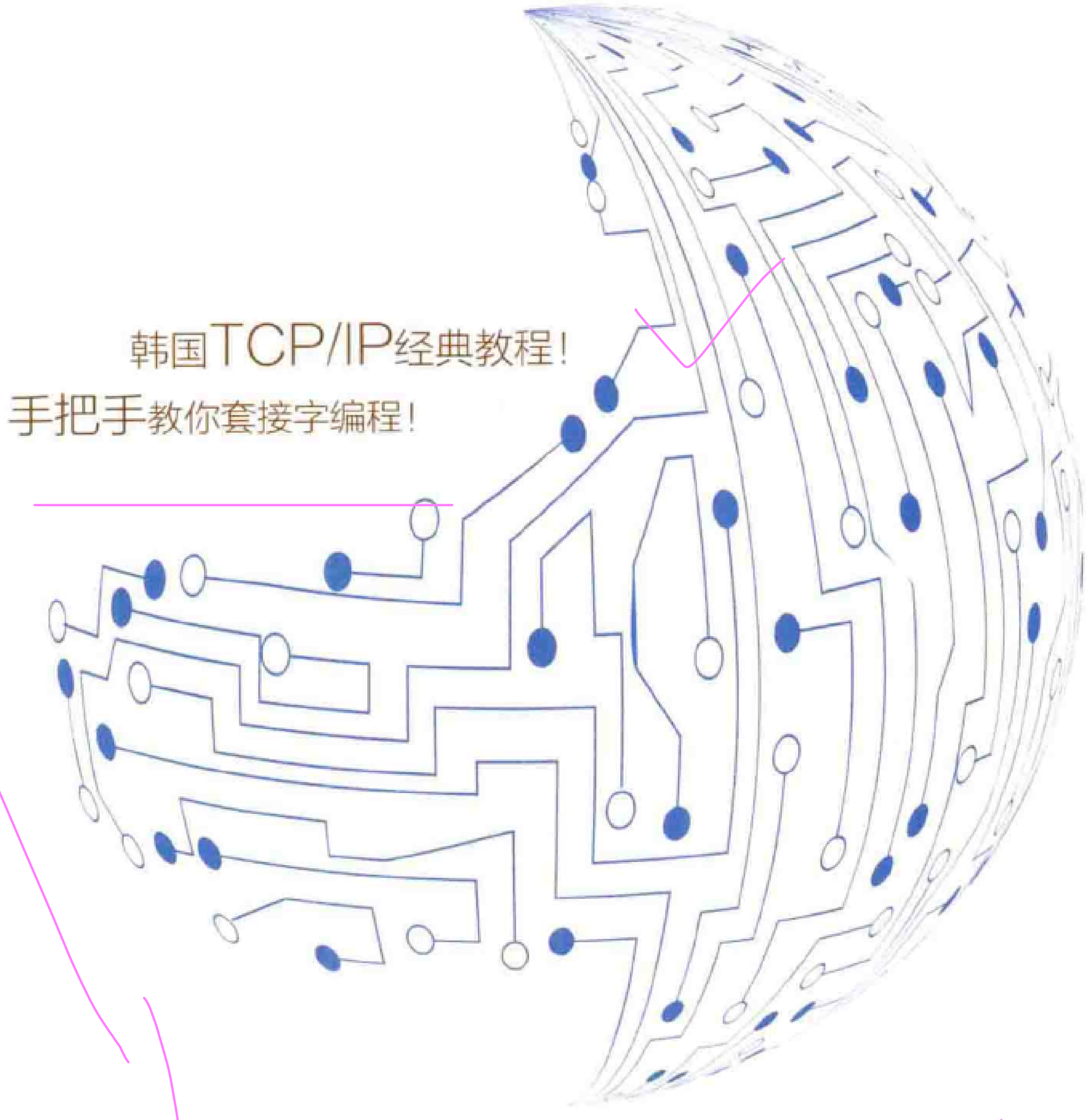
TURING 图灵程序设计丛书

TCP/IP网络编程

【韩】尹圣雨 著　　金国哲 译



韩国TCP/IP经典教程！

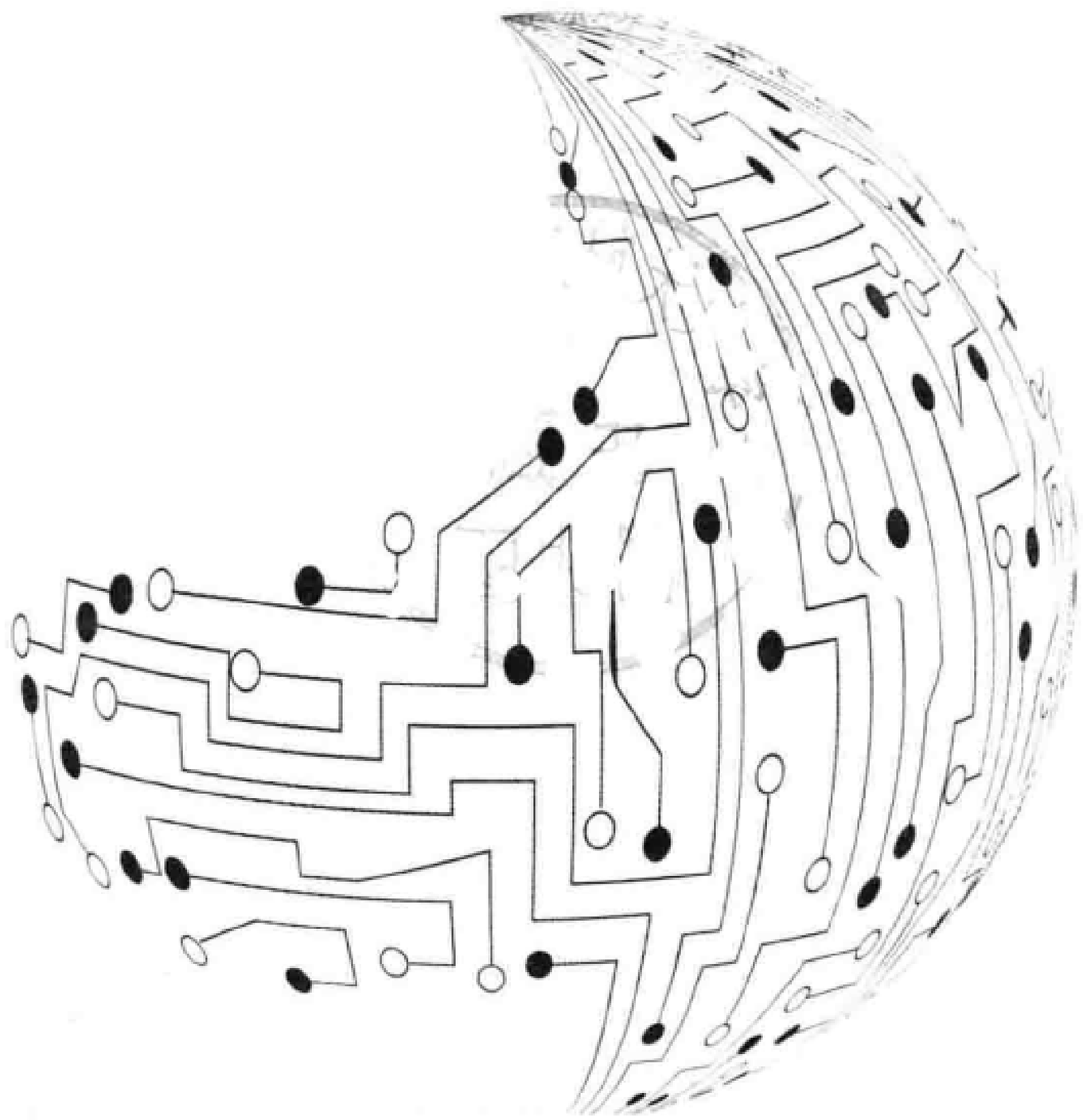
手把手教你套接字编程！

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

TCP/IP网络编程

【韩】尹圣雨 著　　金国哲 译



人民邮电出版社
北京

图书在版编目（CIP）数据

TCP/IP网络编程 / （韩）尹圣雨著 ；金国哲译. -- 北京 ：人民邮电出版社，2014.7
（图灵程序设计丛书）
ISBN 978-7-115-35885-1

Ⅰ. ①T… Ⅱ. ①尹… ②金… Ⅲ. ①计算机网络－通信协议②计算机网络－程序设计 Ⅳ. ①TN915.04 ②TP393.09

中国版本图书馆CIP数据核字(2014)第117505号

内 容 提 要

本书涵盖操作系统、系统编程、TCP/IP 协议等多种内容，结构清晰、讲解细致、通俗易懂。书中收录丰富示例，详细展现了 Linux 和 Windows 平台下套接字编程的共性与个性。特别是从代码角度说明了不同模型服务器端的区别，还包括了条件触发与边缘触发等知识，对开发实践也有很大帮助。

本书针对网络编程初学者，面向具备 C 语言基础的套接字网络编程学习者，适合所有希望学习 Linux 和 Windows 网络编程的人。

◆ 著　　[韩] 尹圣雨
译　　金国哲
责任编辑　傅志红
执行编辑　陈　曦
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：26.25
字数：620千字　　2014年 7 月第 1 版
印数：1－4 000册　　2014年 7 月北京第 1 次印刷
著作权合同登记号　图字：01-2013-8539 号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

版 权 声 明

作 者 序

为初学者准备的网络编程

我曾有一段时间痴迷于学习网络编程，那时关注的重点是网络技术，也因此走上了网络编程之路。现在回想起来也没有什么特别的理由，只是因为我个人认为网络编程是程序员的基本功。当时学完 C 和 C++后，我购买了外国知名作者撰写的网络编程书。虽然是英文书，而且内容较多，但我对自己的网络技术和编程技术相当自信，选书的时候毫不犹豫。但不到一周就实在看不下去了，并不是因为书的质量没有想象的那么好，或者有英文障碍，主要是因为自己连书中示例都无法正常调试通过。

之后，我在大学研究室和公司接触了大量开发人员，逐渐对各个领域有了更深入的认识，也因此产生了重拾书本的勇气。再去读的时候发现原书写得的确非常棒。

我并不是特别聪明或理解力特别强的人，所以花费大量时间学习了属于程序员必修课的操作系统和算法。对我而言，学习知名的计算机理论原著是不小的负担。当时的我最需要的是通俗易懂的书，并不是笼统的叙述，而是详细的说明，同时符合我的水平。

如果各位与我当年的水平一样，那本书正是为大家准备的。对于已经掌握大量网络编程相关知识并希望得到提升的读者而言，本书可能过于简单。而第一次接触网络编程的读者，或者在学习过程中像我一样受过挫折的读者，都能通过本书获得很大帮助。

我在书中也尝试探讨了更多深层问题，但同时又担心读者对此产生抵触情绪。感谢那些选择本书并给予好评的读者们！

借此机会，我要感谢韩浩、智秀、胜熙、朱英及其学生帮我修改病句和错别字。另外，向智敏（不允许我在家工作而只能休息）、智律（对不起，没能抱着你陪你玩）和他们的“队长”燕淑表示深深的歉意。

最后，感谢敬爱的母亲，您一直为深夜还在写书的我而操心。感谢宋盛根组长、李升振组长，你们让我懂得写书并非一人之力。感谢帮助我完善本书的编辑们。感谢对本书提出宝贵意见的同事们，以及鼓励并祝福我的所有朋友们。

尹圣雨

前　言

本书适合人群

本书面向基于套接字的网络编程学习者，所以不需要太多基础知识。但示例使用 C 语言编写而成，因此需要这方面的理解。我相信各位已经具备一定的 C 语言基础。

掌握网络相关基础知识将有助于学习，但并不绝对。本书针对的是网络编程的初学者，因此，书中首先强调的是所有示例的可读性，之后才考虑代码优化问题。

本书适合所有希望学习 Linux 和 Windows 网络编程的人士，所以无论使用哪种操作系统都不会有问题。但我想说，网络编程的特点决定了同时学习两种操作系统平台的网络编程是最有效的学习方法。如果说在一种操作系统下学习网络编程需要十分努力，那么同时学习两种平台仅需十二分功夫，可谓事半功倍。

没有必要为学习本书而特意掌握 Linux 和 Windows 的所有操作方法，只需了解编译方法即可，书中详细讲解了 Linux 平台下的编译方法。大部分示例都在 Linux 和 Windows 平台下实现，很容易找到网络编程中不同操作系统的差异。

本书结构

本书共分 4 个部分，各部分的内容如下。

第一部分主要介绍网络编程基础知识。此部分主要由 Windows 和 Linux 平台网络编程必备基础知识构成，不会过多涉及操作系统特性相关内容。第一部分并非第二部分和第三部分的简化版，而是介绍了两种操作系统的共性。

第一部分的特点决定了本书的叙述方式。如果根据不同操作系统分别展开叙述，则会产生大量重复内容。因此，本书围绕一个操作系统进行讲解，然后指出系统间差异。选择哪一种操作系统也成为困扰我的一个问题，刚开始考虑使用相对流行的 Windows，但最终选了 Linux。Windows 套接字是以 UNIX 系列的 BSD 套接字模型为基础设计而成的，所以我认为先介绍 Linux 平台下的套接字更有助于理解。这个决定也反映出不少程序员的想法，相信同样有助于各位学习。其实基于哪种操作系统展开叙述对第一部分的影响并不大，关于这一点，各位在学习过程中会有切身感悟。

第二部分和第三部分与操作系统有关。不同操作系统提供的系统函数不同，支持的功能也有

差异，因此，有些内容必须分开讨论。第二部分主要是Linux相关内容，而第三部分主要是Windows相关内容。希望从事 Windows 编程的朋友也浏览一下第二部分的内容，即使在 Windows 平台下编程，这部分内容同样会帮助您提高技艺。

第四部分是收尾阶段，各位可以把这部分内容视为对之前学习的总结。其中包含了我作为网络编程先行者的学习建议，希望大家以轻松的心态阅读。

目　录

第四部分　结束网络编程

Part 01

开始网络编程

第1章 理解网络编程和套接字

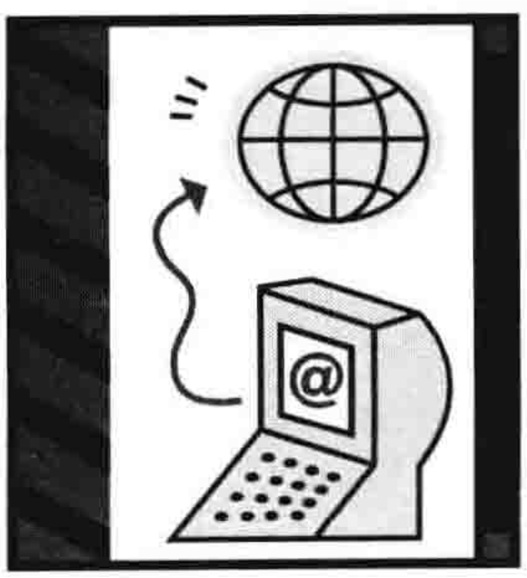
网络编程领域需要一定的操作系统和系统编程知识，同时还需要理解好 TCP/IP 网络数据传输协议。这么说来，网络编程的确需要一定的基础知识，但相比于其他领域，它更有趣，而且没想象中那么难。只要踏踏实实学习，任何人都可以轻松进入网络编程的世界。

深入细节前，本章先帮助各位建立对本书的总体认识，并简要了解后面的内容。希望通过本章的学习，大家能对网络编程有初步了解，摆脱对它的畏惧。

1.1 理解网络编程和套接字

学习C语言时，一般会先学利用printf函数和scanf函数进行控制台输入输出，然后学习文件输入输出。如果各位认真学习过C语言就会发现，控制台输入输出和文件输入输出非常类似。实际上，网络编程也与文件输入输出有很多相似之处，相信大家也能轻松掌握。

网络编程和套接字概要



网络编程就是编写程序使两台连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连接到庞大的互联网，因此不用担心这点。在此基础上，只需考虑如何编写数据传输软件。但实际上这也不用愁，因为操作系统会提供名为“套接字”（socket）的部件。套接字是网络数据传输用的软件设备。即使对网络数据传输原理不太熟悉，我们也能通过套接字完成数据传输。因此，网络编程又称为套接字编程。那为什么要用“套接字”这个词呢？

我们把插头插到插座上就能从电网获得电力供给，同样，为了与远程计算机进行数据传输，需要连接到因特网，而编程中的“套接字”就是用来连接该网络的工具。它本身就带有“连接”的含义，如果将其引申，则还可以表示两台计算机之间的网络连接。

构建接电话套接字

套接字大致分为两种，其中，先要讨论的TCP套接字可以比喻成电话机。实际上，电话机也是通过固定电话网（telephone network）完成语音数据交换的。因此，我们熟悉的固定电话与套接字实际并无太大区别。下面利用电话机讲解套接字的创建及使用方法。

电话机可以同时用来拨打或接听，但对套接字而言，拨打和接听是有区别的。我们先讨论用于接听的套接字创建过程。

调用socket函数（安装电话机）时进行的对话

- 问：“接电话需要准备什么？”
- 答：“当然是电话机!”

有了电话机才能安装电话，接下来，我们就准备一部漂亮的电话机。下列函数创建的就是相当于电话机的套接字。

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

    ➔ 成功时返回文件描述符，失败时返回-1。
```

上述函数及本章涉及的其他函数的详细说明将在以后章节中逐一给出，现在只需掌握“原来是由socket函数生成套接字的”就足够了。另外，我们只需购买机器，剩下的安装和分配电话号码等工作都由电信局的工作人员完成。而套接字需要我们自己安装，这也是套接字编程难点所在，但多安装几次就会发现其实不难。准备好电话机后要考虑分配电话号码的问题，这样别人才能联系到自己。

调用bind函数（分配电话号码）时进行的对话

- 问：“请问您的电话号码是多少？”
- 答：“我的电话号码是123-1234。”

套接字同样如此。就像给电话机分配电话号码一样（虽然不是真的把电话号码给了电话机），利用以下函数给创建好的套接字分配地址信息（IP地址和端口号）。

```
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *myaddr, socklen_t addrlen);
```

➔ 成功时返回0，失败时返回-1。

调用bind函数给套接字分配地址后，就基本完成了接电话的所有准备工作。接下来需要连接电话线并等待来电。

调用listen函数（连接电话线）时进行的对话

❑ 问："已架设完电话机后是否只需连接电话线？"

❑ 答："对，只需连接就能接听电话。"

一连接电话线，电话机就转为可接听状态，这时其他人可以拨打电话请求连接到该机。同样，需要把套接字转化成可接收连接的状态。

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

➔ 成功时返回0，失败时返回-1。

连接好电话线后，如果有人拨打电话就会响铃，拿起话筒才能接听电话。

调用accept函数（拿起话筒）时进行的对话

❑ 问："电话铃响了，我该怎么办？"

❑ 答："难道您真不知道？接听啊！"

拿起话筒意味着接收了对方的连接请求。套接字同样如此，如果有人为了完成数据传输而请求连接，就需要调用以下函数进行受理。

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

➔ 成功时返回文件描述符，失败时返回-1。

网络编程中接受连接请求的套接字创建过程可整理如下。

❑ 第一步：调用socket函数创建套接字。

❑ 第二步：调用bind函数分配IP地址和端口号。

❑ 第三步：调用listen函数转为可接收请求状态。

❑ 第四步：调用accept函数受理连接请求。

记住并掌握这些步骤就相当于为套接字编程勾勒好了轮廓，后续章节会为此轮廓着色。

编写“Hello world!”服务器端

服务器端（server）是能够受理连接请求的程序。下面构建服务器端以验证之前提到的函数调用过程，该服务器端收到连接请求后向请求者返回“Hello world!”答复。除各种函数的调用顺序外，我们还未涉及任何实际编程。因此，阅读代码时请重点关注套接字相关函数的调用过程，不必理解全部示例。

❖ hello_server.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.  void error_handling(char *message);
8.
9.  int main(int argc, char *argv[])
10. {
11.     int serv_sock;
12.     int clnt_sock;
13.
14.     struct sockaddr_in serv_addr;
15.     struct sockaddr_in clnt_addr;
16.     socklen_t clnt_addr_size;
17.
18.     char message[]="Hello World!";
19.
20.     if(argc!=2)
21.     {
22.         printf("Usage : %s <port>\n", argv[0]);
23.         exit(1);
24.     }
25.
26.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
27.     if(serv_sock == -1)
28.         error_handling("socket() error");
29.
30.     memset(&serv_addr, 0, sizeof(serv_addr));
31.     serv_addr.sin_family=AF_INET;
32.     serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
33.     serv_addr.sin_port=htons(atoi(argv[1]));
34.
35.     if(bind(serv_sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr))==-1)
36.         error_handling("bind() error");
37.
38.     if(listen(serv_sock, 5)==-1)
39.         error_handling("listen() error");
```

```
40.
41.     clnt_addr_size=sizeof(clnt_addr);
42.     clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
43.     if(clnt_sock==-1)
44.         error_handling("accept() error");
45.
46.     write(clnt_sock, message, sizeof(message));
47.     close(clnt_sock);
48.     close(serv_sock);
49.     return 0;
50. }
51.
52. void error_handling(char *message)
53. {
54.     fputs(message, stderr);
55.     fputc('\n', stderr);
56.     exit(1);
57. }
```

代码说明

- 第26行：调用socket函数创建套接字。
- 第35行：调用bind函数分配IP地址和端口号。
- 第38行：调用listen函数将套接字转为可接收连接状态。
- 第42行：调用accept函数受理连接请求。如果在没有连接请求的情况下调用该函数，则不会返回，直到有连接请求为止。
- 第46行：稍后将要介绍的write函数用于传输数据，若程序经过第42行代码执行到本行，则说明已经有了连接请求。

编译并运行以上示例，创建等待连接请求的服务器端。目前不必详细分析源代码，只需确认之前4个函数调用过程。稍后将讲解上述示例中调用的write函数。下面讨论如何编写向服务器端发送连接请求的客户端。

构建打电话套接字

服务器端创建的套接字又称为服务器端套接字或监听（listening）套接字。接下来介绍的套接字是用于请求连接的客户端套接字。客户端套接字的创建过程比创建服务器端套接字简单，因此直接进行讲解。

还未介绍打电话（请求连接）的函数，因为其调用的是客户端套接字，如下所示。

```
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen);
```

➔ 成功时返回0，失败时返回-1。

客户端程序只有“调用socket函数创建套接字”和“调用connect函数向服务器端发送连接请求”这两个步骤，因此比服务器端简单。下面给出客户端，查看以下两项内容：第一，调用socket函数和connect函数；第二，与服务器端共同运行以收发字符串数据。

❖ hello_client.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.  void error_handling(char *message);
8.
9.  int main(int argc, char* argv[])
10. {
11.     int sock;
12.     struct sockaddr_in serv_addr;
13.     char message[30];
14.     int str_len;
15.
16.     if(argc!=3)
17.     {
18.         printf("Usage : %s <IP> <port>\n", argv[0]);
19.         exit(1);
20.     }
21.
22.     sock=socket(PF_INET, SOCK_STREAM, 0);
23.     if(sock == -1)
24.         error_handling("socket() error");
25.
26.     memset(&serv_addr, 0, sizeof(serv_addr));
27.     serv_addr.sin_family=AF_INET;
28.     serv_addr.sin_addr.s_addr=inet_addr(argv[1]);
29.     serv_addr.sin_port=htons(atoi(argv[2]));
30.
31.     if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)
32.         error_handling("connect() error!");
33.
34.     str_len=read(sock, message, sizeof(message)-1);
35.     if(str_len==-1)
36.         error_handling("read() error!");
37.
38.     printf("Message from server : %s \n", message);
39.     close(sock);
40.     return 0;
41. }
42.
43. void error_handling(char *message)
44. {
45.     fputs(message, stderr);
46.     fputc('\n', stderr);
47.     exit(1);
48. }
```

- 第22行：创建套接字，但此时套接字并不马上分为服务器端和客户端。如果紧接着调用bind、listen函数，将成为服务器端套接字；如果调用connect函数，将成为客户端套接字。
- 第31行：调用connect函数向服务器端发送连接请求。

这样就编好了服务器端和客户端，相信各位会产生好多疑问（实际上不懂的内容比知道的更多）。接下来的几章将进行解答，请不要着急。

在 Linux 平台下运行

虽未另行说明，但上述两个示例应在Linux环境中编译并执行。接下来将简单介绍Linux下的C语言编译器——GCC（GNU Compiler Collection，GNU编译器集合）。下面是对hello_server.c示例进行编译的命令。

```
gcc hello_server.c -o hserver
```

➔ 编译 hello_server.c 文件并生成可执行文件 hserver。

该命令中的-o是用来指定可执行文件名的可选参数，因此，编译后将生成可执行文件hserver。可如下执行此项命令。

```
./hserver
```

➔ 运行当前目录下的 hserver 文件。

了解编译和运行相关知识的确多多益善，但学习本书只需掌握基本用法。接下来运行程序。服务器端需要在运行时接收客户端的连接请求，因此先运行服务器端。

❖ 运行结果：hello_server.c

```
root@my_linux:/tcpip# gcc hello_server.c -o hserver
root@my_linux:/tcpip# ./hserver 9190
```

正常情况下程序将停留在此状态，因为服务器端调用的accept函数还未返回。接下来运行客户端。

❖ 运行结果：hello_client.c

```
root@my_linux:/tcpip# gcc hello_client.c -o hclient
root@my_linux:/tcpip# ./hclient 127.0.0.1 9190
Message from server: Hello World!
root@my_linux:/tcpip#
```

由此查看客户端消息传输过程。同时发现，完成消息传输后，服务器端和客户端都停止运行。执行过程中输入的127.0.0.1是运行示例用的计算机（本地计算机）的IP地址。如果在同一台计算

机中同时运行服务器端和客户端，将采用这种连接方式。但如果服务器端与客户端在不同计算机中运行，则应采用服务器端所在计算机的IP地址。

提 示

再次运行程序前需等待

上面的服务器端无法立即重新运行。如果想再次运行，则需要更改之前输入的端口号 9190。后面会详细讲解其原因，现在不必对此感到意外。

1.2 基于 Linux 的文件操作

讨论套接字的过程中突然谈及文件也许有些奇怪。但对Linux而言，socket操作与文件操作没有区别，因而有必要详细了解文件。在Linux世界里，socket也被认为是文件的一种，因此在网络数据传输过程中自然可以使用文件I/O的相关函数。Windows则与Linux不同，是要区分socket和文件的。因此在Windows中需要调用特殊的数据传输相关函数。

底层文件访问（Low-Level File Access）和文件描述符（File Descriptor）

即使看到“底层”二字，也会有读者臆测其难以理解。实际上，“底层”这个表达可以理解为“与标准无关的操作系统独立提供的”。稍后讲解的函数是由Linux提供的，而非ANSI标准定义的函数。如果想使用Linux提供的文件I/O函数，首先应该理解好文件描述符的概念。

此处的文件描述符是系统分配给文件或套接字的整数。实际上，学习C语言过程中用过的标准输入输出及标准错误在Linux中也被分配表1-1中的文件描述符。

表1-1 分配给标准输入输出及标准错误的文件描述符

文件描述符	对 象
0	标准输入：Standard Input
1	标准输出：Standard Output
2	标准错误：Standard Error

文件和套接字一般经过创建过程才会被分配文件描述符。而表1-1中的3种输入输出对象即使未经过特殊的创建过程，程序开始运行后也会被自动分配文件描述符。稍后将详细讲解其使用方法及含义。

知识补给站 文件描述符（文件句柄）

学校附近有个服务站，只需打个电话就能复印所需论文。服务站有位常客叫英秀，他每次都要求复印同一篇论文的一部分内容。

“大叔您好！请帮我复印一下《关于随着高度信息化社会而逐步提升地位的触觉、知觉、思维、性格、智力等人类生活质量相关问题特性的人类学研究》这篇论文第26页到第30页。”

这位同学每天这样打好几次电话，更雪上加霜的是语速还特别慢。终于有一天大叔说：

“从现在开始，那篇论文就编为第18号！你就说帮我复印18号论文26页到30页！”

之后英秀也是只复印超过50字标题的论文，大叔也会给每篇论文分配无重复的新号（数字）。这才不会头疼于与英秀的对话，且不影响业务。

该示例中，大叔相当于操作系统，英秀相当于程序员，论文号相当于文件描述符，论文相当于文件或套接字。也就是说，每当生成文件或套接字，操作系统将返回分配给它们的整数。这个整数将成为程序员与操作系统之间良好沟通的渠道。实际上，文件描述符只不过是为了方便称呼操作系统创建的文件或套接字而赋予的数而已。

文件描述符有时也称为文件句柄，但“句柄”主要是Windows中的术语。因此，本书中如果涉及Windows平台将使用“句柄”，如果是Linux平台则用“描述符”。

打开文件

首先介绍打开文件以读写数据的函数。调用此函数时需传递两个参数：第一个参数是打开的目标文件名及路径信息，第二个参数是文件打开模式（文件特性信息）。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int flag);
```

➔ 成功时返回文件描述符，失败时返回-1。

- path　文件名的字符串地址。
- flag　文件打开模式信息。

表1-2是此函数第二个参数flag可能的常量值及含义。如需传递多个参数，则应通过位或运算（OR）符组合并传递。

表1-2 文件打开模式

打开模式	含 义
O_CREAT	必要时创建文件
O_TRUNC	删除全部现有数据
O_APPEND	维持现有数据，保存到其后面
O_RDONLY	只读打开
O_WRONLY	只写打开
O_RDWR	读写打开

稍后将给出此函数的使用示例。接下来先介绍关闭文件和写文件时调用的函数。

关闭文件

各位学习C语言时学过，使用文件后必须关闭。下面介绍关闭文件时调用的函数。

```
#include <unistd.h>

int close(int fd);

    ➔ 成功时返回 0，失败时返回-1。
```

- fd　　需要关闭的文件或套接字的文件描述符。

若调用此函数的同时传递文件描述符参数，则关闭（终止）相应文件。另外需要注意的是，此函数不仅可以关闭文件，还可以关闭套接字。这再次证明了“Linux操作系统 不区分文件与套接字”的特点。

将数据写入文件

接下来介绍的write函数用于向文件输出（传输）数据。当然，Linux中不区分文件与套接字，因此，通过套接字向其他计算机传递数据时也会用到该函数。之前的示例也调用它传递字符串“Hello World!”。

```
#include <unistd.h>

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

    ➔ 成功时返回写入的字节数，失败时返回-1。
```

- fd　　显示数据传输对象的文件描述符。
- buf　　保存要传输数据的缓冲地址值。
- nbytes　　要传输数据的字节数。

此函数定义中，size_t是通过typedef声明的unsigned int类型。对ssize_t来说，size_t前面多加的s代表signed，即ssize_t是通过typedef声明的signed int类型。

知识补给站　以_t为后缀的数据类型

我们已经接触到ssize_t、size_t等陌生的数据类型。这些都是元数据类型（primitive），在sys/types.h头文件中一般由typedef声明定义，算是给大家熟悉的基本数据类型起了别名。既然已经有了基本数据类型，为何还要声明并使用这些新的呢？

人们目前普遍认为int是32位的，因为主流操作系统和计算机仍采用32位。而在过去16位操作系统时代，int类型是16位的。根据系统的不同、时代的变化，数据类型的表现形式也随之改变，需要修改程序中使用的数据类型。如果之前已在需要声明4字节数据类型之处使用了size_t或ssize_t，则将大大减少代码变动，因为只需要修改并编译size_t和ssize_t的typedef声明即可。在项目中，为了给基本数据类型赋予别名，一般会添加大量typedef声明。而为了与程序员定义的新数据类型加以区分，操作系统定义的数据类型会添加后缀_t。

下面通过示例帮助大家更好地理解前面讨论过的函数。此程序将创建新文件并保存数据。

❖ low_open.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <fcntl.h>
4.  #include <unistd.h>
5.  void error_handling(char* message);
6.
7.  int main(void)
8.  {
9.      int fd;
10.     char buf[]="Let's go!\n";
11.
12.     fd=open("data.txt", O_CREAT|O_WRONLY|O_TRUNC);
13.     if(fd==-1)
14.         error_handling("open() error!");
15.     printf("file descriptor: %d \n", fd);
16.
17.     if(write(fd, buf, sizeof(buf))==-1)
18.         error_handling("write() error!");
19.     close(fd);
20.     return 0;
21. }
22.
23. void error_handling(char* message)
```

```
24. {
25.     //与之前示例相同，故省略!
26. }
```

代码说明

- 第12行：文件打开模式为O_CREAT、O_WRONLY和O_TRUNC的组合，因此将创建空文件，并只能写。若存在data.txt文件，则清空文件的全部数据。
- 第17行：向对应于fd中保存的文件描述符的文件传输buf中保存的数据。

❖ 运行结果：low_open.c

```
root@my_linux:/tcpip# gcc low_open.c -o lopen
root@my_linux:/tcpip# ./lopen
file descriptor: 3
root@my_linux:/tcpip# cat data.txt
Let's go!
root@my_linux:/tcpip#
```

运行示例后，利用Linux的cat命令输出data.txt文件内容，可以确认确实已向文件传输数据。

读取文件中的数据

与之前的write函数相对应，read函数用来输入（接收）数据。

```
#include <unistd.h>

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

➔ 成功时返回接收的字节数（但遇到文件结尾则返回 0），失败时返回-1。

- fd　显示数据接收对象的文件描述符。
- buf　要保存接收数据的缓冲地址值。
- nbytes　要接收数据的最大字节数。

下列示例将通过read函数读取data.txt中保存的数据。

❖ low_read.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <fcntl.h>
4.  #include <unistd.h>
5.  #define BUF_SIZE 100
6.  void error_handling(char* message);
7.
8.  int main(void)
```

```
9.  {
10.     int fd;
11.     char buf[BUF_SIZE];
12.
13.     fd=open("data.txt", O_RDONLY);
14.     if( fd==-1)
15.         error_handling("open() error!");
16.     printf("file descriptor: %d \n" , fd);
17.
18.     if(read(fd, buf, sizeof(buf))==-1)
19.         error_handling("read() error!");
20.     printf("file data: %s", buf);
21.     close(fd);
22.     return 0;
23. }
24.
25. void error_handling(char* message)
26. {
27.     //与之前示例相同，故省略!
28. }
```

代码说明

- 第13行：打开读取专用文件data.txt。
- 第18行：调用read函数向第11行中声明的数组buf保存读入的数据。

❖ 运行结果 low_read.c

```
root@my_linux:/tcpip# gcc low_read.c -o lread
root@my_linux:/tcpip# ./lread
file descriptor: 3
file data: Let's go!
root@my_linux:/tcpip#
```

基于文件描述符的I/O操作相关介绍到此结束。希望各位记住，该内容同样适用于套接字。

文件描述符与套接字

下面将同时创建文件和套接字，并用整数型态比较返回的文件描述符值。

❖ fd_seri.c

```
1.  #include <stdio.h>
2.  #include <fcntl.h>
3.  #include <unistd.h>
4.  #include <sys/socket.h>
5.
6.  int main(void)
7.  {
```

```
8.      int fd1, fd2, fd3;
9.      fd1=socket(PF_INET, SOCK_STREAM, 0);
10.     fd2=open("test.dat", O_CREAT|O_WRONLY|O_TRUNC);
11.     fd3=socket(PF_INET, SOCK_DGRAM, 0);
12.
13.     printf("file descriptor 1: %d\n", fd1);
14.     printf("file descriptor 2: %d\n", fd2);
15.     printf("file descriptor 3: %d\n", fd3);
16.
17.     close(fd1); close(fd2); close(fd3);
18.     return 0;
19. }
```

代码说明

- 第9～11行：创建1个文件和2个套接字。
- 第13～15行：输出之前创建的文件描述符的整数值。

❖ 运行结果：fd_seri.c

```
root@my_linux:/tcpip# gcc fd_seri.c -o fds
root@my_linux:/tcpip# ./fds
file descriptor 1: 3
file descriptor 2: 4
file descriptor 3: 5
root@my_linux:/tcpip#
```

从输出的文件描述符整数值可以看出，描述符从3开始以由小到大的顺序编号（numbering），因为0、1、2是分配给标准I/O的描述符（如表1-1所示）。

1.3 基于 Windows 平台的实现

Windows套接字（以下简称Winsock）大部分是参考BSD系列UNIX套接字设计的，所以很多地方都跟Linux套接字类似。因此，只需要更改Linux环境下编好的一部分网络程序内容，就能在Windows平台下运行。本书也会同时讲解Linux和Windows两大平台，这不会给大家增加负担，反而会减轻压力。

同时学习 Linux 和 Windows 的原因

大多数项目都在Linux系列的操作系统下开发服务器端，而多数客户端是在Windows平台下开发的。不仅如此，有时应用程序还需要在两个平台之间相互切换。因此，学习套接字编程的过程中，有必要兼顾Windows和Linux两大平台。另外，这两大平台下的套接字编程非常类似，如果把其中相似的部分放在一起讲解，将大大提高学习效率。这会不会增加学习负担？一点也不。只要理解好其中一个平台下的网络编程方法，就很容易通过分析差异掌握另一平台。

为 Windows 套接字编程设置头文件和库

为了在Winsock基础上开发网络程序，需要做如下准备。

- 导入头文件winsock2.h。
- 链接ws2_32.lib库。

首先介绍项目中链接ws2_32.lib库的方法。我用的环境是Visual Studio 2008版本，接下来的讲解同样适用于更高版本的开发环境，不必因版本不同而感到困惑。打开项目的“属性”页，选择“配置属性”→“输入”→“附加依赖项”，如图1-1所示。当然，也可以通过快捷键Alt+F7打开“属性页”。

图1-1　项目“属性”页

接下来需要在图1-1的“附加依赖项”右边空白处直接写入ws2_32.lib。也可以通过点击空白处右边的按钮弹出如图1-2所示的对话框，并填写库名。

图1-2　链接库

设置库的工作到此结束。现在只需要在源文件中添加头文件，即可调用Winsock相关函数。

提 示

附加依赖项窗口位置可能不同

根据 VC++版本号的不同，图 1-2 中的附加依赖项窗口位置可能不同。一般可通过以下两种路径找到附加依赖项。

- 快捷键 Alt+F7→“配置属性”→“输入”→“附加依赖项”
- 快捷键 Alt+F7→“配置属性”→“链接器”→“输入”→“附加依赖项”

Winsock 的初始化

进行Winsock编程时，首先必须调用WSAStartup函数，设置程序中用到的Winsock版本，并初始化相应版本的库。

```
#include <winsock2.h>

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

➔ 成功时返回 0，失败时返回非零的错误代码值。

- wVersionRequested　程序员要用的Winsock版本信息。
- lpWSAData　WSADATA结构体变量的地址值。

有必要给出上述两个参数的详细说明。先说第一个，Winsock中存在多个版本，应准备WORD类型的（WORD是通过typedef声明定义的unsigned short类型）套接字版本信息，并传递给该函数的第一个参数wVersionRequested。若版本为1.2，则其中1是主版本号，2是副版本号，应传递0x0201。

如前所述，高8位为副版本号，低8位为主版本号，以此进行传递。本书主要使用2.2版本，故应传递0x0202。不过，以字节为单位手动构造版本信息有些麻烦，借助MAKEWORD宏函数则能轻松构建WORD型版本信息。

- MAKEWORD(1, 2);：//主版本为1，副版本为2，返回0x0201。
- MAKEWORD(2, 2);：//主版本为2，副版本为2，返回0x0202。

接下来讲解第二个参数lpWSADATA，此参数中需传入WSADATA型结构体变量地址（LPWSADATA是WSADATA的指针类型）。调用完函数后，相应参数中将填充已初始化的库信息。虽无特殊含义，但为了调用函数，必须传递WSADATA结构体变量地址。下面给出WSAStartup函数调用过程，这段代码几乎已成为Winsock编程的公式。

```
int main(int argc, char* argv[])
{
    WSADATA wsaData;
    . . . .
    if(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
        ErrorHandling("WSAStartup() error!");
    . . . .
    return 0;
}
```

前面已经介绍了Winsock相关库的初始化方法，接下来讲解如何注销该库——利用下面给出的函数。

```
#include <winsock2.h>

int WSACleanup(void);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

调用该函数时，Winsock相关库将归还Windows操作系统，无法再调用Winsock相关函数。从原则上讲，无需再使用Winsock函数时才调用该函数，但通常都在程序结束之前调用。

1.4　基于 Windows 的套接字相关函数及示例

本节介绍的Winsock函数与之前的Linux套接字相关函数相对应。既然只是介绍，就不做详细说明了，目的只在于让各位体会基于Linux和Windows的套接字函数之间的相似性。

基于 Windows 的套接字相关函数

首先介绍的函数与Linux下的socket函数提供相同功能。稍后讲解返回值类型SOCKET。

```
#include <winsock2.h>

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

➔ 成功时返回套接字句柄，失败时返回 INVALID_SOCKET。

下列函数与Linux的bind函数相同，调用其分配IP地址和端口号。

```
#include <winsock2.h>

int bind(SOCKET s, const struct sockaddr * name, int namelen);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

下列函数与Linux的listen函数相同，调用其使套接字可接收客户端连接。

```
#include <winsock2.h>

int listen(SOCKET s,int backlog);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

下列函数与Linux的accept函数相同，调用其受理客户端连接请求。

```
#include <winsock2.h>

SOCKET accept(SOCKET s, struct sockaddr * addr, int * addrlen);
```

➔ 成功时返回套接字句柄，失败时返回 INVALID_SOCKET。

下列函数与Linux的connect函数相同，调用其从客户端发送连接请求。

```
#include <winsock2.h>

int connect(SOCKET s, const struct sockaddr * name, int namelen);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

最后这个函数在关闭套接字时调用。Linux中，关闭文件和套接字时都会调用close函数；而Windows中有专门用来关闭套接字的函数。

```
#include <winsock2.h>

int closesocket(SOCKET s);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

以上就是基于Windows的套接字相关函数，虽然返回值和参数与Linux函数有所区别，但具有相同功能的函数名是一样的。正是这些特点使跨越两大操作系统平台的网络编程更加简单。

Windows 中的文件句柄和套接字句柄

Linux内部也将套接字当作文件，因此，不管创建文件还是套接字都返回文件描述符。之前也通过示例介绍了文件描述符返回及编号的过程。Windows中通过调用系统函数创建文件时，返回“句柄”（handle），换言之，Windows中的句柄相当于Linux中的文件描述符。只不过Windows中要区分文件句柄和套接字句柄。虽然都称为“句柄”，但不像Linux那样完全一致。文件句柄相关函数与套接字句柄相关函数是有区别的，这一点不同于Linux文件描述符。

既然对句柄有了一定理解，接下来再观察基于Windows的套接字相关函数，这将加深各位对SOCKET类型的参数和返回值的理解。的确！这就是为了保存套接字句柄整型值的新数据类型，它由typedef声明定义。回顾socket、listen和accept等套接字相关函数，则更能体会到与Linux中套接字相关函数的相似性。

有些程序员可能会问：“既然Winsock是以UNIX、Linux系列的BSD套接字为原型设计的，为什么不照搬过来，而是存在一定差异呢？”有人认为这是微软为了防止UNIX、Linux服务器端直接移植到Windows而故意为之。从网络程序移植性角度上看，这也是可以理解的。但我有不同意见。从本质上说，两种操作系统内核结构上存在巨大差异，而依赖于操作系统的代码实现风格也不尽相同，连Windows程序员给变量命名的方式也不同于Linux程序员。从各方面考虑，保持这种差异性就显得比较自然。因此我个人认为，Windows套接字与BSD系列的套接字编程方式有所不同是为了保持这种自然差异性。

创建基于 Windows 的服务器端和客户端

接下来将之前基于Linux的服务器端与客户端示例转化到Windows平台。目前想完全理解这些代码有些困难，我们只需验证套接字相关函数的调用过程、套接字库的初始化与注销过程即可。先介绍服务器端示例。

❖ hello_server_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <winsock2.h>
4.  void ErrorHandling(char* message);
5.
6.  int main(int argc, char* argv[])
7.  {
8.      WSADATA wsaData;
9.      SOCKET hServSock, hClntSock;
10.     SOCKADDR_IN servAddr, clntAddr;
```

```
11.
12.     int szClntAddr;
13.     char message[]="Hello World!";
14.     if(argc!=2)
15.     {
16.         printf("Usage : %s <port>\n", argv[0]);
17.         exit(1);
18.     }
19.
20.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
21.         ErrorHandling("WSAStartup() error!");
22.
23.     hServSock=socket(PF_INET, SOCK_STREAM, 0);
24.     if(hServSock==INVALID_SOCKET)
25.         ErrorHandling("socket() error");
26.
27.     memset(&servAddr, 0, sizeof(servAddr));
28.     servAddr.sin_family=AF_INET;
29.     servAddr.sin_addr.s_addr=htonl(INADDR_ANY);
30.     servAddr.sin_port=htons(atoi(argv[1]));
31.
32.     if(bind(hServSock, (SOCKADDR*) &servAddr, sizeof(servAddr))==SOCKET_ERROR)
33.         ErrorHandling("bind() error");
34.
35.     if(listen(hServSock, 5)==SOCKET_ERROR)
36.         ErrorHandling("listen() error");
37.
38.     szClntAddr=sizeof(clntAddr);
39.     hClntSock=accept(hServSock, (SOCKADDR*)&clntAddr,&szClntAddr);
40.     if(hClntSock==INVALID_SOCKET)
41.         ErrorHandling("accept() error");
42.
43.     send(hClntSock, message, sizeof(message), 0);
44.     closesocket(hClntSock);
45.     closesocket(hServSock);
46.     WSACleanup();
47.     return 0;
48. }
49.
50. void ErrorHandling(char* message)
51. {
52.     fputs(message, stderr);
53.     fputc('\n', stderr);
54.     exit(1);
55. }
```

代码说明

- 第20行：初始化套接字库。
- 第23、32行：第23行创建套接字，第32行给该套接字分配IP地址与端口号。
- 第35行：调用listen函数使第23行创建的套接字成为服务器端套接字。
- 第39行：调用accept函数受理客户端连接请求。
- 第43行：调用send函数向第39行连接的客户端传输数据。稍后讲解send函数。
- 第46行：程序终止前注销第20行中初始化的套接字库。

可以看出，除了Winsock库的初始化和注销相关代码、数据类型信息外，其余部分与Linux环境下的示例并无区别。希望各位阅读这部分代码时与之前的Linux服务器端进行逐行比较。接下来介绍与此示例同步的客户端代码。

❖ hello_client_win.c

```
#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>
void ErrorHandling(char* message);

int main(int argc, char* argv[])
{
    WSADATA wsaData;
    SOCKET hSocket;
    SOCKADDR_IN servAddr;

    char message[30];
    int strLen;
    if(argc!=3)
    {
        printf("Usage : %s <IP> <port>\n", argv[0]);
        exit(1);
    }

    if(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
        ErrorHandling("WSAStartup() error!");

    hSocket=socket(PF_INET, SOCK_STREAM, 0);
    if(hSocket==INVALID_SOCKET)
        ErrorHandling("socket() error");

    memset(&servAddr, 0, sizeof(servAddr));
    servAddr.sin_family=AF_INET;
    servAddr.sin_addr.s_addr=inet_addr(argv[1]);
    servAddr.sin_port=htons(atoi(argv[2]));

    if(connect(hSocket, (SOCKADDR*)&servAddr, sizeof(servAddr))==SOCKET_ERROR)
        ErrorHandling("connect() error!");

    strLen=recv(hSocket, message, sizeof(message)-1, 0);
    if(strLen==-1)
        ErrorHandling("read() error!");
    printf("Message from server: %s \n", message);

    closesocket(hSocket);
    WSACleanup();
    return 0;
}

void ErrorHandling(char* message)
{
```

```
47.     fputs(message, stderr);
48.     fputc('\n', stderr);
49.     exit(1);
50. }
```

代码说明

- 第20行：初始化Winsock库。
- 第23、32行：第23行创建套接字，第32行通过此套接字向服务器端发出连接请求。
- 第35行：调用recv函数接收服务器发来的数据。稍后讲解该函数。
- 第41行：注销第20行中初始化的Winsock库。

下面运行以上示例。创建编译项目的过程与各位学习C语言时使用的方法相同，只是增加了设置ws2_32.lib链接库的过程。

❖ 运行结果：hello_server_win.c

```
C:\tcpip>hServerWin 9190
```

运行过程中，假设可执行文件名为hServerWin.exe。如果运行正常，则与Linux相同，程序进入等待状态。这是因为服务器端调用了accept函数。接着运行客户端，假设客户端的可执行文件名为hClientWin.exe。

❖ 运行结果：hello_client_win.c

```
C:\tcpip>hClientWin 127.0.0.1 9190
Message from server: Hello World!
```

基于 Windows 的 I/O 函数

Linux中套接字也是文件，因而可以通过文件I/O函数read和write进行数据传输。而Windows中则有些不同。Windows严格区分文件I/O函数和套接字I/O函数。下面介绍Winsock数据传输函数。

```
#include <winsock2.h>

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

➔ 成功时返回传输字节数，失败时返回 SOCKET_ERROR。

- s　　表示数据传输对象连接的套接字句柄值。
- buf　　保存待传输数据的缓冲地址值。
- len　　要传输的字节数。
- flags　　传输数据时用到的多种选项信息。

此函数与Linux的write函数相比，只是多出了最后的flags参数。后续章节中将给出该参数的详细说明，在此之前只需传递0，表示不设置任何选项。但有一点需要注意，send函数并非Windows独有。Linux中也有同样的函数，它也来自于BSD套接字。只不过我在Linux相关示例中暂时只使用read、write函数，为了强调Linux环境下文件I/O和套接字I/O相同。下面介绍与send函数对应的recv函数。

```
#include <winsock2.h>

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

➔ 成功时返回接收的字节数（收到 EOF 时为 0），失败时返回 SOCKET_ERROR。

- s　表示数据接收对象连接的套接字句柄值。
- buf　保存接收数据的缓冲地址值。
- len　能够接收的最大字节数。
- flags　接收数据时用到的多种选项信息。

我只是在Windows环境下提前介绍了send、recv函数，以后的Linux示例中也会涉及。请不要误认为Linux中的read、write函数就是对应于Windows的send、recv函数。另外，之前的程序代码中也给出了send、recv函数调用过程，故不再另外给出相关示例。

知识补给站　Windows? Linux?

过去要编写服务器端的话，大部分程序员都会想起Linux或UNIX，因为那时的Windows还被认为是只能给个人使用的操作系统。即使Windows已经开始提供运营服务器端所需环境，但绝大多数网络程序员都会选择Linux和UNIX。不过，随着多媒体数据传输要求的提高，程序员的想法也有了变化。他们会根据服务器端的特点和环境选择不同的操作系统。如果各位想成为网络编程专家，就必须具备跨平台编程能力。

1.5　习题

(1) 套接字在网络编程中的作用是什么？为何称它为套接字？

(2) 在服务器端创建套接字后，会依次调用listen函数和accept函数。请比较并说明二者作用。

(3) Linux中，对套接字数据进行I/O时可以直接使用文件I/O相关函数；而在Windows中则不可以。原因为何？

(4) 创建套接字后一般会给它分配地址，为什么？为了完成地址分配需要调用哪个函数？

(5) Linux中的文件描述符与Windows的句柄实际上非常类似。请以套接字为对象说明它们的含义。

(6) 底层文件I/O函数与ANSI标准定义的文件I/O函数之间有何区别?

(7) 参考本书给出的示例low_open.c和low_read.c，分别利用底层文件I/O和ANSI标准I/O编写文件复制程序。可任意指定复制程序的使用方法。

第2章

套接字类型与协议设置

因为涉及套接字编程的基本内容，所以第2章和第3章显得相对枯燥一些。但本章内容是第4章介绍的实际网络编程的基础，希望各位反复精读。

大家已经对套接字的概念有所理解，本章将介绍套接字创建方法及不同套接字的特性。在本章仅需了解创建套接字时调用的socket函数，所以希望大家以放松的心态开始学习。

2.1 套接字协议及其数据传输特性

“协议”这个词给人的第一印象总是相当困难，我在学生时代也这么想。但各位要慢慢熟悉“协议”，因为它几乎是网络编程的全部内容。首先解释其定义。

关于协议（Protocol）

如果相隔很远的两人想展开对话，必须先决定对话方式。如果一方使用电话，那么另一方也只能使用电话，而不是书信。可以说，电话就是两人对话的协议。协议是对话中使用的通信规则，把上述概念拓展到计算机领域可整理为“计算机间对话必备通信规则”。

各位是否已理解了协议的含义？简言之，协议就是为了完成数据交换而定好的约定。

创建套接字

创建套接字所用的socket函数已经在第1章中简单介绍过。但为了完全理解该函数，此处将再次展开讨论，本章的主要目的也在于此。

```
#include <sys/socket.h>

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

➔ 成功时返回文件描述符，失败时返回-1。

- domain　套接字中使用的协议族（Protocol Family）信息。
- type　套接字数据传输类型信息。
- protocol　计算机间通信中使用的协议信息。

第1章并未提及该函数的参数，但它们对创建套接字来说是不可或缺的。下面给出详细说明。

协议族（Protocol Family）

奶油意大利面和番茄酱意大利面均属于意大利面的一种，与之类似，套接字通信中的协议也具有一些分类。通过socket函数的第一个参数传递套接字中使用的协议分类信息。此协议分类信息称为协议族，可分成如下几类。

表2-1　头文件sys/socket.h中声明的协议族

名　称	协　议　族
PF_INET	IPv4互联网协议族
PF_INET6	IPv6互联网协议族
PF_LOCAL	本地通信的UNIX协议族
PF_PACKET	底层套接字的协议族
PF_IPX	IPX Novell协议族

本书将着重讲解表2-1中PF_INET对应的IPv4互联网协议族。其他协议族并不常用或尚未普及，因此本书将重点放在PF_INET协议族上。另外，套接字中实际采用的最终协议信息是通过socket函数的第三个参数传递的。在指定的协议族范围内通过第一个参数决定第三个参数。

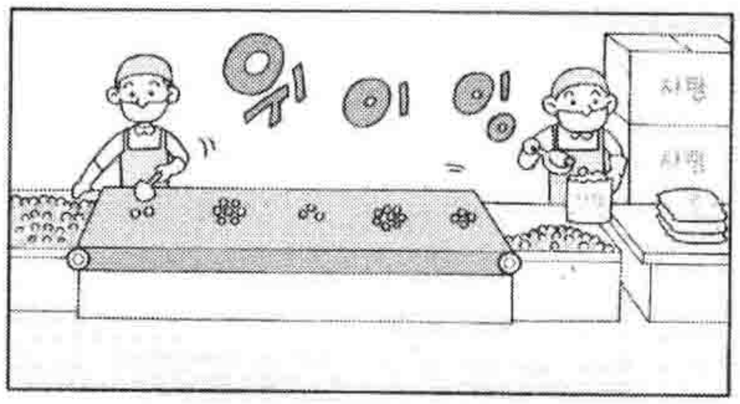
套接字类型（Type）

套接字类型指的是套接字的数据传输方式，通过socket函数的第二个参数传递，只有这样才能决定创建的套接字的数据传输方式。这种说法可能会使各位感到疑惑。已通过第一个参数传递了协议族信息，还要决定数据传输方式？问题就在于，决定了协议族并不能同时决定数据传输方式，换言之，socket函数第一个参数PF_INET协议族中也存在多种数据传输方式。

下面介绍2种具有代表性的数据传输方式。这是理解好套接字的重要前提，请各位务必掌握。

套接字类型 1：面向连接的套接字（SOCK_STREAM）

如果向socket函数的第二个参数传递SOCK_STREAM，将创建面向连接的套接字。面向连接的套接字到底具有哪些特点呢？右图中2位工人通过1条传送带传递物品，这与面向连接的数据传输方式类似。



右图的数据（糖果）传输方式特征整理如下。

- 传输过程中数据不会消失。
- 按序传输数据。
- 传输的数据不存在数据边界（Boundary）。

图中通过独立的传送带传输数据（糖果），只要传送带本身没有问题，就能保证数据不丢失。同时，较晚传递的数据不会先到达，因为传送带保证了数据的按序传递。最后，下面这句话说明的确不存在数据边界：

“100个糖果是分批传递的，但接收者凑齐100个后才装袋。”

这种情形可以适用到之前说过的write和read函数。

“传输数据的计算机通过3次调用write函数传递了100字节的数据，但接收数据的计算机仅通过1次read函数调用就接收了全部100个字节。”

收发数据的套接字内部有缓冲（buffer），简言之就是字节数组。通过套接字传输的数据将保存到该数组。因此，收到数据并不意味着马上调用read函数。只要不超过数组容量，则有可能在数据填充满缓冲后通过1次read函数调用读取全部，也有可能分成多次read函数调用进行读取。也就是说，在面向连接的套接字中，read函数和write函数的调用次数并无太大意义。所以说面向连接的套接字不存在数据边界。稍后将给出示例以查看该特性。

知识补给站　套接字缓冲已满是否意味着数据丢失

之前讲过，为了接收数据，套接字内部有一个由字节数组构成的缓冲。如果这个缓冲被接收的数据填满会发生什么事情？之后传递的数据是否会丢失？

首先调用read函数从缓冲读取部分数据，因此，缓冲并不总是满的。但如果read函数读取速度比接收数据的速度慢，则缓冲有可能被填满。此时套接字无法再接收数据，但即使这样也不会发生数据丢失，因为传输端套接字将停止传输。也就是说，面向连接的套接字会根据接收端的状态传输数据，如果传输出错还会提供重传服务。因此，面向连接的套接字除特殊情况外不会发生数据丢失。

还有一点需要说明。上图中传输和接收端各有1名工人，这说明面向连接的套接字还有如下特点：

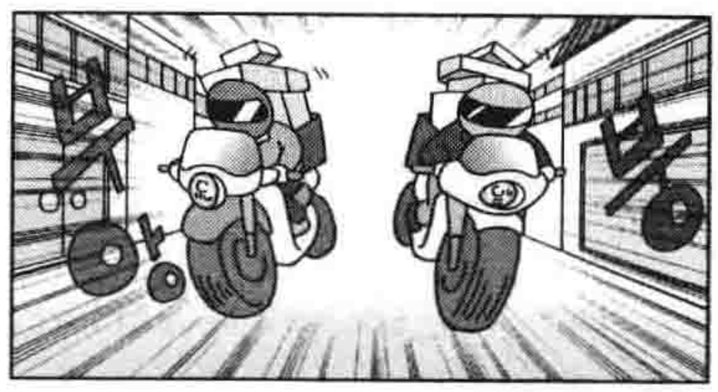
“套接字连接必须一一对应。”

面向连接的套接字只能与另外一个同样特性的套接字连接。用一句话概括面向连接的套接字如下：

“可靠的、按序传递的、基于字节的面向连接的数据传输方式的套接字”

这是我自己的总结，希望各位深入理解其含义，不要仅停留于字面表达。

套接字类型2：面向消息的套接字（SOCK_DGRAM）



如果向socket函数的第二个参数传递SOCK_DGRAM，则将创建面向消息的套接字。面向消息的套接字可以比喻成高速移动的摩托车快递。右图中摩托车快递的包裹（数据）传输方式如下。

- 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。
- 传输的数据有数据边界。
- 限制每次传输的数据大小。

众所周知，快递行业的速度就是生命。用摩托车发往同一目的地的2件包裹无需保证顺序，只要以最快速度交给客户即可。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，若要传递大量包裹，则需分批发送。另外，如果用2辆摩托车分别发送2件包裹，则接收者也需要分2次接收。这种特性就是“传输的数据具有数据边界”。

以上就是面向消息的套接字具有的特性。即，面向消息的套接字比面向连接的套接字具有更快的传输速度，但无法避免数据丢失或损毁。另外，每次传输的数据大小具有一定限制，并存在数据边界。存在数据边界意味着接收数据的次数应和传输次数相同。面向消息的套接字特性总结如下：

“不可靠的、不按序传递的、以数据的高速传输为目的的套接字”

另外，面向消息的套接字不存在连接的概念，这一点将在以后章节介绍。

协议的最终选择

下面讲解socket函数的第三个参数，该参数决定最终采用的协议。各位是否觉得有些困惑？

前面已经通过socket函数的前两个参数传递了协议族信息和套接字数据传输方式，这些信息还不足以决定采用的协议吗？为什么还需要传递第3个参数呢？

正如各位所想，传递前两个参数即可创建所需套接字。所以大部分情况下可以向第三个参数传递0，除非遇到以下这种情况：

“同一协议族中存在多个数据传输方式相同的协议”

数据传输方式相同，但协议不同。此时需要通过第三个参数具体指定协议信息。

下面以前面讲解的内容为基础，构建向socket函数传递的参数。首先创建满足如下要求的套接字：

“IPv4协议族中面向连接的套接字”

IPv4与网络地址系统相关，关于这一点将给出单独说明，目前只需记住：本书是基于IPv4展开的。参数PF_INET指IPv4网络协议族，SOCK_STREAM是面向连接的数据传输。满足这2个条件的协议只有IPPROTO_TCP，因此可以如下调用socket函数创建套接字，这种套接字称为TCP套接字。

```
int tcp_socket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

下面创建满足如下要求的套接字：

“IPv4协议族中面向消息的套接字”

SOCK_DGRAM指的是面向消息的数据传输方式，满足上述条件的协议只有IPPROTO_UDP。因此，可以如下调用socket函数创建套接字，这种套接字称为UDP套接字。

```
int udp_socket = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

前面进行了大量描述以解释这两行代码，这是为了让大家理解它们创建的套接字的特性。

面向连接的套接字：TCP 套接字示例

其他章节将讲解UDP套接字，此处只给出面向连接的TCP套接字示例。本示例是在第1章的如下2个源文件基础上修改而成的。

- hello_server.c → tcp_server.c：无变化！
- hello_client.c → tcp_client.c：更改read函数调用方式！

之前的hello_server.c和hello_client.c是基于TCP套接字的示例，现调整其中一部分代码，以验证TCP套接字的如下特性：

“传输的数据不存在数据边界。”

为验证这一点，需要让write函数的调用次数不同于read函数的调用次数。因此，在客户端中分多次调用read函数以接收服务器端发送的全部数据。

❖ tcp_client.c

```
1.  #include <“头信息与hello_client.c一致，故省略。”>
2.  void error_handling(char *message);
3.
4.  int main(int argc, char* argv[])
5.  {
6.      int sock;
7.      struct sockaddr_in serv_addr;
8.      char message[30];
9.      int str_len=0;
10.     int idx=0, read_len=0;
11.
12.     if(argc!=3){
13.         printf("Usage : %s <IP> <port>\n", argv[0]);
14.         exit(1);
15.     }
16.
17.     sock=socket(PF_INET, SOCK_STREAM, 0);
18.     if(sock == -1)
19.         error_handling("socket() error");
20.
21.     memset(&serv_addr, 0, sizeof(serv_addr));
22.     serv_addr.sin_family=AF_INET;
23.     serv_addr.sin_addr.s_addr=inet_addr(argv[1]);
24.     serv_addr.sin_port=htons(atoi(argv[2]));
25.
26.     if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)
27.         error_handling("connect() error!");
28.
29.     while(read_len=read(sock, &message[idx++], 1))
30.     {
31.         if(read_len==-1)
32.             error_handling("read() error!");
33.
34.         str_len+=read_len;
35.     }
36.
37.     printf("Message from server: %s \n", message);
38.     printf("Function read call count: %d \n", str_len);
39.     close(sock);
40.     return 0;
41. }
42.
43. void error_handling(char *message)
44. {
45.     //与以前示例一致，故省略!
46. }
```

代码说明

- 第17行：创建TCP套接字。若前两个参数传递PF_INET、SOCK_STREAM，则可以省略第三个参数IPPROTO_TCP。
- 第29行：while循环中反复调用read函数，每次读取1个字节。如果read返回0，则循环条件为假，跳出while循环。
- 第34行：执行该语句时，变量read_len的值始终为1，因为第29行每次读取1个字节。跳出while循环后，str_len中存有读取的总字节数。

与该示例配套使用的服务器端tcp_server.c与hello_server.c完全相同，故省略其源代码。执行方式也与hello_server.c和hello_client.c相同，因此只给出最终运行结果。

❖ 运行结果：hello_client.c

```
root@my_linux:/tcpip# gcc tcp_client.c -o hclient
root@my_linux:/tcpip# ./hclient 127.0.0.1 9190
Message from server: Hello World!
Function read call count: 13
```

从运行结果可以看出，服务器端发送了13字节的数据，客户端调用13次read函数进行读取。希望各位通过该示例深入理解TCP套接字的数据传输方式。

2.2　Windows 平台下的实现及验证

前面讲过的套接字类型及传输特性与操作系统无关。Windows平台下的实现方式也类似，不需要过多说明，只需稍加了解socket函数返回类型即可。

Windows 操作系统的 socket 函数

Windows的函数名和参数名都与Linux平台相同，只是返回值类型稍有不同。再次给出socket函数的声明。

```
#include <winsock2.h>

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

➔ 成功时返回 socket 句柄，失败时返回 INVALID_SOCKET。

该函数的参数种类及含义与Linux的socket函数完全相同，故省略，只讨论返回值类型。可以看出返回值类型为SOCKET，此结构体用来保存整数型套接字句柄值。实际上，socket函数返回整数型数据，因此可以通过int型变量接收，就像在Linux中做的一样。但考虑到以后的扩展性，