explore/")
```

www.TopSage.com

可以看到，加载了OS模块之后，在line[2]获得了当前的工作目录。之后，在line[3]创建了一个目录。在line[4]使用了os.listdir列出新创建目录的内容。接下来，执行了os.stat()，这与Bash中的stat命令非常相似()，之后在line[8]对目录重新命名。在line[9]，对目录是否创建进行了验证，最后使用or.rmdir方法删除了创建的目录。

无论如何，我们都需要详细地看一下OS模块。OS模块中有许多方法是处理数据时可能用到的，包括更改权限，创建符号连接等。请参考当前使用Python版本的文档，或者使用IPython的tab自动完成功能来查看OS模块的可用方法。

拷贝、移动、重命名和删除数据

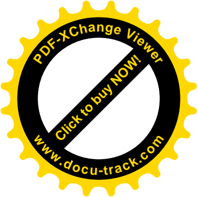
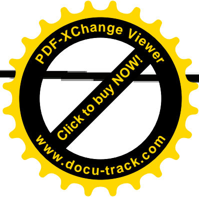
既然在引言中已经提到了数据迁移，你现在也对如何使用OS模块有了一定认识，那么，我们再来学习一个高层模块shutil，该模块能够处理大规模数据。与OS模块十分相似，shutil模块具有复制、移动、重命名和删除数据的方法。但是它可以在整个数据树上执行操作。

在IPython中尝试使用shutil模块是一个熟悉该工具的有效途径。在下面的示例中，我们将用到shutil.copytree，而shutil还有许多其他的方法，能够完成一些不同的功能。请参考Python标准库文档，查看不同的shutil copy方法之间的差别。

例6-2：使用shutil模块复制数据树

```
In [1]: import os

In [2]: os.chdir("/tmp")
In [3]: os.makedirs("test/test_subdir1/test_subdir2")

In [4]: ls -lR
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test/

./test:
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/

./test/test_subdir1:
total 0
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/

./test/test_subdir1/test_subdir2:
In [5]: import shutil

In [6]: shutil.copytree("test", "test-copy")

In [19]: ls -lR
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test/
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test-copy/
```

www.TopSage.com

```
./test:
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/

./test/test_subdir1:
total 0
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/

./test/test_subdir1/test_subdir2:

./test-copy:
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/

./test-copy/test_subdir1:
total 0
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/

./test-copy/test_subdir1/test_subdir2:
```

很明显，本示例非常简单，但极为有用。可以非常容易地将这类代码封装成更复杂且跨平台的数据移动脚本。我们能够想到的这类代码的直接用途，是在事件驱动下将数据从一个文件系统转移动另一个文件系统。在动画环境中，经常需要等最后一帧完成之后，才能将全部图像转换成序列进行编辑。

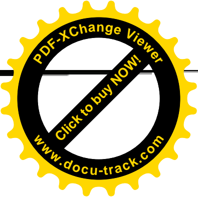
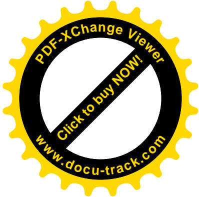
我们可以在cron任务中编写脚本查看目录中的帧数是整否达到了“X”个。当cron任务看到已经达到指定的帧数目时，可以将该目录移动到另一个帧可以在其中被处理的目录，或者直接将目录移动到具有快速I/O的磁盘上，便于回放未经压缩的HD胶片。

shutil模块不仅能够复制文件，它还有一些移动和删除数据树的方法。例6-3显示了数据树的移动，例6-4显示了数据树的删除。

例6-3：使用shutil移动数据树

```
In [20]: shutil.move("test-copy", "test-copy-moved")

In [21]: ls -lR
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test/
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test-copy-moved/

./test:
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/

./test/test_subdir1:
total 0
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/

./test/test_subdir1/test_subdir2:

./test-copy-moved:
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/
```

www.TopSage.com

```
./test-copy-moved/test_subdir1:
total 0
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/

./test-copy-moved/test_subdir1/test_subdir2:
```

例6-4：使用shutil删除数据树

```
In [22]: shutil.rmtree("test-copy-moved")

In [23]: shutil.rmtree("test-copy")
In [24]: ll
```

移动数据树要比删除数据树更令人激动，因为在删除之后不会有任何显示。多个这类简单的示例可以与其他的操作合并到一个更复杂的脚本中。如果要写一个备份工具来向网络存储空间复制一个目录树，并创建一个时间戳记录，这类脚本会非常有用。幸运的是，本章中正好有一个在Python中完成这类工作的示例。

使用路径、目录和文件

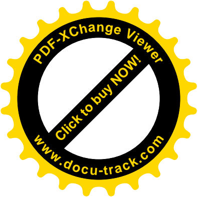
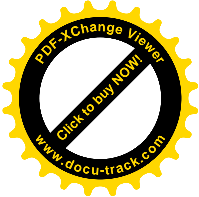
不考虑路径、目录和文件是无法处理数据的。每一名系统管理员都需要至少会写一个工具，该工具可以浏览目录，完成条件搜索，做一些有返回结果的操作。下面将要讨论一些能够实现这些操作的有趣方法。

通常，Python的标准库有一些杀死工具，可以停止作业的执行。Python最出名的就是它的“连电池都包括在内”的特性。例6-5显示了如何创建一个额外的冗余（verbose）目录遍历脚本，具有返回明确的文件、目录和路径的功能。

例6-5：冗余目录遍历脚本

```
import os
path = "/tmp"

def enumeratepaths(path=path):
"""Returns the path to all the files in a directory recursively"""
path_collection = []
for dirpath, dirnames, filenames in os.walk(path):
for file in filenames:
    fullpath = os.path.join(dirpath, file)
    path_collection.append(fullpath)

return path_collection

def enumeratefiles(path=path):
"""Returns all the files in a directory as a list"""
file_collection = []
for dirpath, dirnames, filenames in os.walk(path):
    for file in filenames:
        file_collection.append(file)

return file_collection
```

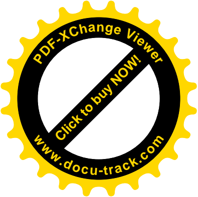
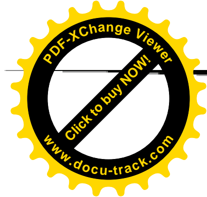
www.TopSage.com

```
def enumeratedir(path=path):
"""Returns all the directories in a directory as a list"""
dir_collection = []
for dirpath, dirnames, filenames in os.walk(path):
    for dir in dirnames:
        dir_collection.append(dir)

return dir_collection

if __name__ == "__main__":
print "\nRecursive listing of all paths in a dir:"
for path in enumeratepaths():
    print path
print "\nRecursive listing of all files in dir:"
for file in enumeratefiles():
    print file
print "\nRecursive listing of all dirs in dir:"
for dir in enumeratedir():
    print dir
```

在一台Mac笔记本中，该脚本的输出结果如下所示：

```
[ngift@Macintosh-7][H:12022][J:0]# python enumarate_file_dir_path.py

Recursive listing of all paths in a dir:
/tmp/.aksusb
/tmp/ARD_ABJMMRT
/tmp/com.hp.launchport
/tmp/error.txt
/tmp/liten.py
/tmp/LitenDeplicationReport.csv
/tmp/ngift.liten.log
/tmp/hsperfdata_ngift/58920
/tmp/launch-h36okI/Render
/tmp/launch-qy1S9C/Listeners
/tmp/launch-RTJzTw/:0
/tmp/launchd-150.wDvODl/sock

Recursive listing of all files in dir:
.aksusb
ARD_ABJMMRT
com.hp.launchport
error.txt
liten.py
LitenDeplicationReport.csv
ngift.liten.log
58920
Render
Listeners
:0
sock

Recursive listing of all dirs in dir:
.X11-unix
hsperfdata_ngift
launch-h36okI
```

```
launch-qy1S9C
launch-RTJzTw
launchd-150.wDvODl
ssh-YcE2t6PfnO
```

需要注意一点，os.walk返回的是一个generator对象，因此，如果在调用时向os.walk传递了值，你可以自行遍历目录树。

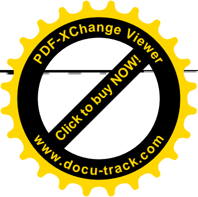
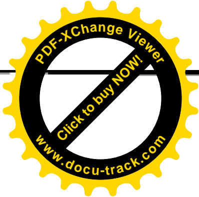
```
In [2]: import os

In [3]: os.walk("/tmp")
Out[3]: [generator object at 0x508e18]
```

这是在IPython中运行示例的情形。注意，使用generator使我们能够调用path.next()。这里不会介绍generator的细节，但需要知道os.walk返回的是一个generator对象。generator对于系统编程非常有用。请访问David Beazely的网站（*http://www.dabeaz.com/generators/*），在那里你可以找到需要了解的相关内容。

```
In [2]: import os

In [3]: os.walk("/tmp")
Out[3]: [generator object at 0x508e18]

    In [4]: path = os.walk("/tmp")

    In [5]: path.
    path.__class__          path.__init__          path.__repr__          path.gi_running
    path.__delattr__        path.__iter__          path.__setattr__       path.next
    path.__doc__            path.__new__           path.__str__           path.send
    path.__getattribute__   path.__reduce__        path.close             path.throw
    path.__hash__           path.__reduce_ex__     path.gi_frame

    In [5]: path.next()
    Out[5]:
    ('/tmp',
    ['.X11-unix',
    'hsperfdata_ngift',
    'launch-h36okI',
    'launch-qy1S9C',
    'launch-RTJzTw',
    'launchd-150.wDvODl',
    'ssh-YcE2t6PfnO'],
    ['.aksusb',
    'ARD_ABJMMRT',
    'com.hp.launchport',
    'error.txt',
    'liten.py',
    'LitenDeplicationReport.csv',
    'ngift.liten.log'])
```

再来多看一点有关generator的细节。让我们首先创建一个cleaner模块，该模块将通过API返回文件、目录及路径。

www.TopSage.com

现在，我们遍历了一个非常基本的目录，接下来要考虑的是如何将其变成一个面向对象的模块，以便于加载和复用。也许需要花些工夫才能创建一个固定编码的模块，但是一旦完成创建，就可以将其作为可复用的通用模块，使用起来非常简单方便。参见例6-6。

例6-6：创建可复用的目录浏览模块

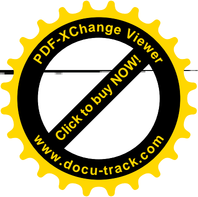
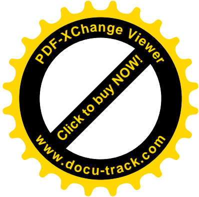
```
import os

class diskwalk(object):
"""API for getting directory walking collections"""
def __init__(self, path):
    self.path = path

def enumeratePaths(self):
    """Returns the path to all the files in a directory as a list"""
    path_collection = []
    for dirpath, dirnames, filenames in os.walk(self.path):
        for file in filenames:
            fullpath = os.path.join(dirpath, file)
            path_collection.append(fullpath)

    return path_collection

def enumerateFiles(self):
    """Returns all the files in a directory as a list"""
    file_collection = []
    for dirpath, dirnames, filenames in os.walk(self.path):
        for file in filenames:
            file_collection.append(file)

    return file_collection

def enumerateDir(self):
    """Returns all the directories in a directory as a list"""
    dir_collection = []
    for dirpath, dirnames, filenames in os.walk(self.path):
        for dir in dirnames:
            dir_collection.append(dir)

return dir_collection
```

可以看到，进行少量修改之后，能够创建一个非常好的接口，以方便将来进一步修改。这个新模块的优点之一是可以将其载入到其他脚本中。

数据比较

数据比较对系统管理员非常重要。你或许会经常问自己，“这两个目录中的文件到底有什么差别？系统中有多少个重复的文件存在？”在这一节中，你会找到这些问题或是类似问题的答案。

www.TopSage.com

在处理大批量的重要数据时，对目录树和文件进行比较以找到其中的差异是非常重要的。如果正要编写大块数据的迁移脚本，这会变得十分关键。一种非常糟糕的情况就是编写了一个会损坏关键生产数据的大数据块迁移脚本。

本节首先介绍一些轻量级的文件和目录进行比较的方法，最后介绍如何对文件进行校验比较。Python标准库有许多模块支持比较操作，这里将介绍filecmp和OS.Listdir。

使用filecmp模块

filecomp模块包括执行快速有效的文件和目录比较函数。filecmp模块会在两个文件上执行os.stat，如果os.stat对两个文件的判断结果相同，则返回True，如果结果不同，则返回False。典型地，os.stat判断两个文件是否使用同一磁盘上相同的i节点，或者它们的大小是否相等，但实际上并没有比较文件的内容。

要全面理解filecmp如何工作，需要创建三个新文件。先将目录切换到*/tmp*目录下，创建一个名为*file0.txt*的文件，并在文件中写入数字“0”。接下来，创建名为*file1.txt*的文件，并在该文件中写入数字“1”。最后，创建名为*file00.txt*的文件，并写入数字“0”。我们将在下面的示例代码中使用这几个文件。

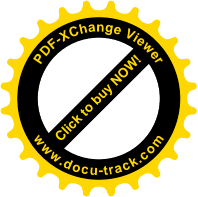
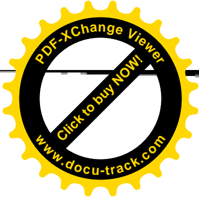
```
In [1]: import filecmp

In [2]: filecmp.cmp("file0.txt", "file1.txt")
Out[2]: False

In [3]: filecmp.cmp("file0.txt", "file00.txt")
Out[3]: True
```

可以看到，在对*file0.txt*和*file00.txt*进行比较时，cmp函数返回True。在对*file1.txt*和*file0.txt*进行比较时返回False。

dircmp函数有一些属性，这些属性可以报告目录树之间的差异。这里不会对每一个属性都进行介绍，但是我们创建了一些非常有用的示例。例如，在目录*/tmp*中创建两个子目录，从先前的示例中复制文件到每一个目录中。在dirB中，我们创建了另一个名为*file11.txt*的文件，并且在其中放入数字“11”：

```
In [1]: import filecmp

In [2]: pwd
Out[2]: '/private/tmp'

In [3]: filecmp.dircmp("dirA", "dirB").diff_files
Out[3]: []

In [4]: filecmp.dircmp("dirA", "dirB").same_files
Out[4]: ['file1.txt', 'file00.txt', 'file0.txt']
```


www.TopSage.com

```
In [5]: filecmp.dircmp("dirA", "dirB").report()
diff dirA dirB
Only in dirB : ['file11.txt']
Identical files : ['file0.txt', 'file00.txt', 'file1.txt']
```

你或许有点奇怪，尽管我们创建了*file11.txt*文件，且该文件中的信息是独一无二的，但是diff_files没有匹配。原因是diff_files仅对文件名相同的文件进行比较。

接下来查看一下same_files的输出结果，注意，它仅对两个目录中的相同文件进行报告。最后，我们对上一示例生成了一个报告，包括了两个目录之间差异的细目。这里仅是对filecmp模块功能作了一个简单介绍，因此我们建议你查阅Python标准库文档从而获得全面的知识，以弥补本书中介绍的不足。

使用os.list

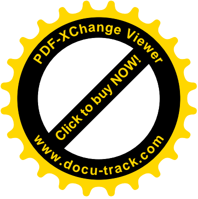
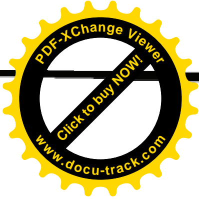
另外一个轻量级的目录比较方法是使用os.listdir。你可以将os.listdir视为ls命令，该命令返回找到的文件列表。因为Python支持许多可以对列表进行处理的有趣的方法，你可以通过将列表转换为一个集合后从一个集合中去掉另一个，使用os.listdir来查看目录自身的差异。下面是一个示例，显示了在IPython中是如何实现的：

```
In [1]: import os

In [2]: dirA = set(os.listdir("/tmp/dirA"))

In [3]: dirA
Out[3]: set(['file1.txt', 'file00.txt', 'file0.txt'])

In [4]: dirB = set(os.listdir("/tmp/dirB"))

In [5]: dirB
Out[5]: set(['file1.txt', 'file00.txt', 'file11.txt', 'file0.txt'])

In [6]: dirA - dirB
Out[6]: set([])

In [7]: dirB-dirA
Out[7]: set(['file11.txt'])
```

这个示例中，我们使用了一个简洁的技巧来将两个列表转换为集合，然后对集合进行减操作来查找差异。注意，line[7]之所以返回了*file11.txt*，是因为dirB是dirA的超集。而在line[6]中之所以结果为空，是因为dirA包含了dirB的全部内容。使用集合也可以方便地创建两个数据结构的合并，具体做法是：参照另一个目录减去一个目录的全路径，然后对差异进行复制。我们在下一章中将讨论数据的合并。

但是，这个方法也有很大的局限性。当两个文件同名时会导致错误。例如一个文件只有0字节，而另一个同名的文件却有200G。在下一节中，我们将介绍一个更好的查找两个目录之间差异并进行内容合并的方法。

www.TopSage.com

合并数据

如果不想简单地对数据文件进行比较，而是想对两个目录树进行合并，该怎么办呢？当要把一个目录树的内容合并到另一个目录树，而不创建任何副本时，这是经常会出现的问题。

你可以直接复制一个目录中的文件到目标目录，然后对目录进行复制，但是在第一步不进行复制将会效率更高。一个合理而简单的解决方法是使用filecmp模块的dircmp函数来对两个目录进行比较，并使用之前介绍的os.listdir复制唯一的结果。还可以使用MD5校验，在接下来一节中我们将介绍MD5校验。

MD5 校验和比较

对一个文件执行MD5检验和（checksum）并拿它与另一个文件比较就像是使用火箭筒向目标射击一样。虽然逐字节地比较是百分之百地精确，但也无法匹敌MD5检验和这个重量级武器。例6-7显示了该函数如何获得指定文件路径，并返回校验和。

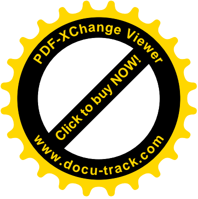
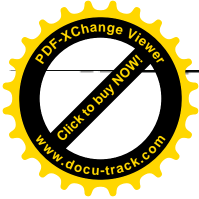
例6-7：执行文件MD5校验和

```
import hashlib

def create_checksum(path):
    """
    Reads in file. Creates checksum of file line by line.
    Returns complete checksum total for file.
    """
    fp = open(path)
    checksum = hashlib.md5()
    while True:
        buffer = fp.read(8192)
        if not buffer:break
        checksum.update(buffer)
    fp.close()
    checksum = checksum.digest()
    return checksum
```

下面是一个迭代示例，该示例在IPython中使用该函数对两个文件进行比较：

```
In [2]: from checksum import createChecksum

In [3]: if createChecksum("image1") == createChecksum("image2"):
...:     print "True"
...:
...:
True

In [5]: if createChecksum("image1") == createChecksum("image_unique"):
print "True"
```


www.TopSage.com

```
...:
...:
```

在示例中，文件的校验是通过手工对比完成的。但是我们也可以使用之前编写的代码，该代码返回路径列表，对目录树进行迭代比较并得到副本数。此外，还可以创建API，通过IPython交互测试我们的解决方案，如果证明确实可行，就可以创建另一个模块。例6-8展示了查找重复的代码。

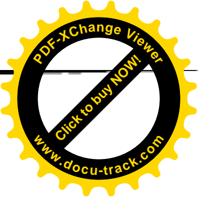
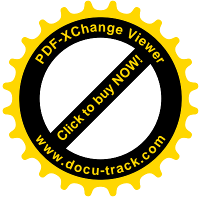
例6-8：执行目录MD5校验和以查找重复

```
In [1]: from checksum import createChecksum

In [2]: from diskwalk_api import diskwalk

In [3]: d = diskwalk('/tmp/duplicates_directory')

In [4]: files = d.enumeratePaths()

In [5]: len(files)
Out[5]: 12

In [6]: dup = []

In [7]: record = {}

In [8]: for file in files:
   compound_key = (getsize(file),create_checksum(file))
   if compound_key in record:
       dup.append(file)
   else:

             record[compound_key] = file

....:
....:
In [9]: print dup
['/tmp/duplicates_directory/image2']
```

这段代码中，仅有一段在之前的示例中没有介绍过，也就是line[8]。创建一个空字典，然后使用一个关键字来存储产生的校验结果，这是用来判断检验和是否已存在的简单方法。如果存在，将文件加入重复列表。现在，将代码分为可再次使用的若干段。例6-9显示了如何实现。

例6-9：查找重复

```
from checksum import create_checksum
from diskwalk_api import diskwalk
from os.path import getsize

def findDupes(path = '/tmp'):
 dup = []
 record = {}
 d = diskwalk(path)
 files = d.enumeratePaths()
```

```
    for file in files:
        compound_key = (getsize(file),create_checksum(file))
        if compound_key in record:
            dup.append(file)
        else:
            #print "Creating compound key record:", compound_key
            record[compound_key] = file
    return dup

if __name__ == "__main__":
    dupes = findDupes()
    for dup in dupes:
        print "Duplicate: %s" % dup
```

执行该脚本时，得到的输出如下：

```
[ngift@Macintosh-7][H:10157][J:0]# python find_dupes.py
Duplicate: /tmp/duplicates_directory/image2
```

从这里你可以看到，哪怕是非常小的一段代码也可以复用成功。现在有了一个通用模块，可以取得目录树并返回所有重复文件的列表。这用起来非常方便。接下来我们再进一步，实现自动删除重复的功能。

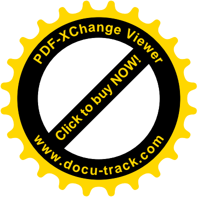
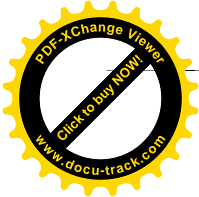
正如你可以使用os.remove (file)一样，在Python中删除文件非常简单。这个示例中，在*/tmp*目录中有一个10MB大小的文件，我们使用os.remove (file)删除其中的一个：

```
In [1]: import os

In [2]: os.remove("10
10mbfile.0 10mbfile.1 10mbfile.2 10mbfile.3 10mbfile.4
10mbfile.5 10mbfile.6 10mbfile.7 10mbfile.8

In [2]: os.remove("10mbfile.1")

In [3]: os.remove("10
10mbfile.0 10mbfile.2 10mbfile.3 10mbfile.4 10mbfile.5
10mbfile.6 10mbfile.7 10mbfile.8
```

值得注意的是，IPython中的tab自动完成功能允许我们查看匹配并为我们显示图像文件的名称。os.remove (file)方法是静态的、持久的。这或许是你希望的，但也可能不是。记住这一点，可以实现一个简单的方法来删除重复的文件，之后再对其进行修改。由于在IPython中测试交互代码十分简单，我们写一个test函数，然后进行测试：

```
In [1]: from find_dupes import findDupes

In [2]: dupes = findDupes("/tmp")

In [3]: def delete(file):
            import os
   ...:     print "deleting %s" % file
   ...:     os.remove(file)
```


www.TopSage.com

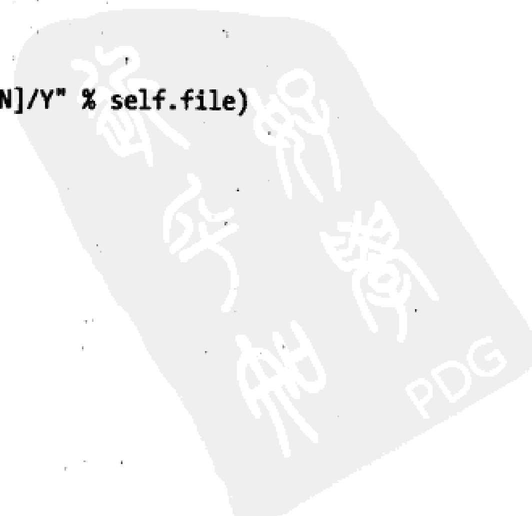



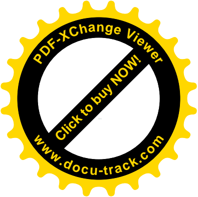
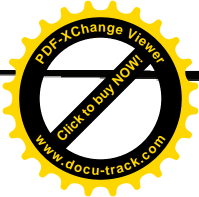
```
...:
...:

In [4]: for dupe in dupes:
...:      delete(dupe)
...:
...:
In [5]: for dupe in dupes:
delete(dupe)
...:
...:
deleting /tmp/10mbfile.2
deleting /tmp/10mbfile.3
deleting /tmp/10mbfile.4
deleting /tmp/10mbfile.5
deleting /tmp/10mbfile.6
deleting /tmp/10mbfile.7
deleting /tmp/10mbfile.8
```

在这个示例中，通过添加打印自动删除文件的语句，增加了delete模块的复杂度。我们已经创建了不少可复用的代码，但不会马上停止。下一步还要创建另一个模块，该模块可以作为一个file对象，执行相关删除操作。它甚至不需要查找重复，可以用来删除任何内容。参见例6-10。

例6-10：删除模块

```
#!/usr/bin/env python
import os

class Delete(object):
"""Delete Methods For File Objects"""

def __init__(self, file):
   self.file = file

def interactive(self):
   """interactive deletion mode"""

   input = raw_input("Do you really want to delete %s [N]/Y" % self.file)
   if input.upper():
       print "DELETING: %s" % self.file
       status = os.remove(self.file)
   else:
       print "Skipping: %s" % self.file
   return
def dryrun(self):
   """simulation mode for deletion"""

   print "Dry Run: %s [NOT DELETED]" % self.file
   return

def delete(self):

   """Performs a delete on a file, with additional conditions
   """
```



```
        print "DELETING: %s" % self.file
        try:
            status = os.remove(self.file)
        except Exception, err:
            print err
            return status

if __name__ == "__main__":
    from find_dupes import findDupes
    dupes = findDupes('/tmp')

for dupe in dupes:
    delete = Delete(dupe)
    #delete.dryrun()
    #delete.delete()
    #delete.interactive()
```

在这个模块中，你将看到三种不同的删除类型。交互（interactive）删除方法提示用户对每一个即将删除的文件进行确认。这似乎有点烦人，但是当其他程序员维护或升级代码时这是非常好的保护措施。

Dry run方法模拟了删除操作。实际上，该方法最终执行了删除方法，能够永久地删除文件。在模块的底部，有一个被注释掉的示例，该示例展示了如何使用这三种不同的删除方法。下面的示例演示了每一方法的执行过程：

- Dry run

```
ngift@Macintosh-7][H:10197][J:0]# python delete.py
Dry Run:  /tmp/10mbfile.1 [NOT DELETED]
Dry Run:  /tmp/10mbfile.2 [NOT DELETED]
Dry Run:  /tmp/10mbfile.3 [NOT DELETED]
Dry Run:  /tmp/10mbfile.4 [NOT DELETED]
Dry Run:  /tmp/10mbfile.5 [NOT DELETED]
Dry Run:  /tmp/10mbfile.6 [NOT DELETED]
Dry Run:  /tmp/10mbfile.7 [NOT DELETED]
Dry Run:  /tmp/10mbfile.8 [NOT DELETED]
```

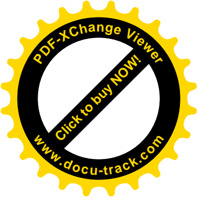
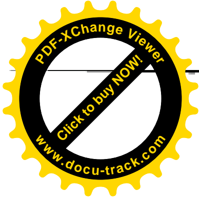
- Interactive

```
ngift@Macintosh-7][H:10201][J:0]# python delete.py
Do you really want to delete /tmp/10mbfile.1 [N]/YY
DELETING:  /tmp/10mbfile.1
Do you really want to delete /tmp/10mbfile.2 [N]/Y
Skipping:  /tmp/10mbfile.2
Do you really want to delete /tmp/10mbfile.3 [N]/Y
```

- Delete

```
[ngift@Macintosh-7][H:10203][J:0]# python delete.py
DELETING:  /tmp/10mbfile.1
DELETING:  /tmp/10mbfile.2
DELETING:  /tmp/10mbfile.3
DELETING:  /tmp/10mbfile.4
```


www.TopSage.com

```
DELETING:  /tmp/10mbfile.5
DELETING:  /tmp/10mbfile.6
DELETING:  /tmp/10mbfile.7
DELETING:  /tmp/10mbfile.8
```

在处理数据的时候，你或许会发现使用封装技术非常简便。因为这样可以通过充分地抽象现有的工作，使问题一般化，从而避免可能出现的新问题。在这种情况下，为了自动删除重复文件，创建了一个普通的查找和删除文件的模块。还可以创建另一个工具来获得文件对象并应用某种形式的压缩。我们实际上仅介绍了该示例的一小部分。

对文件和目录的模式匹配

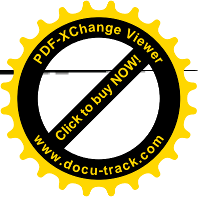
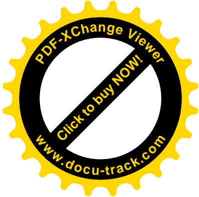
到目前为止，已经介绍了如何处理目录和文件，并执行查找重复、删除目录、移动目录等操作。要掌握了目录树之后，接下来要学习使用模式进行匹配，不论是单独使用还是与之前的技术联合使用。正如Python中的其他操作一样，对文件扩展名或文件名执行模式匹配也十分简单。这一节将介绍通用模式匹配的问题，应用之前使用的技术来创建简单但却功能强大的复用工具。

系统管理员们有一个常见的问题要解决，那就是他们需要追踪、删除、移动、重命名或是复制某一类型的文件。在Pyton中，最直接的方法是使用fnmatch模块或是glob模块。这两个模块之间的主要差异是，对于Unix通配符，fnmatch返回True或False，而glob返回匹配模式的路径列表。作为选择，正则表达式可以用来创建一些更复杂的匹配工具。请参考第3章以获得更多关于使用正则表达式来匹配模式的说明。

例6-11演示了fnmatch和glob是如何使用的。这里将通过从diskwalk_api模块加载diskwalk复用之前的代码。

例6-11：分别使用fnmatch和glob查找文件匹配

```
In [1]: from diskwalk_api import diskwalk

In [2]: files = diskwalk("/tmp")

In [3]: from fnmatch import fnmatch

In [4]: for file in files:
   ...:     if fnmatch(file,"*.txt"):
   ...:         print file
   ...:
   ...:
/tmp/file.txt

In [5]: from glob import glob

In [6]: import os

In [7]: os.chdir("/tmp")
```

```
In [8]: glob("*")
Out[8]: ['file.txt', 'image.iso', 'music.mp3']
```

在示例中，复用了之前的diskwalk模块后得到一个列表，列表中包含/*tmp*目录中的全路径。之后，使用fnmatch来决定是否每个文件都可以匹配模式“*.txt”。glob模块则不同，它是严格地“全局”匹配一个模式，并返回全路径。glob是一个比fnmatch更高级的函数。对于差异比较而言，两者都是非常有用的工具。

当与其他代码联合使用，创建一个在目录树中搜索数据的过滤器时，fnmatch函数尤其有用。在处理目录树时，你往往会对匹配某些模式的文件进行处理。明白这一点之后，我们就可以解决一个经典的系统管理问题，即对目录树中的所有匹配文件重命名。记住，重命名实际上与删除、压缩或对文件进行处理一样简单。其模式如下：

1. 取得目录中某个文件的路径。
2. 执行某种过滤器的可选类别，这会涉及许多过滤器，如文件名、扩展名、大小、唯一值等。
3. 在过滤器上执行操作，包括复制、删除，压缩，读取等。例6-12演示了这操作。

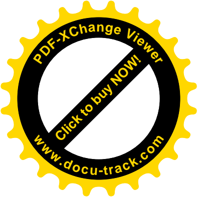
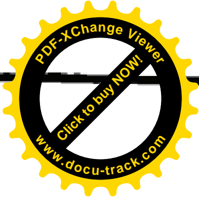
例6-12：将目录树中全部MP3 文件重命名为文本文件

```
In [1]: from diskwalk_api import diskwalk

In [2]: from shutil import move

In [3]: from fnmatch import fnmatch

In [4]: files = diskwalk("/tmp")

In [5]: for file in files:
            if fnmatch(file, "*.mp3"):
            #here we can do anything we want, delete, move, rename...hmmm rename
                move(file, "%s.txt" % file)
In [6]: ls -l /tmp/
total 0
-rw-r--r-- 1 ngift wheel 0 Apr 1 21:50 file.txt
-rw-r--r-- 1 ngift wheel 0 Apr 1 21:50 image.iso
-rw-r--r-- 1 ngift wheel 0 Apr 1 21:50 music.mp3.txt
-rw-r--r-- 1 ngift wheel 0 Apr 1 22:45 music1.mp3.txt
-rw-r--r-- 1 ngift wheel 0 Apr 1 22:45 music2.mp3.txt
-rw-r--r-- 1 ngift wheel 0 Apr 1 22:45 music3.mp3.txt
```

通过使用上面的示例，我们只用了四行简单的Python代码，就将一个由mp3组成的目录树中的全部文件重命名为文本文件。如果你是一个没有阅读过BOFH或Bastard Operator From Hell中任何一个章节的系统管理员，那么，可能对这些代码没有明显的感觉。

想象一下，有一个生产文件服务器，具有高性能的I/O存储功能，但却容量有限。由于一两个不良用户在里面放了几百GB的 MP3文件导致它的空间经常被占满。你可以对每一

www.TopSage.com

个用户所使用的文件空间大小指定一个配额，但是这会引来更多麻烦。一个解决方案是每晚创建一个cron作业，查找所有的MP3文件并对它们做一些操作。每周一，将它们重命名为文本文件，每周二将它们压缩为ZIP文件，每周三，则将它们移动到*/tmp*目录中，到了星期四，则会进行删除，并给文件的所有者发送一个已删除的MP3文件列表。当然，除非你是公司的老板，否则我们不会建议你真的这么去做。而实际上，在BOFH，早期的代码就是这样进行操作的。

包装rsync

你可能已经知道，rsync是一个最初由Andrew Tridgell和Mackerra编写的命令行工具。2007年下半年，rsync version 3测试版发布，它包括了比原始版本更多的类型选项。

多年来，我们都使用rsync作为把数据从点A移动到点B的主要工具。我们建议你仔细阅读rsync的帮助和选项，因为rsync可能会是为系统管理员编写的最有用的命令行工具。

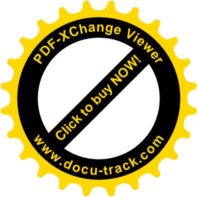
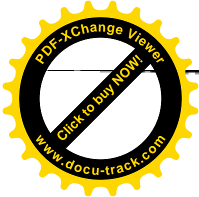
正如前面提到的，Python可以帮助控制或粘贴rsync的方法。有一个问题是如何确保数据在计划的时间内被复制。在需要对文件服务器之间进行数TB的数据同步的情况下，我们不希望去手工监视复制的进程，这也就是Python真正发挥作用的时候。

你可以使用Python为同步添加一定的智能，或是根据需要进行自定义。将Python作为glue代码使用能够让UNIX工具能够完成通常情况下难以完成的工作，你可以创建各种高灵活性和可定制的工具，正所谓想象力有多远，就可以走多远。例6-13显示了一个非常简单的示例，演示了如何对rsync进行包装（wrap）。

例6-13：rsync的简单包装

```
#!/usr/bin/env python
#wraps up rsync to synchronize two directories
from subprocess import call
import sys

source = "/tmp/sync_dir_A/" #Note the trailing slash
target = "/tmp/sync_dir_B"
rsync = "rsync"
arguments = "-a"
cmd = "%s %s %s %s" % (rsync, arguments, source, target)


def sync():

ret = call(cmd, shell=True)
if ret !=0:
    print "rsync failed"
    sys.exit(1)
sync()
```

www.TopSage.com

这个示例是一个对两个目录进行同步的代码，如果命令执行失败，将打印失败信息。可以做一些更有趣的事，解决自己会频繁遇到的问题。我们经常需要对两个非常大的目录进行同步，但不想整晚都去监视数据同步过程。然而，如果没有监视同步过程，一旦进程在执行过程中被破坏、数据也被破坏、整晚时间都被浪费，后果不堪设想，处理过程也需要在第二天重新开始。这种情况下，可以使用Python创建一个更先进的、高度机动的rsync命令。

一个高度机动的rsync命令又会如何去做呢？如果你正在监视两个目录之间的同步，它能完成所有需要做的工作：它会一直对目录进行同步，直到完成，然后会发送一个邮件告之任务已完成。例6-14中的 rsync代码实现了这一功能。

例6-14：一个直至工作完成才结束的rsync命令

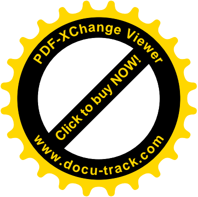
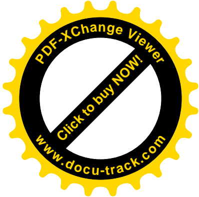
```
#!/usr/bin/env python
#wraps up rsync to synchronize two directories
from subprocess import call
import sys
import time

"""this motivated rsync tries to synchronize forever"""

source = "/tmp/sync_dir_A/" #Note the trailing slash
target = "/tmp/sync_dir_B"
rsync = "rsync"
arguments = "-av"
cmd = "%s %s %s %s" % (rsync, arguments, source, target)

def sync():
while True:
    ret = call(cmd, shell=True)
    if ret !=0:
        print "resubmitting rsync"
        time.sleep(30)
    else:
     print "rsync was succesful"
     subprocess.call("mail -s 'jobs done' bofh@example.com", shell=True)
     sys.exit(0)
sync()
```

这个示例十分简单，并且包含了部分硬编码数据，但是这是一个非常有用工具，使用该工具可以使一些需要人工完成的事情自动化。此外，你还可以添加一些其他特性，包括设置下次尝试的时间间隔，连接次数，以及检测所连接主机的磁盘空间使用情况等。

元数据：关于数据的数据

绝大多数系统管理员所关注的不仅局限于数据，还包括与数据相关的数据。元数据，或者说是关于数据的数据，通常比数据本身更为重要。这里给出一个示例，在电影和电视

www.TopSage.com

中，相同的数据经常出现在文件系统的多个位置或者是多个文件系统上。跟踪这些数据经常涉及创建元数据类型管理系统。

描述文件是如何组织和使用的数据，对于应用程序、动画生产线或是恢复一个备份来说，都是十分关键的。Python可以在这方面有所作为，因为，在Python中使用元数据和写元数据都非常简单。

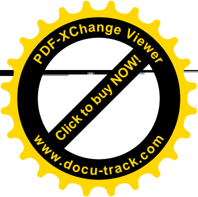
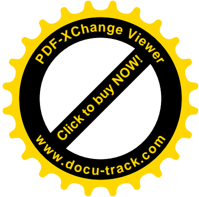
现在看一个使用ORM，SQLAlchemy创建的关于文件系统的元数据。幸运的是，SQLAlchemy的文档非常好，并且SQLAlchemy可以与SQLite一起使用。我们认为，这对于创建自定义的元数据解决方案是一个非常棒的组合。

在上面的示例中，我们体验了实时搜索文件系统，执行操作和查询路径。搜索由几百万个文件组成的大文件系统是非常耗时的，因此实时搜索非常有意义。例6-15中通过将ORM和前面介绍的目录遍历技术结合，展示了一个基本的元数据系统。

例6-15：利用SQLAlchemy创建文件系统的元数据

```
#!/usr/bin/env python
from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey
from sqlalchemy.orm import mapper, sessionmaker
import os

#path
path = " /tmp"

#Part 1: create engine
engine = create_engine('sqlite:///:memory:', echo=False)

#Part 2: metadata
metadata = MetaData()

filesystem_table = Table('filesystem', metadata,
  Column('id', Integer, primary_key=True),
  Column('path', String(500)),
  Column('file', String(255)),
)

metadata.create_all(engine)

#Part 3: mapped class
class Filesystem(object):

def __init__(self, path, file):
    self.path = path
    self.file = file

def __repr__(self):
    return "[Filesystem('%s','%s')]" % (self.path, self.file)

#Part 4: mapper function
mapper(Filesystem,filesystem_table)
```

www.TopSage.com

```
#Part 5: create session
Session = sessionmaker(bind=engine, autoflush=True, transactional=True)
session = Session()

#Part 6: crawl file system and populate database with results
for dirpath, dirnames, filenames in os.walk(path):
    for file in filenames:
        fullpath = os.path.join(dirpath, file)
        record = Filesystem(fullpath, file)
        session.save(record)

#Part 7: commit to the database
session.commit()

#Part 8: query
for record in session.query(Filesystem):
    print "Database Record Number: %s, Path: %s , File: %s " \
    % (record.id,record.path, record.file)
```

可以将代码视为过程的集合，一段代码可以看作由一个过程接着另一个过程组成。在第一部分，创建了一个引擎，这是定义将要使用的数据库的非常不错的方法。在第二部分，定义了一个元数据实例，并且创建了数据库表。在第三部分，创建了一个类，该类会映射到所创建的数据库中的数据表。在第四部分，调用放入ORM的映射函数，它实际上已经将类映射到数据表上了。在第五部分，创建到数据库的会话。需要注意的是，这里设置了一些关键字参数，包括autoflush和transactional。

现在已经非常明确地描述了ORM的安装，在第六部分，完成常规操作，在遍历一个目录树时，获得文件名和完整路径。尽管这里会有一些改动。注意，这里在数据库中为每一个完整路径和每一个文件创建了一个记录，并且在创建时，保存了每一个新创建的记录。之后在第七部分，提交这一处理到SQLite数据库中。

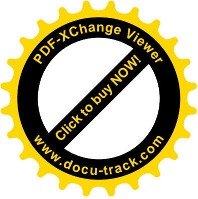
最后，在第八部分，在Python中执行了一个查询，返回了在数据库中放置的记录结果。这一示例创建自定义SQLAchemy元数据解决方案，对你的公司和客户而言都是非常好的方法。也可以扩展这一代码来做一些更有趣的事，如执行相关查询或将结果写入文件等。

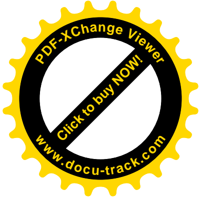
存档、压缩、映像和恢复

处理大量数据是系统管理员每天都要面对的问题。他们通常会使用tar、dd、gzip、bzip、bzip2、hdiutil、asr以及其他工具来完成这些工作。

不管你信不信，“连电池都包括在内”的Python标准库对TAR文件、zlib文件和gzip文件都提供内建支持。如果压缩和归档是你的目标，那么Python为此提供了丰富的工具。让我们来看一个重要的归档包：tar，并且一起了解标准库是如何实现tar的。

www.TopSage.com





使用tarfile模块创建TAR归档

创建一个TAR归档包非常简单，甚至显得过于简单了。在例6-16中，作为示例，我们创建了一个非常大的文件。这里用到的语法对使用者非常友好，在这一点上甚至超过了tar命令本身。

例6-16：创建大文本文件

```
In [1]: f = open("largeFile.txt", "w")

In [2]: statement = "This is a big line that I intend to write over and over again."
In [3]: x = 0
In [4]: for x in xrange(20000):
....:       x += 1
....:       f.write("%s\n" % statement)
....:
....:
In [4]: ls -l
-rw-r--r-- 1 root root 1236992 Oct 25 23:13 largeFile.txt
```

好了，现在有了一个大文件，让我们对其使用TAR命令。参见例6-17。

例6-17：对文件内容使用TAR命令

```
In [1]: import tarfile

In [2]: tar = tarfile.open("largefile.tar", "w")

In [3]: tar.add("largeFile.txt")

In [4]: tar.close()

In [5]: ll

-rw-r--r-- 1 root root 1236992 Oct 25 23:15 largeFile.txt
-rw-r--r-- 1 root root 1236992 Oct 26 00:39 largefile.tar
```

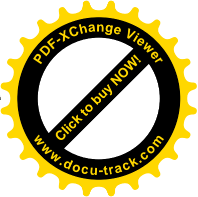
可以看到，与普通的tar命令相比，vanilla TAR归档有更为简单的语法。这确实也令使用IPython shell来完成每天全部的系统管理工作成为可能。

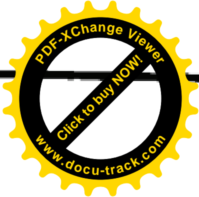
由于使用Python来创建TAR文件非常方便，因此仅对一个文件使用TAR，几乎没有什么意义。使用本章中多次用到的目录遍历模式，我们对*/tmp*目录创建了一个TAR文件，实现方法是遍历目录树然后将每一个文件添加到*/tmp*目录的归档包中。参见例6-18。

例6-18：对目录树使用TAR命令

```
In [27]: import tarfile

In [28]: tar = tarfile.open("temp.tar", "w")

In [29]: import os
```

www.TopSage.com


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

```
In [30]: for root, dir, files in os.walk("/tmp"):
....:     for file in filenames:
....:
KeyboardInterrupt

In [30]: for root, dir, files in os.walk("/tmp"):
for file in files:
....:         fullpath = os.path.join(root,file)
....:         tar.add(fullpath)
....:
....:

In [33]: tar.close()
```

通过遍历目录来添加目录树中的内容非常简单方便，因为它可以与本章中介绍的其他技术相结合。你或许正在归档一个全部由媒体文件组成的目录。如果对重复的文件进行归档，显然有些不明智，因此，在创建TAR文件之前，需要将重复的文件替换为符号链接文件或做其他需要进行的操作。使用本章介绍的方法，可以很容易地编写实现上述目标的代码，从而节省一些磁盘空间。

由于创建一个普通的TAR文档有些乏味，让我们换个口味使用bzip2压缩，它会使你的CPU全速工作，甚至会报怨怎么会有这么多工作。bzip2压缩算法就十分了不起。下面看一个示例，相信它会给我们留下深刻印象。

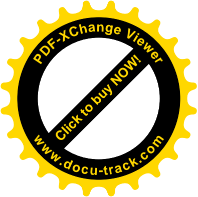
例6-19：创建bzip2 的TAR 归档

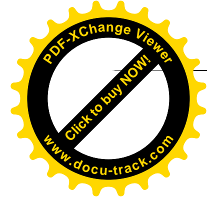
```
In [1: tar = tarfile.open("largefilecompressed.tar.bzip2", "w|bz2")

In [2]: tar.add("largeFile.txt")

In [3]: ls -h
foo1.txt fooDir1/ largeFile.txt largefilecompressed.tar.bzip2*
foo2.txt fooDir2/ largefile.tar

In [4]: tar.close()

In [5]: ls -lh

-rw-r--r-- 1 root root 61M Oct 25 23:15 largeFile.txt
-rw-r--r-- 1 root root 61M Oct 26 00:39 largefile.tar
-rwxr-xr-x 1 root root 10K Oct 26 01:02 largefilecompressed.tar.bzip2*
```

bzip2可以将61M的文本文件压缩到只有10K大小，多么令人惊叹。当然这也不是零代价的，因为即使在双核AMD系统上，它也会花几分钟时间才能完成文件压缩。

让我们总结一下，然后使用其他可用的选项进行另一次压缩归档。接下来使用gzip。gzip的语法略有不同。参见例6-20。

www.TopSage.com


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

例6-20：创建gzip的TAR存档

```
In [10]: tar = tarfile.open("largefile.tar.gzip", "w|gz")

In [11]: tar.add("largeFile.txt")

In [12]: tar.close()

In [13]: ls -lh

-rw-r--r-- 1 root root 61M Oct 26 01:20 largeFile.txt
-rw-r--r-- 1 root root 61M Oct 26 00:39 largefile.tar
-rwxr-xr-x 1 root root 160K Oct 26 01:24 largefile.tar.gzip*
```

这个gzip文件依然是难以置信地小，只有160K左右。但是在我的机器上，完成压缩TAR包的创建只用了几秒钟时间。看起来在大多数环境中，gzip都非常适用。

使用tarfile模块检查TAR文件内容

现在我们有了创建TAR文件的工具，接下来，对TAR文件的内容进行检查就变得非常有意义了。盲目地创建TAR文件却不做内容的检查不是一件好事。作为一名系统管理员，无论你干了多久，都有可能会为一个坏的备份而气恼或是因创建了一个坏的备份而被指责。

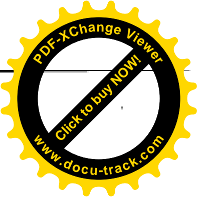
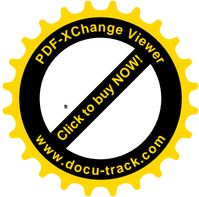
为了将这一问题讲得更生动，进一步强调检验TAR文档内容的重要性，这里讲一个朋友的故事，让我们称之为“TAR文档丢失事件”。故事中人物的姓名、身份和事情都是虚构的，如果这个故事与现实生活雷同，纯属偶然。

这个朋友作为系统管理员，工作在第一流的电视演播室，为一个由某个狂人领导的部门提供支持服务。这个领导以不讲真话，行事冲动，甚至有些疯狂出名。一旦出现由他导致的错误，例如错过了客户的最后期限，不能按指定的说明书完成工作，他会很容易地撒谎并将责任怪罪到其他人头上。这个替罪羊经常是我们的朋友，系统管理员。

不幸的是，这个朋友一直充当着替罪羊的角色。他也曾想过离开，再找一份新的工作，但是，他已经在这个演播室工作许多年了，在这里有许多朋友，他不想因为一些临时的坏遭遇而失去所有的一切。于是他决定从最基本的工作做起，并为此建立了一个日志系统，可以对那个狂人的所有文档自动生成TAR文档，并进行分类。他认定，那个狂人迟早会因为错过了某个最后期限，需要找个借口怪罪到他头上。

一天，我们的朋友Willian从他的老板那里接到一个电话，“Willian，我需要你立刻到我的办公室来，备份出了点问题。”Willian立即走进他的办公室，被告之那个狂人Alex，责怪Willian破坏了给他的文档，导致他错过了与客户约定的最后期限，这令Alex的老板Bob非常不安。老板告诉Willian，Alex告诉他备份里只有空的和受损的文件，而他本来是依靠这些文档来工作的。这时，Willian告诉老板，他早料到自己有可能会因为文档被

www.TopSage.com

破坏而遭到指责，所以悄悄地写了一个Python代码，能够对所有Alex创建的TAR文档进行检测，同时记录文件备份前后的文件属性。检测表明，Alex从来没有创建一个演示文档，并且几个月来，他创建的目录下内容一直是空的。

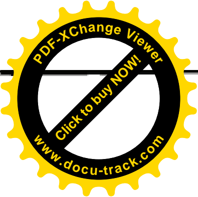
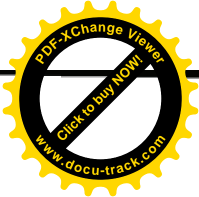
当Alex面对这些信息时，很快就收回之前的言论，并寻找转移注意力的新焦点。不幸的是，这是Alex最后的救命稻草，几个月后，他再也没有出现在工作场合。他或者离开了或者被解雇了，但这已经再不重要，我们的朋友Willian已经成功解决了“TAR文档丢失事件”。

从这个故事中得出的道理就是，当你处理备份时，可以将它们视为核武器，因为它们能够帮你避免许多从未想到的危险。

下面是一些检测TAR文件内容的方法，这些TAR文件是我们之前创建的：

```
In [1]: import tarfile

In [2]: tar = tarfile.open("temp.tar","r")

In [3]: tar.list()
-rw-r--r-- ngift/wheel          2 2008-04-04 15:17:14 tmp/file00.txt
-rw-r--r-- ngift/wheel          2 2008-04-04 15:15:39 tmp/file1.txt
-rw-r--r-- ngift/wheel          0 2008-04-04 20:50:57 tmp/temp.tar
-rw-r--r-- ngift/wheel          2 2008-04-04 16:19:07 tmp/dirA/file0.txt
-rw-r--r-- ngift/wheel          2 2008-04-04 16:19:07 tmp/dirA/file00.txt
-rw-r--r-- ngift/wheel          2 2008-04-04 16:19:07 tmp/dirA/file1.txt
-rw-r--r-- ngift/wheel          2 2008-04-04 16:19:52 tmp/dirB/file0.txt
-rw-r--r-- ngift/wheel          2 2008-04-04 16:19:52 tmp/dirB/file00.txt
-rw-r--r-- ngift/wheel          2 2008-04-04 16:19:52 tmp/dirB/file1.txt
-rw-r--r-- ngift/wheel          3 2008-04-04 16:21:50 tmp/dirB/file11.txt

In [4]: tar.name
Out[4]: '/private/tmp/temp.tar'

In [5]: tar.getnames()
Out[5]:
['tmp/file00.txt',
'tmp/file1.txt',
'tmp/temp.tar',
'tmp/dirA/file0.txt',
'tmp/dirA/file00.txt',
'tmp/dirA/file1.txt',
'tmp/dirB/file0.txt',
'tmp/dirB/file00.txt',
'tmp/dirB/file1.txt',
'tmp/dirB/file11.txt']

In [10]: tar.members
Out[10]:
[<TarInfo 'tmp/file00.txt' at 0x109eff0>,
 <TarInfo 'tmp/file1.txt' at 0x109ef30>,
 <TarInfo 'tmp/temp.tar' at 0x10a4310>,
 <TarInfo 'tmp/dirA/file0.txt' at 0x10a4350>,
```

www.TopSage.com

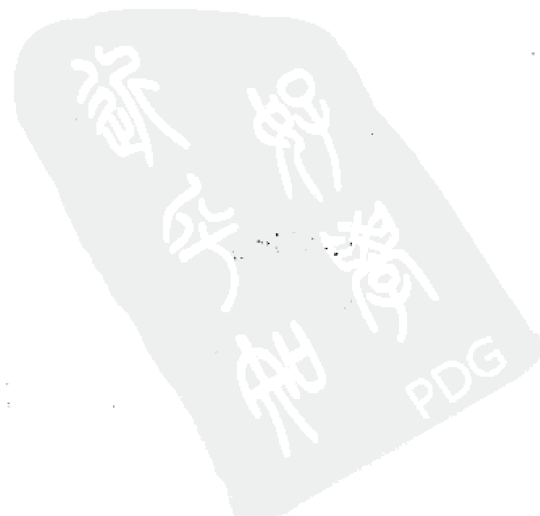
```
<TarInfo 'tmp/dirA/file00.txt' at 0x10a43b0>,
<TarInfo 'tmp/dirA/file1.txt' at 0x10a4410>,
<TarInfo 'tmp/dirB/file0.txt' at 0x10a4470>,
<TarInfo 'tmp/dirB/file00.txt' at 0x10a44d0>,
<TarInfo 'tmp/dirB/file1.txt' at 0x10a4530>,
<TarInfo 'tmp/dirB/file11.txt' at 0x10a4590>]
```

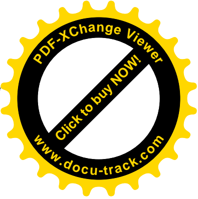
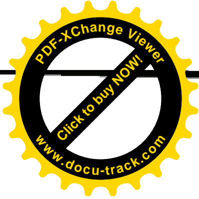
这些示例演示了如何检查TAR文档中的文件名，这在数据验证脚本中是十分有效的。提取文件不是非常费劲的工作。如果想要把整个TAR文档中的内容提取到当前工作目录下，可以简单地执行下面的操作：

```
In [60]: tar.extractall()

drwxrwxrwx 7 ngift wheel     238 Apr 4 22:59 tmp/
```

如果你有疑虑，可以增加这个操作：提取归档文件的内容，并对来自归档中的文件执行随机MD5校验，并将结果与备份之前生成的MD5校验和进行比较。这对于监测数据是否完整非常有效。

没有任何归档解决方法是完全可靠的。至少，对文档进行随机检测应当是自动进行的，每一个单独的文档应该在创建之后被重新打开并验证其有效性。



第7章

SNMP

引言

如果你是一名系统管理员，可以说SNMP能改变你的生活。使用SNMP的好处不像编写几行Python代码来对日志文件进行解析那样立竿见影，但是当一个SNMP基础结构被建立起来之后，在其上工作会让你感到惊喜。

在这一章中，我们涉及SNMP的这样一些方面：自动发现，投票/监测，写代理，设备控制，以及最后的企业SNMP整合。当然，所有这些事情都是在Python下完成的。

如果你对SNMP不熟悉，或许需要重新温习SNMP。我们强烈建议阅读由Douglas Mauro和 Kevin Schmidt（O'Reilly）编著的《Essential SNMP》，或者至少将这本书放在手边以便查阅。一本好的SNMP参考书对于真正理解SNMP，明白使用SNMP到底可以做些什么，是非常重要的。接下来的一节仅介绍了SNMP的一些基本知识，太多的细节超出了本书的范围。事实上，关于SNMP的内容非常丰富，需要完整的一本书才能把在Python中如何使用SNMP介绍清楚。

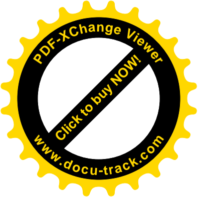
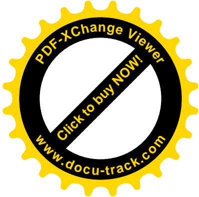
对SNMP的简要介绍

SNMP概述

总体来看，SNMP是IP网络的设备管理协议。典型地，这是通过UDP端口161和162来实现的，尽管也有极小的可能性会选择使用TCP协议。作为数据中心的现代设备支持SNMP，这表示它能够管理的不仅是交换机和路由器，也包括打印服务器，UPS，存储器等设备。

使用SNMP的基本方法是发送UDP包到主机，然后等待响应。这是监测设备非常常用的做法。使用SNMP协议也可以完成其他一些工作，例如控制设备，对特定条件进行响应的写代理。

www.TopSage.com

一些使用SNMP的非常典型的应用包括监测CPU负载，磁盘使用情况和内存空闲状态。你可以使用SNMP来管理和控制交换机，或者更进一步通过SNMP来重新加载配置文件。通常很少人知道，使用SNMP也可以监测软件，例如web应用程序和数据库。最后，SNMP还可以提供对RMON MIB的远程监控，这是支持基于数据流的监控，不同于常规的基于设备的SNMP监控。

既然已经谈到了MIB，接下来对其做进一步说明。SNMP仅是一个协议，并没有对数据进行约定。在被监测的设备上运行代理snmpd。snmpd具有一个保持追踪的对象列表。实际的对象列表由MIB（管理信息基础）进行控制。每一个代理至少实现一个MIB，并且是MIB-II，其在RFC1213中被定义。一种看待MIB的思路是将其视为一个能够将名称翻译为数字的文件，就象DNS一样，但是MIB要更为复杂一些。

MIB文件内是被管理对象的定义。每一个对象有三个属性：名称，类型和语法，以及编码。在这些属性当中，名称是第一个需要知道的内容。名称经常作为OID（对象标识）被引用。OID让你可以告诉代理需要操作的对象是什么。名称有两种类型：数字类型和文本类型（适于阅读）。大多数情况下你更愿意使用适于阅读的文本类型的OID名称，因为数字名称非常长而且很难记忆。最普通的OID是sysDescr。如果使用命令行工具snmpwalk来设置sysDescr OID的值，既可以将其设为文本也可以设为数字：

```
[root@rhel][H:4461][J:0]# snmpwalk -v 2c -c public localhost .1.3.6.1.2.1.1.1.0
SNMPv2-MIB::sysDescr.0 = STRING: Linux localhost
  2.6.18-8.1.15.el5 #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686

[root@rhel][H:4461][J:0]# snmpwalk -v 2c -c public localhost sysDescr
SNMPv2-MIB::sysDescr.0 = STRING: Linux localhost
  2.6.18-8.1.15.el5 #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686
```

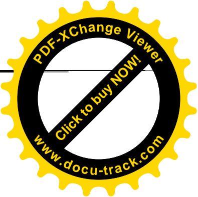
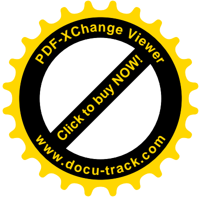
至此，我们给出了一堆术语和一个RFC，面对这么多术语你或许会比较困惑，正对是继续阅读还是放弃而犹豫不决。我们保证很快你就会感觉好一些，并在几分钟内写出代码。

SNMP安装及配置

出于简化的考虑，我们仅介绍Net-SNMP，以及绑定到Net-SNMP的对应Python。仅介绍Net-SNMP并不表示其他一些基于Python的SNMP库（包括PySNMP，这是TwistedSNMP和Zenoss都使用的）不好用。在Zenoss和TwistedSNMP中，PySNMP以异步方式使用。这是非常有效的方式，也值得介绍，但在本章中没有足够的篇幅来对这两者进行介绍。

对于Net-SNMP自身，可以使用两类不同的API。方法一使用subprocess模块来封装Net-SNMP命令行工具；方法二是使用新的Python绑定。每一个方法都有各自的优势和劣势，这取决于其在什么环境中被实现。

www.TopSage.com

最后，我们也介绍了Zenoss，它是开源的，是一个非常出色的Python下的企业级SNMP监测解决方案。通过Zenoss，你不必从头来编写SNMP管理解决方案，并且可以通过它的公共API来与其通信。为zenoss写一个插件、补丁或是扩展Zenoss本身都是可行的。

要使SNMP发挥作用，尤其是Net-SNMP,你必须首先对其进行安装。幸运的是，绝大多数Unix和Linux操作系统已经安装了Net-SNMP，因此如果需要监测一个设备，通常只需要修改*snmpd.conf*文件来适应具体需要，并启动守护进程。如果计划使用Python绑定的Net-SNMP进行开发（这正是本章将介绍的内容），需要编译源代码来安装Python绑定。如果计划封装Net-SNMP命令行工具，例如snmpget，snmpwalk，snmpdf以及其他，只要Net-SNMP已经安装，那么就不需要再做任何事情了。

可以从*http://www.oreilly.com/9780596515829*下载虚拟机，以及这本书的源代码。也可以访问*www.py4sa.com*，这是这本书的合作图书网站，它有最新的关于如何运行本节中示例的信息。

我们已经配置了具有Net-SNMP和Python的虚拟机。你可以通过使用虚拟机来运行所有的示例。如果有足够的可供使用的硬件资源，还可以制作一些虚拟机的副本，并模拟本章中提及的同时与多台主机进行通信的代码。

安装Python绑定，你需要从sourceforge.net下载Net-SNMP，并且需要Net-SNMP 5.4或更高版本。绑定不是默认的，因此需要仔细地依照Python/README目录中的说明来构建。简单地说，首先需要编译该版本的Net-SNMP，然后运行Python目录中的setup.py脚本。我们发现在Red Hat上安装最为容易，因为有RPM资源可以利用。如果决定进行编译，你或许希望首先在Red Hat上试验一下，看其是否能够成功，然后再尝AIX，Solaris，OS X，HPUX等系统平台。如果遇到麻烦，可以先使用虚拟机来运行示例，然后再估计一下之后如何进行编译。

最后，自己编译时还需要注意一点：要确保运行python setup.py进行创建，并且运行python setup.py test进行测试。如果NetSNMP可以与Python协同工作了，你应该会找到适合的编译方法。如果在编译Python时遇到问题，有一个技巧是像下面这样手工执行ldconfig：

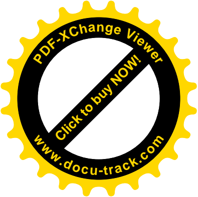
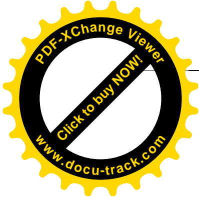
```
ldconfig -v /usr/local/lib/
```

在配置方面，如果碰巧正在想要监测的客户端安装Net-SNMP，应该将Host Resources MIB与Net-SNMP一同编译。一般来说，可以像下面这样操作：

```
./configure -with-mib-modules=host
```

值得注意的是，在运行配置时，它会试图运行一个自动配置脚本。如果不想，就没必要

www.TopSage.com

这样做了。通常创建一个自定义配置文件非常容易。基于Red Hat系统的配置文件通常保存在*/etc/snmp/snmpd.conf*中，可以像下面这样非常简单地完成配置：

```
syslocation "O'Reilly"
syscontact bofh@oreilly.com
rocommunity public
```

仅是这个简单的配置文件对于本章的其他部分以及非SNMPv3查询，就已经够用了。SNMPv3配置起来有点麻烦，并且对本章的大部分内容来说，都略微超出于范围，尽管在生产环境的设备控制中我们确实要使用它。强烈建议大家使用SNMPv3，因为v2版和v1版需要明确的转发。这种情况下，无法使用SNMPv2或v1查询整个互联网，因为会遇到流量拦截。

IPython与Net-SNMP

如果之前没有做过任何SNMP开发，刚一接触或许会给你留下不好的印象。坦诚地讲，确实是这样。使用SNMP还是比较麻烦的，因为它涉及到非常复杂的协议，需要阅读大量的RFC，而且还会有很高的出错率。一种消除开发起始阶段会遇到的这些令人头痛的问题的方法是使用IPython来写SNMP代码，这样可以方便地使用API。

例7-1是运行在本地主机上的非常简单的代码：

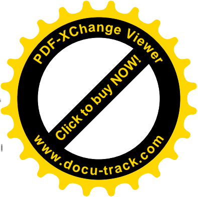
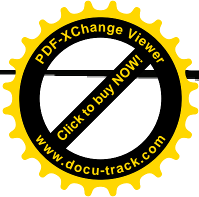
例7-1：使用IPython和绑定Python的Net-SNMP

```
In [1]: import netsnmp

In [2]: oid = netsnmp.Varbind('sysDescr')


In [3]: result = netsnmp.snmpwalk(oid,
...:                            Version = 2,
...:                            DestHost="localhost",
...:                            Community="public")

Out[4]: ('Linux localhost 2.6.18-8.1.14.el5 #1 SMP Thu Aug 27 12:51:54 EDT 2008 i686',)
```

在学习使用一个新库时，使用tab的自动完成功能非常有帮助。在这个示例中，我们完全利用了IPython的tab自动完成功能，并且创建了一个基本的SNMPv2查询。作为一般说明，正如之前所说，sysDescr是一个非常重要的OID查询，可以在主机上执行一些基本的识别。从这个示例的输出部分可以看到，与uname -a相比，虽然不相同但非常相似。

正如在本章后面会看到的，解析来自sysDescr查询的响应是在早期发现数据中心的一个重要手段。不幸的是，正如SNMP的许多其他方面，这一过程不是精确的。一些设备可能不会返回任何响应，部分可能返回一些有用的但是不确切的类似“光交换机”这样的

www.TopSage.com

信息，另外一些会返回一个供应商识别字符串。我们没有时间对如何解决该问题进行深入的讲解，会专门有人处理这些响应差异问题。

正像你在IPython这章中所学到的，当使用IPython在文件中写一个类或函数时，通过键入下面的内容可以在IPython内部切换到Vim：

```
ed some_filename.py
```

然后，当退出Vim时，可以在你的命名空间中取得该模块的属性，并且通过键入who来进行查看。这一技巧对于使用Net-SNMP非常有帮助，因为代码复用本质上适合该问题。让我们继续，在文件*snmp.py*中编写如下代码：

```
ed snmp.py
```

例7-2显示了一个简单的模块，它允许我们在与Net-SNMP创建连接时，提取模板代码。

例7-2：基本Net-SNMP会话模块

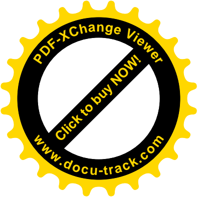
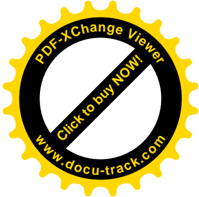
```
#!/usr/bin/env python
import netsnmp

class Snmp(object):
    """A basic SNMP session"""
    def __init__(self,
                oid = "sysDescr",
                Version = 2,
                DestHost = "localhost",
                Community = "public"):
        self.oid = oid
        self.version = Version
        self.destHost = DestHost
        self.community = Community

def query(self):
    """Creates SNMP query session"""
    try:
        result = netsnmp.snmpwalk(self.oid,
                                Version = self.version,
                                DestHost = self.destHost,
                                Community = self.community)
    except Exception, err:
        print err
        result = None
    return result
```

在IPython中保存这个文件，并键入who时，会看到如下所示内容：

```
In [2]: who
Snmp netsnmp
```

现在有了一个面向对象的SNMP接口，可以使用它来查询本地主机：

www.TopSage.com

```
In [3]: s = snmp()

In [4]: s.query()
Out[4]: ('Linux localhost 2.6.18-8.1.14.el5 #1 SMP Thu Sep 27 18:58:54 EDT 2007 i686',)

In [5]: result = s.query()

In [6]: len(result)
Out[6]: 1
```

可以看到，使用模块可以很方便地得到结果，但是这里基本上只运行了一个硬编码脚本。让我们修改OID对象的值来遍历整个系统子树：

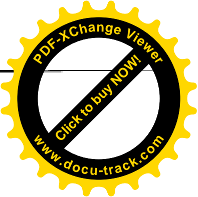
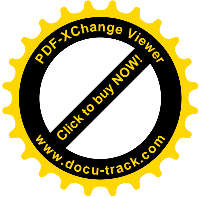
```
In [7]: s.oid
Out[7]: 'sysDescr'

In [8]: s.oid = ".1.3.6.1.2.1.1"

In [9]: result = s.query()

In [10]: print result
('Linux localhost 2.6.18-8.1.14.el5 #1 SMP Thu Sep 27 18:58:54 EDT 2007 i686',
'.1.3.6.1.4.1.8072.3.2.10', '121219', 'me@localhost.com', 'localhost', '"My Local Machine"',
'0', '.1.3.6.1.6.3.10.3.1.1', '.1.3.6.1.6.3.11.3.1.1', '.1.3.6.1.6.3.15.2.1.1',
 '.1.3.6.1.6.3.1',
'.1.3.6.1.2.1.49', '.1.3.6.1.2.1.4', '.1.3.6.1.2.1.50', '.1.3.6.1.6.3.16.2.2.1',
 'The SNMP Management Architecture MIB.',
'The MIB for Message Processing and Dispatching.', 'The management information definitions
 for the SNMP User-based Security Model.',
'The MIB module for SNMPv2 entities', 'The MIB module for managing TCP implementations',
'The MIB module for managing IP and ICMP implementations', 'The MIB module for
 managing UDP [snip]',
'View-based Access Control Model for SNMP.', '0', '0', '0', '0', '0', '0', '0', '0')
```

交互式编程风格使得处理SNMP成了令人感到愉快的事情。在这一点上，如果你还不肯定，可以查看各种各样的其他OID查询，甚至遍历所有的MIB树。遍历MIB树会花去一些时间，因为这需要对大多数OID进行查询。在产品环境中这不是最好的方法，因为它会消耗客户端的资源。

注意： 记住，MIB-II是一个写满了OID的文件，在支持SNMP的大多数系统中都包含该文件。其他特定提供商定义的MIB是附加的文件，代理可以引用并给予响应。如果希望做更进一步的处理，需要查看特定提供商定义的文档来决定在什么样的MIB中查询什么OID。

接下来，使用IPython的一个特定的功能，该功能能够让我们将作业发送到后台：

```
In [11]: bg s.query()
Starting job # 0 in a separate thread.

In [12]: jobs[0].status
Out[12]: 'Completed'
```

```
In [16]: jobs[0].result
Out[16]:
('Linux localhost 2.6.18-8.1.14.el5 #1 SMP Thu Sep 27 18:58:54 EDT 2007 i686',
'.1.3.6.1.4.1.8072.3.2.10', '121219', 'me@localhost.com', 'localhost',
  '"My Local Machine"',
'0', '.1.3.6.1.6.3.10.3.1.1', '.1.3.6.1.6.3.11.3.1.1', '.1.3.6.1.6.3.15.2.1.1',
  '.1.3.6.1.6.3.1',
'.1.3.6.1.2.1.49', '.1.3.6.1.2.1.4', '.1.3.6.1.2.1.50', '.1.3.6.1.6.3.16.2.2.1',
'The SNMP Management Architecture MIB.', 'The MIB for Message Processing and
  Dispatching.',
'The management information definitions for the SNMP User-based Security Model.',
'The MIB module for SNMPv2 entities', 'The MIB module for managing TCP implementations',
'The MIB module for managing IP and ICMP implementations', 'The MIB module for
  managing UDP implementations',
'View-based Access Control Model for SNMP.', '0', '0', '0', '0', '0', '0', '0', '0')
```

看到上面的结果，在你没高兴过头之前，让我告诉你，在IPython中后台线程是非常有吸引力的，它仅与支持异步线程的库一起使用。针对Net-SNMP的Python绑定是同步的。简而言之，你不能像使用C语言写等待响应的代码块那样写多线程代码。幸运的是，在进程与并发一章可以看到，使用processing模块来创建处理并发SNMP请求的进程非常容易。在接下来的一节，我们写了一个简单的工具来自动发现数据中心，后面将具体进行介绍。

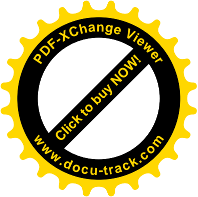
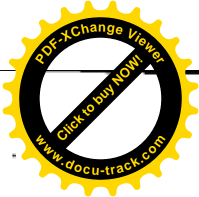
查找数据中心

SNMP的一个更有意义的用途是查找数据中心。简要地讲，对数据中心的查找会收集网络上的设备列表及这些设备的信息。更高级的查找可以用来创建所收集的数据之间的关联，例如，连接在Cisco交换机上的活动服务器的准确Mac地址，或是Brocade光纤交换机的存储层次。

在这一节中，我们将创建一个基本的查找脚本，用来收集可用的IP地址、Mac地址、基本的SNMP信息，并对它们进行记录。这是在你的设备上实现数据中心查找的一个非常有用的基础。我们将通过提取在其他章中介绍的信息，来进一步说明。

我们会接触到一些各不相同的查找算法，这里将介绍其中最简单的一个。可以用一句话描述该算法：发出一些ICMP ping，对于每一个响应的设备，发出一个基本的SNMP查询，解析输出，然后根据结果做进一步查找。另一个会被提及的算法，它类似散弹枪一样发送一系列SNMP查询，然后运行另一个进程来收集响应。但是正如之前所说的，我们将集中介绍第一个算法。参见例7-3。

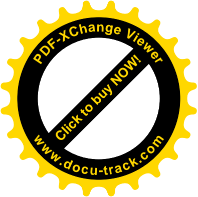
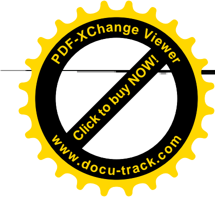
www.TopSage.com

注意：下面的代码中有一点需要注意：由于Net-SNMP库是同步的，我们将创建一个对subprocess.call的调用。对于刚使用过的subprocess.Popen的ping选项，为了保持代码的一致性，我们将为SNMP和ping使用相同的模式。

例7-3：基本数据中心发现

```
#!/usr/bin/env python
from processing import Process, Queue, Pool
import time
import subprocess
import sys
from snmp import Snmp

q = Queue()
oq = Queue()
#ips = IP("10.0.1.0/24")
ips = ["192.19.101.250", "192.19.101.251", "192.19.101.252","192.19.101.253",
"192.168.1.1"]
num_workers = 10

class HostRecord(object):
    """Record for Hosts"""
    def __init__(self, ip=None, mac=None, snmp_response=None):
        self.ip = ip
        self.mac = mac
        self.snmp_response = snmp_response
    def __repr__(self):
        return "[Host Record('%s','%s','%s')]" % (self.ip,
                                                  self.mac,
                                                  self.snmp_response)

def f(i,q,oq):
    while True:
        time.sleep(.1)
        if q.empty():
            sys.exit()
            print "Process Number: %s Exit" % i
        ip = q.get()
        print "Process Number: %s" % i
        ret = subprocess.call("ping -c 1 %s" % ip,
                        shell=True,
                        stdout=open('/dev/null', 'w'),
                        stderr=subprocess.STDOUT)
        if ret == 0:
            print "%s: is alive" % ip
            oq.put(ip)
        else:
            print "Process Number: %s didn't find a response for %s " % (i, ip)
            pass

def snmp_query(i,out):
    while True:
        time.sleep(.1)
        if out.empty():
            sys.exit()
```

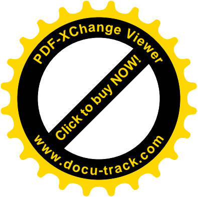
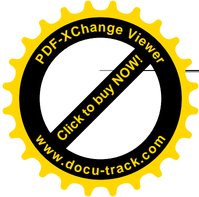
```
            print "Process Number: %s" % i
        ipaddr = out.get()
        s = Snmp()
        h = HostRecord()
        h.ip = ipaddr
        h.snmp_response = s.query()
        print h
        return h
try:
    q.putmany(ips)

finally:
    for i in range(num_workers):
        p = Process(target=f, args=[i,q,oq])
        p.start()
    for i in range(num_workers):
        pp = Process(target=snmp_query, args=[i,oq])
        pp.start()

print "main process joins on queue"
p.join()
#while not oq.empty():
#    print "Validated", oq.get()


print "Main Program finished"
```

运行这个脚本，会看到如下所示的输出结果：

```
[root@giftcsllc02][H:4849][J:0]> python discover.py
Process Number: 0
192.19.101.250: is alive
Process Number: 1
192.19.101.251: is alive
Process Number: 2
Process Number: 3
Process Number: 4
main process joins on queue
192.19.101.252: is alive
192.19.101.253: is alive
Main Program finished
[Host Record('192.19.101.250','None','('Linux linux.host 2.6.18-8.1.15.el5
  #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686',)')]
[Host Record('192.19.101.252','None','('Linux linux.host 2.6.18-8.1.15.el5
  #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686',)')]
[Host Record('192.19.101.253','None','('Linux linux.host 2.6.18-8.1.15.el5
  #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686',)')]
[Host Record('192.19.101.251','None','('Linux linux.host 2.6.18-8.1.15.el5
  #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686',)')]
Process Number: 4 didn't find a response for 192.168.1.1
```

查看一下代码的输出，我们看到这个有趣的查找数据中心的算法的起始部分有一些小问题需要进一步修复，例如还需要添加Mac地址到Host Record对象，从而使用代码更加趋于面向对象。但是这会完全变成另一本书，我们在接下来一节对这一点进一步说明。

www.TopSage.com

使用Net-SNMP获取多个值

通过SNMP获得单一值的实际意义不大，尽管这对于测试响应或执行基于指定值的操作非常有用，例如查询一台主机的OS类型。为了做一些更有意义的事情，需要取回多个值并使用它们来完成一些更重要的事情。

一个非常常见的任务是对数据中心或分中心做一个详细的目录清单，计算所有主机的一些参数集合。下面是一个假想的情况：你正准备对一个重要软件进行升级，并被告之所有的系统都需要至少1GB的内存。你恍惚记得绝大多数主机有至少1GB的内存，但是还是有几千台自己负责的主机没有1GB内存。

很明显，你需要做一个艰难的决定。下面介绍一些可能的选择：

选择1

物理上启动每一台主机然后运行命令或打开主机箱来检查安装了多少内存。很明显，这不是一个很有吸引力的办法。

选择2

通过Shell登录到每一台主机，然后运行命令来查看具体有多少内存。该方法也存在一些问题，但是至少理论上通过使用ssh密钥可以被写成脚本。一个明显的问题是需要写一个跨平台的脚本，因为每一种操作系统都会有些许差别。另一个问题是该方法依赖于对活动主机位置的了解。

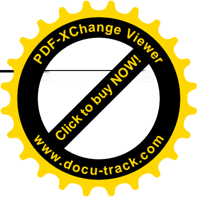
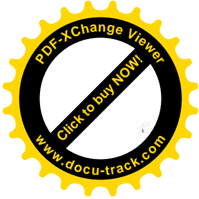
选择3

写一小段脚本，通过SNMP遍历并查询网络中每一设备具有的内存量。

使用选择3，通过SNMP，可以非常容易地产生目录清单报告，报告显示内存量不到1G的主机列表。查询所需要的OID的准确名称是“hrMemorySize”。SNMP在并发处理中优势明显，但是最好不要进行优化，除非绝对需要。下面复用前面示例的代码来进行一次快速测试，通过SNMP获得内存值：

```
In [1]: run snmpinput

In [2]: who
netsnmp Snmp

In [3]: s = Snmp()

In [4]: s.DestHost = "10.0.1.2"

In [5]: s.Community = "public"

In [6]: s.oid = "hrMemorySize"

In [7]: result = int(s.query()[0])
   hrMemorySize = None ( None )
```

www.TopSage.com

```
In [27]: print result
2026124
```

可以看到，这是一个非常简单易懂的脚本。结果值在第6行作为一个元组返回，因此我们提取索引0的值并将其转换为整数。结果值是一个以KB为单位的整数。需要记住的是，不同的机器在计算RAM时会用不同的方法。因此，最好在计算时使用模糊参数，不要硬编码成一个绝对值，因为你可能会得到与所期望的值不同的结果。例如希望查询一个比1G内存稍低的范围值，比如990MB。

在接下来将要讲述的事例中，需要对大约2GB内存进行统计。假如你现在被告之，由于一个新的应用程序需至少2GB内存才能安装，老板希望你能对数据中心中具有2GB内存的主机进行识别。

基于这些信息，我们现在自动对内存多少进行判断。最为有效的做法应该是这样：查询每一台主机，推断出它是否具有2GB的内存，然后将这一信息写入CSV文件，这样就可以很容易被加载到Excel和Open Office中。

接下来可以写一个命令行工具，该工具以子网范围和一个可选的OID关键字（默认是hrmemorySize）作为输入。我们希望可以在子网中迭代一个地址范围。

通常，作为系统管理员，在写代码时会面对一些艰难的选择。应该花几小时或一整天写一个非常长的以后可以复用的脚本，而仅是因为使用的是面向对象技术？还是应该快速直接地完成代码？我们认为在大多数情况下，两者兼得是没有问题的。如果使用IPython，你可以对自己写的脚本进行记录，然后将它们转换为更精炼的脚本。通常来讲，写可复用的代码是一个好主意，因为它会像滚雪球一样，很快具有越来越大的惯性。

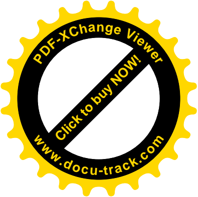
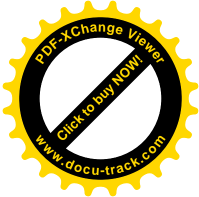
如果之前你还是没有意识到SNMP的强大之外，那么现在恐怕已经理解了。让我们继续写脚本...

查找内存

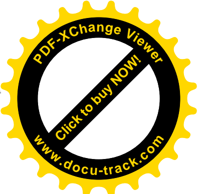
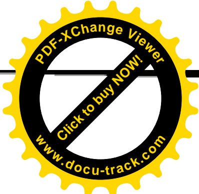
在接下来的示例中，我们写了一个命令行工具，通过SNMP来计算主机中已安装的内存容量：

```
#!/usr/bin/env python
#A command line tool that will grab total memory in a machine

import netsnmp
import optparse
from IPy import IP
```

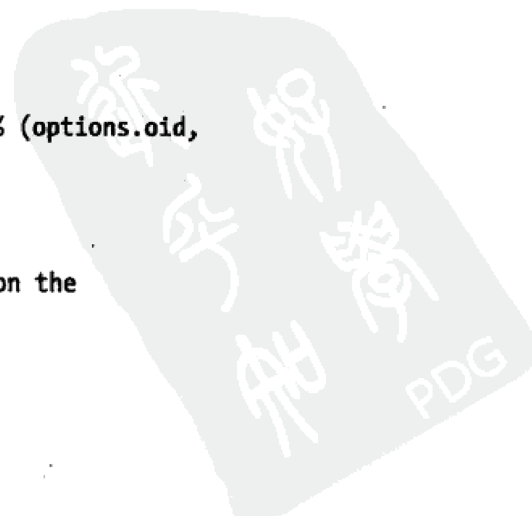

www.TopSage.com

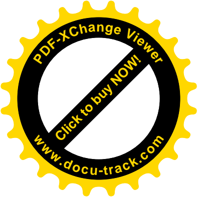
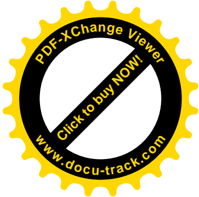



```
class SnmpSession(object):
    """A Basic SNMP Session"""
    def __init__(self,
                oid="hrMemorySize",
                Version=2,
                DestHost="localhost",
                Community="public"):

        self.oid = oid
        self.Version = Version
        self.DestHost = DestHost
        self.Community = Community


    def query(self):
        """Creates SNMP query session"""
        try:
            result = netsnmp.snmpwalk(self.oid,
                                    Version = self.Version,
                                    DestHost = self.DestHost,
                                    Community = self.Community)
        except:
            #Note this is toy code, but let's us know what exception is raised
            import sys
            print sys.exc_info()
                result = None

            return result

    class SnmpController(object):
        """Uses optparse to Control SnmpSession"""

        def run(self):
        results = {}    #A place to hold and collect snmp results
        p = optparse.OptionParser(description="A tool that determines
                                memory installed",
                               prog="memorator",
                               version="memorator 0.1.0a",
                    usage="%prog [subnet range] [options]")
        p.add_option('--community', '-c',help='community string',
          default='public')
        p.add_option('--oid', '-o', help='object identifier',
          default='hrMemorySize')
        p.add_option('--verbose', '-v', action='store_true',
          help='increase verbosity')
        p.add_option('--quiet', '-q', action='store_true',help='
                     suppresses most messages')
        p.add_option('--threshold', '-t', action='store', type="int",
                    help='a number to filter queries with')

        options, arguments = p.parse_args()
                if arguments:
            for arg in arguments:
                try:
                    ips = IP(arg) #Note need to convert instance to str
                except:
                    if not options.quiet:
```

```
                print 'Ignoring %s, not a valid IP address' % arg
            continue

        for i in ips:
            ipAddr = str(i)
            if not options.quiet:
                print 'Running snmp query for: ', ipAddr

            session = SnmpSession(options.oid,
                                  DestHost = ipAddr,
                                  Community = options.community)


            if options.oid == "hrMemorySize":
                try:
                    memory = int(session.query()[0])/1024
                except:
                    memory = None
                output = memory

            else:
                #Non-memory related SNMP query results
                output = session.query()
                if not options.quiet:
                    print "%s returns %s" % (ipAddr,output)

            #Put everything into an IP/result dictionary
            #But only if it is a valid response
            if output != None:
                if options.threshold: #ensures a specific threshold
                    if output < options.threshold:
                        results[ipAddr] = output
                        #allow printing to standard out
                        if not options.quiet:
                            print "%s returns %s" % (ipAddr,output)

                else:
                    results[ipAddr] = output
                    if not options.quiet:
                        print output

        print "Results from SNMP Query %s for %s:\n" % (options.oid,
            arguments), results


    else:
        p.print_help() #note if nothing is specified on the
      command line, help is printed

def _main():
    """
    Runs memorator.
    """
    start = SnmpController()
    start.run()
```



```
if __name__ =='__main__':
    try:
        import IPy
    except:
        print "Please install the IPy module to use this tool"
_main()
```

好了，让我们看一下这段代码，并观察代码是怎样执行的。这里使用了前一个示例中所有的类，并将其放到新模块中。接下来创建了一个控制类，该类通过optparse模块来处理选项操作。IPy模块（这是我们一遍一遍地引用的模块）将自动处理IP地址参数。现在可以放入一些IP地址或是一个子网范围进行处理，我们的模块会寻找SNMP查询，并返回一个由IP地址和SNMP值组成的字典。

这里使用了一个技巧，在最后返回结果不为空时创建一些逻辑，并且额外侦听一个阈值。这表示将它设置为仅当数值小于指定阈值时才返回。通过使用阈值，可以返回有意义的结果，并且兼容由于不同主机对内存计算方式的差异而引起的不同。

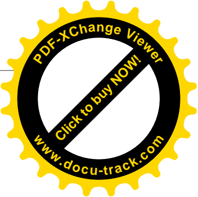
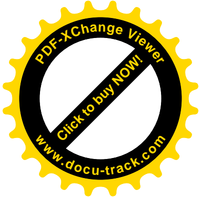
下面看一下示例中这个工具的输出结果：

```
[ngift@ng-lep-lap][H:6518][J:0]> ./memory_tool_netsnmp.py 10.0.1.2 10.0.1.20
Running snmp query for: 10.0.1.2
   hrMemorySize = None ( None )
1978
Running snmp query for: 10.0.1.20
   hrMemorySize = None ( None )
372
Results from SNMP Query hrMemorySize for ['10.0.1.2', '10.0.1.20']:
{'10.0.1.2': 1978, '10.0.1.20': 372}
```

正如你所看到的，结果来自子网10.0.1.0/24中的主机。让我们使用阈值标志来模拟寻找小于2GB内存的主机。正如之前所提及的，不同主机在计算RAM时存在差异，因此为了保险起见，我们使用数据值1800，这大约相当于1800MB的内存。如果一台主机不具有至少1800MB或者说大约2GB的内存，它的信息会出现在报告中。下面是查询的输出结果：

```
[ngift@ng-lep-lap][H:6519][J:0]>
    ./memory_tool_netsnmp.py --threshold 1800 10.0.1.2 10.0.1.20
Running snmp query for: 10.0.1.2
   hrMemorySize = None ( None )
Running snmp query for: 10.0.1.20
   hrMemorySize = None ( None )
10.0.1.20 returns 372
Results from SNMP Query hrMemorySize for ['10.0.1.2', '10.0.1.20']:
{'10.0.1.20': 372}
```

尽管现在的脚本可以胜任这一工作，但是我们还是可以做一些事情来进一步优化这个工具。如果需要查询数千台机器，那么这个工具会花去一天多的时间来进行处理。这或许

会也能满足需要，但是如果希望很快就能看到结果，就需要添加并发处理并使用第三方库来fork每一个查询。我们可以做的另一项改进是从字典中自动产生CSV报告。在将这些任务自动化之前，给你看一个你可能没有注意的另一个优点。代码编写的方式允许查询任何OID，而不是专门用来进行内存计算的。因此我们现在不仅拥有了计算内存的工具，还有了一个通用工具，可以实现SNMP查询，而这一切实现得又是如此简便。

下面看一个实现我们意图的示例：

```
[ngift@ng-lep-lap][H:6522][J:0]> ./memory_tool_netsnmp.py -o sysDescr 10.0.1.2
10.0.1.20
Running snmp query for: 10.0.1.2
   sysDescr = None ( None )
10.0.1.2 returns ('Linux cent 2.6.18-8.1.14.el5 #1 SMP
  Thu Sep 27 19:05:32 EDT 2007 x86_64',)
('Linux cent 2.6.18-8.1.14.el5 #1 SMP Thu Sep 27 19:05:32 EDT 2007 x86_64',)
Running snmp query for: 10.0.1.20
   sysDescr = None ( None )
10.0.1.20 returns ('Linux localhost.localdomain 2.6.18-8.1.14.el5 #1 SMP
 Thu Sep 27 19:05:32 EDT 2007 x86_64',)
('Linux localhost.localdomain 2.6.18-8.1.14.el5 #1 SMP
  Thu Sep 27 19:05:32 EDT 2007 x86_64',)
Results from SNMP Query sysDescr for ['10.0.1.2', '10.0.1.20']:
{'10.0.1.2': ('Linux cent 2.6.18-8.1.14.el5 #1 SMP
Thu Sep 27 19:05:32 EDT 2007 x86_64',), '10.0.1.20':
('Linux localhost.localdomain 2.6.18-8.1.14.el5 #1 SMP
  Thu Sep 27 19:05:32 EDT 2007 x86_64',)}
```

当写一些一次性的工具时，记住这一观点非常实用。为什么不再多花30分钟来使代码更为通用呢？你或许会发现自己又有了一个可以一遍又一遍重复使用的工具与将来会省下的大量时间相比，30分钟只相当于桶中的一滴水。

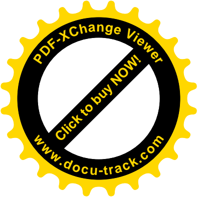
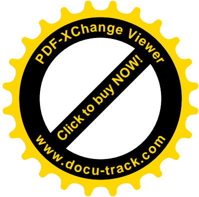
创建混合的SNMP工具

至此我们已经分别演示了一些工具的示例，需要注意的是，这些技术可以被合并起来创建一些更高级的工具。让我们从创建一个完整的一次性工具开始，然后再在更大的脚本中使用这些技术。

有一个被称为snmpstatus的非常有用的工具，可以获得一些不同的snmp查询，并且可以将其合并到“状态”中：

```
import subprocess

class Snmpdf(object):
    """A snmpstatus command-line tool"""
    def __init__(self,
                 Version="-v2c",
                 DestHost="localhost",
```


www.TopSage.com

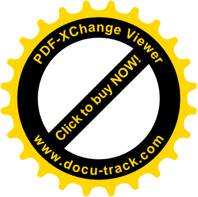
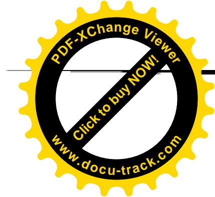
```
                   Community="public",
                   verbose=True):

        self.Version = Version
        self.DestHost = DestHost
        self.Community = Community
        self.verbose = verbose

    def query(self):
        """Creates snmpstatus query session"""
        Version = self.Version
        DestHost = self.DestHost
        Community = self.Community
        verbose = self.verbose

        try:
            snmpstatus = "snmpstatus %s -c %s %s" % (Version, Community, DestHost)
            if verbose:
                print "Running: %s" % snmpstatus
            p = subprocess.Popen(snmpstatus,
                                 shell=True,
                                 stdout=subprocess.PIPE)
            out = p.stdout.read()
            return out

        except:
            import sys
            print >> sys.stderr, "error running %s" % snmpstatus

def _main():
    snmpstatus = Snmpdf()
    result = snmpstatus.query()
    print result
if __name__ == "__main__":
    _main()
```

希望你能注意到这一事实：除了名称不同之外，该脚本与snmpdf命令之间的差异非常小。创建另一级别的抽象并复用通用组件，这是一个非常棒的示例。如果创建一个模块来处理所有的模板代码，我们的新脚本会仅有几行长。记住这一点，稍后会再次提到。

另外一个与SNMP相关的工具是ARP。ARP使用ARP协议。如果物理上是在同一个网络上，那么通过使用ARP协议，可以基于IP地址获得设备的Mac地址。稍后让我们也编写这样的一个工具。

ARP非常容易被整合到脚本中，因此最好通过交互地使用Ipython来演示示例。现在继续，启动IPython，然后尝试一下：

```
import re
import subprocess

#some variables
ARP = "arp"
```

```
IP = "10.0.1.1"
CMD = "%s %s " % (ARP, IP)
macPattern = re.compile(":")

def getMac():
    p = subprocess.Popen(CMD, shell=True, stdout=subprocess.PIPE)
    out = p.stdout.read()
    results = out.split()
    for chunk in results:
        if re.search(macPattern, chunk):
            return chunk

if __name__ == "__main__":
    macAddr = getMac()
    print macAddr
```

这个代码段还无法构成一个可复用的工具，但是可以简单地采纳这一思想，并将其作为数据中心查找库的一部分。

Net-SNMP扩展

正如之前讨论的，Net-SNMP在大多数*nix主机中是作为代理被安装的。虽然代理有一个可以返回的默认的信息集，但是对主机上的代理进行扩展也是可行的。一个合理且直接了当的方式是，写一个代理来收集需要的信息，然后通过SNMP协议返回结果集。

*EXAMPLE.conf*文件是获取扩展Net-SNMP相关信息的最好的地方，它被包括在Net-SNMP中。可以对snmpd.conf使用man命令以获得API文档的详细信息。如果希望进一步学习如何扩展代理，前面介绍的两种方式都是获得相关信息的非常不错的途径。

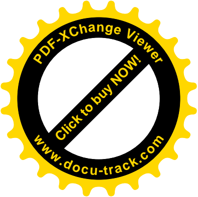
对于一名Python程序员，扩展Net-SNMP是使用SNMP最令人激动的一个方面，因为它允许开发者通过编写代码来监测想查看的内容，并且可以额外地让代理对你指定给它的条件进行内部响应。

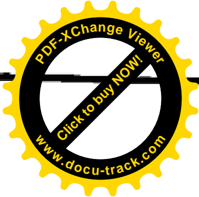
Net-SNMP提供了一些方法来扩展它的代理，但是我们将从编写一个Hello World程序开始，该程序由snmp来进行查询。第一步是创建一个非常简单的*snmpd.conf*文件，该文件在Python中执行我们的Hello world程序。例7-4演示了在Red Hat主机上的运行过程。

例7-4：Hello World的SNMP配置文件

```
syslocation "O'Reilly"
syscontact bofh@oreilly.com
rocommunity public
exec helloworld /usr/bin/python -c "print 'hello world from Python'"
```

接下来需要告诉snmpd重新读取配置文件。我们有三种不同的处理方法。在Red Hat上可以这样使用：





```
service snmpd reload
```

或者可以这样做：

```
ps -ef | grep snmpd
root      12345 1 0 Apr14 ?
00:00:30 /usr/sbin/snmpd -Lsd -Lf /dev/null -p /var/run/snmpd.pid -a
```

然后，将其发送出去：

```
kill -HUP 12345
```

最后，snmpset命令可以给UCD-SNMPMIB::versionUpdateConfig.0指定一个整数（1），告诉snmpd重读配置文件。

现在已经修改了*snmpd.conf*文件，并且告诉snmpd重读配置文件。我们可以继续通过使用snmpwalk命令或是绑定到IPython中的Net-SNMP来查询我们的主机。下面是使用snmpwalk命令的执行结果：

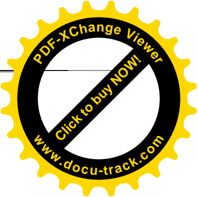
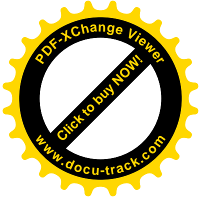
```
[root@giftcsllc02][H:4904][J:0]> snmpwalk -v 1 -c public localhost .1.3.6.1.4.1.2021.8
UCD-SNMP-MIB::extIndex.1 = INTEGER: 1
UCD-SNMP-MIB::extNames.1 = STRING: helloworld
UCD-SNMP-MIB::extCommand.1 = STRING: /usr/bin/python
  -c "print 'hello world from Python'"
UCD-SNMP-MIB::extResult.1 = INTEGER: 0
UCD-SNMP-MIB::extOutput.1 = STRING: hello world from Python
UCD-SNMP-MIB::extErrFix.1 = INTEGER: noError(0)
UCD-SNMP-MIB::extErrFixCmd.1 = STRING:
```

对本查询需要做进一步解释，因为一些观察力比较强的读者可能会奇怪我们从哪里获得的1.3.6.1.4.1.2021.8。这个OID是ucdavis.extTable。在创建一个snmpd.conf扩展时，它会将该值指定给OID。如果你希望查询一个由自己创建的自定义的OID，处理起来会稍微有些复杂。实现这一目标的常规做法是使用iana.org填写一个请求，然后获得一个企业码。可以使用这一代码来创建自定义的对代理的查询。这么做的主要原因是保持一个统一的名字空间，避免与其他将来会遇到的提供商出现编码冲突。

从单行获得输出不是Python的长处，而且这显得有些笨拙。下面是一个脚本示例，它会解析Apache日志中点击Firefox的总数，然后返回自定义的企业编码。这次让我们向后看一下查询时会是什么样子：

```
snmpwalk -v 2c -c public localhost .1.3.6.1.4.1.2021.28664.100
UCD-SNMP-MIB::ucdavis.28664.100.1.1 = INTEGER: 1
UCD-SNMP-MIB::ucdavis.28664.100.2.1 = STRING: "FirefoxHits"
UCD-SNMP-MIB::ucdavis.28664.100.3.1 = STRING:
 "/usr/bin/python /opt/local/snmp_scripts/agent_ext_logs.py"
UCD-SNMP-MIB::ucdavis.28664.100.100.1 = INTEGER: 0
```

www.TopSage.com

```
UCD-SNMP-MIB::ucdavis.28664.100.101.1 = STRING:
  "Total number of Firefox Browser Hits: 15702"
UCD-SNMP-MIB::ucdavis.28664.100.102.1 = INTEGER: 0
UCD-SNMP-MIB::ucdavis.28664.100.103.1 = ""
```

如果查找数值100.101.1，你会看到脚本的输出。脚本使用gnerator管道来解析Apache日志并在日志中查找所有的Firefox点击数，然后，对其进行总计并通过SNMP返回结果。例7-5是查询这个OID时执行的脚本。

例7-5：查询Apache日志文件中firefox的点击数

```
import re

"""Returns Hit Count for Firefox"""

def grep(lines,pattern="Firefox"):
    pat = re.compile(pattern)
    for line in lines:
        if pat.search(line): yield line

def increment(lines):
    num = 0
    for line in lines:
        num += 1
    return num

wwwlog = open("/home/noahgift/logs/noahgift.com-combined-log")
column = (line.rsplit(None,1)[1] for line in wwwlog)
match = grep(column)
count = increment(match)
print "Total Number of Firefox Hits: %s" % count
```

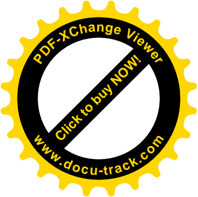
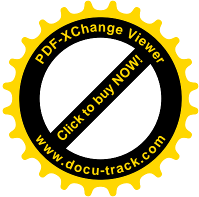
为了使查询操作在第一时间里就能起作用，需要告诉snmpd.conf这个脚本的相关信息。下面是这部分代码的内容：

```
syslocation "O'Reilly"
syscontact bofh@oreilly.com
rocommunity public
exec helloworld /usr/bin/python -c "print 'hello world from Python'"
exec .1.3.6.1.4.1.2021.28664.100 FirefoxHits /usr/bin/python
  /opt/local/snmp_scripts/agent_ext_logs.py
```

关键是最后一行，在这一行中的1.3.6.1.4.1.2021是ucdavis企业编码，28664是我们的企业编码，100是希望使用的预定的值。如果计划扩展SNMP，遵循最好的经验并使用我们的企业编码是非常重要的。主要原因是，如果决定使用一个已经被别人占用的范围值，并通过snmpset进行修改，可以避免引起破坏。

我们希望在即将完成介绍时，能够确立这样的事实：SNMP是这本书最吸引人的主题之一。自定义的Net-SNMP对处理许多事务都非常有帮助，并且如果细心地使用SNMP v3，

www.TopSage.com

你可以通过SNMP协议很容易地做一些令人吃惊的事情。但通常人们会选择ssh或socket。

SNMP设备控制

使用SNMP的最有意义的一件事情是通过SNMP对设备进行控制。很明显，在控制路由器方面，SNMP具有比其他一些工具，如Pyexpect（*http://sourceforge.net/projects/pexpect/*）更大的优势。最主要的原因在于它更简单。

出于简洁的原因，我们仅在示例中介绍SNMP v1，但是如果通过一个不安全的网络与一个设备进行通信，应该使用SNMP v3。在本节，如果你有一个Safari账号或是已经购买了由Kevin Dooley和Ian J. Brown (O'Reilly)编著的《Essential SNMP》和《Cisco IOS Cookbook》，那么最好多参看这些书。书中包括了如何通过SNMP与Cisco设备进行通信以及进行基本配置的相关内容。

通过SNMP重新加载Cisco配置是非常吸引人的方法，看起来像是与设备进行通信及实现其控制的绝佳的选择。在这个示例中，必须从下载IOS文件的路由器上运行 TFTP服务，并且路由器必须被配置成允许通过SNMP进行读写操作。例7-6是Python代码的实现过程：

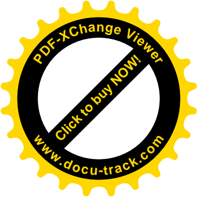
例7-6：上传新的配置

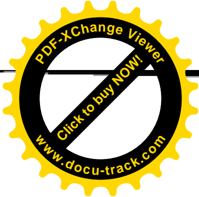
```
import netsnmp

vars = netsnmp.Varbind(netsnmp.VarList(netsnmp.Varbind(".1.2.6.1.4.1.9.2.10.6.0", "1"),
        (netsnmp.Varbind("cisco.example.com.1.3.6.1.4.1.9.2.10.12.172.25.1.1",
                        "iso-config.bin")

result = netsnmp.snmpset(vars,
                        Version = 1,
                        DestHost='cisco.example.com',
                        Community='readWrite')
```

这个示例使用Net-SNMP的VarList来首先给交换机发出删除闪存的指令，然后加载一个新的IOS镜像文件。这或许是为数据中心的每一台交换机实现IOS一次性升级的脚本的基础。书中的所有代码，应该首先在非产品环境中被检测，以免引起破坏。

最后需要指出的一点是，SNMP经常不被认为是一种用来实现设备控制的工具，但它确实是非常棒的通过编程来控制数据中心设备的方法。因为它作为设备控制的统一规范，自1988年就被提出。未来或许会针对SNMP v3版本出现更有意义的应用。

www.TopSage.com





整合Zenoss的企业级SNMP

Zenoss是一个对于企业级SNMP管理系统非常有吸引力的新选择。Zenoss是一个完全开放源代码的工具，由纯Python语言编写。Zenoss是一个新的企业级应用，通过XML-RPC或是ReST API实现了极为强大的功能及可扩展性。要查看与ReST相关的更多信息，可以参阅由Leonard Richardson 和Sam Ruby (O'Reilly)编著的《RESTful Web Services》。最后，如果希望参与开发Zenoss，可以贡献你编写的补丁。

Zenoss API

要获得Zenoss API的最新信息，请访问以下网址：*http://www.zenoss.com/community/docs/howtos/send-events/*。

使用Zendmd

Zenoss不仅是一个强大的SNMP监测与发现系统，它也包括了称为zendmd的高级API。你可以打开一个自定义的shell，然后直接运行Zenoss命令。

```
>>> d = find('build.zenoss.loc')
>>> d.os.interfaces.objectIds()
['eth0', 'eth1', 'lo', 'sit0', 'vmnet1', 'vmnet8']
>>> for d in dmd.Devices.getSubDevices():
>>>     print d.id, d.getManageIp()
```

设备API

你也可以通过XML-RPC API与Zenoss直接通信，添加或删除设备。下面是两个示例。

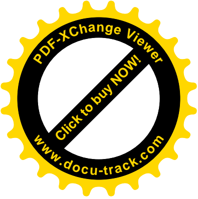
使用ReST：

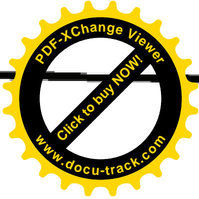
```
[zenos@zenoss $]
wget 'http://admin:zenoss@MYHOST:8080/zport/dmd
/ZenEventManager/manage_addEvent?device=MYDEVICE&component=MYCOMPONENT&summary=¬
  MYSUMMARY&severity=4&eclass=EVENTCLASS&eventClassKey=EVENTCLASSKEY
```

使用XML-RPC：

```
>>> from xmlrpclib import ServerProxy
>>> serv = ServerProxy('http://admin:zenoss@MYHOST:8080/zport/dmd/ZenEventManager')
>>> evt = {'device':'mydevice', 'component':'eth0',
'summary':'eth0 is down','severity':4, 'eventClass':'/Net'}
>>> serv.sendEvent(evt)
```

www.TopSage.com





第8章

操作系统什锦

引言

作为一名系统管理员往往意味着经常会遇到麻烦。一些规则、既定计划，甚至是对操作系统的选择经常会超出你的处置能力。如今，若想成为一名非常高效的系统管理员，需要了解所有的操作系统，从Linux到OS X，再到FreeBSD，都需要掌握。应该说只有时间可以决定哪些操作系统会被长久地使用。像AIX和HP-UX这样的专有操作系统似乎前景暗淡，但是对于许多人来说，仍然需要了解。

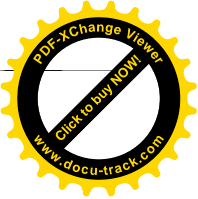
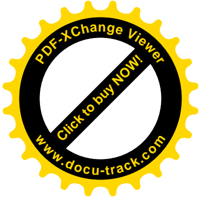
幸运的是，Python再次站出来伸出了援手（希望你注意到这个趋势）。Python提供了一个成熟的标准库，该库包括了一名在多操作系统环境下工作的系统管理员所需要的任何东西。在Python的强大标准库中有一个模块，可以处理从对目录打包，到对文件或目录进行比较，再到对配置文件进行解析等系统管理员想要做的任何事情。Python的成熟完备，以及较好的可读性，使其成为系统管理中的重量级角色。

许多复杂的系统管理工具，例如动画制作、数据中心，都正在从Perl转换到Python上来，因为Python提供了更多可读性强的优秀代码。Ruby也是一个很有吸引力的语言，其自身具有许多Python的优秀特点。但是作为系统管理语言，在对标准库及语言的成熟度进行比较时，Ruby与Python相比尚存在不足。

由于本章是对许多不同操作系统的混合，这里没有时间对其中任何一个进行深入的介绍，但是我们会演示Python作为一种通用的跨平台脚本语言是如何工作的，以及是如何成为适合各种操作系统的强大工具的。最后，介绍一个全新的操作系统，它以数据中心的形式出现，一些人将这一新平台称为云计算。我们还将讨论由Amazon和Google所提供的相关内容。

是否闻到了厨房里传来的香气？这难道不是一份操作系统什锦吗？

www.TopSage.com

Python中跨平台的UNIX编辑

在*nix操作系统之间存在一些重要的差异，但是与差异相比则有更多的共同点。一种弥补不同版本的*nix之间差异的方法是编写跨平台工具和库，以此在不同操作系统之间架设桥梁。最简单也是最有效的方法之一是写一个条件语句对操作系统、平台以及代码版本进行检查。

Python的“连电池都包括在内”的思想具有极深的影响，你想到的任何问题都可以在Python中找到相应的处理工具。对于如何判断自己代码运行在什么平台这一问题，有platform（平台）模块可以利用。让我们看看使用platform模块的要点。

使用platform模块的简单方式是创建一个工具，输出与系统相关的所有可用的信息。参见例8-1。

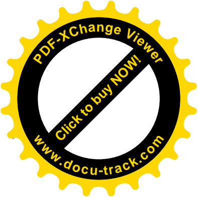
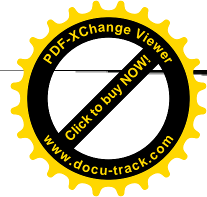
例8-1：使用platform模块输出系统报告

```
#!/usr/bin/env python
import platform

profile = [
platform.architecture(),
platform.dist(),
platform.libc_ver(),
platform.mac_ver(),
platform.machine(),
platform.node(),
platform.platform(),
platform.processor(),
platform.python_build(),
platform.python_compiler(),
platform.python_version(),
platform.system(),
platform.uname(),
platform.version(),
]

for item in profile:
  print item
```

这是脚本在OS X Leopard 10.5.2上的输出结果：

```
[ngift@Macintosh-6][H:10879][J:0]% python cross_platform.py
('32bit', '')
('', '', '')
('', '')
('10.5.2', ('', '', ''), 'i386')
i386
Macintosh-6.local
Darwin-9.2.0-i386-32bit
i386
('r251:54863', 'Jan 17 2008 19:35:17')
```


www.TopSage.com

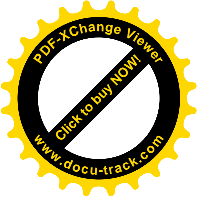
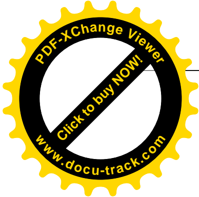



```
GCC 4.0.1 (Apple Inc. build 5465)
2.5.1
Darwin
('Darwin', 'Macintosh-6.local', '9.2.0', 'Darwin Kernel Version 9.2.0:
Tue Feb 5 16:13:22 PST 2008; root:xnu-1228.3.13~1/RELEASE_I386', 'i386', 'i386')
Darwin Kernel Version 9.2.0: Tue Feb 5 16:13:22 PST 2008;
root:xnu-1228.3.13~1/RELEASE_I386
```

这让我们知道了可以收集的信息类型。接下来写了一个跨平台代码来创建一个fingerprint模块，该模块会对平台及版本信息进行采集。这个示例对以下操作系统的相关信息进行了采集：Mac OS X、Ubuntu、 Red Hat/Cent OS、FreeBSD以及Sun OS。参见例8-2。

例8-2：采集操作系统类型信息

```
#!/usr/bin/env python
import platform

"""
Fingerprints the following Operating Systems:

* Mac OS X
* Ubuntu
* Red Hat/Cent OS
* FreeBSD
* SunOS

"""
class OpSysType(object):
  """Determins OS Type using platform module"""

  def __getattr__(self, attr):
    if attr == "osx":
      return "osx"
    elif attr == "rhel":
      return "redhat"
    elif attr == "ubu":
      return "ubuntu"
    elif attr == "fbsd":
      return "FreeBSD"
    elif attr == "sun":
      return "SunOS"
    elif attr == "unknown_linux":
      return "unknown_linux"
    elif attr == "unknown":
      return "unknown"
    else:
      raise AttributeError, attr

def linuxType(self):
  """Uses various methods to determine Linux Type"""

  if platform.dist()[0] == self.rhel:
      return self.rhel
    elif platform.uname()[1] == self.ubu:
      return self.ubu
```

```
        else:
          return self.unknown_linux

    def queryOS(self):
        if platform.system() == "Darwin":
          return self.osx
        elif platform.system() == "Linux":
          return self.linuxType()
        elif platform.system() == self.sun:
          return self.sun
        elif platform.system() == self.fbsd:
            return self.fbsd

    def fingerprint():
        type = OpSysType()
        print type.queryOS()

    if __name__ == "__main__":
        fingerprint()
```

下面看一下在各种不同平台下运行时的输出。

Red Hat:

```
[root@localhost]/# python fingerprint.py
redhat
```

Ubuntu:

```
root@ubuntu:/# python fingerprint.py
ubuntu
```

Solaris 10 or SunOS:

```
bash-3.00# python fingerprint.py
SunOS
```

FreeBSD:

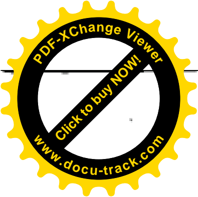
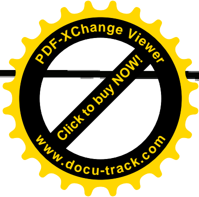
```
# python fingerprint.py
FreeBSD
```

命令的输出不是非常吸引人，但是它确实为我们提供了强有力的帮助。有了这一简单模块，我们就可以编写跨平台代码了，我们可以针对操作系统的类型在字典中进行查询，如果匹配了哪一个，则运行适合该平台的代码。使用跨平台API最切实的好处体现在编写通过ssh密钥进行网络管理的脚本中。代码可以同时在多个平台上运行，却可以得到统一的结果。

使用SSH密钥，挂载NFS的源目录和使用Python实现跨平台系统管理

一种能够管理各种各样*nix主机的方式是联合使用ssh密钥，加载NFS的通用共享源

www.TopSage.com

目录，以及跨平台的Python代码。我们将这一过程细分为若干步骤，便于更为清晰地讲述。

第一步：在你管理的主机系统上创建ssh公钥。注意，这可能会根据平台有所变化。请查询相关操作系统文档或对ssh使用man命令来查看详细内容。参见例8-3。

注意： 针对下面的示例需要指出的一点是，出于演示的需要，这里会为根用户创建ssh密钥，但是为了获得更好的安全性最好创建一个具有使用sudo命令权限的普通用户账号来运行这个脚本。

例8-3：创建一个ssh公钥

```
[ngift@Macintosh-6][H:11026][J:0]% ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
6c:2f:6e:f6:b7:b8:4d:17:05:99:67:26:1c:b9:74:11 root@localhost.localdomain
[ngift@Macintosh-6][H:11026][J:0]%
```

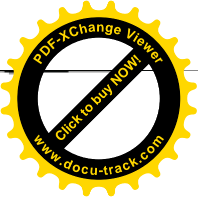
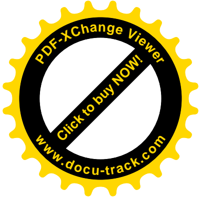
第二步：SCP公钥到主机，并创建一个名为*authorized_keys*的文件。参见例8-4。

例8-4：分发ssh密钥

```
[ngift@Macintosh-6][H:11026][J:0]% scp id_leop_lap.pub root@10.0.1.51:~/.ssh/
root@10.0.1.51's password:
id_leop_lap.pub
  100% 403      0.4KB/s    00:00
[ngift@Macintosh-6][H:11027][J:0]% ssh root@10.0.1.51
root@10.0.1.51's password:
Last login: Sun Mar 2 06:26:10 2008
[root@localhost]~# cd .ssh
[root@localhost]~/.ssh# ll
total 8
-rw-r--r-- 1 root root  403 Mar  2 06:32 id_leop_lap.pub
-rw-r--r-- 1 root root 2044 Feb 14 05:33 known_hosts
[root@localhost]~/.ssh# cat id_leop_lap.pub > authorized_keys
[root@localhost]~/.ssh# exit

Connection to 10.0.1.51 closed.
[ngift@Macintosh-6][H:11028][J:0]% ssh root@10.0.1.51
Last login: Sun Mar 2 06:32:22 2008 from 10.0.1.3
[root@localhost]~#
```

第三步：挂载通用NFS源目录，该目录中包含需要用客户端来运行的模块。通常最简单的方法是使用autofs，然后创建一个符号链接。此外，还可以通过版本控制系统来实现。在版本控制系统中，通过ssh发送命令给远端主机，告诉其升级本地svn库的全部代

www.TopSage.com

码。接下来脚本就可以运行最新的模块了。例如，在一个Red Hatbased系统中，可以这样操作：

```
ln -s /net/nas/python/src /src
```

第四步：写一个分发器，以在网络中各台主机上运行代码。有了ssh密钥和通用的加载NFS的src目录，或是实现版本控制的src目录，这一任务会变得非常简单。不失一般性，让我们从建立最简单的基于ssh的分发系统示例开始。如果之前从没有做过，你会对可以如此简单地实现这么强大的功能而感到惊喜的。在例8-5中，我们运行了一个简单的uname命令。

例8-5：简单的基于ssh的分发器

```
#!/usr/bin/env python
import subprocess

"""
A ssh based command dispatch system

"""
machines = ["10.0.1.40",
"10.0.1.50",
"10.0.1.51",
"10.0.1.60",
"10.0.1.80"]

cmd = "uname"
for machine in machines:
  subprocess.call("ssh root@%s %s" % (machine, cmd), shell=True)
```

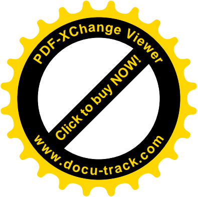
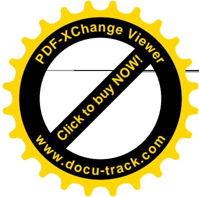
在5个混合了CentOS 5、FreeBSD 7、 Ubuntu 7.1和Solaris 10的IP地址上运行这个脚本，执行过程如下所示：

```
[ngift@Macintosh-6][H:11088][J:0]% python dispatch.py
Linux
Linux
Linux
SunOS
FreeBSD
```

我们编写了一个更准确的操作系统fingerprint（指纹）脚本来取得对主机的更准确的描述。这些主机是我们分发命令的对象，会通过命令临时在远程计算机创建src目录并将代码复制到每一台计算机上。当然，有了分发脚本后，最迫切需要的就是针对该工具的一个健壮的命令行接口（CLI）。因为没有它，我们每次想要做些不同的事情时，都需要像下面这样改动脚本：

```
cmd = "mkdir /src"
or:
cmd = "python /src/fingerprint.py"
```


www.TopSage.com

```
or even:
subprocess.call("scp fingerprint.py root@%s:/src/" % machine, shell=True)
```

我们将在运行fingerprint.py脚本之后进行修改，但是先看一下这个新的cmd：

```
#!/usr/bin/env python
import subprocess

"""
A ssh based command dispatch system

"""
machines = ["10.0.1.40",
"10.0.1.50",
"10.0.1.51",
"10.0.1.60",
"10.0.1.80"]

cmd = "python /src/fingerprint.py"
for machine in machines:
  subprocess.call("ssh root@%s %s" % (machine, cmd), shell=True)
```

现在看一下新的输出结果：

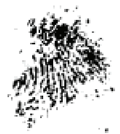
```
[ngift@Macintosh-6][H:11107][J:0]# python dispatch.py
redhat
ubuntu
redhat
SunOS
FreeBSD
```

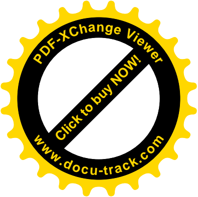
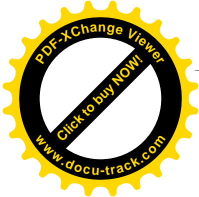
这一结果要归功于fingerprint.py模块。当然，新的分发代码段需要经过大量修改才能使用，因为我们不得不通过编辑脚本来适应需求的变化。我们需要一个更好的工具，现在就来创建一个吧。

创建一个跨平台的系统管理工具

在一个简单的基于ssh的分发系统中使用ssh密钥是非常有效的，但是很难扩展或复用。让我们创建一个之前使用的工具的问题列表，然后列出修复这些问题的必要条件。问题：主机列表是硬编码到脚本中的;发出的命令是硬编码到脚本中的，一次仅能执行一个命令;我们不得不在所有的机器上运行相同的命令列表，不能挑选或进行选择;被分发的代码块需要等待每一命令的返回响应。必要条件：我们需要一个命令行工具，可以根据IP地址读取config文件，执行命令；我们需要带选项的CLI接口来向主机发送命令；我们需要在独立的线程池中执行分发，这样进程不会被阻塞。

这看起来与创建一个基本的配置文件解析语法有些不一样，主机占一节，命令占一节。参见例8-6。



例8-6：分发配置文件

```
[MACHINES]
CENTOS: 10.0.1.40
UBUNTU: 10.0.1.50
REDHAT: 10.0.1.51
SUN: 10.0.1.60
FREEBSD: 10.0.1.80
[COMMANDS]
FINGERPRINT : python /src/fingerprint.py
```

接下来需要写一个函数来阅读config文件，并将MACHINES与COMMANDS分离，这样我们可以一个一个进行迭代。参见例8-7。

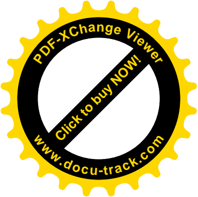
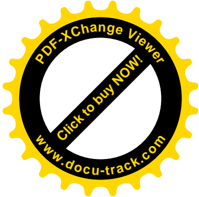
注意： 有一件事情需要注意：我们的命令会从config文件被随机加载。在许多情况下，这是一个优点，因为仅写一个Python文件，就可以将其作为配置文件来使用。

例8-7：高级ssh分发器

```
#!/usr/bin/env python
import subprocess
import ConfigParser

"""
A ssh based command dispatch system
"""

def readConfig(file="config.ini"):
  """Extract IP addresses and CMDS from config file and returns tuple"""
  ips = []
  cmds = []
  Config = ConfigParser.ConfigParser()
  Config.read(file)
  machines = Config.items("MACHINES")
  commands = Config.items("COMMANDS")
  for ip in machines:
    ips.append(ip[1])
for cmd in commands:
cmds.append(cmd[1])
return ips, cmds

ips, cmds = readConfig()

#For every ip address, run all commands
for ip in ips:
  for cmd in cmds:
    subprocess.call("ssh root@%s %s" % (ip, cmd), shell=True)
```

这段代码使用起来很方便。我们可以强行指定一个命令和主机列表，然后立即执行。如果查看命令的输出结果，可以比较一下是否与下面相同：

www.TopSage.com

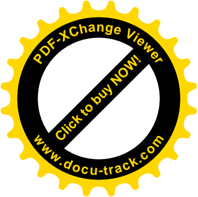
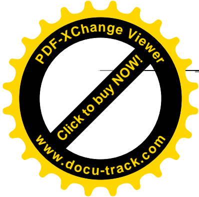
```
[ngift@Macintosh-6][H:11285][J:0]# python advanced_dispatch1.py
redhat
redhat
ubuntu
SunOS
FreeBSD
```

尽管已经有了一个非常高级的工具，但仍不能满足我们最初的需求，即在一个独立的线程池中运行分发命令。幸运的是，可以使用进程一章中的一些技巧来为分发器方便地创建线程池。例8-8演示了添加线程可以做些什么。

例8-8：多线程命令分发工具

```
#!/usr/bin/env python
import subprocess
import ConfigParser
from threading import Thread
from Queue import Queue
import time
"""
A threaded ssh-based command dispatch system

"""
start = time.time()
queue = Queue()

def readConfig(file="config.ini"):
    """Extract IP addresses and CMDS from config file and returns tuple"""
    ips = []
    cmds = []
    Config = ConfigParser.ConfigParser()
    Config.read(file)
    machines = Config.items("MACHINES")
    commands = Config.items("COMMANDS")
    for ip in machines:
        ips.append(ip[1])
    for cmd in commands:
        cmds.append(cmd[1])
    return ips, cmds

def launcher(i,q, cmd):
    """Spawns command in a thread to an ip"""
    while True:
        #grabs ip, cmd from queue
        ip = q.get()
        print "Thread %s: Running %s to %s" % (i, cmd, ip)
        subprocess.call("ssh root@%s %s" % (ip, cmd), shell=True)
        q.task_done()

#grab ips and cmds from config
ips, cmds = readConfig()

#Determine Number of threads to use, but max out at 25
if len(ips) < 25:
    num_threads = len(ips)
```

www.TopSage.com

```
else:
    num_threads = 25

#Start thread pool
for i in range(num_threads):
    for cmd in cmds:
        worker = Thread(target=launcher, args=(i, queue,cmd))
        worker.setDaemon(True)
        worker.start()

print "Main Thread Waiting"
for ip in ips:
    queue.put(ip)
queue.join()
end = time.time()
print "Dispatch Completed in %s seconds" % end - start
```

如果我们查看新的线程化的分发引擎，可以看到命令被分发，并且大约在1.2秒内返回。如果希望查看速度差异，应该在原来的分发器中添加一个计时器，并且对结果进行比较：

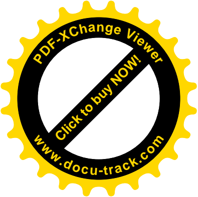
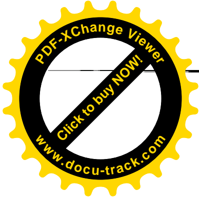
```
[ngift@Macintosh-6][H:11296][J:0]# python threaded_dispatch.py
Main Thread Waiting
Thread 1: Running python /src/fingerprint.py to 10.0.1.51
Thread 2: Running python /src/fingerprint.py to 10.0.1.40
Thread 0: Running python /src/fingerprint.py to 10.0.1.50
Thread 4: Running python /src/fingerprint.py to 10.0.1.60
Thread 3: Running python /src/fingerprint.py to 10.0.1.80
redhat
redhat
ubuntu
SunOS
FreeBSD
Dispatch Completed in 1 seconds
```

通过在原来的分发器中添加一些简单的计时代码，可以得到如下所示的新结果：

```
[ngift@Macintosh-6][H:11305][J:0]# python advanced_dispatch1.py
redhat
redhat
ubuntu
SunOS
FreeBSD
Dispatch Completed in 3 seconds
```

从这个最简单的测试中可以看到，线程化的版本大约快三倍。如果使用我们的分发工具来监测一个布满计算机的网络，例如500台计算机，而不是5台，它会在性能上显示出巨大的差别。到目前为止，我们的跨平台系统管理工具运行得非常好，接下来我们将开始另一个目标，使用它来创建一个跨平台的网络。

www.TopSage.com

注意： 应该注意的是，使用并行IPython或许是一个更好的解决方案。参见：*http://ipython.scipy.org/moin/Parallel_Computing*。

创建一个跨平台网络

我们已经知道如何并行地将作业发布到多台主机上，识别这些主机上运行的操作系统，并最终创建一个带有EMP（EMP可以创建指定提供商的包）的统一说明，那么将所有这些技术合并在一起使用不是会更有意义吗？接下来就使用这三种技术来快速方便地创建跨平台网络。

随着虚拟机技术的出现，为任何非专有*nix操作系统（Debian/Ubuntu、RedHat/CentOS、FreeBSD和Solaris）创建一个虚拟主机都变得非常容易。现在，当你创建了一个需要的工具并想共享给其他人（例如你公司的同事）使用时，可以十分方便地创建一个“build farm”（或许就在你运行脚本的笔记本上创建），然后再为其快速创建一个提供商包。

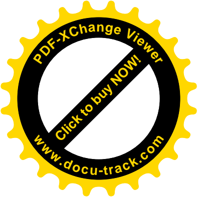
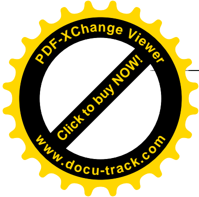
那么，又是如何工作的呢？自动化程度最高的有效方法是创建一个通用的加载了NFS的软件包树，并且赋予所有服务器访问这一挂载点的权限。然后，使用之前创建的工具来创建服务器软件包子树，该子树是一个加载了NFS的目录。因为EPM允许创建一个简单的说明或是文件列表，并且我们已经创建了“fingerprint”脚本，所有比较困难的工作就已经完成了。好的，让我们编写代码来进行实现之。以下示例展示了创建脚本的过程：

```
#!/usr/bin/env python
from fingerprint import fingerprint
from subprocess import call

os = fingerprint()

#Gets epm keyword correct
epm_keyword = {"ubuntu":"dpkg", "redhat":"rpm", "SunOS":"pkg", "osx":"osx"}

try:
   epm_keyword[os]
except Exception, err:
   print err
subprocess.call("epm -f %s helloEPM hello_epm.list" % platform_cmd, shell=True)
```

现在，编辑*config.ini*文件，修改它并运行我们的新脚本。

```
[MACHINES]
CENTOS: 10.0.1.40
UBUNTU: 10.0.1.50
REDHAT: 10.0.1.51
SUN: 10.0.1.60
FREEBSD: 10.0.1.80
```

www.TopSage.com



PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

```
[COMMANDS]
FINGERPRINT = python /src/create_package.py
```

现在，运行线程化的版本分发工具，我们可以在很短的时间里为CentOS、Ubuntu、Red Hat、FreeBSD和 Solaris创建包。由于有的地方需要有错误处理，这一示例还不应视为产品代码，但是这的确是一个非常不错的示例。可以看到，Python在几分钟或几小时内就可以完成处理。

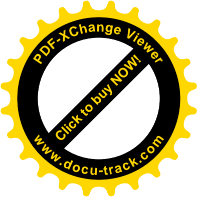
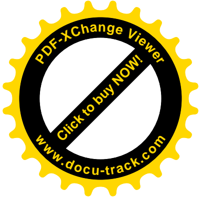
PyInotify

如果有GNU/Linux平台的工作经验，那么你会喜欢PyInotify的。根据文档说明，PyInotify是“一个查看文件系统变化的Python模块”。可以在以下网址看到：官方项目主页*http://pyinotify.sourceforge.net*。例8-9演示了PyInotify是如何工作的。

例8-9：事件监测的Pyinotify脚本

```
import os
import sys
from pyinotify import WatchManager, Notifier, ProcessEvent, EventsCodes

class PClose(ProcessEvent):
    """
    Processes on close event
    """

    def __init__(self, path):
        self.path = path
        self.file = file

    def process_IN_CLOSE(self, event):
        """
        process 'IN_CLOSE_*' events
        can be passed an action function
        """
        path = self.path
        if event.name:
            self.file = "%s" % os.path.join(event.path, event.name)
        else:
            self.file = "%s" % event.path
        print "%s Closed" % self.file
        print "Performing pretend action on %s...." % self.file
        import time
        time.sleep(2)
        print "%s has been processed" % self.file

class Controller(object):

    def __init__(self, path='/tmp'):
        self.path = path

    def run(self):
        self.pclose = PClose(self.path)
```


www.TopSage.com

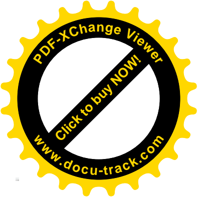
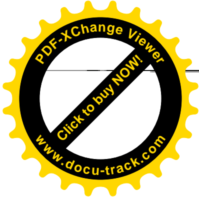
```
        PC = self.pclose
        # only watch these events
        mask = EventsCodes.IN_CLOSE_WRITE | EventsCodes.IN_CLOSE_NOWRITE

        # watch manager instance
        wm = WatchManager()
        notifier = Notifier(wm, PC)

    print 'monitoring of %s started' % self.path

    added_flag = False
        # read and process events
        while True:
            try:
                if not added_flag:
                    # on first iteration, add a watch on path:
                    # watch path for events handled by mask.
                    wm.add_watch(self.path, mask)
                    added_flag = True
                notifier.process_events()
                if notifier.check_events():
                    notifier.read_events()
            except KeyboardInterrupt:
                # ...until c^c signal
                print 'stop monitoring...'
                # stop monitoring
                notifier.stop()
                break
            except Exception, err:
                # otherwise keep on watching
                print err

def main():
    monitor = Controller()
    monitor.run()

if __name__ == '__main__':
    main()
```

如果运行这个脚本，当在*/tmp*目录下存入任何文档时，它都会做一些处理。这会启示你去考虑如何利用其完成一些更有价值的工作，例如添加一个执行动作的回调。在数据一节中的一些代码可以帮助完成自动查找并删除重复，或是执行TAR命令对匹配fnmatch表达式的文档进行打包。总之，在Linux下运行Python模块非常有意义且非常实用。

OS X

OS X的出现可以说非常令人惊喜。一方面，在Cocoa中它有世界上最好的用户界面，另一方面，它完全兼容POSIX的Unix操作系统。每一个Unix操作系统提供商都曾努力去实现这两个目标，但都失败了，而OS X却取得了成功：它将Unix带入主流。具有Leopard的OS X包括Python2.5.1，Twisted和许多其他的Python工具。

www.TopSage.com

OS X也遵从某些奇怪的标准，提供服务器版本和普通版本。对于所有Apple可以正常完成的事情，使用OS X或许需要换个思路重新考虑，我们且将这一点放在以后讨论。操作系统的服务器版本提供给管理员一些更好用的命令行工具，以及一些面向Apple的专用工具，例如，可以访问NetBoot主机，运行LDAP目录服务器等。

脚本DSCL或目录服务工具

DSCL表示目录服务命令行，它可以方便地提供对OS X目录服务API的连接。DSCL允许读取、创建、删除记录，因此Python很自然地胜任此工作。例8-10演示了如何使用IPython来脚本化DSCL，以读取Open Directory的属性及值。

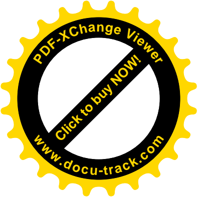
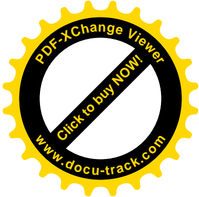
注意： 在示例中我们仅读取属性，如果需要执行其他类似操作也可以使用相同的技术，只须对示例代码进行简单的修改即可。

例8-10：使用DSCL和IPython交互获取用户记录

```
In [42]: import subprocess

In [41]: p = subprocess.Popen("dscl . read /Users/ngift", shell=True,stdout=subprocess.PIPE)

In [42]: out = p.stdout.readlines()

In [43]: for line in out:
line.strip().split()


Out[46]: ['NFSHomeDirectory:', '/Users/ngift']
Out[46]: ['Password:', '********']
Out[46]: ['Picture:']
Out[46]: ['/Library/User', 'Pictures/Flowers/Sunflower.tif']
Out[46]: ['PrimaryGroupID:', '20']
Out[46]: ['RealName:', 'ngift']
Out[46]: ['RecordName:', 'ngift']
Out[46]: ['RecordType:', 'dsRecTypeStandard:Users']
Out[46]: ['UniqueID:', '501']
Out[46]: ['UserShell:', '/bin/zsh']
```

需要指出一点，为了使用dscl命令，Apple对本地以及LDAP/Active目录账号管理都进行了中心化。当与其他的LDAP管理工具进行比较，甚至如果将Python取出，你都会感受到dscl带来的耳目一新的感觉。我们没有时间进一步讲述其中的细节，但是使用Python来使dscl脚本化非常简单，可以方便地实现自动管理本地数据库或是LDAP数据库（例如Open Directory），如果这样做，之前的代码也会给你一些启发。

www.TopSage.com

OS X 脚本 API

通常，为了使用OS X，对于系统管理员来说，了解一些与实际的UI进行交互的高级脚本是必要的。使用OS X Leopard、Python和Ruby，我们可以获得最佳的Scripting Bridge。参考以下链接中的文档可以获得更多的帮助：*http://developer.apple.com/documentation/Cocoa/Conceptual/Ruby/PythonCocoa/Introduction/Introduction.html*。

一个访问OSA或Open Scripting Architecture的方法是使用py-appscript，在以下链接可以看到项目主页：*http://sourceforge.net/projects/appscript*。使用py-appscript是非常有意义的，因为py-appscript的功能非常强大的，且赋予了Python与非常丰富的OSA构架进行交互的能力。在进一步学习之前，让我们构建一个简单的`osascript`命令行工具，来演示脚本化的API是如何工作的。可以使用Leopard编写`osascript`命令行工具，并且像Bash或Python脚本那样执行。接下来就创建这个脚本，且取名为`bofh.osa`，然后将其设为可执行的。参见例8-11。

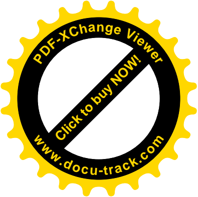
例8-11：osascript脚本

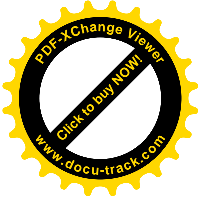
```
#!/usr/bin/osascript
say "Hello, Bastard Operator From Hell" using "Zarvox"
```

如果从命令行运行该脚本，会出现一个陌生的声音对我们说“Hello”。这一处理方式似乎有点蠢笨，但是这就是OS X，希望你也能像这样完成其他的操作。现在进一步使用appscript在Python中访问相同的API，但是这次是在IPython中交互地完成操作。下面是这个示例的一个交互版本，包括`appscript`源代码，它会按字母顺序显示输出所有的运行进程：

```
In [4]: from appscript import app

In [5]: sysevents = app('System Events')

In [6]: processnames = sysevents.application_processes.name.get()

In [7]: processnames.sort(lambda x, y: cmp(x.lower(), y.lower()))

In [8]: print '\n'.join(processnames)
Activity Monitor
AirPort Base Station Agent
AppleSpell
Camino
DashboardClient
DashboardClient
Dock
Finder
Folder Actions Dispatcher
GrowlHelperApp
GrowlMenu
iCal
iTunesHelper
```

www.TopSage.com






```
JavaApplicationStub
loginwindow
mdworker
PandoraBoy
Python
quicklookd
Safari
Spotlight
System Events
SystemUIServer
Terminal
TextEdit
TextMate
```

如果碰巧需要使用针对OS X的应用，来完成工作流自动化任务，appscript是一个非常不错的工具。它可以在Python中完成我们通常使用Applescript来完成的工作。Noah写了一篇文章，对此进行了阐述，参见：*http://www.macdevcenter.com/pub/a/mac/2007/05/08/using-python-and-applescriptto-get-the-most-out-of-your-mac.html*。

系统管理员需要做的一些事情包括创建进行交互的批处理操作，例如，Adobe After Effects。一个最终的建议是：通过Applescript Studio可以非常快速地在OS X上创建Python GUI，并且通过“do shell script”调用Python。一个鲜为人知的事实是，Carbon Copy Cloner的原始版本是在Applescritp studio下编写完成的。如果有充裕的时间，值得在其身上花上一些时间进行研究。

自动再映像主机

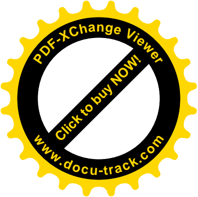
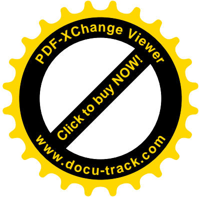
另外一个使用OS X开发的极为神奇的，具有领先水平的工具是ASR命令行工具。该工具是非常流行的自由软件克隆工具Carbon Copy Cloner中的一个关键组件，并且在许多自动再映像主机中起到重要作用。事实上，它可以完全自主开始工作。一个用户应该仅需要重启主机并且按下“N”键选择网络启动就可以了，或者说主机会自行修复。

下面是一个简化的硬编码自动启动脚本，可以在网络启动的映像上运行，自动重新映像一台计算机，当然也可以从硬盘的第二个分区运行。*/Users*目录和任何其他重要的目录都应该被符号链接到另一个分区或者可用的网络上，参见例8-12。

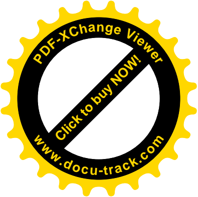
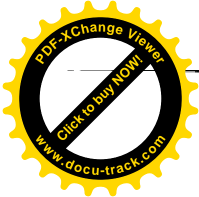
例8-12：自动映射OS X分区并使用WXPython显示进度

```
#!/usr/bin/env pythonw
#automatically reimages partition

import subprocess
import os
import sys
import time
from wx import PySimpleApp, ProgressDialog, PD_APP_MODAL, PD_ELAPSED_TIME
```


www.TopSage.com

```
#commands to rebuild main partition using asr utility
asr = '/usr/sbin/asr -source '

#path variables
os_path = '/Volumes/main'
ipath = '/net/server/image.dmg '
dpath = '-target /Volumes/main -erase -noprompt -noverify &'
reimage_cmd = "%s%s%s" % (asr,ipath, dpath)

#Reboot Variables
reboot = 'reboot'
bless = '/usr/sbin/bless -folder /Volumes/main/System/Library/CoreServices -setOF'

#wxpython portion
application = PySimpleApp()
dialog = ProgressDialog ('Progress', 'Attempting Rebuild of Main Partition',
                        maximum = 100, style = PD_APP_MODAL | PD_ELAPSED_TIME)

def boot2main():
    """Blesses new partition and reboots"""
    subprocess.call(bless, shell=True)
    subprocess.call(reboot, shell=True)

def rebuild():
    """Rebuilds Partition"""
    try:
        time.sleep(5) #Gives dialog time to run
        subprocess.call(reimage_cmd)
    except OSError:
        print "CMD: %s [ERROR: invalid path]" % reimage_cmd
        sys.exit(1)
    time.sleep(30)
    while True:
        if os.path.exists(os_path):
            x = 0
            wxSleep(1)
            dialog.Update ( x + 1, "Rebuild is complete...\n rebooting to main partition\n
                            ...in 5 seconds..")
            wxSleep(5)
            print "repaired volume.." + os_path
            boot2main() #calls reboot/bless function
            break
        else:
            x = 0
            wxSleep(1)
            dialog.Update ( x + 1, 'Reimaging.... ')

def main():
    if os.path.exists(os_path):
        rebuild()
    else:
        print "Could not find valid path...FAILED.."
        sys.exit(1)
if __name__ == "__main__":
    main()
```

回顾上述代码，脚本试图重新映像一个分区并且弹出一个WXPython进度条。如果路径被正确设置，并且没有其他错误，将继续使用ASR命令映像硬件驱动和一个自动升级的进度条，被重新映像的分区再次成为根卷，然后提示重新启动计算机。

该脚本很容易成为企业软件发布和管理系统的基础，因为它可以根据硬件签名发布不同的映像，甚至通过查看硬件驱动的老名字来实现。接下来，软件包可以使用OS X的包管理系统或是开放源码的工具radmind来发布。一种情况是，首先自动重映像一个新的OS X安装，完成一个基本的操作系统安装，然后使用radmind完成剩下的步骤。

如果你正在做一些重要的OS X系统管理工作，那么应该学习radmind。radmind是tripwire系统类型，可以检查文件系统的变化，并且可以基于变化对主机进行恢复。如果需要更多的资料，可以参考*http://rsug.itd.umich.edu/software/radmind/*。尽管radmind没有用Python语言编写，但它可以很容易地在Python中脚本化。

从Python中管理Plist文件

在第3章中，我们使用ElementTree解析了来自`system_profiler`的XML数据流，但是OS X上的Python与`plistlib`是绑定在一起的，`plistlib`允许解析和创建Plist文件。模块自身的名称即为`plistlib`。这里没有时间通过示例对其进行介绍，但是它值得你去认真学习一下。

Red Hat Linux系统管理

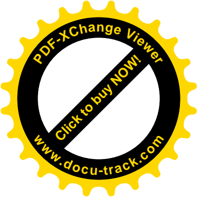
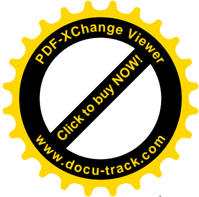
Red Hat使用Python来做许多事情，可以将Python作为一个伙伴或是一个操作系统。在Red Hat中一些最有意义的新应用来自Emerging Technologies group（前瞻技术组）：*http://et.redhat.com/page/Main_Page*。下面是一些使用Python的项目列表：

- Libvert，虚拟API的虚拟计算机管理器
- 使用libvirt VirtInst建立的Python+PyGTK的管理应用
- 一个使用libvirt来简化访客VMs配置的Python库
- Cobbler，针对PXE和虚拟化可以建立全自动的网络启动服务器
- Virt-Factory：基于web的虚拟管理，具有侧重于应用的特点
- FUNC（Fedora统一网络控制器）

Ubuntu管理

在所有的主流Linux发行版中， Ubuntu或许是与Python关联最多的。其中的部分原因是

www.TopSage.com

创建者Mark Shuttleworth是一个资深的Python黑客（可以追溯到90年代初期）。在这里可以找到Ubuntu的一个非常不错的Python源码包：*https://launchpad.net/*。

Solaris系统管理

从90年代后期到本世纪初，Solaris是首选的、Unix的“Big Iron”发行版本。在本世纪初，Linux的metioric迅速被剪裁成Solaris的metioric。Sun在发展过程中确实遇到了一些麻烦。但是，最近一些系统管理员、开发人员以及企业又重新开始谈论Sun。

将来，Sun将在一些有意义的开发方向上以6个月为一个发行周期，就像Ubunto具有18个月的支持窗口一样。同时也将采用Ubuntu的单CD方法，抛弃大的DVD发行版。最后，通过与Solaris社区开发版的整合，融入一些Red Hat与Fedora的思想。你可以下载一张最新的CD版本，或是通过*http://www.opensolaris.com*预定一个。

对于一名使用Python的系统管理员，这意味着什么呢？Sun突然变得令人兴奋，因为它具有一些非常有意义的技术，包括ZFS、容器，以及在某些方面等同于VMware虚拟机的LDOM。Python在Solaris中工作得非常不错，甚至在它的开发包管理系统中都被普遍使用。

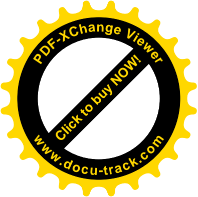
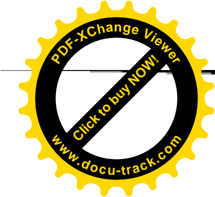
虚拟化

2007年8月14日，VMware上市，募集了数十亿美元，并且巩固了“虚拟化”作为数据中心和系统管理的未来发展方向。预测未来总是有风险的，但是“数据中心操作系统”一词已经在一些大的公司被谈论来谈论去了。每一个来自微软、Red Hat，Oracle的人都会走到支持虚拟化的队伍中来。有一点是毫无疑问的，即虚拟化将彻底改变数据中心和系统管理作业。可以说，在经常形容为“颠覆性技术”的技术中，虚拟化是非常简单的一个。

虚拟化对系统管理员来说是一把双刃剑，一方面它开创了可以很容易地对配置和应用进行测试的方法，另一方面，它也显著增加了管理的复杂性。一台计算机不再只安装一个操作系统，也不再仅能处理小的商业活动，它可以是一个大的数据中心。而所有的效率是需要付出一些代价的，这就是其复杂性超出了普通系统管理员的能力。

你或许正在家中阅读到这里，然后想到：什么事情必须使用Python去做呢？回答是相当多。Noah最近的雇员Racemi曾经在Python中写了一个全面的数据中心管理应用程序，可以处理虚拟化。Python能够以一种基本的方法与虚拟机交互，可以控制虚拟机，可以通过Python API移动物理计算机到虚拟机。Python在虚拟世界中已经占有一席之地，毫无疑问，在未来的数据中心操作系统中Python将起到重要作用。

www.TopSage.com

VMware

正如我们之前提到的，VMware在当前的虚拟化领域中占有重要地位。通过程序对虚拟机进行全面控制，是人们梦寐以求的目标。幸运的是有一些Perl、XML-RPC、Python和C 的API可以使用。在写这本书时，Python在这方面的实现还是有限的，但是下一步情况会有所改变。VMware的新方向出现在XML-RPC API方面。VMware有一些具有不同API的不同的产品。一些你或许希望脚本化的产品包括VMware Site Recovery Manager、VMware ESX Server、VMure Server和VMuare Fusion。

这里没有时间进一步介绍这些技术如何脚本化，因为这超出了本书的范围，但是我们会紧密跟踪这些产品，验证Python在其中的作用。

云计算

当虚拟化刚从喧嚣中走出来时，云计算的出现再次吸引了人们的目光。简单地讲，云计算是根据工作负载需求进行响应的资源使用方法。在云计算方面有两个大的应用是Amazon和Google。在这本书送到出版商之前几周Google刚刚抛出了“云炸弹”。其中Google提供了一个非常有用的twist，当前仅支持Python。这是一本Python编程的书，我们确信它不会令你太失望。

在这一节将介绍一些可用的API，或许你在处理Amazon和Google App引擎时会需要这些API。最后，我们将讨论这一技术是如何影响系统管理员的工作的。

使用Boto的Amazon Web服务

处理Amazon的云计算基础结构的一个非常好的选择是Boto。使用Boto，你可以完成下面的事情：Simple Storage Service, Simple Queue Service, Elastic Compute Cloud, Mechanical Turk和SimpleDB。因为这是非常新且极其强大的API，我们建议你自己查看一下项目主页：*http://code.google.com/p/boto/*。这里有最新的信息，比起我们能给你的要好多了。下面是一个简单的示例，演示了SimpleDB是如何工作的：

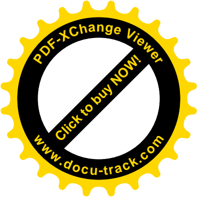
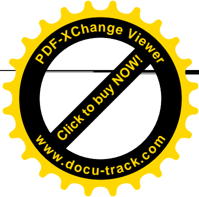
初始化连接:

```
In [1]: import boto
In [2]: sdb = boto.connect_sdb()
```

创建新域:

```
In [3]: domain = sdb.create_domain('my_domain')
```


www.TopSage.com

添加新元素:

```
In [4]: item = domain.new_item('item')
```

以上就是当前API方式的工作，但是你应该在svn库查看测试，以获得处理过程的具体概念：*http://code.google.com/p/boto/source/browse*。值得注意的是，查看测试是理解库是如何工作的一个最好的方式。

Google App引擎

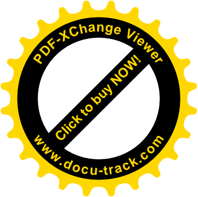
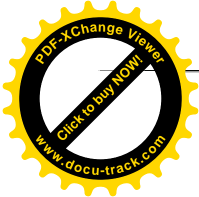
Google App引擎是作为测试服务发布的，从发布之日起就广泛引起注意。它允许你的应用程序在Google的基础框架下免费运行。App Engine现在具有严格的Python API，但是在某些点可能有变化。关于App Engine的另一个有意义的事情是它也可以整合Google的其他服务。

> 名人简介：GOOGLE APP ANGINE TEAM
>
> **Kevin Gibbs**
>
>
>
> Kevin Gibbs是Google App Engine中的先进技术。Kevin在2004年加入Google，在Google App Engine工作之前，Kevin曾在Google的系统基础框架组工作了几年，在那里他从事集群管理系统工作，这部分工作处于Google产品与服务的下层。Kevin也是Google Suggest的创建者，Google Suggest可以在你输入时交互地给出搜索建议。在加入Google之前，Kevin在IBM的Advanced Internet Technology group（互联网高级技术组）工作，主要从事各种工具的开发。如今将自己数据中心中的数据发送到另一个数据中心已日益成为可能，这也进一步影响着系统管理员的工作方式。掌握如何与Google App Engine进行交互的技术可能成为系统管理员的新的“杀手锏”，因此对其进行深入学习也是非常有意义的。

我们会见了一些来自App Engine Team的工作人员，与他们谈论了影响系统管理员工作的因素。他们提到了下面这些任务：

1. 大量数据的上传：*http://code.google.com/appengine/articles/bulkload.html*。系统管理员经常处理移动大量数据的任务，这是一个在Google App Engine的app背景下完成这一任务的工具。
2. 记录日志：*http://code.google.com/appengine/articles/logging.html*。

www.TopSage.com

3. 邮件API：send_mail_to_admins()函数：*http://code.google.com/appengine/docs/mail/functions.html*。

   在系统管理环境中，这对于完成监测任务非常有意义。对于一些重要的异常或关键行为，你可以向app的系统管理员自动发送邮件。

4. 为完成普通任务而设置的Cron作业。

   这不是Google App Engine的最直接的一部分，但是你可以在自己服务器上使用cron以常规间隔向你的app发送请求。例如，你可以有一个cron作业，每隔一小时点击*http://yourapp.com/emailsummary*，这会触发一个邮件发送机制，将邮件发送给系统管理员，邮件中包含了上一小时重要事件的统计结果。

5. 版本管理：*http://code.google.com/appengine/docs/configuringanapp.html#Required_Elements*。

   一个需要为你的app设置的字段是版本。每次你上传一个app都会使用相同的版本ID，它用新的编码进行替换。如果你改变了版本ID，你可以在正式的产品环境中同时运行多个app版本，使用admin控制台来选择哪一个版本接收实时流量。

建立一个Google App Engine的应用示例

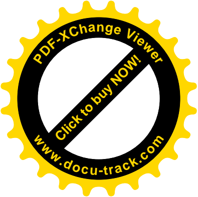
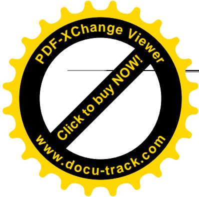
建立一个Google App Engine的应用示例之初，你首先需要下载为Google app引擎开发的SDK，可以从这里下载：*http://code.google.com/appengine/downloads.html*。你可以通过非常不错的Google AppEngine教学（网址：*http://code.google.com/appengine/docs/gettingstarted/*）来完成学习。

在这一节，我们提供了一个与Google App Engine相反的教程，因为已经有一个非常棒的教程了。如果你访问*http://greedycoin.appspot.com/*，你可以测试一个我们正要介绍的版本，以及最近的源码版本。应用程序会将变化作为输入，将其保存到数据库中，然后返回适当的变化结果。它也具有通过Google的认证API进行登录，并且执行一个最近的查询的能力。参见例8-13。

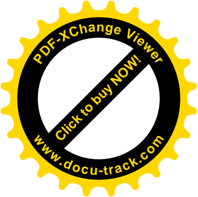
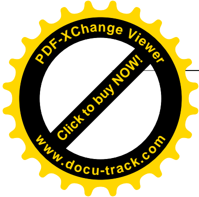
例8-13：贪婪硬币web应用

```
#!/usr/bin/env python2.5
#Noah Gift

import decimal
import wsgiref.handlers
import os

from google.appengine.api import users
from google.appengine.ext import webapp
from google.appengine.ext import db
from google.appengine.ext.webapp import template
```


www.TopSage.com

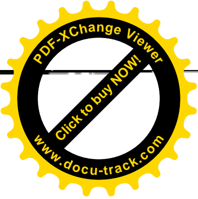
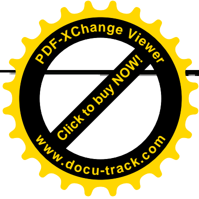
```
class ChangeModel(db.Model):
    user = db.UserProperty()
    input = db.IntegerProperty()
    date = db.DateTimeProperty(auto_now_add=True)

class MainPage(webapp.RequestHandler):
    """Main Page View"""

    def get(self):
        user = users.get_current_user()

        if users.get_current_user():
            url = users.create_logout_url(self.request.uri)
            url_linktext = 'Logout'
        else:
            url = users.create_login_url(self.request.uri)
            url_linktext = 'Login'

        template_values = {
        'url': url,
        'url_linktext': url_linktext,
        }
        path = os.path.join(os.path.dirname(__file__), 'index.html')
        self.response.out.write(template.render(path, template_values))

class Recent(webapp.RequestHandler):
    """Query Last 10 Requests"""

    def get(self):

        #collection
        collection = []
        #grab last 10 records from datastore
        query = ChangeModel.all().order('-date')
        records = query.fetch(limit=10)

        #formats decimal correctly
        for change in records:
            collection.append(decimal.Decimal(change.input)/100)

        template_values = {
        'inputs': collection,
        'records': records,
        }

        path = os.path.join(os.path.dirname(__file__), 'query.html')
        self.response.out.write(template.render(path,template_values))

class Result(webapp.RequestHandler):
    """Returns Page with Results"""
    def __init__(self):
        self.coins = [1,5,10,25]
        self.coin_lookup = {25: "quarters", 10: "dimes", 5: "nickels", 1: "pennies"}

    def get(self):
        #Just grab the latest post
        collection = {}
```

```
        #select the latest input from the datastore
        change = db.GqlQuery("SELECT * FROM ChangeModel ORDER BY date DESC LIMIT 1")
        for c in change:
            change_input = c.input

        #coin change logic
        coin = self.coins.pop()
        num, rem = divmod(change_input, coin)
        if num:
            collection[self.coin_lookup[coin]] = num
        while rem > 0:
            coin = self.coins.pop()
            num, rem = divmod(rem, coin)
            if num:
                collection[self.coin_lookup[coin]] = num

        template_values = {
        'collection': collection,
        'input': decimal.Decimal(change_input)/100,
        }

        #render template
        path = os.path.join(os.path.dirname(__file__), 'result.html')
        self.response.out.write(template.render(path, template_values))

class Change(webapp.RequestHandler):

    def post(self):
        """Printing Method For Recursive Results and While Results"""
        model = ChangeModel()
        try:
            change_input = decimal.Decimal(self.request.get('content'))
            model.input = int(change_input*100)
            model.put()
            self.redirect('/result')
        except decimal.InvalidOperation:
            path = os.path.join(os.path.dirname(__file__), 'submit_error.html')
            self.response.out.write(template.render(path,None))

def main():
    application = webapp.WSGIApplication([('/', MainPage),
                                          ('/submit_form', Change),
                                          ('/result', Result),
                                          ('/recent', Recent)],
                                          debug=True)
    wsgiref.handlers.CGIHandler().run(application)

if __name__ == "__main__":
    main()
```

作为一个相反的示例，让我们从查看运行在*http://greedycoin.appspot.com/*的版本，或者你本地的开发版本开始。这是一个南瓜色（punpkin-colored）的主题，有两个浮动的对话框，在左侧是一个表格让你输入变化，在右侧有一个导航对话框。这些或漂亮或难看的颜色以及层次都是Django模板与CSS合成的结果。Django模板可以在主目录中找到，

www.TopSage.com

也可以在我们使用的CSS在风格页中找到。这确实与Google App Engine关系不多，因此我们在这里仅告诉你若想查看更多内容，可以查阅Django模板参考资源：*http://www.djangoproject.com/documentation/templates/*。

现在我们已经介绍了Google App Engine，接下来让我们实际上感受Google App Engine的效果。或许你已经注意到了在右侧导航框的Login连接，通过用户认证API，它是可以实现的。以下是具体的代码：

```
class MainPage(webapp.RequestHandler):
    """Main Page View"""

def get(self):
    user = users.get_current_user()

    if users.get_current_user():
        url = users.create_logout_url(self.request.uri)
        url_linktext = 'Logout'
    else:
        url = users.create_login_url(self.request.uri)
        url_linktext = 'Login'

    template_values = {
    'url': url,
    'url_linktext': url_linktext,
    }
    path = os.path.join(os.path.dirname(__file__), 'index.html')
    self.response.out.write(template.render(path, template_values))
```

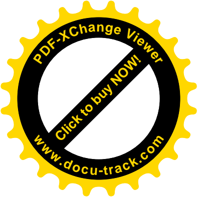
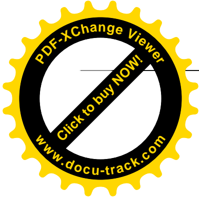
有一个继承自webapp.ReguestHandler的类，如果定义了一个get方法，可以创建一个页面，检查某个用户是否登录。如果你注意到底部的一些行，会看到用户信息被送入模板系统，之后获得Django模板文件*index.html*。平衡Google User Accounts数据库以创建页面认证非常简单，且强大得令人难以置信。如果你查看了之前的代码，这可以简单地等同于：

```
user = users.get_current_user()

if users.get_current_user():
```

在这一点上，我们建议不要太专注于这些代码，尽量添加仅为认证用户显示的代码。你甚至不需要理解事情是怎样处理的，你可以仅使用现存的条件语句来完成一些事情。

现在我们对认证只有一个含糊不清的理解，让我们进一步介绍它强大的功能。数据存储API允许你保存持久数据，并且在整个应用过程中都可以获取。为了实现这一目标，需要加载数据存储（就像之前代码中所显示的那样），然后定义模块，如下所示：

```
class ChangeModel(db.Model):
    user = db.UserProperty()
```

```
        input = db.IntegerProperty()
        date = db.DateTimeProperty(auto_now_add=True)
```

使用这一简单的类，可以创建并使用持久数据。以下是一个类，其中对数据存储使用了Python API，以获取数据库的最近十次变化，然后进行显示：

```
    class Recent(webapp.RequestHandler):
        """Query Last 10 Requests"""

        def get(self):

            #collection
            collection = []
            #grab last 10 records from datastore
            query = ChangeModel.all().order('-date')
            records = query.fetch(limit=10)

            #formats decimal correctly
            for change in records:
                collection.append(decimal.Decimal(change.input)/100)

            template_values = {
            'inputs': collection,
            'records': records,
            }

            path = os.path.join(os.path.dirname(__file__), 'query.html')
            self.response.out.write(template.render(path,template_values))
```

两行最为重要的代码为：

```
    query = ChangeModel.all().order('-date')
    records = query.fetch(limit=10)
```

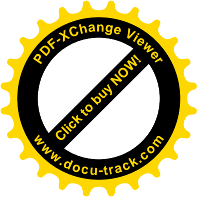
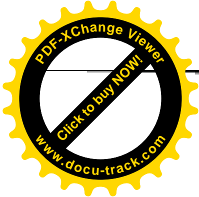
这一示例从数据存储中将结果取出，然后在一次查询中取回10个记录。在这一点上，一个简单而有意义的事情是对这一代码进行测试验，努力取回更多的记录，或是以不同的方式进行存储。这会给你一些即时而有趣的反馈结果。

最后，如果我们仔细查看下面的代码，或许能够对*change.py*文件中每一个URL对应的类进行猜测。在这一点上，我们建议通过部分修改依赖于URL的应用，来调整URL的名称，这会给你对事情如何处理有所了解。

```
    def main():
        application = webapp.WSGIApplication([('/', MainPage),
          ('/submit_form', Change),
          ('/result', Result),
          ('/recent', Recent)],
          debug=True)
        wsgiref.handlers.CGIHandler().run(application)
```

到这里，Google App Engine的反向教程就已经结束了。它给予一些启发，告诉你如何

www.TopSage.com

根据自己的需要来实现一个更好的系统管理工具。如果你还对编写更多的应用抱有浓厚兴趣，你可以查看一下Guido为Google App Engine应用编写的源码：*http://code.google.com/p/rietveld/source/browse*。

使用Zenoss从Linux上管理Windows服务器

如果你不幸有一个或多个Windows服务器的管理任务，任务可能会变得复杂，会有点令人不太愉快。Zenoss是一个可以让人大吃一惊的工具，它完全可以帮助我们。我们在第7章介绍了Zenoss，SNMP。除了业界领先的SNMP工具之外，Zenoss也提供了从Linux上与Windows服务器通过WMI进行会话的工具。当想到它的现实意义与可行性，我们不禁十分愉悦。通过与一些擅长Zenoss的人的讨论，他们将WMI信息上传到Linux上的Samba（现在可能是CIFS）服务器上，然后将其发送到Windows服务器。可能其中最有意义的部分（至少对于这本书的读者）就是你可以将WMI与Python的连接脚本化。

注意： 对WMI的语法及特点的讨论超出了这本书的范围。

目前，Zenoss的文档在介绍如何在Linux上使用Python的WMI功能方面略显不足。但是，这里将要介绍的示例应该能够为你在其上进行开发奠定一个好的基础。首先，我们介绍一个Linux系统中用于WMI与Windows服务器进行通信的非Python工具`wmic`。`wmic`是一个简单的命令行工具，可以将用户名、密码、服务器地址以及WMI请求作为命令行参数，根据给定的密钥连接到合适的服务器，向服务器发送请求，将结果从标准输出设备输出。使用这一工具的语法类似这样：

```
wmic -U username%password //SERVER_IP_ADDRESS_OR_HOSTNAME "some WMI query"
```

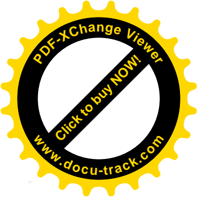
以下是administrator连接到服务器，IP地址是192.168.1.3并进行事件查询的示例：

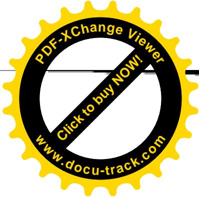
```
wmic -U Administrator%password //192.168.1.3 "SELECT * FROM Win32_NTLogEvent"
```

以下是运行该命令的部分结果：

```
CLASS: Win32_NTLogEvent
Category|CategoryString|ComputerName|Data|EventCode|EventIdentifier|
    EventType|InsertionStrings|Logfile|Message|RecordNumber|SourceName|
    TimeGenerated|TimeWritten|Type|User
...
|3|DCOM|20080320034341.000000+000|20080320034341.000000+000|Information|(null)
0|(null)|MACHINENAME|NULL|6005|2147489653|3|(,,,,14,0,0 )|System|The Event log
service was started.
|2|EventLog|20080320034341.000000+000|20080320034341.000000+000|Information|(null)
0|(null)|MACHINENAME|NULL|6009|2147489657|3|(5.02.,3790,Service Pack
2,Uniprocessor Free)|System|Microsoft (R) Windows (R) 5.02. 3790 Service Pack 2
```

www.TopSage.com


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

```
Uniprocessor Free.
|1|EventLog|20080320034341.000000+000|20080320034341.000000+000|Information|(null)
```

为了写一个类似的Python脚本，首先建立环境。在接下来的示例中，使用Zenoss v2.1.3 VMware应用程序。在这个应用程序中，一些Zenoss代码保存在zenoss用户的主目录中。其中的最重要的部分是添加wmiclient.py模块的目录到PYTHONPATH中。我们添加目录到已经存在的PYTHONPATH中，就像下面这样：

```
export PYTHONPATH=~/Products/ZenWin:$PYTHONPATH
```

一旦我们可以使用Python中需要的库，我们可以像下面这样执行脚本：

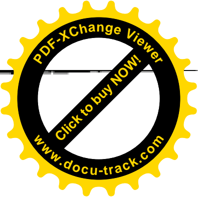
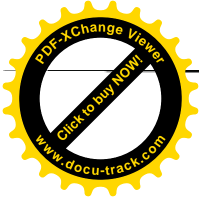
```
#!/usr/bin/env python

from wmiclient import WMI

if __name__ == '__main__':
    w = WMI('winserver', '192.168.1.3', 'Administrator', passwd='foo')
    w.connect()
    q = w.query('SELECT * FROM Win32_NTLogEvent')
    for l in q:
        print "l.timewritten::", l.timewritten
        print "l.message::", l.message
```

不同于wmic示例中输出所有的字段，这个脚本仅输出时间戳和日志信息。该脚本以Administrator身份，使用密码foo，连接到服务器192.168.1.3。然后，执行WMI查询'SELECT * FROM Win32_NTLogEvent'。之后迭代查询结果并输出时间戳和日志。这是非常简单了。

以下是运行该脚本的输出结果：

```
l.timewritten:: 20080320034359.000000+000
l.message:: While validating that \Device\Serial1 was really a serial port, a
fifo was detected. The fifo will be used.

l.timewritten:: 20080320034359.000000+000
l.message:: While validating that \Device\Serial0 was really a serial port, a
fifo was detected. The fifo will be used.

l.timewritten:: 20080320034341.000000+000
l.message:: The COM sub system is suppressing duplicate event log entries for a
duration of 86400 seconds. The suppression timeout can be controlled by a
REG_DWORD value named SuppressDuplicateDuration under the following registry
key: HKLM\Software\Microsoft\Ole\EventLog.

l.timewritten:: 20080320034341.000000+000
l.message:: The Event log service was started.

l.timewritten:: 20080320034341.000000+000
l.message:: Microsoft (R) Windows (R) 5.02. 3790 Service Pack 2 Uniprocessor
Free.
```


www.TopSage.com

但是，我们如何知道可以为这些记录使用timewritten和message属性？只需要一点黑客的专业技术就可以找到这些信息。以下执行的脚本可以帮助我们找到所需的属性：

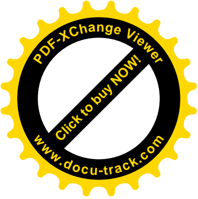
```
#!/usr/bin/env python

from wmiclient import WMI
    if __name__ == '__main__':
    w = WMI('winserver', '192.168.1.3', 'Administrator', passwd='foo')
    w.connect()
    q = w.query('SELECT * FROM Win32_NTLogEvent')
    for l in q:
        print "result set fields::->", l.Properties_.set.keys()
        break
```

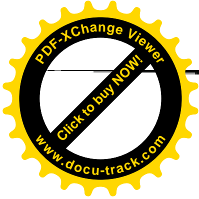
你或许注意到这个脚本看起来与其他WMI脚本十分相似。在这个脚本与其他WMI脚本之间存在两个差异：不是输出时间戳和日志信息，而是输出l.Properties_.set.keys()，在第一个结果输出后，脚本停止执行。我们在其上调用keys()的set对象实质上是一个字典。（这非常有意义，因为keys()是一个字典方法）。从WMI查询返回的每一个结果记录应该有一整套属性，这些属性与这些keys相对应。以下是我们刚刚介绍过的脚本的运行结果：

```
result set fields::-> ['category', 'computername', 'categorystring',
'eventidentifier', 'timewritten', 'recordnumber', 'eventtype', 'eventcode',
'timegenerated', 'sourcename', 'insertionstrings', 'user', 'type', 'message',
'logfile', 'data']
```

我们选择从第一个WMI脚本取得的两个属性'message'和'timewritten'都在这个关键字列表中。

虽然我们对Windows不是特别感兴趣，但是我们意识到有时需要完成的任务对我们需要掌握的技术提出了要求。这个来自Zenoss的工具使得任务轻松了许多。能够从Linux中运行WMI查询是非常有用的。如果你不得不与Windows多次打交道，那么Zenoss可以很容易在工具箱中找到合适的位置。





第9章

包管理

引言

软件包管理在决定软件开发项目是否成功中起着重要作用。包管理可以被理解为电子商务中的物流公司，就像Amazon一样，如果没有物流公司，Amazon不会存在。同样，如果操作系统或一门语言没有一个功能完善且简单健壮的包管理系统，那么它在一定程度上是不完整的。

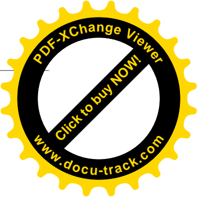
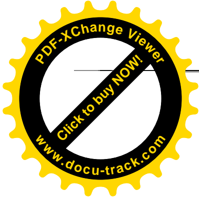
提到“包管理”，你的第一感觉或许是*.rpm*文件和yum，或是*.deb*文件和apt，或是其他操作系统级的包管理系统。我们会在这一章对包管理作进一步讨论，但主要焦点是在对Python代码和Python环境进行打包及管理。Python能够使Python代码可以普遍被整个系统访问。最近在一些项目中，进一步改进和增强了打包、管理和部署Python代码的灵活性和实用性。

这些项目包括setuptools、Buildout和virtualenv。Buildout、setuptools和virtualenv通常与开发平台、开发库相关，并且需要处理开发环境参数。但是实际上，他们经常使用Python以与操作系统无关的方式来部署Python代码（注意，我们这里只是说大部分情况如此）。

另一个部署情况包括创建操作系统特定的软件包，部署软件包到终端用户主机。有时会出现两个完全不同的问题，尽管有一定程度的重叠性。我们将介绍一个开源的名为EPM的源码工具，可以为AIX、Debian/Ubuntu、FreeBSD、HP-UX、IRIX、Mac OS X、NetBSD、OpenBSD、Red Hat、Slackware、Solaris和Tru64 Unix产生本地平台的软件包。

包管理不仅对软件开发者有好处，对于系统管理员也是非常重要的。事实上，一名系统管理员通常会是包管理责无旁贷的负责人。如果你掌握了对Python和其他操作系统包管理的最新技术，你的身价会无限增加。本章会在这方面帮助你。这一章中涉及主题有一个非常有价值的参考，可以在以下地址找到：*http://wiki.python.org/moin/buildout/pycon2008_tutorial*。

www.TopSage.com

Setuptools和Python Egg

根据官方文档，“setuptools是一个对Python distutils的增强集合（Python2.3.5适于大多数平台，但64位平台需要Python2.4的最小化版本），允许你非常容易地建立和发布包，尤其是依赖于其他包的包。”

直到setuptools的出现，distutils一直是创建和安装Python包的主要方式。setuptools是一个增强distutils的库。Eggs涉及最终对Python包和模块的捆绑，非常像一个*.rpm*或*.deb*文件。它们通常以zip格式发布，能以zip格式进行安装或是使用unzip对包的内容进行浏览。Eggs是setuptools库的特色，与easy_install一起工作。根据官方文献描述“简易安装是一个python模块（easy_install），与setuptools捆绑在一起，允许自动下载，创建，安装和管理Python包”。easy_install是一个模块，因此经常被认为是命令行工具，并且以命令行工具的方式进行交互。在这一节中，我们介绍并解析setptools、easy_install和eggs，并且澄清每一个容易混淆的问题。

在这一章中，将会概述我们认为是setuptools和easy_install最有意义的内容。如果希望获得它们的全部相关文档，你可以另行访问：*http://peak.telecommunity.com/DevCenter/setuptools*和*http://peak.telecommunity.com/DevCenter/EasyInstall*。

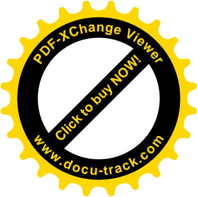
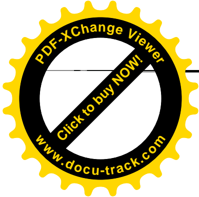
那些能够完成极其复杂事务的工具，通常是很难彻底理解的。setuptools的部分内容很难掌握，正是因为它能直接对一些复杂事务进行处理。这一节的介绍仅作为一个快速开始的指南，之后还会涉及手册中的内容，作为用户或是开发者，你应该能够掌握setuptools、easy_install和Python egg。

使用easy_install

理解和使用基本的easy_install非常容易。阅读本书的大多数人可能非常喜欢使用rpm、yum、apt-get、fink或是类似的包管理工具。短语“Easy Install”（简易安装）即为该命令行工具的名称，可以用来实现与Red Hat系统中yum和Debian系统中的apt-get相类似的工作，但是easy_install是专门为Python包服务的。

工具easy_install可以通过运行（使用希望easy_install协同工作的Python版本）一个名为ea_setup.py的“自举”（bootstrapp）脚本进行安装。

ea_setup.py获取setuptools的最新版本，并且自动安装easy_install。它将easy_install作为脚本安装到“scripts”目录中，对于*nixes系统默认是安装到与python二进制文件相同的目录中。让我们看一下这一操作是多么容易。参见例9-1。

例9-1：自举 easy_install

```
$ curl http://peak.telecommunity.com/dist/ez_setup.py
> ez_setup.py
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 9419 100 9419 0 0 606 0 0:00:15 0:00:15 --:--:-- 83353
$ ls
ez_setup.py
$ sudo python2.5 ez_setup.py
Password:
Searching for setuptools
Reading http://pypi.python.org/simple/setuptools/
Best match: setuptools 0.6c8
Processing setuptools-0.6c8-py2.5.egg
setuptools 0.6c8 is already the active version in easy-install.pth
Installing easy_install script to /usr/local/bin
Installing easy_install-2.5 script to /usr/local/bin

Using /Library/Python/2.5/site-packages/setuptools-0.6c8-py2.5.egg
Processing dependencies for setuptools
Finished processing dependencies for setuptools
$
```

在这种情况下，easy_install以两个不同的名字保存到*/usr/local/bin*目录中。

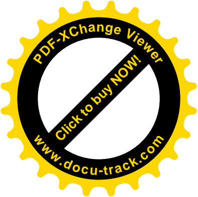
```
$ ls -l /usr/local/bin/easy_install*
-rwxr-xr-x  1 root  wheel  364 Mar  9 18:14 /usr/local/bin/easy_install
-rwxr-xr-x  1 root  wheel  372 Mar  9 18:14 /usr/local/bin/easy_install-2.5
```

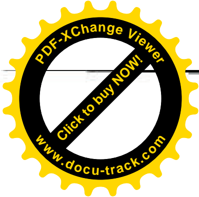
这是Python自身的约定，而且这一约定已经使用一段时间了：安装时，可以指明版本号，以表示使用的Python版本，也可以不指定版本号。当用户没有明确地指出脚本的版本时，未指明版本号的版本会被默认使用，同时这也表示最新安装的版本是默认版本。这一约定非常方便，旧版本仍可以保留下来继续使用。以下是刚刚安装的*/usr/local/bin/easy_install*的内容：

```
#!/System/Library/Frameworks/Python.framework/Versions/2.5/Resources/Python.app/
Contents/MacOS/Python
# EASY-INSTALL-ENTRY-SCRIPT: 'setuptools==0.6c8','console_scripts','easy_install'
__requires__ = 'setuptools==0.6c8'
import sys
from pkg_resources import load_entry_point

sys.exit(
load_entry_point('setuptools==0.6c8', 'console_scripts', 'easy_install')()
)
```

这里首先需要注意的是当你安装setuptools时，将安装名为easy_install的脚本，这样你可以使用它来安装和管理Python代码。我们通过显示easy_install脚本的内容向你展示第二个需要注意的问题，即这是一类脚本，这类脚本会在你定义包时在使用“进入点”

www.TopSage.com





(entrypoint）的时候自动创建。不要对这个脚本的内容或是进入点、或是如何创建这样的脚本心存担忧。我们会在本章的后面部分进行介绍。现在，有了easy_install，我们可以为上传的Python模块安装位于中心库的任何包，通常会涉及PyPI（Python PackageIndex），或者“Cheeseshop”：*http://pypi.python.org/pypi*。

为了安装IPython，在这个示例中专门使用shell，可以发出这样的命令：

```
sudo easy_install ipython
```

值得注意的是，当easy_install安装包到Python的全局site-packages目录时，安装过程中需要具有sudo权限。easy_install也可以放置脚本到操作系统的默认脚本目录，这也是python可以执行文件放置的目录。一般来讲，easy_install在安装一个包时需要有对*site-packages*和Python安装的脚本目录进行写入操作的权限。如果对此有疑问，可以参考本章中介绍使用virtualenv和setuptools的部分。另一种可选操作是，可以编译并安装Python到你自己的一个目录中，例如你的home目录。

在我们开始介绍easy_install工具的高级用途前，这里有一个easy_install基本用法的简要总结：

1. 下载ea_setup.py启动脚本；
2. 使用你希望的Python版本运行ez_setup.py来安装包；
3. 如果有Python的多个版本运行在你的系统上，明确地指定python的版本来运行easy_install。

名人简介：EASY INSTALL

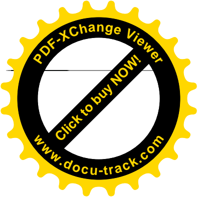
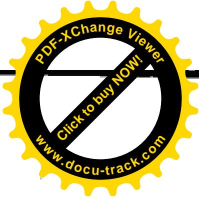
Phillip J. Eby

Phillip J. Eby已经负责多项Python增强建议，还负责WSGI标准，setuptools等。在《Dreaming in Code》（Three Rivers出版）中对其有专门介绍。你可以阅读他的编程博客：*http://dirtsimple.org/programming/*。

easy_install的高级特征

对于大多数临时使用easy_install的用户，只传递一个命令行参数，而不需要任何其他选项就可以满足他们的所有需要。（顺便说一下，给easy_install指定一个参数，一个

www.TopSage.com

包名，可以简单地下载并安装包，就像之前在IPython示例中演示的一样）。对于一些复杂问题，不是仅从Python Package Index下载eggs就能处理的，需要具有更强大的功能。幸运的是，easy_install还有一些非常灵活的技巧，可以对高级事务进行全面的分类处理。

在Web页面上搜索包

正如我们之前看到的，easy_install可以自动搜索包的中心仓库，并且自动进行安装。它可以按你可以想到的任何方式完成包的安装。以下是一个示例，显示了如何搜索web页面，并根据名称和版本号来安装或升级包。

```
$ easy_install -f http://code.google.com/p/liten/ liten
Searching for liten
Reading http://code.google.com/p/liten/
Best match: liten 0.1.3
Downloading http://liten.googlecode.com/files/liten-0.1.3-py2.4.egg
[snip]
```

在这种情况下，可以在*http://code.google.com/p/liten/*获得Python2.4 egg和Python2.5 egg。“easy_install -f”指定了搜索eggs的位置。搜索过程中发现了两个eggs，但安装Python2.4 egg，因为它是最好的匹配。很明显，这一点非常有用，因为easy_install不仅找到egg的链接，也找到了egg正确的版本。

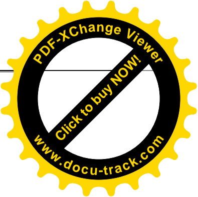
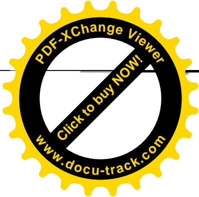
从URL安装源发布

现在，我们从URL自动安装了一个源码发布版本：

```
% easy_install http://superb-west.dl.sourceforge.net/sourceforge
      /sqlalchemy/SQLAlchemy-0.4.3.tar.gz

Downloading http://superb-west.dl.sourceforge.net/sourceforge
/sqlalchemy/SQLAlchemy-0.4.3.tar.gz
Processing SQLAlchemy-0.4.3.tar.gz
Running SQLAlchemy-0.4.3/setup.py -q bdist_egg --dist-dir
  /var/folders/LZ/LZFo5h8JEW4Jzr+ydkXfI++++TI/-Tmp-/
  easy_install-Gw2Xq3/SQLAlchemy-0.4.3/egg-dist-tmp-Mf4jir
zip_safe flag not set; analyzing archive contents...
sqlalchemy.util: module MAY be using inspect.stack
sqlalchemy.databases.mysql: module MAY be using inspect.stack
Adding SQLAlchemy 0.4.3 to easy-install.pth file
Installed /Users/ngift/src/py24ENV/lib/python2.4/site-packages/SQLAlchemy-0.4.3-
py2.4.egg
Processing dependencies for SQLAlchemy==0.4.3
Finished processing dependencies for SQLAlchemy==0.4.3
```

我们将用gzip压缩的tar包的URL传递给easy_install。easy_install能够判断出应该安装源码发布，而无须明确地告诉它怎样做。这是一个技巧，但是必需在根一级目录中包

www.TopSage.com

括*setup.py*文件才能起作用。例如，如果有人将它的包通过递归深埋在多层空目录中，这会失败。

在本地或网络文件系统中安装egg

以下是一个示例，演示了如何在安装文件系统或加载了NFS的存储器上安装egg：

```
easy_install /net/src/eggs/convertWindowsToMacOperatingSystem-py2.5.egg
```

可以在加载了NFS的目录或是一个本地分区中安装egg。这可以非常有效地在*nix环境中发布包，尤其是通过一系列你希望它们能够根据正在运行的代码版本保持彼此同步的主机。本书中还有一些脚本，可能帮助创建一个执行轮流检测的守护进程（polling）。每一个客户端可以运行这样一个守护进程来检测是否需要升级到eggs的中心库。如果有一个新的版本，那么它可以自动升级。

升级包

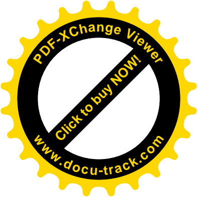
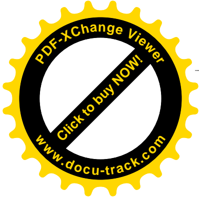
另外一种使用easy_install的方式是通过它来升级包。在接下来的一些示例中，将安装并升级CherryPy包。首先，安装CherryPy2.2.1版本：

```
$ easy_install cherrypy==2.2.1
Searching for cherrypy==2.2.1
Reading http://pypi.python.org/simple/cherrypy/
....
Best match: CherryPy 2.2.1
Downloading http://download.cherrypy.org/cherrypy/2.2.1/CherryPy-2.2.1.tar.gz
....
Processing dependencies for cherrypy==2.2.1
Finished processing dependencies for cherrypy==2.2.1
```

现在，我们为你显示了使用easy_install来安装一些已经安装过的东西时会出现的情况：

```
$ easy_install cherrypy
Searching for cherrypy
Best match: CherryPy 2.2.1
Processing CherryPy-2.2.1-py2.5.egg
CherryPy 2.2.1 is already the active version in easy-install.pth

Using /Users/jmjones/python/cherrypy/lib/python2.5/site-packages/CherryPy-2.2.1-py2.5.egg
Processing dependencies for cherrypy
Finished processing dependencies for cherrypy
```

在你安装了一些版本的包之后，你可以通过明确地声明需要下载并安装的版本，来将一些包升级到新的版本：

```
$ easy_install cherrypy==2.3.0 Searching for
cherrypy==2.3.0
Reading http://pypi.python.org/simple/cherrypy/
....
Best match: CherryPy 2.3.0
Downloading http://download.cherrypy.org/cherrypy/2.3.0/CherryPy-2.3.0.zip
....
Processing dependencies for cherrypy==2.3.0
Finished processing dependencies for cherrypy==2.3.0
```

值得注意的是，在这个示例中没有使用“--upgrade”标志。如果你已经将一些版本的包进行了安装，并且希望升级到该包的最新版本，可以使用“--upgrade”标志。

接下来，使用--upgrade标志升级到CherryPy3.0.0。这里，标志“--upgrade”实际上是可以省略的：

```
$ easy_install --upgrade cherrypy==3.0.0
Searching for cherrypy==3.0.0
Reading http://pypi.python.org/simple/cherrypy/
....
Best match: CherryPy 3.0.0
Downloading http://download.cherrypy.org/cherrypy/3.0.0/CherryPy-3.0.0.zip
....
Processing dependencies for cherrypy==3.0.0
Finished processing dependencies for cherrypy==3.0.0
```

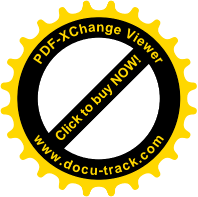
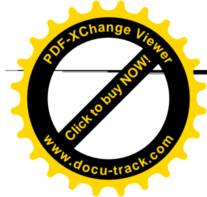
使用“--update”标志，但没有指定版本，将会把包升级到最新的版本。值得注意的是，这与使用“easy_install cherrypy”不同。使用“easy_install cherrypy”，如果CherryPy包的一些版本已经存在，则不会执行任何动作。在接下来的示例中，CherryPy会升级到当前最新的版本：

```
$ easy_install --upgrade cherrypy
Searching for cherrypy
Reading http://pypi.python.org/simple/cherrypy/
....
Best match: CherryPy 3.1.0beta3
Downloading http://download.cherrypy.org/cherrypy/3.1.0beta3/CherryPy-3.1.0beta3.zip
....
Processing dependencies for cherrypy
Finished processing dependencies for cherrypy
```

现在，CherryPy的版本是3.2.0b3。如果我们指定升级到高于3.0.0的版本，因为该版本已经存在，所以实际上系统什么也不会做。

```
$ easy_install --upgrade cherrypy>3.0.0
$
```


www.TopSage.com

在当前工作目录下安装一个已解包的源码发布

尽管这看起来有些锁碎，但它可能是最有用的。不同于使用python setup.py安装例程，可以仅输入下面的内容（少输入一些需要键入的字母，这对于懒人是个不错的技巧）：

```
easy_install
```

提取源码发布到指定的目录

你可以使用下面的示例来查找一个源码发布版或是检查包的URL，然后将其提取到一个指定的目录并进行检测：

```
easy_install --editable --build-directory ~/sandbox liten
```

因为允许easy_install使用一个源码发布，并且将其放到指定的目录中，所以这是非常方便的。由于使用easy_install进行包安装不是总能完成所有的安装（例如文档或代码示例可能无法安装），这是一个查看源码发布所包含内容的好方法。easy_install可以仅下载源码包，如果你需要对包进行安装，需要再次运行easy_install。

修改包的活动版本

这个示例假设你有版本为0.1.3的liten和一些其他的liten版本，并且已经安装。假设其他版本是“活动版本”。以下示例演示了如何再次激活0.1.3版本：

```
easy_install liten=0.1.3
```

如果你需要返回到一个旧版本或是需要返回到一个更新的当前版本，也是可以使用的。

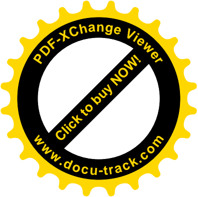
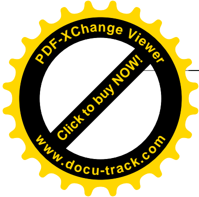
修改独立的.py文件到egg

以下是你如何转换一个普通的独立Python包到一个egg（注意，其中使用了“-f”标志）：

```
easy_install -f "http://svn.colorstudy.com/virtualenv/
    trunk/virtualenv.py#egg=virtualenv-1.0" virtualenv
```

当你希望一个单独的*.py*文件成为一个egg时，这是非常有用的。如果你有时需要使用一个之前被解包的单独的Python文件，使用这个方法是最好的选择。其他可选方法是，希望使用某个单独的模块时，在PYTHONPATH中进行设置。在这个示例中，我们打包项目中的virtualenv.py脚本，并在其中加入我们自己的版本号、名称标签。在URL字符串中，“#egg=virtualenv-1.0”只是简单地定义包的名称和我们选择给这个脚本定义的版本号。在URL字符串之后，给定的参数是我们寻找的包名。在URL字符串和独立的包

www.TopSage.com

名参数之前使用一致的名称是有好处的，因为我们告诉easy_install使用刚刚创建的相同的名称来完成包安装。将两者保持同步是有好处的，你不会感到需要将包名与模块名进行同步的限制。例如：

```
easy_install -f "http://svn.colorstudy.com/virtualenv/
        trunk/virtualenv.py#egg=foofoo-1.0" foofoo
```

除了这里创建了一个名为foofoo的名称，而不是vitualenv，除此以外这与之前的示例几乎完全相同。包的类型名称的选择完全取决于你自己。

认证一个密码保护的站点

在允许从一个需要认证的web网站下载文件之前，或许需要安装egg。在这种情况下，你可以为URL使用以下的语法来指定用户名和密码：

```
easy_install -f http://uid:passwd@example.com/packages
```

你或许正开发一个秘密的项目，不希望你的同事知道其中的细节。一种方法是创建一个简单的*.htaccess*文件，然后告诉easy_install做一个认证升级。

使用配置文件

对于高级用户，easy_install还有另一个技巧。你可以通过使用具有.ini语法格式的配置文件指定默认的选项。对于系统管理员，这是一个非常有用的功能，因为它允许easy_install的客户端声明配置。

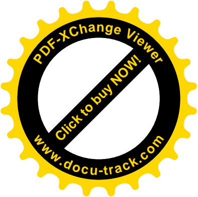
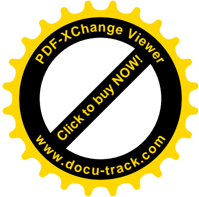
easy_install按下列位置顺序查找配置文件：*current_working_directory/setup.cfg*，*~/.pydistutils.cfg*，以及在distutils包目录中的*distutils.cfg*。

那么你可以在配置文件中放些什么呢？两个最常用的设置项是，默认的用于包下载的内部网站地址和一个自定义的安装包的目录。以下是一个简单的easy_install配置文件的内容：

```
[easy_install]

#Where to look for packages
find_links = http://code.example.com/downloads

#Restrict searches to these domains
allow_hosts = *.example.com

#Where to install packages. Note, this directory has to be on the PYTHONPATH
install_dir = /src/lib/python
```

在我们可以称之为*~/.pydistutils.cfg*的配置文件中，定义了一个特定的搜索包的URL，仅

www.TopSage.com

允许来自example.com（及其子域）的包搜索请求，并且最终将包放入到一个自定义的python包目录中。

easy install的高级特征总结

这里讲述的内容并不能取代广泛使用的easy_install官方文档，我们只是为了满足一些高级用户的需要而对该工具的高级关键特征进行重点强调。因为easy_install仍是一个正在持续开发的工具，所以经常检测*http://peak.telecommunity.com/DevCenter/EasyInstall*来升级文档是一个好主意。该地址也有一个称为distutils-sig的邮件列表（sig表示特殊兴趣组），讨论所有与Python发布相关的事宜。登录*http://mail.python.org/mailman/listinfo/distutils-sig*，可以报告bug并获得easy_install的相关帮助。

最后，通过简单地执行"easy_install --help"，你会发现更多的选项，许多是我们没有介绍的。或许你希望做的一些事情已经包括在easy_install的特征当中，这将是非常不错的。

创建egg

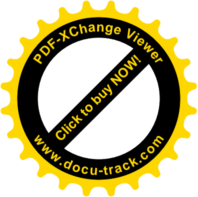
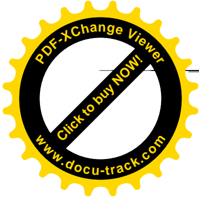
之前谈论过，一个egg是一些Python模块的集合，但是还没有给出一个更好的定义。这里是一个来自setuptools网站的对egg的定义：

> Python egg是EasyInstall首选的二进制发布格式，因为它跨平台（对于纯粹的包）、可以直接载入，且包括项目元数据（包括脚本及项目依赖的相关信息）。可以简单地下载并直接添加到sys.path中，或是放到sys.path的目录中，然后通过egg运行时系统自动发现。

为什么一名系统管理员会对创建eggs感兴趣，我们确实不能给出任何原因。如果你所需要做的事情是写一个一次性的脚本，那么egg对你不会有太多帮助。但是如果你发现一些模式或常规任务是需要频繁使用的，egg可以帮你省去一大堆麻烦。如果你创建一个供自己使用的小型通用任务库，可以将其捆绑为一个egg。如果这样做，你不仅通过代码复用节省了编写代码的时间，也使其在多台主机上更容易安装。

创建Python egg的过程极为简单，真正涉及的内容仅有四步：

1. 安装setuptools；
2. 创建希望在egg中出现的文件；
3. 创建*setup.py*文件；
4. 运行。



PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

```
python setup.py bdist_egg
```

我们已经将stuptools安装完毕，接下来我们继续创建希望在egg中出现的文件：

```
$ cd /tmp
$ mkdir egg-example
$ cd egg-example
$ touch hello-egg.py
```

在这个示例中，仅包含一个空的名为hello-egg.py的Python模块。

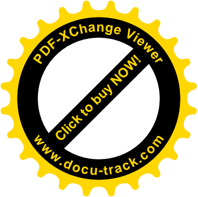
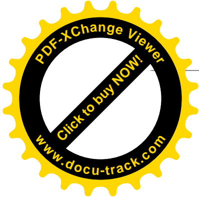
接下来创建*setup.py*文件，该文件或许是最简单的：

```
from setuptools import setup, find_packages
setup(
    name = "HelloWorld",
    version = "0.1",
    packages = find_packages(),
)
```

现在，我们可以创建egg：

```
$ python setup.py bdist_egg
running bdist_egg
running egg_info
creating HelloWorld.egg-info
writing HelloWorld.egg-info/PKG-INFO
writing top-level names to HelloWorld.egg-info/top_level.txt
writing dependency_links to HelloWorld.egg-info/dependency_links.txt
writing manifest file 'HelloWorld.egg-info/SOURCES.txt'
reading manifest file 'HelloWorld.egg-info/SOURCES.txt'
writing manifest file 'HelloWorld.egg-info/SOURCES.txt'
installing library code to build/bdist.macosx-10.5-i386/egg
running install_lib
warning: install_lib: 'build/lib' does not exist -- no Python modules to install
creating build
creating build/bdist.macosx-10.5-i386
creating build/bdist.macosx-10.5-i386/egg
creating build/bdist.macosx-10.5-i386/egg/EGG-INFO
copying HelloWorld.egg-info/PKG-INFO -> build/bdist.macosx-10.5-i386/egg/EGG-INFO
copying HelloWorld.egg-info/SOURCES.txt -> build/bdist.macosx-10.5-i386/egg/EGG-INFO
copying HelloWorld.egg-info/dependency_links.txt -> build/bdist.macosx-10.5-i386/egg/
EGG-INFO
copying HelloWorld.egg-info/top_level.txt -> build/bdist.macosx-10.5-i386/egg/EGG-
INFO
zip_safe flag not set; analyzing archive contents...
creating dist
creating 'dist/HelloWorld-0.1-py2.5.egg' and adding 'build/bdist.macosx-10.5-i386/
egg' to it
removing 'build/bdist.macosx-10.5-i386/egg' (and everything under it)
$ ll
total 8
drwxr-xr-x  6 ngift  wheel  204 Mar 10 06:53 HelloWorld.egg-info
```


www.TopSage.com

```
drwxr-xr-x  3 ngift  wheel  102 Mar 10 06:53 build
drwxr-xr-x  3 ngift  wheel  102 Mar 10 06:53 dist
-rw-r--r--  1 ngift  wheel    0 Mar 10 06:50 hello-egg.py
-rw-r--r--  1 ngift  wheel  131 Mar 10 06:52 setup.py
```

安装egg：

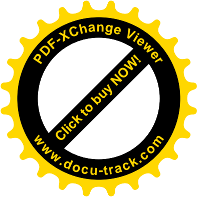
```
$ sudo easy_install HelloWorld-0.1-py2.5.egg
sudo easy_install HelloWorld-0.1-py2.5.egg
Password:
Processing HelloWorld-0.1-py2.5.egg
Removing /Library/Python/2.5/site-packages/HelloWorld-0.1-py2.5.egg
Copying HelloWorld-0.1-py2.5.egg to /Library/Python/2.5/site-packages
Adding HelloWorld 0.1 to easy-install.pth file

Installed /Library/Python/2.5/site-packages/HelloWorld-0.1-py2.5.egg
Processing dependencies for HelloWorld==0.1
Finished processing dependencies for HelloWorld==0.1
```

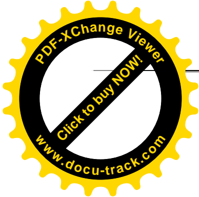
正如你所看到的，创建一个egg非常简单。因为这里的egg实际上是一个空文件，接下来，我们更详细地介绍如何创建一个Python脚本并且构建一个egg。

以下是一个简单的Python脚本，显示目录中的符号链接文件，其相应的实际文件位置，以及实际文件是否存在：

```
#!/usr/bin/env python

import os
import sys

def get_dir_tuple(filename, directory):
    abspath = os.path.join(directory, filename)
    realpath = os.path.realpath(abspath)
    exists = os.path.exists(abspath)
    return (filename, realpath, exists)

def get_links(directory):
    file_list = [get_dir_tuple(f, directory) for f in os.listdir(directory)
            if os.path.islink(os.path.join(directory, f))]
    return file_list

def main():
    if not len(sys.argv) == 2:
        print 'USAGE: %s directory' % sys.argv[0]
        sys.exit(1)
    directory = sys.argv[1]
    print get_links(directory)

if __name__ == '__main__':
    main()
```

接下来，使用setuptools创建一个setup.py。与之前示例相比，这是另一个最小化的*setup.py*文件：





```
from setuptools import setup, find_packages
setup(
    name = "symlinkator",
    version = "0.1",
    packages = find_packages(),
    entry_points = {
        'console_scripts': [
            'linkator = symlinkator.symlinkator:main',
        ],
    },
)
```

这里声明的包名为symlinkator，版本是0.1，并且setuptools会尽力查询适合的Python文件进行包含。这里暂时忽略entry_points部分。

现在，通过运行“python setup.py bdist_egg”创建egg：

```
$ python setup.py bdist_egg
running bdist_egg
running egg_info
creating symlinkator.egg-info
writing symlinkator.egg-info/PKG-INFO
writing top-level names to symlinkator.egg-info/top_level.txt
writing dependency_links to symlinkator.egg-info/dependency_links.txt
writing manifest file 'symlinkator.egg-info/SOURCES.txt'
writing manifest file 'symlinkator.egg-info/SOURCES.txt'
installing library code to build/bdist.linux-x86_64/egg
running install_lib
warning: install_lib: 'build/lib' does not exist -- no Python modules to install
creating build
creating build/bdist.linux-x86_64
creating build/bdist.linux-x86_64/egg
creating build/bdist.linux-x86_64/egg/EGG-INFO
copying symlinkator.egg-info/PKG-INFO -> build/bdist.linux-x86_64/egg/EGG-INFO
copying symlinkator.egg-info/SOURCES.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying symlinkator.egg-info/dependency_links.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying symlinkator.egg-info/top_level.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
zip_safe flag not set; analyzing archive contents...
creating dist
creating 'dist/symlinkator-0.1-py2.5.egg' and adding 'build/bdist.linux-x86_64/egg' to it
removing 'build/bdist.linux-x86_64/egg' (and everything under it)
```

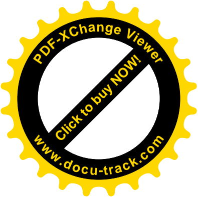
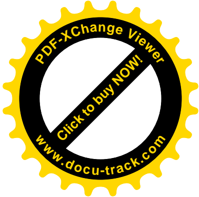
验证egg内容。进入创建的*dist*目录，验证其中包含了一个egg：

```
$ ls -l dist
total 4
-rw-r--r-- 1 jmjones jmjones 825 2008-05-03 15:34 symlinkator-0.1-py2.5.egg
```

现在安装egg：

```
$ easy_install dist/symlinkator-0.1-py2.5.egg
Processing symlinkator-0.1-py2.5.egg
....
```


www.TopSage.com

```
Processing dependencies for symlinkator==0.1
Finished processing dependencies for symlinkator==0.1
```

接下来，启动IPython，加载并使用我们的模块：

```
In [1]: from symlinkator.symlinkator import get_links

In [2]: get_links('/home/jmjones/logs/')
Out[2]: [('fetchmail.log.old', '/home/jmjones/logs/fetchmail.log.3', False),
('fetchmail.log', '/home/jmjones/logs/fetchmail.log.0', True)]
```

以下是运行get_links()函数的目录，这或许是你感兴趣的地方：

```
$ ls -l ~/logs/
total 0
lrwxrwxrwx 1 jmjones jmjones 15 2008-05-03 15:11 fetchmail.log -> fetchmail.log.0
-rw-r--r-- 1 jmjones jmjones  0 2008-05-03 15:09 fetchmail.log.0
-rw-r--r-- 1 jmjones jmjones  0 2008-05-03 15:09 fetchmail.log.1
lrwxrwxrwx 1 jmjones jmjones 15 2008-05-03 15:11 fetchmail.log.old -> fetchmail.log.3
```

进入点及控制台脚本

以下是setuptools的文档页：

> 进入点用来支持服务的动态发现或是项目提供的插件。查看Dynamic Discovery of Services and Plugins（动态服务发现和插件）可以获得详细说明及参数格式的示例。此外，这个关键字还用来支持Automatic ScriptCreation（自动脚本创建）。

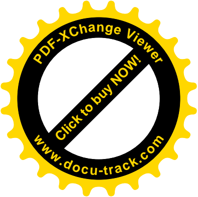
在这本书中介绍的进入点类型，是控制台脚本变量。仅需要给定一些信息，setuptools会自动创建一个控制台脚本，指定的信息可以放在*setup.py*文件中。以下是来自之前一示例*setup.py*中的相关内容：

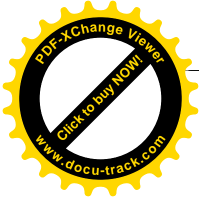
```
entry_points = {
    'console_scripts': [
        'linkator = symlinkator.symlinkator:main',
    ],
},
```

在这个示例中，我们希望有一个名为linkator的脚本，并且当脚本被执行时，我们希望它调用symlinkator.symlinkator模块中的main()函数。当安装了egg之后，这个linkator脚本及python二进制文件都被放置到相同目录下：

```
#!/home/jmjones/local/python/scratch/bin/python
# EASY-INSTALL-ENTRY-SCRIPT: 'symlinkator==0.1','console_scripts','linkator'
__requires__ = 'symlinkator==0.1'
import sys
from pkg_resources import load_entry_point
```

www.TopSage.com


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

```
sys.exit(
    load_entry_point('symlinkator==0.1', 'console_scripts', 'linkator')()
)
```

你看到的所有结果都是由*setuptools*产生。理解该脚本中的所有代码的含义并不重要。实际上，理解脚本中的任何代码可能都不重要，重要的是要知道当你在setup.py中定义一个console_scripts进入点时，setuptools会创建一个脚本，该脚本可以调用你的代码到指定的地方。对比模式调用前一示例中的脚本时，结果如下：

```
$ linkator ~/logs/
[('fetchmail.log.old', '/home/jmjones/logs/fetchmail.log.3', False),
('fetchmail.log', '/home/jmjones/logs/fetchmail.log.0', True)]
```

理解进入点会有一些复杂，但站在更高的层次上来看，唯一需要知道的是，可以使用进入点来安装脚本，它可以作为用户路径上的命令行工具来使用。为了这样做，仅需要遵循上面列出的语法，并且定义一个可以运行命令行工具的函数。

使用Python包索引注册一个包

如果你编写了一个非常好的工具或是模块，很自然地，你会希望与其他人共享你的成果。这是开源软件开发的一个最吸引人的地方。庆幸的是，上传包到Python Package Index（Python包索引）是非常简单的过程。

这个过程仅与创建egg稍有不同。有两件事情需要引起注意，第一件是记住在long_description中包括ReST，reStructuredText以及在long_description中的格式化描述，第二件是指定download_url的值。我们在第4章已经对ReST格式进行了介绍。

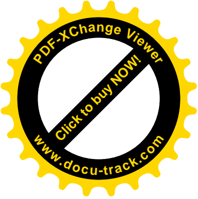
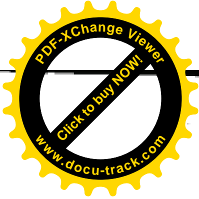
尽管之前讨论了ReST格式，在这里应该再强调一下，以ReST来格式化文档是一个好想法，因为在上传到cheeseshop时，它可以转化为HTML。可以使用Aaron Hillegass创建的工具ReSTless来预览格式化文本，以确保当你预览时文档能被正确地格式化。如果你没有正确地格式化成ReST，当你上传文档的时候，文本将显示成纯文本，而不是HTML。

参阅例9-2，查看为命令行工具使用的setup.py以及由Noah创建的库：

例9-2：用于上传到Python Package Index的简单的setup.py

```
#!/usr/bin/env python

# liten 0.1.4.2 -- deduplication command-line tool
#
# Author: Noah Gift
try:
    from setuptools import setup, find_packages
except ImportError:
    from ez_setup import use_setuptools
```


www.TopSage.com

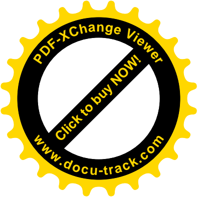
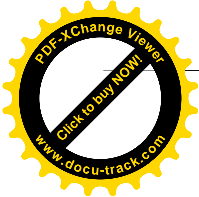



```
    use_setuptools()
    from setuptools import setup, find_packages
import os,sys

version = '0.1.4.2'
f = open(os.path.join(os.path.dirname(__file__), 'docs', 'index.txt'))
long_description = f.read().strip()
f.close()

setup(

      name='liten',
      version='0.1.4.2',
      description='a de-duplication command line tool',
      long_description=long_description,
      classifiers=[
              'Development Status :: 4 - Beta',
              'Intended Audience :: Developers',
              'License :: OSI Approved :: MIT License',
          ],
      author='Noah Gift',
      author_email='noah.gift@gmail.com',
      url='http://pypi.python.org/pypi/liten',
      download_url="http://code.google.com/p/liten/downloads/list",
      license='MIT',
      py_modules=['virtualenv'],
      zip_safe=False,
      py_modules=['liten'],
      entry_points="""
      [console_scripts]
      liten = liten:main
      """,
      )
```

通过*setup.py*文件，现在可以通过使用下面的命令自动在Python Package Index中注册一个包：

```
$ python setup.py register
running register
running egg_info
writing liten.egg-info/PKG-INFO
writing top-level names to liten.egg-info/top_level.txt
writing dependency_links to liten.egg-info/dependency_links.txt
writing entry points to liten.egg-info/entry_points.txt
writing manifest file 'liten.egg-info/SOURCES.txt'
Using PyPI login from /Users/ngift/.pypirc
Server response (200): OK
```

与symlinkator示例相比，setup.py中添加了一些额外的字段。这些额外字段包括description、long_description、classifiers、author和download_url。正如之前介绍的，进入点允许从命令行运行工具并且安装到默认的脚本目录。download_url非常关键，因为它告诉easy_install到哪里搜索你的包。可以包含对页面的一个链接，而

easy_install是足够智能的，可以通过链接找到包或egg，但是也可以明确地创建一个到你创建的egg的链接。long_description复用保存在*/doc*相对目录下的文档，该目录在*index.txt*文件中指定。*index.txt*文件是按ReST进行格式化的，setup.py脚本读入信息，将其放到在Python Package Index注册的字段中。

我们可以从哪里学到更多东西...

以下是一些重要的资源：

Easy install

http://peak.telecommunity.com/DevCenter/EasyInstall

Python eggs

http://peak.telecommunity.com/DevCenter/PythonEggsThe setuptools module

setuptools模块

http://peak.telecommunity.com/DevCenter/setuptoolsThe package resources module

包资源模块

http://peak.telecommunity.com/DevCenter/PkgResources

Architectural overview of pkg_resources and Python eggs in general

Architectural Overview of pkg_resources and Python Eggs in General

不要忘记，还有Python邮件列表*http://mail.python.org/pipermail/distutilssig/*。

Distutils

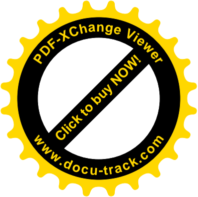
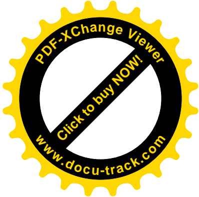
在写这一小节的时候，setuptools是创建包和向多个人进行发布的首选方法，并且看起来部分setuptools库被加入到标准库中是迟早的事情。这也就是说，知道distutils包是如何工作，setuptools在哪里得到了增强，仍是很重要的事情。当distutils用来创建发布包时，典型安装包的方法是这样的：

```
python setup.py
install
```

考虑到创建发布包，我们介绍以下四个主题：

- 如何写一个安装脚本，这里是文件*setup.py*；
- *setup.py*文件中的基本配置选项；
- 如何创建一个源码发布包；
- 创建二进制文件，例如，rpms, Solaris, pkgtool, 和HP-UX swinstall。

www.TopSage.com

演示distutils是如何工作的最好方式如下所示：

第一步：创建一些代码。让我们使用以下这个简单的脚本作为一个发布示例：

```
#!/usr/bin/env python
#A simple python script we will package
#Distutils Example. Version 0.1

class DistutilsClass(object):
    """This class prints out a statement about itself."""
    def __init__(self):
        print "Hello, I am a distutils distributed script." \
            "All I do is print this message."

if __name__ == '__main__':
    DistutilsClass()
```

第二步：在相同的目录下创建*setup.py*作为你的脚本。

```
#Installer for distutils example script

from distutils.core import setup

setup(name="distutils_example",
    version="0.1",
    description="A Completely Useless Script That Prints",
    author="Joe Blow",
    author_email = "joe.blow@pyatl.org",
    url = "http://www.pyatl.org")
```

值得注意的是，我们传递给setup()的多个关键参数，这些参数在之后可以通过元数据对包进行识别。请注意，这是一个非常简单的示例，其中使用了许多选项（例如处理多重依赖等）。这里不会进一步介绍高级配置，但是建议在官方的Python在线文档中阅读更多内容。

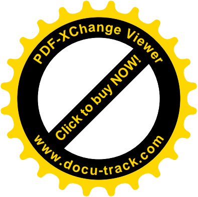
第三步：创建一个发布版。

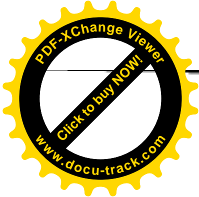
现在我们有了一个基本的setup.py脚本，通过运行这个命令（该命令与脚本、README和*setup.py*文件在相同的目录中），可以比较容易地创建一个源码发布包：

```
python setup.py sdist
```

你会看到下面的输出：

```
running sdist
    warning: sdist: manifest template 'MANIFEST.in' does not exist
      (using default file list)
    writing manifest file 'MANIFEST'
    creating distutils_example-0.1
    making hard links in distutils_example-0.1...
```





```
hard linking README.txt distutils_example-0.1
hard linking setup.py distutils_example-0.1
creating dist
tar -cf dist/distutils_example-0.1.tar distutils_example-0.1
gzip -f9 dist/distutils_example-0.1.tar
removing 'distutils_example-0.1' (and everything under it)
```

可以从输出结果中看出，现在只需要解包，然后安装，如下所示：

```
python setup.py install
```

如果你喜欢创建二进制包，这里有一些示例。需要注意的是，他们依赖底层的操作系统来完成重要工作，因此不能在OS X这样的系统上创建rpm包。由于有许多虚拟产品，因此这对你来说不应该是个问题。将一些虚拟机保持在你可以随时激活的状态，这样在你需要时就可以进行创建。

构建rpm：

```
python setup.py bdist_rpm
```

构建Solaris pkgtool：

```
python setup.py bdist_pkgtool
```

构建HP-UX swinstall：

```
python setup.py bdist_sdux
```

最后，当发布创建的包时，你会希望当你准备安装包时能够自行定义安装目录。通常创建和安装过程是一次性完成的，但是你或许会希望选择一个自定义的创建方式，就像下面这样：

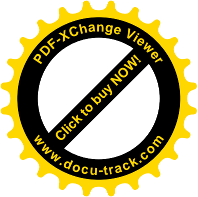
```
python setup.py build --build-base=/mnt/python_src/ascript.py
```

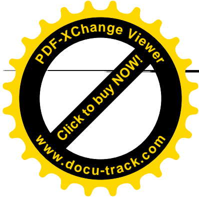
当你实际运行install命令时，它会复制*build*目录下的所有内容到安装目录。在Python环境中，默认的安装目录是*site-packages*，你可以在这个目录中执行命令，但是也可以指定一个自定义的安装目录，例如一个NFS挂载点，正如前一个示例所演示的一样。

Buildout

Buildout是由Zope公司的JimFulton开发的一个工具，可以管理创建的新应用程序。这些应用程序可以是Python程序或是其他程序，例如Apache。Buildout的一个主要目标是可以创建可复用的跨平台程序。JimFulton使用Buildout的经历来自部署Plone 3.x站点。在那之后，他意识到使用Buildout部署Plone 3.x站点仅仅是冰山的一角。

www.TopSage.com





Buildout是一个非常引人注目的新的包管理工具，且Python中提供支持。Buildout允许复杂的应用，这些复杂的应用如果有bootstrap.py和config文件存在，对自举会有复杂的依赖。在接下来的一节，我们将讨论分为两个部分：使用Buildout和使用Buildout进行开发。也建议你阅读Buildout手册（*http://pypi.python.org/pypi/zc.buildout*），可以获得Buildout最新信息。事实上，这个文档与你获得的Buildout文档一样全面，对于一名Buildout用户必须进行阅读。

> 名人简介：BUILDOUT
>
> **Jim Fulton**
>
> Jim Fulton是Zope Object Database的创建者与维护者之一。Jim同时也是Zope Object Publishing Environment的创建者，以及Zope公司的CTO。

使用Buildout

尽管许多使用Zope技术的人会注意到Buildout，但其对于Python世界仍是一个秘密。Buildout是一个被推荐的机制，Plone就是通过该机制进行部署的。可能你还不熟悉Plone，Plone是一个企业级的内容管理系统，其后有巨大的开发社区支持。在Buildout出现之前，安装Plone是极度复杂的。现在，Buildout使得Plone的安装非常简便。

可以使用Buildout来管理Python环境，这是许多人还不知道的事情。Buildout是一个非常智能的软件，使用时仅需要你做两件事情：

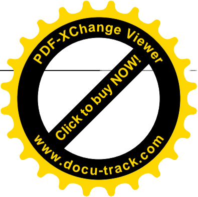
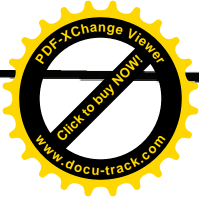
- 最新的bootstrap.py的副本。你可以在以下地址进行下载：*http://svn.zope.org/*checkout*/zc.buildout/trunk/bootstrap/bootstrap.py*。
- 具有"recipes"或"eggs"项的*buildout.cfg*文件来进行安装。

演示Buildout最好的方法是使用它来进行安装。Noah已经写了一个de-duplicate命令行工具，称为liten，它可从中心Python库中找到（PyPI）。我们将使用Buildout来启动一个Python环境，然后运行这个工具。

第一步：下载*bootstrap.py*脚本。

```
mkdir -p ~/src/buildout_demo
curl http://svn.zope.org/*checkout*/zc.buildout/trunk/
bootstrap/bootstrap.py > ~/src/buildout_demo/bootstrap.py
```

第二步：定义一个简单的*buildout.cfg*文件。正如之前提到的，Buildout需要*buildout.cfg*文件。如果不使用*buildout.cfg*文件运行bootstrap.py脚本，会得到以下的输出结果：

```
$ python bootstrap.py
While:
Initializing.
Error: Couldn't open /Users/ngift/src/buildout_demo/buildout.cfg
```

例如，我们在例9-3中创建了配置文件。

例9-3：创建配置文件示例

```
[buildout]
parts = mypython
[mypython]
recipe = zc.recipe.egg
interpreter = mypython
eggs = liten
```

如果我们以buildout.cfg保存这个文件，然后再次运行bootstrap.py脚本，会得到这样的输出结果如例9-4所示。

例9-4：测试buildout环境

```
$ python bootstrap.py
Creating directory '/Users/ngift/src/buildout_demo/bin'.
Creating directory '/Users/ngift/src/buildout_demo/parts'.
Creating directory '/Users/ngift/src/buildout_demo/eggs'.
Creating directory '/Users/ngift/src/buildout_demo/develop-eggs'.
Generated script '/Users/ngift/src/buildout_demo/bin/buildout'.
```

如果打开新创建的目录，会发现可执行的、包含在*bin*目录中的一个自定义的Python解释器：

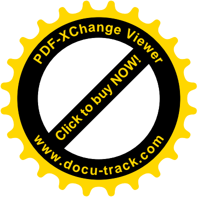
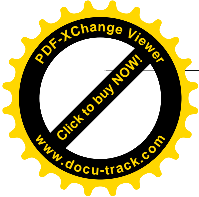
```
$ ls -l bin
total 24
-rwxr-xr-x  1 ngift  staff  362 Mar  4 22:17 buildout
-rwxr-xr-x  1 ngift  staff  651 Mar  4 22:23 mypython
```

现在，我们最终将Buildout工具安装完毕。可以运行它，前面定义的egg也会起作用。参见例9-5。

例9-5：运行 Buildout 并测试安装

```
$ bin/buildout
Getting distribution for 'zc.recipe.egg'.
Got zc.recipe.egg 1.0.0.
Installing mypython.
Getting distribution for 'liten'.
Got liten 0.1.3.
Generated script '/Users/ngift/src/buildout_demo/bin/liten'.
Generated interpreter '/Users/ngift/src/buildout_demo/bin/mypython'.
$ bin/mypython

>>>
$ ls -l bin
```


www.TopSage.com

```
total 24
-rwxr-xr-x  1 ngift  staff  362 Mar  4 22:17 buildout
-rwxr-xr-x  1 ngift  staff  258 Mar  4 22:23 liten
-rwxr-xr-x  1 ngift  staff  651 Mar  4 22:23 mypython
$ bin/mypython

>>> import liten
```

最后，因为liten与进入点一起被创建（进入点在本章前面部分已经介绍过），所以egg除了本地Buildout的*bin*目录中的模块外，能够自动安装一个控制台脚本。如果希望查看这一点，会看到接下来的输出结果：

```
$ bin/liten
Usage: liten [starting directory] [options]

A command-line tool for detecting duplicates using md5 checksums.

Options:
--version             show program's version number and exit
-h, --help            show this help message and exit
-c, --config          Path to read in config file
-s SIZE, --size=SIZE  File Size Example: 10bytes, 10KB, 10MB,10GB,10TB, or
plain number defaults to MB (1 = 1MB)
-q, --quiet           Suppresses all STDOUT.
-r REPORT, --report=REPORT
Path to store duplication report. Default CWD
-t, --test            Runs doctest.
$ pwd
/Users/ngift/src/buildout_demo
```

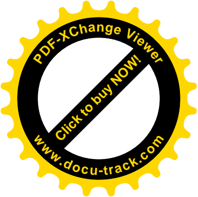
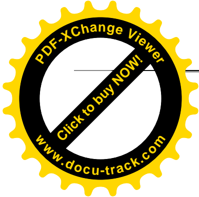
这是一个非常强大而简单的示例，演示了buidlout如何创建一个独立的环境并自动部署项目或环境的正确依赖。为了真正显示Buildout的强大之处，我们应该看一下Buildout的另一个方面。Buildout对运行的目录有完全的控制权，并且每次Buildout运行时，它会读取*buildout.cfg*文件来查找指令。这表示如果我们删除了列出的egg，它会有效删除命令行工具和库，参见例9-6。

例9-6：Buildout 配置文件

```
[buildout]
parts =
```

现在，这是egg和解释器被删除后的Buildout再次运行的情况。值得注意的是，Buildout有一些命令行选项，在这个示例中，使用“-N”，表示仅修改变化的文件。通常，Buildout会在它每次重运行时重新创建。

```
$ bin/buildout -N
Uninstalling mypython.
```

当我们查看*bin*目录时，发现解释器和命令行工具不见了。仅剩下实际的Buildout命令行工具：

```
$ ls -l bin/
total 8
-rwxr-xr-x 1 ngift staff 362 Mar 4 22:17 buildout
```

如果我们查看eggs目录，可以看到安装了egg但没有激活。但是，我们不能运行它，因为它没有解释器：

```
$ ls -l eggs
total 640
drwxr-xr-x  7 ngift  staff    238 Mar  4 22:54 liten-0.1.3-py2.5.egg
-rw-r--r--  1 ngift  staff 324858 Feb 16 23:47 setuptools-0.6c8-py2.5.egg
drwxr-xr-x  5 ngift  staff    170 Mar  4 22:17 zc.buildout-1.0.0-py2.5.egg
drwxr-xr-x  4 ngift  staff    136 Mar  4 22:23 zc.recipe.egg-1.0.0-py2.5.egg
```

使用Buildout进行开发

我们已经介绍了一个简单地创建和销毁Buildout控制环境的示例，我们现在进一步创建一个Buildout控制开发环境。

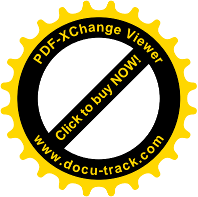
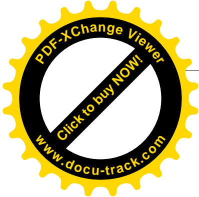
最一般的情况是，Buildout可以方便地使用。一个开发者或许是在一个独立的具有版本控制的开发包上进行开发。开发者检查项目顶级的*src*目录。在*src*目录中，具有一个类似下述配置文件，可以如同之前描述的那样运行Buildout：

```
[buildout]
develop = .
parts = test

[python]
recipe = zc.recipe.egg
interpreter = python
eggs = ${config:mypkgs}

[scripts]
recipe = zc.recipe.egg:scripts
eggs = ${config:mypkgs}

[test]
recipe = zc.recipe.testrunner
eggs = ${config:mypkgs}
```

virtualenv

根据Python Package Index页面的描述，“virtualenv是一个工具，可以创建独立的Python环境”。virtualenv解决的基本问题是消除了包冲突问题。通常会有这样的情况，某个工

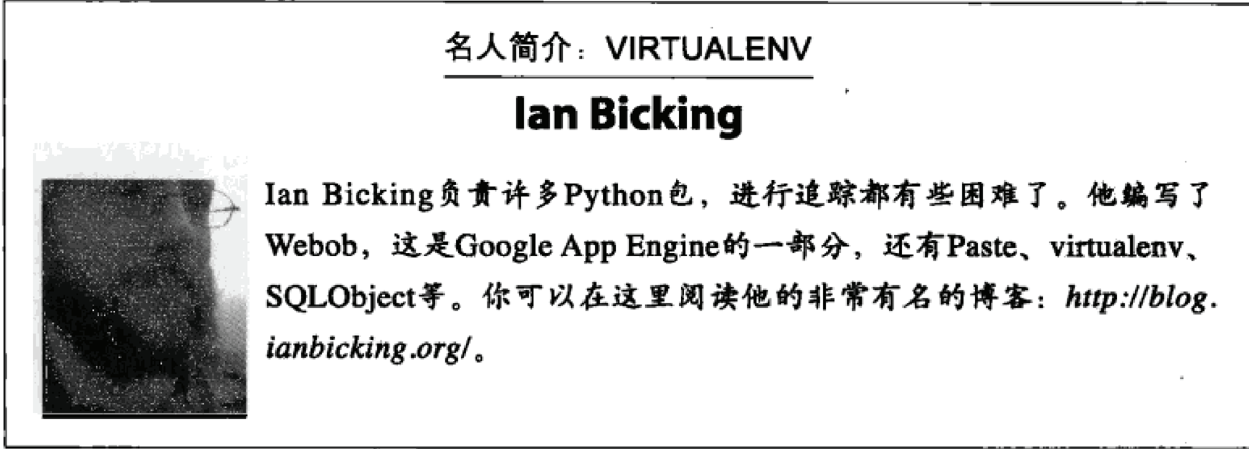
www.TopSage.com

具需要一个包版本，而另一个工具却需要另一个不同的包版本。这会产生一种危险的情况：因为一些人无意地修改全局*site-packages*目录，以希望通过升级包来运行一个不同的工具，一个web应用就很可能被破坏。

一种可选的方法是，一个开发者不具有对一个全局*sit-packages*目录的写权限，并且可以使用virtualenv来保持一个独立的、与系统Python相分离的virtualenv。virtualenv是一个消除之前诸多问题的非常好的方式，因为它允许创建新的发送箱，这或许彻底地与全局*site-packages*目录隔离开。

virtualenv允许开发者通过自定义的环境配置来启动一个虚拟环境。这与Buildout所做的工作是非常相似的，虽然Buildout使用一个声明的配置文件。应该注意的是，Buildout和virtualenv都扩展使用了setuptools，目前 setuptools的维护者是Phillip J. Eby。



名人简介：VIRTUALENV

Ian Bicking

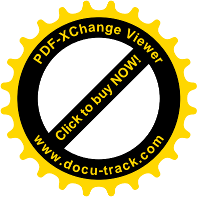
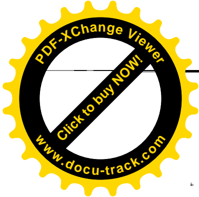
Ian Bicking负责许多Python包，进行追踪都有些困难了。他编写了Webob，这是Google App Engine的一部分，还有Paste、virtualenv、SQLObject等。你可以在这里阅读他的非常有名的博客：*http://blog.ianbicking.org/*。

因此，你如何使用virtualenv？最直接的方法是使用easy_install安装virtualenv：

```
sudo easy_install virtualenv
```

如果你计划仅通过单一版本的Python来使用virtualenv，这个方法非常有用。如果在你的主机上有许多已经安装的Python版本，例如Python 2.4、Python 2.5、Python 2.6或Python3000，它们会共享相同的*bin*主目录（例如*/usr/bin*），那么需要一个可选的方法才能使工作正常进行，因为一次只能有一个virtualenv脚本可以在相同的脚本目录中安装。一种方法是创建多个virtualenv脚本，可以与多个Python版本协同工作。该方法只需下载最新的virtualenv版本，并且创建一个对第一个Python版本的别名即可。以下是具体的操作的步骤：

1. curl *http://svn.colorstudy.com/virtualenv/trunk/virtualenv.py* > *virtualenv.py*；
2. sudo cp *virtualenv.py /usr/local/bin/virtualenv.py*；

3. 在你的Bash或zsh中创建两个别名：

```
alias virtualenv-py24="/usr/bin/python2.4 /usr/local/bin/virtualenv.py"
alias virtualenv-py25="/usr/bin/python2.5 /usr/local/bin/virtualenv.py"
alias virtualenv-py26="/usr/bin/python2.6 /usr/local/bin/virtualenv.py"
```

在多脚本环境背后，可以继续为每一个需要处理的Python版本创建多个virtualenv容器。以下是一个示例，演示了具体实现。

创建Python2.4虚拟环境：

```
$ virtualenv-py24 /tmp/sandbox/py24ENV
New python executable in /tmp/sandbox/py24ENV/bin/python
Installing setuptools.................done.
$ /tmp/sandbox/py24ENV/bin/python
Python 2.4.4 (#1, Dec 24 2007, 15:02:49)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
$ ls /tmp/sandbox/py24ENV/
bin/ lib/
$ ls /tmp/sandbox/py24ENV/bin/
activate             easy_install*      easy_install-2.4*  python*          python2.4@
```

创建Python2.5虚拟环境：

```
$ virtualenv-py25 /tmp/sandbox/py25ENV
New python executable in /tmp/sandbox/py25ENV/bin/python
Installing setuptools.........................done.
$ /tmp/sandbox/py25ENV/bin/python
Python 2.5.1 (r251:54863, Jan 17 2008, 19:35:17)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
$ ls /tmp/sandbox/py25ENV/
bin/ lib/
$ ls /tmp/sandbox/py25ENV/bin/
activate             easy_install*      easy_install-2.5*  python*          python2.5@
```

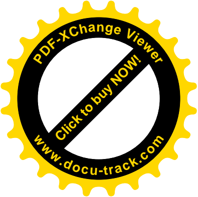
如果查看命令的输出结果，可以看到virtualenv创建了一个相对的`bin`目录和相对的*lib*目录。在`bin`目录中是Python解释器，使用*lib*目录作为它自己的本地*site-packages*目录。另一个明显的特点是easy_install脚本，允许通过easy_install将包安装到虚拟环境中。

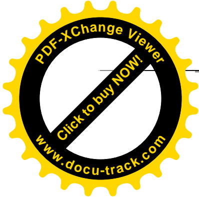
最后值得注意的是，有两种方法可以与你创建的虚拟环境协同工作。你可以明确地调用到虚拟环境的全路径：

```
$ /src/virtualenv-py24/bin/python2.4
```

另一个可选方法是，使用位于virtualenv中的*bin*目录下的活动脚本来设置环境，以使用sandbox，而不需要键入全路径。这是一个可以使用的可选工具，不是必须的，因为总

www.TopSage.com





是可以直接键入到virtualenv的全路径。Doug Hellmann，是这本书的评审者之一，他创造了一个非常智能的黑客工具，可以在这里找到相关信息：*http://www.doughellmann.com/projects/virtualenvwrapper*。该工具使用活动的Bash封闭器菜单来选择每次使用的sandbox。

创建一个自定义的自启动虚拟环境

发布的virtualenv1.0版本（撰写本书时的当前版本），包括对创建virtualenv环境的启动脚本的支持。实现该目标的一个方法是调用virtualenv.create_bootstrap_script(text)。它的作用是创建一个启动脚本，类似于virtualenv，但是具有额外的特征可以扩展选项parsing、adjust_options、以及使用after_install 钩子。

让我们看一下创建一个自定义的启动脚本有多么容易，该脚本会安装virtualenv以及一组自定义的egg到一个新的环境中。作为一个示例，我们回到liten包，使用virtualenv来创建一个新的虚拟环境并且使用liten。例9-7准确演示了如何创建一个自定义的启动脚本，使用该脚本可以安装liten。

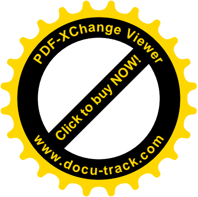
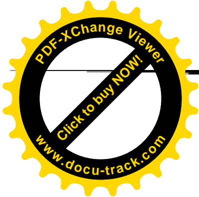
例9-7：创建启动脚本

```
import virtualenv, textwrap
output = virtualenv.create_bootstrap_script(textwrap.dedent("""
import os, subprocess
def after_install(options, home_dir):
    etc = join(home_dir, 'etc')
    if not os.path.exists(etc):
        os.makedirs(etc)
    subprocess.call([join(home_dir, 'bin', 'easy_install'),
                      'liten'])
"""))
f = open('liten-bootstrap.py', 'w').write(output)
```

这个示例由vitralenv文档改编而来，并且最后两行需要格外注意：

```
    subprocess.call([join(home_dir, 'bin', 'easy_install'),
    'liten'])
    """))
    f = open('liten-bootstrap.py', 'w').write(output)
```

从本质上说，以上代码告诉after_install函数在当前称为*liten-bootstrap.py*的工作目录中写一个新文件，然后加入一个自定义的模块liten的easy_install。值得注意的是，代码创建了一个*bootstrap.py*文件，并且这个*bootstrap.py*文件需要被运行。在执行这个脚本时，我们会有一个*liten-bootstrap.py*文件，该文件可以发布给开发者或是终端用户。如果我们不带任何参数执行*liten-bootstrap.py*，我们会得到下面的输出结果：

```
$ python liten-bootstrap.py
You must provide a DEST_DIR
Usage: liten-bootstrap.py [OPTIONS] DEST_DIR

Options:
--version             show program's version number and exit
-h, --help            show this help message and exit
-v, --verbose         Increase verbosity
-q, --quiet           Decrease verbosity
--clear               Clear out the non-root install and start from scratch
--no-site-packages    Don't give access to the global site-packages dir to the
                      virtual environment
```

实际上，当我们带有一个目标目录来运行这个工具时，我们会得到这样的输出结果：

```
$ python liten-bootstrap.py --no-site-packages /tmp/liten-ENV
New python executable in /tmp/liten-ENV/bin/python
Installing setuptools..........................done.
Searching for liten
Best match: liten 0.1.3
Processing liten-0.1.3-py2.5.egg
Adding liten 0.1.3 to easy-install.pth file
Installing liten script to /tmp/liten-ENV/bin

Using /Library/Python/2.5/site-packages/liten-0.1.3-py2.5.egg
Processing dependencies for liten
Finished processing dependencies for liten
```

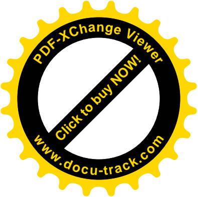
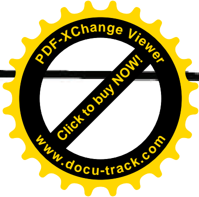
智能启动脚本自动地创建一个模块环境。这样，如果在liten工具中运行到virtualenv全路径，我们会得到下面的结果：

```
$ /tmp/liten-ENV/bin/liten
Usage: liten [starting directory] [options]

A command-line tool for detecting duplicates using md5 checksums.

Options:
--version               show program's version number and exit
-h, --help              show this help message and exit
-c, --config            Path to read in config file
-s SIZE, --size=SIZE    File Size Example: 10bytes, 10KB, 10MB,10GB,10TB, or
plain number defaults to MB (1 = 1MB)
-q, --quiet             Suppresses all STDOUT.
-r REPORT, --report=REPORT
Path to store duplication report. Default CWD
-t, --test              Runs doctest.
```

这是一个需要掌握的非常不错的技巧，允许创建一个完全独立的可启动的虚拟环境。

我们希望，通过这一节对virtualenv的介绍有一点非常明确，即使用和理解virtualenv是非常简单的。virtualenv遵循经典的KISS规则，单是这一个原因就足以考虑使用它来帮助管

www.TopSage.com

理独立的开发环境。如果你有更多的疑问，可以访问virtualenv邮件列表：*http://groups.google.com/group/python-virtualenv/*。

EPM包管理

因为EPM为每一个操作系统创建本地包，所以它首先需要安装在每一个系统中。由于在过去的几年中，在虚拟化方面取得了巨大进步，创建一些适合使用的虚拟机已经非常简便了。为了检验本书的代码示例，我们创建了一些虚拟机，虚拟机的运行模式相当于Red Hat的运行等级3，并且占用尽可能少的内存。一名同事（也是EPM开发的参与者）首先向我介绍了EPM可以做什么。当时我正在寻找一个工具，希望能够允许为我开发的工具创建基于操作系统的软件包，他提到了EPM。在阅读完在线文档（*http://www.epmhome.org/epm-book.html*）之后，我惊喜地看到实现过程可以如此简单。在这一节中，我们将进一步介绍创建适用于在多平台安装的软件包，包括Ubuntu、OS X、Red Hat、Solaris和FreeBSD。这些步骤也可以很容易地应用到其他EPM支持的系统上，例如AIX 或是HP-UX。

在我们开始这一教程之前，有一些EPM的背景需要介绍。根据EPM的官方文档，EPM一开始就被设计成使用通用的软件规格的二进制软件。正是缘于这一设计目标，同样的发布文件适合于所有的操作系统以及所有的发布格式。

EPM包管理器的需求及安装

EPM仅需要一个Bourne类型的shell，一个C编译器，make程序和gzip。即使这些应用程序没有安装到系统中，在几乎所有的*nix系统上也是很容易获得的。在下载EPM源码之后，需要执行下面的命令：

```
./configure
make
make install
```

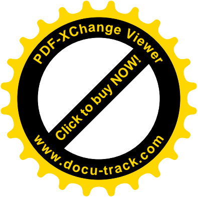
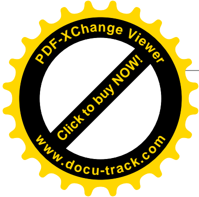
创建一个Hello World命令行发布工具

为了创建适合于每一个*nix操作系统的包，我们首先需要一些实际发布的内容。在传统意义上，我们将要创建一个简单的、称为“hello_epm.py”的命令行工具。参见例9-8。

例9-8：Hello EPM命令行工具

```
#!/usr/bin/env python

import optparse

def main():
```

www.TopSage.com

```
    p = optparse.OptionParser()
    p.add_option('--os', '-o', default="*NIX")
    options, arguments = p.parse_args()

  print 'Hello EPM, I like to make packages on %s' % options.os
  if __name__ == '__main__':
    main()
```

如果我们运行这个工具，我们会得到下面的输出：

```
$ python hello_epm.py
Hello EPM, I like to make packages on *NIX

$ python hello_epm.py --os RedHat
Hello EPM, I like to make packages on RedHat
```

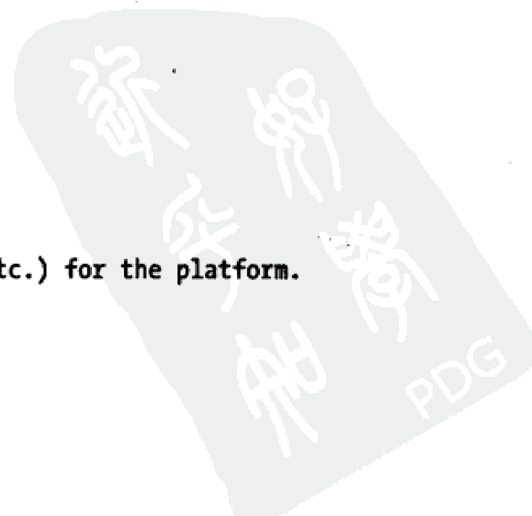
使用EPM创建面向平台的包

基本内容非常简单，以致你或许会惊奇为什么之前从没有使用EPM来打包一个跨平台的软件。EPM读取一系列描述你的软件包的文件。注释以“#”符号开始，命令以“%”符号开始，变量以符号“$”开始，最后，文件、目录、初始化脚本以及符号链接行均以字母开始。

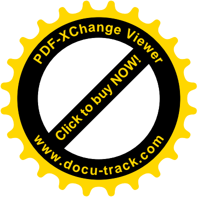
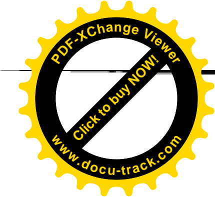
创建一个通用的跨平台的安装脚本，同时也是基于平台的软件包是可以实现的。我们的介绍将集中在创建提供商的包文件。在创建了面向平台的包之后，接下来需要创建一个声明或列表描述我们的包。例9-9是一个模板，我们使用该模板为hello_epm命令行工具创建包。对其略加修改完全可以创建你自己的工具。

例9-9：EPM的“List”模板

```
#EPM List File Can Be Used To Create Package For Any Of These Vendor Platforms
#epm -f format foo bar.list ENTER
#The format option can be one of the following keywords:

#aix - AIX software packages.
#bsd - FreeBSD, NetBSD, or OpenBSD software packages.
#depot or swinstall - HP-UX software packages.
#dpkg - Debian software packages.
#inst or tardist - IRIX software packages.
#native - "Native" software packages (RPM, INST, DEPOT, PKG, etc.) for the platform.
#osx - MacOS X software packages.
#pkg - Solaris software packages.
#portable - Portable software packages (default).
#rpm - Red Hat software packages.
#setld - Tru64 (setld) software packages.
#slackware - Slackware software packages.


# Product Information Section
```



www.TopSage.com

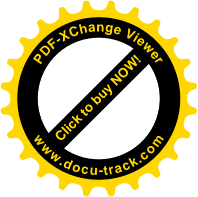
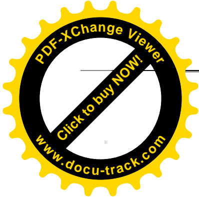
```
%product hello_epm
%copyright 2008 Py4SA
%vendor O'Reilly
%license COPYING
%readme README
%description Command Line Hello World Tool
%version 0.1

# Autoconfiguration Variables

$prefix=/usr
$exec_prefix=/usr
$bindir=${exec_prefix}/bin
$datadir=/usr/share
$docdir=${datadir}/doc/
$libdir=/usr/lib
$mandir=/usr/share/man
$srcdir=.

# Executables

%system all
f 0555 root sys ${bindir}/hello_epm hello_epm.py

# Documentation

%subpackage documentation
f 0444 root sys ${docdir}/README $srcdir/README
f 0444 root sys ${docdir}/COPYING $srcdir/COPYING
f 0444 root sys ${docdir}/hello_epm.html $srcdir/doc/hello_epm.html

# Man pages

%subpackage man
%description Man pages for hello_epm
f 0444 root sys ${mandir}/man1/hello_epm.1 $srcdir/doc/hello_epm.man
```

如果检查这个称为*hello_epm.list*的文件，你会注意到定义的变量`$srcdir`保存了当前的工作目录。为了能够在所有平台上都能成功创建包，现在需要在当前目录下创建下列内容：一个*README*文件，一个*COPYING*文件，一个*doc/hello_epm.html*文件和一个*doc/hello_epm.man*文件，并且我们的脚本`hello_epm.py`也必须在相同的目录内。

如果我们希望“欺骗一下”hello_epm.py工具，只需要在我们的打包目录内放入空文件即可，我们可以这样来做：

```
$ pwd
/tmp/release/hello_epm
$ touch README
$ touch COPYING
$ mkdir doc
$ touch doc/hello_epm.html
$ touch doc/hello_epm.man
```

查看一下目录内部，我们会看到这样的层次：

```
$ ls -lR
total 16
-rw-r--r--  1 ngift  wheel    0 Mar 10 04:45 COPYING
-rw-r--r--  1 ngift  wheel    0 Mar 10 04:45 README
drwxr-xr-x  4 ngift  wheel  136 Mar 10 04:45 doc
-rw-r--r--  1 ngift  wheel 1495 Mar 10 04:44 hello_epm.list
-rw-r--r--@ 1 ngift  wheel  278 Mar 10 04:10 hello_epm.py

./doc:
total 0
-rw-r--r-- 1 ngift wheel 0 Mar 10 04:45 hello_epm.html
-rw-r--r-- 1 ngift wheel 0 Mar 10 04:45 hello_epm.man
```

创建包

现在，我们有一个目录，包含“list”文件，其中包含了能够工作在任何支持EPM指令的平台上。只需要运行的EPM -f命令指定具体的平台以及list文件的名称即可。例9-10显示OS X中的情况。

例9-10：创建一个本地安装的OS X中的EPM

```
$ epm -f osx hello_epm hello_epm.list
epm: Product names should only contain letters and numbers!
^C
$ epm -f osx helloEPM hello_epm.list
$ ll
total 16
-rw-r--r--   1 ngift wheel     0 Mar 10 04:45 COPYING
-rw-r--r--   1 ngift wheel     0 Mar 10 04:45 README
drwxr-xr-x   4 ngift wheel   136 Mar 10 04:45 doc
-rw-r--r--   1 ngift wheel  1495 Mar 10 04:44 hello_epm.list
-rw-r--r--@  1 ngift wheel   278 Mar 10 04:10 hello_epm.py
drwxrwxrwx   6 ngift staff   204 Mar 10 04:52 macosx-10.5-intel
```

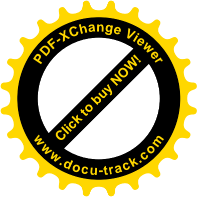
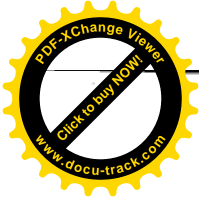
我们注意到，当包名包含下划线时会出现警告。因此，将没有下划线的包重命名，并再次运行。则会创建一个名为*macosx - 10.5-intel*的目录，包含以下内容。

```
$ ls -la macosx-10.5-intel
total 56
drwxrwxrwx  4 ngift  staff   136 Mar 10 04:54 .
drwxr-xr-x  8 ngift  wheel   272 Mar 10 04:54 ..
-rw-r--r--@ 1 ngift  staff 23329 Mar 10 04:54 helloEPM-0.1-macosx-10.5-intel.dmg
drwxr-xr-x  3 ngift  wheel   102 Mar 10 04:54 helloEPM.mpkg
```

这十分方便，因为它使得.DMG图片文件是本地OS X的，还包含installer和本地OS X的installer。

如果运行安装程序，会发现， OS X将会安装空白的帮助页面和文档，并显示空白的许可证文件。最后，我们的工具正确地安装，并创建自定义的名字，如下所示：

www.TopSage.com

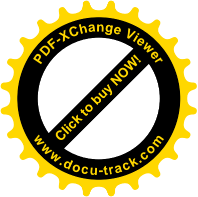


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

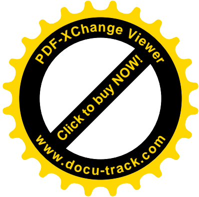
```
$ which hello_epm
/usr/bin/hello_epm
$ hello_epm
Hello EPM, I like to make packages on *NIX
$ hello_epm -h
Usage: hello_epm [options]

Options:
-h, --help      show this help message and exit
-o OS, --os=OS
$
```

EPM总结：真的非常简单

如果我们使用“scp -r”将“/tmp/release/hello-epm”复制到一个Red Hat、Ubuntu或是Solaris主机，除了使用面向本平台的名称之外，我们可以准确地运行相同的命令，而且它会立即执行。在第8章，我们检验了如何使用这一技术创建build farm，这样就可以通过运行脚本来瞬间创建跨平台包。值得注意的是，所有这些源码连同创建的示例包都可以下载。你应该能够很快地对其略加修改来创建适合自己的跨平台包。

还有一些其他的EPM高级特征，但是对其进一步介绍将超出本书的范围。如果你对创建可以处理依赖关系的包、运行pre-install和post-install脚本等内容感兴趣，那么你应该阅读EPM的官方文档，该文档涉及了所有这些案例以及更多内容。





第10章

进程与并发

引言

如果你是一名Unix/Linux系统管理员，那么对进程进行处理将会是工作的主要内容。你需要知道启动脚本、运行等级、守护进程、长时间运行的进程、并发，以及一大堆相关问题。幸运的是，Python处理进程非常容易。从Python2.4开始，Subprocess已经成为一站式模块，允许你派生出新的进程，并且与标准输入、标准输出以及标准错误输出进行会话。与进程进行会话是处理进程的一个方面，同时，理解如何部署和管理长时间运行的进程也是非常重要的。

子进程

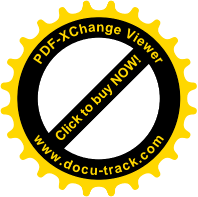
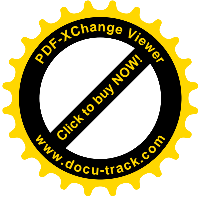
Python2.4支持子进程，并且替换了许多旧的模块，包括：os.system、os.spawn、os.popen和popen2 commands。子进程对于系统管理员以及需要与进程和“shelling out”打交道的开发人员来说是一个革命性的变化。对于许多处理进程的事务，它提供了一站式服务，并且具有管理多个进程的能力。

对于一名系统管理员，subprocess或许是单一的最重要的模块，因为它对于shelling out具有统一的API。Python中的子进程负责处理下列事务：派生新的进程连接标准输入、连接标准输出，连接错误流、侦听返回代码。为了提起你的兴趣，让我们遵循KISS（Keep It Simple Stupid）原则，使用Subprocess完成一个非常简单的处理，创建一个简单的系统调用。参见例10-1。

例10-1：子进程的简单应用

```
In [4]: subprocess.call('df -k', shell=True)
Filesystem    1024-blocks     Used Available Capacity  Mounted on
/dev/disk0s2     97349872 80043824  17050048      83%  /
devfs                 106      106         0     100%  /dev
fdesc                   1        1         0     100%  /dev
map -hosts              0        0         0     100%  /net
```


www.TopSage.com

```
map auto_home           0          0          0    100%     /home
Out[4]: 0
```

使用同样简单的语法来包含shell变量也是可行的。例10-2是一个查找主目录已用空间情况的示例。

例10-2：磁盘使用情况汇总

```
In [7]: subprocess.call('du -hs $HOME', shell=True)
28G     /Users/ngift
Out[7]: 0
```

一个Subprocess非常有意义的功能需要特别指出，即它具有禁止标准输出的能力。一些人仅对运行系统调用感兴趣，而不关心标准输出。在这种情况下，经常需要禁止subprocess.call的标准输出。幸运的是，现在有非常简便的方法可以这样做，参见例10-3。

例10-3：subprocess.call对标准输出的禁止

```
In [3]: import subprocess

In [4]: ret = subprocess.call("ping -c 1 10.0.1.1",
                              shell=True,
                              stdout=open('/dev/null', 'w'),
                              stderr=subprocess.STDOUT)
```

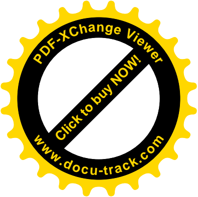
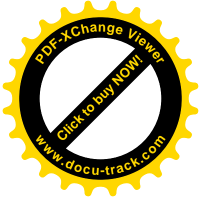
对于这两个示例以及通常的subprocess.call，还有一些事情需要指出。当你对shell命令的输出不感兴趣，只是希望程序被运行，你可以典型地使用subprocess.call。如果你需要捕获命令的输出结果，那么你需要使用subprocess.Popen。在subprocess.call与subprocess.Popen之间，存在一个非常大的差异。subprocess.call会封锁对响应的等待，而subprocess.Popen则不会。

使用subprocess的返回代码

subprocess.call有一个有趣之处需要留意：你可以使用返回代码来判断你的命令是否成功执行。如果你有一定的C或Bash的编程经验，你会对返回代码非常熟悉。“exit code”和“return code”这两个词汇经常互换使用，都是用来描述系统进程的运行状态码。

每一个进程在退出时有一个返回码，返回码的状态可以用来判断程序将要采取什么动作。通常如果一个程序退出时具有除0之外的其他返回代码，则表示程序出错。对于开发者，使用返回码最为明显的用途是判断一个使用return的进程是否具有一个非零的退出代码，如果是，则表示失败。而使用返回码的不是非常明显的用途则有许多可能。对于没有找到的程序，不能被执行的程序，通过Ctrl-C来终止的程序，分别有专门的返回码。我们将在这一节中探索Python编程中的返回码的使用。

www.TopSage.com

让我们查看具有特殊意义的公共返回码列表：

0
: 成功

1
: 普通错误

2
: shell内建的误用

126
: 激活的命令无法执行

127
: 命令无法找到

128
: 非法参数，退出

: 致命错误信号“n”

130
: 通过按Ctrl-C终止脚本执行

255
: 退出状态超出范围

最有用的情况是使用返回码0或1，0或1通常表明你刚刚执行的命令成功或失败。下面看一下一些使用`subprocess.call`的常见示例。参见例10-4。

例10-4：subprocess.call的失败返回码

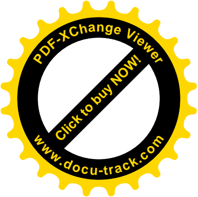
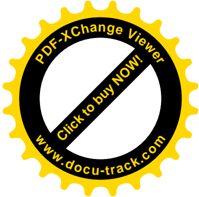
```
In [16]: subprocess.call("ls /foo", shell=True)
ls: /foo: No such file or directory
Out[16]: 1
```

因为目录不存在，我们获得返回码1表示执行失败。我们也可以捕获返回码，使用它来编写条件表达式，参见例10-5。

例10-5：基于返回码true/false的条件语句

```
In [25]: ret = subprocess.call("ls /foo", shell=True)
ls: /foo: No such file or directory

In [26]: if ret == 0:
....:     print "success"
....: else:
....:     print "failure"
....:
```


www.TopSage.com

```
....:
failure
```

这是一个关于返回为“command not found”的示例，返回码是 127。这可能是一个有用的编写运行shell命令工具的方法。你可以先尝试运行 rsync，如果得到的返回码是127，则使用scp -r。参见例10-6。

例10-6：subprocess.call基于返回码127的条件语句

```
In [28]: subprocess.call("rsync /foo /bar", shell=True)
/bin/sh: rsync: command not found
Out[28]: 127
```

让我们看看之前的示例，并试图将它变得更简单一些。我们在编写跨平台的代码时，往往会需要打开若干个*nix的窗口，需要在不同的操作系统中，编译并运行程序，而HP-UX、AIX、Solaris、FreeBSD和Red Hat，这些操作系统的命令工具都或多或少有一些不同。程序应当侦听第一个程序的返回代码，调用subprocess，若返回代码是127，则能够继续执行下一个命令。

然而，不同的操作系统会有不同的退出代码，因此如果你正在编写的是跨平台的脚本，则需要选择使用0或非0作为退出代码。 以下示例演示了Solaris 10的退出代码，与之前在Red Hat Enterprise Linux 5系统中实现的相同：

```
ash-3.00# python
Python 2.4.4 (#1, Jan 9 2007, 23:31:33) [C] on sunos5
Type "help", "copyright", "credits" or "license" for more information.
>>> import subprocess
>>> subprocess.call("rsync", shell=True)
    /bin/sh: rsync: not found
    1
```

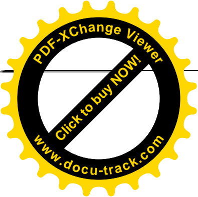
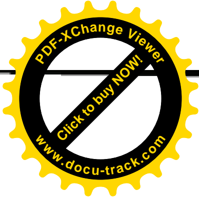
我们仍然可以使用特定的退出代码，但我们可能首先要确定操作系统是什么。确定了操作系统之后，应检查该平台的具体命令。如果发现自己也编写同样类型的代码，那么将十分有利于你去熟悉platform模块。process模块已在第8章中做了详细介绍，你可以参考第8章以获取更多信息。

接下来，让我们看看例10-7是如何使用 platform模块与IPython进行交互的，并且查看将什么参数传递给了subprocess.call。

例10-7：使用platform和Subprocess模块查看在Solaris 10上命令的执行情况

```
In [1]: import platform
In [2]: import subprocess
In [3]: platform?

        Namespace: Interactive
        File: /usr/lib/python2.4/platform.py
```


www.TopSage.com

```
        Docstring:
        This module tries to retrieve as much platform-identifying data as
        possible. It makes this information available via function APIs.

        If called from the command line, it prints the platform
        information concatenated as single string to stdout. The output
        format is useable as part of a filename.


In [4]: if platform.system() == 'SunOS':
....:     print "yes"
....:
yes

In [5]: if platform.release() == '5.10':
....:     print "yes"
....:
yes

In [6]: if platform.system() == 'SunOS':
...:        ret = subprocess.call('cp /tmp/foo.txt /tmp/bar.txt', shell=True)
...:        if ret == 0:
...:            print "Success, the copy was made on %s %s " % (platform.system(),
                 platform.release())
...:
Success, the copy was made on SunOS 5.10
```

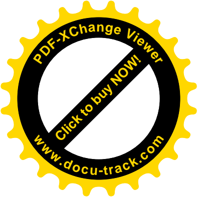
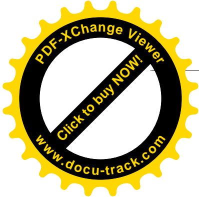
正如你所看到的，使用subprocess.call平台模块在写跨平台代码时可能是一个非常有效的工具。有关使用平台模块编写跨平台*nix代码的详细内容请参阅第8章。接下来请看例10-8。

例10-8：使用Subprocess捕获标准输出

```
In [1]: import subprocess

In [2]: p = subprocess.Popen("df -h", shell=True, stdout=subprocess.PIPE)

In [3]: out = p.stdout.readlines()

In [4]: for line in out:
...:        print line.strip()
...:
...:
Filesystem       Size    Used   Avail Capacity  Mounted on
/dev/disk0s2     93Gi    78Gi    15Gi     85%    /
devfs           107Ki   107Ki     0Bi    100%    /dev
fdesc           1.0Ki   1.0Ki     0Bi    100%    /dev
map -hosts        0Bi     0Bi     0Bi    100%    /net
map auto_home     0Bi     0Bi     0Bi    100%    /home
```

值得注意的是，readlines()返回一个具有换行符的列表。我们不得不使用line.strip()来删除换行符。Subprocess也能够与标准输入和标准输出进行通信。以下是一个简单的与进程的标准输入进行通信的示例。使用Python的一个非常有意义的方面是可以创建管

www.TopSage.com

道工厂，这在Bash中是无法想象的。只需几行代码，我们可以让命令根据参数进行创建和打印，参见例10-9。

例10-9：Subproces管理工厂

```
def multi(*args):
    for cmd in args:
        p = subprocess.Popen(cmd, shell=True, stdout = subprocess.PIPE)
        out = p.stdout.read()
        print out
```

以下是一个非常简单的函数的示例：

```
In [28]: multi("df -h", "ls -l /tmp", "tail /var/log/system.log")
Filesystem      Size   Used  Avail Capacity  Mounted on
/dev/disk0s2     93Gi   80Gi   13Gi    87%    /
devfs           107Ki  107Ki    0Bi   100%    /dev
fdesc           1.0Ki  1.0Ki    0Bi   100%    /dev
map -hosts        0Bi    0Bi    0Bi   100%    /net
map auto_home     0Bi    0Bi    0Bi   100%    /home

lrwxr-xr-x@ 1 root admin 11 Nov 24 23:37 /tmp -> private/tmp


Feb 21 07:18:50 dhcp126 /usr/sbin/ocspd[65145]: starting
Feb 21 07:19:09 dhcp126 login[65151]: USER_PROCESS: 65151 ttys000
Feb 21 07:41:05 dhcp126 login[65197]: USER_PROCESS: 65197 ttys001
Feb 21 07:44:24 dhcp126 login[65229]: USER_PROCESS: 65229 ttys002
```

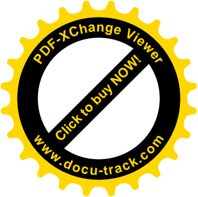
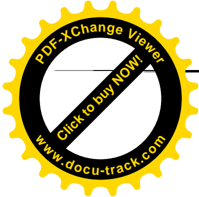
因为python和*args的功能强大，我们可以将我们的函数作为工厂而随意运行命令。根据args.pop(0)语法，每一个命令一开始将获得一个弹出的列表。如果我们使用args.pop()，它会以相反的顺序输出参数。这可能有些令人迷惑，我们可以使用简单的循环来编写同样的命令工厂函数：

```
def multi(*args):
    for cmd in args:
        p = subprocess.Popen(cmd, shell=True, stdout = subprocess.PIPE)
        out = p.stdout.read()
        print out
```

系统管理员需要十分频繁地运行一系列命令，因此创建一个简化这一过程的模块会更有意义。让我们通过一个简单的继承示例演示一下是如何实现的。参见例10-10。

例10-10：创建subprocess模块

```
#!/usr/bin/env python
from subprocess import call
import time
import sys

"""Subtube is module that simplifies and automates some aspects of subprocess"""
```

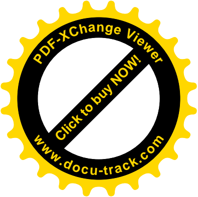
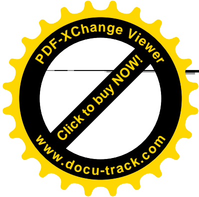



```
class BaseArgs(object):
"""Base Argument Class that handles keyword argument parsing"""

def __init__(self, *args, **kwargs):
    self.args = args
    self.kwargs = kwargs
    if self.kwargs.has_key("delay"):
        self.delay = self.kwargs["delay"]
    else:
        self.delay = 0
    if self.kwargs.has_key("verbose"):
        self.verbose = self.kwargs["verbose"]
    else:
        self.verbose = False

def run (self):
    """You must implement a run method"""
    raise NotImplementedError

    class Runner(BaseArgs):
       """Simplifies subprocess call and runs call over a sequence of commands

          Runner takes N positional arguments, and optionally:

        [optional keyword parameters]
        delay=1, for time delay in seconds
        verbose=True for verbose output

        Usage:

        cmd = Runner("ls -l", "df -h", verbose=True, delay=3)
        cmd.run()

        """
def run(self):
    for cmd in self.args:
       if self.verbose:
         print "Running %s with delay=%s" % (cmd, self.delay)
       time.sleep(self.delay)
       call(cmd, shell=True)
```

让我们看看应当如何使用新创建的模块：

```
In [8]: from subtube import Runner
In [9]: r = Runner("df -h", "du -h /tmp")

In [10]: r.run()
Filesystem      Size   Used   Avail Capacity  Mounted on
/dev/disk0s2     93Gi   80Gi    13Gi    87%    /
devfs           108Ki  108Ki    0Bi    100%    /dev
fdesc           1.0Ki  1.0Ki    0Bi    100%    /dev
map -hosts        0Bi    0Bi    0Bi    100%    /net
map auto_home     0Bi    0Bi    0Bi    100%    /home
4.0K /tmp

In [11]: r = Runner("df -h", "du -h /tmp", verbose=True)
```

www.TopSage.com

```
In [12]: r.run()
Running df -h with delay=0
Filesystem      Size   Used  Avail Capacity  Mounted on
/dev/disk0s2     93Gi   80Gi   13Gi    87%    /
devfs           108Ki  108Ki    0Bi   100%    /dev
fdesc           1.0Ki  1.0Ki    0Bi   100%    /dev
map -hosts        0Bi    0Bi    0Bi   100%    /net
map auto_home     0Bi    0Bi    0Bi   100%    /home
Running du -h /tmp with delay=0
4.0K    /tmp
```

如果我们将ssh密钥安装到所有的系统上，我们可以很轻松地这样来编写代码：

```
machines = ['homer', 'marge','lisa', 'bart']
for machine in machines:
    r = Runner("ssh " + machine + "df -h", "ssh " + machine + "du -h /tmp")
    r.run()
```

这是远端命令执行的一个粗浅的示例，但是其思想意义却非常有价值，因为Red Hat Emerging技术组有一个项目，希望通过Python使大量集群主机的批量脚本简化。根据Func网站的描述"这有意义的且设计良好的示例 —— 重启所有运行httpd的系统。虽然代码是精心设计的，但是也是非常简单的，感谢Func。"我们对第8章的Func将做进一步详细的介绍，我们介绍了一个自制分发系统，该系统工作在*nix平台上。

```
results = fc.Client("*").service.status("httpd")
for (host, returns) in results.iteritems():
if returns == 0:
fc.Client(host).reboot.reboot()
```

因为subprocess为shelling out提供统一的API，我们可以十分容易地向标准输出设备写入。在例10-11中，我们让计算单词的工具侦听标准输入，然后根据单词数目写一串字符到进程中。

例10-11：使用Subprocess与标准输入进行通信

```
In [35]: p = subprocess.Popen("wc -c", shell=True, stdin=subprocess.PIPE)
In [36]: p.communicate("charactersinword")
    16
```

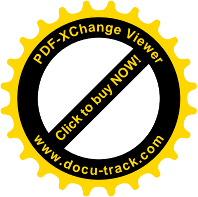
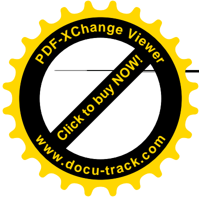
具有相同功能的Bash代码如下所示：

```
> echo charactersinword | wc -c
```

这次让我们模拟Bash并重定向一个文件到标准输入。首先，我们需要向文件中写入数据，我们使用新的Python2.6语法来这样做。记住，如果你使用的是Python2.5，你必须加载idiom：

```
In [5]: from __future__ import with_statement
```

www.TopSage.com

```
In [6]: with open('temp.txt', 'w') as file:
...:      file.write('charactersinword')
```

我们可以用非常典型的方式再次打开文件，读入文件中的数据，并将其作为一个字符串赋值给f：

```
In [7]: file = open('temp.txt')

In [8]: f = file.read()
```

那么，我们将重定向文件的输出到等待进程：

```
In [9]: p = subprocess.Popen("wc -c", shell=True, stdin=subprocess.PIPE)

In [10]: p.communicate(f)

In [11]: p.communicate(f)
16
```

在Bash中，这会等同于下面的命令序列：

```
% echo charactersinword > temp.txt
% wc -c < temp.txt
16
```

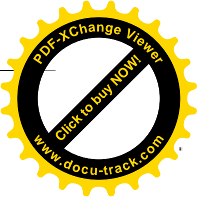
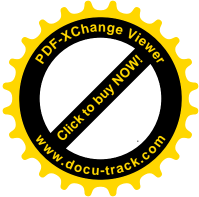
接下来，让我们实际看一下如何将多个命令连接到一起，就像我们在典型的shell环境中所做的那样。先看一下在Bash中一系列命令是如何被连接起来，然后是相同的这一系列命令在Python中被连接起来。我们遇到的一个实际应用是在处理logfiles文件的时候。在例10-12中，我们试图成功登录到Macintosh笔记本的屏保。

例10-12：使用 Subprocess连接命令

```
In Bash here is a simple chain:

[ngift@Macintosh-6][H:10014][J:0]> cat /etc/passwd | grep 0:0 | cut -d ':' -f 7
/bin/sh

Here is the same chain in Python:

In [7]: p1 = subprocess.Popen("cat /etc/passwd", shell=True, stdout=subprocess.PIPE)
In [8]: p2 = subprocess.Popen("grep 0:0", shell=True, stdin=p1.stdout, stdout=subprocess.
PIPE)
In [9]: p3 = subprocess.Popen("cut -d ': ' -f 7", shell=True, stdin=p2.stdout,
   stdout=subprocess.PIPE)

In [10]: print p3.stdout.read()
/bin/sh
```

我们可以使用子进程管道来做一些事情，但这并不表示我们必须这样做。在前一个示例中，我们通过连接一系列命令来获得根用户的shell。Python的内建模块可以为我们完成这一工作，因此多了解相关信息，知道有些时候甚至没必要使用Subprocess也是很重要

www.TopSage.com

的，Python的内建的模块或许可以为你完成一些工作。许多你希望在shell中做的事情，例如tar或zip，Python也可以做。如果你在使用Subprocess做一个非常复杂的shell连接，查看一下，看看Python是否有一个内建的能够完成同样功能模块会是一个好主意。参见例10-13。

例10-13：使用 pwd（密码数据库模块）代替Subprocess

```
In [1]: import pwd

In [2]: pwd.getpwnam('root')
Out[2]: ('root', '********', 0, 0, 'System Administrator', '/var/root', '/bin/sh')

In [3]: shell = pwd.getpwnam('root')[-1]

In [4]: shell
Out[4]: '/bin/sh'
```

Subprocess也可以同时处理发送输入和接收输出，并且也可以侦听标准错误输出。让我们看一个这方面的示例。值得注意的是，在IPython内部，当我们想要写一段类似例10-14这样的代码块时，我们使用“ed upper.py”来自动切换到Vim。

例10-14：发送输入、接收输出并侦听标准错误

```
import subprocess

p = subprocess.Popen("tr a-z A-Z", shell=True,stdin=subprocess.PIPE,
stdout=subprocess.PIPE)
output, error = p.communicate("translatetoupper")
print output
```

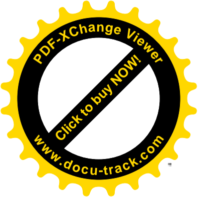
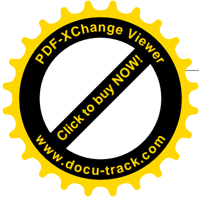
当我们在IPython中退出Vim时，它自动运行这一代码块，而我们可以获得如下结果：

```
    done. Executing edited code...
    TRANSLATETOUPPER
```

使用Supervisor来管理进程

作为一名系统管理员，会经常需要管理和处理进程。当web开发者发现他们的系统管理是一个Python专家时，他们会非常激动的，因为对临时管理长时间运行的进程，许多Python web框架不能提供一个完美的方案。在这方面，Supervisor可以发挥作用。Supervisor通过决定如何对长时间运行的进程进行控制，以及确保在系统重启时进程可以重新启动来实现管理。

Supervisor可以做一些比web应用部署更多的事情，有更多的通用应用可以完成。Supervisor能够作为跨平台控制者来管理和与进程进行交互。它可以启动、停止，重启其他*nix系统中的程序，也可以重启崩溃的进程，而且处理起来非常方便。Supervisor的开发者之一，Chris McDonough，告诉我们，它也可以帮助管理“坏”进程，例如消耗

太多内存的进程或占用过多CPU的进程。Supervisor通过XML-RPC XML-RPC Interface Extensions Event Notification System（扩展事件通告系统的XML-RPC XML-RPC接口）实现远端控制。

大多数*nix系统管理员主要与“supervisord”和“supervisorctl”打交道。“supervisord”是一个守护进程，用于将指定的程序作为子进程来运行；“supervisorctl”是一个客户端程序，可以查看日志并通过统一的会话来控制进程。还有一个web 接口，但本书是关于*nix，因此我们继续讨论。

在写本书时，Supervisor的最新版本是3.0.x。最新版本的手册可以在这里找到：*http://supervisord.org/manual/current/*。安装Supervisor非常简单，因为实际上你可以使用easy_install进行安装。假设你已经使用virtualenv创建了一个虚拟Python安装目录，可以使用下面的命令来安装Supervisor:

```
bin/easy_install supervisor
```

这将把Supervisor安装到*bin*目录中。如果你之前使用esay_install安装你的系统Python，那么它会安装在类似*/usr/local/bin*或是系统脚本目录中。

为了让一个非常基本的Supervisor守护进程开始运行，接下来创建一个简单的脚本，该脚本实现打印输出、休眠10秒，然后销毁。这与常驻进程正好相反，但是这方面正显示了Supervisor的强大功能，其具有自动重启和将程序驻留内存的能力。现在，我们可以简单地通过使用专门的supervisor命令echo_supervisord_conf，显示输出*supervisord.conf*文件的内容。在这个示例中，我们将显示输出的内容保存到*/etc/supervisord.conf*中。有一点需要注意，Supervisor配置文件可以保存到任何位置，因为supervisord守护进程可以在运行时，通过选项来指定配置文件的位置。

```
echo_supervisord_conf > /etc/supervisord.conf
```

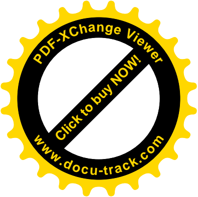
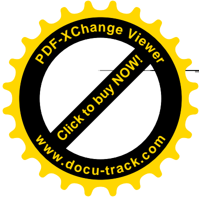
我们已经准备好创建一个非常简单的进程示例，该进程会在运行几秒后销毁。我们使用supervisor的自动重启功能来保持这个进程一直处于活动状态。参见例10-15。

例10-15：Supervisor重启僵死进程的简单示例

```
#!/usr/bin/env python
import time
print "Daemon runs for 3 seconds, then dies"
time.sleep(3)
print "Daemons dies"
```

正如之前介绍的，为了实际上在supervisord中运行一个子程序，需要编辑配置文件，在配置文件中添加我们的程序。我们继续在*/etc/supervisord.conf*中添加几行代码：

www.TopSage.com

```
[program:daemon]
command=/root/daemon.py              ; the program (relative uses PATH, can take args)
autorestart=true                ; retstart at unexpected quit (default: true)
```

现在，可以开始监管并且使用supervisorectl来查看和启动进程：

```
[root@localhost]~# supervisord
[root@localhost]~# supervisorctl
daemon                          RUNNING    pid 32243, uptime 0:00:02
supervisor>
```

在这一点上，可以运行help命令来查看对于supervisorctl可以使用哪些选项：

```
supervisor> help

Documented commands (type help topic):
========================================
EOF   exit  maintail  quit    restart   start   stop  version
clear help  open      reload  shutdown  status  tail
```

接下来，启动在配置文件中称为daemon的进程，然后跟踪它来查看它被销毁然后又神奇地被重新唤醒。它会被运行，然后被杀死，然后再被运行。

```
supervisor> stop daemon
daemon: stopped
supervisor> start daemon
daemon: started
```

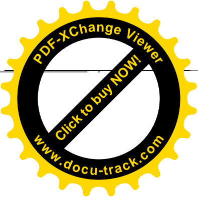
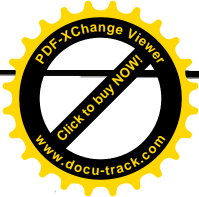
最后要介绍的是，可以交互地跟踪该程序的输出：

```
supervisor> tail -f daemon
== Press Ctrl-C to exit ==
    for 3 seconds, then die
    I just died
    I will run for 3 seconds, then die
```

使用Screen来管理进程

一个管理常驻进程的可选方法是使用GNU的screen应用程序。作为系统管理员，即使你不使用screen，也是值得去了解它的，哪怕你不会使用它来管理Python程序。screen的非常有用的核心特征之一是它允许你分离常驻进程，并且可以再返回。这是非常有用的功能，可以作为主要的Unix技术来学习。让我们看一个典型的应用，其中我们希望从常驻的web应用（例如trac）分离。有一些方法可以配置trac，但是最简单的方法通过screen从独立运行的trac进程分离。

运行进程所需要的只是附加screen到常驻进程的前端，按Ctrl-A，然后按Ctrl-D进行分离。重新连接到该进程，需在screen中键入，然后再次按Ctrl-A。

在例10-16中，我们告诉tracd在screen中运行。一旦进程开始运行，可以简单地使用Ctrl-A进行分离。如果希望重新连接，使用Ctrl-D。

例10-16：*在screen中运行Python进程*

```
screen python2.4 /usr/bin/tracd --hostname=trac.example.com --port 8888
  -r --single-env --auth=*,
/home/noahgift/trac-instance/conf/password,tracadminaccount /home/example/trac-instance/

If I ever need to reattach I can run:

[root@cent ~]# screen -r
There are several suitable screens on:
4797.pts-0.cent     (Detached)
24145.pts-0.cent     (Detached)
Type "screen [-d] -r [pid.]tty.host" to resume one of them.
```

在产品环境中这个方法或许不是最好的，但是当做开发工作时，或是出于个人使用的目的，它确实有它的优点。

Python中的线程

线程可能被描绘成“对一些人来说是无法躲避的恶运”，许多人不喜欢使用线程，尽管线程可以并行处理多个事务。线程不同于进程，因为他们都运行在同一个进程内，共享状态及内存。这是线程最大的优势也是劣势。优势是你可以创建一个数据结构，所有线程可以访问，而无须创建IPC，或是进程间的通信机制。

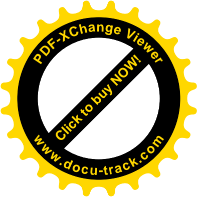
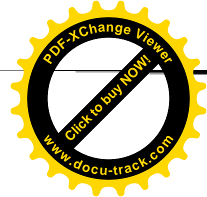
在处理线程时，有一些潜在的复杂问题。通常，一个只有几行代码的简单程序，当引入线程时可以变得极为复杂。线程在没有添加扩展追踪功能的情况下是非常难调试的，甚至由于追踪的结果令人困惑或无法理解，调试会变得更为复杂。一位作者在编写用于发现数据中心的SNMP发现系统时，单纯的需要去创建的线程的规模就令其很难应对。

在处理线程时会有一些策略，通常实现一个功能齐备的追踪库是策略之一。这也就是说，创建追踪库在解决复杂问题时成了一个非常方便的工具。

对于系统管理员，知道一些线程相关的基本编程知识也是必要的。以下是一些线程的使用方法，对于系统管理员每天的管理任务是非常有帮助的：网络自动发现，同时取回多个web页，对服务器进行压力测试，执行网络相关的任务。

为了与KISS主旨相一致，让我们使用一个可能是最简单的线程示例。将模块线程化需要理解面向对象编程，这一点非常重要。这或许会有点挑战，并且如果你对面向对象的编程技术（OOP）经验有限，或根本没有任何经验，那么这个示例或许会令你有一些迷惑。尽管你也可以通过本书的介绍来了解这方面的知识，并且实现这里介绍的一些技

www.TopSage.com

术。但我们建议你阅读Mark Lutz的《LearningPython》(O'Reilly)，来进一步理解一些基本的OOP知识。最后需要指出的，实践OOP编程也是学习OOP的最好方式。

因为本书是关于Python的实用技术，我们使用或许是最简单的线程示例来对线程进行进一步介绍。在这个简单的线程脚本中，我们从线程进行继承。线程设置一个全局计数变量，然后重载线程的run方法。最后，我们发起5个线程，明确地打印他们的号码。

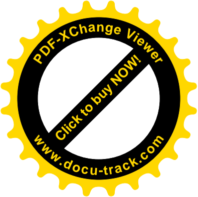
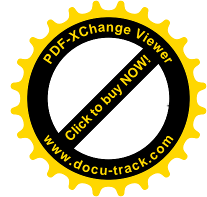
在众多实现方法中，这个示例有些过分简单，而且有一个不太好的设计，因为我们使用一个全局计数器，这样多个线程可以共享状态。通常在使用线程时使用队列是非常不错的方法，因为队列会处理共享状态的复杂性。参见例10-17。

例10-17：KISS 线程示例

```
#subtly bad design because of shared state
import threading
import time
count = 1
class KissThread(threading.Thread):
    def run(self):
        global count
        print "Thread # %s: Pretending to do stuff" % count
        count += 1
        time.sleep(2)
        print "done with stuff"
    for t in range(5):
        KissThread().start()


[ngift@Macintosh-6][H:10464][J:0]> python thread1.py
Thread # 1:  Pretending to do stuff
Thread # 2:  Pretending to do stuff
Thread # 3:  Pretending to do stuff
Thread # 4:  Pretending to do stuff
Thread # 5:  Pretending to do stuff
done with stuff
done with stuff
done with stuff
done with stuff
done with stuff
```

```
    #common.py
    import subprocess
    import time

    IP_LIST = [ 'google.com',
      'yahoo.com',
      'yelp.com',
      'amazon.com',
      'freebase.com',
      'clearink.com',
      'ironport.com' ]

    cmd_stub = 'ping -c 5 %s'
```

www.TopSage.com

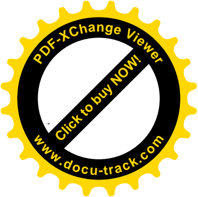
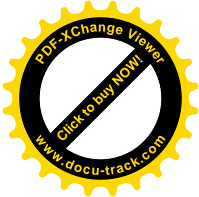



```
def do_ping(addr):
  print time.asctime(), "DOING PING FOR", addr
  cmd = cmd_stub % (addr,)
  return subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE)

from common import IP_LIST, do_ping
import time

z = []
#for i in range(0, len(IP_LIST)):
for ip in IP_LIST:
  p = do_ping(ip)
  z.append((p, ip))

for p, ip in z:
  print time.asctime(), "WAITING FOR", ip
  p.wait()
  print time.asctime(), ip, "RETURNED", p.returncode

jmjones@dinkgutsy:thread_discuss$ python nothread.py
Sat Apr 19 06:45:43 2008 DOING PING FOR google.com
Sat Apr 19 06:45:43 2008 DOING PING FOR yahoo.com
Sat Apr 19 06:45:43 2008 DOING PING FOR yelp.com
Sat Apr 19 06:45:43 2008 DOING PING FOR amazon.com
Sat Apr 19 06:45:43 2008 DOING PING FOR freebase.com
Sat Apr 19 06:45:43 2008 DOING PING FOR clearink.com
Sat Apr 19 06:45:43 2008 DOING PING FOR ironport.com
Sat Apr 19 06:45:43 2008 WAITING FOR google.com
Sat Apr 19 06:45:47 2008 google.com RETURNED 0
Sat Apr 19 06:45:47 2008 WAITING FOR yahoo.com
Sat Apr 19 06:45:47 2008 yahoo.com RETURNED 0
Sat Apr 19 06:45:47 2008 WAITING FOR yelp.com
Sat Apr 19 06:45:47 2008 yelp.com RETURNED 0
Sat Apr 19 06:45:47 2008 WAITING FOR amazon.com
Sat Apr 19 06:45:57 2008 amazon.com RETURNED 1
Sat Apr 19 06:45:57 2008 WAITING FOR freebase.com
Sat Apr 19 06:45:57 2008 freebase.com RETURNED 0
Sat Apr 19 06:45:57 2008 WAITING FOR clearink.com
Sat Apr 19 06:45:57 2008 clearink.com RETURNED 0
Sat Apr 19 06:45:57 2008 WAITING FOR ironport.com
Sat Apr 19 06:46:58 2008 ironport.com RETURNED 0
```

注意： 对于接下来的线程示例，需要注意的是它们是一些比较复杂的示例，因为同样的事情可以通过使用subprocess.Popen来实现。如果你需要启动多个进程，并且等待响应，subprocess.Popen是一个非常好的选择。如果你需要与每一个进程进行通信，那么在线程中使用subprocess.Popen是比较适合的。在演示的多个示例中，需要格外强调的是并发，并发通常是出于平衡的考虑。一个模块可以适合所有的情况通常是困难的，不管它是线程或是进程，或是诸如stackless或twisted这样的异步库。以下示例是ping一个大的IP地址池的最有效的方法。

现在我们有等同于Hello World这样的线程，让我们实际动手来做一些让系统管理员欣赏

www.TopSage.com

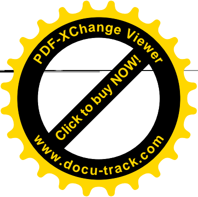
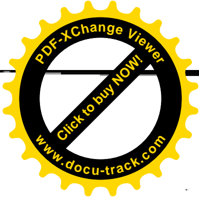
的事情。略微修改示例来创建一个小的脚本来ping一个网络，并等待响应。这可以说是通用网络工具中的入门级工具。参见例10-18。

例10-18：线程化的 ping扫瞄

```
#!/usr/bin/env python
from threading import Thread
import subprocess
from Queue import Queue

num_threads = 3
queue = Queue()
ips = ["10.0.1.1", "10.0.1.3", "10.0.1.11", "10.0.1.51"]
def pinger(i, q):
    """Pings subnet"""
    while True:
        ip = q.get()
        print "Thread %s: Pinging %s" % (i, ip)
        ret = subprocess.call("ping -c 1 %s" % ip,
                        shell=True,
                        stdout=open('/dev/null', 'w'),
                        stderr=subprocess.STDOUT)
        if ret == 0:
            print "%s: is alive" % ip
        else:
            print "%s: did not respond" % ip
        q.task_done()

for i in range(num_threads):

    worker = Thread(target=pinger, args=(i, queue))
    worker.setDaemon(True)
    worker.start()

for ip in ips:
q   ueue.put(ip)

print "Main Thread Waiting"
queue.join()
print "Done"
```

当我们运行这段代码时，可以看到下面的输出结果：

```
[ngift@Macintosh-6][H:10432][J:0]# python ping_thread_basic.py
Thread 0: Pinging 10.0.1.1
Thread 1: Pinging 10.0.1.3
Thread 2: Pinging 10.0.1.11
Main Thread Waiting
10.0.1.1: is alive
Thread 0: Pinging 10.0.1.51
10.0.1.3: is alive
10.0.1.51: is alive
10.0.1.11: did not respond
Done
```

这个示例值得仔细剖析，应该认真理解每一行代码，但是首先要进行一些说明。使用线程开发一个ping扫描程序来对一个子网进行扫描，是使用线程的一个非常不错的示例。一个普通的没有使用线程的Python程序会占用N *（平均响应时间/ping）。ping有两个状态：响应状态和超时状态。一个典型的网络将是一个响应与超时的混合。

这表示如果写一个ping扫描程序，连续检查一个具有254个地址的C类网络，它会占用254*（~3秒），总计12.7分钟。如果使用线程，可以缩短一些时间。这就是为什么线程对于网络编程非常重要的原因。现在进一步考虑一个现实的情况。在一个典型的数据中心中有多少个子网存在？20个？30个？50个？显然，顺序编程会很快地变得不现实，而线程将是一个理想的选择。

现在，重新看看我们使用的这个简单的脚本，查看一些实现的细节。首先要去查看的事情是要载入的模块，而尤其需要查看的两件事情是线程和队列。正如我们在第一个示例中所介绍的，不带队列使用线程会将它变得非常复杂，大大超出许多人可以实际操控的能力。如果你发现需要使用线程，那么使用队列模块会是一个好选择。这是为什么呢？因为队列模块通过信号量的使用会明显减轻数据保护的需要，因为队列本身已经是通过内部的一个信号量进行保护了。

想象一下，你是一个生活在中世纪的农场主或科学家。你已经注意到一群乌鸦（通常称为“杀手”，原因可以咨询Wikipedia），通常20多只组成一队，攻击你的庄稼。

因为这些乌鸦非常聪明，通过扔石子几乎无法将他们吓走，因为你最快每3秒扔一块石子，而乌鸦的数量有时会增加到50只。为了吓走所有的乌鸦，会占用几分钟，而仅仅这几分钟已经可以对你的庄稼造成严重破坏。如果你是一名学习数学或自然科学的学生，你可以理解到这一问题的解决方案其实非常简单。

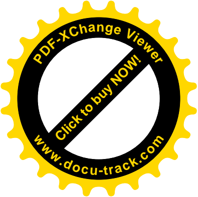
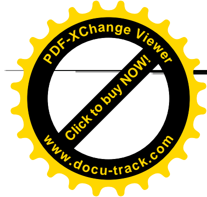
你需要在篮子中创建一个石子队列，然后分配几个工人分别从篮子中取出石子并迅速扔向乌鸦。

使用这个新策略，如果你分配30个工人从篮子中取石子，并扔向乌鸦，你会用不到10秒钟的时间就可以向50只乌鸦扔掷石头。这在Python中是线程和队列的基本关系。你指定一组工人，当队列为空，则工作完成。

在集中方式下，队列代表了任务。示例程序的最为重要的一部分内容是join()。如果查看docstring，会看到queue.join()的声明如下：

```
Namespace:    Interactive
File:        /System/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/
  Queue.py
Definition:     Queue.Queue.join(self)
Docstring:
Blocks until all items in the Queue have been gotten and processed.
```


www.TopSage.com

```
The count of unfinished tasks goes up whenever an item is added to the
queue. The count goes down whenever a consumer thread calls task_done()
to indicate the item was retrieved and all work on it is complete.

When the count of unfinished tasks drops to zero, join() unblocks.
```

join是防止主线程在其他线程获得机会完成队列中的任务之前从程序中退出的方式。回到农场的比喻中，类似于当工人排队等候扔石子时，农场主扔下一篮石子离开了。在我们的示例中，如果对queue.join()进行注释，可以看到相反的结果：首先，注释掉queue.join这一行：

```
print "Main Thread Waiting"
#By commenting out the join, the main program exits before threads have a chance
 to run
#queue.join()
print "Done"
```

接下来，我们看一下例10-19。

例10-19：main线程在worker线程之前退出的示例

```
[ngift@Macintosh-6][H:10189][J:0]# python ping_thread_basic.py
Main Thread Waiting
Done
Unhandled exception in thread started by
Error in sys.excepthook:

Original exception was:
```

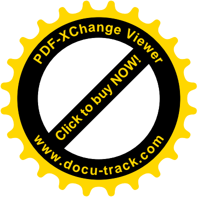
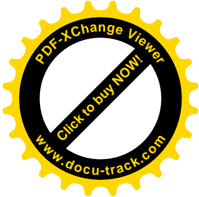
在具有了线程和队列的背景知识之后，我们看一下以下几行代码。这里，我们对通常会传递给一个程序的普通值进行了硬编码。num_threads是工人线程数，queue是队列的实例，最后，ips是IP地址的列表，这些地址会最终放入到队列中：

```
num_threads = 3
queue = Queue()
ips = ["10.0.1.1", "10.0.1.3", "10.0.1.11", "10.0.1.51"]
```

这是一个完成程序中所有工作的函数。当每一线程每次从队列中提取出一个IP地址时，这个函数会由线程执行。值得注意的是，一个新的IP地址出栈时就像它是在一个列表中一样。这样操作允许我们取出每一个元素，直到队列为空。最后请注意，q.task_done()在while循环结束时被调用。这非常重要，因为它告诉join()已经完成从队列中提取元素的工作。或者简单地说，工作完成了。让我们看一下Queun.Queue.task_done中的docstring：

```
File:       /System/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/
  Queue.py
Definition:    Queue.Queue.task_done(self)
Docstring:
```

www.TopSage.com

```
Indicate that a formerly enqueued task is complete.
Used by Queue consumer threads. For each get() used to fetch a task,
a subsequent call to task_done() tells the queue that the processing
on the task is complete.

If a join() is currently blocking, it will resume when all items
have been processed (meaning that a task_done() call was received
for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items
placed in the queue.
```

从这个docstring我们看到在q.get()和q.task_done()以及q.join()之间有一定联系。这非常像一个故事的开始，发展以及结束：

```
def pinger(i, q):
    """Pings subnet"""
    while True:
        ip = q.get()
        print "Thread %s: Pinging %s" % (i, ip)
        ret = subprocess.call("ping -c 1 %s" % ip,
                        shell=True,
                        stdout=open('/dev/null', 'w'),
                        stderr=subprocess.STDOUT)
        if ret == 0:
            print "%s: is alive" % ip
        else:
            print "%s: did not respond" % ip
        q.task_done()
```

接着看下面的示例，其中使用了一个简单的循环作为控制器，对线程池的创建进行管理。需要注意的是，线程池会阻塞或等待，直到队列中有事件发生。

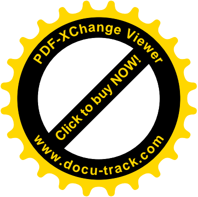
在我们的程序中潜藏着一个微妙的惊喜，可以确保让你逃脱追踪。这需要使用setDaemon(True)。在start方法被调用之前如果没有进行设置，程序会不定期挂起。

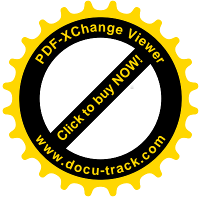
原因非常简单，因为如果守护线程正在运行，程序仅能退出。或许你已经注意到在ping函数中，使用了无限循环。由于线程永远不会死亡，将其声明为守护线程是必然的。仅需要将worker.start()这一行注释掉即可看到结果。为了截断追踪，在没有为线程设置守护标志的情况下，程序会无限挂起。你应该自行进行检测，因为这会破坏进程的一些功能：

```
for i in range(num_threads):
    worker = Thread(target=pinger, args=(i, queue))
    worker.setDaemon(True)
    worker.start()
```

程序中执行到这一点，我们有了一个缓冲池，其中有三个线程等待执行绑定操作。它们

www.TopSage.com





仅需要将每一元素放到它们的队列中。这会向线程发出获取元素的信号，并且执行要求的操作，在这个示例中，执行的操作是ping一个IP地址：

```
for ip in ips:
    queue.put(ip)
```

夹在两行输出语句之间的最为关键的一行代码，最终具有程序的控制权。正如之前讨论的，在一个队列上调用join将导致程序的主线程等待，直到队列为空为止。这也是为什么线程和队列就像巧克力和花生酱一样，两者的味道都是非常不错的，合起来会成为尤其特别的美味。

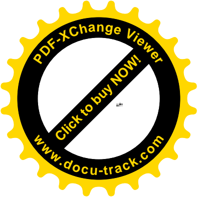
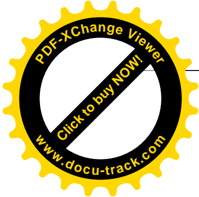
```
print "Main Thread Waiting"
queue.join()
print "Done"
```

为了真正理解线程和队列，需要进一步介绍示例，创建另一个线程池和另一个队列。在第一个示例中，我们ping一个IP地址列表，该列表由线程池从队列中获得。在接下来的示例中，我们会让第一个线程池放置有效的IP地址（该地址是响应ping的地址）到第二个队列中。

接下来，第二个线程池会从第一个队列中取得IP地址，然后执行一个arp命令，返回IP地址，如果能找到Mac地址也将返回Mac地址。这是如何执行的，参见例10-20。

例10-20：多队列与多线程池

```
#!/usr/bin/env python
#This requires Python2.5 or greater
from threading import Thread
import subprocess
from Queue import Queue
import re

num_ping_threads = 3
num_arp_threads = 3
in_queue = Queue()
out_queue = Queue()
ips = ["10.0.1.1", "10.0.1.3", "10.0.1.11", "10.0.1.51"]

def pinger(i, iq, oq):
    """Pings subnet"""
    while True:
        ip = iq.get()
        print "Thread %s: Pinging %s" % (i, ip)
        ret = subprocess.call("ping -c 1 %s" % ip,
                        shell=True,
                        stdout=open('/dev/null', 'w'),
                        stderr=subprocess.STDOUT)
        if ret == 0:
            #print "%s: is alive" % ip
            #place valid ip address in next queue
```

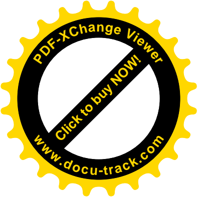
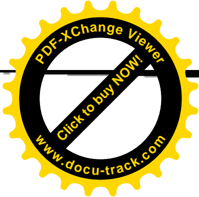



```
            oq.put(ip)
        else:
            print "%s: did not respond" % ip
        iq.task_done()

def arping(i, oq):
    """grabs a valid IP address from a queue and gets macaddr"""
    while True:
        ip = oq.get()
        p = subprocess.Popen("arping -c 1 %s" % ip,
                             shell=True,
                             stdout=subprocess.PIPE)
        out = p.stdout.read()

        #match and extract mac address from stdout
        result = out.split()
        pattern = re.compile(":")
        macaddr = None
        for item in result:
            if re.search(pattern, item):
                macaddr = item
        print "IP Address: %s | Mac Address: %s " % (ip, macaddr)
        oq.task_done()

#Place ip addresses into in queue
for ip in ips:
    in_queue.put(ip)

#spawn pool of ping threads
for i in range(num_ping_threads):

    worker = Thread(target=pinger, args=(i, in_queue, out_queue))
    worker.setDaemon(True)
    worker.start()

#spawn pool of arping threads
for i in range(num_arp_threads):

    worker = Thread(target=arping, args=(i, out_queue))
    worker.setDaemon(True)
    worker.start()

print "Main Thread Waiting"
#ensures that program does not exit until both queues have been emptied
in_queue.join()
out_queue.join()

print "Done"
```

这里我们运行这段代码，以下是代码的输出结果：

```
python2.5 ping_thread_basic_2.py
Main Thread Waiting
Thread 0: Pinging 10.0.1.1
Thread 1: Pinging 10.0.1.3
Thread 2: Pinging 10.0.1.11
Thread 0: Pinging 10.0.1.51
```


www.TopSage.com



PDF-XChange Viewer Click to buy NOW! www.docu-track.com

```
IP Address: 10.0.1.1 | Mac Address: [00:00:00:00:00:01]
IP Address: 10.0.1.51 | Mac Address: [00:00:00:80:E8:02]
IP Address: 10.0.1.3 | Mac Address: [00:00:00:07:E4:03]
10.0.1.11: did not respond
Done
```

为了实现这一解决方案，我们通过添加另一个线程和队列池，略微扩展了第一个示例的功能。这是一个重要的技术，可以放入到你自己的工具包中，因为使用队列模块使得线程更方便且更安全。该技术甚至可以毫无疑问地称为必备技术。

使用threading.Timer的线程延迟

Python中的线程还有另一个功能，可以为系统管理员完成任务提供一些便利。通过使用threading.Timer，在一个线程中运行被定时执行的函数变得非常简单。例10-21是专门设计的线程计时器示例。

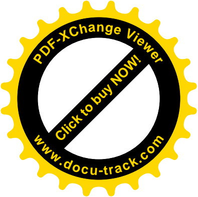
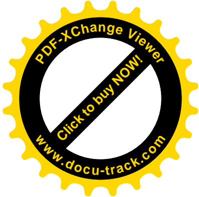
例10-21：线程计时器

```
#!/usr/bin/env python
from threading import Timer
import sys
import time
import copy

#simple error handling
if len(sys.argv) != 2:
        print "Must enter an interval"
        sys.exit(1
#our function that we will run
def hello():
    print "Hello, I just got called after a %s sec delay" % call_time

#we spawn our time delayed thread here
delay = sys.argv[1]
call_time = copy.copy(delay) #we copy the delay to use later
t = Timer(int(delay), hello)
t.start()

#we validate that we are not blocked, and that the main program continues
print "waiting %s seconds to run function" % delay
for x in range(int(delay)):
    print "Main program is still running for %s more sec" % delay
    delay = int(delay) - 1
    time.sleep(1)
```

如果执行这段代码，可以看到一个为函数定时的延迟被触发，而主线程，或是程序，仍继续运行：

```
[ngift@Macintosh-6][H:10468][J:0]# python thread_timer.py 5
waiting 5 seconds to run function
Main program is still running for 5 more sec
```

www.TopSage.com

```
Main program is still running for 4 more sec
Main program is still running for 3 more sec
Main program is still running for 2 more sec
Main program is still running for 1 more sec
Hello, I just got called after a 5 sec delay
```

线程化的事件处理

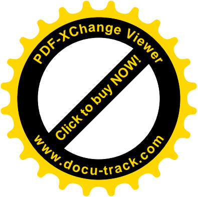
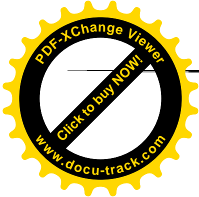
因为这是一本关于系统管理的书，让我们使用之前的技术来看一个实际的应用。在这个示例中，我们采用延迟线程技术，并且混合了一个对两个目录中文件名的差异进行查询的事件循环。我们可能已经变得非常有经验，可以检验文件的修改时间，但是出于保持示例简单性的思想，我们查看一下这个事件循环如何查询注册的事件。事件一旦被触发，一个动作方法会在一个延迟线程中被调用。

这个模块可以非常容易地抽象为更一般的工具，但是现在例10-22还是一个硬代码。该段代码可以保持两个目录同步，如果它们不同步，在后台延迟线程会使用“rsync -av --delete”进行处理。

例10-22：线程化的目录同步工具

```
#!/usr/bin/env python
from threading import Timer
import sys
import time
import copy
import os
from subprocess import call

class EventLoopDelaySpawn(object):

"""An Event Loop Class That Spawns a Method in a Delayed Thread"""

def __init__(self, poll=10,
                    wait=1,
                    verbose=True,
                    dir1="/tmp/dir1",
                    dir2="/tmp/dir2"):

    self.poll = int(poll)
    self.wait = int(wait)
    self.verbose = verbose
    self.dir1 = dir1
    self.dir2 = dir2

def poller(self):
    """Creates Poll Interval"""
    time.sleep(self.poll)
    if self.verbose:
        print "Polling at %s sec interval" % self.poll

def action(self):
    if self.verbose:
```

www.TopSage.com

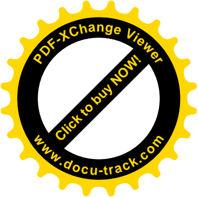
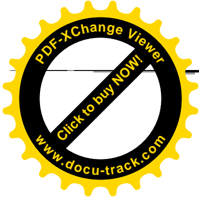
```
        print "waiting %s seconds to run Action" % self.wait
    ret = call("rsync -av --delete %s/ %s" % (self.dir1, self.dir2), shell=True)

def eventHandler(self):
    #if two directories contain same file names
    if os.listdir(self.dir1) != os.listdir(self.dir2):
        print os.listdir(self.dir1)
        t = Timer((self.wait), self.action)
        t.start()
        if self.verbose:
            print "Event Registered"
    else:
        if self.verbose:
            print "No Event Registered"

def run(self):
    """Runs an event loop with a delayed action method"""
    try:
        while True:
            self.eventHandler()
            self.poller()

    except Exception, err:
        print "Error: %s " % err

    finally:
        sys.exit(0)

E = EventLoopDelaySpawn()
E.run()
```

读者可能会认为延迟不是严格需要的，这确实是事实。然而，延迟可以创建一些额外的好处。如果你添加一个延迟，例如5秒，这期间你发现另一个事件发生（例如，如果你的主目录被意外删除），你可以告诉线程进行取消。线程延迟是一个非常不错的机制，可以创建根据条件可以取消的将来的操作。

进程

在Python中线程不是处理并发的唯一方法。事实上，进程相比线程也有一些优势，其可以扩展到多处理器，这与Python中的线程不一样。因为GIL（全局解释器锁）的原因，在某一时刻只有单一线程可以运行，并且这被限制到单处理器。为了让Python可以充分利用CPU，线程已不是一个好的选择。在这样的情况下，使用独立的进程是非常适合的。

如果一个问题需要使用多处理器，那么进程会是一个不错的选择。另外，有许多库不能与线程一起工作。例如，当前Python的Net-SNMP库是同步的，因此写并发代码需要使用fork进程。

线程共享全局状态，进程则是完全独立的，与进程进行通信需要一些技术。幸运的是，通过管道与进程进行通信难度比较小。有一个进程库我们将在这里进一步介绍其中的细节。这里有一些讨论，涉及整合进程库到Python的标准库中，这对于理解非常有帮助。在之前的说明中，我们提及一个可选的使用`subprocess.Popen`来创建多个进程的方法。在许多情况下，并行执行代码是非常不错且非常简便的选择。如果阅读第13章，可以看到一个示例，该示例中我们创建一个可以创建多个dd进程的工具。

Processing模块

那么，我们提到的processing模块又是什么呢？在这本书出版时，其描述是这样的：processing是一个Python语言的软件包，支持使用标准库中threading模块的API创建进程。关于processing模块最重要的内容之一是它或多或少可以映射到线程API。这表示你没必要为了创建进程（而不是线程）学习一个新的API。访问网址：*http://pypi.python.org/pypi/processing*，可以找到有关processing模块的更多信息。

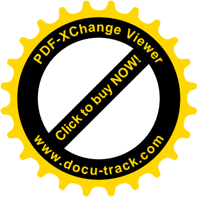
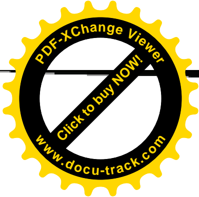
注意： 正如之前谈论的，处理并发没有什么简单的方法。这个示例也可以认为是低效的，因为仅使用了`subprocess.Popen`，而不是processing模块的fork，并且运行了`subprocess.call`。然而，在一些大的应用背景环境下，使用队列类型API有许多好处，还可以作为一个与之前线程示例的合理的对比。有一些将processing模块合并到Subprocess的讨论，因为Subprocess当前缺少像processing模块一样管理大量进程的能力。这种对Subprocess的需求在最初的PEP（Python Enhancement Proposal）中有描述：*http://www.python.org/dev/peps/pep-0324/*。

现在，我们已经具有了一些关于processing模块的背景知识，接下来看一下例10-23。

例10-23：processing模块

```
#!/usr/bin/env python
from processing import Process, Queue
import time

def f(q):
    x = q.get()
    print "Process number %s, sleeps for %s seconds" % (x,x)
    time.sleep(x)
    print "Process number %s finished" % x
q = Queue()

for i in range(10):
    q.put(i)
    i = Process(target=f, args=[q])
    i.start()

print "main process joins on queue"
i.join()
print "Main Program finished"
```

www.TopSage.com

如果查看输出，会看到下面的内容：

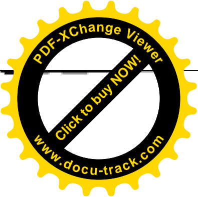
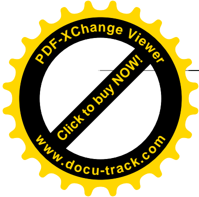
```
[ngift@Macintosh-7][H:11199][J:0]# python processing1.py
Process number 0, sleeps for 0 seconds
Process number 0 finished
Process number 1, sleeps for 1 seconds
Process number 2, sleeps for 2 seconds
Process number 3, sleeps for 3 seconds
Process number 4, sleeps for 4 seconds
main process joins on queue
Process number 5, sleeps for 5 seconds
Process number 6, sleeps for 6 seconds
Process number 8, sleeps for 8 seconds
Process number 7, sleeps for 7 seconds
Process number 9, sleeps for 9 seconds
Process number 1 finished
Process number 2 finished
Process number 3 finished
Process number 4 finished
Process number 5 finished
Process number 6 finished
Process number 7 finished
Process number 8 finished
Process number 9 finished
Main Program finished
```

程序所做的所有工作是告诉每一个进程休眠与其进程号相同的时间。正如你所看到的，这是一个非常简洁的API。

现在已经有了等同的Hello World程序，我们可以做一些更有意义的事情。或许你还记得在线程一节中，我们写了一个简单的线程化的子网发现脚本。因为进程API与线程API十分相似，使用进程而不使用线程，就可以实现一个几乎等同的脚本。参见例10-24。

例10-24：基于进程的ping扫描

```
#!/usr/bin/env python
from processing import Process, Queue, Pool
import time
import subprocess
from IPy import IP
import sys

q = Queue()
ips = IP("10.0.1.0/24")
def f(i,q):
    while True:
        if q.empty():
            sys.exit()
        print "Process Number: %s" % i
        ip = q.get()
        ret = subprocess.call("ping -c 1 %s" % ip,
                        shell=True,
```

```
                        stdout=open('/dev/null', 'w'),
                        stderr=subprocess.STDOUT)
        if ret == 0:
            print "%s: is alive" % ip
        else:
            print "Process Number: %s didn't find a response for %s " % (i, ip)

for ip in ips:
    q.put(ip)

#q.put("192.168.1.1")

for i in range(50):
    p = Process(target=f, args=[i,q])
    p.start()

print "main process joins on queue"
p.join()
print "Main Program finished"
```

这段代码看起来非常类似于之前介绍过的线程化的代码。如果看一下输出，会看到相似的输出结果：

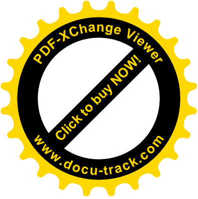
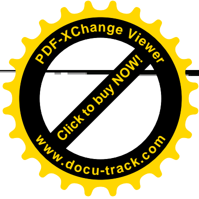
```
[snip]
10.0.1.255: is alive
Process Number: 48 didn't find a response for 10.0.1.216
Process Number: 47 didn't find a response for 10.0.1.217
Process Number: 49 didn't find a response for 10.0.1.218
Process Number: 46 didn't find a response for 10.0.1.219
Main Program finished
[snip]
[ngift@Macintosh-7][H:11205][J:0]#
```

这个代码段还需要进一步解释。尽管API非常相似，但仍略微有些不同。需要注意的是，每一个进程运行在一个无限循环体内，从每一个队列中获得元素。为了告诉进程“远离”processing模块，我们创建了一个条件语句，查看队列是否为空。50个线程中的每一个都首先查看队列是否为空，如果为空，那么它会通过运行sys.exit让自己退出。

如果队列仍不为空，那么进程会获得队列中的元素，在这个示例中，元素包括IP地址以及相应的被指定的作业（本示例中为ping指定的IP地址）。主程序使用了join，就像我们在线程脚本中所做的那样，将队列进行添加，直到队列为空。在所有的工作进程都结束，队列变空之后，接下来的输出语句被执行，表明程序结束。

使用API与使用processing模块一样简便，fork取代了线程使问题变得相对简单。在第7章，我们介绍了一个实际使用processing模块的Net-SNMP的实现，其中Net-SNMP已经同步绑定到Python上。

www.TopSage.com

调度Python进程

现在，已经介绍了在Python中处理进程的全部内容，接下来应该讨论调度这些进程的方法。使用传统风格的cron非常适合在Python中运行进程。

在POSIX系统中，调度目录的出现是cron的一个非常新颖的特点。使用cron非常方便，只需要放python脚本到以下四个默认目录中即可（这也是我们使用cron的仅有的方法）：*/etc/cron.daily*，*/etc/cron.hourly*，*/etc/cron.monthly*和*/etc/cron.weekly*。

有相当一部分系统管理员已经在他们工作期间，编写了非常不错的具有传统风格的反映磁盘使用情况的email。通常是在*/etc/cron.daily*中放一个类似这样的Bash脚本：

```
df -h | mail -s "Nightly Disk Usage Report" staff@example.com
```

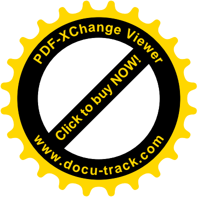
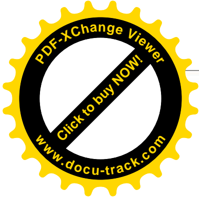
当你将这个脚本放在*/etc/cron.daily/diskusage.sh*中时，邮件看起来会像是以下这样。

```
From: guru-python-sysadmin@example.com
Subject:     Nightly Disk Usage Report
Date:     February 24, 2029 10:18:57 PM EST
To: staff@example.com

Filesystem              Size  Used Avail Use% Mounted on
/dev/hda3                72G   16G   52G  24% /
/dev/hda1                99M   20M   75M  21% /boot
tmpfs                  1010M     0 1010M   0% /dev/shm
```

还有一个更好的方法。甚至cron作业也可以从Python脚本中（而不是Bash或Perl中）获得好处。事实上，cron和Python在一起配合得非常好。让我们看一个Bash的示例，然后将其转化为Python。参见例10-25。

例10-25：Python的基于Cron的磁盘报告邮件

```
import smtplib
import subprocess
import string

p = subprocess.Popen("df -h", shell=True, stdout=subprocess.PIPE)
MSG = p.stdout.read()
FROM = "guru-python-sysadmin@example.com"
TO = "staff@example.com"
SUBJECT = "Nightly Disk Usage Report"
msg = string.join((
    "From: %s" % FROM,
    "To: %s" % TO,
    "Subject: %s" % SUBJECT,
    "",
    MSG), "\r\n")
server = smtplib.SMTP('localhost')
server.sendmail(FROM, TO, msg)
server.quit()
```

这是一个简单的方法，用来创建一个自动的基于cron的磁盘报告。在许多情况下，它应该工作得非常出色。这里是Python如何进行处理的过程。首先，使用subprocess.Popen来读取df的标准输出。接下来，创建变量From、To和Subject，然后将这些字符串连接到一起创建信息。这是该脚本最难的部分。最后，设置smtp发送邮件服务器来使用localhost，并将之前设置的变量传入server.sendmail()。使用这个脚本的典型方法是将该脚本简单地放在*/etc/cron.daily/nightly_disk_report.py*中。

如果你是一个使用Python的新手，你或许希望将这个脚本作为样板代码来让一些工作更加快速完成。在第4章中，我们详细地介绍创建email信息，因此你可以参阅该章以获得更多建议。

daemonizer

守护进程对于在Unix上花费了大量时间的人是个福音。守护进程可以做许多事情，包括处理请求、发送文件到打印机（例如，lpd），处理HTTP请求、以及文件服务（例如，Apache的httpd）等。

那么什么是守护进程？一个守护进程通常被认为是一个不对终端进行控制的后台任务。如果你熟悉Unix的作业控制，你或许会想到执行命令时，在命令的结尾使用&将会创建一个守护进程。或是启动一个进程之后，按Ctrl-z，然后通过bg命令来将其转为守护进程。所有这些都会使一个进程在后台运行，但是它们不会破坏进程与shell进程之间的独立，并且与控制终端（或许是你的shell进程）无关。因此，以下是守护进程的三个特征：在后台运行，与启动它的进程脱离，无须控制终端。使用shell的job命令将进程移至后台只是这三个特征中的第一个。

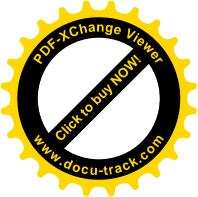
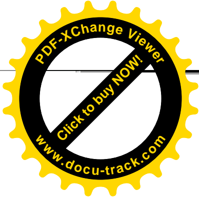
TopSage.com

接下来是一段代码，定义了一个名为daemonize()的函数，该函数使得被调用的代码成为一个在前一段中讨论的守护进程。函数摘自David Ascher编著的《Python Cookbook》（O'Reilly出版）第二版，第 388~389 页的“Forking a Daemon Process on Unix”。接下来的代码与Richard Stevens在《UNIX Network Programming: The Sockets Networking API》（O'Reilly出版）中给出的代码十分相近，为将进程变为守护进程提供了“适合”的方法。对于任何不熟悉Stevens书的人，该书普遍被认为是Unix网络编程的参考书，其中包括如何创建Unix下的守护进程的内容。参见例10-26。

例10-26：Daemonize函数

```
import sys, os
def daemonize (stdin='/dev/null', stdout='/dev/null', stderr='/dev/null'):
    # Perform first fork.
    try:
        pid = os.fork( )
        if pid > 0:
```


www.TopSage.com

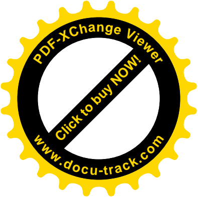
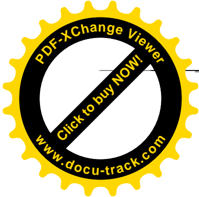



```
            sys.exit(0) # Exit first parent.
    except OSError, e:
        sys.stderr.write("fork #1 failed: (%d) %s\n" % (e.errno, e.strerror))
        sys.exit(1)
    # Decouple from parent environment.
    os.chdir("/")
    os.umask(0)
    os.setsid( )
    # Perform second fork.
    try:
        pid = os.fork( )
        if pid > 0:
            sys.exit(0) # Exit second parent.
    except OSError, e:
        sys.stderr.write("fork #2 failed: (%d) %s\n" % (e.errno, e.strerror))
        sys.exit(1)
    # The process is now daemonized, redirect standard file descriptors.
    for f in sys.stdout, sys.stderr: f.flush( )
    si = file(stdin, 'r')
    so = file(stdout, 'a+')
    se = file(stderr, 'a+', 0)
    os.dup2(si.fileno( ), sys.stdin.fileno( ))
    os.dup2(so.fileno( ), sys.stdout.fileno( ))
    os.dup2(se.fileno( ), sys.stderr.fileno( ))
```

代码做的第一件事是fork()一个进程。fork()创建了一个运行进程的副本，副本被认为是子进程，而原始的进程被认为是父进程。当子进程结束fork，父进程可以自由退出。在使用fork之后，我们通过检测pid进行识别。如果pid是正数，这表明我们在父进程中。如果你之前从没有fork过一个子进程，这或许会令你感到困惑。在调用os.fork()之后，有相同的两个副本在运行。它们都检测fork的返回值，返回值为0则表示在子进程中，返回进程ID号，则表示在父进程中。无论哪个进程有一个非零的返回代码（这只能是父进程），都会退出。如果这时有一个异常发生，进程将退出。如果你从交互式shell（例如，Bash）中调用这个脚本，你将重新得到提示符，因为你启动的进程已经终止了。但是你启动的进程的子进程（例如，子孙进程）仍然运行。

接下来进程要做的三件事情是修改目录到/（os.chdir("/")，设置它的掩码为0（os.umask(0)），创建一个新的会话（os.setsid()）。修改目录到/，将守护进程放到总是存在的目录中。修改目录到/的一个额外的好处是你的常驻进程不会束缚住你卸载一个文件系统的能力（如果碰巧文件系统的目录需要被卸载）。接下来进程所做的事情是修改它的文件模式、创建掩码到最大的允许限度。如果一个守护进程需要创建具有组可读、组可写权限的文件，一个被继承的具有更严格权限的掩码会有反面作用。

这三步中的最后一步（os.setsid()）或许是最不为人熟悉的部分。setsid调用做了一系列事情：首先它使得该进程成为一个新会话的领导者。接下来，它将进程转变成为一个新的进程组的领导者。最后，也许是转化为守护进程的最重要的一步，使该进程不再

控制终端。事实上，不再控制终端意味着该进程不会成为一些终端的意外（或者甚至是有意）作业控制的牺牲品。对于一个不会被中断的常驻进程来说，这一点是非常重要的。

但是有趣之处并不是到此为止。在调用os.setsid()之后，另一个fork发生了。第一个fork和setsid为第二个fork设置环境，他们从控制终端分离，并设置进程为会话的领导者。另一个fork意味着结果进程不是会话的领导者。这表示进程不会获得一个控制终端。这第二个frok不是必须的，更多是一种预防。没有最后的fork，进程可以获得控制终端的仅有的方法是：是否直接打开了一个终端设备，而没有使用O_NOCTTY标志。

这里做的最后一件事情是对一些文件的清除和再调整。标准输出和标准错误输出（sys.stdout和sys.stderr）被清空。这保证了那些数据流的信息在这里被创建。该函数允许调用者定义stdin、stdout和stderr文件，其默认值是/dev/null。代码或者是来自用户定义，或是默认的stdin，stdout和stderr，并设置进程的标准输入、输出和错误输出分别到这些文件。

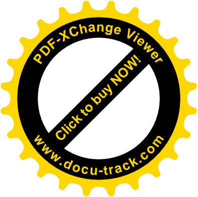
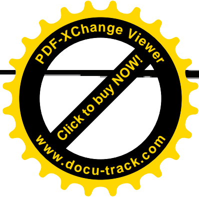
那么，你如何使用daemonizer？假设daemonizer代码是在一个名这daemonize.py的模块中，例10-27是使用它的样例脚本。

例10-27：使用daemonizer

```
from daemonize import daemonize
import time
import sys

def mod_5_watcher():
    start_time = time.time()
    end_time = start_time + 20
    while time.time() < end_time:
        now = time.time()
        if int(now) % 5 == 0:
            sys.stderr.write('Mod 5 at %s\n' % now)
        else:
            sys.stdout.write('No mod 5 at %s\n' % now)
        time.sleep(1)

if __name__ == '__main__':
    daemonize(stdout='/tmp/stdout.log', stderr='/tmp/stderr.log')
    mod_5_watcher()
```

这个脚本首先转化为守护进程，然后指定*/tmp/stdout.log*为标准输出，*/tmp/stderr.log*为标准错误输出。接下来进行20秒检测，在两次检测之间休眠1秒。如果时间以秒为单位，可以被5整除，将其写入标准错误输出。如果时间没能被5整除，写到标准输出。由于进程使用*/tmp/stdout.log*作为标准输出，使用*/tmp/stderr.log*作为标准错误输出，应该在运行这个示例之后查看文件中的结果。

www.TopSage.com

在运行这个脚本之后，我们立即看到一个新的提示符出现了：

```
jmjones@dinkgutsy:code$ python use_daemonize.py
jmjones@dinkgutsy:code$
```

以下是来自示例的结果：

```
jmjones@dinkgutsy:code$ cat /tmp/stdout.log
No mod 5 at 1207272453.18
No mod 5 at 1207272454.18
No mod 5 at 1207272456.18
No mod 5 at 1207272457.19
No mod 5 at 1207272458.19
No mod 5 at 1207272459.19
No mod 5 at 1207272461.2
No mod 5 at 1207272462.2
No mod 5 at 1207272463.2
No mod 5 at 1207272464.2
No mod 5 at 1207272466.2
No mod 5 at 1207272467.2
No mod 5 at 1207272468.2
No mod 5 at 1207272469.2
No mod 5 at 1207272471.2
No mod 5 at 1207272472.2
jmjones@dinkgutsy:code$ cat /tmp/stderr.log
Mod 5 at 1207272455.18
Mod 5 at 1207272460.2
Mod 5 at 1207272465.2
Mod 5 at 1207272470.2
```

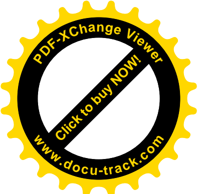
这是一个非常简单的写入守护进程的示例，但是非常有意义的是它包含了基本的概念。你可以使用这个daemonizer来写目录的监测程序，网络的监测程序可以是网络服务器，或是任何你可以想象的，运行一段较长时间（或者没有指定具体时间）的进程。

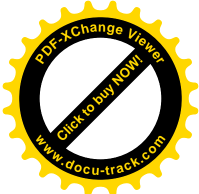
本章小结

本章非常有意义，展示了Python在处理进程时的成熟与强大。Python具有非常完备且精密的线程API，但是记得GIL也总是有好处的。如果你被I/O绑定，那么这不是什么问题，但是如果你需要多处理器，那么使用进程是一个好的选择。一些人认为使用进程比使用线程要好，甚至在GIL不存在的情况下，其主要原因是调试线程非常困难。

最后，如果你还没有准备好，那么熟悉一下Subprocess模块是一个不错的主意。Subprocess为处理子进程提供了一站式服务。

www.TopSage.com





第11章

创建GUI

当人们考虑一名系统管理员的职责包括哪些时，构造GUI应用或许根本不会让人想起。然而，有时你需建立一个GUI应用，通过GUI应用，你的工作会简单得多。在广泛意义上讲，你正在使用GUI，包括传统上使用GTK和QT工具包的GUI应用，也包括基于web的应用。

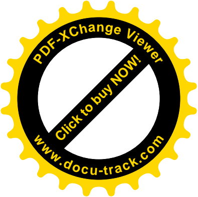
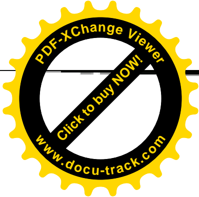
本章将集中在PyGTK、curses和Django的web框架。我们从基本的GUI开始，然后创建一个非常简单的使用PyGTK的应用，然后使用curses和Django建立相同的应用。最后，我们用少量代码向你演示Django作为一个非常好的数据库前台是如何工作的。

GUI创建理论

当写一个控制台应用时，你经常期望它能够运行并结束，不需要用户的干预。当脚本从cron、at或是其他工具中运行时，这是明显的情况。但是当你写一个GUI工具时，为了使事件发生并执行你的工具，需要用户提供一些输出。想一下你的GUI应用程序经验，例如浏览器、电子邮件客户端、word字处理器。你运行这些应用程序，应用程序执行一系列的初始化，或许是加载一些配置并将其设为某种已知的状态。但是通常应用程序只等待用户的一些操作。当然有一些应用程序似乎由其自身来控制执行，例如，Firefox的自动检测升级，不需要明确的请求或是用户的建议，但是这是另一回事。

应用程序等待什么呢？当用户做了一些操作，它是如何知道该如何处理的呢？应用程序等待发生的事件。一个事件是发生在应用中的事情，尤其对于GUI组件引发的事件，例如按下按钮或是复选框被选中。当这些事件发生时，应用程序知道该如何去做，因为程序员将特定的事件与特定的代码相关联。代码是与某一事件相关的代码，可以参考 event handlers（事件句柄）。GUI工具的任务之一是当相关的事件发生时，调用正确的事件句柄。为了更为精确，GUI工具提供一个事件循环，静默地循环以等待事件发生，而当事件发生时，它能正确地进行响应。

www.TopSage.com

行为被事件驱动。当为GUI应用编码时，你将决定当一个用户做了某些操作时应用程序具有何种行为。你需要设置事件句柄，当用户触发事件时，GUI工具可以进行调用。

这里介绍了应用程序的行为，但是表单又是怎么回事呢？也就是说，你如何获得应用中的按钮、文本域、标签和复选框？对该问题的答案可能会有些不同。你可以选择使用为GUI工具使用的GUI编辑器。GUI编辑器列出了GUI应用所需的各种各样的组件，例如按钮、标签、复选框等。例如，如果你工作在Mac主机上，并且选择写一个Cocoa应用程序，那么界面编辑器列出了你可以使用的所有GUI组件；如果你正使用Linux系统下的PyGTK，你可以使用Glade作为界面编辑器；如果你正使用PyQT，你可以使用QT Designer作为界面编辑器。

GUI编辑器非常有用，但是有时你或许希望对GUI有更多的控制，甚至超出了编辑器所能提供的帮助。在这种情况下，通过写少量代码来生成一个GUI并不困难。在PyGTK，每一类型的GUI组件对应于一个Python类。例如，一个窗口是gtk.window类的对象；一个按键是gtk.Button类的对象。为了创建一个简单的具有一个窗口和一个按钮的GUI应用，你需要对gtk.Window和gtk.Button类进行实例化，并将按钮添加到窗口上。如果你希望在单击按钮时，按钮可以执行一些动作，你必然为该按钮的单击事件定义一个事件句柄。

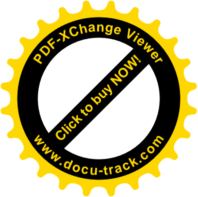
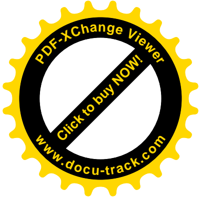
生成一个简单的PyGTK应用

我们将编写一段简单的代码，使用已经介绍的gtk.window和gtk.Button类。以下是一个简单的GUI应用，该程序没有执行任何有意义的动作，只是为了向大家演示一些GUI编程的基本原则。

在运行这个示例或是编写自己的PyGTK应用之前，你必须安装PyGTK。如果你正运行一个相对较新的Linux发布版本，安装非常简单。对Windows来说，看起来甚至更容易。如果你正在运行Ubuntu，应该已经默认安装。如果没有一个针对你所用平台的二进制发布，你可能会遇到些麻烦。参见例11-1。

例11-1：简单的PyGTK 应用（单窗口单按钮）

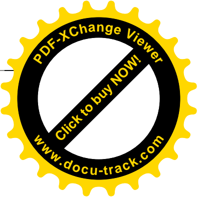
```
#!/usr/bin/env python

import pygtk
import gtk
import time

class SimpleButtonApp(object):
    """This is a simple PyGTK app that has one window and one button.
    When the button is clicked, it updates the button's label with the current time.
    """
```

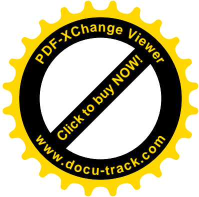



```
    def __init__(self):
        #the main window of the application
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)

        #this is how you "register" an event handler. Basically, this
        #tells the gtk main loop to call self.quit() when the window "emits"
        #the "destroy" signal.
        self.window.connect("destroy", self.quit)

        #a button labeled "Click Me"
        self.button = gtk.Button("Click Me")

        #another registration of an event handler. This time, when the
        #button "emits" the "clicked" signal, the 'update_button_label'
        #method will get called.
        self.button.connect("clicked", self.update_button_label, None)

        #The window is a container. The "add" method puts the button
        #inside the window.
        self.window.add(self.button)

        #This call makes the button visible, but it won't become visible
        #until its container becomes visible as well.
        self.button.show()

        #Makes the container visible
        self.window.show()

    def update_button_label(self, widget, data=None):
        """set the button label to the current time

        This is the handler method for the 'clicked' event of the button
        """
        self.button.set_label(time.asctime())

    def quit(self, widget, data=None):
        """stop the main gtk event loop

        When you close the main window, it will go away, but if you don't
        tell the gtk main event loop to stop running, the application will
        continue to run even though it will look like nothing is really
        happening.
        """
        gtk.main_quit()


    def main(self):
        """start the gtk main event loop"""
        gtk.main()

if __name__ == "__main__":
s = SimpleButtonApp()
s.main()
```

在这个示例中，第一件需要注意的事情是main类继承自object而不是某些GTK类。在PyGTK中创建一个GUI应用不需要必备面向对象的编程经验。你确实必须实例化对象，

www.TopSage.com





但是不必创建自定义的类。然而，对于复杂一些的示例（例如我们正创建的），我们强烈建议创建你自己的类。对于一个GUI应用，创建你自己的类的主要的好处是所有你的GUI组件（包括窗口、按钮、复选框等）都会附加到相同的对象上，这允许应用程序很容易地访问这些组件。

由于选择创建一个自定义的类，首先需要开始理解的是在构造器中发生了什么（__init__()方法）。事实上，在这个示例中，通过观察构造器你可以看到发生了什么。这个示例已经被很好地注释，因此不用在这里重复每一个解释，但是会给出一个总结。我们创建了两个GUI对象，一个是gtk.window，另一个是gtk.Button。将按钮放到窗口中，因为窗口是一个容器对象。我们也创建了窗口和按钮的事件句柄，分别针对销毁和点击事件。如果运行这段代码，这会显示一个具有按钮的窗口，按钮标签为“Click Me”。每一次你点击按钮的时候，它会修改按钮标签为当前的时间。图11-1和图11-2是该应用程序在点击按钮之前和之后的截屏。

图11-1：简单的PyGTK应用 —— 在点击按钮之前

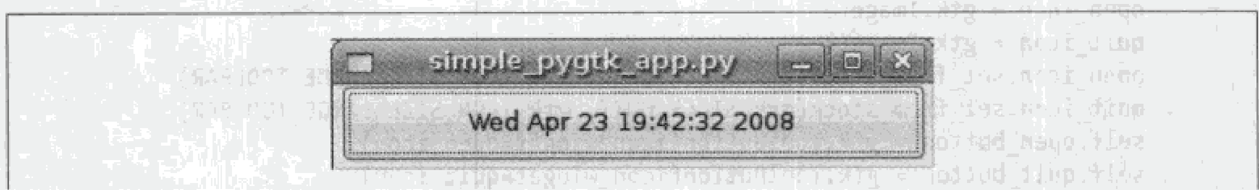


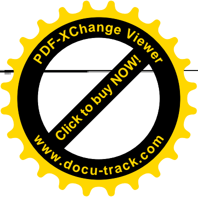
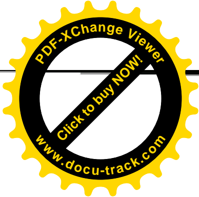
图11-2：简单的PyGTK应用 —— 在点击按钮之后

使用PyGTK创建Apache日志浏览器

现在，已经介绍了创建GUI和使用PyGTK的基本知识，接下来是一个示例，使用PyGTK生成一个更实际的应用。我们将介绍创建一个Apache日志浏览器的过程。在这一应用中将要包括的功能如下：

- 选择和打开指定的日志文件；
- 查看行数，远端主机，状态，发送字节数；
- 通过行号，远端主机，状态或是发送字节数对日志进行排序。

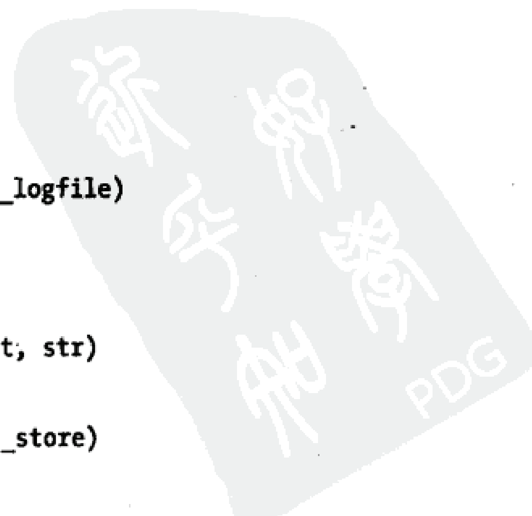
这一示例生成了Apache日志解析代码，这是我们在第3章中编写的代码。

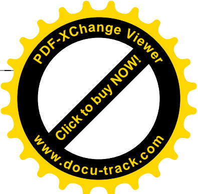
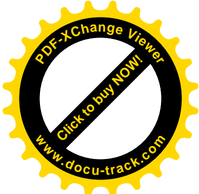
例11-2是日志浏览器的源代码。

例11-2：PyGTK Apache日志浏览器

```
#!/usr/bin/env python

import gtk
from apache_log_parser_regex import dictify_logline

class ApacheLogViewer(object):
    """Apache log file viewer which sorts on various pieces of data"""

    def __init__(self):
        #the main window of the application
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.set_size_request(640, 480)
        self.window.maximize()

        #stop event loop on window destroy
        self.window.connect("destroy", self.quit)

        #a VBox is a container that holds other GUI objects primarily for layout
        self.outer_vbox = gtk.VBox()

        #toolbar which contains the open and quit buttons
        self.toolbar = gtk.Toolbar()

        #create open and quit buttons and icons
        #add buttons to toolbar
        #associate buttons with correct handlers
        open_icon = gtk.Image()
        quit_icon = gtk.Image()
        open_icon.set_from_stock(gtk.STOCK_OPEN, gtk.ICON_SIZE_LARGE_TOOLBAR)
        quit_icon.set_from_stock(gtk.STOCK_QUIT, gtk.ICON_SIZE_LARGE_TOOLBAR)
        self.open_button = gtk.ToolButton(icon_widget=open_icon)
        self.quit_button = gtk.ToolButton(icon_widget=quit_icon)
        self.open_button.connect("clicked", self.show_file_chooser)
        self.quit_button.connect("clicked", self.quit)
        self.toolbar.insert(self.open_button, 0)
        self.toolbar.insert(self.quit_button, 1)

        #a control to select which file to open
        self.file_chooser = gtk.FileChooserWidget()
        self.file_chooser.connect("file_activated", self.load_logfile)

        #a ListStore holds data that is tied to a list view
        #this ListStore will store tabular data of the form:
        #line_numer, remote_host, status, bytes_sent, logline
        self.loglines_store = gtk.ListStore(int, str, str, int, str)

        #associate the tree with the data...
        self.loglines_tree = gtk.TreeView(model=self.loglines_store)
        #...and set up the proper columns for it
        self.add_column(self.loglines_tree, 'Line Number', 0)
        self.add_column(self.loglines_tree, 'Remote Host', 1)
        self.add_column(self.loglines_tree, 'Status', 2)
        self.add_column(self.loglines_tree, 'Bytes Sent', 3)
        self.add_column(self.loglines_tree, 'Logline', 4)
```



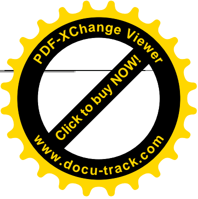
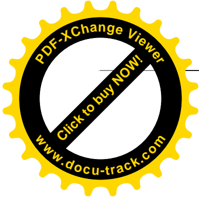
www.TopSage.com

```
        #make the area that holds the apache log scrollable
        self.loglines_window = gtk.ScrolledWindow()

        #pack things together
        self.window.add(self.outer_vbox)
        self.outer_vbox.pack_start(self.toolbar, False, False)
        self.outer_vbox.pack_start(self.file_chooser)
        self.outer_vbox.pack_start(self.loglines_window)
        self.loglines_window.add(self.loglines_tree)

        #make everything visible
        self.window.show_all()
        #but specifically hide the file chooser
        self.file_chooser.hide()

    def add_column(self, tree_view, title, columnId, sortable=True):
        column = gtk.TreeViewColumn(title, gtk.CellRendererText() , text=columnId)
        column.set_resizable(True)
        column.set_sort_column_id(columnId)
        tree_view.append_column(column)

    def show_file_chooser(self, widget, data=None):
        """make the file chooser dialog visible"""
        self.file_chooser.show()

    def load_logfile(self, widget, data=None):
        """load logfile data into tree view"""
        filename = widget.get_filename()
        print "FILE-->", filename
        self.file_chooser.hide()
        self.loglines_store.clear()
        logfile = open(filename, 'r')
        for i, line in enumerate(logfile):
            line_dict = dictify_logline(line)
            self.loglines_store.append([i + 1, line_dict['remote_host'],
                    line_dict['status'], int(line_dict['bytes_sent']), line])
        logfile.close()

    def quit(self, widget, data=None):
        """stop the main gtk event loop"""
        gtk.main_quit()


    def main(self):
        """start the gtk main event loop"""
        gtk.main()

if __name__ == "__main__":
    l = ApacheLogViewer()
    l.main()
```

在PyGTK Apache日志浏览器示例中，主类ApacheLogViewer仅继承自object。这对于main对象没什么特别的，它只是碰巧是放置所有的GUI代码和动作的地方。

接下来，跳到__init__()方法，这里创建了一个window对象。与之前的示例略有不同，

示例中指定了窗口所需的大小。我们初始指定该窗口大小为640×480，然后定义它可以被最大化。两次设置大小参数是有目的的。虽然640×480是一个合理的尺寸（初始大小设为640×480是合理的，绝不是一个不恰当的默认值），但大一些或许会更好，因此设置窗口能够最大化。综上所述，首先设置为640×480（或是其他一些你喜欢的大小）或许是一个好的经验。根据PyGTK文档说明，窗口管理器不会响应maximize()请求。同时，用户在最大化窗口之后可以取消最大化，你或许希望定义取消最大化后的窗口大小。

在创建窗口并设置大小之后，创建VBox。VBox是“垂直滚动条”，这是非常简单的容器对象。GTK具有垂直（VBox）和水平（HBox）滚动条的概念，可以在窗口上进行设置。在滚动条思想背后是你可以拋动滑块来确定距离起始点（对VBox是上部，对于HBox是左侧）或结尾处的相对位置。如果你不知道某个工具具体是什么，只需要知道它也是类似按钮或文本框这样简单的GUI组件即可。通过使用这些工具，可以在窗口上按你的想象进行布置。由于box是容器，可以包括其他box，因此可以自由地将一个box放入另一个box。

在窗口中添加了VBox之后，添加工具栏和工具按钮。工具栏本身就是一个容器，可以为添加到其中的组件提供方法。我们创建为按钮使用的图标、创建按钮、添加按钮的事件句柄。最后，添加按钮到工具栏中。就像使用VBox中的pack_start()一样，我们使用insert()向工具栏中添加按钮。

接下来，创建一个文件选择器组件（这样可以导航到日志文件以进行处理），然后将它关联到事件句柄。这部分内容非常简单，但是下文中还是会重新提及。

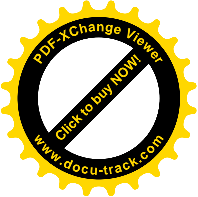
在创建了文件选择器之后，我们创建一个列表组件，列表中包括每一行日志。这个组件分两部分：数据部分（这是列表视图ListStore），以及进行交互的部分（这是树状视图TreeView）。首先通过定义希望在列表中出现的数据类型，来创建数据部分。接下来，创建显示组件，并将数据组件与之相关联。

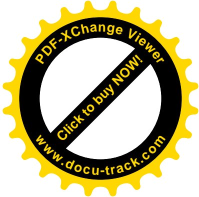
在创建了列表组件之后，我们创建最后一个容器——一个滚动窗口，然后将所有组件组合在一起。将工具条、文件选择器和滚动窗口放到之前创建的VBox中。将列表（这包括日志行）放到滚动窗口中，这样如果有太多的行，可以进行滚动。

最后，设置可见和不可见。使用show_all()调用将主窗口设置为可见。这个调用也使得所有子组件可见。假如在创建的GUI应用程序中，希望文件选择器不可见，直到点击了“open”按钮。那么在创建时，应将文件选择器组件设置为不可见。

当启动这一应用程序时，可以看到它满足了初始的需要。我们能够选择并打开指定的日志文件。行号、远端主机名、状态以及数据的字节数分别在列表控制的每一列中显示。

www.TopSage.com





因此可以很容易地通过浏览每一行来大体了解这些数据，并且可以通过简单地点击相应列的标题对这些列进行排序。

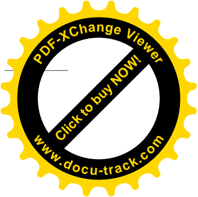
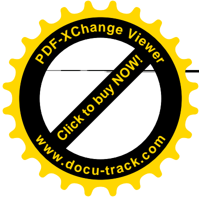
使用Curses创建Apache日志浏览器

curses是一个库，为基于文本的创建交互式的应用程序提供了方便。与GUI工具不同，curses不会遵循事件句柄和回调方法。你可以负责从多个用户获得输入，然后对输入进行处理。然而在GTK中，组件句柄从用户获得输出，当事件发生时，组件会调用句柄函数。在curses与GUI之间的另一个不同之处是：使用GUI工具，组件被添加到容器中，由GUI工具处理绘制和刷新屏幕；使用curses，则是典型地直接在屏幕上绘制文本。

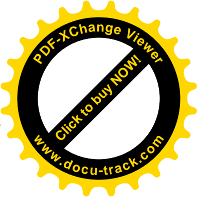
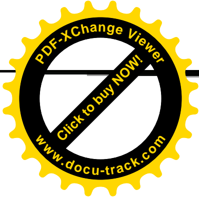
例11-3也是Apache日志浏览器，该浏览器使用Python标准库中的curses模块实现。

例11-3：curses Apache日志浏览器

```
#!/usr/bin/env python

"""
curses based Apache log viewer

Usage:

    curses_log_viewer.py logfile

This will start an interactive, keyboard driven log viewing application. Here
are what the various key presses do:

    u/d     - scroll up/down
    t       - go to the top of the log file
    q       - quit
    b/h/s   - sort by bytes/hostname/status
    r       - restore to initial sort order

"""

import curses
from apache_log_parser_regex import dictify_logline
import sys
import operator

class CursesLogViewer(object):
    def __init__(self, logfile=None):
        self.screen = curses.initscr()
        self.curr_topline = 0
        self.logfile = logfile
        self.loglines = []

    def page_up(self):
        self.curr_topline = self.curr_topline - (2 * curses.LINES)
        if self.curr_topline < 0:
            self.curr_topline = 0
        self.draw_loglines()
```

```
    def page_down(self):
        self.draw_loglines()

    def top(self):
        self.curr_topline = 0
        self.draw_loglines()

    def sortby(self, field):
        #self.loglines = sorted(self.loglines, key=operator.itemgetter(field))
        self.loglines.sort(key=operator.itemgetter(field))
        self.top()

    def set_logfile(self, logfile):
        self.logfile = logfile
        self.load_loglines()

    def load_loglines(self):
        self.loglines = []
        logfile = open(self.logfile, 'r')
        for i, line in enumerate(logfile):
            line_dict = dictify_logline(line)
            self.loglines.append((i + 1, line_dict['remote_host'],
                line_dict['status'], int(line_dict['bytes_sent']), line.rstrip()))
        logfile.close()
        self.draw_loglines()

    def draw_loglines(self):
        self.screen.clear()
        status_col = 4
        bytes_col = 6
        remote_host_col = 16
        status_start = 0
        bytes_start = 4
        remote_host_start = 10
        line_start = 26
        logline_cols = curses.COLS - status_col - bytes_col - remote_host_col - 1
        for i in range(curses.LINES):
            c = self.curr_topline
            try:
                curr_line = self.loglines[c]
            except IndexError:
                break
            self.screen.addstr(i, status_start, str(curr_line[2]))
            self.screen.addstr(i, bytes_start, str(curr_line[3]))
            self.screen.addstr(i, remote_host_start, str(curr_line[1]))
            #self.screen.addstr(i, line_start, str(curr_line[4])[logline_cols])
            self.screen.addstr(i, line_start, str(curr_line[4]), logline_cols)
            self.curr_topline += 1
        self.screen.refresh()

    def main_loop(self, stdscr):
        stdscr.clear()
        self.load_loglines()
        while True:
            c = self.screen.getch()
            try:
                c = chr(c)
```

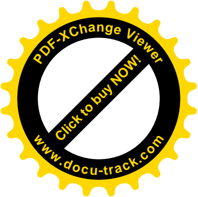



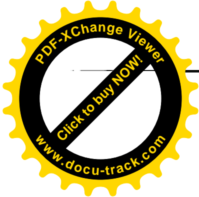
```
            except ValueError:
                continue
            if c == 'd':
                self.page_down()
            elif c == 'u':
                self.page_up()
            elif c == 't':
                self.top()
            elif c == 'b':
                self.sortby(3)
            elif c == 'h':
                self.sortby(1)
            elif c == 's':
                self.sortby(2)
            elif c == 'r':
                self.sortby(0)
            elif c == 'q':
                break

if __name__ == '__main__':
    infile = sys.argv[1]
    c = CursesLogViewer(infile)
    curses.wrapper(c.main_loop)
```

在例11-3中，为使代码结构化，创建了一个单独的类 —— CursesLogViewer。在构造器中，创建了一个curses screen并初始化一些变量。我们在程序的main中实例化CursesLogViewer，并传递希望查阅的日志文件。为了浏览和选择文件，可以在应用程序中设置一个选项，但是它比在PyGTK中实现的日志浏览器更复杂。用户将会在一个shell中运行应用程序，除了从命令行进行文件导航并需要在应用程序启动时进行传递之外没有什么不一样。在初始化CursesLogViewer之后，我们传递main_loop()方法到curses函数wrapper()中。curses函数wrapper()将终端设置为适于使用curses应用的状态，然后调用函数并在返回之前将终端设置回正常模式。

main_loop()方法作为事件循环，等待用户从键盘输入。当用户通过键盘输入时，该循环分配适当的方法（或是至少是适当的行为）对输入进行处理。按下u或d键将通过调用page_up()或是page_down()方法分别实现向上或向下滚动屏幕。page_down()方法简单地调用draw_loglines()，在终端上绘制日志行，且从当前的顶行开始绘制。随着每一行被绘制到屏幕上，当前的顶行移动到下一日志行。由于draw_loglines()仅绘制能容纳到屏幕中的行数，当下一次调用时，它会在屏幕的顶端开始绘制接下来的日志行。因此重复调用draw_loglines()将有屏幕滚动浏览整个文件的视觉效果。page_up()方法将设置当前顶行为前两页，然后通过调用draw_loglines()重绘日志行。这会有向上滚动屏幕的效果。在page_up()中设置当前顶行为前两页的原因是当我们绘制某一页时，当前顶行默认是在当前屏幕的底端。对于设置向下滚动，方法类似。

下面介绍排序。构建排序函数，可以根据主机名、状态以及一次请求中发送的字节数进





行排序。激活任何一个排序类型都会调用sortby()。sortby()方法为CursesLogViewer对象在指定的字段上对日志行列表进行排序，然后调用top()方法。top()方法设置当前顶行为日志列表的第一行，然后绘制日志行的下一页（这将会是第一页）。

应用程序的最后一个事件句柄是quit。quit方法简单地停止事件循环，让main_loop()方法返回到curses wrapper()函数，以进一步进行终端清理。

对于PyGTK应用和curses应用，其代码的行数相当，但curses应用感觉需要更多的代码。其原因也许是由于不得不创建自己的事件循环所引起；或是不得不在某种意义上创建自己的widget；或是其直接在终端屏幕上进行绘制需要做更多的工作。但是，当你掌握了如何将curses应用合并在一起，就会快许多了。

图11-3显示了curses日志浏览器按传输的字节数进行排序的记录。在这个应用中做的一个改进是可以对当前排序方法的结果进行反序。这是一个非常简单的修改，留给读者自己完成。另一个改进是滚动时可以查看整个日志行的内容。这也应该是一个比较简单的修改，也将其作为练习留给读者自己完成。

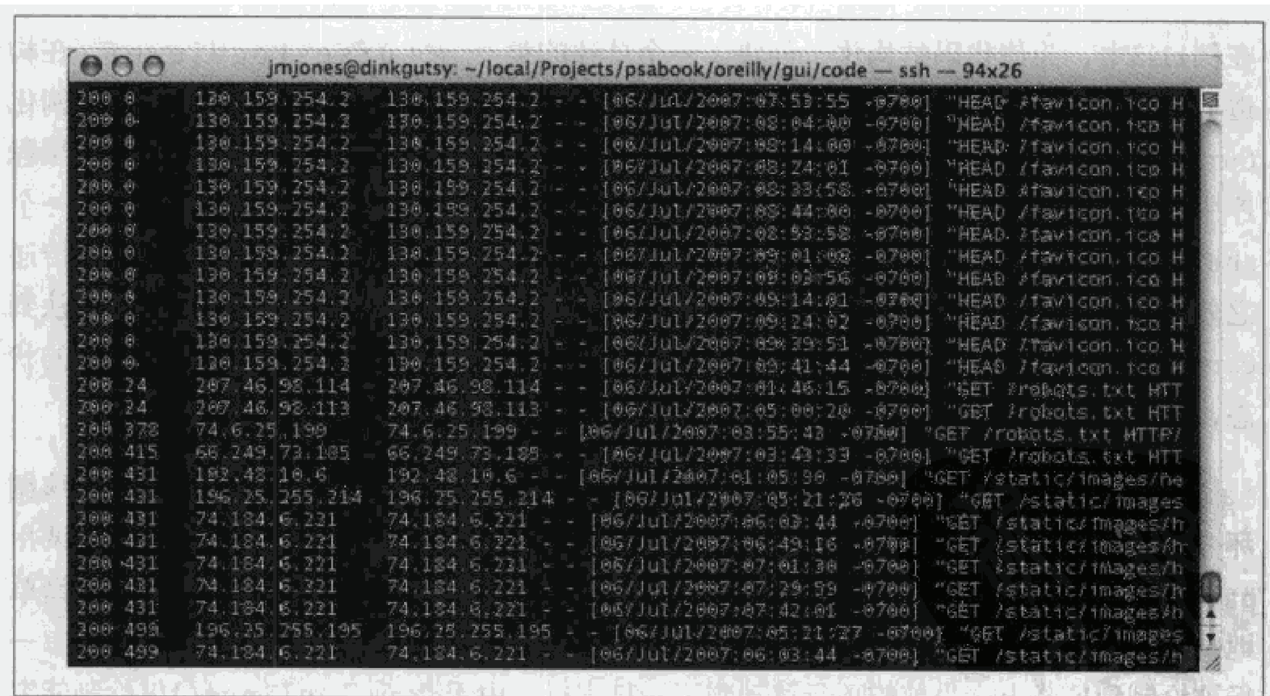
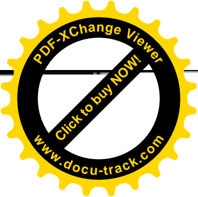
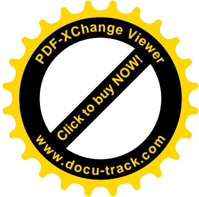


图11-3：Apache日志列表

Web应用

如果用“巨大”来形容Web还是有些保守。Web中充满了人们每天需要依赖的应用程序。为什么Web上有这么多应用程序可用？首先，web应用程序是普遍可用的。这表示

www.TopSage.com

当web应用被部署之后，希望使用它的任何人都可以在浏览器中输入它的URL，然后使用它。用户没必要下载并安装任何东西，除了一个浏览器（通常这是早已经安装过了），除非你正在使用浏览器插件，例如Flash。这一特点之所以吸引人是因为其为用户考虑。第二，web应用为所有用户提供潜在的单方面升级。

这表示一方（应用程序的所有者）可以升级整个应用程序，而不需要其他方面（用户方）做任何操作。当不需要依赖于用户当前的环境特征时，这实际上是非常有用的。例如，你对Flash的升级仅依赖于其新版本的某个特征，而不再是当前用户端需要安装什么。这一优点或许很快就会大行其道，而且这是一个对双方都很有吸引力优点，尽管用户方很少意识到这一点 。另外，浏览器是非常普通的部署平台。虽然有一些跨浏览器的兼容问题存在，但是对于大多数用户，如果不是正在使用特殊的插件，一个工作在某种操作系统上的某个浏览器上的web应用程序，通常是可以工作在别的操作系统的其他浏览器中。这对双方都有吸引力。在开发方需要多做一些工作才能让应用在多浏览器环境中顺利工作，而用户会对其选择的应用非常满意。

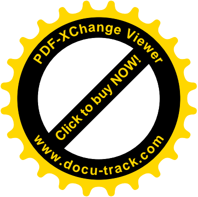
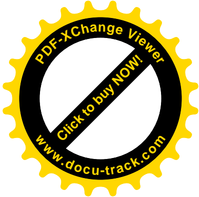
那么作为一名系统管理员，又有哪些是相关的呢？根据生成普通GUI应用与生成web应用的特点，我们列出了所有的原因。对于系统管理员，web应用的第一个好处是web应用可以访问文件系统，并且处理运行它的主机表。这一特殊的web应用功能使得web应用成为系统、应用、用户监测和报告机制中非常不错的选择。而这些类的问题都是在系统管理员的管理范围之内的。

你很有希望体会到这些好处，尽管或许创建一个web应用来服务自己或别人仅是偶然的事情。那么你能够创建一个什么样的web应用呢？由于本书是与Python相关的书，我们当然会建议一个Python解决方案。但是哪一个呢？对Python的批评之一是它有许多不同的web应用框架，就像一年有几百天一样。这一点，最主要的四个选择通常是TurboGears、Django、Pylons和Zope。它们都有自己的优点，但是我们觉得Django尤其适合本书的主题。

Django

Django是一个完整的协议栈网络应用框架。它包含一个模板系统，使用相关对象映射的数据库连接，当然还有可以为应用编写逻辑代码的Python自身。与“完整协议栈”框架相关的是，Django也遵循Model-View-Template（MVT，模型－视图－模板）方法。这种Model-View-Template方法与通常被称为Model-View-Controller（MVC）的方法即使不是相同，也是十分相似的，这两者都是开发应用的方法。数据库代码被分割为一部分，与两种方法中的“模型”相对应。业务逻辑被分割为一部分，与MVT中的“视图”以及MVT中的“控制器”相对应。业务表示被分割为一部分，与MVT中的“模板”以及MVC中的“视图”相对应。

www.TopSage.com

Apache日志视图应用

在接下来的示例中包括了一些代码段，我们将创建一个Apache日志浏览器，与之前使用PyGTK所创建的相类似。我们计划直接打开日志文件，并允许用户查看及排序，因为确实不需要数据库，所以这个示例中没有使用数据库连接。在进一步介绍示例代码之前，我们向你展示如何在Django中创建一个项目以及应用。

可以从*http://www.djangoproject.com/*下载Django代码。在写这本书时，最新的版本是0.96。这是建议安装的版本，来自开发的主版本。一旦你已经下载，使用常规的"python setup.py install"命令即可进行安装。安装之后，在*site-packages*目录中将具有Django库，在*scritps*目录中有一个名为django-admin.py的脚本。在典型的*nix系统上，*scripts*目录将是与python可执行文件是相同的目录。

在安装了Django之后，需要创建一个项目和一个应用程序。项目包括一个或多个应用，通常会作为配置中心，针对你创建的全部的web应用（不要与Django应用相混淆，Django应用是小一些的功能代码段，可以在不同的项目中被重用）。对于Apache日志浏览应用，通过运行"django-admin.py startproject dj_apache"来创建一个称为dj_apache的项目。该步创建了一个目录和一些文件。例11-4是新项目的树状视图。

例11-4：Django项目的树视图

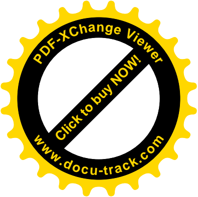
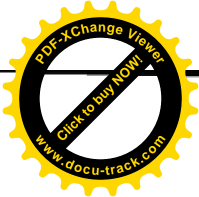
```
jmjones@dinkbuntu:~/code$ tree dj_apache
dj_apache
|-- __init__.py
|-- manage.py
|-- settings.py
`-- urls.py

0 directories, 4 files
```

现在有了一个项目，接下来创建一个应用。首先切换到*dj_apache*目录，然后使用"django-admin.py starapp logview"来创建一个应用。这将在*dj_apache*目录中创建一个logview目录和一些文件。例11-5是树状视图，可以显示我们拥有的所有文件和目录。

例11-5：Django 应用的树视图

```
jmjones@dinkbuntu:~/tmp$ tree dj_apache/
dj_apache/
|-- __init__.py
|-- logview
|   |-- __init__.py
|   |-- models.py
|   `-- views.py
|-- manage.py
|-- settings.py
`-- urls.py
```


www.TopSage.com

可以看到应用程序目录（logview）包括models.py和views.py。Django遵守MVT约定，因此这些文件帮助将整个应用分解为相应的组件。文件models.py包括数据库层，因此它归入MVT中的模块组件。vies.py包含应用逻辑，因此归入MVT的视图组件。模板组件包括全部应用的表示层。有一些方法可以让Django看到我们的模板，对于例11-6，在*logview*目录下创建一个*templates*目录。

例11-6：添加模板目录

```
jmjones@dinkbuntu:~/code$ mkdir dj_apache/logview/templates
jmjones@dinkbuntu:~/code$ tree dj_apache/
dj_apache/
|-- __init__.py
|-- logview
|   |-- __init__.py
|   |-- models.py
|   |-- templates
|   `-- views.py
|-- manage.py
|-- settings.py
`-- urls.py

2 directories, 7 files
```

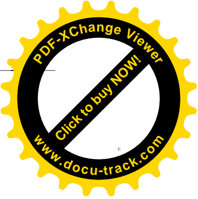
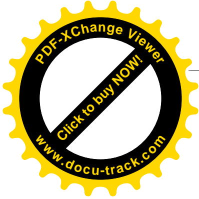
现在，准备开始实现我们的应用。需要我们首先做的事情是决定我们的URL如何工作。这是非常基本的应用，因此URL将是非常简单的。我们希望列出日志文件，可以进行浏览。由于我们的功能是简单而有限的，使用“/”列出需要打开的日志，使用"/viewlog/some_sort_method/some_log_file"根据指定的排序方法查看指定的日志文件。为了将URL与某些活动建立关联，必须在项目的顶级目录中升级*urls.py*文件。例11-7是为我们的日志浏览器应用使用的urls.py。

例11-7：Django URL配置 (urls.py)

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^$', 'dj_apache.logview.views.list_files'),
    (r'^viewlog/(?P<sortmethod>.*?)/(?P<filename>.*?)/$',
      'dj_apache.logview.views.view_log'),
)
```

URL配置文件是非常清楚且极为简单的。配置文件极为依赖正则表达式，通过正则表达式来映射URL，该URL能够匹配指定的正则表达式到一个能够准确地匹配字符串的视图函数。这里映射URL的“/”到函数“dj_apache.logview.views.list_files”。也映射所有的URL匹配正则表达式“^viewlog/(?P<sortmethod>.*?)/(?P<filename>.*?)/$”到视图函数"dj_apache.logview.views.view_log"。当一个浏览器连接到一个Django应用并且为某一资源发送一个请求时，Django查看url.py，寻找正则表达式匹配URL的元素，然后发送请求到匹配的视图函数。

www.TopSage.com

例11-8中的源文件包括这一应用的视图函数以及实用函数。

例11-8：Django视图模式（views.py）

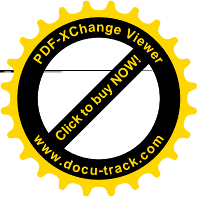
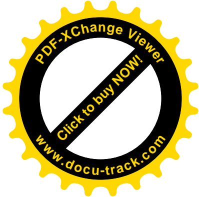
```
# Create your views here.

from django.shortcuts import render_to_response

import os
from apache_log_parser_regex import dictify_logline
import operator

log_dir = '/var/log/apache2'

def get_log_dict(logline):
    l = dictify_logline(logline)
    try:
        l['bytes_sent'] = int(l['bytes_sent'])
    except ValueError:
        bytes_sent = 0
    l['logline'] = logline
    return l

def list_files(request):
    file_list = [f for f in os.listdir(log_dir) if
                os.path.isfile(os.path.join(log_dir, f))]
    return render_to_response('list_files.html', {'file_list': file_list})

def view_log(request, sortmethod, filename):
    logfile = open(os.path.join(log_dir, filename), 'r')
    loglines = [get_log_dict(l) for l in logfile]
    logfile.close()
    try:
        loglines.sort(key=operator.itemgetter(sortmethod))
    except KeyError:
        pass
    return render_to_response('view_logfile.html', {'loglines': loglines,
                                                  'filename': filename})
```

list_files()函数列出由log_dir文件指定的目录中的所有文件，并且传递列表到list_files.html模板中。这是在list_files()函数中真正发生的事情。这个函数可以通过修改log_dir的值来进行配置。另一个用于配置的可选方法是将日志目录放到数据库中。如果选择在数据库中放入日志目录的值，无须重新启动应用就能对值进行修改。

函数view_log()接受的参数包括：排序方法和日志文件名。这两个参数利用*urls.py*文件中使用的正则表达式从URL中提取。在*urls.py*中为排序方法和文件名命名正则表达式组，但是也没必要必须这么做。参数传递给视图函数，该视图函数来自URL且与它们各自组中的序列相同。在URL正则表达式中使用命名组是一个好的经验，这样你可以很容易地说出你从URL中提取的参数以及URL是什么。

view_log()函数打开日志文件，该文件名来自URL。然后view_log()使用从之前示例

www.TopSage.com

中得到的Apache日志解析库来转换第一个日志行到一个元组中，该元组包括status、remote host、 bytes_sent和日志行本身。接下来，view_log()根据从URL中传递达来的排序方法，排序元组列表。最后，view_log()传递这个列表到view_logfile.html模板中进行格式化。

最后剩下的事情是创建模板，该模板是视图函数提供的。在Django中，模板可以继承自其他的模板，因此可以改进代码复用，使编码简化，并建立统一样式的页面。我们将要创建的第一个模板是其他另两个模板将要进行继承的。这个模板将为该应用中的另外两个模板设置一个普通的样式。这也就是为什么我们从它开始的原因。这个文件是base.html。参见例11-9。

例11-9：Django的基本模板(base.html)

```
<html>
    <head>
        <title>{% block title %}Apache Logviewer - File Listing{% endblock %}</title>
    </head>
    <body>
        <div><a href="/">Log Directory</a></div>
        {% block content %}Empty Content Block{% endblock %}
    </body>
</html>
```

这是一个非常简单基本的模板。这或许是你见到的最简单的HTML页面。仅有的有意义的元素是两个“块”：“内容”和“标题”。当你在一个父模板中定义一个块时，一个有其自己内容的子模板可以重载父块。这允许你基于页面部分设置默认的内容，并且允许子模板重载默认设置。“标题”块允许子页面设置值，该值会在页面的标题标签中显示。“内容”块是一个通常的约定，约定升级页面的“main”块时允许页面的其他部分保持不变。

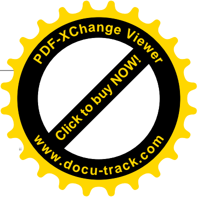
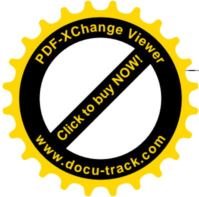
例11-10是一个模板，该模板简单地列出了指定目录中的所有文件。

例11-10：Django 文件列表模板（list_files.html)

```
{% extends "base.html" %}

{% block title %}Apache Logviewer - File Listing{% endblock %}

{% block content %}
<ul>
{% for f in file_list %}
  <li><a href="/viewlog/linesort/{{ f }}/" >{{ f }}</a></li>
{% endfor %}
</ul>
{% endblock %}
```

图11-14展示了文件列表页面的样式。在这个模板中，我们扩展了base.html。这允许我们

www.TopSage.com

获得在base.html中定义的所有内容，并且可以将代码加入到任何指定的代码块中，并重载它们的行为。我们准确地使用"标题"和"内容"块来进行操作。在"内容"块中，循环一个变量file_list，该变量被传递给模板。对于file_list中的每一个元素，创建一个链接，通过该链接打开日志文件并进行解析。

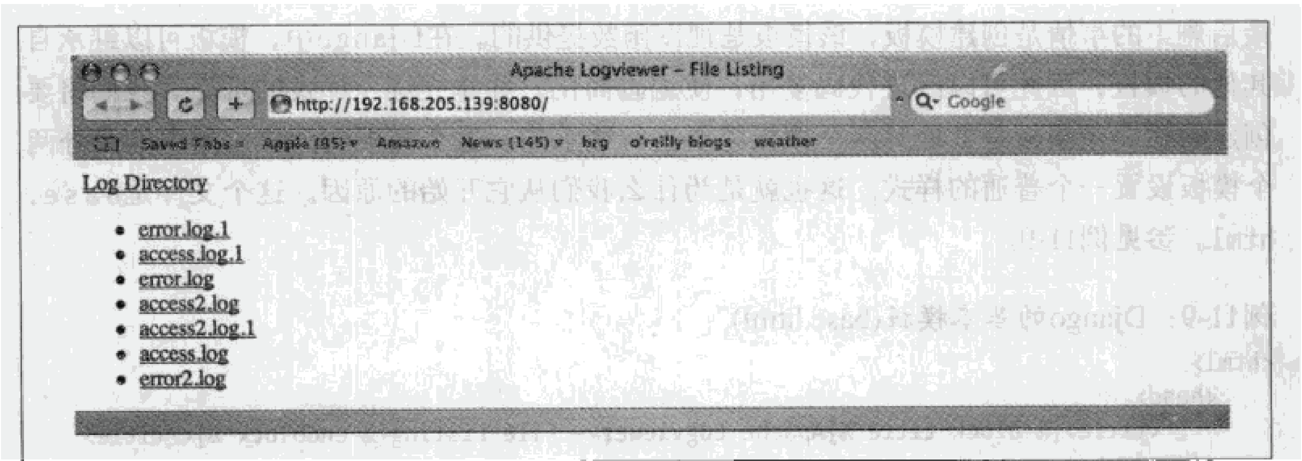


图11-4：Apache日志列表

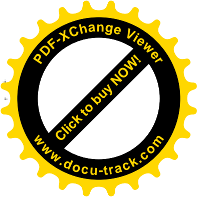
在例11-11中的模板负责创建页面，之前例10-11中的链接可以将用户链接到此页面。该页面显示了指定日志文件的细节。

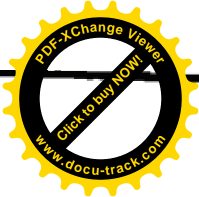
例11-11：Django文件列表模板（view_log file.html）

```
{% extends "base.html" %}

{% block title %}Apache Logviewer - File Viewer{% endblock %}

{% block content %}
<table border="1">
    <tr>
        <td><a href="/viewlog/status/{{ filename }}/">Status</a></td>
        <td><a href="/viewlog/remote_host/{{ filename }}/">Remote Host</a></td>
        <td><a href="/viewlog/bytes_sent/{{ filename }}/">Bytes Sent</a></td>
        <td><a href="/viewlog/linesort/{{ filename }}/">Line</a></td>
    </tr>
{% for l in loglines %}
    <tr>
        <td>{{ l.status }}</td>
        <td>{{ l.remote_host }}</td>
        <td>{{ l.bytes_sent }}</td>
        <td><pre>{{ l.logline }}</pre></td>
    </tr>
{% endfor %}
</table>
{% endblock %}
```

例11-11所示的模板继承自之前介绍的base模板，并且在"内容"区创建了一个表格。表格标题详细说明了每一列的内容：状态、远端主机地址、发送字节数和日志内容。除了

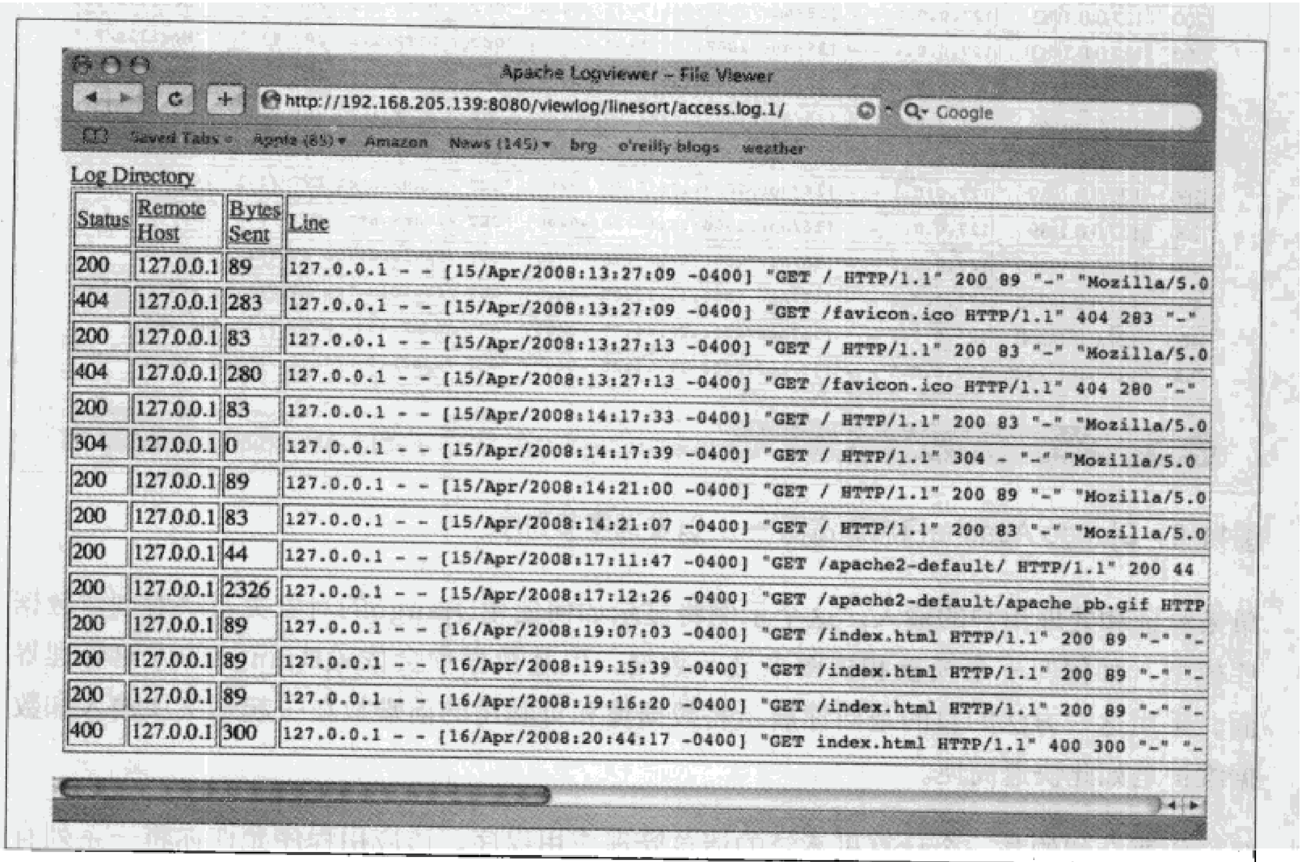
www.TopSage.com





详细列出每列的内容外，标题允许用户指定如何对日志文件进行排序。例如，如果一个用户点击了“Bytes Sent”列标题（这是一个简单的链接），页面会重载并且视图中将依据“bytes sent”对日志行进行排序。单击任何一列的标题（除了“Line”），都会依据该列以升序进行排序。单击“Line”将日志行回到它原始的顺序。

图11-5显示了应用程序在未排序情况下的样式，图11-6显示了以发送字节数排序后的样式。



Status	Remote Host	Bytes Sent	Line
200	127.0.0.1	89	127.0.0.1 - - [15/Apr/2008:13:27:09 -0400] "GET / HTTP/1.1" 200 89 "-" "Mozilla/5.0
404	127.0.0.1	283	127.0.0.1 - - [15/Apr/2008:13:27:09 -0400] "GET /favicon.ico HTTP/1.1" 404 283 "-"
200	127.0.0.1	83	127.0.0.1 - - [15/Apr/2008:13:27:13 -0400] "GET / HTTP/1.1" 200 83 "-" "Mozilla/5.0
404	127.0.0.1	280	127.0.0.1 - - [15/Apr/2008:13:27:13 -0400] "GET /favicon.ico HTTP/1.1" 404 280 "-"
200	127.0.0.1	83	127.0.0.1 - - [15/Apr/2008:14:17:33 -0400] "GET / HTTP/1.1" 200 83 "-" "Mozilla/5.0
304	127.0.0.1	0	127.0.0.1 - - [15/Apr/2008:14:17:39 -0400] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0
200	127.0.0.1	89	127.0.0.1 - - [15/Apr/2008:14:21:00 -0400] "GET / HTTP/1.1" 200 89 "-" "Mozilla/5.0
200	127.0.0.1	83	127.0.0.1 - - [15/Apr/2008:14:21:07 -0400] "GET / HTTP/1.1" 200 83 "-" "Mozilla/5.0
200	127.0.0.1	44	127.0.0.1 - - [15/Apr/2008:17:11:47 -0400] "GET /apache2-default/ HTTP/1.1" 200 44
200	127.0.0.1	2326	127.0.0.1 - - [15/Apr/2008:17:12:26 -0400] "GET /apache2-default/apache_pb.gif HTTP
200	127.0.0.1	89	127.0.0.1 - - [16/Apr/2008:19:07:03 -0400] "GET /index.html HTTP/1.1" 200 89 "-" "-
200	127.0.0.1	89	127.0.0.1 - - [16/Apr/2008:19:15:39 -0400] "GET /index.html HTTP/1.1" 200 89 "-" "-
200	127.0.0.1	89	127.0.0.1 - - [16/Apr/2008:19:16:20 -0400] "GET /index.html HTTP/1.1" 200 89 "-" "-
400	127.0.0.1	300	127.0.0.1 - - [16/Apr/2008:20:44:17 -0400] "GET index.html HTTP/1.1" 400 300 "-" "-

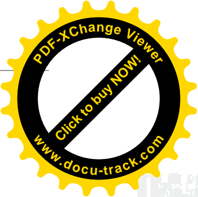
图11-5：Django Apache日志浏览器——原始顺序

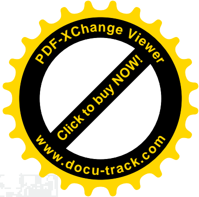
这是非常简单的使用Django生成的应用。事实上，这也是非常典型的应用。绝大多数Django应用将被连接到某种类型的数据库。这里我们还进一步做了一些改进，包括：以逆序排序所有的字段、根据指定的状态代码或是远端主机名进行过滤、根据大于或小于指定的发送字节数进行过滤、进行合并过滤、在其上添加AJAXy。这里不再对这些改进进行介绍，留作读者的练习。

简单的数据库应用

我们曾提到，之前的Django示例从Django应用规范演变而来，所以没有使用数据库。下面的示例将更符合人们使用Django的需要，两者只是关注点略有不同。当人们生成一个Django应用，进行数据库连接时，他们经常写模板来展示来自数据库的数据，也使用表

www.TopSage.com





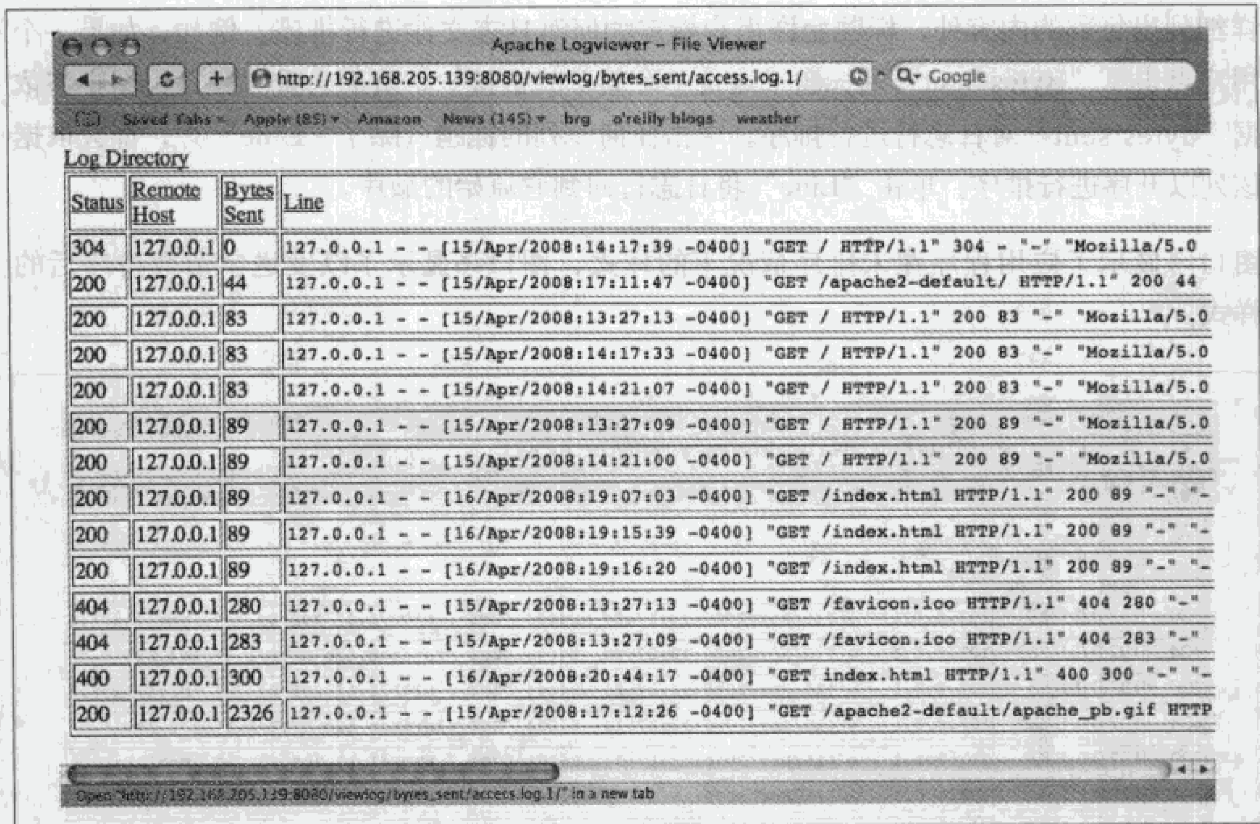


图11-6：Django Apache日志浏览器——以发送字节为序

单来验证和处理用户的输入。这个示例将显示如何使用Django的对象关系映射创建数据库模型，如何写一个模板和视图来显示数据，但是数据项会依赖Django的内建管理界面。采用这一方法的目的是向你展示将数据库和可利用的前端放在一起来实现输入和数据维护是如此快速简便。

我们将要介绍的是一个计算机系统的清单管理应用程序。该应用程序允许你将一系列有关计算机的描述，包括相关的IP地址，启动了哪些服务，服务器的硬件构成等。

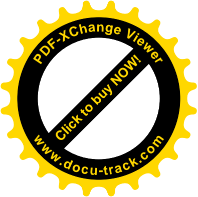
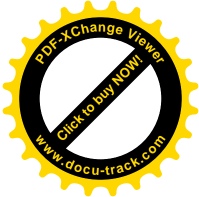
我们将要遵循相同的步骤来创建这个Django项目和应用，就像之前的Django示例一样。下面是创建项目的命令以及使用django-admin命令行工具的应用。

```
jmjones@dinkbuntu:~/code$ django-admin startproject sysmanage
jmjones@dinkbuntu:~/code$ cd sysmanage
jmjones@dinkbuntu:~/code/sysmanage$ django-admin startapp inventory
jmjones@dinkbuntu:~/code/sysmanage$
```

这创建了与基于Django的Apache日志浏览器相同的目录排序结构。以下是一个树状视图，可以浏览我们创建的目录和文件。

```
jmjones@dinkbuntu:~/code/sysmanage$ cd ../
jmjones@dinkbuntu:~/code$ tree sysmanage/
sysmanage/
```


www.TopSage.com

```
|-- __init__.py
|-- inventory
|    |-- __init__.py
|    |-- models.py
|    `-- views.py
|-- manage.py
|-- settings.py
`-- urls.py
```

在创建项目和应用之后，我们需要配置希望连接的数据库。SQLite是一个不错的选择，尤其是如果你正测试或开发一个没有将其转化为产品的应用。如果更多人准备尝试该应用，建议考虑一些鲁棒性更好的数据库，如PostgreSQL。为了配置使用SQLite数据库应用程序，我们修改了项目主目录中的*settings.py*文件中的一些代码。以下是我们修改数据库配置的行：

```
DATABASE_ENGINE = 'sqlite3'
DATABASE_NAME = os.path.join(os.path.dirname(__file__), 'dev.db')
```

我们设置“sqlite3”作为数据库引擎。配置数据库的位置行（`DATABASE_NAME`选项）值得仔细理解。不同于直接指定一个到数据库文件的绝对路径，这里我们对数据库进行配置，这样它总会与*settings.py*文件在相同的目录下。`__file__`保存了*setings.py*文件的绝对路径。调用`os.path.dirname(__file__)`可以获得*setings.py*文件的目录。传递文件所在目录以及我们想去创建的数据库文件的名字到`os.path.join()`，将获得数据库文件的绝对路径，该路径适用于不同目录下的应用。这是一个非常有用的经典技巧，我们应该养成使用自己的配置文件的习惯。

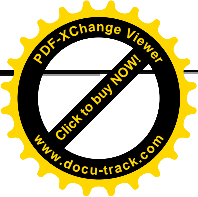
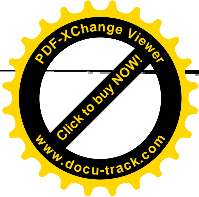
除了配置数据库之外，我们还需要包括Django管理界面以及该项目应用中需要的条目清单。以下是*setings.py*文件的相关内容：

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'sysmanage.inventory',
)
```

添加`django.contrib.admin`和`sysmanage.inventory`到安装应用的列表中。这表示当我们告诉Django创建数据库时，它将为所有包含的项目创建表格。

接下来，修改URL映射，这样该项目将包括管理界面。以下是来自URL配置文件的相关内容：

```
# Uncomment this for admin:
    (r'^admin/', include('django.contrib.admin.urls')),
```

创建urls.py的工具完成了创建，并将系统管理的界面进行了包括，但是该行需要取消注释。你可以看到我们已经简单地从包括管理URL配置文件的行的行首位置删除了“#”符号。

现在已经配置了一个数据库，添加了管理和清单应用，添加了对URL配置文件的管理界面，我们准备开始定义数据库摘要。在Django中，每一个应用有它自己的摘要定义。在每一个应用目录中（在这里的是“inventory”），有一个名为*models.py*的文件，包括应用程序将会使用的表格和列的定义。使用Django，就像许多其他的web框架一样，依赖于ORM。创建并使用一个数据库而不必写一个单独的SQL表达式是可能的。Django的ORM将类转换为表格，并且类的属性添加到这些表格的列中。例如，以下是一段代码，在配置数据库中定义一个表格（这段代码是我们将要介绍的较大示例的一部分）：

```
class HardwareComponent(models.Model):
    manufacturer = models.CharField(max_length=50)
    #types include video card, network card...
    type = models.CharField(max_length=50)
    model = models.CharField(max_length=50, blank=True, null=True)
    vendor_part_number = models.CharField(max_length=50, blank=True, null=True)
    description = models.TextField(blank=True, null=True)
```

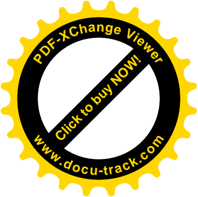
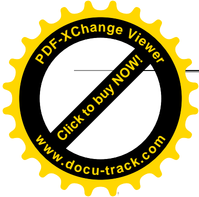
需要注意的是，HardwrarComponent类继承自Django模块类。这表示HardwareComponent类是Model类型，并且会适当地执行动作。我们为硬件组件定义一些属性：manufacturer、type、model、vendor_part_number和description。这些属性来自Django。Django不仅提供一些硬件生产厂商的列表，而且它提供了CharField类型。

在清单管理中的类定义将创建一个inventory_hardwarecomponent表，该表具有6列：id、manufacturer、type、model、vendor_part_number和 description。这与ORM类定义非常相似。当你定义了一个model类，Django将创建一个相应的表格，表格的名字是应用名（小写），然后是一个下划线，最后是小写的类名。如果你没有指定，Django将在你的表格上创建一个id列，它会作为关键字。以下是SQL表格的创建代码，对应于HardwareComponent模型：

```
CREATE TABLE "inventory_hardwarecomponent" (
    "id" integer NOT NULL PRIMARY KEY,
    "manufacturer" varchar(50) NOT NULL,
    "type" varchar(50) NOT NULL,
    "model" varchar(50) NULL,
    "vendor_part_number" varchar(50) NULL,
    "description" text NULL
)
```

如果希望查看Django使用来创建数据库的SQL，在项目目录中简单地运行“python manage.py sql myapp”即可，这里myapp对应于应用名。

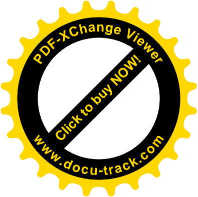
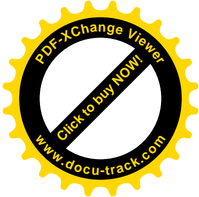
www.TopSage.com

现在已经了解了Django的ORM，接下来将介绍如何为清单管理应用程序创建数据库模型。例11-12是为清单管理应用程序创建的model.py。

例11-12：数据库设计（models.py）

```
from django.db import models

# Create your models here.

class OperatingSystem(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField(blank=True, null=True)

    def __str__(self):
        return self.name

    class Admin:
        pass

class Service(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField(blank=True, null=True)

    def __str__(self):
        return self.name

    class Admin:
        pass

class HardwareComponent(models.Model):
    manufacturer = models.CharField(max_length=50)
    #types include video card, network card...
    type = models.CharField(max_length=50)
    model = models.CharField(max_length=50, blank=True, null=True)
    vendor_part_number = models.CharField(max_length=50, blank=True, null=True)
    description = models.TextField(blank=True, null=True)

    def __str__(self):
        return self.manufacturer

    class Admin:
        pass

class Server(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField(blank=True, null=True)
    os = models.ForeignKey(OperatingSystem)
    services = models.ManyToManyField(Service)
    hardware_component = models.ManyToManyField(HardwareComponent)

    def __str__(self):
        return self.name

    class Admin:
        pass
```

```
class IPAddress(models.Model):
address = models.TextField(blank=True, null=True)
server = models.ForeignKey(Server)

def __str__(self):
    return self.address

class Admin:
    pass
```

我们为模型定义了5个类：OperatingSystem、Service、HardwareComponent、Server和IPAddress。OperatingSystem类允许我们定义不同的操作系统。我们定义该类具有name和description属性，这是我们真正需要的。创建一个OperatingSystemVendor类，并从OperatingSystem进行链接比较合适。但是出于简化及明确的考虑，我们去除了提供商关系。每一个服务器具有一个操作系统。当我们介绍服务器时，将向你展示这一关系。

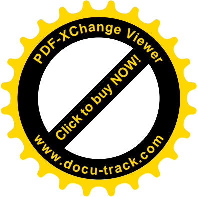
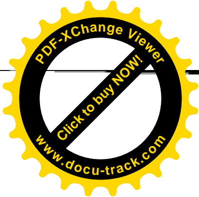
Service类允许我们列出所有可能的运行在服务器上的服务。例如包括Apache web服务器、Postfix邮件服务器、Bind DNS服务器以及OpenSSH服务器。OperatingSystem类具有name和description属性。每一服务器或许有许多服务。我们将向你展示这些类之间的关系。

HardwareComponent类代表我们服务器可能包括的所有硬件组件列表。如果你向系统中添加了硬件或是通过独立的组件建立自己的服务器，该类就非常有意义了。我们为HardwareComponent定义5个属性：manufacturer、type、model、vendor_part_number和description。我们可以为硬件制造商创建其他的类、类型，创建它们之间的关系。但是，同样出于简单原则，我们选择不创建这些关系。

Server类是这个清单管理系统的核心。每一个Server实例具有与之前三个类的单独关系。首先，我们为每一个Server指定一个名字（name）和属性描述（description）。这与曾经指定给其他类的属性是相同的。为了连接到其他类，我们必须指定Server与它们之间有什么关系。每个Server将仅具有一个操作系统，因此对OperationSystem创建一个外键关系。由于虚拟化已经很普遍了，这种类型关系意义不大。一台服务器或许有许多在其上运行的服务，每种类型的服务可能会运行在许多主机上，因此在Server与Service之间创建一个多对多的关系。同样地，每一台服务器或许有许多硬件组件，每种类型的硬件组件或许会在多个服务器上存在。因此，我们在Server与HardwareComponent之间创建另一个多对多关系。

最后，IPAddress是一个包含我们追踪的所有服务器的IP地址列表。我们最后列出这个模型，为的是强调IP地址与服务器之前的关系。我们给IPaddress指定一个属性和一个关系。address是属性并且应该按约定采用*XXX.XXX.XXX.XXX*格式。我们在IPAddress与

www.TopSage.com

Server之间创建一个外键关系，因为一个IP地址应该仅属于一个服务器。这一示例非常简单，但是它可以充分演示在Django的数据组件之间如何建立关系。

现在准备创建sqlite数据库文件。在项目目录中运行“python manage.py syncdb”，将新创建一个包含你在*settings.py*文件中设置的所有应用的表格。如果创建了auth表，也会提示你创建一个超级用户。以下是来自运行“python manage.py syncdb”的输出结果（片段）：

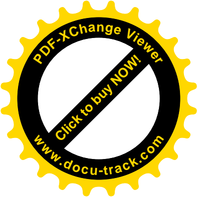
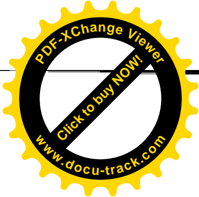
```
jmjones@dinkbuntu:~/code/sysmanage$ python manage.py syncdb
Creating table django_admin_log
Creating table auth_message
. . .
Creating many-to-many tables for Server model
Adding permission 'log entry | Can add log entry'
Adding permission 'log entry | Can change log entry'
Adding permission 'log entry | Can delete log entry'

You just installed Django's auth system, which means you don't have any
superusers defined.
Would you like to create one now? (yes/no): yes
Username (Leave blank to use 'jmjones'): E-mail address: none@none.com
Password:
Password (again): Superuser created successfully.
Adding permission 'message | Can add message'
. . .
Adding permission 'service | Can change service'
Adding permission 'service | Can delete service'
Adding permission 'server | Can add server'
Adding permission 'server | Can change server'
Adding permission 'server | Can delete server'
```

我们现在准备启动Django开发服务器，并且尝试一下管理界面。以下是启动Django开发服务器的命令以及命令产生的输出结果：

```
jmjones@dinkbuntu:~/code/sysmanage$ python manage.py runserver 0.0.0.0:8080
Validating models...
0 errors found

Django version 0.97-pre-SVN-unknown, using settings 'sysmanage.settings'
Development server is running at http://0.0.0.0:8080/
Quit the server with CONTROL-C.
```

图11-7显示了登录表单。一旦登录，就可以添加服务器、硬件、操作系统等。图11-8展示了Django管理主页面，并且图11-9展示了“添加硬件”表单。用数据库工具以连续、简单且可用方式保存并显示数据是有好处的。Django可以完成一些神奇的工作，为数据集提供简单、实用的界面。即使这些就是Django所有可能做的事情，那么它也已经是一个非常有用的工具了。但是这还仅是Django所能做的事情中起始的部分。如果你正在考虑一个方案，希望浏览器可以显示数据，那么你可以尝试让Django去完成，一般来说不会太困难。

www.TopSage.com

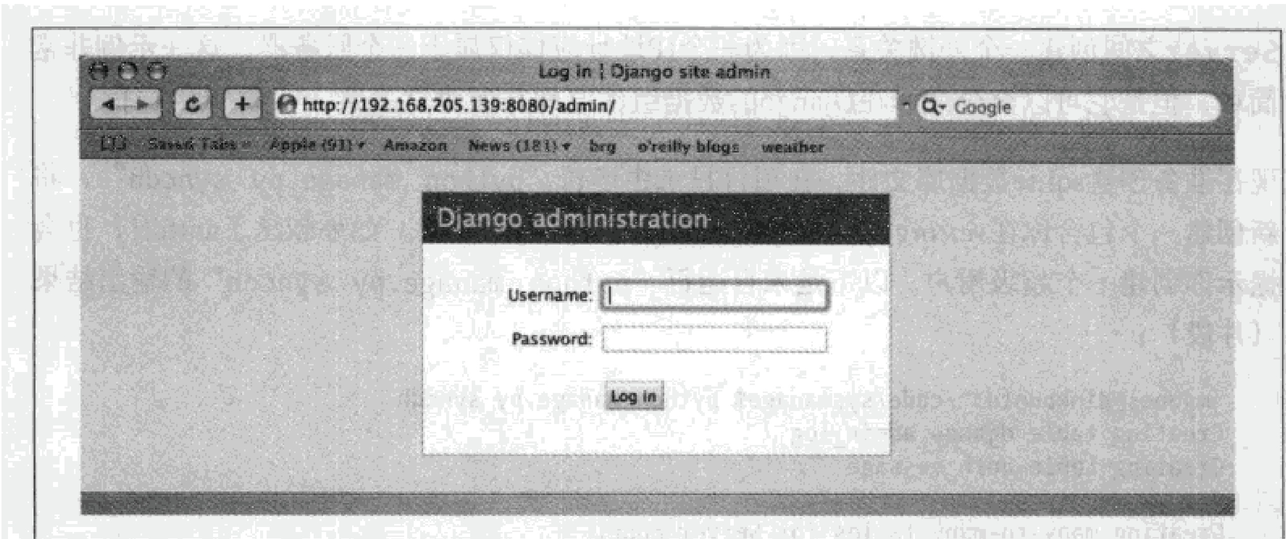


图11-7：Django系统管理登录

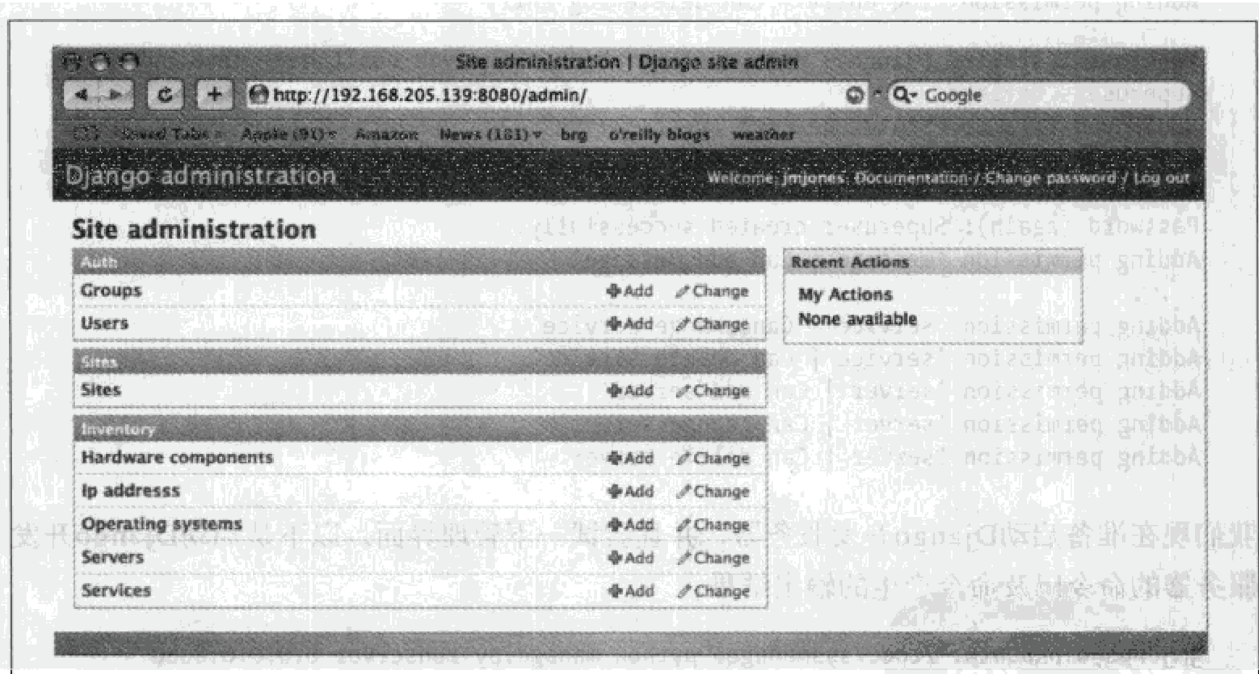
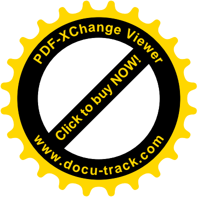


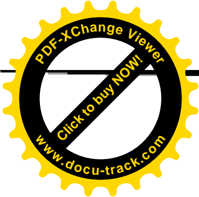
图11-8：Django管理主页面

例如，如果希望有一个具有操作系统、硬件组件、服务等每一个类型的页面，我们可以这样做。如果希望能够在每一个这些独立的条目上点击，然后显示一个仅包含具有独立特征的服务器页面，也可以做。并且如果希望能够在服务器列表的每一个条目上点击，然后显示该服务器的详细信息，同样可以做。接下来，我们使用这些“建议”，继续介绍。

首先，例11-13是一个升级的*urls.py*。

www.TopSage.com





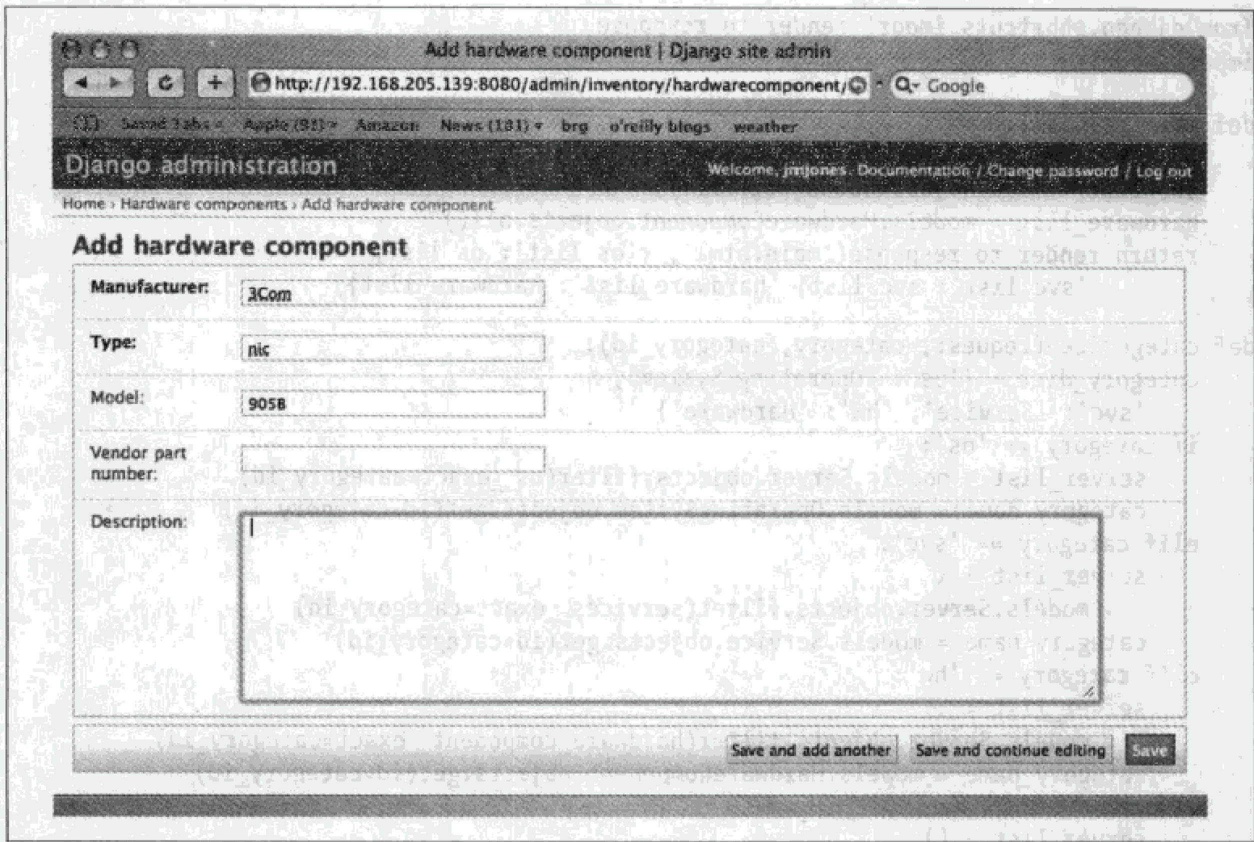


图11-9：Django系统管理添加硬件组件

例11-13：URL映射（urls.py）

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    # Example:
    # (r'^sysmanage/', include('sysmanage.foo.urls')),

    # Uncomment this for admin:
    (r'^admin/', include('django.contrib.admin.urls')),
    (r'^$', 'sysmanage.inventory.views.main'),
    (r'^categorized/(?P<category>.*?)/(?P<category_id>.*?)/$',
            'sysmanage.inventory.views.categorized'),
    (r'^server_detail/(?P<server_id>.*?)/$',
            'sysmanage.inventory.views.server_detail'),
)
```

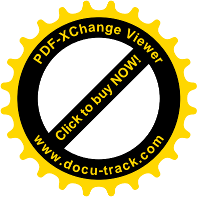
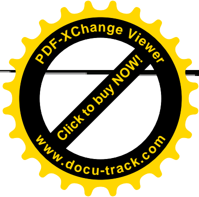
我们添加三个新行，映射非管理URL到函数。这实际与从Apache日志浏览器中看到的没什么不同。我们映射URL的正则表达式到函数，也使用了一些正则表达式组。

接下来将要做的事情是添加函数到views模块，这是在URL映射文件中声明的。例11-14是视图模块。

例11-14：清单视图（views.py）

```
# Create your views here.
```

www.TopSage.com

```
from django.shortcuts import render_to_response
import models

def main(request):
    os_list = models.OperatingSystem.objects.all()
    svc_list = models.Service.objects.all()
    hardware_list = models.HardwareComponent.objects.all()
    return render_to_response('main.html', {'os_list': os_list,
            'svc_list': svc_list, 'hardware_list': hardware_list})

def categorized(request, category, category_id):
    category_dict = {'os': 'Operating System',
        'svc': 'Service', 'hw': 'Hardware'}
    if category == 'os':
        server_list = models.Server.objects.filter(os__exact=category_id)
        category_name = models.OperatingSystem.objects.get(id=category_id)
    elif category == 'svc':
        server_list = \
            models.Server.objects.filter(services__exact=category_id)
        category_name = models.Service.objects.get(id=category_id)
    elif category == 'hw':
        server_list = \
            models.Server.objects.filter(hardware_component__exact=category_id)
        category_name = models.HardwareComponent.objects.get(id=category_id)
    else:
        server_list = []
    return render_to_response('categorized.html', {'server_list': server_list,
        'category': category_dict[category], 'category_name': category_name})

def server_detail(request, server_id):
    server = models.Server.objects.get(id=server_id)
    return render_to_response('server_detail.html', {'server': server})
```

正如添加三个URL映射到*urls.py*文件一样，我们也添加三个函数到*views.py*文件中。首先是main()。该函数简单地获取一个列表，包括所有不同的OS、硬件组件、服务，并传递它们到main.html模板。

例11-14中，我们在应用文件夹中创建了一个*templates*目录。我们将在这里做相同的事情：

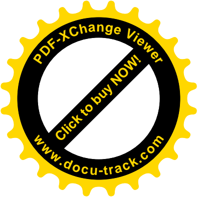
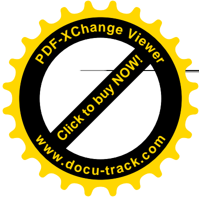
```
jmjones@dinkbuntu:~/code/sysmanage/inventory$ mkdir templates
jmjones@dinkbuntu:~/code/sysmanage/inventory$
```

例11-15是main.html模板，由main()视图函数传递数据到其中。

例11-15：main模板（main.html）

```
{% extends "base.html" %}

{% block title %}Server Inventory Category View{% endblock %}

{% block content %}
<div>
   <h2>Operating Systems</h2>
```


www.TopSage.com

```
    <ul>
    {% for o in os_list %}
      <li><a href="/categorized/os/{{ o.id }}/" >{{ o.name }}</a></li>
    {% endfor %}
    </ul>
</div>
<div>
    <h2>Services</h2>
    <ul>
    {% for s in svc_list %}
      <li><a href="/categorized/svc/{{ s.id }}/" >{{ s.name }}</a></li>
    {% endfor %}
    </ul>
</div>
<div>
    <h2>Hardware Components</h2>
    <ul>
      {% for h in hardware_list %}
      <li><a href="/categorized/hw/{{ h.id }}/" >{{ h.manufacturer }}</a></li>
    {% endfor %}
    </ul>
</div>
{% endblock %}
```

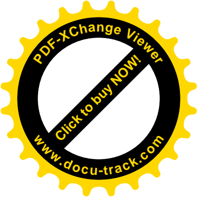
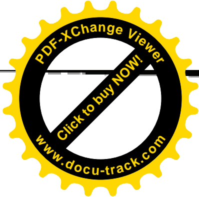
这个模板非常简单。它将页面分为三个部分，每一部分对应一个我们希望看到的类别。对于每一类别逐条列出所有条目，且每一类别条目具有一个链接，可以查看具有指定类别条目的所有服务器。当用户点击这些链接时，它会转到另一个视图函数categorized()。

main模板传递一个类别（os表示操作系统，hw表示硬件组件，svc表示服务）以及类别ID（例如，用户点击的特定组件，如“3Com 905b Network Card”）到categorized()视图函数。categorized()函数采用这些参数并从具有选定组件的数据库中获取所有服务器的列表。在对数据库进行信息查询之后，categorized()函数传递信息到“categorized.html”模板。例11-16显示了“categorized.html”模板的内容。

例11-16：分类模板 (categorized.html)

```
{% extends "base.html" %}

{% block title %}Server List{% endblock %}

{% block content %}
<h1>{{ category }}::{{ category_name }}</h1>
<div>
    <ul>
    {% for s in server_list %}
      <li><a href="/server_detail/{{ s.id }}/" >{{ s.name }}</a></li>
    {% endfor %}
    </ul>
</div>
{% endblock %}
```

www.TopSage.com

"categorized.html"模板显示了一个由categorized()传递给它的所有服务器列表。

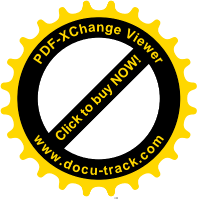
用户可以点击到独立服务器的链接，这会转到server_detail()视图函数。server_detail()视图函数取得服务器id参数，并获取数据库中对应服务器的相关数据，最后传递数据到"sever_detail.html"模板中。

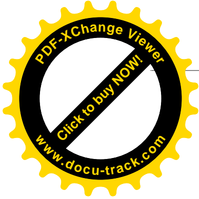
显示在例11-17中的"server_detail.html"模板或许是最长的模板，但是它非常简单。其作用是显示每一服务器的独立数据，例如，服务器上正在运行什么操作系统，服务器具有什么硬件组成，服务器上正在运行什么操作系统，以及服务器的IP地址是什么。

例11-17：服务器详细信息模板(server_detail.html)

```
{% extends "base.html" %}

{% block title %}Server Detail{% endblock %}

{% block content %}
<div>
    Name: {{ server.name }}
</div>
<div>
    Description: {{ server.description }}
</div>
<div>
    OS: {{ server.os.name }}
</div>
<div>
    <div>Services:</div>
    <ul>
    {% for service in server.services.all %}
        <li>{{ service.name }}</li>
    {% endfor %}
    </ul>
</div>
<div>
    <div>Hardware:</div>
    <ul>
    {% for hw in server.hardware_component.all %}
        <li>{{ hw.manufacturer }} {{ hw.type }} {{ hw.model }}</li>
    {% endfor %}
    </ul>
</div>
<div>
    <div>IP Addresses:</div>
    <ul>
    {% for ip in server.ipaddress_set.all %}
        <li>{{ ip.address }}</li>
    {% endfor %}
    </ul>
</div>
{% endblock %}
```

www.TopSage.com





这是一个示例，演示了使用Django如何创建一个非常简单的数据库应用。管理界面提供了一个访问数据库的友好方式，并且仅使用了少量的代码。我们能够创建自定义的排序和数据异航视图，如图11-10、图11-11和图11-12所示。

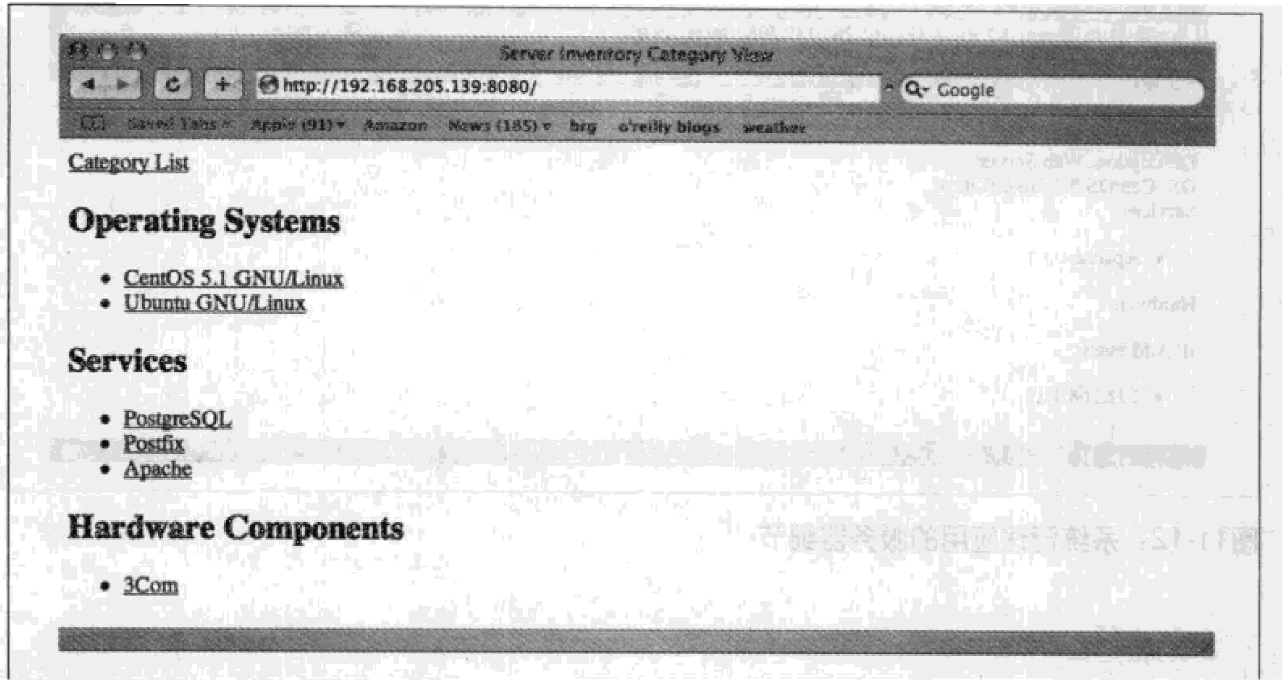


图11-10：系统管理应用主页面

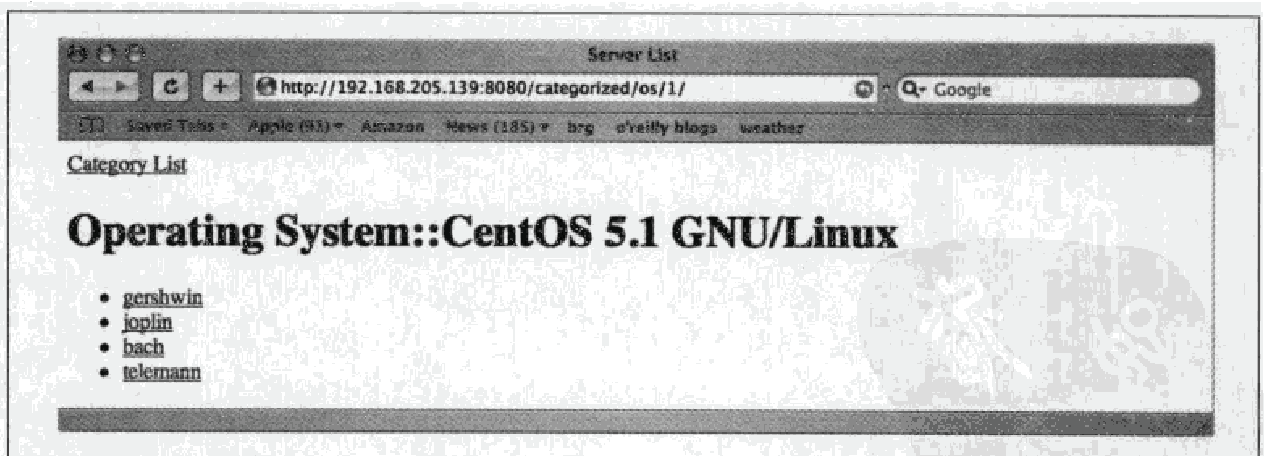
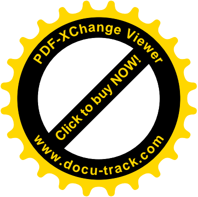
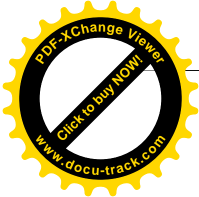


图11-11：系统管理应用的操作系统类型CentOS（分类）

本章小结

创建和使用GUI应用程序似乎不满足传统的系统管理员的相关要求，但是却可以证明这是一个非常有价值的技术。有时，你或许需要为某个用户创建一些简单的应用；其他时间，你或许需要为自己生成一个简单的应用；另一些时候，你或许意识到并不需要它，





但是它或许会将一些任务完成得更流畅。一旦你习惯了创建GUI应用，那么要不了太久，你或许就会惊奇地发现自己是如此频繁地使用它。

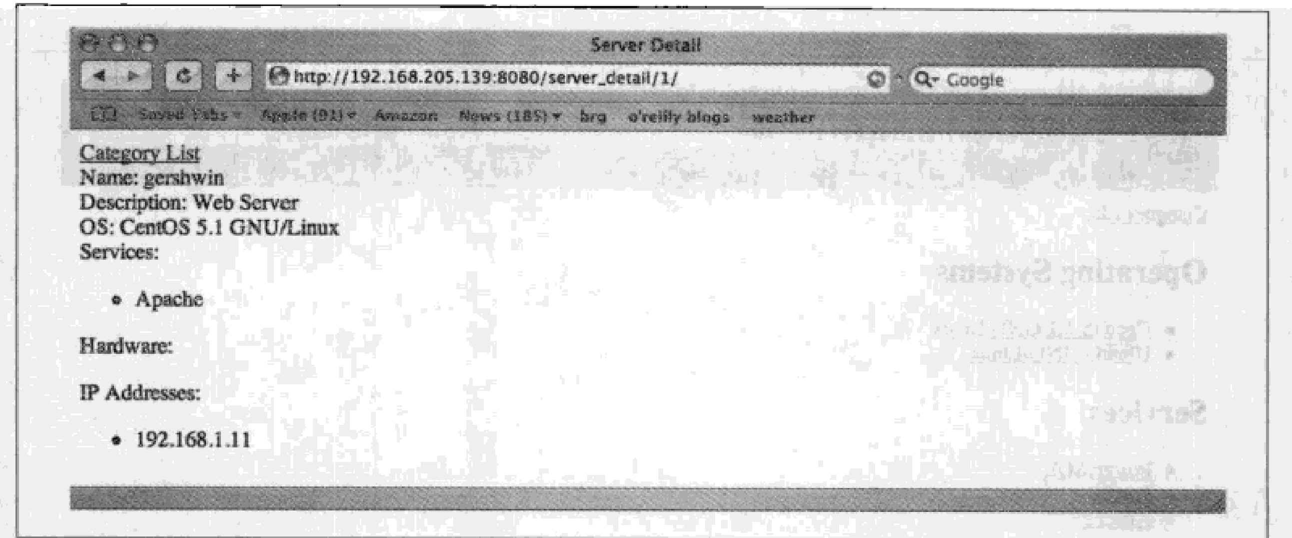
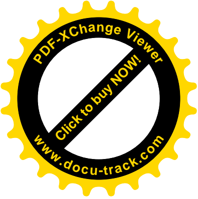
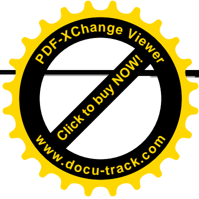


图11-12：系统管理应用的服务器细节

第12章

数据持久性

数据持久性，在一个简单通用的意义上是指为以后的用户保留数据。这表示数据被保留起来以备今后使用，即使保存它的进程终止了，数据也会幸存下来。通过转换数据到一些格式，然后写数据到磁盘，是实现这一目的典型方法。有时，格式是可读的，例如XML或是YAML。另一些时候，格式不是人们可以直接可读的，例如Berkeley DB文件（bdb）或是一个SQLite数据库。

那么，什么样的数据需要保存起来以备今后使用呢？也许你有一个脚本，保存了对上次目录中文件修改日期的记录，你偶尔需要运行它来查看自从上次你运行之后什么文件被修改了。文件中的数据是你希望保存以供今后使用的，“今后”是指你下一次运行该脚本的时候。你可以用持久数据文件的样式保存数据。在另一种情况下，你有一台主机，该主机具有潜在的网络问题，你决定每15分钟运行一个脚本来查看它ping网络上的其他一些主机时有多快。你可以保存ping的时间到一个持久数据文件中以备今后使用。这里的“今后”是指当你计划检验数据的时候，而不是搜集数据的程序需要访问它的时候。

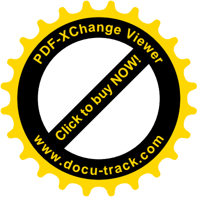
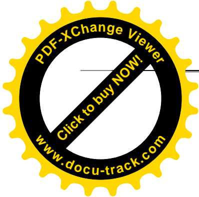
我们将这些需要序列化的问题划分为两类介绍：简单和关系。

简单序列化

有一些方法可以将数据保存到磁盘以备之后使用。我们将单纯地保存数据到磁盘，而不保存数据之间关系的过程称为“简单序列化”。我们将在关系序列化一节中介绍简单序列化与关系序列化之间的差异。

Pickle

首先，最基本的Python的“简单序列化”机制或许是标准库中的pickle模块。或许“pickle”一词会令你想到农业中或烹调中的腌渍技术，通常是为了保存食品，将食品

暂时放到一个坛子中，以备日后可以使用。这一烹调概念较好的诠释了pickle模块发生了什么。使用pickle模块，你提取一个对象，然后将对象写入磁盘，最后退出Python进程。之后的过程正相反，再一次开始Python进程，从磁盘读取对象，然后就可以与该对象进行交互了。

那么什么东西你可以"腌渍"呢？以下是来自Python标准库文档中与pickle相关的一个列表，其中列出了可以被pickle的对象类型：

- 空、真、假；
- 整数、长整数、浮点数、复合数；
- 普通和Unicode字符串；
- 元组、列表、集合以及仅包括可pickle对象的字典；
- 定义在模块顶层的函数；
- 在模块顶级定义的内建函数；
- 在模块顶级定义的类；
- 一些类的实例，要求这些类的__dict__或是__setstate__()是可pickle的。

以下是一个使用pickle模块，如何序列化你的对象到磁盘的示例：

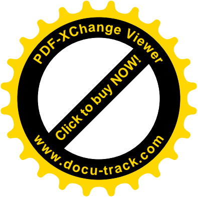
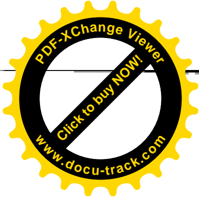
```
In [1]: import pickle

In [2]: some_dict = {'a': 1, 'b': 2}

In [3]: pickle_file = open('some_dict.pkl', 'w')

In [4]: pickle.dump(some_dict, pickle_file)

In [5]: pickle_file.close()
```

以下是pickle文件的样式：

```
jmjones@dinkgutsy:~$ ls -l some_dict.pkl
-rw-r--r-- 1 jmjones jmjones 30 2008-01-20 07:13 some_dict.pkl
jmjones@dinkgutsy:~$ cat some_dict.pkl
(dp0
S'a'
p1
I1
sS'b'
p2
I2
```

你可以学习pickle文件格式，并手动创建一个，但是我们不建议这样做。

www.TopSage.com

以下是如何unpickle一个pickle文件：

```
In [1]: import pickle

In [2]: pickle_file = open('some_dict.pkl', 'r')

In [3]: another_name_for_some_dict = pickle.load(pickle_file)

In [4]: another_name_for_some_dict
Out[4]: {'a': 1, 'b': 2}
```

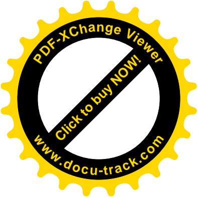
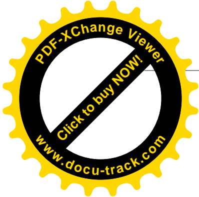
值得注意的是，在pickle之前我们命名的对象，在unpickle相同对象时没有再对对象命名。需要记住，名称仅是引用某一个对象的一种方法。注意到这一点很有意义：不需你的对象与pickle文件之间保持一对一的关系。你可以将多个对象放入到一个pickle文件中，只要有足够的硬盘空间，或是文件系统允许。以下是一个示例，将一些dictionary对象放入一个单独的pickle文件中：

```
In [1]: list_of_dicts = [{str(i): i} for i in range(5)]

In [2]: list_of_dicts
Out[2]: [{'0': 0}, {'1': 1}, {'2': 2}, {'3': 3}, {'4': 4}]

In [3]: import pickle

In [4]: pickle_file = open('list_of_dicts.pkl', 'w')

In [5]: for d in list_of_dicts:
   ...:     pickle.dump(d, pickle_file)
   ...:
   ...:

In [6]: pickle_file.close()
```

我们创建一个字典列表，创建一个可写的文件对象，迭代字典中的列表，并序列化每一项到pickle文件中。值得注意的是，这与之前的写一个对象到一个pickle文件中的示例是完全相同的方法，只是没有迭代和多重dump()调用。

以下是一个示例，演示了从一个包含了多个对象的pickle文件中unpickle对象并进行打印：

```
In [1]: import pickle

In [2]: pickle_file = open('list_of_dicts.pkl', 'r')

In [3]: while 1:
   ...:     try:
   ...:         print pickle.load(pickle_file)
   ...:     except EOFError:
   ...:         print "EOF Error"
   ...:         break
   ...:
   ...:
```

```
{'0': 0}
{'1': 1}
{'2': 2}
{'3': 3}
{'4': 4}
EOF Error
```

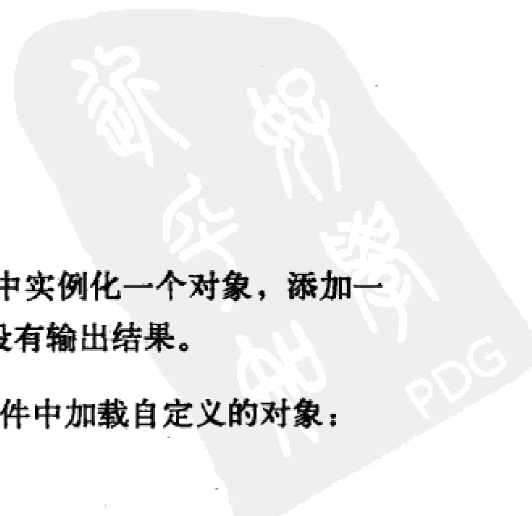
创建一个可读文件对象，指向在之前示例中创建的文件，并试图从文件中加载一个pickle对象，直到遇到一个EOFError。可以看到从pickle文件导出的字典与放入pickle文件中的字典是相同的（相同的顺序）。

不仅可以简单pickle内建对象，也可以pickle自己创建的对象类型。以下是一个模块，在接下来的两个示例中将会使用。这个模块包括一个自定义的类，我们将对其进行pickle和unpickle操作：

```
#!/usr/bin/env python

class MyClass(object):
    def __init__(self):
        self.data = []
    def __str__(self):
        return "Custom Class MyClass Data:: %s" % str(self.data)
    def add_item(self, item):
        self.data.append(item)
```

以下是一个模块，载入包含自定义类的模块，并且pickle一个自定义对象：

```
#!/usr/bin/env python

import pickle
import custom_class

my_obj = custom_class.MyClass()
my_obj.add_item(1)
my_obj.add_item(2)
my_obj.add_item(3)

pickle_file = open('custom_class.pkl', 'w')
pickle.dump(my_obj, pickle_file)
pickle_file.close()
```

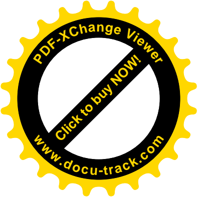
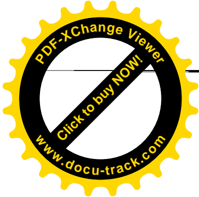


在这个示例中，我们加载带有自定义类的模块，从自定义类中实例化一个对象，添加一个元素到对象中，然后进行序列化。运行该模块，可以看到没有输出结果。

以下是一个模块，载入具有自定义类的模块，然后从pickle文件中加载自定义的对象：

```
#!/usr/bin/env python

import pickle
import custom_class
```


www.TopSage.com

```
pickle_file = open('custom_class.pkl', 'r')
my_obj = pickle.load(pickle_file)
print my_obj
pickle_file.close()
```

以下是从运行的unpickle文件得到的输出结果：

```
jmjones@dinkgutsy:~/code$ python custom_class_unpickle.py
Custom Class MyClass Data:: [1, 2, 3]
```

对于unpickle代码，无须明确地加载正在unpickle的自定义的类。但是，为了unpickle代码能够找到自定义类所在的模块，则是必须的。以下是一个模块，没有载入自定义的类模块：

```
#!/usr/bin/env python

import pickle
##import custom_class ##commented out import of custom class

pickle_file = open('custom_class.pkl', 'r')
my_obj = pickle.load(pickle_file)
print my_obj
pickle_file.close()
```

以下是来自nonimport模块的输出结果：

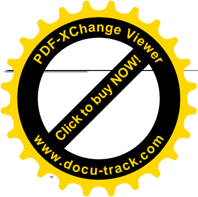
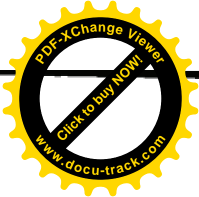
```
jmjones@dinkgutsy:~/code$ python custom_class_unpickle_noimport.py
Custom Class MyClass Data:: [1, 2, 3]
```

在复制它（以及pickle文件）到另一个目录并从这里运行之后，以下是来自相同模块的输出结果：

```
jmjones@dinkgutsy:~/code/cantfind$ python custom_class_unpickle_noimport.py
Traceback (most recent call last):
  File "custom_class_unpickle_noimport.py", line 7, in <module>
    my_obj = pickle.load(pickle_file)
  File "/usr/lib/python2.5/pickle.py", line 1370, in load
    return Unpickler(file).load()
  File "/usr/lib/python2.5/pickle.py", line 858, in load
    dispatch[key](self)
  File "/usr/lib/python2.5/pickle.py", line 1090, in load_global
    klass = self.find_class(module, name)
  File "/usr/lib/python2.5/pickle.py", line 1124, in find_class
    __import__(module)
ImportError: No module named custom_class
```

最后一行的输出表明有一个加载错误，因为pickle无法加载自定义的模块。pickle将尽力查找那些含有你自行定义的类的模块并加载，这样就可以返回一个与你初始pickle的相同类型的对象。所有之前的关于pickle的示例都运行得非常好，但是还有一个可选项

www.TopSage.com

还没有提及。当pickle一个类似的对象pickle.dump（object_to_pickle,pickle_file）时，pickle使用默认的协议。协议是对文件如何进行格式化的说明。默认的协议使用几乎所有可读的格式，这在之前已经演示过。另一个协议选择是二进制格式。如果你注意到pickle对象会花费大量的时间，或许会希望考虑使用二进制协议。以下是一个使用默认协议和二进制协议的对比：

```
In [1]: import pickle

In [2]: default_pickle_file = open('default.pkl', 'w')

In [3]: binary_pickle_file = open('binary.pkl', 'wb')

In [4]: d = {'a': 1}

In [5]: pickle.dump(d, default_pickle_file)

In [6]: pickle.dump(d, binary_pickle_file, -1)

In [7]: default_pickle_file.close()

In [8]: binary_pickle_file.close()
```

第一个创建的pickle文件（名为*default.pkl*）包括pickle数据，该pickle数据采用默认的可读格式。第二个创建的pickle文件（名为*binary.pkl*）包括二进制格式的pickle数据。需要注意的是，以正常写入模式（'w'）打开*default.pkl*，但是以二进制可写模式（'wb'）打开*binary.pkl*。在这些对象之间调用dump的仅有的差别是，调用二进制dump具有更多的参数："-1"，这表示使用最高层的协议（当前是二进制协议）。以下是一个二进制pickle文件的十六进制表示：

```
jmjones@dinkgutsy:~/code$ hexcat binary.pkl
00000000 - 80 02 7d 71 00 55 01 61   71 01 4b 01  73 2e    ..}q.U.aq.K.s.
```

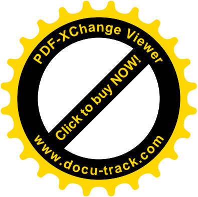
以下是默认pickle文件的十六进制表示：

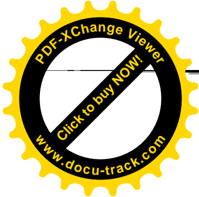
```
jmjones@dinkgutsy:~/code$ hexcat default.pkl
00000000 - 28 64 70 30 0a 53 27 61 27 0a 70 31 0a 49 31 0a (dp0.S'a'.p1.I1.
00000010 - 73 2e                                            s.
```

上述操作实际上是不必要的，因为我们可以使用cat输出并读取文件内容。以下是默认的pickle文件的无格式内容：

```
jmjones@dinkgutsy:~/code$ cat default.pkl
(dp0
S'a'
p1
I1
s.
```


www.TopSage.com





cPickle

在Python标准库中，有另一个应该考虑使用的Pickle库实现，称为cPickle。正如名字所表明的，cPickle是由C语言实现。正如我们建议使用二进制文件，如果你注意到pickle你的对象会花去一些时间，或许希望考虑尝试cPickle模块。对于"普通的"pickle，cPickle语法与Pickle等同。

shelve

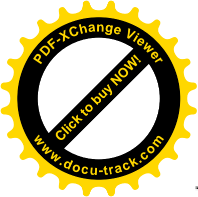
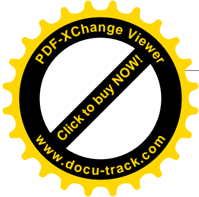
另一个持久化的选择是使用shelve模块。shelve提供一个对象持久化的简单且实用的接口，可以简化对多个对象的持久化。通过它，我们保存多个对象在相同的持久对象存储，并且很容易地将它们取回来。在shelve持久数据存储中保存对象与简单地使用Python字典相似。以下是一个示例，演示了打开一个shelve文件，将数据序列化到其中，然后再次打开它，访问其中的内容：

```
In [1]: import shelve

In [2]: d = shelve.open('example.s')

In [3]: d
Out[3]: {}
In [4]: d['key'] = 'some value'

In [5]: d.close()

In [6]: d2 = shelve.open('example.s')

In [7]: d2
Out[7]: {'key': 'some value'}
```

在使用shelve和使用无格式字典之间的差异在于你可以通过使用shelve.open()而不是实例化dict类或是使用大括号（{}）来创建一个shelve对象。另一个差异是，当你使用shelve处理数据时，你需要在shelve对象上调用close()。

shelve有一些技巧。我们已经在前面介绍过：当你执行操作时，必须调用close()。如果不使用close()关闭shelve对象，对其进行的任何修改不具有持久性。以下是一个示例，演示了由于没有关闭shelve对象而丢失了修改。首先，我们创建并持久化我们的shelve对象，然后退出IPython：

```
In [1]: import shelve

In [2]: d = shelve.open('lossy.s')

In [3]: d['key'] = 'this is a key that will persist'

In [4]: d
Out[4]: {'key': 'this is a key that will persist'}
```

```
In [5]: d.close()

In [6]:
Do you really want to exit ([y]/n)?
```

接下来，再次启动IPython，打开相同的shelve文件，创建另一个元素，然后在没有明确关闭shelve对象的情况下退出：

```
In [1]: import shelve

In [2]: d = shelve.open('lossy.s')

In [3]: d
Out[3]: {'key': 'this is a key that will persist'}

In [4]: d['another_key'] = 'this is an entry that will not persist'

In [5]:
Do you really want to exit ([y]/n)?
```

现在，再次启动IPython，再次打开相同的shelve文件，然后查看我们有些什么：

```
In [1]: import shelve

In [2]: d = shelve.open('lossy.s')

In [3]: d
Out[3]: {'key': 'this is a key that will persist'}
```

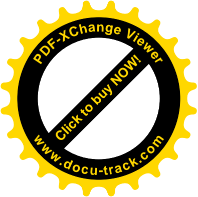
因此，确保关闭了所有shelve对象，这些对象是已经修改过的并且其数据是希望保存的。

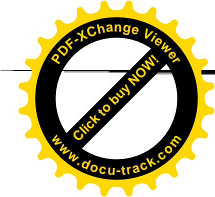
另一个技巧在于修改可变对象（mutable object）。记住，可变对象的值可以在不重新给变量赋值的情况下修改。这里创建了一个shelve对象，创建一个包含可变对象的密钥（在这个示例中是一个列表），修改可变对象，然后关闭shelve对象：

```
In [1]: import shelve

In [2]: d = shelve.open('mutable_lossy.s')

In [3]: d['key'] = []

In [4]: d['key'].append(1)

In [5]: d.close()

In [6]:
Do you really want to exit ([y]/n)?
```

由于你在shelve对象上调用了close()，我们或许期望“密钥”的值是list[1]的值。但是我们搞错了。以下是打开前一示例中的shelve文件并反序列化的结果：

www.TopSage.com





```
In [1]: import shelve

In [2]: d = shelve.open('mutable_lossy.s')

In [3]: d
Out[3]: {'key': []}
```

这没有什么奇怪或是出人意料的。事实上，它来自shelve文档。问题是修改持久性对象默认不会被pickle。但是有一些方法可以来解决这一问题。一种是专门、目标性的方法，另一种是广义、全包含的方法。首先，在专门或面向目标的方法中，你可以重新赋值shelve对象，就像这样：

```
In [1]: import shelve

In [2]: d = shelve.open('mutable_nonlossy.s')

In [3]: d['key'] = []

In [4]: temp_list = d['key']

In [5]: temp_list.append(1)

In [6]: d['key'] = temp_list

In [7]: d.close()

In [8]:
Do you really want to exit ([y]/n)?
```

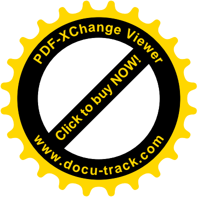
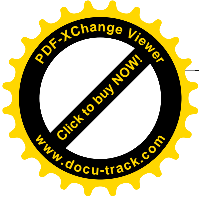
当我们反序列化shelve对象时，以下是得到的结果：

```
In [1]: import shelve

In [2]: d = shelve.open('mutable_nonlossy.s')

In [3]: d
Out[3]: {'key': [1]}
```

我们创建和附加的列表已经被保留了。

接下来，是广泛和全包含的方法：修改shelve对象的writeback标志。我们已经介绍的传递到shelve.open()的参数是shelve文件的文件名。其实还有一些其他的选项，其中之一是writeback 标志。如果writeback标示被设置为真，被访问的shelve对象的任何元素将被缓存在内存中，并且当close()在shelve对象中被调用时，实现持久化。这对于处理可变对象的情况，是非常有用的。但是它确实是双赢的。由于被访问的对象会被缓存并且被持久化（无论修改或没有修改），内存的使用和文件同步时间将按对象（这些对象是你在shelve对象上正在访问的对象）数量的比例增长。因此如果在你正在访问的shelve对象上有大量对象，你或许会考虑不会将writeback标志设置为True。

www.TopSage.com

在接下来的示例中，我们将设置writeback标志为True，并且操控一个列表，而不重新赋值它到shelve对象：

```
In [1]: import shelve

In [2]: d = shelve.open('mutable_nonlossy.s', writeback=True)

In [3]: d['key'] = []

In [4]: d['key'].append(1)

In [5]: d.close()

In [6]:
Do you really want to exit ([y]/n)?
```

现在，看一下我们的修改是否被持久化了。

```
In [1]: import shelve

In [2]: d = shelve.open('mutable_nonlossy.s')

In [3]: d
Out[3]: {'key': [1]}
```

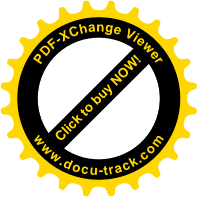
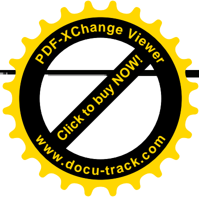
正如我们所希望的，它是持久的。

shelve提供一个简便的方法处理数据持久性。会有一些不足，但是总体来说还是非常有用的模块。

YAML

有人说YAML代表“YAML ain't markup language”（YAML不是标识语言），还有人说YAML代表“yet another markup language”（已经是另一种标识语言），究竟哪一个正确，似乎取决于你问的人是谁。无论哪一个答案，它是一种数据格式，经常以纯文本方式被用来保存、获得和更新数据。数据通常是分层的。或许在Python中开始使用YAML最简单的方法是使用“easy_install PyYAML”。但是当pickle是内建的，又为什么必须安装并使用YAML？有两个选择YAML而不选择pickle的吸引人的原因。虽然这两个原因不能保证YAML在所有的情况下都是正确选择，但是在某些情况下，选择使用YAML是非常有好处的。首先，YAML是可读的。语法看起来类似于配置文件。如果你遇到编辑配置文件是一个好的选择的情况下，YAML或许是一个好选择。第二，YAML解析器已经在许多其他语言中实现。如果你需要在Python应用与以另一种语言编写的应用之间获得数据，YAML是一个好的折中方案。

一旦使用了“easy_install PyYAML”，可以序列化和反序列化YAML数据。以下是一个示例，演示了序列化一个简单的字典：

```
In [1]: import yaml

In [2]: yaml_file = open('test.yaml', 'w')

In [3]: d = {'foo': 'a', 'bar': 'b', 'bam': [1, 2,3]}

In [4]: yaml.dump(d, yaml_file, default_flow_style=False)

In [5]: yaml_file.close()
```

这个示例非常简单，但是还是让我们介绍一下。首先要做的是载入YAML模块（名为yaml）。接下来，我们创建一个可写入的文件，该文件以后会用来保存YAML。之后，创建一个字典（名为d），该字典中包括了我们希望进行序列化的数据。最后，我们使用yaml模块中的dump()函数序列化字典（名为d）。传递给dump()的参数包括正在序列化的字典、YAML输出文件，以及一个参数，该参数告诉YAML库以块方式进行写输出而不是默认方式。这看起来有点像数据对象的字符串转换，而其中的数据对象正是我们正在序列化的对象。

以下是YAML文件的示例：

```
jmjones@dinkgutsy:~/code$ cat test.yaml
bam:
- 1
- 2
- 3
bar: b
foo: a
```

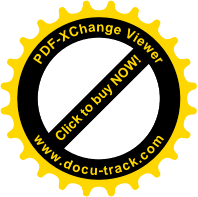
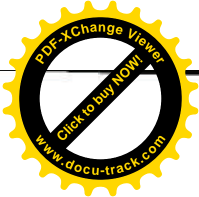
如果希望对文件进行反序列化，执行与dump()示例相反的操作。以下是如何将数据从YAML文件中导出的示例：

```
In [1]: import yaml

In [2]: yaml_file = open('test.yaml', 'r')

In [3]: yaml.load(yaml_file)
Out[3]: {'bam': [1, 2, 3], 'bar': 'b', 'foo': 'a'}
```

如dump()示例中所示，首先必须加载YAML模块（yaml）。接下来，创建一个YAML文件。这次我们从磁盘中的YAML文件创建一个可读的文件对象。最后，从yaml模块调用load()函数。load()返回一个字典，该字典等同于输出字典。

当使用yaml模块时，你或许会找到自己循环创建的数据，dump这些数据到磁盘，然后加载它。

你或许不需要dump你的YAML数据为可读模式。让我们介绍如何序列化字典，该字典来自之前的示例。以下示例演示了如何dump相同的字典：

```
In [1]: import yaml

In [2]: yaml_file = open('nonblock.yaml', 'w')

In [3]: d = {'foo': 'a', 'bar': 'b', 'bam': [1, 2,3]}

In [4]: yaml.dump(d, yaml_file)

In [5]: yaml_file.close()
```

这是YAML文件看起来的样子：

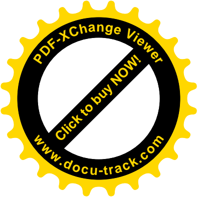
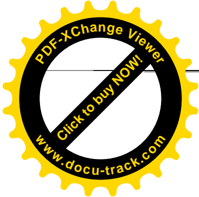
```
jmjones@dinkgutsy:~/code
$ cat nonblock.yaml
bam: [1, 2, 3] bar: b foo: a
```

这看起来与块模式格式非常相似，除了bam列表值之外。当存在一些层次的递归或一些数组式的数据结构（如列表或是字典）的时候，差别就会出现。让我们对比一些示例来展示这些差别。但是在我们这样做之前，如果我们没有保留使用cat显示的YAML文件的内容，就很容易错过这些示例。yaml模块中的dump()函数的文件参数是可选的。（PyYAML文档在涉及“文件”对象时，将文件对象作为“流”对象来引用，但是实际上不是这样）。如果放弃了“文件”（或是“流”）这个参数，dump()将写序列化对象到标准输出。因此，在以下的示例中，我们放弃了文件对象，直接输出YAML的结果。

以下是对一些数据结构的对比，使用了块风格序列化和非块风格序列化。以下是具有default_flow_style风格并使用块格式的示例，以及不具有default_flow_style风格，不使用块格式的示例：

```
In [1]: import yaml

In [2]: d = {'first': {'second': {'third': {'fourth': 'a'}}}}

In [3]: print yaml.dump(d, default_flow_style=False)
first:
  second:
    third:
      fourth: a


In [4]: print yaml.dump(d)
first:
  second:
    third: {fourth: a}


In [5]: d2 = [{'a': 'a'}, {'b': 'b'}, {'c': 'c'}]

In [6]: print yaml.dump(d2, default_flow_style=False)
- a: a
- b: b
- c: c
```

www.TopSage.com

```
In [7]: print yaml.dump(d2)
- {a: a}
- {b: b}
- {c: c}


In [8]: d3 = [{'a': 'a'}, {'b': 'b'}, {'c': [1, 2, 3, 4, 5]}]

In [9]: print yaml.dump(d3, default_flow_style=False)
- a: a
- b: b
- c:
  - 1
  - 2
  - 3
  - 4
  - 5


In [10]: print yaml.dump(d3)
- {a: a}
- {b: b}
- c: [1, 2, 3, 4, 5]
```

如果你希望序列化一个自定义类，那会怎么样呢？yaml模块的行为与pickle自定义类十分相似。以下示例将进一步使用相同的custom_class模块，这是我们在“pickle custom_class”示例中使用的模块。从MyClass创建一个对象，向对象中添加一些元素，然后进行序列化：

```
#!/usr/bin/env python

import yaml
import custom_class

my_obj = custom_class.MyClass()
my_obj.add_item(1)
my_obj.add_item(2)
my_obj.add_item(3)

yaml_file = open('custom_class.yaml', 'w')
yaml.dump(my_obj, yaml_file)
yaml_file.close()
```

当我们运行之前的模块时，以下是我们看到的输出：

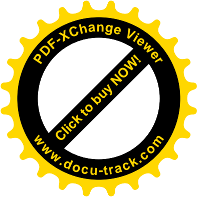
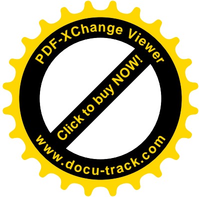
```
jmjones@dinkgutsy:~/code$ python custom_class_yaml.py
jmjones@dinkgutsy:~/code$
```

没有任何输出。这表明所有事情运行正常。

以下是与之前模块相反的示例：

```
#!/usr/bin/env python
```

www.TopSage.com

```
import yaml
import custom_class

yaml_file = open('custom_class.yaml', 'r')
my_obj = yaml.load(yaml_file)
print my_obj
yaml_file.close()
```

这个脚本加载yaml和custom_class模块。从之前创建的YAML创建一个可读文件对象，加载YAML文件到对象中并输出对象。

当我们运行它时，可以看到以下结果：

```
jmjones@dinkgutsy:~/code$ python custom_class_unyaml.py
Custom Class MyClass Data:: [1, 2, 3]
```

这与本章之前运行的unpickle示例的输出结果相同，这是我们希望看到的结果。

ZODB

序列化数据的另一个选择是Zope的ZODB模块。ZODB表示“ZopeObject Database”。ZODB简单灵活的用法与序列化到pickle或是YAML十分相似，但是ZODB具有按需定制的功能。例如，如果在你的操作中需要原子性，ZODB提供了事务处理。如果需要一个更加可扩展的持久存储，可以使用ZEO，这是Zope的发布对象存储。

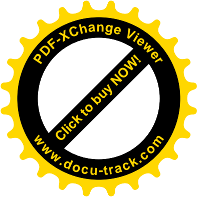
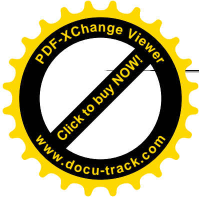
ZODB或许应该编入“关系持久”部分而不是“简单持久”部分。但是，这个对象数据库不能准确地适应我们多年来已经熟悉的关系数据库模式，尽管你可以简单地建立对象之间的关系。我们也展示了一些更基本的ZODB的特征。在示例中，它看起来更像是shelve而不是一个关系数据库。因此，我们决定保留ZODB在“简单持久”部分中。

至于安装ZODB，只需要简单地执行“easy_install ZODB3”即可。ZODB模块有一些依赖，但是easy_install解决了该问题，它可以下载并安装ZODB模块需要的任何依赖。

作为使用ZODB的一个简单的例子，我们创建了一个ZODB存储对象，并添加一个字典和一个列表到其中。以下是实现序列化字典和列表的代码：

```
#!/usr/bin/env python

import ZODB
import ZODB.FileStorage
import transaction

filestorage = ZODB.FileStorage.FileStorage('zodb_filestorage.db')
db = ZODB.DB(filestorage)
conn = db.open()
```


www.TopSage.com

```
root = conn.root()
root['list'] = ['this', 'is', 'a', 'list']
root['dict'] = {'this': 'is', 'a': 'dictionary'}

transaction.commit()
conn.close()
```

与看到的使用pickle或YAML完成的工作相比，ZODB需要更多的代码，但是一旦你有一个创建并初始化的持久存储，用法与pickle或YAML就非常相似了。这个示例具有很好的自解释性，尤其给出了数据持久性的其他一些示例，我们将快速地进行介绍。

首先，加载两个名为ZODB.FileStorage以及transaction的ZODB模块（我们将在这里介绍一点别的内容。假设提供加载的模块不包括一个可识别前缀，那么看起来会有些别扭。对于我们，之前加载的transaction模块应该被赋予ZODB前缀。无论如何你需要注意这一点。现在我们将继续）。接下来，通过指定使用的数据库文件来创建一个FileStorage对象。然后创建一个DB对象并连接它到FileStorage对象。然后，我们使用open()打开数据库对象，获得它的根节点。这时可以更新数据结构（我们使用的是临时的列表和字典）的根对象。之后，使用transaction.commit()提交所做的修改，最后，使用conn.close()关闭数据库连接。

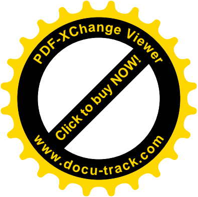
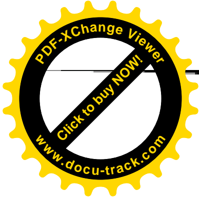
你已经创建了一个ZODB数据存储容器（例如在这个示例中的文件存储对象），并且已经提交了一些数据，但是你可能希望再次取回数据。以下是一个示例，演示了打开相同的数据库，但是这次是读取数据而不是写入数据：

```
#!/usr/bin/env python

import ZODB
import ZODB.FileStorage

filestorage = ZODB.FileStorage.FileStorage('zodb_filestorage.db')
db = ZODB.DB(filestorage)
conn = db.open()

root = conn.root()
print root.items()

conn.close()
```

在运行对数据库进行操作的代码之后，如果你运行这个代码，可以看到如下的输出结果：

```
jmjones@dinkgutsy:~/code$ python zodb_read.py
No handlers could be found for logger "ZODB.FileStorage"
[('list', ['this', 'is', 'a', 'list']), ('dict', {'this': 'is', 'a': 'dictionary'})]
```

我们已经演示了如何使用其他数据持久框架序列化自定义类，接下来将演示如何使用ZODB进行同样的处理。但是这里不使用相同的MyClass示例（我们将在以后进行解

释）。正如其他框架一样，定义一个类，然后从其中创建一个对象，再告诉序列化引擎保存它到磁盘。以下是这次使用的自定义类：

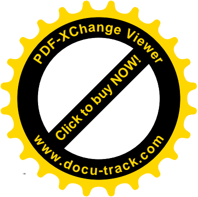
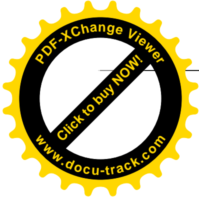
```
#!/usr/bin/env python

import persistent

class OutOfFunds(Exception):
    pass

class Account(persistent.Persistent):
    def__init__(self, name, starting_balance=0):
        self.name = name
        self.balance = starting_balance
    def __str__(self):
        return "Account %s, balance %s" % (self.name, self.balance)
    def __repr__(self):
        return "Account %s, balance %s" % (self.name, self.balance)
    def deposit(self, amount):
        self.balance += amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            raise OutOfFunds
        self.balance -= amount
        return self.balance
```

这是一个非常简单的用于管理资金的account类。我们定义一个OutOfFunds异常，这会在以后进行解析。account类是persistent.Persistent的子类（对于persistent，我们有相同的观点，即属性应该提供一个有意义的前缀，该前缀应该是人们将会使用的模块。如何瞄一眼代码就可以告诉读者ZODB代码在这里被使用了？不能）。从persistent.Persistent而来的子类执行同样的后台操作，使得ZODB很容易被序列化。在类的定义中，我们创建自定义的__str__和__repr__string转换。你将在之后看到这些。我们也创建deposit()和withdraw()方法。这两种方法正向或反向更新对象的balance属性，这取决于被调用的方法。withdraw()方法在减除资金的同时，检查是否有足够的资金在balance属性中。如果没有足够的资金在balance中，withdraw()方法将抛出OutOfFunds异常，该异常是之前已经介绍过的。deposit()和withdraw在对资金或加或减之后都返回收支结余。

以下是一些代码，用于序列化我们介绍的自定义类：

```
#!/usr/bin/env python

import ZODB
import ZODB.FileStorage
import transaction
import custom_class_zodb
```


www.TopSage.com

```
filestorage = ZODB.FileStorage.FileStorage('zodb_filestorage.db')
db = ZODB.DB(filestorage)
conn = db.open()

root = conn.root()
noah = custom_class_zodb.Account('noah', 1000)
print noah
root['noah'] = noah
jeremy = custom_class_zodb.Account('jeremy', 1000)
print jeremy
root['jeremy'] = jeremy

transaction.commit()
conn.close()
```

这个示例与之前的ZODB示例几乎相同。在ZODB示例中我们序列化了一个字典和一个列表。我们加载自己的模块，从一个自定义类中创建两个对象，并且序列化这两个对象为ZODB数据库。这两个对象是noah账户和jeremy账户，这两个创建的账户都有结余1000（假设是$1,000.00 USD，我们没有对货币的单位进行识别）。

以下是这个示例的输出：

```
jmjones@dinkgutsy:~/code$ python zodb_custom_class.py
Account noah, balance 1000
Account jeremy, balance 1000
```

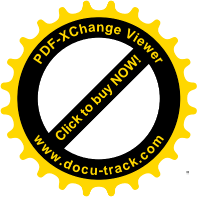
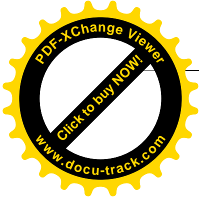
我们运行显示ZODB数据库内容的模块，以下是我们看到的内容：

```
jmjones@dinkgutsy:~/code$ python zodb_read.py
No handlers could be found for logger "ZODB.FileStorage"
[('jeremy', Account jeremy, balance 1000), ('noah', Account noah, balance 1000)]
```

代码不仅创建了我们需要的对象，而且也将它们保存到了磁盘以备将来使用。

我们如何打开数据库并为不同账户修改数据？如果它不允许我们这样做，这个代码用处不大。以下是一个代码段，将打开之前创建的数据库并从noah账户转移300（假设是美元）到jeremy账号：

```
#!/usr/bin/env python

import ZODB
import ZODB.FileStorage
import transaction
import custom_class_zodb

filestorage = ZODB.FileStorage.FileStorage('zodb_filestorage.db')
db = ZODB.DB(filestorage)
conn = db.open()

root = conn.root()
noah = root['noah']
```

```
print "BEFORE WITHDRAWAL"
print "================="
print noah

jeremy = root['jeremy']
print jeremy
print "-----------------"

transaction.begin()
noah.withdraw(300)
jeremy.deposit(300)
transaction.commit()

print "AFTER WITHDRAWAL"
print "================"
print noah
print jeremy
print "----------------"

conn.close()
```

以下是脚本运行后的输出：

```
jmjones@dinkgutsy:~/code$ python zodb_withdraw_1.py
BEFORE WITHDRAWAL
=================

Account noah, balance 1000
Account jeremy, balance 1000
-----------------
AFTER WITHDRAWAL
================

Account noah, balance 700
Account jeremy, balance 1300
----------------
```

之后我们运行ZODB数据库打印脚本，查看数据是否被保留：

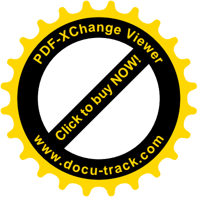
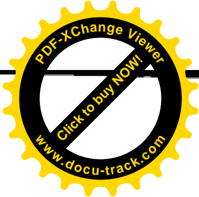
```
jmjones@dinkgutsy:~/code$ python zodb_read.py
[('jeremy', Account jeremy, balance 1300), ('noah', Account noah, balance 700)]
```

可以看到，noah账户由1000变为了700，而jeremy账户由1000变为了1300。

我们偏离MyClass自定义类示例的原因是因为要对事务做些介绍。其中一个典型的例子是银行账户。如果需要确保资金能够成功的从一个账户转到另一个账户而不会出现资金丢失或赠加的情况，采用事务进行处理是一种可行的方式。为了确保资金不会丢失，以下例子在循环中使用了事务处理：

```
#!/usr/bin/env python

import ZODB
import ZODB.FileStorage
```


www.TopSage.com

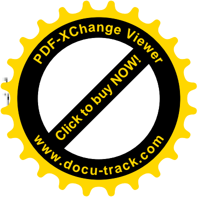



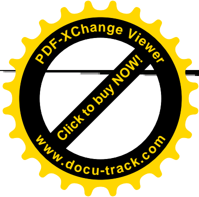
```
import transaction
import custom_class_zodb

filestorage = ZODB.FileStorage.FileStorage('zodb_filestorage.db')
db = ZODB.DB(filestorage)

conn = db.open()

root = conn.root()
noah = root['noah']
print "BEFORE TRANSFER"
print "==============="
print noah
jeremy = root['jeremy']
print jeremy
print "-----------------"

while True:
    try:
        transaction.begin()
        jeremy.deposit(300)
        noah.withdraw(300)
        transaction.commit()
    except custom_class_zodb.OutOfFunds:
        print "OutOfFunds Error"
        print "Current account information:"
        print noah
        print jeremy
        transaction.abort()
        break

print "AFTER TRANSFER"
print "=============="
print noah
print jeremy
print "----------------"

conn.close()
```

这是对之前的转账脚本略微修改之后的版本。与之前仅进行的转账不同，它不断地从noah账户转移300到jeremy账户，直到不存在足够的资金进行转账为止。在不存在足够的资金进行转账的情况下，它将打印一个通知，告诉有异常发生以及当前的账户信息。然后使用about()放弃事务，从循环中退出。该脚本在交易循环之前和之后也打印账户信息。如果交易成功，之前和之后的账户详细信息应该总计2000，因为这两个账户都是从1000开始的。

以下是运行该脚本的结果：

```
jmjones@dinkgutsy:~/code$ python zodb_withdraw_2.py
BEFORE TRANSFER
===============
Account noah, balance 700
Account jeremy, balance 1300
```






```
-----------------
OutOfFunds Error
Current account information:
Account noah, balance 100
Account jeremy, balance 2200
AFTER TRANSFER
==============
Account noah, balance 100
Account jeremy, balance 1900
----------------
```

之前，noah有700，jeremy有1300，总共2000。当OutOfFunds异常发生时，noah有100，jeremy有2200，总共有2300。在程序执行完毕之后，noash有100，有1900，总共有2000。那么在异常过程中，在transaction.abort()之前，有额外300是无法解释的。但是放弃交易修复了这个问题。

ZODB的处理方法很直接，看起来像一个在简化与关系序列化之间的解决方案。序列化到磁盘的对象对应于内存中的一个对象，不管是在序列化之前或是之后。ZODB还有一些类似事务处理这样的高级特征。如果你希望简单对象的映射更为直接，ZODB是一个值得考虑的选择，但是你或许需要进一步深入了解其高级特征。

总之，简单地保存Python对象以备将来使用，这就是简单持久。我们给出的所有选择方案都是非常不错的。每一个都有它的长处和弱点。如果你需要某一功能，必须调查哪一个最适合你和你的项目。

关系序列化

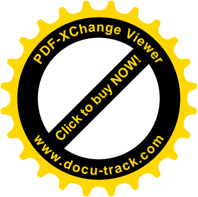
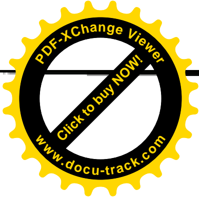
有时简单序列化是不够的，需要具有关系分析的能力。关系序列化包括：序列化Python对象、与其他Python对象的相对连接，保存关系数据（例如，一个关系型数据库），以及提供一个Python对象类的数据接口。

SQLite

以一种更结构化和关系化的方法来保存和处理数据是非常有用的。这里我们将要讨论的就是信息存储之类的问题，涉及关系数据库、RDBMS等。我们假设你曾经使用过关系数据库，例如MySQL、PostgreSQL或是以前的Oracle。如果是这样，你应该对本节没有什么问题。

根据SQLite网站的描述，SQLite是“一个软件库，可以实现自包含、零服务器、零配置、事务型SQL数据库引擎”。那么所有这些描述又意味着什么？不同于你代码中运行在独立服务器进程上的数据库，数据库引擎运行在与你的代码和作为库进行访问的相同的进程上。数据保存在一个文件中而不是分散到多个文件系统的多个目录中。除了必须

www.TopSage.com

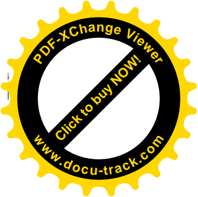



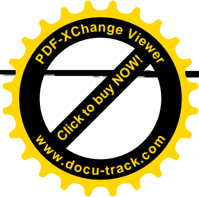
配置主机名、端口号、用户名、密码等来进行连接外，你需要在数据库文件中指定代码，该数据库文件由SQLite库创建。这也意味着，SQLite是一个非常有特点的数据库。简言之，使用SQLite主要有两个好处：它很容易使用并且能够完成大量相同的工作，这是一个“真正”的数据库将必须具备的；另一个好处是它是无处不在的。绝大多数主流的操作系统和程序语言都提供对SQLite的支持。

现在，你知道了为什么应该考虑使用它，接下来让我们看一下如何使用。我们从第11章的Django示例中提取以下的表定义。假设我们有一个名为*inventory.sql*的文件，该文件包含以下的数据：

```
BEGIN;
CREATE TABLE "inventory_ipaddress" (
    "id" integer NOT NULL PRIMARY KEY,
    "address" text NULL,
    "server_id" integer NOT NULL
)
;
CREATE TABLE "inventory_hardwarecomponent" (
    "id" integer NOT NULL PRIMARY KEY,
    "manufacturer" varchar(50) NOT NULL,
    "type" varchar(50) NOT NULL,
    "model" varchar(50) NULL,
    "vendor_part_number" varchar(50) NULL,
    "description" text NULL
)
;
CREATE TABLE "inventory_operatingsystem" (
    "id" integer NOT NULL PRIMARY KEY,
    "name" varchar(50) NOT NULL,
    "description" text NULL
)
;
CREATE TABLE "inventory_service" (
    "id" integer NOT NULL PRIMARY KEY,
    "name" varchar(50) NOT NULL,
    "description" text NULL
)
;
CREATE TABLE "inventory_server" (
    "id" integer NOT NULL PRIMARY KEY,
    "name" varchar(50) NOT NULL,
    "description" text NULL,
    "os_id" integer NOT NULL REFERENCES "inventory_operatingsystem" ("id")
)
;
CREATE TABLE "inventory_server_services" (
    "id" integer NOT NULL PRIMARY KEY,
    "server_id" integer NOT NULL REFERENCES "inventory_server" ("id"),
    "service_id" integer NOT NULL REFERENCES "inventory_service" ("id"),
    UNIQUE ("server_id", "service_id")
```

www.TopSage.com


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com


PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

```
)
;
CREATE TABLE "inventory_server_hardware_component" (
    "id" integer NOT NULL PRIMARY KEY,
    "server_id" integer NOT NULL REFERENCES "inventory_server" ("id"),
    "hardwarecomponent_id" integer
    NOT NULL REFERENCES "inventory_hardwarecomponent" ("id"),
    UNIQUE ("server_id", "hardwarecomponent_id")
)
;
COMMIT;
```

我们可以使用以下的命令行参数创建一个SQLite数据库：

```
jmjones@dinkgutsy:~/code$ sqlite3 inventory.db < inventory.sql
```

在Ubuntu和Debian系统中，SQLite的安装非常简单，类似“apt-get install sqlite3”。在Red Hat系统中，需要你去做的仅是“yum install sqlite”。对于其他可能没有安装SQLite的Linux版本，UNIX或是Windows，你可以下载源码或预编译成二进制编码（*http://www.sqlite.org/download.html*）。

假设你已经安装了SQLite并且创建了一个数据库，继续连接到这个数据库，并处理一些数据。以下是连接到一个SQLite数据库的示例：

```
In [1]: import sqlite3

In [2]: conn = sqlite3.connect('inventory.db')
```

我们必须做的是加载SQLite库，然后在sqlite3模块上调用connect()。connect()返回一个连接对象，该对象称为conn，这也是之后在示例中将要使用的。接下来，在连接对象上执行一个查询操作，插入数据到数据库中：

```
In [3]: cursor = conn.execute("insert into inventory_operatingsystem (name,
description) values ('Linux', '2.0.34 kernel');")
```

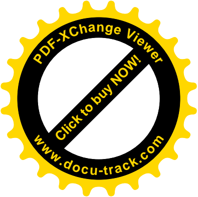
execute()方法返回一个数据库指针对象，我们决定使用cursor对其进行引用。值得注意的是，我们仅提供了name和description字段的值，留下一个字段作为id值，该值是主关键字。由于这是一个插入而不是选择，我们不希望从查询中获得结果集，因此我们仅查看指针并取回处理的任何结果：

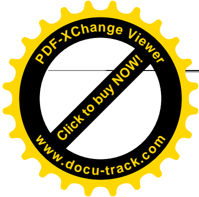
```
In [4]: cursor.fetchall()
Out[4]: []
```

正如我们所期望的，什么也没有输出。现在我们执行提交并继续：

```
In [5]: conn.commit()

In [6]:
```


www.TopSage.com





实际上不必提交这个插入操作。当关闭数据库连接时，我们希望这一修改被清除。

接下来创建数据库并使用SQLite对数据库进行操作。让我们取回数据。首先，打开一个IPyton提示，加载sqlite3并创建一个连接到数据库文件：

```
In [1]: import sqlite3

In [2]: conn = sqlite3.connect('inventory.db')
```

现在，我们执行一个select查询，获得返回结果的指针：

```
In [3]: cursor = conn.execute('select * from inventory_operatingsystem;')
```

最后，我们通过指针取回结果：

```
In [4]: cursor.fetchall()
Out[4]: [(1, u'Linux', u'2.0.34 kernel')]
```

这是在上面插入的数据。name和description字段都是unicode。id字段为整型数据。典型的，当你插入数据到数据库中的时候，不要为主键定义一个值，数据库将帮你将它设置为自动增加，会为这个字段使用独一无二的值。

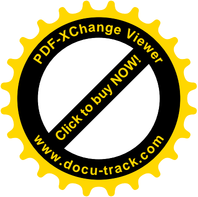
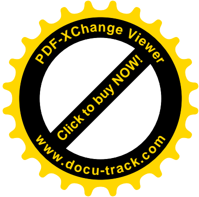
现在我们熟悉了处理SQLite数据库的基本方法。执行插入数据、升级数据，以及更复杂的事情几乎成为一种练习。SQLite在数据存储方面非常出色，其格式尤其适于数据仅能一次被一个脚本访问，或是一次仅有几个用户可以访问的情况。换句话说，该格式对于少量用户是非常友好的。

Storm ORM

为了获取数据，更新，插入及删除数据库中的数据，一个简单的SQL数据库接口是你真正需要的，通常这可以方便地访问数据，不需要背离Python的简单原则。这些年来，数据库访问方面的一个趋势是创建一个面向对象的数据表示，且保存在数据库中。这一趋势被称为Object-Relational Mapping（或ORM）。一个ORM不同于简单地提供一个面向对象的数据库接口。在ORM中，程序语言中的对象对于一个简单的数据库表，可以对应到一单行。具有外键关系的被连接的表，可以作为一个对象属性访问。

Storm是一个ORM，它最近被作为开放源码由Canonical发布，该公司负责创建Linux的Ubuntu发布版。对于Python数据库，Storm相对较新，但是它已经有了一个引起关注的版本，我们希望它成为一流的Python ORM之一。

我们现在使用Storm来访问数据库中的数据，该数据库在SQLite一节中被首先定义。第一件我们必须去做的事情是创建一个表的映射。由于已经访问了inventory_

operatingsystem表，并添加一个记录，我们将继续访问这个表。以下是Storm中映射的示例：

```
import storm.locals

class OperatingSystem(object):
    __storm_table__ = 'inventory_operatingsystem'
    id = storm.locals.Int(primary=True)
    name = storm.locals.Unicode()
    description = storm.locals.Unicode()
```

这是一个非常普通的类定义。没有什么神奇、怪异的事情发生。除了内建的object类型，没有子类。有一些类属性被定义。一个略显陌生的是类的__storm_table__属性。该属性让Storm知道哪一个表是该类应该访问的。这看起来非常简单，直接，没什么神秘的，但混合起来有一点神奇之处。例如，name属性被映射到inventory_operationgsytem表的name列，description属性被映射到inventory_operatingsystem表的description列。这是怎么回事呢？奇怪。你指定到Storm映射类的任何属性都会自动映射到列，该列共享它的名称，且与__storm_table__属性指定的表共享。

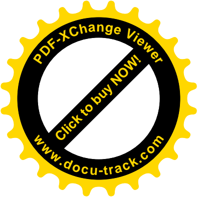
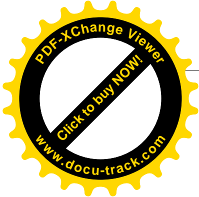
如果你不希望对象的description属性映射到description列？很简单。传递name关键字参数给你正在使用的storm.locals.Type。例如，修改description属性为“this: dsc= storm.locals.Unicode(name='description') ”，连接OperationSystem对象到相同的列（即name和description）。但是，不同于使用mapped_object.description对description进行引用，你将使用mapped_object.dsc对其进行引用。

现在，有一个Python类到数据库表的映射，让我们添加另一行到数据库中。为了继续使用已经的具有2.0.34内核的Linux发布版本，我们添加Windows 3.1.1到操作系统表中：

```
import storm.locals
import storm_model
import os

operating_system = storm_model.OperatingSystem()
operating_system.name = u'Windows'
operating_system.description = u'3.1.1'

db = storm.locals.create_database('sqlite:///%s' % os.path.join(os.getcwd(),
  'inventory.db'))

store = storm.locals.Store(db)
store.add(operating_system)
store.commit()
```

在这个示例中，加载storm.locals，storm_model和os模块。然后，实例化一个OperationSystem对象并对name和description属性进行赋值（需要注意的是，为这些

www.TopSage.com

属性使用unicode值）。然后通过调用create_database()函数创建一个数据库对象，传递它的路径到SQLite数据库文件*inventory.db*中。你或许认为database对象是我们将要使用的添加数据到数据库中的对象。事实上它不是的，至少不直接是。首先必须通过传递数据库到它的构造器来创建一个store对象。在这之后，添加operation_system对象到store对象中。最后，在store上调用commit()将operation_system添加到数据库。

我们也希望看到插入的数据实际上是如何放入数据库的。由于这是一个SQLite数据库，可以使用sqlite3命令行工具。如果这样做，那么使用Storm从数据库中获得数据将无须写代码。以下是一个简单的工具，从inventory_operationsystem表中来获得所有的记录并打印输出（尽管方式非常笨拙的）：

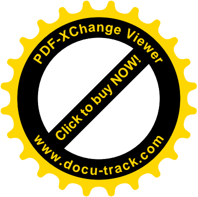
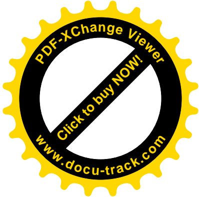
```
import storm.locals
import storm_model
import os

db = storm.locals.create_database('sqlite:///%s' % os.path.join(os.getcwd(),
  'inventory.db'))

store = storm.locals.Store(db)

for o in store.find(storm_model.OperatingSystem):
    print o.id, o.name, o.description
```

示例中代码的前几行与前一示例中的前几行代码极为相似。部分原因是我们通过复制和粘贴将代码从一个文件复制到另一个文件（但是这是无关紧要的因素）。主要原因是两个示例在可以与数据库进行会话前都需要一些共同的创建步骤。我们有与之前的示例相同的加载语句。我们有一个db对象，该对象从create_datebase()函数返回。我们有一个store对象，通过传递db对象到Store构造器来创建。但是现在，不仅添加一个对象到store中，还调用store对象的find()方法。这个特别的调用find()（例如，store.find(storm_model.OperatingSystem)）返回所有storm_model.OperatingSystem对象的结果集。因为我们映射OperationSystem类到inventory_operationsystem表，Storm将查看所有在inventory_operatingsystem表中的相关记录，从中创建OperationSystem对象。对于每一个OPerationSystem对象，我们输出id、name和description属性。这些属性映射到数据库的列值上，且数据库对于每一个记录共享相同的名称。

通过在SQLite一节中的示例，我们应该已经在数据库中有一个记录。让我们查看一下，当我们运行提取脚本时会发生什么。我们应该期望它显示一个记录，哪怕这个记录没有使用Storm库进行插入：

```
jmjones@dinkgutsy:~/code$ python storm_retrieve_os.py
1 Linux 2.0.34 kernel
```

这实际上是我们期望发生的。现在当我们运行add脚本然后再运行retrieve脚本，那么会

发生什么情况呢？它会显示在比较早的数据库中的旧记录（Linux2.0.34 kernet），以及新插入的记录（Windows 3.1.1）

```
jmjones@dinkgutsy:~/code$ python storm_add_os.py
jmjones@dinkgutsy:~/code$ python storm_retrieve_os.py
1 Linux 2.0.34 kernel
2 Windows 3.1.1
```

再说一次，这确实是我们所希望的。

那么，如果你希望过滤数据，又会怎么样呢？假设我们仅希望查看操作系统中以字符串“Lin”开始的记录项。以下代码示例完成了该想法：

```
import storm.locals
import storm_model
import os

db = storm.locals.create_database('sqlite:///%s' % os.path.join(os.getcwd(),
  'inventory.db'))

store = storm.locals.Store(db)

for o in store.find(storm_model.OperatingSystem,
  storm_model.OperatingSystem.name.like(u'Lin%')):
    print o.id, o.name, o.description
```

这个示例与之前示例相似，都使用了store.find()，只是这里传递了第二个参数到stoe.find()中：一个搜索标准。Store.find(storm_model.OperatingSystem,storm_model.OperatingSystem.name.like(u'Lin%'))“告诉Storm查看所有的Operationsystem对象，查看该对象的name属性值是否以unicode值“Lin”开头。对于每一个在结果集中的值，我们打印输出，这等同于之前的示例。

当你运行它时，你将看到这样的结果：

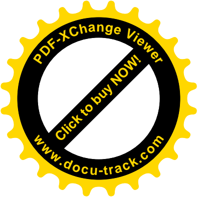
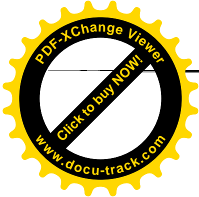
```
jmjones@dinkgutsy:~/code$ python storm_retrieve_os_filter.py
1 Linux 2.0.34 kernel
```

这个数据库仍然具有“windows 3.1.1”的记录项，但是它被过滤掉了，因为“Windows”不是以“Lin”开头的。

SQLAlchemy ORM

随着Storm获得人们的接受并且拥有了自己的社区，SQLAlchemy似乎也一时占据了Python中ORM的主导地位。它的方法与Storm相似，或许更好的表述应该是“Storm的方法与SQLAlchemy相似”，因为SQLAlchemy排在第一位。不管怎么样，我们将为

www.TopSage.com

SQLAlchemy介绍相同的inventory_operationsystem示例（该示例是我们为Storm完成的）。

以下是为inventory_operatingsystem表定义的表格和对象：

```
#!/usr/bin/env python

import os
from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, Text, VARCHAR, MetaData
from sqlalchemy.orm import mapper
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///%s' % os.path.join(os.getcwd(),
        'inventory.db'))

metadata = MetaData()
os_table = Table('inventory_operatingsystem', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', VARCHAR(50)),
    Column('description', Text()),
)

class OperatingSystem(object):
    def __init__(self, name, description):
        self.name = name
        self.description = description

    def __repr__(self):
       return "<OperatingSystem('%s','%s')>" % (self.name, self.description)

mapper(OperatingSystem, os_table)
Session = sessionmaker(bind=engine, autoflush=True, transactional=True)
session = Session()
```

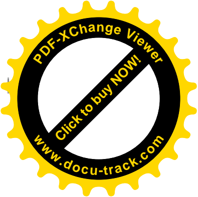
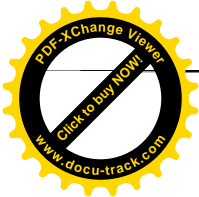
在Storm和SQLAlchemy示例的表定义代码之间最大的差异是，SQLAlchemy使用额外的类而不是table类，并且在这两者之间映射。

现在已经有了一个表定义，可以写一些代码来查询表中的所有记录：

```
#!/usr/bin/env python

from sqlalchemy_inventory_definition import session, OperatingSystem

for os in session.query(OperatingSystem):
    print os
```

如果现在运行它，看到结果如下：

```
$ python sqlalchemy_inventory_query_all.py <OperatingSystem('Linux','2.0.34 kernel')>
<OperatingSystem('Windows','3.1.1')>
</OperatingSystem></OperatingSystem>
```

如果希望创建另一个记录，可以简单地这样做：实例化一个OperationgSystem对象，然后将其添加到会话中：

```
#!/usr/bin/env python

from sqlalchemy_inventory_definition import session, OperatingSystem

ubuntu_710 = OperatingSystem(name='Linux', description='2.6.22-14 kernel')
session.save(ubuntu_710)
session.commit()
```

这会添加另一个Linux内核到表格中，这次是一个更新的内核。再次运行query all脚本，可以看到以下输出：

```
$ python sqlalchemy_inventory_query_all.py
<OperatingSystem('Linux','2.0.34 kernel')>
<OperatingSystem('Windows','3.1.1')>
<OperatingSystem('Linux','2.6.22-14 kernel')>
```

在SQLAlchemy中过滤结果也是非常简单的。例如，如果希望过滤所有的OperatingSystem中，name以“Lin”开头的记录，应该写一个类似这样的脚本：

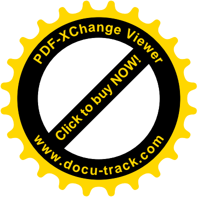
```
#!/usr/bin/env python

from sqlalchemy_inventory_definition import session, OperatingSystem

for os in session.query(OperatingSystem).filter(OperatingSystem.name.like('Lin%')):
    print os
```

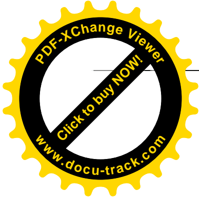
可以看到输出类似这样：

```
$ python sqlalchemy_inventory_query_filter.py
<OperatingSystem('Linux','2.0.34 kernel')>
<OperatingSystem('Linux','2.6.22-14 kernel')>
```

这只是一个简单的关于SQLAlchemy可以完成什么工作的概述。更多的使用SQLAlchemy的信息可以访问网站*http://www.sqlalchemy.org/*，或者你可以查看由 Rick Copeland编著的《Essential SQLAlchemy》（O'Reilly出版）。

www.TopSage.com





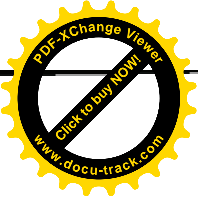
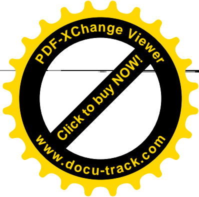
名人简介：SQLAlchemy

Mike Bayer

Michael Bayer效力于NYC-based software，具有处理各种类型和规模的关系数据库的经验。在用C，Java和Perl编写了一些数据库抽象层之后，并具备了多年使用大规模多服务器Oracle系统（为Major League Baseball提供服务）的实践经验之后，编写了SQLAchemy，并将其作为"最终工具集"。SQLAchemy用于产生SQL以及处理数据库，其开发目标是面向世界级的一流Python工具集，帮助创建Python通用流行的编程平台。

本章小结

本章中，我们介绍了一些各不相同的工具，这些工具允许你保存数据以备今后使用。有时你需要一些简单、轻便，类似pickle这样的模块。有时你需要更全面的类似SQLAchemy ORM这样的工具。正如我们所演示的，使用Python时你可以有多种选择，可以非常简便地完成复杂而庞大的工作。

第13章

命令行

引言

命令行与系统管理之间有着特殊的关系。没有其他的工具可以如命令行一样重要和受人喜爱。对命令行的完全掌握是绝大多数系统管理员所走过的必经之路。许多系统管理员很少考虑使用“GUI”来完成工作，宣称GUI管理只起辅助作用。这或许不完全正确，但是这通常是真正领会系统管理艺术的系统管理员的普遍观点。

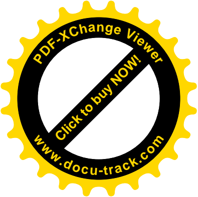
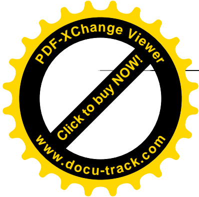
很久以前，UNIX系统包含这样的观点，命令行界面（CLI）优于任何可以开发出来的GUI。在最近的一些事情中，似乎Microsoft也回到了这一观点上来。Jeffrey Snover是Windows Powershell的架构师，他曾说“认为GUI将会、甚至是必将取代CLI是错误的”。

甚至Windows中多年来也一直包含符合现代操作系统特点的最基本的CLI。现在在其当前的Windows PowerShell中可以看到CLI的价值。我们不会在本书中介绍Windows，但这是非常有趣的事实，即掌握命令行是如此重要，创建命令行工具是如此重要。

除了掌握预建Unix命令行工具之外，还有更多的事情。为了真正成为一个命令行高手，你需要创建自己的工具，并且这或许是你在第一时刻拿起这本书的唯一原因。不要担心，这一章不会令你失望。在学习完之后，你将成为在Python中创建命令行工具的大师。

将创建命令行工具放在本书的最后部分是有意的安排。原因是首先让你知道各种各样的Python技术，最后教你如何利用所有这些技术来发挥你的力量，创建一个优秀的命令行工具。

www.TopSage.com

基本标准输入的使用

对于创建命令行工具最简单的介绍或许是需要知道sys模块能够通过sys.argv处理命令行参数。例13-1显示了一些可能是最简单的命令行工具。

例13-1：sysargv.py

```
#!/usr/bin/env python

import sys
print sys.argv
```

在执行命令之后，无论你在命令行输入什么，这两行代码返回到标准输出：

```
./sysargv.py
```

```
['./sysargv.py']
```

以及

```
./sysargv.py foo
```

返回到标准输出

```
['./sysargv.py', 'test']
```

以及

```
./sysargv.py foo bad for you
```

返回到标准输出

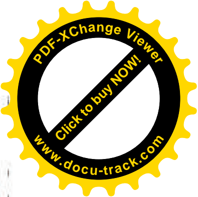
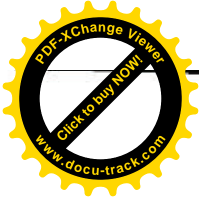
```
['./sysargv.py', 'foo', 'bad', 'for', 'you']
```

让我们更专注一些，略微修改代码来对命令行的参数进行记数，如例13-2。

例13-2：sysargv.py

```
#!/usr/bin/env python
import sys

#Python indexes start at Zero, so let's not count the command itself which is
#sys.argv[0]

num_arguments = len(sys.argv) - 1
print sys.argv, "You typed in ", num_arguments, "arguments"
```

你或许正在考虑，“哇，这非常简单，所有需要做的仅是通过数字引用sys.argv参数，并写一些逻辑对它们进行连接”。你是对的，这样做非常简单。让我们在命令行应用中

添加一些特征。最后一件我们可以做的事是，如果没有参数传递给命令行，发送一个错误信息到标准输出。参见例13-3。

例13-3：sysargv-step2.py

```
#!/usr/bin/env python
import sys

num_arguments = len(sys.argv) - 1

#If there are no arguments to the command, send a message to standard error.
if num_arguments == 0:
        sys.stderr.write('Hey, type in an option silly\n')

else:
        print sys.argv, "You typed in ", num_arguments, "arguments"
```

使用sys.argv来创建命令行工具非常快速，但是经常也会是错误的选择。Python标准库包括optparse模块，该模块处理所有创建一个高质量的命令行工具所遇到的杂乱和恼人的部分。甚至对于小的"演示型"工具，使用optparse而不是sys.argv，也是一个较好的选择。通常"演示型"工具也会发展为产品型工具。在下一节，我们将解释这是为什么，但是一个简短的答案是：一个好的选项解析模块可以为你处理复杂事务。

TopSage.com

Optparse简介

正如我们在前一节中所介绍的，即使是非常小的脚本也可以从使用optparse来处理选项获得收益。一个开始optparse介绍的非常有意义的方式是编写一个"Hello World"示例，该示例处理选项和参数。例13-4是我们的Hello World示例。

例13-4：Hello World optparse

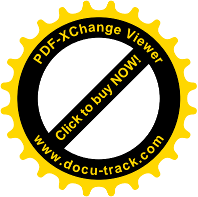
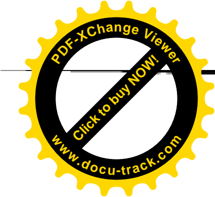
```
#!/usr/bin/env python
import optparse

def main():
    p = optparse.OptionParser()
    p.add_option('--sysadmin', '-s', default="BOFH")
    options, arguments = p.parse_args()
    print 'Hello, %s' % options.sysadmin

if __name__ == '__main__':
    main()
```

当运行这个程序的时候，获得下面不同类型的输出：

```
$ python hello_world_optparse.py
Hello, BOFH
```

```
$ python hello_world_optparse.py --sysadmin Noah
Hello, Noah
```


www.TopSage.com

```
$ python hello_world_optparse.py --s Jeremy
Hello, Jeremy

$ python hello_world_optparse.py --infinity Noah
Usage: hello_world_optparse.py [options]

hello_world_optparse.py: error: no such option: --infinity
```

在我们的小脚本中，可以设置短的“-s”选项，以及长的“--sysadmin”选项，也可以是默认选项。最后，当我们错误地输入一个选项时，我们看一下内建错误处理的强大功能，这在Perl中是没有的。

简单的Optparse使用模式

非选项使用模式

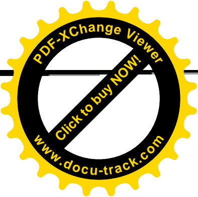
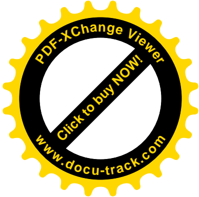
前一节中提到，甚至对于小脚本，optparse也是非常有用的。例13-5是简单的optparse使用模式，在这里甚至没有使用选项，但是仍然发挥了optparse的长处。

例13-5：复制ls命令

```
#!/usr/bin/env python
import optparse
import os

def main():
    p = optparse.OptionParser(description="Python 'ls' command clone",
                              prog="pyls",
                              version="0.1a",
                              usage="%prog [directory]")
    options, arguments = p.parse_args()
    if len(arguments) == 1:
        path = arguments[0]
        for filename in os.listdir(path):
            print filename
    else:
        p.print_help()
if __name__ == '__main__':
    main()
```

在这个示例中，我们重新在Python中实现了ls命令，只是这里仅设置了一个参数，该参数是执行ls命令的路径。我们甚至没有使用选项，但是仍可以通过依赖optparse来处理我们的程序流。在创建一个optparse实例时，首先提供一些实现的方法，添加一个usage（用法）值，该值指导工具的潜在用户如何正确执行它。接下来，我们检查以确保参数的个数只有一个，如果有多于或少于一个参数，则使用内建的help信息p.print_help()来显示简介，该简介介绍了如何再次使用工具。以下是我们在当前目录（或“.”）正确地运行它时得到的输出结果。

www.TopSage.com

```
$ python no_options.py .
.svn
hello_world_optparse.py
no_options.py
```

接下来，查看在不输入任何选项时的结果：

```
$ python no_options.py
Usage: pyls [directory]

Python 'ls' command clone

Options:
--version    show program's version number and exit
-h, --help   show this help message and exit
```

这其中有意义的地方是，如果参数不只是一个，使用p.print_help()调用定义行为，这与我们键入“--help”是完全相同的：

```
$ python no_options.py --help
Usage: pyls [directory]

Python 'ls' command clone

Options:
--version show program's version number and exit
-h, --help show this help message and exit
```

因为我们定义了一个“--version”选项，我们可以看到以下的输出结果：

```
$ python no_options.py --version
0.1a
```

在这个示例中，optparse是发挥作用的，甚至是在简单的“演示性”脚本中，optarse也同样有用（或许是你打算弃用的脚本）。

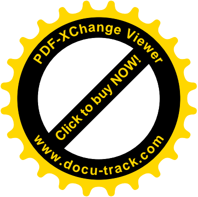
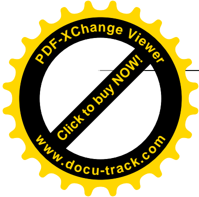
True/False使用模式

在程序中使用一个选项来设置True或False是非常有帮助的。这其中的经典示例涉及设置“--quite”选项（该参数关闭所有标准输出）和“--verbose”选项（将触发额外输出）。例13-6演示的这一过程。

例13-6：增加与减少冗余

```
#!/usr/bin/env python
import optparse
import os

def main():
    p = optparse.OptionParser(description="Python 'ls' command clone",
                              prog="pyls",
```


www.TopSage.com

```
                                   version="0.1a",
                                   usage="%prog [directory]")
    p.add_option("--verbose", "-v", action="store_true",
                help="Enables Verbose Output",default=False)
    options, arguments = p.parse_args()
    if len(arguments) == 1:
        if options.verbose:
            print "Verbose Mode Enabled"
        path = arguments[0]
        for filename in os.listdir(path):
            if options.verbose:
                print "Filename: %s " % filename
            elif options.quiet:
                pass
            else:
                print filename
    else:
        p.print_help()
if __name__ == '__main__':
    main()
```

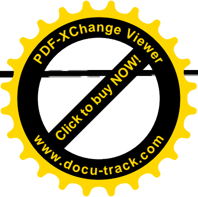
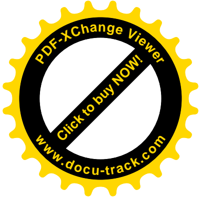
通过使用“--verbose”，我们有效地设置stdout的冗余等级。让我们看一下每一冗余等级的情况。首先是正常方式：

```
$ python true_false.py /tmp
.aksusb
alm.log
amt.log
authTokenData
FLEXnet
helloworld
hsperfdata_ngift
ics10003
ics12158
ics13342
icssuis501
MobileSync.lock.f9e26440fe5adbb6bc42d7bf8f87c1e5fc61a7fe
summary.txt
```

接下来是“--verbose”模式：

```
$ python true_false.py --verbose /tmp
Verbose Mode Enabled
Filename: .aksusb
Filename: alm.log
Filename: amt.log
Filename: authTokenData
Filename: FLEXnet
Filename: helloworld
Filename: hsperfdata_ngift
Filename: ics10003
Filename: ics12158
Filename: ics13342
Filename: icssuis501
```

www.TopSage.com

```
Filename: MobileSync.lock.f9e26440fe5adbb6bc42d7bf8f87c1e5fc61a7fe
Filename: summary.txt
```

当我们设置“--verbose”选项，options.verbose的值为True，作为结果，我们的条件语句被执行，在实际的文件名前输出“Filename”。值得注意的是，在我们的脚本中，设置“default=False”以及“action="store_true"”，这等同于设置默认值为False，除非有人指定了“--option”，且设置选项option的值为True。这是在optparse中有关使用True/False选项的最为精华的内容。

记数选项使用模式

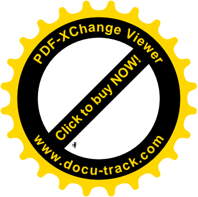
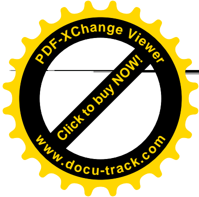
在一个典型的Unix命令行工具中，例如tcpdump，如果你指定了-vvv，相比于仅使用-v或-vv将获得更多的冗余输出。你可以让optparse做相同的事情，通过添加一个计数器，对每次指定的选项计数。例如，如果你希望在你的工具中添加同样级别的冗余，可以参见例13-7。

例13-7：记数选项使用模式

```
#!/usr/bin/env python
import optparse
import os

def main():
    p = optparse.OptionParser(description="Python 'ls' command clone",
                              prog="pyls",
                              version="0.1a",
                              usage="%prog [directory]")
    p.add_option("-v", action="count", dest="verbose")
    options, arguments = p.parse_args()
    if len(arguments) == 1:
        if options.verbose:
            print "Verbose Mode Enabled at Level: %s" % options.verbose
        path = arguments[0]
        for filename in os.listdir(path):
            if options.verbose == 1:
                print "Filename: %s " % filename
            elif options.verbose ==2 :
                fullpath = os.path.join(path,filename)
                print "Filename: %s | Byte Size: %s" % (filename,
                                              os.path.getsize(fullpath))
            else:
                print filename
    else:
        p.print_help()
if __name__ == '__main__':
    main()
```

通过使用一个自动增加计数的设计模式，可以确保仅一个选项，却可以做三件不同的事情。第一次调用中使用“-v”，它设置options.verbose为1，如果使用“--v”，它设

www.TopSage.com

置options.verbose为2。在实际的程序中，没有选项，仅输出文件名；使用“-v”将输出单词Filename以及文件名；使用“-vv”输出字节数以及文件名。以下是使用“-vv”的输出结果：

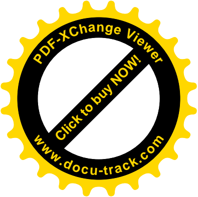
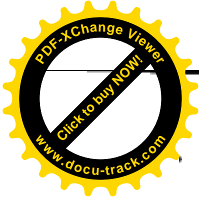
```
$ python verbosity_levels_count.py -vv /tmp
Verbose Mode Enabled at Level: 2
Filename: .aksusb | Byte Size: 0
Filename: alm.log | Byte Size: 1403
Filename: amt.log | Byte Size: 3038
Filename: authTokenData | Byte Size: 32
Filename: FLEXnet | Byte Size: 170
Filename: helloworld | Byte Size: 170
Filename: hsperfdata_ngift | Byte Size: 102
Filename: ics10003 | Byte Size: 0
Filename: ics12158 | Byte Size: 0
Filename: ics13342 | Byte Size: 0
Filename: ics14183 | Byte Size: 0
Filename: icssuis501 | Byte Size: 0
Filename: MobileSync.lock.f9e26440fe5adbb6bc42d7bf8f87c1e5fc61a7fe | Byte Size: 0
Filename: summary.txt | Byte Size: 382
```

选择使用模式

有时展示选项的一些选择是比较容易的。在上一个示例中，我们创建了选项“--verbose”和“--quiet”，我们可以让其从“--chatty”选项的结果中进行选择。使用之前的示例，例13-8展示了使用新选项时的情况。

例13-8：选择使用模式

```
#!/usr/bin/env python
import optparse
import os

def main():
    p = optparse.OptionParser(description="Python 'ls' command clone",
                              prog="pyls",
                              version="0.1a",
                              usage="%prog [directory]")
    p.add_option("--chatty", "-c", action="store", type="choice",
                 dest="chatty",
                 choices=["normal", "verbose", "quiet"],
                 default="normal")
    options, arguments = p.parse_args()
    print options
    if len(arguments) == 1:
        if options.chatty == "verbose":
            print "Verbose Mode Enabled"
        path = arguments[0]
        for filename in os.listdir(path):
            if options.chatty == "verbose":
                print "Filename: %s " % filename
```

```
            elif options.chatty == "quiet":
                pass
            else:
                print filename
    else:
        p.print_help()
if __name__ == '__main__':
    main()
```

如果就像之前示例中所演示的那样，运行不带选项的命令，会得到下面的错误结果：

```
$ python choices.py --chatty
Usage: pyls [directory]

pyls: error: --chatty option requires an argument
```

如果给选项指定了错误的参数，会得到另一个错误，告诉我们可用的选项：

```
$ python choices.py --chatty=nuclear /tmp
Usage: pyls [directory]

pyls: error: option --chatty: invalid choice: 'nuclear' (choose from 'normal',
'verbose', 'quiet')
```

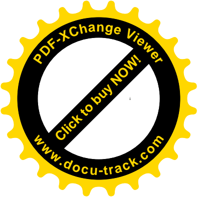
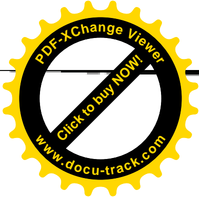
使用选项的一个方便之处是能够减少对用户输入正确的命令参数的依赖。用户仅可以从指定的选项中进行选择。以下是当命令正确运行时，命令的执行结果：

```
$ python choices.py --chatty=verbose /tmp
{'chatty': 'verbose'}
Verbose Mode Enabled
Filename: .aksusb
Filename: alm.log
Filename: amt.log
Filename: authTokenData
Filename: FLEXnet
Filename: helloworld
Filename: hsperfdata_ngift
Filename: ics10003
Filename: ics12158
Filename: ics13342
Filename: ics14183
Filename: icssuis501
Filename: MobileSync.lock.f9e26440fe5adbb6bc42d7bf8f87c1e5fc61a7fe
Filename: summary.txt
```

或许你注意到，在顶部的输出中有“chatty”作为关键字和“verbose”作为值。在上面的示例中，我们为选项放一个print语句来显示其在程序中的样子。以下是一个最终使用“--chatty”和“quiet”选项的示例：

```
$ python choices.py --chatty=quiet /tmp
{'chatty': 'quiet'}
```


www.TopSage.com

具有多参数的选项使用模式

默认情况下，一个optparse选项只能有一个参数，但是可以指定参数的个数也是可以的。例13-9是一个精心设计的示例，其中我们让ls命令同时输出两个目录的内容。

例13-9：对两个目录列表输出

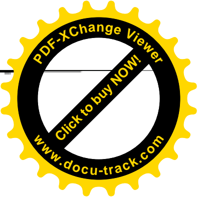
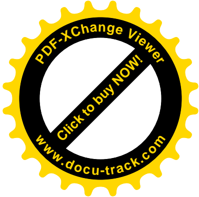
```
#!/usr/bin/env python
import optparse
import os

def main():
    p = optparse.OptionParser(description="Lists contents of two directories",
                              prog="pymultils",
                              version="0.1a",
                              usage="%prog [--dir dir1 dir2]")
    p.add_option("--dir", action="store", dest="dir", nargs=2)
    options, arguments = p.parse_args()
    if options.dir:
        for dir in options.dir:
            print "Listing of %s:\n" % dir
            for filename in os.listdir(dir):
                print filename
    else:
        p.print_help()
if __name__ == '__main__':
    main()
```

如果仅为这个命令“--dir”指定一个参数，从输出结果中会看到下面的错误信息：

```
[ngift@Macintosh-8][H:10238][J:0]# python multiple_option_args.py --dir /tmp ¬
Usage: pymultils [--dir dir1 dir2]

pymultils: error: --dir option requires 2 arguments
```

如果正确指定了“--dir”选项的参数个数，会得到如下的结果：

```
pymultils: error: --dir option requires 2 arguments
[ngift@Macintosh-8][H:10239][J:0]# python multiple_option_args.py --dir /tmp
  /Users/ngift/Music
Listing of /tmp:

.aksusb
FLEXnet
helloworld
hsperfdata_ngift
ics10003
ics12158
ics13342
ics14183
ics15392
icssuis501
MobileSync.lock.f9e26440fe5adbb6bc42d7bf8f87c1e5fc61a7fe
```

```
summary.txt
Listing of /Users/ngift/Music:

.DS_Store
.localized
iTunes
```

Unix Mashups：整合Shell命令到Python命令行工具中

在第10章，我们看到了一些使用subprocess模块的通用方法。通过使用Python以及修改它们的API，来封装已有的命令行工具以创建新的命令行工具。或是混合一个或多个使用Python的Unix命令行工具来创建一个新的命令行工具。这些创建的新工具提供了一个非常有意义的检测方法。封装一个已有的Python命令行工具或修改其行为以适应特定的需要是非常简单的。你或许选择整合一个包含了一些参数（该参数为你使用的选项所需要）的配置文件，或是你选择为其他选项创建默认。不管需求如何，毫无问题，你可以使用subprocess和optparse来修改一个本地UNIX工具。

另外，混合一个命令行工具和纯Pyton可以创建更有意义的工具，而这在C或是Bash中是无法轻松做到的。混合dd命令和线程、队列、tcpdump和Python的正则表达式库或是一个自定义的rsync版本，怎么样？这些Unix2.0的“mashups”与Web2.0非常相似。通过混合Python和Unix工具，我们有了新的思路，问题可以用不同的方式来解决。在这一节，我们探索一些这方面的技术。

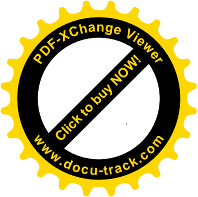
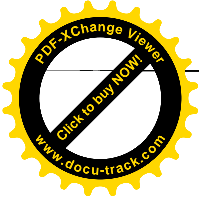
Kudzu使用模式： Python中的封装工具

有时，你发现自己正在使用一个命令行工具，而该工具不是你真正想要的。它或许需要太多的选项，或许参数的顺序与你希望使用的顺序相反。使用Python修改一个工具的功能，使其能够做你希望完成的工作是非常简便的。我们喜欢称其为“Kudzu”设计模式。或许你还不熟悉Kudzu，Kudzu就像是一根快速生长的藤蔓，由日本发展到美国南部。Kudzu遵循自然习惯，创建可选的场景。使用Python，并且如果你选择了Kudzu，就可以在你的Unix环境中做相同的事情。

对于这个示例，我们将封装snmpdf命令与Python，以简化使用。首先查看一下正常运行snmpdf时的输出结果：

```
[ngift@Macintosh-8][H:10285][J:0]# snmpdf -c public -v 2c example.com
Description              size (kB)         Used      Available Used%
Memory Buffers             2067636       249560        1818076    12%
Real Memory                2067636      1990704          76932    96%
Swap Space                 1012084           64        1012020     0%
```


www.TopSage.com

```
/                          74594112      17420740      57173372   23%
/sys                              0             0             0    0%
/boot                        101086         20041         81045   19%
```

或许你还不熟悉snmpdf，这表示在允许SNMP的远端系统中运行，并且允许访问MIB树中的磁盘部分。通常，处理SNMP协议的命令行工具有许多选项，这些选项使得它们难于使用。坦率地讲，创建的工具必须能与SNMP版本1、2和3相兼容。如果不考虑这个，可以说你是一个非常“懒惰”的人。你或许希望制作自己的snmpdf的“Kudzu”版本，且仅将一台主机作为参数。确实是可以这样做，例13-10演示了它的样子。

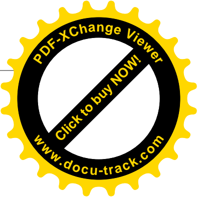
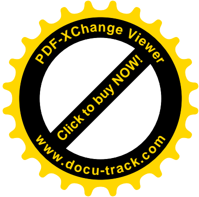
注意： 通常，当你在Python中封装一个Unix工具来修改工具的行为时，它会变得比你使用Bash来进行修改所需的代码行还要多。我们感觉到这是一个成功之处，因为正如你所看到的，它允许你使用更丰富的Python工具集来扩展这个工具。另外，你可以按测试你写的其他工具相同的方法对代码进行测试，往往额外多出的代码正是优势所在。

例13-10：使用Python封装 SNMPDF

```
#!/usr/bin/env python
import optparse
from subprocess import call

def main():
    p = optparse.OptionParser(description="Python wrapped snmpdf command",
                                prog="pysnmpdf",
                                version="0.1a",
                                usage="%prog machine")
    p.add_option("-c", "--community", help="snmp community string")
    p.add_option("-V", "--Version", help="snmp version to use")
    p.set_defaults(community="public",Version="2c")
    options, arguments = p.parse_args()
    SNMPDF = "snmpdf"
    if len(arguments) == 1:
        machine = arguments[0]
        #Our new snmpdf action
        call([SNMPDF, "-c", options.community, "-v",options.Version, machine])
    else:
        p.print_help()
if __name__ == '__main__':
    main()
```

这段脚本大约二十行，但是却能让我们的工作变得简单。通过使用optparse，可以创建具有默认参数的选项，且默认参数会匹配需要。例如，设置SNMP版本选项为默认的版本2（据我们所知道的，我们的数据中心目前仅使用版本2）。例如，设置字符串community为“public”，因为这是在我们的研究和开发实验中所设置的。一个使用optparse进行处理，且不使用硬代码脚本的好处是，我们具有修只需改选项而不修改脚本的灵活性。

www.TopSage.com

值得注意的是，默认参数的设置使用set_defaults方法，这允许我们同时对命令行工具的所有参数设置默认值。同时需要注意subprocess.call的使用。我们嵌入旧的选项，例如“-c”，并且封装新的值来满足需要。在这个示例中该值来自optparse或是options.community。这一技术强调了Python中“Kudzu”的强大作用，可以融合其他工具并进行修改，以满足需要。

混合Kudzu设计模式：封装一个Python工具，并修改其行为

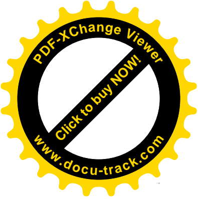
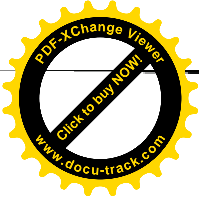
在最后的示例中，snmpdf使用起来变得非常容易，但是我们没有修改工具的基本行为。这两个工具的输出是相同的。另一个我们可以使用的方法是不仅包含一个Unix工具，并使用Python修改工具的基本行为。

在接下来的示例中，我们在函数中使用Python生成器，来过滤我们的snmpdf命令搜索紧急信息的结果，然后附加“CRITICAL”标志。例13-11演示了这一过程。

例13-11：使用generator修改 SNMPDF 命令

```
#!/usr/bin/env python
import optparse
from subprocess import Popen, PIPE
import re

def main():
    p = optparse.OptionParser(description="Python wrapped snmpdf command",
                              prog="pysnmpdf",
                              version="0.1a",
                              usage="%prog machine")
    p.add_option("-c", "--community", help="snmp community string")
    p.add_option("-V", "--Version", help="snmp version to use")
    p.set_defaults(community="public",Version="2c")
    options, arguments = p.parse_args()
    SNMPDF = "snmpdf"
    if len(arguments) == 1:
        machine = arguments[0]

        #We create a nested generator function
        def parse():
            """Returns generator object with line from snmpdf"""
            ps = Popen([SNMPDF, "-c", options.community,
                        "-v",options.Version, machine],
                        stdout=PIPE, stderr=PIPE)
            return ps.stdout

        #Generator Pipeline To Search For Critical Items
        pattern = "9[0-9]%"
        outline = (line.split() for line in parse()) #remove carriage returns
        flag = (" ".join(row) for row in outline if re.search(pattern, row[-1]))
        #patt search, join strings in list if match
        for line in flag: print "%s CRITICAL" % line
        #Sample Return Value
        #Real Memory 2067636 1974120 93516 95% CRITICAL
```

www.TopSage.com

```
    else:
        p.print_help()
if __name__ == '__main__':
    main()
```

如果运行新的snmpdf“修改”版，会在测试机上看到这样的结果：

```
[ngift@Macintosh-8][H:10486][J:0]# python snmpdf_alter.py localhost
Real Memory 2067636 1977208 90428 95% CRITICAL
```

我们现在具有一个完全不同的脚本，如果在snmpdf中的某个值是90%（标识为临界区）或更高将产生输出。我们可以在晚间在数百台主机上在cron作业中运行这个脚本，如果有一个来自该脚本的返回值则发送电子邮件。另外，我们可以进一步扩展这个脚本，设定搜索使用等级为80%、70%，并在达到这些等级时产生警告。将这一技术整合到Google App Engine也是一件容易的事情，例如我们可以创建一个web应用程序监测基础结构中的磁盘使用情况。

查看一下代码，有一些与之前的示例不一样之处需要指出。第一处不同是使用subprocess.Popen而不是使用subprocess.call。如果希望解析一个Unix命令行工具的输出，那么subprocess.Popen是你需要的。值得注意的是，我们使用了stdout.readlines()，它返回一个列表而不是一个字符串。当我们取得输出并通过一系列的generator表达式进行传输时，这是非常重要的。

在generator管道一节，我们将传送generator对象到两个表达式，来查找一个符合我们设置的标准匹配。正如我们之前所说的，我们可以很容易地添加一些generator，以取得阈值在70%与80%之间的结果。

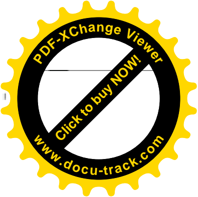
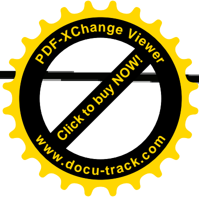
注意： 这个工具比起你希望实现的一个产品化工具或许更复杂。一个较好的想法或许是将其分解为多个小一些的一系列可以加载的小块。

混合Kudzu设计模式：封装Python中的Unix工具来产生进程

上一个示例非常棒，但以一种高效的模式创建多个副本，是另一个非常有意义的改变现有Unix工具行为的方法。确实，这似乎有点令人捉摸不透。然而，有时你需要对工作充满想象力和创造性。这是系统管理中非常有趣的一部分，你必须做一些疯狂的事情来解决产品化中的问题。

在数据一章，我们创建了一个检测脚本，该脚本并行使用dd命令创建映像文件。让我们采用这一思想并运行它，创建一个永久的可以多次复用的命令行工具。最后，我们检测一个新的文件服务器，可以进行一些操作来缩短磁盘I/O时间。参见例13-12。

www.TopSage.com

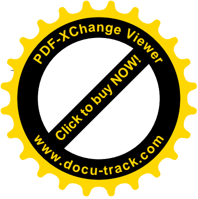
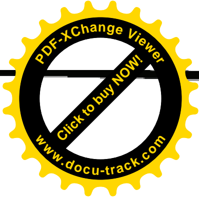
例13-12：多dd命令

```
from subprocess import Popen, PIPE
import optparse
import sys

class ImageFile():
    """Created Image Files Using dd"""
    def __init__(self, num=None, size=None, dest=None):
        self.num = num
        self.size = size
        self.dest = dest

    def createImage(self):
        """creates N 10mb identical image files"""
        value = "%sMB " % str(self.size/1024)
        for i in range(self.num):
            try:
                cmd = "dd if=/dev/zero of=%s/file.%s bs=1024 count=%s"\
                    % (self.dest,i,self.size)
                Popen(cmd, shell=True, stdout=PIPE)
            except Exception, err:
                sys.stderr.write(err)

    def controller(self):
        """Spawn Many dd Commands"""
        p = optparse.OptionParser(description="Launches Many dd",
                                    prog="Many dd",
                                    version="0.1",
                                    usage="%prog [options] dest")
        p.add_option('-n', '--number', help='set many dd',
                    type=int)
        p.add_option('-s', '--size', help='size of image in bytes',
                    type=int)
        p.set_defaults(number=10,
                        size=10240)
        options, arguments = p.parse_args()
        if len(arguments) == 1:
            self.dest = arguments[0]
            self.size = options.size
            self.num = options.number
            #runs dd commands
            self.createImage()

def main():
    start = ImageFile()
    start.controller()

if __name__ == "__main__":
    main()
```

现在，如果运行多个dd命令，可以设置文件的字节数、路径和总的文件或总的进程数。以下是其运行的结果：

```
$ ./subprocess_dd.py /tmp/
$ 10240+0 records in
```


www.TopSage.com

```
10240+0 records out
10485760 bytes transferred in 1.353665 secs (7746199 bytes/sec)
10240+0 records in
10240+0 records out
10485760 bytes transferred in 1.793615 secs (5846160 bytes/sec)
10240+0 records in
10240+0 records out
10485760 bytes transferred in 2.664616 secs (3935186 bytes/sec)

...output supressed for space....
```

这一混合工具的快速使用，将检测磁盘的I/O性能，包括高速光纤SAN或是NAS设备。只需一点工作量，你可以添加hook以产生PDF报告并邮寄结果。有一点需要指出，如果线程看起来适合你需要解决的问题，那么相同的事情也可以通过线程来成功实现。

整合配置文件

整合一个配置文件到一个命令行工具，或许在可用性以及未来的自定义方面导致些许不同。将可用性与命令行放在一起谈论似乎有点奇怪，因为它们经常仅因为GUI或web工具才被提及。就像一个命令行工具与一个GUI工具引起了同样的注意，这是令人遗憾的。

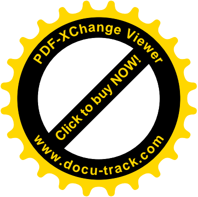
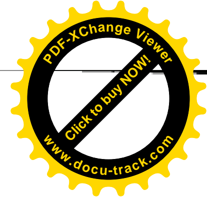
配置文件对于中心化在多主机上运行的命令行工具，可以说是非常有帮助的。配置文件可以通过加载NFS实现共享，然后数百台主机可以通过创建的一个普通的命令行工具读取这个配置文件。另外，或许有一些类型的配置管理系统，可以给你创建的工具发布配置文件。

对于使用*.ini*语法读取和编写配置文件，Python标准库有一个非常不错的模块ConfigParser。*.ini*格式是一个好的媒介，用于读取和写入简单的配置数据，不必使用XML，不需要限制编辑该文件的人必须懂得Python语言。请参阅前一章以获得更多的使用ConfigParser模块的细节信息。

注意： 确信你不必养成依赖配置文件中的条目顺序的习惯。ConfigParser模块使用字典，你需要以正确获得映射的方式引用它。

在开始整合配置文件到命令行工具的时，我们将创建一个“hello world”配置文件。命名文件名为*hello_config.ini*并粘贴下面的内容：

```
[Section A]
phrase=Config
```

现在已经有一个简单的配置文件，可以整合该配置文件到之前的Hello World命令行工具中，如例13-13所示。

例13-13：Hello 配置文件命令行工具

```
#!/usr/bin/env python
import optparse
import ConfigParser

def readConfig(file="hello_config.ini"):
    Config = ConfigParser.ConfigParser()
    Config.read(file)
    sections = Config.sections()
    for section in sections:
        #uncomment line below to see how this config file is parsed
        #print Config.items(section)
        phrase = Config.items(section)[0][1]
        return phrase

def main():
    p = optparse.OptionParser()
    p.add_option('--sysadmin', '-s')
    p.add_option('--config', '-c', action="store_true")
    p.set_defaults(sysadmin="BOFH")

    options, arguments = p.parse_args()
    if options.config:
        options.sysadmin = readConfig()
    print 'Hello, %s' % options.sysadmin

if __name__ == '__main__':
    main()
```

如果不带任何选项运行这个工具，我们获得一个默认值BOFH，就像原始的“hello world”程序一样：

```
[ngift@Macintosh-8][H:10543][J:0]# python hello_config_optparse.py
Hello, BOFH
```

如果选择“--config file”，解析配置文件并得到如下响应：

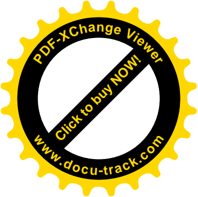
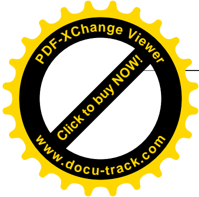
```
[ngift@Macintosh-8][H:10545][J:0]# python hello_config_optparse.py --config
Hello, Config
```

注意：大多数情况下，你可能希望为--config选项设置一个默认路径，允许自定义读取文件的位置。你可以根据以下操作进行设置，而不是直接将选项设置为default_true：

```
p.add_option('--config', '-c',
help='Path to read in config file')
```

如果这是一个大一些的，并且实际上非常实用的程序，可以将它传给不懂Python的人。

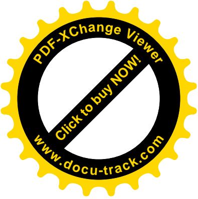
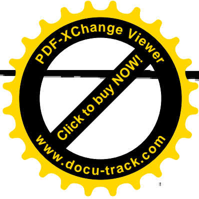
www.TopSage.com

它会允许人们通过修改“parser=Config”的值来重新设定，而不必实际接触代码。即使如果他们有一些Python知识，不必一次又一次地在命令行输入相同的选项也是非常不错的，这保持了工具的灵活性。

本章小结

标准库Optparse和ConfigParser模块很容易使用，并且已经流行了一段时间，因此它们在大多数你接触到的操作系统中都是可用的。如果你发现自己需要写一些命令行工具，那么自己进一步尝试optparse的高级特征也是值得的，例如，使用回调函数，对optparse本身进行扩展。你或许会对查看一些没有出现在标准库中的相关模块感兴趣，例如：CommandLineApp（*http://www.doughellmann.com/projects/CommandLineApp/*），Argparse（*http://pypi.python.org/pypi/argparse*），以及ConfigObj（*http://pypi.python.org/pypi/ConfigObj*）。



PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

第14章

实例

使用Python管理DNS

相比Apache配置文件，管理DNS服务器是一个非常简单的任务。困扰数据中心与web主机提供者的实际的问题是，如何实现可编程的大规模DNS修改。Python在这方面做了非常好的工作，有一个称为dnspython的模块。值得注意的是，还有另一个称为PyDNS的DNS模块，但是我们将重点介绍dnspython。

可以参考官方文档：*http://www.dnspython.org/*。此外，还有一个使用dnspython的更详细的说明：*http://vallista.idyll.org/~grig/articles/*。

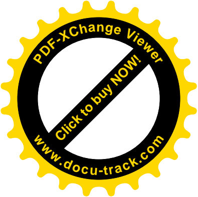
为了开始使用dnspython，需要通过easy_install进行安装，因为该包在Python包索引中已被列出。

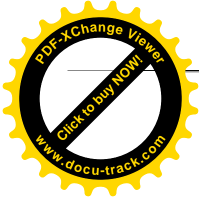
```
ngift@Macintosh-8][H:10048][J:0]# sudo easy_install dnspython
Password:
Searching for dnspython
Reading http://pypi.python.org/simple/dnspython/
[output supressed]
```

接下来，我们介绍IPython的模块，就像介绍这本书中的许多其他事情一样。在这个示例中，我们获得*oreilly.com*的A和MX记录：

```
In [1]: import dns.resolver
In [2]: ip = dns.resolver.query("oreilly.com","A")
In [3]: mail = dns.resolver.query("oreilly.com","MX")
In [4]: for i,p in ip,mail:
   ....:     print i,p
   ....:
   ....:
208.201.239.37 208.201.239.36
20 smtp1.oreilly.com. 20 smtp2.oreilly.com.
```


www.TopSage.com





在例14-1中，我们指定了A记录到ip，并且MX记录到mail。A在上部，MX记录在下部。现在我们有了一些它是如何工作的初步感受，让我们写一个脚本，收集所有主机的A记录。

例14-1：查询一组主机

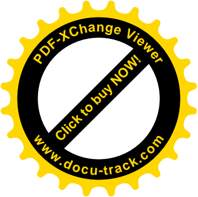
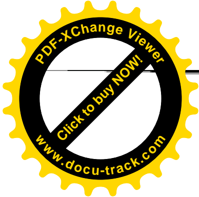
```
import dns.resolver

hosts = ["oreilly.com", "yahoo.com", "google.com", "microsoft.com", "cnn.com"]

def query(host_list=hosts):
    collection = []
    for host in host_list:
        ip = dns.resolver.query(host,"A")
        for i in ip:
            collection.append(str(i))
    return collection

if __name__ == "__main__":
    for arec in query():
        print arec
```

如果运行这个脚本，获得这些主机的所有的A记录，看起来类似下面这样：

```
[ngift@Macintosh-8][H:10046][J:0]# python query_dns.py
208.201.239.37
208.201.239.36
216.109.112.135
66.94.234.13
64.233.167.99
64.233.187.99
72.14.207.99
207.46.197.32
207.46.232.182
64.236.29.120
64.236.16.20
64.236.16.52
64.236.24.12
```

这里解决的一个明显的问题是程序化检测，检测是否所有的主机在文件中都有正确的A记录。

dnspython还可以做其他一些事情：可以管理DNS分区并执行更复杂的查询，而不是我们在这里描述的。如果你对查看一些更复杂的示例有兴趣，请参阅之前提到的URL地址。

使用OpenLDAP、Active Directory以及其他Python工具实现LDAP

LDAP在绝大多数公司都是一个强意词，其作者之一甚至运行LDAP数据库来管理他的家庭网络。或许你不熟悉LDAP，它代表Lightweight Directory Access Protocol。我们听到过的最好的对LDAP的定义来自Wikipedia："LDAP是一个运行在TCP/IP基础上的用于查询和修改目录服务的应用协议"。该服务的示例是认证，这是目前使用该协议的最为流行的应用。支持LDAP协议的目录示例是OpenDirectory、OpenLDAP、Red Hat Directory Server和Active Directory。Python-ldap API支持与OpenLDAP和Active Directory之间的通信。

有一个使用LDAP的Python API，称为python-ldap，它包括在Python API中，该API支持使用OpenLDAP2.x面向对象封装。也有支持其他LDAP相关的项，包括处理LDIF文件和LDAPv3。开始的时候，需要从python-ldap源码项目中下载包：*http://python-ldap.sourceforge.net/download.shtml*。

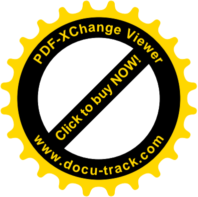
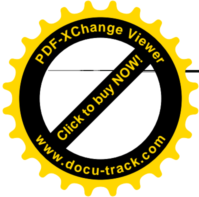
在安装python-ldap之后，你会希望首先尝试一下IPython中的库。以下是一个交互式会话的过程，其中成功实现了对公共ldap服务的绑定，以及一个不成功的绑定。对配置LDAP进行详细介绍超出了本书的范畴，但是可以使用密歇根大学的公共LDAP服务器测试python-ldap API。

```
In [1]: import ldap

In [2]: l = ldap.open("ldap.itd.umich.edu")

In [3]: l.simple_bind()
Out[3]: 1
```

这个简单的绑定告诉我们已经成功了，让我们查看一下失败的情况，并查看输出的结果：

```
In [5]: try:
   ....:     l = ldap.open("127.0.0.1")
   ....: except Exception,err:
   ....:     print err
   ....:
   ....:

In [6]: l.simple_bind()
---------------------------------------------------------------------------
SERVER_DOWN                               Traceback (most recent call last)

/root/<ipython console>
```


www.TopSage.com

```
/usr/lib/python2.4/site-packages/ldap/ldapobject.py in simple_bind(self, who, cred,
  serverctrls, clientctrls)
    167        simple_bind([who='' [,cred='']]) -> int
    168        """
--> 169        return self._ldap_call(self._l.simple_bind,who,cred,EncodeControlTuples
  (serverctrls),EncodeControlTuples(clientctrls))
    170
    171      def simple_bind_s(self,who='',cred='',serverctrls=None,clientctrls=None):

/usr/lib/python2.4/site-packages/ldap/ldapobject.py in _ldap_call(self, func, *args,
  **kwargs)
     92        try:
     93          try:
---> 94            result = func(*args,**kwargs)
     95          finally:
     96          self._ldap_object_lock.release()

SERVER_DOWN: {'desc': "Can't contact LDAP server"}
```

正如我们所看到的，在这个示例中，有一个LDAP服务器在运行，并且我们的代码运行良好。

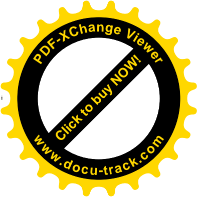
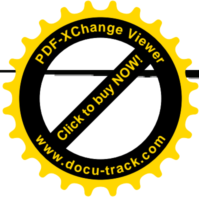
加载LDIF文件

创建一个到公共LDAP目录的简单连接对于帮助你完成工作不是非常有用的。以下是一个示例，实现了一个异步LDIF加载：

```
import ldap
import ldap.modlist as modlist

ldif = "somefile.ldif"
def create():
    l = ldap.initialize("ldaps://localhost:636/")
    l.simple_bind_s("cn=manager,dc=example,dc=com","secret")
    dn="cn=root,dc=example,dc=com"
    rec = {}
    rec['objectclass'] = ['top','organizationalRole','simpleSecurityObject']
    rec['cn'] = 'root'
    rec['userPassword'] = 'SecretHash'
    rec['description'] = 'User object for replication using slurpd'
    ldif = modlist.addModlist(attrs)
    l.add_s(dn,ldif)
    l.unbind_s()
```

查看这个示例，首先初始化一个本地LDAP服务器，然后创建一个对象类，当我们大规模异步加载LDIF文件时，该类会映射到LDAP数据库。值得注意的是，`l.add_s`展示了我们正创建一个异步API调用。

至此，你已经有了一些使用Python和LDAP的基本知识，但是你应该参考在本章开始部

分给出的资源，以获得使用python-ldap的更多帮助。特别地，有一些示例详细介绍了LDAPv3，包括创建、读取、更新、删除（CRUD）等。

需要说明的最后一件事是有一个不错的名为web2ldap的工具，它是一个Python， web-based的LDAP前端，由python-ldap的作者创建。你或许考虑尝试一下一些针对LDAP的其他web-based管理解决方案。官方文档可以参考：*http://www.web2ldap.de/*。LDAPv3支持很好的结构化。

Apache日志报告

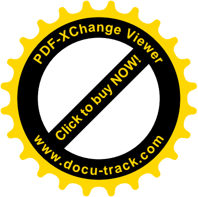
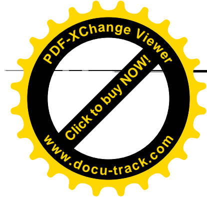
当前，网上大约50%的域采用的web服务器是Apache。接下来的示例专门展示了一个用于报告Apache日志文件的方法。这个示例仅关注Apache日志可用信息的一个方面，但是你应该可以将这个方法，应用到任何一类包括这些日志的数据上。这个方法也可扩展以适应大数据文件，或是大量文件。

在第3章，我们给出一些示例，解析Apche web服务器日志并提取信息。在这个示例中，我们重用在第3章编写的模块，以展示如何从一个或多个日志文件中产生可读报告。除了分别处理所有分别指定的日志文件，还可以让这个脚本来整合多个日志，产生单一报告。例14-2显示了脚本的代码。

例14-2：合并Apache日志文件报告

```
#!/usr/bin/env python

from optparse import OptionParser

def open_files(files):
    for f in files:
        yield (f, open(f))

def combine_lines(files):
    for f, f_obj in files:
        for line in f_obj:
            yield line

def obfuscate_ipaddr(addr):
    return ".".join(str((int(n) / 10) * 10) for n in addr.split('.'))

if __name__ == '__main__':
    parser = OptionParser()
    parser.add_option("-c", "--consolidate", dest="consolidate", default=False,
        action='store_true', help="consolidate log files")
    parser.add_option("-r", "--regex", dest="regex", default=False,
        action='store_true', help="use regex parser")

    (options, args) = parser.parse_args()
    logfiles = args
```

www.TopSage.com

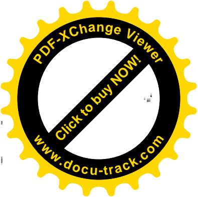
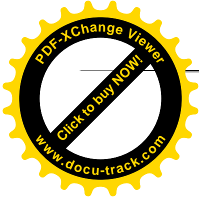
```
    if options.regex:
        from apache_log_parser_regex import generate_log_report
    else:
        from apache_log_parser_split import generate_log_report

    opened_files = open_files(logfiles)

    if options.consolidate:
        opened_files = (('CONSOLIDATED', combine_lines(opened_files)),)

    for filename, file_obj in opened_files:
        print "*" * 60
        print filename
        print "-" * 60
        print "%-20s%s" % ("IP ADDRESS", "BYTES TRANSFERRED")
        print "-" * 60
        report_dict = generate_log_report(file_obj)
        for ip_addr, bytes in report_dict.items():
            print "%-20s%s" % (obfuscate_ipaddr(ip_addr), sum(bytes))
        print "=" * 60
```

在脚本的上部，定义了两个函数：open_file()和combine_lines()。在脚本的后部，这两个函数允许我们之后使用一些适当的generator来进一步简化代码。open_files()是一个产生器函数，用于获得文件名列表（实质上，可以是任何具有重复特征的数据结构）。对于每一个文件名，它产生一个文件名三元组和一个相应的打开的文件对象。combine_lines()将一个可迭代的打开的文件对象作为参数。使用for循环，在文件对象上多次迭代。对于这些文件中的每一个，它迭代文件中的每一行，并且输出迭代的每一行。我们从combine_lines()获得的可迭代性是与文件对象的使用相对应的：在文件中的行上进行迭代。

接下来，我们使用optparse来解析来自用户的命令行参数。我们仅接受两个参数，且都是Boolean：整合的日志文件和使用正则表达式库。consolidate选项告诉脚本将所有文件作为一个文件看待。在某种意义上，如果这个选项被传递，我们将这些文件连接到一起。我们将在后面介绍这个内容。regex选项告诉脚本使用我们在第3章写的正则表达式库而不是“split”库。这两者都提供了相同的功能，但是“split”库更快一些。

然后检测regex标志是否被传递进来。如果是这样，使用regex模块。如果没有，使用split模块。我们实际上包括这个标志并加载条件来比较这两个库的性能。但是，我们将在以后运行并测试该脚本的性能。

接下来，我们在由用户传递的文件名列表上调用open_files()。正如我们已经谈到的，open_files()是一个产生器函数，从我们传递给它的文件名列表中产生文件对象。这表示它实际上不会打开文件，直到产生它。现在我们已经有一个可迭代的打开的文件对象，我们可以利用它做一些事情。我们可以或者迭代所有产生的文件并报告每一个文件，或者合并日志文件并作为一个文件进行报告。这也正是combine_lines()函数的由

来。如果用户传递“consolidate”标志，被迭代的文件实际上是一个单独的文件类对象：所有文件中所有行的产生器。

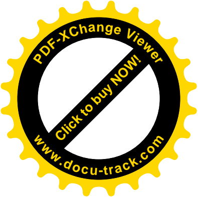
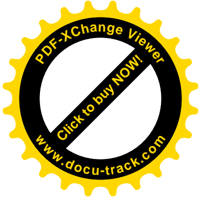
因此，无论是一个实际的文件或是一个合并的文件，我们都会传递给适当的generate_log_report()函数，其返回一个IP地址和发送给该IP地址的字节数。对于每一个文件，我们输出一些分隔符字符串以及格式化的字符串，包含generate_log_report()的结果。在一个单独的28KB日志上的输出结果如下所示：

```
************************************************************
access.log
------------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0           17479
200.80.230.0          45346
200.40.90.110         8276
130.150.250.0         0
70.0.10.140           2115
70.180.0.220          76992
200.40.90.110         23860
190.20.250.190        499
190.20.250.210        431
60.210.40.20          27681
60.240.70.180         20976
70.0.20.120           1265
190.20.250.210        4268
190.50.200.210        4268
60.100.200.230        0
70.0.20.190           378
190.20.250.250        5936
============================================================
```

这三个日志文件（实际上，是三个相同的日志文件，具有相同的多次复制的日志数据）的输出结果如下所示：

```
************************************************************
access.log
------------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0           17479
200.80.230.0          45346
<snip>
70.0.20.190           378
190.20.250.250        5936
============================================================
************************************************************
access_big.log
------------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0           1747900
```

www.TopSage.com

```
200.80.230.0          4534600
<snip>
70.0.20.190           37800
190.20.250.250        593600
============================================================
************************************************************
access_bigger.log
------------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0           699160000
200.80.230.0          1813840000
<snip>
70.0.20.190           15120000
190.20.250.250        237440000
============================================================
```

三个文件合并后的输出结果如下所示：

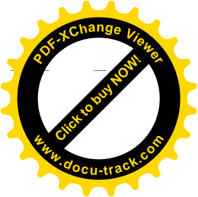
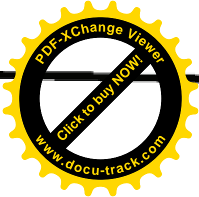
```
************************************************************
CONSOLIDATED
------------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0           700925379
200.80.230.0          1818419946
<snip>
190.20.250.250        238039536
============================================================
```

那么该脚本是如何执行的？内存消耗会是怎样的？在这一节中的所有的测评都是运行在Ubuntu Gutsy服务器上，使用AMD Athlon 64 X2 5400+ 2.8 GHz, 2 GB内存，一个希捷7200RPM磁盘驱动器。我们使用了大约1GB大小的文件：

```
jmjones@ezr:/data/logs$ ls -l access*log
-rw-r--r-- 1 jmjones jmjones 1157080000 2008-04-18 12:46 access_bigger.log
```

以下是运行时间。

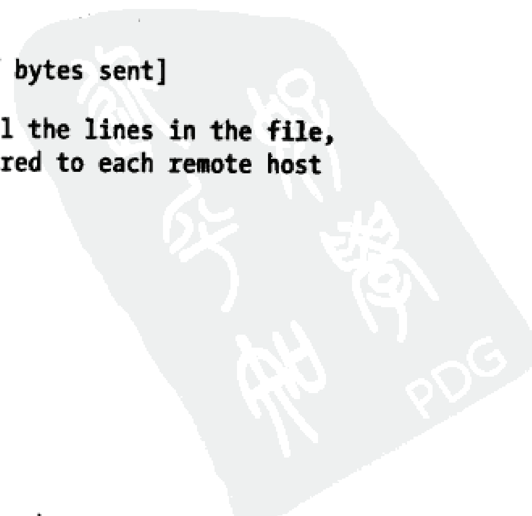
```
$ time python summarize_logfiles.py --regex access_bigger.log
************************************************************
access_bigger.log
------------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0           699160000
<snip>
190.20.250.250        237440000
============================================================

real    0m46.296s
user    0m45.547s
sys     0m0.744s
```

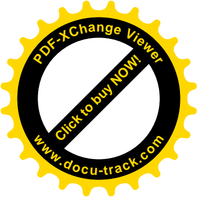
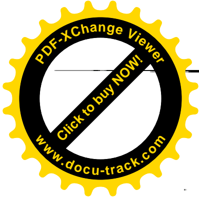
```
jmjones@ezr:/data/logs$ time python summarize_logfiles.py access_bigger.log
************************************************************
access_bigger.log
------------------------------------------------------------
IP ADDRESS           BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0          699160000
<snip>
190.20.250.250       237440000
============================================================

real    0m34.261s
user    0m33.354s
sys     0m0.896s
```

对于数据提取库的正则表达式版本，花了大约46秒。对于使用`string.split()`的版本，花了大约34秒。但是内存使用非常大，用了大约130MB内存。其原因是`generate_log_report()`在日志文件中保存了为每一个IP地址传输的字节列表。由此，更大的文件也就会消耗更多的内存。但是我们可以做一些处理。以下是解析库的一个占用内存较少的版本：

```
#!/usr/bin/env python

def dictify_logline(line):
    '''return a dictionary of the pertinent pieces of an apache combined log file

    Currently, the only fields we are interested in are remote host and bytes sent,
    but we are putting status in there just for good measure.
    '''
    split_line = line.split()
    return {'remote_host': split_line[0],
            'status': split_line[8],
            'bytes_sent': split_line[9],
}

def generate_log_report(logfile):
    '''return a dictionary of format remote_host=>[list of bytes sent]

    This function takes a file object, iterates through all the lines in the file,
    and generates a report of the number of bytes transferred to each remote host
    for each hit on the webserver.
    '''
    report_dict = {}
    for line in logfile:
        line_dict = dictify_logline(line)
        host = line_dict['remote_host']
        #print line_dict
        try:
            bytes_sent = int(line_dict['bytes_sent'])
        except ValueError:
            ##totally disregard anything we don't understand
            continue
        report_dict[host] = report_dict.setdefault(host, 0) + bytes_sent
    return report_dict
```

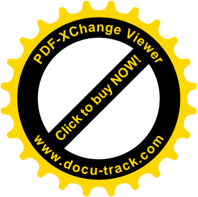
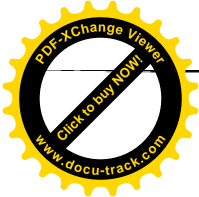


www.TopSage.com

当其运行的时候，大体上符合bytes_sent，而不是创建的调用函数符合。以下是略微修改的summarize_logfiles脚本，添加了新的选项，该选项针对加载占用内存较少版本的库：

```
#!/usr/bin/env python

from optparse import OptionParser

def open_files(files):
    for f in files:
        yield (f, open(f))

def combine_lines(files):
    for f, f_obj in files:
        for line in f_obj:
            yield line

def obfuscate_ipaddr(addr):
    return ".".join(str((int(n) / 10) * 10) for n in addr.split('.'))

if __name__ == '__main__':
    parser = OptionParser()
    parser.add_option("-c", "--consolidate", dest="consolidate", default=False,
        action='store_true', help="consolidate log files")
    parser.add_option("-r", "--regex", dest="regex", default=False,
        action='store_true', help="use regex parser")
    parser.add_option("-m", "--mem", dest="mem", default=False,
        action='store_true', help="use mem parser")

    (options, args) = parser.parse_args()
    logfiles = args

    if options.regex:
        from apache_log_parser_regex import generate_log_report
    elif options.mem:
        from apache_log_parser_split_mem import generate_log_report
    else:
        from apache_log_parser_split import generate_log_report

    opened_files = open_files(logfiles)

    if options.consolidate:
        opened_files = (('CONSOLIDATED', combine_lines(opened_files)),)

    for filename, file_obj in opened_files:
        print "*" * 60
        print filename
        print "-" * 60
        print "%-20s%s" % ("IP ADDRESS", "BYTES TRANSFERRED")
        print "-" * 60
        report_dict = generate_log_report(file_obj)
        for ip_addr, bytes in report_dict.items():
            if options.mem:
                print "%-20s%s" % (obfuscate_ipaddr(ip_addr), bytes)
            else:
                print "%-20s%s" % (obfuscate_ipaddr(ip_addr), sum(bytes))
        print "=" * 60
```

并且，这实际上比内存不足版本更快：

```
jmjones@ezr:/data/logs$ time ./summarize_logfiles_mem.py --mem access_bigger.log
************************************************************
access_bigger.log
------------------------------------------------------------
IP ADDRESS              BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0             699160000
<snip>
190.20.250.250          237440000
============================================================

real    0m30.508s
user    0m29.866s
sys     0m0.636s
```

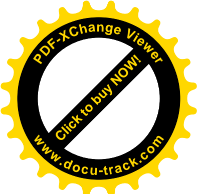
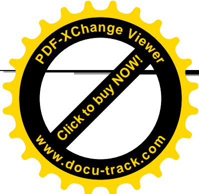
对于每个运行周期，内存的消耗稳定在大约4MB。这个脚本将每分钟处理大约2GB的日志文件。理论上讲，文件大小可以是不确定的，内存不会增长，就像它在之前版本中那样。然而，由于这里使用了一个字典，并且每个关键字是独一无二的IP地址，内存的使用会随着独立IP地址而增长。如果内存消耗成为问题，你可以将字典与一个持久数据库（或是关系数据库，甚至是Berkeley DB）进行交换。

FTP镜像

接下来的示例演示了如何连接到一个FTP服务器，递归获得用户指定目录中的所有在那个服务器上的文件。在获得文件之后，也允许进行删除。你或许奇怪“这个脚本的关键点在哪里？rsync不能处理所有这些工作么？”回答是肯定的“是的，它不能”。如果rsync不能安装到你正使用的服务器上并且你不允许安装它，将怎么办？（对于系统管理员这似乎是不太可能的，但是它碰巧发生了）。或者如果你没有使用SSH或rsync对你希望下载数据的服务器进行访问的权限，怎么办？这里有一个办法。以下是mirror脚本的源码：

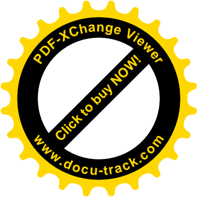
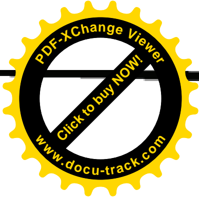
```
#!/usr/bin/env python

import ftplib
import os

class FTPSync(object):
    def __init__(self, host, username, password, ftp_base_dir,
                                        local_base_dir, delete=False):
        self.host = host
        self.username = username
        self.password = password
        self.ftp_base_dir = ftp_base_dir
        self.local_base_dir = local_base_dir
        self.delete = delete
```


www.TopSage.com

```
        self.conn = ftplib.FTP(host, username, password)
        self.conn.cwd(ftp_base_dir)
        try:
            os.makedirs(local_base_dir)
        except OSError:
            pass
        os.chdir(local_base_dir)
    def get_dirs_files(self):
        dir_res = []
        self.conn.dir('.', dir_res.append)
        files = [f.split(None, 8)[-1] for f in dir_res if f.startswith('-')]
        dirs = [f.split(None, 8)[-1] for f in dir_res if f.startswith('d')]
        return (files, dirs)
    def walk(self, next_dir):
        print "Walking to", next_dir
        self.conn.cwd(next_dir)
        try:
            os.mkdir(next_dir)
        except OSError:
            pass
        os.chdir(next_dir)

        ftp_curr_dir = self.conn.pwd()
        local_curr_dir = os.getcwd()

        files, dirs = self.get_dirs_files()
        print "FILES:", files
        print "DIRS:", dirs
        for f in files:
            print next_dir, ':', f
            outf = open(f, 'wb')
            try:
                self.conn.retrbinary('RETR %s' % f, outf.write)
            finally:
                outf.close()
            if self.delete:
                print "Deleting", f
                self.conn.delete(f)
        for d in dirs:
            os.chdir(local_curr_dir)
            self.conn.cwd(ftp_curr_dir)
            self.walk(d)

    def run(self):
        self.walk('.')

if __name__ == '__main__':
    from optparse import OptionParser
    parser = OptionParser()
    parser.add_option("-o", "--host", dest="host",
        action='store', help="FTP host")
    parser.add_option("-u", "--username", dest="username",
        action='store', help="FTP username")
    parser.add_option("-p", "--password", dest="password",
        action='store', help="FTP password")
    parser.add_option("-r", "--remote_dir", dest="remote_dir",
```

www.TopSage.com

```
        action='store', help="FTP remote starting directory")
    parser.add_option("-l", "--local_dir", dest="local_dir",
        action='store', help="Local starting directory")
    parser.add_option("-d", "--delete", dest="delete", default=False,
        action='store_true', help="use regex parser")

        (options, args) = parser.parse_args()
        f = FTPSync(options.host, options.username, options.password,
            options.remote_dir, options.local_dir, options.delete)
    f.run()
```

这个脚本对于使用类进行编写略微有些容易。构造器获得一些参数。为了连接并登录，你必然传递给它host、username以及password。为了进入远端服务器的合适位置，以及你的本地服务器的合适位置，必须传递ftp_base_dir和local_base_dir。delete是一个标志，指定一旦你下载远端服务器上的文件完毕，是否进行删除 —— 在构造器中我们设置的默认值为False。

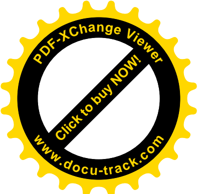
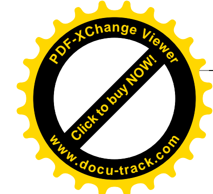
一旦用对象属性设置完这些值，我们连接到指定的FTP服务器并登录。然后，改变目录到指定的服务器起始目录，并且改变本地目录到本地主机的起始目录。在实际修改到本地启始目录之前，我们首先进行创建。如果该目录已经存在，我们将获得一个OSError异常，我们将忽略该异常。

有三个额外定义的方法：get_dirs_files()、walk()和run()。get_dirs_files()决定在当前目录中哪个是文件哪个是目录。（顺便说一句，这是Unix服务器上唯一需要做的工作）。它通过一个目录列表，查看列出的每一行的第一个字符，识别出哪个是文件，哪个是目录。如果字符是“d”，那么它就是目录。如果字符是“-”，那么它就是文件。这表示我们不用跟踪链接，也不用处理块设备。

下一个我们定义的方法是walk()。该方法是大量处理工作发生的地方。walk()方法只有单一的参数：下一个被访问的目录。在更进一步了解之前，需要指出这是一个递归函数。我们让它调用自己。如果任何目录包括其他其子目录，这些子目录都会被逐一进入。在walk()方法中的代码首先修改FTP服务器的目录到指定的目录。接下来改变到本地服务器的目录上（如果需要的话就进行创建）。然后，保存当前在FTP服务器的位置到变量ftp_curr_dir和本地位置到变量local_curr_dir中，以备将来使用。接下来，使用已经介绍过的get_dirs_files()方法下载这个目录中的文件和目录。对于目录中的每一个文件，通过使用retrbinary() FTP方法来获得。如果删除标志被传递进来，我们也删除文件。接下来，改变目录到FTP服务器的当前目录，调用walk()来进入到更低级的目录。我们再次改变目录到当前目录的原因是当更低层的walk()调用返回时，我们可以回到我们所在的位置。

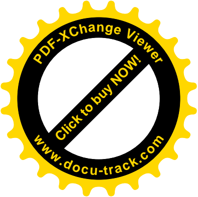
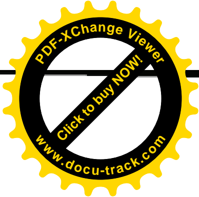
我们定义的最后一个方法是run()。run()是一个简单而便捷的方法。调用run()简单地调用walk()，并传递它到当前的FTP目录。

www.TopSage.com

在这个脚本中有一些非常基本的错误和异常处理。首先，不用检测所有的命令行参数，并且确信至少host、username和password被传入。如果没有特别指定，脚本将很快被执行。如果异常发生，我们不用再次尝试下载文件。如果一些事情导致下载失败，我们将得到一个异常。在这种情况下，程序将终止。如果脚本在下载的中间终止，下次开始的时候，脚本将再次开始下载文件。不用删除它已经下载的文件部分。

www.TopSage.com

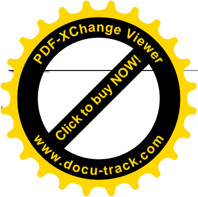
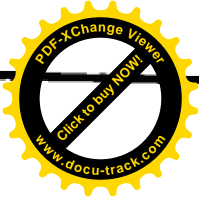
附录

回调

回调（callback）和传递函数的概念或许对你来说还很陌生。如果真是这样，它显然值得你进行深入研究，能够理解得足够透彻而使用它，或者最起码，当你看到它被使用时，知道它是怎么运行的。在Python中，函数是“第一个类”，这意味着你可以传递它们，将它们视为对象——因为它们事实上就是一类对象。参见例A-1。

例A-1：函数作为“第一个类”

```
In [1]: def foo():
...:        print foo
...:
...:

In [2]: foo
Out[2]: <function foo at 0x1233270>

In [3]: type(foo)
Out[3]: <type 'function'>

In [4]: dir(foo)
Out[4]:
['__call__',
 '__class__',
 '__delattr__',
 '__dict__',
 '__doc__',
 '__get__',
 '__getattribute__',
 '__hash__',
 '__init__',
 '__module__',
 '__name__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__str__',
```

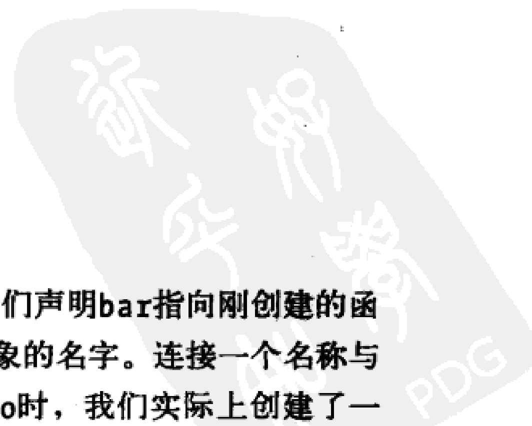



```
'func_closure',
'func_code',
'func_defaults',
'func_dict',
'func_doc',
'func_globals',
'func_name']
```

简单地对函数进行引用，例如在之前示例中的foo，并非对函数进行调用。引用函数的名字能够获得函数所具有的任何属性，甚至之后用不同的名称引用函数仍然如此。参见例A-2。

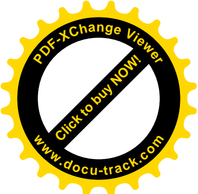
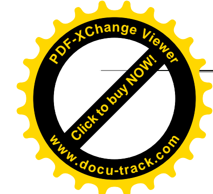
例A-2：通过函数名引用函数

```
In [1]: def foo():
   ...:     """this is a docstring"""
   ...:     print "IN FUNCTION FOO"
   ...:
   ...:

In [2]: foo
Out[2]: <function foo at 0x8319534>

In [3]: foo.__doc__
Out[3]: 'this is a docstring'

In [4]: bar = foo

In [5]: bar
Out[5]: <function foo at 0x8319534>

In [6]: bar.__doc__
Out[6]: 'this is a docstring'

In [7]: foo.a = 1

In [8]: bar.a
Out[8]: 1

In [9]: foo()
IN FUNCTION FOO

In [10]: bar()
IN FUNCTION FOO
```



我们创建了一个新函数foo，它包括一个docstring。之后，我们声明bar指向刚创建的函数foo。在Python中，你通常考虑可作为变量的往往是某个对象的名字。连接一个名称与一个对象的过程称为“名称绑定”。因此当我们创建函数foo时，我们实际上创建了一个函数对象，然后绑定foo到一个新的函数。使用IPython提示符来查看，我们可以了解的关于foo的基本信息，它向回报告这是一个foo函数。有趣的是，它说的是与名称bar相同的事情，也就是说它是一个foo函数。我们设置名为foo的函数的属性，并且可以通过bar来访问它。调用foo和bar将产生相同的结果。

www.TopSage.com

在本书中我们使用回调是在第5章的网络部分。在这一章的FTP示例中，传递函数允许运行时的动态机制，具有实现代码－时间可扩展性，并能够改进代码的重用性。即使你认为自己不会使用回调，它仍然是一个值得你记住的思维过程。

www.TopSage.com