1. 实现IO复用的传统方法select和poll，但是性能不满意，因此有**[Linux](http://lib.csdn.net/base/linux" \o "Linux知识库" \t "_blank)**的epoll，BSD的kqueue，Solaris的/dev/poll,Windows的IOCP。

如，select不适合以web服务器端开发为主流的现代开发环境。

a. 调用select后常见的针对所有文件描述符的循环语句；

b. 每次调用select函数时都需要向函数传递监视对象信息。

—— 致命弱点

2. select的优点：

a. 程序具有兼容性；

b. 服务器端接入者少。

3. epoll可以克服select的缺点。

epoll服务器端用到的三个函数

epoll\_create：创建保存epoll文件描述符的空间 //对应 fd\_set

epoll\_ctl：向空间（位数组）注册并注销文件描述符 //对应FD\_SET，FD\_CLR

epoll\_wait：等待文件描述符发生变化 //对应select

1. //epoll将发生事件的文件描述符集中在一起，放在epoll\_event结构体中
2. **struct** epoll\_event{
3. \_\_uint32\_t events;
4. epoll\_data\_t data;
5. }
7. **typedef** **union** epoll\_data{
8. **void** \* ptr;
9. **int** fd;
10. \_\_uint32\_t u32;
11. \_\_uint64\_t u64;
12. }epoll\_data\_t;

4. epoll\_create函数

调用epoll\_create函数时创建的文件描述符保存空间称为“epoll例程”，size只是建议epoll例程大小，实际大小由**[操作系统](http://lib.csdn.net/base/operatingsystem" \o "操作系统知识库" \t "_blank)**决定。

Linux2.6.8之后，内核忽略size参数，会根据情况自动调整。

epoll\_create函数创建的资源与套接字相同，由操作系统管理，终止时要close。

1. #include <sys/epoll.h>
3. **int** epoll\_create(**int** size)
5. 成功时返回epoll文件描述符，失败返回-1
7. size epoll实例的大小

5. epoll\_ctl函数

1. **int** epoll\_ctl(**int** epfd, **int** op, **int** fd, **struct** epoll\_event \*event)
3. 成功时返回0，失败-1
5. epfd：用于注册监视对象的epoll例程（事件发生的监视范围）的文件描述符
6. op：用于指定监视对象的添加、删除或更改//EPOLL\_CTL\_ADD,EPOLL\_CTL\_DEL,EPOLL\_CTL\_MOD
7. fd：需要注册的监视对象文件描述符
8. event：监视对象事件类型
9. **struct** epoll\_event event;
11. event.events = EPOLLIN; //
12. event.data.fd = sockfd;
13. epoll\_ctl(epfd,EPOLL\_CTL\_ADD,sockfd,&event);
15. events 成员：
16. EPOLLIN：需要读取数据的情况
17. EPOLLOUT：输出缓冲为空，可以立即发送数据的情况
18. EPOLLPRI：受到OOB数据情况
19. EPOLLRDHUP：断开连接或半关闭的情况
20. EPOLLERR：发生错误的情况；
21. EPOLLET：以边缘触发的方式得到事件通知
22. EPOLLONESHOT：发生一次事件后，相应的文件描述符不再受到事件通知

6. epoll\_wait函数

1. **int** epoll\_wait(**int** epfd,**struct** epoll\_event \* events,**int** maxevents,**int** timeout)
3. 成功时返回事件的文件描述符数，失败-1
5. epfd：事件发生监视范围epoll例程的文件描述符
6. events：保存发生事件的文件描述符集合的结构体地址值
7. maxevents：第二个参数中可以保存的最大事件数
8. timeout：以1/1000秒为单位等待时间

7. epoll实现的echo服务器

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>
9. #define BUF\_SIZE    100
10. #define EPOLL\_SIZE  50
12. **void** error\_handling(**char** \*message);
14. **int** main(**int** argc, **char** \*argv[]){
15. **int** serv\_sock,clnt\_sock;
16. **char** buf[BUF\_SIZE];
17. **struct** sockaddr\_in serv\_adr;
18. **struct** sockaddr\_in clnt\_adr;
19. socklen\_t adr\_sz;
20. **int** str\_len,i;
21. **struct** epoll\_event \*ep\_events;
22. **struct** epoll\_event event;
23. **int** epfd,event\_cnt;
25. **if**(argc != 2){
26. printf("Usage : %s <port>\n",argv[0]);
27. exit(1);
28. }
30. serv\_sock = socket(PF\_INET,SOCK\_STREAM,0);
31. **if**(serv\_sock == -1){
32. error\_handling("socket error");
33. }
35. memset(&serv\_adr,0,**sizeof**(serv\_adr));
36. serv\_adr.sin\_family=AF\_INET;
37. serv\_adr.sin\_addr.s\_addr=htonl(INADDR\_ANY);
38. serv\_adr.sin\_port=htons(atoi(argv[1]));
40. **if**(bind(serv\_sock,(**struct** sockaddr\*)&serv\_adr,**sizeof**(serv\_adr)) == -1){
41. error\_handling("bind() error");
42. }
44. **if**(listen(serv\_sock,5) == -1){
45. error\_handling("listen() error");
46. }
48. epfd = epoll\_create(EPOLL\_SIZE);
49. ep\_events = malloc(**sizeof**(**struct** epoll\_event)\*EPOLL\_SIZE);
51. event.events = EPOLLIN;
52. event.data.fd = serv\_sock;
53. epoll\_ctl(epfd,EPOLL\_CTL\_ADD,serv\_sock,&event);
55. **while**(1){
56. event\_cnt = epoll\_wait(epfd,ep\_events,EPOLL\_SIZE,-1);
58. **if**(event\_cnt  == -1){
59. puts("epoll wait error");
60. **break**;
61. }
63. **for**(i=0;i<event\_cnt;i++){
64. **if**(ep\_events[i].data.fd == serv\_sock){
65. adr\_sz = **sizeof**(clnt\_adr);
66. clnt\_sock = accept(serv\_sock,(**struct** sockaddr \*)&clnt\_adr,&adr\_sz);
67. event.events = EPOLLIN;
68. event.data.fd = clnt\_sock;
69. epoll\_ctl(epfd,EPOLL\_CTL\_ADD,clnt\_sock,&event);
70. printf("connected client : %d \n", clnt\_sock);
71. }**else**{  //read message
72. str\_len = read(ep\_events[i].data.fd,buf,BUF\_SIZE);
74. **if**(str\_len == 0){   //close
75. epoll\_ctl(epfd,EPOLL\_CTL\_DEL,ep\_events[i].data.fd,NULL);
76. close(ep\_events[i].data.fd);
77. printf("closed client %d \n",ep\_events[i].data.fd);
78. }**else**{
79. write(ep\_events[i].data.fd,buf,str\_len);    //echo

82. }
83. }
84. }
85. }
86. close(serv\_sock);
87. close(epfd);
88. **return** 0;
89. }
91. **void** error\_handling(**char** \*message){
93. fputs(message,stderr);
94. fputs("\n",stderr);
95. exit(1);
96. }