

TCP/IP简介

1117次阅读

虽然大家现在对互联网很熟悉，但是计算机网络的出现比互联网要早很多。

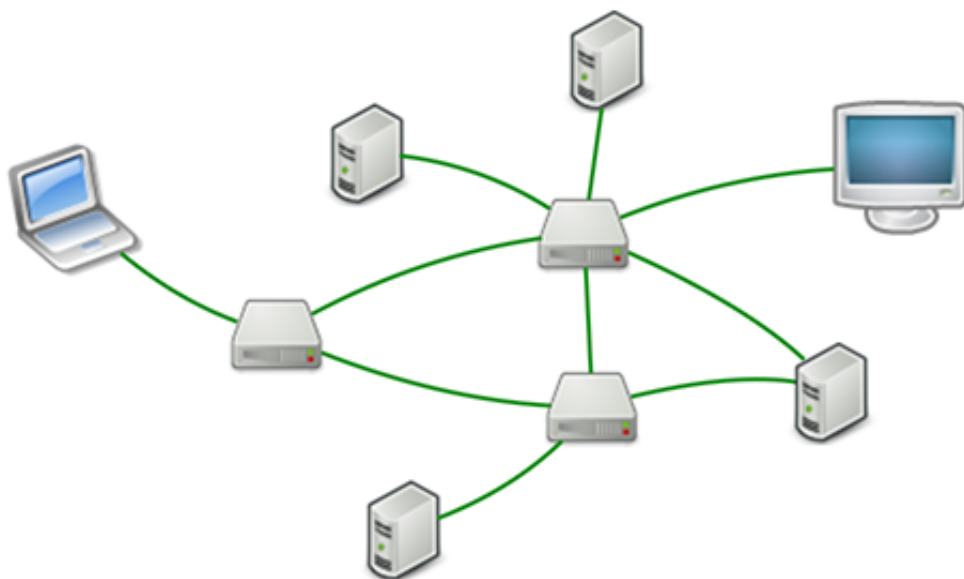
计算机为了联网，就必须规定通信协议，早期的计算机网络，都是由各厂商自己规定一套协议，IBM、Apple和Microsoft都有各自的网络协议，互不兼容，这就好比一群人有的说英语，有的说中文，有的说德语，说同一种语言的人可以交流，不同的语言之间就不行了。

为了把全世界的所有不同类型的计算机都连接起来，就必须规定一套全球通用的协议，为了实现互联网这个目标，互联网协议簇（Internet Protocol Suite）就是通用协议标准。Internet是由inter和net两个单词组合起来的，原意就是连接“网络”的网络，有了Internet，任何私有网络，只要支持这个协议，就可以联入互联网。

因为互联网协议包含了上百种协议标准，但是最重要的两个协议是TCP和IP协议，所以，大家把互联网的协议简称TCP/IP协议。

通信的时候，双方必须知道对方的标识，好比发邮件必须知道对方的邮件地址。互联网上每个计算机的唯一标识就是IP地址，类似123.123.123.123。如果一台计算机同时接入到两个或更多的网络，比如路由器，它就会有两个或多个IP地址，所以，IP地址对应的实际上是计算机的网络接口，通常是网卡。

IP协议负责把数据从一台计算机通过网络发送到另一台计算机。数据被分割成一小块一小块，然后通过IP包发送出去。由于互联网链路复杂，两台计算机之间经常有多条线路，因此，路由器就负责决定如何把一个IP包转发出去。IP包的特点是按块发送，途径多个路由，但不保证能到达，也不保证顺序到达。



TCP协议则是建立在IP协议之上的。TCP协议负责在两台计算机之间建立可靠连接，保证数据包按顺序到达。TCP协议会通过握手建立连接，然后，对每个IP包编号，确保对方按顺序收到，如果包丢掉了，就自动重发。

许多常用的更高级的协议都是建立在TCP协议基础上的，比如用于浏览器的HTTP协议、发送邮件的SMTP协议等。

一个IP包除了包含要传输的数据外，还包含源IP地址和目标IP地址，源端口和目标端口。

端口有什么作用？在两台计算机通信时，只发IP地址是不够的，因为同一台计算机上跑着多个

网络程序。一个IP包来了之后，到底是交给浏览器还是QQ，就需要端口号来区分。每个网络程序都向操作系统申请唯一的端口号，这样，两个进程在两台计算机之间建立网络连接就需要各自的IP地址和各自的端口号。

一个进程也可能同时与多个计算机建立链接，因此它会申请很多端口。

了解了TCP/IP协议的基本概念，IP地址和端口的概念，我们就可以开始进行网络编程了。

TCP编程

1152次阅读

Socket是网络编程的一个抽象概念。通常我们用一个Socket表示“打开了一个网络链接”，而打开一个Socket需要知道目标计算机的IP地址和端口号，再指定协议类型即可。

客户端

大多数连接都是可靠的TCP连接。创建TCP连接时，主动发起连接的叫客户端，被动响应连接的叫服务器。

举个例子，当我们在浏览器中访问新浪时，我们自己的计算机就是客户端，浏览器会主动向新浪的服务器发起连接。如果一切顺利，新浪的服务器接受了我们的连接，一个TCP连接就建立起来的，后面的通信就是发送网页内容了。

所以，我们要创建一个基于TCP连接的Socket，可以这样做：

```
# 导入socket库：
import socket
# 创建一个socket：
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 建立连接：
s.connect(('www.sina.com.cn', 80))
```

创建Socket时，AF_INET指定使用IPv4协议，如果要用更先进的IPv6，就指定为AF_INET6。SOCK_STREAM指定使用面向流的TCP协议，这样，一个Socket对象就创建成功，但是还没有建立连接。

客户端要主动发起TCP连接，必须知道服务器的IP地址和端口号。新浪网站的IP地址可以用域名www.sina.com.cn自动转换到IP地址，但是怎么知道新浪服务器的端口号呢？

答案是作为服务器，提供什么样的服务，端口号就必须固定下来。由于我们想要访问网页，因此新浪提供网页服务的服务器必须把端口号固定在80端口，因为80端口是Web服务的标准端口。其他服务都有对应的标准端口号，例如SMTP服务是25端口，FTP服务是21端口，等等。端口号小于1024的是Internet标准服务的端口，端口号大于1024的，可以任意使用。

因此，我们连接新浪服务器的代码如下：

```
s.connect(('www.sina.com.cn', 80))
```

注意参数是一个tuple，包含地址和端口号。

建立TCP连接后，我们就可以向新浪服务器发送请求，要求返回首页的内容：

```
# 发送数据：
s.send('GET / HTTP/1.1\r\nHost: www.sina.com.cn\r\nConnection: close\r\n\r\n')
```

TCP连接创建的是双向通道，双方都可以同时给对方发数据。但是谁先发谁后发，怎么协调，要根据具体的协议来决定。例如，HTTP协议规定客户端必须先发请求给服务器，服务器收到后才发数据给客户端。

发送的文本格式必须符合HTTP标准，如果格式没问题，接下来就可以接收新浪服务器返回的数据了：

```
# 接收数据：
```

```
buffer = []
while True:
    # 每次最多接收1k字节:
    d = s.recv(1024)
    if d:
        buffer.append(d)
    else:
        break
data = ''.join(buffer)
```

接收数据时，调用`recv(max)`方法，一次最多接收指定的字节数，因此，在一个`while`循环中反复接收，直到`recv()`返回空数据，表示接收完毕，退出循环。

当我们接收完数据后，调用`close()`方法关闭Socket，这样，一次完整的网络通信就结束了：

```
# 关闭连接:
s.close()
```

接收到的数据包括HTTP头和网页本身，我们只需要把HTTP头和网页分离一下，把HTTP头打印出来，网页内容保存到文件：

```
header, html = data.split('\r\n\r\n', 1)
print header
# 把接收的数据写入文件:
with open('sina.html', 'wb') as f:
    f.write(html)
```

现在，只需要在浏览器中打开这个`sina.html`文件，就可以看到新浪的首页了。

服务器

和客户端编程相比，服务器编程就要复杂一些。

服务器进程首先要绑定一个端口并监听来自其他客户端的连接。如果某个客户端连接过来了，服务器就分配一个随机端口号与该客户端建立连接，随后的通信就靠这个端口了。

所以，服务器会打开固定端口（比如80）监听，每来一个客户端连接，就打开一个新端口号创建该连接。由于服务器会有大量来自客户端的连接，但是每个连接都会分配不同的端口号，所以，服务器要给哪个客户端发数据，只要发到分配的端口就行。

但是服务器还需要同时响应多个客户端的请求，所以，每个连接都需要一个新的进程或者新的线程来处理，否则，服务器一次就只能服务一个客户端了。

我们来编写一个简单的服务器程序，它接收客户端连接，把客户端发过来的字符串加上Hello再发回去。

首先，创建一个基于IPv4和TCP协议的Socket：

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

然后，我们要绑定监听的地址和端口。服务器可能有多块网卡，可以绑定到某一块网卡的IP地址上，也可以用0.0.0.0绑定到所有的网络地址，还可以用127.0.0.1绑定到本机地址。127.0.0.1是一个特殊的IP地址，表示本机地址，如果绑定到这个地址，客户端必须同时在本机运行才能连接，也就是说，外部的计算机无法连接进来。

端口号需要预先指定。因为我们写的这个服务不是标准服务，所以用9999这个端口号。请注意，小于1024的端口号必须要有管理员权限才能绑定：

```
# 监听端口:
s.bind(('127.0.0.1', 9999))
```

紧接着，调用`listen()`方法开始监听端口，传入的参数指定等待连接的最大数量：

```
s.listen(5)
print 'Waiting for connection...'
```

接下来，服务器程序通过一个永久循环来接受来自客户端的连接，`accept()`会等待并返回一个客户端的连接：

```
while True:
    # 接受一个新连接:
    sock, addr = s.accept()
    # 创建新线程来处理TCP连接:
    t = threading.Thread(target=tcplink, args=(sock, addr))
    t.start()
```

每个连接都必须创建新线程（或进程）来处理，否则，单线程在处理连接的过程中，无法接受其他客户端的连接：

```
def tcplink(sock, addr):
    print 'Accept new connection from %s:%s...' % addr
    sock.send('Welcome!')
    while True:
        data = sock.recv(1024)
        time.sleep(1)
        if data == 'exit' or not data:
            break
        sock.send('Hello, %s!' % data)
    sock.close()
    print 'Connection from %s:%s closed.' % addr
```

连接建立后，服务器首先发一条欢迎消息，然后等待客户端数据，并加上Hello再发送给客户端。如果客户端发送了exit字符串，就直接关闭连接。

要测试这个服务器程序，我们还需要编写一个客户端程序：

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 建立连接:
s.connect(('127.0.0.1', 9999))
# 接收欢迎消息:
print s.recv(1024)
for data in ['Michael', 'Tracy', 'Sarah']:
    # 发送数据:
    s.send(data)
    print s.recv(1024)
s.send('exit')
s.close()
```

我们需要打开两个命令行窗口，一个运行服务器程序，另一个运行客户端程序，就可以看到效果了：



```
socket — Python — 50x11
Mavericks:socket michael$ python echo_server.py
Waiting for connection...
Accept new connection from 127.0.0.1:64398...
Connection from 127.0.0.1:64398 closed.

socket — bash — 50x11
Mavericks:socket michael$ python echo_client.py
Welcome!
Hello, Michael!
Hello, Tracy!
Hello, Sarah!
Mavericks:socket michael$
```

需要注意的是，客户端程序运行完毕就退出了，而服务器程序会永远运行下去，必须按Ctrl+C退出程序。

小结

用TCP协议进行Socket编程在Python中十分简单，对于客户端，要主动连接服务器的IP和指定端口，对于服务器，要首先监听指定端口，然后，对每一个新的连接，创建一个线程或进程来处理。通常，服务器程序会无限运行下去。

同一个端口，被一个Socket绑定了以后，就不能被别的Socket绑定了。

源码参考: <https://github.com/michaelliao/learn-python/tree/master/socket>

UDP编程

572次阅读

TCP是建立可靠连接，并且通信双方都可以以流的形式发送数据。相对TCP，UDP则是面向无连接的协议。

使用UDP协议时，不需要建立连接，只需要知道对方的IP地址和端口号，就可以直接发数据包。但是，能不能到达就不知道了。

虽然用UDP传输数据不可靠，但它的优点是和TCP比，速度快，对于不要求可靠到达的数据，就可以使用UDP协议。

我们来看看如何通过UDP协议传输数据。和TCP类似，使用UDP的通信双方也分为客户端和服务端。服务端首先需要绑定端口：

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# 绑定端口:
s.bind(('127.0.0.1', 9999))
```

创建Socket时，SOCK_DGRAM指定了这个Socket的类型是UDP。绑定端口和TCP一样，但是不需要调用listen()方法，而是直接接收来自任何客户端的数据：

```
print 'Bind UDP on 9999...'
while True:
    # 接收数据:
    data, addr = s.recvfrom(1024)
    print 'Received from %s:%s.' % addr
    s.sendto('Hello, %s!' % data, addr)
```

recvfrom()方法返回数据和客户端的地址与端口，这样，服务端收到数据后，直接调用sendto()就可以把数据用UDP发给客户端。

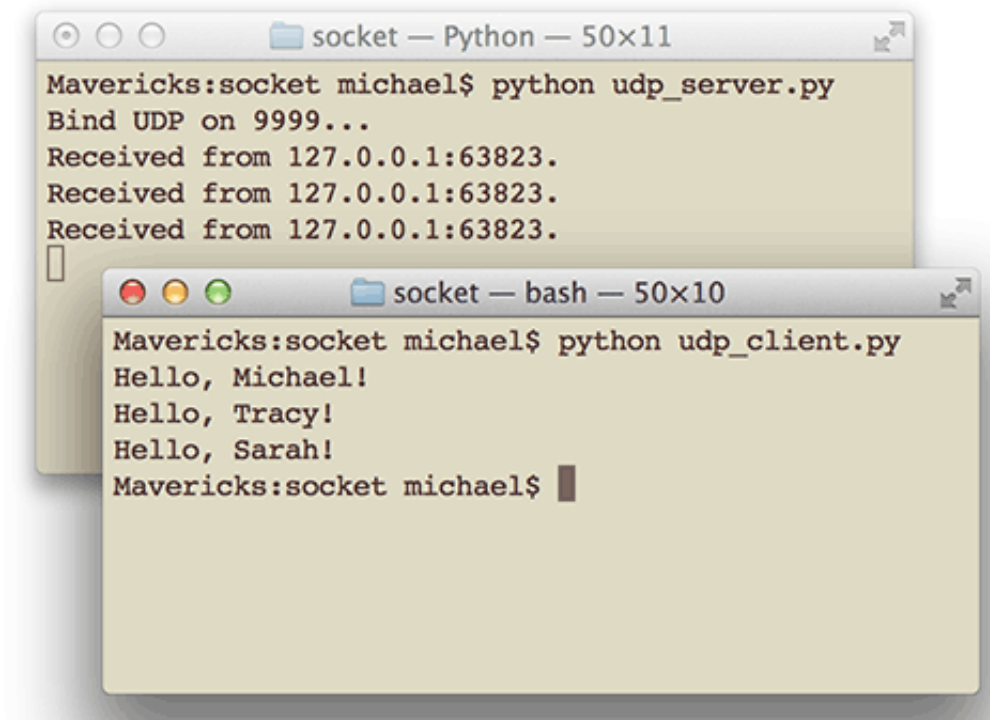
注意这里省掉了多线程，因为这个例子很简单。

客户端使用UDP时，首先仍然创建基于UDP的Socket，然后，不需要调用connect()，直接通过sendto()给服务端发数据：

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
for data in ['Michael', 'Tracy', 'Sarah']:
    # 发送数据:
    s.sendto(data, ('127.0.0.1', 9999))
    # 接收数据:
    print s.recv(1024)
s.close()
```

从服务端接收数据仍然调用recv()方法。

仍然用两个命令行分别启动服务端和客户端测试，结果如下：



The image shows two overlapping terminal windows. The top window, titled 'socket — Python — 50x11', displays the output of a Python script 'udp_server.py'. It shows the server binding to port 9999 and receiving three messages from 127.0.0.1:63823. The bottom window, titled 'socket — bash — 50x10', shows the output of a Python script 'udp_client.py' which sends three messages: 'Hello, Michael!', 'Hello, Tracy!', and 'Hello, Sarah!' to the server. The prompt in the bottom window is 'Mavericks:socket michael\$'.

```
Mavericks:socket michael$ python udp_server.py
Bind UDP on 9999...
Received from 127.0.0.1:63823.
Received from 127.0.0.1:63823.
Received from 127.0.0.1:63823.
█

Mavericks:socket michael$ python udp_client.py
Hello, Michael!
Hello, Tracy!
Hello, Sarah!
Mavericks:socket michael$ █
```

小结

UDP的使用与TCP类似，但是不需要建立连接。此外，服务器绑定UDP端口和TCP端口互不冲突，也就是说，UDP的9999端口与TCP的9999端口可以各自绑定。

源码参考：<https://github.com/michaelliao/learn-python/tree/master/socket>

电子邮件

473次阅读

Email的历史比Web还要久远，直到现在，Email也是互联网上应用非常广泛的服务。

几乎所有的编程语言都支持发送和接收电子邮件，但是，先等等，在我们开始编写代码之前，有必要搞清楚电子邮件是如何在互联网上运作的。

我们来看看传统邮件是如何运作的。假设你现在在北京，要给一个香港的朋友发一封信，怎么做呢？

首先你得写好信，装进信封，写上地址，贴上邮票，然后就近找个邮局，把信仍进去。

信件会从就近的小邮局转运到大邮局，再从大邮局往别的城市发，比如先发到天津，再走海运到达香港，也可能走京九线到香港，但是你不用关心具体路线，你只需要知道一件事，就是信件走得很慢，至少要几天时间。

信件到达香港的某个邮局，也不会直接送到朋友的家里，因为邮局的叔叔是很聪明的，他怕你的朋友不在家，一趟一趟地白跑，所以，信件会投递到你的朋友的邮箱里，邮箱可能在公寓的一层，或者家门口，直到你的朋友回家的时候检查邮箱，发现信件后，就可以取到邮件了。

电子邮件的流程基本上也是按上面的方式运作的，只不过速度不是按天算，而是按秒算。

现在我们回到电子邮件，假设我们自己的电子邮件地址是me@163.com，对方的电子邮件地址是friend@sina.com（注意地址都是虚构的哈），现在我们用Outlook或者Foxmail之类的软件写好邮件，填上对方的Email地址，点“发送”，电子邮件就发出去了。这些电子邮件软件被称为MUA：Mail User Agent——邮件用户代理。

Email从MUA发出去，不是直接到达对方电脑，而是发到MTA：Mail Transfer Agent——邮件传输代理，就是那些Email服务提供商，比如网易、新浪等等。由于我们自己的电子邮件是163.com，所以，Email首先被投递到网易提供的MTA，再由网易的MTA发到对方服务商，也就是新浪的MTA。这个过程中间可能还会经过别的MTA，但是我们不关心具体路线，我们只关心速度。

Email到达新浪的MTA后，由于对方使用的是@sina.com的邮箱，因此，新浪的MTA会把Email投递到邮件的最终目的地MDA：Mail Delivery Agent——邮件投递代理。Email到达MDA后，就静静地躺在新浪的某个服务器上，存放在某个文件或特殊的数据库里，我们将这个长期保存邮件的地方称之为电子邮箱。

同普通邮件类似，Email不会直接到达对方的电脑，因为对方电脑不一定开机，开机也不一定联网。对方要取到邮件，必须通过MUA从MDA上把邮件取到自己的电脑上。

所以，一封电子邮件的旅程就是：

发件人 -> MUA -> MTA -> MTA -> 若干个MTA -> MDA <- MUA <- 收件人

有了上述基本概念，要编写程序来发送和接收邮件，本质上就是：

1. 编写MUA把邮件发到MTA；
2. 编写MUA从MDA上收邮件。

发邮件时，MUA和MTA使用的协议就是SMTP：Simple Mail Transfer Protocol，后面的MTA到另一个MTA也是用SMTP协议。

收邮件时，MUA和MDA使用的协议有两种：POP：Post Office Protocol，目前版本是3，俗称POP3；IMAP：Internet Message Access Protocol，目前版本是4，优点是不但能取邮件，还可以直接操作MDA上存储的邮件，比如从收件箱移到垃圾箱，等等。

邮件客户端软件在发邮件时，会让你先配置SMTP服务器，也就是你要发到哪个MTA上。假设你正在使用163的邮箱，你就不能直接发到新浪的MTA上，因为它只服务新浪的用户，所以，你得填163提供的SMTP服务器地址：smtp.163.com，为了证明你是163的用户，SMTP服务器还要求你填写邮箱地址和邮箱口令，这样，MUA才能正常地把Email通过SMTP协议发送到MTA。

类似的，从MDA收邮件时，MDA服务器也要求验证你的邮箱口令，确保不会有人冒充你收取你的邮件，所以，Outlook之类的邮件客户端会要求你填写POP3或IMAP服务器地址、邮箱地址和口令，这样，MUA才能顺利地通过POP或IMAP协议从MDA取到邮件。

在使用Python收发邮件前，请先准备好至少两个电子邮件，如xxx@163.com，xxx@sina.com，xxx@qq.com等，注意两个邮箱不要用同一家邮件服务商。

SMTP发送邮件

725次阅读

SMTP是发送邮件的协议，Python内置对SMTP的支持，可以发送纯文本邮件、HTML邮件以及带附件的邮件。

Python对SMTP支持有smtplib和email两个模块，email负责构造邮件，smtplib负责发送邮件。

首先，我们来构造一个最简单的纯文本邮件：

```
from email.mime.text import MIMEText
msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')
```

注意到构造MIMEText对象时，第一个参数就是邮件正文，第二个参数是MIME的subtype，传入'plain'，最终的MIME就是'text/plain'，最后一定要用utf-8编码保证多语言兼容性。

然后，通过SMTP发出去：

```
# 输入Email地址和口令：
from_addr = raw_input('From: ')
password = raw_input('Password: ')
# 输入SMTP服务器地址：
smtp_server = raw_input('SMTP server: ')
# 输入收件人地址：
to_addr = raw_input('To: ')

import smtplib
server = smtplib.SMTP(smtp_server, 25) # SMTP协议默认端口是25
server.set_debuglevel(1)
server.login(from_addr, password)
server.sendmail(from_addr, [to_addr], msg.as_string())
server.quit()
```

我们用set_debuglevel(1)就可以打印出和SMTP服务器交互的所有信息。SMTP协议就是简单的文本命令和响应。login()方法用来登录SMTP服务器，sendmail()方法就是发邮件，由于可以一次发给多个人，所以传入一个list，邮件正文是一个str，as_string()把MIMEText对象变成str。

如果一切顺利，就可以在收件人信箱中收到我们刚发送的Email：



hello, send by Python...

仔细观察，发现如下问题：

1. 邮件没有主题；
2. 收件人的名字没有显示为友好的名字，比如Mr Green <green@example.com>;
3. 明明收到了邮件，却提示不在收件人中。

这是因为邮件主题、如何显示发件人、收件人等信息并不是通过SMTP协议发给MTA，而是包含

在发给MTA的文本中的，所以，我们必须把From、To和Subject添加到MIMEText中，才是一封完整的邮件：

```
# -*- coding: utf-8 -*-

from email import encoders
from email.header import Header
from email.mime.text import MIMEText
from email.utils import parseaddr, formataddr
import smtplib

def _format_addr(s):
    name, addr = parseaddr(s)
    return formataddr(( \
        Header(name, 'utf-8').encode(), \
        addr.encode('utf-8') if isinstance(addr, unicode) else addr))

from_addr = raw_input('From: ')
password = raw_input('Password: ')
to_addr = raw_input('To: ')
smtp_server = raw_input('SMTP server: ')

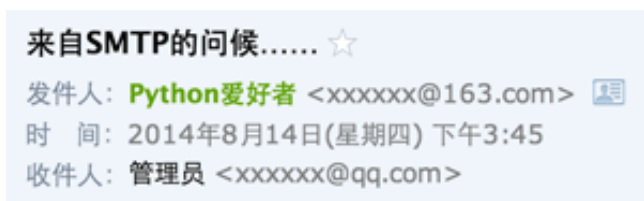
msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')
msg['From'] = _format_addr(u'Python爱好者 <%s>' % from_addr)
msg['To'] = _format_addr(u'管理员 <%s>' % to_addr)
msg['Subject'] = Header(u'来自SMTP的问候.....', 'utf-8').encode()

server = smtplib.SMTP(smtp_server, 25)
server.set_debuglevel(1)
server.login(from_addr, password)
server.sendmail(from_addr, [to_addr], msg.as_string())
server.quit()
```

我们编写了一个函数_format_addr()来格式化一个邮件地址。注意不能简单地传入name <addr@example.com>，因为如果包含中文，需要通过Header对象进行编码。

msg['To']接收的是字符串而不是list，如果有多个邮件地址，用,分隔即可。

再发送一遍邮件，就可以在收件人邮箱中看到正确的标题、发件人和收件人：



hello, send by Python...

你看到的收件人的名字很可能不是我们传入的管理员，因为很多邮件服务商在显示邮件时，会把收件人名字自动替换为用户注册的名字，但是其他收件人名字的显示不受影响。

如果我们查看Email的原始内容，可以看到如下经过编码的邮件头：

```
From: =?utf-8?b?UH10aG9u54ix5aW96ICF?= <xxxxxx@163.com>
To: =?utf-8?b?566h55CG5ZGY?= <xxxxxx@qq.com>
Subject: =?utf-8?b?5p216IeqU01UU0eah0mXruWAmEKApuKApg==?=
```

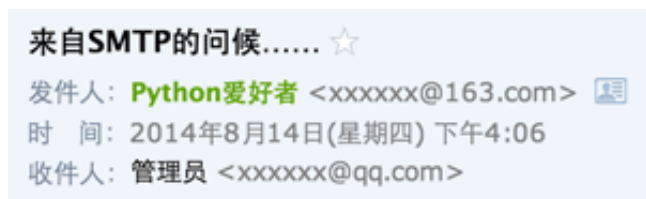
这就是经过Header对象编码的文本，包含utf-8编码信息和Base64编码的文本。如果我们自己来手动构造这样的编码文本，显然比较复杂。

发送HTML邮件

如果我们要发送HTML邮件，而不是普通的纯文本文件怎么办？方法很简单，在构造MIMEText对象时，把HTML字符串传进去，再把第二个参数由plain变为html就可以了：

```
msg = MIMEText('<html><body><h1>Hello</h1>' +
               '<p>send by <a href="http://www.python.org">Python</a>...</p>' +
               '</body></html>', 'html', 'utf-8')
```

再发送一遍邮件，你将看到以HTML显示的邮件：



Hello

send by [Python...](#)

发送附件

如果Email中要加上附件怎么办？带附件的邮件可以看做包含若干部分的邮件：文本和各个附件本身，所以，可以构造一个MIMEMultipart对象代表邮件本身，然后往里面加上一个MIMEText作为邮件正文，再继续往里面加上表示附件的MIMEBase对象即可：


```
# 邮件对象：
msg = MIMEMultipart()
msg['From'] = _format_addr(u'Python爱好者 <%s>' % from_addr)
msg['To'] = _format_addr(u'管理员 <%s>' % to_addr)
msg['Subject'] = Header(u'来自SMTP的问候.....', 'utf-8').encode()

# 邮件正文是MIMEText:
msg.attach(MIMEText('send with file...', 'plain', 'utf-8'))

# 添加附件就是加上一个MIMEBase，从本地读取一个图片：
with open('/Users/michael/Downloads/test.png', 'rb') as f:
    # 设置附件的MIME和文件名，这里是png类型：
    mime = MIMEBase('image', 'png', filename='test.png')
    # 加上必要的头信息：
    mime.add_header('Content-Disposition', 'attachment', filename='test.png')
    mime.add_header('Content-ID', '<0>')
    mime.add_header('X-Attachment-Id', '0')
    # 把附件的内容读进来：
    mime.set_payload(f.read())
    # 用Base64编码：
    encoders.encode_base64(mime)
    # 添加到MIMEMultipart:
    msg.attach(mime)
```


然后，按正常发送流程把msg（注意类型已变为MIMEMultipart）发送出去，就可以收到如下带附件的邮件：

来自SMTP的问候..... ☆


发件人: Python爱好者 <xxxxxx@163.com> 

时 间: 2014年8月14日(星期四) 下午5:08

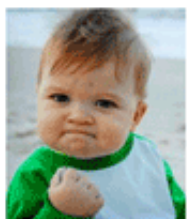
收件人: 管理员 <xxxxxx@qq.com>

附 件: 1 个 ( test.png)

send with file...

 附件(1 个)

普通附件



test.png (80.13K)

[下载](#) [预览](#) [收藏](#) [转存](#) ▼

发送图片

如果要把一个图片嵌入到邮件正文中怎么做？直接在HTML邮件中链接图片地址行不行？答案是，大部分邮件服务商都会自动屏蔽带有外链的图片，因为不知道这些链接是否指向恶意网站。


要把图片嵌入到邮件正文中，我们只需按照发送附件的方式，先把邮件作为附件添加进去，然后，在HTML中通过引用src="cid:0"就可以把附件作为图片嵌入了。如果有多个图片，给它们依次编号，然后引用不同的cid:x即可。

把上面代码加入MIMEMultipart的MIMEText从plain改为html，然后在适当的位置引用图片：

```
msg.attach(MIMEText(' <html><body><h1>Hello</h1>' +  
    '<p></p>' +  
    '</body></html>', 'html', 'utf-8'))
```

再次发送，就可以看到图片直接嵌入到邮件正文的效果：

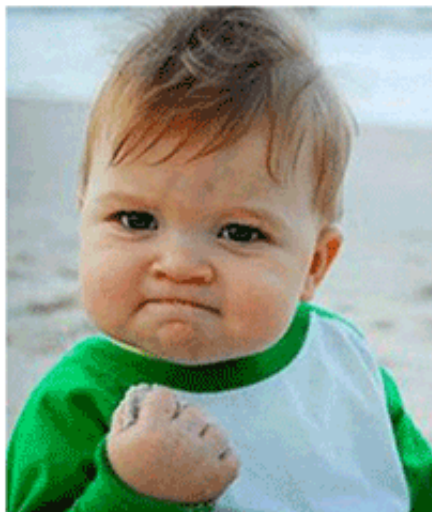
来自SMTP的问候..... ☆

发件人: Python爱好者 <asklxf@163.com> 

时 间: 2014年8月14日(星期四) 下午5:27

收件人: Xuefeng <18224514@qq.com>

Hello



同时支持HTML和Plain格式

如果我们发送HTML邮件，收件人通过浏览器或者Outlook之类的软件是可以正常浏览邮件内容的，但是，如果收件人使用的设备太古老，查看不了HTML邮件怎么办？

办法是在发送HTML的同时再附加一个纯文本，如果收件人无法查看HTML格式的邮件，就可以自动降级查看纯文本邮件。

利用MIMEMultipart就可以组合一个HTML和Plain，要注意指定subtype是alternative：

```
msg = MIMEMultipart('alternative')
msg['From'] = ...
msg['To'] = ...
msg['Subject'] = ...

msg.attach(MIMEText('hello', 'plain', 'utf-8'))
msg.attach(MIMEText('<html><body><h1>Hello</h1></body></html>', 'html', 'utf-8'))
# 正常发送msg对象...
```

加密SMTP

使用标准的25端口连接SMTP服务器时，使用的是明文传输，发送邮件的整个过程可能会被窃听。要更安全地发送邮件，可以加密SMTP会话，实际上就是先创建SSL安全连接，然后再使用SMTP协议发送邮件。

某些邮件服务商，例如Gmail，提供的SMTP服务必须要加密传输。我们来看看如何通过Gmail提供的安全SMTP发送邮件。

必须知道，Gmail的SMTP端口是587，因此，修改代码如下：


```
smtp_server = 'smtp.gmail.com'
smtp_port = 587
server = smtplib.SMTP(smtp_server, smtp_port)
server.starttls()
# 剩下的代码和前面的一模一样:
server.set_debuglevel(1)
...
```

只需要在创建SMTP对象后，立刻调用`starttls()`方法，就创建了安全连接。后面的代码和前面的发送邮件代码完全一样。

如果因为网络问题无法连接Gmail的SMTP服务器，请相信我们的代码是没有问题的，你需要对你的网络设置做必要的调整。

小结

使用Python的`smtplib`发送邮件十分简单，只要掌握了各种邮件类型的构造方法，正确设置好邮件头，就可以顺利发出。

构造一个邮件对象就是一个`Message`对象，如果构造一个`MIMEText`对象，就表示一个文本邮件对象，如果构造一个`MIMEImage`对象，就表示一个作为附件的图片，要把多个对象组合起来，就用`MIMEMultipart`对象，而`MIMEBase`可以表示任何对象。它们的继承关系如下：

```
Message
+- MIMEBase
  +- MIMEMultipart
  +- MIMENonMultipart
    +- MIMEMessage
    +- MIMEText
    +- MIMEImage
```

这种嵌套关系就可以构造出任意复杂的邮件。你可以通过[email.mime文档](#)查看它们所在的包以及详细的用法。

源码参考：

<https://github.com/michaelliao/learn-python/tree/master/email>

POP3收取邮件

1619次阅读

SMTP用于发送邮件，如果要收取邮件呢？

收取邮件就是编写一个MUA作为客户端，从MDA把邮件获取到用户的电脑或者手机上。收取邮件最常用的协议是POP协议，目前版本号是3，俗称POP3。

Python内置一个poplib模块，实现了POP3协议，可以直接用来收邮件。

注意到POP3协议收取的不是一个已经可以阅读的邮件本身，而是邮件的原始文本，这和SMTP协议很像，SMTP发送的也是经过编码后的一大段文本。

要把POP3收取的文本变成可以阅读的邮件，还需要用email模块提供的各种类来解析原始文本，变成可阅读的邮件对象。

所以，收取邮件分两步：

第一步：用poplib把邮件的原始文本下载到本地；

第二部：用email解析原始文本，还原为邮件对象。

通过POP3下载邮件

POP3协议本身很简单，以下面的代码为例，我们来获取最新的一封邮件内容：

```
import poplib

# 输入邮件地址，口令和POP3服务器地址：
email = raw_input('Email: ')
password = raw_input('Password: ')
pop3_server = raw_input('POP3 server: ')

# 连接到POP3服务器：
server = poplib.POP3(pop3_server)
# 可以打开或关闭调试信息：
# server.set_debuglevel(1)
# 可选：打印POP3服务器的欢迎文字：
print(server.getwelcome())
# 身份认证：
server.user(email)
server.pass_(password)
# stat() 返回邮件数量和占用空间：
print('Messages: %s. Size: %s' % server.stat())
# list() 返回所有邮件的编号：
resp, mails, octets = server.list()
# 可以查看返回的列表类似['1 82923', '2 2184', ...]
print(mails)
# 获取最新一封邮件，注意索引号从1开始：
index = len(mails)
resp, lines, octets = server.retr(index)
# lines存储了邮件的原始文本的每一行，
# 可以获得整个邮件的原始文本：
msg_content = '\r\n'.join(lines)
# 稍后解析出邮件：
msg = Parser().parsestr(msg_content)
# 可以根据邮件索引号直接从服务器删除邮件：
# server.dele(index)
```

```
# 关闭连接:
server.quit()
```

用POP3获取邮件其实很简单, 要获取所有邮件, 只需要循环使用`retr()`把每一封邮件内容拿到即可。真正麻烦的是把邮件的原始内容解析为可以阅读的邮件对象。

解析邮件

解析邮件的过程和上一节构造邮件正好相反, 因此, 先导入必要的模块:

```
import email
from email.parser import Parser
from email.header import decode_header
from email.utils import parseaddr
```

只需要一行代码就可以把邮件内容解析为`Message`对象:

```
msg = Parser().parsestr(msg_content)
```

但是这个`Message`对象本身可能是一个`MIMEMultipart`对象, 即包含嵌套的其他`MIMEBase`对象, 嵌套可能还不止一层。

所以我们要递归地打印出`Message`对象的层次结构:

```
# indent用于缩进显示:
def print_info(msg, indent=0):
    if indent == 0:
        # 邮件的From, To, Subject存在于根对象上:
        for header in ['From', 'To', 'Subject']:
            value = msg.get(header, '')
            if value:
                if header == 'Subject':
                    # 需要解码Subject字符串:
                    value = decode_str(value)
                else:
                    # 需要解码Email地址:
                    hdr, addr = parseaddr(value)
                    name = decode_str(hdr)
                    value = u'%s <%s>' % (name, addr)
                print('%s%s: %s' % (' ' * indent, header, value))
    if (msg.is_multipart()):
        # 如果邮件对象是一个MIMEMultipart,
        # get_payload()返回列表, 包含所有的子对象:
        parts = msg.get_payload()
        for n, part in enumerate(parts):
            print('%spart %s' % (' ' * indent, n))
            print('%s-----' % (' ' * indent))
            # 递归打印每一个子对象:
            print_info(part, indent + 1)
    else:
        # 邮件对象不是一个MIMEMultipart,
        # 就根据content_type判断:
        content_type = msg.get_content_type()
        if content_type == 'text/plain' or content_type == 'text/html':
            # 纯文本或HTML内容:
            content = msg.get_payload(decode=True)
            # 要检测文本编码:
            charset = guess_charset(msg)
            if charset:
                content = content.decode(charset)
            print('%sText: %s' % (' ' * indent, content + '...'))
        else:
```

```
# 不是文本, 作为附件处理:
print('%sAttachment: %s' % ( ' ' * indent, content_type))
```

邮件的Subject或者Email中包含的名字都是经过编码后的str, 要正常显示, 就必须decode:

```
def decode_str(s):
    value, charset = decode_header(s)[0]
    if charset:
        value = value.decode(charset)
    return value
```

decode_header() 返回一个list, 因为像Cc、Bcc这样的字段可能包含多个邮件地址, 所以解析出来的会有多个元素。上面的代码我们偷了个懒, 只取了第一个元素。

文本邮件的内容也是str, 还需要检测编码, 否则, 非UTF-8编码的邮件都无法正常显示:

```
def guess_charset(msg):
    # 先从msg对象获取编码:
    charset = msg.get_charset()
    if charset is None:
        # 如果获取不到, 再从Content-Type字段获取:
        content_type = msg.get('Content-Type', '').lower()
        pos = content_type.find('charset=')
        if pos >= 0:
            charset = content_type[pos + 8:].strip()
    return charset
```

把上面的代码整理好, 我们就可以来试试收取一封邮件。先往自己的邮箱发一封邮件, 然后用浏览器登录邮箱, 看看邮件收到没, 如果收到了, 我们就来用Python程序把它收到本地:



Python可以[使用POP3收取邮件](#).....

运行程序, 结果如下:

```
+OK Welcome to coremail Mail Pop3 Server (163coms[...])
Messages: 126. Size: 27228317
```

```
From: Test <xxxxxxx@qq.com>
To: Python爱好者 <xxxxxxx@163.com>
Subject: 用POP3收取邮件
part 0
```

```
-----
part 0
```

```
-----
Text: Python可以使用POP3收取邮件.....
part 1
```

```
-----
Text: Python可以<a href="...">使用POP3</a>收取邮件.....
part 1
```

```
-----
Attachment: application/octet-stream
```

我们从打印的结构可以看出，这封邮件是一个MIMEMultipart，它包含两部分：第一部分又是一个MIMEMultipart，第二部分是一个附件。而内嵌的MIMEMultipart是一个alternative类型，它包含一个纯文本格式的MIMEText和一个HTML格式的MIMEText。

小结

用Python的poplib模块收取邮件分两步：第一步是用POP3协议把邮件获取到本地，第二步是用email模块把原始邮件解析为Message对象，然后，用适当的形式把邮件内容展示给用户即可。

源码参考：

<https://github.com/michaelliao/learn-python/tree/master/email>

访问数据库

598次阅读

程序运行的时候，数据都是在内存中的。当程序终止的时候，通常都需要将数据保存到磁盘上，无论是保存到本地磁盘，还是通过网络保存到服务器上，最终都会将数据写入磁盘文件。

而如何定义数据的存储格式就是一个大问题。如果我们自己来定义存储格式，比如保存一个班级所有学生的成绩单：

名字	成绩
Michael	99
Bob	85
Bart	59
Lisa	87

你可以用一个文本文件保存，一行保存一个学生，用,隔开：

```
Michael,99
Bob,85
Bart,59
Lisa,87
```

你还可以用JSON格式保存，也是文本文件：

```
[
  {"name":"Michael","score":99},
  {"name":"Bob","score":85},
  {"name":"Bart","score":59},
  {"name":"Lisa","score":87}
]
```

你还可以定义各种保存格式，但是问题来了：

存储和读取需要自己实现，JSON还是标准，自己定义的格式就各式各样的；

不能做快速查询，只有把数据全部读到内存中才能自己遍历，但有时候数据的大小远远超过了内存（比如蓝光电影，40GB的数据），根本无法全部读入内存。

为了便于程序保存和读取数据，而且，能直接通过条件快速查询到指定的数据，就出现了数据库（Database）这种专门用于集中存储和查询的软件。

数据库软件诞生的历史非常久远，早在1950年数据库就诞生了。经历了网状数据库，层次数据库，我们现在广泛使用的关系数据库是20世纪70年代基于关系模型的基础上诞生的。

关系模型有一套复杂的数学理论，但是从概念上是十分容易理解的。举个学校的例子：

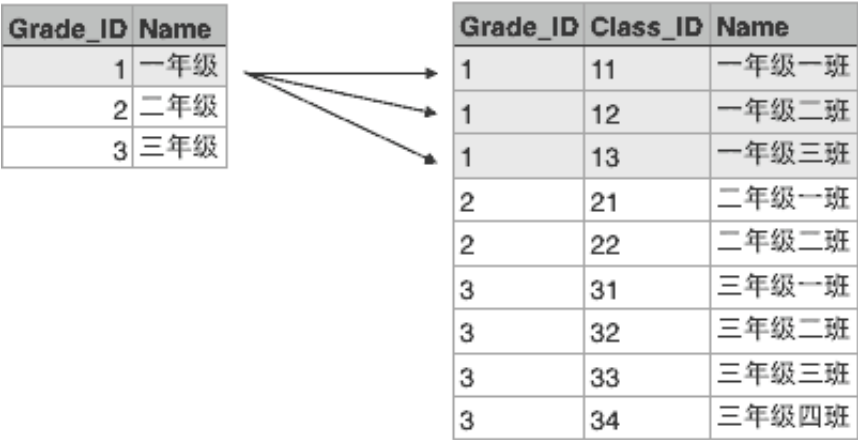
假设某个XX省YY市ZZ县第一实验小学有3个年级，要表示出这3个年级，可以在Excel中用一个表格画出来：

Grade_ID	Name
1	一年级
2	二年级
3	三年级

每个年级又有若干个班级，要把所有班级表示出来，可以在Excel中再画一个表格：

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

这两个表格有个映射关系，就是根据Grade_ID可以在班级表中查找到对应的所有班级：



也就是Grade表的每一行对应Class表的多行，在关系数据库中，这种基于表（Table）的一对多的关系就是关系数据库的基础。

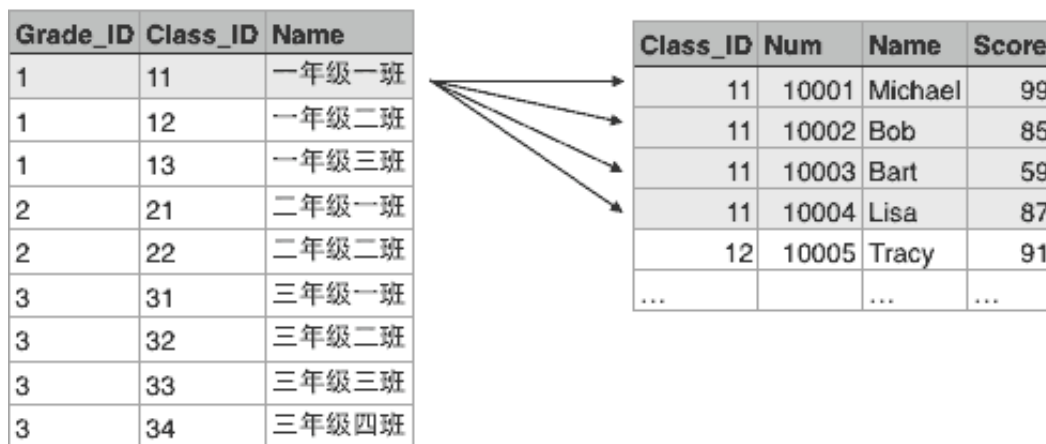
根据某个年级的ID就可以查找所有班级的行，这种查询语句在关系数据库中称为SQL语句，可以写成：

```
SELECT * FROM classes WHERE grade_id = '1';
```

结果也是一个表：

grade_id	class_id	name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班

类似的，Class表的一行记录又可以关联到Student表的多行记录：



由于本教程不涉及到关系数据库的详细内容，如果你想从零学习关系数据库和基本的SQL语句，推荐Coursera课程：

英文：<https://www.coursera.org/course/db>

中文：<http://c.open.163.com/coursera/courseIntro.htm?cid=12>

NoSQL

你也许还听说过NoSQL数据库，很多NoSQL宣传其速度和规模远远超过关系数据库，所以很多同学觉得有了NoSQL是否就不需要SQL了呢？千万不要被他们忽悠了，连SQL都不明白怎么可能搞明白NoSQL呢？

数据库类别

既然我们要使用关系数据库，就必须选择一个关系数据库。目前广泛使用的关系数据库也就这么几种：

付费的商用数据库：

- Oracle，典型的高富帅；
- SQL Server，微软自家产品，Windows定制专款；
- DB2，IBM的产品，听起来挺高端；
- Sybase，曾经跟微软是好基友，后来关系破裂，现在家境惨淡。

这些数据库都是不开源而且付费的，最大的好处是花了钱出了问题可以找厂家解决，不过在Web的世界里，常常需要部署成千上万的数据库服务器，当然不能把大把大把的银子扔给厂家，所以，无论是Google、Facebook，还是国内的BAT，无一例外都选择了免费的开源数据库：

- MySQL，大家都在用，一般错不了；
- PostgreSQL，学术气息有点重，其实挺不错，但知名度没有MySQL高；
- sqlite，嵌入式数据库，适合桌面和移动应用。

作为Python开发工程师，选择哪个免费数据库呢？当然是MySQL。因为MySQL普及率最高，出了错，可以很容易找到解决方法。而且，围绕MySQL有一大堆监控和运维的工具，安装和使用很方便。

为了能继续后面的学习，你需要从MySQL官方网站下载并安装[MySQL Community Server 5.6](#)，这个版本是免费的，其他高级版本是要收钱的（请放心，收钱的功能我们用不上）。

使用SQLite

765次阅读

SQLite是一种嵌入式数据库，它的数据库就是一个文件。由于SQLite本身是C写的，而且体积很小，所以，经常被集成到各种应用程序中，甚至在iOS和Android的App中都可以集成。

Python就内置了SQLite3，所以，在Python中使用SQLite，不需要安装任何东西，直接使用。

在使用SQLite前，我们先要搞清楚几个概念：

表是数据库中存放关系数据的集合，一个数据库里面通常都包含多个表，比如学生的表，班级的表，学校的表，等等。表和表之间通过外键关联。

要操作关系数据库，首先需要连接到数据库，一个数据库连接称为Connection；

连接到数据库后，需要打开游标，称之为Cursor，通过Cursor执行SQL语句，然后，获得执行结果。

Python定义了一套操作数据库的API接口，任何数据库要连接到Python，只需要提供符合Python标准的数据库驱动即可。

由于SQLite的驱动内置在Python标准库中，所以我们可以直接来操作SQLite数据库。

我们在Python交互式命令行实践一下：

```
# 导入SQLite驱动:
>>> import sqlite3
# 连接到SQLite数据库
# 数据库文件是test.db
# 如果文件不存在，会自动在当前目录创建:
>>> conn = sqlite3.connect('test.db')
# 创建一个Cursor:
>>> cursor = conn.cursor()
# 执行一条SQL语句，创建user表:
>>> cursor.execute('create table user (id varchar(20) primary key, name varchar(20))')
<sqlite3.Cursor object at 0x10f8aa260>
# 继续执行一条SQL语句，插入一条记录:
>>> cursor.execute('insert into user (id, name) values (\`1\`, \`Michael\` )')
<sqlite3.Cursor object at 0x10f8aa260>
# 通过rowcount获得插入的行数:
>>> cursor.rowcount
1
# 关闭Cursor:
>>> cursor.close()
# 提交事务:
>>> conn.commit()
# 关闭Connection:
>>> conn.close()
```

我们再试试查询记录：

```
>>> conn = sqlite3.connect('test.db')
>>> cursor = conn.cursor()
# 执行查询语句:
>>> cursor.execute('select * from user where id=?', '1')
<sqlite3.Cursor object at 0x10f8aa340>
# 获得查询结果集:
>>> values = cursor.fetchall()
```

```
>>> values
[(u'1', u'Michael')]
>>> cursor.close()
>>> conn.close()
```

使用Python的DB-API时，只要搞清楚Connection和Cursor对象，打开后一定记得关闭，就可以放心地使用。

使用Cursor对象执行insert, update, delete语句时，执行结果由rowcount返回影响的行数，就可以拿到执行结果。

使用Cursor对象执行select语句时，通过fetchall()可以拿到结果集。结果集是一个list，每个元素都是一个tuple，对应一行记录。

如果SQL语句带有参数，那么需要把参数按照位置传递给execute()方法，有几个?占位符就必须对应几个参数，例如：

```
cursor.execute('select * from user where id=?', '1')
```

SQLite支持常见的标准SQL语句以及几种常见的数据类型。具体文档请参阅SQLite官方网站。

小结

在Python中操作数据库时，要先导入数据库对应的驱动，然后，通过Connection对象和Cursor对象操作数据。

要确保打开的Connection对象和Cursor对象都正确地被关闭，否则，资源就会泄露。

如何才能确保出错的情况下也关闭掉Connection对象和Cursor对象呢？请回忆try...catch...finally...的用法。

使用MySQL

1079次阅读

MySQL是Web世界中使用的最广泛的数据库服务器。SQLite的特点是轻量级、可嵌入，但不能承受高并发访问，适合桌面和移动应用。而MySQL是为服务器端设计的数据库，能承受高并发访问，同时占用的内存也远远大于SQLite。

此外，MySQL内部有多种数据库引擎，最常用的引擎是支持数据库事务的InnoDB。

安装MySQL

可以直接从MySQL官方网站下载最新的[Community Server 5.6.x](#)版本。MySQL是跨平台的，选择对应的平台下载安装文件，安装即可。

安装时，MySQL会提示输入root用户的口令，请务必记清楚。如果怕记不住，就把口令设置为password。

在Windows上，安装时请选择UTF-8编码，以便正确地处理中文。

在Mac或Linux上，需要编辑MySQL的配置文件，把数据库默认的编码全部改为UTF-8。MySQL的配置文件默认存放在/etc/my.cnf或者/etc/mysql/my.cnf：

```
[client]
default-character-set = utf8

[mysqld]
default-storage-engine = INNODB
character-set-server = utf8
collation-server = utf8_general_ci
```

重启MySQL后，可以通过MySQL的客户端命令行检查编码：

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor...
...
```

```
mysql> show variables like '%char%';
```

Variable_name	Value
character_set_client	utf8
character_set_connection	utf8
character_set_database	utf8
character_set_filesystem	binary
character_set_results	utf8
character_set_server	utf8
character_set_system	utf8
character_sets_dir	/usr/local/mysql-5.1.65-osx10.6-x86_64/share/charsets/

```
8 rows in set (0.00 sec)
```

看到utf8字样就表示编码设置正确。

安装MySQL驱动

由于MySQL服务器以独立的进程运行，并通过网络对外服务，所以，需要支持Python的MySQL驱

动来连接到MySQL服务器。

目前，有两个MySQL驱动：

- `mysql-connector-python`：是MySQL官方的纯Python驱动；
- `MySQL-python`：是封装了MySQL C驱动的Python驱动。

可以把两个都装上，使用的时候再决定用哪个：

```
$ easy_install mysql-connector-python
$ easy_install MySQL-python
```

我们以`mysql-connector-python`为例，演示如何连接到MySQL服务器的`test`数据库：

```
# 导入MySQL驱动：
>>> import mysql.connector
# 注意把password设为你的root口令：
>>> conn = mysql.connector.connect(user='root', password='password', database='test', use_unicode=True)
>>> cursor = conn.cursor()
# 创建user表：
>>> cursor.execute('create table user (id varchar(20) primary key, name varchar(20))')
# 插入一行记录，注意MySQL的占位符是%s：
>>> cursor.execute('insert into user (id, name) values (%s, %s)', ['1', 'Michael'])
>>> cursor.rowcount
1
# 提交事务：
>>> conn.commit()
>>> cursor.close()
# 运行查询：
>>> cursor = conn.cursor()
>>> cursor.execute('select * from user where id = %s', '1')
>>> values = cursor.fetchall()
>>> values
[(u'1', u'Michael')]
# 关闭Cursor和Connection：
>>> cursor.close()
True
>>> conn.close()
```

由于Python的DB-API定义都是通用的，所以，操作MySQL的数据库代码和SQLite类似。

小结

- MySQL的SQL占位符是%s；
 - 通常我们在连接MySQL时传入`use_unicode=True`，让MySQL的DB-API始终返回Unicode。
-

使用SQLAlchemy

502次阅读

数据库表是一个二维表，包含多行多列。把一个表的内容用Python的数据结构表示出来的话，可以用一个list表示多行，list的每一个元素是tuple，表示一行记录，比如，包含id和name的user表：

```
[
    ('1', 'Michael'),
    ('2', 'Bob'),
    ('3', 'Adam')
]
```

Python的DB-API返回的数据结构就是像上面这样表示的。

但是用tuple表示一行很难看出表的结构。如果把一个tuple用class实例来表示，就可以更容易地看出表的结构来：

```
class User(object):
    def __init__(self, id, name):
        self.id = id
        self.name = name

[
    User('1', 'Michael'),
    User('2', 'Bob'),
    User('3', 'Adam')
]
```

这就是传说中的ORM技术：Object-Relational Mapping，把关系数据库的表结构映射到对象上。是不是很简单？

但是由谁来做这个转换呢？所以ORM框架应运而生。

在Python中，最有名的ORM框架是SQLAlchemy。我们来看看SQLAlchemy的用法。

首先通过easy_install或者pip安装SQLAlchemy：

```
$ easy_install sqlalchemy
```

然后，利用上次我们在MySQL的test数据库中创建的用户表，用SQLAlchemy来试试：

第一步，导入SQLAlchemy，并初始化DBSession：

```
# 导入：
from sqlalchemy import Column, String, create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

# 创建对象的基类：
Base = declarative_base()

# 定义User对象：
class User(Base):
    # 表的名字：
    __tablename__ = 'user'

    # 表的结构：
```

```

id = Column(String(20), primary_key=True)
name = Column(String(20))

# 初始化数据库连接:
engine = create_engine('mysql+mysqlconnector://root:password@localhost:3306/test')
# 创建DBSession类型:
DBSession = sessionmaker(bind=engine)

```

以上代码完成SQLAlchemy的初始化和具体每个表的class定义。如果有多个表，就继续定义其他class，例如School:

```

class School(Base):
    __tablename__ = 'school'
    id = ...
    name = ...

```

create_engine()用来初始化数据库连接。SQLAlchemy用一个字符串表示连接信息:

'数据库类型+数据库驱动名称://用户名:口令@机器地址:端口号/数据库名'

你只需要根据需要替换掉用户名、口令等信息即可。

下面，我们看看如何向数据库表中添加一行记录。

由于有了ORM，我们向数据库表中添加一行记录，可以视为添加一个User对象:

```

# 创建session对象:
session = DBSession()
# 创建新用户对象:
new_user = User(id='5', name='Bob')
# 添加到session:
session.add(new_user)
# 提交即保存到数据库:
session.commit()
# 关闭session:
session.close()

```

可见，关键是获取session，然后把对象添加到session，最后提交并关闭。Session对象可视为当前数据库连接。

如何从数据库表中查询数据呢？有了ORM，查询出来的可以不再是tuple，而是User对象。SQLAlchemy提供的查询接口如下:

```

# 创建Session:
session = DBSession()
# 创建Query查询，filter是where条件，最后调用one()返回唯一行，如果调用all()则返回所有行:
user = session.query(User).filter(User.id=='5').one()
# 打印类型和对象的name属性:
print 'type:', type(user)
print 'name:', user.name
# 关闭Session:
session.close()

```

运行结果如下:

```

type: <class '__main__.User'>
name: Bob

```

可见，ORM就是把数据库表的行与相应的对象建立关联，互相转换。

由于关系数据库的多个表还可以用外键实现一对多、多对多等关联，相应地，ORM框架也可以

提供两个对象之间的一对多、多对多等功能。

例如，如果一个User拥有多个Book，就可以定义一对多关系如下：

```
class User(Base):
    __tablename__ = 'user'

    id = Column(String(20), primary_key=True)
    name = Column(String(20))
    # 一对多:
    books = relationship('Book')

class Book(Base):
    __tablename__ = 'book'

    id = Column(String(20), primary_key=True)
    name = Column(String(20))
    # “多”的一方的book表是通过外键关联到user表的:
    user_id = Column(String(20), ForeignKey('user.id'))
```

当我们查询一个User对象时，该对象的books属性将返回一个包含若干个Book对象的list。

小结

ORM框架的作用就是把数据库表的一行记录与一个对象互相做自动转换。

正确使用ORM的前提是了解关系数据库的原理。
