

44 | Socket内核数据结构：如何成立特大项目合作部？

2019-07-08 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超


时长 20:23 大小 18.67M



上一节我们讲了 Socket 在 TCP 和 UDP 场景下的调用流程。这一节，我们就沿着这个流程到内核里面一探究竟，看看在内核里面，都创建了哪些数据结构，做了哪些事情。

解析 socket 函数

我们从 Socket 系统调用开始。


 复制代码

```
1 SYSCALL_DEFINE3(socket, int, family, int, type, int, pr
2 {
3     int retval;
4     struct socket *sock;
5     int flags;
6     .....
7     if (SOCK_NONBLOCK != O_NONBLOCK && (flags & SOC
8         flags = (flags & ~SOCK_NONBLOCK) | O_NC
9
10    retval = sock_create(family, type, protocol, &s
11    .....
12    retval = sock_map_fd(sock, flags & (O_CLOEXEC |
13    .....
14    return retval;
15 }
```

这里面的代码比较容易看懂，Socket 系统调用会调用 sock_create 创建一个 struct socket 结构，然后通过 sock_map_fd 和文件描述符对应起来。

在创建 Socket 的时候，有三个参数。

一个是**family**，表示地址族。不是所有的 Socket 都要通过 IP 进行通信，还有其他的通信方式。例如，下面的定义中，domain sockets 就是通过本地文件进行通信的，不需要 IP 地址。只不过，通过 IP 地址是最常用的模式，所以我们这里着重分析这种模式。


 复制代码

```
1 #define AF_UNIX 1/* Unix domain sockets */
2 #define AF_INET 2/* Internet IP Protocol */
```

第二个参数是**type**，也即 Socket 的类型。类型是比较少的。

第三个参数是**protocol**，是协议。协议数目是比较多的，也就是说，多个协议会属于同一种类型。

常用的 Socket 类型有三种，分别是 SOCK_STREAM、SOCK_DGRAM 和 SOCK_RAW。

 复制代码

```
1 enum sock_type {
2     SOCK_STREAM = 1,
3     SOCK_DGRAM = 2,
4     SOCK_RAW = 3,
```


```
5 .....
6 }
```

SOCK_STREAM 是面向数据流的，协议 IPPROTO_TCP 属于这种类型。SOCK_DGRAM 是面向数据报的，协议 IPPROTO_UDP 属于这种类型。如果在内核里面看的话，IPPROTO_ICMP 也属于这种类型。SOCK_RAW 是原始的 IP 包，IPPROTO_IP 属于这种类型。

这一节，我们重点看 SOCK_STREAM 类型和 IPPROTO_TCP 协议。

为了管理 family、type、protocol 这三个分类层次，内核会创建对应的数据结构。

接下来，我们打开 sock_create 函数看一下。它会调用 __sock_create。

 复制代码

```
1 int __sock_create(struct net *net, int family, int type
2                   struct socket **res, int kern)
3 {
4     int err;
5     struct socket *sock;
6     const struct net_proto_family *pf;
```


```

7 .....
8         sock = sock_alloc();
9 .....
10        sock->type = type;
11 .....
12        pf = rcu_dereference(net_families[family]);
13 .....
14        err = pf->create(net, sock, protocol, kern);
15 .....
16        *res = sock;
17
18        return 0;
19 }

```



这里先是分配了一个 struct socket 结构。接下来我们要用到 family 参数。这里有一个 net_families 数组，我们可以以 family 参数为下标，找到对应的 struct net_proto_family。

 复制代码

```

1 /* Supported address families. */
2 #define AF_UNSPEC        0
3 #define AF_UNIX          1        /* Unix domain sockets
4 #define AF_LOCAL         1        /* POSIX name for AF_UNIX
5 #define AF_INET          2        /* Internet IP Protocol
6 .....
7 #define AF_INET6         10       /* IP version 6
8 .....
9 #define AF_MPLS          28       /* MPLS */
10 .....


```

```

11 #define AF_MAX          44      /* For now.. */
12 #define NPROTO          AF_MAX
13
14 struct net_proto_family __rcu *net_families[NPROTO] __r

```

我们可以找到 `net_families` 的定义。每一个地址族在这个数组里面都有一项，里面的内容是 `net_proto_family`。每一种地址族都有自己的 `net_proto_family`，IP 地址族的 `net_proto_family` 定义如下，里面最重要的就是，`create` 函数指向 `inet_create`。


 复制代码

```

1 //net/ipv4/af_inet.c
2 static const struct net_proto_family inet_family_ops =
3     .family = PF_INET,
4     .create = inet_create, // 这个用于 socket 系统调用
5     .....
6 }

```

我们回到函数 `_sock_create`。接下来，在这里面，这个 `inet_create` 会被调用。

 复制代码

```

1 static int inet_create(struct net *net, struct socket *

```

```

2  {
3      struct sock *sk;
4      struct inet_protosw *answer;
5      struct inet_sock *inet;
6      struct proto *answer_prot;
7      unsigned char answer_flags;
8      int try_loading_module = 0;
9      int err;
10
11     /* Look for the requested type/protocol pair. */
12 lookup_protocol:
13     list_for_each_entry_rcu(answer, &inetsw[sock->type],
14                             struct inet_protosw, protosw) {
15         err = 0;
16         /* Check the non-wild match. */
17         if (protocol == answer->protocol) {
18             if (protocol != IPPROTO_IP)
19                 break;
20         } else {
21             /* Check for the two wild cases
22             if (IPPROTO_IP == protocol) {
23                 protocol = answer->protocol;
24                 break;
25             }
26             if (IPPROTO_IP == answer->protocol)
27                 break;
28         }
29         err = -EPROTONOSUPPORT;
30     }
31     .....
32     sock->ops = answer->ops;
33     answer_prot = answer->prot;
34     answer_flags = answer->flags;
35     .....
36     sk = sk_alloc(net, PF_INET, GFP_KERNEL, answer_
37     .....

```


```

37     inet = inet_sk(sk);
38     inet->nodelfrag = 0;
39     if (SOCK_RAW == sock->type) {
40         inet->inet_num = protocol;
41         if (IPPROTO_RAW == protocol)
42             inet->hdrincl = 1;
43     }
44     inet->inet_id = 0;
45     sock_init_data(sock, sk);
46
47     sk->sk_destruct    = inet_sock_destruct;
48     sk->sk_protocol    = protocol;
49     sk->sk_backlog_rcv = sk->sk_prot->backlog_rcv;
50
51     inet->uc_ttl       = -1;
52     inet->mc_loop       = 1;
53     inet->mc_ttl        = 1;
54     inet->mc_all        = 1;
55     inet->mc_index      = 0;
56     inet->mc_list       = NULL;
57     inet->rcv_tos       = 0;
58
59     if (inet->inet_num) {
60         inet->inet_sport = htons(inet->inet_num
61             /* Add to protocol hash chains. */
62             err = sk->sk_prot->hash(sk);
63     }
64
65     if (sk->sk_prot->init) {
66         err = sk->sk_prot->init(sk);
67     }
68     .....
69 }

```


在 `inet_create` 中，我们会先看到一个循环 `list_for_each_entry_rcu`。在这里，第二个参数 `type` 开始起作用。因为循环查看的是 `inetsw[sock->type]`。


这里的 `inetsw` 也是一个数组，`type` 作为下标，里面的内容是 `struct inet_protosw`，是协议，也即 `inetsw` 数组对于每个类型有一项，这一项里面是属于这个类型的协议。

 复制代码

```
1 static struct list_head inetsw[SOCK_MAX];
2
3 static int __init inet_init(void)
4 {
5     .....
6     /* Register the socket-side information for inetsw
7     for (r = &inetsw[0]; r < &inetsw[SOCK_MAX]; ++r)
8         INIT_LIST_HEAD(r);
9     for (q = inetsw_array; q < &inetsw_array[INETSW_ARRAY_SIZE]; ++q)
10         inet_register_protosw(q);
11     .....
12 }
```

`inetsw` 数组是在系统初始化的时候初始化的，就像下面代码里面实现的一样。

首先，一个循环会将 `inetsw` 数组的每一项，都初始化为一个链表。咱们前面说了，一个 `type` 类型会包含多个 `protocol`，因而我们需要一个链表。接下来一个循环，是将 `inetsw_array` 注册到 `inetsw` 数组里面去。`inetsw_array` 的定义如下，这个数组里面的内容很重要，后面会用到它们。

 复制代码

```
1 static struct inet_protosw inetsw_array[] =
2 {
3     {
4         .type =      SOCK_STREAM,
5         .protocol =  IPPROTO_TCP,
6         .prot =      &tcp_prot,
7         .ops =       &inet_stream_ops,
8         .flags =     INET_PROTOSW_PERMANENT |
9                     INET_PROTOSW_ICSK,
10    },
11    {
12        .type =      SOCK_DGRAM,
13        .protocol =  IPPROTO_UDP,
14        .prot =      &udp_prot,
15        .ops =       &inet_dgram_ops,
16        .flags =     INET_PROTOSW_PERMANENT,
17    },
18    {
19        .type =      SOCK_DGRAM,
20        .protocol =  IPPROTO_ICMP,
21        .prot =      &ping_prot,
22        .ops =       &inet_sockraw_ops,
23        .flags =     INET_PROTOSW_REUSE,
24    },
25    {
```

```
26         .type =          SOCK_RAW,
27         .protocol =      IPPROTO_IP,    /* wild card */
28         .prot =          &raw_prot,
29         .ops =           &inet_sockraw_ops,
30         .flags =         INET_PROTOSW_REUSE,
31     }
32 }
```




我们回到 `inet_create` 的 `list_for_each_entry_rcu` 循环中。到这里就好理解了，这是在 `inetsw` 数组中，根据 `type` 找到属于这个类型的列表，然后依次比较列表中的 `struct inet_protosw` 的 `protocol` 是不是用户指定的 `protocol`；如果是，就得到了符合用户指定的 `family->type->protocol` 的 `struct inet_protosw *answer` 对象。

接下来，`struct socket *sock` 的 `ops` 成员变量，被赋值为 `answer` 的 `ops`。对于 TCP 来讲，就是 `inet_stream_ops`。后面任何用户对于这个 `socket` 的操作，都是通过 `inet_stream_ops` 进行的。

接下来，我们创建一个 `struct sock *sk` 对象。这里比较让人困惑。`socket` 和 `sock` 看起来几乎一样，容易让人混淆，这里需要说明一下，`socket` 是用于负责对上给用户提供接口，并且和文件系统关联。而 `sock`，负责向下对接内核网络协议栈。

在 `sk_alloc` 函数中, `struct inet_protosw *answer` 结构的 `tcp_prot` 赋值给了 `struct sock *sk` 的 `sk_prot` 成员。
`tcp_prot` 的定义如下, 里面定义了很多的函数, 都是 `sock` 之下内核协议栈的动作。

 复制代码


```
1 struct proto tcp_prot = {
2     .name           = "TCP",
3     .owner          = THIS_MODULE,
4     .close          = tcp_close,
5     .connect        = tcp_v4_connect,
6     .disconnect     = tcp_disconnect,
7     .accept         = inet_csk_accept,
8     .ioctl          = tcp_ioctl,
9     .init           = tcp_v4_init_sock,
10    .destroy         = tcp_v4_destroy_sock,
11    .shutdown        = tcp_shutdown,
12    .setsockopt      = tcp_setsockopt,
13    .getsockopt      = tcp_getsockopt,
14    .keepalive       = tcp_set_keepalive,
15    .recvmsg         = tcp_recvmsg,
16    .sendmsg         = tcp_sendmsg,
17    .sendpage        = tcp_sendpage,
18    .backlog_rcv     = tcp_v4_do_rcv,
19    .release_cb      = tcp_release_cb,
20    .hash            = inet_hash,
21    .get_port        = inet_csk_get_port,
22    .....
23 }
```

在 `inet_create` 函数中，接下来创建一个 `struct inet_sock` 结构，这个结构一开始就是 `struct sock`，然后扩展了一些其他的信息，剩下的代码就填充这些信息。这一幕我们会经常看到，将一个结构放在另一个结构的开始位置，然后扩展一些成员，通过对于指针的强制类型转换，来访问这些成员。

socket 的创建至此结束。

解析 `bind` 函数

接下来，我们来看 `bind`。


 复制代码

```
1 SYSCALL_DEFINE3(bind, int, fd, struct sockaddr __user *
2 {
3     struct socket *sock;
4     struct sockaddr_storage address;
5     int err, fput_needed;
6
7     sock = sockfd_lookup_light(fd, &err, &fput_neec
8     if (sock) {
9         err = move_addr_to_kernel(umyaddr, addr
10         if (err >= 0) {
11             err = sock->ops->bind(sock,
12                                     (
13                                     &
14             }
15         fput_light(sock->file, fput_needed);
```

```
16         }
17         return err;
18     }
```



在 bind 中，sockfd_lookup_light 会根据 fd 文件描述符，找到 struct socket 结构。然后将 sockaddr 从用户态拷贝到内核态，然后调用 struct socket 结构里面 ops 的 bind 函数。根据前面创建 socket 的时候的设定，调用的是 inet_stream_ops 的 bind 函数，也即调用 inet_bind。

 复制代码

```
1 int inet_bind(struct socket *sock, struct sockaddr *uac
2 {
3     struct sockaddr_in *addr = (struct sockaddr_in
4     struct sock *sk = sock->sk;
5     struct inet_sock *inet = inet_sk(sk);
6     struct net *net = sock_net(sk);
7     unsigned short snum;
8     .....
9     snum = ntohs(addr->sin_port);
10    .....
11    inet->inet_rcv_saddr = inet->inet_saddr = addr-
12    /* Make sure we are allowed to bind here. */
13    if ((snum || !inet->bind_address_no_port) &&
14        sk->sk_prot->get_port(sk, snum)) {
15    .....
16    }
17    inet->inet_sport = htons(inet->inet_num);
18    inet->inet_daddr = 0;
```

```
19         inet->inet_dport = 0;
20         sk_dst_reset(sk);
21     }
```




bind 里面会调用 sk_prot 的 get_port 函数，也即 inet_csk_get_port 来检查端口是否冲突，是否可以绑定。如果允许，则会设置 struct inet_sock 的本方的地址 inet_saddr 和本方的端口 inet_sport，对方的地址 inet_daddr 和对方的端口 inet_dport 都初始化为 0。

bind 的逻辑相对比较简单，就到这里了。

解析 listen 函数

接下来我们来看 listen。


 复制代码

```
1 SYSCALL_DEFINE2(listen, int, fd, int, backlog)
2 {
3     struct socket *sock;
4     int err, fput_needed;
5     int somaxconn;
6
7     sock = sockfd_lookup_light(fd, &err, &fput_neec
8     if (sock) {
9         somaxconn = sock_net(sock->sk)->core.sy
10         if ((unsigned int)backlog > somaxconn)
```

```
11             backlog = somaxconn;
12             err = sock->ops->listen(sock, backlog);
13             fput_light(sock->file, fput_needed);
14         }
15         return err;
16     }
```




在 listen 中，我们还是通过 sockfd_lookup_light，根据 fd 文件描述符，找到 struct socket 结构。接着，我们调用 struct socket 结构里面 ops 的 listen 函数。根据前面创建 socket 的时候的设定，调用的是 inet_stream_ops 的 listen 函数，也即调用 inet_listen。

 复制代码

```
1 int inet_listen(struct socket *sock, int backlog)
2 {
3     struct sock *sk = sock->sk;
4     unsigned char old_state;
5     int err;
6     old_state = sk->sk_state;
7     /* Really, if the socket is already in listen s
8      * we can only allow the backlog to be adjustec
9      */
10    if (old_state != TCP_LISTEN) {
11        err = inet_csk_listen_start(sk, backlog);
12    }
13    sk->sk_max_ack_backlog = backlog;
14 }
```


如果这个 socket 还不在于 TCP_LISTEN 状态，会调用 inet_csk_listen_start 进入监听状态。

 复制代码

```
1 int inet_csk_listen_start(struct sock *sk, int backlog)
2 {
3     struct inet_connection_sock *icsk = inet_csk(sk)
4     struct inet_sock *inet = inet_sk(sk);
5     int err = -EADDRINUSE;
6
7     reqsk_queue_alloc(&icsk->icsk_accept_queue);
8
9     sk->sk_max_ack_backlog = backlog;
10    sk->sk_ack_backlog = 0;
11    inet_csk_delack_init(sk);
12
13    sk_state_store(sk, TCP_LISTEN);
14    if (!sk->sk_prot->get_port(sk, inet->inet_num))
15        .....
16    }
17    .....
18 }
```

这里面建立了一个新的结构 inet_connection_sock，这个结构一开始是 struct inet_sock，inet_csk 其实做了一次强制类型转换，扩大了结构，看到了吧，又是这个套路。

struct inet_connection_sock 结构比较复杂。如果打开它，你能看到处于各种状态的队列，各种超时时间、拥塞控制等字眼。我们说 TCP 是面向连接的，就是客户端和服务端都是有一个结构维护连接的状态，就是指这个结构。我们这里先不详细分析里面的变量，因为太多了，后面我们遇到一个分析一个。

首先，我们遇到的是 icsk_accept_queue。它是干什么的呢？

在 TCP 的状态里面，有一个 listen 状态，当调用 listen 函数之后，就会进入这个状态，虽然我们写程序的时候，一般要等待服务端调用 accept 后，等待在哪里的时候，让客户端就发起连接。其实服务端一旦处于 listen 状态，不用 accept，客户端也能发起连接。其实 TCP 的状态中，没有一个是否被 accept 的状态，那 accept 函数的作用是什么呢？

在内核中，为每个 Socket 维护两个队列。一个是已经建立了连接的队列，这时候连接三次握手已经完毕，处于 established 状态；一个是还没有完全建立连接的队列，这个时候三次握手还没完成，处于 syn_rcvd 的状态。


服务端调用 `accept` 函数，其实是在第一个队列中拿出一个已经完成的连接进行处理。如果还没有完成就阻塞等待。这里的 `icsk_accept_queue` 就是第一个队列。

初始化完之后，将 TCP 的状态设置为 `TCP_LISTEN`，再次调用 `get_port` 判断端口是否冲突。

至此，`listen` 的逻辑就结束了。

解析 `accept` 函数

接下来，我们解析服务端调用 `accept`。

 复制代码

```
1 SYSCALL_DEFINE3(accept, int, fd, struct sockaddr __user
2                 int __user *, upeer_addrln)
3 {
4     return sys_accept4(fd, upeer_sockaddr, upeer_ac
5 }
6
7 SYSCALL_DEFINE4(accept4, int, fd, struct sockaddr __use
8                 int __user *, upeer_addrln, int, flags
9 {
10     struct socket *sock, *newsock;
11     struct file *newfile;
12     int err, len, newfd, fput_needed;
13     struct sockaddr_storage address;
14     .....
15     sock = sockfd_lookup_light(fd, &err, &fput_need
```

```

16         newsock = sock_alloc();
17         newsock->type = sock->type;
18         newsock->ops = sock->ops;
19         newfd = get_unused_fd_flags(flags);
20         newfile = sock_alloc_file(newsock, flags, sock-
21         err = sock->ops->accept(sock, newsock, sock->fi
22         if (upeer_sockaddr) {
23             if (newsock->ops->getname(newsock, (str
24             }
25             err = move_addr_to_user(&address,
26                                     len, upeer_sock
27         }
28         fd_install(newfd, newfile);
29         .....
30     }

```

accept 函数的实现，印证了 socket 的原理中说的那样，原来的 socket 是监听 socket，这里我们会找到原来的 struct socket，并基于它去创建一个新的 newsock。这才是连接 socket。除此之外，我们还会创建一个新的 struct file 和 fd，并关联到 socket。

这里面还会调用 struct socket 的 sock->ops->accept，也即会调用 inet_stream_ops 的 accept 函数，也即 inet_accept。


```

1 int inet_accept(struct socket *sock, struct socket *new
2 {
3     struct sock *sk1 = sock->sk;
4     int err = -EINVAL;
5     struct sock *sk2 = sk1->sk_prot->accept(sk1, fl
6     sock_rps_record_flow(sk2);
7     sock_graft(sk2, newsock);
8     newsock->state = SS_CONNECTED;
9 }

```



inet_accept 会调用 struct sock 的 sk1->sk_prot->accept, 也即 tcp_prot 的 accept 函数, inet_csk_accept 函数。

 复制代码

```

1 /*
2  * This will accept the next outstanding connection.
3  */
4 struct sock *inet_csk_accept(struct sock *sk, int flags
5 {
6     struct inet_connection_sock *icsk = inet_csk(sk
7     struct request_sock_queue *queue = &icsk->icsk_
8     struct request_sock *req;
9     struct sock *newsk;
10    int error;
11
12    if (sk->sk_state != TCP_LISTEN)
13        goto out_err;
14
15    /* Find already established connection */

```

```

16         if (reqsk_queue_empty(queue)) {
17             long timeo = sock_rcvtimeo(sk, flags &
18                 error = inet_csk_wait_for_connect(sk, t
19         }
20         req = reqsk_queue_remove(queue, sk);
21         newsk = req->sk;
22         .....
23     }
24
25     /*
26      * Wait for an incoming connection, avoid race conditio
27      * with the socket locked.
28      */
29     static int inet_csk_wait_for_connect(struct sock *sk, l
30     {
31         struct inet_connection_sock *icsk = inet_csk(sk
32         DEFINE_WAIT(wait);
33         int err;
34         for (;;) {
35             prepare_to_wait_exclusive(sk_sleep(sk),
36                                     TASK_INTERRUPTIB
37             release_sock(sk);
38             if (reqsk_queue_empty(&icsk->icsk_accep
39                 timeo = schedule_timeout(timeo)
40             sched_annotate_sleep();
41             lock_sock(sk);
42             err = 0;
43             if (!reqsk_queue_empty(&icsk->icsk_acce
44                 break;
45             err = -EINVAL;
46             if (sk->sk_state != TCP_LISTEN)
47                 break;
48             err = sock_intr_errno(timeo);
49             if (signal_pending(current))
50                 break;

```

```
51             err = -EAGAIN;
52             if (!timeo)
53                 break;
54         }
55         finish_wait(sk_sleep(sk), &wait);
56         return err;
57     }
```

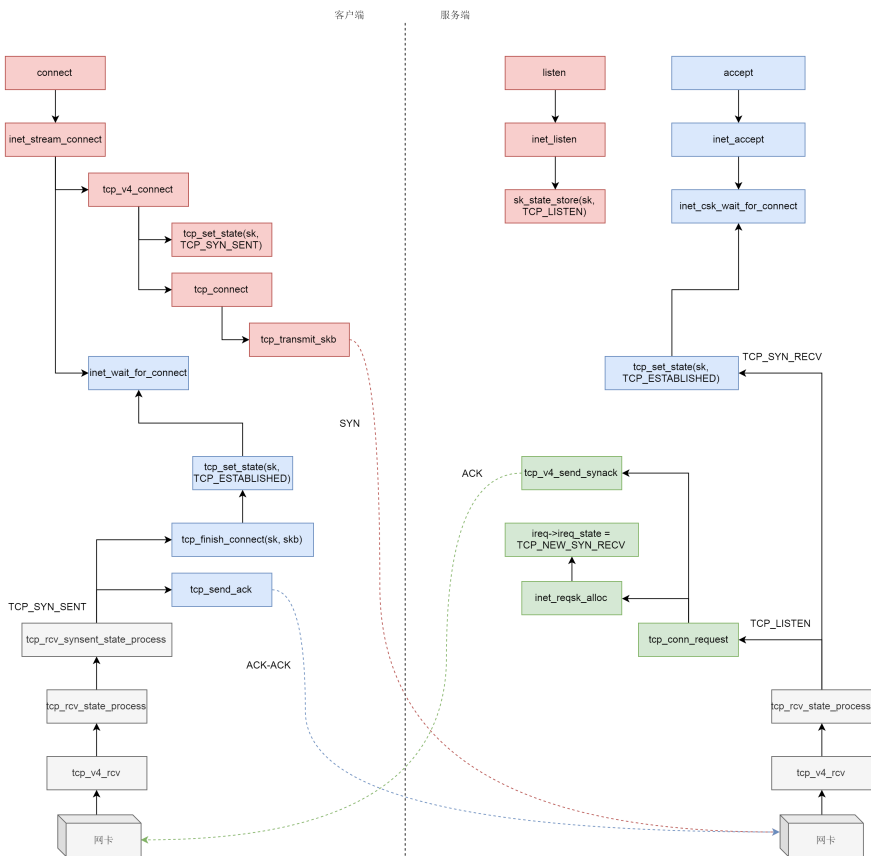


inet_csk_accept 的实现，印证了上面我们讲的两个队列的逻辑。如果 icsk_accept_queue 为空，则调用 inet_csk_wait_for_connect 进行等待；等待的时候，调用 schedule_timeout，让出 CPU，并且将进程状态设置为 TASK_INTERRUPTIBLE。

如果再次 CPU 醒来，我们会接着判断 icsk_accept_queue 是否为空，同时也会调用 signal_pending 看有没有信号可以处理。一旦 icsk_accept_queue 不为空，就从 inet_csk_wait_for_connect 中返回，在队列中取出一个 struct sock 对象赋值给 newsk。

解析 connect 函数

什么情况下，icsk_accept_queue 才不为空呢？当然是三次握手结束才可以。接下来我们来分析三次握手的过程。



三次握手一般是由客户端调用 connect 发起。

复制代码

```

1 SYSCALL_DEFINE3(connect, int, fd, struct sockaddr __use
2                     int, addrlen)
3 {
4     struct socket *sock;
5     struct sockaddr_storage address;
6     int err, fput_needed;
7     sock = sockfd_lookup_light(fd, &err, &fput_need

```




```

8         err = move_addr_to_kernel(uservaddr, addrlen, &
9         err = sock->ops->connect(sock, (struct sockaddr
10 }

```



connect 函数的实现一开始你应该很眼熟，还是通过 sockfd_lookup_light，根据 fd 文件描述符，找到 struct socket 结构。接着，我们会调用 struct socket 结构里面 ops 的 connect 函数，根据前面创建 socket 的时候的设定，调用 inet_stream_ops 的 connect 函数，也即调用 inet_stream_connect。

 复制代码

```

1  /*
2   *      Connect to a remote host. There is regrettably
3   *      TCP 'magic' in here.
4   */
5  int __inet_stream_connect(struct socket *sock, struct s
6                          int addr_len, int flags, int
7  {
8      struct sock *sk = sock->sk;
9      int err;
10     long timeo;
11
12     switch (sock->state) {
13     .....
14     case SS_UNCONNECTED:
15         err = -EISCONN;
16         if (sk->sk_state != TCP_CLOSE)
17             goto out;

```


```

18
19         err = sk->sk_prot->connect(sk, uaddr, &
20         sock->state = SS_CONNECTING;
21         break;
22     }
23
24     timeo = sock_sndtimeo(sk, flags & O_NONBLOCK);
25
26     if ((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF
27     .....
28         if (!timeo || !inet_wait_for_connect(sk
29             goto out;
30
31         err = sock_intr_errno(timeo);
32         if (signal_pending(current))
33             goto out;
34     }
35     sock->state = SS_CONNECTED;
36 }

```



在 `_inet_stream_connect` 里面，我们发现，如果 socket 处于 `SS_UNCONNECTED` 状态，那就调用 struct sock 的 `sk->sk_prot->connect`，也即 `tcp_prot` 的 `connect` 函数——`tcp_v4_connect` 函数。

 复制代码

```

1 int tcp_v4_connect(struct sock *sk, struct sockaddr *ua
2 {
3     struct sockaddr_in *usin = (struct sockaddr_in
4     struct inet_sock *inet = inet_sk(sk);

```


```

5      struct tcp_sock *tp = tcp_sk(sk);
6      __be16 orig_sport, orig_dport;
7      __be32 daddr, nexthop;
8      struct flowi4 *fl4;
9      struct rtable *rt;
10     .....
11     orig_sport = inet->inet_sport;
12     orig_dport = usin->sin_port;
13     rt = ip_route_connect(fl4, nexthop, inet->inet_
14                             RT_CONN_FLAGS(sk), sk->sk
15                             IPPROTO_TCP,
16                             orig_sport, orig_dport, s
17     .....
18     tcp_set_state(sk, TCP_SYN_SENT);
19     err = inet_hash_connect(tcp_death_row, sk);
20     sk_set_txhash(sk);
21     rt = ip_route_newports(fl4, rt, orig_sport, ori
22                             inet->inet_sport, inet->
23     /* OK, now commit destination to socket. */
24     sk->sk_gso_type = SKB_GSO_TCPV4;
25     sk_setup_caps(sk, &rt->dst);
26     if (likely(!tp->repair)) {
27         if (!tp->write_seq)
28             tp->write_seq = secure_tcp_seq(
29
30
31
32             tp->tsoffset = secure_tcp_ts_off(sock_r
33                                     inet->
34                                     inet->
35     }
36     rt = NULL;
37     .....
38     err = tcp_connect(sk);
39     .....

```

在 `tcp_v4_connect` 函数中, `ip_route_connect` 其实是一个路由的选择。为什么呢? 因为三次握手马上就要发送一个 SYN 包了, 这就要凑齐源地址、源端口、目标地址、目标端口。目标地址和目标端口是服务端的, 已经知道源端口是客户端随机分配的, 源地址应该用哪一个呢? 这时候要选择一条路由, 看从哪个网卡出去, 就应该填写哪个网卡的 IP 地址。

接下来, 在发送 SYN 之前, 我们先将客户端 socket 的状态设置为 `TCP_SYN_SENT`。然后初始化 TCP 的 seq num, 也即 `write_seq`, 然后调用 `tcp_connect` 进行发送。

 复制代码

```
1 /* Build a SYN and send it off. */
2 int tcp_connect(struct sock *sk)
3 {
4     struct tcp_sock *tp = tcp_sk(sk);
5     struct sk_buff *buff;
6     int err;
7     .....
8     tcp_connect_init(sk);
9     .....
10    buff = sk_stream_alloc_skb(sk, 0, sk->sk_allocation
11    .....
```

```

12         tcp_init_nondata_skb(buff, tp->write_seq++, TCF
13         tcp_mstamp_refresh(tp);
14         tp->retrans_stamp = tcp_time_stamp(tp);
15         tcp_connect_queue_skb(sk, buff);
16         tcp_ecn_send_syn(sk, buff);
17
18         /* Send off SYN; include data in Fast Open. */
19         err = tp->fastopen_req ? tcp_send_syn_data(sk,
20             tcp_transmit_skb(sk, buff, 1, sk->sk_allc
21         .....
22         tp->snd_nxt = tp->write_seq;
23         tp->pushed_seq = tp->write_seq;
24         buff = tcp_send_head(sk);
25         if (unlikely(buff)) {
26             tp->snd_nxt      = TCP_SKB_CB(buff)->sec
27             tp->pushed_seq   = TCP_SKB_CB(buff)->sec
28         }
29         .....
30         /* Timer for repeating the SYN until an answer.
31         inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS
32             inet_csk(sk)->icsk_rt
33         return 0;
34     }

```



在 `tcp_connect` 中，有一个新的结构 `struct tcp_sock`，如果打开他，你会发现他是 `struct inet_connection_sock` 的一个扩展，`struct inet_connection_sock` 在 `struct tcp_sock` 开头的位置，通过强制类型转换访问，故伎重演又一次。


struct tcp_sock 里面维护了更多的 TCP 的状态，咱们同样是遇到了再分析。

接下来 tcp_init_nondata_skb 初始化一个 SYN 包，tcp_transmit_skb 将 SYN 包发送出去，inet_csk_reset_xmit_timer 设置了一个 timer，如果 SYN 发送不成功，则再次发送。

发送网络包的过程，我们放到下一节讲解。这里我们姑且认为 SYN 已经发送出去了。

我们回到 __inet_stream_connect 函数，在调用 sk->sk_prot->connect 之后，inet_wait_for_connect 会一直等待客户端收到服务端的 ACK。而我们知道，服务端在 accept 之后，也是在等待中。

网络包是如何接收的呢？对于解析的详细过程，我们会在下一节讲解，这里为了解析三次握手，我们简单的看网络包接收到 TCP 层做的部分事情。

 复制代码


```
1 static struct net_protocol tcp_protocol = {
2     .early_demux      =      tcp_v4_early_demux,
3     .early_demux_handler = tcp_v4_early_demux,
4     .handler           =      tcp_v4_rcv,
```

```

5      .err_handler      =      tcp_v4_err,
6      .no_policy        =      1,
7      .netns_ok         =      1,
8      .icmp_strict_tag_validation = 1,
9  }

```

我们通过 struct net_protocol 结构中的 handler 进行接收，调用的函数是 tcp_v4_rcv。接下来的调用链为 tcp_v4_rcv->tcp_v4_do_rcv->tcp_rcv_state_process。tcp_rcv_state_process，顾名思义，是用来处理接收一个网络包后引起状态变化的。

 复制代码

```

1  int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
2  {
3      struct tcp_sock *tp = tcp_sk(sk);
4      struct inet_connection_sock *icsk = inet_csk(sk);
5      const struct tcphdr *th = tcp_hdr(skb);
6      struct request_sock *req;
7      int queued = 0;
8      bool acceptable;
9
10     switch (sk->sk_state) {
11     .....
12     case TCP_LISTEN:
13     .....
14         if (th->syn) {
15             acceptable = icsk->icsk_af_ops->tcp_listen_
16             if (!acceptable)

```


```

17                                     return 1;
18                                     consume_skb(skb);
19                                     return 0;
20                                 }
21     .....
22 }

```



目前服务端是处于 TCP_LISTEN 状态的，而且发过来的包是 SYN，因而就有了上面的代码，调用 `icsk->icsk_af_ops->conn_request` 函数。struct `inet_connection_sock` 对应的操作是 `inet_connection_sock_af_ops`，按照下面的定义，其实调用的是 `tcp_v4_conn_request`。

 复制代码

```

1  const struct inet_connection_sock_af_ops ipv4_specific
2      .queue_xmit          = ip_queue_xmit,
3      .send_check          = tcp_v4_send_check,
4      .rebuild_header      = inet_sk_rebuild_header,
5      .sk_rx_dst_set       = inet_sk_rx_dst_set,
6      .conn_request        = tcp_v4_conn_request,
7      .syn_recv_sock       = tcp_v4_syn_recv_sock,
8      .net_header_len      = sizeof(struct iphdr),
9      .setsockopt           = ip_setsockopt,
10     .getsockopt           = ip_getsockopt,
11     .addr2sockaddr        = inet_csk_addr2sockaddr,
12     .sockaddr_len         = sizeof(struct sockaddr_in)
13     .mtu_reduced          = tcp_v4_mtu_reduced,
14 };


```


tcp_v4_conn_request 会调用 tcp_conn_request，这个函数也比较长，里面调用了 send_synack，但实际调用的是 tcp_v4_send_synack。具体发送的过程我们不去管它，看注释我们能知道，这是收到了 SYN 后，回复一个 SYN-ACK，回复完毕后，服务端处于 TCP_SYN_RECV。

复制代码

[illegible]

这个时候，轮到客户端接收网络包了。都是 TCP 协议栈，所以过程和服务端没有太多区别，还是会走到 `tcp_rcv_state_process` 函数的，只不过由于客户端目前处于 `TCP_SYN_SENT` 状态，就进入了下面的代码分支。


 复制代码

```
1 int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
2 {
3     struct tcp_sock *tp = tcp_sk(sk);
4     struct inet_connection_sock *icsk = inet_csk(sk);
5     const struct tcphdr *th = tcp_hdr(skb);
6     struct request_sock *req;
7     int queued = 0;
8     bool acceptable;
9
10    switch (sk->sk_state) {
11    .....
12    case TCP_SYN_SENT:
13        tp->rx_opt.saw_tstamp = 0;
14        tcp_mstamp_refresh(tp);
15        queued = tcp_rcv_synsent_state_process(
16            sk, th, req, &acceptable);
17        if (queued >= 0)
18            return queued;
19        /* Do step6 onward by hand. */
20        tcp_urg(sk, skb, th);
21        __kfree_skb(skb);
22        tcp_data_snd_check(sk);
23        return 0;
24    .....
25 }
```

tcp_rcv_synsent_state_process 会调用 tcp_send_ack, 发送一个 ACK-ACK, 发送后客户端处于 TCP_ESTABLISHED 状态。

又轮到服务端接收网络包了, 我们还是归

tcp_rcv_state_process 函数处理。由于服务端目前处于状态 TCP_SYN_RECV 状态, 因而又走了另外的分支。当收到这个网络包的时候, 服务端也处于 TCP_ESTABLISHED 状态, 三次握手结束。

 复制代码

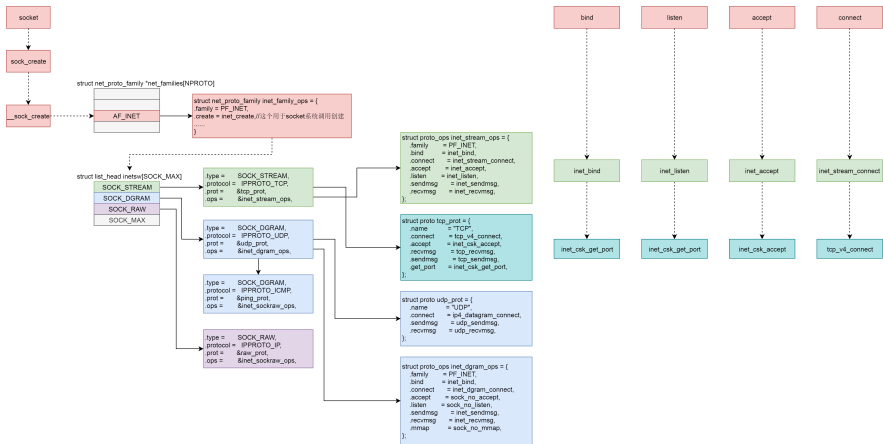
```
1 int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
2 {
3     struct tcp_sock *tp = tcp_sk(sk);
4     struct inet_connection_sock *icsk = inet_csk(sk);
5     const struct tcphdr *th = tcp_hdr(skb);
6     struct request_sock *req;
7     int queued = 0;
8     bool acceptable;
9     .....
10    switch (sk->sk_state) {
11        case TCP_SYN_RECV:
12            if (req) {
13                inet_csk(sk)->icsk_retransmits
14                reqsk_fastopen_remove(sk, req,
15            } else {
16                /* Make sure socket is routed,
17                icsk->icsk_af_ops->rebuild_heac
```

```
18             tcp_call_bpf(sk, BPF_SOCKET_OPS_F
19             tcp_init_congestion_control(sk)
20
21             tcp_mtup_init(sk);
22             tp->copied_seq = tp->rcv_nxt;
23             tcp_init_buffer_space(sk);
24         }
25         smp_mb();
26         tcp_set_state(sk, TCP_ESTABLISHED);
27         sk->sk_state_change(sk);
28         if (sk->sk_socket)
29             sk_wake_async(sk, SOCK_WAKE_IO,
30             tp->snd_una = TCP_SKB_CB(skb)->ack_seq;
31             tp->snd_wnd = ntohs(th->window) << tp->
32             tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);
33             break;
34     .....
35 }
```



总结时刻

这一节除了网络包的接收和发送，其他的系统调用我们都分析到了。可以看出来，它们有一个统一的数据结构和流程。具体如下图所示：



首先，Socket 系统调用会有三级参数 family、type、protocol，通过这三级参数，分别在 net_proto_family 表 中找到 type 链表，在 type 链表中找到 protocol 对应的操作。这个操作分为两层，对于 TCP 协议来讲，第一层是 inet_stream_ops 层，第二层是 tcp_prot 层。

于是，接下来的系统调用规律就都一样了：

bind 第一层调用 inet_stream_ops 的 inet_bind 函数，第二层调用 tcp_prot 的 inet_csk_get_port 函数；

listen 第一层调用 inet_stream_ops 的 inet_listen 函数，第二层调用 tcp_prot 的 inet_csk_get_port 函数；

accept 第一层调用 inet_stream_ops 的 inet_accept 函数，第二层调用 tcp_prot 的 inet_csk_accept 函数；

connect 第一层调用 `inet_stream_ops` 的 `inet_stream_connect` 函数，第二层调用 `tcp_prot` 的 `tcp_v4_connect` 函数。

课堂练习

TCP 的三次握手协议非常重要，请你务必跟着代码走读一遍。另外我们这里重点关注了 TCP 的场景，请走读代码的时候，也看一下 UDP 是如何实现各层的函数的。

欢迎留言和我分享你的疑惑和见解，也欢迎可以收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。




趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院
云计算技术部首席架构师



新版升级：点击「 请朋友读」，10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 43 | Socket通信：遇上特大项目，要学会和其他公司...

精选留言 (2)

写留言



飞翔

2019-07-08

syn到底是个什么东西呀？是个integer还是char类型

展开 ∨



杨怀

2019-07-08

老师好，同一个TCP链接上先后发送2次rpc请求，后发送的请求其结果先返回，先发送的请求结果后返回，这样有没有问题呢，系统能区分各自的返回结果么，靠什么机制保证的呢？一直没有想明白

展开 ∨



