1. 两者区别：在于发生事件的时间点，条件触发：只要输入缓冲有数据就一直通知该事件；边缘触发：输入缓冲受到数据时仅注册一次事件。

epoll默认以条件触发方式工作，select也是以条件触发模式工作的。

2. 条件触发事件特性：

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    4
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. puts("return epoll wait");
64. **for**(i=0;i<event\_cnt;i++){
65. **if**(ep\_events[i].data.fd == serv\_sock){
66. adr\_sz = **sizeof**(clnt\_adr);
67. clnt\_sock = accept(serv\_sock,(**struct** sockaddr \*)&clnt\_adr,&adr\_sz);
68. event.events = EPOLLIN;
69. event.data.fd = clnt\_sock;
70. epoll\_ctl(epfd,EPOLL\_CTL\_ADD,clnt\_sock,&event);
71. printf("connected client : %d \n", clnt\_sock);
72. }**else**{  //read message
73. str\_len = read(ep\_events[i].data.fd,buf,BUF\_SIZE);
75. **if**(str\_len == 0){   //close
76. epoll\_ctl(epfd,EPOLL\_CTL\_DEL,ep\_events[i].data.fd,NULL);
77. close(ep\_events[i].data.fd);
78. printf("closed client %d \n",ep\_events[i].data.fd);
79. }**else**{
80. write(ep\_events[i].data.fd,buf,str\_len);    //echo

83. }
84. }
85. }
86. }
87. close(serv\_sock);
88. close(epfd);
89. **return** 0;
90. }
92. **void** error\_handling(**char** \*message){
94. fputs(message,stderr);
95. fputs("\n",stderr);
96. exit(1);
97. }

3. 边缘触发

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. #include <errno.h>
9. #include <fcntl.h>
11. #define BUF\_SIZE    4
12. #define EPOLL\_SIZE  50
14. **void** error\_handling(**char** \*message);
15. **void** setnonblockingmode(**int** fd);
17. **int** main(**int** argc, **char** \*argv[]){
18. **int** serv\_sock,clnt\_sock;
19. **char** buf[BUF\_SIZE];
20. **struct** sockaddr\_in serv\_adr;
21. **struct** sockaddr\_in clnt\_adr;
22. socklen\_t adr\_sz;
23. **int** str\_len,i;
24. **struct** epoll\_event \*ep\_events;
25. **struct** epoll\_event event;
26. **int** epfd,event\_cnt;
28. **if**(argc != 2){
29. printf("Usage : %s <port>\n",argv[0]);
30. exit(1);
31. }
33. serv\_sock = socket(PF\_INET,SOCK\_STREAM,0);
34. **if**(serv\_sock == -1){
35. error\_handling("socket error");
36. }
38. memset(&serv\_adr,0,**sizeof**(serv\_adr));
39. serv\_adr.sin\_family=AF\_INET;
40. serv\_adr.sin\_addr.s\_addr=htonl(INADDR\_ANY);
41. serv\_adr.sin\_port=htons(atoi(argv[1]));
43. **if**(bind(serv\_sock,(**struct** sockaddr\*)&serv\_adr,**sizeof**(serv\_adr)) == -1){
44. error\_handling("bind() error");
45. }
47. **if**(listen(serv\_sock,5) == -1){
48. error\_handling("listen() error");
49. }
51. epfd = epoll\_create(EPOLL\_SIZE);
52. ep\_events = malloc(**sizeof**(**struct** epoll\_event)\*EPOLL\_SIZE);
54. event.events = EPOLLIN;
55. event.data.fd = serv\_sock;
56. epoll\_ctl(epfd,EPOLL\_CTL\_ADD,serv\_sock,&event);
58. **while**(1){
59. event\_cnt = epoll\_wait(epfd,ep\_events,EPOLL\_SIZE,-1);
61. **if**(event\_cnt  == -1){
62. puts("epoll wait error");
63. **break**;
64. }
66. puts("return epoll wait");
67. **for**(i=0;i<event\_cnt;i++){
68. **if**(ep\_events[i].data.fd == serv\_sock){
69. adr\_sz = **sizeof**(clnt\_adr);
70. clnt\_sock = accept(serv\_sock,(**struct** sockaddr \*)&clnt\_adr,&adr\_sz);
71. setnonblockingmode(clnt\_sock);
72. event.events = EPOLLIN | EPOLLET;<span style="white-space:pre"> </span>//设置成边缘触发
73. event.data.fd = clnt\_sock;
74. epoll\_ctl(epfd,EPOLL\_CTL\_ADD,clnt\_sock,&event);
75. printf("connected client : %d \n", clnt\_sock);
76. }**else**{  //read message
77. **while**(1){
78. str\_len = read(ep\_events[i].data.fd,buf,BUF\_SIZE);
80. **if**(str\_len == 0){   //close
81. epoll\_ctl(epfd,EPOLL\_CTL\_DEL,ep\_events[i].data.fd,NULL);
82. close(ep\_events[i].data.fd);
83. printf("closed client %d \n",ep\_events[i].data.fd);
84. **break**;
85. }**else** **if**(str\_len < 0){
86. **if**(errno == EAGAIN){<span style="white-space:pre">  </span>//无数据可读
87. **break**;
88. }
89. }**else**{
90. write(ep\_events[i].data.fd,buf,str\_len);    //echo
91. }
92. }
93. }
94. }
95. }
96. close(serv\_sock);
97. close(epfd);
98. **return** 0;
99. }
101. **void** error\_handling(**char** \*message){
103. fputs(message,stderr);
104. fputs("\n",stderr);
105. exit(1);
106. }
108. **void** setnonblockingmode(**int** fd){
109. **int** flag = fcntl(fd,F\_GETFL,0);<span style="white-space:pre">   </span>//设置成非阻塞形式
110. fcntl(fd,F\_SETFL,flag | O\_NONBLOCK);
111. }

5. 条件触发与边缘触发比较：应该从服务器端实现模型角度考虑。

边缘触发能够做到接收数据与处理数据的时间点分离。