PyMOTW 文档

版本 1.4

作者： Doug Hellmann[1]
翻译： PyMOTW中文翻译小组[2]

June 11, 2009

[1]Doug Hellmann<doug.hellmann@gmail.com>
[2]主页<http://code.google.com/p/pymotwcn>

CONTENTS

CHAPTER

ONE

PYMOTW: CONFIGPARSER

- 模块： ConfigParser
- 目的： 读取/写入配置文件,类似于Windows的INI文件
- python版本： 1.5+

1.1 描述

ConfigParser模块可以为你的应用程序创建用户可编辑的配置文件． 这个配置文件由一个个节组成,每个节可以包含配置数据的名字-值对.支持通过使用Python的格式化字符串进行值的插入， 以此来构建那些依赖于其他值的数据值(这对路径或URL来说是尤其方便的)．

在工作中,当我们把东西移动到svn和 trac 之前， 我们开发推出了自己的用于进行分布式代码复查的工具． 为了准备好需要复查的代码,一个开发者常常需要写完一个''approach''摘要文件,然后附上被修改后的代码(即和原代码有区别的地方).这个 approach文档支持通过Web页面添加注释， 因此开发者不在我们的主要办公室里也可以复查代码． 但是唯一的麻烦之处是， 发表代码的不同之处让人感到有点痛苦． 想要让部分处理过程变得简单些,我写了一个命令行工具， 运行他可以针对CVS沙盒,自动的寻找出并发表代码的不同之处．

为了能够让这个工具即时更新approach文档中的区别， 需要知道怎样到达存放approach文档的网络服务器． 由于我们开发者不总是在办公室， 从任意给定主机到达服务器的URL可能是通过SSH端口转发过来的． 为了不强迫每个开发者都使用同样的端口转发协议?这个工具应使用一个简单的配置文件来记住这个URL．

一个开发者的配置文件可能会是这个样子:

```
[portal]
url = http://%(host)s:%(port)s/Portal
username = dhellmann
host = localhost
password = SECRET
port = 8080
```

portal小节表示approach文件的网址． 一旦代码的区别被发送到这个网址， 我们的工具应该下载这个配置文件,通过ConfigParser模块来访问URL.这可能看起来像这样:

```
from ConfigParser import ConfigParser
import os

filename = os.path.join(os.environ['HOME'], '.approachrc')

config = ConfigParser()
config.read([filename])

url = config.get('portal', 'url')
```

上述的例子中, 变量url的值为''http://localhost:8080/Portal``.配置文件中的变量url中包含两个格式化字符串:''%(host)s'' 和 ``%(port)s''. 通过 get() 方法, 自动地将变量host和port的值替换到格式化字符串中.

当然, 这是基于Python2.1的旧代码. 在近来的版本中,ConfigParser模块已经被改进了很多.SafeConfigParser类是a drop?用来取代ConfigParser, 以改善插值处理.

对于这个工具,我只需要字符串选项. ConfigParser支持其他的选项类型:整型, 浮点型和布尔型.由于可选文件格式不提供直接使用一个值来关联一种类型的方式,所以调用者需要知道何时需要使用一种不同的函数来查询那些其他类型的选项.例如,为了查找一个布尔选项, 使用 getboolean() 函数而不是 get() 函数.函数的参数是一样的,但是选项的值在返回之前被转换为一个布尔类型.类似地,还有独立的 getint() 和 getfloat() 函数.

ConfigParser类也支持增加和删除小节到指定文件并保存结果.这使得创建一个用于编辑程序的配置的用户界面,或是利用配置文件格式存放简单数据文件成为可能. 例如, 一个应用需要存储小量的类似于数据库格式的数据, 可以利用ConfigParser类,这样一来生成的文件也是人类可读的.

CHAPTER

TWO

PYMOTW: QUEUE

- 模块：Queue
- 目的：提供一个线程安全的FIFO功能。
- python版本：1.4+

2.1 描述

Queue提供了FIFO功能，一般常用于多线程编程，它可以在生产者和消费者线程中安全的传递消息或者其他数据。调用者会自动创建锁，当使用Queue对象，你可以根据需求创建多个线程。一个Queue的大小(元素的个数)受可用内存的限制。

本文假设你已经了解基本的Queue特点，如果你还不清楚，可以阅读参考后继续后面内容：

- Queue data structures
- FIFO

2.2 示例

举例说明如何在多线程中使用Queue对象，我们创建一个简单的 podcasting 客户端，这个客户端读取一个或者多个RSS feeds，依次将需下载的内容置于队列中，然后采用多线程模式同时处理多个下载。这比较简单，也许没有多大实用价值，但这个框架代码很好的说明了如何来利用Queue模块。

开始，我们加载一些有用的模块：

```
from Queue import Queue

from threading import Thread
import time

import feedparser
```

首先，需要创建一些运行参数，通常这些来自用户输入(可以任何东西，比如参数，数据库)，在我们的例子中，我们硬编码几个值。

```
# Set up some global variables
num_fetch_threads = 2
enclosure_queue = Queue()

# A real app wouldn't use hard-coded data...
feed_urls = [ 'http://www.castsampler.com/cast/feed/rss/guest',]
```

接下来，我们需要在工作线程中定义相应函数来处理下载。再次，这里为了便于说明模拟下载，实际下载可以参考 urllib 模块(这再以后会介绍)。在这个示例中，我们只根据线程id，使其sleep一段时间。

```
def downloadEnclosures(i, q):
  """This is the worker thread function.
  It processes items in the queue one after another.
  These daemon threads go into an infinite loop,
  and only exit when the main thread ends.
  """
  while True:
    print '%s: Looking for the next enclosure' % i
    url = q.get()
    print '%s: Downloading:' % i, url
    time.sleep(i + 2) # instead of really downloading the URL, we just pretend

    q.task_done()
```

一旦定义好目标函数，我们就可以启动工作线程。注意，函数downloadEnclosures()在“url = q.get()”会阻塞，直到队列有东西返回，因此，当队列中有东西时，启动线程总是安全的。

```
# Set up some threads to fetch the enclosures
for i in range(num_fetch_threads):
  worker = Thread(target=downloadEnclosures, args=(i, enclosure_queue,))
  worker.setDaemon(True)
  worker.start()
```

现在，我们开始检索feed的内容（使用Mark Pilgrim的 feedparser 模块）和一个url集合。当第一个url添加到队列后，一个工作线程即可选择它并启动下载。循环将继续运行并添加相应的feed，直到全部加完，工作线程将轮流取出url去下载它们。

```
# Download the feed(s) and put the enclosure URLs into the queue.
for url in feed_urls:
  response = feedparser.parse(url, agent='fetch_podcasts.py')
  for entry in response['entries']:
    for enclosure in entry.get('enclosures', []):
      print 'Queuing:', enclosure['url']
      enclosure_queue.put(enclosure['url'])
```

剩下就可以等待队列为空。

```
# Now wait for the queue to be empty, indicating that we have
# processed all of the downloads.
print '*** Main thread waiting'
enclosure_queue.join()
print '*** Done'
```

下载如下 示例代码 ，运行即可看到如下输出：

```
0: Looking for the next enclosure
1: Looking for the next enclosure
Queuing: http://http.earthcache.net/htc-01.media.globix.net/COMP009996MOD1/Danny_Meyer.mp3
Queuing: http://feeds.feedburner.com/~r/drmoldawer/~5/104445110/moldawerinthemorning_show34_032607.mp3
Queuing: http://www.podtrac.com/pts/redirect.mp3/twit.cachefly.net/MBW-036.mp3
Queuing: http://media1.podtech.net/media/2007/04/PID_010848/Podtech_calacaniscast22_ipod.mp4
Queuing: http://media1.podtech.net/media/2007/03/PID_010592/Podtech_SXSW_KentBrewster_ipod.mp4
Queuing: http://media1.podtech.net/media/2007/02/PID_010171/Podtech_IDM_ChrisOBrien2.mp3
Queuing: http://feeds.feedburner.com/~r/drmoldawer/~5/96188661/moldawerinthemorning_show30_022607.mp3
*** Main thread waiting
0: Downloading: http://http.earthcache.net/htc-01.media.globix.net/COMP009996MOD1/Danny_Meyer.mp3
1: Downloading: http://feeds.feedburner.com/~r/drmoldawer/~5/104445110/moldawerinthemorning_show34_032607.mp3
```

```
0: Looking for the next enclosure
0: Downloading: http://www.podtrac.com/pts/redirect.mp3/twit.cachefly.net/MBW-036.mp3
1: Looking for the next enclosure
1: Downloading: http://media1.podtech.net/media/2007/04/PID_010848/Podtech_calacaniscast22_ipod.mp4
0: Looking for the next enclosure
0: Downloading: http://media1.podtech.net/media/2007/03/PID_010592/Podtech_SXSW_KentBrewster_ipod.mp4
0: Looking for the next enclosure
0: Downloading: http://media1.podtech.net/media/2007/02/PID_010171/Podtech_IDM_ChrisOBrien2.mp3
1: Downloading: http://feeds.feedburner.com/~r/drmoldawer/~5/96188661/moldawerinthemorning_show30_022607.mp3
1: Looking for the next enclosure
*** Done
```

CHAPTER

THREE

PYMOTW: STRINGIO AND CSTRINGIO

- 模块: StringIO 和 cStringIO
- 目的： 类似于file操作的文本缓冲区API
- python版本: StringIO: 1.4+, cStringIO: 1.5+

3.1 描述

类StringIO提供了一个在内存中方便处理文本的类文件(读, 写等操作)API. 他有两个独立的实现, 一个是用c实现的cStringIP模块,速度较快, 另一个是StringIO模块,他用python实现的以增强其可移植性. 使用cStringIO来处理大字符串可以提高运行性能,优于其他字符串串联技术.

3.2 例子

这里是一个好的,标准的,简单的.使用StringIO缓冲的例子:

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

"""
Simple examples with StringIO module
"""

# Find the best implementation available on this platform

try:
    from cStringIO import StringIO
except:
    from StringIO import StringIO

# Writing to a buffer
output = StringIO()
output.write('This goes into the buffer. ')

print >>output, 'And so does this.'
# Retrieve the value written
print output.getvalue()

output.close() # discard buffer memory

# Initialize a read buffer
input = StringIO('Inital value for read buffer')

# Read from the buffer
print input.read()
```

这个例子中使用了 `read()` ,但当然,函数 `readline()` 和 `readlines()` 也都是可用的.类StringIO提供 `seek()` 函数,因此在读取数据时可以任意跳到某个点上, 这也当你使用某些前看解析算法可以回头读取.

现实世界中,StringIO的应用包括一个网络应用程序栈,栈的各个部分都可以增加文本到响应response对象中,或者测试由程序某段输出(典型是写入到文件)的数据.

我们想在创建的工程应用中包括一个shell脚本接口,他是以多个命令行程序的形式.这些程序中的某些是负责从数据库中取数据,然后将这些数据转储到控制台(可以是现实给用户,也可以是文本中,这样就可以作为另一个命令的输入). 这些命令他们共享了一组格式化插件以便产生一个对象的多种文本表示(XML, bash语法, 人可读的,等等).

因为格式化可以将输出数据标准化后写到标准输出,如果没有StringIO模块,所得的测试结果可能会有些奇怪. 而使用了StringIO来拦截格式化的输出,这样以便我们用一个简单的方式来收集内存中的输出数据,来对预期的结果做比较.

3.3 参考

- The StringIO module ::: www.effbot.org
- Efficient String Concatenation in Python

CHAPTER

FOUR

PYMOTW: TEXTWRAP

- 模块：textwrap
- 目的： 通过调整段落中的换行符位置来格式化文本
- python版本：2.5

4.1 描述

textwrap模块可以用来格式化文本，使其在某些场合输出更美观。他提供了一些类似于在很多文本编辑器中都有的段落包装或填充特性的程序功能.

4.2 例子

```
import textwrap

# Provide some sample text
sample_text = '''

 The textwrap module can be used to format text for output in situations
 where pretty-printing is desired.  It offers programmatic functionality similar
 to the paragraph wrapping or filling features found in many text editors.

'''
```

fill()将文本作为输入，格式化文本作为输出。让我们看下面是如何对样本文本进行格式化的

```
print 'No dedent:\n'
print textwrap.fill(sample_text)
```

结果比我们想要的结果要少：

```
No dedent:

        The textwrap module can be used to format text for output in
        situations         where pretty-printing is desired.  It offers
        programmatic functionality similar         to the paragraph wrapping
        or filling features found in many text editors.
```

Note: 注意嵌入的tab符号和多余的空格被混合在输出文本中。这个看起来是很粗糙的。当然，我们可以做的更好。我们想在样本文本中的每一行的开始处删掉所有普通空格前缀。这个允许我们在去除代码本身的格式化时直接从Python代码中使用文档字符串或者嵌入式多行字符串。下面的样本字符串引入了一个人工的缩进层次以便更好的说明这个特征。

删除样本行中的普通空格前缀

```
dedented_text = textwrap.dedent(sample_text).strip()
print 'Dedented:\n'
print dedented_text
```

结果看上去似乎好点：

```
Dedented:

The textwrap module can be used to format text for output in situations
where pretty-printing is desired.  It offers programmatic functionality similar
to the paragraph wrapping or filling features found in many text editors.
```

由于“dedent”是“indent”的相反，结果就是将每行开始的普通空白符删除了。如果某行已经比另一行多了个缩进层次，那么对应的空格不会被去掉。

```
>>> a="""
...     one tab
...             two tab
...     one tab
... """
>>> import textwrap
>>> dedented_text = textwrap.dedent(a).strip()
>>> print dedented_text
one tab
    two tab
one tab
>>> print a
    one tab
        two tab
    one tab
>>>
```

接下来，让我们看下如果我们传递非缩进格式的文本给fill()，并使用一些不同的宽度值，会发生什么。

使用不同行宽值进行格式化输出：

```
# Format the output with a few different max line width values
for width in [ 20, 60, 80 ]:
    print
    print '%d Columns:\n' % width
    print textwrap.fill(dedented_text, width=width)
```

在指定不同宽度时会有以下不同的输出结果：

```
20 Columns:

The textwrap module
can be used to
format text for
output in situations
where pretty-
printing is desired.
It offers
programmatic
functionality
similar to the
paragraph wrapping
or filling features
```

```
found in many text
editors.
```

60 Columns:

```
The textwrap module can be used to format text for output in
situations where pretty-printing is desired.  It offers
programmatic functionality similar to the paragraph wrapping
or filling features found in many text editors.
```

80 Columns:

```
The textwrap module can be used to format text for output in situations where
pretty-printing is desired.  It offers programmatic functionality similar to the
paragraph wrapping or filling features found in many text editors.
```

除了制定输出中的宽度，你可以控制首行缩进，他独立于接下来的行。

```
# Demonstrate how to produce a hanging indent
print '\nHanging indent:\n'
print textwrap.fill(dedented_text, initial_indent='', subsequent_indent='    ')
```

这个看起来很容易就能实现文本的悬挂缩进，也就是首行要比后继行有少的缩进。

```
Hanging indent:

The textwrap module can be used to format text for output in
   situations where pretty-printing is desired.  It offers
   programmatic functionality similar to the paragraph wrapping or
   filling features found in many text editors.
```

缩进值也可以是非空格字符，因此，可以用*作为前缀产生bullet点，等等。在我转换老的zwiki内容以便将其导入到trac中是很灵活的。我使用Zope中的StructuredText包来解析zwiki数据，然后创建一个格式化器产生一个trac的wiki标记作为输出。使用textwrap就可以格式化输出页，因此转换后就几乎不需要再进行手工操作整个转换过程几乎没有手工进行。

4.3 参考

- textwrap_example.py

CHAPTER

FIVE

PYMOTW: LINECACHE

- 模块： linecache
- 目的： 从文件或者导入模块中检索文本行，对结果采用缓冲来提高读文件的效率。
- python版本： 1.4+

5.1 描述

python标准库处理python源文件中linecache模块被广泛的使用。缓冲的实现是读取文件的内容，并解析成行，保存在内存的字典中。API 根据索引返回列表中的请求行。在读取文件和寻找需要的行信息上可以节省一定时间。这对于从同一个文件中查询多行内容是非常有用的，比如为一个error report产生trackback。

5.2 示例

```
import linecache
import os
import tempfile
```

我们使用Lorem Ipsum generator产生的文本作为输入样例：

```
lorem = '''Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Vivamus eget elit. In posuere mi non risus. Mauris id quam posuere

lectus sollicitudin varius. Praesent at mi. Nunc eu velit. Sed augue
massa, fermentum id, nonummy a, nonummy sit amet, ligula. Curabitur
eros pede, egestas at, ultricies ac, pellentesque eu, tellus.

Sed sed odio sed mi luctus mollis. Integer et nulla ac augue convallis
accumsan. Ut felis. Donec lectus sapien, elementum nec, condimentum ac,
interdum non, tellus. Aenean viverra, mauris vehicula semper porttitor,
ipsum odio consectetuer lorem, ac imperdiet eros odio a sapien. Nulla
mauris tellus, aliquam non, egestas a, nonummy et, erat. Vivamus

sagittis porttitor eros.'''

# Create a temporary text file with some text in it
fd, temp_file_name = tempfile.mkstemp()

os.close(fd)
f = open(temp_file_name, 'wt')

try:
```

```
    f.write(lorem)
finally:
    f.close()
```

现在我们有了一个可用的临时文件，让我们更深入一步。从文件中读取的第5行是单一行。注意，在linecache中的行号是从1开始的。但是我们自己对字符串进行分割，那么索引号是从0开始。我们还需要从缓冲中返回的值进行过滤去除换行符。

```
# Pick out the same line from source and cache.
# (Notice that linecache counts from 1)
print 'SOURCE: ', lorem.split('\n')[4]
print 'CACHE : ', linecache.getline(temp_file_name, 5).rstrip()
```

下一步看下，如果我们需要的行为空将会发生什么。

```
# Blank lines include the newline
print '\nBLANK : "%s"' % linecache.getline(temp_file_name, 6)
```

如果请求的行号超过了文件中有效行号的范围，那么linecache会返回一个空字符串。

```
# The cache always returns a string, and uses
# an empty string to indicate a line which does
# not exist.
not_there = linecache.getline(temp_file_name, 500)
print '\nNOT THERE: "%s" includes %d characters' %  (not_there, len(not_there))
```

即使这个文件不存在，模块也不会抛出任何异常。

```
# Errors are even hidden if linecache cannot find the file
no_such_file = linecache.getline('this_file_does_not_exist.txt', 1)
print '\nNO FILE: ', no_such_file
```

虽然linecache模块经常用在输出tracebacks上，另一个重要特性是可以通过指定模块名在sys.path中寻找python模块源码。如果在当前路径中无法找到文件，那么linecache中的缓冲直接搜索sys.path中的模块。

```
# Look for the linecache module, using
# the built in sys.path search.
module_line = linecache.getline('linecache.py', 3)
print '\nMODULE : ', module_line
```

5.3 示例输出

```
SOURCE:  eros pede, egestas at, ultricies ac, pellentesque eu, tellus.
CACHE :  eros pede, egestas at, ultricies ac, pellentesque eu, tellus.

BLANK : "
"

NOT THERE: "" includes 0 characters

NO FILE:

MODULE :  This is intended to read lines from modules imported -- hence if a filename
```

5.4 参考

- PyMOTW

CHAPTER

SIX

PYMOTW: BISECT

- 模块： bisect
- 目的： 维持一个有序列表，当每次增加一个元素到列表时无需调用sort过程。
- python版本： 1.4+

6.1 描述

bisect模块实现了一个算法，用于向一个有序列表中插入一个元素。这比重复排序一个列表，或重构一个很大的有序列表要高效的多.

6.2 示例

使用 bisect.insort() 的简单示例，插入元素到一个有序列表中。

```
import bisect
import random

# Use a constant seed to ensure that we see
# the same pseudo-random numbers each time
# we run the loop.
random.seed(1)
# Generate 20 random numbers and
# insert them into a list in sorted
# order.
l = []
for i in range(1, 20):
    r = random.randint(1, 100)
    position = bisect.bisect(l, r)
    bisect.insort(l, r)
    print '%2d %2d' % (r, position), l
```

上述脚本的输出如下：

```
14 0 [14]
85 1 [14, 85]
77 1 [14, 77, 85]
26 1 [14, 26, 77, 85]
50 2 [14, 26, 50, 77, 85]
45 2 [14, 26, 45, 50, 77, 85]
66 4 [14, 26, 45, 50, 66, 77, 85]
79 6 [14, 26, 45, 50, 66, 77, 79, 85]
10 0 [10, 14, 26, 45, 50, 66, 77, 79, 85]
```

```
3 0 [3, 10, 14, 26, 45, 50, 66, 77, 79, 85]
84 9 [3, 10, 14, 26, 45, 50, 66, 77, 79, 84, 85]
44 4 [3, 10, 14, 26, 44, 45, 50, 66, 77, 79, 84, 85]
77 9 [3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]
1 0 [1, 3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]
45 7 [1, 3, 10, 14, 26, 44, 45, 45, 50, 66, 77, 77, 79, 84, 85]
73 10 [1, 3, 10, 14, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85]
23 4 [1, 3, 10, 14, 23, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85]
95 17 [1, 3, 10, 14, 23, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85, 95]
91 17 [1, 3, 10, 14, 23, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85, 91, 95]
```

第一列显示了新的随机数，第二列显示了数被插入到列表中的位置。最后是当前排序列表中的元素。

这是一个很简单的示例，我们处理的速度由于列表规模小以及每次只需排序一次，变的非常快速。但对于一个很长的list，利用这种方法能得到时间和内存上的节省。

你可能会注意到上述结果中存在一些重复值（45和77）.bisect模块提供了2种方法来处理重复，新值可以插入到已经存在值的左边或者右边。对应的是 insort_right()函数，可以将值插入已有值的后面（右边），insort_left()函数可以插入到之前（左边）。

如果我们使用bisect_left()和bisect_right()来处理同样的数据，那么最后获得的list是相同的，只是中间插入的位置会有不同。

```
# Reset the seed
random.seed(1)

# Use bisect_left and insort_left.
l = []
for i in range(1, 20):
    r = random.randint(1, 100)
    position = bisect.bisect_left(l, r)
    bisect.insort_left(l, r)
    print '%2d %2d' % (r, position), l
```

```
14 0 [14]
85 1 [14, 85]
77 1 [14, 77, 85]
26 1 [14, 26, 77, 85]
50 2 [14, 26, 50, 77, 85]
45 2 [14, 26, 45, 50, 77, 85]
66 4 [14, 26, 45, 50, 66, 77, 85]
79 6 [14, 26, 45, 50, 66, 77, 79, 85]
10 0 [10, 14, 26, 45, 50, 66, 77, 79, 85]
3 0 [3, 10, 14, 26, 45, 50, 66, 77, 79, 85]
84 9 [3, 10, 14, 26, 45, 50, 66, 77, 79, 84, 85]
44 4 [3, 10, 14, 26, 44, 45, 50, 66, 77, 79, 84, 85]
77 8 [3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]
1 0 [1, 3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]
45 6 [1, 3, 10, 14, 26, 44, 45, 45, 50, 66, 77, 77, 79, 84, 85]
73 10 [1, 3, 10, 14, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85]
23 4 [1, 3, 10, 14, 23, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85]
95 17 [1, 3, 10, 14, 23, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85, 95]
91 17 [1, 3, 10, 14, 23, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85, 91, 95]
```

除了python实现外，还有一个更快的c实现，如果c版本存在，那么 import bisect 模块时会自动调用c版本而不是调用python版本。

6.3 参考

- Insertion Sort

CHAPTER

SEVEN

PYMOTW: LOGGING

- 模块：logging
- 目的：为python模块提供状态、错误、信息输出的标准接口
- python版本：2.3

7.1 描述

logging模块定义了一个标准API，用于报告所有你使用的模块的错误和状态信息.标准库模块中提供logging API的最重要意义是所有python模块可以参与到日志记录中，因此你的应用程序日志可以包含来自第三方模块的信息.

当然，在不同层次上或因不同目的来记录日志信息是有必要的. 将日志信息写入到文件，如，HTTP GET/POST的地理信息，通过SMTP发送的邮件，一般的sockets，或者特定OS的日志机制都是被标准模块支持的. 如果你有特殊需求，任何内置模块都不能满足的话，你也可以创建你自己的日志目标类.

7.2 例子

大多数应用程序可能会将日志写入到文件中，所以让我从这个例子开始讲述. 我们使用basicConfig()函数来设置默认的处理用于将调试信息写入到文件.

```
import logging
LOG_FILENAME = '/tmp/logging_example.out'
logging.basicConfig(filename=LOG_FILENAME,level=logging.DEBUG,)

logging.debug('This message should go to the log file')
```

现在如果我们打开这个文件，看看里面是什么，我们应该可以找到以下的日志信息：

```
f = open(LOG_FILENAME, 'rt')
try:
    body = f.read()
finally:
    f.close()
    print 'FILE:'
    print body
    print


FILE:
DEBUG:root:This message should go to the log file
```

如果我们重复运行之前的脚本，那么另外的日志信息会附加到文件末尾． 为了每次能够创建一个新的文件，你可以传递一个filemode参数值为 'w' 给basicConfig().尽管你自己不能控制这个日志文件大小，但，可以使用RotatingFileHandler，这更方便：

```
import glob
import logging
import logging.handlers

LOG_FILENAME = '/tmp/logging_rotatingfile_example.out'

# Set up a specific logger with our desired output level
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

# Add the log message handler to the logger
handler = logging.handlers.RotatingFileHandler(LOG_FILENAME, maxBytes=20, backupCount=5)
my_logger.addHandler(handler)

for i in range(20):
    my_logger.debug('i = %d' % i)

# See what files are created
logfiles = glob.glob('%s*' % LOG_FILENAME)

for filename in logfiles:
    print filename
```

结果应该是6个独立的文件，每个都含有应用程序的日志历史：

```
/tmp/logging_rotatingfile_example.out
/tmp/logging_rotatingfile_example.out.1
/tmp/logging_rotatingfile_example.out.2
/tmp/logging_rotatingfile_example.out.3
/tmp/logging_rotatingfile_example.out.4
/tmp/logging_rotatingfile_example.out.5 ## 5      backupCount=5
```

当前日志文件总是为/tmp/logging_rotatingfile_example.out，每次当文件大小达到限制时，就以后缀.1来重命名． 每个已存的备份文件也依次重命名为原先后缀增一(如，.1成为.2)，.5文件会被擦除．

显然的，这个例子中设置了日志的长度太太太小了． 所以在实际程序下，你可以为maxBytes设置一个合适的值．

使用日志API的另外一个有用的地方是能够在不同日志层次上产生不同的信息． 这能够让你书写的代码中带有调试信息，例如，降低日志层次以便这些调试信息不输出到你的生产系统中．

```
CRITICAL 50
ERROR 40
WARNING 30
INFO 20
DEBUG 10
UNSET 0
```

日志记录器，handler，日志信息可以分别调用不同的层次 ． 一条日志信息，只有当处理和日志记录器被设置为和它一样的层次或比它低层次时，才被输出． 例如，如果一个信息是CRITICAL，记录器被设置为ERROR，那么这个消息会输出来． 如果一个信息是WARNING，记录器被设置为ERROR，那么这个信息不被输出．

```
import logging
import sys

LEVELS = { 'debug':logging.DEBUG,
```

```
            'info':logging.INFO,
            'warning':logging.WARNING,
            'error':logging.ERROR,
            'critical':logging.CRITICAL,
}

if len(sys.argv) > 1:
    level_name = sys.argv[1]
    level = LEVELS.get(level_name, logging.NOTSET)
    logging.basicConfig(level=level)

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical error message')
```

运行这个脚本时指定参数，如 'debug' 或 'warning'，看看在不同层次上，哪些信息会显示出来：

```
$ python logging_level_example.py debug
DEBUG:root:This is a debug message
INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical error message

$ python logging_level_example.py info
INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical error message
```

你可能不会注意到这些日志信息中都含有 'root'。这个日志模块支持一个不同名字日志记录器的层次结构。一个告知某条日志信息来自于哪个日志器的简单方式是对每个模块使用独立的日志器对象。每个新的日志器从它的父亲中''继承''一些配置，日志信息发送到一个包含父日志器名字的日志器。可选的，每个日志器可以配置不同，以便让来自不同模块的信息按不同的方式处理。让我们看个简单的例子看怎样记录来自不同模块的信息，这也便于追踪信息的对应源代码：

```
import logging

logging.basicConfig(level=logging.WARNING)

logger1 = logging.getLogger('package1.module1')
logger2 = logging.getLogger('package2.module2')

logger1.warning('This message comes from one module')
logger2.warning('And this message comes from another module')
```

输出为：

```
$ python logging_modules_example.py
WARNING:package1.module1:This message comes from one module
WARNING:package2.module2:And this message comes from another module
```

还有许多许多许多配置日志记录的选项，包括不同日志信息格式化选项，将信息发送到多个记录器，使用socket接口改变一个正在运行的长时间程序的配置。所有这些选项进一步在 库模块文档 中深入。

7.3 参考

- PEP 282
- Python Standard Logging

CHAPTER

EIGHT

PYMOTW: LOCALE

- 模块： locale
- 目的： POSIX标准的本地化API
- python版本： 1.5,在2.5版本中有所扩展

8.1 描述

locale模块是Python国际化和本地化支持库的一部分． 他提供一种用于处理那些可能依赖于你用户语言或位置的操作的标准方式． 例如， 货币格式化， 比较字符串以便排序， 处理时间日期． 他没有包含翻译(可参见gettext模块)或Unicode编码．

由于可以在应用程序范围内改变本地化设置， 所以推荐用户避免在库中改变值而是让应用程序一次性设置． 在下面的例子中， 我会改变本地的一些时间以便说明目的．这更像是一旦你的应用程序启动就去设置本地化参数．

8.2 例子

让用户改变一个应用程序的本地设置的最一般的方式是通过一个环境变量（LC__ALL， LC__CTYPE， LANG， 或LANGUAGE， 这依赖于你的平台）． 然后程序会调用locale.setlocale()， 没有使用硬编码值， 而是使用环境变量．

```
import locale
import os
import pprint

print 'Environment settings:'
for env_name in [ 'LC_ALL', 'LC_CTYPE', 'LANG', 'LANGUAGE' ]:
    print '\t%s = %s' % (env_name, os.environ.get(env_name, ''))

# What is the default locale?
print
print 'Default locale:', locale.getdefaultlocale()

# Default settings based on the user's environment.
locale.setlocale(locale.LC_ALL, '')

# If we do not have a locale, assume US English.
print 'From environment:', locale.getlocale()

pprint.pprint(locale.localeconv())
```

在我的Mac上， 这个程序输出类似如下：

```
$ python locale_env_example.py
Environment settings:
 LC_ALL =
 LC_CTYPE =
 LANG =
 LANGUAGE =

Default locale: (None, 'mac-roman')
From environment: (None, None)
{'currency_symbol': '',
 'decimal_point': '.',
 'frac_digits': 127,
 'grouping': [127],
 'int_curr_symbol': '',
 'int_frac_digits': 127,
 'mon_decimal_point': '',
 'mon_grouping': [127],
 'mon_thousands_sep': '',
 'n_cs_precedes': 127,
 'n_sep_by_space': 127,
 'n_sign_posn': 127,
 'negative_sign': '',
 'p_cs_precedes': 127,
 'p_sep_by_space': 127,
 'p_sign_posn': 127,
 'positive_sign': '',
 'thousands_sep': ''}
```

现在如果我们设置好LANG值后再运行同样的脚本，可以看到本地设置和默认编码因此改变：

法国：

```
$ LANG=fr_FR python locale_env_example.py
Environment settings:
 LC_ALL =
 LC_CTYPE =
 LANG = fr_FR
 LANGUAGE =

Default locale: (None, 'mac-roman')
From environment: ('fr_FR', 'ISO8859-1')
{'currency_symbol': 'Eu',
 'decimal_point': ',',
 'frac_digits': 2,
 'grouping': [127],
 'int_curr_symbol': 'EUR ',
 'int_frac_digits': 2,
 'mon_decimal_point': ',',
 'mon_grouping': [3, 3, 0],
 'mon_thousands_sep': ' ',
 'n_cs_precedes': 0,
 'n_sep_by_space': 1,
 'n_sign_posn': 2,
 'negative_sign': '-',
 'p_cs_precedes': 0,
 'p_sep_by_space': 1,
 'p_sign_posn': 1,
 'positive_sign': '',
 'thousands_sep': ''}
```

西班牙：

```
$ LANG=es_ES python locale_env_example.py
Environment settings:
 LC_ALL =
 LC_CTYPE =
 LANG = es_ES
 LANGUAGE =

Default locale: (None, 'mac-roman')
From environment: ('es_ES', 'ISO8859-1')
{'currency_symbol': 'Eu',
 'decimal_point': ',',
 'frac_digits': 2,
 'grouping': [127],
 'int_curr_symbol': 'EUR ',
 'int_frac_digits': 2,
 'mon_decimal_point': ',',
 'mon_grouping': [3, 3, 0],
 'mon_thousands_sep': '.',
 'n_cs_precedes': 1,
 'n_sep_by_space': 1,
 'n_sign_posn': 1,
 'negative_sign': '-',
 'p_cs_precedes': 1,
 'p_sep_by_space': 1,
 'p_sign_posn': 1,
 'positive_sign': '',
 'thousands_sep': ''}
```

葡萄牙：

```
$ LANG=pt_PT python locale_env_example.py
Environment settings:
 LC_ALL =
 LC_CTYPE =
 LANG = pt_PT
 LANGUAGE =

Default locale: (None, 'mac-roman')
From environment: ('pt_PT', 'ISO8859-1')
{'currency_symbol': 'Eu',
 'decimal_point': ',',
 'frac_digits': 2,
 'grouping': [127],
 'int_curr_symbol': 'EUR ',
 'int_frac_digits': 2,
 'mon_decimal_point': '.',
 'mon_grouping': [3, 3, 0],
 'mon_thousands_sep': '.',
 'n_cs_precedes': 0,
 'n_sep_by_space': 1,
 'n_sign_posn': 1,
 'negative_sign': '-',
 'p_cs_precedes': 0,
 'p_sep_by_space': 1,
 'p_sign_posn': 1,
 'positive_sign': '',
 'thousands_sep': ' '}
```

波兰：

```
$ LANG=pl_PL python locale_env_example.py
Environment settings:
 LC_ALL =
 LC_CTYPE =
 LANG = pl_PL
 LANGUAGE =

Default locale: (None, 'mac-roman')
From environment: ('pl_PL', 'ISO8859-2')
{'currency_symbol': 'z?\x82',
 'decimal_point': ',',
 'frac_digits': 2,
 'grouping': [3, 3, 0],
 'int_curr_symbol': 'PLN ',
 'int_frac_digits': 2,
 'mon_decimal_point': ',',
 'mon_grouping': [3, 3, 0],
 'mon_thousands_sep': ' ',
 'n_cs_precedes': 1,
 'n_sep_by_space': 2,
 'n_sign_posn': 4,
 'negative_sign': '-',
 'p_cs_precedes': 1,
 'p_sep_by_space': 2,
 'p_sign_posn': 4,
 'positive_sign': '',
 'thousands_sep': ' '}
```

所以你可以看到货币符号(currency_symbol)设置改变了, 从小数中分离出整个数字的分割字符(decimal_point)也改变了, 等等. 现在以不同的地区设置(US 美元, 欧元, 和Polish złoty)格式输出同样的信息:

```
sample_locales = [ ('USA', 'en_US'),
                   ('France', 'fr_FR'),
                   ('Spain', 'es_ES'),
                   ('Portugal', 'pt_PT'),
                   ('Poland', 'pl_PL'),
                 ]

for name, loc in sample_locales:
    locale.setlocale(locale.LC_ALL, loc)
    print '%20s: %s' % (name, locale.currency(1234.56))
```

输出一个小的表格:

```
$ python locale_currency_example.py
USA: $1234.56
France: 1234,56 Eu
Spain: Eu 1234,56
Portugal: 1234.56 Eu
Poland: zł 1234,56
```

除了以不同的格式输出外, 本地化模块还可以帮助解析输入. 不同的文化对数字格式化使用不同的转换(上面已列出). 本地化模块提供atoi()和atof()函数分别用来进行字符串与整数和浮点数之间的转换.

```
sample_data = [ ('USA', 'en_US', '1234.56'),
                ('France', 'fr_FR', '1234,56'),
                ('Spain', 'es_ES', '1234,56'),
                ('Portugal', 'pt_PT', '1234.56'),
```

```
                ('Poland', 'pl_PL', '1234,56'),
              ]

for name, loc, a in sample_data:
    locale.setlocale(locale.LC_ALL, loc)
    f = locale.atof(a)
    locale.setlocale(locale.LC_ALL, 'en_US')
    print '%20s: %7s => %f' % (name, a, f)


$ python locale_atof_example.py
USA: 1234.56 => 1234.560000
France: 1234,56 => 1234.560000
Spain: 1234,56 => 1234.560000
Portugal: 1234.56 => 1234.560000
Poland: 1234,56 => 1234.560000
```

另一个本地化的重要方面是时间和日期的格式化：

```
import locale
import time

sample_locales = [ ('USA', 'en_US'),
                   ('France', 'fr_FR'),
                   ('Spain', 'es_ES'),
                   ('Portugal', 'pt_PT'),
                   ('Poland', 'pl_PL'),
                 ]

for name, loc in sample_locales:
    locale.setlocale(locale.LC_ALL, loc)
    print '%20s: %s' % (name, time.strftime(locale.nl_langinfo(locale.D_T_FMT)))


$ python locale_date_example.py

USA: Sun May 20 10:19:54 2007
France: Dim 20 mai 10:19:54 2007
Spain: dom 20 may 10:19:54 2007
Portugal: Dom 20 Mai 10:19:54 2007
Poland: ndz 20 maj 10:19:54 2007
```

这个星期我只阐述了本地化模块中的一些高层函数． 还有其他低层(格式化字符串)或那些管理你应用程序本地化的函数(resetlocale)． 和往常一样，你可以参考Python库文档来查看些细节．

8.3 参考

- Locale - Wikipedia
- Internationalization and localization - Wikipedia
- OpenI18N.org - The Free standards Group Open Internationalisation Initiative
- MSDN - National Language Support Constants
- Internationalizing Python

CHAPTER

NINE

PYMOTW: OS

- 模块：os
- 目的：为访问操作系统的特定属性提供方便。
- python版本：1.4+

9.1 描述

os模块提供了对特定平台模块(如posix，nt，mac)的封装，函数提供的api在很多平台上都可以相同使用，所以使用os模块会变得很方便。但不是所有函数在所有平台上都可用，比如在本文中到的一些管理函数在windows上无法使用。

在Python文档中os模块的副标题是“操作系统混合接口”，模块包含的大部分函数用于创建和管理活动进程和文件系统（文件和目录），当然除此之外还有其它一些函数。本文中，我们对如何获取和设置进程参数来进行讨论。

Note: 以下示例代码有的只能在linux平台上工作。

9.2 属主处理

首先讨论用来检查和改变进程属主id的函数。在守护进程和特殊的系统程序需要改变执行权限而不使用root情况下这往往是非常有用的。这里我不会太过详细的解释linux的安全，进程属主等具体含义，这些可以见参考中的文章来获得更详细的介绍。

我们给出一段脚本来获取一个进程的有效用户和组信息，然后改变这些有效值。这类似于一个守护进程在系统启动时以root身份启动加载，然后降低权限并作为一个普通用户运行。如果你下载示例并试运行，你可以设置user相应的值为TEST_GID和TEST_UID。

```
import os

TEST_GID=501
TEST_UID=527

def show_user_info():
     print 'Effective User  :', os.geteuid()
     print 'Effective Group :', os.getegid()
     print 'Actual User    :', os.getuid(), os.getlogin()
     print 'Actual Group   :', os.getgid()
     print 'Actual Groups   :', os.getgroups()
     return

print 'BEFORE CHANGE:'
show_user_info()
print
```

```
try:
     os.setegid(TEST_GID)
except OSError:
     print 'ERROR: Could not change effective group.  Re-run as root.'
else:
     print 'CHANGED GROUP:'
     show_user_info()
     print

try:
     os.seteuid(TEST_UID)
except OSError:
     print 'ERROR: Could not change effective user.  Re-run as root.'
else:
     print 'CHANGE USER:'
     show_user_info()
     print
```

当我运行在DELL D630 Ubuntu上时，得到的结果如下：

```
~ 16:51:33> ./a.py
BEFORE CHANGE:
Effective User  : 1000
Effective Group : 1000
Actual User     : 1000 cjj
Actual Group  : 1000
Actual Groups   : [4, 20, 24, 25, 29, 30, 44, 46, 104, 108, 110, 115, 117, 1000]

ERROR: Could not change effective group.  Re-run as root.
ERROR: Could not change effective user.  Re-run as root.
```

注意，当我使用非root运行时，值未被改变，我所启动的进程不可以改变他们自身有效的属性。如果我试图设置其他的用户名和组id，那么会抛出OSError异常。

下面，我们以root权限来运行这段脚本：

```
~ 16:51:10> sudo ./a.py
[sudo] password for cjj:
BEFORE CHANGE:
Effective User  : 0
Effective Group : 0
Actual User     : 0 cjj
Actual Group  : 0
Actual Groups   : [0]

CHANGED GROUP:
Effective User  : 0
Effective Group : 501
Actual User     : 0 cjj
Actual Group  : 0
Actual Groups   : [0]

CHANGE USER:
Effective User  : 527
Effective Group : 501
Actual User     : 0 cjj
Actual Group  : 0
Actual Groups   : [0]
```

在这个例子中，如果我们以root权限运行，那么我们可以改变这个进程的用户和组属性。一旦我们改变了UID，那么进程将受这个用户的权限限制，非root用户是无法改变有效用户组，所以首先我们需要

改变用户组，然后再改变用户名。

除了查找和改变进程属主，还有其他函数可以获取当前进程和父进程的id，查找和改变其进程用户组和会话id，与控制终端id是一样的。在你编写复杂程序（如自己的终端命令行程序）中使用这些函数可以帮助你在进程之间传递信号。

9.3 环境处理

通过os模块，你的程序可以访问的另一个操作系统特性是系统环境。通过os.environ和os.getenv()可以访问在环境中设置的变量字符串。环境变量常用来作为配置像搜索路径，文件路径、调试标志的值。下面的示例检索了一个环境变量，并且通过子进程改变了这个值。

```
print 'Initial value:', os.environ.get('TESTVAR', None)
print 'Child process:'
os.system('echo $TESTVAR')

os.environ['TESTVAR'] = 'THIS VALUE WAS CHANGED'

print
print 'Changed value:', os.environ['TESTVAR']
print 'Child process:'
os.system('echo $TESTVAR')

del os.environ['TESTVAR']

print
print 'Removed value:', os.environ.get('TESTVAR', None)
print 'Child process:'
os.system('echo $TESTVAR')
```

os.environ对象遵循标准的Python映射API以便检索和设置值。 os.environ值的改变将被输出到子进程中。

```
$ python os_environ_example.py
Initial value: None
Child process:


Changed value: THIS VALUE WAS CHANGED
Child process:
THIS VALUE WAS CHANGED

Removed value: None
Child process:
```

9.4 工作目录处理

在操作系统的文件系统结构中有一个概念是“当前工作目录”。在文件系统中，当前进程在访问用相对路径表示的文件时，就把这个目录当作默认目录位置。

```
print 'Starting:', os.getcwd()
print os.listdir(os.curdir)

print 'Moving up one:', os.pardir
os.chdir(os.pardir)
```

```
print 'After move:', os.getcwd()
print os.listdir(os.curdir)
```

注意os.curdir()和os.pardir()是指向当前目录和父目录的一种快捷方式。结果很显然：

```
Starting: /Users/dhellmann/Documents/PyMOTW/PyMOTW/os
['.svn', '__init__.py', 'os_cwd_example.py', 'os_environ_example.py',
'os_process_id_example.py', 'os_process_user_example.py']
Moving up one: ..
After move: /Users/dhellmann/Documents/PyMOTW/PyMOTW
['.svn', '__init__.py', 'bisect', 'ConfigParser', 'fileinput', 'linecache',
'locale', 'logging', 'os', 'Queue', 'StringIO', 'textwrap']
```

9.5 后续...

这里我们仅介绍了os模块中查找和设置进程参数的一些函数。下一次，我们将介绍os模块来管理文件系统对象。

9.6 参考

- Python Reference Manual, Process Parameters
- Speaking UNIX, Part 8: UNIX processes
- geteuid
- getsid
- setpgrp

CHAPTER

TEN

PYMOTW: OS(2)

10.1 描述

上一部分，我们讨论了进程参数，现在我们讨论一下os模块提供的输入/输出特性。

10.2 管道

os模块提供了一些函数，这些函数利用管道来管理子进程的IO操作。这些函数的工作方式基本相同，但根据输入/输出的需求类型返回不同的文件句柄。相对于2.4版本中的 subprocess 模块这些函数是过时了，但这是一个很好的机会，你可以在已有的代码中看到它们。

管道中经常使用的是popen()函数，它创建一个新的进程用于运行给定的命令并且根据模式选项附加给这个进程一个单一的输入输出数据流 。虽然在Windows中可以使用popen()，但以下例子假设以Unix shell方式运行，其中流的概念也是unix技术。

stdin：进程（文件描述符0）的标准输入流，对于这个进程来说是可读的，通常指终端输入。

stdout：进程（文件描述符1）的标准输出流，对于这个进程来说是可写的，通常用于给用户显示非错误信息。

stderr：进程（文件描述符2）的标准错误流，对于这个进程来说是可写的，通常用于传递错误信息。

```
import os

print '\npopen, read:'
pipe_stdout = os.popen('echo "to stdout"', 'r')
try:
    stdout_value = pipe_stdout.read()
finally:
    pipe_stdout.close()
print '\tstdout:', repr(stdout_value)

print '\npopen, write:'
pipe_stdin = os.popen('cat -', 'w')
try:
    pipe_stdin.write('\tstdin: to stdin\n')
finally:
    pipe_stdin.close()
```

```
popen, read:
     stdout: 'to stdout\n'

popen, write:
     stdin: to stdin
```

从子进程的流中读取或者写入的方法是比较受限的，popen提供了额外的流，如stdin、stdout、stderr来以便使用。

比如，popen2()函数返回一个与子进程标准输入绑定的只写流和一个与子进程标准输出绑定的只读流。

```
print '\npopen2:'
pipe_stdin, pipe_stdout = os.popen2('cat -')
try:
    pipe_stdin.write('through stdin to stdout')
finally:
    pipe_stdin.close()

try:
    stdout_value = pipe_stdout.read()
finally:
    pipe_stdout.close()
print '\tpass through:', repr(stdout_value)
```

这个简单例子解释了双向通信方式，从stdin写入的值被cat命令读取（'-`参数的作用），然后由stdout输出。显然，一个复杂的进程通过管道可以来回传递其它类型的信息，甚至是序列化对象。

```
popen2:
     pass through: 'through stdin to stdout'
```

有些情况下，希望同时访问stdout和stderr，stdout常用于输出信息，stderr常用于抛出错误。因此分别读取他们可以减少解析错误消息的复杂度，而popen3函数返回一个新进程的3个流stdin、stdout、stderr。

```
print '\npopen3:'
pipe_stdin, pipe_stdout, pipe_stderr = os.popen3('cat -; echo ";to stderr" 1>&2')
try:
    pipe_stdin.write('through stdin to stdout')
finally:
    pipe_stdin.close()
try:
    stdout_value = pipe_stdout.read()
finally:
    pipe_stdout.close()
print '\tpass through:', repr(stdout_value)
try:
    stderr_value = pipe_stderr.read()
finally:
    pipe_stderr.close()
print '\tstderr:', repr(stderr_value)
```

注意，我们需要分别读取和关闭这些流，在处理多进程的IO中，还涉及到流程控制和排序，I/O即为缓冲器，如果想读取流中的所有数据，那么子进程必须关闭这个流来表示文件的结束 ，更多信息可以参考Python库文档 Flow Control Issues

```
popen3:
     pass through: 'through stdin to stdout'
     stderr: ';to stderr\n'
```

最后，popen4()返回两个流，stdin和stdout/stderr的组合，这对于命令的结果需要被记录，但不需要解析是很有用的。

```
print '\npopen4:'
pipe_stdin, pipe_stdout_and_stderr = os.popen4('cat -; echo ";to stderr" 1>&2')
try:
    pipe_stdin.write('through stdin to stdout')
```

```
finally:
    pipe_stdin.close()
try:
    stdout_value = pipe_stdout_and_stderr.read()
finally:
    pipe_stdout.close()
print '\tcombined output:', repr(stdout_value)
```

```
popen4:
     combined output: 'through stdin to stdout;to stderr\n'
```

另外，除了接收简单的字符串命令来传递给shell解析，popen2()、popen3()、popen4()函数同样接收字符串序列（命令，加参数），这种情况中，参数不是传递给shell的。

```
print '\npopen2, cmd as sequence:'
pipe_stdin, pipe_stdout = os.popen2(['cat', '-'])
try:
    pipe_stdin.write('through stdin to stdout')
finally:
    pipe_stdin.close()
try:
    stdout_value = pipe_stdout.read()
finally:
    pipe_stdout.close()
print '\tpass through:', repr(stdout_value)
```

```
popen2, cmd as sequence:
     pass through: 'through stdin to stdout'
```

10.3 后续

下次，我们将讨论如何来控制文件描述符。

10.4 参考

- Unix Concepts for more discussion of stdin, stdout, and stderr
- File Object Creation with the os module
- subprocess
- Flow Control Issues

CHAPTER

ELEVEN

PYMOTW: OS(3)

11.1 描述

前面讲述了如何来处理进程参数和输入/输出，本周我将探讨一些操作文件和目录的函数。

11.2 文件描述符

os模块中包含了一些函数集用于处理底层的文件描述符（当前进程打开属主文件所使用的整型），相比file()对象来说这些是更底层的API，在本文中将不会解释什么是文件描述符，它通常可以很好的和file()对象协同工作，更多细节可以参考 这里 来了解如何使用文件描述符。

11.3 文件系统权限

os.access()可以测试一个进程对一个文件是否有可访问权限。

```
import os

print 'Testing:', __file__
print 'Exists:', os.access(__file__, os.F_OK)
print 'Readable:', os.access(__file__, os.R_OK)
print 'Writable:', os.access(__file__, os.W_OK)
print 'Executable:', os.access(__file__, os.X_OK)
```

这个结果将取决于你如何来运行这个示例程序，可能会显示如下：

```
$ python os_access.py
Testing: os_access.py
Exists: True
Readable: True
Writable: True
Executable: False
```

os.access()模块包含了2个特殊的含义，首先，在实际使用open()函数之前使用os.access()函数来判断一个文件是否可访问是没有意义的。这里有个小事实，在函数的两次调用之间可能会改变文件的权限。另外一个含义是该函数适合于大部分扩展的POSIX许可语义的网络文件系统。文件系统对于一个进程对文件有访问权限的POSIX调用会做出响应，在调用open()时，因为某些原因没有通过POSIX的调用测试，那么会报告失败。总之，最好时在特定的模式中使用open()，如果出现错误还可以捕获IOError异常。

如果想获得更多关于文件的信息，可以查阅stat()或者os.lstat（如果你查看的文件一个动态链接的话）。

```
import os
import sys
import time

if len(sys.argv) == 1:
  filename = __file__
  else:
    filename = sys.argv[1]

stat_info = os.stat(filename)

print 'os.stat(%s):' % filename
print '\tSize:', stat_info.st_size
print '\tPermissions:', oct(stat_info.st_mode)
print '\tOwner:', stat_info.st_uid
print '\tDevice:', stat_info.st_dev
print '\tLast modified:', time.ctime(stat_info.st_mtime)
```

再次申明，你得到的结果将取决于你运行的方式，可以尝试向os_stat.py传递不同的文件名看看。

```
$ python os_stat.py
os.stat(os_stat.py):
    Size: 1547
    Permissions: 0100644
    Owner: 527
    Device: 234881026
    Last modified: Sun Jun 10 08:13:26 2007
```

在Unix类型系统上，文件权限可以由chmod()来修改，以整形形式传递。形式值可以用stat模块的常量值来b表示。以下示例了如何来触发用户的可执行权限位。

```
import os
import stat

# Determine what permissions are already set using stat
existing_permissions = stat.S_IMODE(os.stat(__file__).st_mode)

if not os.access(__file__, os.X_OK):
  print 'Adding execute permission'
  new_permissions = existing_permissions | stat.S_IXUSR
else:
  print 'Removing execute permission'
  # use xor to remove the user execute permission
  new_permissions = existing_permissions ^ stat.S_IXUSR

os.chmod(__file__, new_permissions)
```

运行该脚本前假设你有修改文件模式的权限。

```
$ python os_stat_chmod.py
Adding execute permission
$ python os_stat_chmod.py
Removing execute permission
```

11.4 目录

同样提供了一些处理文件系统中目录的函数，包括创建内容列表和删除它们。

```
import os

dir_name = 'os_directories_example'

print 'Creating', dir_name
os.makedirs(dir_name)

file_name = os.path.join(dir_name, 'example.txt')
print 'Creating', file_name
f = open(file_name, 'wt')
try:
  f.write('example file')
finally:
  f.close()

print 'Listing', dir_name
print os.listdir(dir_name)

print 'Cleaning up'
os.unlink(file_name)
os.rmdir(dir_name)
```

```
$ python os_directories.py
Creating os_directories_example
Creating os_directories_example/example.txt
Listing os_directories_example
['example.txt']
Cleaning up
```

有2个函数集用来创建和删除目录，当使用os.mkdir()创建一个新的目录时，其父目录必须存在。当使用os.rmdir()来删除一个目录时候，那么只有目录树的叶子节点（目录的最后一级）可以被删除。相比下，os.makedirs()和os.removedirs()可以操作当前路径下的所有目录，os.makedirs()可以创建路径不存在的目录，os.removedirs()可以删除包含父目录的子目录（当然前提有这个权限）。

11.5 符号链接

很多文件系统和平台都支持它，同样有一些函数可以用来处理它们。

```
import os, tempfile

link_name = tempfile.mktemp()

print 'Creating link %s->%s' % (link_name, __file__)
os.symlink(__file__, link_name)

stat_info = os.lstat(link_name)
print 'Permissions:', oct(stat_info.st_mode)

print 'Points to:', os.readlink(link_name)

# Cleanup
os.unlink(link_name)
```

虽然os中包含了os.temparm()来创建临时文件，当时相比tempfile模块还不够安全，在使用中可能会产生RuntimeWarning信息，更好的方法使用tempfile模块。

```
$ python os_symlinks.py
Creating link /tmp/tmpRxRiHn->os_symlinks.py
Permissions: 0120755
Points to: os_symlinks.py
```

11.6 访问目录树

os.walk()可以递归遍历一个目录,对于每一个目录，可以产生一个包含目录路径、当前路径的子目录列表，以及在子目录中的文件，以下示例展示了一个遍历目录的简单方法：

```
import os, sys

# If we are not given a path to list, use /tmp
if len(sys.argv) == 1:
  root = '/tmp'
else:
  root = sys.argv[1]

for dir_name, sub_dirs, files in os.walk(root):
  print '\n', dir_name
  # Make the subdirectory names stand out with /
  sub_dirs = [ '%s/' % n for n in sub_dirs ]
  # Mix the directory contents together
  contents = sub_dirs + files
  contents.sort()
  # Show the contents
  for c in contents:
    print '\t%s' % c


$ python os_walk.py

/tmp
   .KerberosLogin-0--1074266944 (inited,root,local)/
   .KerberosLogin-527-4839472 (inited,gui,tty,local)/
   527/
   cs_cache_lock_527
   cs_cache_lock_92
   emacs527/
   fry.log
   hsperfdata_dhellmann/
   objc_sharing_ppc_4294967294
   objc_sharing_ppc_527
   objc_sharing_ppc_92
   svn.arg.1835159
   var_backups/

/tmp/.KerberosLogin-527-4839472 (inited,gui,tty,local)
   KLLCCache.lock

/tmp/527
   /tmp/emacs527
   server
   /tmp/hsperfdata_dhellmann
   976

/tmp/var_backups
   infodir.bak
   local.nidump
```

11.7 后续

下次，我们讨论os模块中创建和管理进程的函数。

11.8 参考

- Working with Files and Directories
- tempfile module

CHAPTER

TWELVE

PYMOTW: OS(4)

12.1 描述

这周,我总结整个os模块(但保留os.path的内容作为将来独立的一篇)并讨论一些有利于处理多进程的函数.我在part2中已经介绍了管道的使用， 这周我们来看下system(),fork(),exec()这3个函数和他们之间的关系.

12.2 申明

这里的许多函数都有可移植性限制.可以查看subprocess模块以获得一种更一致的平台独立的处理进程方式.

12.3 运行外部命令

最简单的运行一条单独命令,没有一点交互的方式是使用os.system()． 他获取一个字符串,这个字符串就是一将被命令行执行的命令,通过一个shell中的子进程来执行.

```
import os

# Simple command
os.system('ls -l')
```

```
$ python os_system_example.py
total 168
-rw-r--r-- 1 dhellman dhellman 0 May 27 06:58 __init__.py
-rw-r--r-- 1 dhellman dhellman 1391 Jun 10 09:36 os_access.py
-rw-r--r-- 1 dhellman dhellman 1383 May 27 09:23 os_cwd_example.py
-rw-r--r-- 1 dhellman dhellman 1535 Jun 10 09:36 os_directories.py
-rw-r--r-- 1 dhellman dhellman 1613 May 27 09:23 os_environ_example.py
-rw-r--r-- 1 dhellman dhellman 2816 Jun 3 08:34 os_popen_examples.py
-rw-r--r-- 1 dhellman dhellman 1438 May 27 09:23 os_process_id_example.py
-rw-r--r-- 1 dhellman dhellman 1887 May 27 09:23 os_process_user_example.py
-rw-r--r-- 1 dhellman dhellman 1545 Jun 10 09:36 os_stat.py
-rw-r--r-- 1 dhellman dhellman 1638 Jun 10 09:36 os_stat_chmod.py
-rw-r--r-- 1 dhellman dhellman 1452 Jun 10 09:36 os_symlinks.py
-rw-r--r-- 1 dhellman dhellman 1279 Jun 17 12:17 os_system_example.py
-rw-r--r-- 1 dhellman dhellman 1672 Jun 10 09:36 os_walk.py
```

由于命令是直接被传递到处理shell中,所以它可以包含shell语法,比如通配符或环境变量:

```
# Command with shell expansion
os.system('ls -l $HOME')
```

```
total 40
-rwx------ 1 dhellman dhellman 1328 Dec 13 2005 %backup%~
drwx------ 11 dhellman dhellman 374 Jun 17 12:11 Desktop
drwxr-xr-x 15 dhellman dhellman 510 May 27 07:50 Devel
drwx------ 29 dhellman dhellman 986 May 31 17:01 Documents
drwxr-xr-x 45 dhellman dhellman 1530 Jun 17 12:12 DownloadedApps
drwx------ 55 dhellman dhellman 1870 May 22 14:53 Library
drwx------ 8 dhellman dhellman 272 Mar 4 2006 Movies
drwx------ 10 dhellman dhellman 340 Feb 14 10:54 Music
drwx------ 12 dhellman dhellman 408 Jun 17 01:00 Pictures
drwxr-xr-x 5 dhellman dhellman 170 Oct 1 2006 Public
drwxr-xr-x 15 dhellman dhellman 510 May 12 15:19 Sites
drwxr-xr-x 4 dhellman dhellman 136 Jan 23 2006 iPod
-rw-r--r-- 1 dhellman dhellman 105 Mar 7 11:48 pgadmin.log
drwxr-xr-x 3 dhellman dhellman 102 Apr 29 16:32 tmp
```

除非你是直接在后台中运行这命令， 不然的话，直到命令执行完毕，调用 `os.system()` 的程序都会处于阻断状态． 子进程中的标准输入，输出，和错误输出默认被绑定到调用者的合适的流中． 但是也可以通过shell语法重定向到其他地方．

```
import os
import time

print 'Calling...'
os.system('date; (sleep 3; date) &')

print 'Sleeping...'
time.sleep(5)
```

这就是shell的魔力，尽管还有更好的实现方式．

```
$ python os_system_background.py
Calling...
Sun Jun 17 12:27:20 EDT 2007
Sleeping...
Sun Jun 17 12:27:23 EDT 2007
```

12.4 使用os.fork()创建进程

符合POSIX标准的函数 `fork()` 和 `exec*()` (在Mac OS X, Linux和其他类UNIX系统上可用)通过os模块都是可用的． 很多书已经很全面可靠的描述了这些函数的使用，所以检查你的库手册，或者去书店寻找进一步细节．

创建一个新进程作为当前进程的一个复本，可以使用 `os.fork()` ：

```
pid = os.fork()

if pid:
    print 'Child process id:', pid
else:
    print 'I am the child'
```

每次运行这个事例代码时，你的输出变化给予你系统的当前状态， 但是它应该看起来像如下：

```
$ python os_fork_example.py
Child process id: 5883
I am the child
```

当fork之后, 你结束这两个运行着相同代码的进程. 可以检查返回值来直到你在哪个进程中. 如果它是0, 表示你在子进程中,如果不是0, 则表示你在父进程中,它返回的值是其子进程的进程id.

对于父进程来说,发送给子进程信号是有必要的. 这个的设置有点复杂, 使用signal模块, 让我们通过具体代码来描述其使用吧. 首先我们定义一个信号处理句柄, 以便在收到相应信号时调用.

```
import os
import signal
import time

def signal_usr1(signum, frame):
    pid = os.getpid()
    print 'Received USR1 in process %s' % pid
```

然后我们创建子进程, 并在父进程中,通过 os.kill() 发送一个USR1信号之前暂停一段时间.这短的暂停让子进程有时间去设置信号处理句柄.

```
print 'Forking...'
child_pid = os.fork()
if child_pid: ##
    print 'PARENT: Pausing before sending signal...'
    time.sleep(1)
    print 'PARENT: Signaling %s' % child_pid
    os.kill(child_pid, signal.SIGUSR1)
```

在子进程中,我们设置信号处理句柄后睡眠一段时间来让父进程有时间去发送信号给我们:

```
else:
    print 'CHILD: Setting up signal handler'
    signal.signal(signal.SIGUSR1, signal_usr1)
    print 'CHILD: Pausing to wait for signal'
    time.sleep(5)
```

当然, 在实际的程序中,你也可能不需要(不想)调用sleep。

```
$ python os_kill_example.py
Forking...
PARENT: Pausing before sending signal...
CHILD: Setting up signal handler
CHILD: Pausing to wait for signal
PARENT: Signaling 6053
Received USR1 in process 6053
```

正如你所看到的, 一个简单的处理子进程各自行为的方式是简单 fork() 函数的返回值并使用if分支实现. 对于更复杂的行为, 就需要更多的分离(独立)的代码,而不是简单的分支. 在其他的例子中, 你可以使用一个已经封装好的程序. 对于这两种情况,你可以使用 os.exec*() 系列函数来运行其他程序. 当你''exec''一个程序, 程序中的代码会代替你进程中已存在的那些代码.

```
child_pid = os.fork()
if child_pid:
    os.waitpid(child_pid, 0)
else:
    os.execlp('ls', 'ls', '-l', '/tmp/') ##
```

```
$ python os_exec_example.py

total 40
drwxr-xr-x 2 dhellman wheel 68 Jun 17 14:35 527
prw------- 1 root wheel 0 Jun 15 19:24 afpserver_PIPE
drwx------ 3 dhellman wheel 102 Jun 17 12:13 emacs527
drwxr-xr-x 2 dhellman wheel 68 Jun 16 05:01 hsperfdata_dhellmann
-rw------- 1 nobody wheel 12 Jun 17 13:55 objc_sharing_ppc_4294967294
-rw------- 1 dhellman wheel 144 Jun 17 14:32 objc_sharing_ppc_527
-rw------- 1 security wheel 24 Jun 17 07:09 objc_sharing_ppc_92
drwxr-xr-x 4 dhellman dhellman 136 Jun 8 03:16 var_backups
```

有很多 `exec*()` 的变种，它们依赖于你可能使用的参数，如,你是否想要路径和父进程的环境变量都被复制到子进程中,等等. 细节可参见库文档.

对于所有变种，它们的第一个参数是一个路径或者文件名，剩下的参数控制如何运行相应程序. 它们要么作为命令行参数被传递，要么覆盖进程''环境''(可查看 `os.environ` 和 `os.getenv`).

12.5 等待一个子进程

假设说你使用了多个进程来突破Python的线程限制和 `GIL`.如果你开始了多个进程来运行各自的任务，你希望在开始新的进程之前等待其中一个或多个的结束，以此来避免服务器的超载. 这里有一些使用 `wait()` 和相关函数来实现它的不同方法.

如果你不关心，或者你已经知道，哪个可能会首先退出 `os.wait()` 的子进程,并且这个子进程会尽快的返回任何存在:

```
import os
import sys
import time

for i in range(3):
    print 'PARENT: Forking %s' % i
    worker_pid = os.fork()
    if not worker_pid:
        print 'WORKER %s: Starting' % i
        time.sleep(2 + i)
        print 'WORKER %s: Finishing' % i
        sys.exit(i)

 for i in range(3):
     print 'PARENT: Waiting for %s' % i
     done = os.wait()
     print 'PARENT:', done
```

Note: `os.wait()` 的返回值是包含进程号和退出状态(一个16位的数字，它的低字节是一个杀死该进程的信号数字，它的高字节是退出状态)的一个元组.

```
$ python os_wait_example.py

PARENT: Forking 0
PARENT: Forking 1
PARENT: Forking 2
PARENT: Waiting for 0
WORKER 0: Starting
WORKER 1: Starting
WORKER 2: Starting
WORKER 0: Finishing
PARENT: (6501, 0)
```

```
PARENT: Waiting for 1
WORKER 1: Finishing
PARENT: (6502, 256)
PARENT: Waiting for 2
WORKER 2: Finishing
PARENT: (6503, 512)
```

如果你想等待一个特定的进程，可以使用 `os.waitpid()` .

```
import os
import sys
import time

workers = []
for i in range(3):
    print 'PARENT: Forking %s' % i
    worker_pid = os.fork()
    if not worker_pid:
        print 'WORKER %s: Starting' % i
        time.sleep(2 + i)
        print 'WORKER %s: Finishing' % i
        sys.exit(i)
    workers.append(worker_pid)

for pid in workers:
    print 'PARENT: Waiting for %s' % pid
    done = os.waitpid(pid, 0)
    print 'PARENT:', done
```

```
$ python os_waitpid_example.py
PARENT: Forking 0
WORKER 0: Starting
PARENT: Forking 1
WORKER 1: Starting
PARENT: Forking 2
WORKER 2: Starting
PARENT: Waiting for 6547
WORKER 0: Finishing
PARENT: (6547, 0)
PARENT: Waiting for 6548
WORKER 1: Finishing
PARENT: (6548, 256)
PARENT: Waiting for 6549
WORKER 2: Finishing
PARENT: (6549, 512)
```

`wait3()` 和 `wait4()` 函数也是类似的方式，但它们返回更多关于子进程的细节信息,如进程号，退出状态，资源使用情况等.

12.6 Spawn（孵化）

方便起见，`os.spawn*()` 系列函数将 `fork()` 和 `exec*()` 调用写在一条语句中:

```
os.spawnlp(os.P_WAIT, 'ls', 'ls', '-l', '/tmp/')
```

```
$ python os_exec_example.py
total 40
```

```
drwxr-xr-x 2 dhellman wheel 68 Jun 17 14:35 527
prw------- 1 root wheel 0 Jun 15 19:24 afpserver_PIPE
drwx------ 3 dhellman wheel 102 Jun 17 12:13 emacs527
drwxr-xr-x 2 dhellman wheel 68 Jun 16 05:01 hsperfdata_dhellmann
-rw------- 1 nobody wheel 12 Jun 17 13:55 objc_sharing_ppc_4294967294
-rw------- 1 dhellman wheel 144 Jun 17 14:32 objc_sharing_ppc_527
-rw------- 1 security wheel 24 Jun 17 07:09 objc_sharing_ppc_92
drwxr-xr-x 4 dhellman dhellman 136 Jun 8 03:16 var_backups
```

12.7 结论

还有其他很多在处理多进程时需要考虑的东西,比如,信号处理， 多进程文件读写等.所有这些话题都在参考书目(如 Advanced Programming in the UNIX(R) Environment)中有讲述.

12.8 参考

- Delve into UNIX process creation

CHAPTER

THIRTEEN

PYMOTW: PICKLE & CPICKLE

Python对象序列化

- 模块：pickle 和 cPickle
- 目的： Python对象序列化
- python版本：pickle至少1.4，cPickle 至少1.5

13.1 描述

pickle模块可以实现任意的Python对象转换为一系列字节(即序列化对象)的算法。这些字节流可以被传输或存储，接着也可以重构为一个和原先对象具有相同特征的新对象。

cPickle模块实现了同样的算法，但它是用c而不是python。因此，它比python实现的快上好几倍，但是不允许使用者去继承Pickle。如果继承对于你的使用不是很重要，那么你大可以使用cPickle。

Note: pickle的文档清晰的表明它不提供安全保证。所以慎用pickle来作为内部进程通信或者数据存储，也不要相信那些你不能验证安全性的数据。

13.2 例子

第一个pickle示例是将一个数据结构编码为一个字符串，然后将其输出到控制台。

```
try:
   import cPickle as pickle
except:
   import pickle
   import pprint
```

我们首先尝试导入cPickle，并指定别名为''pickle''。如果因为某种原因导入pickle失败，我们则导入由Python实现的pickle模块。如果cPickle是可用的，会给我们带来更快的实现，但如果不可用，也会有正确的实现。

接下来，我们定义一个完全由基本类型组成的数据结构。任何类的实例都可被pickle，这会在下一个例子中表述。我选择基本数据类型以便更简单的示范。

```
data = [ { 'a':'A', 'b':2, 'c':3.0 } ]
print 'DATA:',
pprint.pprint(data)
```

现在我们就可以使用pickle.dumps()来创建数据值的字符串表示。

```
data_string = pickle.dumps(data)
print 'PICKLE:', data_string
```

默认情况下，pickle仅使用ASCII字符。也可以使用高效的二进制格式。但这些示例依然使用了ASCII格式 。

```
$ python pickle_string.py
DATA:[{'a': 'A', 'b': 2, 'c': 3.0}]
PICKLE: (lp1
(dp2
S'a'
S'A'
sS'c'
F3
sS'b'
I2
sa.
```

一旦数据被序列化，你就可以把他写入到文件、socket、管道等等中。之后你可以读取这个文件，unpickle这些数据来构造具有相同值的新对象。

```
data1 = [ { 'a':'A', 'b':2, 'c':3.0 } ]
print 'BEFORE:',
pprint.pprint(data1)

data1_string = pickle.dumps(data1)

data2 = pickle.loads(data1_string)
print 'AFTER:',
pprint.pprint(data2)

print 'SAME?:', (data1 is data2)
print 'EQUAL?:', (data1 == data2)
```

正像你看到的那样，新构造的对象等于原来的对象，但他们又不是相同的对象。这里不足为奇。

```
$ python pickle_unpickle.py
BEFORE:[{'a': 'A', 'b': 2, 'c': 3.0}]
AFTER:[{'a': 'A', 'b': 2, 'c': 3.0}]
SAME?: False
EQUAL?: True
```

pickle除了提供dumps()和loads()，还提供非常方便的函数用于操作类文件流。支持同时写多个对象到同一个流中，然后在不知道有多少个对象或不知道它们有多大时，能够从这个流中读取多个对象也是可能的，

```
try:
   import cPickle as pickle
except:
   import pickle
import pprint
from StringIO import StringIO

class SimpleObject(object):

   def __init__(self, name):
      self.name = name
      l = list(name)
      l.reverse()
      self.name_backwards = ''.join(l)
      return

data = []
data.append(SimpleObject('pickle'))
```

```
data.append(SimpleObject('cPickle'))
data.append(SimpleObject('last'))

# Simulate a file with StringIO
out_s = StringIO()

# Write to the stream
for o in data:
   print 'WRITING: %s (%s)' % (o.name, o.name_backwards)
   pickle.dump(o, out_s)
   out_s.flush()

# Set up a read-able stream
in_s = StringIO(out_s.getvalue())

# Read the data
while True:
   try:
      o = pickle.load(in_s)
   except EOFError:
      break
   else:
      print 'READ: %s (%s)' % (o.name, o.name_backwards)
```

这个例子使用StringIO缓冲区来模拟流，因此我们在建立可读流时得玩点小花样。一个简单数据库格式也可以使用pickle来存储对象，虽然使用shelve模块可能会更简单。

```
$ python pickle_stream.py
WRITING: pickle (elkcip)
WRITING: cPickle (elkciPc)
WRITING: last (tsal)
READ: pickle (elkcip)
READ: cPickle (elkciPc)
READ: last (tsal)
```

除了用于存储数据，pickle在用于内部进程通信时是非常灵活的。比如，使用os.fork()和os.pipe()，可以建立一些工作进程，它们从一个管道中读取任务说明并把结果输出到另一个管道。操作这些工作池、发送任务和接受反应的核心代码可以重复利用，因为任务和反应对象不是一个特殊的类。如果你使用管道或者sockets，就不要忘记在dump每个对象后刷新它们并通过其间的连接将数据推送到另外一个进程。

在处理自定义类时，你应该保证这些被pickled的类会在进程名空间出现 。只有数据实例才能被pickle，而不能是定义的类。在unpickle时，类的名字被用于寻找构造器以便创建新对象。接下来这个例子，是将一个类实例写入到文件中：

```
try:
   import cPickle as pickle
except:
   import pickle
   import sys

class SimpleObject(object):

   def __init__(self, name):
      self.name = name
      l = list(name)
      l.reverse()
      self.name_backwards = ''.join(l)
      return

if __name__ == '__main__':
```

```
   data = []
   data.append(SimpleObject('pickle'))
   data.append(SimpleObject('cPickle'))
   data.append(SimpleObject('last'))
   try:
      filename = sys.argv[1]
      except IndexError:
      raise RuntimeError('Please specify a filename as an argument to %s' % sys.argv[0])
   out_s = open(filename, 'wb')
   try:
      # Write to the stream
      for o in data:
         print 'WRITING: %s (%s)' % (o.name, o.name_backwards)
         pickle.dump(o, out_s)
   finally:
      out_s.close()
```

当我运行这个脚本时，它会创建名为我在命令行中输入的参数的文件：

```
$ python pickle_dump_to_file_1.py test.dat
WRITING: pickle (elkcip)
WRITING: cPickle (elkciPc)
WRITING: last (tsal)
```

一个简单的尝试将刚才的pickle结果对象装载进来可以是如下的样子：

```
try:
   import cPickle as pickle
except:
   import pickle
import pprint
from StringIO import StringIO
import sys

try:
   filename = sys.argv[1]
except IndexError:
   raise RuntimeError('Please specify a filename as an argument to %s' % sys.argv[0])

in_s = open(filename, 'rb')
try:
   # Read the data
   while True:
      try:
         o = pickle.load(in_s)
      except EOFError:
         break
      else:
         print 'READ: %s (%s)' % (o.name, o.name_backwards)
finally:
   in_s.close()
```

这个版本失败了，因为这里没有可用的SimpleObject类

```
$ python pickle_load_from_file_1.py test.dat
Traceback (most recent call last):
File "pickle_load_from_file_1.py", line 52, in
   o = pickle.load(in_s)
AttributeError: 'module' object has no attribute 'SimpleObject'
```

一个正确版本，它从pickle_dump_to_file_1导入了SimpleObject类，可以成功运行。 增加：

```
from pickle_dump_to_file_1 import SimpleObject
```

到导入列表的最后，然后运行这个脚本：

```
$ python pickle_load_from_file_2.py test.dat
READ: pickle (elkcip)
READ: cPickle (elkciPc)
READ: last (tsal)
```

在pickle那些不能被pickle的数据(如sockets、文件句柄、数据库连接等等)时，需要考虑一些特殊之处。那些不能被pickle的类可以定义 `__getstate__()` 和 `__setstate__()` 来返回实例在被pickle时的状态。新风格的类也可以定义`__getnewargs__()`，它返回传递给类内存分配者(`C.__new__()`)的参数。关于这些特征的更详细的使用描述可以在标准库文档中找到。

13.3 参考

- Pickle: An interesting stack language by Alexandre Vassalotti

CHAPTER

FOURTEEN

PYMOTW: GLOB

- 模块： glob
- 目的： 使用Unix Shell规则来寻找文件名匹配某一模式的文件。
- python版本： 1.4+

14.1 描述

即使glob API非常简单，但这个模块具有强大的力量。在很多情况下，尤其是你的程序需要寻找出文件系统中，文件名匹配特定模式的文件时，是非常有用的。如果你需要包含一个特定扩展名，或前缀，或含有任何普通字符串的文件列表，可以直接使用glob代替手工编程扫描目录内容。

glob中模式规则不是正则表达式，而是，符合标准Uinx路径扩展规则。但是Shell变量名和符号(~)是不被扩充的，只有一些特殊的字符：两个不同的通配符和字母范围被支持。模块规则适合于文件名的片段(以/为分隔)，但模式中的路径可以是相对或者绝对路径。

14.2 示例数据

假设当前工作目录下包含有以下一些文件。

```
dir/
dir/file.txt
dir/file1.txt
dir/file2.txt
dir/filea.txt
dir/fileb.txt
dir/subdir/
dir/subdir/subfile.txt
```

使用glob_maketestdata.py脚本可以创建这些文件。

14.3 通配符

* 匹配名字片段中的0个或多个字符，例如，dir/* 。

```
import glob
print glob.glob('dir/*')
```

这个模式匹配在目录dir中的任何文件或子目录，但没有进一步递归匹配子目录。

```
$ python glob_asterisk.py
['dir/file.txt', 'dir/file1.txt', 'dir/file2.txt',
'dir/filea.txt', 'dir/fileb.txt', 'dir/subdir']
```

如果要列出子目录中的文件，你应该在模式中包含相应子目录：

```
print 'Named explicitly:'
print glob.glob('dir/subdir/*')

print 'Named with wildcard:'
print glob.glob('dir/*/*')
```

上面的第一个例子直接列出了指定子目录名的文件，而第二个例子则依赖于通配符来寻找子目录。

```
$ python glob_subdir.py
Named explicitly:
['dir/subdir/subfile.txt']
Named with wildcard:
['dir/subdir/subfile.txt']
```

在这个例子中，结果是一样的。如果还有其他的子目录，那么，通配符匹配所有子目录及其他们中包含的文件。

14.4 单一字符通配符

其他的被支持的通配符是问号(?)。它匹配在对应位置的任一单个字符。例如：

```
print glob.glob('dir/file?.txt')
```

匹配所有以''file''开头，之后包含一个任何字符并以''.txt''结尾的文件。

```
$ python glob_question.py
['dir/file1.txt', 'dir/file2.txt',
'dir/filea.txt', 'dir/fileb.txt']
```

14.5 字符范围

当你需要匹配一个特定字符时，可以使用一个字符范围来替代问号。例如，为了找到所有文件名中在扩展名之前包含数字的文件时：

```
print glob.glob('dir/*[0-9].*')
```

字符范围[0-9]匹配任何单一数字。这个范围是基于每个字符/数字的字符编码顺序，破折号(-)表示一个范围。上面的范围也可直接用[0123456789]来表示。

```
$ python glob_charrange.py
['dir/file1.txt', 'dir/file2.txt']
```

14.6 参考

- Pattern Matching Notation

CHAPTER

FIFTEEN

PYMOTW: SHELVE

- 模块：shelve
- 目的：shelve模块实现了对任意可被pickle的Python对象进行持久性存储，也提供类字典API给我们使用。
- python版本：1.4+

15.1 描述

当使用关系数据库是一种浪费的时候，shelve模块可以为Python对象提供一个简单的持久性存储选择。就像使用字典一样，通过关键字访问shelf对象。其值经过pickle，写入到由anydbm创建和管理的数据库。

15.2 创建一Shelf对象

最简单的使用shelve模块的方式是通过DbfilenameShelf类。 使用函数 `shelve.open()` （使用的是 `anydbm` ）来存储数据。你可以直接使用类，或者简单的调用：

```
import shelve

s = shelve.open('test_shelf.db')
try:
    s['key1'] = { 'int': 10, 'float':9.5, 'string':'Sample data' }
finally:
    s.close()
```

为了再次访问数据，打开shelf并且像字典一样使用它。

```
s = shelve.open('test_shelf.db')
try:
    existing = s['key1']
finally:
    s.close()

print existing
```

如果你运行了上面两个脚本，你应该看到：

```
$ python shelve_create.py
$ python shelve_existing.py
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

dbm模块不支持多个应用同时写入同一数据库。如果你确定客户端不会修改shelf， 请指定shelve以只读方式打开数据库。

```
s = shelve.open('test_shelf.db', flag='r')
try:
    existing = s['key1']
finally:
    s.close()

print existing
```

当数据库以只读方式打开时，你又尝试着更改数据库，这将引起一个访问出错异常。这一异常类型依赖于在创建数据库时被anydbm选择的数据库模块。

15.3 写回

默认情况下，Shelves不去追踪可变对象的修改。意思就是，如果你改变了已存储在shelf中的一个项目的内容，就必须重新存储该项目来更新shelf。

```
s = shelve.open('test_shelf.db')
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
finally:
    s.close()
```

在这个例子中，没有对字典里的关键字''key1''的内容进行存储，因此，重新打开shelf的时候，还没保存所做的改变。

```
$ python shelve_create.py
$ python shelve_withoutwriteback.py
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

为了自动捕捉储存在shelf中的可变对象所发生的改变，置writeback功能可用。writeback标志导致shelf使用一缓存来记住从数据库中调出的所有对象 。当shelf关闭的时候，每一个缓存中的对象也重新写入数据库。

```
s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
    print s['key1']
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
finally:
    s.close()
```

虽然使用writeback模式可以减少程序员出错机率，也能更加透明化对象持久性，但是，不是每种情况都要使用writeback 模式的。当shelf打开的时候，缓存就要占据额外的空间，并且，当shelf关闭的时候，也会花费额外的时间去将所有缓存中的对象写入到数据库中。即使不知道缓存中的对象有没有改变，都要写回数据库。如果你的应用程序读取数据多于写入数据，那么writeback模式将增大开销。

```
$ python shelve_create.py
$ python shelve_writeback.py
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
```

15.4 指定Shelf类型

上面的例子全都使用了默认的shelf实现。使用 `shelve.open()` 直接代替一种shelf实现，是常见用法，尤其是在不关心用哪种数据库存储数据的时候。然而，有时常会关心它。如果是在这种情况下，通常就会直接使用DbfilenameShelf或者BsdDbShelf ，更或者是通过Shelf的子类来解决问题。

15.5 参考

- feedcache uses shelve as a default storage option

CHAPTER

SIXTEEN

PYMOTW: OPTPARSE

- 模块：optparse
- 目的：命令行参数解析，可以取代getopt
- python版本：2.3

16.1 描述

optparse是一个当前可选的命令行解析模块，它提供了一些在getopt中不含有的特性，如type conversion(类型转换)，option callbacks(参数回调)以及automatice help generation(自动化帮助生成)。本文没有详细介绍optparse的很多特性，但它可以帮助你在写命令行程序时能够快速入门。

16.2 创建一个OptionParser

optparser解析参数需要经过2个阶段。首先，构建OptionParser实例并配置相关的选项，然后填入一个参数序列并处理。

```
import optparse
parser = optparse.OptionParser()
```

通常，一旦分析器被建立，每一个选项需要明确的添加到parser中，并说明当命令行遇到相关的选项时需要如何处理。在构建OptionParser时也可以传入一个选项列表，但这种形式不经常使用。

16.3 定义选项

利用add_option()方法可以每次增加一个选项。在参数列表的开始，任何未命名的字符串参数都将被视为选项名。如果要为一个选项创建别名，比如为同一个选项增加一个短的或长的命名，那么简单传递同名字符串即可。

不同于getopt，只能分析选项，optparse是一个完整的选项分析库，Option(选项)可以被不同的方法处理，通过在add_option()方法中指定action(行为)参数。支持的行为包括存储参数(单独或作为列表的一部分)，当一个选项出现时(包括对布尔开关true/false的特殊处理)存储其常量值，计算一个选项出现的次数以及调用一个callback(回调函数)。

默认的行为是存储这个选项的参数。如果给定了type(类型)，那么在存储前，这个参数值将被转化成这个类型。如果给定了dest(目标参数)，那么当命令行参数被解析时，选项值被存储在该选项对象的dest中。

16.4 分析一个命令行

一旦所有的选项被定义好，命令行被作为一个参数字符串传递给parse_args()方法。一般，参数可以从sys.argv[1:]中得到，当然你可以传递自己的列表。选项处理时使用GNU/POSIX语法，因此，选项和参数值可以在参数序列中混合使用。

从parse_args()方法返回的是一个二维元组，包含一个optparse Values实例和在命令行中未被解析的参数列表。Values实例将选项值作为属性，如果你定义了一个选项的dest为“myoption”，可以通过 option.myoption访问该选项的值。

16.5 简单示例

如下一个简单例子有三个不同的选项，一个布尔选项(-a)，一个字符串选项(-b)和一个整型选项(-c)。

```
import optparse
parser = optparse.OptionParser()
parser.add_option('-a', action="store_true", default=False)
parser.add_option('-b', action="store", dest="b")
parser.add_option('-c', action="store", dest="c", type="int")
print parser.parse_args(['-a', '-bval', '-c', '3'])
```

命令行中选项解析的规则和getopt.gnu_getopt()一样，因此有两种方法传递单字符串选项的值，上述示例使用了两种方法-bval和-c val

```
$ python optparse_short.py
(<Values at 0xe29b8: {'a': True, 'c': 3, 'b': 'val'}>, [])
```

注意，c所关联的值的类型是整型，OptionParser在存储之前会转换成指定类型。不同于getopt，optparse处理长选项名时和短选项名是没有任何区别的。

```
parser = optparse.OptionParser()
parser.add_option('--noarg', action="store_true", default=False)
parser.add_option('--witharg', action="store", dest="witharg")
parser.add_option('--witharg2', action="store", dest="witharg2", type="int")
print parser.parse_args([ '--noarg', '--witharg', 'val', '--witharg2=3' ])
```

结果相同的：

```
$ python optparse_long.py
(<Values at 0xd3ad0: {'noarg': True, 'witharg': 'val', 'witharg2': 3}>, [])
```

16.6 与getopt的比较

如下实现一个与getopt之前示例相同功能的optparse例子

```
import optparse
import sys
print 'ARGV      :', sys.argv[1:]
parser = optparse.OptionParser()
parser.add_option('-o', '--output',
            dest="output_filename",
            default="default.out",
            )
parser.add_option('-v', '--verbose',
```

```
            dest="verbose",
            default=False,
            action="store_true",
            )
parser.add_option('--version',
            dest="version",
            default=1.0,
            type="float",
            )
options, remainder = parser.parse_args()
print 'VERSION   :', options.version
print 'VERBOSE   :', options.verbose
print 'OUTPUT    :', options.output_filename
print 'REMAINING :', remainder
```

注意，-o和--output选项是如何在同一时刻被定义的，命令行中可以使用任何一种选项。

```
$ python optparse_getoptcomparison.py -o output.txt
ARGV      : ['-o', 'output.txt']
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : output.txt
REMAINING : []
$ python optparse_getoptcomparison.py --output output.txt
ARGV      : ['--output', 'output.txt']
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : output.txt
REMAINING : []
```

另外，长选项名的唯一前缀也可以被使用。

```
$ python optparse_getoptcomparison.py --out output.txt
ARGV      : ['--out', 'output.txt']
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : output.txt
REMAINING : []
```

16.7 Option Callbacks(选项回调)

除了直接为选项存储参数，另一种选择是定义callback function，当命令行中出现该选项时调用，选项的callbacks有4个参数，分别是引起callback的optparse.Option实例，命令行中的选项字符串，选项关联的参数值以及处理解析工作的optparse.OptionParser实例。

```
import optparse

def flag_callback(option, opt_str, value, parser):
    print 'flag_callback:'
    print '\toption:', repr(option)
    print '\topt_str:', opt_str
    print '\tvalue:', value
    print '\tparser:', parser
    return

def with_callback(option, opt_str, value, parser):
    print 'with_callback:'
    print '\toption:', repr(option)
```

```
    print '\topt_str:', opt_str
    print '\tvalue:', value
    print '\tparser:', parser
    return

parser = optparse.OptionParser()
parser.add_option('--flag', action="callback", callback=flag_callback)
parser.add_option('--with',
    action="callback",
    callback=with_callback,
    type="string",
    help="Include optional feature")
parser.parse_args(['--with', 'foo', '--flag'])
```

在这个例子中，--with选项被配置成处理字符串参数(当然，其他类型也是同样支持的)。

```
$ python optparse_callback.py
with_callback:
     option: <Option at 0x78b98: --with>
     opt_str: --with
     value: foo
     parser: <optparse.OptionParser instance at 0x78b48>
flag_callback:
     option: <Option at 0x7c620: --flag>
     opt_str: --flag
     value: None
     parser: <optparse.OptionParser instance at 0x78b48>
```

16.8 帮助信息

OptionParser自动为每个选项集合包含一个help选项，因此，用户在运行程序时在命令行输入--help来看介绍，帮助信息为所有选项指示它们是否需要传入一个参数，也可以通过在add_option()中定义帮助文本来为一个选项定义更多的描述。

```
parser = optparse.OptionParser()
parser.add_option('--no-foo', action="store_true",
    default=False,
    dest="foo",
    help="Turn off foo",
)
parser.add_option('--with', action="store", help="Include optional feature")
parser.parse_args()
```

选项按字母顺序显示，别名显示在同一行，当选项带有一个参数时，dest值将作为参数名字出现在help输出中，帮助信息将出现在这列中。

```
$ python optparse_help.py --help
Usage: optparse_help.py [options]

Options:
-h, --help   show this help message and exit
--no-foo     Turn off foo
--with=WITH  Include optional feature
```

利用nargs选项可以配置callbacks接收多个参数。

```
def with_callback(option, opt_str, value, parser):
    print 'with_callback:'
    print '\toption:', repr(option)
    print '\topt_str:', opt_str
    print '\tvalue:', value
    print '\tparser:', parser
    return

parser = optparse.OptionParser()
parser.add_option('--with',
    action="callback",
    callback=with_callback,
    type="string",
    nargs=2,
    help="Include optional feature")
parser.parse_args(['--with', 'foo', 'bar'])
```

在这个例子中，参数作为一个元组传递给callback function的value参数。

```
$ python optparse_callback_nargs.py
with_callback:
     option: <Option at 0x7c4e0: --with>
     opt_str: --with
     value: ('foo', 'bar')
     parser: <optparse.OptionParser instance at 0x78a08>
```

CHAPTER

SEVENTEEN

PYMOTW: SHUTIL

- 模块：shutil
- 目的： 高层次的文件操作.
- python版本：1.4+

shutil模块提供了一些高层次的文件操作,比如复制,设置权限等等.

17.1 描述:

shutil模块提供了一些用于复制和删除整个文件的函数.

17.2 复制文件:

copyfile() 将源文件内容完全复制给目标文件.如果没有写入目标文件的权限,会引起IOError.由于该函数是为了读取文件内容而打开此输入文件,而不管它的类型是什么,特殊类型的文件使用copyfile()是不能拷贝的，比如管道文件。

```
import os
from shutil import *

print 'BEFORE:', os.listdir(os.getcwd())
copyfile('shutil_copyfile.py', 'shutil_copyfile.py.copy')
print 'AFTER:', os.listdir(os.getcwd())
```

```
$ python shutil_copyfile.py
BEFORE: ['__init__.py', 'shutil_copyfile.py']
AFTER: ['__init__.py', 'shutil_copyfile.py', 'shutil_copyfile.py.copy']
```

copyfile()底层调用了copyfileobj()函数.文件名参数传递给copyfile()后,进而将此文件句柄传递给copyfileobj().第三个可选参数是一个缓冲区长度,以块读入(默认情况下,一次性读取整个文件).

```
import os
from StringIO import StringIO
import sys
from shutil import *

class VerboseStringIO(StringIO):
    def read(self, n=-1):
        next = StringIO.read(self, n)
        print 'read(%d) =>' % n, next
        return next
```

```
lorem_ipsum = '''Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
              Vestibulum aliquam mollis dolor. Donec vulputate nunc ut diam.
              Ut rutrum mi vel sem. Vestibulum ante ipsum.'''

print 'Default:'
input = VerboseStringIO(lorem_ipsum)
output = StringIO()
copyfileobj(input, output)

print

print 'All at once:'
input = VerboseStringIO(lorem_ipsum)
output = StringIO()
copyfileobj(input, output, -1)

print

print 'Blocks of 20:'
input = VerboseStringIO(lorem_ipsum)
output = StringIO()
copyfileobj(input, output, 20)
```

默认的行为是以大块读取:

```
$ python shutil_copyfileobj.py
Default:
read(16384) => Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Vestibulum aliquam mollis dolor. Donec vulputate nunc ut diam.
Ut rutrum mi vel sem. Vestibulum ante ipsum.
read(16384) =>
```

使用-1表示一次性读取所有输入:

```
All at once:
read(-1) => Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Vestibulum aliquam mollis dolor. Donec vulputate nunc ut diam.
Ut rutrum mi vel sem. Vestibulum ante ipsum.
read(-1) =>
```

使用一个正整数设置块大小:

```
Blocks of 20:
read(20) => Lorem ipsum dolor si
read(20) => t amet, consectetuer
read(20) =>  adipiscing elit.
V
read(20) => estibulum aliquam mo
read(20) => llis dolor. Donec vu
read(20) => lputate nunc ut diam
read(20) => .
Ut rutrum mi vel
read(20) => sem. Vestibulum ante
read(20) =>  ipsum.
read(20) =>
```

copy()函数类似于Unix命令cp.如果目标参数是一个目录而不是一个文件,那么在这个目录中复制一个源文件副本(它与源文件同名).文件的权限也随之复制.

```
import os
from shutil import *

os.mkdir('example')
print 'BEFORE:', os.listdir('example')
copy('shutil_copy.py', 'example')
print 'AFTER:', os.listdir('example')
```

```
$ python shutil_copy.py
BEFORE: []
AFTER: ['shutil_copy.py']
```

copy2()函数类似于copy(),但是它将一些元信息,如文件最后一次被读取时间和修改时间等,也复制至新文件中.

```
import os
from shutil import *

def show_file_info(filename):
    stat_info = os.stat(filename)
    print '\tMode    :', stat_info.st_mode
    print '\tCreated :', time.ctime(stat_info.st_ctime)
    print '\tAccessed:', time.ctime(stat_info.st_atime)
    print '\tModified:', time.ctime(stat_info.st_mtime)

os.mkdir('example')
print 'SOURCE:'
show_file_info('shutil_copy2.py')
copy2('shutil_copy2.py', 'example')
print 'DEST:'
show_file_info('example/shutil_copy2.py')
```

```
$ python shutil_copy2.py

SOURCE:
        Mode    : 33188
        Created : Sun Oct 21 15:16:07 2007
        Accessed: Sun Oct 21 15:16:11 2007
        Modified: Sun Oct 21 15:16:07 2007
        DEST:
        Mode    : 33188
        Created : Sun Oct 21 15:16:11 2007
        Accessed: Sun Oct 21 15:16:11 2007
        Modified: Sun Oct 21 15:16:07 2007
```

17.3 复制文件元信息:

默认情况下,在Unix下,一个新创建的文件的权限会根据当前用户的umask值来设置.把一个文件的权限复制给另一个文件，可以使用copymode()函数.

```
from commands import *
from shutil import *

print 'BEFORE:', getstatus('file_to_change.txt')
copymode('shutil_copymode.py', 'file_to_change.txt')
print 'AFTER :', getstatus('file_to_change.txt')
```

首先,需要创建一个文件.然后对权限做些修改:

```
$ touch file_to_change.txt
$ chmod ugo+w file_to_change.txt
```

然后,运行刚才的示例脚本会改变之前的权限:

```
$ python shutil_copymode.py
BEFORE: -rw-rw-rw-   1 dhellman  dhellman  0 Oct 21 14:43 file_to_change.txt
AFTER : -rw-r--r--   1 dhellman  dhellman  0 Oct 21 14:43 file_to_change.txt
```

复制文件的其他元信息(权限,最后读取时间,最后修改时间)可以使用copystat().

```
import os
from shutil import *
import time

def show_file_info(filename):
    stat_info = os.stat(filename)
    print '\tMode    :', stat_info.st_mode
    print '\tCreated :', time.ctime(stat_info.st_ctime)
    print '\tAccessed:', time.ctime(stat_info.st_atime)
    print '\tModified:', time.ctime(stat_info.st_mtime)

print 'BEFORE:'
show_file_info('file_to_change.txt')
copystat('shutil_copystat.py', 'file_to_change.txt')
print 'AFTER :'
show_file_info('file_to_change.txt')
```

```
$ python shutil_copystat.py
BEFORE:
        Mode    : 33206
        Created : Sun Oct 21 15:01:23 2007
        Accessed: Sun Oct 21 14:43:26 2007
        Modified: Sun Oct 21 14:43:26 2007
        AFTER :
        Mode    : 33188
        Created : Sun Oct 21 15:01:44 2007
        Accessed: Sun Oct 21 15:01:43 2007
        Modified: Sun Oct 21 15:01:39 2007
```

17.4 目录树:

shutil模块包含3个操作目录树的函数.使用copytree()来复制目录,它会递归复制整个目录结构.目标目录必须不存在.其中, symlinks参数控制符号链接是否作为链接或文件被复制,默认是将其内容复制成一个新文件.如果此选项为true,新的链接会在目标目录树中创建.

Note: 注意:copytree()文档中说,它一般作为一个样本实现,而不是一个工具.你可以修改其源码,让它变得更稳定,或者增加一些功能,比如说进度条.

```
from commands import *
from shutil import *

print 'BEFORE:'
print getoutput('ls -rlast /tmp/example')
copytree('example', '/tmp/example')
```

```
print 'AFTER:'
print getoutput('ls -rlast /tmp/example')
```

```
$ python shutil_copytree.py
BEFORE:
ls: /tmp/example: No such file or directory
AFTER:
total 8
8 -rw-r--r--    1 dhellman  wheel  1627 Oct 21 15:16 shutil_copy2.py
0 drwxr-xr-x    3 dhellman  wheel   102 Oct 21 15:16 .
0 drwxrwxrwt   18 root      wheel   612 Oct 21 15:26 ..
```

使用rmtree()可以删除整个目录树.里面若产生错误会作为异常抛出.但是如果它的第二个参数是目录树,那么错误会被忽略,第三个参数可以指定为一个特殊出错处理函数句柄.

```
from commands import *
from shutil import *

print 'BEFORE:'
print getoutput('ls -rlast /tmp/example')
rmtree('example', '/tmp/example')
print 'AFTER:'
print getoutput('ls -rlast /tmp/example')
```

```
$ python shutil_rmtree.py

BEFORE:
total 8
8 -rw-r--r--    1 dhellman  wheel  1627 Oct 21 15:16 shutil_copy2.py
0 drwxr-xr-x    3 dhellman  wheel   102 Oct 21 15:16 .
0 drwxrwxrwt   18 root      wheel   612 Oct 21 15:26 ..
AFTER:
ls: /tmp/example: No such file or directory
```

移动文件或目录可以使用move(),这很类似于Unix命令mv.如果源文件或目录和目标文件或目录在同一个文件系统下,那么源文件或目录会直接重命名.否则源文件或目录会复制到目标文件或目录,接着删除源文件或目录.

```
import os
from shutil import *

print 'BEFORE: example : ', os.listdir('example')
move('example', 'example2')
print 'AFTER : example2: ', os.listdir('example2')
```

```
$ python shutil_move.py
BEFORE: example :  ['shutil_copy.py']
AFTER : example2:  ['shutil_copy.py']
```

17.5 参考

- PyMOTW

CHAPTER

EIGHTEEN

PYMOTW: URLPARSE

urlparse模块提供了切分统一资源定位符(URL)的接口。

- 模块： urlparse
- 目的： 将URL切分为几个组成部分。
- python版本： 1.4+

18.1 描述

urlparse模块提供了一些将URL切分成对应与RFC中定义的组成部分的函数。

18.2 Parsing:分解

urlparse函数返回的值是一个包含6个元素的类似于元组的对象。

```
from urlparse import urlparse
parsed = urlparse('http://netloc/path;parameters?query=argument#fragment')
print parsed
```

通过借口可以获得URL的各部分组成，网络位置，路径，参数，查询变量和剩余的部分。在下面的例子中，使用http语法来代替''scheme''。

```
$ python urlparse_urlparse.py
('http', 'netloc', '/path', 'parameters', 'query=argument', 'fragment')
```

虽然返回的值是类似于元组，但它不是元组的子类，它支持通过属性名字而不是索引来获取URL对应的部分。这在当你不记得索引顺序时是非常有用的。除了让程序员更容易的使用外，这些属性API还可以获得一些在元组API中不可用的值。

```
from urlparse import urlparse
parsed = urlparse('http://user:pass@NetLoc:80/path;parameters?query=argument#fragment')
print 'scheme :', parsed.scheme
print 'netloc :', parsed.netloc
print 'path :', parsed.path
print 'params :', parsed.params
print 'query :', parsed.query
print 'fragment:', parsed.fragment
print 'username:', parsed.username
print 'password:', parsed.password
print 'hostname:', parsed.hostname, '(netloc in lower case)'
print 'port :', parsed.port
```

当URL中有包含用户名和密码时username和password就会有对应值，如果没有出现则为空。hostname和netloc的值是一样的，但所有字符被强制转换为小写字母。如果出现端口的话，port值被转换成一个对应的整数值，如果没有出现则为None。

```
$ python urlparse_urlparseattrs.py
scheme : http
netloc : user:pass@NetLoc:80
path : /path
params : parameters
query : query=argument
fragment: fragment
username: user
password: pass
hostname: netloc (netloc in lower case)
port : 80
```

urlsplit() 函数是 urlparse() 的一种替代函数。它不将参数从URL中切分出来。这对于遵循RFC 2396标准的URL，即支持每段路径中包含参数时，是很有用的。

```
from urlparse import urlsplit
parsed = urlsplit('http://user:pass@NetLoc:80/path;parameters/path2;parameters2?query=argument#fragment')
print parsed
print 'scheme :', parsed.scheme
print 'netloc :', parsed.netloc
print 'path :', parsed.path
print 'query :', parsed.query
print 'fragment:', parsed.fragment
print 'username:', parsed.username
print 'password:', parsed.password
print 'hostname:', parsed.hostname, '(netloc in lower case)'
print 'port :', parsed.port
```

因为参数没有被切分出来，所以返回的元组只有5个元素而不是6个，因此也没有params属性。

```
$ python urlparse_urlsplit.py
('http', 'user:pass@NetLoc:80', '/path;parameters/path2;parameters2', 'query=argument', 'fragment')
scheme : http
netloc : user:pass@NetLoc:80
path : /path;parameters/path2;parameters2
query : query=argument
fragment: fragment
username: user
password: pass
hostname: netloc (netloc in lower case)
port : 80
```

为了简单的从URL中获得fragment标识符，因为你有可能需要寻找URL指向的页面中特定的位置名称，可以使用urldefrag。

```
from urlparse import urldefrag
original = 'http://netloc/path;parameters?query=argument#fragment'
print original
url, fragment = urldefrag(original)
print url
print fragment
```

返回的是包含基本URL和片段的元组。

```
$ python urlparse_urldefrag.py
http://netloc/path;parameters?query=argument#fragment
```

```
http://netloc/path;parameters?query=argument
fragment
```

18.3 组装

有很多方法可以将URL的各个部分组合回原来的单个字符串。被解析后的URL对象有一个叫做 `geturl()` 的方法。

```
from urlparse import urlparse
original = 'http://netloc/path;parameters?query=argument#fragment'
print 'ORIG :', original
parsed = urlparse(original)
print 'PARSED:', parsed.geturl()
```

当然，只有urlparse或者urlsplit返回的对象才起作用。

```
$ python urlparse_geturl.py
ORIG : http://netloc/path;parameters?query=argument#fragment
PARSED: http://netloc/path;parameters?query=argument#fragment
```

如果你有一个元组，可以使用 `urlunparse()` 将它组装成URL。

```
from urlparse import urlparse, urlunparse
original = 'http://netloc/path;parameters?query=argument#fragment'
print 'ORIG :', original
parsed = urlparse(original)
print 'PARSED:', type(parsed), parsed
t = parsed[:]
print 'TUPLE :', type(t), t
print 'NEW :', urlunparse(t)
```

urlparse返回的ParseResult可以当元组用，如上面的例子中，创建了一个新的元组，并且urlunparse也可以处理一般的元组。

```
$ python urlparse_urlunparse.py
ORIG : http://netloc/path;parameters?query=argument#fragment
PARSED: <class 'urlparse.ParseResult'> ('http', 'netloc', '/path', 'parameters', 'query=argument', 'fragment')
TUPLE : <type 'tuple'> ('http', 'netloc', '/path', 'parameters', 'query=argument', 'fragment')
NEW : http://netloc/path;parameters?query=argument#fragment
```

如果输入的URL包含多余的部分，那么，这些部分可能会被丢弃。

```
from urlparse import urlparse, urlunparse
original = 'http://netloc/path;?#'
print 'ORIG :', original
parsed = urlparse(original)
print 'PARSED:', type(parsed), parsed
t = parsed[:]
print 'TUPLE :', type(t), t
print 'NEW :', urlunparse(t)
```

在这个例子中，原始的URL中缺少参数，查询，片段。之后产生新的URL可能和原来的不一样，但是两者是等价的。

```
$ python urlparse_urlunparseextra.py
ORIG : http://netloc/path;?#
PARSED: <class 'urlparse.ParseResult'> ('http', 'netloc', '/path', '', '', '')
TUPLE : <type 'tuple'> ('http', 'netloc', '/path', '', '', '')
NEW : http://netloc/path
```

18.4 连接

除了解析URL之外，urlparse模块包含 `urljoin()` 函数。用来从关联片段中构造绝对URL。

```
from urlparse import urljoin
print urljoin('http://www.example.com/path/file.html', 'anotherfile.html')
print urljoin('http://www.example.com/path/file.html', '../anotherfile.html')
```

Note: 相对路径(``../''`)被作为第二URL来计算。

```
$ python urlparse_urljoin.py
http://www.example.com/path/anotherfile.html
http://www.example.com/anotherfile.html
```

18.5 参考

- RFC 1378
- RFC 2396

CHAPTER

NINETEEN

PYMOTW: OS.PATH

- 模块：os.path
- 目的：对文件名和路径进行解析，创建，测试和其他操作.
- python版本：1.4+

19.1 描述

我们可以利用os.path模块提供的函数更容易地在跨平台上处理文件。即使我们的程序不是用于夸平台，也应该使用os.path来让路径名字更加可靠。

19.2 解析路径

os.path中的第一个函数集可以用于解析文件名字符串为不同部分。要注意到这些函数的解析不依赖于被解析的路径是否真正存在，他们只处理字符串。

路径解析依赖于一些os实现定义好的变量，如：

- os.sep ：表示路径的分隔符(如，``/'')。
- os.extsep ：表示文件名和文件扩展名的分隔符(如，``.'')。
- os.pardir ：表示上一层目录，即父目录(如，``..'')。
- os.curdir ：表示当前目录(如，``.'')。

split() 函数将路径切分成两个两部分并返回一个元组，它的第二个元素是路径的最后一部份，第一个元素是路径的前面部分。

```
import os.path

for path in [ '/one/two/three','/one/two/three/','/',.','']:
    print '"%s" : "%s"' % (path, os.path.split(path))
```

```
$ python ospath_split.py
"/one/two/three" : "('/one/two', 'three')"
"/one/two/three/" : "('/one/two/three', '')"
"/" : "('/', '')"
"." : "('', '.')"
"" : "('', '')"
```

basename() 返回的值和 split() 返回的第二个值相同。

```
import os.path

for path in [ '/one/two/three','/one/two/three/','/','.','']:
    print '"%s" : "%s"' % (path, os.path.basename(path))
```

```
$ python ospath_basename.py
"/one/two/three" : "three"
"/one/two/three/" : ""
"/" : ""
"." : "."
"" : ""
```

dirname() 返回的值是和 split() 返回的第一个值相同。

```
import os.path

for path in [ '/one/two/three', '/one/two/three/','/','.','']:
    print '"%s" : "%s"' % (path, os.path.dirname(path))
```

```
$ python ospath_dirname.py

"/one/two/three" : "/one/two"
"/one/two/three/" : "/one/two/three"
"/" : "/"
"." : ""
"" : ""
```

splitext() 和 split() 类似但是分隔路径的扩展名，而不是目录名。

```
import os.path

for path in [ 'filename.txt', 'filename', '/path/to/filename.txt', '/', '' ]:
    print '"%s" :' % path, os.path.splitext(path)
```

```
$ python ospath_splitext.py
"filename.txt" : ('filename', '.txt')
"filename" : ('filename', '')
"/path/to/filename.txt" : ('/path/to/filename', '.txt')
"/" : ('/', '')
"" : ('', '')
```

commonprefix() 取一个路径列表作为参数，返回一个单一的字符串表示这些路径公共的前缀。这个值可能是一个实际上不存在的路径。路径分割符是被忽略的，所以前缀可能在在分割处被截断。

```
import os.path

paths = ['/one/two/three/four','/one/two/threefold','/one/two/three/',]
print paths
print os.path.commonprefix(paths)
```

```
$ python ospath_commonprefix.py
['/one/two/three/four', '/one/two/threefold', '/one/two/three/']
/one/two/three
```

19.3 创建路径

除了将现有的路径分隔外，你可能经常会将多个字符串组合成一个路径。

可以使用 `join()` 将多个路径部分组合成一个单个值：

```
import os.path

for parts in [ ('one', 'two', 'three'),('/', 'one', 'two', 'three'),('/one', '/two', '/three'),]:
    print parts, ':', os.path.join(*parts)
```

```
$ python ospath_join.py
('one', 'two', 'three') : one/two/three
('/', 'one', 'two', 'three') : /one/two/three
('/one', '/two', '/three') : /three
```

如果路径中包含变量部分，也能自动将她扩展出来． 例如， `expanduser()` 可以将波浪线(~)扩展成用户的主目录。

```
import os.path

for user in [ '', 'dhellmann', 'postgres' ]:
    lookup = '~' + user
    print lookup, ':', os.path.expanduser(lookup)
```

```
$ python ospath_expanduser.py
~ : /Users/dhellmann
~dhellmann : /Users/dhellmann
~postgres : /var/empty
```

`expandvars()` 是能更一般的扩展出现在路径中的环境变量。

```
import os.path
import os

os.environ['MYVAR'] = 'VALUE'

print os.path.expandvars('/path/to/$MYVAR')
```

```
$ python ospath_expandvars.py
/path/to/VALUE
```

19.4 标准化路径

使用 `join()` 组装成的，或嵌入了变量的Paths路径可能会以多余的分隔符结束或含有相对路径部份。使用 `normpath()` 将这些清除：

```
import os.path

for path in [ 'one//two//three', 'one/./two/./three', 'one/../one/two/three',]:
    print path, ':', os.path.normpath(path)
```

```
$ python ospath_normpath.py
one//two//three : one/two/three
one/./two/./three : one/two/three
one/../one/two/three : one/two/three
```

使用 **abspath()** 将一个相对路径转换成绝对路径。

```
import os.path

for path in [ '.', '..', './one/two/three', '../one/two/three']:
    print '"%s" : "%s"' % (path, os.path.abspath(path))
```

```
$ python ospath_abspath.py
"." : "/Users/dhellmann/Documents/PyMOTW/in_progress/ospath"
".." : "/Users/dhellmann/Documents/PyMOTW/in_progress"
"./one/two/three" : "/Users/dhellmann/Documents/PyMOTW/in_progress/ospath/one/two/three"
"../one/two/three" : "/Users/dhellmann/Documents/PyMOTW/in_progress/one/two/three"
```

19.5 文件时间

除了处理路径外， os.path 还可以包含一些用于检索文件属性的函数，他们比 os.stat() 更方便：

```
import os.path
import time

print 'File :', __file__
print 'Access time :', time.ctime(os.path.getatime(__file__))
print 'Modified time:', time.ctime(os.path.getmtime(__file__))
print 'Change time :', time.ctime(os.path.getctime(__file__))
print 'Size :', os.path.getsize(__file__)
```

```
$ python ospath_properties.py
File : /Users/dhellmann/Documents/PyMOTW/in_progress/ospath/ospath_properties.py
Access time : Sun Jan 27 15:40:20 2008
Modified time: Sun Jan 27 15:39:06 2008
Change time : Sun Jan 27 15:39:06 2008
Size : 478
```

19.6 测试文件

当你的程序含一个路径名时，他通常需要知道这个路径是否是文件还是目录。如果你的平台支持它，你需要知道这个路径是指向一个符号链接还是是一个挂载点。你也可能想测试路径是否存在。 os.path 提供测试这些条件的函数。

```
import os.path

for file in [ __file__, os.path.dirname(__file__), '/', './broken_link']:
    print 'File :', file
    print 'Absolute :', os.path.isabs(file)
    print 'Is File? :', os.path.isfile(file)
    print 'Is Dir? :', os.path.isdir(file)
    print 'Is Link? :', os.path.islink(file)
    print 'Mountpoint? :', os.path.ismount(file)
    print 'Exists? :', os.path.exists(file)
```

```
    print 'Link Exists?:', os.path.lexists(file)
    print


$ ln -s /does/not/exist broken_link
$ python ospath_tests.py
File : /Users/dhellmann/Documents/PyMOTW/in_progress/ospath/ospath_tests.py
Absolute : True
Is File? : True
Is Dir? : False
Is Link? : False
Mountpoint? : False
Exists? : True
Link Exists?: True

File : /Users/dhellmann/Documents/PyMOTW/in_progress/ospath
Absolute : True
Is File? : False
Is Dir? : True
Is Link? : False
Mountpoint? : False
Exists? : True
Link Exists?: True

File : /
Absolute : True
Is File? : False
Is Dir? : True
Is Link? : False
Mountpoint? : True
Exists? : True
Link Exists?: True

File : ./broken_link
Absolute : False
Is File? : False
Is Dir? : False
Is Link? : True
Mountpoint? : False
Exists? : False
Link Exists?: True
```

19.7 遍历目录树

os.path.walk() 遍历树中的所有目录，并调用一个你提供的函数，同时将目录名和目录中包含内容的名字传递给这个函数。下面的例子将递归的列出目录，但忽略.svn目录。

```
import os.path
import pprint

def visit(arg, dirname, names):
    print dirname, arg
    for name in names:
        subname = os.path.join(dirname, name)
        if os.path.isdir(subname):
            print ' %s/' % name
        else:
            print ' %s' % name
    # Do not recurse into .svn directory
```

```
    if '.svn' in names:
        names.remove('.svn')
    print

os.path.walk('..', visit, '(User data)')
```

```
$ python ospath_walk.py
.. (User data)
   .svn/
   ospath/

../ospath (User data)
   .svn/
   __init__.py
   ospath_abspath.py
   ospath_basename.py
   ospath_commonprefix.py
   ospath_dirname.py
   ospath_expanduser.py
   ospath_expandvars.py
   ospath_join.py
   ospath_normpath.py
   ospath_properties.py
   ospath_split.py
   ospath_splitext.py
   ospath_tests.py
   ospath_walk.py
```

CHAPTER

TWENTY

PYMOTW: TIME

time模块提供了操作日期和时间的函数

- 模块：time
- 目的：操作times的函数
- python版本：1.4+

20.1 描述

time模块是利用了c函数来处理日期和时间 ，也就是说它绑定了c的实现，一些特定的细节（比如纪元的开始时间、日期的最大值）是和平台相关的，具体可以参考 这里 。

20.2 Wall Clock Time

time模块的核心函数之一就是time.time()函数，它返回一个自公元开始的总秒数（浮点型）。

本工具包含三个文件：

```
import time
print 'The time is:', time.time()
```

虽然返回的值是浮点型，但精度是依赖于不同的系统平台的。

```
$ python time_time.py
The time is: 1205079300.54
```

当存储和比较日期时，浮点型一般是很有用的，但这种方式不易阅读，为了更有用的记录和输出时间可以使用time.ctime()。

```
import time
print 'The time is      :', time.ctime()
later = time.time() + 15
print '15 secs from now :', time.ctime(later)
```

上面第二行示范了如何来利用ctime()函数对当前时间进行格式化。

```
$ python time_ctime.py
The time is      : Sun Mar  9 12:18:02 2008
15 secs from now : Sun Mar  9 12:18:17 2008
```

20.3 处理器时钟

time()函数返回的是现实世界的时间，而clock()函数返回的是cpu时钟。clock()函数返回值常用作性能测试，benchmarking等。它们常常反映了程序运行的真实时间，比time()函数返回的值要精确。

```
import hashlib
import time

# Data to use to calculate md5 checksums
data = open(__file__, 'rt').read()

for i in range(5):
    h = hashlib.sha1()
    print time.ctime(), ': %0.3f %0.3f' % (time.time(), time.clock())
    for i in range(100000):
        h.update(data)
    cksum = h.digest()
```

在这个例子中，ctime()把time()函数返回的浮点型表示转化为标准时间，每个迭代循环使用了clock()。如果想在机器上测试这个例子，那么可以增加循环次数，或者处理大一点的数据，这样才能看到不同点。

```
$ python time_clock.py
Sun Mar  9 12:41:53 2008 : 1205080913.260 0.030
Sun Mar  9 12:41:53 2008 : 1205080913.682 0.440
Sun Mar  9 12:41:54 2008 : 1205080914.103 0.860
Sun Mar  9 12:41:54 2008 : 1205080914.518 1.270
Sun Mar  9 12:41:54 2008 : 1205080914.932 1.680
```

一般，如果程序没有做任何事情，处理器时钟是不会计时。

```
import time

for i in range(6, 1, -1):
    print '%s %0.2f %0.2f' % (time.ctime(), time.time(), time.clock())
    print 'Sleeping', i
    time.sleep(i)
```

在这个例子中，每次迭代，循环中处理了很少的任务就进入了sleep，当进程在睡眠中时，time.time()函数的返回值依然会增加。但是time.clock()是不会增加的。

```
$ python time_clock_sleep.py
Sun Mar  9 12:46:36 2008 1205081196.20 0.02
Sleeping 6
Sun Mar  9 12:46:42 2008 1205081202.20 0.02
Sleeping 5
Sun Mar  9 12:46:47 2008 1205081207.20 0.02
Sleeping 4
Sun Mar  9 12:46:51 2008 1205081211.20 0.02
Sleeping 3
Sun Mar  9 12:46:54 2008 1205081214.21 0.02
Sleeping 2
```

time.sleep函数控制当前的线程，让它等待直到系统重新唤醒它，如果应用中只有一个线程，那么它会阻塞当前进程，使其不做任何事情。

20.4 struct_time

某些时候，使用逝去的秒数来表示时间是很有用的。有时候你需要获取日期的单独部分（如年、月等等），time模块定义了struct_time来存储日期和时间值并作为其部分以便获取。提供了多种函数将struct_time转化为float。

```
import time

print 'gmtime   :', time.gmtime()
print 'localtime:', time.localtime()
print 'mktime   :', time.mktime(time.localtime())

print
t = time.localtime()
print 'Day of month:', t.tm_mday
print ' Day of week:', t.tm_wday
print ' Day of year:', t.tm_yday
```

gmtime()返回当前的UTC时间，localtime()返回当前时间域的当前时间，mktime()接收struct_time参数并将其转化为浮点型来表示。

```
$ python time_struct.py
gmtime   : (2008, 3, 9, 16, 58, 19, 6, 69, 0)
localtime: (2008, 3, 9, 12, 58, 19, 6, 69, 1)
mktime   : 1205081899.0

Day of month: 9
Day of week: 6
Day of year: 69
```

20.5 解析和格式化时间

函数strptime()和strftime()可以使struct_time和时间值字符串相互转化。有一个很长的格式化说明列表可以用来支持输入和输出不同的风格。完整的列表在time模块的的库文档中有介绍。

下面示例把当前时间（字符串）转化为struct_time实例，然后再转化为字符串。

```
import time

now = time.ctime()
print now
parsed = time.strptime(now)
print parsed
print time.strftime("%a %b %d %H:%M:%S %Y", parsed)
```

输出和输入字符串不是完全的一致，主要表现在月份前加了一个0前缀。

```
$ python time_strptime.py
Sun Mar  9 13:01:19 2008
(2008, 3, 9, 13, 1, 19, 6, 69, -1)
Sun Mar 09 13:01:19 2008
```

20.6 使用Time Zone(时区)

无论是你的程序，还是为系统使用一个默认的时区，检测当前时间的函数依赖于当前Time Zone（时间域)的设置。改变时区设置是不会改变实际时间，只会改变表示时间的方法。

通过设置环境变量TZ可以改变时区，然后调用tzset()。环境变量TZ可以对时区来详细的设置，比如白天保存时间的起始点。通常使用时区名称是比较简单的，如果需要了解更多信息可以参考库。

下面这个示例改变了time zone中的一些值，展示了这种改变如何来影响time模块中的其它设置。

```
import time
import os

def show_zone_info():
    print '\tTZ    :', os.environ.get('TZ', '(not set)')
    print '\ttzname:', time.tzname
    print '\tZone  : %d (%d)' % (time.timezone, (time.timezone / 3600))
    print '\tDST   :', time.daylight
    print '\tTime  :', time.ctime()
    print

print 'Default :'
show_zone_info()

for zone in [ 'US/Eastern', 'US/Pacific', 'GMT', 'Europe/Amsterdam' ]:
    os.environ['TZ'] = zone
    time.tzset()
    print zone, ':'
    show_zone_info()
```

我的时区是US/Eastern，所以设置TZ不会起作用。如果是其它时区，则会改变tzname、daylight flag以及 timezone偏移值。

```
$ python time_timezone.py
Default :
  TZ    : (not set)
  tzname: ('EST', 'EDT')
  Zone  : 18000 (5)
  DST   : 1
  Time  : Sun Mar  9 13:06:53 2008

US/Eastern :
  TZ    : US/Eastern
  tzname: ('EST', 'EDT')
  Zone  : 18000 (5)
  DST   : 1
  Time  : Sun Mar  9 13:06:53 2008

US/Pacific :
  TZ    : US/Pacific
  tzname: ('PST', 'PDT')
  Zone  : 28800 (8)
  DST   : 1
  Time  : Sun Mar  9 10:06:53 2008

GMT :
  TZ    : GMT
  tzname: ('GMT', 'GMT')
  Zone  : 0 (0)
  DST   : 0
  Time  : Sun Mar  9 17:06:53 2008
```

```
Europe/Amsterdam :
  TZ    : Europe/Amsterdam
  tzname: ('CET', 'CEST')
  Zone  : -3600 (-1)
  DST   : 1
  Time  : Sun Mar  9 18:06:53 2008
```

20.7 参考

- datetime module
- locale module
- calendar module

CHAPTER

TWENTYONE

PYMOTW: DATETIME

datetime 模块包含了一些用于时间解析、格式化、计算的函数。

- 模块： datetime
- 目的： 日期/时间处理
- python版本： 2.3+

21.1 时间

时间值由time类来表示，Times有小时，分，秒和微秒属性，以及包含时区信息。初始化time实例的参数是可选的，但这样的话，你将获得初始值0（也许不是你所想要的）。

```
import datetime

t = datetime.time(1, 2, 3)
print t
print 'hour :', t.hour
print 'minute:', t.minute
print 'second:', t.second
print 'microsecond:', t.microsecond
print 'tzinfo:', t.tzinfo


$ python datetime_time.py
01:02:03
hour  : 1
minute: 2
second: 3
microsecond: 0
tzinfo: None
```

一个time实例只包含时间值，不包含日期值。

```
import datetime

print 'Earliest  :', datetime.time.min
print 'Latest    :', datetime.time.max
print 'Resolution:', datetime.time.resolution
```

类属性中的最大最小值反应了一天中的时间范围。

```
$ python datetime_time_minmax.py
Earliest  : 00:00:00
```

```
Latest    : 23:59:59.999999
Resolution: 0:00:00.000001
```

时间的最小取值是微秒，更精确的位数就被截断了。

```
import datetime

for m in [ 1, 0, 0.1, 0.6 ]:
    print '%02.1f :' % m, datetime.time(0, 0, 0, microsecond=m)
```

实际中，如果使用浮点型作为微秒参数，那么将产生一些警告信息。

```
$ python datetime_time_resolution.py
/home/cjj/python/pymotw/datetime_time_resolution.py:14: DeprecationWarning: integer argument expected, got floa


print '%02.1f :' % m, datetime.time(0, 0, 0, microsecond=m)


1.0 : 00:00:00.000001
0.0 : 00:00:00
0.1 : 00:00:00
0.6 : 00:00:00
```

21.2 日期

日期值可以由date类来表示，实例有年、月、日属性，使用data类的 `today()` 方法可以方便的表示出今天的日期。

```
import datetime

today = datetime.date.today()
print today
print 'ctime:', today.ctime()
print 'tuple:', today.timetuple()
print 'ordinal:', today.toordinal()
print 'Year:', today.year
print 'Mon :', today.month
print 'Day :', today.day
```

示例演示了今天日期的多种表示方法：

```
$ python datetime_date.py
2008-03-13
ctime: Thu Mar 13 00:00:00 2008
tuple: (2008, 3, 13, 0, 0, 0, 3, 73, -1)
ordinal: 733114
Year: 2008
Mon : 3
Day : 13
```

使用整数（从阳历的第1年1月1号开始）或者POSIX标准时间戳可以类实例。

```
import datetime
import time

o = 733114
print 'o:', o
```

```
print 'fromordinal(o):', datetime.date.fromordinal(o)
t = time.time()
print 't:', t
print 'fromtimestamp(t):', datetime.date.fromtimestamp(t)
```

示例显示了函数 `fromordinal()` 和 `fromtimestamp()` 返回了不同的结果。

```
$ python datetime_date_fromordinal.py
o: 733114
fromordinal(o): 2008-03-13
t: 1205436039.53
fromtimestamp(t): 2008-03-13
```

日期的最大和最小范围可以使用属性max和min来表示。

```
import datetime

print 'Earliest :', datetime.date.min
print 'Latest :', datetime.date.max
print 'Resolution:', datetime.date.resolution
```

一个日期的单位就是1天。

```
$ python datetime_date_minmax.py
Earliest  : 0001-01-01
Latest    : 9999-12-31
Resolution: 1 day, 0:00:00
```

对于一个存在的日期，可使用replace函数可以创建出一个新的日期实例。比如你可以改变年数，只保留月份和日。

```
import datetime

d1 = datetime.date(2008, 3, 12)
print 'd1:', d1

d2 = d1.replace(year=2009)
print 'd2:', d2


$ python datetime_date_replace.py
d1: 2008-03-12
d2: 2009-03-12
```

21.3 timedelta

除了 `replace()` 函数可以计算过去或者未来的时间，还可以使用timedelta类对日期值进行基本运算。通过timedelta可以加减一个日期来产生另外一个日期。timedelta中的内部值可以用天、秒和微秒来表示。

```
import datetime

print "microseconds:", datetime.timedelta(microseconds=1)
print "milliseconds:", datetime.timedelta(milliseconds=1)
print "seconds :", datetime.timedelta(seconds=1)
print "minutes :", datetime.timedelta(minutes=1)
print "hours :", datetime.timedelta(hours=1)
```

```
print "days :", datetime.timedelta(days=1)
print "weeks :", datetime.timedelta(weeks=1)
```

传递给构造器的中间值被转换为天、秒和微秒。

```
$ python datetime_timedelta.py

microseconds: 0:00:00.000001
milliseconds: 0:00:00.001000
seconds : 0:00:01
minutes : 0:01:00
hours : 1:00:00
days : 1 day, 0:00:00
weeks : 7 days, 0:00:00
```

21.4 比较

时间和日期值都可以通过标准的操作符来进行比较。

```
import datetime
import time

print 'Times:'
t1 = datetime.time(12, 55, 0)
print '\tt1:', t1
t2 = datetime.time(13, 5, 0)
print '\tt2:', t2
print '\tt1 < t2:', t1 < t2

print 'Dates:'
d1 = datetime.date.today()
print '\td1:', d1
d2 = datetime.date.today() + datetime.timedelta(days=1)
print '\td2:', d2
print '\td1 > d2:', d1 > d2


$ python datetime_comparing.py
Times:
     t1: 12:55:00
     t2: 13:05:00
     t1 < t2: True

Dates:
     d1: 2008-03-13
     d2: 2008-03-14
     d1 > d2: False
```

21.5 日期和时间组合

使用datetime类可以存储日期和时间的组合部分，类似于使用date。有多种方法可以创建datetime。

```
import datetime

print 'Now :', datetime.datetime.now()
```

```
print 'Today :', datetime.datetime.today()
print 'UTC Now:', datetime.datetime.utcnow()

d = datetime.datetime.now()
for attr in [ 'year', 'month', 'day', 'hour', 'minute', 'second', 'microsecond']:
    print attr, ':', getattr(d, attr)
```

同时，datetime实例拥有date和time对象的所有属性。

```
$ python datetime_datetime.py
Now : 2008-03-15 22:58:14.770074
Today : 2008-03-15 22:58:14.779804
UTC Now: 2008-03-16 03:58:14.779858
year : 2008
month : 3
day : 15
hour : 22
minute : 58
second : 14
microsecond : 780399
```

datetime类提供了一些类方法来创建新的实例，当然它也包含 `fromordinal()` 和 `fromtimestamp()`，如果你已经有一个日期实例和时间实例，并需要创建datetime的话，combine()方法比较有用。

```
import datetime

t = datetime.time(1, 2, 3)
print 't :', t
d = datetime.date.today()
print 'd :', d
dt = datetime.datetime.combine(d, t)
print 'dt:', dt
```

```
$ python datetime_datetime_combine.py
t : 01:02:03
d : 2008-03-16
dt: 2008-03-16 01:02:03
```

21.6 格式化和解析

datetime 对象的字符串表示方法默认使用的是 ISO 8601 格式（YYYY-MM-DDTHH:MM:SS.mmmmmm），使用 `strftime()` 可以产生其他格式，同样，如果你的输入值是用 `time.strptime()` 解析的时间戳，那么 `strptime()` 是一个合适的方法来把它转换为datetime实例。

```
import datetime

format = "%a %b %d %H:%M:%S %Y"

today = datetime.datetime.today()
print 'ISO :', today

s = today.strftime(format)
print 'strftime:', s

d = datetime.datetime.strptime(s, format)
print 'strptime:', d.strftime(format)
```

```
$ python datetime_datetime_strptime.py
ISO     : 2008-03-16 08:08:16.275134
strftime: Sun Mar 16 08:08:16 2008
strptime: Sun Mar 16 08:08:16 2008
```

21.7 时区

时区是由子类datetime.tzinfo来表示的，tzinfo是一个抽象的基类，你需要定义子类，并提供相应的方法去实现一些方法。很可惜，dateime不包含任何实际可用的实现，可以参考 文档 来获取一些示例。

21.8 参考

- PLEAC - Dates and Times
- WikiPedia: Proleptic Gregorian calendar

CHAPTER

TWENTYTWO

PYMOTW: URLLIB

urllib模块提供了一个访问网络资源的简单接口。

- 模块：urllib
- 目的：访问不需要认证的远程资源
- python版本：1.4+

虽然urllib可以与gopher和ftp协议一起使用，但下面的例子都是用了http协议。

22.1 HTTP GET:

这些例子的测试服务器是在BaseHTTPServer_GET.py中，这个脚本在PyMOTW例子的BaseHTTPServer模块中.在一个终端窗口中启动服务器，然后在另一个窗口中运行以下这些例子.

HTTP GET 是urllib最简单的操作。简单把URL传递给urlopen()来获取一个用于操作远程数据的类文件句柄。

```
import urllib

response = urllib.urlopen('http://localhost:8080/')
print 'RESPONSE:', response
print 'URL :', response.geturl()

headers = response.info()
print 'DATE :', headers['date']
print 'HEADERS :'
print '---------'
print headers

data = response.read()
print 'LENGTH :', len(data)
print 'DATA :'
print '---------'
print data
```

该示例服务器取得传入的值，并且返回格式化的纯文本response。从urlopen()返回的值通过info()方法给出HTTP服务器的headers的入口，并且通过read()和readlines()等方法获得远程资源的数据。

```
$ python urllib_urlopen.py
RESPONSE: <addinfourl at 10180248 whose fp = <socket._fileobject object at 0x935c30>>
URL : http://localhost:8080/
DATE : Sun, 30 Mar 2008 16:27:10 GMT
HEADERS :
```

```
---------
Server: BaseHTTP/0.3 Python/2.5.1
Date: Sun, 30 Mar 2008 16:27:10 GMT

LENGTH : 221
DATA :
---------
CLIENT VALUES:
client_address=('127.0.0.1', 54354) (localhost)
command=GET
path=/
real path=/
query=
request_version=HTTP/1.0

SERVER VALUES:
server_version=BaseHTTP/0.3
sys_version=Python/2.5.1
protocol_version=HTTP/1.0
```

类文件对象也是可以迭代的：

```
import urllib

response = urllib.urlopen('http://localhost:8080/')
for line in response:
    print line.rstrip()
```

因为返回的每一行都有换行符和完整的框架回车符 -艳 盛 11/21/08 1:31 PM ，所以在输出之前先去掉他们。

```
$ python urllib_urlopen_iterator.py
CLIENT VALUES:
client_address=('127.0.0.1', 54380) (localhost)
command=GET
path=/
real path=/
query=
request_version=HTTP/1.0

SERVER VALUES:
server_version=BaseHTTP/0.3
sys_version=Python/2.5.1
protocol_version=HTTP/1.0
```

22.2 编码参数：

将参数编码并且追加在URL之后，传给服务器。

```
import urllib
query_args = { 'q':'query string', 'foo':'bar' }
encoded_args = urllib.urlencode(query_args) ##  ,     a=aa&b=bb
print 'Encoded:', encoded_args

url = 'http://localhost:8080/?' + encoded_args
print urllib.urlopen(url).read()
```

注意query，在客户端的值的列表中包含了已编码的参数query。

```
$ python urllib_urlencode.py
Encoded: q=query+string&foo=bar
CLIENT VALUES:
client_address=('127.0.0.1', 54415) (localhost)
command=GET
path=/?q=query+string&foo=bar
real path=/
query=q=query+string&foo=bar
request_version=HTTP/1.0

SERVER VALUES:
server_version=BaseHTTP/0.3
sys_version=Python/2.5.1
protocol_version=HTTP/1.0
```

在查询字符串中使用单独的变量来传递值序列时，需传递doseq=True给urlencode()。

```
import urllib
query_args = { 'foo':['foo1', 'foo2'] }
print 'Single :', urllib.urlencode(query_args)
print 'Sequence:', urllib.urlencode(query_args, doseq=True)
```

```
$ python urllib_urlencode_doseq.py
Single : foo=%5B%27foo1%27%2C+%27foo2%27%5D
Sequence: foo=foo1&foo=foo2
```

为了解码查询字符串，可查看cgi模块中的FieldStorage类。

在查询参数里的一些特别字符，在传递给urlencode()后，在服务器端可能和URL一起引起解析错误。可以直接使用quote()或者quote_plus()函数在本地引用他们以生成安全的字符串。

```
import urllib

url = 'http://localhost:8080/~dhellmann/'
print 'urlencode() :', urllib.urlencode({'url':url})
print 'quote() :', urllib.quote(url)
print 'quote_plus():', urllib.quote_plus(url)
```

Note: quote_plus()能够替换更多的特殊字符。

```
$ python urllib_quote.py
urlencode() : url=http%3A%2F%2Flocalhost%3A8080%2F%7Edhellmann%2F
quote() : http%3A//localhost%3A8080/%7Edhellmann/
quote_plus(): http%3A%2F%2Flocalhost%3A8080%2F%7Edhellmann%2F
```

视情况而定，用unquote()或者unquote_plus()来还原quote操作。

```
import urllib

print urllib.unquote('http%3A//localhost%3A8080/%7Edhellmann/')
print urllib.unquote_plus('http%3A%2F%2Flocalhost%3A8080%2F%7Edhellmann%2F')
```

```
$ python urllib_unquote.py
http://localhost:8080/~dhellmann/
http://localhost:8080/~dhellmann/
```

22.3 HTTP POST:

这些例子的测试服务器是在BaseHTTPServer_POST.py中， 这个脚本在PyMOTW例子的BaseHTTPServer模块中.在一个终端窗口中启动服务器，然后在另一个窗口中运行以下这些例子.

通过POST代替GET方式传递数据给远程服务器，仅仅是把已编码的查询参数当作数据传递给urlopen().

```
import urllib
query_args = { 'q':'query string', 'foo':'bar' }
encoded_args = urllib.urlencode(query_args)
url = 'http://localhost:8080/'
print urllib.urlopen(url, encoded_args).read()
```

```
$ python urllib_urlopen_post.py
Client: ('127.0.0.1', 54545)
Path: /
Form data:
     q=query string
     foo=bar
```

如果服务器需要的不是已编码url形式的参数，你可以传递任一字节字符串作为发送的数据。

22.4 Paths vs. URLs:

一些操作系统使用不同的方法分离本地文件路径和URL。为了使代码简捷，你应该反复地使用函数pathname2url() 和 url2pathname()。因为我在Mac上工作，我必须明确引入Windows上的函数版本。使用由urllib导出的函数版本可以让你默认在正确平台下，因此就不用自己做了。

```
import os

from urllib import pathname2url, url2pathname

print '== Default =='
path = '/a/b/c'
print 'Original:', path
print 'URL :', pathname2url(path)
print 'Path :', url2pathname('/d/e/f')
print

from nturl2path import pathname2url, url2pathname

print '== Windows, without drive letter =='
path = path.replace('/', '\\')
print 'Original:', path
print 'URL :', pathname2url(path)
print 'Path :', url2pathname('/d/e/f')
print

print '== Windows, with drive letter =='
path = 'C:\\' + path.replace('/', '\\')
print 'Original:', path
print 'URL :', pathname2url(path)
print 'Path :', url2pathname('/d/e/f')
```

有两个Windows例子，分别是路径的前缀中有和没有驱动器名。

```
$ python urllib_pathnames.py
== Default ==
Original: /a/b/c
URL : /a/b/c
Path : /d/e/f

== Windows, without drive letter ==
Original: \a\b\c
URL : /a/b/c
Path : \d\e\f

== Windows, with drive letter ==
Original: C:\\a\b\c
URL : ///C|/a/b/c
Path : \d\e\f
```

22.5 带Cache简单检索：

检索数据是常见的操作，urllib包括urlretrieve()函数，因此你不用自己写它。urlretrieve()带有URL中的参数，一个用于存储数据的临时文件，一个用于报告下载进度的函数，和URL中要POST数据。如果没有给定文件名，urlretrieve()就建立一个临时文件。你自己能删除它，或者把它看作一个cache，可以用urlcleanup()移除它。

这个例子使用GET从web服务器中检索数据。

```
import urllib
import os

def reporthook(blocks_read, block_size, total_size):
    if not blocks_read:
        print 'Connection opened'
        return
    if total_size < 0:
        # Unknown size
        print 'Read %d blocks' % blocks_read
    else:
        amount_read = blocks_read * block_size
        print 'Read %d blocks, or %d/%d' % (blocks_read, amount_read, total_size)
        return


try:
    filename, msg = urllib.urlretrieve('http://blog.doughellmann.com/', reporthook=reporthook)
    print
    print 'File:', filename
    print 'Headers:'
    print msg
    print 'File exists before cleanup:', os.path.exists(filename)
finally:
    urllib.urlcleanup()
    print 'File still exists:', os.path.exists(filename)
```

由于服务器没有返回header中的Content-length，urlretrieve()不知道数据应该有多大，所以将-1传给reporthook()中的参数total_size。

```
$ python urllib_urlretrieve.py
Connection opened
Read 1 blocks
Read 2 blocks
```

```
Read 3 blocks
Read 4 blocks
Read 5 blocks
Read 6 blocks
Read 7 blocks
Read 8 blocks
Read 9 blocks
Read 10 blocks
Read 11 blocks
Read 12 blocks
Read 13 blocks
Read 14 blocks
Read 15 blocks
Read 16 blocks
Read 17 blocks
Read 18 blocks
Read 19 blocks

File: /var/folders/9R/9R1t+tR02Raxzk+F71Q50U+++Uw/-Tmp-/tmp3HRpZP
Headers:
Content-Type: text/html; charset=UTF-8
Last-Modified: Tue, 25 Mar 2008 23:09:10 GMT
Cache-Control: max-age=0 private
ETag: "904b02e0-c7ff-47f6-9f35-cc6de5d2a2e5"
Server: GFE/1.3
Date: Sun, 30 Mar 2008 17:36:48 GMT
Connection: Close

File exists before cleanup: True
File still exists: False
```

22.6 URLopener:

urllib提供了一个URLopener基类，并且默认使用FancyURLopener处理支持的协议。如果你想要改变其行为，你可能需要查看Python2.1中新加的urllib2模块（PyMOTW将会阐述）。

22.7 参考

- RFC 2616 - HTTP Specification
- cgi - For decoding query arguments
- PyMOTW: BaseHTTPServer
- urllib2 - For more complex URL access needs
- Python Module of the Week Home

CHAPTER

TWENTYTHREE

PYMOTW: FNMATCH

使用fnmatch模块处理Unix风格的文件名的比较。

- 模块： fnmatch
- 目的： 对文件名和Unix风格模式进行比较。
- python版本：1.4+

23.1 描述

fnmatch模块用来全局模式上比较文件名(比如在Unix Shell中的模式)。

23.2 简单匹配

fnmatch() 比较一个简单的文件名和一个模式并且返回一个布尔类型，即匹配返回True，不匹配返回Fasle。如果操作系统使用了一个大小写不敏感的文件系统，那么这种比较也是大小写不敏感的，否则是大小写敏感的。

```
import fnmatch
import os

pattern = 'fnmatch_*.py'
print 'Pattern :', pattern
print

files = os.listdir('.')
for name in files:
    print 'Filename: %-25s %s' % (name, fnmatch.fnmatch(name, pattern))
```

这个例子中，模式匹配所有以fnmatch_开头，以 .py 结尾的文件名。

```
$ python fnmatch_fnmatch.py

Pattern : fnmatch_*.py

Filename: .svn False
Filename: __init__.py False
Filename: fnmatch_filter.py True
Filename: fnmatch_fnmatch.py True
Filename: fnmatch_fnmatchcase.py True
Filename: fnmatch_translate.py True
```

为了在各个不同文件系统或操作系统设置也能强制匹配大小写，可以使用 fnmatchcase() 。

```
import fnmatch
import os

pattern = 'FNMATCH_*.PY'
print 'Pattern :', pattern
print

files = os.listdir('.')

for name in files:
    print 'Filename: %-25s %s' % (name, fnmatch.fnmatchcase(name, pattern))
```

由于我的笔记本使用大小写敏感的文件系统，所以修改后的模式不匹配任何一个文件。

```
$ python fnmatch_fnmatchcase.py
Pattern : FNMATCH_*.PY

Filename: .svn False
Filename: __init__.py False
Filename: fnmatch_filter.py False
Filename: fnmatch_fnmatch.py False
Filename: fnmatch_fnmatchcase.py False
Filename: fnmatch_translate.py False
```

23.3 过滤

你可以使用 `filter()` 来测试一系列的文件名。它返回匹配模式参数的名字列表。

```
import fnmatch
import os

pattern = 'fnmatch_*.py'
print 'Pattern :', pattern

files = os.listdir('.')
print 'Files :', files

print 'Matches :', fnmatch.filter(files, pattern)
```

在这个例子中， `filter()` 返回这篇文章中所有示例源文件的名字.# 即他们都以fnmatch_开头。

```
$ python fnmatch_filter.py

Pattern : fnmatch_*.py
Files : ['.svn', '__init__.py', 'fnmatch_filter.py', 'fnmatch_fnmatch.py', 'fnmatch_fnmatchcase.py', 'fnmatch_t
Matches : ['fnmatch_filter.py', 'fnmatch_fnmatch.py', 'fnmatch_fnmatchcase.py', 'fnmatch_translate.py']
```

23.4 翻译模式

在内部，fnmatch将这种全局模式转换成一个正则式，然后使用re模块来比较名字和模式。
`translate()` 函数是一个公共API用于将全局模式转换成正则式。

```
import fnmatch

pattern = 'fnmatch_*.py'
```

```
print 'Pattern :', pattern
print 'Regex :', fnmatch.translate(pattern)
```

Note: 为了得到一个有效的表达式，有些特殊字符被转义。

```
$ python fnmatch_translate.py
Pattern : fnmatch_*.py
Regex : fnmatch\_.*\.py$
```

23.5 参考

- glob module documentation

CHAPTER

TWENTYFOUR

PYMOTW: COOKIE

- 模块： Cookie
- 目的： 处理来自服务器端的HTTP cookies
- python版本： 2.1+

24.1 描述

很久以前，Cookies就已成为HTTP协议的一部分。现有的web开发框架提供了简单的访问cookies的接口。因此，程序员几乎不用担心怎样去格式化cookies数据或者确保头的正确发送。明白cookies是怎样工作以及有哪些工作模式是很让人受启发的事情。

Cookie模块实现了对cookies的解析，其大多是兼容RFC 2109的。它没有严重按照标准是因为MSIE 3.0x不支持整个标准。

24.2 创建和设置Cookie

Cookies可以用作状态管理， 通常被服务器存储并由客户端返回。最普通的创建cookies的用法可以如下的样子：

```
import Cookie

c = Cookie.SimpleCookie()
c['mycookie'] = 'cookie_value'
print c
```

输出是一个有效的Set-Cookie头，之后会作为HTTP响应传递给客户端。

```
$ python Cookie_setheaders.py
Set-Cookie: mycookie=cookie_value
```

24.3 Morsels

控制cookie的其他方面也是有必要的，比如说期限、路径、域。事实上，RFC对应的所有的cookies属性可以通过代表cookie值的Morsel对象来操作。

```
import Cookie
import datetime

def show_cookie(c):
```

```
    print c
    for key, morsel in c.iteritems():
    print
    print 'key =', morsel.key
    print ' value =', morsel.value
    print ' coded_value =', morsel.coded_value
    for name in morsel.keys():
        if morsel[name]:
            print ' %s = %s' % (name, morsel[name])

c = Cookie.SimpleCookie()

# A cookie with a value that has to be encoded to fit into the header
c['encoded_value_cookie'] = '"cookie_value"'
c['encoded_value_cookie']['comment'] = 'Notice that this cookie value has escaped quotes'

# A cookie that only applies to part of a site
c['restricted_cookie'] = 'cookie_value'
c['restricted_cookie']['path'] = '/sub/path'
c['restricted_cookie']['domain'] = 'PyMOTW'
c['restricted_cookie']['secure'] = True

# A cookie that expires in 5 minutes
c['with_max_age'] = 'expires in 5 minutes'
c['with_max_age']['max-age'] = 300 # seconds

# A cookie that expires at a specific time
c['expires_at_time'] = 'cookie_value'
expires = datetime.datetime.now() + datetime.timedelta(hours=1)
c['expires_at_time']['expires'] = expires.strftime('%a, %d %b %Y %H:%M:%S')

show_cookie(c)
```

以上的例子包括了两个不同的设置cookies期限的方法。你可以设置max-age为一些秒数，或者指定一个cookie失效的时间和日期。

```
$ python Cookie_Morsel.py
Set-Cookie: encoded_value_cookie="\"cookie_value\""; Comment=Notice that this cookie value has escaped quotes
Set-Cookie: expires_at_time=cookie_value; expires=Sun, 01 Jun 2008 11:37:00
Set-Cookie: restricted_cookie=cookie_value; Domain=PyMOTW; Path=/sub/path; secure
Set-Cookie: with_max_age="expires in 5 minutes"; Max-Age=300

key = restricted_cookie
value = cookie_value
coded_value = cookie_value
domain = PyMOTW
secure = True
path = /sub/path

key = with_max_age
value = expires in 5 minutes
coded_value = "expires in 5 minutes"
max-age = 300

key = encoded_value_cookie
value = "cookie_value"
coded_value = "\"cookie_value\""
comment = Notice that this cookie value has escaped quotes

key = expires_at_time
value = cookie_value
coded_value = cookie_value
```

```
expires = Sun, 01 Jun 2008 11:37:00
```

Cookie和Morsel对象都像是一个字典。Morsel对应以下固定的键值：

- expires 期限
- path 路径
- comment 注释
- domain 域
- max-age 最大时间
- secure 安全性
- version 版本

一个Cookie对象的键是一些独立的会被cookie存储的名字。来自于Morsel的键属性的信息也是可用的。

24.4 编码后的值

cookie头可能会需要编码后的值以便它们被正确的解析。

```
import Cookie

c = Cookie.SimpleCookie()
c['integer'] = 5
c['string_with_quotes'] = 'He said, "Hello, World!"'

for name in ['integer', 'string_with_quotes']:
    print c[name].key
    print ' %s' % c[name]
    print ' value=%s' % c[name].value, type(c[name].value)
    print ' coded_value=%s' % c[name].coded_value
    print
```

Morsel.value常常是cookie中已经被解码的值，而 Morsel.coded_value 的值是以一种符合传递给客户端要求的形式来表示的。

```
$ python Cookie_coded_value.py
integer
  Set-Cookie: integer=5
  value=5 <type 'str'>
  coded_value=5

string_with_quotes
  Set-Cookie: string_with_quotes="He said, \"Hello, World!\""
  value=He said, "Hello, World!" <type 'str'>
  coded_value="He said, \"Hello, World!\""
```

24.5 接收和解析Cookie头

一旦客户端收到Set-Cookie头，它会将这些cookies在接下来的请求中作为新的Cookie头返回给服务器。那么传入的头看起来是：

```
Cookie: integer=5; string_with_quotes="He said, \"Hello, World!\""
```

cookies既可以直接从HTTP响应头，或环境变量HTTP_COOKIE，这依赖于你的web服务器/框架。实例化时，将经过解码的没有头前缀的字符串传递给SimpleCookie，或者使用 `load()` 。

```
import Cookie

HTTP_COOKIE = r'integer=5; string_with_quotes="He said, \"Hello, World!\""'

print 'From constructor:'
c = Cookie.SimpleCookie(HTTP_COOKIE)
print c

print
print 'From load():'
c = Cookie.SimpleCookie()
c.load(HTTP_COOKIE)
print c
```

```
$ python Cookie_parse.py
From constructor:
Set-Cookie: integer=5
Set-Cookie: string_with_quotes="He said, \"Hello, World!\""

From load():
Set-Cookie: integer=5
Set-Cookie: string_with_quotes="He said, \"Hello, World!\""
```

24.6 选择输出格式

除了使用Set-Cookie头外，使用JavaScript给客户端增加cookies也是可以的。SimpleCookie和Morsel提供一种JavaScript输出格式，通过使用 `js_output()` 方法：

```
import Cookie

c = Cookie.SimpleCookie()
c['mycookie'] = 'cookie_value'
c['another_cookie'] = 'second value'
print c.js_output()
```

```
$ python Cookie_js_output.py

 <script type="text/javascript">
 <!-- begin hiding
 document.cookie = "another_cookie="second value"";
 // end hiding -->
 </script>

 <script type="text/javascript">
 <!-- begin hiding
 document.cookie = "mycookie=cookie_value";
 // end hiding -->
 </script>
```

24.7 不推荐使用的类

上面所有的例子中都是使用的SimpleCookie类。Cookie模块也提供2个其他的类，SerialCookie和SmartCookie。SerialCookie可以处理任何可以被pickle的值。SmartCookie指明了一个值是否需要被unpickle或者是否是一个简单的值。由于他们两者都使用了pickle，他们是你应用中的潜在安全漏洞，所以你最好不要使用他们。在服务器上存贮cookie状态。然后传递给客户端一个会话key，这是更安全的。

24.8 参考

- cookielib
- RFC 2109, HTTP State Management Mechanism

CHAPTER

TWENTYFIVE

PYMOTW: BASE64

base64模块提供一些函数l来把二进制数据转换为ASCII集，通常在明文协议的传输中使用。

- 模块： base64
- 目的： 编码二进制数据转化为ASCII码
- python版本： 1.4+

25.1 描述

base64、base32、base16可以分别编码转化8位字节为6位、5位、4位，允许非ASCII字节以编码为ASCII码的协议中传输，例如SMTP，“base”值对应是在每一个编码中字母表的长度。有一些原始编码的url类型会使用略有不同的结果。

25.2 Base64 编码

简单的文本编码示例如下：

```
import base64

initial_data = open(__file__, 'rt').read()

encoded_data = base64.b64encode(initial_data)

num_initial = len(initial_data)
padding = { 0:0, 1:2, 2:1 }[num_initial % 3]

print '%d bytes before encoding' % num_initial
print 'Expect %d padding bytes' % padding
print '%d bytes after encoding' % len(encoded_data)
print
print encoded_data
```

输出显示原来529字节的文本在编码之后被扩展到了708个字节，从编码过程来看，每一个24位序列（3个字节）作为输入，输出时候则增加了4个字节，最后2个字符“==”，则是简单的追加，因为原始字符串的位数不能被24整除。

在标准输出中时没有很多回车府，但是为了在文档中有好的可读性，在如下显示中稍作了变化。

```
$ python base64_b64encode.py
529 bytes before encoding
    Expect 2 padding bytes
    708 bytes after encoding
```

```
IyEvdXNyL2Jpbi9lbnYgcHl0aG9uCiMgZW5jb2Rpbmc
6IHV0Zi04CiMKIyBDb3B5cmlnaHQgKGMpIDIwMDggRG
91ZyBIZWxsbWFubiBBbGwgcmlnaHRzIHJlc2VydmVkL
gojCiIiIgoiIiIKCl9fdmVyc2lvbl9fID0gIiRJZDog
cHltb3R3LnB5IDEyMzkgMjAwOC0wMS0xNiAxMDo1NTo
xOVogZGhlbGxtYW5uICQiCgppbXBvcnQgYmFzZTY0Cg
ppbml0aWFsX2RhdGEgPSBvcGVuKF9fZmlsZV9fLCAnc
nQnKS5yZWFkKCkKCmVuY29kZWRfZGF0YSA9IGJhc2U2
NC5iNjRlbmNvZGUoaW5pdGlhbF9kYXRhKQoKbnVtX2l
uaXRpYWwgPSBsZW4oaW5pdGlhbF9kYXRhKQpwYWRkaW
5nID0geyAwOjAsIDE6MiwgMjoxIH1bbnVtX2luaXRpY
WwgJSAzXQoKcHJpbnQgJyVkIGJ5dGVzIGJlZm9yZSBl
bmNvZGluZycgJSBudW1faW5pdGlhbApwcmludCAnRXh
wZWN0ICVkIHBhZGRpbmcgYnl0ZXMnICUgcGFkZGluZw
pwcmludCAnJWQgYnl0ZXMgYWZ0ZXIgZW5jb2Rpbmcn
CUgbGVuKGVuY29kZWRfZGF0YSkKcHJpbnQKcHJpbnQg
ZW5jb2RlZF9kYXRhCg==
```

25.3 Base64 解码

编码的字符串可以转换为原来的格式，利用反向查询，把4个字节转换为3个字节。b64decode()函数可以帮助你。

```
import base64

original_string = 'This is the data, in the clear.'
print 'Original:', original_string

encoded_string = base64.b64encode(original_string)
print 'Encoded :', encoded_string

decoded_string = base64.b64decode(encoded_string)
print 'Decoded :', decoded_string
```

```
$ python base64_b64decode.py
Original: This is the data, in the clear.
Encoded : VGhpcyBpcyB0aGUgZGF0YSwgaW4gdGhlIGNsZWFyLg==
Decoded : This is the data, in the clear.
```

25.4 URL-Safe变化

默认的base64字母表可能会使用+和/，而这些字符可能出现在url中，因此必须为这些字符指定可选择的编码情况，+由a-来代替，(_)来代替/，其他字母表还是相同。

```
import base64

for original in [ '\xfb\xef', '\xff\xff' ]:
    print 'Original         :', repr(original)
    print 'Standard encoding:', base64.standard_b64encode(original)
    print 'URL-safe encoding:', base64.urlsafe_b64encode(original)
    print
```

```
$ python base64_urlsafe.py
Original         : '\xfb\xef'
Standard encoding: ++8=
URL-safe encoding: --8=

Original         : '\xff\xff'
Standard encoding: //8=
URL-safe encoding: __8=
```

25.5 其他编码

除了base 64以外，还有base 32和base 16（16进制）提供函数用于编码数据。

```
import base64

original_string = 'This is the data, in the clear.'
print 'Original:', original_string

encoded_string = base64.b32encode(original_string)
print 'Encoded :', encoded_string

decoded_string = base64.b32decode(encoded_string)
print 'Decoded :', decoded_string


$ python base64_base32.py
Original: This is the data, in the clear.
Encoded : KRUGS4ZANFZSA5DIMUQGIYLUMEWCA2LOEB2GQZJAMNWGKYLSFY======
Decoded : This is the data, in the clear.
```

base 16中的函数是以16进制方式工作。

```
import base64

original_string = 'This is the data, in the clear.'
print 'Original:', original_string

encoded_string = base64.b16encode(original_string)
print 'Encoded :', encoded_string

decoded_string = base64.b16decode(encoded_string)
print 'Decoded :', decoded_string


$ python base64_base16.py
Original: This is the data, in the clear.
Encoded : 546869732069732074686520646174612C20696E2074686520636C6561722E
Decoded : This is the data, in the clear.
```

25.6 参考

- RFC 3548 - The Base16, Base32, and Base64 Data Encodings

CHAPTER

TWENTYSIX

PYMOTW: WEBBROWSER

利用webbrowser模块可以向用户显示web页面。

- 模块：webbrowser
- 目的：在浏览器中打开web页面
- python版本：python2.1.3+

26.1 描述

在一个交互式的浏览程序中，webbrowser模块提供了一些用于打开URL链接的函数。在系统中安装的浏览器，通过模块的许多选项可以来获取利用他们。也可通过环境变量BROWSER来控制。

26.2 简单示例

在浏览器中打开一个页面，可以使用open()函数。

```
import webbrowser

webbrowser.open('http://docs.python.org/lib/module-webbrowser.html')
```

这个url会在一个新窗口中打开，当然如果当前已经有一个浏览器窗口，那么会做为一个新标签打开。

26.3 窗口 Vs 标签

如果你只想在新窗口中打开页面，那么可以使用 open_new() 。

```
import webbrowser

webbrowser.open_new('http://docs.python.org/lib/module-webbrowser.html')
```

如果你想在新的标签中打开，那么可以使用 open_new_tab() 。

26.4 使用特定的浏览器

因为某些原因，你的应用程序可能需要使用特定的浏览器，可以利用get()函数来取得浏览器访问对象，该对象提供了open()、open_new()和open_new_tab()函数，下面示例演示了如何利用lynx浏览器。

```
import webbrowser

b = webbrowser.get('lynx')
b.open('http://docs.python.org/lib/module-webbrowser.html')
```

可以参考模块文档来了解浏览器类型列表。

26.5 BROWSER 变量

用户可以在你的应用程序之外为BROWSER变量设置浏览器名字或者命令来控制webbrowser模块，值可以由一系列的浏览器名字组成，中间由系统分割符os.pathsep来分割。如果名字中包含%s，那么将被解释成命令，并且在URL中%s将被直接替换执行。否则，值将被传递给get()来获取控制对象。

举个例子，如下示例是打开一个lynx浏览器，假设它是可以获取的，不管是否还存在其它的浏览器。

```
BROWSER=lynx python webbrowser_open.py
```

如果BROWSER中的名字中没有一个可以正确工作，那么webbrowse会返回执行它的默认行为。

26.6 命令行接口

webbrowser模块的所有特性可以通过命令行获取，类似运行一个python程序。

```
$ python -m webbrowser
Usage: /Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/webbrowser.py [-n | -t] url
   -n: open new window
   -t: open new tab
```

CHAPTER

TWENTYSEVEN

PYMOTW: ANYDBM

- 模块： anydbm
- 目的： anydbm提供了针对DBM-style、String-keyed数据库的字典接口。
- python版本： 1.4+

27.1 描述

anydbm面向DBM数据库 ，利用简单的字符串值作为key来访问包含字符串的记录。它利用whichdb模块来识别dbhash、gdbm和dbm数据库，并使用appropriate模块来打开它们，常常在shelve 中作为后端使用，比如我们知道如何利用pickle来存储对象。她常被用作为shelve的后端，就像我们知道如何利用pickle来存储对象。

27.2 创建一个新的数据库

新数据库的存储格式是可以通过查询如下模块来选择：

- dbhash
- gdbm
- dbm
- dumbdbm

open函数通过flags标志来控制如何处理数据库文件，当在必要时创建一个新的数据库时候，使用''c'',当经常要创建一个新数据库时，使用''n''。

c和n的区别 ： 也就是说用c，如果不存在则创建，如果存在就不创建新的了，用n的话，不管存不存在都是创建新的空数据库。

开始，我们加载一些有用的模块：

```
import anydbm

db = anydbm.open('/tmp/example.db', 'n')
db['key'] = 'value'
db['today'] = 'Sunday'
db['author'] = 'Doug'
db.close()
```

在这个例子中，这个文件会总是被重新初始化，如果想知道被创建的数据库类型，可以使用whichdb。

```
import whichdb

print whichdb.whichdb('/tmp/example.db')
```

你得到的结果可能会不同，它取决于你安装在系统中的模块。

```
$ python anydbm_whichdb.py
dbhash
```

27.3 打开一个存在的数据库

要打开一个存在的数据库，使用标记''r''（只读）或者''w''（读写）。你不需要担心格式问题，因为数据库格式会自动由whichdb来识别，如果一个文件可以被识别，那么对应的模块会打开它。

```
import anydbm

db = anydbm.open('/tmp/example.db', 'r')
try:
    print 'keys():', db.keys()
    for k, v in db.iteritems():
        print 'iterating:', k, v
    print 'db["author"] =', db['author']
 finally:
    db.close()
```

一旦打开，db就是一个字典对象，支持一些常规方法：

```
$ python anydbm_existing.py
keys(): ['key', 'today', 'author']
iterating: key value
iterating: today Sunday
iterating: author Doug
db["author"] = Doug
```

27.4 错误案例

数据库关键词必须是字符串。

```
import anydbm

db = anydbm.open('/tmp/example.db', 'w')
try:
    db[1] = 'one'
finally:
    db.close()
```

传递其它类型结果将导致TypeError异常。

```
$ python anydbm_intkeys.py
Traceback (most recent call last):
File "/Users/dhellmann/Documents/PyMOTW/in_progress/anydbm/PyMOTW/anydbm/anydbm_intkeys.py", line 16, in <modul
db[1] = 'one'
File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/bsddb/__init__.py", line 230, in __setite
_DeadlockWrap(wrapF) # self.db[key] = value
File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/bsddb/dbutils.py", line 62, in DeadlockWr
```

```
return function(*_args, **_kwargs)
File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/bsddb/__init__.py", line 229, in wrapF
self.db[key] = value
TypeError: Integer keys only allowed for Recno and Queue DB's
```

值必须是字符串或者是空。

```
import anydbm

db = anydbm.open('/tmp/example.db', 'w')
try:
    db['one'] = 1
finally:
    db.close()
```

如果值是非字符串，那么同样会抛出TypeError异常。

```
$ python anydbm_intvalue.py
Traceback (most recent call last):
File "/Users/dhellmann/Documents/PyMOTW/in_progress/anydbm/PyMOTW/anydbm/anydbm_intvalue.py", line 16, in <modul
db['one'] = 1
File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/bsddb/__init__.py", line 230, in __setite
_DeadlockWrap(wrapF) # self.db[key] = value
File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/bsddb/dbutils.py", line 62, in DeadlockWr
return function(*_args, **_kwargs)
File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/bsddb/__init__.py", line 229, in wrapF
self.db[key] = value
TypeError: Data values must be of type string or None.
```

27.5 参考

- 标准库文件：anydbm

CHAPTER

TWENTYEIGHT

PYMOTW: SMTPLIB

- 模块：smtplib
- 目的： 与smtp服务器交互，提供邮件发送
- python版本：1.5.2+

smtplib包含了SMTP类，用于与邮件服务器进行邮件通信。

Note: 在以下示例中，邮件地址、主机名称、ip地址都是虚假的，但是举例说明的命令副本和响应的信息都是存在的。

28.1 发送一封邮件

SMTP最常用的方法是连接服务器并发送一封邮件，在构造器中指定邮件服务器名和端口名，或者你可以使用connect()方法来指定。一旦建立连接，就可以调用sendmail()函数，并附带信封体参数和消息内容，消息文本应该与RFC2822兼容。smtplib不会自动修改消息内容和头信息，这就意味着你需要自己添加From和To等头信息。

```
import smtplib
import email.utils
from email.mime.text import MIMEText

# Create the message
msg = MIMEText('This is the body of the message.')
msg['To'] = email.utils.formataddr(('Recipient', 'recipient@example.com'))
msg['From'] = email.utils.formataddr(('Author', 'author@example.com'))
msg['Subject'] = 'Simple test message'

server = smtplib.SMTP('mail')
server.set_debuglevel(True) # show communication with the server
try:
    server.sendmail('author@example.com', ['recipient@example.com'], msg.as_string())
finally:
    server.quit()
```

在这个示例中，调试开关被打开了，这样可以显示客户端和服务端之间的通讯信息，否则，示例不会显示任何信息。

```
$ python smtplib_sendmail.py
send: 'ehlo localhost.local\r\n'
reply: '250-mail.example.com Hello [192.168.1.17], pleased to meet you\r\n'
reply: '250-ENHANCEDSTATUSCODES\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-8BITMIME\r\n'
reply: '250-SIZE\r\n'
```

```
reply: '250-DSN\r\n'
reply: '250-ETRN\r\n'
reply: '250-AUTH GSSAPI DIGEST-MD5 CRAM-MD5\r\n'
reply: '250-DELIVERBY\r\n'
reply: '250 HELP\r\n'
reply: retcode (250); Msg: mail.example.com Hello [192.168.1.17], pleased to meet you
ENHANCEDSTATUSCODES
PIPELINING
8BITMIME
SIZE
DSN
ETRN
AUTH GSSAPI DIGEST-MD5 CRAM-MD5
DELIVERBY
HELP
send: 'mail FROM:<author@example.com> size=266\r\n'
reply: '250 2.1.0 <author@example.com>... Sender ok\r\n'
reply: retcode (250); Msg: 2.1.0 <author@example.com>... Sender ok
send: 'rcpt TO:<recipient@example.com>\r\n'
reply: '250 2.1.5 <recipient@example.com>... Recipient ok\r\n'
reply: retcode (250); Msg: 2.1.5 <recipient@example.com>... Recipient ok
send: 'data\r\n'
reply: '354 Enter mail, end with "." on a line by itself\r\n'
reply: retcode (354); Msg: Enter mail, end with "." on a line by itself
data: (354, 'Enter mail, end with "." on a line by itself')
send: 'From nobody Sun Sep 28 10:02:48 2008\r\nContent-Type: text/plain; charset="us-ascii"\r\nMIME-Version: 1.
reply: '250 2.0.0 m8SE2mpc015614 Message accepted for delivery\r\n'
reply: retcode (250); Msg: 2.0.0 m8SE2mpc015614 Message accepted for delivery
data: (250, '2.0.0 m8SE2mpc015614 Message accepted for delivery')
send: 'quit\r\n'
reply: '221 2.0.0 mail.example.com closing connection\r\n'
reply: retcode (221); Msg: 2.0.0 mail.example.com closing connection
```

注意sendmail的第二个参数，收件人信息需要是一个list结构，你可以在list写上很多的邮件地址，message会依次把消息发送给他们． 由于信封信息和和邮件头是分开的，所以你可以通过一些方法参数来指定密送一些人，但不可以在邮件头中设置。

28.2 认证和加密

SMTP同样可以处理认证和TLS(一种底层通讯的安全协议)加密。如果服务器支持它们，你可以自己来检测服务器是否支持TLS，可以直接调用ehlo()来鉴定并询问服务器支持何种类型扩展 。然后通过调用has_extn()来检查结果。一旦启动TLS，你可以在认证之前再次调用ehlo()。

```
import smtplib
import email.utils
from email.mime.text import MIMEText
import getpass

# Prompt the user for connection info
to_email = raw_input('Recipient: ')
servername = raw_input('Mail server name: ')
username = raw_input('Mail user name: ')
password = getpass.getpass("%s's password: " % username)

# Create the message
msg = MIMEText('Test message from PyMOTW.')
msg.set_unixfrom('author')
msg['To'] = email.utils.formataddr(('Recipient', to_email))
msg['From'] = email.utils.formataddr(('Author', 'author@example.com'))
```

```
msg['Subject'] = 'Test from PyMOTW'

server = smtplib.SMTP(servername)
try:
    server.set_debuglevel(True)

    # identify ourselves, prompting server for supported features
    server.ehlo()

    # If we can encrypt this session, do it
    if server.has_extn('STARTTLS'):
         server.starttls()
         server.ehlo() # re-identify ourselves over TLS connection

    server.login(username, password)
    server.sendmail('author@example.com', [to_email], msg.as_string())
finally:
    server.quit()
```

注意STARTTLS不会出现在扩展列表中，因为启动了TLS。

```
$ python smtplib_authenticated.py
Recipient: recipient@example.com
Mail server name: smtpauth.isp.net
Mail user name: user@isp.net
user@isp.net's password:
send: 'ehlo localhost.local\r\n'
reply: '250-elasmtp-isp.net Hello localhost.local [<your IP here>]\r\n'
reply: '250-SIZE 14680064\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-AUTH PLAIN LOGIN CRAM-MD5\r\n'
reply: '250-STARTTLS\r\n'
reply: '250 HELP\r\n'
reply: retcode (250); Msg: elasmtp-isp.net Hello localhost.local [<your IP here>]
SIZE 14680064
PIPELINING
AUTH PLAIN LOGIN CRAM-MD5
STARTTLS
HELP
send: 'STARTTLS\r\n'
reply: '220 TLS go ahead\r\n'
reply: retcode (220); Msg: TLS go ahead
send: 'ehlo localhost.local\r\n'
reply: '250-elasmtp-isp.net Hello localhost.local [<your IP here>]\r\n'
reply: '250-SIZE 14680064\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-AUTH PLAIN LOGIN CRAM-MD5\r\n'
reply: '250 HELP\r\n'
reply: retcode (250); Msg: elasmtp-isp.net Hello farnsworth.local [<your IP here>]
SIZE 14680064
PIPELINING
AUTH PLAIN LOGIN CRAM-MD5
HELP
send: 'AUTH CRAM-MD5\r\n'
reply: '334 PDExNjkyLjEyMjI2MTI1NzlAZWxhc210cC1tZWFseS5hdGwuc2EuZWFydGhsaW5rLm5ldD4=\r\n'
reply: retcode (334); Msg: PDExNjkyLjEyMjI2MTI1NzlAZWxhc210cC1tZWFseS5hdGwuc2EuZWFydGhsaW5rLm5ldD4=
send: 'ZGhlbGxtYW5uQGVhcnRobGluay5uZXQgN2Q1YjAyYTRmMGQ1YzZjM2NjOTNjZDc1MDQxN2ViYjg=\r\n'
reply: '235 Authentication succeeded\r\n'
reply: retcode (235); Msg: Authentication succeeded
send: 'mail FROM:<author@example.com> size=221\r\n'
reply: '250 OK\r\n'
reply: retcode (250); Msg: OK
```

```
send: 'rcpt TO:<recipient@example.com>\r\n'
reply: '250 Accepted\r\n'
reply: retcode (250); Msg: Accepted
send: 'data\r\n'
reply: '354 Enter message, ending with "." on a line by itself\r\n'
reply: retcode (354); Msg: Enter message, ending with "." on a line by itself
data: (354, 'Enter message, ending with "." on a line by itself')
send: 'Content-Type: text/plain; charset="us-ascii"\r\nMIME-Version: 1.0\r\nContent-Transfer-Encoding: 7bit\r\n
reply: '250 OK id=1KjxNj-00032a-Ux\r\n'
reply: retcode (250); Msg: OK id=1KjxNj-00032a-Ux
data: (250, 'OK id=1KjxNj-00032a-Ux')
send: 'quit\r\n'
reply: '221 elasmtp-isp.net closing connection\r\n'
reply: retcode (221); Msg: elasmtp-isp.net closing connection
```

28.3 验证一个邮件地址

SMTP协议包含一个命令可以询问服务器一个邮件地址是否合法，通常VRFY是关闭的，以防止垃圾邮件发送者找到合法的邮件地址，但是，如果它打开，你可以向服务器询问这个邮件地址并接受一个状态码，如果是可用的，那么会返回一个可用的完整用户名。

```
import smtplib

server = smtplib.SMTP('mail')
server.set_debuglevel(True) # show communication with the server
try:
    dhellmann_result = server.verify('dhellmann')
    notthere_result = server.verify('notthere')
finally:
    server.quit()

print 'dhellmann:', dhellmann_result
print 'notthere :', notthere_result
```

最后二行输出中表示，地址 dhellmann是合法的，notthere是非法的。

```
$ python smtplib_verify.py
send: 'vrfy <dhellmann>\r\n'
reply: '250 2.1.5 Doug Hellmann <dhellmann@mail.example.com>\r\n'
reply: retcode (250); Msg: 2.1.5 Doug Hellmann <dhellmann@mail.example.com>
send: 'vrfy <notthere>\r\n'
reply: '550 5.1.1 <notthere>... User unknown\r\n'
reply: retcode (550); Msg: 5.1.1 <notthere>... User unknown
send: 'quit\r\n'
reply: '221 2.0.0 mail.example.com closing connection\r\n'
reply: retcode (221); Msg: 2.0.0 mail.example.com closing connection
dhellmann: (250, '2.1.5 Doug Hellmann <dhellmann@mail.example.com>')
notthere : (550, '5.1.1 <notthere>... User unknown')
```

28.4 参考

- RFC 821: Simple Mail Transfer Protocol
- RFC 822: Standard for the Format of ARPA Internet Text Messages
- RFC 1869: SMTP Service Extensions

- RFC 2822: Internet Message Format
- 标准库文档: smtplib

CHAPTER

TWENTYNINE

PYMOTW: TRACE

- 模块：Trace
- 目的： 监控程序语句和函数运行情况,并且产生报告信息.
- python版本：2.3+

trace - 跟踪正在执行的Python语句

trace模块帮助你明白程序的运行过程.你可以跟踪执行的语句,产生报表,也能获取函数间的调用关系.

29.1 命令行接口

可以很简单的直接从命令行使用trace.给定以下的Python脚本：

```
from recurse import recurse

def main():
     print 'This is the main program.'
     recurse(2)
     return

if __name__ == '__main__':
     main()

def recurse(level):
     print 'recurse(%s)' % level
     if level:
         recurse(level-1)
     return

def not_called():
     print 'This function is never called.'
```

29.2 跟踪时的异常

我们可以使用--trace选项来查看程序运行时哪条语句正在被执行.

```
$ python -m trace --trace trace_example/main.py
--- modulename: threading, funcname: settrace
threading.py(70): _trace_hook = func
--- modulename: trace, funcname: <module>
<string>(1): --- modulename: trace, funcname: <module>
```

```
main.py(7): """
main.py(12): from recurse import recurse
--- modulename: recurse, funcname: <module>
recurse.py(7): """
recurse.py(12): def recurse(level):
main.py(14): def main():
main.py(19): if __name__ == '__main__':
main.py(20): main()
--- modulename: trace, funcname: main
main.py(15): print 'This is the main program.'
This is the main program.
main.py(16): recurse(2)
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(2)
recurse.py(14): if level:
recurse.py(15): recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(1)
recurse.py(14): if level:
recurse.py(15): recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(0)
recurse.py(14): if level:
recurse.py(16): return
recurse.py(16): return
recurse.py(16): return
main.py(17): return
```

输出结构的第一部分表明了trace的一个安装操作.剩下来的输出显示了每个函数的入口信息,包括函数位于哪个模块,然后是原脚本文件中的语句行.你可以看到函数recurse()被进入了3次,正如你在main()中调用的那样.

29.3 代码报告

从命令行中运行trace并使用--count选项可以产生代码信息报告,因此可以看到哪些行是被执行的,哪些被跳过了.因为你的程序通常是多个文件组成,那就会为每个文件产生独立的报表.默认下,报表文件在和模块的同一目录下被创建,并以模块名命名,而且使用.cover后缀名替换.py.

```
$ python -m trace --count trace_example/main.py
This is the main program.
recurse(2)
recurse(1)
recurse(0)
```

两个输出文件, trace_example/main.cover:

```
1: from recurse import recurse

1: def main():
1:     print 'This is the main program.'
1:     recurse(2)
1:     return

1: if __name__ == '__main__':
1:     main()
```

trace_example/recurse.cover:

```
1: def recurse(level):
3:     print 'recurse(%s)' % level
3:     if level:
2:         recurse(level-1)
3:     return
```

Note: 虽然代码行def recurse(level):有一个1数值, 这不意味着这个函数仅运行一次,而是意味着这个函数definition仅被执行一次. 使用不同的选项来多次运行程序是有可能的,并且保存报告数据,产生一个联合报告.

```
$ python -m trace --coverdir coverdir1 --count --file coverdir1/coverage_report.dat trace_example/main.py
This is the main program.
recurse(2)
recurse(1)
recurse(0)
Skipping counts file 'coverdir1/coverage_report.dat': [Errno 2] No such file or directory: 'coverdir1/coverage_


$ python -m trace --coverdir coverdir1 --count --file coverdir1/coverage_report.dat trace_example/main.py
This is the main program.
recurse(2)
recurse(1)
recurse(0)


$ python -m trace --coverdir coverdir1 --count --file coverdir1/coverage_report.dat trace_example/main.py
This is the main program.
recurse(2)
recurse(1)
recurse(0)


$ find coverdir1
coverdir1
coverdir1/coverage_report.dat
```

一旦报告信息被记录到.cover文件中,你可以使用--report选项产生报告.

```
$ python -m trace --coverdir coverdir1 --report --summary --missing --file coverdir1/coverage_report.dat trace_
lines cov% module (path)
533 0% threading (/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/threading.py)
8 100% trace_example.main (trace_example/main.py)
8 87% trace_example.recurse (trace_example/recurse.py)
$ find coverdir1
coverdir1
coverdir1/coverage_report.dat
coverdir1/threading.cover
coverdir1/trace_example.main.cover
coverdir1/trace_example.recurse.cover
```

程序一共运行了3次,因此在报告中显示的值要比第一份报告中的值高3倍.--summary选项在输出信息中增加了百分比信息.模块recurse只有 87%被报告.从这个报告中还可看到not_called()这个函数从未被运行,这个是由前缀>>>>>>表示.

```
3: def recurse(level):
9:     print 'recurse(%s)' % level
9:     if level:
6:         recurse(level-1)
9:     return
```

```
3: def not_called():
>>>>>> print 'This function is never called.'
```

29.4 调用关系

除了以上覆盖信息,trace还可以收集函数间调用关系.使用--listfuncs可以在结果中输出简单的函数调用关系:

```
$ python -m trace --listfuncs trace_example/main.py
This is the main program.
recurse(2)
recurse(1)
recurse(0)

functions called:
filename: /Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/threading.py, modulename: threading, f
filename: <string>, modulename: <string>, funcname: <module>
filename: trace_example/main.py, modulename: main, funcname: <module>
filename: trace_example/main.py, modulename: main, funcname: main
filename: trace_example/recurse.py, modulename: recurse, funcname: <module>
filename: trace_example/recurse.py, modulename: recurse, funcname: recurse
```

可以使用--trackcalls获得更多信息,比如说谁调用了函数.

```
$ python -m trace --listfuncs --trackcalls trace_example/main.py
This is the main program.
recurse(2)
recurse(1)
recurse(0)

calling relationships:

*** /Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/trace.py ***
--> /Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/threading.py
trace.Trace.run -> threading.settrace
--> <string>
trace.Trace.run -> <string>.<module>

*** <string> ***
--> trace_example/main.py
<string>.<module> -> main.<module>

*** trace_example/main.py ***
main.<module> -> main.main
--> trace_example/recurse.py
main.<module> -> recurse.<module>
main.main -> recurse.recurse

*** trace_example/recurse.py ***
recurse.recurse -> recurse.recurse
```

29.5 编程接口

通过trace接口增加更多的控制,你可以在你的程序中使用Trace对象.Trace可以让你设置fixtures和其他依赖关系在运行单个函数前或执行一个用于跟踪的Python命令.

```
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=False, trace=True)
tracer.run('recurse(2)')
```

由于例子只跟踪到recurse()函数, 所以结果中没有把main.py的信息包含进来.

```
$ python trace_run.py
--- modulename: threading, funcname: settrace
threading.py(70): _trace_hook = func
--- modulename: trace_run, funcname: <module>
<string>(1): --- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(2)
recurse.py(14): if level:
recurse.py(15): recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(1)
recurse.py(14): if level:
recurse.py(15): recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(0)
recurse.py(14): if level:
recurse.py(16): return
recurse.py(16): return
recurse.py(16): return
```

使用runfunc()也可以得到上述同样的输出.runfunc()接收任意位置和关键字参数,他们在函数被tracer调用时都被传递给函数.

```
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=False, trace=True)
tracer.runfunc(recurse, 2)
```

```
$ python trace_runfunc.py
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(2)
recurse.py(14): if level:
recurse.py(15): recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(1)
recurse.py(14): if level:
recurse.py(15): recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(0)
recurse.py(14): if level:
recurse.py(16): return
recurse.py(16): return
recurse.py(16): return
```

29.6 保存结果数据

就像在命令行中使用一样， 计算和报告信息也可以被记录下来.使用Trace对象的CoverageResults可以将这些数据明确的保存下来.

```
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=True, trace=False)
tracer.runfunc(recurse, 2)

results = tracer.results()
results.write_results(coverdir='coverdir2')
```

```
$ python trace_CoverageResults.py
recurse(2)
recurse(1)
recurse(0)

$ find coverdir2
coverdir2/
coverdir2//trace_example.recurse.cover

$ cat coverdir2/trace_example.recurse.cover
#!/usr/bin/env python
# encoding: utf-8
#
# Copyright (c) 2008 Doug Hellmann All rights reserved.
#
"""
"""

#__version__ = "$Id: recurse.py 1732 2008-10-12 14:50:28Z dhellmann $"
#end_pymotw_header

>>>>>> def recurse(level):
3: print 'recurse(%s)' % level
3: if level:
2: recurse(level-1)
3: return

>>>>>> def not_called():
>>>>>> print 'This function is never called.'
```

为了在生成报告时也保存计算数据,可以使用参数infile和outfile.

```
mport trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=True, trace=False, outfile='trace_report.dat')
tracer.runfunc(recurse, 2)

report_tracer = trace.Trace(count=False, trace=False, infile='trace_report.dat')
results = tracer.results()
results.write_results(summary=True, coverdir='/tmp')
```

传递给参数infile一个文件名来余弦读取存储的数据， 参数outfile指定在跟踪之后需要新建的一个结果文件名.如果infile和outfile是相同的,那么,就相当于在原有文件中增加新的数据.

```
$ python trace_report.py
recurse(2)
recurse(1)
recurse(0)
lines cov% module (path)
7 57% trace_example.recurse (trace_example/recurse.py)
```

29.7 Trace选项

Trace构造器可以带多个可选参数以便更好的控制运行行为.

- count: 布尔型.打开行号计数.默认是True.
- countfuncs: 布尔型.打开运行中函数调用列表.默认是False
- countcallers: 布尔型.打开跟踪时的调用者和被调用者信息.默认是False.
- ignoremods: 序列.在跟踪报告中需要忽略的模块或包列表.默认是一个空元祖.
- ignoredirs: 序列.在跟踪报告中需要忽略的目录(其中包含模块或包)列表.默认是一个空元祖.
- infile: 包含缓存信息的文件名,作为输入.默认是None.
- outfile: 用于存储缓存信息的文件名,作为输入.默认是None,也就是数据不被存储.

29.8 参考

- 标准库文档: trace

CHAPTER

THIRTY

PYMOTW: STRUCT

- 模块：Struct
- 目的： 实现字符串和二进制数据之间的相互转换
- python版本：1.4 +

struct模块包含了实现字符串字节和Python本地数据类型(如数字和字符串)间的相互转换的函数.

30.1 函数 vs Struct类

这里有一系列用来处理结构化数值的模块级函数， 同样存在Struct类(Python 2.5中新加入的)． 格式化描述即将字符串格式转化为可编译形式,类似于正则式的方式． 这种转换需要消耗一些资源,所以一旦创建Struct实例后并调用Struct实例的方法而不使用模块级的方法,这样是更有效的.下面举些使用Struct类的例子.

30.2 封装和解封

Structs支持将数据封装成字符串,也能够通过格式化描述(它是由一些代表特定数据类型的字符,可选的个数和字节序指示符组成的)从字符串中解封数据． 进一步的细节,可以参考 标准库文档

在下面的这个例子中， 格式化描述调用了一个整型或者长整型数， 一个2个字符组成的字符串和一个浮点数． 为了清晰描述，在格式化描述中包含了空格，但格式被编译后将忽略这个空格.

```
import struct
import binascii

values = (1, 'ab', 2.7)
s = struct.Struct('I 2s f')
packed_data = s.pack(*values)

print 'Original values:', values
print 'Format string  :', s.format
print 'Uses           :', s.size, 'bytes'
print 'Packed Value   :', binascii.hexlify(packed_data)
```

封装后的值被转换成16进制字节流输出,所以有些字符显示为空

```
$ python struct_pack.py
Original values: (1, 'ab', 2.7000000000000002)
Format string  : I 2s f
Uses           : 12 bytes
Packed Value   : 0100000061620000cdcc2c40
```

如果我们将封装后的值传递给unpack(), 可以基本上得到原来的数值(注意其中浮点数的差异).

```
import struct
import binascii

packed_data = binascii.unhexlify('0100000061620000cdcc2c40')

s = struct.Struct('I 2s f')
unpacked_data = s.unpack(packed_data)
print 'Unpacked Values:', unpacked_data
```

```
$ python struct_unpack.py
Unpacked Values: (1, 'ab', 2.7000000476837158)
```

30.3 字节序

默认情况下,使用标准C库中''字节序''的概念将数值编码.通过在字符串格式中直接指定一个明确的字节序可以简单的覆盖这个选项.

```
import struct
import binascii

values = (1, 'ab', 2.7)
print 'Original values:', values

endianness = [
    ('@', 'native, native'),
    ('=', 'native, standard'),
    ('<', 'little-endian'),
    ('>', 'big-endian'),
    ('!', 'network'),
]


for code, name in endianness:
    s = struct.Struct(code + ' I 2s f')
    packed_data = s.pack(*values)
    print
    print 'Format string  :', s.format, 'for', name
    print 'Uses           :', s.size, 'bytes'
    print 'Packed Value   :', binascii.hexlify(packed_data)
    print 'Unpacked Value :', s.unpack(packed_data)
```

```
$ python struct_endianness.py
Original values: (1, 'ab', 2.7000000000000002)

Format string  : @ I 2s f for native, native
Uses           : 12 bytes
Packed Value   : 0100000061620000cdcc2c40
Unpacked Value : (1, 'ab', 2.7000000476837158)

Format string  : = I 2s f for native, standard
Uses           : 10 bytes
Packed Value   : 010000006162cdcc2c40
Unpacked Value : (1, 'ab', 2.7000000476837158)

Format string  : < I 2s f for little-endian
Uses           : 10 bytes
```

```
Packed Value   : 010000006162cdcc2c40
Unpacked Value : (1, 'ab', 2.7000000476837158)

Format string  : > I 2s f for big-endian
Uses           : 10 bytes
Packed Value   : 000000016162402ccccd
Unpacked Value : (1, 'ab', 2.7000000476837158)

Format string  : ! I 2s f for network
Uses           : 10 bytes
Packed Value   : 000000016162402ccccd
Unpacked Value : (1, 'ab', 2.7000000476837158)
```

30.4 缓冲

在高性能的敏感情况或者通过通过第三方模块来传递数据经常会要求对二进制数据进行封装.一种优化的方式是避免为每一个封装结构分配新的缓冲区.函数pack_into()和unpack_from()支持直接写入到预分配的缓冲区中.

```
import struct
import binascii

s = struct.Struct('I 2s f')
values = (1, 'ab', 2.7)
print 'Original:', values

print
print 'ctypes string buffer'

import ctypes
b = ctypes.create_string_buffer(s.size)
print 'Before  :', binascii.hexlify(b.raw)
s.pack_into(b, 0, *values)
print 'After   :', binascii.hexlify(b.raw)
print 'Unpacked:', s.unpack_from(b, 0)

print
print 'array'

import array
a = array.array('c', '\0' *s.size)
print 'Before  :', binascii.hexlify(a)
s.pack_into(a, 0, *values)
print 'After   :', binascii.hexlify(a)
print 'Unpacked:', s.unpack_from(a, 0)
```

```
$ python struct_buffers.py

Original: (1, 'ab', 2.7000000000000002)

ctypes string buffer
Before  : 00000000000000000000000000
After   : 01000000061620000cdcc2c40
Unpacked: (1, 'ab', 2.7000000476837158)

array
Before  : 00000000000000000000000000
```

```
After   : 0100000061620000cdcc2c40
Unpacked: (1, 'ab', 2.7000000476837158)
```

30.5 参考

- struct
- array：用于处理固定类型的序列.
- binascii：用于产生二进制数据的ASCII表示.
- WikiPedia: Endianness

字节序,其实是数据的二进制形式的排列顺序,在内存存贮顺序,或者是传输时的顺序,或者还有其他特殊的规定.

CHAPTER

THIRTYONE

PYMOTW: ARRAY

time模块提供了操作日期和时间的函数

- 模块： array
- 目的： 有效管理固定数值序列。
- python版本：1.4+

数组模块定义了一个序列型的数据结构，非常像一个列表，只是其中元素的类型是相同的。支持的数据类型在 标准库文档 中列出了。他们是所有数值型或其他固定大小的基本数据类型，如bytes。

31.1 数组的初始化

一个数组实例化时需要一个描述数据类型的参数，还可能需要一个初始化序列。

```
import array
import binascii

s = 'This is the array.'
a = array.array('c', s)

print 'As string:', s
print 'As array :', a
print 'As hex :', binascii.hexlify(a)
```

在这个例子中，数组保存的是一字节序列并且用一个简单的字符串来初始化。

```
$ python array_string.py
As string: This is the array.
As array : array('c', [84, 104, 105, 115, 32, 105, 115,
32, 116, 104, 101, 32, 97, 114, 114, 97, 121, 46])
As hex : 546869732069732074686520617272617 92e
```

31.2 处理数组

一个数组可以被扩展，否则也可以与其他Python序列的相同的方式处理。

```
import array

a = array.array('i', xrange(5))
print 'Initial :', a
```

```
a.extend(xrange(5))
print 'Extended:', a

print 'Slice :', a[3:6]

print 'Iterator:', list(enumerate(a))
```

```
$ python array_sequence.py
Initial : array('i', [0, 1, 2, 3, 4])
Extended: array('i', [0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
Slice : array('i', [3, 4, 0])
Iterator: [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4),
(5, 0), (6, 1), (7, 2), (8, 3), (9, 4)]
```

31.3 数组和文件

使用一些编码高效的内置方法可以从文件中读入一个数组的内容，或者将数组内容写入文件中。

```
import array
import binascii
import tempfile

a = array.array('i', xrange(5))
print 'A1:', a

# Write the array of numbers to the file
output = tempfile.NamedTemporaryFile()
a.tofile(output.file) # must pass an *actual* file
output.flush()

# Read the raw data
input = open(output.name, 'rb')
raw_data = input.read()
print 'Raw Contents:', binascii.hexlify(raw_data)

# Read the data into an array
input.seek(0)
a2 = array.array('i')
a2.fromfile(input, len(a))
print 'A2:', a2
```

这个例子中，直接从二进制文件中读取原数据，并将它读入一个新的数组并将其转换为合适的类型。

```
$ python array_file.py
A1: array('i', [0, 1, 2, 3, 4])
Raw Contents: 00000000010000000200000003000000040000000 ## ?
A2: array('i', [0, 1, 2, 3, 4])
```

31.4 交替字节排序

如果数组中的数据不是按照本地字节序排列，或者在写入到文件之前需要进行交换来适合不同系统的不同字节序，很容易对整个数组进行这种转换。

```
import array

import binascii

def to_hex(a):
    chars_per_item = a.itemsize * 2 # 2 hex digits
    hex_version = binascii.hexlify(a)
    num_chunks = len(hex_version) / chars_per_item
    for i in xrange(num_chunks):
        start = i*chars_per_item
        end = start + chars_per_item
        yield hex_version[start:end]

a1 = array.array('i', xrange(5))
a2 = array.array('i', xrange(5))
a2.byteswap()

fmt = '%10s %10s %10s %10s'
print fmt % ('A1 hex', 'A1', 'A2 hex', 'A2')
print fmt % (('-' * 10,) * 4)
for values in zip(to_hex(a1), a1, to_hex(a2), a2):
    print fmt % values
```

```
$ python array_byteswap.py
   A1 hex       A1         A2 hex       A2
 ---------- ---------- ---------- ----------
  00000000          0   00000000          0
  01000000          1   00000001   16777216
  02000000          2   00000002   33554432
  03000000          3   00000003   50331648
  04000000          4   00000004   67108864
```

31.5 参考

- array
- Numerical Python NumPy是Python针对大数据集的有效处理模块。

CHAPTER

THIRTYTWO

PYMOTW: GETPASS

- 模块： getpass
- 目的： 提示用户输入一个值(通常为密码)，输入时不显示内容.
- python版本： 1.5.2

许多通过终端与用户交互的程序需要向用户询问密码而在用户输入时不将其内容显示出来，getpass 这个模块提供了一种简单的方法来安全地处理这个密码输入问题.

32.1 例子

getpass() 函数提示用户输入，并获取用户回车之前的内容. 输入内容以字符串的形式回传给调用者.

```
import getpass

p = getpass.getpass()
print 'You entered:', p
```

如果调用者没有指定提示内容，默认为''Password:''.

```
$ python getpass_defaults.py
Password:
You entered: sekret
```

当然提示可以是当然提示可以是程序所需要的任何内容.

```
import getpass

p = getpass.getpass(prompt='What is your favorite color? ')
if p.lower() == 'blue':
    print 'Right. Off you go.'
else:
    print 'Auuuuugh!'
```

我不推荐使用这种不安全的认证方式，但它仅仅只是为了说明这一点.

```
$ python getpass_prompt.py
What is your favorite color?
Right. Off you go.
$ python getpass_prompt.py
What is your favorite color?
Auuuuugh!
```

默认情况下，getpass() 在stdout中显示提示字符串． 程序可以产生有用的信息到sys.stdout，那么也可以将提示发到另一个流中，比如sys.stderr．

```
import getpass
import sys

p = getpass.getpass(stream=sys.stderr)
print 'You entered:', p
```

这种方式将标准输出重定向到一个pipe或者文件，这种方式将标准输出重定向到一个pipe或者文件，因而不显示密码提示． 用户输入的值同样不会回显到屏幕上．

```
$ python getpass_stream.py >/dev/null
Password:
```

32.2 在非终端中使用getpass

在Unix环境中，getpass() 总是需要一个通过termios控制的tty，因此可以控制密码不回显． 这也意味着无法从一个重定向到标准输入流的非终端流中获取输入．

```
$ echo "sekret" | python getpass_defaults.py
Traceback (most recent call last):
File "getpass_defaults.py", line 34, in
  p = getpass.getpass()
  File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/getpass.py", line 32, in unix_getpass
  old = termios.tcgetattr(fd) # a copy to save
termios.error: (25, 'Inappropriate ioctl for device')
```

由调用者决定是否需要确定输入流是否为一个tty，以及在非tty时使用备选方法．

```
import getpass
import sys

if sys.stdin.isatty():
    p = getpass.getpass('Using getpass: ')
else:
    print 'Using readline'
    p = sys.stdin.readline().rstrip()

print 'Read: ', p
```

在tty中：

```
$ python ./getpass_noterminal.py
Using getpass:
Read: sekret
```

非tty环境：

```
$ echo "sekret" | python ./getpass_noterminal.py
Using readline
Read: sekret
```

32.3 参考

- getpass

CHAPTER

THIRTYTHREE

PYMOTW: STRING

- 模块：string
- 目的：包括文本处理中的常量和类.
- python版本：2.5

string模块始于Python的最早版本. 2.0版本中，许多之前只在模块中实现的函数被转移为string对象的方法. 之后的版本中，虽然这些函数仍然可用，但是不推荐使用，并且在Python 3.0中将被去掉. string模块也包含了一些有用的常量和类来处理字符串和unicode对象，后面的讨论会集中在这个方面.

33.1 常量

string模块中的常量，例如ascii_letters和digits等，用来指定字符的种类. 有些常量是依赖于系统的，比如lowercase，因此会受用户语言设置的影响而改变. 而其它的常量，如hexdigits，不会因本地设置(区域选项)的改变而改变.

```
import string

for n in dir(string):
    if n.startswith('_'):
        continue
    v = getattr(string, n)
    if isinstance(v, basestring):
        print '%s=%s' % (n, repr(v))
        print
```

大部分常量的名字是很直观的.

```
$ python string_constants.py
ascii_letters='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
ascii_lowercase='abcdefghijklmnopqrstuvwxyz'
ascii_uppercase='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
digits='0123456789'
hexdigits='0123456789abcdefABCDEF'
letters='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
lowercase='abcdefghijklmnopqrstuvwxyz'
octdigits='01234567'
printable='0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\
punctuation='!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
uppercase='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
whitespace='\t\n\x0b\x0c\r '
```

33.2 函数

有两个函数没有从string模块中移除． capwords()将一个字符串所有单词首字母大写．

```
import string

s = 'The quick brown fox jumped over the lazy dog.'

print s
print string.capwords(s)
```

得到的结果和调用split()，将结果列表中的每个单词首字母大写，然后调用join()连接这些单词这一系列动作的结果相同．

```
$ python string_capwords.py
The quick brown fox jumped over the lazy dog.
The Quick Brown Fox Jumped Over The Lazy Dog.
```

另一个函数创建了一个翻译表． 这个翻译表作为translate()方法的参数，用来将某一集合中的字符改成另一个集合中的字符．

```
import string

leet = string.maketrans('abegiloprstz', '463611092572')

s = 'The quick brown fox jumped over the lazy dog.'

print s
print s.translate(leet)
```

在这个例子中，一些字符被替换为其 133t 数字．

```
$ python string_maketrans.py
The quick brown fox jumped over the lazy dog.
Th3 qu1ck 620wn f0x jum93d 0v32 7h3 142y d06.
```

33.3 模板

字符串模板是在Python 2.4中增加的，作为 PEP 292 的一部分，以及用作内置的占位符表达式的另一种实现形式． 若使用了 string.Template 的占位符，前缀为$的单词就被认为是变量(如$var)，如果需要将其在上下文中区别出来的话，也可以将变量名包括在大括号中(如${var})．

```
import string

values = { 'var':'foo' }
t = string.Template("""
$var
$$
${var}iable
""")

print 'TEMPLATE:', t.substitute(values)

s = """
%(var)s
%%
%(var)siable
```

```
"""

print 'INTERPLOATION:', s % values
```

如您所见, 两种形式中, 触发字符($或%)若重复两次则被转义为普通字符.

```
$ python string_template.py
TEMPLATE:
foo
$
fooiable

INTERPLOATION:
foo
%
fooiable
```

模板和标准字符串占位符的一个关键区别就是模板不会考虑参数类型. 参数的值将被转为字符串并插入到模板中. 模板中没有格式选项. 比如, 模板中无法控制显示一个浮点数时数字的个数.

而使用模板的一个好处是调用 `safe_substitute()` 方法, 当模板需要的参数值没有全部提供时, 可以避免了异常的产生.

```
import string

values = { 'var':'foo' }

t = string.Template("$var is here but $missing is not provided")

try:
    print 'TEMPLATE:', t.substitute(values)
except KeyError, err:
    print 'ERROR:', str(err)

print 'TEMPLATE:', t.safe_substitute(values)
```

因为missing这个变量的值没有出现在参数字典里, 所以 `substitue()` 会引发一个KeyError异常. 而 `safe_substitute()` 则捕获了这个异常并将这个变量表达式保留在文本中.

```
$ python string_template_missing.py
TEMPLATE: ERROR: 'missing'
TEMPLATE: foo is here but $missing is not provided
```

33.4 模板的高级应用

如果string.Template的默认表达式无法满足你的要求, 你可以通过调整用于匹配模板正文中变量名的正则表达式来达到你的目的. 一种简单的方法就是改变delimiter和idpattern这两个类属性.

```
import string

class MyTemplate(string.Template):
    delimiter = '%'
    idpattern = '[a-z]+_[a-z]+'

t = MyTemplate('%% %with_underscore %notunderscored')
d = { 'with_underscore':'replaced',
          'notunderscored':'not replaced',
}
```

```
print t.safe_substitute(d)
```

在这个例子中， 变量名必须在中间的某个位置包含一个下划线， 因此%notunderscored不会被替换为任何东西.

```
$ python string_template_advanced.py
% replaced %notunderscored
```

如果需要更复杂的变化，你可以重载pattern属性，定义一个全新的正则表达式. 新的pattern属性必须包含4个命名的组来分别匹配定界符，已命名的变量，区分变量的括号类型，和无效界定符模式.

让我们看一下默认的模式:

```
import string

t = string.Template('$var')
print t.pattern.pattern
```

因为t.pattern是已经被编译的正则表达式，我们只能通过它的pattern属性来看真实的字符串.

```
$ python string_template_defaultpattern.py

\$(?:
(?P<escaped>\$) | # Escape sequence of two delimiters
(?P<named>[_a-z][_a-z0-9]*) | # delimiter and a Python identifier
{(?P<braced>[_a-z][_a-z0-9]*)} | # delimiter and a braced identifier
(?P<invalid>) # Other ill-formed delimiter exprs
)
```

如果希望创建一个新的模板，如，以{{var}}作为变量表达式，可以使用这样的一个pattern:

```
import re
import string

class MyTemplate(string.Template):
    delimiter = '{{'
    pattern = re.compile(r'''
              \{\{(?:
              (?P<escaped>\{\{)|
              (?P<named>[_a-z][_a-z0-9]*)\}\}|
              (?P<braced>[_a-z][_a-z0-9]*)\}\}|
              (?P<invalid>)
              )
    ''', re.VERBOSE | re.DOTALL)

t = MyTemplate('''
    {{{{
    {{var}}
''')

print 'MATCHES:', t.pattern.findall(t.template)
print 'SUBSTITUTED:', t.safe_substitute(var='replacement')
```

即使named和braced是一样的，仍然需要将它们都描述出来. 下面是输出:

```
$ python string_template_newsyntax.py
MATCHES: [('{{', '', '', ''), ('', 'var', '', '')]
SUBSTITUTED:
{{
replacement
```

33.5 不推荐使用的函数

那些不推荐使用的函数已经被转移到string和unicode类中, 可参考手册中的 String Methods.

33.6 参考

- string
- PEP 292

CHAPTER

THIRTYFOUR

PYMOTW: EXCEPTIONS

- 模块： exceptions
- 目的： excpetions模块定义了在整个标准库和解释器中使用的内置错误.
- python版本： 1.5+

34.1 描述

Python之前就支持将简单的字符串信息和类一样作为异常. 从Python 1.5开始, 所有的标准库模块使用类作为异常. 而从Python 2.5开始, 字符串的异常会导致一个DeprecationWarning, 而对于字符串异常的支持会在未来的版本中去掉.

34.2 基类

在标准库文档的描述中, 异常类以分层结构定义. 这样做除了结构清晰之外, 异常的继承关系也很有用, 可以通过捕获基类的异常来捕获相关的异常. 大部分情况下, 这些基类异常不会直接被引发.

34.2.1 BaseException

所有异常的基类. 实现逻辑为使用 str() 将传给构造函数的参数创建为一个异常的字符串表达式.

34.2.2 Exception

不会导致运行的应用程序退出的异常的基类. 所有用户自定义的异常应该使用Exception作为基类.

34.2.3 StandardError

在标准库中使用的内置异常的基类.

34.2.4 ArithmeticError

数学相关错误的基类.

34.2.5 LookupError

因无法找到某些东西产生的错误的基类.

34.2.6 EnvironmentError

因Python之外的原因(操作系统，文件系统，等等)产生的错误的基类.

34.3 异常的引发

34.3.1 AssertionError

一个AssertionError由一个失败的 `assert` 语句引发.

```
assert False, 'The assertion failed'
```

```
$ python exceptions_AssertionError_assert.py
Traceback (most recent call last):
File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_AssertionError_asse
assert False, 'The assertion failed'
AssertionError: The assertion failed
```

也可以在 `unittest` 模块中像 `failIf()` 一样使用.

```
import unittest

class AssertionExample(unittest.TestCase):
    def test(self):
        self.failUnless(False)

unittest.main()
```

```
$ python exceptions_AssertionError_unittest.py
F
======================================================================
FAIL: test (__main__.AssertionExample)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_AssertionError_un
  self.failUnless(False)
AssertionError

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

34.3.2 AttributeError

若引用一个属性或对一个属性赋值时发生错误，则会引发AttributeError.

例如，当引用一个不存在的属性时:

```
class NoAttributes(object):
    pass

o = NoAttributes()
print o.attribute
```

```
$ python exceptions_AttributeError.py
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_AttributeError.py
  print o.attribute
AttributeError: 'NoAttributes' object has no attribute 'attribute'
```

或者尝试修改一个只读属性：

```
class MyClass(object):

    @property
    def attribute(self):
        return 'This is the attribute value'

o = MyClass()
print o.attribute
o.attribute = 'New value'
```

```
$ python exceptions_AttributeError_assignment.py
This is the attribute value
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_AttributeError_as
  o.attribute = 'New value'
AttributeError: can't set attribute
```

34.3.3 EOFError

当一个内置函数，如 `input()` 或 `raw_input()`，直到输入流的结尾也没有读到任何数据时，会引发EOFError异常．而对于文件的方法，如 `read()`，如果文件为空，则会在文件末尾处返回一个空字符串．

```
while True:
    data = raw_input('prompt:')
    print 'READ:', data
```

```
$ echo hello | python PyMOTW/exceptions/exceptions_EOFError.py
prompt:READ: hello
prompt:Traceback (most recent call last):
  File "PyMOTW/exceptions/exceptions_EOFError.py", line 13, in <module>
  data = raw_input('prompt:')
EOFError: EOF when reading a line
```

34.3.4 FloatingPointError

若浮点异常控制(fpectl)启用，则会在浮点运算出错时引发FloatingPointError．启用:mod:*fpectl* 需要有 `--with-fpectl` 选项编译的解释器．`fpectl` *在stdlib文档 <http://docs.python.org/lib/module-fpectl.html>中不推荐使用。*

```
import math
import fpectl

print 'Control off:', math.exp(1000)
fpectl.turnon_sigfpe()
print 'Control on:', math.exp(1000)
```

34.3.5 GeneratorExit

若调用generator的 `close()` 方法，则在generator内引发GeneratorExit异常.

```
def my_generator():
    try:
        for i in range(5):
            print 'Yielding', i
            yield i
    except GeneratorExit:
        print 'Exiting early'

g = my_generator()
print g.next()
g.close()
```

```
$ python exceptions_GeneratorExit.py
Yielding 0
0
Exiting early
```

34.3.6 IOError

输入或输出失败时引发IOError. 例如，磁盘已满或输入文件不存在.

```
f = open('/does/not/exist', 'r')
```

```
$ python exceptions_IOError.py
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_IOError.py", line
  f = open('/does/not/exist', 'r')
IOError: [Errno 2] No such file or directory: '/does/not/exist'
```

34.3.7 ImportError

当一个模块，或者模块的成员，不能被引入时引发ImportError. 以下是几种可能引发ImportError的情况.

1. 若一个模块不存在.

```
import module_does_not_exist
```

```
$ python exceptions_ImportError_nomodule.py
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_ImportError_nomod
  import module_does_not_exist
ImportError: No module named module_does_not_exist
```

1. 若执行了 `from X import Y`，而X模块中不存在Y，则引发ImportError.

```
from exceptions import MadeUpName
```

```
$ python exceptions_ImportError_missingname.py
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_ImportError_missi
  from exceptions import MadeUpName
ImportError: cannot import name MadeUpName
```

34.3.8 IndexError

当引用的序列下标越界时引发IndexError.

```
my_seq = [ 0, 1, 2 ]
print my_seq[3]
```

```
$ python exceptions_IndexError.py
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_IndexError.py", l
  print my_seq[3]
IndexError: list index out of range
```

34.3.9 KeyError

类似地，当一个字典中的键无法找到时引发KeyError.

```
d = { 'a':1, 'b':2 }
print d['c']
```

```
$ python exceptions_KeyError.py
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_KeyError.py", lin
  print d['c']
KeyError: 'c'
```

KeyboardInterrupt ~~~~~~~~~~~~~~~~~~~`

KeyboardInterrupt发生在用户按下Ctrl-C(或Delete)使一个正在运行的程序停止时。 和大部分别的异常不同， KeyboardInterrupt直接继承了BaseException， 以防止其被用来捕获Exception的全局异常处理器捕获.

```
try:
    print 'Press Return or Ctrl-C:',
    ignored = raw_input()
except Exception, err:
    print 'Caught exception:', err
except KeyboardInterrupt, err:
    print 'Caught KeyboardInterrupt'
else:
    print 'No exception'
```

在提示符下按下Ctrl-C导致KeyboardInterrupt异常.

```
$ python PyMOTW/exceptions/exceptions_KeyboardInterrupt.py
Press Return or Ctrl-C: ^CCaught KeyboardInterrupt
```

34.3.10 MemoryError

如果程序用完了内存并且有可能恢复(例如，通过删除某些对象)时，引发MemoryError.

```
import itertools

# Try to create a MemoryError by allocating a lot of memory
l = []
for i in range(3):
    try:
        for j in itertools.count(1):
            print i, j
            l.append('*' * (2**30))
    except MemoryError:
        print '(error, discarding existing list)'
        l = []
```

```
$ python PyMOTW/exceptions/exceptions_MemoryError.py
0 1
0 2
0 3
python(13482) malloc: *** mmap(size=1073745920) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
(error, discarding existing list)
1 1
1 2
1 3
python(13482) malloc: *** mmap(size=1073745920) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
(error, discarding existing list)
2 1
2 2
2 3
python(13482) malloc: *** mmap(size=1073745920) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
(error, discarding existing list)
```

34.3.11 NameError

若代码引用当前作用域下不存在的名字，则引发NameError. 例如，一个不存在的变量名.

```
def func():
    print unknown_name

func()
```

```
$ python exceptions_NameError.py
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_NameError.py", li
    func()
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_NameError.py", li
    print unknown_name
NameError: global name 'unknown_name' is not defined
```

34.3.12 NotImplementedError

用户自定义基类可以引发NotImplementedError来模拟 *接口* ，意思是一个方法或行为需要在子类中定义.

```
class BaseClass(object):
    """Defines the interface"""
    def __init__(self):
        super(BaseClass, self).__init__()
    def do_something(self):
        """The interface, not implemented"""
        raise NotImplementedError(self.__class__.__name__ + '.do_something')

class SubClass(BaseClass):
    """Implementes the interface"""
    def do_something(self):
        """really does something"""
        print self.__class__.__name__ + ' doing something!'


SubClass().do_something()
BaseClass().do_something()
```

```
$ python exceptions_NotImplementedError.py
SubClass doing something!
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_NotImplementedErr
    BaseClass().do_something()
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_NotImplementedErr
    raise NotImplementedError(self.__class__.__name__ + '.do_something')
NotImplementedError: BaseClass.do_something
```

34.3.13 OSError

OSError是为 `os` 模块服务的错误类，当一个系统相关的函数返回错误时引发.

```
import os

for i in range(10):
    print i, os.ttyname(i)
```

```
$ python exceptions_OSError.py
0
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_OSError.py", line
    print i, os.ttyname(i)
OSError: [Errno 25] Inappropriate ioctl for device
```

34.3.14 OverflowError

当一个算术操作超过了变量类型的限制时，引发OverflowError. 长整型会在其值增加时分配更多的空间，因此会最终引发MemoryError. 浮点数的异常处理不是标准化的，所以不会检查浮点数. 普通的整数在需要时会被转化为长整型.

```
import sys

print 'Regular integer: (maxint=%s)' % sys.maxint
try:
    i = sys.maxint * 3
    print 'No overflow for ', type(i), 'i =', i
except OverflowError, err:
    print 'Overflowed at ', i, err

print
print 'Long integer:'
for i in range(0, 100, 10):
    print '%2d' % i, 2L ** i

print
print 'Floating point values:'
try:
    f = 2.0**i
    for i in range(100):
        print i, f
        f = f ** 2
except OverflowError, err:
    print 'Overflowed after ', f, err
```

```
$ python exceptions_OverflowError.py
Regular integer: (maxint=2147483647)
No overflow for  <type 'long'> i = 6442450941

Long integer:
0 1
10 1024
20 1048576
30 1073741824
40 1099511627776
50 1125899906842624
60 1152921504606846976
70 1180591620717411303424
80 1208925819614629174706176
90 1237940039285380274899124224

Floating point values:
0 1.23794003929e+27
1 1.53249554087e+54
2 2.34854258277e+108
3 5.5156522631e+216
Overflowed after  5.5156522631e+216 (34, 'Result too large')
```

34.3.15 ReferenceError

当使用:mod:*weakref* 代理访问已经被作为垃圾回收的对象时，引发ReferenceError.

```
import gc
import weakref

class ExpensiveObject(object):
    def __init__(self, name):
        self.name = name
    def __del__(self):
        print '(Deleting %s)' % self
```

```
obj = ExpensiveObject('obj')
p = weakref.proxy(obj)

print 'BEFORE:', p.name
obj = None
print 'AFTER:', p.name
```

```
$ python exceptions_ReferenceError.py
BEFORE: obj
(Deleting <__main__.ExpensiveObject object at 0x844f0>)
AFTER:
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_ReferenceError.py
    print 'AFTER:', p.name
ReferenceError: weakly-referenced object no longer exists
```

34.3.16 RuntimeError

RuntimeError在没有其它具体的异常可用时被引发． 解释器本身并不会经常引发这个异常，但是一些用户代码可能会．

34.3.17 StopIteration

当一个迭代器完成后，它的 `next()` 方法引发StopIteration． 这个异常不被认为是错误．

```
l=[0,1,2]
i=iter(l)

print i
print i.next()
print i.next()
print i.next()
print i.next()
```

```
$ python exceptions_StopIteration.py
<listiterator object at 0x808b0>
0
1
2
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_StopIteration.py"
    print i.next()
StopIteration
```

34.3.18 SyntaxError

分析器发现无法理解的源码时引发SyntaxError． 可能是在引入一个模块，调用exec，或 `eval()` 时发生． 异常的属性确切找到输入的哪一部分导致了异常．

```
try:
    print eval('five times three')
except SyntaxError, err:
    print 'Syntax error %s (%s-%s): %s' % \
```

```
            (err.filename, err.lineno, err.offset, err.text)
    print err


$ python exceptions_SyntaxError.py
Syntax error <string> (1-10): five times three
invalid syntax (<string>, line 1)
```

34.3.19 SystemError

解释器本身发生错误，并且有可能继续正常运行时，引发SystemError. SystemError可能表示解释器本身的bug，并且需要报告给维护者.

34.3.20 SystemExit

当调用sys.exit()时，引发SystemExit，而不是立即退出. 这样允许 try:finally 代码块的完成清理工作，以及调用者(比如调试器和测试框架)捕获异常，从而避免退出.

34.3.21 TypeError

连接类型不正确的对象，或使用类型不正确的对象调用函数时，引发TypeError.

```
result = ('tuple',) + 'string'


$ python exceptions_TypeError.py
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_TypeError.py", li
    result = ('tuple',) + 'string'
TypeError: can only concatenate tuple (not "str") to tuple
```

34.3.22 UnboundLocalError

UnboundLocalError是NameError的一种，针对本地变量名.

```
def throws_global_name_error():
    print unknown_global_name

def throws_unbound_local():
    local_val = local_val + 1
    print local_val

try:
    throws_global_name_error()
except NameError, err:
    print 'Global name error:', err

try:
    throws_unbound_local()
except UnboundLocalError, err:
    print 'Local name error:', err
```

全局NameError和UnboundLocal的区别是使用变量名的方式. 因为变量名''local_val''出现在表达式的左边，所以被解释为一个本地变量名.

```
$ python exceptions_UnboundLocalError.py
Global name error: global name 'unknown_global_name' is not defined
Local name error: local variable 'local_val' referenced before assignment
```

34.3.23 UnicodeError

UnicodeError是ValueError的子类, 当出现Unicode问题时引发. UnicodeError有不同的子类: UnicodeEncodeError, UnicodeDecodeError和UnicodeTranslateError.

34.3.24 ValueError

当一个函数得到一个类型正确但是非有效值时, 引发ValueError.

```
print chr(1024)
```

```
$ python exceptions_ValueError.py
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_ValueError.py", l
    print chr(1024)
ValueError: chr() arg not in range(256)
```

34.3.25 ZeroDivisionError

当0作为一个除法操作的分母时, 引发ZeroDivisionError.

```
print 1/0
```

```
$ python exceptions_ZeroDivisionError.py
Traceback (most recent call last):
  File "/Users/dhellmann/Documents/PyMOTW/in_progress/exceptions/PyMOTW/exceptions/exceptions_ZeroDivisionError
    print 1/0
ZeroDivisionError: integer division or modulo by zero
```

34.4 Warning列表

还有一些异常用来在:mod:*warnings* 模块中使用.

- Warning: 所有警告的基类.
- UserWarning: 所有来自用户代码的警告的基类.
- DeprecationWarning: 用在不再被维护的特性上.
- PendingDeprecationWarning: 用在很快就会不推荐使用的特性上.
- SyntaxWarning: 用在值得商榷的表达式上.
- RuntimeWarning: 运行时可能发生问题的事件的警告.
- FutureWarning: 对于语言或库的改变会在之后实现时警告.
- ImportWarning: 导入一个模块的问题的警告.
- UnicodeWarning: 关于unicode文本的问题的警告.

34.5 参考

- exceptions

CHAPTER

THIRTYFIVE

PYMOTW: ITERTOOLS

- 模块： itertools
- 目的： 用于高效循环的迭代函数
- python版本： 2.3+

35.1 描述

这个模块中提供的函数具有和''lazy functional programming language'' Haskell 和 SML 相似的特点． 他们都是为了跑得更快和更有效的使用内存． 但他们也被牵扯在一起以表示更为复杂的迭代算法．

由于某些原因， 基于迭代的代码可能更优于使用列表的代码． 由于数据只有在需要它的时候才产生，所以所有的数据不会同时被存储在内存中.节省内存使用可以减少数据的交换次数和其他大数据集操作的副作用， 从而提高性能．

以下所有的例子都是使用from itertools import * 来导入itertools的．

35.2 合并和切分迭代器

chain() 函数将多个迭代器作为参数， 但只返回单个迭代器， 它产生所有参数迭代器的内容， 就好像他们是来自于一个单一的序列．

```
for i in chain([1, 2, 3], ['a', 'b', 'c']):
    print i


$ python itertools_chain.py
1
2
3
a
b
c
```

izip() 函数返回一个合并了多个迭代器为一个元组的迭代器． 它类似于内置函数zip()， 只是它返回的是一个迭代器而不是一个列表．

```
for i in izip([1, 2, 3], ['a', 'b', 'c']):
    print i
```

```
$ python itertools_izip.py
(1, 'a')
(2, 'b')
(3, 'c')
```

islice() 函数返回的迭代器是返回了输入迭代器根据索引来选取的项.

```
print 'Stop at 5:'
for i in islice(count(), 5):
    print i


class count(__builtin__.object)
  |  count([firstval]) --> count object
  |
  |  Return a count object whose .next() method returns consecutive
  |  integers starting from zero or, if specified, from firstval.


Stop at 5:
0
1
2
3
4
```

它可以使用和列表的slice操作相同的参数: start, stop和step. start和step参数是可选的.

```
print 'Start at 5, Stop at 10:'
for i in islice(count(), 5, 10):
    print i


Start at 5, Stop at 10:
5
6
7
8
9
```

```
print 'By tens to 100:'
for i in islice(count(), 0, 100, 10):
    print i


By tens to 100:
0
10
20
30
40
50
60
70
80
90
```

tee() 函数返回一些基于单个原始输入的独立迭代器(默认为2). 它和Unix上的tee工具有点语义相似, 也就是说它们都重复读取输入设备中的值并将值写入到一个命名文件和标准输出中.

```
r = islice(count(), 5)
i1, i2 = tee(r)

for i in i1:
    print 'i1:', i
for i in i2:
    print 'i2:', i


$ python itertools_tee.py
i1: 0
i1: 1
i1: 2
i1: 3
i1: 4
i2: 0
i2: 1
i2: 2
i2: 3
i2: 4
```

因为 `tee()` 新建的迭代器共享了输入，所以你就不需要使用原始的迭代器． 如果你使用了原始输入中的值，新的迭代器就不会产生对应的值：

```
r = islice(count(), 5)
i1, i2 = tee(r)

for i in r:
    print 'r:', i
    if i > 1:
        break

for i in i1:
    print 'i1:', i
for i in i2:
    print 'i2:', i


$ python itertools_tee_error.py
r: 0
r: 1
r: 2
i1: 3
i1: 4
i2: 3
i2: 4
```

35.3 转换输入

`imap()` 函数返回一个迭代器，它是调用了一个其值在输入迭代器上的函数，返回结果． 它类似于内置函数 `map()` ，只是前者在任意输入迭代器结束后就停止(而不是插入None值来补全所有的输入)．

在下面的第一个例子中，lambda函数将输入的值乘上2：

```
print 'Doubles:'
for i in imap(lambda x:2*x, xrange(5)):
    print i
```

```
$ python itertools_imap.py
Doubles:
0
2
4
6
8
```

在第二个例子中，lambda函数将2个参数相乘，这两个参数各自取自两个独立的迭代器并返回一个原始参数和计算结果的元组.

```
print 'Multiples:'
for i in imap(lambda x,y:(x, y, x*y), xrange(5), xrange(5,10)):
    print '%d * %d = %d' % i
```

```
Multiples:
0 * 5 = 0
1 * 6 = 6
2 * 7 = 14
3 * 8 = 24
4 * 9 = 36
```

starmap() 函数类似于 imap() ，但是在从多个迭代器中构造元组时，它先将各个项切分成单个迭代器并将它作为参数以*语法传递给映射函数. imap() 的映射函数被称为f(i1, i2)，startmap() 的映射函数被称为f(*i).

```
$ python itertools_starmap.py
0 * 5 = 0
1 * 6 = 6
2 * 7 = 14
3 * 8 = 24
4 * 9 = 36
```

35.4 产生新值

count() 函数返回一个不断产生连续整数的迭代器. 第一个数可以由参数指定，默认为0. 它没有上届参数(可参见内置函数 xrange() ，它更好的控制结果集). 在下面的例子中，迭代器由于参数列表结束而停止.

```
for i in izip(count(1), ['a', 'b', 'c']):
     print i
```

```
$ python itertools_count.py
(1, 'a')
(2, 'b')
(3, 'c')
```

cycle() 函数返回一个不断重复参数内容的迭代器. 由于它必须记住整个输入迭代器的内容，所以如果输入迭代器很长的话，它可能会消耗大量的内存. 在下面的例子中，一个计数变量用于在一定数量的循环后，跳出循环.

```
i = 0
for item in cycle(['a', 'b', 'c']):
    i += 1
    if i == 10:
```

```
        break
    print (i, item)
```

```
$ python itertools_cycle.py
(1, 'a')
(2, 'b')
(3, 'c')
(4, 'a')
(5, 'b')
(6, 'c')
(7, 'a')
(8, 'b')
(9, 'c')
```

repeat() 函数返回一个每次都产生相同值的迭代器． 它也是永远继续的，除非你设置了times参数来限制．

```
for i in repeat('over-and-over', 5):
    print i
```

```
$ python itertools_repeat.py
over-and-over
over-and-over
over-and-over
over-and-over
over-and-over
```

当其他迭代器使用的是一个固定值时，将 repeat() 和 izip() 或 imap() 联合起来使用是非常有用的．

```
for i, s in izip(count(), repeat('over-and-over', 5)):
     print i, s
```

```
$ python itertools_repeat_izip.py
0 over-and-over
1 over-and-over
2 over-and-over
3 over-and-over
4 over-and-over
```

```
$ python itertools_repeat_imap.py
2 * 0 = 0
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
```

35.5 过滤

dropwhile() 函数返回一个当条件为false之后的输入迭代器中剩余元素的迭代器． 它不过滤输入迭代器中的每一个项；在条件为false之后的第一次，返回迭代器中剩下来的项．

```
def should_drop(x):
    print 'Testing:', x
    return (x<1)

for i in dropwhile(should_drop, [ -1, 0, 1, 2, 3, 4, 1, -2 ]):
    print 'Yielding:', i
```

```
$ python itertools_dropwhile.py
Testing: -1
Testing: 0
Testing: 1
Yielding: 1
Yielding: 2
Yielding: 3
Yielding: 4
Yielding: 1
Yielding: -2
```

和 dropwhile() 相反的是， tabkewhile() ， 它返回的是一个产生输入迭代器中只要测试函数返回true的项的迭代器.

```
def should_take(x):
    print 'Testing:', x
    return (x<2)

for i in takewhile(should_take, [ -1, 0, 1, 2, 3, 4, 1, -2 ]):
    print 'Yielding:', i
```

```
$ python itertools_takewhile.py
Testing: -1
Yielding: -1
Testing: 0
Yielding: 0
Testing: 1
Yielding: 1
Testing: 2
```

ifilter() 返回的是迭代器类似于针对列表的内置函数 filter() ， 它只包括当测试函数返回true时的项. 它不同于 dropwhile() 的是每个项是在被返回之前进行测试的.

```
def check_item(x):
    print 'Testing:', x
    return (x<1)

for i in ifilter(check_item, [ -1, 0, 1, 2, 3, 4, 1, -2 ]):
    print 'Yielding:', i
```

```
$ python itertools_ifilter.py
Testing: -1
Yielding: -1
Testing: 0
Yielding: 0
Testing: 1
Testing: 2
Testing: 3
Testing: 4
Testing: 1
```

```
Testing: -2
Yielding: -2
```

和 ifilter() 函数相反的是，ifilterfalse() 返回一个包含那些测试函数返回false的项的迭代器.

```
def check_item(x):
    print 'Testing:', x
    return (x<1)

for i in ifilterfalse(check_item, [ -1, 0, 1, 2, 3, 4, 1, -2 ]):
    print 'Yielding:', i
```

```
$ python itertools_ifilterfalse.py Testing: -1
Testing: 0
Testing: 1
Yielding: 1
Testing: 2
Yielding: 2
Testing: 3
Yielding: 3
Testing: 4
Yielding: 4
Testing: 1
Yielding: 1
Testing: -2
```

35.6 分组数据

groupby() 函数返回一个产生按照key进行分组后的值集合的迭代器.

下面的例子来自于标准库文档，它表明怎样将一个字典根据值将关键字分组.

```
from itertools import *
from operator import itemgetter

d = dict(a=1, b=2, c=1, d=2, e=1, f=2, g=3)
di = sorted(d.iteritems(), key=itemgetter(1))
for k, g in groupby(di, key=itemgetter(1)):
    print k, map(itemgetter(0), g)
```

```
$ python itertools_groupby.py
1 ['a', 'c', 'e']
2 ['b', 'd', 'f']
3 ['g']
```

下面一个更复杂的例子说明了如何基于一些属性来对值进行分组的． 注意了，输入的序列需要按照关键字进行排序，这样就可以得到预期的分组结果了：

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Point(%s, %s)' % (self.x, self.y)
```

```
    def __cmp__(self, other):
        return cmp((self.x, self.y), (other.x, other.y)) ##

# Create a dataset of Point instances
data = list(imap(Point, cycle(islice(count(), 3)), islice(count(), 10),))
print 'Data:', data
print

# Try to group the unsorted data based on X values
print 'Grouped, unsorted:'
for k, g in groupby(data, lambda o:o.x):
    print k, list(g)
print

# Sort the data
data.sort()
print 'Sorted:', data
print

# Group the sorted data based on X values
print 'Grouped, sorted:'
for k, g in groupby(data, lambda o:o.x):
    print k, list(g)
print
```

```
$ python itertools_groupby_seq.py
Data: [Point(0, 0), Point(1, 1), Point(2, 2), Point(0, 3),
  Point(1, 4), Point(2, 5), Point(0, 6), Point(1, 7),
  Point(2, 8), Point(0, 9)]

Grouped, unsorted:
0 [Point(0, 0)]
1 [Point(1, 1)]
2 [Point(2, 2)]
0 [Point(0, 3)]
1 [Point(1, 4)]
2 [Point(2, 5)]
0 [Point(0, 6)]
1 [Point(1, 7)]
2 [Point(2, 8)]
0 [Point(0, 9)]

Sorted: [Point(0, 0), Point(0, 3), Point(0, 6), Point(0, 9),
  Point(1, 1), Point(1, 4), Point(1, 7), Point(2, 2),
  Point(2, 5), Point(2, 8)]

Grouped, sorted:
0 [Point(0, 0), Point(0, 3), Point(0, 6), Point(0, 9)]
1 [Point(1, 1), Point(1, 4), Point(1, 7)]
2 [Point(2, 2), Point(2, 5), Point(2, 8)]
```

35.7 参考

- The Standard ML Basis Library
- Definition of Haskell and the Standard Libraries

CHAPTER

THIRTYSIX

PYMOTW: ZIPFILE

- 模块： zipfile
- 目的： 读写ZIP档案文件.
- python版本： 1.6+

zipfile模块能用来处理ZIP档案文件.

36.1 局限

zipfile模块不支持附加评论的或者多磁盘ZIP文件，支持大于4GB使用ZIP64扩展的ZIP文件.

36.2 测试ZIP文件

is_zipfile() 函数返回一个布尔值，判断给定的文件是否是一个有效的ZIP文件.

```
import zipfile

for filename in [ 'README.txt', 'example.zip', 'bad_example.zip', 'notthere.zip' ]:
    print '%20s %s' % (filename, zipfile.is_zipfile(filename))
```

注意：如果文件不存在，is_zipfile() 返回False.

```
$ python zipfile_is_zipfile.py
README.txt False
example.zip True
bad_example.zip False
notthere.zip False
```

36.3 从ZIP存档中读取元数据

使用ZipFile类直接处理ZIP存档，它支持从现有存档中读取数据也支持向存档中加入其它文件更改存档.

使用 namelist() 函数读取现有存档中所有文件的名字.

```
import zipfile

zf = zipfile.ZipFile('example.zip', 'r')
print zf.namelist()
```

返回的是存档内容名字的字符串列表.

```
$ python zipfile_namelist.py
['README.txt']
```

然而，名字列表只是存档中可用信息的一小部分，使用 `infolist()` 或者 `getinfo()` 方法来访问存档内容的所有元数据.

```
import datetime
import zipfile

def print_info(archive_name):
    zf = zipfile.ZipFile(archive_name)
    for info in zf.infolist():
        print info.filename
        print '\tComment:\t', info.comment
        print '\tModified:\t', datetime.datetime(*info.date_time)
        print '\tSystem:\t\t', info.create_system, '(0 = Windows, 3 = Unix)'
        print '\tZIP version:\t', info.create_version
        print '\tCompressed:\t', info.compress_size, 'bytes'
        print '\tUncompressed:\t', info.file_size, 'bytes'
        print

if __name__ == '__main__':
    print_info('example.zip')
```

除了这儿输出的一些信息外，还有别的东西，但是需要仔细阅读ZIP文件说明书上的 PKZIP应用注释 才能将其解密成有用的东西.

```
$ python zipfile_infolist.py
README.txt
  Comment:
  Modified: 2007-12-16 10:08:52
  System: 3 (0 = Windows, 3 = Unix)
  ZIP version: 23
  Compressed: 63 bytes
  Uncompressed: 75 bytes
```

如果你已经知道了存档中各文件的名字，你也可以通过 `getinfo()` 方法获得它的ZipInfo对象.

```
import zipfile

zf = zipfile.ZipFile('example.zip')
for filename in [ 'README.txt', 'notthere.txt' ]:
    try:
        info = zf.getinfo(filename)
    except KeyError:
        print 'ERROR: Did not find %s in zip file' % filename
    else:
        print '%s is %d bytes' % (info.filename, info.file_size)
```

如果存档中的某个文件不存在，`getinfo()` 方法会产生一个KeyError.

```
$ python zipfile_getinfo.py
README.txt is 75 bytes
ERROR: Did not find notthere.txt in zip file
```

36.4 从ZIP档案中提取文件

为了访问存档文件的数据，使用 `read()` 方法，并将该成员的名字传递给它.

```
import zipfile

zf = zipfile.ZipFile('example.zip')
for filename in [ 'README.txt', 'notthere.txt' ]:
    try:
        data = zf.read(filename)
    except KeyError:
        print 'ERROR: Did not find %s in zip file' % filename
    else:
        print filename, ':'
        print repr(data)
        print
```

必要时，数据会自动解压缩.

```
$ python zipfile_read.py
README.txt :
'The examples for the zipfile module use this file and example.zip as data.\n'

ERROR: Did not find notthere.txt in zip file
```

36.5 创建一个新的档案

为了创建一个新的档案，以‘w’模式简单实例化ZipFile对象. 档案中任何现有文件会被清空，开始新档案. 使用 `write()` 方法可以在档案中增加文件.

```
from zipfile_infolist import print_info
import zipfile

print 'creating archive'
zf = zipfile.ZipFile('zipfile_write.zip', mode='w')
try:
    print 'adding README.txt'
    zf.write('README.txt')
finally:
    print 'closing'
    zf.close()

print
print_info('zipfile_write.zip')
```

默认情况下，档案的文件不会被压缩：

```
$ python zipfile_write.py
creating archive
adding README.txt
closing

README.txt
  Comment:
  Modified: 2007-12-16 10:08:50
  System: 3 (0 = Windows, 3 = Unix)
  ZIP version: 20
```

```
  Compressed: 75 bytes
  Uncompressed: 75 bytes
```

zlib模块提供压缩功能． 如果zlib是可用的，你能使用zipfile.ZIP_DEFLATED对个人文件或者整个档案设置压缩模式． 默认压缩模式为zipfile.ZIP_STORED．

```
from zipfile_infolist import print_info
import zipfile
try:
    import zlib
    compression = zipfile.ZIP_DEFLATED
except:
    compression = zipfile.ZIP_STORED

modes = { zipfile.ZIP_DEFLATED: 'deflated',
    zipfile.ZIP_STORED: 'stored',
}

print 'creating archive'
zf = zipfile.ZipFile('zipfile_write_compression.zip', mode='w')
try:
    print 'adding README.txt with compression mode', modes[compression]
    zf.write('README.txt', compress_type=compression)
finally:
    print 'closing'
    zf.close()

print
print_info('zipfile_write_compression.zip')
```

这次，归档中的文件被压缩了：

```
$ python zipfile_write_compression.py creating archive
adding README.txt with compression mode deflated
closing

README.txt
  Comment:
  Modified: 2007-12-16 10:08:50
  System: 3 (0 = Windows, 3 = Unix)
  ZIP version: 20
  Compressed: 63 bytes
  Uncompressed: 75 bytes
```

36.6 使用备选的存档成员名

传递arcname参数给 wirte() 可以很容易将一个文件添加到存档中，但命名不能是原始文件名．

```
from zipfile_infolist import print_info
import zipfile

zf = zipfile.ZipFile('zipfile_write_arcname.zip', mode='w')
try:
    zf.write('README.txt', arcname='NOT_README.txt')
finally:
    zf.close()
print_info('zipfile_write_arcname.zip')
```

在档案中，新的文件没有使用原来的文件名.

```
$ python zipfile_write_arcname.py
NOT_README.txt
  Comment:
  Modified: 2007-12-16 10:08:50
  System: 3 (0 = Windows, 3 = Unix)
  ZIP version: 20
  Compressed: 75 bytes
  Uncompressed: 75 bytes
```

36.7 从源而非文件上写数据

有时候，将那些不是来自现有文件的数据直接写入到ZIP档案中也是有必要的，而不是通过先把这些数据写入到一个文件中，再把这个文件添加到ZIP档案中，你可以使用writestr()函数将字符串字节流直接写入到档案中.

```
from zipfile_infolist import print_info
import zipfile

msg = 'This data did not exist in a file before being added to the ZIP file'
zf = zipfile.ZipFile('zipfile_writestr.zip',
 mode='w',
 compression=zipfile.ZIP_DEFLATED,
)
try:
    zf.writestr('from_string.txt', msg)
finally:
    zf.close()

print_info('zipfile_writestr.zip')

zf = zipfile.ZipFile('zipfile_writestr.zip', 'r')
print zf.read('from_string.txt')
```

上述实例中，我在ZipFile中使用compress参数来压缩数据，但 writestr() 方法中不支持该参数.

```
$ python zipfile_writestr.py
from_string.txt
  Comment:
  Modified: 2007-12-16 11:38:14
  System: 3 (0 = Windows, 3 = Unix)
  ZIP version: 20
  Compressed: 62 bytes
  Uncompressed: 68 bytes

This data did not exist in a file before being added to the ZIP file
```

36.8 通过ZipInfo实例写入

默认情况下，当你在档案中加入文件或者字符串时，需要计算修改日期. 当使用 writestr() 方法时，也需要传递一个ZipInfo实例给它，该实例包含了修改日期和别的自定义元数据.

```
import time
import zipfile
from zipfile_infolist import print_info
```

```
msg = 'This data did not exist in a file before being added to the ZIP file'
zf = zipfile.ZipFile('zipfile_writestr_zipinfo.zip', mode='w',)
try:
    info = zipfile.ZipInfo('from_string.txt', date_time=time.localtime(time.time()),)
    info.compress_type=zipfile.ZIP_DEFLATED
    info.comment='Remarks go here'
    info.create_system=0
    zf.writestr(info, msg)
finally:
    zf.close()

print_info('zipfile_writestr_zipinfo.zip')
```

在这个例子中，我修改时间为当前时间． 压缩数据，赋create_system值为0． 还增添了评论．

```
$ python zipfile_writestr_zipinfo.pyfrom_string.txt
  Comment: Remarks go here
  Modified: 2007-12-16 11:44:14
  System: 0 (0 = Windows, 3 = Unix)
  ZIP version: 20
  Compressed: 62 bytes
      Uncompressed: 68 bytes
```

36.9 追加文件

除了创建一个新档案之外，还可以在现有档案上追加一个文件或在一个现有文件(如a.exe，自解压档案文件)的末尾增加一个档案文件． 使用模式‘a‘打开文件以便追加．

```
from zipfile_infolist import print_info
import zipfile

print 'creating archive'
zf = zipfile.ZipFile('zipfile_append.zip', mode='w')
try:
    zf.write('README.txt')
finally:
    zf.close()

print
print_info('zipfile_append.zip')

print 'appending to the archive'
zf = zipfile.ZipFile('zipfile_append.zip', mode='a')
try:
    zf.write('README.txt', arcname='README2.txt')
finally:
    zf.close()

print
print_info('zipfile_append.zip')
```

结果档案有2个文件：

```
$ python zipfile_append.py
creating archive

README.txt
  Comment:
```

```
  Modified: 2007-12-16 10:08:50
  System: 3 (0 = Windows, 3 = Unix)
  ZIP version: 20
  Compressed: 75 bytes
  Uncompressed: 75 bytes

appending to the archive

README.txt
  Comment:
  Modified: 2007-12-16 10:08:50
  System: 3 (0 = Windows, 3 = Unix)
  ZIP version: 20
  Compressed: 75 bytes
  Uncompressed: 75 bytes

README2.txt
  Comment:
  Modified: 2007-12-16 10:08:50
  System: 3 (0 = Windows, 3 = Unix)
  ZIP version: 20
  Compressed: 75 bytes
  Uncompressed: 75 bytes
```

36.10 Python ZIP档案

如果存档出现在sys.path中，Python 2.3及以后版本都有能力从ZIP档案内部引入模块。使用类zpfile.PyZipFile可以构造一个模块来适合这种用法。当你使用其他方法 `writepy()` 时,PyZipFile浏览目录寻找.py文件，并且将关联文件 `.pyo` 或 `.pyc` 加入到档案中。如果两者都不存在，则生成一个.pyc文件，并将其加入到档案中.

```
import sys
import zipfile

if __name__ == '__main__':
    zf = zipfile.PyZipFile('zipfile_pyzipfile.zip', mode='w') ##             py
    try:
        zf.debug = 3
        print 'Adding python files'
        zf.writepy('.')
    finally:
        zf.close()
        for name in zf.namelist():
            print name

    print
    sys.path.insert(0, 'zipfile_pyzipfile.zip')
    import zipfile_pyzipfile
    print 'Imported from:', zipfile_pyzipfile.__file__
```

当我设置PyZipFile的属性debug=3，就激活了verbose debugging，这在编译每一个.py文件时可以看到.

```
$ python zipfile_pyzipfile.py
Adding python files
Adding package in . as .
Compiling ./__init__.py
Adding ./__init__.pyc
```

```
Compiling ./zipfile_append.py
Adding ./zipfile_append.pyc
Compiling ./zipfile_getinfo.py
Adding ./zipfile_getinfo.pyc
Compiling ./zipfile_infolist.py
Adding ./zipfile_infolist.pyc
Compiling ./zipfile_is_zipfile.py
Adding ./zipfile_is_zipfile.pyc
Compiling ./zipfile_namelist.py
Adding ./zipfile_namelist.pyc
Compiling ./zipfile_printdir.py
Adding ./zipfile_printdir.pyc
Compiling ./zipfile_pyzipfile.py
Adding ./zipfile_pyzipfile.pyc
Compiling ./zipfile_read.py
Adding ./zipfile_read.pyc
Compiling ./zipfile_write.py
Adding ./zipfile_write.pyc
Compiling ./zipfile_write_arcname.py
Adding ./zipfile_write_arcname.pyc
Compiling ./zipfile_write_compression.py
Adding ./zipfile_write_compression.pyc
Compiling ./zipfile_writestr.py
Adding ./zipfile_writestr.pyc
Compiling ./zipfile_writestr_zipinfo.py
Adding ./zipfile_writestr_zipinfo.pyc
__init__.pyc
zipfile_append.pyc
zipfile_getinfo.pyc
zipfile_infolist.pyc
zipfile_is_zipfile.pyc
zipfile_namelist.pyc
zipfile_printdir.pyc
zipfile_pyzipfile.pyc
zipfile_read.pyc
zipfile_write.pyc
zipfile_write_arcname.pyc
zipfile_write_compression.pyc
zipfile_writestr.pyc
zipfile_writestr_zipinfo.pyc

Imported from: zipfile_pyzipfile.zip/zipfile_pyzipfile.pyc
```

36.11 参考

- PKZIP Application Note
- zipimport module

CHAPTER

THIRTYSEVEN

PYMOTW: BASEHTTPSERVER

- 模块： BaseHTTPServer
- 目的： 提供实现web服务的基础类.
- python版本： 1.4+

BaseHTTPServer模块包括了能够构成基本的web服务器的类.

37.1 描述

BaseHTTPServer使用 SocketServer 中的类来创建HTTP服务基本类， HTTPServer可以被直接使用， 但是其中的BaseHTTPRequestHandler需要被扩展去处理各种协议方法(如GET， POST等).

37.2 简单的GET请求例子

实现 do_METHOD() 方法可以让你的请求类支持HTTP方法， 这里的METHOD用具体的HTTP方法名字代替. 例如， do_GET()， do_POST() 等. 为了保持一致， 这些方法不含参数. 用于请求的所有的参数被BaseHTTPRequestHandler解析， 然后被存储在一个实例属性集合中， 这样你可以很方便地检索它们.

下面的例子中， 请求处理对象说明了如何返回一个响应对象给客户端， 一些本地属性在构造响应对象中很有用的.

```
from BaseHTTPServer import BaseHTTPRequestHandler
import urlparse

class GetHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        parsed_path = urlparse.urlparse(self.path)
        message = '\n'.join([
          'CLIENT VALUES:',
          'client_address=%s (%s)' % (self.client_address, self.address_string()),
          'command=%s' % self.command,
          'path=%s' % self.path,
          'real path=%s' % parsed_path.path,
          'query=%s' % parsed_path.query,
          'request_version=%s' % self.request_version,
          '',
          'SERVER VALUES:',
          'server_version=%s' % self.server_version,
          'sys_version=%s' % self.sys_version,
          'protocol_version=%s' % self.protocol_version,
          '',
```

```
        ])
        self.send_response(200)
        self.end_headers()

        self.wfile.write(message)

        return
```

消息正文被组装好后就写入到self.wfile中，这个文件处理对象封装了整个响应socket．每个响应对象需要一个响应代码，它可以通过self.send_response()来设置．如果使用了一个错误代码(如404，501等)，那么对应的默认错误信息会被包含在消息头中，或者，你能使用错误码来传递信息．

在服务器中运行请求处理对象，将这个请求处理对象传递给HTTPServer构造器．

```
if __name__ == '__main__':
    from BaseHTTPServer import HTTPServer
    server = HTTPServer(('localhost', 8080), GetHandler)
    print 'Starting server, use <Ctrl-C> to stop'
    server.serve_forever()
```

接下来启动服务器：

```
$ python BaseHTTPServer_GET.py
Starting server, use <Ctrl-C> to stop
```

在另外一终端中，使用curl访问：

```
$ curl -i http://localhost:8080/?foo=barHTTP/1.0 200 OK
Server: BaseHTTP/0.3 Python/2.5.1
Date: Sun, 09 Dec 2007 16:00:34 GMT

CLIENT VALUES:
client_address=('127.0.0.1', 51275) (localhost)
command=GET
path=/?foo=bar
real path=/
query=foo=bar
request_version=HTTP/1.1

SERVER VALUES:
server_version=BaseHTTP/0.3
sys_version=Python/2.5.1
protocol_version=HTTP/1.0
```

37.3 线程和进程

HTTPServer是SocketServer.TCPServer的一个简单子类．它不使用多线程或多进程来处理请求．要增加多线程和多进程，可以使用SocketServer中的合适的混用类来创建一个新的类．

```
from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler
from SocketServer import ThreadingMixIn
import threading

class Handler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_response(200)
        self.end_headers()
```

```
        message = threading.currentThread().getName() ##  threading
        self.wfile.write(message)
        self.wfile.write('\n')
        return

class ThreadedHTTPServer(ThreadingMixIn, HTTPServer):
    """Handle requests in a separate thread."""

if __name__ == '__main__':
    server = ThreadedHTTPServer(('localhost', 8080), Handler)
    print 'Starting server, use <Ctrl-C> to stop'
    server.serve_forever()
```

每次当一个请求过来的时候，一个新的线程或进程会被创建来处理它：

```
$ curl http://localhost:8080/
Thread-1
$ curl http://localhost:8080/
Thread-2
$ curl http://localhost:8080/
Thread-3
```

如果把上面的ThreadingMixIn换成ForkingMixIn，也可以获得类似的结果，但是后者是使用了独立的进程而不是线程.

37.4 POST

支持POST请求需要多一点的工作，因为基本类不会为我们解析表单数据. cgi 模块提供了用于解析表单的FieldStorage类,只要我们提供正确的输入格式.

```
from BaseHTTPServer import BaseHTTPRequestHandler
import cgi

class PostHandler(BaseHTTPRequestHandler):

    def do_POST(self):
        # Parse the form data posted
        form = cgi.FieldStorage(
        fp=self.rfile,
        headers=self.headers,
        environ={'REQUEST_METHOD':'POST',
          'CONTENT_TYPE':self.headers['Content-Type'],
        })

        # Begin the response
        self.send_response(200)
        self.end_headers()
        self.wfile.write('Client: %s\n' % str(self.client_address))
        self.wfile.write('Path: %s\n' % self.path)
        self.wfile.write('Form data:\n')


        # Echo back information about what was posted in the form
        for field in form.keys():
            field_item = form[field]
            if field_item.filename:
                # The field contains an uploaded file
                file_data = field_item.file.read()
                file_len = len(file_data)
```

```
                del file_data
                self.wfile.write('\tUploaded %s (%d bytes)\n' % (field, file_len))
            else:
                # Regular form value
                self.wfile.write('\t%s=%s\n' % (field, form[field].value))
        return

if __name__ == '__main__':
    from BaseHTTPServer import HTTPServer
    server = HTTPServer(('localhost', 8080), PostHandler)
    print 'Starting server, use <Ctrl-C> to stop'
    server.serve_forever()
```

再次使用curl， 我们可以包含表单数据， 它自动以POST方式发送． 最后的参数， -F datafile=@BaseHTTPServer_GET.py， 将文件BaseHTTPServer_GET.py的内容发送， 从而表示已从表单中读取了文件数据．

```
$ curl http://localhost:8080/ -F name=dhellmann -F foo=bar -F datafile=@BaseHTTPServer_GET.py
Client: ('127.0.0.1', 51128)
Path: /
Form data:
        name=dhellmann
        foo=bar
        Uploaded datafile (2222 bytes)
```

37.5 Errors

错误处理可以使用 send_error() 方法． 简单的传递给它合适的错误代码以及一个可选的错误信息，那么就会为你生成整个响应对象(包括头， 状态码， 消息体)．

```
from BaseHTTPServer import BaseHTTPRequestHandler

class ErrorHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_error(404)
        return

if __name__ == '__main__':
    from BaseHTTPServer import HTTPServer
    server = HTTPServer(('localhost', 8080), ErrorHandler)
    print 'Starting server, use <Ctrl-C> to stop'
    server.serve_forever()
```

在这种情况下， 一直返回404错误．

```
$ curl -i http://localhost:8080/
HTTP/1.0 404 Not Found
Server: BaseHTTP/0.3 Python/2.5.1
Date: Sun, 09 Dec 2007 15:49:44 GMT
Content-Type: text/html
Connection: close

<head>
<title>Error response</title>
</head>
<body>
<h1>Error response</h1>
```

```
<p>Error code 404.
<p>Message: Not Found.
<p>Error code explanation: 404 = Nothing matches the given URI.
</body>
```

CHAPTER

THIRTYEIGHT

PYMOTW: PPRINT

- 模块： pprint
- 目的： 用于更加好看的显示数据
- python版本： 1.4+

pprint模块让你的数据的显示结构更加美观.

38.1 描述

pprint模块中使用的格式化可以按照一种格式正确的显示数据， 这种格式即可被解析器解析， 又很易读. 输出保存在一个单行内， 但如果有必要， 在分割多行数据时也可使用缩进表示.

这里的例子全部依赖于pprint_data.py这个脚本， 它包含:

```
data = [ (i, { 'a':'A',
 'b':'B',
 'c':'C',
 'd':'D',
 'e':'E',
 'f':'F',
 'g':'G',
 'h':'H',
 })
 for i in xrange(3)
 ]
```

38.2 输出

使用这个模块的最简单方式是使用 pprint() 函数. 它格式化你的对象并作为参数写入到数据流中(默认为sys.stdout)中.

```
from pprint import pprint
from pprint_data import data

print 'PRINT:'
print data
print
print 'PPRINT:'
pprint(data)
```

```
$ python pprint_pprint.py
PRINT:
[(0, {'a': 'A', 'c': 'C', 'b': 'B', 'e': 'E', 'd': 'D', 'g': 'G', 'f': 'F', 'h': 'H'}), (1, {'a': 'A', 'c': 'C'


PPRINT:
[(0,
 {'a': 'A',
  'b': 'B',
  'c': 'C',
  'd': 'D',
  'e': 'E',
  'f': 'F',
  'g': 'G',
  'h': 'H'}),
 (1,
 {'a': 'A',
  'b': 'B',
  'c': 'C',
  'd': 'D',
  'e': 'E',
  'f': 'F',
  'g': 'G',
  'h': 'H'}),
 (2,
 {'a': 'A',
  'b': 'B',
  'c': 'C',
  'd': 'D',
  'e': 'E',
  'f': 'F',
  'g': 'G',
  'h': 'H'})]
```

38.3 格式化

如果你需要格式化一个数据结构， 但不想直接写入到流中(比如， 为了记录日志)， 你可以使用 pformat() 来构建一个字符串传递给其他函数.

```
import logging
from pprint import pformat
from pprint_data import data

logging.basicConfig(level=logging.DEBUG,
 format='%(asctime)s %(levelname)-8s %(message)s',
)

logging.debug('Logging pformatted data')
logging.debug(pformat(data))
```

```
$ python pprint_pformat.py
2007-10-21 18:10:32,881 DEBUG Logging pformatted data
2007-10-21 18:10:32,884 DEBUG [(0,
{'a': 'A',
 'b': 'B',
 'c': 'C',
 'd': 'D',
 'e': 'E',
```

```
 'f': 'F',
 'g': 'G',
 'h': 'H'}),
 (1,
 {'a': 'A',
 'b': 'B',
 'c': 'C',
 'd': 'D',
 'e': 'E',
 'f': 'F',
 'g': 'G',
 'h': 'H'}),
 (2,
 {'a': 'A',
 'b': 'B',
 'c': 'C',
 'd': 'D',
 'e': 'E',
 'f': 'F',
 'g': 'G',
 'h': 'H'})]
```

38.4 其他类

pprint() 中使用到的PrettyPrinter类也支持自定义类，前提是在自定义类中，定义了 __repr__ 方法.

```
from pprint import pprint

class node(object):
    def __init__(self, name, contents=[]):
        self.name = name
        self.contents = contents[:]

    def __repr__(self):
        return 'node(' + repr(self.name) + ', ' + repr(self.contents) + ')'

trees = [ node('node-1'),
    node('node-2', [ node('node-2-1')]),
    node('node-3', [ node('node-3-1')]),
]
pprint(trees)
```

```
$ python pprint_arbitrary_object.py
[node('node-1', []),
 node('node-2', [node('node-2-1', [])]),
 node('node-3', [node('node-3-1', [])])]
```

38.5 递归

递归的数据可用一个指向原始数据的引用来表示，具体的形式为 *<Recursion on typename with id=number>*. 例如:

```
local_data = [ 'a', 'b', 1, 2 ]
local_data.append(local_data)
```

```
print 'id(local_data) =>', id(local_data)
pprint(local_data)
```

```
$ python pprint_recursion.py
id(local_data) => 486936
['a', 'b', 1, 2, <Recursion on list with id=486936>]
```

38.6 限制嵌套输出

对于每一个深层次的数据结构，你可能不想输出所有的细节． 也可能无法适当的格式化数据，或者要格式化的文本很大而难控制，或者你可能需要全部使用． 在这种情况下，指定depth参数可控制嵌套数据结构显示的深度．

```
from pprint import pprint
from pprint_data import data

pprint(data, depth=1)
```

```
$ python pprint_depth.py
[(0, {...}), (1, {...}), (2, {...})]
```

38.7 控制输出宽度

默认的格式化文本输出宽度是80列． 指定 pprint() 中的width参数可以调整文本输出宽度．

```
from pprint import pprint
from pprint_data import data

for d in data:
    for c in 'defgh':
        del d[1][c]

for width in [ 80, 20, 5 ]:
    print 'WIDTH =', width
    pprint(data, width=width)
    print
```

注意： 当width被定义的太小而不能容纳所有的格式化数据时，行是不会被截断． 如果有出现无效的语法时，也不会被包裹起来．

```
$ python pprint_width.py
WIDTH = 80
[(0, {'a': 'A', 'b': 'B', 'c': 'C'}),
 (1, {'a': 'A', 'b': 'B', 'c': 'C'}),
 (2, {'a': 'A', 'b': 'B', 'c': 'C'})]

WIDTH = 20
[(0,
 {'a': 'A',
 'b': 'B',
 'c': 'C'}),
 (1,
 {'a': 'A',
```

```
 'b': 'B',
 'c': 'C'}),
 (2,
 {'a': 'A',
 'b': 'B',
 'c': 'C'})]

WIDTH = 5
[(0,
 {'a': 'A',
 'b': 'B',
 'c': 'C'}),
 (1,
 {'a': 'A',
 'b': 'B',
 'c': 'C'}),
 (2,
 {'a': 'A',
 'b': 'B',
 'c': 'C'})]
```

CHAPTER

THIRTYNINE

PYMOTW: SOCKETSERVER

- 模块： SocketServer
- 目的： 创建网络服务器.
- python版本： 1.4+

SocketServer模块是一个用于创建网络服务器的框架. 他提供了处理TCP, UDP, Unix流和Unix数据报的基本类和支持线程和进程服务器, 这依赖于具体的应用情况.

39.1 描述

SocketServer模块定义了处理同步网络请求(服务器请求处理时会阻塞直到请求完成)的类. 它也提供了一个接口类, 可以方便地将服务器转换使其对于每个请求使用独立线程或进程.

处理一个请求需要先将server(服务器)类和request handler(请求处理)类分开来. 服务器处理通信事宜(如列出一个socket, 接受连接等等), 而请求处理类处理''协议''事宜(解释来到的数据并处理他, 返回数据给客户端). 这种任务的划分意味着, 在许多情况下, 你可以简单地并且可以不加修改的使用一个现有的服务器类, 并为之提供一个符合用户特定需要的请求处理类.

39.2 服务器类型

在SocketServer模块中定义了5种不同的服务器类. BaseServer仅定义了API, 但没有实例化很多方法所以不能直接使用. TCPServer使用TCP/IP sockets来通信. UDPServer使用数据报sockets. UnixStreamServer和UnixDatagramServer使用Unix-domain sockets, 这仅在Unix平台上可用.

39.3 服务器对象

构造一个服务器对象, 需要传递给它一个监听请求的地址和一个请求处理类(不是实例). 地址格式根据服务器的类型和所使用的socket族. 细节可以参考 socket模块文档.

一旦一个服务器对象被实例化, 使用 handle_request() 或者 serve_forever() 来处理请求. serve_forever() 方法简单的在一个无穷循环中调用 handle_request(), 所以如果你需要在服务器中集成其他事件循环, 或者使用 select() 监视多个不同服务器的sockets, 你可以独立调用 handle_request(). 可以看下面的例子.

39.4 实现一个服务器

如果你要创建一个服务器, 它通常是复用现有的类并简单提供自定义的请求处理类. 如果这不能满足你的需求, 还可以使用BaseServer并在其子类中重载一些方法:

verify_request(request, client_address) - 返回True来处理请求，或者False表示忽略这个请求. 比如，你也可以拒绝从一个IP范围来的请求，假如你想要阻断某些客户端访问服务器.

process_request(request, client_address) - 它通常是调用finish_request()来完成实际工作. 但它也看创建一个独立的线程或进程，作为混合类来使用(如下).

finish_request(request, client_address) - 使用在服务器构造时指定的类来创建一个请求处理实例. 她调用请求处理类的 handle() 来处理请求.

39.5 请求处理者

请求处理者做了大部分的接受到达的请求和决定如何处理. 处理者应该实现socket层面的 *protocol* (例如, HTTP或XML-RPC). 请求处理者从到达的数据通道中读取请求, 处理它, 并写回一个response. 有下面的3个方法可以被重载.

setup() - 准备一个请求处理者. 在StreamRequestHandler中，例如，setup() 方法创建一个类文件对象用于读取和写入socket.

handle() - 做实际处理请求的工作. 解析到来的请求，处理数据并发送一个response.

finish() - 清除在setup()中创建的所有东西.

在很多情况下，你可以简单提供handle()方法就可.

39.6 Echo例子

在这个例子中，让我们看下一对简单的server/request处理对象，它们接受TCP连接，并将接收的数据返回给客户端. 例子代码中，唯一实际需要的方法是 EchoRequestHandler.handle()，但我已经重载上述描述的所有方法并插入了日志功能调用以便输出示例程序调用时的执行顺序.

```
import logging
import sys
import SocketServer

logging.basicConfig(level=logging.DEBUG,
  format='%(name)s: %(message)s',
)

class EchoRequestHandler(SocketServer.BaseRequestHandler):

    def __init__(self, request, client_address, server):
        self.logger = logging.getLogger('EchoRequestHandler')
        self.logger.debug('__init__')
        SocketServer.BaseRequestHandler.__init__(self, request, client_address, server)
        return

    def setup(self):
        self.logger.debug('setup')
        return SocketServer.BaseRequestHandler.setup(self)

    def handle(self):
        self.logger.debug('handle')

        # Echo the back to the client
        data = self.request.recv(1024)
        self.logger.debug('recv()->"%s"', data)
        self.request.send(data)
        return

    def finish(self):
```

```
        self.logger.debug('finish')
        return SocketServer.BaseRequestHandler.finish(self)

class EchoServer(SocketServer.TCPServer):

    def __init__(self, server_address, handler_class=EchoRequestHandler):
        self.logger = logging.getLogger('EchoServer')
        self.logger.debug('__init__')
        SocketServer.TCPServer.__init__(self, server_address, handler_class)
        return

    def server_activate(self):
        self.logger.debug('server_activate')
        SocketServer.TCPServer.server_activate(self)
        return

    def serve_forever(self):
        self.logger.debug('waiting for request')
        self.logger.info('Handling requests, press <Ctrl-C> to quit')
        while True:
            self.handle_request()
        return

    def handle_request(self):
        self.logger.debug('handle_request')
        return SocketServer.TCPServer.handle_request(self)

    def verify_request(self, request, client_address):
        self.logger.debug('verify_request(%s, %s)', request, client_address)
        return SocketServer.TCPServer.verify_request(self, request, client_address)

    def process_request(self, request, client_address):
        self.logger.debug('process_request(%s, %s)', request, client_address)
        return SocketServer.TCPServer.process_request(self, request, client_address)

    def server_close(self):
        self.logger.debug('server_close')
        return SocketServer.TCPServer.server_close(self)

    def finish_request(self, request, client_address):
        self.logger.debug('finish_request(%s, %s)', request, client_address)
        return SocketServer.TCPServer.finish_request(self, request, client_address)

    def close_request(self, request_address):
        self.logger.debug('close_request(%s)', request_address)
        return SocketServer.TCPServer.close_request(self, request_address)
```

这是一个简单的程序，他创建了服务器，在一个线程中运行，连接它能返回哪些方法被调用的信息.

```
if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0) # let the kernel give us a port
    server = EchoServer(address, EchoRequestHandler)
    ip, port = server.server_address # find out what port we were given

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True) # don't hang on exit
    t.start()

    logger = logging.getLogger('client')
```

```
    logger.info('Server on %s:%s', ip, port)

    # Connect to the server
    logger.debug('creating socket')
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    logger.debug('connecting to server')
    s.connect((ip, port))

    # Send the data
    message = 'Hello, world'
    logger.debug('sending data: "%s"', message)
    len_sent = s.send(message)

    # Receive a response
    logger.debug('waiting for response')
    response = s.recv(len_sent)
    logger.debug('response from server: "%s"', response)

    # Clean up
    logger.debug('closing socket')
    s.close()
    logger.debug('done')
    server.socket.close()
```

程序输出的结果如下：

```
$ python SocketServer_echo.py
EchoServer: __init__
EchoServer: server_activate
EchoServer: waiting for request
EchoServer: Handling requests, press to quit
EchoServer: handle_request
client: Server on 127.0.0.1:53477
client: creating socket
client: connecting to server
EchoServer: verify_request(, ('127.0.0.1', 53478))
EchoServer: process_request(, ('127.0.0.1', 53478))
EchoServer: finish_request(, ('127.0.0.1', 53478))
EchoRequestHandler: __init__
EchoRequestHandler: setup
EchoRequestHandler: handle
client: sending data: "Hello, world"
EchoRequestHandler: recv()->"Hello, world"
EchoRequestHandler: finish
EchoServer: close_request()
EchoServer: handle_request
client: waiting for response
client: response from server: "Hello, world"
client: closing socket
client: done
```

程序使用的端口号会在你每次运行时改变，因为内核自动分配给他可用的端口。如果你想让服务器每次运行时都监听固定的端口，可以为地址元组提供一个数字而不是0.

上述例子的简单版本，没有日志记录，如下：

```
import SocketServer

class EchoRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        # Echo the back to the client
```

```
        data = self.request.recv(1024)
        self.request.send(data)
        return

if __name__ == '__main__':
    import socket
    import threading
    address = ('localhost', 0) # let the kernel give us a port
    server = SocketServer.TCPServer(address, EchoRequestHandler)
    ip, port = server.server_address # find out what port we were given

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True) # don't hang on exit
    t.start()

    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Send the data
    message = 'Hello, world'
    print 'Sending : "%s"' % message
    len_sent = s.send(message)

    # Receive a response
    response = s.recv(len_sent)
    print 'Received: "%s"' % response

    # Clean up
    s.close()
    server.socket.close()
```

Note: 这种情况下，不需要特殊的服务器类，因为TCPServer已经符合我们的需要了.

```
$ python SocketServer_echo_simple.py
Sending : "Hello, world"
Received: "Hello, world"
```

39.7 线程和进程

为服务器增加线程或forking支持，只需要简单的在类层次结构中增加包含合适的混合类. 这个混合类重载 process_request()，当要处理一个请求时开始一个新的线程或进程，并且会在一个新的孩子线程或进程中完成工作.

对于线程，使用ThreadingMixIn:

```
import threading
import SocketServer

class ThreadedEchoRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        # Echo the back to the client
        data = self.request.recv(1024)
        cur_thread = threading.currentThread()
        response = '%s: %s' % (cur_thread.getName(), data)
        self.request.send(response)
        return
```

```
class ThreadedEchoServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    pass

if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0) # let the kernel give us a port
    server = ThreadedEchoServer(address, ThreadedEchoRequestHandler)
    ip, port = server.server_address # find out what port we were given

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True) # don't hang on exit
    t.start()
    print 'Server loop running in thread:', t.getName()

    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Send the data
    message = 'Hello, world'
    print 'Sending : "%s"' % message
    len_sent = s.send(message)

    # Receive a response
    response = s.recv(1024)
    print 'Received: "%s"' % response

    # Clean up
    s.close()
    server.socket.close()
```

从服务器返回的response包括了处理请求的线程id:

```
$ python SocketServer_threaded.py
Server loop running in thread: Thread-1
Sending : "Hello, world"
Received: "Thread-2: Hello, world"
```

使用独立的进程，可以使用ForkingMixIn:

```
import os
import SocketServer

class ForkingEchoRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        # Echo the back to the client
        data = self.request.recv(1024)
        cur_pid = os.getpid()
        response = '%s: %s' % (cur_pid, data)
        self.request.send(response)
        return

class ForkingEchoServer(SocketServer.ForkingMixIn, SocketServer.TCPServer):
    pass

if __name__ == '__main__':
    import socket
    import threading
```

```
    address = ('localhost', 0) # let the kernel give us a port
    server = ForkingEchoServer(address, ForkingEchoRequestHandler)
    ip, port = server.server_address # find out what port we were given

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True) # don't hang on exit
    t.start()

    print 'Server loop running in process:', os.getpid()
    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Send the data
    message = 'Hello, world'
    print 'Sending : "%s"' % message
    len_sent = s.send(message)

    # Receive a response
    response = s.recv(1024)
    print 'Received: "%s"' % response

    # Clean up
    s.close()

    server.socket.close()
```

在这种情况下，从服务器返回的response包含了一个子进程的id:

```
$ python SocketServer_forking.py
Server loop running in process: 20173
Sending : "Hello, world"
Received: "20175: Hello, world"
```

39.8 参考

- effbot.org: Sockets

CHAPTER

FORTY

PYMOTW: ASYNCORE

- 模块： asyncore
- 目的： 异步I/O操作
- python版本： 1.5.2+

asyncore模块包含了使用I/O对象如sockets以方便我们异步的进行操作(可以代替例如线程)。 他提供的主要类是dispatcher，是一个封装了一个socket并提供了处理如连接，读取，写入等的事件钩子，这些会在主循环函数loop()中被调用.

40.1 客户端

创建一个异步客户端，继承dispatcher并提供了创建，读取和写入socket的实现。 让我们拿HTTP客户端来做示例，它基于一个来自标准库文档的例子.

```
import asyncore
import logging
import socket
from cStringIO import StringIO
import urlparse

class HttpClient(asyncore.dispatcher):

    def __init__(self, url):
        self.url = url
        self.logger = logging.getLogger(self.url)
        self.parsed_url = urlparse.urlparse(url)
        asyncore.dispatcher.__init__(self)
        self.write_buffer = 'GET %s HTTP/1.0\r\n\r\n' % self.url
        self.read_buffer = StringIO()
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        address = (self.parsed_url.netloc, 80)
        self.logger.debug('connecting to %s', address)
        self.connect(address)

    def handle_connect(self):
        self.logger.debug('handle_connect()')

    def handle_close(self):
        self.logger.debug('handle_close()')
        self.close()

    def writable(self):
        is_writable = (len(self.write_buffer) > 0)
        if is_writable:
            self.logger.debug('writable() -> %s', is_writable)
```

```
        return is_writable

    def readable(self):
        self.logger.debug('readable() -> True')
        return True

    def handle_write(self):
        sent = self.send(self.write_buffer)
        self.logger.debug('handle_write() -> "%s"', self.write_buffer[:sent])
        self.write_buffer = self.write_buffer[sent:]

    def handle_read(self):
        data = self.recv(8192)
        self.logger.debug('handle_read() -> %d bytes', len(data))
        self.read_buffer.write(data)


if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG,
      format='%(name)s: %(message)s',
    )

    clients = [
      HttpClient('http://www.python.org/'),
      HttpClient('http://www.doughellmann.com/PyMOTW/contents.html'),
    ]

    logging.debug('LOOP STARTING')
    asyncore.loop()
    logging.debug('LOOP DONE')
    for c in clients:
        response_body = c.read_buffer.getvalue()
        print c.url, 'got', len(response_body), 'bytes'
```

首先，通过在 `__init__()` 中使用基类的 `create_socket()` 方法创建socket． 当然还有其他的方法，但在这个例子中． 我们创建的是一个TCP/IP socket． 因此使用基类已经可以满足需要了．

`handle_connect()`钩子简单的显示了被调用的信息． 其他客户端还可以在 `handle_connect()` 中进行一些握手或者协议交涉等类似的处理．

`handle_close()`同样的显示了被调用的信息． 基类可以正确的关闭socket，如果你不想作额外的工作可以交给函数来处理．

asyncore循环中， 使用 `writeable()` 和同属方法 `readable()` 决定什么样的操作来处理每个dispatcher． 实际上，sockets的 `poll()` 和 `select()` 方法或由每个dispatcher操作的文件描述符会在asyncore代码内部处理． 因此你不必自己来实现这些． 只需简单的指出某dispatcher是否是对读和写都要处理． 在这个HTTP client的例子中， 只要存在请求发送到服务器的数据，`writable()` 会返回True，而 `readable()` 一直返回True，因为我们想读取所有到来的数据．

在每次循环中，当 `writable()` 正确响应，`handle_write()` 会被调用． 在这个例子中，在 `__init__()` 中创建的HTTP请求字符串会被发送到服务器，写缓冲区会清除成功发送的数据．

类似地， 当 `readable()` 正确响应， 也就是有数据要读取了． 那么，`handle_read()` 会被调用．

例子中`__main__`之后的代码首先配置一个日志记录以便调试， 然后创建了两个客户端分别下载网页．创建客户端需要在一个由asyncore内部保存的 *map* 中注册． 随着主循环的开始，客户端的下载也开始，当一个客户端从可读socket中读取0个字节， 这会被解释成关闭连接然后．

下面是这个例子中客户端app运行出的可能结果：

```
$ python asyncore_http_client.py
http://www.python.org/: connecting to ('www.python.org', 80)
http://www.doughellmann.com/PyMOTW/contents.html: connecting to ('www.doughellmann.com', 80)
root: LOOP STARTING
```

```
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_connect()
http://www.doughellmann.com/PyMOTW/contents.html: handle_write() -> "GET http://www.doughellmann.com/PyMOTW/con

"
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 3163 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 1448 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 2896 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 2896 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 2896 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 2896 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 1448 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 1448 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 1448 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 1448 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 895 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_close()
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 0 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.python.org/: handle_connect()
http://www.python.org/: handle_write() -> "GET http://www.python.org/ HTTP/1.0
```

```
"
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1257 bytes
http://www.python.org/: readable() -> True
http://www.python.org/: handle_close()
http://www.python.org/: handle_read() -> 0 bytes
root: LOOP DONE
http://www.python.org/ got 18009 bytes
http://www.doughellmann.com/PyMOTW/contents.html got 22882 bytes
```

40.2 服务器

下面的例子, 通过重新实现SocketServe中的EchoServer例子来说明如何在服务器上使用异步. 这里主要有3个类: EchoServer用于接收来自客户端的连接并创建各自的EchoHandler实例, EchoClient是类似于HTTPClient的异步dispatche.

```
import asyncore
import logging

class EchoServer(asyncore.dispatcher):
    """Receives connections and establishes handlers for each client.
    """

    def __init__(self, address):
        self.logger = logging.getLogger('EchoServer')
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.bind(address)
        self.address = self.socket.getsockname()
        self.logger.debug('binding to %s', self.address)
        self.listen(1)
        return

    def handle_accept(self):
        # Called when a client connects to our socket
```

```
        client_info = self.accept()
        self.logger.debug('handle_accept() -> %s', client_info[1])
        EchoHandler(sock=client_info[0])
        # We only want to deal with one client at a time,
        # so close as soon as we set up the handler.
        # Normally you would not do this and the server
        # would run forever or until it received instructions
        # to stop.
        self.handle_close()
        return

    def handle_close(self):
        self.logger.debug('handle_close()')
        self.close()
        return

class EchoHandler(asyncore.dispatcher):
    """Handles echoing messages from a single client.
    """

    def __init__(self, sock, chunk_size=256):
        self.chunk_size = chunk_size
        self.logger = logging.getLogger('EchoHandler%s' % str(sock.getsockname()))
        asyncore.dispatcher.__init__(self, sock=sock)
        self.data_to_write = []
        return

    def writable(self):
        """We want to write if we have received data."""
        response = bool(self.data_to_write)
        self.logger.debug('writable() -> %s', response)
        return response


    def handle_write(self):
        """Write as much as possible of the most recent message we have received."""
        data = self.data_to_write.pop()
        sent = self.send(data[:self.chunk_size])
        if sent < len(data):
            remaining = data[sent:]
            self.data_to_write.append(remaining)
        self.logger.debug('handle_write() -> (%d) "%s"', sent, data[:sent])
        if not self.writable():
            self.handle_close()

    def handle_read(self):
        """Read an incoming message from the client and put it into our outgoing queue."""
        data = self.recv(self.chunk_size)
        self.logger.debug('handle_read() -> (%d) "%s"', len(data), data)
        self.data_to_write.insert(0, data)

    def handle_close(self):
        self.logger.debug('handle_close()')
        self.close()



class EchoClient(asyncore.dispatcher):
    """Sends messages to the server and receives responses.
    """

    def __init__(self, host, port, message, chunk_size=512):
```

```
        self.message = message
        self.to_send = message
        self.received_data = []
        self.chunk_size = chunk_size
        self.logger = logging.getLogger('EchoClient')
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.logger.debug('connecting to %s', (host, port))
        self.connect((host, port))
        return

    def handle_connect(self):
        self.logger.debug('handle_connect()')

    def handle_close(self):
        self.logger.debug('handle_close()')
        self.close()
        received_message = ''.join(self.received_data)
        if received_message == self.message:
            self.logger.debug('RECEIVED COPY OF MESSAGE')
        else:
            self.logger.debug('ERROR IN TRANSMISSION')
            self.logger.debug('EXPECTED "%s"', self.message)
            self.logger.debug('RECEIVED "%s"', received_message)
        return

    def writable(self):
        self.logger.debug('writable() -> %s', bool(self.to_send))
        return bool(self.to_send)

    def handle_write(self):
        sent = self.send(self.to_send[:self.chunk_size])
        self.logger.debug('handle_write() -> (%d) "%s"', sent, self.to_send[:sent])
        self.to_send = self.to_send[sent:]

    def handle_read(self):
        data = self.recv(self.chunk_size)
        self.logger.debug('handle_read() -> (%d) "%s"', len(data), data)
        self.received_data.append(data)

if __name__ == '__main__':
    import socket
    import threading

    logging.basicConfig(level=logging.DEBUG,
      format='%(name)s: %(message)s',
    )

    address = ('localhost', 0) # let the kernel give us a port
    server = EchoServer(address)
    ip, port = server.address # find out what port we were given
    client = EchoClient(ip, port, message=open('lorem.txt', 'r').read())
    asyncore.loop()
```

EchoServer和EchoHandler被定义在独立的类中因为他们各自处理不同的事情． 当EchoServer接收一个连接，一个新的socket被建立． 一个利用socket map(这是由asyncore内部维护的)的EchoHandler被创建，而不是在EchoServer内部进行各个客户端请求的分配．

```
$ python asyncore_echo_server.py
EchoServer: binding to ('127.0.0.1', 52235)
EchoClient: connecting to ('127.0.0.1', 52235)
EchoClient: writable() -> True
```

```
EchoServer: handle_accept() -> ('127.0.0.1', 52236)
EchoServer: handle_close()
EchoClient: handle_connect()
EchoClient: handle_write() -> (512) "Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec
egestas, enim et consectetuer ullamcorper, lectus ligula rutrum leo, a
elementum elit tortor eu quam. Duis tincidunt nisi ut ante. Nulla
facilisi. Sed tristique eros eu libero. Pellentesque vel arcu. Vivamus
purus orci, iaculis ac, suscipit sit amet, pulvinar eu,
lacus. Praesent placerat tortor sed nisl. Nunc blandit diam egestas
dui. Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Aliquam viverra f"
EchoClient: writable() -> True
EchoHandler('127.0.0.1', 52235): writable() -> False
EchoHandler('127.0.0.1', 52235): handle_read() -> (256) "Lorem ipsum dolor sit amet, consectetuer adipiscing el
egestas, enim et consectetuer ullamcorper, lectus ligula rutrum leo, a
elementum elit tortor eu quam. Duis tincidunt nisi ut ante. Nulla
facilisi. Sed tristique eros eu libero. Pellentesque ve"
EchoClient: handle_write() -> (225) "ringilla
leo. Nulla feugiat augue eleifend nulla. Vivamus mauris. Vivamus sed
mauris in nibh placerat egestas. Suspendisse potenti. Mauris massa. Ut
eget velit auctor tortor blandit sollicitudin. Suspendisse imperdiet
justo.
"
EchoClient: writable() -> False
EchoHandler('127.0.0.1', 52235): writable() -> True
EchoHandler('127.0.0.1', 52235): handle_read() -> (256) "l arcu. Vivamus
purus orci, iaculis ac, suscipit sit amet, pulvinar eu,
lacus. Praesent placerat tortor sed nisl. Nunc blandit diam egestas
dui. Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Aliquam viverra f"
EchoHandler('127.0.0.1', 52235): handle_write() -> (256) "Lorem ipsum dolor sit amet, consectetuer adipiscing el
egestas, enim et consectetuer ullamcorper, lectus ligula rutrum leo, a
elementum elit tortor eu quam. Duis tincidunt nisi ut ante. Nulla
facilisi. Sed tristique eros eu libero. Pellentesque ve"
EchoHandler('127.0.0.1', 52235): writable() -> True
EchoClient: writable() -> False
EchoHandler('127.0.0.1', 52235): writable() -> True
EchoClient: handle_read() -> (256) "Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec
egestas, enim et consectetuer ullamcorper, lectus ligula rutrum leo, a
elementum elit tortor eu quam. Duis tincidunt nisi ut ante. Nulla
facilisi. Sed tristique eros eu libero. Pellentesque ve"
EchoHandler('127.0.0.1', 52235): handle_read() -> (225) "ringilla
leo. Nulla feugiat augue eleifend nulla. Vivamus mauris. Vivamus sed
mauris in nibh placerat egestas. Suspendisse potenti. Mauris massa. Ut
eget velit auctor tortor blandit sollicitudin. Suspendisse imperdiet
justo.
"
EchoHandler('127.0.0.1', 52235): handle_write() -> (256) "l arcu. Vivamus
purus orci, iaculis ac, suscipit sit amet, pulvinar eu,
lacus. Praesent placerat tortor sed nisl. Nunc blandit diam egestas
dui. Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Aliquam viverra f"
EchoHandler('127.0.0.1', 52235): writable() -> True
EchoClient: writable() -> False
EchoHandler('127.0.0.1', 52235): writable() -> True
EchoClient: handle_read() -> (256) "l arcu. Vivamus
purus orci, iaculis ac, suscipit sit amet, pulvinar eu,
lacus. Praesent placerat tortor sed nisl. Nunc blandit diam egestas
dui. Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Aliquam viverra f"
EchoHandler('127.0.0.1', 52235): handle_write() -> (225) "ringilla
leo. Nulla feugiat augue eleifend nulla. Vivamus mauris. Vivamus sed
```

```
mauris in nibh placerat egestas. Suspendisse potenti. Mauris massa. Ut
eget velit auctor tortor blandit sollicitudin. Suspendisse imperdiet
justo.
"
EchoHandler('127.0.0.1', 52235): writable() -> False
EchoHandler('127.0.0.1', 52235): handle_close()
EchoClient: writable() -> False
EchoClient: handle_read() -> (225) "ringilla
leo. Nulla feugiat augue eleifend nulla. Vivamus mauris. Vivamus sed
mauris in nibh placerat egestas. Suspendisse potenti. Mauris massa. Ut
eget velit auctor tortor blandit sollicitudin. Suspendisse imperdiet
justo.
"
EchoClient: writable() -> False
EchoClient: handle_close()
EchoClient: RECEIVED COPY OF MESSAGE
EchoClient: handle_read() -> (0) ""
```

在这个例子中，服务器，处理者，客户端对象都被asyncore在一个单独进程中使用一相同的socket map维护． 将服务器和客户端分开，简单的将相关代码分开，并且在各自程序中运行 asyncore.loop()． 当一个dispatcher被关闭了，会从map中删掉，整个循环会在map为空时停下来．

40.3 其他循环事件的处理

有时候需必须在已有的应用事件中加入异步事件． 例如，一个GUI应用不想在所有异步操作处理时阻断UI-这是违背了异步的目的． 为了使这种集成更方便，asycore.loop() 接收一个用于设置超时的参数和一个用于限制循环次数的参数，重新使用第一个例子中的HttpClient，我们可以看到他们各自的效果．

```
import asyncore
import logging

from asyncore_http_client import HttpClient

logging.basicConfig(level=logging.DEBUG,
  format='%(name)s: %(message)s',
)

clients = [
  HttpClient('http://www.doughellmann.com/PyMOTW/contents.html'),
  HttpClient('http://www.python.org/'),
]

loop_counter = 0
while asyncore.socket_map:
    loop_counter += 1
    logging.debug('loop_counter=%s', loop_counter)
    asyncore.loop(timeout=1, count=1)
```

我们可以看到，在 asyncore.loop() 中调用时，客户端仅读或写数据一次，替代我们自己的while循环． 我们可以在GUI工具包活其他需要这种功能机制的地方类似的调用 asyncore.loop() (ui不会忙于处理其他事件)．

```
$ python asyncore_loop.py

http://www.doughellmann.com/PyMOTW/contents.html: connecting to ('www.doughellmann.com', 80)
http://www.python.org/: connecting to ('www.python.org', 80)
root: loop_counter=1
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
```

```
http://www.doughellmann.com/PyMOTW/contents.html: writable() -> True
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_connect()
http://www.doughellmann.com/PyMOTW/contents.html: handle_write() -> "GET http://www.doughellmann.com/PyMOTW/con

"
root: loop_counter=2
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 267 bytes
root: loop_counter=3
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 8192 bytes
root: loop_counter=4
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 6288 bytes
root: loop_counter=5
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 1448 bytes
root: loop_counter=6
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 1448 bytes
root: loop_counter=7
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 2896 bytes
root: loop_counter=8
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 2343 bytes
root: loop_counter=9
http://www.doughellmann.com/PyMOTW/contents.html: readable() -> True
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.doughellmann.com/PyMOTW/contents.html: handle_close()
http://www.doughellmann.com/PyMOTW/contents.html: handle_read() -> 0 bytes
root: loop_counter=10
http://www.python.org/: readable() -> True
http://www.python.org/: writable() -> True
http://www.python.org/: handle_connect()
http://www.python.org/: handle_write() -> "GET http://www.python.org/ HTTP/1.0

"
root: loop_counter=11
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
root: loop_counter=12
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
root: loop_counter=13
```

```
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
root: loop_counter=14
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
root: loop_counter=15
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
root: loop_counter=16
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
root: loop_counter=17
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
root: loop_counter=18
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
root: loop_counter=19
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
root: loop_counter=20
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
root: loop_counter=21
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
root: loop_counter=22
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1396 bytes
root: loop_counter=23
http://www.python.org/: readable() -> True
http://www.python.org/: handle_read() -> 1257 bytes
root: loop_counter=24
http://www.python.org/: readable() -> True
http://www.python.org/: handle_close()
http://www.python.org/: handle_read() -> 0 bytes
```

40.4 文件处理

一般情况下，你可能只想在sockets中使用asyncore，但有时异步的读取文件也是有用的(当测试网络服务器而不需要网络设置使用文件，或者是部分读取，写入大数据文件）在这些情况下，asyncore提供了file_dispatcher和file_wrapper类.

```
import asyncore
import os

class FileReader(asyncore.file_dispatcher):

    def writable(self):
        return False

    def handle_read(self):
        data = self.recv(256)
        print 'READ: (%d) "%s"' % (len(data), data)

    def handle_expt(self):
        # Ignore events that look like out of band data
        pass
```

```
    def handle_close(self):
        self.close()

lorem_fd = os.open('lorem.txt', os.O_RDONLY)
reader = FileReader(lorem_fd)
asyncore.loop()
```

这个例子在Python2.5.2下进行测试，我使用了 `os.open()` 获得文件描述符． 对于Python2.6之后，file_dispatcher自动将所有具有 `fileno()` 方法的东西转换成一个文件描述符．

```
$ python asyncore_file_dispatcher.py
READ: (256) "Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec
egestas, enim et consectetuer ullamcorper, lectus ligula rutrum leo, a
elementum elit tortor eu quam. Duis tincidunt nisi ut ante. Nulla
facilisi. Sed tristique eros eu libero. Pellentesque ve"
READ: (256) "l arcu. Vivamus
purus orci, iaculis ac, suscipit sit amet, pulvinar eu,
lacus. Praesent placerat tortor sed nisl. Nunc blandit diam egestas
dui. Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Aliquam viverra f"
READ: (225) "ringilla
leo. Nulla feugiat augue eleifend nulla. Vivamus mauris. Vivamus sed
mauris in nibh placerat egestas. Suspendisse potenti. Mauris massa. Ut
eget velit auctor tortor blandit sollicitudin. Suspendisse imperdiet
justo.
"
READ: (0) ""
```

40.5 参考

- asyncore: 该模块的标准库文档

CHAPTER

FORTYONE

PYMOTW: TEMPFILE

使用tempfile模块可以安全的生成临时文件和目录.

- 模块: tempfile
- 目的: 创建临时的文件系统资源.
- Python 版本: 1.4 + (主安全修订版本2.3)

41.1 描述

许多程序需要将产生的中间数据保存在文件中. 安全地创建一些具有唯一名字的文件, 使得想要中断应用的人无法猜测到这些文件的名字, 这是具有挑战性的. tempfile模块提供一些用于安全处理文件系统资源的函数. TemporaryFile()打开并返回一个未命名文件. NamedTemporaryFile() 打开并返回一个已命名文件, mktemp()可以创建一个临时目录并返回它的名字.

41.2 临时文件

如果你的应用需要用一临时文件来存储数据, 但不想和其他程序共享数据的话, 创建文件的最好方式是使用TemporaryFile()函数. 它在任意一个可能的平台上创建文件, 并且能够随时断开链接. 这让其他程序不可能找到或访问这个文件,因为在文件系统表中没有它的引用.这个由TemporaryFile()创建的文件在关闭的时候自动删除.

```
import os
import tempfile

print 'Building a file name yourself:'
filename = '/tmp/guess_my_name.%s.txt' % os.getpid()
temp = open(filename, 'w+b')
try:
    print 'temp:', temp
    print 'temp.name:', temp.name
finally:
    temp.close()
    # Clean up the temporary file yourself
    os.remove(filename)

print
print 'TemporaryFile:'
temp = tempfile.TemporaryFile()
try:
    print 'temp:', temp
    print 'temp.name:', temp.name
finally:
```

```
    # Automatically cleans up the file
    temp.close()
```

这个例子说明了使用一个普通模式作为名字的文件和使用TemporaryFile()创建的文件的区别． 注意： TemporaryFile()返回的文件不含有名字．

```
$ python tempfile_TemporaryFile.py
Building a file name yourself:
temp: <open file '/tmp/guess_my_name.7297.txt', mode 'w+b' at 0x5c338>
temp.name: /tmp/guess_my_name.7297.txt

TemporaryFile:
temp: <open file '<fdopen>', mode 'w+b' at 0x5c410>
temp.name: <fdopen>
```

默认情况下， 文件以模式'w+b'方式被创建， 所以在所有平台上都是一致的， 你的程序都能写入和读取它．

```
import os
import tempfile

temp = tempfile.TemporaryFile()
try:
    temp.write('Some data')
    temp.seek(0)

    print temp.read()
finally:
    temp.close()
```

在写入之后， 你需使用seek()将当前文件指针指向之前的位置以便能够读取刚才的写入的数据．

```
$ python tempfile_TemporaryFile_binary.py
Some data
```

如果你想要的是文本模式的临时文件， 可以传递mode='w+t':

```
import tempfile

f = tempfile.TemporaryFile(mode='w+t')
try:
    f.writelines(['first\n', 'second\n'])
    f.seek(0)

    for line in f:
        print line.rstrip()
finally:
    f.close()
```

这个文件把数据看成是文本字符串：

```
$ python tempfile_TemporaryFile_text.py
first
second
```

41.3 命名临时文件

在一些情况下，也需要命名临时文件． 如果你的程序跨越多个进程，或者甚至是主机，命名文件是一种最简单的在程序各部分间传递数据的方式． NamedTemporaryFile()函数创建了一个有名字的，即可以按名访问的文件．

```
import os
import tempfile

temp = tempfile.NamedTemporaryFile()
try:
    print 'temp:', temp
    print 'temp.name:', temp.name
finally:
    # Automatically cleans up the file
    temp.close()
    print 'Exists after close:', os.path.exists(temp.name)
```

即使文件被命名了，他仍然可以在处理结束后删除．

```
$ python tempfile_NamedTemporaryFile.py
temp: <open file '<fdopen>', mode 'w+b' at 0x5c338>
temp.name: /var/folders/9R/9R1t+tR02Raxzk+F71Q50U+++Uw/-Tmp-/tmplBKZMv
Exists after close: False
```

41.4 mkdtemp

如果你需要多个临时文件，可以创建一个临时目录然后把所有文件放在这个目录中． 使用mkdtemp()函数来创建一个临时目录．

```
import os
import tempfile

directory_name = tempfile.mkdtemp()
print directory_name
# Clean up the directory yourself
os.removedirs(directory_name)
```

由于目录不是''opened''，你需要手动将它删除．

```
$ python tempfile_mkdtemp.py
/var/folders/9R/9R1t+tR02Raxzk+F71Q50U+++Uw/-Tmp-/tmpOOsHPg
```

41.5 预测文件名

为了便于调试，需要将原始临时文件的一些信息保留． 当然这样明显地比匿名临时文件不安全很多，包括名字中的一部分信息能够让你检测这个文件是否正在被你的程序使用． 到目前为止描述的所有函数需要3个参数来在一定程度上控制你的文件名． 命名规则如下：

```
dir + prefix + random + suffix
```

除了random之外的所有参数值都被传递给TemporaryFile()， NamedTemporaryFile()， 和mkdtemp()． 例如：

```
import tempfile

temp = tempfile.NamedTemporaryFile(suffix='_suffix',
                                   prefix='prefix_',
                                   dir='/tmp',
                                   )
try:
    print 'temp:', temp
    print 'temp.name:', temp.name
finally:
    temp.close()
```

prefix 和suffix参数再联合一个随机字符串生成一个文件名，dir参数指定新文件所在的位置.

```
$ python tempfile_NamedTemporaryFile_args.py
temp: <open file '<fdopen>', mode 'w+b' at 0x5c338>
temp.name: /tmp/prefix_zy-7H3_suffix
```

41.6 临时文件的位置

如果你没有使用dir参数来指定一个目标目录，那么，临时文件所在的真实路径会根据你平台和设置来确定． tempfile模块包含2个用于查询运行时间相关设置的函数：

```
import tempfile

print 'gettempdir():', tempfile.gettempdir()
print 'gettempprefix():', tempfile.gettempprefix()
```

gettempdir()返回放所有临时文件的默认目录． gettempprefix()返回新文件和目录名字的字符串前缀.

```
$ python tempfile_settings.py
gettempdir(): /var/folders/9R/9R1t+tR02Raxzk+F71Q50U+++Uw/-Tmp-
gettempprefix(): tmp
```

gettempdir() 返回值的确定是基于一个直接查找算法，它从一个位置列表中找到第一个可以创建文件的目录． 库文档中说明：

Python查找一个标准目录列表，将第一个用户有权限在其中创建文件的目录来设置tempdir ． 这个列表是：

1. 环境变量TMPDIR中的目录名.
2. 环境变量TEMP中的目录.
3. 环境变量TMP中的目录.
4. 平台指定的位置：
    - 在RiscOS上，由Wimp$ScrapDir指定目录名字.
    - 在Windows上，以C:$backslash$TEMP，C:$backslash$TMP，$backslash$TEMP，和 $backslash$TMP按序查找目录.
    - 在其他平台上，以/tmp，/var/tmp，和/usr/tmp按序查找目录.
5. 最后一个是当前工作目录.

如果你的程序需要一个在全局位置下存放所有的临时文件，你需要明确设置这个位置但又不想通过环境变量来设置，这样的话，你可以直接设置tempfile.tempdir来指定.

```
import tempfile

tempfile.tempdir = '/I/changed/this/path'
print 'gettempdir():', tempfile.gettempdir()
```

```
$ python tempfile_tempdir.py
gettempdir(): /I/changed/this/path
```

41.7 参考

- Python Module of the Week Home
- Download Sample Code

CHAPTER

FORTYTWO

PYMOTW: SCHED

sched模块实现了一般事件调度功能，能在指定时间执行某个任务.

- 模块: sched
- 目的: 一般事件调度.
- Python 版本: 1.4 +

42.1 描述

scheduler类使用一般的事件调度接口. 它使用time函数来获得当前时间，delay函数用于等待一段特定时间. 这里，真正使用什么样的时间单位不是很重要，因为这能让接口更具灵活性，可用于多种用途.

time函数调用时不需要给定任何参数，应返回当前时间的字符串表示. 而delay函数需要一个整型参数，和time函数使用相同的时间刻度，该函数在返回前需要等待特定个时间单元. 例如，time.time()和time.sleep()这两个函数符合这些要求.

为了支持多线程应用，在生成每个线程之后，调用参数为0的delay函数，这样来保证其他线程有机会运行.

42.2 延迟后运行事件

事件可以在延迟一段时间后，或在指定时间点上调度执行. enter()方法使这些事件在延迟一段时间后被调度，它需要4个参数:

- A number representing the delay 代表延迟多长时间的数字
- A priority value 优先级值
- The function to call 需要被调用的函数
- A tuple of arguments for the function 函数的参数元组

下面这个例子中，分别在2和3秒之后调度2个不同的事件. 当到达某事件的调度时刻，print_event()被调用，显示出目前时间和传递给事件的参数名字.

```
import sched
import time

scheduler = sched.scheduler(time.time, time.sleep)

def print_event(name):
    print 'EVENT:', time.time(), name
```

```
print 'START:', time.time()
scheduler.enter(2, 1, print_event, ('first',))
scheduler.enter(3, 1, print_event, ('second',))

scheduler.run()
```

输出如下：

```
$ python sched_basic.py
START: 1190727943.36
EVENT: 1190727945.36 first
EVENT: 1190727946.36 second
```

第一个事件的时间信息是调度开始2秒后，第二个事件的时间信息是调度开始3秒后.

42.3 事件重叠

run()一直被阻塞，直到所有事件被全部执行完. 每个事件在同一线程中运行，所以如果一个事件的执行时间大于其他事件的延迟时间，那么，就会产生重叠. 重叠的解决方法是推迟后来事件的执行时间. 这样保证没有丢失任何事件，但这些事件的调用时刻会比原先设定的迟. 在下面的例子中，long_event()中通过睡眠2秒钟来延迟调度，同样延迟调度很容易通过运行长时间计算或阻塞I/O来实现.

```
import sched
import time

scheduler = sched.scheduler(time.time, time.sleep)

def long_event(name):
    print 'BEGIN EVENT :', time.time(), name
    time.sleep(2)
    print 'FINISH EVENT:', time.time(), name

print 'START:', time.time()
scheduler.enter(2, 1, long_event, ('first',))
scheduler.enter(3, 1, long_event, ('second',))

scheduler.run()
```

第二个事件在第一个事件运行结束后立即运行，因为第一个事件的执行时间足够长，已经超过第二个事件的预期开始时刻.

```
$ python sched_overlap.py
START: 1190728573.16
BEGIN EVENT : 1190728575.16 first
FINISH EVENT: 1190728577.16 first
BEGIN EVENT : 1190728577.16 second
FINISH EVENT: 1190728579.16 second
```

42.4 事件优先级

如果在相同的时刻点上有多个事件需要被执行，那么它们的优先级参数决定他们的执行顺序.

```
now = time.time()
print 'START:', now
```

```
scheduler.enterabs(now+2, 2, print_event, ('first',))
scheduler.enterabs(now+2, 1, print_event, ('second',))
scheduler.run()
```

为了保证事件准确的在同一时刻执行，使用了enterabs()方法而不是enter()方法． enterabs()的第一个参数是运行事件的确切时间，而不是延迟时间量．

```
$ python sched_priority.py
START: 1190728789.4
EVENT: 1190728791.4 second
EVENT: 1190728791.4 first
```

42.5 取消事件

enter()和enterabs()返回一事件的引用，该引用可被用于事件的取消． 由于run()阻塞，所以事件的取消操作需要在另外一个线程中进行． 如下例子，在一个子线程开始执行调度，而主处理线程用于取消某个事件．

```
import sched
import threading
import time

scheduler = sched.scheduler(time.time, time.sleep)

# Set up a global to be modified by the threads
counter = 0

def increment_counter(name):

    global counter
    print 'EVENT:', time.time(), name
    counter += 1
    print 'NOW:', counter


print 'START:', time.time()
e1 = scheduler.enter(2, 1, increment_counter, ('E1',))
e2 = scheduler.enter(3, 1, increment_counter, ('E2',))

# Start a thread to run the events
t = threading.Thread(target=scheduler.run)
t.start()

# Back in the main thread, cancel the first scheduled event.
scheduler.cancel(e1)

# Wait for the scheduler to finish running in the thread
t.join()

print 'FINAL:', counter
```

两个事件被安排调度，但之后取消了第一个事件． 只有第二个事件执行了，所以我们看到计数器仅累加了一次．

```
$ python sched_cancel.py
START: 1190729094.13
EVENT: 1190729097.13 E2
```

```
NOW: 1
FINAL: 1
```

42.6 参考

- Python Module of the Week Home
- Download Sample Code

CHAPTER

FORTYTHREE

PYMOTW: CSV

- 模块： csv
- 目的： 对以分号分隔的数值文件进行读写
- Python 版本： 2.3+

43.1 描述

csv 模块在处理那些从电子数据表格或数据库中导入到文本文件的数据时，是很有用的． 这里并没有很好的定义标准，因此csv模块使用了''dialects''，通过使用不同的参数来解析csv文件． 对于一般的读和写，这个模块也能处理Microsoft Excel格式数据．

43.2 局限性

Python 2.5 版本的csv不支持unicode数据，而对于ASCII的NUL字符处理也有点问题，所以推荐使用UTF-8或可打印ASCII字符．

43.3 读取

从csv文件中读取数据，可以使用reader()函数来创建一个读取对象． 这个读取对象顺序处理文件的每一行，可以把它当成迭代器使用，例如：

```
import csv
import sys

f = open(sys.argv[1], 'rt')
try:

    reader = csv.reader(f)
    for row in reader:
        print row

finally:

    f.close()
```

reader()的第一个参数指示源文本行，在这个例子中，是一个文件，但它可以是任何可转换的对象(StringIO对象，lists等)． 指定其他可选的参数可用于控制输入的数据如何被解析．

例子文件''testdata.csv''是从NeoOffice中导入的，其内容如下．

```
$ cat testdata.csv
"Title 1","Title 2","Title 3"
1,"a",08/18/07
2,"b",08/19/07
3,"c",08/20/07
4,"d",08/21/07
5,"e",08/22/07
6,"f",08/23/07
7,"g",08/24/07
8,"h",08/25/07
9,"i",08/26/07
```

它被读取时，输入数据的每一行被转换为一个字符串列表.

```
$ python csv_reader.py testdata.csv
['Title 1', 'Title 2', 'Title 3']
['1', 'a', '08/18/07']
['2', 'b', '08/19/07']
['3', 'c', '08/20/07']
['4', 'd', '08/21/07']
['5', 'e', '08/22/07']
['6', 'f', '08/23/07']
['7', 'g', '08/24/07']
['8', 'h', '08/25/07']
['9', 'i', '08/26/07']
```

如果你知道特定的列具有特定的类型，你就可以自行转换，但csv不会自动转换. 它会自动处理嵌入在一行字符串中(这个行和输入源文件的''行''意思是不同的)的换行符.

```
$ cat testlinebreak.csv
"Title 1","Title 2","Title 3"
1,"first line ##     line
second line",08/18/07

$ python csv_reader.py testlinebreak.csv
['Title 1', 'Title 2', 'Title 3']
['1', 'first line\nsecond line', '08/18/07'] ##  csv  row
```

43.4 写入

当你想把数据导入到其他应用程序中，对CSV文件的写入也是非常方便的. 使用writer()函数来创建一个写入对象，对于每一行，使用writerow()来输出一行.

```
import csv
import sys

f = open(sys.argv[1], 'wt')
try:

    writer = csv.writer(f)
    writer.writerow( ('Title 1', 'Title 2', 'Title 3') )
    for i in range(10):
        writer.writerow( (i+1, chr(ord('a') + i), '08/%02d/07' % (i+1)) )

finally:

    f.close()
```

这个例子的输出和上述读取例子的导出数据看起来不怎么一样．

```
$ python csv_writer.py testout.csv
$ cat testout.csv
Title 1,Title 2,Title 3
1,a,08/01/07
2,b,08/02/07
3,c,08/03/07
4,d,08/04/07
5,e,08/05/07
6,f,08/06/07
7,g,08/07/07
8,h,08/08/07
9,i,08/09/07
10,j,08/10/07
```

写入对象没有使用默认的引号，所以每列字符串没有用引号引起来． 但如果增加额外的引用参数即可将非数值数据用引号引起来．

```
writer = csv.writer(f, quoting=csv.QUOTE_NONNUMERIC)
```

现在每个字符串都被引起来了：

```
$ python csv_writer_quoted.py testout_quoted.csv
$ cat testout_quoted.csv
"Title 1","Title 2","Title 3"
1,"a","08/01/07"
2,"b","08/02/07"
3,"c","08/03/07"
4,"d","08/04/07"
5,"e","08/05/07"
6,"f","08/06/07"
7,"g","08/07/07"
8,"h","08/08/07"
9,"i","08/09/07"
10,"j","08/10/07"
```

43.5 引用

还有4种不同的引用选项，它们作为常量定义在csv模块中．

QUOTE_ALL 不管是什么类型，任何内容都加上引号

QUOTE_MINIMAL 这是默认的，使用指定的字符引用各个域(如果解析器被配置为相同的dialect和选项时，可能会让解析器在解析时产生混淆)

QUOTE_NONNUMERIC 引用那些不是整数或浮点数的域． 当使用读取对象时，如果输入的域是没有引号，那么它们会被转换成浮点数．

QUOTE_NONE 对所有的输出内容都不加引用，当使用读取对象时，引用字符看作是包含在每个域的值里(但在正常情况下，他们被当成定界符而被去掉)

43.6 Dialects

有很多参数可以控制csv模块如何解析或读取数据． 但这不是通过各自传递给读取对象和写入对象相关参数，而是统一起来，使用一个''dialect''对象． Dialect类可以通过名字注册，因此csv模块调用它时可以不必预先知道相关的参数设置． 标准库包含两种dialects： excel和excel-tabs．

``excel'' dialect是用于处理默认来自 Microsoft Excel格式的数据的，同样，也可以处理OpenOffice 或 NeoOffice的数据． 更多详细的dialect参数及其使用在csv模块的 节9.1.2 中有说明． ## dialect就是一些参数(定界符，换行符等等)设置，预先设置好的，但同样我们也可以自己设定，

43.7 DictReader 和DictWriter

另外，在处理数据序列时，csv模块包含了一些将行作为字典进行处理的类． 类DictReader和类DictWriter将每一行转成字典对象，可以传递字典键值，或者从输入文件的第一行中推断出键值．

```
import csv
import sys

f = open(sys.argv[1], 'rt')
try:

    reader = csv.DictReader(f)
    for row in reader:
        print row

finally:
     f.close()
```

基于字典的读取和写入对象可以当作是基于序列对象的进一步实现，它们使用相同的参数和API．唯一的差别就是前者把每一行当成是字典而不是列表或元组．

```
$ python csv_dictreader.py testdata.csv
{'Title 1': '1', 'Title 3': '08/18/07', 'Title 2': 'a'}
{'Title 1': '2', 'Title 3': '08/19/07', 'Title 2': 'b'}
{'Title 1': '3', 'Title 3': '08/20/07', 'Title 2': 'c'}
{'Title 1': '4', 'Title 3': '08/21/07', 'Title 2': 'd'}
{'Title 1': '5', 'Title 3': '08/22/07', 'Title 2': 'e'}
{'Title 1': '6', 'Title 3': '08/23/07', 'Title 2': 'f'}
{'Title 1': '7', 'Title 3': '08/24/07', 'Title 2': 'g'}
{'Title 1': '8', 'Title 3': '08/25/07', 'Title 2': 'h'}
{'Title 1': '9', 'Title 3': '08/26/07', 'Title 2': 'i'}
```

DictWriter必须指定一个域名字的列表，因为这样它才在输出时知道每个列的顺序．

```
import csv
import sys

f = open(sys.argv[1], 'wt')
try:

    fieldnames = ('Title 1', 'Title 2', 'Title 3')
    writer = csv.DictWriter(f, fieldnames=fieldnames)
    headers = {}
    for n in fieldnames:
        headers[n] = n
    writer.writerow(headers)
    for i in range(10):
        writer.writerow({ 'Title 1':i+1,
                          'Title 2':chr(ord('a') + i),
                          'Title 3':'08/%02d/07' % (i+1),
                          })

finally:

    f.close()
```

```
$ python csv_dictwriter.py testout.csv
$ cat testout.csv
Title 1,Title 2,Title 3
1,a,08/01/07
2,b,08/02/07
3,c,08/03/07
4,d,08/04/07
5,e,08/05/07
6,f,08/06/07
7,g,08/07/07
8,h,08/08/07
9,i,08/09/07
10,j,08/10/07
```

43.8 参考

- Python Module of the Week Home
- Download Sample Code
- PEP 305, CSV File API

CHAPTER

FORTYFOUR

PYMOTW: CALENDAR

- 模块： calendar
- 目的： 模块实现了面向年/月/星期的日期操作类
- Python 版本： 1.4+ 但在2.5中有更新

44.1 描述

calendar模块定义了一个Calendar类，封装了一些日期值(比如指定某月或某年的某星期的日期值)的计算． 另外，TextCalendar和HTMLCalendar类提供预先格式化好的输出．

44.2 格式化的例子

下面是一个非常简单的利用TextCalendar类及使用其prmonth()方法产生特定月份的格式化文本输出的例子．

```
import calendar

c = calendar.TextCalendar(calendar.SUNDAY)
c.prmonth(2007, 7)
```

这里，我告诉TextCalendar类以Sunday为一星期的开始，这遵循了美国人的习惯． 但是它默认是以Monday开始的，这是欧洲人的习惯．

输出看起来是这样的：

```
$ python PyMOTW/calendar/calendar_textcalendar.py
 July 2007
Su Mo Tu We Th Fr Sa
 1 2 3 4 5 6 7
 8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

同样时间的HTML输出稍有不同，但她没有prmonth()函数：

```
import calendar

c = calendar.HTMLCalendar(calendar.SUNDAY)
print c.formatmonth(2007, 7)
```

输出结果大致相同．

```
July 2007

Sun Mon     Tue     Wed     Thu     Fri     Sat
1   2       3       4       5       6       7
8   9       10      11      12      13      14
15  16      17      18      19      20      21
22  23      24      25      26      27      28
29  30      31
```

你可以看到，每一个表格单元都具有一个class属性，它对应于一星期的特定天.

```
<table border="0" cellpadding="0" cellspacing="0" class="month">
<tr><th colspan="7" class="month">July 2007</th></tr>
<tr><th class="sun">Sun</th><th class="mon">Mon</th><th class="tue">Tue</th><th class="wed">Wed</th><th class="
<tr><td class="sun">1</td><td class="mon">2</td><td class="tue">3</td><td class="wed">4</td><td class="thu">5</
<tr><td class="sun">8</td><td class="mon">9</td><td class="tue">10</td><td class="wed">11</td><td class="thu">1
<tr><td class="sun">15</td><td class="mon">16</td><td class="tue">17</td><td class="wed">18</td><td class="thu"
<tr><td class="sun">22</td><td class="mon">23</td><td class="tue">24</td><td class="wed">25</td><td class="thu"
<tr><td class="sun">29</td><td class="mon">30</td><td class="tue">31</td><td class="noday"> </td><td class="noda
</table>
```

如果你想要输出的格式不同于默认的格式，你可以使用calendar模块计算日期并且将这些数值组织成你给定的星期和月份范围，剩余的会自动转换. 类Calendar的函数weekheader(), monthcalendar()和yeardays2calendar()都可以用来完成这种任务.

调用yeardays2calendar()可以产生一个''月份 行''列表. 每个列表包含的月份可作为其他的星期列表. 星期是由日期数字(1-31)和星期数字(0-6)组成的元组列表. 如果某天落在月份之外，那么它的天数字为0.

```
pprint.pprint(calendar.Calendar(calendar.SUNDAY).yeardays2calendar(2007, 2))
```

这里，调用了yeardays2calendar(2007, 2) 返回2007年，以每行2月组织的数据.

```
$ python calendar_yeardays2calendar.py
 [[[(0, 6), (1, 0), (2, 1), (3, 2), (4, 3), (5, 4), (6, 5)],
 [(7, 6), (8, 0), (9, 1), (10, 2), (11, 3), (12, 4), (13, 5)],
 [(14, 6), (15, 0), (16, 1), (17, 2), (18, 3), (19, 4), (20, 5)],
 [(21, 6), (22, 0), (23, 1), (24, 2), (25, 3), (26, 4), (27, 5)],
 [(28, 6), (29, 0), (30, 1), (31, 2), (0, 3), (0, 4), (0, 5)]],
 [[(0, 6), (0, 0), (0, 1), (0, 2), (1, 3), (2, 4), (3, 5)],
 [(4, 6), (5, 0), (6, 1), (7, 2), (8, 3), (9, 4), (10, 5)],
 [(11, 6), (12, 0), (13, 1), (14, 2), (15, 3), (16, 4), (17, 5)],
 [(18, 6), (19, 0), (20, 1), (21, 2), (22, 3), (23, 4), (24, 5)],
 [(25, 6), (26, 0), (27, 1), (28, 2), (0, 3), (0, 4), (0, 5)]]],
 [[[(0, 6), (0, 0), (0, 1), (0, 2), (1, 3), (2, 4), (3, 5)],
 [(4, 6), (5, 0), (6, 1), (7, 2), (8, 3), (9, 4), (10, 5)],
 [(11, 6), (12, 0), (13, 1), (14, 2), (15, 3), (16, 4), (17, 5)],
 [(18, 6), (19, 0), (20, 1), (21, 2), (22, 3), (23, 4), (24, 5)],
 [(25, 6), (26, 0), (27, 1), (28, 2), (29, 3), (30, 4), (31, 5)]],
 [[(1, 6), (2, 0), (3, 1), (4, 2), (5, 3), (6, 4), (7, 5)],
 [(8, 6), (9, 0), (10, 1), (11, 2), (12, 3), (13, 4), (14, 5)],
 [(15, 6), (16, 0), (17, 1), (18, 2), (19, 3), (20, 4), (21, 5)],
 [(22, 6), (23, 0), (24, 1), (25, 2), (26, 3), (27, 4), (28, 5)],
 [(29, 6), (30, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5)]]],
 [[[(0, 6), (0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5)],
 [(6, 6), (7, 0), (8, 1), (9, 2), (10, 3), (11, 4), (12, 5)],
 [(13, 6), (14, 0), (15, 1), (16, 2), (17, 3), (18, 4), (19, 5)],
 [(20, 6), (21, 0), (22, 1), (23, 2), (24, 3), (25, 4), (26, 5)],
 [(27, 6), (28, 0), (29, 1), (30, 2), (31, 3), (0, 4), (0, 5)]],
 [[(0, 6), (0, 0), (0, 1), (0, 2), (0, 3), (1, 4), (2, 5)],
 [(3, 6), (4, 0), (5, 1), (6, 2), (7, 3), (8, 4), (9, 5)],
```

```
 [(10, 6), (11, 0), (12, 1), (13, 2), (14, 3), (15, 4), (16, 5)],
 [(17, 6), (18, 0), (19, 1), (20, 2), (21, 3), (22, 4), (23, 5)],
 [(24, 6), (25, 0), (26, 1), (27, 2), (28, 3), (29, 4), (30, 5)]],
 [[(1, 6), (2, 0), (3, 1), (4, 2), (5, 3), (6, 4), (7, 5)],
 [(8, 6), (9, 0), (10, 1), (11, 2), (12, 3), (13, 4), (14, 5)],
 [(15, 6), (16, 0), (17, 1), (18, 2), (19, 3), (20, 4), (21, 5)],
 [(22, 6), (23, 0), (24, 1), (25, 2), (26, 3), (27, 4), (28, 5)],
 [(29, 6), (30, 0), (31, 1), (0, 2), (0, 3), (0, 4), (0, 5)]],
 [[(0, 6), (0, 0), (0, 1), (1, 2), (2, 3), (3, 4), (4, 5)],
 [(5, 6), (6, 0), (7, 1), (8, 2), (9, 3), (10, 4), (11, 5)],
 [(12, 6), (13, 0), (14, 1), (15, 2), (16, 3), (17, 4), (18, 5)],
 [(19, 6), (20, 0), (21, 1), (22, 2), (23, 3), (24, 4), (25, 5)],
 [(26, 6), (27, 0), (28, 1), (29, 2), (30, 3), (31, 4), (0, 5)]]],
 [[[(0, 6), (0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 5)],
 [(2, 6), (3, 0), (4, 1), (5, 2), (6, 3), (7, 4), (8, 5)],
 [(9, 6), (10, 0), (11, 1), (12, 2), (13, 3), (14, 4), (15, 5)],
 [(16, 6), (17, 0), (18, 1), (19, 2), (20, 3), (21, 4), (22, 5)],
 [(23, 6), (24, 0), (25, 1), (26, 2), (27, 3), (28, 4), (29, 5)],
 [(30, 6), (0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5)]],
 [[(0, 6), (1, 0), (2, 1), (3, 2), (4, 3), (5, 4), (6, 5)],
 [(7, 6), (8, 0), (9, 1), (10, 2), (11, 3), (12, 4), (13, 5)],
 [(14, 6), (15, 0), (16, 1), (17, 2), (18, 3), (19, 4), (20, 5)],
 [(21, 6), (22, 0), (23, 1), (24, 2), (25, 3), (26, 4), (27, 5)],
 [(28, 6), (29, 0), (30, 1), (31, 2), (0, 3), (0, 4), (0, 5)]]],
 [[[(0, 6), (0, 0), (0, 1), (0, 2), (1, 3), (2, 4), (3, 5)],
 [(4, 6), (5, 0), (6, 1), (7, 2), (8, 3), (9, 4), (10, 5)],
 [(11, 6), (12, 0), (13, 1), (14, 2), (15, 3), (16, 4), (17, 5)],
 [(18, 6), (19, 0), (20, 1), (21, 2), (22, 3), (23, 4), (24, 5)],
 [(25, 6), (26, 0), (27, 1), (28, 2), (29, 3), (30, 4), (0, 5)]],
 [[(0, 6), (0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 5)],
 [(2, 6), (3, 0), (4, 1), (5, 2), (6, 3), (7, 4), (8, 5)],
 [(9, 6), (10, 0), (11, 1), (12, 2), (13, 3), (14, 4), (15, 5)],
 [(16, 6), (17, 0), (18, 1), (19, 2), (20, 3), (21, 4), (22, 5)],
 [(23, 6), (24, 0), (25, 1), (26, 2), (27, 3), (28, 4), (29, 5)],
 [(30, 6), (31, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5)]]]
```

这等价于使用formatyear()函数获得的数据

```
print calendar.TextCalendar(calendar.SUNDAY).formatyear(2007, 2, 1, 1, 2)
```

以相同的参数值传入，其输出结果：

```
$ python ./calendar_formatyear.py
 2007

  January               February
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6     1  2  3
 7  8  9 10 11 12 13  4  5  6  7  8  9 10
14 15 16 17 18 19 20 11 12 13 14 15 16 17
21 22 23 24 25 26 27 18 19 20 21 22 23 24
28 29 30 31           25 26 27 28

   March                 April
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
 1  2  3  1  2  3  4     1  2  3  4  5  6  7
 4  5  6  7  8  9 10  8  9 10 11 12 13 14
11 12 13 14 15 16 17 15 16 17 18 19 20 21
18 19 20 21 22 23 24 22 23 24 25 26 27 28
25 26 27 28 29 30 31 29 30
```

```
 May June
Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa
 1 2 3 4 5 1 2
 6 7 8 9 10 11 12 3 4 5 6 7 8 9
13 14 15 16 17 18 19 10 11 12 13 14 15 16
20 21 22 23 24 25 26 17 18 19 20 21 22 23
27 28 29 30 31 24 25 26 27 28 29 30

 July August
Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa
 1 2 3 4 5 6 7 1 2 3 4
 8 9 10 11 12 13 14 5 6 7 8 9 10 11
15 16 17 18 19 20 21 12 13 14 15 16 17 18
22 23 24 25 26 27 28 19 20 21 22 23 24 25
29 30 31 26 27 28 29 30 31

 September October
Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa
 1 1 2 3 4 5 6
 2 3 4 5 6 7 8 7 8 9 10 11 12 13
 9 10 11 12 13 14 15 14 15 16 17 18 19 20
16 17 18 19 20 21 22 21 22 23 24 25 26 27
23 24 25 26 27 28 29 28 29 30 31
30

 November December
Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa
 1 2 3 1
 4 5 6 7 8 9 10 2 3 4 5 6 7 8
11 12 13 14 15 16 17 9 10 11 12 13 14 15
18 19 20 21 22 23 24 16 17 18 19 20 21 22
25 26 27 28 29 30 23 24 25 26 27 28 29
30 31
```

如果你由于某些原因(如想在HTML输出中包含一些链接)需要自己组织输出格式， 你可以使用day_name， day_abbr， month_name， 和month_abbr这些模块属性． 它们可以为当前环境自动配置正确．

44.3 计算例子

虽然calendar模块大部分关注日历的不同格式的输出，但它也提供了一些使用其他方式处理日期的函数，如计算用于循环事件中的日期． 例如， Python Atlanta User's Group 在每个月的第一个星期四会面． 为了计算一年的所有会面日期，你可以使用monthcalendar()．

```
pprint.pprint(calendar.monthcalendar(2007, 7))
```

注意这里某些天的数字是0． 这是由于这个月的星期中的某些天和其他月中的重叠了．

```
 $ python calendar_monthcalendar.py
[[0, 0, 0, 0, 0, 0, 1],
 [2, 3, 4, 5, 6, 7, 8],
 [9, 10, 11, 12, 13, 14, 15],
 [16, 17, 18, 19, 20, 21, 22],
 [23, 24, 25, 26, 27, 28, 29],
 [30, 31, 0, 0, 0, 0, 0]]
```

记住，默认情况下，星期是以Monday作为一星期的开始，这可以通过函数setfirstweekday()来改变． 另一方面，由于calendar模块包含了一些用于在日期范围(由monthcalendar()返回得到)中进行定位的常量，在这个例子中直接跳过会更加方便．

为了计算PyATL在2007年的会面日期，假设是每个月的第二个星期四，我们可以使用0值来表明第一个星期的星期二是否被包含在某月中(如果某个月是以星期五开始的，那么对应的这周中的星期四的值为0)。

```
import calendar

# Show every month
for month in range(1, 13):

    # Compute the dates for each week which overlaps the month
    c = calendar.monthcalendar(2007, month)
    first_week = c[0]
    second_week = c[1]
    third_week = c[2]

    # If there is a Thursday in the first week, the second Thursday
    # is in the second week. Otherwise the second Thursday must
    # be in the third week.
    if first_week[calendar.THURSDAY]:
        meeting_date = second_week[calendar.THURSDAY]
    else:
        meeting_date = third_week[calendar.THURSDAY]

    print '%3s: %2s' % (month, meeting_date)
```

可以得到PyATL的会面日期为：

```
$ python calendar_secondthursday.py
1: 11
2: 8
3: 8
4: 12
5: 10
6: 14
7: 12
8: 9
9: 13
10: 11
11: 8
12: 13
```

44.4 参考

- Python Module of the Week Home
- Download Sample Code

CHAPTER

FORTYFIVE

PYMOTW: COMMANDS

commands模块包含一些用于处理Unix下shell命令及其输出的函数.

- 模块: commands
- 目的: 运行外部shell命令并能捕获退出状态码和输出结果.
- Python版本: 1.4+

45.1 描述

Note: 这个模块相对于 subprocess 来说是已经过时了.

commands模块主要有3个用于处理外部命令的函数.这些函数具有shell感知并且能返回被执行命令的输出和状态码.

45.2 getstatusoutput()

getstatusoutput() 函数通过shell运行一个命令, 之后返回退出状态码和文本输出(包含stdout和stderr的信息). 退出状态码是和C函数的wait()或os.wait()一样的, 是一个16位整数. 低字节包含杀死该进程的信号标识符. 当信号标识符为0时, 高字节表示了程序的退出状态. 如果产生了一个核心文件, 低字节的最高比特位会被设置1.

```
from commands import *

def run_command(cmd):

    print 'Running: "%s"' % cmd
    status, text = getstatusoutput(cmd)
    exit_code = status >> 8
    signal_num = status % 256
    print 'Signal: %d' % signal_num
    print 'Exit : %d' % exit_code
    print 'Core? : %s' % bool(exit_code / 256)
    print 'Output:'
    print text
    print


run_command('ls -l *.py')
run_command('ls -l *.notthere')
run_command('echo "WAITING TO BE KILLED"; read input')
```

这个例子中，运行的前2个命令正常退出，而第三个会一直阻塞(等待输入)直到从另一个shell中将它杀死。不要简单的使用Ctrl-C来中断程序，而是在另一终端中，使用ps和grep获得相关进程的标识，并使用kill发送信号给它.

```
$ python commands_getstatusoutput.py
Running: "ls -l *.py"
Signal: 0
Exit : 0
Core? : False
Output:
-rw-r--r-- 1 dhellman dhellman 1191 Oct 21 09:41 __init__.py
-rw-r--r-- 1 dhellman dhellman 1321 Oct 21 09:48 commands_getoutput.py
-rw-r--r-- 1 dhellman dhellman 1265 Oct 21 09:50 commands_getstatus.py
-rw-r--r-- 1 dhellman dhellman 1626 Oct 21 10:10 commands_getstatusoutput.py

Running: "ls -l *.notthere"
Signal: 0
Exit : 1
Core? : False
Output:
ls: *.notthere: No such file or directory

Running: "echo "WAITING TO BE KILLED"; read input"
Signal: 1
Exit : 0
Core? : False
Output:
WAITING TO BE KILLED
```

我使用了''kill -HUP $PID''来杀死这个读进程.

45.3 getoutput()

如果退出状态码对于你的应用来说是没有用的，你使用getoutput()可以仅仅获得文本输出.

```
from commands import *

text = getoutput('ls -l *.py')
print 'ls -l *.py:'
print text

print

text = getoutput('ls -l *.notthere')
print 'ls -l *.py:'
print text


$ python commands_getoutput.py
ls -l *.py:
-rw-r--r-- 1 dhellman dhellman 1191 Oct 21 09:41 __init__.py
-rw-r--r-- 1 dhellman dhellman 1321 Oct 21 09:48 commands_getoutput.py
-rw-r--r-- 1 dhellman dhellman 1265 Oct 21 09:50 commands_getstatus.py
-rw-r--r-- 1 dhellman dhellman 1626 Oct 21 10:10 commands_getstatusoutput.py

ls -l *.py:
ls: *.notthere: No such file or directory
```

45.4 getstatus()

和你期望的可能不一样，getstatus()函数在运行一个命令之后不是返回状态码． 而是，传递一个参数给它，这个参数是一个文件名，被合并到''ls -ld''中，运行该命令之后返回相应文本输出，即获得该文件的相关信息．

```
from commands import *

status = getstatus('commands_getstatus.py')
print 'commands_getstatus.py:', status
status = getstatus('notthere.py')
print 'notthere.py:', status
status = getstatus('$filename')
print '$filename:', status
```

从输出可以看到，参数中的字符$不会被转义，所以相关环境变量也不会被扩展．

```
$ python commands_getstatus.py
commands_getstatus.py: -rw-r--r-- 1 dhellman dhellman 1387 Oct 21 10:19 commands_getstatus.py
notthere.py: ls: notthere.py: No such file or directory
$filename: ls: $filename: No such file or directory
```

45.5 参考

- Python Module of the Week Home
- Download Sample Code

CHAPTER

FORTYSIX

PYMOTW: ATEXIT

- 模块： atexit
- 目的： 当一程序关闭时，注册一个需要被调用的函数.
- Python版本： 2.1.3+

46.1 描述

atexit模块提供了一个简单的接口，一般情况下，用于注册当程序关闭时需要调用的函数. sys模块虽然也提供了类似功能的钩子，sys.exitfunc，但是它只能注册一个函数. 而atexit注册表可以被多个模块和库同时使用.

46.2 示例

一个简单的例子，它通过atexit.register()注册一个函数:

```
import atexit

def all_done():

    print 'all_done()'


print 'Registering'
atexit.register(all_done)
print 'Registered'
```

由于上面的程序实际上不做任何其他事情，所以在程序关闭时立即调用了all_done():

```
$ python atexit_simple.py
Registering
Registered
all_done()
```

注册多个函数也是有可能的，并且可以传递参数给它们. 这在安全地断开数据库连接，或删除临时文件等情况下，都是非常有用的. 因为可以将参数传递给注册函数，所以我们甚至不需要保留一个单独需要清理东西的列表 -- 我们只需要多次注册一个清理函数.

```
def my_cleanup(name):

    print 'my_cleanup(%s)' % name
```

```
atexit.register(my_cleanup, 'first')
atexit.register(my_cleanup, 'second')
atexit.register(my_cleanup, 'third')
```

注意： 这里函数调用的顺序是按照它们注册顺序的逆序的． 这就允许模块按照它们被导入顺序的逆序(由此来注册它们的atexit函数)来清理， 这样可以减少模块间的依赖关系．

```
$ python atexit_multiple.py
my_cleanup(third)
my_cleanup(second)
my_cleanup(first)
```

46.3 什么时候atexit函数不被调用?

由atexit注册的那些回调函数不会被调用的情况有以下几种:

- 程序由于收到信号退出． ## 这个信号是??
- 直接调用os._exit()
- 在python解释器中， 检测到很严重的错误．

为了举例程序是通过信号被杀死的， 我们可以修改 subprocess summary 中的一个例子． 这里有2个文件需要被调用， 分别是父进程和子进程:

```
import os
import signal
import subprocess
import time

proc = subprocess.Popen('atexit_signal_child.py')
print 'PARENT: Pausing before sending signal...'
time.sleep(1)
print 'PARENT: Signaling %s' % proc.pid
os.kill(proc.pid, signal.SIGTERM)
```

子进程中设置atexit回调函数， 以证明它没有被调用．

```
import atexit
import time

def not_called():
    print 'CHILD: atexit handler should not have been called'

print 'CHILD: Registering atexit handler'
atexit.register(not_called)

print 'CHILD: Pausing to wait for signal'
time.sleep(5)
```

运行之后， 输出信息如下:

```
$ python atexit_signal_parent.py
CHILD: Registering atexit handler
CHILD: Pausing to wait for signal
PARENT: Pausing before sending signal...
PARENT: Signaling 2038
```

注意到子进程中没有调用not_called(), 所以就没有打印出相应的信息. 类似的, 如果一个程序绕过正常的退出路径的话, 它也不会执行atexit回调函数.

```
import atexit
import os

def not_called():

    print 'This should not be called'


print 'Registering'
atexit.register(not_called)
print 'Registered'

print 'Exiting...'
os._exit(0)
```

由于我们直接调用os._eixt()退出程序, 所以atexit的回调函数不会被调用.

```
$ python atexit_os_exit.py
Registering
Registered
Exiting...
```

如果我们使用sys.exit()的话, atexit的回调函数仍然会被执行.

```
import atexit
import sys

def all_done():

    print 'all_done()'


print 'Registering'
atexit.register(all_done)
print 'Registered'

print 'Exiting...'
sys.exit()


$ python atexit_sys_exit.py
Registering
Registered
Exiting...
all_done()
```

在Python解释器中模拟出一个严重错误来验证程序的退出也没有调用atexit回调函数,,,这个就留给读者了吧. :-)

46.4 在atexit回调函数中的异常

在atexit回调函数中引发异常后的回溯信息会被输出到控制台, 最后引发的异常会重新被抛出以作为程序的最终错误信息.

```
def exit_with_exception(message):
    raise RuntimeError(message)

atexit.register(exit_with_exception, 'Registered first')
atexit.register(exit_with_exception, 'Registered second')
```

Note: 注册时的顺序决定了执行的顺序． 如果一个回调函数中的错误引入了另外一个错误(比他先注册但是比他后调用)，那么，最终的信息可能对于用户来说不是最有用的． ## 这个例子，程序首先在second上抛出异常，但最终显示的是first异常信息，这主要还是因为atexit回调函数不是按照注册顺序来执行的．

```
$ python atexit_exception.py
Error in atexit._run_exitfuncs:
Traceback (most recent call last):
 File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/atexit.py", line 24, in _run_exitfuncs
 func(*targs, **kargs)
 File "atexit_exception.py", line 36, in exit_with_exception
 raise RuntimeError(message)
RuntimeError: Registered second
Error in atexit._run_exitfuncs:
Traceback (most recent call last):
 File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/atexit.py", line 24, in _run_exitfuncs
 func(*targs, **kargs)
 File "atexit_exception.py", line 36, in exit_with_exception
 raise RuntimeError(message)
RuntimeError: Registered first
Error in sys.exitfunc:
Traceback (most recent call last):
 File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/atexit.py", line 24, in _run_exitfuncs
 func(*targs, **kargs)
 File "atexit_exception.py", line 36, in exit_with_exception
 raise RuntimeError(message)
RuntimeError: Registered first
```

一般情况下，你可以设置在清理函数中安静的处理和记录所有异常，这样在程序退出时，就不会使输出信息显得很乱．

46.5 参考

- Python Module of the Week
- Sample Code

CHAPTER

FORTYSEVEN

PYMOTW: COLLECTIONS

collections模块包含了一些除了内置类型，如列表，字典外的容器数据类型.

- 模块: collections
- 目的: 数据类型的包含容器.
- Python 版本: 2.4 +

47.1 双端队列

一个双头队列，或者''双端队列''，支持从每一端上增加和删除元素. 更常用的像栈和队列，它们可看成是双端队列的特殊情况，即被限制为输入和输出只能从一端进行.

因为双端队列是一种序列容器，所以它们支持一些列表也支持的相同操作，如利用__getitem__()检查内部元素，计算长度，根据标识符的匹配与否来移除某个元素.

```
import collections

d = collections.deque('abcdefg')

print 'Deque:', d
print 'Length:', len(d)
print 'Left end:', d[0]
print 'Right end:', d[-1]


d.remove('c')

print 'remove(c):', d


$ python collections_deque.py
Deque: deque(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
Length: 7
Left end: a
Right end: g
remove(c): deque(['a', 'b', 'd', 'e', 'f', 'g'])
```

一个双端队列可以从每一端填入元素，在Python实现中使用词语 ``left'' 和 ``right''.

```
import collections

# Add to the right ##    ,  extend append
d = collections.deque()
d.extend('abcdefg')
print 'extend :', d
```

```
d.append('h')
print 'append :', d

# Add to the left ##    ,  extendleft appendleft
d = collections.deque()
d.extendleft('abcdefg')
print 'extendleft:', d
d.appendleft('h')
print 'appendleft:', d
```

Note: extendleft()将对所有的输入进行，其执行效果等价于对每一个元素进行appendleft()。最终的结果是这个双端队列包含了一个逆序的输入元素序列。

```
$ python collections_deque_populating.py
extend : deque(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
append : deque(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
extendleft: deque(['g', 'f', 'e', 'd', 'c', 'b', 'a'])
appendleft: deque(['h', 'g', 'f', 'e', 'd', 'c', 'b', 'a'])
```

类似的，双端队列可以同时从两端或只从一端获取元素，具体得看你在算法中是如何写的。

```
import collections

print 'From the right:'
d = collections.deque('abcdefg')

while True:
    try:
        print d.pop() ##
    except IndexError:
        break


print 'From the left:'
d = collections.deque('abcdefg')

while True:
    try:
        print d.popleft() ##
    except IndexError:
        break


$ python collections_deque_consuming.py
From the right:
g
f
e
d
c
b
a
From the left:
a
b
c
d
e
f
g
```

因为双端队列是线程安全的，所以你甚至可以在独立线程中从它的两端同时获取元素.

```
import collections
import threading
import time

candle = collections.deque(xrange(11))

def burn(direction, nextSource):
    while True:
        try:
            next = nextSource()
        except IndexError:
            break
        else:
            print '%8s: %s' % (direction, next)
            time.sleep(0.1)
    print '%8s done' % direction
    return


left = threading.Thread(target=burn, args=('Left', candle.popleft))
right = threading.Thread(target=burn, args=('Right', candle.pop))

left.start()
right.start()

left.join()
right.join()
```

```
$ python collections_deque_both_ends.py
Left: 0
Right: 10
Left: 1
Right: 9
Left: 2
Right: 8
Left: 3
Right: 7
Left: 4
Right: 6
Left: 5
Right done
Left done
```

另外一个双端队列有用的功能是在每一个方向上转动一些项，以跳过某些项.

```
import collections

d = collections.deque(xrange(10))
print 'Normal :', d

d = collections.deque(xrange(10))
d.rotate(2) ##
print 'Right rotation:', d

d = collections.deque(xrange(10))
d.rotate(-2) ##
print 'Left rotation :', d
```

从右边旋转(使用一个正数)双端队列，将项向右移动至右端末尾，对于超过右边界的项又被移动到双端队列的左边. 从左边旋转(使用一个负数)双端队列，将项向左边移至左端末尾，对于超过左边界的

项又被移动到双端队列的右边.

```
$ python collections_deque_rotate.py
Normal : deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Right rotation: deque([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])
Left rotation : deque([2, 3, 4, 5, 6, 7, 8, 9, 0, 1])
```

47.2 defaultdict

标准的字典包含了setdefault()，用于设置一个默认值，即当查找一个不存在的键值时用这个默认值来代替. 同样的，defaultdict能够让你在初始化时指定默认值.

```
import collections

def default_factory():
    return 'default value'


d = collections.defaultdict(default_factory, foo='bar')
print d
print d['foo']
print d['bar']


$ python collections_defaultdict.py
defaultdict(<function default_factory at 0x7ca70>, {'foo': 'bar'})
bar
default value
```

这个例子中，所有键都使用相同的默认值. 当默认的是一个用于集成或累计值的类型，如一个列表，集合，甚至是整型时会更有用处. 标准库文档包含了许多使用defaultdict的例子.

##更多的defaultdict例子

defaultdict的第一个参数default_factory，提供了初始值，默认为None，余下的参数被看作是字典的键值对.

例子1: 字典值默认是一个空列表

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
 d[k].append(v)

>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

例子2: 和上同样的效果，只是使用了dict

```
>>> d = {}
>>> for k, v in s:
    d.setdefault(k, []).append(v)

>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

例子3: 值默认为整型，其整型值默认为0

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
 d[k] += 1

>>> d.items()
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

例子4:

```
>>> def constant_factory(value):
... return itertools.repeat(value).next
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d ##     ,      key
'John ran to <missing>'
```

例子5:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
 d[k].add(v)

>>> d.items()
[('blue', set([2, 4])), ('red', set([1, 3]))]
```

47.3 参考

- Wikipedia: Deque
- Deque Recipes
- defaultdict examples
- James Tauber: Evolution of Default Dictionaries in Python
- Python Module of the Week Home
- Download Sample Code

CHAPTER

FORTYEIGHT

PYMOTW: FILEINPUT

- 模块： fileinput
- 目的： 创建命令行过滤程序.
- Python 版本： ?

48.1 描述

让我们从 fileinput 模块开始这个系列的学习吧. 这是一个非常有用的模块, 用于创建处理文本文件中过滤信息的命令行程序. 例如, 最近为我的朋友 Patrick 写了个应用 m3utorss , 用于将一些小demo转成podcastable格式便于记录.

程序的输入是一个或多个m3u文件, 列出了所有的mp3文件. 输出是一个XML文件, 看起来有点像RSS feed(简单起见, 输出到stdout). 为了处理输入, 我需要对一个文件名列表一次处理:

- 打开每个文件.
- 读取一个文件的每一行.
- 标记出指向一个mp3文件的行.
- 如果有, 提出取mp3文件的信息用于生成RSS feed.
- 输出.

我本应该手工写所有文件的处理代码. 但没有非常复杂, 在一些测试之后, 我确信, 连错误处理都可以正确处理. 但是使用fileinput模块, 我可以不用这么麻烦的考虑很多东西, 主需要写如下:

```
import fileinput
import sys


def generate_item(filename):
    """Process the named file go generate an RSS item.
    """
    print filename

for line in fileinput.input(sys.argv[1:]):
    mp3filename = line.strip()
    if not mp3filename or mp3filename.startswith('#'):
            continue
    generate_item(mp3filename)
```

这段代码中相关的代码是for循环中的. fileinput.input()函数将参数看成是要检测是文件名列表. 如果这个列表是空的, 那么模块会从标准输入中获取. 它返回的是一个迭代器, 依次返回正在处理的文本文件中的每一行. 因此, 我所要做的就是循环处理每行, 跳过空白和注释, 寻找mp3文件.

在这个例子中, 我不需要关心正在处理哪个文件和具体的哪个行. 可能还是其他工具(例如, 类grep搜索工具). fileinput模块也包含了访问这些信息的函数(filename(), filelineno(), lineno(), 等等). 具体使用可参考fileinput的标准库文档.

48.2 参考

- fileinput 该模块的标准库文档.
- Patrick Bryant 一位歌曲/歌词作家.

CHAPTER

FORTYNINE

PYMOTW: GETOPT

- 模块： getopt
- 目的： 命令行选项解析
- python版本： 1.4+

49.1 描述

getopt模块是老派的命令行选项解析器， 兼容Unix函数getopt(). 它解析一个参数序列， 如sys.argv, 返回(option, argument)对和其他非选项的参数序列.

支持的选项语法包括:

- -a
- -bval
- -b val
- --noarg
- --witharg=val
- --witharg val

49.2 函数参数

getopt函数可带三个参数:

第一个参数是待解析的参数序列， 它通常来自sys.argv[1:](忽略sys.arg[0], 因为它是程序名字).

第二个参数是选项定义字符串用于指示单个字符选项. 如果一个选项需要一个参数， 那么选项字符之后会跟着个冒号. ## 这个冒号代表该选项的值.

第三个参数， 如果使用的话， 应该是一个长类型选项名字序列. 长类型选项包含多个字符， 如--noarg或--witharg. 序列中的选项名字不应该包含前缀符'-`. 如果任何一个长选项需要一个参数， 那么它需要后缀符''=''.

短形式和长形式选项可以在一个调用中结合起来定义.

49.3 短形式选项

如果一个程序需要带2个选项， -a和-b, b选项需要一个参数, 那么值应为''ab:''.

```
print getopt.getopt(['-a', '-bval', '-c', 'val'], 'ab:c:')
```

```
$ python getopt_short.py
([('-a', ''), ('-b', 'val'), ('-c', 'val')], [])
```

49.4 长形式选项

如果程序带2个选项，-noarg和-witharg，其参数序列应为[`noarg', `witharg='].

```
print getopt.getopt([ '--noarg', '--witharg', 'val', '--witharg2=another' ],'',[ 'noarg', 'witharg=', 'witharg2
```

```
$ python getopt_long.py
([('--noarg', ''), ('--witharg', 'val'), ('--witharg2', 'another')], [])
```

例子：

接下来一个复杂点的例子，它带5个选项： -o，-v，--output，--verbose，和 --version. 选项-o，--output和--version需要携带参数.

```
import getopt
import sys

version = '1.0'
verbose = False
output_filename = 'default.out'

print 'ARGV :', sys.argv[1:]

options, remainder = getopt.getopt(sys.argv[1:], 'o:v', ['output=',
                                    'verbose',
                                    'version=',
                     ])
print 'OPTIONS :', options

for opt, arg in options:
    if opt in ('-o', '--output'):
        output_filename = arg
    elif opt in ('-v', '--verbose'):
        verbose = True
    elif opt == '--version':
        version = arg

print 'VERSION :', version
print 'VERBOSE :', verbose
print 'OUTPUT :', output_filename
print 'REMAINING :', remainder
```

程序可以多种方式调用.

```
$ python ./getopt_example.py
ARGV : []
OPTIONS : []
VERSION : 1.0
VERBOSE : False
OUTPUT : default.out
REMAINING : []
```

可以将单个字符选项和参数分隔开：

```
$ python ./getopt_example.py -o foo
ARGV : ['-o', 'foo']
OPTIONS : [('-o', 'foo')]
VERSION : 1.0
VERBOSE : False
OUTPUT : foo
REMAINING : []
```

或者结合起来：

```
$ python ./getopt_example.py -ofoo
ARGV : ['-ofoo']
OPTIONS : [('-o', 'foo')]
VERSION : 1.0
VERBOSE : False
OUTPUT : foo
REMAINING : []
```

长形式选项可以被简单的分离：

```
$ python ./getopt_example.py --output foo
ARGV : ['--output', 'foo']
OPTIONS : [('--output', 'foo')]
VERSION : 1.0
VERBOSE : False
OUTPUT : foo
REMAINING : []
```

或者使用'='结合：

```
$ python ./getopt_example.py --output=foo
ARGV : ['--output=foo']
OPTIONS : [('--output', 'foo')]
VERSION : 1.0
VERBOSE : False
OUTPUT : foo
REMAINING : []
```

49.5 长形式选项的缩写

对于长形式的选项， 我们可以不必全部拼写出来， 而只要提供一个唯一的前缀以确定到底是哪个选项即可：

```
$ python ./getopt_example.py --o foo
ARGV : ['--o', 'foo']
OPTIONS : [('--output', 'foo')]
VERSION : 1.0
VERBOSE : False
OUTPUT : foo
REMAINING : []
```

如果唯一前缀不存在， 则会有抛出异常.

```
$ python ./getopt_example.py --ver 2.0
ARGV : ['--ver', '2.0']
Traceback (most recent call last):
```

```
  File "./getopt_example.py", line 43, in 'version=',
  File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/getopt.py", line 89, in getopt opts, ar
  File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/getopt.py", line 153, in do_longs has_a
  File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/getopt.py", line 180, in long_has_args
raise GetoptError('option --%s not a unique prefix' % opt, opt)
getopt.GetoptError: option --ver not a unique prefix
```

选项解析过程会在遇到第一个非选项参数之后马上停止.

```
$ python ./getopt_example.py -v not_an_option --output foo
ARGV : ['-v', 'not_an_option', '--output', 'foo']
OPTIONS : [('-v', '')]
VERSION : 1.0
VERBOSE : True
OUTPUT : default.out
REMAINING : ['not_an_option', '--output', 'foo']
```

49.6 GNU风格的选项解析

这是Python 2.3中的新特性，它提供了一个函数，叫做gnu_getopt(). 该函数允许选项和非选项参数以任意顺序混合在命令行中. 我们改变了先前的那个例子，可以很明显的看出差别所在:

```
import getopt
import sys

version = '1.0'
verbose = False
output_filename = 'default.out'

print 'ARGV      :', sys.argv[1:]

options, remainder = getopt.gnu_getopt(sys.argv[1:], 'o:v',
                ['output=',
                 'verbose',
                 'version=',
                 ])

print 'OPTIONS   :', options

for opt, arg in options:
    if opt in ('-o', '--output'):
        output_filename = arg
    elif opt in ('-v', '--verbose'):
        verbose = True
    elif opt == '--version':
        version = arg

print 'VERSION   :', version
print 'VERBOSE   :', verbose
print 'OUTPUT    :', output_filename
print 'REMAINING :', remainder
```

输出结果如下:

```
$ python ./getopt_gnu.py -v not_an_option --output foo
ARGV : ['-v', 'not_an_option', '--output', 'foo']
OPTIONS : [('-v', ''), ('--output', 'foo')]
VERSION : 1.0
```

```
VERBOSE : True
OUTPUT : foo
REMAINING : ['not_an_option']
```

49.7 特殊的例子: --

如果getopt在输入参数序列中遇到--, 它就会马上停止剩余参数的解析.

```
$ python ./getopt_example.py -v -- --output foo
ARGV : ['-v', '--', '--output', 'foo']
OPTIONS : [('-v', '')]
VERSION : 1.0
VERBOSE : True
OUTPUT : default.out
REMAINING : ['--output', 'foo']
```

49.8 参考

- getopt - Linux Command - Unix Command

CHAPTER

FIFTY

PYMOTW: HASHLIB

- 模块： hashlib
- 目的： 加密哈希和消息摘要
- python版本： 2.5

50.1 描述

hashlib模块封装了md5和sha模块，形成一致的API．想使用特定的哈希算法，可以使用合适的构造函数来创建一个哈希对象．不管具体使用了什么样的哈希算法，都可以使用相同的API来操作．

hashlib是基于OpenSSL的，库中提供的所有算法应该是可用的，包括：

- md5()
- sha1()
- sha224()
- sha256()
- sha384()
- sha512()

50.2 MD5例子

计算一块数据(这里是一个ASCII字符串)的MD5摘要，创建哈希，增加数据，然后计算摘要．

```
import hashlib

from hashlib_data import lorem

h = hashlib.md5()
h.update(lorem)
print h.hexdigest()
```

这个例子使用了hexdigest()方法而不是digest()方法，这样可让输出结果可打印出来．如果想获得二进制的摘要值，你可以使用digest()．

```
$ python hashlib_md5.py
c3abe541f361b1bfbbcfecbf53aad1fb
```

50.3 SHA1例子

对刚才相同的数据产生一个SHA1摘要，其计算过程是类似的：

```
import hashlib

from hashlib_data import lorem

h = hashlib.sha1()
h.update(lorem)
print h.hexdigest()
```

当然，产生的摘要值由于使用了不同的算法而不同.

```
$ python hashlib_sha1.py
ac2a96a4237886637d5352d606d7a7b6d7ad2f29
```

50.4 new()

有时候，通过一个字符串形式的名字来引用算法，而不是直接使用构造函数，这可能在使用时更加方便. 例如，可以在一个配置文件中存储哈希类型. 在这些情况下，我们可以直接使用new()函数来创建一个新的哈希计算器.

```
import hashlib
import sys

try:
    hash_name = sys.argv[1]
except IndexError:
    print 'Specify the hash name as the first argument.'
else:
    try:
        data = sys.argv[2]
    except IndexError:
        from hashlib_data import lorem as data

    h = hashlib.new(hash_name)
    h.update(data)
    print h.hexdigest()
```

当使用不同的参数运行时：

```
$ python hashlib_new.py sha1
ac2a96a4237886637d5352d606d7a7b6d7ad2f29
$ python hashlib_new.py sha256
88b7404fc192fcdb9bb1dba1ad118aa1ccd580e9faa110d12b4d63988cf20332
$ python hashlib_new.py sha512
f58c6935ef9d5a94d296207ee4a7d9bba411539d8677482b7e9d60e4b7137f68d25f9747cab62fe752ec5ed1e5b2fa4cdbc8c9203267f99
$ python hashlib_new.py md5
c3abe541f361b1bfbbcfecbf53aad1fb
```

50.5 调用update()多次

哈希计算器的update()函数可以重复被调用. 每次会根据新增加的文本更新摘要值. 例如：当文件非常大，而不能一次性读入内存时，使用不断update()会很有效果.

```
import hashlib

from hashlib_data import lorem

h = hashlib.md5()
h.update(lorem)
all_at_once = h.hexdigest()

def chunkize(size, text):
    "Return parts of the text in size-based increments."
    start = 0
    while start < len(text):
        chunk = text[start:start+size]
        yield chunk
        start += size
    return

h = hashlib.md5()
for chunk in chunkize(64, lorem):
    h.update(chunk)
line_by_line = h.hexdigest()

print 'All at once :', all_at_once
print 'Line by line:', line_by_line
print 'Same :', (all_at_once == line_by_line)
```

这个例子中有一点点小花招, 因为他也可以处理小文件, 但她说明了在不断读取数据时如何递增的使用update()来产生新的摘要.

```
$ python hashlib_update.py
All at once : c3abe541f361b1bfbbcfecbf53aad1fb
Line by line: c3abe541f361b1bfbbcfecbf53aad1fb
Same : True
```

50.6 参考

- Voidspace: IronPython and Hashlib
- hmac module

CHAPTER

FIFTYONE

PYMOTW: UNITTEST

- 模块： unittest
- 目的： 自动测试框架
- python版本： 2.1+

51.1 描述

Python的unittest模块，有时被称为PyUnit，它是基于由Kent Beck 和Erich Gamma设计的XUnit框架的. 这种模型被重复使用在其他很多语言(如C，perl，Java和Smalltalk)中. 这个由unittest实现的框架支持fixtures，test suites，和a test runner，以便能自动测试你的代码.

51.2 基本测试结构

unittest模块中定义的测试包含两个部分： 管理测试“fixtures”的代码，和本身的测试代码. 每个测试继承unittest.TestCase并被创建，并且它可以被重载或增加相关方法. 例如：

```
import unittest

class SimplisticTest(unittest.TestCase):
    def test(self):
        self.failUnless(True)

if __name__ == '__main__':
    unittest.main()
```

在这个例子中，SimplisticTest仅包含一个test()方法，如果不是True而是false时即会失败.

51.3 测试运行

最简单的运行unitest测试的方式是在每个测试文件的底部包含下面的语句：

```
if __name__ == '__main__':
    unittest.main()
```

然后，直接从命令行中运行这个脚本：

```
$ python unittest_simple.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

简短的输出中包含了测试所需的时间信息，也包含每项测试的状态指标(第一行的''.''表示通过一个测试项)。使用-v选项可以在测试结果中显示更详细的信息。

```
$ python unittest_simple.py -v
test (__main__.SimplisticTest) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

51.4 测试结果输出

测试包含3个可能的结果输出：

ok：测试通过。FAIL：测试没有通过，并且引发一个AssertionError异常。ERROR：测试过程中引发一个不是AssertionError的异常。

这里不能直接让一个测试''pass''，所以测试的状态由是否存在某个异常决定。

```
class OutcomesTest(unittest.TestCase):

    def testPass(self):
        return

    def testFail(self):
        self.failIf(True)

    def testError(self):
        raise RuntimeError('Test error!')
```

当测试失败或产生一个错误，那么在输出中会包含相关回溯信息。

```
$ python unittest_outcomes.py
EF.
======================================================================
ERROR: testError (__main__.OutcomesTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittest_outcomes.py", line 43, in testError
  raise RuntimeError('Test error!')
RuntimeError: Test error!

======================================================================
FAIL: testFail (__main__.OutcomesTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittest_outcomes.py", line 40, in testFail
  self.failIf(True)
AssertionError

----------------------------------------------------------------------
```

```
Ran 3 tests in 0.002s

FAILED (failures=1, errors=1)
```

在上面的例子中，testFail()失败，回溯信息显示了引起失败的代码行． 大部分人可以阅读代码的测试输出来找出引起测试失败的语义． 为了能更容易的理解测试失败的本质原因，可以使用 fail*() 和 assert*()方法，并让它们接收msg参数，指示在输出中显示更详细的错误信息．

```
class FailureMessageTest(unittest.TestCase):
    def testFail(self):
        self.failIf(True, 'failure message goes here')


$ python unittest_failwithmessage.py -vtestFail (__main__.FailureMessageTest) ... FAIL

======================================================================
FAIL: testFail (__main__.FailureMessageTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittest_failwithmessage.py", line 37, in testFail
  self.failIf(True, 'failure message goes here')
AssertionError: failure message goes here

----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

51.5 断言的本质

大多数的测试在特定条件下进行断言测试． 编写truth-checking测试的方法也有很多，采用哪个方法主要由测试者的个人习惯和想获得什么样的测试结果来决定． 如果由代码产生的值可视为真，那么可以使用 failUnless() 和 assertTrue()方法，如果该值可被评价为假，那么，使用failIf() 和 assertFalse()方法会更有意义．

```
class TruthTest(unittest.TestCase):
    def testFailUnless(self):
        self.failUnless(True)

    def testAssertTrue(self):
        self.assertTrue(True)

    def testFailIf(self):
        self.failIf(False)

    def testAssertFalse(self):
        self.assertFalse(False)
```

51.6 等价测试

这是一个特殊的测试类型，unittest包含了测试俩个值是否相等的方法．

```
class EqualityTest(unittest.TestCase):
    def testEqual(self):
        self.failUnlessEqual(1, 3-2)
```

```
    def testNotEqual(self):
        self.failIfEqual(2, 3-2)
```

这些特殊的测试使用比较方便，因为当测试失败时，被比较的两个值会显示在失败信息中.

```
class InequalityTest(unittest.TestCase):
    def testEqual(self):
        self.failIfEqual(1, 3-2)

    def testNotEqual(self):
        self.failUnlessEqual(2, 3-2)
```

当运行这些测试，可以看到:

```
$ python unittest_notequal.py -v
testEqual (__main__.EqualityTest) ... FAIL
testNotEqual (__main__.EqualityTest) ... FAIL

======================================================================
FAIL: testEqual (__main__.EqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittest_notequal.py", line 37, in testEqual
  self.failIfEqual(1, 3-2)
AssertionError: 1 == 1

======================================================================
FAIL: testNotEqual (__main__.EqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittest_notequal.py", line 40, in testNotEqual
  self.failUnlessEqual(2, 3-2)
AssertionError: 2 != 1

----------------------------------------------------------------------
Ran 2 tests in 0.002s

FAILED (failures=2)
```

51.7 近似相等?

除了严格的相等外，对于浮点数来说，可以测试两个数是否近似相等，这种情况下，可以使用 `failIfAlmostEqual()` 和 `failUnlessAlmostEqual()`.

```
class AlmostEqualTest(unittest.TestCase):
    def testNotAlmostEqual(self):
        self.failIfAlmostEqual(1.1, 3.3-2.0, places=1)

    def testAlmostEqual(self):
        self.failUnlessAlmostEqual(1.1, 3.3-2.0, places=0)
```

它们的参数为2个待比较的数值，`places`表示小数位数，指示测试时要考虑的小数位数.

```
$ python unittest_almostequal.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s
```

```
OK
```

51.8 测试中的异常

之前也提到过，如果一个测试引发了一个异常，这会在测试过程中会被看成是一个错误．这有利于显示在你修改了现有的测试代码后会出现的错误．然而有时，当你想在测试时确认是哪些代码产生了异常，如一个对象的某个属性被赋于一无效值，这些情况下，使用TestCase.fallUnlessRaises()比直接捕获异常更容易简洁代码．比较下面的两个测试：

```
def raises_error(*args, **kwds):
    print args, kwds
    raise ValueError('Invalid value: ' + str(args) + str(kwds))

class ExceptionTest(unittest.TestCase):
    def testTrapLocally(self):
        try:
            raises_error('a', b='c')
        except ValueError:
            pass
        else:
            self.fail('Did not see ValueError')

    def testFailUnlessRaises(self):
        self.failUnlessRaises(ValueError, raises_error, 'a', b='c')
```

两个测试的结果是一样的，但第二个测试使用了failUnlessRaises()，显得更加简洁．

```
$ python unittest_exception.py -v
testFailUnlessRaises (__main__.ExceptionTest) ... ('a',) {'b': 'c'}
ok
testTrapLocally (__main__.ExceptionTest) ... ('a',) {'b': 'c'}
ok

----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

51.9 Test Fixtures

Fixtures是一个测试过程中所有需要的资源．例如，如果你正在写多个针对同一个类的测试用例，这些测试用例都需要这个类的一个实例来作测试，其他测试要用到的fixtures包括数据库连接和临时文件(许多人争论使用外部资源来让这些测试不像是''单元''测试，但是他们仍然使用它们来测试，结果也仍然是可用的)．TestCase包括一些特殊的钩子用于配置和清理这些fixtures．重载setUp()用于配置fixtures．重载tearDown()用于清理fixtures．

```
class FixturesTest(unittest.TestCase):
    def setUp(self):
        print 'In setUp()'
        self.fixture = range(1, 10)

    def tearDown(self):
        print 'In tearDown()'
        del self.fixture
```

```
    def test(self):
        print 'in test()'
        self.failUnlessEqual(self.fixture, range(1, 10))
```

上面的测试运行之后，你可以看到fixtures和测试方法的执行顺序：

```
$ python unittest_fixtures.py
In setUp()
in test()
In tearDown()
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

51.10 Test Suites(测试整合)

标准库文档讲述了怎样去手工组织test suites，我一般也不直接使用test suites，因为我更喜欢自动建立suites(它们是自动生成的测试集)。 自动构建test suites对于大型工程来说尤其有用，因为相关的测试不是全部都在一个地方。 使用像nose 和 Proctor 的这些工具，对于遍布多个文件和目录的测试来说，更加容易操作。

CHAPTER

FIFTYTWO

PYMOTW: HEAPQ

- 模块： heapq
- 目的： 就地堆排序算法
- python版本: New in 2.3 with additions in 2.5 2.3+, 2.5中有所增加

heapq实现了适用于Python列表的小顶堆排序算法.

52.1 描述

堆是一种树型数据结构，其父子节点间具有顺序关系． 二进制堆可以使用一个列表或数组来表示，其中元素N的孩子所在位置为2*N+1 和 2*N+2(以0开始计算位置)． 这种特征让就地重排成为可能，这样在增加或删除元素时就不需要重新分配内存空间.

大顶堆确保每个父元素都大于或等于他的任一个孩子元素． 而小顶堆则需要每个父元素都要小于或等于他的任一个孩子元素． Python的heapq模块实现的是小顶堆.

52.2 示例数据

本文的例子中使用的是如下的示例数据:

```
data = [19, 9, 4, 10, 11, 8, 2]
```

52.3 创建一个堆

有两个基本的堆创建方式，分别是heappush()和heapify().

使用heappush()，堆中元素排序顺序是随着新元素的不断增加而不断更新的.

```
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

heap = []
print 'random :', data
print

for n in data:
    print 'add %3d:' % n
    heapq.heappush(heap, n)
    show_tree(heap)
```

```
$ python heapq_heappush.py
random : [19, 9, 4, 10, 11, 8, 2]

add  19:

                19
----------------------------------

add   9:

                9
        19
----------------------------------

add   4:

                4
        19              9
----------------------------------

add  10:
                4
        10              9
    19
----------------------------------

add  11:

                4
        10              9
    19      11
----------------------------------

add   8:

                4
        10              8
   19       11      9
----------------------------------

add   2:

                2
        10              4
    19      11      9       8
----------------------------------
```

如果数据已经在内存中了，使用heapify()进行就地排序会更有效.

```
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

print 'random :', data
heapq.heapify(data)
print 'heapified :'
show_tree(data)
```

```
$ python heapq_heapify.py
random : [19, 9, 4, 10, 11, 8, 2]
heapified :
```

```
                2
        9                   4
    10        11        8         19
------------------------------------
```

52.4 访问堆

成功建立堆之后，可以使用heappop()删除堆中最小的元素． 下面的例子改编自标准库文档中的例子，heapify()和heappop()用于对一个列表进行排序．

```
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

print 'random :', data
heapq.heapify(data)
print 'heapified :'
show_tree(data)
print

inorder = []
while data:
    smallest = heapq.heappop(data)
    print 'pop %3d:' % smallest
    show_tree(data)
    inorder.append(smallest)
print 'inorder :', inorder
```

```
$ python heapq_heappop.py
 random    : [19, 9, 4, 10, 11, 8, 2]
 heapified :

                  2
         9                   4
     10        11        8         19
 ------------------------------------


 pop      2:

                  4
         9                   8
     10        11        19
 ------------------------------------

 pop      4:

                  8
         9                   19
     10        11
 ------------------------------------

 pop      8:

                  9
         10                  19
     11
 ------------------------------------
```

```
pop      9:

                 10
        11                19
-----------------------------------

pop     10:

                 11
        19
-----------------------------------

pop     11:

                 19
-----------------------------------

pop     19:

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

inorder    : [2, 4, 8, 9, 10, 11, 19]
```

使用heapreplace()可以删除现有元素和用新的值替换已存元素.

```
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

heapq.heapify(data)
print 'start:'
show_tree(data)

for n in [0, 7, 13, 9, 5]:
    smallest = heapq.heapreplace(data, n)
    print 'replace %2d with %2d:' % (smallest, n)
    show_tree(data)
```

这个功能让你维持了一个固定大小的堆，这在具有优先级任务队列中是很用的.

```
$ python heapq_heapreplace.py
start:

                 2
        9                 4
    10       11       8        19
-----------------------------------

replace  2 with  0:

                 0
        9                 4
    10       11       8        19
-----------------------------------

replace  0 with  7:

                 4
        9                 7
    10       11       8        19
-----------------------------------
```

```
replace  4 with 13:

                7
        9                13
    10      11      13      19
------------------------------------

replace  7 with  9:

                8
        9                9
    10      11      13      19
------------------------------------

replace  8 with  5:

                5
        9                9
    10      11      13      19
------------------------------------
```

52.5 数据极值

heapq也包含了2个用于检查迭代对象中最大或最小的值范围． 使用nlargest()和nsmallest()可以获得相对最小或最大的n个数， n一般大于1， 但在有些情况下不能获得正确的值．

```
import heapq
from heapq_heapdata import data

print 'all :', data
print '3 largest :', heapq.nlargest(3, data)
print 'from sort :', list(reversed(sorted(data)[-3:]))
print '3 smallest:', heapq.nsmallest(3, data)
print 'from sort :', sorted(data)[:3]
```

```
$ python heapq_extremes.py
all : [19, 9, 4, 10, 11, 8, 2]
3 largest : [19, 11, 10]
from sort : [19, 11, 10]
3 smallest: [2, 4, 8]
from sort : [2, 4, 8]
```

52.6 参考

- heapq Theory
- WikiPedia - Heap Data Structure