理解 Linux Kernel（8）— 网络*

防风

January 15, 2019

1 源起

虽然“UNIX 一切皆文件”的理念已经被广泛地接受，但是针对网络，似乎无论如何都无法将其全盘纳入到“文件”的概念下。最简单的例子，块设备、字符设备等等都被 Linux 内核视作是一个个独立的文件，并在 *proc* 文件系统提供了一个统一的展示窗口。可对于网络设备，很显然没有这方面的处理。

连内核都把网络设备单独拎出来而无法整合到**文件**里，可想而知，从学习和理解上就不得不另起炉灶而无法借助于现有的内核关于“文件”的实现啦。当然，话也没有那么绝对，不论如何，最后在用户态使用过程中，我们看到以及操作的总还是一个抽象的文件。比如 *recvfrom* 和 *sendto* 函数还都是以文件描述符作为第一入参。而且，真正使得**网络设备**无法被表示成**文件**的原因之一：就是多层次的网络模型使用了多种多样的通信协议，并且无法在打开文件的第一时间确定建立连接的所有选项。

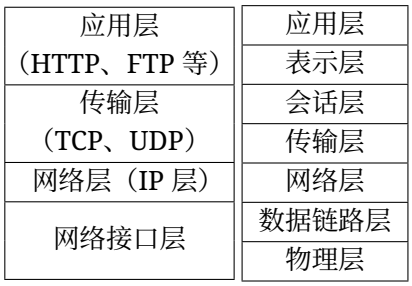


图 1: TCP/IP 网络模型 & ISO/OSI 网络模型

对于网络，我们接触最多的就是如图 1所展示的网络模型了。不论是 4 层模型还是 7 层模型，核心的总不过是每层都由完全分离的代码来实现，只通过明确约定的接口进行紧邻上层/下层之间的交互。那么用户态的程序将如何借用图 1的模型来完成网络信息交换呢？这又得引出**套接字**了。

套接字最早出现在 BSD UNIX，随后被 POSIX 收录，由此其事实上成为了类 UNIX 操作系统关于网络实现方面的标准。一般来说，我们使用套接字与使

*鉴于 Linux0.11 版本没有关于网络的相关实现，本篇所述内容将以 Linux2.6.24 版本为基准

用普通文件读写在流程上大体类似。图 2展示了在 Linux 系统下客户端与服务端使用套接字做交互的相关步骤，看着似乎与操作正规文件有很大的区别。但如果摒弃监听以及建立连接的流程，recv 与 send 就分别等同于文件读写中的 read 和 write。

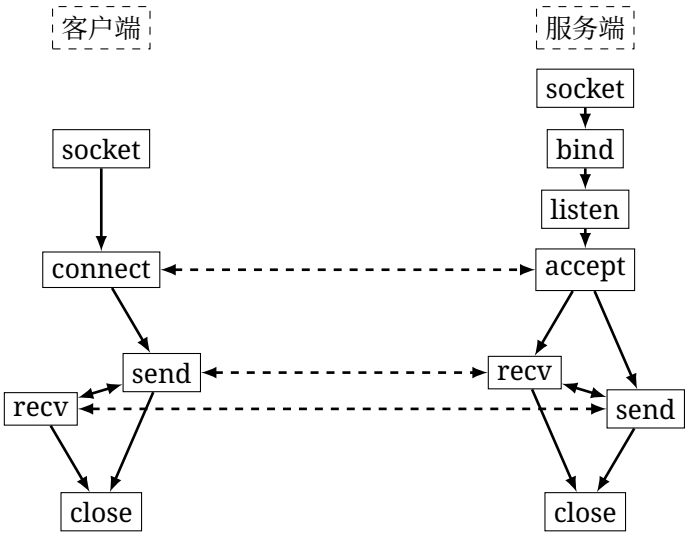


图 2: 套接字执行流

不论如何，就姑且认为各位读者都对套接字的使用有着基础性的认识。那么，数据流究竟是如何产生并被 recv 所接受？数据流要如何通过 send 发送并由套接字的对端接收？这些问题显然不是只通过进程的用户态所能够办到的。换个问题，大家是否曾经见过图 3描述的 I/O 模型，是否对所谓内核中的相关操作感到好奇？那究竟内核态逻辑做了哪些工作，且看下文内容。

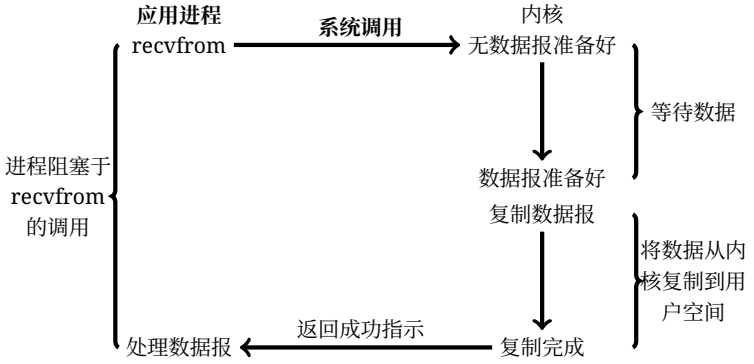


图 3: 阻塞式 I/O 模型

2 内核套接字实现概览

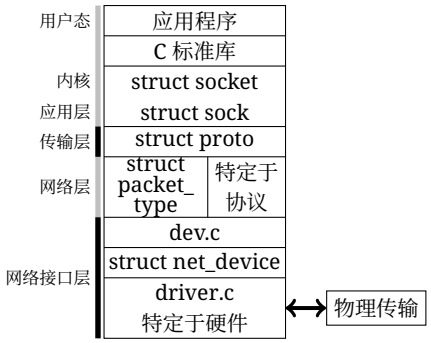


图 4: 内核网络分层模型

图 4展示了内核 C 代码关于网络的分层实现，对比图 1也很容易就可以联系起来。分层实现必然涉及到各层的交互，必不可少的也就有数据交互的需求。内核使用了被称为**套接字缓冲区 (socket buffer)** 的数据结构来进行支持，结构体中针对同一个字符数组，提供指针 *mac_header, *network_header, *transport_header 分别定位 MAC 帧、IP 报及传输层报文。

```
struct sk_buff {
    /* These two members must be first. */
    struct sk_buff      *next;
    struct sk_buff      *prev;

    struct sock     *sk;
    ktime_t         tstamp;
    struct net_device   *dev;

    struct  dst_entry   *dst;
    struct  sec_path    *sp;

    char            cb[48];

    unsigned int        len,
                data_len;
    __u16           mac_len,
                hdr_len;
    union {
        __wsum      csum;
        struct {
            __u16   csum_start;
            __u16   csum_offset;
        };
    };
    __u32           priority;
    __u8            local_df:1,
                cloned:1,
```

```
                ip_summed:2,
                nohdr:1,
                nfctinfo:3;
    __u8            pkt_type:3,
                fclone:2,
                ipvs_property:1,
                nf_trace:1;
    __be16          protocol;

    void            (*destructor)(struct sk_buff *skb);

    __u32           mark;

    sk_buff_data_t      transport_header;
    sk_buff_data_t      network_header;
    sk_buff_data_t      mac_header;
    /* These elements must be at the end, see alloc_skb() for details.
     */
    sk_buff_data_t      tail;
    sk_buff_data_t      end;
    unsigned char       *head, *data;
    unsigned int        truesize;
    atomic_t        users;
};
```

至于多组套接字缓冲区，内核使用了 struct sk_buff_head 来维护整个等待队列。具体形式如图 5所示，通过 prev 和 next 指针来维护起双向列表。

```
struct sk_buff_head {
    /* These two members must be first. */
    struct sk_buff  *next;
    struct sk_buff  *prev;

    __u32       qlen;
    spinlock_t  lock;
};
```

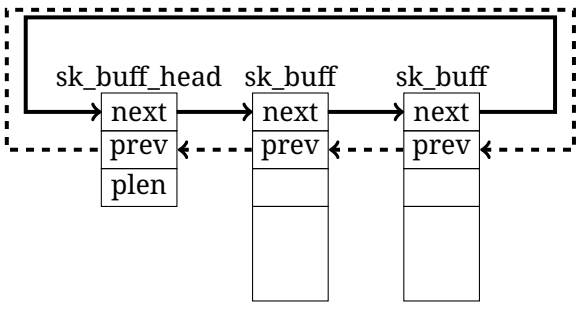
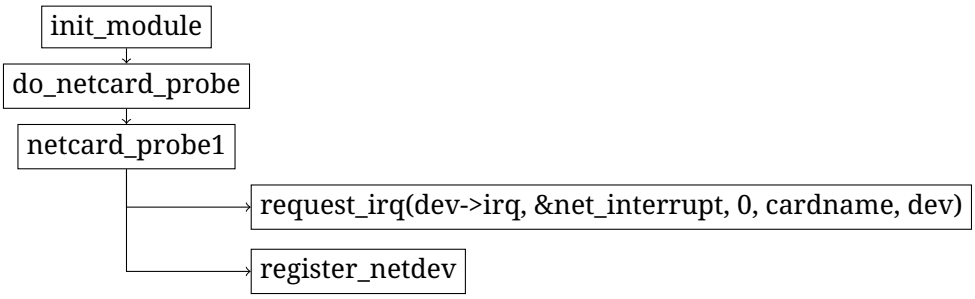


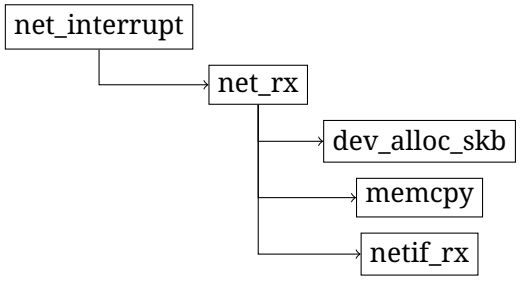
图 5: 等待队列

3 网络设备接收数据

了解过了网络分层模型的数据交互模型，本节将详细描述数据经由物理链路到达网络设备，并最终被写入到套接字缓冲区的流程。最初的步骤从向内核注册网卡驱动开始，随后注册网络中断 (net_interrupt)，注册网络设备，再等待内核设备接收数据流并发起 IRQ。



简单地了解了网络设备向内核的注册流程后，接下来就是等待网络设备发起 IRQ 了。在网络设备接收到数据流之后，会向内核发起网络中断 (net_interrupt)，随后快速进入 net_rx 函数，初始化一个新的套接字缓冲区并从网络设备中获取数据填充缓冲区（由此标志着数据已经从网络设备到达了物理内存）。

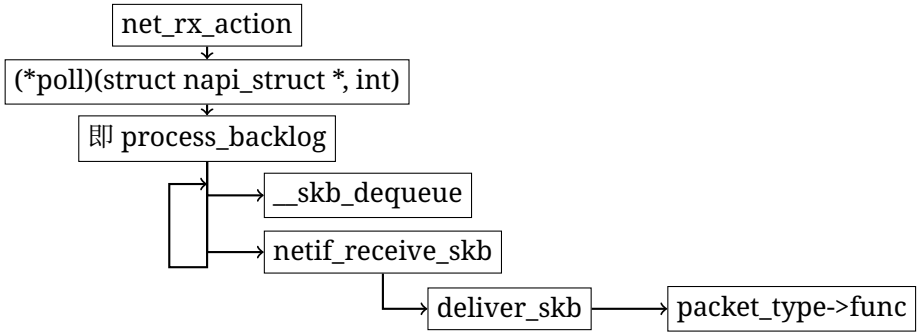


之前的 net_interrupt、net_rx 函数都是特定于驱动模块的实现，而 netif_rx 函数位于/net/core/dev.c，调用该函数标志着 CPU 控制权由特定于网络设备的代码转移到了网络层的通用接口部分。首先，当然先来看看 netif_rx 的相关操作。

```
int netif_rx(struct sk_buff *skb)
{
    struct softnet_data *queue;
    unsigned long flags;

    /* if netpoll wants it, pretend we never saw it */
    if (netpoll_rx(skb))
        return NET_RX_DROP;

    if (!skb->tstamp.tv64)
        net_timestamp(skb);

```

```
    /*
     * The code is rearranged so that the path is the most
     * short when CPU is congested, but is still operating.
     */
    local_irq_save(flags);
    queue = &__get_cpu_var(softnet_data);

    __get_cpu_var(netdev_rx_stat).total++;
    /* 确认是否超过等待队列限长(netdev_max_backlog) */
    if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
        /* 队列非空，向队尾插入新的skb*/
        if (queue->input_pkt_queue.qlen) {
enqueue:
            dev_hold(skb->dev);
            __skb_queue_tail(&queue->input_pkt_queue, skb);
            local_irq_restore(flags);
            return NET_RX_SUCCESS;
        }

        /* 空队列，先设置NET_RX_SOFTIRQ软中断 */
        napi_schedule(&queue->backlog);
        goto enqueue;
    }

    __get_cpu_var(netdev_rx_stat).dropped++;
    local_irq_restore(flags);

    kfree_skb(skb);
    return NET_RX_DROP;
}
```

将套接字缓冲区 (struct sk_buff *skb) 加入到 softnet_data 队列之后，由网络设备发起的硬件中断 net_interrupt 到此结束，CPU 就此退出中断子任务，回归正常程序。而后续的相关操作，例如将套接字缓冲区的数据逐层解析最终交付用户程序等，都由硬件中断退出前设置的软中断 NET_RX_SOFTIRQ 处理程序 net_rx_action 进行处理。



到此为止，套接字缓冲区的数据即上升到了网络层进行继续处理。

3.1 高速网络

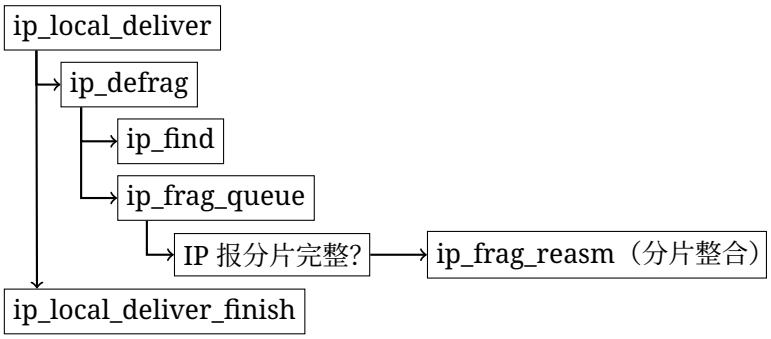
正常情况下，我们都默认 CPU 的运转拥有这最高的速度。但是，随着网络设备的发展，高速网络的出现导致出现了 CPU 对网络中断的处理速度赶不上网络设备发起网络中断的速度。由此，也就引发了所谓的“中断风暴”。为了解决这个问题，NAPI（区别与原有的中断处理方式）使用了 IRQ 与轮询的组合：即在网络设备第一次接收到数据分组发起 IRQ 之后，内核中断处理程序主动禁止 IRQ，并将该设备加入轮询表中，并在处理完第一个数据分组后，以轮询 (poll) 的方式确认轮询表中的各个网络设备是否还有更多的数据分组需要处理。当然，整个轮询过程（即此次中断处理流程）不能超过一个 jiffie，否则强制结束此次中断；并且在结束中断时重新允许 IRQ。

4 网络层

上一节套接字缓冲区控制权由网络接口层转交网络层是通过 packet_type->func 实现的。这里的 *func 是一个函数指针，由具体的 IP 层处理协议预置，目前我们只描述 IPv4 协议下的网络层处理逻辑。IPv4 协议下的 packet_type 实例对象是 ip_packet_type，由下面的代码块可以看到最终调用了 ip_rcv。

```
static struct packet_type ip_packet_type = {
    .type = __constant_htons(ETH_P_IP),
    .func = ip_rcv,
    .gso_send_check = inet_gso_send_check,
    .gso_segment = inet_gso_segment,
};
```

ip_rcv 对套接字缓冲区数据进行解析，处理 IP 报头相关的数据，并判断报文的合法性，最终通过钩子函数 NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,ip_rcv_finish) 进行路由选择，确定是将数据传递到更高的协议层或者直接转发。此处只探究接收报文且传递给传输层的逻辑，路由后最终由 ip_local_deliver 继续处理，通过 ip_defrag 进行 IP 报文的重新装配（大家都知道 IP 报文受限于低层协议荷载限制，大多数时候都不得不将报文分割成多份进行发送）。

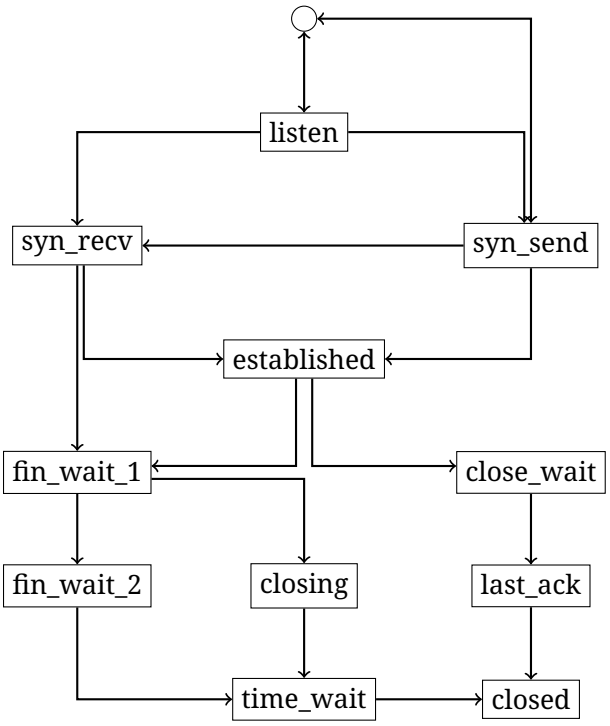


如果 ip_defrag 返回 NULL，表示该 IP 报文不完整，则结束处理，等待网

络接口层继续提供更多的数据来补全 IP 报内容；否则将通过 NF_HOOK 进行 ip_local_deliver_finish 进行最终处理，根据 IP 报文中传输层协议首部确定传输层处理逻辑，并移交控制权。

5 传输层

基于 IP 的传输层协议主要有 TCP 和 UDP，前者用于建立安全的、面向连接的服务；后者用于发送数据报。此处指简单描述 TCP 相关的处理逻辑：套接字缓冲区如何从网络层经由传输层最终提交给应用层。由于 TCP 是面向连接的，也就有了建立连接、释放连接、数据流按序传输三个流程。



套接字缓冲区上升到传输层，诸如 TCP 协议就需要面对三次握手，四次挥手的流程。不过，相信这点不会成为我们学习内核网络实现的大问题。当然，事实上的 TCP 协议实现就显得复杂得多得多，这里不会继续探究。总之，就接收数据而言，传输层最终将有效的套接字缓冲区加入到 struct sock 的接收队列 sk_receive_queue。

6 应用层

等到缓冲区数据终于上升到应用层，应该可以想象，马上就要和用户程序打交道了，也就是终于要和各位程序员自己的代码实现交互了。在开始前，优先介

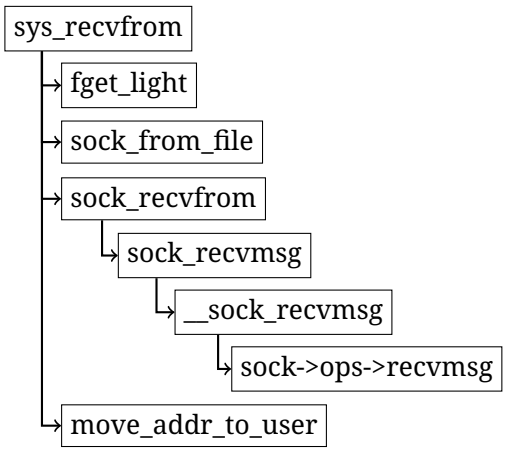
绍图 4描述的应用层核心的两个数据结构。

```
/**
 *  struct socket - general BSD socket
 *  @state: socket state (%SS_CONNECTED, etc)
 *  @flags: socket flags (%SOCK_ASYNC_NOSPACE, etc)
 *  @ops: protocol specific socket operations
 *  @fasync_list: Asynchronous wake up list
 *  @file: File back pointer for gc
 *  @sk: internal networking protocol agnostic socket representation
 *  @wait: wait queue for several uses
 *  @type: socket type (%SOCK_STREAM, etc)
 */
struct socket {
    socket_state        state;
    unsigned long       flags;
    const struct proto_ops  *ops;
    struct fasync_struct    *fasync_list;
    struct file     *file;
    struct sock     *sk;
    wait_queue_head_t   wait;
    short           type;
};
```

```
struct sock_common {
    unsigned short      skc_family;
    volatile unsigned char  skc_state;
    unsigned char       skc_reuse;
    int         skc_bound_dev_if;
    struct hlist_node   skc_node;
    struct hlist_node   skc_bind_node;
    atomic_t        skc_refcnt;
    unsigned int        skc_hash;
    struct proto        *skc_prot;
    struct net      *skc_net;
};
struct sock {
    struct sock_common  __sk_common;
    ...
    unsigned char       sk_shutdown : 2,
                sk_no_check : 2,
                sk_userlocks : 4;
    unsigned char       sk_protocol;
    unsigned short      sk_type;
    int         sk_rcvbuf;
    socket_lock_t       sk_lock;
    struct {
        struct sk_buff *head;
        struct sk_buff *tail;
    } sk_backlog;
    wait_queue_head_t   *sk_sleep;
    struct dst_entry    *sk_dst_cache;
    struct xfrm_policy  *sk_policy[2];
```

```
    rwlock_t        sk_dst_lock;
    atomic_t        sk_rmem_alloc;
    atomic_t        sk_wmem_alloc;
    atomic_t        sk_omem_alloc;
    int         sk_sndbuf;
    struct sk_buff_head sk_receive_queue;
    struct sk_buff_head sk_write_queue;
    struct sk_buff_head sk_async_wait_queue;
    ...
};
```

以接收消息举例，用户程序主动调用 recvfrom（C 标准库提供的函数），随后由 C 库程序发起一次系统调用，最终陷入内核态调用 sys_recvfrom。从流程图上可以看到，sys_recvfrom 做完成一系列准备工作之后，调用 sock_recvfrom 完成 struct socket 到 struct sock 的数据结构交互，随后由 sock_recvmsg 提供已经添加到 sock 接收队列 sk_receive_queue 上的套接字缓冲区。这里还需要注意的是，看到 sock_recvmsg 调用链调用了 sock->ops->recvmsg，这是一个函数指针，用于区分 TCP、UDP 或其它自定义的传输层协议的具体处理逻辑。

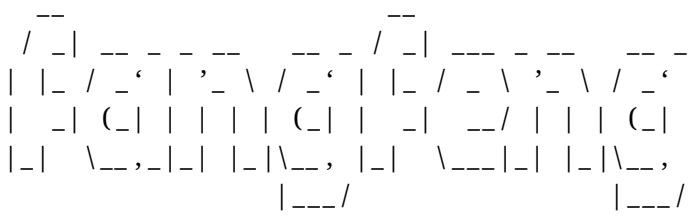


如果这个套接字创建时声明的是 TCP，则 sock->ops->recvmsg 指向的是 tcp_recvmsg。tcp_recvmsg 主要负责两类操作，如果能够从 sock 接收队列 sk_receive_queue 上取得数据，则直接返回，并最终通过 move_addr_to_user 完成数据从内核数据区到用户程序数据区的拷贝；否则，调用 sk_wait_data 使当前进程陷入睡眠状态，等待 sk_recvice_queue 接收队列上被添加上数据，随后主动唤醒等待消息的进程。当然，这里是直接设置了一个确定的等待时间，如果最终没有收到消息，则超时报错再退出。

7 Final

用 LaTeX 写的第一篇文档，踩了很多的坑，但总算是写完了这样一篇有史以来添加了最多的插图的博客:）就网络来说，本篇描述的仅仅是网络阻塞式 I/O 模

型中的关于接收消息的简单一瞥。诸如非阻塞式 I/O 模型、Linux 内核 select、poll、epoll 的实现根本还毫无概念。下次有机会再写吧!



References

[1] Wolfgang Mauerer and 郭旭. 深入 *Linux* 内核架构. 人民邮电出版社, 2010.