

## 55 | 网络虚拟化：如何成立独立的合作部？

2019-08-02 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 17:34 大小 16.09M



上一节，我们讲了存储虚拟化，这一节我们来讲网络虚拟化。

网络虚拟化有和存储虚拟化类似的地方，例如，它们都是基于 virtio 的，因而我们在看网络虚拟化的过程中，会看到和存储虚拟化很像的数据结构和原理。但是，网络虚拟化也有自己的特殊性。例如，存储虚拟化是将宿主机上的文件作为客户机上的硬盘，而网络虚拟化需要依赖于内核协议栈进行网络包的封装与解封装。那怎么实现客户机和宿主机之间的互通呢？我们就一起来看一看。

### 解析初始化过程


我们还是从 Virtio Network Device 这个设备的初始化讲起。

```
1 static const TypeInfo device_type_info = {
2     .name = TYPE_DEVICE,
3     .parent = TYPE_OBJECT,
4     .instance_size = sizeof(DeviceState),
5     .instance_init = device_initfn,
6     .instance_post_init = device_post_init,
7     .instance_finalize = device_finalize,
8     .class_base_init = device_class_base_init,
9     .class_init = device_class_init,
10    .abstract = true,
11    .class_size = sizeof(DeviceClass),
12 };
13
14 static const TypeInfo virtio_device_info = {
15     .name = TYPE_VIRTIO_DEVICE,
16     .parent = TYPE_DEVICE,
17     .instance_size = sizeof(VirtIODevice),
18     .class_init = virtio_device_class_init,
19     .instance_finalize = virtio_device_instance_finalize,
20     .abstract = true,
21     .class_size = sizeof(VirtioDeviceClass),
22 };
23
24 static const TypeInfo virtio_net_info = {
25     .name = TYPE_VIRTIO_NET,
26     .parent = TYPE_VIRTIO_DEVICE,
27     .instance_size = sizeof(VirtIONet),
28     .instance_init = virtio_net_instance_init,
29     .class_init = virtio_net_class_init,
30 };
31
32 static void virtio_register_types(void)
33 {
34     type_register_static(&virtio_net_info);
35 }
36
37 type_init(virtio_register_types)
```

Virtio Network Device 这种类的定义是有多层继承关系的，TYPE\_VIRTIO\_NET 的父类是 TYPE\_VIRTIO\_DEVICE，TYPE\_VIRTIO\_DEVICE 的父类是 TYPE\_DEVICE，TYPE\_DEVICE 的父类是 TYPE\_OBJECT，继承关系到头了。

type\_init 用于注册这种类。这里面每一层都有 class\_init，用于从 TypeImpl 生产 xxxClass，也有 instance\_init，会将 xxxClass 初始化为实例。

TYPE\_VIRTIO\_NET 层的 class\_init 函数 virtio\_net\_class\_init，定义了 DeviceClass 的 realize 函数为 virtio\_net\_device\_realize，这一点和存储块设备是一样的。

 复制代码

```
1 static void virtio_net_device_realize(DeviceState *dev, Error **errp)
2 {
3     VirtIODevice *vdev = VIRTIO_DEVICE(dev);
4     VirtIONet *n = VIRTIO_NET(dev);
5     NetClientState *nc;
6     int i;
7     .....
8     virtio_init(vdev, "virtio-net", VIRTIO_ID_NET, n->config_size);
9
10    /*
11     * We set a lower limit on RX queue size to what it always was.
12     * Guests that want a smaller ring can always resize it without
13     * help from us (using virtio 1 and up).
14     */
15    if (n->net_conf.rx_queue_size < VIRTIO_NET_RX_QUEUE_MIN_SIZE ||
16        n->net_conf.rx_queue_size > VIRTQUEUE_MAX_SIZE ||
17        !is_power_of_2(n->net_conf.rx_queue_size)) {
18        .....
19        return;
20    }
21
22    if (n->net_conf.tx_queue_size < VIRTIO_NET_TX_QUEUE_MIN_SIZE ||
23        n->net_conf.tx_queue_size > VIRTQUEUE_MAX_SIZE ||
24        !is_power_of_2(n->net_conf.tx_queue_size)) {
25        .....
26        return;
27    }
28
29    n->max_queues = MAX(n->nic_conf.peers.queues, 1);
30    if (n->max_queues * 2 + 1 > VIRTIO_QUEUE_MAX) {
31        .....
32        return;
33    }
34    n->vqs = g_malloc0(sizeof(VirtIONetQueue) * n->max_queues);
35    n->curr_queues = 1;
36    .....
37    n->net_conf.tx_queue_size = MIN(virtio_net_max_tx_queue_size(n),
38                                    n->net_conf.tx_queue_size);
39
40    for (i = 0; i < n->max_queues; i++) {
41        virtio_net_add_queue(n, i);
42    }
43
44    n->ctrl_vq = virtio_add_queue(vdev, 64, virtio_net_handle_ctrl);
45    qemu_macaddr_default_if_unset(&n->nic_conf.macaddr);
46    memcpy(&n->mac[0], &n->nic_conf.macaddr, sizeof(n->mac));
```

```

47     n->status = VIRTIO_NET_S_LINK_UP;
48
49     if (n->netclient_type) {
50         n->nic = qemu_new_nic(&net_virtio_info, &n->nic_conf,
51                               n->netclient_type, n->netclient_name, n);
52     } else {
53         n->nic = qemu_new_nic(&net_virtio_info, &n->nic_conf,
54                               object_get_typename(OBJECT(dev)), dev->id, n);
55     }
56     .....
57 }


```

这里面创建了一个 VirtIODevice，这一点和存储虚拟化也是一样的。virtio\_init 用来初始化这个设备。VirtIODevice 结构里面有一个 VirtQueue 数组，这就是 virtio 前端和后端互相传数据的队列，最多有 VIRTIO\_QUEUE\_MAX 个。

刚才我们说的都是一样的地方，其实也有不一样的地方，我们下面来看。

你会发现，这里面有这样的语句  $n \rightarrow \text{max\_queues} * 2 + 1 > \text{VIRTIO\_QUEUE\_MAX}$ 。为什么要乘以 2 呢？这是因为，对于网络设备来讲，应该分发送队列和接收队列两个方向，所以乘以 2。

接下来，我们调用 virtio\_net\_add\_queue 来初始化队列，可以看出来，这里面就有发送 tx\_vq 和接收 rx\_vq 两个队列。

 复制代码

```

1  typedef struct VirtIONetQueue {
2      VirtQueue *rx_vq;
3      VirtQueue *tx_vq;
4      QEMUTimer *tx_timer;
5      QEMUBH *tx_bh;
6      uint32_t tx_waiting;
7      struct {
8          VirtQueueElement *elem;
9      } async_tx;
10     struct VirtIONet *n;
11 } VirtIONetQueue;
12
13 static void virtio_net_add_queue(VirtIONet *n, int index)
14 {
15     VirtIODevice *vdev = VIRTIO_DEVICE(n);
16

```


```

17     n->vqs[index].rx_vq = virtio_add_queue(vdev, n->net_conf.rx_queue_size, virtio_net_l
18
19     .....
20
21     n->vqs[index].tx_vq = virtio_add_queue(vdev, n->net_conf.tx_queue_size, virtio_net_l
22     n->vqs[index].tx_bh = qemu_bh_new(virtio_net_tx_bh, &n->vqs[index]);
23     n->vqs[index].n = n;
24 }

```



每个 VirtQueue 中，都有一个 vring 用来维护这个队列里面的数据；另外还有函数 virtio\_net\_handle\_rx 用于处理网络包的接收；函数 virtio\_net\_handle\_tx\_bh 用于网络包的发送，这个函数我们后面会用到。

 复制代码

```

1 NICState *qemu_new_nic(NetClientInfo *info,
2                       NICConf *conf,
3                       const char *model,
4                       const char *name,
5                       void *opaque)
6 {
7     NetClientState **peers = conf->peers.ncs;
8     NICState *nic;
9     int i, queues = MAX(1, conf->peers.queues);
10    .....
11    nic = g_malloc0(info->size + sizeof(NetClientState) * queues);
12    nic->ncs = (void *)nic + info->size;
13    nic->conf = conf;
14    nic->opaque = opaque;
15
16    for (i = 0; i < queues; i++) {
17        qemu_net_client_setup(&nic->ncs[i], info, peers[i], model, name, NULL);
18        nic->ncs[i].queue_index = i;
19    }
20
21    return nic;
22 }
23
24 static void qemu_net_client_setup(NetClientState *nc,
25                                  NetClientInfo *info,
26                                  NetClientState *peer,
27                                  const char *model,
28                                  const char *name,
29                                  NetClientDestructor *destructor)
30 {
31     nc->info = info;
32     nc->model = g_strdup(model);

```

```

33     if (name) {
34         nc->name = g_strdup(name);
35     } else {
36         nc->name = assign_name(nc, model);
37     }
38
39     QTAILQ_INSERT_TAIL(&net_clients, nc, next);
40
41     nc->incoming_queue = qemu_new_net_queue(qemu_deliver_packet_iov, nc);
42     nc->destructor = destructor;
43     QTAILQ_INIT(&nc->filters);
44 }


```

接下来，`qemu_new_nic` 会创建一个虚拟机里面的网卡。

## qemu 的启动过程中的网络虚拟化

初始化过程解析完毕以后，我们接下来从 `qemu` 的启动过程看起。

对于网卡的虚拟化，`qemu` 的启动参数里面有关的是下面两行：


 复制代码

```

1 -netdev tap,fd=32,id=hostnet0,vhost=on,vhostfd=37
2 -device virtio-net-pci,netdev=hostnet0,id=net0,mac=fa:16:3e:d1:2d:99,bus=pci.0,addr=0x3

```

`qemu` 的 `main` 函数会调用 `net_init_clients` 进行网络设备的初始化，可以解析 `net` 参数，也可以在 `net_init_clients` 中解析 `netdev` 参数。

 复制代码

```

1 int net_init_clients(Error **errp)
2 {
3     QTAILQ_INIT(&net_clients);
4     if (qemu_opts_foreach(qemu_find_opts("netdev"),
5                           net_init_netdev, NULL, errp)) {
6         return -1;
7     }
8     if (qemu_opts_foreach(qemu_find_opts("nic"), net_param_nic, NULL, errp)) {
9         return -1;
10    }
11    if (qemu_opts_foreach(qemu_find_opts("net"), net_init_client, NULL, errp)) {

```


```

12         return -1;
13     }
14     return 0;
15 }

```

net\_init\_clients 会解析参数。上面的参数 netdev 会调用 net\_init\_netdev->net\_client\_init->net\_client\_init1。

net\_client\_init1 会根据不同的 driver 类型，调用不同的初始化函数。


 复制代码

```

1 static int (* const net_client_init_fun[NET_CLIENT_DRIVER__MAX])(
2     const Netdev *netdev,
3     const char *name,
4     NetClientState *peer, Error **errp) = {
5     [NET_CLIENT_DRIVER_NIC]      = net_init_nic,
6     [NET_CLIENT_DRIVER_TAP]      = net_init_tap,
7     [NET_CLIENT_DRIVER_SOCKET]   = net_init_socket,
8     [NET_CLIENT_DRIVER_HUBPORT]  = net_init_hubport,
9     .....
10 };

```

由于我们配置的 driver 的类型是 tap，因而这里会调用 net\_init\_tap->net\_tap\_init->tap\_open。

 复制代码

```

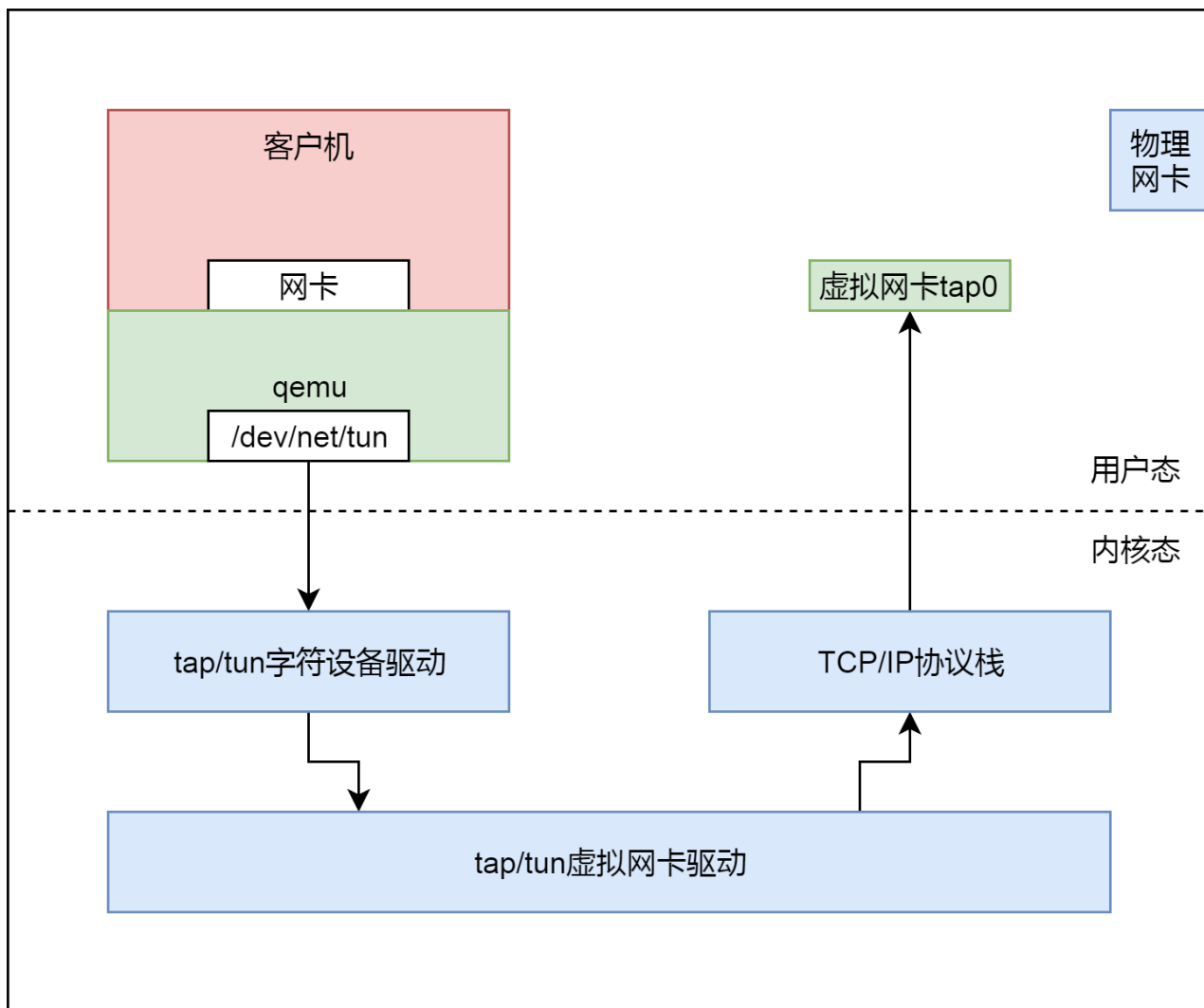
1 #define PATH_NET_TUN "/dev/net/tun"
2
3 int tap_open(char *ifname, int ifname_size, int *vnet_hdr,
4             int vnet_hdr_required, int mq_required, Error **errp)
5 {
6     struct ifreq ifr;
7     int fd, ret;
8     int len = sizeof(struct virtio_net_hdr);
9     unsigned int features;
10
11     TFR(fd = open(PATH_NET_TUN, O_RDWR));
12     memset(&ifr, 0, sizeof(ifr));
13     ifr.ifr_flags = IFF_TAP | IFF_NO_PI;
14
15     if (ioctl(fd, TUNGETFEATURES, &features) == -1) {

```

```
16         features = 0;
17     }
18
19     if (features & IFF_ONE_QUEUE) {
20         ifr.ifr_flags |= IFF_ONE_QUEUE;
21     }
22
23     if (*vnet_hdr) {
24         if (features & IFF_VNET_HDR) {
25             *vnet_hdr = 1;
26             ifr.ifr_flags |= IFF_VNET_HDR;
27         } else {
28             *vnet_hdr = 0;
29         }
30         ioctl(fd, TUNSETVNETHDRSZ, &len);
31     }
32     .....
33     ret = ioctl(fd, TUNSETIFF, (void *) &ifr);
34     .....
35     fcntl(fd, F_SETFL, O_NONBLOCK);
36     return fd;
37 }
```

在 `tap_open` 中，我们打开一个文件 `"/dev/net/tun"`，然后通过 `ioctl` 操作这个文件。这是 Linux 内核的一项机制，和 KVM 机制很像。其实这就是一种通过打开这个字符设备文件，然后通过 `ioctl` 操作这个文件和内核打交道，来使用内核的能力。





为什么需要使用内核的机制呢？因为网络包需要从虚拟机里面发送到虚拟机外面，发送到宿主机上的时候，必须是一个正常的网络包才能被转发。要形成一个网络包，我们那就需要经过复杂的协议栈，协议栈的复杂咱们在[发送网络包](#)那一节讲过了。

客户机会将网络包发送给 qemu。qemu 自己没有网络协议栈，现去实现一个也不可能，太复杂了。于是，它就要借助内核的力量。

qemu 会将客户机发给它的网络包，然后转换成为文件流，写入 `/dev/net/tun` 字符设备。就像写一个文件一样。内核中 TUN/TAP 字符设备驱动会收到这个写入的文件流，然后交给 TUN/TAP 的虚拟网卡驱动。这个驱动会将文件流再次转成网络包，交给 TCP/IP 栈，最终从虚拟 TAP 网卡 tap0 发出来，成为标准的网络包。后面我们会看到这个过程。


现在我们到内核里面，看一看打开 `/dev/net/tun` 字符设备后，内核会发生什么事情。内核的实现在 `drivers/net/tun.c` 文件中。这是一个字符设备驱动程序，应该符合字符设备的格式。

```
1 module_init(tun_init);
2 module_exit(tun_cleanup);
3 MODULE_DESCRIPTION(DRV_DESCRIPTION);
4 MODULE_AUTHOR(DRV_COPYRIGHT);
5 MODULE_LICENSE("GPL");
6 MODULE_ALIAS_MISCDEV(TUN_MINOR);
7 MODULE_ALIAS("devname:net/tun");
8
9 static int __init tun_init(void)
10 {
11     .....
12     ret = rtnl_link_register(&tun_link_ops);
13     .....
14     ret = misc_register(&tun_miscdev);
15     .....
16     ret = register_netdevice_notifier(&tun_notifier_block);
17     .....
18 }
```

这里面注册了一个 tun\_miscdev 字符设备，从它的定义可以看出，这就是"/dev/net/tun"字符设备。

```
1 static struct miscdevice tun_miscdev = {
2     .minor = TUN_MINOR,
3     .name = "tun",
4     .nodename = "net/tun",
5     .fops = &tun_fops,
6 };
7
8 static const struct file_operations tun_fops = {
9     .owner = THIS_MODULE,
10    .llseek = no_llseek,
11    .read_iter = tun_chr_read_iter,
12    .write_iter = tun_chr_write_iter,
13    .poll = tun_chr_poll,
14    .unlocked_ioctl = tun_chr_ioctl,
15    .open = tun_chr_open,
16    .release = tun_chr_close,
17    .fsync = tun_chr_fsync,
18 };
```


qemu 的 tap\_open 函数会打开这个字符设备 PATH\_NET\_TUN。打开字符设备的过程我们不再重复。我就说一下，到了驱动这一层，调用的是 tun\_chr\_open。

 复制代码

```
1 static int tun_chr_open(struct inode *inode, struct file * file)
2 {
3     struct tun_file *tfile;
4     tfile = (struct tun_file *)sk_alloc(net, AF_UNSPEC, GFP_KERNEL,
5                                         &tun_proto, 0);
6     RCU_INIT_POINTER(tfile->tun, NULL);
7     tfile->flags = 0;
8     tfile->ifindex = 0;
9
10    init_waitqueue_head(&tfile->wq.wait);
11    RCU_INIT_POINTER(tfile->socket.wq, &tfile->wq);
12
13    tfile->socket.file = file;
14    tfile->socket.ops = &tun_socket_ops;
15
16    sock_init_data(&tfile->socket, &tfile->sk);
17
18    tfile->sk.sk_write_space = tun_sock_write_space;
19    tfile->sk.sk_sndbuf = INT_MAX;
20
21    file->private_data = tfile;
22    INIT_LIST_HEAD(&tfile->next);
23
24    sock_set_flag(&tfile->sk, SOCK_ZEROCOPY);
25
26    return 0;
27 }
```

在 tun\_chr\_open 的参数里面，有一个 struct file，这是代表什么文件呢？它代表的就是打开的字符设备文件"/dev/net/tun"，因而往这个字符设备文件中写数据，就会通过这个 struct file 写入。这个 struct file 里面的 file\_operations，按照字符设备打开的规则，指向的就是 tun\_fops。

另外，我们还需要在 tun\_chr\_open 创建了一个结构 struct tun\_file，并且将 struct file 的 private\_data 指向它。

 复制代码

```
1 /* A tun_file connects an open character device to a tuntap netdevice. It
```

```

2  * also contains all socket related structures
3  * to serve as one transmit queue for tuntap device.
4  */
5  struct tun_file {
6      struct sock sk;
7      struct socket socket;
8      struct socket_wq wq;
9      struct tun_struct __rcu *tun;
10     struct fasync_struct *fasync;
11     /* only used for fasnyc */
12     unsigned int flags;
13     union {
14         u16 queue_index;
15         unsigned int ifindex;
16     };
17     struct list_head next;
18     struct tun_struct *detached;
19     struct skb_array tx_array;
20 };
21
22 struct tun_struct {
23     struct tun_file __rcu  *tfiles[MAX_TAP_QUEUES];
24     unsigned int           numqueues;
25     unsigned int           flags;
26     kuid_t                 owner;
27     kgid_t                 group;
28
29     struct net_device      *dev;
30     netdev_features_t      set_features;
31     int                    align;
32     int                    vnet_hdr_sz;
33     int                    sndbuf;
34     struct tap_filter      txflt;
35     struct sock_fprog      fprog;
36     /* protected by rtnl lock */
37     bool                   filter_attached;
38     spinlock_t lock;
39     struct hlist_head flows[TUN_NUM_FLOW_ENTRIES];
40     struct timer_list flow_gc_timer;
41     unsigned long ageing_time;
42     unsigned int numdisabled;
43     struct list_head disabled;
44     void *security;
45     u32 flow_count;
46     u32 rx_batched;
47     struct tun_pcpu_stats __percpu *pcpu_stats;
48 };
49
50 static const struct proto_ops tun_socket_ops = {
51     .peek_len = tun_peek_len,
52     .sendmsg = tun_sendmsg,
53     .recvmsg = tun_recvmsg,

```

在 struct tun\_file 中，有一个成员 struct tun\_struct，它里面有一个 struct net\_device，这个用来表示宿主机上的 tuntap 网络设备。在 struct tun\_file 中，还有 struct socket 和 struct sock，因为要用到内核的网络协议栈，所以需要这两个结构，这在[网络协议](#)那一节已经分析过了。

所以，按照 struct tun\_file 的注释说的，这是一个很重要的数据结构。"/dev/net/tun"对应的 struct file 的 private\_data 指向它，因而可以接收 qemu 发过来的数据。除此之外，它还可以通过 struct sock 来操作内核协议栈，然后将网络包从宿主机上的 tuntap 网络设备发出去，宿主机上的 tuntap 网络设备对应的 struct net\_device 也归它管。

在 qemu 的 tap\_open 函数中，打开这个字符设备文件之后，接下来要做的事情是，通过 ioctl 来设置宿主机的网卡 TUNSETIFF。

接下来，ioctl 到了内核里面，会调用 tun\_chr\_ioctl。

 复制代码

```

1 static long __tun_chr_ioctl(struct file *file, unsigned int cmd,
2                             unsigned long arg, int ifreq_len)
3 {
4     struct tun_file *tfile = file->private_data;
5     struct tun_struct *tun;
6     void __user* argp = (void __user*)arg;
7     struct ifreq ifr;
8     kuid_t owner;
9     kgid_t group;
10    int sndbuf;
11    int vnet_hdr_sz;
12    unsigned int ifindex;
13    int le;
14    int ret;
15
16    if (cmd == TUNSETIFF || cmd == TUNSETQUEUE || _IOC_TYPE(cmd) == SOCK_IOC_TYPE) {
17        if (copy_from_user(&ifr, argp, ifreq_len))
18            return -EFAULT;
19    }
20    .....
21    tun = __tun_get(tfile);
22    if (cmd == TUNSETIFF) {
23        ifr.ifr_name[IFNAMSIZ-1] = '\0';


```

```

24         ret = tun_set_iff(sock_net(&tfile->sk), file, &ifr);
25     .....
26         if (copy_to_user(argp, &ifr, ifreq_len))
27             ret = -EFAULT;
28     }
29     .....
30 }

```

在 `_tun_chr_ioctl` 中，我们首先通过 `copy_from_user` 把配置从用户态拷贝到内核态，调用 `tun_set_iff` 设置 tuntap 网络设备，然后调用 `copy_to_user` 将配置结果返回。

 复制代码

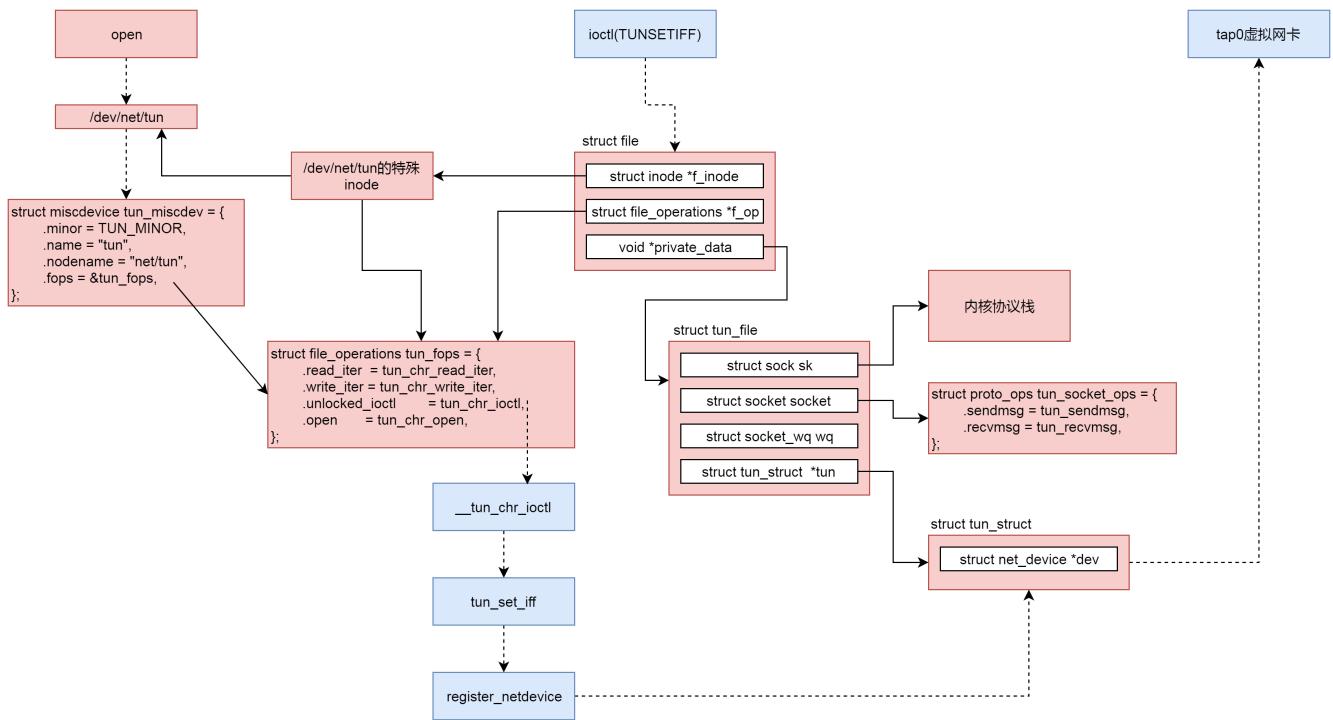
```

1 static int tun_set_iff(struct net *net, struct file *file, struct ifreq *ifr)
2 {
3     struct tun_struct *tun;
4     struct tun_file *tfile = file->private_data;
5     struct net_device *dev;
6     .....
7     char *name;
8     unsigned long flags = 0;
9     int queues = ifr->ifr_flags & IFF_MULTI_QUEUE ?
10                 MAX_TAP_QUEUES : 1;
11
12     if (ifr->ifr_flags & IFF_TUN) {
13         /* TUN device */
14         flags |= IFF_TUN;
15         name = "tun%d";
16     } else if (ifr->ifr_flags & IFF_TAP) {
17         /* TAP device */
18         flags |= IFF_TAP;
19         name = "tap%d";
20     } else
21         return -EINVAL;
22
23     if (*ifr->ifr_name)
24         name = ifr->ifr_name;
25
26     dev = alloc_netdev_mqs(sizeof(struct tun_struct), name,
27                           NET_NAME_UNKNOWN, tun_setup, queues,
28                           queues);
29
30     err = dev_get_valid_name(net, dev, name);
31     dev_net_set(dev, net);
32     dev->rtnl_link_ops = &tun_link_ops;
33     dev->ifindex = tfile->ifindex;
34     dev->sysfs_groups[0] = &tun_attr_group;
35

```

```
36     tun = netdev_priv(dev);
37     tun->dev = dev;
38     tun->flags = flags;
39     tun->txflt.count = 0;
40     tun->vnet_hdr_sz = sizeof(struct virtio_net_hdr);
41
42     tun->align = NET_SKB_PAD;
43     tun->filter_attached = false;
44     tun->sndbuf = tfile->socket.sk->sk_sndbuf;
45     tun->rx_batched = 0;
46
47     tun_net_init(dev);
48     tun_flow_init(tun);
49
50     err = tun_attach(tun, file, false);
51     err = register_netdevice(tun->dev);
52
53     netif_carrier_on(tun->dev);
54
55     if (netif_running(tun->dev))
56         netif_tx_wake_all_queues(tun->dev);
57
58     strcpy(ifr->ifr_name, tun->dev->name);
59     return 0;
60 }
```

tun\_set\_iff 创建了 struct tun\_struct 和 struct net\_device , 并且将这个 tuntap 网络设备通过 register\_netdevice 注册到内核中。这样, 我们就能在宿主机上通过 ip addr 看到这个网卡了。



至此宿主机上的内核的数据结构也完成了。

## 关联前端设备驱动和后端设备驱动

下面，我们来解析在客户机中发送一个网络包的时候，会发生哪些事情。

虚拟机里面的进程发送一个网络包，通过文件系统和 Socket 调用网络协议栈，到达网络设备层。只不过这个不是普通的网络设备，而是 virtio\_net 的驱动。

virtio\_net 的驱动程序代码在 Linux 操作系统的源代码里面，文件名为 drivers/net/virtio\_net.c。

复制代码

```

1 static __init int virtio_net_driver_init(void)
2 {
3     ret = register_virtio_driver(&virtio_net_driver);
4     .....
5 }
6 module_init(virtio_net_driver_init);
7 module_exit(virtio_net_driver_exit);
8
9 MODULE_DEVICE_TABLE(virtio, id_table);
10 MODULE_DESCRIPTION("Virtio network driver");
11 MODULE_LICENSE("GPL");
12
13 static struct virtio_driver virtio_net_driver = {

```




```

14     .driver.name = KBUILD_MODNAME,
15     .driver.owner = THIS_MODULE,
16     .id_table = id_table,
17     .validate = virtnet_validate,
18     .probe = virtnet_probe,
19     .remove = virtnet_remove,
20     .config_changed = virtnet_config_changed,
21     .....
22 };

```

在 virtio\_net 的驱动程序的初始化代码中，我们需要注册一个驱动函数 virtio\_net\_driver。

当一个设备驱动作为一个内核模块被初始化的时候，probe 函数会被调用，因而我们来看一下 virtnet\_probe。

 复制代码

```

1 static int virtnet_probe(struct virtio_device *vdev)
2 {
3     int i, err;
4     struct net_device *dev;
5     struct virtnet_info *vi;
6     u16 max_queue_pairs;
7     int mtu;
8
9     /* Allocate ourselves a network device with room for our info */
10    dev = alloc_etherdev_mq(sizeof(struct virtnet_info), max_queue_pairs);
11
12    /* Set up network device as normal. */
13    dev->priv_flags |= IFF_UNICAST_FLT | IFF_LIVE_ADDR_CHANGE;
14    dev->netdev_ops = &virtnet_netdev;
15    dev->features = NETIF_F_HIGHDMA;
16
17    dev->ethtool_ops = &virtnet_ethtool_ops;
18    SET_NETDEV_DEV(dev, &vdev->dev);
19    .....
20    /* MTU range: 68 - 65535 */
21    dev->min_mtu = MIN_MTU;
22    dev->max_mtu = MAX_MTU;
23
24    /* Set up our device-specific information */
25    vi = netdev_priv(dev);
26    vi->dev = dev;
27    vi->vdev = vdev;
28    vdev->priv = vi;

```


```

29     vi->stats = alloc_percpu(struct virtnet_stats);
30     INIT_WORK(&vi->config_work, virtnet_config_changed_work);
31     .....
32     vi->max_queue_pairs = max_queue_pairs;
33
34     /* Allocate/initialize the rx/tx queues, and invoke find_vqs */
35     err = init_vqs(vi);
36     netif_set_real_num_tx_queues(dev, vi->curr_queue_pairs);
37     netif_set_real_num_rx_queues(dev, vi->curr_queue_pairs);
38
39     virtnet_init_settings(dev);
40
41     err = register_netdev(dev);
42     virtio_device_ready(vdev);
43     virtnet_set_queues(vi, vi->curr_queue_pairs);
44     .....
45 }

```

在 `virtnet_probe` 中，会创建 `struct net_device`，并且通过 `register_netdev` 注册这个网络设备，这样在客户机里面，就能看到这个网卡了。

在 `virtnet_probe` 中，还有一件重要的事情就是，`init_vqs` 会初始化发送和接收的 `virtqueue`。

 复制代码

```

1 static int init_vqs(struct virtnet_info *vi)
2 {
3     int ret;
4
5     /* Allocate send & receive queues */
6     ret = virtnet_alloc_queues(vi);
7     ret = virtnet_find_vqs(vi);
8     .....
9     get_online_cpus();
10    virtnet_set_affinity(vi);
11    put_online_cpus();
12
13    return 0;
14 }
15
16 static int virtnet_alloc_queues(struct virtnet_info *vi)
17 {
18     int i;
19
20     vi->sq = kzalloc(sizeof(*vi->sq) * vi->max_queue_pairs, GFP_KERNEL);

```


```

21     vi->rq = kzalloc(sizeof(*vi->rq) * vi->max_queue_pairs, GFP_KERNEL);
22
23     INIT_DELAYED_WORK(&vi->refill, refill_work);
24     for (i = 0; i < vi->max_queue_pairs; i++) {
25         vi->rq[i].pages = NULL;
26         netif_napi_add(vi->dev, &vi->rq[i].napi, virtnet_poll,
27                         napi_weight);
28         netif_tx_napi_add(vi->dev, &vi->sq[i].napi, virtnet_poll_tx,
29                           napi_tx ? napi_weight : 0);
30
31         sg_init_table(vi->rq[i].sg, ARRAY_SIZE(vi->rq[i].sg));
32         ewma_pkt_len_init(&vi->rq[i].mrg_avg_pkt_len);
33         sg_init_table(vi->sq[i].sg, ARRAY_SIZE(vi->sq[i].sg));
34     }
35
36     return 0;
37 }

```

按照上一节的 virtio 原理，virtqueue 是一个介于客户机前端和 qemu 后端的一个结构，用于在这两端之间传递数据，对于网络设备来讲有发送和接收两个方向的队列。这里建立的 struct virtqueue 是客户机前端对于队列的管理的数据结构。

队列的实体需要通过函数 virtnet\_find\_vqs 查找或者生成，这里还会指定接收队列的 callback 函数为 skb\_recv\_done，发送队列的 callback 函数为 skb\_xmit\_done。那当 buffer 使用发生变化的时候，我们可以调用这个 callback 函数进行通知。

 复制代码

```

1 static int virtnet_find_vqs(struct virtnet_info *vi)
2 {
3     vq_callback_t **callbacks;
4     struct virtqueue **vqs;
5     int ret = -ENOMEM;
6     int i, total_vqs;
7     const char **names;
8
9     /* Allocate space for find_vqs parameters */
10    vqs = kzalloc(total_vqs * sizeof(*vqs), GFP_KERNEL);
11    callbacks = kmalloc(total_vqs * sizeof(*callbacks), GFP_KERNEL);
12    names = kmalloc(total_vqs * sizeof(*names), GFP_KERNEL);
13
14    /* Allocate/initialize parameters for send/receive virtqueues */
15    for (i = 0; i < vi->max_queue_pairs; i++) {
16        callbacks[rxq2vq(i)] = skb_recv_done;
17        callbacks[txq2vq(i)] = skb_xmit_done;
18        names[rxq2vq(i)] = vi->rq[i].name;

```

```

19         names[txq2vq(i)] = vi->sq[i].name;
20     }
21
22     ret = vi->vdev->config->find_vqs(vi->vdev, total_vqs, vqs, callbacks, names, ct:
23     .....
24     for (i = 0; i < vi->max_queue_pairs; i++) {
25         vi->rq[i].vq = vqs[rxq2vq(i)];
26         vi->rq[i].min_buf_len = mergeable_min_buf_len(vi, vi->rq[i].vq);
27         vi->sq[i].vq = vqs[txq2vq(i)];
28     }
29     .....
30 }

```



这里的 `find_vqs` 是在 `struct virtnet_info` 里的 `struct virtio_device` 里的 `struct virtio_config_ops *config` 里面定义的。

根据 `virtio_config_ops` 的定义，`find_vqs` 会调用 `vp_modern_find_vqs`，到这一步和块设备是一样的了。

在 `vp_modern_find_vqs` 中，`vp_find_vqs` 会调用 `vp_find_vqs_intx`。在 `vp_find_vqs_intx` 中，通过 `request_irq` 注册一个中断处理函数 `vp_interrupt`。当设备向队列中写入信息时，会产生一个中断，也就是 `vq` 中断。中断处理函数需要调用相应的队列的回调函数，然后根据队列的数目，依次调用 `vp_setup_vq` 完成 `virtqueue`、`vring` 的分配和初始化。

同样，这些数据结构会和 `virtio` 后端的 `VirtIODevice`、`VirtQueue`、`vring` 对应起来，都应该指向刚才创建的那一段内存。


客户机同样会通过调用专门给外部设备发送指令的函数 `iowrite` 告诉外部的 `pci` 设备，这些共享内存的地址。

至此前端设备驱动和后端设备驱动之间的两个收发队列就关联好了，这两个队列的格式和块设备是一样的。

## 发送网络包过程

接下来，我们来看当真的发送一个网络包的时候，会发生什么。

当网络包经过客户机的协议栈到达 virtio\_net 驱动的时候，按照 net\_device\_ops 的定义，start\_xmit 会被调用。

 复制代码

```
1 static const struct net_device_ops virtnet_netdev = {
2     .ndo_open          = virtnet_open,
3     .ndo_stop          = virtnet_close,
4     .ndo_start_xmit     = start_xmit,
5     .ndo_validate_addr = eth_validate_addr,
6     .ndo_set_mac_address = virtnet_set_mac_address,
7     .ndo_set_rx_mode    = virtnet_set_rx_mode,
8     .ndo_get_stats64    = virtnet_stats,
9     .ndo_vlan_rx_add_vid = virtnet_vlan_rx_add_vid,
10    .ndo_vlan_rx_kill_vid = virtnet_vlan_rx_kill_vid,
11    .ndo_xdp             = virtnet_xdp,
12    .ndo_features_check  = passthru_features_check,
13 };
```


接下来的调用链为：start\_xmit->xmit\_skb-> virtqueue\_add\_outbuf->virtqueue\_add，将网络包放入队列中，并调用 virtqueue\_notify 通知接收方。

 复制代码

```
1 static netdev_tx_t start_xmit(struct sk_buff *skb, struct net_device *dev)
2 {
3     struct virtnet_info *vi = netdev_priv(dev);
4     int qnum = skb_get_queue_mapping(skb);
5     struct send_queue *sq = &vi->sq[qnum];
6     int err;
7     struct netdev_queue *txq = netdev_get_tx_queue(dev, qnum);
8     bool kick = !skb->xmit_more;
9     bool use_napi = sq->napi.weight;
10    .....
11    /* Try to transmit */
12    err = xmit_skb(sq, skb);
13    .....
14    if (kick || netif_xmit_stopped(txq))
15        virtqueue_kick(sq->vq);
16    return NETDEV_TX_OK;
17 }
18
19 bool virtqueue_kick(struct virtqueue *vq)
20 {
21     if (virtqueue_kick_prepare(vq))
22         return virtqueue_notify(vq);
23     return true;
```

写入一个 I/O 会使得 qemu 触发 VM exit，这个逻辑我们在解析 CPU 的时候看到过。

接下来，我们那会调用 VirtQueue 的 handle\_output 函数。前面我们已经设置过这个函数了，其实就是 virtio\_net\_handle\_tx\_bh。

 复制代码

```
1 static void virtio_net_handle_tx_bh(VirtIODevice *vdev, VirtQueue *vq)
2 {
3     VirtIONet *n = VIRTIO_NET(vdev);
4     VirtIONetQueue *q = &n->vqs[vq2q(virtio_get_queue_index(vq))];
5
6     q->tx_waiting = 1;
7
8     virtio_queue_set_notification(vq, 0);
9     qemu_bh_schedule(q->tx_bh);
10 }
```

virtio\_net\_handle\_tx\_bh 调用了 qemu\_bh\_schedule，而在 virtio\_net\_add\_queue 中调用 qemu\_bh\_new，并把函数设置为 virtio\_net\_tx\_bh。

virtio\_net\_tx\_bh 函数调用发送函数 virtio\_net\_flush\_tx。

 复制代码

```
1 static int32_t virtio_net_flush_tx(VirtIONetQueue *q)
2 {
3     VirtIONet *n = q->n;
4     VirtIODevice *vdev = VIRTIO_DEVICE(n);
5     VirtQueueElement *elem;
6     int32_t num_packets = 0;
7     int queue_index = vq2q(virtio_get_queue_index(q->tx_vq));
8
9     for (;;) {
10         ssize_t ret;
11         unsigned int out_num;
12         struct iovec sg[VIRTQUEUE_MAX_SIZE], sg2[VIRTQUEUE_MAX_SIZE + 1], *out_sg;
13         struct virtio_net_hdr_mrg_rxbuf mhdr;
14
15         elem = virtqueue_pop(q->tx_vq, sizeof(VirtQueueElement));
```

```


16         out_num = elem->out_num;
17         out_sg = elem->out_sg;
18     .....
19         ret = qemu_sendv_packet_async(qemu_get_subqueue(n->nic, queue_index), out_sg, out_num);
20     }
21     .....
22     return num_packets;
23 }

```

virtio\_net\_flush\_tx 会调用 virtqueue\_pop。这里面，我们能看到对于 vring 的操作，也即从这里面将客户机里面写入的数据读取出来。

然后，我们调用 qemu\_sendv\_packet\_async 发送网络包。接下来的调用链为：  
 qemu\_sendv\_packet\_async->qemu\_net\_queue\_send\_iov->qemu\_net\_queue\_flush->qemu\_net\_queue\_deliver。

在 qemu\_net\_queue\_deliver 中，我们会调用 NetQueue 的 deliver 函数。前面 qemu\_new\_net\_queue 会把 deliver 函数设置为 qemu\_deliver\_packet\_iov。它会调用 nc->info->receive\_iov。

 复制代码


```

1 static NetClientInfo net_tap_info = {
2     .type = NET_CLIENT_DRIVER_TAP,
3     .size = sizeof(TAPState),
4     .receive = tap_receive,
5     .receive_raw = tap_receive_raw,
6     .receive_iov = tap_receive_iov,
7     .poll = tap_poll,
8     .cleanup = tap_cleanup,
9     .has_ufo = tap_has_ufo,
10    .has_vnet_hdr = tap_has_vnet_hdr,
11    .has_vnet_hdr_len = tap_has_vnet_hdr_len,
12    .using_vnet_hdr = tap_using_vnet_hdr,
13    .set_offload = tap_set_offload,
14    .set_vnet_hdr_len = tap_set_vnet_hdr_len,
15    .set_vnet_le = tap_set_vnet_le,
16    .set_vnet_be = tap_set_vnet_be,
17 };

```

根据 `net_tap_info` 的定义调用的是 `tap_receive_iov`。他会调用 `tap_write_packet->writev` 写入这个字符设备。

在内核的字符设备驱动中，`tun_chr_write_iter` 会被调用。

 复制代码

```
1 static ssize_t tun_chr_write_iter(struct kiocb *iocb, struct iov_iter *from)
2 {
3     struct file *file = iocb->ki_filp;
4     struct tun_struct *tun = tun_get(file);
5     struct tun_file *tfile = file->private_data;
6     ssize_t result;
7
8     result = tun_get_user(tun, tfile, NULL, from,
9                          file->f_flags & O_NONBLOCK, false);
10
11     tun_put(tun);
12     return result;
13 }
```

当我们使用 `writev()` 系统调用向 `tun/tap` 设备的字符设备文件写入数据时，`tun_chr_write` 函数将被调用。它会使用 `tun_get_user`，从用户区接收数据，将数据存入 `skb` 中，然后调用关键的函数 `netif_rx_ni(skb)`，将 `skb` 送给 `tcp/ip` 协议栈处理，最终完成虚拟网卡的数据接收。

至此，从虚拟机内部到宿主机的网络传输过程才算结束。

## 总结时刻

最后，我们把网络虚拟化场景下网络包的发送过程总结一下。

在虚拟机里面的用户态，应用程序通过 `write` 系统调用写入 `socket`。

写入的内容经过 `VFS` 层，内核协议栈，到达虚拟机里面的内核的网络设备驱动，也即 `virtio_net`。

`virtio_net` 网络设备有一个操作结构 `struct net_device_ops`，里面定义了发送一个网络包调用的函数为 `start_xmit`。



在 virtio\_net 的前端驱动和 qemu 中的后端驱动之间，有两个队列 virtqueue，一个用于发送，一个用于接收。然后，我们需要在 start\_xmit 中调用 virtqueue\_add，将网络包放入发送队列，然后调用 virtqueue\_notify 通知 qemu。

qemu 本来处于 KVM\_RUN 的状态，收到通知后，通过 VM exit 指令退出客户机模式，进入宿主机模式。发送网络包的时候，virtio\_net\_handle\_tx\_bh 函数会被调用。

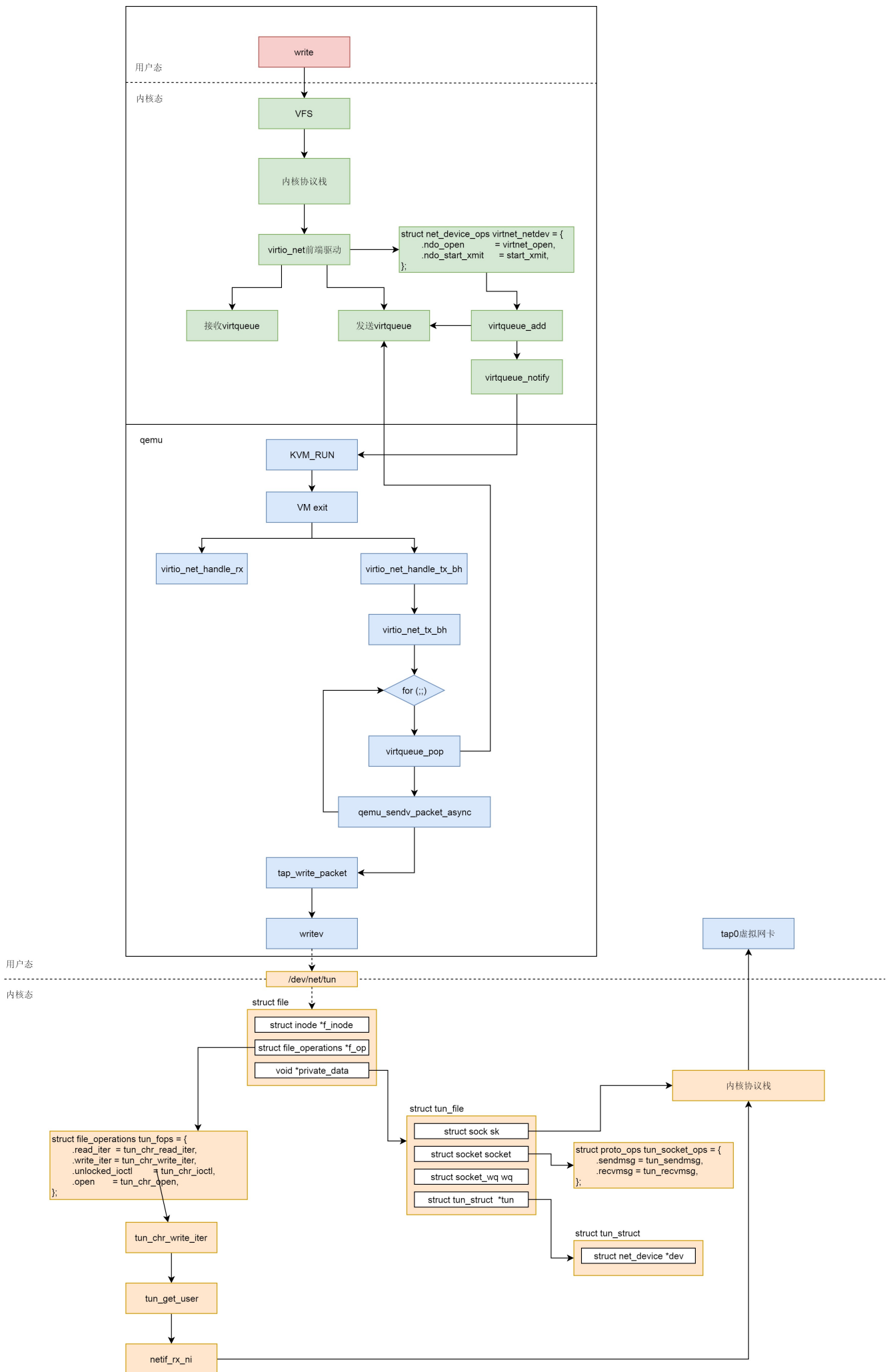
接下来是一个 for 循环，我们需要在循环中调用 virtqueue\_pop，从传输队列中获取要发送的数据，然后调用 qemu\_sendv\_packet\_async 进行发送。

qemu 会调用 writev 向字符设备文件写入，进入宿主机的内核。

在宿主机内核中字符设备文件的 file\_operations 里面的 write\_iter 会被调用，也即会调用 tun\_chr\_write\_iter。

在 tun\_chr\_write\_iter 函数中，tun\_get\_user 将要发送的网络包从 qemu 拷贝到宿主机内核里面来，然后调用 netif\_rx\_ni 开始调用宿主机内核协议栈进行处理。

宿主机内核协议栈处理完毕之后，会发送给 tap 虚拟网卡，完成从虚拟机里面到宿主机的整个发送过程。



## 课堂练习

这一节我们解析的是发送过程，请你根据类似的思路，解析一下接收过程。

欢迎留言和我分享你的疑惑和见解，也欢迎收藏本节内容，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。

 极客时间

# 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 54 | 存储虚拟化（下）：如何建立自己保管的单独档案库？

下一篇 56 | 容器：大公司为保持创新，鼓励内部创业

## 精选留言 (3)

写留言



kkxue

2019-08-04

学了这么多年的虚拟网络，不及老师一节课的深度啊

展开 ∨





盛

2019-08-03

学习了：跟完刘老师的趣谈网络协议再跟着linux系统，发现收获又不一样；同时在跟老师的网络协议的过程中，还被迫去跟着学习刘文浩老师的计算机组成原理-否则没法理解老师的一些概念。

这大概就是老师之前说的学习方法吧：书阅读越厚、读书的过程中不断去相应的扩展、学习、提升理解，然后整理出自己的东西-书就薄了；虽然书薄了，可是笔记和自己...  
展开 ∨



安排

2019-08-02

网络包是什么样的？经过协议栈处理之前的还是之后的？这样看来虚拟机里面发送网络数据要走两次协议栈吗？因为虚拟机本身也有自己的协议栈，经过虚拟机协议栈处理的数据qemu会进行拆包重新还原出原始的数据吗？

展开 ∨

