Linux 网络编程

一、 socket 简介

socket 是网络编程的一个接口，它是一种特殊的 I/O，用 socket 函数建立一个 socket 连接，函数返回一个整型的 socket 描述符，随后进行数据传输。

socket 分为三种类型：流式 socket，数据报 socket 和原始 socket。

二、 socket 套接口简介

套接结构包含 IP 地址和端口信息。一般套接口地址结构都以“sockaddr”开头。socket 根据所使用的的协议的不同，可以分为 TCP 套接口和 UDP 套接口，有时也称为流式套接口和数据套接口。

三、 socket 套接口的数据结构

sockaddr 和 sockaddr_in。两个结构类型都是用来保存 socket 信息的，如 IP 地址，通信端口等。

```
struct sockaddr {
    unsigned short sa_family; /* address family, AF_xxx */
    /*sa_family 一般为 AF_INET，代表 Internet（TCP/IP）地址族*/
    char sa_data[14]; /* 14 bytes of protocol address */
    /*sa_data 则包含该 socket 的 IP 地址和端口号*/
    };
```

```
struct sockaddr_in {
    short int sin_family; /*地址族 */
    unsigned short int sin_port; /* 端口号 */
    struct in_addr sin_addr; /* IP 地址 */
    unsigned char sin_zero[8]; /* 填充 0 以保持与 struct sockaddr 同样的大小 */
    };
```

sin_family：指代协议族，在 socket 编程中只能是 AF_INET

sin_port：存储端口号（使用网络字节顺序）

sin_addr：存储 IP 地址，使用 in_addr 这个数据结构

sin_zero：是为了让 sockaddr 与 sockaddr_in 两个数据结构保持大小相同而保留的空字节，可以用 bzero()函数将其置 0。

而其中 in_addr 结构的定义如下：

```
typedef struct in_addr {
    union {
    struct{ unsigned char s_b1,s_b2, s_b3,s_b4;} S_un_b;
    struct{ unsigned short s_w1, s_w2;} S_un_w;
    unsigned long S_addr;
    } S_un;
    } IN_ADDR;
```

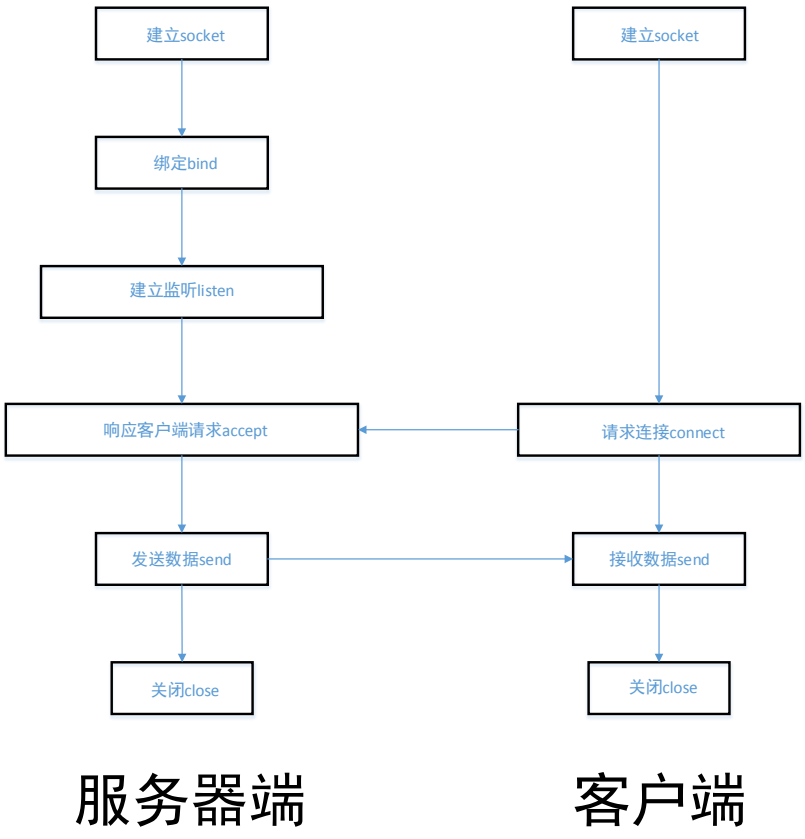
四、 TCP 编程

网络上绝大多数的通信服务采用服务器机制（Client/Server），TCP 提供的是一种可靠的、面向连接的服务。

基于 TCP 网络编程常用函数及其功能

函数名	功能
bind	bind()将 socket 与本机的一个端口绑定，随后就可以在该端口监听服务请求。
connect	面向连接的客户程序使用 connect()函数来配置 socket，并与远程服务器建立一个 TCP 连接。
listen	listen()函数使 socket 处于被动的监听模式，并为该 socket 建立一个输入数据队列，将到达的服务请求保存在队列中，直到程序处理它们。
accept	accept()函数让服务器接收客户的连接请求。
close	停止在该 socket 上的任何数据请求。
send	数据发送函数。
recv	数据接收函数。

例. 服务器通过 socket 连接后，向客户端发送字符串“连接上了”。在服务器上显示客户端的 IP 地址或域名。



程序中主要语句说明如下：

1. 服务器端

1) 建立 socket。程序调用 socket()函数，该函数返回一个类似于文件描述符的句柄，同时也意味着为一个 socket 数据结构分配存储空间。用语句实现：

```
socket(AF_INET,SOCK_STREAM,0);
```

其中，参数 AF_INET 表示采用 IPv4 协议进行通信；参数 SOCK_STREAM 表示采用流式 socket，即 TCP。

2) 绑定 bind。将 socket 与本机上的一个端口绑定，随后就可以在该端口监听服务请求。用语句实现：

```
bind(sockfd,(struct sockaddr *)&my_addr,sizeof(struct sockaddr));
```

其中，参数 my_addr 表示指向包含本机 IP 地址及端口号等信息。

3) 建立监听 listen。使 socket 出于被动的监听模式，并为该 socket 建立一个输入数据队列，将到达的服务请求保存在此队列中，直到程序处理它们。用语句实现：

```
listen(sockfd,BACKLOG);
```

其中，参数 BACKLOG 表示最大连接数。

4) 响应客户请求 accept。用函数 accept 生成一个新的套接口描述符，让服务器接收客户的连接请求。用语句实现：

```
accept(sockfd,(struct sockaddr *)&remote_addr,&sin_size);
```

其中，参数 remote_addr 用于接收客户端地址信息；参数 sin_size 用于存放地址的长度。

5) 发送数据 send。该函数用于在面向连接的 socket 上进行数据发送。用语句实现：

```
send(client_fd,'连接上了 \n',26,0);
```

其中，26 表示以字节为单位的长度。

6) 关闭 close。停止在该 socket 上的任何数据操作。用语句实现：

```
close(client_fd);
```

2. 客户端

1) 建立 socket。程序调用 socket()函数，该函数返回一个类似于文件描述符的句柄，同时也意味着为一个 socket 数据结构分配存储空间。用语句实现：

```
socket(AF_INET,SOCKET_STREAM,0);
```

2) 请求连接 connect。启动和远端主机的直接连接。用语句实现：

```
connect(sockfd, (struct sockaddr*) &serv_addr, sizeof(struct sockaddr));
```

3) 接收数据 recv。该函数用于面向连接的 socket 上进行数据接收。用语句实现：

```
recv(sockfd,buf,MAXDATASIZE,0);
```

4) 关闭 close。停止在该 socket 上的任何数据操作。用语句实现：

```
close(sockfd);
```

服务器端代码：

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * File:      main.c
 * Author: Mr Mahaitao
 *
 * Created on March 6, 2018, 3:44 AM
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <arpa/inet.h>
#define SERVPORT 3333 /*服务器监听端口号*/
#define BACKLOG 10 /*最大同时连接请求数*/
/*
 *
 */
int main(int argc, char** argv) {

	int sockfd, client_fd; /*sock_fd:监听 socket; client_fd:数据传输 socket*/
	struct sockaddr_in my_addr; /*本机地址信息*/
	struct sockaddr_in remote_addr; /*客户端信息*/
	int sin_size;

	if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
		perror("socket  创建失败！");
		exit(1);
	}

	my_addr.sin_family = AF_INET;
	my_addr.sin_port = htons(SERVPORT); /*htons()把 16 位值从主机字节序转换成网络字节序*/
	my_addr.sin_addr.s_addr = htonl(INADDR_ANY); /*IP 地址设置成 INADDR_ANY,让系统自动获取本机的 IP
地址*/
	bzero(&(my_addr.sin_zero), 8); /*保持与 struct sockaddr 同样大小*/

	if (bind(sockfd, (struct sockaddr *) &my_addr, sizeof (struct sockaddr)) == -1) {
		perror("bind  失败！");
		exit(1);
	}

	if (listen(sockfd, BACKLOG) == -1) {
		perror("lisnten  失败！");
		exit(1);
	}

	while (1) {
```

```
            sin_size = sizeof (struct sockaddr_in);
            if (client_fd = accept(sockfd, (struct sockaddr *) &remote_addr, &sin_size) == -1) {
                perror("accept  失败!");
                continue;
            }
            printf("收到一个来自：%s\n", inet_ntoa(remote_addr.sin_addr));
            if (!fork()) { /*子进程代码段*/
                if (send(client_fd, "连接上了  \n", 26, 0) == -1)
                    perror("send  失败！ ");
                close(client_fd);
                exit(0);
            }
            close(client_fd);
        }
        return (EXIT_SUCCESS);
}
```

注意：代码实例中的 fork()函数生成一个子进程来处理数据传输部分，fork()语句对于子进程返回的值为 0。所以，包含 fork 函数的 if 语句是子进程代码部分，它与 if 语句后面的父进程代码是并发执行的。

客户端代码：

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * File:     client.c
 * Author: Mr Mahaitao
 *
 * Created on March 6, 2018, 5:21 AM
 */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
```

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#define SERVPORT 3333
#define MAXDATASIZE 100 /*每次最大数据传输量*/

/*
 *
 */
int main(int argc, char** argv) {

	int sockfd, recvbytes;
	char buf[MAXDATASIZE];
	struct hostent *host;
	struct sockaddr_in serv_addr;
	if (argc < 2) {
		fprintf(stderr, "Please enter the server's hostname!\n");
		exit(1);
	}
	if ((host = gethostbyname(argv[1])) == NULL) {
		herror("gethostbyname error! ");
		exit(1);
	}
	if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
		perror("socket create error! ");
		exit(1);
	}
	serv_addr.sin_family = AF_INET;
	serv_addr.sin_port = htons(SERVPORT);
	serv_addr.sin_addr = *((struct in_addr *) host->h_addr);
	bzero(&(serv_addr.sin_zero), 8);
	if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof (struct sockaddr)) == -1) {
		perror("connect error! ");
		exit(1);
	}
	if ((recvbytes = recv(sockfd, buf, MAXDATASIZE, 0)) == -1) {
		perror("recv error！ ");
		exit(1);
	}
	buf[recvbytes] = '\0';
	printf("收到：  %s", buf);
	close(sockfd);
	return (EXIT_SUCCESS);
}
```

相关函数介绍：

socket 建立的函数说明如下：

所需的头文件	#include <sys/types.h> #include <sys/socket.h>
函数功能	socket 的建立
函数原型	int socket(int domain, int type, int protocol);
函数传入值	domain　AF_INET：Ipv4 协议 AF_INET6：Ipv6 协议 AF_LOCAL：UNIX 域协议 AF_ROUTE：路由套接字 AF_KEY：秘钥套接字 type　SOCK_STREAM：字节数据流套接字 SOCK_DGRAM：数据报套接字 SOCK_RAW：原始套接字 protocol　设置为 0，表示自动选择
函数返回值	若成功，则返回 socket 描述符；若失败，则返回-1
备注	socket 描述符是一个指向内部数据结构的指针，它指向描述符表入口。调用 socket 函数时，socket 执行体将建立一个 socket，实际上“建立一个 socket”意味着为一个 socket 数据结构分配存储空间。

bind 函数说明如下

所需的头文件	#include <sys/socket.h>
函数功能	将 socket 与本机上的一个端口绑定，随后就可以在该端口监听服务请求
函数原型	int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
函数传入值	sockfd　调用 socket 函数返回的 socket 描述符 my_addr　指向包含有远端主机 IP 地址及端口号等信息的 sockaddr 类型的指针 addrlen　指结构体
函数返回值	若成功，则返回 0；若失败，则返回-1
备注	通过 socket 调用返回一个 socket 描述符后，在使用 socket 进行网络传输以前，必须配置该 socket。面向连接的 socket 客户端通过调用 connect 函数，在 socket 数据结构中保存本地地址和远端信息。无连接 socket 的客户端和服务器端以及面向连接 socket 的服务器端通过调用 bind 函数来配置本地信息

connect 函数说明如下：

所需的头文件	#include <sys/socket.h>
函数功能	面向连接的客户程序使用 connect 函数来配置 socket 并与远程服务器建立一个 TCP 连接
函数原型	int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
函数传入值	sockfd 调用 socket 函数返回的 socket 描述符 my_addr 指向包含有本机 IP 地址及端口号等信息的 sockaddr 类型的指针 addrlen 指结构体长度 sizeof(struct sockaddr)
函数返回值	若成功，则返回 0；若失败，则返回-1
备注	connect 函数启动和远端主机的直接连接。只有面向连接的客户程序使用 socket 时，才需要将此 socket 与远端主机相连。无连接协议从不建立直接连接，面向连接的服务器从不启动一个连接，它只是被动地在协议端口监听客户的请求。

listen 函数说明如下：

所需的头文件	#include <sys/socket.h>
函数功能	listen 函数使 socket 处于被动的监听模式，并为该 socket 建立一个输入数据队列，将到达的服务请求保存在此队列中，直到程序处理它们
函数原型	int listen(int sockfd, int backlog);
函数传入值	sockfd 调用 socket 函数返回的 socket 描述符 backlog 最大主机连接数
函数返回值	若成功，则返回 0；若失败，则返回-1

accept 函数说明如下：

所需的头文件	#include <sys/types.h> #include <sys/socket.h>
函数功能	accept 函数让服务器接收客户的连接请求
函数原型	int accept(int sockfd,struct sockaddr *addr,socklen_t *addrlen);
函数传入值	sockfd 调用 socket 函数返回的 socket 描述符 addr 这是一个结果参数，它用来接受一个返回值，该返回值指定客户端的地址，当然这个地址是通过某个地址结构来描述的，用户应该知道这一个什么样的地址结构。如果对客户的地址不感兴趣，那么可以把这个值设置为 NULL addrlen 它也是结果的参数，用来接受上述 addr 的结构的大小的，它指明 addr 结构所占有的字节个数。同样的，它也可以被设置为 NULL
函数返回值	若成功，则返回一个新的套接字描述符；若失败，则返回-1。

备注	如果 accept 成功返回，则服务器与客户已经正确建立连接了，此时服务器通过 accept 返回的套接字来完成与客户的通信。

close 函数说明如下：

所需的头文件	#include <sys/socket.h>
函数功能	停止在该 socket 上的任何数据操作
函数原型	int close(int sockfd);
函数传入值	socket 描述符
函数返回值	若成功，则返回 0；若失败，则返回-1

send 函数说明如下：

所需的头文件	#include <sys/types.h> #include <sys/socket.h>
函数功能	数据发送
函数原型	int send(int sockfd, const void *msg, int len, int flags);
函数传入值	sockfd 用来传输数据的 socket 描述符 msg 一个指向要发送数据的指针 len 以字节为单位的数据的长度 flag 一般情况下置为 0
函数返回值	若成功，则返回实际上发送出去的字节数；若失败，则返回-1

recv 函数说明如下：

所需的头文件	#include <sys/types.h> #include <sys/socket.h>
函数功能	数据接收
函数原型	int recv(int sockfd, void *buf, int len, unsigned int flags);
函数传入值	sockfd 用来接收数据的 socket 描述符 buf 是存放接收数据的缓冲区 len 是缓冲区的长度 flags 一般情况下置为 0
函数返回值	若成功，则返回实际上接收的字节数；若失败，则返回-1

listen 和 accept 的比较：

来自于 Stack Overflow 上面的一个问答

Ask： I've been reading this tutorial to learn about socket programming. It seems that the listen() and accept() system calls both do the same thing, which is block and wait for a client to connect to the socket that was created with the socket() system call. Why do you need two separate steps for this? Why not just use one system call?
By the way, I have googled this question and found similar questions, but none of the answers were satisfactory. For example, one of them said that accept() creates the socket, which makes no sense, since I know that the socket is created by socket().

Answer1： The listen() function basically sets a flag in the internal socket structure marking the socket as a passive listening socket, one that you can call accept on. It opens the bound port so the socket can then start receiving connections from clients.
The accept() function asks a listening socket to accept the next incoming connection and return a socket descriptor for that connection. So, in a sense, accept() does create a socket, just not the one you use to listen() for incoming connections on.

Answer2： It is all part of the historic setup. listen prepares socket for the next accept call. Listen also allows one to setup the backlog - the number of connections which will be accepted by the system, and than put to wait until your program can really accept them. Everything which comes after the backlog is full well be rejected by the system right away. listen never blocks, while accept will block (unless the socket is in non-blocking mode) until the next connection comes along. Obviously, this does not have to be two separate functions - it is conceivable that accept() function could do everything listen does.

维基百科上的内容：

- listen() is used on the server side, and causes a bound TCP socket to enter listening state.
- accept() is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.

五、 UDP 编程

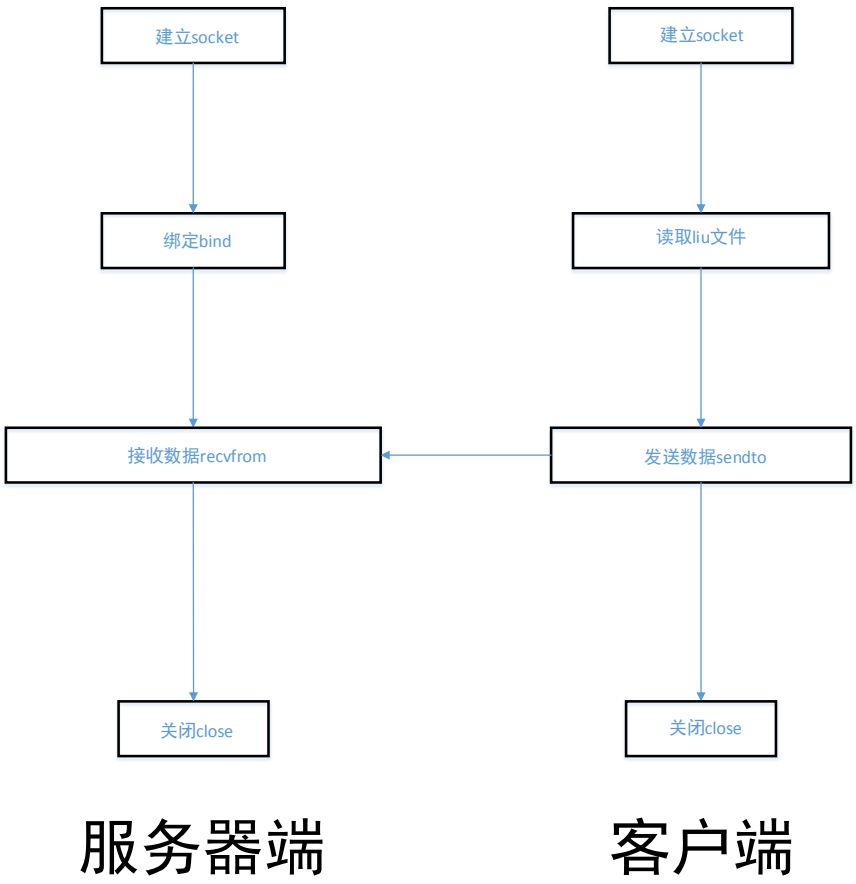
UDP 是 User Datagram Protocol 的简称， 中文名是用户数据报协议。其主要特点是：

- UDP 传送数据前并不与对方建立连接，即 UDP 是无连接的，在传输数据前，发送方和接收方相互交换信息使双方同步。
- UDP 不对收到的数据进行排序，在 UDP 报文的首部中并没有关于数据顺序的信息（如 TCP 所采用的序号），而且报文不一定按顺序到达的，所以接收端无从排起。
- UDP 对接收到的数据报不发送确认信号，发送端不知道数据是否被正确接收，也不会重发数据。
- UDP 传送数据较 TCP 快速，系统开销也少。

基于 UDP 网络编程常用函数及功能

函数名	功能
bind	将 socket 与本机上的一个端口绑定，随后就可以在该端口监听服务请求
close	停止在该 socket 上的任何数据操作
sendto	数据发送函数
recvfrom	数据接收函数

例. 服务器端接收客户端发送的字符串。客户端将打开 liu 文件，读取文件中的三个字符串，传送给服务器端，当传送给服务器端的字符串为“stop”时，终止数据传送并断开连接。



程序中主要语句说明如下：

1. 服务器端

1) 建立 socket。程序调用 socket()函数，该函数返回一个类似于文件描述符的句柄，同时也意味着为一个 socket 数据结构分配存储空间。用语句实现：

```
socket(AF_INET,SOCK_DGRAM,0);
```

2) 绑定 bind。将 socket 与本机上的一个端口绑定，随后就可以在该端口监听服务请求。用语句实现：

```
bind(sockfd,(struct sockaddr *)&adr_inet,sizeof(adr_inet));
```

3) 接收数据 recvfrom。该函数用于数据接收。用语句实现：

```
recvfrom(sockfd,buf,0,( struct sockaddr *)&adr_clnt,&len);
```

4) 关闭 close。停止在该 socket 上的任何数据操作。用语句实现：

```
close(sockfd);
```

2. 客户端

1) 建立 socket。程序调用 socket()函数，该函数返回一个类似于文件描述符的句柄，同时也意味着为一个 socket 数据结构分配存储空间。用语句实现：

```
socket(AF_INET,SOCK_DGRAM,0);
```

2) 读取 liu 文件。读取 liu 文件里面的内容。用语句实现：

```
fopen("liu","r");
```

3) 发送数据 sendto。该函数用于在面向连接的 socket 上进行数据传输。用语句实现：

```
sendto(sockfd,buf,sizeof(buf),0,(struct sockaddr *)&adr_srvr,sizeof(adr_srvr));
```

4) 关闭 close。停止在该 socket 上的任何数据操作，用语句实现：

```
close(sockfd);
```

服务器端代码：

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * File:     server.c
 * Author: Mr Mahaitao
 *
 * Created on March 7, 2018, 12:50 AM
 */

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

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>
#include <sys/types.h>
#define PORT 8888

/*
 *
 */
int main(int argc, char** argv) {
	int sockfd;
	int len;
	int z;
	char buf[256];
	struct sockaddr_in adr_inet;
	struct sockaddr_in adr_clnt;
	printf("等待客户端....\n");

	/*建立 IP 地址*/
	adr_inet.sin_family = AF_INET;
	adr_inet.sin_addr.s_addr = htonl(INADDR_ANY); /*自动获取 IP 地址*/
	adr_inet.sin_port = PORT;
	bzero(&(adr_inet.sin_zero), 8);
	len = sizeof (adr_clnt);

	/*建立 socket*/
	sockfd = socket(AF_INET, SOCK_DGRAM, 0);
	if (sockfd == -1) {
		perror("socket  出错  ");
		exit(1);
	}

	/*绑定 socket*/
	z = bind(sockfd, (struct sockaddr *) & adr_inet, sizeof (adr_inet));
	if (z == -1) {
		perror("bind  出错  ");
		exit(1);
	}

	while (1) {
		/*接收传来的信息*/
		z = recvfrom(sockfd, buf, sizeof (buf), 0, (struct sockaddr *) &adr_clnt, &len);
		if (z < 0) {
```

```
                perror("recvfrom  出错  ");
                exit(1);
            }

            buf[z] = '\0';
            printf("接收：%s", buf);
            /*收到 stop 字符串，终止连接*/
            if (strncmp(buf, "stop", 4) == 0) {
                printf("结束....\n");
                break;
            }
        }
        close(sockfd);
        return (EXIT_SUCCESS);
}
```

客户端代码：

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * File:      client.c
 * Author: Mr Mahaitao
 *
 * Created on March 7, 2018, 2:23 AM
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>
#include <sys/types.h>
#define PORT 8888

/*
```

```
 *
 */
int main(int argc, char** argv) {
	int sockfd;
	int i = 0;
	int z;
	char buf[80], strl[80];
	struct sockaddr_in adr_srvr;
	FILE *fp;

	printf("打开文件.....\n");
	/*以只读的方式打开 liu 文件*/
	fp = fopen("liu.txt","r");
	if (fp == NULL) {
		perror("打开文件失败  ");
		exit(1);
	}

	printf("连接服务器...\n");
	/*建立 IP 地址*/
	adr_srvr.sin_family = AF_INET;
	adr_srvr.sin_addr.s_addr = htonl(INADDR_ANY);
	adr_srvr.sin_port = PORT;
	bzero(&(adr_srvr.sin_zero), 8);
	sockfd = socket(AF_INET, SOCK_DGRAM, 0);
	if (sockfd == -1) {
		perror("socket  失败");
		exit(1);
	}
	printf("发送文件....\n");
	/*读取三行数据，传给 udpserver*/
	for (i = 0; i < 3; i++) {
		fgets(strl, 80, fp);
		printf("%d:%s", i, strl);
		sprintf(buf, "%d:%s", i, strl);
		z = sendto(sockfd, buf, sizeof (buf), 0, (struct sockaddr*) &adr_srvr, sizeof (adr_srvr));
		if (z < 0) {
			perror("sendto  失败");
			exit(1);
		}
	}
	printf("发送....\n");
	sprintf(buf, "stop\n");
	z = sendto(sockfd, buf, sizeof (buf), 0, (struct sockaddr*) &adr_srvr, sizeof (adr_srvr));
```

```
    if (z < 0) {
        perror("sendto \"stop\"  失败");
        exit(1);
    }

    fclose(fp);
    close(sockfd);
    exit(0);
    return (EXIT_SUCCESS);
}
```

相关函数介绍：

sendto 函数说明如下：

所需的头文件	#include <sys/types.h> #include <sys/socket.h>
函数功能	数据发送
函数原型	int sendto(int sockfd, void *buf, int len, unsigned int flags, const struct sockaddr* to, int tolen);
函数传入值	sockfd 用来传送数据的 socket 描述符 buf　缓存器指针，用来存放要传送的信息 len　sizeof(buf) flags　一般为 0 to　接收端网络地址 tolen　sizeof(to)
函数返回值	若成功，则返回传送的位数；若失败，则返回-1

recvfrom 函数说明如下：

所需的头文件	#include <sys/types.h> #include <sys/socket.h>
函数功能	数据接收
函数原型	int recvfrom(int sockfd, void *buf, size_t len, unsigned int flags, struct sockaddr *from, socket_t *fromlen);
函数传入值	sockfd 用来传送数据的 socket 描述符 buf　缓存器指针，用来存放要传送的信息 len　sizeof(buf) flags　一般为 0 from　服务器端的网络地址 tolen　sizeof(to)
函数返回值	若成功，则返回接收的位数；若失败，则返回-1

Linux 网络高级编程

一、 高级编程简介

前面介绍了在 socket 通信中常用的函数，利用这些函数可以满足基本的 socket 通信需求；同时，也介绍了利用这些函数来编写面向连接的网络通信程序和面向无连接的网络通信程序。但是在 socket 应用中，还有一个非常重要的特性，就是任何处理阻塞，解决 I/O 多路复用问题。

在数据通信中，当服务器运行 accept()函数时，假设没有客户机连接请求到来，那么服务器就一直会停在 accept()语句上，等待客户机连接请求到来，出现这样的情况就称为阻塞。下面将具体介绍出来阻塞的方法。

二、 select 方法

1. 例 1　程序运行当中，在 10.5 秒判断有没有按下回车键，有则返回“输入了”，否则返回“超时”。

代码如下：

```
/*
* To change this license header, choose License Headers in Project Properties.
* To change this template file, choose Tools | Templates
* and open the template in the editor.
*/

/*
* File:     main.c
* Author: Mr Henry Ma
*
* Created on March 7, 2018, 4:24 AM
*/

#include <stdio.h>
```

```
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#define STDIN 0 /*标准输入设备的描述符是 0*/

/*
*
*/
int main(int argc, char** argv) {
struct timeval tv;
fd_set readfds;
tv.tv_sec = 10;
tv.tv_usec = 500000;/*微秒*/
FD_ZERO(&readfds);
FD_SET(STDIN,&readfds);
/*don't care about writefds and exceptfds*/
select(STDIN+1,&readfds,NULL,NULL,&tv);
if(FD_ISSET(STDIN,&readfds))
printf("输入了！ \n");
else
printf("超时！ \n");
return (EXIT_SUCCESS);
}
```

select 函数说明如下：

所需头文件	#include <stdio.h> #include <sys/time.h> #include <unistd.h>
函数功能	select() allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform the corresponding I/O operation (e.g., read(2)) without blocking.
函数原型	int select(int nfds, fd_set *readfds, fd_set *writefds,fd_set *exceptfds, struct timeval *timeout);
函数传入值	nfds　这个参数的值一般设置为读集合（readfds）、写集合（writefds）以及异常集合（exceptfds）中最大的描述符（fd）+1，当然也可以设置为 FD_SETSIZE。FD_SETSIZE 是操作 定义的一个宏，一般是 1024。也就是说读写以及异常集合大小的最大值为 1024，所以使用 select 最多只能管理 1024 个连接。如果大于 1024 个连接，select 将会产生不确定行为。 readfds　指向可读描述符集的指针，如果我们关心连接的可读事件，需要把连接的描述符设置到读集合中。 writefds　指向可写描述符集的指针，如果我们关心连接的可写事件，需要把连接的描述符设置到读集合中。

	exceptfds 指向异常描述符集的指针，如果我们关心连接是否发生异常，需要把连接的描述符设置到异常描述符集合中。
	如果对某一个的条件不感兴趣，就可以把它设为空指针。struct fd_set 可以理解为一个集合，这个集合中存放的是文件描述符，可通过以下四个宏进行设置：
	void FD_ZERO(fd_set *fdset);　　//清空集合
	void FD_SET(int fd, fd_set *fdset);　//将一个给定的文件描述符加入集合之中
	void FD_CLR(int fd, fd_set *fdset);　//将一个给定的文件描述符从集合中删除
	int FD_ISSET(int fd, fd_set *fdset);　// 检查集合中指定的文件描述符是否可以读写
	timeout　告知内核等待所指定描述字中的任何一个就绪可花多少时间。其 timeval 结构用于指定这段时间的秒数和微秒数。
	struct timeval {
	long tv_sec; //秒数
	long tv_usec; //微妙数
	}

函数返回值	若成功则返回就绪描述符的数目；超时返回 0；出错返回-1

使用 select 函数的过程一般是：

先调用宏 FD_ZERO 将指定的 fd_set 清零，然后调用宏 FD_SET 将需要测试的 fd 加入 fd_set，接着调用函数 select 测试 fd_set 中的所有 fd，最后用宏 FD_ISSET 检查某个 fd 在函数 select 调用后，相应位是否仍然为 1。

宏 FD_ISSET 判断描述符 fd 是否在给定的描述符集 fdset 中，通常配合 select 函数使用，由于 select 函数成功返回时会将未准备好的描述符位清零。通常我们使用 FD_ISSET 是为了检查在 select 函数返回后，某个描述符是否准备好，以便进行接下来的处理操作。

当描述符 fd 在描述符集 fdset 中返回非零值，否则，返回零。

FD_ZERO，FD_SET，FD_CLR，FD_ISSET 四个宏的具体实现：

```
/* Access macros for `fd_set'.  */
#define FD_SET(fd, fdsetp)      __FD_SET (fd, fdsetp)
#define FD_CLR(fd, fdsetp)      __FD_CLR (fd, fdsetp)
#define FD_ISSET(fd, fdsetp)    __FD_ISSET (fd, fdsetp)
#define FD_ZERO(fdsetp)         __FD_ZERO (fdsetp)
//每个 ulong 为 32 位，可以表示 32 个 bit。
```

```
//fd   >> 5  即  fd / 32，找到对应的 ulong 下标 i；fd & 31  即 fd % 32，找到在 ulong[i]内部的位置
#define __FD_SET(fd, fdsetp)     (((fd_set *)(fdsetp))->fds_bits[(fd) >> 5] |= (1<<((fd) & 31)))                    //设置对应的 bit
#define __FD_CLR(fd, fdsetp)     (((fd_set *)(fdsetp))->fds_bits[(fd) >> 5] &= ~(1<<((fd) & 31)))                   //清除对应的 bit
#define __FD_ISSET(fd, fdsetp)     ((((fd_set *)(fdsetp))->fds_bits[(fd) >> 5] & (1<<((fd) & 31))) != 0)       //判断对应的 bit 是否为 1
#define __FD_ZERO(fdsetp)     (memset (fdsetp, 0, sizeof (*(fd_set *)(fdsetp))))                                  //memset bitmap
```

fcntl 函数说明如下：

所需头文件	#include <fcntl.h>
函数功能	该函数可以改变已打开的文件的性质

函数原型	int fcntl(int fd, int cmd);
	int fcntl(int fd, int cmd, long arg);
	int fcntl(int fd, int cmd, struct flock *lock);

函数返回值	fcntl()的返回值与命令有关。如果出错，所有命令都返回－1，如果成功则返回某个其他值。下列三个命令有特定返回值：F_DUPFD , F_GETFD , F_GETFL 以及 F_GETOWN。 F_DUPFD　返回新的文件描述符 F_GETFD　返回相应标志 F_GETFL , F_GETOWN　返回一个正的进程 ID 或负的进程组 ID
备注	获得（设置）文件状态标记 cmd=F_GETFL（cmd=F_SETFL）

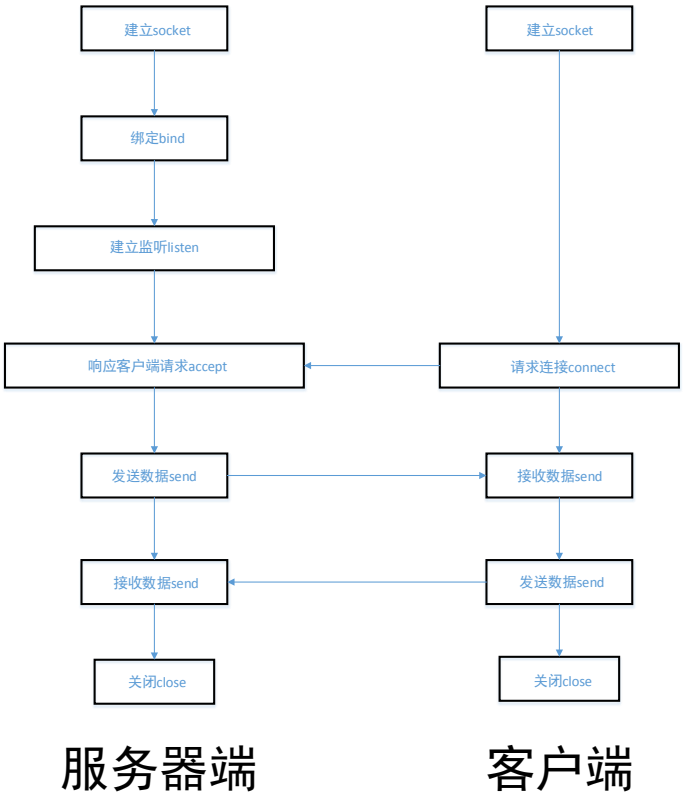
如果希望套接口不阻塞，就可以使用系统调用函数 fcntl()函数：

```
#include <unistd.h>
#include <fcntl.h>
… …
… …
sockfd = socket(AF_INET,SOCK_STREAM,0);
fcntl(sockfd,F_SETFL,O_NONBLOCK); /*设置成非阻塞*/
```

如果用此函数设置为非阻塞模式，缺点就是频繁地询问套接口，以便检查有无信息到来，这样会降低系统效率。

注意：对于 fcntl()函数里面的参数，在调用此函数作为非阻塞处理时，都用其固定参数（sockfd,F_SETFL,O_NONBLOCK）。

2. 例 2　使用 select()和 fcntl()函数编写一个网络聊天程序。



服务器端代码：

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * File:      server.c
 * Author: Henry Ma
 *
 * Created on March 8, 2018, 12:45 AM
 */

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

```
#include <fcntl.h>
#include <arpa/inet.h>

#define MAXDATASIZE 256

#define SERVERPORT 4444 /*服务器监听端口*/
#define BACKLOG 1            /*最大同时连接请求数*/
#define STDIN 0              /*标准输入文件描述符*/

int main(int argc, char *argv[]) {

    FILE *fp; //定义文件类型指针
    int sockfd, client_fd; //监听 sockfd，数据传输 sockefd
    int sin_size;
    struct sockaddr_in my_addr, remote_addr; // 本机地址信息，客户端地址信息
    char message_buff[256]; /*用于聊天的缓冲区*/

    char name_buff[256]; //用于输入用户名的缓冲区
    char send_str[256]; //最多发出去的字符数不能超过 256
    int recvbytes;
    fd_set rfd_set, wfd_set, efd_set; //被 select 监听的读，写，异常处理的文件描述符集合
    struct timeval timeout; // 本次 select 的超时结束时间
    int ret; // 与 client 连接的结果
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) { //错误检测
        perror("socket 失败");
        exit(1);
    }

    /*填充 sockaddr 结构*/
    bzero(&my_addr, sizeof (struct sockaddr_in));
    my_addr.sin_family = AF_INET; // 地址族
    my_addr.sin_port = htons(SERVERPORT); // htons 函数将一个 32 位数从主机字节顺序转换成网络字节
顺序
    inet_aton("127.0.0.1", &my_addr.sin_addr);

    if (bind(sockfd, (struct sockaddr*) &my_addr, sizeof (struct sockaddr)) == -1) { // 错误检测
        perror("bind 失败");
        exit(1);
    }

    if(listen(sockfd,BACKLOG) == -1){ //错误检测
        perror("listen 失败");
        exit(1);
    }
```

```
sin_size = sizeof (struct sockaddr_in);
if ((client_fd = accept(sockfd, (struct sockaddr *)&remote_addr, &sin_size)) == -1) { // 错误检测
        perror("accept  失败");
        exit(1);
}

fcntl(client_fd, F_SETFD, O_NONBLOCK); // 服务器设置为非阻塞
recvbytes = recv(client_fd, name_buff, MAXDATASIZE, 0);
/*接收从客户端传来的用户名*/
name_buff[recvbytes] = '\0';
fflush(stdout); //fflush(stdout)刷新标准输出缓冲区，把输出缓冲区里的东西打印到标准输出设备上
/*强制立即内容*/
if ((fp = fopen("name.txt", "a+")) == NULL) {
        printf("can not open file,exit...\n");
        return -1;
}
fprintf(fp, "%s\n", name_buff);
/*将用户名写入 name.txt 中*/
while (1) {
        FD_ZERO(&rfd_set); // 将 select()监视的读的文件描述符集合清除
        FD_ZERO(&wfd_set); // 将 select()监视的写的文件描述符集合清除
        FD_ZERO(&efd_set); // 将 select()监视的异常的文件描述符集合清除

        /*将标准输入文件描述符加到 select()监视的读的文件描述符集合中*/
        FD_SET(STDIN, &rfd_set);

        /*将新建的描述符加到 select()监视的读的文件描述符集合中*/
        FD_SET(client_fd, &rfd_set);

        /*将新建的描述符加到 select()监视的写的文件描述符集合中*/
        FD_SET(client_fd, &wfd_set);

        /*将新建的描述符加到 select()监视的异常的文件描述符集合中*/
        FD_SET(client_fd, &efd_set);

        timeout.tv_sec = 10; //select  在被监视的窗口等待的秒数
        timeout.tv_usec = 0; //select  在被监视的窗口等待的微秒数

        ret = select(client_fd + 1, &rfd_set, &wfd_set, &efd_set, &timeout);
        if (ret == 0) {
                //printf("select  超时\n");
                continue;
        }
```

```
            if (ret < 0) {
                perror("select  失败");
                exit(1);
            }
            /*判断标准输入文件描述符是否还在给定的描述符集 rfd_set 中,也就说判断是否有数据从标准输入进来*/
            if (FD_ISSET(STDIN, &rfd_set)) {
                fgets(send_str, 256, stdin); //从标准输入中取出要发送的内容
                send_str[strlen(send_str) - 1] = '\0';
                if (strncmp("quit", send_str, 4) == 0) {//  退出程序
                    //printf("deubg1\n");
                    close(client_fd);
                    close(sockfd); //关闭套接字
                    exit(0);
                }
                send(client_fd, send_str, strlen(send_str), 0);
            }

            /*判断新建的描述符是否还在给定的描述符集 wfd_set 中,也就说判断是否有数据从客户端发来*/
            if (FD_ISSET(client_fd, &rfd_set)) {
                recvbytes = recv(client_fd, message_buff, MAXDATASIZE, 0); //接收从客户端发来的聊天消息
                if (recvbytes == 0) {
                    //printf("debug2\n");
                    close(client_fd);
                    close(sockfd); //关闭套接字
                    exit(0);
                }
                message_buff[recvbytes] = '\0';
                printf("%s:%s\n", name_buff, message_buff);
                printf("Server: ");
                fflush(stdout);
            }

            /*判断新建的描述符是否还在给定的描述符集 efd_set 中,也就说判断是否有异常产生*/
            if (FD_ISSET(client_fd, &efd_set)) {
                //printf("debug3\n");
                close(client_fd);
                exit(1);
            }
        }
        return 1;
}
```

客户端代码：

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * File:      client.c
 * Author: Henry Ma
 *
 * Created on March 8, 2018, 9:16 PM
 */

#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#define SERVERPORT 4444 //服务器监听端口号
#define MAXDATASIZE 256 //最大同时连接请求
#define STDIN 0                 //标准输入文件描述符

/*
 *
 */
int main(int argc, char** argv) {

        int sockfd; //套接字描述符
        int recvbytes;
        char buf[MAXDATASIZE]; //用于处理输入的缓冲区
        char *str;

        char name[MAXDATASIZE]; //定义用户名
        char send_str[MAXDATASIZE]; //最多发出的字符不能超过 MAXDATASIZE
        struct sockaddr_in serv_addr; //Internet  套接字地址结构
        fd_set rfd_set, wfd_set, efd_set; //被 select()监视的读，写，异常处理的文件描述符集合
        struct timeval timeout; //本次 select 的超时结束时间
```

```
int ret; //与 server 连接的结果

/*创建一个 socket*/
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) { //错误检测
    perror("socket  失败");
    exit(1);
}

/*填充 sockaddr 结构*/
bzero(&serv_addr, sizeof (struct sockaddr_in));
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(SERVERPORT);
inet_aton("127.0.0.1", &serv_addr.sin_addr);

/*请求连接*/
if (connect(sockfd, (struct sockaddr*) &serv_addr, sizeof (struct sockaddr)) == -1) { //错误检测
    perror("connect  失败");
    exit(1);
}
fcntl(sockfd, F_SETFD, O_NONBLOCK);

printf("要聊天请首先输入你的名字：");
scanf("%s", name);
name[strlen(name)] = '\0';
printf("%s: ", name);
fflush(stdout);

send(sockfd, name, strlen(name), 0); //  发送用户名到 sockfd

while (1) {
    FD_ZERO(&rfd_set); //  将 select()监视的读的文件描述符集合清除
    FD_ZERO(&wfd_set); //  将 select()监视的写的文件描述符集合清除
    FD_ZERO(&efd_set); //  将 select()监视的异常的文件描述符集合清除

    /*将标准输入文件描述符加到 select()监视的读的文件描述符集合中*/
    FD_SET(STDIN, &rfd_set);

    /*将新建的描述符加到 select()监视的读的文件描述符集合中*/
    FD_SET(sockfd, &rfd_set);

    /*将新建的描述符加到 select()监视的异常的文件描述符集合中*/
    FD_SET(sockfd, &efd_set);

    timeout.tv_sec = 10; //select  在被监视的窗口等待的秒数
```

```
        timeout.tv_usec = 0; //select  在被监视的窗口等待的微秒数

        ret = select(sockfd + 1, &rfd_set, &wfd_set, &efd_set, &timeout);
        if (ret == 0) {
            //printf("select  超时\n");
            continue;
        }

        if (ret < 0) {
            perror("select error: ");
            exit(1);
        }

        /*判断标准输入文件描述符是否还在给定的描述符集 rfd_set 中,也就说判断是否有数据从标准输入进来*/
        if (FD_ISSET(STDIN, &rfd_set)) {
            fgets(send_str, 256, stdin);
            send_str[strlen(send_str) - 1] = '\0';
            if (strncmp("quit", send_str, 4) == 0) { //退出程序
                close(sockfd);
                exit(0);
            }
            send(sockfd, send_str, strlen(send_str), 0);
        }
        /*判断新建的描述符是否还在给定的描述符集 rfd_set 中,也就说判断是否有数据从服务器端发来*/
        if (FD_ISSET(sockfd, &rfd_set)) {
            recvbytes = recv(sockfd, buf, MAXDATASIZE, 0);
            if (recvbytes == 0) {
                close(sockfd);
                exit(1);
            }

            buf[recvbytes] = '\0';
            printf("Server: %s\n", buf);

            printf("%s: ", name);
            fflush(stdout);
        }

        /*判断新建的描述符是否还在给定的描述符集 efd_set 中,也就说判断是否有异常产生*/
        if (FD_ISSET(sockfd, &efd_set)) {
            close(sockfd);
            exit(1);
```

```
            }
        }
        return (EXIT_SUCCESS);
}
```

三、 poll 方法

轮询函数 poll 提供了与函数 select 相似的功能，它可以将套接口设置为非阻塞模式。但是和 select()不一样，poll()没有使用低效的三个基于位的文件描述符集合，而是采用了一个单独的结构体 pollfd 数组。

技巧：非阻塞的 I/O 模式一般采用的是轮询的方法，可以使用一个循环测试语句来实现。

poll 函数说明如下：

所需头文件	#include <poll.h>
函数功能	poll() performs a similar task to select(): it waits for one of a set of file descriptors to become ready to perform I/O.
函数原型	int poll(struct pollfd *fds, nfds_t nfds, int timeout);

函数传入值	fds：可以传递多个结构体，也就是说可以监测多个驱动设备所产生的事件，只要有一个产生了请求事件，就能立即返回。
	结构 pollfd 的定义如下：
	struct pollfd {
	int fd; /* file descriptor */
	short events; /* requested events */
	short revents; /* returned events */
	};
	结构成员 fd 表示要测试的描述符。如果不关心某个描述符，可以将 pollfd 结构中的 fd 成员设为负值。
	每一个 pollfd 结构体指定一个被监视的文件描述符，可以传递多个结构体，指示 poll()监视多个文件描述符。每个结构体的 events 域是监视该文件描述符的事件掩码，由用户来设置这个域。revents 域是文件描述符的操作结果事件掩码。内核在调用返回时设置这个域。events 域中请求的任何事件都可以在 revents 域中返回。合法的事件如下：
	POLLIN：有数据可读
	POLLRDNORM：有普通数据可读（建议用 POLLIN 代替）
	POLLRDBAND：有优先数据可读
	POLLPRI：有紧迫数据可读
	POLLOUT：写数据不会导致阻塞
	POLLWRNORM：写普通数据不会导致阻塞
	POLLMSG：SIGPOLL 消息可用
	此外，revents 域中还可以返回下列事件：

	POLLER：指定的文件描述符发生错误。 POLLHUP：指定的文件描述符挂起事件。 POLLNVAL：指定的文件描述符非法。 ********** POLLIN|POLLPRI 等价于 select()的读事件 POLLOUT|POLLWRBAND 等价于 select()的写事件 POLLIN 等价于 POLLRDNORM|POLLRDBAND POLLOUT 等价于 POLLWRNORM ********** 例如：要同时监视一个文件描述符是否可读可写，我们可以设置 events 为 POLLIN|POLLOUT。在 poll 返回时，我们可以检查 revents 中的标志，对应于文件描述符请求的 events 值。如果 POLLIN 事件被设置，则文件描述符可以被读取而不阻塞。如果 POLLOUT 被设置，则文件描述符可以写入而不导致阻塞。这些标志并不是互斥的：它们可以被同时设置，表示这个文件描述符的读取和写入操作都会正常返回而不阻塞。 nfds：要监视的描述符的数目。 timeout：超时时间，单位为 ms，它有三种情况：1.当大于 0 时，则等待指定长度的时间；2.当等于 0 时，表示立即返回；3.当为 INFTIM 时，则表示永远等待，包含在<poll.h>，也有些系统包含在<sys/stropts.h>，也有些系统没有定义，需要自己定义#define INFTIM -1。
函数返回值	如果成功，poll()将返回结构体中 revents 域不为 0 的文件描述符个数；如果超时，则返回 0；如果出错，则返回-1，并设置 errno 为下列值之一： EBADF：一个或者多个结构体中指定的文件描述符无效。 EFAULT：fds 指针指向的地址超出进程的地址空间。 EINTR：请求的事件之间产生一个信号，调用可以重新发起。 EINVAL：nfds 参数超过 PLIMIT_NOFILE 值。 ENOMEM：可用内存不足，无法完成请求。
备注	一般来说，轮询不是一个好方法，它将使程序一直处于等待状态查询套接口的数据，从而浪费大量的 CPU 的时间，并且它的实现比较麻烦。

例 1 代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h> /*文件控制*/
#include <sys/select.h>
#include <sys/time.h> /*时间方面的函数*/
#include <errno.h> /*有关错误方面的宏*/
#include<sys/poll.h> //poll()
#include<fcntl.h>
#include<string.h> //memset()

int main(int argc, char *argv[]) {
    int fd;
    int key_value;
```

```
    int ret;
    struct pollfd event; //创建一个 struct pollfd 结构体变量，存放文件描述符、要等待发生的事件
    fd = open("/dev/key", O_RDWR);
    if (fd < 0) {
        perror("open /dev/key error!\n");
        exit(1);
    }
    printf("open /dev/key sucessfully!\n");
    while (1) {   //poll 结束后 struct pollfd 结构体变量的内容被全部清零，需要再次设置
        memset(&event, 0, sizeof (event));   //memst 函数对对象的内容设置为同一值
        event.fd = fd;   //存放打开的文件描述符
        event.events = POLLIN;   //存放要等待发生的事件
        ret = poll((struct pollfd *) &event, 1, 5000);   //监测 event，一个对象，等待 5000 毫秒后超时,-1 为无限等待
        //判断 poll 的返回值，负数是出错，0 是设定的时间超时，整数表示等待的时间发生
        if (ret < 0) {
            printf("poll error!\n");
            exit(1);
        }
        if (ret == 0) {
            printf("Time out!\n");
            continue;
        }
        if (event.revents & POLLERR) {   //revents 是由内核记录的实际发生的事件，events 是进程等待的事件
            printf("Device error!\n");
            exit(1);
        }
        if (event.revents & POLLIN) {
            read(fd, &key_value, sizeof (key_value));
            printf("Key value is '%d'\n", key_value);
        }
    }
    close(fd);
    return 0;
}
```

例 2 代码：

```
#include<stdio.h>
#include<errno.h>
#include<string.h>
#include<poll.h>

int main(int argc,char *argv[]) {
```

```
    int read_fd = 0; //  标准输入描述符
    int write_fd = 1; //  标准输出描述符
    struct pollfd _poll_fd[2]; //  指定被监视的描述符，可以传递多个
    _poll_fd[0].fd = read_fd;
    _poll_fd[0].events = POLLIN; //  有数据可读
    _poll_fd[0].revents = 0; //  返回值

    _poll_fd[1].events = POLLOUT; //  写数据不会导致阻塞
    _poll_fd[1].fd = write_fd;
    _poll_fd[1].revents = 0;

    nfds_t fds = 2; //要监视的描述符的数目
    int timeout = 1000; //  定时器
    char buf[1024]; //  缓冲区
    memset(buf, '\0', sizeof (buf)); //  初始化缓冲区
    while (1) {
        switch (poll(_poll_fd, 2, timeout)) {
            case 0: // poll 返回值为 0 时表示超时
                printf("timeout\n");
                break;
            case -1: // poll 返回值为-时表示出错
                perror("poll error\n");
                break;
            default: //  否则返回准备好的描述符的个数
            {
                //int i=0;
                if ((_poll_fd[0].revents) & (POLLIN)) { //  有输入
                    ssize_t _size = read(0, buf, sizeof (buf) - 1); //  读进缓冲区
                    if (_size > 0) { //  读成功
                        buf[_size] = '\0';
                        if ((_poll_fd[1].revents) & (POLLOUT)) { //  可以写
                            printf("%s\n", buf); //  打印出来
                        }
                    }
                }
            }
                break;
        }
    }
    return 0;
}
```

例 3 服务器端代码：

```
/* server.c ---
 * Filename: server.c
 * Created: Mon Jan  8 16:45:14 2018 (+0800)
 */

/* Copyright Hamlet_Jiaxiaoyu.
 *
 * 允许免费使用，拷贝，修改，发布，但在所有的拷贝上必须保留上述
 * copyright 部分和本使用声明部分，除非显示声明，copyright 的持有者
 * 不得作为再发布软件的广告。copyright 的持有者对使用本软件的适用范围不做任何声明，
 *
 * THE COPYRIGHT HOLDERS DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
 * EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR
 * CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
 * DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
 * TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
 * OF THIS SOFTWARE.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include <netinet/in.h>
#include <sys/socket.h>
#include <poll.h>
#include <unistd.h>
#include <sys/types.h>
#include <arpa/inet.h>

#define IPADDRESS     "127.0.0.1"
#define PORT           8787
#define MAXLINE        1024
#define LISTENQ        5
#define OPEN_MAX       1000
#define INFTIM         -1

//函数声明
//创建套接字并进行绑定
static int socket_bind(const char* ip, int port);
//IO 多路复用 poll
static void do_poll(int listenfd);
//处理多个连接
```

```
static void handle_connection(struct pollfd *connfds, int num);


int main(int argc, char *argv[]) {
        int listenfd, connfd, sockfd;
        struct sockaddr_in cliaddr;
        socklen_t cliaddrlen;
        listenfd = socket_bind(IPADDRESS, PORT);
        listen(listenfd, LISTENQ);
        do_poll(listenfd);
        return 0;
}


static int socket_bind(const char* ip, int port) {
        int listenfd;
        struct sockaddr_in servaddr;
        listenfd = socket(AF_INET, SOCK_STREAM, 0);
        if (listenfd == -1) {
                perror("socket error:");
                exit(1);
        }
        bzero(&servaddr, sizeof (servaddr));
        servaddr.sin_family = AF_INET;
        inet_pton(AF_INET, ip, &servaddr.sin_addr);
        servaddr.sin_port = htons(port);
        if (bind(listenfd, (struct sockaddr*) &servaddr, sizeof (servaddr)) == -1) {
                perror("bind error: ");
                exit(1);
        }
        return listenfd;
}


static void do_poll(int listenfd) {
        int connfd, sockfd;
        struct sockaddr_in cliaddr;
        socklen_t cliaddrlen;
        struct pollfd clientfds[OPEN_MAX];
        int maxi;
        int i;
        int nready;
        //添加监听描述符
        clientfds[0].fd = listenfd;
        clientfds[0].events = POLLIN;
        //初始化客户连接描述符
        for (i = 1; i < OPEN_MAX; i++)
```

```
        clientfds[i].fd = -1;
    maxi = 0;
    //循环处理
    for (;;) {
        //获取可用描述符的个数
        nready = poll(clientfds, maxi + 1, INFTIM);
        if (nready == -1) {
            perror("poll error:");
            exit(1);
        }
        //测试监听描述符是否准备好
        if (clientfds[0].revents & POLLIN) {
            cliaddrlen = sizeof (cliaddr);
            //接受新的连接
            if ((connfd = accept(listenfd, (struct sockaddr*) &cliaddr, &cliaddrlen)) == -1) {
                if (errno == EINTR)
                    continue;
                else {
                    perror("accept error:");
                    exit(1);
                }
            }
            fprintf(stdout, "accept a new client: %s:%d\n", inet_ntoa(cliaddr.sin_addr), cliaddr.sin_port);
            //将新的连接描述符添加到数组中
            for (i = 1; i < OPEN_MAX; i++) {
                if (clientfds[i].fd < 0) {
                    clientfds[i].fd = connfd;
                    break;
                }
            }
            if (i == OPEN_MAX) {
                fprintf(stderr, "too many clients.\n");
                exit(1);
            }
            //将新的描述符添加到读描述符集合中
            clientfds[i].events = POLLIN;
            //记录客户连接套接字的个数
            maxi = (i > maxi ? i : maxi);
            if (--nready <= 0)
                continue;
        }
        //处理客户连接
        handle_connection(clientfds, maxi);
    }
```

```
}


static void handle_connection(struct pollfd *connfds, int num) {
        int i, n;
        char buf[MAXLINE];
        memset(buf, 0, MAXLINE);
        for (i = 1; i <= num; i++) {
                if (connfds[i].fd < 0)
                        continue;
                //测试客户描述符是否准备好
                if (connfds[i].revents & POLLIN) {
                        //接收客户端发送的信息
                        n = read(connfds[i].fd, buf, MAXLINE);
                        if (n == 0) {
                                close(connfds[i].fd);
                                connfds[i].fd = -1;
                                continue;
                        }
                        // printf("read msg is: ");
                        write(STDOUT_FILENO, buf, n);
                        //向客户端发送 buf
                        write(connfds[i].fd, buf, n);
                }
        }
}
```

例 3 客户端代码

```
/* client.c ---
 * Filename: client.c
 * Created: Mon Jan  8 16:45:33 2018 (+0800)
 */

/* Copyright Hamlet_Jiaxiaoyu.
 *
 * 允许免费使用，拷贝，修改，发布，但在所有的拷贝上必须保留上述
 * copyright 部分和本使用声明部分，除非显示声明，copyright 的持有者
 * 不得作为再发布软件的广告。copyright 的持有者对使用本软件的适用范围不做任何声明，
 *
 * THE COPYRIGHT HOLDERS DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
 * EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR
 * CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
```

```
 * DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
 * TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
 * OF THIS SOFTWARE.
 */
#include <netinet/in.h>
#include <sys/socket.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <poll.h>
#include <time.h>
#include <unistd.h>
#include <sys/types.h>

#define MAXLINE         1024
#define IPADDRESS      "127.0.0.1"
#define SERV_PORT      8787

#define max(a,b) (a > b) ? a : b

static void handle_connection(int sockfd);

int main(int argc, char *argv[]) {
        int sockfd;
        struct sockaddr_in servaddr;
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        bzero(&servaddr, sizeof (servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(SERV_PORT);
        inet_pton(AF_INET, IPADDRESS, &servaddr.sin_addr);
        connect(sockfd, (struct sockaddr*) &servaddr, sizeof (servaddr));
        //处理连接描述符
        handle_connection(sockfd);
        return 0;
}

static void handle_connection(int sockfd) {
        char sendline[MAXLINE], recvline[MAXLINE];
        int maxfdp, stdineof;
        struct pollfd pfds[2];
        int n;
        //添加连接描述符
        pfds[0].fd = sockfd;
        pfds[0].events = POLLIN;
```

```
    //添加标准输入描述符
    pfds[1].fd = STDIN_FILENO;
    pfds[1].events = POLLIN;
    for (;;) {
        poll(pfds, 2, -1);
        if (pfds[0].revents & POLLIN) {
            n = read(sockfd, recvline, MAXLINE);
            if (n == 0) {
                fprintf(stderr, "client: server is closed.\n");
                close(sockfd);
            }
            write(STDOUT_FILENO, recvline, n);
        }
        //测试标准输入是否准备好
        if (pfds[1].revents & POLLIN) {
            n = read(STDIN_FILENO, sendline, MAXLINE);
            if (n == 0) {
                shutdown(sockfd, SHUT_WR);
                continue;
            }
            write(sockfd, sendline, n);
        }
    }
}
```

四、 epoll 方法

为了克服以上缺点，linux2.6 内核加入了 epoll 机制。epoll 是 linux2.6 加入的用于 I/O 事件多路分离的一组函数，这组函数简化了反应式 socket 服务器的编程，并且很大程度上提高了性能。从一定程度上弥补了 Linux 内核对异步 I/O 支持的不足，epoll 机制加上非阻塞 I/O 可以模拟实现异步 I/O。

设想一下如下场景：有 100 万个客户端同时与一个服务器进程保持着 TCP 连接。而每一时刻，通常只有几百上千个 TCP 连接是活跃的(事实上大部分场景都是这种情况)。如何实现这样的高并发？

在 select/poll 时代，服务器进程每次都把这 100 万个连接告诉操作系统(从用户态复制句柄数据结构到内核态)，让操作系统内核去查询这些套接字上是否有事件发生，轮询完后，再将句柄数据复制到用户态，让服务器应用程序轮询处理已发生的网络事件，这一过程资源消耗较大，因此，select/poll 一般只能处理几千的并发连接。（从用户态复制到内核态，再从内核态复制到用户态，这一过程资源消耗很大）

epoll 的设计和实现与 select 完全不同。epoll 通过在 Linux 内核中申请一个简易的文件系统(文件系统一般用什么数据结构实现？B+树)。把原先的 select/poll 调用分成了 3 个部分：

1）调用 epoll_create()建立一个 epoll 对象(在 epoll 文件系统中为这个句柄对象分配资源)

2）调用 epoll_ctl 向 epoll 对象中添加这 100 万个连接的套接字

3）调用 epoll_wait 收集发生的事件的连接

如此一来，要实现上面说是的场景，只需要在进程启动时建立一个 epoll 对象，然后在需要的时候向这个 epoll 对象中添加或者删除连接。同时，epoll_wait 的效率也非常高，因为调用 epoll_wait 时，并没有一股脑的向操作系统复制这 100 万个连接的句柄数据，内核也不需要去遍历全部的连接。下面来看看 Linux 内核具体的 epoll 机制实现思路。

当某一进程调用 epoll_create 方法时，Linux 内核会创建一个 eventpoll 结构体，这个结构体中有两个成员与 epoll 的使用方式密切相关。eventpoll 结构体如下所示：

```
struct eventpoll{
    ....
    /*红黑树的根节点，这颗树中存储着所有添加到 epoll 中的需要监控的事件*/
    struct rb_root  rbr;
```

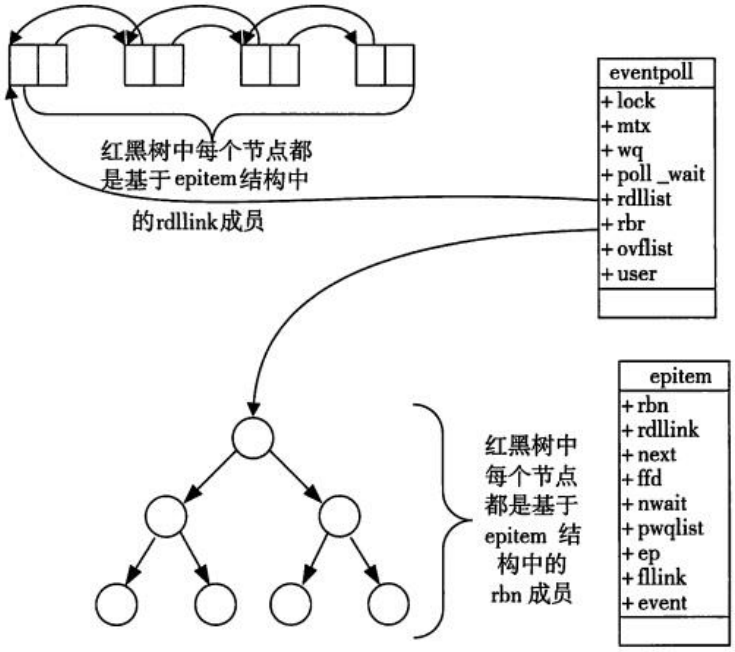
```
    /*双链表中则存放着将要通过 epoll_wait 返回给用户的满足条件的事件*/
    struct list_head rdlist;
    ....
};
```

每一个 epoll 对象都有一个独立的 eventpoll 结构体，用于存放通过 epoll_ctl 方法向 epoll 对象中添加进来的事件。这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来(红黑树的插入时间效率是 lgn，其中 n 为树的高度)。而所有添加到 epoll 中的事件都会与设备(网卡)驱动程序建立回调关系，也就是说，当相应的事件发生时会调用这个回调方法。这个回调方法在内核中叫 ep_poll_callback,它会将发生的事件添加到 rdlist 双链表中。

在 epoll 中，对于每一个事件，都会建立一个 epitem 结构体，如下所示：

```
struct epitem {
    struct rb_node    rbn;//红黑树节点
    struct list_head      rdllink;//双向链表节点
    struct epoll_filefd   ffd;   //事件句柄信息
    struct eventpoll *ep;      //指向其所属的 eventpoll 对象
    struct epoll_event event; //期待发生的事件类型
}
```

当调用 epoll_wait 检查是否有事件发生时，只需要检查 eventpoll 对象中的 rdlist 双链表中是否有 epitem 元素即可。如果 rdlist 不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户。



epoll 数据结构示意图

从上面的讲解可知：通过红黑树和双链表数据结构，并结合回调机制，造就了 epoll 的高效。

OK，讲解完了 Epoll 的机理，我们便能很容易掌握 epoll 的用法了。一句话描述就是：三步曲。

第一步：epoll_create()系统调用。此调用返回一个句柄，之后所有的使用都依靠这个句柄来标识。

第二步：epoll_ctl()系统调用。通过此调用向 epoll 对象中添加、删除、修改感兴趣的事件，返回 0 标识成功，返回-1 表示失败。

第三步：epoll_wait()系统调用。通过此调用收集在 epoll 监控中已经发生的事件。

epoll 用到的数据结构和函数：

1. 数据结构

```
typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;
```

```
struct epoll_event {
    uint32_t events;   /* Epoll events */
    epoll_data_t data;/* User data variable */
};
```

结构体 epoll_event 被用于注册所感兴趣的事件和回传所发生待处理的事件，其中 epoll_data 联合体用来保存触发事件的某个文件描述符相关的数据，例如一个 client 连接到服务器，服务器通过调用 accept 函数可以得到关于这个 client 对应的 socket 文件描述符，可以把这个文件描述符赋值给 epoll_data 的 fd 字段以便后面的读写在这个文件描述符上进行。

epoll_event 结构体的 events 字段表示感兴趣的事件和被触发的事件，可能的取值为：

events 可以是以下几个宏的集合：

EPOLLIN ：表示对应的文件描述符可以读（包括对端 SOCKET 正常关闭）；

EPOLLOUT：表示对应的文件描述符可以写；

EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；

EPOLLERR：表示对应的文件描述符发生错误；

EPOLLHUP：表示对应的文件描述符被挂断；

EPOLLET： 将 EPOLL 设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。

EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个 socket 的话，需要再次把这个 socket 加入到 EPOLL 队列里

2. 函数

1) epoll_create 函数

函数申明	int epoll_create(int size);
函数作用	该函数生成一个 epoll 专用的文件描述符，其中的参数是指定内核分配 size 个相关内核对象。

2) epoll_ctl 函数

函数申明	int epoll_ctl(int epfd,int op,int fd,struct epoll_event *event);
函数作用	该函数用于控制某个文件描述符上的事件，可以注册事件，修改事件，删除事件。
参数	epfd：由 epoll_create 生成的专用的文件描述符 op：要进行的操作，例如注册事件，可能的取值 EPOLL_CTL_ADD 注册、EPOLL_CTL_MOD 修改、EPOLL_CTL_DEL 删除。 fd：关联的文件描述符 event：指向 epoll_event 的指针
返回值	如果调用成功返回 0，不成功返回-1

3) epoll_wait 函数

函数申明	int epoll_wait(int epfd,struct epoll_event* events,int maxevents,int timeout);
函数作用	该函数用于轮询 I/O 事件的发生。
参数	epfd：由 epoll_create 生成的专用的文件描述符 events：用于回传待处理事件的数组 maxevents：每次能处理的事件数 timeout：等待 I/O 事件发生的超时值
返回值	When successful, epoll_wait() returns the number of file descriptors ready for the requested I/O, or zero if no file descriptor became ready during the requested timeout milliseconds. When an error occurs,epoll_wait() returns -1 and errno is set appropriately.

3. 实际例子

epoll 服务器端：

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

#define MAXBUF 1024
#define MAXEPOLLSIZE 10000

/*
 * 设置句柄为非阻塞方式
 */
int setnonblocking(int sockfd) {
    if (fcntl(sockfd, F_SETFL, fcntl(sockfd, F_GETFD, 0) | O_NONBLOCK) == -1) {
        return -1;
    }
    return 0;
}
/*****************************************************************************
 * filename: epoll-server.c
 * 演示 epoll 接受海量 socket 并进行处理响应的方法
```

```
 ********************************************************************/


int main(int argc, char **argv) {
        int listenfd, connfd, epfd, sockfd, nfds, n, curfds;
        socklen_t len;

        struct sockaddr_in my_addr, their_addr;
        unsigned int myport, lisnum;
        char buf[MAXBUF + 1];

        // 声明 epoll_event 结构体的变量，ev 用于注册事件，events 数组用于回传要处理的事件
        struct epoll_event ev;
          struct epoll_event events[MAXEPOLLSIZE];

        if (argv[1])
                myport = atoi(argv[1]);
        else
                myport = 7838;

        if (argv[2])
                lisnum = atoi(argv[2]);
        else
                lisnum = 2;

        // 开启 socket 监听
        if ((listenfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
                perror("socket");
                exit(1);
        } else
                printf("socket 创建成功！\n");

        // 把 socket 设置为非阻塞方式
        setnonblocking(listenfd);

        bzero(&my_addr, sizeof (my_addr));
        my_addr.sin_family = PF_INET;
        my_addr.sin_port = htons(myport);
        if (argv[3])
                my_addr.sin_addr.s_addr = inet_addr(argv[3]);
        else
                my_addr.sin_addr.s_addr = INADDR_ANY;

        if (bind(listenfd, (struct sockaddr *) &my_addr, sizeof (struct sockaddr)) == -1) {
                perror("bind");
```

```
        exit(1);
    } else
        printf("IP 地址和端口绑定成功\n");

    if (listen(listenfd, lisnum) == -1) {
        perror("listen");
        exit(1);
    } else
        printf("开启服务成功！\n");

    // 创建 epoll 句柄，把监听 socket 加入到 epoll 集合里 */
    epfd = epoll_create(MAXEPOLLSIZE); /*epoll 专用的文件描述符*/
    len = sizeof (struct sockaddr_in);
    ev.events = EPOLLIN | EPOLLET;
    ev.data.fd = listenfd;

    // 将 listenfd 注册到 epoll 事件
    if (epoll_ctl(epfd, EPOLL_CTL_ADD, listenfd, &ev) < 0) {
        fprintf(stderr, "epoll set insertion error: fd=%d\n", listenfd);
        return -1;
    } else
        printf("监听 socket 加入 epoll 成功！\n");
    curfds = 1;

    while (1) {
        // 等待有事件发生
        nfds = epoll_wait(epfd, events, curfds, -1);

        if (nfds == -1) {
            perror("epoll_wait");
            break;
        }

        // 处理所有事件
        for (n = 0; n < nfds; ++n) {
            // 如果新监测到一个 SOCKET 用户连接到了绑定的 SOCKET 端口，建立新的连接
            if (events[n].data.fd == listenfd) {     // 有新的连接过来
                len = sizeof (struct sockaddr);
                connfd = accept(listenfd, (struct sockaddr *) &their_addr, &len);   // 为新的连接创建 socket
                if (connfd < 0) {
                    perror("accept");
                    continue;
                }
                else {
```

```
                printf("有连接来自于： %s:%hu，分配的 socket 为:%d\n", inet_ntoa(their_addr.sin_addr), ntohs(their_addr.sin_port), connfd);
            }
            setnonblocking(connfd);

            // 设置用于注册的 读操作 事件
            ev.events = EPOLLIN | EPOLLET;
            // 设置用于读操作的文件描述符
            ev.data.fd = connfd;

            //注册 ev
            epoll_ctl(epfd, EPOLL_CTL_ADD, connfd, &ev);
            curfds++;
        }
        else if (events[n].events & EPOLLIN) // 如果是已经连接的用户，并且收到数据，那么进行读入
        {
            printf("EPOLLIN\n");
            if ((sockfd = events[n].data.fd) < 0)
                continue;
            int len;
            bzero(buf, MAXBUF + 1);

            /* 接收客户端的消息 */
            /*len = read(sockfd, buf, MAXBUF);*/

            len = recv(sockfd, buf, MAXBUF, 0);
            if (len > 0)
                printf("%d 接收消息成功:'%s'，共%d 个字节的数据\n", sockfd, buf, len);
            else {
                if (len < 0) {
                    printf("消息接收失败！错误代码是%d，错误信息是'%s'\n", errno, strerror(errno));
                    epoll_ctl(epfd, EPOLL_CTL_DEL, sockfd, &ev);
                    curfds--;
                    continue;
                }
            }

            // 设置用于写操作的文件描述符
            ev.data.fd = sockfd;
            // 设置用于注册的写操作事件
            ev.events = EPOLLOUT | EPOLLET;

            /*修改 sockfd 上要处理的事件为 EPOLLOUT*/
            epoll_ctl(epfd, EPOLL_CTL_MOD, sockfd, &ev); //修改标识符，等待下一个循环时发送数据，异步处理的精髓!!!!! ?????
        }
```

```
                else if (events[n].events & EPOLLOUT) //  如果有数据发送
                {
                    printf("EPOLLOUT\n");
                    sockfd = events[n].data.fd;

                    bzero(buf, MAXBUF + 1);
                    strcpy(buf, "Server already processes!");

                    send(sockfd, buf, strlen(buf), 0);

                    //  设置用于读操作的文件描述符
                    ev.data.fd = sockfd;
                    //  设置用于注册的读操作事件
                    ev.events = EPOLLIN | EPOLLET;

                    //  修改 sockfd 上要处理的事件为 EPOLIN
                    epoll_ctl(epfd, EPOLL_CTL_MOD, sockfd, &ev);
                }
            }
        }

        close(listenfd);
        return 0;
}
```

epoll 客户端：

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * File:      main.c
 * Author: root
 *
 * Created on March 10, 2018, 6:33 AM
 */

#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/socket.h>
#include <resolv.h>
```

```
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>

#define MAXBUF 1024

/**************************************************************************
 * filename: select-client.c
 * 演示网络异步通讯，这是客户端程序
 **************************************************************************/

int main(int argc, char **argv) {
    int sockfd, len;

    struct sockaddr_in dest;
    char buffer[MAXBUF + 1];

    fd_set rfds;
    struct timeval tv;

    int retval, maxfd = -1;

    if (argc != 3) {
        printf("参数格式错误！正确用法如下：\n\t\t%s IP 地址 端口\n\t 比如:\t%s 127.0.0.1 80\n 此程序用来从某个 IP 地址的服务器某个端口接收最多 MAXBUF 个字节的消息", argv[0], argv[0]);
        exit(0);
    }

    // 创建一个 socket 用于 tcp 通信
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket");
        exit(errno);
    }

    // 初始化服务器端（对方）的地址和端口信息
    bzero(&dest, sizeof (dest));
    dest.sin_family = AF_INET;
    dest.sin_port = htons(atoi(argv[2]));

    if (inet_aton(argv[1], (struct in_addr *) &dest.sin_addr.s_addr) == 0) {
        perror(argv[1]);
```

```
        exit(errno);
    }

    // 连接服务器
    if (connect(sockfd, (struct sockaddr *) &dest, sizeof (dest)) != 0) {
        perror("Connect ");
        exit(errno);
    }

    printf("\n 准备就绪，可以开始聊天了……直接输入消息回车即可发信息给对方\n");

    while (1) {
        // 把集合清空
        FD_ZERO(&rfds);
        // 把标准输入句柄 0 加入到集合中
        FD_SET(0, &rfds);

        maxfd = 0;
        // 把当前连接句柄 sockfd 加入到集合中
        FD_SET(sockfd, &rfds);

        if (sockfd > maxfd)
            maxfd = sockfd;

        // 设置最大等待时间
        tv.tv_sec = 3;
        tv.tv_usec = 0;

        // 开始等待
        retval = select(maxfd + 1, &rfds, NULL, NULL, &tv);

        if (retval == -1) {
            printf("将退出，select 出错！ %s", strerror(errno));
            break;
        }
        else if (retval == 0) {
            /* printf("没有任何消息到来，用户也没有按键，继续等待……\n"); */
            continue;
        }
        else {
            if (FD_ISSET(sockfd, &rfds)) // 连接的 socket 上有消息到来则接收对方发过来的消息并显示
            {
                bzero(buffer, MAXBUF + 1);
                // 接收对方发过来的消息，最多接收 MAXBUF 个字节
```

```
                    len = recv(sockfd, buffer, MAXBUF, 0);


                    if (len > 0) {
                        printf("接收消息成功:'%s'，共%d 个字节的数据\n", buffer, len);
                    } else {
                        if (len < 0) {
                            printf("消息接收失败！错误代码是%d，错误信息是'%s'\n", errno, strerror(errno));
                        } else {
                            printf("对方退出了，聊天终止！\n");
                        }
                        break;
                    }
                }

                if (FD_ISSET(0, &rfds)) // 用户按键了，则读取用户输入的内容发送出去
                {
                    bzero(buffer, MAXBUF + 1);
                    fgets(buffer, MAXBUF, stdin);
                    if (!strncasecmp(buffer, "quit", 4)) {
                        printf("自己请求终止聊天！\n");
                        break;
                    }

                    // 发消息给服务器
                    len = send(sockfd, buffer, strlen(buffer) - 1, 0);
                    if (len < 0) {
                        printf("消息'%s'发送失败！错误代码是%d，错误信息是'%s'\n", buffer, errno, strerror(errno));
                        break;
                    }
                    else {
                        printf("消息：%s\t 发送成功，共发送了%d 个字节！\n", buffer, len);
                    }
                }
            }
        }

        // 关闭连接
        close(sockfd);
        return 0;
}
```

五、 总结

Nginx 使用最新的 epoll（Linux2.6 内核）和 kqueue（FreeBSD）网络 I/O 模型，而 Apache 则使用的是传统的 select 模型。目前 Linux 下能够承受高并发访问的 Squid、Memcached 都采用的是 epoll 网络 I/O 模型。

处理大量连续的读写，Apache 所采用的的 select 网络 I/O 模型比较低效。下面用一个比喻来解析 Apache 采用的 select 模型和 Nginx 采用的 epoll 模型之间的区别：

假设你在大学读书，住的宿舍楼有很多间房间，你的朋友要来找你。select 版宿舍大妈就会带着你的朋友挨个房间去找，直到找到你为止。而 epoll 版宿舍大妈会先记下每位进入同学的房间号，你的朋友来时，只需要告诉你的朋友你住在那个房间即可，不必亲自带着你的朋友满宿舍楼找人。如果来了 10000 个人，都要找自己住在这栋楼的同学时，select 版和 epoll 版宿管大妈，谁的效率更高，大家应该清楚了。同理，在高并发服务器中，轮询 I/O 是最耗时间的操作之一，select 和 epoll 的性能谁更高，同样十分明了。

	select	epoll
性能	随着连接数的增加，急剧下降。处理成千上万并发连接数时，性能很差。	随着连接数增加，性能基本上没有什么下降。处理成千上万个并发连接时，性能很好。
连接数	连接数有限制，处理的最大连接数不超过 1024.如果要处理超过 1024 个连接数，则需要修改 FD_SETSIZE 宏，并重新编译。	连接数无限制。
内在处理机制	线性轮询	回调 callback

开发复杂性	低	中

几个重要的地址转换函数介绍

inet_addr(), inet_aton(), inet_ntoa()和 inet_ntop(), inet_pton()

inet_addr()函数介绍如下：

头文件	arpa/inet.h
功能	inet_addr()函数用于将点分十进制 IP 地址转换成网络字节序 IP 地址;
原型	in_addr_t inet_addr(const char *cp);
返回值	如果正确执行将返回一个无符号长整数型数。如果传入的字符串不是一个合法的 IP 地址，将返回 INADDR_NONE;

inet_aton()函数介绍如下：

头文件	sys/socket.h
功能	inet_aton()函数用于将点分十进制 IP 地址转换成网络字节序 IP 地址;
原型	int inet_aton(const char *string, struct in_addr *addr);
返回值	如果这个函数成功，函数的返回值非零，如果输入地址不正确则会返回零;

inet_ntoa()函数介绍如下：

头文件	arpa/inet.h
功能	inet_ntoa()函数用于网络字节序 IP 转化成点分十进制 IP;
原型	char *inet_ntoa (struct in_addr);
返回值	若无错误发生，inet_ntoa()返回一个字符指针。否则的话，返回 NULL。其中的数据应在下一个 WINDOWS 套接口调用前复制出来;

应用实例：

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
int main()
{
    int i;
```

```
    char lo[] = "127.0.0.1";
    struct in_addr netAddr;
    netAddr.s_addr = inet_addr(lo);
    printf("NetIP: 0x%x\n", netAddr.s_addr);
    char *strAddr = inet_ntoa(netAddr);
    printf("StrIP: %s\n", strAddr);
    int ret = inet_aton(strAddr, &netAddr);
    printf("NetIP: 0x%x\n", netAddr.s_addr);
    return 0;
}
```

输出结果：

```
NetIP: 0x100007f(网络序)
StrIP: 127.0.0.1
NetIP: 0x100007f(网络序)
```

inet_ntop()函数介绍如下：

头文件	arpa/inet.h
功能	inet_ntop()函数用于将网络字节序的二进制地址转换成文本字符串;
原型	const char *inet_pton(int domain, const void *restrict addr, char *restrict str, socklen_t size);
返回值	若成功，返回地址字符串指针；若出错，返回 NULL;

inet_pton()函数介绍如下：

头文件	arpa/inet.h
功能	inet_pton()函数用于将文本字符串格式转换成网络字节序二进制地址;
原型	int inet_pton(int domain, const char *restrict str, void *restrict addr);
返回值	若成功，返回 1；若格式无效，返回 0；若出错，返回-1;

应用实例：

```
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>
int main()
{
    struct in_addr addr;
```

```
    if(inet_pton(AF_INET, "127.0.0.1", &addr.s_addr) == 1)
        printf("NetIP: %x\n", addr.s_addr);
    char str[20];
    if(inet_ntop(AF_INET, &addr.s_addr, str, sizeof str))
        printf("StrIP: %s\n", str);
    return 0;
}
```

输出结果：

```
NetIP: 100007f
StrIP: 127.0.0.1
```