socket编程 及select poll epoll示例

1、关于字节排序    网际协议采用大端字节序，来传输多字节整数。    系统提供了转换的宏定义，如果主机与网际协议相同，则宏定义为空。

2、客户端    socket -> connect(阻塞,三次握手)-> rcv

3、服务器端    socket -> bind -> listen -> accept(阻塞,三次握手)-> send4、函数介绍

## socket

          1)函数原型 int socket(int family, int type, int protocol)

          2)参数：

family: 协议族AF\_INET,IPv4协议 ...

type : type 套接字类型SOCK\_STREAM 字节流套接字

protocol: IPPROCO\_TCP IPPROCO\_UDP IPPROCO\_SCTP

          3)返回值 成功:返回套接字符 错误:返回INVALID\_SOCKET(-1)

      4)示例

**[cpp]** [view plain](http://blog.csdn.net/mmz_xiaokong/article/details/8704988) [copy](http://blog.csdn.net/mmz_xiaokong/article/details/8704988)

1. #include <netinet/in.h>
2. #include <sys/types.h>
3. #include <sys/socket.h>
4. **int** main()
5. {
6. **int** socketfd;
7. **struct** sockaddr\_in servaddr;
9. **if**((socketfd = socket(AF\_INET, SOCK\_STREAM, 0)) == -1)
10. {
11. **return** -1;
12. }
13. }

## connect

1)函数原型 int connect(int sockfd, const struct sockaddr \*servaddr, socklen\_t addrlen)  
2)参数：      
            sockfd: socket 函数返回的套接字描述符  
            servaddr : 服务器的IP和端口  
            addrlen: 长度(sizeof(servaddr))  
3)返回值

成功:0

错误:返回INVALID\_SOCKET(-1)  
4)示例

**[cpp]** [view plain](http://blog.csdn.net/mmz_xiaokong/article/details/8704988) [copy](http://blog.csdn.net/mmz_xiaokong/article/details/8704988)

1. #include <stdio.h>
2. #include <string.h>
3. #include <netinet/in.h>
4. #include <sys/types.h>
5. #include <sys/socket.h>
7. **int** main()
8. {
9. **int** socketfd;
10. **struct** sockaddr\_in servaddr;
12. **if**((socketfd = socket(AF\_INET, SOCK\_STREAM, 0)) == -1)
13. {
14. printf("socket error\n");
15. **return** -1;
16. }
18. bzero(&servaddr, **sizeof**(servaddr));
20. servaddr.sin\_addr.s\_addr = inet\_addr("192.168.0.218");
21. servaddr.sin\_family = AF\_INET;
22. servaddr.sin\_port = htons(55000);
24. **if**(connect(socketfd, (**struct** sockaddr\*) &servaddr, **sizeof**(servaddr)) < 0)
25. {
26. printf("connect error\n");
27. }
29. **return** 0;
30. }

## bind

        1)函数原型 int bind(int sockfd, const struct sockaddr \*servaddr, socklen\_t addrlen)  
        2)参数：      
            sockfd: socket 函数返回的套接字描述符  
            servaddr : 服务器的IP和端口  
            addrlen: 长度(sizeof(servaddr))  
        3)返回值  
            成功:0  
            错误:返回INVALID\_SOCKET(-1)  
            

## listen

1)函数原型 int listen(int sockfd, int backlog)  
     2)参数：      
            sockfd: socket 函数返回的套接字描述符  
            backlog : 内核中套接字排队的最大个数  
     3)返回值  
            成功:0  
            错误:返回INVALID\_SOCKET

## accept

1)函数原型 int accept(int sockfd, const struct sockaddr \*servaddr, socklen\_t \*addrlen)  
     2)参数：      
            sockfd: socket 函数返回的套接字描述符  
  
     3)返回值  
            servaddr : 客户进程的IP和端口(可设为null)  
            addrlen: 长度(sizeof(servaddr))(可设为null)  
            成功:从监听套接字返回已连接套接字  
            错误:  
            如果对客户信息不感兴趣，后两个参数可以置空。

        4）示例

**[cpp]** [view plain](http://blog.csdn.net/mmz_xiaokong/article/details/8704988) [copy](http://blog.csdn.net/mmz_xiaokong/article/details/8704988)

1. #include <stdio.h>
2. #include <string.h>
3. #include <netinet/in.h>
4. #include <sys/types.h>
5. #include <sys/socket.h>
7. **int** main()
8. {
9. **int** count = 0;
10. **int** listenfd, socketfd;
11. **int** nread;
12. **struct** sockaddr\_in servaddr;
13. **struct** timeval timeoutval;
14. **char** readbuf[256];
16. printf("accept started\n");
18. //socket
19. **if**((listenfd = socket(AF\_INET, SOCK\_STREAM, 0)) == -1)
20. {
21. printf("socket error\n");
22. **return** -1;
23. }
25. bzero(&servaddr, **sizeof**(servaddr));
26. servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);
27. servaddr.sin\_family = AF\_INET;
28. servaddr.sin\_port = htons(59000);
30. //bind
31. **if**(bind(listenfd, (**struct** sockaddr\*)&servaddr, **sizeof**(servaddr)) < 0)
32. {
33. printf("bind error\n");
34. //return -1;
35. }
37. //listen
38. listen(listenfd, 5);
40. //accept
41. socketfd = accept(listenfd, NULL, NULL);


45. **while**(1)
46. {
47. printf("start receive %d...\n", count++);
48. memset(readbuf, **sizeof**(readbuf), 0);
50. nread = recv(socketfd, readbuf, 10, 0);
51. **if**(nread>0)
52. {
53. readbuf[10] = '\0';
54. printf("receiveed %s, nread = %d\n\n", readbuf, nread);
55. }
56. }
58. **return** 0;
59. }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
    从 I/O 事件分派机制来看,使用 select()是不合适的,因为它所支持的并发连接数有限(通  
    常在 1024 个以内)。如果考虑性能,poll()也是不合适的,尽管它可以支持的较高的 TCP 并发数,但是由于其采用“轮询”机制,当并发数较高时,其运行效率相当低,并可能存在 I/O 事件分派不均,导致部分 TCP 连接上的 I/O 出现“饥饿”现象。而如果使用 epoll 或 AIO,则没有上述问题(早期 Linux 内核的 AIO 技术实现是通过在内核中为每个 I/O 请求创建一个线程来实现的,这种实现机制在高并发 TCP 连接的情形下使用其实也有严重的性能问题。但在最新的Linux 内核中,AIO 的实现已经得到改进)。  
    支持一个进程打开大数目的 socket 描述符(FD)select 最不能忍受的是一个进程所打开的FD 是有一定限制的,由 FD\_SETSIZE 设置,默认值是 2048。对于那些需要支持的上万连接数目的 IM 服务器来说显然太少了。  
    这时候你一是可以选择修改这个宏然后重新编译内核,不过资料  
    也同时指出这样会带来网络效率的下降,二是可以选择多进程的解决方案(传统的 Apache 方案),不过虽然 linux 上面创建进程的代价比较小,但仍旧是不可忽视的,加上进程间数据同步远比不上线程间同步的高效,所以也不是一种完美的方案。不过 epoll 则没有这个限制,它所支持的 FD 上限是最大可以打开文件的数目,这个数字一般远大于 2048,举个例子,在 1GB 内存的机器上大约是 10 万左右,具体数目可以 cat /proc/sys/fs/file-max 察看,一般来说这个数  
    目和系统内存关系很大。  
    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/        

## select函数

    1)函数原型 int select(int maxfdp,fd\_set \*readfds,fd\_set \*writefds,fd\_set \*errorfds,struct timeval \*timeout);  
    2)参数：      
        sockfd: socket 函数返回的套接字描述符  
        readfds : 读描述符集合  
        writefds: 写描述符集合  
        errorfds: 错误描述符集合  
        timeout:  超时  
    3)返回值  
        成功:返回值 0:无 >0:描述符就绪的总位数  
        错误:返回INVALID\_SOCKET(-1)  
    4）包含头文件: include <sys/select.h> include <sys/time.h>  
    5）示例

**[cpp]** [view plain](http://blog.csdn.net/mmz_xiaokong/article/details/8704988) [copy](http://blog.csdn.net/mmz_xiaokong/article/details/8704988)

1. /\* 实现功能：通过select处理多个socket
2. \* 监听一个端口,监听到有链接时,添加到select的w.
3. \*/
4. #include "select.h"
5. #include <stdio.h>
6. #include <stdlib.h>
7. #include <sys/socket.h>
8. #include <sys/select.h>
9. #include <sys/time.h>
10. #include <netinet/in.h>
12. **typedef** **struct** \_CLIENT{
13. **int** fd;
14. **struct** sockaddr\_in addr; /\* client's address information \*/
15. } CLIENT;
17. #define MYPORT 59000
19. //最多处理的connect
20. #define BACKLOG 5
22. //最多处理的connect
23. CLIENT client[BACKLOG];
25. //当前的连接数
26. **int** currentClient = 0;
28. //数据接受 buf
29. #define REVLEN 10
30. **char** recvBuf[REVLEN];
31. //显示当前的connection
32. **void** showClient();
34. **int** main()
35. {
36. **int** i, ret, sinSize;
37. **int** recvLen = 0;
38. fd\_set readfds, writefds;
39. **int** sockListen, sockSvr, sockMax;
40. **struct** timeval timeout;
41. **struct** sockaddr\_in server\_addr;
42. **struct** sockaddr\_in client\_addr;
44. **for**(i=0; i<BACKLOG; i++)
45. {
46. client[i].fd = -1;
47. }
49. //socket
50. **if**((sockListen=socket(AF\_INET, SOCK\_STREAM, 0)) < 0)
51. {
52. printf("socket error\n");
53. **return** -1;
54. }
56. bzero(&server\_addr, **sizeof**(server\_addr));
57. server\_addr.sin\_family  =  AF\_INET;
58. server\_addr.sin\_port = htons(MYPORT);
59. server\_addr.sin\_addr.s\_addr  =  htonl(INADDR\_ANY);
61. //bind
62. **if**(bind(sockListen, (**struct** sockaddr\*)&server\_addr, **sizeof**(server\_addr)) < 0)
63. {
64. printf("bind error\n");
65. **return** -1;
66. }
68. //listen
69. **if**(listen(sockListen, 5) < 0)
70. {
71. printf("listen error\n");
72. **return** -1;
73. }
75. **for**(i=0; i<BACKLOG; i++)
76. {
77. client[i].fd = -1;
78. }
80. //select
81. **while**(1)
82. {
83. FD\_ZERO(&readfds);
84. FD\_SET(sockListen, &readfds);
85. sockMax = sockListen;
87. //加入client
88. **for**(i=0; i<BACKLOG; i++)
89. {
90. **if**(client[i].fd >0)
91. {
92. FD\_SET(client[i].fd, &readfds);
93. **if**(sockMax<client[i].fd)
94. sockMax = client[i].fd;
95. }
96. }
98. timeout.tv\_sec=3;
99. timeout.tv\_usec=0;
100. //select
101. ret = select((**int**)sockMax+1, &readfds, NULL, NULL, &timeout);
102. **if**(ret < 0)
103. {
104. printf("select error\n");
105. **break**;
106. }
107. **else** **if**(ret == 0)
108. {
109. printf("timeout ...\n");
110. **continue**;
111. }
112. printf("test111\n");
114. //读取数据
115. **for**(i=0; i<BACKLOG; i++)
116. {
117. **if**(client[i].fd>0 && FD\_ISSET(client[i].fd, &readfds))
118. {
119. **if**(recvLen != REVLEN)
120. {
121. **while**(1)
122. {
123. //recv数据
124. ret = recv(client[i].fd, (**char** \*)recvBuf+recvLen, REVLEN-recvLen, 0);
125. **if**(ret == 0)
126. {
127. client[i].fd = -1;
128. recvLen = 0;
129. **break**;
130. }
131. **else** **if**(ret < 0)
132. {
133. client[i].fd = -1;
134. recvLen = 0;
135. **break**;
136. }
137. //数据接受正常
138. recvLen = recvLen+ret;
139. **if**(recvLen<REVLEN)
140. {
141. **continue**;
142. }
143. **else**
144. {
145. //数据接受完毕
146. printf("%s, buf = %s\n", inet\_ntoa(client[i].addr.sin\_addr) , recvBuf);
147. //close(client[i].fd);
148. //client[i].fd = -1;
149. recvLen = 0;
150. **break**;
151. }
152. }
153. }
154. }
155. }
157. //如果可读
158. **if**(FD\_ISSET(sockListen, &readfds))
159. {
160. printf("isset\n");
161. sockSvr = accept(sockListen, NULL, NULL);//(struct sockaddr\*)&client\_addr
163. **if**(sockSvr == -1)
164. {
165. printf("accpet error\n");
166. }
167. **else**
168. {
169. currentClient++;
170. }
172. **for**(i=0; i<BACKLOG; i++)
173. {
174. **if**(client[i].fd < 0)
175. {
176. client[i].fd = sockSvr;
177. client[i].addr = client\_addr;
178. printf("You got a connection from %s \n",inet\_ntoa(client[i].addr.sin\_addr) );
179. **break**;
180. }
181. }
182. //close(sockListen);
183. }
184. }
186. printf("test\n");
187. **return** 0;
188. }
190. //显示当前的connection
191. **void** showClient()
192. {
193. **int** i;
194. printf("client count = %d\n", currentClient);
196. **for**(i=0; i<BACKLOG; i++)
197. {
198. printf("[%d] = %d", i, client[i].fd);
199. }
200. printf("\n");
201. }

## poll函数

    1)函数原型 int select(int maxfdp,fd\_set \*readfds,fd\_set \*writefds,fd\_set \*errorfds,struct timeval \*timeout);  
    2)参数：      
        sockfd: socket 函数返回的套接字描述符  
        readfds : 读描述符集合  
        writefds: 写描述符集合  
        errorfds: 错误描述符集合  
        timeout:  超时  
    3)返回值  
        成功:返回值 0:无 >0:描述符就绪的总位数  
        错误:返回INVALID\_SOCKET(-1)  
    4）包含头文件: include <sys/select.h> include <sys/time.h>

    5) 示例

**[cpp]** [view plain](http://blog.csdn.net/mmz_xiaokong/article/details/8704988) [copy](http://blog.csdn.net/mmz_xiaokong/article/details/8704988)

1. /\* 实现功能：通过poll, 处理多个socket
2. \* 监听一个端口,监听到有链接时,添加到poll.
3. \*/
4. #include "select.h"
5. #include <stdio.h>
6. #include <stdlib.h>
7. #include <string.h>
8. #include <sys/socket.h>
9. #include <poll.h>
10. #include <sys/time.h>
11. #include <netinet/in.h>
13. **typedef** **struct** \_CLIENT{
14. **int** fd;
15. **struct** sockaddr\_in addr; /\* client's address information \*/
16. } CLIENT;
18. #define MYPORT 59000
20. //最多处理的connect
21. #define BACKLOG 5
23. //当前的连接数
24. **int** currentClient = 0;
26. //数据接受 buf
27. #define REVLEN 10
28. **char** recvBuf[REVLEN];
30. #define OPEN\_MAX 1024
32. **int** main()
33. {
34. **int** i, ret, sinSize;
35. **int** recvLen = 0;
36. fd\_set readfds, writefds;
37. **int** sockListen, sockSvr, sockMax;
38. **int** timeout;
39. **struct** sockaddr\_in server\_addr;
40. **struct** sockaddr\_in client\_addr;
42. **struct** pollfd clientfd[OPEN\_MAX];

45. //socket
46. **if**((sockListen=socket(AF\_INET, SOCK\_STREAM, 0)) < 0)
47. {
48. printf("socket error\n");
49. **return** -1;
50. }
52. bzero(&server\_addr, **sizeof**(server\_addr));
53. server\_addr.sin\_family  =  AF\_INET;
54. server\_addr.sin\_port = htons(MYPORT);
55. server\_addr.sin\_addr.s\_addr  =  htonl(INADDR\_ANY);
57. //bind
58. **if**(bind(sockListen, (**struct** sockaddr\*)&server\_addr, **sizeof**(server\_addr)) < 0)
59. {
60. printf("bind error\n");
61. **return** -1;
62. }
64. //listen
65. **if**(listen(sockListen, 5) < 0)
66. {
67. printf("listen error\n");
68. **return** -1;
69. }

72. //clientfd 初始化
73. clientfd[0].fd = sockListen;
74. clientfd[0].events = POLLIN; //POLLRDNORM;
75. sockMax = 0;
76. **for**(i=1; i<OPEN\_MAX; i++)
77. {
78. clientfd[i].fd = -1;
79. }
81. //select
82. **while**(1)
83. {
84. timeout=3000;
85. //select
86. ret = poll(clientfd, sockMax+1, timeout);
88. **if**(ret < 0)
89. {
90. printf("select error\n");
91. **break**;
92. }
93. **else** **if**(ret == 0)
94. {
95. printf("timeout ...\n");
96. **continue**;
97. }
99. **if** (clientfd[0].revents & POLLIN)//POLLRDNORM
100. {
101. sockSvr = accept(sockListen, NULL, NULL);//(struct sockaddr\*)&client\_addr
103. **if**(sockSvr == -1)
104. {
105. printf("accpet error\n");
106. }
107. **else**
108. {
109. currentClient++;
110. }
112. **for**(i=0; i<OPEN\_MAX; i++)
113. {
114. **if**(clientfd[i].fd<0)
115. {
116. clientfd[i].fd = sockSvr;
117. **break**;
118. }
119. }
120. **if**(i==OPEN\_MAX)
121. {
122. printf("too many connects\n");
123. **return** -1;
124. }
125. clientfd[i].events = POLLIN;//POLLRDNORM;
126. **if**(i>sockMax)
127. sockMax = i;
128. }
130. //读取数据
131. **for**(i=1; i<=sockMax; i++)
132. {
133. **if**(clientfd[i].fd < 0)
134. **continue**;
136. **if** (clientfd[i].revents & (POLLIN | POLLERR))//POLLRDNORM
137. {
138. **if**(recvLen != REVLEN)
139. {
140. **while**(1)
141. {
142. //recv数据
143. ret = recv(clientfd[i].fd, (**char** \*)recvBuf+recvLen, REVLEN-recvLen, 0);
144. **if**(ret == 0)
145. {
146. clientfd[i].fd = -1;
147. recvLen = 0;
148. **break**;
149. }
150. **else** **if**(ret < 0)
151. {
152. clientfd[i].fd = -1;
153. recvLen = 0;
154. **break**;
155. }
156. //数据接受正常
157. recvLen = recvLen+ret;
158. **if**(recvLen<REVLEN)
159. {
160. **continue**;
161. }
162. **else**
163. {
164. //数据接受完毕
165. printf("buf = %s\n",  recvBuf);
166. //close(client[i].fd);
167. //client[i].fd = -1;
168. recvLen = 0;
169. **break**;
170. }
171. }
172. }
173. }
174. }
175. }
177. **return** 0;
178. }

## epoll函数

        2. 常用模型的缺点  
        如果不摆出来其他模型的缺点，怎么能对比出 Epoll 的优点呢。  
        2.1 PPC/TPC 模型  
        这两种模型思想类似，就是让每一个到来的连接一边自己做事去，别再来烦我 。只是 PPC 是为它开了一个进程，而 TPC 开了一个线程。可是别烦我是有代价的，它要时间和空间啊，连接多了之后，那么多的进程 / 线程切换，这开销就上来了；因此这类模型能接受的最大连接数都不会高，一般在几百个左右。  
        2.2 select 模型  
        1. 最大并发数限制，因为一个进程所打开的 FD （文件描述符）是有限制的，由 FD\_SETSIZE 设置，默认值是 1024/2048 ，因此 Select 模型的最大并发数就被相应限制了。自己改改这个 FD\_SETSIZE ？想法虽好，可是先看看下面吧 …  
        2. 效率问题， select 每次调用都会线性扫描全部的 FD 集合，这样效率就会呈现线性下降，把 FD\_SETSIZE 改大的后果就是，大家都慢慢来，什么？都超时了？？！！  
        3. 内核 / 用户空间 内存拷贝问题，如何让内核把 FD 消息通知给用户空间呢？在这个问题上 select 采取了内存拷贝方法。  
        2.3 poll 模型  
        基本上效率和 select 是相同的， select 缺点的 2 和 3 它都没有改掉。  
        3. Epoll 的提升  
        把其他模型逐个批判了一下，再来看看 Epoll 的改进之处吧，其实把 select 的缺点反过来那就是 Epoll 的优点了。  
        3.1. Epoll 没有最大并发连接的限制，上限是最大可以打开文件的数目，这个数字一般远大于 2048, 一般来说这个数目和系统内存关系很大 ，具体数目可以 cat /proc/sys/fs/file-max 察看。  
        3.2. 效率提升， Epoll 最大的优点就在于它只管你“活跃”的连接 ，而跟连接总数无关，因此在实际的网络环境中， Epoll 的效率就会远远高于 select 和 poll 。  
        3.3. 内存拷贝， Epoll 在这点上使用了“共享内存 ”，这个内存拷贝也省略了。  
           
        4. Epoll 为什么高效  
        Epoll 的高效和其数据结构的设计是密不可分的，这个下面就会提到。  
        首先回忆一下 select 模型，当有 I/O 事件到来时， select 通知应用程序有事件到了快去处理，而应用程序必须轮询所有的 FD 集合，测试每个 FD 是否有事件发生，并处理事件；代码像下面这样：  
  
  
        int res = select(maxfd+1, &readfds, NULL, NULL, 120);  
        if (res > 0)  
        {  
            for (int i = 0; i < MAX\_CONNECTION; i++)  
            {  
                if (FD\_ISSET(allConnection[i], &readfds))  
                {  
                    handleEvent(allConnection[i]);  
                }  
            }  
        }  
        // if(res == 0) handle timeout, res < 0 handle error  
           
        Epoll 不仅会告诉应用程序有I/0 事件到来，还会告诉应用程序相关的信息，这些信息是应用程序填充的，因此根据这些信息应用程序就能直接定位到事件，而不必遍历整个FD 集合。  
        int res = epoll\_wait(epfd, events, 20, 120);  
        for (int i = 0; i < res;i++)  
        {  
            handleEvent(events[n]);  
        }  
        5. Epoll 关键数据结构  
        前面提到 Epoll 速度快和其数据结构密不可分，其关键数据结构就是：  
        struct epoll\_event {  
            \_\_uint32\_t events;      // Epoll events  
            epoll\_data\_t data;      // User data variable  
        };  
        typedef union epoll\_data {  
            void \*ptr;  
            int fd;  
            \_\_uint32\_t u32;  
            \_\_uint64\_t u64;  
        } epoll\_data\_t;  
          
        可见 epoll\_data 是一个 union 结构体 , 借助于它应用程序可以保存很多类型的信息 :fd 、指针等等。有了它，应用程序就可以直接定位目标了。  
        6. 使用 Epoll  
        既然 Epoll 相比 select 这么好，那么用起来如何呢？会不会很繁琐啊 … 先看看下面的三个函数吧，就知道 Epoll 的易用了。  
           
        int epoll\_create(int size);  
        生成一个 Epoll 专用的文件描述符，其实是申请一个内核空间，用来存放你想关注的 socket fd 上是否发生以及发生了什么事件。 size 就是你在这个 Epoll fd 上能关注的最大 socket fd 数，大小自定，只要内存足够。  
          
        int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event );  
        控制某个 Epoll 文件描述符上的事件：注册、修改、删除。其中参数 epfd 是 epoll\_create() 创建 Epoll 专用的文件描述符。相对于 select 模型中的 FD\_SET 和 FD\_CLR 宏。  
        op:EPOLL\_CTL\_ADD  
                 Register the target file descriptor fd on the epoll instance   
              EPOLL\_CTL\_MOD  
                  Change the event event associated with the target file descriptor fd.  
              EPOLL\_CTL\_DEL  
                   Remove  (deregister)  the  target  file descriptor fd from the epoll instance  
  
  
          
        int epoll\_wait(int epfd,struct epoll\_event \* events,int maxevents,int timeout);  
        等待 I/O 事件的发生；参数说明：  
        epfd: 由 epoll\_create() 生成的 Epoll 专用的文件描述符；  
        epoll\_event: 用于回传代处理事件的数组；  
        maxevents: 每次能处理的事件数；  
        timeout: 等待 I/O 事件发生的超时值,单位 ms  
        返回发生事件数。  
        相对于 select 模型中的 select 函数。

**[cpp]** [view plain](http://blog.csdn.net/mmz_xiaokong/article/details/8704988) [copy](http://blog.csdn.net/mmz_xiaokong/article/details/8704988)

1. /\* 实现功能：通过epoll, 处理多个socket
2. \* 监听一个端口,监听到有链接时,添加到epoll\_event
3. \*/
4. #include "select.h"
5. #include <stdio.h>
6. #include <stdlib.h>
7. #include <string.h>
8. #include <sys/socket.h>
9. #include <poll.h>
10. #include <sys/epoll.h>
11. #include <sys/time.h>
12. #include <netinet/in.h>
14. **typedef** **struct** \_CLIENT{
15. **int** fd;
16. **struct** sockaddr\_in addr; /\* client's address information \*/
17. } CLIENT;
19. #define MYPORT 59000
21. //最多处理的connect
22. #define MAX\_EVENTS 500
24. //当前的连接数
25. **int** currentClient = 0;
27. //数据接受 buf
28. #define REVLEN 10
29. **char** recvBuf[REVLEN];
31. //EPOLL相关
32. //epoll描述符
33. **int** epollfd;
34. //事件数组
35. **struct** epoll\_event eventList[MAX\_EVENTS];
37. **void** AcceptConn(**int** srvfd);
38. **void** RecvData(**int** fd);
40. **int** main()
41. {
42. **int** i, ret, sinSize;
43. **int** recvLen = 0;
44. fd\_set readfds, writefds;
45. **int** sockListen, sockSvr, sockMax;
46. **int** timeout;
47. **struct** sockaddr\_in server\_addr;
48. **struct** sockaddr\_in client\_addr;
50. //socket
51. **if**((sockListen=socket(AF\_INET, SOCK\_STREAM, 0)) < 0)
52. {
53. printf("socket error\n");
54. **return** -1;
55. }
57. bzero(&server\_addr, **sizeof**(server\_addr));
58. server\_addr.sin\_family  =  AF\_INET;
59. server\_addr.sin\_port = htons(MYPORT);
60. server\_addr.sin\_addr.s\_addr  =  htonl(INADDR\_ANY);
62. //bind
63. **if**(bind(sockListen, (**struct** sockaddr\*)&server\_addr, **sizeof**(server\_addr)) < 0)
64. {
65. printf("bind error\n");
66. **return** -1;
67. }
69. //listen
70. **if**(listen(sockListen, 5) < 0)
71. {
72. printf("listen error\n");
73. **return** -1;
74. }
76. //1. epoll 初始化
77. epollfd = epoll\_create(MAX\_EVENTS);
78. **struct** epoll\_event event;
79. event.events = EPOLLIN|EPOLLET;
80. event.data.fd = sockListen;
82. //2. epoll\_ctrl
83. **if**(epoll\_ctl(epollfd, EPOLL\_CTL\_ADD, sockListen, &event) < 0)
84. {
85. printf("epoll add fail : fd = %d\n", sockListen);
86. **return** -1;
87. }
89. //epoll
90. **while**(1)
91. {
92. timeout=3000;
93. //3. epoll\_wait
94. **int** ret = epoll\_wait(epollfd, eventList, MAX\_EVENTS, timeout);
96. **if**(ret < 0)
97. {
98. printf("epoll error\n");
99. **break**;
100. }
101. **else** **if**(ret == 0)
102. {
103. printf("timeout ...\n");
104. **continue**;
105. }
107. //直接获取了事件数量,给出了活动的流,这里是和poll区别的关键
108. **int** n = 0;
109. **for**(n=0; n<ret; n++)
110. {
111. //错误退出
112. **if** ((eventList[n].events & EPOLLERR) ||
113. (eventList[n].events & EPOLLHUP) ||
114. !(eventList[n].events & EPOLLIN))
115. {
116. printf ( "epoll error\n");
117. close (eventList[n].data.fd);
118. **return** -1;
119. }
121. **if** (eventList[n].data.fd == sockListen)
122. {
123. AcceptConn(sockListen);
125. }**else**{
126. RecvData(eventList[n].data.fd);
127. //不删除
128. //   epoll\_ctl(epollfd, EPOLL\_CTL\_DEL, pEvent->data.fd, pEvent);
129. }
130. }
131. }
133. close(epollfd);
134. close(sockListen);
136. printf("test\n");
137. **return** 0;
138. }
140. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
141. 函数名：AcceptConn
142. 功能：接受客户端的链接
143. 参数：srvfd：监听SOCKET
144. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
145. **void** AcceptConn(**int** srvfd)
146. {
147. **struct** sockaddr\_in sin;
148. socklen\_t len = **sizeof**(**struct** sockaddr\_in);
149. bzero(&sin, len);
151. **int** confd = accept(srvfd, (**struct** sockaddr\*)&sin, &len);
153. **if** (confd < 0)
154. {
155. printf("bad accept\n");
156. **return**;
157. }**else**
158. {
159. printf("Accept Connection: %d", confd);
160. }
162. //setnonblocking(confd);
164. //4. epoll\_wait
165. //将新建立的连接添加到EPOLL的监听中
166. **struct** epoll\_event event;
167. event.data.fd = confd;
168. event.events =  EPOLLIN|EPOLLET;
169. epoll\_ctl(epollfd, EPOLL\_CTL\_ADD, confd, &event);
170. }
172. //读取数据
173. **void** RecvData(**int** fd)
174. {
175. **int** ret;
176. **int** recvLen = 0;
178. memset(recvBuf, 0, REVLEN);
179. printf("RecvData function\n");
181. **if**(recvLen != REVLEN)
182. {
183. **while**(1)
184. {
185. //recv数据
186. ret = recv(fd, (**char** \*)recvBuf+recvLen, REVLEN-recvLen, 0);
187. **if**(ret == 0)
188. {
189. recvLen = 0;
190. **break**;
191. }
192. **else** **if**(ret < 0)
193. {
194. recvLen = 0;
195. **break**;
196. }
197. //数据接受正常
198. recvLen = recvLen+ret;
199. **if**(recvLen<REVLEN)
200. {
201. **continue**;
202. }
203. **else**
204. {
205. //数据接受完毕
206. printf("buf = %s\n",  recvBuf);
207. recvLen = 0;
208. **break**;
209. }
210. }
211. }
213. printf("content is %s", recvBuf);
214. }