

21 | poll: 另一种I/O多路复用

2019-09-25 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 11:14 大小 10.30M



你好，我是盛延敏，这是网络编程实战第 21 讲，欢迎回来。


上一讲我们讲到了 I/O 多路复用技术，并以 select 为核心，展示了 I/O 多路复用技术的能力。select 方法是多个 UNIX 平台支持的非常常见的 I/O 多路复用技术，它通过描述符集合来表示检测的 I/O 对象，通过三个不同的描述符集合来描述 I/O 事件：可读、可写和异常。但是 select 有一个缺点，那就是所支持的文件描述符的个数是有限的。在 Linux 系统中，select 的默认最大值为 1024。

那么有没有别的 I/O 多路复用技术可以突破文件描述符个数限制呢？当然有，这就是 poll 函数。这一讲，我们就来学习一下另一种 I/O 多路复用的技术：poll。

poll 函数介绍


poll 是除了 select 之外，另一种普遍使用的 I/O 多路复用技术，和 select 相比，它和内核交互的数据结构有所变化，另外，也突破了文件描述符的个数限制。

下面是 poll 函数的原型：

 复制代码


```
1 int poll(struct pollfd *fds, unsigned long nfds, int timeout);
2
3 返回值：若有就绪描述符则为其数目，若超时则为 0，若出错则为 -1
```

这个函数里面输入了三个参数，第一个参数是一个 pollfd 的数组。其中 pollfd 的结构如下：

 复制代码

```
1 struct pollfd {
2     int    fd;        /* file descriptor */
3     short  events;     /* events to look for */
4     short  revents;    /* events returned */
5 };
```

这个结构体由三个部分组成，首先是描述符 fd，然后是描述符上待检测的事件类型 events，注意这里的 events 可以表示多个不同的事件，具体的实现可以通过使用二进制掩码位操作来完成，例如，POLLIN 和 POLLOUT 可以表示读和写事件。


 复制代码

```
1 #define POLLIN    0x0001    /* any readable data available */
2 #define POLLPRI   0x0002    /* 00B/Urgent readable data */
3 #define POLLOUT   0x0004    /* file descriptor is writeable */
```

和 select 非常不同的地方在于，poll 每次检测之后的结果不会修改原来的传入值，而是将结果保留在 revents 字段中，这样就不需要每次检测完都得重置待检测的描述符和感兴趣的事件。我们可以把 revents 理解成 “returned events”。

events 类型的事件可以分为两大类。


第一类是可读事件，有以下几种：

 复制代码

```
1 #define POLLIN      0x0001    /* any readable data available */
2 #define POLLPRI     0x0002    /* OOB/Urgent readable data */
3 #define POLLRDNORM  0x0040    /* non-OOB/URG data available */
4 #define POLLRDBAND  0x0080    /* OOB/Urgent readable data */
```

一般我们在程序里面有 POLLIN 即可。套接字可读事件和 select 的 readset 基本一致，是系统内核通知应用程序有数据可以读，通过 read 函数执行操作不会被阻塞。


第二类是可写事件，有以下几种：

 复制代码

```
1 #define POLLOUT      0x0004    /* file descriptor is writeable */
2 #define POLLWRNORM  POLLOUT    /* no write type differentiation */
3 #define POLLWRBAND   0x0100    /* OOB/Urgent data can be written */
```

一般我们在程序里面统一使用 POLLOUT。套接字可写事件和 select 的 writeset 基本一致，是系统内核通知套接字缓冲区已准备好，通过 write 函数执行写操作不会被阻塞。

以上两大类的事件都可以在 “returned events” 得到复用。还有另一大类事件，没有办法通过 poll 向系统内核递交检测请求，只能通过 “returned events” 来加以检测，这类事件是各种错误事件。

 复制代码

```
1 #define POLLERR      0x0008    /* 一些错误发送 */
2 #define POLLHUP      0x0010    /* 描述符挂起 */
3 #define POLLNVAL     0x0020    /* 请求的事件无效 */
```

我们再回过头看一下 poll 函数的原型。参数 nfds 描述的是数组 fds 的大小，简单说，就是向 poll 申请的事件检测的个数。

最后一个参数 timeout，描述了 poll 的行为。

如果是一个 <0 的数，表示在有事件发生之前永远等待；如果是 0，表示不阻塞进程，立即返回；如果是一个 >0 的数，表示 poll 调用方等待指定的毫秒数后返回。

关于返回值，当有错误发生时，poll 函数的返回值为 -1；如果在指定的时间到达之前没有任何事件发生，则返回 0，否则就返回检测到的事件个数，也就是“returned events”中非 0 的描述符个数。

poll 函数有一点非常好，如果我们**不想对某个 pollfd 结构进行事件检测**，可以把它对应的 pollfd 结构的 fd 成员设置成一个负值。这样，poll 函数将忽略这样的 events 事件，检测完成以后，所对应的“returned events”的成员值也将设置为 0。

和 select 函数对比一下，我们发现 poll 函数和 select 不一样的地方就是，在 select 里面，文件描述符的个数已经随着 fd_set 的实现而固定，没有办法对此进行配置；而在 poll 函数里，我们可以控制 pollfd 结构的数组大小，这意味着我们可以突破原来 select 函数最大描述符的限制，在这种情况下，应用程序调用者需要分配 pollfd 数组并通知 poll 函数该数组的大小。

基于 poll 的服务器程序

下面我们将开发一个基于 poll 的服务器程序。这个程序可以同时处理多个客户端连接，并且一旦有客户端数据接收后，同步地回显回去。这已经是一个颇具高并发处理的服务器原型了，再加上后面讲到的非阻塞 I/O 和多线程等技术，基本上就是可使用的准生产级别了。

所以，让我们打起精神，一起来看这个程序。

 复制代码

```
1 #define INIT_SIZE 128
2
3 int main(int argc, char **argv) {
4     int listen_fd, connected_fd;
5     int ready_number;
6     ssize_t n;
7     char buf[MAXLINE];
```

```

8     struct sockaddr_in client_addr;
9
10    listen_fd = tcp_server_listen(SERV_PORT);
11
12    // 初始化 pollfd 数组，这个数组的第一个元素是 listen_fd，其余的用来记录将要连接的 connect
13    struct pollfd event_set[INIT_SIZE];
14    event_set[0].fd = listen_fd;
15    event_set[0].events = POLLRDNORM;
16
17    // 用 -1 表示这个数组位置还没有被占用
18    int i;
19    for (i = 1; i < INIT_SIZE; i++) {
20        event_set[i].fd = -1;
21    }
22
23    for (;;) {
24        if ((ready_number = poll(event_set, INIT_SIZE, -1)) < 0) {
25            error(1, errno, "poll failed ");
26        }
27
28        if (event_set[0].revents & POLLRDNORM) {
29            socklen_t client_len = sizeof(client_addr);
30            connected_fd = accept(listen_fd, (struct sockaddr *) &client_addr, &client_
31
32            // 找到一个可以记录该连接套接字的位置
33            for (i = 1; i < INIT_SIZE; i++) {
34                if (event_set[i].fd < 0) {
35                    event_set[i].fd = connected_fd;
36                    event_set[i].events = POLLRDNORM;
37                    break;
38                }
39            }
40
41            if (i == INIT_SIZE) {
42                error(1, errno, "can not hold so many clients");
43            }
44
45            if (--ready_number <= 0)
46                continue;
47        }
48
49        for (i = 1; i < INIT_SIZE; i++) {
50            int socket_fd;
51            if ((socket_fd = event_set[i].fd) < 0)
52                continue;
53            if (event_set[i].revents & (POLLRDNORM | POLLERR)) {
54                if ((n = read(socket_fd, buf, MAXLINE)) > 0) {
55                    if (write(socket_fd, buf, n) < 0) {
56                        error(1, errno, "write error");
57                    }
58                } else if (n == 0 || errno == ECONNRESET) {
59                    close(socket_fd);

```



```

60         event_set[i].fd = -1;
61     } else {
62         error(1, errno, "read error");
63     }
64
65     if (--ready_number <= 0)
66         break;
67 }
68 }
69 }
70 }

```

当然，一开始需要创建一个监听套接字，并绑定在本地的地址和端口上，这在第 10 行调用 `tcp_server_listen` 函数来完成。

在第 13 行，我初始化了一个 `pollfd` 数组，并命名为 `event_set`，之所以叫这个名字，是引用 `pollfd` 数组确实代表了检测的事件集合。这里数组的大小固定为 `INIT_SIZE`，这在实际的生产环境肯定是需要改进的。

我在前面讲过，监听套接字上如果有连接建立完成，也是可以通过 I/O 事件复用来检测到的。在第 14-15 行，将监听套接字 `listen_fd` 和对应的 `POLLRDNORM` 事件加入到 `event_set` 里，表示我们期望系统内核检测监听套接字上的连接建立完成事件。

在前面介绍 `poll` 函数时，我们提到过，如果对应 `pollfd` 里的文件描述字 `fd` 为负数，`poll` 函数将会忽略这个 `pollfd`，所以我们在第 18-21 行将 `event_set` 数组里其他没有用到的 `fd` 统统设置为 `-1`。这里 `-1` 也表示了当前 `pollfd` 没有被使用的意思。

下面我们的程序进入一个无限循环，在这个循环体内，第 24 行调用 `poll` 函数来进行事件检测。`poll` 函数传入的参数为 `event_set` 数组，数组大小 `INIT_SIZE` 和 `-1`。这里之所以传入 `INIT_SIZE`，是因为 `poll` 函数已经能保证可以自动忽略 `fd` 为 `-1` 的 `pollfd`，否则我们每次都需要计算一下 `event_size` 里真正需要被检测的元素大小；`timeout` 设置为 `-1`，表示在 I/O 事件发生之前 `poll` 调用一直阻塞。

如果系统内核检测到监听套接字上的连接建立事件，就进入到第 28 行的判断分支。我们看到，使用了如 `event_set[0].revent` 来和对应的事件类型进行位与操作，这个技巧大家一定要记住，这是因为 `event` 都是通过二进制位来进行记录的，位与操作是和对应的二进制位进行操作，一个文件描述字是可以对应到多个事件类型的。

在这个分支里，调用 `accept` 函数获取了连接描述字。接下来，33-38 行做了一件事，就是把连接描述字 `connect_fd` 也加入到 `event_set` 里，而且说明了我们感兴趣的事件类型为 `POLLRDNORM`，也就是套集字上有数据可以读。在这里，我们从数组里查找一个没有占用的位置，也就是 `fd` 为 `-1` 的位置，然后把 `fd` 设置为新的连接套接字 `connect_fd`。

如果在数组里找不到这样一个位置，说明我们的 `event_set` 已经被很多连接充满了，没有办法接收更多的连接了，这就是第 41-42 行所做的事情。

第 45-46 行是一个加速优化能力，因为 `poll` 返回的一个整数，说明了这次 I/O 事件描述符的个数，如果处理完监听套接字之后，就已经完成了这次 I/O 复用所要处理的事情，那么我们就可以跳过后面的处理，再次进入 `poll` 调用。

接下来的循环处理是查看 `event_set` 里面其他的事件，也就是已连接套接字的可读事件。这是通过遍历 `event_set` 数组来完成的。


如果数组里的 `pollfd` 的 `fd` 为 `-1`，说明这个 `pollfd` 没有递交有效的检测，直接跳过；来到第 53 行，通过检测 `revents` 的事件类型是 `POLLRDNORM` 或者 `POLLERR`，我们可以进行读操作。在第 54 行，读取数据正常之后，再通过 `write` 操作回显给客户端；在第 58 行，如果读到 EOF 或者是连接重置，则关闭这个连接，并且把 `event_set` 对应的 `pollfd` 重置；第 61 行读取数据失败。

和前面的优化加速处理一样，第 65-66 行是判断如果事件已经被完全处理完之后，直接跳过对 `event_set` 的循环处理，再次来到 `poll` 调用。

实验

我们启动这个服务器程序，然后通过 `telnet` 连接到这个服务器程序。为了检验这个服务器程序的 I/O 复用能力，我们可以多开几个 `telnet` 客户端，并且在屏幕上输入各种字符串。


客户端 1:

 复制代码

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

```
6 a
7 aaaaaaaaaa
8 aaaaaaaaaa
9 afafasfa
10 afafasfa
11 fb aa
12 fb aa
13 ^]
14
15
16 telnet> quit
17 Connection closed.
```

客户端 2:

 复制代码

```
1 telnet 127.0.0.1 43211
2 Trying 127.0.0.1...
3 Connected to 127.0.0.1.
4 Escape character is '^]'.
5 b
6 b
7 bbbbbbb
8 bbbbbbb
9 bbbbbbb
10 bbbbbbb
11 ^]
12
13
14 telnet> quit
15 Connection closed.
```

可以看到，这两个客户端互不影响，每个客户端输入的字符很快会被回显到客户端屏幕上。一个客户端断开连接，也不会影响到其他客户端。

总结

`poll` 是另一种在各种 UNIX 系统上被广泛支持的 I/O 多路复用技术，虽然名声没有 `select` 那么响，能力一点不比 `select` 差，而且因为可以突破 `select` 文件描述符的个数限制，在高并发的场景下尤其占优势。这一讲我们编写了一个基于 `poll` 的服务器程序，希望你从中学会 `poll` 的用法。

思考题

和往常一样，给大家留两道思考题：

第一道，在我们的程序里 `event_set` 数组的大小固定为 `INIT_SIZE`，这在实际的生产环境肯定是需要改进的。你知道如何改进吗？

第二道，如果我们进行了改进，那么接下来把连接描述符 `connect_fd` 也加入到 `event_set` 里，如何配合进行改造呢？

欢迎你在评论区写下你的思考，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 20 | 大名鼎鼎的select：看我如何同时感知多个I/O事件

下一篇 22 | 非阻塞I/O：提升性能的加速器

精选留言 (8)

写留言



d

2019-09-25

老师可否简单讲下底层实现，比如底层是数组，队列，红黑树等。

作者回复: 好问题，我收集一下素材。



👍 2



徐凯

2019-09-25

- 1.采用动态分配数组的方式
- 2.如果内存不够 进行realloc 或者申请一块更大的内存 然后把源数组拷贝过来

展开 ∨

作者回复: 鼓励动手来一个。



👍 2



Hale

2019-09-26

能讲讲为什么不用POLLIN来判断套接字可读？

展开 ∨

作者回复: POLLIN包括了OOB等带外数据的检测，POLLRDNORM则不包括这部分。

```
#define POLLIN    0x0001    /* any readable data available */
#define POLLRDNORM 0x0040    /* non-OOB/URG data available */
```



👍 1



LDxy

2019-09-25

为什么程序里使用POLLRDNORM而不是POLLIN呢？这两者又何不同？

作者回复:

```
#define POLLRDNORM 0x0040    /* non-OOB/URG data available */
#define POLLIN    0x0001    /* any readable data available */
```



👍 1

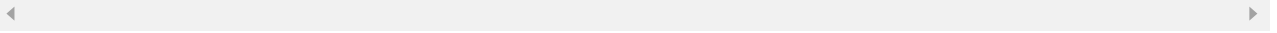


Jimmy Xiong

2019-09-28

请问老师，例子的全代码（可以直接运行起来）哪里可以找到？

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



CCC

2019-09-26

> lsof -i:43211

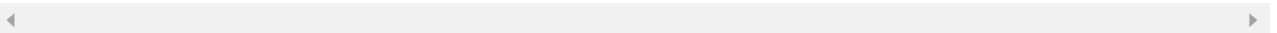
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME

pollserve 56364 jinhaoplus 3u IPv4 0xd6b66f6bf4c36f21 0t0 TCP *:43211 (LISTEN)

pollserve 56364 jinhaoplus 4u IPv4 0xd6b66f6bf3b80f21 0t0 TCP localhost:43211->localhost:63829 (ESTABLISHED)...

展开 ▾

作者回复: 63829, 63851, 63851就是三个不同的telnet客户端连接端口，由此断定肯定是三个不同的连接套接字。



初见

2019-09-25

老师您好~

我们既然有了poll，是不是代表着select 可以废弃了呢？

还是说他们各自仍然有不同的使用场景？

展开 ▾

作者回复: 场景不同，select仍然活跃在各种场合。



安排

2019-09-25

fd可读事件是不是有可能会误触发？也就是fd发生了可读事件，但是实际上并没有数据可读？所以我们要用非阻塞的读。这是内核的一个bug吗？还是。。。

展开 ▾

作者回复: 你是说使用阻塞I/O配合poll来使用?

