I/O复用与并行程序

并发与复用

- 回顾之前的程序（无论UDP/TCP）
  - 其一：单个Server，单个Client
  - 其二：单个Server，多个Client
    - 但客户端都是依次被服务器端受理并执行
    - 函数是阻塞的，怎么办

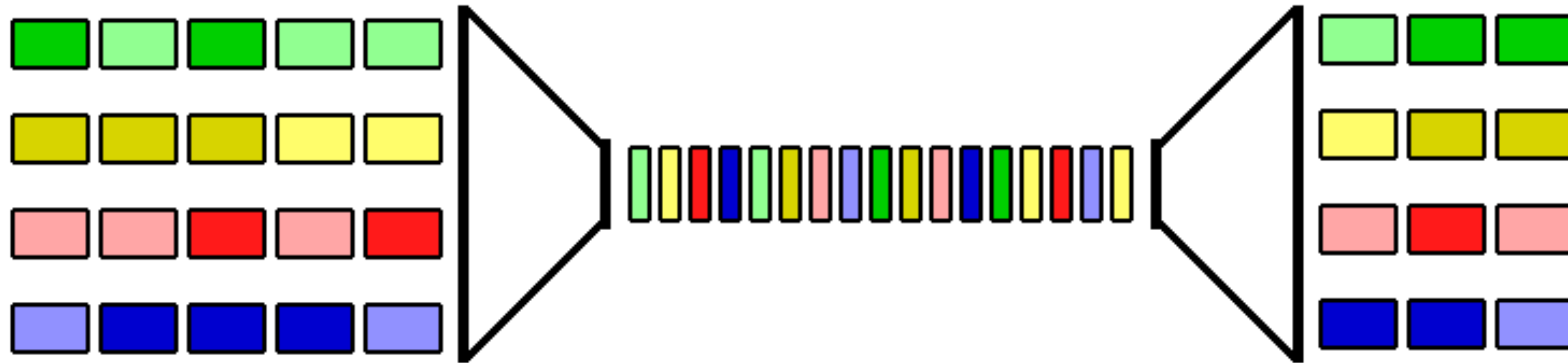
- 两种类型的服务器端
  - 第一种：第一个连接等待受理时间为0s，第50个等待受理时间为50s，第100个连接等待受理时间为100s，依次类推。每个连接服务时间仅1s
  - 第二种：所有连接请求的受理时间不超过1s，平均服务时间越2-3s
- 解决方案——并发服务器
  - 多进程同时处理客户端的连接请求
  - I/O复用的方式轮询客户端的连接请求 [单进程！]

I/O复用-单进程的并发服务器实现

- 用多进程并发服务器的缺点
  - 每次请求都需要创建新的进程，创建、维护进程需要消耗运算和内存空间
  - 进程独立存在，可能涉及数据交换
  - 实现多进程的技术手段较复杂

- 不创建进程而实现并发服务器
  - 同时向多个客户端提供服务，每个客户端不阻塞其它客户端
  - 复用（Multiplexing）技术
    - 复用的思维被用在多个领域

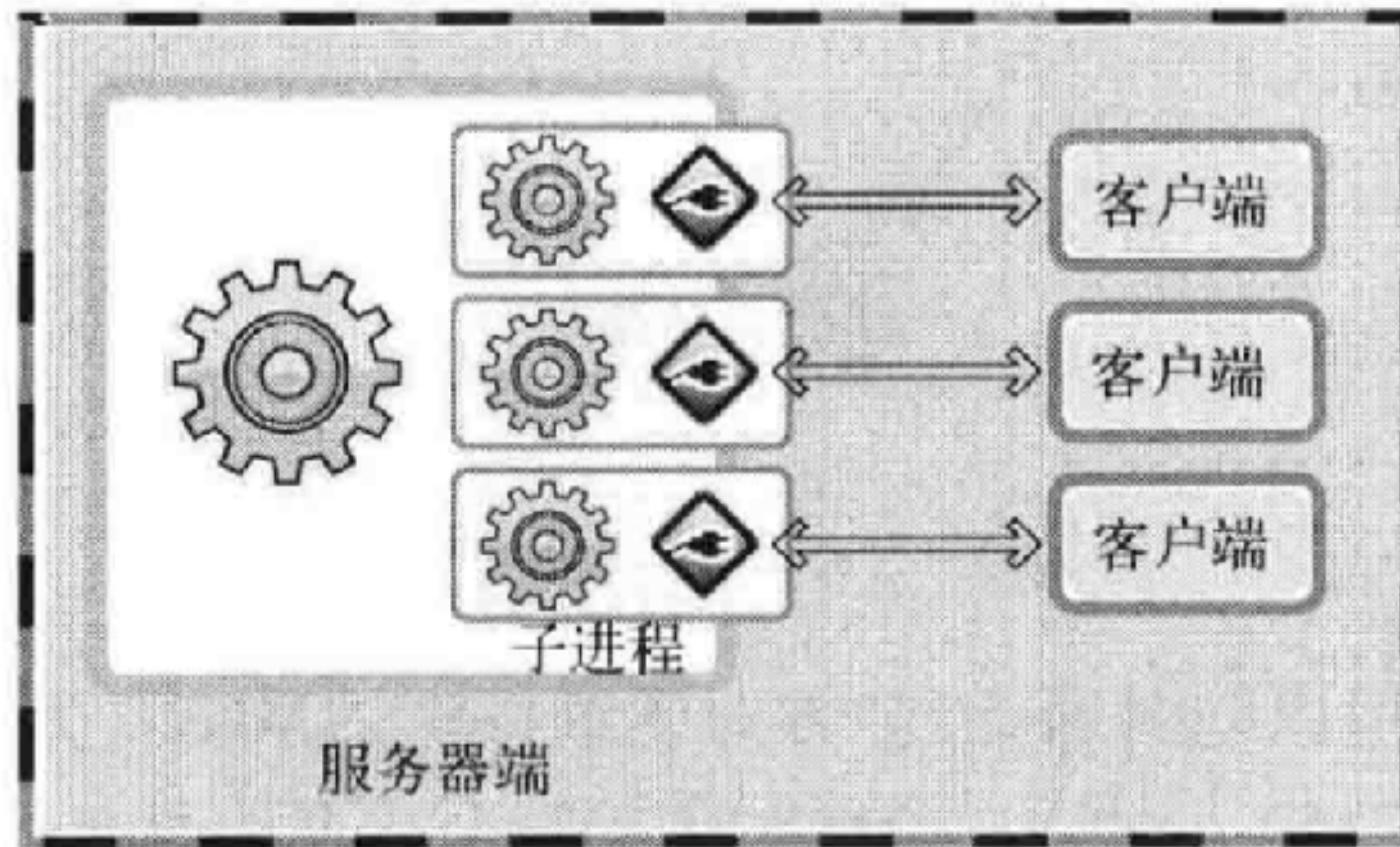
- 不创建进程而实现并发服务器
  - 复用（Multiplexing）技术
    - [通信工程] 在同一个通信信道离传递多个数据
    - [网络] 提高物理设备的效率，用最少的物理要素传递最多的数据



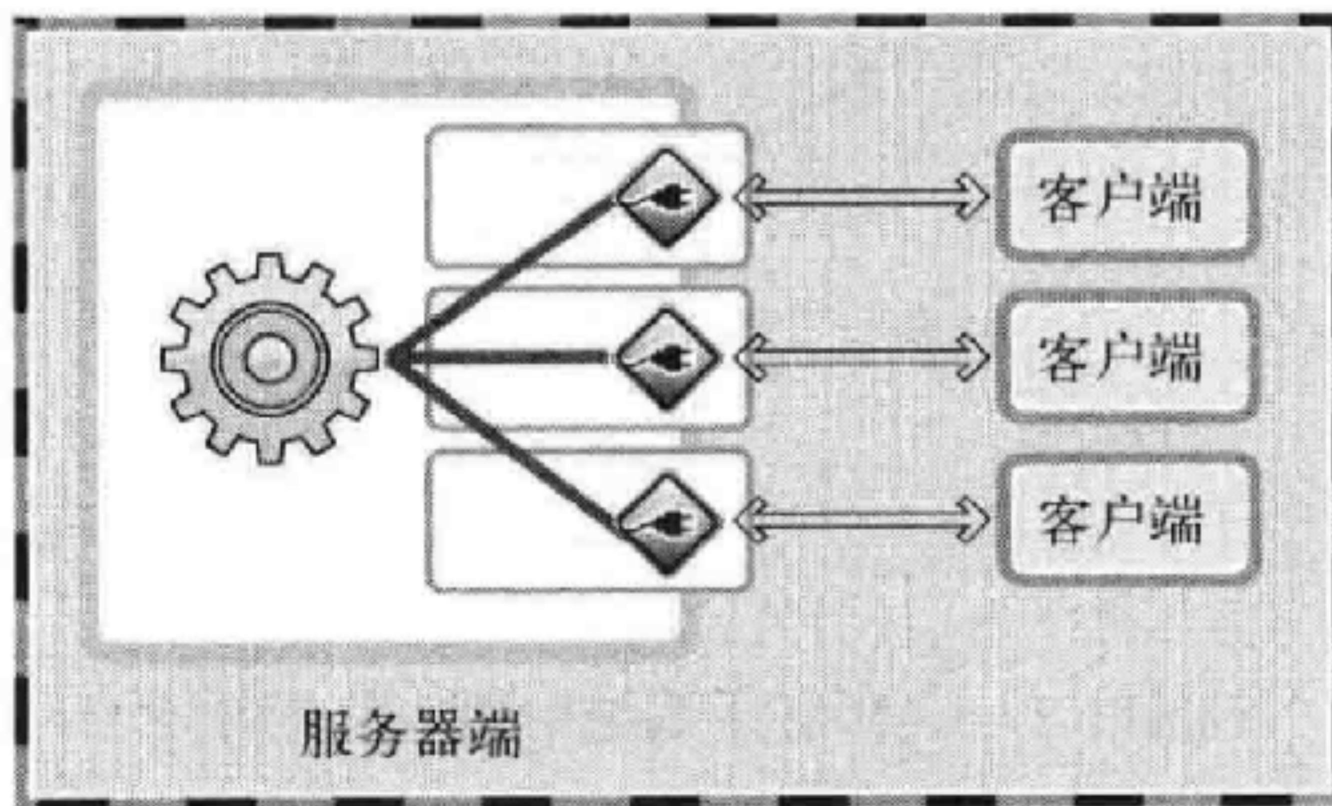
- 通信中的多路复用（复用）模型
  - 时分多路
  - 频分多路
  - 码分多路
  - 空分多路

- “纸杯电话”的复用

Socket服务器上的复用



- 服务器端并发处理多个客户端的连接
- 需要建立多个进程/线程分别处理



- 引入复用技术，对服务器端的应用层程序进行复用
- 实现单个进程处理多个客户端的输入请求
- 我们称之为“I/O复用”

- 具体来说，服务器端的I/O复用即：
  - 服务器的应用监测指定的客户端
  - 若某客户端有请求产生，则处理
  - 若客户端无请求产生，则略过

I/O复用的一个类比

- 上课回答问题
  - 方式一：给每个学生配备一个老师，一对一，一旦某学生有问题老师能马上处理，但是该学生没有问题的时候老师必须等待
  - 方式二：一个~~聪明的~~老师，多个学生，~~聪明的~~老师周期性地扫视整个课堂，有举手的学生则点他起来处理问题，没有的话就继续保持观测

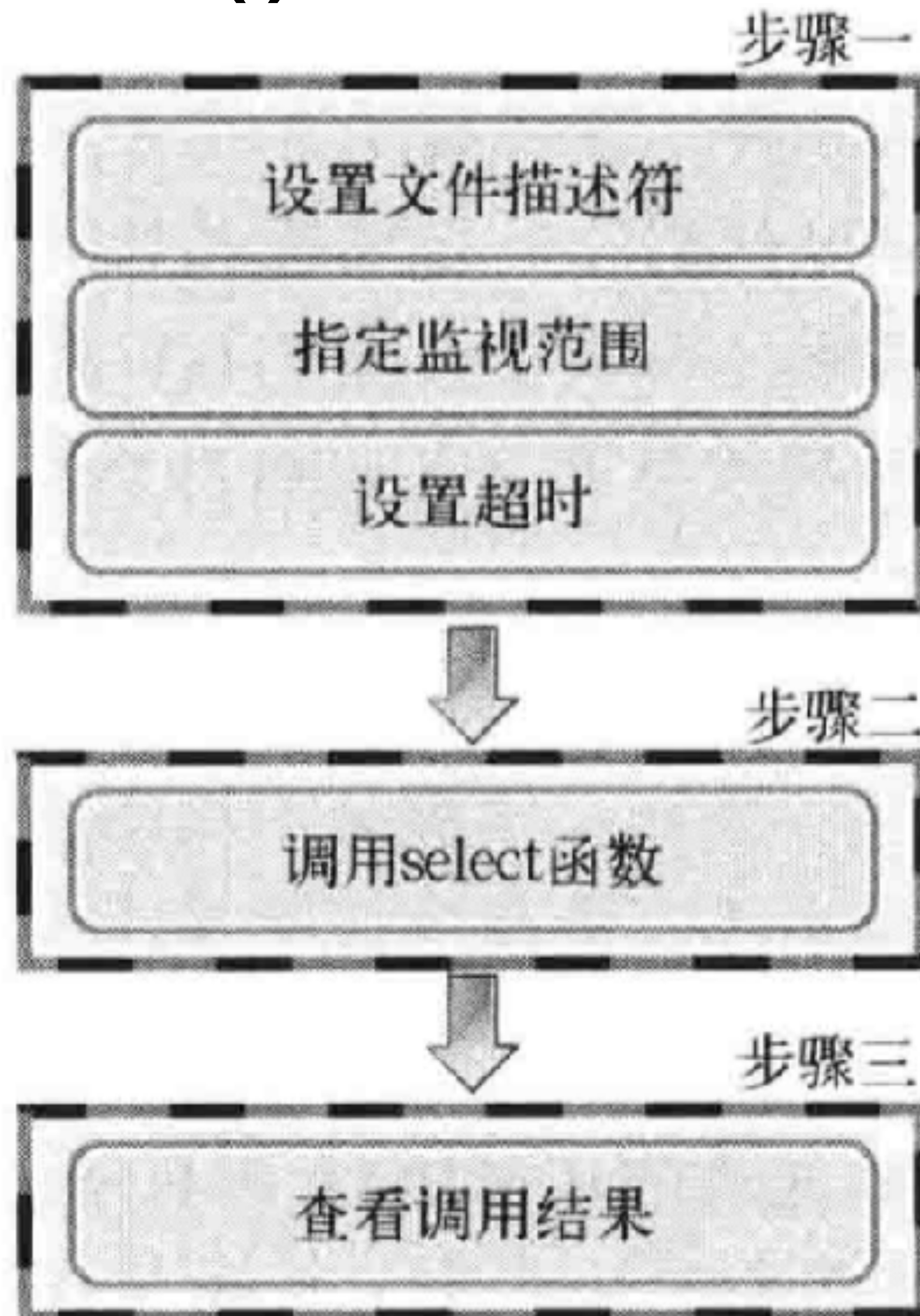
select()函数

- 最具有代表性的实现复用服务器的方法
- 常被程序员忽略的方法
- winsock和BSD socket等等都有该方法（或其扩展）

select()函数主要功能

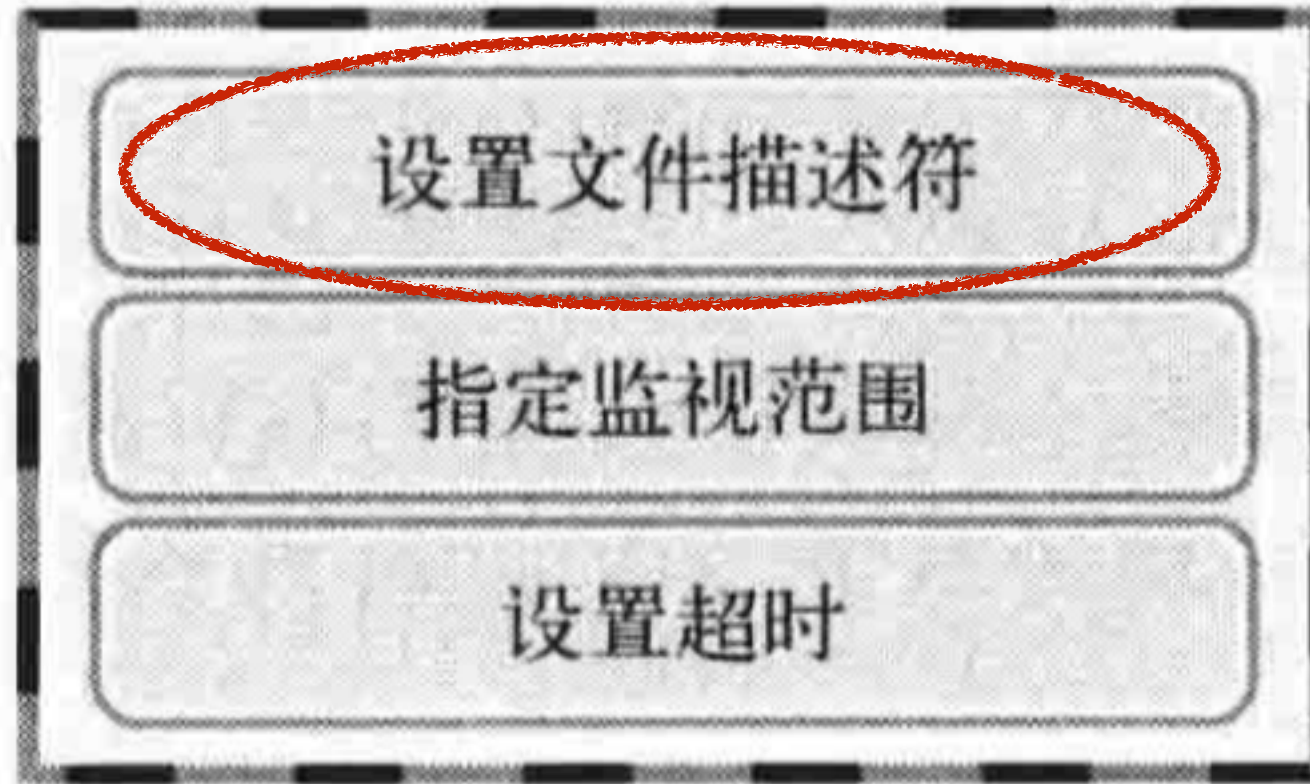
- 将多个文件描述符集中在一起统一监视
  - 是否存在套接字有待接收数据?
  - 是否存在套接字有（无阻塞的）待传输数据
  - 是否存在套接字发生了异常

select()函数的用法

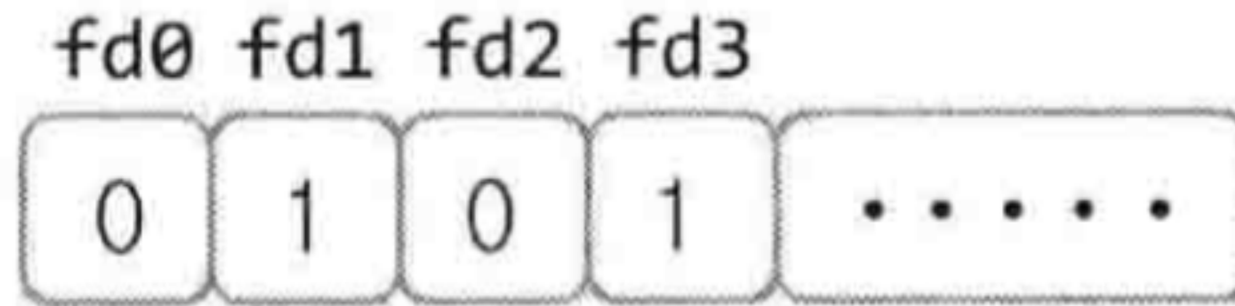


select()函数用法-步骤一

- 设置文件描述符
  - select()可以同时监视多个文件描述符
    - 注意文件描述符可以是stdio、文件、设备、socket等等
  - 文件描述符被集中监视
    - 维护一张表，表示同类的文件描述符
  - 监视项也是分类的
    - 待接收、待传输、发生异常



- 监视文件描述符的表（fd_set)
- 如何定义与初始化



- 设置一个文件描述符的表fd_set，结构如图
  - 一个数组，每一位表示系统中的一个文件描述符，如图第一位表示fd0，第三位表示fd2……
  - 每位的元素可为0/1
    - 若为0，表示select不监视此fd
    - 若为1，……

- 如何改变 ：

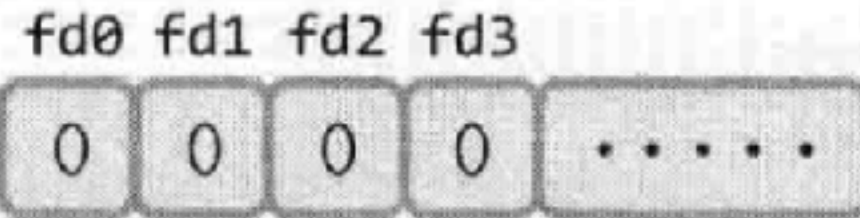
fd0 fd1 fd2 fd3

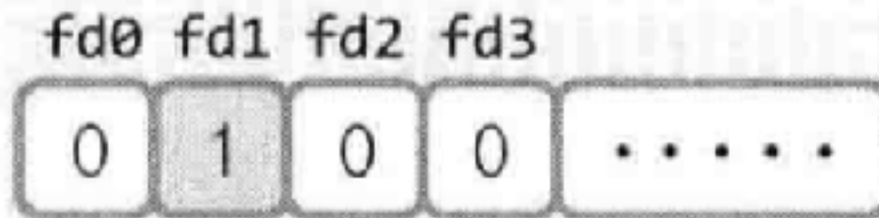
0	1	0	1	·····

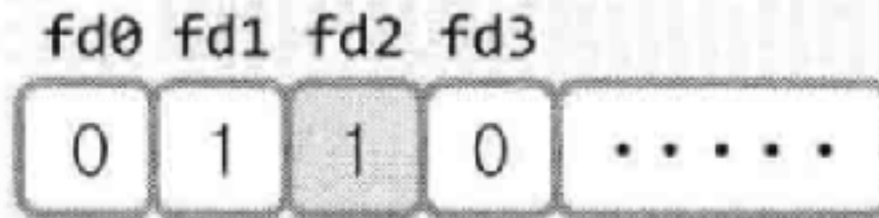
fd_set set

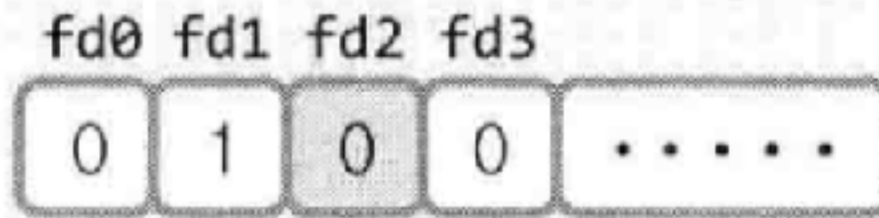
- 利用宏
  - FD_ZERO(fd_set *);
  - FD_SET(int, fd_set *);
  - FD_CLR(int, fd_set *);
  - FD_ISSET(int, fd_set *);

```
int main(void)
{

    fd_set  set;

    FD_ZERO(&set);

    FD_SET(1, &set);

    FD_SET(2, &set);

    FD_CLR(2, &set);

}
```









select()函数用法-步骤二

- 设置监视的范围和超时时间（timeout）
  - select函数监视不同类型的几组文件描述符（待接收、待传输、发生异常）
  - select函数监视时也发生阻塞
    - 通过设置timeout value来跳出阻塞

- select函数结构
- int select(

```
int              maxfd,
fd_set  *  readset,
fd_set  *  writeset,
fd_set  *  exceptset,
const struct timeval  *  timeout
);
```

- 因此，使用select函数最关键的两个输入参数包括
  - 文件描述符（fd）的监视范围（即我们要监视的客户端的集合）
  - 超时的时间
    - 通过struct timeval结构体来初始化并输入

- struct timeval

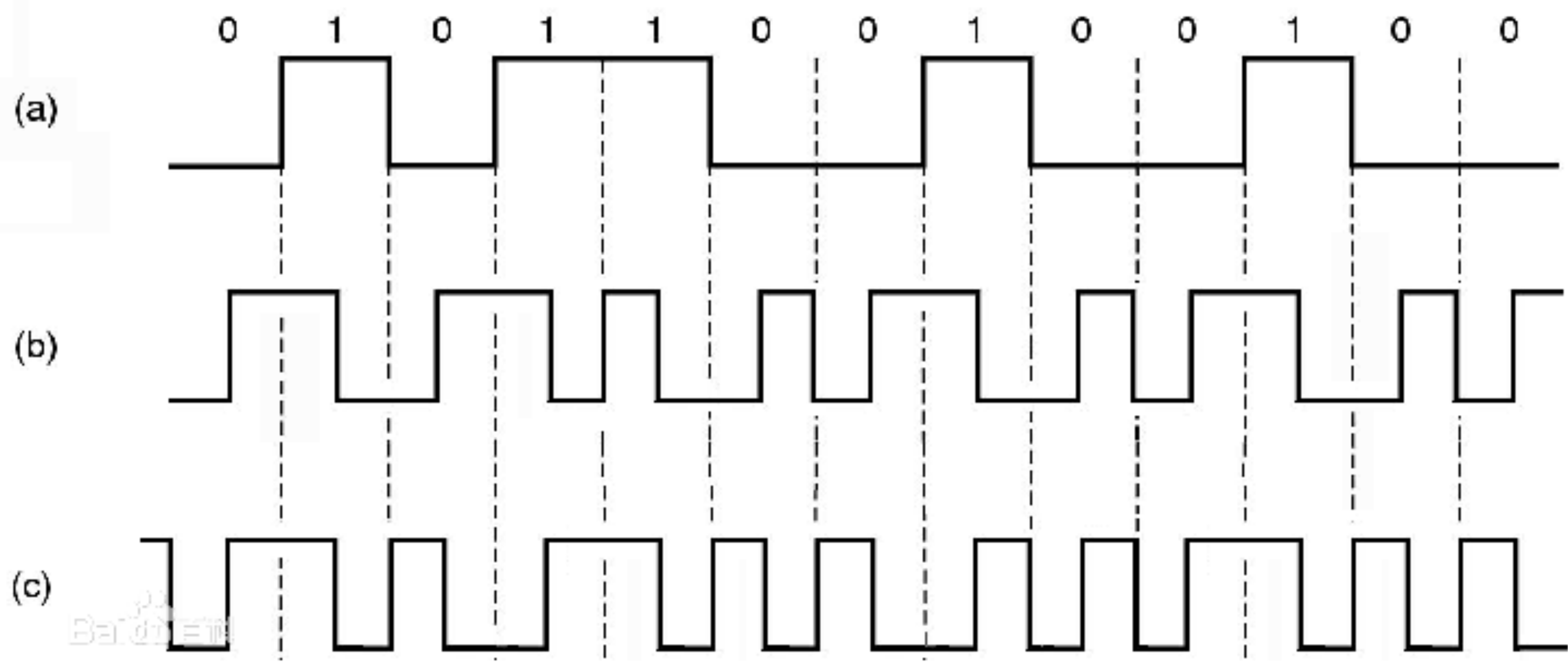
```
{
    long tv_sec;         //sec
    long tv_usec;        //microsec
}
```

- select监视fd，如果fd们不发生变化则阻塞
- 过了指定时间timeval后，如果fd依然无变化，函数返回0

- 执行select时， struct timeval * timeout的变化
  - 为什么timeout是指针
  - timeout会在select执行时改变

select()函数用法-步骤三

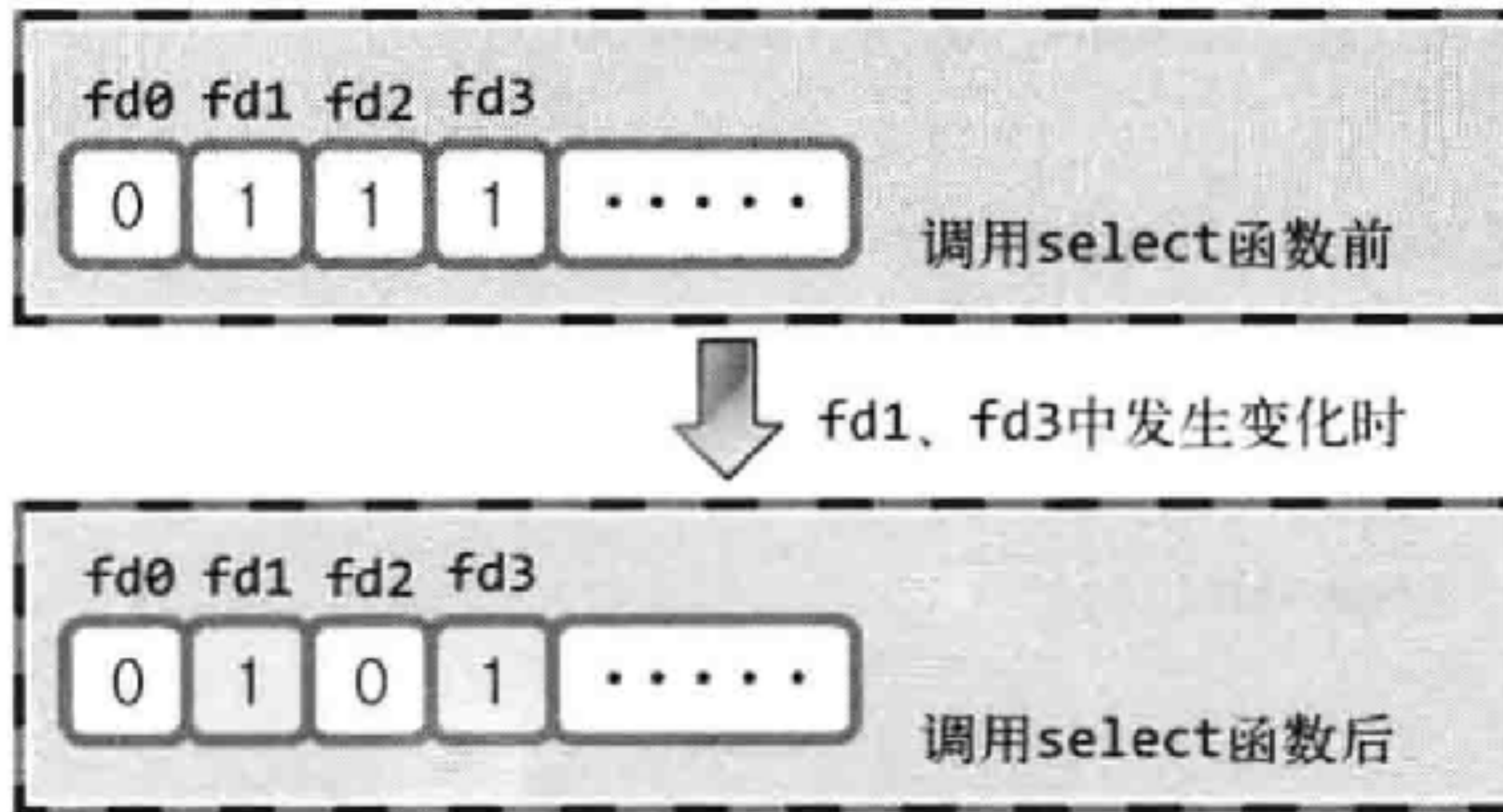
- 查看select的结果
  - 如果select函数返回大于0的数，则说明部分监视的fd发生了变化
    - 例如：经过查询，有些客户端套接字产生了输入
  - 此时应检查三张fd_set表有什么变化
    - 若没有产生变化，原来为1的位被置为0



不归零码

曼彻斯特

差分曼彻斯特



- 返回结果后带来的困难?
- 下一次select前需要重置fd_set们

select函数-代码分析

❖ select.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <sys/time.h>
4.  #include <sys/select.h>
5.  #define BUF_SIZE 30
6.
7.  int main(int argc, char *argv[])
8.  {
9.      fd_set reads, temps;
10.     int result, str_len;
11.     char buf[BUF_SIZE];
12.     struct timeval timeout;
13.
```

```
14.     FD_ZERO(&reads);
15.     FD_SET(0, &reads); // 0 is standard input(console)
16.
17.     /*
18.     timeout.tv_sec=5;
19.     timeout.tv_usec=5000;
20.     */
21.
```

- 此处设置timeout被注释掉了
- 因为timeout在select过程中会改变，在此处赋值不科学

```
22.     while(1)
23.     {
24.         temps=reads;
25.         timeout.tv_sec=5;
26.         timeout.tv_usec=0;
27.         result=select(1, &temps, 0, 0, &timeout);
28.         if(result==-1)
29.         {
30.             puts("select() error!");
31.             break;
32.         }
33.         else if(result==0)
34.         {
35.             puts("Time-out!");
36.         }
37.         else
38.         {
39.             if(FD_ISSET(0, &temps))
40.             {
41.                 str_len=read(0, buf, BUF_SIZE);
42.                 buf[str_len]=0;
43.                 printf("message from console: %s", buf);
44.             }
45.         }
46.     }
47.     return 0;
```

注意传入select的
是变量temps而不是reads

检查返回值也是检查
变量temps而不是reads

回声服务器-再讨论

- 之前讨论的回声（echo）服务器
- “回声”（echo）服务器，满足如下（比较简化的）设计要求：
  - 服务器端同一时刻只处理一个客户端请求
  - 服务器端依次向五个客户端提供服务，然后退出
  - 客户端发送文本给服务器，服务器回传（echo back）
  - 客户端输入退出指令，结束一个客户端的请求

```
42.     for(i=0; i<5; i++)
43.     {
44.         clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
45.         if(clnt_sock==-1)
46.             error_handling("accept() error");
47.         else
48.             printf("Connected client %d \n", i+1);
49.
50.         while((str_len=read(clnt_sock, message, BUF_SIZE))!=0)
51.             write(clnt_sock, message, str_len);
52.
53.         close(clnt_sock);
54.     }
55.     close(serv_sock);
56.     return 0;
57. }
```

非I/O复用的回声服务器

基于I/O复用的回声服务器

❖ echo_selectserv.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.  #include <sys/time.h>
8.  #include <sys/select.h>
9.
10. #define BUF_SIZE 100
11. void error_handling(char *buf);
12.
13. int main(int argc, char *argv[])
14. {
15.     int serv_sock, clnt_sock;
16.     struct sockaddr_in serv_adr, clnt_adr;
17.     struct timeval timeout;
18.     fd_set reads, cpy_reads;
19.
20.     socklen_t adr_sz;
```

Server

```
21.     int fd_max, str_len, fd_num, i;
22.     char buf[BUF_SIZE];
23.     if(argc!=2) {
24.         printf("Usage : %s <port>\n", argv[0]);
25.         exit(1);
26.     }
27.
28.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
29.     memset(&serv_adr, 0, sizeof(serv_adr));
30.     serv_adr.sin_family=AF_INET;
31.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
32.     serv_adr.sin_port=htons(atoi(argv[1]));
33.
34.     if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
35.         error_handling("bind() error");
36.     if(listen(serv_sock, 5)==-1)
37.         error_handling("listen() error");
38.
39.     FD_ZERO(&reads);
40.     FD_SET(serv_sock, &reads);
41.     fd_max=serv_sock;
```

Server

```
43.     while(1)
44.     {
45.         cpy_reads=reads;
46.         timeout.tv_sec=5;
47.         timeout.tv_usec=5000;
48.
49.         if((fd_num=select(fd_max+1, &cpy_reads, 0, 0, &timeout))==-1)
50.             break;
51.         if(fd_num==0)
52.             continue;
```

Server

接while循环体……

```
53.
54.         for(i=0; i<fd_max+1; i++)
55.         {
56.             if(FD_ISSET(i, &cpy_reads))
57.             {
58.                 if(i==serv_sock)    // connection request!
59.                 {
60.                     adr_sz=sizeof(clnt_adr);
61.                     clnt_sock=
62.                         accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
63.                     FD_SET(clnt_sock, &reads);
64.                     if(fd_max<clnt_sock)
65.                         fd_max=clnt_sock;
66.                     printf("connected client: %d \n", clnt_sock);
67.                 }
68.                 else    // read message!
69.                 {
70.                     str_len=read(i, buf, BUF_SIZE);
71.                     if(str_len==0)   // close request!
72.                     {
73.                         FD_CLR(i, &reads);
74.                         close(i);
75.                         printf("closed client: %d \n", i);
76.                     }
77.                     else
78.                     {
79.                         write(i, buf, str_len); // echo!
80.                     }
81.                 }
82.             }
83.         }
84.     }
85.     close(serv_sock);
86.     return 0;
87. }
```

Server

客户端不需要做出变化

总结：select模型

- I/O多路复用（I/O Multiplexing）
  - 操作系统赋予的一个功能，使得socket（或称fd）产生变化时，能返回一个“通知”
  - 适用于非阻塞模式的socket适时地进行操作，保证每次操作都能“有明确效果”
  - 监视多个socket（或称fd）
  - 在同一个线程完成

- socket编程中select产生监听事件的情况
  - 可读（readset）：
    - socket内核接收缓冲区的内容足够多
    - socket通信对方关闭连接
    - socket上有新的连接请求
    - socket上有未处理的错误

- socket编程中select产生监听事件的情况
  - 可写（writeset）：
    - socket内核发送缓冲区的内容足够多
    - socket的读端关闭
    - socket使用connect连接成功
    - socket上有未处理的错误

select的优点

- 兼容性非常好
  - 比较“传统”的I/O多路复用的方法
  - 多个操作系统都支持select函数
  - 实现逻辑简单，适用于服务器端接入少的情况

select的缺点

- select遇到的问题
  - 实践中，接入数百个客户端时性能降低直至失效
  - 部分操作系统对select的最大监视容量有限制
  - 从原理上会产生性能瓶颈

- select函数的缺点分析
- int select(…, fd_set *readset, fd_set *writeset, fd_set *exceptset, …);
  - 每次调用select函数，都必须将监视对象（fd）的信息全部进行参数传递
  - 由于select会修改各fd_set，因此每次调用select都需要复制各fd_set
  - 需要对fd_set监视的所有对象进行循环轮询

select之外的I/O复用

- 如何克服select的缺点
  - 减少传递监视对象表的次数
  - 监视对象的内容发生变化时才产生通知
- 因I/O复用是操作系统层面实现，因此需要操作系统的函数支持
- linux下可以使用epoll

epoll方法

- epoll的优点正好弥补了select的不足
  - epoll不需要编写循环语句用于轮询所有文件描述符的状态变化
    - 一旦有变化会返回变化的文件描述符
  - 调用epoll系列函数中查询fd变化情况的函数时，不需要每次都向操作系统传递信息

- epoll方法的三个函数
    - epoll_create：创建保存epoll文件描述符的空间
    - epoll_ctl：向空间注册或注销文件描述符
    - epoll_wait：等待文件描述符fd发生变化并返回变化的fd的集合
- epoll工作流程
    - 创建epoll、注册监视的fd、等待发生变化的fd消息

- 保存监视的文件描述符fd
  - select怎么实现的？fd_set结构的变量
  - epoll保存在操作系统中
    - 具体来说，向操作系统请求创建保存fd的空间（使用epoll_create）
    - 保存监视的fd信息，以及返回发生事件变化的fd信息，采用epoll_event结构体

- epoll_event

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

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

epoll工作过程简介

- epoll_create函数
  - Linux 2.5.44以上版本
  - #include <sys/epoll.h>
  - int epoll_create(int size)
    - 用epoll_create在操作系统中创建的fd保存空间叫做“epoll例程”
    - size决定epoll例程大小，但操作系统仅仅只参考！

- epoll_ctl函数
  - int epoll_ctl(

        int   epfd,

        int   op,

        int   fd,

        struct epoll_event * event

    );
  - 成功时返回0，失败返回-1

- epoll_ctl函数的例子
  - epoll_ctl(A, EPOLL_CTL_ADD, FD, EVENT)
    - 向操作系统注册文件描述符FD，主要用于监测EVENT事件
  - epoll_ctl(A, EPOLL_CTL_DEL, FD, NULL)
    - 从操作系统删除文件描述符FD

- epoll_ctl函数之event参数
  - event参数是一个epoll_event结构
  - 监视的事件类型（读取数据，发送数据，...）
  - 与第三个参数有部分内容重叠，见示例

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

标记的部分在第三项参数出现过

- epoll_wait函数
  - int epoll_wait(

```
    int     epfd,
    struct epoll_event  *  events,
    int     maxevents,
    int     timeout
);
```

  - 成功时返回发生事件的fd数量，失败返回-1

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

events参数需要动态分配

基于epoll的回声服务器端-代码

❖ echo_epollserv.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.  #include <sys/epoll.h>
8.
9.  #define BUF_SIZE 100
10. #define EPOLL_SIZE 50
11. void error_handling(char *buf);
```

```
13. int main(int argc, char *argv[])
14. {
15.         int serv_sock, clnt_sock;
16.         struct sockaddr_in serv_adr, clnt_adr;
17.         socklen_t adr_sz;
18.         int str_len, i;
19.         char buf[BUF_SIZE];
20.
21.         struct epoll_event *ep_events;
22.         struct epoll_event event;
23.         int epfd, event_cnt;
24.
25.         if(argc!=2) {
26.             printf("Usage : %s <port>\n", argv[0]);
27.             exit(1);
```

注意区别

```
30.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
31.     memset(&serv_adr, 0, sizeof(serv_adr));
32.     serv_adr.sin_family=AF_INET;
33.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
34.     serv_adr.sin_port=htons(atoi(argv[1]));
35.
36.     if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
37.         error_handling("bind() error");
38.     if(listen(serv_sock, 5)==-1)
39.         error_handling("listen() error");
```

epoll第一步

```
41.     epfd=epoll_create(EPOLL_SIZE);
42.     ep_events=malloc(sizeof(struct epoll_event)*EPOLL_SIZE);
43.
44.     event.events=EPOLLIN;
45.     event.data.fd=serv_sock;
46.     epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);
```

epoll第二步

注意event的用法

```
48.     while(1)
49.     {
50.         event_cnt=epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
51.         if(event_cnt==-1)
52.         {
53.             puts("epoll_wait() error");
54.             break;
55.         }
```

处理serv_sock这个fd的事件

```
57.         for(i=0; i<event_cnt; i++)
58.         {
59.             if(ep_events[i].data.fd==serv_sock)
60.             {
61.                 adr_sz=sizeof(clnt_adr);
62.                 clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
63.                 event.events=EPOLLIN;
64.                 event.data.fd=clnt_sock;
65.                 epoll_ctl(epfd, EPOLL_CTL_ADD, clnt_sock, &event);
66.                 printf("connected client: %d \n", clnt_sock);
67.             }
68.             else
69.             {
70.                     str_len=read(ep_events[i].data.fd, buf, BUF_SIZE);
71.                     if(str_len==0)  // close request!
72.                     {
73.                         epoll_ctl(
74.                             epfd, EPOLL_CTL_DEL, ep_events[i].data.fd, NULL);
75.                         close(ep_events[i].data.fd);
76.                         printf("closed client: %d \n", ep_events[i].data.fd);
77.                     }
78.                     else
79.                     {
80.                         write(ep_events[i].data.fd, buf, str_len);    // echo!
81.                     }
```

处理clnt_sock这个fd的事件

- 最后记得要关闭创建的fd们
  - 关闭socket（服务器和客户端）
  - 关闭epoll例程

```
86.     close(serv_sock);
87.     close(epfd);
88.     return 0;
```

多进程网络程序设计

- 两种类型的服务器端
  - 第一种：第一个连接等待受理时间为0s，第50个等待受理时间为50s，第100个连接等待受理时间为100s，依次类推。每个连接服务时间仅1s
  - 第二种：所有连接请求的受理时间不超过1s，平均服务时间越2-3s
- 解决方案——并发服务器
  - 多进程同时处理客户端的连接请求
  - I/O复用的方式轮询客户端的连接请求 [单进程！]

进程和线程-基本概念

- 进程是表示资源分配的基本单位，又是调度运行的基本单位
  - 例如，用户运行自己的程序，系统就创建一个进程，并为它分配系统资源
  - 该进程放人进程的就绪队列。进程调度程序选中它，为它分配CPU以及其它有关资源，该进程才真正运行
- 进程是系统中的并发执行的单位

yangzhang — -bash — 119×24

```
Yangs-MacBook-Pro:~ yangzhang$ ps aux
USER              PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
yangzhang        3025   7.8  2.6  3824784 219416   ??  S    六02下午  74:00.93 /Applications/QQ.app/Contents/MacOS/QQ
yangzhang         298   0.8  0.3  3368736  24016   ??  S    六02下午   8:58.83 /System/Library/CoreServices/Dock.app/
yangzhang         302   0.7  0.3  2686540  21016   ??  S    六02下午   2:50.71 /System/Library/CoreServices/SystemUIS
_windowserver     172   0.6  1.4  7108544 115380   ??  Ss   六02下午  70:40.81 /System/Library/PrivateFrameworks/SkyL
yangzhang       48142   0.1  0.6  2727180  48228   ??  S     7:29下午   0:05.58 /Applications/Utilities/Terminal.app/C
yangzhang       43443   0.1  1.6  3387064 136056   ??  S     5:55下午   0:40.00 /Applications/Nutstore.app/Contents/Re
yangzhang       51567   0.0  0.2  2491236  19052   ??  S     8:38下午   0:00.23 /System/Library/Frameworks/CoreService
yangzhang       51162   0.0  0.3   468920  28520   ??  S     8:30下午   0:00.24 /System/Library/Frameworks/CoreService
yangzhang       51159   0.0  0.1  2501020   7252   ??  Ss    8:30下午   0:00.07 /System/Library/PrivateFrameworks/Xpro
yangzhang       50417   0.0  0.1  2501476   9936   ??  S     8:16下午   0:00.08 /System/Library/Frameworks/CoreService
yangzhang       50413   0.0  0.1  2501476  10020   ??  S     8:16下午   0:00.09 /System/Library/Frameworks/CoreService
yangzhang       50412   0.0  0.1  2501476   9932   ??  S     8:16下午   0:00.07 /System/Library/Frameworks/CoreService
yangzhang       50411   0.0  0.1  2501476   9928   ??  S     8:16下午   0:00.07 /System/Library/Frameworks/CoreService
yangzhang       50410   0.0  0.0  2499736   3604   ??  S     8:16下午   0:00.07 /System/Library/PrivateFrameworks/Core
yangzhang       50409   0.0  0.1  2501476   9988   ??  S     8:16下午   0:00.10 /System/Library/Frameworks/CoreService
yangzhang       50408   0.0  0.1  2501476   9996   ??  S     8:16下午   0:00.12 /System/Library/Frameworks/CoreService
yangzhang       50407   0.0  0.1  2501476  10024   ??  S     8:16下午   0:00.15 /System/Library/Frameworks/CoreService
yangzhang       50406   0.0  0.1  2501476   9988   ??  S     8:16下午   0:00.11 /System/Library/Frameworks/CoreService
yangzhang       50388   0.0  0.3  2761372  27728   ??  S     8:15下午   0:02.95 /Applications/Calendar.app/Contents/Ma
yangzhang       49088   0.0  0.1  2483672  12224   ??  S     7:48下午   0:00.25 /System/Library/Frameworks/CoreService
yangzhang       48217   0.0  0.0  2451344    608 s000  T     7:30下午   0:00.01 less
yangzhang       48206   0.0  0.1  2478756  11000   ??  S     7:30下午   0:00.27 /System/Library/Frameworks/CoreService
```

- 在采用微内核结构的操作系统中，进程的功能发生了变化
  - 它只是资源分配的单位，而不再是调度运行的单位
  - 在微内核系统中，真正调度运行的基本单位是线程。真正实现并发功能的单位是线程

- 线程
    - 线程是进程中执行运算的最小单位，亦即执行处理机调度的基本单位
    - 如果把进程理解为在逻辑上操作系统所完成的任务，那么线程表示完成该任务的许多可能的子任务之一
    - 线程可以在处理器上独立调度执行，在多处理器环境下就允许几个线程各自在单独处理器上进行。操作系统提供线程方便而有效地实现这种并发性

- 线程的优点
    - 易于调度，提高并发性
    - 创建开销少
    - 利于充分发挥多处理器的功能

进程和线程-联系

- 线程是指进程内的一个执行单元，也是进程内的可调度实体
  - 一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程
  - 进程独享分配的资源，线程共享同一进程的所有资源
  - CPU核心分给线程，真正在CPU上运行的是线程
  - 线程在执行过程中需要协作同步。不同进程的线程间要利用消息通信的办法实现同步

进程和线程-区别

- 调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位

- 拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源

- 系统开销：在创建或撤消进程时，由于系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤消线程时的开销

通信问题-进程间通信

- 管道通信：
  - 无名管道用于具有亲缘关系的父子进程间的通信
  - 有名管道除了具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。
- 信号(Signal)：软件层次上对中断机制的一种模拟，用于通知进程有某事件发生，进程收到信号与处理器收到中断请求效果上一致

- 消息队列(Message queue)：具有写权限得进程可以按照一定得规则向消息队列中添加新信息，有读权限得进程则可以从消息队列中读取信息

- 共享内存(Shared memory)：多个进程可以访问同一块内存空间，需要依靠某种同步操作，如互斥锁和信号量等。

- 信号量(Semaphore)：进程/线程之间得同步和互斥手段。

- 套接字(Socket)：更为一般得进程间通信机制，它可用于网络中不同机器之间的进程间通信，应用非常广泛。

通信问题-线程间通信

- 信号量
  - 互斥型信号量、二进制信号量、整数型信号量、记录型信号量
- 消息队列
- 事件（Event）

创建一个进程：fork函数

- #include <unistd.h>
- pid_t   fork(void)
  - 成功时返回PID
  - 失败返回-1

fork的工作原理

- fork创建当前进程的副本
- fork执行后才执行需要并行的代码
- fork函数通过pid来判断执行哪一段代码

- 调用fork函数返回的pid
  - 父进程：返回子进程的pid
  - 子进程：返回0
- 其中
  - 父进程：调用fork的主体，原进程
  - 子进程：父进程复制出的进程

✓ 父进程

```
int gval=10;
int main(void)
{
  int lval=20;
  lval+=5;
  gval++;
  pid_t pid=fork();
  if(pid == 0)
    gval++;
  else
    lval++;
  . . . . .
}
```

pid为子进程ID

复制发生点

运行

复制

✓ 子进程

```
// gval复制为11
int main(void)
{
  // lval复制为25
  . . . .
  pid_t pid=fork();
  if(pid == 0)
    gval++;
  else
    lval++;
  . . . . .
}
```

pid为0!

运行

❖ fork.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.
4.  int gval=10;
5.  int main(int argc, char *argv[])
6.  {
7.      pid_t pid;
8.      int lval=20;
9.      gval++, lval+=5;
10.
11.     pid=fork();
12.     if(pid==0)  // if Child Process
13.         gval+=2, lval+=2;
14.     else        // if Parent Process
15.         gval-=2, lval-=2;
16.
17.     if(pid==0)
18.         printf("Child Proc: [%d, %d] \n", gval, lval);
19.     else
20.         printf("Parent Proc: [%d, %d] \n", gval, lval);
21.     return 0;
22. }
```

完