11.1 作为客户端与HTTP服务交互¶

问题¶

你需要通过HTTP协议以客户端的方式访问多种服务。例如，下载数据或者与基于REST的API进行交互。

解决方案¶

对于简单的事情来说，通常使用 `urllib.request` 模块就够了。例如，发送一个简单的HTTP GET请求到远程的服务上，可以这样做：

```
from urllib import request, parse

# Base URL being accessed
url = 'http://httpbin.org/get'

# Dictionary of query parameters (if any)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}

# Encode the query string
querystring = parse.urlencode(parms)

# Make a GET request and read the response
u = request.urlopen(url+'?' + querystring)
resp = u.read()
```

如果你需要使用POST方法在请求主体中发送查询参数，可以将参数编码后作为可选参数提供给 `urlopen()` 函数，就像这样：

```
from urllib import request, parse

# Base URL being accessed
url = 'http://httpbin.org/post'

# Dictionary of query parameters (if any)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}

# Encode the query string
querystring = parse.urlencode(parms)

# Make a POST request and read the response
u = request.urlopen(url, querystring.encode('ascii'))
resp = u.read()
```

如果你需要在发出的请求中提供一些自定义的HTTP头，例如修改 `user-agent` 字段,可以创建一个包含字段值的字典，并创建一个Request实例然后将其传给 `urlopen()` ，如下：

```
from urllib import request, parse
...

# Extra headers
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

req = request.Request(url, querystring.encode('ascii'), headers=headers)

# Make a request and read the response
```

```
u = request.urlopen(req)
resp = u.read()
```

如果需要交互的服务比上面的例子都要复杂，也许应该去看看 requests 库（https://pypi.python.org/pypi/requests）。例如，下面这个示例采用requests库重新实现了上面的操作：

```
import requests

# Base URL being accessed
url = 'http://httpbin.org/post'

# Dictionary of query parameters (if any)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}

# Extra headers
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

resp = requests.post(url, data=parms, headers=headers)

# Decoded text returned by the request
text = resp.text
```

关于requests库，一个值得一提的特性就是它能以多种方式从请求中返回响应结果的内容。从上面的代码来看，`resp.text` 带给我们的是以Unicode解码的响应文本。但是，如果去访问 `resp.content`，就会得到原始的二进制数据。另一方面，如果访问 `resp.json`，那么就会得到JSON格式的响应内容。

下面这个示例利用 `requests` 库发起一个HEAD请求，并从响应中提取出一些HTTP头数据的字段：

```
import requests

resp = requests.head('http://www.python.org/index.html')

status = resp.status_code
last_modified = resp.headers['last-modified']
content_type = resp.headers['content-type']
content_length = resp.headers['content-length']
```

下面是一个利用requests通过基本认证登录Pypi的例子：

```
import requests

resp = requests.get('http://pypi.python.org/pypi?:action=login',
                    auth=('user','password'))
```

下面是一个利用requests将HTTP cookies从一个请求传递到另一个的例子：

```
import requests

# First request
resp1 = requests.get(url)
...

# Second requests with cookies received on first requests
resp2 = requests.get(url, cookies=resp1.cookies)
```

最后但并非最不重要的一个例子是用requests上传内容：

```
import requests
url = 'http://httpbin.org/post'
files = { 'file': ('data.csv', open('data.csv', 'rb')) }

r = requests.post(url, files=files)
```

讨论¶

对于真的很简单HTTP客户端代码，用内置的 `urllib` 模块通常就足够了。但是，如果你要做的不仅仅只是简单的GET或POST请求，那就真的不能再依赖它的功能了。这时候就是第三方模块比如 `requests` 大显身手的时候了。

例如，如果你决定坚持使用标准的程序库而不考虑像 `requests` 这样的第三方库，那么也许就不得不使用底层的 `http.client` 模块来实现自己的代码。比方说，下面的代码展示了如何执行一个HEAD请求：

```
from http.client import HTTPConnection
from urllib import parse

c = HTTPConnection('www.python.org', 80)
c.request('HEAD', '/index.html')
resp = c.getresponse()

print('Status', resp.status)
for name, value in resp.getheaders():
    print(name, value)
```

同样地，如果必须编写涉及代理、认证、cookies以及其他一些细节方面的代码，那么使用 `urllib` 就显得特别别扭和啰嗦。比方说，下面这个示例实现在Python包索引上的认证：

```
import urllib.request

auth = urllib.request.HTTPBasicAuthHandler()
auth.add_password('pypi','http://pypi.python.org','username','password')
opener = urllib.request.build_opener(auth)

r = urllib.request.Request('http://pypi.python.org/pypi?:action=login')
u = opener.open(r)
resp = u.read()

# From here. You can access more pages using opener
...
```

坦白说，所有的这些操作在 `requests` 库中都变得简单的多。

在开发过程中测试HTTP客户端代码常常是很令人沮丧的，因为所有棘手的细节问题都需要考虑（例如cookies、认证、HTTP头、编码方式等）。要完成这些任务，考虑使用httpbin服务（http://httpbin.org）。这个站点会接收发出的请求，然后以JSON的形式将相应信息回传回来。下面是一个交互式的例子：

```
>>> import requests
>>> r = requests.get('http://httpbin.org/get?name=Dave&n=37',
...     headers = { 'User-agent': 'goaway/1.0' })
>>> resp = r.json()
>>> resp['headers']
{'User-Agent': 'goaway/1.0', 'Content-Length': '', 'Content-Type': '',
'Accept-Encoding': 'gzip, deflate, compress', 'Connection':
'keep-alive', 'Host': 'httpbin.org', 'Accept': '*/*'}
>>> resp['args']
{'name': 'Dave', 'n': '37'}
>>>
```

在要同一个真正的站点进行交互前，先在 httpbin.org 这样的网站上做实验常常是可取的办法。尤其是当我们面对3次登录失败就会关闭账户这样的风险时尤为有用（不要尝试自己编写HTTP认证客户端来登录你的银行账户）。

尽管本节没有涉及， `request` 库还对许多高级的HTTP客户端协议提供了支持，比如OAuth。 `requests` 模块的文档（http://docs.python-requests.org)质量很高（坦白说比在这短短的一节的篇幅中所提供的任何信息都好），可以参考文档以获得更多地信息。

11.10 在网络服务中加入SSL¶

问题¶

你想实现一个基于sockets的网络服务，客户端和服务器通过SSL协议认证并加密传输的数据。

解决方案¶

`ssl` 模块能为底层socket连接添加SSL的支持。 `ssl.wrap_socket()` 函数接受一个已存在的socket作为参数并使用SSL层来包装它。 例如，下面是一个简单的应答服务器，能在服务器端为所有客户端连接做认证。

```
from socket import socket, AF_INET, SOCK_STREAM
import ssl

KEYFILE = 'server_key.pem'   # Private key of the server
CERTFILE = 'server_cert.pem' # Server certificate (given to client)

def echo_client(s):
    while True:
        data = s.recv(8192)
        if data == b'':
            break
        s.send(data)
    s.close()
    print('Connection closed')

def echo_server(address):
    s = socket(AF_INET, SOCK_STREAM)
    s.bind(address)
    s.listen(1)

    # Wrap with an SSL layer requiring client certs
    s_ssl = ssl.wrap_socket(s,
                            keyfile=KEYFILE,
                            certfile=CERTFILE,
                            server_side=True
                            )
    # Wait for connections
    while True:
        try:
            c,a = s_ssl.accept()
            print('Got connection', c, a)
            echo_client(c)
        except Exception as e:
            print('{}: {}'.format(e.__class__.__name__, e))

echo_server(('', 20000))
```

下面我们演示一个客户端连接服务器的交互例子。客户端会请求服务器来认证并确认连接：

```
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> import ssl
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s_ssl = ssl.wrap_socket(s,
                cert_reqs=ssl.CERT_REQUIRED,
                ca_certs = 'server_cert.pem')
>>> s_ssl.connect(('localhost', 20000))
>>> s_ssl.send(b'Hello World?')
12
>>> s_ssl.recv(8192)
b'Hello World?'
>>>
```

这种直接处理底层socket方式有个问题就是它不能很好的跟标准库中已存在的网络服务兼容。 例如，绝大部分服务器代码（HTTP、XML-RPC等）实际上是基于 `socketserver` 库的。 客户端代码在一个较高层上实现。我们需要另外一种稍微不同的方式来将SSL添加到已存在的服务中：

首先，对于服务器而言，可以通过像下面这样使用一个mixin类来添加SSL：

```
import ssl

class SSLMixin:
    '''
    Mixin class that adds support for SSL to existing servers based
    on the socketserver module.
    '''
    def __init__(self, *args,
                 keyfile=None, certfile=None, ca_certs=None,
                 cert_reqs=ssl.CERT_NONE,
                 **kwargs):
        self._keyfile = keyfile
        self._certfile = certfile
        self._ca_certs = ca_certs
        self._cert_reqs = cert_reqs
        super().__init__(*args, **kwargs)

    def get_request(self):
        client, addr = super().get_request()
        client_ssl = ssl.wrap_socket(client,
                                     keyfile = self._keyfile,
                                     certfile = self._certfile,
                                     ca_certs = self._ca_certs,
                                     cert_reqs = self._cert_reqs,
                                     server_side = True)
        return client_ssl, addr
```

为了使用这个mixin类，你可以将它跟其他服务器类混合。例如，下面是定义一个基于SSL的XML-RPC服务器例子：

```
# XML-RPC server with SSL

from xmlrpc.server import SimpleXMLRPCServer

class SSLSimpleXMLRPCServer(SSLMixin, SimpleXMLRPCServer):
    pass

Here's the XML-RPC server from Recipe 11.6 modified only slightly to use SSL:

import ssl
from xmlrpc.server import SimpleXMLRPCServer
from sslmixin import SSLMixin

class SSLSimpleXMLRPCServer(SSLMixin, SimpleXMLRPCServer):
    pass

class KeyValueServer:
    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys']
    def __init__(self, *args, **kwargs):
        self._data = {}
        self._serv = SSLSimpleXMLRPCServer(*args, allow_none=True, **kwargs)
        for name in self._rpc_methods_:
            self._serv.register_function(getattr(self, name))

    def get(self, name):
        return self._data[name]

    def set(self, name, value):
        self._data[name] = value

    def delete(self, name):
        del self._data[name]

    def exists(self, name):
        return name in self._data

    def keys(self):
        return list(self._data)
```

```
    def serve_forever(self):
        self._serv.serve_forever()

if __name__ == '__main__':
    KEYFILE='server_key.pem'    # Private key of the server
    CERTFILE='server_cert.pem'  # Server certificate
    kvserv = KeyValueServer(('', 15000),
                            keyfile=KEYFILE,
                            certfile=CERTFILE)
    kvserv.serve_forever()
```

使用这个服务器时，你可以使用普通的 `xmlrpc.client` 模块来连接它。 只需要在URL中指定 `https:` 即可，例如：

```
>>> from xmlrpc.client import ServerProxy
>>> s = ServerProxy('https://localhost:15000', allow_none=True)
>>> s.set('foo','bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
>>> s.get('foo')
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')
>>> s.exists('spam')
False
>>>
```

对于SSL客户端来讲一个比较复杂的问题是如何确认服务器证书或为服务器提供客户端认证（比如客户端证书）。 不幸的是，暂时还没有一个标准方法来解决这个问题，需要自己去研究。 不过，下面给出一个例子，用来建立一个安全的XML-RPC连接来确认服务器证书：

```
from xmlrpc.client import SafeTransport, ServerProxy
import ssl

class VerifyCertSafeTransport(SafeTransport):
    def __init__(self, cafile, certfile=None, keyfile=None):
        SafeTransport.__init__(self)
        self._ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
        self._ssl_context.load_verify_locations(cafile)
        if certfile:
            self._ssl_context.load_cert_chain(certfile, keyfile)
        self._ssl_context.verify_mode = ssl.CERT_REQUIRED

    def make_connection(self, host):
        # Items in the passed dictionary are passed as keyword
        # arguments to the http.client.HTTPSConnection() constructor.
        # The context argument allows an ssl.SSLContext instance to
        # be passed with information about the SSL configuration
        s = super().make_connection((host, {'context': self._ssl_context}))

        return s

# Create the client proxy
s = ServerProxy('https://localhost:15000',
                transport=VerifyCertSafeTransport('server_cert.pem'),
                allow_none=True)
```

服务器将证书发送给客户端，客户端来确认它的合法性。这种确认可以是相互的。 如果服务器想要确认客户端，可以将服务器启动代码修改如下：

```
if __name__ == '__main__':
    KEYFILE='server_key.pem'   # Private key of the server
    CERTFILE='server_cert.pem' # Server certificate
    CA_CERTS='client_cert.pem' # Certificates of accepted clients

    kvserv = KeyValueServer(('', 15000),
                            keyfile=KEYFILE,
```

```
                            certfile=CERTFILE,
                            ca_certs=CA_CERTS,
                            cert_reqs=ssl.CERT_REQUIRED,
                            )
    kvserv.serve_forever()
```

为了让XML-RPC客户端发送证书，修改 `ServerProxy` 的初始化代码如下：

```
# Create the client proxy
s = ServerProxy('https://localhost:15000',
                transport=VerifyCertSafeTransport('server_cert.pem',
                                                  'client_cert.pem',
                                                  'client_key.pem'),
                allow_none=True)
```

讨论¶

试着去运行本节的代码能测试你的系统配置能力和理解SSL。 可能最大的挑战是如何一步步的获取初始配置key、证书和其他所需依赖。

我解释下到底需要啥，每一个SSL连接终端一般都会有一个私钥和一个签名证书文件。 这个证书包含了公钥并在每一次连接的时候都会发送给对方。 对于公共服务器，它们的证书通常是被权威证书机构比如Verisign、Equifax或其他类似机构（需要付费的）签名过的。 为了确认服务器签名，客户端回保存一份包含了信任授权机构的证书列表文件。 例如，web浏览器保存了主要的认证机构的证书，并使用它来为每一个HTTPS连接确认证书的合法性。 对本小节示例而言，只是为了测试，我们可以创建自签名的证书，下面是主要步骤：

::

bash % openssl req -new -x509 -days 365 -nodes -out server_cert.pem
-keyout server_key.pem

Generating a 1024 bit RSA private key ……++++++ …++++++

writing new private key to ‘server_key.pem’

You are about to be asked to enter information that will be incorporated into your certificate request. What you are about to enter is what is called a Distinguished Name or a DN. There are quite a few fields but you can leave some blank For some fields there will be a default value, If you enter ‘.’, the field will be left blank.

Country Name (2 letter code) [AU]:US State or Province Name (full name) [Some-State]:Illinois Locality Name (eg, city) []:Chicago Organization Name (eg, company) [Internet Widgits Pty Ltd]:Dabeaz, LLC Organizational Unit Name (eg, section) []: Common Name (eg, YOUR name) []:localhost Email Address []: bash %

在创建证书的时候，各个值的设定可以是任意的，但是”Common Name“的值通常要包含服务器的DNS主机名。 如果你只是在本机测试，那么就使用”localhost“，否则使用服务器的域名。

::

—–BEGIN RSA PRIVATE KEY—– MIICXQIBAAKBgQCZrCNLoEyAKF+f9UNcFaz5Osa6jf7qkbUl8si5xQrY3ZYC7juu
nL1dZLn/VbEFIITaUOgvBtPv1qUWTJGwga62VSG1oFE0ODIx3g2Nh4sRf+rySsx2
L4442nx0z4O5vJQ7k6eRNHAZUUnCL50+YvjyLyt7ryLSjSuKhCcJsbZgPwIDAQAB
AoGAB5evrr7eyL4160tM5rHTeATlaLY3UBOe5Z8XN8Z6gLiB/ucSX9AysviVD/6F
3oD6z2aL8jbeJc1vHqjt0dC2dwwm32vVl8mRdyoAsQpWmiqXrkvP4Bsl04VpBeHw
Qt8xNSW9SFhceL3LEvw9M8i9MV39viih1ILyH8OuHdvJyFECQQDLEjl2d2ppxND9
PoLqVFAirDfX2JnLTdWbc+M11a9Jdn3hKF8TcxfEnFVs5Gav1MusicY5KB0ylYPb
YbTvqKc7AkEAwbnRBO2VYEZsJZp2X0IZqP9ovWokkpYx+PE4+c6MySDgaMcigL7v
WDIHJG1CHudD09GbqENasDzyb2HAIW4CzQJBAKDdkv+xoW6gJx42Auc2WzTcUHCA
eXR/+BLpPrhKykzbvOQ8YvS5W764SUO1u1LWs3G+wnRMvrRvlMCZKgggBjkCQQCG
Jewto2+a+WkOKQXrNNScCDE5aPTmZQc5waCYq4UmCZQcOjkUOiN3ST1U5iuxRqfb
V/yX6fw0qh+fLWtkOs/JAkA+okMSxZwqRtfgOFGBfwQ8/iKrnizeanTQ3L6scFXI
CHZXdJ3XQ6qUmNxNn7iJ7S/LDawo1QfWkCfD9FYoxBlg —–END RSA PRIVATE KEY—–

服务器证书文件server_cert.pem内容类似下面这样：

::

```
——BEGIN CERTIFICATE——
MIIC+DCCAmGgAwIBAgIJAPMd+vi45js3MA0GCSqGSIb3DQEBBQUAMFwxCzAJBgNV
BAYTAlVTMREwDwYDVQQIEwhJbGxpbm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIG
A1UEChMLRGFiZWF6LCBMTEMxEjAQBgNVBAMTCWxvY2FsaG9zdDAeFw0xMzAxMTEx
ODQyMjdaFw0xNDAxMTExODQyMjdaMFwxCzAJBgNVBAYTAlVTMREwDwYDVQQIEwhJ
bGxpbm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIGA1UEChMLRGFiZWF6LCBMTEMx
EjAQBgNVBAMTCWxvY2FsaG9zdDCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA
mawjS6BMgChfn/VDXBWs+TrGuo3+6pG1JfLIucUK2N2WAu47rpy9XWS5/1WxBSCE
2lDoLwbT79alFkyRsIGutlUhtaBRNDgyMd4NjYeLEX/q8krMdi+OONp8dM+DubyU
O5OnkTRwGVFJwi+dPmL48i8re68i0o0rioQnCbG2YD8CAwEAAaOBwTCBvjAdBgNV
HQ4EFgQUrtoLHHgXiDZTr26NMmgKJLJLFtIwgY4GA1UdIwSBhjCBg4AUrtoLHHgX
iDZTr26NMmgKJLJLFtKhYKReMFwxCzAJBgNVBAYTAlVTMREwDwYDVQQIEwhJbGxp
bm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIGA1UEChMLRGFiZWF6LCBMTEMxEjAQ
BgNVBAMTCWxvY2FsaG9zdIIJAPMd+vi45js3MAwGA1UdEwQFMAMBAf8wDQYJKoZI
hvcNAQEFBQADgYEAFci+dqvMG4xF8UTnbGVvZJPIzJDRee6Nbt6AHQo9pOdAIMAu
WsGCplSOaDNdKKzl+b2UT2Zp3AIW4Qd51bouSNnR4M/gnr9ZD1ZctFd3jS+C5XRp
D3vvcW5lAnCCC80P6rXy7d7hTeFu5EYKtRGXNvVNd/06NALGDflrrOwxF3Y= ——END CERTIFICATE——
```

在服务器端代码中，私钥和证书文件会被传给SSL相关的包装函数。证书来自于客户端， 私钥应该在保存在服务器中，并加以安全保护。

在客户端代码中，需要保存一个合法证书授权文件来确认服务器证书。 如果你没有这个文件，你可以在客户端复制一份服务器的证书并使用它来确认。 连接建立后，服务器会提供它的证书，然后你就能使用已经保存的证书来确认它是否正确。

服务器也能选择是否要确认客户端的身份。如果要这样做的话，客户端需要有自己的私钥和认证文件。 服务器也需要保存一个被信任证书授权文件来确认客户端证书。

如果你要在真实环境中为你的网络服务加上SSL的支持，这小节只是一个入门介绍而已。 你还应该参考其他的文档，做好花费不少时间来测试它正常工作的准备。反正，就是得慢慢折腾吧~^_^

11.11 进程间传递Socket文件描述符¶

问题¶

你有多个Python解释器进程在同时运行，你想将某个打开的文件描述符从一个解释器传递给另外一个。 比如，假设有个服务器进程相应连接请求，但是实际的相应逻辑是在另一个解释器中执行的。

解决方案¶

为了在多个进程中传递文件描述符，你首先需要将它们连接到一起。在Unix机器上，你可能需要使用Unix域套接字，而在windows上面你需要使用命名管道。不过你无需真的需要去操作这些底层， 通常使用 `multiprocessing` 模块来创建这样的连接会更容易一些。

一旦一个连接被创建，你可以使用 `multiprocessing.reduction` 中的 `send_handle()` 和 `recv_handle()` 函数在不同的处理器直接传递文件描述符。 下面的例子演示了最基本的用法：

```
import multiprocessing
from multiprocessing.reduction import recv_handle, send_handle
import socket

def worker(in_p, out_p):
    out_p.close()
    while True:
        fd = recv_handle(in_p)
        print('CHILD: GOT FD', fd)
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM, fileno=fd) as s:
            while True:
                msg = s.recv(1024)
                if not msg:
                    break
                print('CHILD: RECV {!r}'.format(msg))
                s.send(msg)

def server(address, in_p, out_p, worker_pid):
    in_p.close()
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(address)
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)
        send_handle(out_p, client.fileno(), worker_pid)
        client.close()

if __name__ == '__main__':
    c1, c2 = multiprocessing.Pipe()
    worker_p = multiprocessing.Process(target=worker, args=(c1,c2))
    worker_p.start()

    server_p = multiprocessing.Process(target=server,
                args=(('', 15000), c1, c2, worker_p.pid))
    server_p.start()

    c1.close()
    c2.close()
```

在这个例子中，两个进程被创建并通过一个 `multiprocessing` 管道连接起来。 服务器进程打开一个socket并等待客户端连接请求。 工作进程仅仅使用 `recv_handle()` 在管道上面等待接收一个文件描述符。 当服务器接收到一个连接，它将产生的socket文件描述符通过 `send_handle()` 传递给工作进程。 工作进程接收到socket后向客户端回应数据，然后此次连接关闭。

如果你使用Telnet或类似工具连接到服务器，下面是一个演示例子：

bash % python3 passfd.py SERVER: Got connection from ('127.0.0.1', 55543) CHILD: GOT FD 7 CHILD: RECV b'Hellorn' CHILD: RECV b'Worldrn'

此例最重要的部分是服务器接收到的客户端socket实际上被另外一个不同的进程处理。 服务器仅仅只是将其转手并关闭此连接，然后等待下一个连接。

讨论¶

对于大部分程序员来讲在不同进程之间传递文件描述符好像没什么必要。 但是，有时候它是构建一个可扩展系统的很有用的工具。例如，在一个多核机器上面， 你可以有多个Python解释器实例，将文件描述符传递给其它解释器来实现负载均衡。

`send_handle()` 和 `recv_handle()` 函数只能够用于 `multiprocessing` 连接。 使用它们来代替管道的使用（参考11.7节），只要你使用的是Unix域套接字或Windows管道。 例如，你可以让服务器和工作者各自以单独的程序来启动。下面是服务器的实现例子：

```
# servermp.py
from multiprocessing.connection import Listener
from multiprocessing.reduction import send_handle
import socket

def server(work_address, port):
    # Wait for the worker to connect
    work_serv = Listener(work_address, authkey=b'peekaboo')
    worker = work_serv.accept()
    worker_pid = worker.recv()

    # Now run a TCP/IP server and send clients to worker
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(('', port))
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)

        send_handle(worker, client.fileno(), worker_pid)
        client.close()

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: server.py server_address port', file=sys.stderr)
        raise SystemExit(1)

    server(sys.argv[1], int(sys.argv[2]))
```

运行这个服务器，只需要执行 *python3 servermp.py /tmp/servconn 15000* ，下面是相应的工作者代码：

```
# workermp.py

from multiprocessing.connection import Client
from multiprocessing.reduction import recv_handle
import os
from socket import socket, AF_INET, SOCK_STREAM

def worker(server_address):
    serv = Client(server_address, authkey=b'peekaboo')
    serv.send(os.getpid())
    while True:
        fd = recv_handle(serv)
        print('WORKER: GOT FD', fd)
        with socket(AF_INET, SOCK_STREAM, fileno=fd) as client:
            while True:
                msg = client.recv(1024)
                if not msg:
                    break
```

```
                print('WORKER: RECV {!r}'.format(msg))
                client.send(msg)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print('Usage: worker.py server_address', file=sys.stderr)
        raise SystemExit(1)

    worker(sys.argv[1])
```

要运行工作者，执行执行命令 *python3 workermp.py /tmp/servconn* . 效果跟使用Pipe()例子是完全一样的。 文件描述符的传递会涉及到UNIX域套接字的创建和套接字的 `sendmsg()` 方法。 不过这种技术并不常见，下面是使用套接字来传递描述符的另外一种实现：

```
# server.py
import socket

import struct

def send_fd(sock, fd):
    '''
    Send a single file descriptor.
    '''
    sock.sendmsg([b'x'],
                 [(socket.SOL_SOCKET, socket.SCM_RIGHTS, struct.pack('i', fd))])
    ack = sock.recv(2)
    assert ack == b'OK'

def server(work_address, port):
    # Wait for the worker to connect
    work_serv = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    work_serv.bind(work_address)
    work_serv.listen(1)
    worker, addr = work_serv.accept()

    # Now run a TCP/IP server and send clients to worker
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(('',port))
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)
        send_fd(worker, client.fileno())
        client.close()

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: server.py server_address port', file=sys.stderr)
        raise SystemExit(1)

    server(sys.argv[1], int(sys.argv[2]))
```

下面是使用套接字的工作者实现：

```
# worker.py
import socket
import struct

def recv_fd(sock):
    '''
    Receive a single file descriptor
    '''
    msg, ancdata, flags, addr = sock.recvmsg(1,
                                     socket.CMSG_LEN(struct.calcsize('i')))

    cmsg_level, cmsg_type, cmsg_data = ancdata[0]
    assert cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS
```

```
    sock.sendall(b'OK')

    return struct.unpack('i', cmsg_data)[0]

def worker(server_address):
    serv = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    serv.connect(server_address)
    while True:
        fd = recv_fd(serv)
        print('WORKER: GOT FD', fd)
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM, fileno=fd) as client:
            while True:
                msg = client.recv(1024)
                if not msg:
                    break
                print('WORKER: RECV {!r}'.format(msg))
                client.send(msg)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print('Usage: worker.py server_address', file=sys.stderr)
        raise SystemExit(1)

    worker(sys.argv[1])
```

如果你想在你的程序中传递文件描述符，建议你参阅其他一些更加高级的文档， 比如 `Unix Network Programming by W. Richard Stevens (Prentice Hall, 1990)`. 在Windows上传递文件描述符跟Unix是不一样的，建议你研究下 `multiprocessing.reduction` 中的源代码看看其工作原理。

11.12 理解事件驱动的IO¶

问题¶

你应该已经听过基于事件驱动或异步I/O的包，但是你还不能完全理解它的底层到底是怎样工作的， 或者是如果使用它的话会对你的程序产生什么影响。

解决方案¶

事件驱动I/O本质上来讲就是将基本I/O操作（比如读和写）转化为你程序需要处理的事件。 例如，当数据在某个socket上被接受后，它会转换成一个 `receive` 事件，然后被你定义的回调方法或函数来处理。 作为一个可能的起始点，一个事件驱动的框架可能会以一个实现了一系列基本事件处理器方法的基类开始：

```
class EventHandler:
    def fileno(self):
        'Return the associated file descriptor'
        raise NotImplemented('must implement')

    def wants_to_receive(self):
        'Return True if receiving is allowed'
        return False

    def handle_receive(self):
        'Perform the receive operation'
        pass

    def wants_to_send(self):
        'Return True if sending is requested'
        return False

    def handle_send(self):
        'Send outgoing data'
        pass
```

这个类的实例作为插件被放入类似下面这样的事件循环中：

```
import select

def event_loop(handlers):
    while True:
        wants_recv = [h for h in handlers if h.wants_to_receive()]
        wants_send = [h for h in handlers if h.wants_to_send()]
        can_recv, can_send, _ = select.select(wants_recv, wants_send, [])
        for h in can_recv:
            h.handle_receive()
        for h in can_send:
            h.handle_send()
```

事件循环的关键部分是 `select()` 调用，它会不断轮询文件描述符从而激活它。 在调用 `select()` 之前，事件循环会询问所有的处理器来决定哪一个想接受或发生。 然后它将结果列表提供给 `select()` 。然后 `select()` 返回准备接受或发送的对象组成的列表。 然后相应的 `handle_receive()` 或 `handle_send()` 方法被触发。

编写应用程序的时候，`EventHandler` 的实例会被创建。例如，下面是两个简单的基于UDP网络服务的处理器例子：

```
import socket
import time

class UDPServer(EventHandler):
    def __init__(self, address):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind(address)

    def fileno(self):
        return self.sock.fileno()
```

```
    def wants_to_receive(self):
        return True

class UDPTimeServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(1)
        self.sock.sendto(time.ctime().encode('ascii'), addr)

class UDPEchoServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(8192)
        self.sock.sendto(msg, addr)

if __name__ == '__main__':
    handlers = [ UDPTimeServer(('',14000)), UDPEchoServer(('',15000))  ]
    event_loop(handlers)
```

测试这段代码，试着从另外一个Python解释器连接它：

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'',('localhost',14000))
0
>>> s.recvfrom(128)
(b'Tue Sep 18 14:29:23 2012', ('127.0.0.1', 14000))
>>> s.sendto(b'Hello',('localhost',15000))
5
>>> s.recvfrom(128)
(b'Hello', ('127.0.0.1', 15000))
>>>
```

实现一个TCP服务器会更加复杂一点，因为每一个客户端都要初始化一个新的处理器对象。 下面是一个TCP应答客户端例子：

```
class TCPServer(EventHandler):
    def __init__(self, address, client_handler, handler_list):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
        self.sock.bind(address)
        self.sock.listen(1)
        self.client_handler = client_handler
        self.handler_list = handler_list

    def fileno(self):
        return self.sock.fileno()

    def wants_to_receive(self):
        return True

    def handle_receive(self):
        client, addr = self.sock.accept()
        # Add the client to the event loop's handler list
        self.handler_list.append(self.client_handler(client, self.handler_list))

class TCPClient(EventHandler):
    def __init__(self, sock, handler_list):
        self.sock = sock
        self.handler_list = handler_list
        self.outgoing = bytearray()

    def fileno(self):
        return self.sock.fileno()

    def close(self):
        self.sock.close()
        # Remove myself from the event loop's handler list
        self.handler_list.remove(self)

    def wants_to_send(self):
        return True if self.outgoing else False
```

```
    def handle_send(self):
        nsent = self.sock.send(self.outgoing)
        self.outgoing = self.outgoing[nsent:]

class TCPEchoClient(TCPClient):
    def wants_to_receive(self):
        return True

    def handle_receive(self):
        data = self.sock.recv(8192)
        if not data:
            self.close()
        else:
            self.outgoing.extend(data)

if __name__ == '__main__':
   handlers = []
   handlers.append(TCPServer(('',16000), TCPEchoClient, handlers))
   event_loop(handlers)
```

TCP例子的关键点是从处理器中列表增加和删除客户端的操作。 对每一个连接，一个新的处理器被创建并加到列表中。当连接被关闭后，每个客户端负责将其从列表中删除。 如果你运行程序并试着用Telnet或类似工具连接，它会将你发送的消息回显给你。并且它能很轻松的处理多客户端连接。

讨论¶

实际上所有的事件驱动框架原理跟上面的例子相差无几。实际的实现细节和软件架构可能不一样， 但是在最核心的部分，都会有一个轮询的循环来检查活动socket，并执行响应操作。

事件驱动I/O的一个可能好处是它能处理非常大的并发连接，而不需要使用多线程或多进程。 也就是说，`select()` 调用（或其他等效的）能监听大量的socket并响应它们中任何一个产生事件的。 在循环中一次处理一个事件，并不需要其他的并发机制。

事件驱动I/O的缺点是没有真正的同步机制。 如果任何事件处理器方法阻塞或执行一个耗时计算，它会阻塞所有的处理进程。 调用那些并不是事件驱动风格的库函数也会有问题，同样要是某些库函数调用会阻塞，那么也会导致整个事件循环停止。

对于阻塞或耗时计算的问题可以通过将事件发送个其他单独的线程或进程来处理。 不过，在事件循环中引入多线程和多进程是比较棘手的， 下面的例子演示了如何使用 `concurrent.futures` 模块来实现：

```
from concurrent.futures import ThreadPoolExecutor
import os

class ThreadPoolHandler(EventHandler):
    def __init__(self, nworkers):
        if os.name == 'posix':
            self.signal_done_sock, self.done_sock = socket.socketpair()
        else:
            server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server.bind(('127.0.0.1', 0))
            server.listen(1)
            self.signal_done_sock = socket.socket(socket.AF_INET,
                                                  socket.SOCK_STREAM)
            self.signal_done_sock.connect(server.getsockname())
            self.done_sock, _ = server.accept()
            server.close()

        self.pending = []
        self.pool = ThreadPoolExecutor(nworkers)

    def fileno(self):
        return self.done_sock.fileno()

    # Callback that executes when the thread is done
    def _complete(self, callback, r):
```

```
        self.pending.append((callback, r.result()))
        self.signal_done_sock.send(b'x')

    # Run a function in a thread pool
    def run(self, func, args=(), kwargs={},*,callback):
        r = self.pool.submit(func, *args, **kwargs)
        r.add_done_callback(lambda r: self._complete(callback, r))

    def wants_to_receive(self):
        return True

    # Run callback functions of completed work
    def handle_receive(self):
        # Invoke all pending callback functions
        for callback, result in self.pending:
            callback(result)
            self.done_sock.recv(1)
        self.pending = []
```

在代码中，`run()` 方法被用来将工作提交给回调函数池，处理完成后被激发。 实际工作被提交给 `ThreadPoolExecutor` 实例。 不过一个难点是协调计算结果和事件循环，为了解决它，我们创建了一对socket并将其作为某种信号量机制来使用。 当线程池完成工作后，它会执行类中的 `_complete()` 方法。 这个方法再某个socket上写入字节之前会讲挂起的回调函数和结果放入队列中。 `fileno()` 方法返回另外的那个socket。 因此，这个字节被写入时，它会通知事件循环，然后 `handle_receive()` 方法被激活并为所有之前提交的工作执行回调函数。 坦白讲，说了这么多连我自己都晕了。 下面是一个简单的服务器，演示了如何使用线程池来实现耗时的计算：

```
# A really bad Fibonacci implementation
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

class UDPFibServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(128)
        n = int(msg)
        pool.run(fib, (n,), callback=lambda r: self.respond(r, addr))

    def respond(self, result, addr):
        self.sock.sendto(str(result).encode('ascii'), addr)

if __name__ == '__main__':
    pool = ThreadPoolHandler(16)
    handlers = [ pool, UDPFibServer(('',16000))]
    event_loop(handlers)
```

运行这个服务器，然后试着用其它Python程序来测试它：

```
from socket import *
sock = socket(AF_INET, SOCK_DGRAM)
for x in range(40):
    sock.sendto(str(x).encode('ascii'), ('localhost', 16000))
    resp = sock.recvfrom(8192)
    print(resp[0])
```

你应该能在不同窗口中重复的执行这个程序，并且不会影响到其他程序，尽管当数字便越来越大时候它会变得越来越慢。

已经阅读完了这一小节，那么你应该使用这里的代码吗？也许不会。你应该选择一个可以完成同样任务的高级框架。不过，如果你理解了基本原理，你就能理解这些框架所使用的核心技术。 作为对回调函数编程的替代，事件驱动编码有时候会使用到协程，参考12.12小节的一个例子。

11.13 发送与接收大型数组¶

问题¶

你要通过网络连接发送和接受连续数据的大型数组，并尽量减少数据的复制操作。

解决方案¶

下面的函数利用 `memoryviews` 来发送和接受大数组：

```
# zerocopy.py

def send_from(arr, dest):
    view = memoryview(arr).cast('B')
    while len(view):
        nsent = dest.send(view)
        view = view[nsent:]

def recv_into(arr, source):
    view = memoryview(arr).cast('B')
    while len(view):
        nrecv = source.recv_into(view)
        view = view[nrecv:]
```

为了测试程序，首先创建一个通过socket连接的服务器和客户端程序：

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.bind(('', 25000))
>>> s.listen(1)
>>> c,a = s.accept()
>>>
```

在客户端（另外一个解释器中）：

```
>>> from socket import *
>>> c = socket(AF_INET, SOCK_STREAM)
>>> c.connect(('localhost', 25000))
>>>
```

本节的目标是你能通过连接传输一个超大数组。这种情况的话，可以通过 `array` 模块或 `numpy` 模块来创建数组：

```
# Server
>>> import numpy
>>> a = numpy.arange(0.0, 50000000.0)
>>> send_from(a, c)
>>>

# Client
>>> import numpy
>>> a = numpy.zeros(shape=50000000, dtype=float)
>>> a[0:10]
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
>>> recv_into(a, c)
>>> a[0:10]
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>>
```

讨论¶

在数据密集型分布式计算和平行计算程序中，自己写程序来实现发送/接受大量数据并不常见。 不过，要是你确实想这样做，你可能需要将你的数据转换成原始字节，以便给低层的网络函数使用。 你可能还需要将数据切割成多个块，因为大部分和网络相关的函数并不能一次性发送或接受超大数据块。

一种方法是使用某种机制序列化数据——可能将其转换成一个字节字符串。 不过，这样最终会创建数据的一个复制。 就算你只是零碎的做这些，你的代码最终还是会有大量的小型复制操作。

本节通过使用内存视图展示了一些魔法操作。 本质上，一个内存视图就是一个已存在数组的覆盖层。不仅仅是那样，内存视图还能以不同的方式转换成不同类型来表现数据。 这个就是下面这个语句的目的：

```
view = memoryview(arr).cast('B')
```

它接受一个数组 arr并将其转换为一个无符号字节的内存视图。这个视图能被传递给socket相关函数， 比如 `socket.send()` 或 `send.recv_into()` 。 在内部，这些方法能够直接操作这个内存区域。例如，`sock.send()` 直接从内存中发生数据而不需要复制。 `send.recv_into()` 使用这个内存区域作为接受操作的输入缓冲区。

剩下的一个难点就是socket函数可能只操作部分数据。 通常来讲，我们得使用很多不同的 `send()` 和 `recv_into()` 来传输整个数组。 不用担心，每次操作后，视图会通过发送或接受字节数量被切割成新的视图。 新的视图同样也是内存覆盖层。因此，还是没有任何的复制操作。

这里有个问题就是接受者必须事先知道有多少数据要被发送， 以便它能预分配一个数组或者确保它能将接受的数据放入一个已经存在的数组中。 如果没办法知道的话，发送者就得先将数据大小发送过来，然后再发送实际的数组数据。

11.2 创建TCP服务器¶

问题¶

你想实现一个服务器，通过TCP协议和客户端通信。

解决方案¶

创建一个TCP服务器的一个简单方法是使用 `socketserver` 库。例如，下面是一个简单的应答服务器：

```
from socketserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        while True:

            msg = self.request.recv(8192)
            if not msg:
                break
            self.request.send(msg)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

在这段代码中，你定义了一个特殊的处理类，实现了一个 `handle()` 方法，用来为客户端连接服务。 `request` 属性是客户端socket，`client_address` 有客户端地址。 为了测试这个服务器，运行它并打开另外一个Python进程连接这个服务器：

```
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect(('localhost', 20000))
>>> s.send(b'Hello')
5
>>> s.recv(8192)
b'Hello'
>>>
```

很多时候，可以很容易的定义一个不同的处理器。下面是一个使用 `StreamRequestHandler` 基类将一个类文件接口放置在底层socket上的例子：

```
from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # self.rfile is a file-like object for reading
        for line in self.rfile:
            # self.wfile is a file-like object for writing
            self.wfile.write(line)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

讨论¶

`socketserver` 可以让我们很容易的创建简单的TCP服务器。 但是，你需要注意的是，默认情况下这种服务器是单线程的，一次只能为一个客户端连接服务。 如果你想处理多个客户端，可以初始化一个 `ForkingTCPServer` 或者是 `ThreadingTCPServer` 对象。例如：

```
from socketserver import ThreadingTCPServer
```

```
if __name__ == '__main__':
    serv = ThreadingTCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

使用fork或线程服务器有个潜在问题就是它们会为每个客户端连接创建一个新的进程或线程。 由于客户端连接数是没有限制的，因此一个恶意的黑客可以同时发送大量的连接让你的服务器奔溃。

如果你担心这个问题，你可以创建一个预先分配大小的工作线程池或进程池。 你先创建一个普通的非线程服务器，然后在一个线程池中使用 `serve_forever()` 方法来启动它们。

```
if __name__ == '__main__':
    from threading import Thread
    NWORKERS = 16
    serv = TCPServer(('', 20000), EchoHandler)
    for n in range(NWORKERS):
        t = Thread(target=serv.serve_forever)
        t.daemon = True
        t.start()
    serv.serve_forever()
```

一般来讲，一个 `TCPServer` 在实例化的时候会绑定并激活相应的 `socket` 。 不过，有时候你想通过设置某些选项去调整底下的 *socket*`，可以设置参数 `bind_and_activate=False` 。如下：

```
if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler, bind_and_activate=False)
    # Set up various socket options
    serv.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    # Bind and activate
    serv.server_bind()
    serv.server_activate()
    serv.serve_forever()
```

上面的 `socket` 选项是一个非常普遍的配置项，它允许服务器重新绑定一个之前使用过的端口号。 由于要被经常使用到，它被放置到类变量中，可以直接在 `TCPServer` 上面设置。 在实例化服务器的时候去设置它的值，如下所示：

```
if __name__ == '__main__':
    TCPServer.allow_reuse_address = True
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

在上面示例中，我们演示了两种不同的处理器基类（ `BaseRequestHandler` 和 `StreamRequestHandler` ）。 `StreamRequestHandler` 更加灵活点，能通过设置其他的类变量来支持一些新的特性。比如：

```
import socket

class EchoHandler(StreamRequestHandler):
    # Optional settings (defaults shown)
    timeout = 5                     # Timeout on all socket operations
    rbufsize = -1                   # Read buffer size
    wbufsize = 0                    # Write buffer size
    disable_nagle_algorithm = False  # Sets TCP_NODELAY socket option
    def handle(self):
        print('Got connection from', self.client_address)
        try:
            for line in self.rfile:
                # self.wfile is a file-like object for writing
                self.wfile.write(line)
        except socket.timeout:
            print('Timed out!')
```

最后，还需要注意的是绝大部分Python的高层网络模块（比如HTTP、XML-RPC等）都是建立在 `socketserver` 功能之上。 也就是说，直接使用 `socket` 库来实现服务器也并不是很难。 下面是一个使用 `socket` 直接编程实现的一个服务器简单例子：

```
from socket import socket, AF_INET, SOCK_STREAM

def echo_handler(address, client_sock):
```

```
    print('Got connection from {}'.format(address))
    while True:
        msg = client_sock.recv(8192)
        if not msg:
            break
        client_sock.sendall(msg)
    client_sock.close()

def echo_server(address, backlog=5):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(backlog)
    while True:
        client_sock, client_addr = sock.accept()
        echo_handler(client_addr, client_sock)

if __name__ == '__main__':
    echo_server(('', 20000))
```

11.3 创建UDP服务器¶

问题¶

你想实现一个基于UDP协议的服务器来与客户端通信。

解决方案¶

跟TCP一样，UDP服务器也可以通过使用 `socketserver` 库很容易的被创建。 例如，下面是一个简单的时间服务器：

```
from socketserver import BaseRequestHandler, UDPServer
import time

class TimeHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # Get message and client socket
        msg, sock = self.request
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
    serv = UDPServer(('', 20000), TimeHandler)
    serv.serve_forever()
```

跟之前一样，你先定义一个实现 `handle()` 特殊方法的类，为客户端连接服务。 这个类的 `request` 属性是一个包含了数据报和底层socket对象的元组。`client_address` 包含了客户端地址。

我们来测试下这个服务器，首先运行它，然后打开另外一个Python进程向服务器发送消息：

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'', ('localhost', 20000))
0
>>> s.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 20000))
>>>
```

讨论¶

一个典型的UDP服务器接收到达的数据报(消息)和客户端地址。如果服务器需要做应答， 它要给客户端回发一个数据报。对于数据报的传送， 你应该使用socket的 `sendto()` 和 `recvfrom()` 方法。 尽管传统的 `send()` 和 `recv()` 也可以达到同样的效果， 但是前面的两个方法对于UDP连接而言更普遍。

由于没有底层的连接，UPD服务器相对于TCP服务器来讲实现起来更加简单。 不过，UDP天生是不可靠的（因为通信没有建立连接，消息可能丢失）。 因此需要由你自己来决定该怎样处理丢失消息的情况。这个已经不在本书讨论范围内了， 不过通常来说，如果可靠性对于你程序很重要，你需要借助于序列号、重试、超时以及一些其他方法来保证。UDP通常被用在那些对于可靠传输要求不是很高的场合。例如，在实时应用如多媒体流以及游戏领域， 无需返回恢复丢失的数据包（程序只需简单的忽略它并继续向前运行）。

`UDPServer` 类是单线程的，也就是说一次只能为一个客户端连接服务。 实际使用中，这个无论是对于UDP还是TCP都不是什么大问题。 如果你想要并发操作，可以实例化一个 `ForkingUDPServer` 或 `ThreadingUDPServer` 对象：

```
from socketserver import ThreadingUDPServer

   if __name__ == '__main__':
    serv = ThreadingUDPServer(('',20000), TimeHandler)
    serv.serve_forever()
```

直接使用 `socket` 来实现一个UDP服务器也不难，下面是一个例子：

```
from socket import socket, AF_INET, SOCK_DGRAM
```

```
import time

def time_server(address):
    sock = socket(AF_INET, SOCK_DGRAM)
    sock.bind(address)
    while True:
        msg, addr = sock.recvfrom(8192)
        print('Got message from', addr)
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), addr)

if __name__ == '__main__':
    time_server(('', 20000))
```

11.4 通过CIDR地址生成对应的IP地址集¶

问题¶

你有一个CIDR网络地址比如“123.45.67.89/27”，你想将其转换成它所代表的所有IP （比如，“123.45.67.64”, “123.45.67.65”, …, “123.45.67.95”）

解决方案¶

可以使用 `ipaddress` 模块很容易的实现这样的计算。例如：

```
>>> import ipaddress
>>> net = ipaddress.ip_network('123.45.67.64/27')
>>> net
IPv4Network('123.45.67.64/27')
>>> for a in net:
...     print(a)
...
123.45.67.64
123.45.67.65
123.45.67.66
123.45.67.67
123.45.67.68
...
123.45.67.95
>>>

>>> net6 = ipaddress.ip_network('12:3456:78:90ab:cd:ef01:23:30/125')
>>> net6
IPv6Network('12:3456:78:90ab:cd:ef01:23:30/125')
>>> for a in net6:
...     print(a)
...
12:3456:78:90ab:cd:ef01:23:30
12:3456:78:90ab:cd:ef01:23:31
12:3456:78:90ab:cd:ef01:23:32
12:3456:78:90ab:cd:ef01:23:33
12:3456:78:90ab:cd:ef01:23:34
12:3456:78:90ab:cd:ef01:23:35
12:3456:78:90ab:cd:ef01:23:36
12:3456:78:90ab:cd:ef01:23:37
>>>
```

`Network` 也允许像数组一样的索引取值，例如：

```
>>> net.num_addresses
32
>>> net[0]

IPv4Address('123.45.67.64')
>>> net[1]
IPv4Address('123.45.67.65')
>>> net[-1]
IPv4Address('123.45.67.95')
>>> net[-2]
IPv4Address('123.45.67.94')
>>>
```

另外，你还可以执行网络成员检查之类的操作：

```
>>> a = ipaddress.ip_address('123.45.67.69')
>>> a in net
True
>>> b = ipaddress.ip_address('123.45.67.123')
>>> b in net
False
```

```
>>>
```

一个IP地址和网络地址能通过一个IP接口来指定，例如：

```
>>> inet = ipaddress.ip_interface('123.45.67.73/27')
>>> inet.network
IPv4Network('123.45.67.64/27')
>>> inet.ip
IPv4Address('123.45.67.73')
>>>
```

讨论¶

`ipaddress` 模块有很多类可以表示IP地址、网络和接口。 当你需要操作网络地址（比如解析、打印、验证等）的时候会很有用。

要注意的是，`ipaddress` 模块跟其他一些和网络相关的模块比如 `socket` 库交集很少。 所以，你不能使用 `IPv4Address` 的实例来代替一个地址字符串，你首先得显式的使用 `str()` 转换它。例如：

```
>>> a = ipaddress.ip_address('127.0.0.1')
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect((a, 8080))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'IPv4Address' object to str implicitly
>>> s.connect((str(a), 8080))
>>>
```

更多相关内容，请参考 An Introduction to the ipaddress Module

11.5 创建一个简单的REST接口¶

问题¶

你想使用一个简单的REST接口通过网络远程控制或访问你的应用程序，但是你又不想自己去安装一个完整的web框架。

解决方案¶

构建一个REST风格的接口最简单的方法是创建一个基于WSGI标准（PEP 3333）的很小的库，下面是一个例子：

```
# resty.py

import cgi

def notfound_404(environ, start_response):
    start_response('404 Not Found', [ ('Content-type', 'text/plain') ])
    return [b'Not Found']

class PathDispatcher:
    def __init__(self):
        self.pathmap = { }

    def __call__(self, environ, start_response):
        path = environ['PATH_INFO']
        params = cgi.FieldStorage(environ['wsgi.input'],
                                  environ=environ)
        method = environ['REQUEST_METHOD'].lower()
        environ['params'] = { key: params.getvalue(key) for key in params }
        handler = self.pathmap.get((method,path), notfound_404)
        return handler(environ, start_response)

    def register(self, method, path, function):
        self.pathmap[method.lower(), path] = function
        return function
```

为了使用这个调度器，你只需要编写不同的处理器，就像下面这样：

```
import time

_hello_resp = '''\
<html>
  <head>
     <title>Hello {name}</title>
   </head>
   <body>
     <h1>Hello {name}!</h1>
   </body>
</html>'''

def hello_world(environ, start_response):
    start_response('200 OK', [ ('Content-type','text/html')])
    params = environ['params']
    resp = _hello_resp.format(name=params.get('name'))
    yield resp.encode('utf-8')

_localtime_resp = '''\
<?xml version="1.0"?>
<time>
  <year>{t.tm_year}</year>
  <month>{t.tm_mon}</month>
  <day>{t.tm_mday}</day>
  <hour>{t.tm_hour}</hour>
  <minute>{t.tm_min}</minute>
  <second>{t.tm_sec}</second>
</time>'''
```

```
def localtime(environ, start_response):
    start_response('200 OK', [ ('Content-type', 'application/xml') ])
    resp = _localtime_resp.format(t=time.localtime())
    yield resp.encode('utf-8')

if __name__ == '__main__':
    from resty import PathDispatcher
    from wsgiref.simple_server import make_server

    # Create the dispatcher and register functions
    dispatcher = PathDispatcher()
    dispatcher.register('GET', '/hello', hello_world)
    dispatcher.register('GET', '/localtime', localtime)

    # Launch a basic server
    httpd = make_server('', 8080, dispatcher)
    print('Serving on port 8080...')
    httpd.serve_forever()
```

要测试下这个服务器，你可以使用一个浏览器或 `urllib` 和它交互。例如：

```
>>> u = urlopen('http://localhost:8080/hello?name=Guido')
>>> print(u.read().decode('utf-8'))
<html>
  <head>
     <title>Hello Guido</title>
   </head>
   <body>
     <h1>Hello Guido!</h1>
   </body>
</html>

>>> u = urlopen('http://localhost:8080/localtime')
>>> print(u.read().decode('utf-8'))
<?xml version="1.0"?>
<time>
  <year>2012</year>
  <month>11</month>
  <day>24</day>
  <hour>14</hour>
  <minute>49</minute>
  <second>17</second>
</time>
>>>
```

讨论¶

在编写REST接口时，通常都是服务于普通的HTTP请求。但是跟那些功能完整的网站相比，你通常只需要处理数据。这些数据以各种标准格式编码，比如XML、JSON或CSV。 尽管程序看上去很简单，但是以这种方式提供的API对于很多应用程序来讲是非常有用的。

例如，长期运行的程序可能会使用一个REST API来实现监控或诊断。 大数据应用程序可以使用REST来构建一个数据查询或提取系统。 REST还能用来控制硬件设备比如机器人、传感器、工厂或灯泡。 更重要的是，REST API已经被大量客户端编程环境所支持，比如Javascript, Android, iOS等。 因此，利用这种接口可以让你开发出更加复杂的应用程序。

为了实现一个简单的REST接口，你只需让你的程序代码满足Python的WSGI标准即可。 WSGI被标准库支持，同时也被绝大部分第三方web框架支持。 因此，如果你的代码遵循这个标准，在后面的使用过程中就会更加的灵活！

在WSGI中，你可以像下面这样约定的方式以一个可调用对象形式来实现你的程序。

```
import cgi

def wsgi_app(environ, start_response):
    pass
```

environ 属性是一个字典，包含了从web服务器如Apache[参考Internet RFC 3875]提供的CGI接口中获取的值。 要将这些不同的值提取出来，你可以像这么这样写：

```
def wsgi_app(environ, start_response):
    method = environ['REQUEST_METHOD']
    path = environ['PATH_INFO']
    # Parse the query parameters
    params = cgi.FieldStorage(environ['wsgi.input'], environ=environ)
```

我们展示了一些常见的值。environ['REQUEST_METHOD'] 代表请求类型如GET、POST、HEAD等。 environ['PATH_INFO'] 表示被请求资源的路径。 调用 cgi.FieldStorage() 可以从请求中提取查询参数并将它们放入一个类字典对象中以便后面使用。

start_response 参数是一个为了初始化一个请求对象而必须被调用的函数。 第一个参数是返回的HTTP状态值，第二个参数是一个(名,值)元组列表，用来构建返回的HTTP头。例如：

```
def wsgi_app(environ, start_response):
    pass
    start_response('200 OK', [('Content-type', 'text/plain')])
```

为了返回数据，一个WSGI程序必须返回一个字节字符串序列。可以像下面这样使用一个列表来完成：

```
def wsgi_app(environ, start_response):
    pass
    start_response('200 OK', [('Content-type', 'text/plain')])
    resp = []
    resp.append(b'Hello World\n')
    resp.append(b'Goodbye!\n')
    return resp
```

或者，你还可以使用 yield：

```
def wsgi_app(environ, start_response):
    pass
    start_response('200 OK', [('Content-type', 'text/plain')])
    yield b'Hello World\n'
    yield b'Goodbye!\n'
```

这里要强调的一点是最后返回的必须是字节字符串。如果返回结果包含文本字符串，必须先将其编码成字节。 当然，并没有要求你返回的一定是文本，你可以很轻松的编写一个生成图片的程序。

尽管WSGI程序通常被定义成一个函数，不过你也可以使用类实例来实现，只要它实现了合适的 __call__() 方法。例如：

```
class WSGIApplication:
    def __init__(self):
        ...
    def __call__(self, environ, start_response)
       ...
```

我们已经在上面使用这种技术创建 PathDispatcher 类。 这个分发器仅仅只是管理一个字典，将(方法,路径)对映射到处理器函数上面。 当一个请求到来时，它的方法和路径被提取出来，然后被分发到对应的处理器上面去。 另外，任何查询变量会被解析后放到一个字典中，以 environ['params'] 形式存储。 后面这个步骤太常见，所以建议你在分发器里面完成，这样可以省掉很多重复代码。 使用分发器的时候，你只需简单的创建一个实例，然后通过它注册各种WSGI形式的函数。 编写这些函数应该超级简单了，只要你遵循 start_response() 函数的编写规则，并且最后返回字节字符串即可。

当编写这种函数的时候还需注意的一点就是对于字符串模板的使用。 没人愿意写那种到处混合着 print() 函数 、XML和大量格式化操作的代码。 我们上面使用了三引号包含的预先定义好的字符串模板。 这种方式的可以让我们很容易的在以后修改输出格式(只需要修改模板本身，而不用动任何使用它的地方)。

最后，使用WSGI还有一个很重要的部分就是没有什么地方是针对特定web服务器的。 因为标准对于服务器和框架是中立的，你可以将你的程序放入任何类型服务器中。 我们使用下面的代码测试测试本节代码：

```
if __name__ == '__main__':
```

```
from wsgiref.simple_server import make_server

# Create the dispatcher and register functions
dispatcher = PathDispatcher()
pass

# Launch a basic server
httpd = make_server('', 8080, dispatcher)
print('Serving on port 8080...')
httpd.serve_forever()
```

上面代码创建了一个简单的服务器，然后你就可以来测试下你的实现是否能正常工作。 最后，当你准备进一步扩展你的程序的时候，你可以修改这个代码，让它可以为特定服务器工作。

WSGI本身是一个很小的标准。因此它并没有提供一些高级的特性比如认证、cookies、重定向等。 这些你自己实现起来也不难。不过如果你想要更多的支持，可以考虑第三方库，比如 `WebOb` 或者 `Paste`

11.6 通过XML-RPC实现简单的远程调用¶

问题¶

你想找到一个简单的方式去执行运行在远程机器上面的Python程序中的函数或方法。

解决方案¶

实现一个远程方法调用的最简单方式是使用XML-RPC。下面我们演示一下一个实现了键-值存储功能的简单服务器：

```
from xmlrpc.server import SimpleXMLRPCServer

class KeyValueServer:
    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys']
    def __init__(self, address):
        self._data = {}
        self._serv = SimpleXMLRPCServer(address, allow_none=True)
        for name in self._rpc_methods_:
            self._serv.register_function(getattr(self, name))

    def get(self, name):
        return self._data[name]

    def set(self, name, value):
        self._data[name] = value

    def delete(self, name):
        del self._data[name]

    def exists(self, name):
        return name in self._data

    def keys(self):
        return list(self._data)

    def serve_forever(self):
        self._serv.serve_forever()

# Example
if __name__ == '__main__':
    kvserv = KeyValueServer(('', 15000))
    kvserv.serve_forever()
```

下面我们从一个客户端机器上面来访问服务器：

```
>>> from xmlrpc.client import ServerProxy
>>> s = ServerProxy('http://localhost:15000', allow_none=True)
>>> s.set('foo', 'bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
>>> s.get('foo')
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')
>>> s.exists('spam')
False
>>>
```

讨论¶

XML-RPC 可以让我们很容易的构造一个简单的远程调用服务。你所需要做的仅仅是创建一个服务器实例， 通过它的方法 `register_function()` 来注册函数，然后使用方法 `serve_forever()` 启动它。 在上面我们将这些步骤放在一起写到一个类中，不够这并不是必须的。比如你还可以像下面这样创建一个服务器：

```
from xmlrpc.server import SimpleXMLRPCServer
def add(x,y):
    return x+y

serv = SimpleXMLRPCServer(('', 15000))
serv.register_function(add)
serv.serve_forever()
```

XML-RPC暴露出来的函数只能适用于部分数据类型，比如字符串、整形、列表和字典。 对于其他类型就得需要做些额外的功课了。 例如，如果你想通过 XML-RPC 传递一个对象实例，实际上只有他的实例字典被处理：

```
>>> class Point:
...     def __init__(self, x, y):
...             self.x = x
...             self.y = y
...
>>> p = Point(2, 3)
>>> s.set('foo', p)
>>> s.get('foo')
{'x': 2, 'y': 3}
>>>
```

类似的，对于二进制数据的处理也跟你想象的不太一样：

```
>>> s.set('foo', b'Hello World')
>>> s.get('foo')
<xmlrpc.client.Binary object at 0x10131d410>

>>> _.data
b'Hello World'
>>>
```

一般来讲，你不应该将 XML-RPC 服务以公共API的方式暴露出来。 对于这种情况，通常分布式应用程序会是一个更好的选择。

XML-RPC的一个缺点是它的性能。`SimpleXMLRPCServer` 的实现是单线程的， 所以它不适合于大型程序，尽管我们在11.2小节中演示过它是可以通过多线程来执行的。 另外，由于 XML-RPC 将所有数据都序列化为XML格式，所以它会比其他的方式运行的慢一些。 但是它也有优点，这种方式的编码可以被绝大部分其他编程语言支持。 通过使用这种方式，其他语言的客户端程序都能访问你的服务。

虽然XML-RPC有很多缺点，但是如果你需要快速构建一个简单远程过程调用系统的话，它仍然值得去学习的。 有时候，简单的方案就已经足够了。

11.7 在不同的Python解释器之间交互¶

问题¶

你在不同的机器上面运行着多个Python解释器实例，并希望能够在这些解释器之间通过消息来交换数据。

解决方案¶

通过使用 `multiprocessing.connection` 模块可以很容易的实现解释器之间的通信。 下面是一个简单的应答服务器例子：

```
from multiprocessing.connection import Listener
import traceback

def echo_client(conn):
    try:
        while True:
            msg = conn.recv()
            conn.send(msg)
    except EOFError:
        print('Connection closed')

def echo_server(address, authkey):
    serv = Listener(address, authkey=authkey)
    while True:
        try:
            client = serv.accept()

            echo_client(client)
        except Exception:
            traceback.print_exc()

echo_server(('', 25000), authkey=b'peekaboo')
```

然后客户端连接服务器并发送消息的简单示例：

```
>>> from multiprocessing.connection import Client
>>> c = Client(('localhost', 25000), authkey=b'peekaboo')
>>> c.send('hello')
>>> c.recv()
'hello'
>>> c.send(42)
>>> c.recv()
42
>>> c.send([1, 2, 3, 4, 5])
>>> c.recv()
[1, 2, 3, 4, 5]
>>>
```

跟底层socket不同的是，每个消息会完整保存（每一个通过send()发送的对象能通过recv()来完整接受）。 另外，所有对象会通过pickle序列化。因此，任何兼容pickle的对象都能在此连接上面被发送和接受。

讨论¶

目前有很多用来实现各种消息传输的包和函数库，比如ZeroMQ、Celery等。 你还有另外一种选择就是自己在底层socket基础之上来实现一个消息传输层。 但是你想要简单一点的方案，那么这时候 `multiprocessing.connection` 就派上用场了。 仅仅使用一些简单的语句即可实现多个解释器之间的消息通信。

如果你的解释器运行在同一台机器上面，那么你可以使用另外的通信机制，比如Unix域套接字或者是Windows命名管道。 要想使用UNIX域套接字来创建一个连接，只需简单的将地址改写一个文件名即可：

```
s = Listener('/tmp/myconn', authkey=b'peekaboo')
```

要想使用Windows命名管道来创建连接，只需像下面这样使用一个文件名：

```
s = Listener(r'\\.\pipe\myconn', authkey=b'peekaboo')
```

一个通用准则是，你不要使用 `multiprocessing` 来实现一个对外的公共服务。 `Client()` 和 `Listener()` 中的 `authkey` 参数用来认证发起连接的终端用户。 如果密钥不对会产生一个异常。此外，该模块最适合用来建立长连接（而不是大量的短连接）， 例如，两个解释器之间启动后就开始建立连接并在处理某个问题过程中会一直保持连接状态。

如果你需要对底层连接做更多的控制，比如需要支持超时、非阻塞I/O或其他类似的特性， 你最好使用另外的库或者是在高层socket上来实现这些特性。

11.8 实现远程方法调用¶

问题¶

你想在一个消息传输层如 sockets 、multiprocessing connections 或 ZeroMQ 的基础之上实现一个简单的远程过程调用（RPC）。

解决方案¶

将函数请求、参数和返回值使用pickle编码后，在不同的解释器直接传送pickle字节字符串，可以很容易的实现RPC。下面是一个简单的PRC处理器，可以被整合到一个服务器中去：

```
# rpcserver.py

import pickle
class RPCHandler:
    def __init__(self):
        self._functions = { }

    def register_function(self, func):
        self._functions[func.__name__] = func

    def handle_connection(self, connection):
        try:
            while True:
                # Receive a message
                func_name, args, kwargs = pickle.loads(connection.recv())
                # Run the RPC and send a response
                try:
                    r = self._functions[func_name](*args,**kwargs)
                    connection.send(pickle.dumps(r))
                except Exception as e:
                    connection.send(pickle.dumps(e))
        except EOFError:
             pass
```

要使用这个处理器，你需要将它加入到一个消息服务器中。你有很多种选择， 但是使用 multiprocessing 库是最简单的。下面是一个RPC服务器例子：

```
from multiprocessing.connection import Listener
from threading import Thread

def rpc_server(handler, address, authkey):
    sock = Listener(address, authkey=authkey)
    while True:
        client = sock.accept()
        t = Thread(target=handler.handle_connection, args=(client,))
        t.daemon = True
        t.start()

# Some remote functions
def add(x, y):
    return x + y

def sub(x, y):
    return x - y

# Register with a handler
handler = RPCHandler()
handler.register_function(add)
handler.register_function(sub)

# Run the server
rpc_server(handler, ('localhost', 17000), authkey=b'peekaboo')
```

为了从一个远程客户端访问服务器，你需要创建一个对应的用来传送请求的RPC代理类。例如

```
import pickle

class RPCProxy:
    def __init__(self, connection):
        self._connection = connection
    def __getattr__(self, name):
        def do_rpc(*args, **kwargs):
            self._connection.send(pickle.dumps((name, args, kwargs)))
            result = pickle.loads(self._connection.recv())
            if isinstance(result, Exception):
                raise result
            return result
        return do_rpc
```

要使用这个代理类，你需要将其包装到一个服务器的连接上面，例如：

```
>>> from multiprocessing.connection import Client
>>> c = Client(('localhost', 17000), authkey=b'peekaboo')
>>> proxy = RPCProxy(c)
>>> proxy.add(2, 3)

5
>>> proxy.sub(2, 3)
-1
>>> proxy.sub([1, 2], 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "rpcserver.py", line 37, in do_rpc
    raise result
TypeError: unsupported operand type(s) for -: 'list' and 'int'
>>>
```

要注意的是很多消息层（比如 `multiprocessing` ）已经使用pickle序列化了数据。 如果是这样的话，对 `pickle.dumps()` 和 `pickle.loads()` 的调用要去掉。

讨论¶

`RPCHandler` 和 `RPCProxy` 的基本思路是很比较简单的。 如果一个客户端想要调用一个远程函数，比如 `foo(1, 2, z=3)` ,代理类创建一个包含了函数名和参数的元组 `('foo', (1, 2), {'z': 3})` 。 这个元组被pickle序列化后通过网络连接发生出去。 这一步在 `RPCProxy` 的 `__getattr__()` 方法返回的 `do_rpc()` 闭包中完成。 服务器接收后通过pickle反序列化消息，查找函数名看看是否已经注册过，然后执行相应的函数。 执行结果(或异常)被pickle序列化后返回发送给客户端。我们的实例需要依赖 `multiprocessing` 进行通信。 不过，这种方式可以适用于其他任何消息系统。例如，如果你想在ZeroMQ之上实习RPC， 仅仅只需要将连接对象换成合适的ZeroMQ的socket对象即可。

由于底层需要依赖pickle，那么安全问题就需要考虑了 （因为一个聪明的黑客可以创建特定的消息，能够让任意函数通过pickle反序列化后被执行）。 因此你永远不要允许来自不信任或未认证的客户端的RPC。特别是你绝对不要允许来自Internet的任意机器的访问， 这种只能在内部被使用，位于防火墙后面并且不要对外暴露。

作为pickle的替代，你也许可以考虑使用JSON、XML或一些其他的编码格式来序列化消息。 例如，本机实例可以很容易的改写成JSON编码方案。还需要将 `pickle.loads()` 和 `pickle.dumps()` 替换成 `json.loads()` 和 `json.dumps()` 即可：

```
# jsonrpcserver.py
import json

class RPCHandler:
    def __init__(self):
        self._functions = { }

    def register_function(self, func):
        self._functions[func.__name__] = func

    def handle_connection(self, connection):
        try:
            while True:
```

```
                # Receive a message
                func_name, args, kwargs = json.loads(connection.recv())
                # Run the RPC and send a response
                try:
                    r = self._functions[func_name](*args,**kwargs)
                    connection.send(json.dumps(r))
                except Exception as e:
                    connection.send(json.dumps(str(e)))
        except EOFError:
             pass

# jsonrpcclient.py
import json

class RPCProxy:
    def __init__(self, connection):
        self._connection = connection
    def __getattr__(self, name):
        def do_rpc(*args, **kwargs):
            self._connection.send(json.dumps((name, args, kwargs)))
            result = json.loads(self._connection.recv())
            return result
        return do_rpc
```

实现RPC的一个比较复杂的问题是如何去处理异常。至少，当方法产生异常时服务器不应该奔溃。 因此，返回给客户端的异常所代表的含义就要好好设计了。 如果你使用pickle，异常对象实例在客户端能被反序列化并抛出。如果你使用其他的协议，那得想想另外的方法了。 不过至少，你应该在响应中返回异常字符串。我们在JSON的例子中就是使用的这种方式。

对于其他的RPC实现例子，我推荐你看看在XML-RPC中使用的 `SimpleXMLRPCServer` 和 `ServerProxy` 的实现， 也就是11.6小节中的内容。

11.9 简单的客户端认证¶

问题¶

你想在分布式系统中实现一个简单的客户端连接认证功能，又不想像SSL那样的复杂。

解决方案¶

可以利用 `hmac` 模块实现一个连接握手，从而实现一个简单而高效的认证过程。下面是代码示例：

```
import hmac
import os

def client_authenticate(connection, secret_key):
    '''
    Authenticate client to a remote service.
    connection represents a network connection.
    secret_key is a key known only to both client/server.
    '''
    message = connection.recv(32)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    connection.send(digest)

def server_authenticate(connection, secret_key):
    '''
    Request client authentication.
    '''
    message = os.urandom(32)
    connection.send(message)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    response = connection.recv(len(digest))
    return hmac.compare_digest(digest,response)
```

基本原理是当连接建立后，服务器给客户端发送一个随机的字节消息（这里例子中使用了 `os.urandom()` 返回值）。客户端和服务器同时利用hmac和一个只有双方知道的密钥来计算出一个加密哈希值。然后客户端将它计算出的摘要发送给服务器， 服务器通过比较这个值和自己计算的是否一致来决定接受或拒绝连接。摘要的比较需要使用 `hmac.compare_digest()` 函数。 使用这个函数可以避免遭到时间分析攻击，不要用简单的比较操作符（==）。 为了使用这些函数，你需要将它集成到已有的网络或消息代码中。例如，对于sockets，服务器代码应该类似下面：

```
from socket import socket, AF_INET, SOCK_STREAM

secret_key = b'peekaboo'
def echo_handler(client_sock):
    if not server_authenticate(client_sock, secret_key):
        client_sock.close()
        return
    while True:

        msg = client_sock.recv(8192)
        if not msg:
            break
        client_sock.sendall(msg)

def echo_server(address):
    s = socket(AF_INET, SOCK_STREAM)
    s.bind(address)
    s.listen(5)
    while True:
        c,a = s.accept()
        echo_handler(c)

echo_server(('', 18000))
```

Within a client, you would do this:

```
from socket import socket, AF_INET, SOCK_STREAM

secret_key = b'peekaboo'

s = socket(AF_INET, SOCK_STREAM)
s.connect(('localhost', 18000))
client_authenticate(s, secret_key)
s.send(b'Hello World')
resp = s.recv(1024)
```

讨论¶

hmac 认证的一个常见使用场景是内部消息通信系统和进程间通信。 例如，如果你编写的系统涉及到一个集群中多个处理器之间的通信， 你可以使用本节方案来确保只有被允许的进程之间才能彼此通信。 事实上，基于 hmac 的认证被 multiprocessing 模块使用来实现子进程直接的通信。

还有一点需要强调的是连接认证和加密是两码事。 认证成功之后的通信消息是以明文形式发送的，任何人只要想监听这个连接线路都能看到消息（尽管双方的密钥不会被传输）。

hmac认证算法基于哈希函数如MD5和SHA-1，关于这个在IETF RFC 2104中有详细介绍。

第十一章：网络与Web编程¶

本章是关于在网络应用和分布式应用中使用的各种主题。主题划分为使用Python编写客户端程序来访问已有的服务，以及使用Python实现网络服务端程序。也给出了一些常见的技术，用于编写涉及协同或通信的的代码。

Contents: