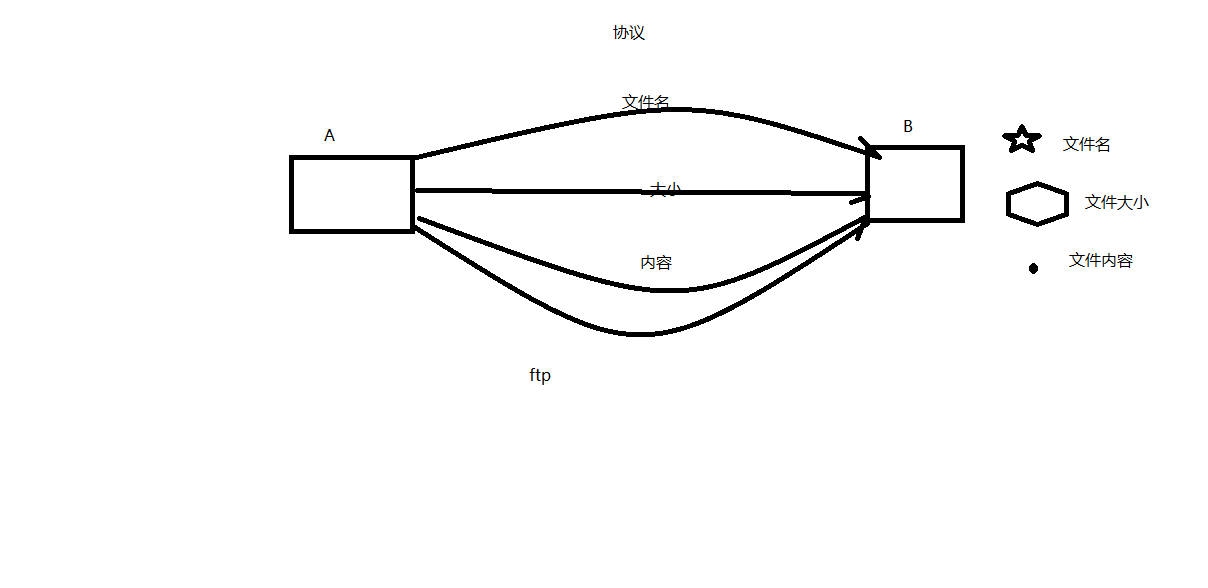
## 01P-复习-Linux网络编程

## 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

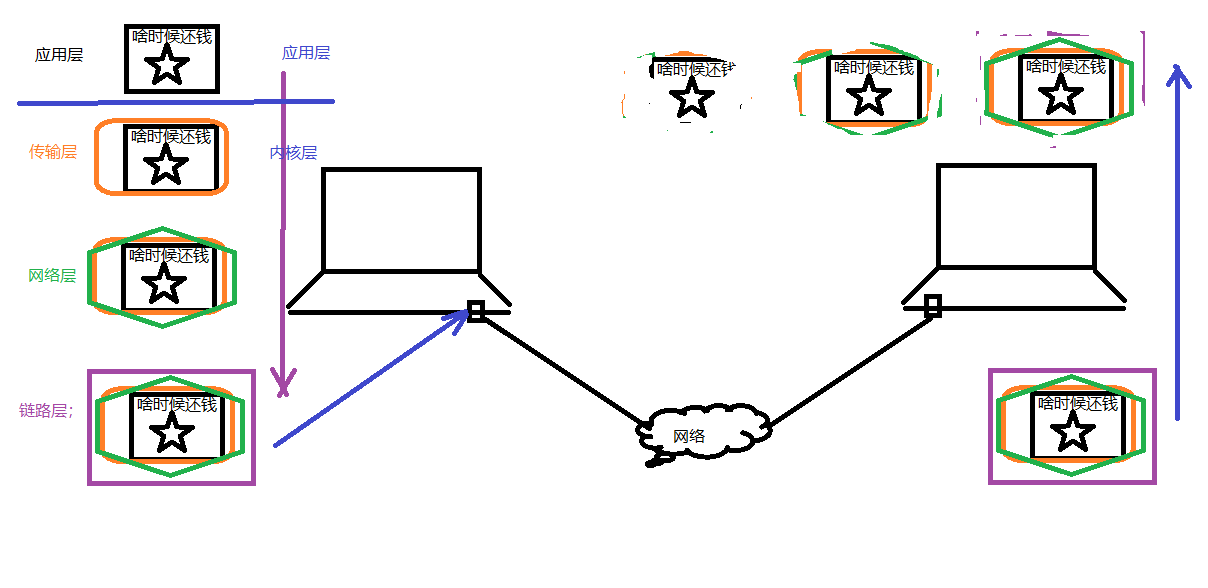
应用层：http、ftp、nfs、ssh、telnet。。。

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

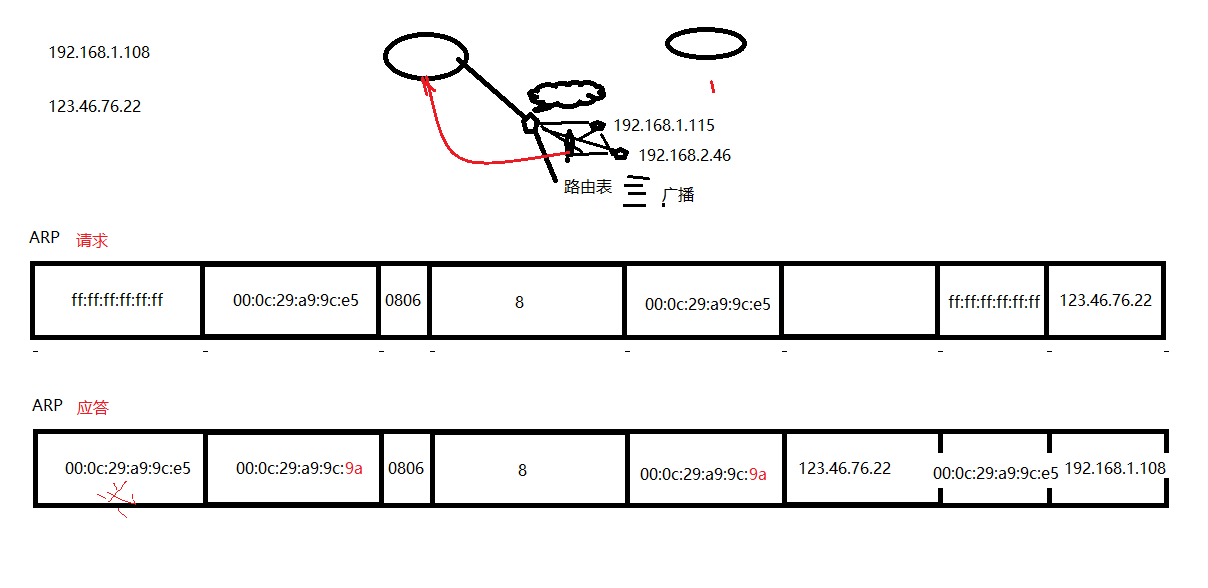


网络传输流程：

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

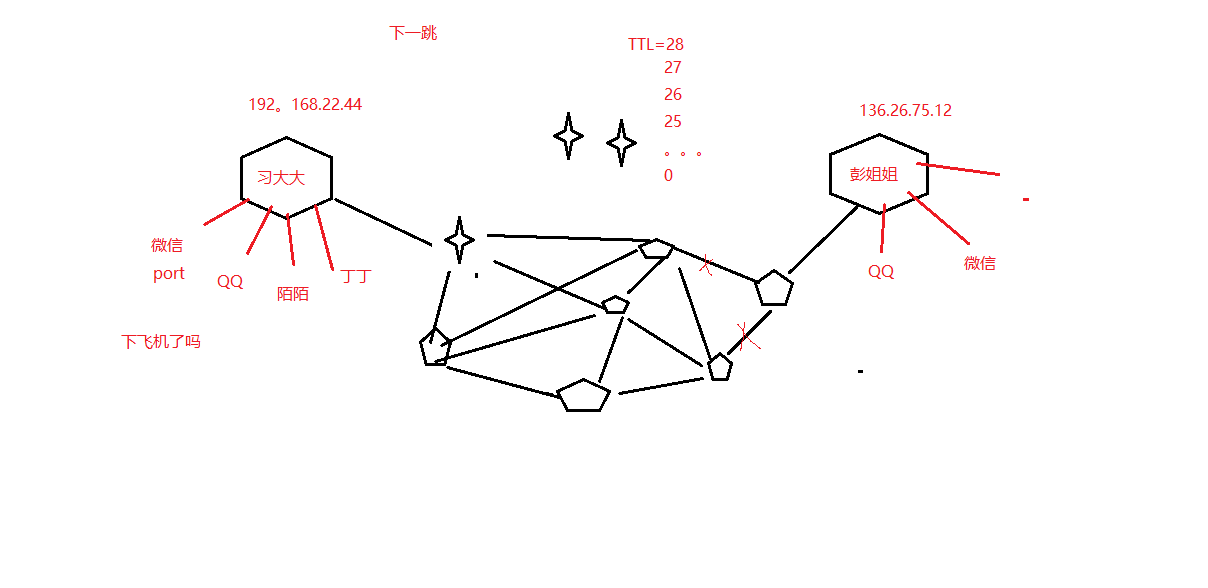
IP协议：

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

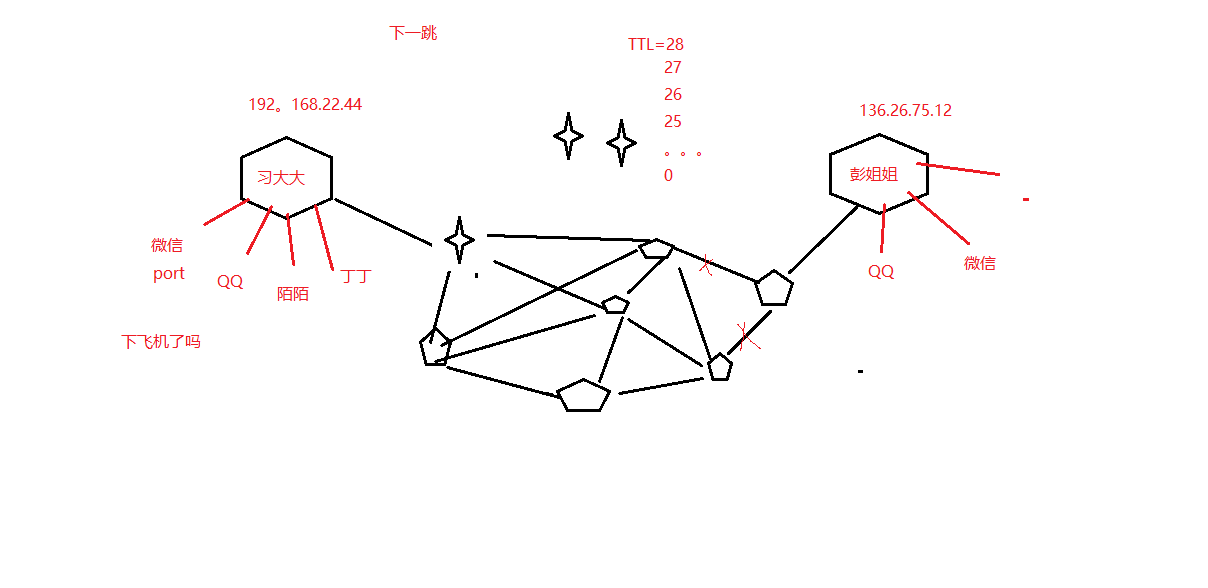


## 08P-端口号和UDP协议

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

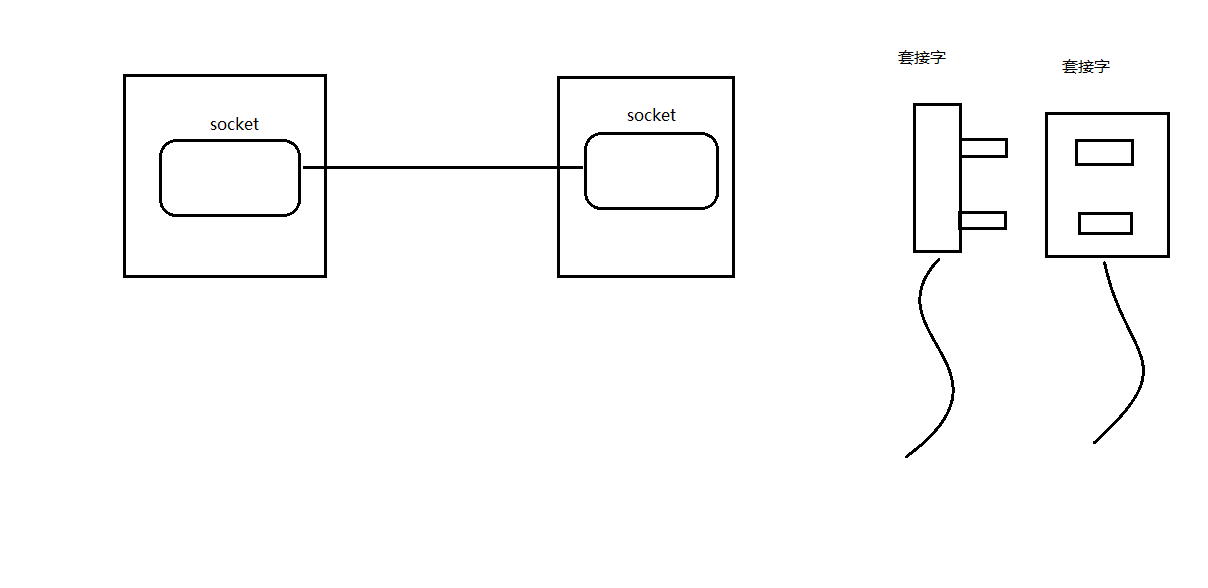
缺点： 安全性、跨平台、开发工作量较大 不能缓存大量数据、严格遵守 http

## 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

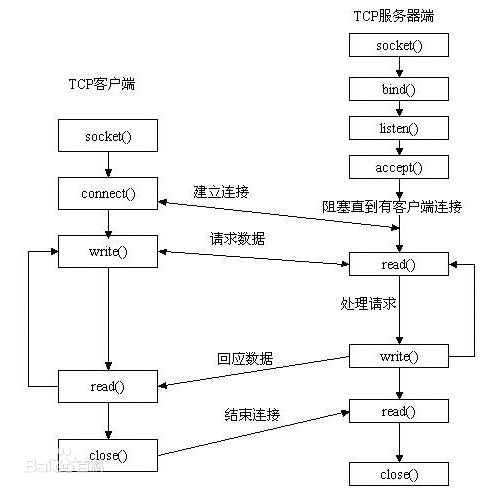
dst：本地字节序（string IP）

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;

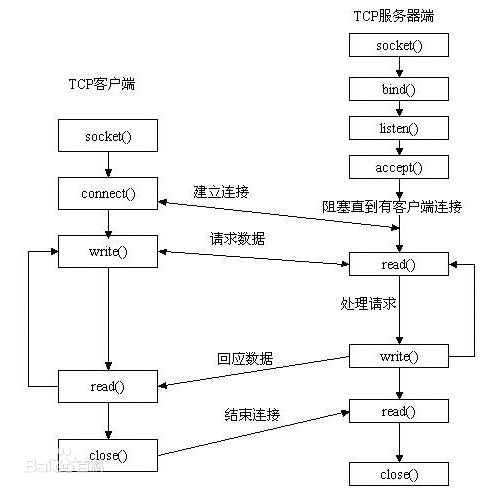
inet\_pton(AF\_INET, "192.157.22.45", (void \*)&dst);

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\_adrr.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()

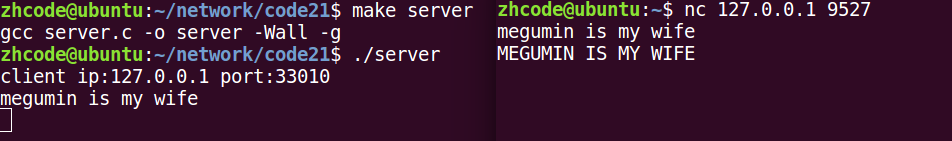
## 21P-server的实现

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>
11. #define SERV\_PORT 9527

14. **void** sys\_err(**const** **char** \*str)
15. {
16. perror(str);
17. exit(1);
18. }
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];
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
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
33. lfd = socket(AF\_INET, SOCK\_STREAM, 0);      //创建一个 socket
34. **if** (lfd == -1) {
35. sys\_err("socket error");
36. }
38. bind(lfd, (**struct** sockaddr \*)&serv\_addr, **sizeof**(serv\_addr));//给服务器socket绑定地址结构（IP+port)
40. listen(lfd, 128);                   //  设置监听上限
42. clit\_addr\_len = **sizeof**(clit\_addr);  //  获取客户端地址结构大小
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");
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
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
56. **for** (i = 0; i < ret; i++)                // 小写 -- 大写
57. buf[i] = toupper(buf[i]);
59. write(cfd, buf, ret);                   // 将大写，写回给客户端。
60. }
62. close(lfd);
63. close(cfd);
65. **return** 0;
66. }

编译测试，结果如下：



## 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>
10. #define SERV\_PORT 9527
12. **void** sys\_err(**const** **char** \*str)
13. {
14. perror(str);
15. exit(1);
16. }
18. **int** main(**int** argc, **char** \*argv[])
19. {
20. **int** cfd;
21. **int** conter = 10;
22. **char** buf[BUFSIZ];
24. **struct** sockaddr\_in serv\_addr;          //服务器地址结构
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);
31. cfd = socket(AF\_INET, SOCK\_STREAM, 0);
32. **if** (cfd == -1)
33. sys\_err("socket error");
35. **int** ret = connect(cfd, (**struct** sockaddr \*)&serv\_addr, **sizeof**(serv\_addr));
36. **if** (ret != 0)
37. sys\_err("connect err");
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. }
46. close(cfd);
48. **return** 0;
49. }

编译运行，结果如下：



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。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）

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

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\_adrr.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--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

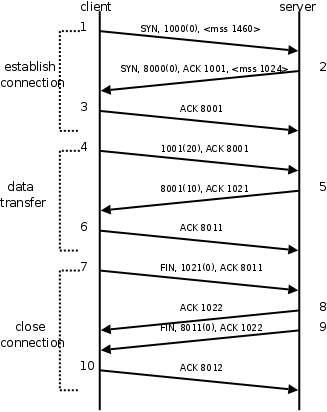
4. read() 读转换后的数据。

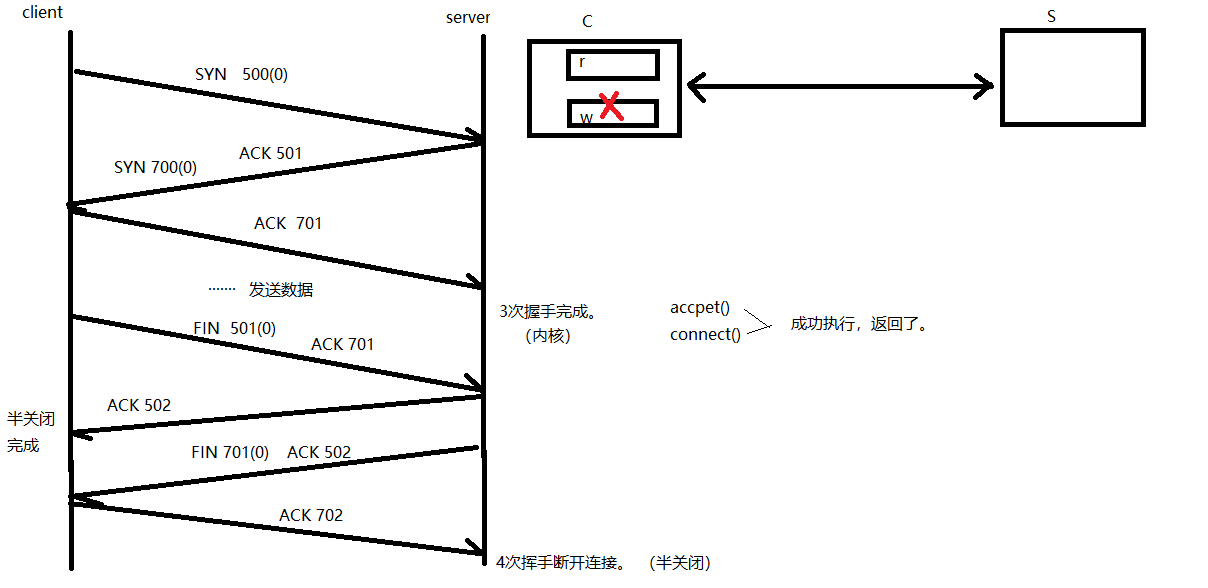
5. 显示读取结果

6. close()

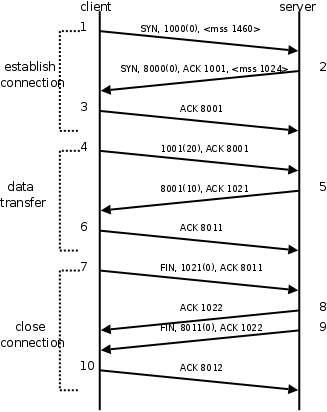
## 25P-复习

## 26P-三次握手建立连接



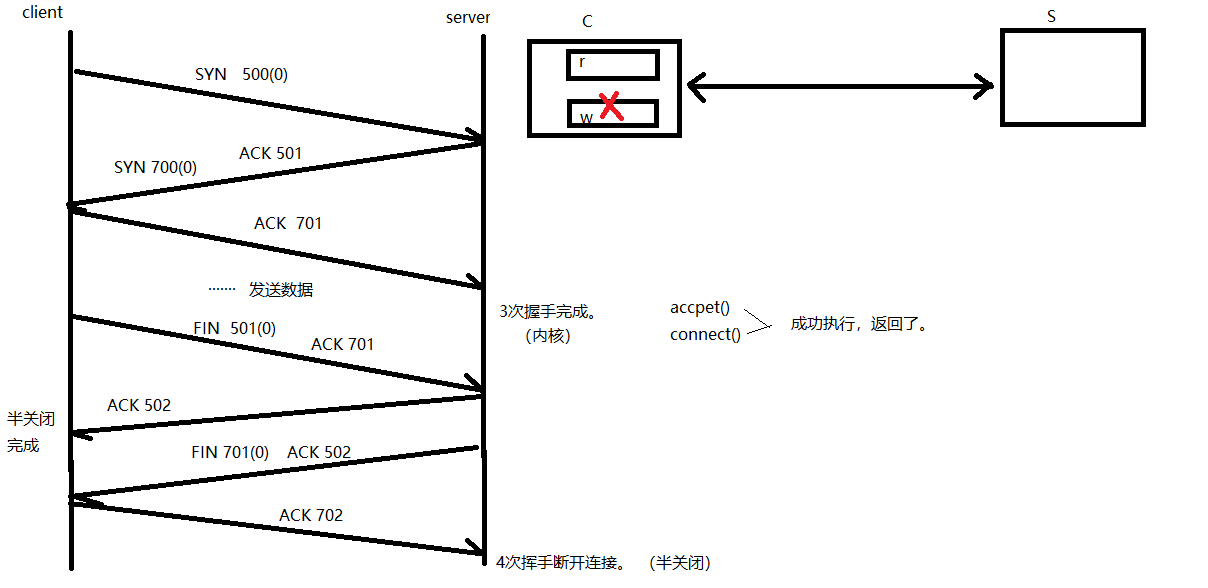


## 27P-数据通信



并不是一次发送，一次应答。也可以批量应答

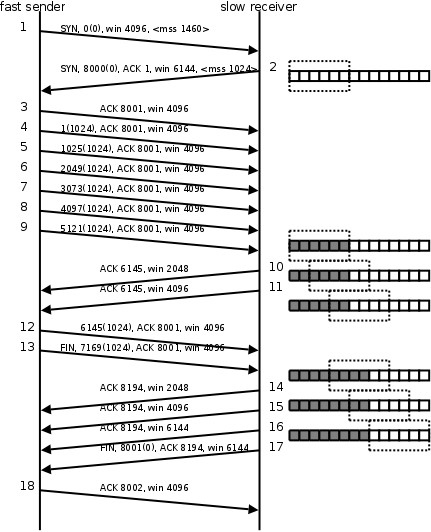
## 28P-四次握手关闭连接



## 29P-半关闭补充说明

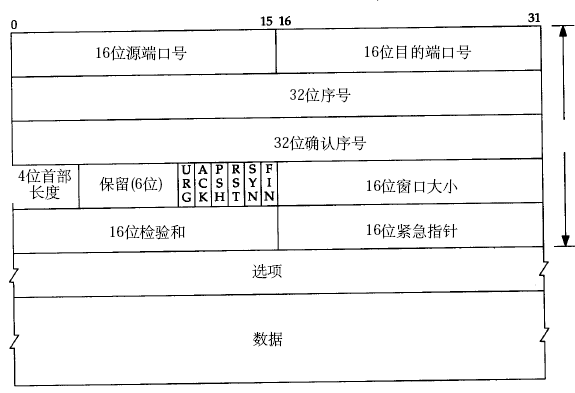
这里其实就是想说明，完成两次挥手后，不是说两端的连接断开了，主动端关闭了写缓冲区，不能再向对端发送数据，被动端关闭了读缓冲区，不能再从对端读取数据。然而主动端还是能够读取对端发来的数据。

## 30P-滑动窗口和TCP数据包格式

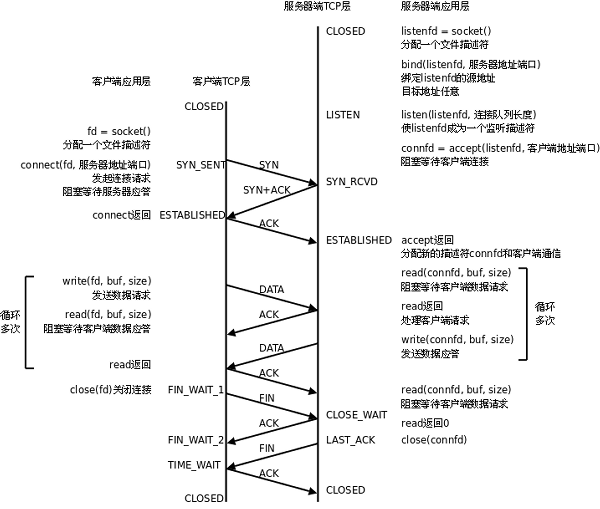


滑动窗口：

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\_
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);
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>
7. **void** perr\_exit(**const** **char** \*s)
8. {
9. perror(s);
10. exit(-1);
11. }
13. **int** Accept(**int** fd, **struct** sockaddr \*sa, socklen\_t \*salenptr)
14. {
15. **int** n;
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. }
27. **int** Bind(**int** fd, **const** **struct** sockaddr \*sa, socklen\_t salen)
28. {
29. **int** n;
31. **if** ((n = bind(fd, sa, salen)) < 0)
32. perr\_exit("bind error");
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 存放 网络通信相关常用 自定义函数原型(声明)。

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 存放 网络通信相关常用 自定义函数原型(声明)。

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 scokaddr\_in addr;

3. Listen();

4. while (1) {

cfd = Accept(lfd, );

pthread\_create(&tid, NULL, tfn, (void \*)cfd);

pthread\_detach(tid); // pthead\_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）;

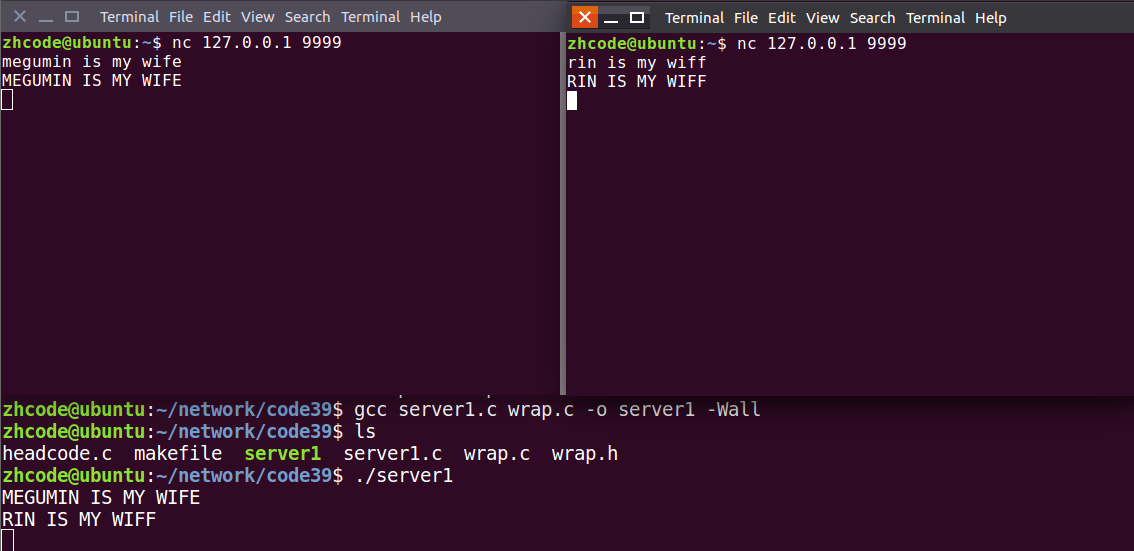
}

## 39P-多进程并发服务器实现

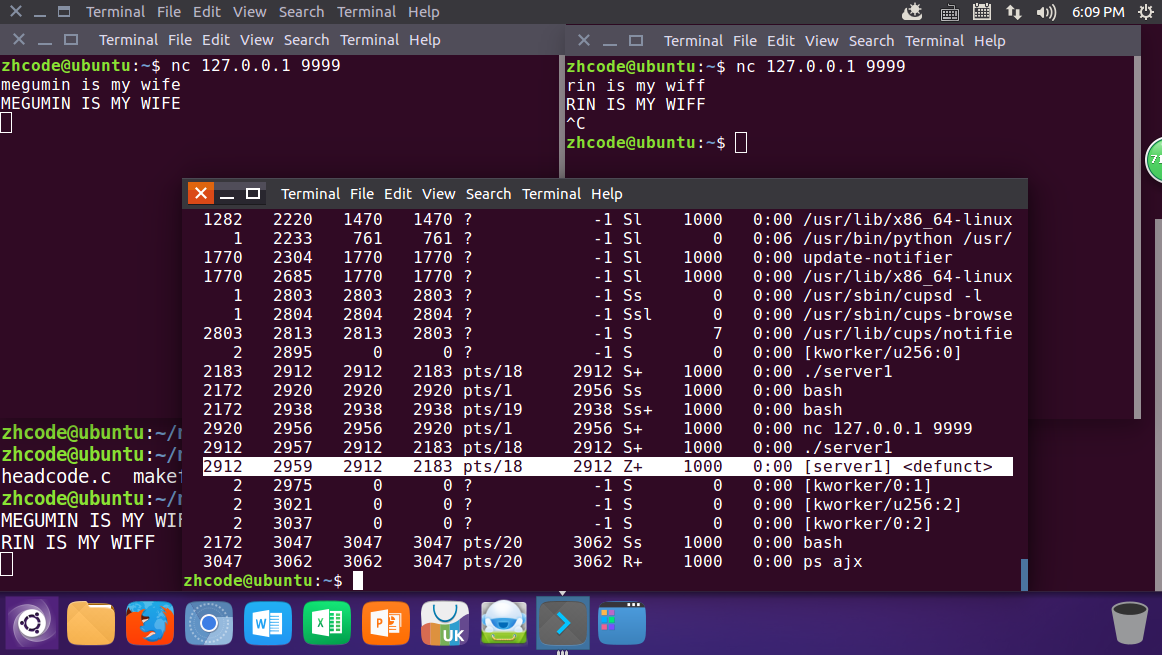
第一个版本的代码如下：

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>
14. #include "wrap.h"
16. #define SRV\_PORT 9999
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;
27. //memset(&srv\_addr, 0, sizeof(srv\_addr));                 // 将地址结构清零
28. bzero(&srv\_addr, **sizeof**(srv\_addr));
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);
34. lfd = Socket(AF\_INET, SOCK\_STREAM, 0);
36. Bind(lfd, (**struct** sockaddr \*)&srv\_addr, **sizeof**(srv\_addr));
38. Listen(lfd, 128);
40. clt\_addr\_len = **sizeof**(clt\_addr);
42. **while** (1) {
44. cfd = Accept(lfd, (**struct** sockaddr \*)&clt\_addr, &clt\_addr\_len);
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. }
58. **if** (pid == 0) {
59. **for** (;;) {
60. ret = Read(cfd, buf, **sizeof**(buf));
61. **if** (ret == 0) {
62. close(cfd);
63. exit(1);
64. }
66. **for** (i = 0; i < ret; i++)
67. buf[i] = toupper(buf[i]);
69. write(cfd, buf, ret);
70. write(STDOUT\_FILENO, buf, ret);
71. }
72. }
74. **return** 0;
75. }

编译运行，结果如下：



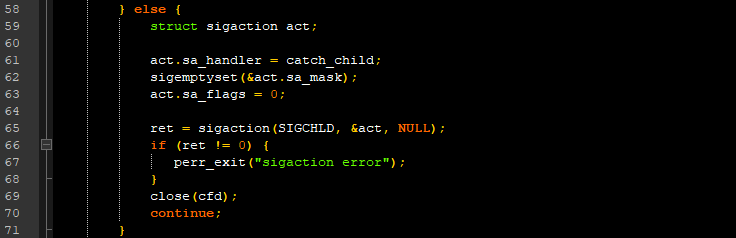
这个代码，有问题。我们Ctrl+C终止一个连接进程，会发现，有僵尸进程。

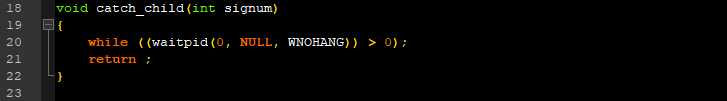


如上图所示，有个僵尸进程。这是因为父进程在阻塞等待，没来得及去回收这个子进程。

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：





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>
14. #include "wrap.h"
16. #define SRV\_PORT 9999
18. **void** catch\_child(**int** signum)
19. {
20. **while** ((waitpid(0, NULL, WNOHANG)) > 0);
21. **return** ;
22. }
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;
33. //memset(&srv\_addr, 0, sizeof(srv\_addr));                 // 将地址结构清零
34. bzero(&srv\_addr, **sizeof**(srv\_addr));
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);
40. lfd = Socket(AF\_INET, SOCK\_STREAM, 0);
42. Bind(lfd, (**struct** sockaddr \*)&srv\_addr, **sizeof**(srv\_addr));
44. Listen(lfd, 128);
46. clt\_addr\_len = **sizeof**(clt\_addr);
48. **while** (1) {
50. cfd = Accept(lfd, (**struct** sockaddr \*)&clt\_addr, &clt\_addr\_len);
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;
61. act.sa\_handler = catch\_child;
62. sigemptyset(&act.sa\_mask);
63. act.sa\_flags = 0;
65. ret = sigaction(SIGCHLD, &act, NULL);
66. **if** (ret != 0) {
67. perr\_exit("sigaction error");
68. }
69. close(cfd);
70. **continue**;
71. }
72. }
74. **if** (pid == 0) {
75. **for** (;;) {
76. ret = Read(cfd, buf, **sizeof**(buf));
77. **if** (ret == 0) {
78. close(cfd);
79. exit(1);
80. }
82. **for** (i = 0; i < ret; i++)
83. buf[i] = toupper(buf[i]);
85. write(cfd, buf, ret);
86. write(STDOUT\_FILENO, buf, ret);
87. }
88. }
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

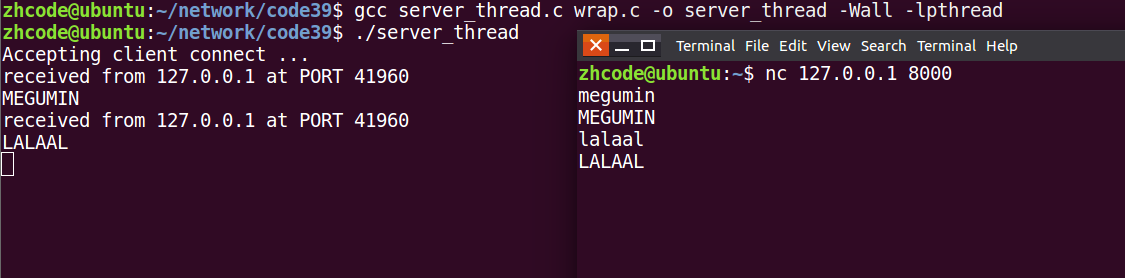
scp -r 源地址 目标地址

## 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>
9. #include "wrap.h"
11. #define MAXLINE 8192
12. #define SERV\_PORT 8000
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. };
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
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)
36. **for** (i = 0; i < n; i++)
37. buf[i] = toupper(buf[i]);                           //小写-->大写
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);
44. **return** (**void** \*)0;
45. }
47. **int** main(**void**)
48. {
49. **struct** sockaddr\_in servaddr, cliaddr;
50. socklen\_t cliaddr\_len;
51. **int** listenfd, connfd;
52. pthread\_t tid;
54. **struct** s\_info ts[256];      //创建结构体数组.
55. **int** i = 0;
57. listenfd = Socket(AF\_INET, SOCK\_STREAM, 0);                     //创建一个socket, 得到lfd
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
64. Bind(listenfd, (**struct** sockaddr \*)&servaddr, **sizeof**(servaddr));             //绑定
66. Listen(listenfd, 128);                                                      //设置同一时刻链接服务器上限数
68. printf("Accepting client connect ...\n");
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;
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. }
81. **return** 0;
82. }

编译运行，结果如下：



## 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 存放 网络通信相关常用 自定义函数原型(声明)。

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 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()）;

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 scokaddr\_in addr;

3. Listen();

4. while (1) {

cfd = Accept(lfd, );

pthread\_create(&tid, NULL, tfn, (void \*)cfd);

pthread\_detach(tid); // pthead\_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)

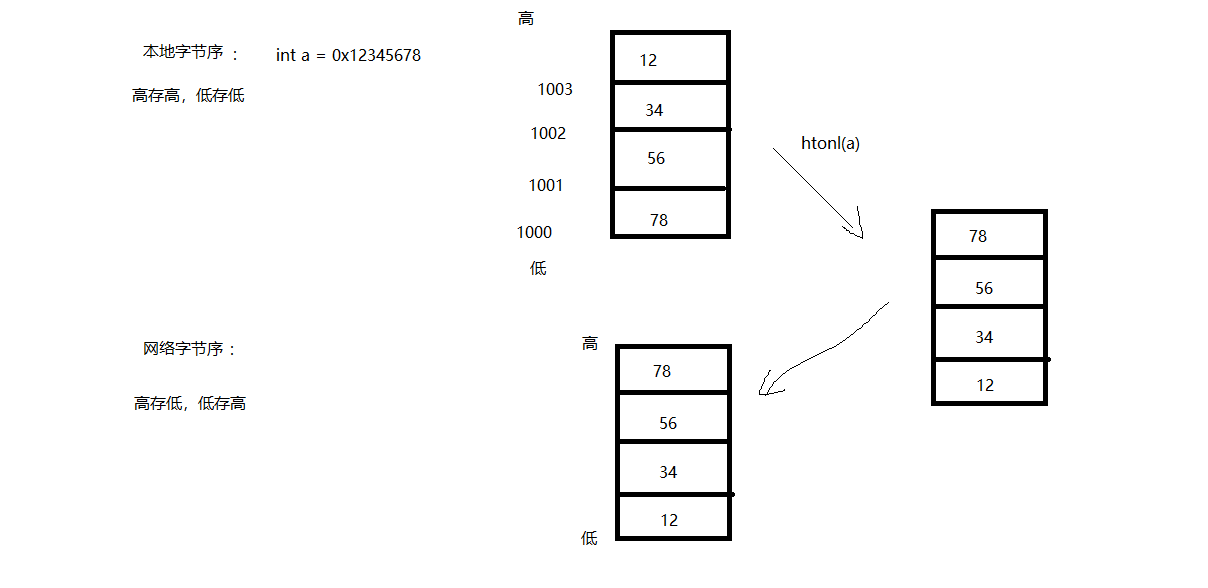
小--大

write(cfd)

pthread\_exit（(void \*)10）;

}

## 44P-复习



## 45P-TCP状态-主动发起连接

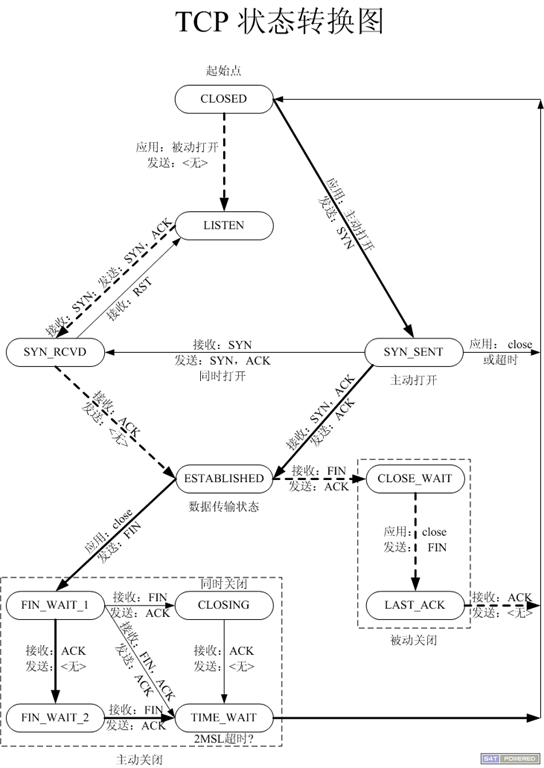
## 46P-TCP状态-主动关闭连接

## 47P-TCP状态-被动接收连接

## 48P-TCP状态-被动关闭连接

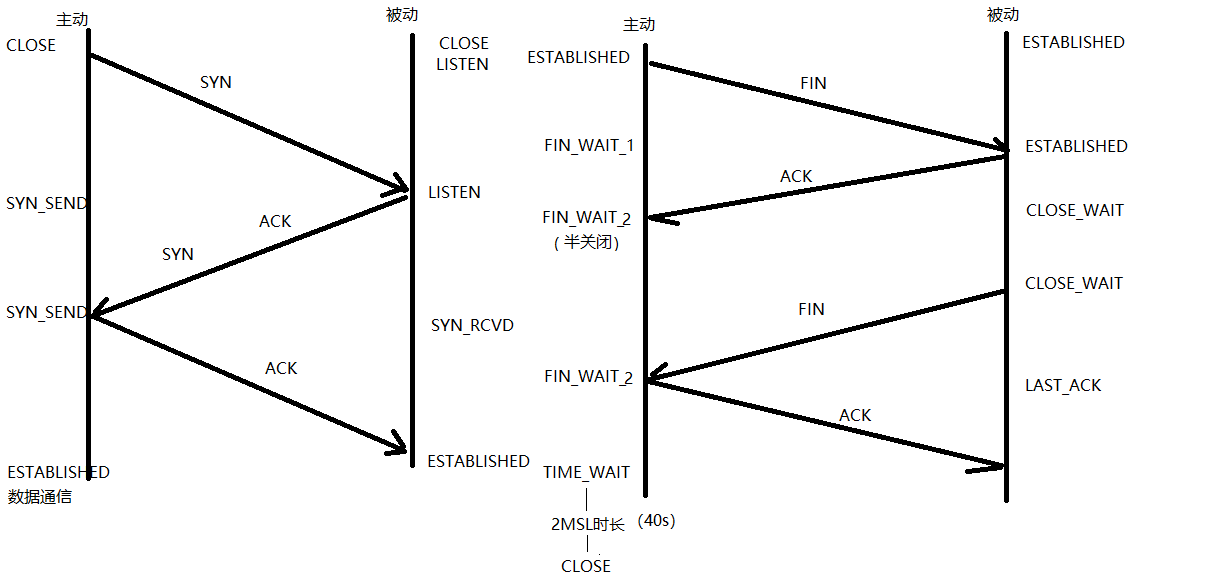
## 49P-2MSL时长

## 50P-TCP状态-其他状态



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);

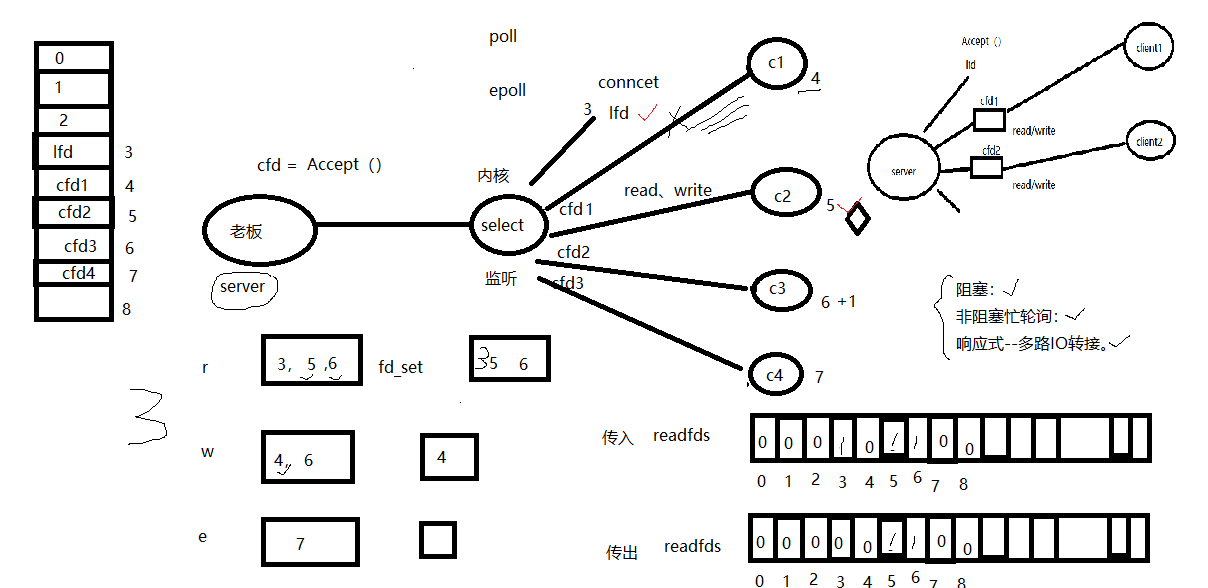
how: SHUT\_RD 关读端

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 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

## 55P-中午复习

## 56P-select函数原型分析

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

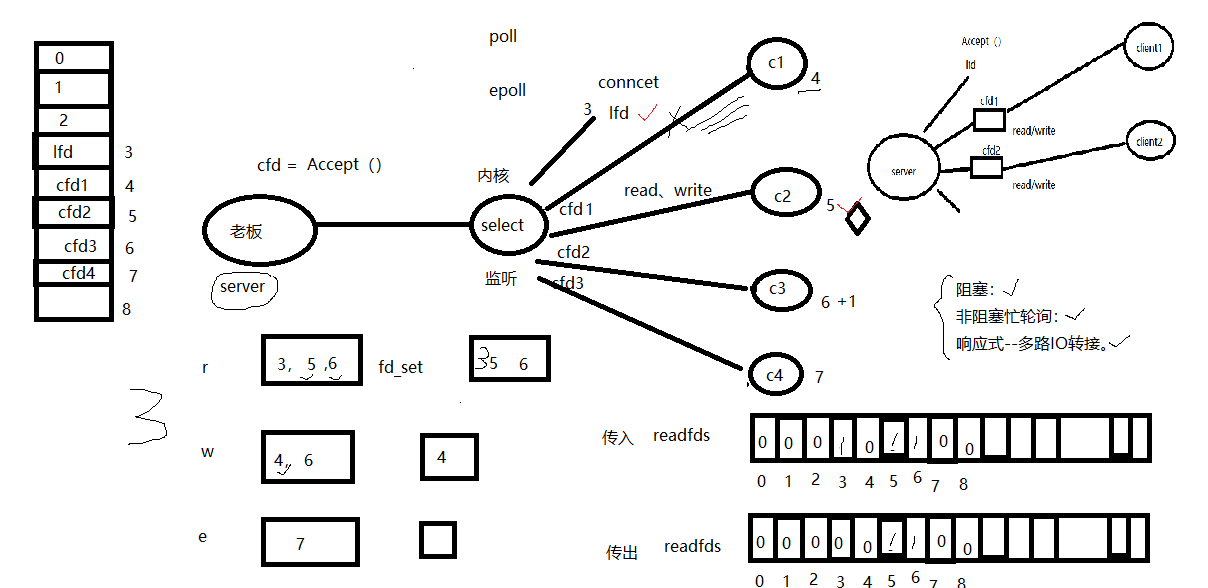
0： 非阻塞监听，轮询

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



## 57P-select相关函数参数分析

void FD\_CLR(int fd, fd\_set \*set) 把某一个fd清除出去

int FD\_ISSET(int fd, fd\_set \*set) 判定某个fd是否在位图中

void FD\_SET(int fd, fd\_set \*set) 把某一个fd添加到位图

void FD\_ZERO(fd\_set \*set) 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

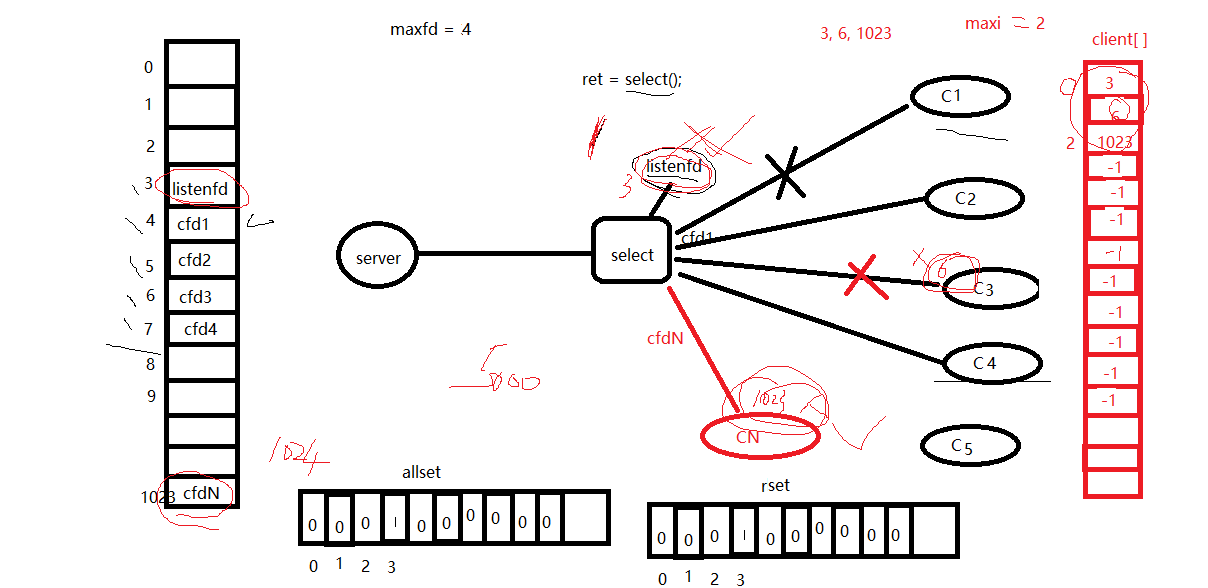
write();

}

}

}

## 59P-select实现多路IO转接-代码review

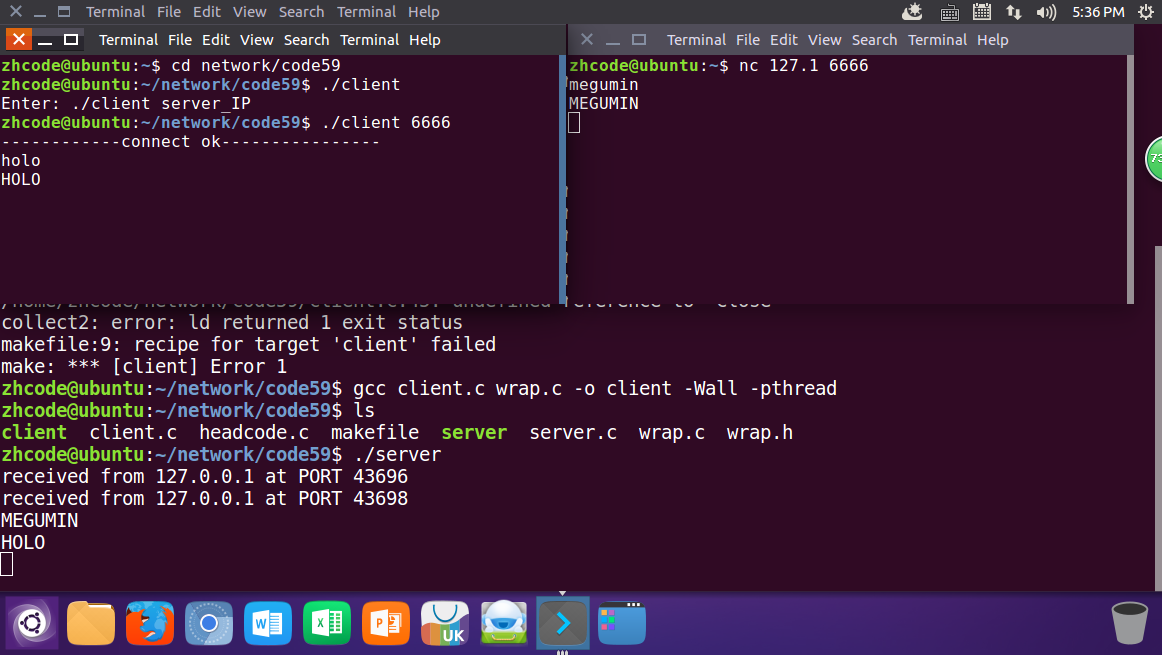


代码如下：

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>
8. #include "wrap.h"
10. #define SERV\_PORT 6666
12. **int** main(**int** argc, **char** \*argv[])
13. {
14. **int** i, j, n, nready;
16. **int** maxfd = 0;
18. **int** listenfd, connfd;
20. **char** buf[BUFSIZ];         /\* #define INET\_ADDRSTRLEN 16 \*/
22. **struct** sockaddr\_in clie\_addr, serv\_addr;
23. socklen\_t clie\_addr\_len;
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);

36. fd\_set rset, allset;                            /\* rset 读事件文件描述符集合 allset用来暂存 \*/
38. maxfd = listenfd;
40. FD\_ZERO(&allset);
41. FD\_SET(listenfd, &allset);                                  /\* 构造select监控文件描述符集 \*/
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");
49. **if** (FD\_ISSET(listenfd, &rset)) {                        /\* 说明有新的客户端链接请求 \*/
51. clie\_addr\_len = **sizeof**(clie\_addr);
52. connfd = Accept(listenfd, (**struct** sockaddr \*)&clie\_addr, &clie\_addr\_len);       /\* Accept 不会阻塞 \*/
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 \*/
56. **if** (maxfd < connfd)
57. maxfd = connfd;
59. **if** (0 == --nready)                                  /\* 只有listenfd有事件, 后续的 for 不需执行 \*/
60. **continue**;
61. }
63. **for** (i = listenfd+1; i <= maxfd; i++) {                 /\* 检测哪个clients 有数据就绪 \*/
65. **if** (FD\_ISSET(i, &rset)) {
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 \*/
71. } **else** **if** (n > 0) {
73. **for** (j = 0; j < n; j++)
74. buf[j] = toupper(buf[j]);
75. Write(i, buf, n);
76. }
77. }
78. }
79. }
81. Close(listenfd);
83. **return** 0;
84. }

编译运行，结果如下：



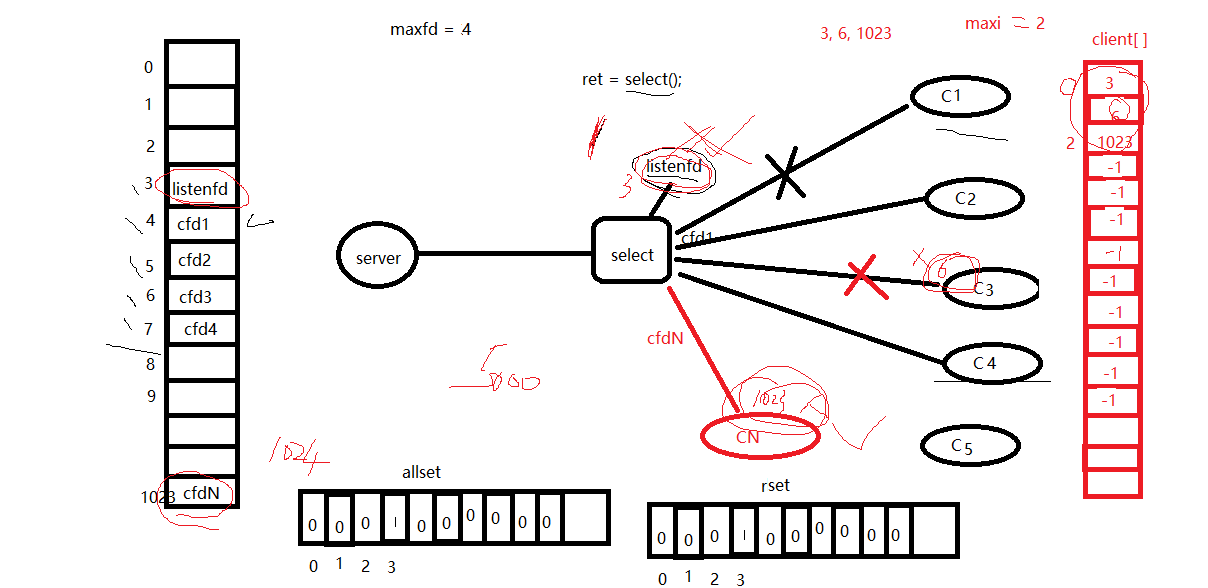
如图，借助select也可以实现多线程

## 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.

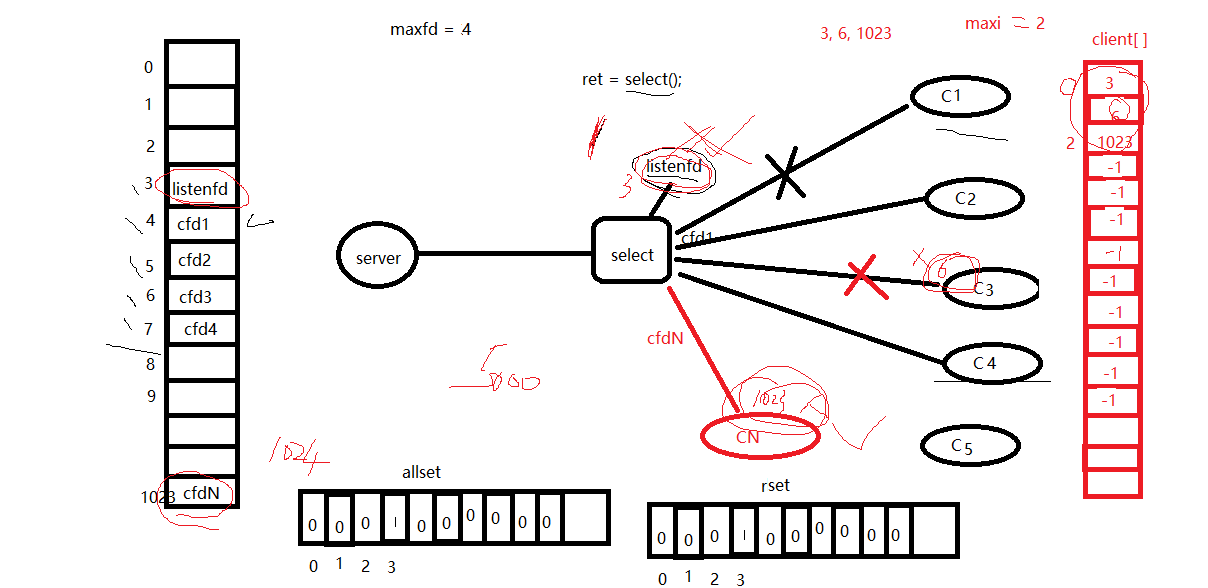
检测满足条件的fd， 自己添加业务逻辑提高小。 提高了编码难度。

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

## 63P-添加一个自定义数组提高效率

这里就是改进之前代码的问题，之前的代码，如果最大fd是1023，每次确定有事件发生的fd时，就要扫描3-1023的所有文件描述符，这看起来很蠢。于是定义一个数组，把要监听的文件描述符存下来，每次扫描这个数组就行了。看起来科学得多。



如图，加个client数组，存要监听的描述符。

代码如下，挺长的

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>
8. #include "wrap.h"
10. #define SERV\_PORT 6666
12. **int** main(**int** argc, **char** \*argv[])
13. {
14. **int** i, j, n, maxi;
16. **int** nready, client[FD\_SETSIZE];                 /\* 自定义数组client, 防止遍历1024个文件描述符  FD\_SETSIZE默认为1024 \*/
17. **int** maxfd, listenfd, connfd, sockfd;
18. **char** buf[BUFSIZ], str[INET\_ADDRSTRLEN];         /\* #define INET\_ADDRSTRLEN 16 \*/
20. **struct** sockaddr\_in clie\_addr, serv\_addr;
21. socklen\_t clie\_addr\_len;
22. fd\_set rset, allset;                            /\* rset 读事件文件描述符集合 allset用来暂存 \*/
24. listenfd = Socket(AF\_INET, SOCK\_STREAM, 0);
26. **int** opt = 1;
27. setsockopt(listenfd, SOL\_SOCKET, SO\_REUSEADDR, &opt, **sizeof**(opt));
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);
34. Bind(listenfd, (**struct** sockaddr \*)&serv\_addr, **sizeof**(serv\_addr));
35. Listen(listenfd, 128);
37. maxfd = listenfd;                                           /\* 起初 listenfd 即为最大文件描述符 \*/
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[] \*/
43. FD\_ZERO(&allset);
44. FD\_SET(listenfd, &allset);                                  /\* 构造select监控文件描述符集 \*/
46. **while** (1) {
47. rset = allset;                                          /\* 每次循环时都重新设置select监控信号集 \*/
49. nready = select(maxfd+1, &rset, NULL, NULL, NULL);  //2  1--lfd  1--connfd
50. **if** (nready < 0)
51. perr\_exit("select error");
53. **if** (FD\_ISSET(listenfd, &rset)) {                        /\* 说明有新的客户端链接请求 \*/
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));
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. }
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. }
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 \*/
74. **if** (connfd > maxfd)
75. maxfd = connfd;                                 /\* select第一个参数需要 \*/
77. **if** (i > maxi)
78. maxi = i;                                       /\* 保证maxi存的总是client[]最后一个元素下标 \*/
80. **if** (--nready == 0)
81. **continue**;
82. }
84. **for** (i = 0; i <= maxi; i++) {                               /\* 检测哪个clients 有数据就绪 \*/
86. **if** ((sockfd = client[i]) < 0)
87. **continue**;
88. **if** (FD\_ISSET(sockfd, &rset)) {
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, 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。

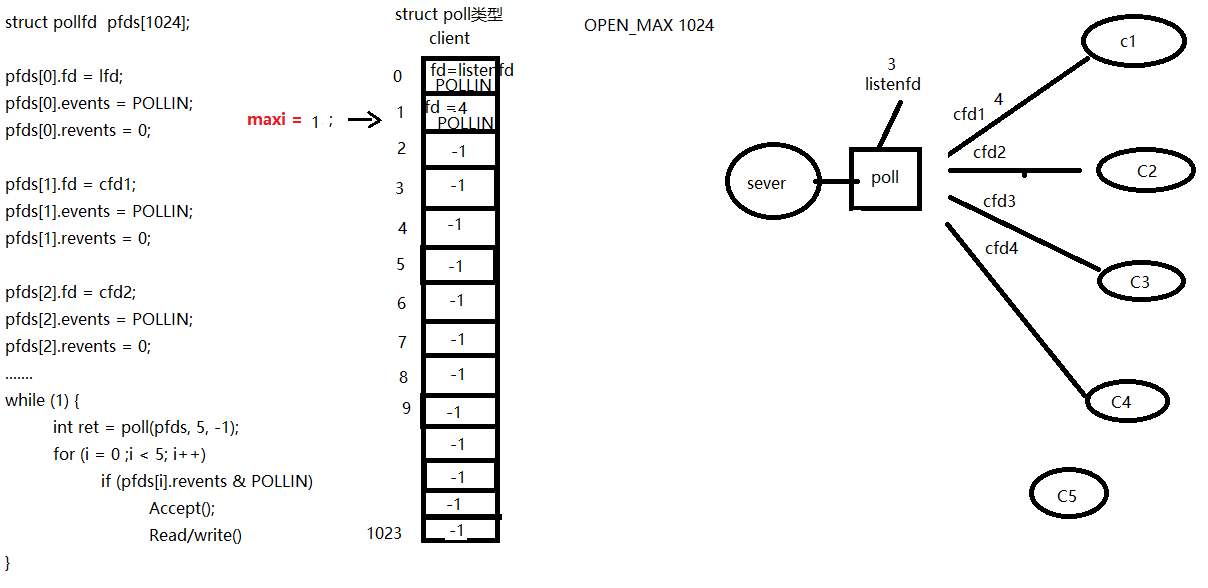
拓展 监听上限。 超出 1024限制。

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。

## 67P-poll函数使用注意事项示例



## 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"
11. #define MAXLINE 80
12. #define SERV\_PORT 6666
13. #define OPEN\_MAX 1024
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;
25. listenfd = Socket(AF\_INET, SOCK\_STREAM, 0);
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);
32. Bind(listenfd, (**struct** sockaddr \*)&servaddr, **sizeof**(servaddr));
34. Listen(listenfd, 20);
36. client[0].fd = listenfd;
37. client[0].events = POLLRDNORM;                  /\* listenfd监听普通读事件 \*/
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 \*/
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. }
58. **if** (i == OPEN\_MAX)
59. perr\_exit("too many clients");
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。

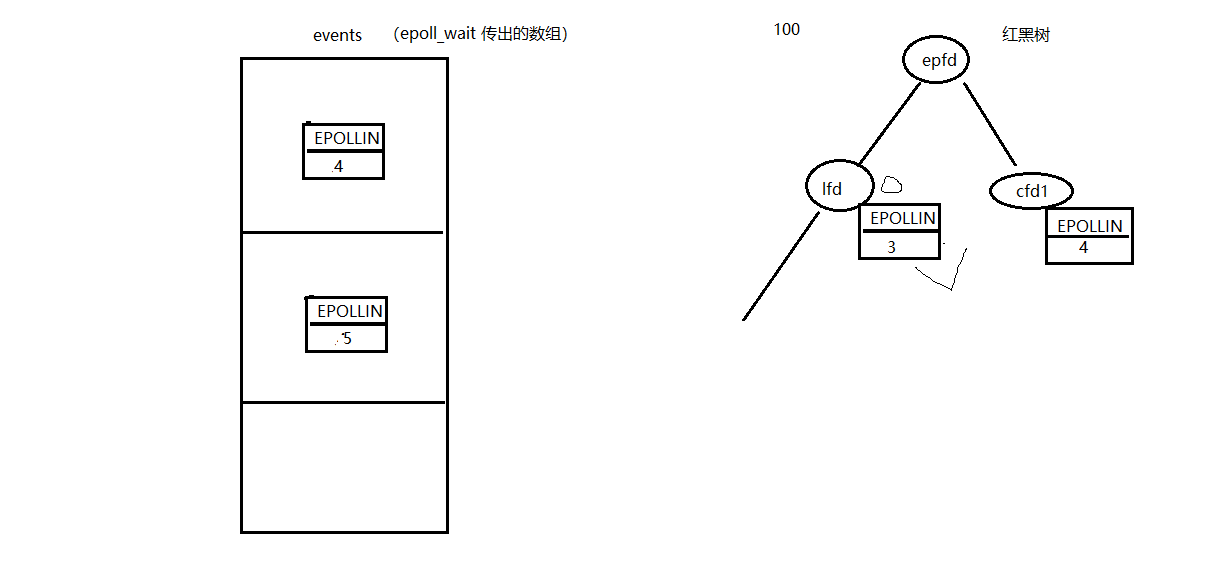
拓展 监听上限。 超出 1024限制。

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>
10. #include "wrap.h"
12. #define MAXLINE 8192
13. #define SERV\_PORT 8000
15. #define OPEN\_MAX 5000
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;
25. **struct** sockaddr\_in cliaddr, servaddr;
26. **struct** epoll\_event tep, ep[OPEN\_MAX];       //tep: epoll\_ctl参数  ep[] : epoll\_wait参数
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);
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");
42. tep.events = EPOLLIN;
43. tep.data.fd = listenfd;           //指定lfd的监听时间为"读"
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");
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");
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**;
59. **if** (ep[i].data.fd == listenfd) {    //判断满足事件的fd是不是lfd
60. clilen = **sizeof**(cliaddr);
61. connfd = Accept(listenfd, (**struct** sockaddr \*)&cliaddr, &clilen);    //接受链接
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);
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");
73. } **else** {                                                    //不是lfd,
74. sockfd = ep[i].data.fd;
75. n = Read(sockfd, buf, MAXLINE);
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);                                      //关闭与该客户端的链接
82. printf("client[%d] closed connection\n", sockfd);
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);
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
93. Write(STDOUT\_FILENO, buf, n);
94. Writen(sockfd, buf, n);
95. }
96. }
97. }
98. }
99. Close(listenfd);
100. Close(efd);
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

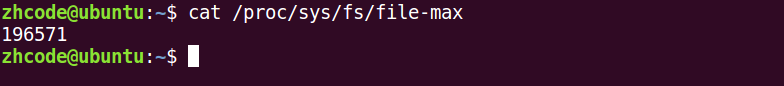
修改：

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

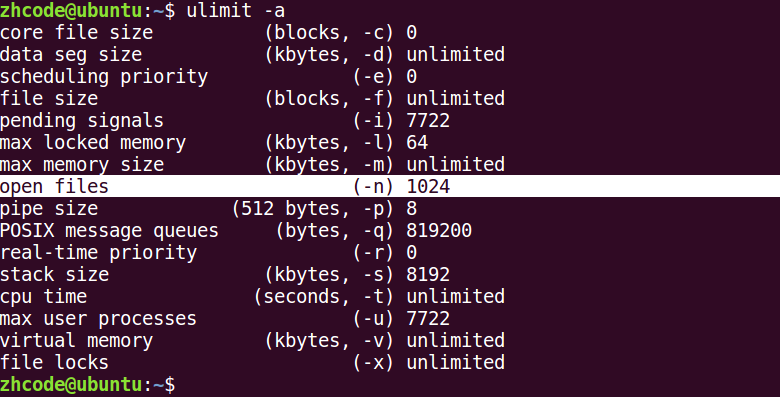
\* soft nofile 65536 --> 设置默认值， 可以直接借助命令修改。 【注销用户，使其生效】

\* hard nofile 100000 --> 命令修改上限。

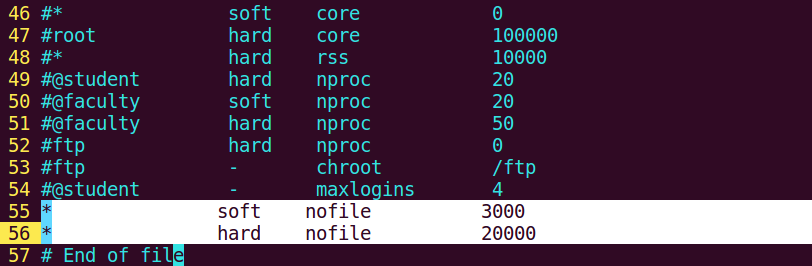
cat /proc/sys/fs/file-max 查看最大文件描述符上限



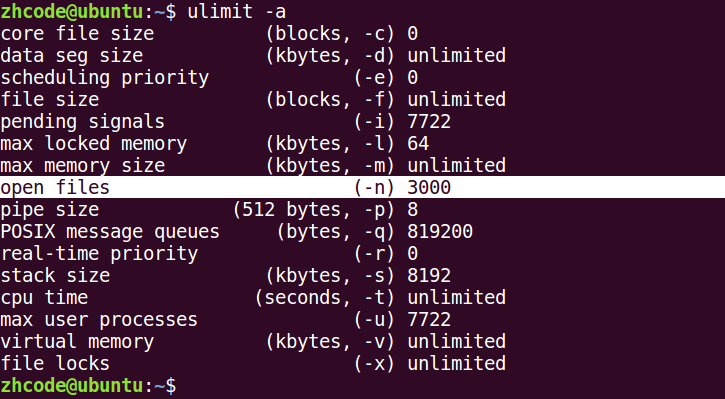
ulimit -a



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

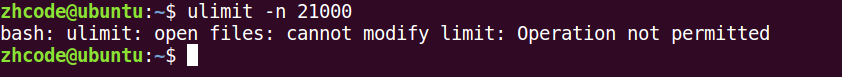


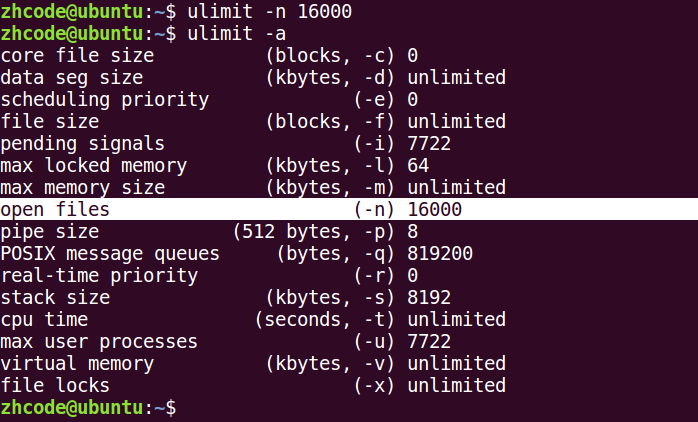
修改之后，注销用户重新登录，查看文件描述符上限：



如图，已经修改成功了。

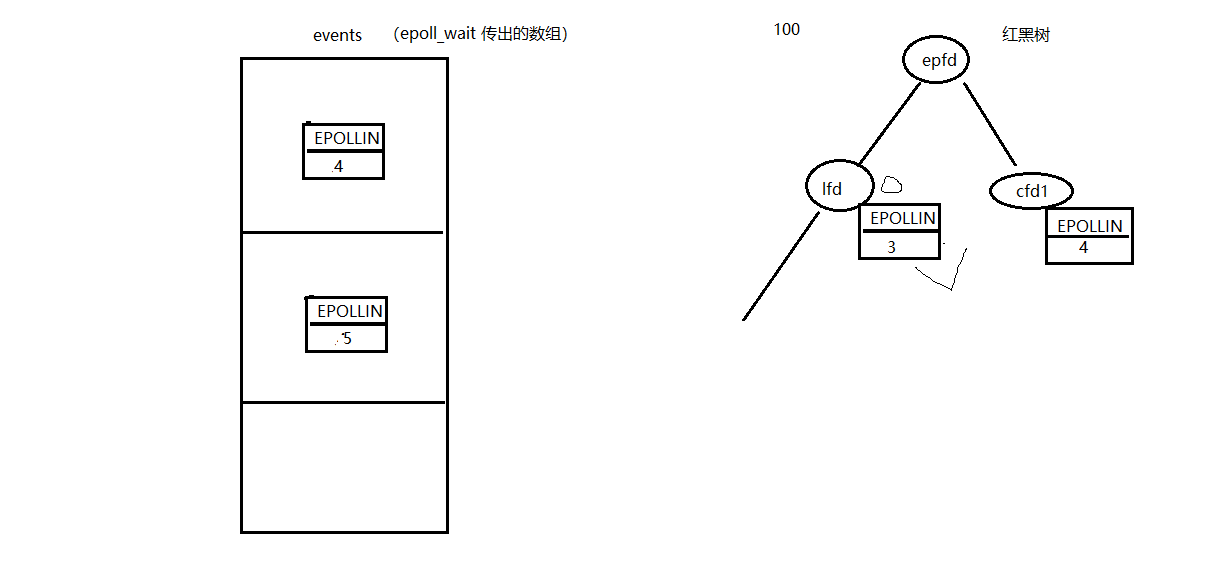
如果使用ulimit -n 来修改，会受到之前设置的hard的限制：





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和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

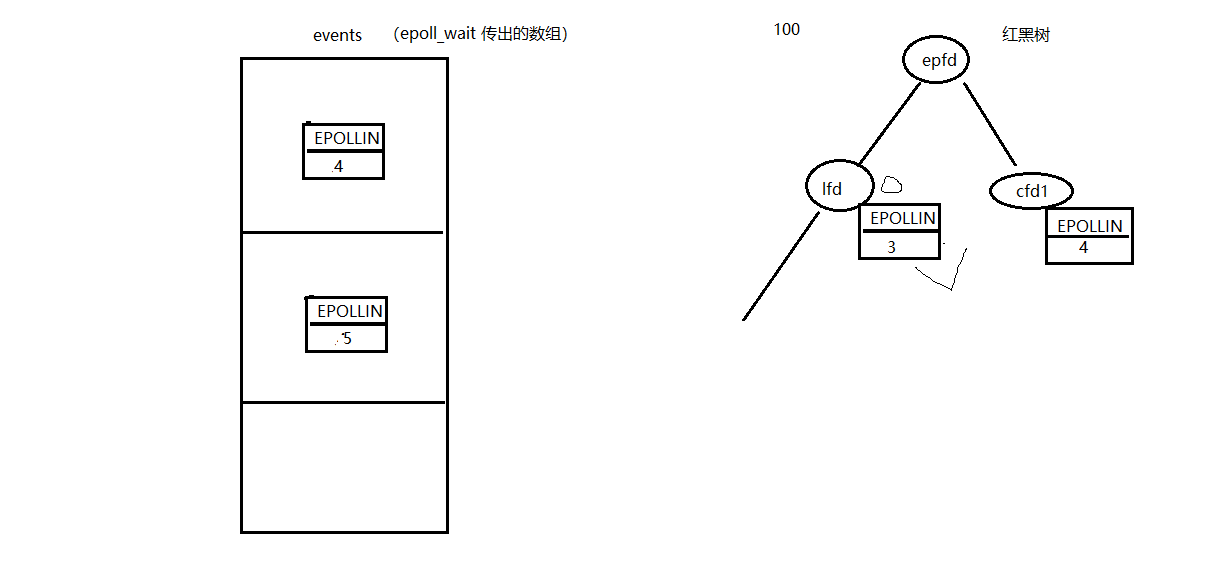
>0: 超时时间 （毫秒）

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--大

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

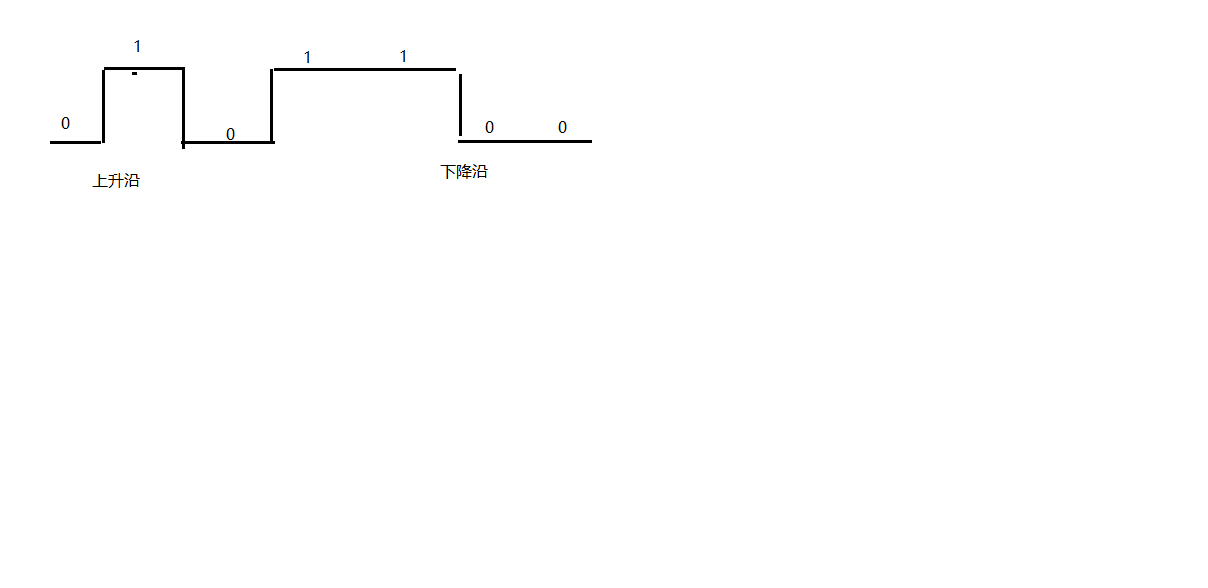
}

}

}

## 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>
7. #define MAXLINE 10
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';
16. pipe(pfd);
17. pid = fork();
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]);
38. } **else** **if** (pid > 0) {       //父 读
39. **struct** epoll\_event event;
40. **struct** epoll\_event resevent[10];        //epoll\_wait就绪返回event
41. **int** res, len;
43. close(pfd[1]);
44. efd = epoll\_create(10);
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);
51. **while** (1) {
52. res = epoll\_wait(efd, resevent, 10, -1);
53. printf("res %d\n", res);
54. **if** (resevent[0].data.fd == pfd[0]) {
55. len = read(pfd[0], buf, MAXLINE/2);
56. write(STDOUT\_FILENO, buf, len);
57. }
58. }
60. close(pfd[0]);
61. close(efd);
63. } **else** {
64. perror("fork");
65. exit(-1);
66. }
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>
11. #define MAXLINE 10
12. #define SERV\_PORT 9000
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;
23. listenfd = socket(AF\_INET, SOCK\_STREAM, 0);
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);
30. bind(listenfd, (**struct** sockaddr \*)&servaddr, **sizeof**(servaddr));
32. listen(listenfd, 20);
34. **struct** epoll\_event event;
35. **struct** epoll\_event resevent[10];
36. **int** res, len;
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 \*/
42. printf("Accepting connections ...\n");
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));
50. event.data.fd = connfd;
51. epoll\_ctl(efd, EPOLL\_CTL\_ADD, connfd, &event);
53. **while** (1) {
54. res = epoll\_wait(efd, resevent, 10, -1);
56. printf("res %d\n", res);
57. **if** (resevent[0].data.fd == connfd) {
58. len = read(connfd, buf, MAXLINE/2);         //readn(500)
59. write(STDOUT\_FILENO, buf, len);
60. }
61. }
63. **return** 0;
64. }

client代码如下：

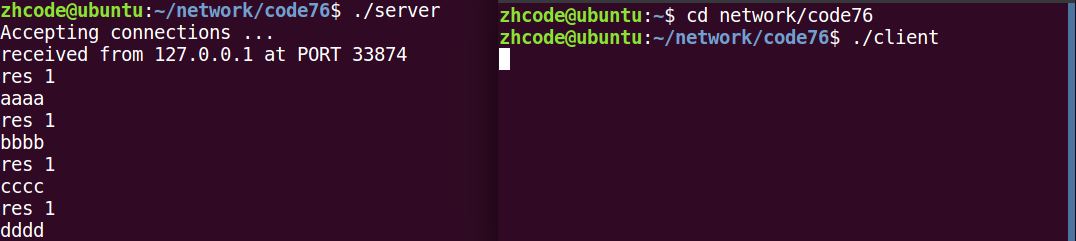
1. #include <stdio.h>
2. #include <string.h>
3. #include <unistd.h>
4. #include <arpa/inet.h>
5. #include <netinet/in.h>
7. #define MAXLINE 10
8. #define SERV\_PORT 9000
10. **int** main(**int** argc, **char** \*argv[])
11. {
12. **struct** sockaddr\_in servaddr;
13. **char** buf[MAXLINE];
14. **int** sockfd, i;
15. **char** ch = 'a';
17. sockfd = socket(AF\_INET, SOCK\_STREAM, 0);
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);
24. connect(sockfd, (**struct** sockaddr \*)&servaddr, **sizeof**(servaddr));
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);
43. **return** 0;
44. }

server边沿触发，编译运行，结果如下：



运行后，每过5秒钟服务器才输出一组字符，这是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：



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>
11. #define MAXLINE 10
12. #define SERV\_PORT 8000
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;
23. listenfd = socket(AF\_INET, SOCK\_STREAM, 0);
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);
30. bind(listenfd, (**struct** sockaddr \*)&servaddr, **sizeof**(servaddr));
32. listen(listenfd, 20);
34. ///////////////////////////////////////////////////////////////////////
35. **struct** epoll\_event event;
36. **struct** epoll\_event res\_event[10];
37. **int** res, len;
39. efd = epoll\_create(10);
41. event.events = EPOLLIN | EPOLLET;     /\* ET 边沿触发，默认是水平触发 \*/
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));
51. flag = fcntl(connfd, F\_GETFL);          /\* 修改connfd为非阻塞读 \*/
52. flag |= O\_NONBLOCK;
53. fcntl(connfd, F\_SETFL, flag);
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);
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. }
68. **return** 0;
69. }

其实就是多了这几行：



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>
15. #define MAX\_EVENTS  1024                                    //监听上限数
16. #define BUFLEN 4096
17. #define SERV\_PORT   8080
19. **void** recvdata(**int** fd, **int** events, **void** \*arg);
20. **void** senddata(**int** fd, **int** events, **void** \*arg);
22. /\* 描述就绪文件描述符相关信息 \*/
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. };
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

39. /\*将结构体 myevent\_s 成员变量 初始化\*/
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
52. **return**;
53. }
55. /\* 向 epoll监听的红黑树 添加一个 文件描述符 \*/
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
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. }
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);
75. **return** ;
76. }
78. /\* 从epoll 监听的 红黑树中删除一个 文件描述符\*/
80. **void** eventdel(**int** efd, **struct** myevent\_s \*ev)
81. {
82. **struct** epoll\_event epv = {0, {0}};
84. **if** (ev->status != 1)                                        //不在红黑树上
85. **return** ;
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
92. **return** ;
93. }
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 \*/
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;
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. }
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
116. **if** (i == MAX\_EVENTS) {
117. printf("%s: max connect limit[%d]\n", \_\_func\_\_, MAX\_EVENTS);
118. **break**;                                                      //跳出do while(0) 不执行后续代码
119. }
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. }
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,监听读事件
131. } **while**(0);
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. }
138. **void** recvdata(**int** fd, **int** events, **void** \*arg)
139. {
140. **struct** myevent\_s \*ev = (**struct** myevent\_s \*)arg;
141. **int** len;
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
145. eventdel(g\_efd, ev);        //将该节点从红黑树上摘除
147. **if** (len > 0) {
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);
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
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[%ld], 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. }
165. **return**;
166. }
168. **void** senddata(**int** fd, **int** events, **void** \*arg)
169. {
170. **struct** myevent\_s \*ev = (**struct** myevent\_s \*)arg;
171. **int** len;
173. len = send(fd, ev->buf, ev->len, 0);                    //直接将数据 回写给客户端。未作处理
175. eventdel(g\_efd, ev);                                //从红黑树g\_efd中移除
177. **if** (len > 0) {
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
183. } **else** {
184. close(ev->fd);                                      //关闭链接
185. printf("send[fd=%d] error %s\n", fd, strerror(errno));
186. }
188. **return** ;
189. }
191. /\*创建 socket, 初始化lfd \*/
193. **void** initlistensocket(**int** efd, **short** port)
194. {
195. **struct** sockaddr\_in sin;
197. **int** lfd = socket(AF\_INET, SOCK\_STREAM, 0);
198. fcntl(lfd, F\_SETFL, O\_NONBLOCK);                                            //将socket设为非阻塞
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);
205. bind(lfd, (**struct** sockaddr \*)&sin, **sizeof**(sin));
207. listen(lfd, 20);
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]);
212. /\* void eventadd(int efd, int events, struct myevent\_s \*ev) \*/
213. eventadd(efd, EPOLLIN, &g\_events[MAX\_EVENTS]);
215. **return** ;
216. }
218. **int** main(**int** argc, **char** \*argv[])
219. {
220. unsigned **short** port = SERV\_PORT;
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
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));
229. initlistensocket(g\_efd, port);                      //初始化监听socket
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);
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 \*/
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**;
245. **long** duration = now - g\_events[checkpos].last\_active;       //客户端不活跃的世间
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. }
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. }
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;
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. }
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 --> 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。

## 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 == EAGIN 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 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。

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()

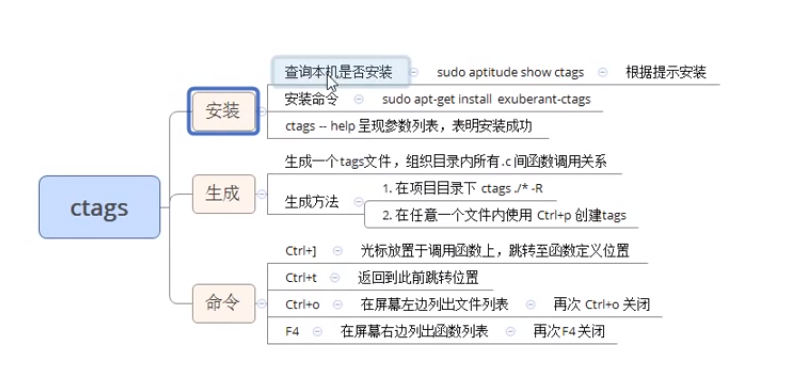
write --- send();

## 87P-复习

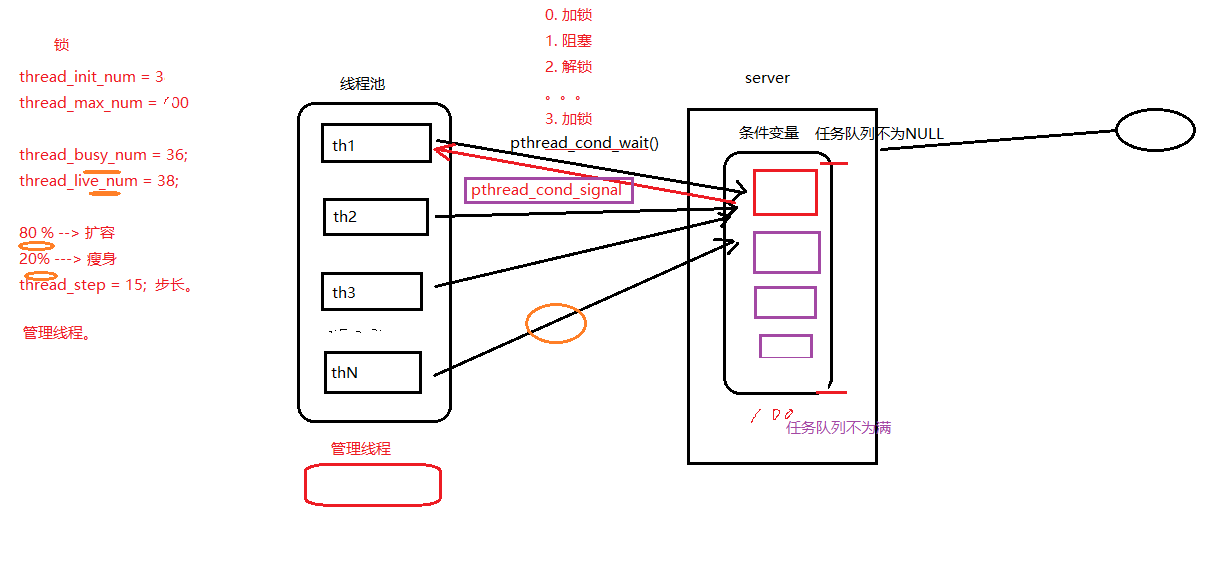
## 88P-补充说明epoll的man手册

## 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 \*/

## 92P-线程池描述结构体

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 \*/

};

## 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：

传输速度块。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

accpet(); ---- 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 errn。 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>
7. #define SERV\_PORT 8000
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;
18. sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);
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);
25. bind(sockfd, (**struct** sockaddr \*)&serv\_addr, **sizeof**(serv\_addr));
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");
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));
38. **for** (i = 0; i < n; i++)
39. buf[i] = toupper(buf[i]);
41. n = sendto(sockfd, buf, n, 0, (**struct** sockaddr \*)&clie\_addr, **sizeof**(clie\_addr));
42. **if** (n == -1)
43. perror("sendto error");
44. }
46. close(sockfd);
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>
7. #define SERV\_PORT 8000
9. **int** main(**int** argc, **char** \*argv[])
10. {
11. **struct** sockaddr\_in servaddr;
12. **int** sockfd, n;
13. **char** buf[BUFSIZ];
15. sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);
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);
22. bind(sockfd, (**struct** sockaddr \*)&servaddr, **sizeof**(servaddr));
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");
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");
33. write(STDOUT\_FILENO, buf, n);
34. }
36. close(sockfd);
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\_adrr;

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>
11. #include "wrap.h"
13. #define SERV\_ADDR  "serv.socket"
15. **int** main(**void**)
16. {
17. **int** lfd, cfd, len, size, i;
18. **struct** sockaddr\_un servaddr, cliaddr;
19. **char** buf[4096];
21. lfd = Socket(AF\_UNIX, SOCK\_STREAM, 0);
23. bzero(&servaddr, **sizeof**(servaddr));
24. servaddr.sun\_family = AF\_UNIX;
25. strcpy(servaddr.sun\_path, SERV\_ADDR);
27. len = offsetof(**struct** sockaddr\_un, sun\_path) + strlen(servaddr.sun\_path);     /\* servaddr total len \*/
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) \*/
32. Listen(lfd, 20);
34. printf("Accept ...\n");
35. **while** (1) {
36. len = **sizeof**(cliaddr);  //AF\_UNIX大小+108B
38. cfd = Accept(lfd, (**struct** sockaddr \*)&cliaddr, (socklen\_t \*)&len);
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 \*/
43. printf("client bind filename %s\n", cliaddr.sun\_path);
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);
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>
12. #include "wrap.h"
14. #define SERV\_ADDR "serv.socket"
15. #define CLIE\_ADDR "clie.socket"
17. **int** main(**void**)
18. {
19. **int**  cfd, len;
20. **struct** sockaddr\_un servaddr, cliaddr;
21. **char** buf[4096];
23. cfd = Socket(AF\_UNIX, SOCK\_STREAM, 0);
25. bzero(&cliaddr, **sizeof**(cliaddr));
26. cliaddr.sun\_family = AF\_UNIX;
27. strcpy(cliaddr.sun\_path,CLIE\_ADDR);
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 \*/
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\*/

35. bzero(&servaddr, **sizeof**(servaddr));                                          /\* 构造server 地址 \*/
36. servaddr.sun\_family = AF\_UNIX;
37. strcpy(servaddr.sun\_path, SERV\_ADDR);
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 \*/
41. Connect(cfd, (**struct** sockaddr \*)&servaddr, len);
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. }
49. close(cfd);
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](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](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。

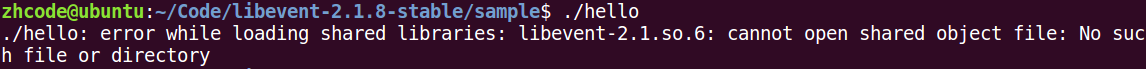
编译使用库的 .c 时，需要加 -levent 选项。

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：



解决办法：

[解决这个问题的博客](https://blog.csdn.net/deeplan_1994/article/details/83927832?utm_medium=distribute.pc_relevant_right.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase&depth_1-utm_source=distribute.pc_relevant_right.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase)



完事儿运行测试，结果如下：



## 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

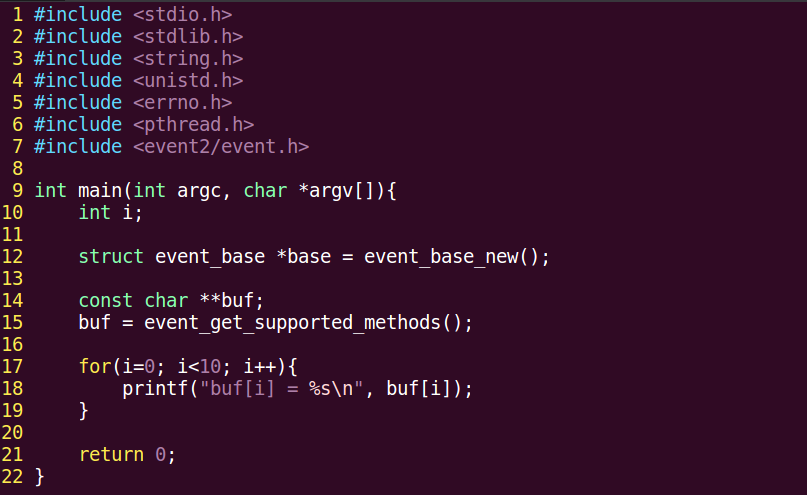
特性：

基于“事件”异步通信模型。--- 回调。

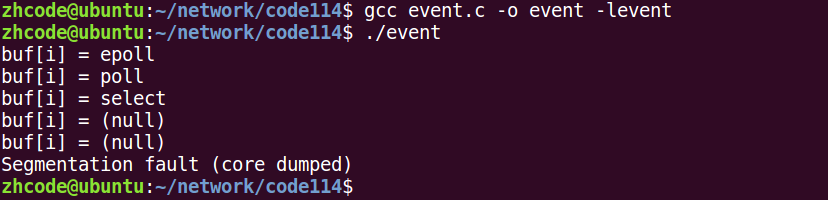
## 114P-框架相关的不常用函数

查看支持哪些多路IO：

代码如下：



编译运行，结果如下：



## 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\_WRTIE 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>
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};
16. **int** len = read(fd, buf, **sizeof**(buf));
18. printf("read event: %s \n", what & EV\_READ ? "Yes" : "No");
19. printf("data len = %d, buf = %s\n", len, buf);
21. sleep(1);
22. }

25. // 读管道
26. **int** main(**int** argc, **const** **char**\* argv[])
27. {
28. unlink("myfifo");
30. //创建有名管道
31. mkfifo("myfifo", 0664);
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. }
42. // 创建个event\_base
43. **struct** event\_base\* base = NULL;
44. base = event\_base\_new();
46. // 创建事件
47. **struct** event\* ev = NULL;
48. ev = event\_new(base, fd, EV\_READ | EV\_PERSIST, read\_cb, NULL);
50. // 添加事件
51. event\_add(ev, NULL);
53. // 事件循环
54. event\_base\_dispatch(base);  // while（1） { epoll();}
56. // 释放资源
57. event\_free(ev);
58. event\_base\_free(base);
59. close(fd);
61. **return** 0;
62. }

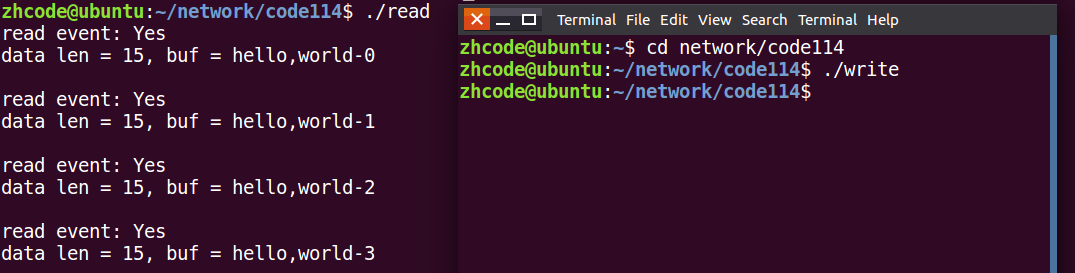
如代码所示，这个也遵循libevent搭积木的过程

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>
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};
16. **static** **int** num = 0;
17. sprintf(buf, "hello,world-%d\n", num++);
18. write(fd, buf, strlen(buf)+1);
20. sleep(1);
21. }

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. }
36. // 写管道
37. **struct** event\_base\* base = NULL;
38. base = event\_base\_new();
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);
46. // 添加事件
47. event\_add(ev, NULL);
49. // 事件循环
50. event\_base\_dispatch(base);
52. // 释放资源
53. event\_free(ev);
54. event\_base\_free(base);
55. close(fd);
57. **return** 0;
58. }

编译运行，结果如下：



## 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

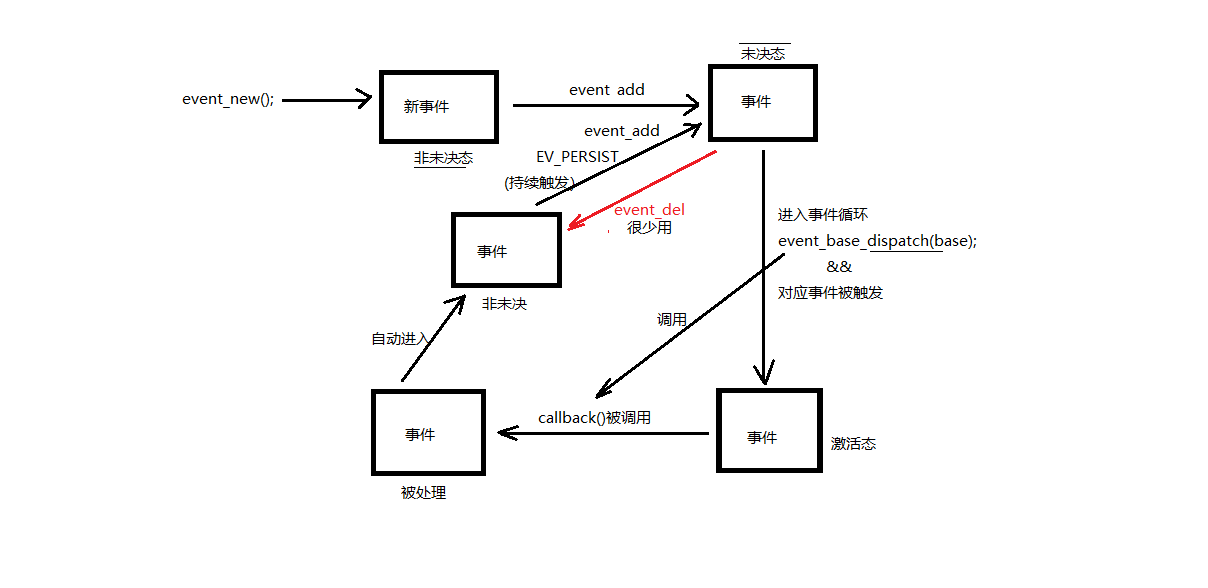
未决和非未决：

非未决: 没有资格被处理

未决： 有资格被处理，但尚未被处理

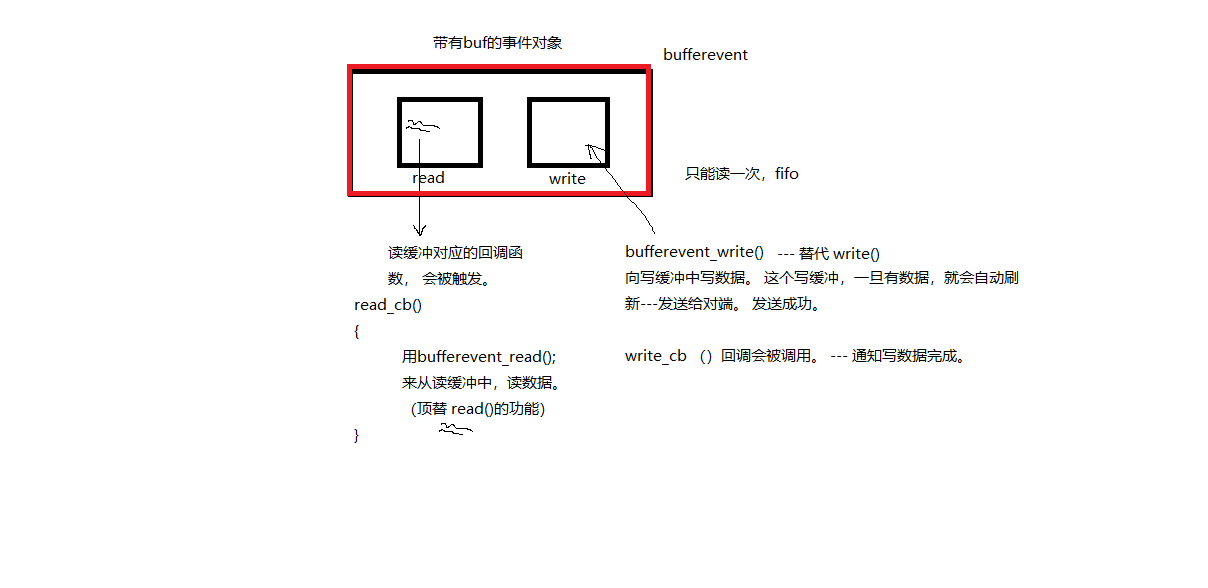
event\_new --> event ---> 非未决 --> event\_add --> 未决 --> dispatch() && 监听事件被触发 --> 激活态

--> 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 \* bufev,

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 bufsize);

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-客户端和服务器连接和监听

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 \* listner

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

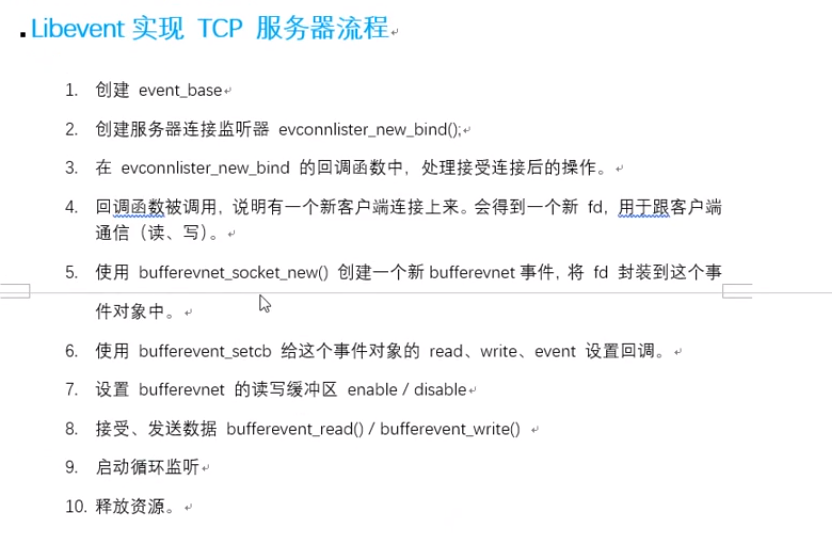
5. 使用 evconnlistener\_new\_bind 创建监听服务器， 设置其回调函数，当有客户端成功连接时，这个回调函数会被调用。

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\_dispath();

8. 释放连接。



## 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>
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);
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. }
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. }
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. }
42. bufferevent\_free(bev);
43. printf("buffevent 资源已经被释放...\n");
44. }


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");
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);
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. }

68. **int** main(**int** argc, **const** **char**\* argv[])
69. {
71. // init server
72. **struct** sockaddr\_in serv;
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);
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));
89. event\_base\_dispatch(base);
91. evconnlistener\_free(listener);
92. event\_base\_free(base);
94. **return** 0;
95. }

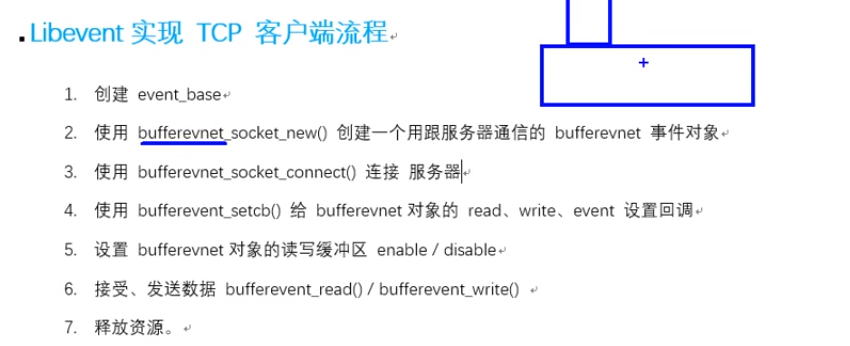
## 128P-服务器注意事项

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>
12. **void** sys\_err(**const** **char** \*str)
13. {
14. perror(str);
15. exit(1);
16. }
18. // 读事件回调
19. **void** read\_cb(**struct** bufferevent \*bev, **void** \*arg)
20. {
21. **char** buf[1024] = {0};
23. // 借助读缓冲，从客户端拿数据
24. bufferevent\_read(bev, buf, **sizeof**(buf));
25. printf("clinet write: %s\n", buf);
27. // 借助写缓冲，写数据回给客户端
28. bufferevent\_write(bev, "abcdefg", 7);
29. }
31. // 写事件回调
32. **void** write\_cb(**struct** bufferevent \*bev, **void** \*arg)
33. {
34. printf("-------fwq------has wrote\n");
35. }
37. // 其他事件回调
38. **void** event\_cb(**struct** bufferevent \*bev,  **short** events, **void** \*ctx)
39. {
41. }
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;
50. // 创建bufferevent 对象
51. **struct** bufferevent \*bev = NULL;
52. bev = bufferevent\_socket\_new(base, sock, BEV\_OPT\_CLOSE\_ON\_FREE);
54. // 给bufferevent 对象 设置回调 read、write、event
55. **void** bufferevent\_setcb(**struct** bufferevent \* bufev,
56. bufferevent\_data\_cb readcb,
57. bufferevent\_data\_cb writecb,
58. bufferevent\_event\_cb eventcb,
59. **void** \*cbarg );
61. // 设置回调函数
62. bufferevent\_setcb(bev, read\_cb, write\_cb, NULL, NULL);
64. // 启动 read 缓冲区的 使能状态
65. bufferevent\_enable(bev, EV\_READ);
67. **return** ;
68. }

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);
80. // 创建event\_base
81. **struct** event\_base \*base = event\_base\_new();
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. \*/
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));
100. // 启动监听循环
101. event\_base\_dispatch(base);
103. // 销毁event\_base
104. evconnlistener\_free(listener);
105. event\_base\_free(base);
107. **return** 0;
108. }

## 129P-客户端流程简析和回顾



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>
11. **void** read\_cb(**struct** bufferevent \*bev, **void** \*arg)
12. {
13. **char** buf[1024] = {0};
14. bufferevent\_read(bev, buf, **sizeof**(buf));
16. printf("fwq say:%s\n", buf);
18. bufferevent\_write(bev, buf, strlen(buf)+1);
19. sleep(1);
20. }
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. }
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. }
43. // 释放资源
44. bufferevent\_free(bev);
45. }
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));
54. **struct** bufferevent\* bev = (**struct** bufferevent\*)arg;
55. // 发送数据
56. bufferevent\_write(bev, buf, len+1);
57. }
59. **int** main(**int** argc, **const** **char**\* argv[])
60. {
61. **struct** event\_base\* base = NULL;
62. base = event\_base\_new();
64. **int** fd = socket(AF\_INET, SOCK\_STREAM, 0);
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);
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);
77. // 连接服务器
78. bufferevent\_socket\_connect(bev, (**struct** sockaddr\*)&serv, **sizeof**(serv));
80. // 设置回调
81. bufferevent\_setcb(bev, read\_cb, write\_cb, event\_cb, NULL);
83. // 设置读回调生效
84. // bufferevent\_enable(bev, EV\_READ);
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);
92. event\_base\_dispatch(base);
94. event\_free(ev);
96. event\_base\_free(base);
98. **return** 0;
99. }

## 130P-总结

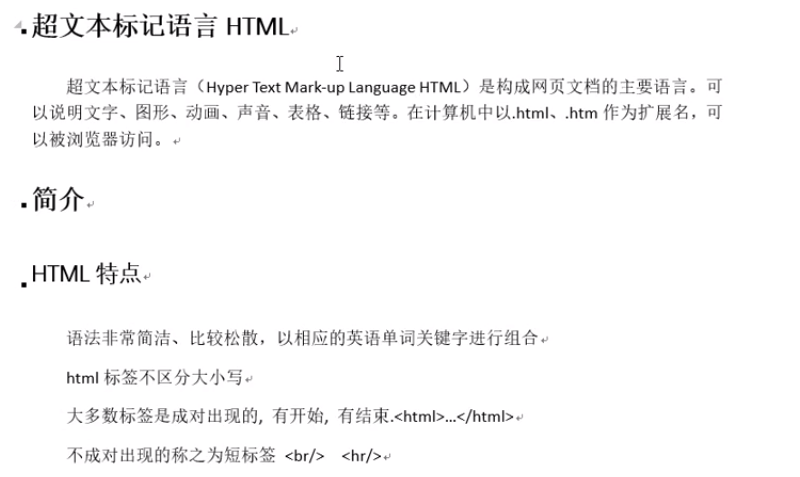
[linux网络编程资料\day6\1-教学资料\课堂笔记.txt](linux网络编程资料/day6/1-教学资料/课堂笔记.txt)

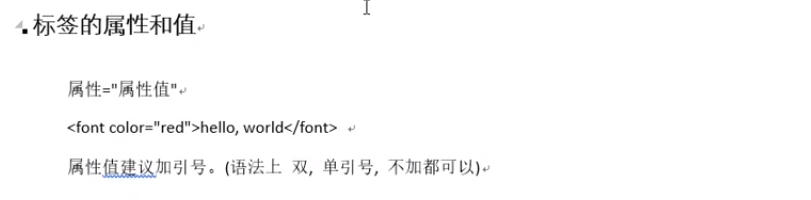
## 131P-复习

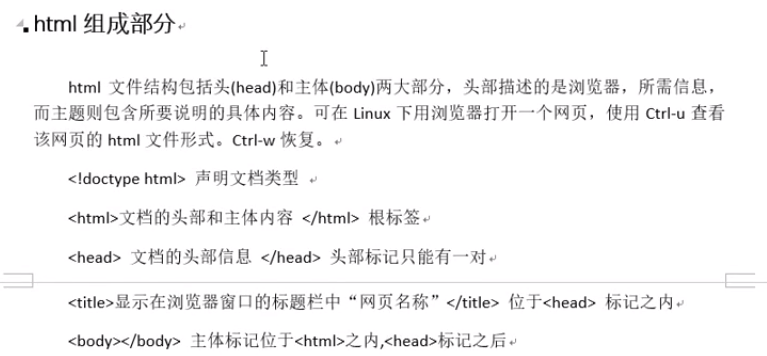
## 132P-web大练习的概述

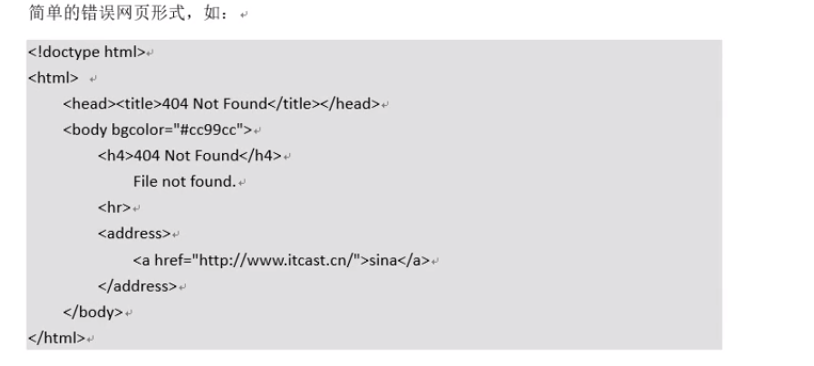
写一个供用户访问主机文件的web服务器

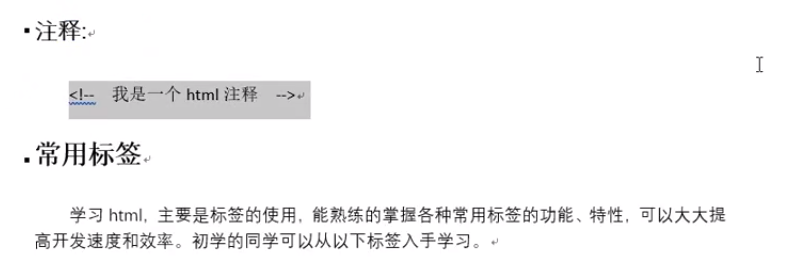
## 133P-HTML文本和标题

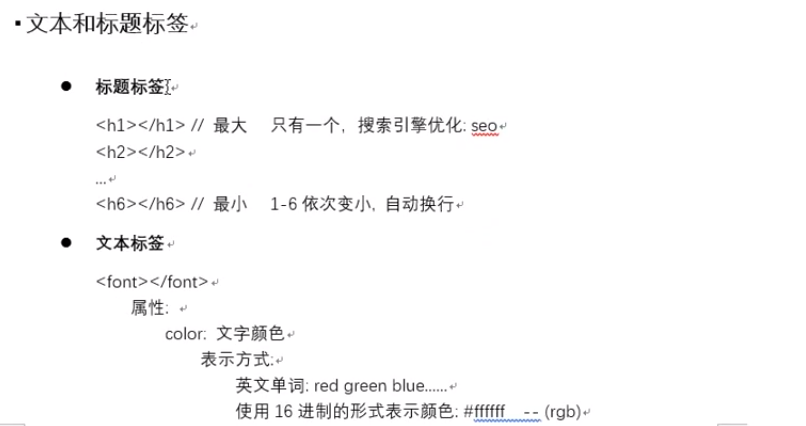


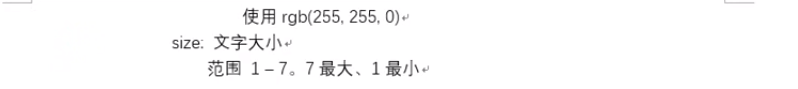


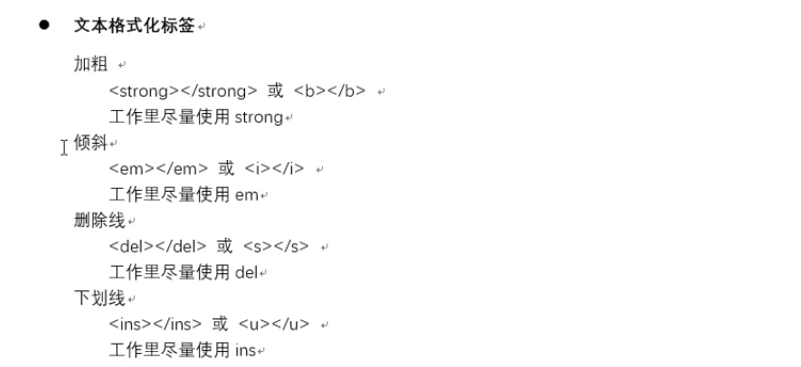


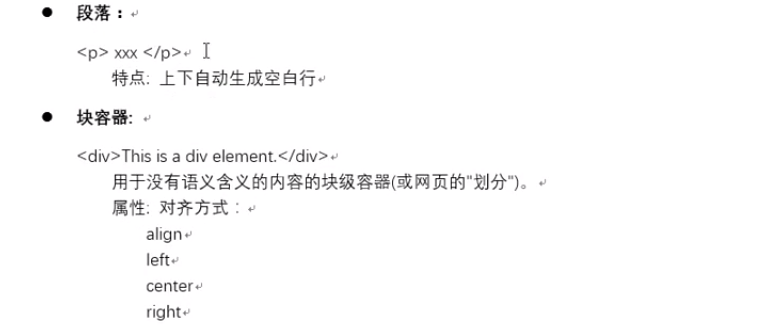


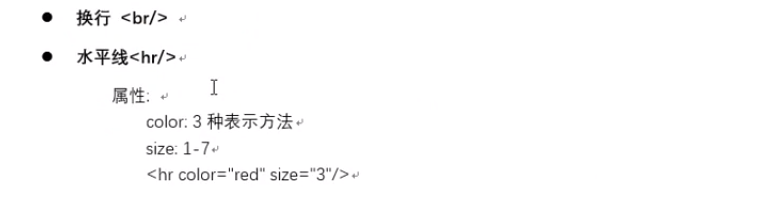










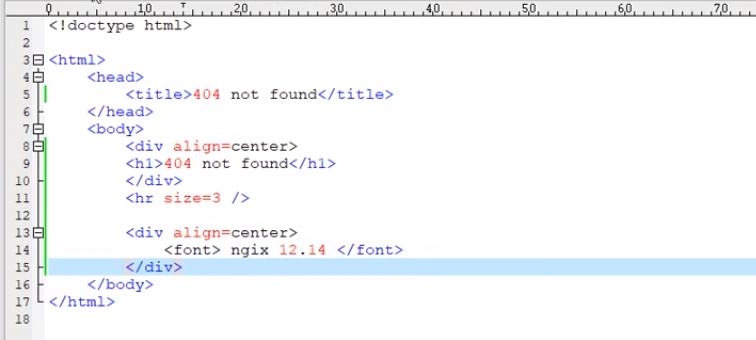


## 134P-HTML文本和标题

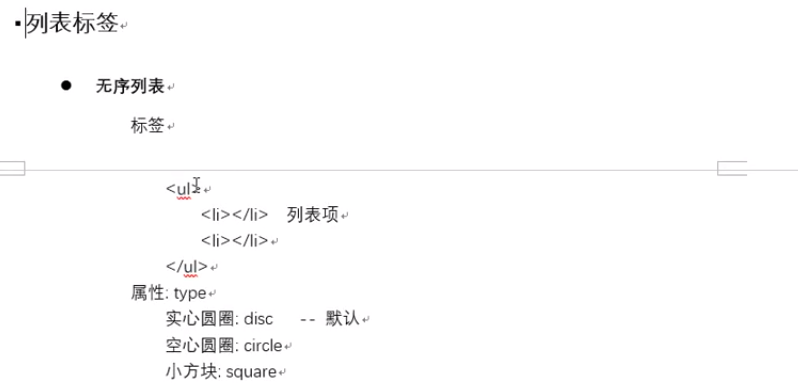
和上一话重复，僵硬，跳过

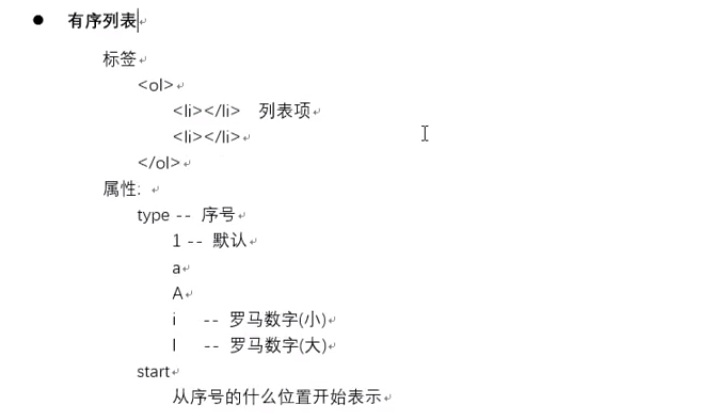
## 135P-错误页面html

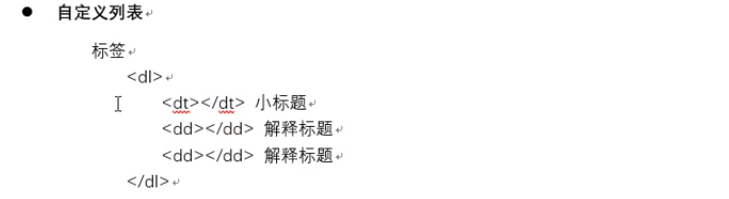
代码比较简单

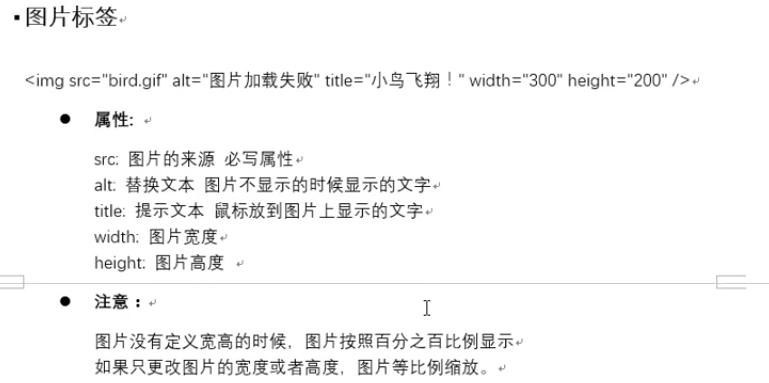


## 136P-列表、图片和超链接

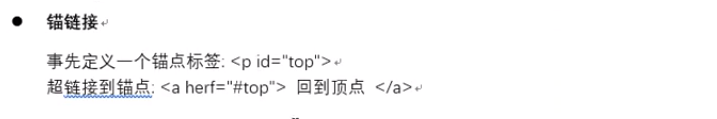




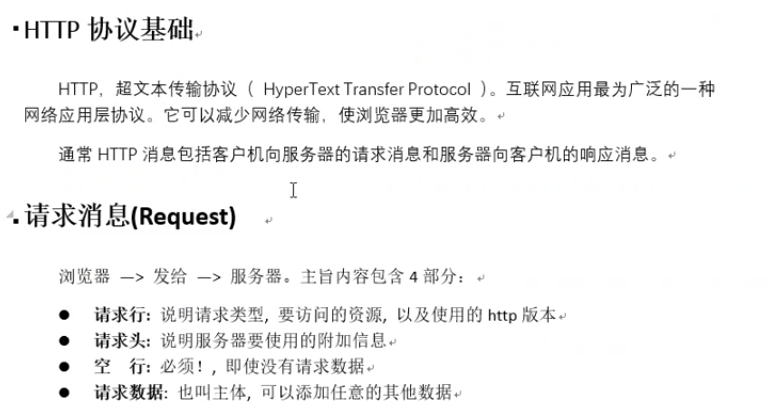




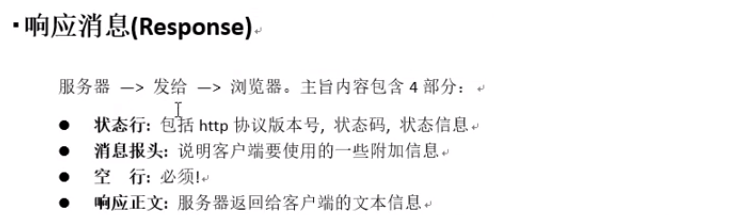




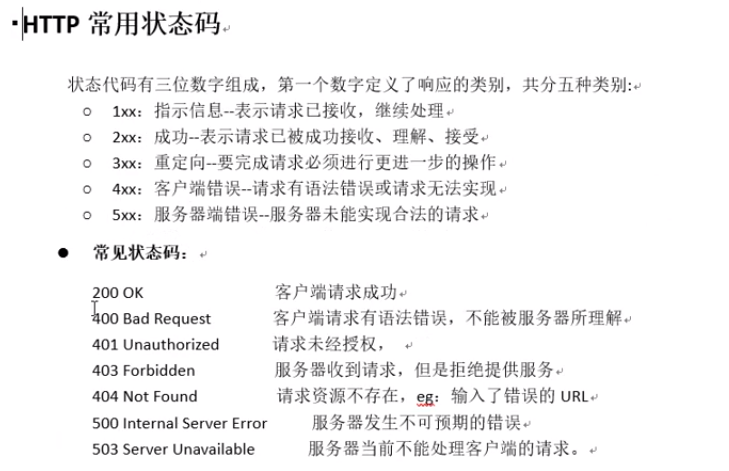
## 137P-http协议请求、应答协议基础格式











## 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>
12. #define MAXSIZE 2048


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;
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);
32. // 端口复用
33. **int** opt = 1;
34. setsockopt(lfd, SOL\_SOCKET, SO\_REUSEADDR, &opt, **sizeof**(opt));
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. }
49. // lfd 添加到 epoll 树上
50. **struct** epoll\_event ev;
51. ev.events = EPOLLIN;
52. ev.data.fd = lfd;
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. }
60. **return** lfd;
61. }
63. **void** do\_accept(**int** lfd, **int** epfd)
64. {
65. **struct** sockaddr\_in clt\_addr;
66. socklen\_t clt\_addr\_len = **sizeof**(clt\_addr);
68. **int** cfd = accept(lfd, (**struct** sockaddr\*)&clt\_addr, &clt\_addr\_len);
69. **if** (cfd == -1) {
70. perror("accept error");
71. exit(1);
72. }
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);
80. // 设置 cfd 非阻塞
81. **int** flag = fcntl(cfd, F\_GETFL);
82. flag |= O\_NONBLOCK;
83. fcntl(cfd, F\_SETFL, flag);
85. // 将新节点cfd 挂到 epoll 监听树上
86. **struct** epoll\_event ev;
87. ev.data.fd = cfd;
89. // 边沿非阻塞模式
90. ev.events = EPOLLIN | EPOLLET;
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. }
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. }
105. **void** epoll\_run(**int** port)
106. {
107. **int** i = 0;
108. **struct** epoll\_event all\_events[MAXSIZE];
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. }
117. // 创建lfd，并添加至监听树
118. **int** lfd = init\_listen\_fd(port, epfd);
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. }
128. **for** (i=0; i<ret; ++i) {
130. // 只处理读事件, 其他事件默认不处理
131. **struct** epoll\_event \*pev = &all\_events[i];
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
139. do\_accept(lfd, epfd);
141. } **else** {                        // 读数据
143. do\_read(pev->data.fd, epfd);
144. }
145. }
146. }
147. }

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. }
158. // 获取用户输入的端口
159. **int** port = atoi(argv[1]);
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. }
168. // 启动 epoll监听
169. epoll\_run(port);
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 01 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

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

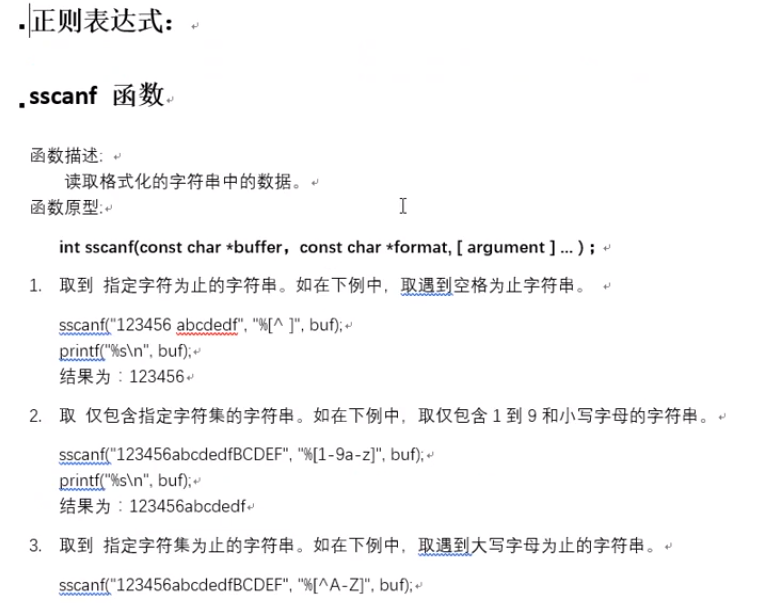
6. 先写 http 应答协议头 ： http/1.1 200 ok

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];
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，检查到客户端关闭....\n");
10. disconnect(cfd, epfd);
11. } **else** {
13. sscanf(line, "%[^ ] %[^ ] %[^ ]", method, path, protocol);
14. printf("method=%s, path=%s, protocol=%s\n", method, path, protocol);
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. }
25. }
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
31. http\_request(cfd, file);
33. disconnect(cfd, epfd);
34. }
35. }





## 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;
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. }
14. **if**(S\_ISREG(sbuf.st\_mode)) {     // 是一个普通文件
16. // 回发 http协议应答
17. //send\_respond(cfd, 200, "OK", " Content-Type: text/plain; charset=iso-8859-1", sbuf.st\_size);
18. send\_respond(cfd, 200, "OK", "Content-Type:image/jpeg", -1);
19. //send\_respond(cfd, 200, "OK", "audio/mpeg", -1);
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};
6. sprintf(buf, "HTTP/1.1 %d %s\r\n", no, disp);
7. send(cfd, buf, strlen(buf), 0);
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);
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};
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. }
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. }
25. close(fd);
26. }

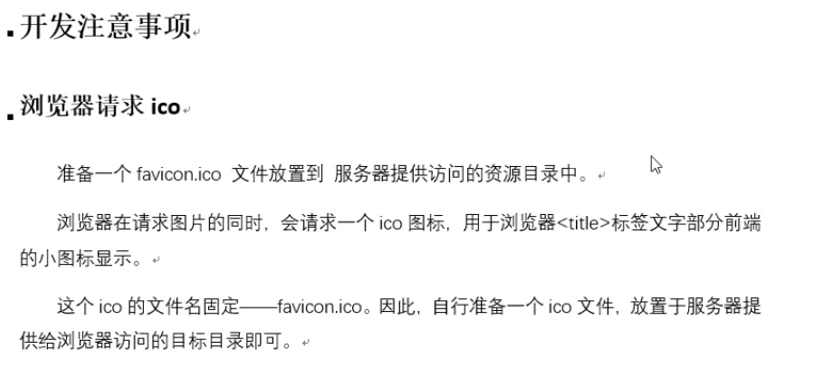
## 146P-文件类型区分

## 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



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};
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);
12. memset(buf, 0, **sizeof**(buf));
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</h4>\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);
20. **return** ;
21. }

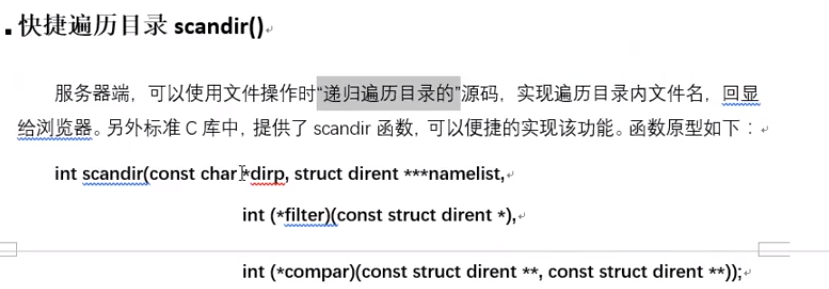
直接看完整代码吧：

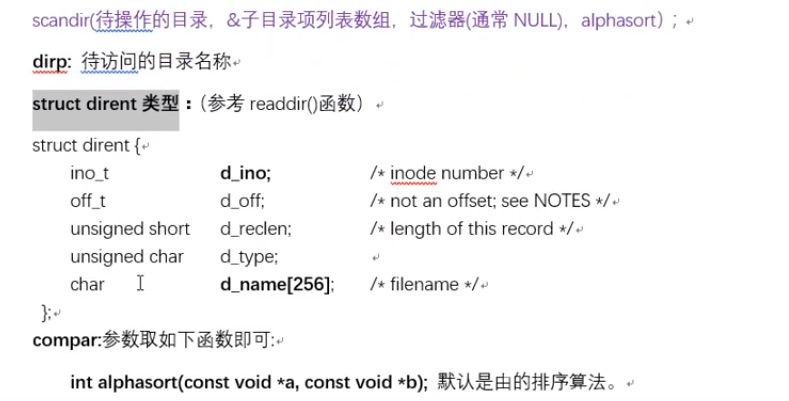
[epoll\_server.c](linux网络编程资料/day8/4-源代码/http-server/epoll-http-server/epoll_server.c)

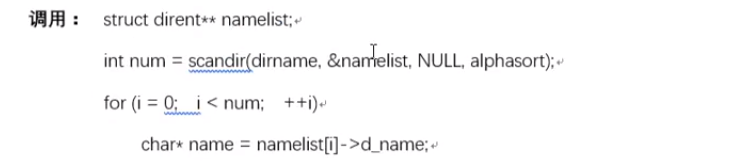
## 149P-关于浏览器请求ico文件

## 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);
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);
13. **char**\* file = path+1; // 去掉path中的/ 获取访问文件名
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. }
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. }
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. }



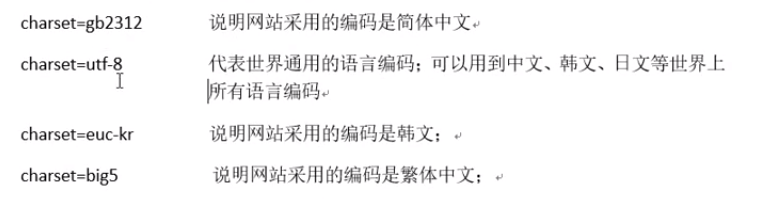




## 151P-判断文件类型

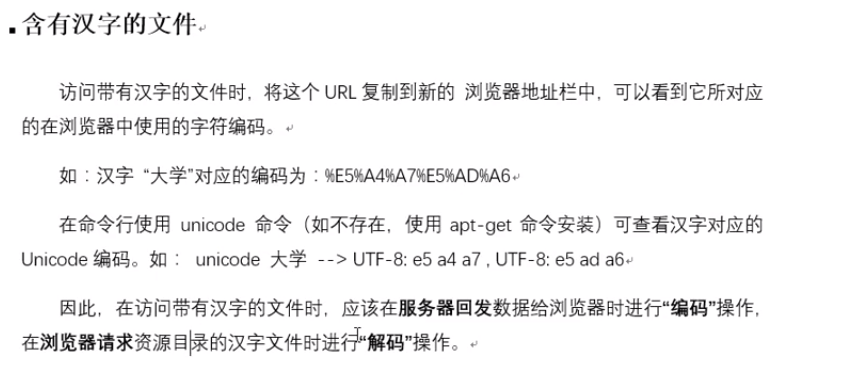
1. // 通过文件名获取文件的类型
2. **const** **char** \*get\_file\_type(**const** **char** \*name)
3. {
4. **char**\* dot;
6. // 自右向左查找‘.’字符, 如不存在返回NULL
7. dot = strrchr(name, '.');
8. **if** (dot == NULL)
9. **return** "text/plain; charset=utf-8";
10. **if** (strcmp(dot, ".html") == 0 || strcmp(dot, ".htm") == 0)
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, ".vrml") == 0 || strcmp(dot, ".wrl") == 0)
31. **return** "model/vrml";
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";
41. **return** "text/plain; charset=utf-8";
42. }





## 152P-汉字字符编码和解码

URL中的汉字默认是存为Unicode码



## 153P-libevent实现的web服务器

直接源码啃起来吧

## 154P-telnet调试

