TCP/IP 网络编程

心血来潮想看看unp这本书，玛德看了几页想睡觉。看unp感觉有点难度，就复习一手入门的书吧。。。想在这做一个总的目录，但是一个一个的写太麻烦了，直接cv各章的md文件到这来吧。

ch01 理解网络编程和套接字

1. 理解网络编程和套接字

网络编程中接受请求的套接字的创建过程如下。

1. 调用 `socket` 函数创建套接字。

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
// Create a new socket of type TYPE in domain DOMAIN, using
// protocol PROTOCOL. If PROTOCOL is zero, one is chosen automatically.
// Returns a file descriptor for the new socket, or -1 for errors.
```

2. 调用 `bind` 函数分配IP地址和端口号。

```
#include <sys/socket.h>
int bind(int __fd, const struct sockaddr *__addr, socklen_t __len);
// Give the socket FD the local address ADDR (which is LEN bytes long).
// 成功时返回0，失败时返回-1。
```

3. 调用 `listen` 函数转为可接受请求的状态。

```
#include <sys/socket.h>
int listen(int __fd, int __n);
// Prepare to accept connections on socket FD.
// N connection requests will be queued before further requests are refused.
// Returns 0 on success, -1 for errors.
```

4. 调用 `accept` 函数受理连接请求。

```
#include <sys/socket.h>
int accept(int __fd, struct sockaddr *__restrict__ __addr, socklen_t
*__restrict__ __addr_len);
// Await a connection on socket FD.
// When a connection arrives, open a new socket to communicate with it,
// set *ADDR (which is *ADDR_LEN bytes long) to the address of the connecting
// peer and *ADDR_LEN to the address's actual length, and return the
// new socket's descriptor, or -1 for errors.

// This function is a cancellation point and therefore not marked with
// __THROW.
```

服务器程序：hello_server.c

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请求" 这两个步骤。

```
int connect(int __fd, const struct sockaddr *__addr, socklen_t __len)
// Open a connection on socket FD to peer at ADDR (which LEN bytes long).
// For connectionless socket types, just set the default address to send to
// and the only address from which to accept transmissions.
// Return 0 on success, -1 for errors.

// This function is a cancellation point and therefore not marked with
// __THROW.
```

客户端程序：hello_client.c

2. 基于Linux的文件操作

在Linux世界，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。关于标准输入输出，可参考《Linux命令行与shell脚本编程大全》笔记。

1. 打开文件

```
int open(const char *__file, int __oflag, ...)
// 第一个参数是文件名和路径信息，第二个参数是文件打开模式
// 成功时返回文件描述符，失败时返回-1
// 详见手册
```

low_open.c

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

2. 关闭文件

```
#include <unistd.h>
int close(int fd);
// 成功时返回0，失败时返回-1
```

3. 将数据写入文件

```
#include <unistd.h>
ssize_t write(int __fd, const void *__buf, size_t __n)
// Write N bytes of BUF to FD. Return the number written, or -1.

// This function is a cancellation point and therefore not marked with
// __THROW.
```

> **知识补给站：** `size_t` 是通过 `typedef` 定义的 `unsigned int` 类型。`ssize_t` 前面多加的s代表signed，即 `ssize_t` 是通过 `typedef` 定义的 `signed int` 类型。这些类型都是基本数据类型的别名。
> 人们普遍认为int是32位的，因为主流的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是16位的。根据系统的不同、时代的变化，数据类型的表现也随之改变，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`，则将大大减少代码的改动，因为只需要修改并编译 `size_t` 和 `ssize_t` 类型的 `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续。

4. 读取文件中的数据

```
#include <unistd.h>
ssize_t read(int __fd, void *__buf, size_t __nbytes)
// Read NBYTES into BUF from FD. Return the
// number read, -1 for errors or 0 for EOF.

// This function is a cancellation point and therefore not marked with
// __THROW.
```

low_read.c

5. 文件描述符与套接字

fd_seric.c

```
lxc@Lxc:~/C/tcpip_src/ch01-理解网络编程和套接字$ bin/fd_seri
file descriptor 1: 3
file descriptor 2: 4
file descriptor 3: 5
```

从输出的文件描述符数值可以看出，描述符从3开始由小到大的顺序编号，因为0、1、2是分配给标准I/O的描述符。

ch02 套接字类型与协议设置

1. 套接字协议及其数据传输特性

1. 创建套接字

```
int socket(int domain, int type, int protocol)
// Create a new socket of type TYPE in domain DOMAIN, using
// protocol PROTOCOL. If PROTOCOL is zero, one is chosen automatically.
// Returns a file descriptor for the new socket, or -1 for errors.
```

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*： 计算机间通信使用的协议信息。

2. 协议族(Protocol Family)

协议族主要有以下几类。通过 `socket` 函数的第一个参数传递。

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

3. 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。

1. 面向连接的套接字(SOCK_STREAM)

可靠、有序、基于字节的面向连接的数据传输方式的套接字。套接字必须一一对应。这种套接字称为TCP套接字。

2. 面向消息的套接字(SOCK_DGRAM)

不可靠、无序、以数据的高速传输为目的的套接字。这种套接字成为UDP套接字。

3. 协议的最终选择

`socket` 函数的第三个参数决定最终采用的协议。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，除非在同一协议族中存在多个数据传输方式相同的协议。这个时候，数据传输方式相同，协议不同，此时需要第三个参数具体指定协议信息。

3. TCP套接字示例

tcp_client.c。该示例验证TCP套接字具有以下特性：传输的数据不存在边界。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_server.c 与第一章的hello_server.c相比无变化。

ch03 地址族与数据序列

1. 分配给套接字的IP地址与端口号

1. 网络地址

IP地址分两类：

- IPv4，4字节地址族。
- IPv6，16字节地址族。

2. 网络地址分类与主机地址边界

- A类地址的首字节范围：0~127，即首位以0开始。
- B类地址的首字节范围：128~191，即前2位以10开始。
- C类地址的首字节范围：192~223，即前3位以110开始。

CIDR了解一下?

3. 用于区分套接字的端口号

端口号由16位构成，可分配的端口号范围为0~65535。虽然端口号不能重复，但TCP套接字和UDP套接字不会共用端口号，所以允许重复。例如，某TCP套接字使用8888端口号，则其它TCP套接字就无法使用该端口号，但UDP套接字可以使用。

2. 地址信息的表示

1. 表示IPv4地址的结构体

```
struct sockaddr_in
{
    sa_family_t sin_family; // 地址族(Address Family)
    uint16_t sin_port; // 16位TCP/UDP端口号
    struct in_addr sin_addr; // 32位IP地址
    char sin_zero[8]; // 不使用
}
// 该结构体中提到的另一个结构体 in_addr 定义如下，它用来存放32位IP地址。
struct in_addr
{
    in_addr_t s_addr; // 32 位IPv4地址。
}
```

数据类型参考如下POSIX定义表。

数据类型名称	数据类型说明	声明的头文件
int8_t	signed 8 bit int	sys/types.h
uint8_t	unsigned 8 bit int(char)	sys/types.h
int16_t	signed 16 bit int	sys/types.h
uint16_t	unsigned 16 bit int(unsigned short)	sys/types.h
int32_t	signed 32 bit int	sys/types.h
uint32_t	unsigned 32 bit int(unsigned long)	sys/types.h
sa_family_t	地址族(Address Family)	sys/socket.h
socklen_t	长度(length of struct)	sys/socket.h
in_addr_t	IP地址，声明为uint32_t	netinet/in.h
in_port_t	端口号，声明为uint16_t	netinet/in.h

2. 结构体 `sockaddr_in` 的成员分析

1. 成员 `sin_family`

每种协议适用的地址族均不同。比如，IPv4使用4字节地址族，IPv6使用16字节地址族。

地址族	含义
AF_INET	IPv4网络协议中使用的地址族
AF_INET6	IPv6网络协议中使用的地址族
AF_LOCAL	本地通信中采用的UNIX协议的地址族

2. 成员 `sin_port`

以网络字节序（大端）保存16位端口号。

3. 成员 `sin_addr`

以网络字节序（大端）保存32位IP地址。结构体 `in_addr` 声明为uint32_t，因此只需当作32位整数即可。

4. 成员 `sin_zero`

无特殊含义，只是为了使结构体 `sockaddr_in` 的大小与 `sockaddr` 结构体保持一致而插入的成员。必须填充为0，否则无法得到想要的结果。参考如下 `bind` 函数，重点关注参数传递和类型转换部分。

```
if(bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == -1)
        error_handing("bind() error");
```

此处重要的是第二个参数的传递。实际上 `bind` 函数的第二个参数期望得到 `sockaddr` 结构体变量的地址值，包括地址族、端口号、IP地址等。从下列代码可以看出，直接向 `sockaddr` 结构体填充这些信息会很麻烦。

```
struct sockaddr
{
    sa_family_t sin_family; // 地址族（Address Family）
    char sa_data[14]; // 地址信息
}
```

此结构体成员 `sa_data` 保存的地址信息中需包含IP地址和端口号，剩余部分应填充为0，这也是 `bind` 函数要求的。而这对于包含地址信息来讲非常麻烦，继而就有了新的结构体 `sockaddr_in`。若按照之前的讲解填充 `sockaddr_in` 结构体，则将生成符合 `bind` 函数要求的字节流。最后转换为 `sockaddr` 型的结构体变量，再传递给 `bind` 函数即可。

> **知识补给站：** `sockaddr_in` 是保存IPv4地址信息的结构体，那为何还要通过 `sin_family` 来单独指定地址族信息呢？这与 `sockaddr` 结构体有关。结构体 `sockaddr` 并非只为 IPv4 设计，这从保存地址信息的数组 `sa_data` 长度为14字节也可看出。因此，结构体 `sockaddr` 要求在 `sin_family` 中指定地址族信息。为了与 `sockaddr` 保持一致，`sockaddr_in` 结构体中也有地址族信息。

3. 网络字节序与地址变换

1. 字节序与网络字节序

- 大端序（Big Endian）（网络字节序为大端序）：高位字节存放在低位地址。
- 小端序（Little Endian）：高位字节存放在高位地址。

2. 字节序转换（Endian Conversations）

- htons
- ntohs
- htonl
- ntohl

其中，h代表主机（host）字节序，n代表网络（network）字节序。s 指的是short(unsigned short)，l 指的是long(unsigned long，Linux中long占4个字节)。比如，`htons` 是把short型数据从主机字节序转换为网络字节序。
以s作为后缀的函数中，s代表两个字节short，因此用于端口号转换；以l作为后缀的函数中，l代表4个字节，因此用于IP地址转换。

endian_conv.c

```
lxc@Lxc:~/C/tcpip_src/ch03-地址族与数据序列$ bin/endian_conv
Host Byte Order Port: 0x1234
NewWork Byte Order Port: 0x3412
Host Byte Order Address: 0x12345678
NewWork Byte Order Address: 0x78563412
```

这是在小端字节序CPU中运行的结果。Intel和AMD系列的CPU都采用小端序标准。

> **知识补给站：** 数据在传输之前都要经过转换吗？数据收发过程中有自动转换机制。除了向 `sockaddr_in` 结构体变量填充数据以外，其他情况无须考虑字节序问题。

4. 网络地址的初始化与分配

1. 将字符串信息转换为网络字节序的整数型

对于IP地址的表示，我们熟悉的是点分十进制表示法（Dotted Decimal Notation），而非整数型数据表示法。幸运的是，有个函数会帮我们将字符串形式的IP地址转换为32位整数型数据。此函数在转换类型的同时还进行网络字节序的转化。

```
#include <arpa/inet.h>
in_addr_t inet_addr(const char* string);
// 成功时返回32位大端序整数型值，失败时返回 INADDR_NONE。
// 详见手册
```

inet_add.c

```
lxc@Lxc:~/C/tcpip_src/ch03-地址族与数据序列$ bin/inet_addr
NewWork Byte Order Integer Address: 0x6fc6a87f
Error Occured!
```

从运行结果可以看出，`inet_addr` 函数不仅可以把IP地址转成32位整数型，而且可以检测无效的IP地址。另外，从输出结果可以验证确实转换为网络字节序。

`inet_aton` 函数与 `inet_addr` 函数在功能上完全相同，也将字符串形式IP地址转换为32位网络字节序整数并返回。只不过该函数利用了 `in_addr` 结构体，且其使用频率更高。

```
#include <arpa/inet.h>
int inet_aton(const char* string, struct in_addr* addr)
// 成功时返回1（true），失败时返回0（false）。
```

其中，*string* 参数为需要转换的IP地址信息的字符串地址值。*addr* 将保存转换结果的 `in_addr` 结构体变量的地址值。在使用 `inet_addr` 函数时，需将转换后的IP地址信息代入 `sockaddr_in` 结构体变量中声明的 `in_addr` 结构体变量。而 `inet_aton` 函数则不需要此过程。原因在于，若传递 `in_addr` 结构体变量地址值，函数会自动把结果填入该结构体变量。

inet_aton.c

```
lxc@Lxc:~/C/tcpip_src/ch03-地址族与数据序列$ bin/inet_aton
Newwork ordered integer addr: 0xdeea7c7b
```

2. 将网络字节序的整数型转换为字符串

`inet_ntoa` 看到名字你应该就知道这个函数是干嘛的了，此函数将网络字节序的整数型转换为字符串。

```
#include <arpa/inet.h>
char *inet_ntoa(struct in_addr adr)
// 成功时返回字符串地址，失败时返回-1
```

在调用该函数时要小心，返回值为char类型的指针。返回字符串地址意味着字符串已保存到内存空间，但该函数未向程序员要求分配内存，而是在内部申请了内存并保存了字符串。也就是说，调用完该函数后，应立即将字符串信息复制到其他内存空间。因为，若再次调用该函数，则会覆盖之前保存的字符串信息。

inet_ntoa.c

```
lxc@Lxc:~/C/tcpip_src/ch03-地址族与数据序列$ bin/inet_ntoa
Dotted-Decimal notation1: 1.2.3.4
Dotted-Decimal notation2: 1.1.1.1
Dotted-Decimal notation3: 1.2.3.4
```

3. 网络地址初始化

结合前面所学的内容，现在介绍套接字创建过程中常见的网络地址信息初始化的方法。

```
struct sockaddr_in addr;
char *serv_ip = "123.234.123.111" // 声明IP地址字符串
char *serv_port = "9999" // 声明端口号字符串
memset(&addr, 0, sizeof(addr)); // 结构体变量 addr 的所有成员赋值为0
addr.sin_family = AF_INET; // 指定地址族
addr.sin_addr.s_addr = inet_addr(serv_ip); // 基于字符串的IP地址初始化
addr.sin_port = htons(atoi(serv_port)); // 基于字符串的端口号初始化
```

上述代码中 `memset` 将每个字节都初始化为一个值，第一个参数为结构体变量 `addr` 的地址值，即初始化对象为 `addr`；第二个参数为0，因此初始化为0；最后一个参数传入 `addr` 的长度，因此 `addr` 的所有字节均初始化为0。这么做是为了将 `sockaddr_in` 结构体成员 `sin_zero` 初始化为0。另外，最后一行代码调用的 `atoi` 函数把字符串类型的值转换为整数型。总之，上述代码利用字符串格式的IP地址和端口号初始化了 `sockaddr_in` 结构体。

4. 客户端地址信息初始化

服务器端的准备工作通过 `bind` 函数完成，而客户端则通过 `connect` 函数完成。服务器端声明 `sockadr_in` 结构体变量，将其赋予服务器端IP和套接字的端口号，然后调用 `bind` 函数；而客户端则声明 `sockaddr_in` 结构体，并初始化为要与之连接的服务器端套接字的IP和端口号，然后调用 `connect` 函数。

5. `INADDR_ANY`

每次创建服务器端套接字时都输入IP地址有些繁琐，此时可如下初始化地址信息。

```
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

利用常数 `INADDR_ANY` 分配服务器端的IP地址。若采用这种方法，则可自动获取运行服务器端的IP地址，不必亲自输入。而且，若同一计算机中已分配多个IP地址（多宿主(Multi-homed)计算机，一般路由器属于这一类），则只要端口号一致，就可以从不同IP地址接收数据信息。因此，服务器端优先考虑这种方式。而客户端中除非带有一部分服务器端功能，否则不会采用。

ch04 基于TCP的服务器端/客户端

1. 理解TCP和UDP

略。。。

2. 实现基于TCP的服务器端和客户端

1. TCP服务器端的默认函数调用顺序

1. `socket` 创建套接字
2. `bind` 分配套接字地址
3. `listen` 等待连接请求状态
4. `accept` 允许连接
5. `read/write` 数据交换
6. `close` 断开连接

2. 进入等待请求连接状态

我们已经调用 `bind` 函数给套接字分配了地址，接下来就要通过调用 `listen` 函数进入等待连接请求状态。只有调用了 `listen` 函数，客户端才能进入可发出连接请求的状态。换言之，这时客户端才能调用 `connect` 函数（若提前调用将发生错误）。

```c
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
int listen(int sockfd, int backlog);
// 成功时返回0，失败时返回-1
// 详见手册
```

sockfd 为希望进入等待连接请求状态的套接字文件描述符，传递的描述符套接字参数成为服务器端套接字（监听套接字）。*backlog* 连接请求等待队列的长度，若为5，则队列长度为5，表示最多使5个连接请求进入队列。

3. 受理客户端连接请求

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

sock 服务器端套接字的文件描述符，*addr* 保存发起连接请求的客户端地址信息的变量地址值，调用函数后向传递来的地址变量参数填充客户端地址信息。*addrlen* 第二个参数 *addr* 结构体的长度，但是存有长度的变量地址值。函数调用完成后，该变量即被填入客户端地址长度。
`accept` 函数受理连接请求等待队列中待处理的客户端连接请求。函数调用成功时，`accept` 函数内部将产生用于数据I/O的套接字，并返回其文件描述符。需要强调的是，套接字是自动创建的，并自动与发起连接请求的客户端建立连接。

回顾 Hello World服务器端：
hello_server.c

4. TCP客户端的默认函数调用顺序

1. `socket` 创建套接字
2. `connect` 请求连接
3. `read/write` 交换数据
4. `close` 断开连接

与服务器端相比，区别就在于 "请求连接"，它是创建客户端套接字后向服务器端发起的连接请求。服务器端调用 `listen` 函数后创建连接请求队列，之后客户端即可请求连接。

```c
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr,
                socklen_t addrlen);
// 成功时返回0，失败时返回-1
// 详见手册
```

sock 客户端套接字文件描述符，*servaddr* 保存目标服务器地址信息的变量地址值，*addrlen* 以字节为单位传递第二个结构体参数 *servaddr* 的地址变量长度。

客户端调用 `connect` 函数后，发生以下情况之一才会返回（完成函数调用）。

- 服务器端接收连接请求。
- 发生断网等异常情况而中断连接请求。

需要注意，所谓的 "接收连接" 并不意味着服务器端调用 `accept` 函数，其实是服务器端把连接请求信息记录到等待队列。因此，`connect` 函数返回后并不立即进行数据交换。

> **知识补给站：** 客户端套接字何时、何地、如何分配地址呢？

- 何时？ 调用 `connect` 函数时
- 何地？ 操作系统内核
- 如何？ IP用计算机（主机）的IP，端口随机

客户端的IP地址和端口号在调用 `connect` 函数时自动分配，无需调用标记的 `bind` 函数进行分配。

回顾Hello World客户端：hello_client.c

服务器端创建套接字后连续调用 `bind`、`listen` 函数进入等待状态，客户端通过调用 `connect` 函数发起连接请求。需要注意的是，客户端只能等到服务器端调用 `listen` 后才能调用 `connect` 函数。同时要清楚，客户端调用 `connect` 函数前，服务器端有可能率先调用 `accept` 函数，当然，此时服务器端在调用 `accept` 函数时进入阻塞状态，直到客户端调用 `connect` 函数为止。

3. 实现迭代服务器端/客户端

本节编写回声服务器端/客户端。

1. 迭代回声服务器端/客户端

基本运作方式如下：

- 服务器端在同一时刻只与一个客户端相连，并提供回声服务。
- 服务器端依次向5个客户端提供服务并退出。
- 客户端接收用户输入的字符串并发送到服务器端。
- 服务器端将接收到的字符串数据传回客户端，即回声。
- 客户端输入q退出。

echo_server.c
echo_client.c

2. 回声客户端存在的问题（TCP不存在数据边界）

```
write(sock, message, strlen(message));
str_len = read(sock, message, BUF_SIZE);
```

以上代码有个错误假设，"每次调用 `read`、`write` 函数时都会以字符串为单位执行实际的I/O操作"。当然，每次调用 `write` 函数都会传递一个字符串，因此这种假设在某种程度上也算合理。但是 "TCP不存在数据边界"（参见第二章）。上述客户端是基于TCP的，因此，多次调用 `write` 函数传递的字符串有可能一次性传递到服务器端。此时客户端客户端有可能从服务器端收到多个字符串，这不是我们希望看到的结果。还需考虑服务器端的以下情况："字符串太长，需要分2个数据包发送"。服务器端希望通过一次 `write` 函数调用传输数据，但如果数据太大，操作系统有可能把数据分成多个数据包发送到客户端。另外，在此过程中，客户端有可能在尚未收到全部数据包时就调用 `read` 函数。所有的这些问题都源自TCP的数据传输特性。那该如何解决呢？参见第5章。

ch05 基于TCP的服务器端/客户端（2）

1. 回声客户端的完美实现

1. 回声服务器端没有问题，只有回声客户端有问题?

问题不在服务器端，而在客户端。先回顾一下回声服务器端的I/O相关代码。

```
while ((str_len = read(clnt_sock, message, BUF_SIZE)) != 0)
    write(clnt_sock, message, str_len);
```

接着回顾回声客户端的代码。

```
while (1)
{
    fputs("Input Message(q/Q to quit): ", stdout);
    fgets(message, BUF_SIZE, stdin);
    if(! strcmp(message, "q\n") || ! strcmp(message, "Q\n"))
        break;
    write(sock, message, strlen(message));
    str_len = read(sock, message, BUF_SIZE);
    // message [str_len] = 0;
    printf("Message from server: %s\n", message);
}
```

二者都在循环调用 `read` 或 `write` 函数。实际上回声客户端会100%接收自己传输的数据，只不过接收数据时的单位有些问题。回声客户端传输的是字符串，而且是通过调用 `write` 函数一次性发送的，之后还调用一次 `read` 函数，期待着接收自己传输的数据。这就是问题所在。

> "既然回声客户端会收到所有字符串数据，是否只需多等待一会儿？过一段时间后再调用 `read` 函数是否可以一次性读取所有的字符串数据？"

的确，过一段时间后即可接收，但需等多久？要等10分钟嘛？这不现实，理想的客户端应在收到字符串数据时立即读取并输出。

2. 回声客户端问题解决方法

解决 第4章的问题。

解法一： 因为回声客户端可以提前确定接收数据的大小，那我们就可以计算我们已经接收的数据直到达到预期即可。

echo_client2.c

```
str_len = write(sock, message, strlen(message));
recv_tot = 0;
while (recv_tot < str_len)
{
    recv_cur = read(sock, message, BUF_SIZE - 1);
    if(recv_cur == -1)
        error_handling("write() error");
    recv_tot + = recv_cur;
}
message [str_len] = 0;
printf("Message from server: %s\n", message);
```

3. 如果问题不在于回声客户端：定义应用层协议

回声客户端可以提前知道接收的数据长度，但我们应该认识到，更多情况下这不太可能。既然如此，若无法预知接收数据长度时应如何收发数据？此时需要的就是应用层协议的定义。之前的回声服务器/客户端中定义了如下协议。"收到Q就立即终止连接" 同样，收发数据过程中也需要定好规则（协议）以表示数据的边界，或提前告知收发数据的大小。服务器端/客户端实现过程中逐步定义的这些规则集合就是应用层协议。

下面编写程序以体验应用层协议的定义过程。该程序中，服务器端从客户端获得多个数字和运算符信息。服务器端收到数字后对其进行加减乘除运算，然后把结果传回客户端。例如，向服务器传递3、5、9的同时请求加法运算，则客户端收到3+5+9的运算结果。

该程序设计的大致协议如下：

- 客户端连接到服务器端后以1字节整数形式传递待算数字个数。
- 客户端向服务器端传递的每个整型数据占用4个字节。
- 传递整数型数据后接着传递运算符。运算符信息占用1字节。
- 选择字符+、-、*之一传递。
- 服务器端以4字节整数型向客户端传回运算结果。
- 客户端得到运算结果后终止与服务器端的连接。

这种程度的协议相当于实现了一半程序。

op_client.c op_server.c

2. TCP原理

1. TCP套接字中的I/O缓冲

如前所述，TCP套接字的数据收发无边界。服务器端即使调用1次 `write` 函数传输40字节的数据，客户端也有可能通过4次 `read` 函数调用每次读取10字节。服务器端一次性传输了40字节，而客户端可以分批次接收这是因为缓冲的存在。

实际上， `write` 函数调用后并非立即传输数据， `read` 函数调用后也并非马上接收数据。 `write` 函数调用的瞬间，数据将移至输出缓冲； `read` 函数调用瞬间，从输入缓冲读取数据。
这些I/O缓冲特性如下：

- I/O缓冲在每个套接字中单独存在
- I/O缓冲在创建套接字时自动生成

- 即使关闭套接字也会继续传递输出缓冲中遗留的数据
- 关闭套接字将丢失输入缓冲中的数据

ch06 基于UDP的服务器端/客户端

1. 理解UDP

...略

2. 实现基于UDP的服务器端/客户端

1. UDP中的服务器端和客户端没有连接

UDP服务器端/客户端不像TCP那样在连接状态下交换数据，因此与TCP不同，无需经过连接过程。也就是说，不必调用TCP连接过程中调用的 `listen` 函数和 `accept` 函数。UDP中只有创建套接字的过程和数据交换过程。

2. UDP服务器端和客户端均只需一个套接字

TCP中套接字之间应该是一对一的关系。若要向10个客户端提供服务，则除了守门的服务器套接字外，还需要10个服务器端套接字。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。也就是说，只需要1个UDP套接字就能和多台主机通信。

3. 基于UDP的数据I/O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，因此，每次传输数据都要添加目标地址信息。

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
// 成功时返回传输的字节数，失败时返回-1。
// 详见手册
```

- *sockfd*： 用于传输数据的UDP套接字文件描述符；
- *buf*： 保存待传输数据的缓冲地址值；
- *len*： 待传输的数据长度，以字节为单位；
- *flags*： 可选项参数，若没有则传递0；
- *dest_addr*： 存有目标地址信息的 `sockaddr` 结构体变量的地址值；
- *addrlen*： 传递给参数 *dest_addr* 的结构体变量的长度。

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
// 成功时返回接收的字节数，失败时返回-1
// 详见手册
```

- *sockfd*： 用于接收数据的UDP套接字文件描述符；
- *buf*： 保存接收数据的缓冲地址值；
- *len*： 可接收的最大字节数，故无法超过参数 *buff* 所指的缓冲大小；
- *flags*： 可选项参数，若没有则传0；
- *src_addr*： 存有发送端地址信息的 `sockaddr` 结构体变量的地址值；
- *addrlen*： 保存参数 *src_addr* 的结构体变量长度的变量地址值；

4. 基于UDP的回声服务器端/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。只是因其提供服务而称为服务器端。

uecho_server.c uecho_client.c

5. UDP客户端套接字的地址分配

> TCP客户端调用 `connect` 函数完成IP和端口号的分配，UDP客户端在何时分配IP和端口号呢？

UDP程序中，调用 `sendto` 函数传输数据前应完成对套接字的地址分配工作，因此调用 `bind` 函数。当然， `bind` 函数在TCP程序中出现过，但 `bind` 函数不区分TCP和UDP，也就是说，在UDP程序中同样可以调用。另外，如果调用 `sendto` 函数时发现尚未分配地址信息，则在首次调用 `sendto` 函数时给相应套接字自动分配IP和端口。而且此时分配的地址一直保留到程序结束为止，因此也可用来与其他UDP套接字进行数据交换。当然，IP用主机IP，端口号选尚未使用的任意端口号。

综上所述，调用 `sendto` 函数时自动分配IP和端口号，因此，UDP客户端通常无需额外的地址分配过程。所以之前的示例省略了该过程，这也是普遍的实现方式。

3. UDP的数据传输特性和调用 `connect` 函数

TCP传输的数据不存在数据边界，UDP数据传输中存在数据边界。

1. 存在数据边界的UDP套接字

TCP传输的数据不存在数据边界，这表示 **“数据传输过程中调用I/O函数的次数不具有任何意义。”** UDP是具有数据边界的协议，传输中调用I/O函数的次数非常重要。因此，**输入函数的调用次数和输出函数的调用次数完全一致，这样才能保证接收全部已发送数据**。

bound_host1.c bound_host2.c

2. 已连接（connected）UDP套接字与未连接（unconnected）UDP套接字

TCP套接字中需注册待传输数据的目标IP和端口号，而UDP中则无需注册。因此，通过 `sendto` 函数传输数据的过程大致可分为以下3个阶段。

1. 向UDP套接字注册目标IP和端口号
2. 传输数据
3. 删除UDP套接字中注册的目标地址信息

每次都变更目标地址，因此可以重复利用同一UDP套接字向不同目标传输数据。这种未注册目标地址信息的套接字称为未连接套接字，反之，注册了目标地址的套接字称为已连接套接字。显然，UDP套接字属于未连接套接字。

在与同一主机进行长时间通信时，将UDP套接字变成已连接套接字会提高效率。上述三个阶段中，第一个阶段和第三个阶段占整个通信过程近1/3的时间，缩短这部分时间将大大提高性能。

3. 创建已连接UDP套接字

创建已连接UDP套接字只需针对UDP套接字调用 `connect` 函数即可。

```
sock = socket(PF_INET, SOCK_DGRAM, 0);
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = ....
addr.sin_port = ....
connect(sock, (struct sockaddr*)&addr, sizeof(addr));
```

针对UDP套接字调用 `connect` 函数并不意味着要与对方UDP套接字连接，这只是向UDP套接字注册目标IP和端口信息。之后就与TCP套接字一样了，每次调用 `sendto` 函数时只需传输数据。因为已经指定了收发对象，所以不仅可以使用 `sendto`、`recvfrom` 函数，还可以使用 `write`、`read` 函数进行通信。
下面示例改自 `uecho_client.c`，可以结合 `uecho_server.c` 程序运行。

uecho_con_client.c

ch07 优雅地断开套接字连接

1. 基于TCP的半关闭

1. 单方面断开连接带来的问题

Linux的 `close` 函数意味着完全断开连接。完全断开连接不仅指无法传输数据，而且也不能接收数据。

2. 套接字和流

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。在套接字的流中，数据只能向一个方向流动。因此，为了进行双向通信，需要如下图的2个流。

一旦两台主机建立了套接字连接，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和另一个主机的输出流相连，而输出流和另一个主机的输入流相连。本章所谓的优雅地断开，只断开其中的一个流，而非同时断开两个流。

3. 针对优雅断开的 `shutdown` 函数

```
NAME
    shutdown - shut down part of a full-duplex connection
SYNOPSIS
    #include <sys/socket.h>
    int shutdown(int sockfd, int how);
// 成功时返回0，失败时返回-1
// 详见手册
```

- *sockfd*： 需要断开的套接字文件描述符

- *how*： 传递断开方式信息

第二个参数决定断开连接的方式，有以下取值：

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

SHUT_RD，断开输入流，套接字无法接收数据，输入缓冲中收到的数据也会被抹除，而且无法调用输入相关函数。
SHUT_WR，断开输出流，无法传输数据。但如果输出缓冲中还有未传输的数据，则传递给目标主机。
SHUT_RDWR，中断I/O流，相当于分两次分别调用上面两个函数。

4. 为何需要半关闭

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。可以接收对方的数据。

5. 基于半关闭的文件传输程序

file_server.c file_client.c

ch08 域名及网络地址

1. 域名系统

略

2. IP地址和域名之间的转换

1. 程序中有必要使用域名吗?

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。一旦注册域名可能永久不变，因此利用域名编写程序会更好一点。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。

2. 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 *name);
// 成功时返回 hostent 结构体地址，失败时返回NULL指针
```

`hostent` 结构体定义如下：

```c
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
}
```

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，除官方域名外还可以指定其他域名。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，则保存4。IPv6时，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。

gethostbyname.c

```
lxc@Lxc:~/C/tcpip_src/ch08-域名及网络地址$ bin/gethostbyname www.baidu.com
official name: www.a.shifen.com
Aliases 1: www.baidu.com
Address type: AF_INET
IP addr 1: 39.156.66.14
IP addr 2: 39.156.66.18
```

3. 利用IP地址获取域名

`gethostbyname` 利用IP地址获取域相关信息。

```c
SYNOPSIS
#include <sys/socket.h>       /* for AF_INET */
struct hostent *gethostbyaddr(const void *addr,
                              socklen_t len, int type);
// 成功时返回 hostent 结构体变量地址，失败时返回NULL指针。
```

- *addr* ： 含有IP地址信息的 in_addr 结构体（参见第三章）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。

gethostbyaddr.c

ch09 套接字的多种选项

1. 套接字可选项和I/O缓冲大小

1. 套接字的多种可选项

我们之前写的程序都是创建好套接字后直接使用的，此时通过默认的套接字特性进行数据通信。

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
IPPROTO_TCP	TCP_KEEPALIVE	O	O
	TCP_NODELAY	O	O
	TCP_MAXSEG	O	O

从上表可以看出，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。

2. `getsockopt` & `setsockopt`

可选项的读取和设置可以通过如下两个函数完成。

```
SYNOPSIS
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
int getsockopt(int sockfd, int level, int optname,
               void *optval, socklen_t *optlen);
// 成功时返回0，失败时返回-1。
```

- *sock*： 用于待查看选项的套接字文件描述符。
- *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。

下面是更改可选项的函数。

```
SYNOPSIS
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
int setsockopt(int sockfd, int level, int optname,
                       const void *optval, socklen_t optlen);
// 成功时返回0，失败时返回-1。
```

- *sock*： 用于更改可选项的文件描述符
- *level*： 要更该的可选项协议层
- *optname*： 要更该的可选项名
- *optval*： 保存要更该的选项信息的缓冲地址值
- *optlen*： 向第四个参数传递的可选项信息的字节数

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）。关于 `setsockopt` 函数的调用方法会在其他示例中给出。

sock_type.c

```
lxc@Lxc:~/C/tcpip_src/ch09-套接字的多种可选项$ bin/sock_type
SOCK_STREAM: 1
SOCK_DGRAM: 2
Socket type one: 1
Socket type two: 2
```

3. `SO_SNDBUF` & `SO_RCVBUF`

`SO_SNDBUF` 是输出缓冲大小相关可选项，`SO_RCVBUF` 是输入缓冲大小相关可选项。用这两个可选项既可以读取当前I/O缓冲大小，也可以进行更改。

get_buf.c

```
lxc@Lxc:~/C/tcpip_src/ch09-套接字的多种可选项$ bin/get_buf
Input buffer size: 131072
Output buffer size: 16384
```

set_buf.c

```
lxc@Lxc:~/C/tcpip_src/ch09-套接字的多种可选项$ bin/set_buf
Input buffer size: 20480
Output buffer size: 204800
```

在设置缓冲的大小时输出的结果和我们设置的结果完全不同，这很合理。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请求。

2. SO_REUSEADDR

本节讲解可选项及其相关的 `Time-wait` 状态。

1. 发生地址分配错误（Binding Error）

当服务器端先断开连接时，使用同一端口号重新运行服务器端时，会输出 "bind() error" 的消息。即服务器端无法立即使用同一端口重新运行。再等大约3分钟后，即可使用该端口重新运行服务器。

2. Time-wait状态

先断开连接的主机在断开连接时会经过 `Time-wait` 状态（四次挥手不再赘述，你应该很熟悉才对）。套接字处于 Time-wait 过程时，相应的端口是正在使用的状态，所以在重新使用该端口时会发生 "bind() error" 的错误。

> 客户端套接字不会经过 Time-wait 状态吗?

不管是服务器端还是客户端，套接字都会有 Time-wait 状态。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是任意指定的。与服务器端不同，客户端每次运行时都会动态分配端口号，因此无需关注Time-wait状态。

3. 地址再分配

Time-wait看似重要，但并不讨人喜欢。
我们可以在套接字的可选项中更改 `SO_REUSEADDR` 的状态，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为1。

reuseaddr_eserver.c

```
# 进程1
lxc@Lxc:~/C/tcpip_src/ch09-套接字的多种可选项$ bin/reuseaddr_eserver 9999
123 # 这个123是来自客户端的123
^C

# 进程2
lxc@Lxc:~/C/tcpip_src/ch09-套接字的多种可选项$ bin/echo_client 127.0.0.1 9999
Connected...........
Input message(Q to quit): 123
Message from server: 123
Input message(Q to quit): q

# 进程1
lxc@Lxc:~/C/tcpip_src/ch09-套接字的多种可选项$ bin/reuseaddr_eserver 9999 # 注意这里重新使用9999的端口号，并未出现绑定错误的消息。
```

```
123 # 这个123也是来自客户端的123

# 进程2
lxc@Lxc:~/C/tcpip_src/ch09-套接字的多种可选项$ bin/echo_client 127.0.0.1 9999
Connected...........
Input message(Q to quit): 123
Message from server: 123
Input message(Q to quit): q
```

3. TCP_NODELAY

1. Nagle 算法

为防止因数据包过多而发生网络过载，Nagle算法在1984年就诞生了，应用于TCP层。

TCP套接字默认使用Nagle算法，因此会最大限度地进行缓冲，直到收到ACK。

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更快。最典型的是传输大文件数据时。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，不使用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。

2. 禁用Nagle算法

get_nagle.c

```
TCP_NODELAY: 0
After setting, the value of TCP_NODELAY is: 1
```

ch10 多进程服务器端

1. 进程的概念及应用

1. 两种类型的服务器端

。。。略

2. 并发服务器端的实现方法

下面列出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

3. 理解进程

略

4. 通过调用 `fork` 函数创建进程

```
NAME
fork - create a child process
SYNOPSIS
#include <sys/types.h>
#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。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。因为通过同一个进程、复制相同的内存空间，之后的程序根据 `fork` 函数的返回值加以区分。

- 父进程：`fork` 函数返回子进程ID。
- 子进程：`fork` 函数返回0。

fork.c

```
lxc@Lxc:~/C/tcpip_src/ch10-多进程服务器端$ bin/fork
Parent Proc: gval: 9, lval: 23
Child Proc: gval: 13, lval: 27
# 从运行结果可以看出，调用 fork 函数后，父子进程拥有完全独立的内存结构。
```

2. 进程和僵尸进程

`fork` 函数产生子进程的终止方式。

- 传递参数并调用 `exit` 函数
- main 函数中执行 `return` 语句并返回值

向 `exit` 函数传递的参数值和main函数的 `return` 语句返回的值都会传递给操作系统。而操作系统不会销毁子进程，直到把这些值传递给产生该子进程的父进程。处在这种状态下的进程就是僵尸进程。既然如此，此僵尸进程何时被销毁呢？***"应该向创建子进程的父进程传递子进程的 `exit` 参数值或 `return` 语句的返回值"***。如何向操作系统传递这些值呢？操作系统不会主动把这些值传递给父进程。只有父进程主动发起请求时，操作系统才会传递该值。换言之，如果父进程未主动要求获得子进程的结束状态值，操作系统将一直保存，并让子进程长时间处于僵尸状态。
这个示例将创建僵尸进程。

zombie.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char* argv[])
{
    pid_t pid = fork();

```

```
    if(pid == 0)
    {
        puts("Hi! I'm a child process\n");
    }
    else
    {
        // 输出子进程ID，通过该值可以查看子进程状态。
        printf("Child Process ID: %d\n", pid);
        // 睡30秒觉，如果父进程终止，处于僵尸状态的子进程将同时销毁。
        // 所以，让父进程睡觉以验证僵尸进程。
        sleep(30);
    }

    if(pid == 0)
        puts("End child process");
    else
        puts("End parent process");

    return 0;
}
```

下面是输出：

```
lxc@Lxc:~/C/tcpip_src/ch10-多进程服务器端$ bin/zombie
Child Process ID: 12510
Hi! I\'m a child process

End child process
End parent process

lxc@Lxc:~/C/tcpip_src/ch10-多进程服务器端$ ps au
lxc        12509  0.0  0.0   2500   512 pts/1    S+   11:21   0:00 bin/zombie
lxc        12510  0.0  0.0      0     0 pts/1    Z+   11:21   0:00 [zombie]
<defunct>
lxc        12762  0.0  0.0  14592  3284 pts/2    R+   11:21   0:00 ps au
# 可以看到，PID为12510的进程状态为僵尸进程(Z+)，
# 经过30s后，PID为12509的父进程和僵尸子进程会同时销毁。
```

1. 销毁僵尸进程1：利用 `wait` 函数

如前所述，为了销毁子进程，父进程应该主动请求获取子进程的返回值。

```
NAME
       wait, waitpid, waitid - wait for process to change state
SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>
       pid_t wait(int *wstatus);
// 成功时返回终止的子进程ID，失败时返回-1。
```

调用此函数时如果已有子进程终止，那么子进程终止时传递的返回值（`exit` 函数的参数值，`return` 的返回值）将保存到该函数参数所指的内存空间。但函数参数指向的内存单元中还包含其他信息，因此需要以下宏分离。

- `WIFEXITED` 子进程正常终止时返回 true
- `WEXITSTATUS` 返回子进程的返回值

也就是说，向 `wait` 函数传递变量 *wstatus* 的地址时，调用 `wait` 函数后应编写如下代码：

```
if(WIFEXIED(wstatus)) // 是正常终止吗?
{
    puts("Normal termination");
    printf("Child pass num: %d", WEXITSTATUS(wstatus)); // 那么返回值是多少呢?
}
```

调用此函数时，如果没有已终止的子进程，那么程序将阻塞直到有子进程终止，因此需谨慎调用此函数。

wait.c

```
lxc@Lxc:~/C/tcpip_src/ch10-多进程服务器端$ bin/wait
Child PID: 14100
Child PID: 14101
Child send one: 3
Child send two: 7
# 你可以通过 ps au 查看确认无僵尸进程
```

2. 销毁僵尸进程2：使用 `waitpid` 函数

`wait` 函数会引起程序阻塞，可以考虑使用 `waitpid` 函数。

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *wstatus, int options);
// 成功时返回终止的子进程ID（或0），失败时返回-1。
```

- *pid* 等待终止的子进程的ID，若传递-1，则等待任意子进程终止；
- *wstatus*，与 `wait` 函数中的 `wstatus` 同义；
- *options*，若传递 `WNOHANG`，则即使没有子进程终止也不会进入阻塞状态，而是返回0并退出函数。

waitpid.c

```
lxc@Lxc:~/C/tcpip_src/ch10-多进程服务器端$ bin/waitpid
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
Child send 24
# 可以看出，waipid 函数并未阻塞。
```

3. 信号处理

"子进程到底何时终止？调用 `waitpid` 函数后要无休止的等待吗？"

父进程往往和子进程一样繁忙，因此不能只调用 `waitpid` 函数以等待子进程终止。接下来讨论解决方案。

1. 向操作系统求助

信号处理（Signal Handling）机制。信号是在特定事件发生时由操作系统向进程发送的消息。为了响应该消息，执行与消息相关的自定义操作的过程称为处理或信号处理。

2. 信号与 `signal` 函数

- 进程："嘿，操作系统！如果我之前创建的子进程终止，就帮我调用 *zombie_handler* 函数。"
- 操作系统："好的！如果你的子进程终止，我会帮你调用 *zombie_handler* 函数，你先把该函数要执行的语句编写好！"

上述对话中进程所讲的相当于"注册信号"过程，即进程发现自己的子进程结束，请求操作系统调用特定函数。该请求通过如下函数调用完成（因此称此函数为信号注册函数）。

```
SYNOPSIS
       #include <signal.h>
       typedef void (*sighandler_t)(int);
       sighandler_t signal(int signum, sighandler_t handler);
// 为了在产生信号时调用，返回之前注册的函数指针。
```

下面给出可以在该函数中注册的部分信号和对应的常数：

- SIGALRM：已到通过调用 `alarm` 函数注册的时间
- SIGINT：输入Ctrl+C
- SIGCHLD：子进程终止

例如：

子进程终止时调用 *mychild* 函数。

```
signal(SIGCHLD, mychild);
```

已到 `alarm` 函数注册的时间，调用 *timeout* 函数。

```
signal(SIGALRM, timeout);
```

输入Ctrl+C时调用 *keycontrol* 函数。

```
signal(SIGINT, keycontrol);
```

下面介绍 `alarm` 函数。

```
NAME
       alarm - set an alarm clock for delivery of a signal
SYNOPSIS
       #include <unistd.h>
       unsigned int alarm(unsigned int seconds);
// 返回0或以秒为单位的距 SIGALRM 信号发生所剩的时间
```

如果调用该函数时向它传递一个正整型参数，响应时间后将产生 SIGALRM 信号。若向该函数传递0，则之前对 SIGALRM 信号的预约将取消。如果通过该函数预约信号后未指定该信号对应的处理函数，则（通过调用 `signal` 函数）终止进程，不做任何处理。

signal.c

```
lxc@Lxc:~/C/tcpip_src/ch10-多进程服务器端$ bin/signal
Wait...
Time out!
Wait...
Time out!
Wait...
Time out!
lxc@Lxc:~/C/tcpip_src/ch10-多进程服务器端$ bin/signal
Wait...
^C Ctrl+C pressed
Wait...
^C Ctrl+C pressed
Wait...
^C Ctrl+C pressed
```

调用函数的主体是操作系统，但进程处于休眠状态时无法调用函数。因此，产生信号时，为了调用信号处理器（信号处理函数），将唤醒由于调用 `sleep` 函数而进入阻塞状态的进程。而且，进程一旦被唤醒，就不会再进入休眠状态。即使还未到 `sleep` 函数中规定的时间也是如此。

3. 利用 `sigaction` 函数进行信号处理

实际上现在很少使用 `signal` 函数编写程序，它只是为了保持对旧程序的兼容。`signal` 函数在UNIX系列的不同操作系统中可能存在区别，但 `sigaction` 函数完全相同。下面介绍 `sigaction` 函数，不过只会讲解可替换 `signal` 函数的部分。

```

SYNOPSIS
       #include <signal.h>
       int sigaction(int signum, const struct sigaction *act,
                     struct sigaction *oldact);
// 成功时返回0，失败时返回-1
```

- *signum*： 与 `signal` 函数相同，传递信号信息
- *act*： 对应于第一个参数的信号处理函数的信息
- *oldact*： 通过此函数获取之前注册的信号处理函数指针，若不需要则传0

`sigaction` 结构体定义如下：

```
struct sigaction
{
    void     (*sa_handler)(int);
    // void     (*sa_sigaction)(int, siginfo_t *, void *) ; 忽略
    sigset_t   sa_mask;
    int        sa_flags;
    // void     (*sa_restorer)(void); 忽略
};
```

sa_handler 成员保存信号处理函数的指针值。*sa_mask* 和 *sa_flags* 的所有位均初始化为0即可。这两个成员用于指定信号相关的选项和特性，而我们的目的主要是防止产生僵尸进程，故省略。

sigaction.c

```
lxc@Lxc:~/C/tcpip_src/ch10-多进程服务器端$ bin/sigaction
Wait...
Time out!
Wait...
Time out!
Wait...
Time out!
```

4. 利用信号处理技术消灭僵尸进程

子进程终止时将产生 `SIGCHLD` 信号，知道这一点就很容易完成。接下来利用 `sigaction` 函数编写示例。

remove_zombie.c

```
lxc@Lxc:~/C/tcpip_src/ch10-多进程服务器端$ bin/remove_zombie
Child proc id: 8491
Child proc id: 8492
Wait...
Hi! I'm child process
Hi! I'm child process
Wait...
Removed proc id: 8491
Child send: 12
Wait...
Wait...
Wait...
```

可以看出子进程并未变成僵尸进程，而是正常终止了。

4. 基于多任务的并发服务器

1. 基于进程的并发服务器模型

2. 实现并发服务器

echo_mpserv.c

```
lxc@Lxc:~/C/tcpip_src/ch10-多进程服务器端$ bin/echo_mpserver 9999
New client connected...
New client connected...

lxc@Lxc:~/C/tcpip_src/ch10-多进程服务器端$ bin/echo_mpclient 127.0.0.1 9999
123
Message from server: 123

lxc@Lxc:~/C/tcpip_src/ch10-多进程服务器端$ bin/echo_mpclient 127.0.0.1 9999
456
Message from server: 456
```

3. 通过 `fork` 函数复制文件描述符

echo_mpserv.c 中的 `fork` 函数调用过程如下图所示。调用 `fork` 函数后，2个文件描述符指向同一套接字。

1个套接字中存在2个文件描述符时，只有2个文件描述符都销毁后，才能销毁套接字。因此，调用 `fork` 函数后，要将无关的套接字关掉。如下图所示：

为了将文件描述符整理成图10-4的形式，示例 *echo_mpserv.c* 的第64行和第73行调用了 `close` 函数。

5. 分割TCP的I/O程序

1. 分割TCP的I/O程序

echo_client.c 的数据回声方式如下："向服务器端传递数据并等待服务器端回复，无条件等待，直到接收完服务器端的回声数据后，才能传递下一批数据"。现在可以创建多个进程，因此可以分割数据收发过程，提高频繁交换数据的程序性能。如图10-5所示：

2. 回声客户端的I/O程序分割

echo_mpclient.c

```
lxc@Lxc:~/C/tcpip_src/ch10-多进程服务器端$ cat -n echo_mpclient.c | sed 's/
//; s/\t/ /'
 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 void read_routine(int sock, char *buf);
11 void write_routine(int sock, char *buf);
```

```c

int main(int argc, char* argv[])
{
    int serv_sock;
    struct sockaddr_in serv_addr;
    pid_t pid;
    char buf[BUF_SIZE];

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

    serv_sock = socket(PF_INET, SOCK_STREAM, 0);
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
    serv_addr.sin_port = htons(atoi(argv[2]));

    if(connect(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))
== -1)
        error_handling("connect() error");

    pid = fork();
    if(pid == 0)
        read_routine(serv_sock, buf);
    else
        write_routine(serv_sock, buf);

    close(serv_sock);
    return 0;
}

void read_routine(int sock, char* buf)
{
    while (1)
    {
        int str_len = read(sock, buf, BUF_SIZE);
        if (str_len == 0)
            return;

        buf[str_len] = 0;
        printf("Message from server: %s\n", buf);
    }
}

void write_routine(int sock, char* buf)
{
    while (1)
    {
        fgets(buf, BUF_SIZE, stdin);
        if(!strcmp(buf, "q\n") || !strcmp(buf, "Q\n"))
        {
            shutdown(sock, SHUT_WR);
            return;
```

```
68 67         }
69 68         write(sock, buf, strlen(buf));
70 69     }
71 70 }
72 71
73 72 void error_handling(char *message)
74 73 {
75 74     fputs(message, stderr);
76 75     fputc('\n', stderr);
77 76     exit(1);
78 77 }
```

注意第65行，调用 `shutdown` 函数向服务器端传递EOF。当然，执行了第66行的 `return` 语句后，可以调用第41行的 `close` 函数传递EOF，但现在已通过第35行的 `fork` 函数调用复制了文件描述符，此时无法通过一次 `close` 函数调用传递EOF，因此需要通过 `shutdown` 函数调用另外传递。

ch11 进程间通信

1. 进程间通信的基本概念

进程间通信(Inter Process Communication，IPC)意味着两个不同进程间可以交换数据，为了完成这一点，操作系统应该提供两个进程可以同时访问的内存空间。

1. 对进程间通信的基本理解

进程A和B之间的如下谈话内容就是一种进程间通信规则。

“如果我有1个面包，变量 *bread* 的值就变为1。如果吃掉这个面包，*bread* 的值又变为0；因此，你可以通过变量 *bread* 的值判断我的状态。”

也就是说，进程A通过变量 *bread* 将自己的状态通知给了进程B，进程B通过变量 *bread* 听到了进程A的话。因此，只要有两个进程可以同时访问的内存空间，就可以通过此内存空间交换数据。但正如第10章所讲，进程具有完全独立的内存结构。就连通过 `fork` 函数创建的子进程也不会与父进程共享内存空间。因此，进程间通信只能通过其他特殊方法完成。

2. 通过管道实现进程间通信

下图表示基于管道的进程间通信模型。

从上图中可以看到，为了完成进程间通信，需要创建管道。管道并非属于进程的资源，而是和套接字一样，属于操作系统。所以两个进程通过操作系统提供的内存空间进行通信。下面介绍创建管道的函数。

```
1 #include <unistd.h>
2 int pipe(int filedes[2]);
3 // 成功时返回0，失败时返回-1
```

- *filedes[0]*： 通过管道接收数据时使用的文件描述符，即管道出口；
- *filedes[1]*： 通过管道传输数据时使用的文件描述符，即管道入口。

来个例子：

pipe1.c

```
lxc@Lxc:~/C/tcpip_src/ch11-进程间通信$ bin/pipe1
Who are you?
```

该示例中的通信方法及路径如下图所示。

3. 通过管道进行进程间的双向通信

下面创建2个进程通过1个管道进行双向数据交换，其通信方式如下图所示：

pipe2.c

注意第19行的 `sleep` 函数调用。***向管道传递数据时，先读的进程会把数据取走***。简言之，数据进入管道后成为无主数据。也就是通过 `read` 函数先读取数据的进程将得到数据，即使是该进程将数据传到了管道。所以，注释第19行将产生问题，在下一行，子进程将读回自己向管道中发送到数据。结果，父进程调用 `read` 后将无限期等待数据进入管道。

从上述示例可以看到，只用1个管道进行双向通信并非易事。为了实现这一点，程序需要预测并控制运行流程，这在每种系统中都不同，可以视为不可能完成的任务。那可咋办呢?
非常简单，1个管道无法完成双向通信的任务，那创建2个管道，各自负责不同的数据流动即可。如下图所示。

pipe3.c

2. 运用进程间通信

1. 保存消息的回声服务器端

下面扩展第10章的 *echo_mpserv.c*，添加如下功能：将回声客户端传输的字符串按序保存到文件中。

echo_storeserv.c。该示例可与任意回声客户端配合运行，我们用第10章的 *echo_mpclient.c* 。

ch12 I/O复用

本章将讨论并发服务器的第二种实现方法———基于I/O复用（Multiplexing）的服务器端构建。

1. 基于I/O复用的服务器端

1. 多进程服务器端的缺点

简言之，创建进程的代价太大，这需要大量的运算和内存空间，由于每个进程都具有独立的内存空间，所以相互间的数据交换也要求采用相对复杂的方法（IPC属于相对复杂的方法）。

2. 理解复用

简言之，就是在一个通信频道中传递多个数据（信号）的技术。时分、频分。。。略

3. 复用技术在服务器端的应用

服务器端引用复用技术可以减少所需进程数。为便于比较，先给出第10章的多进程服务器模型。如下图所示，

引入复用技术后，可以减少进程数。重要的是，无论连接多少个客户端，提供服务的进程只有一个。

2. 理解 `select` 并实现服务器端

运用 `select` 函数是最具代表性的实现复用服务器端方法。Windows平台下也有同名函数提供相同功能，因此具有良好的移植性。

1. `select` 函数的功能和调用顺序

使用 `select` 函数时可以将多个文件描述符集中到一起统一监视，项目如下：

- 是否存在套接字接收数据？
- 无需阻塞传输的数据的套接字有哪些？
- 哪些套接字发生了异常？

> 提示： 监视项称为事件（event）。发生了监视项对应情况时，称 "发生了事件"。这是最常见的表达。

`select` 函数的调用方法和顺序如下：

2. 设置文件描述符

利用 `select` 函数可以同时监视多个文件描述符。当然，监视文件描述符也可视为监视套接字。此时首先需要将要监视的文件描述符集中到一起。集中时也要按照监视项（接收、传输、异常）进行区分，即按照上述3种监视项分成3类。

使用 `fd_set` 数组变量执行此操作，如下图所示，该数组是存有0和1的位数组。

针对 `fd_set` 变量的操作都是以位为单位进行的，在该变量中注册和更改值的操作都由以下宏完成。

- FD_ZERO(fd_set* fdset)：将 *fdset* 变量的所有位初始化为0。
- FD_SET(int fd, fd_set* fdset)：在参数 *fdset* 指向的变量中注册文件描述符 *fd* 的信息。
- FD_CLR(int fd, fd_set* fdset)：在参数 *fdset* 指向的变量中清除文件描述符 *fd* 的信息。
- FD_ISSET(int fd, fd_set* fdset)：若参数 *fdset* 指向的变量中包含文件描述符 *fd* 的信息，则返回真。

这些函数的功能如下图所示，无须赘述。

3. 设置检查（监视）范围及超时

下面讲解图12-5中步骤一的剩余内容，在此之前先简单介绍 `select` 函数。

```
#include <sys/select.h>
#include <sys/time.h>
int select(int nfds, fd_set *readfds, fd_set *writefds,
                   fd_set *exceptfds, struct timeval *timeout);
// 发生错误时返回-1，超时返回0.因发生关注的事件返回时，返回发生事件的文件描述符数。
```

- *nfds*： 监视对象文件描述符的数量
- *readset*： 将所有关注 "是否存在待读取数据" 的文件描述符注册到 *fd_set* 型变量，并传递其地址值。
- *writefds*： 将所有关注 "是否可传输无阻塞数据" 的文件描述符注册到 *fd_set* 型变量，并传递其地址值
- *exceptfds*： 将所有关注 "是否发生异常" 的文件描述符注册到 *fd_set* 型变量，并传递其地址值。
- *timeout*： 调用 `select` 函数后，为防止陷入无限阻塞的状态，传递超时信息

`select` 函数要求通过第一个参数传递监视对象文件描述符的数量。因此，需要得到注册在 *fd_set* 变量中的文件描述符数。每次新建文件描述符时，其值都会增1。故只需要将最大的文件描述符值加1再传递到 `select` 函数即可。加1是因为文件描述符的值从0开始。

`select` 函数的超时时间与最后一个参数有关，其中 `timeval` 结构体定义如下：

```
struct timeval
{
    long    tv_sec;         /* seconds */
    long    tv_usec;        /* microseconds */
};
```

本来 `select` 函数只有在监视的文件描述符发生变化时才返回。如果未发生变化，就会进入阻塞状态。指定超时时间以防止无限期阻塞。如果不想设置超时，则传递NULL即可。

4. 调用 `select` 函数后查看结果

`select` 函数返回正整数时，怎样获知哪些文件描述符发生了变化？向 `select` 函数的第二到第四个参数传递的 *fd_set* 变量中将产生如下的变化。

由上图可知，`select` 函数调用完成后，向其传递的 *fd_set* 变量中将发生变化。原来为1的所有位均变为0，但发生变化的文件描述符对应位除外。因此，值仍为1的位置上的文件描述符发生了变化。

5. `select` 函数调用示例

select.c

```
lxc@Lxc:~/C/tcpip_src/ch12-IO复用$ cat -n select.c | sed 's/    //;s/\t/ /'
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <sys/time.h>
 4 #include <sys/select.h>
```

```
 5
 6 #define BUF_SIZE 30
 7
 8 int main(int argc, char* argv[])
 9 {
10     fd_set reads, temps;
11     int result, str_len;
12     char buf[BUF_SIZE];
13     struct timeval timeout;
14
15     FD_ZERO(&reads);
16     FD_SET(0, &reads);
17
18     while(1)
19     {
20         temps = reads;
21         timeout.tv_sec = 5;
22         timeout.tv_usec = 0;
23         result = select(1, &temps, 0, 0, &timeout);
24         if(result == -1)
25         {
26             puts("select() error!");
27             break;
28         }
29         else if(result == 0)
30         {
31             puts("Time-out!");
32         }
33         else
34         {
35             if(FD_ISSET(0, &temps))
36             {
37                 str_len = read(0, buf, BUF_SIZE);
38                 buf[str_len] = 0;
39                 printf("Message from console: %s\n", buf);
40             }
41         }
42     }
43
44     return 0;
45 }
```

- 第 20 行：将准备好的 *fd_set* 变量 *reads* 的内容复制到 *temps* 变量，因为之前讲过，调用 `select` 函数后，除发生变化的文件描述符对应位外，剩下的所有位将初始化为0。因此，为了记住初始值，必须经过这种复制过程。这是使用 `select` 函数的通用方法，希望各位牢记。
- 第 21、22 行：调用 `select` 函数后，结构体 `timeval` 的成员 `tv_sec` 和 `tv_usec` 的值将被替换为超时前剩余的时间。因此每次调用 `select` 函数前，都需要初始化 `timeval` 结构体变量。

```
lxc@Lxc:~/C/tcpip_src/ch12-IO复用$ bin/select
123
Message from console: 123

321
Message from console: 321

Time-out!
Time-out!
6qw
Message from console: 6qw
```

6. 实现I/O复用服务器端

[echo_selectserv.c]

```
lxc@Lxc:~/C/tcpip_src/ch12-IO复用$ bin/echo_selectserver 9999
Connected client: 4
Connected client: 5
Closed client: 4
Closed client: 5
^C

lxc@Lxc:~/C/tcpip_src/ch12-IO复用$ bin/echo_mpclient 127.0.0.1 9999
qwe
Message from server: qwe

q

lxc@Lxc:~/C/tcpip_src/ch12-IO复用$ bin/echo_mpclient 127.
0.0.1 9999
123
Message from server: 123

q
```

ch13 多种I/O函数

1. `send` & `recv` 函数

1. Linux中的 `send` & `recv`

```
NAME
       send, sendto, sendmsg - send a message on a socket
SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>
       ssize_t send(int sockfd, const void *buf, size_t len, int flags);
    // 成功时返回发送的字节数，失败时返回-1。
    //    ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
                    //   const struct sockaddr *dest_addr, socklen_t
addrlen);
```

```
    //    ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

- *sockfd*： 表示与数据传输对象的连接的套接字文件描述符
- *buf*： 保存待传输数据的缓冲地址值
- *len*： 待传输的字节数
- *flags*： 传输数据时指定的可选项信息

```
NAME
       recv, recvfrom, recvmsg - receive a message from a socket
SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>
       ssize_t recv(int sockfd, void *buf, size_t len, int flags);
    // 成功时返回接收到的字节数（收到EOF时返回0），失败时返回-1
    //    ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
    //                     struct sockaddr *src_addr, socklen_t *addrlen);
    //    ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

- *sockfd*： 表示数据接收对象的连接的套接字文件描述符
- *buf*： 表示接收数据的缓冲地址值
- *len*： 可接收的最大字节数
- *flags*： 接收数据时指定的可选项信息

可选项可以通过位或同时传递多个信息。下表展示了部分可选项信息。

可选项(Option)	含义	send	recv
MSG_OOB	用于传输带外数据（out-of-band data）	1	1
MSG_PEEK	验证输入缓冲中是否存在接收的数据	0	1
MSG_DONTROUTE	数据传输过程中，不参照路由表，在本地网络中寻找目的地	1	0
MSG_DONTWAIT	调用I/O函数时不阻塞，用于使用非阻塞（Non-blocking）I/O	1	1
MSG_WAITALL	防止函数返回，直接接收全部请求的字节数	0	1

在上表中，1表示该函数支持该可选项，0表示不支持。为了方便这样列到了一起。不同操作系统中对上述可选项的支持也不同。

2. MSG_OOB 发送紧急消息

oob_send.c

```
lxc@Lxc:~/C/tcpip_src/ch13-多种IO函数$ cat -n oob_send.c | sed 's/    //;s/\t/
/'
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <string.h>
```

```
 5 #include <sys/socket.h>
 6 #include <arpa/inet.h>
 7
 8 #define BUF_SIZE 30
 9 void error_handling(char *message);
10
11 int main(int argc, char *argv[])
12 {
13     int sock;
14     struct sockaddr_in recv_adr;
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     memset(&recv_adr, 0, sizeof(recv_adr));
24     recv_adr.sin_family = AF_INET;
25     recv_adr.sin_addr.s_addr = inet_addr(argv[1]);
26     recv_adr.sin_port = htons(atoi(argv[2]));
27
28     if (connect(sock, (struct sockaddr *)&recv_adr, sizeof(recv_adr)) ==
-1)
29         error_handling("connect() error!");
30
31     write(sock, "123", strlen("123"));
32     send(sock, "4", strlen("4"), MSG_OOB);
33     write(sock, "567", strlen("567"));
34     send(sock, "890", strlen("890"), MSG_OOB);
35     close(sock);
36     return 0;
37 }
38
39 void error_handling(char *message)
40 {
41     fputs(message, stderr);
42     fputc('\n', stderr);
43     exit(1);
44 }
```

- 第 31~34 行：传输数据。第32和34行紧急传输数据，正常的顺序应该是123、4、567、890，但紧急传输了4和890，因此可知接收顺序也将改变。

oob_recv.c

```
lxc@Lxc:~/C/tcpip_src/ch13-多种IO函数$ cat -n oob_recv.c | sed 's/    //;s/\t/
/'
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <string.h>
 5 #include <signal.h>
 6 #include <sys/socket.h>
```

```c
#include <netinet/in.h>
#include <fcntl.h>

#define BUF_SIZE 30
void error_handling(char *message);
void urg_handler(int signo);

int serv_sock;
int clnt_sock;

int main(int argc, char *argv[])
{
    struct sockaddr_in serv_addr, clnt_addr;
    int str_len, state;
    socklen_t clnt_addr_sz;
    struct sigaction act;
    char buf[BUF_SIZE];

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

    act.sa_handler = urg_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    serv_sock = socket(PF_INET, SOCK_STREAM, 0);
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(atoi(argv[1]));

    if (bind(serv_sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr))
== -1)
        error_handling("bind() error");
    listen(serv_sock, 5);

    clnt_addr_sz = sizeof(clnt_addr);
    clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr,
&clnt_addr_sz);

    fcntl(clnt_sock, F_SETOWN, getpid());
    state = sigaction(SIGURG, &act, 0);

    while ((str_len = recv(clnt_sock, buf, sizeof(buf), 0)) != 0)
    {
        if (str_len == -1)
            continue;
        buf[str_len] = 0;
        puts(buf);
    }
    close(clnt_sock);
    close(serv_sock);
    return 0;
```

```
61 }
62
63 void urg_handler(int signo)
64 {
65     int str_len;
66     char buf[BUF_SIZE];
67     str_len = recv(clnt_sock, buf, sizeof(buf) - 1, MSG_OOB);
68     buf[str_len] = 0;
69     printf("Urgent message: %s \n", buf);
70 }
71
72 void error_handling(char *message)
73 {
74     fputs(message, stderr);
75     fputc('\n', stderr);
76     exit(1);
77 }
```

- 第31、49行：收到 `MSG_OOB` 紧急消息时，操作系统将产生 `SIGURG` 信号，并调用信号处理函数。另外需要注意的是，第63行的信号处理函数内部调用了接收紧急消息的 `recv` 函数。

其中第48行的 `fcntl` 函数用于控制文件描述符。该调用语句的含义如下：将文件描述符 recv_sock 指向的套接字拥有者(F_SETOWN)改为 以 `getpid` 函数返回值作为PID的进程。各位或许感觉套接字拥有者的概念有点生疏。操作系统实际创建并管理套接字，所以从严格意义上说，套接字拥有者是操作系统。只是此处所谓的拥有者是指负责套接字所有事务的主体。上述描述可简要概括如下：文件描述符 recv_sock 指向的套接字引发的 `SIGURG` 信号处理进程变为以 `getpid` 函数返回值作为PID的进程。

之前讲过，多个进程可以共同拥有一个套接字的文件描述符。例如，通过调用 `fork` 函数创建子进程并同时复制文件描述符。此时如果发生 `SIGURG` 信号，应该调用哪个进程的信号处理函数呢？应该可以肯定的是，不会调用所有进程的信号处理函数。因此，处理 `SIGURG` 信号时必须指定处理信号的进程，而 `getpid` 函数返回调用此函数的进程ID。上述调用语句指定当前进程为处理 `SIGURG` 信号的主体。该程序中只创建了一个进程，因此，理应由该进程处理 `SIGURG` 信号。

```
lxc@Lxc:~/C/tcpip_src/ch13-多种IO函数$ bin/oob_recv 9999
Urgent message: 4
Urgent message: 0
123
56789
lxc@Lxc:~/C/tcpip_src/ch13-多种IO函数$ bin/oob_recv 9993
123456789
# 多次运行的结果不一样。。。
```

的确，通过 `MSG_OOB` 可选项传递数据时不会加快数据传输速度，而且通过信号处理函数 `urg_handler` 读取数据时也只能读取1个字节。剩余数据只能通过未设置 `MSG_OOB` 可选项的普通输入函数读取。这是因为TCP不存在真正意义上的 "带外数据"。实际上，`MSG_OOB` 中的OOB是指 out-of-band，而带外数据的真正含义是：通过完全不同的路径传输的数据。即真正意义的Out-of-band需要通过单独的通信路径高速传输数据，但TCP不另外提供，只利用TCP的紧急模式（Urgent mode）进行传输。

3. 紧急模式工作原理

我反正是没听明白书上在讲什么，我复述一遍给你听听吧。

`MSG_OOB` 的真正意义在于督促数据接收对象尽快处理数据，而TCP保持传输顺序的传输特性依然成立。
“那怎么能称为紧急消息呢？”
这确实是紧急消息！因为发送消息者是在催促数据处理的情况下传输数据的。急诊患者及时救治需要如下两个条件。

- 迅速入院
- 医院急救

无法把病人送到医院，并不意味着不需要医院进行急救。TCP的紧急消息无法保证及时入院，但可以急救。当然，急救措施应由程序员完成。之前的 *oob_recv* 的运行过程中也传递了紧急消息，这可以通过事件处理函数确认。这就是 `MSG_OOB` 模式数据传输的实际意义。下面给出设置 `MSG_OOB` 可选项状态下的数据传输过程，如下图所示：

也就是说，实际只用1个字节表示紧急消息信息，这一点可以通过下图看的更清楚。

TCP数据包包含更多的内容，但上图中只标注了与我们主题相关的内容。TCP头部中如下两种信息。

- URG=1 ： 载有紧急消息的数据包
- URG指针 ： 紧急指针位于偏移量为3的位置（在上图中）。

指定 `MSG_OOB` 选项的数据包本身就是紧急数据包，并通过紧急指针表示紧急消息所在位置。但通过图13-2无法得知以下事实：“紧急消息是字符串890，还是90？如若不是，是否为单个字符0？”。
但这并不重要。如前所述，除紧急指针的前面1个字节外，数据接收方将通过调用常用的输入函数读取剩余部分。换言之，紧急消息的意义在于督促消息处理，而非紧急传输形式受限的消息。

4. 检查输入缓冲

同时设置 `MSG_PEEK` 选项和 `MSG_DONTWAIT` 选项，以验证输入缓冲中是否存在接收的数据。设置 `MSG_PEEK` 选项并调用 `recv` 函数时，即使读取了输入缓冲中的数据也不会删除。因此，该选项通常与 `MSG_DONTWAIT` 合作，用于调用以非阻塞方式验证待读数据存在与否的函数。

peek_send.c peek_recv.c

```
lxc@Lxc:~/C/tcpip_src/ch13-多种IO函数$ bin/peek_recv 9999
Buffering 4 bytes: 123
Read again: 123
lxc@Lxc:~/C/tcpip_src/ch13-多种IO函数$ bin/peek_send 127.0.0.1 9999
```

通过运行结果可以验证，仅发送1次的数据被读取了2次，因为第一次调用 `recv` 函数时设置了 `MSG_PEEK` 选项。

2. `readv` & `writev` 函数

1. 使用 `readv` & `write` 函数

这两个函数的功能可概括如下：

“对数据进行整合传输及发送的函数。”

也就是说，通过 `writev` 函数可以将分散保存在多个缓冲中的数据一并发送，通过 `readv` 函数可以由多个缓冲分别接收。因此，适当使用这2个函数可以减少I/O函数的调用次数。

```
SYNOPSIS
       #include <sys/uio.h>
       ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
// 成功时返回发送的字节数，失败时发送-1。
```

- *fd*： 表示数据传输对象的套接字文件描述符。但该函数并不只限于套接字，因此，可以像 `read` 函数一样向其传递文件或标准输出描述符。
- *iov*： `iovec` 结构体数组的地址值，结构体 `iovec` 中包含待发送数据的位置和大小信息。
- *iovcnt*： 向第2个参数传递的数组长度。

`iovec` 结构体的声明如下：

```
struct iovec
{
    void *iov_base; /* Pointer to data.  */
    size_t iov_len; /* Length of data.  */
};
```

`iovec` 结构体由保存待发送数据的缓冲地址值和实际发送的数据长度信息构成。

来个例子：

第一个参数1是文件描述符，因此向控制台输出数据。*ptr* 是存有待发送数据信息的 `iovec` 数组指针。第三个参数为2，因此，从 *ptr* 指向的地址开始，共浏览2个 `iovec` 结构体变量，发送这些指针指向的缓冲数据。

writev.c

```
lxc@Lxc:~/C/tcpip_src/ch13-多种IO函数$ bin/writev
ABCDEFG1234567
Write 15 bytes
```

下面介绍 `readv` 函数，它与 `writev` 函数正相反。

```
SYNOPSIS
       #include <sys/uio.h>
       ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
```

- *fd*： 包含接收数据的文件（或套接字）描述符

- *iov*： 包含数据保存位置和大小信息的 `iovec` 结构体数组的地址值
- *iovcnt*： 第二个参数中数组的长度

readv.c

```
lxc@Lxc:~/C/tcpip_src/ch13-多种IO函数$ bin/readv
I like TCP/IP socket programming~
Read 34 Bytes
First message: I lik
Second message e TCP/IP socket programming~
```

2. 合理使用 `readv` & `writev` 函数

哪种情况适合使用 `writev` 和 `readv` 函数？实际上，能使用该函数的所有情况都适用。例如，需要传输的数据分别位于不同的缓冲（数组）时，需要多次调用 `write` 函数。此时可以通过1次 `writev` 函数替代操作，当然会提高效率。同样，需要将输入缓冲中的数据读入不同位置时，可以不必多次调用 `read` 函数，而是利用1次 `readv` 函数就能大大提高效率。

ch14 多播与广播

向大量客户端发送相同数据时，会对服务器端和网络流量产生负面影响，可以使用多播技术解决该问题。

1. 多播

多播（Multicast）方式的数据传输是基于UDP完成的。因此，与UDP服务器端/客户端的实现方式非常接近。区别在于，UDP数据传输以单一目标进行，而多播数据同时传递到加入（注册）特定组的大量主机。换言之，采用多播方式时，可以同时向多个主机传递数据。

1. 多播的数据传输方式及流量方面的优点

- 多播服务器端针对特定多播组，只发送1次数据
- 即使只发送1次数据，该组内的所有客户端都会接收数据
- 多播组数可以在IP地址范围内任意增加
- 加入特定组即可接收发往该多播组的数据

多播组是D类IP地址（224.0.0.0~239.255.255.255），加入多播组可以理解为通过程序完成如下声明：
“在D类IP地址中，我希望接收发往目标239.234.218.234的多播数据。”
多播是基于UDP完成的，也就是说，多播数据包的格式与UDP数据包相同。只是与一般的UDP数据包不同，向网络传递1个多播数据包时，路由器将复制该数据包并传递到多个主机。像这样，多播需要借助路由器完成。如图14-1所示。

图14-1表示传输至AAA组的多播数据包借助路由器传递到加入AAA组的所有主机的过程。

另外，理论上可以完成多播通信，但不少路由器并不支持多播，或即便支持也因网络拥堵问题而故意阻断多播。因此，为了在不支持多播的路由器中完成多播通信，也会使用隧道（Tunneling）技术（这并非多播程序开发人员需要考虑的问题）。我们只讨论支持多播服务的环境下的编程方法。

2. 路由（Routing）和TTL（Time Ti Live，生存时间），以及加入组的方法

TTL用整数表示，每经过一个路由器就减1，当TTL变为0时，数据包就被销毁。因此，TTL的值设置的过大将影响网络流量。设置的过小也会无法传递到目标。

TTL的设置是通过第9章的套接字可选项完成的。与设置TTL相关的协议层为IPPROTO_IP，选项名为`IP_MULTICAST_TTL`。因此，可以使用如下代码设置TTL。

```
int send_sock;
int time_live = 64;
......
send_sock = socket(PF_INET, SOCK_DGRAM, 0);
setsockopt(send_sock, IPPROTO_IP,
        IP_MULTICAST_TTL, (void*)&time_live, sizeof(time_live));
```

加入多播组也通过设置套接字选项完成。加入多播组的相关协议层为IPPROTO_IP，选项名为`IP_ADD_MEMBERSHIP`。可通过如下代码加入多播组：

```
int recv_sock;
struct ip_mreq join_addr;
......
recv_sock = socket(PF_INET, SOCK_DGRAM, 0);
join_addr.imr_multiaddr.s_addr = "多播组地址信息";
join_addr.imr_interface.s_addr = "加入多播组的主机地址信息";
setsockopt(recv_sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, (void*)&join_addr,
sizeof(join_addr));
......
```

`ip_mreq`结构体的定义如下：

```
struct ip_mreq
{
    struct in_addr imr_multiaddr;
    struct in_addr imr_interface;
}
```

第3章讲过`in_addr`结构体。成员`imr_multiaddr`中写入加入的组IP地址。成员`imr_interface`是加入该组的套接字所属主机的IP地址，也可以使用`INADDR_ANY`。

3. 实现多播 Sender 和 Receiver

多播中用“发送者（Sender）”和“接受者（Receiver）”替代服务器端和客户端。顾名思义，此处的Sender是多播数据的发送主体，Receiver是需要多播组加入过程的数据接收主体。下面是示例，该示例的运行场景如下。

- Sender： 向AAA组广播文件中保存的新闻信息。
- Receiver： 接收传递到AAA组的新闻信息。

news_sender.c news_receiver.c

```
lxc@Lxc:~/C/tcpip_src/ch14-多播与广播$ bin/news_sender 127.0.0.1 9190

lxc@Lxc:~/C/tcpip_src/ch14-多播与广播$ bin/news_receiver 127.0.0.1 9190
Although CPAs will not be among the committee`s members, accountants say that
the committee will in all likelihood tackle issues that have been previously
raised by them with the ministry.
......
```

运行顺序并不重要，因为不像TCP套接字在连接状态下收发数据。只是因为多播属于广播范畴，如果延迟运行Receiver，则无法接收之前传输的多播数据。

> 知识补给站：MBone（Multicast BackBone，多播主干网）。多播是基于MBone这个虚拟网络工作的。各位或许对虚拟网络感到陌生，可以将其理解为“通过网络中的特殊协议工作的软件概念上的网络”。也就是说，Mbone并非可以触及的物理网络。它是以物理网络为基础，通过软件方法实现的多播通信必备虚拟网络。

2. 广播

广播（Broadcast）在“一次性向多个主机发送数据”这一点上与多播类似，但传输数据的范围有区别。多播即使在跨越不同网络的情况下，只要加入多播组就能接收数据。相反，广播只能向同一网络中的主机传输数据。

1. 广播的理解及实现方法

广播是向同一网络中的所有主机传输数据的方法。与多播相同，广播也是基于UDP完成的。根据传输数据时使用的IP地址的形式，广播分为如下2种。

- 直接广播（Directed Broadcast）
- 本地广播（Local Broadcast）

二者在代码实现上的差别主要在于IP地址。
直接广播的IP地址中除了网络地址外，其余主机的地址全部设置为1。例如，希望向网络地址 192.12.34 中的所有主机传输数据时，可以想192.12.34.255传输。换言之，可以采用直接广播的方式向特定区域内所有主机传输数据。
本地广播中使用的IP地址限定为255.255.255.255。例如，192.32.24网络中的主机向255.255.255.255传输数据时，数据将传递到192.32.24网络中的所有主机。

默认生成的套接字会阻止广播，只需通过如下代码更改默认设置。

```
int send_sock;
int bcast = 1;
......
send_sock = socket(PF_INET, SOCK_DGRAM, 0);
......
setsockopt(send_sock, SOL_SOCKET, SO_BROADCAST, (void*)&bcast, sizeof(bcast));
```

调用 `setsockopt` 函数，将 `SO_BROADCAST` 选项设置为 `bcast` 变量中的值1。这意味着可以进行数据广播。当然，上述套接字选项只需在Sender中更改，Receiver的实现不需要该过程。

2. 实现广播数据的Receiver 和 Sender

news_sender_brd.c news_receiver_brd.c

ch15 套接字和标准I/O

我们之前采用的都是默认数据通信手段 `read` & `write` 函数及各种系统I/O函数，你可能想用用学C时掌握的标准I/O函数。

1. 标准I/O函数的优点

1. 标准I/O函数的两个优点

- 标准I/O函数具有良好的移植性（Protability）。
- 标准I/O函数可以利用缓冲提高性能。

关于移植性无需多言。关于第二个优点，使用标准I/O函数时会得到额外的缓冲支持。创建套接字时，操作系统将生成用于I/O的缓冲。此缓冲在执行TCP协议时发挥着非常重要的作用。此时若使用标准I/O函数，将得到额外的另一缓冲支持，如图15-1所示。

从图15-1中可以看到，使用标准I/O函数传输数据时，经过2个缓冲。例如，通过 `fputs` 函数传输字符串 "Hello" 时，首先将数据传递到标准I/O函数的缓冲。然后将数据移动到套接字输出缓冲中，最后将数据发送到对方主机。

既然知道了两个缓冲的关系，接下来再说明各自的用途，设置缓冲的目的主要是为了提高性能，但套接字中的缓冲主要是为了实现TCP协议而设立的。例如，TCP传输中丢失数据时将再次传递，而再次发送数据意味着在某地保存了数据，存在什么地方呢？套接字的输出缓冲。与之相反，使用标准I/O缓冲的主要目的是为了提高性能。

"使用缓冲可以大大提高性能吗？"

实际上缓冲并非在所有情况下都能带来卓越的性能。但需要传输的数据越多，有无缓冲带来的性能差异越大。可以通过如下两种角度说明性能的提高。

- 传输的数据量
- 数据向输出缓冲移动的次数

比较1个字节发送10次（10个数据包）的情况和累计10个字节发送1次的情况。发送数据时使用的数据包含有头信息。头信息与数据大小无关，是按照一定格式填入的。即使假设该头信息占用40字节，需要传递的数据量也存在较大差别。

- 1个字节 10次： 40 * 10 = 400字节
- 10个字节 1次： 40 * 1 = 40字节

所以，在有缓冲的情况下，我们可以缓冲较多数据，减少发送次数，减少了传递的数据量。
另外，为了发送数据，向套接字输出缓冲移动数据也会消耗不少时间。但这同样与移动次数有关。1个字节数据移动10次花费的时间将近10个字节数据移动1次花费时间的10倍。

2. 标准I/O函数和系统函数之间的性能对比

syscpy.c stdcpy.c

复制对象仅限于文本文件，最好是300M以上的，可以看出标准I/O函数明显快于系统I/O函数。

3. 标准I/O函数的几个缺点

- 不易进行双向通信
- 有时可能频繁调用 `fflush` 函数
- 需要以FILE结构体指针的形式返回文件描述符

打开文件时，如果希望同时进行读写操作，则应以r+、w+、a+模式打开。但因为缓冲的缘故，每次切换工作状态时应调用 `fflush` 函数。这也会影响基于缓冲的性能提高。而且，为了使用标准I/O函数，需要FILE结构体指针。而创建套接字时默认返回文件描述符，因此需要将文件描述符转化为FILE指针。

2. 使用标准I/O函数

如前所述，创建套接字时返回文件描述符，而为了使用标准I/O函数，只能将其转换为FILE结构体指针。先介绍其转换方法。

1. 利用 `fdopen` 函数转换为 FILE 结构体指针

```
SYNOPSIS
#include <stdio.h>
          FILE *fdopen(int fd, const char *mode);
// 成功时返回转换的FILE结构体指针，失败时返回NULL。
```

- *fd*： 需要转换的文件描述符。
- *mode*： 将要创建的FILE结构体指针的模式信息，与 `fopen` 函数中的打开模式相同，常见的 读模式 "r"，写模式 "w"。

desto.c

```
lxc@Lxc:~/C/tcpip_src/ch15-套接字和标准IO$ bin/desto
lxc@Lxc:~/C/tcpip_src/ch15-套接字和标准IO$ cat data.dat
I Like ....
```

2. 利用 `fileno` 函数转换为文件描述符

```
SYNOPSIS
       #include <stdio.h>
       int fileno(FILE *stream);
// 成功时返回转换后的文件描述符，失败时返回-1。
```

todes.c

```
lxc@Lxc:~/C/tcpip_src/ch15-套接字和标准IO$ bin/todes
First file descriptor: 3
Second file descriptor: 3
```

3. 基于套接字的标准I/O函数使用

echo_stdserv.c
echo_stdclnt.c

第4章的回声客户端需要将接收的数据转换为字符串（数据的尾部插入0），但本章的回声客户端没有这一过程。因为，使用标准I/O函数后可以以字符串为单位进行数据交换。

ch16 关于I/O流分离的其他内容

调用 `fopen` 函数打开文件后可以与文件交换数据，因此说调用 `fopen` 函数后创建了 "流(Stream)"。此处的 "流" 是指 "数据流动"，但通常可以比喻为 "以数据收发为目的的一种桥梁"。希望各位将 "流" 理解为数据收发路径。

1. 分离I/O流

1. 2次I/O流分离

我们之前通过2种方法分离过I/O流，第一种是第10章的 "TCP I/O过程分离"。这种方法通过调用 `fork` 函数复制出1个文件描述符，以区分输入和输出中使用的文件描述符。虽然文件描述符本身不会根据输入输出进行区分，但我们分开了2个文件描述符的用途，因此这也属于 "流" 的分离。
第二种分离是在第15章。通过2次 `fdopen` 函数的调用，创建读模式FILE指针（FILE结构体指针）和写模式FILE指针。换言之，我们分离了输入工具和输出工具，因此也可视为 "流" 的分离。

2. 分离 "流" 的好处

第10章的 "流" 分离和第15章的 "流" 分离在目的上有一定差异。首先分析第10章 "流" 分离目的。

- 通过分开输入过程（代码）和输出过程降低实现难度
- 与输入无关的输出操作可以提高速度

接下来给出第15章 "流" 分离的目的。

- 为了将FILE指针按读模式和写模式加以区分
- 可以通过区分读写模式降低实现难度
- 通过区分I/O缓冲提高缓冲性能

3. "流" 分离带来的EOF问题

sep_serv.c

```
lxc@Lxc:~/C/tcpip_src/ch16-关于IO流分离的其他内容$ cat -n sep_serv.c | sed 's/
//;s/\t/ /'
 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 #define BUF_SIZE 1024
 8
 9 int main(int argc, char *argv[])
10 {
```

```c
int serv_sock, clnt_sock;
FILE *readfp;
FILE *writefp;

struct sockaddr_in serv_adr, clnt_adr;
socklen_t clnt_adr_sz;
char buf[BUF_SIZE] = {
    0,
};

serv_sock = socket(PF_INET, SOCK_STREAM, 0);
memset(&serv_adr, 0, sizeof(serv_adr));
serv_adr.sin_family = AF_INET;
serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_adr.sin_port = htons(atoi(argv[1]));

bind(serv_sock, (struct sockaddr *)&serv_adr, sizeof(serv_adr));
listen(serv_sock, 5);
clnt_adr_sz = sizeof(clnt_adr);
clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_adr, &clnt_adr_sz);

readfp = fdopen(clnt_sock, "r");
writefp = fdopen(clnt_sock, "w");

fputs("FROM SERVER: Hi~ client? \n", writefp);
fputs("I love all of the world \n", writefp);
fputs("You are awesome! \n", writefp);
fflush(writefp);

fclose(writefp);
fgets(buf, sizeof(buf), readfp);
fputs(buf, stdout);
fclose(readfp);

return 0;
}
```

- 第32、33行： 通过clnt_sock保存的文件描述符创建读模式FILE指针和写模式FILE指针
- 第35~38行：向客户端发送字符串，调用 `fflush` 函数结束发送过程
- 第40、41行：第40行针对写模式FILE指针调用 `fclose` 函数。调用 `fclose` 函数终止套接字时，对方主机将收到EOF。但还剩下读模式FILE指针。那还能不能通过第41行的函数调用接收客户端最后发送的字符串呢？（剧透下，不能）当然，最后的字符串是客户端收到EOF后发送的。

下面是客户端代码：

sep_clnt.c

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

```c
#define BUF_SIZE 1024

int main(int argc, char *argv[])
{
 int sock;
 char buf[BUF_SIZE];
 struct sockaddr_in serv_addr;

 FILE *readfp;
 FILE *writefp;

 sock = socket(PF_INET, SOCK_STREAM, 0);
 memset(&serv_addr, 0, sizeof(serv_addr));
 serv_addr.sin_family = AF_INET;
 serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
 serv_addr.sin_port = htons(atoi(argv[2]));

 connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
 readfp = fdopen(sock, "r");
 writefp = fdopen(sock, "w");

 while (1)
 {
     if (fgets(buf, sizeof(buf), readfp) == NULL)
         break;
     fputs(buf, stdout);
     fflush(stdout);
 }

 fputs("FROM CLIENT: Thank you! \n", writefp);
 fflush(writefp);
 fclose(writefp);
 fclose(readfp);

 return 0;
}
```

- 第25、26行：为了调用标准I/O函数，创建读模式和写模式FILE指针。
- 第30行：收到EOF时，`fgets`函数将返回NULL指针。因此，添加`if`语句使收到NULL时退出循环。
- 第36行：通过该行语句向服务器端发送最后的字符串。当然，该字符串是在收到服务器端的EOF后发送的。

下面进行验证：

```
lxc@Lxc:~/C/tcpip_src/ch16-关于IO流分离的其他内容$ bin/sep_serv 9090
lxc@Lxc:~/C/tcpip_src/ch16-关于IO流分离的其他内容$

lxc@Lxc:~/C/tcpip_src/ch16-关于IO流分离的其他内容$ bin/sep_clnt 127.0.0.1 9090
FROM SERVER: Hi~ client?
I love all of the world
You are awesome!
```

得出结论：

"服务器端未能接收最后的字符串!"

很容易判断其原因：sep_serv.c 示例的第40行调用的 `fclose` 函数完全终止了套接字，而不是半关闭。以上就是需要通过本章解决的问题。半关闭在多种情况下都非常有用，各位必须能够针对 `fdopen` 函数调用时生成的FILE指针进行半关闭操作。

2. 文件描述符的复制和半关闭

1. 终止 "流" 时无法半关闭的原因

图16-1描述的是sep_serv.c示例中的2个FILE指针、文件描述符及套接字之间的关系。

从图16-1中可以看到，实例 sep_serv.c 中的读模式FILE指针和写模式FILE指针都是基于同一文件描述符创建的。因此，针对任意一个FILE指针调用 `fclose` 函数时都会关闭文件描述符，也就终止套接字，如图16-2所示。

从图16-2中可以看到，销毁套接字再也无法进行数据交换。那如何进入可以输入但无法输出的半关闭状态呢？其实很简单，如图16-3所示，创建FILE指针前先复制文件描述符即可。

如图16-3所示，复制后另外创建一个文件描述符，然后利用各自的文件描述符生成读模式FILE指针和写模式FILE指针。这就为半关闭准备好了环境，因为套接字与文件描述符之间有如下关系：

"销毁所有文件描述符后才能销毁套接字"

也就是说，针对写模式FILE指针调用 `fclose` 函数时，只能销毁与该FILE指针相关的文件描述符，无法销毁套接字，参考图16-4。

如图16-4所示，调用 `fclose` 函数后还剩1个文件描述符，因此没有销毁套接字。那此时的状态是否为半关闭状态？不是！！图16-3中讲过，只是准备好了半关闭环境。要进入真正的半关闭状态还需要特殊处理。"图16-4好像已经进入半关闭状态了啊？" 仔细观察，还剩一个文件描述符呢，而且该文件描述符可以同时进行IO（你的FILE指针是读模式，但不妨碍文件描述符可以进行IO啊，我甚至还能再复制文件描述符，然后再搞一个写模式FILE指针呢）。***因此，不但没有发送EOF，而且仍然可以利用文件描述符进行输出。*** 稍后介绍发送EOF并进入半关闭状态的方法，在这之前我们先讲讲如何复制文件描述符，之前的 `fork` 不在考虑范围内。

2. 复制文件描述符

此处讨论的复制并非针对整个进程，而是在同一进程内完成描述符的复制，如图16-5所示。

图16-5给出的是同一进程内存在2个文件描述符可以同时访问文件的情况。当然，文件描述符的值不能重复，因此各使用5和7的整数值，为了形成这种结构需要复制文件描述符。此处的复制具有如下的含义：

"为了访问同一文件或套接字，创建另一个文件描述符"

通常的复制很容易让人理解为将包括文件描述符整数值在内的所有内容的复制，而此处的复制则不同。

3. `dup` & `dup2`

```
SYNOPSIS
       #include <unistd.h>
       int dup(int oldfd);
       int dup2(int oldfd, int newfd);
// 成功时返回复制的文件描述符，失败时返回-1。
```

- *oldfd*： 需要复制的文件描述符
- *newfd*： 明确指定的文件描述符整数值

`dup2` 函数明确指定复制的文件描述符整数值。向其传递大于0且小于进程能生成的最大文件描述符值时，该值将成为复制出的文件描述符值。

dup.c

```
lxc@Lxc:~/C/tcpip_src/ch16-关于IO流分离的其他内容$ bin/dup
cfd1: 3 cfd2: 7
Hi~
It's a nice day~~
Hi~
```

4. 复制文件描述符后"流"的分离

下面通过服务器端的半关闭状态接收客户端最后发送的字符串。

sep_serv2.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 #define BUF_SIZE 1024
 8
 9 int main(int argc, char *argv[])
10 {
11  int serv_sock, clnt_sock;
12  FILE * readfp;
13  FILE * writefp;
14
15  struct sockaddr_in serv_adr, clnt_adr;
16  socklen_t clnt_adr_sz;
17  char buf[BUF_SIZE]={0,};
18
19  serv_sock=socket(PF_INET, SOCK_STREAM, 0);
20  memset(&serv_adr, 0, sizeof(serv_adr));
21  serv_adr.sin_family=AF_INET;
22  serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
23  serv_adr.sin_port=htons(atoi(argv[1]));
24
25  bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr));
26  listen(serv_sock, 5);
```

```
27  clnt_adr_sz=sizeof(clnt_adr);
28  clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr,&clnt_adr_sz);
29
30  readfp=fdopen(clnt_sock, "r");
31  writefp=fdopen(dup(clnt_sock), "w");
32
33  fputs("FROM SERVER: Hi~ client? \n", writefp);
34  fputs("I love all of the world \n", writefp);
35  fputs("You are awesome! \n", writefp);
36  fflush(writefp);
37
38  shutdown(fileno(readfp), SHUT_WR); // 注意这一行，也可以改成下面注释的那一行。
39  // shutdown(fileno(writefp), SHUT_WR);
40  fclose(writefp);
41
42  fgets(buf, sizeof(buf), readfp); fputs(buf, stdout);
43  fclose(readfp);
44  return 0;
45 }
```

- 第30、31行：通过 `fdopen` 函数生成FILE指针。特别是第31行针对 `dup` 函数的返回值生成FILE指针，因此函数调用后进入图16-3的状态。
- 第38行：针对 `fileno` 函数返回的文件描述符调用 `shutdown` 函数。***因此，服务器端进入半关闭状态，并向客户端发送EOF。这一行就是之前所说的发送EOF的方法。调用 `shutdown` 函数时，无论复制出多少文件描述符，套接字都会进入半关闭状态，同时传递EOF。***

```
lxc@Lxc:~/C/tcpip_src/ch16-关于IO流分离的其他内容$ bin/sep_serv2 9999
FROM CLIENT: Thank you!
lxc@Lxc:~/C/tcpip_src/ch16-关于IO流分离的其他内容$ bin/sep_clnt 127.0.0.1 9999
FROM SERVER: Hi~ client?
I love all of the world
You are awesome!
```

运行结果证明了服务器端在半关闭状态下向客户端发送了EOF。通过该示例希望各位掌握一点：

无论复制出多少文件描述符，均应调用 `shutdown` 函数发送EOF并进入半关闭状态

第10章的echo_mpclient.c示例运用过 `shutdown` 函数的这种功能，当时通过 `fork` 函数生成了2个文件描述符，并在这种情况下调用了 `shutdown` 函数发送了EOF。

ch17 优于 select 的 epoll

实现I/O复用的传统方法有 `select` 函数和 `poll` 函数。我们介绍了 `select` 函数的使用方法，但各种原因导致这些方法无法得到令人满意的性能，因此有了Linux下的 `epoll`、BSD的 `kqueue`、Solaris 的 `/dev/poll` 和Windows下的 IOCP 等复用技术。本章将讲解Linux的 `epoll` 技术。

1. `epoll` 理解及应用

`select` 方法因为性能问题并不适合以Web服务器端开发为主流的现代开发环境，所以咱们学一学 `epoll` 吧。

1. 基于 `select` 的I/O复用技术速度慢的原因

最主要的两点如下：

- 每次调用 `select` 函数后常见的针对所有文件描述符的循环语句
- 每次调用 `select` 函数时都需要向该函数传递监视对象信息

上述两点可以从第12章示例echo_selectserv.c的第50、52以及57行代码得到确认。调用 `select` 后，并不是把发生变化的文件描述符单独集中到一起，而是通过观察作为监视对象的 `fd_set` 变量的变化找出发生变化的文件描述符（示例 echo_selectserv.c 的第57、59行），因此无法避免针对所有监视对象的循环语句。而且，作为监视对象的 `fd_set` 变量会发生变化，所以调用 `select` 函数前应复制并保存原有信息（参考echo_selectserv.c的第50行），并在每次调用 `select` 函数时传递新的监视对象信息。

相对于循环语句，影响性能的更大障碍是每次传递监视对象信息。因为传递监视对象信息具有如下的含义：

"每次调用 `select` 函数时向操作系统传递监视对象信息"。

应用程序向操作系统传递数据将对程序造成很大的负担，而且无法通过优化代码解决，因此将成为性能上的致命弱点。

"那为何需要把监视对象信息传递给操作系统呢？"

有些函数不需要操作系统的帮助就能完成功能，而有些则必须借助操作系统。假设各位定义了四则运算相关函数，此时无需操作系统的帮助。但 `select` 函数与文件描述符有关，更准确地说，是监视套接字变化的函数。而套接字是由操作系统管理的，所以 `select` 函数绝对需要借助于操作系统的帮助才能完成功能。 `select` 函数的这一缺点可以通过如下方式弥补：

"仅向操作系统传递1次监视对象，监视范围或内容发生变化时只通知发生变化的事项"

这样就无需每次调用 `select` 函数时都向操作系统传递监视对象信息，但前提是操作系统支持这种处理方式。（每种操作系统支持的程度和方式存在差异）。Linux的支持方式是 `epoll`。

2. `select` 也有优点

本章的 `epoll` 只在Linux下支持，也就是说该I/O复用模型不具有兼容性。相反，大部分操作系统都支持 `select` 函数。只要满足如下两个条件，即使在Linux平台也不应拘泥于 `epoll`。

- 服务器端接入者少
- 程序应具有兼容性

实际并不存在适用于所有情况的模型，各位应理解好各种模型优缺点。

3. 实现 `epoll` 时必要的函数和结构体

能够克服 `select` 函数缺点的 `epoll` 函数具有如下优点，这些优点正好与之前的 `select` 函数缺点相反。

- 无需编写以监视状态变化为目的的针对所有文件描述符的循环语句
- 调用对应于 `select` 函数的 `epoll_wait` 函数时无需每次传递监视对象信息

下面介绍epoll服务器端实现中需要的3个函数。

- `epoll_create`： 创建保存epoll文件描述符的空间
- `epoll_ctl`： 向空间注册并注销文件描述符

- `epoll_wait`： 与 `select` 函数类似，等待文件描述符发生变化

`select` 方式中为了保存监视对象文件描述符，直接声明了 `fd_set` 变量。但epoll方式下由操作系统负责保存监视对象文件描述符，因此需要向操作系统请求创建保存文件描述符的空间，此时使用的函数就是 `epoll_create`。
此外，为了添加和删除监视对象文件描述符，`select` 方式中需要 `FD_SET`、`FD_CLR`函数。但epoll方式中，通过 `epoll_ctl` 函数请求操作系统完成。最后，`select` 方式下调用 `select` 函数等待文件描述符的变化，而epoll中调用 `epoll_wait` 函数。`select` 方式中通过 `fd_set` 变量查看监视对象的状态变化（事件发生与否），而epoll方式中通过如下结构体 `epoll_event` 将发生变化的（发生事件的）文件描述符单独集中到一起。

```
struct epoll_event
{
    __uint32_t events;
    epoll_data_t data;
}

typedef union epoll_data
{
    void* ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
} epoll_data_t;
```

声明足够大的 `epoll_event` 结构体数组后，传递给 `epoll_wait` 函数时，发生变化的文件描述符信息将被填入该数组。因此，无需像 `select` 函数那样针对所有文件描述符进行循环。

4. `epoll_create`

epoll是从Linux的2.5.44版内核开始引入的。不过各位使用的Linux内核肯定在这个版本之上，无需担心。可以通过如下命令查看自己的Linux内核版本：

```
lxc@Lxc:~/C/tcpip_src$ cat /proc/sys/kernel/osrelease
5.15.0-91-generic
```

下面查看 `epoll_wait` 函数

```
SYNOPSIS
       #include <sys/epoll.h>
       int epoll_create(int size);
// 成功时返回epoll文件描述符，失败时返回-1
```

- *size*： epoll实例的大小

调用 `epoll_create` 函数时创建的文件描述符保存空间称为 "epoll例程" 。通过参数 *size* 传递的值并非用来决定epoll例程的大小，而是仅供操作系统参考。

> **提示：** 操作系统将完全忽略传递给 `epoll_create` 的参数
> Linux 2.6.8之后的内核将完全忽略传入 `epoll_create` 函数的 *size* 参数，因为内核会根据情况调整 epoll例程的大小。

`epoll_create` 创建的资源与套接字相同，也由操作系统管理。因此，该函数和创建套接字的情况相同，也会返回文件描述符。也就是说，该函数返回的文件描述符主要用于区分epoll例程。需要终止时，与其他文件描述符相同，也要调用 `close` 函数。

5. `epoll_ctl`

生成epoll例程后，应在其内部注册监视对象文件描述符，此时使用 `epoll_ctl` 函数。

```
SYNOPSIS
       #include <sys/epoll.h>
       int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
// 成功时返回0，失败时返回-1
```

- *epfd*： 用于注册监视对象的epoll例程的文件描述符
- *op*： 用于指定监视对象的添加、删除或更改等操作
- *fd*： 需要注册的监视对象文件描述符
- *event*： 监视对象的事件类型

下面举几个例子:

```
epoll_ctl(A, EPOLL_CTL_ADD, B, C);
```

"在epoll例程A中注册文件描述符B，主要目的是监测参数C中的事件"

```
epoll_ctl(A, EPOLL_CTL_DEL, B, NULL);
```

"从epoll例程A中删除文件描述符B"

从上述调用语句中可以看到，从监视对象中删除时，不需要监视类型（事件信息），因此向第四个参数传递NULL。

接下来介绍可以向 `epoll_ctl` 第二个参数传递的常量及含义。

- `EPOLL_CTL_ADD`： 将文件描述符注册到epoll例程
- `EPOLL_CTL_CEL`： 从epoll例程中删除文件描述符
- `EPOLL_CTL_MOD`： 更改注册的文件描述符的关注事件发生情况

如前所述，向 `epoll_ctl` 的第二个参数传递 `EPOLL_CTL_DEL` 时，应同时向第四个参数传递NULL。但Linux 2.6.9之前的内核不允许传递NULL。虽然被忽略掉，但也应该传递 `epoll_event` 结构体变量的地址值（本书示例将传递NULL）。其实这是BUG，但也没必要因此怀疑epoll的功能，因为我们使用标准函数中也存在BUG。

下面讲解 `epoll_ctl` 函数的第四个参数，其类型是之前讲过的 `epoll_event` 结构体指针。
`epoll_event` 结构体用于保存发生事件的文件描述符集合。

```
struct epoll_event event;
......
event.events = EPOLLIN; // 发生需要读取数据的情况时
event.data.fd = sockfd;
epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, &event);
......
```

接下来给出 `epoll_event` 的成员 `events` 中可以保存的常量及所指的事件类型。

- `EPOLLIN` ： 需要读取数据的情况
- `EPOLLOUT` ： 输出缓冲为空，可以立即发送数据的情况
- `EPOLLPRI` ： 收到OOB数据的情况
- `EPOLLRDHUP` ： 断开连接或半关闭的情况，这在边缘触发方式下非常有用
- `EPOLLERR` ： 发生错误的情况
- `EPOLLET` ： 以边缘触发的方式得到事件通知
- `EPOLLONSHOT` ：发生一次事件后，相应文件描述符不再收到事件通知。因此需要向 `epoll_ctl` 函数的第二个参数传递 `EPOLL_CTL_MOD` ，再次设置事件。

可以通过位或运算同时传递多个上述参数。

6. `epoll_wait`

最后介绍与 `select` 函数对应的 `epoll_wait` 函数，epoll相关函数中默认最后调用该函数。

```
SYNOPSIS
       #include <sys/epoll.h>
       int epoll_wait(int epfd, struct epoll_event *events,
                      int maxevents, int timeout);
// 成功时返回发生事件的文件描述符数，失败时返回-1
```

- *epfd* ： 表示事件发生监视范围的epoll例程的文件描述符
- *events* ： 保存发生事件的文件描述符集合的结构体地址值
- *maxevents* ： 第二个参数中可以保存的最大事件数
- *timeout* ： 以毫秒为单位的等待时间，传递-1时，一直等待到发生事件。

该函数的调用方式如下。需要注意的是第二个参数所指缓冲需要动态分配。

```
int event_cnt;
struct epoll_event * ep_events;
......
ep_events = malloc(sizeof(struct epoll_event) * EPOLL_SIZE); // EPOLL_SIZE 是宏常量
......
event_cnt = epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
```

调用函数后，返回发生事件的文件描述符数，同时在第二个参数所指向的缓冲中保存事件的文件描述符集合。因此，无需像 `select` 那样插入针对所有文件描述符的循环。

7. 基于 epoll 的回声服务器端

echo_epollserv.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <arpa/inet.h>
```

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

#define BUF_SIZE 100
#define EPOLL_SIZE 50
void error_handling(char *buf);

int main(int argc, char* argv[])
{
    int serv_sock, clnt_sock;
    struct sockaddr_in serv_addr, clnt_addr;
    socklen_t clnt_addr_sz;
    int str_len;
    char buf[BUF_SIZE];

    int epfd, event_cnt;
    struct epoll_event event;
    struct epoll_event* ep_events;

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

    serv_sock = socket(PF_INET, SOCK_STREAM, 0);
    if(serv_sock == -1)
        error_handling("socket() error");
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(atoi(argv[1]));

    if(bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))
== -1)
        error_handling("bind() error");
    if(listen(serv_sock, 5) == -1)
        error_handling("listen() error");

    epfd = epoll_create(EPOLL_SIZE);
    ep_events = malloc(sizeof(struct epoll_event) * EPOLL_SIZE);

    event.events = EPOLLIN;
    event.data.fd = serv_sock;
    epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);

    while(1)
    {
        event_cnt = epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
        if(event_cnt == -1)
        {
            puts("epoll_wait() error");
            break;
        }

        for(int i = 0; i < event_cnt; i++)
```

```
61         {
62             if(ep_events[i].data.fd == serv_sock)
63             {
64                 clnt_addr_sz = sizeof(clnt_addr);
65                 clnt_sock = accept(serv_sock,
66                                    (struct sockaddr *)&clnt_addr,
&clnt_addr_sz);
67                 if(clnt_sock == -1)
68                     error_handling("accept() error");
69
70                 event.events = EPOLLIN;
71                 event.data.fd = clnt_sock;
72                 epoll_ctl(epfd, EPOLL_CTL_ADD, clnt_sock, &event);
73                 printf("Connected client: %d\n", clnt_sock);
74             }
75             else
76             {
77                 str_len = read(ep_events[i].data.fd, buf, BUF_SIZE);
78                 if(str_len == 0)
79                 {
80                     epoll_ctl(epfd, EPOLL_CTL_DEL, ep_events[i].data.fd,
NULL);
81                     close(ep_events[i].data.fd);
82                     printf("Closed client: %d\n", ep_events[i].data.fd);
83                 }
84                 else
85                 {
86                     write(ep_events[i].data.fd, buf, str_len);
87                 }
88             }
89         }
90     }
91     close(serv_sock);
92     close(epfd);
93
94     return 0;
95 }
96
97 void error_handling(char *buf)
98 {
99      fputs(buf, stderr);
100     fputc('\n', stderr);
101     exit(1);
102}
```

之前解释过关键代码，而且程序结构与 `select` 方式没有区别，故没有代码说明。如果你有些地方难以理解，说明未掌握本章之前的内容和 `select` 模型，建议复习。

2. 条件触发和边缘触发

1. 条件触发和边缘触发的区别在于发生事件的时间点

- 条件触发方式中，只要输入缓冲中有数据就会一直通知该事件。
- 边缘触发方式中输入缓冲收到数据时仅注册1次该事件，即使输入缓冲中还留有数据，也不会再进行注册。

条件触发更准确的说法是水平触发吧，目前我还没看过其它书，但作为ee的学生，水平触发和边沿触发还是很清楚的。

2. 掌握条件触发的事件特性

epoll默认以条件触发方式工作。下面看个例子。

echo_EPLTserv.c

该示例与之前的 echo_epollserv.c 之间的差异如下：

- 将调用 `read` 函数时使用的缓冲大小缩减为4个字节（第10行）
- 插入验证 `epoll_wait` 函数调用次数的语句（第60行）

减少缓冲大小是为了阻止服务器端一次性读取接收的数据。换言之，调用 `read` 函数后，输入缓冲中仍有数据需要读取。而且会因此注册新的事件并从 `epoll_wait` 函数返回时将循环输出 "return epoll_wait" 字符串。

```
lxc@Lxc:~/C/tcpip_src/ch17-优于select的epoll$ bin/echo_EPLTserv 9998
return epoll_wait()
Connected client: 5
return epoll_wait()
return epoll_wait()
return epoll_wait()
return epoll_wait()
return epoll_wait()
return epoll_wait()
return epoll_wait()
return epoll_wait()
return epoll_wait()
return epoll_wait()
return epoll_wait()
Closed client: 5

lxc@Lxc:~/C/tcpip_src/ch17-优于select的epoll$ bin/echo_stdclnt 127.0.0.1 9998
Connected...
Input message(q/Q to quit): 123
Message from server: 123

Input message(q/Q to quit): It's my life.I am your father!
Message from server: It's my life.I am your father!

Input message(q/Q to quit): q
```

从运行结果中可以看出，每当收到客户端数据时，都会注册该事件，并因此多次调用 `epoll_wait` 函数。

> **提示：** `select` 模型是条件触发还是边缘触发?
> `select` 模型是以条件触发的方式工作的，输入缓冲中如果还剩有数据，肯定会注册事件。

3. 边缘触发服务器端实现中必知的两点

下面讲解边缘触发服务器端的实现方法。在此之前，先说明两点，这些是实现边缘触发的必知内容。

- 通过 `errno` 变量验证错误原因
- 为了完成非阻塞（Non-blocking）I/O，更改套接字特性

Linux的套接字相关函数通过返回-1通知发生了错误。虽然知道发生了错误，但仅凭这些内容无法得知产生错误的原因。因此，为了在发生错误时提供额外的信息，Linux声明了如下全局变量：

```
int errno
```

为了访问该变量需要引入 `error.h` 头文件，因为此头文件中有上述变量的 `extern` 声明。另外，每种函数发生错误时，保存到 `errno` 变量中的值都不同，没必要记住所有可能的值。本节只介绍如下类型的错误：

" `read` 函数发现输入缓冲中没有数据可读时返回-1，同时在 `errno` 中保存 `EAGAIN` 常量。"

下面讲解将套接字改为非阻塞方式的方法。Linux提供更改或读取文件属性的如下方法（曾在第13章使用过）。

```
NAME
       fcntl - manipulate file descriptor
SYNOPSIS
       #include <unistd.h>
       #include <fcntl.h>
       int fcntl(int fd, int cmd, ... /* arg */ );
// 成功时返回 cmd 参数相关值，失败时返回-1
```

- *fd*： 属性更改目标的文件描述符
- *cmd*： 表示函数调用的目的

从上述声明中可以看到，`fcntl` 具有可变参数的形式。如果向第二个参数传递 `F_GETFL`，可以获得第一个参数所指的文件描述符属性（int型）。反之，如果传递 `F_SETFL`，可以更改文件描述符属性。若希望将文件（套接字）改为非阻塞模式，需要如下2条语句。

```
int flag = fcntl(fd, F_FETFL, 0);
fcntl(fd, F_SETFL, flag|O_NONBLOCK);
```

通过第一条语句获取之前设置的属性信息，通过第二条语句在此基础上添加非阻塞 `O_NONBLOCK` 标志。

4. 实现边缘触发的回声服务器端

之所以介绍读取错误原因的方法和非阻塞模式套接字创建方法，原因在于二者都与边缘触发的服务器端实现有密切联系。首先说明为何需要 `errno` 确认错误原因。

"边缘触发方式中，接收数据时仅注册1次该事件"

就因为这种特点，一旦发生输入相关事件，就应该读取输入缓冲中的全部数据。因此需要检验输入缓冲是否为空。

" `read` 函数返回-1，变量 `errno` 中的值为 `EAGAIN` 时，说明没有数据可读。"

既然如此，为何还需要将套接字变成非阻塞模式？边缘触发方式下，以阻塞方式工作的 `read` & `write` 函数有可能引起服务器端长时间停顿。因此，边缘触发方式中一定要采用非阻塞 `read` & `write` 函数。接下来给出以边缘触发方式工作的回声服务器端示例。

echo_EPETserv.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <fcntl.h>
6  #include <errno.h>
7  #include <arpa/inet.h>
8  #include <sys/socket.h>
9  #include <sys/epoll.h>
10
11 #define BUF_SIZE 4
12 #define EPOLL_SIZE 50
13 void setnonblockingmode(int fd);
14 void error_handling(char *buf);
15
16 int main(int argc, char* argv[])
17 {
18  int serv_sock, clnt_sock;
19  struct sockaddr_in serv_addr, clnt_addr;
20  socklen_t clnt_addr_sz;
21  int str_len;
22  char buf[BUF_SIZE];
23
24  int epfd, event_cnt;
25  struct epoll_event *ep_events;
26  struct epoll_event event;
27
28  if(argc != 2)
29  {
30      printf("Usage: %s <port>\n", argv[0]);
31      exit(1);
32  }
33
34  serv_sock = socket(PF_INET, SOCK_STREAM, 0);
35  if(serv_sock == -1)
36      error_handling("socket() error");
37  memset(&serv_addr, 0, sizeof(serv_addr));
38  serv_addr.sin_family = AF_INET;
39  serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
40  serv_addr.sin_port = htons(atoi(argv[1]));
41
42  if(bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) ==
-1)
43      error_handling("bind() error");
44  if(listen(serv_sock, 5) == -1)
45      error_handling("listen() error");
46
```

```
47  epfd = epoll_create(EPOLL_SIZE);
48  event.data.fd = serv_sock;
49  event.events = EPOLLIN;
50  setnonblockingmode(serv_sock);
51  epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);
52  ep_events = malloc(sizeof(struct epoll_event) * EPOLL_SIZE);
53
54  while(1)
55  {
56      event_cnt = epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
57      if(event_cnt == -1)
58      {
59          puts("epoll_wait() error");
60          break;
61      }
62      puts("return epoll_wait()");
63
64      for(int i = 0; i < event_cnt; i++)
65      {
66          if(ep_events[i].data.fd == serv_sock)
67          {
68              clnt_addr_sz = sizeof(clnt_addr);
69              clnt_sock = accept(serv_sock,
70                  (struct sockaddr*)&clnt_addr, &clnt_addr_sz);
71              setnonblockingmode(clnt_sock);
72              if(clnt_sock == -1)
73                  error_handling("aceept() error");
74
75              event.events = EPOLLIN | EPOLLET;
76              event.data.fd = clnt_sock;
77              epoll_ctl(epfd, EPOLL_CTL_ADD, clnt_sock, &event);
78              printf("Connected client: %d\n", clnt_sock);
79          }
80          else
81          {
82              while (1)
83              {
84                  str_len = read(ep_events[i].data.fd, buf, BUF_SIZE);
85                  if (str_len == 0)
86                  {
87                      epoll_ctl(epfd, EPOLL_CTL_DEL,
88                                ep_events[i].data.fd, NULL);
89                      close(ep_events[i].data.fd);
90                      printf("Close client: %d\n", ep_events[i].data.fd);
91                  }
92                  else if(str_len < 0)
93                  {
94                      if(errno == EAGAIN)
95                          break;
96                  }
97                  else
98                  {
99                      write(ep_events[i].data.fd, buf, str_len);
100                 }
101             }
102         }
```

```
103     }
104 }
105 close(serv_sock);
106 close(epfd);
107
108 return 0;
109 }
110
111 void setnonblockingmode(int fd)
112 {
113     int flag=fcntl(fd, F_GETFL, 0);
114     fcntl(fd, F_SETFL, flag|O_NONBLOCK);
115 }
116
117 void error_handling(char *buf)
118 {
119     fputs(buf, stderr);
120     fputc('\n', stderr);
121     exit(1);
122 }
```

- 第11行：为了验证边缘触发的工作方式，将缓冲设置为4字节
- 第62行：为了观察事件发生数而添加的输出字符串的语句
- 第75、76行：第75行将 `accept` 函数创建的套接字改为非阻塞模式。第76行向 `EPOLLIN` 添加 `EPOLLET` 标志，将套接字事件注册方式改为边缘触发
- 第92行：`read` 函数返回-1且 `errno` 值为 `EAGAIN` 时，意味着读取了输入缓冲中的全部数据，因此需要通过 `break` 语句跳出循环

```
lxc@Lxc:~/C/tcpip_src/ch17-优于select的epoll$ bin/echo_EPETserv 9983
return epoll_wait()
Connected client: 5
return epoll_wait()
return epoll_wait()
return epoll_wait()
Close client: 5
^C

lxc@Lxc:~/C/tcpip_src/ch17-优于select的epoll$ bin/echo_stdclnt 127.0.0.1 9983
Connected...
Input message(q/Q to quit): I like computer programming
Message from server: I like computer programming

Input message(q/Q to quit): Do you like computer programing????
Message from server: Do you like computer programing????

Input message(q/Q to quit): q
```

上述运行结果中需要注意的是，客户端发送消息次数和服务器端 `epoll_wait` 函数调用次数。客户端从请求连接到断开连接共发送了3次数据，服务器端也相应产生了3次事件。

5. 条件触发和边缘触发孰优孰劣

我们从理论和代码的角度充分理解了条件触发和边缘触发，但仅凭这些还无法理解边缘触发相对于条件触发的优点。边缘触发方式可以做到如下这点：

"可以分离接收数据和处理数据的时间点"

虽然比较简单，但非常准确的说明了边缘触发的优点。现阶段给出如下情景帮助大家理解，如图17-1所示。

图17-1的运行流程如下：

1. 服务器端分别从客户端A、B、C接收数据
2. 服务器端按照A、B、C的顺序重新组合收到的数据
3. 组合的数据将发送给任意主机

为了完成该过程，若能按如下流程运行程序，服务器端的实现并不难。

1. 客户端按照A、B、C的顺序连接服务器端，并依次向服务器端发送数据
2. 需要接收数据的客户端应在客户端A、B、C之前连接到服务器端并等待

但现实中可能频繁出现以下这些情况，换言之，以下这些情况更符合实际。

- 客户端C和B正在向服务器发送数据，但A尚未连接到服务器。
- 客户端A、B、C乱序发送数据
- 服务器端收到数据，但要接收数据的目标客户端还未连接到服务器端

因此，即使输入缓冲中收到数据（注册相应事件），服务器端也能决定读取和处理这些数据的时间点，这样就给服务器端的实现带来了巨大的灵活性。

"条件触发中无法区分数据接收的处理吗？"

并非不可能。但在输入缓冲收到数据的情况下，如果不读取（延迟处理），则每次调用 `epoll_wait` 函数时都会产生相应事件。而且事件数会增加，服务器端能承受吗？这在现实中是不可能的（本身并不合理，因此是根本不会做的事）。

从实现模型的角度看，边缘触发更有可能带来高性能，但不能简单地认为 "只要使用边缘触发就一定能提高速度"。

ch18 多线程服务器端的实现

1. 理解线程的概念

1. 引入线程的背景

多进程模型的缺点可概括如下：

- 创建进程的过程会带来一定的开销
- 为了完成进程间数据交换需要特殊的IPC技术
- 上下文切换是多进程模型中最大的开销

线程相比于进程有如下优点：

- 线程的创建和上下文切换相对应的比进程的更快

- 线程间交换数据时无需特殊技术

2. 线程和进程的差异

线程是为了解决如下困惑登场的：

"嘿！为了得到多条代码执行流而复制整个内存区域的负担太重了！"

每个进程的内存空间都由保存全局变量的 "数据区"、向 `malloc` 等函数的动态分配提供空间的堆(Heap)、函数运行时使用的栈(Stack)构成。每个进程都拥有这种独立空间，多个进程的内存结构如图18-1所示。

但如果以获得多个代码执行流为主要目的，则不应像图18-1那样完全分离内存结构，而只需分离栈区域。通过这种方式可以获得如下优势。

- 上下文切换时不需要切换数据区和堆
- 可以利用数据区和堆交换数据

实际上这就是线程。线程为了保持多条代码执行流而隔开了栈区域，因此具有图18-2所示的内存结构。

如图18-2所示，多个线程将共享数据区和堆，为了保持这种结构，线程将在进程内创建并运行。也就是说，进程和线程可以定义为如下形式：

- 进程：在操作系统构成单独执行流的单位
- 线程：在进程构成单独执行流的单位

如果说进程在操作系统内部生成多个执行流，那么线程就在同一进程内部创建多条执行流。因此，操作系统、进程、线程之间的关系可以通过图18-3表示。

2. 线程创建及运行

POSIX是为了提高Unix系列操作系统间的移植性而制定的API规范。下面要介绍的线程创建方法也是以POSIX标准为依据的。因此，它不仅适用于Linux，也适用于大部分Unix系列的操作系统。

1. 线程的创建和执行流程

线程具有单独的执行流，因此需要单独定义线程的 `main` 函数，还需要请求操作系统在单独执行流中执行该函数，完成该功能的函数如下。

```
NAME
       pthread_create - create a new thread
SYNOPSIS
       #include <pthread.h>
       int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);
       Compile and link with -pthread.
// 成功时返回0，失败时返回其他值。
```

- *thread*： 保存新创建线程ID的变量地址值。线程与进程相同，也需要用于区分不同线程的ID。
- *attr*： 用于传递线程属性的参数，传递NULL时，创建默认的线程。

- *start_routine*： 相当于线程的 `main` 函数、在单独执行流中执行的函数地址值（函数指针）。
- *arg*： 通过第三个参数传递调用函数时包含传递参数信息的变量地址值。

来个示例:

thread1.c

```
lxc@Lxc:~/C/tcpip_src/ch18-多线程服务器端的实现$ bin/thread1
running thread
running thread
running thread
running thread
running thread
end of main
```

不多说废话了，直接看吧。

```
NAME
       pthread_join - join with a terminated thread
SYNOPSIS
       #include <pthread.h>
       int pthread_join(pthread_t thread, void **retval);
       Compile and link with -pthread.
// 成功时返回0，失败时返回其他值
```

- *thread*： 该参数值ID的线程终止后才会从该函数返回
- *status*： 保存线程的 `main` 函数返回值的指针变量地址值。

调用该函数的进程（线程）将进入等待状态，直到第一个参数为ID的线程终止为止。而且可以得到该线程的 `main` 函数的返回值。

thread2.c

```
lxc@Lxc:~/C/tcpip_src/ch18-多线程服务器端的实现$ bin/thread2
running thread
running thread
running thread
running thread
running thread
Thread return message: Hello, I'm thread~
```

2. 可在临界区内调用的函数

根据临界区是否引起问题，函数可分为以下2类。

- 线程安全函数（Thread-safe function）
- 非线程安全函数（Thread-unsafe funtion）

线程安全函数被多个线程调用时也不会引起问题，非线程安全函数被同时调用时会引发问题。
幸运的是大多数标准函数都是线程安全的函数。更幸运的是，我们不需要自己区分线程安全的函数和非线程安全的函数（Windows中同样如此）。因为这些平台在定义非线程安全函数的同时，提供了具有相同功能的线程安全函数。比如，第8章介绍过的如下函数就不是线程安全的函数：

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

同时提供了线程安全的同一功能的函数。

```
struct hostent* gethostbyname_r(
    const char* name, struct hostent* result, char* buffer, int buflen,
    int* h_errnop
);
```

线程安全的函数的名称后缀通常为_r（这与Windows平台不同）。既然如此，多个线程同时访问的代码块中应该调用 `gethostbyname_r` 而不是 `gethostbyname`？当然！但这种方法会给程序员带来沉重的负担。幸好可以通过如下方法自动将 `gethostbyname` 函数调用改为 `gethostbyname_r` 函数调用！

"声明头文件前定义 `_REENTRANT` 宏"

`gethostbyname` 函数和 `gethostbyname_r` 函数和参数声明都不同，因此，这种宏声明方式有巨大的吸引力。另外，无需为了上述宏定义特意添加 #define 语句，可以在编译时通过添加 `-D_REENTRANT` 选项定义宏。

下面编译线程相关代码时均默认添加 `-D_REENTRANT` 选项。

3. 工作（Worker）线程模型

这个示例将计算1到10的和，但并不是在main函数中进行累加运算，而是创建2个线程，其中一个线程计算1到5的和，另一个线程计算6到10的和，main函数只负责输出运算结果。这种方式的编程模型称为"工作线程(Worker thread)模型"。计算1到5之和的线程与计算6到10之和的线程将成为main线程管理的工作。最后给出程序执行流程图。如图18-7所示。

thread3.c

```
lxc@Lxc:~/C/tcpip_src/ch18-多线程服务器端的实现$ bin/thread3
Result: 55
```

运行结果是55，虽然正确，但示例本身存在问题。此处存在临界区相关问题，因此再介绍另一示例。该示例与上述示例相似，只是增加了发生临界区相关错误的可能性。

thread4.c

```
lxc@Lxc:~/C/tcpip_src/ch18-多线程服务器端的实现$ bin/thread4
sizeof long long: 8
Result: 152586
lxc@Lxc:~/C/tcpip_src/ch18-多线程服务器端的实现$ bin/thread4
sizeof long long: 8
Result: 58508
lxc@Lxc:~/C/tcpip_src/ch18-多线程服务器端的实现$ bin/thread4
sizeof long long: 8
Result: 302218
```

运行结果并不是0！而且每次运行的结果均不同。

3. 线程存在的问题和临界区

1. 多个线程访问同一变量是问题

略。。不想阐述，都讲烂了的东西了 😂😂

2. 临界区位置

下面观察示例 thread4.c 的2个main函数。

```
void* thread_inc(void* arg)
{
    for(int i = 0; i < 5000000; i++)
        num += 1; // 临界区
    return NULL;
}

void* thread_dec(void* arg)
{
    for(int i = 0; i < 5000000; i++)
        num -= 1; // 临界区
    return NULL;
}
```

4. 线程同步

前面讨论了线程中存在的问题，接下来就要讨论解决方法——线程同步。

1. 同步的两面性

线程同步用于解决线程访问顺序引发的问题。需要同步的情况可以从如下两方面考虑。

- 同时访问同一内存空间时发生的情况
- 需要指定访问同一内存空间的线程执行顺序的情况

同时访问同一内存空间时发生的情况就是thread4.c中发生的问题。现在讨论第二种情况。这是 "控制线程执行顺序" 的相关内容。假设有A、B两个线程，线程A负责向指定内存写入数据，线程B负责取走该数据。这种情况下，线程A首先应该访问约定的内存空间并保存数据。万一线程B先访问并取走数据，将导致错误结果。像这种需要控制执行顺序的情况也需要使用同步技术。

稍后将介绍 "互斥量(Mutex)" 和 "信号量(Semaphore)" 这2种同步技术。二者概念上十分接近，只要理解了互斥量就很容易掌握信号量。而且大部分同步技术的原理都大同小异，因此，只要掌握了本章介绍的同步技术，就很容易掌握并运用Windows平台下的同步技术。

2. 互斥量

互斥量是 "Mutual Exclusion" 的简写，表示不允许多个线程同时访问。互斥量是一把优秀的锁，接下来介绍互斥量的创建及销毁函数。

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
*mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
// 成功时返回0，失败时返回其他值。
```

- *mutex*： 创建互斥量时传递保存互斥量的变量地址值，销毁时传递需要销毁的互斥量地址值。
- *attr*： 传递即将创建的互斥量属性，没有特别需要指定的属性时传递NULL。

从上述函数声明中也可看到，为了创建相当于锁系统的互斥量，需要声明如下 `pthread_mutex_t` 型变量。

`pthread_mutex_t` mutex;

该变量的地址传递给 `pthread_mutex_init` 函数，用来保存操作系统创建的互斥量（锁系统）。如果不需要配置特殊的互斥量属性，则向第二个参数传递NULL时，可以利用 `PTHREAD_MUTEX_INITIALIZER` 宏进行如下声明：

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

但推荐各位尽可能使用 `pthread_mutex_init` 函数进行初始化，因为通过宏进行初始化时很难发现发生的错误。接下来介绍利用互斥量锁住或释放临界区时使用的函数。

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
// 成功时返回0，失败时返回其他值。
```

创建好互斥量的前提下，可以通过如下结构保护临界区。

```
pthread_mutex_lock(&mutex);
// 临界区开始
......
// 临界区结束
pthread_mutex_destroy(&mutex);
```

简言之就是利用lock和unlock函数围住临界区的两端。此时互斥量相当于一把锁，阻止多个线程同时访问。还有一点需要注意，线程退出临界区时，如果忘了调用 `pthread_mutex_unlock` 函数，那么其他为了进入临界区而调用 `pthread_mutex_lock` 函数的线程就无法摆脱阻塞状态。这种情况称为 "死锁(Dead-lock)"，需要格外注意。接下来利用互斥量解决示例 thread4.c 中遇到的问题。

mutex.c

```
lxc@Lxc:~/C/tcpip_src/ch18-多线程服务器端的实现$ bin/mutex
Result: 0
```

从运行结果可以看出，已解决示例thread4.c中的问题。但确认运行结果需要等待较长时间。因为互斥量lock、unlock函数的调用过程要比想象中花费更长的时间。

```
void *thread_dec(void *arg)
{

    pthread_mutex_lock(&mutex);
    for (int i = 0; i < 5000000; i++)
        num += 1;
    pthread_mutex_unlock(&mutex);

    return NULL;
}
```

以上临界区划分范围较大，但这是考虑到如下优点所做的决定：

"最大限度减少互斥量lock、unlock函数的调用次数。"

你可以将lock与unlock函数的调用放到for循环内，即用锁仅包围临界区，这样会调用5000000次lock与unlock（书上是5千万次，我缩小了10倍），这样你可以感受一下多慢。我们的做法是扩展了临界区，但是变量增加到5000000前不允许其他线程访问，这反而也是一个缺点。其实这里没有正确答案，需要根据不同程序酌情考虑究竟是扩大还是缩小临界区。

3. 信号量

信号量与互斥量很相似，在互斥量的基础上很容易理解信号量。此处只涉及利用 "二进制信号量" 完成 "控制线程顺序" 为中心的同步方法。下面给出信号量创建及销毁方法。

```
NAME
       sem_init - initialize an unnamed semaphore
SYNOPSIS
       #include <semaphore.h>
       int sem_init(sem_t *sem, int pshared, unsigned int value);
       Link with -pthread.
```

- *sem*： 创建信号量时传递保存信号量的变量地址值，销毁时传递需要销毁的信号量变量地址值。
- *pshared*： 传递其他值时，创建可由多个进程共享的信号量。传递0时，创建只允许1个进程内部使用的信号量。我们需要完成同一进程内的线程同步，故传递0。
- *value*： 指定新创建的信号量初始值。

上述函数的 *pshared* 参数超出了我们关注的范围，故默认向其传递0。接下来介绍信号量中相当于互斥量lock、unlock的函数。

```
#include <semaphore.h>
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
```

- *sem*： 传递保存信号量读取值的变量地址值，传递给 `sem_post` 时信号量增1，传递给 `sem_wait` 时信号量减1。

调用 `sem_init` 函数时，操作系统将创建信号量对象，此对象中记录着 "信号量值(Semaphore Value)" 整数。该值在调用 `sem_post` 函数时增1，调用 `sem_wait` 函数时减1。但信号量的值不能小于0，因此，在信号量为0的情况下调用 `sem_wait` 函数时，调用的线程将进入阻塞状态。当然，如果此时有其他线程调用 `sem_post` 时信号量的值将变为1，而原本阻塞的线程可以将该信号量重新减为0并跳出阻塞状态。实际上就是通过这种特性完成对临界区的同步操作，可以通过如下形式同步临界区（假设信号量的初始值为1）。

```
sem_wait(&sem); // 信号量变为0
// 临界区的开始
......
// 临界区的结束
sem_post(&sem); // 信号量变为1...
```

上述代码结构中，调用 `sem_wait` 函数进入临界区的线程在调用 `sem_post` 函数前不允许其他线程进入临界区。信号量的值在0和1之间跳转，因此，具有这种特性的机制称为 "二进制信号量"。接下来给出信号量的示例，关于控制访问顺序的同步。该示例的场景如下：

"线程A从用户输入得到值后存入全局变量 `num`，此时线程B将取走该值并累加。该过程共进行5次，完成后输出总和并退出程序。"

semaphore.c

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

void* read(void* arg);
void* accu(void* arg);
static sem_t sem_one;
static sem_t sem_two;
static int num;

int main(int argc, char* argv[])
{
    pthread_t id_t1, id_t2;
    sem_init(&sem_one, 0, 0);
    sem_init(&sem_two, 0, 1);

    pthread_create(&id_t1, NULL, read, NULL);
    pthread_create(&id_t2, NULL, accu, NULL);

    pthread_join(id_t1, NULL);
    pthread_join(id_t2, NULL);

    sem_destroy(&sem_one);
    sem_destroy(&sem_two);

    return 0;
}

void* read(void* arg)
{
    for(int i = 0; i < 5; i++)
    {
        fputs("Input num: ", stdout);

        sem_wait(&sem_two);
        scanf("%d", &num);
        sem_post(&sem_one);
    }

    return NULL;
}

void* accu(void* arg)
{
    int sum = 0;
    for(int i = 0; i < 5; i++)
    {
        sem_wait(&sem_one);
        sum += num;
        sem_post(&sem_two);
```

```
51      }
52      printf("Result: %d\n", sum);
53
54      return NULL;
55  }
```

- 第14、15行：生成2个信号量，一个信号量的值为0，另一个为1。
- 第35、50行：利用信号量变量 `sem_two` 调用 `wait` 函数和 `post` 函数。这是为了防止在调用 `accu` 的函数的线程还未取走数据的情况下，调用 `read` 函数的线程覆盖原值。
- 第37、46行：利用信号量变量 `sem_one` 调用 `wait` 和 `post` 函数。这是为了防止调用 `read` 函数的线程写入新值前，`accu` 函数再取走旧值。

```
lxc@Lxc:~/C/tcpip_src/ch18-多线程服务器端的实现$ bin/semaphore
Input num: 12
Input num: 13
Input num: 14
Input num: 15
Input num: 16
Result: 70
```

5. 线程的销毁和多线程并发服务器端的实现

1. 销毁线程的3种方法

Linux线程不是在首次调用的线程 `mian` 函数返回时自动销毁，所以用如下2种方法之一加以明确。否则由线程创建的内存空间将一直存在。

- 调用 `pthread_join` 函数
- 调用 `pthread_detach` 函数

之前调用过 `pthread_join` 函数。调用该函数时，不仅会等待线程终止，还会引导线程销毁。但该函数的问题是，线程终止前，调用该函数的线程将进入阻塞状态。因此，通常会通过如下函数调用引导线程销毁。

```
NAME
       pthread_detach - detach a thread
SYNOPSIS
       #include <pthread.h>
       int pthread_detach(pthread_t thread);
       Compile and link with -pthread.
// 成功时返回0，失败时返回其他值。
```

- *thread*：终止的同时需要销毁的线程ID。

调用上述函数不会引起线程终止或进入阻塞状态，可以通过该函数引导销毁线程创建的内存空间。调用该函数后不能再针对相应线程调用 `pthread_join` 函数，这需要格外注意。虽然还有其他方法在创建线程时可以指定销毁时机，但与 `pthread_detach` 方式相比，结果上没有太大差异，故省略其说明。

2. 多线程并发服务器端的实现

chat_server.c chat_clnt.c

```
lxc@Lxc:~/C/tcpip_src/ch18-多线程服务器端的实现$ bin/chat_serv 9898
Connected client IP: 127.0.0.1
Connected client IP: 127.0.0.1

lxc@Lxc:~/C/tcpip_src/ch18-多线程服务器端的实现$ bin/chat_clnt 127.0.0.1 9898
lxc
[lsd] hahaha
woshinidie
[lxc] woshinidie
[lsd] woshinidye
wuer
[lxc] wuer
q

lxc@Lxc:~/C/tcpip_src/ch18-多线程服务器端的实现$ bin/chat_clnt 127.0.0.1 9898
lsd
hahaha
[lsd] hahaha
[lxc] woshinidie
woshinidye
[lsd] woshinidye
[lxc] wuer
q
```

ch24 制作HTTP服务器端

没啥好说的，只实现了 `GET` 方法。

webserv_linux.c

```
./webserv_linux 9999
```

然后如下图就行。

就这样吧。。。