

赠书 | AI专栏 (AI圣经! 《深度学习》中文版) 每周荐书: Kotlin、分布式、Keras (评论送书) 【获奖公布】征文 | 你会为 AI 转型么?

## Linux网络编程--epoll 模型原理详解以及实例

标签: 网络编程 epoll I-O多路复用 epoll-wait epoll-ctl

2015-10-08 16:53 1786人阅读 评论(2) 收藏 举报

分类: Linux高性能网络编程 (32)

版权声明: 【本文为博主原创】 【未经博主允许不得转载】

目录(?)

### 1.简介

Linux I/O多路复用技术在比较多的TCP网络服务器中有使用，即比较多的用到select函数。linux 2.6内核中有提高网络I/O性能的新方法，即epoll。

epoll是什么？按照man手册的说法是为处理大批量句柄而作了改进的poll。要使用epoll只需要以下的三个系统函数调用：epoll\_create(2)，epoll\_ctl(2)，epoll\_wait(2)。

### 2.select模型的缺陷

- (1) 在Linux内核中，select所用到的FD\_SET是有限的  
内核中有个参数\_FD\_SETSIZE定义了每个FD\_SET的句柄个数：#define \_FD\_SETSIZE 1024。也就是说，如果想要同时检测1025个句柄的可读状态是不可能用select实现的；或者同时检测1025个句柄的可写状态也是不可能的。
- (2) 内核中实现select是使用轮询方法  
每次检测都会遍历所有FD\_SET中的句柄，显然select函数的执行时间与FD\_SET中句柄的个数有一个比例关系，即select要检测的句柄数越多就会越费时

### 3.Windows IOCP模型的缺陷

windows完成端口实现的AIO，实际上也只是使用内部用线程池实现的，最后的结果是IO有个线程池，你的应用程序也需要一个线程池。很多文档其实已经指出了这引发的线程context-switch所带来的代价。

### 4.EPOLL模型的优点

- (1) 支持一个进程打开大数目的socket描述符(FD)  
epoll没有select模型中的限制，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select 所支持的2048。下面是我的小PC机上的显示：  
pt@ubuntu:~\$ cat /proc/sys/fs/file-max  
6815744  
那么对于服务器而言,这个数目会更大。
- (2) IO效率不随FD数目增加而线性下降  
传统select/poll的另一个致命弱点就是当你拥有一个很大的socket集合，由于网络得延时，使得任一时间只有部分的socket是“活跃”的，而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。于是，只有“活跃”的socket才会主动去调用callback函数，其他idle状态的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也不比select/poll低多少效率，但若过多使用的调用epoll\_ctl，效率稍微有些下降。然而一旦使用idle connections模拟WAN环境，那么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  
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显得很重要。在这点上，epoll是通过内核于用户空间mmap同一块内存实现。

### 5.EPOLL模型的工作模式

- (1) LT模式  
LT：level triggered，这是缺省的工作方式，同时支持block和no-block socket，在这种模式中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表。
- (2) ET模式  
LT：edge-triggered，这是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核就通过epoll告诉你，然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作而导致那个文件描述符不再是就绪状态(比如你在发送，接收或是接受请求，或者发送接收的数据少于一定量时导致了一个EWOULDBLOCK 错误)。但是请注意，如果一直不对这个fd作IO操作(从而导致它再次变成未就绪)，内核就不会发送更多的通知(only once)。不过在TCP协议中，ET模式的加速效用仍需要更多的benchmark确认。

### 6.EPOLL模型的使用方法

epoll用到的所有函数都是在头文件sys/epoll.h中声明的，下面简要说明所用到的数据结构和函数：

(1) epoll\_data、epoll\_data\_t、epoll\_event

```
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 */
```

个人资料



奔跑吧，行者



访问: 168863次

积分: 3600

等级: BLOG > 5

排名: 第8947名

原创: 172篇

转载: 10篇

译文: 2篇

评论: 25条

友情链接

奔跑吧，程序员

文章搜索

博客专栏



苹果移动iOS开发  
文章: 0篇  
阅读: 0



Linux c/c++常用源代码  
文章: 3篇  
阅读: 2563



Linux网络编程  
文章: 16篇  
阅读: 19354

文章分类

- C/C++语言编程 (29)
- JavaSE基础编程 (6)
- IOS移动开发实践 (13)
- linux进程线程基础 (4)
- Linux高性能网络编程 (33)
- linux多进程并发编程 (8)
- linux多线程并发编程 (0)
- 数据结构与算法 (21)
- Unix/Linux系统 (18)
- Linux&shell编程 (18)
- mysql数据库 (6)
- DB2数据库 (14)
- vi/vim的使用技巧 (4)
- 编码的测试和技巧 (2)
- 内存的使用与管理 (1)
- libevent开源介绍 (5)
- 常用源代码 (6)
- 面试题剖析 (8)
- 杂谈 (2)

文章存档

2016年11月 (2)

2016年10月 (4)



- Linux网络编程--sendfile (4360)
- 使用C语言调用mysql数据库 (4027)
- 用C语言操作MySQL数据库 (3713)
- DB2 insert语句三种格式 (3645)
- Linux网络编程--recv函数 (3360)
- Linux网络编程--服务端判 (3216)
- 用static声明的函数和变量 (2712)
- objective-c中如何从UIlm (2361)



```
epoll_data_t data; /* User data variable */
};
```

结构体epoll\_event 被用于注册所感兴趣的事件和回传所发生待处理的事件。epoll\_event 结构体的events字段是表示感兴趣的事件和被触发的事件，可能的取值为：

EPOLLIN：表示对应的文件描述符可以读；

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；

EPLLERR：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：表示对应的文件描述符有事件发生；

联合体epoll\_data用来保存触发事件的某个文件描述符相关的数据。例如一个client连接到服务器，服务器通过调用accept函数可以得到于这个client对应的socket文件描述符，可以把这文件描述符赋给epoll\_data的fd字段，以便后面的读写操作在这个文件描述符上进行。

(2)epoll\_create

函数声明：intepoll\_create(intsize)

函数说明：该函数生成一个epoll专用的文件描述符，其中的参数是指定生成描述符的最大范围。

(3) epoll\_ctl函数

函数声明：intepoll\_ctl(int epfd,int op, int fd, struct epoll\_event \*event)

函数说明：该函数用于控制某个文件描述符上的事件，可以注册事件、修改事件、删除事件。

epfd：由 epoll\_create 生成的epoll专用的文件描述符；

op：要进行的操作，可能的取值EPOLL\_CTL\_ADD 注册、EPOLL\_CTL\_MOD 修改、EPOLL\_CTL\_DEL 删除；

fd：关联的文件描述符；

event：指向epoll\_event的指针；

如果调用成功则返回0，不成功则返回-1。

(4) epoll\_wait函数

函数声明：int epoll\_wait(int epfd, structepoll\_event \* events, int maxevents, int timeout)

函数说明：该函数用于轮询I/O事件的发生。

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。

## 7 设计思路及模板

首先通过create\_epoll(int maxfds)来创建一个epoll的句柄，其中maxfds为你的epoll所支持的最大句柄数。这个函数会返回一个新的epoll句柄，之后的所有操作都将通过这个句柄来进行操作。在用完之后，记得用close()来关闭这个创建出来的epoll句柄。

然后在你的网络主循环里面，调用epoll\_wait(int epfd, epoll\_event events, int max\_events,int timeout)来查询所有的网络接口，看哪一个可以读，哪一个可以写。基本的语法为：

```
nfds = epoll_wait(kdpfd, events, maxevents, -1);
```

其中kdpfd为用epoll\_create创建之后的句柄，events是一个epoll\_event\*的指针，当epoll\_wait函数操作成功之后，events里面将储存所有的读写事件。max\_events是当前需要监听的所有socket句柄数。最后一个timeout参数指示 epoll\_wait的超时条件，为0时表示马上返回；为-1时表示函数会一直等下去直到有事件返回；为任意正整数时表示等这么长的时间，如果一直没有事件，则会返回。一般情况下如果网络主循环是单线程的话，可以用-1来等待，这样可以保证一些效率，如果是和主循环在同一个线程的话，则可以用0来保证主循环的效率。epoll\_wait返回之后，应该进入一个循环，以便遍历所有的事件。

对epoll 的操作就这么简单，总共不过4个API：epoll\_create, epoll\_ctl,epoll\_wait和close。以下是man中的一个例子。

```
1 struct epoll_event ev, *events;
2 for(;;)
3 {
4     nfds = epoll_wait(kdpfd, events, maxevents, -1);    //等待IO事件
5     for(n = 0; n < nfds; ++n)
6     {
7         //如果是主socket的事件，则表示有新连接进入，需要进行新连接的处理。
8         if(events[n].data.fd == listener)
9         {
10             client = accept(listener, (struct sockaddr *) &local,  &addrlen);
11 if(client < 0)
12         {
13             perror("accept error");
14             continue;
15         }
16         // 将新连接置于非阻塞模式
17         setnonblocking(client);
18         ev.events = EPOLLIN | EPOLLET;
19         //注意这里的参数EPOLLIN | EPOLLET并没有设置对写socket的监听，
20         //如果有写操作的话，这个时候epoll是不会返回事件的，
21         //如果要对写操作也监听的话，应该是EPOLLIN | EPOLLOUT | EPOLLET。
22         // 并且将新连接也加入EPOLL的监听队列
23         ev.data.fd = client;
24         // 设置好event之后，将这个新的event通过epoll_ctl
25         if (epoll_ctl(kdpfd, EPOLL_CTL_ADD, client, &ev) < 0)
26         {
27             //加入到epoll的监听队列里，这里用EPOLL_CTL_ADD
28             //来加一个新的 epoll事件。可以通过EPOLL_CTL_DEL来减少
29             //一个epoll事件，通过EPOLL_CTL_MOD来改变一个事件的监听方式。
30             fprintf(stderr, "epoll set insertion error: fd=%d", client);
31             return -1;
32         }
33     }
34     else
35         // 如果不是主socket的事件的话，则代表这是一个用户的socket的事件，
36         // 则用来处理这个用户的socket的事情是，比如说read(fd,xxx)之类，或者一些其他的处理。
37         do_use_fd(events[n].data.fd);
38 }
39 }
```

## 8 EPOLL模型的简单实例

```
1 #include <iostream>
2 #include <sys/socket.h>
3 #include <sys/epoll.h>
```





```
4 #include <netinet/in.h>
5 #include <arpa/inet.h>
6 #include <fcntl.h>
7 #include <unistd.h>
8 #include <stdio.h>
9
10 #define MAXLINE 10
11 #define OPEN_MAX 100
12 #define LISTENQ 20
13 #define SERV_PORT 5555
14 #define INFTIM 1000
15
16 void setnonblocking(int sock)
17 {
18     int opts;
19     opts = fcntl(sock, F_GETFL);
20     if(opts < 0)
21     {
22         perror("fcntl(sock, GETFL)");
23         exit(1);
24     }
25     opts = opts | O_NONBLOCK;
26     if(fcntl(sock, F_SETFL, opts) < 0)
27     {
28         perror("fcntl(sock,SETFL,opts)");
29         exit(1);
30     }
31 }
32
33 int main()
34 {
35     int i, maxi, listenfd, connfd, sockfd, epfd, nfd;
36     ssize_t n;
37     char line[MAXLINE];
38     socklen_t clilen;
39     //声明epoll_event结构体的变量，ev用于注册事件，events数组用于回传要处理的事件
40     struct epoll_event ev,events[20];
41     //生成用于处理accept的epoll专用的文件描述符，指定生成描述符的最大范围为256
42     epfd = epoll_create(256);
43     struct sockaddr_in clientaddr;
44     struct sockaddr_in serveraddr;
45     listenfd = socket(AF_INET, SOCK_STREAM, 0);
46
47     setnonblocking(listenfd); //把用于监听的socket设置为非阻塞方式
48     ev.data.fd = listenfd; //设置与要处理的事件相关的文件描述符
49     ev.events = EPOLLIN | EPOLLET; //设置要处理的事件类型
50     epoll_ctl(epfd, EPOLL_CTL_ADD, listenfd, &ev); //注册epoll事件
51     bzero(&serveraddr, sizeof(serveraddr));
52     serveraddr.sin_family = AF_INET;
53     char *local_addr = "200.200.200.204";
54     inet_aton(local_addr, &(serveraddr.sin_addr));
55     serveraddr.sin_port = htons(SERV_PORT); //或者htons(SERV_PORT);
56     bind(listenfd,(sockaddr *)&serveraddr, sizeof(serveraddr));
57     listen(listenfd, LISTENQ);
58
59     maxi = 0;
60     for( ; ; )
61     {
62         nfd = epoll_wait(epfd, events, 20, 500); //等待epoll事件的发生
63         for(i = 0; i < nfd; ++i) //处理所发生的所有事件
64         {
65             if(events[i].data.fd == listenfd) //监听事件
66             {
67                 connfd = accept(listenfd, (sockaddr *)&clientaddr, &clilen);
68                 if(connfd < 0)
69                 {
70                     perror("connfd<0");
71                     exit(1);
72                 }
73                 setnonblocking(connfd); //把客户端的socket设置为非阻塞方式
74                 char *str = inet_ntoa(clientaddr.sin_addr);
75                 std::cout << "connect from " << str <<std::endl;
76                 ev.data.fd=connfd; //设置用于读操作的文件描述符
77                 ev.events=EPOLLIN | EPOLLET; //设置用于注册的读操作事件
78                 epoll_ctl(epfd, EPOLL_CTL_ADD, connfd, &ev);
79                 //注册ev事件
80             }
81             else if(events[i].events&EPOLLIN) //读事件
82             {
83                 if ( (sockfd = events[i].data.fd) < 0)
84                 {
85                     continue;
86                 }
87                 if ( (n = read(sockfd, line, MAXLINE)) < 0) // 这里和IOCP不同
88                 {
89                     if (errno == ECONNRESET)
90                     {
91                         close(sockfd);
92                         events[i].data.fd = -1;
93                     }
94                     else
95                     {
96                         std::cout<<"readline error"<<std::endl;
97                     }
98                 }
99                 else if (n == 0)
100                 {
101                     close(sockfd);
102                     events[i].data.fd = -1;
103                 }
104                 ev.data.fd=sockfd; //设置用于写操作的文件描述符
105                 ev.events=EPOLLOUT | EPOLLET; //设置用于注册的写操作事件
106                 //修改sockfd上要处理的事件为EPOLLOUT
107                 epoll_ctl(epfd, EPOLL_CTL_MOD, sockfd, &ev);
108             }
109             else if(events[i].events&EPOLLOUT)//写事件
110             {
111                 sockfd = events[i].data.fd;
112                 write(sockfd, line, n);
113                 ev.data.fd = sockfd; //设置用于读操作的文件描述符
114                 ev.events = EPOLLIN | EPOLLET; //设置用于注册的读操作事件
115                 //修改sockfd上要处理的事件为EPOLIN
```



```
116     epoll_ctl(epfd, EPOLL_CTL_MOD, sockfd, &ev);
117 }
118 }
119 }
120 }
```

## 9.epoll进阶思考

### 9.1. 问题来源

最近学习EPOLL模型，介绍中说将EPOLL与Windows IOCP模型进行比较，说其的优势在于解决了IOCP模型大量线程上下文切换的开销，于是可以看出，EPOLL模型不需要多线程，即单线程中可以处理EPOLL逻辑。如果引入多线程反而会引起一些问题。但是EPOLL模型的服务器端到底可以不可以用多线程技术，如果可以，改怎么取舍，这成了困扰我的问题。上网查了一下，有这样几种声音：

- (1) “要么事件驱动(如epoll)，要么多线程，要么多进程，把这几个综合起来使用，感觉更加麻烦。” ；
- (2) “单线程使用epoll，但是不能发挥多核；多线程不用epoll。” ；
- (3) “主通信线程使用epoll所有需要监控的FD，有事件交给多线程去处理” ；
- (4) “既然用了epoll，那么线程就不应该看到fd，而只看到的是一个一个的业务请求/响应； epoll将网络数据组装成业务数据后，转交给业务线程进行处理。这就是常说的半同步半异步” 。

我比较赞同上述(3)、(4)中的观点

EPOLLOUT只有在缓冲区已经满了，不可以发送了，过了一会儿缓冲区中有空间了，就会触发EPOLLOUT，而且只触发一次。如果你编写的程序的网络IO不大，一次写入的数据不多的时候，通常都是epoll\_wait立刻就会触发 EPOLLOUT；如果你不调用 epoll，直接写 socket，那么情况就取决于这个socket的缓冲区是不是足够了。如果缓冲区足够，那么写就成功。如果缓冲区不足，那么取决你的socket是不是阻塞的，要么阻塞到写完成，要么出错返回。所以EPOLLOUT事件具有较大的随机性，ET模式一般只用于EPOLLIN, 很少用于EPOLLOUT。

### 9.2. 具体做法

- (1) 主通信线程使用epoll所有需要监控的FD，负责监控listenfd和connfd，这里只监听EPOLLIN事件，不监听EPOLLOUT事件；
- (2) 一旦从Client收到了数据以后，将其构造成一个消息，放入消息队列中；
- (3) 若干工作线程竞争，从消息队列中取出消息并进行处理，然后把处理结果发送给客户端。发送客户端的操作由工作线程完成。直接进行write。write到EAGAIN或EWOULDBLOCK后，线程循环continue等待缓冲区队列

发送函数代码如下：

```
1  bool send_data(int connfd, char *pbuffer, unsigned int &len,int flag)
2  {
3      if ((connfd < 0) || (0 == pbuffer))
4      {
5          return false;
6      }
7
8      int result = 0;
9      int remain_size = (int) len;
10     int send_size = 0;
11     const char *p = pbuffer;
12
13     time_t start_time = time(NULL);
14     int time_out = 3;
15
16     do
17     {
18         if (time(NULL) > start + time_out)
19         {
20             return false;
21         }
22
23         send_size = send(connfd, p, remain_size, flag);
24         if (nSentSize < 0)
25         {
26             if ((errno == EAGAIN) || (errno == EWOULDBLOCK) || (errno == EINTR))
27             {
28                 continue;
29             }
30             else
31             {
32                 len -= remain_size;
33                 return false;
34             }
35         }
36
37         p += send_size;
38         remain_size -= send_size;
39     }while(remain_size > 0);
40
41     return true;
42 }
```

## 10 epoll 实现服务器和客户端例子

最后我们用C++实现一个简单的客户端回射，所用到的代码文件是

```
1  net.h  server.cpp  client.cpp
```

服务器端：epoll实现的，干两件事分别为：1.等待客户端的连接，2.接收来自客户端的数据并且回射；

客户端：select实现，干两件事为:1.等待键盘输入，2.发送数据到服务器端并且接收服务器端回射的数据；

```
1  /*****
2  net.h
3  *****/
4  #include <stdio.h>
5
6  #ifndef _NET_H
7  #define _NET_H
8
9  #include <iostream>
10 #include <vector>
11 #include <algorithm>
12
13 #include <stdio.h>
14 #include <sys/types.h>
15 #include <sys/epoll.h> //epoll ways file
16 #include <sys/socket.h>
17 #include <fcntl.h>      //block and noblock
18
19 #include <stdlib.h>
```





```
99     ev.events = EPOLLIN | EPOLLET ;
100     if(epoll_ctl(epfd, EPOLL_CTL_ADD, cli_sock, &ev)< 0)
101         hand_error("epoll_ctl");
102     }
103
104     else if( events[num].events & EPOLLIN)
105     {
106         cli_sock = events[num].data.fd;
107         if(cli_sock < 0)
108             hand_error("cli_sock");
109         char  recvbuf[1024];
110         memset(recvbuf, 0 , sizeof(recvbuf));
111         int num = read( cli_sock, recvbuf, sizeof(recvbuf));
112         if(num == -1)
113             hand_error("read have some problem:");
114         if( num == 0 ) //stand of client have exit
115         {
116             cout<<"client have exit"<<endl;
117             close(cli_sock);
118             ev_remov = events[num];
119             epoll_ctl(epfd, EPOLL_CTL_DEL, cli_sock, &ev_remov);
120             clients.erase(remove(clients.begin(), clients.end(), cli_sock),clients.end());
121         }
122         fputs(recvbuf,stdout);
123         write(cli_sock, recvbuf, strlen(recvbuf));
124     }
125 }
126 }
127
128 return 0;
129 }
130
```

```
1  /*****
2  client.c
3  *****/
4
5  #include "net.h"
6
7  int main()
8  {
9      signal(SIGPIPE,SIG_IGN);
10     int sock;
11     sock = socket( AF_INET, SOCK_STREAM,0 ); //create a socket stream
12     if( sock< 0 )
13         hand_error( "socket_create");
14
15     struct sockaddr_in my_addr;
16
17     //memset my_addr;
18     memset(&my_addr, 0, sizeof(my_addr));
19     my_addr.sin_family = AF_INET;
20     my_addr.sin_port = htons(18000); //here is host sequeue
21     // my_addr.sin_addr.s_addr = htonl( INADDR_ANY );
22     my_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
23
24     int conn = connect(sock, (struct sockaddr *)&my_addr, sizeof(my_addr)) ;
25     if(conn != 0)
26         hand_error("connect");
27
28     char recvbuf[1024] = {0};
29     char sendbuf[1024] = {0};
30     fd_set rset;
31     FD_ZERO(&rset);
32
33     int nready = 0;
34     int maxfd;
35     int stdinof = fileno(stdin);
36     if( stdinof > sock)
37         maxfd = stdinof;
38     else
39         maxfd = sock;
40     while(1)
41     {
42         //select返回后把原来待检测的但是仍没就绪的描述字清0了。所以每次调用select前都要重新设置一下待检测的描述字
43         FD_SET(sock, &rset);
44         FD_SET(stdinof, &rset);
45         nready = select(maxfd+1, &rset, NULL, NULL, NULL);
46         cout<<"nready = "<<nready<<" "<<"maxfd = "<<maxfd<<endl;
47         if(nready == -1 )
48             break;
49         else if( nready == 0)
50             continue;
51         else
52         {
53             if( FD_ISSET(sock, &rset) ) //检测sock是否已经在集合rset里面。
54             {
55                 int ret = read( sock, recvbuf, sizeof(recvbuf)); //读数据
56                 if( ret == -1)
57                     hand_error("read");
58                 else if( ret == 0)
59                 {
60                     cout<<"sever have close"<<endl;
61                     close(sock);
62                     break;
63                 }
64                 else
65                 {
66                     fputs(recvbuf,stdout); //输出数据
67                     memset(recvbuf, 0, strlen(recvbuf));
68                 }
69             }
70
71             if( FD_ISSET(stdinof, &rset)) //检测stdin的文件描述符是否在集合里面
72             {
73                 if(fgets(sendbuf, sizeof(sendbuf), stdin) != NULL)
74                 {
75                     int num = write(sock, sendbuf, strlen(sendbuf)); //写数据
76                     cout<<"sent num = "<<num<<endl;
77                     memset(sendbuf, 0, sizeof(sendbuf));
78                 }
79             }
80         }
81     }
82 }
```





```
79         }  
80     }  
81 }  
82 return 0;  
83 }
```

顶 踩  
0 0

上一篇 [Linux c 源码 \( nMAsciiHexToBinary : 将16进制字符串格式转换为ASCII码形式 \)](#)  
下一篇 [Linux 内存泄露检测技巧](#)

相关文章推荐

- [epoll基本模型案例实现](#)
- [我读过最好的Epoll模型讲解](#)
- [Epoll模型详解](#)
- [朴素、Select、Poll和Epoll网络编程模型实现和分...](#)
- [epoll模型和使用详解（精髓）epoll - I/O event not...](#)

- [61 OrangePi Linux内核里的spi控制器驱动](#)
- [57 linux内核的i2c设备驱动模型](#)
- [56 linux内核里声明I2C设备的方法](#)
- [JMySQL操作命令语句实例](#)
- [高性能网络编程总结及《TCP/IP Sockets编程\(C语...](#)

### 猜你在找

- 【直播】机器学习&数据挖掘7周实训--韦玮

【直播】3小时掌握Docker最佳实战-徐西宁

【直播】计算机视觉原理及实战--屈教授

【直播】机器学习之矩阵--黄博士

【直播】机器学习之凸优化--马博士
- 【套餐】系统集成项目管理工程师顺利通关--徐朋

【套餐】机器学习系列套餐（算法+实战）--唐宇迪


【套餐】微信订阅号+服务号Java版 v2.0--翟东平

【套餐】微信订阅号+服务号Java版 v2.0--翟东平


【套餐】Javascript 设计模式实战--曾亮

查看评论

1楼 [bobkentblog](#) 2015-10-10 09:57发表

 后边的程序例子不错，赞一下博主

Re: [奔跑吧，行者](#) 2015-10-10 10:26发表

 回复**bobkentblog**: 恩 还好吧 我自己机器也编译测试通过了，达到了使用**epoll**的效果，继续努力。

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

\* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

[公司简介](#) | [招贤纳士](#) | [广告服务](#) | [联系方式](#) | [版权声明](#) | [法律顾问](#) | [问题报告](#) | [合作伙伴](#) | [论坛反馈](#)

[网站客服](#)   [杂志客服](#)   [微博客服](#)   [webmaster@csdn.net](#)   400-660-0108 | 北京创新乐知信息技术有限公司 版权所有 | [江苏知之为计算机有限公司](#) | [江苏乐知网络技术有限公司](#)

京 ICP 证 09002463 号 | Copyright © 1999-2017, CSDN.NET, All Rights Reserved 