1.socket 是什么？套接字是什么？

网络编程就是编写程序使两台联网的计算机相互交换数据。这就是全部内容了吗？是的！网络编程要比想象中的简单许多。

那么，这两台计算机之间用什么传输数据呢？首先需要物理连接。如今大部分计算机都已经连接到互联网，因此不用担心这一点。

在此基础上，只需要考虑如何编写数据传输程序。但实际上这点也不用愁，因为操作系统已经提供了 socket。即使对网络数据传输的原理不太熟悉，我们也能通过 socket 来编程。

什么是 socket？

socket 的原意是“插座”，在计算机通信领域，socket 被翻译为“套接字”，它是计算机之间进行通信的一种约定或一种方式。通过 socket 这种约定，一台计算机可以接收其他计算机的数据，也可以向其他计算机发送数据。

我们把插头插到插座上就能从电网获得电力供应，同样，为了与远程计算机进行数据传输，需要连接到因特网，而 socket 就是用来连接到因特网的工具。

socket 的典型应用就是 Web 服务器和浏览器：浏览器获取用户输入的 URL，向服务器发起请求，服务器分析接收到的 URL，将对应的网页内容返回给浏览器，浏览器再经过解析和渲染，就将文字、图片、视频等元素呈现给用户。

学习 socket，也就是学习计算机之间如何通信，并编写出实用的程序。

UNIX/Linux 中的 socket 是什么？

在 UNIX/Linux 系统中，为了统一对各种硬件的操作，简化接口，不同的硬件设备也都被看成一个文件。对这些文件的操作，等同于对磁盘上普通文件的操作。

你也许听很多高手说过，UNIX/Linux 中的一切都是文件！那个家伙说的没错。

为了表示和区分已经打开的文件，UNIX/Linux 会给每个文件分配一个 ID，这个 ID 就是一个整数，被称为文件描述符（File Descriptor）。例如：

- 通常用 0 来表示标准输入文件（stdin），它对应的硬件设备就是键盘；
- 通常用 1 来表示标准输出文件（stdout），它对应的硬件设备就是显示器。

UNIX/Linux 程序在执行任何形式的 I/O 操作时，都是在读取或者写入一个文件描述符。一个文件描述符只是一个和打开的文件相关联的整数，它的背后可能是一个硬盘上的普通文件、FIFO、管道、终端、键盘、显示器，甚至是一个网络连接。

请注意，网络连接也是一个文件，它也有文件描述符！你必须理解这句话。

我们可以通过 socket() 函数来创建一个网络连接，或者说打开一个网络文件，socket() 的返回值就是文件描述符。有了文件描述符，我们就可以使用普通的文件操作函数来传输数据了，例如：

- 用 read() 读取从远程计算机传来的数据；
- 用 write() 向远程计算机写入数据。

你看，只要用 socket() 创建了连接，剩下的就是文件操作了，网络编程原来就是如此简单！

Window 系统中的 socket 是什么？

Windows 也有类似“文件描述符”的概念，但通常被称为“文件句柄”。因此，本教程如果涉及 Windows 平台将使用“句柄”，如果涉及 Linux 平台则使用“描述符”。

与 UNIX/Linux 不同的是，Windows 会区分 socket 和文件，Windows 就把 socket 当做一个网络连接来对待，因此需要调用专门针对 socket 而设计的数据传输函数，针对普通文件的输入输出函数就无效了。

2. 套接字有哪些类型？socket 有哪些类型？

这个世界上有很多种套接字（socket），比如 DARPA Internet 地址（Internet 套接字）、本地节点的路径名（Unix 套接字）、CCITT X.25 地址（X.25 套接字）等。但本教程只讲第一种套接字——Internet 套接字，它是最具代表性的，也是最经典最常用的。以后我们提及套接字，指的都是 Internet 套接字。

根据数据的传输方式，可以将 Internet 套接字分成两种类型。通过 socket() 函数创建连接时，必须告诉它使用哪种数据传输方式。

Internet 套接字其实还有很多其它数据传输方式，但是我可不想吓到你，本教程只讲常用的两种。

流格式套接字（SOCK_STREAM）

流格式套接字（Stream Sockets）也叫“面向连接的套接字”，在代码中使用 SOCK_STREAM 表示。

SOCK_STREAM 是一种可靠的、双向的通信数据流，数据可以准确无误地到达另一台计算机，如果损坏或丢失，可以重新发送。

流格式套接字有自己的纠错机制，在此我们就不讨论了。

SOCK_STREAM 有以下几个特征：

- 数据在传输过程中不会消失；
- 数据是按照顺序传输的；
- 数据的发送和接收不是同步的（有的教程也称“不存在数据边界”）。

可以将 SOCK_STREAM 比喻成一条传送带，只要传送带本身没有问题（不会断网），就能保证数据不丢失；同时，较晚传送的数据不会先到达，较早传送的数据不会晚到达，这就保证了数据是按照顺序传递的。

为什么流格式套接字可以达到高质量的数据传输呢？这是因为它使用了 TCP 协议（The Transmission Control Protocol，传输控制协议），TCP 协议会控制你的数据按照顺序到达并且没有错误。

你也许见过 TCP，是因为你经常听说“TCP/IP”。TCP 用来确保数据的正确性，IP（Internet Protocol，网络协议）用来控制数据如何从源头到达目的地，也就是常说的“路由”。

那么，“数据的发送和接收不同步”该如何理解呢？

假设传送带传送的是水果，接收者需要凑齐 100 个后才能装袋，但是传送带可能把这 100 个水果分批传送，比如第一批传送 20 个，第二批传送 50 个，第三批传送 30 个。接收者不需要和传送带保持同步，只要根据自己的节奏来装袋即可，不用管传送带传送了几批，也不用每到一批就装袋一次，可以等到凑够了 100 个水果再装袋。

流格式套接字的内部有一个缓冲区（也就是字符数组），通过 socket 传输的数据将保存到这个缓冲区。接收端在收到数据后并不一定立即读取，只要数据不超过缓冲区的容量，接收端有可能在缓冲区被填满以后一次性地读取，也可能分成好几次读取。

也就是说，不管数据分几次传送过来，接收端只需要根据自己的要求读取，不用非得在数据到达时立即读取。传送端有自己的节奏，接收端也有自己的节奏，它们是不一致的。

流格式套接字有什么实际的应用场景吗？浏览器所使用的 http 协议就基于面向连接的套接字，因为必须要确保数据准确无误，否则加载的 HTML 将无法解析。

数据报格式套接字（SOCK_DGRAM）

数据报格式套接字（Datagram Sockets）也叫“无连接的套接字”，在代码中使用 SOCK_DGRAM 表示。

计算机只管传输数据，不作数据校验，如果数据在传输中损坏，或者没有到达另一台计算机，是没有办法补救的。也就是说，数据错了就错了，无法重传。

因为数据报套接字所做的校验工作少，所以在传输效率方面比流格式套接字要高。

可以将 SOCK_DGRAM 比喻成高速移动的摩托车快递，它有以下特征：

- 强调快速传输而非传输顺序；
- 传输的数据可能丢失也可能损毁；
- 限制每次传输的数据大小；
- 数据的发送和接收是同步的（有的教程也称“存在数据边界”）。

众所周知，速度是快递行业的生命。用摩托车发往同一地点的两件包裹无需保证顺序，只要以最快的速度交给客户就行。这种方式存在损坏或丢失的风险，而且包裹大小有一定限制。因此，想要传递大量包裹，就得分配发送。

另外，用两辆摩托车分别发送两件包裹，那么接收者也需要分两次接收，所以“数据的发送和接收是同步的”；换句话说，接收次数应该和发送次数相同。

总之，数据报套接字是一种不可靠的、不按顺序传递的、以追求速度为目的的套接字。

数据报套接字也使用 IP 协议作路由，但是它不使用 TCP 协议，而是使用 UDP 协议（User Datagram Protocol，用户数据报协议）。

QQ 视频聊天和语音聊天就使用 SOCK_DGRAM 来传输数据，因为首先要保证通信的效率，尽量减小延迟，而数据的正确性是次要的，即使丢失很小的一部分数据，视频和音频也可以正常解析，最多出现噪点或杂音，不会对通信质量有实质的影响。

注意：SOCK_DGRAM 没有想象中的糟糕，不会频繁的丢失数据，数据错误只是小概率事件。

3. 面向连接和无连接的套接字到底有什么区别

上一节《套接字有哪些类型》提到，流格式套接字（Stream Sockets）就是“面向连接的套接字”，它基于 TCP 协议；数据报格式套接字（Datagram Sockets）就是“无连接的套接字”，它基于 UDP 协议。

这给大家造成一种印象，面向连接就是可靠的通信，无连接就是不可靠的通信，实际情况是这样吗？

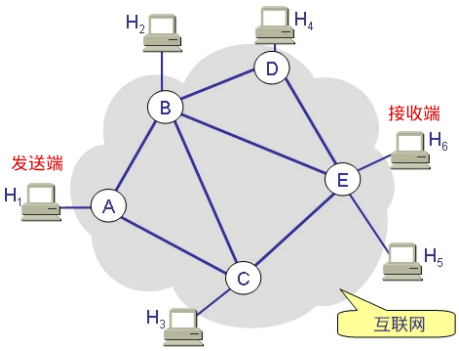
另外，不管是哪种数据传输方式，都得通过整个 Internet 网络的物理线路将数据传输过去，从这个层面理解，所有的 socket 都是有物理连接的呀，为什么还有无连接的 socket 呢？

本节就来给大家解开种种谜团，加深大家对数据传输方式的认识。

从字面上理解，面向连接好像有一条管道，它连接发送端和接收端，数据包都通过这条管道来传输。当然，两台计算机在通信之前必须先搭建好管道。

无连接好像没头苍蝇乱撞，数据包从发送端到接收端并没有固定的线路，爱怎么走就怎么走，只要能到达就行。每个数据包都比较自私，不和别人分享自己的线路，但是，大家最终都能殊途同归，到达接收端。

这样理解没错，但是我相信这还不够深入，大家还是感觉云里雾里，没有看到本质。好，接下来就是见证奇迹的时刻，我会用实例给大家演示！



上图是一个简化的互联网模型，H1 ~ H6 表示计算机，A~E 表示路由器，发送端发送的数据必须经过路由器的转发才能到达接收端。

假设 H1 要发送若干个数据包给 H6，那么有多条路径可以选择，比如：

- 路径①：H1 --> A --> C --> E --> H6
- 路径②：H1 --> A --> B --> E --> H6
- 路径③：H1 --> A --> B --> D --> E --> H6
- 路径④：H1 --> A --> B --> C --> E --> H6
- 路径⑤：H1 --> A --> C --> B --> D --> E --> H6

数据包的传输路径是路由器根据算法来计算出来的，算法会考虑很多因素，比如网络的拥堵状况、下一个路由器是否忙碌等。

无连接的套接字

对于无连接的套接字，每个数据包可以选择不同的路径，比如第一个数据包选择路径④，第二个数据包选择路径①，第三个数据包选择路径②……当然，它们也可以选择相同的路径，那也只不过是巧合而已。

每个数据包之间都是独立的，各走各的路，谁也不影响谁，除了迷路的或者发生意外的数据包，最后都能到达 H6。但是，到达的顺序是不确定的，比如：

- 第一个数据包选择了一条比较长的路径（比如路径⑤），第三个数据包选择了一条比较短的路径（比如路径①），虽然第一个数据包很早就出发了，但是走的路比较远，最终还是晚于第三个数据包达到。
- 第一个数据包选择了一条比较短的路径（比如路径①），第三个数据包选择了一条比较长的路径（比如路径⑤），按理说第一个数据包应该先到达，但是非常不幸，第一个数据包走的路比较拥堵，这条路上的数据量非常大，路由器处理得很慢，所以它还是晚于第三个数据包达到了。

还有一些意外情况会发生，比如：

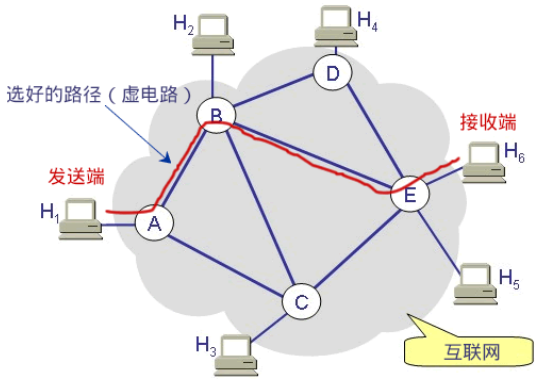
- 第一个数据包选择了路径①，但是路由器 C 突然断电了，那它就到不了 H6 了。
- 第三个数据包选择了路径②，虽然路不远，但是太拥堵，以至于它等待的时间太长，路由器把它丢弃了。

总之，对于无连接的套接字，数据包在传输过程中会发生各种不测，也会发生各种奇迹。H1 只负责把数据包发出，至于它什么时候到达，先到达还是后到达，有没有成功到达，H1 都不管了；H6 也没有选择的权利，只能被动接收，收到什么算什么，爱用不用。

无连接套接字遵循的是「尽最大努力交付」的原则，就是尽力而为，实在做不到了也没办法。无连接套接字提供的没有质量保证的服务。

面向连接的套接字

面向连接的套接字在正式通信之前要先确定一条路径，没有特殊情况的话，以后就固定地使用这条路径来传递数据包了。当然，路径被破坏的话，比如某个路由器断电了，那么会重新建立路径。



这条路径是由路由器维护的，路径上的所有路由器都要存储该路径的信息（实际上只需要存储上游和下游的两个路由器的位置就行），所以路由器是有开销的。

H1 和 H6 通信完毕后，要断开连接，销毁路径，这个时候路由器也会把之前存储的路径信息擦除。

在很多网络通信教程中，这条预先建立好的路径被称为“虚电路”，就是一条虚拟的通信电路。

为了保证数据包准确、顺序地到达，发送端在发送数据包以后，必须得到接收端的确认才发送下一个数据包；如果数据包发出去了，一段时间以后仍然没有得到接收端的回应，那么发送端会重新再发送一次，直到得到接收端的回应。这样一来，发送端发送的所有数据包都能到达接收端，并且是按照顺序到达的。

发送端发送一个数据包，如何得到接收端的确认呢？很简单，为每一个数据包分配一个 ID，接收端接收到数据包以后，再给发送端返回一个数据包，告诉发送端我接收到了 ID 为 xxx 的数据包。

面向连接的套接字会比无连接的套接字多出很多数据包，因为发送端每发送一个数据包，接收端就会返回一个数据包。此外，建立连接和断开连接的过程也会传递很多数据包。

不但是数量多了，每个数据包也变大了：除了源端口和目的端口，面向连接的套接字还包括序号、确认信号、数据偏移、控制标志（通

常说的 URG、ACK、PSH、RST、SYN、FIN）、窗口、校验和、紧急指针、选项等信息；而无连接的套接字则只包含长度和校验和信息。

有连接的数据包比无连接大很多，这意味着更大的负载和更大的带宽。许多即时聊天软件采用 UDP 协议（无连接套接字），与此有莫大的关系。

总结

两种套接字各有优缺点：

- 无连接套接字传输效率高，但是不可靠，有丢失数据包、捣乱数据的风险；
- 有连接套接字非常可靠，万无一失，但是传输效率低，耗费资源多。

两种套接字的特点决定了它们的应用场景，有些服务对可靠性要求比较高，必须数据包能够完整无误地送达，那就得选择有连接的套接字（TCP 服务），比如 HTTP、FTP 等；而另一些服务，并不需要那么高的可靠性，效率和实时才是它们所关心的，那就可以选择无连接的套接字（UDP 服务），比如 DNS、即时聊天工具等。

4. OSI 网络七层模型简明教程

如果你读过计算机专业，或者学习过网络通信，那你一定听说过 OSI 模型，它曾无数次让你头大。OSI 是 Open System Interconnection 的缩写，译为“开放式系统互联”。

OSI 模型把网络通信的工作分为 7 层，从下到上分别是物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。

OSI 只是存在于概念和理论上的一种模型，它的缺点是分层太多，增加了网络工作的复杂性，所以没有大规模应用。后来人们对 OSI 进行了简化，合并了一些层，最终只保留了 4 层，从下到上分别是接口层、网络层、传输层和应用层，这就是大名鼎鼎的 TCP/IP 模型。

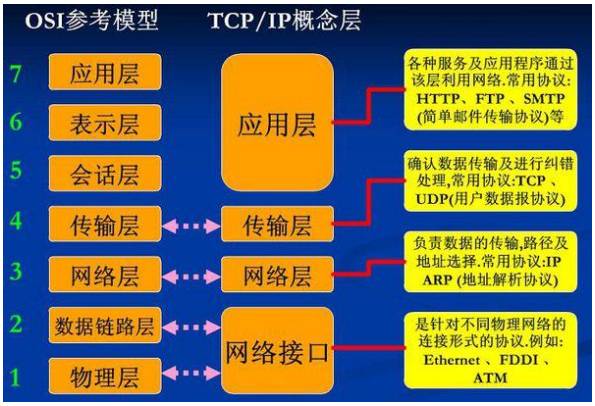


图 1：OSI 七层网络模型和 TCP/IP 四层网络模型的对比

这个网络模型究竟是干什么呢？简而言之就是进行数据封装的。

我们平常使用的程序（或者说软件）一般都是通过应用层来访问网络的，程序产生的数据会一层一层地往下传输，直到最后的网络接口层，就通过网线发送到互联网上去了。数据每往下走一层，就会被这一层的协议增加一层包装，等到发送到互联网上时，已经比原始数据多了四层包装。整个数据封装的过程就像俄罗斯套娃。

当另一台计算机接收到数据包时，会从网络接口层再一层一层往上传输，每传输一层就拆开一层包装，直到最后的应用层，就得到了最原始的数据，这才是程序要使用的数据。

给数据加包装的过程，实际上就是在数据的头部增加一个标志（一个数据块），表示数据经过了这一层，我已经处理过了。给数据拆包装的过程正好相反，就是去掉数据头部的标志，让它逐渐现出原形。

你看，在互联网上传输一份数据是多么地复杂啊，而我们却感受不到，这就是网络模型的厉害之处。我们只需要在代码中调用一个函数，就能让下面的所有网络层为我们工作。

我们所说的 socket 编程，是站在传输层的基础上，所以可以使用 TCP/UDP 协议，但是不能干「访问网页」这样的事情，因为访问网页所需要的 http 协议位于应用层。

两台计算机进行通信时，必须遵守以下原则：

- 必须是同一层次进行通信，比如，A 计算机的应用层和 B 计算机的传输层就不能通信，因为它们不在一个层次，数据的拆包会遇到问题。
- 每一层的功能都必须相同，也就是拥有完全相同的网络模型。如果网络模型都不同，那不就乱套了，谁都不认识谁。
- 数据只能逐层传输，不能跃层。
- 每一层可以使用下层提供的服务，并向上层提供服务。

5. TCP/IP 协议族

上节《OSI 网络七层模型简明教程》中讲到，目前实际使用的网络模型是 TCP/IP 模型，它对 OSI 模型进行了简化，只包含了四层，从上到下分别是应用层、传输层、网络层和链路层（网络接口层），每一层都包含了若干协议。

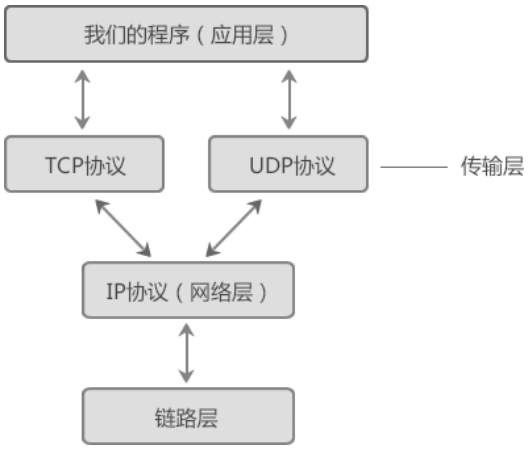
协议（Protocol）就是网络通信过程中的约定或者合同，通信的双方必须都遵守才能正常收发数据。协议有很多种，例如 TCP、UDP、IP 等，通信的双方必须使用同一协议才能通信。协议是一种规范，由计算机组织制定，规定了很多细节，例如，如何建立连接，如何相互识别等。

协议仅仅是一种规范，必须由计算机软件来实现。例如 IP 协议规定了如何找到目标计算机，那么各个开发商在开发自己的软件时就必须遵守该协议，不能另起炉灶。

TCP/IP 模型包含了 TCP、IP、UDP、Telnet、FTP、SMTP 等上百个互为关联的协议，其中 TCP 和 IP 是最常用的两种底层协议，所以把它们统称为“TCP/IP 协议族”。

也就是说，“TCP/IP 模型”中所涉及到的协议称为“TCP/IP 协议族”，你可以区分这两个概念，也可以认为它们是等价的，随便你怎么想。

本教程所讲的 socket 编程是基于 TCP 和 UDP 协议的，它们的层级关系如下图所示：



【扩展阅读】开放式系统（Open System）

把协议分成多个层次有哪些优点？协议设计更容易？当然这也足以成为优点之一。但是还有更重要的原因，就是为了通过标准化操作设计成开放式系统。

标准本身就是对外公开的，会引导更多的人遵守规范。以多个标准为依据设计的系统称为开放式系统（Open System），我们现在学习的 TCP/IP 协议族也属于其中之一。

接下来了解一下开放式系统具有哪些优点。

路由器用来完成 IP 层的交互任务。某个网络原来使用 A 公司的路由器，现要将其替换成 B 公司的，是否可行？这并非难事，并不一定要换成同一公司的同一型号路由器，因为所有生产商都会按照 IP 层标准制造。

再举个例子。大家的计算机是否装有网络接口卡，也就是所谓的网卡？尚未安装也无妨，其实很容易买到，因为所有网卡制造商都会遵守链路层的协议标准。这就是开放式系统的优点。

标准的存在意味着高速的技术发展，这也是开放式系统设计最大的原因所在。实际上，软件工程中的“面向对象（Object Oriented）”的诞生背景中也有标准化的影子。也就是说，标准对于技术发展起着举足轻重的作用。

6. IP、MAC 和端口号——网络通信中确认身份信息的三要素

在茫茫的互联网海洋中，要找到一台计算机非常不容易，有三个要素必须具备，它们分别是 IP 地址、MAC 地址和端口号。

IP 地址

IP 地址是 Internet Protocol Address 的缩写，译为“网际协议地址”。

目前大部分软件使用 IPv4 地址，但 IPv6 也正在被人们接受，尤其是在教育网中，已经大量使用。

一台计算机可以拥有一个独立的 IP 地址，一个局域网也可以拥有一个独立的 IP 地址（对外就好像只有一台计算机）。对于目前广泛使用 IPv4 地址，它的资源是非常有限的，一台计算机一个 IP 地址是不现实的，往往是一个局域网才拥有一个 IP 地址。

在因特网上进行通信时，必须要知道对方的 IP 地址。实际上数据包中已经附带了 IP 地址，把数据包发送给路由器以后，路由器会根据 IP 地址找到对方的地里位置，完成一次数据的传递。路由器有非常高效和智能的算法，很快就会找到目标计算机。

MAC 地址

现实的情况是，一个局域网往往才能拥有一个独立的 IP；换句话说，IP 地址只能定位到一个局域网，无法定位到具体的一台计算机。这可怎么办呀？这样也没法通信啊。

其实，真正能唯一标识一台计算机的是 MAC 地址，每个网卡的 MAC 地址在全世界都是独一无二的。计算机出厂时，MAC 地址已经被写死到网卡里面了（当然通过某些“奇巧淫技”也是可以修改的）。局域网中的路由器/交换机会记录每台计算机的 MAC 地址。

MAC 地址是 Media Access Control Address 的缩写，直译为“媒体访问控制地址”，也称为局域网地址（LAN Address），以太网地址（Ethernet Address）或物理地址（Physical Address）。

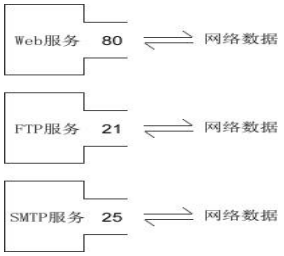
数据包中除了会附带对方的 IP 地址，还会附带对方的 MAC 地址，当数据包达到局域网以后，路由器/交换机会根据数据包中的 MAC 地址找到对应的计算机，然后把数据包转交给它，这样就完成了数据的传递。

端口号

有了 IP 地址和 MAC 地址，虽然可以找到目标计算机，但仍然不能进行通信。一台计算机可以同时提供多种网络服务，例如 Web 服务（网站）、FTP 服务（文件传输服务）、SMTP 服务（邮箱服务）等，仅有 IP 地址和 MAC 地址，计算机虽然可以正确接收到数据包，但是却不知道要将数据包交给哪个网络程序来处理，所以通信失败。

为了区分不同的网络程序，计算机会为每个网络程序分配一个独一无二的端口号（Port Number），例如，Web 服务的端口号是 80，FTP 服务的端口号是 21，SMTP 服务的端口号是 25。

端口（Port）是一个虚拟的、逻辑上的概念。可以将端口理解为一道门，数据通过这道门流入流出，每道门有不同的编号，就是端口号。如下图所示：



7. Linux 下的 socket 演示程序

和 C 语言教程一样，我们从一个简单的“Hello World!”程序切入 socket 编程。

本节演示了 Linux 下的代码，server.cpp 是服务器端代码，client.cpp 是客户端代码，要实现的功能是：客户端从服务器读取一个字符串并打印出来。

服务器端代码 server.cpp：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(){
    //创建套接字
    int serv_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    //将套接字和 IP、端口绑定
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));  //每个字节都用 0 填充
    serv_addr.sin_family = AF_INET;  //使用 IPv4 地址
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的 IP 地址
    serv_addr.sin_port = htons(1234);  //端口
    bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

    //进入监听状态，等待用户发起请求
    listen(serv_sock, 20);

    //接收客户端请求
    struct sockaddr_in clnt_addr;
    socklen_t clnt_addr_size = sizeof(clnt_addr);
    int clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);

    //向客户端发送数据
    char str[] = "http://c.biancheng.net/socket/";
```

```
    write(clnt_sock, str, sizeof(str));

    //关闭套接字
    close(clnt_sock);
    close(serv_sock);

    return 0;
}
```

客户端代码 client.cpp：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

int main(){
    //创建套接字
    int sock = socket(AF_INET, SOCK_STREAM, 0);

    //向服务器（特定的 IP 和端口）发起请求
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));  //每个字节都用 0 填充
    serv_addr.sin_family = AF_INET;  //使用 IPv4 地址
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的 IP 地址
    serv_addr.sin_port = htons(1234);  //端口
    connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

    //读取服务器传回的数据
    char buffer[40];
    read(sock, buffer, sizeof(buffer)-1);

    printf("Message form server: %s\n", buffer);

```

```
26.     //关闭套接字
27.     close(sock);
28.
29.     return 0;
30. }
```

启动一个终端（Shell），先编译 server.cpp 并运行：

```
[admin@localhost ~]$ g++ server.cpp -o server
[admin@localhost ~]$ ./server
#等待请求的到来
```

正常情况下，程序运行到 accept() 函数就会被阻塞，等待客户端发起请求。

接下再启动一个终端，编译 client.cpp 并运行：

```
[admin@localhost ~]$ g++ client.cpp -o client
[admin@localhost ~]$ ./client
Message form server: http://c.biancheng.net/socket/
```

client 接收到从 server 发送过来的字符串就运行结束了，同时，server 完成发送字符串的任务也运行结束了。大家可以通过两个打开的终端来观察。

client 运行后，通过 connect() 函数向 server 发起请求，处于监听状态的 server 被激活，执行 accept() 函数，接受客户端的请求，然后执行 write() 函数向 client 传回数据。client 接收到传回的数据后，connect() 就运行结束了，然后使用 read() 将数据读取出来。

server 只接受一次 client 请求，当 server 向 client 传回数据后，程序就运行结束了。如果想再次接收到服务器的数据，必须再次运行 server，所以这是一个非常简陋的 socket 程序，不能够一直接受客户端的请求。

源码解析

先说一下 server.cpp 中的代码。

1）第 11 行通过 socket() 函数创建了一个套接字，参数 AF_INET 表示使用 IPv4 地址，SOCK_STREAM 表示使用面向连接的套接字，IPPROTO_TCP 表示使用 TCP 协议。在 Linux 中，socket 也是一种文件，有文件描述符，可以使用 write() / read() 函数进行 I/O 操作，这一点已在《socket 是什么》中进行了讲解。

第 19 行通过 bind() 函数将套接字 serv_sock 与特定的 IP 地址和端口绑定，IP 地址和端口都保存在 sockaddr_in 结构体中。

socket() 函数确定了套接字的各种属性，bind() 函数让套接字与特定的 IP 地址和端口对应起来，这样客户端才能连接到该套接字。

第 22 行让套接字处于被动监听状态。所谓被动监听，是指套接字一直处于“睡眠”中，直到客户端发起请求才会被“唤醒”。

第 27 行的 accept() 函数用来接收客户端的请求。程序一旦执行到 accept() 就会被阻塞（暂停运行），直到客户端发起请求。

第 31 行的 write() 函数用来向套接字文件中写入数据，也就是向客户端发送数据。

和普通文件一样，socket 在使用完毕后也要用 close() 关闭。

2) 再说一下 client.cpp 中的代码。client.cpp 中的代码和 server.cpp 中有一些区别。

第 19 行代码通过 connect() 向服务器发起请求，服务器的 IP 地址和端口号保存在 sockaddr_in 结构体中。直到服务器传回数据后，connect() 才运行结束。
第 23 行代码通过 read() 从套接字文件中读取数据。

8. Windows 下的 socket 演示程序

上节演示了 Linux 下的 socket 程序，这节来看一下 Windows 下的 socket 程序。同样，server.cpp 为服务器端代码，client 为客户端代码。

服务器端代码 server.cpp：

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加载 ws2_32.dll

int main(){
    //初始化 DLL
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    //创建套接字
    SOCKET servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

    //绑定套接字
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用 0 填充
    sockAddr.sin_family = PF_INET;  //使用 IPv4 地址
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的 IP 地址
    sockAddr.sin_port = htons(1234);  //端口
    bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));

    //进入监听状态
    listen(servSock, 20);

    //接收客户端请求
    SOCKADDR clntAddr;
    int nSize = sizeof(SOCKADDR);
    SOCKET clntSock = accept(servSock, (SOCKADDR*)&clntAddr, &nSize);

    //向客户端发送数据
    char *str = "Hello World!";
    send(clntSock, str, strlen(str)+sizeof(char), NULL);
```

```
32.
33.     //关闭套接字
34.     closesocket(clntSock);
35.     closesocket(servSock);
36.
37.     //终止 DLL 的使用
38.     WSACleanup();
39.
40.     return 0;
41. }
```

客户端代码 client.cpp：

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <WinSock2.h>
4.  #pragma comment(lib, "ws2_32.lib")  //加载 ws2_32.dll
5.
6.  int main(){
7.      //初始化 DLL
8.      WSADATA wsaData;
9.      WSAStartup(MAKEWORD(2, 2), &wsaData);
10.
11.     //创建套接字
12.     SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
13.
14.     //向服务器发起请求
15.     sockaddr_in sockAddr;
16.     memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用 0 填充
17.     sockAddr.sin_family = PF_INET;
18.     sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
19.     sockAddr.sin_port = htons(1234);
20.     connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
21.
22.     //接收服务器传回的数据
23.     char szBuffer[MAXBYTE] = {0};
```

```
    recv(sock, szBuffer, MAXBYTE, NULL);

    //输出接收到的数据
    printf("Message form server: %s\n", szBuffer);

    //关闭套接字
    closesocket(sock);

    //终止使用 DLL
    WSACleanup();

    system("pause");
    return 0;
}
```

将 server.cpp 和 client.cpp 分别编译为 server.exe 和 client.exe，先运行 server.exe，再运行 client.exe，输出结果为：
Message form server: Hello World!

Windows 下的 socket 程序和 Linux 思路相同，但细节有所差别：
1) Windows 下的 socket 程序依赖 Winsock.dll 或 ws2_32.dll，必须提前加载。DLL 有两种加载方式，请查看：动态链接库 DLL 的加载

2) Linux 使用“文件描述符”的概念，而 Windows 使用“文件句柄”的概念；Linux 不区分 socket 文件和普通文件，而 Windows 区分；Linux 下 socket() 函数的返回值为 int 类型，而 Windows 下为 SOCKET 类型，也就是句柄。

3) Linux 下使用 read() / write() 函数读写，而 Windows 下使用 recv() / send() 函数发送和接收。

4) 关闭 socket 时，Linux 使用 close() 函数，而 Windows 使用 closesocket() 函数。

9. Windows 下使用 WSAStartup()函数加载 DLL

WinSock（Windows Socket）编程依赖于系统提供的动态链接库(DLL)，有两个版本：

- 较早的 DLL 是 wsock32.dll，大小为 28KB，对应的头文件为 winsock1.h；
- 最新的 DLL 是 ws2_32.dll，大小为 69KB，对应的头文件为 winsock2.h。

几乎所有的 Windows 操作系统都已经支持 ws2_32.dll，包括个人操作系统 Windows 95 OSR2、Windows 98、Windows Me、Windows 2000、XP、Vista、Win7、Win8、Win10 以及服务器操作系统 Windows NT 4.0 SP4、Windows Server 2003、Windows Server 2008 等，所以你可以毫不犹豫地使用最新的 ws2_32.dll。

使用 DLL 之前必须把 DLL 加载到当前程序，你可以在编译时加载，也可以在程序运行时加载，我们已在《动态链接库 DLL 的加载：隐式加载(载入时加载)和显式加载(运行时加载)》进行了讲解。

这里使用`#pragma`命令，在编译时加载：

```
#pragma comment (lib, "ws2_32.lib")
```

WSAStartup() 函数

使用 DLL 之前，还需要调用 WSAStartup() 函数进行初始化，以指明 WinSock 规范的版本，它的原型为：

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);
```

wVersionRequested 为 WinSock 规范的版本号，低字节为主版本号，高字节为副版本号（修正版本号）；lpWSAData 为指向 WSAData 结构体的指针。

关于 WinSock 规范

WinSock 规范的最新版本号为 2.2，较早的有 2.1、2.0、1.1、1.0，ws2_32.dll 支持所有的规范，而 wsock32.dll 仅支持 1.0 和 1.1。

wsock32.dll 已经能够很好的支持 TCP/IP 通信程序的开发，ws2_32.dll 主要增加了对其他协议的支持，不过建议使用最新的 2.2 版本。

wVersionRequested 参数用来指明我们希望使用的版本号，它的类型为 WORD，等价于 unsigned short，是一个整数，所以需要用 MAKEWORD() 宏函数对版本号进行转换。例如：

```
MAKEWORD(1, 2);  //主版本号为 1，副版本号为 2，返回 0x0201
MAKEWORD(2, 2);  //主版本号为 2，副版本号为 2，返回 0x0202
```

关于 WSAData 结构体

WSAStartup() 函数执行成功后，会将与 ws2_32.dll 有关的信息写入 WSAData 结构体变量。WSAData 的定义如下：

```
typedef struct WSAData {
    WORD           wVersion;  //ws2_32.dll 建议我们使用的版本号
    WORD           wHighVersion;  //ws2_32.dll 支持的最高版本号
    //一个以 null 结尾的字符串，用来说明 ws2_32.dll 的实现以及厂商信息
    char           szDescription[WSADESCRIPTION_LEN+1];
```

```
    //一个以 null 结尾的字符串，用来说明 ws2_32.dll 的状态以及配置信息
    char            szSystemStatus[WSASYS_STATUS_LEN+1];
    unsigned short iMaxSockets;  //2.0 以后不再使用
    unsigned short iMaxUdpDg;  //2.0 以后不再使用
    char FAR       *lpVendorInfo;  //2.0 以后不再使用
} WSADATA, *LPWSADATA;
```

最后 3 个成员已弃之不用，szDescription 和 szSystemStatus 包含的信息基本没有实用价值，读者只需关注前两个成员即可。请看下面的代码：

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")

int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    printf("wVersion: %d.%d\n", LOBYTE(wsaData.wVersion), HIBYTE(wsaData.wVersion));
    printf("wHighVersion: %d.%d\n", LOBYTE(wsaData.wHighVersion), HIBYTE(wsaData.wHighVersion));
    printf("szDescription: %s\n", wsaData.szDescription);
    printf("szSystemStatus: %s\n", wsaData.szSystemStatus);

    return 0;
}
```

运行结果：
wVersion: 2.2
wHighVersion: 2.2
szDescription: WinSock 2.0
szSystemStatus: Running

ws2_32.dll 支持的最高版本为 2.2，建议使用的版本也是 2.2。

综上所述：WinSock 编程的第一步就是加载 ws2_32.dll，然后调用 WSAStartup() 函数进行初始化，并指明要使用的版本号。

10. socket()函数用法详解：创建套接字

不管是 Windows 还是 Linux，都使用 socket() 函数来创建套接字。socket() 在两个平台下的参数是相同的，不同的是返回值。

在《socket 是什么》一节中我们讲到了 Windows 和 Linux 在对待 socket 方面的区别。

Linux 中的一切都是文件，每个文件都有一个整数类型的文件描述符；socket 也是一个文件，也有文件描述符。使用 socket() 函数创建套接字以后，返回值就是一个 int 类型的文件描述符。

Windows 会区分 socket 和普通文件，它把 socket 当做一个网络连接来对待，调用 socket() 以后，返回值是 SOCKET 类型，用来表示一个套接字。

Linux 下的 socket() 函数

在 Linux 下使用 <sys/socket.h> 头文件中 socket() 函数来创建套接字，原型为：

```
int socket(int af, int type, int protocol);
```

1) af 为地址族（Address Family），也就是 IP 地址类型，常用的有 AF_INET 和 AF_INET6。AF 是“Address Family”的简写，INET 是“Inetnet”的简写。AF_INET 表示 IPv4 地址，例如 127.0.0.1；AF_INET6 表示 IPv6 地址，例如 1030::C9B4:FF12:48AA:1A2B。

大家需要记住 127.0.0.1，它是一个特殊 IP 地址，表示本机地址，后面的教程会经常用到。

你也可以使用 PF 前缀，PF 是“Protocol Family”的简写，它和 AF 是一样的。例如，PF_INET 等价于 AF_INET，PF_INET6 等价于 AF_INET6。

2) type 为数据传输方式/套接字类型，常用的有 SOCK_STREAM（流格式套接字/面向连接的套接字）和 SOCK_DGRAM（数据报套接字/无连接的套接字），我们已经在《套接字有哪些类型》一节中进行了介绍。

3) protocol 表示传输协议，常用的有 IPPROTO_TCP 和 IPPTOTO_UDP，分别表示 TCP 传输协议和 UDP 传输协议。

有了地址类型和数据传输方式，还不足以决定采用哪种协议吗？为什么还需要第三个参数呢？

正如大家所想，一般情况下有了 af 和 type 两个参数就可以创建套接字了，操作系统会自动推演出协议类型，除非遇到这样的情况：有两种不同的协议支持同一种地址类型和数据传输类型。如果我们不指明使用哪种协议，操作系统是没办法自动推演的。

本教程使用 IPv4 地址，参数 af 的值为 PF_INET。如果使用 SOCK_STREAM 传输数据，那么满足这两个条件的协议只有 TCP，因此可以这样来调用 socket() 函数：

```
int tcp_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);  //IPPROTO_TCP 表示 TCP 协议
```

这种套接字称为 TCP 套接字。

如果使用 SOCK_DGRAM 传输方式，那么满足这两个条件的协议只有 UDP，因此可以这样来调用 socket() 函数：

```
int udp_socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);  //IPPROTO_UDP 表示 UDP 协议
```

这种套接字称为 UDP 套接字。

上面两种情况都只有一种协议满足条件，可以将 protocol 的值设为 0，系统会自动推演出应该使用什么协议，如下所示：

```
int tcp_socket = socket(AF_INET, SOCK_STREAM, 0);  //创建 TCP 套接字
int udp_socket = socket(AF_INET, SOCK_DGRAM, 0);  //创建 UDP 套接字
```

后面的教程中多采用这种简化写法。

在 Windows 下创建 socket

Windows 下也使用 socket() 函数来创建套接字，原型为：

```
SOCKET socket(int af, int type, int protocol);
```

除了返回值类型不同，其他都是相同的。Windows 不把套接字作为普通文件对待，而是返回 SOCKET 类型的句柄。请看下面的例子：

```
SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);  //创建 TCP 套接字
```

11. bind()和 connect()函数：绑定套接字并建立连接

socket() 函数用来创建套接字，确定套接字的各种属性，然后服务器端要用 bind() 函数将套接字与特定的 IP 地址和端口绑定起来，只有这样，流经该 IP 地址和端口的数据才能交给套接字处理。类似地，客户端也要用 connect() 函数建立连接。

bind() 函数

bind() 函数的原型为：

```
int bind(int sock, struct sockaddr *addr, socklen_t addrlen);  //Linux
int bind(SOCKET sock, const struct sockaddr *addr, int addrlen);  //Windows
```

下面以 Linux 为例进行讲解，Windows 与此类似。

sock 为 socket 文件描述符，addr 为 sockaddr 结构体变量的指针，addrlen 为 addr 变量的大小，可由 sizeof() 计算得出。

下面的代码，将创建的套接字与 IP 地址 127.0.0.1、端口 1234 绑定：

```
//创建套接字
int serv_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

//创建 sockaddr_in 结构体变量
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));  //每个字节都用 0 填充
serv_addr.sin_family = AF_INET;  //使用 IPv4 地址
serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的 IP 地址
serv_addr.sin_port = htons(1234);  //端口

//将套接字和 IP、端口绑定
bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
```

这里我们使用 sockaddr_in 结构体，然后再强制转换为 sockaddr 类型，后边会讲解为什么这样做。

sockaddr_in 结构体

接下来不妨先看一下 sockaddr_in 结构体，它的成员变量如下：

```
struct sockaddr_in{
    sa_family_t    sin_family;   //地址族（Address Family），也就是地址类型
    uint16_t       sin_port;     //16 位的端口号
    struct in_addr  sin_addr;    //32 位 IP 地址
```

```
    char            sin_zero[8];  //不使用，一般用 0 填充
};
```

1) sin_family 和 socket() 的第一个参数的含义相同，取值也要保持一致。

2) sin_prot 为端口号。uint16_t 的长度为两个字节，理论上端口号的取值范围为 0~65536，但 0~1023 的端口一般由系统分配给特定的服务程序，例如 Web 服务的端口号为 80，FTP 服务的端口号为 21，所以我们的程序要尽量在 1024~65536 之间分配端口号。

端口号需要用 htons() 函数转换，后面会讲解为什么。

3) sin_addr 是 struct in_addr 结构体类型的变量，下面会详细讲解。

4) sin_zero[8] 是多余的 8 个字节，没有用，一般使用 memset() 函数填充为 0。上面的代码中，先用 memset() 将结构体的全部字节填充为 0，再给前 3 个成员赋值，剩下的 sin_zero 自然就是 0 了。

in_addr 结构体

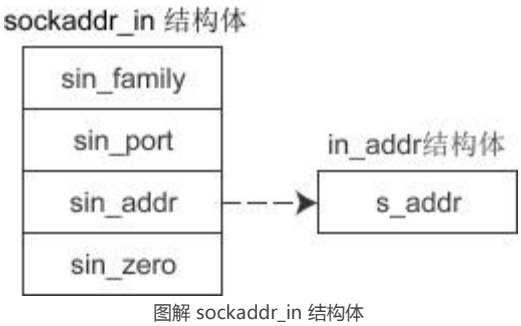
sockaddr_in 的第 3 个成员是 in_addr 类型的结构体，该结构体只包含一个成员，如下所示：

```
struct in_addr{
    in_addr_t  s_addr;  //32 位的 IP 地址
};
```

in_addr_t 在头文件 <netinet/in.h> 中定义，等价于 unsigned long，长度为 4 个字节。也就是说，s_addr 是一个整数，而 IP 地址是一个字符串，所以需要 inet_addr() 函数进行转换，例如：

```
unsigned long ip = inet_addr("127.0.0.1");
printf("%ld\n", ip);
```

运行结果：
16777343



图解 sockaddr_in 结构体

为什么要搞这么复杂，结构体中嵌套结构体，而不用 sockaddr_in 的一个成员变量来指明 IP 地址呢？socket() 函数的第一个参数已经指明了地址类型，为什么在 sockaddr_in 结构体中还要再说明一次呢，这不是啰嗦吗？

这些繁琐的细节确实给初学者带来了一定的障碍，我想，这或许是历史原因吧，后面的接口总要兼容前面的代码。各位读者一定要有耐心，暂时不理解没有关系，根据教程中的代码“照猫画虎”即可，时间久了自然会接受。

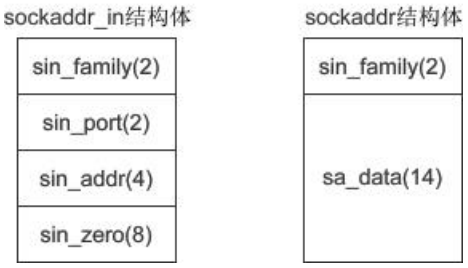
为什么使用 sockaddr_in 而不使用 sockaddr

bind() 第二个参数的类型为 sockaddr，而代码中却使用 sockaddr_in，然后再强制转换为 sockaddr，这是为什么呢？

sockaddr 结构体的定义如下：

```
struct sockaddr{
    sa_family_t  sin_family;   //地址族（Address Family），也就是地址类型
    char         sa_data[14];  //IP 地址和端口号
};
```

下图是 sockaddr 与 sockaddr_in 的对比（括号中的数字表示所占用的字节数）：



sockaddr 和 sockaddr_in 的长度相同，都是 16 字节，只是将 IP 地址和端口号合并到一起，用一个成员 sa_data 表示。要想给 sa_data 赋值，必须同时指明 IP 地址和端口号，例如”127.0.0.1:80 “，遗憾的是，没有相关函数将这个字符串转换成需要的形式，也就很难给 sockaddr 类型的变量赋值，所以使用 sockaddr_in 来代替。这两个结构体的长度相同，强制转换类型时不会丢失字节，也没有多余的字节。

可以认为，sockaddr 是一种通用的结构体，可以用来保存多种类型的 IP 地址和端口号，而 sockaddr_in 是专门用来保存 IPv4 地址的结构体。另外还有 sockaddr_in6，用来保存 IPv6 地址，它的定义如下：

```
struct sockaddr_in6 {
    sa_family_t sin6_family;  //(2)地址类型，取值为 AF_INET6
    in_port_t sin6_port;  //(2)16 位端口号
    uint32_t sin6_flowinfo;  //(4)IPv6 流信息
    struct in6_addr sin6_addr;  //(4)具体的 IPv6 地址
    uint32_t sin6_scope_id;  //(4)接口范围 ID
};
```

正是由于通用结构体 sockaddr 使用不便，才针对不同的地址类型定义了不同的结构体。

connect() 函数

connect() 函数用来建立连接，它的原型为：

```
int connect(int sock, struct sockaddr *serv_addr, socklen_t addrlen);  //Linux
int connect(SOCKET sock, const struct sockaddr *serv_addr, int addrlen);  //Windows
```

各个参数的说明和 bind() 相同，不再赘述。

12. listen()和 accept()函数：让套接字进入监听状态并响应客户端请求

对于服务器端程序，使用 bind() 绑定套接字后，还需要使用 listen() 函数让套接字进入被动监听状态，再调用 accept() 函数，就可以随时响应客户端的请求了。

listen() 函数

通过 listen() 函数可以让套接字进入被动监听状态，它的原型为：

```
1.  int listen(int sock, int backlog);  //Linux
2.  int listen(SOCKET sock, int backlog);  //Windows
```

sock 为需要进入监听状态的套接字，backlog 为请求队列的最大长度。

所谓被动监听，是指当没有客户端请求时，套接字处于“睡眠”状态，只有当接收到客户端请求时，套接字才会被“唤醒”来响应请求。

请求队列

当套接字正在处理客户端请求时，如果有新的请求进来，套接字是没法处理的，只能把它放进缓冲区，待当前请求处理完毕后，再从缓冲区中读取出来处理。如果不断有新的请求进来，它们就按照先后顺序在缓冲区中排队，直到缓冲区满。这个缓冲区，就称为请求队列（Request Queue）。

缓冲区的长度（能存放多少个客户端请求）可以通过 listen() 函数的 backlog 参数指定，但究竟为多少并没有什么标准，可以根据你的需求来定，并发量小的话可以是 10 或者 20。

如果将 backlog 的值设置为 SOMAXCONN，就由系统来决定请求队列长度，这个值一般比较大，可能是几百，或者更多。

当请求队列满时，就不再接收新的请求，对于 Linux，客户端会收到 ECONNREFUSED 错误，对于 Windows，客户端会收到 WSAECONNREFUSED 错误。

注意：listen() 只是让套接字处于监听状态，并没有接收请求。接收请求需要使用 accept() 函数。

accept() 函数

当套接字处于监听状态时，可以通过 accept() 函数来接收客户端请求。它的原型为：

```
1.  int accept(int sock, struct sockaddr *addr, socklen_t *addrlen);  //Linux
2.  SOCKET accept(SOCKET sock, struct sockaddr *addr, int *addrlen);  //Windows
```

它的参数与 listen() 和 connect() 是相同的：sock 为服务器端套接字，addr 为 sockaddr_in 结构体变量，addrlen 为参数 addr 的长度，可由 sizeof() 求得。

accept() 返回一个新的套接字来和客户端通信，addr 保存了客户端的 IP 地址和端口号，而 sock 是服务器端的套接字，大家注意区分。后面和客户端通信时，要使用这个新生成的套接字，而不是原来服务器端的套接字。

最后需要说明的是：listen() 只是让套接字进入监听状态，并没有真正接收客户端请求，listen() 后面的代码会继续执行，直到遇到 accept()。accept() 会阻塞程序执行（后面代码不能被执行），直到有新的请求到来。

13. send()/recv()和 write()/read()：发送数据和接收数据

在 Linux 和 Windows 平台下，使用不同的函数发送和接收 socket 数据，下面我们分别讲解。

Linux 下数据的接收和发送

Linux 不区分套接字文件和普通文件，使用 write() 可以向套接字中写入数据，使用 read() 可以从套接字中读取数据。

前面我们说过，两台计算机之间的通信相当于两个套接字之间的通信，在服务器端用 write() 向套接字写入数据，客户端就能收到，然后再使用 read() 从套接字中读取出来，就完成了一次通信。

write() 的原型为：

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

fd 为要写入的文件的描述符，buf 为要写入的数据的缓冲区地址，nbytes 为要写入的数据的字节数。

size_t 是通过 typedef 声明的 unsigned int 类型；ssize_t 在 "size_t" 前面加了一个"s"，代表 signed，即 ssize_t 是通过 typedef 声明的 signed int 类型。

write() 函数会将缓冲区 buf 中的 nbytes 个字节写入文件 fd，成功则返回写入的字节数，失败则返回 -1。

read() 的原型为：

```
ssize_t read(int fd, void *buf, size_t nbytes);
```

fd 为要读取的文件的描述符，buf 为要接收数据的缓冲区地址，nbytes 为要读取的数据的字节数。

read() 函数会从 fd 文件中读取 nbytes 个字节并保存到缓冲区 buf，成功则返回读取到的字节数（但遇到文件结尾则返回 0），失败则返回 -1。

Windows 下数据的接收和发送

Windows 和 Linux 不同，Windows 区分普通文件和套接字，并定义了专门的接收和发送的函数。

从服务器端发送数据使用 send() 函数，它的原型为：

```
int send(SOCKET sock, const char *buf, int len, int flags);
```

sock 为要发送数据的套接字，buf 为要发送的数据的缓冲区地址，len 为要发送的数据的字节数，flags 为发送数据时的选项。

返回值和前三个参数不再赘述，最后的 flags 参数一般设置为 0 或 NULL，初学者不必深究。

在客户端接收数据使用 recv() 函数，它的原型为：

```
int recv(SOCKET sock, char *buf, int len, int flags);
```

14. 使用 socket 编程实现回声客户端

所谓“回声”，是指客户端向服务器发送一条数据，服务器再将数据原样返回给客户端，就像声音一样，遇到障碍物会被“反弹回来”。

对！客户端也可以使用 write() / send() 函数向服务器发送数据，服务器也可以使用 read() / recv() 函数接收数据。

考虑到大部分初学者使用 Windows 操作系统，本节将实现 Windows 下的回声程序，Linux 下稍作修改即可，不再给出代码。

服务器端 server.cpp：

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加载 ws2_32.dll

#define BUF_SIZE 100

int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    //创建套接字
    SOCKET servSock = socket(AF_INET, SOCK_STREAM, 0);

    //绑定套接字
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用 0 填充
    sockAddr.sin_family = PF_INET;  //使用 IPv4 地址
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的 IP 地址
    sockAddr.sin_port = htons(1234);  //端口
    bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));

    //进入监听状态
    listen(servSock, 20);

    //接收客户端请求
    SOCKADDR clntAddr;
    int nSize = sizeof(SOCKADDR);
    SOCKET clntSock = accept(servSock, (SOCKADDR*)&clntAddr, &nSize);
    char buffer[BUF_SIZE];  //缓冲区
```

```
    int strLen = recv(clntSock, buffer, BUF_SIZE, 0);  //接收客户端发来的数据
    send(clntSock, buffer, strLen, 0);  //将数据原样返回

    //关闭套接字
    closesocket(clntSock);
    closesocket(servSock);

    //终止 DLL 的使用
    WSACleanup();

    return 0;
}
```

客户端 client.cpp：

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")  //加载 ws2_32.dll

#define BUF_SIZE 100

int main(){
    //初始化 DLL
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    //创建套接字
    SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

    //向服务器发起请求
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用 0 填充
    sockAddr.sin_family = PF_INET;
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    sockAddr.sin_port = htons(1234);
```

```
    connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
    //获取用户输入的字符串并发送给服务器
    char bufSend[BUF_SIZE] = {0};
    printf("Input a string: ");
    scanf("%s", bufSend);
    send(sock, bufSend, strlen(bufSend), 0);
    //接收服务器传回的数据
    char bufRecv[BUF_SIZE] = {0};
    recv(sock, bufRecv, BUF_SIZE, 0);

    //输出接收到的数据
    printf("Message form server: %s\n", bufRecv);

    //关闭套接字
    closesocket(sock);

    //终止使用 DLL
    WSACleanup();

    system("pause");
    return 0;
}
```

先运行服务器端，再运行客户端，执行结果为：
Input a string: c-language java cpp↙
Message form server: c-language

scanf() 读取到空格时认为一个字符串输入结束，所以只能读取到“c-language”；如果不希望把空格作为字符串的结束符，可以使用 gets() 函数。

通过本程序可以发现，客户端也可以向服务器端发送数据，这样服务器端就可以根据不同的请求作出不同的响应，http 服务器就是典型的例子，请求的网址不同，返回的页面也不同。

15. 如何让服务器端持续不断地监听客户端的请求？

前面的程序，不管服务器端还是客户端，都有一个问题，就是处理完一个请求立即退出了，没有太大的实际意义。能不能像 Web 服务器那样一直接受客户端的请求呢？能，使用 while 循环即可。

修改前面的回声程序，使服务器端可以不断响应客户端的请求。

服务器端 server.cpp：

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加载 ws2_32.dll

#define BUF_SIZE 100

int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    //创建套接字
    SOCKET servSock = socket(AF_INET, SOCK_STREAM, 0);

    //绑定套接字
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用 0 填充
    sockAddr.sin_family = PF_INET;  //使用 IPv4 地址
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的 IP 地址
    sockAddr.sin_port = htons(1234);  //端口
    bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));

    //进入监听状态
    listen(servSock, 20);

    //接收客户端请求
    SOCKADDR clntAddr;
    int nSize = sizeof(SOCKADDR);
    char buffer[BUF_SIZE] = {0};  //缓冲区
    while(1){
        SOCKET clntSock = accept(servSock, (SOCKADDR*)&clntAddr, &nSize);
```

```
        int strLen = recv(clntSock, buffer, BUF_SIZE, 0);  //接收客户端发来的数据
        send(clntSock, buffer, strLen, 0);  //将数据原样返回

        closesocket(clntSock);  //关闭套接字
        memset(buffer, 0, BUF_SIZE);  //重置缓冲区
    }

    //关闭套接字
    closesocket(servSock);

    //终止 DLL 的使用
    WSACleanup();

    return 0;
}
```

客户端 client.cpp：

```
#include <stdio.h>
#include <WinSock2.h>
#include <windows.h>
#pragma comment(lib, "ws2_32.lib")  //加载 ws2_32.dll

#define BUF_SIZE 100

int main(){
    //初始化 DLL
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    //向服务器发起请求
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用 0 填充
    sockAddr.sin_family = PF_INET;
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    sockAddr.sin_port = htons(1234);
```

```
19.
20.     char bufSend[BUF_SIZE] = {0};
21.     char bufRecv[BUF_SIZE] = {0};
22.
23.     while(1){
24.         //创建套接字
25.         SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
26.         connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
27.         //获取用户输入的字符串并发送给服务器
28.         printf("Input a string: ");
29.         gets(bufSend);
30.         send(sock, bufSend, strlen(bufSend), 0);
31.         //接收服务器传回的数据
32.         recv(sock, bufRecv, BUF_SIZE, 0);
33.         //输出接收到的数据
34.         printf("Message form server: %s\n", bufRecv);
35.
36.         memset(bufSend, 0, BUF_SIZE);  //重置缓冲区
37.         memset(bufRecv, 0, BUF_SIZE);  //重置缓冲区
38.         closesocket(sock);  //关闭套接字
39.     }
40.
41.     WSACleanup();  //终止使用 DLL
42.     return 0;
43. }
```

先运行服务器端，再运行客户端，结果如下：
Input a string: c language
Message form server: c language
Input a string: C 语言中文网
Message form server: C 语言中文网
Input a string: 学习 C/C++编程的好网站
Message form server: 学习 C/C++编程的好网站

while(1) 让代码进入死循环，除非用户关闭程序，否则服务器端会一直监听客户端的请求。客户端也是一样，会不断向服务器发起连接。

需要注意的是：server.cpp 中调用 closesocket() 不仅会关闭服务器端的 socket，还会通知客户端连接已断开，客户端也会清理 socket 相关资源，所以 client.cpp 中需要将 socket() 放在 while 循环内部，因为每次请求完毕都会清理 socket，下次发起请求时需要重新创建。后续我们会进行详细讲解。

16. socket 缓冲区以及阻塞模式详解

在《socket 数据的接收和发送》一节中讲到，可以使用 write()/send() 函数发送数据，使用 read()/recv() 函数接收数据，本节就来看看数据是如何传递的。

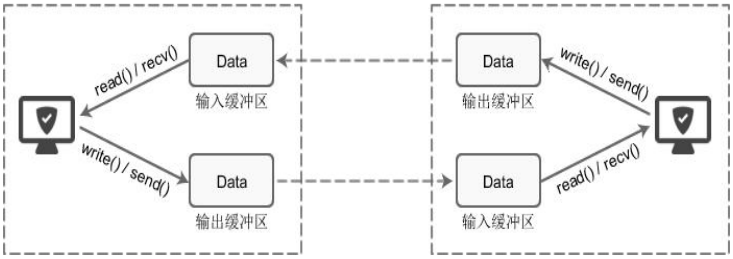
socket 缓冲区

每个 socket 被创建后，都会分配两个缓冲区，输入缓冲区和输出缓冲区。

write()/send() 并不立即向网络中传输数据，而是先将数据写入缓冲区中，再由 TCP 协议将数据从缓冲区发送到目标机器。一旦将数据写入到缓冲区，函数就可以成功返回，不管它们有没有到达目标机器，也不管它们何时被发送到网络，这些都是 TCP 协议负责的事情。

TCP 协议独立于 write()/send() 函数，数据有可能刚被写入缓冲区就发送到网络，也可能在缓冲区中不断积压，多次写入的数据被一次性发送到网络，这取决于当时的网络情况、当前线程是否空闲等诸多因素，不由程序员控制。

read()/recv() 函数也是如此，也从输入缓冲区中读取数据，而不是直接从网络中读取。



图：TCP 套接字的 I/O 缓冲区示意图

这些 I/O 缓冲区特性可整理如下：

- I/O 缓冲区在每个 TCP 套接字中单独存在；
- I/O 缓冲区在创建套接字时自动生成；
- 即使关闭套接字也会继续传送输出缓冲区中遗留的数据；
- 关闭套接字将丢失输入缓冲区中的数据。

输入输出缓冲区的默认大小一般都是 8K，可以通过 getsockopt() 函数获取：

```
unsigned optVal;
int optLen = sizeof(int);
getsockopt(servSock, SOL_SOCKET, SO_SNDBUF, (char*)&optVal, &optLen);
printf("Buffer length: %d\n", optVal);
```

运行结果：
Buffer length: 8192

这里仅给出示例，后面会详细讲解。

阻塞模式

对于 TCP 套接字（默认情况下），当使用 write()/send() 发送数据时：
1) 首先会检查缓冲区，如果缓冲区的可用空间长度小于要发送的数据，那么 write()/send() 会被阻塞（暂停执行），直到缓冲区中的数据被发送到目标机器，腾出足够的空间，才唤醒 write()/send() 函数继续写入数据。

2) 如果 TCP 协议正在向网络发送数据，那么输出缓冲区会被锁定，不允许写入，write()/send() 也会被阻塞，直到数据发送完毕缓冲区解锁，write()/send() 才会被唤醒。

3) 如果要写入的数据大于缓冲区的最大长度，那么将分批写入。

4) 直到所有数据被写入缓冲区 write()/send() 才能返回。

当使用 read()/recv() 读取数据时：
1) 首先会检查缓冲区，如果缓冲区中有数据，那么就读取，否则函数会被阻塞，直到网络上有数据到来。

2) 如果要读取的数据长度小于缓冲区中的数据长度，那么就不能一次性将缓冲区中的所有数据读出，剩余数据将不断积压，直到有 read()/recv() 函数再次读取。

3) 直到读取到数据后 read()/recv() 函数才会返回，否则就一直被阻塞。

这就是 TCP 套接字的阻塞模式。所谓阻塞，就是上一步动作没有完成，下一步动作将暂停，直到上一步动作完成后才能继续，以保持同步性。

TCP 套接字默认情况下是阻塞模式，也是最常用的。当然你也可以更改为非阻塞模式，后续我们会讲解。

17. TCP 协议的粘包问题（数据的无边界性）

上节我们讲到了 socket 缓冲区和数据的传递过程，可以看到数据的接收和发送是无关的，read()/recv() 函数不管数据发送了多少次，都会尽可能多的接收数据。也就是说，read()/recv() 和 write()/send() 的执行次数可能不同。

例如，write()/send() 重复执行三次，每次都发送字符串"abc"，那么目标机器上的 read()/recv() 可能分三次接收，每次都接收"abc"；也可能分两次接收，第一次接收"abcab"，第二次接收"cabc"；也可能一次就接收到字符串"abcabcabc"。

假设我们希望客户端每次发送一位学生的学号，让服务器端返回该学生的姓名、住址、成绩等信息，这时候可能就会出现问题，服务器端不能区分学生的学号。例如第一次发送 1，第二次发送 3，服务器可能当成 13 来处理，返回的信息显然是错误的。

这就是数据的“粘包”问题，客户端发送的多个数据包被当做一个数据包接收。也称数据的无边界性，read()/recv() 函数不知道数据包的开始或结束标志（实际上也没有任何开始或结束标志），只把它们当做连续的数据流来处理。

下面的代码演示了粘包问题，客户端连续三次向服务器端发送数据，服务器端却一次性接收到所有数据。

服务器端代码 server.cpp：

```
#include <stdio.h>
#include <windows.h>
#pragma comment (lib, "ws2_32.lib")  //加载 ws2_32.dll

#define BUF_SIZE 100

int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    //创建套接字
    SOCKET servSock = socket(AF_INET, SOCK_STREAM, 0);

    //绑定套接字
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用 0 填充
    sockAddr.sin_family = PF_INET;  //使用 IPv4 地址
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的 IP 地址
    sockAddr.sin_port = htons(1234);  //端口
    bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));

    //进入监听状态
    listen(servSock, 20);

```

```
    //接收客户端请求
    SOCKADDR clntAddr;
    int nSize = sizeof(SOCKADDR);
    char buffer[BUF_SIZE] = {0};  //缓冲区
    SOCKET clntSock = accept(servSock, (SOCKADDR*)&clntAddr, &nSize);

    Sleep(10000);  //注意这里，让程序暂停 10 秒

    //接收客户端发来的数据，并原样返回
    int recvLen = recv(clntSock, buffer, BUF_SIZE, 0);
    send(clntSock, buffer, recvLen, 0);

    //关闭套接字并终止 DLL 的使用
    closesocket(clntSock);
    closesocket(servSock);
    WSACleanup();

    return 0;
}
```

客户端代码 client.cpp：

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#include <windows.h>
#pragma comment(lib, "ws2_32.lib")  //加载 ws2_32.dll

#define BUF_SIZE 100

int main(){
    //初始化 DLL
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    //向服务器发起请求
```

```
15.     sockaddr_in sockAddr;
16.     memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用 0 填充
17.     sockAddr.sin_family = PF_INET;
18.     sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
19.     sockAddr.sin_port = htons(1234);
20.
21.     //创建套接字
22.     SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
23.     connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
24.
25.     //获取用户输入的字符串并发送给服务器
26.     char bufSend[BUF_SIZE] = {0};
27.     printf("Input a string: ");
28.     gets(bufSend);
29.     for(int i=0; i<3; i++){
30.         send(sock, bufSend, strlen(bufSend), 0);
31.     }
32.     //接收服务器传回的数据
33.     char bufRecv[BUF_SIZE] = {0};
34.     recv(sock, bufRecv, BUF_SIZE, 0);
35.     //输出接收到的数据
36.     printf("Message form server: %s\n", bufRecv);
37.
38.     closesocket(sock);  //关闭套接字
39.     WSACleanup();  //终止使用 DLL
40.
41.     system("pause");
42.     return 0;
43. }
```

先运行 server，再运行 client，并在 10 秒内输入字符串"abc"，再等数秒，服务器就会返回数据。运行结果如下：
Input a string: abc
Message form server: abcabcabc

本程序的关键是 server.cpp 第 31 行的代码 Sleep(10000);，它让程序暂停执行 10 秒。在这段时间内，client 连续三次发送字符串"abc"，由于 server 被阻塞，数据只能堆积在缓冲区中，10 秒后，server 开始运行，从缓冲区中一次性读出所有积压的数据，并返回给客户端。

另外还需要说明的是 client.cpp 第 34 行代码。client 执行到 recv() 函数，由于输入缓冲区中没有数据，所以会被阻塞，直到 10 秒后

server 传回数据才开始执行。用户看到的直观效果就是，client 暂停一段时间才输出 server 返回的结果。

client 的 send() 发送了三个数据包，而 server 的 recv() 却只接收到一个数据包，这很好的说明了数据的粘包问题。

18. 图解 TCP 数据报结构以及三次握手（非常详细）

TCP（Transmission Control Protocol，传输控制协议）是一种面向连接的、可靠的、基于字节流的通信协议，数据在传输前要建立连接，传输完毕后还要断开连接。

客户端在收发数据前要使用 connect() 函数和服务器建立连接。建立连接的目的是保证 IP 地址、端口、物理链路等正确无误，为数据的传输开辟通道。

TCP 建立连接时要传输三个数据包，俗称三次握手（Three-way Handshaking）。可以形象的比喻为下面的对话：

- [Shake 1] 套接字 A：“你好，套接字 B，我这里有数据要传送给你，建立连接吧。”
- [Shake 2] 套接字 B：“好的，我这边已准备就绪。”
- [Shake 3] 套接字 A：“谢谢你受理我的请求。”

TCP 数据报结构

我们先来看一下 TCP 数据报的结构：

带阴影的几个字段需要重点说明一下：

1) 序号：Seq（Sequence Number）序号占 32 位，用来标识从计算机 A 发送到计算机 B 的数据包的序号，计算机发送数据时对此进行标记。

2) 确认号：Ack（Acknowledge Number）确认号占 32 位，客户端和服务器端都可以发送，Ack = Seq + 1。

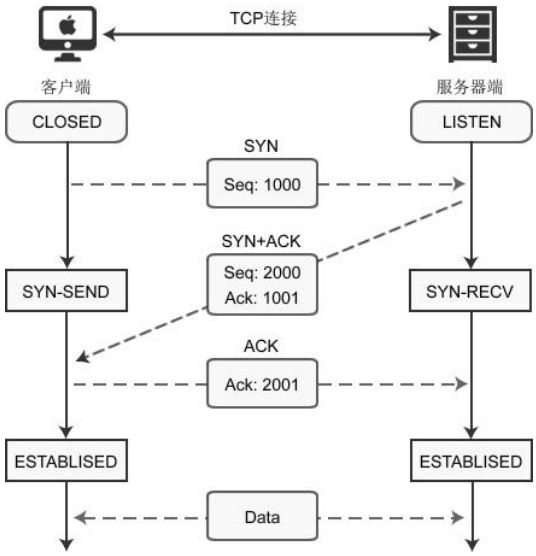
3) 标志位：每个标志位占用 1Bit，共有 6 个，分别为 URG、ACK、PSH、RST、SYN、FIN，具体含义如下：

- URG：紧急指针（urgent pointer）有效。
- ACK：确认序号有效。
- PSH：接收方应该尽快将这个报文交给应用层。
- RST：重置连接。
- SYN：建立一个新连接。
- FIN：断开一个连接。

对英文字母缩写的总结：Seq 是 Sequence 的缩写，表示序列；Ack(ACK) 是 Acknowledge 的缩写，表示确认；SYN 是 Synchronous 的缩写，愿意是“同步的”，这里表示建立同步连接；FIN 是 Finish 的缩写，表示完成。

连接的建立（三次握手）

使用 connect() 建立连接时，客户端和服务器端会相互发送三个数据包，请看下图：



客户端调用 socket() 函数创建套接字后，因为没有建立连接，所以套接字处于 CLOSED 状态；服务器端调用 listen() 函数后，套接字进入 LISTEN 状态，开始监听客户端请求。

这个时候，客户端开始发起请求：
1) 当客户端调用 connect() 函数后，TCP 协议会组建一个数据包，并设置 SYN 标志位，表示该数据包是用来建立同步连接的。同时生成一个随机数字 1000，填充“序号（Seq）”字段，表示该数据包的序号。完成这些工作，开始向服务器端发送数据包，客户端就进入了 SYN-SEND 状态。

2) 服务器端收到数据包，检测到已经设置了 SYN 标志位，就知道这是客户端发来的建立连接的“请求包”。服务器端也会组建一个数据包，并设置 SYN 和 ACK 标志位，SYN 表示该数据包用来建立连接，ACK 用来确认收到了刚才客户端发送的数据包。

服务器生成一个随机数 2000，填充“序号（Seq）”字段。2000 和客户端数据包没有关系。

服务器将客户端数据包序号（1000）加 1，得到 1001，并用这个数字填充“确认号（Ack）”字段。

服务器将数据包发出，进入 SYN-RECV 状态。

3) 客户端收到数据包，检测到已经设置了 SYN 和 ACK 标志位，就知道这是服务器发来的“确认包”。客户端会检测“确认号（Ack）”字段，看它的值是否为 1000+1，如果是就说明连接建立成功。

接下来，客户端会继续组建数据包，并设置 ACK 标志位，表示客户端正确接收了服务器发来的“确认包”。同时，将刚才服务器发来的数据包序号（2000）加 1，得到 2001，并用这个数字来填充“确认号（Ack）”字段。

客户端将数据包发出，进入 ESTABLISED 状态，表示连接已经成功建立。

4) 服务器端收到数据包，检测到已经设置了 ACK 标志位，就知道这是客户端发来的“确认包”。服务器会检测“确认号（Ack）”字段，看它的值是否为 2000+1，如果是就说明连接建立成功，服务器进入 ESTABLISED 状态。

至此，客户端和服务器都进入了 ESTABLISED 状态，连接建立成功，接下来就可以收发数据了。

最后的说明

三次握手的关键是要确认对方收到了自己的数据包，这个目标就是通过“确认号（Ack）”字段实现的。计算机会记录下自己发送的数据包序号 Seq，待收到对方的数据包后，检测“确认号（Ack）”字段，看 `Ack = Seq + 1` 是否成立，如果成立说明对方正确收到了自己的数据包。

19. 详细分析 TCP 数据的传输过程

建立连接后，两台主机就可以相互传输数据了。如下图所示：

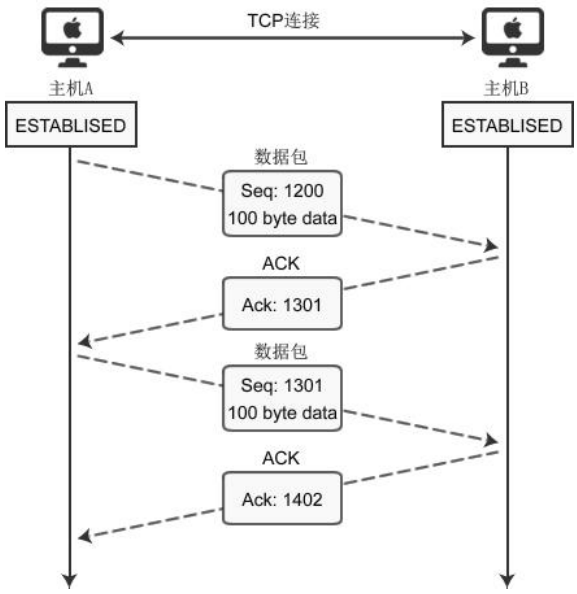


图 1：TCP 套接字的数据交换过程

上图给出了主机 A 分 2 次（分 2 个数据包）向主机 B 传递 200 字节的过程。首先，主机 A 通过 1 个数据包发送 100 个字节的数据，数据包的 Seq 号设置为 1200。主机 B 为了确认这一点，向主机 A 发送 ACK 包，并将 Ack 号设置为 1301。

为了保证数据准确到达，目标机器在收到数据包（包括 SYN 包、FIN 包、普通数据包等）包后必须立即回传 ACK 包，这样发送方才能确认数据传输成功。

此时 Ack 号为 1301 而不是 1201，原因在于 Ack 号的增量为传输的数据字节数。假设每次 Ack 号不加传输的字节数，这样虽然可以确认数据包的传输，但无法明确 100 字节全部正确传递还是丢失了一部分，比如只传递了 80 字节。因此按如下的公式确认 Ack 号：

```
Ack 号 = Seq 号 + 传递的字节数 + 1
```

与三次握手协议相同，最后加 1 是为了告诉对方要传递的 Seq 号。

下面分析传输过程中数据包丢失的情况，如下图所示：

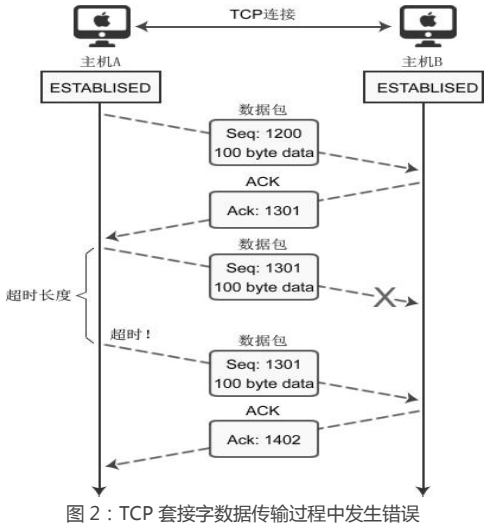


图 2：TCP 套接字数据传输过程中发生错误

上图表示通过 Seq 1301 数据包向主机 B 传递 100 字节的数据，但中间发生了错误，主机 B 未收到。经过一段时间后，主机 A 仍未收到对于 Seq 1301 的 ACK 确认，因此尝试重传数据。

为了完成数据包的重传，TCP 套接字每次发送数据包时都会启动定时器，如果在一定时间内没有收到目标机器传回的 ACK 包，那么定时器超时，数据包会重传。

上图演示的是数据包丢失的情况，也会有 ACK 包丢失的情况，一样会重传。

重传超时时间（RTO, Retransmission Time Out）

这个值太大了会导致不必要的等待，太小会导致不必要的重传，理论上最好是网络 RTT 时间，但又受制于网络距离与瞬态时延变化，所以实际上使用自适应的动态算法（例如 Jacobson 算法和 Karn 算法等）来确定超时时间。

往返时间（RTT，Round-Trip Time）表示从发送端发送数据开始，到发送端收到来自接收端的 ACK 确认包（接收端收到数据后便立即确认），总共经历的时延。

重传次数

TCP 数据包重传次数根据系统设置的不同而有所区别。有些系统，一个数据包只会被重传 3 次，如果重传 3 次后还未收到该数据包的 ACK 确认，就不再尝试重传。但有些要求很高的业务系统，会不断地重传丢失的数据包，以尽最大可能保证业务数据的正常交互。

最后需要说明的是，发送端只有在收到对方的 ACK 确认包后，才会清空输出缓冲区中的数据。

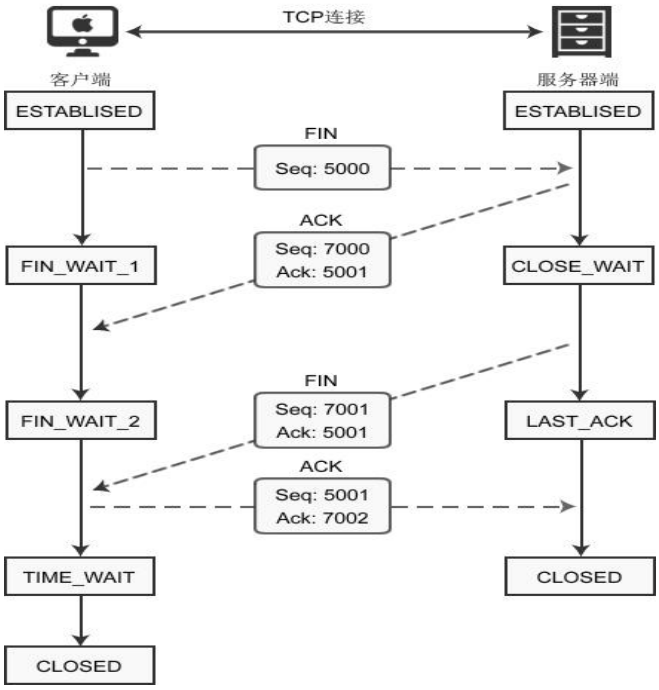
20. 图解 TCP 四次握手断开连接

建立连接非常重要，它是数据正确传输的前提；断开连接同样重要，它让计算机释放不再使用的资源。如果连接不能正常断开，不仅会造成数据传输错误，还会导致套接字不能关闭，持续占用资源，如果并发量高，服务器压力堪忧。

建立连接需要三次握手，断开连接需要四次握手，可以形象的比喻为下面的对话：

- [Shake 1] 套接字 A：“任务处理完毕，我希望断开连接。”
- [Shake 2] 套接字 B：“哦，是吗？请稍等，我准备一下。”
- 等待片刻后……
- [Shake 3] 套接字 B：“我准备好了，可以断开连接了。”
- [Shake 4] 套接字 A：“好的，谢谢合作。”

下图演示了客户端主动断开连接的场景：



建立连接后，客户端和服务器都处于 `ESTABLISED` 状态。这时，客户端发起断开连接的请求：
1) 客户端调用 close() 函数后，向服务器发送 FIN 数据包，进入 `FIN_WAIT_1` 状态。FIN 是 Finish 的缩写，表示完成任务需要断开连接。

2) 服务器收到数据包后，检测到设置了 FIN 标志位，知道要断开连接，于是向客户端发送“确认包”，进入 `CLOSE_WAIT` 状态。

注意：服务器收到请求后并不是立即断开连接，而是先向客户端发送“确认包”，告诉它我知道了，我需要准备一下才能断开连接。

3) 客户端收到“确认包”后进入 `FIN_WAIT_2` 状态，等待服务器准备完毕后再次发送数据包。

4) 等待片刻后，服务器准备完毕，可以断开连接，于是再主动向客户端发送 FIN 包，告诉它我准备好了，断开连接吧。然后进入 `LAST_ACK` 状态。

5) 客户端收到服务器的 FIN 包后，再向服务器发送 ACK 包，告诉它你断开连接吧。然后进入 `TIME_WAIT` 状态。

6) 服务器收到客户端的 ACK 包后，就断开连接，关闭套接字，进入 `CLOSED` 状态。

关于 TIME_WAIT 状态的说明

客户端最后一次发送 ACK 包后进入 TIME_WAIT 状态，而不是直接进入 CLOSED 状态关闭连接，这是为什么呢？

TCP 是面向连接的传输方式，必须保证数据能够正确到达目标机器，不能丢失或出错，而网络是不稳定的，随时可能会毁坏数据，所以机器 A 每次向机器 B 发送数据包后，都要求机器 B”确认“，回传 ACK 包，告诉机器 A 我收到了，这样机器 A 才能知道数据传送成功了。如果机器 B 没有回传 ACK 包，机器 A 会重新发送，直到机器 B 回传 ACK 包。

客户端最后一次向服务器回传 ACK 包时，有可能会因为网络问题导致服务器收不到，服务器会再次发送 FIN 包，如果这时客户端完全关闭了连接，那么服务器无论如何也收不到 ACK 包了，所以客户端需要等待片刻、确认对方收到 ACK 包后才能进入 CLOSED 状态。那么，要等待多久呢？

数据包在网络中是有生存时间的，超过这个时间还未到达目标主机就会被丢弃，并通知源主机。这称为报文最大生存时间（MSL，Maximum Segment Lifetime）。TIME_WAIT 要等待 2MSL 才会进入 CLOSED 状态。ACK 包到达服务器需要 MSL 时间，服务器重传 FIN 包也需要 MSL 时间，2MSL 是数据包往返的最大时间，如果 2MSL 后还未收到服务器重传的 FIN 包，就说明服务器已经收到了 ACK 包。

21. 如何优雅地断开 TCP 连接？

调用 close()/closesocket() 函数意味着完全断开连接，即不能发送数据也不能接收数据，这种“生硬”的方式有时候会显得不太“优雅”。

图 1：close()/closesocket() 断开连接

上图演示了两台正在进行双向通信的主机。主机 A 发送完数据后，单方面调用 close()/closesocket() 断开连接，之后主机 A、B 都不能再接受对方传输的数据。实际上，是完全无法调用与数据收发有关的函数。

一般情况下这不会有问题，但有些特殊时刻，需要只断开一条数据传输通道，而保留另一条。

使用 shutdown() 函数可以达到这个目的，它的原型为：

```
1.  int shutdown(int sock, int howto);  //Linux
2.  int shutdown(SOCKET s, int howto);  //Windows
```

sock 为需要断开的套接字，howto 为断开方式。

howto 在 Linux 下有以下取值：

- SHUT_RD：断开输入流。套接字无法接收数据（即使输入缓冲区收到数据也被抹去），无法调用输入相关函数。
- SHUT_WR：断开输出流。套接字无法发送数据，但如果输出缓冲区中还有未传输的数据，则将传递到目标主机。
- SHUT_RDWR：同时断开 I/O 流。相当于分两次调用 shutdown()，其中一次以 SHUT_RD 为参数，另一次以 SHUT_WR 为参数。

howto 在 Windows 下有以下取值：

- SD_RECEIVE：关闭接收操作，也就是断开输入流。
- SD_SEND：关闭发送操作，也就是断开输出流。
- SD_BOTH：同时关闭接收和发送操作。

至于什么时候需要调用 shutdown() 函数，下节我们会以文件传输为例进行讲解。

close()/closesocket()和 shutdown()的区别

确切地说，close() / closesocket() 用来关闭套接字，将套接字描述符（或句柄）从内存清除，之后再也不能使用该套接字，与 C 语言中的 fclose() 类似。应用程序关闭套接字后，与该套接字相关的连接和缓存也失去了意义，TCP 协议会自动触发关闭连接的操作。

shutdown() 用来关闭连接，而不是套接字，不管调用多少次 shutdown()，套接字依然存在，直到调用 close() / closesocket() 将套接字从内存清除。

调用 close()/closesocket() 关闭套接字时，或调用 shutdown() 关闭输出流时，都会向对方发送 FIN 包。FIN 包表示数据传输完毕，计算机收到 FIN 包就知道不会再有数据传送过来了。

默认情况下，close()/closesocket() 会立即向网络中发送 FIN 包，不管输出缓冲区中是否还有数据，而 shutdown() 会等输出缓冲区中的数据传输完毕再发送 FIN 包。也就意味着，调用 close()/closesocket() 将丢失输出缓冲区中的数据，而调用 shutdown() 不会。

22. socket 编程实现文件传输功能

这节我们来完成 socket 文件传输程序，这是一个非常实用的例子。要实现的功能为：client 从 server 下载一个文件并保存到本地。

编写这个程序需要注意两个问题：
1) 文件大小不确定，有可能比缓冲区大很多，调用一次 write()/send() 函数不能完成文件内容的发送。接收数据时也会遇到同样的情况。

要解决这个问题，可以使用 while 循环，例如：

```
//Server 代码
int nCount;
while( (nCount = fread(buffer, 1, BUF_SIZE, fp)) > 0 ){
    send(sock, buffer, nCount, 0);
}

//Client 代码
int nCount;
while( (nCount = recv(clntSock, buffer, BUF_SIZE, 0)) > 0 ){
    fwrite(buffer, nCount, 1, fp);
}
```

对于 Server 端的代码，当读取到文件末尾，fread() 会返回 0，结束循环。

对于 Client 端代码，有一个关键的问题，就是文件传输完毕后让 recv() 返回 0，结束 while 循环。
注意：读取完缓冲区中的数据 recv() 并不会返回 0，而是被阻塞，直到缓冲区中再次有数据。
2) Client 端如何判断文件接收完毕，也就是上面提到的问题——何时结束 while 循环。

最简单的结束 while 循环的方法当然是文件接收完毕后让 recv() 函数返回 0，那么，如何让 recv() 返回 0 呢？recv() 返回 0 的唯一时机就是收到 FIN 包时。

FIN 包表示数据传输完毕，计算机收到 FIN 包后就知道对方不会再向自己传输数据，当调用 read()/recv() 函数时，如果缓冲区中没有数据，就会返回 0，表示读到了”socket 文件的末尾“。

这里我们调用 shutdown() 来发送 FIN 包：server 端直接调用 close()/closesocket() 会使输出缓冲区中的数据失效，文件内容很有可能没有传输完毕连接就断开了，而调用 shutdown() 会等待输出缓冲区中的数据传输完毕。

本节以 Windows 为例演示文件传输功能，Linux 与此类似，不再赘述。请看下面完整的代码。

服务器端 server.cpp：

```
#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加载 ws2_32.dll

#define BUF_SIZE 1024
```

```
7.
8.  int main(){
9.      //先检查文件是否存在
10.     char *filename = "D:\\send.avi";  //文件名
11.     FILE *fp = fopen(filename, "rb");  //以二进制方式打开文件
12.     if(fp == NULL){
13.         printf("Cannot open file, press any key to exit!\n");
14.         system("pause");
15.         exit(0);
16.     }
17.
18.     WSADATA wsaData;
19.     WSAStartup( MAKEWORD(2, 2), &wsaData);
20.     SOCKET servSock = socket(AF_INET, SOCK_STREAM, 0);
21.
22.     sockaddr_in sockAddr;
23.     memset(&sockAddr, 0, sizeof(sockAddr));
24.     sockAddr.sin_family = PF_INET;
25.     sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
26.     sockAddr.sin_port = htons(1234);
27.     bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
28.     listen(servSock, 20);
29.
30.     SOCKADDR clntAddr;
31.     int nSize = sizeof(SOCKADDR);
32.     SOCKET clntSock = accept(servSock, (SOCKADDR*)&clntAddr, &nSize);
33.
34.     //循环发送数据，直到文件结尾
35.     char buffer[BUF_SIZE] = {0};  //缓冲区
36.     int nCount;
37.     while( (nCount = fread(buffer, 1, BUF_SIZE, fp)) > 0 ){
38.         send(clntSock, buffer, nCount, 0);
39.     }
40.
41.     shutdown(clntSock, SD_SEND);  //文件读取完毕，断开输出流，向客户端发送 FIN 包
42.     recv(clntSock, buffer, BUF_SIZE, 0);  //阻塞，等待客户端接收完毕
```

```
43.
44.     fclose(fp);
45.     closesocket(clntSock);
46.     closesocket(servSock);
47.     WSACleanup();
48.
49.     system("pause");
50.     return 0;
51. }
```

客户端代码：

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <WinSock2.h>
4.  #pragma comment(lib, "ws2_32.lib")
5.
6.  #define BUF_SIZE 1024
7.
8.  int main(){
9.      //先输入文件名，看文件是否能创建成功
10.     char filename[100] = {0};  //文件名
11.     printf("Input filename to save: ");
12.     gets(filename);
13.     FILE *fp = fopen(filename, "wb");  //以二进制方式打开（创建）文件
14.     if(fp == NULL){
15.         printf("Cannot open file, press any key to exit!\n");
16.         system("pause");
17.         exit(0);
18.     }
19.
20.     WSADATA wsaData;
21.     WSAStartup(MAKEWORD(2, 2), &wsaData);
22.     SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
23.
24.     sockaddr_in sockAddr;
```

```
25.     memset(&sockAddr, 0, sizeof(sockAddr));
26.     sockAddr.sin_family = PF_INET;
27.     sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
28.     sockAddr.sin_port = htons(1234);
29.     connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
30.
31.     //循环接收数据，直到文件传输完毕
32.     char buffer[BUF_SIZE] = {0};  //文件缓冲区
33.     int nCount;
34.     while( (nCount = recv(sock, buffer, BUF_SIZE, 0)) > 0 ){
35.         fwrite(buffer, nCount, 1, fp);
36.     }
37.     puts("File transfer success!");
38.
39.     //文件接收完毕后直接关闭套接字，无需调用 shutdown()
40.     fclose(fp);
41.     closesocket(sock);
42.     WSACleanup();
43.     system("pause");
44.     return 0;
45. }
```

在 D 盘中准备好 send.avi 文件，先运行 server，再运行 client：
Input filename to save: D:\\recv.avi↙
//稍等片刻后
File transfer success!

打开 D 盘就可以看到 recv.avi，大小和 send.avi 相同，可以正常播放。

注意 server.cpp 第 42 行代码，recv() 并没有接收到 client 端的数据，当 client 端调用 closesocket() 后，server 端会收到 FIN 包，recv() 就会返回，后面的代码继续执行。

23. 网络数据传输时的大小端问题

不同 CPU 中，4 字节整数 1 在内存空间的存储方式是不同的。4 字节整数 1 可用 2 进制表示如下：

```
00000000 00000000 00000000 00000001
```

有些 CPU 以上面的顺序存储到内存，另外一些 CPU 则以倒序存储，如下所示：

```
00000001 00000000 00000000 00000000
```

若不考虑这些就收发数据会发生问题，因为保存顺序的不同意味着对接收数据的解析顺序也不同。

大端序和小端序

CPU 向内存保存数据的方式有两种：

- 大端序（Big Endian）：高位字节存放到低位地址（高位字节在前）。
- 小端序（Little Endian）：高位字节存放到高位地址（低位字节在前）。

仅凭描述很难解释清楚，不妨来看一个实例。假设在 0x20 号开始的地址中保存 4 字节 int 型数据 0x12345678，大端序 CPU 保存方式如下图所示：

0x20号	0x21号	0x22号	0x23号
0x12	0x34	0x56	0x78

图 1：整数 0x12345678 的大端序字节表示

对于大端序，最高位字节 0x12 存放到低位地址，最低位字节 0x78 存放到高位地址。小端序的保存方式如下图所示：

0x20号	0x21号	0x22号	0x23号
0x78	0x56	0x34	0x12

图 2：整数 0x12345678 的小端序字节表示

不同 CPU 保存和解析数据的方式不同（主流的 Intel 系列 CPU 为小端序），小端序系统和大端序系统通信时会发生数据解析错误。因此在发送数据前，要将数据转换为统一的格式——网络字节序（Network Byte Order）。网络字节序统一为大端序。

主机 A 先把数据转换成大端序再进行网络传输，主机 B 收到数据后先转换为自己的格式再解析。

网络字节序转换函数

在《bind()和 connect()函数：绑定套接字并建立连接》一节中讲解了 sockaddr_in 结构体，其中就用到了网络字节序转换函数，如下所示：

```
1.  //创建 sockaddr_in 结构体变量
2.  struct sockaddr_in serv_addr;
3.  memset(&serv_addr, 0, sizeof(serv_addr));  //每个字节都用 0 填充
4.  serv_addr.sin_family = AF_INET;  //使用 IPv4 地址
```

```
serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的 IP 地址
serv_addr.sin_port = htons(1234);  //端口号
```

htons() 用来将当前主机字节序转换为网络字节序，其中 h 代表主机（host）字节序，n 代表网络（network）字节序，s 代表 short，htons 是 h、to、n、s 的组合，可以理解为”将 short 型数据从当前主机字节序转换为网络字节序“。

常见的网络字节转换函数有：

- htons()：host to network short，将 short 类型数据从主机字节序转换为网络字节序。
- ntohs()：network to host short，将 short 类型数据从网络字节序转换为主机字节序。
- htonl()：host to network long，将 long 类型数据从主机字节序转换为网络字节序。
- ntohl()：network to host long，将 long 类型数据从网络字节序转换为主机字节序。

通常，以 s 为后缀的函数中，s 代表 2 个字节 short，因此用于端口号转换；以 l 为后缀的函数中，l 代表 4 个字节的 long，因此用于 IP 地址转换。

举例说明上述函数的调用过程：

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")

int main(){
    unsigned short host_port = 0x1234, net_port;
    unsigned long host_addr = 0x12345678, net_addr;

    net_port = htons(host_port);
    net_addr = htonl(host_addr);

    printf("Host ordered port: %#x\n", host_port);
    printf("Network ordered port: %#x\n", net_port);
    printf("Host ordered address: %#lx\n", host_addr);
    printf("Network ordered address: %#lx\n", net_addr);

    system("pause");
    return 0;
}
```

运行结果：
Host ordered port: 0x1234
Network ordered port: 0x3412

Host ordered address: 0x12345678
Network ordered address: 0x78563412

另外需要说明的是，sockaddr_in 中保存 IP 地址的成员为 32 位整数，而我们熟悉的是点分十进制表示法，例如 127.0.0.1，它是一个字符串，因此为了分配 IP 地址，需要将字符串转换为 4 字节整数。

inet_addr() 函数可以完成这种转换。inet_addr() 除了将字符串转换为 32 位整数，同时还进行网络字节序转换。请看下面的代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")

int main(){
    char *addr1 = "1.2.3.4";
    char *addr2 = "1.2.3.256";

    unsigned long conv_addr = inet_addr(addr1);
    if(conv_addr == INADDR_NONE){
        puts("Error occured!");
    }else{
        printf("Network ordered integer addr: %#lx\n", conv_addr);
    }

    conv_addr = inet_addr(addr2);
    if(conv_addr == INADDR_NONE){
        puts("Error occured!");
    }else{
        printf("Network ordered integer addr: %#lx\n", conv_addr);
    }

    system("pause");
    return 0;
}
```

运行结果：
Network ordered integer addr: 0x4030201
Error occured!

从运行结果可以看出，inet_addr() 不仅可以把 IP 地址转换为 32 位整数，还可以检测无效 IP 地址。

注意：为 sockaddr_in 成员赋值时需要显式地将主机字节序转换为网络字节序，而通过 write()/send() 发送数据时 TCP 协议会自动转换为网络字节序，不需要再调用相应的函数。

24. 在 socket 编程中使用域名

客户端中直接使用 IP 地址会有很大的弊端，一旦 IP 地址变化（IP 地址会经常变动），客户端软件就会出现错误。

而使用域名会方便很多，注册后的域名只要每年续费就永远属于自己的，更换 IP 地址时修改域名解析即可，不会影响软件的正常使用。

关于域名注册、域名解析、host 文件、DNS 服务器等本节并未详细讲解，请读者自行脑补。本节重点讲解如何使用域名。

通过域名获取 IP 地址

域名仅仅是 IP 地址的一个助记符，目的是方便记忆，通过域名并不能找到目标计算机，通信之前必须要将域名转换成 IP 地址。

gethostbyname() 函数可以完成这种转换，它的原型为：

```
struct hostent *gethostbyname(const char *hostname);
```

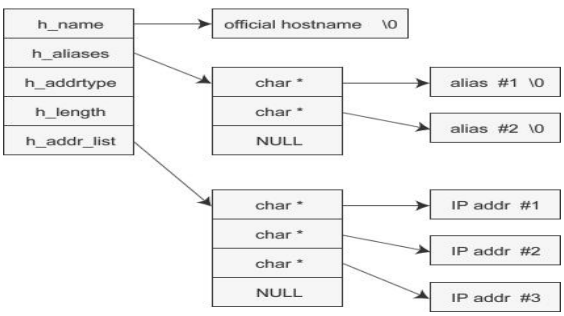
hostname 为主机名，也就是域名。使用该函数时，只要传递域名字符串，就会返回域名对应的 IP 地址。返回的地址信息会装入 hostent 结构体，该结构体的定义如下：

```
struct hostent{
    char *h_name;  //official name
    char **h_aliases;  //alias list
    int  h_addrtype;  //host address type
    int  h_length;  //address lenght
    char **h_addr_list;  //address list
}
```

从该结构体可以看出，不只返回 IP 地址，还会附带其他信息，各位读者只需关注最后一个成员 h_addr_list。下面是对各成员的说明：

- h_name：官方域名（Official domain name）。官方域名代表某一主页，但实际上一些著名公司的域名并未用官方域名注册。
- h_aliases：别名，可以通过多个域名访问同一主机。同一 IP 地址可以绑定多个域名，因此除了当前域名还可以指定其他域名。
- h_addrtype：gethostbyname() 不仅支持 IPv4，还支持 IPv6，可以通过此成员获取 IP 地址的地址族（地址类型）信息，IPv4 对应 AF_INET，IPv6 对应 AF_INET6。
- h_length：保存 IP 地址长度。IPv4 的长度为 4 个字节，IPv6 的长度为 16 个字节。
- h_addr_list：这是最重要的成员。通过该成员以整数形式保存域名对应的 IP 地址。对于用户较多的服务器，可能会分配多个 IP 地址给同一域名，利用多个服务器进行均衡负载。

hostent 结构体变量的组成如下图所示：



下面的代码主要演示 gethostbyname() 的应用，并说明 hostent 结构体的特性：

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")

int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    struct hostent *host = gethostbyname("www.baidu.com");
    if(!host){
        puts("Get IP address error!");
        system("pause");
        exit(0);
    }

    //别名
    for(int i=0; host->h_aliases[i]; i++){
        printf("Aliases %d: %s\n", i+1, host->h_aliases[i]);
    }

    //地址类型
    printf("Address type: %s\n", (host->h_addrtype==AF_INET) ? "AF_INET": "AF_INET6");

    //IP 地址
    for(int i=0; host->h_addr_list[i]; i++){
        printf("IP addr %d: %s\n", i+1, inet_ntoa( *(struct in_addr*)host->h_addr_list[i] ) );
    }

    system("pause");
    return 0;
}
```

运行结果：
Aliases 1: www.baidu.com

Address type: AF_INET
IP addr 1: 61.135.169.121
IP addr 2: 61.135.169.125

25. 再谈 UDP 和 TCP

TCP 是面向连接的传输协议，建立连接时要经过三次握手，断开连接时要经过四次握手，中间传输数据时也要回复 ACK 包确认，多种机制保证了数据能够正确到达，不会丢失或出错。

UDP 是非连接的传输协议，没有建立连接和断开连接的过程，它只是简单地把数据丢到网络中，也不需要 ACK 包确认。

UDP 传输数据就好像我们邮寄包裹，邮寄前需要填好寄件人和收件人地址，之后送到快递公司即可，但包裹是否正确送达、是否损坏我们无法得知，也无法保证。UDP 协议也是如此，它只管把数据包发送到网络，然后就不管了，如果数据丢失或损坏，发送端是无法知道的，当然也不会重发。

既然如此，TCP 应该是更加优质的传输协议吧？

如果只考虑可靠性，TCP 的确比 UDP 好。但 UDP 在结构上比 TCP 更加简洁，不会发送 ACK 的应答消息，也不会给数据包分配 Seq 序号，所以 UDP 的传输效率有时会比 TCP 高出很多，编程中实现 UDP 也比 TCP 简单。

UDP 的可靠性虽然比不上 TCP，但也不会像想象中那么频繁地发生数据损毁，在更加重视传输效率而非可靠性的情况下，UDP 是一种很好的选择。比如视频通信或音频通信，就非常适合采用 UDP 协议；通信时数据必须高效传输才不会产生“卡顿”现象，用户体验才更加流畅，如果丢失几个数据包，视频画面可能会出现“雪花”，音频可能会夹带一些杂音，这些都是无妨的。

与 UDP 相比，TCP 的生命在于流控制，这保证了数据传输的正确性。

最后需要说明的是：TCP 的速度无法超越 UDP，但在收发某些类型的数据时有可能接近 UDP。例如，每次交换的数据量越大，TCP 的传输速率就越接近于 UDP。

26. 基于 UDP 的服务器端和客户端

前面的文章中我们给出了几个 TCP 的例子，对于 UDP 而言，只要能理解前面的内容，实现并非难事。

UDP 中的服务器端和客户端没有连接

UDP 不像 TCP，无需在连接状态下交换数据，因此基于 UDP 的服务器端和客户端也无需经过连接过程。也就是说，不必调用 listen() 和 accept() 函数。UDP 中只有创建套接字的过程和数据交换的过程。

UDP 服务器端和客户端均只需 1 个套接字

TCP 中，套接字是一对一的关系。如要向 10 个客户端提供服务，那么除了负责监听的套接字外，还需要创建 10 套接字。但在 UDP 中，不管是服务器端还是客户端都只需要 1 个套接字。之前解释 UDP 原理的时候举了邮寄包裹的例子，负责邮寄包裹的快递公司可以比喻为 UDP 套接字，只要有 1 个快递公司，就可以通过它向任意地址邮寄包裹。同样，只需 1 个 UDP 套接字就可以向任意主机传送数据。

基于 UDP 的接收和发送函数

创建好 TCP 套接字后，传输数据时无需再添加地址信息，因为 TCP 套接字将保持与对方套接字的连接。换言之，TCP 套接字知道目标地址信息。但 UDP 套接字不会保持连接状态，每次传输数据都要添加目标地址信息，这相当于在邮寄包裹前填写收件人地址。

发送数据使用 sendto() 函数：

```
1.  ssize_t sendto(int sock, void *buf, size_t nbytes, int flags, struct sockaddr *to, socklen_t addrlen);  //Linux
2.  int sendto(SOCKET sock, const char *buf, int nbytes, int flags, const struct sockadr *to, int addrlen);  //Windows
```

Linux 和 Windows 下的 sendto() 函数类似，下面是详细参数说明：

- sock：用于传输 UDP 数据的套接字；
- buf：保存待传输数据的缓冲区地址；
- nbytes：带传输数据的长度（以字节计）；
- flags：可选项参数，若没有可传递 0；
- to：存有目标地址信息的 sockaddr 结构体变量的地址；
- addrlen：传递给参数 to 的地址值结构体变量的长度。

UDP 发送函数 sendto() 与 TCP 发送函数 write()/send() 的最大区别在于，sendto() 函数需要向他传递目标地址信息。

接收数据使用 recvfrom() 函数：

```
1.  ssize_t recvfrom(int sock, void *buf, size_t nbytes, int flags, struct sockadr *from, socklen_t *addrlen);  //Linux
2.  int recvfrom(SOCKET sock, char *buf, int nbytes, int flags, const struct sockaddr *from, int *addrlen);  //Windows
```

由于 UDP 数据的发送端不定，所以 recvfrom() 函数定义为可接收发送端信息的形式，具体参数如下：

- sock：用于接收 UDP 数据的套接字；
- buf：保存接收数据的缓冲区地址；
- nbytes：可接收的最大字节数（不能超过 buf 缓冲区的大小）；

- flags：可选项参数，若没有可传递 0；
- from：存有发送端地址信息的 sockaddr 结构体变量的地址；
- addrlen：保存参数 from 的结构体变量长度的变量地址值。

基于 UDP 的回声服务器端/客户端

下面结合之前的内容实现回声客户端。需要注意的是，UDP 不同于 TCP，不存在请求连接和受理过程，因此在某种意义上无法明确区分服务器端和客户端，只是因为其提供服务而称为服务器端，希望各位读者不要误解。

下面给出 Windows 下的代码，Linux 与此类似，不再赘述。

服务器端 server.cpp：

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加载 ws2_32.dll

#define BUF_SIZE 100

int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    //创建套接字
    SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);

    //绑定套接字
    sockaddr_in servAddr;
    memset(&servAddr, 0, sizeof(servAddr));  //每个字节都用 0 填充
    servAddr.sin_family = PF_INET;  //使用 IPv4 地址
    servAddr.sin_addr.s_addr = htonl(INADDR_ANY); //自动获取 IP 地址
    servAddr.sin_port = htons(1234);  //端口
    bind(sock, (SOCKADDR*)&servAddr, sizeof(SOCKADDR));

    //接收客户端请求
    SOCKADDR clntAddr;  //客户端地址信息
    int nSize = sizeof(SOCKADDR);
    char buffer[BUF_SIZE];  //缓冲区
    while(1){
```

```
27.         int strLen = recvfrom(sock, buffer, BUF_SIZE, 0, &clntAddr, &nSize);
28.         sendto(sock, buffer, strLen, 0, &clntAddr, nSize);
29.     }
30.
31.     closesocket(sock);
32.     WSACleanup();
33.     return 0;
34. }
```

代码说明：
1) 第 12 行代码在创建套接字时，向 socket() 第二个参数传递 SOCK_DGRAM，以指明使用 UDP 协议。

2) 第 18 行代码中使用 `htonl(INADDR_ANY)` 来自动获取 IP 地址。

利用常数 INADDR_ANY 自动获取 IP 地址有一个明显的好处，就是当软件安装到其他服务器或者服务器 IP 地址改变时，不用再更改源码重新编译，也不用在启动软件时手动输入。而且，如果一台计算机中已分配多个 IP 地址（例如路由器），那么只要端口号一致，就可以从不同的 IP 地址接收数据。所以，服务器中优先考虑使用 INADDR_ANY；而客户端中除非带有一部分服务器功能，否则不会采用。

客户端 client.cpp：

```
1.  #include <stdio.h>
2.  #include <WinSock2.h>
3.  #pragma comment(lib, "ws2_32.lib")  //加载 ws2_32.dll
4.
5.  #define BUF_SIZE 100
6.
7.  int main(){
8.      //初始化 DLL
9.      WSADATA wsaData;
10.     WSAStartup(MAKEWORD(2, 2), &wsaData);
11.
12.     //创建套接字
13.     SOCKET sock = socket(PF_INET, SOCK_DGRAM, 0);
14.
15.     //服务器地址信息
16.     sockaddr_in servAddr;
17.     memset(&servAddr, 0, sizeof(servAddr));  //每个字节都用 0 填充
18.     servAddr.sin_family = PF_INET;
19.     servAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

```
    servAddr.sin_port = htons(1234);

    //不断获取用户输入并发送给服务器，然后接受服务器数据
    sockaddr fromAddr;
    int addrLen = sizeof(fromAddr);
    while(1){
        char buffer[BUF_SIZE] = {0};
        printf("Input a string: ");
        gets(buffer);
        sendto(sock, buffer, strlen(buffer), 0, (struct sockaddr*)&servAddr, sizeof(servAddr));
        int strLen = recvfrom(sock, buffer, BUF_SIZE, 0, &fromAddr, &addrLen);
        buffer[strLen] = 0;
        printf("Message form server: %s\n", buffer);
    }

    closesocket(sock);
    WSACleanup();
    return 0;
}
```

先运行 server，再运行 client，client 输出结果为：

```
Input a string: C 语言中文网
Message form server: C 语言中文网
Input a string: c.biancheng.net Founded in 2012
Message form server: c.biancheng.net Founded in 2012
Input a string:
```

从代码中可以看出，server.cpp 中没有使用 listen() 函数，client.cpp 中也没有使用 connect() 函数，因为 UDP 不需要连接。