
Python UNIX和Linux系统 管理指南

Noah Gift & Jeremy M. Jones 著
杨明华 谭励 等译



Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo
O'Reilly Media, Inc.授权机械工业出版社出版

机械工业出版社

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

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

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

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

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

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

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：

```
→ 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')
```

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 1686',
 '.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')
```

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。

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。

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 snmpininput
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 )
```

```
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
```

```

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",
```

```

        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
```

```
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generator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
```

关键是最末一行，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，

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》，那么最好多参考这些书。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.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。

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)
```

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？

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')
```

```
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):
    """Determines 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

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

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"
```

```
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 = []
    cmd = []
    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:
        cmd.append(cmd[1])
    return ips, cmd

ips, cmd = readConfig()

#For every ip address, run all commands
for ip in ips:
    for cmd in cmd:
        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：

```
[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 = []
    cmd = []
    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:
        cmd.append(cmd[1])
    return ips, cmd

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 cmd from config
ips, cmd = readConfig()

#Determine Number of threads to use, but max out at 25
if len(ips) < 25:
    num_threads = len(ips)
```

```
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。

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EPM（EPM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
```

```
[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)
```

```

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。

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。

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
```

```
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Applescript 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
```

```

#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：

- Libvirt，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

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Ubuntu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。

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、VMware Server和VMware 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()
```

创建新区域：

```
→ In [3]: domain = sdb.create_domain('my_domain')
```

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。

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>。

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
```

```

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（pumpkin-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，

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

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
```

```
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.
```

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*，或其他操作系统级的包管理系统。我们会在这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。

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`setuptools`、`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”

(`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`ez_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) 中对其进行有专门介绍。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

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/LZF05h8JEW4Jzr+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

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  
$
```

在当前工作目录下安装一个已解包的源码发布

尽管这看起来有些琐碎，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

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virtualenv，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

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。

```
▶▶▶ 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
```

```
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
....
```

```
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
```

```
    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
```

```

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，`rpm`s, `Solaris`, `pkgtool`, 和`HP-UX swinstall`。

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。

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
```

```
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buildout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，具有一个类似于下述配置文件，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

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`site-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

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

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():
```

```
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
```

```

%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：

```
→ $ 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 Python 2.4 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

Python 2.4 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
```

```
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。

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"
....:
```

```
....:  
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：

```
In [1]: 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
```

```
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

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：Subprocess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)

```

```
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
```

```
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][]: cat /etc/passwd | grep o: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 o: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

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：

```
▶ [program:daemon]
command=/root/daemon.py ; the program (relative uses PATH, can take args)
autorestart=true ; restart at unexpected quit (default: true)
```

现在，可以开始监管并且使用supervisorctl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中，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

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
```

```
#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'
```

```
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 06:45:47 2008 amazon.com RETURNED 1
Sat Apr 19 06:45:47 2008 WAITING FOR freebase.com
Sat Apr 19 06:45:47 2008 freebase.com RETURNED 0
Sat Apr 19 06:45:47 2008 WAITING FOR clearink.com
Sat Apr 19 06:45:47 2008 clearink.com RETURNED 0
Sat Apr 19 06:45:47 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

的事情。略微修改示例来创建一个脚本来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扫描

```
#!/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:
    queue.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][]:# 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 $N * (\text{平均响应时间}/\text{ping})$ 。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 $254 * (~3\text{秒})$ ，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.

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`Queue.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:
```

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

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

```
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
```

```
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:
```

```

        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"
```

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。

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。

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:
```

```

        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`fork`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。

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。

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。

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):
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.connect("destroy", self._on_destroy)
        self.window.set_title("Simple GTK Application")
        self.window.set_size_request(300, 200)

        self.button = gtk.Button("Click Me")
        self.button.connect("clicked", self._on_button_clicked)
        self.button.set_label("Current Time: %s" % time.strftime("%c"))

        self.window.add(self.button)
        self.window.show_all()

    def _on_button_clicked(self, button):
        button.set_label("Current Time: %s" % time.strftime("%c"))

    def _on_destroy(self, window):
        gtk.main_quit()

if __name__ == "__main__":
    app = SimpleButtonApp()
    gtk.main()
```

```

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，

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

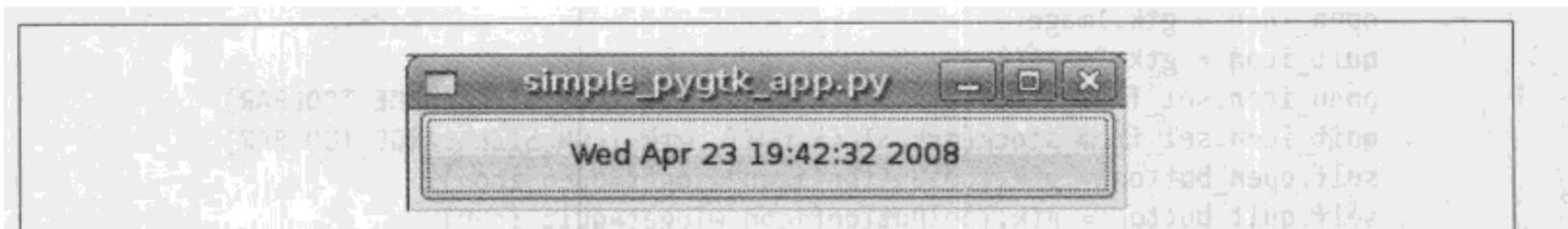


图11-2：简单的PyGTK应用 —— 在点击按钮之后

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_number, 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)
```

```

#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抛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。

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.curr_topline = self.curr_topline + (2 * curses.LINES)
        if self.curr_topline > len(self.loglines):
            self.curr_topline = len(self.loglines)
        self.draw_loglines()

    def draw_loglines(self):
        self.screen.clear()
        for i in range(self.curr_topline, min(self.curr_topline + 2 * curses.LINES, len(self.loglines))):
            logline = self.loglines[i]
            if logline['status'] == '200':
                color = 'green'
            else:
                color = 'red'
            self.screen.addstr(i, 0, logline['logline'], curses.color_pair(color))
        self.screen.refresh()
```

```
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。



圖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

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。

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`scripts`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 startapp 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
```

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。`views.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。

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

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 %}



{% for f in file_list %}
  <li><a href="/viewlog/linesort/{{ f }}/">{{ f }}</a></li>
{% endfor %}


{% 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

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。

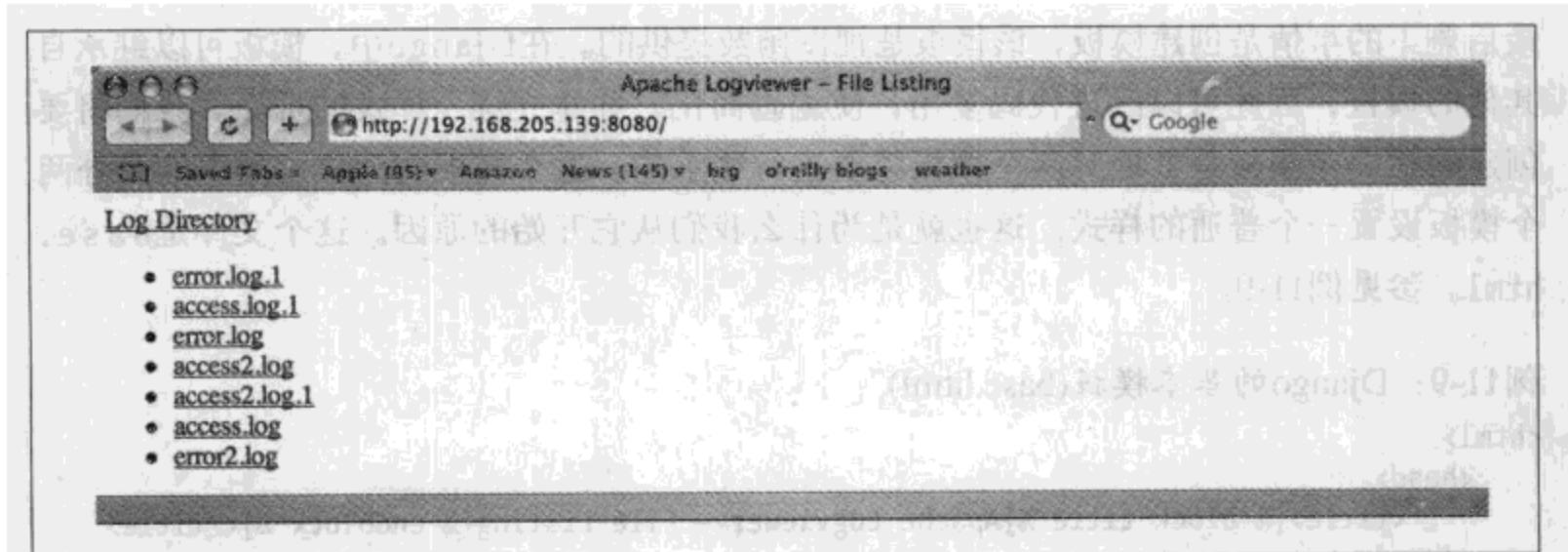


图11-4：Apache日志列表

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 %}


|                                                      |                                                                |                                                              |                                                      |
|------------------------------------------------------|----------------------------------------------------------------|--------------------------------------------------------------|------------------------------------------------------|
| <a href="/viewlog/status/{{ filename }}/">Status</a> | <a href="/viewlog/remote_host/{{ filename }}/">Remote Host</a> | <a href="/viewlog/bytes_sent/{{ filename }}/">Bytes Sent</a> | <a href="/viewlog/linesort/{{ filename }}/">Line</a> |
|                                                      |                                                                |                                                              |                                                      |
| {{ l.status }}                                       | {{ l.remote_host }}                                            | {{ l.bytes_sent }}                                           | <pre> {{ l.logline }}</pre>                          |
|                                                      |                                                                |                                                              |                                                      |


{% 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

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。

The screenshot shows a web-based log viewer titled "Apache Logviewer - File Viewer". The URL in the address bar is `http://192.168.205.139:8080/viewlog/linesort/access.log.1/`. The page displays a table of log entries with columns: Status, Remote Host, Bytes Sent, and Line. The "Line" column contains the full log line. The log entries show various HTTP requests from the same IP address (127.0.0.1) at different times, with status codes like 200, 404, and 304, and byte counts ranging from 0 to 300.

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/1.1" 200 2326
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

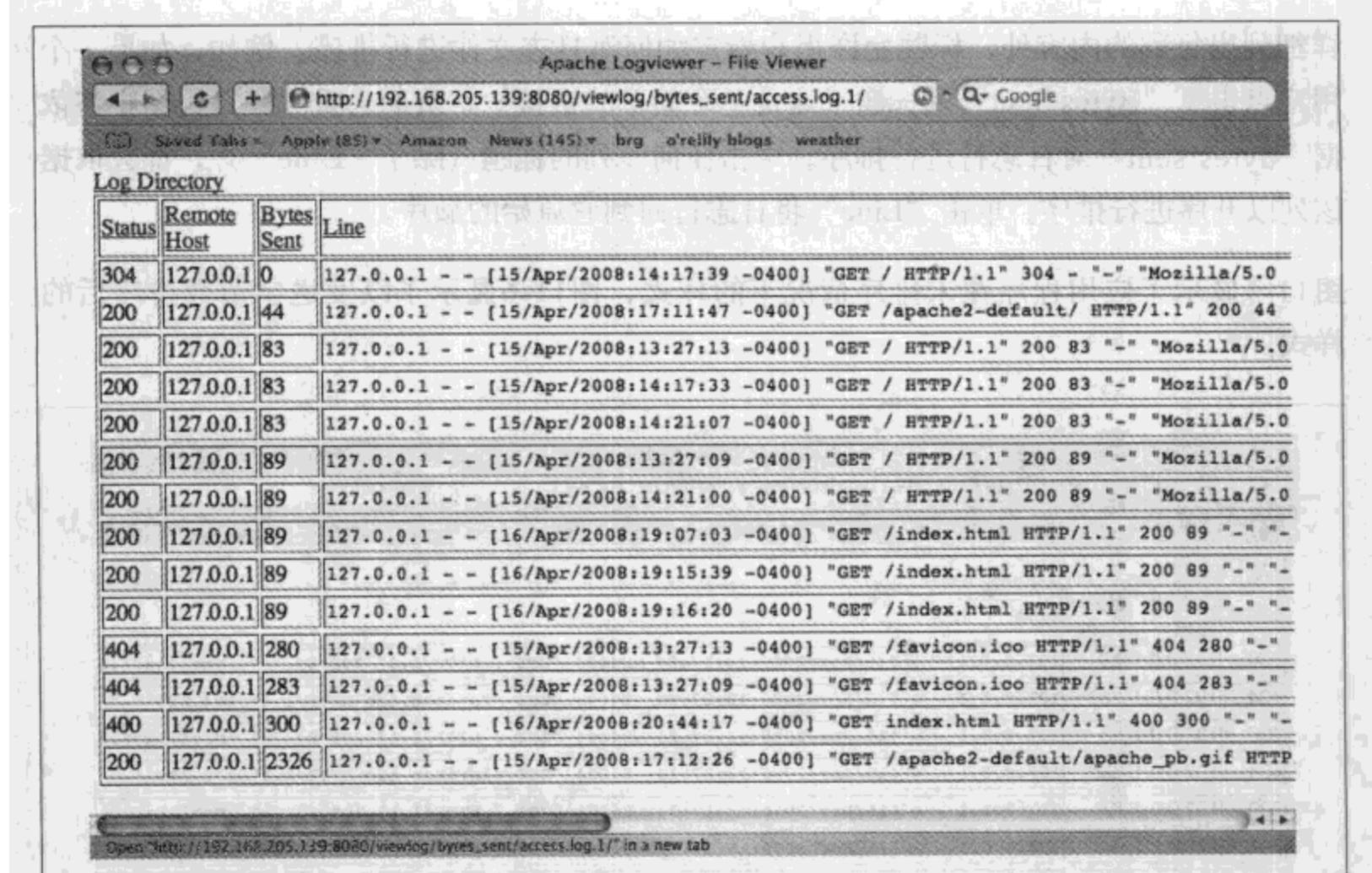


图11-6：Django Apache日志浏览器——以发送字节为序

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/
```

```
-- __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*settings.py*文件的绝对路径。调用*os.path.dirname(__file__)*可以获得*settings.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*settings.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*dango.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，*HardwareComponent*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。

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中心。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`OperatingSystem`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

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。

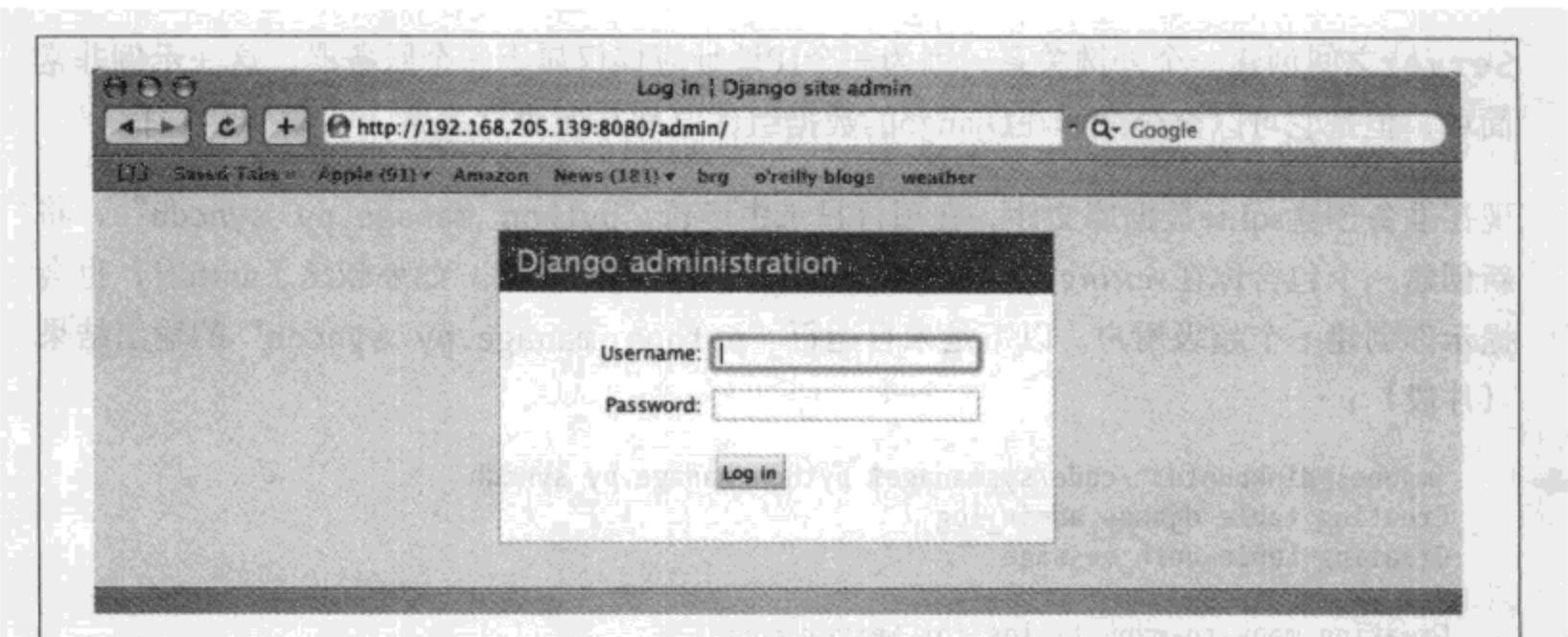


图11-7：Django系统管理登录

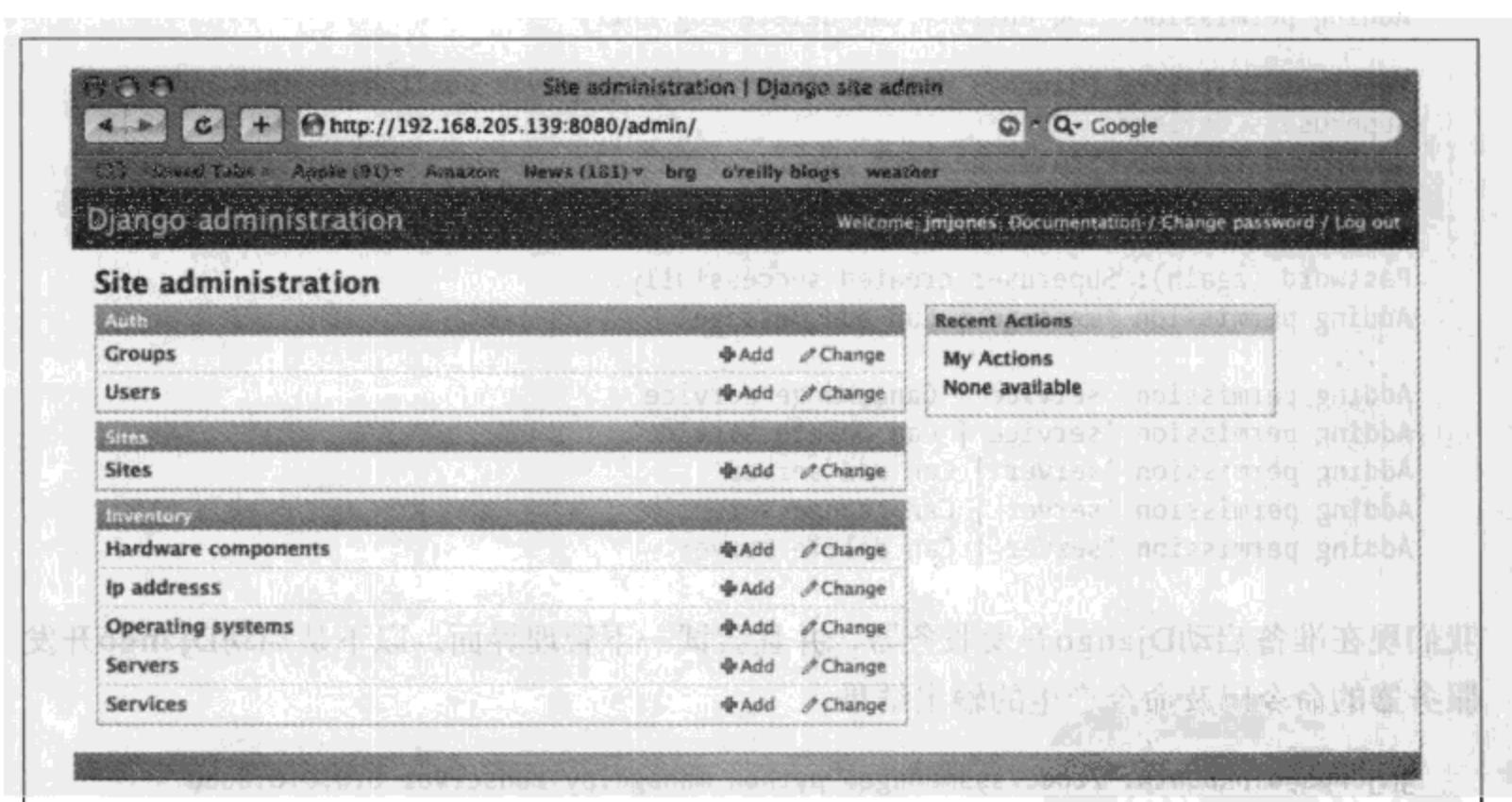


图11-8：Django管理主页面

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*。

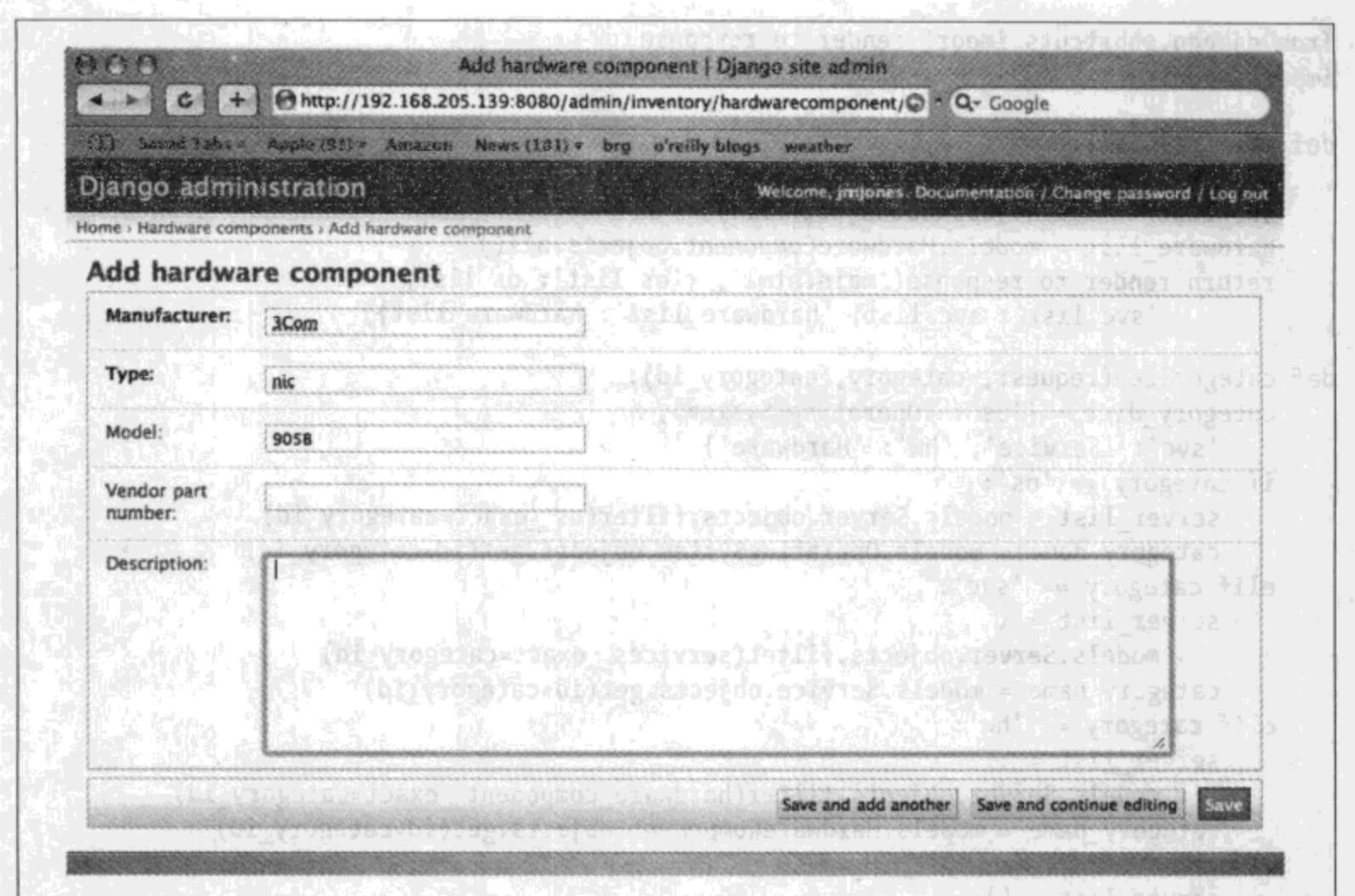


图11-9：Django系统管理添加硬件组件

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.
```

```

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>
```

```

<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 %}

```

“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“server_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 %}
```

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。

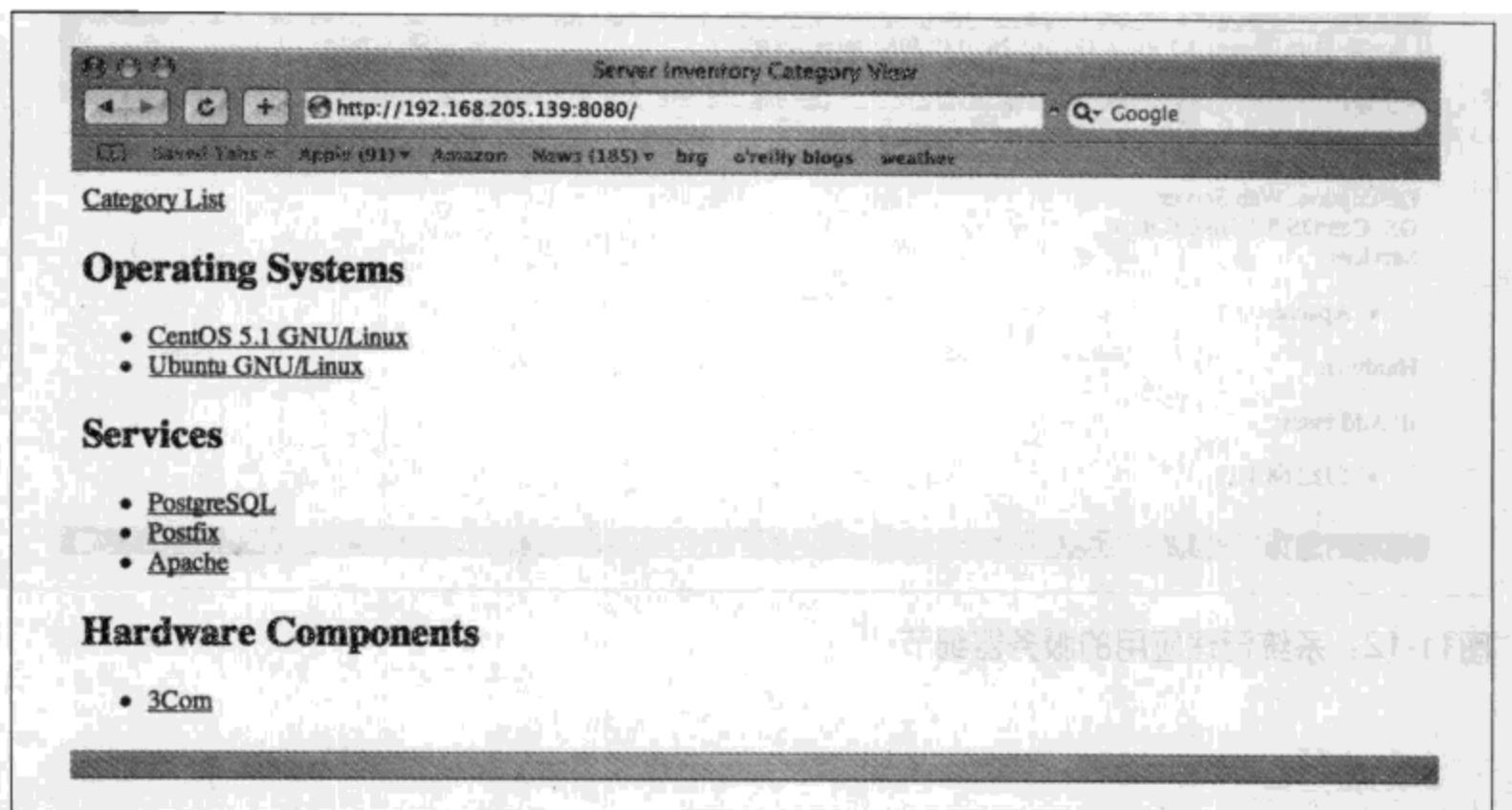


图11-10：系统管理应用主页面

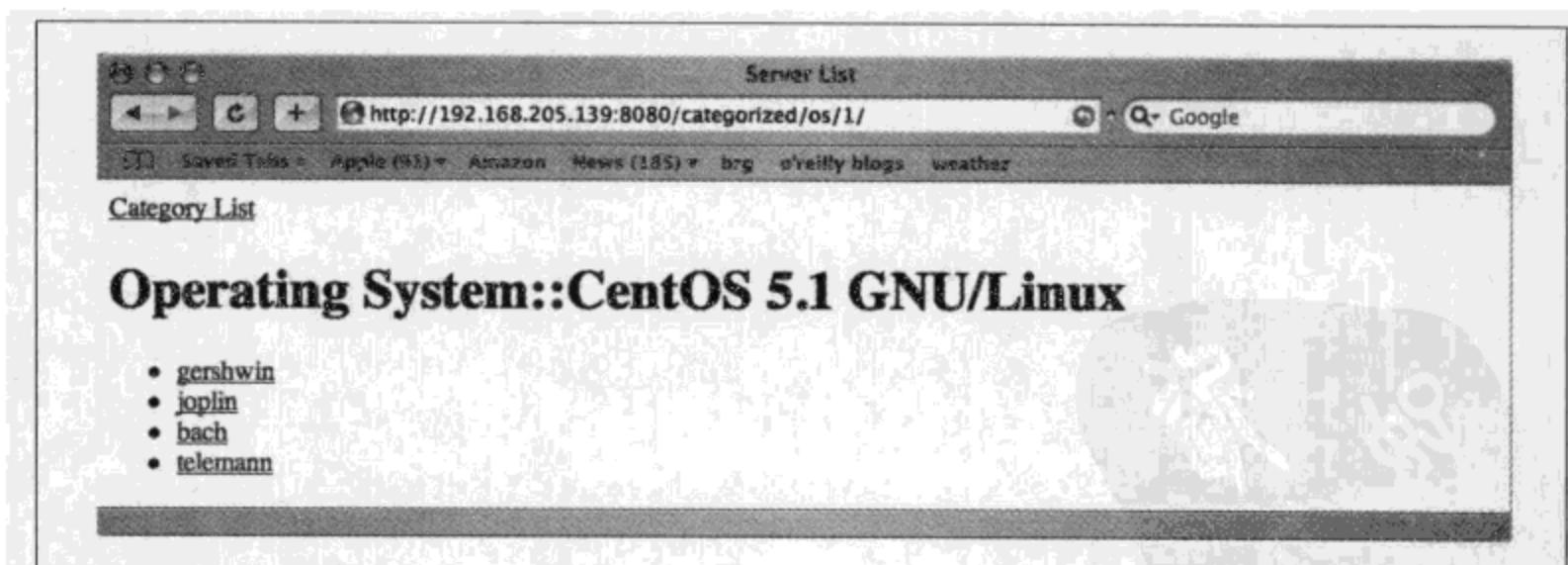


图11-11：系统管理应用的操作系统类型CentOS（分类）

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。

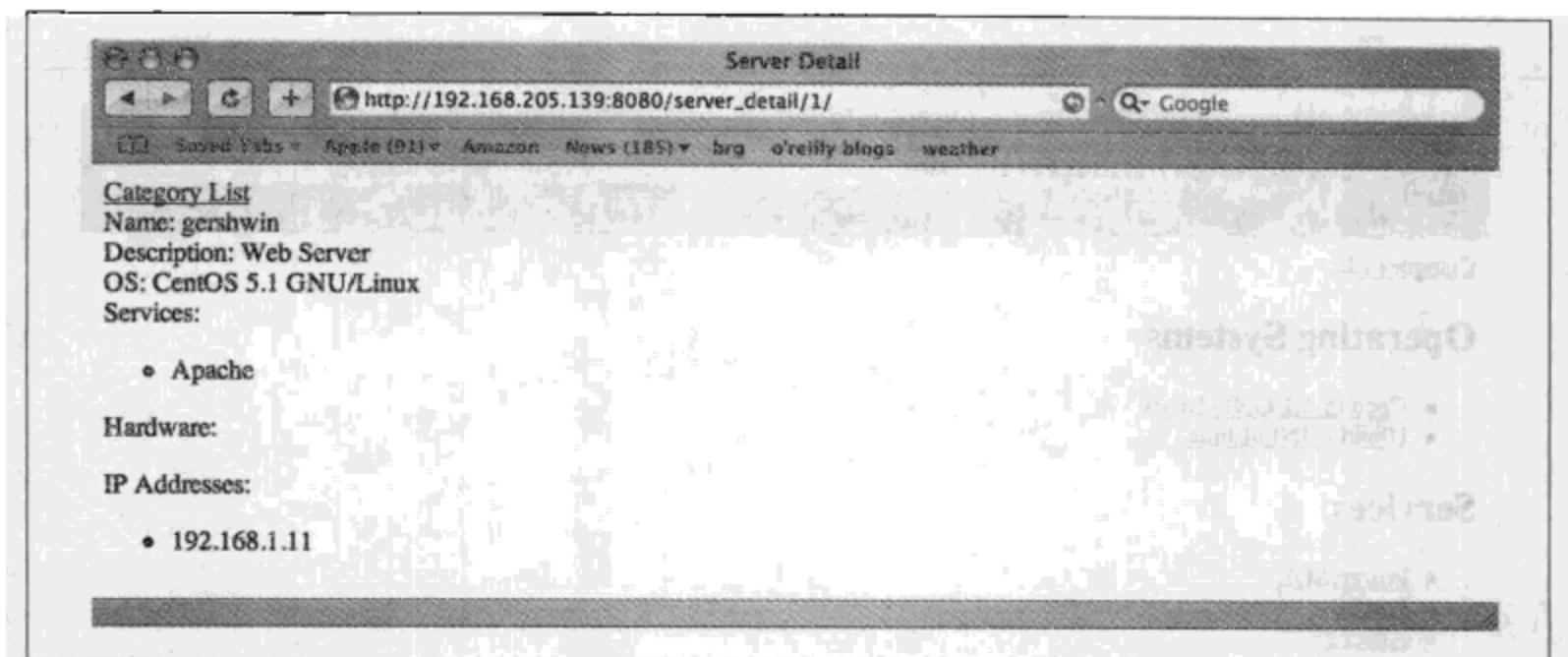


图11-12：系统管理应用的服务器细节

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  
S'S'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。

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
```

```
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

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.
```

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：

```
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。

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
```

```
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
```

```
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()
```

```
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__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
```

```
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增加的情况，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
```

```

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，必须调查哪一個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

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")
```

```
)  
;  
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]:
```

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IPython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_operatingsystem`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`OperatingSystem`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`OperatingSystem`对象并对`name`和`description`属性进行赋值（需要注意的是，为这些

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_database()`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`OperatingSystem`类到*inventory_operationsystem*表，Storm将查看所有在*inventory_operationsystem*表中的相关记录，从中创建`OperatingSystem`对象。对于每一个`OperatingSystem`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 kernel），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:///{}'.format(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`store.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`OperatingSystem`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

SQLAlchemy介绍相同的inventory_operatingsystem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`OperatingSystem`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）。

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SQLAlchemy，并将其作为“最终工具集”。SQLAlchemy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SQLAlchemy 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，但是这通常才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，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。

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计数，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。

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
```

```
▶ $ 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。以下是在当前目录（或“.”）正确地运行它时得到的输出结果。

```
→ $ 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
```

这其中有趣的地方是，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，`optparse`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",
                             version="0.1a")
```

```

        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

```

```
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

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'}
```

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：

```
[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Python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%
```

/	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。

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
```

```
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。

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"\n
                           % (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
```

```
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 suppressed 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。

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>)。

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 suppressed]
```

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.
```

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>
```

```

/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Apache 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
```

```

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
```

```
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

```

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="mem", 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
```

```
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",
```

```

    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。

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。



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。

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。即使你自己不会使用回调，它仍然是一个值得你记住的思维过程。

