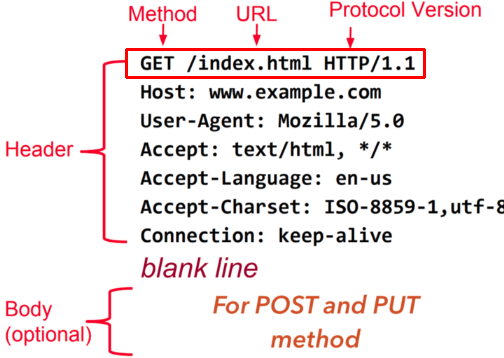
# realize http

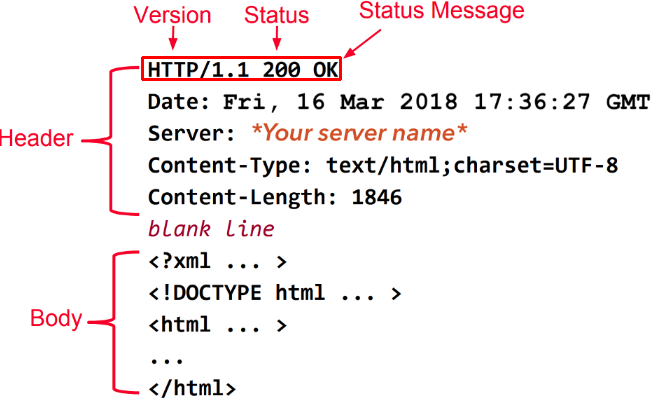
## 1.1 basic

### 1.1.1 req/res format

Send HTTP Request - Write lines to socket



HTTP Response - Read lines from socket



### 1.1.1 TCP send/recv max size

有个问题，我们在使用TCP发送数据的时候使用send, 接受数据的时候使用recv,那么两者的发送接受最大长度是多少呢？

send和recv都会返回发送和接受的长度.

先明确一个概念：每个TCP socket在内核中都有一个发送缓冲区和一个接收缓冲区，TCP的全双工的工作模式以及TCP的滑动窗口便是依赖于这两个独立的buffer以及此buffer的填充状态。接收缓冲区把数据缓存入内核，应用进程一 直没有调用read进行读取的话，此数据会一直缓存在相应 socket的接收缓冲区内。再啰嗦一点，不管进程是否读取socket，对端发来的数据都会经由内核接收并且缓存到socket的内核接收缓冲区之中。 read所做的工作，就是把内核缓冲区中的数据拷贝到应用层用户的buffer里面，仅此而已。进程调用send发送的数据的时候，最简单情况(也是一般情况)，将数据拷贝进入socket的内核发送缓冲区之中，然后send便会在上层返回。换句话说，send返回之时，数据不一定会发送到对端去(和 write写文件有点类似)，send仅仅是把应用层buffer的数据拷贝进socket的内核发送buffer中。后续我会专门用一篇文章介绍 read和send所关联的内核动作。每个UDP socket都有一个接收缓冲区，没有发送缓冲区，从概念上来说就是只要有数据就发，不管对方是否可以正确接收，所以不缓冲，不需要发送缓冲区。

接收缓冲区被TCP和UDP用来缓存网络上来的数据，一直保存到应用进程读走为止。对于TCP，如果应用进程一直没有读取，buffer满了之后，发生的动作是：通知对端TCP协议中的窗口关闭。这个便是滑动窗口的实现。保证TCP套接口接收缓冲区不会溢出，从而保证了TCP是可靠传输。因为对方不允许发出超过所通告窗口大小的数据。 这就是TCP的流量控制，如果对方无视窗口大小而发出了超过窗口大小的数据，则接收方TCP将丢弃它。 UDP：当套接口接收缓冲区满时，新来的数据报无法进入接收缓冲区，此数据报就被丢弃。UDP是没有流量控制的;快的发送者可以很容易地就淹没慢的接收者，导致接收方的UDP丢弃数据报。

查看测试机的socket发送缓冲区大小，cat /proc/sys/net/ipv4/tcp\_wmem

xiongyu@ubuntu:~$ cat /proc/sys/net/ipv4/tcp\_wmem  
4096 16384 4194304  
(4kb) (16kb) (4MB)

第一个值是一个限制值，socket发送缓存区的最少字节数；

第二个值是默认值；

第三个值是一个限制值，socket发送缓存区的最大字节数；

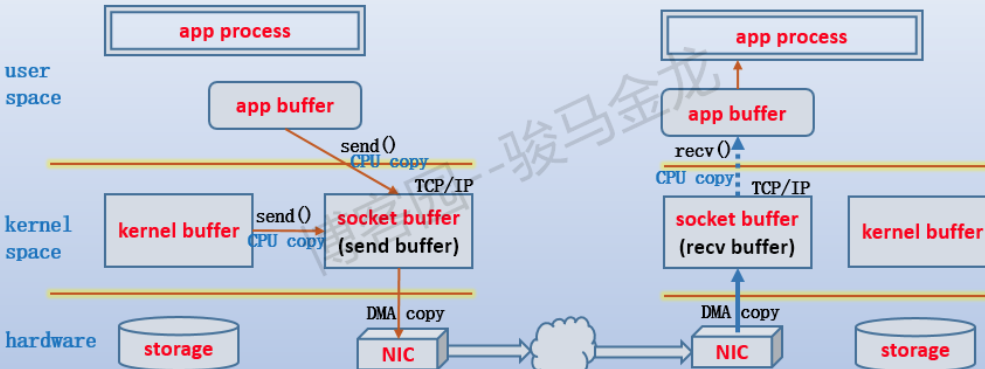
TCP协议栈维护着两个socket缓冲区：send buffer和recv buffer。

要通过TCP连接发送出去的数据都先拷贝到send buffer，可能是从用户空间进程的app buffer拷入的，也可能是从内核的kernel buffer拷入的，拷入的过程是通过send()函数完成的，由于也可以使用write()函数写入数据，所以也把这个过程称为写数据，相应的send buffer也就有了别称write buffer。不过send()函数比write()函数更有效率。

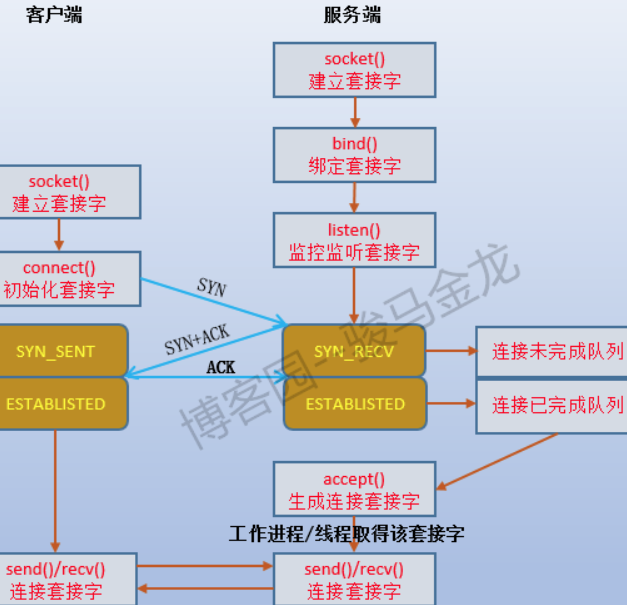
最终数据是通过网卡流出去的，所以send buffer中的数据需要拷贝到网卡中。由于一端是内存，一端是网卡设备，可以直接使用DMA的方式进行拷贝，无需CPU的参与。也就是说，send buffer中的数据通过DMA的方式拷贝到网卡中并通过网络传输给TCP连接的另一端：接收端。

当通过TCP连接接收数据时，数据肯定是先通过网卡流入的，然后同样通过DMA的方式拷贝到recv buffer中，再通过recv()函数将数据从recv buffer拷入到用户空间进程的app buffer中。

大致过程如下图：



1. 连接的具体过程分析



How to send messages with larger length than the buffer in socket programming?

I have a 200-byte-length char array for sending strings over the socket.

My problem is when sending messages which are larger than the char array, so I decided to send them in multiple chunks but I have no idea how to do it.

答：

Remember back in the section about send(), above, when I said that send() might not send all the bytes you asked it to? That is, you want it to send 512 bytes, but it returns 412. What happened to the remaining 100 bytes?

Well, they're still in your little buffer waiting to be sent out. Due to circumstances beyond your control, the kernel decided not to send all the data out in one chunk, and now, my friend, it's up to you to get the data out there.

You could write a function like this to do it, too:

int sendall(int s, char \*buf, int \*len)  
{  
 int total = 0; // how many bytes we've sent  
 int bytesleft = \*len; // how many we have left to send  
 int n;  
   
 while(total < \*len) {  
 n = send(s, buf+total, bytesleft, 0);  
 if (n == -1) { break; }  
 total += n;  
 bytesleft -= n;  
 }  
   
 \*len = total; // return number actually sent here  
 return n==-1?-1:0; // return -1 onm failure, 0 on success

}

For TCP, keep in mind that you are sending/receiving a stream of bytes, rather than a series of individual packets, and that so the sizes you pass to send() and recv() will not generally correspond to the sizes of the network packets anyway. That means that calling send() 6 times, each with 100 bytes of data, isn't much different from calling send() 1 time with 600 bytes of data, and so on. As others have noted, you do have to carefully check the return value of each send() and recv() call to see how many bytes were actually sent or received, since the number sent/received can (and will) sometimes be smaller than the number of bytes you requested. You need to know how many bytes were actually sent in order to know which bytes are appropriate to send on the next call, and you need to know how many bytes were actually received in order to know how many bytes are now valid in your receive-buffer.

https://stackoverflow.com/a/27623860/6329006

<https://www.cnblogs.com/f-ck-need-u/p/7623252.html>

## 1.1 simple realize 1

### 1.1.1 server 1

下面给出一个最简单的伪http服务(其实就是TCP服务，只不过给客户端返回的数据是HTTP格式的).

其中，返回的数据如下(将换行替换为了\r\n)：

RESPONSE = b"""\  
HTTP/1.1 200 OK  
Content-type: text/html  
Content-length: 15  
  
<h1>Hello!</h1>""".replace(b"\n", b"\r\n")

服务如下：其实就是TCP服务，然后传回上面的HTTP格式的字符串.

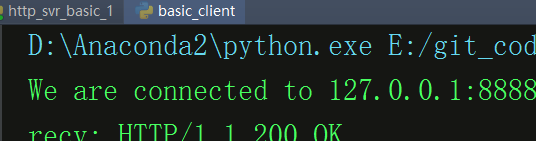
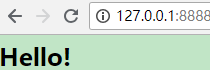
def test\_simple():  
 server\_sock = socket.socket()  
 server\_sock.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)  
 server\_sock.bind((HOST, PORT))  
 server\_sock.listen(0)  
 print "Listening on %s:%s..." % (HOST, PORT)  
  
 while 1:  
 client\_sock, client\_addr = server\_sock.accept()  
 print "New connection from %s." % client\_addr  
 client\_sock.sendall(RESPONSE)

上面有个问题尤其要注意：bind(host, port)时，一般服务器会有几个网卡，**如果我们绑定的是127.0.0.1， 那么在另一台机器上，connect这台机器的时候(使用电信ip:176:22:34:5)，会连接不上**。

测试代码如下：

def basic\_connect\_rcv():  
 s = socket.socket()  
 s.connect(('127.0.0.1', 8888))  
 print("We are connected to %s:%d" % s.getpeername())  
 print 'recv: %s' % str(s.recv(1024))

运行如下：

### 1.1.1 issue

Send HTTP Request - Write lines to socket

## 1.1 cited

### 1.1.1

Send HTTP Request - Write lines to socket

<https://defn.io/2018/02/25/web-app-from-scratch-01/>

<https://ruslanspivak.com/lsbaws-part1/>

https://docs.python.org/3.0/library/http.client.html

<https://github.com/embeddedmz/httpclient-cpp/blob/master/CMakeLists.txt>

<https://github.com/mrtazz/restclient-cpp>

<https://www.eurovps.com/blog/what-is-hypertext-transfer-protocol/#section06>

<https://www.eurovps.com/blog/300-errors/>

Python PEP-8编码风格指南中文版

<https://alvinzhu.xyz/2017/10/07/python-pep-8/>

python requests lib可以参考下。

使用python向服务器POST大文件

https://blog.geekli.cn/archives/129