



下载APP



40 | 瞧一瞧Linux：详解socket的接口实现

2021-08-09 LMOS

《操作系统实战45讲》

课程介绍 >



讲述：陈晨

时长 09:54 大小 9.07M



你好，我是 LMOS。

上节课，我们一起了解了套接字的工作机制和数据结构，但套接字有哪些基本接口实现呢？相信学完这节课，你就能够解决这个问题了。

今天我会和你探讨套接字从创建、协议接口注册与初始化过程，还会为你深入分析套接字系统，是怎样调用各个功能函数的。通过这节课，相信你可以学会基于套接字来编写网络应用程序。有了之前的基础，想理解这节课并不难，让我们正式开始吧。



套接字接口

套接字接口最初是 BSD 操作系统的一部分，在应用层与 TCP/IP 协议栈之间提供了一套标准的独立于协议的接口。

Linux 内核实现的套接字接口，将 UNIX 的“一切都是文件操作”的概念应用在了网络连接访问上，让应用程序可以用常规文件操作 API 访问网络连接。

从 TCP/IP 协议栈的角度来看，传输层以上的都是应用程序的一部分，Linux 与传统的 UNIX 类似，TCP/IP 协议栈驻留在内核中，与内核的其他组件共享内存。传输层以上执行的网络功能，都是在用户地址空间完成的。

Linux 使用内核套接字概念与用户空间套接字通信，这样可以让实现和操作变得更简单。Linux 提供了一套 API 和套接字数据结构，这些服务向下与内核接口，向上与用户空间接口，应用程序正是使用这一套 API 访问内核中的网络功能。

套接字的创建

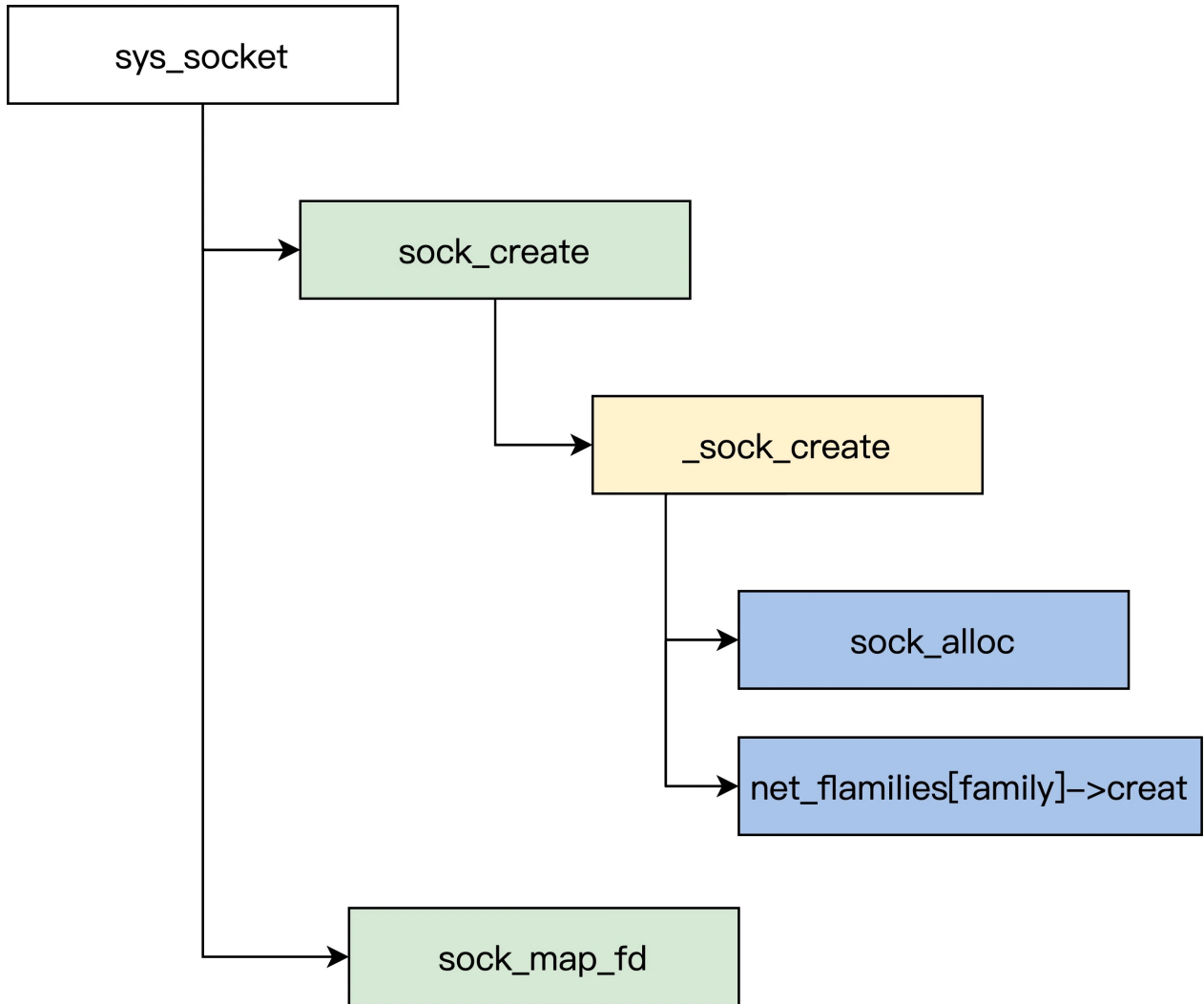
在应用程序使用 TCP/IP 协议栈的功能之前，我们必须调用套接字库函数 API 创建一个新的套接字，创建好以后，对库函数创建套接字的调用，就会转换为内核套接字创建函数的系统调用。

这时，完成的是通用套接字创建的初始化功能，跟具体的协议族并不相关。

这个过程具体是这样的，在应用程序中执行 socket 函数，socket 产生系统调用中断执行内核的套接字分路函数 sys_socketcall，在 sys_socketcall 套接字函数分路器中将调用传送到 sys_socket 函数，由 sys_socket 函数调用套接字的通用创建函数 sock_create。

sock_create 函数完成通用套接字创建、初始化任务后，再调用特定协议族的套接字创建函数。

这样描述你可能还没有直观感受，我特意画了图，帮你梳理 socket 创建的流程，你可以对照图片仔细体会调用过程。



socket创建示意图

结合图解，我再用一个具体例子帮你加深理解，比如由 AF_INET 协议族的 inet_create 函数完成套接字与特定协议族的关联。

一个新的 struct socket 数据结构起始由 sock_create 函数创建，**该函数直接调用 __sock_create 函数，__sock_create 函数的任务是为套接字预留需要的内存空间，由 sock_alloc 函数完成这项功能。**


这个 sock_alloc 函数不仅会为 struct socket 数据结构实例预留空间，也会为 struct inode 数据结构实例分配需要的内存空间，这样可以使两个数据结构的实例相关联。
__sock_create 函数代码如下。

[复制代码](#)

```
1 static int __sock_create(struct net *net, int family, int type, int protocol,  
2     struct socket **res, int kern)
```

```
3 {
4  int err;
5  struct socket *sock;
6  const struct net_proto_family *pf;
7  // 首先检验是否支持协议族
8  /*
9   * 检查是否在内核支持的socket范围内
10 */
11 if (family < 0 || family >= NPROTO)
12 return -EAFNOSUPPORT;
13 if (type < 0 || type >= SOCK_MAX)
14 return -EINVAL;
15 /*
16 * 为新的套接字分配内存空间, 分配成功后返回新的指针
17 */
18
19 sock = sock_alloc();
20 }
```

sock_alloc 函数如下所示。

 复制代码

```
1 static struct socket *sock_alloc(void) {
2  struct inode *inode;
3  struct socket *sock;
4  // 初始化一个可用的inode节点, 在fs/inode.c中
5  inode = new_inode(sock_mnt->mnt_sb);
6  if (!inode)
7  return NULL;
8  // 实际创建的是socket_alloc复合对象, 因此要使用SOCKET_I宏从inode中取出关联的socket
9  sock = SOCKET_I(inode);
10
11  kmemcheck_annotate_bitfield(sock, type);
12  // 文件类型为套接字
13  inode->i_mode = S_IFSOCK | S_IRWXUGO;
14  inode->i_uid = current_fsuid();
15  inode->i_gid = current_fsgid();
16
17  percpu_add(sockets_in_use, 1);
18 return sock;
19 }
```

当具体的协议与新套接字相连时, 其内部状态的管理由协议自身维护。

现在, 函数将 struct socket 数据结构的 struct proto_ops *ops 设置为 NULL。随后, 当某个协议族中的协议成员的套接字创建函数被调用时, ops 将指向协议实例的操作函数。


这时将 struct socket 数据结构的 flags 数据域设置为 0，创建时还没有任何标志需要设置。

在之后的调用中，应用程序调用 send 或 receive 套接字库函数时会设置 flags 数据域。最后将其他两个数据域 sk 和 file 初始化为 NULL。sk 数据域随后会把由协议特有的套接字创建函数设置为指向内部套接字结构。file 将在调用 sock_mapfd 函数时设置为分配的文件返回的指针。

文件指针用于访问打开套接字的虚拟文件系统的文件状态。在 sock_alloc 函数返回后，sock_create 函数调用协议族的套接字创建函数 err = pf->create(net, sock, protocol)，它通过访问 net_families 数组获取协议族的创建函数，对于 TCP/IP 协议栈，协议族将设置为 AF_INET。

套接字的绑定

创建完套接字后，应用程序需要调用 sys_bind 函数把套接字和地址绑定起来，代码如下所示。

 复制代码

```
1  asmlinkage long sysbind (bind, int, fd, struct sockaddr __user *, umyaddr, int
2  {
3      struct socket *sock;
4      struct sockaddr_storage address;
5      int err, fput_needed;
6
7      /*
8       * 获取socket实例。
9       */
10     sock = sockfd_lookup_light(fd, &err, &fput_needed);
11     if (sock) {
12         err = move_addr_to_kernel(umyaddr, addrlen, (struct sockaddr *)&address);
13         if (err >= 0) {
14             err = security_socket_bind(sock,
15                                     (struct sockaddr *)&address,
16                                     addrlen);
17             /*
18              * 如果是TCP套接字，sock->ops指向的是inet_stream_ops，
19              * sock->ops是在inet_create()函数中初始化，所以bind接口
20              * 调用的是inet_bind()函数。
21              */
22             if (!err)
23                 err = sock->ops->bind(sock,
24                                     (struct sockaddr *)
```

```

25         &address, addrlen);
26     }
27     fput_light(sock->file, fput_needed);
28 }
29 return err;
30 }

```

结合代码，我们可以看到，`sys_bind` 函数首先会查找套接字对应的 `socket` 实例，调用 **`sockfd_lookup_light`**。在绑定之前，将用户空间的地址拷贝到内核空间的缓冲区中，在拷贝过程中会检查用户传入的地址是否正确。

等上述的准备工作完成后，就会调用 **`inet_bind` 函数**来完成绑定操作。**`inet_bind` 函数**代码如下所示。

[复制代码](#)

```

1 int inet_bind(struct socket *sock, struct sockaddr *uaddr, int addr_len)
2 {
3     struct sockaddr_in *addr = (struct sockaddr_in *)uaddr;
4     struct sock *sk = sock->sk;
5     struct inet_sock *inet = inet_sk(sk);
6     unsigned short snum;
7     int chk_addr_ret;
8     int err;
9
10    if (sk->sk_prot->bind) { /* 如果传输层接口上实现了bind调用，则回调它。目前只有SOCK_
11        err = sk->sk_prot->bind(sk, uaddr, addr_len);
12        goto out;
13    }
14    err = -EINVAL;
15    if (addr_len < sizeof(struct sockaddr_in))
16        goto out;
17    err = -EADDRNOTAVAIL;
18    if (!sysctl_ip_nonlocal_bind && /* 必须绑定到本地接口的地址 */
19        !inet->freebind &&
20        addr->sin_addr.s_addr != INADDR_ANY && /* 绑定地址不合法 */
21        chk_addr_ret != RTN_LOCAL &&
22        chk_addr_ret != RTN_MULTICAST &&
23        chk_addr_ret != RTN_BROADCAST)
24        goto out;
25
26    snum = ntohs(addr->sin_port);
27    err = -EACCES;
28    if (snum && snum < PROT_SOCK && !capable(CAP_NET_BIND_SERVICE))
29        goto out;
30
31    lock_sock(sk); /* 对套接口进行加锁，因为后面要对其状态进行判断 */

```



```


32  /* Check these errors (active socket, double bind). */
33  err = -EINVAL;
34  /**
35   * 如果状态不为CLOSE，表示套接口已经处于活动状态，不能再绑定
36   * 或者已经指定了本地端口号，也不能再绑定
37   */
38  if (sk->sk_state != TCP_CLOSE || inet->num)
39      goto out_release_sock;
40
41  /* 设置地址到传输控制块中 */
42  inet->rcv_saddr = inet->saddr = addr->sin_addr.s_addr;
43  /* 如果是广播或者多播地址，则源地址使用设备地址。 */
44  if (chk_addr_ret == RTN_MULTICAST || chk_addr_ret == RTN_BROADCAST)
45      inet->saddr = 0; /* Use device */
46
47  /* 调用传输层的get_port来进行地址绑定。如tcp_v4_get_port或udp_v4_get_port */
48  if (sk->sk_prot->get_port(sk, snum)) {
49      ...
50  }
51
52  /* 设置标志，表示已经绑定了本地地址和端口 */
53  if (inet->rcv_saddr)
54      sk->sk_userlocks |= SOCK_BINDADDR_LOCK;
55  if (snum)
56      sk->sk_userlocks |= SOCK_BINDPORT_LOCK;
57  inet->sport = htons(inet->num);
58  /* 还没有连接到对方，清除远端地址和端口 */
59  inet->daddr = 0;
60  inet->dport = 0;
61  /* 清除路由缓存 */
62  sk_dst_reset(sk);
63  err = 0;
64  out_release_sock:
65      release_sock(sk);
66  out:
67      return err;
68  }
69

```

主动连接

因为应用程序处理的是面向连接的网络服务（SOCK_STREAM 或 SOCK_SEQPACKET），所以在交换数据之前，需要在请求连接服务的进程（客户）与提供服务的进程（服务器）之间建立连接。

当应用程序调用 **connect** 函数发出连接请求时，内核会启动函数 **sys_connect**，详细代码如下。

 复制代码

```
1 int __sys_connect(int fd, struct sockaddr __user *useraddr, int addrlen)
2 {
3     int ret = -EBADF;
4     struct fd f;
5     f = fdget(fd);
6     if (f.file) {
7         struct sockaddr_storage address;
8         ret = move_addr_to_kernel(useraddr, addrlen, &address);
9         if (!ret)
10             // 调用__sys_connect_file
11             ret = __sys_connect_file(f.file, &address, addrlen, 0);
12         fdput(f);
13     }
14     return ret;
15 }
```

连接成功会返回 socket 的描述符，否则会返回一个错误码。

监听套接字

调用 listen 函数时，应用程序触发内核的 **sys_listen** 函数，把套接字描述符 fd 对应的套接字设置为监听模式，观察连接请求。详细代码你可以看看后面的内容。

 复制代码

```
1 int __sys_listen(int fd, int backlog)
2 {
3     struct socket *sock;
4     int err, fput_needed;
5     int somaxconn;
6     // 通过套接字描述符找到struct socket
7     sock = sockfd_lookup_light(fd, &err, &fput_needed);
8     if (sock) {
9         somaxconn = sock_net(sock->sk)->core.sysctl_somaxconn;
10        if ((unsigned int)backlog > somaxconn)
11            backlog = somaxconn;
12        err = security_socket_listen(sock, backlog);
13        if (!err)
14            // 根据套接字类型调用监听函数
15            err = sock->ops->listen(sock, backlog);
16        fput_light(sock->file, fput_needed);
17    }
18    return err;
19 }
```


被动接收连接

前面说过主动连接，我们再来看看被动接受连接的情况。接受一个客户端的连接请求会调用 **accept** 函数，应用程序触发内核函数 **sys_accept**，等待接收连接请求。如果允许连接，则重新创建一个代表该连接的套接字，并返回其套接字描述符，代码如下。

[复制代码](#)

```
1 int __sys_accept4_file(struct file *file, unsigned file_flags,
2                       struct sockaddr __user *upeer_sockaddr,
3                       int __user *upeer_addrln, int flags,
4                       unsigned long nofile)
5 {
6     struct socket *sock, *newsock;
7     struct file *newfile;
8     int err, len, newfd;
9     struct sockaddr_storage address;
10    if (flags & ~(SOCK_CLOEXEC | SOCK_NONBLOCK))
11        return -EINVAL;
12    if (SOCK_NONBLOCK != O_NONBLOCK && (flags & SOCK_NONBLOCK))
13        flags = (flags & ~SOCK_NONBLOCK) | O_NONBLOCK;
14    sock = sock_from_file(file, &err);
15    if (!sock)
16        goto out;
17    err = -ENFILE;
18    // 创建一个新套接字
19    newsock = sock_alloc();
20    if (!newsock)
21        goto out;
22    newsock->type = sock->type;
23    newsock->ops = sock->ops;
24    __module_get(newsock->ops->owner);
25    newfd = __get_unused_fd_flags(flags, nofile);
26    if (unlikely(newfd < 0)) {
27        err = newfd;
28        sock_release(newsock);
29        goto out;
30    }
31    newfile = sock_alloc_file(newsock, flags, sock->sk->sk_prot_creator->name);
32    if (IS_ERR(newfile)) {
33        err = PTR_ERR(newfile);
34        put_unused_fd(newfd);
35        goto out;
36    }
37    err = security_socket_accept(sock, newsock);
38    if (err)
39        goto out_fd;
40    // 根据套接字类型调用不同的函数inet_accept
41    err = sock->ops->accept(sock, newsock, sock->file->f_flags | file_flags,
42                          false);
```

```
43     if (err < 0)
44         goto out_fd;
45     if (upeer_sockaddr) {
46         len = newsock->ops->getname(newsock,
47             (struct sockaddr *)&address, 2);
48         if (len < 0) {
49             err = -ECONNABORTED;
50             goto out_fd;
51         }
52         // 从内核复制到用户空间
53         err = move_addr_to_user(&address,
54             len, upeer_sockaddr, upeer_addrlen);
55         if (err < 0)
56             goto out_fd;
57     }
58     /* File flags are not inherited via accept() unlike another OSes. */
59     fd_install(newfd, newfile);
60     err = newfd;
61 out:
62     return err;
63 out_fd:
64     fput(newfile);
65     put_unused_fd(newfd);
66     goto out;
67 }
```

这个新的套接字描述符与最初创建套接字时，设置的套接字地址族与套接字类型、使用的协议一样。原来创建的套接字不与连接关联，它继续在原套接字上侦听，以便接收其他连接请求。

发送数据

套接字应用中最简单的传送函数是 **send**，send 函数的作用类似于 write，但 send 函数允许应用程序指定标志，规定如何对待传送数据。调用 send 函数时，会触发内核的 **sys_send** 函数，把发送缓冲区的数据发送出去。

sys_send 函数具体调用流程如下。

1. 应用程序的数据被复制到内核后，sys_send 函数调用 **sock_sendmsg**，依据协议族类型来执行发送操作。
2. 如果是 INET 协议族套接字，sock_sendmsg 将调用 inet_sendmsg 函数。

3. 如果采用 TCP 协议，inet_sendmsg 函数将调用 tcp_sendmsg，并按照 TCP 协议规则来发送数据包。


send 函数返回发送成功，并不意味着在连接的另一端的进程可以收到数据，这里只能保证发送 send 函数执行成功，发送给网络设备驱动程序的数据没有出错。

接收数据

recv 函数与文件读 read 函数类似，recv 函数中可以指定标志来控制如何接收数据，调用 recv 函数时，应用程序会触发内核的 sys_recv 函数，把网络中的数据递交到应用程序。当然，read、recvfrom 函数也会触发 sys_recv 函数。具体流程如下。

1. 为把内核的网络数据转入应用程序的接收缓冲区，sys_recv 函数依次调用 **sys_recvfrom**、**sock_recvfrom** 和 **__sock_recvmsg**，并依据协议族类型来执行具体的接收操作。
2. 如果是 INET 协议族套接字，__sock_recvmsg 将调用 sock_common_recvmsg 函数。
3. 如果采用 TCP 协议，sock_common_recvmsg 函数将调用 tcp_recvmsg，按照 TCP 协议规则来接收数据包

如果接收方想获取数据包发送端的标识符，应用程序可以调用 **sys_recvfrom** 函数来获取数据包发送方的源地址，下面是 **sys_recvfrom** 函数的实现。

 复制代码

```
1 int __sys_recvfrom(int fd, void __user *ubuf, size_t size, unsigned int flags,
2     struct sockaddr __user *addr, int __user *addr_len)
3 {
4     struct socket *sock;
5     struct iovec iov;
6     struct msghdr msg;
7     struct sockaddr_storage address;
8     int err, err2;
9     int fput_needed;
10    err = import_single_range(READ, ubuf, size, &iov, &msg.msg_iter);
11    if (unlikely(err))
12        return err;
13    // 通过套接字描述符找到struct socket
14    sock = sockfd_lookup_light(fd, &err, &fput_needed);
15    if (!sock)
```

```
16     goto out;
17     msg.msg_control = NULL;
18     msg.msg_controllen = 0;
19     /* Save some cycles and don't copy the address if not needed */
20     msg.msg_name = addr ? (struct sockaddr *)&address : NULL;
21     /* We assume all kernel code knows the size of sockaddr_storage */
22     msg.msg_namelen = 0;
23     msg.msg_iocb = NULL;
24     msg.msg_flags = 0;
25     if (sock->file->f_flags & O_NONBLOCK)
26         flags |= MSG_DONTWAIT;
27     // sock_recvmmsg为具体的接收函数
28     err = sock_recvmmsg(sock, &msg, flags);
29     if (err >= 0 && addr != NULL) {
30         // 从内核复制到用户空间
31         err2 = move_addr_to_user(&address,
32                                 msg.msg_namelen, addr, addr_len);
33         if (err2 < 0)
34             err = err2;
35     }
36     fput_light(sock->file, fput_needed);
37 out:
38     return err;
39 }
```

关闭连接

最后，我们来看看如何关闭连接。当应用程序调用 shutdown 函数关闭连接时，内核会启动函数 sys_shutdown，代码如下。

[复制代码](#)

```
1 int __sys_shutdown(int fd, int how)
2 {
3     int err, fput_needed;
4     struct socket *sock;
5     sock = sockfd_lookup_light(fd, &err, &fput_needed); /* 通过套接字，描述符找到对应
6     if (sock != NULL) {
7         err = security_socket_shutdown(sock, how);
8         if (!err)
9             /* 根据套接字协议族调用关闭函数*/
10             err = sock->ops->shutdown(sock, how);
11             fput_light(sock->file, fput_needed);
12     }
13     return err;
14 }
```

重点回顾

好，这节课的内容告一段落了，我来给你做个总结。这节课我们继续研究了套接字在 Linux 内核中的实现。

套接字是 UNIX 兼容系统的一大特色，Linux 在此基础上实现了内核套接字与应用程序套接字接口，在用户地址空间与内核地址空间之间提供了一套标准接口，实现应用套接字库函数与内核功能之间的——对应，简化了用户地址空间与内核地址空间交换数据的过程。

通过应用套接字 API 编写网络应用程序，我们可以利用 Linux 内核 TCP/IP 协议栈提供的网络通信服务，在网络上实现应用数据快速、有效的传送。除此之外，套接字编程还可以使我们获取网络、主机的各种管理、统计信息。

创建套接字应用程序一般要经过后面这 6 个步骤。

1. 创建套接字。
2. 将套接字与地址绑定，设置套接字选项。
3. 建立套接字之间的连接。
4. 监听套接字
5. 接收、发送数据。
6. 关闭、释放套接字。

思考题

我们了解的 TCP 三次握手，发生在 socket 的哪几个函数中呢？

欢迎你在留言区跟我交流，也推荐你把这节课转发给有需要的朋友。

我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 **20元** 现金奖励

👍 赞 2

💡 提建议

© 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 39 | 瞧一瞧Linux：详解socket实现与网络编程接口

下一篇 41 | 服务接口：如何搭建沟通桥梁？

小争哥新书

数据结构与算法之美

图书+专栏，双管齐下，拿下算法

打包价 **¥159** 原价¥319

仅限 300 套



精选留言 (3)

💬 写留言



MacBao

2021-08-09

服务器端处于listen状态，客户端connect发起TCP三次握手？

💬

👍 1



pedro

2021-08-09

今天的问题不好回答，因为文中无明显三次握手的代码，而且三次握手的机制其实比较复杂，涉及到几个状态和几个队列之间的切换，笼统的 connect 和 accept 函数是说不清楚的，感兴趣可以看看这里：

<https://blog.csdn.net/tennysonsky/article/details/45621341>

...

展开 ▾

作者回复: 好的 期待



💬 1

👍 1

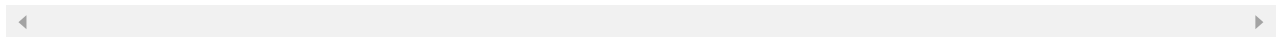


pedro 

2021-08-09

这里有一篇三次握手的源码图解：<https://mp.weixin.qq.com/s/vlrzGc5bFrPlr9a7Hlr2eA>

作者回复: 谢谢 老铁



💬

👍