要写网络程序就必须用Socket，这是程序员都知道的。而且，面试的时候，我们也会问对方会不会Socket编程？一般来说，很多人都会说，Socket编程基本就是listen，accept以及send，write等几个基本的操作。是的，就跟常见的文件操作一样，只要写过就一定知道。

对于网络编程，我们也言必称TCP/IP，似乎其它网络协议已经不存在了。对于TCP/IP，我们还知道TCP和UDP，前者可以保证数据的正确和可靠性，后者则允许数据丢失。最后，我们还知道，在建立连接前，必须知道对方的IP地址和端口号。除此，普通的程序员就不会知道太多了，很多时候这些知识已经够用了。最多，写服务程序的时候，会使用多线程来处理并发访问。

我们还知道如下几个事实：

1. 一个指定的端口号不能被多个程序共用。比如，如果IIS占用了80端口，那么Apache就不能也用80端口了
2. 很多防火墙只允许特定目标端口的数据包通过
3. 服务程序在listen某个端口并accept某个连接请求后，会生成一个新的socket来对该请求进行处理

于是，一个困惑了我很久的问题就产生了。如果一个socket创建后并与80端口绑定后，是否就意味着该socket占用了80端口呢？如果是这样的，那么当其accept一个请求后，生成的新的socket到底使用的是什么端口呢（我一直以为系统会默认给其分配一个空闲的端口号）？如果是一个空闲的端口，那一定不是80端口了，于是以后的TCP数据包的目标端口就不是80了--防火墙一定会组织其通过的！实际上，我们可以看到，防火墙并没有阻止这样的连接，而且这是最常见的连接请求和处理方式。我的不解就是，为什么防火墙没有阻止这样的连接？它是如何判定那条连接是因为connet80端口而生成的？是不是TCP数据包里有什么特别的标志？或者防火墙记住了什么东西？

后来，我又仔细研读了TCP/IP的协议栈的原理，对很多概念有了更深刻的认识。比如，在TCP和UDP同属于传输层，共同架设在IP层（网络层）之上。而IP层主要负责的是在节点之间（End to End）的数据包传送，这里的节点是一台网络设备，比如计算机。因为IP层只负责把数据送到节点，而不能区分上面的不同应用，所以TCP和UDP协议在其基础上加入了端口的信息，端口于是标识的是一个节点上的一个应用。除了增加端口信息，UPD协议基本就没有对IP层的数据进行任何的处理了。而TCP协议还加入了更加复杂的传输控制，比如滑动的数据发送窗口（Slice Window），以及接收确认和重发机制，以达到数据的可靠传送。不管应用层看到的是怎样一个稳定的TCP数据流，下面传送的都是一个个的IP数据包，需要由TCP协议来进行数据重组。

所以，我有理由怀疑，防火墙并没有足够的信息判断TCP数据包的更多信息，除了IP地址和端口号。而且，我们也看到，所谓的端口，是为了区分不同的应用的，以在不同的IP包来到的时候能够正确转发。

TCP/IP只是一个协议栈，就像操作系统的运行机制一样，必须要具体实现，同时还要提供对外的操作接口。就像操作系统会提供标准的编程接口，比如Win32编程接口一样，TCP/IP也必须对外提供编程接口，这就是Socket编程接口--原来是这么回事啊！

在Socket编程接口里，设计者提出了一个很重要的概念，那就是socket。这个socket跟文件句柄很相似，实际上在BSD系统里就是跟文件句柄一样存放在一样的进程句柄表里。这个socket其实是一个序号，表示其在句柄表中的位置。这一点，我们已经见过很多了，比如文件句柄，窗口句柄等等。这些句柄，其实是代表了系统中的某些特定的对象，用于在各种函数中作为参数传入，以对特定的对象进行操作--这其实是C语言的问题，在C++语言里，这个句柄其实就是this指针，实际就是对象指针啦。

现在我们知道，socket跟TCP/IP并没有必然的联系。Socket编程接口在设计的时候，就希望也能适应其他的网络协议。所以，socket的出现只是可以更方便的使用TCP/IP协议栈而已，其对TCP/IP进行了抽象，形成了几个最基本的函数接口。比如create，listen，accept，connect，read和write等等。

现在我们明白，如果一个程序创建了一个socket，并让其监听80端口，其实是向TCP/IP协议栈声明了其对80端口的占有。以后，所有目标是80端口的TCP数据包都会转发给该程序（这里的程序，因为使用的是Socket编程接口，所以首先由Socket层来处理）。所谓accept函数，其实抽象的是TCP的连接建立过程。accept函数返回的新socket其实指代的是本次创建的连接，而一个连接是包括两部分信息的，一个是源IP和源端口，另一个是宿IP和宿端口。所以，accept可以产生多个不同的socket，而这些socket里包含的宿IP和宿端口是不变的，变化的只是源IP和源端口。这样的话，这些socket宿端口就可以都是80，而Socket层还是能根据源/宿对来准确地分辨出IP包和socket的归属关系，从而完成对TCP/IP协议的操作封装！而同时，放火墙的对IP包的处理规则也是清晰明了，不存在前面设想的种种复杂的情形。

明白socket只是对TCP/IP协议栈操作的抽象，而不是简单的映射关系，这很重要！

TCP连接

手机能够使用联网功能是因为手机底层实现了TCP/IP协议，可以使手机终端通过无线网络建立TCP连接。TCP协议可以对上层网络提供接口，使上层网络数据的传输建立在“无差别”的网络之上。

建立起一个TCP连接需要经过“三次握手”：

- 第一次握手：客户端发送syn包(syn=j)到服务器，并进入SYN_SEND状态，等待服务器确认；
- 第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN_RECV状态；
- 第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=k+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。断开连接时服务器和客户端均可以主动发起断开TCP连接的请求，断开过程需要经过“四次握手”（过程就不细写了，就是服务器和客户端交互，最终确定断开）

SOCKET原理

> 套接字（socket）是通信的基石，是支持TCP/IP协议的网络通信的基本操作单元。它是网络通信过程中端点的抽象表示，包含进行网络通信必须的五种信息：连接使用的协议，本地主机的IP地址，本地进程的协议端口，远地主机的IP地址，远地进程的协议端口。

应用层通过传输层进行数据通信时，TCP会遇到同时为多个应用程序进程提供并发服务的问题。多个TCP连接或多个应用程序进程可能需要通过同一个 TCP协议端口传输数据。为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与TCP／IP协议交互提供了套接字(Socket)接口。应用层可以和传输层通过Socket接口，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。