AF_XDP 协议

1. 简介

AF_XDP是一个协议族（例如AF_NET），主要用于高性能报文处理。

通过XDP_REDIRECT可以将报文重定向到其他设备发送出去或者重定向到其他的CPU继续进行处理。而AF_XDP则利用 bpf_redirect_map()函数，实现将报文重定向到用户态一块指定的内存中。

使用普通的 socket() 系统调用创建一个AF_XDP套接字（XSK）。每个XSK都有两个ring：

- RX RING
- TX RING

套接字可以在 RX RING 上接收数据包，并且可以在 TX RING 环上发送数据包。

这些环分别通过 setockopts() 的 XDP_RX_RING 和 XDP_TX_RING 进行注册和调整大小。每个 socket 必须至少有一个这样的环。

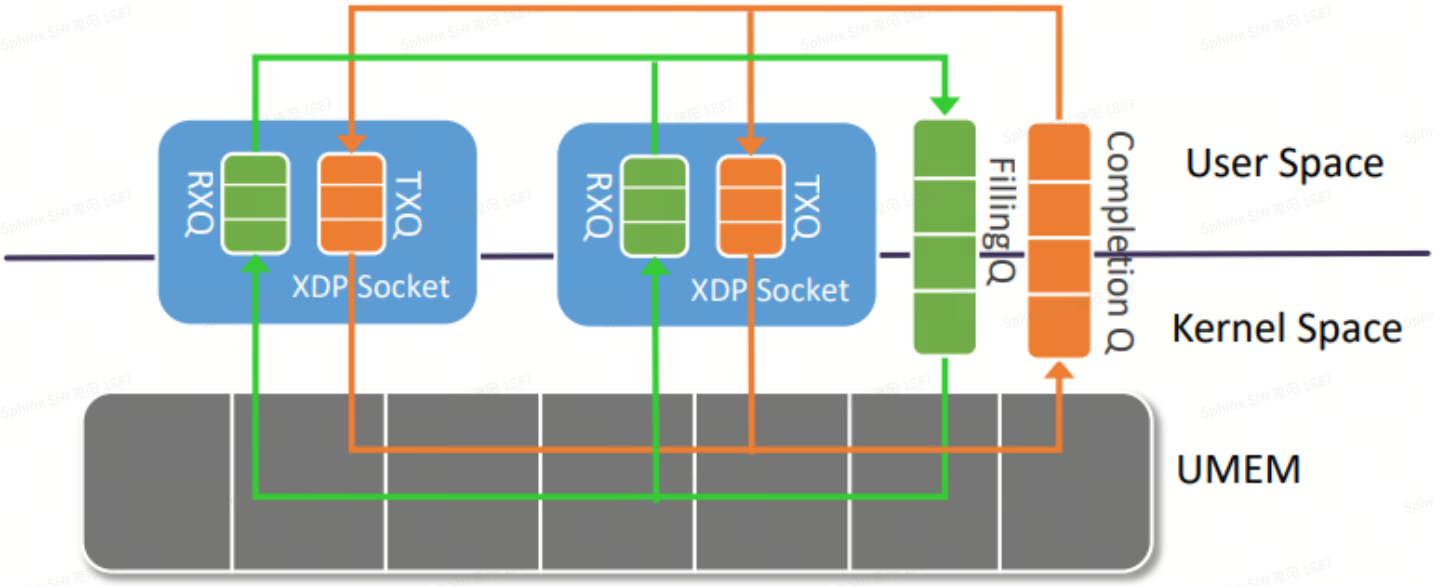
RX或TX描述符环指向存储区域（称为UMEM）中的数据缓冲区。RX和TX可以共享同一UMEM，因此不必在RX和TX之间复制数据包。

1.1 UMEM

UMEM有两个 ring：

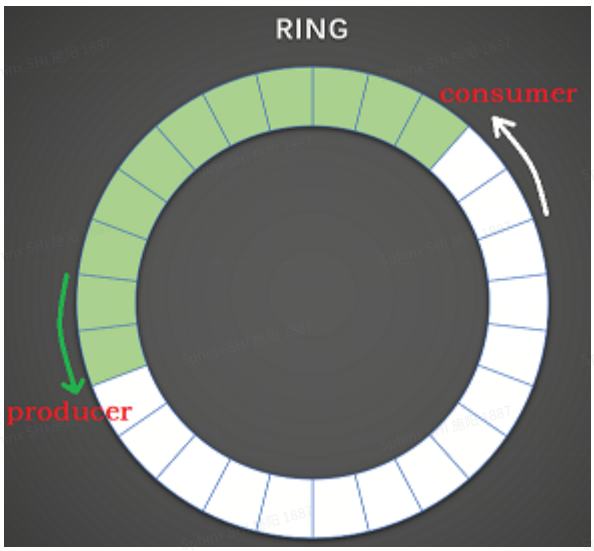
- FILL RING
- COMPLETION RING

应用程序使用 FILL RING 向内核发送可以承载报文的 addr (该 addr 指向UMEM中某个chunk)，以供内核填充RX数据包数据。每当收到数据包，对这些 chunks 的引用就会出现在RX环中。另一方面，COMPLETION RING包含内核已完全传输的 chunks 地址，可以由用户空间再次用于 TX 或 RX。



1.2 Ring

ring是一个固定长度的数组，并且同时拥有一个生产者和一个消费者，生产者向数组中逐个填写数据，消费者从数组中逐个读取生产者填充的数据，生产者和消费者都用数组的下标表示，不断累加，像一个环一样不断重复生产然后消费的动作，因此得名ring。



AF_XDP socket不再通过 send()/recv()等函数实现报文收发，而是通过直接操作ring来实现报文收发。

1. FILL RING

fill_ring 的生产者是用户态程序，消费者是内核态中的XDP程序；

用户态程序通过 fill_ring 将可以用来承载报文的 UMEM frames 传到内核，然后内核消耗 fill_ring 中的元素（后文统一称为 desc），并将报文拷贝到desc中指定地址（该地址即UMEM frame的地址）；

2. COMPLETION RING

completion_ring 的生产者是XDP程序，消费者是用户态程序；

当内核完成XDP报文的发送，会通过 completion_ring 来通知用户态程序，哪些报文已经成功发送，然后用户态程序消耗 completion_ring 中 desc(只是更新consumer计数相当于确认)；

3. RX RING

rx_ring的生产者是XDP程序，消费者是用户态程序；

XDP程序消耗 fill_ring，获取可以承载报文的 desc并将报文拷贝到desc中指定的地址，然后将desc填充到 rx_ring 中，并通过socket IO机制通知用户态程序从 rx_ring 中接收报文；

4. TX RING

tx_ring的生产者是用户态程序，消费者是XDP程序；

用户态程序将要发送的报文拷贝 tx_ring 中 desc指定的地址中，然后 XDP程序 消耗 tx_ring 中的 desc，将报文发送出去，并通过 completion_ring 将成功发送的报文的desc告诉用户态程序；

2. Code

2.1 UMEM

2.1.1 创建AF_XDP的socket

代码块

```
xsk_fd = socket(AF_XDP, SOCK_RAW, 0);
```

2.1.2 为UMEM申请内存

上文提到UMEM是一块包含固定大小chunk的内存，可以通过malloc/mmap/hugepages申请。

下文大部分代码出自kernel samples。

代码块

```
bufs = mmap(NULL, NUM_FRAMES * opt_xsk_frame_size,
            PROT_READ | PROT_WRITE,
            MAP_PRIVATE | MAP_ANONYMOUS | opt_mmap_flags, -1, 0);

if (bufs == MAP_FAILED) {
    printf("ERROR: mmap failed\n");
    exit(EXIT_FAILURE);
}
```

2.1.3 向AF_XDP socket注册UMEM

代码块

```
struct xdp_umem_reg mr;memset(&mr, 0, sizeof(mr));
mr.addr = (uintptr_t)umem_area; // umem_area即上面通过mmap申请到内存起始地址
```

```
mr.len = size;
mr.chunk_size = umem->config.frame_size;
mr.headroom = umem->config.frame_headroom;
mr.flags = umem->config.flags;
err = setsockopt(umem->fd, SOL_XDP, XDP_UMEM_REG, &mr, sizeof(mr));
if (err) {
    err = -errno;
    goto out_socket;
}
```

其中xdp_umem_reg结构定义在 usr/include/linux/if_xdp.h中：

代码块

```
struct xdp_umem_reg {
    __u64 addr; /* Start of packet data area */
    __u64 len; /* Length of packet data area */
    __u32 chunk_size;
    __u32 headroom;
    __u32 flags;
};
```

成员解析：

- addr就是UMEM内存的起始地址；
- len是整个UMEM内存的总长度；
- chunk_size就是每个chunk的大小；
- headroom，如果设置了，那么报文数据将不是从每个chunk的起始地址开始存储，而是要预留出headroom大小的内存，再开始存储报文数据，headroom在隧道网络中非常常见，方便封装外层头部；
- flags, UMEM还有一些更复杂的用法，通过flag设置，后面再进一步展开；

2.2 Ring

通过 setsockopt() 设置 FILL/COMPLETION/RX/TX ring的大小

> 这个过程相当于创建，不设置大小的ring是没有办法使用的

FILL RING 和 COMPLETION RING是UMEM必须，RX和TX则是 AF_XDP socket二选一的，例如AF_XDP socket只收包那么只需要设置RX RING的大小即可。

代码块

```
err = setsockopt(umem->fd, SOL_XDP, XDP_UMEM_FILL_RING,
```

```
                &umem->config.fill_size,
                sizeof(umem->config.fill_size));
    if (err) {
        err = -errno;
        goto out_socket;
    }
    err = setsockopt(umem->fd, SOL_XDP, XDP_UMEM_COMPLETION_RING,
                &umem->config.comp_size,
                sizeof(umem->config.comp_size));
    if (err) {
        err = -errno;
        goto out_socket;
    }
```

上述操作相当于创建了 FILL RING 和 和 COMPLETION RING，创建ring的过程主要是初始化 producer 和 consumer 的下标，以及创建ring数组。

问题来了：

上文提到，用户态程序是 FILL RING 的生产者和 CONPLETION RING 的消费者，上面2个 ring 的创建是在内核中创建了 ring 并初始化了其相关成员。那么用户态程序如何操作这两个位于内核中的 ring 呢？所以接下来我们需要将整个 ring 映射到用户态空间。

2.2.1 FILL RING

第一步是获取内核中ring结构各成员的偏移，因为从5.4版本开始后，ring结构中除了 producer、consumer、desc外，又新增了一个flag成员。

所以用户态程序需要先获取 ring 结构中各成员的准确便宜，才能在mmap() 之后准确识别内存中各成员位置。

代码块

```
    err = xsk_get_mmap_offsets(umem->fd, &off);
    if (err) {
        err = -errno;
        goto out_socket;
    }
```

xsk_get_mmap_offsets() 函数主要是通过getsockopt函数实现这一功能：

代码块

```
    err = getsockopt(fd, SOL_XDP, XDP_MMAP_OFFSETS, off, &optlen);if (err)return
    err;
```

一切就绪，开始将内核中的 FILL RING 映射到用户态程序中：

代码块

```
map = mmap(NULL, off.fr.desc + umem->config.fill_size * sizeof(__u64),
           PROT_READ | PROT_WRITE, MAP_SHARED | MAP_POPULATE, umem->fd,
           XDP_UMEM_PGOFF_FILL_RING);if (map == MAP_FAILED) {
        err = -errno;
        goto out_socket;
}
umem->fill = fill;
fill->mask = umem->config.fill_size - 1;
fill->size = umem->config.fill_size;
fill->producer = map + off.fr.producer;
fill->consumer = map + off.fr.consumer;
fill->flags = map + off.fr.flags;
fill->ring = map + off.fr.desc;
fill->cached_cons = umem->config.fill_size;
```

上面代码需要关注的一点是 mmap() 函数中指定内存的长度——**off.fr.desc + umem->config.fill_size * sizeof(__u64)**，umem->config.fill_size * sizeof(__u64)没什么好说的，就是ring数组的长度，而 off.fr.desc 则是ring结构体的长度，我们先看下内核中ring结构的定义：

代码块

```
struct xdp_ring_offset {
        __u64 producer;
        __u64 consumer;
        __u64 desc;
};
```

这是没有flag的定义，无伤大雅。这里desc的地址其实就是ring数组的起始地址了。而off.fr.desc是desc相对 ring 结构体起始地址的偏移，相当于结构体长度。

用一张图来看下ring所在内存的结构分布：

umem->fill 是用户态程序自定义的一个结构体，其成员 producer、consumer、flags、ring都是指针，分别指向实际ring结构中的对应成员，umem->fill中的其他成员主要在后面报文收发时用到，起辅助作用。

2.2.2 COMPLETION RING

跟上面 FILL RING 的映射一样

代码块

```
map = mmap(NULL, off.cr.desc + umem->config.comp_size * sizeof(__u64),
                    PROT_READ | PROT_WRITE, MAP_SHARED | MAP_POPULATE, umem->fd,
                    XDP_UMEM_PGOFF_COMPLETION_RING);
if (map == MAP_FAILED) {
        err = -errno;
        goto out_mmap;
}

umem->comp = comp;
comp->mask = umem->config.comp_size - 1;
comp->size = umem->config.comp_size;
comp->producer = map + off.cr.producer;
comp->consumer = map + off.cr.consumer;
comp->flags = map + off.cr.flags;
comp->ring = map + off.cr.desc;
```

2.2.3 RX/TX Ring

这里和 FILL RING 以及 COMPLETION RING的做法基本完全一致：

代码块

```
if (rx) {
        err = setsockopt(xsk->fd, SOL_XDP, XDP_RX_RING,
                         &xsk->config.rx_size,sizeof(xsk->config.rx_size));if
(err) {
                err = -errno;
                goto out_socket;
        }
}if (tx) {
        err = setsockopt(xsk->fd, SOL_XDP, XDP_TX_RING,
                         &xsk->config.tx_size,sizeof(xsk->config.tx_size));if
(err) {
                err = -errno;
                goto out_socket;
        }
}
err = xsk_get_mmap_offsets(xsk->fd, &off);if (err) {
        err = -errno;
        goto out_socket;
}
if (rx) {
        rx_map = mmap(NULL, off.rx.desc +
                      xsk->config.rx_size * sizeof(struct xdp_desc),
                      PROT_READ | PROT_WRITE, MAP_SHARED | MAP_POPULATE,
```

```
                        xsk->fd, XDP_PGOFF_RX_RING);if (rx_map == MAP_FAILED) {
                err = -errno;
                goto out_socket;
        }
        rx->mask = xsk->config.rx_size - 1;
        rx->size = xsk->config.rx_size;
        rx->producer = rx_map + off.rx.producer;
        rx->consumer = rx_map + off.rx.consumer;
        rx->flags = rx_map + off.rx.flags;
        rx->ring = rx_map + off.rx.desc;
}
xsk->rx = rx;
if (tx) {
        tx_map = mmap(NULL, off.tx.desc +
                        xsk->config.tx_size * sizeof(struct xdp_desc),
                        PROT_READ | PROT_WRITE, MAP_SHARED | MAP_POPULATE,
                        xsk->fd, XDP_PGOFF_TX_RING);if (tx_map == MAP_FAILED) {
                err = -errno;
                goto out_mmap_rx;
        }
        tx->mask = xsk->config.tx_size - 1;
        tx->size = xsk->config.tx_size;
        tx->producer = tx_map + off.tx.producer;
        tx->consumer = tx_map + off.tx.consumer;
        tx->flags = tx_map + off.tx.flags;
        tx->ring = tx_map + off.tx.desc;
        tx->cached_cons = xsk->config.tx_size;
}
xsk->tx = tx;
```

调用bind()将AF_XDP socket绑定的指定设备的某一队列

代码块

```
        sxdp.sxdp_family = PF_XDP;
        sxdp.sxdp_ifindex = xsk->ifindex;
        sxdp.sxdp_queue_id = xsk->queue_id;
        sxdp.sxdp_flags = xsk->config.bind_flags;
        err = bind(xsk->fd, (struct sockaddr *)&sxdp, sizeof(sxdp));if (err) {
                err = -errno;
                goto out_mmap_tx;
        }
```

2.3 内核态

相比用户态程序的一堆操作，内核态XDP程序看起来要简单的多。

XDP程序利用 bpf_reditrct() 函数可以将报文重定向到其他设备发送出去或者重定向到其他CPU继续处理，后来又发展出了bpf_redirect_map()函数，可以将重定向的目的地保存在map中。AF_XDP 正是利用了 bpf_redirect_map() 函数以及 BPF_MAP_TYPE_XSKMAP 类型的 map 实现将报文重定向到用户态程序。

2.1 创建BPF_MAP_TYPE_XSKMAP类型的map

该类型map的key是网口设备的queue_id，value则是该queue上绑定的AF_XDP socket fd，所以通常需要为每个网口设备各自创建独立的map，并在用户态将对应的queue_id->xsk_fd存储到map中。

代码块

```
static int xsk_create_bpf_maps(struct xsk_socket *xsk)
{int max_queues;int fd;
        max_queues = xsk_get_max_queues(xsk);if (max_queues < 0)return
max_queues;
        fd = bpf_create_map_name(BPF_MAP_TYPE_XSKMAP, "xsks_map",sizeof(int),
sizeof(int), max_queues, 0);if (fd < 0)return fd;
        xsk->xsks_map_fd = fd;
return 0;
}
```

bpf_create_map_name参数详解：

- BPF_MAP_TYPE_XSKMAP，map类型
- "xsks_map"，map的名字
- sizeof(int)，分别指定key和vlue的size
- max_queues，map大小
- 0, map_flags

2.2 XDP程序代码

代码块

```
/* This is the C-program:
 * SEC("xdp_sock") int xdp_sock_prog(struct xdp_md *ctx)
 * {
 *     int index = ctx->rx_queue_index;
 *
 *     // A set entry here means that the correspnding queue_id
 *     // has an active AF_XDP socket bound to it.
 *     if (bpf_map_lookup_elem(&xsks_map, &index))
 *         return bpf_redirect_map(&xsks_map, index, 0);
```

```
 *
 *     return XDP_PASS;
 * }
 */
```

是不是非常的简单，真正的redirect操作只有一行代码。

2.3 XDP程序的加载

代码块

```
static int xsk_load_xdp_prog(struct xsk_socket *xsk)
{
    static const int log_buf_size = 16 * 1024;
    char log_buf[log_buf_size];
    int err, prog_fd;
    /* This is the C-program:
    * SEC("xdp_sock") int xdp_sock_prog(struct xdp_md *ctx)
    * {
    *     int index = ctx->rx_queue_index;
    *
    *     // A set entry here means that the correspnding queue_id
    *     // has an active AF_XDP socket bound to it.
    *     if (bpf_map_lookup_elem(&xsks_map, &index))
    *         return bpf_redirect_map(&xsks_map, index, 0);
    *
    *     return XDP_PASS;
    * }
    */
    struct bpf_insn prog[] = {/* r1 = *(u32 *)(r1 + 16) */
            BPF_LDX_MEM(BPF_W, BPF_REG_1, BPF_REG_1, 16),/* *(u32 *)(r10 - 4)
= r1 */
            BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_1, -4),
            BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
            BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),
            BPF_LD_MAP_FD(BPF_REG_1, xsk->xsks_map_fd),
            BPF_EMIT_CALL(BPF_FUNC_map_lookup_elem),
            BPF_MOV64_REG(BPF_REG_1, BPF_REG_0),
            BPF_MOV32_IMM(BPF_REG_0, 2),/* if r1 == 0 goto +5 */
            BPF_JMP_IMM(BPF_JEQ, BPF_REG_1, 0, 5),/* r2 = *(u32 *)(r10 - 4) */
            BPF_LD_MAP_FD(BPF_REG_1, xsk->xsks_map_fd),
            BPF_LDX_MEM(BPF_W, BPF_REG_2, BPF_REG_10, -4),
            BPF_MOV32_IMM(BPF_REG_3, 0),
            BPF_EMIT_CALL(BPF_FUNC_redirect_map),/* The jumps are to this
instruction */
            BPF_EXIT_INSN(),
```

```
34      };
35
36      size_t insns_cnt = sizeof(prog) / sizeof(struct bpf_insn);
37      prog_fd = bpf_load_program(BPF_PROG_TYPE_XDP, prog, insns_cnt,
38                                 "LGPL-2.1 or BSD-2-Clause", 0, log_buf,
39                                   log_buf_size);
40      if (prog_fd < 0) {
41              pr_warning("BPF log buffer:\n%s", log_buf);return prog_fd;
42      }
43      err = bpf_set_link_xdp_fd(xsk->ifindex, prog_fd, xsk->config.xdp_flags);
44      if (err) {
45              close(prog_fd);
46              return err;
47      }
48      xsk->prog_fd = prog_fd;
49
50      return 0;
51  }
```

XDP程序的load

调用函数 bpf_load_program() 之前的代码不用关心。

通常 eBPF 程序使用 C 语言的一个子集（restricted C）编写，然后通过 LLVM 编译成字节码注入到内核执行。

由于本例中XDP程序代码比较简单，功力深厚的作者直接将其编写为 eBPF（JIT）可识别的字节码，然后直接调用 bpf_load_program() 函数将字节码程序加载到内核中。

XDP程序的attach

XDP程序加载成功会返回对应的fd（后面统称为prog_fd），但是此时XDP程序还不会被执行（所有的eBPF都需要经过load和attach两步才能被触发执行，load只是将程序加载到内核中，attach将程序添加到hook点后，程序才能真正被触发执行）。

调用函数 bpf_set_link_xdp_fd() 函数将XDP程序attach到指定网口设备的驱动中的hook点。

> **注意：** AF_XDP socket是跟指定网口设备的队列绑定，而XDP程序则是跟指定的网口设备绑定（attach）。

2.4 用户态

经过前面两步，AF_XDP socket、UMEM、FILL/COMPLETION/RX/TX RING 都创建设置好了，XSKMAP 和XDP PROG 也都加载好了。但是要想让XDP程序把报文传到用户态程序，还得再进行两步操作。

2.4.1 将AF_XDP socket存储到XSKMAP中

代码块

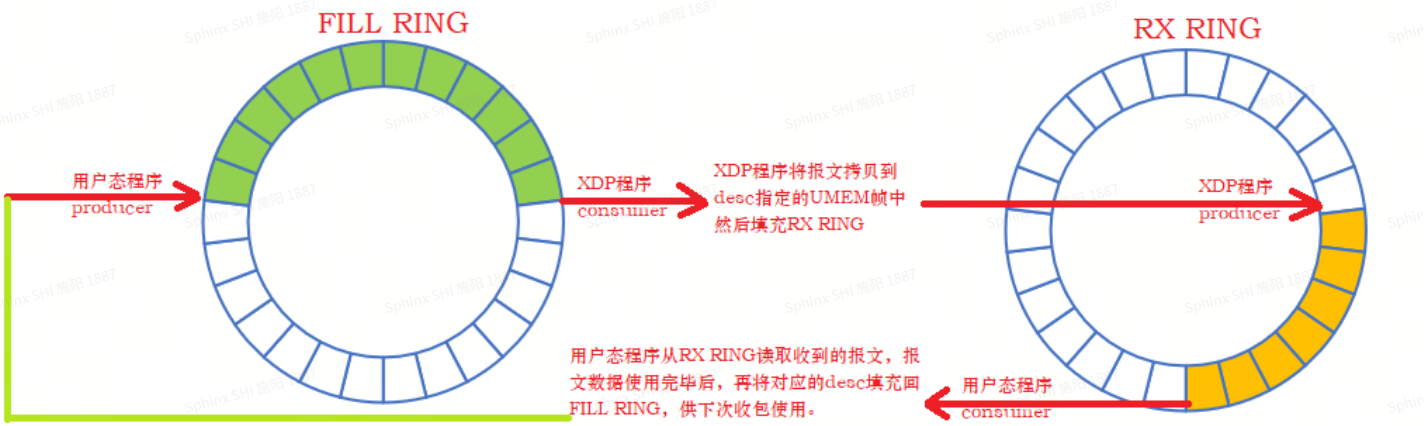
```
static int xsk_set_bpf_maps(struct xsk_socket *xsk)
{
    return bpf_map_update_elem(xsk->xsks_map_fd, &xsk->queue_id,
                               &xsk->fd, 0);
}
```

2.4.2 Ring

前面介绍过4种ring，分别对应收发包两个场景

- 收包：
  - FILL
  - RX ring，
- 发包：
  - TX
  - COMPLETION RING

2.4.2.1 收包



收包过程是由XDP程序触发的，但是XDP程序收包，需要依赖用户态程序填充FILL RING，将可以承载报文的desc告诉XDP程序。所以在用户态程序初始化阶段，需要先填充FILL RING，直接看代码：

代码块

```

ret = xsk_ring_prod__reserve(&xsk->umem->fq,
                             XSK_RING_PROD__DEFAULT_NUM_DESCS,
                             &idx);
if (ret != XSK_RING_PROD__DEFAULT_NUM_DESCS)
        exit_with_error(-ret);
```

```
7   for (i = 0; i < XSK_RING_PROD__DEFAULT_NUM_DESCS; i++)
8           *xsk_ring_prod__fill_addr(&xsk->umem->fq, idx++) =
9                   i * opt_xsk_frame_size;
10  xsk_ring_prod__submit(&xsk->umem->fq,
11                        XSK_RING_PROD__DEFAULT_NUM_DESCS);
```

三个经过封装的函数

1. **xsk_ring_prod__reserve**

代码块

```
1   static inline size_t xsk_ring_prod__reserve(struct xsk_ring_prod *prod,size_t
    nb, __u32 *idx){if (xsk_prod_nb_free(prod, nb) < nb)return 0;
2           *idx = prod->cached_prod;
3           prod->cached_prod += nb;
4   return nb;
5   }
```

这个函数前面先判断一下：现在想生产nb个数据，ring里有没有足够的地方放啊？没有的话直接退出，等会再试试。

> vhostuser里再这块有个BUG，前端程序想发包发现ring里空间不够了，而后端驱动处理又由于有有问题的判断，导致报文已发的报文一直不被处理，结果造成死锁。

如果有足够的空间，那么会将生产者当前下标（cached_prog）赋值给idx，因为退出函数后会根据从这个idx指向的位置开始生产desc，最后cached_prod + nb。

为什么要有个cached_prog呢?

因为生产数据这个过程需要分几步完成，所以这个应该为了多线程同步吧。

2. **xsk_ring_prod__fill_addr**

代码块

```
1   static inline __u64 *xsk_ring_prod__fill_addr(struct xsk_ring_prod *fill,
2                                                 __u32 idx)
3   {
4       __u64 *addrs = (__u64 *)fill->ring;
5       return &addrs[idx & fill->mask];
6   }
```

看这段代码前，我们先看下ring中元素xdp_desc的成员结构：

代码块

```
struct xdp_desc {
        __u64 addr;
        __u32 len;
        __u32 options;
};
```

成员解析

- addr指向UMEM中某个帧的具体位置，并且不是真正的虚拟内存地址，而是相对UMEM内存起始地址的偏移。
- len则是指报文的具体的长度，当XDP程序向desc填充报文的时候需要设置len，但是用户态程序向FILL RING中填充desc则不用关心len。

所以上面xsk_ring_prod__fill_addr的功能就好理解了，返回的ring中下标为idx处的desc中addr的指针；并且在函数返回后对addr进行了赋值，再看下这块代码，可以看到赋值给addr是个偏移量：

代码块

```
for (i = 0; i < XSK_RING_PROD__DEFAULT_NUM_DESCS; i++)
    *xsk_ring_prod__fill_addr(&xsk->umem->fq, idx++) = i * opt_xsk_frame_size;
```

3. xsk_ring_prod__submit

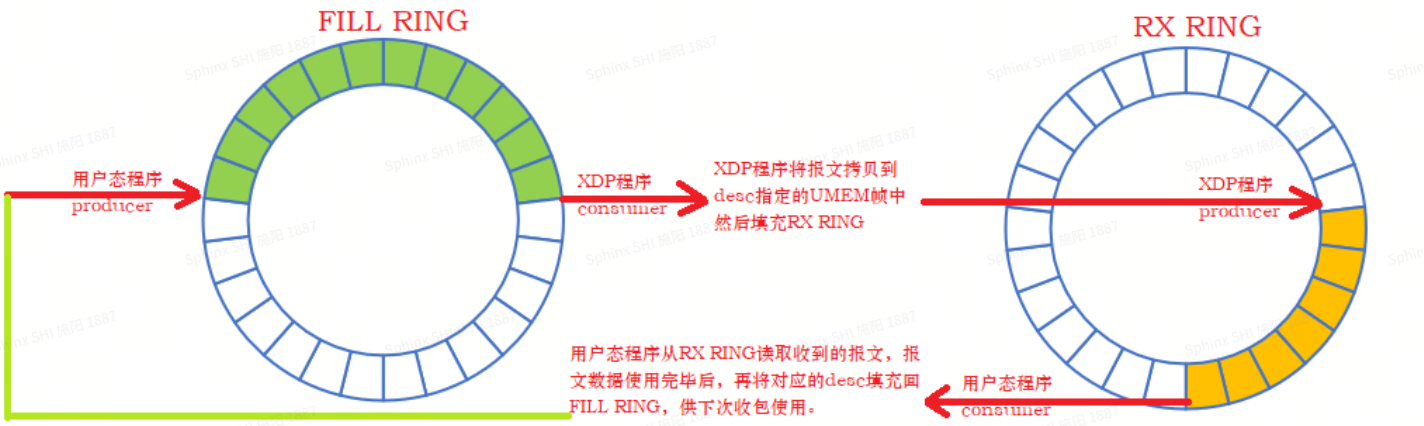
代码块

```
static inline void xsk_ring_prod__submit(struct xsk_ring_prod *prod, size_t nb)
{/* Make sure everything has been written to the ring before indicating
        * this to the kernel by writing the producer pointer.
        */libbpf_smp_wmb();
        *prod->producer += nb;
}
```

数据填充完毕，更新生产者下标。

> 说明：下标永远指向下一个可填充数据位置。

收包流程解析



AF_XDP socket毕竟也是socket，所以select/poll/epoll这些函数都能用的。

看具体从一个AF_XDP socket收包的过程:

代码块

```
1  static void rx_drop(struct xsk_socket_info *xsk, struct pollfd *fds)
2  {
3          unsigned int rcvd, i;u32 idx_rx = 0, idx_fq = 0;
4          int ret;
5          rcvd = xsk_ring_cons__peek(&xsk->rx, BATCH_SIZE, &idx_rx);if (!rcvd)
   {if (xsk_ring_prod__needs_wakeup(&xsk->umem->fq))
6                          ret = poll(fds, num_socks, opt_timeout);return;
7          }
8          ret = xsk_ring_prod__reserve(&xsk->umem->fq, rcvd, &idx_fq);while (ret
   != rcvd) {if (ret < 0)exit_with_error(-ret);if
   (xsk_ring_prod__needs_wakeup(&xsk->umem->fq))
9                          ret = poll(fds, num_socks, opt_timeout);
10                 ret = xsk_ring_prod__reserve(&xsk->umem->fq, rcvd, &idx_fq);
11         }
12 for (i = 0; i < rcvd; i++) {u64 addr = xsk_ring_cons__rx_desc(&xsk->rx,
   idx_rx)->addr;u32 len = xsk_ring_cons__rx_desc(&xsk->rx, idx_rx++)->len;u64
   orig = xsk_umem__extract_addr(addr);
13                 addr = xsk_umem__add_offset_to_addr(addr);char *pkt =
   xsk_umem__get_data(xsk->umem->buffer, addr);
14 hex_dump(pkt, len, addr);
15                 *xsk_ring_prod__fill_addr(&xsk->umem->fq, idx_fq++) = orig;
16         }
17 xsk_ring_prod__submit(&xsk->umem->fq, rcvd);xsk_ring_cons__release(&xsk->rx,
   rcvd);
18         xsk->rx_npkts += rcvd;
19 }
```

该函数并没有对报文做什么复杂处理，只是hex_dump了一下，整个收发包分五个步骤：

1. **xsk_ring_cons__peek()**

开始对RX RING进行消费，返回消费者下标和消费个数，并累加cached_cons；

2. **xsk_ring_prod__reserve**

开始对FILL RING进行生产，返回生产者下标和生产个数，并累加cached_prod;

3. **报文处理**

处理从RX RING中收到的报文，并回填到FILL RING中；

代码块

```
1           for (i = 0; i < rcvd; i++) {u64 addr = xsk_ring_cons__rx_desc(&xsk-
    >rx, idx_rx)->addr;u32 len = xsk_ring_cons__rx_desc(&xsk->rx, idx_rx++)-
    >len;u64 orig = xsk_umem__extract_addr(addr);
2                   addr = xsk_umem__add_offset_to_addr(addr);char *pkt =
    xsk_umem__get_data(xsk->umem->buffer, addr);
3   hex_dump(pkt, len, addr);
4                   *xsk_ring_prod__fill_addr(&xsk->umem->fq, idx_fq++) = orig;
5           }
```

从desc中读取addr，并通过 xsk_umem__get_data() 函数得到报文真正的虚拟地址，然后hex_dump()下。

代码块

```
1   static inline void *xsk_umem__get_data(void *umem_area, __u64 addr){return &
    ((char *)umem_area)[addr];
2   }
```

然后将处理完报文所在的 UMEM 帧回填到FILL RING中：

代码块

```
1   *xsk_ring_prod__fill_addr(&xsk->umem->fq, idx_fq++) = orig;
```

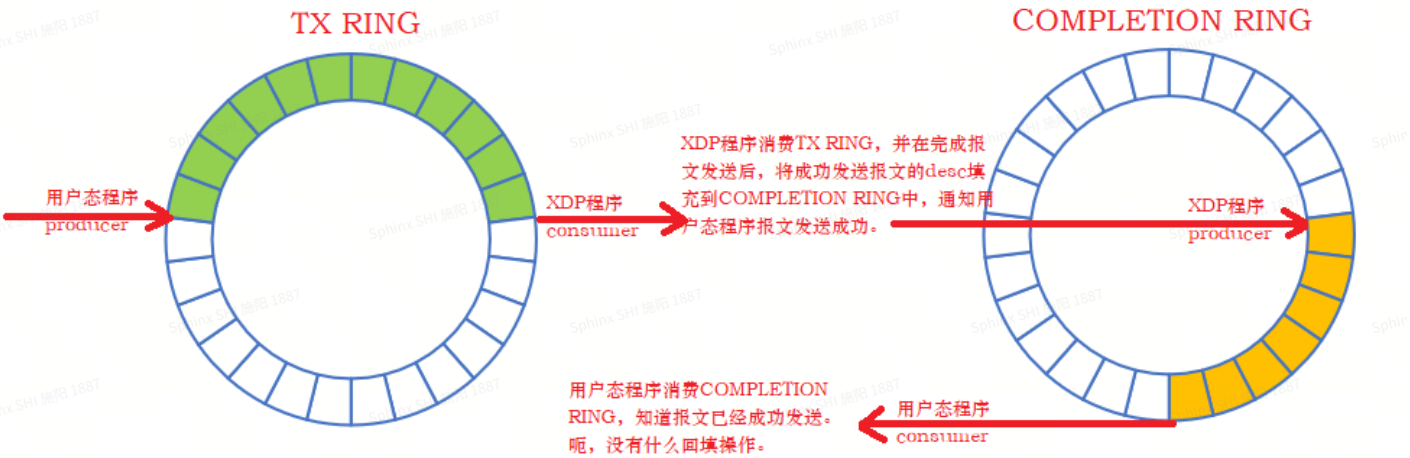
4. **xsk_ring_prod__submit(&xsk->umem->fq, rcvd)**

完成对RX RING的消费，更新消费者下标；

5. **xsk_ring_cons__release(&xsk->rx, rcvd)**

完成对FILL RING的生产，更新生产者下标；

2.4.2.2 发包



发包真的没啥好说的。初始化的时候不用管，想发包的时候直接就发啦。

Refs

https://colobu.com/2023/04/17/use-af-xdp-socket/

https://rexrock.github.io/post/xdp1/