

Python模块学习：random 随机数生成

原文出处：[DarkBull](#)

Python中的random模块用于生成随机数。下面介绍一下random模块中最常用的几个函数。

random.random

random.random()用于生成一个0到1的随机浮点数: $0 \leq n < 1.0$

random.uniform

random.uniform的函数原型为：random.uniform(a, b)，用于生成一个指定范围内的随机浮点数，两个参数其中一个是上限，一个是下限。如果 $a > b$ ，则生成的随机数 n : $b \leq n \leq a$ 。如果 $a < b$ ，则 $a \leq n \leq b$ 。

Python

```
print
random.uniform(10, 20)
```

1 print random.uniform(10, 20)
2 print random.uniform(20, 10)
3 #--- 结果（不同机器上的结果不一样）
4 #18.7356606526
5 #12.5798298022

random.randint

random.randint()的函数原型为：random.randint(a, b)，用于生成一个指定范围内的整数。其中参数a是下限，参数b是上限，生成的随机数 n : $a \leq n \leq b$

Python

```
print random.randint(12,
20) #生成的随机数n:
```

1 print random.randint(12, 20) #生成的随机数n: $12 \leq n \leq 20$
2 print random.randint(20, 20) #结果永远是20
3 #print random.randint(20, 10) #该语句是错误的。下限必须小于上限。

random.randrange

random.randrange的函数原型为：random.randrange([start], stop[, step])，从指定范围内，按指定基数递增的集合中 获取一个随机数。如：random.randrange(10, 100, 2)，结果相当于从[10, 12, 14, 16, ... 96, 98]序列中获取一个随机数。random.randrange(10, 100, 2)在结果上与random.choice(range(10, 100, 2)) 等效。

random.choice

random.choice从序列中获取一个随机元素。其函数原型为：random.choice(sequence)。参数sequence表示一个有序类型。这里要**说明**一下：sequence在python不是一种特定的类型，而是泛指一系列的类型。list, tuple, 字符串都属于sequence。有关sequence可以查看python手册数据模型这一章，也可以参考：<http://www.17xie.com/read-37422.html>。下面是使用choice的一些例子：

Python

```
print random.choice("学习Python")
```

```
1 print random.choice("学习Python")
2 print random.choice(["JGood", "is", "a", "handsome", "boy"])
3 print random.choice(("Tuple", "List", "Dict"))
```

random.shuffle

random.shuffle的函数原型为：random.shuffle(x[, random])，用于将一个列表中的元素打乱。如：

Python

```
p = ["Python", "is", "powerful", "simple",
```

```
1 p = ["Python", "is", "powerful", "simple", "and so on..."]
2 random.shuffle(p)
3 print p
4 #--- 结果（不同机器上的结果可能不一样。）
5 #['powerful', 'simple', 'is', 'Python', 'and so on...']
```

random.sample

random.sample的函数原型为：random.sample(sequence, k)，从指定序列中随机获取指定长度的片断。sample函数不会修改原有序列。

Python

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
1 list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 slice = random.sample(list, 5) #从list中随机获取5个元素，作为一个片断返回
3 print slice
4 print list #原有序列并没有改变。
```

上面这些方法是random模块中最常用的，在Python手册中，还介绍其他的方法。感兴趣的朋友可以通过查询Python手册了解更详细的信息。

1 赞 收藏 [评论](#)

Python模块学习：threading 多线程控制和处理

原文出处： [Darkbull](#)

[上一篇](#) 介绍了thread模块，今天来学习Python中另一个操作线程的模块：threading。threading通过对thread模块进行二次封装，提供了更方便的API来操作线程。今天内容比较多，闲话少说，现在就开始切入正题！

threading.Thread

Thread 是threading模块中最重要的类之一，可以使用它来创建线程。有两种方式来创建线程：一种是通过继承Thread类，重写它的run方法；另一种是创建一个threading.Thread对象，在它的初始化函数（__init__）中将可调用对象作为参数传入。下面分别举例说明。先来看看通过继承threading.Thread类来创建线程的例子：

Python

```
#coding=gbk
1 #coding=gbk
2 import threading, time, random
3 count = 0
4 class Counter(threading.Thread):
5     def __init__(self, lock, threadName):
6         """@summary: 初始化对象。
7
8         @param lock: 锁对象。
9         @param threadName: 线程名称。
10        """
11        super(Counter, self).__init__(name = threadName) #注意：一定要显式的调用父类的初始
12 化函数。
13        self.lock = lock
14
15    def run(self):
16        """@summary: 重写父类run方法，在线程启动后执行该方法内的代码。
17        """
18        global count
19        self.lock.acquire()
20        for i in xrange(10000):
21            count = count + 1
22        self.lock.release()
23 lock = threading.Lock()
24 for i in range(5):
25     Counter(lock, "thread-" + str(i)).start()
26 time.sleep(2) #确保线程都执行完毕
27 print count
```

在代码中，我们创建了一个Counter类，它继承了threading.Thread。初始化函数接收两个参数，一个是锁对象，另一个是线程的名称。在Counter中，重写了从父类继承的run方法，run方法将一个全局变量逐一的增加10000。在接下来的代码中，创建了五个Counter对象，分别调用其start方法。最后打印结果。这里要说明一下**run方法**和**start方法**：它们都是从Thread继承而来的，run()方法将在线程开启后执行，可以把相关的逻辑写到run方法中（通常把run方法称为活动[Activity]。）；start()方

法用于启动线程。

再看看另外一种创建线程的方法：

Python

```
import threading, time, random

1  import threading, time, random
2  count = 0
3  lock = threading.Lock()
4  def doAdd():
5      """@summary: 将全局变量count 逐一的增加10000。
6      """
7      global count, lock
8      lock.acquire()
9      for i in xrange(10000):
10         count = count + 1
11     lock.release()
12 for i in range(5):
13     threading.Thread(target = doAdd, args = (), name = 'thread-' + str(i)).start()
14 time.sleep(2) #确保线程都执行完毕
15 print count
```

在这段代码中，我们定义了方法doAdd，它将全局变量count 逐一的增加10000。然后创建了5个Thread对象，把函数对象doAdd 作为参数传给它的初始化函数，再调用Thread对象的start方法，线程启动后将执行doAdd函数。这里有必要介绍一下threading.Thread类的初始化函数原型：

```
def __init__(self, group=None, target=None, name=None, args=(), kwargs={})
```

- 参数group是预留的，用于将来扩展；
- 参数target是一个可调用对象（也称为活动[activity]），在线程启动后执行；
- 参数name是线程的名字。默认值为“Thread-N”，N是一个数字。
- 参数args和kwargs分别表示调用target时的参数列表和关键字参数。

Thread类还定义了以下常用方法与属性：

Thread.getName()

Thread.setName()

Thread.name

用于获取和设置线程的名称。

Thread.ident

获取线程的标识符。线程标识符是一个非零整数，只有在调用了start()方法之后该属性才有效，否则它只返回None。

Thread.is_alive()

Thread.isAlive()

判断线程是否是激活的（alive）。从调用start()方法启动线程，到run()方法执行完毕或遇到未处理异常而中断 这段时间内，线程是激活的。

Thread.join([timeout])

调用Thread.join将会使主调线程堵塞，直到被调用线程运行结束或超时。参数timeout是一个数值类型，表示超时时间，如果未提供该参数，那么主调线程将一直堵塞到被调线程结束。下面举个例子说明join()的使用：

Python

```
import threading, time
def doWaiting():
    1 import threading, time
    2 def doWaiting():
    3     print 'start waiting:', time.strftime('%H:%M:%S')
    4     time.sleep(3)
    5     print 'stop waiting', time.strftime('%H:%M:%S')
    6 thread1 = threading.Thread(target = doWaiting)
    7 thread1.start()
    8 time.sleep(1) #确保线程thread1已经启动
    9 print 'start join'
    10 thread1.join() #将一直堵塞，直到thread1运行结束。
    11 print 'end join'
```

threading.RLock和threading.Lock

在threading模块中，定义两种类型的锁：threading.Lock和threading.RLock。它们之间有一点细微的区别，通过比较下面两段代码来说明：

Python

```
import threading
lock = threading.Lock()

1 import threading
2 lock = threading.Lock() #Lock对象
3 lock.acquire()
4 lock.acquire() #产生了死锁。
5 lock.release()
6 lock.release()
```

Python

```
import threading
rLock =

1 import threading
2 rLock = threading.RLock() #RLock对象
3 rLock.acquire()
4 rLock.acquire() #在同一线程内，程序不会堵塞。
5 rLock.release()
6 rLock.release()
```

这两种锁的主要区别是：RLock允许在同一线程中被多次acquire。而Lock却不允许这种情况。注意：如果使用RLock，那么acquire和release必须成对出现，即调用了n次acquire，必须调用n次的release才能真正释放所占用的锁。

threading.Condition

可以把Condition理解为一把高级的锁，它提供了比Lock, RLock更高级的功能，允许我们能够控制复杂的线程同步问题。threading.Condition在内部维护一个锁对象（默认是RLock），可以在创建Condition对象的时候把锁对象作为参数传入。Condition也提供了acquire, release方法，其含义与锁的acquire, release方法一致，其实它只是简单的调用内部锁对象的对应的方法而已。Condition还提供了如下方法(特别要注意：这些方法只有在占用锁(acquire)之后才能调用，否则将会报RuntimeError异常。):

Condition.wait([timeout]):

wait方法释放内部所占用的锁，同时线程被挂起，直至接收到通知被唤醒或超时（如果提供了timeout参数的话）。当线程被唤醒并重新占有锁的时候，程序才会继续执行下去。

Condition.notify():

唤醒一个挂起的线程（如果存在挂起的线程）。注意：notify()方法不会释放所占用的锁。


Condition.notify_all()

Condition.notifyAll()

唤醒所有挂起的线程（如果存在挂起的线程）。注意：这些方法不会释放所占用的锁。

现在写个捉迷藏的游戏来具体介绍threading.Condition的基本使用。假设这个游戏由两个人来玩，一个藏(Hider)，一个找(Seeker)。游戏的规则如下：1. 游戏开始之后，Seeker先把自己眼睛蒙上，蒙上眼睛后，就通知Hider；2. Hider接收通知后开始找地方将自己藏起来，藏好之后，再通知Seeker可以找了；3. Seeker接收到通知之后，就开始找Hider。Hider和Seeker都是独立的个体，在程序中用两个独立的线程来表示，在游戏过程中，两者之间的行为有一定的时序关系，我们通过Condition来控制这种时序关系。

Python



```
#--- Condition
1 #--- Condition
2 #--- 捉迷藏的游戏
3 import threading, time
4 class Hider(threading.Thread):
5     def __init__(self, cond, name):
6         super(Hider, self).__init__()
7         self.cond = cond
8         self.name = name
9
10    def run(self):
```

```

11     time.sleep(1) #确保先运行Seeker中的方法
12
13     self.cond.acquire() #b
14     print self.name + ': 我已经把眼睛蒙上了'
15     self.cond.notify()
16     self.cond.wait() #c
17     #f
18     print self.name + ': 我找到你了 ~_~'
19     self.cond.notify()
20     self.cond.release()
21     #g
22     print self.name + ': 我赢了' #h
23
24 class Seeker(threading.Thread):
25     def __init__(self, cond, name):
26         super(Seeker, self).__init__()
27         self.cond = cond
28         self.name = name
29     def run(self):
30         self.cond.acquire()
31         self.cond.wait() #a #释放对锁的占用，同时线程挂起在这里，直到被notify并重新占
32 有锁。
33         #d
34         print self.name + ': 我已经藏好了，你快来找我吧'
35         self.cond.notify()
36         self.cond.wait() #e
37         #h
38         self.cond.release()
39         print self.name + ': 被你找到了，哎~~~'
40
41 cond = threading.Condition()
42 seeker = Seeker(cond, 'seeker')
43 hider = Hider(cond, 'hider')
44 seeker.start()
45 hider.start()

```

threading.Event

Event实现与Condition类似的功能，不过比Condition简单一点。它通过维护内部的标识符来实现线程间的同步问题。（threading.Event和.NET中的System.Threading.ManualResetEvent类实现同样的功能。）

Event.wait([timeout])

堵塞线程，直到Event对象内部标识位被设为True或超时（如果提供了参数timeout）。

Event.set()

将标识位设为True

Event.clear()


将标识位设为False。

Event.isSet()

判断标识位是否为True。

下面使用Event来实现捉迷藏的游戏(可能用Event来实现不是很形象)

Python



```
1 #--- Event
2 #--- 捉迷藏的游戏
3 import threading, time
4 class Hider(threading.Thread):
5     def __init__(self, cond, name):
6         super(Hider, self).__init__()
7         self.cond = cond
8         self.name = name
9
10    def run(self):
11        time.sleep(1) #确保先运行Seeker中的方法
12
13        print self.name + ': 我已经把眼睛蒙上了'
14
15        self.cond.set()
16
17        time.sleep(1)
18
19        self.cond.wait()
20        print self.name + ': 我找到你了 ~_~'
21
22        self.cond.set()
23
24        print self.name + ': 我赢了'
25
26 class Seeker(threading.Thread):
27     def __init__(self, cond, name):
28         super(Seeker, self).__init__()
29         self.cond = cond
30         self.name = name
31     def run(self):
32         self.cond.wait()
33
34         print self.name + ': 我已经藏好了，你快来找我吧'
35         self.cond.set()
36
37         time.sleep(1)
38         self.cond.wait()
39
40         print self.name + ': 被你找到了，哎~~~'
41
42 cond = threading.Event()
43 seeker = Seeker(cond, 'seeker')
44 hider = Hider(cond, 'hider')
45 seeker.start()
46 hider.start()
```

threading.Timer

threading.Timer是threading.Thread的子类，可以在指定时间间隔后执行某个操作。下面是Python手册上提供的一个例子：

Python

```
def hello():  
    print "hello, world"
```

```
1 def hello():  
2     print "hello, world"  
3 t = Timer(3, hello)  
4 t.start() # 3秒钟之后执行hello函数。
```

threading模块中还有一些常用的方法没有介绍：

threading.active_count()

threading.activeCount()

获取当前活动的(alive)线程的个数。

threading.current_thread()

threading.currentThread()

获取当前的线程对象（Thread object）。

threading.enumerate()

获取当前所有活动线程的列表。

threading.settrace(func)

设置一个跟踪函数，用于在run()执行之前被调用。

threading.setprofile(func)

设置一个跟踪函数，用于在run()执行完毕之后调用。

threading模块的内容很多，一篇文章很难写全，更多关于threading模块的信息，请查询Python手册 [threading](#)模块。

1 赞 6 收藏 [1 评论](#)

Python模块学习：thread 多线程处理

原文出处：[DarkBull](#)

这段时间一直在用 Python 写一个游戏的服务器程序。在编写过程中，不可避免的要多线程来处理与客户端的交互。Python 标准库提供了 thread 和 threading 两个模块来对多线程进行支持。其中，thread 模块以低级、原始的方式来处理和控制线程，而 threading 模块通过对 thread 进行二次封装，提供了更方便的 api 来处理线程。虽然使用 thread 没有 threading 来的方便，但它更灵活。今天先介绍 thread 模块的基本使用，[下一篇](#) 将介绍 threading 模块。

在介绍 thread 之前，先看一段代码，猜猜程序运行完成之后，在控制台上输出的结果是什么？

Python

```
#coding=gbk
import thread, time, random

1 #coding=gbk
2 import thread, time, random
3 count = 0
4 def threadTest():
5     global count
6     for i in xrange(10000):
7         count += 1
8 for i in range(10):
9     thread.start_new_thread(threadTest, ()) #如果对start_new_thread函数不是很了解，不要着急，马上就会讲解
10 time.sleep(3)
11 print count #count是多少呢？是10000 * 10 吗？
```

thread.start_new_thread (*function* , *args* [, *kwargs*])

函数将创建一个新的线程，并返回该线程的标识符（标识符为整数）。参数 function 表示线程创建之后，立即执行的函数，参数 args 是该函数的参数，它是一个元组类型；第二个参数 kwargs 是可选的，它为函数提供了命名参数字典。函数执行完毕之后，线程将自动退出。如果函数在执行过程中遇到未处理的异常，该线程将退出，但不会影响其他线程的执行。下面是一个简单的例子：

Python

```
#coding=gbk
import thread, time

1 #coding=gbk
2 import thread, time
3 def threadFunc(a = None, b = None, c = None, d = None):
4     print time.strftime('%H:%M:%S', time.localtime()), a
5     time.sleep(1)
6     print time.strftime('%H:%M:%S', time.localtime()), b
7     time.sleep(1)
8     print time.strftime('%H:%M:%S', time.localtime()), c
9     time.sleep(1)
10    print time.strftime('%H:%M:%S', time.localtime()), d
11    time.sleep(1)
12    print time.strftime('%H:%M:%S', time.localtime()), 'over'
13
```

```
14 thread.start_new_thread(threadFunc, (3, 4, 5, 6)) #创建线程，并执行threadFunc函数。
15 time.sleep(5)
```

thread.exit ()

结束当前线程。调用该函数会触发 SystemExit 异常，如果没有处理该异常，线程将结束。

thread.get_ident ()

返回当前线程的标识符，标识符是一个非零整数。

thread.interrupt_main ()

在主线程中触发 [KeyboardInterrupt](#) 异常。子线程可以使用该方法来中断主线程。下面的例子演示了在子线程中调用 interrupt_main ，在主线程中捕获异常：

Python

```
import thread, time
thread.start_new_thread(
1 import thread, time
2 thread.start_new_thread(lambda : (thread.interrupt_main(), ), ())
3 try:
4     time.sleep(2)
5 except KeyboardInterrupt, e:
6     print 'error:', e
7 print 'over'
```

下面介绍 thread 模块中的锁，锁可以保证在任何时刻，最多只有一个线程可以访问共享资源。

thread.LockType 是 thread 模块中定义的锁类型。它有如下方法：

lock.acquire ([waitflag])

获取锁。函数返回一个布尔值，如果获取成功，返回 True ，否则返回 False 。参数 *waitflag* 的默认值是一个非零整数，表示如果锁已经被其他线程占用，那么当前线程将一直等待，只到其他线程释放，然后获取访锁。如果将参数 *waitflag* 置为 0 ，那么当前线程会尝试获取锁，不管锁是否被其他线程占用，当前线程都不会等待。

lock.release ()

释放所占用的锁。

lock.locked ()

判断锁是否被占用。

现在我们回过头来看文章开始处给出的那段代码：代码中定义了一个函数 threadTest ，它将全局变量逐一的增加 10000 ，然后在主线程中开启了 10 个子线程来调用 threadTest 函数。但结果并不是预料中的 $10000 * 10$ ，原因主要是对 count 的并发操作引来的。全局变量 count 是共享资源，对它的操作应该串行的进行。下面对那段代码进行修改，在对 count 操作的时候，进行加锁处理。看看程序运行的结果是否和预期一致。修改后的代码：

Python

```
#coding=gbk
import thread, time, random

1 #coding=gbk
2 import thread, time, random
3 count = 0
4 lock = thread.allocate_lock() #创建一个锁对象
5 def threadTest():
6     global count, lock
7     lock.acquire() #获取锁
8
9     for i in xrange(10000):
10         count += 1
11
12     lock.release() #释放锁
13 for i in xrange(10):
14     thread.start_new_thread(threadTest, ())
15 time.sleep(3)
16 print count
```

thread模块是不是并没有想像中的那么难！简单就是美，这就是Python。更多关于thread模块的内容，请参考Python手册 [thread](#) 模块

1 赞 3 收藏 [评论](#)

Python模块学习：httplib HTTP协议客户端实现

原文出处：[DarkBull](#)

httplib 是 python中http 协议的客户端实现，可以使用该模块来与 HTTP 服务器进行交互。httplib的内容不是很多，也比较简单。以下是一个非常简单的例子，使用httplib获取google首页的html：

Python

```
#coding=gbk
import httplib

1 #coding=gbk
2 import httplib
3 conn = httplib.HTTPConnection("www.google.cn")
4 conn.request('get', '/')
5 print conn.getresponse().read()
6 conn.close()
```

下面详细介绍httplib提供的常用类型和方法。

httplib.HTTPConnection (host [, port [, strict [, timeout]]])

HTTPConnection类的构造函数，表示一次与服务器之间的交互，即请求/响应。参数host表示服务器主机，如：www.csdn.net；port为端口号，默认值为80；参数strict的默认值为false，表示在无法解析服务器返回的状态行时(status line)（比较典型的状态行如： HTTP/1.0 200 OK ），是否抛BadStatusLine 异常；可选参数timeout 表示超时时间。
HTTPConnection提供的方法：

HTTPConnection.request (method , url [, body [, headers]])

调用request 方法会向服务器发送一次请求，method 表示请求的方法，常用有方法有get 和post ； url 表示请求的资源 url ； body 表示提交到服务器的数据，必须是字符串（如果method 是"post" ，则可以把body 理解为html 表单中的数据）； headers 表示请求的http 头。

HTTPConnection.getresponse ()

获取Http 响应。返回的对象是HTTPResponse 的实例，关于HTTPResponse 在[下面](#) 会讲解。

HTTPConnection.connect ()

连接到Http 服务器。

HTTPConnection.close ()

关闭与服务器的连接。

HTTPConnection.set_debuglevel (level)

设置高度的级别。参数level 的默认值为0 ，表示不输出任何调试信息。

httplib.HTTPResponse

HTTPResponse表示服务器对客户端请求的响应。往往通过调用HTTPConnection.getresponse()来创建，它有如下方法和属性：

HTTPResponse.read([amt])

获取响应的消息体。如果请求的是一个普通的网页，那么该方法返回的是页面的html。可选参数amt表示从响应流中读取指定字节的数据。

HTTPResponse.getheader(name[, default])

获取响应头。Name表示头域(header field)名，可选参数default在头域名不存在的情况下作为默认值返回。

HTTPResponse.getheaders()

以列表的形式返回所有的头信息。

HTTPResponse.msg

获取所有的响应头信息。

HTTPResponse.version

获取服务器所使用的http协议版本。11表示http/1.1；10表示http/1.0。

HTTPResponse.status


获取响应的状态码。如：200表示请求成功。

HTTPResponse.reason

返回服务器处理请求的结果说明。一般为“OK”

下面通过一个例子来熟悉HTTPResponse中的方法：

Python



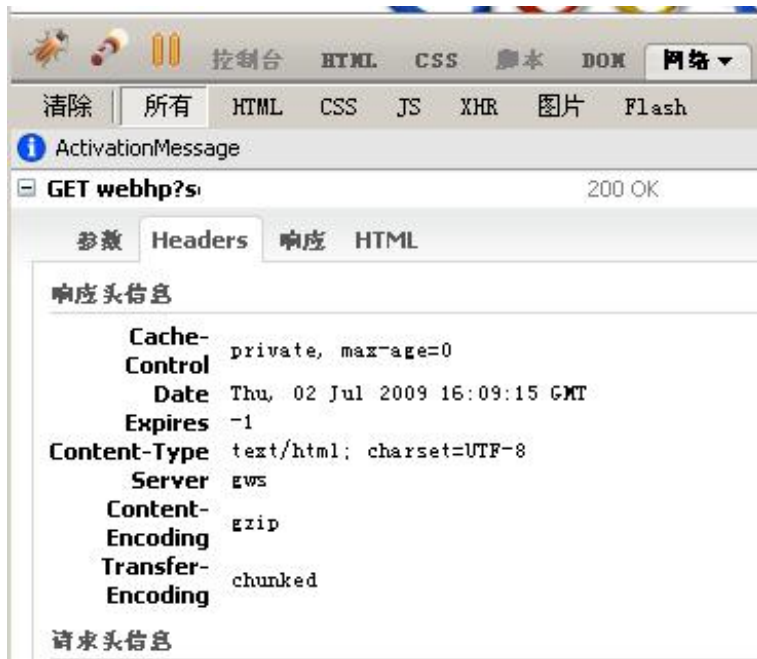
```
1 #coding=gbk
2 import httplib
3 conn = httplib.HTTPConnection("www.g.cn", 80, False)
4 conn.request('get', '/', headers = {"Host": "www.google.cn",
5                                     "User-Agent": "Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.9.1) Gecko/20090624"
```

```

6 Firefox/3.5",
7             "Accept": "text/plain"})
8 res = conn.getresponse()
9 print 'version:', res.version
10 print 'reason:', res.reason
11 print 'status:', res.status
12 print 'msg:', res.msg
13 print 'headers:', res.getheaders()
14 #html
15 #print '\n' + '-' * 50 + '\n'
16 #print res.read()
    conn.close()

```

这是我用firebug跟踪到响应头：



Httpplib模块中还定义了许多常量，如：

Httpplib.HTTP_PORT 的值为80，表示默认的端口号为80；

Httpplib.OK 的值为200，表示请求成功返回；

Httpplib.NOT_FOUND 的值为404，表示请求的资源不存在；

可以通过**httplib.responses** 查询相关变量的含义，如：

```
Print httplib.responses[httplib.NOT_FOUND]  #not found
```


1 赞 3 收藏 [1 评论](#)

python模块学习：smtpplib 邮件发送

原文出处： [DarkBull](#)

在基于互联网的应用中，程序经常需要自动地发送电子邮件。如：一个网站的注册系统会在用户注册时发送一封邮件来确认注册；当用户忘记登陆密码的时候，通过邮件来取回密码。smtpplib模块是python中smtp(简单邮件传输协议)的客户端实现。我们可以使用smtpplib模块，轻松的发送电子邮件。下面的例子用了不到十行代码来发送电子邮件：

Python



```
1 #coding=gbk
2 import smtpplib
3
4 smtp = smtpplib.SMTP()
5 smtp.connect("smtp.yeah.net", "25")
6 smtp.login('用户名', '密码')
7 smtp.sendmail('from@yeah.net', 'to@21cn.com', 'From: from@yeah.net/r/nTo: to@21cn.com/r/nSubject: this is a email
8 from python demo/r/n/r/nJust for test~_~')
9 smtp.quit()
```

这个例子够简单吧^_^！下面详细介绍smtpplib模块中的类和方法。

smtpplib.SMTP([host[, port[, local_hostname[, timeout]]]])

SMTP类构造函数，表示与SMTP服务器之间的连接，通过这个连接我们可以向smtp服务器发送指令，执行相关操作（如：登陆、发送邮件）。该类提供了许多方法，将在下面介绍。它的所有参数都是可选的，其中host参数表示smtp服务器主机名，上面例子中的smtp主机为”smtp.yeah.net”；port表示smtp服务的端口，默认是25；如果在创建SMTP对象的时候提供了这两个参数，在初始化的时候会自动调用connect方法去连接服务器。

smtpplib模块还提供了SMTP_SSL类和LMTP类，对它们的操作与SMTP基本一致。

smtpplib.SMTP提供的方法：

SMTP.set_debuglevel(level)

设置是否为调试模式。默认为False，即非调试模式，表示不输出任何调试信息。

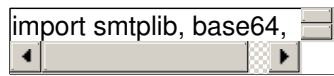
SMTP.connect([host[, port]])

连接到指定的smtp服务器。参数分别表示smtp主机和端口。注意：也可以在host参数中指定端口号（如：smtp.yeah.net:25），这样就没必要给出port参数。

SMTP.docmd(cmd[, argstring])

向smtp服务器发送指令。可选参数argstring表示指令的参数。下面的例子完全通过调用docmd方法向服务器发送指令来实现邮件的发送（在smtp.yeah.net邮件服务器上试验通过。其他邮件服务器没有试过）：

Python



```
1 import smtplib, base64, time
2 userName = base64.encodestring('from').strip()
3 password = base64.encodestring('password').strip()
4 smtp = smtplib.SMTP()
5 smtp.connect("smtp.yeah.net:25")
6 print smtp.docmd('helo', 'from')
7 print smtp.docmd('auth login')
8 print smtp.docmd(userName)
9 print smtp.docmd(password)
10 print smtp.docmd('mail from:', '<from@yeah.net>')
11 print smtp.docmd('rcpt to:', '<from@yeah.net>')
12 #data 指令表示邮件内容
13 print smtp.docmd('data')
14 print smtp.docmd("""from: from@yeah.net
15 to: from@yeah.net
16 subject: subject
17 email body
18 .
19 """)
20 smtp.quit()
```

SMTP.helo([hostname])

使用“helo”指令向服务器确认身份。相当于告诉smtp服务器“我是谁”。

SMTP.has_extn(name)

判断指定名称在服务器邮件列表中是否存在。出于安全考虑，smtp服务器往往屏蔽了该指令。

SMTP.verify(address)

判断指定邮件地址是否在服务器中存在。出于安全考虑，smtp服务器往往屏蔽了该指令。

SMTP.login(user, password)

登陆到smtp服务器。现在几乎所有的smtp服务器，都必须在验证用户信息合法之后才允许发送邮件。

SMTP.sendmail(from_addr, to_addrs, msg[, mail_options, rcpt_options])

发送邮件。这里要注意一下第三个参数，msg是字符串，表示邮件。我们知道邮件一般由标题，发信人，收件人，邮件内容，附件等构成，发送邮件的时候，要注意msg的格式。这个格式就是smtp协议中定义的格式。在上面的例子中，msg的值为：

Python

A screenshot of a code editor window showing a single line of Python code: `'''From: from@yeah.net'''`. The editor has a light gray background and a small scrollbar on the right.

```
1 '''From: from@yeah.net
2 To: to@21cn.com
3 Subject: test
4 just for test'''
5
```

这个字符串的意思表示邮件发件人为“from@yeah.net”，收件人为“to@21cn.com”，邮件标题为“test”，邮件内容为“just for test”。细心的你可能会疑问：如果要发送的邮件内容很复杂，包含图片、视频、附件等内容，按照MIME的格式来拼接字符串，将是一件非常麻烦的事。不用担心，python已经考虑到了这点，它为我们提供了email模块，使用该模块可以轻松的发送带图片、视频、附件等复杂内容的邮件。在介绍完smtpplib模块之后，我会简单介绍email模块的基本使用。

SMTP.quit()

断开与smtp服务器的连接，相当于发送“quit”指令。

email及其相关子模块

emial模块用来处理邮件消息，包括MIME和其他基于[RFC 2822](#)的消息文档。使用这些模块来定义邮件的内容，是非常简单的。下面是一些常用的类：

class email.mime.multipart. MIMEMultipart: 多个MIME对象的集合。

class email.mime.audio. MIMEAudio: MIME音频对象。

class email.mime.image. MIMEImage: MIME二进制文件对象。

class email.mime.text. MIMEText: MIME文本对象。

看上面的解释可能会觉得云里雾里，其实我对smtp, MIME的理解也很肤浅。但在大多数时候，我们只要会用就可以了。下面是一个简单的例子来演示如何使用这些类来发送带附件的邮件：

Python

A screenshot of a code editor window showing the first line of a Python script: `#coding=gbk`. The editor has a light gray background and a small scrollbar on the right.

```
1 #coding=gbk
2 import smtplib, mimetypes
3 from email.mime.text import MIMEText
4 from email.mime.multipart import MIMEMultipart
5 from email.mime.image import MIMEImage
6
7 msg = MIMEMultipart()
8 msg['From'] = "from@yeah.net"
9 msg['To'] = 'to@21cn.com'
10 msg['Subject'] = 'email for tesing'
11
12 #添加邮件内容
```

```
13 txt = MIMEText("这是邮件内容~~")
14 msg.attach(txt)
15
16 #添加二进制附件
17 fileName = r'e:/PyQt4.rar'
18 ctype, encoding = mimetypes.guess_type(fileName)
19 if ctype is None or encoding is not None:
20     ctype = 'application/octet-stream'
21 maintype, subtype = ctype.split('/', 1)
22 att1 = MIMEImage((lambda f: (f.read(), f.close()))(open(fileName, 'rb'))[0], _subtype = subtype)
23 att1.add_header('Content-Disposition', 'attachment', filename = fileName)
24 msg.attach(att1)
25
26 #发送邮件
27 smtp = smtplib.SMTP()
28 smtp.connect('smtp.yeah.net:25')
29 smtp.login('from', '密码')
30 smtp.sendmail('from@yeah.net', 'to@21cn.com', msg.as_string())
31 smtp.quit()
32 print '邮件发送成功'
```

是不是很简单。简单就是美，用最少的代码把问题解决，这就是Python。更多关于smtplib的信息，请参考Python手册 [smtplib](#)模块。

1 赞 1 收藏 [1 评论](#)

一起写一个 Web 服务器 (1)

本文由 [伯乐在线](#) - [高世界](#) 翻译, [艾凌风](#) 校稿。未经许可, 禁止转载!

英文出处: [ruslanspivak](#)。欢迎加入[翻译组](#)。

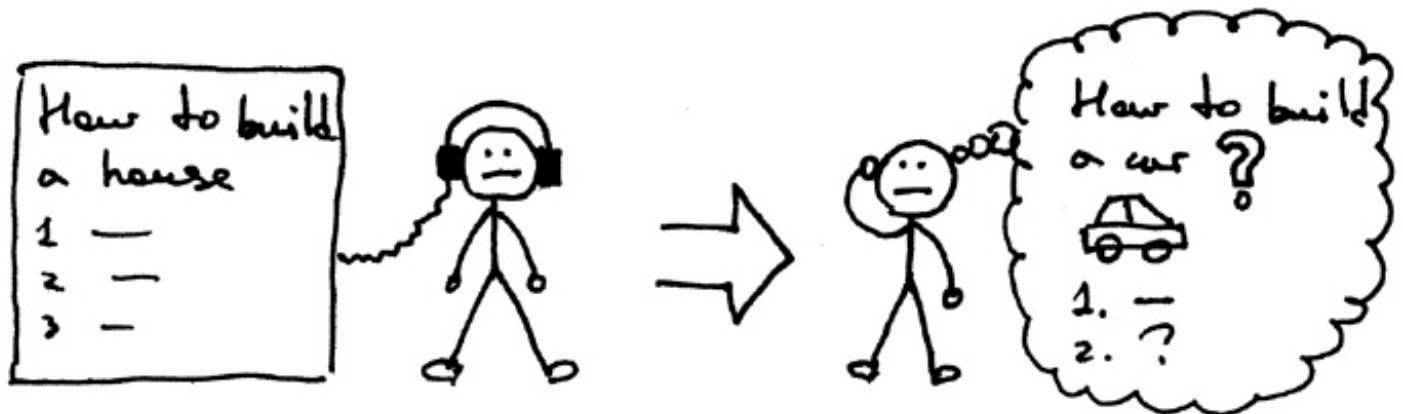
有天一个女士出门散步, 路过一个建筑工地, 看到三个男人在干活。她问第一个男人, “你在干什么呢?”, 第一个男人被问得很烦, 咆哮道, “你没看到我在码砖吗? ”。她对回答不满意, 然后问第二个男人他在干什么。第二个男人回答, “我正在砌墙”, 然后转移注意力到第一个男人, 他说, “嘿, 你码过头了, 你要把最后一块砖拿掉。”。她还是对回答不满意, 然后问第三个男人在干什么。第三个男人仰望着天空对她说, “我正在建造世界上最大的教堂。”。当他站在那里仰望天空的时候, 另外两个男人开始争论砖位置不对的问题。第三个男人转向前两个男人说, “嘿, 伙计们, 别担心那块砖了, 那是里面的墙, 它会被灰泥堵塞起来, 然后没人会看到那块砖。去另一层干活吧。”

故事的寓意是说, 当你了解整个系统, 理解不同的部分如何组织到一起的 (砖、墙、教堂), 你就能找出问题并快速解决之 (砖位置不对) 。

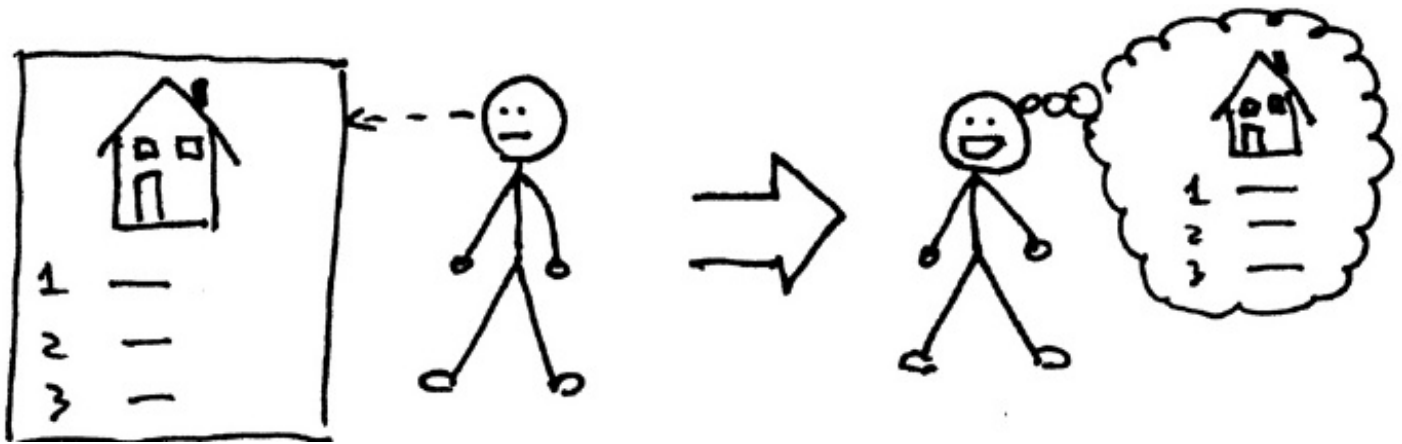
这跟从零开始搭建你的WEB服务器有什么关系呢?

我相信, 要成为优秀的开发者, 你必须对你每天都用的底层的软件系统有进一步的理解, 包括编程语言、编译器和解释器、数据库和操作系统、WEB服务器和WEB框架。为了更好更深入的理解这些系统, 你可以从零开始一块砖地, 一面墙地, 重建它们。

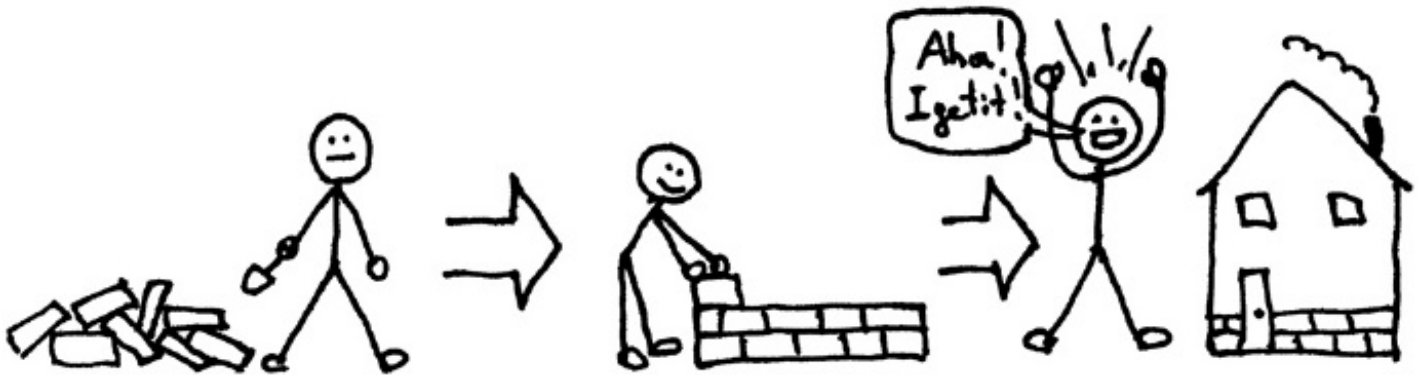
子曰: 闻之我也野, 视之我也饶, 行之我也明



“我看过的, 我还记得。”



“我做过的，我都理解了。”

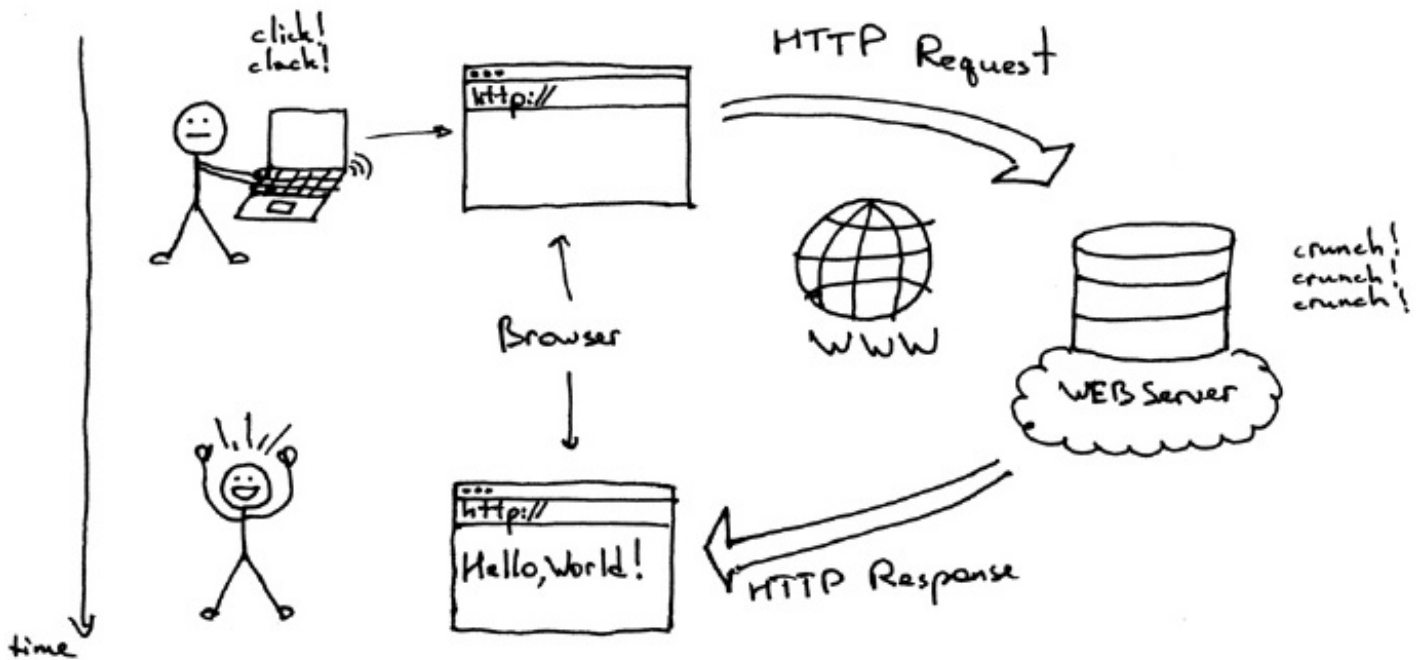


（子曰：闻之我也野，视之我也饶，行之我也明）

此时我希望你能够相信，从重建不同的软件系统来开始来学习它们是如何工作的，是一个好主意。

在这个由3部分组成的系列文章中，我会向你展示怎样搭建一个基本的WEB服务器。咱们开始吧。

重中之重，什么是WEB服务器？



简而言之，它是一个位于一个物理服务器上的网络服务器（呀，服务器上的服务器），它等待客户端发送请求。当它接收到一个请求，就会生成一个响应并回发给客户端。客户端和服务端使用HTTP协议通信。客户端可以是浏览器或者别的使用HTTP协议的软件。

一个非常简单的WEB服务器实现长什么样呢？以下是我写的一个。例子是用Python语言写的，但是即使你不会Python（它是一个非常易学的语言，试试！），你仍然可以通过代码和下面的解释理解相关概念：

Python

```
import socket

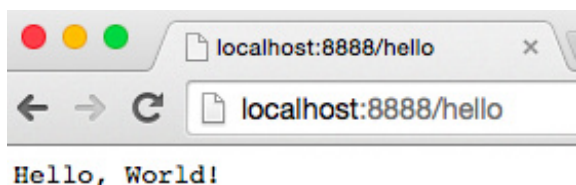
1  import socket
2
3  HOST, PORT = "", 8888
4
5  listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6  listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
7  listen_socket.bind((HOST, PORT))
8  listen_socket.listen(1)
9  print 'Serving HTTP on port %s ...' % PORT
10 while True:
11     client_connection, client_address = listen_socket.accept()
12     request = client_connection.recv(1024)
13     print request
14
15     http_response = """
16 HTTP/1.1 200 OK
17
18 Hello, World!
19 """
20     client_connection.sendall(http_response)
21     client_connection.close()
```

把上面的代码保存到webserver1.py或者直接从[GitHub](https://github.com)下载，然后像下面这样在命令行运行它

Python

```
$ python webserver1.py
Serving HTTP on port
1 $ python webserver1.py
2 Serving HTTP on port 8888 ...
```

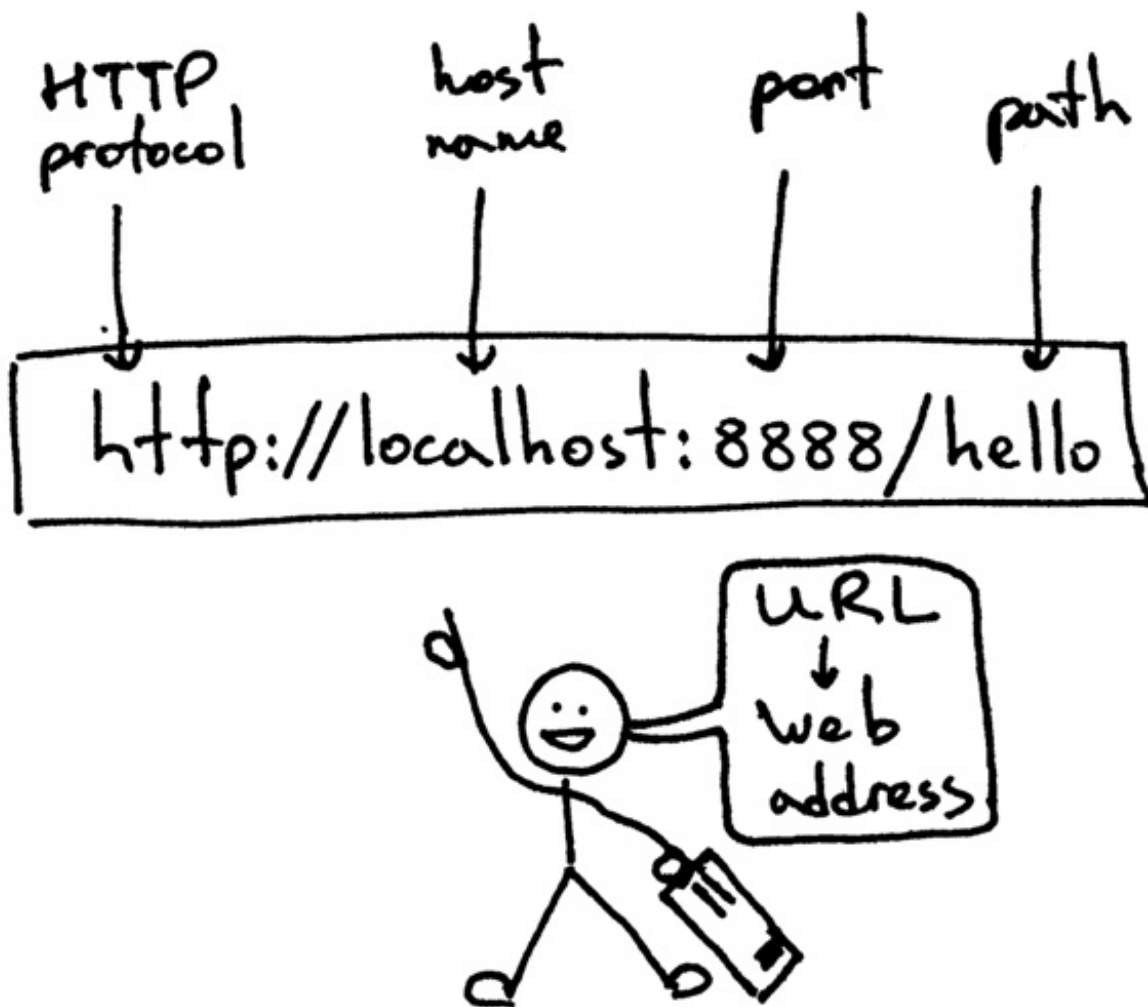
现在在你的WEB浏览器地址栏里输入以下URL `http://localhost:8888/hello`，敲回车，见证奇迹的时刻。你会看到浏览器显示“Hello, World!”，像这样：



认真做一下吧，我会等你的。

做完了？很好。现在我们讨论一下它到底怎么工作的。

首先我们从你刚才键入的WEB地址开始。它叫URL，这是它的基本结构：



这个就表示怎样告诉浏览器要查找和连接的WEB服务器地址，和你要获取的服务器上的页面（路径）。但是在浏览器发送HTTP请求前，浏览器需要先和WEB服务器建立TCP连接。然后浏览器在TCP连接上发送HTTP请求，然后等待服务器回发HTTP响应。当浏览器接收到响应后，显示响应，在本次例子中，浏览器显示“Hello, World!”。

我们再详细探索一下客户端和服务端在发送HTTP请求和响应前如何建立TCP连接的。在建立连接，它们必须使用所谓的sockets。用你命令行下的telnet手动模拟浏览器吧，而不是直接使用浏览器。

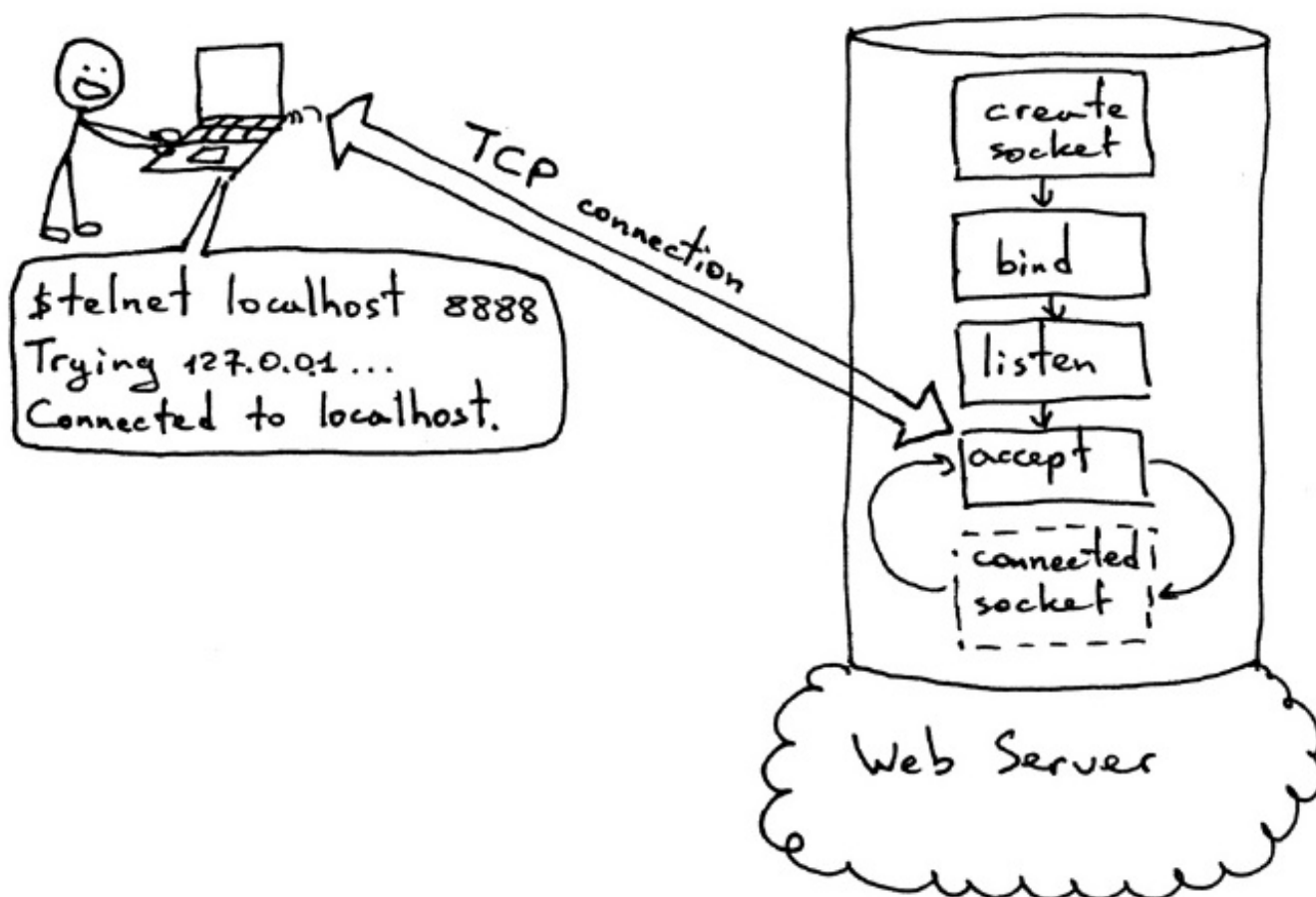
在运行WEB服务器的同一台电脑上，在命令行启动一个telnet会话，指定连接到localhost主机，连接端口为8888，然后按回车：

Python

```
$ telnet localhost 8888
Trying 127.0.0.1 ...
```

```
1 $ telnet localhost 8888
2 Trying 127.0.0.1 ...
3 Connected to localhost.
```

此时，你已经和运行在你本地主机的服务器建立了TCP连接，已经准备好发送并接收HTTP消息了。下图中你可以看到一个服务器要经过的标准步骤，然后才能接受新的TCP连接。

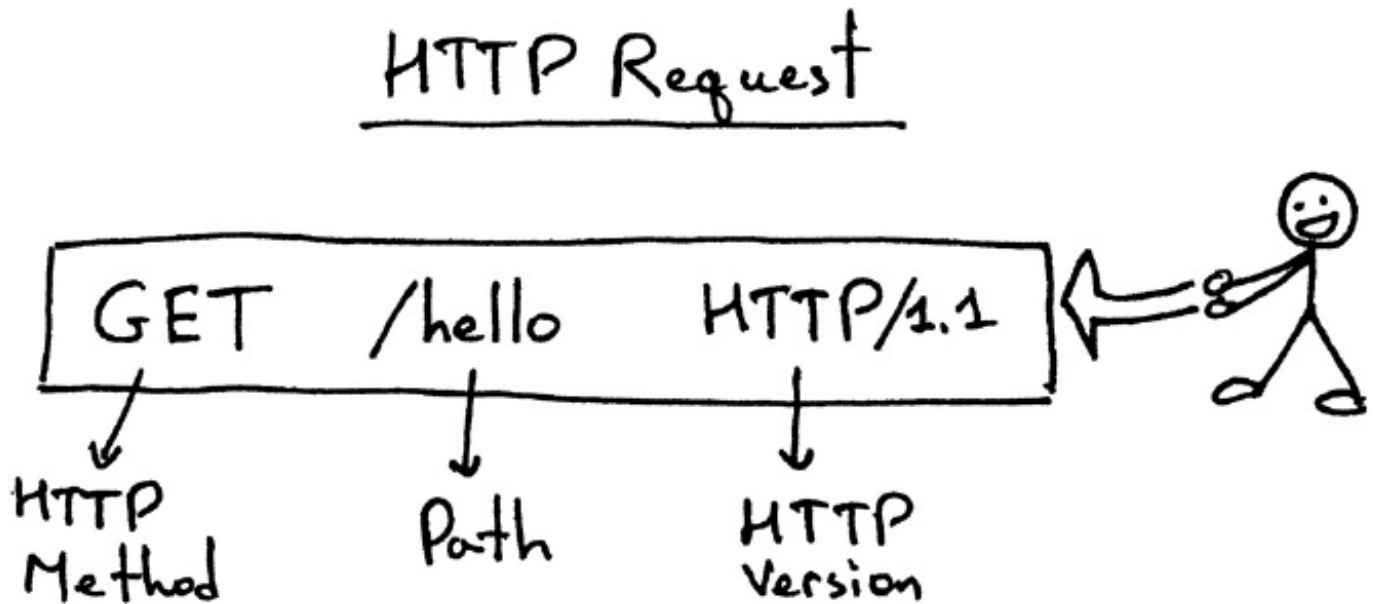


在同一个telnet会话中，输入 GET /hello HTTP/1.1然后敲回车：

Python


```
$ telnet localhost 8888
Trying 127.0.0.1 ...
1 $ telnet localhost 8888
2 Trying 127.0.0.1 ...
3 Connected to localhost.
4 GET /hello HTTP/1.1
5
6 HTTP/1.1 200 OK
7 Hello, World!
```

你完成了手动模拟浏览器！你发送了一个HTTP请求并得到了一个HTTP响应。这是HTTP请求的基本结构：



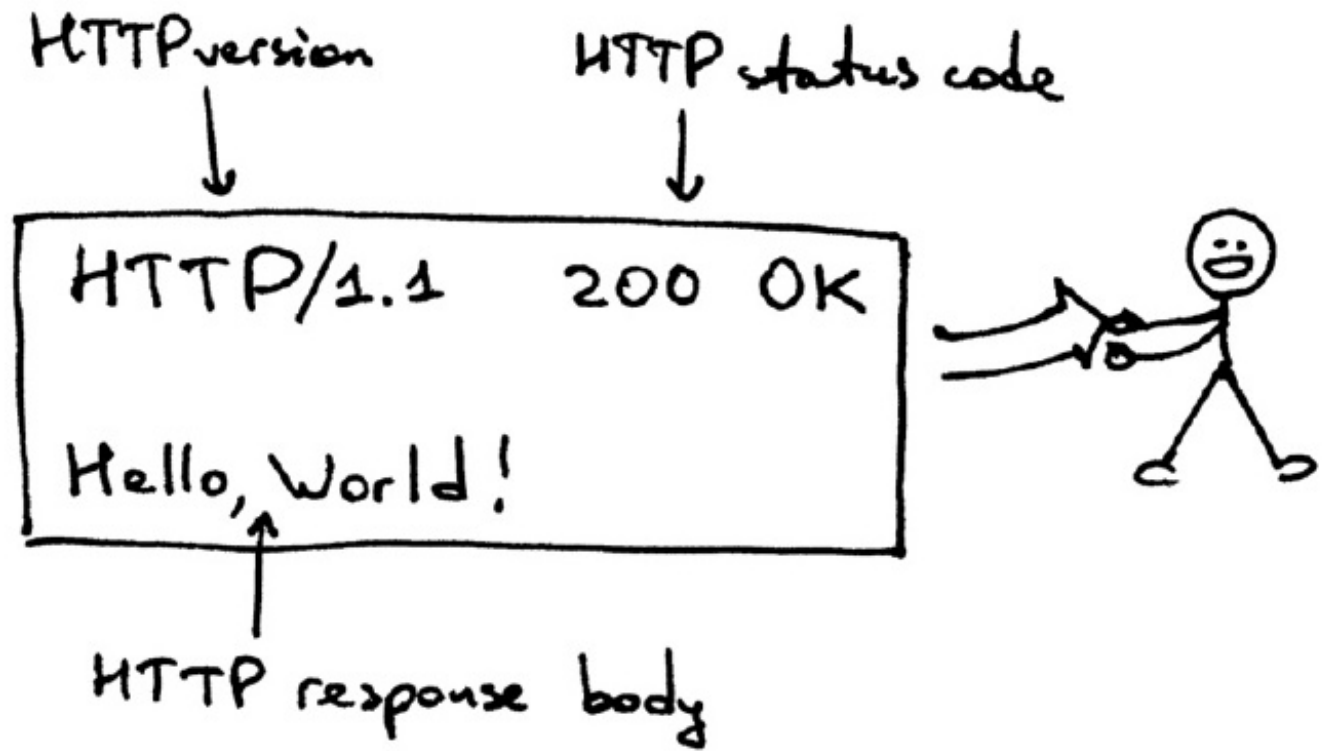
HTTP请求由行组成。行指示了HTTP方法（GET，因为我们请求我们的服务器返回给我们一些东西）、代表我们想要的服务器上的“页面”的路径 /hello和协议版本。

为了简单起见，此时我们的WEB服务器完全忽略了上面的请求行。你也可以输入任何垃圾字符取代“GET /hello HTTP/1.1”，你仍然会得到“Hello, World!”响应。

一旦你输入了请求行，敲了回车，客户端就发送请求给服务器，服务器读取请求行，打印出来然后返回相应的HTTP响应。

以下是服务器回发给客户端（这个例子中是telnet）的HTTP响应：

HTTP Response



咱们分析一下它，响应包含了状态行HTTP/1.1 200 OK，随后一个必须的空行，和HTTP响应body。

响应状态行HTTP/1.1 200 OK包含了HTTP版本，HTTP状态码和HTTP状态码理由短语OK。浏览器得到响应时，它就显示响应的body，所以你就看到了“Hello, World!”

这就是WEB浏览器怎么工作的基本模型。总结来说：WEB服务器创建一个监听socket然后开始循环接受新连接。客户端初始化一个TCP连接，在连接成功后，客户端发送HTTP请求到服务器，服务器响应一个显示给用户的HTTP响应。客户端和服务器都使用socket建立TCP连接。

你现在你拥有了一个非常基础的WEB服务器，你可以用浏览器或其他HTTP客户端测试它。正如你看到的，使用telnet手动输入HTTP请求，你也就成了一个人肉 HTTP 客户端。

对你来说有一个问题：“怎样在你的刚完成的WEB服务器下运行 Django 应用、Flask 应用和 Pyramid 应用？在不单独修改服务器来适应这些不同的 WEB 框架的情况下。”

我会在本系列的第 2 部分秀给你看的。请保持关注哦。

顺便说下，我在写一本书《**一起构建WEB服务器：第一步**》，它解释了从零开始写一个基本的WEB服务器，还更详细地讲解了我上面提到的话题。订阅邮件组来获取关于书籍和发布时间和最近更新。

灵感来自于 [Lead with a Story: A Guide to Crafting Business Narratives That Captivate, Convince, and Inspire](#)

6 赞 38 收藏 [15 评论](#)

关于作者： [高世界](#)



我翻译得越多，发现知道的越少，我就要更多地翻译。论得的地正确用法。我是php开发者，对python，c/c++，linux感兴趣。 [个人主页](#) · [我的文章](#) · [17](#) ·

Python模块学习：logging 日志记录

原文出处：[DarkBull](#)

许多应用程序中都会有日志模块，用于记录系统在运行过程中的一些关键信息，以便于对系统的运行状况进行跟踪。在.NET平台中，有非常著名的第三方开源日志组件log4net，c++中，有人们熟悉的log4cpp，而在python中，我们不需要第三方的日志组件，因为它已经为我们提供了简单易用、且功能强大的日志模块：logging。logging模块支持将日志信息保存到不同的目标域中，如：保存到日志文件中；以邮件的形式发送日志信息；以http get或post的方式提交日志到web服务器；以windows事件的形式记录等等。这些日志保存方式可以组合使用，每种方式可以设置自己的日志级别以及日志格式。日志模块的内容比较多，今天先学习logging模块的基本使用，下次具体学习日志的处理。

先看一个比较简单的例子，让我们对logging模块有个感性的认识：

Python

```
import logging
logging.basicConfig(file

1 import logging
2 logging.basicConfig(filename = os.path.join(os.getcwd(), 'log.txt'), level = logging.DEBUG)
3 logging.debug('this is a message')
```

运行上面例子的代码，将会在程序的根目录下创建一个log.txt文件，打开该文件，里面有一条日志记录：“DEBUG:root:this is a message”。

4个主要的组件

logger: 日志类，应用程序往往通过调用它提供的api来记录日志；

handler: 对日志信息处理，可以将日志发送(保存)到不同的目标域中；

filter: 对日志信息进行过滤；

formatter: 日志的格式化；

日志级别

在记录日志时，日志消息都会关联一个级别(“级别”本质上是一个非负整数)。系统默认提供了6个级别，它们分别是：

级别	对应的值
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10

级别	对应的值
----	------

可以给日志对象(Logger Instance)设置日志级别，低于该级别的日志消息将会被忽略，也可以给Handler设置日志级别，对于低于该级别的日志消息，Handler也会忽略。

logging模块中的常用函数：

logging.basicConfig(**kwargs):

为日志模块配置基本信息。kwargs 支持如下几个关键字参数：

filename：日志文件的保存路径。如果配置了些参数，将自动创建一个FileHandler作为Handler；
filemode：日志文件的打开模式。默认值为'a'，表示日志消息以追加的形式添加到日志文件中。如果设为'w'，那么每次程序启动的时候都会创建一个新的日志文件；
format：设置日志输出格式；
datefmt：定义日期格式；
level：设置日志的级别.对低于该级别的日志消息将被忽略；
stream：设置特定的流用于初始化StreamHandler；

下面是一个简单的例子：

Python

```
#coding=gbk
import logging

1 #coding=gbk
2 import logging
3 logging.basicConfig(filename = os.path.join(os.getcwd(), 'log.txt'), /
4     level = logging.WARN, filemode = 'w', format = '%(asctime)s - %(levelname)s: %(message)s')
5 logging.debug('debug') #被忽略
6 logging.info('info') #被忽略
7 logging.warning('warn')
8 logging.error('error')
9 #----- 结果
10 #2009-07-13 21:42:15,592 - WARNING: warn
11 #2009-07-13 21:42:15,640 - ERROR: error
```

logging.getLogger([name])

创建Logger对象。日志记录的工作主要由Logger对象来完成。在调用getLogger时要提供Logger的名称（注：多次使用相同名称来调用getLogger，返回的是同一个对象的引用。），Logger实例之间有层次关系，这些关系通过Logger名称来体现，如：

```
p = logging.getLogger("root")
```

```
c1 = logging.getLogger("root.c1")
```

```
c2 = logging.getLogger("root.c2")
```

例子中，p是父logger，c1,c2分别是p的子logger。c1, c2将继承p的设置。如果省略了name参数，getLogger将返回日志对象层次关系中的根Logger。

logging.setLoggerClass(klass)

logging.getLoggerClass()

获取/设置日志类型。用户可以自定义日志类来代替系统提供的logging.Logger类。

logging.getLevelName(lvl)

获取日志级别对应的名称。例如：

Python

```
print
logging.getLevelName(lvl)

1 print logging.getLevelName(logging.NOTSET)
2 print logging.getLevelName(10) #logging.DEBUG
3 print logging.getLevelName(logging.DEBUG)
4 print logging.getLevelName(30) #logging.WARN
5 print logging.getLevelName(logging.ERROR)
6 print logging.getLevelName(50) #logging.CRITICAL
```

logging.shutdown()

当不再使用日志系统的时候，调用该方法，它会将日志flush到对应的目标域上。一般在系统退出的时候调用。

Logger对象 通过调用logging.getLogger(name)来创建，它有如下常用的方法和属性：

Logger.setLevel(lvl):

设置日志的级别。对于低于该级别的日志消息将被忽略。下面一个例子演示setLevel方法：

Python

```
#coding=gbk
import logging

1 #coding=gbk
2 import logging
3 logging.basicConfig(filename = os.path.join(os.getcwd(), 'log.txt'), level = logging.DEBUG)
4 log = logging.getLogger('root.test')
5 log.setLevel(logging.WARN) #日志记录级别为WARNNING
6 log.info('info') #不会被记录
7 log.debug('debug') #不会被记录
8 log.warning('warning')
9 log.error('error')
```

Logger.debug(msg [, *args [, **kwargs]])

记录DEBUG级别的日志信息。参数msg是信息的格式，args与kwargs分别是格式参数。

Python

```
import logging
logging.basicConfig(file
```

```
1 import logging
2 logging.basicConfig(filename = os.path.join(os.getcwd(), 'log.txt'), level = logging.DEBUG)
3 log = logging.getLogger('root')
4 log.debug('%s, %s, %s', *('error', 'debug', 'info'))
5 log.debug('%(module)s, %(info)s', {'module': 'log', 'info': 'error'})
```

Logger.info(msg[, *args[, **kwargs]])

Logger.warnings(msg[, *args[, **kwargs]])

Logger.error(msg[, *args[, **kwargs]])

Logger.critical(msg[, *args[, **kwargs]])

记录相应级别的日志信息。参数的含义与Logger.debug一样。

Logger.log(lvl, msg[, *args[, **kwargs]])

记录日志，参数lvl用户设置日志信息的级别。参数msg, *args, **kwargs的含义与Logger.debug一样。

Logger.exception(msg[, *args])

以ERROR级别记录日志消息，异常跟踪信息将被自动添加到日志消息里。Logger.exception通过用在异常处理块中，如：

Python

```
import logging
logging.basicConfig(file
```

```
1 import logging
2 logging.basicConfig(filename = os.path.join(os.getcwd(), 'log.txt'), level = logging.DEBUG)
3 log = logging.getLogger('root')
4 try:
5     raise Exception, 'this is a exception'
6 except:
7     log.exception('exception') #异常信息被自动添加到日志消息中
```

Logger.addFilter(filt)

Logger.removeFilter(filt)

添加/移除日志消息过滤器。在讲述Filter时具体介绍。

Logger.addHandler(hdlr)

Logger.removeHandler(hdlr)

添加/移除日志消息处理器。在讲述Handler时具体介绍。

Logger.makeRecord(name, lvl, fn, lno, msg, args, exc_info[, func, extra])

创建LogRecord对象。日志消息被实例为一个LogRecord对象，并在日志类内处理。

1 赞 4 收藏 [评论](#)

Python模块学习： zipfile zip文件操作

原文出处：[DarkBull](#)

最近在写一个网络客户端下载程序，用于下载服务器上的数据。有些数据（如文本，office文档）如果直接传输的话，将会增加通信的数据量，使下载时间变长。服务器在传输这些数据之前先对其进行压缩，客户端接收到数据之后进行解压，这样可以减小网通传输数据的通信量，缩短下载的时间，从而增加客户体验。以前用C#做类似应用程序的时候，我会用SharpZipLib这个开源组件，现在用Python做类似的工作，只要使用zipfile模块提供的api就可以轻松的完成。

zip文件格式是通用的文档压缩标准，在ziplib模块中，使用ZipFile类来操作zip文件，下面具体介绍一下：

class zipfile.ZipFile(file[, mode[, compression[, allowZip64]]])

创建一个ZipFile对象，表示一个zip文件。参数file表示文件的路径或类文件对象(file-like object)；参数mode指示打开zip文件的模式，默认值为'r'，表示读已经存在的zip文件，也可以为'w'或'a'，'w'表示新建一个zip文档或覆盖一个已经存在的zip文档，'a'表示将数据附加到一个现存的zip文档中。参数compression表示在写zip文档时使用的压缩方法，它的值可以是zipfile.ZIP_STORED 或zipfile.ZIP_DEFLATED。如果要操作的zip文件大小超过2G，应该将allowZip64设置为True。

ZipFile还提供了如下常用的方法和属性：

ZipFile.getinfo(name):

获取zip文档内指定文件的信息。返回一个zipfile.ZipInfo对象，它包括文件的详细信息。将在[下面](#) 具体介绍该对象。

ZipFile.infolist()

获取zip文档内所有文件的信息，返回一个zipfile.ZipInfo的列表。

ZipFile.namelist()

获取zip文档内所有文件的名称列表。

ZipFile.extract(member[, path[, pwd]])

将zip文档内的指定文件解压到当前目录。参数member指定要解压的文件名称或对应的ZipInfo对象；参数path指定了解析文件保存的文件夹；参数pwd为解压密码。下面一个例子将保存在程序根目录下的txt.zip内的所有文件解压到D:/Work目录：

Python

```
import zipfile, os
zipFile =
1 import zipfile, os
```

```
2 zipFile = zipfile.ZipFile(os.path.join(os.getcwd(), 'txt.zip'))
3 for file in zipFile.namelist():
4     zipFile.extract(file, r'd:/Work')
5 zipFile.close()
```

ZipFile.extractall([path[, members[, pwd]]])

解压zip文档中的所有文件到当前目录。参数members的默认值为zip文档内的所有文件名称列表，也可以自己设置，选择要解压的文件名称。

ZipFile.printdir()

将zip文档内的信息打印到控制台上。

ZipFile.setpassword(pwd)

设置zip文档的密码。

ZipFile.read(name[, pwd])

获取zip文档内指定文件的二进制数据。下面的例子演示了read()的使用，zip文档内包括一个txt.txt的文本文件，使用read()方法读取其二进制数据，然后保存到D:/txt.txt。

Python

```
#coding=gbk
import zipfile, os

1 #coding=gbk
2 import zipfile, os
3 zipFile = zipfile.ZipFile(os.path.join(os.getcwd(), 'txt.zip'))
4 data = zipFile.read('txt.txt')
5 (lambda f, d: (f.write(d), f.close()))(open(r'd:/txt.txt', 'wb'), data) #一行语句就完成了写文件操作。仔细琢磨哦~~
6 zipFile.close()
```

ZipFile.write(filename[, arcname[, compress_type]])

将指定文件添加到zip文档中。filename为文件路径，arcname为添加到zip文档之后保存的名称，参数compress_type表示压缩方法，它的值可以是zipfile.ZIP_STORED 或zipfile.ZIP_DEFLATED。下面的例子演示了如何创建一个zip文档，并将文件D:/test.doc添加到压缩文档中。

Python

```
import zipfile, os
zipFile =

1 import zipfile, os
2 zipFile = zipfile.ZipFile(r'D:/test.zip'), 'w')
3 zipFile.write(r'D:/test.doc', 'ok.doc', zipfile.ZIP_DEFLATED)
4 zipFile.close()
```

ZipFile.writestr(zinfo_or_arcname, bytes)

writestr()支持将二进制数据直接写入到压缩文档。

Class ZipInfo

ZipFile.getinfo(name) 方法返回的是一个ZipInfo对象，表示zip文档中相应文件的信息。它支持如下属性：

ZipInfo.filename： 获取文件名称。

ZipInfo.date_time： 获取文件最后修改时间。返回一个包含6个元素的元组：(年, 月, 日, 时, 分, 秒)

ZipInfo.compress_type： 压缩类型。

ZipInfo.comment： 文档说明。

ZipInfo.extr： 扩展项数据。

ZipInfo.create_system： 获取创建该zip文档的系统。

ZipInfo.create_version： 获取 创建zip文档的PKZIP版本。

ZipInfo.extract_version： 获取 解压zip文档所需的PKZIP版本。

ZipInfo.reserved： 预留字段，当前实现总是返回0。

ZipInfo.flag_bits： zip标志位。

ZipInfo.volume： 文件头的卷标。

ZipInfo.internal_attr： 内部属性。

ZipInfo.external_attr： 外部属性。

ZipInfo.header_offset： 文件头偏移位。

ZipInfo.CRC： 未压缩文件的CRC-32。

ZipInfo.compress_size： 获取压缩后的大小。

ZipInfo.file_size： 获取未压缩的文件大小。

下面一个简单的例子说明这些属性的意思：

Python

```
import zipfile, os
zipFile =
1  import zipfile, os
2  zipFile = zipfile.ZipFile(os.path.join(os.getcwd(), 'txt.zip'))
3  zipInfo = zipFile.getinfo('doc.doc')
4  print 'filename:', zipInfo.filename
5  print 'date_time:', zipInfo.date_time
6  print 'compress_type:', zipInfo.compress_type
7  print 'comment:', zipInfo.comment
8  print 'extra:', zipInfo.extra
9  print 'create_system:', zipInfo.create_system
10 print 'create_version:', zipInfo.create_version
11 print 'extract_version:', zipInfo.extract_version
12 print 'extract_version:', zipInfo.reserved
13 print 'flag_bits:', zipInfo.flag_bits
14 print 'volume:', zipInfo.volume
15 print 'internal_attr:', zipInfo.internal_attr
16 print 'external_attr:', zipInfo.external_attr
17 print 'header_offset:', zipInfo.header_offset
18 print 'CRC:', zipInfo.CRC
19 print 'compress_size:', zipInfo.compress_size
20 print 'file_size:', zipInfo.file_size
21 zipFile.close()
```

感觉使用zipfile模块来处理zip文件真的很简单。想当初在.NET平台下，使用sharpziplib压缩、解压一个文件，我花了N多时间，找了N多英文资源，才写出一个能压缩文件的demo。而现在使用Python，通过阅读python手册，一两个小时就掌握了zipfile模块的基本使用。哈哈，使用Python，真爽！

1 赞 收藏 [评论](#)

Python模块学习： subprocess 创建子进程

原文出处： [DarkBull](#)

最近，我们老大要我写一个守护者程序，对服务器进程进行守护。如果服务器不幸挂掉了，守护者能即时的重启应用程序。上网Google了一下，发现Python有很几个模块都可以创建进程。最终我选择使用subprocess模块，因为在Python手册中有这样一段话：

This module intends to replace several other, older modules and functions, such as: os.system、os.spawn*、os.popen*、popen2.*、commands.*

subprocess被用来替换一些老的模块和函数，如：os.system、os.spawn*、os.popen*、popen2.*、commands.*。可见，subprocess是被推荐使用的模块。

下面是一个很简单的例子，创建一个新进程，执行app1.exe，传入相当的参数，并打印出进程的返回值：

Python

```
import subprocess

1 import subprocess
2
3 returnCode = subprocess.call('app1.exe -a -b -c -d')
4 print 'returncode:', returnCode
5
6 #---- 结果 -----
7 #Python is powerful
8 #app1.exe
9 #-a
10 #-b
11 #-c
12 #-d
13 returncode: 0
```

app1.exe是一个非常简单的控制台程序，它只打印出传入的参数，代码如下：

Python

```
#include <iostream>

1 #include <iostream>
2 using namespace std;
3
4 int main(int argc, const char *argv[])
5 {
6     cout <<< "Python is powerful" <<< endl;
7     for (int i = 0; i < argc; i++)
8     {
9         cout <<< argv[i] <<< endl;
10    }
11
12    return 0;
13 }
```

闲话少说，下面开始详细介绍subprocess模块。subprocess模块中只定义了一个类: Popen。可以使用Popen来创建进程，并与进程进行复杂的交互。它的构造函数如下：

```
subprocess.Popen(args, bufsize=0, executable=None, stdin=None, stdout=None,  
stderr=None, preexec_fn=None, close_fds=False, shell=False, cwd=None, env=None,  
universal_newlines=False, startupinfo=None, creationflags=0)
```

参数args可以是字符串或者序列类型（如：list，元组），用于指定进程的可执行文件及其参数。如果是序列类型，第一个元素通常是可执行文件的路径。我们也可以显式的使用executable参数来指定可执行文件的路径。在windows操作系统上，Popen通过调用CreateProcess()来创建子进程,CreateProcess接收一个字符串参数，如果args是序列类型，系统将会通过list2cmdline()函数将序列类型转换为字符串。

参数bufsize：指定缓冲。我到现在还不清楚这个参数的具体含义，望各个大牛指点。

参数executable用于指定可执行程序。一般情况下我们通过args参数来设置所要运行的程序。如果将参数shell设为True，executable将指定程序使用的shell。在windows平台下，默认的shell由COMSPEC环境变量来指定。

参数stdin, stdout, stderr分别表示程序的标准输入、输出、错误句柄。他们可以是PIPE，文件描述符或文件对象，也可以设置为None，表示从父进程继承。

参数preexec_fn只在Unix平台下有效，用于指定一个可执行对象（callable object），它将在子进程运行之前被调用。

参数Close_fds：在windows平台下，如果close_fds被设置为True，则新创建的子进程将不会继承父进程的输入、输出、错误管道。我们不能将close_fds设置为True同时重定向子进程的标准输入、输出与错误(stdin, stdout, stderr)。

如果参数shell设为true，程序将通过shell来执行。

参数cwd用于设置子进程的当前目录。

参数env是字典类型，用于指定子进程的环境变量。如果env = None，子进程的环境变量将从父进程中继承。

参数Universal_newlines:不同操作系统下，文本的换行符是不一样的。如：windows下用'\r\n'表示换，而Linux下用'\n'。如果将此参数设置为True，Python统一把这些换行符当作'\n'来处理。

参数startupinfo与creationflags只在windows下用效，它们将被传递给底层的CreateProcess()函数，用于设置子进程的一些属性，如：主窗口的外观，进程的优先级等等。

subprocess.PIPE

在创建Popen对象时，subprocess.PIPE可以初始化stdin, stdout或stderr参数。表示与子进程通信的标准流。

subprocess.STDOUT

创建Popen对象时，用于初始化stderr参数，表示将错误通过标准输出流输出。

Popen的方法：

Popen.poll()

用于检查子进程是否已经结束。设置并返回returncode属性。

Popen.wait()

等待子进程结束。设置并返回returncode属性。

Popen.communicate(input=None)

与子进程进行交互。向stdin发送数据，或从stdout和stderr中读取数据。可选参数input指定发送到子进程的参数。Communicate()返回一个元组：(stdoutdata, stderrdata)。注意：如果希望通过进程的stdin向其发送数据，在创建Popen对象的时候，参数stdin必须被设置为PIPE。同样，如果希望从stdout和stderr获取数据，必须将stdout和stderr设置为PIPE。

Popen.send_signal(signal)

向子进程发送信号。

Popen.terminate()

停止(stop)子进程。在windows平台下，该方法将调用Windows API TerminateProcess () 来结束子进程。

Popen.kill()

杀死子进程。

Popen.stdin

如果在创建Popen对象是，参数stdin被设置为PIPE，Popen.stdin将返回一个文件对象用于策子进程发送指令。否则返回None。

Popen.stdout

如果在创建Popen对象是，参数stdout被设置为PIPE，Popen.stdout将返回一个文件对象用于策子进程发送指令。否则返回None。

Popen.stderr

如果在创建Popen对象是，参数stdout被设置为PIPE，Popen.stdout将返回一个文件对象用于策子进程发送指令。否则返回None。

Popen.pid

获取子进程的进程ID。

Popen.returncode

获取进程的返回值。如果进程还没有结束，返回None。

下面是一个非常简单的例子，来演示subprocess模块如何与一个控件台应用程序进行交互。

Python

```
import subprocess

1 import subprocess
2
3 p = subprocess.Popen("app2.exe", stdin = subprocess.PIPE, /
4     stdout = subprocess.PIPE, stderr = subprocess.PIPE, shell = False)
5
6 p.stdin.write('3/n')
7 p.stdin.write('4/n')
8 print p.stdout.read()
9
10 #--- 结果 ---
11 input x:
12 input y:
13 3 + 4 = 7
```

app2.exe也是一个非常简单的控制台程序，它从界面上接收两个数值，执行加操作，并将结果打印到控制台上。代码如下：

Python

```
#include
<iostream>

1 #include <iostream>
2 using namespace std;
3
4 int main(int argc, const char *argv[])
5 {
6     int x, y;
7     cout <<< "input x: " <<< endl;
8     cin >>> x;
9     cout <<< "input y: " <<< endl;
10    cin >>> y;
11    cout <<< x <<< " + " <<< y <<< " = " <<< x + y <<< endl;
12
13    return 0;
14 }
```

subprocess模块提供了一些函数，方便我们用于创建进程。

subprocess.call(*popenargs, **kwargs)

运行命令。该函数将一直等待到子进程运行结束，并返回进程的returncode。文章一开始的例子就演

示了call函数。如果子进程不需要进行交互,就可以使用该函数来创建。

subprocess.check_call(*popenargs, **kwargs)

与subprocess.call(*popenargs, **kwargs)功能一样,只是如果子进程返回的returncode不为0的话,将触发CalledProcessError异常。在异常对象中,包括进程的returncode信息。

subprocess模块的内容就这么多。在Python手册中,还介绍了如何使用subprocess来替换一些老的模块,老的函数的例子。赶兴趣的朋友可以看一下。

参考文档:

[subprocess — Subprocess management](#)

[PyMoTW:subprocess](#)

1 赞 1 收藏 [评论](#)

Python模块学习：tempfile 临时文件(夹)操作

原文出处：[DarkBull](#)

应用程序经常要保存一些临时的信息，这些信息不是特别重要，没有必要写在配置文件里，但又不能没有，这时候就可以把这些信息写到临时文件里。其实很多程序在运行的时候，都会产生一大堆临时文件，有些用于保存日志，有些用于保存一些临时数据，还有一些保存一些无关紧要的设置。在windows操作系统中，临时文件一般被保存在这个文件夹下：C:/Documents and Settings/User/Local Settings/Temp。其实我们最常用的IE浏览器在浏览网页的时候，会产生大量的临时文件，这些临时文件一般是我们浏览过的网页的本地副本。Python提供了一个tempfile模块，用来对临时数据进行操作。查阅Python手册，里面介绍了如下常用的方法：

tempfile.mkstemp([suffix="", prefix='tmp', dir=None, text=False])

mkstemp方法用于**创建**一个临时文件。该方法**仅仅用于创建临时文件**，调用tempfile.mkstemp函数后，返回包含两个元素的元组，第一个元素指示操作该临时文件的安全级别，第二个元素指示该临时文件的路径。参数suffix和prefix分别表示临时文件名称的后缀和前缀；dir指定了临时文件所在的目录，如果没有指定目录，将根据系统环境变量TMPDIR, TEMP或者TMP的设置来保存临时文件；参数text指定了是否以文本的形式来操作文件，默认为False，表示以二进制的形式来操作文件。

tempfile.mkdtemp([suffix="", prefix='tmp', dir=None])

该函数用于创建一个临时文件夹。参数的意思与tempfile.mkdtemp一样。它返回临时文件夹的绝对路径。

tempfile.mktemp([suffix="", prefix='tmp', dir=None])

mktemp用于返回一个临时文件的路径，但并不创建该临时文件。

tempfile.tempdir

该属性用于指定创建的临时文件（夹）所在的默认文件夹。如果没有设置该属性或者将其设为None，Python将返回以下环境变量TMPDIR, TEMP, TEMP指定的目录，如果没有定义这些环境变量，临时文件将被创建在当前工作目录。

tempfile.gettempdir()

gettempdir()则用于返回保存临时文件的文件夹路径。

tempfile.TemporaryFile([mode='w+b', bufsize=-1, suffix="", prefix='tmp', dir=None])

该函数返回一个 类文件 对象(file-like)用于临时数据保存（实际上对应磁盘上的一个临时文件）。当文件对象被close或者被del的时候，临时文件将从磁盘上删除。mode、bufsize参数的单方与open()函数一样；suffix和prefix指定了临时文件名的后缀和前缀；dir用于设置临时文件默认的保存路径。返

回的文件对象有一个file属性，它指向真正操作的底层的file对象。

tempfile.NamedTemporaryFile([mode='w+b', bufsize=-1, suffix='', prefix='tmp', dir=None, delete=True])

tempfile.NamedTemporaryFile函数的行为与tempfile.TemporaryFile类似，只不过它多了一个delete参数，用于指定文件对象close或者被del之后，是否也一同删除磁盘上的临时文件（当delete = True的时候，行为与TemporaryFile一样）。

tempfile.SpooledTemporaryFile([max_size=0, mode='w+b', bufsize=-1, suffix='', prefix='tmp', dir=None])

tempfile.SpooledTemporaryFile函数的行为与tempfile.TemporaryFile类似。不同的是向文件对象写数据的时候，数据长度只有到达参数max_size指定大小时，或者调用文件对象的fileno()方法，数据才会真正写入到磁盘的临时文件中。

蛮简单、实用的一个模块，不是吗？

1 赞 1 收藏 [评论](#)

Python模块学习：zlib 数据压缩

原文出处： [DarkBull](#)

Python标准模块中，有多个模块用于数据的压缩与解压缩，如zipfile，gzip, bz2等等。上次介绍了zipfile模块，今天就来讲讲zlib模块。

zlib.compress(string[, level])

zlib.decompress(string[, wbits[, bufsize]])

zlib.compress用于压缩流数据。参数string指定了要压缩的数据流，参数level指定了压缩的级别，它的取值范围是1到9。压缩速度与压缩率成反比，1表示压缩速度最快，而压缩率最低，而9则表示压缩速度最慢但压缩率最高。zlib.decompress用于解压数据。参数string指定了需要解压的数据，wbits和bufsize分别用于设置系统缓冲区大小(window buffer)与输出缓冲区大小(output buffer)。下面用一个例子来演示如何使用这两个方法：

Python

```
#coding=gbk
1  #coding=gbk
2
3  import zlib, urllib
4
5  fp = urllib.urlopen('http://localhost/default.html')
6  str = fp.read()
7  fp.close()
8
9  #---- 压缩数据流。
10 str1 = zlib.compress(str, zlib.Z_BEST_COMPRESSION)
11 str2 = zlib.decompress(str1)
12 print len(str)
13 print len(str1)
14 print len(str2)
15 # ---- 结果
16 #5783
17 #1531
18 #5783
19
```

我们也可以使用Compress/Decompress对象来对数据进行压缩/解压缩。**zlib.compressobj([level])**与**zlib.decompress(string[, wbits[, bufsize]])** 分别创建Compress/Decompress对象。通过对象对数据进行压缩和解压缩的使用方式与上面介绍的zlib.compress,zlib.decompress非常类似。但两者对数据的压缩还是有区别的，这主要体现在对大量数据进行操作的情况下。假如现在要压缩一个非常大的数据文件（上百M），如果使用zlib.compress来压缩的话，必须先一次性将文件里的数据读到内存里，然后将数据进行压缩。这样势必会占用太多的内存。如果使用对象来进行压缩，那么没有必要一次性读取文件的所有数据，可以先读一部分数据到内存里进行压缩，压缩完后写入文件，然后再读其他部分的数据压缩，如此循环重复，直到压缩完整个文件。下面一个例子来演示这之间的区别：

Python



```
1 #coding=gbk
2
3 import zlib, urllib
4
5 fp = urllib.urlopen('http://localhost/default.html') # 访问的到的网址。
6 data = fp.read()
7 fp.close()
8
9 #---- 压缩数据流
10 str1 = zlib.compress(data, zlib.Z_BEST_COMPRESSION)
11 str2 = zlib.decompress(str1)
12 print '原始数据长度: ', len(data)
13 print '-' * 30
14 print 'zlib.compress压缩后: ', len(str1)
15 print 'zlib.decompress解压后: ', len(str2)
16 print '-' * 30
17
18 #---- 使用Compress, Decompress对象对数据流进行压缩/解压缩
19 com_obj = zlib.compressobj(zlib.Z_BEST_COMPRESSION)
20 decom_obj = zlib.decompressobj()
21
22 str_obj = com_obj.compress(data)
23 str_obj += com_obj.flush()
24 print 'Compress.compress压缩后: ', len(str_obj)
25
26 str_obj1 = decom_obj.decompress(str_obj)
27 str_obj1 += decom_obj.flush()
28 print 'Decompress.decompress解压后: ', len(str_obj1)
29 print '-' * 30
30
31 #---- 使用Compress, Decompress对象, 对数据进行分块压缩/解压缩。
32 com_obj1 = zlib.compressobj(zlib.Z_BEST_COMPRESSION)
33 decom_obj1 = zlib.decompressobj()
34 chunk_size = 30;
35
36 #原始数据分块
37 str_chunks = [data[i * chunk_size:(i + 1) * chunk_size] /
38               for i in range((len(data) + chunk_size) / chunk_size)]
39
40 str_obj2 = ""
41 for chunk in str_chunks:
42     str_obj2 += com_obj1.compress(chunk)
43 str_obj2 += com_obj1.flush()
44 print '分块压缩后: ', len(str_obj2)
45
46 #压缩数据分块解压
47 str_chunks = [str_obj2[i * chunk_size:(i + 1) * chunk_size] /
48               for i in range((len(str_obj2) + chunk_size) / chunk_size)]
49 str_obj2 = ""
50 for chunk in str_chunks:
51     str_obj2 += decom_obj1.decompress(chunk)
52 str_obj2 += decom_obj1.flush()
53 print '分块解压后: ', len(str_obj2)
54 # ---- 结果 -----
55 原始数据长度: 5783
56 -----
57 zlib.compress压缩后: 1531
58 zlib.decompress解压后: 5783
59 -----
60 Compress.compress压缩后: 1531
```

```
61 Decompress.decompress解压后: 5783
62 -----
63 分块压缩后: 1531
64 分块解压后: 5783
65
```

Python手册对zlib模块的介绍比较详细，更具体的应用，可以参考Python手册。

1 赞 2 收藏 [1 评论](#)

Python模块学习：pickle, cPickle 对象序列化/反序列化

原文出处：[DarkBull](#)

上次学习过marshal模块用于序列化和反序列化，但marshal的功能比较薄弱，只支持部分内置数据类型的序列化/反序列化，对于用户自定义的类型就无能为力，同时marshal不支持自引用(递归引用)的对象的序列化。所以直接使用marshal来序列化/反序列化可能不是很方便。还好，python标准库提供了功能更加强大且更加安全的pickle和cPickle模块。

cPickle模块是使用C语言实现的，所以在运行效率上比pickle要高。但是cPickle模块中定义的类型不能被继承（其实大多数时候，我们不需要从这些类型中继承。）。cPickle和pickle的序列化/反序列化规则是一样的，我们可以使用pickle序列化一个对象，然后使用cPickle来反序列化。同时，这两个模块在处理自引用类型时会变得更加“聪明”，它不会无限制的递归序列化自引用对象，对于同一对象的多次引用，它只会序列化一次。例如：

Python

```
import marshal, pickle

1 import marshal, pickle
2 list = [1]
3 list.append(list)
4 byt1 = marshal.dumps(list) #出错, 无限制的递归序列化
5 byt2 = pickle.dumps(list) #No problem
6
```

pickle的序列化规则

Python规范（Python-specific）提供了pickle的序列化规则。这就不必担心不同版本的Python之间序列化兼容性问题。默认情况下，pickle的序列化是基于文本的，我们可以直接用文本编辑器查看序列化的文本。我们也可以序列成二进制格式的数据，这样的结果体积会更小。更详细的内容，可以参考Python手册pickle模块。

下面就开始使用pickle吧~

`pickle.dump(obj, file[, protocol])`

序列化对象，并将结果数据流写入到文件对象中。参数protocol是序列化模式，默认值为0，表示以文本的形式序列化。protocol的值还可以是1或2，表示以二进制的形式序列化。

`pickle.load(file)`

反序列化对象。将文件中的数据解析为一个Python对象。下面通过一个简单的例子来演示上面两个方法的使用：

```
#coding=gbk

1
2
3 #coding=gbk
4 import pickle, StringIO
5
6 class Person(object):
7     """自定义类型。
8     """
9     def __init__(self, name, address):
10         self.name = name
11         self.address = address
12
13     def display(self):
14         print 'name:', self.name, 'address:', self.address
15 jj = Person("JGood", "中国 杭州")
16 jj.display()
17 file = StringIO.StringIO()
18 pickle.dump(jj, file, 0) #序列化
19 #print file.getvalue() #打印序列化后的结果
20 #del Person #反序列的时候，必须能找到对应类的定义。否则反序列化操作失败。
21 file.seek(0)
22 jj1 = pickle.load(file) #反序列化
23 jj1.display()
24 file.close()
25
26
```

注意：在反序列化的时候，必须能找到对应类的定义，否则反序列化将失败。在上面的例子中，如果取消`#del Person`的注释，在运行时将抛`AttributeError`异常，提示当前模块找不到`Person`的定义。

`pickle.dumps(obj[, protocol])`

`pickle.loads(string)`

我们也可以直接获取序列化后的数据流，或者直接从数据流反序列化。方法`dumps`与`loads`就完成这样的功能。`dumps`返回序列化后的数据流，`loads`返回的序列化生成的对象。

`python`模块中还定义了两个类，分别用来序列化、反序列化对象。

`class pickle.Pickler(file[, protocol]):`

该类用于序列化对象。参数`file`是一个类文件对象(file-like object)，用于保存序列化结果。可选参数表示序列化模式。它定义了两个方法：

`dump(obj):`

将对象序列化，并保存到类文件对象中。参数`obj`是要序列化的对象。

`clear_memo()`

清空pickler的“备忘”。使用Pickler实例在序列化对象的时候，它会“记住”已经被序列化的对象引用，所以对同一对象多次调用dump(obj)，pickler不会“傻傻”的去多次序列化。下面是一个简单的例子：

Python

```
#coding=gbk
import pickle, StringIO

1
2
3
4 #coding=gbk
5 import pickle, StringIO
6 class Person(object):
7     """自定义类型。
8     """
9     def __init__(self, name, address):
10         self.name = name
11         self.address = address
12
13     def display(self):
14         print 'name:', self.name, 'address:', self.address
15
16 file = StringIO.StringIO()
17 pick = pickle.Pickler(file)
18 person = Person("JGood", "Hangzhou China")
19 pick.dump(person)
20 val1 = file.getvalue()
21 print len(val1)
22 pick.clear_memo() #注释此句，再看看运行结果
23 pick.dump(person) #对同一引用对象再次进行序列化
24 val2 = file.getvalue()
25 print len(val2)
26 #---- 结果 ----
27 #148
28 #296
29 #
30 #将这行代码注释掉：pick.clear_memo()
31 #结果为：
32 #148
33 #152
34
35
```

class pickle.Unpickler(file):

该类用于反序列化对象。参数file是一个类文件(file-like object)对象，Unpickler从该参数中获取数据进行反序列化。

load():

反序列化对象。该方法会根据已经序列化的数据流，自动选择合适的反序列化模式。

Python

```
#.... 接上个例子中的代
码
```

```
1 #.... 接上个例子中的代码
2 fle.seek(0)
3 unpick = pickle.Unpickler(fle)
4 print unpick.load()
5
```

上面介绍了pickle模块的基本使用，但和marshal一样，并不是所有的类型都可以通过pickle序列化的。例如对于一个嵌套的类型，使用pickle序列化就失败。例如：

Python

```
class A(object):
    class B(object):
1
2 class A(object):
3     class B(object):
4         def __init__(self, name):
5             self.name = name
6
7     def __init__(self):
8         print 'init A'
9 b = A.B("my name")
10 print b
11 c = pickle.dumps(b, 0) #失败哦
12 print pickle.loads(c)
```

关于pickle支持的序列化类型，可以参考Python手册。

Python手册中的pickle模块，介绍了更高级的主题，例如自定义序列化过程。有时间再和大家分享。

1 赞 1 收藏 [评论](#)

Python模块学习：marshal 对象的序列化

原文出处：[DarkBull](#)

有时候，要把内存中的一个对象持久化保存到磁盘上，或者序列化成二进制流通过网络发送到远程主机上。Python中有很多模块提供了序列化与反序列化的功能，如：marshal, pickle, cPickle等等。今天就讲讲marshal模块。

注意： marshal并不是一个通用的模块，在某些时候它是一个不被推荐使用的模块，因为使用marshal序列化的二进制数据格式还没有文档化，在不同版本的Python中，marshal的实现可能不一样。也就是说，用python2.5序列为一个对象，用python2.6的程序反序列化所得到的对象，可能与原来的对象是不一样的。但这个模块存在的意义，正如Python手册中所说：The marshal module exists mainly to support reading and writing the “pseudo-compiled” code for Python modules of .pyc files.

下面是marshal模块中定义的一些与序列化/反序列化有关的函数：

marshal.dump(value, file[, version])

将值写入到一个打开的输出流里。参数value表示待序列化的值。file表示打开的输出流。如:以“wb”模式打开的文件，sys.stdout或者os.popen。对于一些不支持序列类的类型，dump方法将抛出ValueError异常。要特别说明一下，并不是所有类型的对象都可以使用marshal模块来序列化/反序列化的。在python2.6中，支持的类型包括：None, integers, long integers, floating point numbers, strings, Unicode objects, tuple, list, set, dict, 和 code objects。对于tuple, list, set, dict等集合对象，其中的元素必须也是上述类型之一。

marshal.load(file)

执行与marshal.dump相反的操作，将二进制数据反序列为Python对象。下面是一个例子，演示这两个方法的使用：

Python

```
# coding=gbk

1
2
3
4 # coding=gbk
5 import marshal, sys, os
6 lst = [ 1, ( 2, "string" ), { "key": "Value" } ]
7 # 序列化到文件中
8 file = open ( os . path . join ( os . getcwd ( ), ' file . txt ' ), ' wb ' )
9 marshal . dump ( lst , file )
10 file . close ( )
11 # 反序列化
12 fle1 = open ( os . path . join ( os . getcwd ( ), ' fle . txt ' ), ' rb ' )
13 lst1 = marshal . load ( fle1 )
14 fle1 . close ( )
15 # 打印结果
16 print lst
```

```
17 print lst1
18 # ---- 结果 ----
19 # [1, (2, 'string'), {'key': 'Value'}]
20 # [1, (2, 'string'), {'key': 'Value'}]
21
22
23
```

marshal.dumps(value[, version])

该方法与上面讲的marshal.dump()功能类似，只是它返回的是序列化之后的二进制流，而不是将这些数据直接写入到文件中。

marshal.load(string)

将二进制流反序列化为对象。下面的一段代码，演示这两个方法的使用：

Python

```
import marshal, sys, os
1
2
3 import marshal, sys, os
4 lst = [ 1, ( 2, "string" ), { "key" : " Value " } ]
5 byt1 = marshal.dumps ( lst )
6 lst1 = marshal.loads ( byt1 )
7 # 打印结果
8 print lst
9 print lst1
10 # ---- 结果 ----
11 # [1, (2, 'string'), {'key': 'Value'}]
12 # [1, (2, 'string'), {'key': 'Value'}]
13
14
```

更多关于marshal的内容，请参考Python手册。

1 赞 收藏 [评论](#)

Python模块学习：copy 对象拷贝

原文出处：[DarkBull](#)

copy模块用于对象的拷贝操作。该模块非常简单，只提供了两个主要的方法：copy.copy 与 copy.deepcopy，分别表示浅复制与深复制。什么是浅复制，什么是深复制，网上有一卡车一卡车的资料，这里不作详细介绍。复制操作只对复合对象有效。用简单的例子来分别介绍这两个方法。

浅复制只复制对象本身，没有复制该对象所引用的对象。

Python

```
#coding=gbk
import copy

1 #coding=gbk
2 import copy
3 l1 = [1, 2, [3, 4]]
4 l2 = copy.copy(l1)
5 print l1
6 print l2
7 l2[2][0] = 50
8 print l1
9 print l2
10 #---- 结果 ----
11 [1, 2, [3, 4]]
12 [1, 2, [3, 4]]
13 [1, 2, [50, 4]]
14 [1, 2, [50, 4]]
```

同样的代码，使用深复制，结果就不一样：

Python

```
import copy
l1 = [1, 2, [3, 4]]

1 import copy
2 l1 = [1, 2, [3, 4]]
3 l2 = copy.deepcopy(l1)
4 print l1
5 print l2
6 l2[2][0] = 50
7 print l1
8 print l2
9 #---- 结果 ----
10 [1, 2, [3, 4]]
11 [1, 2, [3, 4]]
12 [1, 2, [3, 4]]
13 [1, 2, [50, 4]]
```

改变copy的默认行为

在定义类的时候，通过定义__copy__和__deepcopy__方法，可以改变copy的默认行为。下面是一个简单的例子：

Python

```
class CopyObj(object):  
    def __repr__(self):
```

```
1 class CopyObj(object):  
2     def __repr__(self):  
3         return "CopyObj"  
4  
5     def __copy__(self):  
6         return "Hello"  
7 obj = CopyObj()  
8 obj1 = copy.copy(obj)  
9 print obj  
10 print obj1  
11 #---- 结果 ----  
12 CopyObj  
13 Hello
```

1 赞 收藏 [评论](#)

在Python 3中实现类型检查器

本文由 [伯乐在线](#) - [PyPer](#) 翻译, [Daetalus](#) 校稿。未经许可, 禁止转载!
英文出处: www.enotagain.com。欢迎加入[翻译组](#)。

示例函数

为了开发类型检查器, 我们需要一个简单的函数对其进行实验。欧几里得算法就是一个完美的例子:

Python

```
def gcd(a, b):  
    """Return the greatest  
1 def gcd(a, b):  
2     """Return the greatest common divisor of a and b."""  
3     a = abs(a)  
4     b = abs(b)  
5     if a < b:  
6         a, b = b, a  
7     while b != 0:  
8         a, b = b, a % b  
9     return a
```

在上面的示例中, 参数 `a` 和 `b` 以及返回值应该是 `int` 类型的。预期的类型将会以函数注解的形式来表达, 函数注解是 Python 3 的一个新特性。接下来, 类型检查机制将会以一个装饰器的形式实现, 注解版本的第一行代码是:

Python

```
def gcd(a: int, b: int) -> int:  
1 def gcd(a: int, b: int) -> int:
```

使用“`gcd.__annotations__`”可以获得一个包含注解的字典:

Python

```
>>>  
gcd.__annotations__  
1 >>> gcd.__annotations__  
2 {'return': <class 'int'>, 'b': <class 'int'>, 'a': <class 'int'>}  
3 >>> gcd.__annotations__['a']  
4 <class 'int'>
```

需要注意的是, 返回值的注解存储在键“`return`”下。这是有可能的, 因为“`return`”是一个关键字, 所以不能用作一个有效的参数名。

检查返回值类型

返回值注解存储在字典“`__annotations__`”中的“`return`”键下。我们将使用这个值来检查返回值（假设

注解存在)。我们将参数传递给原始函数，如果存在注解，我们将通过注解中的值来验证其类型：

Python

```
def typecheck(f):
    def wrapper(*args,
1 def typecheck(f):
2     def wrapper(*args, **kwargs):
3         result = f(*args, **kwargs)
4         return_type = f.__annotations__.get('return', None)
5         if return_type and not isinstance(result, return_type):
6             raise RuntimeError("{} should return {}".format(f.__name__, return_type.__name__))
7         return result
8     return wrapper
```

我们可以用“a”替换函数gcd的返回值来测试上面的代码：

Python

```
Traceback (most recent
call last):
1 Traceback (most recent call last):
2 File "typechecker.py", line 9, in <module>
3     gcd(1, 2)
4 File "typechecker.py", line 5, in wrapper
5     raise RuntimeError("{} should return {}".format(f.__name__, return_type.__name__))
6 RuntimeError: gcd should return int
```

由上面的结果可知，确实检查了返回值的类型。

检查参数类型

函数的参数存在于关联代码对象的“co_varnames”属性中，在我们的例子中是“gcd.__code__.co_varnames”。元组包含了所有局部变量的名称，并且该元组以参数开始，参数数量存储在“co_nlocals”中。我们需要遍历包括索引在内的所有变量，并从参数“args”中获取参数值，最后对其进行类型检查。

得到了下面的代码：

Python

```
def typecheck(f):
    def wrapper(*args,
def typecheck(f):
1     def wrapper(*args, **kwargs):
2         for i, arg in enumerate(args[:f.__code__.co_nlocals]):
3             name = f.__code__.co_varnames[i]
4             expected_type = f.__annotations__.get(name, None)
5             if expected_type and not isinstance(arg, expected_type):
6                 raise RuntimeError("{} should be of type {}; {} specified".format(name, expected_type.__name__,
7 type(arg).__name__))
8             result = f(*args, **kwargs)
9             return_type = f.__annotations__.get('return', None)
10            if return_type and not isinstance(result, return_type):
```



```

11         raise RuntimeError("{} should return {}".format(f.__name__, return_type.__name__))
12     return result
13     return wrapper

```

在上面的循环中，`i`是数组`args`中参数的以0起始的索引，`arg`是包含其值的字符串。可以利用“`f.__code__.co_varnames[i]`”读取到参数的名称。类型检查代码与返回值类型检查完全一样（包括错误消息的异常）。

为了对关键字参数进行类型检查，我们需要遍历参数`kwargs`。此时的类型检查几乎与第一个循环中相同：

Python

```

for name, arg in
kwargs.items():
1 for name, arg in kwargs.items():
2     expected_type = f.__annotations__.get(name, None)
3     if expected_type and not isinstance(arg, expected_type):
4         raise RuntimeError("{} should be of type {}; {} specified".format(name, expected_type.__name__,
type(arg).__name__))

```

得到的装饰器代码如下：

Python

```

def typecheck(f):
    def wrapper(*args,
def typecheck(f):
1     def wrapper(*args, **kwargs):
2         for i, arg in enumerate(args[:f.__code__.co_nlocals]):
3             name = f.__code__.co_varnames[i]
4             expected_type = f.__annotations__.get(name, None)
5             if expected_type and not isinstance(arg, expected_type):
6                 raise RuntimeError("{} should be of type {}; {} specified".format(name, expected_type.__name__,
7 type(arg).__name__))
8         for name, arg in kwargs.items():
9             expected_type = f.__annotations__.get(name, None)
10            if expected_type and not isinstance(arg, expected_type):
11                raise RuntimeError("{} should be of type {}; {} specified".format(name, expected_type.__name__,
12 type(arg).__name__))
13            result = f(*args, **kwargs)
14            return_type = f.__annotations__.get('return', None)
15            if return_type and not isinstance(result, return_type):
16                raise RuntimeError("{} should return {}".format(f.__name__, return_type.__name__))
17            return result
    return wrapper

```

将类型检查代码写成一个函数将会使代码更加清晰。为了简化代码，我们修改错误信息，而当返回值是无效的类型时，将会使用到这些错误信息。我们也可以利用 `functools` 模块中的 `wraps` 方法，将包装函数的一些属性复制到 `wrapper` 中（这使得 `wrapper` 看起来更像原来的函数）：

Python

```

def typecheck(f):
    def

```

```
1 def typecheck(f):
2     def do_typecheck(name, arg):
3         expected_type = f.__annotations__.get(name, None)
4         if expected_type and not isinstance(arg, expected_type):
5             raise RuntimeError("{} should be of type {} instead of {}".format(name, expected_type.__name__,
6 type(arg).__name__))
7
8     @functools.wraps(f)
9     def wrapper(*args, **kwargs):
10         for i, arg in enumerate(args[:f.__code__.co_nlocals]):
11             do_typecheck(f.__code__.co_varnames[i], arg)
12         for name, arg in kwargs.items():
13             do_typecheck(name, arg)
14
15         result = f(*args, **kwargs)
16
17         do_typecheck('return', result)
18         return result
19     return wrapper
```

结论

注解是 Python 3 中的一个新元素，本文例子中的使用方法很普通，你也可以想象很多特定领域的应用。虽然上面的实现代码并不能满足实际产品要求，但它的目的本来就是用作概念验证。可以对其进行以下改善：

- 处理额外的参数（args 中意想不到的项目）
- 默认值类型检查
- 支持多个类型
- 支持模板类型（例如，int 型列表）

1 赞 收藏 [评论](#)

关于作者：[PyPer](#)



一名就读于羊城某高校的学生，主要关注 Python、Perl、PowerShell 等脚本技术，熟悉 MSSQL、Oracle、Redis 等数据库，新浪微博：<http://weibo.com/LlwianwpIO>，微信公众号“NETEC”。[个人主页](#) · [我的文章](#) · [11](#)

python模块学习：Cookie

原文出处：[DarkBull](#)

最近在用GAE开发自己的博客程序。虽然GAE的API没有显式的提供操作Cookie的方法，但他现有的架构，使我们有足够的自由来操作Cookie。

Cookie 模块，顾名思义，就是用来操作Cookie的模块。Cookie这块小蛋糕，玩过Web的人都知道，它是Server与Client保持会话时用到的信息切片。Http协议本身是无状态的，也就是说，同一个客户端发送的两次请求，对于Web服务器来说，没有直接的关系。既然这样，有人会问，既然Http是无状态的，为什么有些网页，只有输入了用户名与密码通过验证之后才可以访问？那是因为：对于通过身份验证的用户，Server会偷偷的在发往Client的数据中添加 Cookie，Cookie中一般保存一个标识该Client的唯一的ID，Client在接下来对服务器的请求中，会将该ID以Cookie的形式一并发往Server，Server从回传回来的Cookie中提取ID并与相应的用户绑定起来，从而实现身份验证。**说白了，Cookie就是一个在服务器与客户端之间相互传递的字符串（下图通过FireFox的FireBug插件查看访问google.com时的Cookie）。**越扯越远了，回到我们的主题：Python标准模块 — Cookie。

Request Headers

```
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.1)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1, utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: PREF=ID=31b15d39a5b1a284:TM=1270039200:LM=1270039200:S=
1dOLQqddQ-BgZZJHhISG1ETDWD5pfSZgZQ6DqJvxj5Rhr7q18-ND4M
yNCWYzXLdovQIVQ-YzokqRLQ_O_UOVcjNhZOR9ssHzSR71Pj11n-V1
,KSLP-A-kQaVWPrub8UTmrP
```

（上图是Http请求头中的Cookie信息）

Headers Response

Response Headers

```
Location: http://www.google.com.hk/url?sa=p&ck=PREF=3DID=3D31b15d39a5b1a284:FF=3D2:LD=3D
:LM=3D1270127533:SM=3DLj-ZUvgFsuCnbpCQ&q=http://www.google.com.hk/&ust=12701275E
Wd9M5fEx7KeyAmEQVBQ___uow
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Set-Cookie: NID=33=SCeHd0nhYOBokQCjZJeoXnlGYbIHRAMNi889Io7r2QzSw1qc9D8NcK9umZqC8hrhVh2d5GJF
: expires=Fri, 01-Oct-2010 13:12:13 GMT; path=/; domain=.google.com; HttpOnly
Date: Thu, 01 Apr 2010 13:12:13 GMT
```

（上图是Http响应中的Cookie信息）

Cooke模块中定义了4个直接操作Cookie的类：BaseCookie、SimpleCookie、SerialCookie、SmartCookie。其中，BaseCookie是基类，定义了操作Cookie的公共部分，其他3个类都继承自BaseCookie，它们之间的区别仅仅在于序列化数据的方式不同。下面简单讲解这些类的使用。

BaseCookie基类：BaseCookies的行为非常像dict，可以用键/值对的形式来操作它，但是kye必须是字符串，value是Morsel对象（下面会讲到Morsel）。BaseCookies定义了编码/解码，输入/输出

操作的公共规范：

BaseCookie.value_encode(val)：对数据进行序列化/反序列化。这些方法都返回字符串，以便通过Http传输。

BaseCookie.output()：返回字符串，该字符串可以作为Http响应头发往客户端。

BaseCookie.js_output()：返回嵌入js脚本的字符串，浏览器通过执行该脚本，就可以得到cookie数据。

BaseCookie.load(newdata)：解析字符串为Cookie数据。

SimpleCookie、SerialCookie、SmartCookie都继承自BaseCookie，具有一致的行为，它们各自对BaseCookie的value_decode, value_encode进行了重写并实现自己的序列化/反序列化策略，其中：

- SimpleCookie内部使用str()来对数据进行序列化；
- SerialCookie则通过pickle模块来序列化反序列化数据；
- SmartCookie相对聪明点，对于非字符串数据，使用pickle序列/反序列化，否则将字符串原样返回。

下面的例子简单的说明如何使用Cookie模块：

Python

```
import Cookie<br>
<br>
1 import Cookie<br>
2 <br>
3 c = Cookie.SimpleCookie()<br>
4 c['name'] = 'DarkBull'<br>
5 c['address'] = 'ChinaHangZhou'<br>
6 c['address']['path'] = '/' # 路径<br>
7 c['address']['domain'] = 'appspot.com' # domain<br>
8 c['address']['expires'] = 'Fri, 01-Oct-2010 20:00:00 GMT' # 过期时间<br>
9 print c.output()<br>
10 print c.js_output()<br>
11 <br>
12 # 输出结果,与上图对照<br>
13 # Set-Cookie: address=ChinaHangZhou; Domain=appspot.com; expires=Fri, 01-Oct-2010 20:00:00 GMT; Path=/<br>
14 # Set-Cookie: name=DarkBull<br>
15 <br>
16 # 作为脚本输出<br>
17 # <script type="text/javascript"><br>
18 # document.cookie = "address=ChinaHangZhou; Domain=appspot.com; expires=Fri, 01-Oct-2010 20:00:00 GMT;
19 Path=/"<br>
20 # </script><br>
21 <br>
22 # <script type="text/javascript"><br>
23 # document.cookie = "name=DarkBull";<br>
24 # </script>
```

Morsel类：用于表示Cookie中每一项数据的属性而抽象的类。这些属性包括：expires, path, comment, domain, max-age, secure, version等等（看上图下划线标注部分）。如果你玩过web，对这些应该不会陌生，可以在RFC2109中找到他们的具体定义

Morsel.key, Morsel.value: Cookie数据项的key/value(value可以是二进制数据);

Morsel.coded_value: 数据编码后得到的字符串。Http协议是基于文本的协议, Server无法直接向Client发送二进制数据, 只有序列化成字符串后, 才能发往Client;

Morsel.set(key, value, coded_value): 设置Cookie数据项的key、value、coded_value;

Morsel.isReversvedKey(key): 如果key是expires, path, comment, domain, max-age, secure, version, httponly中的一个, 返回True, 否则返回False;

Morsel.output(): 返回型如“Set-Cookie: ...”的字符串, 表示一个Cookie数据项;

Morsel.js_output(): 返回Cookie数据项的脚本字符串;

Morsel.OutputString(): 返回Morsel的字符串表示;

Morsel使用示例:

Python

```
import Cookie<br>
<br>
1 import Cookie<br>
2 <br>
3 m = Cookie.Morsel()<br>
4 m.set('name', 'DarkBull', 'DarkBull')<br>
5 m['expires'] = 'Fri, 01-Oct-2010 20:00:00 GMT'<br>
6 m['domain'] = 'appspot.com'<br>
7 print m.output()<br>
8 <br>
9 # 结果<br>
10 # Set-Cookie: name=DarkBull; Domain=appspot.com; expires=Fri, 01-Oct-2010 20:00:00 GMT
```

关于Cookie模块更详细的内容, 可以Python手册。

1 赞 1 收藏 [评论](#)

Python模块学习：datetime

原文出处：[DarkBull](#)

Python提供了多个内置模块用于操作日期时间，像calendar, time, datetime。time模块我在之前的文章已经有所介绍，它提供的接口与C标准库time.h基本一致。相比于time模块，datetime模块的接口则更直观、更容易调用。今天就来讲讲datetime模块。

datetime模块定义了两个常量：datetime.MINYEAR和datetime.MAXYEAR，分别表示datetime所能表示的最小、最大年份。其中，MINYEAR = 1，MAXYEAR = 9999。（对于偶等玩家，这个范围已经足够用矣~~）

datetime模块定义了下面这几个类：

- datetime.date：表示日期的类。常用的属性有year, month, day；
- datetime.time：表示时间的类。常用的属性有hour, minute, second, microsecond；
- datetime.datetime：表示日期时间。
- datetime.timedelta：表示时间间隔，即两个时间点之间的长度。
- datetime.tzinfo：与时区有关的相关信息。（这里不详细充分讨论该类，感兴趣的童鞋可以参考python手册）

注：上面这些类型的对象都是不可变（immutable）的。

下面详细介绍这些类的使用方式。

date类

date类表示一个日期。日期由年、月、日组成（地球人都知道~~）。date类的构造函数如下：

class datetime.date(year, month, day)：参数的意义就不多作解释了，只是有几点要注意一下：

- year的范围是[MINYEAR, MAXYEAR]，即[1, 9999]；
- month的范围是[1, 12]。（月份是从1开始的，不是从0开始的~~）；
- day的最大值根据给定的year, month参数来决定。例如闰年2月份有29天；

date类定义了一些常用的类方法与类属性，方便我们操作：

- date.max、date.min：date对象所能表示的最大、最小日期；
- date.resolution：date对象表示日期的最小单位。这里是天。
- date.today()：返回一个表示当前本地日期的date对象；
- date.fromtimestamp(timestamp)：根据给定的时间戳，返回一个date对象；
- datetime.fromordinal(ordinal)：将Gregorian日历时间转换为date对象；（Gregorian Calendar：一种日历表示方法，类似于我国的农历，西方国家使用比较多，此处不详细展开讨论。）

使用例子：

Python

```
from datetime import *
import time

1 from datetime import *
2 import time
3
4 print 'date.max:', date.max
5 print 'date.min:', date.min
6 print 'date.today():', date.today()
7 print 'date.fromtimestamp():', date.fromtimestamp(time.time())
8
9 ## ---- 结果 ----
10 # date.max: 9999-12-31
11 # date.min: 0001-01-01
12 # date.today(): 2010-04-06
13 # date.fromtimestamp(): 2010-04-06
```

date提供的实例方法和属性：

- date.year、date.month、date.day：年、月、日；
- date.replace(year, month, day)：生成一个新的日期对象，用参数指定的年，月，日代替原有对象中的属性。（原有对象仍保持不变）
- date.timetuple()：返回日期对应的time.struct_time对象；
- date.toordinal()：返回日期对应的Gregorian Calendar日期；
- date.weekday()：返回weekday，如果是星期一，返回0；如果是星期二，返回1，以此类推；
- date.isoweekday()：返回weekday，如果是星期一，返回1；如果是星期二，返回2，以此类推；
- date.isocalendar()：返回格式如(year, month, day)的元组；
- date.isoformat()：返回格式如'YYYY-MM-DD'的字符串；
- date.strftime(fmt)：自定义格式化字符串。在下面详细讲解。

使用例子：

Python

```
now = date(2010, 04, 06)

1 now = date(2010, 04, 06)
2 tomorrow = now.replace(day = 07)
3 print 'now:', now, ', tomorrow:', tomorrow
4 print 'timetuple():', now.timetuple()
5 print 'weekday():', now.weekday()
6 print 'isoweekday():', now.isoweekday()
7 print 'isocalendar():', now.isocalendar()
8 print 'isoformat():', now.isoformat()
9
10 ## ---- 结果 ----
11 # now: 2010-04-06 , tomorrow: 2010-04-07
12 # timetuple(): (2010, 4, 6, 0, 0, 0, 1, 96, -1)
13 # weekday(): 1
14 # isoweekday(): 2
15 # isocalendar(): (2010, 14, 2)
16 # isoformat(): 2010-04-06
```

date还对某些操作进行了重载，它允许我们对日期进行如下一些操作：

- `date2 = date1 + timedelta` # 日期加上一个间隔，返回一个新的日期对象（`timedelta`将在下面介绍，表示时间间隔）
- `date2 = date1 - timedelta` # 日期隔去间隔，返回一个新的日期对象
- `timedelta = date1 - date2` # 两个日期相减，返回一个时间间隔对象
- `date1 < date2` # 两个日期进行比较

注：对日期进行操作时，要防止日期超出它所能表示的范围。

使用例子：

Python

```
now = date.today()
tomorrow =
1 now = date.today()
2 tomorrow = now.replace(day = 7)
3 delta = tomorrow - now
4 print 'now:', now, ' tomorrow:', tomorrow
5 print 'timedelta:', delta
6 print now + delta
7 print tomorrow > now
8
9 ## ---- 结果 ----
10 # now: 2010-04-06 tomorrow: 2010-04-07
11 # timedelta: 1 day, 0:00:00
12 # 2010-04-07
13 # True
```

Time类

time类表示时间，由时、分、秒以及微秒组成。（我不是从火星来的~~）time类的构造函数如下：

`class datetime.time(hour[, minute[, second[, microsecond[, tzinfo]]]])`：各参数的意义不作解释，这里留意一下参数`tzinfo`，它表示时区信息。注意一下各参数的取值范围：`hour`的范围为[0, 24)，`minute`的范围为[0, 60)，`second`的范围为[0, 60)，`microsecond`的范围为[0, 1000000)。

time类定义类属性：

- `time.min`、`time.max`：time类所能表示的最小、最大时间。其中，`time.min = time(0, 0, 0, 0)`，`time.max = time(23, 59, 59, 999999)`；
- `time.resolution`：时间的最小单位，这里是1微秒；

time类提供的实例方法和属性：

- `time.hour`、`time.minute`、`time.second`、`time.microsecond`：时、分、秒、微秒；
- `time.tzinfo`：时区信息；
- `time.replace([hour[, minute[, second[, microsecond[, tzinfo]]]])`：创建一个新的时间对象，用参数指定的时、分、秒、微秒代替原有对象中的属性（原有对象仍保持不变）；
- `time.isoformat()`：返回型如“HH:MM:SS”格式的字符串表示；

- `time.strftime(fmt)`：返回自定义格式化字符串。在下面详细介绍；

使用例子：

Python

```
from datetime import *
tm = time(23, 46, 10)

1 from datetime import *
2 tm = time(23, 46, 10)
3 print 'tm:', tm
4 print 'hour: %d, minute: %d, second: %d, microsecond: %d' /
5 % (tm.hour, tm.minute, tm.second, tm.microsecond)
6 tm1 = tm.replace(hour = 20)
7 print 'tm1:', tm1
8 print 'isoformat():', tm.isoformat()
9
10 # # ---- 结果 ----
11 # tm: 23:46:10
12 # hour: 23, minute: 46, second: 10, microsecond: 0
13 # tm1: 20:46:10
14 # isoformat(): 23:46:10
```

像`date`一样，也可以对两个`time`对象进行比较，或者相减返回一个时间间隔对象。这里就不提供例子了。

datetime类

`datetime`是`date`与`time`的结合体，包括`date`与`time`的所有信息。它的构造函数如

下：`datetime.datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]])`，各参数的含义与`date`、`time`的构造函数中的一样，要注意参数值的范围。

`datetime`类定义类属性与方法：

- `datetime.min`、`datetime.max`：`datetime`所能表示的最小值与最大值；
- `datetime.resolution`：`datetime`最小单位；
- `datetime.today()`：返回一个表示当前本地时间的`datetime`对象；
- `datetime.now([tz])`：返回一个表示当前本地时间的`datetime`对象，如果提供了参数`tz`，则获取`tz`参数所指时区的本地时间；
- `datetime.utcnow()`：返回一个当前`utc`时间的`datetime`对象；
- `datetime.fromtimestamp(timestamp[, tz])`：根据时间戳创建一个`datetime`对象，参数`tz`指定时区信息；
- `datetime.utcfromtimestamp(timestamp)`：根据时间戳创建一个`datetime`对象；
- `datetime.combine(date, time)`：根据`date`和`time`，创建一个`datetime`对象；
- `datetime.strptime(date_string, format)`：将格式字符串转换为`datetime`对象；

使用例子：

Python

```
from datetime import *
import time
```

```
1 from datetime import *
2 import time
3
4 print 'datetime.max:', datetime.max
5 print 'datetime.min:', datetime.min
6 print 'datetime.resolution:', datetime.resolution
7 print 'today():', datetime.today()
8 print 'now():', datetime.now()
9 print 'utcnow():', datetime.utcnow()
10 print 'fromtimestamp(tmstamp):', datetime.fromtimestamp(time.time())
11 print 'utcfromtimestamp(tmstamp):', datetime.utcfromtimestamp(time.time())
12
13 # ---- 结果 ----
14 # datetime.max: 9999-12-31 23:59:59.999999
15 # datetime.min: 0001-01-01 00:00:00
16 # datetime.resolution: 0:00:00.000001
17 # today(): 2010-04-07 09:48:16.234000
18 # now(): 2010-04-07 09:48:16.234000
19 # utcnow(): 2010-04-07 01:48:16.234000 # 中国位于+8时间，与本地时间相差8
20 # fromtimestamp(tmstamp): 2010-04-07 09:48:16.234000
21 # utcfromtimestamp(tmstamp): 2010-04-07 01:48:16.234000
```

datetime类提供的实例方法与属性（很多属性或方法在date和time中已经出现过，在此有类似的意义，这里只罗列这些方法名，具体含义不再逐个展开介绍，可以参考上文对date与time类的讲解。）：

- datetime.year、month、day、hour、minute、second、microsecond、tzinfo：
- datetime.date()：获取date对象；
- datetime.time()：获取time对象；
- datetime.replace([year[, month[, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]]]])：
- datetime.timetuple()
- datetime.utctimetuple()
- datetime.toordinal()
- datetime.weekday()
- datetime.isocalendar()
- datetime.isoformat([sep])
- datetime.ctime()：返回一个日期时间的C格式字符串，等效于time.ctime(time.mktime(dt.timetuple()))；
- datetime.strftime(format)

像date一样，也可以对两个datetime对象进行比较，或者相减返回一个时间间隔对象，或者日期时间加上一个间隔返回一个新的日期时间对象。这里不提供详细的例子，看客自己动手试一下~~

格式字符串

datetime、date、time都提供了strftime()方法，该方法接收一个格式字符串，输出日期时间的字符串表示。下表是从python手册中拉过来的，我对些进行了简单的翻译（翻译的有点噢口~~）。

格式字符 意义

%a: 星期的简写。如 星期三为Web
%A: 星期的全写。如 星期三为Wednesday
%b: 月份的简写。如4月份为Apr
%B: 月份的全写。如4月份为April
%c: 日期时间的字符串表示。（如： 04/07/10 10:43:39）
%d: 日在这个月中的天数（是这个月的第几天）
%f: 微秒（范围[0,999999]）
%H: 小时（24小时制，[0, 23]）
%I: 小时（12小时制，[0, 11]）
%j: 日在年中的天数 [001,366]（是当年的第几天）
%m: 月份（[01,12]）
%M: 分钟（[00,59]）
%p: AM或者PM
%S: 秒（范围为[00,61]，为什么不是[00, 59]，参考python手册~_~）
%U: 周在当年的周数当年的第几周），星期天作为周的第一天
%w: 今天在这周的天数，范围为[0, 6]，6表示星期天
%W: 周在当年的周数（是当年的第几周），星期一作为周的第一天
%x: 日期字符串（如： 04/07/10）
%X: 时间字符串（如： 10:43:39）
%y: 2个数字表示的年份
%Y: 4个数字表示的年份
%z: 与utc时间的间隔（如果是本地时间，返回空字符串）
%Z: 时区名称（如果是本地时间，返回空字符串）
%%: %% => %

例子：

Python

```
dt = datetime.now()

1 dt = datetime.now()
2 print '(%Y-%m-%d %H:%M:%S %f): ', dt.strftime('%Y-%m-%d %H:%M:%S %f')
3 print '(%Y-%m-%d %H:%M:%S %p): ', dt.strftime('%y-%m-%d %I:%M:%S %p')
4 print '%%a: %s ' % dt.strftime('%a')
5 print '%%A: %s ' % dt.strftime('%A')
6 print '%%b: %s ' % dt.strftime('%b')
7 print '%%B: %s ' % dt.strftime('%B')
8 print '日期时间%%c: %s ' % dt.strftime('%c')
9 print '日期%%x: %s ' % dt.strftime('%x')
10 print '时间%%X: %s ' % dt.strftime('%X')
11 print '今天是这周的第%s天 ' % dt.strftime('%w')
12 print '今天是今年的第%s天 ' % dt.strftime('%j')
13 print '今周是今年的第%s周 ' % dt.strftime('%U')
14
15 # # ---- 结果 ----
16 # (%Y-%m-%d %H:%M:%S %f): 2010-04-07 10:52:18 937000
17 # (%Y-%m-%d %H:%M:%S %p): 10-04-07 10:52:18 AM
18 # %a: Wed
19 # %A: Wednesday
```

```
20 # %b: Apr
21 # %B: April
22 # 日期时间%c: 04/07/10 10:52:18
23 # 日期%x: 04/07/10
24 # 时间%X: 10:52:18
25 # 今天是这周的第3天
26 # 今天是今年的第097天
27 # 今周是今年的第14周
```

这些就是datetime模块的基本内容，总算写完了~~oh yeah~~

1 赞 1 收藏 [评论](#)

Python模块学习：filecmp 文件比较

原文出处：[DarkBull](#)

filecmp模块用于比较文件及文件夹的内容，它是一个轻量级的工具，使用非常简单。python标准库还提供了difflib模块用于比较文件的内容。关于difflib模块，且听下回分解。

filecmp定义了两个函数，用于方便地比较文件与文件夹：

filecmp.cmp(f1, f2[, shallow]):

比较两个文件的内容是否匹配。参数f1, f2指定要比较的文件的路径。可选参数shallow指定比较文件时是否需要考虑文件本身的属性（通过os.stat函数可以获得文件属性）。如果文件内容匹配，函数返回True，否则返回False。

filecmp.cmpfiles(dir1, dir2, common[, shallow]):

比较两个文件夹内**指定文件**是否相等。参数dir1, dir2指定要比较的文件夹，参数common指定要比较的文件名列表。函数返回包含3个list元素的元组，分别表示匹配、不匹配以及错误的文件列表。错误的文件指的是不存在的文件，或文件被锁定不可读，或没权限读文件，或者由于其他原因访问不了该文件。

filecmp模块中定义了一个dircmp类，用于比较文件夹，通过该类比较两个文件夹，可以获取一些详细的比较结果（如只在A文件夹存在的文件列表），并支持子文件夹的递归比较。

dircmp提供了三个方法用于**报告比较的结果**：

- report(): 只比较指定文件夹中的内容（文件与文件夹）
- report_partial_closure(): 比较文件夹及第一级子文件夹的内容
- report_full_closure(): 递归比较所有的文件夹的内容

dircmp还提供了下面这些属性用于**获取比较的详细结果**：

- left_list: 左边文件夹中的文件与文件夹列表；
- right_list: 右边文件夹中的文件与文件夹列表；
- common: 两边文件夹中都存在的文件或文件夹；
- left_only: 只在左边文件夹中存在的文件或文件夹；
- right_only: 只在右边文件夹中存在的文件或文件夹；
- common_dirs: 两边文件夹都存在的子文件夹；
- common_files: 两边文件夹都存在的子文件；
- common_funny: 两边文件夹都存在的子文件夹；
- same_files: 匹配的文件；
- diff_files: 不匹配的文件；
- funny_files: 两边文件夹中都存在，但无法比较的文件；
- subdirs: 我没看明白这个属性的意思，python手册中的解释如下：A dictionary mapping names in common_dirs to dircmp objects

简单就是美！我只要文件比较的结果，不想去关心文件是如何是比较的，hey，就用python吧~~

1 赞 收藏 [评论](#)

Python模块学习：urllib

原文出处：[DarkBull](#)

urllib模块提供的上层接口，使我们可以像读取本地文件一样读取www和ftp上的数据。每当使用这个模块的时候，老是会想起公司产品的客户端，同事用C++下载Web上的图片，那种“痛苦”的表情。我以前翻译过libcurl教程，这是在C/C++环境下比较方便实用的网络操作库，相比起libcurl，Python的urllib模块的使用门槛则低多了。可能有些人又会用效率来批评Python，其实在操作网络，或者在集群交互的时候，语言的执行效率绝不是瓶颈。这种情况下，一个比较好的方法是，将python嵌入到C/C++中，让Python来完成一些不是核心的逻辑处理。又扯远了，废话少说，开始urllib之旅吧~~（前几天我用这个模块写了个蜘蛛，感兴趣的同学可以在以前的博客中找到代码）

先看一个例子，这个例子把Google首页的html抓取下来并显示在控制台上：

Python

```
# 别惊讶，整个程序确实只用了两行代码
```

```
1 # 别惊讶，整个程序确实只用了两行代码
2 import urllib
3 print urllib.urlopen('http://www.google.com').read()
```

urllib.urlopen(url[, data[, proxies]]) :

创建一个表示远程url的类文件对象，然后像本地文件一样操作这个类文件对象来获取远程数据。参数url表示远程数据的路径，一般是网址；参数data表示以post方式提交到url的数据(玩过web的人应该知道提交数据的两种方式：post与get。如果你不清楚，也不必太在意，一般情况下很少用到这个参数)；参数proxies用于设置代理（这里不详细讲怎么使用代理，感兴趣的看客可以去翻阅Python手册urllib模块）。urlopen返回一个类文件对象，他提供了如下方法：

- read(), readline(), readlines(), fileno(), close()：这些方法的使用方式与文件对象完全一样；
- info()：返回一个httplib.HTTPMessage对象，表示远程服务器返回的头信息；
- getcode()：返回Http状态码。如果是http请求，200表示请求成功完成;404表示网址未找到；
- geturl()：返回请求的url；

下面来扩充一下上面的例子，看官可以运行一下这个例子，加深对urllib的印象：

Python

```
google = urllib.urlopen('http://www.google.com')
```

```
1 google = urllib.urlopen('http://www.google.com')
2 print 'http header:\n', google.info()
3 print 'http status:', google.getcode()
4 print 'url:', google.geturl()
5 for line in google: # 就像在操作本地文件
6     print line,
7 google.close()
```


urllib.urlretrieve(url[, filename[, reporthook[, data]]]):

urlretrieve方法直接将远程数据下载到本地。参数filename指定了保存到本地的路径（如果未指定该参数，urllib会生成一个临时文件来保存数据）；参数reporthook是一个回调函数，当连接上服务器、以及相应的数据块传输完毕的时候会触发该回调。我们可以利用这个回调函数来显示当前的下载进度，下面的例子会展示。参数data指post到服务器的数据。该方法返回一个包含两个元素的元组(filename, headers)，filename表示保存到本地的路径，header表示服务器的响应头。下面通过例子来演示一下这个方法的使用，这个例子将新浪首页的html抓取到本地，保存在D:/sina.html文件中，同时显示下载的进度。

Python

```
def cbk(a, b, c):  
    &#039;&#039;&#039;  
1 def cbk(a, b, c):  
2     &#039;&#039;&#039;回调函数  
3     @a: 已经下载的数据块  
4     @b: 数据块的大小  
5     @c: 远程文件的大小  
6     &#039;&#039;&#039;  
7     per = 100.0 * a * b / c  
8     if per > 100:  
9         per = 100  
10    print &#039;%.2f%%&#039; % per  
11  
12 url = &#039;http://www.sina.com.cn&#039;  
13 local = &#039;d:/sina.html&#039;  
14 urllib.urlretrieve(url, local, cbk)
```

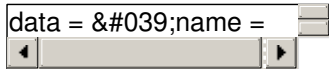
上面介绍的两个方法是urllib中最常用的方法，这些方法在获取远程数据的时候，内部会使用URLopener或者FancyURLopener类。作为urllib的使用者，我们很少会用到这两个类，这里我不想多讲。如果对urllib的实现感兴趣，或者希望urllib支持更多的协议，可以研究这两个类。在Python手册中，urllib的作者还列出了这个模块的缺陷和不足，感兴趣的同学可以打开Python手册了解一下。

urllib中还提供了一些辅助方法，用于对url进行编码、解码。url中是不能出现一些特殊的符号的，有些符号有特殊的用途。我们知道以get方式提交数据的时候，会在url中添加key=value这样的字符串，所以在value中是不允许有'='，因此要对其进行编码；与此同时服务器接收到这些参数的时候，要进行解码，还原成原始的数据。这个时候，这些辅助方法会很有用：

- urllib.quote(string[, safe]): 对字符串进行编码。参数safe指定了不需要编码的字符；
- urllib.unquote(string) : 对字符串进行解码；
- urllib.quote_plus(string[, safe]) : 与urllib.quote类似，但这个方法用'+'来替换'='，而quote用'%20'来代替' '；
- urllib.unquote_plus(string) : 对字符串进行解码；
- urllib.urlencode(query[, doseq]): 将dict或者包含两个元素的元组列表转换成url参数。例如字典{'name': 'dark-bull', 'age': 200}将被转换为"name=dark-bull&age=200"
- urllib.pathname2url(path): 将本地路径转换成url路径；
- urllib.url2pathname(path): 将url路径转成本地路径；

用一个例子来体验一下这些方法吧~~：

Python



```
1 data = &#039;name = ~a+3&#039;
2
3 data1 = urllib.quote(data)
4 print data1 # result: name%20%3D%20%7Ea%2B3
5 print urllib.unquote(data1) # result: name = ~a+3
6
7 data2 = urllib.quote_plus(data)
8 print data2 # result: name+%3D+%7Ea%2B3
9 print urllib.unquote_plus(data2) # result: name = ~a+3
10
11 data3 = urllib.urlencode({ &#039;name&#039;: &#039;dark-bull&#039;, &#039;age&#039;: 200 })
12 print data3 # result: age=200&amp;name=dark-bull
13
14 data4 = urllib.pathname2url(r&#039;d:/a/b/c/23.php&#039;)
15 print data4 # result: ///D//a/b/c/23.php
16 print urllib.url2pathname(data4) # result: D:/a/b/c/23.php
```

urllib模块的基本使用，就这么简单。oh~~yeah~~又一个模块写完了，想想，我已经写了将近30个模块了，有时间我要好好整理一下@@@@

1 赞 3 收藏 [评论](#)

Python模块学习：atexit

原文出处：[DarkBull](#)

atexit模块很简单，只定义了一个register函数用于注册程序退出时的回调函数，我们可以在这个回调函数中做一些资源清理的操作。

注：如果程序是非正常crash，或者通过os._exit()退出，注册的回调函数将不会被调用。

我们也可以通过sys.exitfunc来注册回调，但通过它只能注册一个回调，而且还不支持参数。所以建议大家使用atexit来注册回调函数。但千万不要在程序中同时使用这两种方式，否则通过atexit注册的回调可能不会被正常调用。其实通过查阅atexit的源码，你会发现原来它内部是通过sys.exitfunc来实现的，它先把注册的回调函数放到一个列表中，当程序退出的时候，按先进后出的顺序调用注册的回调。如果回调函数在执行过程中抛出了异常，atexit会打印异常的文字信息，并继续执行下一下回调，直到所有的回调都执行完毕，它会重新抛出最后接收到的异常。

如果使用的python版本是2.6，还可以用装饰器的语法来注册回调函数。

下面是一个例子，展示了atexit模块的使用：

Python

```
import atexit

1  import atexit
2
3  def exit0(*args, **kwargs):
4      print 'exit0'
5      for arg in args:
6          print ' '*4, arg
7
8      for item in kwargs.items():
9          print ' '*4, item
10
11 def exit1():
12     print 'exit1'
13     raise Exception, 'exit1'
14
15 def exit2():
16     print 'exit2'
17
18 atexit.register(exit0, *[1, 2, 3], **{ "a": 1, "b": 2, })
19 atexit.register(exit1)
20 atexit.register(exit2)
21
22 @atexit.register
23 def exit3():
24     print 'exit3'
25
26 if __name__ == '__main__':
27     pass
```

下面是程序运行的结果，可以看到回调函数执行的顺序与它们被注册的顺序刚才相反。

```
exit3
exit2
exit1
Error in atexit._run_exitfuncs:
Traceback (most recent call last):
  File "c:\python26\lib\atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
  File "study_atexit.py", line 15, in exit1
    raise Exception, 'exit1'
Exception: exit1
exit0
1
2
3
('a', 1)
('b', 2)
Error in sys.exitfunc:
Traceback (most recent call last):
  File "c:\python26\lib\atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
  File "study_atexit.py", line 15, in exit1
    raise Exception, 'exit1'
Exception: exit1
```

打印异常信息

所有回调执行完毕后，
重新抛出异常

1 赞 1 收藏 [1 评论](#)