作者：彭东林

邮箱：[pengdonglin137@163.com](mailto: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](#_Toc69054679)

[2. virtio device注册 4](#_Toc69054680)

[3. virtio driver的注册 6](#_Toc69054681)

[4. 前端向后端发送数据 11](#_Toc69054682)

[5. 后端接收前端发送过来的数据 13](#_Toc69054683)

[6. 后端向前端发送数据 15](#_Toc69054684)

[7. 前端接收后端发送的数据 16](#_Toc69054685)

以mmio类型的virtio为例进行分析。

# 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+ |  |  |  |

# 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";

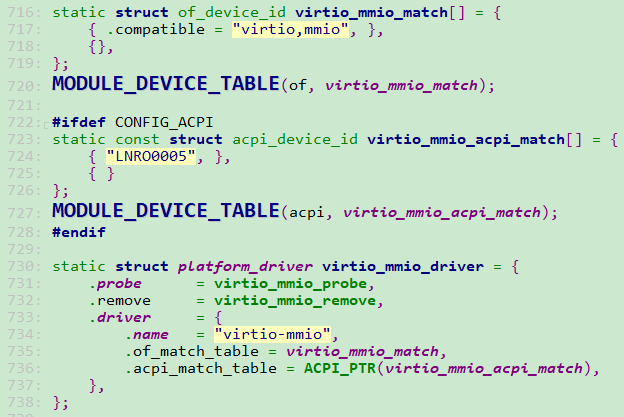
reg = <0x1e000 0x200>;

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。

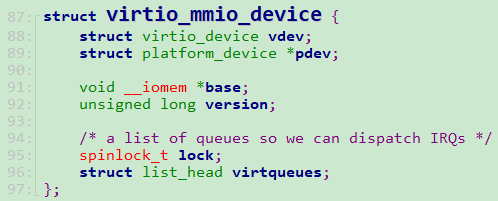
对应的内核驱动为：drivers\virtio\virtio\_mmio.c



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.

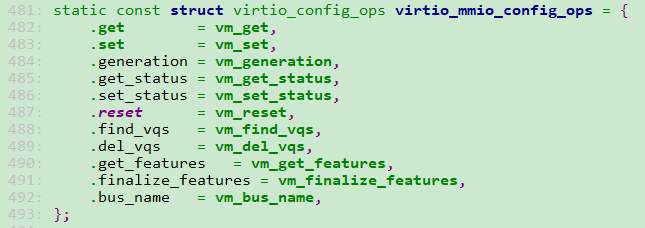


|-- vm\_dev->vdev.dev.parent = &pdev->dev

|-- vm\_dev->pdev = pdev

|-- vm\_dev->vdev.config = &**virtio\_mmio\_config\_ops**

这个ops后面会被virtio driver回调。



|-- 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

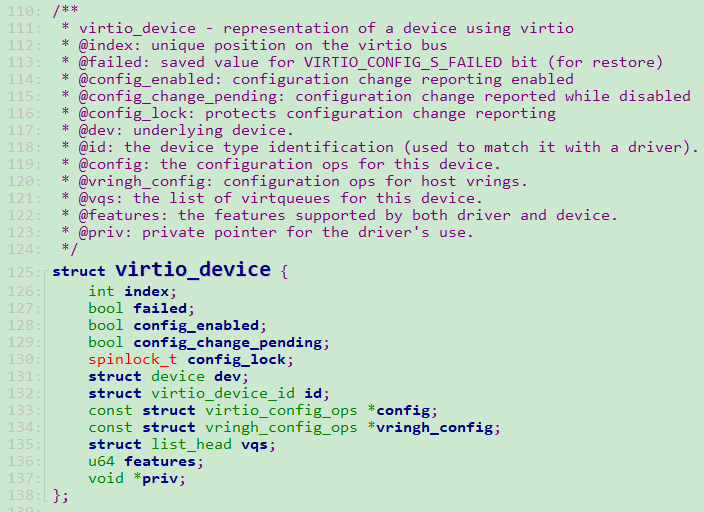
后面在利用这里的device id和vendor id去匹配对应virtio driver

|-- 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



|-- 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

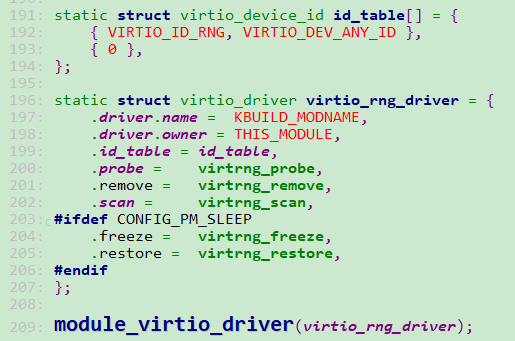
VIRTIO\_MMIO\_STATUS寄存器

|-- INIT\_LIST\_HEAD(&dev->vqs)

|-- device\_register(&dev->dev)

# 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。



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。

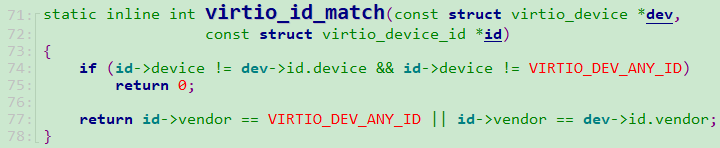
先看一下virio\_bus的match逻辑：

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：



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**



|-- 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

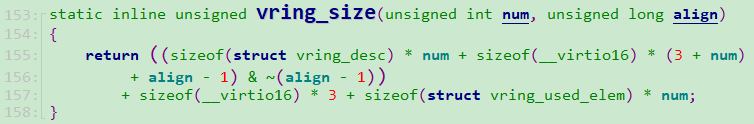
|-- 检查传入的元素最大个数是否为2的幂次方，不是的话，返回错误

|-- 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





\-- 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：



|-- 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

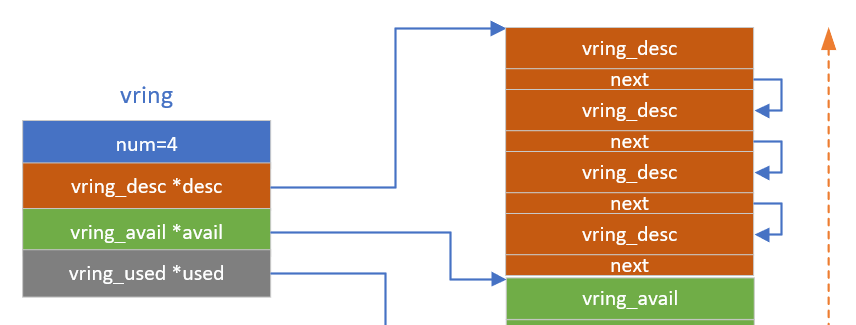
VRING\_AVAIL\_F\_NO\_INTERRUPT，表示前端没有callback可供回调，如果

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。

# 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++

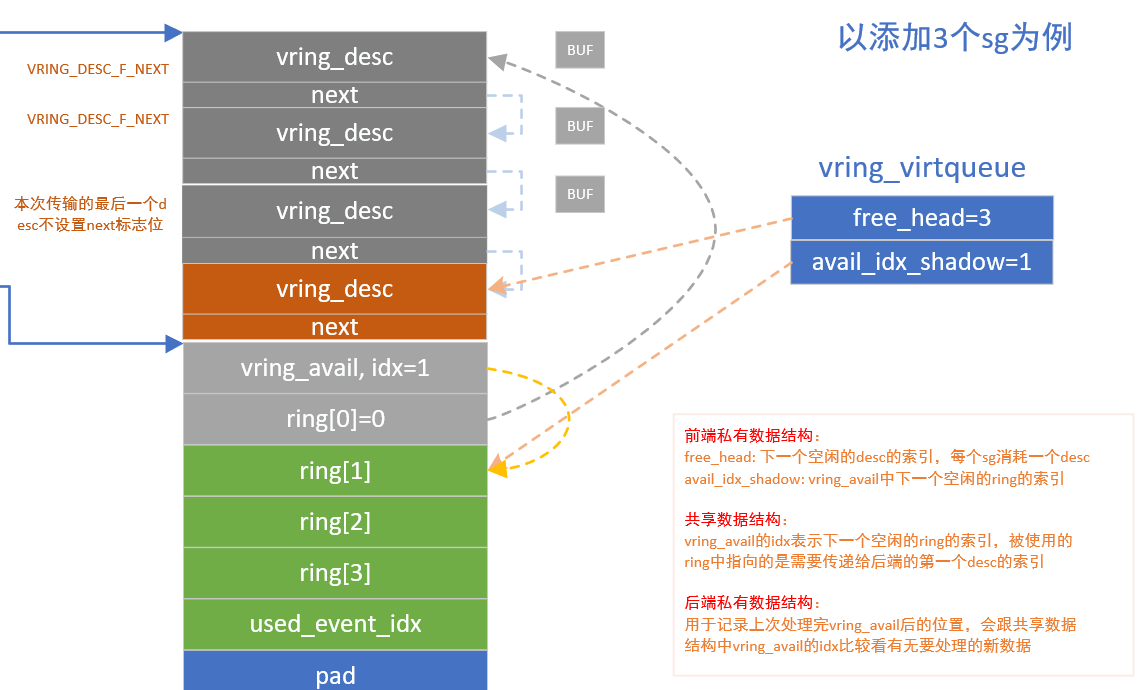
|-- 如果num\_added正好等于(1 << 16) – 1，调用virtqueue\_kick(\_vq)，

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

# 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



# 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：



# 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）：



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

