作者：彭东林
邮箱：pengdonglin137@163.com

内核版本：4.14
Qemu：5.1
OpenAMP：2020.10

下面分几个部分来分析：

1. virtio 规范 1
2. virtio device 注册 4
3. virtio driver 的注册 6
4. 前端向后端发送数据 11
5. 后端接收前端发送过来的数据 13
6. 后端向前端发送数据 15
7. 前端接收后端发送的数据 16

以 mmio 类型的 virtio 为例进行分析。

1. virtio 规范

https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.pdf
内核中 virtio 子系统（前端）与后端的交互都遵守这个协议。

下面是协议中规定的 mmio 总线类型的后端 virtio 设备的寄存器布局信息（4.2.2）：

名称	偏移	读写	功能	备注
Magic Value（幻数）	0x0	只读	固定为 0x74726976，小端格式存储，等价于”virt”字符串	前端代码初始化时回检查该值
Version（版本）	0x4	只读	设备版本号，最新的是 2，老的设备是 1	
Device ID （设备 ID）	0x8	只读	Virtio 子系统设备 ID，表示不同类型的 virtio 设备，比如 network card，block device，console，rpmsg，SCSI host 等，参考	

Vendor IDI （厂商 ID）	0xC	只读	Virtio 子系统厂商 ID	
DeviceFeatures （设备特性）	0x10	只读	返回一个 32 位的 flag 标志位，可以通过 DeviceFeaturesSel 控制输出哪一组特性集： DeviceFeaturesSel*32~ DeviceFeaturesSel*32+31	
DeviceFeaturesSel （设备特性选择）	0x14	只写	设备（宿主机）特性集选择	
DriverFeatures （驱动特性）	0x20	只写	能够被前端驱动支持并激活的设备特性，通过 DriverFeaturesSel 可以控制当前配置的是那一组特性集	
DriverFeaturesSel （驱动特性选择）	0x24	只写	激活的（客户机）特性集选择	
QueueSel （队列选择）	0x30	只写	虚拟队列索引，从 0 开始	
QueueNumMax	0x34	只读	被 QueueSel 选中的虚拟队列中元素的最大个数	
QueueNum	0x38	只写	驱动实际使用的虚拟队列（QueueSel）中元素的个数	
QueueReady	0x44	读写	驱动向这个 bit 写 1 来通知设备它可以从这个虚拟队列（QueueSel）上执行请求	

QueueNotify	0x50	只写	驱动向这个寄存器写值表示虚拟队列有新的 buffer 需要处理。如果还没有协商 VIRTIO_F_NOTIFICATION_DATA，那么需要写入虚拟队列的编号，否则需要写入如下的数据结构：
			le32 {
			vqn:16; // 虚拟队列的编号
			next_off:15;
			next_wrap:1;
			}，这个结构中包含的信息使得后端代码不需要查询内存中的 virtqueue 就可以知道虚拟队列中可用的数据，使效率得到提升	

InterruptStatus	0x60	只读	驱动读写这个寄存器来获得设备中断状态，bit0 表示设备使用了虚拟队列中的一个 bufer，bit1 表示设备的配置发生变化	
InterruptACK	0x64	只写	写入 InterruptStatus 定义的值来通	

			知设备对应的中断已经处理完毕	
Status	0x70	读写	获取或者设置当前设备的状态，写入 0 会触发设备复位	
QueueDescLow	0x80	只写	Vring 中 desc 的起始物理地址的低 32 位	Payload
QueueDescHigh	0x84	只写	Vring 中 desc 的起始物理地址的高 32 位	
QueueDriverLow	0x90	只写	Vring 中 avail 的起始物理地址的低 32 位	前端->后端
QueueDriverHigh	0x94	只写	Vring 中 avail 的起始物理地址的高 32 位	
QueueDeviceLow	0xa0	只写	Vring 中 used 的起始物理地址的低 32 位	后端->前端
QueueDeviceHigh	0xa4	只写	Vring 中 used 的起始物理地址的高 32 位	
ConfigGeneration	0xfc	只读	用于获取判断设备的 config 是否在 driver 配置期间发生变化，具体是驱动先读取该值，然后根据 config 的内容开始配置，配置结束后，再读写该值，如果两次读到的值相同的，配置完毕，否则还还需要从新读取 config 进行配置，知道配置前后从这个寄存器读到两个相同的值	
Config	0x100+	读写	字节对齐，存放设备特定的配置，空间大小取决于设备和驱动	

下面是老的 virtio 设备跟上面的不同之处：

名称	偏移	读写	功能	备注
Magic Value	0x0			
Version	0x4			
DeviceID	0x8			
VendorID	0xC			
HostFeatures	0x10			
HostFeaturesSel	0x14			
GuestFeatures	0x20			
GuestFeaturesSel	0x24			
GuestPageSize	0x28	只写	驱动使用的页的大小，单位是字节，满足 2 的指数次方	
QueueSel	0x30			
QueueNumMax	0x34			
QueueNUM	0x38			
QueueAlign	0x3C	只写	Used Ring 的对齐大小，字节为单	

			位，满足 2 的指数次方	
QueuePFN	0x40	读写	Vring 的 desc 区域的物理起始地址以页大小为单位的索引。当驱动不再使用该 queue 时，需要写入 0	
QueueNotify	0x50			
InerruptStatus	0x60			
InterruptACK	0x64			
Status	0x70			
Config	0x100+			

2. virtio device 注册

按照规范，在启动客户机的内核的时候，它的设备树文件中包含了将要创建的 virtio 设备的相关硬件信息，比如寄存器地址返回以及中断号，如下所示：

```
// EXAMPLE: virtio_block device taking 512 bytes at 0x1e000, interrupt 42.
virtio_block@1e000 {
        compatible = "virtio,mmio";
        reg = <0x1e000 0x200>;
        interrupts = <42>;
}
```

这里的寄存器遵循上面的提到的 mmio 总线的内存布局规范，中断号用于后端设备来通知前端驱动，比如 used buffer 中有新的数据包需要处理，或者后端设备的配置发生改变。

对应的内核驱动为：drivers\virtio\virtio_mmio.c

```
716: static struct of_device_id virtio_mmio_match[] = {
717:     { .compatible = "virtio,mmio", },
718:     {},
719: };
720: MODULE_DEVICE_TABLE(of, virtio_mmio_match);
721:
722: #ifdef CONFIG_ACPI
723: static const struct acpi_device_id virtio_mmio_acpi_match[] = {
724:     { "LNRO0005", },
725:     { }
726: };
727: MODULE_DEVICE_TABLE(acpi, virtio_mmio_acpi_match);
728: #endif
729:
730: static struct platform_driver virtio_mmio_driver = {
731:     .probe      = virtio_mmio_probe,
732:     .remove     = virtio_mmio_remove,
733:     .driver     = {
734:         .name   = "virtio-mmio",
735:         .of_match_table = virtio_mmio_match,
736:         .acpi_match_table = ACPI_PTR(virtio_mmio_acpi_match),
737:     },
738: };
```

注册驱动时，会找到对应的 platform device，然后第 731 行的 virtio_mmio_probe 被回调：

virtio_mmio_probe

|-- 分配一个结构体 struct virtio_mmio_device 的空间，用 vm_dev 指向它.

```c
struct virtio_mmio_device {
    struct virtio_device vdev;
    struct platform_device *pdev;

    void __iomem *base;
    unsigned long version;

    /* a list of queues so we can dispatch IRQs */
    spinlock_t lock;
    struct list_head virtqueues;
};
```

|-- vm_dev->vdev.dev.parent = &pdev->dev

|-- vm_dev->pdev = pdev

|-- vm_dev->vdev.config = &**virtio_mmio_config_ops**

这个 ops 后面会被 virtio driver 回调。

```c
static const struct virtio_config_ops virtio_mmio_config_ops = {
    .get        = vm_get,
    .set        = vm_set,
    .generation = vm_generation,
    .get_status = vm_get_status,
    .set_status = vm_set_status,
    .reset      = vm_reset,
    .find_vqs   = vm_find_vqs,
    .del_vqs    = vm_del_vqs,
    .get_features   = vm_get_features,
    .finalize_features = vm_finalize_features,
    .bus_name   = vm_bus_name,
};
```

|-- INIT_LIST_HEAD(&vm_dev->virtqueues)

|-- vm_dev->base 被赋值为这段寄存器地址空间映射后的虚拟起始地址

|-- 读取 VIRTIO_MMIO_MAGIC_VALUE 寄存器，检查 magic number

|-- 读取 VIRTIO_MMIO_VERSION 寄存器，检查协议版本号，目前仅支持 1 和 2

|-- 读取 VIRTIO_MMIO_DEVICE_ID，赋值给 vm_dev->vdev.id.device

|-- 读取 VIRTIO_MMIO_VENDOR_ID，赋值给 vm_dev->vdev.id.vendor

后面在利用这里的 device id 和 vendor id 去匹配对应 virtio driver

|-- 如果协议版本是 1，即老的后端设备，那么需要按照规范，将 PAGE_SIZE 写入到 VIRTIO_MMIO_GUEST_PAGE_SIZE 寄存器中

|-- 设置 dma mask

|-- platform_set_drvdata(pdev, vm_dev)

\-- 调用 register_virtio_device(&vm_dev->vdev)注册 virtio_device

```c
/**
 * virtio_device - representation of a device using virtio
 * @index: unique position on the virtio bus
 * @failed: saved value for VIRTIO_CONFIG_S_FAILED bit (for restore)
 * @config_enabled: configuration change reporting enabled
 * @config_change_pending: configuration change reported while disabled
 * @config_lock: protects configuration change reporting
 * @dev: underlying device.
 * @id: the device type identification (used to match it with a driver).
 * @config: the configuration ops for this device.
 * @vringh_config: configuration ops for host vrings.
 * @vqs: the list of virtqueues for this device.
 * @features: the features supported by both driver and device.
 * @priv: private pointer for the driver's use.
 */
struct virtio_device {
    int index;
    bool failed;
    bool config_enabled;
    bool config_change_pending;
    spinlock_t config_lock;
    struct device dev;
    struct virtio_device_id id;
    const struct virtio_config_ops *config;
    const struct vringh_config_ops *vringh_config;
    struct list_head vqs;
    u64 features;
    void *priv;
};
```

|-- dev->dev.bus = &**virtio_bus**;

后端会调用 virio_bus 的 virtio_dev_match 函数判断 virtio_device 跟 virtio_driver 的 virtio_device_id 列表的某一项是否匹配

|-- 为 virtio_device 分配唯一的 id，赋值给其 index 成员

|-- 初始化 dev->config_enabled 为 false，表示还没有进行配置

|-- 初始化 dev->config_change_pending 为 false，表示还没有配置改变的请求需要处理

|-- dev->config->reset(dev) 复位后端设备，这里会调用

virtio_mmio_config_ops 的 reset 回调函数（vm_reset），这个函数按照规范，向 VIRTIO_MMIO_STATUS 寄存器写入了 0

|-- 获取后端设备状态，将其或上 VIRTIO_CONFIG_S_ACKNOWLEDG，然后写入，这里会回调 **virtio_mmio_config_ops** 的 get_status 和 set_status 回调函数（vm_get_status 和 vm_set_status），操作的是 VIRTIO_MMIO_STATUS 寄存器

|-- INIT_LIST_HEAD(&dev->vqs)

|-- device_register(&dev->dev)

3. virtio driver 的注册

为了简单起见，这里分析 drivers\char\hw_random\virtio-rng.c 驱动，这个驱动需要一个 virtqueue。

```
191: static struct virtio_device_id id_table[] = {
192:     { VIRTIO_ID_RNG, VIRTIO_DEV_ANY_ID },
193:     { 0 },
194: };
195:
196: static struct virtio_driver virtio_rng_driver = {
197:     .driver.name =  KBUILD_MODNAME,
198:     .driver.owner = THIS_MODULE,
199:     .id_table = id_table,
200:     .probe =    virtrng_probe,
201:     .remove =   virtrng_remove,
202:     .scan =     virtrng_scan,
203: #ifdef CONFIG_PM_SLEEP
204:     .freeze =   virtrng_freeze,
205:     .restore =  virtrng_restore,
206: #endif
207: };
208:
209: module_virtio_driver(virtio_rng_driver);
```

第 209 的宏 module_virtio_driver 会调用 register_virtio_driver。

第 191 行，表示这个 virtio driver 支持的 virtio device 的 id 信息，其中 device id 为 VIRTIO_ID_RNG（4）表示类型信息，vendor id 为 VIRTIO_DEV_ANY_ID 表示厂商 ID，这里表示支持所有的厂商 ID

第 200 行，当匹配成功后，会回调这里的 virtrng_probe 函数。

先看一下 virio_bus 的 match 逻辑：

virtio_dev_match(struct device *_dv, struct device_driver *_dr)

|-- struct virtio_device *dev = dev_to_virtio(_dv)

|-- const struct virtio_device_id *ids = drv_to_virtio(_dr)->id_table

|-- 遍历 ids 中的每一项，进行对比：

```
71: static inline int virtio_id_match(const struct virtio_device *dev,
72:                                   const struct virtio_device_id *id)
73: {
74:     if (id->device != dev->id.device && id->device != VIRTIO_DEV_ANY_ID)
75:         return 0;
76:
77:     return id->vendor == VIRTIO_DEV_ANY_ID || id->vendor == dev->id.vendor;
78: }
```

match 通过后，首先调用的是 virtio_bus 的 probe 函数：virtio_dev_probe

virtio_dev_probe

- |-- 回调 dev->config->set_status，将后端设备状态或上 VIRTIO_CONFIG_S_DRIVER
- |-- 接下来回调 dev->config->get_features，获取后端设备的特性信息，然后跟 virtio_dirver 支持的特性信息执行“与”逻辑，得到设备和驱动均支持的特性，赋给 dev->features
- |-- 如果 drv->validate 非空，那么回调 drv->validate(dev)检查特性是否合法
- |-- 回调 dev->config->finalize_features，这里是 vm_finalize_features，这个函数会将最终的 vdev->features 写入到 VIRTIO_MMIO_DRIVER_FEATURES 寄存器中，这里需要结合 VIRTIO_MMIO_DRIVER_FEATURES_SEL 寄存器，将 vdev->features 的高 32 位和低 32 位分别写入 sel1 和 sel0 中
- |-- 调用 virtio_driver 的 probe 函数，即 **virtrng_probe**，等下分析
- |-- 获取后端设备的状态是否包含了 VIRTIO_CONFIG_S_DRIVER_OK，如果没有设置的话，这里会进行设置
- |-- 如果 virtio_driver 的 scan 非空的话，回调 scan 函数，目前这里设置的是 virtrng_scan，这个函数会调用 hwrng_register 注册随机数控制器
- |-- 将 vdev->config_enabled 设置 true，表示已经配置完毕
- |-- 如果有配置改变事件需要处理，那么会回调 drv->config_changed(dev)
- |-- 将 vdev->config_change_pending 设置位 false，表示没有 config change 事件需要处理

下面分析具体 virtio_driver 的 probe 函数：**virtrng_probe**

virtrng_probe

- \-- probe_common(struct virtio_device *vdev)
  - |-- 分配私有结构 struct virtrng_info *vi，其中会有成员来记录后面创建的 virtqueue 结构的指针
  - |-- 初始化 vi 的成员：name、have_data 以及 hwrng
  - |-- 调用 **virtio_find_single_vq**(vdev, random_recv_done, "input")创建 virtqueue 以及 vring 结构。

virtio_find_single_vq (struct virtio_device *vdev, vq_callback_t *c, const char *n)

- |-- 第一个参数是将要使用的 virtio_deivce，第二个参数表示的回调函数在后端通过中断的方式通知前端后，在前端的中断处理程序中会进行回调，第三个参数表示 virqueue 的名字
- \-- 回调 vdev->config->find_vqs(vdev, 1, &vq, callbacks, names, NULL, NULL)，也就是 vm_find_vqs，这个函数会申请创建一个 virtqueue
  - |-- 第一个参数表示 virtio_device，第二个参数表示需要创建的 virtqueue 的个数，第三个参数是存放创建出来的 virtqueue 地址的指针数组，第四个参数也是一个指针数组，存放的是每一个 virtqueue 对应的回调函数，当 virtqueue 发生中断后，在中断处理程序中会回调对应的 callback，第五个参数同样是指针数组，存放的是每个 virtqueue 的名字，第六个参数表示上下文
  - |-- 获取 virtio device 的中断资源，然后申请注册中断处理程序 **vm_interrupt**，这个后面分析

\-- 循环调用调用 vm_setup_vq 创建每一个 virtqueue，将放回的 virtqueue 的地址存放到传入的 vq 指针数组中

vm_setup_vq(vdev, i, callbacks[i], names[i], ctx ? ctx[i] : false)

|-- 参数 i 表示要创建的 virtqueue 的索引，callback 表示回调函数，names 为 virtqueue 的名称，如果 ctx 为 NULL，传入 false

|-- 将 index 写入 VIRTIO_MMIO_QUEUE_SEL 寄存器，设置下面要操控的是后端哪一个 queue（对应 virtqueue）

|-- 判断要设置 queue 是否已经配置过，对于新的设备，如果 queue 没有配置的话，VIRTIO_MMIO_QUEUE_READY 应该是 0，如果是老设备，VIRTIO_MMIO_QUEUE_READY 应该为 0

|-- 分配一个 struct virtio_mmio_vq_info *info 结构

|-- 读取 VIRTIO_MMIO_QUEUE_NUM_MAX 寄存器的值，获得 virtio device 支持的 virtqueue 中元素的最大个数

\-- 调用 **vring_create_virtqueue** 返回创建的一个 virtqueue 指针，返回地址存放在 struct virtqueue *vq 中

|-- 获取上面创建的 virtqueu 的 vring 中元素的个数，存放到 VIRTIO_MMIO_QUEUE_NUM 中，这样后端设备就知道前端 vring 中实际使用了几个元素

|-- 如果是老设备，将 PAGE_SIZE 存入 VIRTIO_MMIO_QUEUE_ALIGN，将 vring 中 desc 区域的起始物理地址对应的页框号写入 VIRTIO_MMIO_QUEUE_PFN 寄存器

|-- 如果是新设备，将 vring 中 desc 区域的起始物理地址高低 32 位分别写入 VIRTIO_MMIO_QUEUE_DESC_HIGH 和 VIRTIO_MMIO_QUEUE_DESC_LOW 寄存器中

|-- 将 vring 中 avail 区域的起始物理地址的高低 32 位分别存入 VIRTIO_MMIO_QUEUE_AVAIL_HIGH 和 VIRTIO_MMIO_QUEUE_AVAIL_LOW 寄存器中

|-- 将 vring 中 used 区域的起始物理地址的高低 32 位分别存入 VIRTIO_MMIO_QUEUE_AVAIL_LOW 和 VIRTIO_MMIO_QUEUE_USED_LOW 寄存器中

|-- 向 VIRTIO_MMIO_QUEUE_READY 寄存器写入 1，表示 queue 可以接受处理请求

|-- 将上面创建的 virtqueue 的地址赋给 info->vq，将 info 赋给 vq->priv

|-- 将 info 加入到 vm_dev->virtqueues 链表中，这样通过 virtio_mmio_device 的 virtqueues 链表就可以找到属于该 virtio device 的所有的 virtqueue

|-- 返回上面创建的 virtqueue 的地址

vring_create_virtqueue

```
389:     /* Create the vring */
390:     vq = vring_create_virtqueue(index, num, VIRTIO_MMIO_VRING_ALIGN, vdev,
391:                 true, true, ctx, vm_notify, callback, name);
```

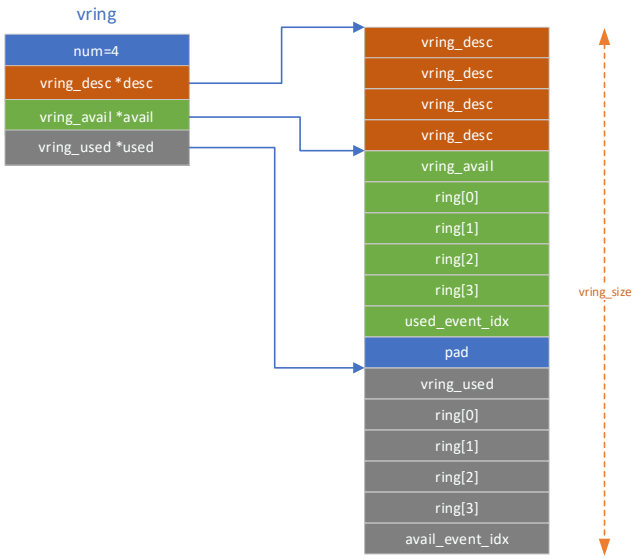
|-- 第一个参数 index 表示 virtqueue 的索引，第二个参数表示 virtqueue 中元素的最大个数，第三个参数表示 vring 的对齐大小，这里传入的是 PAGE_SIZE，第四个参数为

virtio_device，第五和六个参数先忽略，第七个参数表示上下文，第八个参数 vm_notify 为前端驱动通知后端有新的 avail buffer 需要处理，第九个参数 callback 为 virtqueue 的回调函数，第十个参数表示 virtqueue 的名称

|-- 检查传入的元素最大个数是否为 2 的幂次方，不是的话，返回错误

|-- 计算 vring 占用的空间大小，然后分配对齐后的大小，并得到这块内容的虚拟起始地址 queue 和物理起始地址 dma_addr，还要保证这部分内存物理连续，因为这部分内存将来要跟宿主机共享，其中 vring_desc 需要 16 字节对齐，vring_avail 需要 2 字节对齐，vring_used 需要 4 字节对齐

```c
static inline unsigned vring_size(unsigned int num, unsigned long align)
{
    return ((sizeof(struct vring_desc) * num + sizeof(__virtio16) * (3 + num)
         + align - 1) & ~(align - 1))
        + sizeof(__virtio16) * 3 + sizeof(struct vring_used_elem) * num;
}
```



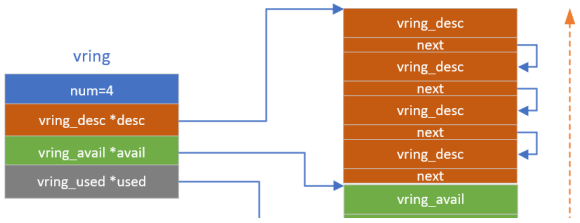
\-- vq = **__vring_new_virtqueue**(index, vring, vdev, weak_barriers, context, notify, callback, name) 这个函数负责创建 virtqueue 并返回

|-- 分配一个 vring_virtqueue 结构（其中包含 virtqueue），并在结尾多分配 vring.num 个 vring_desc_state 结构 vq，这个结构如下：

```c
struct vring_virtqueue {
    struct virtqueue vq;

    /* Actual memory layout for this queue */
    struct vring vring;

    /* Can we use weak barriers? */
    bool weak_barriers;

    /* Other side has made a mess, don't try any more. */
    bool broken;

    /* Host supports indirect buffers */
    bool indirect;

    /* Host publishes avail event idx */
    bool event;

    /* Head of free buffer list. */
    unsigned int free_head;
    /* Number we've added since last sync. */
    unsigned int num_added;

    /* Last used index we've seen. */
    u16 last_used_idx;

    /* Last written value to avail->flags */
    u16 avail_flags_shadow;

    /* Last written value to avail->idx in guest byte order */
    u16 avail_idx_shadow;

    /* How to notify other side. FIXME: commonalize hcalls! */
    bool (*notify)(struct virtqueue *vq);

    /* DMA, allocation, and size information */
    bool we_own_ring;
    size_t queue_size_in_bytes;
    dma_addr_t queue_dma_addr;

    /* Per-descriptor state. */
    struct vring_desc_state desc_state[];
} « end vring_virtqueue » ;

#define to_vvq(_vq) container_of(_vq, struct vring_virtqueue, vq)
```

|-- 将 vring 赋给 vq**->ring**

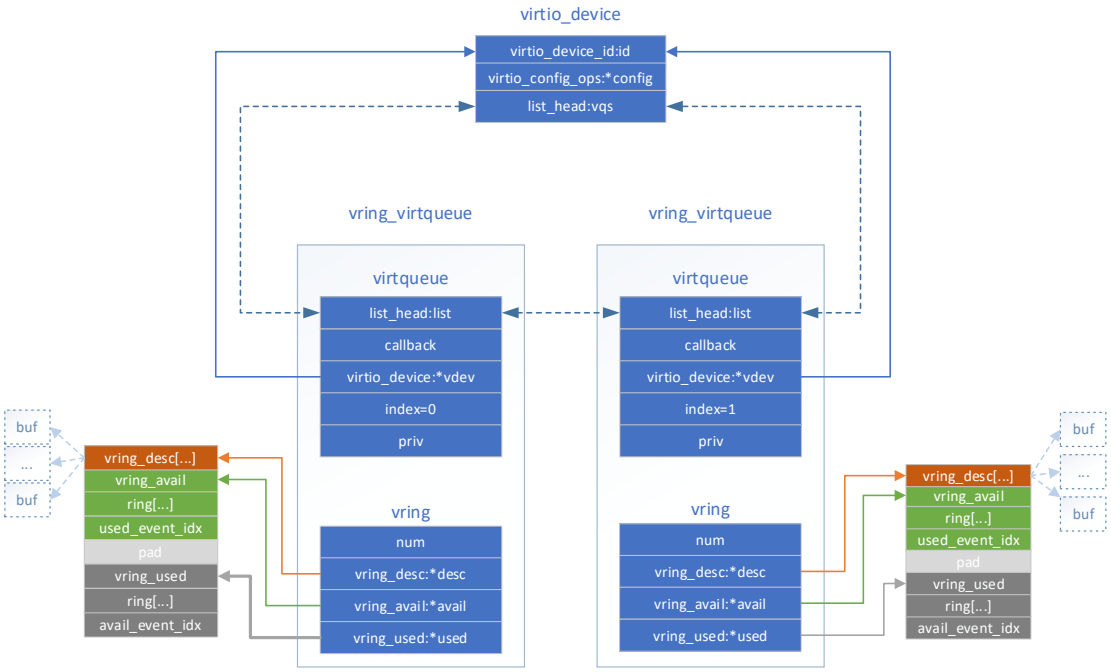
|-- 将 callback 函数赋给 vq->vq.callback，来中断后，会在中断处理程序里回调该函数
|-- 将 vdev 赋给 vq->vq.vdev
|-- 将 name 赋给 vq->vq.name
|-- vq->vq.num_free = vring.num，表示当前空闲的 desc 个数
|-- 将 index 赋给 vq->vq.index，表示这个 virtqueue 的 index
|-- 将 notify 赋给 vq->notify，表示前端通知后端有 avail buffer 需要处理
|-- 将 vq->last_used_idx 设为 0，表示 use_buffer 中下一个可用的索引
|-- 将 vq->avail_flags_shadow 设置为 0
|-- 将 vq->avail_idx_shadow 设置为 0，表示 vring_avail 中下一个空闲的 ring 的索引
|-- 将 vq->num_added 设置为 0
|-- 将 vq->vq 加入到 vdev->vqs 链表中
|-- 如果 context 为 false，并且 vdev 支持 VIRTIO_RING_F_INDIRECT_DESC，那么设置 vq->indirect 为 true，表示使用的 desc 空间不用之前分配，而会另外分配
|-- 如果 vdev 支持 VIRTIO_RING_F_EVENT_IDX，设置 vq->event 为 true，否则为 false
|-- 如果 callback 为 NULL，会将 vq->avail_flags_shadow 或上 VRING_AVAIL_F_NO_INTERRUPT，表示前端没有 callback 可供回调，如果 vq->event 也会 NULL，会将 vq->avail_flags_shadow 赋给 vq->vring.avail->flags
|-- 设置 vq->free_head 为 0，表示下一个空闲的 desc 的索引
|-- 将 vq->vring 的 desc 前后连接起来：



|-- 将这个函数开始分配的 vring_desc_state 数组空间清零
|-- 返回&vq->vq，即刚创建出来的 virtqueue 的地址
|-- 将 vring 共享内存的物理首地址赋给 vring_virtqueue 的 queue_dma_addr
|-- 将 vring_size 赋给 vring_virtqueue 的 queue_size_in_bytes
|-- 将 vring_virtqueue 的 we_own_ring 赋给 true

至此，virtio_device 和 virtio_driver 的注册就分析完了，主要就是 virtio_driver 会跟 virtio_device 进行匹配，匹配成功后，调用 virtio_driver 的 probe，在 probe 中会分配 virtqueue、vring 以及 vring 指向的共享内存。

下面是 virtio_device、virtqueue、vring 以及环形队列的关系：



下面分析数据流。

4. 前端向后端发送数据

还是以之前的例子 drivers\char\hw_random\virtio-rng.c。
当上层需要获取随机数时，最终会回调到 virtio_read(struct hwrng *rng, void *buf, size_t size, bool wait)

\-- register_buffer(vi, buf, size) 这个函数会分配一个 sg 结构，然后更新 avail_buffer，最后通知后端处理 avail_buffer 中的数据

|-- sg_init_one(&sg, buf, size) 将 buf 和 size 存入一个 sg 结构

|-- virtqueue_add_inbuf(vi->vq, &sg, 1, buf, GFP_KERNEL)

|-- virtqueue_add(vq, &sg, num, 0, 1, data, NULL, gfp)

|-- 第一个参数表示 virtqueue，第二个参数表示 sg 列表，其中记录了要发送数据的地址和长度，第三个参数表示 sg 的总个数，第四个参数表示要求宿主机 read 的 sg 的个数，第五个参数表示要求宿主机写的 sg 的个数，每个 sg 会对应一个 vring_desc，这里只传递了一个 sg，并且是要宿主机写的

|-- 根据 virtqueue 获取 vring_virtqueue 结构 vq，因为 virtqueue 是 vring_virtqueue 的成员

|-- 如果 vq->indirect 非空，那么会单独分配需要的 vring_desc，否则会使用之前 vring 中指向的 vring_desc

|-- 遍历需要宿主机读的 sg，逐一填充 vring_desc，对于非 indirect，这里会从 vq->free_head 索引的 vring_desc 开始填充，对于 indirect，是从索引为 0 的 vring_desc 开始。如果是 indect 的

话，会从 vring_desc 的第 0 个开始填充，vring_desc 的 addr 是物理地址，所以会调用 vring_map_one_sg(vq, sg, DMA_TO_DEVICE)获取 sg 中存放数据的内存的物理地址，DMA_TO_DEVICE 告诉系统需要 flush cache，如果 sg 有多个，那么会将这些 vring_desc 用 next 前后连接，并且设置这些 vring_desc 的 flags 设置为 VRING_DESC_F_NEXT

|-- 遍历需要宿主机写的 sg，不同之处是，在获取 sg 中 buffer 的物理地址的时候传递的是 DMA_FROM_DEVICE，此时系统会 invalidate cache，并且将 vring_desc 的 flags 赋值为 VRING_DESC_F_NEXT | VRING_DESC_F_WRITE

|-- 将上面填充好的这些 vring_desc 的最后一个的 flags 的 VRING_DESC_F_NEXT 清除，表示这个本次传输的最后一个 vring_desc

|-- 对于 indirect，填充 vq->vring.desc[vq->free_head]，特殊之处是 vring_desc 的 flags 被设置为了 VRING_DESC_F_INDIRECT，addr 被设置为了上面单独分配的 vring_desc 的物理首地址，len 为分配的 vring_desc 的总长度，这样后端处理时根据 indirect 首先获取真正的 vring_desc 的物理地址，然后再做进一步处理

|-- 更新 vq->vq.num_free，表示 vq->vring.vring_desc 数组中剩余可用的 vring_desc，对于 indirect，因为只是用了 vq->vring.desc[vq->free_head]，真正的 vring_desc 是单独分配的，所以 numfree 减 1 即可，对于非 indirect，需要减去上面填充的 vring_desc 的个数，也就是 total_sg

|-- 更新 vq->free_head, 表示 vq->vring->vring_desc 数组中下一个空闲的 vring_desc 的索引，对于 indirect，free_head 设置为 vq->vring.desc[vq->free_head].next，对于非 indirect，设置为上面填充完的 vring_desc 中最后一个的 next 的值

|-- 将传入的 data 赋值给 vq->desc_state[vq->free_head].data

|-- 对于 indirect，将 vq->desc_state[vq->free_head].indir_desc 赋值为新分配的 vring_desc 的首地址

|-- 对于非 indiect，将 vq->desc_state[head].indir_desc 赋值为传入的 ctx 参数

|-- 获取 vq->vring.avail 中空闲的 ring 的索引（vq->avail_idx_shadow），然后将该 ring 其赋值为 vq->free_head，即 vq->vring.avail->ring[avail]=vq->free_head

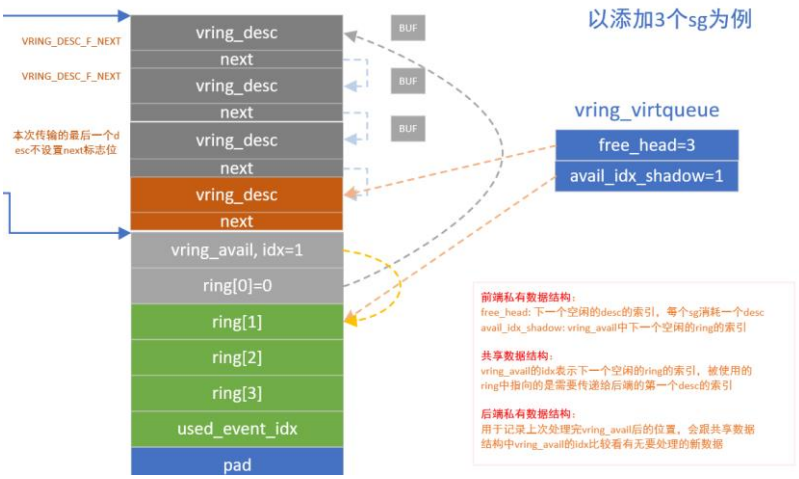
|-- vq->avail_idx_shadow++

|-- vq->vring.avail->idx = vq->avail_idx_shadow

|-- vq->num_added++

|-- 如果 num_added 正好等于(1 << 16) – 1, 调用 virtqueue_kick(_vq), 因为 num_added 表示单次要处理的 vring_desc 的个数，最大只有 1<<16-1 个，所以这里需要 kick，在 kick 中会将 num_added 重新设置为 0

|-- 下面是此时的 vring 结构的状态：（假设 sg 的个数是 3 个 out）



|-- virtqueue_kick(vi->vq)

|-- 调用 virtqueue_kick_prepare(vq)，这个函数用于判断是否需要执行下面的 notify，同时也会把 num_used 清零，如果 vring_used 设置了 VRING_USED_F_NO_NOTIFY，表示不需要通知后端，返回 false。如果返回 true 的话，才会真正调用 virtqueue_notify(vq)。

|-- wait_for_completion_killable(&vi->have_data) 等待 callback 函数 random_recv_done 被回调，当宿主机跟新了 use_buffer 后，会将客户机注入中断，此时中断处理程序会被执行，在其中会回调 vq->callback

5. 后端接收前端发送过来的数据

openamp 中后端的处理逻辑：lib\virtio\virtqueue.c：rpmsg_virtio_get_rx_buffer

大致的处理逻辑是：

在后端有一个私有的数据结构来记录上次处理完（初始值为 0）的 vring_avail 的 vring 的索引，也即上次 vring_avail->idx 的值（idx 指向的是紧接着下一个空闲的 ring 的索引值）。

1、 比较自己记录的 vq_available_idx 跟 vq_ring.avail->idx，如果不相等，表示确实有数据要处理，那么继续下面的步骤
2、 获取 vq_ring.avail->ring[vq_available_idx]的内容，这里存放的是要处理的第一个 desc 的索引编号 desc_idx
3、 获取 desc 的值 vq_ring.desc[desc_idx].addr，由于这个值存放的是前端看到的物理地址，所以还需要将其映射为后端自己的地址 buffer
4、 同时还要获取 vq_ring.desc[desc_idx].len，表示数据长度
5、 这样后端就获得了要处理的数据

Qemu 中后端的处理逻辑： hw\virtio\virtio-rng.c：chr_read

后端的私有数据结构里记录了上次处理完 vring_avail 后的索引 last_avail_idx，是下一个空闲的 ring 的索引。

handle_input

\-- virtio_rng_process

|-- size = get_request_size(vrng->vq, quota)

计算后端只写的 desc 的 len 的累加值，后面会分配大小为 size 的 buffer，并将 vring_desc 中的数据拷贝到 buffer 中。

\-- virtqueue_get_avail_bytes(vq, &in, &out, quota, 0)

这个函数的逻辑如下：根据之前记录的 last_avail_idx 索引到对应的 desc，然后从这个 desc 开始利用 next 字段进行遍历，直到遇到没有标记 VRING_DESC_F_NEXT 的 desc 位置，并且将后端只读和只写的 desc 的 len 分别累加到 out 和 in 中

|-- 返回 in 的值，因为 rng 只需要后端写

|-- rng_backend_request_entropy(vrng->rng, size, chr_read, vrng)

这里会创建一个 RngRequest 结构，将 chr_read 设置为回调函数，分配 size 大小的 buffer，最后将 RngRequest 添加到一个 aio 队列中，后面会 chr_read 会被回调

chr_read (void *opaque, const void *buf, size_t size)

|-- 这里的 buf 就是刚才分配的，size 表示 buffer 的大小

|-- elem = **virtqueue_pop**(vrng->vq, sizeof(VirtQueueElement))

这个函数从 last_avail_idx 记录的位置开始遍历 desc，将每个要处理的 desc 指向的空间映射为 iovec 数组的中的一项，遍历结束后，填充一个 VirtQueueElement 结构，其中记录了每个 desc 的转换后的信息，这个函数会将 last_avail_idx 加 1，最后返回 VirtQueueElement 结构的地址。下面是 VirtQueueElement 的成员说明：

unsigned int index;	last_avail_idx 递增之前指向的 ring 的值，即要处理的第一个 desc 的索引
unsigned int len;	表示将来后端实际写入的字节数
unsigned int ndescs;	1
unsigned int out_num;	后端只读的 desc 的个数
unsigned int in_num;	后端只写的 desc 的个数
hwaddr *in_addr;	后端只写的 desc 记录的缓冲区的物理地址列表
hwaddr *out_addr;	后端只读的 desc 记录的缓冲区的物理地址列表
struct iovec *in_sg;	后端只写的 desc 记录的缓冲区的映射后的地址和长度列表
struct iovec *out_sg;	后端只读的 desc 记录的缓冲区的映射后的地址和长度列表（因为物理地址是前端看到的，后端需要将其映射为自己的看到的地址）

|-- len = iov_from_buf(elem->in_sg, elem->in_num, 0, buf + offset, size - offset)

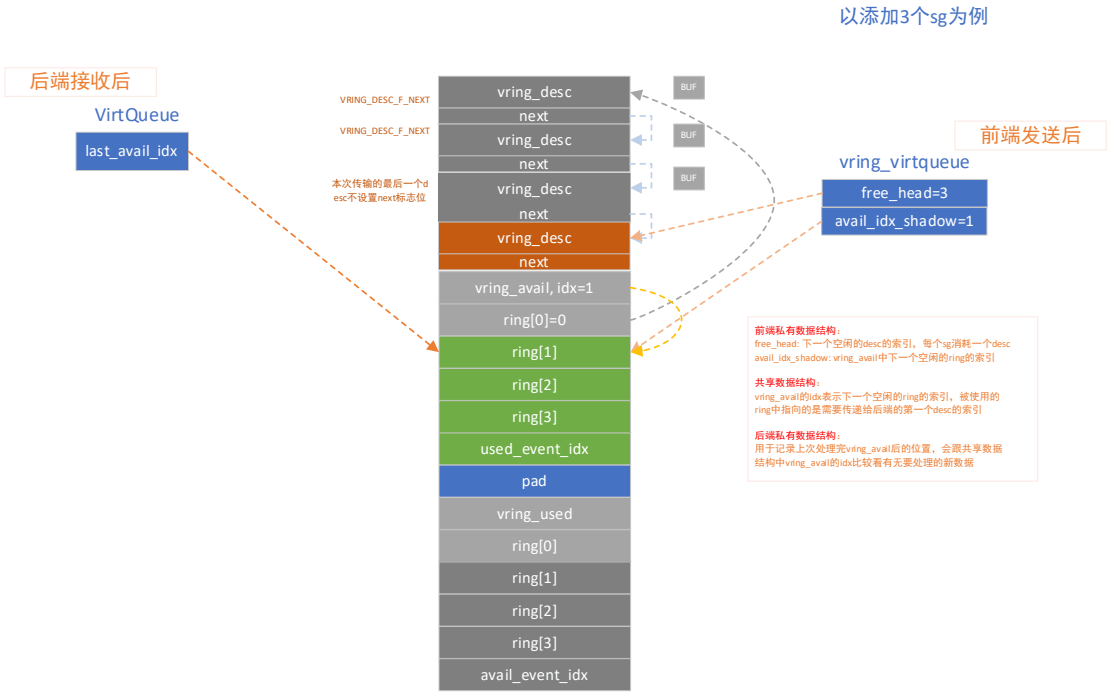
将 buf 中的数据拷贝到 VirtQueueElement 结构指定的空间

|-- **virtqueue_push**(vrng->vq, elem, len)

更新 vring_used，len 为直接写入的数据字节数，接下来分析

|-- virtio_notify(vdev, vrng->vq)

通知前端



6. 后端向前端发送数据

在 Qemu 会调用 **virtqueue_push** 更新 vring_used，然后通知前端

virtqueue_push

|-- virtqueue_fill(vq, elem, len, 0)

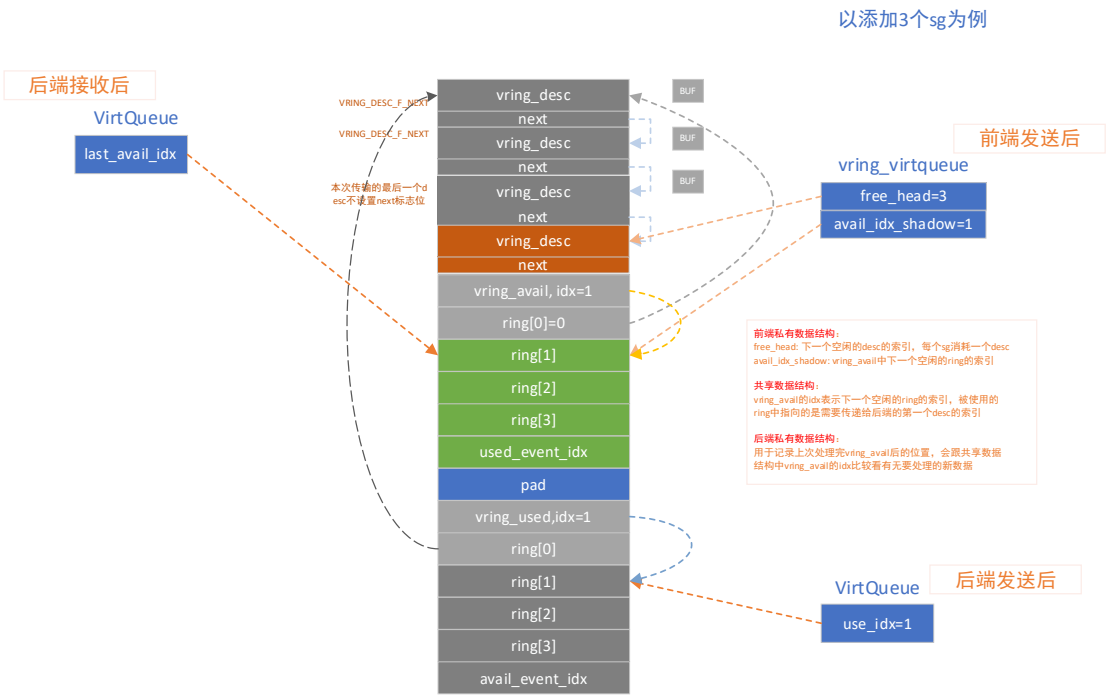
私有数据结构 VirtQueue 的 use_idx 记录了 vring_used 中空闲的 ring 的索引，初始值为 0

|-- 获取 use_idx，填充一个 VRingUsedElem 结构，也就是 vring_used 的 ring 的数据类型，其中记录了 vring_desc 的索引以及后端写入的数据长度，然后调用 vring_used_write 将创建的 VRingUsedElem 写到索引为 use_idx 的 ring 中。这里 VRingUsedElem 的 id 来自 **elem->index**，len 来自传入的 len。

|-- virtqueue_flush(vq, 1)

将 VirtQueue 中记录的 use_idx 加 1，表示 vring_used 中下一个空闲的 vring 的索引。

此时，数据结构变化如下：



7. 前端接收后端发送的数据

后端填充完 vring_used 后，会通知前端，然后前端去读取后端发送来的数据。

接着分析 linux 的 drivers\char\hw_random\virtio-rng.c。当后端通知前端后，会触发中断，在建立 virtqueue 的时候，会注册中断（drivers\virtio\virtio_mmio.c）：

```
446: static int vm_find_vqs(struct virtio_device *vdev, unsigned nvqs,
447:                       struct virtqueue *vqs[],
448:                       vq_callback_t *callbacks[],
449:                       const char * const names[],
450:                       const bool *ctx,
451:                       struct irq_affinity *desc)
452: {
453:     struct virtio_mmio_device *vm_dev = to_virtio_mmio_device(vdev);
454:     unsigned int irq = platform_get_irq(vm_dev->pdev, 0);
455:     int i, err;
456:
457:     err = request_irq(irq, vm_interrupt, IRQF_SHARED,
458:             dev_name(&vdev->dev), vm_dev);
459:     if (err)
460:         return err;
461:
462:     for (i = 0; i < nvqs; ++i) {
463:         vqs[i] = vm_setup_vq(vdev, i, callbacks[i], names[i],
464:                              ctx ? ctx[i] : false);
465:         if (IS_ERR(vqs[i])) {
466:             vm_del_vqs(vdev);
467:             return PTR_ERR(vqs[i]);
468:         }
469:     }
470:
471:     return 0;
472: } « end vm_find_vqs »
```

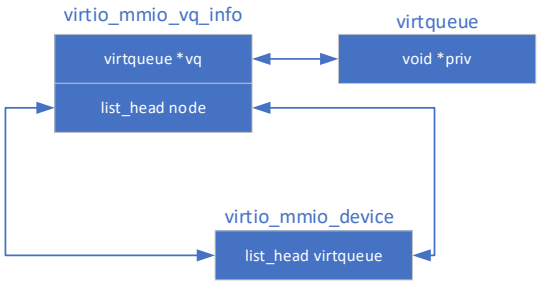
发生中断后，vm_interrupt 会被执行：

vm_interrupt

|-- 这个函数读取 VIRTIO_MMIO_INTERRUPT_STATUS，然后将读出的值写入 VIRTIO_MMIO_INTERRUPT_ACK 来清中断

|-- 如果上面读出的 status 里的 VIRTIO_MMIO_INT_CONFIG 被置位，表示后端的配置发生变化，会回调 virtio_driver 的 config_changed 回调函数

|-- 如果 status 的 VIRTIO_MMIO_INT_VRING 被置位，表示有数据要处理，接下来回遍历 virtio_mmio_device 下所有的 virtqueue，依次调用 vring_interrupt(irq, info->vq)



\-- vring_interrupt(irq, info->vq)

|-- 比较本地记录的 last_used_idx 跟共享内存中 vring_used 的 idx，不相等的话，表示确实有数据要处理，否则 return

|-- 回调 virtqueue 的 callback：vq->vq.callback(&vq->vq)，也就是 virtio_driver（drivers\char\hw_random\virtio-rng.c）中的 random_recv_done

\-- random_recv_done

\-- virtqueue_get_buf(vi->vq, &vi->data_avail)

读取数据。

\-- virtqueue_get_buf_ctx(_vq, len, NULL)

根据本地记录的 last_used_idx 去索引 vring.used->ring[last_used_idx]，获得后端刚才填写的 id 和 len，id 是 desc 的索引，len 是后端实际写入的数据长度，从这个索引指向的 desc 开始遍历，知道遇到当初自己没有设置 last_used_idx 的 desc 为止，然后将索引到的最后一个 desc 的 next 成员赋值为 vq->free_head，保持 desc 之间的连接关系，更新 vq->free_head 为索引都的第一个 desc 的编号，vq.num_free 增加索引到的 desc 的个数，num_free 表示空闲的 desc 个数，最后将 last_used_idx 加 1，表示 vring_used 中下一个空闲的 ring 的编号