```
42.         strLen=recv(sock, message, sizeof(message)-1, 0);
43.         message[strLen]=0;
44.         printf("Message from server: %s", message);
45.     }
46.     closesocket(sock);
47.     WSACleanup();
48.     return 0;
49. }
50.
51. void ErrorHandling(char *message)
52. {
53.     fputs(message, stderr);
54.     fputc('\n', stderr);
55.     exit(1);
56. }
```

上述客户端示例利用已连接UDP套接字进行输入输出，因此用send、recv函数替换sendto、recvfrom函数。此外也如实反映了已连接UDP套接字的好处。

6.5 习题

(1) UDP为什么TCP速度快？为什么TCP数据传输可靠而UDP数据传输不可靠？

(2) 下列不属于UDP特点的是？

a. UDP不同于TCP，不存在连接的概念，所以不像TCP那样只能进行一对一的数据传输。
b. 利用UDP传输数据时，如果有2个目标，则需要2个套接字。
c. UDP套接字中无法使用已分配给TCP的同一端口号。
d. UDP套接字和TCP套接字可以共存。若需要，可以在同一主机进行TCP和UDP数据传输。
e. 针对UDP函数也可以调用connect函数，此时UDP套接字跟TCP套接字相同，也需要经过3次握手过程。

(3) UDP数据报向对方主机的UDP套接字传递过程中，IP和UDP分别负责哪些部分？

(4) UDP一般比TCP快，但根据交换数据的特点，其差异可大可小。请说明何种情况下UDP的性能优于TCP？

(5) 客户端TCP套接字调用connect函数时自动分配IP和端口号。UDP中不调用bind函数，那合适分配IP和端口号？

(6) TCP客户端必需调用connect函数，而UDP中可以选择性调用。请问，在UDP中调用connect函数有哪些好处？

(7) 请参考本章给出的uecho_server.c和uecho_client.c，编写示例使服务器端和客户端轮流收发消息。收发的消息均要输出到控制台窗口。

第7章

优雅地断开套接字连接

本章将讨论如何优雅地断开相互连接的套接字。之前用的方法不够优雅是因为，我们是调用 close 或 closesocket 函数单方面断开连接的。

7.1 基于 TCP 的半关闭

TCP中的断开连接过程比建立连接过程更重要，因为连接过程中一般不会出现大的变数，但断开过程有可能发生预想不到的情况，因此应准确掌控。只有掌握了下面要讲解的半关闭（Half-close），才能明确断开过程。

单方面断开连接带来的问题

Linux的close函数和Windows的closesocket函数意味着完全断开连接。完全断开不仅指无法传输数据，而且也不能接收数据。因此，在某些情况下，通信一方调用close或closesocket函数断开连接就显得不太优雅，如图7-1所示。

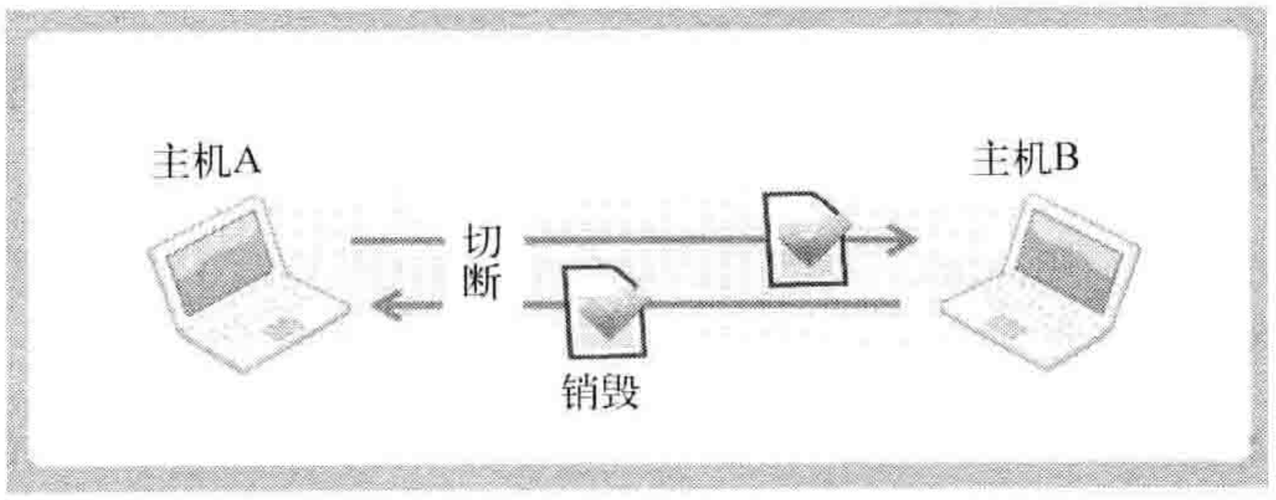


图7-1　单方面断开连接

图7-1描述的是2台主机正在进行双向通信。主机A发送完最后的数据后，调用close函数断开了连接，之后主机A无法再接收主机B传输的数据。实际上，是完全无法调用与接收数据相关的

函数。最终，由主机B传输的、主机A必须接收的数据也销毁了。

为了解决这类问题，“只关闭一部分数据交换中使用的流”（Half-close）的方法应运而生。断开一部分连接是指，可以传输数据但无法接收，或可以接收数据但无法传输。顾名思义就是只关闭流的一半。

套接字和流（Stream）

两台主机通过套接字建立连接后进入可交换数据的状态，又称“流形成的状态”。也就是把建立套接字后可交换数据的状态看作一种流。

此处的流可以比作水流。水朝着一个方向流动，同样，在套接字的流中，数据也只能向一个方向移动。因此，为了进行双向通信，需要如图7-2所示的2个流。

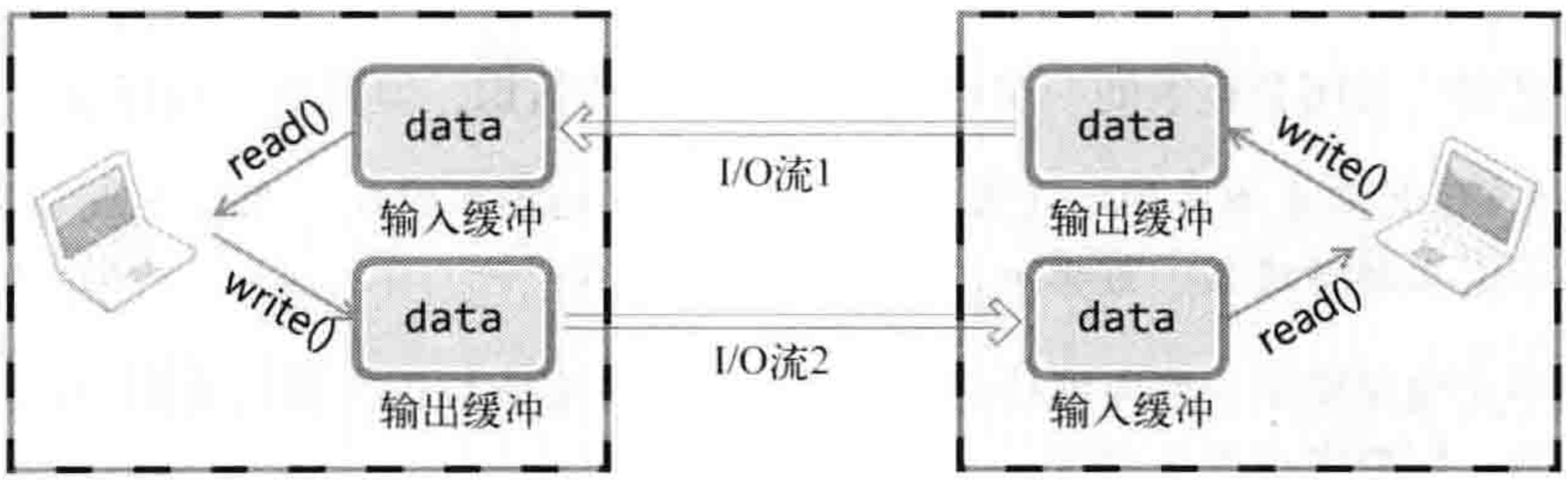


图7-2 套接字中生成的两个流

一旦两台主机间建立了套接字连接，每个主机就会拥有单独的输入流和输出流。当然，其中一个主机的输入流与另一主机的输出流相连，而输出流则与另一主机的输入流相连。另外，本章讨论的“优雅地断开连接方式”只断开其中1个流，而非同时断开两个流。Linux的close和Windows的closesocket函数将同时断开这两个流，因此与“优雅”二字还有一段距离。

针对优雅断开的 shutdown 函数

接下来介绍用于半关闭的函数。下面这个shutdown函数就用来关闭其中1个流。

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

int shutdown(int sock, int howto);
```

➔ 成功时返回 0，失败时返回-1。

- sock　需要断开的套接字文件描述符。
- howto　传递断开方式信息。

调用上述函数时，第二个参数决定断开连接的方式，其可能值如下所示。

- SHUT_RD：断开输入流。
- SHUT_WR：断开输出流。
- SHUT_RDWR：同时断开I/O流。

若向shutdown的第二个参数传递SHUT_RD，则断开输入流，套接字无法接收数据。即使输入缓冲收到数据也会抹去，而且无法调用输入相关函数。如果向shutdown函数的第二个参数传递SHUT_WR，则中断输出流，也就无法传输数据。但如果输出缓冲还留有未传输的数据，则将传递至目标主机。最后，若传入SHUT_RDWR，则同时中断I/O流。这相当于分2次调用shutdown，其中一次以SHUT_RD为参数，另一次以SHUT_WR为参数。

为何需要半关闭

相信各位已对“关闭套接字的一半连接”有了充分的认识，但还有一些疑惑。

“究竟为什么需要半关闭？是否只要留出足够长的连接时间，保证完成数据交换即可？只要不急于断开连接，好像也没必要使用半关闭。”

这句话也不完全是错的。如果保持足够的时间间隔，完成数据交换后再断开连接，这时就没必要使用半关闭。但要考虑如下情况：

“一旦客户端连接到服务器端，服务器端将约定的文件传给客户端，客户端收到后发送字符串‘Thank you’给服务器端。”

此处字符串“Thank you”的传递实际是多余的，这只是用来模拟客户端断开连接前还有数据需要传递的情况。此时程序实现的难度并不小，因为传输文件的服务器端只需连续传输文件数据即可，而客户端则无法知道需要接收数据到何时。客户端也没办法无休止地调用输入函数，因为这有可能导致程序阻塞（调用的函数未返回）。

“是否可以让服务器端和客户端约定一个代表文件尾的字符？”

这种方式也有问题，因为这意味着文件中不能有与约定字符相同的内容。为解决该问题，服务器端应最后向客户端传递EOF表示文件传输结束。客户端通过函数返回值接收EOF，这样可以避免与文件内容冲突。剩下最后一个问题：服务器如何传递EOF?

“断开输出流时向对方主机传输EOF。”

当然，调用close函数的同时关闭I/O流，这样也会向对方发送EOF。但此时无法再接收对方传输的数据。换言之，若调用close函数关闭流，就无法接收客户端最后发送的字符串“Thank you”。这时需要调用shutdown函数，只关闭服务器的输出流（半关闭）。这样既可以发送EOF，同时又保留了输入流，可以接收对方数据。下面结合已学内容实现收发文件的服务器端/客户端。

基于半关闭的文件传输程序

上述文件传输服务器端和客户端的数据流可整理如图7-3，稍后将根据此图编写示例。希望各位通过此例理解传递EOF的必要性和半关闭的重要性。

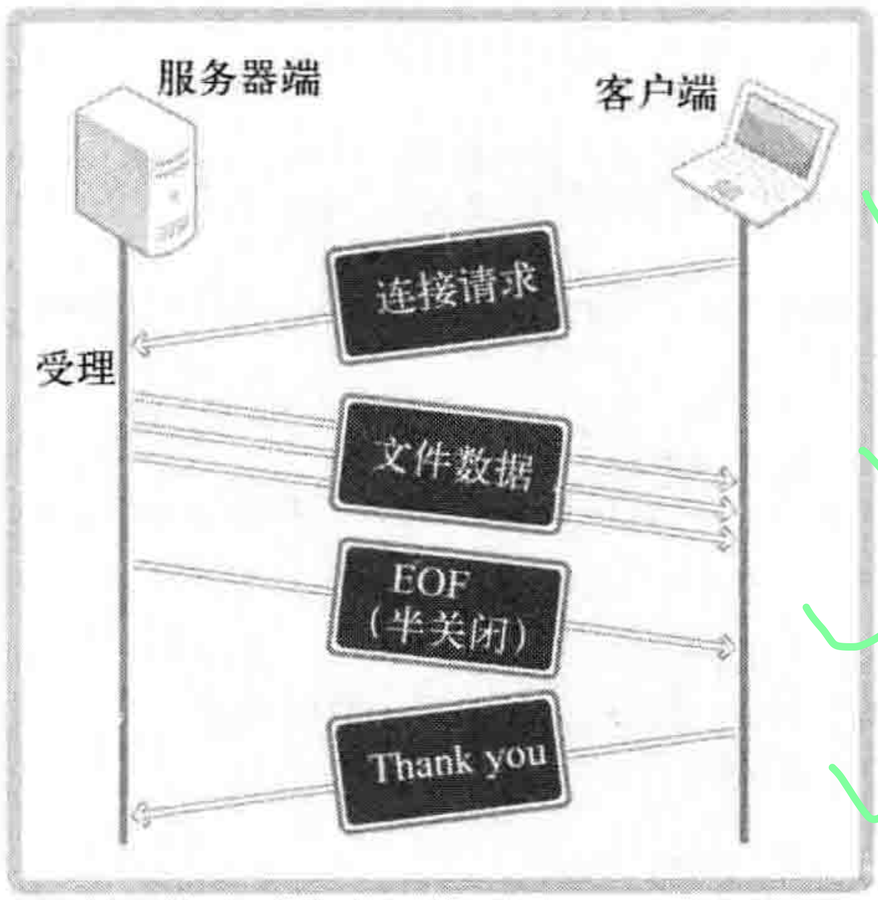


图7-3 文件传输数据流程图

首先介绍服务器端。该示例与之前示例不同，省略了大量错误处理代码，希望大家注意。这种处理只是为了便于分析代码，实际编写中不应省略。

❖ file_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.
8.  #define BUF_SIZE 30
9.  void error_handling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     int serv_sd, clnt_sd;
14.     FILE * fp;
15.     char buf[BUF_SIZE];
16.     int read_cnt;
17.
18.     struct sockaddr_in serv_adr, clnt_adr;
19.     socklen_t clnt_adr_sz;
20.
21.     if(argc!=2) {
```

```
22.         printf("Usage: %s <port>\n", argv[0]);
23.         exit(1);
24.     }
25.
26.     fp=fopen("file_server.c", "rb");
27.     serv_sd=socket(PF_INET, SOCK_STREAM, 0);
28.
29.     memset(&serv_adr, 0, sizeof(serv_adr));
30.     serv_adr.sin_family=AF_INET;
31.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
32.     serv_adr.sin_port=htons(atoi(argv[1]));
33.
34.     bind(serv_sd, (struct sockaddr*)&serv_adr, sizeof(serv_adr));
35.     listen(serv_sd, 5);
36.
37.     clnt_adr_sz=sizeof(clnt_adr);
38.     clnt_sd=accept(serv_sd, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
39.
40.     while(1)
41.     {
42.         read_cnt=fread((void*)buf, 1, BUF_SIZE, fp);
43.         if(read_cnt<BUF_SIZE)
44.         {
45.             write(clnt_sd, buf, read_cnt);
46.             break;
47.         }
48.         write(clnt_sd, buf, BUF_SIZE);
49.     }
50.
51.     shutdown(clnt_sd, SHUT_WR);
52.     read(clnt_sd, buf, BUF_SIZE);
53.     printf("Message from client: %s \n", buf);
54.
55.     fclose(fp);
56.     close(clnt_sd); close(serv_sd);
57.     return 0;
58. }
59.
60. void error_handling(char *message)
61. {
62.     fputs(message, stderr);
63.     fputc('\n', stderr);
64.     exit(1);
65. }
```

代码说明

- 第26行：打开文件以向客户端传输服务器端源文件file_server.c。
- 第40~49行：为向客户端传输文件数据而编写的循环语句。此客户端是在第38行的accept函数调用中连接的。
- 第51行：发送文件后针对输出流进行半关闭。这样就向客户端传输了EOF，而客户端也知道文件传输已完成。
- 第52行：只关闭了输出流，依然可以通过输入流接收数据。

❖ file_client.c

```
1.  #include <"与file_server.c头声明一致，故省略。">
2.  #define BUF_SIZE 30
3.  void error_handling(char *message);
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int sd;
8.      FILE *fp;
9.
10.     char buf[BUF_SIZE];
11.     int read_cnt;
12.     struct sockaddr_in serv_adr;
13.     if(argc!=3) {
14.         printf("Usage: %s <IP> <port>\n", argv[0]);
15.         exit(1);
16.     }
17.
18.     fp=fopen("receive.dat", "wb");
19.     sd=socket(PF_INET, SOCK_STREAM, 0);
20.
21.     memset(&serv_adr, 0, sizeof(serv_adr));
22.     serv_adr.sin_family=AF_INET;
23.     serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
24.     serv_adr.sin_port=htons(atoi(argv[2]));
25.
26.     connect(sd, (struct sockaddr*)&serv_adr, sizeof(serv_adr));
27.
28.     while((read_cnt=read(sd, buf, BUF_SIZE ))!=0)
29.         fwrite((void*)buf, 1, read_cnt, fp);
30.
31.     puts("Received file data");
32.     write(sd, "Thank you", 10);
33.     fclose(fp);
34.     close(sd);
35.     return 0;
36. }
37.
38. void error_handling(char *message)
39. {
40.     fputs(message, stderr);
41.     fputc('\n', stderr);
42.     exit(1);
43. }
```

代码说明

- 第18行：创建新文件以保存服务器端传输的文件数据。
- 第28、29行：接收数据并保存到第18行创建的文件，直到接收EOF为止。
- 第32行：向服务器端发送表示感谢的消息。若服务器端未关闭输入流，则可接收此消息。

下面是上述示例的运行结果。运行后查看客户端的receive.dat文件，可以验证数据正常接收。

特别需要注意的是，还可以确认服务器端已正常接收客户端最后传输的消息“Thank you”。

❖ 运行结果：file_server.c

```
root@my_linux:/tcpip# gcc file_server.c -o fserver
root@my_linux:/tcpip# ./fserver 9190
Message from client: Thank you
root@my_linux:/tcpip#
```

❖ 运行结果：file_client.c

```
root@my_linux:/tcpip# gcc file_client.c -o fclient
root@my_linux:/tcpip# ./fclient 127.0.0.1 9190
Received file data
root@my_linux:/tcpip#
```

7.2　基于 Windows 的实现

Windows平台同样通过调用shutdown函数完成半关闭，只是向其传递的参数名略有不同，需要确认。

```
#include <winsock2.h>

int shutdown(SOCKET sock, int howto);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

- sock　要断开的套接字句柄。
- howto 断开方式的信息。

上述函数中第二个参数的可能值及其含义可整理如下。

- SD_RECEIVE：断开输入流。
- SD_SEND：断开输出流。
- SD_BOTH：同时断开I/O流。

虽然这些常量名不同于Linux中的名称，但其值完全相同。SD_RECEIVE、SHUT_RD都是0，SD_SEND、SHUT_WR都是1，SD_BOTH、SHUT_RDWR都是2。当然，这些并没有太多实际意义。最后，给出Windows平台下的示例。

❖ file_server_win.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock2.h>

#define BUF_SIZE 30
void ErrorHandling(char *message);

int main(int argc, char *argv[])
{
    WSADATA wsaData;
    SOCKET hServSock, hClntSock;
    FILE * fp;
    char buf[BUF_SIZE];
    int readCnt;

    SOCKADDR_IN servAdr, clntAdr;
    int clntAdrSz;

    if(argc!=2) {
        printf("Usage: %s <port>\n", argv[0]);
        exit(1);
    }
    if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
        ErrorHandling("WSAStartup() error!");

    fp=fopen("file_server_win.c", "rb");
    hServSock=socket(PF_INET, SOCK_STREAM, 0);

    memset(&servAdr, 0, sizeof(servAdr));
    servAdr.sin_family=AF_INET;
    servAdr.sin_addr.s_addr=htonl(INADDR_ANY);
    servAdr.sin_port=htons(atoi(argv[1]));

    bind(hServSock, (SOCKADDR*)&servAdr, sizeof(servAdr));
    listen(hServSock, 5);

    clntAdrSz=sizeof(clntAdr);
    hClntSock=accept(hServSock, (SOCKADDR*)&clntAdr, &clntAdrSz);

    while(1)
    {
        readCnt=fread((void*)buf, 1, BUF_SIZE, fp);
        if(readCnt<BUF_SIZE)
        {
            send(hClntSock, (char*)&buf, readCnt, 0);
            break;
        }
        send(hClntSock, (char*)&buf, BUF_SIZE, 0);
    }

```

```
52.     shutdown(hClntSock, SD_SEND);
53.     recv(hClntSock, (char*)buf, BUF_SIZE, 0);
54.     printf("Message from client: %s \n", buf);
55.
56.     fclose(fp);
57.     closesocket(hClntSock); closesocket(hServSock);
58.     WSACleanup();
59.     return 0;
60. }
61.
62. void ErrorHandling(char *message)
63. {
64.     fputs(message, stderr);
65.     fputc('\n', stderr);
66.     exit(1);
67. }
```

❖ file_client_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.
6.  #define BUF_SIZE 30
7.  void ErrorHandling(char *message);
8.
9.  int main(int argc, char *argv[])
10. {
11.     WSADATA wsaData;
12.     SOCKET hSocket;
13.     FILE *fp;
14.
15.     char buf[BUF_SIZE];
16.     int readCnt;
17.     SOCKADDR_IN servAdr;
18.
19.     if(argc!=3) {
20.         printf("Usage: %s <IP> <port>\n", argv[0]);
21.         exit(1);
22.     }
23.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
24.         ErrorHandling("WSAStartup() error!");
25.
26.     fp=fopen("receive.dat", "wb");
27.     hSocket=socket(PF_INET, SOCK_STREAM, 0);
28.
29.     memset(&servAdr, 0, sizeof(servAdr));
30.     servAdr.sin_family=AF_INET;
31.     servAdr.sin_addr.s_addr=inet_addr(argv[1]);
32.     servAdr.sin_port=htons(atoi(argv[2]));
33.
```

```
34.     connect(hSocket, (SOCKADDR*)&servAdr, sizeof(servAdr));
35.
36.     while((readCnt=recv(hSocket, buf, BUF_SIZE, 0))!=0)
37.         fwrite((void*)buf, 1, readCnt, fp);
38.
39.     puts("Received file data");
40.     send(hSocket, "Thank you", 10, 0);
41.     fclose(fp);
42.     closesocket(hSocket);
43.     WSACleanup();
44.     return 0;
45. }
46.
47. void ErrorHandling(char *message)
48. {
49.     fputs(message, stderr);
50.     fputc('\n', stderr);
51.     exit(1);
52. }
```

运行结果及源代码内容与之前的file_server.c、file_client.c并无太大区别，故省略。

7.3 习题

(1) 解释TCP中“流”的概念。UDP中能否形成流？请说明原因。

(2) Linux中的close函数或Windows中的closesocket函数属于单方面断开连接的方法，有可能带来一些问题。什么是单方面断开连接？什么情形下会出现问题？

(3) 什么是半关闭？针对输出流执行半关闭的主机处于何种状态？半关闭会导致对方主机接收什么消息？

第8章

域名及网络地址

8

随着互联网用户的不断增加，现在 DNS（Domain Name System，域名系统）几乎无人不知。人们也经常谈论 DNS 相关的专业话题。而且，不懂 DNS 的人用不了 5 分钟就能在网上搜索并学习 DNS 知识。网络的发展促使我们每个人都成了半个网络专家。

8.1 域名系统

DNS是对IP地址和域名进行相互转换的系统，其核心是DNS服务器。

什么是域名

提供网络服务的服务器端也是通过IP地址区分的，但几乎不可能以非常难记的IP地址形式交换服务器端地址信息。因此，将容易记、易表述的域名分配并取代IP地址。

DNS 服务器

在浏览器地址栏中输入Naver网站的IP地址222.122.195.5即可浏览Naver网站主页。但我们通常输入Naver网站的域名www.naver.com访问网站。二者之间究竟有何区别？

从进入Naver网站主页这一结果上看，没有区别，但接入过程不同。域名是赋予服务器端的虚拟地址，而非实际地址。因此，需要将虚拟地址转化为实际地址。那如何将域名变为IP地址呢？DNS服务器担此重任，可以向DNS服务器请求转换地址。

“请问DNS服务器，www.naver.com的IP地址是多少？”

所有计算机中都记录着默认DNS服务器地址，就是通过这个默认DNS服务器得到相应域名的IP地址信息。在浏览器地址栏中输入域名后，浏览器通过默认DNS服务器获取该域名对应的IP地

址信息，之后才真正接入该网站。

提 示

ping & nslookup

除非商业需要，否则一般不会轻易改变服务器域名，但会相对频繁地改变服务器 IP 地址。如果各位想了解某个域名对应的 IP 地址信息，可以在控制台窗口输入如下内容：

```
ping www.naver.com
```

这样即可知道某一域名对应的 IP 地址。ping 命令用来验证 IP 数据报是否到达目的地，但执行过程中会同时经过“域名到 IP 地址”的转换过程，因此可以通过此命令查看 IP 地址。另外，若各位想知道自己计算机中注册的默认 DNS 服务器地址，可以输入如下命令：

```
nslookup
```

在 Linux 系统中输入上述命令后，会提示进一步输入信息，此时可以输入 server 得到默认 DNS 服务器地址。

计算机内置的默认DNS服务器并不知道网络上所有域名的IP地址信息。若该DNS服务器无法解析，则会询问其他DNS服务器，并提供给用户，如图8-1所示。

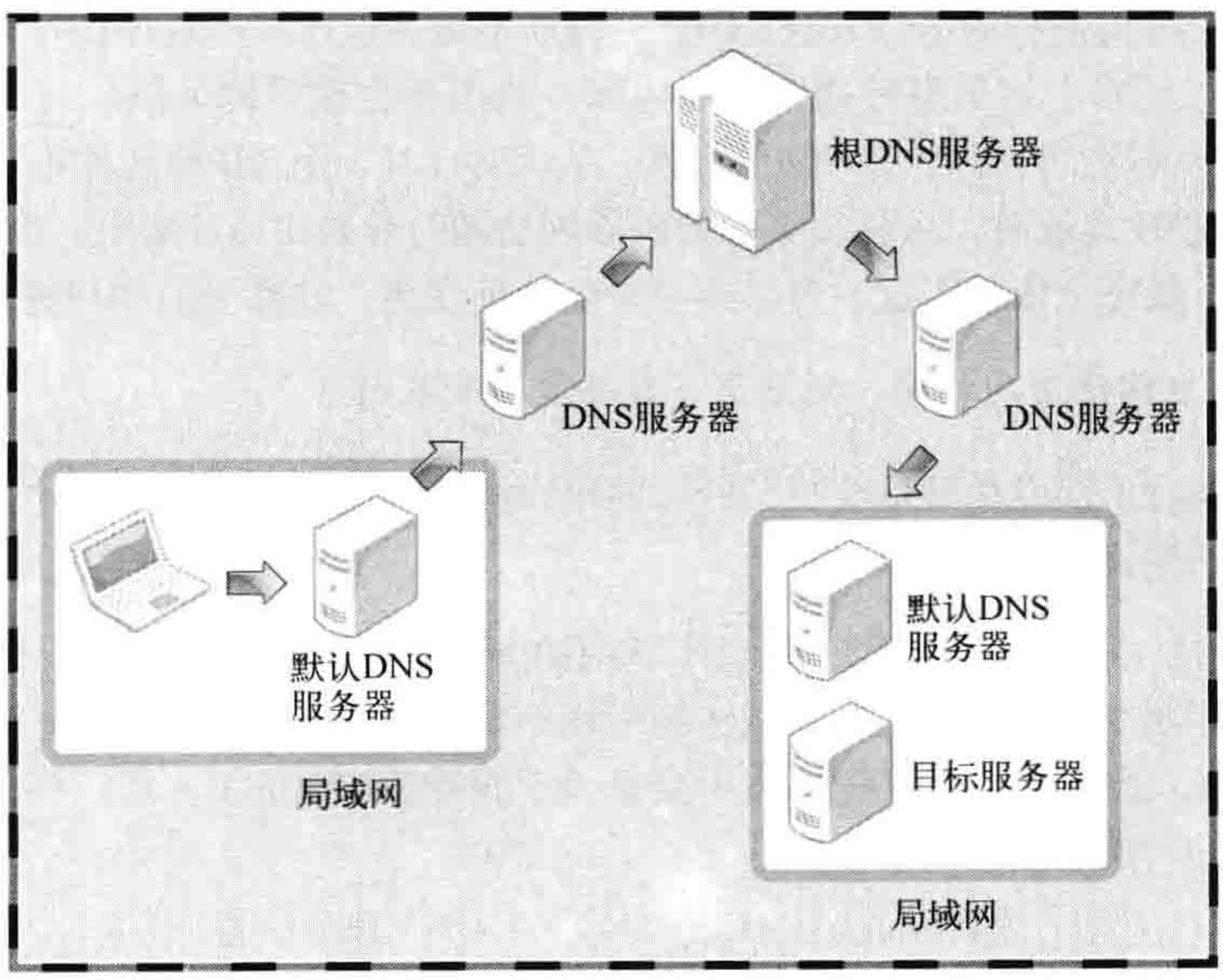


图8-1 DNS和请求获取IP地址信息

图8-1展示了默认DNS服务器无法解析主机询问的域名IP地址时的应答过程。可以看出，默认DNS服务器收到自己无法解析的请求时，向上级DNS服务器询问。通过这种方式逐级向上传递信息，到达顶级DNS服务器——根DNS服务器时，它知道该向哪个DNS服务器询问。向下级DNS传递解析请求，得到IP地址后原路返回，最后将解析的IP地址传递到发起请求的主机。DNS就是这样层次化管理的一种分布式数据库系统。

8.2　IP 地址和域名之间的转换

8.1节讲解了域名和IP地址之间的转换过程，本节介绍通过程序向DNS服务器发出解析请求的方法。

程序中有必要使用域名吗？

“所有学习都要在开始前认识到其必要性！”这是我经常挂在嘴边的一句话。从语言的基本语法到系统函数，若无法回答“这到底有何必要？”学习过程将变得枯燥无味，而且很容易遗忘。最头疼的是，学完之后很难应用。我们为什么需要将要讨论的转换函数？为了查看某一域名的IP地址吗？当然不是！下面通过示例解释原因。假设各位是运营www.SuperOrange.com域名的公司系统工程师，需要开发客户端使用公司提供的服务。该客户端需要接入如下服务器地址：

IP 211.102.204.12, PORT 2012

应向程序用户提供便利的运行方法，因此，程序不能像运行示例程序那样要求用户输入IP和端口信息。那该如何将上述信息传递到程序内部？难道要直接将地址信息写入程序代码吗？当然，这样便于运行程序，但这种方案也有问题。系统运行时，保持IP地址并不容易。特别是依赖ISP服务提供者维护IP地址时，系统相关的各种原因会随时导致IP地址变更。虽然ISP会保证维持原有IP，但程序不能完全依赖于这一点。万一发生地址变更，就需要向用户进行如下解释：

“请卸载当前使用的程序，到主页下载并重新安装v1.2。”

那么，因为随时可能发生地址变更，所以向用户提供源代码，每次变更地址时让用户改变IP和端口号，并重新编译程序，这又如何？

IP地址比域名发生变更的概率要高，所以利用IP地址编写程序并非上策。还有什么办法呢？一旦注册域名可能永久不变，因此利用域名编写程序会好一些。这样，每次运行程序时根据域名获取IP地址，再接入服务器，这样程序就不会依赖于服务器IP地址了。所以说，程序中也需要IP地址和域名之间的转换函数。

利用域名获取IP地址

使用以下函数可以通过传递字符串格式的域名获取IP地址。

```
#include <netdb.h>

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

➔ 成功时返回 hostent 结构体地址，失败时返回 NULL 指针。

这个函数使用方便。只要传递域名字符串，就会返回域名对应的IP地址。只是返回时，地址信息装入hostent结构体。此结构体定义如下。

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

从上述结构体定义中可以看出，不只返回IP信息，同时还连带着其他信息。各位不用想得太过复杂。域名转IP时只需关注h_addr_list。下面简要说明上述结构体各成员。

- h_name

该变量中存有官方域名（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

可以通过多个域名访问同一主页。同一IP可以绑定多个域名，因此，除官方域名外还可指定其他域名。这些信息可以通过h_aliases获得。

- h_addrtype

gethostbyname函数不仅支持IPv4，还支持IPv6。因此可以通过此变量获取保存在h_addr_list的IP地址的地址族信息。若是IPv4，则此变量存有AF_INET。

- h_length

保存IP地址长度。若是IPv4地址，因为是4个字节，则保存4；IPv6时，因为是16个字节，故

保存16。

h_addr_list

这是最重要的成员。通过此变量以整数形式保存域名对应的IP地址。另外，用户较多的网站有可能分配多个IP给同一域名，利用多个服务器进行负载均衡。此时同样可以通过此变量获取IP地址信息。

调用gethostbyname函数后返回的hostent结构体的变量结构如图8-2所示，该图在实际编程中非常有用，希望大家结合之前的hostent结构体定义加以理解。

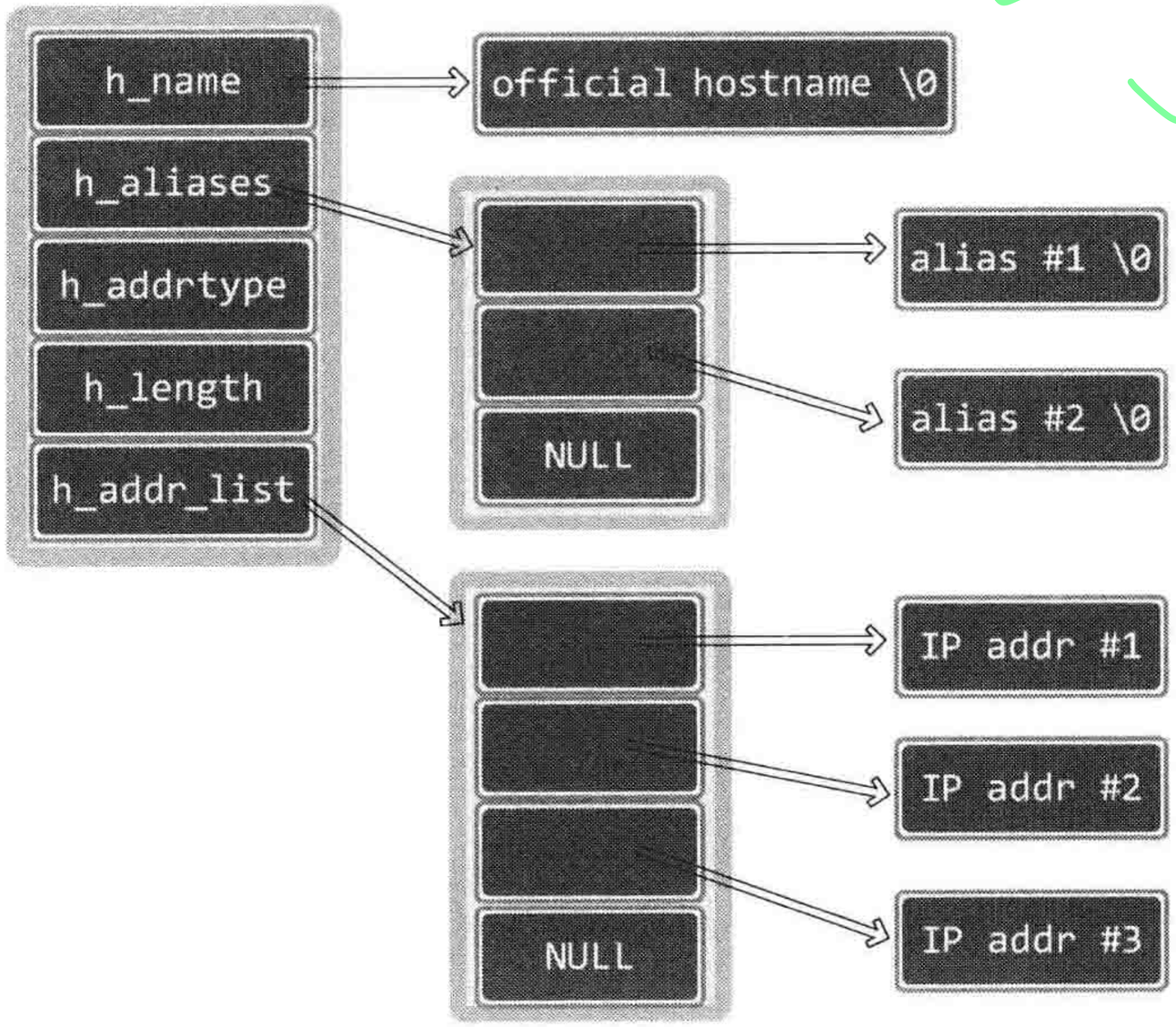


图8-2　hostent结构体变量

下列示例主要演示gethostbyname函数的应用，并说明hostent结构体变量的特性。

gethostbyname.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <arpa/inet.h>
5.  #include <netdb.h>
6.  void error_handling(char *message);
7.
8.  int main(int argc, char *argv[])
9.  {
10.     int i;
```

```
11.     struct hostent *host;
12.     if(argc!=2) {
13.         printf("Usage : %s <addr>\n", argv[0]);
14.         exit(1);
15.     }
16.
17.     host=gethostbyname(argv[1]);
18.     if(!host)
19.         error_handling("gethost... error");
20.
21.     printf("Official name: %s \n", host->h_name);
22.     for(i=0; host->h_aliases[i]; i++)
23.         printf("Aliases %d: %s \n", i+1, host->h_aliases[i]);
24.     printf("Address type: %s \n",
25.         (host->h_addrtype==AF_INET)?"AF_INET":"AF_INET6");
26.     for(i=0; host->h_addr_list[i]; i++)
27.         printf("IP addr %d: %s \n", i+1,
28.             inet_ntoa(*(struct in_addr*)host->h_addr_list[i]));
29.     return 0;
30. }
31.
32. void error_handling(char *message)
33. {
34.     fputs(message, stderr);
35.     fputc('\n', stderr);
36.     exit(1);
37. }
```

代码说明

- 第17行：将通过main函数传递的字符串用作参数调用gethostbyname。
- 第21行：输出官方域名。
- 第22、23行：输出除官方域名以外的域名。这样编写循环语句的原因可从图8-2中找到答案。
- 第26~28行：输出IP地址信息。但多了令人感到困惑的类型转换。关于这一点稍后将给出说明。

❖ 运行结果：gethostbyname.c

```
root@my_linux:/tcpip# gcc gethostbyname.c -o hostname
root@my_linux:/tcpip# ./hostname www.naver.com
Official name: www.g.naver.com
Aliases 1: www.naver.com
Address type: AF_INET
IP addr 1: 202.131.29.70
IP addr 2: 222.122.195.6
```

我利用Naver网站域名运行了上述示例，大家可以任选一个域名验。现在讨论上述示例的第26~28行。若只看hostent结构体的定义，结构体成员h_addr_list指向字符串指针数组（由多个字符

串地址构成的数组）。但字符串指针数组中的元素实际指向的是（实际保存的是）in_addr结构体变量地址值而非字符串，如图8-3所示。

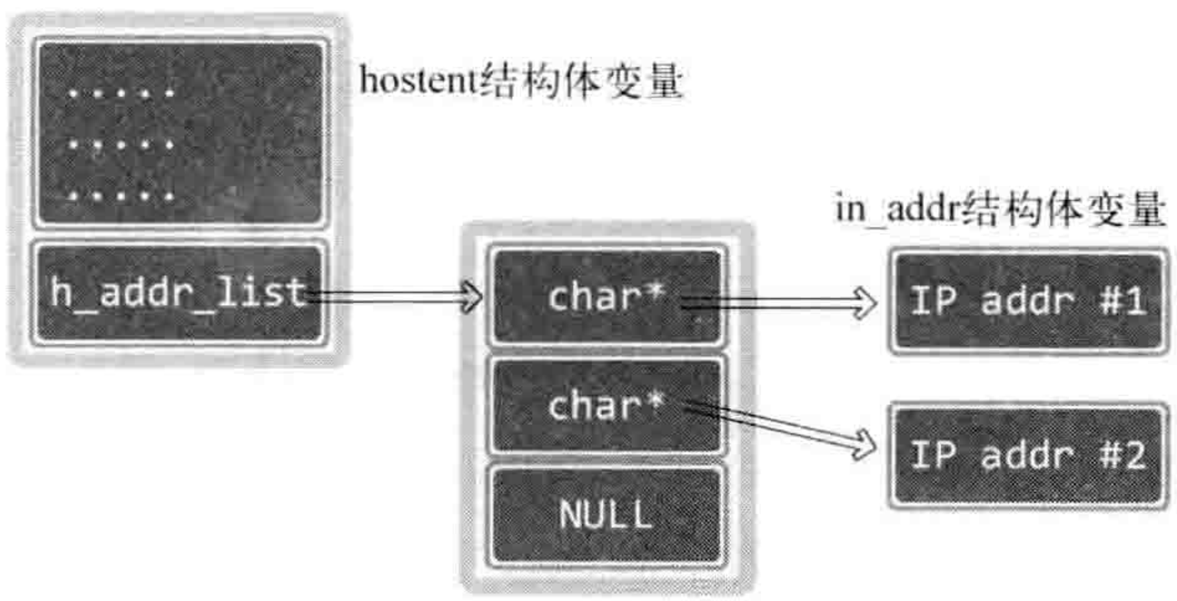


图8-3　h_addr_list结构体成员

图8-3给出了h_addr_list结构体的参照关系。正因如此，示例的第28行需要进行类型转换，并调用inet_ntoa函数。另外，in_addr结构体的声明可以参考第3章。

提　示

为什么是 char* 而不是 in_addr*

hostent 结构体的成员 h_addr_list 指向的数组类型并不是 in_addr 结构体的指针数组，而是采用了 char 指针。各位也许对这一点感到困惑，但我认为大家应该能料到。hostent 结构体并非只为 IPv4 准备。h_addr_list 指向的数组中也可以保存 IPv6 地址信息。因此，考虑到通用性，声明为 char 指针类型的数组。

“声明为 void 指针类型是否更合理？”

若能想到这一点，说明对 C 语言掌握非常到位。当然如此。指针对象不明确时，更适合使用 void 指针类型。但各位目前学习的套接字相关函数都是在 void 指针标准化之前定义的，而当时无法明确指出指针类型时采用的是 char 指针。

利用 IP 地址获取域名

之前介绍的gethostbyname函数利用域名获取包括IP地址在内的域相关信息。而gethostbyaddr函数利用IP地址获取域相关信息。

```
#include <netdb.h>

struct hostent * gethostbyaddr(const char * addr, socklen_t len, int family);
```

➔ 成功时返回 hostent 结构体变量地址值，失败时返回 NULL 指针。

- addr　含有IP地址信息的in_addr结构体指针。为了同时传递IPv4地址之外的其他信息，该变量的类型声明为char指针。
- len　向第一个参数传递的地址信息的字节数，IPv4时为4，IPv6时为16。
- family　传递地址族信息，IPv4时为AF_INET，IPv6时为AF_INET6。

如果已经彻底掌握gethostbyname函数，那么上述函数理解起来并不难。下面通过示例演示该函数的使用方法。

❖ gethostbyaddr.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <netdb.h>
7.  void error_handling(char *message);
8.
9.  int main(int argc, char *argv[])
10. {
11.     int i;
12.     struct hostent *host;
13.     struct sockaddr_in addr;
14.     if(argc!=2) {
15.         printf("Usage : %s <IP>\n", argv[0]);
16.         exit(1);
17.     }
18.
19.     memset(&addr, 0, sizeof(addr));
20.     addr.sin_addr.s_addr=inet_addr(argv[1]);
21.     host=gethostbyaddr((char*)&addr.sin_addr, 4, AF_INET);
22.     if(!host)
23.         error_handling("gethost... error");
24.
25.     printf("Official name: %s \n", host->h_name);
26.     for(i=0; host->h_aliases[i]; i++)
27.         printf("Aliases %d: %s \n", i+1, host->h_aliases[i]);
28.     printf("Address type: %s \n",
29.         (host->h_addrtype==AF_INET)?"AF_INET":"AF_INET6");
30.     for(i=0; host->h_addr_list[i]; i++)
31.         printf("IP addr %d: %s \n", i+1,
32.             inet_ntoa(*(struct in_addr*)host->h_addr_list[i]));
```

```
33.     return 0;
34. }
35.
36. void error_handling(char *message)
37. {
38.     fputs(message, stderr);
39.     fputc('\n', stderr);
40.     exit(1);
41. }
```

除第21行的gethostbyaddr函数调用过程外，与gethostbyname.c并无区别，因为函数调用的结果是通过hostent结构体变量地址值传递的。

❖ 运行结果：gethostbyaddr.c

```
root@my_linux:/tcpip# gcc gethostbyaddr.c -o hostaddr
root@my_linux:/tcpip# ./hostaddr 74.125.19.106
Official name: nuq04s01-in-f106.google.com
Address type: AF_INET
IP addr 1: 74.125.19.106
```

我通过ping命令得到了Google的IP地址，并利用此信息运行了示例。从运行结果可以看到，记录于DNS的官方主页地址具有特殊格式。

8.3　基于 Windows 的实现

Windows平台中也有类似功能的同名函数，因此无需经过太多变更。先介绍gethostbyname函数。

```
#include <winsock2.h>

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

➔ 成功时返回 hostent 结构体变量地址值，失败时返回 NULL 指针。

函数名、参数及返回类型与Linux中没有区别，故省略，继续介绍下一函数。

```
#include <winsock2.h>

struct hostent * gethostbyaddr(const char * addr, int len, int type);
```

➔ 成功时返回 hostent 结构体变量地址值，失败时返回 NULL 指针。

上述函数也与Linux中的函数完全一致，故省略，下面在示例中进行实际调用。

❖ gethostbyname_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.      int i;
10.     struct hostent *host;
11.     if(argc!=2) {
12.         printf("Usage : %s <addr>\n", argv[0]);
13.         exit(1);
14.     }
15.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
16.         ErrorHandling("WSAStartup() error!");
17.
18.     host=gethostbyname(argv[1]);
19.     if(!host)
20.         ErrorHandling("gethost... error");
21.
22.     printf("Official name: %s \n", host->h_name);
23.     for(i=0; host->h_aliases[i]; i++)
24.         printf("Aliases %d: %s \n", i+1, host->h_aliases[i]);
25.     printf("Address type: %s \n",
26.         (host->h_addrtype==AF_INET)?"AF_INET":"AF_INET6");
27.     for(i=0; host->h_addr_list[i]; i++)
28.         printf("IP addr %d: %s \n", i+1,
29.             inet_ntoa(*(struct in_addr*)host->h_addr_list[i]));
30.     WSACleanup();
31.     return 0;
32. }
33.
34. void ErrorHandling(char *message)
35. {
36.     fputs(message, stderr);
37.     fputc('\n', stderr);
38.     exit(1);
39. }
```

❖ gethostbyaddr_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.  void ErrorHandling(char *message);
```

```
6.
7.  int main(int argc, char *argv[])
8.  {
9.      WSADATA wsaData;
10.     int i;
11.     struct hostent *host;
12.     SOCKADDR_IN addr;
13.     if(argc!=2) {
14.         printf("Usage : %s <IP>\n", argv[0]);
15.         exit(1);
16.     }
17.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
18.         ErrorHandling("WSAStartup() error!");
19.
20.     memset(&addr, 0, sizeof(addr));
21.     addr.sin_addr.s_addr=inet_addr(argv[1]);
22.     host=gethostbyaddr((char*)&addr.sin_addr, 4, AF_INET);
23.     if(!host)
24.         ErrorHandling("gethost... error");
25.
26.     printf("Official name: %s \n", host->h_name);
27.     for(i=0; host->h_aliases[i]; i++)
28.         printf("Aliases %d: %s \n", i+1, host->h_aliases[i]);
29.     printf("Address type: %s \n",
30.         (host->h_addrtype==AF_INET)?"AF_INET":"AF_INET6");
31.     for(i=0; host->h_addr_list[i]; i++)
32.         printf("IP addr %d: %s \n", i+1,
33.             inet_ntoa(*(struct in_addr*)host->h_addr_list[i]));
34.     WSACleanup();
35.     return 0;
36. }
37.
38. void ErrorHandling(char *message)
39. {
40.     fputs(message, stderr);
41.     fputc('\n', stderr);
42.     exit(1);
43. }
```

各位可能也会认为没必要给出运行结果，故省略。基于Windows的实现相关讲解到此结束。

8.4　习题

(1) 下列关于DNS的说法错误的是?

a. 因为DNS存在，故可以用域名替代IP。

b. DNS服务器实际上是路由器，因为路由器根据域名决定数据路径。

c. 所有域名信息并非集中于1台DNS服务器，但可以获取某一DNS服务器中未注册的IP地址。

d. DNS服务器根据操作系统进行区分，Windows下的DNS服务器和Linux下的DNS服务器是不同的。

(2) 阅读如下对话，并说明东秀的解决方案是否可行。这些都是大家可以在大学计算机实验室验证的内容。

- 静洙："东秀吗？我们学校网络中使用的默认DNS服务器发生了故障，无法访问我要投简历的公司主页！有没有办法解决？"
- 东秀："网络连接正常，但DNS服务器发生了故障？"
- 静洙："恩！有没有解决方法？是不是要去周围的网吧？"
- 东秀："有那必要吗？我把我们学校的DNS服务器IP地址告诉你，你改一下你的默认DNS服务器地址。"
- 静洙："这样可以吗？默认DNS服务器必须连接到本地网络吧！"
- 东秀："不是！上次我们学校DNS服务器发生故障时，网管就给了我们其他DNS服务器的IP地址呢。"
- 静洙："那是因为你们学校有多台DNS服务器！"
- 东秀："是吗？你的话好像也有道理。那你快去网吧吧！"

(3) 在浏览器地址栏输入www.orentec.co.kr，并整理出主页显示过程。假设浏览器访问的默认DNS服务器中并没有关于www.orentec.co.kr的IP地址信息。

第9章

套接字的多种可选项

套接字具有多种特性，这些特性可通过可选项更改。本章将介绍更改套接字可选项的方法，并以此为基础进一步观察套接字内部。

9.1 套接字可选项和 I/O 缓冲大小

我们进行套接字编程时往往只关注数据通信，而忽略了套接字具有的不同特性。但是，理解这些特性并根据实际需要进行更改也十分重要。

套接字多种可选项

我们之前写的程序都是创建好套接字后（未经特别操作）直接使用的，此时通过默认的套接字特性进行数据通信。之前的示例较为简单，无需特别操作套接字特性，但有时的确需要更改。表9-1列出了一部分套接字可选项。

表9-1 可设置套接字的多种可选项

协议层	选项名	读取	设置
SOL_SOCKET	SO_SNDBUF	O	O
	SO_RCVBUF	O	O
	SO_REUSEADDR	O	O
	SO_KEEPALIVE	O	O
	SO_BROADCAST	O	O
	SO_DONTROUTE	O	O
	SO_OOBINLINE	O	O
	SO_ERROR	O	X
	SO_TYPE	O	X
IPPROTO_IP	IP_TOS	O	O
	IP_TTL	O	O
	IP_MULTICAST_TTL	O	O
	IP_MULTICAST_LOOP	O	O
	IP_MULTICAST_IF	O	O

（续）

协 议 层	选 项 名	读 取	设 置
IPPROTO_TCP	TCP_KEEPALIVE	O	O
	TCP_NODELAY	O	O
	TCP_MAXSEG	O	O

从表9-1中可以看出，套接字可选项是分层的。IPPROTO_IP层可选项是IP协议相关事项，IPPROTO_TCP层可选项是TCP协议相关的事项，SOL_SOCKET层是套接字相关的通用可选项。

也许有人看到表格会产生畏惧感，但现在无需全部背下来或理解，因此不必有负担。实际能够设置的可选项数量是表9-1的好几倍，也无需一下子理解所有可选项，实际工作中逐一掌握即可。接触的可选项多了，自然会掌握大部分重要的。本书也只介绍其中一部分重要的可选项含义及更改方法。

getsockopt & setsockopt

我们几乎可以针对表9-1中的所有可选项进行读取（Get）和设置（Set）（当然，有些可选项只能进行一种操作）。可选项的读取和设置通过如下2个函数完成。

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

int getsockopt(int sock, int level, int optname, void *optval, socklen_t *optlen);
```

➔ 成功时返回0，失败时返回-1。

- sock　用于查看选项套接字文件描述符。
- level　要查看的可选项的协议层。
- optname　要查看的可选项名。
- optval　保存查看结果的缓冲地址值。
- optlen　向第四个参数optval传递的缓冲大小。调用函数后，该变量中保存通过第四个参数返回的可选项信息的字节数。

上述函数用于读取套接字可选项，并不难。接下来介绍更改可选项时调用的函数。

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

int setsockopt(int sock, int level, int optname, const void *optval, socklen_t optlen);
```

➔ 成功时返回0，失败时返回-1。

sock	用于更改可选项的套接字文件描述符。
level	要更改的可选项协议层。
optname	要更改的可选项名。
optval	保存要更改的选项信息的缓冲地址值。
optlen	向第四个参数optval传递的可选项信息的字节数。

接下来介绍这些函数的调用方法。关于setsockopt函数的调用方法在其他示例中给出，先介绍getsockopt函数的调用方法。下列示例用协议层为SOL_SOCKET、名为SO_TYPE的可选项查看套接字类型（TCP或UDP）。

❖ sock_type.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <sys/socket.h>
5.  void error_handling(char *message);
6.
7.  int main(int argc, char *argv[])
8.  {
9.      int tcp_sock, udp_sock;
10.     int sock_type;
11.     socklen_t optlen;
12.     int state;
13.
14.     optlen=sizeof(sock_type);
15.     tcp_sock=socket(PF_INET, SOCK_STREAM, 0);
16.     udp_sock=socket(PF_INET, SOCK_DGRAM, 0);
17.     printf("SOCK_STREAM: %d \n", SOCK_STREAM);
18.     printf("SOCK_DGRAM: %d \n", SOCK_DGRAM);
19.
20.     state=getsockopt(tcp_sock, SOL_SOCKET, SO_TYPE, (void*)&sock_type, &optlen);
21.     if(state)
22.         error_handling("getsockopt() error!");
23.     printf("Socket type one: %d \n", sock_type);
24.
25.     state=getsockopt(udp_sock, SOL_SOCKET, SO_TYPE, (void*)&sock_type, &optlen);
26.     if(state)
27.         error_handling("getsockopt() error!");
28.     printf("Socket type two: %d \n", sock_type);
29.     return 0;
30. }
31.
32. void error_handling(char *message)
33. {
34.     fputs(message, stderr);
35.     fputc('\n', stderr);
36.     exit(1);
37. }
```

代码说明

- 第15、16行：分别生成TCP、UDP套接字。
- 第17、18行：输出创建TCP、UDP套接字时传入的SOCK_STREAM、SOCK_DGRAM。
- 第20、25行：获取套接字类型信息。如果是TCP套接字，将获得SOCK_STREAM常数值1；如果是UDP套接字，则获得SOCK_DGRAM的常数值2。

❖ 运行结果：sock_type.c

```
root@my_linux:/tcpip# gcc sock_type.c -o socktype
root@my_linux:/tcpip# ./socktype
SOCK_STREAM: 1
SOCK_DGRAM: 2
Socket type one: 1
Socket type two: 2
```

上述示例给出了调用getsockopt函数查看套接字信息的方法。另外，用于验证套接字类型的SO_TYPE是典型的只读可选项，这一点可以通过下面这句话解释：

“套接字类型只能在创建时决定，以后不能再更改。”

SO_SNDBUF & SO_RCVBUF

前面介绍过，创建套接字将同时生成I/O缓冲。如果各位忘了这部分内容，可以复习第5章。接下来将介绍I/O缓冲相关可选项。

SO_RCVBUF是输入缓冲大小相关可选项，SO_SNDBUF是输出缓冲大小相关可选项。用这2个可选项既可以读取当前I/O缓冲大小，也可以进行更改。通过下列示例读取创建套接字时默认的I/O缓冲大小。

❖ get_buf.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <sys/socket.h>
5.  void error_handling(char *message);
6.
7.  int main(int argc, char *argv[])
8.  {
9.      int sock;
10.     int snd_buf, rcv_buf, state;
11.     socklen_t len;
12.
13.     sock=socket(PF_INET, SOCK_STREAM, 0);
14.     len=sizeof(snd_buf);
15.     state=getsockopt(sock, SOL_SOCKET, SO_SNDBUF, (void*)&snd_buf, &len);
```

```
    if(state)
        error_handling("getsockopt() error");

    len=sizeof(rcv_buf);
    state=getsockopt(sock, SOL_SOCKET, SO_RCVBUF, (void*)&rcv_buf, &len);
    if(state)
        error_handling("getsockopt() error");

    printf("Input buffer size: %d \n", rcv_buf);
    printf("Output buffer size: %d \n", snd_buf);
    return 0;
}

void error_handling(char *message)
{
    fputs(message, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

❖ 运行结果：get_buf.c

```
root@my_linux:/tcpip# gcc get_buf.c -o getbuf
root@my_linux:/tcpip# ./getbuf
Input buffer size: 87380
Output buffer size: 16384
```

这是我系统中的运行结果，与各位的运行结果相比可能有较大差异。接下来的程序中将更改I/O缓冲大小。

❖ set_buf.c

```
#include <"头声明与get_buf.c一致，故省略。">
void error_handling(char *message);

int main(int argc, char *argv[])
{
    int sock;
    int snd_buf=1024*3, rcv_buf=1024*3;
    int state;
    socklen_t len;

    sock=socket(PF_INET, SOCK_STREAM, 0);
    state=setsockopt(sock, SOL_SOCKET, SO_RCVBUF, (void*)&rcv_buf, sizeof(rcv_buf));
    if(state)
        error_handling("setsockopt() error!");

    state=setsockopt(sock, SOL_SOCKET, SO_SNDBUF, (void*)&snd_buf, sizeof(snd_buf));
    if(state)
        error_handling("setsockopt() error!");
```

```
19.
20.     len=sizeof(snd_buf);
21.     state=getsockopt(sock, SOL_SOCKET, SO_SNDBUF, (void*)&snd_buf, &len);
22.     if(state)
23.         error_handling("getsockopt() error!");
24.
25.     len=sizeof(rcv_buf);
26.     state=getsockopt(sock, SOL_SOCKET, SO_RCVBUF, (void*)&rcv_buf, &len);
27.     if(state)
28.         error_handling("getsockopt() error!");
29.
30.     printf("Input buffer size: %d \n", rcv_buf);
31.     printf("Output buffer size: %d \n", snd_buf);
32.     return 0;
33. }
34.
35. void error_handling(char *message)
36. {
37.     fputs(message, stderr);
38.     fputc('\n', stderr);
39.     exit(1);
40. }
```

- 第12、16行：I/O缓冲大小更改为3M字节。
- 第21、26行：为了验证I/O缓冲的更改，读取缓冲大小。

❖ 运行结果：set_buf.c

```
root@my_linux:/tcpip# gcc set_buf.c -o setbuf
root@my_linux:/tcpip# ./setbuf
Input buffer size: 6144
Output buffer size: 6144
```

输出结果跟我们预想的完全不同，但也算合理。缓冲大小的设置需谨慎处理，因此不会完全按照我们的要求进行，只是通过调用setsockopt函数向系统传递我们的要求。如果把输出缓冲设置为0并如实反映这种设置，TCP协议将如何进行？如果要实现流控制和错误发生时的重传机制，至少要有一些缓冲空间吧？上述示例虽没有100%按照我们的请求设置缓冲大小，但也大致反映出了通过setsockopt函数设置的缓冲大小。

9.2 SO_REUSEADDR

本节的可选项SO_REUSEADDR及其相关的Time-wait状态很重要，希望大家务必理解并掌握。

发生地址分配错误（Binding Error）

学习SO_REUSEADDR可选项之前，应理解好Time-wait状态。我们读完下列示例后再讨论后续内容。

❖ reuseadr_eserver.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.
8.  #define TRUE 1
9.  #define FALSE 0
10. void error_handling(char *message);
11.
12. int main(int argc, char *argv[])
13. {
14.     int serv_sock, clnt_sock;
15.     char message[30];
16.     int option, str_len;
17.     socklen_t optlen, clnt_adr_sz;
18.     struct sockaddr_in serv_adr, clnt_adr;
19.     if(argc!=2) {
20.         printf("Usage : %s <port>\n", argv[0]);
21.         exit(1);
22.     }
23.
24.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
25.     if(serv_sock==-1)
26.         error_handling("socket() error");
27.     /*
28.     optlen=sizeof(option);
29.     option=TRUE;
30.     setsockopt(serv_sock, SOL_SOCKET, SO_REUSEADDR, (void*)&option, optlen);
31.     */
32.
33.     memset(&serv_adr, 0, sizeof(serv_adr));
34.     serv_adr.sin_family=AF_INET;
35.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
36.     serv_adr.sin_port=htons(atoi(argv[1]));
37.
38.     if(bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr)))
39.         error_handling("bind() error");
40.     if(listen(serv_sock, 5)==-1)
41.         error_handling("listen error");
42.     clnt_adr_sz=sizeof(clnt_adr);
43.     clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr,&clnt_adr_sz);
44.
```

```
45.     while((str_len=read(clnt_sock,message, sizeof(message)))!= 0)
46.     {
47.         write(clnt_sock, message, str_len);
48.         write(1, message, str_len);
49.     }
50.     close(clnt_sock);
51.     close(serv_sock);
51.     return 0;
52. }
53.
54. void error_handling(char *message)
55. {
56.     fputs(message, stderr);
57.     fputc('\n', stderr);
58.     exit(1);
59. }
```

此示例是之前已实现过多次的回声服务器端，可以结合第4章介绍过的回声客户端运行。下面运行该示例，第28～30行应保持注释状态。通过如下方式终止程序：

“在客户端控制台输入Q消息，或通过CTRL+C终止程序。”

也就是说，让客户端先通知服务器端终止程序。在客户端控制台输入Q消息时调用close函数（参考第4章的echo_client.c），向服务器端发送FIN消息并经过四次握手过程。当然，输入CTRL+C时也会向服务器传递FIN消息。强制终止程序时，由操作系统关闭文件及套接字，此过程相当于调用close函数，也会向服务器端传递FIN消息。

“但看不到什么特殊现象啊？”

是的，通常都是由客户端先请求断开连接，所以不会发生特别的事情。重新运行服务器端也不成问题，但按照如下方式终止程序时则不同。

“服务器端和客户端已建立连接的状态下，向服务器端控制台输入CTRL+C，即强制关闭服务器端。”

这主要模拟了服务器端向客户端发送FIN消息的情景。但如果以这种方式终止程序，那服务器端重新运行时将产生问题。如果用同一端口号重新运行服务器端，将输出“bind() error”消息，并且无法再次运行。但在这种情况下，再过大约3分钟即可重新运行服务器端。

上述2种运行方式唯一的区别就是谁先传输FIN消息，但结果却迥然不同，原因何在呢？

Time-wait 状态

相信各位已对四次握手有了很好的理解，先观察该过程，如图9-1所示。

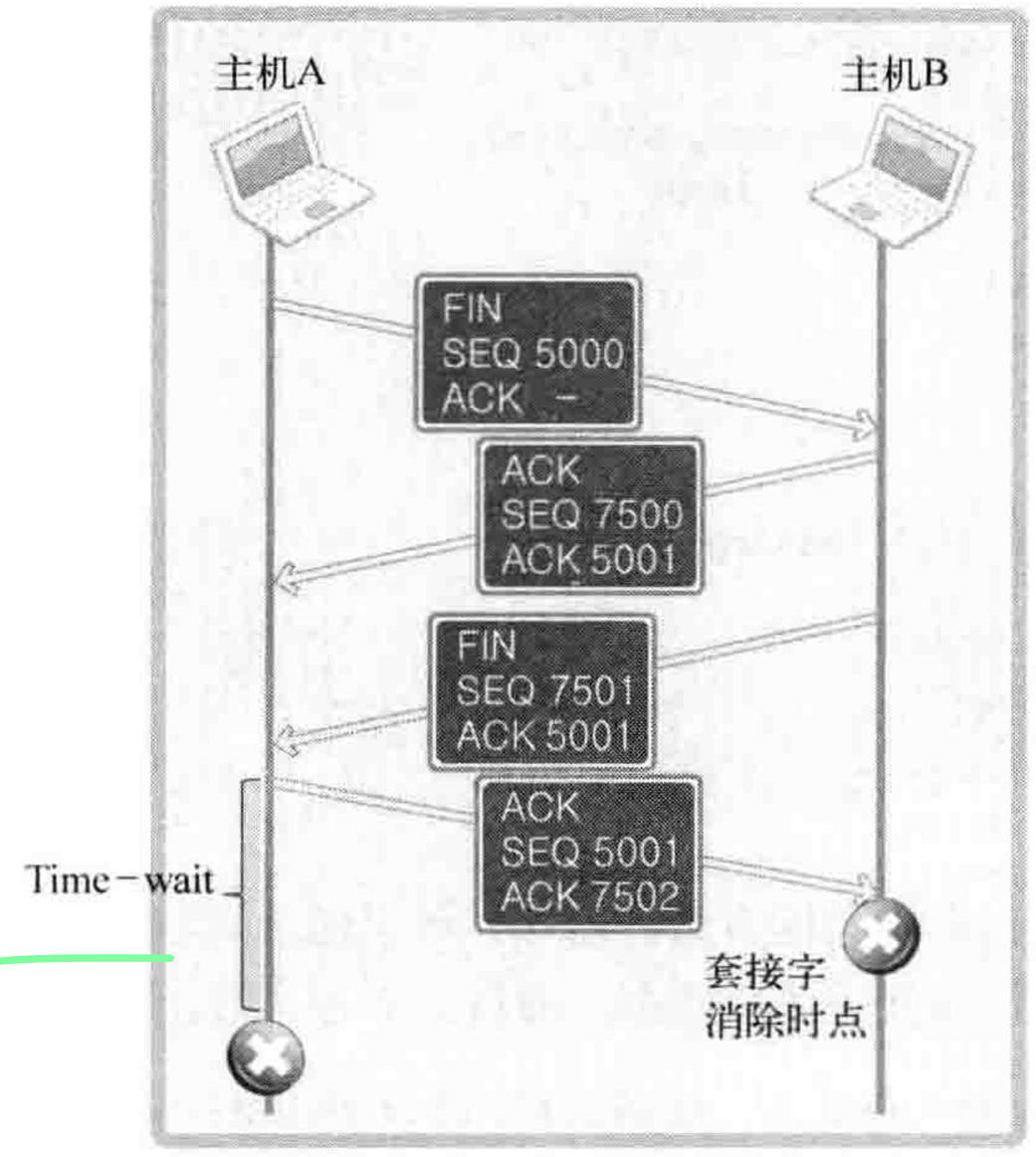


图9-1　Time-wait状态下的套接字

假设图9-1中主机A是服务器端，因为是主机A向B发送FIN消息，故可以想象成服务器端在控制台输入CTRL+C。但问题是，套接字经过四次握手过程后并非立即消除，而是要经过一段时间的Time-wait状态。当然，只有先断开连接的（先发送FIN消息的）主机才经过Time-wait状态。因此，若服务器端先断开连接，则无法立即重新运行。套接字处在Time-wait过程时，相应端口是正在使用的状态。因此，就像之前验证过的，bind函数调用过程中当然会发生错误。

提　示

客户端套接字不会经过 Time-wait 过程吗?

有些人会误以为 Time-wait 过程只存在于服务器端。但实际上，不管是服务器端还是客户端，套接字都会有 Time-wait 过程。先断开连接的套接字必然会经过 Time-wait 过程。但无需考虑客户端 Time-wait 状态。因为客户端套接字的端口号是任意指定的。与服务器端不同，客户端每次运行程序时都会动态分配端口号，因此无需过多关注 Time-wait 状态。

到底为什么会有Time-wait状态呢？图9-1中假设主机A向主机B传输ACK消息（SEQ 5001、ACK 7502）后立即消除套接字。但最后这条ACK消息在传递途中丢失，未能传给主机B。这时会发生什么？主机B会认为之前自己发送的FIN消息（SEQ 7501、ACK 5001）未能抵达主机A，继而试图重传。但此时主机A已是完全终止的状态，因此主机B永远无法收到从主机A最后传来的ACK消息。相反，若主机A的套接字处在Time-wait状态，则会向主机B重传最后的ACK消息，主

机B也可以正常终止。基于这些考虑，先传输FIN消息的主机应经过Time-wait过程。

地址再分配

Time-wait看似重要，但并不一定讨人喜欢。考虑一下系统发生故障从而紧急停止的情况。这时需要尽快重启服务器端以提供服务，但因处于Time-wait状态而必须等待几分钟。因此，Time-wait并非只有优点，而且有些情况下可能引发更大问题。图9-2演示了四次握手时不得不延长Time-wait过程的情况。

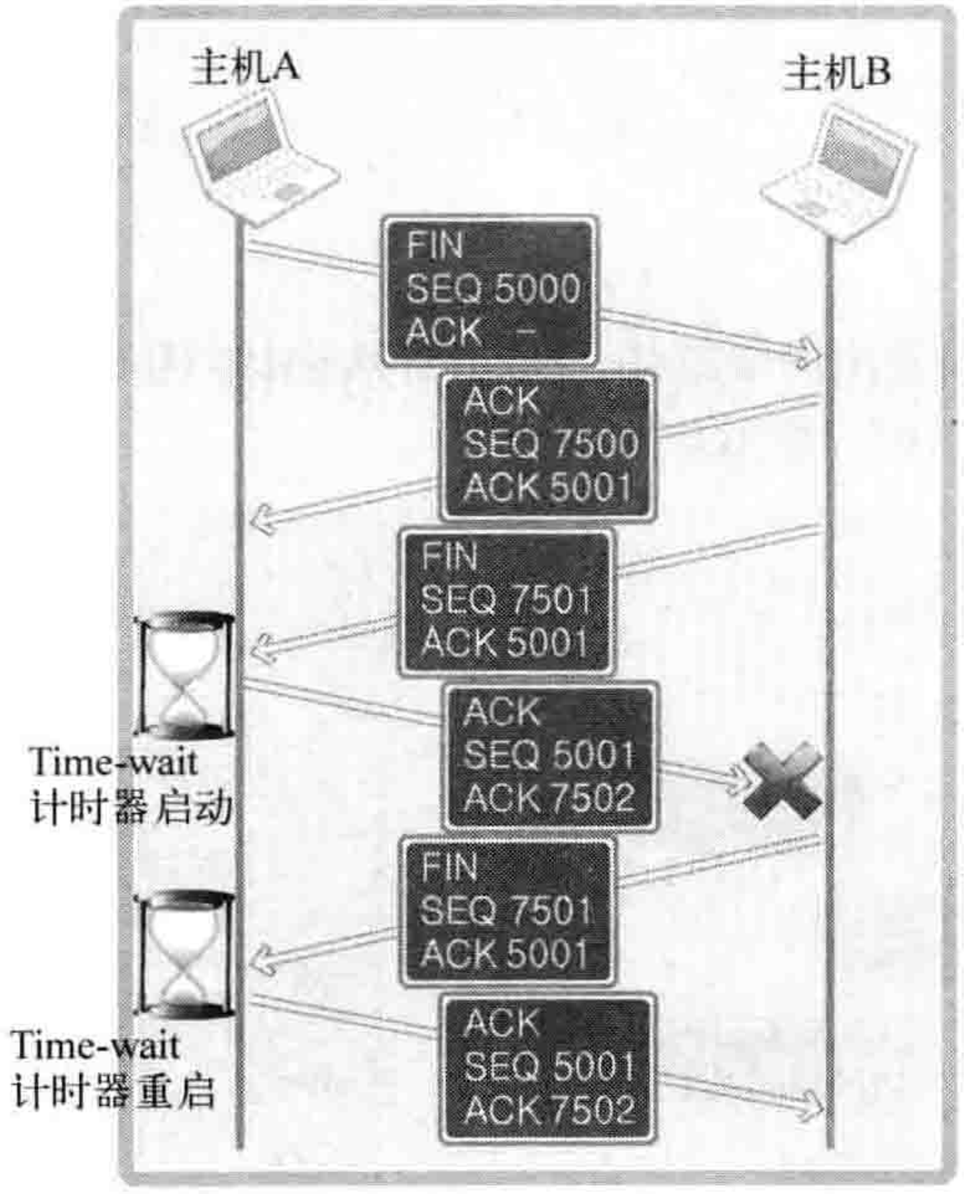


图9-2 重启Time-wait计时器

如图9-2所示，在主机A的四次握手过程中，如果最后的数据丢失，则主机B会认为主机A未能收到自己发送的FIN消息，因此重传。这时，收到FIN消息的主机A将重启Time-wait计时器。因此，如果网络状况不理想，Time-wait状态将持续。

解决方案就是在套接字的可选项中更改SO_REUSEADDR的状态。适当调整该参数，可将Time-wait状态下的套接字端口号重新分配给新的套接字。SO_REUSEADDR的默认值为0（假），这就意味着无法分配Time-wait状态下的套接字端口号。因此需要将这个值改成1（真）。具体做法已在示例reuseadr_eserver.c中给出，只需去掉下述代码的注释即可。

```
optlen=sizeof(option);
option=TRUE;
setsockopt(serv_sock, SOL_SOCKET, SO_REUSEADDR, (void*) &option, optlen);
```

各位是否去掉了注释？既然服务器端reuseadr_eserver.c已变成可随时运行的状态，希望大家在Time-wait状态下验证其能否重新运行。

9.3　TCP_NODELAY

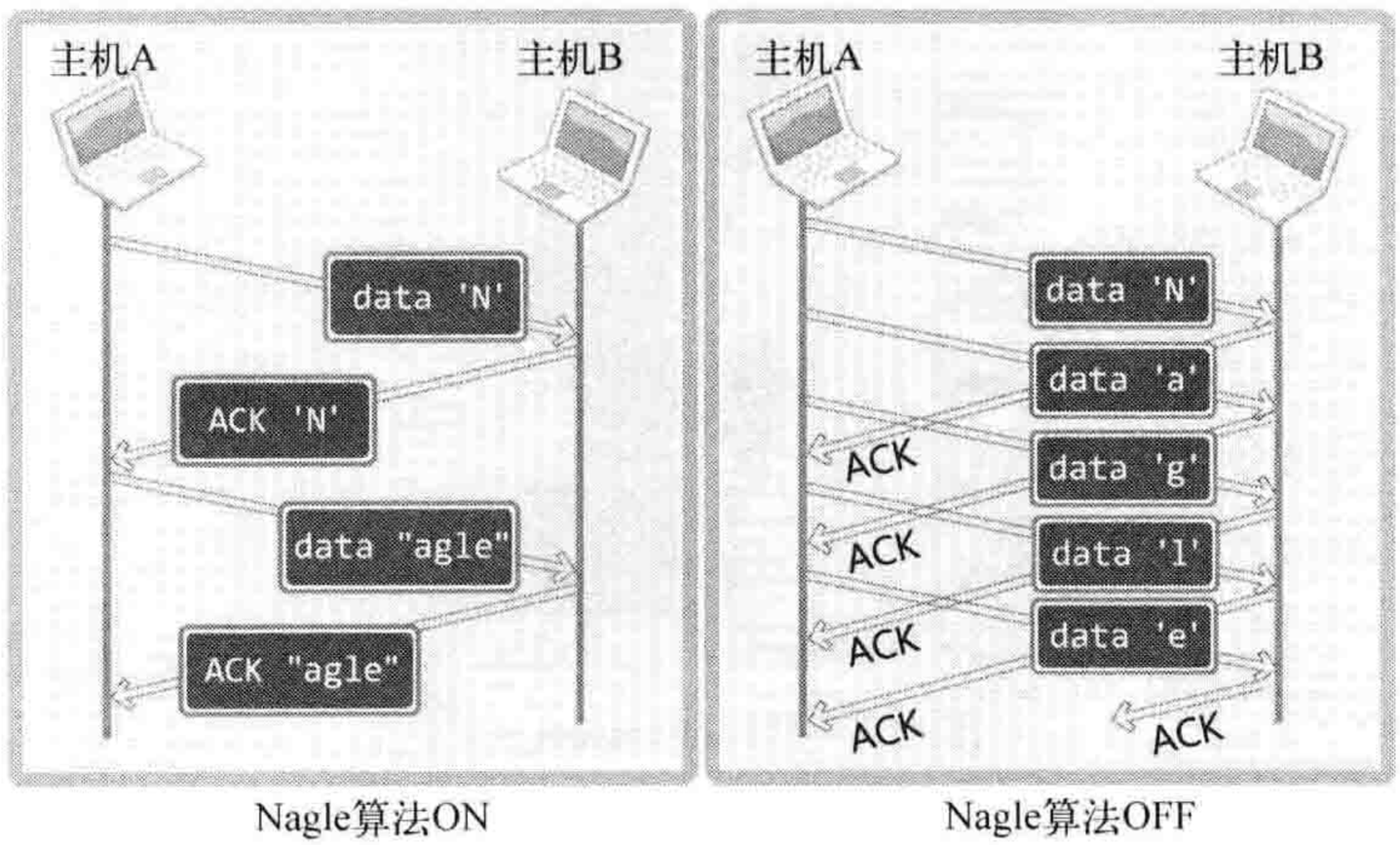
我教Java网络编程时，经常被问及如下问题：

“什么是Nagle算法？使用该算法能够获得哪些数据通信特性？”

我被问到这个问题时感到特别高兴，因为开发人员容易忽视的一个问题就是Nagle算法，下面进行详细讲解。

Nagle 算法

为防止因数据包过多而发生网络过载，Nagle算法在1984年诞生了。它应用于TCP层，非常简单。其使用与否会导致如图9-3所示差异。



Nagle算法ON　　Nagle算法OFF

图9-3　Nagle算法

图9-3展示了通过Nagle算法发送字符串“Nagle”和未使用Nagle算法的差别。可以得到如下结论：

“只有收到前一数据的ACK消息时，Nagle算法才发送下一数据。”

TCP套接字默认使用Nagle算法交换数据，因此最大限度地进行缓冲，直到收到ACK。图9-3左侧正是这种情况。为了发送字符串“Nagle”，将其传递到输出缓冲。这时头字符“N”之前没有其他数据（没有需接收的ACK），因此立即传输。之后开始等待字符“N”的ACK消息，等待过程中，剩下的“agle”填入输出缓冲。接下来，收到字符“N”的ACK消息后，将输出缓冲的

“agle”装入一个数据包发送。也就是说，共需传递4个数据包以传输1个字符串。

接下来分析未使用Nagle算法时发送字符串“Nagle”的过程。假设字符“N”到“e”依序传到输出缓冲。此时的发送过程与ACK接收与否无关，因此数据到达输出缓冲后将立即被发送出去。从图9-3右侧可以看到，发送字符串“Nagle”时共需10个数据包。由此可知，不使用Nagle算法将对网络流量（Traffic: 指网络负载或混杂程度）产生负面影响。即使只传输1个字节的数据，其头信息都有可能是几十个字节。因此，为了提高网络传输效率，必须使用Nagle算法。

提 示

图 9-3 是极端情况的演示

在程序中将字符串传给输出缓冲时并不是逐字传递的，故发送字符串“Nagle”的实际情况并非如图 9-3 所示。但如果隔一段时间再把构成字符串的字符传到输出缓冲（如果存在此类数据传递）的话，则有可能产生类似图 9-3 的情况。图 9-3 中就是隔一段时间向输出缓冲传递待发送数据的。

但Nagle算法并不是什么时候都适用。根据传输数据的特性，网络流量未受太大影响时，不使用Nagle算法要比使用它时传输速度快。最典型的是“传输大文件数据”。将文件数据传入输出缓冲不会花太多时间，因此，即便不使用Nagle算法，也会在装满输出缓冲时传输数据包。这不仅不会增加数据包的数量，反而会在无需等待ACK的前提下连续传输，因此可以大大提高传输速度。

一般情况下，不适用Nagle算法可以提高传输速度。但如果无条件放弃使用Nagle算法，就会增加过多的网络流量，反而会影响传输。因此，未准确判断数据特性时不应禁用Nagle算法。

禁用 Nagle 算法

刚才说过的“大文件数据”应禁用Nagle算法。换言之，如果有必要，就应禁用Nagle算法。

“Nagle算法使用与否在网络流量上差别不大，使用Nagle算法的传输速度更慢”

禁用方法非常简单。从下列代码也可看出，只需将套接字可选项TCP_NODELAY改为1（真）即可。

```
int opt_val=1;
setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (void *) &opt_val, sizeof(opt_val));
```

可以通过TCP_NODELAY的值查看Nagle算法的设置状态。

```
int opt_val;
socklen_t opt_len;
```

```
opt_len=sizeof(opt_val);
getsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (void*) &opt_val, &opt_len);
```

如果正在使用Nagle算法，opt_val变量中会保存0；如果已禁用Nagle算法，则保存1。

9.4　基于 Windows 的实现

套接字可选项及其相关内容与操作系统无关，特别是本章的可选项，它们是TCP套接字的相关内容，因此在Windows平台与Linux平台下并无区别。接下来介绍更改和读取可选项的2个函数。

```
#include <winsock2.h>

int getsockopt(SOCKET sock, int level, int optname, char * optval, int * optlen);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

- sock　要查看可选项的套接字句柄。
- level　要查看的可选项协议层。
- optname　要查看的可选项名。
- optval　保存查看结果的缓冲地址值。
- optlen　向第四个参数optval传递的缓冲大小。调用结束后，该变量中保存通过第四个参数返回的可选项字节数。

可以看到，除了optval类型变成char指针外，与Linux中的getsockopt函数相比并无太大区别（Linux中是void型指针）。将Linux中的示例移植到Windows时，应做适当的类型转换。接下来给出setsockopt函数。

```
#include <winsock2.h>

int setsockopt(SOCKET sock, int level, int optname, const char* optval, int
optlen);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

- sock　要更改可选项的套接字句柄。
- level　要更改的可选项协议层。
- optname　要更改的可选项名。
- optval　保存要更改的可选项信息的缓冲地址值。
- optlen　传入第四个参数optval的可选项信息的字节数。

setsockopt函数也与Linux版的毫无二致。各位应更关注可选项的含义而非设置方法。最后，利用上述2个函数编写示例。之前在Linux中验证过套接字I/O缓冲大小，现将其改成基于Windows的实现。

❖ buf_win.c

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

int main(int argc, char *argv[])
{
    WSADATA wsaData;
    SOCKET hSock;
    int sndBuf, rcvBuf, state;
    if(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
        ErrorHandling("WSAStartup() error!");

    hSock=socket(PF_INET, SOCK_STREAM, 0);
    ShowSocketBufSize(hSock);

    sndBuf=1024*3, rcvBuf=1024*3;
    state=setsockopt(hSock, SOL_SOCKET, SO_SNDBUF, (char*)&sndBuf, sizeof(sndBuf));
    if(state==SOCKET_ERROR)
        ErrorHandling("setsockopt() error!");

    state=setsockopt(hSock, SOL_SOCKET, SO_RCVBUF, (char*)&rcvBuf, sizeof(rcvBuf));
    if(state==SOCKET_ERROR)
        ErrorHandling("setsockopt() error!");

    ShowSocketBufSize(hSock);
    closesocket(hSock);
    WSACleanup();
    return 0;
}

void ShowSocketBufSize(SOCKET sock)
{
    int sndBuf, rcvBuf, state, len;

    len=sizeof(sndBuf);
    state=getsockopt(sock, SOL_SOCKET, SO_SNDBUF, (char*)&sndBuf, &len);
    if(state==SOCKET_ERROR)
        ErrorHandling("getsockopt() error");

    len=sizeof(rcvBuf);
    state=getsockopt(sock, SOL_SOCKET, SO_RCVBUF, (char*)&rcvBuf, &len);
    if(state==SOCKET_ERROR)
        ErrorHandling("getsockopt() error");
```

```
47.
48.     printf("Input buffer size: %d \n", rcvBuf);
49.     printf("Output buffer size: %d \n", sndBuf);
50. }
51.
52. void ErrorHandling(char *message)
53. {
54.     fputs(message, stderr);
55.     fputc('\n', stderr);
56.     exit(1);
57. }
```

❖ 运行结果：buf_win.c

```
Input buffer size: 8192
Output buffer size: 8192
Input buffer size: 3072
Output buffer size: 3072
```

系统不同可能导致不同结果。但可以通过上述示例获取系统默认I/O缓冲大小，同时也可以得到更改之后实际使用的缓冲大小。

9.5　习题

(1) 下列关于Time-wait状态的说法错误的是？

a. Time-wait状态只在服务器端的套接字中发生。

b. 断开连接的四次握手过程中，先传输FIN消息的套接字将进入Time-wait状态。

c. Time-wait状态与断开连接的过程无关，而与请求连接过程中SYN消息的传输顺序有关。

d. Time-wait状态通常并非必要，应尽可能通过更改套接字可选项防止其发生。

(2) TCP_NODELAY可选项与Nagle算法有关，可通过它禁用Nagle算法。请问何时应考虑禁用Nagle算法？结合收发数据的特性给出说明。

第 10 章

多进程服务器端

10

大家已对套接字编程有了一定的理解，但要想实现真正的服务器端，只凭这些内容还不够。因此，现在开始学习构建实际网络服务所需内容。

10.1 进程概念及应用

利用之前学习到的内容，我们可以构建按序向第一个客户端到第一百个客户端提供服务的服务器端。当然，第一个客户端不会抱怨服务器端，但如果每个客户端的平均服务时间为0.5秒，则第100个客户端会对服务器端产生相当大的不满。

两种类型的服务器端

如果真正为客户端着想，应提高客户端满意度平均标准。如果有下面这种类型的服务器端，各位应该感到满意了吧。

“第一个连接请求的受理时间为0秒，第50个连接请求的受理时间为50秒，第100个连接请求的受理时间为100秒！但只要受理，服务只需1秒钟。”

如果排在前面的请求数能用一只手数清，客户端当然会对服务器端感到满意。但只要超过这个数，客户端就会开始抱怨。还不如用下面这种方式提供服务。

“所有连接请求的受理时间不超过1秒，但平均服务时间为2~3秒。”

大家无需过多考虑到底哪种服务器端好一些，只需假设收看网络视频课程，而且其顺序是第100位，就能得出结论。因此，接下来讨论如何提高客户端满意度平均标准。

并发服务器端的实现方法

即使有可能延长服务时间，也有必要改进服务器端，使其同时向所有发起请求的客户端提供

服务，以提高平均满意度。而且，网络程序中数据通信时间比CPU运算时间占比更大，因此，向多个客户端提供服务是一种有效利用CPU的方式。接下来讨论同时向多个客户端提供服务的并发服务器端。下面列出的是具有代表性的并发服务器端实现模型和方法。

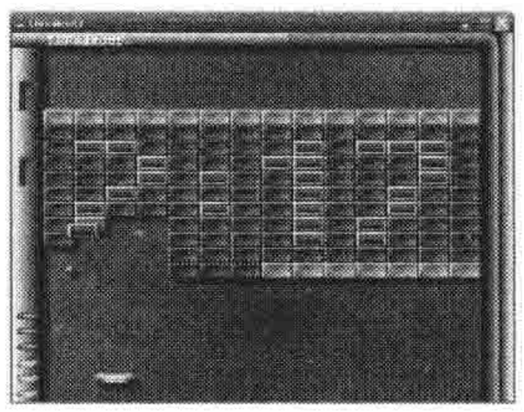
- 多进程服务器：通过创建多个进程提供服务。
- 多路复用服务器：通过捆绑并统一管理I/O对象提供服务。
- 多线程服务器：通过生成与客户端等量的线程提供服务。

先讲解第一种方法：多进程服务器。这种方法不适合在Windows平台下（Windows不支持）讲解，因此将重点放在Linux平台。若各位不太关心基于Linux的实现，可以直接跳到第12章。不过还是希望大家尽可能浏览一下，因为本章内容有助于理解服务器端构建方法。

理解进程（Process）

接下来了解多进程服务器实现的重点内容——进程，其定义如下：

“占用内存空间的正在运行的程序”



假如各位从网上下载了LBreakout游戏并安装到硬盘。此时的游戏并非进程，而是程序。因为游戏并未进入运行状态。下面开始运行程序。此时游戏被加载到主内存并进入运行状态，这时才可称为进程。如果同时运行多个LBreakout程序，则会生成相应数量的进程，也会占用相应进程数的内存空间。

再举个例子。假设各位需要进行文档相关操作，这时应打开文档编辑软件。如果工作的同时还想听音乐，应打开MP3播放器。另外，为了与朋友聊天，再打开MSN软件。此时共创建3个进程。从操作系统的角度看，进程是程序流的基本单位，若创建多个进程，则操作系统将同时运行。有时一个程序运行过程中也会产生多个进程。接下来要创建的多进程服务器就是其中的代表。编写服务器端前，先了解一下通过程序创建进程的方法。

提　示

CPU 核的个数与进程数

拥有 2 个运算设备的 CPU 称作双核（Daul）CPU，拥有 4 个运算器的 CPU 称作 4 核（Quad）CPU。也就是说，1 个 CPU 中可能包含多个运算设备（核）。核的个数与可同时运行的进程数相同。相反，若进程数超过核数，进程将分时使用 CPU 资源。但因为 CPU 运转速度极快，我们会感到所有进程同时运行。当然，核数越多，这种感觉越明显。

进程 ID

讲解创建进程方法前，先简要说明进程ID。无论进程是如何创建的，所有进程都会从操作系统分配到ID。此ID称为“进程ID”，其值为大于2的整数。1要分配给操作系统启动后的（用于协助操作系统）首个进程，因此用户进程无法得到ID值1。接下来观察Linux中正在运行的进程。

❖ 运行结果：ps 命令语句

```
root@my_linux:/tcpip# ps au
USER     PID   %CPU %MEM   VSZ    RSS TTY      STAT  START   TIME  COMMAND
root     4400  0.0  0.1   1780    524 tty4     Ss+   15:57   0:00  /sbin/getty 384
root     4401  0.0  0.1   1780    524 tty5     Ss+   15:57   0:00  /sbin/getty 384
root     4408  0.0  0.1   1780    520 tty2     Ss+   15:57   0:00  /sbin/getty 384
root     4409  0.0  0.1   1780    524 tty3     Ss+   15:57   0:00  /sbin/getty 384
root     4410  0.0  0.1   1780    524 tty6     Ss+   15:57   0:00  /sbin/getty 384
root     5155  2.2  2.9  23728  15136 tty7     Rs+   15:57   0:41  /usr/X11R6/bin/
root     5320  0.0  0.1   1780    528 tty1     Ss+   15:58   0:00  /sbin/getty 384
swyoon   6926  0.2  0.5   5736   3028 pts/0    Ss    16:27   0:00  bash
root     6946  0.0  0.2   4128   1532 pts/0    S     16:28   0:00  su
root     6952  0.0  0.3   4312   1808 pts/0    R     16:28   0:00  bash
swyoon   7006  0.3  0.5   5788   3084 pts/1    Ss+   16:28   0:00  bash
root     7033  0.0  0.1   3136   1008 pts/0    R+    16:28   0:00  ps au
```

可以看出，通过ps指令可以查看当前运行的所有进程。特别需要注意的是，该命令同时列出了PID（进程ID）。另外，上述示例通过指定a和u参数列出了所有进程详细信息。

通过调用 fork 函数创建进程

创建进程的方法很多，此处只介绍用于创建多进程服务器端的fork函数。

```
#include <unistd.h>

pid_t fork(void);
```

➔ 成功时返回进程 ID，失败时返回-1。

fork函数将创建调用的进程副本（概念上略难）。也就是说，并非根据完全不同的程序创建进程，而是复制正在运行的、调用fork函数的进程。另外，两个进程都将执行fork函数调用后的语句（准确地说是在fork函数返回后）。但因为通过同一个进程、复制相同的内存空间，之后的程序

流要根据fork函数的返回值加以区分。即利用fork函数的如下特点区分程序执行流程。

- 父进程：fork函数返回子进程ID。
- 子进程：fork函数返回0。

此处“父进程”（Parent Process）指原进程，即调用fork函数的主体，而“子进程”（Child Process）是通过父进程调用fork函数复制出的进程。接下来讲解调用fork函数后的程序运行流程，如图10-1所示。

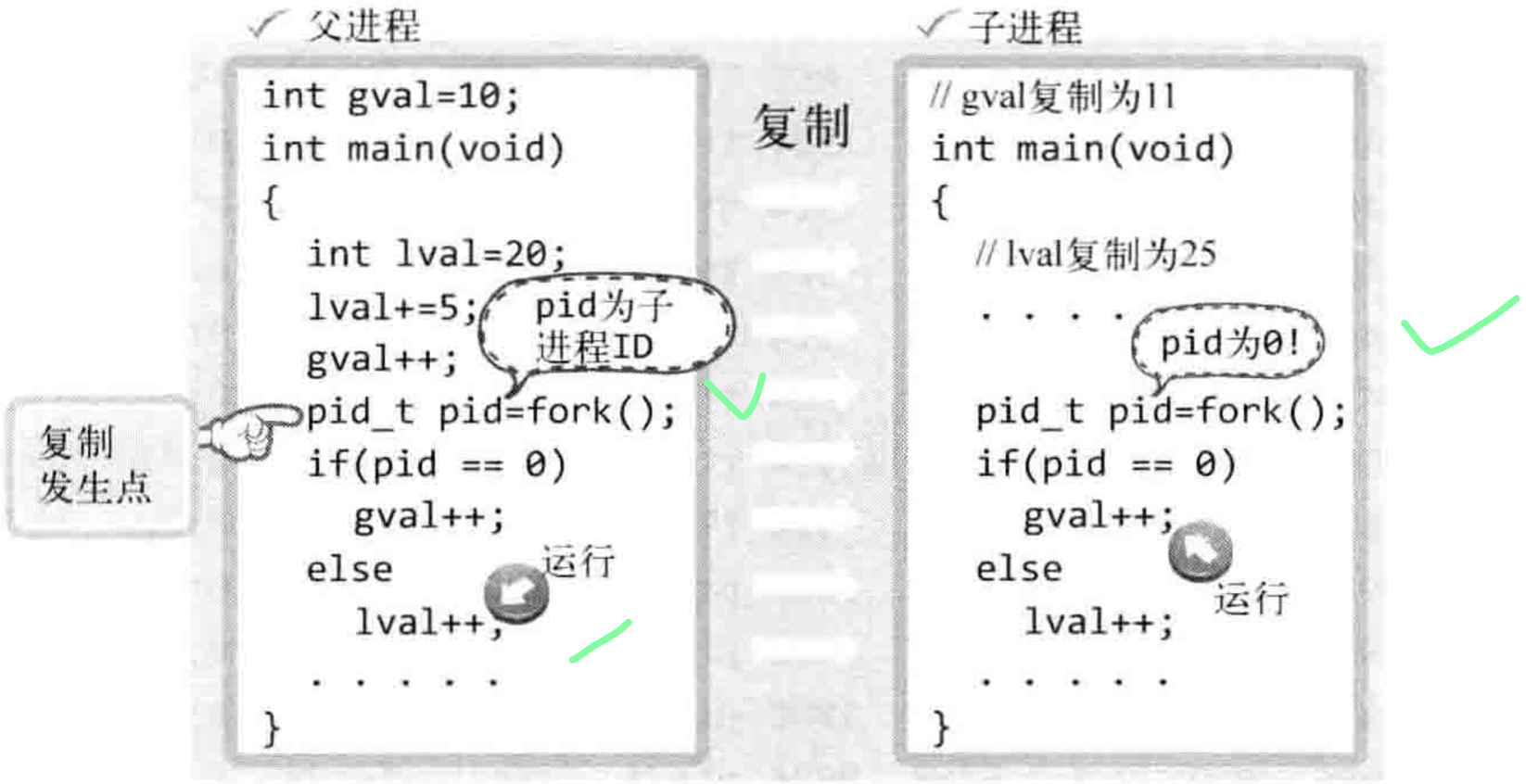


图10-1　fork函数的调用

从图10-1中可以看到，父进程调用fork函数的同时复制出子进程，并分别得到fork函数的返回值。但复制前，父进程将全局变量gval增加到11，将局部变量lval的值增加到25，因此在这种状态下完成进程复制。复制完成后根据fork函数的返回类型区分父子进程。父进程将lval的值加1，但这不会影响子进程的lval值。同样，子进程将gval的值加1也不会影响到父进程的gval。因为fork函数调用后分成了完全不同的进程，只是二者共享同一代码而已。接下来给出示例验证之前的内容。

❖ fork.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.
4.  int gval=10;
5.  int main(int argc, char *argv[])
6.  {
7.      pid_t pid;
8.      int lval=20;
9.      gval++, lval+=5;
10.
11.     pid=fork();
12.     if(pid==0)  // if Child Process
13.         gval+=2, lval+=2;
```