ISCAS
中国科学院软件研究所
Institute of Software,Chinese Academy of Sciences

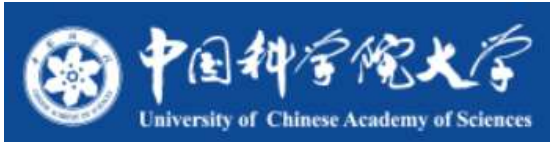
中国科学院大学
University of Chinese Academy of Sciences

网络管理

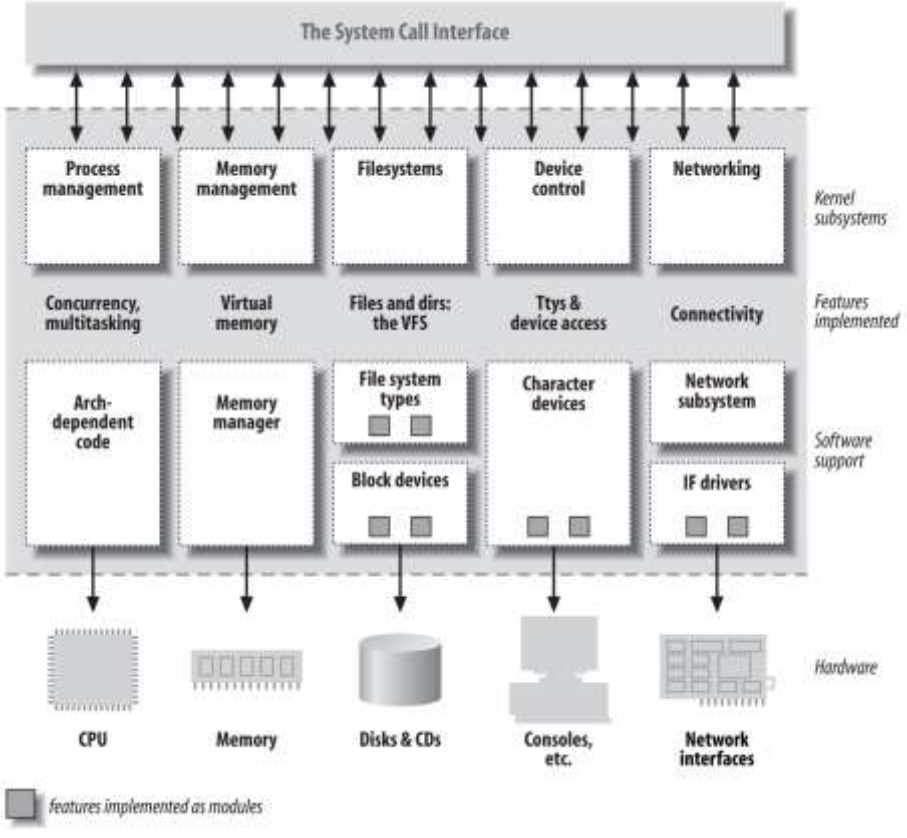
李鹏

改编声明

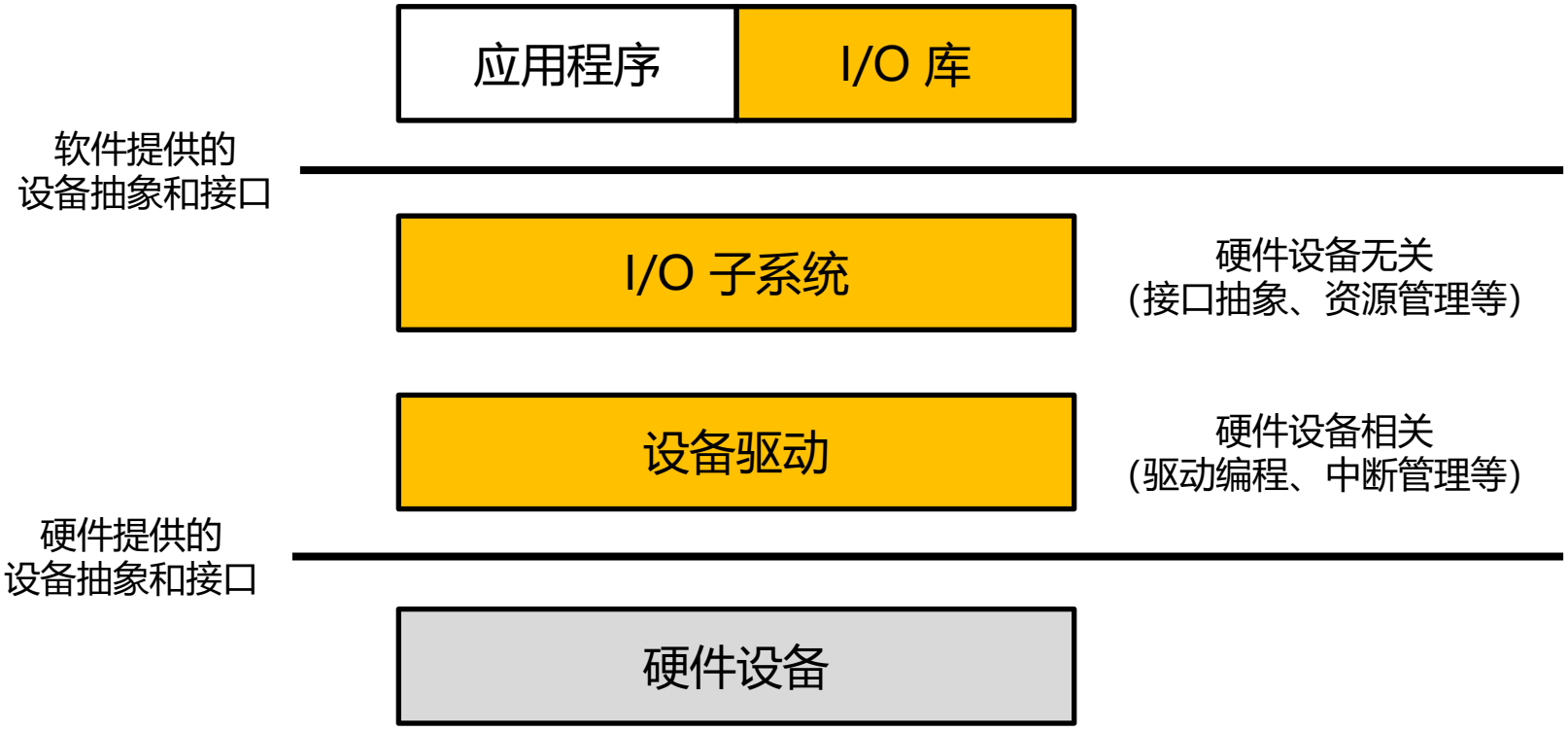
- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，已获得原作者授权，原课程官网：
  - https://ipads.se.sjtu.edu.cn/courses/os/index.shtml
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。



操作系统内核剖分图

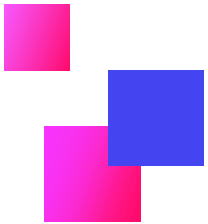

The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependent code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
Hardware
features implemented as modules

操作系统的I/O层次


应用程序
I/O 库
软件提供的
设备抽象和接口
I/O 子系统
硬件设备无关
(接口抽象、资源管理等)
设备驱动
硬件设备相关
(驱动编程、中断管理等)
硬件提供的
设备抽象和接口
硬件设备


中国互联网诞生地
BIRTHPLACE OF CHINA INTERNET
1994年4月20日，由中国科学院主持，联合北京大学、清华大学共同实施的世界银行贷款项目"中关村地区教育与科研示范网络（NCFC）"，通过美国Sprint公司连入Internet的64K国际专线开通，首次实现了中国与Internet的全功能连接。当时的中国连接点就设在旁边这幢红色办公楼的三层机房。
On April 20, 1994, the NCFC project (Education and Scientific Research Demonstration Network in Zhongguancun District, which was also named as National Computing and Networking Facility of China) opened a 64Kbps international dedicated line to the Internet through Sprint Co. Ltd of the United States, which achieved China's first full-functional connection to the Internet. The NCFC project was in the charge of Chinese Academy of Sciences, and was jointly implemented by Peking University and Tsinghua University, using the loan of the World Bank. The access point was just on the 3rd floor of the red building behind.
NCFC主干网及CASnet
NCFC's Backbone and CASnet
SHOT ON REDMI 10X
AI TRIPLE CAMERA

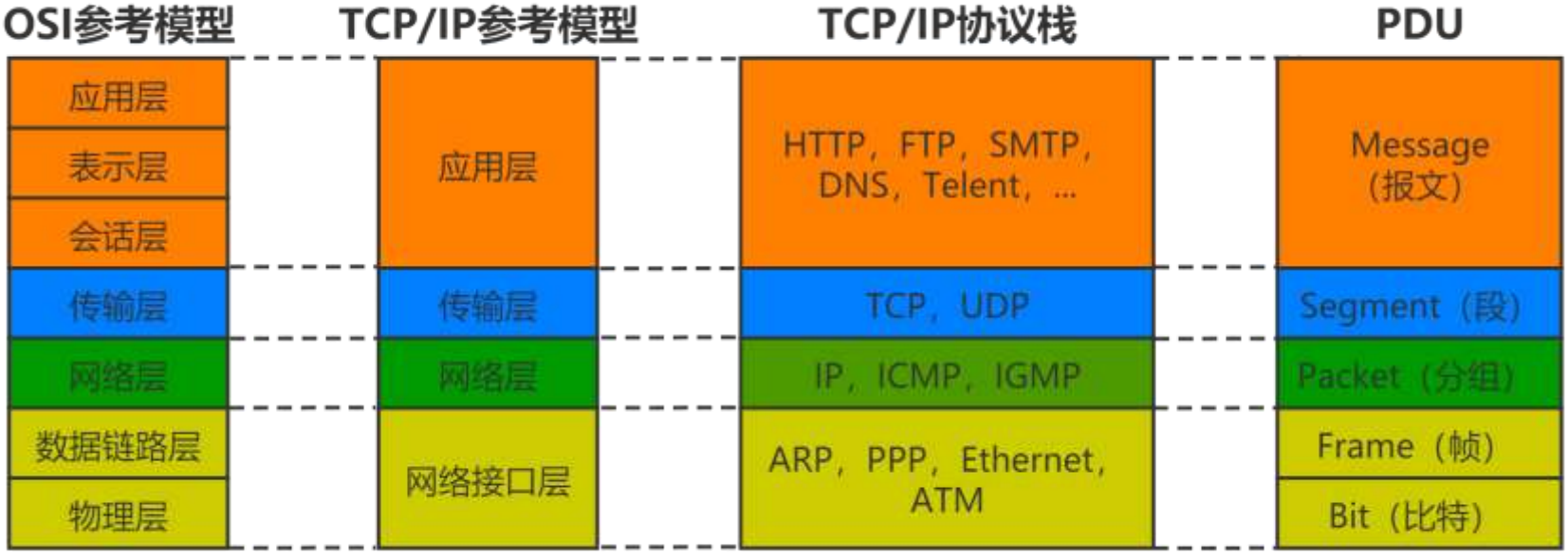
CONTENTS



PART / 1

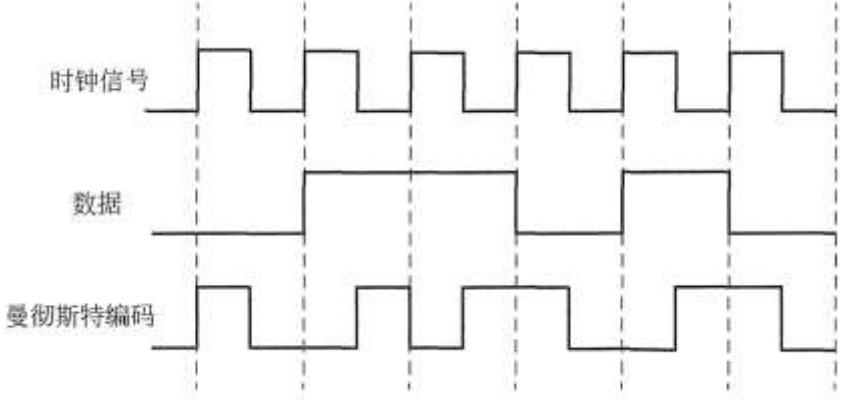
计算机网络简介

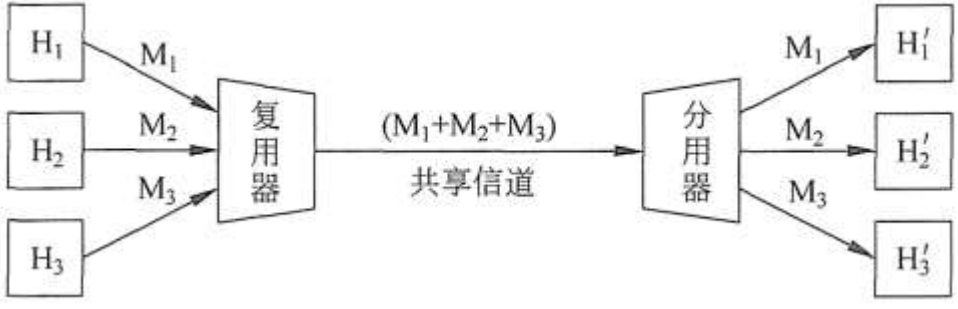
TCP/IP体系结构


OSI参考模型
TCP/IP参考模型
TCP/IP协议栈
PDU
应用层
表示层
会话层
传输层
网络层
数据链路层
物理层
应用层
传输层
网络层
网络接口层
HTTP，FTP，SMTP，
DNS，Telent，...
TCP，UDP
IP，ICMP，IGMP
ARP，PPP，Ethernet，
ATM
Message
（报文）
Segment（段）
Packet（分组）
Frame（帧）
Bit（比特）

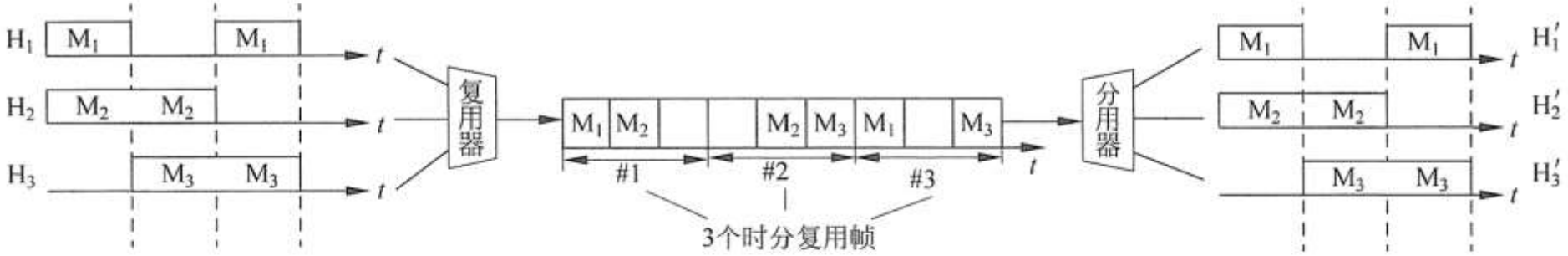
物理层

- **主要功能**
  - 两个直连机器间如何利用信道传递信号
- **信道及信号传递**
  - 导引型媒介：同轴电缆或光纤
  - 非导引型媒介
- **信号的调制与解调**
- **信道的复用**

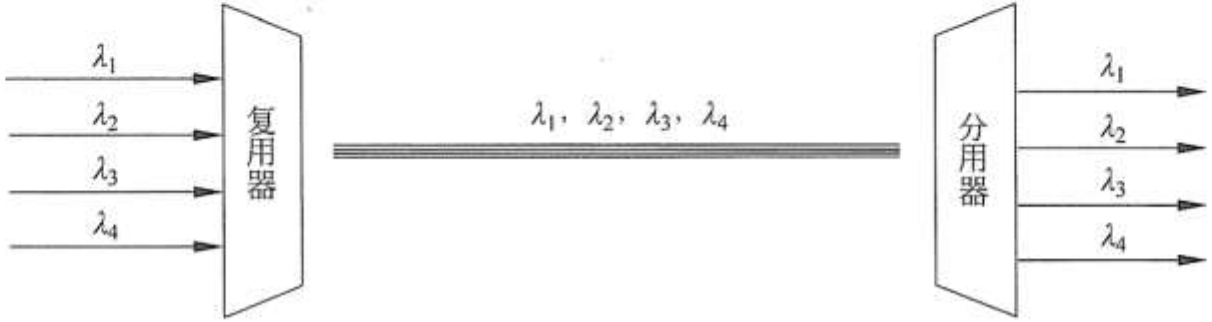




时分复用与波分复用



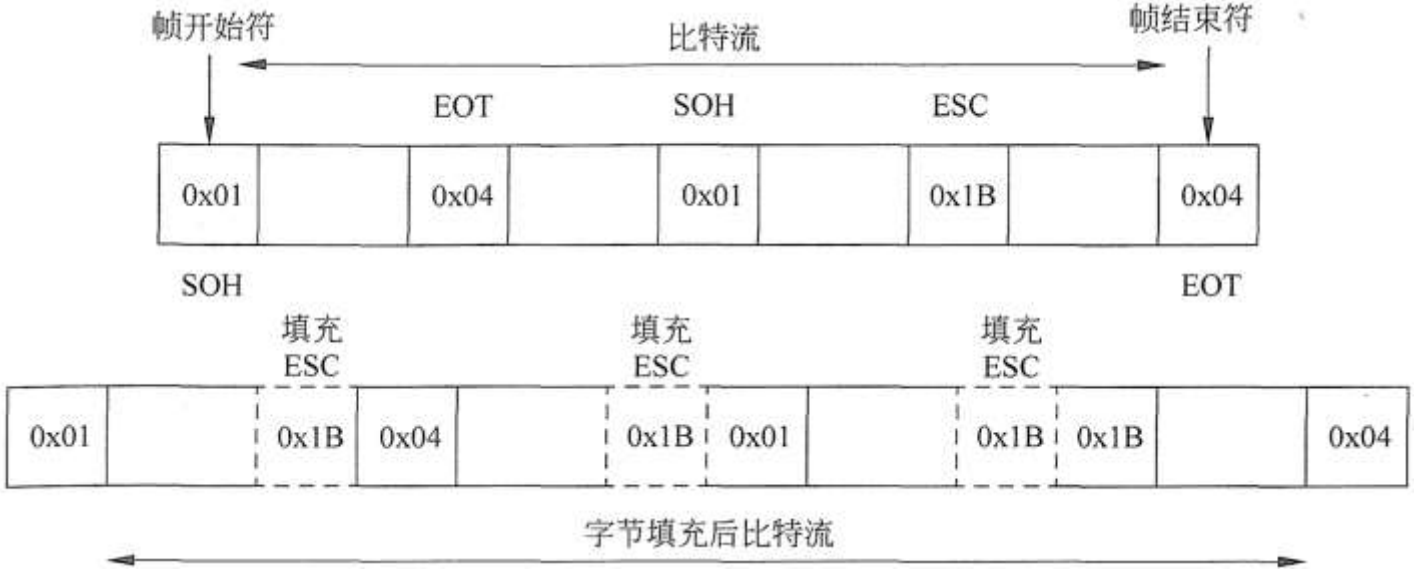
(a) 时分复用



(b) 波分复用

数据链路层

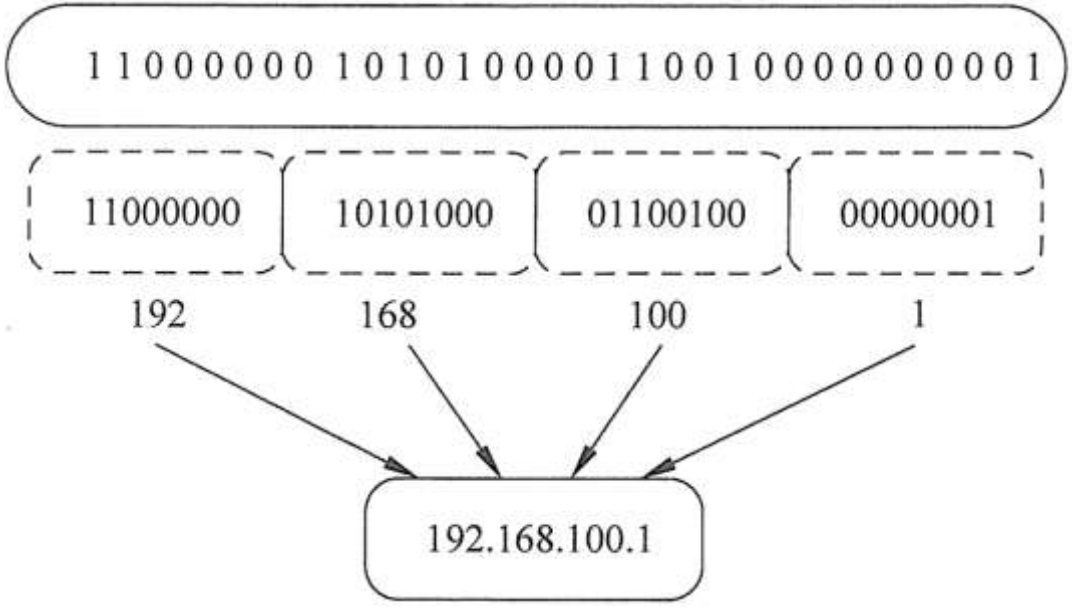
- **主要功能**
  - 局域网内将数据从一个计算机网卡发送到另一个计算机网卡
- **如何在局域网寻址**
  - MAC地址
  - 48位
- **封装成帧**
  - SOH 0x01
  - EOT 0x04
- **透明传输**
- **差错控制**
  - 奇偶校验
  - CRC



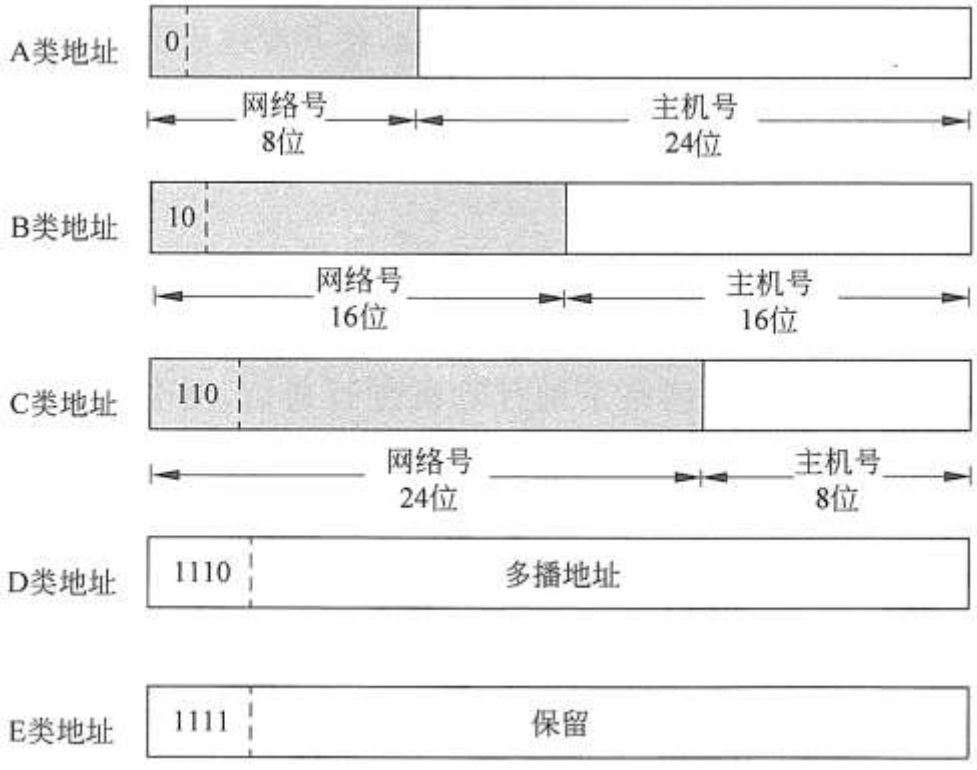
以太网的最大数据帧是1518Bytes

网络层

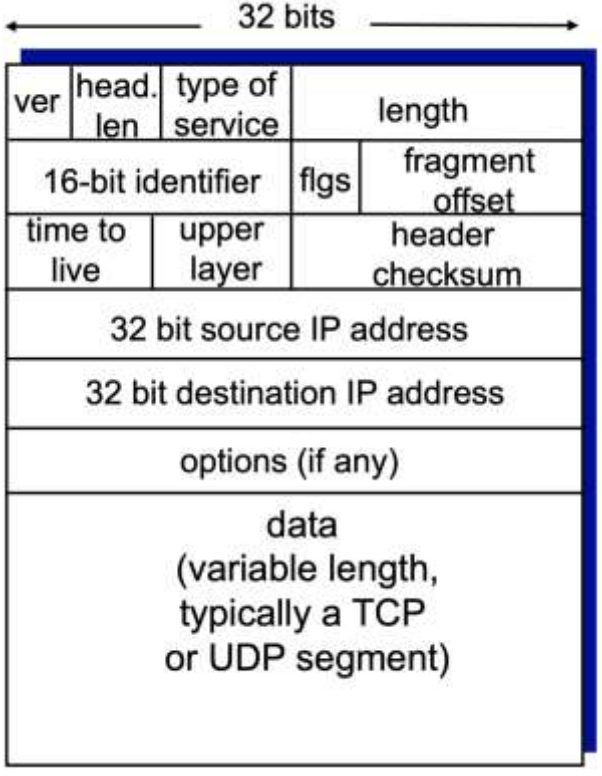
- **主要功能**
  - 跨网络通讯
- **如何在互联网中寻址**
  - MAC地址太复杂
  - 有兼容性问题
  - IP地址
    - IPv4
    - IPv6



IP地址分类


A类地址
0
网络号
8位
主机号
24位
B类地址
10
网络号
16位
主机号
16位
C类地址
110
网络号
24位
主机号
8位
D类地址
1110
多播地址
E类地址
1111
保留

IP数据包格式



- 版本号：规定数据报的IP协议版本（IPv4或IPv6）
- 首部长度：确定实际数据部分从哪开始
- 总长度：IP数据报的总长度（首部+数据）
- 标识、标志、片偏移：与IP分片有关
- 生存时间：确保数据报不会在路由器中循环，没经过一个路由器TTL减一
- 协议：到达终点后确定使用哪个运输层协议。6标识TCP，17标识UDP
- 首部检验和：用于帮助路由器检测收到的IP数据报中的比特错误

ARP协议

- **根据IP地址找到MAC地址**
  - 在局域网广播带有接收方 IP 地址的 ARP 请求报文
  - 局域网内所有主机都会接收该报文
    - 把自己的IP地址与报文里的目标 IP 地址进行比较
    - 如果两者相同，则向发送方返回带有自己 MAC 地址的 ARP 响应报文
    - 否则忽略此请求报文
  - 发送方接收到响应报文后，会缓存 IP 地址与 MAC 地址的对应关系
    - 后续可向该机器发送报文，而无须再广播 ARP 请求报文

传输层

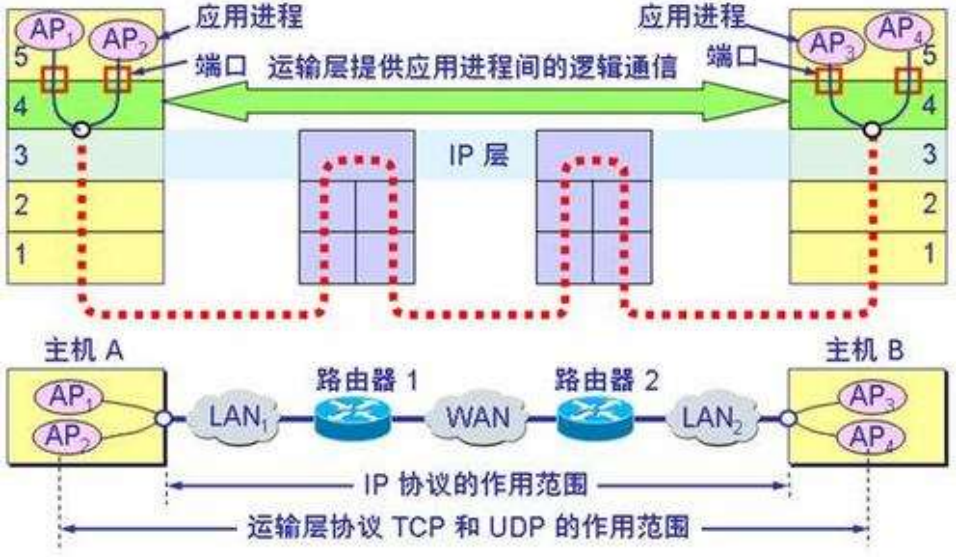
- **主要功能**
  - 基于网络层为进程间提供逻辑通讯链路
  - 如何标识进程
  - 如何基于不可靠的网络/数据链路/物理层，构建可靠交付的通信连接
  - 如何构造简单高效（不可靠）的通信链路

端口号

- **端口号**
  - 用来确定数据包要送往的具体进程
  - 16比特数字，0-65535
  - 周知端口：1-1023，分配给公认的协议和服务
    - FTP：21
    - telnet：23
    - DNS：53
  - 注册端口：1024-49151
    - 由IANA (Internet Assigned Numbers Authority)负责静态分配
  - 动态或私有端口：49151-65535
    - 客户端端口号

TCP与UDP

- **TCP**
  - 面向连接的运输
  - 可靠且有序传输
    - 建立点到点的连接
    - 拥塞控制、流量控制
- **UDP**
  - 无连接的运输
  - 不可靠且无序传输
    - 尽力而为的传输
    - 常用于语音、视频等实时数据传输



TCP数据报文

- **序列号：数据流的每一个字节都有序号，序列号的值是发送数据的第一个字节的序号**
- **确认序号：是发送确认的一端所期望收到的下一个序号**
  - ACK标志位：确认比特ACK=1时，确认号字段才有效
- **SYN标志位：在连接建立时用来同步序号**
- **FIN标志位：用来释放一个连接**
- **窗口大小：本地接收窗口（接收缓冲区）的大小**

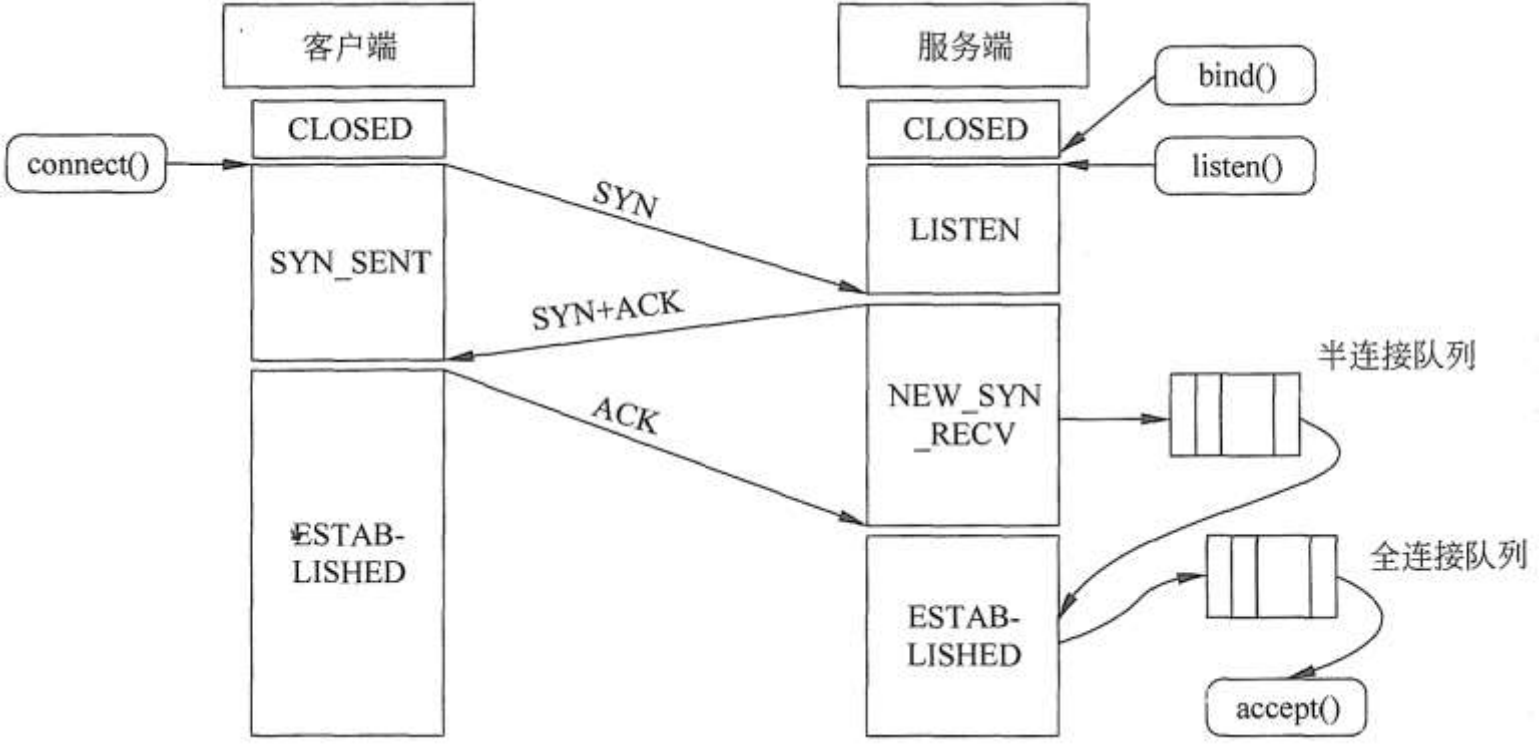
段的数据部分最大长度为1460B

问题：如何确定数据长度？

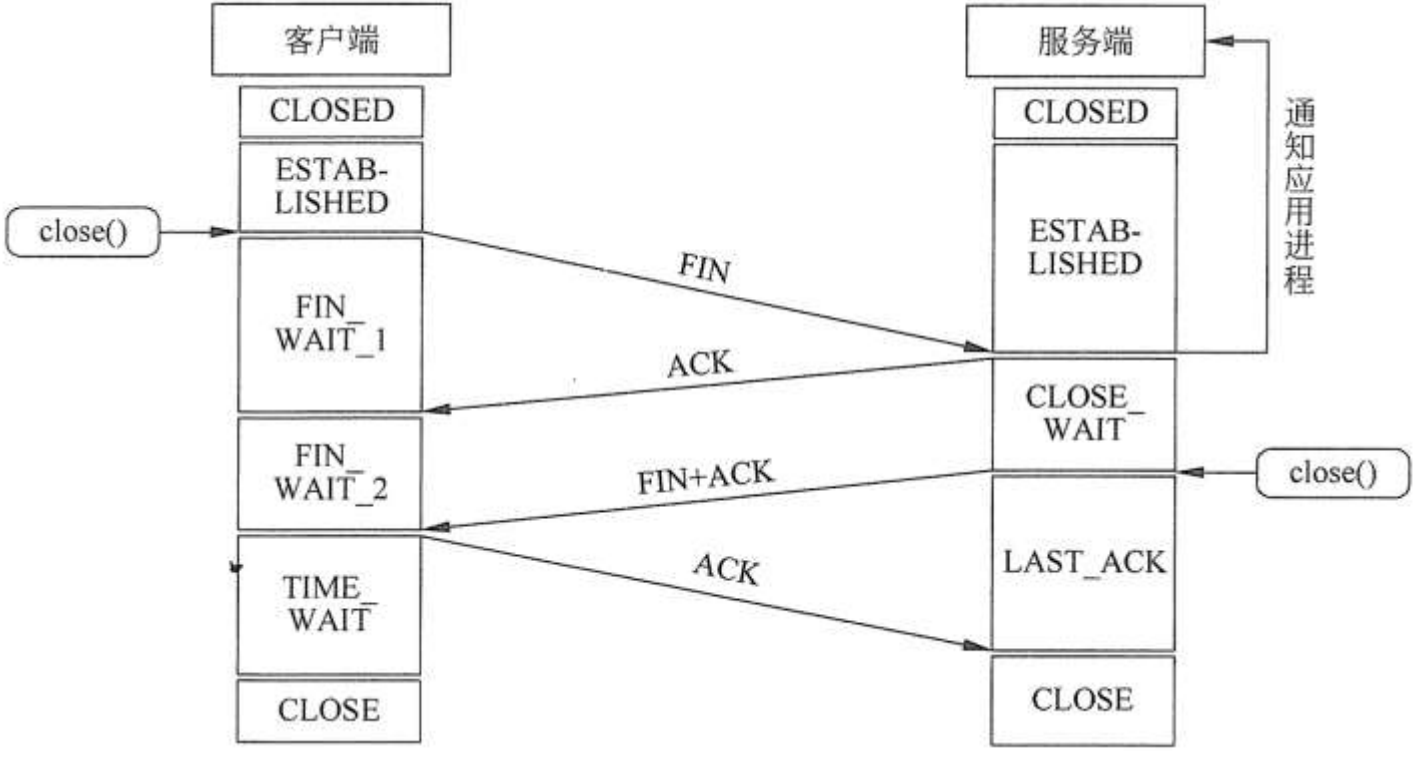
TCP可靠交付

- **确认应答**
  - 接收方接收到 TCP 数据段后给发送方回应确认应答 (Acknowledge) 信号
  - 发送方接收到应答信号后发送后续的段
- **超时重传**
  - 发送方为每个发送的段设置一个超时定时器
  - 如果在定时时间内没有收到应答信号，则认为数据段丢失
  - 此时需要重发该数据段并再次设置定时器，
  - 此过程可重复，直至确认发送成功
- **滑动窗口**
  - 发送方在无须等待应答信号的情况下，连续多个数据段

TCP建立连接

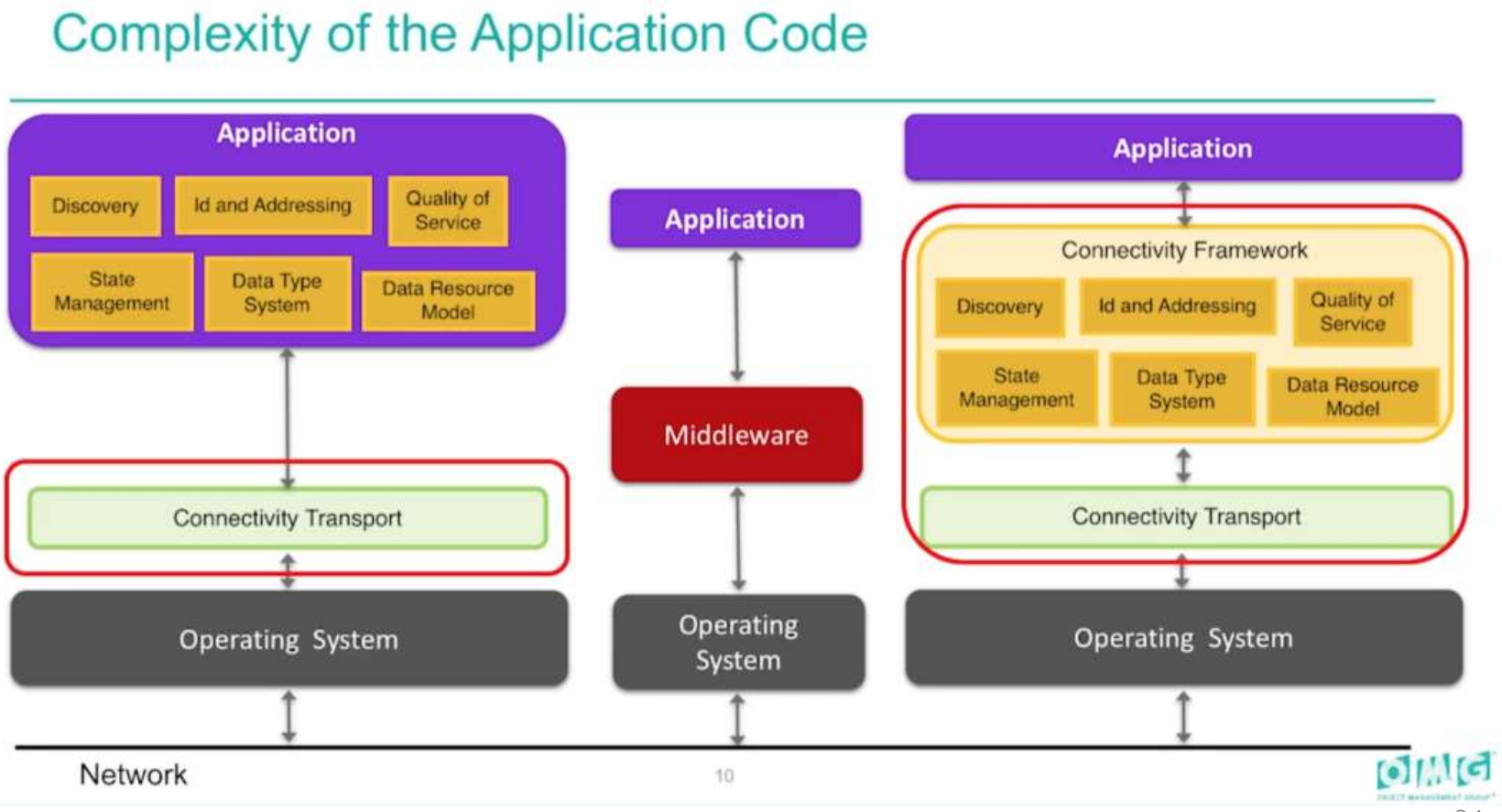

客户端
服务端
CLOSED
CLOSED
bind()
listen()
connect()
SYN
SYN_SENT
LISTEN
SYN+ACK
半连接队列
NEW_SYN
_RECV
ACK
ESTAB-
LISHED
全连接队列
ESTAB-
LISHED
accept()

TCP断开连接

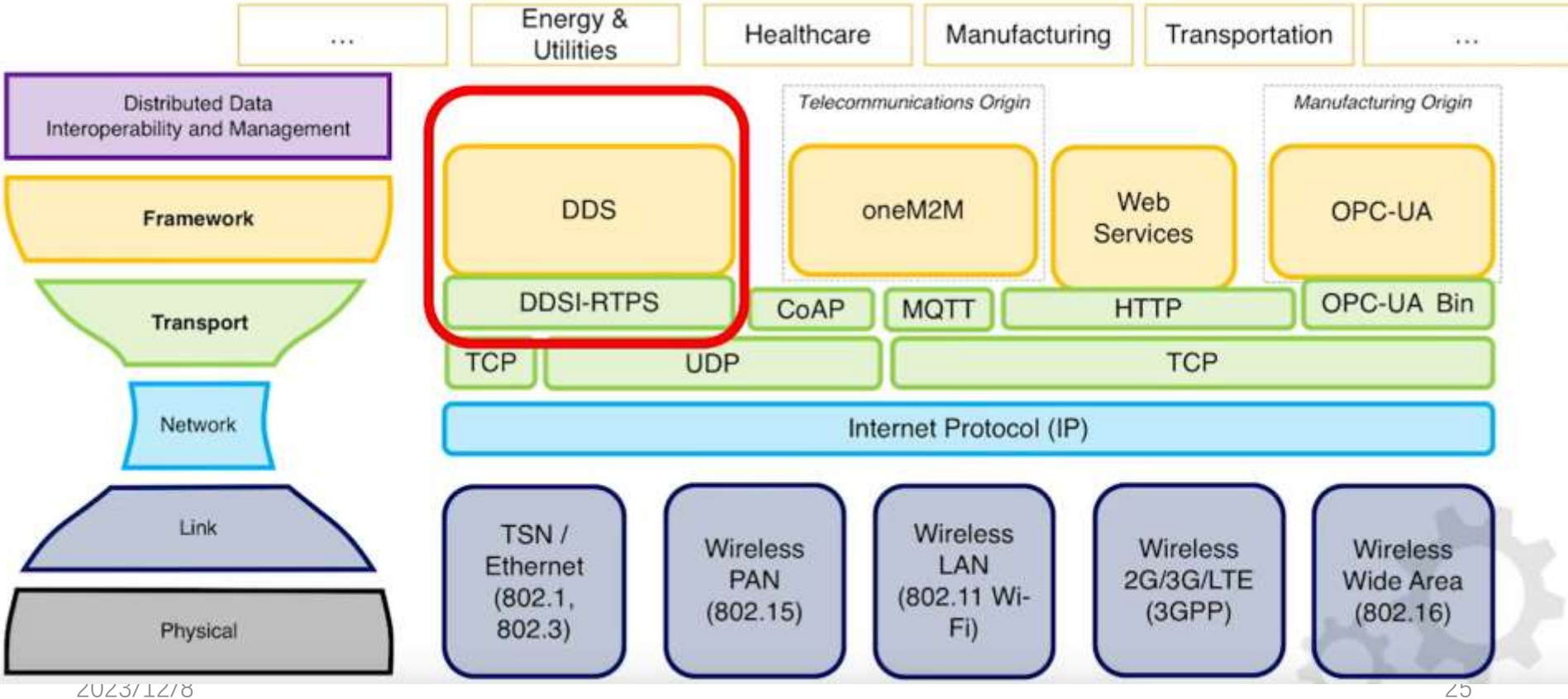

客户端
服务端
CLOSED
CLOSED
ESTAB-
LISHED
ESTAB-
LISHED
close()
FIN
FIN_
WAIT_1
ACK
CLOSE_
WAIT
通知应用进程
FIN_
WAIT_2
FIN+ACK
close()
TIME_
WAIT
ACK
LAST_ACK
CLOSE
CLOSE

UDP数据报文

会话层与表示层


Complexity of the Application Code
Application
Discovery
Id and Addressing
Quality of Service
State Management
Data Type System
Data Resource Model
Connectivity Transport
Operating System
Application
Middleware
Operating System
Application
Connectivity Framework
Discovery
Id and Addressing
Quality of Service
State Management
Data Type System
Data Resource Model
Connectivity Transport
Operating System
Network
10

会话层与表示层

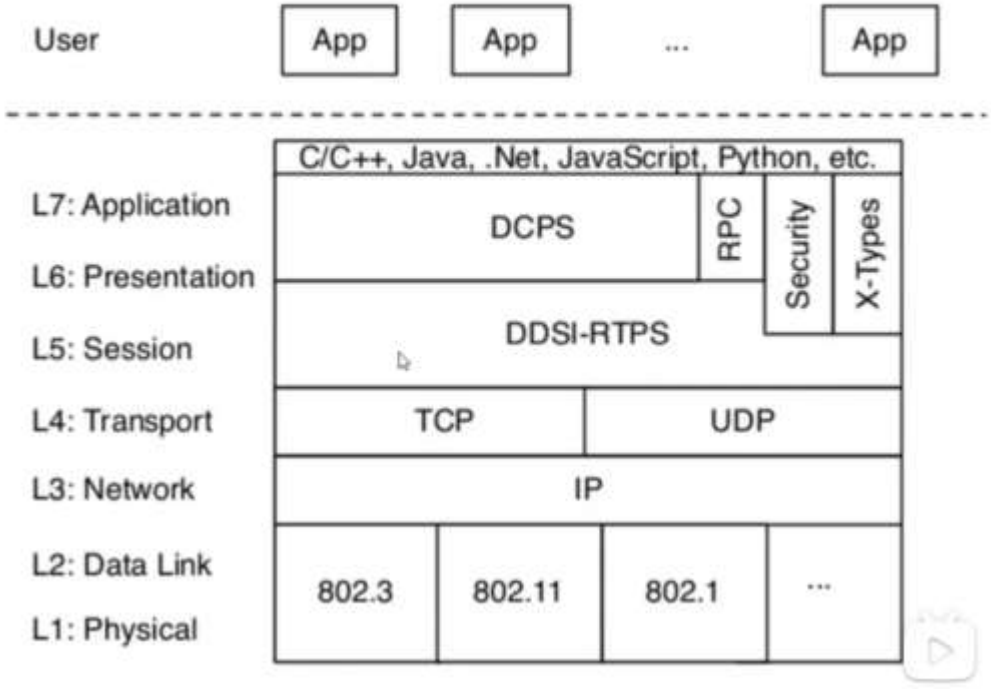

...
Energy & Utilities
Healthcare
Manufacturing
Transportation
...
Distributed Data Interoperability and Management
Framework
Transport
Network
Link
Physical
Telecommunications Origin
Manufacturing Origin
DDS
oneM2M
Web Services
OPC-UA
DDSI-RTPS
CoAP
MQTT
HTTP
OPC-UA Bin
TCP
UDP
TCP
Internet Protocol (IP)
TSN / Ethernet (802.1, 802.3)
Wireless PAN (802.15)
Wireless LAN (802.11 Wi-Fi)
Wireless 2G/3G/LTE (3GPP)
Wireless Wide Area (802.16)

Data Distribution Service

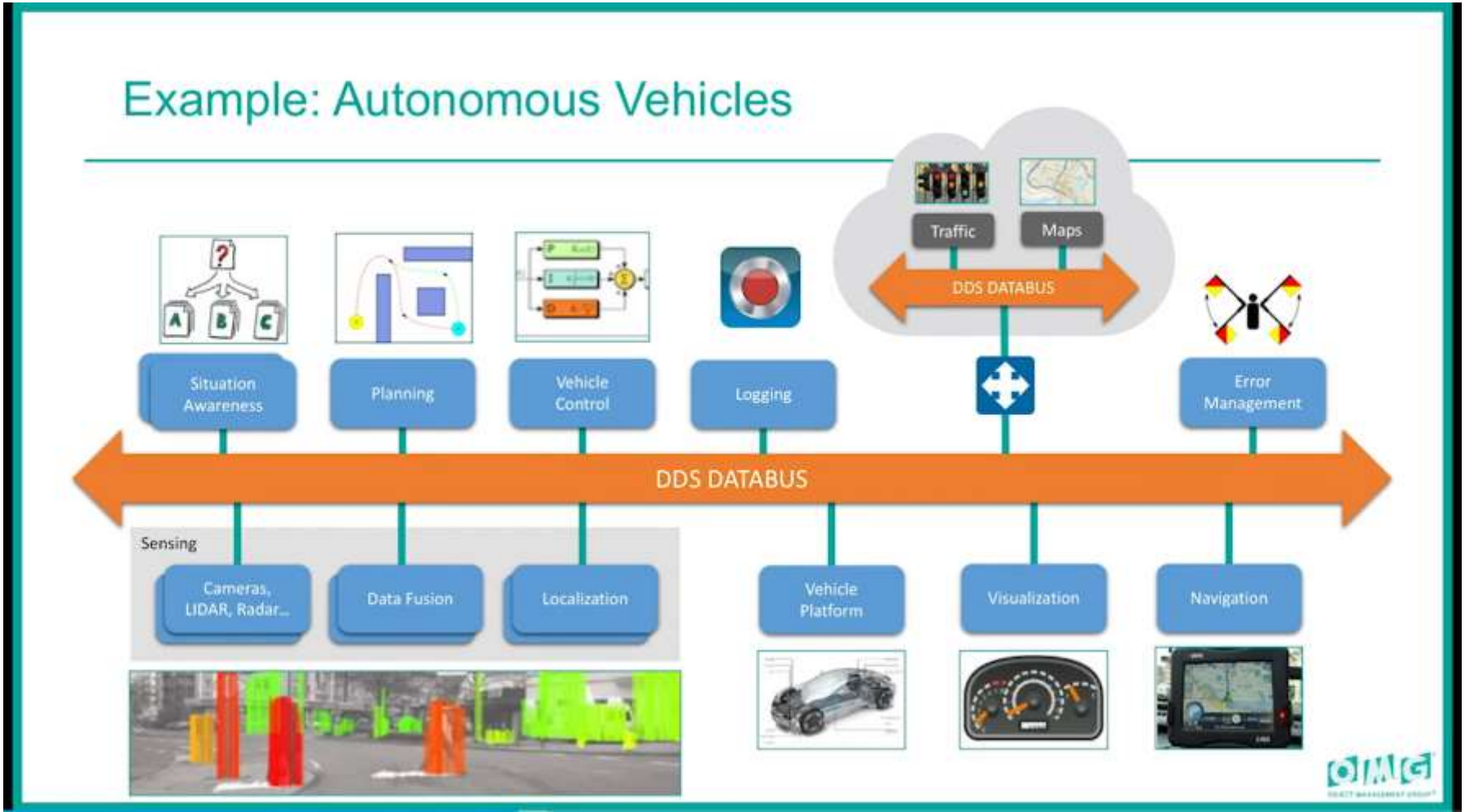
industrial internet CONSORTIUM

DDS
The IoT Data Sharing Standard

OMG
WE SET THE STANDARD



自动驾驶实例


Example: Autonomous Vehicles
Traffic
Maps
DDS DATABUS
Situation Awareness
Planning
Vehicle Control
Logging
Error Management
DDS DATABUS
Sensing
Cameras, LIDAR, Radar...
Data Fusion
Localization
Vehicle Platform
Visualization
Navigation

DDS Model: Virtual Global Data Space

DDS DOMAIN

Topic A

QoS

Topic B : "Turbine State"

QoS

Source (Key)	Speed	Power	Phase
WPT1	37.4	122.0	-12.20
WPT2	10.7	74.0	-12.23
WPTN	50.2	150.07	-11.98

Topic C

QoS

Topic D

QoS

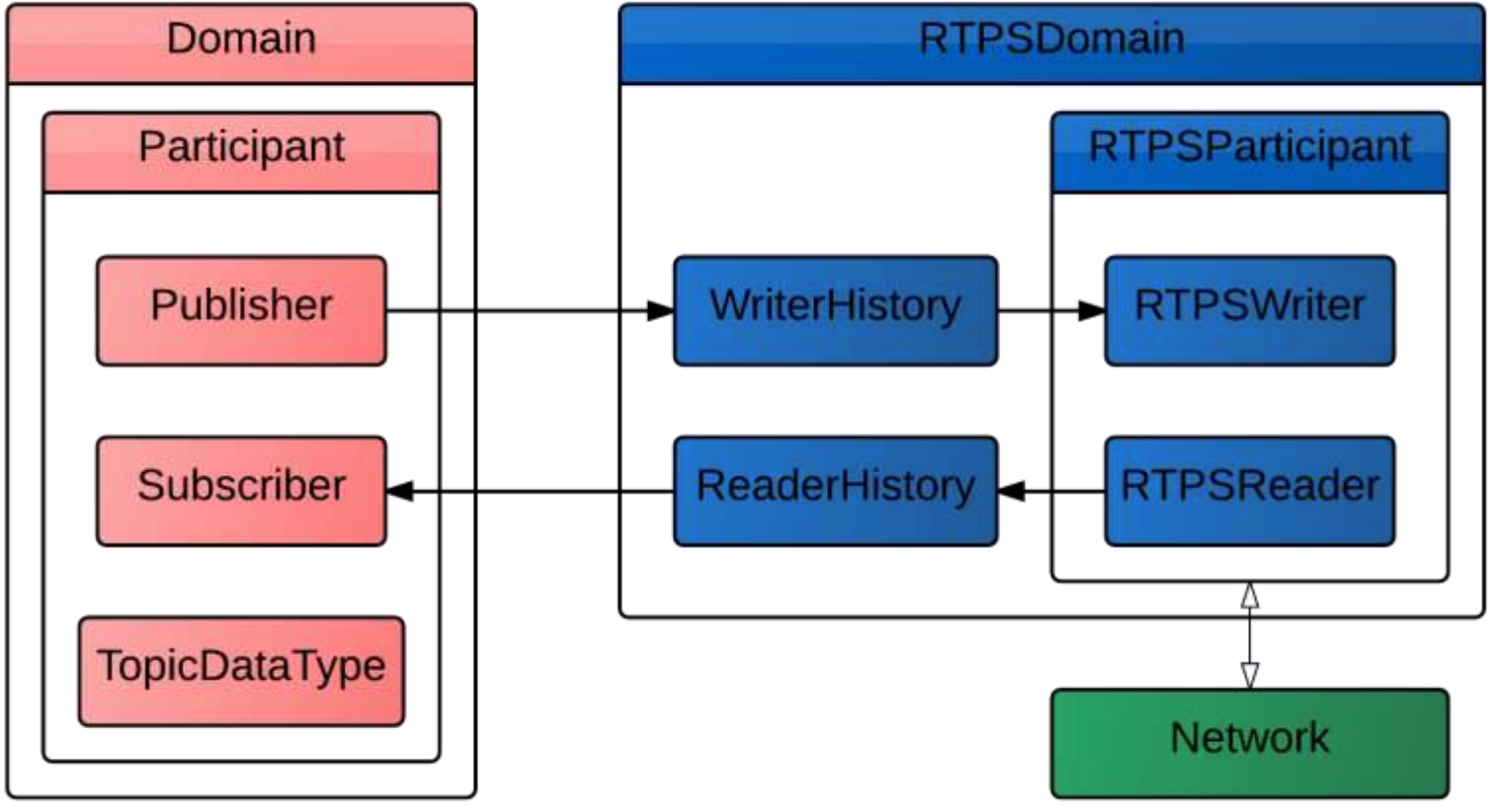
DDS

CRUD operations

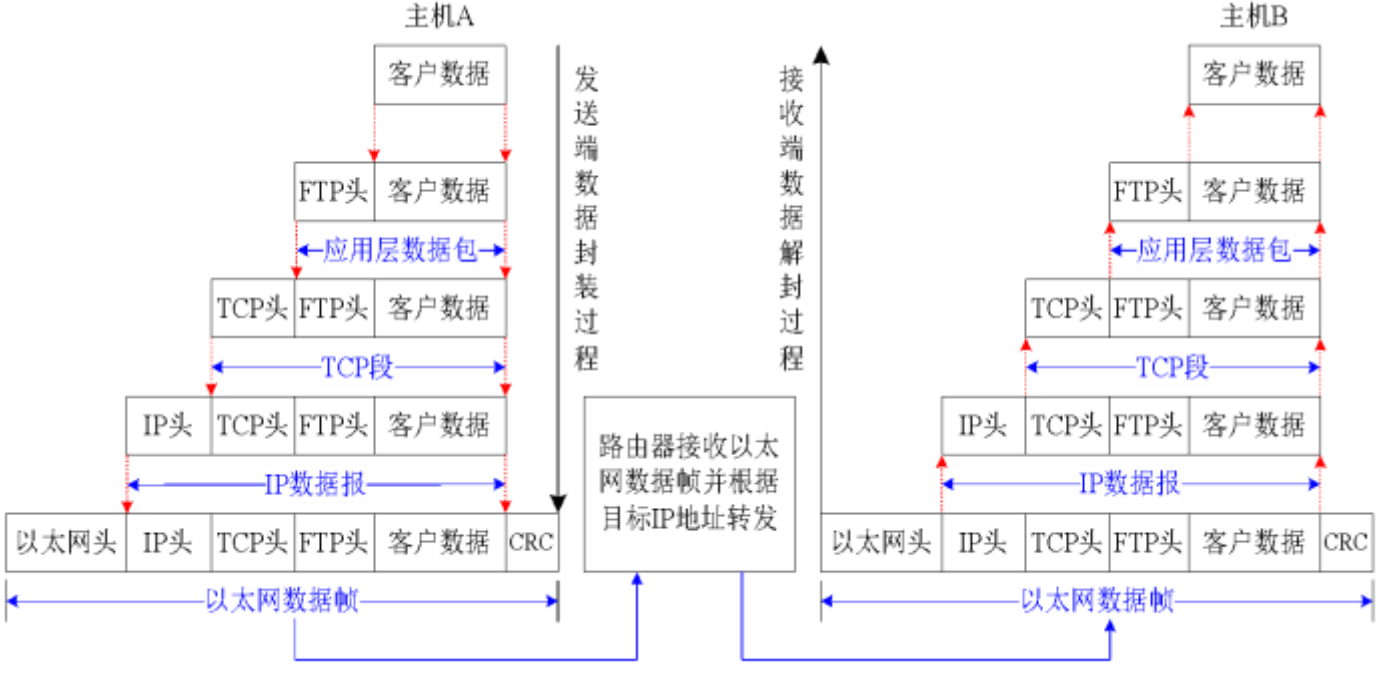
Persistence Service

Recording Service

会话层与表示层

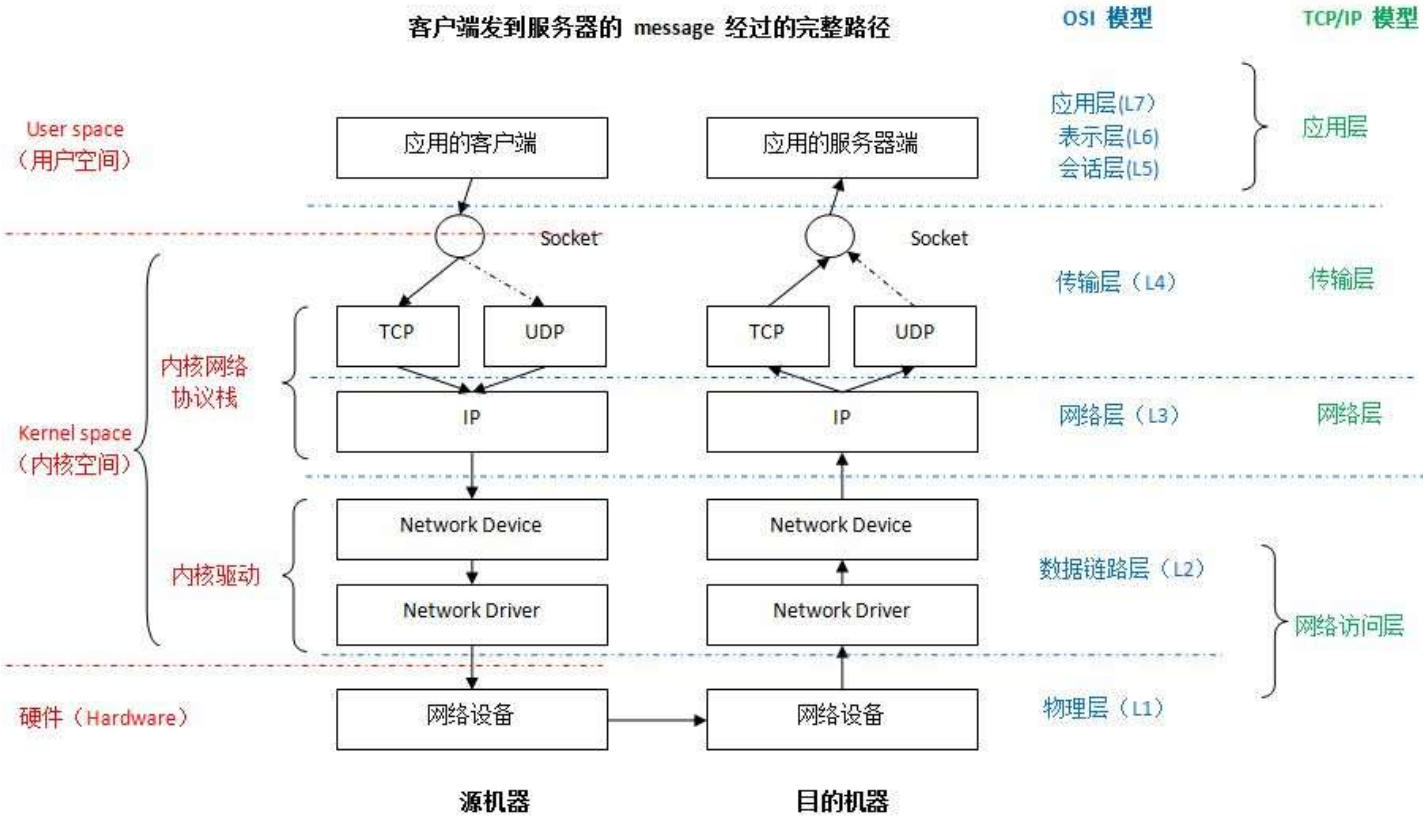

Domain
Participant
Publisher
Subscriber
TopicDataType
RTPSDomain
RTPSParticipant
WriterHistory
RTPSWriter
ReaderHistory
RTPSReader
Network

数据的封装与传递过程

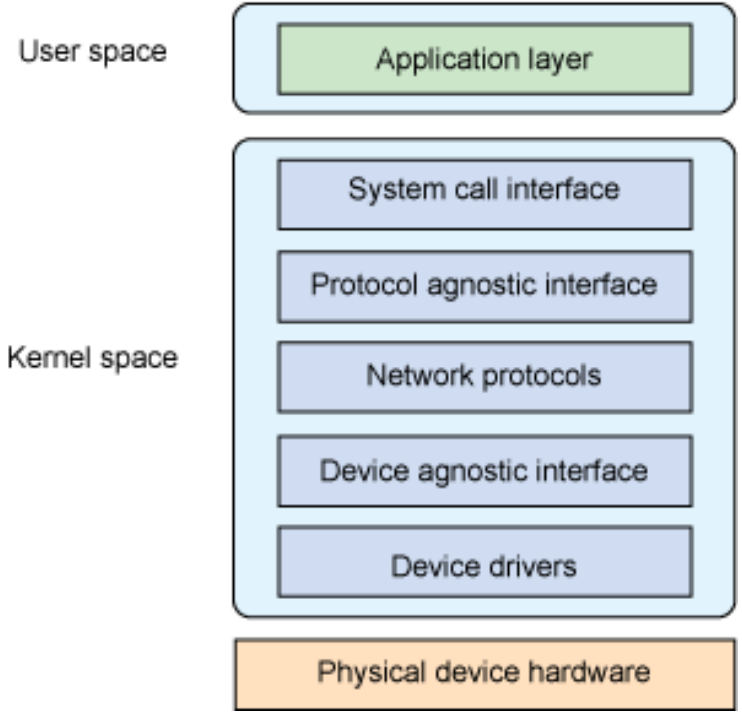

主机A
客户数据
FTP头
客户数据
应用层数据包
TCP头
FTP头
客户数据
TCP段
IP头
TCP头
FTP头
客户数据
IP数据报
以太网头
IP头
TCP头
FTP头
客户数据
CRC
以太网数据帧
发送端数据封装过程
路由器接收以太网数据帧并根据目标IP地址转发
接收端数据解封过程
主机B
客户数据
FTP头
客户数据
应用层数据包
TCP头
FTP头
客户数据
TCP段
IP头
TCP头
FTP头
客户数据
IP数据报
以太网头
IP头
TCP头
FTP头
客户数据
CRC
以太网数据帧

PART / 2

Linux网络简介


客户端发到服务器的 message 经过的完整路径
OSI 模型
TCP/IP 模型
User space
（用户空间）
应用的客户端
应用的服务器端
应用层(L7)
表示层(L6)
会话层(L5)
应用层
Socket
Socket
传输层（L4）
传输层
TCP
UDP
TCP
UDP
内核网络
协议栈
IP
IP
网络层（L3）
网络层
Kernel space
（内核空间）
Network Device
Network Device
内核驱动
Network Driver
Network Driver
数据链路层（L2）
网络访问层
硬件（Hardware）
网络设备
网络设备
物理层（L1）
源机器
目的机器

Linux网络栈架构



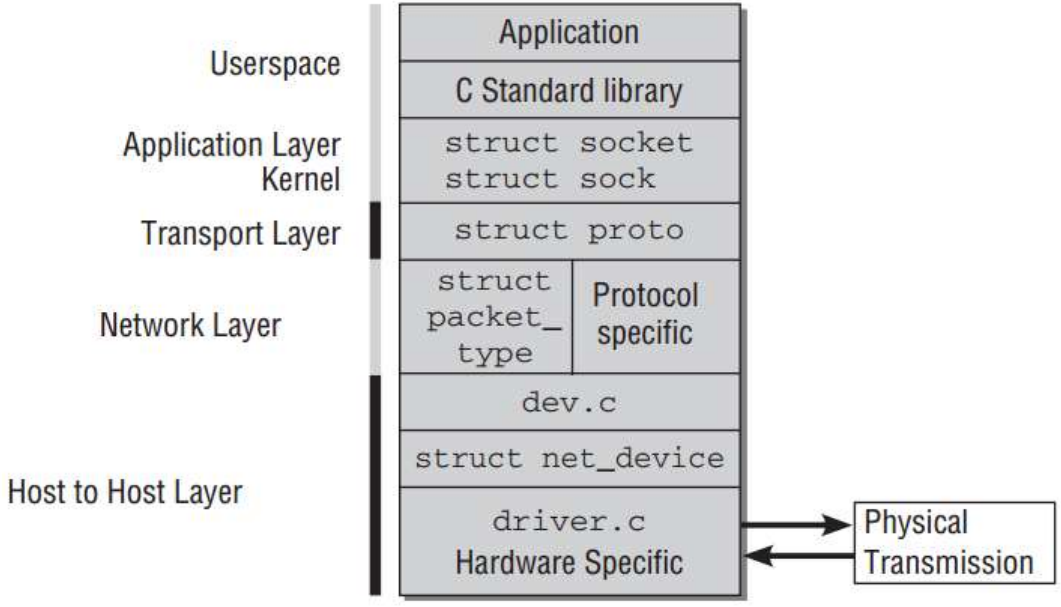
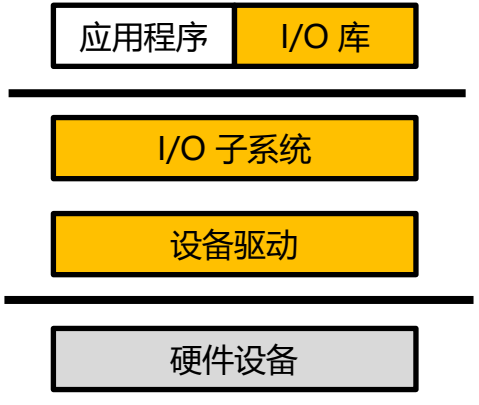
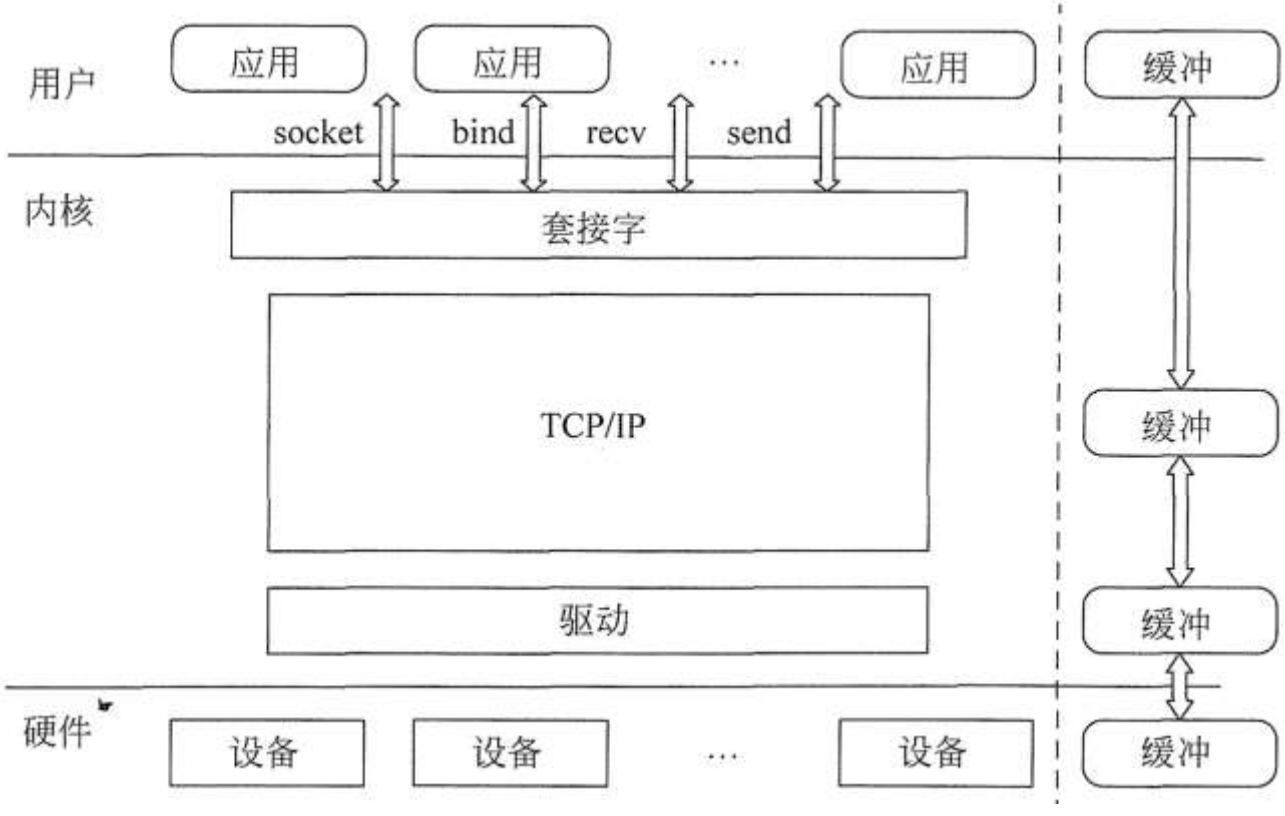


Figure 12-3: Implementation of the layer model in the kernel.

Linux网络协议栈逻辑架构

- 收包三阶段
- 发包三阶段

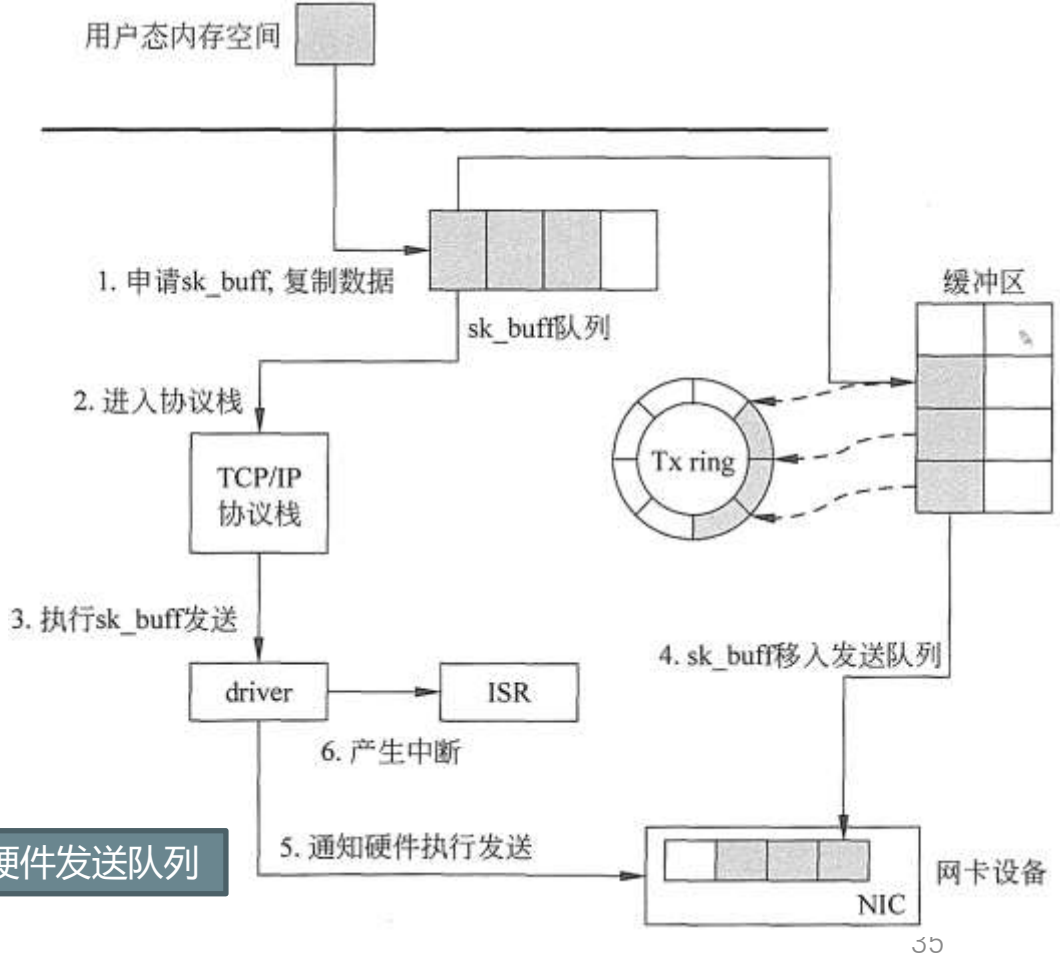




数据报文发送的整体流程

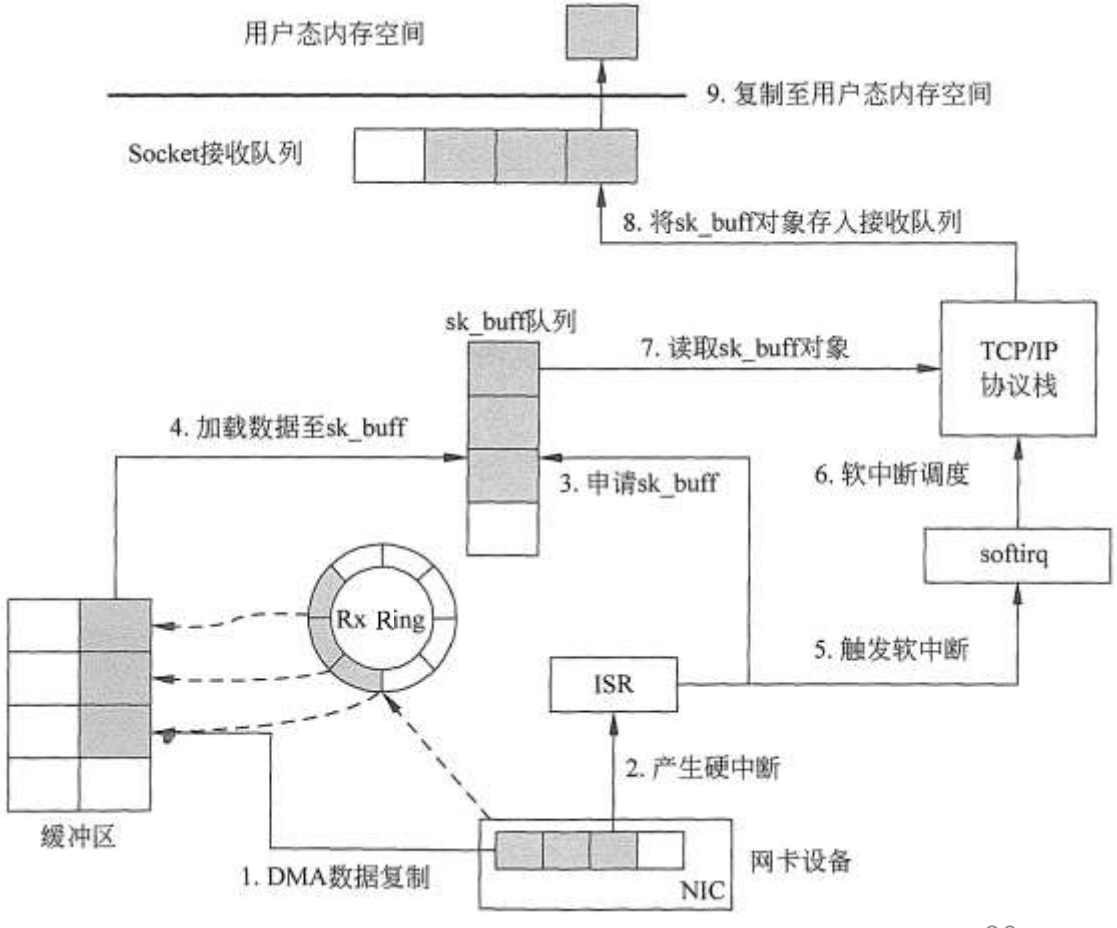
- **用户态内存空间到sk_buff**
  - 系统调用
  - 申请sk_buff对象
  - 数据包复制
  - 控制权移交
- **sk_buff到TxRing**
  - 封装各层协议的头部
  - 通知驱动程序接收sk_buff
  - 把sk_buff对象移入Tx ring所指的缓冲区
- **TxRing到网卡硬件发送队列**
  - 驱动程序发送数据
  - 如果成功，通过中断通知协议栈释放sk_buff
  - 如果失败，通知协议栈执行软中断

用户态内存空间->内核态内存空间->硬件发送队列



数据报文接收的整体流程

- 网卡硬件接收队列到Rx ring
- Rx ring到sk_buff
- sk_buff到用户态内存空间

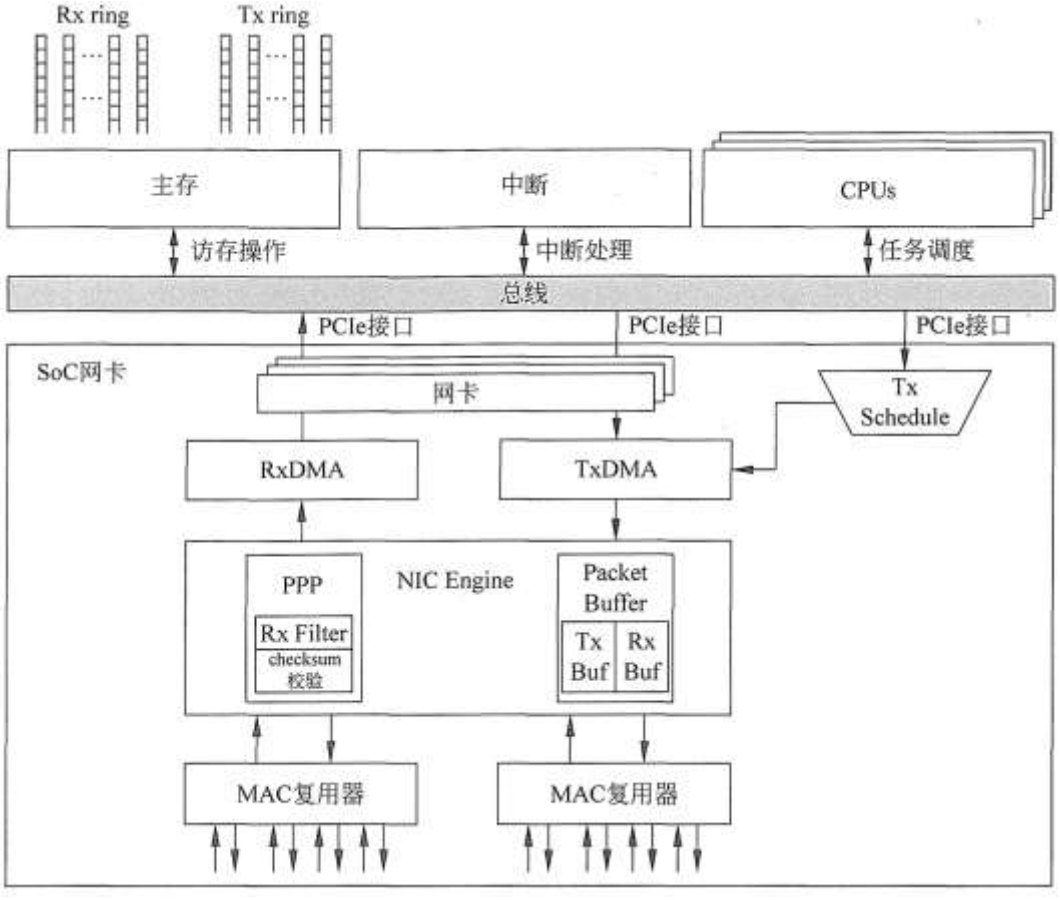


PART / 3

网卡驱动程序

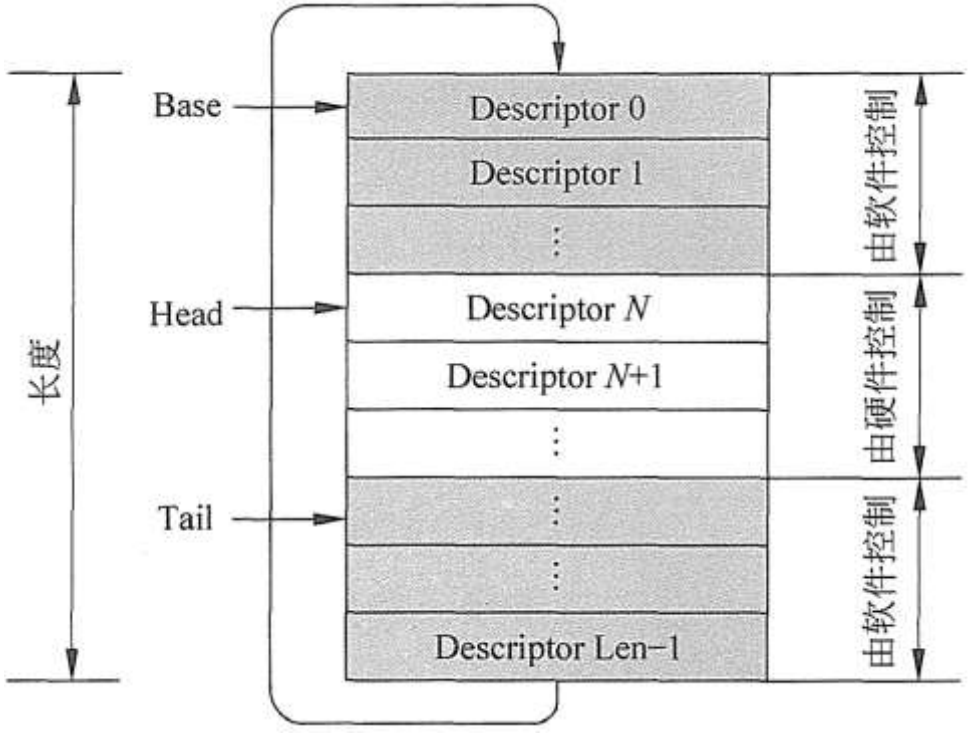
鲲鹏920SoC网卡结构框图

- **Packet Buffer**
- **DMA**
- **MAC复用器**
  - 将单个物理网卡分解成多个虚拟网卡
  - 便于多个CPU共享物理端口带宽
- **Programmable Packet Process**
  - 配置Rx Filter，设置白名单
- **Flow Director**
  - 将数据包分发给不同的CPU



Rx ring buffer

- **存储于内存中**
- **三个指针**
  - Base: Rx ring的首地址
  - Head：空闲描述符的头部
    - 由网卡管理
  - Tail：空闲描述符的尾部
    - 由驱动程序管理



Descriptor

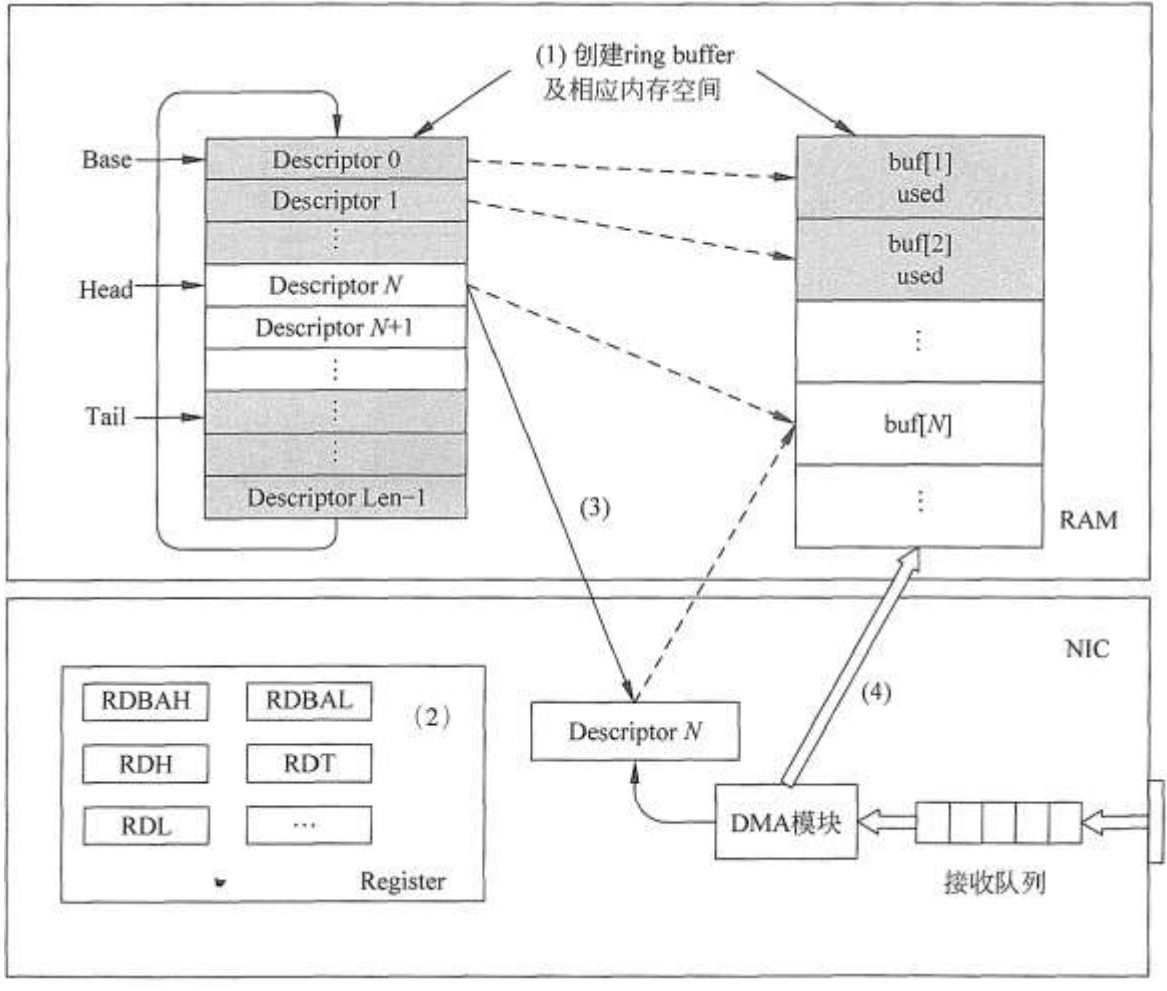
- 描述符的具体格式与硬件网卡相关
- 128 bit

```
497 struct e1000_rx_desc {
498         __le64 buffer_addr;     /* Address of the descriptor's data buffer */
499         __le16 length;          /* Length of data DMAed into data buffer */
500         __le16 csum;            /* Packet checksum */
501         u8 status;              /* Descriptor status */
502         u8 errors;              /* Descriptor Errors */
503         __le16 special;
504 };
```

/drivers/net/ethernet/intel/e1000/e1000_hw.h

NIC与Rx ring的交互

- **Rx ring buffer**
  - Base
  - Head
  - Tail
  - Length
- **驱动程序在初始化时创建Rx ring及其描述符对应的数据包缓存**
- **驱动程序将相关信息写入网卡寄存器**
- **网卡根据RDBA和RDH获取空闲描述符**
- **把接收到的数据包写入指定地址，Head++**

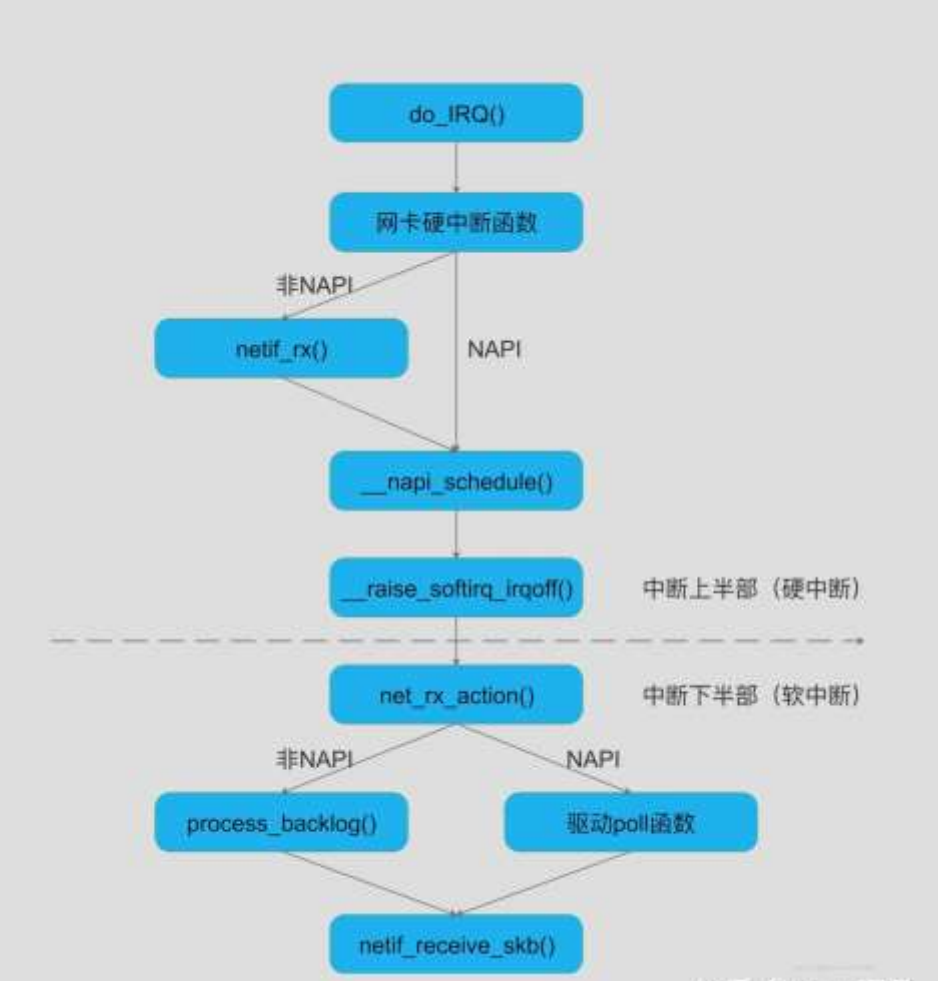


第二次数据复制：从Rx ring到协议栈（sk_buff）

- **CPU收到网卡中断信号，进入中断服务程序，从Rx ring中取数据**
- **千兆以太网**
  - 1G位/秒/（1518字节/数据包*8位/字节）=82345数据包/秒
- **轮询vs中断**
- **NAPI**
  - 在接收数据时关闭网卡中断事件
  - 开始轮询处理已接收到的数据包
  - 为了避免单次轮询处理时间过长
    - 为每次轮询设置处理数据包配额(quota) 及超时时间

网络收包中断处理简介

- 上半部
- 下半部

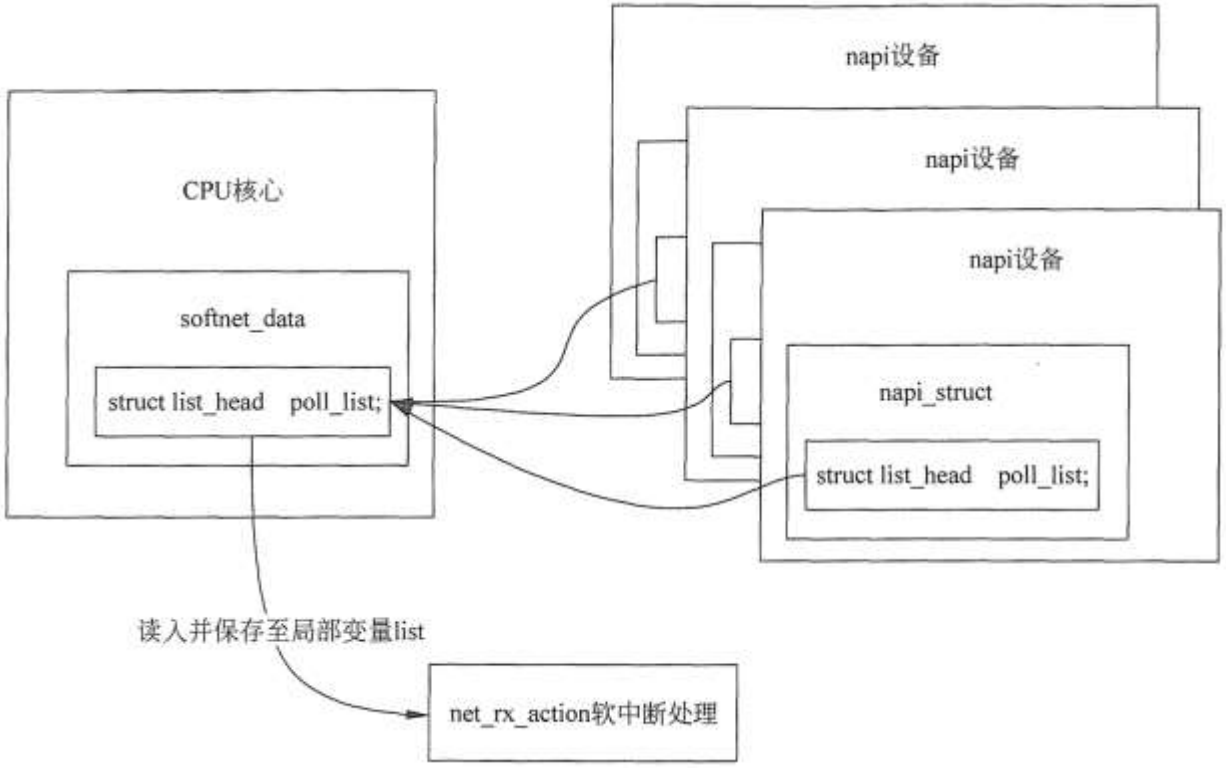


中断处理上半部

```
1.    //源文件: drivers/net/ethernet/intel/e1000/e1000_main.c
2.    static irqreturn_t e1000_intr(int irq, void * data) {
3.        ew32(IMC, ~0);          //关网卡中断
4.        //将设备的 NAPI 对象添加到软中断轮询列表并触发软中断
5.        __napi_schedule(&adapter->napi);
6.        ...
7.    }
```

```
1.    //源文件: net/core/dev.c
2.    static inline void __napi_schedule(struct softnet_data * sd,
3.                                       struct napi_struct * napi) {
4.        //将当前设备的 napi 添加到 softdata
5.        list_add_tail(&napi->poll_list, &sd->poll_list);
6.        __raise_softirq_irqoff(NET_RX_SOFTIRQ);        //触发软中断
7.    }
```

当前CPU能处理的NAPI设备：softnet_data



注册软中断

```
1.    //源文件：net/core/dev.c
2.    static int __init net_dev_init(void) {
3.        open_softirq(NET_RX_SOFTIRQ, net_rx_action);     //注册软中断函数
4.        ...
5.    }
```

软中断处理函数——中断处理下半部

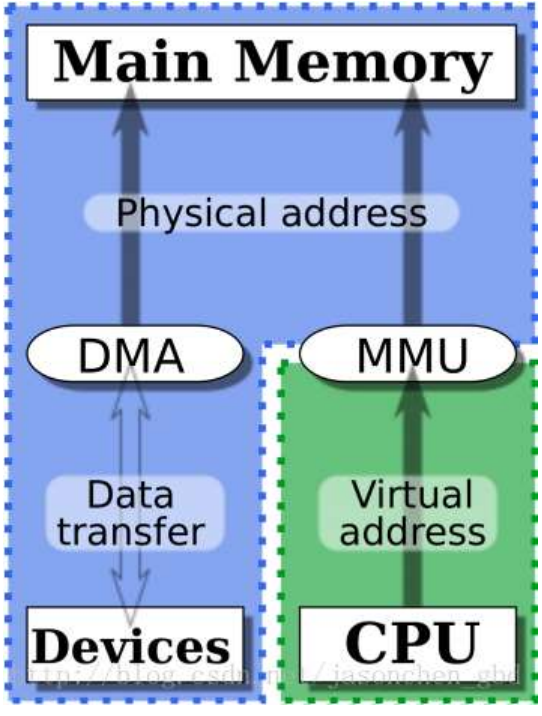
```
1.    //源文件：net/core/dev.c
2.    int netdev_budget = 300;        //每个设备每次最多只能处理 300 个数据包
3.    static __latent_entropy void net_rx_action(struct softirq_action * h) {
4.        int budget = netdev_budget;
5.        //循环获取 napi 结构，并调用其对应的 poll 函数
6.        for (;;) {
7.            //获取设备的 napi 结构
8.            n = list_first_entry(&list, struct napi_struct, poll_list);
9.            //每次调用轮询函数都会将 budget 减去所处理的数据包个数
10.           budget -= napi_poll(n, &repoll);
11.           //如果当前设备处理的数据包超过 300 个或者超时，就退出当前循环
12.           if (unlikely(budget <= 0 || time_after_eq(jiffies,
13.                                                     time_limit))) {
14.               break;
15.           }
16.       }
17.       ...
18.   }
```

轮询函数napi-poll的实现

- 调用栈
  - net_rx_action
  - e1000_clean
  - e1000_clean_rx_irq
  - e1000_receive_skb
  - netif_receive_skb

```
1.  //源文件：drivers/net/ethernet/intel/e1000/e1000_main.c
2.  static bool e1000_clean_rx_irq(struct e1000_adapter *adapter,
3.                                  struct e1000_rx_ring *rx_ring,
4.                                  int *work_done, int work_to_do) {
5.      //获取当前要处理的描述符指针，即 Tail 指针
6.      unsigned int i = rx_ring->next_to_clean;
7.      //根据 Tail 指针获取对应数据包缓冲区信息
8.      struct e1000_rx_buffer *buffer_info = &rx_ring->buffer_info[i];
9.      //根据描述符状态循环处理所有就绪描述符(DD: Descriptor Done)
10.     while (rx_desc->status & E1000_RXD_STAT_DD) {
11.         //将 data 指向的内存空间里的数据包复制到新建的 skb
12.         struct sk_buff *skb = e1000_copybreak(adapter, buffer_info,
13.                                               length, data);
14.         //将 sk_buff 上报给协议栈
15.         e1000_receive_skb(adapter, status, rx_desc->special, skb);
16.         //解除该数据包缓冲区的 DMA 映射
17.         dma_unmap_single(&pdev->dev, buffer_info->dma,
18.                          adapter->rx_buffer_len,
19.                          DMA_FROM_DEVICE);
20.         //清除缓冲区信息
21.         buffer_info->dma = 0;
22.         buffer_info->rxbuf.data = NULL;
23.         //处理下一个描述符
24.         if (++i == rx_ring->count)
25.             i = 0;
26.     }
27.     //设置下一次轮询时的起点，即设置指针 Tail
28.     rx_ring->next_to_clean = i;
29. }
```

dma_unmap_single




```c
void dma_cache_sync(struct device *dev, void *vaddr, size_t size,
            enum dma_data_direction direction)
{
    void *addr;

    addr = __in_29bit_mode() ?
           (void *)CAC_ADDR((unsigned long)vaddr) : vaddr;

    switch (direction) {
    case DMA_FROM_DEVICE:       /* invalidate only */
        __flush_invalidate_region(addr, size);
        break;
    case DMA_TO_DEVICE:     /* writeback only */
        __flush_wback_region(addr, size);
        break;
    case DMA_BIDIRECTIONAL:     /* writeback and invalidate */
        __flush_purge_region(addr, size);
        break;
    default:
        BUG();
    }
}
```

PART / 4

协议栈

套接字缓存(socket buffer, skb)

- **sk_buff是Linux网络协议栈中最重要的数据结构，用于描述已接收或待发送的数据报文信息**
  - 多个不同的网络分层都会使用这个结构，当该结构从一个分层传到另一个分层时，其不同字段会随之发生变化

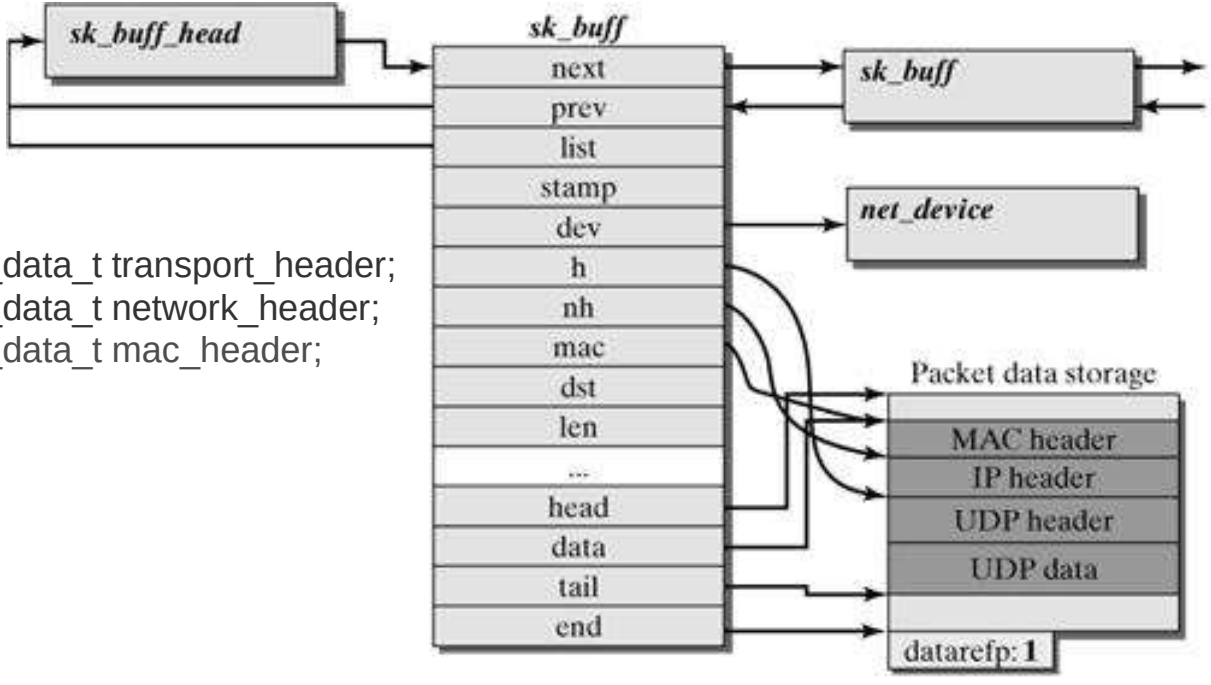
- **设计套接字缓存的初衷**
  - 为了方便地处理可变长缓存，因为接受和发送的数据报长度不是固定的
  - 在添加和移除数据时需要尽量避免数据的复制

sk_buff结构定义

- **UDP数据包分片**
  - next
  - prev
  - rbnode
- **数据包大小**
  - len, datalen

```
1.    //源文件: include/linux/skbuff.h
2.    struct sk_buff {
3.        union {
4.            struct {
5.                struct sk_buff     * next;
6.                struct sk_buff     * prev;
7.                ...
8.            };
9.            struct rb_node   rbnode;
10.       };
11.       unsigned int     len, data_len;
12.       __u16            transport_header;
13.       __u16            network_header;
14.       __u16            mac_header;
15.       sk_buff_data_t   tail;
16.       sk_buff_data_t   end;
17.       unsigned char    * head, * data;
18.       ...
19.   };
```

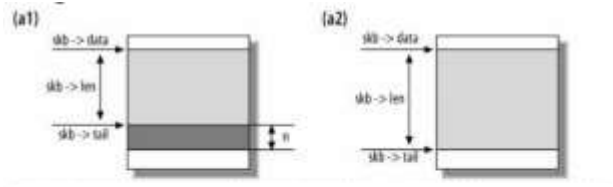
套接字缓存(socket buffer)



sk_buff_data_t transport_header;
sk_buff_data_t network_header;
sk_buff_data_t mac_header;

套接字缓存常用功能函数

- skb_put
- skb_push
- skb_pull
- skb_reserve



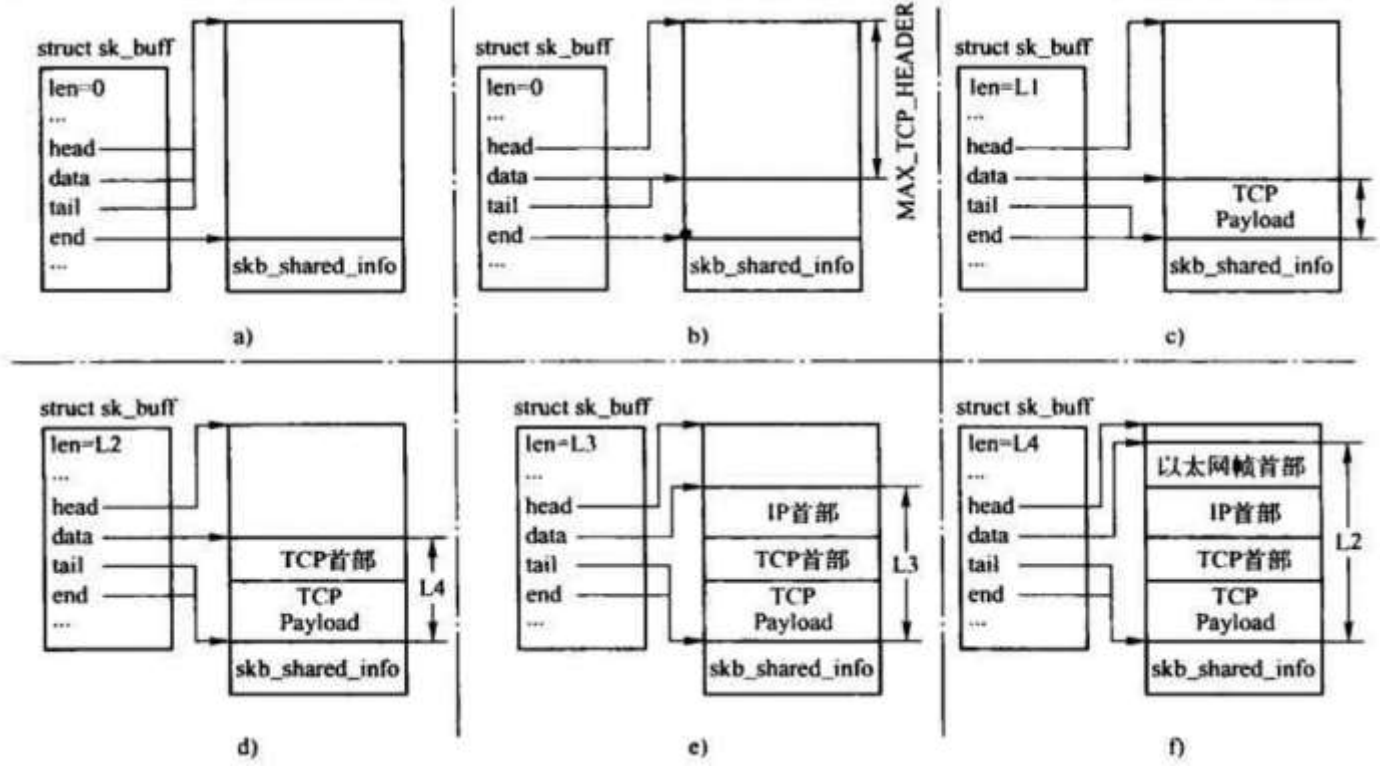

```
void *skb_put(struct sk_buff *skb, unsigned int len)
{
    /* 获取当前skb->tail */
    void *tmp = skb_tail_pointer(skb);
    /* 要求skb数据区必须为线性 */
    SKB_LINEAR_ASSERT(skb);
    /* skb尾部增加len字节 */
    skb->tail += len;
    /* skb数据总长度增加len字节 */
    skb->len  += len;
    /* 如果增加之后的tail > end , 则panic */
    if (unlikely(skb->tail > skb->end))
        skb_over_panic(skb, len, __builtin_return_address(0));
    //返回添加数据的第一个字节位置
    return tmp;
}
```

```
void *skb_push(struct sk_buff *skb, unsigned int len)
{
    /* 数据区data指针前移len字节 */
    skb->data -= len;
    /* 数据总长度增加len字节 */
    skb->len  += len;
    /* 添加数据长度溢出过header , panic*/
    if (unlikely(skb->data < skb->head))
        skb_under_panic(skb, len, __builtin_return_address(0));
    /* 返回新的data指针 */
    return skb->data;
}
```

```
static inline void *__skb_pull(struct sk_buff *skb, unsigned int len)
{
    /* 数据总长度减去len字节 */
    skb->len -= len;
    /* 数据总长度是否有异常 */
    BUG_ON(skb->len < skb->data_len);
    /*
     * data指针移动len字节
     * 返回移除之后新的data指针
     */
    return skb->data += len;
}
```

```
static inline void skb_reserve(struct sk_buff *skb, int len)
{
    /* 数据区data指针增加len字节*/
    skb->data += len;
    /* 数据区tail指针增加len字节 */
    skb->tail += len;
}
```

TCP层向链路层传递时数据的填充过程



拆包过程——从驱动层到网络层

```
1.    //源文件：net/core/dev.c
2.     static int __netif_receive_skb_core(struct sk_buff * skb,
3.                 bool pfmemalloc, struct packet_type ** ppt_prev) {
4.        //根据公式 skb->network_header = skb->data - skb->head
5.        //计算网络层头部相对于内存区头部的偏移
6.        skb_reset_network_header(skb);
7.        skb = skb_vlan_untag(skb);  //如果数据包是 vlan 包，就去掉 vlan 头部
8.        //将数据包交给具体网络层协议的入口函数，协议包括 IPv4、IPv6 或 ARP 等
9.        deliver_skb();
10.       ...
11.   }
```

拆包过程——从网络层到传输层

```
1.    //源文件：net/ipv4/ip_input.c
2.     int ip_local_deliver(struct sk_buff * skb) {
3.        if (ip_is_fragment(ip_hdr(skb))) {        //判断数据包是否分片
4.            //对分片数据包进行重组
5.            ip_defrag(net, skb, IP_DEFRAG_LOCAL_DELIVER);
6.        }
7.        return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN,
8.                net, NULL, skb, skb->dev, NULL,
9.                ip_local_deliver_finish);
10.   }
```

拆包过程——从网络层到传输层

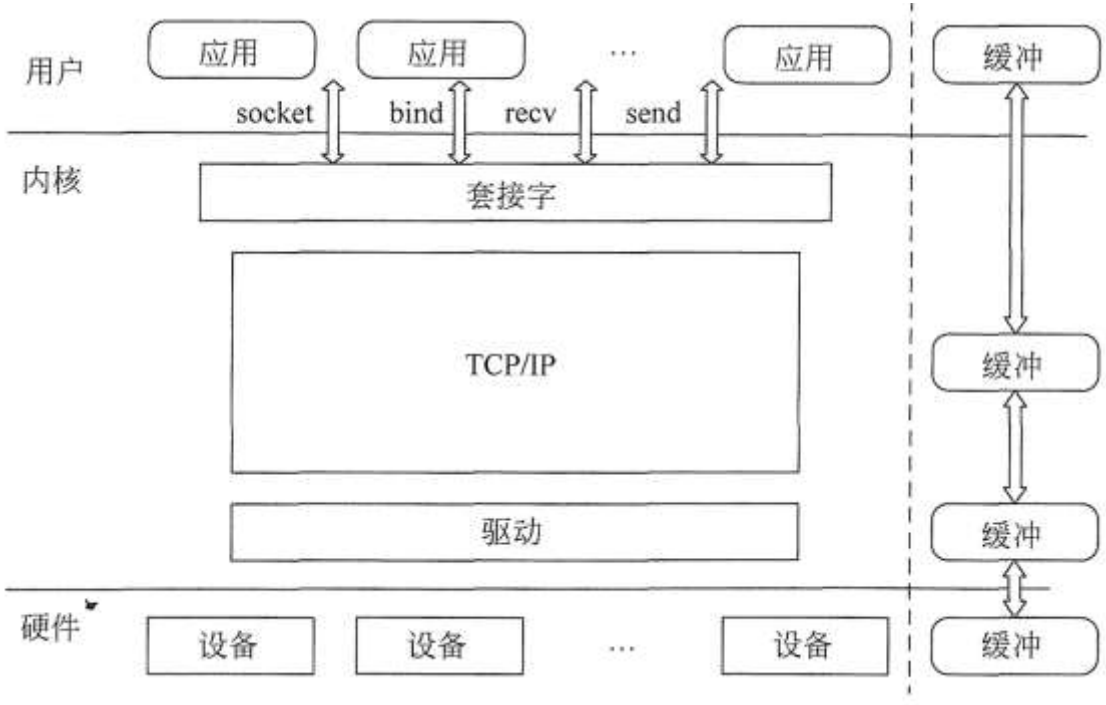
```
/* 如果忽略掉原始套接字和IPSec，则该函数仅仅是根据IP头部中的协议字段选择上层L4协议，并交给它来处理 */
static int ip_local_deliver_finish(struct sk_buff *skb)
{
    /* 跳过IP头部 */
    __skb_pull(skb, ip_hdrlen(skb));

    /* 设置传输层头部位置 */
    skb_reset_transport_header(skb);

    rcu_read_lock();
    {
    ... ...
    /* 查找注册的L4层协议处理结构。 */
        if ((ipprot = rcu_dereference(inet_protos[hash])) != NULL) {
            int ret;
    /* 调用L4层协议处理函数 */
    /* 通常会是tcp_v4_rcv, udp_rcv, icmp_rcv和igmp_rcv */
    /* 如果注册了其他的L4层协议处理，则会进行相应的调用。 */
            ret = ipprot->handler(skb);
        }
 out:
    rcu_read_unlock();

    return 0;
}
```

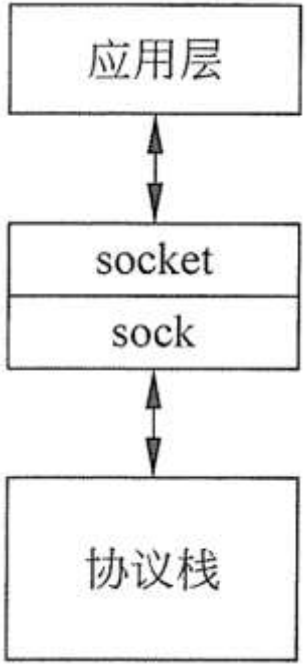
拆包过程——从传输层到接收队列

```
1.    //源文件：net/ipv4/udp.c
2.    int __udp4_lib_rcv(struct sk_buff * skb, struct udp_table * udptable,
3.                                                          int proto) {
4.        if (udp4_csum_init(skb, uh, proto))     //校验数据包
5.            goto csum_error;
6.        //根据 uh->source 和 uh->dest 查找 udptable 获得结构体 sock
7.        struct sock * sk = __udp4_lib_lookup_skb(skb, uh->source,
8.                                                 uh->dest, udptable);
9.        udp_unicast_rcv_skb(sk, skb, uh);       //将 skb 存入 sk 的接收队列
10.   }
```

- 对收到的数据包进行校验
- 将sk_buff添加到接收队列sock->sk_receive_queue

TCP/IP协议栈与Socket层的接口

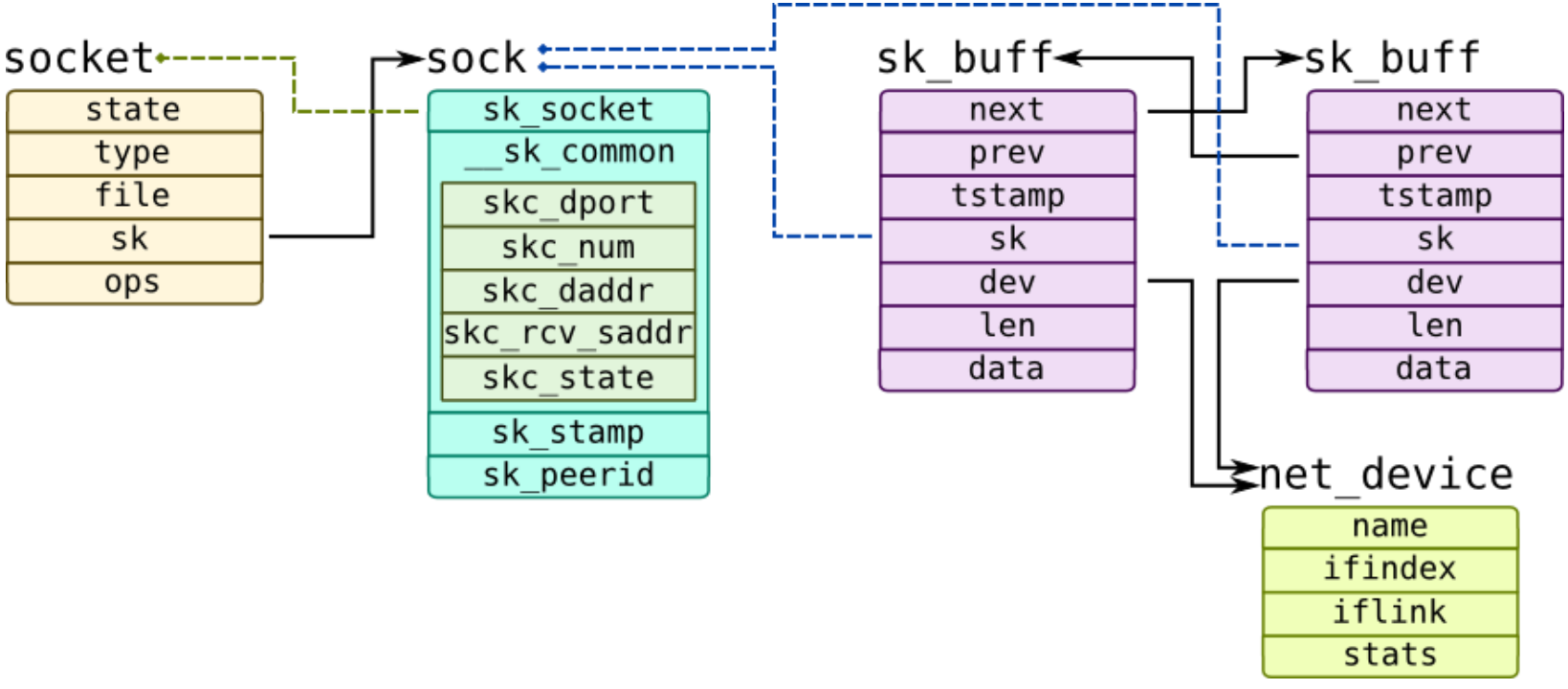




Socket数据结构

```
1.    //源文件：include/linux/net.h
2.    struct socket {
3.        //Socket 类型可以是流式 Socket、数据报 Socket 或者原始 Socket
4.        short   type;
5.        //Socket 对应的结构体 file，使得用户可以通过文件描述符操作 Socket
6.        struct file   * file;
7.        sock    * sk;
8.        //Socket 的相关操作，如 connect、listen 等
9.        const struct proto_ops   * ops;
10.       ...
11.   };
```

对象关系图

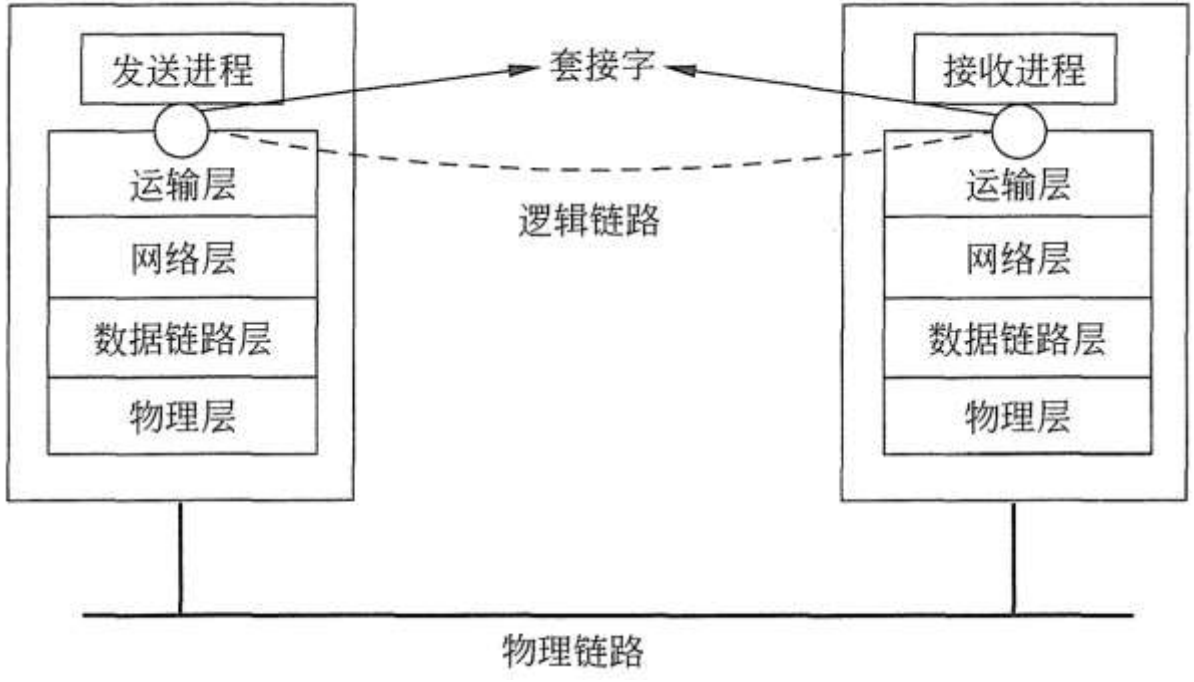

socket
state
type
file
sk
ops
sock
sk_socket
__sk_common
skc_dport
skc_num
skc_daddr
skc_rcv_saddr
skc_state
sk_stamp
sk_peerid
sk_buff
next
prev
tstamp
sk
dev
len
data
sk_buff
next
prev
tstamp
sk
dev
len
data
net_device
name
ifindex
iflink
stats

PART / 5

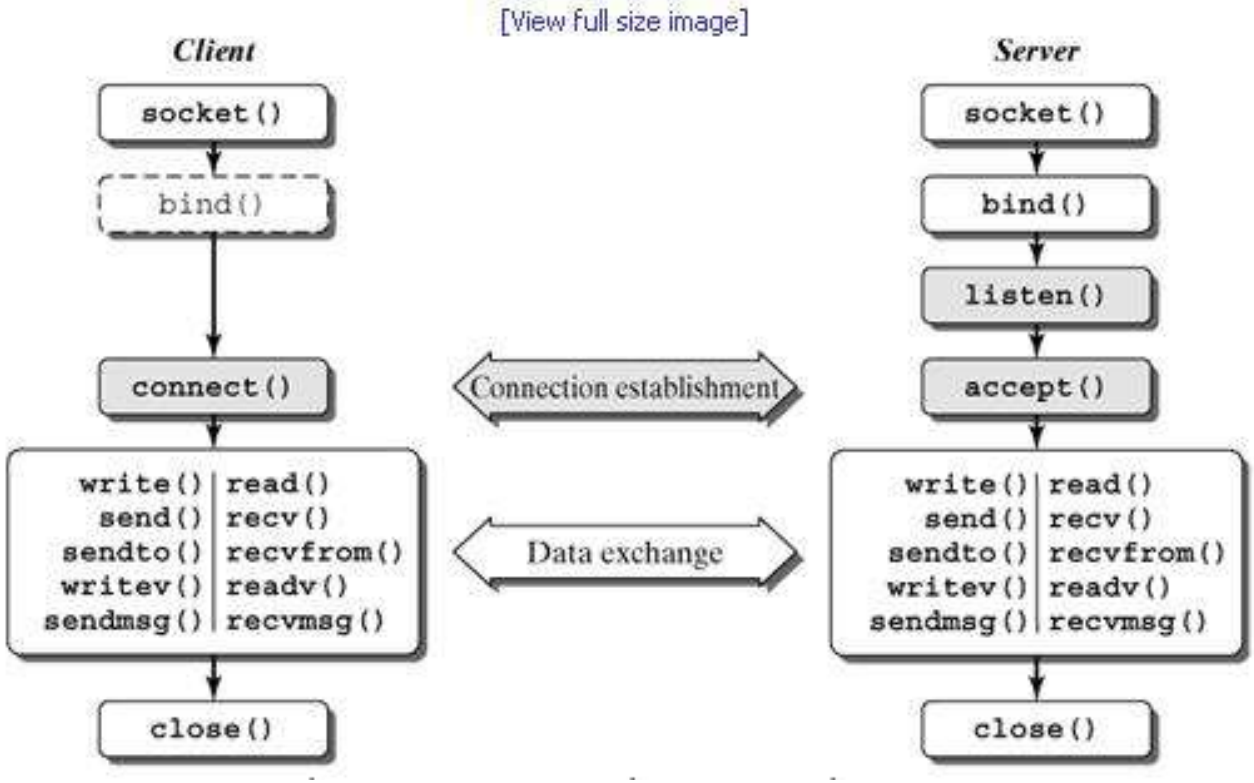
套接字

Socket套接字

- **进程间通信的抽象**
  - 本地与网络
- **网络编程的入口**
  - 位于传输层协议之上
  - 屏蔽了不同网络协议之间的差异
  - 提供了大量的系统调用，
  - 构成了网络程序的主体
  - 在Linux系统中，Socket属于文件系统的一部分
- **一个Socket对象表示通信双方中的一方**



套接字操作


[View full size image]
Client
Server
socket()
bind()
connect()
write()
read()
send()
recv()
sendto()
recvfrom()
writev()
readv()
sendmsg()
recvmsg()
close()
Connection establishment
Data exchange
socket()
bind()
listen()
accept()
write()
read()
send()
recv()
sendto()
recvfrom()
writev()
readv()
sendmsg()
recvmsg()
close()

系统调用接口

- **主要功能**
  - 实质是一个面向用户空间应用程序的接口调用库，向用户空间应用程序提供使用网络服务的接口
- **用户发起网络调用**
  - 通过系统特有的Socket网络调用进入内核
  - 最终调用sys_socketcall(./net/socket.c），在sys_socketcall()中会根据网络系统调用号调用具体的功能

```
SYSCALL_DEFINE3(socket, int, family, int, type, int, protocol)
SYSCALL_DEFINE3(bind, int, fd, struct sockaddr __user *, umyaddr, int, addrlen)
SYSCALL_DEFINE2(listen, int, fd, int, backlog)
SYSCALL_DEFINE3(accept, int, fd, struct sockaddr __user *, upeer_sockaddr, int __user *, upeer_addrlen)
```

- **普通文件操作作为网络 I/O**
  - 套接口的输入/输出操作可以当作典型的文件读写操作来进行
  - 特定于网络的操作如调用socket创建socket，调用connect将socket连接到目的地等

套接字地址族、类型和协议

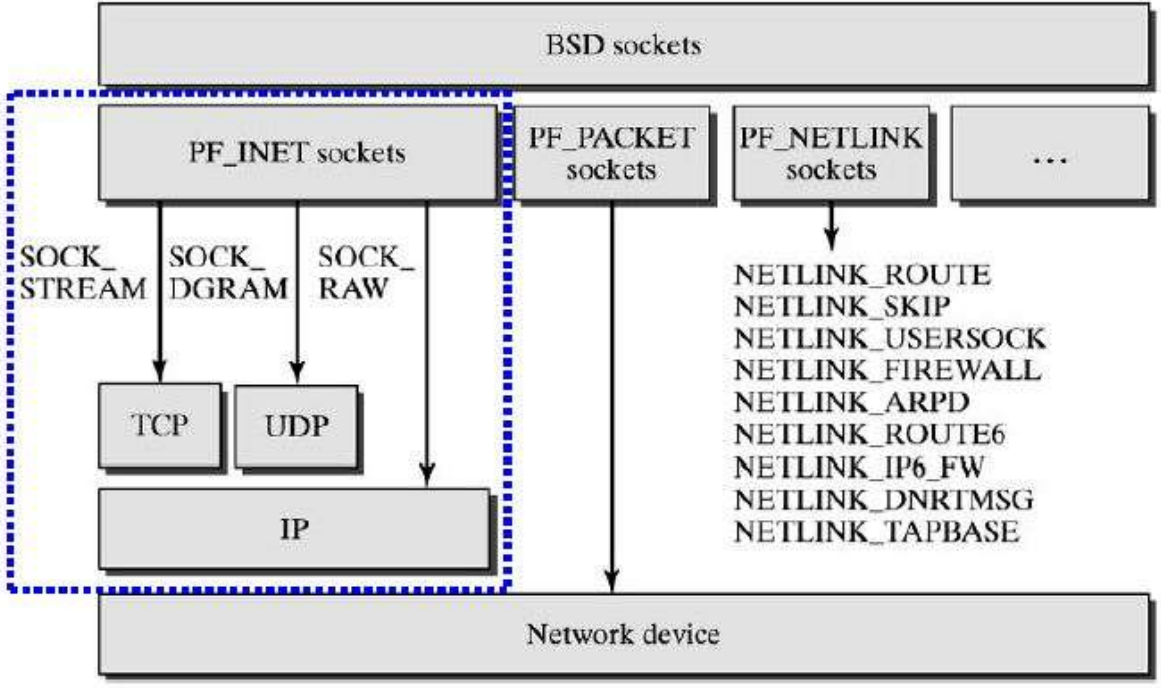
名称	含义	名称	含义
PF_UNIX,PF_LOCAL	本地通信	PF_X25	ITU-T X25 / ISO-8208协议
AF_INET,PF_INET	IPv4 Internet协议	PF_AX25	Amateur radio AX.25
PF_INET6	IPv6 Internet协议	PF_ATMPVC	原始ATM PVC访问
PF_IPX	IPX-Novell协议	PF_APPLETALK	Appletalk
PF_NETLINK	内核用户界面设备	PF_PACKET	底层包访问

类型	说明
SOCK_STREAM	字节流套接字
SOCK_DGRAM	数据报套接字
SOCK_RAW	原始套接字

协议	说明
IPPROTO_TCP	TCP传输协议
IPPROTO_UDP	UDP传输协议
IPPROTO_IP	IP传输协议
IPPROTO_ICMP	ICMP传输协议（询问、差错）

socket (int family, int type, int protocol)

BSD Sockets


BSD sockets
PF_INET sockets
PF_PACKET sockets
PF_NETLINK sockets
...
SOCK_ STREAM
SOCK_ DGRAM
SOCK_ RAW
TCP
UDP
IP
NETLINK_ROUTE
NETLINK_SKIP
NETLINK_USERSOCK
NETLINK_FIREWALL
NETLINK_ARPD
NETLINK_ROUTE6
NETLINK_IP6_FW
NETLINK_DNRTMSG
NETLINK_TAPBASE
Network device

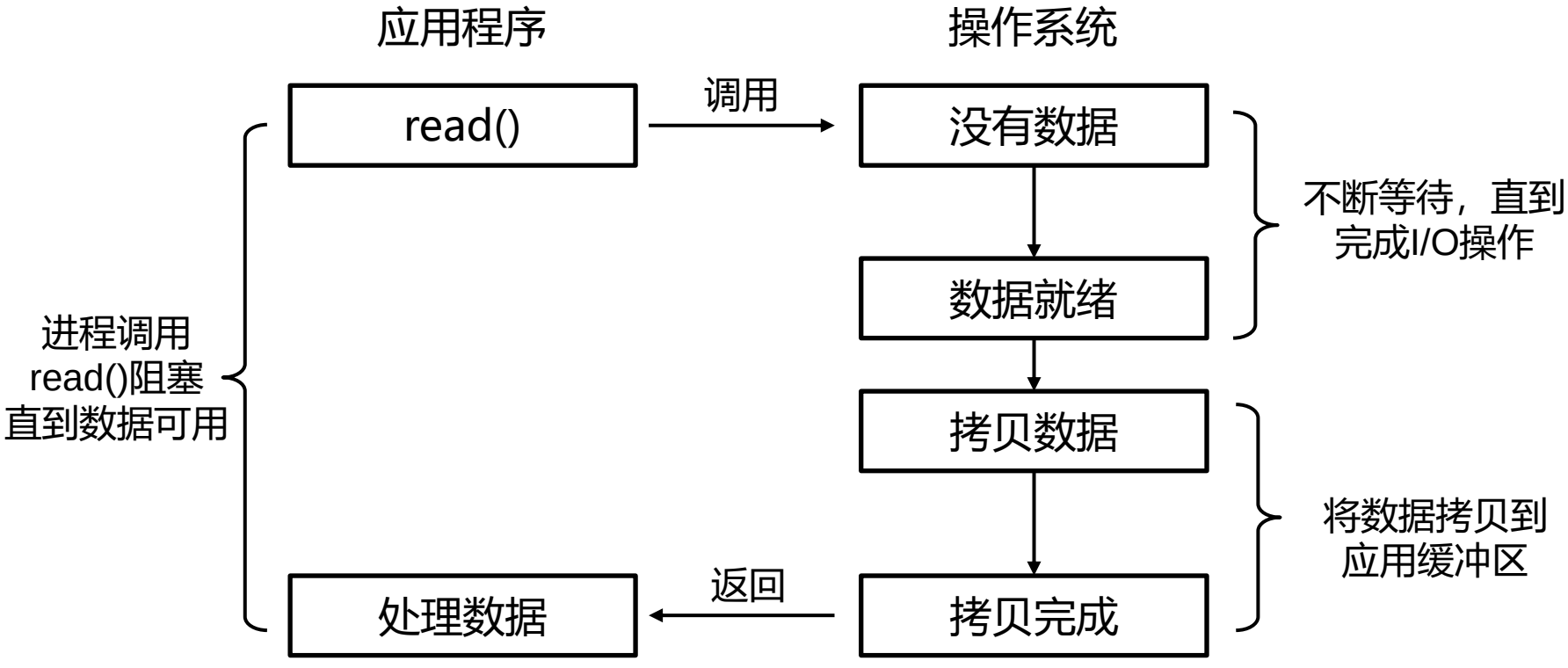
套接字地址

- 协议族
- IP地址
- 端口号

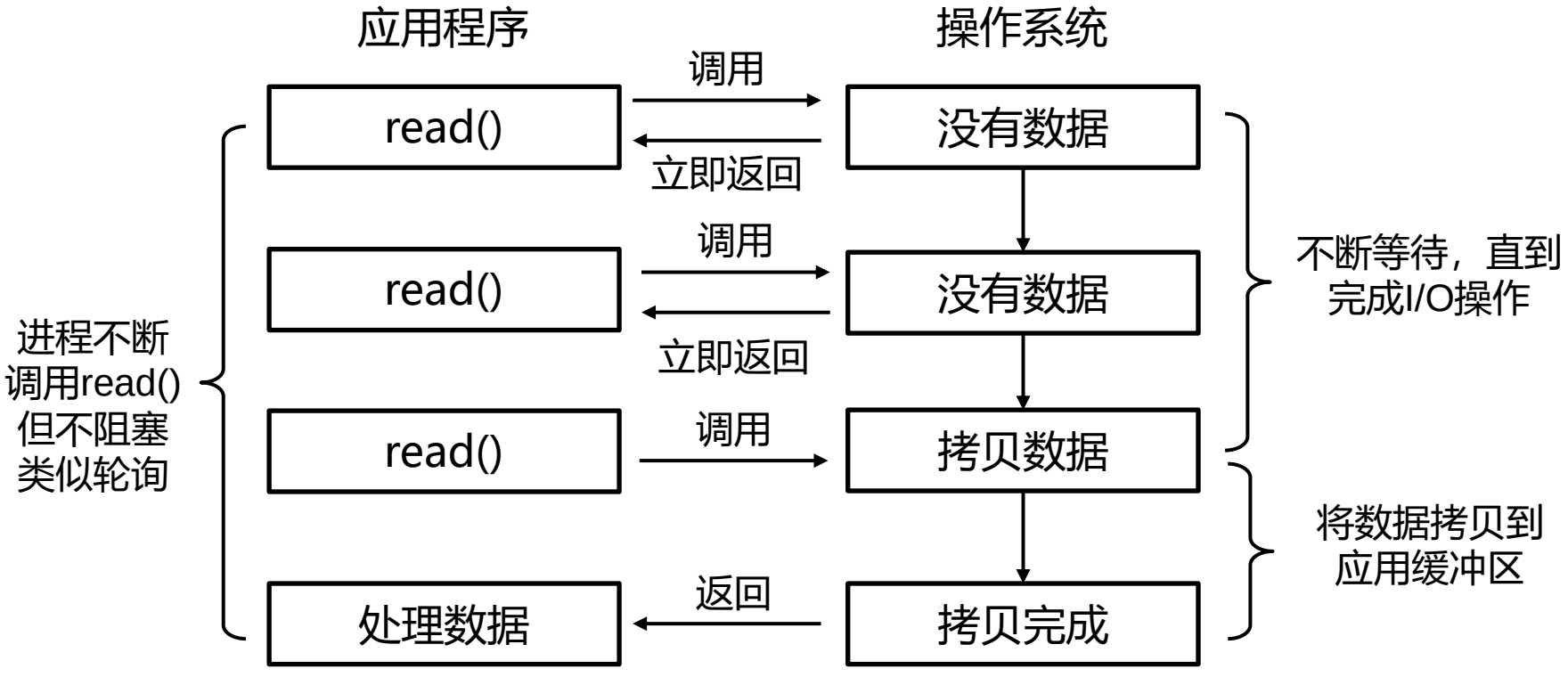
int bind(int sockfd, struct sockaddr *uaddr, socketlen_t uaddrlen)

```
//usr/include/sys/socket.h
typedef unsigned short    sa_family_t;
//通用socket地址
struct sockaddr {
    sa_family_t    sa_family;    /* address family, AF_xxx,协议簇*/
    char        sa_data[14];    /* 14 bytes of protocol address    */
};

//usr/include/netinet/in.h
//INET地址簇的socket地址
struct in_addr {
                    __u32 s_addr;
};
struct sockaddr_in   {
   sa_family_t          sin_family;     /* Address family: AF_INET */
   unsigned short int   sin_port;       /* Port number,端口*/
   struct in_addr       sin_addr;       /* Internet address,IP地址*/

   /* Pad to size of 'struct sockaddr' . */
   unsigned char sin_zero[sizeof (struct sockaddr) -
                          sizeof (sa_family_t) -
                          sizeof (uint16_t) -
                          sizeof (struct in_addr)];
};
```

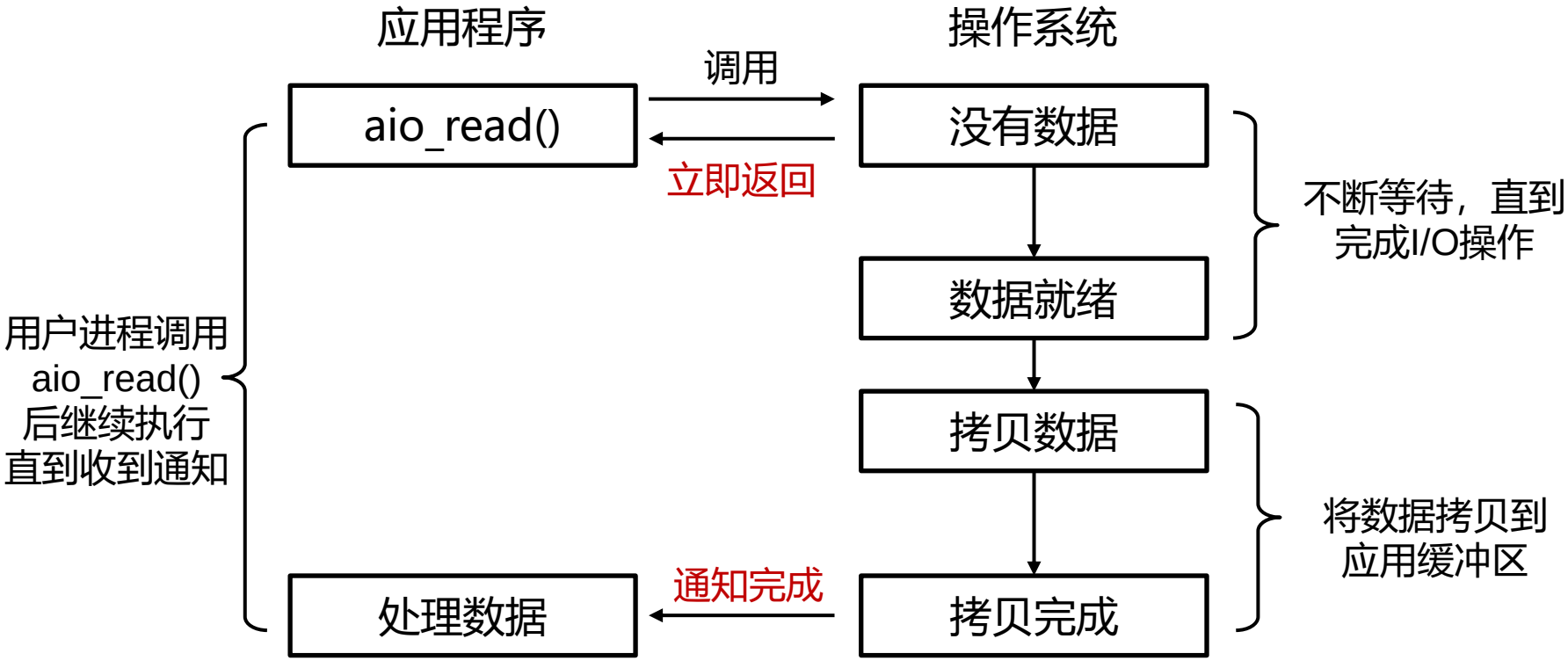
I/O模型：阻塞I/O模型


应用程序
操作系统
read()
调用
没有数据
数据就绪
拷贝数据
拷贝完成
返回
处理数据
不断等待，直到
完成I/O操作
将数据拷贝到
应用缓冲区
进程调用
read()阻塞
直到数据可用

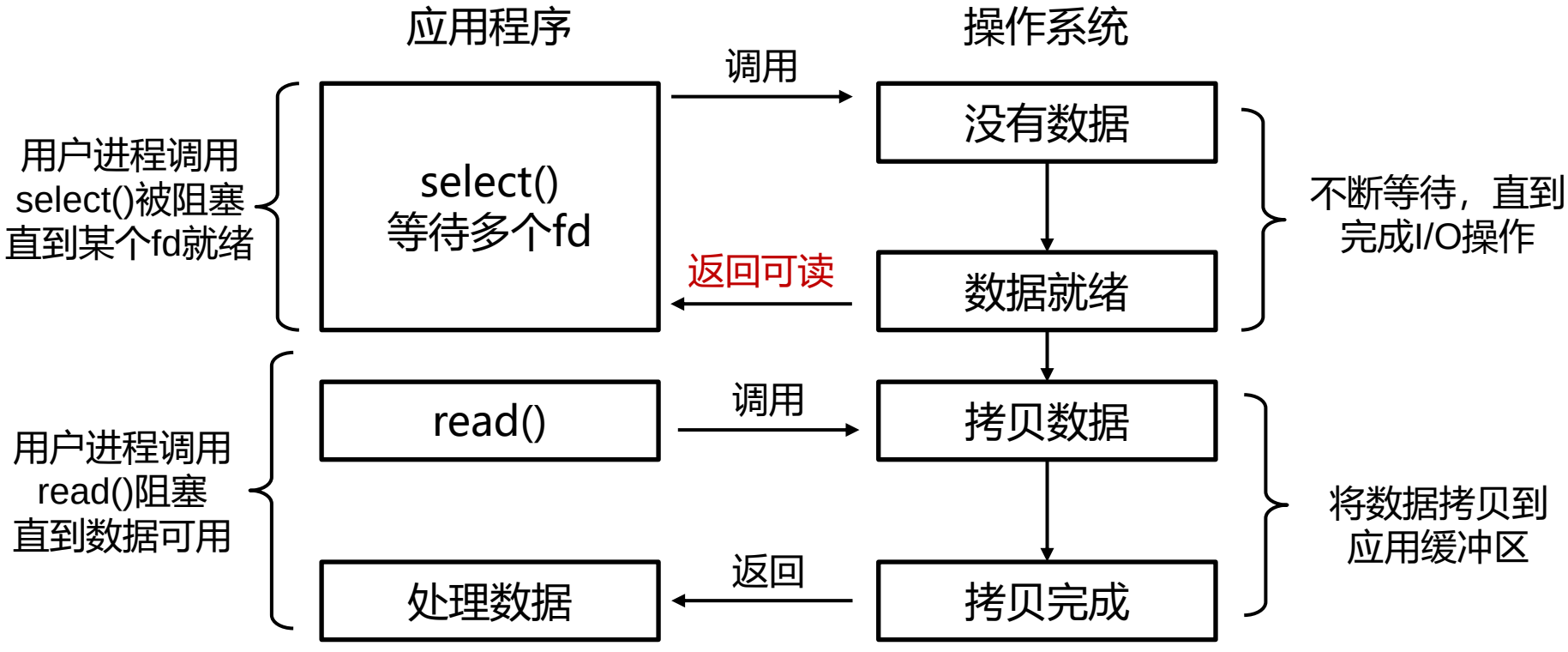
I/O模型：非阻塞I/O模型



I/O模型：异步I/O模型


应用程序
操作系统
调用
aio_read()
立即返回
没有数据
数据就绪
拷贝数据
拷贝完成
通知完成
处理数据
不断等待，直到
完成I/O操作
将数据拷贝到
应用缓冲区
用户进程调用
aio_read()
后继续执行
直到收到通知

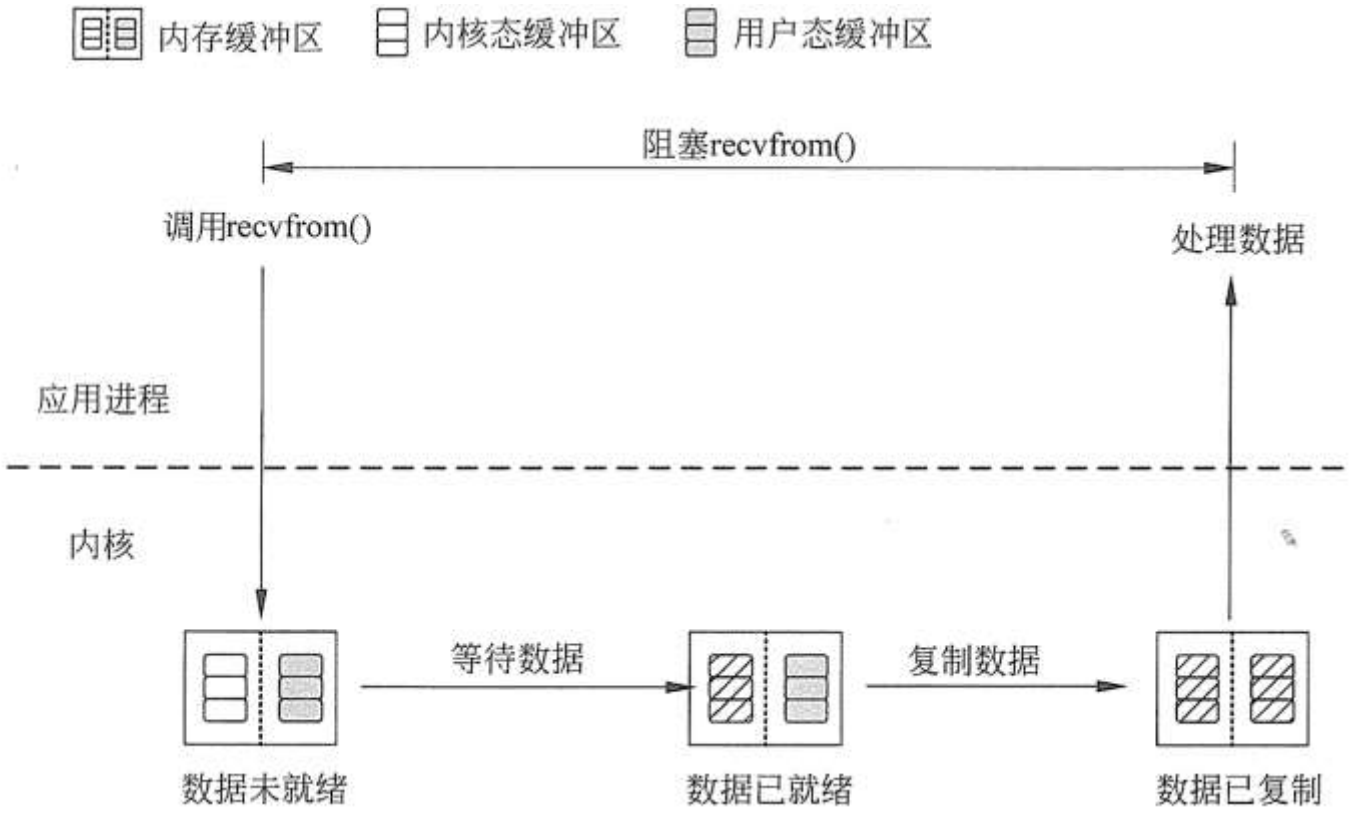
I/O模型：I/O多路复用模型



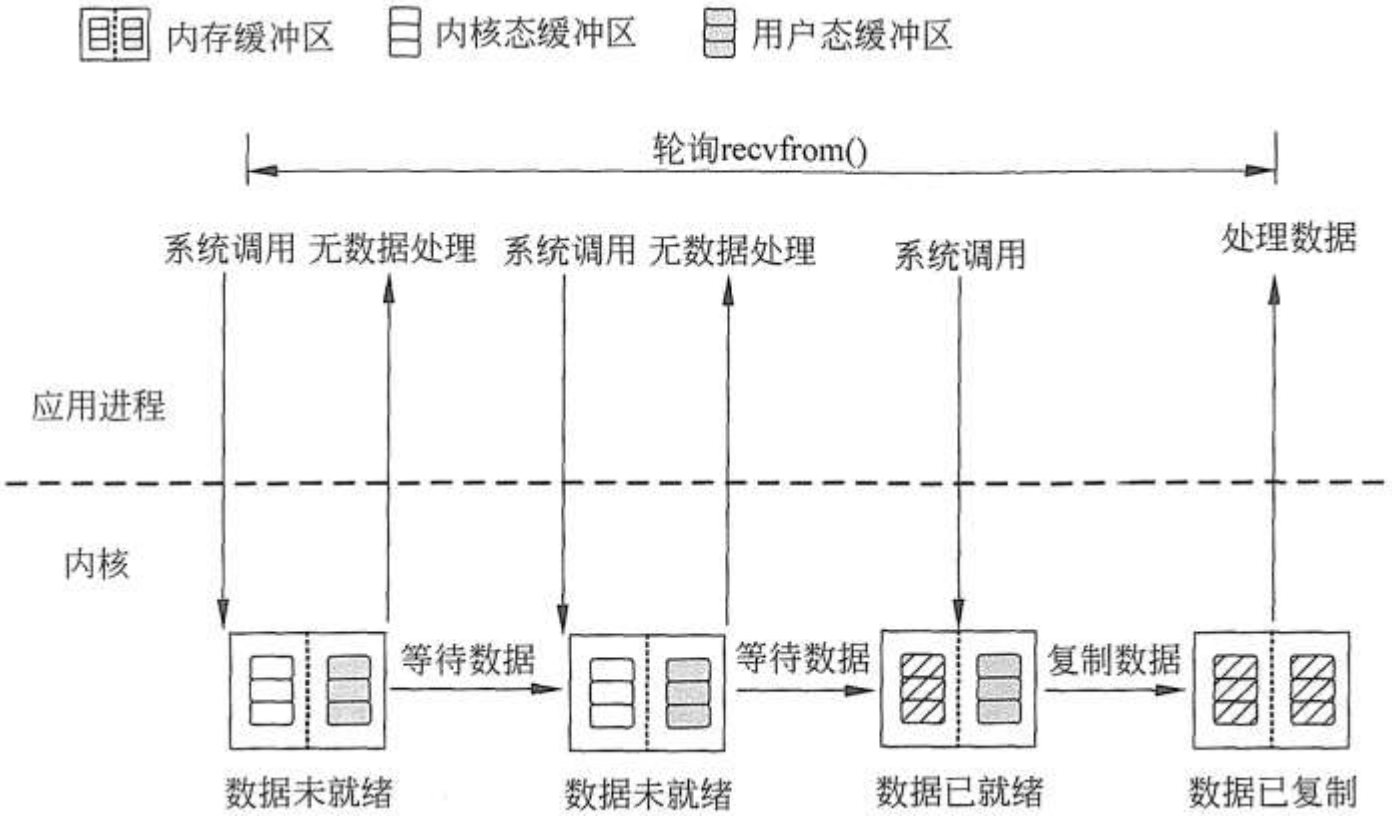
I/O模型小节

- **阻塞I/O：一直等待**
  - 进程请求读数据不得，将其挂起，直到数据来了再将其唤醒
  - 进程请求写数据不得，将其挂起，直到设备准备好了再将其唤醒
- **非阻塞I/O：不等待**
  - 读写请求后直接返回（可能读不到数据或者写失败）
- **异步I/O：稍后再来**
  - 等读写请求成功后再通知用户
  - 用户执行并不停滞（类似DMA之于CPU）
- **I/O多路复用：同时监听多个请求，只要有一个就绪就不再等待**

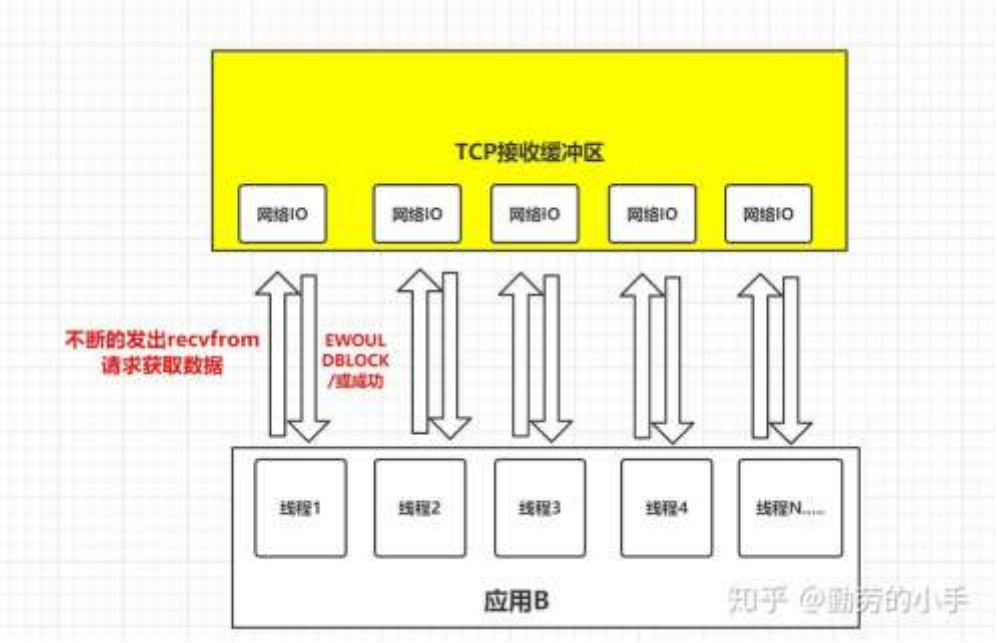
阻塞Socket I/O模型

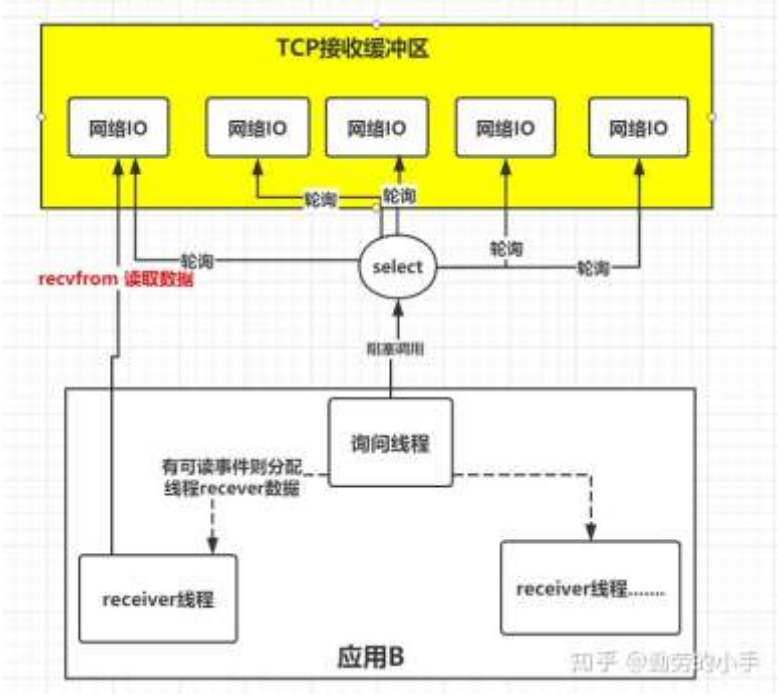


非阻塞Socket I/O模型

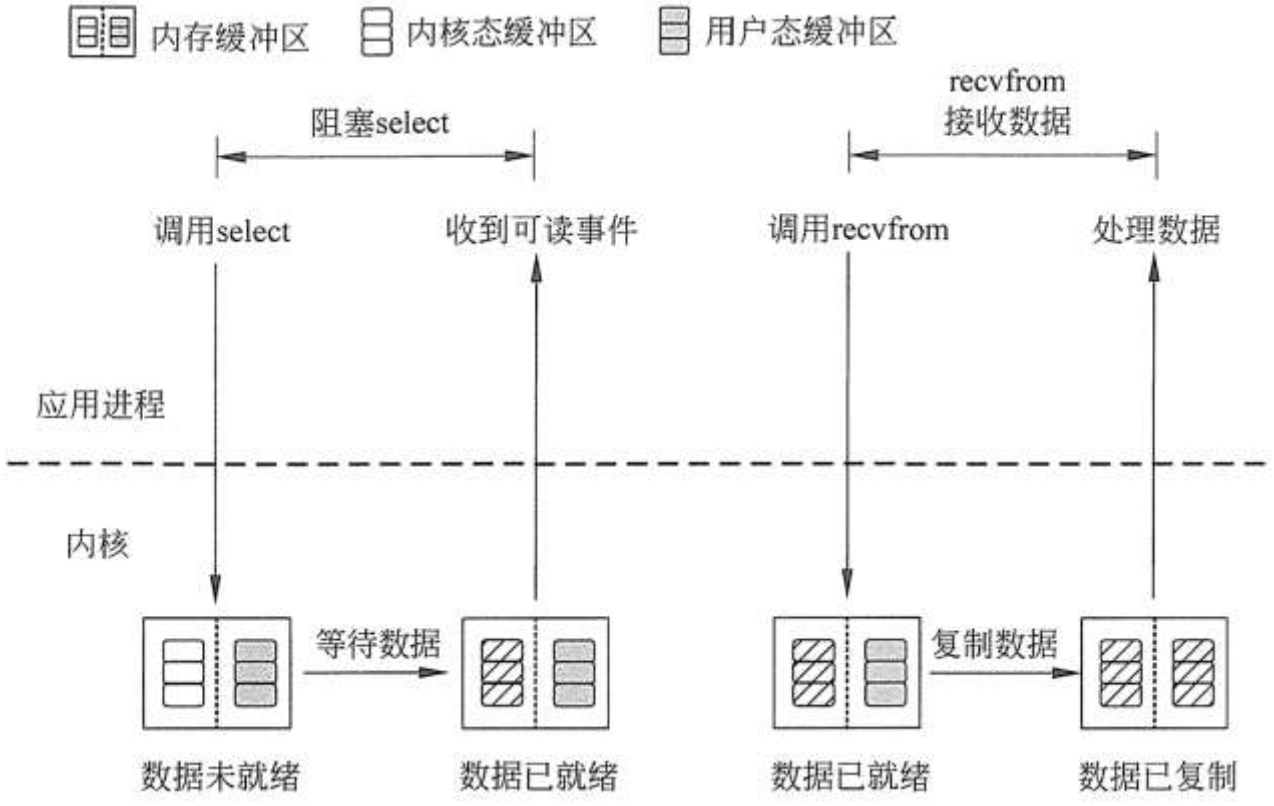


Socket I/O复用模型





Socket I/O复用模型



Select核心函数

```
1.    //源文件：fs/select.c
2.    //mask: POLLIN_SET(fd 可读),POLLOUT_SET(fd 可写),POLLEX_SET(fd 异常)
3.    static int do_select(int n, fd_set_bits * fds,
4.                         struct timespec64 * end_time) {
5.        for (;;) {
6.            for (i = 0; i < n; ++rinp, ++routp, ++rexp) {
7.                mask = vfs_poll(f.file, wait);   //调用轮询函数查询设备状态
8.            //如果没有轮询到对应事件,那么阻塞该进程,并设置为 TASK_INTERRUPTIBLE
9.                poll_schedule_timeout(&table, TASK_INTERRUPTIBLE,
10.                                                    to, slack);
11.               //根据变量 mask 将结果存入变量 res_in,最后再存入 fds 集合
12.               if ((mask & POLLIN_SET) && (in & bit)) {
13.                   res_in | = bit;
14.               }
15.             ...
16.           }
17.       }
18.   }
```

Select与epoll

- **Select**
  - 优点：可以使进程在不被阻塞的前提下监听多个 Socket 描述符
  - 缺点：在可能有上百万连接的高并发场合，当某连接的数据就绪时，操作系统可能要遍历所有的 Socket 描述符
- **Epoll**
  - 将就绪的而不是所有的 Socket 描述符返回给进程
  - 避免像 select 那样让进程遍历所有描述符
  - epoll 支持的描述符数量没有限制
  - 较好地胜任高并发的场合

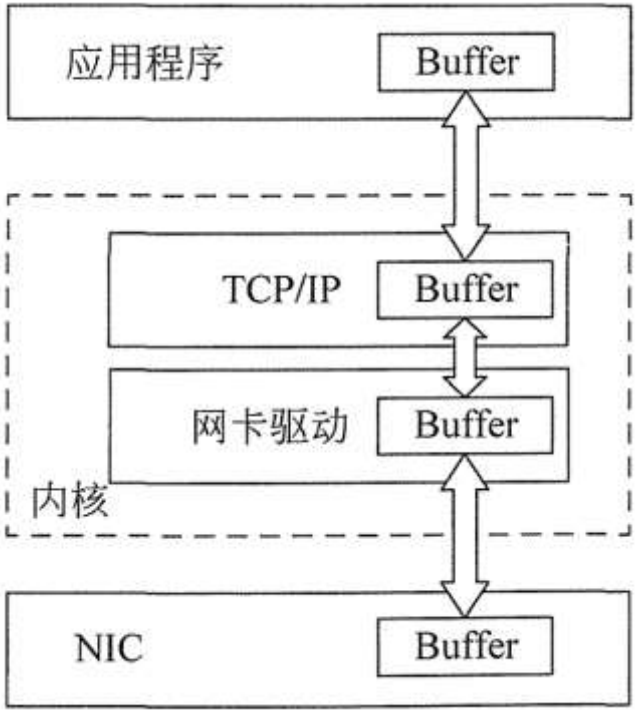
PART / 6

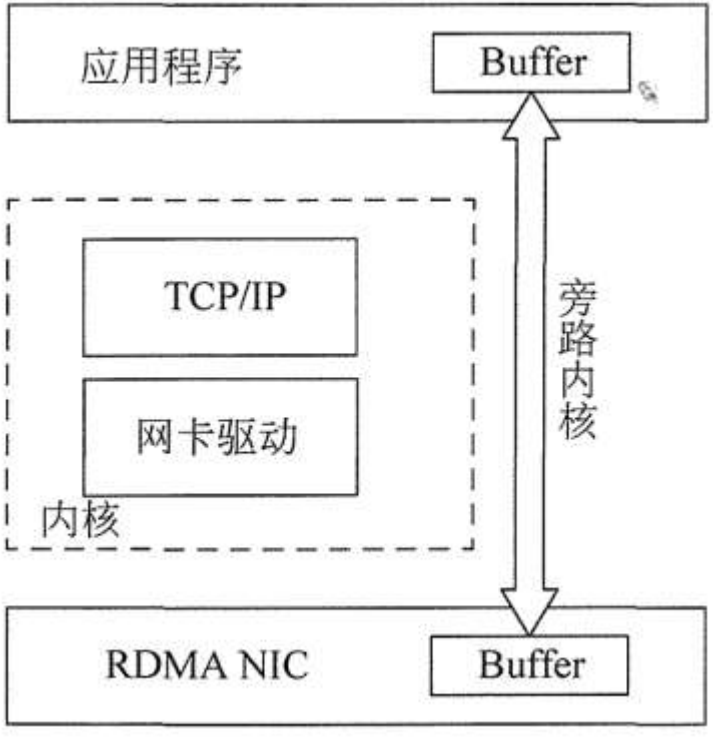
新型网络加速技术

新型网络加速技术

- **10Gb/s网卡已在数据中心普及**
- **100Gb/s网卡的应用已成为趋势**
- **CPU性能增速放缓**
- **基于操作系统内核的TCP/IP协议栈主要运行在主机CPU上**
  - 以软件方式运行
- **I/O处理消耗了大量主机CPU周期，留给应用逻辑的CPU性能不足**
- **基于内核的网络软处理开销日益成为服务器中的性能瓶颈**

RDMA

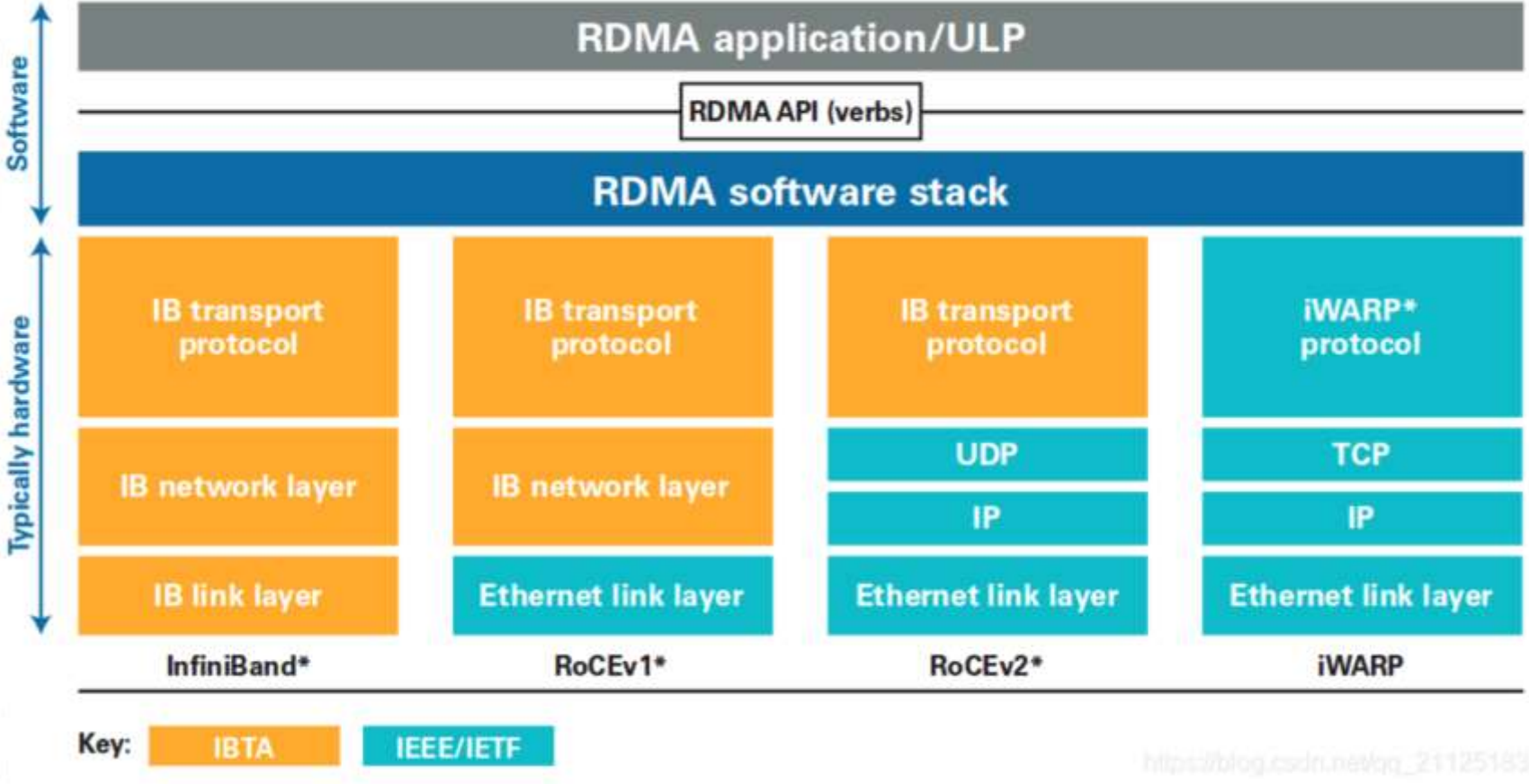

应用程序
Buffer
TCP/IP
Buffer
网卡驱动
Buffer
内核
NIC
Buffer


应用程序
Buffer
TCP/IP
网卡驱动
内核
旁路内核
RDMA NIC
Buffer

RDMA的性能优势

- 绕过主机 CPU
  - 应用程序可直接访问远程机中的内存，而不消耗远程主机的 CPU 周期
- 绕过内核
  - 应用程序之间直接在用户空间进行数据传输，而不涉及内核态与用户态的切换
- 零拷贝
  - 应用程序只需将数据发送到缓冲区或从缓冲区接收数据。数据不需要在用户空间和内核空间来回拷贝

RDMA的三种实现

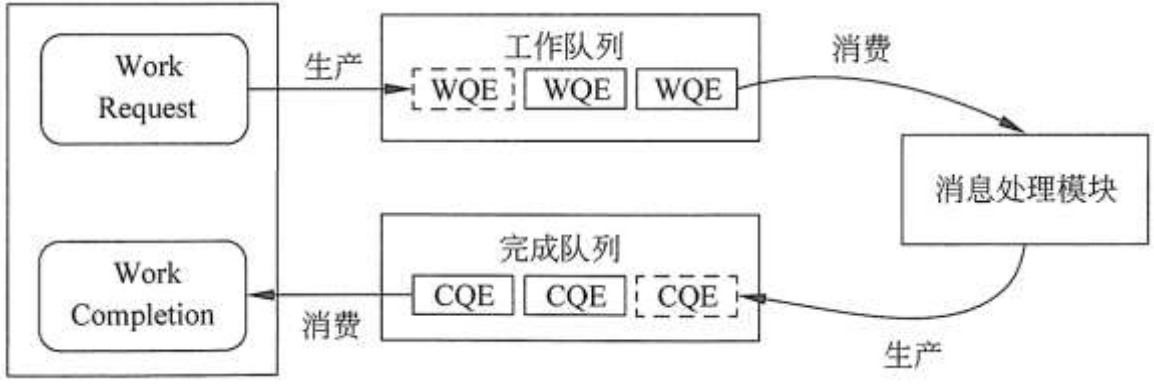

Software
Typically hardware
RDMA application/ULP
RDMA API (verbs)
RDMA software stack
IB transport protocol
IB network layer
IB link layer
InfiniBand*
IB transport protocol
IB network layer
Ethernet link layer
RoCEv1*
IB transport protocol
UDP
IP
Ethernet link layer
RoCEv2*
iWARP* protocol
TCP
IP
Ethernet link layer
iWARP
Key:
IBTA
IEEE/IETF

内存注册（Memory Registration）

- **解决问题**
  - 应将数据发送到对端内存的哪个位置
  - 如何获得访问权限?
- **应用程序将内存与key绑定，并将结果注册到网卡上**
- **拥有key表示拥有对该内存的访问权限**
  - Local key
  - Remote key

RDMA队列

- 如何组织管理消息，使得异步收发更高效
- 网卡内部队列
- **Work queue**
  - Send Queue
  - Receive Queue
- **Completion Queue**

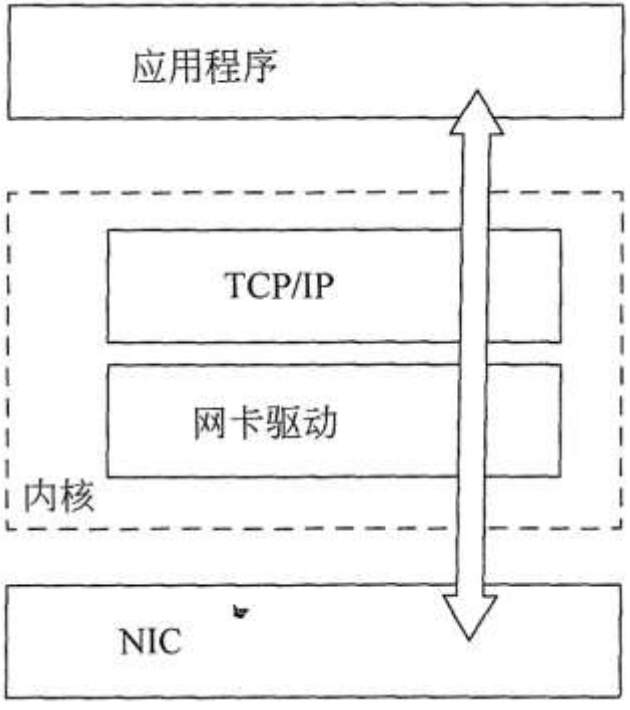


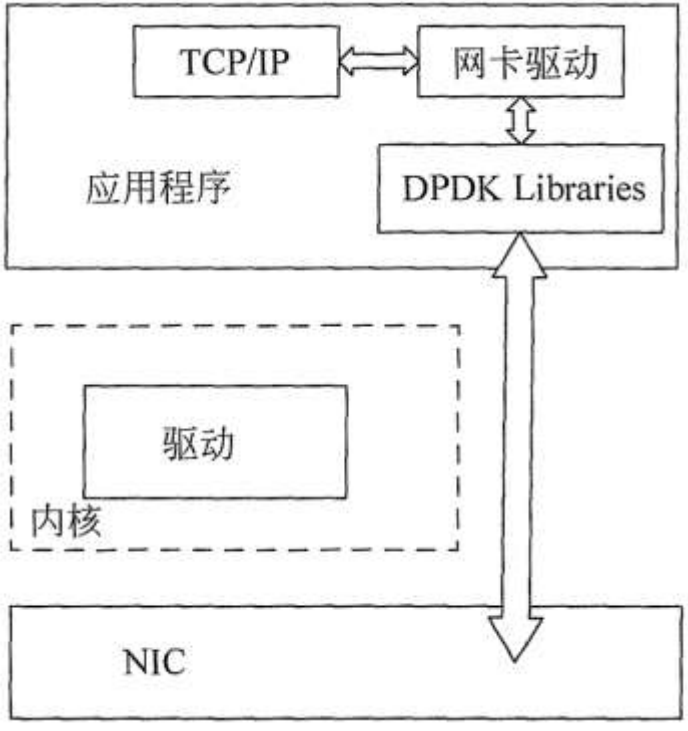
Data Plane Development Kit，DPDK

- **Data Plane vs Control Plane**
- **Intel 提供的用户态数据面解决方案**
  - 将数据面从内核中分离出来
  - 将访问控制和硬件配置等控制面功能依然保留在内核中
- **提供一个能够直接与硬件设备进行交互的用户态库**
  - 使得应用程序可以直接操作I/O数据
  - 在用户空间重载网卡驱动
  - 摒弃了 Linux 内核协议栈，而采用用户态的协议栈
  - 收到数据包后，用户态的网卡驱动将数据包直接存入应用程序的缓存

零复制、无系统调用

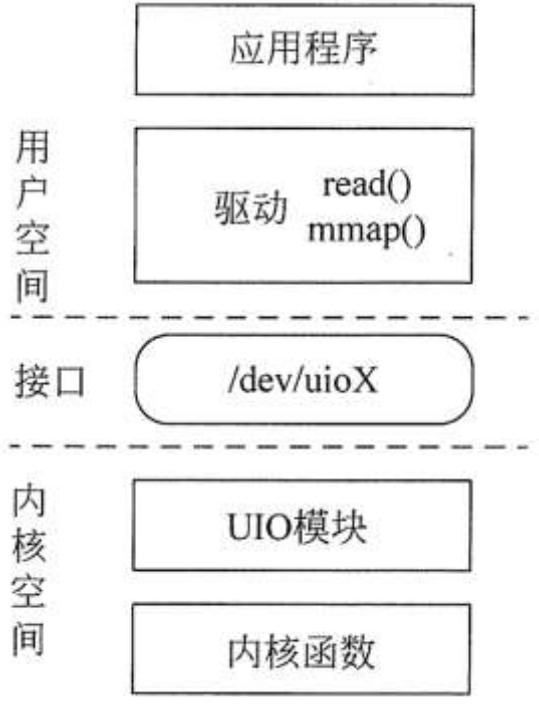
Data Plane Development Kit，DPDK





Data Plane Development Kit，DPDK

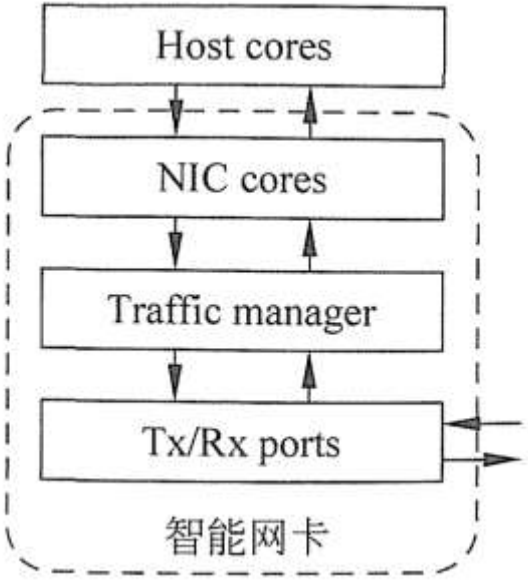
- **Uio**
  - 让网卡驱动运行在用户空间
  - 通过mmap将设备内部存储空间映射到用户空间
  - 组成文件的形式供应用程序使用
- **Poll Mode Driver**
  - 屏蔽网卡中断，改用轮询方式
  - 以轮询方式直接访问保存在用户态内存空间的Rx ring描述符
- **Interrupt DPDK**
  - 类似于NAPI
  - 有数据接收时轮询
  - 无数据接收时空闲



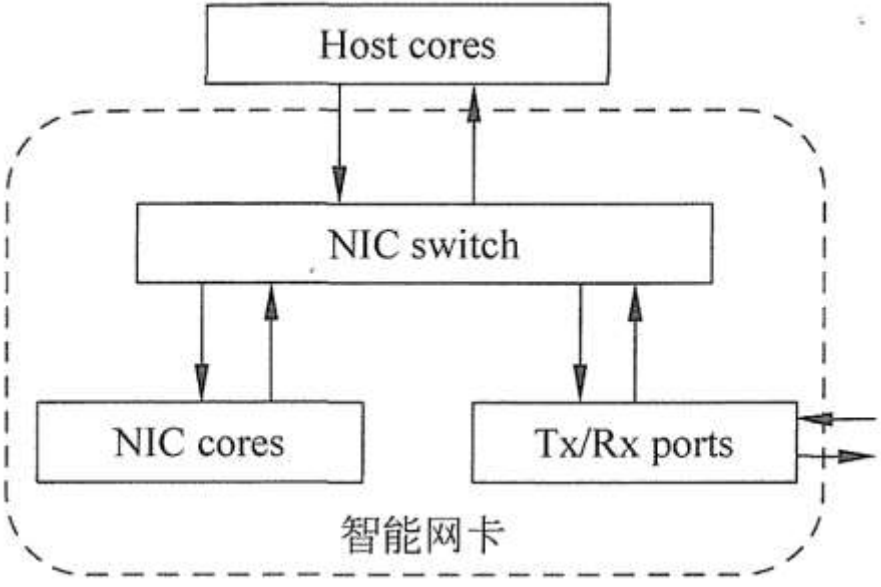
智能网卡

- **传统网卡仅实现数据链路层和物理层功能**
- **智能网卡还支持协议栈、网络虚拟化等功能**
- **智能网卡通常集成了通用计算单元，具有可编程能力**
  - 传统的网络功能
    - 虚拟交换机
    - 负载均衡
  - 通用计算任务
- **实现方式**
  - ASIC、FPGA、SoC

On-path vs off-path

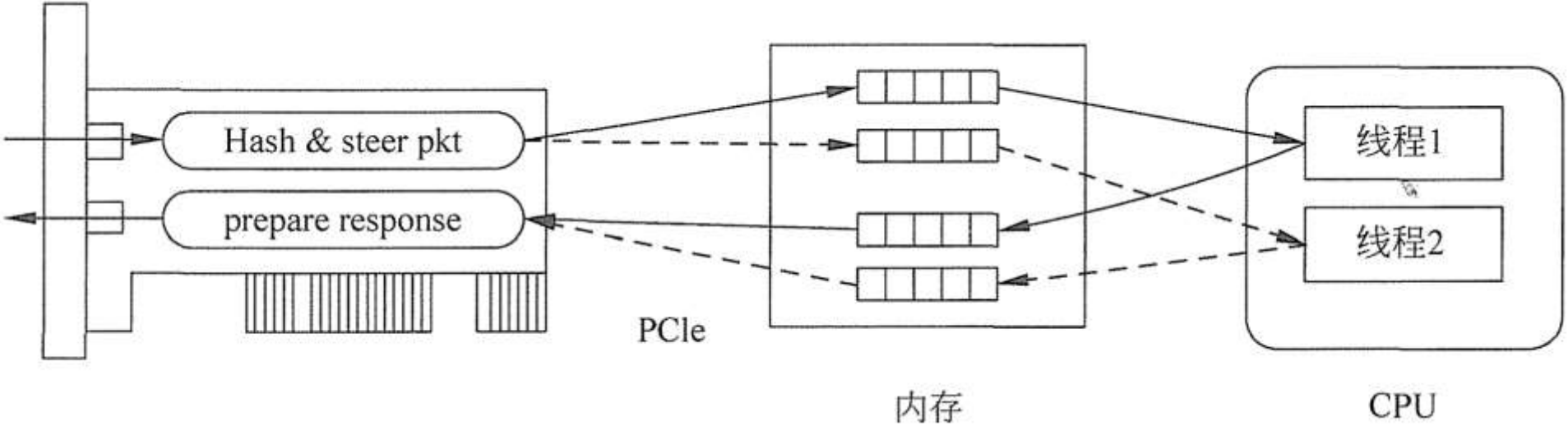


On-path



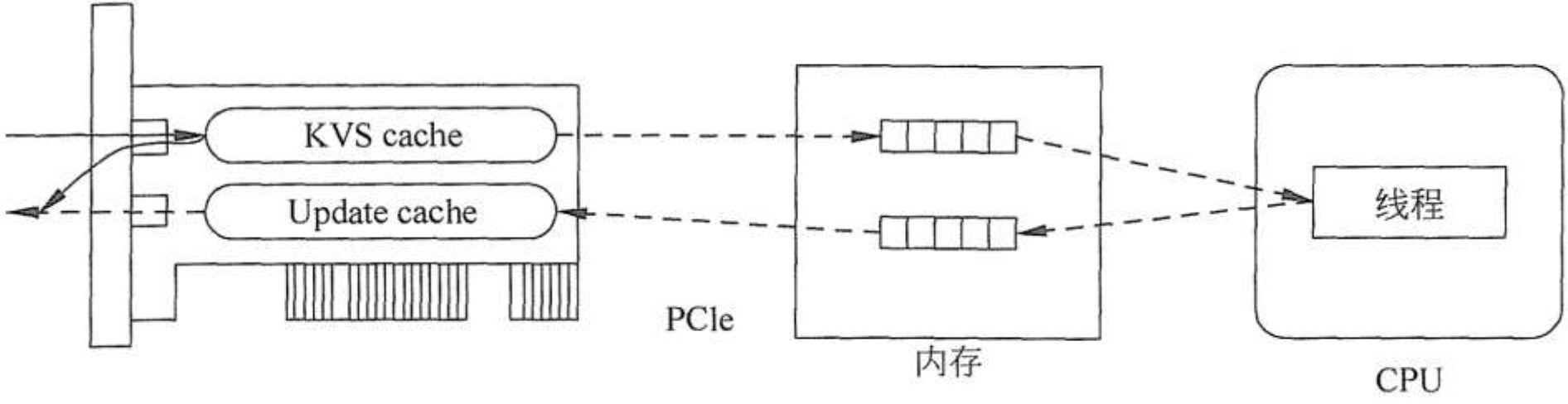
Off-path

智能网卡的应用1—KV-store



计算哈希值，增删改查操作

智能网卡应用2—将网卡作为缓存



软件定义网络SDN

传统网络设备

- 通过自学习的方式建立转发表
- 静态等级架构
- 网络维护成本高、容易出错

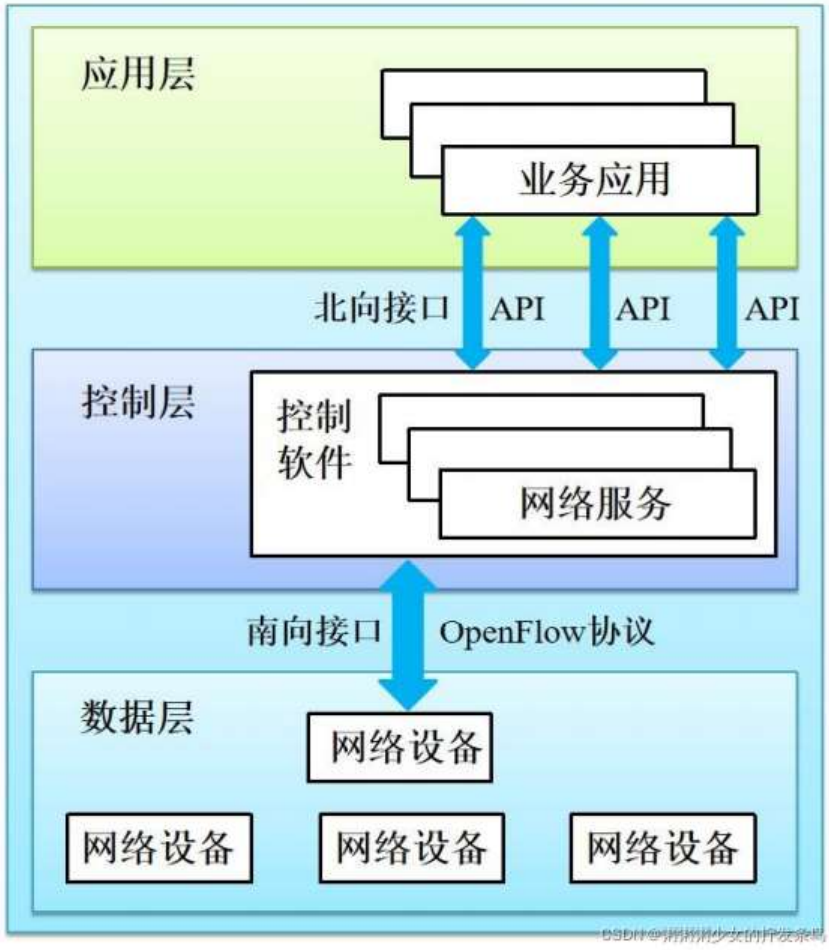
SDN

- 动态调整网络架构
- 从全局对网络设备整体部署
- 将通信设备的控制平面与数据平面解耦
- 将网络上所有通信设备的控制平面集中起来实现统一管理

被MIT Technology Review评为2009年“全球十大突破性技术”

SDN架构

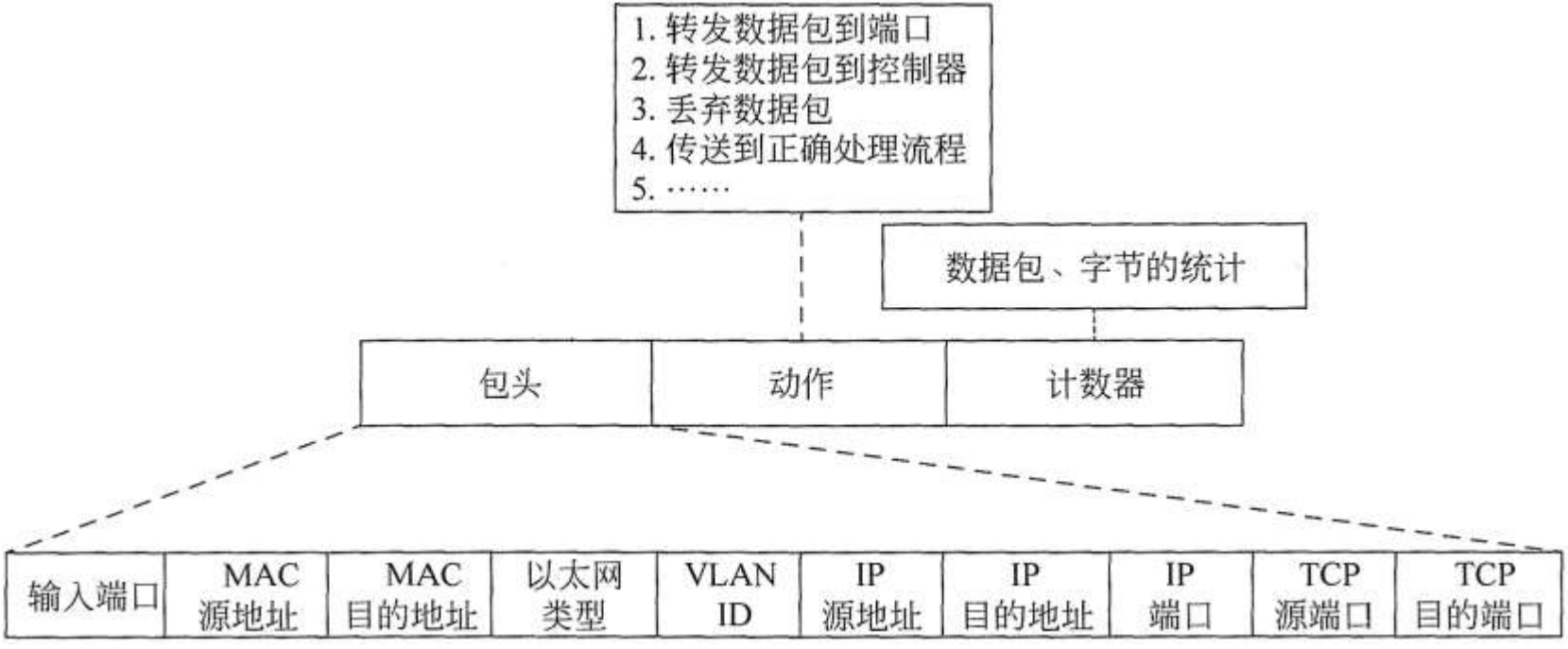
- 三个层面、两个接口
- **数据层/基础设施层**
  - 实际做数据转发的通信设备
- **控制层**
  - 控制平面
- **应用层**
  - 网络管理应用
- **南向接口**
  - 数据包过滤和转发的规则
- **北向接口**
  - 应用的控制逻辑



基础设施层

- 将控制平面与数据平面解耦
- 添加接口层接收控制层发送过来的规则
- 网络设备（根据规则）专注于数据包的转发

控制层——OpenFlow



对基础设施层的网络设备进行管理，包括拓扑管理、表项下发及策略制订

应用层

- **开发者可以基于控制器所开放出的接口设计应用**
- **应用可以访问全局网络资源**
- **应用层可以看作一个客户端**
  - 监控节点
  - 统计流量的网络可视化工具
  - 网络性能自动调优的自动化工具

参考资料

- openEuler操作系统，任炬、张尧学、彭许红编著，清华大学出版社，2020
- 深入理解Linux网络技术内幕，Benvenuti 著，夏安、闫江毓、黄景昌译，中国电力出版社，2009

谢谢！