

2019-07-29 刘超

## 趣谈Linux操作系统

[进入课程 >](#)

**讲述：刘超**

时长 12:38 大小 11.58M



前面几节，我们讲了 CPU 和内存的虚拟化。我们知道，完全虚拟化是很慢的，而通过内核的 KVM 技术和 EPT 技术，加速虚拟机对于物理 CPU 和内存的使用，我们称为硬件辅助虚拟化。

对于一台虚拟机而言，除了要虚拟化 CPU 和内存，存储和网络也需要虚拟化，存储和网络都属于外部设备，这些外部设备应该如何虚拟化呢？

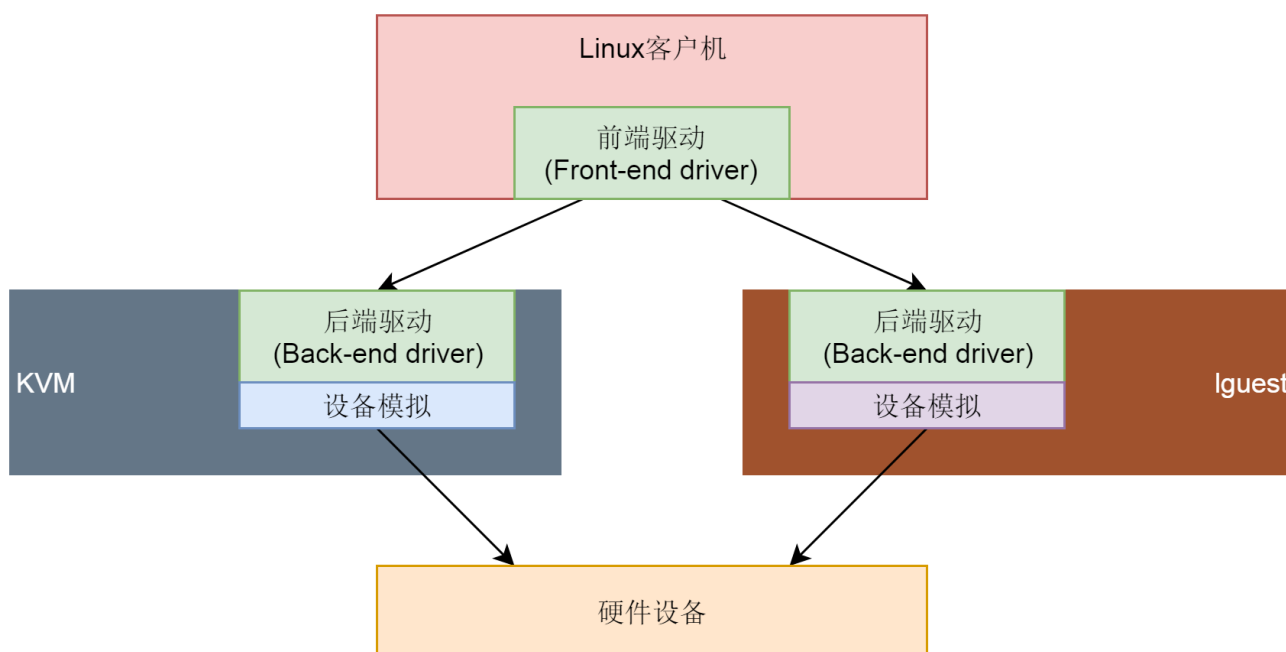
当然一种方式还是完全虚拟化。比如，有什么样的硬盘设备或者网卡设备，我们就用 qemu 模拟一个一模一样的软件的硬盘和网卡设备，这样在虚拟机里面的操作系统看来，使用这些设备和使用物理设备是一样的。当然缺点就是，qemu 模拟的设备又是一个翻译官的角色。虽然这个时候虚拟机里面的操作系统，意识不到自己是运行在虚拟机里面的，但是这种每个指令都翻译的方式，实在是太慢了。

另外一种方式就是，虚拟机里面的操作系统不是一个通用的操作系统，它知道自己是运行在虚拟机里面的，使用的硬盘设备和网络设备都是虚拟的，应该加载特殊的驱动才能运行。这些特殊的驱动往往要通过虚拟机里面和外面配合工作的模式，来加速对于物理存储和网络设备的使用。

## virtio 的基本原理

在虚拟化技术的早期，不同的虚拟化技术会针对不同硬盘设备和网络设备实现不同的驱动，虚拟机里面的操作系统也要根据不同的虚拟化技术和物理存储和网络设备，选择加载不同的驱动。但是，由于硬盘设备和网络设备太多了，驱动纷繁复杂。

后来慢慢就形成了一定的标准，这就是**virtio**，就是**虚拟化 I/O 设备**的意思。virtio 负责对于虚拟机提供统一的接口。也就是说，在虚拟机里面的操作系统加载的驱动，以后都统一加载 virtio 就可以了。



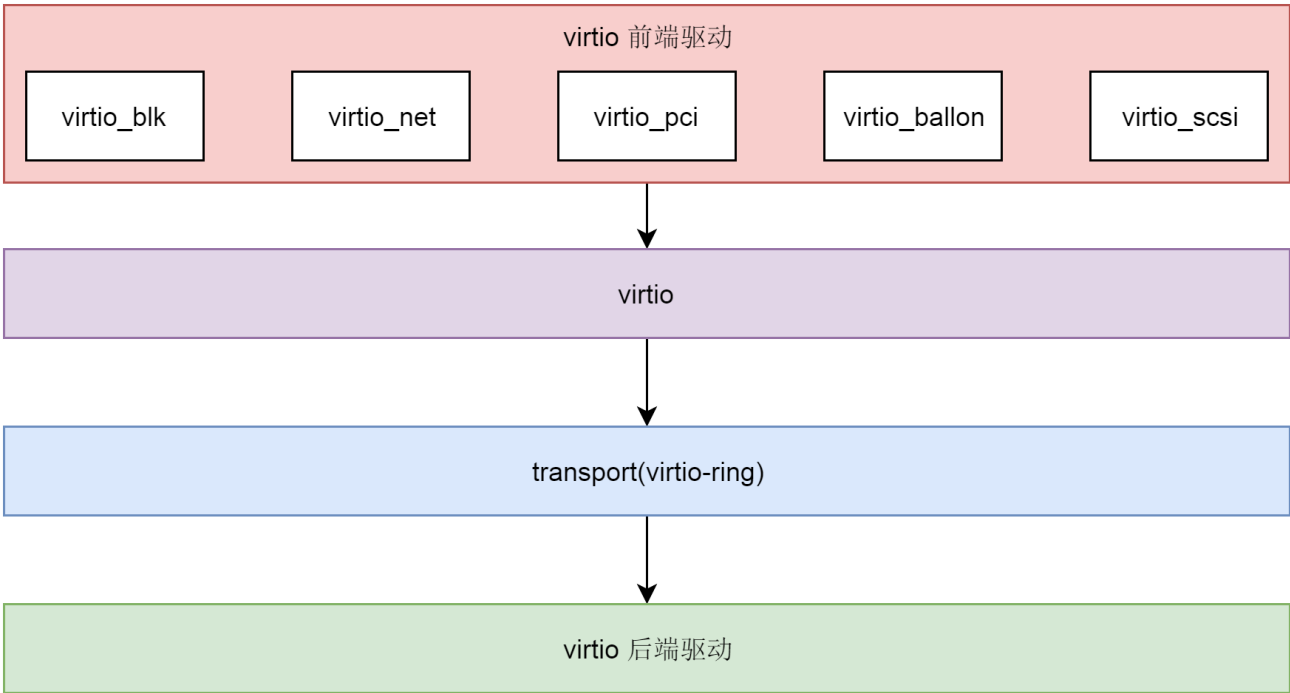
在虚拟机外，我们可以实现不同的 virtio 的后端，来适配不同的物理硬件设备。那 virtio 到底长什么样子呢？我们一起来看看。

virtio 的架构可以分为四层。

首先，在虚拟机里面的 virtio 前端，针对不同类型的设备有不同的**驱动程序**，但是接口都是统一的。例如，硬盘就是 virtio\_blk，网络就是 virtio\_net。

其次，在宿主机的 qemu 里面，实现 virtio 后端的逻辑，主要就是**操作硬件的设备**。例如通过写一个物理机硬盘上的文件来完成虚拟机写入硬盘的操作。再如向内核协议栈发送一个网络包完成虚拟机对于网络的操作。

在 virtio 的前端和后端之间，有一个通信层，里面包含**virtio 层**和**virtio-ring 层**。virtio 这一层实现的是虚拟队列接口，算是前后端通信的桥梁。而 virtio-ring 则是该桥梁的具体实现。



virtio 使用 virtqueue 进行前端和后端的高速通信。不同类型的设备队列数目不同。virtio-net 使用两个队列，一个用于接受，另一个用于发送；而 virtio-blk 仅使用一个队列。

如果客户机要向宿主机发送数据，宿主机会将数据的 buffer 添加到 virtqueue 中，然后通过写入寄存器通知宿主机。这样宿主机就可以从 virtqueue 中收到的 buffer 里面的数据。

了解了 virtio 的基本原理，接下来，我们以硬盘写入为例，具体看一下存储虚拟化的过程。

### 初始化阶段的存储虚拟化

和咱们在学习 CPU 的时候看到的一样，Virtio Block Device 也是一种类。它的继承关系如下：

```
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_blk_info = {
25     .name = TYPE_VIRTIO_BLK,
26     .parent = TYPE_VIRTIO_DEVICE,
27     .instance_size = sizeof(VirtIOBlock),
28     .instance_init = virtio_blk_instance_init,
29     .class_init = virtio_blk_class_init,
30 };
31
32 static void virtio_register_types(void)
33 {
34     type_register_static(&virtio_blk_info);
35 }
36
37 type_init(virtio_register_types)
```

Virtio Block Device 这种类的定义是有多层继承关系的。TYPE\_VIRTIO\_BLK 的父类是 TYPE\_VIRTIO\_DEVICE，TYPE\_VIRTIO\_DEVICE 的父类是 TYPE\_DEVICE，TYPE\_DEVICE 的父类是 TYPE\_OBJECT。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。还有 instance\_init，可以将 xxxClass 初始化为实例。

在 TYPE\_VIRTIO\_BLK 层的 class\_init 函数 virtio\_blk\_class\_init 中，定义了 DeviceClass 的 realize 函数为 virtio\_blk\_device\_realize，这一点在[CPU](#)那一节也有类似的结构。

 复制代码

```
1 static void virtio_blk_device_realize(DeviceState *dev, Error **errp)
2 {
3     VirtIODevice *vdev = VIRTIO_DEVICE(dev);
4     VirtIOBlock *s = VIRTIO_BLK(dev);
5     VirtIOBlkConf *conf = &s->conf;
6     .....
7     blkconf_blocksizes(&conf->conf);
8     virtio_blk_set_config_size(s, s->host_features);
9     virtio_init(vdev, "virtio-blk", VIRTIO_ID_BLOCK, s->config_size);
10    s->blk = conf->conf.blk;
11    s->rq = NULL;
12    s->sector_mask = (s->conf.conf.logical_block_size / BDRV_SECTOR_SIZE) - 1;
13    for (i = 0; i < conf->num_queues; i++) {
14        virtio_add_queue(vdev, conf->queue_size, virtio_blk_handle_output);
15    }
16    virtio_blk_data_plane_create(vdev, conf, &s->dataplane, &err);
17    s->change = qemu_add_vm_change_state_handler(virtio_blk_dma_restart_cb, s);
18    blk_set_dev_ops(s->blk, &virtio_block_ops, s);
19    blk_set_guest_block_size(s->blk, s->conf.conf.logical_block_size);
20    blk_iostatus_enable(s->blk);
21 }
```

在 virtio\_blk\_device\_realize 函数中，我们首先是通过 virtio\_init 初始化 VirtIODevice 结构。

 复制代码

```
1 void virtio_init(VirtIODevice *vdev, const char *name,
2                 uint16_t device_id, size_t config_size)
3 {
4     BusState *qbus = qdev_get_parent_bus(DEVICE(vdev));
5     VirtioBusClass *k = VIRTIO_BUS_GET_CLASS(qbus);
6     int i;
7     int nvectors = k->query_nvectors ? k->query_nvectors(qbus->parent) : 0;
8
9     if (nvectors) {
10         vdev->vector_queues =
11             g_malloc0(sizeof(*vdev->vector_queues) * nvectors);
12     }
13     vdev->device_id = device_id;
14     vdev->status = 0;
15     atomic_set(&vdev->isr, 0);
```


```

16     vdev->queue_sel = 0;
17     vdev->config_vector = VIRTIO_NO_VECTOR;
18     vdev->vq = g_malloc0(sizeof(VirtQueue) * VIRTIO_QUEUE_MAX);
19     vdev->vm_running = runstate_is_running();
20     vdev->broken = false;
21     for (i = 0; i < VIRTIO_QUEUE_MAX; i++) {
22         vdev->vq[i].vector = VIRTIO_NO_VECTOR;
23         vdev->vq[i].vdev = vdev;
24         vdev->vq[i].queue_index = i;
25     }
26     vdev->name = name;
27     vdev->config_len = config_size;
28     if (vdev->config_len) {
29         vdev->config = g_malloc0(config_size);
30     } else {
31         vdev->config = NULL;
32     }
33     vdev->vmstate = qemu_add_vm_change_state_handler(virtio_vmstate_change,
34                                                         vdev);
35     vdev->device_endian = virtio_default_endian();
36     vdev->use_guest_notifier_mask = true;
37 }

```

从 `virtio_init` 中可以看出，`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 `VIRTIO_QUEUE_MAX` 个。

我们回到 `virtio_blk_device_realize` 函数。接下来，根据配置的队列数目 `num_queues`，对于每个队列都调用 `virtio_add_queue` 来初始化队列。

 复制代码

```

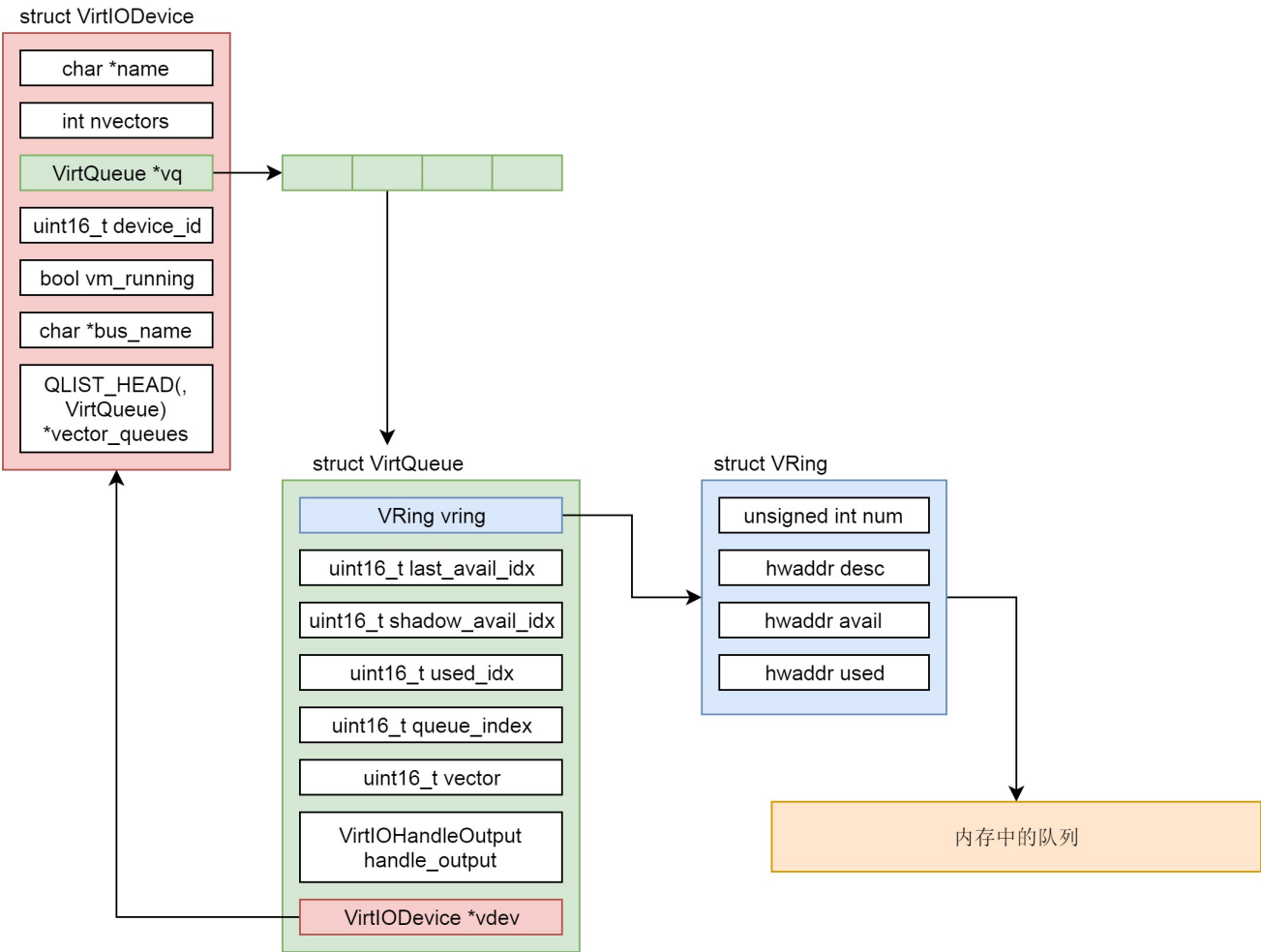
1 VirtQueue *virtio_add_queue(VirtIODevice *vdev, int queue_size,
2                             VirtIOHandleOutput handle_output)
3 {
4     int i;
5     vdev->vq[i].vring.num = queue_size;
6     vdev->vq[i].vring.num_default = queue_size;
7     vdev->vq[i].vring.align = VIRTIO_PCI_VRING_ALIGN;
8     vdev->vq[i].handle_output = handle_output;
9     vdev->vq[i].handle_aio_output = NULL;
10
11     return &vdev->vq[i];
12 }

```



在每个 VirtQueue 中，都有一个 vring，用来维护这个队列里面的数据；另外还有一个函数 virtio\_blk\_handle\_output，用于处理数据写入，这个函数我们后面会用到。


至此，VirtIODevice，VirtQueue，vring 之间的关系如下图所示。这是在 qemu 里面的对应关系，请你记好，后面我们还能看到类似的结构。



## qemu 启动过程中的存储虚拟化

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。


对于硬盘的虚拟化，qemu 的启动参数里面有关的是下面两行：

 复制代码

```
1 -drive file=/var/lib/nova/instances/1f8e6f7e-5a70-4780-89c1-464dc0e7f308/disk,if=none,id=drive-virtio-disk0
2 -device virtio-blk-pci,scsi=off,bus=pci.0,addr=0x4,drive=drive-virtio-disk0,id=virtio-disk0
```


其中，第一行指定了宿主机硬盘上的一个文件，文件的格式是 qcow2，这个格式我们这里不准备解析它，你只要明白，对于宿主机上的一个文件，可以被 qemu 模拟称为客户机上的的一块硬盘就可以了。

而第二行说明了，使用的驱动是 virtio-blk 驱动。

 复制代码


```
1 configure_blockdev(&bdo_queue, machine_class, snapshot);
```

在 qemu 启动的 main 函数里面，初始化块设备，是通过 configure\_blockdev 调用开始的。

 复制代码

```
1 static void configure_blockdev(BlockdevOptionsQueue *bdo_queue, MachineClass *machine_c:
2 {
3     .....
4     if (qemu_opts_foreach(qemu_find_opts("drive"), drive_init_func,
5                           &machine_class->block_default_type, &error_fatal)) {
6     .....
7     }
8 }
9
10 static int drive_init_func(void *opaque, QemuOpts *opts, Error **errp)
11 {
12     BlockInterfaceType *block_default_type = opaque;
13     return drive_new(opts, *block_default_type, errp) == NULL;
14 }
```

在 configure\_blockdev 中，我们能看到对于 drive 这个参数的解析，并且初始化这个设备要调用 drive\_init\_func 函数，这里面会调用 drive\_new 创建一个设备。

 复制代码

```
1 DriveInfo *drive_new(QemuOpts *all_opts, BlockInterfaceType block_default_type, Error *
2 {
3     const char *value;
4     BlockBackend *blk;
5     DriveInfo *dinfo = NULL;
6     QDict *bs_opts;
```



```


7     QemuOpts *legacy_opts;
8     DriveMediaType media = MEDIA_DISK;
9     BlockInterfaceType type;
10    int max_devs, bus_id, unit_id, index;
11    const char *werror, *rerror;
12    bool read_only = false;
13    bool copy_on_read;
14    const char *filename;
15    Error *local_err = NULL;
16    int i;
17    .....
18    legacy_opts = qemu_opts_create(&qemu_legacy_drive_opts, NULL, 0,
19                                  &error_abort);
20    .....
21    /* Add virtio block device */
22    if (type == IF_VIRTIO) {
23        QemuOpts *devopts;
24        devopts = qemu_opts_create(qemu_find_opts("device"), NULL, 0,
25                                  &error_abort);
26        qemu_opt_set(devopts, "driver", "virtio-blk-pci", &error_abort);
27        qemu_opt_set(devopts, "drive", qdict_get_str(bs_opts, "id"),
28                    &error_abort);
29    }
30
31    filename = qemu_opt_get(legacy_opts, "file");
32    .....
33    /* Actual block device init: Functionality shared with blockdev-add */
34    blk = blockdev_init(filename, bs_opts, &local_err);
35    .....
36    /* Create legacy DriveInfo */
37    dinfo = g_malloc0(sizeof(*dinfo));
38    dinfo->opts = all_opts;
39
40    dinfo->type = type;
41    dinfo->bus = bus_id;
42    dinfo->unit = unit_id;
43
44    blk_set_legacy_dinfo(blk, dinfo);
45
46    switch(type) {
47    case IF_IDE:
48    case IF_SCSI:
49    case IF_XEN:
50    case IF_NONE:
51        dinfo->media_cd = media == MEDIA_CDROM;
52        break;
53    default:
54        break;
55    }
56    .....
57 }

```

在 `drive_new` 里面，会解析 `qemu` 的启动参数。对于 `virtio` 来讲，会解析 `device` 参数，把 `driver` 设置为 `virtio-blk-pci`；还会解析 `file` 参数，就是指向那个宿主机上的文件。


接下来，`drive_new` 会调用 `blockdev_init`，根据参数进行初始化，最后会创建一个 `DriveInfo` 来管理这个设备。

我们重点来看 `blockdev_init`。在这里面，我们发现，如果 `file` 不为空，则应该调用 `blk_new_open` 打开宿主机上的硬盘文件，返回的结果是 `BlockBackend`，对应我们上面讲原理的时候的 `virtio` 的后端。

 复制代码

```
1 BlockBackend *blk_new_open(const char *filename, const char *reference,
2                             QDict *options, int flags, Error **errp)
3 {
4     BlockBackend *blk;
5     BlockDriverState *bs;
6     uint64_t perm = 0;
7     .....
8     blk = blk_new(perm, BLK_PERM_ALL);
9     bs = bdrv_open(filename, reference, options, flags, errp);
10    blk->root = bdrv_root_attach_child(bs, "root", &child_root,
11                                       perm, BLK_PERM_ALL, blk, errp);
12    return blk;
13 }
```

接下来的调用链为：`bdrv_open->bdrv_open_inherit->bdrv_open_common`.

 复制代码

```
1 static int bdrv_open_common(BlockDriverState *bs, BlockBackend *file,
2                             QDict *options, Error **errp)
3 {
4     int ret, open_flags;
5     const char *filename;
6     const char *driver_name = NULL;
7     const char *node_name = NULL;
8     const char *discard;
9     QemuOpts *opts;
10    BlockDriver *drv;
11    Error *local_err = NULL;
```

```

12 .....
13     drv = bdrv_find_format(driver_name);
14 .....
15     ret = bdrv_open_driver(bs, drv, node_name, options, open_flags, errp);
16 .....
17 }
18
19 static int bdrv_open_driver(BlockDriverState *bs, BlockDriver *drv,
20                             const char *node_name, QDict *options,
21                             int open_flags, Error **errp)
22 {
23 .....
24     bs->drv = drv;
25     bs->read_only = !(bs->open_flags & BDRV_O_RDWR);
26     bs->opaque = g_malloc0(drv->instance_size);
27
28     if (drv->bdrv_open) {
29         ret = drv->bdrv_open(bs, options, open_flags, &local_err);
30     }
31 .....
32 }

```

在 `bdrv_open_common` 中，根据硬盘文件的格式，得到 `BlockDriver`。因为虚拟机的硬盘文件格式有很多种，`qcow2` 是一种，`raw` 是一种，`vmdk` 是一种，各有优缺点，启动虚拟机的时候，可以自由选择。

对于不同的格式，打开的方式不一样，我们拿 `qcow2` 来解析。它的 `BlockDriver` 定义如下：

 复制代码

```

1 BlockDriver bdrv_qcow2 = {
2     .format_name      = "qcow2",
3     .instance_size    = sizeof(BDRVQcow2State),
4     .bdrv_probe        = qcow2_probe,
5     .bdrv_open         = qcow2_open,
6     .bdrv_close        = qcow2_close,
7     .....
8     .bdrv_snapshot_create = qcow2_snapshot_create,
9     .bdrv_snapshot_goto   = qcow2_snapshot_goto,
10    .bdrv_snapshot_delete  = qcow2_snapshot_delete,
11    .bdrv_snapshot_list    = qcow2_snapshot_list,
12    .bdrv_snapshot_load_tmp = qcow2_snapshot_load_tmp,
13    .bdrv_measure          = qcow2_measure,
14    .bdrv_get_info          = qcow2_get_info,
15    .bdrv_get_specific_info = qcow2_get_specific_info,


```

```

16
17     .bdrv_save_vmstate      = qcow2_save_vmstate,
18     .bdrv_load_vmstate     = qcow2_load_vmstate,
19
20     .supports_backing       = true,
21     .bdrv_change_backing_file = qcow2_change_backing_file,
22
23     .bdrv_refresh_limits    = qcow2_refresh_limits,
24     .....
25 };

```

根据上面的定义，对于 qcow2 来讲，bdrv\_open 调用的是 qcow2\_open。

 复制代码

```

1 static int qcow2_open(BlockDriverState *bs, QDict *options, int flags,
2                       Error **errp)
3 {
4     BDRVQcow2State *s = bs->opaque;
5     QCow2OpenCo qoc = {
6         .bs = bs,
7         .options = options,
8         .flags = flags,
9         .errp = errp,
10        .ret = -EINPROGRESS
11    };
12
13    bs->file = bdrv_open_child(NULL, options, "file", bs, &child_file,
14                              false, errp);
15    qemu_coroutine_enter(qemu_coroutine_create(qcow2_open_entry, &qoc));
16    .....
17 }


```

在 qcow2\_open 中，我们会通过 qemu\_coroutine\_enter 进入一个协程 coroutine。什么叫协程呢？我们可以简单地将它理解为用户态自己实现的线程。

前面咱们讲线程的时候说过，如果一个程序想实现并发，可以创建多个线程，但是线程是一个内核的概念，创建的每一个线程内核都能看到，内核的调度也是以线程为单位的。这对于普通的进程没有什么问题，但是对于 qemu 这种虚拟机，如果在用户态和内核态切换来切换去，由于还涉及虚拟机的状态，代价比较大。

但是，qemu 的设备也是需要多线程能力的，怎么办呢？我们就在用户态实现一个类似线程的东西，也就是协程，用于实现并发，并且不被内核看到，调度全部在用户态完成。

从后面的读写过程可以看出，协程在后端经常使用。这里打开一个 qcow2 文件就是使用一个协程，创建一个协程和创建一个线程很像，也需要指定一个函数来执行，qcow2\_open\_entry 就是协程的函数。

 复制代码

```
1 static void coroutine_fn qcow2_open_entry(void *opaque)
2 {
3     QCow2OpenCo *qoc = opaque;
4     BDRVQcow2State *s = qoc->bs->opaque;
5
6     qemu_co_mutex_lock(&s->lock);
7     qoc->ret = qcow2_do_open(qoc->bs, qoc->options, qoc->flags, qoc->errp);
8     qemu_co_mutex_unlock(&s->lock);
9 }
```

我们可以看到，qcow2\_open\_entry 函数前面有一个 coroutine\_fn，说明它是一个协程函数。在 qcow2\_do\_open 中，qcow2\_do\_open 根据 qcow2 的格式打开硬盘文件。这个格式[官网](#)就有，我们这里就不花篇幅解析了。

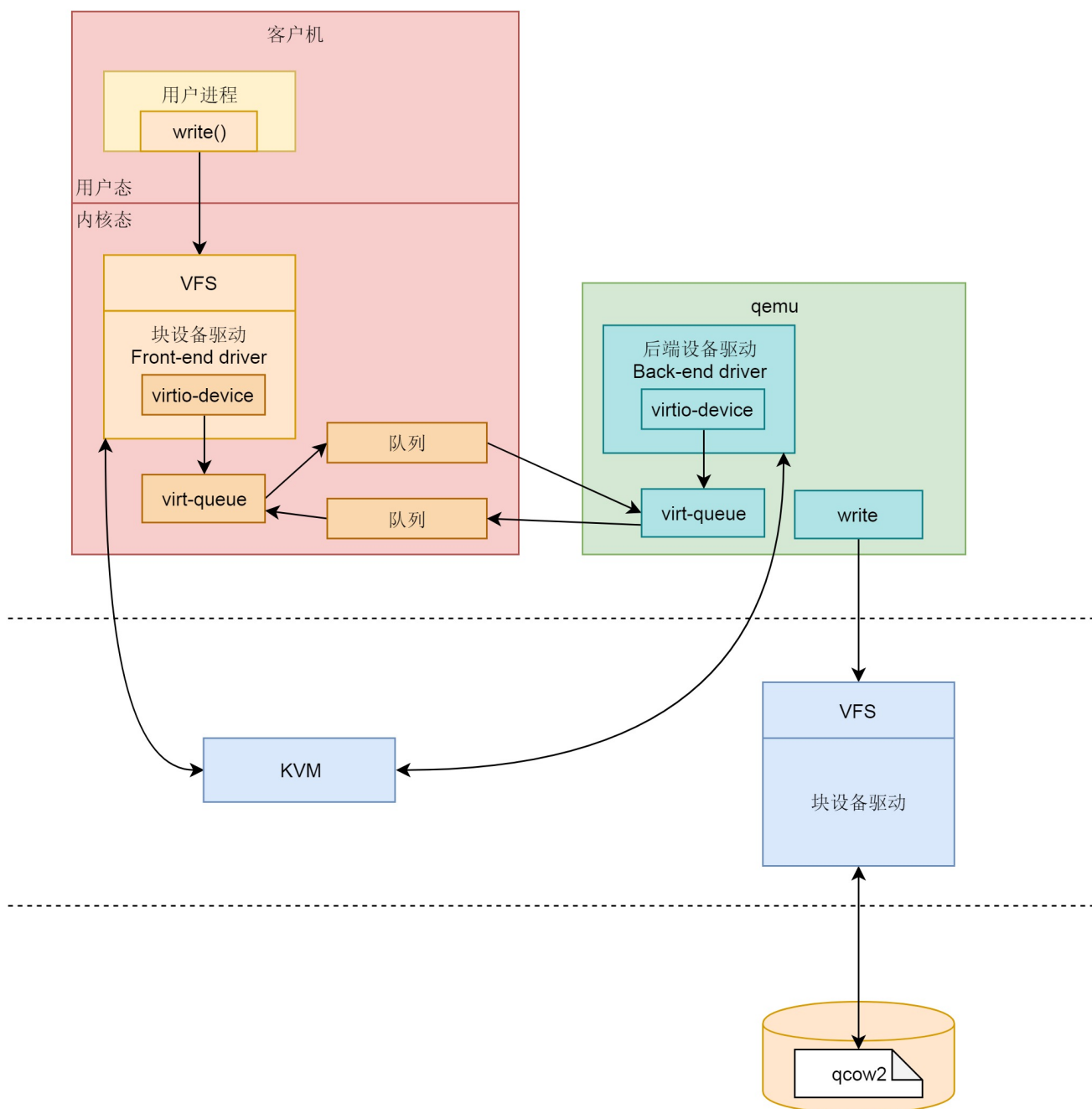
## 总结时刻

我们这里来总结一下，存储虚拟化的过程分为前端、后端和中间的队列。

前端有前端的块设备驱动 Front-end driver，在客户机的内核里面，它符合普通设备驱动的格式，对外通过 VFS 暴露文件系统接口给客户机里面的应用。这一部分这一节我们没有讲，放在下一节解析。

后端有后端的设备驱动 Back-end driver，在宿主机的 qemu 进程中，当收到客户机的写入请求的时候，调用文件系统的 write 函数，写入宿主机的 VFS 文件系统，最终写到物理硬盘设备上的 qcow2 文件。

中间的队列用于前端和后端之间传输数据，在前端的设备驱动和后端的设备驱动，都有类似的数据结构 virt-queue 来管理这些队列，这一部分这一节我们也没有讲，也放到下一节解析。



## 课堂练习

对于 qemu-kvm 来讲，qcow2 是一种常见的文件格式。它有精妙的格式设计，从而适应虚拟化的场景，请你研究一下这个文件格式。

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。

# 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 52 | 计算虚拟化之内存：如何建立独立的办公室？

## 精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。