<https://blog.csdn.net/tianmohust/article/details/6677985>

一）.Epoll 介绍

Epoll 可是当前在 Linux 下开发大规模并发网络程序的热门人选， Epoll 在 Linux2.6 内核中正式引入，和 select 相似，其实都 **I/O 多路复用技术而已**，并没有什么神秘的。其实在 Linux 下设计并发网络程序，向来不缺少方法，比如典型的 Apache 模型（ Process Per Connection ，简称 PPC ）， TPC （ Thread Per Connection ）模型，以及 select 模型和 poll 模型，那为何还要再引入 Epoll 这个东东呢？那还是有得说说的 …

二）. 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。

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 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。

总结为：1.**连接数受限**  2.**查找配对速度慢** 3.**数据由内核拷贝到用户态**

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。

三）. 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。

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。

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. 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。

 四）. 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：

**[cpp]** [view plain](https://blog.csdn.net/tianmohust/article/details/6677985) [copy](https://blog.csdn.net/tianmohust/article/details/6677985)

1. **int** res = select(maxfd+1, &readfds, NULL, NULL, 120);
2. **if** (res > 0)
3. {
4. **for** (**int** i = 0; i < MAX\_CONNECTION; i++)
5. {
6. **if** (FD\_ISSET(allConnection[i], &readfds))
7. {
8. handleEvent(allConnection[i]);
9. }
10. }
11. }
12. // 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。

**[cpp]** [view plain](https://blog.csdn.net/tianmohust/article/details/6677985) [copy](https://blog.csdn.net/tianmohust/article/details/6677985)

1. **int** res = epoll\_wait(epfd, events, 20, 120);
2. **for** (**int** i = 0; i < res;i++)
3. {
4. handleEvent(events[n]);
5. }

五）. 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：

**[cpp]** [view plain](https://blog.csdn.net/tianmohust/article/details/6677985) [copy](https://blog.csdn.net/tianmohust/article/details/6677985)

1. **struct** epoll\_event {
2. \_\_uint32\_t events;      // Epoll events
3. epoll\_data\_t data;      // User data variable
4. };
5. **typedef** **union** epoll\_data {
6. **void** \*ptr;
7. **int** fd;
8. \_\_uint32\_t u32;
9. \_\_uint64\_t u64;
10. } 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。

六）. 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。

**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；

返回发生事件数。

相对于 select 模型中的 select 函数。

七）. 例子程序

下面是一个简单 Echo Server 的例子程序，麻雀虽小，五脏俱全，还包含了一个简单的超时检查机制，简洁起见没有做错误处理。

**[cpp]** [view plain](https://blog.csdn.net/tianmohust/article/details/6677985) [copy](https://blog.csdn.net/tianmohust/article/details/6677985)

1. //
2. // a simple echo server using epoll in linux
3. //
4. // 2009-11-05
5. // by sparkling
6. //
7. #include <sys/socket.h>
8. #include <sys/epoll.h>
9. #include <netinet/in.h>
10. #include <arpa/inet.h>
11. #include <fcntl.h>
12. #include <unistd.h>
13. #include <stdio.h>
14. #include <errno.h>
15. #include <iostream>
16. **using** **namespace** std;
17. #define MAX\_EVENTS 500
18. **struct** myevent\_s
19. {
20. **int** fd;
21. **void** (\*call\_back)(**int** fd, **int** events, **void** \*arg);
22. **int** events;
23. **void** \*arg;
24. **int** status; // 1: in epoll wait list, 0 not in
25. **char** buff[128]; // recv data buffer
26. **int** len;
27. **long** last\_active; // last active time
28. };
29. // set event
30. **void** EventSet(myevent\_s \*ev, **int** fd, **void** (\*call\_back)(**int**, **int**, **void**\*), **void** \*arg)
31. {
32. ev->fd = fd;
33. ev->call\_back = call\_back;
34. ev->events = 0;
35. ev->arg = arg;
36. ev->status = 0;
37. ev->last\_active = time(NULL);
38. }
39. // add/mod an event to epoll
40. **void** EventAdd(**int** epollFd, **int** events, myevent\_s \*ev)
41. {
42. **struct** epoll\_event epv = {0, {0}};
43. **int** op;
44. epv.data.ptr = ev;
45. epv.events = ev->events = events;
46. **if**(ev->status == 1){
47. op = EPOLL\_CTL\_MOD;
48. }
49. **else**{
50. op = EPOLL\_CTL\_ADD;
51. ev->status = 1;
52. }
53. **if**(epoll\_ctl(epollFd, op, ev->fd, &epv) < 0)
54. printf("Event Add failed[fd=%d]/n", ev->fd);
55. **else**
56. printf("Event Add OK[fd=%d]/n", ev->fd);
57. }
58. // delete an event from epoll
59. **void** EventDel(**int** epollFd, myevent\_s \*ev)
60. {
61. **struct** epoll\_event epv = {0, {0}};
62. **if**(ev->status != 1) **return**;
63. epv.data.ptr = ev;
64. ev->status = 0;
65. epoll\_ctl(epollFd, EPOLL\_CTL\_DEL, ev->fd, &epv);
66. }
67. **int** g\_epollFd;
68. myevent\_s g\_Events[MAX\_EVENTS+1]; // g\_Events[MAX\_EVENTS] is used by listen fd
69. **void** RecvData(**int** fd, **int** events, **void** \*arg);
70. **void** SendData(**int** fd, **int** events, **void** \*arg);
71. // accept new connections from clients
72. **void** AcceptConn(**int** fd, **int** events, **void** \*arg)
73. {
74. **struct** sockaddr\_in sin;
75. socklen\_t len = **sizeof**(**struct** sockaddr\_in);
76. **int** nfd, i;
77. // accept
78. **if**((nfd = accept(fd, (**struct** sockaddr\*)&sin, &len)) == -1)
79. {
80. **if**(errno != EAGAIN && errno != EINTR)
81. {
82. printf("%s: bad accept", \_\_func\_\_);
83. }
84. **return**;
85. }
86. **do**
87. {
88. **for**(i = 0; i < MAX\_EVENTS; i++)
89. {
90. **if**(g\_Events[i].status == 0)
91. {
92. **break**;
93. }
94. }
95. **if**(i == MAX\_EVENTS)
96. {
97. printf("%s:max connection limit[%d].", \_\_func\_\_, MAX\_EVENTS);
98. **break**;
99. }
100. // set nonblocking
101. **if**(fcntl(nfd, F\_SETFL, O\_NONBLOCK) < 0) **break**;
102. // add a read event for receive data
103. EventSet(&g\_Events[i], nfd, RecvData, &g\_Events[i]);
104. EventAdd(g\_epollFd, EPOLLIN|EPOLLET, &g\_Events[i]);
105. printf("new conn[%s:%d][time:%d]/n", inet\_ntoa(sin.sin\_addr), ntohs(sin.sin\_port), g\_Events[i].last\_active);
106. }**while**(0);
107. }
108. // receive data
109. **void** RecvData(**int** fd, **int** events, **void** \*arg)
110. {
111. **struct** myevent\_s \*ev = (**struct** myevent\_s\*)arg;
112. **int** len;
113. // receive data
114. len = recv(fd, ev->buff, **sizeof**(ev->buff)-1, 0);
115. EventDel(g\_epollFd, ev);
116. **if**(len > 0)
117. {
118. ev->len = len;
119. ev->buff[len] = '/0';
120. printf("C[%d]:%s/n", fd, ev->buff);
121. // change to send event
122. EventSet(ev, fd, SendData, ev);
123. EventAdd(g\_epollFd, EPOLLOUT|EPOLLET, ev);
124. }
125. **else** **if**(len == 0)
126. {
127. close(ev->fd);
128. printf("[fd=%d] closed gracefully./n", fd);
129. }
130. **else**
131. {
132. close(ev->fd);
133. printf("recv[fd=%d] error[%d]:%s/n", fd, errno, strerror(errno));
134. }
135. }
136. // send data
137. **void** SendData(**int** fd, **int** events, **void** \*arg)
138. {
139. **struct** myevent\_s \*ev = (**struct** myevent\_s\*)arg;
140. **int** len;
141. // send data
142. len = send(fd, ev->buff, ev->len, 0);
143. ev->len = 0;
144. EventDel(g\_epollFd, ev);
145. **if**(len > 0)
146. {
147. // change to receive event
148. EventSet(ev, fd, RecvData, ev);
149. EventAdd(g\_epollFd, EPOLLIN|EPOLLET, ev);
150. }
151. **else**
152. {
153. close(ev->fd);
154. printf("recv[fd=%d] error[%d]/n", fd, errno);
155. }
156. }
157. **void** InitListenSocket(**int** epollFd, **short** port)
158. {
159. **int** listenFd = socket(AF\_INET, SOCK\_STREAM, 0);
160. fcntl(listenFd, F\_SETFL, O\_NONBLOCK); // set non-blocking
161. printf("server listen fd=%d/n", listenFd);
162. EventSet(&g\_Events[MAX\_EVENTS], listenFd, AcceptConn, &g\_Events[MAX\_EVENTS]);
163. // add listen socket
164. EventAdd(epollFd, EPOLLIN|EPOLLET, &g\_Events[MAX\_EVENTS]);
165. // bind & listen
166. sockaddr\_in sin;
167. bzero(&sin, **sizeof**(sin));
168. sin.sin\_family = AF\_INET;
169. sin.sin\_addr.s\_addr = INADDR\_ANY;
170. sin.sin\_port = htons(port);
171. bind(listenFd, (**const** sockaddr\*)&sin, **sizeof**(sin));
172. listen(listenFd, 5);
173. }
174. **int** main(**int** argc, **char** \*\*argv)
175. {
176. **short** port = 12345; // default port
177. **if**(argc == 2){
178. port = atoi(argv[1]);
179. }
180. // create epoll
181. g\_epollFd = epoll\_create(MAX\_EVENTS);
182. **if**(g\_epollFd <= 0) printf("create epoll failed.%d/n", g\_epollFd);
183. // create & bind listen socket, and add to epoll, set non-blocking
184. InitListenSocket(g\_epollFd, port);
185. // event loop
186. **struct** epoll\_event events[MAX\_EVENTS];
187. printf("server running:port[%d]/n", port);
188. **int** checkPos = 0;
189. **while**(1){
190. // a simple timeout check here, every time 100, better to use a mini-heap, and add timer event
191. **long** now = time(NULL);
192. **for**(**int** i = 0; i < 100; i++, checkPos++) // doesn't check listen fd
193. {
194. **if**(checkPos == MAX\_EVENTS) checkPos = 0; // recycle
195. **if**(g\_Events[checkPos].status != 1) **continue**;
196. **long** duration = now - g\_Events[checkPos].last\_active;
197. **if**(duration >= 60) // 60s timeout
198. {
199. close(g\_Events[checkPos].fd);
200. printf("[fd=%d] timeout[%d--%d]./n", g\_Events[checkPos].fd, g\_Events[checkPos].last\_active, now);
201. EventDel(g\_epollFd, &g\_Events[checkPos]);
202. }
203. }
204. // wait for events to happen
205. **int** fds = epoll\_wait(g\_epollFd, events, MAX\_EVENTS, 1000);
206. **if**(fds < 0){
207. printf("epoll\_wait error, exit/n");
208. **break**;
209. }
210. **for**(**int** i = 0; i < fds; i++){
211. myevent\_s \*ev = (**struct** myevent\_s\*)events[i].data.ptr;
212. **if**((events[i].events&EPOLLIN)&&(ev->events&EPOLLIN)) // read event
213. {
214. ev->call\_back(ev->fd, events[i].events, ev->arg);
215. }
216. **if**((events[i].events&EPOLLOUT)&&(ev->events&EPOLLOUT)) // write event
217. {
218. ev->call\_back(ev->fd, events[i].events, ev->arg);
219. }
220. }
221. }
222. // free resource
223. **return** 0;
224. }

3、epoll  
3.1、poll(select)的限制  
      Poll函数起源于SVR3，最初局限于流设备，SVR4取消了这种限制。总是来说，poll比select要高效一些，但是，它有可移植性问题，例如，windows就只支持select。  
一个poll的简单例子：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

[复制代码](http://www.cnblogs.com/hustcat/archive/2010/04/10/1708909.html)

#include <stdio.h>  
#include <unistd.h>  
#include <sys/poll.h>  
#define TIMEOUT 5       /\* poll timeout, in seconds \*/  
int main (void)  
{  
        struct pollfd fds[2];  
        int ret;  
        /\* watch stdin for input \*/  
        fds[0].fd = STDIN\_FILENO;  
        fds[0].events = POLLIN;  
        /\* watch stdout for ability to write (almost always true) \*/  
        fds[1].fd = STDOUT\_FILENO;  
        fds[1].events = POLLOUT;  
        /\* All set, block! \*/  
        ret = poll (fds, 2, TIMEOUT \* 1000);  
        if (ret == -1) {  
                perror ("poll");  
                return 1;  
        }  
        if (!ret) {  
                printf ("%d seconds elapsed.\n", TIMEOUT);  
                return 0;  
        }  
        if (fds[0].revents & POLLIN)  
                printf ("stdin is readable\n");  
        if (fds[1].revents & POLLOUT)  
                printf ("stdout is writable\n");  
        return 0;  
}

[复制代码](http://www.cnblogs.com/hustcat/archive/2010/04/10/1708909.html)

     select模型与此类例。内核必须遍历所有监视的描述符，而应用程序也必须遍历所有描述符，检查哪些描述符已经准备好。当描述符成百上千时，会变得非常低效——这是select(poll)模型低效的根源所在。考虑这些情况，2.6以后的内核都引进了epoll模型。

3.2、核心数据结构与接口  
Epoll模型由3个函数构成，epoll\_create、epoll\_ctl和epoll\_wait。  
3.2.1创建epoll实例(Creating a New Epoll Instance)  
     epoll环境通过epoll\_create函数创建：  
     #include <sys/epoll.h>  
      int epoll\_create (int size)  
      调用成功则返回与实例关联的文件描述符，该文件描述符与真实的文件没有任何关系，仅作为接下来调用的函数的句柄。size是给内核的一个提示，告诉内核将要监视的文件描述符的数量，它不是最大值；但是，传递合适的值能够提高系统性能。发生错误时，返回-1。  
例子：

int epfd;  
epfd = epoll\_create (100);  /\* plan to watch ~100 fds \*/  
if (epfd < 0)  
        perror ("epoll\_create");

3.2.2、控制epoll(Controlling Epoll)  
通过epoll\_ctl，可以加入文件描述符到epoll环境或从epoll环境移除文件描述符。

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

[复制代码](http://www.cnblogs.com/hustcat/archive/2010/04/10/1708909.html)

#include <sys/epoll.h>  
int epoll\_ctl (int epfd,  
               int op,  
               int fd,  
               struct epoll\_event \*event);  
  
struct epoll\_event {  
        \_ \_u32 events;  /\* events \*/  
        union {  
                void \*ptr;  
                int fd;  
                \_ \_u32 u32;  
                \_ \_u64 u64;  
        } data;  
};

[复制代码](http://www.cnblogs.com/hustcat/archive/2010/04/10/1708909.html)

epfd为epoll\_create返回的描述符。op表示对描述符fd采取的操作，取值如下：  
EPOLL\_CTL\_ADD  
Add a monitor on the file associated with the file descriptor fd to the epoll instance associated with epfd, per the events defined in event.

EPOLL\_CTL\_DEL  
Remove a monitor on the file associated with the file descriptor fd from the epollinstance associated with epfd.  
EPOLL\_CTL\_MOD  
Modify an existing monitor of fd with the updated events specified by event.

epoll\_event结构中的events字段，表示对该文件描述符所关注的事件，它的取值如下：  
EPOLLET  
Enables edge-triggered behavior for the monitor of the file .The default behavior is level-  
triggered.  
EPOLLHUP  
A hangup occurred on the file. This event is always monitored, even if it’s not specified.  
EPOLLIN  
The file is available to be read from without blocking.  
EPOLLONESHOT  
After an event is generated and read, the file is automatically no longer monitored.A new event mask must be specified via EPOLL\_CTL\_MOD to reenable the watch.  
EPOLLOUT  
The file is available to be written to without blocking.  
EPOLLPRI  
There is urgent out-of-band data available to read.  
而epoll\_event结构中的fd是epoll高效的根源所在，当描述符准备好。应用程序不用遍历所有描述符，而只用检查发生事件的描述符。  
将一个描述符加入epoll环境：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

[复制代码](http://www.cnblogs.com/hustcat/archive/2010/04/10/1708909.html)

struct epoll\_event event;  
int ret;  
event.data.fd = fd; /\* return the fd to us later \*/  
event.events = EPOLLIN | EPOLLOUT;  
ret = epoll\_ctl (epfd, EPOLL\_CTL\_ADD, fd, &event);  
if (ret)  
        perror ("epoll\_ctl");

[复制代码](http://www.cnblogs.com/hustcat/archive/2010/04/10/1708909.html)

3.2.3、等待事件(Waiting for Events with Epoll)

#include <sys/epoll.h>  
int epoll\_wait (int epfd,  
                struct epoll\_event \*events,  
                int maxevents,  
                int timeout);  
等待事件的产生，类似于select()调用。参数events用来从内核得到事件的集合，maxevents告之内核这个events有多大，这个 maxevents的值不能大于创建epoll\_create()时的size，参数timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有 说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表示已超时。  
一个简单示例：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

[复制代码](http://www.cnblogs.com/hustcat/archive/2010/04/10/1708909.html)

#define MAX\_EVENTS    64  
struct epoll\_event \*events;  
int nr\_events, i, epfd;  
events = malloc (sizeof (struct epoll\_event) \* MAX\_EVENTS);  
if (!events) {  
        perror ("malloc");  
        return 1;  
}  
nr\_events = epoll\_wait (epfd, events, MAX\_EVENTS, -1);  
if (nr\_events < 0) {  
        perror ("epoll\_wait");  
        free (events);  
        return 1;  
}  
//只需要检查发生事件的文件描述符,而不需要遍历所有描述符  
for (i = 0; i < nr\_events; i++) {  
        printf ("event=%ld on fd=%d\n",  
                events[i].events,  
                events[i].data.fd);  
        /\*  
         \* We now can, per events[i].events, operate on  
         \* events[i].data.fd without blocking.  
         \*/  
}  
free (events);

[复制代码](http://www.cnblogs.com/hustcat/archive/2010/04/10/1708909.html)

3.2.4、epoll的典型用法

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

[复制代码](http://www.cnblogs.com/hustcat/archive/2010/04/10/1708909.html)

struct epoll\_event ev, \*events;  
for(;;) {  
    nfds = epoll\_wait(kdpfd, events, maxevents, -1);  
    for(n = 0; n < nfds; ++n) {  
        if(events[n].data.fd == listener) {  
            //新的连接  
            client = accept(listener, (struct sockaddr \*) &local,  
                            &addrlen);  
            if(client < 0){  
                perror("accept");  
                continue;  
            }  
            setnonblocking(client);  
            ev.events = EPOLLIN | EPOLLET;  
            ev.data.fd = client;  
        // 设置好event之后，将这个新的event通过epoll\_ctl加入到epoll的监听队列里面  
            if (epoll\_ctl(kdpfd, EPOLL\_CTL\_ADD, client, &ev) < 0) {  
                fprintf(stderr, "epoll set insertion error: fd=%d0,  
                        client);  
                return -1;  
            }  
        }  
        else  
            do\_use\_fd(events[n].data.fd);  
    }  
}

[复制代码](http://www.cnblogs.com/hustcat/archive/2010/04/10/1708909.html)

3.3、综合示例

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

[复制代码](http://www.cnblogs.com/hustcat/archive/2010/04/10/1708909.html)

//echo\_epoll\_server.c  
#include "echo.h"  
#include <sys/epoll.h>  
#include <fcntl.h>  
  
#define EVENT\_ARR\_SIZE 20  
#define EPOLL\_SIZE     20  
  
void setnonblocking(  
    int sockfd  
);  
  
int  
main(int argc, char \*\*argv)  
{  
    int        i,  listenfd, connfd, sockfd, epfd;  
    ssize\_t        n;  
    char            buf[MAXLINE];  
    socklen\_t        clilen;  
    struct sockaddr\_in    cliaddr, servaddr;  
    struct epoll\_event ev, evs[EVENT\_ARR\_SIZE];  
    int   nfds;  
  
    if((listenfd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)  
        err\_sys("create socket error!\n");  
    setnonblocking(listenfd);  
  
    epfd = epoll\_create(EPOLL\_SIZE);  
    ev.data.fd = listenfd;  
    ev.events = EPOLLIN | EPOLLET;  
    if(epoll\_ctl(epfd, EPOLL\_CTL\_ADD, listenfd, &ev) < 0)  
        err\_sys("epoll\_ctl listenfd error!\n");  
      
    bzero(&servaddr, sizeof(servaddr));  
    servaddr.sin\_family      = AF\_INET;  
    //servaddr.sin\_addr.s\_addr = INADDR\_ANY;  
    servaddr.sin\_addr.s\_addr = inet\_addr("211.67.28.128");  
    servaddr.sin\_port        = htons(SERV\_PORT);  
  
    if(bind(listenfd, (struct sockaddr\*) &servaddr, sizeof(servaddr)) < 0)  
        err\_sys("bind error!\n");  
  
    if(listen(listenfd, LISTENQ) < 0)  
        err\_sys("listen error!\n");  
  
    printf("server is listening....\n");  
  
    for ( ; ; ) {  
        if((nfds = epoll\_wait(epfd, evs, EVENT\_ARR\_SIZE, -1)) < 0)  
            err\_sys("epoll\_wait error!\n");  
  
        for(i = 0; i < nfds; i++)  
        {  
                if(evs[i].data.fd == listenfd)  
                {  
                    clilen = sizeof(cliaddr);  
                    connfd = accept(listenfd, (struct sockaddr\*) &cliaddr, &clilen);  
                    if(connfd < 0)  
                        continue;  
                          
                    setnonblocking(connfd);  
                    ev.data.fd = connfd;  
                    ev.events = EPOLLIN | EPOLLET;  
                    if (epoll\_ctl(epfd, EPOLL\_CTL\_ADD, connfd, &ev) < 0)  
                        err\_sys("epoll\_ctl connfd error!\n");              
                }  
                else if(evs[i].events & EPOLLIN)  
                {  
                    sockfd = evs[i].data.fd;  
                    if (sockfd < 0)  
                        continue;  
                    if ( (n = read(sockfd, buf, MAXLINE)) == 0) {  
                        epoll\_ctl(epfd, EPOLL\_CTL\_DEL, sockfd, &ev);  
                        close(sockfd);  
                        evs[i].data.fd = -1;  
                    }   
                    else if(n < 0)  
                        err\_sys("read socket error!\n");  
                    else  
                    {  
                        printf("write %d bytes\n", n);  
                        write(sockfd, buf, n);  
                    }  
                }  
                else  
                    printf("other event!\n");  
        }  
    }  
    return 0;  
}  
  
  
void setnonblocking(  
    int sockfd  
)  
{  
    int flag;  
      
    flag = fcntl(sockfd, F\_GETFL);  
    if(flag < 0)  
            err\_sys("fcnt(F\_GETFL) error!\n");  
    flag |= O\_NONBLOCK;  
    if(fcntl(sockfd, F\_SETFL, flag) < 0)  
        err\_sys("fcon(F\_SETFL) error!\n");  
}  
  
//echo.h  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
  
#include <unistd.h>  
#include <stdlib.h>  
#include <string.h>  
#include <stdio.h>  
#include <errno.h>  
  
  
#define SERV\_PORT     9877  
#define MAXLINE        4096  
#define LISTENQ        5  
  
void  
err\_sys(const char \*fmt, ...);  
  
ssize\_t                          
readn(int fd, void \*vptr, size\_t n);

[复制代码](http://www.cnblogs.com/hustcat/archive/2010/04/10/1708909.html)

**补充部分：**

select()系统调用提供一个机制来实现同步多元I/O：

|  |
| --- |
| #include <sys/time.h> #include <sys/types.h> #include <unistd.h>  int select (int n, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);  FD\_CLR(int fd, fd\_set \*set); FD\_ISSET(int fd, fd\_set \*set); FD\_SET(int fd, fd\_set \*set); FD\_ZERO(fd\_set \*set); |

调用select()将阻塞，直到指定的文件描述符准备好执行I/O，或者可选参数timeout指定的时间已经过去。  
监视的文件描述符分为三类set，每一种对应等待不同的事件。readfds中列出的文件描述符被监视是否有数据可供读取（如果读取操作完成则不会阻塞）。writefds中列出的文件描述符则被监视是否写入操作完成而不阻塞。最后，exceptfds中列出的文件描述符则被监视是否发生异常，或者无法控制的数据是否可用（这些状态仅仅应用于套接字）。这三类set可以是NULL，这种情况下select()不监视这一类事件。  
select()成功返回时，每组set都被修改以使它只包含准备好I/O的文件描述符。例如，假设有两个文件描述符，值分别是7和9，被放在readfds中。当select()返回时，如果7仍然在set中，则这个文件描述符已经准备好被读取而不会阻塞。如果9已经不在set中，则读取它将可能会阻塞（我说可能是因为数据可能正好在select返回后就可用，这种情况下，下一次调用select()将返回文件描述符准备好读取）。  
第一个参数n，等于所有set中最大的那个文件描述符的值加1。因此，select()的调用者负责检查哪个文件描述符拥有最大值，并且把这个值加1再传递给第一个参数。  
timeout参数是一个指向timeval结构体的指针，timeval定义如下：

|  |
| --- |
| #include <sys/time.h> struct timeval { long tv\_sec; /\* seconds \*/ long tv\_usec; /\* 10E-6 second \*/ }; |

如果这个参数不是NULL，则即使没有文件描述符准备好I/O，select()也会在经过tv\_sec秒和tv\_usec微秒后返回。当select()返回时，timeout参数的状态在不同的系统中是未定义的，因此每次调用select()之前必须重新初始化timeout和文件描述符set。实际上，当前版本的Linux会自动修改timeout参数，设置它的值为剩余时间。因此，如果timeout被设置为5秒，然后在文件描述符准备好之前经过了3秒，则这一次调用select()返回时tv\_sec将变为2。  
如果timeout中的两个值都设置为0，则调用select()将立即返回，报告调用时所有未决的事件，但不等待任何随后的事件。  
文件描述符set不会直接操作，一般使用几个助手宏来管理。这允许Unix系统以自己喜欢的方式来实现文件描述符set。但大多数系统都简单地实现set为位数组。FD\_ZERO移除指定set中的所有文件描述符。每一次调用select()之前都应该先调用它。  
fd\_set writefds;  
FD\_ZERO(&writefds);  
FD\_SET添加一个文件描述符到指定的set中，FD\_CLR则从指定的set中移除一个文件描述符：  
FD\_SET(fd, &writefds); /\* add 'fd' to the set \*/  
FD\_CLR(fd, &writefds); /\* oops, remove 'fd' from the set \*/  
设计良好的代码应该永远不使用FD\_CLR，而且实际情况中它也确实很少被使用。  
FD\_ISSET测试一个文件描述符是否指定set的一部分。如果文件描述符在set中则返回一个非0整数，不在则返回0。FD\_ISSET在调用select()返回之后使用，测试指定的文件描述符是否准备好相关动作：  
if (FD\_ISSET(fd, &readfds))  
/\* 'fd' is readable without blocking! \*/  
因为文件描述符set是静态创建的，它们对文件描述符的最大数目强加了一个限制，能够放进set中的最大文件描述符的值由FD\_SETSIZE指定。在Linux中，这个值是1024。本章后面我们还将看到这个限制的衍生物。  
**返回值和错误代码**  
select()成功时返回准备好I/O的文件描述符数目，包括所有三个set。如果提供了timeout，返回值可能是0；错误时返回-1，并且设置errno为下面几个值之一：  
EBADF，给某个set提供了无效文件描述符。  
EINTR，等待时捕获到信号，可以重新发起调用。  
EINVAL，参数n为负数，或者指定的timeout非法。  
ENOMEM，不够可用内存来完成请求。  
--------------------------------------------------------------------------------------------------------------

poll()系统调用是System V的多元I/O解决方案。它解决了select()的几个不足，尽管select()仍然经常使用（多数还是出于习惯，或者打着可移植的名义）：

|  |
| --- |
| #include <sys/poll.h> int poll (struct pollfd \*fds, unsigned int nfds, int timeout); |

和select()不一样，poll()没有使用低效的三个基于位的文件描述符set，而是采用了一个单独的结构体pollfd数组，由fds指针指向这个组。pollfd结构体定义如下：

|  |
| --- |
| #include <sys/poll.h>  struct pollfd { int fd; /\* file descriptor \*/ short events; /\* requested events to watch \*/ short revents; /\* returned events witnessed \*/ }; |

每一个pollfd结构体指定了一个被监视的文件描述符，可以传递多个结构体，指示poll()监视多个文件描述符。每个结构体的events域是监视该文件描述符的事件掩码，由用户来设置这个域。revents域是文件描述符的操作结果事件掩码。内核在调用返回时设置这个域。events域中请求的任何事件都可能在revents域中返回。合法的事件如下：  
POLLIN，有数据可读。  
POLLRDNORM，有普通数据可读。  
POLLRDBAND，有优先数据可读。  
POLLPRI，有紧迫数据可读。  
POLLOUT，写数据不会导致阻塞。  
POLLWRNORM，写普通数据不会导致阻塞。  
POLLWRBAND，写优先数据不会导致阻塞。  
POLLMSG，SIGPOLL消息可用。  
此外，revents域中还可能返回下列事件：  
POLLER，指定的文件描述符发生错误。  
POLLHUP，指定的文件描述符挂起事件。  
POLLNVAL，指定的文件描述符非法。  
这些事件在events域中无意义，因为它们在合适的时候总是会从revents中返回。使用poll()和select()不一样，你不需要显式地请求异常情况报告。POLLIN | POLLPRI等价于select()的读事件，POLLOUT | POLLWRBAND等价于select()的写事件。POLLIN等价于POLLRDNORM | POLLRDBAND，而POLLOUT则等价于POLLWRNORM。  
例如，要同时监视一个文件描述符是否可读和可写，我们可以设置events为POLLIN | POLLOUT。在poll返回时，我们可以检查revents中的标志，对应于文件描述符请求的events结构体。如果POLLIN事件被设置，则文件描述符可以被读取而不阻塞。如果POLLOUT被设置，则文件描述符可以写入而不导致阻塞。这些标志并不是互斥的：它们可能被同时设置，表示这个文件描述符的读取和写入操作都会正常返回而不阻塞。timeout参数指定等待的毫秒数，无论I/O是否准备好，poll都会返回。timeout指定为负数值表示无限超时；timeout为0指示poll调用立即返回并列出准备好I/O的文件描述符，但并不等待其它的事件。这种情况下，poll()就像它的名字那样，一旦选举出来，立即返回。  
**返回值和错误代码**  
成功时，poll()返回结构体中revents域不为0的文件描述符个数；如果在超时前没有任何事件发生，poll()返回0；失败时，poll()返回-1，并设置errno为下列值之一：  
EBADF，一个或多个结构体中指定的文件描述符无效。  
EFAULT，fds指针指向的地址超出进程的地址空间。  
EINTR，请求的事件之前产生一个信号，调用可以重新发起。  
EINVAL，nfds参数超出PLIMIT\_NOFILE值。  
ENOMEM，可用内存不足，无法完成请求。

以上内容来自《OReilly.Linux.System.Programming - Talking.Directly.to.the.Kernel.and.C.Library.2007》

**Epoll的优点：**  
1.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。

2.IO效率不随FD数目增加而线性下降  
    传统的select/poll另一个致命弱点就是当你拥有一个很大的socket集合，不过由于网络延时，任一时间只有部分的socket是"活跃"的，但是select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是epoll不存在这个问题，它只会对"活跃"的socket进行操作---这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有"活跃"的socket才会主动的去调用 callback函数，其他idle状态socket则不会，在这点上，epoll实现了一个"伪"AIO，因为这时候推动力在os内核。在一些 benchmark中，如果所有的socket基本上都是活跃的---比如一个高速LAN环境，epoll并不比select/poll有什么效率，相反，如果过多使用epoll\_ctl,效率相比还有稍微的下降。但是一旦使用idle connections模拟WAN环境,epoll的效率就远在select/poll之上了。

3.使用mmap加速内核与用户空间的消息传递。  
    这点实际上涉及到epoll的具体实现了。无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。而如果你想我一样从2.5内核就关注epoll的话，一定不会忘记手工 mmap这一步的。

4.内核微调  
    这一点其实不算epoll的优点了，而是整个linux平台的优点。也许你可以怀疑linux平台，但是你无法回避linux平台赋予你微调内核的能力。比如，内核TCP/IP协议栈使用内存池管理sk\_buff结构，那么可以在运行时期动态调整这个内存pool(skb\_head\_pool)的大小--- 通过echo XXXX>/proc/sys/net/core/hot\_list\_length完成。再比如listen函数的第2个参数(TCP完成3次握手的数据包队列长度)，也可以根据你平台内存大小动态调整。更甚至在一个数据包面数目巨大但同时每个数据包本身大小却很小的特殊系统上尝试最新的NAPI网卡驱动架构。

########################################################

## select/epoll的特点

select的特点：select 选择句柄的时候，是遍历所有句柄，也就是说句柄有事件响应时，select需要遍历所有句柄才能获取到哪些句柄有事件通知，因此效率是非常低。但是如果连接很少的情况下， select和epoll的LT触发模式相比， 性能上差别不大。  
这 里要多说一句，select支持的句柄数是有限制的， 同时只支持1024个，这个是句柄集合限制的，如果超过这个限制，很可能导致溢出，而且非常不容易发现问题， TAF就出现过这个问题， 调试了n天，才发现：）当然可以通过修改linux的socket内核调整这个参数。  
epoll的特点：epoll对于句柄事件的选择不是遍历的，是事件响应的，就是句柄上事件来就马上选择出来，不需要遍历整个句柄链表，因此效率非常高，内核将句柄用红黑树保存的。相比于select，epoll最大的好处在于它不会随着监听fd数目的增长而降低效率。因为在内核中的select实现中，它是采用轮询来处理的，轮询的fd数目越多，自然耗时越多。并且，在linux/posix\_types.h头文件有这样的声明：  
#define \_\_FD\_SETSIZE    1024  
表示select最多同时监听1024个fd，当然，可以通过修改头文件再重编译内核来扩大这个数目，但这似乎并不治本。

对于epoll而言还有ET和LT的区别，LT表示水平触发，ET表示边缘触发，两者在性能以及代码实现上差别也是非常大的。

## epoll的LT和ET的区别

LT：水平触发，效率会低于ET触发，尤其在大并发，大流量的情况下。但是LT对代码编写要求比较低，不容易出现问题。LT模式服务编写上的表现是：只要有数据没有被获取，内核就不断通知你，因此不用担心事件丢失的情况。  
ET：边缘触发，效率非常高，在并发，大流量的情况下，会比LT少很多epoll的系统调用，因此效率高。但是对编程要求高，需要细致的处理每个请求，否则容易发生丢失事件的情况。

## epoll相关ＡＰＩ

epoll的接口非常简单，一共就三个函数：

### 1. int epoll\_create(int size);

创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大。这个参数不同于select()中的第一个参数，给出最大监听的fd+1的值。需要注意的是，当创建好epoll句柄后，它就是会占用一个fd值，在linux下如果查看/proc/进程id/fd/，是能够看到这个fd的，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

### 2. int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

epoll的事件注册函数，它不同与select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。第一个参数是epoll\_create()的返回值，第二个参数表示动作，用三个宏来表示：  
EPOLL\_CTL\_ADD：注册新的fd到epfd中；  
EPOLL\_CTL\_MOD：修改已经注册的fd的监听事件；  
EPOLL\_CTL\_DEL：从epfd中删除一个fd；  
第三个参数是需要监听的fd，第四个参数是告诉内核需要监听什么事，struct epoll\_event结构如下：  
  
typedef union epoll\_data {  
    void \*ptr;  
    int fd;  
    \_\_uint32\_t u32;  
    \_\_uint64\_t u64;  
} epoll\_data\_t;  
  
  
struct epoll\_event {  
    \_\_uint32\_t events; /\* Epoll events \*/  
    epoll\_data\_t data; /\* User data variable \*/  
};  
  
events可以是以下几个宏的集合：  
EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；  
EPOLLOUT：表示对应的文件描述符可以写；  
EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；  
EPOLLERR：表示对应的文件描述符发生错误；  
EPOLLHUP：表示对应的文件描述符被挂断；  
EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。  
EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

### 3. int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

等待事件的产生，类似于select()调用。参数events用来从内核得到事件的集合，maxevents告之内核这个events有多大，这个 maxevents的值不能大于创建epoll\_create()时的size，参数timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表示已超时。

## 示例代码

### epoll服务器

1. #include <sys/epoll.h>
2. #include <netinet/in.h>
3. #include <sys/types.h>          /\* See NOTES \*/
4. #include <sys/socket.h>
5. #include <string.h>
6. #include <stdio.h>
7. #include <unistd.h>
8. #include <fcntl.h>
10. #include <errno.h>
11. #include <stdlib.h>
12. **typedef** **struct** sockaddr\_in sockaddr\_in ;
13. **typedef** **struct** sockaddr     sockaddr ;
15. #define SER\_PORT    8080
17. **int** nonblock(**int** fd){
18. **int** opt ;
19. opt = fcntl(fd,F\_GETFL);
20. opt |= O\_NONBLOCK ;
21. **return** fcntl(fd,F\_SETFL,opt);
22. }
24. **int** main(**int** argc,**char**\*\*argv){
25. sockaddr\_in srv, cli ;
26. **int** listen\_fd ,con\_fd ;
27. socklen\_t  len;
28. **int** res ,nsize,ws;
29. **char** buf[255];
31. **int** epfd,ers;
32. **struct** epoll\_event evn,events[50];
33. **int** i;
35. bzero(&srv,**sizeof**(srv));
36. bzero(&cli,**sizeof**(cli));
37. srv.sin\_port= SER\_PORT ;
38. srv.sin\_family = AF\_INET ;
39. listen\_fd = socket(AF\_INET,SOCK\_STREAM,0);
41. **int** yes = 1;
42. setsockopt(listen\_fd,SOL\_SOCKET,SO\_REUSEADDR,&yes,**sizeof**(**int**));
44. **if**(bind(listen\_fd,(sockaddr\*)&srv,**sizeof**(sockaddr))<0)  {
45. perror("bind");
46. exit(0);
47. }
48. listen(listen\_fd,100);
49. nonblock(listen\_fd);
50. epfd = epoll\_create(200);
51. evn.events = EPOLLIN|EPOLLET ;
52. evn.data.fd = listen\_fd;
53. epoll\_ctl(epfd,EPOLL\_CTL\_ADD,listen\_fd,&evn);
54. **static** **int** count ;
55. **while**(1){
56. ers = epoll\_wait(epfd,events,100,5000);
57. **if**(ers<0 ){
58. perror("epoll\_wait:");exit(0);
59. }**else** **if**(ers==0){
60. printf("time out:%d\n",count++);
61. **continue** ;
62. }
63. **for**(i=0;i<ers;i++){
64. **if**(events[i].data.fd == listen\_fd){
65. con\_fd = accept(listen\_fd,(sockaddr\*)&cli ,&len);
66. nonblock(con\_fd);
67. printf("connect from:%s\n",inet\_ntoa(cli.sin\_addr));
68. evn.data.fd = con\_fd;
69. evn.events = EPOLLIN | EPOLLET ;
70. epoll\_ctl(epfd,EPOLL\_CTL\_ADD,con\_fd,&evn);
72. }**else** **if**(events[i].events & EPOLLIN){
74. nsize = 0;
75. **while**((res=read(events[i].data.fd,buf+nsize,**sizeof**(buf)-1))>0){
76. nsize+= res;
77. }
78. **if**(res==0){
79. epoll\_ctl(epfd,EPOLL\_CTL\_DEL,events[i].data.fd,NULL);
80. printf("a client over\n");
81. close(con\_fd);
82. **continue** ;
83. }**else** **if**(res<0 && errno!=EAGAIN){
84. perror("read");
85. **continue** ;
86. }
87. buf[nsize]=0;
88. evn.data.fd = events[i].data.fd;
89. evn.events=EPOLLOUT|EPOLLET ;
90. epoll\_ctl(epfd,EPOLL\_CTL\_MOD,events[i].data.fd,&evn);
92. }**else** **if**(events[i].events & EPOLLOUT){
93. nsize = strlen(buf);
94. ws = 0;
95. **while**(nsize>0){
96. ws=write(events[i].data.fd,buf,nsize);
97. nsize-=ws;
98. }
99. evn.data.fd = events[i].data.fd;
100. evn.events=EPOLLIN|EPOLLET ;
101. epoll\_ctl(epfd,EPOLL\_CTL\_MOD,events[i].data.fd,&evn);
102. }**else**{
103. printf("others\n");
105. }
106. }
108. }
110. close(listen\_fd);
112. **return** 0;
113. }

### 客户端测试代码：

1. #include <sys/epoll.h>
2. #include <netinet/in.h>
3. #include <sys/types.h>          /\* See NOTES \*/
4. #include <sys/socket.h>
5. #include <strings.h>
6. #include <stdio.h>
7. #include <stdlib.h>
8. #include <unistd.h>
9. **typedef** **struct** sockaddr\_in sockaddr\_in ;
10. **typedef** **struct** sockaddr     sockaddr ;
12. #define SER\_PORT    8080
13. #define IP\_ADDR     "10.33.28.230"
15. **int** main(**int** argc,**char**\*\*argv){
16. sockaddr\_in srv, cli ;
17. **int** listen\_fd ,con\_fd ;
18. socklen\_t  len;
19. **int** res,ws ;
20. **char** buf[255];
22. bzero(&srv,**sizeof**(srv));
23. bzero(&cli,**sizeof**(cli));
24. srv.sin\_port= SER\_PORT ;
25. srv.sin\_family = AF\_INET ;
26. inet\_pton(AF\_INET,IP\_ADDR,&srv.sin\_addr);
28. listen\_fd = socket(AF\_INET,SOCK\_STREAM,0);
29. **if**(connect(listen\_fd,(sockaddr\*)&srv,**sizeof**(sockaddr))<0){
30. perror("connect");
31. exit(0);
32. }
33. **while**(1){
34. res = read(STDIN\_FILENO,buf,**sizeof**(buf)-1);
35. ws = write(listen\_fd,buf,res);
36. res = read(listen\_fd,buf,**sizeof**(buf)-1);
37. ws = write(STDOUT\_FILENO,buf,res);
38. }
40. close(listen\_fd);
42. **return** 0;
43. }

参考：

<http://www.kuqin.com/linux/20120912/330528.htm>  
<http://blog.sina.com.cn/s/blog_81c2545a01018m78.html>

<http://blog.csdn.net/ljx0305/article/details/4065058>