Python套接字编程

摘要：使用Python基本套接字功能开发网络应用

1预备知识

1.1 关于本指导手册

Python是一种非常流行的面向对象的脚本语言，具有语法简洁的特点，并且拥有非常广泛的开发者基础。它是通用目的语言，能够在各种设定中使用。它在刚开始编程的初学者也非常受欢迎，就是20世纪70年代的BASIC语言那样。

本教程的侧重点是Python语言在网络编程方面的应用。 介绍了Python的基本套接字，除此之外，还提供异步套接字的一些Python的其他类。 我还详细介绍了Python的应用层协议类，展示了如何构建Web客户端，邮件服务器和客户端等等。

我还演示了一个简单的聊天服务器来说明Python套接字在套接字应用方面的强大。

在阅读本文前，你应该对标准BSD套接字APIs有一定的理解，且有GNU/Linux环境下工作的经验。熟悉面向对象编程也是非常有益的。

1.2 依赖条件

本教程和演示中的示例依赖于Python的2.4版本。 您可以从Python官方网站下载此版本（[请参阅参考资料中的链接](https://www.ibm.com/developerworks/linux/tutorials/l-pysocks/#resources)）。 要构建Python解释器，需要GNU C编译器（gcc）和configure / make工具（它们是任何一种标准GNU / Linux发行版的一部分）。

2介绍Python

首先，让我们体验一下Python。

2.1 什么是Python?

Python是一个通用的面向对象的脚本语言，可以应用于各种各样的问题。 它是在20世纪90年代初期在阿姆斯特丹的CWI创建的，其后在Python软件基金会下继续发展到今天。

Python有着不可思议的移植性，几乎可以在所有的操作系统上发现它的身影。

Python是解释型语言，且很容易扩展。你可以通过添加模块扩展Python，Python模块可以包含函数，变量，或者通过C/C++编译的函数类型。

您还可以轻松地将Python嵌入到C或C ++程序中，使您可以使用脚本功能扩展应用程序。

Python最有用的方面之一就是其拥有大量的扩展模块。这些模块不仅提供了诸如字符串或列表处理的标准功能，还提供了用于视频和图像处理，音频处理以及网络的应用层模块。

2.2 体验Python

现在让我来告诉你Python是什么吧！

作为一种解释型语言，使用Python轻松实现想法并快速建立原型软件非常简单。 Python程序可以整体解释，或一行一行解释。

您可以通过先运行Python来测试以下Python代码片段，然后逐个输入一行。 Python被调用后，出现一个提示符（>>>），允许您输入命令。 请注意，缩进在Python中很重要，因此一行中的前面的空格不能被忽略：

例1．尝试一些Python例子

# 打开一个文件, 逐行读取,然后打印出来

for line in open('file.txt'):

print line

# 创建一个新文件，写入内容

file = open("test.txt", "w")

file.write("test line\n")

file.close()

# 创建字典，包含名字和年龄，

family = {'Megan': 13, 'Elise': 8, 'Marc': 6}

# 读取8

family['Elise']

# 移除一个键值对

del family['Elise']

# 创建一个列表和函数，函数对列表中的每个元素求平方。

# 将函数与列表中的元素建立键值对关系。

arr = [1, 2, 3, 4, 5]

def double(x): return x\*x

map(double, arr)

# 通过继承创建一个类，并实例化，且调用它的方法

class Simple:

def \_\_init\_\_(self, name):

self.name = name

def hello(self):

print self.name+" says hi."

class Simple2(Simple):

def goodbye(self):

print self.name+" says goodbye."

me = Simple2("Tim")

me.hello()

me.goodbye()

2.3 为什么使用Python?

学习和使用Python的头号原因是它的受欢迎程度。其用户群的规模以及使用Python构建的应用程序数量不断增加，这是一项有价值的投资。

您可以在几个开发领域找到Python，例如，构建系统实用程序，程序集成，Internet应用程序和快速搭建原型程序等方面。

与其他脚本语言相比，Python也有一些优势。 它的语法简单，概念清晰，易于学习。 使用复杂的数据结构（如列表，字典和元组）时，Python也更容易且更具描述性。 Python也可以扩展语言，反过来也可以被语言扩展。

我发现Python的语法比Perl更具可读性和可维护性，但不如Ruby。Python相对于Ruby的优势在于可用的大量库和模块。 使用这些，您可以使用少量自定义代码来构建功能丰富的程序。

Python使用缩进来标识块范围可能相当烦人，但其简单性往往弥补了这个小缺陷。

现在，我们来深入Python中的套接字编程吧。

3 Python 套接字模块

3.1 基本Python 套接字模块

Python提供了两个基本的套接字模块。 首先，Socket提供了标准的BSD套接字API。 第2个，SocketServer提供了一个以服务器为中心的类，它简化了网络服务器的开发。 它以异步方式执行此操作，您可以在其中提供插件类来执行服务器的特定于应用程序的作业。 表1列出了本节涵盖的类和模块。

表1. Python类和模块

|  |  |
| --- | --- |
| 类/模块 | 描述 |
| Socket | 低级网络接口（根据BSD API） |
| SocketServer | 提供简化网络服务器开发的类 |

让我们来看看这些模块，了解他们可以为您做些什么。

3.2 Socket

Socket模块提供了UNIX程序员最熟悉的基本网络服务（也称为BSD API）。 这个模块拥有构建套接字服务器和客户端所需的一切。

这个API和标准C API的区别在于它的面向对象。 在C中，从套接字调用中检索套接字描述符，然后将其用作BSD API函数的参数。 在Python中，套接字方法返回套接字方法可应用的套接字对象。 表2提供了一些类方法，表3显示了实例方法的一个子集。

表2. Socket模块的类方法

|  |  |
| --- | --- |
| 类方法 | 描述 |
| Socket | 底层网络接口（BSD API） |
| socket.socket(family, type) | 创建并返回一个新的套接字对象 |
| socket.getfqdn(name) | 将字符串IP地址转换为完全限定的域名 |
| socket.gethostbyname(hostname) | 根据主机名得到IP地址字符串 |
| socket.fromfd(fd, family, type) | 根据现有的文件描述符创建一个套接字对象 |

表3. Socket模块的实例化方法

|  |  |
| --- | --- |
| 实例化方法 | 描述 |
| sock.bind((adrs, port) ) | 将套接字绑定到地址和端口 |
| sock.accept() | 返回一个客户端套接字（带有对等地址信息） |
| sock.listen(backlog) | 将套接字置于监听状态，能够挂起未完成的连接请求 |
| sock.connect((adrs, port) ) | 将套接字连接到定义的主机和端口 |
| sock.recv( buflen[, flags] ) | 从套接字接收数据，最多为buflen字节 |
| sock.recvfrom( buflen[, flags] ) | 从套接字接收数据，直到buflen字节，还返回数据来自的远程主机和端口 |
| sock.send( data[, flags] ) | 通过套接字发送数据 |
| sock.sendto( data[, flags], addr ) | 通过套接字发送数据，根据指定的地址 |
| sock.close() | 关闭套接字 |
| sock.getsockopt( lvl, optname ) | 获取指定套接字选项的值 |
| sock.setsockopt(lvl, optname, val ) | 设置指定套接字选项的值 |

类方法和实例方法的不同之处在于，实例方法需要执行一个套接字实例（从套接字返回），而类方法则不需要。

3.3 SocketServer

SocketServer模块是一个有趣的模块，可以简化套接字服务器的开发。 关于它的使用的讨论远远超出了本教程的范围，但是我将演示它的基本用法，点击[这个链接地址](https://www.ibm.com/developerworks/linux/tutorials/l-pysocks/#resources)，以获得更详细讨论的链接。

例2，简单示例。在这里，我实现了一个简单的“Hello World”服务器，它在客户端连接时发出消息。 我首先创建一个继承了SocketServer.StreamRequestHandler类的请求处理程序。 我定义了一个称为handle的方法来处理服务器的请求。 服务器所做的一切必须在这个函数的上下文中处理（最后，套接字被关闭）。 这个过程很简单，但是你可以用这个类来实现简单的HTTP服务器。 在处理方法中，我发出我的招呼，然后退出。

现在连接处理程序已准备就绪，剩下的就是创建套接字服务器。我使用  
SocketServer.TCPServer类，提供地址和端口号（服务器将绑定到）和我的请求处理程序方法。得到的是一个TCPServer对象。 调用serve\_forevermethod启动服务器并使其可用于连接。

例2. 用SocketServer模块实现一个简单的服务器

import SocketServer

class hwRequestHandler(SocketServer.StreamRequestHandler):

def handle(self):

self.wfile.write("Hello World!\n")

server = SocketServer.TCPServer ( ("", 2525), hwRequestHandler )

server. serve\_forever()

就是这样！ Python允许在这个主题上有许多变体，包括UDPServers，进程和线程服务器。

4 Python套接字编程

在使用套接字的语言中，套接字通常是相同的 - 它是两个可以相互通信的应用程序之间的通道。

4.1 依赖条件

无论你是用Python，Perl，Ruby，Scheme还是其他有用的语言编写套接字应用程序（并且我的意思是有套接字接口的语言），套接字都是相同的。 这是两个应用程序之间的通道，可以相互通信（本地在单个机器上，或者在两个不同位置的机器之间）。

与像Python这样的语言编程的套接字的区别在于可以简化套接字编程的辅助类和方法。 在本节中，我将演示Python套接字API。 您可以使用脚本执行Python解释器，或者如果您自己执行Python，则可以一次一行地与它交互。 这样，你可以看到每个方法调用的结果。

以下示例说明了与Python解释器的交互。 在这里，我使用套接字类方法gethostbyname将完全限定的域名（www.baidu.com）解析为字符串四段的IP地址（'61.135.169.125'）：

[camus]$ python  
Python 2.4 (#1, Feb 20 2005, 11:25:45)  
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2  
Type "help", "copyright", "credits" or "license" for more  
information.  
>>> import socket  
>>> socket.gethostbyname('www.baidu.com')  
'61.135.169.125'  
>>>

在导入套接字模块后，我调用gethostbyname类方法将域名解析为IP地址。

现在，我将讨论基本套接字方法和通过套接字进行通信。带着你的python自由跟随吧。

4.1创建和销毁套接字

要创建一个新的套接字，可以使用套接字类的套接字方法。 这是一个类方法，因为您还没有从中应用方法的套接字对象。 套接字方法与BSD API相似，如创建一个流（TCP）和数据报（UDP）套接字所示：

例4.创建流和数据报套接字

streamSock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM )  
dgramSock = socket.socket ( socket.AF\_INET, socket.SOCK\_DGRAM )

在每种情况下，返回一个套接字对象。 AF\_INET符号 - 参数1表示您正在请求Internet协议（IP）套接字，特别是IPv4。 第二个参数是传输协议类型（SOCK\_STREAM用于TCP套接字，SOCK\_DGRAM用于UDP套接字）。 如果您的底层操作系统支持IPv6，则还可以指定socket.AF\_INET6来创建IPv6套接字。

要关闭连接的套接字，请使用close方法：

streamSock.close()

最后，你可以用del语句删除一个套接字：

del streamSock

该语句永久删除套接字对象。 此后尝试引用套接字会产生错误。

4.2套接字地址

套接字的端点地址是由接口地址和端口号组成的元组。 因为Python可以很容易地表示元组，所以地址和端口也就很容易表示。这说明接口地址192.168.1.1和端口80的端点：

( '192.168.1.1', 80 )

您也可以在此处使用完全限定的域名，例如：

( 'www.ibm.com', 25 )

这个例子很简单，肯定会击败C中需要的sockaddr\_in操作。下面的讨论提供了Python中地址的例子。

4.3服务器套接字

服务器套接字通常是在网络上公开服务的套接字。由于服务器和客户端套接字是以不同的方式创建的，我将独立讨论它们。

创建套接字后，您使用bind方法将地址绑定到它，使用listen方法将其置于侦听状态，最后使用accept方法接受新的客户端连接。如下所示：

例5.使用服务器套接字

sock = socket.socket ( socket.AF\_INET, socket.SOCK\_STREAM )  
sock.bind(('', 2525) )  
sock.listen( 5 )  
newsock, (remhost, remport) = sock.accept()

对于这个服务器，使用地址（''，2525），这意味着通配符用于接口地址（''），允许来自主机上任何接口的传入连接。 您还绑定到端口号2525。

请注意，accept方法不仅返回表示客户端连接（newsock）的新套接字对象，还返回地址元组（套接字对端的远程地址和端口号）。 Python的SocketServer模块可以进一步简化这个过程，如上所示。

您也可以创建数据报服务器，但它们是无连接的，因此没有关联的接受方法。 以下示例创建一个数据报服务器套接字：

例6.创建一个数据报服务器套接字

sock = socket.socket( socket.AF\_INET, socket.SOCK\_DGRAM )  
sock.bind( ('', 2525) )

即将到来的关于套接字I/O的讨论显示了I/O如何适用于流和数据报套接字。

现在，我们来探讨客户端如何创建套接字并将其连接到服务器。

4.4 客户端套接字

创建和连接客户端套接字的机制与服务器套接字的设置相似。 在创建套接字时，需要一个地址 - 不要在本地绑定套接字（就像服务器一样），而是要确定套接字应该附加到哪里。 假设主机上有一台服务器，其接口IP地址为“192.168.1.1”，端口为2525.以下代码创建一个新的套接字并将其连接到定义的服务器：

例7.创建一个流套接字并连接到服务器

sock = socket.socket( socket.AF\_INET, socket.SOCK\_STREAM)  
sock.connect(('192.168.1.1', 2525))

对于数据报套接字，过程有点不同。 回想一下，数据报套接字本质上是断开的。 一种思考的方法如下：尽管流套接字是两个端点之间的管道，但是数据报套接字是基于消息的，能够同时与多个对等体进行通信。 这是一个数据报客户端的例子。

例8.创建一个数据报套接字并连接到服务器

sock = socket.socket( socket.AF\_INET, sock.sock\_DGRAM )  
sock.connect( ('192.168.1.1', 2525) )

这里的不同之处在于，即使我使用了连接方法，客户端和服务器之间也没有真正的连接。 这里的连接是对后面的I/O的简化。通常在数据报套接字中，您必须提供您要发送的数据的目标信息。通过使用连接，我已经与客户端缓存这个信息和发送方法可以发生很像流套接字版本（没有目标地址是必要的）。 您可以再次调用connect来重新指定数据报客户端消息的目标。

4.5 流套接字I/O

在Python中通过流套接字发送或接收数据很简单。存在几种通过流套接字移动数据的方法（如发送，接收，读取和写入）。

这第一个例子演示了一个服务器和客户端的流套接字。在这个演示中，服务器回应从客户端收到的任何内容。

echo服务器如例9所示。创建一个新的流套接字时，绑定一个地址（接受来自任何接口和端口45000的连接），然后调用listen方法来启用传入连接。echo服务器然后进入客户端连接的循环。 accept方法被调用，并阻塞（即不返回），直到新客户端连接，此时新客户端套接字与远程客户端的地址信息一起返回。使用这个新的客户端套接字，我调用recv从对端获取一个字符串，然后将这个字符串写回到套接字。然后，立即关闭了这次连接。

例9.简单的Python echo服务器

import socket

srvsock = socket.socket ( socket.AF\_INET, socket.SOCK\_STREAM )  
srvsock.bind( ('', 23000) )  
srvsock.listen( 5 )  
while 1:  
 clisock, (remhost, remport) = srvsock.accept()  
 str = clisock.recv(100)  
 clisock.send( str )  
 clisock.close()

例10显示了与例9中的服务器相对应的echo客户机。创建一个新的流套接字时，使用connect方法将此套接字连接到服务器。 当连接时（当连接方法返回时），客户端用send方法发出一个简单的文本消息，然后用recv方法等待回声。 print语句用于发出所读的内容。 完成后，执行close方法关闭套接字。

例10.简单的Python echo服务器

import socket

clisock = socket.socket( socket.AF\_INET, socket.SOCK\_STREAM )

clisock.connect( ('', 23000) )

clisock.send("Hello World\n")

print clisock.recv(100)

clisock.close()

4.6 数据报套接字I/O

数据报套接字本质上是断开的，这意味着通信要求提供目的地址。同样，当通过套接字接收到消息时，必须返回数据的来源。recvfrom和sendto方法支持额外的地址信息，正如您在数据报回显服务器和客户端实现中所看到的那样。

例11显示了数据报echo服务器。首先创建一个套接字，然后使用bind方法绑定到一个地址。 然后服务器进入一个无限循环，等待客户端的请求。recvfrom方法接收来自数据报套接字的消息，不仅返回消息，而且还返回消息源的地址。然后用sendto方法将这些信息转换回来，再发送给客户端。

例11. 简单的Python数据报echo服务器

import socket

dgramSock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM )

dgramSock.bind(('', 23000) )

while 1:

msg, (addr, port) = dgramSock.recvfrom(100)

dgramSock.sendto(msg,(addr, port) )

数据报客户端更简单。创建数据报套接字后，我使用sendto方法将消息发送到特定的地址。 （记住：Datagrams没有连接。）sendto完成后，我用recv等待echo回应，然后打印它。 请注意，我不会在这里使用recvfrom，因为我对对等地址信息不感兴趣。

例12.简单Python数据报echo客户端

import socket

dgramSock = socket.socket( socket.AF\_INET, socket.SOCK\_DGRAM )  
dgramSock.sendto( "Hello World\n", ('', 23000) )

print dgramSock.recv( 100 )

dgramSock.close()

4.7 套接字选项

套接字默认为一组标准行为，但可以使用选项来改变套接字的行为。 您可以使用setsockopt方法操作套接字选项，并使用getsockopt方法捕获它们。

在Python中使用套接字选项很简单，如例13所示。在第一个示例中，我读取套接字发送缓冲区的大小。 在第二个例子中，我得到SO\_REUSEADDR选项的值（在TIME\_WAIT周期内重新使用该地址），然后启用它。

例13. 使用Socket套接字

sock = <strong>socket.socket(socket.AF\_INET, socket.SOCK\_STREAM )

# 获取socket发送缓存的大小

bufsize = sock.<strong>getsockopt( socket.SOL\_SOCKET, socket.SO\_SNDBUF )

# 获取SO\_REUSEADDR选项

state = sock. getsockopt ( socket.SOL\_SOCKET, socket.SO\_REUSEADDR )

# 使能SO\_REUSEADDR选项

sock.setsockopt( socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1 )

SO\_REUSEADDR选项最常用于套接字服务器开发。您可以增加套接字发送和接收缓冲区以获得更好的性能，但是鉴于您使用的是解释型脚本语言，因此可能无法为您带来太多好处。

4.8 异步I/O

Python提供异步I / O作为select模块的一部分。这个功能类似于C select机制，但有一些简化。我将首先介绍select，然后向您展示如何在Python中使用它。

select方法允许您监听多个套接字的复用事件和几个不同的事件。例如，您可以指示select在套接字有数据可用时通知您，可以通过套接字写入数据时以及在套接字上发生错误时通知您; 您可以同时为多个套接字执行这些操作。

当C用位工作时，Python使用列表来表示要监视的描述符以及其约束满足的返回描述符。 考虑下面的例子，你等待来自标准输入的一些输入：

例14. 等待来自stdin的输入

rlist, wlist, elist = select.select( [sys.stdin], [], [] )

print sys.stdin.read()

传递给select的参数是表示读取事件，写入事件和错误事件的列表。 select方法返回三个包含事件满足的对象的列表（读，写，异常）。 在这个例子中，返回rlist时应该是[sys.stdin]，表明数据可以在stdin上读取。 然后用读取方法读取这些数据。

select方法也适用于套接字描述符。在下面的例子中（见清单15），创建了两个客户端套接字并连接到远程对等端。 然后使用select方法来确定哪个套接字有数据可供读取。 数据然后被读取并发送到stdout。

例15. 用多个套接字演示select方法

import socket

import select

sock1 = socket.socket( socket.AF\_INET, socket.SOCK\_STREAM )

sock2 = socket.socket( socket.AF\_INET, socket.SOCK\_STREAM )

sock1.connect(('192.168.1.1', 25) )

sock2.connect(('192.168.1.1', 25) )

while 1:

# 等待一个read事件

rlist, wlist, elist = select.select([sock1,sock2], [], [], 5 )

# 测试timeout

if [rlist, wlist, elist] == [ [], [], [] ]:

print "Five seconds elapsed.\n"

else:

# 遍历rlist列表中的每一个Socket, 读取并打印可用的数据

for sock in rlist:

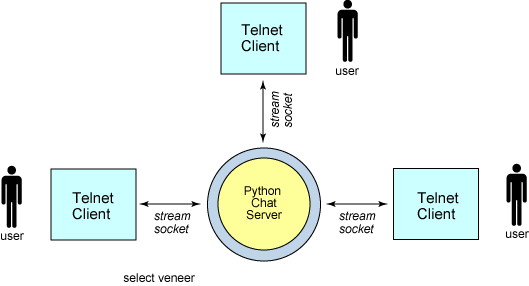
print sock.recv(100)

5构建一个Python聊天服务器

5.1 一个简单的聊天服务器

您已经探索了Python的基本网络API;现在你可以把这些知识用在一个简单的应用程序中。 在本节中，您将构建一个简单的聊天服务器。使用Telnet，客户端可以连接到您的Python聊天服务器，并在全服务器范围内相互通信。提交给聊天服务器的消息被其他人查看（除了管理信息，例如客户端加入或离开聊天服务器）。该模型如图1所示。

图1. 聊天服务器使用select方法来支持任意数量的客户端



对于这个聊天服务器，一个重要的要求是它必须是可扩展的。服务器必须能够支持任意数量的流（TCP）客户端。

要支持任意数量的客户端，请使用select方法异步管理客户端列表。但是你也使用select的特性来作为你的服务器套接字。select的select事件决定了客户端何时有数据可读，但是也可以用来确定服务器套接字何时有新客户端尝试连接。 您利用此行为来简化服务器的开发。

接下来，我将探索Python聊天服务器的源代码，并确定Python如何帮助简化实现。

5.2 ChatServer类

我们先看看Python聊天服务器类和\_\_init\_\_方法 - 创建新实例时调用的构造函数。

该类由四个方法组成。run方法被调用来启动服务器并允许客户端连接。 broadcast\_string和accept\_new\_connection方法在类中内部使用，稍后将会讨论。

\_\_init\_\_方法是在创建类的新实例时调用的特殊方法。请注意，所有方法都可以self参数，即对类实例本身的引用（非常类似于C ++中的this参数）。 您将看到self参数，这是所有实例方法的一部分，用于访问实例变量。

\_\_init\_\_方法创建三个实例变量。port是服务器的端口号（在构造函数中传递）。srvsock是此实例的套接字对象，描述符是包含该类的每个套接字对象的列表。您可以在select方法中使用此列表来标识读取事件列表。

最后，例16显示了\_\_init\_\_方法的代码。创建一个流套接字之后，启用SO\_REUSEADDR套接字选项，以便服务器可以在必要时快速重新启动。通配符地址与定义的端口号绑定。然后调用listen方法来允许即将接入的连接。服务器套接字被添加到描述符列表（目前唯一的元素），但所有客户端套接字将在到达时添加（请参阅accept\_new\_connection方法）。标准输出提供给标准输出，表示服务器已经启动。

例16. 带有init 方法的ChatServer类

import socket

import select

class ChatServer:

def \_\_init\_\_( self, port ):

self.port = port;

self.srvsock = ocket.socket(socket.AF\_INET, socket.SOCK\_STREAM )

self.srvsock.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1 )

self.srvsock.bind( ("", port) )

self.srvsock.listen(5)

self.descriptors = [self.srvsock]

print 'ChatServer started on port %s' % port

def run( self ):

...

def broadcast\_string( self, str, omit\_sock ):

...

def accept\_new\_connection( self ):

5.2 run方法

run方法是聊天服务器的循环（参见例17）。当被调用时，它进入一个无限循环，提供连接的客户端之间的通信。

I pass the descriptor list (which contains all the server's sockets) as the read event list to select (and null lists for write and exception). When a read event is detected, it's returned as sread. (I ignore the swrite and sexc lists.) The sread list contains thesocket objects that will be serviced. I iterate through the sread list, checking each socket object found.

服务器的核心是select方法。我将描述符列表（包含所有服务器的套接字）作为读取事件列表进行选择（对于写入和异常，列表为空列表）。 当检测到读取事件时，它将作为脚本返回。 （我忽略swrite和sexc列表。）sread列表包含将被服务的套接字对象。 我遍历sread列表，检查每个找到的套接字对象。

The first check in the iterator loop is if the socket object is the server. If it is, a new client is trying to connect and the accept\_new\_connection method is called. Otherwise, the client socket is read. If a null is returned from recv, the peer socket closed.

迭代器循环中的第一个检查是如果套接字对象是服务器。 如果是，则新客户端正在尝试连接，并调用accept\_new\_connection方法。 否则，读取客户端套接字。 如果从recv返回null，则关闭对等套接字。

In this case, I construct a message and send it to all connected clients, close the peer socket, and remove the corresponding object from the descriptor list. If the recv return is not null, a message is available and stored in str. This message is distributed to all other clients using broadcast\_string.

Listing 17. The chat server run method is the core of the chat server

def run( self ):

while 1:

# 等待一个可读的套接字描述符的事件

(sread, swrite, sexc) = select.select( self.descriptors, [], [] )

# 遍历标记的可读描述符

for sock in sread:

# 收到一个连接到服务器（侦听）套接字

if sock == self.srvsock:

self.accept\_new\_connection()

else:

# Received something on a client socket

str = sock. recv (100)

# Check to see if the peer socket closed

if str == '':

host,port = sock.getpeername()

str = 'Client left %s:%s\r\n' % (host, port)

self.broadcast\_string( str, sock )

sock.close

self.descriptors.remove(sock)

else:

host,port = sock.getpeername()

newstr = '[%s:%s] %s' % (host, port, str)

self.broadcast\_string( newstr, sock )

Helper方法

The two helper methods in the chat server class provide methods for accepting new client connections and broadcasting messages to the connected clients.

The accept\_new\_connection method (see Listing 18) is called when a new client is detected on the incoming connection queue. The accept method is used to accept the connection, which returns the new socket object and remote address information. I immediately add the new socket to the descriptors list, then send a salutation to the new client welcoming the client to the chat. I create a string identifying that the client has connected and broadcast this information to the group using broadcast\_string (see Listing 19).

Note that in addition to the string being broadcast, a socket object is also passed. The reason is that I want to selectively omit some sockets from getting certain messages. For example, when a client sends a message to the group, the message goes to the group but not back to itself. When I generate the status message identifying a new client joining the group, it goes to the group but not the new client. This task is performed in broadcast\_string with the omit\_sock argument. This method walks through the descriptors list and sends the string to all sockets that are not the server socket and notomit\_sock.

Listing 18. Accepting a new client connection on the chat server

def accept\_new\_connection( self ):

newsock, (remhost, remport) = self.srvsock.<strong>accept</strong>()

self.descriptors.append( newsock )

newsock.send("You're connected to the Python chatserver\r\n")

str = 'Client joined %s:%s\r\n' % (remhost, remport)

self.broadcast\_string( str, newsock )

Listing 19. Broadcasting a message to the chat group

def broadcast\_string( self, str, omit\_sock ):

for sock in self.descriptors:

if sock != self.srvsock and sock != omit\_sock:

sock.<strong>send</strong>(str)

print str,

Instantiating a new ChatServer

Now that you've seen the Python chat server (under 50 lines of code), let's see how to instantiate a new chat server object in Python.

Start the server by creating a new ChatServer object (passing the port number to be used), then calling the run method to start the server and allow incoming connections:

Listing 20. Instantiating a new chat server

myServer = ChatServer( 2626 )

myServer.run()

At this point, the server is running and you can connect to it from one or more clients. You can also chain methods together to simplify this process (as if it needs to be simpler):

Listing 21. Chaining methods

myServer = ChatServer( 2626 ).run()

which achieves the same result. I'll show the ChatServer class in operation.

Demonstrating the ChatServer

Here's the ChatServer in action. I show the output of the ChatServer (see Listing 22 and the dialog between two clients (see Listing 23 and Listing 24). The user-entered text appears in bold.

Listing 22. Output from the ChatServer

[plato]$ <**strong**>python pchatsrvr.py</**strong**>

ChatServer started on port 2626

Client joined 127.0.0.1:37993

Client joined 127.0.0.1:37994

[127.0.0.1:37994] Hello, is anyone there?

[127.0.0.1:37993] Yes, I'm here.

[127.0.0.1:37993]  Client left 127.0.0.1:37993

Listing 23. Output from Chat Client #1

[plato]$ <**strong**>telnet localhost 2626</**strong**>

Trying 127.0.0.1...

Connected to localhost.localdomain (127.0.0.1).

Escape character is '^]'.

You're connected to the Python chatserver

Client joined 127.0.0.1:37994

[127.0.0.1:37994] Hello, is anyone there?

<**strong**>Yes, I'm here.</**strong**><**strong**>^]</**strong**>

telnet> <**strong**>close</**strong**>

Connection closed.

[plato]$

Listing 24. Output from Chat Client #2

[plato]$ <**strong**>telnet localhost 2626</**strong**>

Trying 127.0.0.1...

Connected to localhost.localdomain (127.0.0.1).

Escape character is '^]'.

You're connected to the Python chatserver

<**strong**>Hello, is anyone there?</**strong**>

[127.0.0.1:37993] Yes, I'm here.

[127.0.0.1:37993] Client left 127.0.0.1:37993

As you see in Listing 22, all dialog between all clients is emitted to stdout, including client connect and disconnect messages.

High-level networking classes

Networking modules

Python includes several specialized modules for application-layer protocols (built on the standard socket module). The available modules are wide and varied and they provide module implementations of the Hypertext Transfer Protocol (HTTP), the Simple Mail Transfer Protocol (SMTP), Internet Message Access Protocol (IMAP) and Post Office Protocol (POP3), the Network News Transfer Protocol (NNTP), XML-RPC (remote procedure call), FTP, and many others.

This section demonstrates the modules shown in Table 4.

Table 4. Useful application-layer protocol modules

|  |  |
| --- | --- |
| 模块 | Protocol implemented |
| httplib | HTTP client |
| smtplib | SMTP client |
| poplib | POP3 client |

The httplib (HTTP client)

The HTTP client interface can be useful when developing Web robots or other Internet scraping agents. The Web protocol is request/response in nature over stream sockets. Python makes it easy to build Web robots through a simple Web interface.

Listing 25 demonstrates the httplib module. You create a new HTTP client instance with HTTPConnection, providing the Web site to which you want to connect. With this new object (httpconn), you can request files with the request method. Within request, you specify the HTTP GETmethod (which requests a file from the server, compared to HEAD which simply retrieves information about the file). The getresponse method parses the HTTP response header to understand if an error was returned. If the file was successfully retrieved, the read method on the new response object returns and prints the text.

Listing 25. Building a simple (non-rendering) HTTP client with httplib

import httplib

httpconn = <strong>httplib.HTTPConnection</strong>("www.ibm.com")

httpconn.<strong>request</strong>("GET", "/developerworks/index.html")

resp = httpconn.<strong>getresponse</strong>()

if resp.reason == "OK":

resp\_data = resp.<strong>read</strong>()

print resp\_data

httpconn.<strong>close</strong>()

The smptlib (SMTP client)

SMTP allows you to send e-mail messages to a mail server which can be useful in networking systems to relay status about the operation of a device. The Python module for sending e-mail messages is simple and consists of creating an SMTP object, sending an e-mail message using the sendmail method, then closing the connection with the quit method.

The example in Listing 26 demonstrates sending a simple e-mail message. The msg string contains the body of the message (which should include the subject line).

Listing 26. Sending a short e-mail message with smtplib

import smtplib

fromAdrs = 'mtj@mtjones.com'

toAdrs = 'you@mail.com'

msg = 'From: me@mail.com\r\nTo: you@mail.com\r\nSubject:Hello\r\nHi!\r\n'

mailClient = <strong>smtplib.SMTP</strong>('192.168.1.1')

mailClient.<strong>sendmail</strong>( fromAdrs, toAdrs, msg )

mailClient.<strong>quit</strong>

The poplib (POP3 client)

POP3 is another useful application-layer protocol for which a module exists within Python. The POP3 protocol allows you to connect to a mail server and download new mail, which can be useful for remote commanding -- embedding commands within the body of an e-mail message. After executing the embedded command, you can use smptlib to return a response e-mail message to the source.

This demonstration in Listing 27 shows a simple application that connects to a mail server and emits the subject lines for all pending e-mail for the user.

The poplib is relatively simple but offers several methods for gathering and managing e-mail at the server. In this example, I create a new POP3 object with the POP3 method, specifying the mail server. The user and pass\_ methods authenticate the application to the server; the stat method returns the number of messages waiting for the user and the total number of bytes taken up by all messages.

Next, I loop through each available message and use the retr method to grab the next e-mail message. This method returns a list of the form:

(response, ['line, ...], octets)

where response is the POP3 response for the particular message, the linelist represents the individual lines of the e-mail message, and the final element, octets, is the number of bytes for the e-mail message. The inner loop simply iterates over the second element ([1]) of the list which is the list of the e-mail message body. For each line, I test whether 'Subject:' is present; if so, I print this line.

After all e-mail messages have been checked, a call to the quit method ends the POP3 session.

Instead of using the retr method, you could also use the top method to extract the header for the e-mail message. This step would be faster and minimize the amount of data transferred to this client.

Listing 27. Retrieving e-mail messages from a POP3 mail server and emitting the subject line

import poplib

import re

popClient = <strong>poplib.POP3</strong>('192.168.1.1')

popClient.<strong>user</strong>('user')

popClient.<strong>pass\_</strong>('password')

numMsgs, mboxSize = popClient.<strong>stat</strong>()

print "Number of messages ", numMsgs

print "Mailbox size", mboxSize

print

for id in range (numMsgs):

for mail in popClient.<strong>retr</strong>(id+1)[1]:

if re.search( 'Subject:', mail ):

print mail

print

popClient.<strong>quit</strong>()

Summary

This tutorial reviewed the basics of the sockets API and showed how to build networking applications in Python. The standard sockets module was introduced as a way to build both client and server applications, as well as the SocketServer module which simplifies the construction of simple socket servers. I presented a simple chat server implemented in Python that offered support for a scalable number of clients using the select method. In closing, I previewed some of Python's high-level networking classes that simplify the development of applications requiring application-layer networking protocols.

Python is an interesting and useful language that is well worth your time to learn.