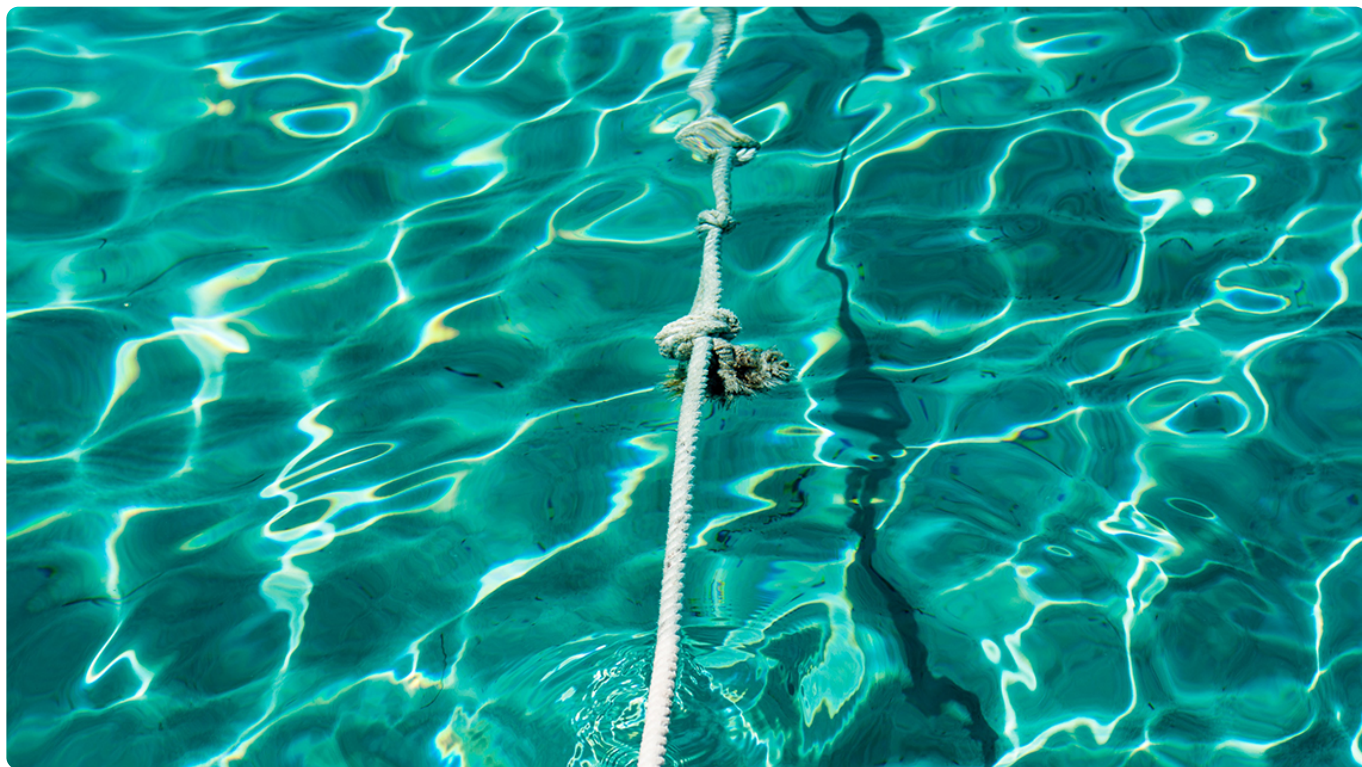


22 | 非阻塞I/O：提升性能的加速器

2019-09-27 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 10:26 大小 9.56M



你好，我是盛延敏，这里是网络编程实战第 22 讲，欢迎回来。

在性能篇的前两讲中，我分别介绍了 select 和 poll 两种不同的 I/O 多路复用技术。在接下来的这一讲中，我将带大家进入非阻塞 I/O 模式的世界。事实上，非阻塞 I/O 配合 I/O 多路复用，是高性能网络编程中的常见技术。

阻塞 VS 非阻塞

当应用程序调用阻塞 I/O 完成某个操作时，应用程序会被挂起，等待内核完成操作，感觉上应用程序像是被“阻塞”了一样。实际上，内核所做的事情是将 CPU 时间切换给其他有需要的进程，网络应用程序在这种情况下就会得不到 CPU 时间做该做的事情。

非阻塞 I/O 则不然，当应用程序调用非阻塞 I/O 完成某个操作时，内核立即返回，不会把 CPU 时间切换给其他进程，应用程序在返回后，可以得到足够的 CPU 时间继续完成其他事情。

如果拿去书店买书举例子，阻塞 I/O 对应什么场景呢？你去了书店，告诉老板（内核）你想要某本书，然后你就一直在那里等着，直到书店老板翻箱倒柜找到你想要的书，有可能还要帮你联系全城其它分店。注意，这个过程中你一直滞留在书店等待老板的回复，好像在书店老板这里"阻塞"住了。

那么非阻塞 I/O 呢？你去了书店，问老板有没有你心仪的那本书，老板查了下电脑，告诉你没有，你就悻悻离开了。一周以后，你又来这个书店，再问这个老板，老板一查，有了，于是你买了这本书。注意，这个过程中，你没有被阻塞，而是在不断轮询。

但轮询的效率太低了，于是你向老板提议：“老板，到货给我打电话吧，我再来付钱取书。”这就是前面讲到的 I/O 多路复用。

再进一步，你连去书店取书也想省了，得了，让老板代劳吧，你留下地址，付了书费，让老板到货时寄给你，你直接在家里拿到就可以看了。这就是我们将会在第 30 讲中讲到的异步 I/O。

这几个 I/O 模型，再加上进程、线程模型，构成了整个网络编程的知识核心。

按照使用场景，非阻塞 I/O 可以被用到读操作、写操作、接收连接操作和发起连接操作上。接下来，我们对它们一一解读。

非阻塞 I/O

读操作

如果套接字对应的接收缓冲区没有数据可读，在非阻塞情况下 read 调用会立即返回，一般返回 EWOULDBLOCK 或 EAGAIN 出错信息。在这种情况下，出错信息是需要小心处理，比如后面再次调用 read 操作，而不是直接作为错误直接返回。这就好像去书店买书没买到离开一样，需要不断进行又一次轮询处理。

写操作

不知道你有没有注意到，在阻塞 I/O 情况下，write 函数返回的字节数，和输入的参数总是一样的。如果返回值总是和输入的数据大小一样，write 等写入函数还需要定义返回值吗？我不知道你是不是和我一样，刚接触到这一部分知识的时候有这种困惑。

这里就要引出我们所说的非阻塞 I/O。在非阻塞 I/O 的情况下，如果套接字的发送缓冲区已达到了极限，不能容纳更多的字节，那么操作系统内核会**尽最大可能**从应用程序拷贝数据到发送缓冲区中，并立即从 write 等函数调用中返回。可想而知，在拷贝动作发生的瞬间，有可能一个字符也没拷贝，有可能所有请求字符都被拷贝完成，那么这个时候就需要返回一个数值，告诉应用程序到底有多少数据被成功拷贝到了发送缓冲区中，应用程序需要再次调用 write 函数，以输出未完成拷贝的字节。

write 等函数是可以同时作用到阻塞 I/O 和非阻塞 I/O 上的，为了复用一个函数，处理非阻塞和阻塞 I/O 多种情况，设计出了写入返回值，并用这个返回值表示实际写入的数据大小。


也就是说，非阻塞 I/O 和阻塞 I/O 处理的方式是不一样的。

非阻塞 I/O 需要这样：拷贝→返回→再拷贝→再返回。

而阻塞 I/O 需要这样：拷贝→直到所有数据拷贝至发送缓冲区完成→返回。

不过在实战中，你可以不用区别阻塞和非阻塞 I/O，使用循环的方式来写入数据就好了。只不过在阻塞 I/O 的情况下，循环只执行一次就结束了。

我在前面的章节中已经介绍了类似的方案，你可以在文稿里看到 writen 函数的实现。

 复制代码

```
1 /* 向文件描述符 fd 写入 n 字节数 */
2 ssize_t writen(int fd, const void * data, size_t n)
3 {
4     size_t    nleft;
5     ssize_t    nwritten;
6     const char *ptr;
7
8     ptr = data;
9     nleft = n;
10    // 如果还有数据没被拷贝完成，就一直循环
11    while (nleft > 0) {
12        if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
```

```

13         /* 这里 EINTR 是非阻塞 non-blocking 情况下，通知我们再次调用 write() */
14         if (nwritten < 0 && errno == EINTR)
15             nwritten = 0;
16         else
17             return -1;          /* 出错退出 */
18     }
19
20     /* 指针增大，剩下字节数变小 */
21     nleft -= nwritten;
22     ptr   += nwritten;
23 }
24 return n;
25 }

```

下面我通过一张表来总结一下 read 和 write 在阻塞模式和非阻塞模式下的不同行为特性：

操作	系统内核缓冲区状态	阻塞模式	非阻塞模式
read()	接收缓冲区有数据	立即返回	立即返回
	接收缓冲区没有数据	一直等待数据到来	立即返回，带有EWOULDBLOCK或EAGAIN错误
write()	发送缓冲区空闲	全部数据都写入发送缓冲区才返回	能写入多少就写入多少，立即返回
	发送缓冲区不空闲	等待发送缓冲区空闲	立即返回，带有EWOULDBLOCK或EAGAIN错误


关于 read 和 write 还有几个结论，你需要把握住：

1. read 总是在接收缓冲区有数据时就立即返回，不是等到应用程序给定的数据充满才返回。当接收缓冲区为空时，阻塞模式会等待，非阻塞模式立即返回 -1，并有 EWOULDBLOCK 或 EAGAIN 错误。
2. 和 read 不同，阻塞模式下，write 只有在发送缓冲区足以容纳应用程序的输出字节时才返回；而非阻塞模式下，则是能写入多少就写入多少，并返回实际写入的字节数。
3. 阻塞模式下的 write 有个特例，就是对方主动关闭了套接字，这个时候 write 调用会立即返回，并通过返回值告诉应用程序实际写入的字节数，如果再次对这样的套接字进行 write 操作，就会返回失败。失败是通过返回值 -1 来通知到应用程序的。

accept


当 `accept` 和 I/O 多路复用 `select`、`poll` 等一起配合使用时，如果在监听套接字上触发事件，说明有连接建立完成，此时调用 `accept` 肯定可以返回已连接套接字。这样看来，似乎把监听套接字设置为非阻塞，没有任何好处。

为了说明这个问题，我们构建一个客户端程序，其中最关键的是，一旦连接建立，设置 `SO_LINGER` 套接字选项，把 `l_onoff` 标志设置为 1，把 `l_linger` 时间设置为 0。这样，连接被关闭时，TCP 套接字上将会发送一个 RST。

 复制代码

```
1 struct linger ling;
2 ling.l_onoff = 1;
3 ling.l_linger = 0;
4 setsockopt(socket_fd, SOL_SOCKET, SO_LINGER, &ling, sizeof(ling));
5 close(socket_fd);
```

服务器端使用 `select` I/O 多路复用，不过，监听套接字仍然是 blocking 的。如果监听套接字上有事件发生，休眠 5 秒，以便模拟高并发场景下的情形。

 复制代码

```
1 if (FD_ISSET(listen_fd, &readset)) {
2     printf("listening socket readable\n");
3     sleep(5);
4     struct sockaddr_storage ss;
5     socklen_t slen = sizeof(ss);
6     int fd = accept(listen_fd, (struct sockaddr *) &ss, &slen);
```

这里的休眠时间非常关键，这样，在监听套接字上有可读事件发生时，并没有马上调用 `accept`。由于客户端发生了 RST 分节，该连接被接收端内核从自己的已完成队列中删除了，此时再调用 `accept`，由于没有已完成连接（假设没有其他已完成连接），`accept` 一直阻塞，更为严重的是，该线程再也没有机会对其他 I/O 事件进行分发，相当于该服务器无法对新连接和其他 I/O 进行服务。

如果我们将监听套接字设为非阻塞，上述的情形就不会再发生。只不过对于 `accept` 的返回值，需要正确地处理各种看似异常的错误，例如忽略 `EWOULDBLOCK`、`EAGAIN` 等。

这个例子给我们的启发是，一定要将监听套接字设置为非阻塞的，尽管这里休眠时间 5 秒有点夸张，但是在极端情况下处理不当的服务器程序是有可能碰到文稿中例子所阐述的情况，为了让服务器程序在极端情况下工作正常，这点工作还是非常值得的。

connect

在非阻塞 TCP 套接字上调用 connect 函数，会立即返回一个 EINPROGRESS 错误。TCP 三次握手会正常进行，应用程序可以继续做其他初始化的事情。当该连接建立成功或者失败时，通过 I/O 多路复用 select、poll 等可以进行连接的状态检测。

非阻塞 I/O + select 多路复用

文稿中给出了一个非阻塞 I/O 搭配 select 多路复用的例子。

 复制代码

```
1 #define MAX_LINE 1024
2 #define FD_INIT_SIZE 128
3
4 char rot13_char(char c) {
5     if ((c >= 'a' && c <= 'm') || (c >= 'A' && c <= 'M'))
6         return c + 13;
7     else if ((c >= 'n' && c <= 'z') || (c >= 'N' && c <= 'Z'))
8         return c - 13;
9     else
10         return c;
11 }
12
13 // 数据缓冲区
14 struct Buffer {
15     int connect_fd; // 连接字
16     char buffer[MAX_LINE]; // 实际缓冲
17     size_t writeIndex; // 缓冲写入位置
18     size_t readIndex; // 缓冲读取位置
19     int readable; // 是否可以读
20 };
21
22 struct Buffer *alloc_Buffer() {
23     struct Buffer *buffer = malloc(sizeof(struct Buffer));
24     if (!buffer)
25         return NULL;
26     buffer->connect_fd = 0;
27     buffer->writeIndex = buffer->readIndex = buffer->readable = 0;
28     return buffer;
29 }
30
31 void free_Buffer(struct Buffer *buffer) {
```

```

32     free(buffer);
33 }
34
35 int onSocketRead(int fd, struct Buffer *buffer) {
36     char buf[1024];
37     int i;
38     ssize_t result;
39     while (1) {
40         result = recv(fd, buf, sizeof(buf), 0);
41         if (result <= 0)
42             break;
43
44         for (i = 0; i < result; ++i) {
45             if (buffer->writeIndex < sizeof(buffer->buffer))
46                 buffer->buffer[buffer->writeIndex++] = rot13_char(buf[i]);
47             if (buf[i] == '\n') {
48                 buffer->readable = 1; // 缓冲区可以读
49             }
50         }
51     }
52
53     if (result == 0) {
54         return 1;
55     } else if (result < 0) {
56         if (errno == EAGAIN)
57             return 0;
58         return -1;
59     }
60
61     return 0;
62 }
63
64 int onSocketWrite(int fd, struct Buffer *buffer) {
65     while (buffer->readIndex < buffer->writeIndex) {
66         ssize_t result = send(fd, buffer->buffer + buffer->readIndex, buffer->writeIndex - buffer->readIndex, 0);
67         if (result < 0) {
68             if (errno == EAGAIN)
69                 return 0;
70             return -1;
71         }
72
73         buffer->readIndex += result;
74     }
75
76     if (buffer->readIndex == buffer->writeIndex)
77         buffer->readIndex = buffer->writeIndex = 0;
78
79     buffer->readable = 0;
80
81     return 0;
82 }
83

```

```

84 int main(int argc, char **argv) {
85     int listen_fd;
86     int i, maxfd;
87
88     struct Buffer *buffer[FD_INIT_SIZE];
89     for (i = 0; i < FD_INIT_SIZE; ++i) {
90         buffer[i] = alloc_Buffer();
91     }
92
93     listen_fd = tcp_nonblocking_server_listen(SERV_PORT);
94
95     fd_set readset, writeset, exset;
96     FD_ZERO(&readset);
97     FD_ZERO(&writeset);
98     FD_ZERO(&exset);
99
100    while (1) {
101        maxfd = listen_fd;
102
103        FD_ZERO(&readset);
104        FD_ZERO(&writeset);
105        FD_ZERO(&exset);
106
107        // listener 加入 readset
108        FD_SET(listen_fd, &readset);
109
110        for (i = 0; i < FD_INIT_SIZE; ++i) {
111            if (buffer[i]->connect_fd > 0) {
112                if (buffer[i]->connect_fd > maxfd)
113                    maxfd = buffer[i]->connect_fd;
114                FD_SET(buffer[i]->connect_fd, &readset);
115                if (buffer[i]->readable) {
116                    FD_SET(buffer[i]->connect_fd, &writeset);
117                }
118            }
119        }
120
121        if (select(maxfd + 1, &readset, &writeset, &exset, NULL) < 0) {
122            error(1, errno, "select error");
123        }
124
125        if (FD_ISSET(listen_fd, &readset)) {
126            printf("listening socket readable\n");
127            sleep(5);
128            struct sockaddr_storage ss;
129            socklen_t slen = sizeof(ss);
130            int fd = accept(listen_fd, (struct sockaddr *) &ss, &slen);
131            if (fd < 0) {
132                error(1, errno, "accept failed");
133            } else if (fd > FD_INIT_SIZE) {
134                error(1, 0, "too many connections");
135                close(fd);

```



```

136         } else {
137             make_nonblocking(fd);
138             if (buffer[fd]->connect_fd == 0) {
139                 buffer[fd]->connect_fd = fd;
140             } else {
141                 error(1, 0, "too many connections");
142             }
143         }
144     }
145
146     for (i = 0; i < maxfd + 1; ++i) {
147         int r = 0;
148         if (i == listen_fd)
149             continue;
150
151         if (FD_ISSET(i, &readset)) {
152             r = onSocketRead(i, buffer[i]);
153         }
154         if (r == 0 && FD_ISSET(i, &writeset)) {
155             r = onSocketWrite(i, buffer[i]);
156         }
157         if (r) {
158             buffer[i]->connect_fd = 0;
159             close(i);
160         }
161     }
162 }
163 }

```

第 93 行，调用 `fcntl` 将监听套接字设置为非阻塞。

 复制代码

```

1 fcntl(fd, F_SETFL, O_NONBLOCK);

```


第 121 行调用 `select` 进行 I/O 事件分发处理。

131-142 行在处理新的连接套接字，注意这里也把连接套接字设置为非阻塞的。

151-156 行在处理连接套接字上的 I/O 读写事件，这里我们抽象了一个 Buffer 对象，Buffer 对象使用了 `readIndex` 和 `writeIndex` 分别表示当前缓冲的读写位置。


实验

启动该服务器：

 复制代码

```
1 $./nonblockingserver
```

使用多个 telnet 客户端连接该服务器，可以验证交互正常。

 复制代码

```
1 $telnet 127.0.0.1 43211
2 Trying 127.0.0.1...
3 Connected to localhost.
4 Escape character is '^]'.
5 fasfasfasf
6 snfsnfsnfs
```


总结

非阻塞 I/O 可以使用在 read、write、accept、connect 等多种不同的场景，在非阻塞 I/O 下，使用轮询的方式引起 CPU 占用率高，所以一般将非阻塞 I/O 和 I/O 多路复用技术 select、poll 等搭配使用，在非阻塞 I/O 事件发生时，再调用对应事件的处理函数。这种方式，极大地提高了程序的健壮性和稳定性，是 Linux 下高性能网络编程的首选。

思考题

给大家布置两道思考题：

第一道，程序中第 133 行这个判断说明了什么？如果要改进的话，你有什么想法？

 复制代码

```
1 else if (fd > FD_INIT_SIZE) {
2     error(1, 0, "too many connections");
3     close(fd);
```

第二道，你可以仔细阅读一下数据读写部分 Buffer 的代码，你觉得用一个 Buffer 对象，而不是两个的目的是什么？

欢迎在评论区写下你的思考，我会和你一起交流，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。




网络编程实战

从底层到实战，深度解析网络编程

盛延敏

前大众点评云平台首席架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 21 | poll：另一种I/O多路复用

下一篇 23 | Linux利器：epoll的前世今生

精选留言 (10)

 写留言



MoonGod

2019-09-27

感觉这篇的解释和前面的比起来太不细致了...很多地方都没说明。老师能不能多一些说明啊

作者回复：我在代码里多加一些注释，可以看最新的代码

<https://github.com/froghui/yolanda>



2



石将从

2019-09-28

老师代码能不能写多点注释呀？基础差的同学看得费劲

展开

作者回复: 好的，我在github上的提交多加一些注释。

<https://github.com/froghui/yolanda>



1



沉淀的梦想

2019-09-30

老师代码中进行rot13_char编码的目的是啥？

展开



沉淀的梦想

2019-09-30

老师那个accept阻塞的实验，在我电脑（linux-4.18.0，ubuntu18.10）上的行为有点不太一样，无论是把listen_fd设置为阻塞还是非阻塞，sleep 5s还是10s或者更长，行为总是：accept成功获取到客户端连接，然后读取到客户端的RST

展开



沉淀的梦想

2019-09-29

文中说“对方主动关闭套接字，阻塞write调用会立即返回实际字节数，如果再次write，则返回失败”，这是的“关闭”只指两个方向关闭吗？如果对方只是半关闭的话，理论上本机还是可以继续write的吧



刘立伟

2019-09-29

ubuntu18.04环境下编译整个工程失败。提示如下
CMakeFiles/aio01.dir/aio01.c.o: In function `main':

aio01.c:(.text+0x19b): undefined reference to `aio_write'
aio01.c:(.text+0x1f4): undefined reference to `aio_error'
aio01.c:(.text+0x208): undefined reference to `aio_error'...

展开 ▾



yusuf

2019-09-29

1、133行判断是否超过了文件描述符的最大值，如果超过了，就会报错。可以考虑使用动态分配的方式，但如果超过了1024的话，使用上一节中的poll来处理会更好些
2、认为是考虑到对同一个fd的同一缓冲区进行读写操作，只用一个Buffer对象足够了

展开 ▾



石将从

2019-09-28

没有注释，看onSocketWrite和onSocketRead函数很费劲

作者回复: 其实还是蛮简单的，稍微解释一下：

onSocketRead是通过套接字读取数据，数据存放在Buffer对象里，Buffer对象通过了writeIndex记录当前数

据区可写的位置；

onSocketWrite通过套接字写数据，数据来源于Buffer缓冲对象，Buffer缓冲对象的readIndex记录了当前缓冲区读的位置。



刘丹

2019-09-27

感觉 readable 这个标记的处理有点小问题，没太考虑多个 \n 的情况

作者回复: 如果有多个\n，这里也认为客户端结束了，只不过\n会发送给客户端。

可以加一个处理，如果读到\n，就不要再继续读下去了。





程序水果宝
2019-09-27

应该把函数tcp_nonblocking_server_listen(SERV_PORT)的实现代码也给出来的，不然新手可能不知道怎么select 就变成了非阻塞了

展开 ▾

作者回复: 代码都在:<https://github.com/froghui/yolanda>

这里贴一段:

```
int listenfd;  
listenfd = socket(AF_INET, SOCK_STREAM, 0);  
fcntl(listenfd, F_SETFL, O_NONBLOCK);
```



💬 1

