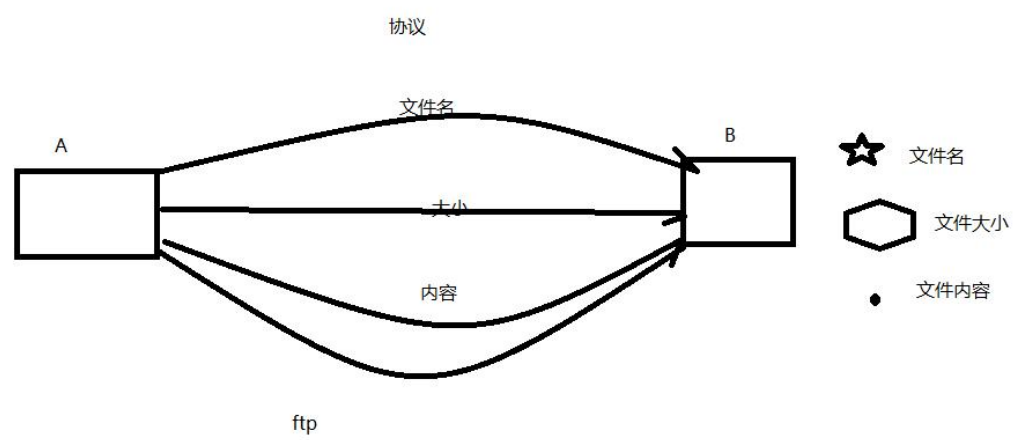


01P-复习-Linux 网络编程

02P-信号量生产者复习

03P-协议

协议：
一组规则。



04P-7 层模型和 4 层模型及代表协议

分层模型结构：

OSI 七层模型： 物、数、网、传、会、表、应

TCP/IP 4 层模型：网（链路层/网络接口层）、网、传、应

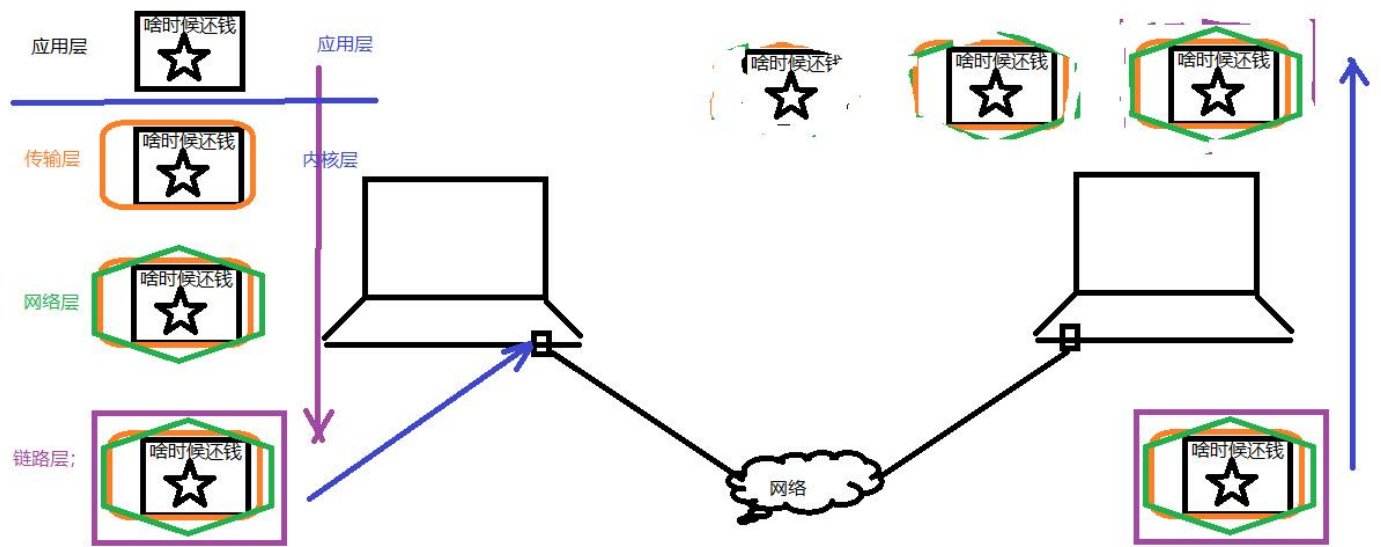
应用层：http、ftp、nfs、ssh、telnet。。。.

传输层：TCP、UDP

网络层：IP、ICMP、IGMP

链路层：以太网帧协议、ARP

05P-网络传输数据封装流程

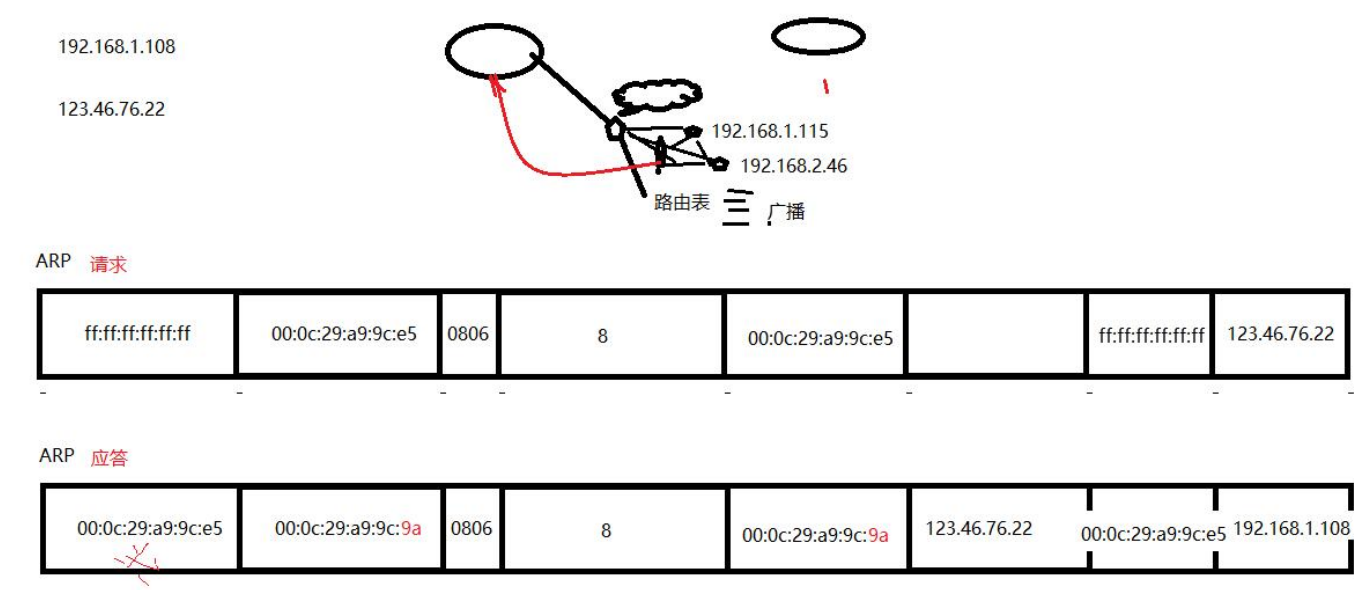


网络传输流程：

数据没有封装之前，是不能在网络中传递。

数据-》应用层-》传输层-》网络层-》链路层 —— 网络环境

06P-以太网帧和 ARP 请求



以太网帧协议：

ARP 协议：根据 Ip 地址获取 mac 地址。

以太网帧协议：根据 mac 地址，完成数据包传输。

07P-IP 协议

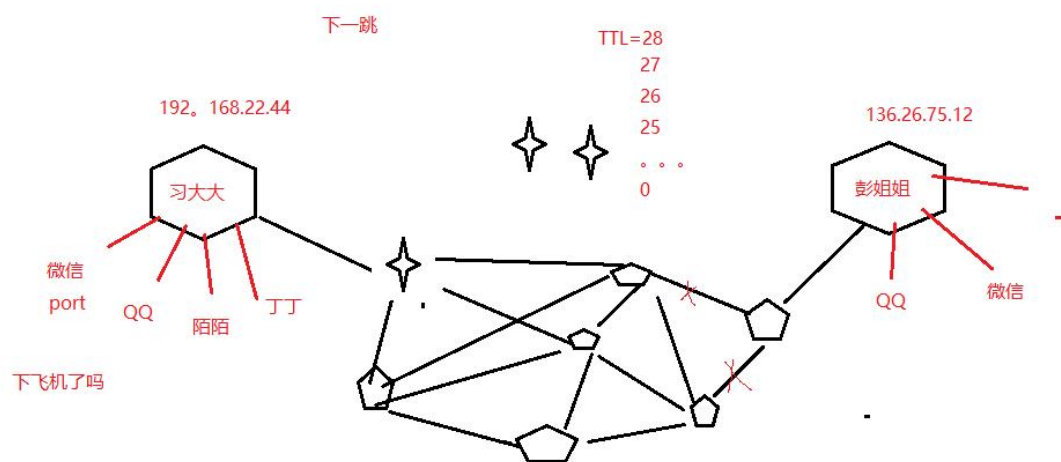
IP 协议:

版本: IPv4、IPv6 -- 4 位

TTL: time to live 。 设置数据包在路由节点中的跳转上限。每经过一个路由节点, 该值-1, 减为 0 的路由, 有义务将该数据包丢弃

源 IP: 32 位。--- 4 字节 192.168.1.108 --- 点分十进制 IP 地址 (string) --- 二进制

目的 IP: 32 位。--- 4 字节

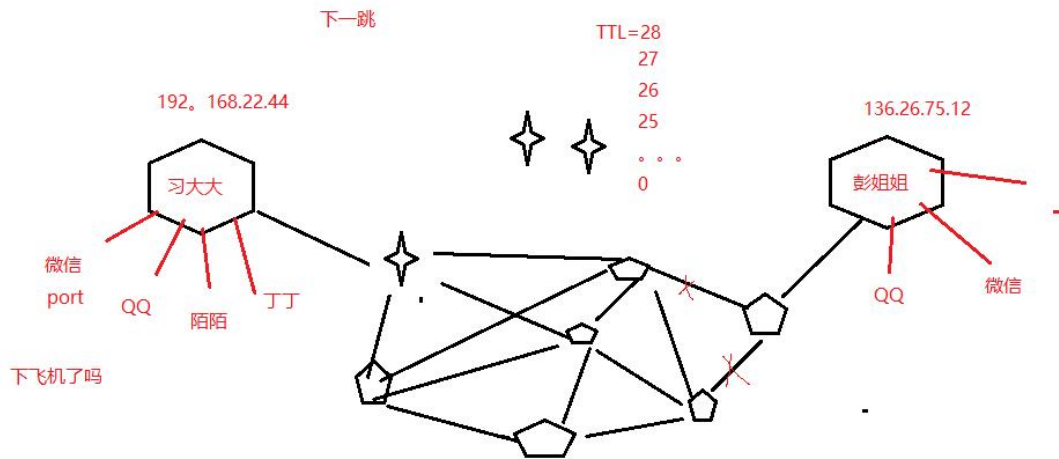


08P-端口号和 UDP 协议

UDP:

16 位：源端口号。 $2^{16} = 65536$

16 位：目的端口号。



IP 地址：可以在网络环境中，唯一标识一台主机。

端口号：可以网络的一台主机上，唯一标识一个进程。

ip 地址+端口号：可以在网络环境中，唯一标识一个进程。

09P-TCP 协议

TCP 协议：

16 位：源端口号。 $2^{16} = 65536$

16 位：目的端口号。

32 序号；

32 确认序号。

6 个标志位。

16 位窗口大小。 $2^{16} = 65536$

10P-BS 和 CS 模型对比

c/s 模型:

client-server

b/s 模型:

browser-server

C/S

B/S

优点: 缓存大量数据、协议选择灵活

安全性、跨平台、开发工作量较小

速度快

缺点: 安全性、跨平台、开发工作量较大

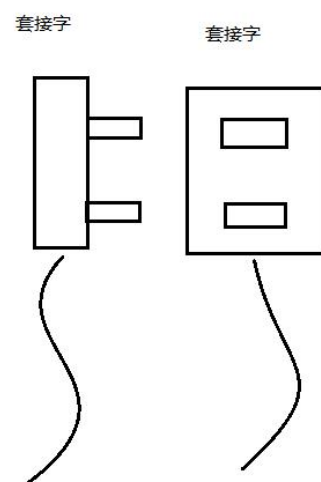
不能缓存大量数据、严格遵守 http

11P-套接字

网络套接字： socket

一个文件描述符指向一个套接字（该套接字内部由内核借助两个缓冲区实现。）

在通信过程中， 套接字一定是成对出现的。



12P-回顾

13P-网络字节序

网络字节序：

小端法：（pc 本地存储） 高位存高地址。地位存低地址。 `int a = 0x12345678`

大端法：（网络存储） 高位存低地址。地位存高地址。

`htonl --> 本地--> 网络 (IP)` `192.168.1.11 --> string --> atoi --> int -->`
`htonl --> 网络字节序`

`htons --> 本地--> 网络 (port)`

`ntohl --> 网络--> 本地 (IP)`

`ntohs --> 网络--> 本地 (Port)`

14P-IP 地址转换函数

IP 地址转换函数：

`int inet_pton(int af, const char *src, void *dst);` 本地字节序 (string IP) ---> 网络字节序

af: AF_INET、AF_INET6

src: 传入, IP 地址 (点分十进制)

dst: 传出, 转换后的 网络字节序的 IP 地址。

返回值:

成功: 1

异常: 0, 说明 src 指向的不是一个有效的 ip 地址。

失败: -1

`const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);` 网络字节序 ---> 本地字节序 (string IP)

af: AF_INET、AF_INET6

src: 网络字节序 IP 地址

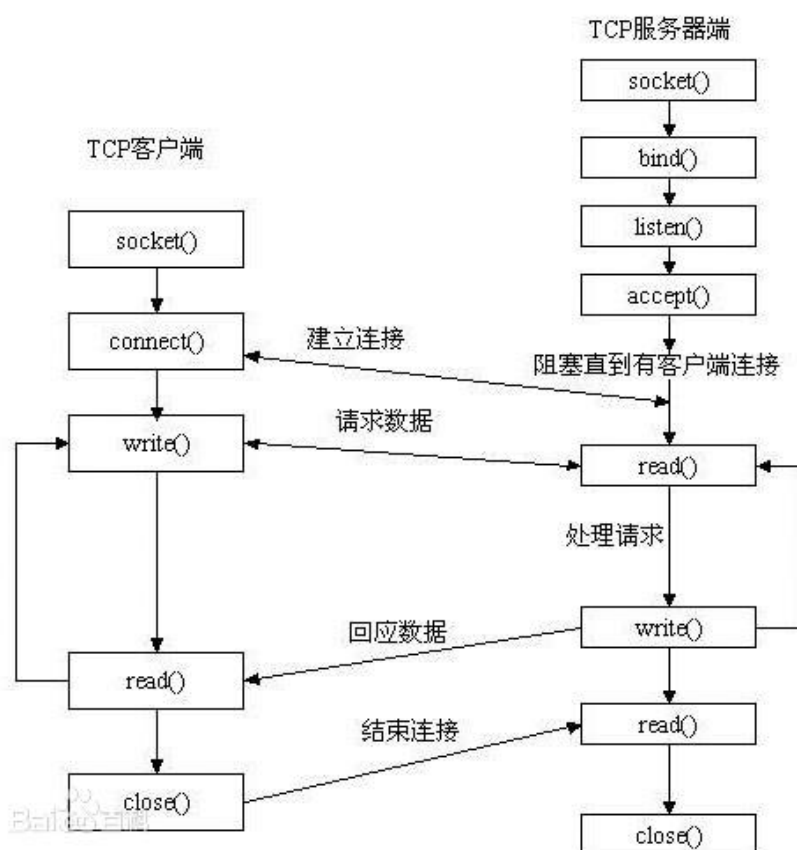
dst: 本地字节序 (string IP)

size: dst 的大小。

返回值: 成功: dst。

失败: NULL

15P-sockaddr 地址结构



sockaddr 地址结构: IP + port --> 在网络环境中唯一标识一个进程。

```
struct sockaddr_in addr;
```

```
addr.sin_family = AF_INET/AF_INET6          man 7 ip
```

```
addr.sin_port = htons(9527);
```

```
int dst;
```

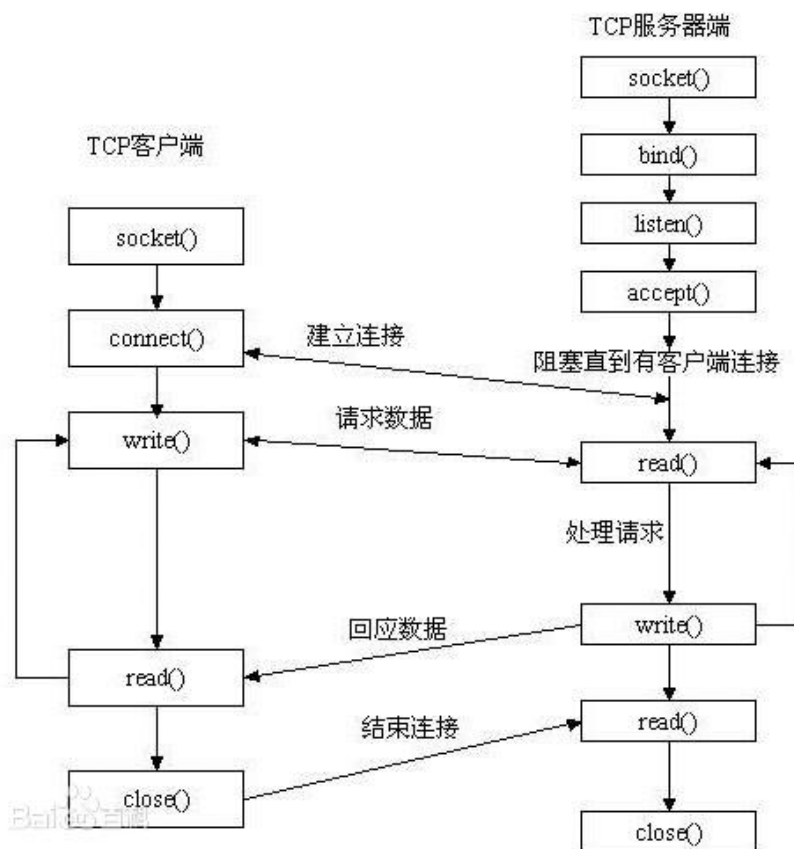
```
inet_pton(AF_INET, "192.157.22.45", (void *)&dst);
```

```
addr.sin_addr.s_addr = dst;
```

【*】 `addr.sin_addr.s_addr = htonl(INADDR_ANY);` 取出系统中有效的任意 IP 地址。二进制类型。

```
bind(fd, (struct sockaddr *)&addr, size);
```

16P-socket 模型创建流程分析



17P-socket 和 bind

socket 函数:

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

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

 创建一个 套接字

domain: AF_INET、AF_INET6、AF_UNIX

type: SOCK_STREAM、SOCK_DGRAM

protocol: 0

返回值:

成功: 新套接字所对应文件描述符

失败: -1 errno

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

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

 给
socket 绑定一个 地址结构 (IP+port)

sockfd: socket 函数返回值

```
struct sockaddr_in addr;
```

```
addr.sin_family = AF_INET;
```

```
addr.sin_port = htons(8888);
```

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

addr: 传入参数(struct sockaddr *)&addr

addrlen: sizeof(addr) 地址结构的大小。

返回值:

成功: 0

失败: -1 errno

18P-listen 和 accept

`int listen(int sockfd, int backlog);` 设置同时与服务器建立连接的上限数。（同时进行 3 次握手的客户端数量）

sockfd: socket 函数返回值

backlog: 上限数值。最大值 128.

返回值:

成功: 0

失败: -1 errno

`int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);` 阻塞等待客户端建立连接，成功的话，返回一个与客户端成功连接的 socket 文件描述符。

sockfd: socket 函数返回值

addr: 传出参数。成功与服务器建立连接的那个客户端的地址结构（IP+port）

`socklen_t clit_addr_len = sizeof(addr);`

addrlen: 传入传出。 &clit_addr_len

入: addr 的大小。 出: 客户端 addr 实际大小。

返回值:

成功: 能与客户端进行数据通信的 socket 对应的文件描述。

失败: -1 , errno

19P-connect

`int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);` 使用现有的 socket 与服务器建立连接

sockfd: socket 函数返回值

```
struct sockaddr_in srv_addr;            // 服务器地址结构
```

```
srv_addr.sin_family = AF_INET;
```

```
srv_addr.sin_port = 9527      跟服务器 bind 时设定的 port 完全一致。
```

```
inet_pton(AF_INET, "服务器的 IP 地址", &srv_addr.sin_addr.s_addr);
```

addr: 传入参数。服务器的地址结构

addrlen: 服务器的地址结构的大小

返回值:

成功: 0

失败: -1 errno

如果不使用 bind 绑定客户端地址结构, 采用“隐式绑定”。

20P-CS 模型的 TCP 通信分析

TCP 通信流程分析：

server:

1. `socket()` 创建 socket
2. `bind()` 绑定服务器地址结构
3. `listen()` 设置监听上限
4. `accept()` 阻塞监听客户端连接
5. `read(fd)` 读 socket 获取客户端数据
6. 小--大写 `toupper()`
7. `write(fd)`
8. `close()`;

client:

1. `socket()` 创建 socket
2. `connect()`; 与服务器建立连接
3. `write()` 写数据到 socket
4. `read()` 读转换后的数据。
5. 显示读取结果
6. `close()`

21P-server 的实现

代码如下：

```
1. #include <stdio.h>
2. #include <ctype.h>
3. #include <sys/socket.h>
4. #include <arpa/inet.h>
5. #include <stdlib.h>
6. #include <string.h>
7. #include <unistd.h>
8. #include <errno.h>
9. #include <pthread.h>
10.
11. #define SERV_PORT 9527
12.
13.
14. void sys_err(const char *str)
15. {
16.     perror(str);
17.     exit(1);
18. }
19.
20. int main(int argc, char *argv[])
21. {
22.     int lfd = 0, cfd = 0;
23.     int ret, i;
24.     char buf[BUFSIZ], client_IP[1024];
25.
26.     struct sockaddr_in serv_addr, clit_addr; // 定义服务器地址结构 和 客户端地址结构
27.     socklen_t clit_addr_len;                // 客户端地址结构大小
28.
29.     serv_addr.sin_family = AF_INET;          // IPv4
30.     serv_addr.sin_port = htons(SERV_PORT);   // 转为网络字节序的 端口号
31.     serv_addr.sin_addr.s_addr = htonl(INADDR_ANY); // 获取本机任意有效 IP
32.
33.     lfd = socket(AF_INET, SOCK_STREAM, 0);    // 创建一个 socket
34.     if (lfd == -1) {
35.         sys_err("socket error");
36.     }
37.
38.     bind(lfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)); // 给服务器 socket 绑定地址结构 (IP+port)
39.
40.     listen(lfd, 128);                          // 设置监听上限
41.
42.     clit_addr_len = sizeof(clit_addr);         // 获取客户端地址结构大小
43.
44.     cfd = accept(lfd, (struct sockaddr *)&clit_addr, &clit_addr_len); // 阻塞等待客户端连接请求
```

```

45.     if (cfd == -1)
46.         sys_err("accept error");
47.
48.     printf("client ip:%s port:%d\n",
49.           inet_ntop(AF_INET, &clit_addr.sin_addr.s_addr, client_IP, sizeof(client_IP)),
50.           ntohs(clit_addr.sin_port));           // 根据 accept 传出参数, 获取客户端 ip 和 port
51.
52.     while (1) {
53.         ret = read(cfd, buf, sizeof(buf));       // 读客户端数据
54.         write(STDOUT_FILENO, buf, ret);         // 写到屏幕查看
55.
56.         for (i = 0; i < ret; i++)               // 小写 -- 大写
57.             buf[i] = toupper(buf[i]);
58.
59.         write(cfd, buf, ret);                   // 将大写, 写回给客户端。
60.     }
61.
62.     close(lfd);
63.     close(cfd);
64.
65.     return 0;
66. }

```

编译测试, 结果如下:

<pre> zhcode@ubuntu:~/network/code21\$ make server gcc server.c -o server -Wall -g zhcode@ubuntu:~/network/code21\$./server client ip:127.0.0.1 port:33010 megumin is my wife </pre>	<pre> zhcode@ubuntu:~\$ nc 127.0.0.1 9527 megumin is my wife MEGUMIN IS MY WIFE </pre>
---	--

22P-获取客户端地址结构

```
cfd = accept(lfd, (struct sockaddr *)&clit_addr, &clit_addr_len);
```

accept 函数中的 clit_addr 传出的就是客户端地址结构，IP+port

于是，在代码中增加此段代码，可获取客户端信息：

```
printf("client ip:%s port:%d\n",  
       inet_ntop(AF_INET,&clit_addr.sin_addr.s_addr, client_IP, sizeof(client_IP)),  
       ntohs(clit_addr.sin_port));
```

上一节代码中已经有这段代码，这里就不再跑一遍了。

23P-client 的实现

```
1. #include <stdio.h>
2. #include <sys/socket.h>
3. #include <arpa/inet.h>
4. #include <stdlib.h>
5. #include <string.h>
6. #include <unistd.h>
7. #include <errno.h>
8. #include <pthread.h>
9.
10. #define SERV_PORT 9527
11.
12. void sys_err(const char *str)
13. {
14.     perror(str);
15.     exit(1);
16. }
17.
18. int main(int argc, char *argv[])
19. {
20.     int cfd;
21.     int conter = 10;
22.     char buf[BUFSIZ];
23.
24.     struct sockaddr_in serv_addr;           //服务器地址结构
25.
26.     serv_addr.sin_family = AF_INET;
27.     serv_addr.sin_port = htons(SERV_PORT);
28.     //inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr.s_addr);
29.     inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);
30.
31.     cfd = socket(AF_INET, SOCK_STREAM, 0);
32.     if (cfd == -1)
33.         sys_err("socket error");
34.
35.     int ret = connect(cfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
36.     if (ret != 0)
37.         sys_err("connect err");
38.
39.     while (--conter) {
40.         write(cfd, "hello\n", 6);
41.         ret = read(cfd, buf, sizeof(buf));
42.         write(STDOUT_FILENO, buf, ret);
43.         sleep(1);
44.     }
45.
46.     close(cfd);
```



```
47.  
48.     return 0;  
49. }
```

编译运行，结果如下：

```
zhcode@ubuntu:~/network/code21$ ./server zhcode@ubuntu:~/network/code21$ ./client  
client ip:127.0.0.1 port:33030 HELLO  
hello HELLO  
hello HELLO  
hello HELLO  
hello HELLO  
hello HELLO  
hello HELLO  
hello HELLO  
hello HELLO  
hello  
[
```

这里遇到过一个问题，如果之前运行 server，用 Ctrl+z 终止进程，ps aux 列表里会有服务器进程残留，这个会影响当前服务器。解决方法是 kill 掉这些服务器进程。不然端口被占用，当前运行的服务器进程接收不到东西，没有回显。

24P-总结

协议：

一组规则。

分层模型结构：

OSI 七层模型： 物、数、网、传、会、表、应

TCP/IP 4 层模型：网（链路层/网络接口层）、网、传、应

应用层：http、ftp、nfs、ssh、telnet。。。

传输层：TCP、UDP

网络层：IP、ICMP、IGMP

链路层：以太网帧协议、ARP

c/s 模型：

client-server

b/s 模型：

browser-server

C/S

B/S

优点： 缓存大量数据、协议选择灵活

安全性、跨平台、开发工作量较小

速度快

缺点： 安全性、跨平台、开发工作量较大

不能缓存大量数据、严格遵守 http

网络传输流程：

数据没有封装之前，是不能在网络中传递。

数据-》应用层-》传输层-》网络层-》链路层 —— 网络环境

以太网帧协议：

ARP 协议：根据 Ip 地址获取 mac 地址。

以太网帧协议：根据 mac 地址，完成数据包传输。

IP 协议：

版本： IPv4、IPv6 -- 4 位

TTL: time to live 。 设置数据包在路由节点中的跳转上限。每经过一个路由节点，该值-1，减为 0 的路由，有义务将该数据包丢弃

源 IP: 32 位。--- 4 字节 192.168.1.108 --- 点分十进制 IP 地址 (string) --- 二进制

目的 IP: 32 位。--- 4 字节

IP 地址：可以在网络环境中，唯一标识一台主机。

端口号：可以网络的一台主机上，唯一标识一个进程。

ip 地址+端口号：可以在网络环境中，唯一标识一个进程。

UDP:

16 位：源端口号。 $2^{16} = 65536$

16 位：目的端口号。

TCP 协议:

16 位：源端口号。 $2^{16} = 65536$

16 位：目的端口号。

32 序号;

32 确认序号。

6 个标志位。

16 位窗口大小。 $2^{16} = 65536$

网络套接字: socket

一个文件描述符指向一个套接字（该套接字内部由内核借助两个缓冲区实现。）

在通信过程中，套接字一定是成对出现的。

网络字节序：

小端法：（pc 本地存储） 高位存高地址。低位存低地址。 `int a = 0x12345678`

大端法：（网络存储） 高位存低地址。低位存高地址。

`htonl` --> 本地--> 网络 （IP） `192.168.1.11` --> `string` --> `atoi` --> `int` -->
`htonl` --> 网络字节序

`htons` --> 本地--> 网络 (port)

`ntohl` --> 网络--> 本地 (IP)

`ntohs` --> 网络--> 本地 (Port)

IP 地址转换函数：

`int inet_pton(int af, const char *src, void *dst);` 本地字节序 (string IP) --->
网络字节序

af: AF_INET、AF_INET6

src: 传入，IP 地址（点分十进制）

dst: 传出，转换后的 网络字节序的 IP 地址。

返回值：

成功： 1

异常： 0， 说明 src 指向的不是一个有效的 ip 地址。

失败： -1

`const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);` 网
网络字节序 ---> 本地字节序 (string IP)

af: AF_INET、AF_INET6

src: 网络字节序 IP 地址

dst: 本地字节序 (string IP)

size: dst 的大小。

返回值： 成功： dst。

失败： NULL

sockaddr 地址结构: IP + port --> 在网络环境中唯一标识一个进程。

```
struct sockaddr_in addr;
```

```
addr.sin_family = AF_INET/AF_INET6          man 7 ip
```

```
addr.sin_port = htons(9527);
```

```
int dst;
```

```
inet_pton(AF_INET, "192.157.22.45", (void *)&dst);
```

```
addr.sin_addr.s_addr = dst;
```

【*】 addr.sin_addr.s_addr = htonl(INADDR_ANY); 取出系统中有效的任意 IP 地址。二进制类型。

```
bind(fd, (struct sockaddr *)&addr, size);
```

socket 函数:

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

```
int socket(int domain, int type, int protocol);      创建一个 套接字
```

domain: AF_INET、AF_INET6、AF_UNIX

type: SOCK_STREAM、SOCK_DGRAM

protocol: 0

返回值:

成功: 新套接字所对应文件描述符

失败: -1 errno

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

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);      给  
socket 绑定一个 地址结构 (IP+port)
```

sockfd: socket 函数返回值

```
struct sockaddr_in addr;
```

```
addr.sin_family = AF_INET;
```

```
addr.sin_port = htons(8888);
```

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

addr: 传入参数(struct sockaddr *)&addr

addrlen: sizeof(addr) 地址结构的大小。

返回值:

成功: 0

失败: -1 errno

```
int listen(int sockfd, int backlog);
```

设置同时与服务器建立连接的上限数。(同时进行 3 次握手的客户端数量)

sockfd: socket 函数返回值

backlog: 上限数值。最大值 128.

返回值:

成功: 0

失败: -1 errno

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

阻塞等待客户端建立连接, 成功的话, 返回一个与客户端成功连接的 socket 文件描述符。

sockfd: socket 函数返回值

addr: 传出参数。成功与服务器建立连接的那个客户端的地址结构 (IP+port)

```
socklen_t clit_addr_len = sizeof(addr);
```

addrlen: 传入传出。 &clit_addr_len

入: addr 的大小。 出: 客户端 addr 实际大小。

返回值:

成功: 能与客户端进行数据通信的 socket 对应的文件描述。

失败: -1 , errno

`int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);` 使用
现有的 `socket` 与服务器建立连接

`sockfd`: `socket` 函数返回值

`struct sockaddr_in srv_addr;` // 服务器地址结构

`srv_addr.sin_family = AF_INET;`

`srv_addr.sin_port = 9527` 跟服务器 `bind` 时设定的 `port` 完全一致。

`inet_pton(AF_INET, "服务器的 IP 地址", &srv_addr.sin_addr.s_addr);`

`addr`: 传入参数。服务器的地址结构

`addrlen`: 服务器的地址结构的大小

返回值:

成功: 0

失败: -1 `errno`

如果不使用 `bind` 绑定客户端地址结构, 采用“隐式绑定”。

TCP 通信流程分析:

server:

1. `socket()` 创建 `socket`
2. `bind()` 绑定服务器地址结构
3. `listen()` 设置监听上限
4. `accept()` 阻塞监听客户端连接
5. `read(fd)` 读 `socket` 获取客户端数据
6. 小一大写 `toupper()`
7. `write(fd)`
8. `close();`

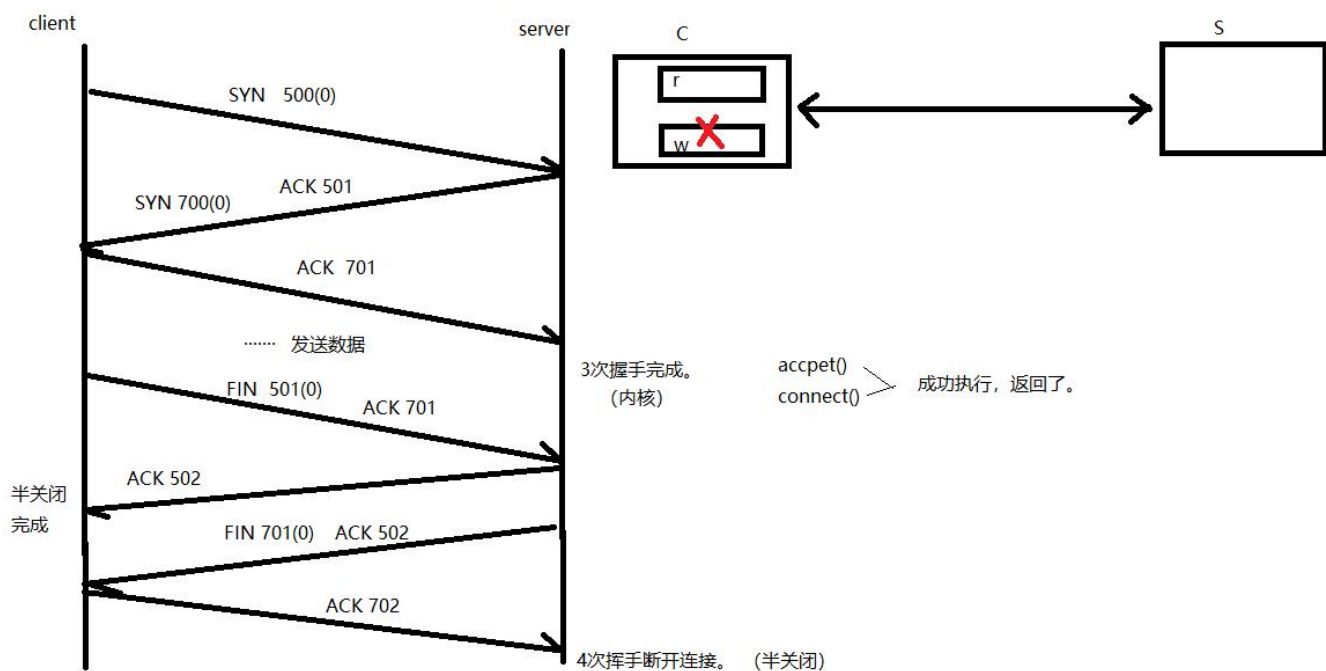
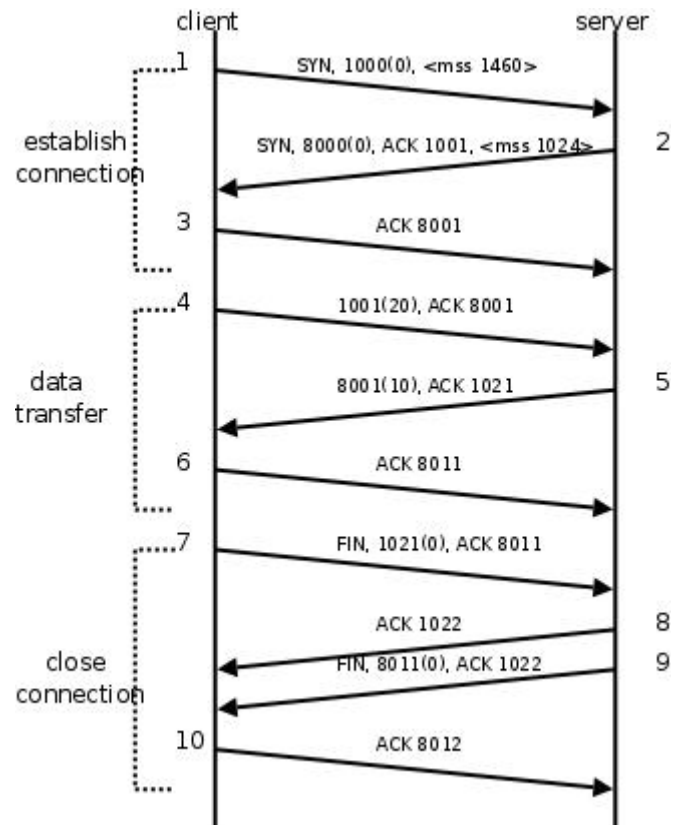
client:

1. `socket()` 创建 `socket`

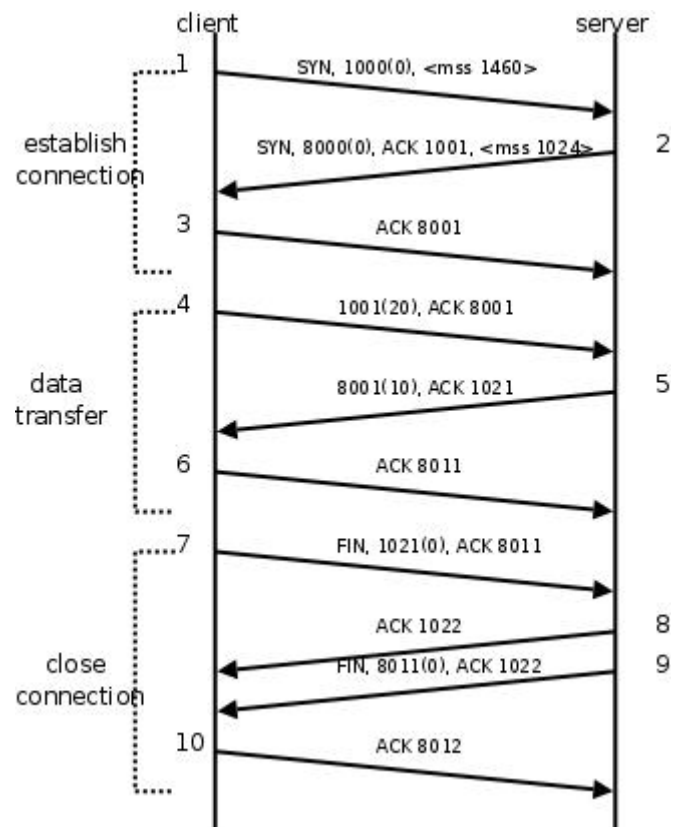
2. `connect()`; 与服务器建立连接
3. `write()` 写数据到 socket
4. `read()` 读转换后的数据。
5. 显示读取结果
6. `close()`

25P-复习

26P-三次握手建立连接

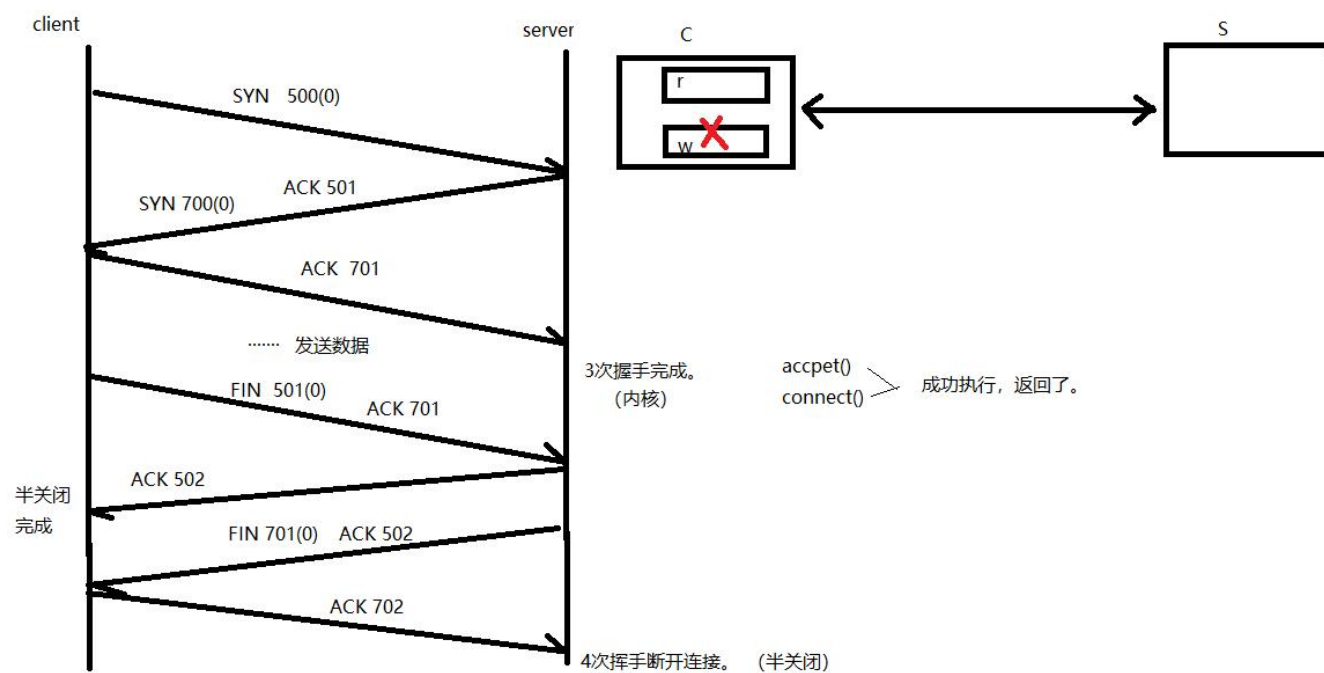


27P-数据通信



并不是一次发送，一次应答。也可以批量应答

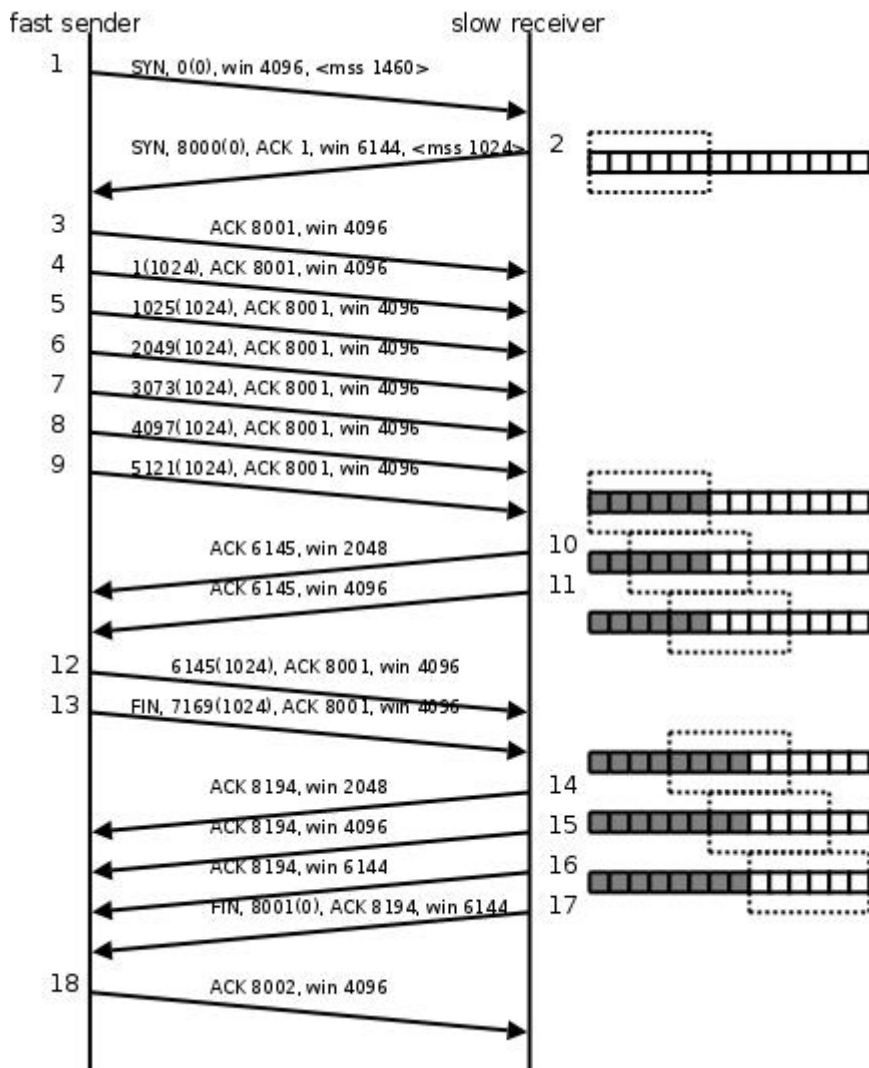
28P-四次握手关闭连接



29P-半关闭补充说明

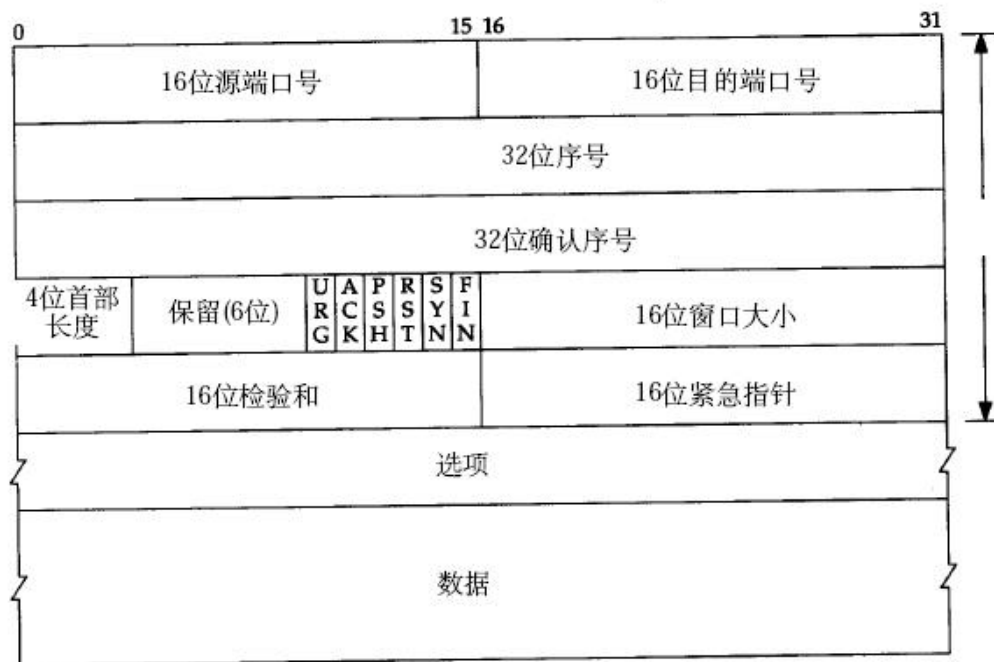
这里其实就是想说明，完成两次挥手后，不是说两端的连接断开了，主动端关闭了写缓冲区，不能再向对端发送数据，被动端关闭了读缓冲区，不能再从对端读取数据。然而主动端还是能够读取对端发来的数据。

30P-滑动窗口和TCP 数据包格式

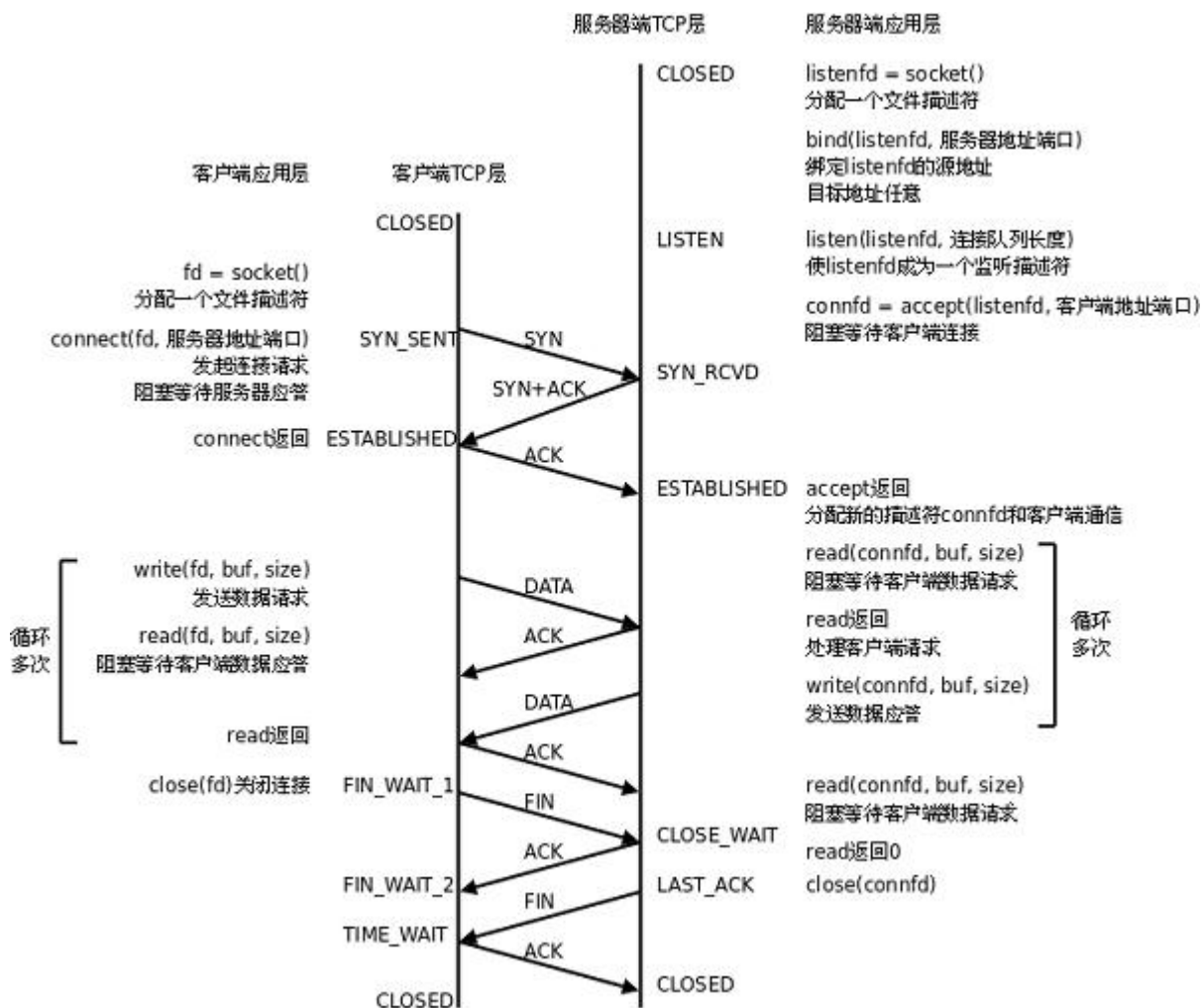


滑动窗口：

发送给连接对端，本端的缓冲区大小（实时），保证数据不会丢失。



31P-通信时序与代码对应关系



32P-TCP 通信时序总结

三次握手：

主动发起连接请求端，发送 SYN 标志位，请求建立连接。携带序号号、数据字节数(0)、滑动窗口大小。

被动接受连接请求端，发送 ACK 标志位，同时携带 SYN 请求标志位。携带序号、确认序号、数据字节数(0)、滑动窗口大小。

主动发起连接请求端，发送 ACK 标志位，应答服务器连接请求。携带确认序号。

四次挥手：

主动关闭连接请求端，发送 FIN 标志位。

被动关闭连接请求端，应答 ACK 标志位。----- 半关闭完成。

被动关闭连接请求端，发送 FIN 标志位。

主动关闭连接请求端，应答 ACK 标志位。----- 连接全部关闭

滑动窗口：

发送给连接对端，本端的缓冲区大小（实时），保证数据不会丢失。

33P-错误处理函数的封装思路

wrap.h 文件如下，就是包裹函数的声明

```
1. #ifndef __WRAP_H_
2. #define __WRAP_H_
3.
4. void perr_exit(const char *s);
5. int Accept(int fd, struct sockaddr *sa, socklen_t *salenptr);
6. int Bind(int fd, const struct sockaddr *sa, socklen_t salen);
7. int Connect(int fd, const struct sockaddr *sa, socklen_t salen);
8. int Listen(int fd, int backlog);
9. int Socket(int family, int type, int protocol);
10. ssize_t Read(int fd, void *ptr, size_t nbytes);
11. ssize_t Write(int fd, const void *ptr, size_t nbytes);
12. int Close(int fd);
13. ssize_t Readn(int fd, void *vptr, size_t n);
14. ssize_t Writen(int fd, const void *vptr, size_t n);
15. ssize_t my_read(int fd, char *ptr);
16. ssize_t Readline(int fd, void *vptr, size_t maxlen);
17.
18. #endif
```

wrap.c 随便取一部分，如下，就是包裹函数的代码：

```
1. #include <stdlib.h>
2. #include <stdio.h>
3. #include <unistd.h>
4. #include <errno.h>
5. #include <sys/socket.h>
6.
7. void perr_exit(const char *s)
8. {
9.     perror(s);
10.    exit(-1);
11. }
12.
13. int Accept(int fd, struct sockaddr *sa, socklen_t *salenptr)
14. {
15.     int n;
16.
17.    again:
18.    if ((n = accept(fd, sa, salenptr)) < 0) {
19.        if ((errno == ECONNABORTED) || (errno == EINTR))
20.            goto again;
21.        else
22.            perr_exit("accept error");
23.    }
24.    return n;
```

```
25. }
26.
27. int Bind(int fd, const struct sockaddr *sa, socklen_t salen)
28. {
29.     int n;
30.
31.     if ((n = bind(fd, sa, salen)) < 0)
32.         perr_exit("bind error");
33.
34.     return n;
35. }
```

这里原函数和包裹函数的函数名差异只有首字母大写，这是因为 man page 对字母大小写不敏感，同名的包裹函数一样可以跳转至 man page

34P-错误处理函数封装

就是重新包裹需要检查返回值的函数，让代码不那么肥胖。

35P-封装思想总结和 readn、readline 封装思想说明

错误处理函数：

封装目的：

在 server.c 编程过程中突出逻辑，将出错处理与逻辑分开，可以直接跳转 man 手册。

【wrap.c】

【wrap.h】

存放网络通信相关常用 自定义函数
型(声明)。

存放 网络通信相关常用 自定义函数原

命名方式：系统调用函数首字符大写，方便查看 man 手册

如：Listen()、Accept()；

函数功能：调用系统调用函数，处理出错场景。

在 server.c 和 client.c 中调用 自定义函数

联合编译 server.c 和 wrap.c 生成 server

client.c 和 wrap.c 生成 client

readn:

读 N 个字节

readline:

读一行

36P-中午复习

三次握手：

主动发起连接请求端，发送 SYN 标志位，请求建立连接。携带序号号、数据字节数(0)、滑动窗口大小。

被动接受连接请求端，发送 ACK 标志位，同时携带 SYN 请求标志位。携带序号、确认序号、数据字节数(0)、滑动窗口大小。

主动发起连接请求端，发送 ACK 标志位，应答服务器连接请求。携带确认序号。

四次挥手：

主动关闭连接请求端， 发送 FIN 标志位。

被动关闭连接请求端， 应答 ACK 标志位。 ----- 半关闭完成。

被动关闭连接请求端， 发送 FIN 标志位。

主动关闭连接请求端， 应答 ACK 标志位。 ----- 连接全部关闭

滑动窗口：

发送给连接对端，本端的缓冲区大小（实时），保证数据不会丢失。

错误处理函数：

封装目的：

在 server.c 编程过程中突出逻辑，将出错处理与逻辑分开，可以直接跳转 man 手册。

【wrap.c】

【wrap.h】

存放网络通信相关常用 自定义函数
型(声明)。

存放 网络通信相关常用 自定义函数原

命名方式：系统调用函数首字符大写，方便查看 man 手册

如：Listen()、Accept()；

函数功能：调用系统调用函数，处理出错场景。

在 server.c 和 client.c 中调用 自定义函数

联合编译 server.c 和 wrap.c 生成 server

client.c 和 wrap.c 生成 client

readn:

读 N 个字节

readline:

读一行

read 函数的返回值:

1. > 0 实际读到的字节数
2. $= 0$ 已经读到结尾（对端已经关闭）【！重！点！】
3. -1 应进一步判断 errno 的值:

errno = EAGAIN or EWOULDBLOCK: 设置了非阻塞方式 读。 没有数据到达。

errno = EINTR 慢速系统调用被 中断。

errno = “其他情况” 异常。

37P-多进程并发服务器思路分析

```
1. Socket();      创建 监听套接字 lfd
2. Bind()  绑定地址结构 Strcut scokaddr_in addr;
3. Listen();
4. while (1) {

    cfd = Accpet();      接收客户端连接请求。
    pid = fork();
    if (pid == 0) {      子进程 read(cfd) --- 小-》大 --- write(cfd)

        close(lfd)      关闭用于建立连接的套接字 lfd

        read()
        小--大
        write()

    } else if (pid > 0) {

        close(cfd);      关闭用于与客户端通信的套接字 cfd
        contiue;
    }
}
```

5. 子进程:

```
close(lfd)

read()

小--大

write()
```

父进程:

```
close(cfd);
```

注册信号捕捉函数: SIGCHLD

在回调函数中, 完成子进程回收

```
while (waitpid());
```


38P-多线程并发服务器分析

多线程并发服务器： server.c

1. Socket(); 创建 监听套接字 lfd

2. Bind() 绑定地址结构 Strcut sockaddr_in addr;

3. Listen();

4. while (1) {

 cfd = Accept(lfd,);

 pthread_create(&tid, NULL, tfn, (void *)cfd);

 pthread_detach(tid); // pthread_join(tid, void **); 新线程---专用于回收子线程。
}

5. 子线程:

void *tfn(void *arg)

{

 // close(lfd) 不能关闭。 主线程要使用 lfd

 read(cfd)

 小--大

 write(cfd)

 pthread_exit ((void *)10);

}

39P-多进程并发服务器实现

第一个版本的代码如下：

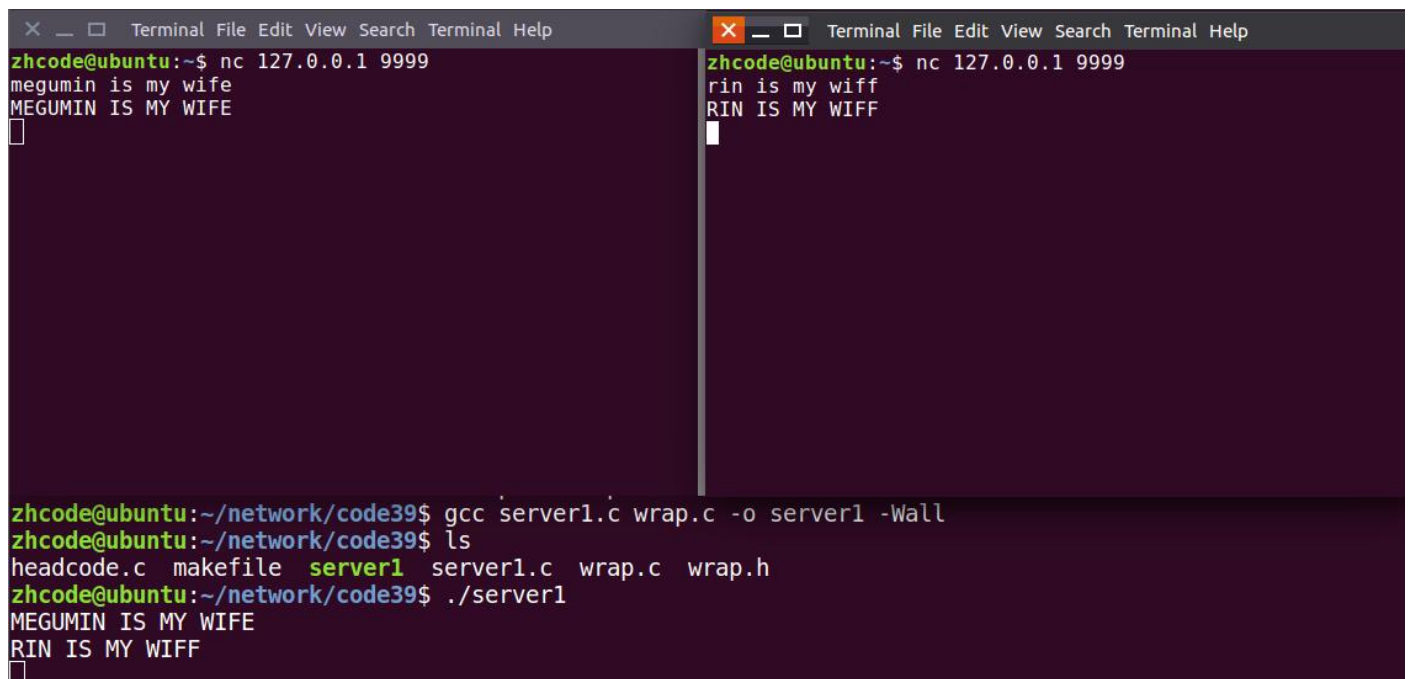
```
1. #include <stdio.h>
2. #include <ctype.h>
3. #include <stdlib.h>
4. #include <sys/wait.h>
5. #include <string.h>
6. #include <strings.h>
7. #include <unistd.h>
8. #include <errno.h>
9. #include <signal.h>
10. #include <sys/socket.h>
11. #include <arpa/inet.h>
12. #include <pthread.h>
13.
14. #include "wrap.h"
15.
16. #define SRV_PORT 9999
17.
18. int main(int argc, char *argv[])
19. {
20.     int lfd, cfd;
21.     pid_t pid;
22.     struct sockaddr_in srv_addr, clt_addr;
23.     socklen_t clt_addr_len;
24.     char buf[BUFSIZ];
25.     int ret, i;
26.
27.     //memset(&srv_addr, 0, sizeof(srv_addr));           // 将地址结构清零
28.     bzero(&srv_addr, sizeof(srv_addr));
29.
30.     srv_addr.sin_family = AF_INET;
31.     srv_addr.sin_port = htons(SRV_PORT);
32.     srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
33.
34.     lfd = Socket(AF_INET, SOCK_STREAM, 0);
35.
36.     Bind(lfd, (struct sockaddr *)&srv_addr, sizeof(srv_addr));
37.
38.     Listen(lfd, 128);
39.
40.     clt_addr_len = sizeof(clt_addr);
41.
42.     while (1) {
43.
44.         cfd = Accept(lfd, (struct sockaddr *)&clt_addr, &clt_addr_len);
45.
```

```

46.     pid = fork();
47.     if (pid < 0) {
48.         perr_exit("fork error");
49.     } else if (pid == 0) {
50.         close(lfd);
51.         break;
52.     } else {
53.         close(cfd);
54.         continue;
55.     }
56. }
57.
58. if (pid == 0) {
59.     for (;;) {
60.         ret = Read(cfd, buf, sizeof(buf));
61.         if (ret == 0) {
62.             close(cfd);
63.             exit(1);
64.         }
65.
66.         for (i = 0; i < ret; i++)
67.             buf[i] = toupper(buf[i]);
68.
69.         write(cfd, buf, ret);
70.         write(STDOUT_FILENO, buf, ret);
71.     }
72. }
73.
74. return 0;
75. }

```

编译运行，结果如下：



```

zhcode@ubuntu:~$ nc 127.0.0.1 9999
megumin is my wife
MEGUMIN IS MY WIFE

```

```

zhcode@ubuntu:~$ nc 127.0.0.1 9999
rin is my wiff
RIN IS MY WIFF

```

```

zhcode@ubuntu:~/network/code39$ gcc server1.c wrap.c -o server1 -Wall
zhcode@ubuntu:~/network/code39$ ls
headcode.c  makefile  server1  server1.c  wrap.c  wrap.h
zhcode@ubuntu:~/network/code39$ ./server1
MEGUMIN IS MY WIFE
RIN IS MY WIFF

```

这个代码，有问题。我们Ctrl+C 终止一个连接进程，会发现，有僵尸进程。

```
zhcode@ubuntu:~$ nc 127.0.0.1 9999
megumin is my wife
MEGUMIN IS MY WIFE
^C
zhcode@ubuntu:~$ ps -ef
USER          PID  PPID  FFD  ST   TTY          TIME CMD
root           1      0   0  SL   s1    0:00 /usr/lib/x86_64-linux
root           1      0   0  SL   s1    0:06 /usr/bin/python /usr/
root          1770    1   0  SL   s1    0:00 update-notifier
root          1770    1   0  SL   s1    0:00 /usr/lib/x86_64-linux
root           1      0   0  Ss   s1    0:00 /usr/sbin/cupsd -l
root           1      0   0  Ssl  s1    0:00 /usr/sbin/cups-browse
root          2803    1   0  S    s1    0:00 /usr/lib/cups/notifie
root           2      0   0  S    s1    0:00 [kworker/u256:0]
root          2183    1   0  S+   s1    0:00 ./server1
root          2172    1   0  Ss   s1    0:00 bash
root          2172    1   0  Ss+  s1    0:00 bash
root          2920    1   0  S+   s1    0:00 nc 127.0.0.1 9999
root          2912    1   0  S+   s1    0:00 ./server1
root          2912    1   0  Z+   s1    0:00 [server1] <defunct>
root           2      0   0  S    s1    0:00 [kworker/0:1]
root           2      0   0  S    s1    0:00 [kworker/u256:2]
root           2      0   0  S    s1    0:00 [kworker/0:2]
root          2172    1   0  Ss   s1    0:00 bash
root          3047    1   0  R+   s1    0:00 ps ajx
```

如上图所示，有个僵尸进程。这是因为父进程在阻塞等待，没来得及去回收这个子进程。

所以需要修改代码，增加子进程回收，用信号捕捉来实现。

修改部分如图所示：

```
58     } else {
59         struct sigaction act;
60
61         act.sa_handler = catch_child;
62         sigemptyset(&act.sa_mask);
63         act.sa_flags = 0;
64
65         ret = sigaction(SIGCHLD, &act, NULL);
66         if (ret != 0) {
67             perr_exit("sigaction error");
68         }
69         close(cfd);
70         continue;
71     }
72
73 void catch_child(int signum)
74 {
75     while ((waitpid(0, NULL, WNOHANG)) > 0);
76     return ;
77 }
```

完整代码如下：

1. #include <stdio.h>
2. #include <ctype.h>
3. #include <stdlib.h>
4. #include <sys/wait.h>
5. #include <string.h>
6. #include <strings.h>
7. #include <unistd.h>

```
8. #include <errno.h>
9. #include <signal.h>
10. #include <sys/socket.h>
11. #include <arpa/inet.h>
12. #include <pthread.h>
13.
14. #include "wrap.h"
15.
16. #define SRV_PORT 9999
17.
18. void catch_child(int signum)
19. {
20.     while ((waitpid(0, NULL, WNOHANG)) > 0);
21.     return ;
22. }
23.
24. int main(int argc, char *argv[])
25. {
26.     int lfd, cfd;
27.     pid_t pid;
28.     struct sockaddr_in srv_addr, clt_addr;
29.     socklen_t clt_addr_len;
30.     char buf[BUFSIZ];
31.     int ret, i;
32.
33.     //memset(&srv_addr, 0, sizeof(srv_addr));           // 将地址结构清零
34.     bzero(&srv_addr, sizeof(srv_addr));
35.
36.     srv_addr.sin_family = AF_INET;
37.     srv_addr.sin_port = htons(SRV_PORT);
38.     srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
39.
40.     lfd = Socket(AF_INET, SOCK_STREAM, 0);
41.
42.     Bind(lfd, (struct sockaddr *)&srv_addr, sizeof(srv_addr));
43.
44.     Listen(lfd, 128);
45.
46.     clt_addr_len = sizeof(clt_addr);
47.
48.     while (1) {
49.
50.         cfd = Accept(lfd, (struct sockaddr *)&clt_addr, &clt_addr_len);
51.
52.         pid = fork();
53.         if (pid < 0) {
54.             perr_exit("fork error");
55.         } else if (pid == 0) {
56.             close(lfd);
```

```

57.         break;
58.     } else {
59.         struct sigaction act;
60.
61.         act.sa_handler = catch_child;
62.         sigemptyset(&act.sa_mask);
63.         act.sa_flags = 0;
64.
65.         ret = sigaction(SIGCHLD, &act, NULL);
66.         if (ret != 0) {
67.             perr_exit("sigaction error");
68.         }
69.         close(cfd);
70.         continue;
71.     }
72. }
73.
74. if (pid == 0) {
75.     for (;;) {
76.         ret = Read(cfd, buf, sizeof(buf));
77.         if (ret == 0) {
78.             close(cfd);
79.             exit(1);
80.         }
81.
82.         for (i = 0; i < ret; i++)
83.             buf[i] = toupper(buf[i]);
84.
85.         write(cfd, buf, ret);
86.         write(STDOUT_FILENO, buf, ret);
87.     }
88. }
89.
90. return 0;
91. }

```

这样，当子进程退出时，父进程收到信号，就会去回收子进程了，不会出现僵尸进程。

40P-多进程服务器测试 IP 地址调整

使用桥接模式，让自己主机和其他人主机处于同一个网段

41P-服务器程序上传外网服务器并访问

scp -r 命令，将本地文件拷贝至远程服务器上目标位置

scp -r 源地址 目标地址

42P-多线程服务器代码 review

代码如下：

```
1. #include <stdio.h>
2. #include <string.h>
3. #include <arpa/inet.h>
4. #include <pthread.h>
5. #include <ctype.h>
6. #include <unistd.h>
7. #include <fcntl.h>
8.
9. #include "wrap.h"
10.
11. #define MAXLINE 8192
12. #define SERV_PORT 8000
13.
14. struct s_info {                                //定义一个结构体，将地址结构跟 cfd 捆绑
15.     struct sockaddr_in cliaddr;
16.     int connfd;
17. };
18.
19. void *do_work(void *arg)
20. {
21.     int n,i;
22.     struct s_info *ts = (struct s_info*)arg;
23.     char buf[MAXLINE];
24.     char str[INET_ADDRSTRLEN];                //define INET_ADDRSTRLEN 16 可用"+d"查看
25.
26.     while (1) {
27.         n = Read(ts->connfd, buf, MAXLINE);    //读客户端
28.         if (n == 0) {
29.             printf("the client %d closed...\n", ts->connfd);
30.             break;                            //跳出循环,关闭 cfd
31.         }
32.         printf("received from %s at PORT %d\n",
33.             inet_ntop(AF_INET, &(ts->cliaddr.sin_addr, str, sizeof(str)),
34.             ntohs((ts->cliaddr.sin_port));      //打印客户端信息(IP/PORT)
35.
36.         for (i = 0; i < n; i++)
37.             buf[i] = toupper(buf[i]);          //小写-->大写
38.
39.         Write(STDOUT_FILENO, buf, n);          //写出至屏幕
40.         Write(ts->connfd, buf, n);             //回写给客户端
41.     }
42.     Close(ts->connfd);
43.
44.     return (void *)0;
45. }
```

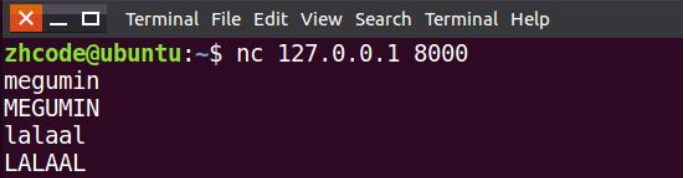
```

46.
47. int main(void)
48. {
49.     struct sockaddr_in servaddr, cliaddr;
50.     socklen_t cliaddr_len;
51.     int listenfd, connfd;
52.     pthread_t tid;
53.
54.     struct s_info ts[256];    //创建结构体数组.
55.     int i = 0;
56.
57.     listenfd = Socket(AF_INET, SOCK_STREAM, 0);    //创建一个 socket, 得到 lfd
58.
59.     bzero(&servaddr, sizeof(servaddr));    //地址结构清零
60.     servaddr.sin_family = AF_INET;
61.     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);    //指定本地任意 IP
62.     servaddr.sin_port = htons(SERV_PORT);    //指定端口号
63.
64.     Bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));    //绑定
65.
66.     Listen(listenfd, 128);    //设置同一时刻链接服
        务器上限数
67.
68.     printf("Accepting client connect ...\n");
69.
70.     while (1) {
71.         cliaddr_len = sizeof(cliaddr);
72.         connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);    //阻塞监听客户端链接
        请求
73.         ts[i].cliaddr = cliaddr;
74.         ts[i].connfd = connfd;
75.
76.         pthread_create(&tid, NULL, do_work, (void*)&ts[i]);
77.         pthread_detach(tid);    //子线程分离,防止僵
        线程产生.
78.         i++;
79.     }
80.
81.     return 0;
82. }

```

编译运行，结果如下：

```
zhcode@ubuntu:~/network/code39$ gcc server_thread.c wrap.c -o server_thread -Wall -lpthread
zhcode@ubuntu:~/network/code39$ ./server_thread
Accepting client connect ...
received from 127.0.0.1 at PORT 41960
MEGUMIN
received from 127.0.0.1 at PORT 41960
LALAAL
█
```



```
zhcode@ubuntu:~$ nc 127.0.0.1 8000
megumin
MEGUMIN
lalaal
LALAAL
```

43P-read 返回值和总结

三次握手：

主动发起连接请求端，发送 SYN 标志位，请求建立连接。携带序号、数据字节数(0)、滑动窗口大小。

被动接受连接请求端，发送 ACK 标志位，同时携带 SYN 请求标志位。携带序号、确认序号、数据字节数(0)、滑动窗口大小。

主动发起连接请求端，发送 ACK 标志位，应答服务器连接请求。携带确认序号。

四次挥手：

主动关闭连接请求端， 发送 FIN 标志位。

被动关闭连接请求端， 应答 ACK 标志位。 ----- 半关闭完成。

被动关闭连接请求端， 发送 FIN 标志位。

主动关闭连接请求端， 应答 ACK 标志位。 ----- 连接全部关闭

滑动窗口：

发送给连接对端，本端的缓冲区大小（实时），保证数据不会丢失。

错误处理函数：

封装目的：

在 server.c 编程过程中突出逻辑，将出错处理与逻辑分开，可以直接跳转 man 手册。

【wrap.c】

【wrap.h】

存放网络通信相关常用 自定义函数
型(声明)。

存放 网络通信相关常用 自定义函数原

命名方式：系统调用函数首字符大写，方便查看 man 手册

如：Listen()、Accept()；

函数功能：调用系统调用函数，处理出错场景。

在 server.c 和 client.c 中调用 自定义函数

联合编译 server.c 和 wrap.c 生成 server

client.c 和 wrap.c 生成 client

readn:

读 N 个字节

readline:

读一行

read 函数的返回值:

1. > 0 实际读到的字节数
2. $= 0$ 已经读到结尾（对端已经关闭）【！重！点！】
3. -1 应进一步判断 errno 的值:

errno = EAGAIN or EWOULDBLOCK: 设置了非阻塞方式 读。 没有数据到达。

errno = EINTR 慢速系统调用被 中断。

errno = “其他情况” 异常。

多进程并发服务器: server.c

1. Socket(); 创建 监听套接字 lfd
2. Bind() 绑定地址结构 Strcut sockaddr_in addr;
3. Listen();
4. while (1) {

 cfd = Accept(); 接收客户端连接请求。
 pid = fork();
 if (pid == 0) { 子进程 read(cfd) --- 小 --- 大 --- write(cfd)

 close(lfd) 关闭用于建立连接的套接字 lfd

 read()
 小--大
 write()

 } else if (pid > 0) {

```

        close(cfd);          关闭用于与客户端通信的套接字 cfd
        continue;
    }
}

```

5. 子进程:

```
close(lfd)
```

```
read()
```

小--大

```
write()
```

父进程:

```
close(cfd);
```

注册信号捕捉函数: SIGCHLD

在回调函数中, 完成子进程回收

```
while (waitpid());
```

多线程并发服务器: server.c

1. Socket(); 创建 监听套接字 lfd

2. Bind() 绑定地址结构 struct sockaddr_in addr;

3. Listen();

4. while (1) {

```
    cfd = Accept(lfd, );
```

```
    pthread_create(&tid, NULL, tfn, (void *)cfd);
```

```
    pthread_detach(tid);          // pthread_join(tid, void **); 新线程---专用于
回收子线程。
}
```

5. 子线程:

```
void *tfn(void *arg)
{
```

```
// close(lfd)          不能关闭。 主线程要使用 lfd  
  
read(cfd)  
  
小--大  
  
write(cfd)  
  
pthread_exit ((void *)10) ;  
}
```

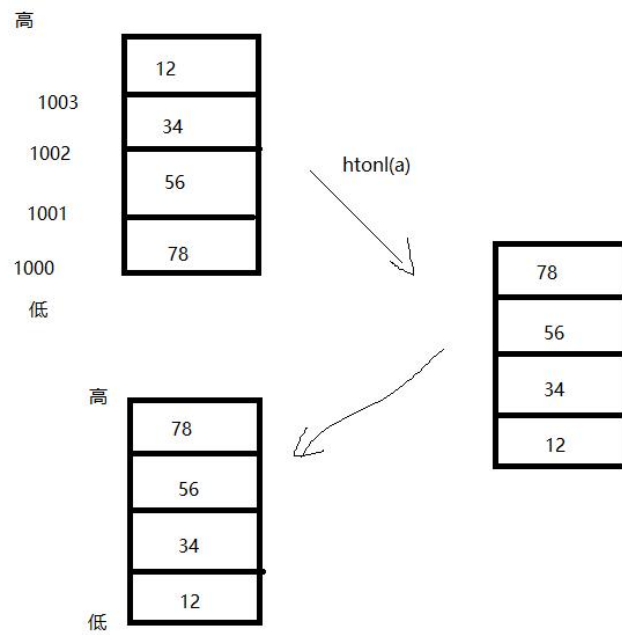
44P-复习

本地字节序： `int a = 0x12345678`

高存高，低存低

网络字节序：

高存低，低存高



45P-TCP 状态-主动发起连接

46P-TCP 状态-主动关闭连接

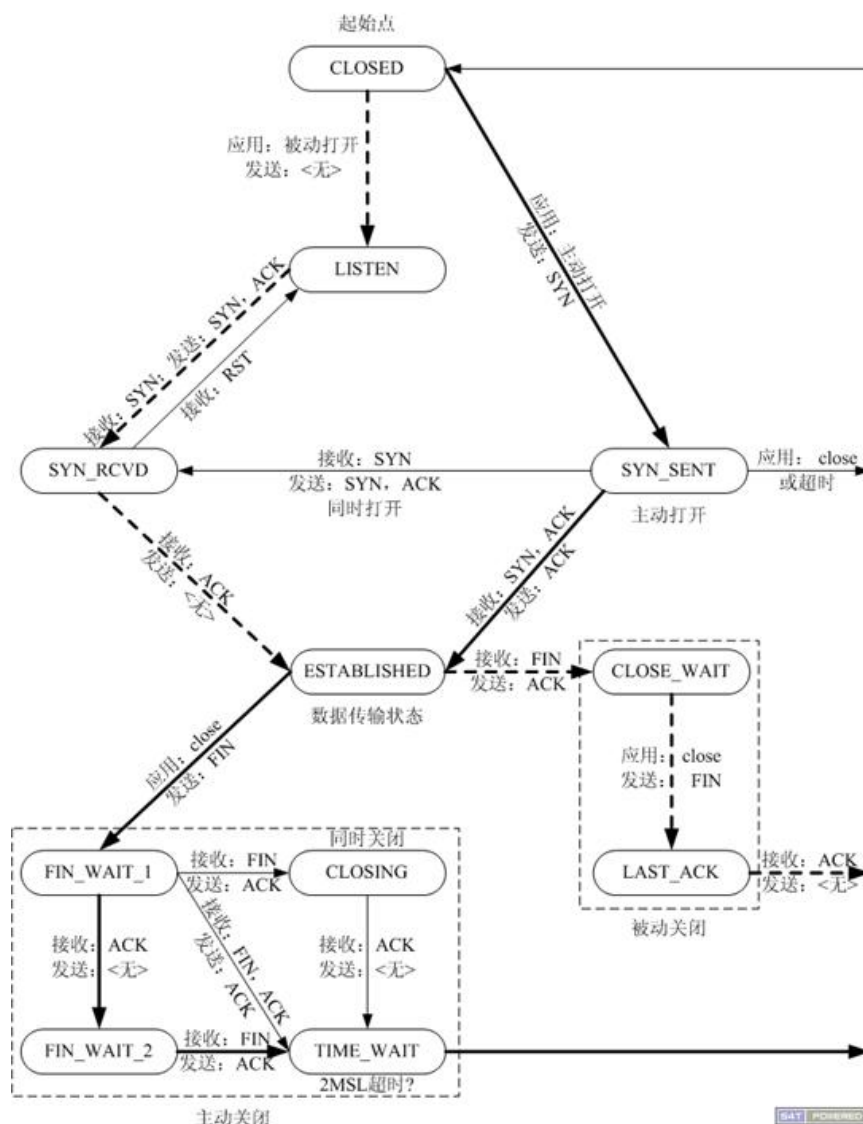
47P-TCP 状态-被动接收连接

48P-TCP 状态-被动关闭连接

49P-2MSL 时长

50P-TCP 状态-其他状态

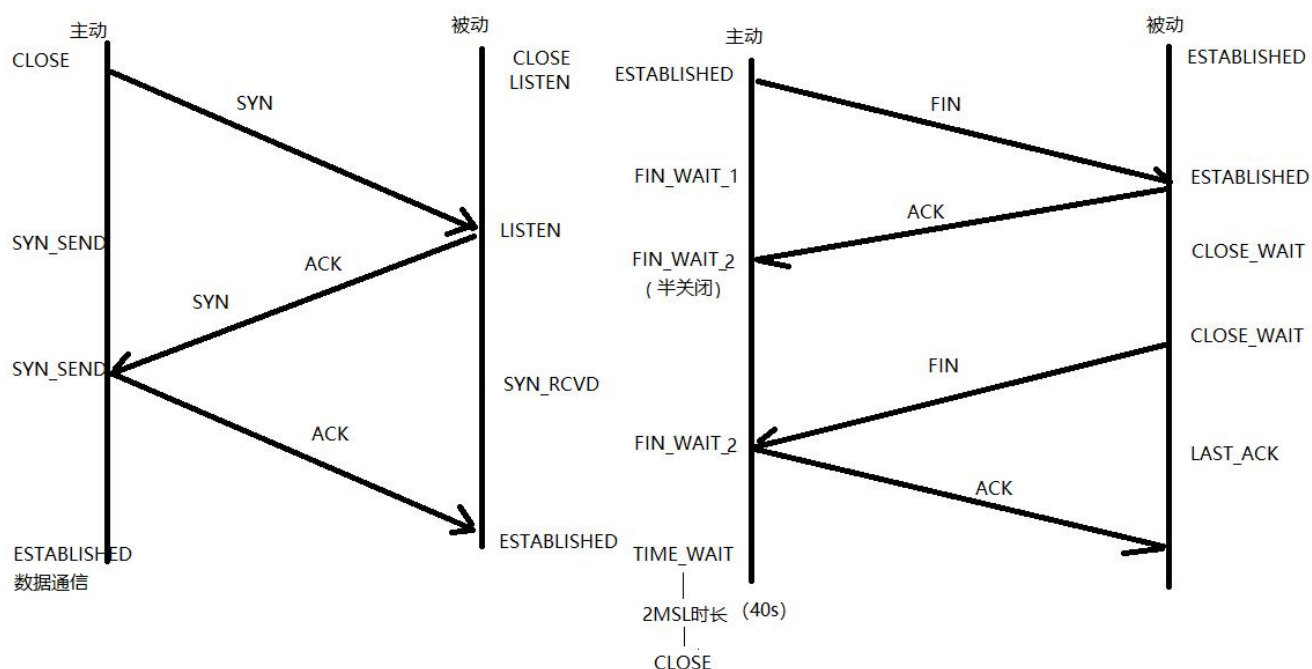
TCP 状态转换图



netstat -apn | grep client 查看客户端网络连接状态

netstat -apn | grep port

查看端口的网络连接状态



TCP 状态时序图:

结合三次握手、四次挥手 理解记忆。

1. 主动发起连接请求端: `CLOSE` -- 发送 `SYN` -- `SEND_SYN` -- 接收 `ACK`、`SYN` -- `SEND_SYN` -- 发送 `ACK` -- `ESTABLISHED` (数据通信态)

2. 主动关闭连接请求端: `ESTABLISHED` (数据通信态) -- 发送 `FIN` -- `FIN_WAIT_1` -- 接收 `ACK` -- `FIN_WAIT_2` (半关闭)

-- 接收对端发送 `FIN` -- `FIN_WAIT_2` (半关闭) -- 回发 `ACK` -- `TIME_WAIT` (只有主动关闭连接方, 会经历该状态)

-- 等 2MSL 时长 -- `CLOSE`

3. 被动接收连接请求端: `CLOSE` -- `LISTEN` -- 接收 `SYN` -- `LISTEN` -- 发送 `ACK`、`SYN` -- `SYN_RCVD` -- 接收 `ACK` -- `ESTABLISHED` (数据通信态)

4. 被动关闭连接请求端: `ESTABLISHED` (数据通信态) -- 接收 `FIN` -- `ESTABLISHED` (数据通信态) -- 发送 `ACK`

-- `CLOSE_WAIT` (说明对端【主动关闭连接端】处于半关闭状态) -- 发送 `FIN` -- `LAST_ACK` -- 接收 `ACK` -- `CLOSE`

重点记忆: `ESTABLISHED`、`FIN_WAIT_2` <--> `CLOSE_WAIT`、`TIME_WAIT` (2MSL)

```
netstat -apn | grep 端口号
```

2MSL 时长:

一定出现在【主动关闭连接请求端】。 --- 对应 TIME_WAIT 状态。

保证，最后一个 ACK 能成功被对端接收。（等待期间，对端没收到我发的 ACK，对端会再次发送 FIN 请求。）

51P-端口复用函数

52P-半关闭及 shutdown 函数

端口复用:

```
int opt = 1;          // 设置端口复用。

setsockopt(lfd, SOL_SOCKET, SO_REUSEADDR, (void *)&opt, sizeof(opt));
```

半关闭:

通信双方中，只有一端关闭通信。 --- FIN_WAIT_2

```
close (cfd) ;
```

```
shutdown(int fd, int how);
```

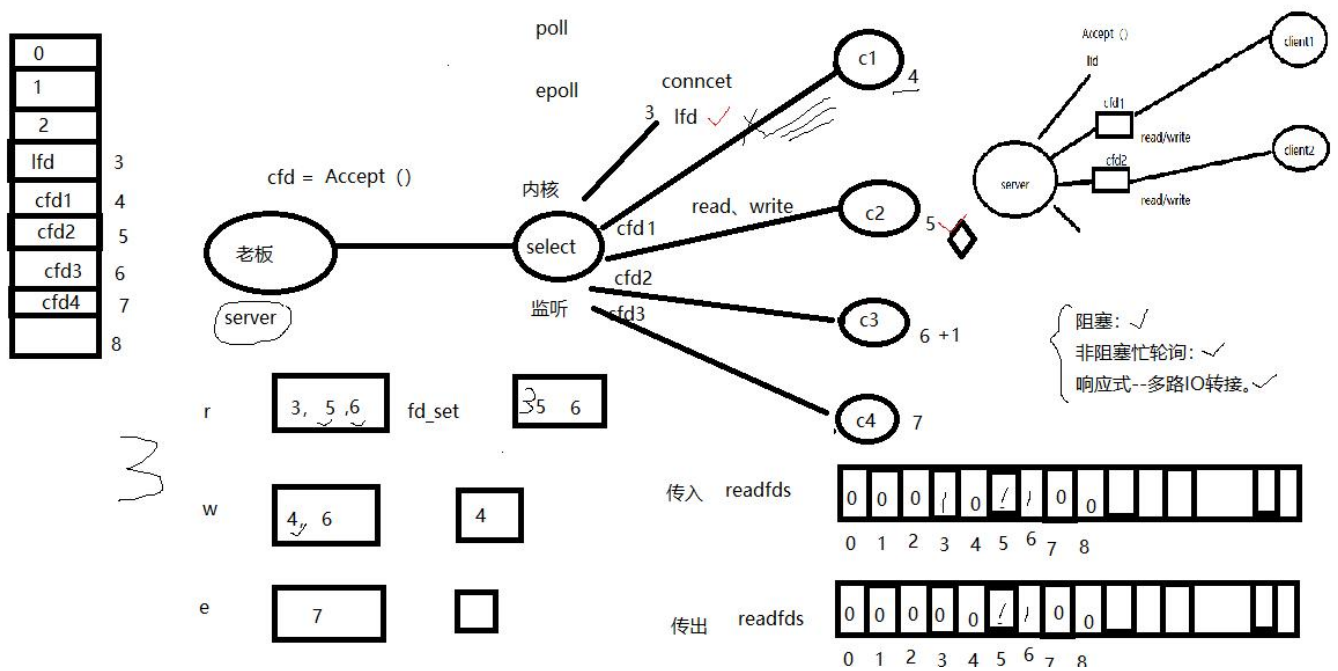
how: SHUT_RD 关读端

SHUT_WR 关写端

SHUT_RDWR 关读写

shutdown 在关闭多个文件描述符应用的文件时，采用全关闭方法。close，只关闭一个。

53P-多路 IO 转接服务器设计思路



54P-select 函数参数简介

```
int select(int nfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

nfd: 监听的所有文件描述符中，最大文件描述符+1

readfds: 读 文件描述符监听集合。 传入、传出参数

writefds: 写 文件描述符监听集合。 传入、传出参数 NULL

exceptfds: 异常 文件描述符监听集合 传入、传出参数 NULL

timeout: > 0: 设置监听超时时长。

NULL: 阻塞监听

0: 非阻塞监听，轮询

返回值:

> 0: 所有监听集合（3 个）中， 满足对应事件的总数。

0: 没有满足监听条件的文件描述符

-1: errno

55P-中午复习

56P-select 函数原型分析

```
int select(int nfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

nfd: 监听的所有文件描述符中，最大文件描述符+1

readfds: 读 文件描述符监听集合。 传入、传出参数

writefds: 写 文件描述符监听集合。 传入、传出参数 NULL

exceptfds: 异常 文件描述符监听集合 传入、传出参数 NULL

timeout: > 0: 设置监听超时时长。

NULL: 阻塞监听

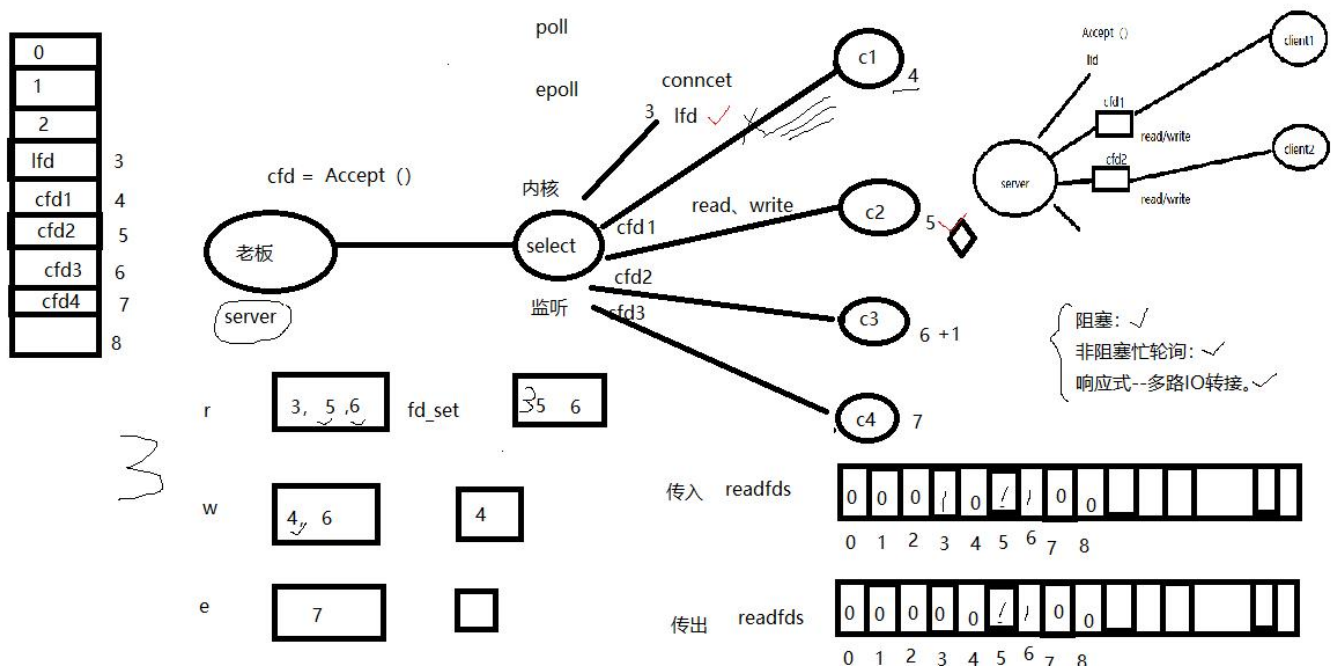
0: 非阻塞监听，轮询

返回值:

> 0: 所有监听集合（3 个）中， 满足对应事件的总数。

0: 没有满足监听条件的文件描述符

-1: errno



57P-select 相关函数参数分析

<code>void FD_CLR(int fd, fd_set *set)</code>	把某一个 fd 清除出去
<code>int FD_ISSET(int fd, fd_set *set)</code>	判定某个 fd 是否在位图中
<code>void FD_SET(int fd, fd_set *set)</code>	把某一个 fd 添加到位图
<code>void FD_ZERO(fd_set *set)</code>	位图所有二进制位置零

select 多路 IO 转接:

原理: 借助内核, select 来监听, 客户端连接、数据通信事件。

`void FD_ZERO(fd_set *set);` --- 清空一个文件描述符集合。

```
fd_set rset;
```

```
FD_ZERO(&rset);
```

`void FD_SET(int fd, fd_set *set);` --- 将待监听的文件描述符, 添加到监听集合中

```
FD_SET(3, &rset); FD_SET(5, &rset); FD_SET(6, &rset);
```

`void FD_CLR(int fd, fd_set *set);` --- 将一个文件描述符从监听集合中 移除。

```
FD_CLR (4, &rset) ;
```

`int FD_ISSET(int fd, fd_set *set);` --- 判断一个文件描述符是否在监听集合中。

返回值: 在: 1; 不在: 0;

```
FD_ISSET (4, &rset) ;
```


58P-select 实现多路 IO 转接设计思路

思路分析：

```
int maxfd = 0;

lfd = socket() ;           创建套接字

maxfd = lfd;

bind();                    绑定地址结构

listen();                  设置监听上限

fd_set rset,  allset;      创建 r 监听集合

FD_ZERO(&allset);         将 r 监听集合清空

FD_SET(lfd, &allset);      将 lfd 添加至读集合中。

while (1) {

    rset = allset;         保存监听集合

    ret = select(lfd+1,  &rset,  NULL,  NULL,  NULL);    监听文件描述符集合对应事件。

    if (ret > 0) {         有监听的描述符满足对应事件

        if (FD_ISSET(lfd, &rset)) {           // 1 在。 0 不在。

            cfd = accept ();                   建立连接，返回用于通信的文件描述符

            maxfd = cfd;

            FD_SET(cfd, &allset);              添加到监听通信描述符集合中。
        }

        for (i = lfd+1; i <= 最大文件描述符; i++) {

            FD_ISSET(i, &rset)                 有 read、write 事件

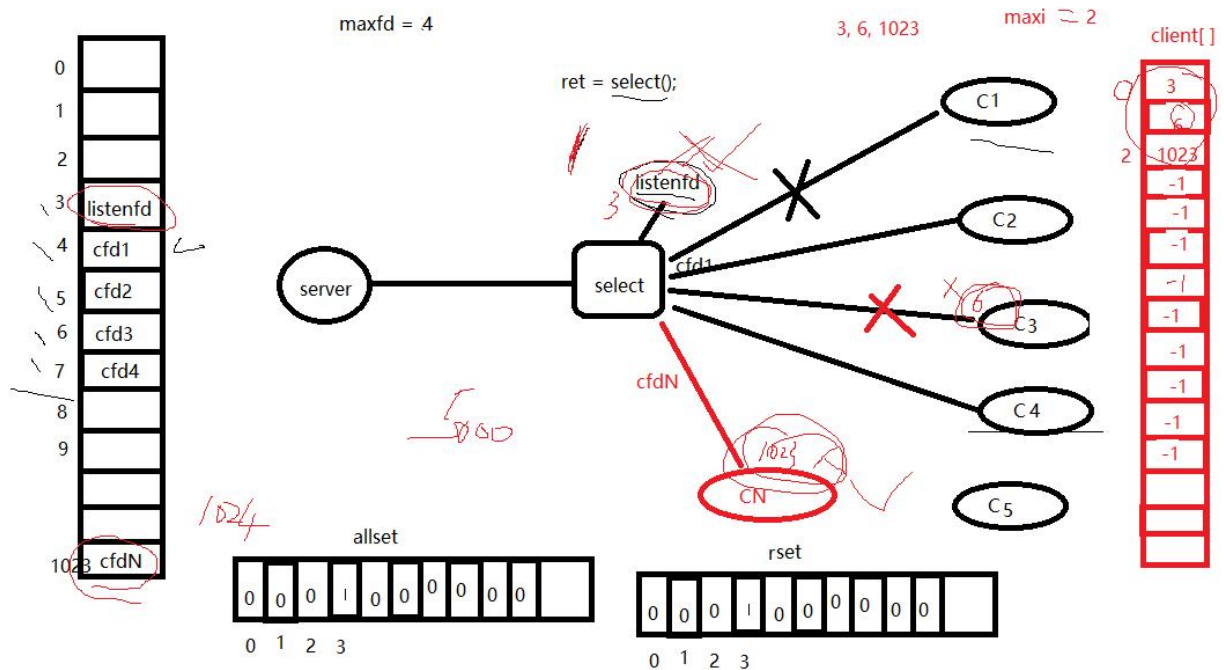
            read ()

            小 -- 大

            write();

        }
    }
}
```


59P-select 实现多路 IO 转接-代码 review



代码如下：

```

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <string.h>
5. #include <arpa/inet.h>
6. #include <ctype.h>
7.
8. #include "wrap.h"
9.
10. #define SERV_PORT 6666
11.
12. int main(int argc, char *argv[])
13. {
14.     int i, j, n, nready;
15.
16.     int maxfd = 0;
17.
18.     int listenfd, connfd;
19.
20.     char buf[BUFSIZ];          /* #define INET_ADDRSTRLEN 16 */
21.
22.     struct sockaddr_in clie_addr, serv_addr;
23.     socklen_t clie_addr_len;
24.
25.     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

```

```

26.     int opt = 1;
27.     setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
28.     bzero(&serv_addr, sizeof(serv_addr));
29.     serv_addr.sin_family= AF_INET;
30.     serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
31.     serv_addr.sin_port= htons(SERV_PORT);
32.     Bind(listenfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
33.     Listen(listenfd, 128);
34.
35.
36.     fd_set rset, allset;                                /* rset 读事件文件描述符集合 allset 用来暂存 */
37.
38.     maxfd = listenfd;
39.
40.     FD_ZERO(&allset);
41.     FD_SET(listenfd, &allset);                          /* 构造 select 监控文件描述符集 */
42.
43.     while (1) {
44.         rset = allset;                                    /* 每次循环时都从新设置 select 监控信号
集 */
45.         nready = select(maxfd+1, &rset, NULL, NULL, NULL);
46.         if (nready < 0)
47.             perr_exit("select error");
48.
49.         if (FD_ISSET(listenfd, &rset)) {                  /* 说明有新的客户端链接请求 */
50.
51.             clie_addr_len = sizeof(clie_addr);
52.             connfd = Accept(listenfd, (struct sockaddr *)&clie_addr, &clie_addr_len);    /* Acc
ept 不会阻塞 */
53.
54.             FD_SET(connfd, &allset);                      /* 向监控文件描述符集合 allset 添加新的
文件描述符 connfd */
55.
56.             if (maxfd < connfd)
57.                 maxfd = connfd;
58.
59.             if (0 == --nready)                            /* 只有 listenfd 有事件, 后续的 for 不
需执行 */
60.                 continue;
61.         }
62.
63.         for (i = listenfd+1; i <= maxfd; i++) {          /* 检测哪个 clients 有数据就绪 */
64.
65.             if (FD_ISSET(i, &rset)) {
66.
67.                 if ((n = Read(i, buf, sizeof(buf))) == 0) { /* 当 client 关闭链接时, 服务器端也关闭对
应链接 */
68.                     Close(i);

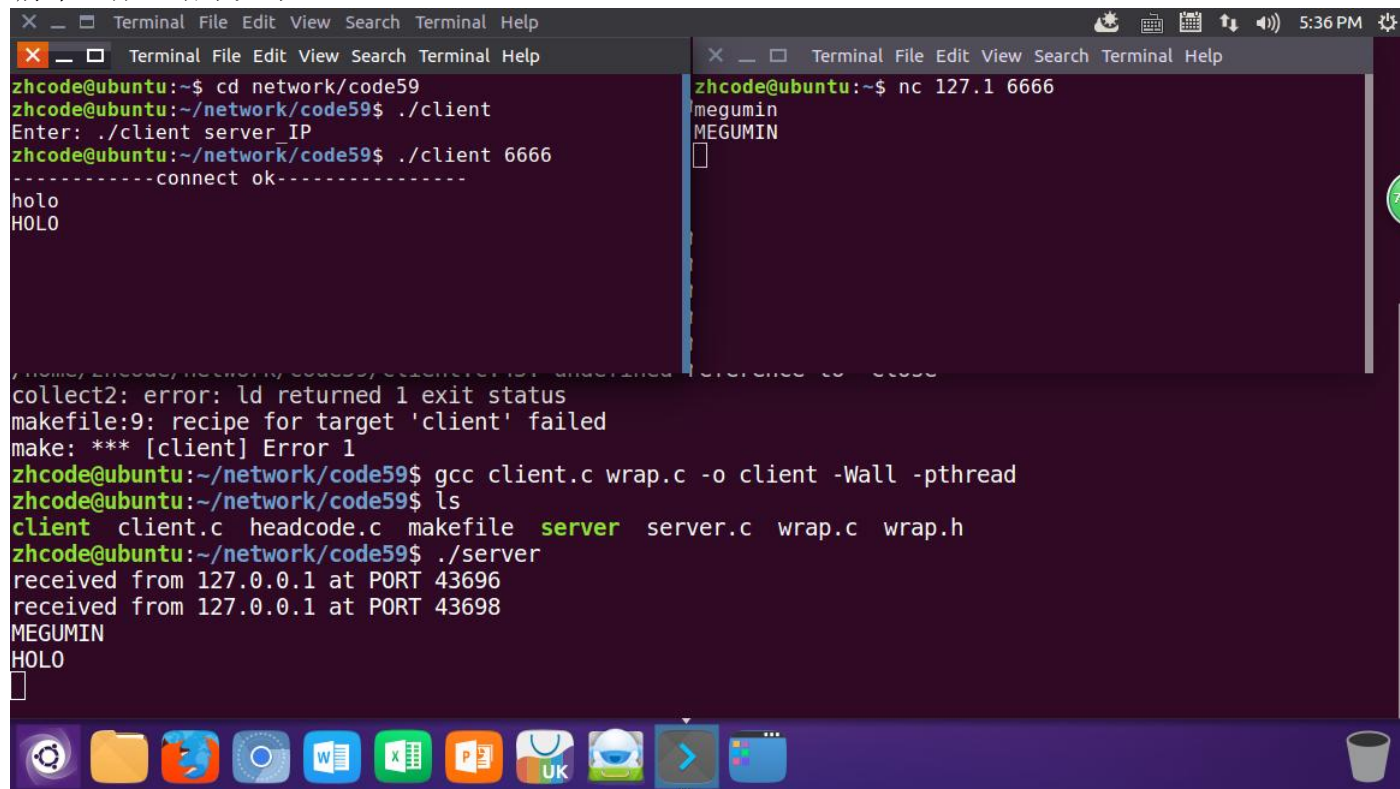
```

```

69.             FD_CLR(i, &allset);                               /* 解除 select 对此文件描述符的监
   控 */
70.
71.             } else if (n > 0) {
72.
73.                 for (j = 0; j < n; j++)
74.                     buf[j] = toupper(buf[j]);
75.                 Write(i, buf, n);
76.             }
77.         }
78.     }
79. }
80.
81. Close(listenfd);
82.
83. return 0;
84. }

```

编译运行，结果如下：



```

zhcode@ubuntu:~$ cd network/code59
zhcode@ubuntu:~/network/code59$ ./client
Enter: ./client server_IP
zhcode@ubuntu:~/network/code59$ ./client 6666
-----connect ok-----
hoLO
HOLO

collect2: error: ld returned 1 exit status
makefile:9: recipe for target 'client' failed
make: *** [client] Error 1
zhcode@ubuntu:~/network/code59$ gcc client.c wrap.c -o client -Wall -pthread
zhcode@ubuntu:~/network/code59$ ls
client client.c headcode.c makefile server server.c wrap.c wrap.h
zhcode@ubuntu:~/network/code59$ ./server
received from 127.0.0.1 at PORT 43696
received from 127.0.0.1 at PORT 43698
MEGUMIN
HOLO

```

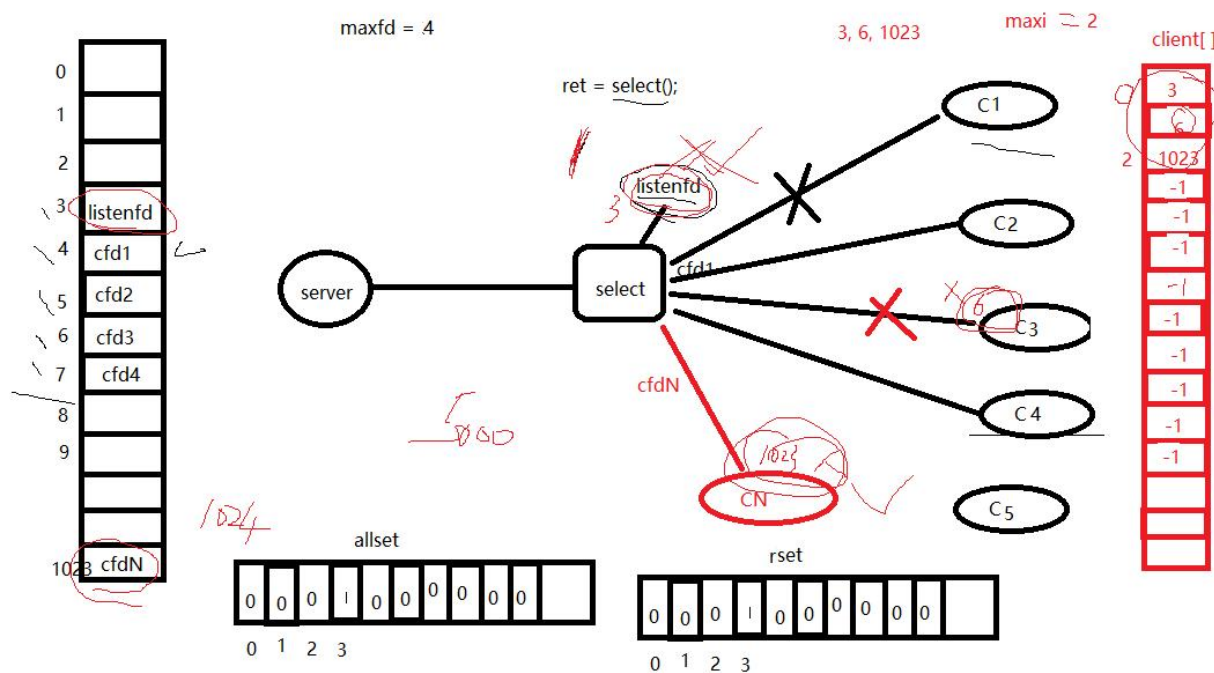
如图，借助 select 也可以实现多线程

60P-select 实现多路 IO 转接-代码实现

61P-select 实现多路 IO 转接-添加注释

代码太长了，直接看 59 话吧

62P-select 优缺点



select 优缺点:

缺点： 监听上限受文件描述符限制。 最大 1024.

检测满足条件的 fd， 自己添加业务逻辑提高小。 提高了编码难度。

优点： 跨平台。win、linux、macOS、Unix、类 Unix、mips

select 代码里有个可以优化的地方，用数组存下文件描述符，这样就不需要每次扫描一大堆无关文件描述符了


```

24.     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
25.
26.     int opt = 1;
27.     setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
28.
29.     bzero(&serv_addr, sizeof(serv_addr));
30.     serv_addr.sin_family= AF_INET;
31.     serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
32.     serv_addr.sin_port= htons(SERV_PORT);
33.
34.     Bind(listenfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
35.     Listen(listenfd, 128);
36.
37.     maxfd = listenfd;                                /* 起初 listenfd 即为最大文件描述符 */
38.
39.     maxi = -1;                                        /* 将来用作 client[]的下标, 初始值指向 0 个元素之前下标位置 */
40.     for (i = 0; i < FD_SETSIZE; i++)
41.         client[i] = -1;                                /* 用-1 初始化 client[] */
42.
43.     FD_ZERO(&allset);
44.     FD_SET(listenfd, &allset);                        /* 构造 select 监控文件描述符集 */
45.
46.     while (1) {
47.         rset = allset;                                /* 每次循环时都重新设置 select 监控信号集 */
48.
49.         nready = select(maxfd+1, &rset, NULL, NULL, NULL); //2 1--lfd 1--connfd
50.         if (nready < 0)
51.             perr_exit("select error");
52.
53.         if (FD_ISSET(listenfd, &rset)) {                /* 说明有新的客户端链接请求 */
54.
55.             clie_addr_len = sizeof(clie_addr);
56.             connfd = Accept(listenfd, (struct sockaddr *)&clie_addr, &clie_addr_len); /* Accept 不会阻塞 */
57.             printf("received from %s at PORT %d\n",
58.                 inet_ntop(AF_INET, &clie_addr.sin_addr, str, sizeof(str)),
59.                 ntohs(clie_addr.sin_port));
60.
61.             for (i = 0; i < FD_SETSIZE; i++)
62.                 if (client[i] < 0) {                    /* 找 client[]中没有使用的位置 */
63.                     client[i] = connfd;                /* 保存 accept 返回的文件描述符到 client[]里 */
64.                     break;
65.                 }
66.

```

```

67.         if (i == FD_SETSIZE) {                                /* 达到 select 能监控的文件个数上
限 1024 */
68.             fputs("too many clients\n", stderr);
69.             exit(1);
70.         }
71.
72.         FD_SET(connfd, &allset);                                /* 向监控文件描述符集合 allset 添加新的
文件描述符 connfd */
73.
74.         if (connfd > maxfd)
75.             maxfd = connfd;                                    /* select 第一个参数需要 */
76.
77.         if (i > maxi)
78.             maxi = i;                                          /* 保证 maxi 存的总是 client[] 最后一个
元素下标 */
79.
80.         if (--nready == 0)
81.             continue;
82.     }
83.
84.     for (i = 0; i <= maxi; i++) {                                /* 检测哪个 clients 有数据就
绪 */
85.
86.         if ((sockfd = client[i]) < 0)
87.             continue;
88.         if (FD_ISSET(sockfd, &rset)) {
89.
90.             if ((n = Read(sockfd, buf, sizeof(buf))) == 0) {    /* 当 client 关闭链接时,服务器端也
关闭对应链接 */
91.                 Close(sockfd);
92.                 FD_CLR(sockfd, &allset);                        /* 解除 select 对此文件描述符的监
控 */
93.                 client[i] = -1;
94.             } else if (n > 0) {
95.                 for (j = 0; j < n; j++)
96.                     buf[j] = toupper(buf[j]);
97.                 Write(sockfd, buf, n);
98.                 Write(STDOUT_FILENO, buf, n);
99.             }
100.            if (--nready == 0)
101.                break;                                          /* 跳出 for, 但还在 while 中 */
102.        }
103.    }
104. }
105. Close(listenfd);
106. return 0;
107.}

```

编译运行和改进前没啥区别，这里就不贴图了

64P-总结

TCP 状态时序图：

结合三次握手、四次挥手 理解记忆。

1. 主动发起连接请求端： CLOSE -- 发送 SYN -- SEND_SYN -- 接收 ACK、SYN -- SEND_SYN -- 发送 ACK -- ESTABLISHED（数据通信态）

2. 主动关闭连接请求端： ESTABLISHED（数据通信态） -- 发送 FIN -- FIN_WAIT_1 -- 接收 ACK -- FIN_WAIT_2（半关闭）

-- 接收对端发送 FIN -- FIN_WAIT_2（半关闭） -- 回发 ACK -- TIME_WAIT（只有主动关闭连接方，会经历该状态）

-- 等 2MSL 时长 -- CLOSE

3. 被动接收连接请求端： CLOSE -- LISTEN -- 接收 SYN -- LISTEN -- 发送 ACK、SYN -- SYN_RCVD -- 接收 ACK -- ESTABLISHED（数据通信态）

4. 被动关闭连接请求端： ESTABLISHED（数据通信态） -- 接收 FIN -- ESTABLISHED（数据通信态） -- 发送 ACK

-- CLOSE_WAIT（说明对端【主动关闭连接端】处于半关闭状态） -- 发送 FIN -- LAST_ACK -- 接收 ACK -- CLOSE

重点记忆： ESTABLISHED、FIN_WAIT_2 <--> CLOSE_WAIT、TIME_WAIT（2MSL）

netstat -apn | grep 端口号

2MSL 时长：

一定出现在【主动关闭连接请求端】。 --- 对应 TIME_WAIT 状态。

保证，最后一个 ACK 能成功被对端接收。（等待期间，对端没收到我发的 ACK，对端会再次发送 FIN 请求。）

端口复用：

```
int opt = 1;          // 设置端口复用。
```

```
setsockopt(lfd, SOL_SOCKET, SO_REUSEADDR, (void *)&opt, sizeof(opt));
```

半关闭：

通信双方中，只有一端关闭通信。 --- FIN_WAIT_2

```
close (cfd) ;
```

```
shutdown(int fd, int how);
```

how: SHUT_RD 关读端

SHUT_WR 关写端

SHUT_RDWR 关读写

shutdown 在关闭多个文件描述符应用的文件时，采用全关闭方法。close，只关闭一个。

select 多路 IO 转接：

原理： 借助内核， select 来监听， 客户端连接、数据通信事件。

void FD_ZERO(fd_set *set); --- 清空一个文件描述符集合。

```
fd_set rset;
```

```
FD_ZERO(&rset);
```

void FD_SET(int fd, fd_set *set); --- 将待监听的文件描述符，添加到监听集合中

```
FD_SET(3, &rset); FD_SET(5, &rset); FD_SET(6, &rset);
```

void FD_CLR(int fd, fd_set *set); --- 将一个文件描述符从监听集合中 移除。

```
FD_CLR (4, &rset) ;
```

int FD_ISSET(int fd, fd_set *set); --- 判断一个文件描述符是否在监听集合中。

返回值： 在： 1； 不在： 0；

```
FD_ISSET (4, &rset) ;
```

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

nfds: 监听的所有文件描述符中，最大文件描述符+1

readfds: 读 文件描述符监听集合。 传入、传出参数

writefds: 写 文件描述符监听集合。 传入、传出参数 NULL

exceptfds: 异常 文件描述符监听集合 传入、传出参数 NULL

timeout: > 0: 设置监听超时时长。

NULL: 阻塞监听

0: 非阻塞监听, 轮询

返回值:

> 0: 所有监听集合 (3 个) 中, 满足对应事件的总数。

0: 没有满足监听条件的文件描述符

-1: errno

思路分析:

```
int maxfd = 0;
```

```
lfd = socket() ;          创建套接字
```

```
maxfd = lfd;
```

```
bind();                  绑定地址结构
```

```
listen();               设置监听上限
```

```
fd_set rset, allset;    创建 r 监听集合
```

```
FD_ZERO(&allset);       将 r 监听集合清空
```

```
FD_SET(lfd, &allset);   将 lfd 添加至读集合中。
```

```
while (1) {
```

```
    rset = allset;       保存监听集合
```

```
    ret = select(lfd+1, &rset, NULL, NULL, NULL);    监听文件描述符集合对应事件。
```

```
    if (ret > 0) {       有监听的描述符满足对应事件
```

```
        if (FD_ISSET(lfd, &rset)) {                // 1 在。 0 不在。
```

```
            cfd = accept ();                        建立连接, 返回用于通信的文件描述符
```

```
            maxfd = cfd;
```

```
            FD_SET(cfd, &allset);                    添加到监听通信描述符集合中。
```

```

    }

    for (i = lfd+1; i <= 最大文件描述符; i++) {

        FD_ISSET(i, &rset)           有 read、write 事件

        read ()

        小 -- 大

        write();
    }
}

```

select 优缺点:

缺点: 监听上限受文件描述符限制。 最大 1024.

检测满足条件的 fd, 自己添加业务逻辑提高小。 提高了编码难度。

优点: 跨平台。win、linux、macOS、Unix、类 Unix、mips

65P-复习

66P-poll 函数原型分析

poll 是对 select 的改进，但是它是个半成品，相对 select 提升不大。最终版本是 epoll，所以 poll 了解一下就完事儿，重点掌握 epoll。

poll:

```
int poll(struct pollfd *fds, nfd_t nfd, int timeout);
```

fds: 监听的文件描述符【数组】

```
struct pollfd {
```

```
    int fd:    待监听的文件描述符
```

```
    short events:    待监听的文件描述符对应的监听事件
```

取值: POLLIN、POLLOUT、POLLERR

```
    short revnets:    传入时，给 0。如果满足对应事件的话，返回 非 0 --> POLLIN、POLLOUT、POLLERR
}
```

nfd: 监听数组的，实际有效监听个数。

timeout: > 0: 超时时长。单位: 毫秒。

-1: 阻塞等待

0: 不阻塞

返回值: 返回满足对应监听事件的文件描述符 总个数。

优点:

自带数组结构。 可以将 监听事件集合 和 返回事件集合 分离。

拓展 监听上限。 超出 1024 限制。

缺点:

不能跨平台。 Linux

无法直接定位满足监听事件的文件描述符， 编码难度较大。

67P-poll 函数使用注意事项示例

```
struct pollfd pfds[1024];
```

```
pfds[0].fd = lfd;
pfds[0].events = POLLIN;
pfds[0].revents = 0;
```

maxi = 1 ; →

```
pfds[1].fd = cfd1;
pfds[1].events = POLLIN;
pfds[1].revents = 0;
```

```
pfds[2].fd = cfd2;
pfds[2].events = POLLIN;
pfds[2].revents = 0;
```

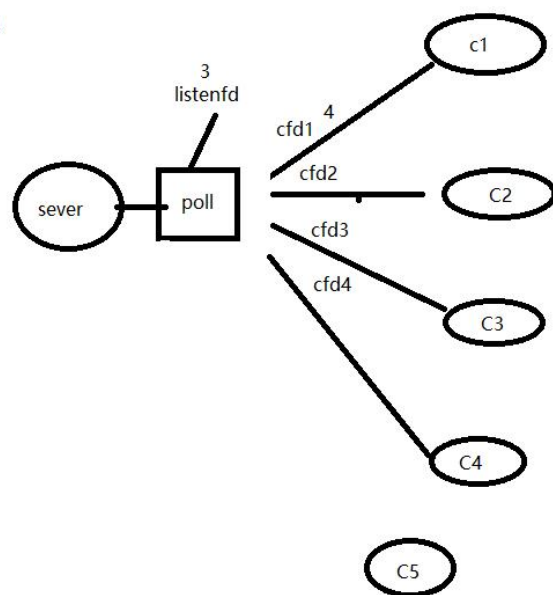
.....

```
while (1) {
    int ret = poll(pfds, 5, -1);
    for (i = 0 ; i < 5; i++)
        if (pfds[i].revents & POLLIN)
            Accept();
            Read/write()
}
```

struct poll类型
client

0	fd=listenfd POLLIN
1	fd=4 POLLIN
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1
8	-1
9	-1
	-1
	-1
	-1
	-1
1023	-1

OPEN_MAX 1024



68P-poll 函数实现服务器

这个东西用得少，基本都用 epoll，从讲义上挂个代码过来，看看视频里思路就完事儿

```
1.  /* server.c */
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <netinet/in.h>
6.  #include <arpa/inet.h>
7.  #include <poll.h>
8.  #include <errno.h>
9.  #include "wrap.h"
10.
11. #define MAXLINE 80
12. #define SERV_PORT 6666
13. #define OPEN_MAX 1024
14.
15. int main(int argc, char *argv[])
16. {
17.     int i, j, maxi, listenfd, connfd, sockfd;
18.     int nready;
19.     ssize_t n;
20.     char buf[MAXLINE], str[INET_ADDRSTRLEN];
21.     socklen_t clilen;
22.     struct pollfd client[OPEN_MAX];
23.     struct sockaddr_in cliaddr, servaddr;
24.
25.     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
26.
27.     bzero(&servaddr, sizeof(servaddr));
28.     servaddr.sin_family = AF_INET;
29.     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
30.     servaddr.sin_port = htons(SERV_PORT);
31.
32.     Bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
33.
34.     Listen(listenfd, 20);
35.
36.     client[0].fd = listenfd;
37.     client[0].events = POLLRDNORM;          /* listenfd 监听普通读事件 */
38.
39.     for (i = 1; i < OPEN_MAX; i++)
40.         client[i].fd = -1;                /* 用-1 初始化 client[]里剩下元素 */
41.     maxi = 0;                             /* client[]数组有效元素中最大元素下标 */
42.
43.     for ( ; ; ) {
44.         nready = poll(client, maxi+1, -1); /* 阻塞 */
45.         if (client[0].revents & POLLRDNORM) { /* 有客户端链接请求 */
```

```

46.         clilen = sizeof(cliaddr);
47.         connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &clilen);
48.         printf("received from %s at PORT %d\n",
49.             inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
50.             ntohs(cliaddr.sin_port));
51.         for (i = 1; i < OPEN_MAX; i++) {
52.             if (client[i].fd < 0) {
53.                 client[i].fd = connfd; /* 找到 client[] 中空闲的位置, 存放 accept 返回的 connfd */
54.                 break;
55.             }
56.         }
57.
58.         if (i == OPEN_MAX)
59.             perr_exit("too many clients");
60.
61.         client[i].events = POLLRDNORM; /* 设置刚刚返回的 connfd, 监控读事件 */
62.         if (i > maxi)
63.             maxi = i; /* 更新 client[] 中最大元素下标 */
64.         if (--nready <= 0)
65.             continue; /* 没有更多就绪事件时, 继续回到 poll 阻塞 */
66.     }
67.     for (i = 1; i <= maxi; i++) { /* 检测 client[] */
68.         if ((sockfd = client[i].fd) < 0)
69.             continue;
70.         if (client[i].revents & (POLLRDNORM | POLLERR)) {
71.             if ((n = Read(sockfd, buf, MAXLINE)) < 0) {
72.                 if (errno == ECONNRESET) { /* 当收到 RST 标志时 */
73.                     /* connection reset by client */
74.                     printf("client[%d] aborted connection\n", i);
75.                     Close(sockfd);
76.                     client[i].fd = -1;
77.                 } else {
78.                     perr_exit("read error");
79.                 }
80.             } else if (n == 0) {
81.                 /* connection closed by client */
82.                 printf("client[%d] closed connection\n", i);
83.                 Close(sockfd);
84.                 client[i].fd = -1;
85.             } else {
86.                 for (j = 0; j < n; j++)
87.                     buf[j] = toupper(buf[j]);
88.                 Writen(sockfd, buf, n);
89.             }
90.             if (--nready <= 0)
91.                 break; /* no more readable descriptors */
92.         }
93.     }
94. }

```

```
95.     return 0;
```

```
96. }
```

69P-poll 总结

优点：

自带数组结构。 可以将 监听事件集合 和 返回事件集合 分离。

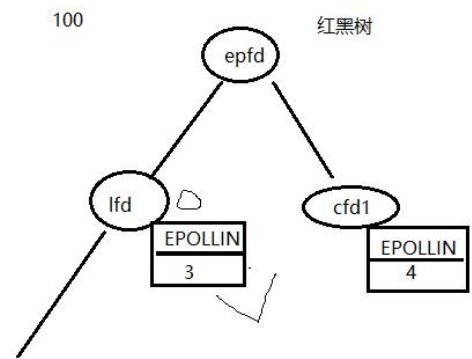
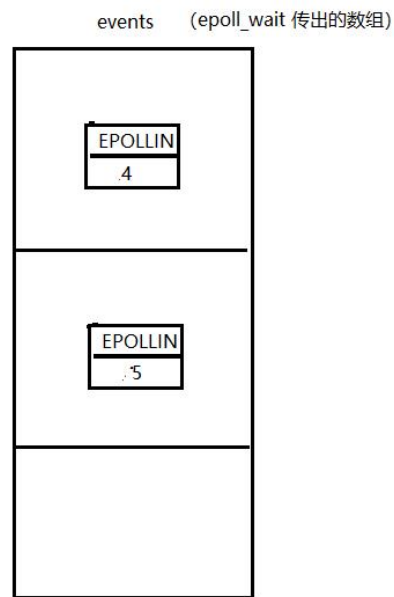
拓展 监听上限。 超出 1024 限制。

缺点：

不能跨平台。 Linux

无法直接定位满足监听事件的文件描述符， 编码难度较大。

70P-epoll 函数实现的多路 IO 转接



代码如下：

```
1. #include <stdio.h>
2. #include <unistd.h>
3. #include <stdlib.h>
4. #include <string.h>
5. #include <arpa/inet.h>
6. #include <sys/epoll.h>
7. #include <errno.h>
8. #include <ctype.h>
9.
10. #include "wrap.h"
11.
12. #define MAXLINE 8192
13. #define SERV_PORT 8000
14.
15. #define OPEN_MAX 5000
16.
17. int main(int argc, char *argv[])
18. {
19.     int i, listenfd, connfd, sockfd;
20.     int n, num = 0;
21.     ssize_t nready, efd, res;
22.     char buf[MAXLINE], str[INET_ADDRSTRLEN];
23.     socklen_t clilen;
24.
25.     struct sockaddr_in cliaddr, servaddr;
26.     struct epoll_event tep, ep[OPEN_MAX]; //tep: epoll_ctl 参数 ep[] : epoll_wait 参数
27.
```

```

28.  listenfd = Socket(AF_INET, SOCK_STREAM, 0);
29.  int opt = 1;
30.  setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)); //端口复用
31.  bzero(&servaddr, sizeof(servaddr));
32.  servaddr.sin_family = AF_INET;
33.  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
34.  servaddr.sin_port = htons(SERV_PORT);
35.  Bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
36.  Listen(listenfd, 20);
37.
38.  efd = epoll_create(OPEN_MAX); //创建 epoll 模型, efd 指向红黑树根节点
39.  if (efd == -1)
40.      perr_exit("epoll_create error");
41.
42.  tep.events = EPOLLIN;
43.  tep.data.fd = listenfd; //指定 lfd 的监听时间为"读"
44.
45.  res = epoll_ctl(efd, EPOLL_CTL_ADD, listenfd, &tep); //将 lfd 及对应的结构体设置到树上,efd 可找到
    该树
46.  if (res == -1)
47.      perr_exit("epoll_ctl error");
48.
49.  for ( ; ; ) {
50.      /*epoll 为 server 阻塞监听事件, ep 为 struct epoll_event 类型数组, OPEN_MAX 为数组容量, -1 表永久阻
    塞*/
51.      nready = epoll_wait(efd, ep, OPEN_MAX, -1);
52.      if (nready == -1)
53.          perr_exit("epoll_wait error");
54.
55.      for (i = 0; i < nready; i++) {
56.          if (!(ep[i].events & EPOLLIN)) //如果不是"读"事件, 继续循环
57.              continue;
58.
59.          if (ep[i].data.fd == listenfd) { //判断满足事件的 fd 是不是 lfd
60.              cliilen = sizeof(cliaddr);
61.              connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &cliilen); //接受链接
62.
63.              printf("received from %s at PORT %d\n",
64.                  inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
65.                  ntohs(cliaddr.sin_port));
66.              printf("cfd %d---client %d\n", connfd, ++num);
67.
68.              tep.events = EPOLLIN; tep.data.fd = connfd;
69.              res = epoll_ctl(efd, EPOLL_CTL_ADD, connfd, &tep); //加入红黑树
70.              if (res == -1)
71.                  perr_exit("epoll_ctl error");
72.
73.          } else { //不是 lfd,
74.              sockfd = ep[i].data.fd;

```

```

75.         n = Read(sockfd, buf, MAXLINE);
76.
77.         if (n == 0) {                                     //读到 0,说明客户端关闭链接
78.             res = epoll_ctl(efd, EPOLL_CTL_DEL, sockfd, NULL); //将该文件描述符从红黑树摘
除
79.             if (res == -1)
80.                 perr_exit("epoll_ctl error");
81.             Close(sockfd);                                //关闭与该客户端的连接
82.             printf("client[%d] closed connection\n", sockfd);
83.
84.         } else if (n < 0) {                                //出错
85.             perror("read n < 0 error: ");
86.             res = epoll_ctl(efd, EPOLL_CTL_DEL, sockfd, NULL); //摘除节点
87.             Close(sockfd);
88.
89.         } else {                                           //实际读到了字节数
90.             for (i = 0; i < n; i++)
91.                 buf[i] = toupper(buf[i]);                 //转大写,写回给客户端
92.
93.             Write(STDOUT_FILENO, buf, n);
94.             Writen(sockfd, buf, n);
95.         }
96.     }
97. }
98. }
99. Close(listenfd);
100. Close(efd);
101.
102. return 0;
103.}

```


71P-突破 1024 文件描述符设置

突破 1024 文件描述符限制:

`cat /proc/sys/fs/file-max` --> 当前计算机所能打开的最大文件个数。 受硬件影响。

`ulimit -a` ——> 当前用户下的进程，默认打开文件描述符个数。 缺省为 1024

修改:

打开 `sudo vi /etc/security/limits.conf`， 写入:

* soft nofile 65536 --> 设置默认值， 可以直接借助命令修改。 【注销用户，使其生效】

* hard nofile 100000 --> 命令修改上限。

`cat /proc/sys/fs/file-max` 查看最大文件描述符上限

```
zhcode@ubuntu:~$ cat /proc/sys/fs/file-max
196571
zhcode@ubuntu:~$
```

`ulimit -a`

```
zhcode@ubuntu:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size               (blocks, -f) unlimited
pending signals         (-i) 7722
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files              (-n) 1024
pipe size               (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority      (-r) 0
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) 7722
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
zhcode@ubuntu:~$
```

sudo vi /etc/security/limits.conf 修改上限

```
46 #*                soft    core      0
47 #root             hard    core      100000
48 #*                hard    rss       10000
49 #@student         hard    nproc     20
50 #@faculty         soft    nproc     20
51 #@faculty         hard    nproc     50
52 #ftp              hard    nproc     0
53 #ftp              -       chroot    /ftp
54 #@student         -       maxlogins 4
55 *                 soft    nofile    3000
56 *                 hard    nofile    20000
57 # End of file
```

修改之后，注销用户重新登录，查看文件描述符上限：

```
zhcode@ubuntu:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size               (blocks, -f) unlimited
pending signals         (-i) 7722
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files              (-n) 3000
pipe size                (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority      (-r) 0
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) 7722
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
zhcode@ubuntu:~$
```

如图，已经修改成功了。

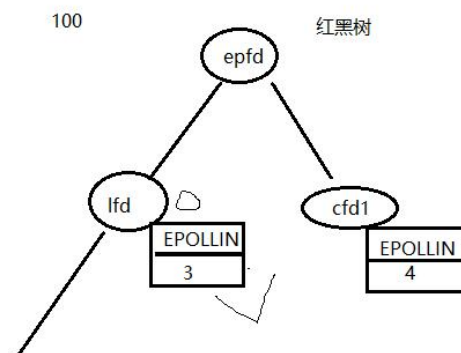
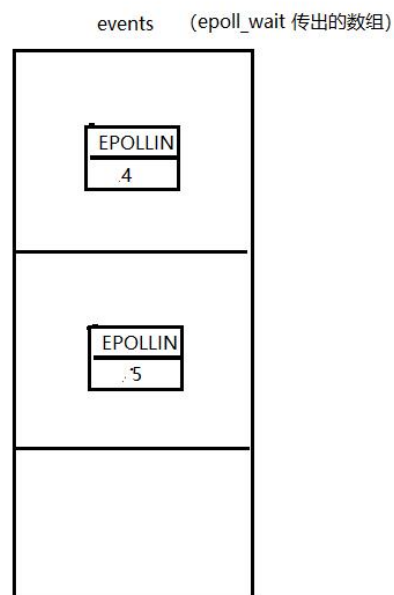
如果使用 `ulimit -n` 来修改，会受到之前设置的 `hard` 的限制：

```
zhcode@ubuntu:~$ ulimit -n 21000
bash: ulimit: open files: cannot modify limit: Operation not permitted
zhcode@ubuntu:~$
```

```
zhcode@ubuntu:~$ ulimit -n 16000
zhcode@ubuntu:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size               (blocks, -f) unlimited
pending signals         (-i) 7722
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files              (-n) 16000
pipe size               (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority      (-r) 0
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) 7722
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
zhcode@ubuntu:~$
```

用 `ulimit -n` 设置之后，往下调可以，往上调需要注销用户再登录。

72P-epoll_create と epoll_ctl



epoll:

```
int epoll_create(int size);
```

创建一棵监听红黑树

size: 创建的红黑树的监听节点数量。(仅供内核参考。)

返回值：指向新创建的红黑树的根节点的 fd。

失败: -1 errno

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event); 操作监听红黑树

epfd: epoll_create 函数的返回值。 epfd

op: 对该监听红黑数所做的操作。

EPOLL_CTL_ADD 添加 fd 到 监听红黑树

EPOLL_CTL_MOD 修改 fd 在 监听红黑树上的监听事件。

EPOLL_CTL_DEL 将一个 fd 从监听红黑树上摘下（取消监听）

fd:

待监听的 fd

event: 本质 struct epoll_event 结构体 地址

成员 events:

EPOLLIN / EPOLLOUT / EPOLLERR

成员 data: 联合体（共用体）:

int fd; 对应监听事件的 fd

void *ptr;

uint32_t u32;

uint64_t u64;

返回值: 成功 0; 失败: -1 errno

73P-epoll_wait 函数

`int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);` 阻塞监听。

`epfd`: `epoll_create` 函数的返回值。 `epfd`

`events`: 传出参数，【数组】， 满足监听条件的 那些 `fd` 结构体。

`maxevents`: 数组 元素的总个数。 1024

`struct epoll_event evnets[1024]`
`timeout`:

-1: 阻塞

0: 不阻塞

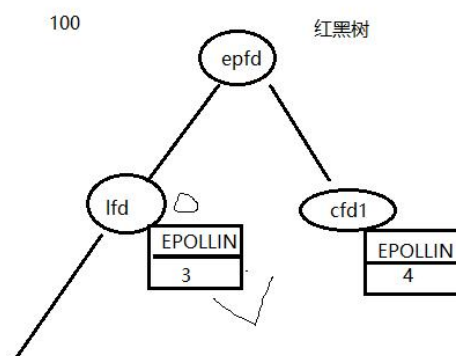
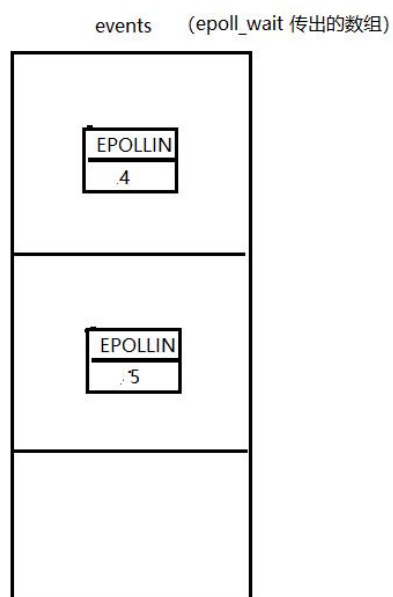
>0: 超时时间 （毫秒）

返回值:

> 0: 满足监听的 总个数。 可以用作循环上限。

0: 没有 `fd` 满足监听事件

-1: 失败。 `errno`



74P-中午复习

epoll 实现多路 IO 转接思路:

```
lfd = socket ();          监听连接事件 lfd
bind();
listen();
```

```
int epfd = epoll_create(1024);          epfd, 监听红黑树的树根。
```

```
struct epoll_event tep, ep[1024];      tep, 用来设置单个 fd 属性, ep 是 epoll_wait()
传出的满足监听事件的数组。
```

```
tep.events = EPOLLIN;                  初始化 lfd 的监听属性。
tep.data.fd = lfd
```

```
epoll_ctl(epfd, EPOLL_CTL_ADD, lfd, &tep);      将 lfd 添加到监听红黑树上。
```

```
while (1) {
```

```
    ret = epoll_wait(epfd, ep, 1024, -1);        实施监听
```

```
    for (i = 0; i < ret; i++) {
```

```
        if (ep[i].data.fd == lfd) {                // lfd 满足读事件, 有新的客户端发起连接请
求
```

```
            cfd = Accept();
```

```
            tep.events = EPOLLIN;                    初始化 cfd 的监听属性。
            tep.data.fd = cfd;
```

```
            epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &tep);
```

```
        } else {                                    cfd 们 满足读事件, 有客户端写数据来。
```

```
            n = read(ep[i].data.fd, buf, sizeof(buf));
```

```
            if ( n == 0) {
```

```
                close(ep[i].data.fd);
```

```
                epoll_ctl(epfd, EPOLL_CTL_DEL, ep[i].data.fd , NULL); // 将关闭的 cfd,
从监听树上摘下。
```

```
            } else if (n > 0) {
```

小一大

```
write(ep[i].data.fd, buf, n);
```

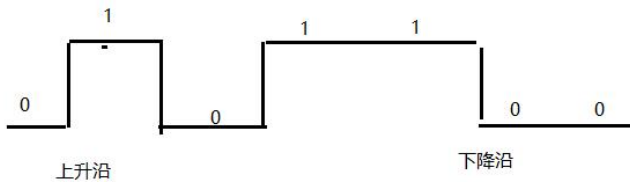
```
}
```

```
}
```

```
}
```


75P-ET 和 LT 模式

epoll 是 Linux 下多路复用 IO 接口 select/poll 的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统 CPU 利用率，因为它会复用文件描述符集合来传递结果而不用迫使开发者每次等待事件之前都必须重新准备要被侦听的文件描述符集合，另一点原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核 IO 事件异步唤醒而加入 Ready 队列的描述符集合就行了。



EPOLL 事件有两种模型：

Edge Triggered (ET) 边缘触发只有数据到来才触发，不管缓存区中是否还有数据。

Level Triggered (LT) 水平触发只要有数据都会触发。

视频中 epoll 测试代码如下，用一个子进程来写内容，用 ET 和 LT 模式来读取，结果很能说明问题：

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <sys/epoll.h>
4. #include <errno.h>
5. #include <unistd.h>
6.
7. #define MAXLINE 10
8.
9. int main(int argc, char *argv[])
10. {
11.     int efd, i;
12.     int pfd[2];
13.     pid_t pid;
14.     char buf[MAXLINE], ch = 'a';
15.
16.     pipe(pfd);
17.     pid = fork();
18.
19.     if (pid == 0) { //子 写
20.         close(pfd[0]);
21.         while (1) {
22.             //aaaa\n
23.             for (i = 0; i < MAXLINE/2; i++)
24.                 buf[i] = ch;
25.             buf[i-1] = '\n';
26.             ch++;
```

```

27.         //bbbb\n
28.         for (; i < MAXLINE; i++)
29.             buf[i] = ch;
30.         buf[i-1] = '\n';
31.         ch++;
32.         //aaaa\nbbbb\n
33.         write(pfd[1], buf, sizeof(buf));
34.         sleep(5);
35.     }
36.     close(pfd[1]);
37.
38. } else if (pid > 0) {      //父 读
39.     struct epoll_event event;
40.     struct epoll_event reevent[10];      //epoll_wait 就绪返回 event
41.     int res, len;
42.
43.     close(pfd[1]);
44.     efd = epoll_create(10);
45.
46.     event.events = EPOLLIN | EPOLLET;      // ET 边沿触发
47.     // event.events = EPOLLIN;      // LT 水平触发 (默认)
48.     event.data.fd = pfd[0];
49.     epoll_ctl(efd, EPOLL_CTL_ADD, pfd[0], &event);
50.
51.     while (1) {
52.         res = epoll_wait(efd, reevent, 10, -1);
53.         printf("res %d\n", res);
54.         if (reevent[0].data.fd == pfd[0]) {
55.             len = read(pfd[0], buf, MAXLINE/2);
56.             write(STDOUT_FILENO, buf, len);
57.         }
58.     }
59.
60.     close(pfd[0]);
61.     close(efd);
62.
63. } else {
64.     perror("fork");
65.     exit(-1);
66. }
67.
68. return 0;
69. }

```

简单理解就是，水平触发就是有数据就触发，边沿触发是有新数据进来才触发。学电子的就比较清楚，触发器就有这个分类。

76P-网络中 ET 和 LT 模式

直接看代码，server 代码如下：

```
1. #include <stdio.h>
2. #include <string.h>
3. #include <netinet/in.h>
4. #include <arpa/inet.h>
5. #include <signal.h>
6. #include <sys/wait.h>
7. #include <sys/types.h>
8. #include <sys/epoll.h>
9. #include <unistd.h>
10.
11. #define MAXLINE 10
12. #define SERV_PORT 9000
13.
14. int main(void)
15. {
16.     struct sockaddr_in servaddr, cliaddr;
17.     socklen_t cliaddr_len;
18.     int listenfd, connfd;
19.     char buf[MAXLINE];
20.     char str[INET_ADDRSTRLEN];
21.     int efd;
22.
23.     listenfd = socket(AF_INET, SOCK_STREAM, 0);
24.
25.     bzero(&servaddr, sizeof(servaddr));
26.     servaddr.sin_family = AF_INET;
27.     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
28.     servaddr.sin_port = htons(SERV_PORT);
29.
30.     bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
31.
32.     listen(listenfd, 20);
33.
34.     struct epoll_event event;
35.     struct epoll_event revent[10];
36.     int res, len;
37.
38.     efd = epoll_create(10);
39.     event.events = EPOLLIN | EPOLLET;    /* ET 边沿触发 */
40.     //event.events = EPOLLIN;           /* 默认 LT 水平触发 */
41.
42.     printf("Accepting connections ...\n");
43.
44.     cliaddr_len = sizeof(cliaddr);
45.     connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
```

```

46.     printf("received from %s at PORT %d\n",
47.           inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
48.           ntohs(cliaddr.sin_port));
49.
50.     event.data.fd = connfd;
51.     epoll_ctl(efd, EPOLL_CTL_ADD, connfd, &event);
52.
53.     while (1) {
54.         res = epoll_wait(efd, revent, 10, -1);
55.
56.         printf("res %d\n", res);
57.         if (revent[0].data.fd == connfd) {
58.             len = read(connfd, buf, MAXLINE/2);           //readn(500)
59.             write(STDOUT_FILENO, buf, len);
60.         }
61.     }
62.
63.     return 0;
64. }

```

client 代码如下:

```

1. #include <stdio.h>
2. #include <string.h>
3. #include <unistd.h>
4. #include <arpa/inet.h>
5. #include <netinet/in.h>
6.
7. #define MAXLINE 10
8. #define SERV_PORT 9000
9.
10. int main(int argc, char *argv[])
11. {
12.     struct sockaddr_in servaddr;
13.     char buf[MAXLINE];
14.     int sockfd, i;
15.     char ch = 'a';
16.
17.     sockfd = socket(AF_INET, SOCK_STREAM, 0);
18.
19.     bzero(&servaddr, sizeof(servaddr));
20.     servaddr.sin_family = AF_INET;
21.     inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
22.     servaddr.sin_port = htons(SERV_PORT);
23.
24.     connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
25.
26.     while (1) {

```

```

27.     //aaaa\n
28.     for (i = 0; i < MAXLINE/2; i++)
29.         buf[i] = ch;
30.     buf[i-1] = '\n';
31.     ch++;
32.     //bbbb\n
33.     for (; i < MAXLINE; i++)
34.         buf[i] = ch;
35.     buf[i-1] = '\n';
36.     ch++;
37.     //aaaa\nbbbb\n
38.     write(sockfd, buf, sizeof(buf));
39.     sleep(5);
40. }
41. close(sockfd);
42.
43. return 0;
44. }

```

server 边沿触发，编译运行，结果如下：

```

zhcode@ubuntu:~/network/code76$ ./server
Accepting connections ...
received from 127.0.0.1 at PORT 33872
res 1
aaaa
res 1
bbbb
res 1
cccc
█

zhcode@ubuntu:~$ cd network/code76
zhcode@ubuntu:~/network/code76$ ./client

```

运行后，每过 5 秒钟服务器才输出一组字符，这就是边沿触发的效果。

更改服务器为水平触发模式，运行程序，如下：

```

zhcode@ubuntu:~/network/code76$ ./server
Accepting connections ...
received from 127.0.0.1 at PORT 33874
res 1
aaaa
res 1
bbbb
res 1
cccc
res 1
dddd
█

zhcode@ubuntu:~$ cd network/code76
zhcode@ubuntu:~/network/code76$ ./client

```

运行后，每 5 秒输出两组字符串，这是因为只写入了两组，这个模式的服务器，缓冲区有多少读多少。

ET 模式:

边沿触发:

缓冲区剩余未读尽的数据不会导致 `epoll_wait` 返回。 新的事件满足, 才会触发。

```
struct epoll_event event;
```

```
event.events = EPOLLIN | EPOLLET;
```

LT 模式:

水平触发 -- 默认采用模式。

缓冲区剩余未读尽的数据会导致 `epoll_wait` 返回。

77P-epoll 的 ET 非阻塞模式

readn 调用的阻塞，比如设定读 500 个字符，但是只读到 498，完事儿阻塞了，等另剩下的 2 个字符，然而在 server 代码里，一旦 read 变为 readn 阻塞了，它就不会被唤醒了，因为 epoll_wait 因为 readn 的阻塞不会循环执行，读不到新数据。有点死锁的意思，差俩字符所以阻塞，因为阻塞，读不到新字符。

LT(level triggered): LT 是缺省的工作方式，并且同时支持 block 和 no-block socket。在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的 fd 进行 IO 操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的 select/poll 都是这种模型的代表。

ET(edge-triggered): ET 是高速工作方式，只支持 no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过 epoll 告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知。请注意，如果一直不对这个 fd 作 IO 操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once)。

用 fcntl 设置阻塞

非阻塞 epoll 的服务器代码如下：

```
1. #include <stdio.h>
2. #include <string.h>
3. #include <netinet/in.h>
4. #include <arpa/inet.h>
5. #include <sys/wait.h>
6. #include <sys/types.h>
7. #include <sys/epoll.h>
8. #include <unistd.h>
9. #include <fcntl.h>
10.
11. #define MAXLINE 10
12. #define SERV_PORT 8000
13.
14. int main(void)
15. {
16.     struct sockaddr_in servaddr, cliaddr;
17.     socklen_t cliaddr_len;
18.     int listenfd, connfd;
19.     char buf[MAXLINE];
20.     char str[INET_ADDRSTRLEN];
21.     int efd, flag;
22.
23.     listenfd = socket(AF_INET, SOCK_STREAM, 0);
24.
25.     bzero(&servaddr, sizeof(servaddr));
26.     servaddr.sin_family = AF_INET;
27.     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```

28.     servaddr.sin_port = htons(SERV_PORT);
29.
30.     bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
31.
32.     listen(listenfd, 20);
33.
34.     //////////////////////////////////////
35.     struct epoll_event event;
36.     struct epoll_event res_event[10];
37.     int res, len;
38.
39.     efd = epoll_create(10);
40.
41.     event.events = EPOLLIN | EPOLLET;      /* ET 边沿触发，默认是水平触发 */
42.
43.     //event.events = EPOLLIN;
44.     printf("Accepting connections ...\n");
45.     cliaddr_len = sizeof(cliaddr);
46.     connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
47.     printf("received from %s at PORT %d\n",
48.           inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
49.           ntohs(cliaddr.sin_port));
50.
51.     flag = fcntl(connfd, F_GETFL);          /* 修改 connfd 为非阻塞读 */
52.     flag |= O_NONBLOCK;
53.     fcntl(connfd, F_SETFL, flag);
54.
55.     event.data.fd = connfd;
56.     epoll_ctl(efd, EPOLL_CTL_ADD, connfd, &event);    //将 connfd 加入监听红黑树
57.     while (1) {
58.         printf("epoll_wait begin\n");
59.         res = epoll_wait(efd, res_event, 10, -1);      //最多 10 个，阻塞监听
60.         printf("epoll_wait end res %d\n", res);
61.
62.         if (res_event[0].data.fd == connfd) {
63.             while ((len = read(connfd, buf, MAXLINE/2)) > 0)    //非阻塞读，轮询
64.                 write(STDOUT_FILENO, buf, len);
65.         }
66.     }
67.
68.     return 0;
69. }

```

其实就是多了这几行：

```

51     flag = fcntl(connfd, F_GETFL);          /* 修改connfd为非阻塞读 */
52     flag |= O_NONBLOCK;
53     fcntl(connfd, F_SETFL, flag);

```


结论:

epoll 的 ET 模式, 高效模式, 但是只支持 非阻塞模式。 --- 忙轮询。

```
struct epoll_event event;  
  
event.events = EPOLLIN | EPOLLET;  
  
epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &event);  
  
int flg = fcntl(cfd, F_GETFL);  
  
flg |= O_NONBLOCK;  
  
fcntl(cfd, F_SETFL, flg);
```

优点:

高效。突破 1024 文件描述符。

缺点:

不能跨平台。 Linux。

后面使用 epoll 就用这种非阻塞的

78P-epoll 优缺点总结

优点：

高效。突破 1024 文件描述符。

缺点：

不能跨平台。 Linux。

79P-补充对比 ET 和 LT

这里重要的就一点，当使用非阻塞读时，读取数据需要轮询。

比如使用 `readn` 的时候，数据没读够，因为非阻塞，跑了，想读剩下的，就得轮询。

80P-epoll 反应堆模型总述

epoll 反应堆模型：

epoll ET 模式 + 非阻塞、轮询 + void *ptr。

原来： socket、bind、listen -- epoll_create 创建监听 红黑树 -- 返回 epfd -- epoll_ctl() 向树上添加一个监听 fd -- while (1) --

-- epoll_wait 监听 -- 对应监听 fd 有事件产生 -- 返回 监听满足数组。 -- 判断返回数组元素 -- lfd 满足 -- Accept -- cfd 满足

-- read() --- 小->大 -- write 回去。

反应堆：不但要监听 cfd 的读事件、还要监听 cfd 的写事件。

socket、bind、listen -- epoll_create 创建监听 红黑树 -- 返回 epfd -- epoll_ctl() 向树上添加一个监听 fd -- while (1) --

-- epoll_wait 监听 -- 对应监听 fd 有事件产生 -- 返回 监听满足数组。 -- 判断返回数组元素 -- lfd 满足 -- Accept -- cfd 满足

-- read() --- 小->大 -- cfd 从监听红黑树上摘下 -- EPOLLOUT -- 回调函数 -- epoll_ctl() -- EPOLL_CTL_ADD 重新放到红黑上监听写事件

-- 等待 epoll_wait 返回 -- 说明 cfd 可写 -- write 回去 -- cfd 从监听红黑树上摘下 -- EPOLLIN

-- epoll_ctl() -- EPOLL_CTL_ADD 重新放到红黑上监听读事件 -- epoll_wait 监听

反应堆的理解：加入 IO 转接之后，有了事件，server 才去处理，这里反应堆也是这样，由于网络环境复杂，服务器处理数据之后，可能并不能直接写回去，比如遇到网络繁忙或者对方缓冲区已经满了这种情况，就不能直接写回给客户端。反应堆就是在处理数据之后，监听写事件，能写会客户端了，才去做写回操作。写回之后，再改为监听读事件。如此循环。

81P-epoll 反应堆 main 逻辑

直接上代码，这就略微有点长了：

```
1.  /*
2.  *epoll 基于非阻塞 I/O 事件驱动
3.  */
4.  #include <stdio.h>
5.  #include <sys/socket.h>
6.  #include <sys/epoll.h>
7.  #include <arpa/inet.h>
8.  #include <fcntl.h>
9.  #include <unistd.h>
10. #include <errno.h>
11. #include <string.h>
12. #include <stdlib.h>
13. #include <time.h>
14.
15. #define MAX_EVENTS  1024                //监听上限数
16. #define BUFLen 4096
17. #define SERV_PORT   8080
18.
19. void recvddata(int fd, int events, void *arg);
20. void senddata(int fd, int events, void *arg);
21.
22. /* 描述就绪文件描述符相关信息 */
23.
24. struct myevent_s {
25.     int fd;                //要监听的文件描述符
26.     int events;            //对应的监听事件
27.     void *arg;            //泛型参数
28.     void (*call_back)(int fd, int events, void *arg); //回调函数
29.     int status;            //是否在监听:1->在红黑树上(监听), 0->不在
                             (不监听)
30.     char buf[BUFLen];
31.     int len;
32.     long last_active;      //记录每次加入红黑树 g_efd 的时间值
33. };
34.
35. int g_efd;                //全局变量, 保存 epoll_create 返回的文件描
                             述符
36. struct myevent_s g_events[MAX_EVENTS+1]; //自定义结构体类型数组. +1-->listen fd
37.
38.
39. /*将结构体 myevent_s 成员变量 初始化*/
40.
41. void eventset(struct myevent_s *ev, int fd, void (*call_back)(int, int, void *), void *arg)
42. {
43.     ev->fd = fd;
```

```

44.     ev->call_back = call_back;
45.     ev->events = 0;
46.     ev->arg = arg;
47.     ev->status = 0;
48.     memset(ev->buf, 0, sizeof(ev->buf));
49.     ev->len = 0;
50.     ev->last_active = time(NULL);           //调用 eventset 函数的时间
51.
52.     return;
53. }
54.
55. /* 向 epoll 监听的红黑树 添加一个 文件描述符 */
56.
57. //eventadd(efd, EPOLLIN, &g_events[MAX_EVENTS]);
58. void eventadd(int efd, int events, struct myevent_s *ev)
59. {
60.     struct epoll_event epv = {0, {0}};
61.     int op;
62.     epv.data.ptr = ev;
63.     epv.events = ev->events = events;       //EPOLLIN 或 EPOLLOUT
64.
65.     if (ev->status == 0) {                  //已经在红黑树 g_efd 里
66.         op = EPOLL_CTL_ADD;                //将其加入红黑树 g_efd, 并将 status 置 1
67.         ev->status = 1;
68.     }
69.
70.     if (epoll_ctl(efd, op, ev->fd, &epv) < 0) //实际添加/修改
71.         printf("event add failed [fd=%d], events[%d]\n", ev->fd, events);
72.     else
73.         printf("event add OK [fd=%d], op=%d, events[%0X]\n", ev->fd, op, events);
74.
75.     return ;
76. }
77.
78. /* 从 epoll 监听的 红黑树中删除一个 文件描述符*/
79.
80. void eventdel(int efd, struct myevent_s *ev)
81. {
82.     struct epoll_event epv = {0, {0}};
83.
84.     if (ev->status != 1)                    //不在红黑树上
85.         return ;
86.
87.     //epv.data.ptr = ev;
88.     epv.data.ptr = NULL;
89.     ev->status = 0;                          //修改状态
90.     epoll_ctl(efd, EPOLL_CTL_DEL, ev->fd, &epv); //从红黑树 efd 上将 ev->fd 摘除
91.
92.     return ;

```

```

93. }
94.
95. /* 当有文件描述符就绪，epoll 返回，调用该函数 与客户端建立链接 */
96.
97. void acceptconn(int lfd, int events, void *arg)
98. {
99.     struct sockaddr_in cin;
100.     socklen_t len = sizeof(cin);
101.     int cfd, i;
102.
103.     if ((cfd = accept(lfd, (struct sockaddr *)&cin, &len)) == -1) {
104.         if (errno != EAGAIN && errno != EINTR) {
105.             /* 暂时不做出错处理 */
106.         }
107.         printf("%s: accept, %s\n", __func__, strerror(errno));
108.         return ;
109.     }
110.
111.     do {
112.         for (i = 0; i < MAX_EVENTS; i++) //从全局数组 g_events 中找一个空闲元素
113.             if (g_events[i].status == 0) //类似于 select 中找值为-1的元素
114.                 break; //跳出 for
115.
116.         if (i == MAX_EVENTS) {
117.             printf("%s: max connect limit[%d]\n", __func__, MAX_EVENTS);
118.             break; //跳出 do while(0) 不执行后
119.         } //续代码
120.
121.         int flag = 0;
122.         if ((flag = fcntl(cfd, F_SETFL, O_NONBLOCK)) < 0) { //将 cfd 也设置为非阻塞
123.             printf("%s: fcntl nonblocking failed, %s\n", __func__, strerror(errno));
124.             break;
125.         }
126.
127.         /* 给 cfd 设置一个 myevent_s 结构体，回调函数 设置为 recvdata */
128.         eventset(&g_events[i], cfd, recvdata, &g_events[i]);
129.         eventadd(g_efd, EPOLLIN, &g_events[i]); //将 cfd 添加到红黑树 g_efd 中，
130.         //监听读事件
131.     } while(0);
132.
133.     printf("new connect [%s:%d][time:%ld], pos[%d]\n",
134.           inet_ntoa(cin.sin_addr), ntohs(cin.sin_port), g_events[i].last_active, i);
135.     return ;
136. }
137.

```

```

138. void recvdata(int fd, int events, void *arg)
139. {
140.     struct myevent_s *ev = (struct myevent_s *)arg;
141.     int len;
142.
143.     len = recv(fd, ev->buf, sizeof(ev->buf), 0);           //读文件描述符，数据存入 myevent_s 成员
                                                                buf 中
144.
145.     eventdel(g_efd, ev);           //将该节点从红黑树上摘除
146.
147.     if (len > 0) {
148.
149.         ev->len = len;
150.         ev->buf[len] = '\0';           //手动添加字符串结束标记
151.         printf("C[%d]:%s\n", fd, ev->buf);
152.
153.         eventset(ev, fd, senddata, ev);           //设置该 fd 对应的回调函数为 senddata
154.         eventadd(g_efd, EPOLLOUT, ev);           //将 fd 加入红黑树 g_efd 中,监听其写事件
155.
156.     } else if (len == 0) {
157.         close(ev->fd);
158.         /* ev-g_events 地址相减得到偏移元素位置 */
159.         printf("[fd=%d] pos[%d], closed\n", fd, ev-g_events);
160.     } else {
161.         close(ev->fd);
162.         printf("recv[fd=%d] error[%d]:%s\n", fd, errno, strerror(errno));
163.     }
164.
165.     return;
166. }
167.
168. void senddata(int fd, int events, void *arg)
169. {
170.     struct myevent_s *ev = (struct myevent_s *)arg;
171.     int len;
172.
173.     len = send(fd, ev->buf, ev->len, 0);           //直接将数据 回写给客户端。未作处理
174.
175.     eventdel(g_efd, ev);           //从红黑树 g_efd 中移除
176.
177.     if (len > 0) {
178.
179.         printf("send[fd=%d], [%d]%s\n", fd, len, ev->buf);
180.         eventset(ev, fd, recvdata, ev);           //将该 fd 的 回调函数改为 recvdata
181.         eventadd(g_efd, EPOLLIN, ev);           //从新添加到红黑树上， 设为监听读事件
182.
183.     } else {
184.         close(ev->fd);           //关闭链接
185.         printf("send[fd=%d] error %s\n", fd, strerror(errno));

```



```

186.     }
187.
188.     return ;
189.}
190.
191./*创建 socket, 初始化 lfd */
192.
193.void initlistensocket(int efd, short port)
194.{
195.    struct sockaddr_in sin;
196.
197.    int lfd = socket(AF_INET, SOCK_STREAM, 0);
198.    fcntl(lfd, F_SETFL, O_NONBLOCK); //将 socket 设为非阻塞
199.
200.    memset(&sin, 0, sizeof(sin)); //bzero(&sin, sizeof(sin))
201.    sin.sin_family = AF_INET;
202.    sin.sin_addr.s_addr = INADDR_ANY;
203.    sin.sin_port = htons(port);
204.
205.    bind(lfd, (struct sockaddr *)&sin, sizeof(sin));
206.
207.    listen(lfd, 20);
208.
209.    /* void eventset(struct myevent_s *ev, int fd, void (*call_back)(int, int, void *), void *arg);
    */
210.    eventset(&g_events[MAX_EVENTS], lfd, acceptconn, &g_events[MAX_EVENTS]);
211.
212.    /* void eventadd(int efd, int events, struct myevent_s *ev) */
213.    eventadd(efd, EPOLLIN, &g_events[MAX_EVENTS]);
214.
215.    return ;
216.}
217.
218.int main(int argc, char *argv[])
219.{
220.    unsigned short port = SERV_PORT;
221.
222.    if (argc == 2)
223.        port = atoi(argv[1]); //使用用户指定端口.如未指定,用默认端口
224.
225.    g_efd = epoll_create(MAX_EVENTS+1); //创建红黑树,返回给全局 g_efd
226.    if (g_efd <= 0)
227.        printf("create efd in %s err %s\n", __func__, strerror(errno));
228.
229.    initlistensocket(g_efd, port); //初始化监听 socket
230.
231.    struct epoll_event events[MAX_EVENTS+1]; //保存已经满足就绪事件的文件描述符数组

```

```

232.     printf("server running:port[%d]\n", port);
233.
234.     int checkpos = 0, i;
235.     while (1) {
236.         /* 超时验证, 每次测试 100 个链接, 不测试 listenfd 当客户端 60 秒内没有和服务器通信, 则关闭此客户端链
           接 */
237.
238.         long now = time(NULL);                                //当前时间
239.         for (i = 0; i < 100; i++, checkpos++) {                //一次循环检测 100 个。使用 checkpos 控制检测对
           象
240.             if (checkpos == MAX_EVENTS)
241.                 checkpos = 0;
242.             if (g_events[checkpos].status != 1)                //不在红黑树 g_efd 上
243.                 continue;
244.
245.             long duration = now - g_events[checkpos].last_active; //客户端不活跃的时间
246.
247.             if (duration >= 60) {
248.                 close(g_events[checkpos].fd);                  //关闭与该客户端链接
249.                 printf("[fd=%d] timeout\n", g_events[checkpos].fd);
250.                 eventdel(g_efd, &g_events[checkpos]);          //将该客户端 从红黑树 g_efd
           移除
251.             }
252.         }
253.
254.         /*监听红黑树 g_efd, 将满足的事件的文件描述符加至 events 数组中, 1 秒没有事件满足, 返回 0*/
255.         int nfd = epoll_wait(g_efd, events, MAX_EVENTS+1, 1000);
256.         if (nfd < 0) {
257.             printf("epoll_wait error, exit\n");
258.             break;
259.         }
260.
261.         for (i = 0; i < nfd; i++) {
262.             /*使用自定义结构体 myevent_s 类型指针, 接收 联合体 data 的 void *ptr 成员*/
263.             struct myevent_s *ev = (struct myevent_s *)events[i].data.ptr;
264.
265.             if ((events[i].events & EPOLLIN) && (ev->events & EPOLLIN)) { //读就绪事件
266.                 ev->call_back(ev->fd, events[i].events, ev->arg);
267.                 //lfd  EPOLLIN
268.             }
269.             if ((events[i].events & EPOLLOUT) && (ev->events & EPOLLOUT)) { //写就绪事件
270.                 ev->call_back(ev->fd, events[i].events, ev->arg);
271.             }
272.         }
273.     }
274.
275.     /* 退出前释放所有资源 */
276.     return 0;
277. }

```

main 逻辑：创建套接字—》初始化连接—》超时验证—》监听—》处理读事件和写事件

82P-epoll 反应堆-给 lfd 和 cfd 指定回调函数

`eventset` 函数指定了不同事件对应的回调函数，所以虽然读写事件都用的 `call_back` 来回调，但实际上调用的是不同的函数。

83P-epoll 反应堆 initlistensocket 小总结

eventset 函数:

设置回调函数。 lfd --》 acceptconn()

cfd --> recvddata();

cfd --> senddata();

eventadd 函数:

将一个 fd, 添加到 监听红黑树。 设置监听 read 事件, 还是监听写事件。

84P-epoll 反应堆 wait 被触发后 read 和 write 回调及监听

网络编程中: read --- recv()

 write --- send();

85P-epoll 反应堆-超时时间

用一个 `last_active` 存储上次活跃时间，完事儿用当前时间和上次活跃时间来计算不活跃时间长度，不活跃时间超过一定阈值，就踢掉这个客户端。

86-总结

多路 IO 转接:

select:

poll:

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

`fds`: 监听的文件描述符【数组】

```
struct pollfd {
```

```
    int fd:    待监听的文件描述符
```

```
    short events:    待监听的文件描述符对应的监听事件
```

取值: POLLIN、POLLOUT、POLLERR

```
    short revnets:    传入时, 给 0。如果满足对应事件的话, 返回 非 0 --> POLLIN、POLLOUT、POLLERR
```

```
}
```

`nfds`: 监听数组的, 实际有效监听个数。

`timeout`: > 0 : 超时时长。单位: 毫秒。

-1 : 阻塞等待

0 : 不阻塞

返回值: 返回满足对应监听事件的文件描述符 总个数。

优点:

自带数组结构。 可以将 监听事件集合 和 返回事件集合 分离。

拓展 监听上限。 超出 1024 限制。

缺点:

不能跨平台。 Linux

无法直接定位满足监听事件的文件描述符, 编码难度较大。

read 函数返回值:

> 0 : 实际读到的字节数

=0: socket 中, 表示对端关闭。close ()

-1: 如果 errno == EINTR 被异常终端。需要重启。

如果 errno == EAGAIN 或 EWOULDBLOCK 以非阻塞方式读数据, 但是没有数据。需要, 再次读。

如果 errno == ECONNRESET 说明连接被重置。需要 close (), 移除监听队列。

错误。

突破 1024 文件描述符限制:

cat /proc/sys/fs/file-max --> 当前计算机所能打开的最大文件个数。受硬件影响。

ulimit -a --> 当前用户下的进程, 默认打开文件描述符个数。缺省为 1024

修改:

打开 sudo vi /etc/security/limits.conf, 写入:

* soft nfile 65536 --> 设置默认值, 可以直接借助命令修改。【注销用户, 使其生效】

* hard nfile 100000 --> 命令修改上限。

epoll:

int epoll_create(int size); 创建一棵监听红黑树

size: 创建的红黑树的监听节点数量。(仅供内核参考。)

返回值: 指向新创建的红黑树的根节点的 fd。

失败: -1 errno

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event); 操作监听红黑树

epfd: epoll_create 函数的返回值。epfd

op: 对该监听红黑数所做的操作。

EPOLL_CTL_ADD 添加 fd 到 监听红黑树

EPOLL_CTL_MOD 修改 fd 在 监听红黑树上的监听事件。

EPOLL_CTL_DEL 将一个 fd 从监听红黑树上摘下 (取消监听)

fd:

待监听的 fd

event: 本质 struct epoll_event 结构体 地址

成员 events:

EPOLLIN / EPOLLOUT / EPOLLERR

成员 data: 联合体（共用体）:

int fd; 对应监听事件的 fd

void *ptr;

uint32_t u32;

uint64_t u64;

返回值: 成功 0; 失败: -1 errno

int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
阻塞监听。

epfd: epoll_create 函数的返回值。 epfd

events: 传出参数,【数组】, 满足监听条件的 哪些 fd 结构体。

maxevents: 数组 元素的总个数。 1024

struct epoll_event evnets[1024]
timeout:

-1: 阻塞

0: 不阻塞

>0: 超时时间 （毫秒）

返回值:

> 0: 满足监听的 总个数。 可以用作循环上限。

0: 没有 fd 满足监听事件

-1: 失败。 errno

epoll 实现多路 IO 转接思路:

```

lfd = socket ( ) ;          监听连接事件 lfd
bind();
listen();

int epfd = epoll_create(1024);          epfd, 监听红黑树的树根。

struct epoll_event tep, ep[1024];          tep, 用来设置单个 fd 属性, ep 是 epoll_wait()
传出的满足监听事件的数组。

tep.events = EPOLLIN;          初始化 lfd 的监听属性。
tep.data.fd = lfd

epoll_ctl(epfd, EPOLL_CTL_ADD, lfd, &tep);          将 lfd 添加到监听红黑树上。

while (1) {

    ret = epoll_wait(epfd, ep, 1024, -1);          实施监听

    for (i = 0; i < ret; i++) {

        if (ep[i].data.fd == lfd) {          // lfd 满足读事件, 有新的客户端发起连接请
求

            cfd = Accept();

            tep.events = EPOLLIN;          初始化 cfd 的监听属性。
            tep.data.fd = cfd;

            epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &tep);

        } else {          cfd 们 满足读事件, 有客户端写数据来。

            n = read(ep[i].data.fd, buf, sizeof(buf));

            if ( n == 0) {

                close(ep[i].data.fd);

                epoll_ctl(epfd, EPOLL_CTL_DEL, ep[i].data.fd , NULL); // 将关闭的 cfd,
从监听树上摘下。

            } else if (n > 0) {

                小--大
                write(ep[i].data.fd, buf, n);
            }
        }
    }
}

```

```
}
```

epoll 事件模型:

ET 模式:

边沿触发:

缓冲区剩余未读尽的数据不会导致 `epoll_wait` 返回。 新的事件满足, 才会触发。

```
struct epoll_event event;
```

```
event.events = EPOLLIN | EPOLLET;
```

LT 模式:

水平触发 -- 默认采用模式。

缓冲区剩余未读尽的数据会导致 `epoll_wait` 返回。

结论:

epoll 的 ET 模式, 高效模式, 但是只支持 非阻塞模式。 --- 忙轮询。

```
struct epoll_event event;
```

```
event.events = EPOLLIN | EPOLLET;
```

```
epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &event);
```

```
int flg = fcntl(cfd, F_GETFL);
```

```
flg |= O_NONBLOCK;
```

```
fcntl(cfd, F_SETFL, flg);
```

优点:

高效。突破 1024 文件描述符。

缺点:

不能跨平台。 Linux。

epoll 反应堆模型:

epoll ET 模式 + 非阻塞、轮询 + `void *ptr`。

原来: `socket`、`bind`、`listen` -- `epoll_create` 创建监听 红黑树 -- 返回 `epfd` -- `epoll_ctl()` 向树上添加一个监听 `fd` -- `while (1)` --

-- epoll_wait 监听 -- 对应监听 fd 有事件产生 -- 返回 监听满足数组。 -- 判断返回数组元素 -- lfd 满足 -- Accept -- cfd 满足

-- read() --- 小->大 -- write 回去。

反应堆：不但要监听 cfd 的读事件、还要监听 cfd 的写事件。

socket、bind、listen -- epoll_create 创建监听 红黑树 -- 返回 epfd -- epoll_ctl() 向树上添加一个监听 fd -- while (1) --

-- epoll_wait 监听 -- 对应监听 fd 有事件产生 -- 返回 监听满足数组。 -- 判断返回数组元素 -- lfd 满足 -- Accept -- cfd 满足

-- read() --- 小->大 -- cfd 从监听红黑树上摘下 -- EPOLLOUT -- 回调函数 -- epoll_ctl() -- EPOLL_CTL_ADD 重新放到红黑上监听写事件

-- 等待 epoll_wait 返回 -- 说明 cfd 可写 -- write 回去 -- cfd 从监听红黑树上摘下 -- EPOLLIN

-- epoll_ctl() -- EPOLL_CTL_ADD 重新放到红黑上监听读事件 -- epoll_wait 监听

eventset 函数：

设置回调函数。 lfd --> acceptconn()

cfd --> recvdata();

cfd --> senddata();

eventadd 函数：

将一个 fd， 添加到 监听红黑树。 设置监听 read 事件，还是监听写事件。

网络编程中： read --- recv()

write --- send();

87P-复习

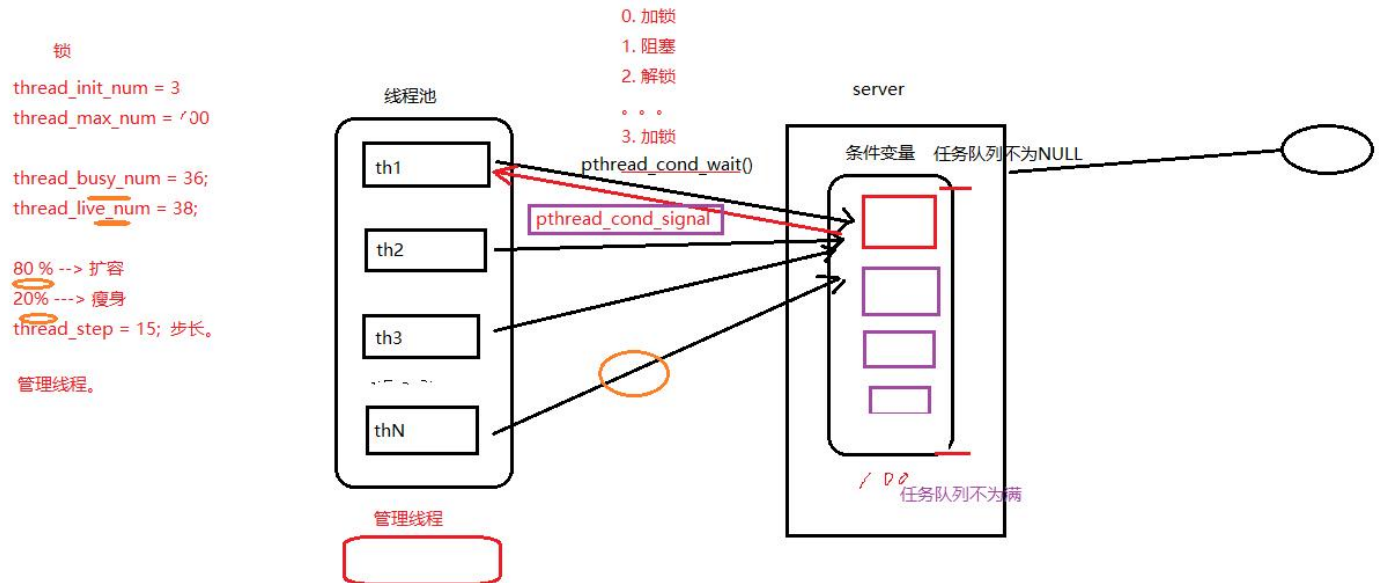
88P-补充说明 epoll 的 man 手册

89P-epoll 反应堆再说明

90P-ctags 使用



91P 线程池模型原理分析



```
struct threadpool_t {

    pthread_mutex_t lock; /* 用于锁住本结构体 */
    pthread_mutex_t thread_counter; /* 记录忙状态线程个数 de 琐 -- busy_thr_num */

    pthread_cond_t queue_not_full; /* 当任务队列满时，添加任务的线程阻塞，等待此条
    件变量 */
    pthread_cond_t queue_not_empty; /* 任务队列里不为空时，通知等待任务的线程 */

    pthread_t *threads; /* 存放线程池中每个线程的 tid。数组 */
    pthread_t adjust_tid; /* 存管理线程 tid */
    threadpool_task_t *task_queue; /* 任务队列 (数组首地址) */

    int min_thr_num; /* 线程池最小线程数 */
    int max_thr_num; /* 线程池最大线程数 */
    int live_thr_num; /* 当前存活线程个数 */
    int busy_thr_num; /* 忙状态线程个数 */
    int wait_exit_thr_num; /* 要销毁的线程个数 */

    int queue_front; /* task_queue 队头下标 */
    int queue_rear; /* task_queue 队尾下标 */
    int queue_size; /* task_queue 队中实际任务数 */
    int queue_max_size; /* task_queue 队列可容纳任务数上限 */

    int shutdown; /* 标志位，线程池使用状态，true 或 false */
};
```

```
typedef struct {  
  
    void *(*function)(void *);    /* 函数指针，回调函数 */  
    void *arg;                    /* 上面函数的参数 */  
}
```


93P-线程池 main 架构

1. `main()`;

创建线程池。

向线程池中添加任务。 借助回调处理任务。

销毁线程池。

94P-线程池-pthreadpool_create

2. pthreadpool_create();

创建线程池结构体 指针。

初始化线程池结构体 { N 个成员变量 }

创建 N 个任务线程。

创建 1 个管理者线程。

失败时，销毁开辟的所有空间。（释放）

95P-子线程回调函数

3. threadpool_thread ()

进入子线程回调函数。

接收参数 void *arg --》 pool 结构体

加锁 --》 lock --》 整个结构体锁

判断条件变量 --》 wait -----170

96P-管理者线程

4. adjust_thread ()

循环 10 s 执行一次。

进入管理者线程回调函数

接收参数 void *arg --》 pool 结构体

加锁 --》 lock --》 整个结构体锁

获取管理线程池要用的到 变量。 task_num, live_num, busy_num

根据既定算法，使用上述 3 变量，判断是否应该 创建、销毁线程池中 指定步长的线程。

97P-threadpool_add 函数

5. threadpool_add ()

总功能：

模拟产生任务。 num[20]

设置回调函数， 处理任务。 sleep (1) 代表处理完成。

内部实现：

加锁

初始化 任务队列结构体成员。 回调函数 function, arg

利用环形队列机制，实现添加任务。 借助队尾指针挪移 % 实现。

唤醒阻塞在 条件变量上的线程。

解锁

98P-条件满足，子线程 wait 被唤醒后处理任务

6. 从 3. 中的 wait 之后继续执行，处理任务。

加锁

获取 任务处理回调函数，及参数

利用环形队列机制，实现处理任务。 借助队头指针挪移 % 实现。

唤醒阻塞在 条件变量 上的 server。

解锁

加锁

改忙线程数++

解锁

执行处理任务的线程

加锁

改忙线程数--

解锁

99P-线程池扩容和销毁

7. 创建 销毁线程

管理者线程根据 `task_num`, `live_num`, `busy_num`

根据既定算法，使用上述 3 变量，判断是否应该 创建、销毁线程池中 指定步长的线程。

如果满足 创建条件

```
pthread_create();    回调 任务线程函数。        live_num++
```

如果满足 销毁条件

```
wait_exit_thr_num = 10;
```

signal 给 阻塞在条件变量上的线程 发送 假条件满足信号

跳转至 --170 wait 阻塞线程会被 假信号 唤醒。判断： `wait_exit_thr_num > 0`
`pthread_exit();`

100P-TCP 和 UDP 通信优缺点

TCP 通信和 UDP 通信各自的优缺点：

TCP： 面向连接的，可靠数据包传输。对于不稳定的网络层，采取完全弥补的通信方式。 丢包重传。

优点：

稳定。

数据流量稳定、速度稳定、顺序

缺点：

传输速度慢。相率低。开销大。

使用场景：数据的完整型要求较高，不追求效率。

大数据传输、文件传输。

UDP： 无连接的，不可靠的数据报传递。对于不稳定的网络层，采取完全不弥补的通信方式。默认还原网络状况

优点：

传输速度快。相率高。开销小。

缺点：

不稳定。

数据流量。速度。顺序。

使用场景：对时效性要求较高场合。稳定性其次。

游戏、视频会议、视频电话。 腾讯、华为、阿里 --- 应用层数据校验协议，
弥补 udp 的不足。

101P-UDP 通信 server 和 client 流程

UDP 实现的 C/S 模型:

recv()/send() 只能用于 TCP 通信。 替代 read、write

accept(); ---- connect(); ---被舍弃

server:

lfd = socket(AF_INET, STREAM, 0); SOCK_DGRAM --- 报式协议。

bind();

listen(); --- 可有可无

while (1) {

read(cfd, buf, sizeof) --- 被替换 --- recvfrom() --- 涵盖 accept 传出地址结构。

小-- 大

write();--- 被替换 --- sendto() ---- connect
}

close();

client:

connfd = socket(AF_INET, SOCK_DGRAM, 0);

sendto('服务器的地址结构', 地址结构大小)

recvfrom()

写到屏幕

close();

102P-recvfrom 和 sendto 函数

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);
```

sockfd: 套接字

buf: 缓冲区地址

len: 缓冲区大小

flags: 0

src_addr: (struct sockaddr *) &addr 传出。 对端地址结构

addrlen: 传入传出。

返回值: 成功接收数据字节数。 失败: -1 errno。 0: 对端关闭。

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen);
```

sockfd: 套接字

buf: 存储数据的缓冲区

len: 数据长度

flags: 0

src_addr: (struct sockaddr *) &addr 传入。 目标地址结构

addrlen: 地址结构长度。

返回值: 成功写出数据字节数。 失败 -1, errno

103P-UDP 实现的并发服务器和客户端

直接上代码，啃，啃就完事儿，这是服务器代码

```
1. #include <string.h>
2. #include <stdio.h>
3. #include <unistd.h>
4. #include <arpa/inet.h>
5. #include <ctype.h>
6.
7. #define SERV_PORT 8000
8.
9. int main(void)
10. {
11.     struct sockaddr_in serv_addr, clie_addr;
12.     socklen_t clie_addr_len;
13.     int sockfd;
14.     char buf[BUFSIZ];
15.     char str[INET_ADDRSTRLEN];
16.     int i, n;
17.
18.     sockfd = socket(AF_INET, SOCK_DGRAM, 0);
19.
20.     bzero(&serv_addr, sizeof(serv_addr));
21.     serv_addr.sin_family = AF_INET;
22.     serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
23.     serv_addr.sin_port = htons(SERV_PORT);
24.
25.     bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
26.
27.     printf("Accepting connections ...\n");
28.     while (1) {
29.         clie_addr_len = sizeof(clie_addr);
30.         n = recvfrom(sockfd, buf, BUFSIZ, 0, (struct sockaddr *)&clie_addr, &clie_addr_len);
31.         if (n == -1)
32.             perror("recvfrom error");
33.
34.         printf("received from %s at PORT %d\n",
35.             inet_ntop(AF_INET, &clie_addr.sin_addr, str, sizeof(str)),
36.             ntohs(clie_addr.sin_port));
37.
38.         for (i = 0; i < n; i++)
39.             buf[i] = toupper(buf[i]);
40.
41.         n = sendto(sockfd, buf, n, 0, (struct sockaddr *)&clie_addr, sizeof(clie_addr));
42.         if (n == -1)
43.             perror("sendto error");
44.     }
45.
```

```
46.     close(sockfd);
47.
48.     return 0;
49. }
```

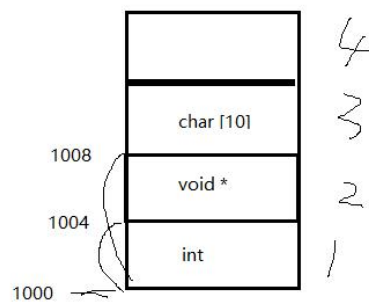
下面是客户端代码：

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <unistd.h>
4.  #include <arpa/inet.h>
5.  #include <ctype.h>
6.
7.  #define SERV_PORT 8000
8.
9.  int main(int argc, char *argv[])
10. {
11.     struct sockaddr_in servaddr;
12.     int sockfd, n;
13.     char buf[BUFSIZ];
14.
15.     sockfd = socket(AF_INET, SOCK_DGRAM, 0);
16.
17.     bzero(&servaddr, sizeof(servaddr));
18.     servaddr.sin_family = AF_INET;
19.     inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
20.     servaddr.sin_port = htons(SERV_PORT);
21.
22.     bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
23.
24.     while (fgets(buf, BUFSIZ, stdin) != NULL) {
25.         n = sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr *)&servaddr, sizeof(servaddr));
26.         if (n == -1)
27.             perror("sendto error");
28.
29.         n = recvfrom(sockfd, buf, BUFSIZ, 0, NULL, 0); //NULL:不关心对端信息
30.         if (n == -1)
31.             perror("recvfrom error");
32.
33.         write(STDOUT_FILENO, buf, n);
34.     }
35.
36.     close(sockfd);
37.
38.     return 0;
39. }
```

104P-借助 TCP 的 CS 模型，改写 UDP 的 CS 模型

看懂前面的，问题就不大了。可以再看一下视频复习复习

105P-本地套接字和网络套接字比较



本地套接字:

IPC: pipe、fifo、mmap、信号、本地套 (domain) —— CS 模型

对比网络编程 TCP C/S 模型, 注意以下几点:

1. `int socket(int domain, int type, int protocol);` 参数 `domain`: `AF_INET` --> `AF_UNIX/AF_LOCAL`

`type`: `SOCK_STREAM/SOCK_DGRAM` 都可以。

2. 地址结构: `sockaddr_in` --> `sockaddr_un`

```
struct sockaddr_in srv_addr; --> struct sockaddr_un srv_addr;
```

```
srv_addr.sin_family = AF_INET; --> srv_addr.sun_family = AF_UNIX;
```

•

```
srv_addr.sin_port = htons(8888); strcpy(srv_addr.sun_path, "srv.socket")
```

```
srv_addr.sin_addr.s_addr = htonl(INADDR_ANY); len = offsetof(struct  
sockaddr_un, sun_path) + strlen("srv.socket");
```

```
bind(fd, (struct sockaddr *)&srv_addr, sizeof(srv_addr)); --> bind(fd,  
(struct sockaddr *)&srv_addr, len);
```

3. `bind()` 函数调用成功, 会创建一个 socket。因此为保证 `bind` 成功, 通常我们在 `bind` 之前, 可以使用 `unlink("srv.socket");`

4. 客户端不能依赖 “隐式绑定”。并且应该在通信建立过程中, 创建且初始化 2 个地址结构:

1) `client_addr` --> `bind()`

2) `server_addr` --> `connect()`;

106P-本地套接字通信

服务器代码:

```
1. #include <stdio.h>
2. #include <unistd.h>
3. #include <sys/socket.h>
4. #include <strings.h>
5. #include <string.h>
6. #include <ctype.h>
7. #include <arpa/inet.h>
8. #include <sys/un.h>
9. #include <stddef.h>
10.
11. #include "wrap.h"
12.
13. #define SERV_ADDR  "serv.socket"
14.
15. int main(void)
16. {
17.     int lfd, cfd, len, size, i;
18.     struct sockaddr_un servaddr, cliaddr;
19.     char buf[4096];
20.
21.     lfd = Socket(AF_UNIX, SOCK_STREAM, 0);
22.
23.     bzero(&servaddr, sizeof(servaddr));
24.     servaddr.sun_family = AF_UNIX;
25.     strcpy(servaddr.sun_path, SERV_ADDR);
26.
27.     len = offsetof(struct sockaddr_un, sun_path) + strlen(servaddr.sun_path);    /* servaddr total len */
28.
29.     unlink(SERV_ADDR);    /* 确保 bind 之前 serv.sock 文件不存在,bind 会创建该文件 */
30.     Bind(lfd, (struct sockaddr *)&servaddr, len);    /* 参3 不能是 sizeof(servaddr) */
31.
32.     Listen(lfd, 20);
33.
34.     printf("Accept ...\n");
35.     while (1) {
36.         len = sizeof(cliaddr);    //AF_UNIX 大小+108B
37.
38.         cfd = Accept(lfd, (struct sockaddr *)&cliaddr, (socklen_t *)&len);
39.
40.         len -= offsetof(struct sockaddr_un, sun_path);    /* 得到文件名的长度 */
41.         cliaddr.sun_path[len] = '\0';    /* 确保打印时,没有乱码出现 */
42.
43.         printf("client bind filename %s\n", cliaddr.sun_path);
```

```

44.
45.     while ((size = read(cfd, buf, sizeof(buf))) > 0) {
46.         for (i = 0; i < size; i++)
47.             buf[i] = toupper(buf[i]);
48.         write(cfd, buf, size);
49.     }
50.     close(cfd);
51. }
52. close(lfd);
53.
54. return 0;
55. }

```

客户端代码：

```

1. #include <stdio.h>
2. #include <unistd.h>
3. #include <sys/types.h>
4. #include <sys/socket.h>
5. #include <strings.h>
6. #include <string.h>
7. #include <ctype.h>
8. #include <arpa/inet.h>
9. #include <sys/un.h>
10. #include <stddef.h>
11.
12. #include "wrap.h"
13.
14. #define SERV_ADDR "serv.socket"
15. #define CLIE_ADDR "clie.socket"
16.
17. int main(void)
18. {
19.     int cfd, len;
20.     struct sockaddr_un servaddr, cliaddr;
21.     char buf[4096];
22.
23.     cfd = Socket(AF_UNIX, SOCK_STREAM, 0);
24.
25.     bzero(&cliaddr, sizeof(cliaddr));
26.     cliaddr.sun_family = AF_UNIX;
27.     strcpy(cliaddr.sun_path, CLIE_ADDR);
28.
29.     len = offsetof(struct sockaddr_un, sun_path) + strlen(cliaddr.sun_path); /* 计算客户端地址结构有效长度 */
30.
31.     unlink(CLIE_ADDR);

```

```
32.     Bind(cfd, (struct sockaddr *)&cliaddr, len);                                /* 客户端也需要
    bind, 不能依赖自动绑定*/
33.
34.
35.     bzero(&servaddr, sizeof(servaddr));                                        /* 构造 server 地
    址 */
36.     servaddr.sun_family = AF_UNIX;
37.     strcpy(servaddr.sun_path, SERV_ADDR);
38.
39.     len = offsetof(struct sockaddr_un, sun_path) + strlen(servaddr.sun_path);  /* 计算服务器端地址
    结构有效长度 */
40.
41.     Connect(cfd, (struct sockaddr *)&servaddr, len);
42.
43.     while (fgets(buf, sizeof(buf), stdin) != NULL) {
44.         write(cfd, buf, strlen(buf));
45.         len = read(cfd, buf, sizeof(buf));
46.         write(STDOUT_FILENO, buf, len);
47.     }
48.
49.     close(cfd);
50.
51.     return 0;
52. }
```

107P-本地套接字和网络套接字实现对比

由于布局原因，直接看课程笔记比较科学。

[linux 网络编程资料\day5\1-教学资料\课堂笔记.txt](#)

108P-总结

直接看课程笔记 [linux 网络编程资料\day5\1-教学资料\课堂笔记.txt](#)

109P-复习

110P-libevent 简介

libevent 库

开源。精简。跨平台（Windows、Linux、maxos、unix）。专注于网络通信。

111P-libevent 库的下载和安装

源码包安装： 参考 README、readme

`./configure` 检查安装环境 生成 makefile

`make` 生成 .o 和 可执行文件

`sudo make install` 将必要的资源 cp 置系统指定目录。

进入 sample 目录，运行 demo 验证库安装使用情况。

编译使用库的 .c 时，需要加 `-levent` 选项。

库名 `libevent.so` --> `/usr/local/lib` 查看的到。

特性：

基于“事件”异步通信模型。--- 回调。

这里遇到一个问题：

```
zhcode@ubuntu:~/Code/libevent-2.1.8-stable/sample$ ./hello
./hello: error while loading shared libraries: libevent-2.1.so.6: cannot open shared object file: No such file or directory
```

解决办法：

[解决这个问题的博客](#)

方法2：

```
1 1.将用户用到的库统一放到一个目录，如 /usr/local/lib
2 # cp libXXX.so.X /usr/local/lib/
3
4 2.向库配置文件中，写入库文件所在目录
5 # vim /etc/ld.so.conf.d/usr-libs.conf
6 /usr/local/lib
7
8 3.更新/etc/ld.so.cache文件
9 # ldconfig
```

完事儿运行测试，结果如下：

```
Terminal File Edit View Search Terminal Help
zhcode@ubuntu:~$ cd ~/Code/libevent-2.1.8-stable/
zhcode@ubuntu:~/Code/libevent-2.1.8-stable$ cd -
/home/zhcode
zhcode@ubuntu:~$ nc 127.1 9995
Hello, World!
zhcode@ubuntu:~$

ld.so.conf.d/usr-libs.conf
/etc/ld.so.conf.d/usr-libs.conf

/ld.so.cache~: Permission denied
onfig

er.o          signal-test.c
m             signal-test.o
c             time-test
ostname_validation.c  time-test.c
ostname_validation.h  time-test.o
st

zhcode@ubuntu:~/Code/libevent-2.1.8-stable/sample$ ./hello
flushed answer

```

112P-libevent 封装的框架思想

libevent 框架:

1. 创建 event_base (乐高底座)
2. 创建 事件 evnet
3. 将事件 添加到 base 上
4. 循环监听事件满足
5. 释放 event_base

1. 创建 event_base (乐高底座)

```
struct event_base *event_base_new(void);
```

```
struct event_base *base = event_base_new();
```

2. 创建 事件 evnet

```
常规事件 event --> event_new();
```

```
bufferevent --> bufferevent_socket_new();
```

3. 将事件 添加到 base 上

```
int event_add(struct event *ev, const struct timeval *tv)
```

4. 循环监听事件满足

```
int event_base_dispatch(struct event_base *base);
```

```
event_base_dispatch(base);
```

5. 释放 event_base

```
event_base_free(base);
```

113P-结合 helloworld 初识 libevent

特性：

基于“事件”异步通信模型。--- 回调。

114P-框架相关的不常用函数

查看支持哪些多路 IO:

代码如下:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <errno.h>
6 #include <pthread.h>
7 #include <event2/event.h>
8
9 int main(int argc, char *argv[]){
10     int i;
11
12     struct event_base *base = event_base_new();
13
14     const char **buf;
15     buf = event_get_supported_methods();
16
17     for(i=0; i<10; i++){
18         printf("buf[i] = %s\n", buf[i]);
19     }
20
21     return 0;
22 }
```

编译运行, 结果如下:

```
zhcode@ubuntu:~/network/code114$ gcc event.c -o event -levent
zhcode@ubuntu:~/network/code114$ ./event
buf[i] = epoll
buf[i] = poll
buf[i] = select
buf[i] = (null)
buf[i] = (null)
Segmentation fault (core dumped)
zhcode@ubuntu:~/network/code114$
```

115P-创建事件对象

创建事件 event:

```
struct event *ev;
```

```
struct event *event_new(struct event_base *base, evutil_socket_t fd, short what,  
event_callback_fn cb; void *arg);
```

base: event_base_new() 返回值。

fd: 绑定到 event 上的 文件描述符

what: 对应的事件 (r、w、e)

EV_READ 一次 读事件

EV_WRITE 一次 写事件

EV_PERSIST 持续触发。 结合 event_base_dispatch 函数使用，生效。

cb: 一旦事件满足监听条件，回调的函数。

```
typedef void (*event_callback_fn)(evutil_socket_t fd, short, void *)
```

arg: 回调的函数的参数。

返回值: 成功创建的 event

116P-事件 event 操作

添加事件到 event_base

```
int event_add(struct event *ev, const struct timeval *tv);
```

ev: event_new() 的返回值。

tv: NULL

销毁事件

```
int event_free(struct event *ev);
```

ev: event_new() 的返回值。

117P-使用 fifo 的读写

读端的代码如下：

```
1. #include <stdio.h>
2. #include <unistd.h>
3. #include <stdlib.h>
4. #include <sys/types.h>
5. #include <sys/stat.h>
6. #include <string.h>
7. #include <fcntl.h>
8. #include <event2/event.h>
9.
10. // 对操作处理函数
11. void read_cb(evutil_socket_t fd, short what, void *arg)
12. {
13.     // 读管道
14.     char buf[1024] = {0};
15.
16.     int len = read(fd, buf, sizeof(buf));
17.
18.     printf("read event: %s \n", what & EV_READ ? "Yes" : "No");
19.     printf("data len = %d, buf = %s\n", len, buf);
20.
21.     sleep(1);
22. }
23.
24.
25. // 读管道
26. int main(int argc, const char* argv[])
27. {
28.     unlink("myfifo");
29.
30.     //创建有名管道
31.     mkfifo("myfifo", 0664);
32.
33.     // open file
34.     //int fd = open("myfifo", O_RDONLY | O_NONBLOCK);
35.     int fd = open("myfifo", O_RDONLY);
36.     if(fd == -1)
37.     {
38.         perror("open error");
39.         exit(1);
40.     }
41.
42.     // 创建个 event_base
43.     struct event_base* base = NULL;
44.     base = event_base_new();
45.
```



```

46. // 创建事件
47. struct event* ev = NULL;
48. ev = event_new(base, fd, EV_READ | EV_PERSIST, read_cb, NULL);
49.
50. // 添加事件
51. event_add(ev, NULL);
52.
53. // 事件循环
54. event_base_dispatch(base); // while (1) { epoll();}
55.
56. // 释放资源
57. event_free(ev);
58. event_base_free(base);
59. close(fd);
60.
61. return 0;
62. }

```

如代码所示，这个也遵循 libevent 搭积木的过程

写管道代码如下：

```

1. #include <stdio.h>
2. #include <unistd.h>
3. #include <stdlib.h>
4. #include <sys/types.h>
5. #include <sys/stat.h>
6. #include <string.h>
7. #include <fcntl.h>
8. #include <event2/event.h>
9.
10. // 对操作处理函数
11. void write_cb(evutil_socket_t fd, short what, void *arg)
12. {
13.     // write 管道
14.     char buf[1024] = {0};
15.
16.     static int num = 0;
17.     sprintf(buf, "hello,world-%d\n", num++);
18.     write(fd, buf, strlen(buf)+1);
19.
20.     sleep(1);
21. }
22.
23.
24. // 写管道
25. int main(int argc, const char* argv[])
26. {

```

```

27. // open file
28. //int fd = open("myfifo", O_WRONLY | O_NONBLOCK);
29. int fd = open("myfifo", O_WRONLY);
30. if(fd == -1)
31. {
32.     perror("open error");
33.     exit(1);
34. }
35.
36. // 写管道
37. struct event_base* base = NULL;
38. base = event_base_new();
39.
40. // 创建事件
41. struct event* ev = NULL;
42. // 检测的写缓冲区是否有空间写
43. //ev = event_new(base, fd, EV_WRITE , write_cb, NULL);
44. ev = event_new(base, fd, EV_WRITE | EV_PERSIST, write_cb, NULL);
45.
46. // 添加事件
47. event_add(ev, NULL);
48.
49. // 事件循环
50. event_base_dispatch(base);
51.
52. // 释放资源
53. event_free(ev);
54. event_base_free(base);
55. close(fd);
56.
57. return 0;
58. }

```

编译运行，结果如下：

```

zhcode@ubuntu:~/network/code114$ ./read
read event: Yes
data len = 15, buf = hello,world-0

read event: Yes
data len = 15, buf = hello,world-1

read event: Yes
data len = 15, buf = hello,world-2

read event: Yes
data len = 15, buf = hello,world-3

```

```

Terminal File Edit View Search Terminal Help
zhcode@ubuntu:~$ cd network/code114
zhcode@ubuntu:~/network/code114$ ./write
zhcode@ubuntu:~/network/code114$

```

118P-使用 fifo 的读写编码实现

这个基本上就是把前面代码写了一遍，复习一下，问题不大

119P-未决和非未决

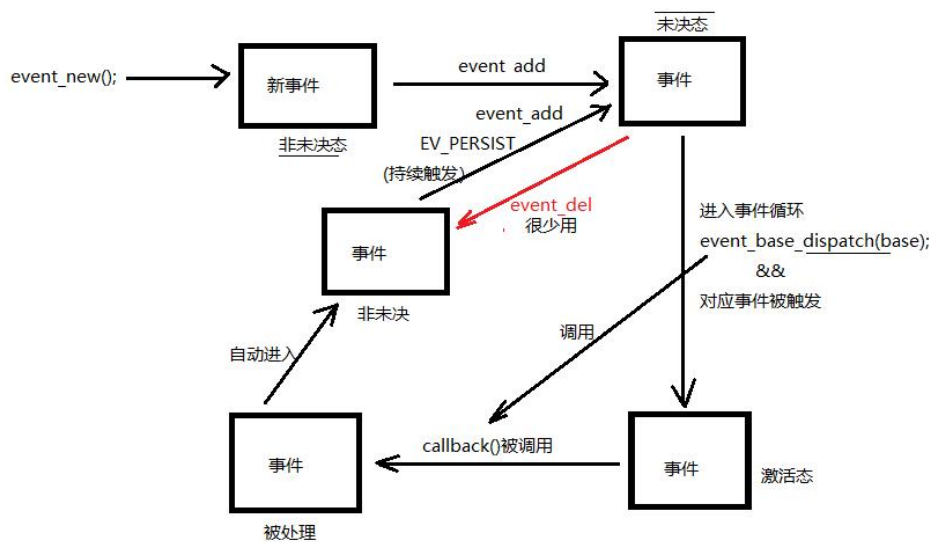
未决和非未决：

非未决：没有资格被处理

未决：有资格被处理，但尚未被处理

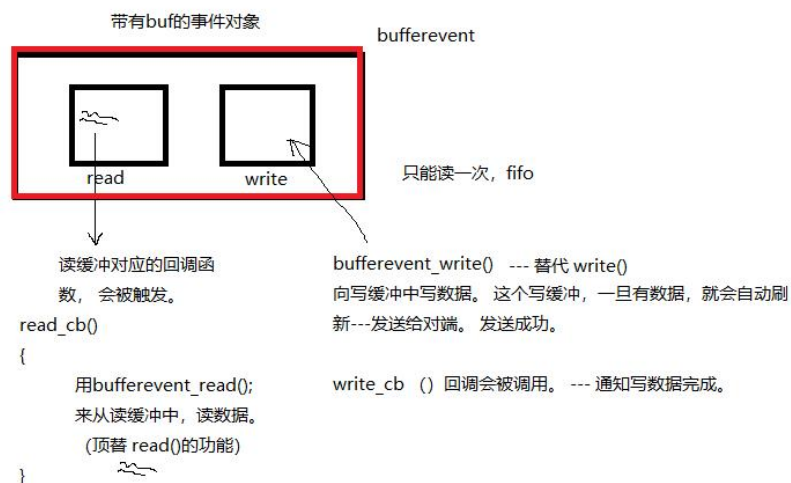
event_new --> event ---> 非未决 --> event_add --> 未决 --> dispatch() && 监听事件被触发 --> 激活态

--> 执行回调函数 --> 处理态 --> 非未决 event_add && EV_PERSIST --> 未决 --> event_del --> 非未决



120P-中午复习

121P-bufferevent 特性



带缓冲区的事件 bufferevent

```
#include <event2/bufferevent.h>
```

read/write 两个缓冲. 借助 队列.

122P-bufferevent 事件对象创建、销毁

创建、销毁 bufferevent:

```
struct bufferevent *ev;
```

```
struct bufferevent *bufferevent_socket_new(struct event_base *base, evutil_socket_t fd,  
enum bufferevent_options options);
```

base: event_base

fd: 封装到 bufferevent 内的 fd

options: BEV_OPT_CLOSE_ON_FREE

返回: 成功创建的 bufferevent 事件对象。

```
void bufferevent_socket_free(struct bufferevent *ev);
```

123P-给 bufferevent 事件对象设置回调

给 bufferevent 设置回调:

对比 event: event_new(fd, callback); event_add() -- 挂 到
event_base 上。

```
bufferevent_socket_new (fd)    bufferevent_setcb ( callback )
```

```
void bufferevent_setcb(struct bufferevent * bevf,  
    bufferevent_data_cb readcb,  
    bufferevent_data_cb writecb,  
    bufferevent_event_cb eventcb,  
    void *cbarg );
```

bufev: bufferevent_socket_new() 返回值

readcb: 设置 bufferevent 读缓冲, 对应回调 read_cb{ bufferevent_read() 读数据 }

writecb: 设置 bufferevent 写缓冲, 对应回调 write_cb { } -- 给调用者, 发送写成功通知。 可以 NULL

eventcb: 设置 事件回调。 也可传 NULL

```
typedef void (*bufferevent_event_cb)(struct bufferevent *bev,    short events, void  
*ctx);
```

```
void event_cb(struct bufferevent *bev,    short events, void *ctx)  
{  
  
    .....  
}
```

events: BEV_EVENT_CONNECTED

cbarg: 上述回调函数使用的 参数。

read 回调函数类型:

```
typedef void (*bufferevent_data_cb)(struct bufferevent *bev, void*ctx);  
  
void read_cb(struct bufferevent *bev, void *cbarg )  
{  
  
    .....  
    bufferevent_read();    --- read();  
}
```


bufferevent_read() 函数的原型:

```
size_t bufferevent_read(struct bufferevent *bev, void *buf, size_t bufsz);
```

write 回调函数类型:

```
int bufferevent_write(struct bufferevent *bufev, const void *data, size_t size);
```

124P-缓冲区开启和关闭

启动、关闭 bufferevent 的 缓冲区：

```
void bufferevent_enable(struct bufferevent *bufev, short events);    启动
```

```
events:  EV_READ、EV_WRITE、EV_READ|EV_WRITE
```

默认、write 缓冲是 enable、read 缓冲是 disable

```
bufferevent_enable(evev, EV_READ);    -- 开启读缓冲。
```

125P-客户端和服务端连接和监听

连接客户端：

```
socket();connect();
```

```
int bufferevent_socket_connect(struct bufferevent *bev, struct sockaddr *address, int addrlen);
```

bev: bufferevent 事件对象（封装了 fd）

address、len: 等同于 connect() 参 2/3

创建监听服务器：

```
----- socket();bind();listen();accept();
```

```
struct evconnlistener *listener
```

```
struct evconnlistener *evconnlistener_new_bind (  
    struct event_base *base,  
    evconnlistener_cb cb,  
    void *ptr,  
    unsigned flags,  
    int backlog,  
    const struct sockaddr *sa,  
    int socklen);
```

base: event_base

cb: 回调函数。 一旦被回调，说明在其内部应该与客户端完成， 数据读写操作，进行通信。

ptr: 回调函数的参数

flags: LEV_OPT_CLOSE_ON_FREE | LEV_OPT_REUSEABLE

backlog: listen() 2 参。 -1 表最大值

sa: 服务器自己的地址结构体

socklen: 服务器自己的地址结构体大小。

返回值: 成功创建的监听器。

释放监听服务器：

```
void evconnlistener_free(struct evconnlistener *lev);
```

126P-libevent 实现 TCP 服务器流程

服务器端 libevent 创建 TCP 连接:

1. 创建 event_base
2. 创建 bufferevent 事件对象。bufferevent_socket_new();
3. 使用 bufferevent_setcb() 函数给 bufferevent 的 read、write、event 设置回调函数。
4. 当监听的事件满足时，read_cb 会被调用，在其内部 bufferevent_read();读
5. 使用 evconnlistener_new_bind 创建监听服务器，设置其回调函数，当有客户端成功连接时，这个回调函数会被调用。
6. 封装 listner_cb() 在函数内部。完成与客户端通信。
7. 设置读缓冲、写缓冲的使能状态 enable、disable
7. 启动循环 event_base_dispatch();
8. 释放连接。

Libevent 实现 TCP 服务器流程

1. 创建 event_base
2. 创建服务器连接监听器 evconnlistener_new_bind();
3. 在 evconnlistener_new_bind 的回调函数中，处理接受连接后的操作。
4. 回调函数被调用，说明有一个新客户端连接上来。会得到一个新 fd，用于跟客户端通信（读、写）。
5. 使用 bufferevent_socket_new() 创建一个新 bufferevent 事件，将 fd 封装到这个事件对象中。
6. 使用 bufferevent_setcb 给这个事件对象的 read、write、event 设置回调。
7. 设置 bufferevent 的读写缓冲区 enable / disable
8. 接受、发送数据 bufferevent_read() / bufferevent_write()
9. 启动循环监听
10. 释放资源。

127P-libevent 实现 TCP 服务器源码分析

服务器源码如下：

```
1. #include <stdio.h>
2. #include <unistd.h>
3. #include <stdlib.h>
4. #include <sys/types.h>
5. #include <sys/stat.h>
6. #include <string.h>
7. #include <event2/event.h>
8. #include <event2/listener.h>
9. #include <event2/bufferevent.h>
10.
11. // 读缓冲区回调
12. void read_cb(struct bufferevent *bev, void *arg)
13. {
14.     char buf[1024] = {0};
15.     bufferevent_read(bev, buf, sizeof(buf));
16.     printf("client say: %s\n", buf);
17.
18.     char *p = "我是服务器，已经成功收到你发送的数据!";
19.     // 发数据给客户端
20.     bufferevent_write(bev, p, strlen(p)+1);
21.     sleep(1);
22. }
23.
24. // 写缓冲区回调
25. void write_cb(struct bufferevent *bev, void *arg)
26. {
27.     printf("I'm 服务器，成功写数据给客户端,写缓冲区回调函数被回调...\n");
28. }
29.
30. // 事件
31. void event_cb(struct bufferevent *bev, short events, void *arg)
32. {
33.     if (events & BEV_EVENT_EOF)
34.     {
35.         printf("connection closed\n");
36.     }
37.     else if(events & BEV_EVENT_ERROR)
38.     {
39.         printf("some other error\n");
40.     }
41.
42.     bufferevent_free(bev);
43.     printf("buffevent 资源已经被释放...\n");
44. }
45.
```

```
46.
47.
48. void cb_listener(
49.     struct evconnlistener *listener,
50.     evutil_socket_t fd,
51.     struct sockaddr *addr,
52.     int len, void *ptr)
53. {
54.     printf("connect new client\n");
55.
56.     struct event_base* base = (struct event_base*)ptr;
57.     // 通信操作
58.     // 添加新事件
59.     struct bufferevent *bev;
60.     bev = bufferevent_socket_new(base, fd, BEV_OPT_CLOSE_ON_FREE);
61.
62.     // 给 bufferevent 缓冲区设置回调
63.     bufferevent_setcb(bev, read_cb, write_cb, event_cb, NULL);
64.     bufferevent_enable(bev, EV_READ);
65. }
66.
67.
68. int main(int argc, const char* argv[])
69. {
70.
71.     // init server
72.     struct sockaddr_in serv;
73.
74.     memset(&serv, 0, sizeof(serv));
75.     serv.sin_family = AF_INET;
76.     serv.sin_port = htons(9876);
77.     serv.sin_addr.s_addr = htonl(INADDR_ANY);
78.
79.     struct event_base* base;
80.     base = event_base_new();
81.     // 创建套接字
82.     // 绑定
83.     // 接收连接请求
84.     struct evconnlistener* listener;
85.     listener = evconnlistener_new_bind(base, cb_listener, base,
86.                                         LEV_OPT_CLOSE_ON_FREE | LEV_OPT_REUSEABLE,
87.                                         36, (struct sockaddr*)&serv, sizeof(serv));
88.
89.     event_base_dispatch(base);
90.
91.     evconnlistener_free(listener);
92.     event_base_free(base);
93.
94.     return 0;
```


128P-服务器注意事项

bufev_server 的代码，锤起来：

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <unistd.h>
5. #include <errno.h>
6. #include <sys/socket.h>
7. #include <event2/event.h>
8. #include <event2/bufferevent.h>
9. #include <event2/listener.h>
10. #include <pthread.h>
11.
12. void sys_err(const char *str)
13. {
14.     perror(str);
15.     exit(1);
16. }
17.
18. // 读事件回调
19. void read_cb(struct bufferevent *bev, void *arg)
20. {
21.     char buf[1024] = {0};
22.
23.     // 借助读缓冲，从客户端拿数据
24.     bufferevent_read(bev, buf, sizeof(buf));
25.     printf("client write: %s\n", buf);
26.
27.     // 借助写缓冲，写数据回给客户端
28.     bufferevent_write(bev, "abcdefg", 7);
29. }
30.
31. // 写事件回调
32. void write_cb(struct bufferevent *bev, void *arg)
33. {
34.     printf("-----fwq-----has wrote\n");
35. }
36.
37. // 其他事件回调
38. void event_cb(struct bufferevent *bev, short events, void *ctx)
39. {
40.
41. }
42.
43. // 被回调，说明有客户端成功连接， cfd 已经传入该参数内部。 创建 bufferevent 事件对象
44. // 与客户端完成读写操作。
45. void listener_cb(struct evconnlistener *listener, evutil_socket_t sock,
```



```

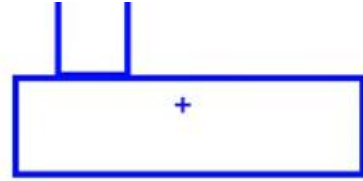
46.         struct sockaddr *addr, int len, void *ptr)
47. {
48.     struct event_base *base = (struct event_base *)ptr;
49.
50.     // 创建 bufferevent 对象
51.     struct bufferevent *bev = NULL;
52.     bev = bufferevent_socket_new(base, sock, BEV_OPT_CLOSE_ON_FREE);
53.
54.     // 给 bufferevent 对象 设置回调 read、write、event
55.     void bufferevent_setcb(struct bufferevent * befv,
56.                           bufferevent_data_cb readcb,
57.                           bufferevent_data_cb writecb,
58.                           bufferevent_event_cb eventcb,
59.                           void *cbarg );
60.
61.     // 设置回调函数
62.     bufferevent_setcb(bev, read_cb, write_cb, NULL, NULL);
63.
64.     // 启动 read 缓冲区的 使能状态
65.     bufferevent_enable(bev, EV_READ);
66.
67.     return ;
68. }
69.
70.
71. int main(int argc, char *argv[])
72. {
73.     // 定义服务器地址结构
74.     struct sockaddr_in srv_addr;
75.     bzero(&srv_addr, sizeof(srv_addr));
76.     srv_addr.sin_family = AF_INET;
77.     srv_addr.sin_port = htons(8765);
78.     srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
79.
80.     // 创建 event_base
81.     struct event_base *base = event_base_new();
82.
83.     /*
84.     struct evconnlistener *evconnlistener_new_bind (
85.         struct event_base *base,
86.         evconnlistener_cb cb,
87.         void *ptr,
88.         unsigned flags,
89.         int backlog,
90.         const struct sockaddr *sa,
91.         int socklen);
92.     */
93.
94.     // 创建服务器监听器:

```

```
95.     struct evconnlistener *listener = NULL;
96.     listener = evconnlistener_new_bind(base, listener_cb, (void *)base,
97.                                       LEV_OPT_CLOSE_ON_FREE | LEV_OPT_REUSEABLE, -1,
98.                                       (struct sockaddr *)&srv_addr, sizeof(srv_addr));
99.
100.    // 启动监听循环
101.    event_base_dispatch(base);
102.
103.    // 销毁 event_base
104.    evconnlistener_free(listener);
105.    event_base_free(base);
106.
107.    return 0;
108.}
```

129P-客户端流程简析和回顾

Libevent 实现 TCP 客户端流程



1. 创建 event_base
2. 使用 `bufferevent_socket_new()` 创建一个用跟服务器通信的 bufferevent 事件对象
3. 使用 `bufferevent_socket_connect()` 连接 服务器
4. 使用 `bufferevent_setcb()` 给 bufferevent 对象的 read、write、event 设置回调
5. 设置 bufferevent 对象的读写缓冲区 enable / disable
6. 接受、发送数据 `bufferevent_read()` / `bufferevent_write()`
7. 释放资源。

代码走起：

```
1. #include <stdio.h>
2. #include <unistd.h>
3. #include <stdlib.h>
4. #include <sys/types.h>
5. #include <sys/stat.h>
6. #include <string.h>
7. #include <event2/bufferevent.h>
8. #include <event2/event.h>
9. #include <arpa/inet.h>
10.
11. void read_cb(struct bufferevent *bev, void *arg)
12. {
13.     char buf[1024] = {0};
14.     bufferevent_read(bev, buf, sizeof(buf));
15.
16.     printf("fwq say:%s\n", buf);
17.
18.     bufferevent_write(bev, buf, strlen(buf)+1);
19.     sleep(1);
20. }
21.
22. void write_cb(struct bufferevent *bev, void *arg)
23. {
24.     printf("-----我是客户端的写回调函数,没卵用\n");
25. }
26.
27. void event_cb(struct bufferevent *bev, short events, void *arg)
28. {
29.     if (events & BEV_EVENT_EOF)
30.     {
31.         printf("connection closed\n");
```

```
32.     }
33.     else if(events & BEV_EVENT_ERROR)
34.     {
35.         printf("some other error\n");
36.     }
37.     else if(events & BEV_EVENT_CONNECTED)
38.     {
39.         printf("已经连接服务器...\(^o^)/...\n");
40.         return;
41.     }
42.
43.     // 释放资源
44.     bufferevent_free(bev);
45. }
46.
47. // 客户端与用户交互, 从终端读取数据写给服务器
48. void read_terminal(evutil_socket_t fd, short what, void *arg)
49. {
50.     // 读数据
51.     char buf[1024] = {0};
52.     int len = read(fd, buf, sizeof(buf));
53.
54.     struct bufferevent* bev = (struct bufferevent*)arg;
55.     // 发送数据
56.     bufferevent_write(bev, buf, len+1);
57. }
58.
59. int main(int argc, const char* argv[])
60. {
61.     struct event_base* base = NULL;
62.     base = event_base_new();
63.
64.     int fd = socket(AF_INET, SOCK_STREAM, 0);
65.
66.     // 通信的 fd 放到 bufferevent 中
67.     struct bufferevent* bev = NULL;
68.     bev = bufferevent_socket_new(base, fd, BEV_OPT_CLOSE_ON_FREE);
69.
70.     // init server info
71.     struct sockaddr_in serv;
72.     memset(&serv, 0, sizeof(serv));
73.     serv.sin_family = AF_INET;
74.     serv.sin_port = htons(9876);
75.     inet_pton(AF_INET, "127.0.0.1", &serv.sin_addr.s_addr);
76.
77.     // 连接服务器
78.     bufferevent_socket_connect(bev, (struct sockaddr*)&serv, sizeof(serv));
79.
80.     // 设置回调
```

```
81.     bufferevent_setcb(bev, read_cb, write_cb, event_cb, NULL);
82.
83.     // 设置读回调生效
84.     // bufferevent_enable(bev, EV_READ);
85.
86.     // 创建事件
87.     struct event* ev = event_new(base, STDIN_FILENO, EV_READ | EV_PERSIST,
88.                                   read_terminal, bev);
89.     // 添加事件
90.     event_add(ev, NULL);
91.
92.     event_base_dispatch(base);
93.
94.     event_free(ev);
95.
96.     event_base_free(base);
97.
98.     return 0;
99. }
```

130P-总结

[linux 网络编程资料\day6\1-教学资料\课堂笔记.txt](#)

131P-复习

132P-web 大练习的概述

写一个供用户访问主机文件的 web 服务器

4. 超文本标记语言 HTML

I

超文本标记语言（Hyper Text Mark-up Language HTML）是构成网页文档的主要语言。可以说明文字、图形、动画、声音、表格、链接等。在计算机中以.html、.htm 作为扩展名，可以被浏览器访问。

简介

HTML 特点

语法非常简洁、比较松散，以相应的英语单词关键字进行组合

html 标签不区分大小写

大多数标签是成对出现的，有开始，有结束.<html>...</html>

不成对出现的称之为短标签
 <hr/>

I

4. 标签的属性和值

属性="属性值"

hello, world

属性值建议加引号。(语法上 双, 单引号, 不加都可以)

4. html 组成部分

I

html 文件结构包括头(head)和主体(body)两大部分，头部描述的是浏览器，所需信息，而主题则包含所要说明的具体内容。可在 Linux 下用浏览器打开一个网页，使用 Ctrl-u 查看该网页的 html 文件形式。Ctrl-w 恢复。

<!doctype html> 声明文档类型

<html>文档的头部和主体内容 </html> 根标签

<head> 文档的头部信息 </head> 头部标记只能有一对

<title>显示在浏览器窗口的标题栏中“网页名称” </title> 位于<head> 标记之内

<body></body> 主体标记位于<html>之内,<head>标记之后

简单的错误网页形式，如：

```
<!doctype html>
<html>
  <head><title>404 Not Found</title></head>
  <body bgcolor="#cc99cc">
    <h4>404 Not Found</h4>
    File not found.
    <hr>
    <address>
      <a href="http://www.itcast.cn/">sina</a>
    </address>
  </body>
</html>
```

▪ 注释：

```
<!-- 我是一个 html 注释 -->
```

I

▪ 常用标签

学习 html，主要是标签的使用，能熟练的掌握各种常用标签的功能、特性，可以大大提高开发速度和效率。初学的同学可以从以下标签入手学习。

▪ 文本和标题标签

- 标题标签

```
<h1></h1> // 最大 只有一个，搜索引擎优化: seo
<h2></h2>
...
<h6></h6> // 最小 1-6 依次变小，自动换行
```

- 文本标签

```
<font></font>
属性：
  color: 文字颜色
  表示方式：
    英文单词: red green blue.....
    使用 16 进制的形式表示颜色: #ffffff -- (rgb)
```

使用 `rgb(255, 255, 0)` ↵
size: 文字大小 ↵
范围 1 - 7。7 最大、1 最小 ↵

- 文本格式化标签 ↵

加粗 ↵

`` 或 `` ↵

工作里尽量使用 `strong` ↵

I 倾斜 ↵

`` 或 `<i></i>` ↵

工作里尽量使用 `em` ↵

删除线 ↵

`` 或 `<s></s>` ↵

工作里尽量使用 `del` ↵

下划线 ↵

`<ins></ins>` 或 `<u></u>` ↵

工作里尽量使用 `ins` ↵

- 段落 : ↵

`<p> xxx </p>` ↵ I

特点: 上下自动生成空白行 ↵

- 块容器: ↵

`<div>This is a div element.</div>` ↵

用于没有语义含义的内容的块级容器(或网页的"划分")。 ↵

属性: 对齐方式: ↵

`align` ↵

`left` ↵

`center` ↵

`right` ↵

- 换行 `
` ↵

- 水平线 `<hr/>` ↵

属性: ↵ I

color: 3 种表示方法 ↵

size: 1-7 ↵

`<hr color="red" size="3"/>` ↵

134P-HTML 文本和标题

和上一话重复，僵硬，跳过

135P-错误页面 html

代码比较简单

```
1 <!doctype html>
2
3 <html>
4   <head>
5     <title>404 not found</title>
6   </head>
7   <body>
8     <div align=center>
9       <h1>404 not found</h1>
10    </div>
11    <hr size=3 />
12
13    <div align=center>
14      <font> nginx 12.14 </font>
15    </div>
16  </body>
17 </html>
18
```

136P-列表、图片和超链接

• 列表标签

- 无序列表

标签

```
<ul>  
  <li></li> 列表项  
  <li></li>  
</ul>
```

属性: type

实心圆圈: disc -- 默认

空心圆圈: circle

小方块: square

- 有序列表

标签

```
<ol>  
  <li></li> 列表项  
  <li></li>  
</ol>
```

属性:

type -- 序号

1 -- 默认

a

A

i -- 罗马数字(小)

I -- 罗马数字(大)

start

从序号的什么位置开始表示

- 自定义列表

标签

```
<dl>
  I  <dt></dt> 小标题
      <dd></dd> 解释标题
      <dd></dd> 解释标题
</dl>
```

- 图片标签

```

```

- 属性:

src: 图片的来源 必写属性

alt: 替换文本 图片不显示的时候显示的文字

title: 提示文本 鼠标放到图片上显示的文字

width: 图片宽度

height: 图片高度

- 注意:

I

图片没有定义宽高的时候, 图片按照百分之百比例显示

如果只更改图片的宽度或者高度, 图片等比例缩放。

- 超链接标签

```
<a href="http://jd.com" title="A dog" target="_blank"> 超链接 </a>
```

- 属性:

href: 去往的路径 (跳转的页面) 必写属性

title: 提示文本, 鼠标放到链接上显示的文字

target 取值:

_self: 默认值 在自身页面打开 (关闭自身页面, 打开链接页面)

_blank: 打开新页面 (自身页面不关闭, 打开一个新的链接页面)

示例:

```
<a href="http://www.baidu.com">百度一下</a>
```

- 锚链接

事先定义一个锚点标签: <p id="top">

超链接到锚点: 回到顶点

137P-http 协议请求、应答协议基础格式

• HTTP 协议基础

HTTP, 超文本传输协议 (HyperText Transfer Protocol)。互联网应用最为广泛的一种网络应用层协议。它可以减少网络传输, 使浏览器更加高效。

通常 HTTP 消息包括客户机向服务器的请求消息和服务器向客户机的响应消息。

1. 请求消息(Request)

浏览器 → 发给 → 服务器。主旨内容包含 4 部分:

- 请求行: 说明请求类型, 要访问的资源, 以及使用的 http 版本
- 请求头: 说明服务器要使用的附加信息
- 空 行: 必须!, 即使没有请求数据
- 请求数据: 也叫主体, 可以添加任意的其他数据

以下是浏览器发送给服务器的 http 协议头内容举例, 注意: 9 行的空行(\r\n)也是协议头的一部分:

```
1. GET /hello.c HTTP/1.1
2. Host: localhost:2222
3. User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:24.0) Gecko/20100101 Firefox/24.0
4. Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5. Accept-Language: zh-cn,zh;q=0.8,en-us;q=0.5,en;q=0.3
6. Accept-Encoding: gzip, deflate
7. Connection: keep-alive
8. If-Modified-Since: Fri, 18 Jul 2014 08:36:36 GMT
9.
```

• 响应消息(Response)

服务器 → 发给 → 浏览器。主旨内容包含 4 部分:

- 状态行: 包括 http 协议版本号, 状态码, 状态信息
- 消息报头: 说明客户端要使用的一些附加信息
- 空 行: 必须!
- 响应正文: 服务器返回给客户端的文本信息

以下是经服务器按照 http 协议，写回给浏览器的内容举例，1~9 行是**协议头**部分。注意：9 行\r\n的空行不可忽略。

```
1. HTTP/1.1 200 Ok
2. Server: xhttpd
3. Date: Fri, 18 Jul 2014 14:34:26 GMT
4. Content-Type: text/plain; charset=iso-8859-1 (必选项)
5. Content-Length: 32 (要么不写 或者 传-1, 要写务必精确！)
6. Content-Language: zh-CN
7. Last-Modified: Fri, 18 Jul 2014 08:36:36 GMT
8. Connection: close
9.
10. #include <stdio.h>
11.
12. int main(void)
13. {
14.     printf("Welcome to itcast ... \n");
15.
16.     return 0;
17. }
```

• HTTP 常用状态码

状态代码有三位数字组成，第一个数字定义了响应的类别，共分五种类别：

- 1xx: 指示信息--表示请求已接收，继续处理
- 2xx: 成功--表示请求已被成功接收、理解、接受
- 3xx: 重定向--要完成请求必须进行更进一步的操作
- 4xx: 客户端错误--请求有语法错误或请求无法实现
- 5xx: 服务器端错误--服务器未能实现合法的请求

● 常见状态码：

200 OK	客户端请求成功
400 Bad Request	客户端请求有语法错误，不能被服务器所理解
401 Unauthorized	请求未经授权，
403 Forbidden	服务器收到请求，但是拒绝提供服务
404 Not Found	请求资源不存在， <u>eg</u> : 输入了错误的 URL
500 Internal Server Error	服务器发生不可预期的错误
503 Server Unavailable	服务器当前不能处理客户端的请求。

138P-服务器框架复习和 getline 函数

代码如下：

```
1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>
4. #include <netinet/in.h>
5. #include <arpa/inet.h>
6. #include <sys/wait.h>
7. #include <sys/types.h>
8. #include <sys/epoll.h>
9. #include <unistd.h>
10. #include <fcntl.h>
11.
12. #define MAXSIZE 2048
13.
14.
15.
16. int init_listen_fd(int port, int epfd)
17. {
18.     // 创建监听的套接字 lfd
19.     int lfd = socket(AF_INET, SOCK_STREAM, 0);
20.     if (lfd == -1) {
21.         perror("socket error");
22.         exit(1);
23.     }
24.     // 创建服务器地址结构 IP+port
25.     struct sockaddr_in srv_addr;
26.
27.     bzero(&srv_addr, sizeof(srv_addr));
28.     srv_addr.sin_family = AF_INET;
29.     srv_addr.sin_port = htons(port);
30.     srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
31.
32.     // 端口复用
33.     int opt = 1;
34.     setsockopt(lfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
35.
36.     // 给 lfd 绑定地址结构
37.     int ret = bind(lfd, (struct sockaddr*)&srv_addr, sizeof(srv_addr));
38.     if (ret == -1) {
39.         perror("bind error");
40.         exit(1);
41.     }
42.     // 设置监听上限
43.     ret = listen(lfd, 128);
44.     if (ret == -1) {
45.         perror("listen error");
```

```
46.     exit(1);
47. }
48.
49. // lfd 添加到 epoll 树上
50. struct epoll_event ev;
51. ev.events = EPOLLIN;
52. ev.data.fd = lfd;
53.
54. ret = epoll_ctl(epfd, EPOLL_CTL_ADD, lfd, &ev);
55. if (ret == -1) {
56.     perror("epoll_ctl add lfd error");
57.     exit(1);
58. }
59.
60. return lfd;
61. }
62.
63. void do_accept(int lfd, int epfd)
64. {
65.     struct sockaddr_in clt_addr;
66.     socklen_t clt_addr_len = sizeof(clt_addr);
67.
68.     int cfd = accept(lfd, (struct sockaddr*)&clt_addr, &clt_addr_len);
69.     if (cfd == -1) {
70.         perror("accept error");
71.         exit(1);
72.     }
73.
74.     // 打印客户端 IP+port
75.     char client_ip[64] = {0};
76.     printf("New Client IP: %s, Port: %d, cfd = %d\n",
77.           inet_ntop(AF_INET, &clt_addr.sin_addr.s_addr, client_ip, sizeof(client_ip)),
78.           ntohs(clt_addr.sin_port), cfd);
79.
80.     // 设置 cfd 非阻塞
81.     int flag = fcntl(cfd, F_GETFL);
82.     flag |= O_NONBLOCK;
83.     fcntl(cfd, F_SETFL, flag);
84.
85.     // 将新节点 cfd 挂到 epoll 监听树上
86.     struct epoll_event ev;
87.     ev.data.fd = cfd;
88.
89.     // 边沿非阻塞模式
90.     ev.events = EPOLLIN | EPOLLET;
91.
92.     int ret = epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &ev);
93.     if (ret == -1) {
94.         perror("epoll_ctl add cfd error");
```

```
95.         exit(1);
96.     }
97. }
98.
99. void do_read(int cfd, int epfd)
100. {
101.     // read cfd 小 -- 大 write 回
102.     // 读取一行 http 协议, 拆分, 获取 get 文件名 协议号
103. }
104.
105. void epoll_run(int port)
106. {
107.     int i = 0;
108.     struct epoll_event all_events[MAXSIZE];
109.
110.     // 创建一个 epoll 监听树根
111.     int epfd = epoll_create(MAXSIZE);
112.     if (epfd == -1) {
113.         perror("epoll_create error");
114.         exit(1);
115.     }
116.
117.     // 创建 lfd, 并添加至监听树
118.     int lfd = init_listen_fd(port, epfd);
119.
120.     while (1) {
121.         // 监听节点对应事件
122.         int ret = epoll_wait(epfd, all_events, MAXSIZE, -1);
123.         if (ret == -1) {
124.             perror("epoll_wait error");
125.             exit(1);
126.         }
127.
128.         for (i=0; i<ret; ++i) {
129.
130.             // 只处理读事件, 其他事件默认不处理
131.             struct epoll_event *pev = &all_events[i];
132.
133.             // 不是读事件
134.             if (!(pev->events & EPOLLIN)) {
135.                 continue;
136.             }
137.             if (pev->data.fd == lfd) {           // 接受连接请求
138.
139.                 do_accept(lfd, epfd);
140.
141.             } else {                             // 读数据
142.
143.                 do_read(pev->data.fd, epfd);
```

```
144.     }
145.     }
146. }
147.}
148.
149.
150.int main(int argc, char *argv[])
151.{
152.    // 命令行参数获取 端口 和 server 提供的目录
153.    if (argc < 3)
154.    {
155.        printf("./server port path\n");
156.    }
157.
158.    // 获取用户输入的端口
159.    int port = atoi(argv[1]);
160.
161.    // 改变进程工作目录
162.    int ret = chdir(argv[2]);
163.    if (ret != 0) {
164.        perror("chdir error");
165.        exit(1);
166.    }
167.
168.    // 启动 epoll 监听
169.    epoll_run(port);
170.
171.    return 0;
172.}
```

139P-复习

请求协议： --- 浏览器组织，发送

```
GET /hello.c Http1.1\r\n
2. Host: localhost:2222\r\n
3. User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:24.0) Gecko/201001
Firefox/24.0\r\n
4. Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
5. Accept-Language: zh-cn,zh;q=0.8,en-us;q=0.5,en;q=0.3\r\n
6. Accept-Encoding: gzip, deflate\r\n
7. Connection: keep-alive\r\n
8. If-Modified-Since: Fri, 18 Jul 2014 08:36:36 GMT\r\n
【空行】\r\n
```

01

应答协议：

```
Http1.1 200 OK
2. Server: xhttpd
Content-Type: text/plain; charset=iso-8859-1
3. Date: Fri, 18 Jul 2014 14:34:26 GMT
5. Content-Length: 32 （ 要么不写 或者 传-1， 要写务必精确 ! ）
6. Content-Language: zh-CN
7. Last-Modified: Fri, 18 Jul 2014 08:36:36 GMT
8. Connection: close
\r\n
[数据起始.....
.....
。。。数据终止]
```

140P-单文件通信流程分析

1. `getline()` 获取 http 协议的第一行。
2. 从首行中拆分 GET、文件名、协议版本。 获取用户请求的文件名。
3. 判断文件是否存在。 `stat()`
4. 判断是文件还是目录。
5. 是文件-- `open` -- `read` -- 写回给浏览器
6. 先写 http 应答协议头 : `http/1.1 200 ok`

`Content-Type: text/plain; charset=iso-8859-1`

141P-处理出错返回

142P-正则表达式获取文件名

```
1. void do_read(int cfd, int epfd)
2. {
3.     // 读取一行 http 协议, 拆分, 获取 get 文件名 协议号
4.     char line[1024] = {0};
5.     char method[16], path[256], protocol[16];
6.
7.     int len = get_line(cfd, line, sizeof(line)); //读 http 请求协议首行 GET /hello.c HTTP/1.1
8.     if (len == 0) {
9.         printf("服务器, 检测到客户端关闭...\n");
10.        disconnect(cfd, epfd);
11.    } else {
12.
13.        sscanf(line, "%[^ ] %[^ ] %[^ ]", method, path, protocol);
14.        printf("method=%s, path=%s, protocol=%s\n", method, path, protocol);
15.
16.        while (1) {
17.            char buf[1024] = {0};
18.            len = get_line(cfd, buf, sizeof(buf));
19.            if (buf[0] == '\n') {
20.                break;
21.            } else if (len == -1)
22.                break;
23.        }
24.
25.    }
26.
27.    if (strncasecmp(method, "GET", 3) == 0)
28.    {
29.        char *file = path+1;    // 取出 客户端要访问的文件名
30.
31.        http_request(cfd, file);
32.
33.        disconnect(cfd, epfd);
34.    }
35. }
```

正则表达式:

sscanf 函数

函数描述:

读取格式化的字符串中的数据。

函数原型:

I

```
int sscanf(const char *buffer, const char *format, [ argument ] ... );
```

1. 取到 指定字符为止的字符串。如在下例中, 取遇到空格为止字符串。

```
sscanf("123456 abcdedf", "%[^ ]", buf);
```

```
printf("%s\n", buf);
```

结果为: 123456

2. 取 仅包含指定字符集的字符串。如在下例中, 取仅包含 1 到 9 和小写字母的字符串。

```
sscanf("123456abcdedfBCDEF", "%[1-9a-z]", buf);
```

```
printf("%s\n", buf);
```

结果为: 123456abcdedf

3. 取到 指定字符集为止的字符串。如在下例中, 取遇到大写字母为止的字符串。

```
sscanf("123456abcdedfBCDEF", "%[^A-Z]", buf);
```

```
printf("%s\n", buf);
```

结果为: 123456abcdedf

143P-判断文件是否存在

```
1. // 处理 http 请求， 判断文件是否存在， 回发
2. void http_request(int cfd, const char *file)
3. {
4.     struct stat sbuf;
5.
6.     // 判断文件是否存在
7.     int ret = stat(file, &sbuf);
8.     if (ret != 0) {
9.         // 回发浏览器 404 错误页面
10.        perror("stat");
11.        exit(1);
12.    }
13.
14.    if(S_ISREG(sbuf.st_mode)) { // 是一个普通文件
15.
16.        // 回发 http 协议应答
17.        //send_respond(cfd, 200, "OK", " Content-Type: text/plain; charset=iso-8859-1", sbuf.st_si
ze);
18.        send_respond(cfd, 200, "OK", "Content-Type:image/jpeg", -1);
19.        //send_respond(cfd, 200, "OK", "audio/mpeg", -1);
20.
21.        // 回发 给客户端请求数据内容。
22.        send_file(cfd, file);
23.    }
24. }
```

144P-写出 http 应答协议头

```
1. // 客户端端的 fd, 错误号, 错误描述, 回发文件类型, 文件长度
2. void send_respond(int cfd, int no, char *disp, char *type, int len)
3. {
4.     char buf[4096] = {0};
5.
6.     sprintf(buf, "HTTP/1.1 %d %s\r\n", no, disp);
7.     send(cfd, buf, strlen(buf), 0);
8.
9.     sprintf(buf, "Content-Type: %s\r\n", type);
10.    sprintf(buf+strlen(buf), "Content-Length:%d\r\n", len);
11.    send(cfd, buf, strlen(buf), 0);
12.
13.    send(cfd, "\r\n", 2, 0);
14. }
```

145P-写数据给浏览器

```
1. // 发送服务器本地文件 给浏览器
2. void send_file(int cfd, const char *file)
3. {
4.     int n = 0, ret;
5.     char buf[4096] = {0};
6.
7.     // 打开的服务器本地文件。 --- cfd 能访问客户端的 socket
8.     int fd = open(file, O_RDONLY);
9.     if (fd == -1) {
10.        // 404 错误页面
11.        perror("open error");
12.        exit(1);
13.    }
14.
15.    while ((n = read(fd, buf, sizeof(buf))) > 0) {
16.        ret = send(cfd, buf, n, 0);
17.        if (ret == -1) {
18.            perror("send error");
19.            exit(1);
20.        }
21.        if (ret < 4096)
22.            printf("-----send ret: %d\n", ret);
23.    }
24.
25.    close(fd);
26. }
```

146P-文件类型区分

147P-错误原因及说明

MP3 请求错误的原因在于，做错误判断时太粗略，`errno=EAGAIN` 或者 `errno=EINTR` 时，并不算错误，此时继续执行循环读取数据就行。

然而原来的程序是直接退出了，所以没接收到数据。

148P-错误页面展示

开发注意事项

浏览器请求 ico

准备一个 favicon.ico 文件放置到 服务器提供访问的资源目录中。



浏览器在请求图片的同时，会请求一个 ico 图标，用于浏览器<title>标签文字部分前端的小图标显示。

这个 ico 的文件名固定——favicon.ico。因此，自行准备一个 ico 文件，放置于服务器提供给浏览器访问的目标目录即可。

错误页面部分的代码：

```
1. void send_error(int cfd, int status, char *title, char *text)
2. {
3.     char buf[4096] = {0};
4.
5.     sprintf(buf, "%s %d %s\r\n", "HTTP/1.1", status, title);
6.     sprintf(buf+strlen(buf), "Content-Type:%s\r\n", "text/html");
7.     sprintf(buf+strlen(buf), "Content-Length:%d\r\n", -1);
8.     sprintf(buf+strlen(buf), "Connection: close\r\n");
9.     send(cfd, buf, strlen(buf), 0);
10.    send(cfd, "\r\n", 2, 0);
11.
12.    memset(buf, 0, sizeof(buf));
13.
14.    sprintf(buf, "<html><head><title>%d %s</title></head>\n", status, title);
15.    sprintf(buf+strlen(buf), "<body bgcolor=\"#cc99cc\"><h2 align=\"center\">%d %s</h2>\n", status,
        title);
16.    sprintf(buf+strlen(buf), "%s\n", text);
17.    sprintf(buf+strlen(buf), "<hr>\n</body>\n</html>\n");
18.    send(cfd, buf, strlen(buf), 0);
19.
20.    return ;
21. }
```

直接看完整代码吧：

[epoll_server.c](#)

149P-关于浏览器请求 ico 文件

150P-浏览器请求目录

```
1. // http 请求处理
2. void http_request(const char* request, int cfd)
3. {
4.     // 拆分 http 请求行
5.     char method[12], path[1024], protocol[12];
6.     sscanf(request, "%[^ ] %[^ ] %[^ ]", method, path, protocol);
7.     printf("method = %s, path = %s, protocol = %s\n", method, path, protocol);
8.
9.     // 转码 将不能识别的中文乱码 -> 中文
10.    // 解码 %23 %34 %5f
11.    decode_str(path, path);
12.
13.    char* file = path+1; // 去掉 path 中的/ 获取访问文件名
14.
15.    // 如果没有指定访问的资源, 默认显示资源目录中的内容
16.    if(strcmp(path, "/") == 0) {
17.        // file 的值, 资源目录的当前位置
18.        file = "./";
19.    }
20.
21.    // 获取文件属性
22.    struct stat st;
23.    int ret = stat(file, &st);
24.    if(ret == -1) {
25.        send_error(cfd, 404, "Not Found", "NO such file or direntry");
26.        return;
27.    }
28.
29.    // 判断是目录还是文件
30.    if(S_ISDIR(st.st_mode)) { // 目录
31.        // 发送头信息
32.        send_respond_head(cfd, 200, "OK", get_file_type(".html"), -1);
33.        // 发送目录信息
34.        send_dir(cfd, file);
35.    } else if(S_ISREG(st.st_mode)) { // 文件
36.        // 发送消息报头
37.        send_respond_head(cfd, 200, "OK", get_file_type(file), st.st_size);
38.        // 发送文件内容
39.        send_file(cfd, file);
40.    }
41. }
```

■ 快捷遍历目录 `scandir()`

服务器端，可以使用文件操作时“递归遍历目录”的源码，实现遍历目录内文件名，回显给浏览器。另外标准 C 库中，提供了 `scandir` 函数，可以便捷的实现该功能。函数原型如下：

```
int scandir(const char *dirp, struct dirent ***namelist,  
            int (*filter)(const struct dirent *),  
            int (*compar)(const struct dirent **, const struct dirent **));
```

`scandir`(待操作的目录, &子目录项列表数组, 过滤器(通常 NULL), alphasort) ;

`dirp`: 待访问的目录名称

struct dirent 类型 : (参考 `readdir()`函数)

```
struct dirent {  
    ino_t      d_ino;          /* inode number */  
    off_t      d_off;          /* not an offset; see NOTES */  
    unsigned short d_reclen;    /* length of this record */  
    unsigned char d_type;       /*  
    char        d_name[256];    /* filename */  
};
```

`compar`:参数取如下函数即可:

`int alphasort(const void *a, const void *b)`; 默认是由的排序算法。

调用 : `struct dirent** namelist;`

```
int num = scandir(dirname, &namelist, NULL, alphasort);
```

```
for (i = 0; i < num; ++i)
```

```
    char* name = namelist[i]->d_name;
```

151P-判断文件类型

```
1. // 通过文件名获取文件的类型
2. const char *get_file_type(const char *name)
3. {
4.     char* dot;
5.
6.     // 自右向左查找'.'字符, 如不存在返回 NULL
7.     dot = strrchr(name, '.');
8.     if (dot == NULL)
9.         return "text/plain; charset=utf-8";
10.    if (strcmp(dot, ".html") == 0 || strcmp(dot, ".htm") == 0)
11.        return "text/html; charset=utf-8";
12.    if (strcmp(dot, ".jpg") == 0 || strcmp(dot, ".jpeg") == 0)
13.        return "image/jpeg";
14.    if (strcmp(dot, ".gif") == 0)
15.        return "image/gif";
16.    if (strcmp(dot, ".png") == 0)
17.        return "image/png";
18.    if (strcmp(dot, ".css") == 0)
19.        return "text/css";
20.    if (strcmp(dot, ".au") == 0)
21.        return "audio/basic";
22.    if (strcmp(dot, ".wav") == 0)
23.        return "audio/wav";
24.    if (strcmp(dot, ".avi") == 0)
25.        return "video/x-msvideo";
26.    if (strcmp(dot, ".mov") == 0 || strcmp(dot, ".qt") == 0)
27.        return "video/quicktime";
28.    if (strcmp(dot, ".mpeg") == 0 || strcmp(dot, ".mpe") == 0)
29.        return "video/mpeg";
30.    if (strcmp(dot, ".vrm1") == 0 || strcmp(dot, ".wrl") == 0)
31.        return "model/vrm1";
32.    if (strcmp(dot, ".midi") == 0 || strcmp(dot, ".mid") == 0)
33.        return "audio/midi";
34.    if (strcmp(dot, ".mp3") == 0)
35.        return "audio/mpeg";
36.    if (strcmp(dot, ".ogg") == 0)
37.        return "application/ogg";
38.    if (strcmp(dot, ".pac") == 0)
39.        return "application/x-ns-proxy-autoconfig";
40.
41.    return "text/plain; charset=utf-8";
42. }
```

▪ 常见网络文件类型：

普通文件: text/plain; charset=iso-8859-1

*.html : text/html; charset=iso-8859-1

*.jpg: image/jpeg I

*.gif : image/gif

*.png: image/png

*.wav: audio/wav

*.avi: video/x-msvideo

*.mov : video/quicktime

*.mp3 : audio/mpeg

charset=iso-8859-1 西欧的编码，说明网站采用英文编码

charset=gb2312 说明网站采用的编码是简体中文

charset=utf-8 I 代表世界通用的语言编码：可以用到中文、韩文、日文等世界上所有语言编码

charset=euc-kr 说明网站采用的编码是韩文：

charset=big5 说明网站采用的编码是繁体中文：

152P-汉字字符编码和解码

URL 中的汉字默认是存为 Unicode 码

■ 含有汉字的文件

访问带有汉字的文件时，将这个 URL 复制到新的 浏览器地址栏中，可以看到它所对应的在浏览器中使用的字符编码。

如：汉字 “大学”对应的编码为：`%E5%A4%A7%E5%AD%A6`

在命令行使用 `unicode` 命令（如不存在，使用 `apt-get` 命令安装）可查看汉字对应的 Unicode 编码。如：`unicode 大学 --> UTF-8: e5 a4 a7 , UTF-8: e5 ad a6`

因此，在访问带有汉字的文件时，应该在服务器回发数据给浏览器时进行“编码”操作，在浏览器请求资源目录的汉字文件时进行“解码”操作。

153P-libevent 实现的 web 服务器

直接源码啃起来吧

154P-telnet 调试

■ 借助 telnet 调试

可使用 telnet 命令，借助 IP 和 port，模拟浏览器行为，在终端中对访问的服务器进行调试，方便查看服务器回发给浏览器的 http 协议数据。使用方式如：

命令行键入：telnet 127.0.0.1 9999 回车，手动写入 http 请求协议头，如：

```
GET /hello.c http/1.1 回车
```

此时，终端中可查看到服务器回发给浏览器的 http 应答协议及数据内容。可根据该信息进行调试。