QEMU网络虚拟化分析

2018-08-04 | 2.虚拟化， QEMU | 828 | 0
881 | 1 分钟

前言

简单记录一下。

正文

相关数据结构

初始化分析

QEMU中创建net client的流程如下：

int main(int argc, char **argv, char **envp) –> net_init_clients –> net_init_netdev –> net_client_init –> net_client_init1 –> net_client_init_fun

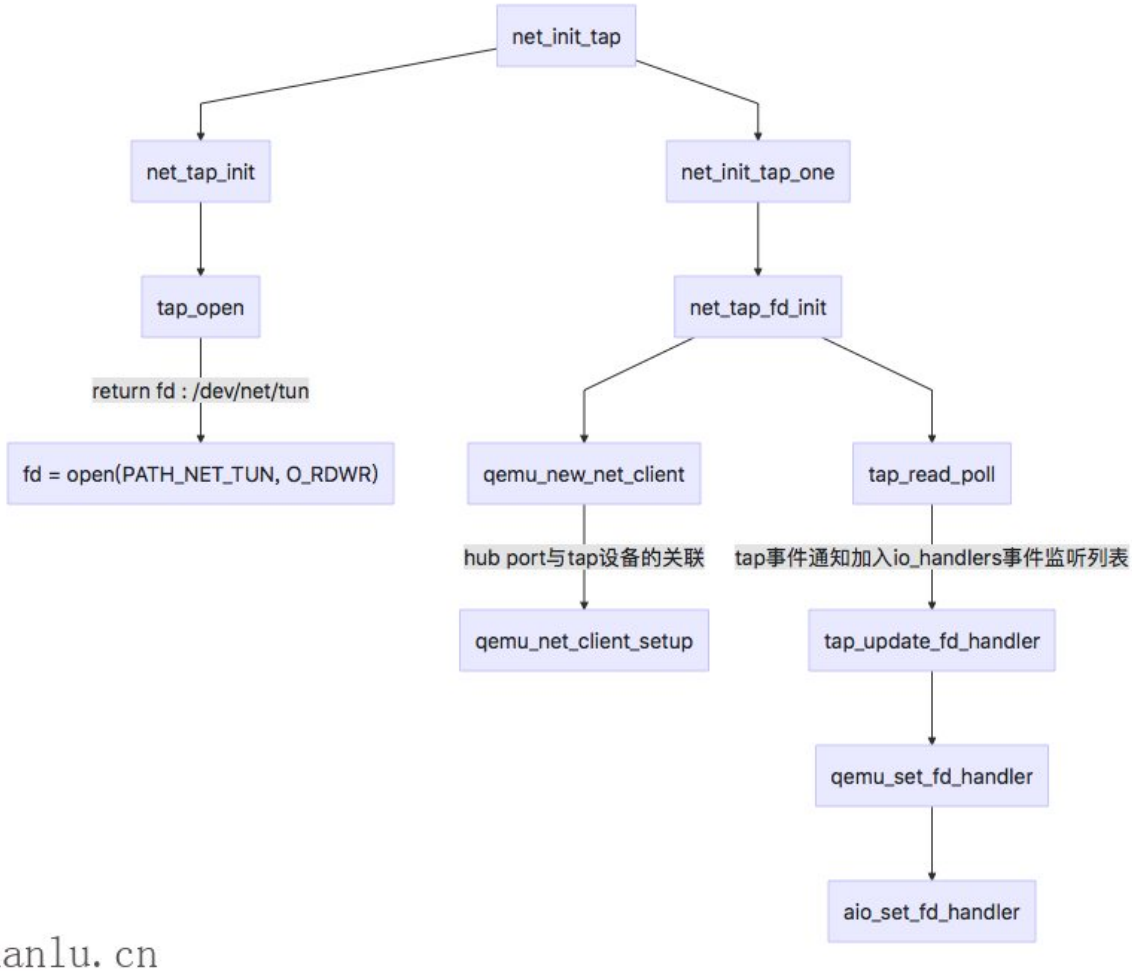
QEMU中各类net client对应的初始化函数在net.c的net_client_init_fun中。

```
static int (* const net_client_init_fun[NET_CLIENT_DRIVER__MAX])(
    const Netdev *netdev,
    const char *name,
    NetClientState *peer, Error **errp) = {
        [NET_CLIENT_DRIVER_NIC]       = net_init_nic,
#ifdef CONFIG_SLIRP
        [NET_CLIENT_DRIVER_USER]      = net_init_slirp,
#endif
        [NET_CLIENT_DRIVER_TAP]       = net_init_tap,
        [NET_CLIENT_DRIVER_SOCKET]    = net_init_socket,
#ifdef CONFIG_VDE
        [NET_CLIENT_DRIVER_VDE]       = net_init_vde,
#endif
#ifdef CONFIG_NETMAP
        [NET_CLIENT_DRIVER_NETMAP]    = net_init_netmap,
#endif
        [NET_CLIENT_DRIVER_DUMP]      = net_init_dump,
#ifdef CONFIG_NET_BRIDGE
        [NET_CLIENT_DRIVER_BRIDGE]    = net_init_bridge,
#endif
        [NET_CLIENT_DRIVER_HUBPORT]   = net_init_hubport,
#ifdef CONFIG_VHOST_NET_USED
        [NET_CLIENT_DRIVER_VHOST_USER] = net_init_vhost_user,
#endif
#ifdef CONFIG_L2TPV3
        [NET_CLIENT_DRIVER_L2TPV3]    = net_init_l2tpv3,
85%
};
```

下面主要分析nic、tap和hubport的初始化过程。

Tap初始化

初始化主要函数调用流程如下：



fd = open(PATH_NET_TUN, O_RDWR)返回文件描述符fd；

```
#define PATH_NET_TUN "/dev/net/tun"
```

tap_update_fd_handler将Tap设备的事件通知加入QEMU的io_handlers事件监听列表中，如下图，fd_read事件对应的动作为tap_send(),fd_write事件对应的动作为tap_writable()

```
static void tap_update_fd_handler(TAPState *s)
{
    qemu_set_fd_handler(s->fd,
                        s->read_poll && s->enabled ? tap_send : NULL,
                        s->write_poll && s->enabled ? tap_writable : NULL,
                        s);
}
```

hub port与tap设备的关联，Tap设备对应的hub port的NetClientState的peer指向了TAPState的NetClient，而TAPState的NetClient的peer指向了hub port的NetClientState。

```
static void qemu_net_client_setup(NetClientState *nc,
                                  NetClientInfo *info,
                                  NetClientState *peer,
                                  const char *model,
                                  const char *name,
                                  NetClientDestructor *destructor)
{
    nc->info = info;
    nc->model = g_strdup(model);
    if (name) {
        nc->name = g_strdup(name);
    } else {
        nc->name = assign_name(nc, model);
    }

    if (peer) {
        assert(!peer->peer);
        nc->peer = peer;
        peer->peer = nc;
    }
    QTAILQ_INSERT_TAIL(&net_clients, nc, next);

    nc->incoming_queue = qemu_new_net_queue(qemu_deliver_packet_iov, nc);
    nc->destructor = destructor;
    QTAILQ_INIT(&nc->filters);
} « end qemu_net_client_setup »
```


hubport初始化

net_init_hubport –> net_hub_add_port –> net_hub_port_new –> qemu_new_net_client –> qemu_net_client_setup

```
int net_init_hubport(const Netdev *netdev, const char *name,
                     NetClientState *peer, Error **errp)
{
    const NetdevHubPortOptions *hubport;

    assert(netdev->type == NET_CLIENT_DRIVER_HUBPORT);
    assert(!peer);
    hubport = &netdev->u.hubport;

    net_hub_add_port(hubport->hubid, name);
    return 0;
}
```

```c
/**
 * Create a port on a given hub
 * @name: Net client name or NULL for default name.
 *
 * If there is no existing hub with the given id then a new hub is created.
 */
NetClientState *net_hub_add_port(int hub_id, const char *name)
{
    NetHub *hub;
    NetHubPort *port;

    QLIST_FOREACH(hub, &hubs, next) {
        if (hub->id == hub_id) {
            break;
        }
    }

    if (!hub) {
        hub = net_hub_new(hub_id);
    }

    port = net_hub_port_new(hub, name);
    return &port->nc;
}
```



```c
static NetHubPort *net_hub_port_new(NetHub *hub, const char *name)
{
    NetClientState *nc;
    NetHubPort *port;
    int id = hub->num_ports++;
    char default_name[128];

    if (!name) {
        snprintf(default_name, sizeof(default_name),
                 "hub%dport%d", hub->id, id);
        name = default_name;
    }

    nc = qemu_new_net_client(&net_hub_port_info, NULL, "hub", name);
    port = DO_UPCAST(NetHubPort, nc, nc);
    port->id = id;
    port->hub = hub;

    QLIST_INSERT_HEAD(&hub->ports, port, next);

    return port;
} « end net_hub_port_new »
```


nic初始化

nic设备与hub port的关联：nd->netdev = peer

```
static int net_init_nic(const Netdev *netdev, const char *name,
                        NetClientState *peer, Error **errp)
{
    int idx;
    NICInfo *nd;
    const NetLegacyNicOptions *nic;

    assert(netdev->type == NET_CLIENT_DRIVER_NIC);
    nic = &netdev->u.nic;

    idx = nic_get_free_idx();
    if (idx == -1 || nb_nics >= MAX_NICS) {
        error_setg(errp, "too many NICs");
        return -1;
    }

    nd = &nd_table[idx];

    memset(nd, 0, sizeof(*nd));

    if (nic->has_netdev) {
        nd->netdev = qemu_find_netdev(nic->netdev);
        if (!nd->netdev) {
            error_setg(errp, "netdev '%s' not found", nic->netdev);
            return -1;
        }
    } else {
        assert(peer);
        nd->netdev = peer;
    }
```


e1000_class_init

```
static void e1000_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);
    PCIDeviceClass *k = PCI_DEVICE_CLASS(klass);
    E1000BaseClass *e = E1000_DEVICE_CLASS(klass);
    const E1000Info *info = data;

    k->realize = pci_e1000_realize;
    k->exit = pci_e1000_uninit;
    k->romfile = "efi-e1000.rom";
    k->vendor_id = PCI_VENDOR_ID_INTEL;
    k->device_id = info->device_id;
    k->revision = info->revision;
    e->phy_id2 = info->phy_id2;
    k->class_id = PCI_CLASS_NETWORK_ETHERNET;
    set_bit(DEVICE_CATEGORY_NETWORK, dc->categories);
    dc->desc = "Intel Gigabit Ethernet";
    dc->reset = qdev_e1000_reset;
    dc->vmsd = &vmstate_e1000;
    dc->props = e1000_properties;
} « end e1000_class_init »
```


pci_e1000_realize

```
static void pci_e1000_realize(PCIDevice *pci_dev, Error **errp)
{
    DeviceState *dev = DEVICE(pci_dev);
    E1000State *d = E1000(pci_dev);
    uint8_t *pci_conf;
    uint8_t *macaddr;

    pci_dev->config_write = e1000_write_config;

    pci_conf = pci_dev->config;

    /* TODO: RST# value should be 0, PCI spec 6.2.4 */
    pci_conf[PCI_CACHE_LINE_SIZE] = 0x10;

    pci_conf[PCI_INTERRUPT_PIN] = 1; /* interrupt pin A */

    e1000_mmio_setup(d);

    pci_register_bar(pci_dev, 0, PCI_BASE_ADDRESS_SPACE_MEMORY, &d->mmio);

    pci_register_bar(pci_dev, 1, PCI_BASE_ADDRESS_SPACE_IO, &d->io);

    qemu_macaddr_default_if_unset(&d->conf.macaddr);
    macaddr = d->conf.macaddr.a;

    e1000x_core_prepare_eeprom(d->eeprom_data,
                               e1000_eeprom_template,
                               sizeof(e1000_eeprom_template),
                               PCI_DEVICE_GET_CLASS(pci_dev)->device_id,
                               macaddr);

    d->nic = qemu_new_nic(&net_e1000_info, &d->conf,
                          object_get_typename(OBJECT(d)), dev->id, d);

    qemu_format_nic_info_str(qemu_get_queue(d->nic), macaddr);

    d->autoneg_timer = timer_new_ms(QEMU_CLOCK_VIRTUAL, e1000_autoneg_timer, d);
    d->mit_timer = timer_new_ns(QEMU_CLOCK_VIRTUAL, e1000_mit_timer, d);
} « end pci_e1000_realize »
```


qemu_new_nic

```
NICState *qemu_new_nic(NetClientInfo *info,
                       NICConf *conf,
                       const char *model,
                       const char *name,
                       void *opaque)
{
    NetClientState **peers = conf->peers.ncs;
    NICState *nic;
    int i, queues = MAX(1, conf->peers.queues);

    assert(info->type == NET_CLIENT_DRIVER_NIC);
    assert(info->size >= sizeof(NICState));

    nic = g_malloc0(info->size + sizeof(NetClientState) * queues);
    nic->ncs = (void *)nic + info->size;
    nic->conf = conf;
    nic->opaque = opaque;

    for (i = 0; i < queues; i++) {
        qemu_net_client_setup(&nic->ncs[i], info, peers[i], model, name,
                              NULL);
        nic->ncs[i].queue_index = i;
    }

    return nic;
} « end qemu_new_nic »
```


联系我

你可以直接在下方留言，也可以E-Mail联系我。

打赏

QEMU 网络

CentOS配置本地yum源 Virtio原理简介