中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

1924-2024 中山大學 世纪华诞 100th ANNIVERSARY SUN YAT-SEN UNIVERSITY

网络协议栈与系统实现

SSE202/204: 操作系统原理

苏玉鑫

suyx35@mail.sysu.edu.cn

助教：龙玉丹 单诗雯 毛晨希 沈志轩 郑灿峰 胡伟峰

版权信息

- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
  - https://ipads.se.sjtu.edu.cn/courses/os/
- 其它参考资料：
  - 清华大学操作系统公开课
    - https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351
    - 介绍标准内容，适合考研
  - 南京大学计算机软件研究所
    - http://jyywiki.cn/OS/2025/
    - https://space.bilibili.com/202224425/channel/collectiondetail?sid=192498
    - 比较有趣

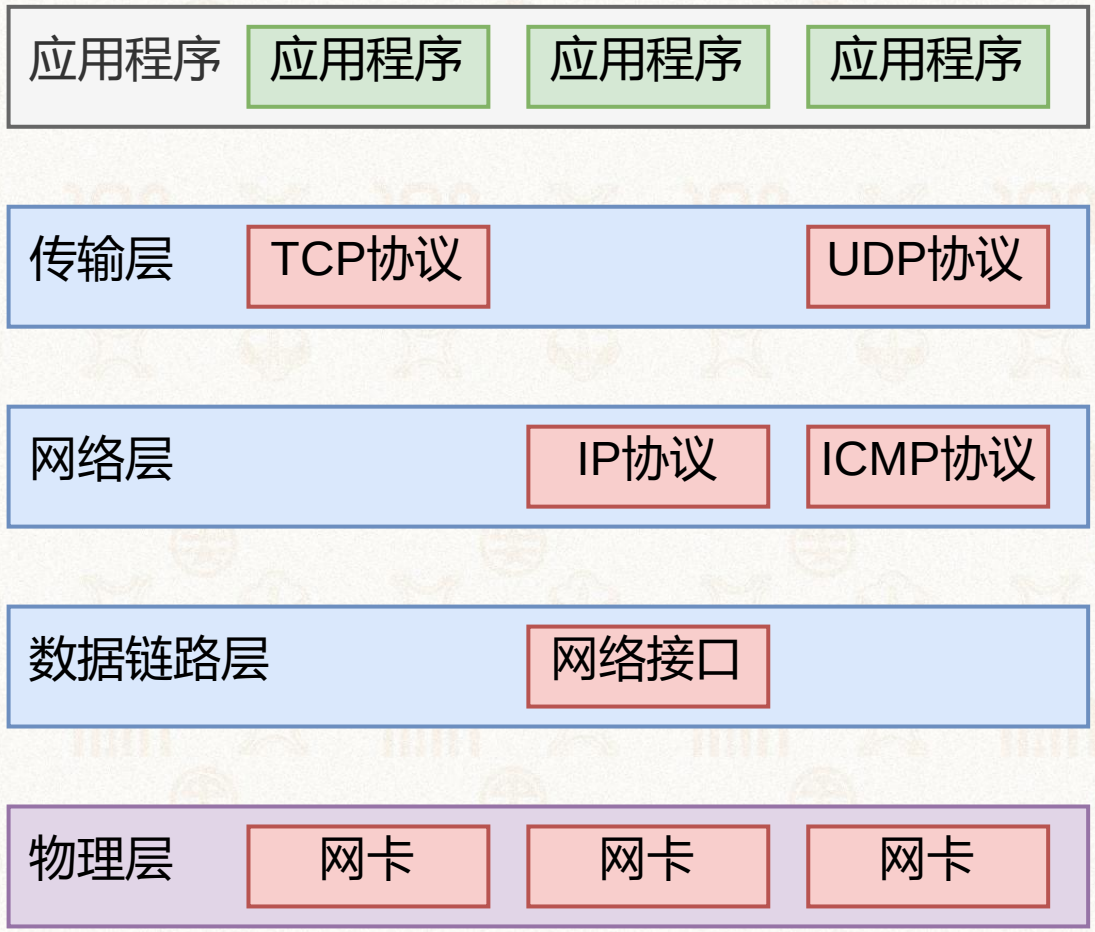
大纲

- 越俎代庖部分
  - 网络协议的分层模型
  - 套接字模型
- 网络驱动模型
- Linux系统收包过程
  - 函数视角
  - 数据视角
- Linux系统发包过程
- 网络处理性能优化
  - 挑战
  - 数据面控制面分离
- Intel DPDK 软件优化方案
  - 总体框架
  - 无锁环
  - 内存池
  - 其它模块
  - 扩展框架
- 硬件优化方案

大纲

- 越俎代庖部分
  - 网络协议的分层模型
  - 套接字模型
- 网络驱动模型
- Linux系统收包过程
  - 函数视角
  - 数据视角
- Linux系统发包过程
- 网络处理性能优化
  - 挑战
  - 数据面控制面分离
- Intel DPDK 软件优化方案
  - 总体框架
  - 无锁环
  - 内存池
  - 其它模块
  - 扩展框架
- 硬件优化方案

网络协议栈的分层模型



游戏进程或聊天进程

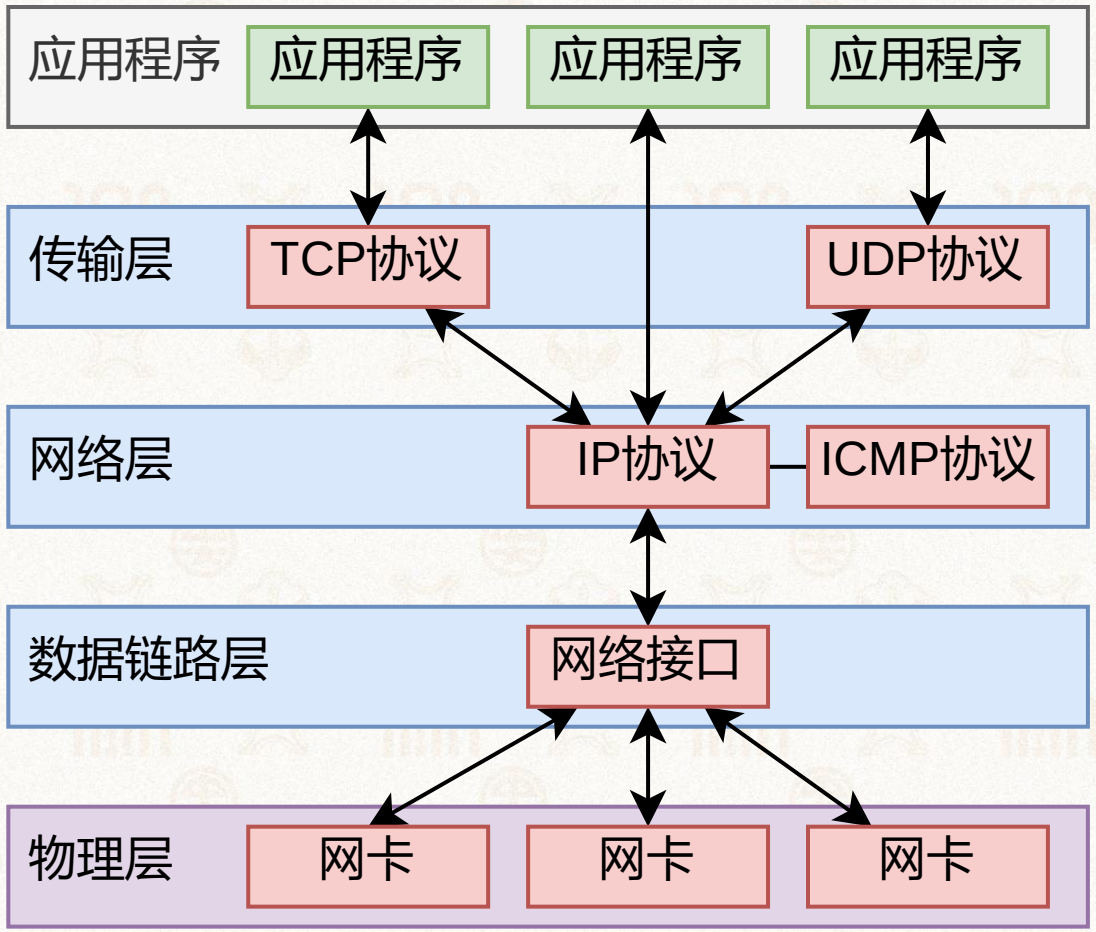
TCP或UDP协议：端口

IP协议：主机地址

网络设备驱动

网络设备

网络协议栈的分层模型



游戏进程或聊天进程

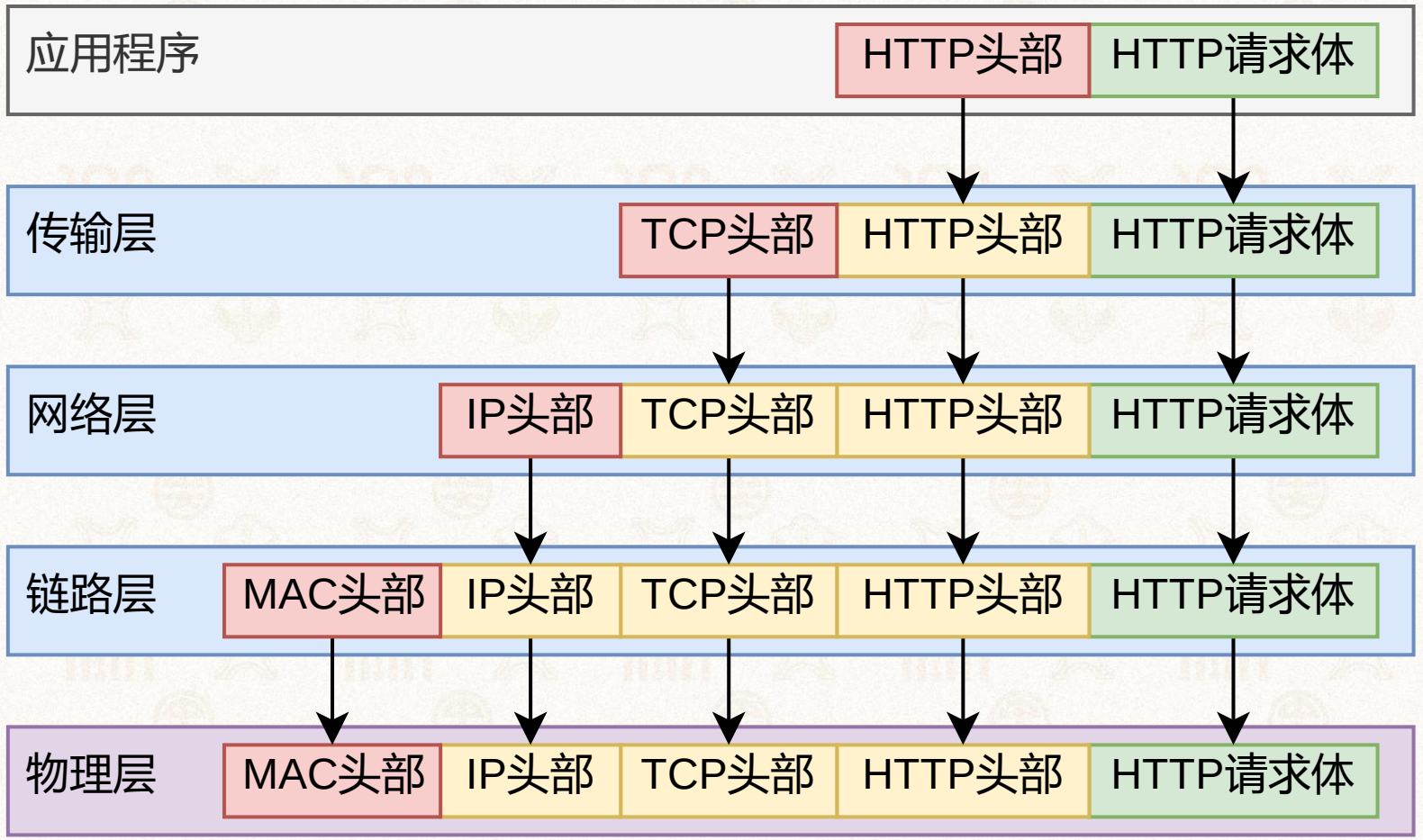
TCP或UDP协议：端口

IP协议：主机地址

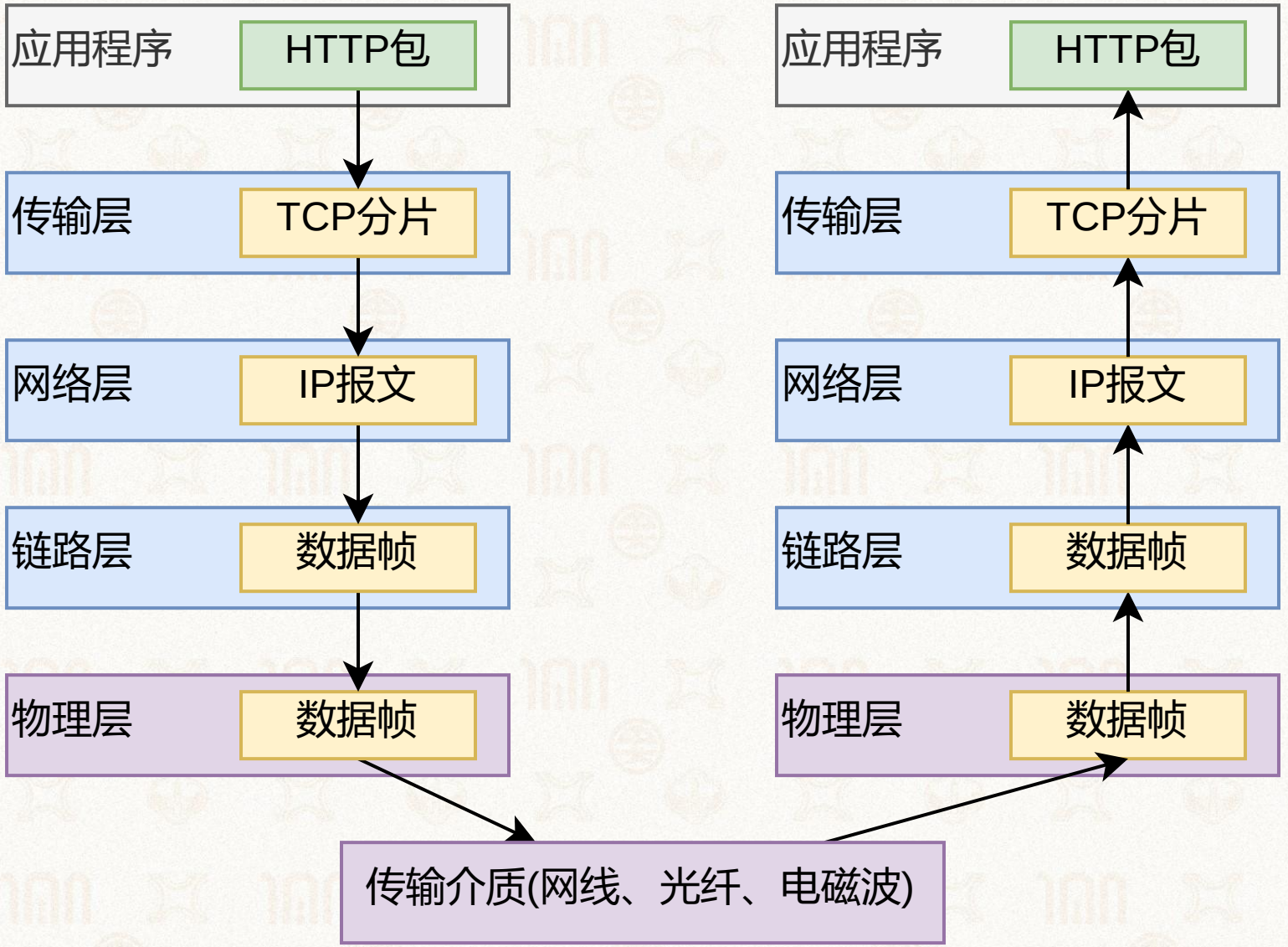
网络设备驱动

网络设备

网络包的发送阶段


应用程序
HTTP头部
HTTP请求体
传输层
TCP头部
HTTP头部
HTTP请求体
网络层
IP头部
TCP头部
HTTP头部
HTTP请求体
链路层
MAC头部
IP头部
TCP头部
HTTP头部
HTTP请求体
物理层
MAC头部
IP头部
TCP头部
HTTP头部
HTTP请求体

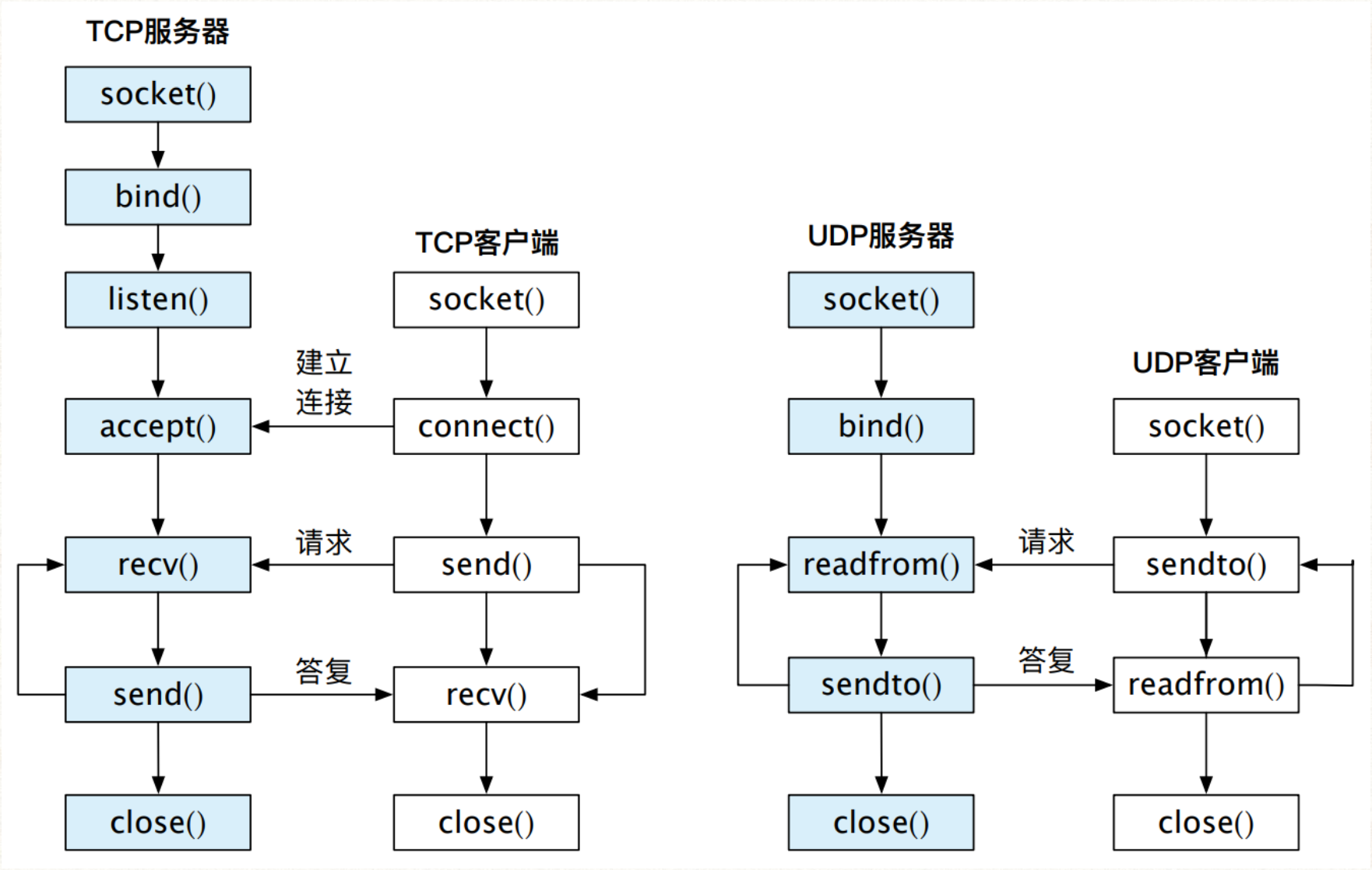
网络包的跨机器传输过程



大纲

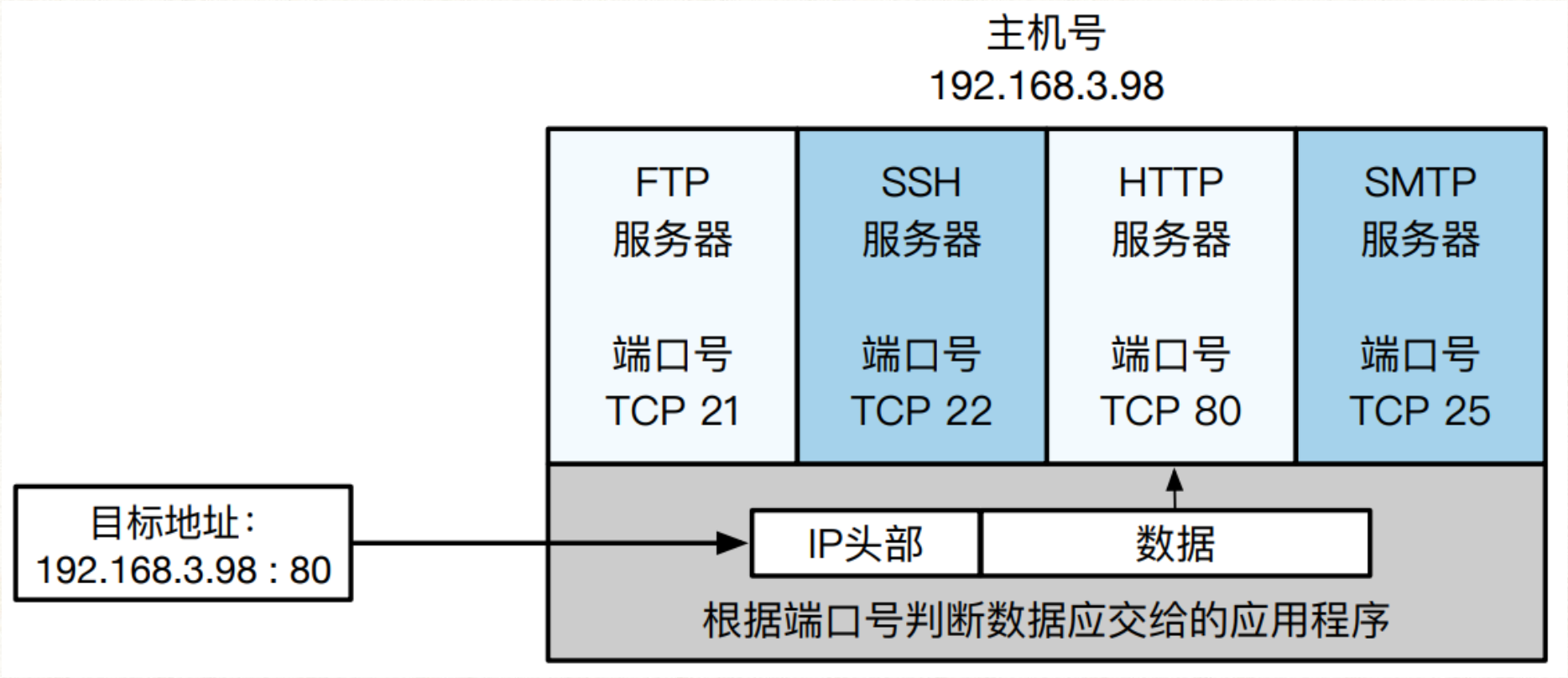
- 越俎代庖部分
  - 网络协议的分层模型
  - 套接字模型
- 网络驱动模型
- Linux系统收包过程
  - 函数视角
  - 数据视角
- Linux系统发包过程
- 网络处理性能优化
  - 挑战
  - 数据面控制面分离
- Intel DPDK 软件优化方案
  - 总体框架
  - 无锁环
  - 内存池
  - 其它模块
  - 扩展框架
- 硬件优化方案

网络编程模型：套接字编程



网络编程模型：套接字编程

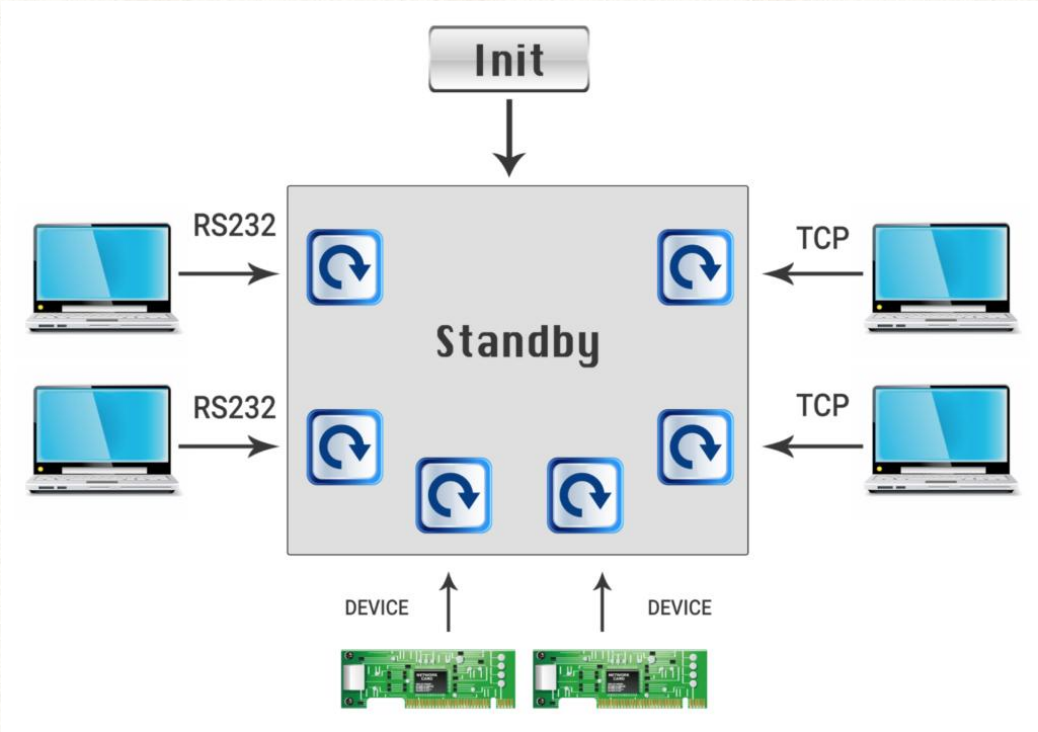
- 网络协议栈根据端口号识别应用进程



网络编程模型：套接字编程

- 每处理一次请求都创建一个进程太浪费资源

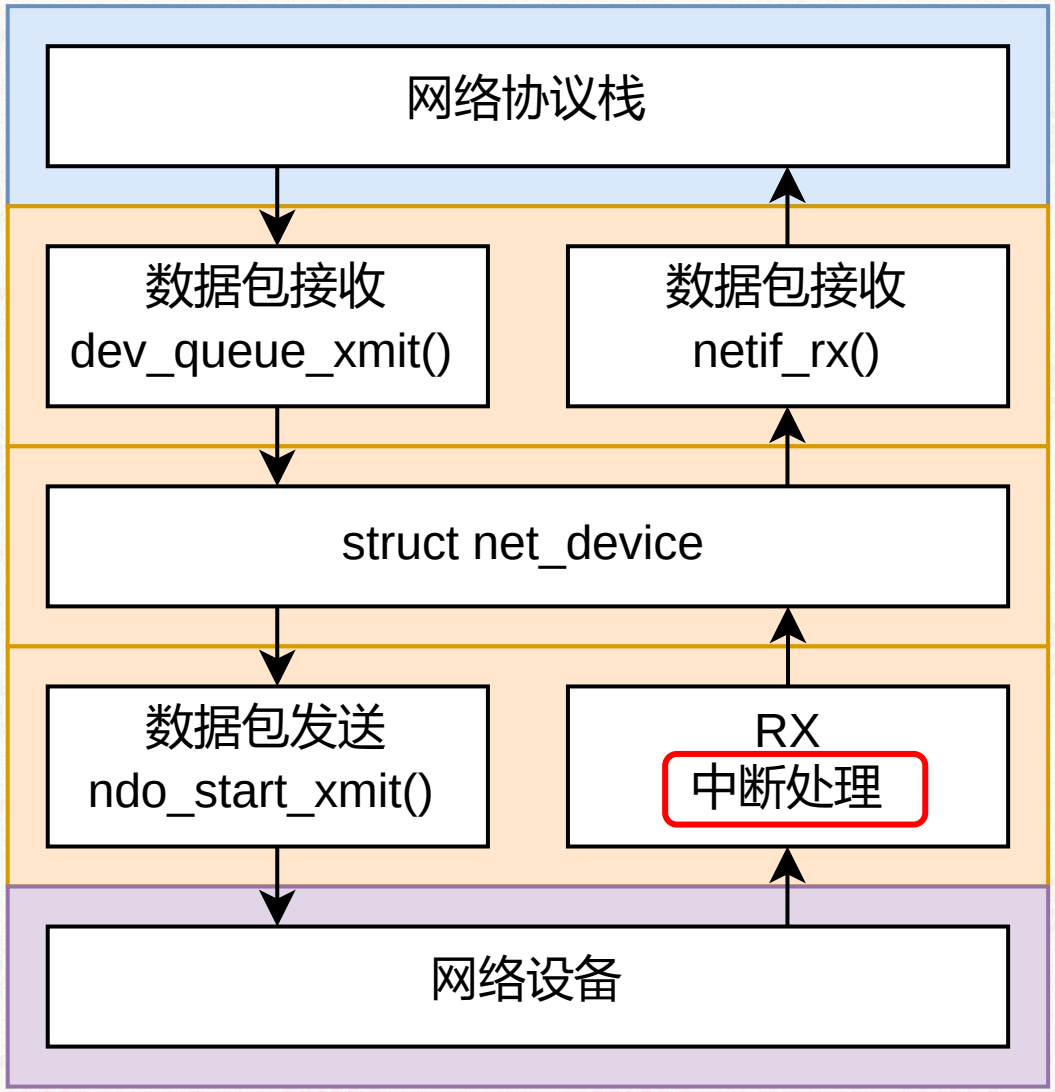
- 利用纤程，减少上下文切换带来的资源开销
- 利用多路复用，提高处理效率
  - select
  - poll
  - epoll
  - kqueue



大纲

- 越俎代庖部分
  - 网络协议的分层模型
  - 套接字模型
- 网络驱动模型
- Linux系统收包过程
  - 函数视角
  - 数据视角
- Linux系统发包过程
- 网络处理性能优化
  - 挑战
  - 数据面控制面分离
- Intel DPDK 软件优化方案
  - 总体框架
  - 无锁环
  - 内存池
  - 其它模块
  - 扩展框架
- 硬件优化方案

Linux网络驱动模型



传输层、网络层...

网络协议接口层

网络设备接口层

设备驱动层

物理层

网卡硬中断（ISR）


```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
 17:          0          0          0          0   GICv2  29 Level     arch_timer
 18:    7331554    2731032     433991     492919   GICv2  30 Level     arch_timer
 23:      26740          0          0          0   GICv2 114 Level     DMA IRQ
 31:     757858          0          0          0   GICv2  65 Level     fe00b880.mailbox
 34:       6556          0          0          0   GICv2 153 Level     uart-pl011
 36:          0          0          0          0   GICv2 169 Level     brcmstb_thermal
 37:    8457672          0          0          0   GICv2 158 Level     mmc1, mmc0
 43:          0          0          0          0   GICv2 106 Level     v3d
 45:    7567287          0          0          0   GICv2 189 Level     eth0
 52:         51          0          0          0   GICv2  66 Level     VCHIQ doorbell
 53:          0          0          0          0   GICv2 175 Level     PCIe PME, aerdrv
 54:         40          0          0          0 Brcm_MSI 524288 Edge      xhci_hcd
```

网卡软中断（softirq）

```
$ cat /proc/softirqs
```

	CPU0	CPU1	CPU2	CPU3
HI:	2	0	0	0
TIMER:	4709143	1000453	238535	156693
NET_TX:	12764	272	293	196
NET_RX:	650451	4930	7150	5162
BLOCK:	0	0	0	0
IRQ_POLL:	0	0	0	0
TASKLET:	6775576	36	24	33
SCHED:	4719393	1043269	255401	165523
HRTIMER:	0	0	0	0
RCU:	2878697	423063	251156	170016

网卡收发的情况

```
$ ifconfig
eth0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
      ether dc:a6:32:4b:c4:00    txqueuelen 1000   (Ethernet)

wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 192.168.10.194   netmask 255.255.0.0   broadcast 192.168.255.255
      inet6 fe80::aa72:beb8:1888:e82e   prefixlen 64  scopeid 0x20<link>
      ether dc:a6:32:4b:c4:01   txqueuelen 1000       (Ethernet)
      RX packets 655811          bytes 164726673 (157.0 MiB)
      RX errors 0    dropped 0   overruns 0    frame 0
      TX packets 21714    bytes 2496958 (2.3 MiB)
      TX errors 0           dropped 0 overruns 0     carrier 0     collisions 0
```

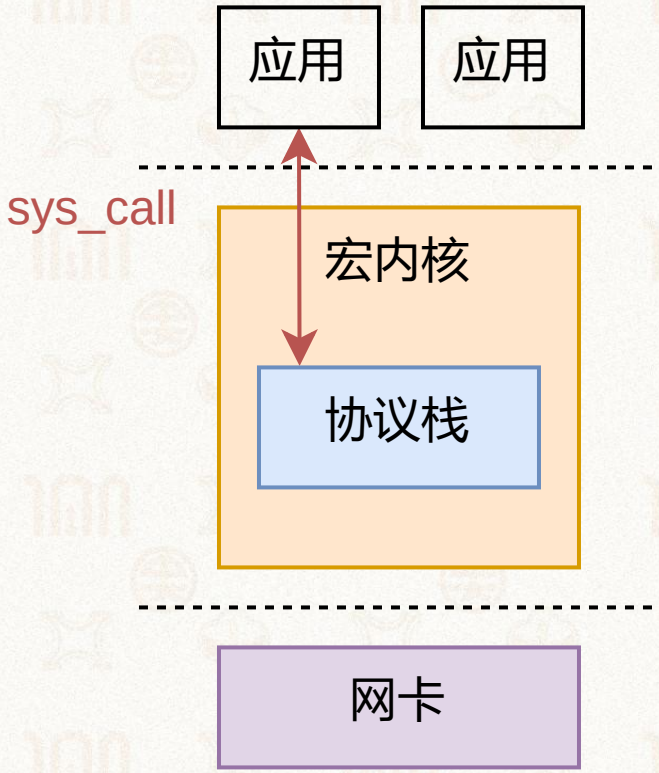
中断合并

- Interrupt coalescing
  - 当外设中断次数累计到一定阈值时，再向CPU发送中断
  - 或者到某个timeout，向CPU发送中断

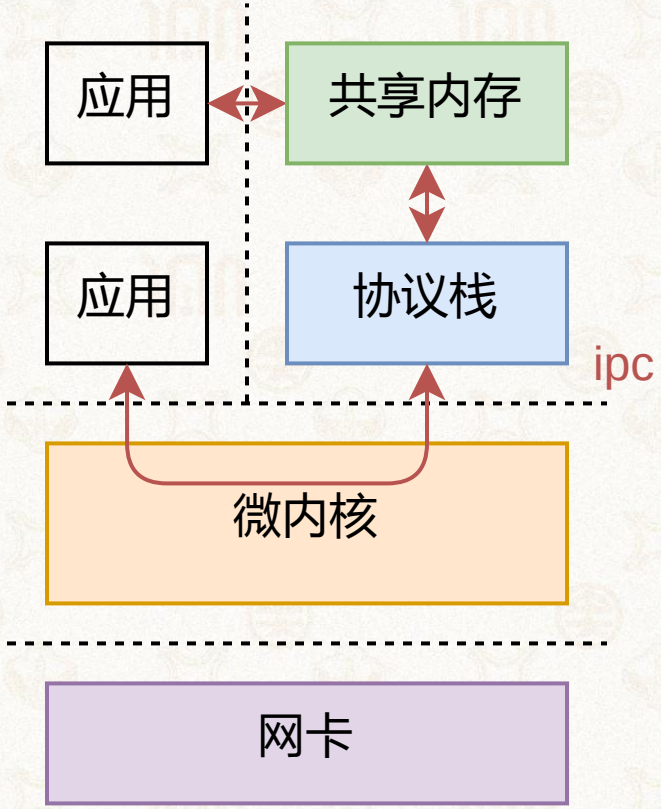
- 可避免“中断风暴”

- 更少的中断次数意味着：
  - 更高的吞吐量
  - 但也增加了中断响应的延迟

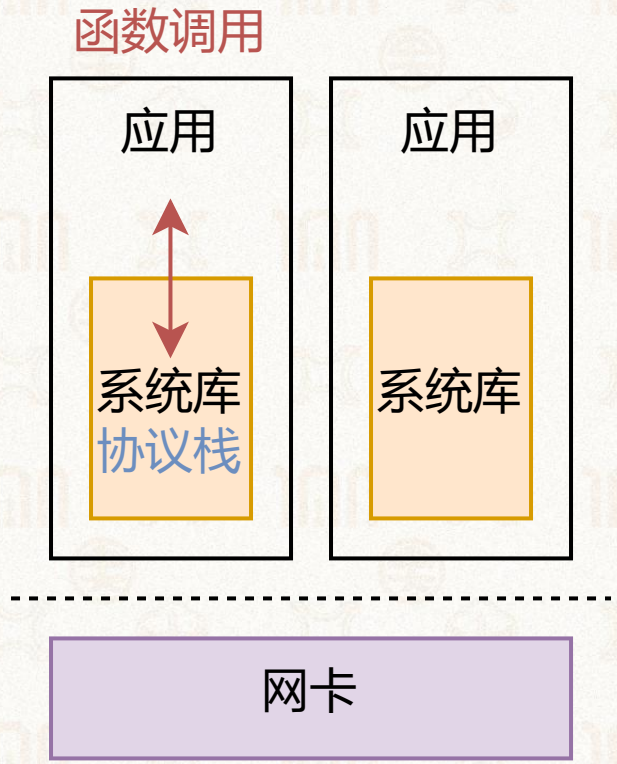
架构对比



宏内核系统



微内核系统

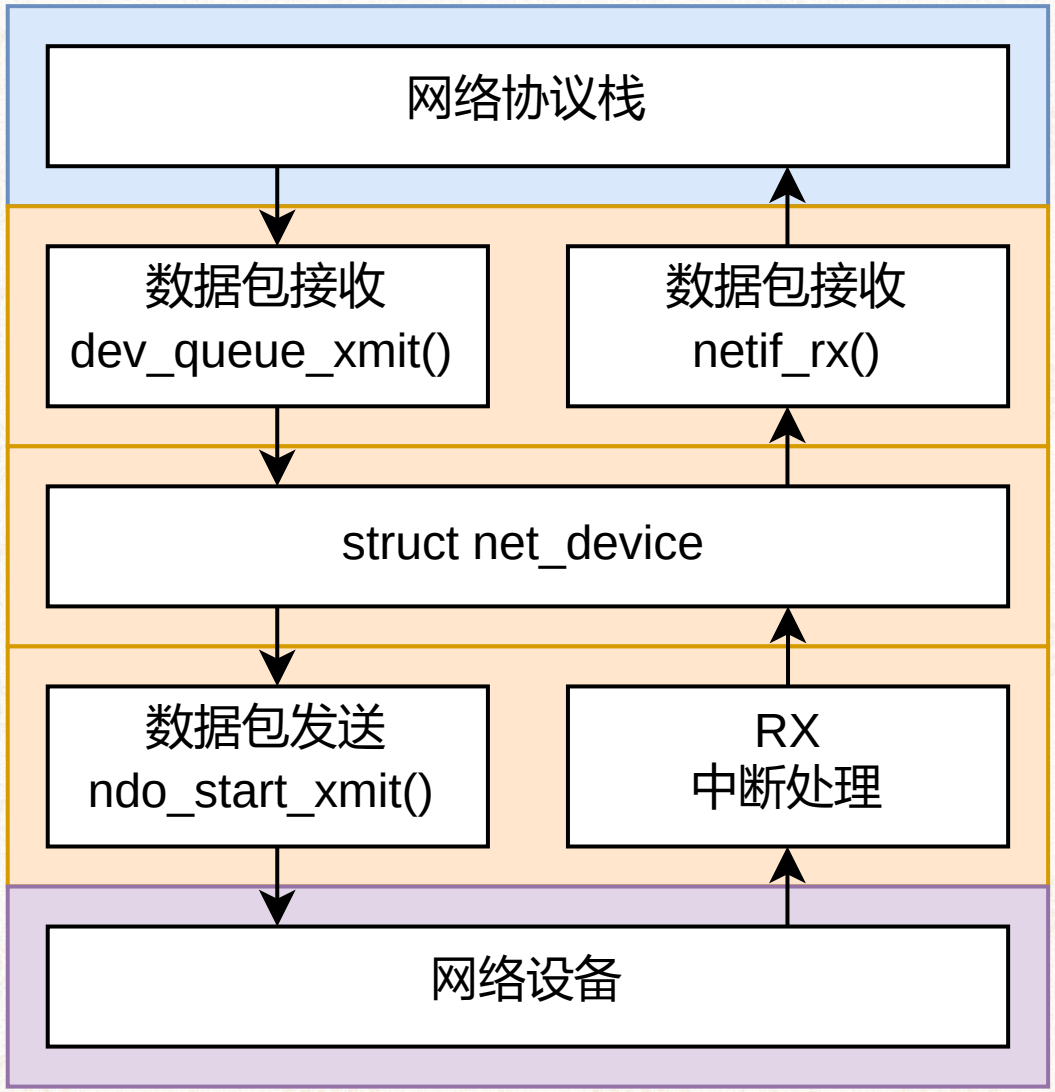


LibOS系统

大纲

- 越俎代庖部分
  - 网络协议的分层模型
  - 套接字模型
- 网络驱动模型
- Linux系统收包过程
  - 函数视角
  - 数据视角
- Linux系统发包过程
- 网络处理性能优化
  - 挑战
  - 数据面控制面分离
- Intel DPDK 软件优化方案
  - 总体框架
  - 无锁环
  - 内存池
  - 其它模块
  - 扩展框架
- 硬件优化方案

Linux网络驱动模型



传输层、网络层...

网络协议接口层

网络设备接口层

设备驱动层

物理层

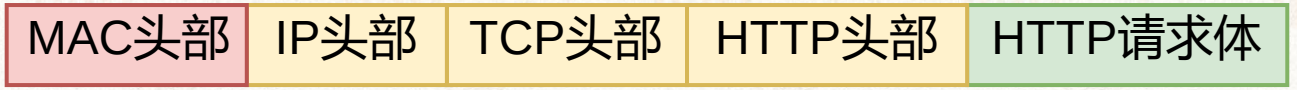
Linux收包过程中关键数据结构：sk_buff

- sk_buff：
  - socket buffer简称，缩写为skb
  - 管理内核数据包

- 本身不存数据，由指针指向真正数据包内存空间
  - data指向缓冲区里保存的数据包的首地址
  - head指向当前协议层所要处理的头部首地址
  - tail指向数据包的尾地址
  - end指向包含数据包的内存块的尾地址

```
struct sk_buff {
    union {
        struct {
            struct sk_buff      *next;
            struct sk_buff      *prev;
            // ...
        };
        struct rb_node      rbnode; // 红黑树
    };
    unsigned int len, data_len;
    __u16           transport_header;
    __u16           network_header;
    __u16           mac_header;
    sk_buff_data_t      tail;
    sk_buff_data_t      end;
    unsigned char       *head, *data;
    // ...
};
```

同一数据包的不同分片



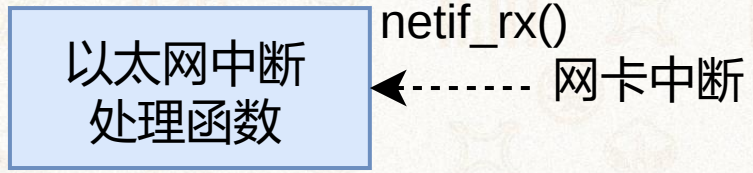
https://elixir.bootlin.com/linux/v5.16.14/source/include/linux/skbuff.h#L731

大纲

- 越俎代庖部分
  - 网络协议的分层模型
  - 套接字模型
- 网络驱动模型
- Linux系统收包过程
  - 函数视角
  - 数据视角
- Linux系统发包过程
- 网络处理性能优化
  - 挑战
  - 数据面控制面分离
- Intel DPDK 软件优化方案
  - 总体框架
  - 无锁环
  - 内存池
  - 其它模块
  - 扩展框架
- 硬件优化方案

Linux网络数据包接收处理过程：函数视角

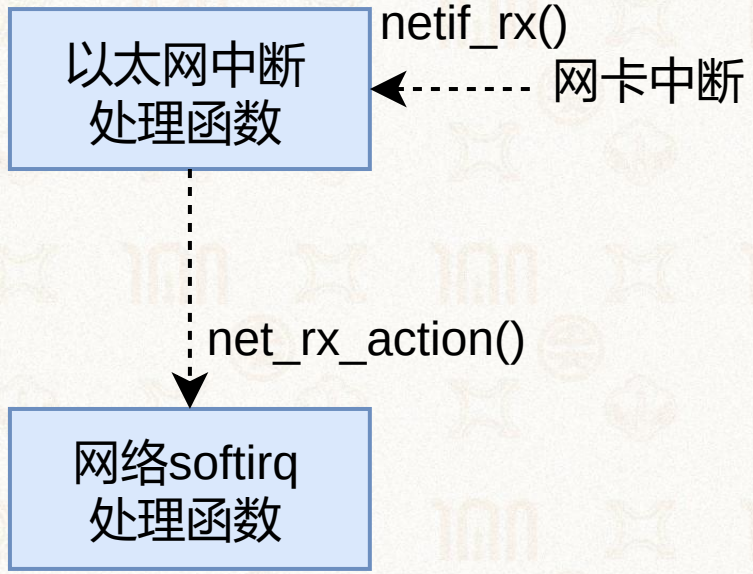
- 网卡收到数据包后，通过DMA将数据拷贝到内核驱动事先分配好的接收队列（RX Ring），
- 随后产生硬中断，触发netif_rx()中断处理函数



netif_rx(): https://elixir.bootlin.com/linux/v5.16.14/source/net/core/dev.c#L4951

Linux网络数据包接收处理过程：函数视角

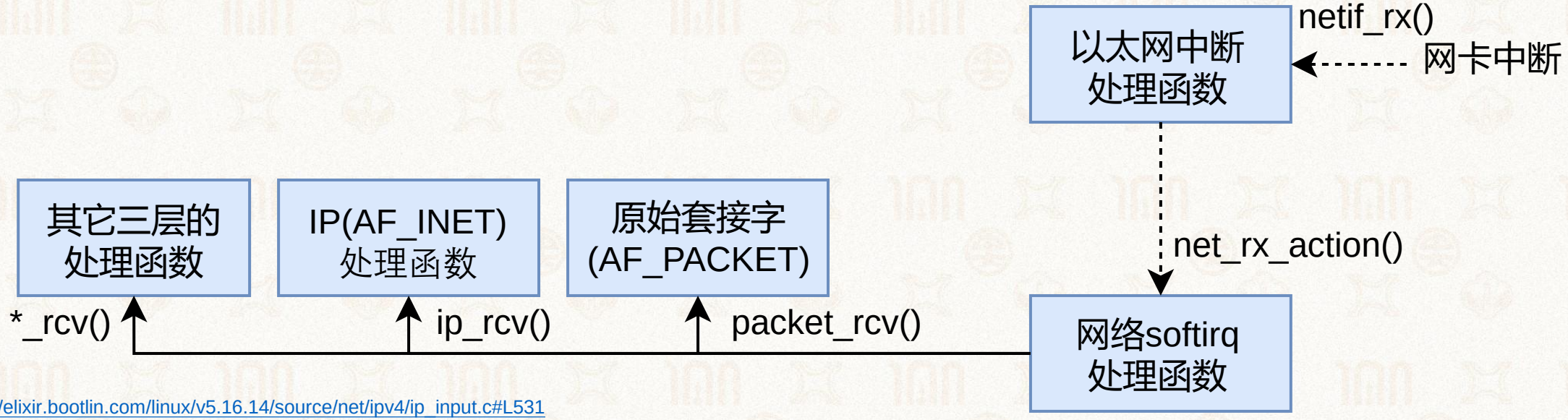
- 网卡驱动程序在软中断处理函数内net_rx_action()为数据包申请sk_buff缓冲区对象
- 同时将数据从接收队列拷贝至sk_buff对象



net_rx_action(): https://elixir.bootlin.com/linux/v5.16.14/source/net/core/dev.c#L7154

Linux网络数据包接收处理过程：函数视角

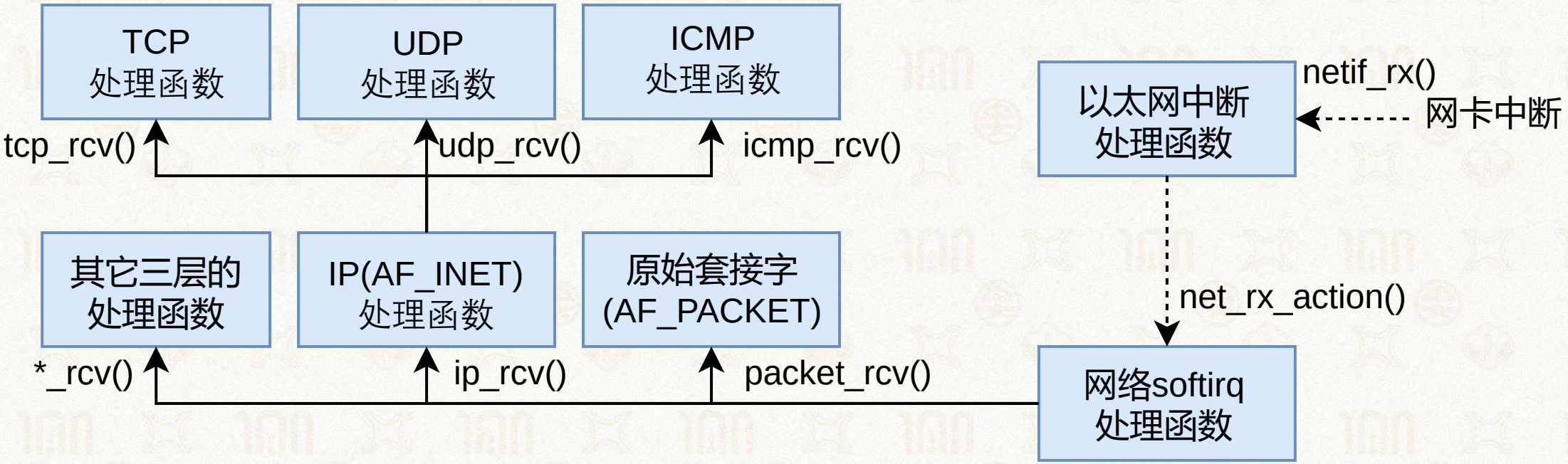
- 驱动层将sk_buff上抛给内核协议栈，
- 由协议栈负责完成协议解析处理：网络层解析



ip_rcv: https://elixir.bootlin.com/linux/v5.16.14/source/net/ipv4/ip_input.c#L531
packet_rcv: https://elixir.bootlin.com/linux/v5.16.14/source/net/packet/af_packet.c#L2109

Linux网络数据包接收处理过程：函数视角

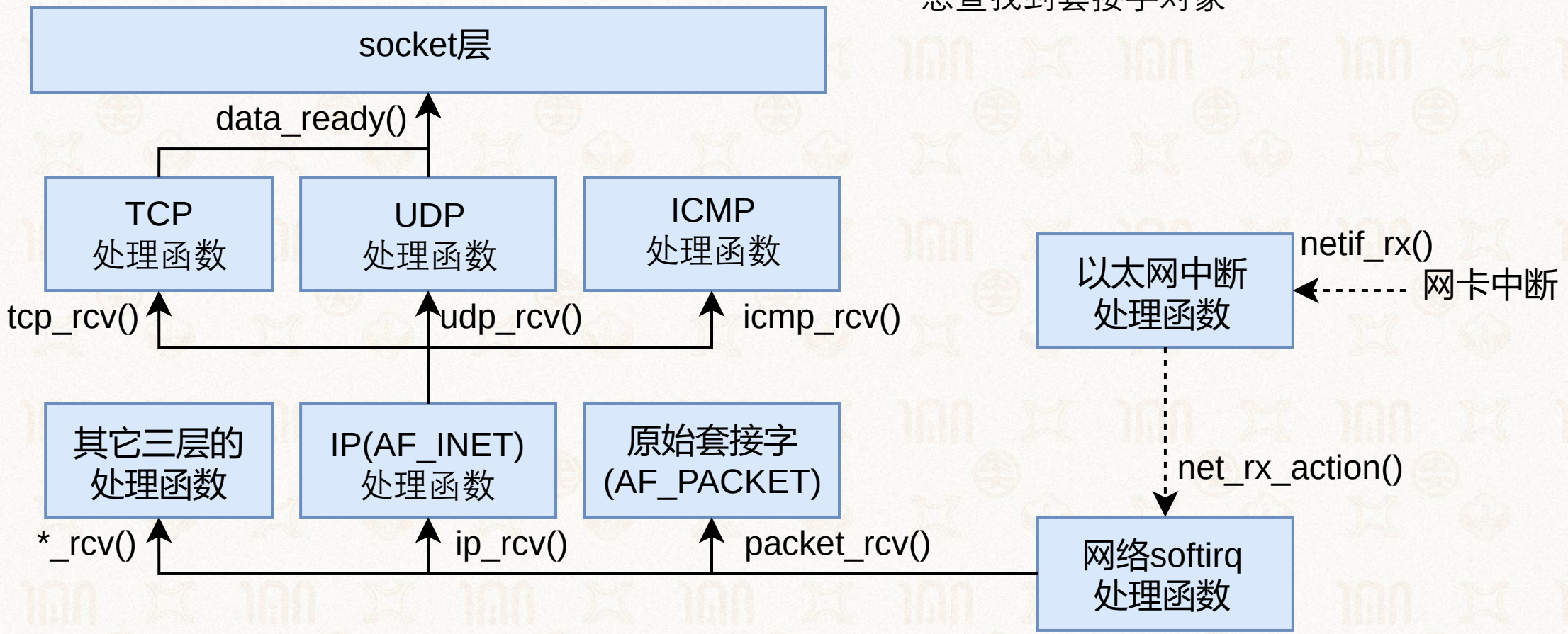
- 驱动层将sk_buff上抛给内核协议栈，
- 由协议栈负责完成协议解析处理：传输层解析



icmp_rcv(): https://elixir.bootlin.com/linux/v5.16.14/source/net/ipv4/icmp.c#L1175

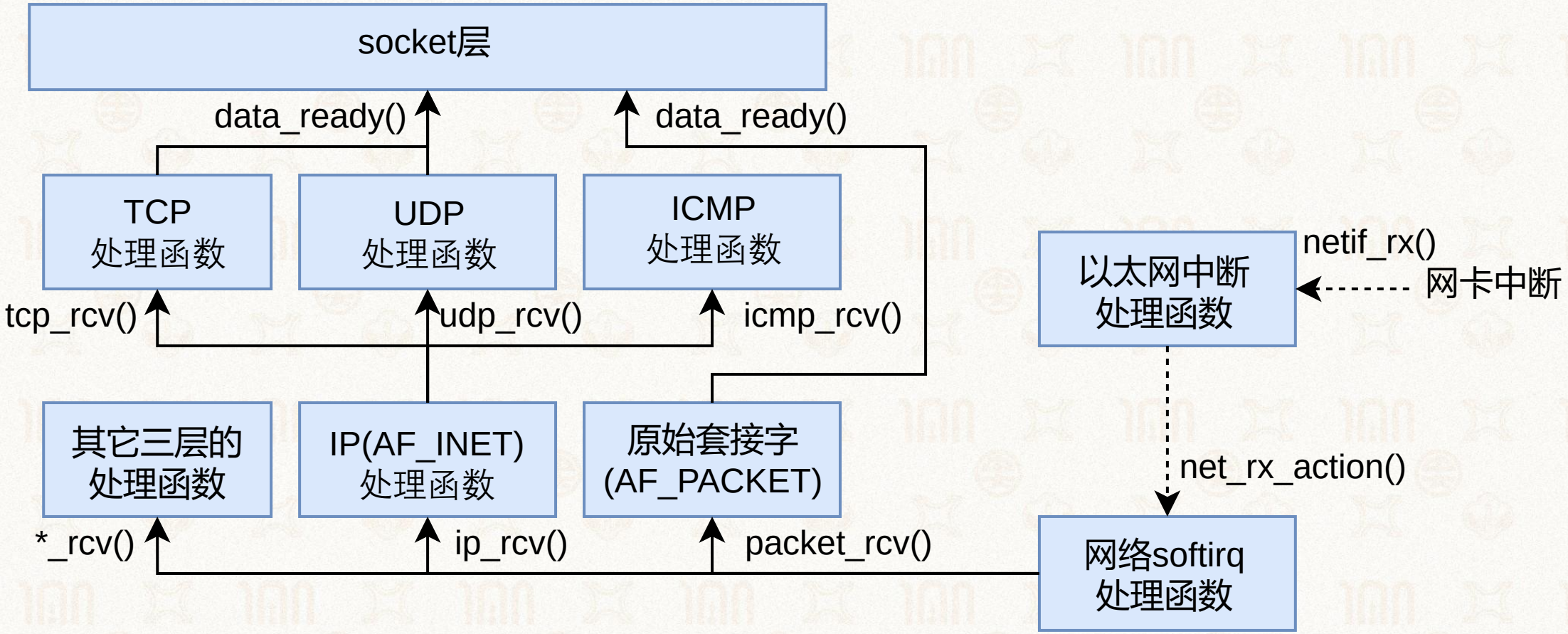
Linux网络数据包接收处理过程：函数视角

- 驱动层将sk_buff上抛给内核协议栈，
- 由协议栈负责完成协议解析处理并根据传输层信息查找到套接字对象

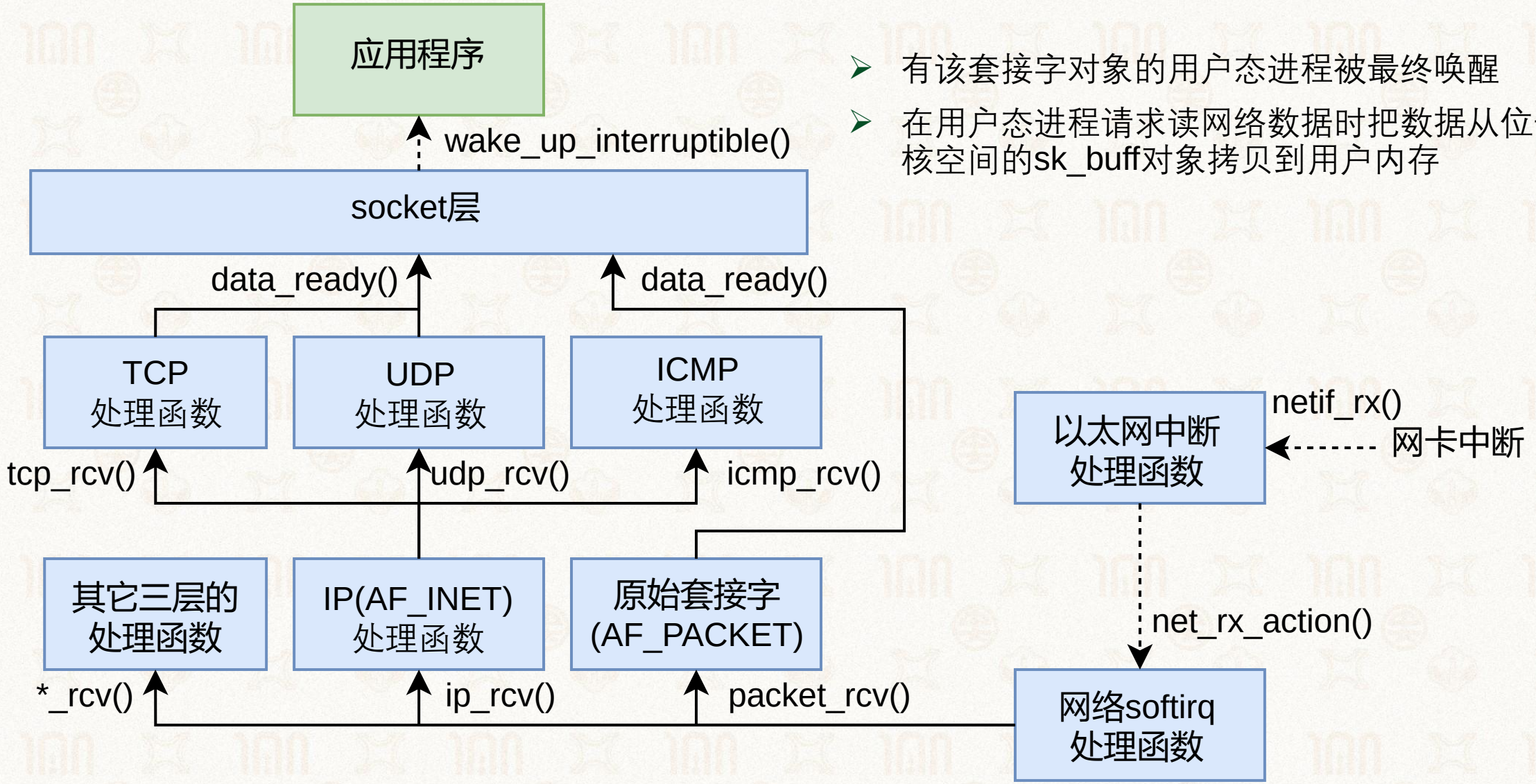


Linux网络数据包接收处理过程：函数视角

- 驱动层将sk_buff上抛给内核协议栈，
- 或者原始套接字可直接查找到套接字对象

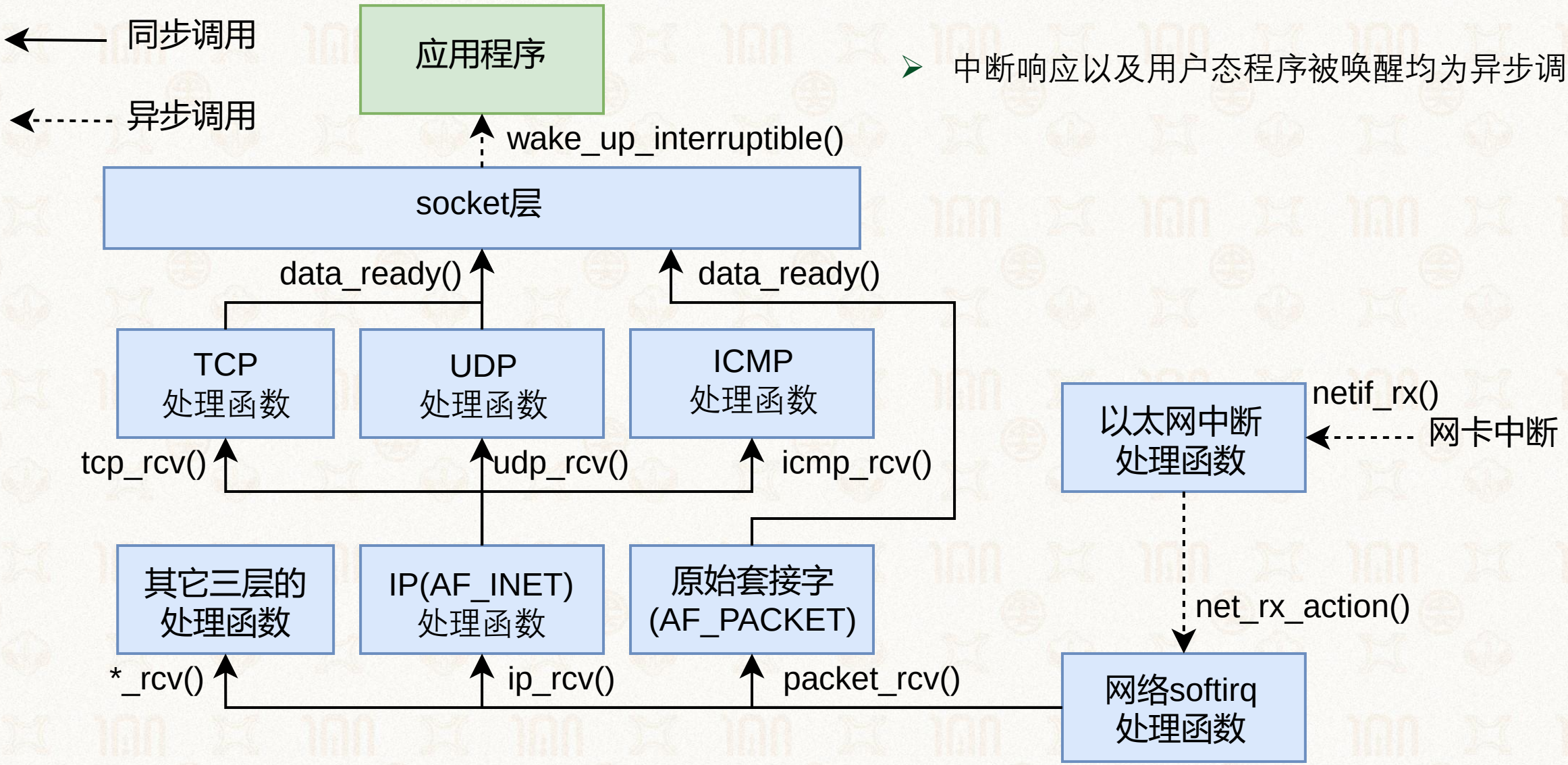


Linux网络数据包接收处理过程：函数视角



- 有该套接字对象的用户态进程被最终唤醒
- 在用户态进程请求读网络数据时把数据从位于内核空间的sk_buff对象拷贝到用户内存

Linux网络数据包接收处理过程：函数视角



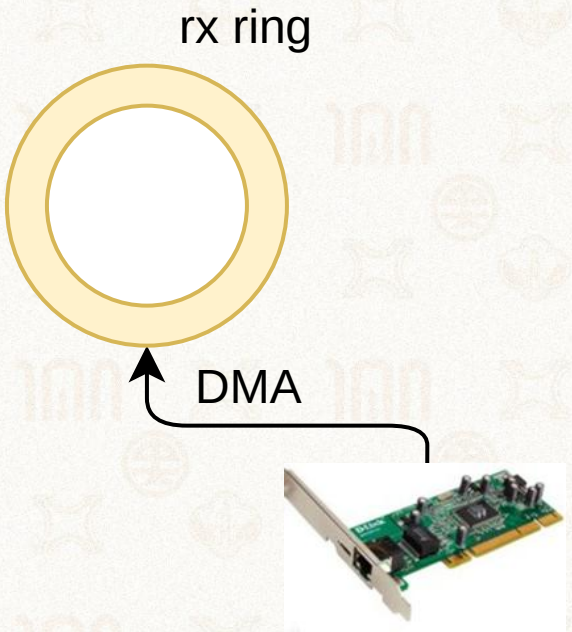
- 中断响应以及用户态程序被唤醒均为异步调用

大纲

- 越俎代庖部分
  - 网络协议的分层模型
  - 套接字模型
- 网络驱动模型
- Linux系统收包过程
  - 函数视角
  - 数据视角
- Linux系统发包过程
- 网络处理性能优化
  - 挑战
  - 数据面控制面分离
- Intel DPDK 软件优化方案
  - 总体框架
  - 无锁环
  - 内存池
  - 其它模块
  - 扩展框架
- 硬件优化方案

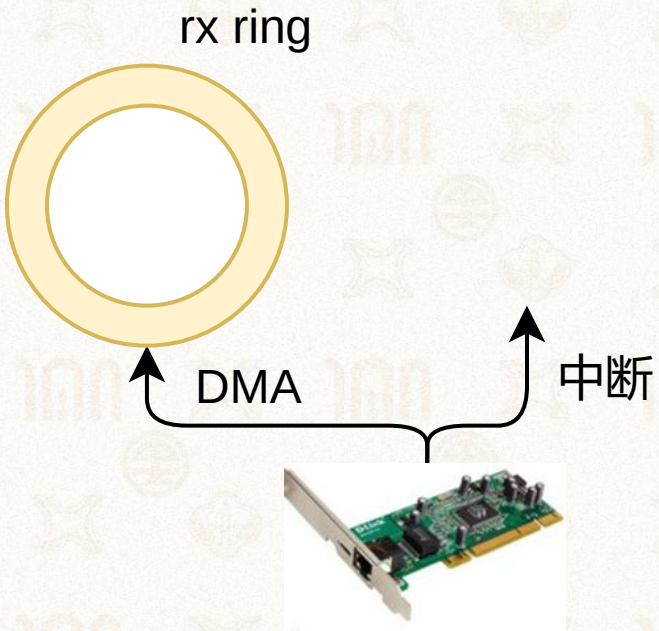
Linux网络数据包接收处理过程：数据视角

- 网卡收到到数据包（以太网帧）：
  - DMA 将数据帧传送至内核内存中的rx_ring



Linux网络数据包接收处理过程：数据视角

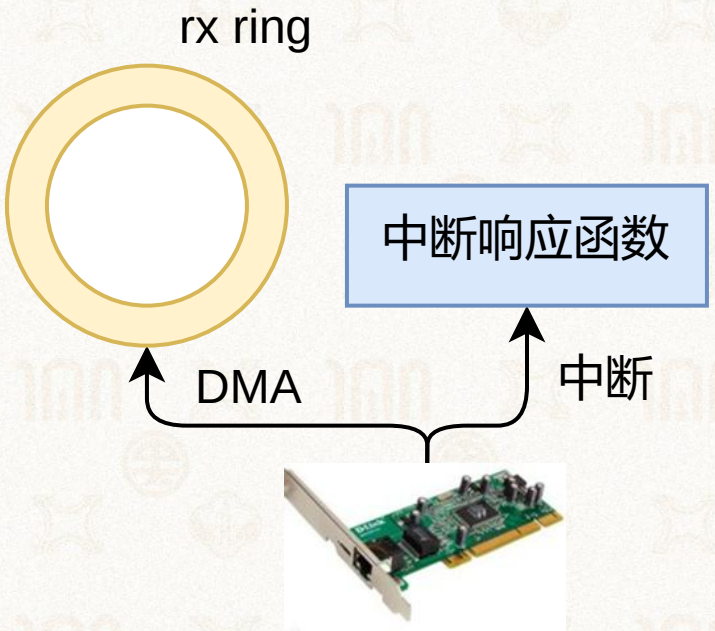
- 网卡收到到数据包（以太网帧）：
  - DMA 将数据帧传送至内核内存中的rx_ring
  - 网卡中断被触发



Linux网络数据包接收处理过程：数据视角

- 网卡收到到数据包（以太网帧）：
  - DMA 将数据帧传送至内核内存中的rx_ring
  - 网卡中断被触发

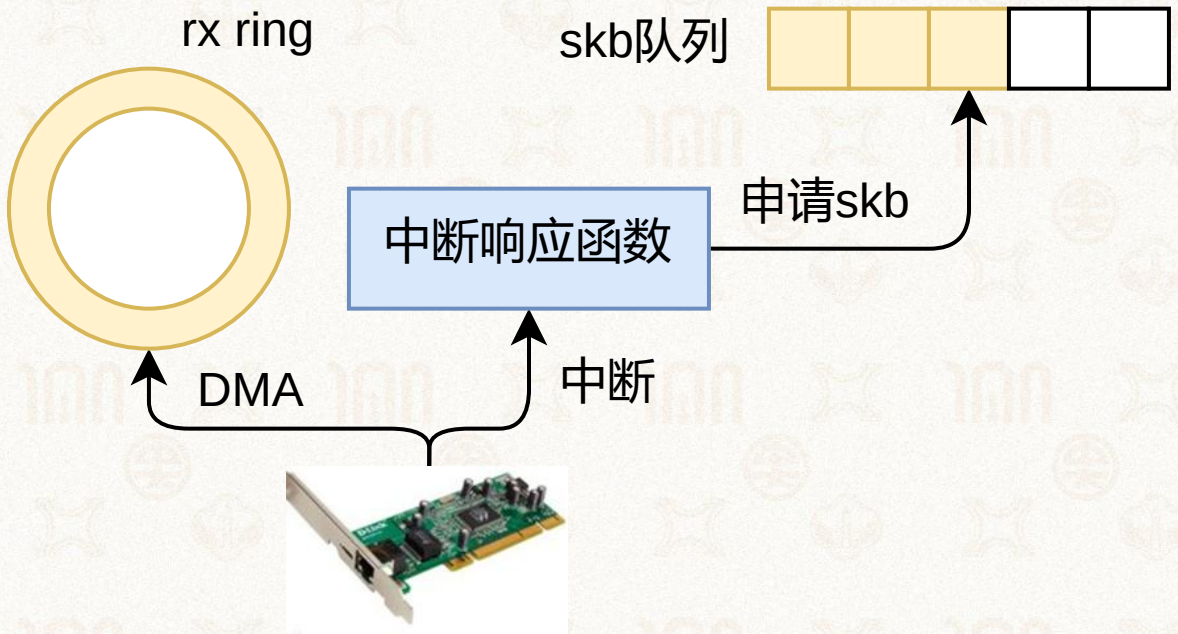
- CPU收到网卡中断，调用网卡ISR（上半部）



Linux网络数据包接收处理过程：数据视角

- 网卡收到到数据包（以太网帧）：
  - DMA 将数据帧传送至内核内存中的rx_ring
  - 网卡中断被触发

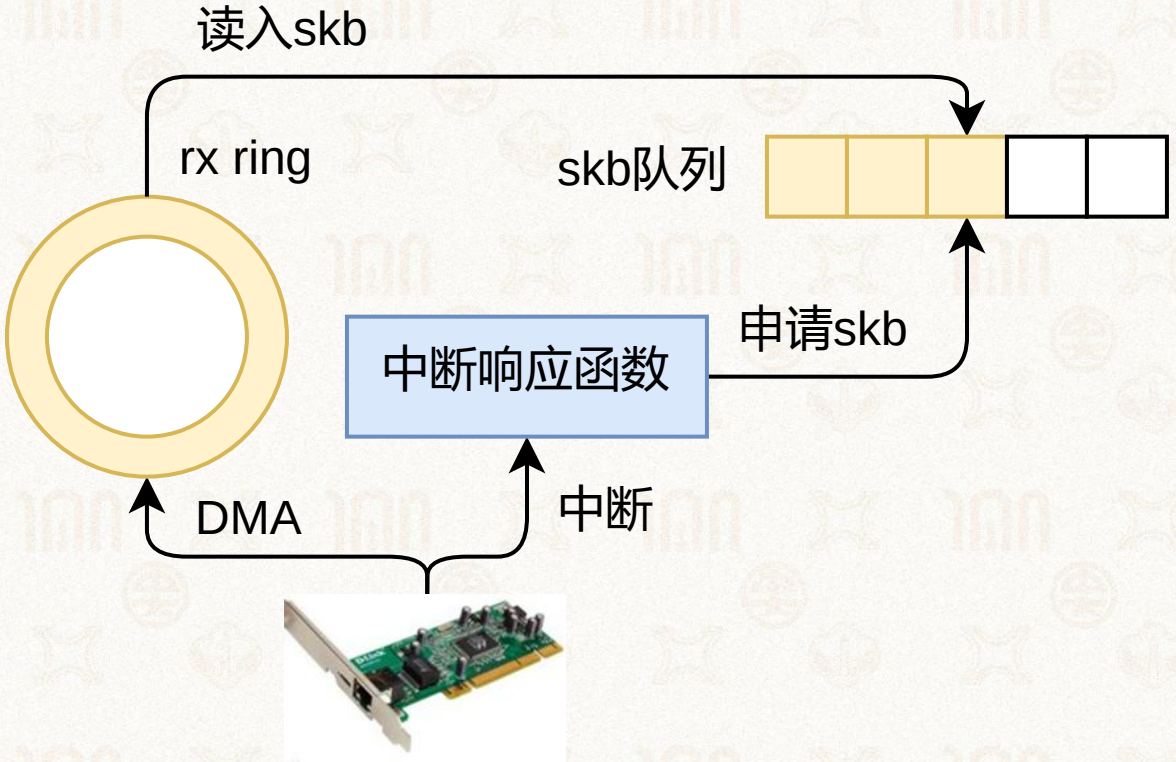
- CPU收到网卡中断，调用网卡ISR（上半部）：
  - 分配 sk_buff (skb) 数据结构，负责管理rx_ring中的数据包



Linux网络数据包接收处理过程：数据视角

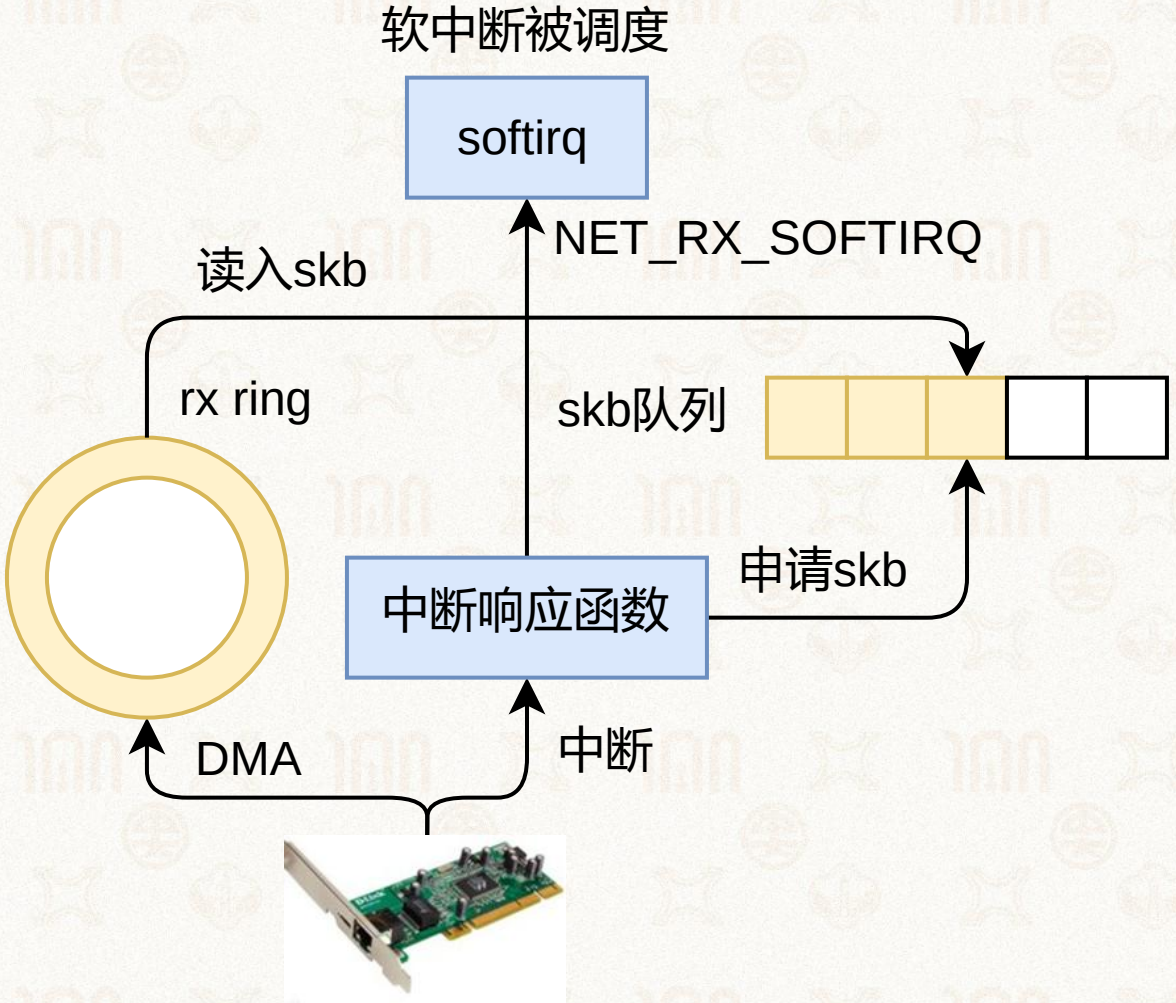
- 网卡收到到数据包（以太网帧）：
  - DMA 将数据帧传送至内核内存中的rx_ring
  - 网卡中断被触发

- CPU收到网卡中断，调用网卡ISR（上半部）：
  - 分配 sk_buff (skb) 数据结构，负责管理rx_ring中的数据包
  - 将skb包入队（input_pkt_queue）



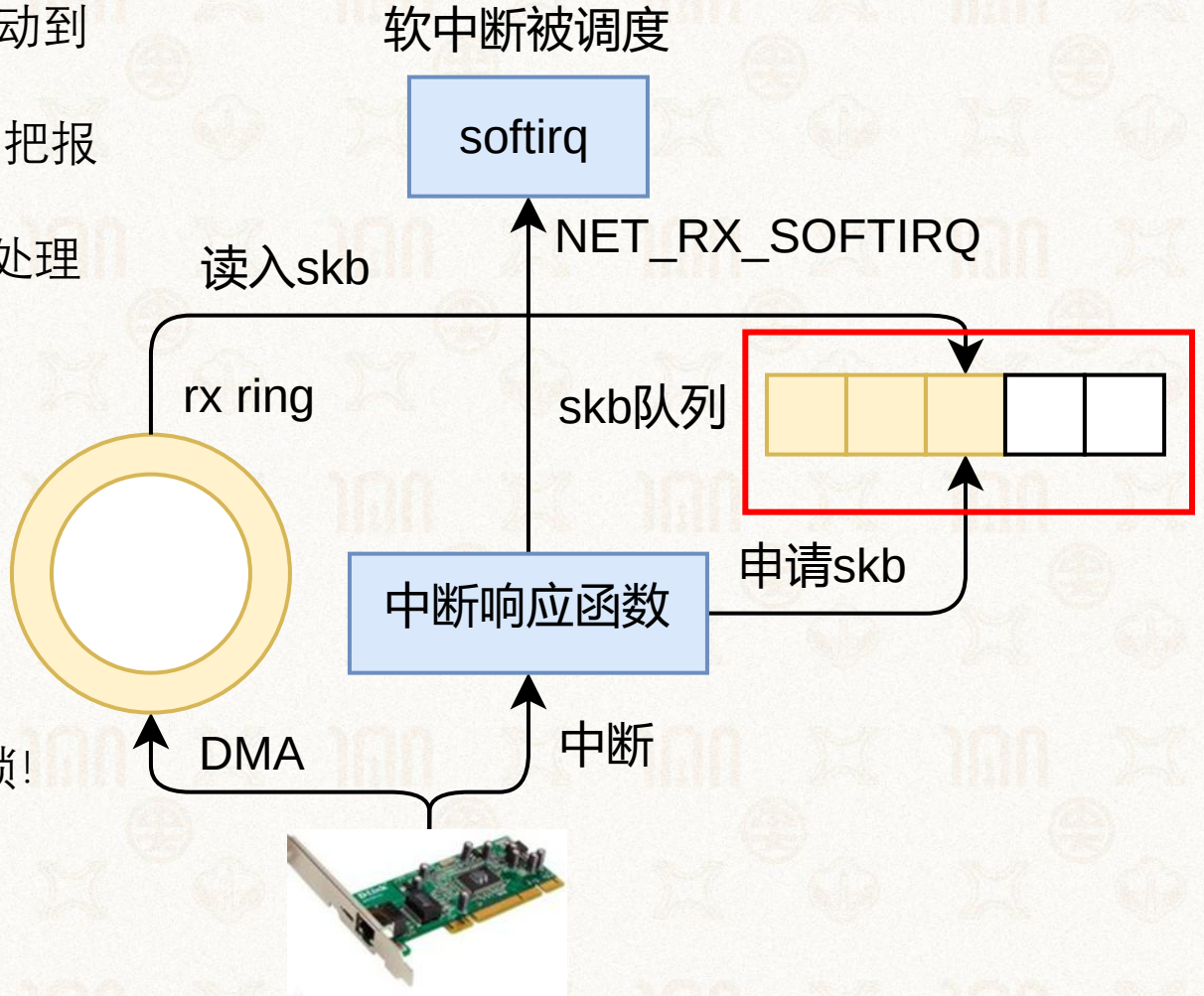
Linux网络数据包接收处理过程：数据视角

- 网卡收到到数据包（以太网帧）：
  - DMA 将数据帧传送至内核内存中的rx_ring
  - 网卡中断被触发
- CPU收到网卡中断，调用网卡ISR（上半部）：
  - 分配 sk_buff (skb) 数据结构，负责管理rx_ring中的数据包
  - 将skb包入队（input_pkt_queue）
- 上半部发出一个软中断（NET_RX_SOFTIRQ）：
  - 通知内核处理skb包



Linux网络数据包接收处理过程：数据视角

- 进入软中断处理流程（下半部）：
  - 把 input_pkt_queue 的skb移动到process_queue 处理队列中
  - 根据报文类型(ARP或是IP)，把报文递交给对应协议进行处理
  - 调用网络层协议的的handler处理skb包
- 移动skb：操作指针
- 如果接收队列已满（input_pkt_queue）：
  - 丢弃后续数据：可能发生活锁!

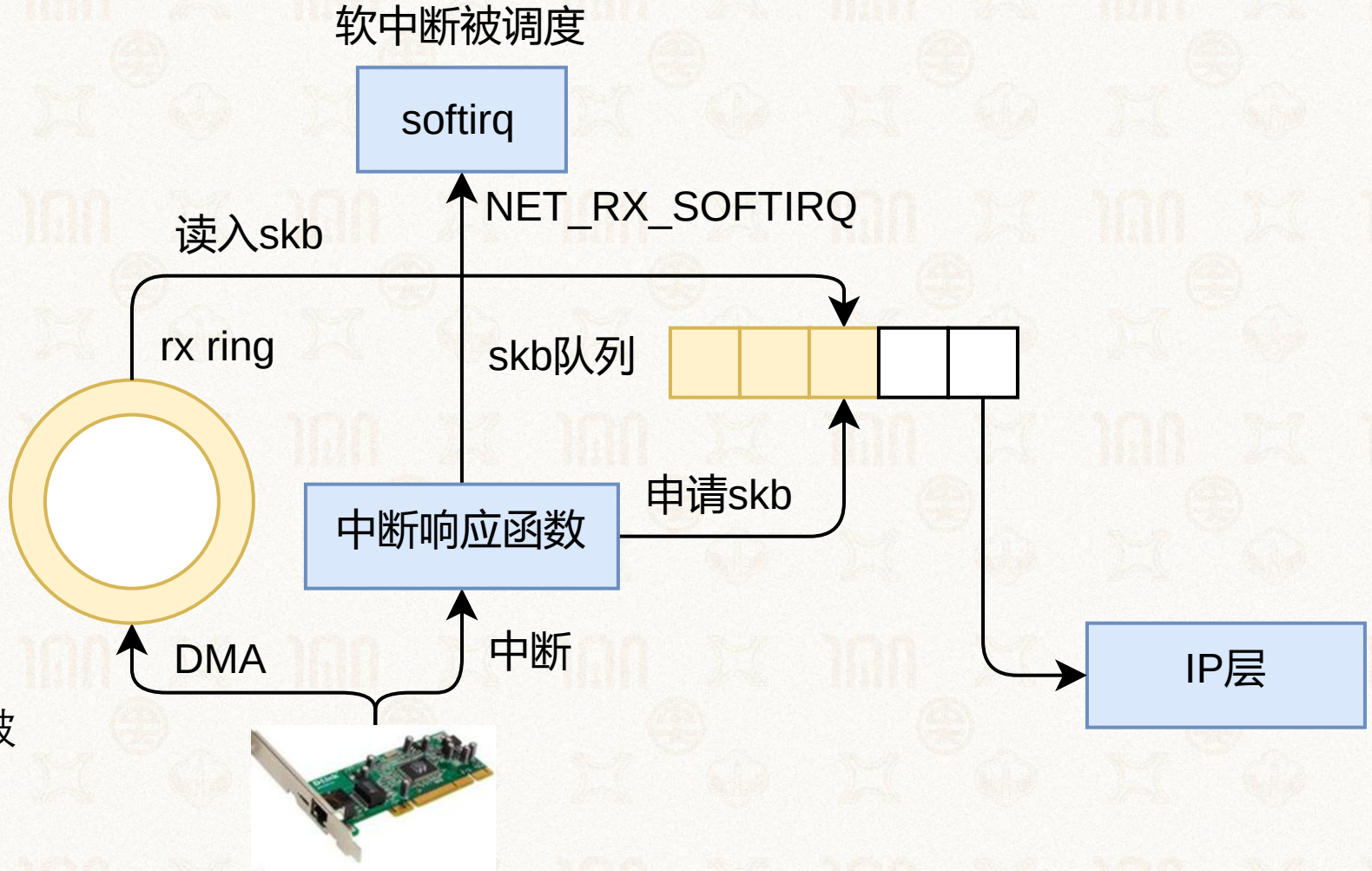


Linux网络数据包接收处理过程：数据视角

- 网络层：

- IP 层的入口函数 ip_rcv()：
  - 检查是否为IP包
  - 检查IP版本号
  - 对完整性（checksum）和长度进行检查

- ip_rcv()结束调用 ip_router_input()，进行路由处理
  - 查找路由
  - 决定该数据包（报文）是发到本机，还是被转发，或是被丢弃



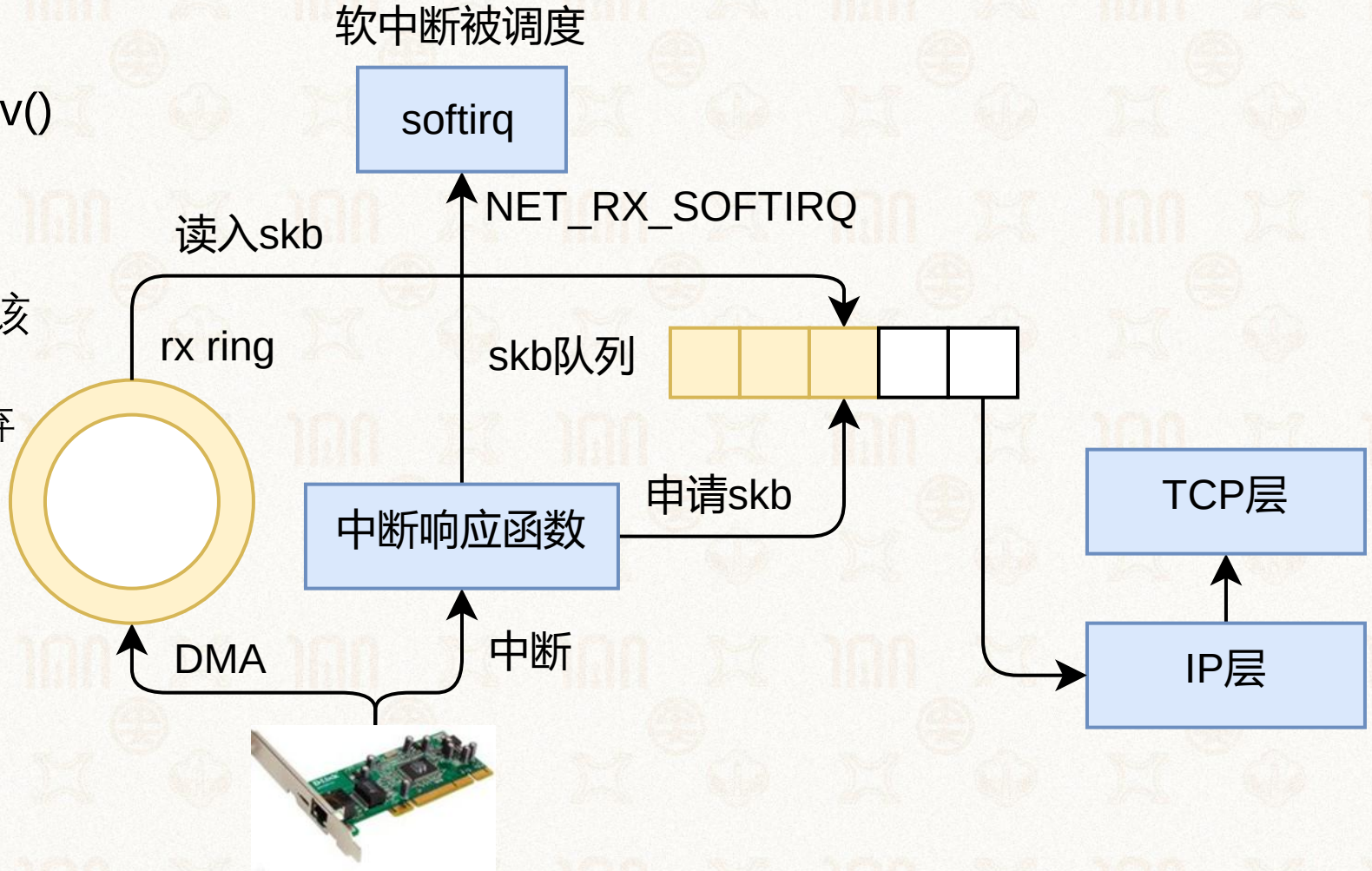
ip_rcv: https://elixir.bootlin.com/linux/v5.16.14/source/net/ipv4/ip_input.c#L531

Linux网络数据包接收处理过程：数据视角

- 传输层：

- 传输层的处理入口 tcp_v4_rcv()
  - 对 TCP header 进行检查

- 调用 _tcp_v4_lookup，查找该数据包对应的open socket
  - 如果找不到，该数据包被丢弃
  - 否则检查 socket 和 connection 的状态



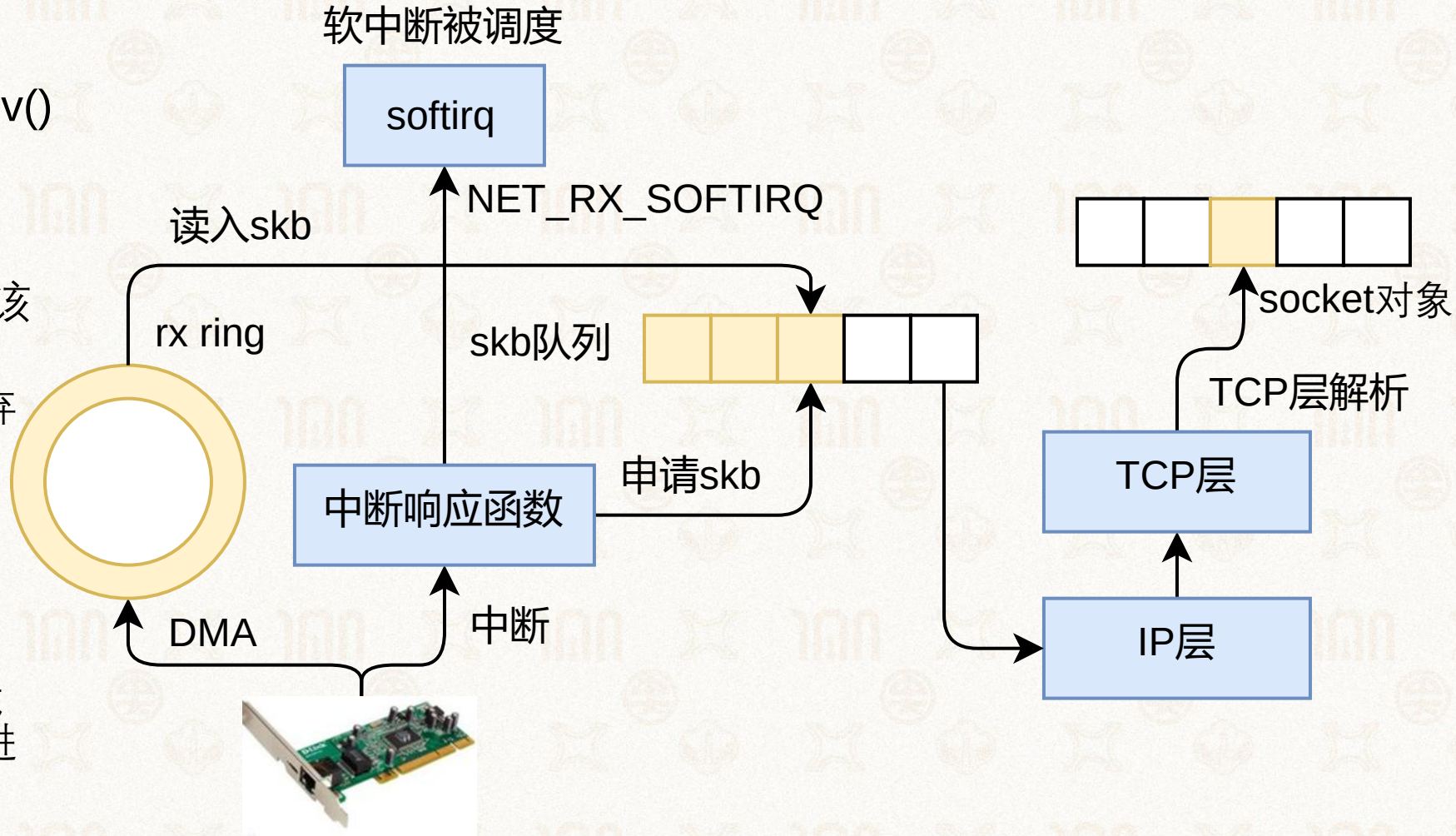
Linux网络数据包接收处理过程：数据视角

- 传输层：

- 传输层的处理入口 tcp_v4_rcv()
  - 对 TCP header 进行检查

- 调用 _tcp_v4_lookup，查找该数据包对应的open socket
  - 如果找不到，该数据包被丢弃
  - 否则检查 socket 和 connection 的状态

- socket 和 connection 正常
  - 调用 tcp_prequeue() 使tcp载荷从内核进入用户空间，放进 socket 接收队列

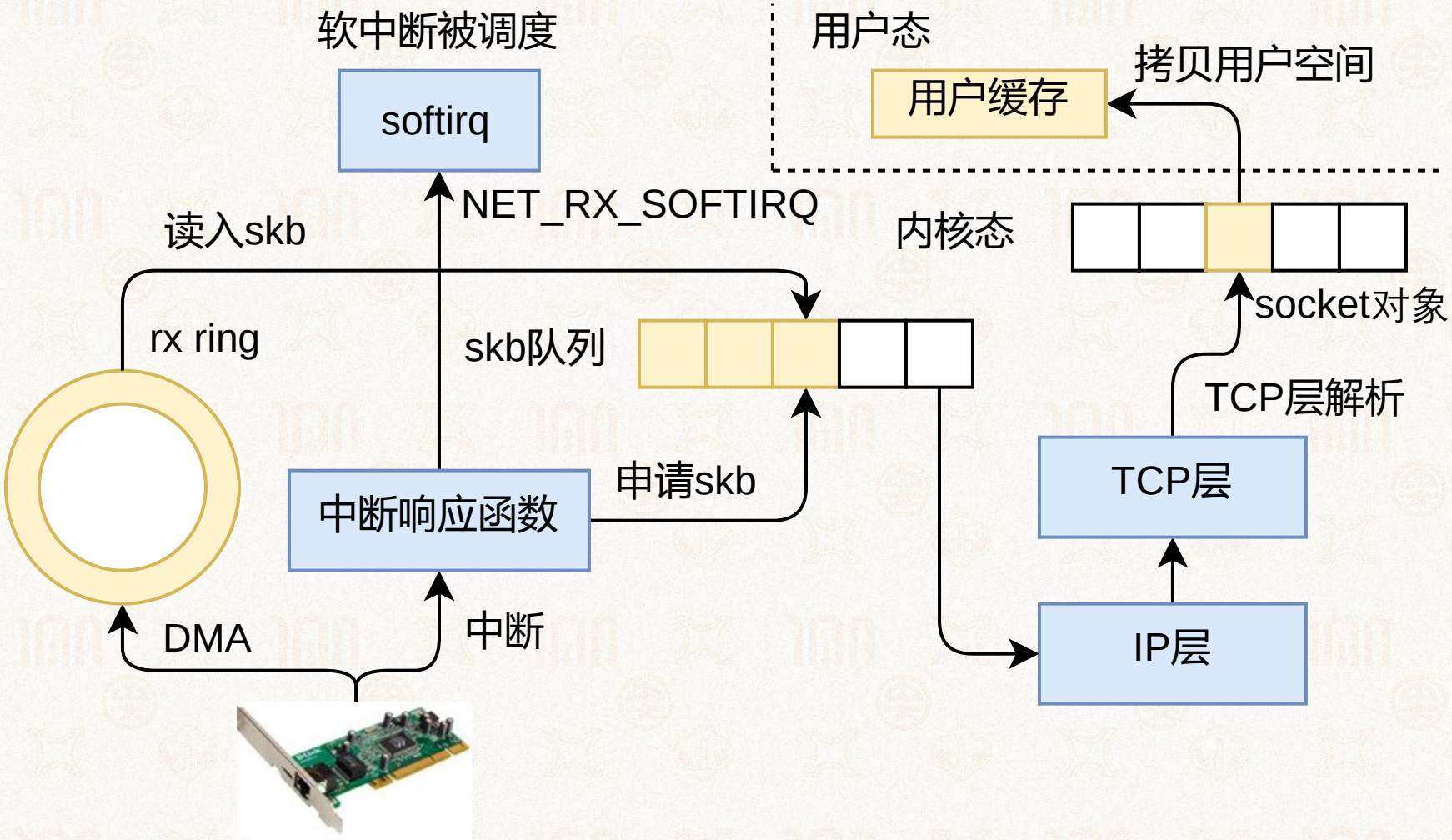


Linux网络数据包接收处理过程：数据视角

- 应用层：

- socket 被唤醒，调用 system call，并最终调用 tcp_recvmsg()，从 socket 接收队列 中获取数据

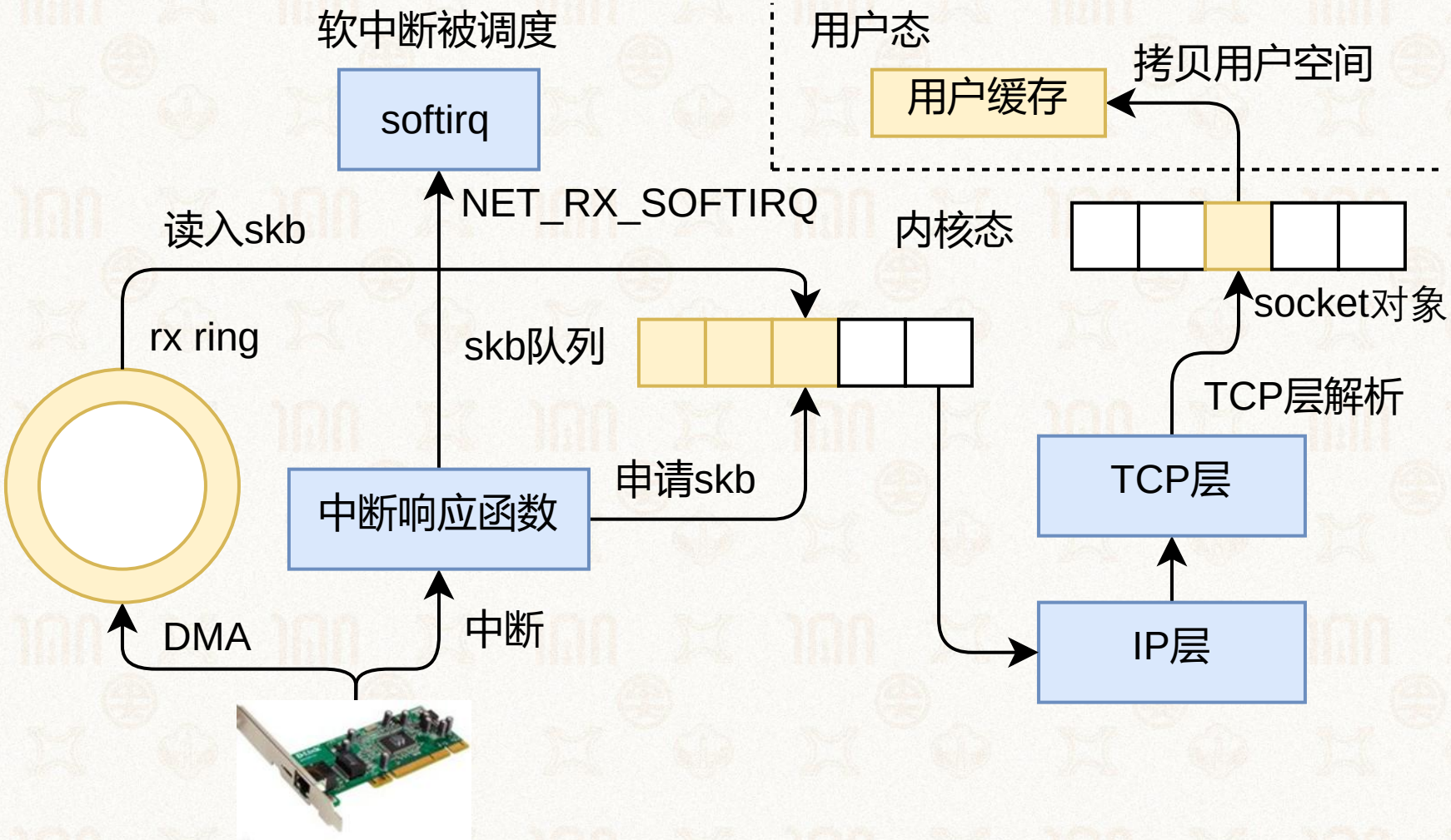
- 用户态调用 read 或者 recvfrom ，转化为 sys_recvfrom 调用
  - 对 TCP 来说，调用 tcp_recvmsg()
  - 该函数从socket buffer 中拷贝数据到用户缓存(user buffer)



tcp_recvmsg(): https://elixir.bootlin.com/linux/v5.16.14/source/net/ipv4/tcp.c#L2527

网络包的管理

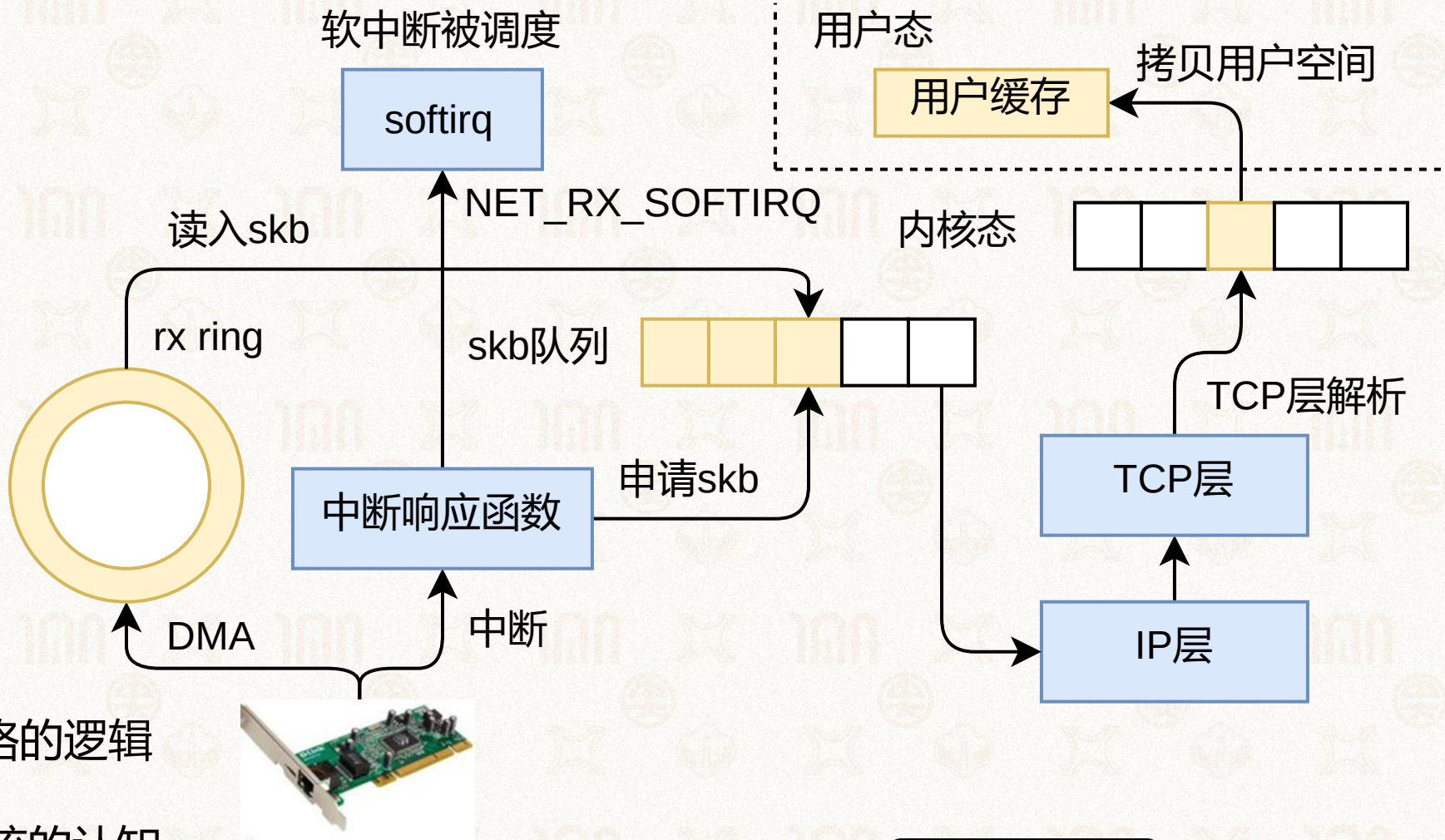
- 要求高效地处理分层
  - 发包时需要不断添加新的头部，收包则相反
- 避免连续存放网络包
  - 移动过程中数据拷贝会有很大开销
- sk_buff (skb)
  - 让分层的处理变得高效：零拷贝
  - 快速申请和释放内存：防止内存碎片



主观题 10分

设置

在Linux内核协议栈的收包过程中，从网卡到用户进程，一个网络包的数据至少要拷贝几次？



TCP几次握手是在考计算机网络的逻辑

而这道题集中反映了对操作系统的认知

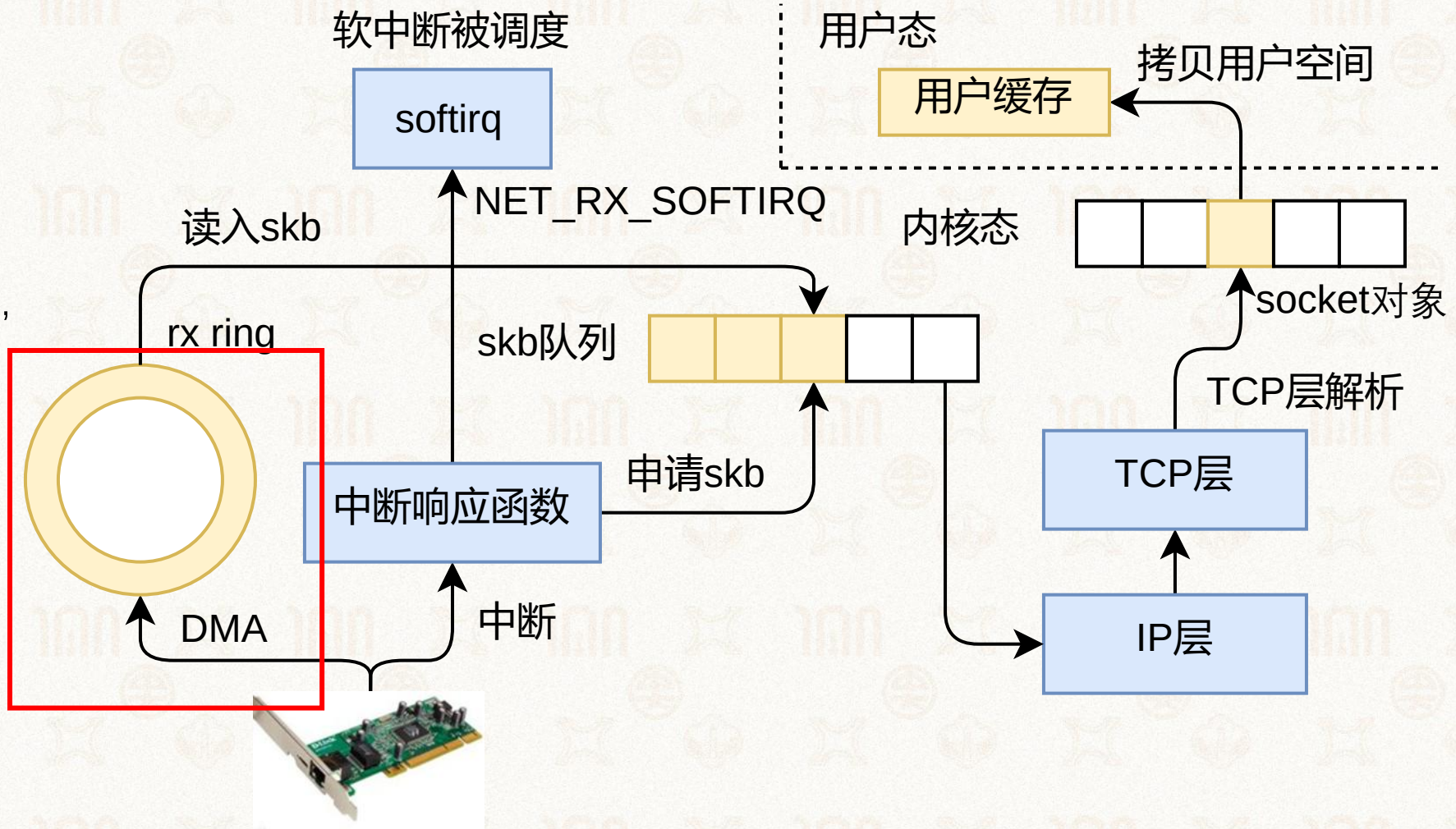
作答

网络包的管理：一个经典的面试题

- 在Linux内核协议栈的收包过程中，从网卡到用户进程，一个网络包的数据至少要拷贝几次？

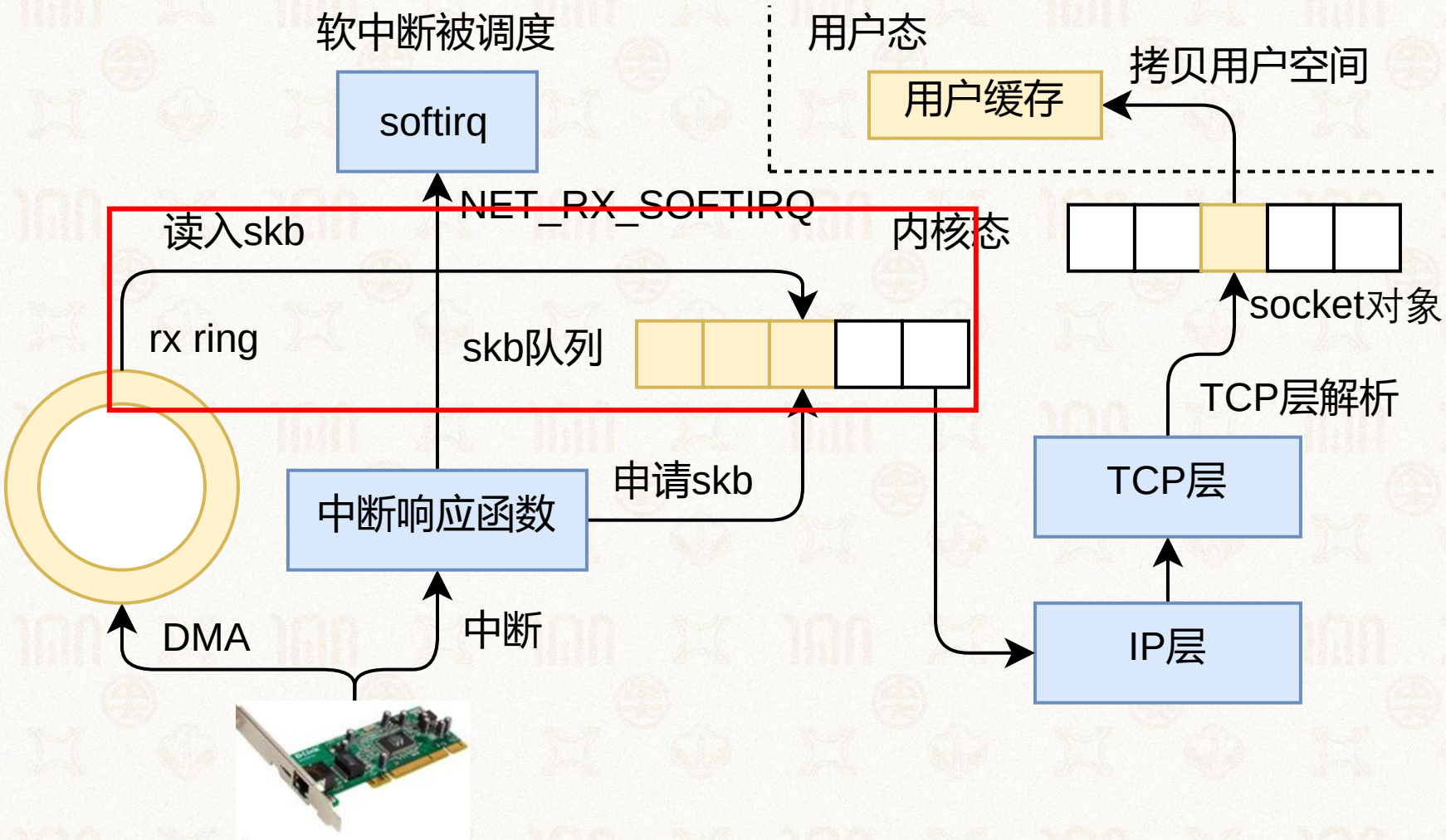
- 网络数据在网卡缓存中准备好时，会首先发起DMA操作，复制数据到内核态的内存缓冲区

- 这是一次数据拷贝



网络包的管理：一个经典的面试题

- 在Linux内核协议栈的收包过程中，从网卡到用户进程，一个网络包的数据至少要拷贝几次？
- 由于skb只保存指针，不保存数据
- 这里可以省略一次数据拷贝
- 但过时/错误的资料会说这里也有一次数据拷贝

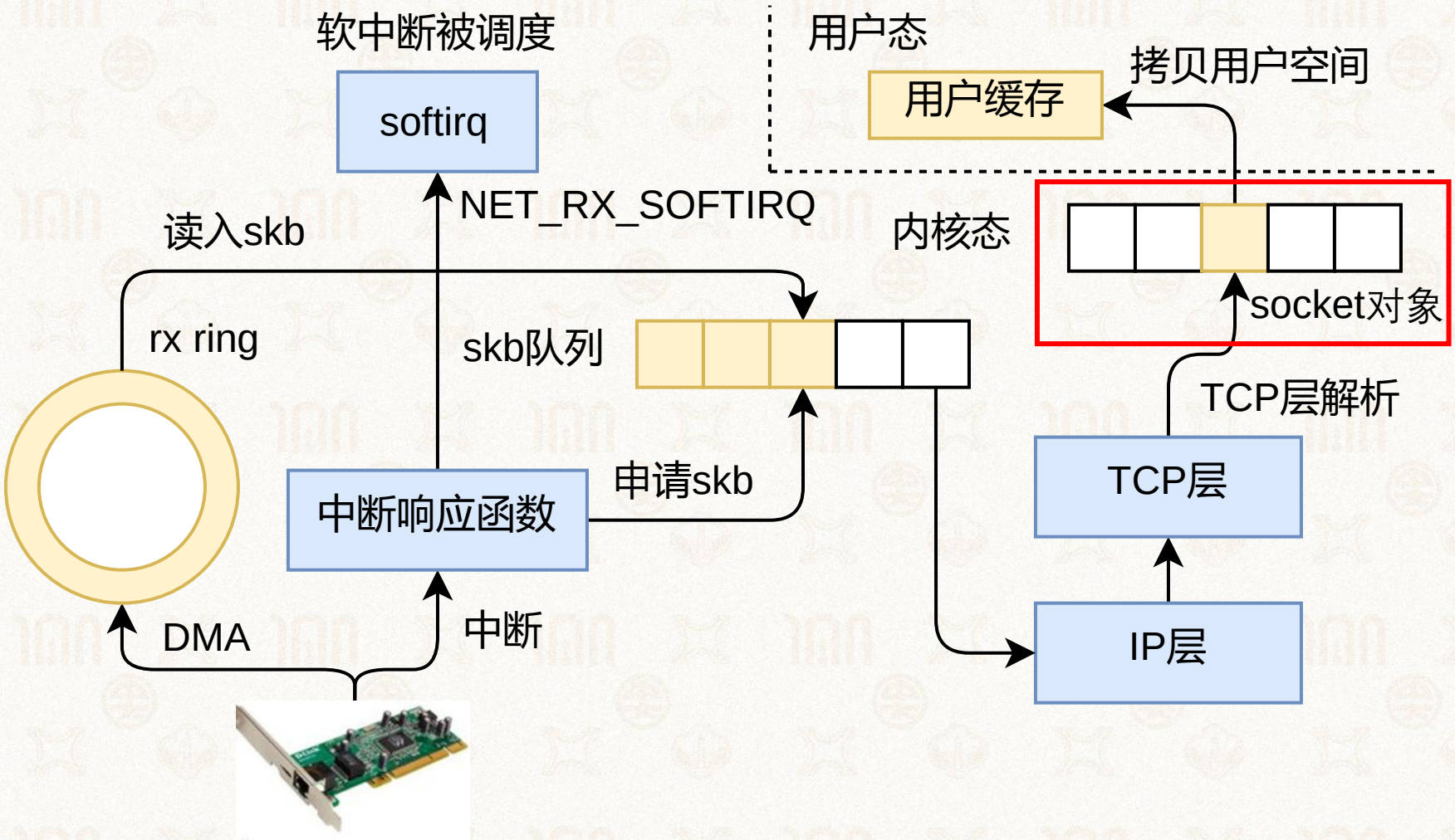


网络包的管理：一个经典的面试题

- 在Linux内核协议栈的收包过程中，从网卡到用户进程，一个网络包的数据至少要拷贝几次？

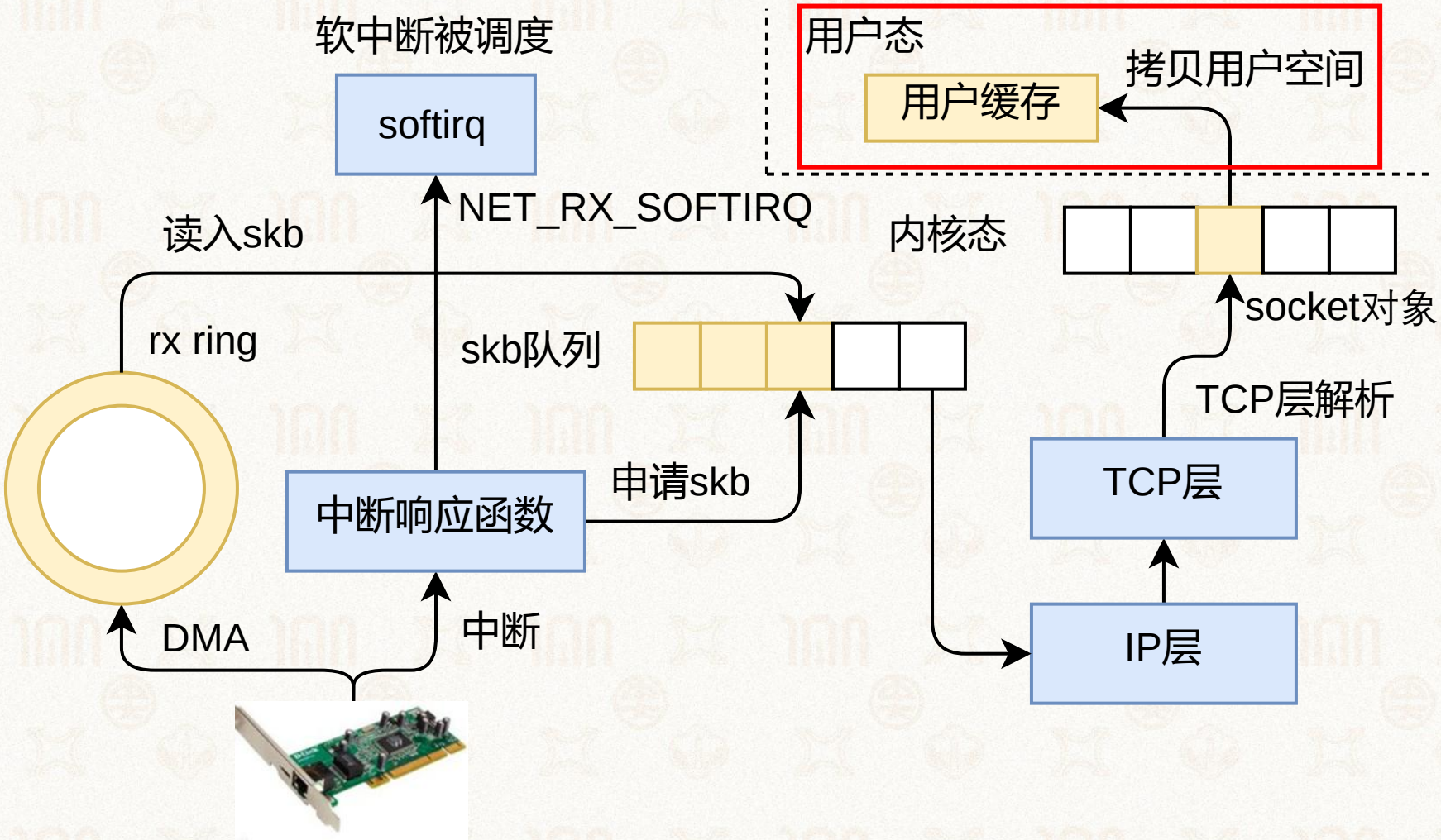
- 只要多层处理足够快，且接收(rx ring)缓存足够大，socket对象就不用自己维护数据内容，直接复用缓存数据

- 这样也可以省略一次数据拷贝



网络包的管理：一个经典的面试题

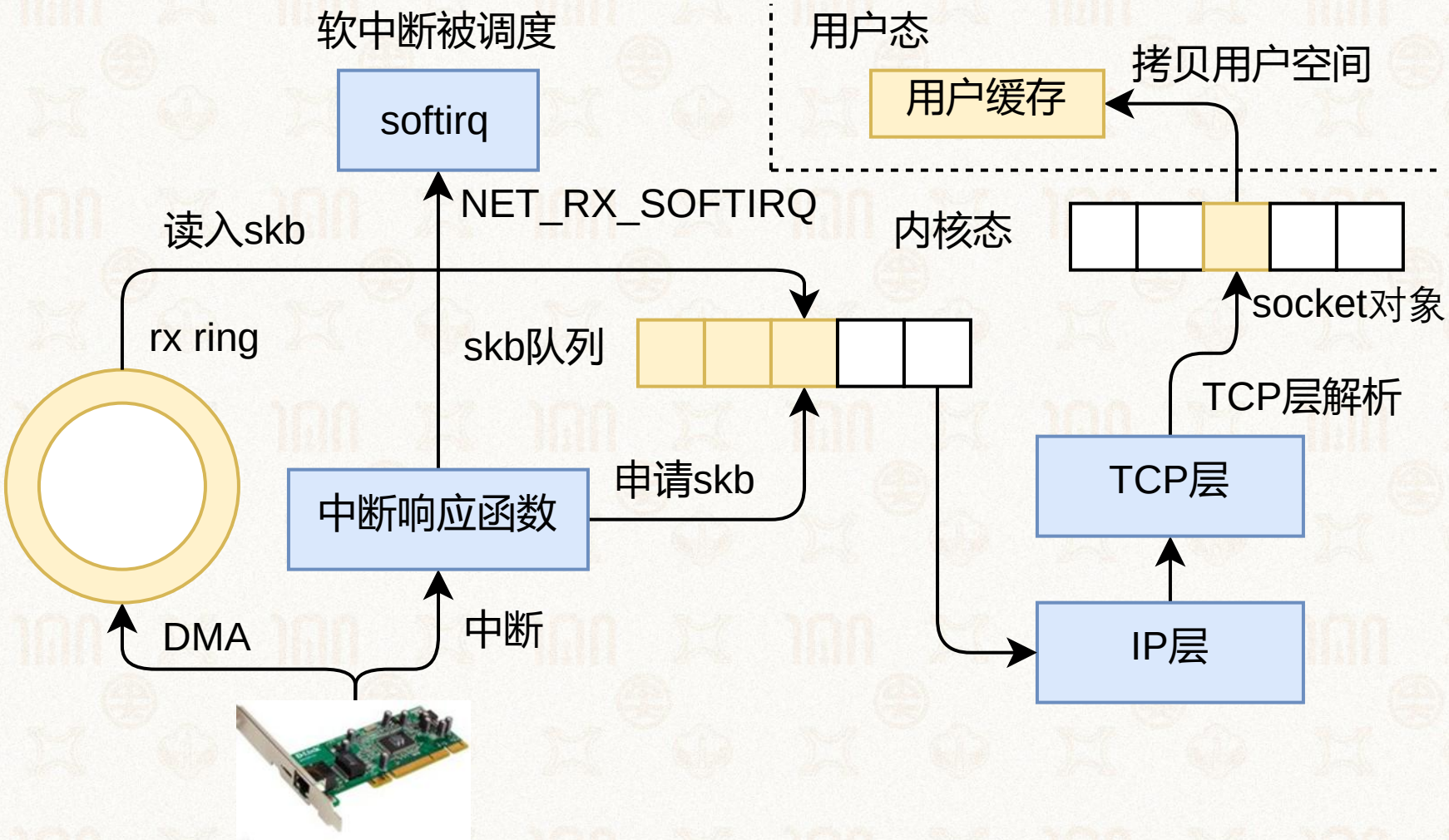
- 在Linux内核协议栈的收包过程中，从网卡到用户进程，一个网络包的数据至少要拷贝几次？
- 把内核态的数据传递到用户态的缓存，这里不能再缓存复用了
- 因为内核态数据和用户态数据需要隔离
- 这是一次数据拷贝



网络包的管理：一个经典的面试题

- 在Linux内核协议栈的收包过程中，从网卡到用户进程，一个网络包的数据至少要拷贝几次？

- 答案是至少两次数据拷贝
  - DMA一次
  - 内核态到用户态转换一次



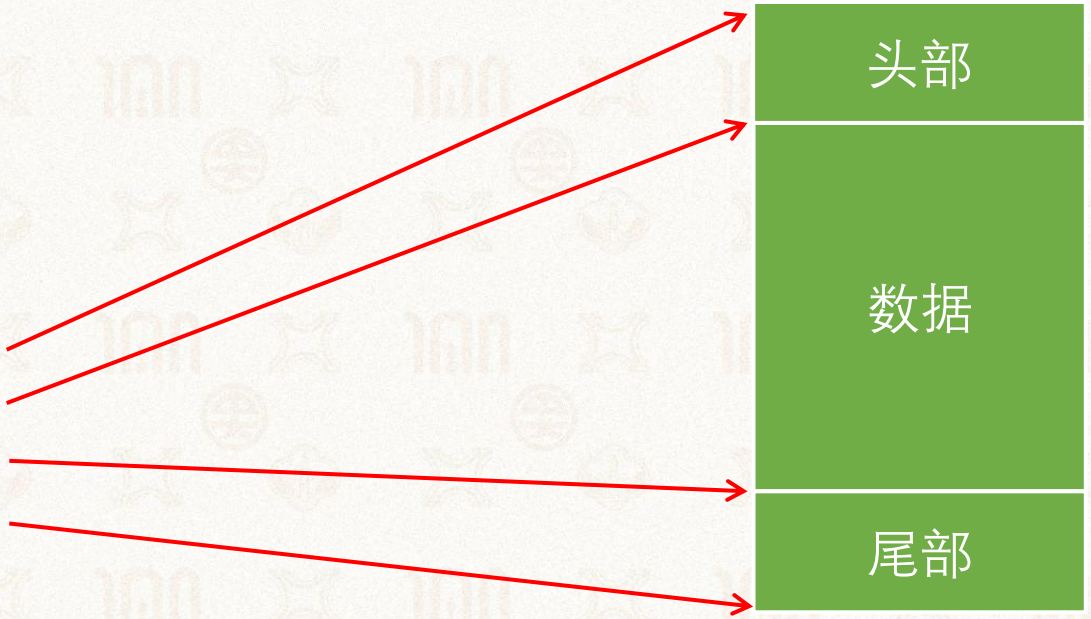
大纲

- 越俎代庖部分
  - 网络协议的分层模型
  - 套接字模型
- 网络驱动模型
- Linux系统收包过程
  - 函数视角
  - 数据视角
- Linux系统发包过程
- 网络处理性能优化
  - 挑战
  - 数据面控制面分离
- Intel DPDK 软件优化方案
  - 总体框架
  - 无锁环
  - 内存池
  - 其它模块
  - 扩展框架
- 硬件优化方案

sk_buff中的数据

```
typedef unsigned char *sk_buff_data_t;

struct sk_buff {
    union {
        struct {
            struct sk_buff* next;
            struct sk_buff* prev;
            // ...
        };

    };
    unsigned int len;
    unsigned int data_len;

    unsigned char*  head;
    unsigned char*  data;
    sk_buff_data_t  tail;
    sk_buff_data_t  end;
    // ...
};
```

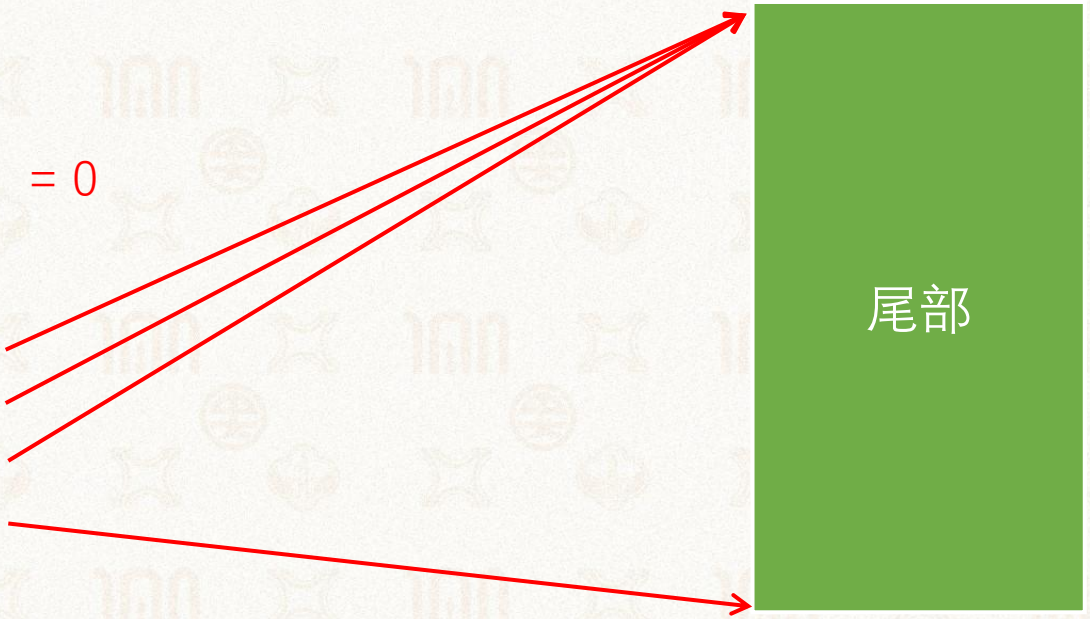
- sk_buff本身不存储报文
  - 通过指针指向真正的报文内存空间
- 在各层传递时
  - 只需调整指针相应位置即可



sk_buff：发送数据包

```
typedef unsigned char *sk_buff_data_t;

struct sk_buff {
    union {
        struct {
            struct sk_buff* next;
            struct sk_buff* prev;
            // ...
        };

    };
    unsigned int len;       = 0
    unsigned int data_len;

    unsigned char*  head;
    unsigned char*  data;
    sk_buff_data_t  tail;
    sk_buff_data_t  end;
    // ...
};
```

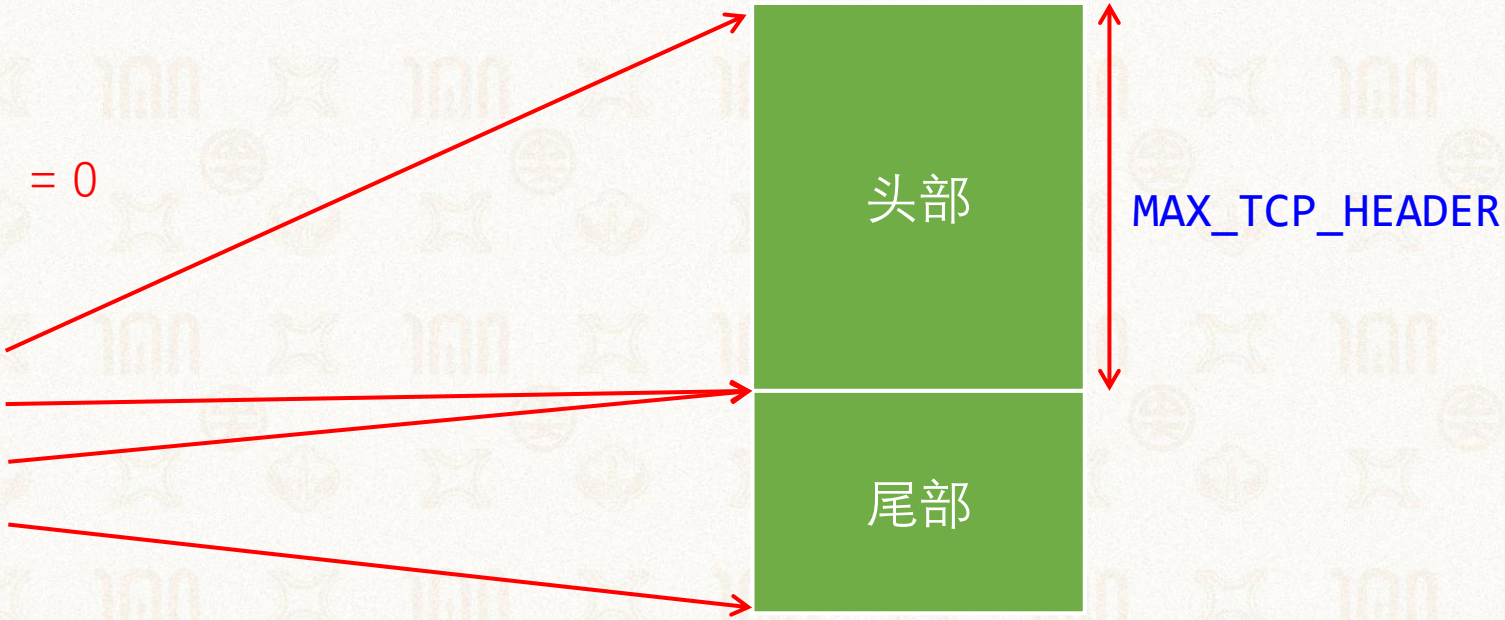
- Step1：
  - TCP层发数据时，首先用alloc_skb申请缓冲区



sk_buff：发送数据包

```
typedef unsigned char *sk_buff_data_t;

struct sk_buff {
    union {
        struct {
            struct sk_buff* next;
            struct sk_buff* prev;
            // ...
        };

    };
    unsigned int len;       = 0
    unsigned int data_len;

    unsigned char*  head;
    unsigned char*  data;
    sk_buff_data_t  tail;
    sk_buff_data_t  end;
    // ...
};
```

- Step2：
  - TCP用skb_reserve来保留足量空间存储所有头部

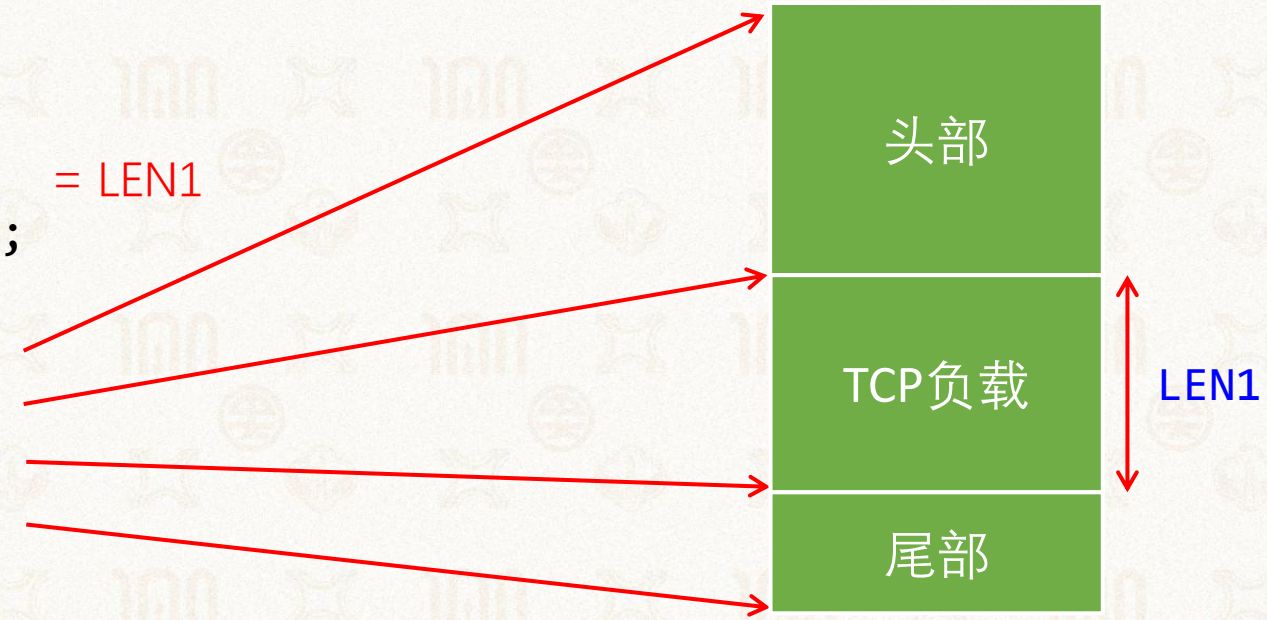



```
#define MAX_TCP_HEADER  L1_CACHE_ALIGN(128 + MAX_HEADER)
```

sk_buff：发送数据包

```
typedef unsigned char *sk_buff_data_t;

struct sk_buff {
    union {
        struct {
            struct sk_buff* next;
            struct sk_buff* prev;
            // ...
        };

    };
    unsigned int len;          = LEN1
    unsigned int data_len;

    unsigned char*  head;
    unsigned char*  data;
    sk_buff_data_t  tail;
    sk_buff_data_t  end;
    // ...
};
```

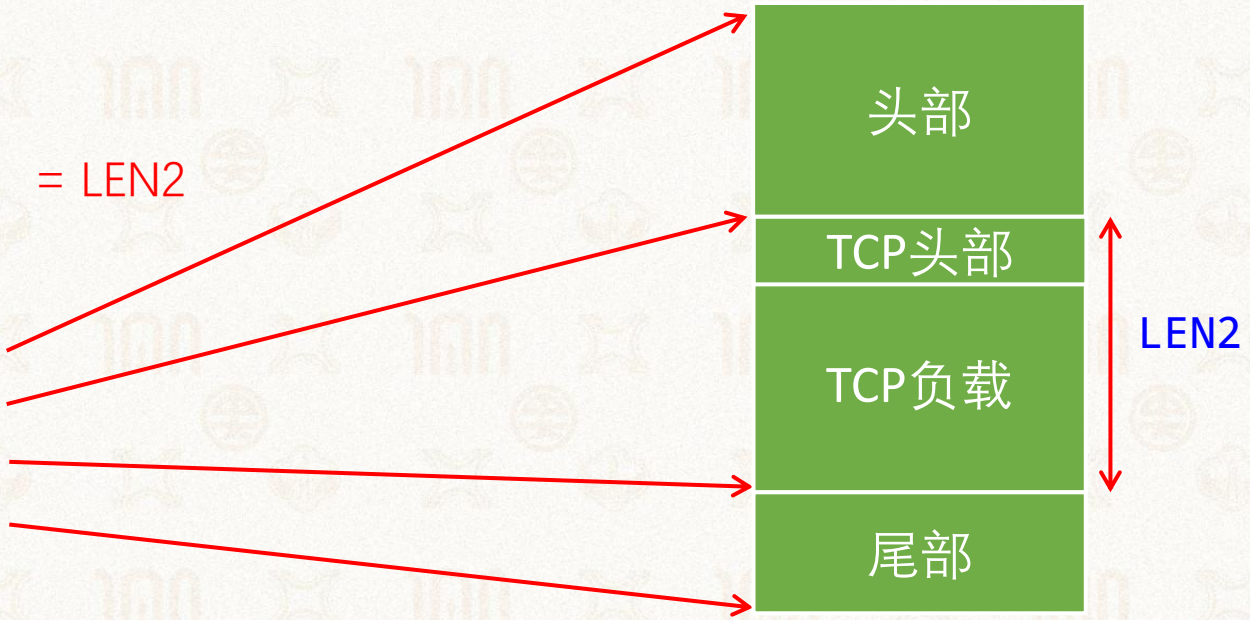
- Step3：
  - TCP层填入TCP负载（应用层数据）



sk_buff：发送数据包

```
typedef unsigned char *sk_buff_data_t;

struct sk_buff {
    union {
        struct {
            struct sk_buff* next;
            struct sk_buff* prev;
            // ...
        };

    };
    unsigned int len;          = LEN2
    unsigned int data_len;

    unsigned char*  head;
    unsigned char*  data;
    sk_buff_data_t  tail;
    sk_buff_data_t  end;
    // ...
};
```

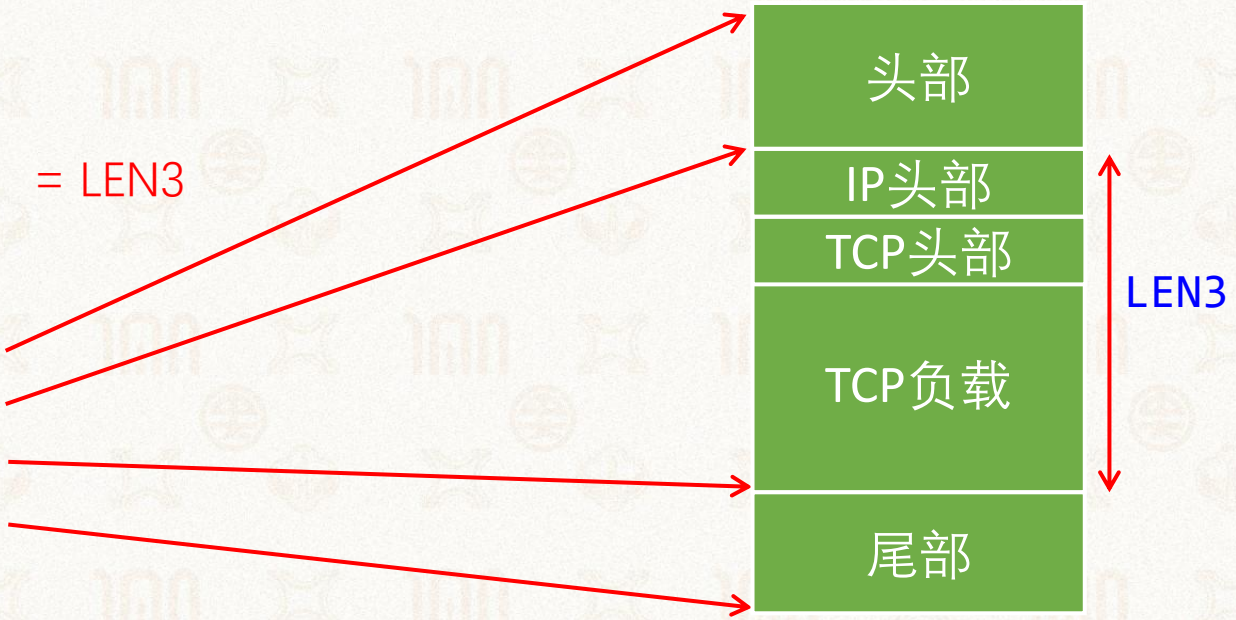
- Step4：
  - TCP层填入TCP头部



sk_buff：发送数据包

```
typedef unsigned char *sk_buff_data_t;

struct sk_buff {
    union {
        struct {
            struct sk_buff* next;
            struct sk_buff* prev;
            // ...
        };

    };
    unsigned int len;          = LEN3
    unsigned int data_len;

    unsigned char*  head;
    unsigned char*  data;
    sk_buff_data_t  tail;
    sk_buff_data_t  end;
    // ...
};
```

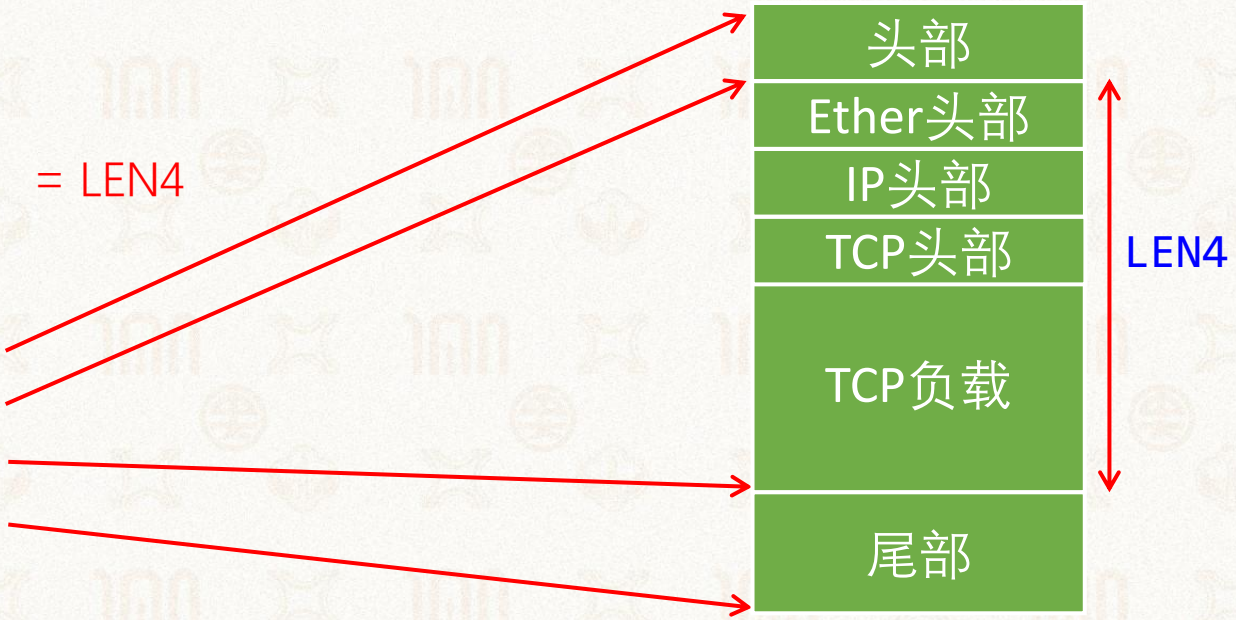
- Step5：
  - 传给IP层，并添加IP头部



sk_buff：发送数据包

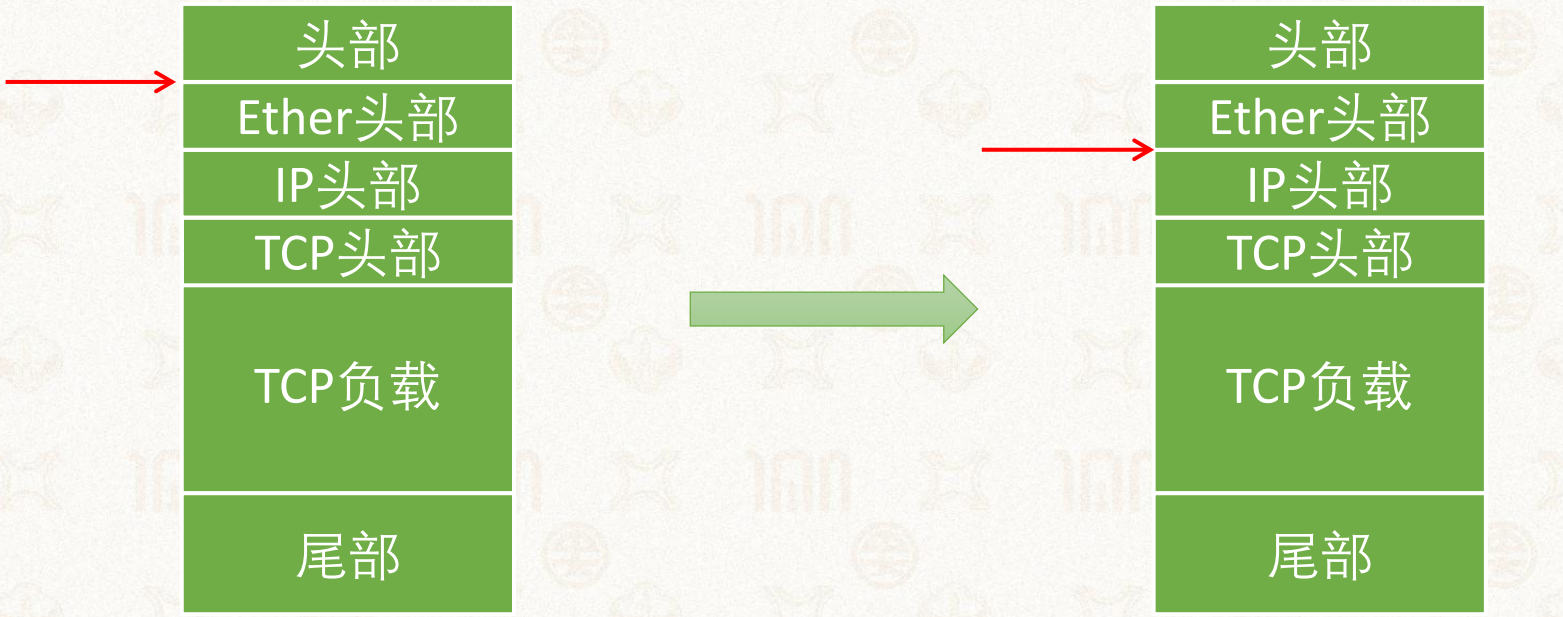
```
typedef unsigned char *sk_buff_data_t;

struct sk_buff {
    union {
        struct {
            struct sk_buff* next;
            struct sk_buff* prev;
            // ...
        };

    };
    unsigned int len;          = LEN4
    unsigned int data_len;

    unsigned char*  head;
    unsigned char*  data;
    sk_buff_data_t  tail;
    sk_buff_data_t  end;
    // ...
};
```

- Step6：
  - 传给链路层，并添加以太网头部



sk_buff：协议栈头部

- 当前层处理skb时：
  - 同时负责将下一层头部指针初始化好，并移动data指针



处理链路层数据时

即将交给网络层之前

sk_buff: control buffer

```
struct sk_buff {
    union {
        struct {
            struct sk_buff* next;
            struct sk_buff* prev;
            // ...
        };

    };

    char     cb[48] __aligned(8);

    unsigned int len;
    unsigned int data_len;

    sk_buff_data_t  tail;
    sk_buff_data_t  end;
    unsigned char*  head;
    unsigned char*  data;
    // ...
};
```

用于每层维护私有的信息

- 私有信息:
  - TCP中的Seq编号
  - ACK序列编号
  - 是否重发
  - 应答时间戳
- 更多信息详见下面代码:

tcp_skb_cb: https://elixir.bootlin.com/linux/v5.16.14/source/include/net/tcp.h#L842

sk_buff的组织

- 用双向链表对sk_buff进行管理：

```
struct sk_buff_head {
    /* These two members must be first to match sk_buff. */
    struct_group_tagged(sk_buff_list, list,
        struct sk_buff  *next;
        struct sk_buff  *prev;
    );
    __u32       qlen;
    spinlock_t  lock;
};
```

- Linux NAPI的批处理
  - 让网卡中断处理的下半部softirq积累足量的skb
  - NAPI周期性轮询，并一次性处理完所有skb（batching）
  - netif_receive_skb_list()

sk_buff: 操作函数

- 分配：`struct sk_buff *alloc_skb(unsigned int size, gfp_t priority);`
  - GFP_ATOMIC：分配过程不能被中断，用于中断上下文中分配内存
  - GFP_KERNEL：分配过程可以被中断，分配请求被放到等待队列中

- 浅拷贝：`struct sk_buff *skb_clone(struct sk_buff *skb, gfp_t priority);`
  - 克隆出新的sk_buff控制结构，指向同一报文（用于tcpdump等抓包工具）

- 深拷贝：`struct sk_buff *skb_copy(const struct sk_buff *skb, gfp_t priority);`
  - 同时复制sk_buff以及指向的报文（用于修改报文，如NAT地址转换）

- 释放：`void kfree_skb(struct sk_buff *skb);`

sk_buff: GFP_ATOMIC

- alloc_skb()
  - GFP_ATOMIC：分配过程不能被中断，用于中断上下文中分配内存
  - GFP_KERNEL：分配过程可以被中断，分配请求被放到等待队列中

- 在上半部ISR中：
  - GFP_ATOMIC保证分配过程不会再有ISR打断

- 在下半部软中断中：
  - GFP_ATOMIC告诉内核，如果申请失败，不能进入睡眠

sk_buff: 操作函数

- `void *skb_push(struct sk_buff *skb, unsigned int len);`
  - 在存储空间的头部增加存储网络报文的空间，用于发送网络报文时添加包头
- `void *skb_put(struct sk_buff *skb, unsigned int len);`
  - 在存储空间的尾部增加存储网络报文的空间，用于发送网络报文时追加数据
- `void *skb_pull(struct sk_buff *skb, unsigned int len);`
  - 使data指针指向下一层网络报文的头部，用于接收网络报文时调整头部
- `void skb_reserve(struct sk_buff *skb, int len);`
  - 在存储空间的头部预留len长度的空隙，用于为协议头部保留空间
- `void skb_trim(struct sk_buff *skb, unsigned int len);`
  - 将网络报文的长度缩减到len，用于丢弃网络报文尾部的填充值

skb_push: https://elixir.bootlin.com/linux/latest/source/net/core/skbuff.c#L2034
skb_put: https://elixir.bootlin.com/linux/latest/source/net/core/skbuff.c#L2013
skb_reserve: https://elixir.bootlin.com/linux/latest/source/include/linux/skbuff.h#L2633
skb_trim: https://elixir.bootlin.com/linux/latest/source/net/core/skbuff.c#L2093

sk_buffer总结

- sk_buff
  - 用于 Linux 网络子系统中各层之间的数据传递
  - 不同协议层的处理函数通过控制sk_buff结构来共享网络报文

- 收包:
  - 网卡收到数据包后，将以太网帧数据转换为sk_buff数据结构
  - 各层剥去相应的协议头部，直至交给用户

- 发包:
  - 网络模块必须建立一个包含待传输的数据包的sk_buff
  - 各层在sk_buff 中添加对应的协议头部直至交给网卡发送

大纲

- 越俎代庖部分
  - 网络协议的分层模型
  - 套接字模型
- 网络驱动模型
- Linux系统收包过程
  - 函数视角
  - 数据视角
- Linux系统发包过程
- 网络处理性能优化
  - 挑战
  - 数据面控制面分离
- Intel DPDK 软件优化方案
  - 总体框架
  - 无锁环
  - 内存池
  - 其它模块
  - 扩展框架
- 硬件优化方案

谢谢

微信：suyuxin
钉钉：苏玉鑫
B站：https://space.bilibili.com/502854403
软工集市课程专区：https://ssemarket.cn/new/course
匿名提问箱：https://suask.me/ask-teacher/106/苏玉鑫