关于python socket

2018年10月14日 11:53

import socket

TCP连接：

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((ip, port))
```

读一次要自己写

```
BUFFER_SIZE = 1024

def rec_all(sock):
    allData = ''
    while True:
        data = sock.recv(BUFFER_SIZE)
        allData += data
        if len(data) < BUFFER_SIZE:
            return allData
```

发送一次则sock.sendall(msg),

用完需要sock.close() . 另外，一个sock被关闭之后，就不能再用了。必须重新创建。

一次发送以\r\n结尾。不发\r\n就不代表结束。这估计是FTP的要求，因为TCP是一个流，无所谓一次消息。 FTP自己定义\r\n是一个命令的终结。

TCP的server：

让系统随机分配一个端口并绑定，开始监听。

```
        sock.bind(('', 0))# 0表示系统自动分配一个端口。 非0则是直接绑定到特定端口
        ip, port = sock.getsockname()
        sock.listen(1)
```

当有人连接上来之后，connect, address = sock.accept(), 然后connect就可以当socket用。

对一个socket可以反复关闭，没关系。所以为了确保socket被关闭，可以多用sock.close()

client实现

2018年10月18日 18:56

handler接受用户的输入strip之后的结果，就是一行用户命令

FTP是一问一答模式的，
client发送一个命令，服务器会返回一条消息。

发送port xxx命令, 服务器先回复一个ok。
再发送一个实际的数据传输命令，服务器开始连接你的socket, 并给你返回一个ok
数据传输完了，服务器再返回一个汇总统计。

PORT h1,h2,h3,h4,p1,p2后面不能加空格。要求非常严格

发送pasv命令，则服务器回复一个ip + address, 客户端连上去，读完了数据，服务器返回一个统计。

谁发送的消息谁负责把这条消息引起的服务器回传消息全部接受完

一个可以用的公共server：
144.208.69.31
21
dlpuser@dlptest.com
e73jzTRTNqCN9PYAAjjn

关于数据传输阶段服务器返回的code，
可能是 150 File status okay; about to open data connection. 也就是说传retr过去的时候，服务器还没有开数据连接。
125 Data connection already open; transfer starting. 也就是说，retr传过去的时候，服务器已经开了数据连接。（就是pasv命令过去，服务器聚已经开了一个数据连接）
两种都可以

server开发环境

2018年10月19日 10:37

在src目录下，
scp ../src/* gpu106:/home/youkaichao/code/server/src ; ssh gpu106 'bash -s' < command.sh
即可部署并测试。

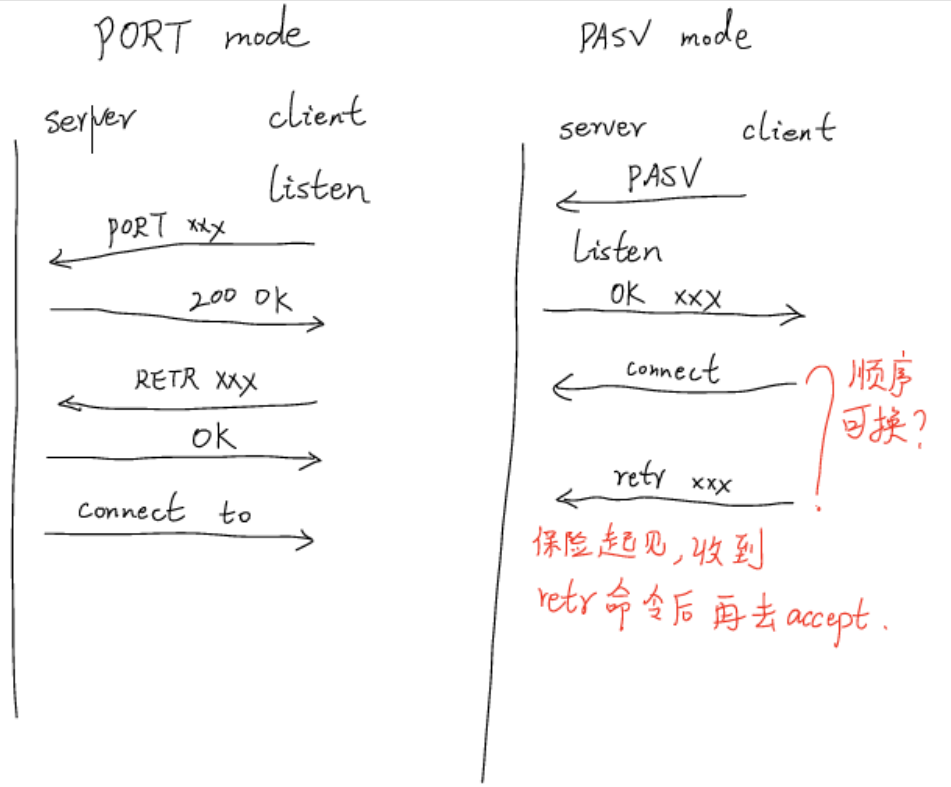
在服务器上用gdb调吧。

Linux提供的工具：
telnet ip port, 仅仅是一个client的tcp socket
nc -l port 创建一个server tcp socket
这样两个人就可以开始玩了。。。

server开发

2018年10月19日　　10:44

关于数据连接的pasv和port模式：



Berkeley Sockets API

2018年10月19日 11:39

使用一个socket:

```
#include <sys/types.h>        /* See NOTES */
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

domain :

AF_INET IPv4 Internet protocols **AF: address family**

AF_INET6 IPv6 Internet protocols

type :

SOCK_STREAM Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported. **typically TCP**

SOCK_DGRAM Supports datagrams (connectionless, unreliable messages of a fixed maximum length). **typically UDP**

protocol: 一般来讲一个domain 里面只有一个protocol, 所以写上0，让系统自己选就可以了。

这样就得到一个socket，返回fd，-1表示失败。

bind一个socket：只有server要bind，因为client并不关心使用哪个端口。

bind(fd, addr, addrlen)

然后server开始listen(fd, backlog)

对于client就简单多了，创建之后直接connect(fd, addr, addrlen)

地址的声明：

```
struct sockaddr {
 uint8_t sa_len;
 sa_family_t sa_family;
 char sa_data[22];
};
```

这是通用的地址。

对于ip地址，需要

#include <netinet/in.h>

就有

```
struct sockaddr_in {
 uint8_t sin_len;
 sa_family_t sin_family; //所有的socket都需要
 in_port_t sin_port; //把22字节的区域划分出6字节，实际存储ip地址
 struct in_addr sin_addr;
 char sin_pad[16];
};
```

sockaddr_in6用来存ipv6地址。

所以需要显式类型转换：

struct sockaddr_in addr;
(bind(fd, (struct sockaddr *) &addr, sizeof(addr))

其中ip地址本身只有一个int，而我们写的都是字符串，所以
#include <arpa/inet.h>
int inet_pton(int af, const char *src, void *dst);
inet_pton(AF_INET, “130.209.240.1”, &addr.sin_addr);

端口也需要转换：
htonl, htons, ntohl, ntohs convert values between host and network byte order
h代表host，即本机的大小端方式，n代表network，有一种大小端方式。 s和l代表short 和 long, 分别对应16位和32位

server在listen之后，
int connfd;
struct sockaddr_in cliaddr;
socklen_t cliaddrlen = sizeof(cliaddr);
...
connfd = accept(fd, &cliaddr, &cliaddrlen);
这个connfd才是用来读写的。fd是server socket file descriptor
Data read from the connection is not null terminated.

是不是listen就是已经开始监听了，后台维护了一个队列。accept才是阻塞的根源。
是的。创建server socket -> bind -> listen, 一系列动作都是非阻塞的，然后就已经起了一个server了，accept在没有人来创建连接的时候就阻塞起来了。

C里面随机分配一个端口来监听：
端口写0，然后再
```
socklen_t n = sizeof addr;
getsockname(listenfd, (struct sockaddr*)&addr, &n);
```
注意，一定要把n正确地初始化。 然后就可以用(int)(ntohs(addr.sin_port))来得到实际端口号了

如果没有读完，连接就被关闭了，会怎么样?
发送方： 阻塞send -> close
接收方： 阻塞receive, 如果close很快，会出问题吗?
不会的。接收方能读完所有消息，再接受到close

如何判断一次连接已经结束

2018年11月7日 18:01

在python上，接受消息的代码为

```
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((ip, port))
sock.recv(n)
```

关键在于其中的sock.recv(n), 这一行是否是阻塞的？如何判断本次连接已经关闭?

经过自己动手实验后，我发现：

1. 如果本次连接已经关闭，sock.recv(n)立马返回，且返回值为0
2. 如果本次连接还没有关闭，sock.recv(n)从缓冲区中尝试着读n个字节的数据。不足n个则直接返回读到的所有数据。如果当前没有数据可以读，则阻塞直到有数据。

这里和文件读写不一样。普通的文件读写，如果我们要求读1024字节，即调用read(fd, buffer, 1024), 如果返回的值小于1024我们就可以判断文件已经读完了。之所以是这样，是因为文件是一个确定大小的buffer，而TCP连接是一个字节流，没有开始和结束的概念，只有当连接被关闭了，你才读不到数据了。

不过在Linux上，socket相关的有一个是WAITALL flag, 这样，read不读取到要求的size是不会停止的，这样子处理一下，socket看起来就和普通的文件流完全一样了。但是这并不适用于服务器，因为一个命令可能小于1024字节，如果直接阻塞地读取1024个字节，那么就无法正确地处理命令了。

如果我们统一一下对文件的读写标准，即使最后一次读到了末尾，只读到了不到1024个字节，我们再尝试着读一下，这样返回的肯定是0字节。于是，我们可以把文件读完了和数据连接的数据传送完了统一为read(fd, buffer, 1024) == 0.

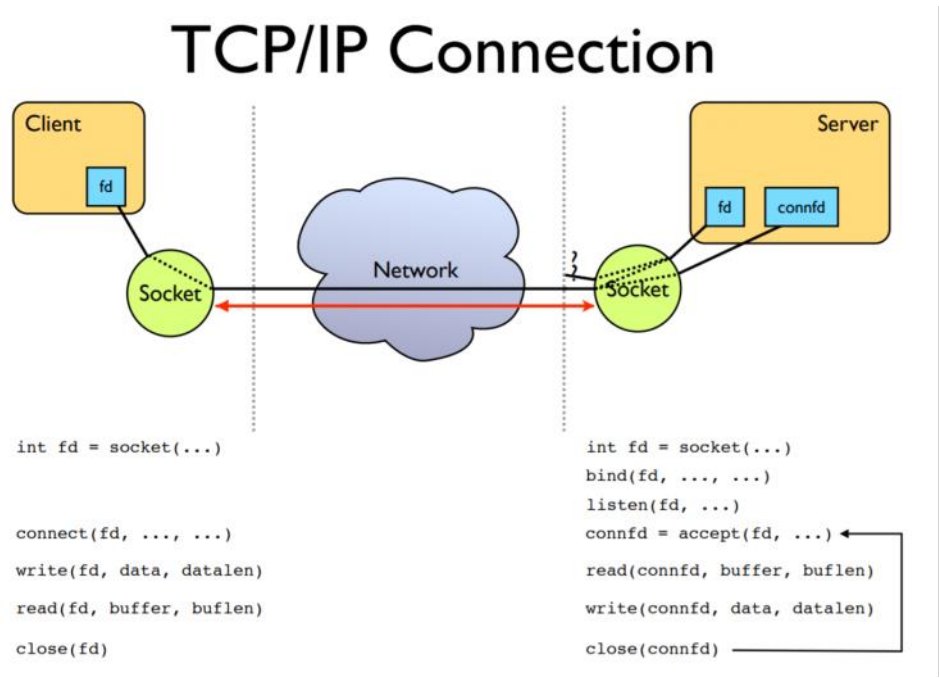
估计pipe也是跟socket一样的吧

收获

2018年11月7日 18:59

以后肯定不写C的socket了，但是python的socket还是可能写的，这里总结一下。
C那边，细节太多，就不看了。需要注意的就是，
一个栈上的东西是不能传指针给别人的，这个位置可能后来被别人用了。
pconnfd和thread_id,得传值，不能穿指针。这个教训。主线程传给子线程数据，可以传一个指针。其实，一个指针就很多了，new出来一个结构体，再把指针传过去，其实不就是可以传任何东西了吗？

https://csperkins.org/teaching/2007-2008/networked-systems/lecture04.pdf 这个lecture写的还是不错的



client这边，直接connect，其实他也会占用一个端口，但是我们不管。我们关心的是怎么连上server的某个端口。

server这边，先创建一个socket，再尝试注册一个端口（bind），然和才监听。这些都是非阻塞的，只有accept是可能阻塞的，如果没有接受到连接，就会阻塞。否则也是非阻塞的。
不过python里面accept返回的是conn, address, 返回的第一个东西才是一个可以用来读写的。

I’ m not going to talk about it here, except to warn you that you need to use flush

on sockets. These are buffered “files”, and a common mistake is to write something, and then read for a reply. Without a flush in there, you may wait forever for the reply, because the request may still be in your output buffer.
对一个socket又读又写的时候（用read/write)，需要注意flush

A protocol like HTTP uses a socket for only one transfer. The client sends a request, then reads a reply. That’s it. The socket is discarded. This means that a client can detect the end of the reply by receiving 0 bytes.

Strictly speaking, you’re supposed to use shutdown on a socket before you close it. The shutdown is an advisory to the socket at the other end. Depending on the argument you pass it, it can mean “I’m not going to send anymore, but I’ll still listen”, or “I’m not listening, good riddance!”. Most socket libraries, however, are so used to programmers neglecting to use this piece of etiquette that normally a close is the same as shutdown(); close(). So in most situations, an explicit shutdown is not needed.

In fact, on Windows I usually use threads (which work very, very well) with my sockets. Face it, if you want any kind of performance, your code will look very different on Windows than on Unix.

Finally, remember that even though blocking sockets are somewhat slower than non-blocking, in many cases they are the “right” solution. After all, if your app is driven by the data it receives over a socket, there’s not much sense in complicating the logic just so your app can wait on select instead of recv.