黑客攻防

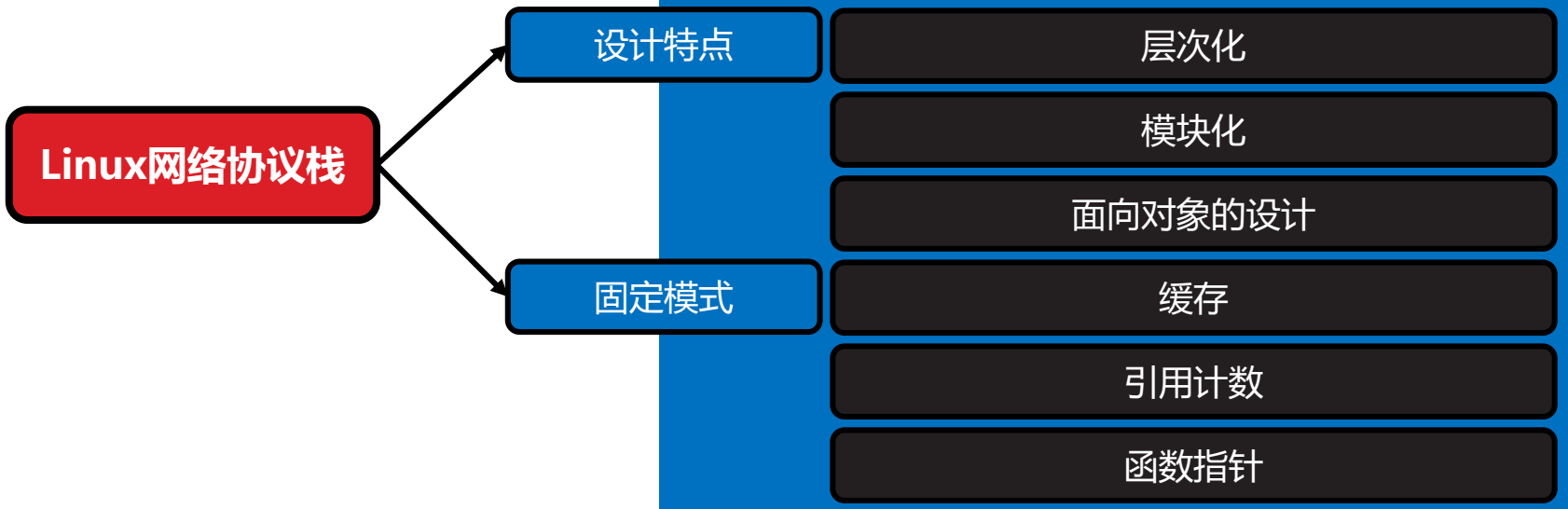
网络协议栈

Unit02

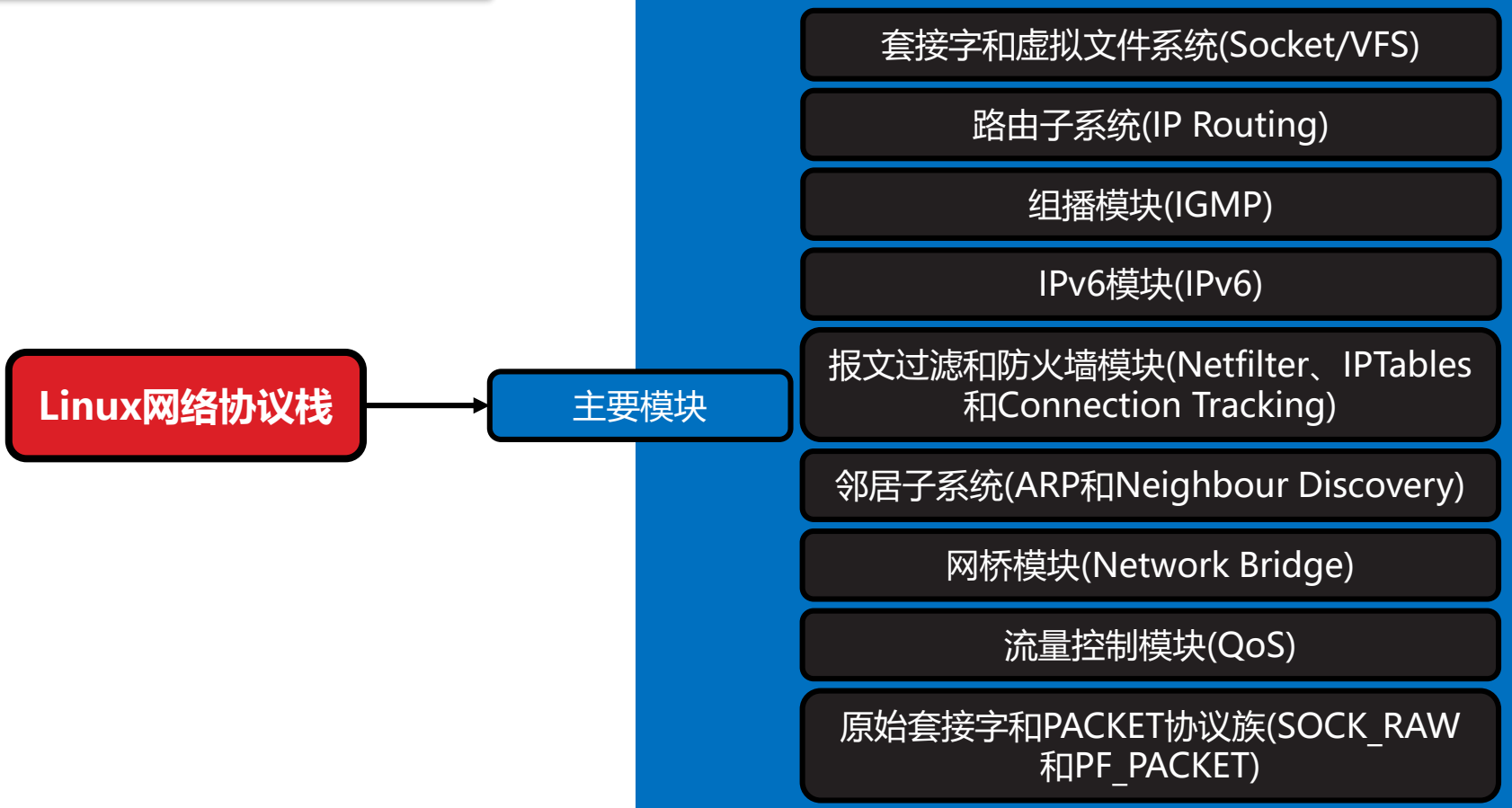
内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	Linux网络协议栈
	10:30 ~ 11:20	
	11:30 ~ 12:00	
下午	14:00 ~ 14:50	Linux报文收发流
	15:00 ~ 15:50	
	16:00 ~ 16:50	
	17:00 ~ 17:30	总结和答疑

Linux网络协议栈



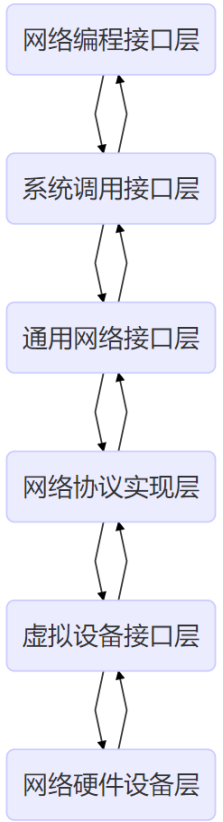
Linux网络协议栈


Linux网络协议栈
主要模块
套接字和虚拟文件系统(Socket/VFS)
路由子系统(IP Routing)
组播模块(IGMP)
IPv6模块(IPv6)
报文过滤和防火墙模块(Netfilter、IPTables和Connection Tracking)
邻居子系统(ARP和Neighbour Discovery)
网桥模块(Network Bridge)
流量控制模块(QoS)
原始套接字和PACKET协议族(SOCK_RAW和PF_PACKET)

设计特点

层次化

- 网络系统的设计采用了层次化体系结构，这样的结构为网络协议的设计与实现提供了很大的方便：
  - 上层结构通过预先定义的接口访问下层结构提供的功能
  - 任何一层实现的改变都不会影响整个协议栈的总体架构
  - 便于网络子系统与其它子系统及不同硬件设备协调工作
- Linux网络协议栈的层次结构如下图所示：
  - 网络编程接口层
    - 提供符合BSD Socket API规范的接口函数，即套接字函数族，如socket、bind、listen、accept、connect、send、recv等。应用程序通过调用这些函数实现网络通信功能
  - 系统调用接口层
    - 通过调用系统内核提供的接口，如sys_send、sys_recv等，发送和接收通信报文



知识讲解

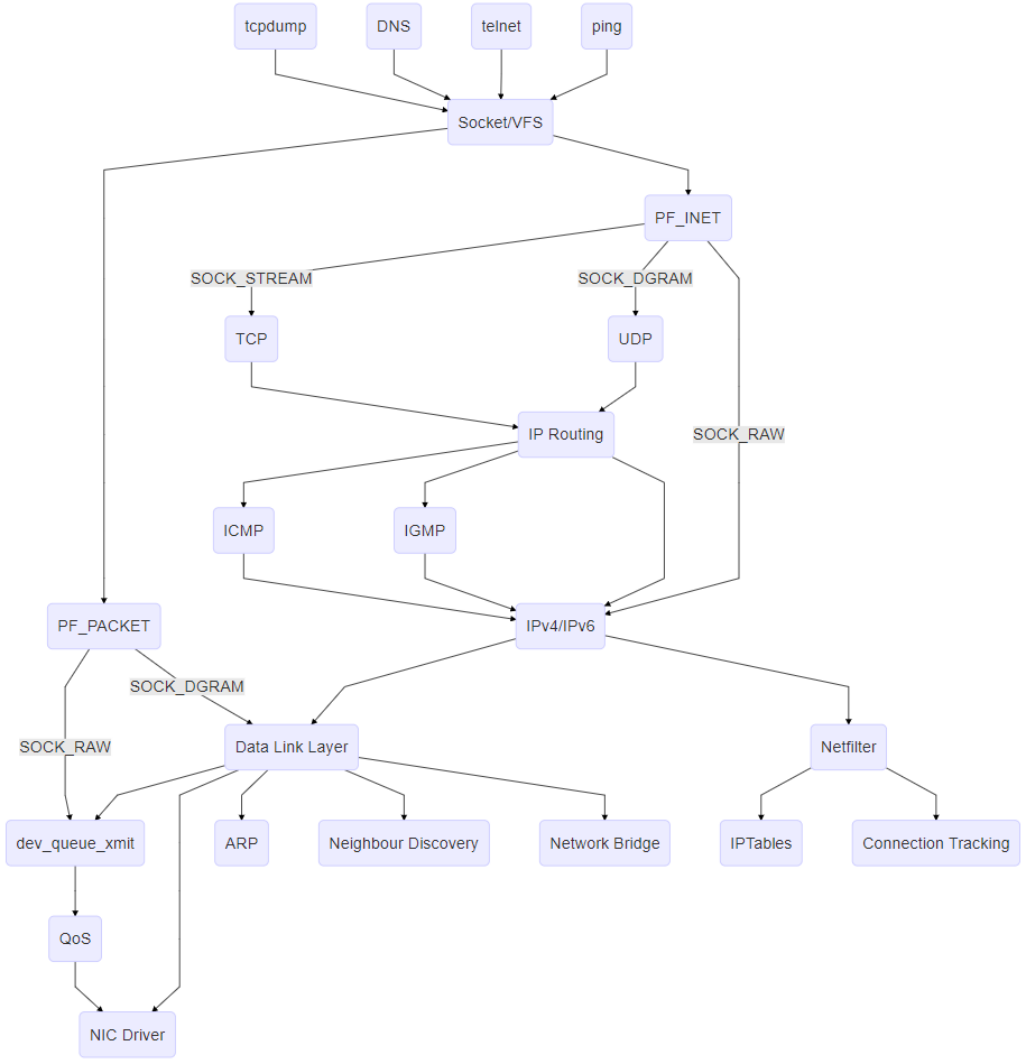
层次化

- Linux网络协议栈的层次结构如下图所示：
  - 通用网络接口层
    - 为各种不同的网络协议提供形式统一的接口封装。这样系统中的其它子系统就可以通过该层直接使用网络服务，而无需考虑不同协议之间的差别
  - 网络协议实现层
    - 各种网络协议的具体实现，如TCP、UDP、IP、ARP等
  - 虚拟设备接口层
    - 通过名为net_device的数据结构抽象地表示系统中的每个网络硬件设备，该结构中包含了所有网络硬件设备都必须支持的属性和方法，以此屏蔽底层硬件的差异性，使上层结构得以通过统一的接口访问各种不同的网络硬件设备，收发通信报文
  - 网络硬件设备层
    - 通过设备厂商提供的驱动程序操作相应的网络硬件设备

知识讲解

模块化

- Linux网络协议栈支持的协议和功能比较多，而这些协议和功能在一般情况下并不会同时使用，因此协议栈将不同的协议和功能实现为独立的模块，采用按需加载策略，根据需要选择相应模块
- Linux TCP/IP网络协议栈的主要模块如下图所示：



知识讲解

模块化

- 另外，采用模块化的实现方式，还有助于协议栈的扩展，针对新增的协议和功能，只需添加相应的模块即可，对协议栈的整体结构影响甚小
- 每个模块的设计目标都很明确，一个模块只用来完成一项任务。为了降低耦合度，模块与模块之间多通过函数指针相互调用。这样当一个模块的实现发生变化时，不会影响其它使用该模块的模块

知识讲解

面向对象的设计

- 在Linux网络协议栈的实现中，多处采用了面向对象的程序设计理念：
  - 邻居表
    - ARP协议是获取网络节点IP地址和MAC地址映射关系的协议。但在其它通信协议中还存在另外几种不同的地址映射关系。Linux网络协议栈对此进行了适度的抽象，采用“邻居”的概念表示相邻计算机，不同协议可通过相同接口管理“邻居”
  - 套接字
    - Linux网络协议栈支持多达20余种通信协议，但它仅向用户暴露一套统一的应用编程接口，即伯克利套接字接口(BSD Socket API)。这样做最大的优点是，用户只需要调用一套函数就可以实现基于不同协议的数据通信
  - 多态性
    - Linux网络协议栈的多态性是通过函数指针实现的。同样的一个调用语句，实际被执行的函数会因调用者所持有函数指针的不同指向而异。当需要切换不同协议时，只需更新函数指针的值即可，调用代码无需做任何修改。这种设计极大地提高了程序的可扩展性和灵活性

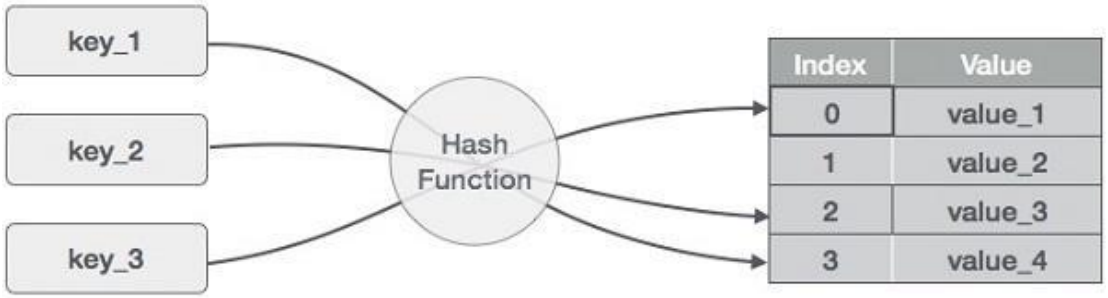
知识讲解

固定模式

缓存

- Linux网络协议栈为了提高对数据的处理效率而大量地使用缓存，如缓存路由结果的rtable结构和ARP缓存等
- Linux网络协议栈中的缓存多采用哈希表的形式。内核中已经包含了构建哈希表的基本数据结构，如数组、单向链表、双向链表等，至于具体的哈希函数则因不同的缓存对象而异。特殊情况下，还会在哈希表的键中增加一些随机特征，用于防止针对哈希表的拒绝服务攻击

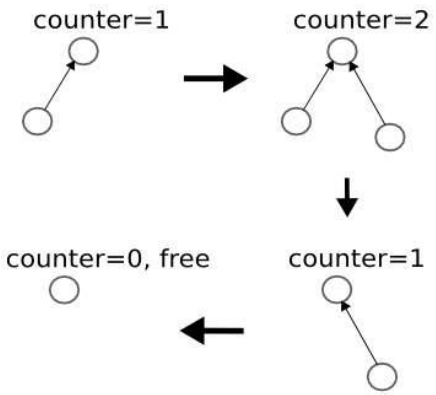
知识讲解



引用计数

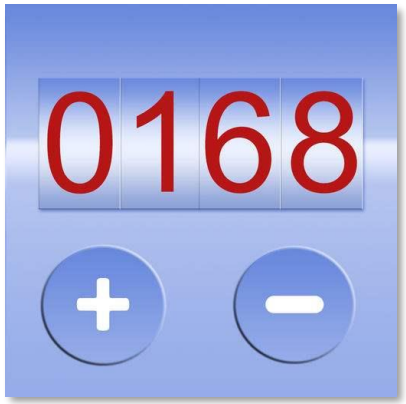
- 操作系统内核中的数据结构经常会被多个进程所共享，这就涉及如何进行垃圾回收的问题，即只有不被任何进程使用的数据结构才能被释放，否则就会引发诸如非法指针访问等严重错误
- Linux网络协议栈中的许多数据结构都带有一个引用计数字段，使用该结构的用户在使用之前先增加引用计数的值，使用之后再减少其值，当其值为0时，表示该结构已不再被任何用户使用，其所占内存可被释放

知识讲解



引用计数

- 使用带有引用计数的数据结构时需要注意：
  - 使用后忘记减少引用计数，结构所占内存可能永远得不到释放，形成内存泄漏
  - 使用前忘记增加引用计数，结构所占内存可能被提前释放，导致非法指针访问
  - 于表结构中搜索特定元素，搜索函数通常会自动增加该元素的引用计数，这时：
    - 既不要重复增加其引用计数
    - 也不要忘记减少其引用计数
- Linux内核源码中用于增加引用计数的函数通常名为xxx_hold，而用于减少引用计数的函数则名为xxx_release或者xxx_put



知识讲解

函数指针

- 通过在结构体中定义函数指针字段，并将该指针初始化为不同的函数地址，以执行不同的函数代码，可以使一种静态形式的调用语句，表现出某种动态形式的行为特征，即所谓多态性
- 在Linux网络协议栈中使用函数指针的三种情况：
  - 作为层间接口实现一对多的映射关系
  - 根据当前状态或其它模块的处理结果选择具体要执行的函数
  - 实现一些可选的自定义处理
- 在Linux源码中遇到函数指针，务必搞清楚其具体指向哪个函数。例如：
  - 处理接收报文的过程中，下层模块根据包头中的上层协议类型字段选择适当的上层函数，初始化函数指针
  - ARP模块根据缓存状态让函数指针指向正确的处理函数

知识讲解

主要模块

套接字和虚拟文件系统(Socket/VFS)

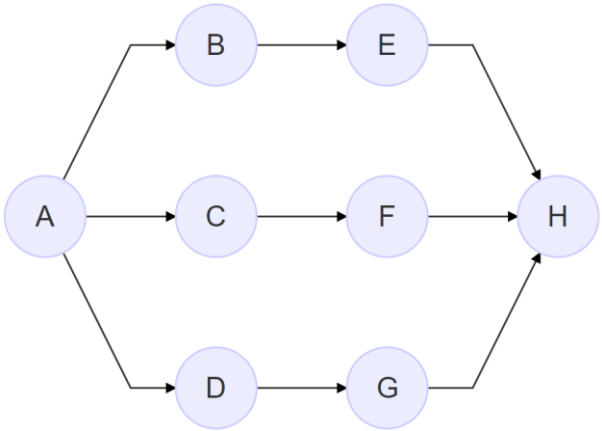
- 套接字是对BSD Socket接口规范的实现，旨在为应用程序的开发者提供一套抽象而统一的标准接口，以用于网络编程。虚拟文件系统则实现了针对套接字的读写操作，让用户可以象访问普通文件一样收发网络上的数据
- 套接字支持PF_INET和PF_PACKET两种协议族。其中PF_INET表示互联网上使用的TCP/IP协议族，该协议族提供以下三种服务类型：
  - SOCK_STREAM：实现基于TCP协议的网络通信
  - SOCK_DGRAM：实现基于UDP协议的网络通信
  - SOCK_RAW：实现基于IP协议的网络通信。用户在自行构造IP层之上的各层封装之后，直接将数据交由IP层打包发送，接收时再逆向解除封装，获得数据内容

知识讲解

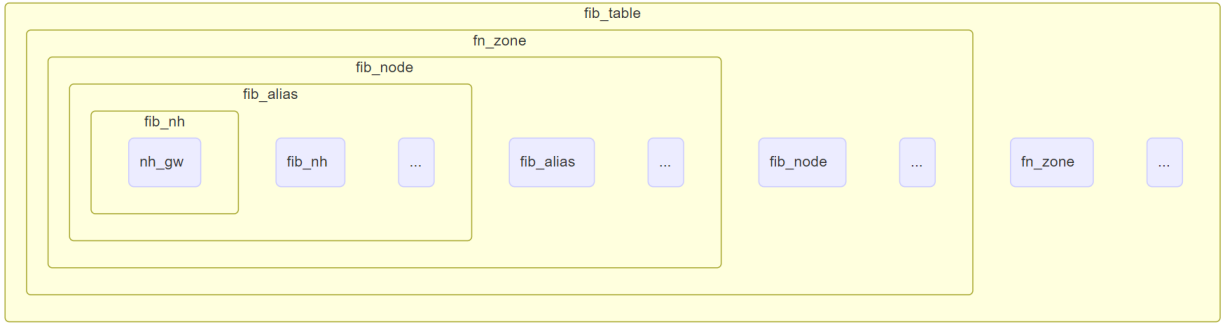
路由子系统(IP Routing)

知识讲解

- 当主机向外发送IP分组时，需要根据分组的目的地址查询路由，以确定下一跳的地址。如下图所示：



- Linux网络协议栈的路由信息库包括路由规则和路由表集两部分。针对满足特定条件的报文，依据路由规则，从相应的路由表中获取路由信息
- 每张路由表由多条路由表项组成，每条路由表项中均包含了目的网络和下一跳的地址。一个目的网络可能存在多个下一跳地址，即多路径路由
- 路由表采用层次化结构保存路由信息，如下图所示：



路由子系统(IP Routing)

- Linux网络协议栈将与IP路由有关的信息存放在转发信息块(Forwarding Information Block, FIB)中：
  - 在转发信息块中，用fib_table结构表示一张路由表
  - 根据子网掩码的长度[0,32]，将目的网络划分成33个区，每个区用一个fn_zone结构表示
  - 在每一个区中根据目的网络的不同，划分成若干节点，每个节点用一个fib_node结构表示
  - 在每一个节点中根据服务类型(Type of Service, ToS)的不同，划分成若干别名，每个别名用一个fib_alias结构表示
  - 每个别名均包含若干下一跳(Next Hop)地址，每个地址均保存在一个fib_nh结构的nh_gw字段中

知识讲解

路由子系统(IP Routing)

- 路由子系统对外暴露的接口是名为fib_lookup的函数，它会根据是否启用路由策略等条件找到合适的路由表，并在该表中检索与给定目的网络相对应的路由表项
- Linux网络协议栈的转发信息块是一个复杂的数据结构，从中检索需要的路由信息会比较缓慢，为此Linux网络协议栈针对经常使用的路由信息实现了缓存，以加快路由检索的速度
- 发送分组时，通过路由子系统确定下一跳的地址，收到一个分组，同样通过路由子系统确定是接收还是转发该分组

知识讲解

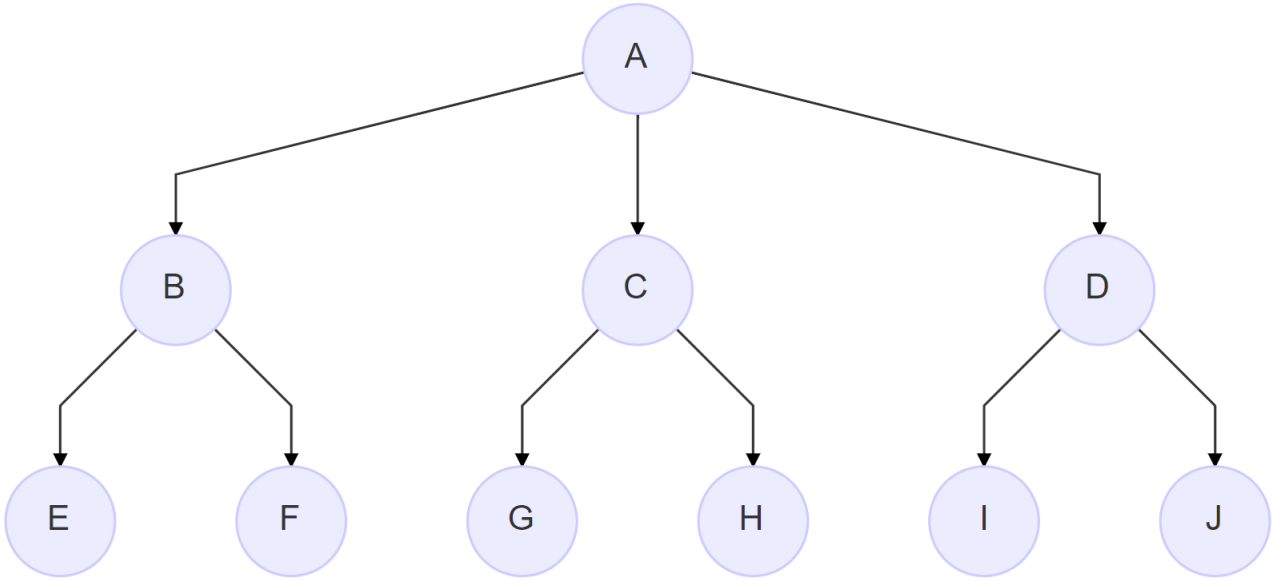
组播模块(IGMP)

- 互联网上的大多数网络应用都属于单播通信，即将分组从一台主机发送到另一台主机。但有些应用，如视频会议等，要求同时存在多台发送主机和多台接收主机，这样的通信模式称为组播或多播
- 为了支持组播通信，Linux网络协议栈实现了以下两种功能：
  - 管理参与组播的成员：将IGMP协议数据作为IP分组的负载，并在IP包头中标明负载的类型
  - 向组播成员分发数据：借助DVMRP、MOSPF等组播路由算法，将组播成员组织成树状结构
    - 树根节点主机负责发送
    - 树叶节点主机负责接收
    - 中间节点主机负责转发

知识讲解

组播模块(IGMP)

- 组播树的逻辑结构如下图所示：



- 组播路由算法以守护进程的形式实现于用户空间，内核协议栈负责对组播报文的转发和接收

知识讲解

IPv6模块(IPv6)

- IPv6将IPv4的32位IP地址扩展至128位，同时在包头格式和包头扩展选项等方面有别于IPv4
- 分组仅在以下三种情况下会进入IPv6模块：
  - 数据链路层包头显示网络层使用IPv6协议
  - 应用层调用了传输层的IPv6函数
  - 通过IPv6发送ICMP报文

知识讲解

报文过滤和防火墙模块(Netfilter、IPTables和Connection Tracking)

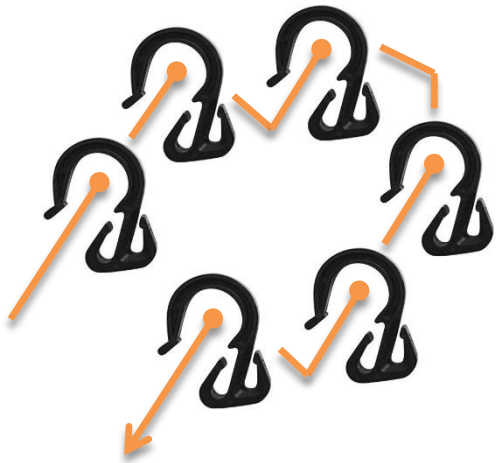
- 从Linux2.4内核开始，内核的设计者在网络协议栈中预留了若干钩子函数，即通过函数指针调用开发人员自己定义的函数，以实现诸如报文过滤、报文处理、 网络地址转换(Network Address Translation, NAT)等功能
- 这套钩子函数即构成了Netfilter框架，具体包括以下三个部分：
  - 为每种通信协议，如IPv4、IPv6等，均定义了一套钩子函数。这些钩子函数在报文流经网络协议栈的若干挂接点时被调用。当报文流到某个特定的挂接点时，网络协议栈会将该报文连同特定的钩子标识一起传递给Netfilter框架，框架根据钩子标识找到对应的钩子函数指针，以报文为参数调用该钩子函数，从钩子函数返回后再继续挂接点之后的流程

知识讲解

报文过滤和防火墙模块(Netfilter、IPTables和Connection Tracking)

- 这套钩子函数即构成了Netfilter框架，具体包括以下三个部分：
  - 程序员可将自己定义的函数挂接到Netfilter框架，即将该函数的指针与特定协议的特定钩子标识建立关联。当框架发现与该钩子标识相关联的函数指针非空时，即可通过该指针调用其指向的函数。程序员可在该函数中检查报文以决定是否丢弃，或将报文传递至用户空间队列等待后续处理
  - 用户进程可以异步的方式处理传递至用户空间的报文，并在处理完成以后令内核从相应的钩子函数中返回，继续执行网络协议栈的后续处理流程

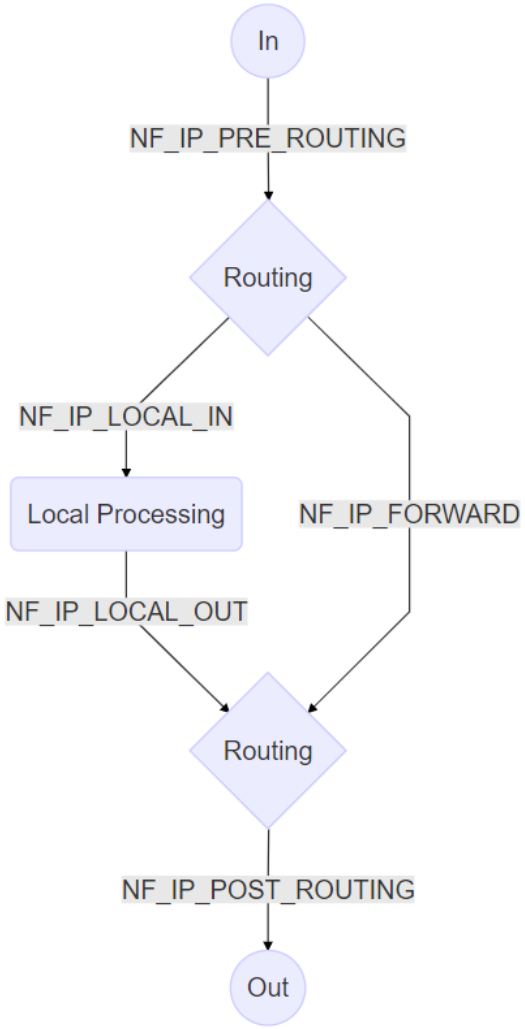
知识讲解



报文过滤和防火墙模块(Netfilter、IPTables和Connection Tracking)

知识讲解

- IPv4协议共有5个可挂接钩子函数的挂接点，如下图所示：
  - 主机收到报文并完成IP校验后，会经过第一个挂接点NF_IP_PRE_ROUTING，然后进入路由，以决定是本机处理还是转发
    - 若本机处理，则经过NF_IP_LOCAL_IN挂接点进入上层协议栈
    - 如果是转发，则经过NF_IP_FORWARD挂接点直接进入路由
  - 本机产生的报文，经过NF_IP_LOCAL_OUT挂接点进入路由
  - 所有需要发出的报文，都要经过NF_IP_POST_ROUTING挂接点才能传到网上



报文过滤和防火墙模块(Netfilter、IPTables和Connection Tracking)

- IPTables是一个基于Netfilter框架的报文过滤和修改工具。该模块可以创建若干张规则表，并在报文流经特定的规则表时，对其进行过滤或修改。IPTables预定义了三张规则表：
  - 报文处理(Packet Mangling)规则表：通过全部五个挂接点接入Netfilter框架，实现对报文的修改或附加带外数据
  - 网络地址转换(NAT)规则表：通过NF_IP_PRE_ROUTING、NF_IP_POST_ROUTING、NF_IP_LOCAL_IN和NF_IP_LOCAL_OUT四个挂接点接入Netfilter框架，分别对转发报文源地址、目的地址和本机报文源地址、目的地址进行转换
  - 报文过滤(Packet Filtering)规则表：通过NF_IP_LOCAL_IN、NF_IP_LOCAL_OUT和NF_IP_FORWARD三个挂接点接入Netfilter框架，分别对接收、发送和转发报文进行过滤

知识讲解

报文过滤和防火墙模块(Netfilter、IPTables和Connection Tracking)

- 每一张IPTables规则表都会在Netfilter框架的若干挂接点上挂接钩子函数，当报文到达这些挂接点时，就会进入相应的钩子函数进行处理，而处理过程就是依据系统管理员事先制定的规则，逐一匹配，若满足其中某条规则的约束条件则执行相应的处理，不满足则继续检查下一条规则。若该报文对所有规则约束都不满足，则按缺省策略处理。系统管理员可以通过iptables命令管理防火墙并为其设置规则

知识讲解

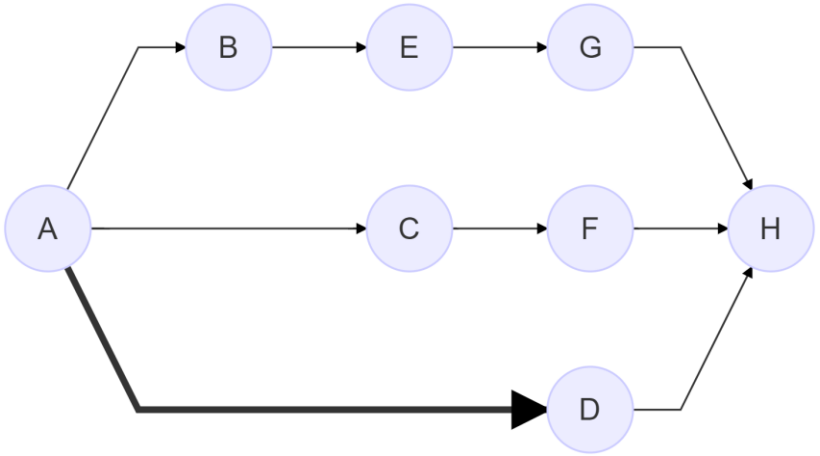
报文过滤和防火墙模块(Netfilter、IPTables和Connection Tracking)

知识讲解

- 连接跟踪(Connection Tracking)模块通过NF_IP_PRE_ROUTING和NF_IP_LOCAL_OUT两个挂接点接入Netfilter框架，且比相同挂接点的其它钩子函数享有更高优先级。这两个挂接点分别是接收和发送报文流经的第一个挂接点
- 连接跟踪模块能够识别出每个报文所属的连接，并为每个连接建立状态表，利用该表跟踪连接的状态。防火墙不仅根据规则表过滤和处理报文，还会考虑报文是否符合连接所处的状态，所谓状态检测

邻居子系统(ARP和Neighbour Discovery)

- 在通过网络发送分组的过程中，一台主机需要将分组转发给距离目标主机更近的相邻主机，并且需要知道相邻主机第三层地址(即网络层地址，如IP地址)和第二层地址(即数据链路层地址，如MAC地址)间的映射关系。如下图所示：

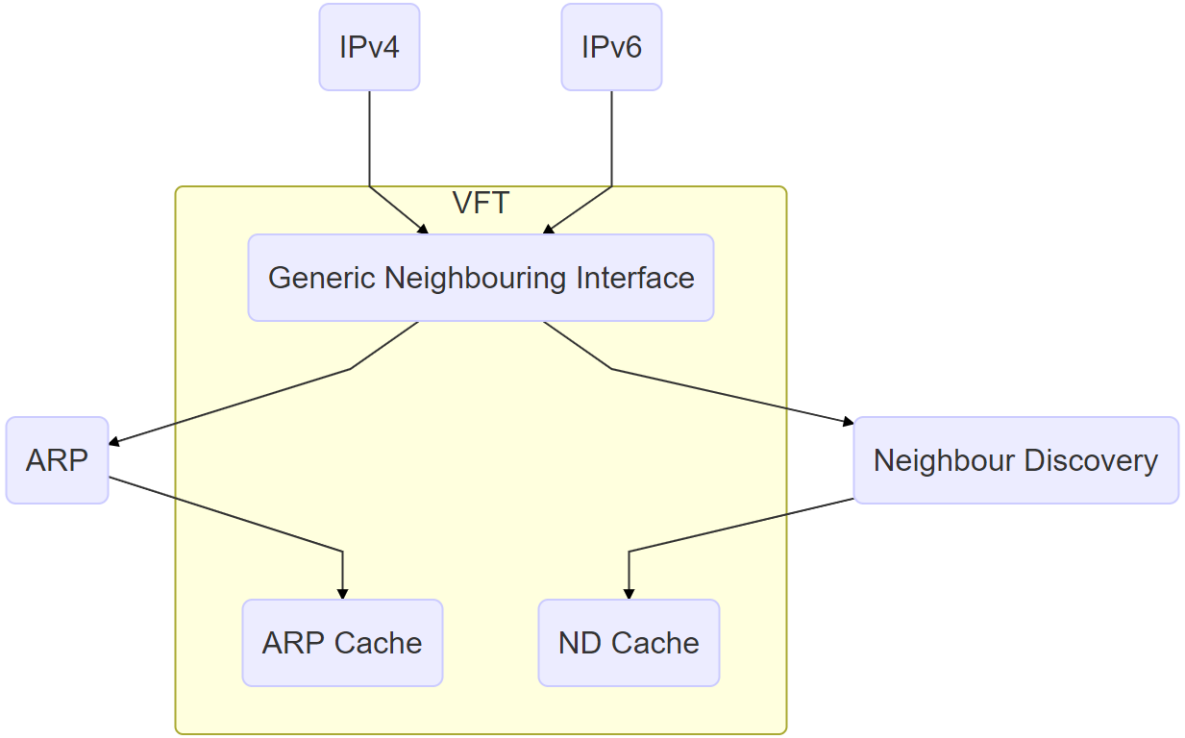


- 在TCP/IP协议族中完成上述功能的是ARP协议和邻居发现(Neighbour Discovery, ND)协议，前者对应IPv4，而后者对应IPv6

知识讲解

邻居子系统(ARP和Neighbour Discovery)

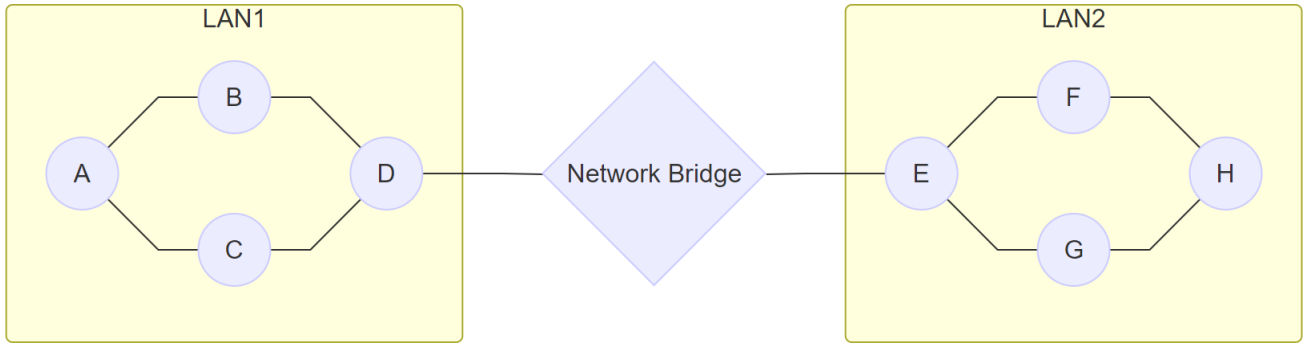
- 虽然不同的网络协议会通过不同的方法实现地址映射，但它们的目的都是相同的，因此Linux网络协议栈的邻居子系统将这些相对通用的部分抽象为一个协议无关的服务接口——通用邻居接口(Generic Neighbouring Interface, VFT)，而具体的实现细节则被隐藏在该接口之下。此外，通用邻居接口还提供带有老化超时机制的地址映射缓存，如下图所示：



知识讲解

网桥模块(Network Bridge)

- 网桥在数据链路层上将两个或多个物理上独立的局域网连接成为一个网段，如下图所示：



- 物理独立的局域网LAN1和LAN2通过网桥连接后，网桥收到来自LAN1的数据链路包，检查其目的地址，如果该包是发往LAN1中某台主机的，则将其滤除，而如果该包是发往LAN2中某台主机的，则将其转发到LAN2。由此可见，网桥的主要功能就是根据数据链路层地址，如MAC地址，处理针对数据链路包的过滤和转发

知识讲解

网桥模块(Network Bridge)

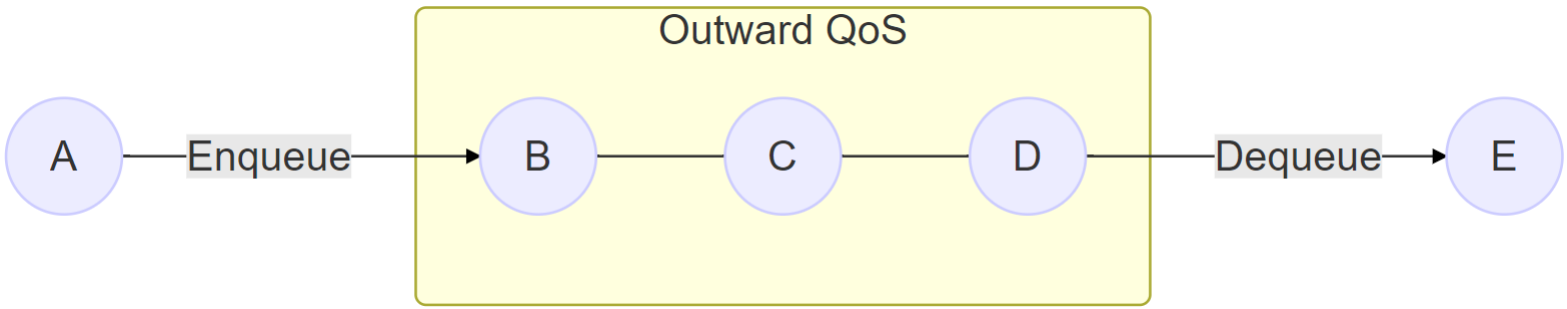
- 网桥还有自动学习能力，可在转发数据链路包的过程中智能地构建转发表，同时借助生成树协议(Spanning Tree Protocol, STP)消除网络中可能出现的环路
- 网桥模块通过EBTables (Ethernet Bridge Tables)框架允许在数据链路包的转发路径上挂接钩子函数，以获得处理被网桥转发的数据链路包的机会，如替换目的MAC地址等
- 通过IPTables和EBTables的结合，可以构建网桥式防火墙，从外部看是一个网桥，但却具备防火墙的功能，可以对流经网桥的内容进行过滤和监控

知识讲解

流量控制模块(QoS)

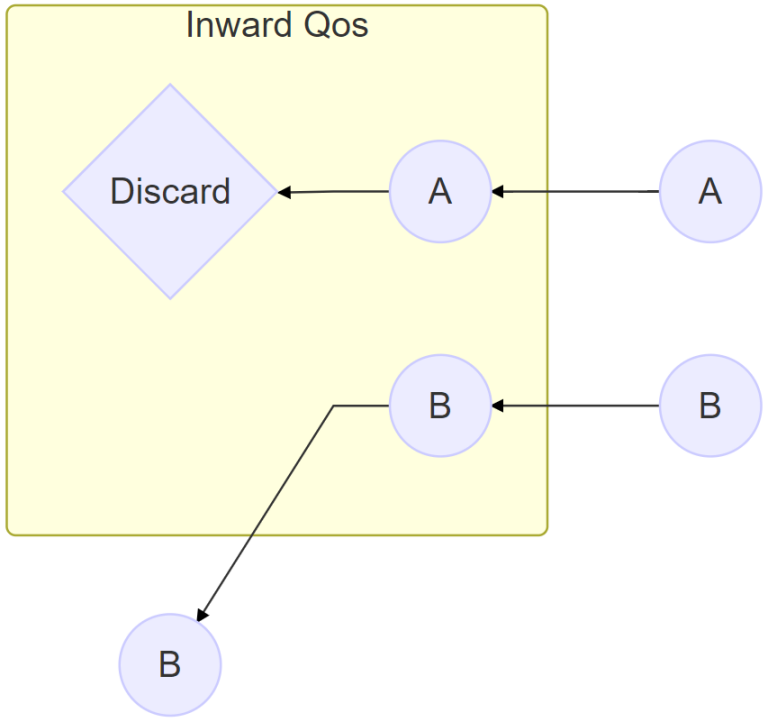
- 目前大多数路由器和交换机都支持通过服务质量(Quality of Service, QoS)管理进行流量控制，如保证某些应用获得足够的带宽，同时限制另一些应用所能占用的带宽
- Linux网络协议栈既能控制向外的流量，也能控制向内的流量：
  - 对于向外的流量，可以采用很多不同的队列调度算法已达到各种不同的控制目的，超出流量限制的报文，可以延迟发送也可以直接丢弃：

知识讲解



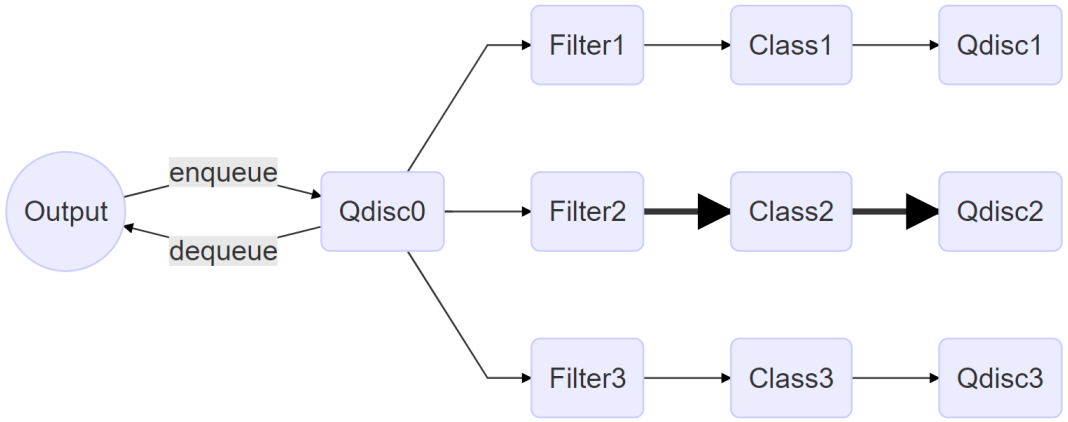
流量控制模块(QoS)

- Linux网络协议栈既能控制向外的流量，也能控制向内的流量：
  - 对于向内的流量，则只是根据预先设定的策略选择丢弃或者接收：



流量控制模块(QoS)

- 流量控制包括以下三部分内容：
  - 排队规则(Queueing Discipline, Qdisc)：发送报文被压入和弹出输出队列的规则，如FIFO规则等
  - 类别(Class)：某些排队规则支持分类细化，每个子类又有自己的排队规则，最终形成一棵规则树
  - 过滤器(Filter)：对报文进行分类，以确定它们所隶属的类别，进而确定它们按何种规则进出队列
- 它们的关系如下图所示：



知识讲解

流量控制模块(QoS)

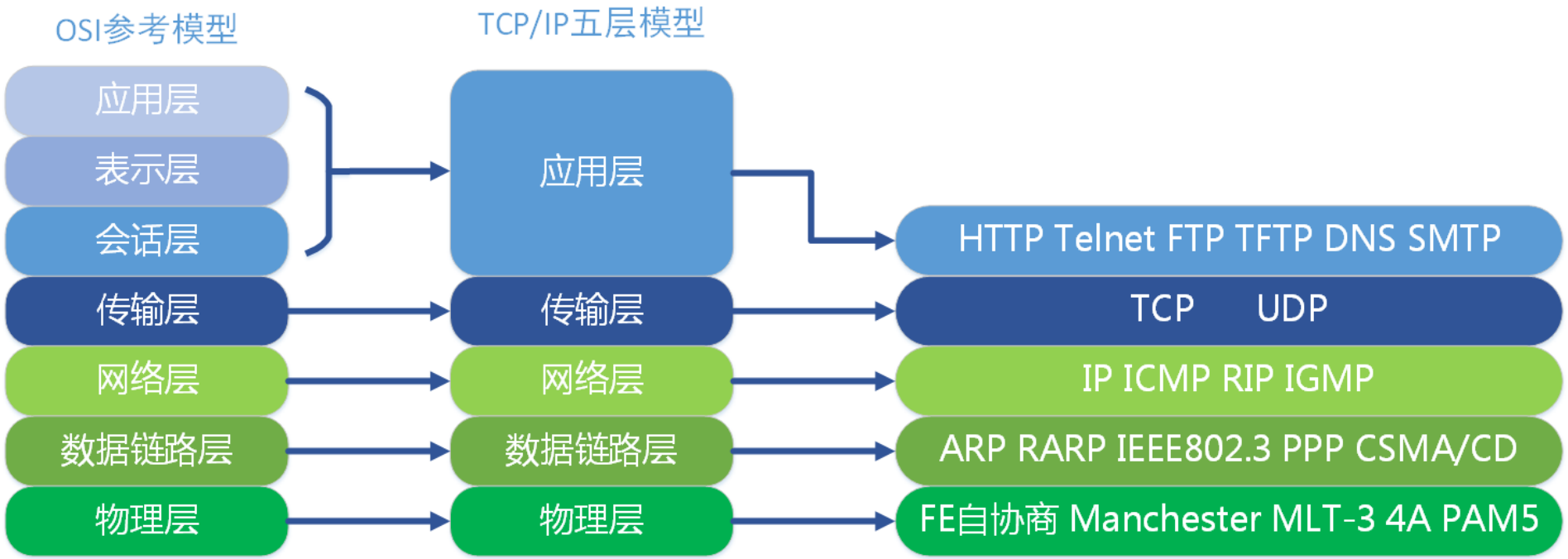
- 例如：
  - 内核调用enqueue函数将输出报文交给一个支持分类的排队规则Qdisc0
  - 排队规则Qdisc0通过一系列过滤器Filter1、Filter2和Filter3对报文实施过滤，最终确定其隶属于Class2
  - 将该报文加入到与其类别Class2相对应的子排队规则Qdisc2中
- 注意：向内流量的enqueue函数实际上并没有将报文加入到任何队列中，因此也不需要为其实现dequeue函数
- 在启用QoS功能后，用户可以通过tc命令配置流量控制模块

知识讲解

原始套接字和PACKET协议族(SOCK_RAW和PF_PACKET)

- 开放式系统互联(Open Systems Interconnection, OSI)标准的七层网络模型与TCP/IP协议的五层网络模型存在如下图所示的对应关系，其中的各个层次分别实现了不同的网络协议：

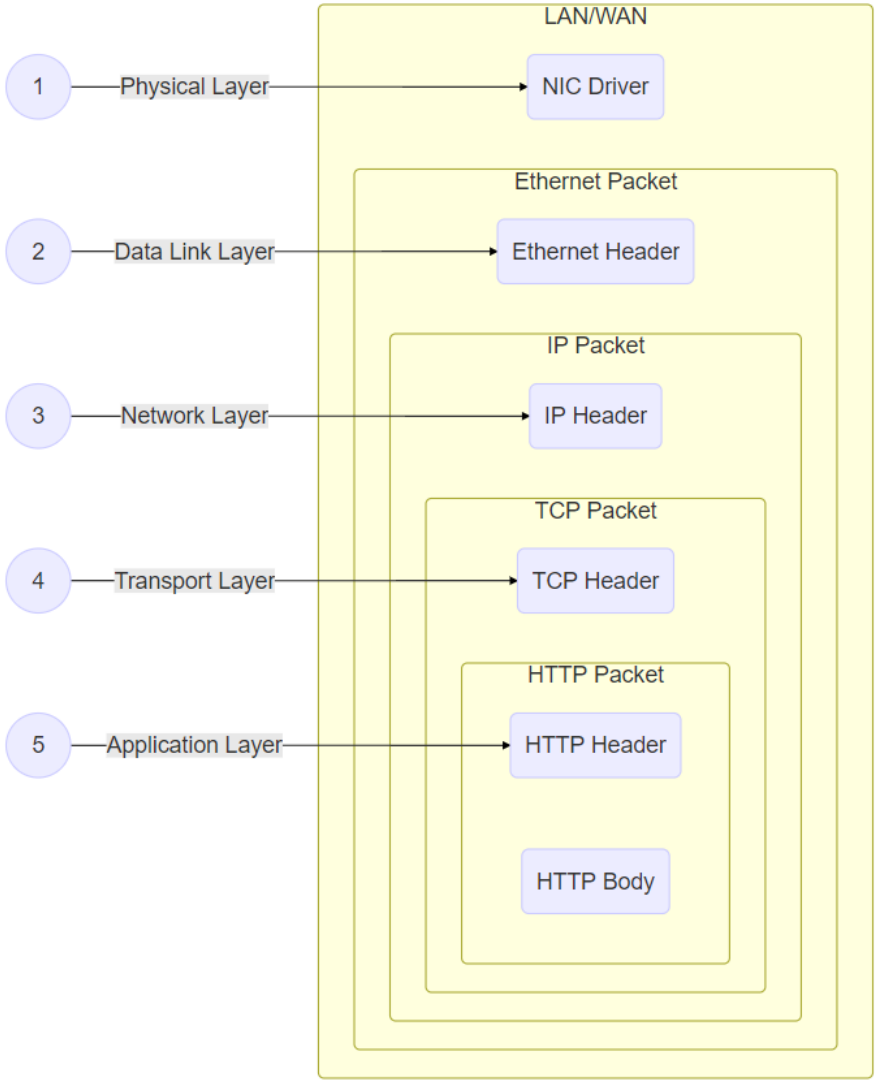
知识讲解



原始套接字和PACKET协议族(SOCK_RAW和PF_PACKET)

知识讲解

- 网络模型中的每一层在发送报文的过程中，以上一层组织的数据包作为负载，添加本层协议包头，形成本层数据包，再传递给下一层。相反在接收报文的过程中，以下一层解析出的数据包为基础，分析并移除本层协议包头，将数据包的负载部分传递给上一层。如下图所示：

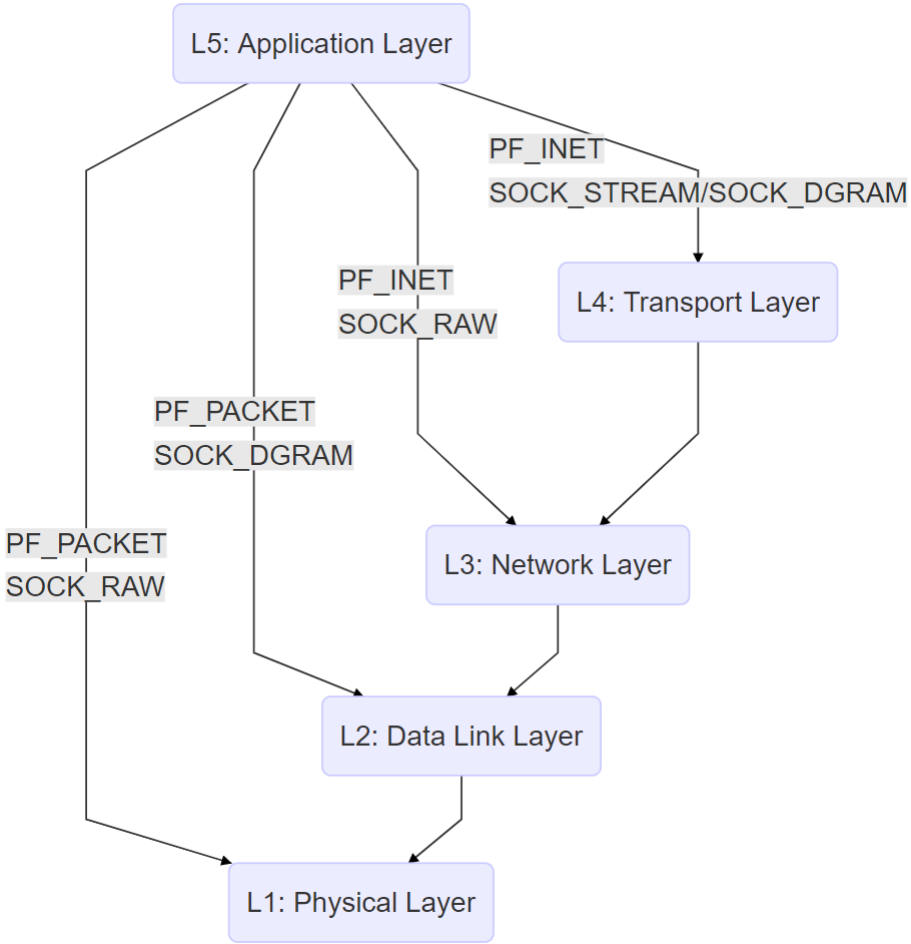


原始套接字和PACKET协议族(SOCK_RAW和PF_PACKET)

- Linux网络协议栈除了支持基于TCP和UDP协议的报文收发外，还支持以下三种特殊的报文收发方式：
  - INET协议族(PF_INET)的原始套接字(SOCK_RAW)：允许应用层跨越传输层直接与网络层相连。用户程序自己组织和解析IP包的负载部分，内核只负责针对IP包头的处理。如果使用IP_HDRINCL选项，用户程序甚至可以自己构建IP包头
  - PACKET协议族(PF_PACKET)的数据报套接字(SOCK_DGRAM)：允许应用层跨越传输层和网络层直接与数据链路层相连。用户程序自己组织和解析以太网包的负载部分，内核只负责针对以太网包头的处理
  - PACKET协议族(PF_PACKET)的原始套接字(SOCK_RAW)：允许应用层跨越传输层、网络层和数据链路层直接与物理层的网络设备驱动交互。用户程序自己组织和解析包括数据链路层在内的各层协议包头。利用这种方法，应用程序甚至可以在用户态实现一个自己的网络协议栈

原始套接字和PACKET协议族(SOCK_RAW和PF_PACKET)

- 协议族和套接字类型与网络协议栈的关系如下图所示：



知识讲解

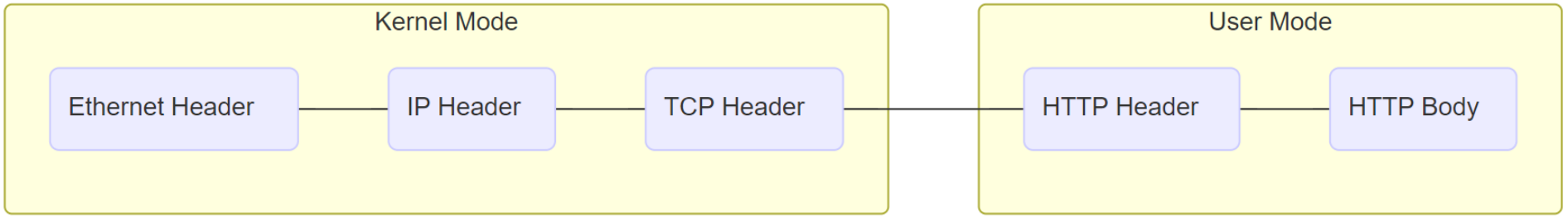
原始套接字和PACKET协议族(SOCK_RAW和PF_PACKET)

- 通过原始套接字和PACKET协议族，用户程序可以自行处理某些协议的包头。二者之间的区别在于用户程序处理包头的能力有所不同：
  - INET协议族(PF_INET)的原始套接字(SOCK_RAW)：用户程序能够处理传输层及以上各层协议的包头，配合IP_HDRINCL选项也可以构建IP包头，但是网络层只能使用IP协议。接收时，用户程序只能获得IP包的负载部分而无法获得IP包头
  - PACKET协议族(PF_PACKET)的数据报套接字(SOCK_DGRAM)：用户程序能够处理网络层及以上各层协议的包头。接收时，用户程序可以获得以太网包的负载部分
  - PACKET协议族(PF_PACKET)的原始套接字(SOCK_RAW)：用户程序能够处理包括数据链路层在内的所有各层协议的包头，因此适用于基于各种网络协议的通信。接收时，用户程序甚至可以获得以太网包的包头

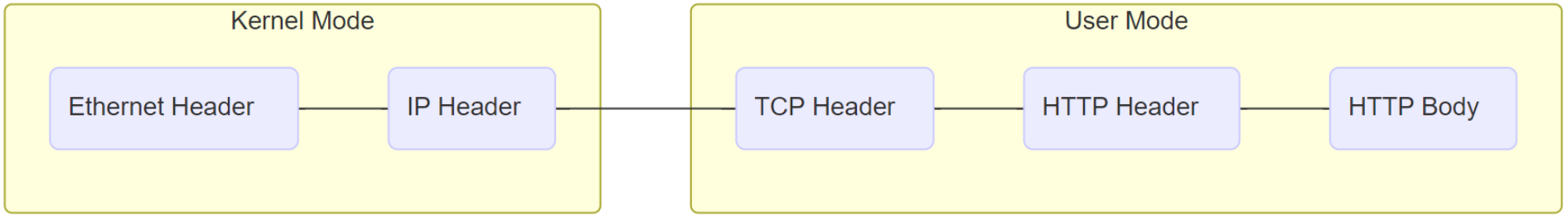
知识讲解

原始套接字和PACKET协议族(SOCK_RAW和PF_PACKET)

- 以处理HTTP报文为例，用户态与内核态的分工因协议族和套接字类型的不同而异：
  - INET协议族(PF_INET)的流式套接字(SOCK_STREAM)：



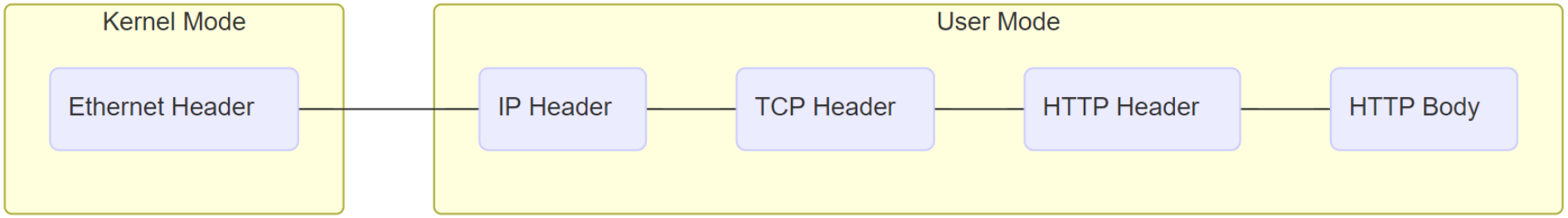
  - INET协议族(PF_INET)的原始套接字(SOCK_RAW)：



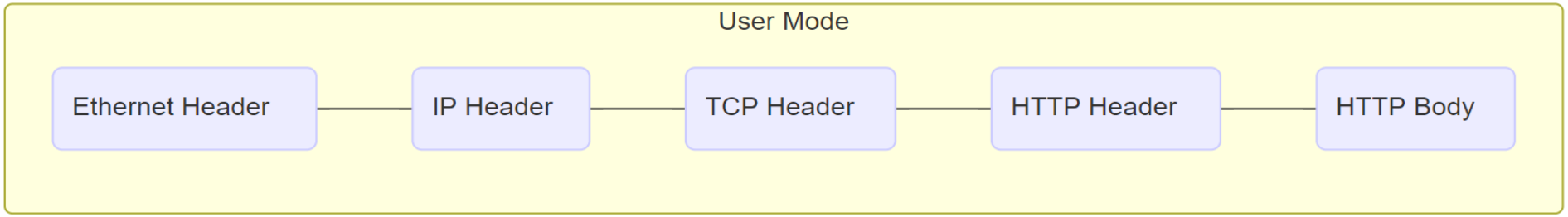
知识讲解

原始套接字和PACKET协议族(SOCK_RAW和PF_PACKET)

- 以处理HTTP报文为例，用户态与内核态的分工因协议族和套接字类型的不同而异：
  - PACKET协议族(PF_PACKET)的数据报套接字(SOCK_DGRAM)：



  - PACKET协议族(PF_PACKET)的原始套接字(SOCK_RAW)：

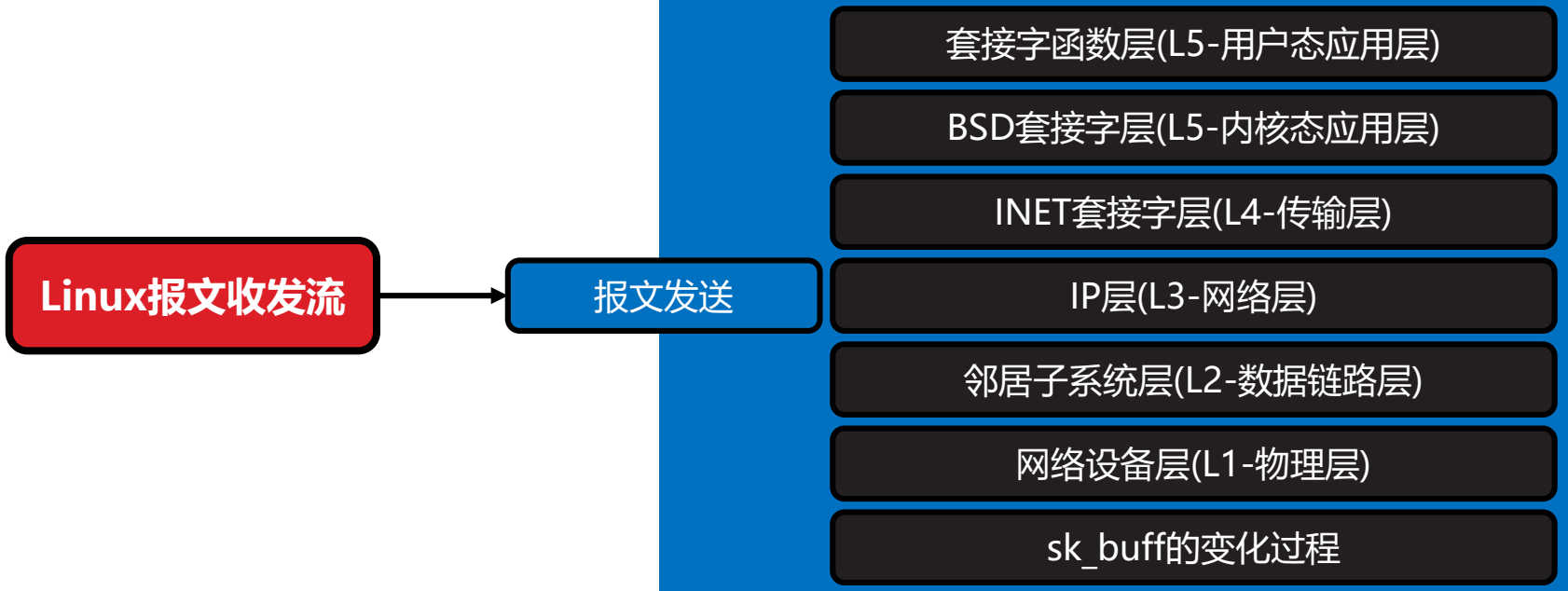


知识讲解

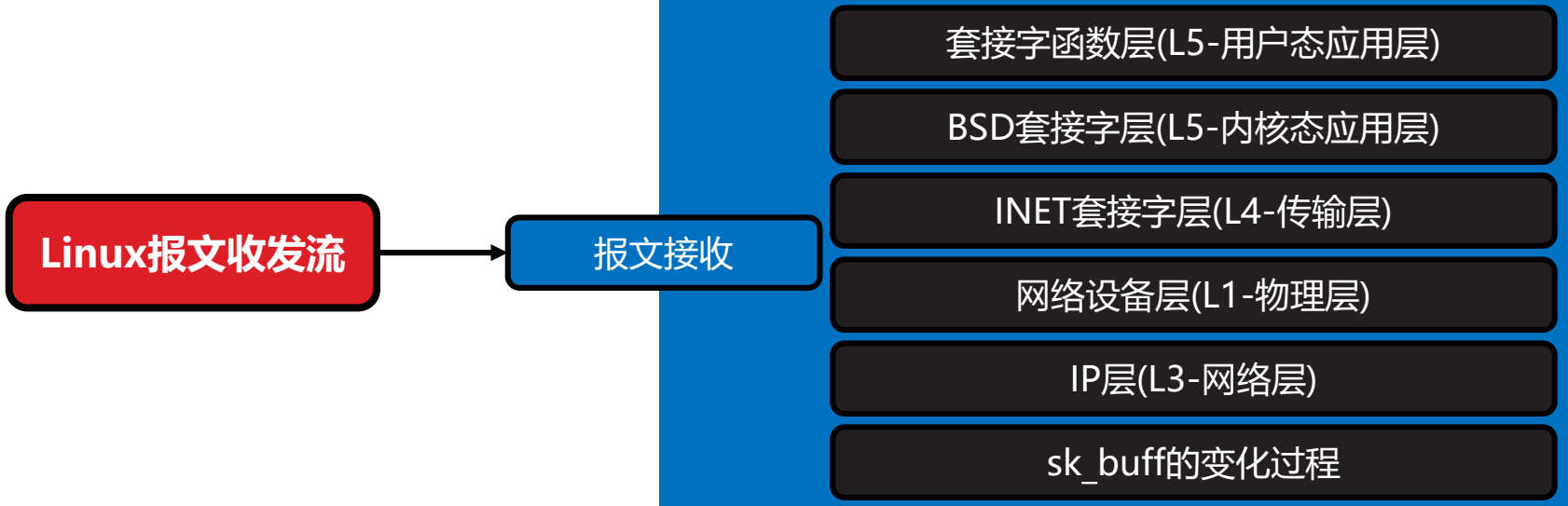
Linux报文收发流


Linux报文收发流
报文表示
批量传输
sk_buff中的数据布局字段
sk_buff中的链表队列字段
sk_buff中的协议通用字段
sk_buff中的特定功能字段
sk_buff中的管理函数字段
批量传输

Linux报文收发流


Linux报文收发流
报文发送
套接字函数层(L5-用户态应用层)
BSD套接字层(L5-内核态应用层)
INET套接字层(L4-传输层)
IP层(L3-网络层)
邻居子系统层(L2-数据链路层)
网络设备层(L1-物理层)
sk_buff的变化过程

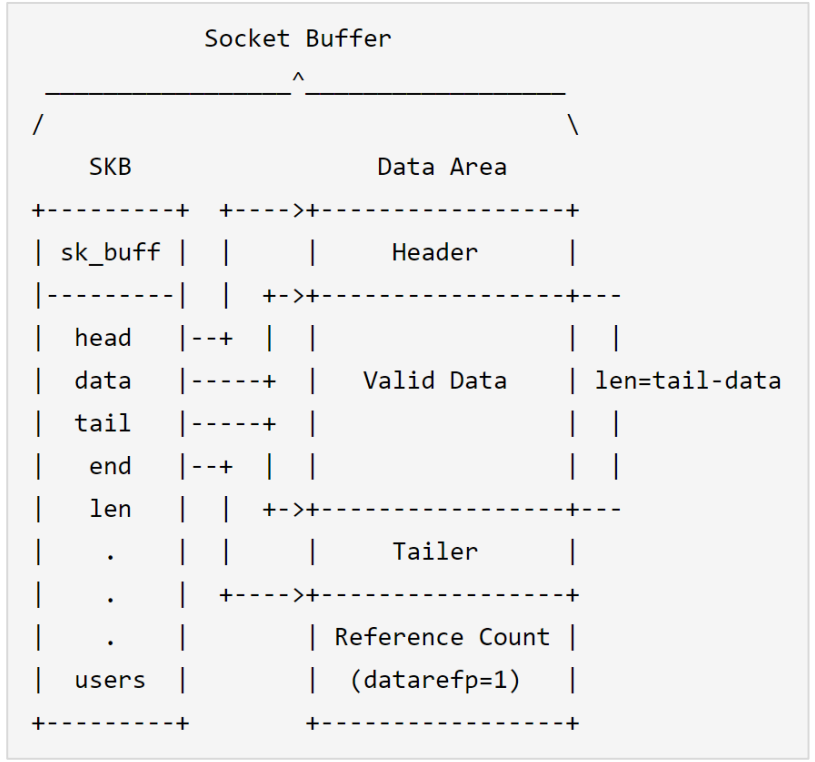
Linux报文收发流


Linux报文收发流
报文接收
套接字函数层(L5-用户态应用层)
BSD套接字层(L5-内核态应用层)
INET套接字层(L4-传输层)
网络设备层(L1-物理层)
IP层(L3-网络层)
sk_buff的变化过程

报文表示

报文表示

- 内核用套接字缓存(Socket Buffer)的概念作为对各种协议报文的抽象。一个套接字缓存，即一个被处理中的报文。每个套接字缓存均由管理区(sk_buff, SKB)和数据区(Data Area)两部分组成，如下图所示：
- sk_buff结构中的字段众多，分为五类：
  - 数据布局字段
  - 链表队列字段
  - 协议通用字段
  - 特定功能字段
  - 管理函数字段



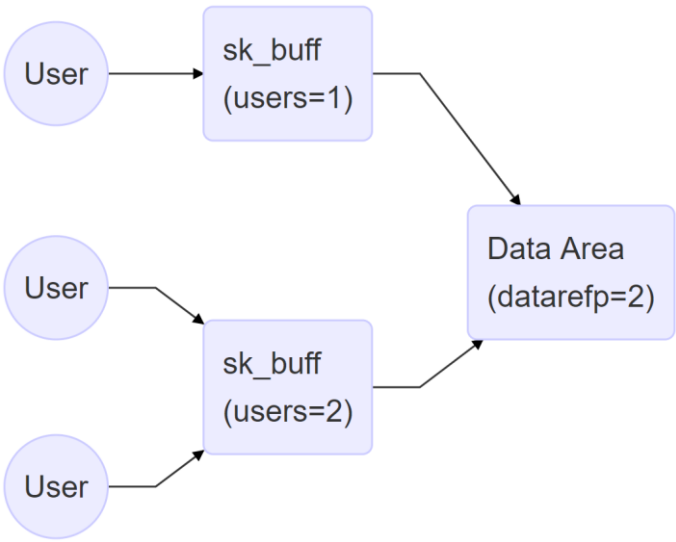
sk_buff中的数据布局字段

- sk_buff中的四个指针将数据区分成三个部分：
  - [head, data)：头部空间
  - [data, tail)：有效数据，报文负载和各层协议包头
  - [tail, end)：尾部空间
- 网络协议栈在处理报文的过程中经常需要添加或移除协议包头，这就需要不断改变有效数据区的大小：
  - 调整data指针的位置，改变头部空间的大小
  - 调整tail指针的位置，改变尾部空间的大小
  - 通过这两种方式都可以改变有效数据区域的大小，便于实现针对报文头部和尾部的处理

知识讲解

sk_buff中的数据布局字段

- sk_buff和数据区之间存在多对一的映射关系：
  - 多个套接字可以共享同一个数据区，减少不必要的内存分配，提高空间利用率
  - 数据区自己维护一个引用计数(datarefp)，记录当前引用该数据区的sk_buff数量：
    - 每增加一个引用该数据区的sk_buff，引用计数加1
    - 每减少一个引用该数据区的sk_buff，引用计数减1
    - 当引用计数为0时，表明已经没有任何sk_buff引用该数据区，可释放其所占内存
- sk_buff中还维护着一个关于自己的引用计数字段(users)，以记录该结构当前被多少个用户使用
- 这些引用计数的关系如下图所示：

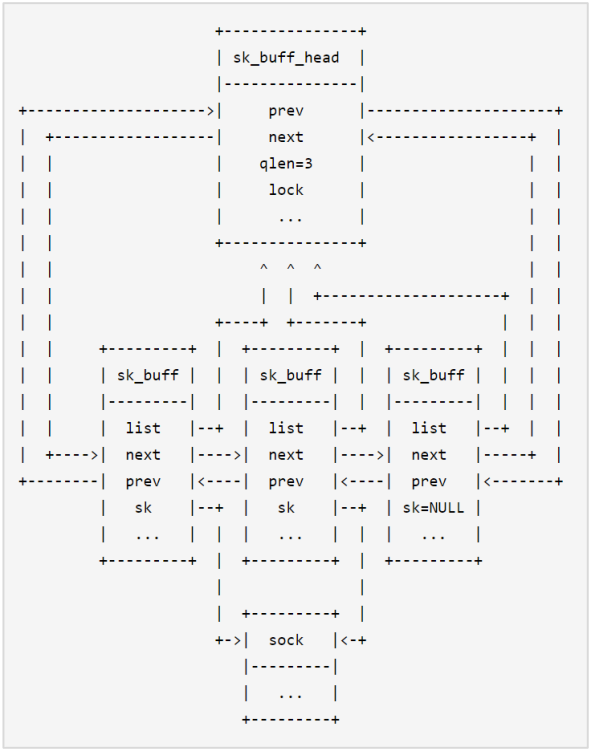


知识讲解

sk_buff中的链表队列字段

知识讲解

- 在Linux网络协议栈中，一个sk_buff表示一个报文，但很多报文之间存在着某种相关性，例如，在同一个套接字上等待接收的报文，或在同一个网络设备上等待发送的报文。为此协议栈采用双向链表结构将这些与报文相关的sk_buff组织成队列的形式。如下图所示：



- 双向链表的头部用sk_buff_head结构表示，其中的next和prev字段分别指向sk_buff队列的首尾两端，qlen字段记录队列的长度，lock字段为防止并发冲突的互斥锁
- sk_buff中除了维系双向链表结构的next和prev指针，还有一个list指针，指向该链表的头部，sk字段则指向一个sock结构，表示拥有该sk_buff的套接字。对于转发报文，其sk_buff中的sk字段为空

sk_buff中的协议通用字段

- sk_buff中的协议通用字段，即处理每种通信协议报文时都要用到的字段，如：

```
struct sk_buff {
    ...
    struct net_device * dev; // 网络设备
    struct dst_entry  * dst; // 路由缓存
    union { ... }       h;   // 传输层协议(如TCP、UDP等)包头
    union { ... }       nh;  // 网络层协议(如IP、ARP、IPX等)包头
    union { ... }       mac; // 数据链路层协议(如以太网等)包头
    ...
};
```

- 其中每层网络协议包头均为不同类型指针的联合，每种类型对应一种该层协议下的包头格式，如：

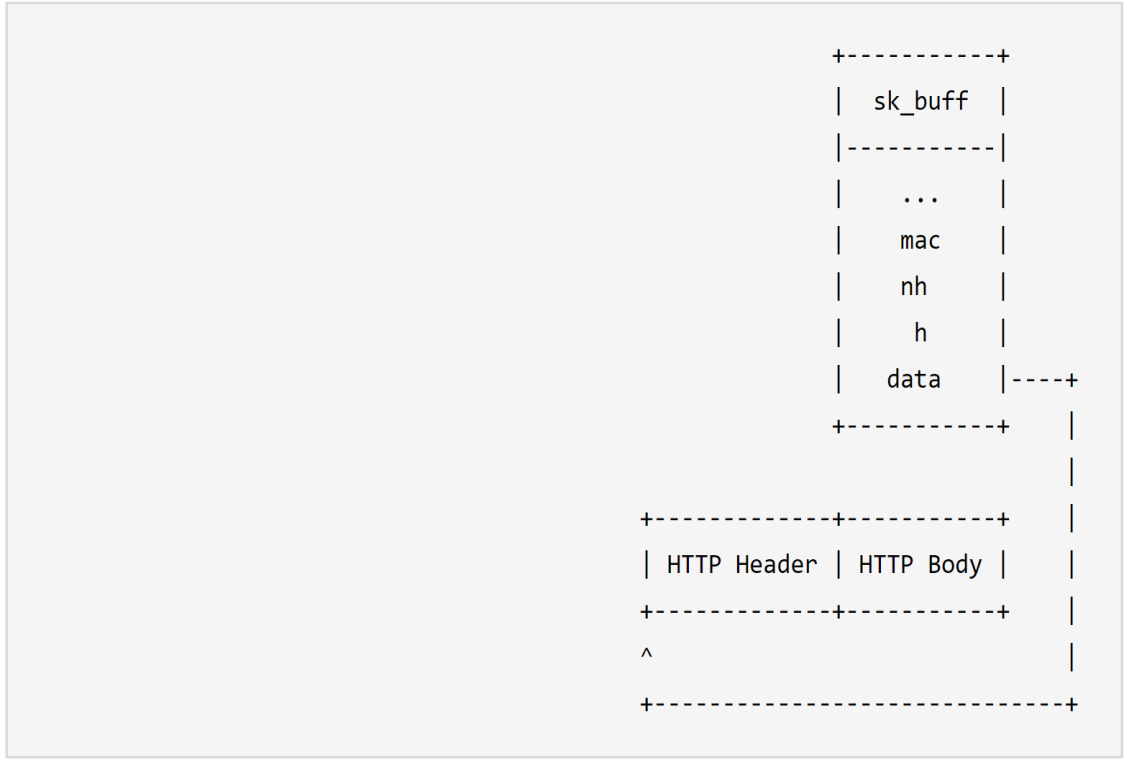
```
union {
    struct iphdr   * iph;   // IPv4协议包头
    struct ipv6hdr * ipv6h; // IPv6协议包头
    struct arphdr  * arph;  // ARP协议包头
    struct ipxhdr  * ipxh;  // IPX协议包头
    unsigned char  * raw;   // 尚未识别出具体协议时初始化
}   nh;
```

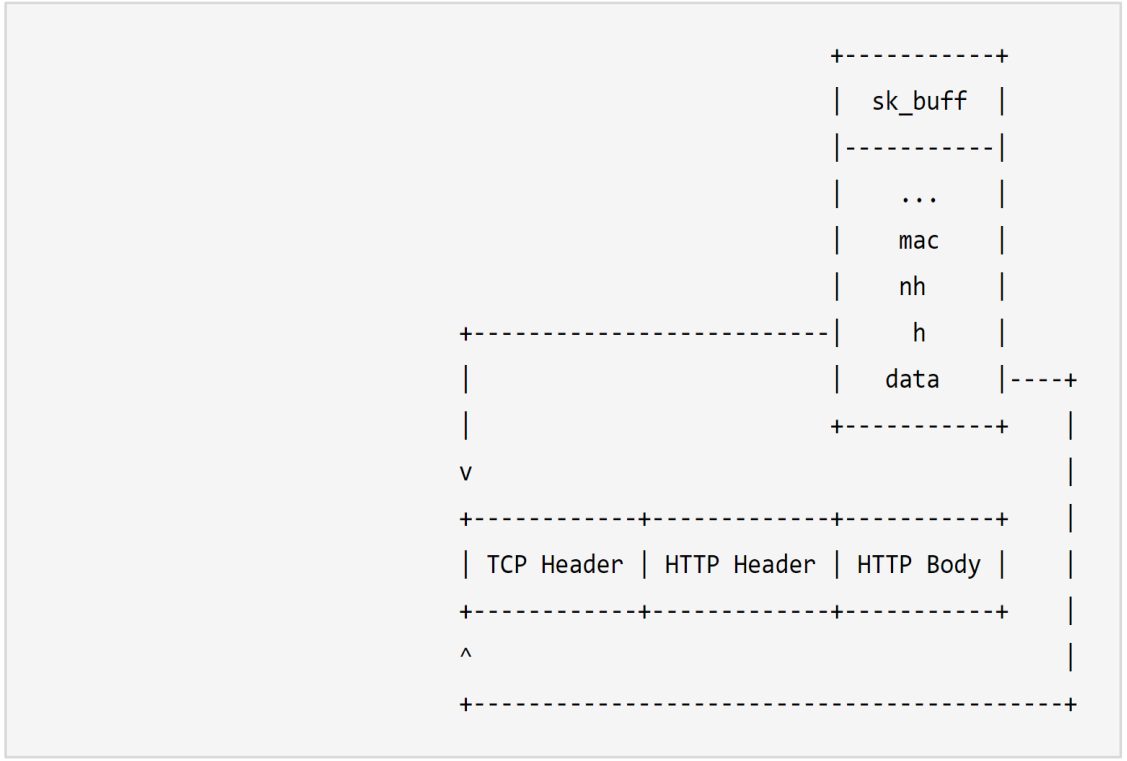
知识讲解

sk_buff中的协议通用字段

- sk_buff中的协议包头指针字段会在报文发送的过程中，逐层获得正确的协议包头地址：
  - 从应用层进入传输层：
  - 从传输层进入网络层：

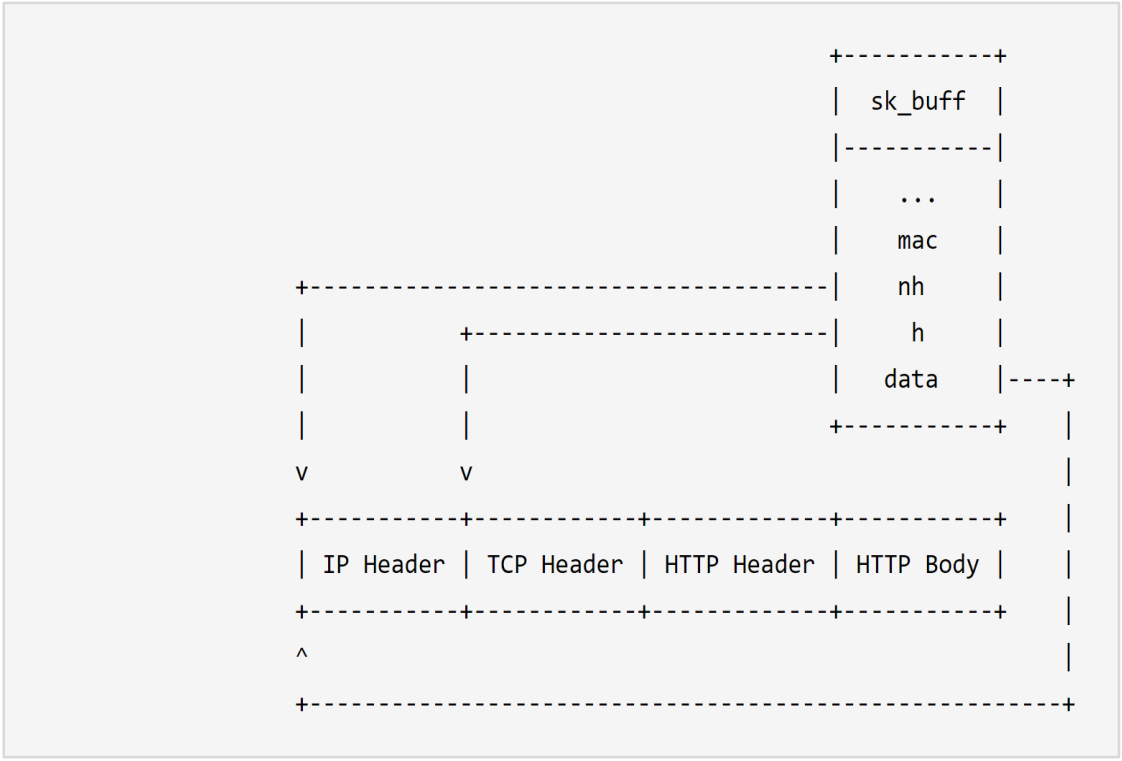
知识讲解



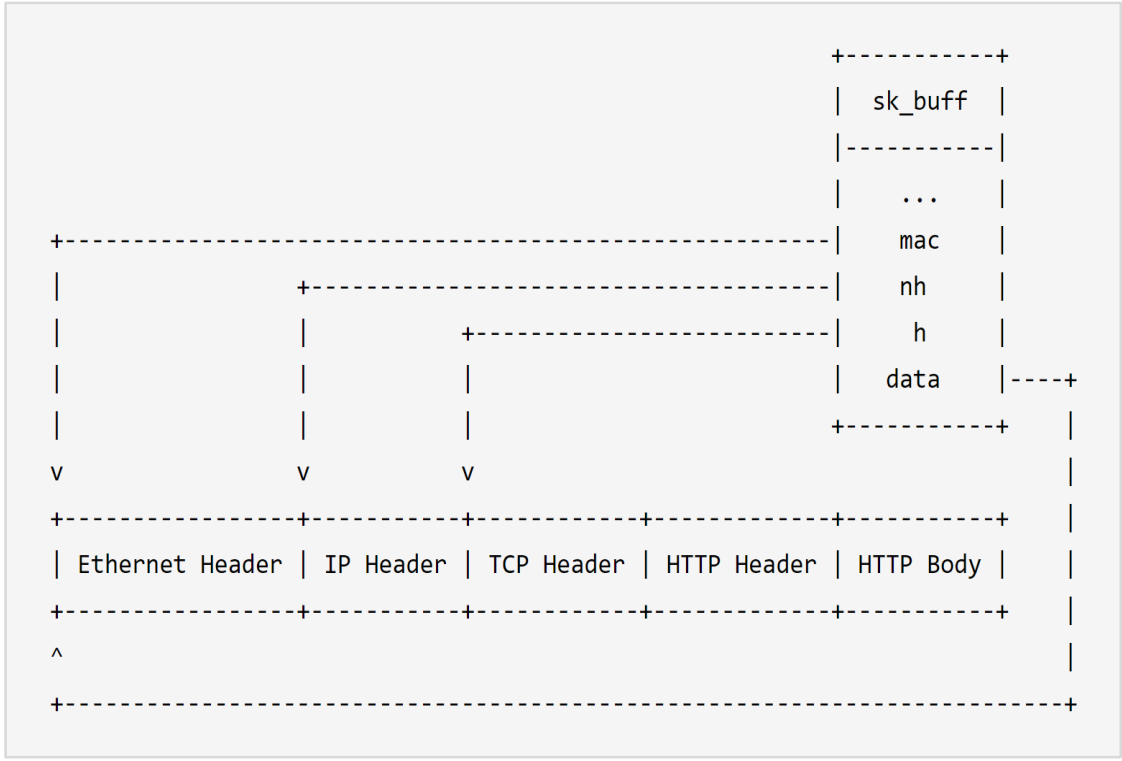


sk_buff中的协议通用字段

- sk_buff中的协议包头指针字段会在报文发送的过程中，逐层获得正确的协议包头地址：
  - 从网络层进入数据链路层：



  - 从数据链路层进入物理层：

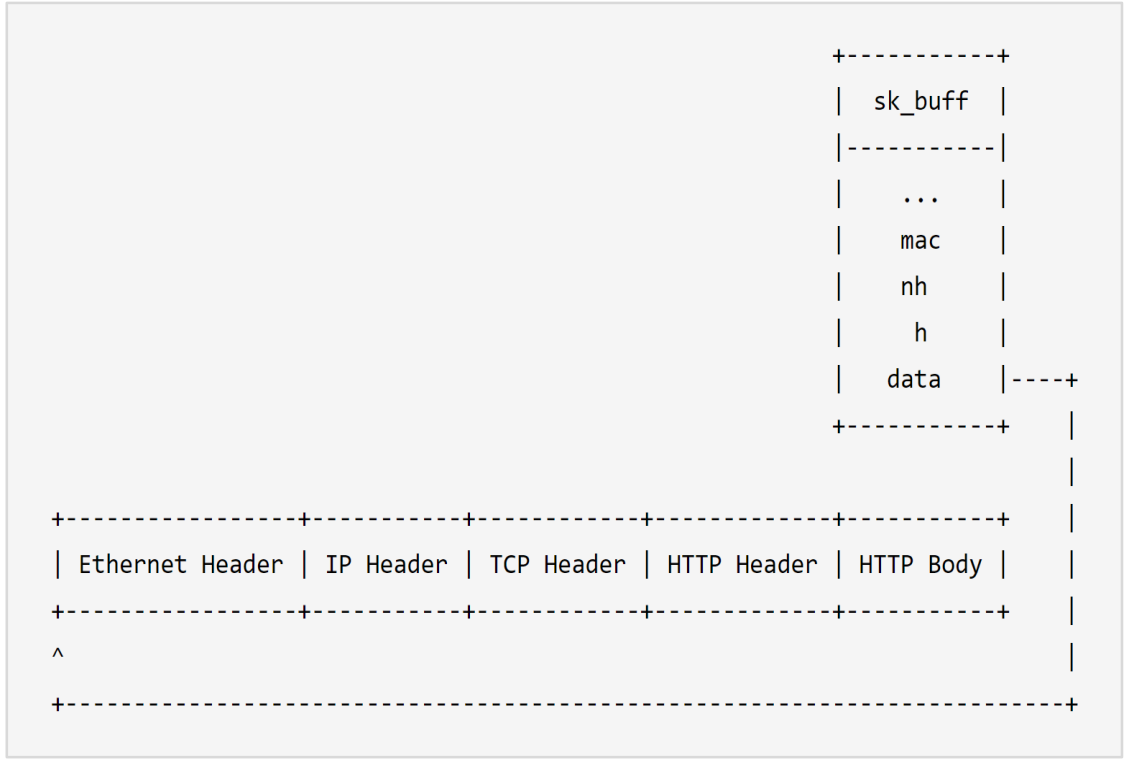


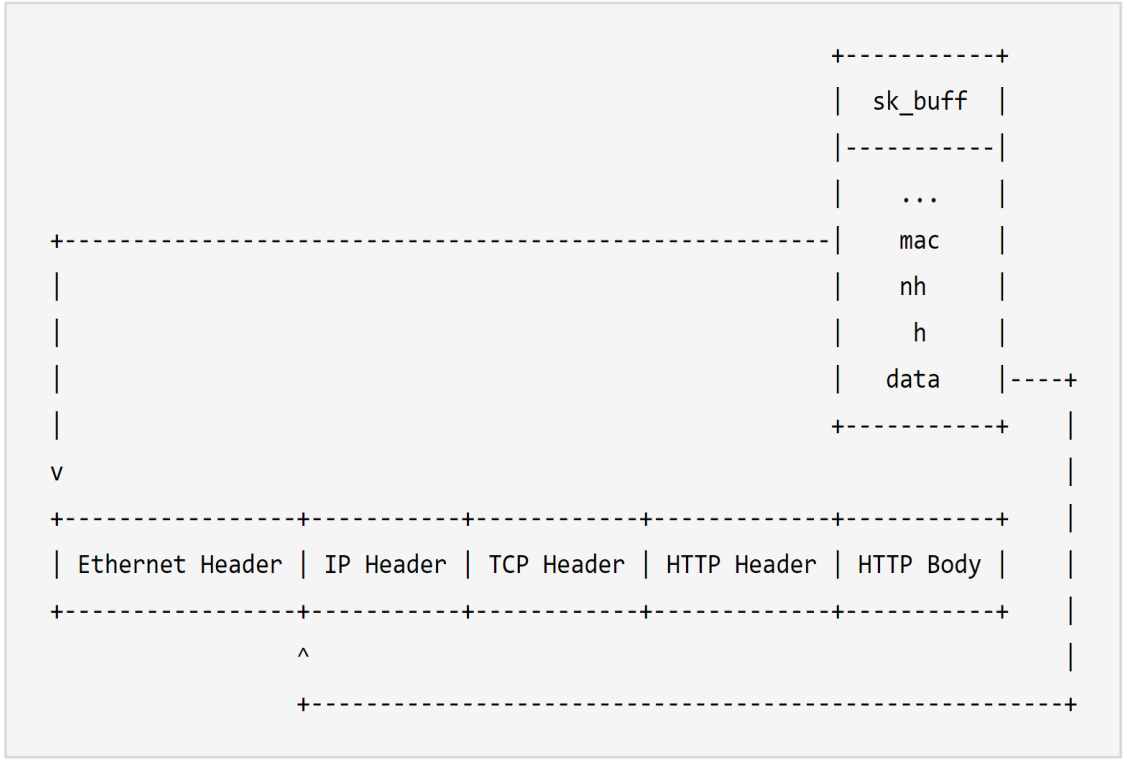
知识讲解

sk_buff中的协议通用字段

- sk_buff中的协议包头指针字段会在报文接收的过程中，逐层获得正确的协议包头地址：
  - 从物理层进入数据链路层：
  - 从数据链路层进入网络层：

知识讲解

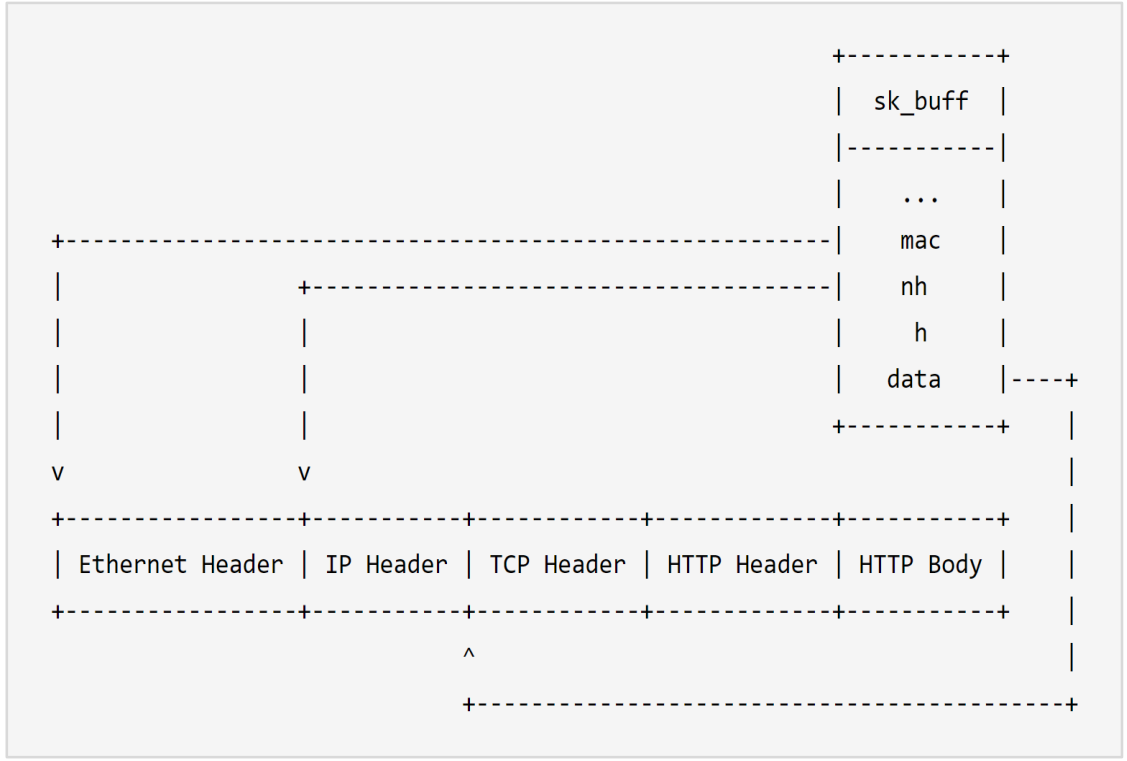


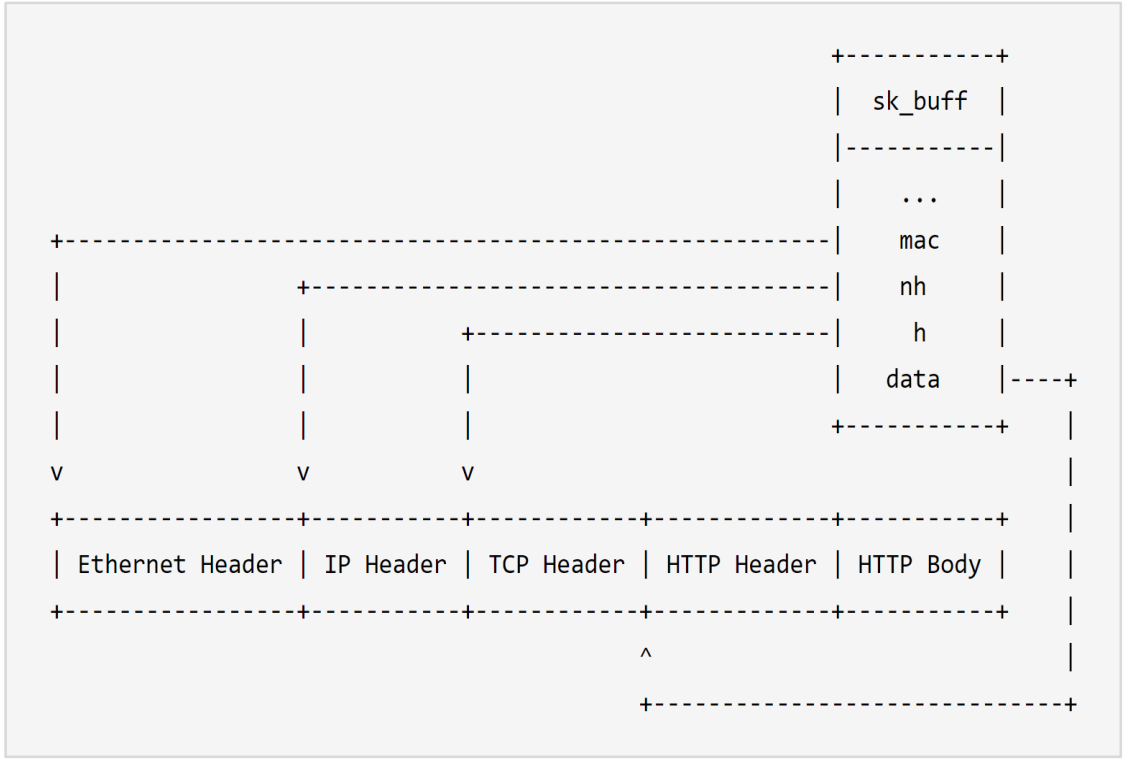


sk_buff中的协议通用字段

- sk_buff中的协议包头指针字段会在报文接收的过程中，逐层获得正确的协议包头地址：
  - 从网络层进入传输层：
  - 从传输层进入应用层：

知识讲解





sk_buff中的特定功能字段

- Linux网络协议栈中的很多功能都是按需提供的，如Netfilter等，为此sk_buff中定义了一些与特定功能有关的字段。这些字段只在需要实现相应的特定功能时才会用到

知识讲解

sk_buff中的管理函数字段

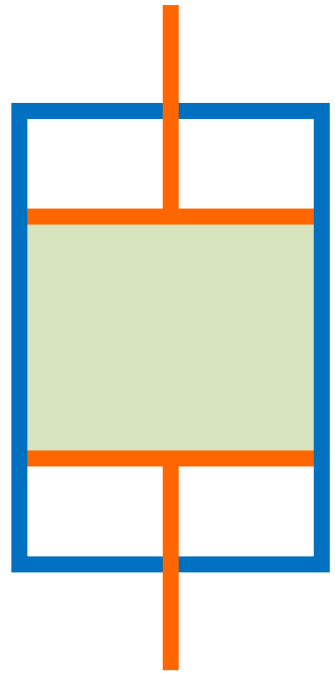
- 系统内核中用于管理sk_buff及其链表队列的函数通常都有两个版本：
  - 带下划线的版本，如_do_something，完成实际的功能
  - 不带下划线的版本，如do_something，先完成加锁、更新引用计数等额外工作，在条件满足时调用带下划线的版本
- sk_buff的创建与销毁：
  - alloc_skb函数用于创建sk_buff实例，在新创建的sk_buff中：
    - head、data和tail字段全部指向数据区开始的位置
    - end字段指向数据区结束的位置
  - kfree_skb函数用于销毁sk_buff实例，将sk_buff中的引用计数字段users减1，当其为0时：
    - 调用_kfree_skb函数，释放sk_buff所占用的内存
    - 调用kfree_skbmem函数，处理数据区的复用

知识讲解

sk_buff中的管理函数字段

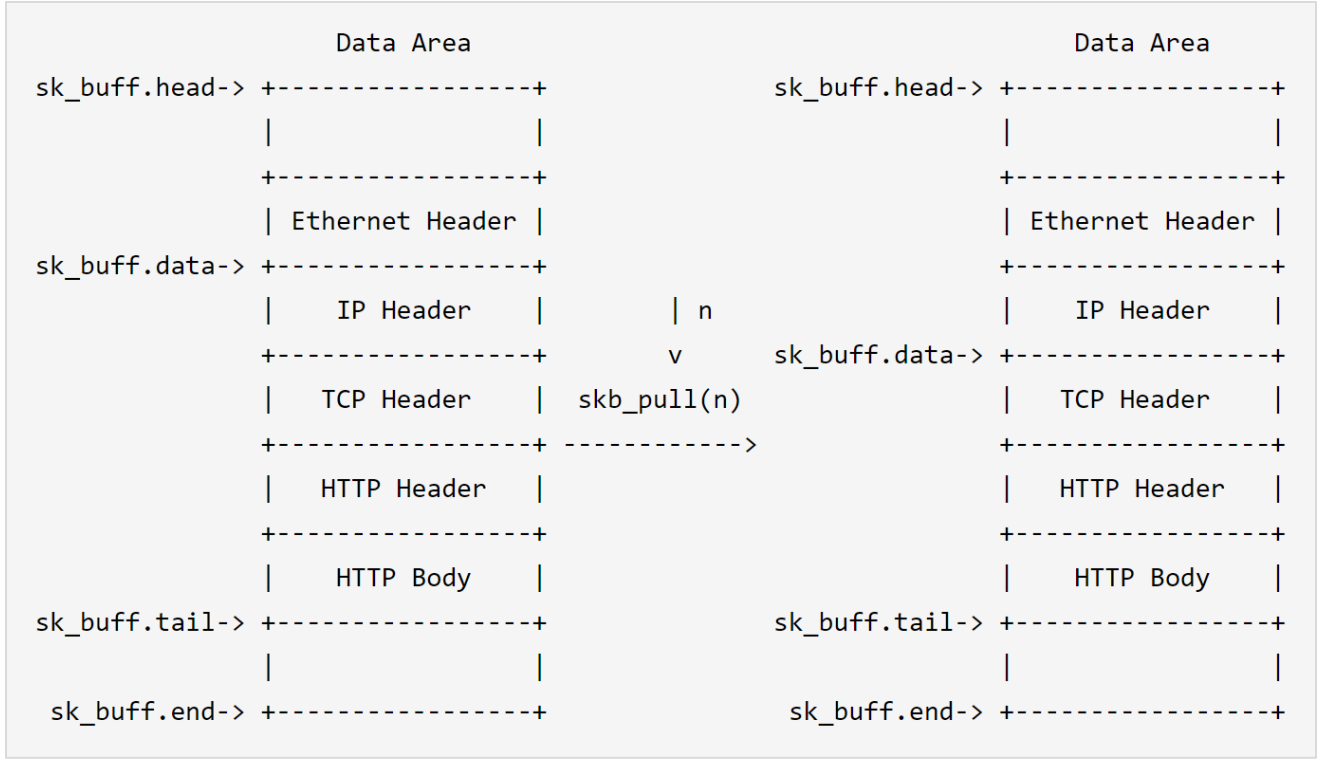
- sk_buff中的管理函数字段为一组函数指针，其所指向的函数用于管理sk_buff及其链表队列：
  - skb_reserve(n)：预留头部空间n个字节
  - skb_push(n)：上推data指针n个字节
  - skb_pull(n)：下拉data指针n个字节
  - skb_put(n)：下压tail指针n个字节
  - skb_trim(n)：调整tail指针使len为n个字节

知识讲解



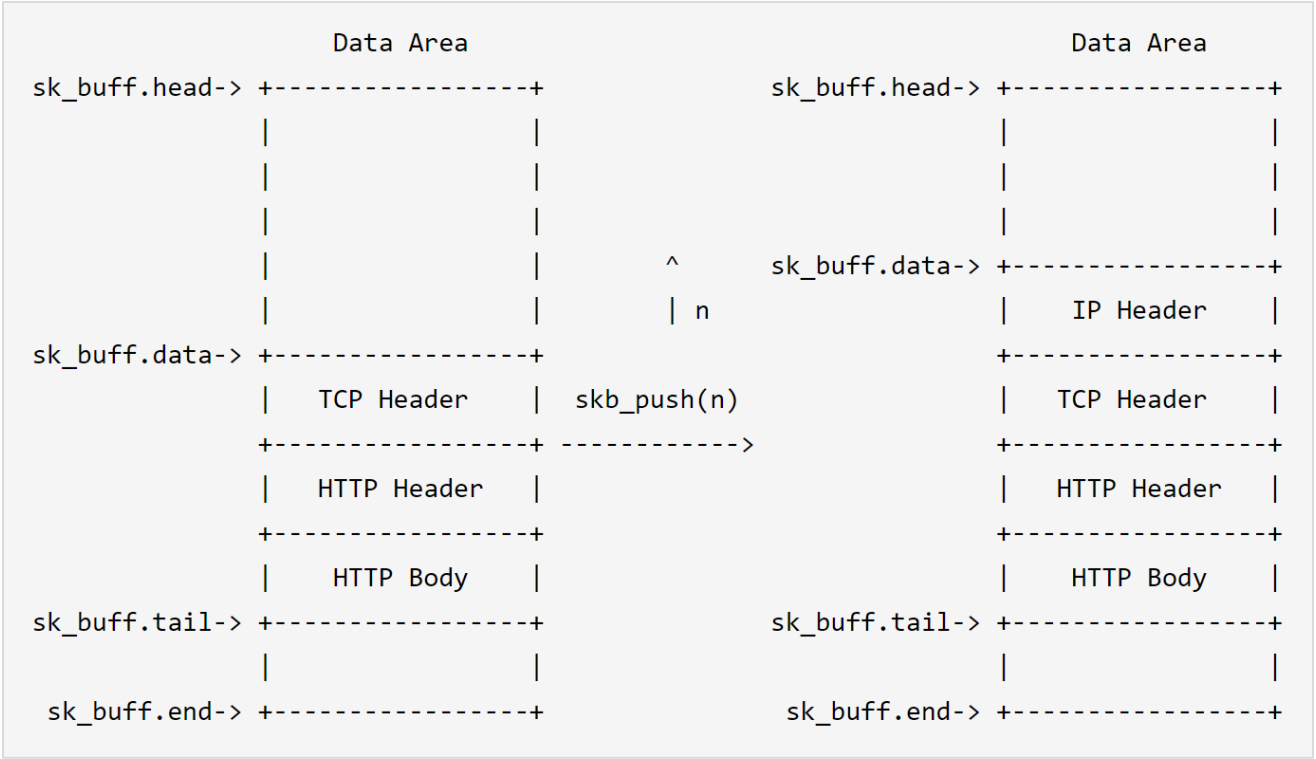
sk_buff中的管理函数字段

- 一旦完成数据区内存的分配，sk_buff中的head和end字段就固定不变，对有效数据的调整全在data和tail字段。例如：
  - 接收报文从数据链路层进入网络层：



sk_buff中的管理函数字段

- 一旦完成数据区内存的分配，sk_buff中的head和end字段就固定不变，对有效数据的调整全在data和tail字段。例如：
  - 发送报文从传输层进入网络层：

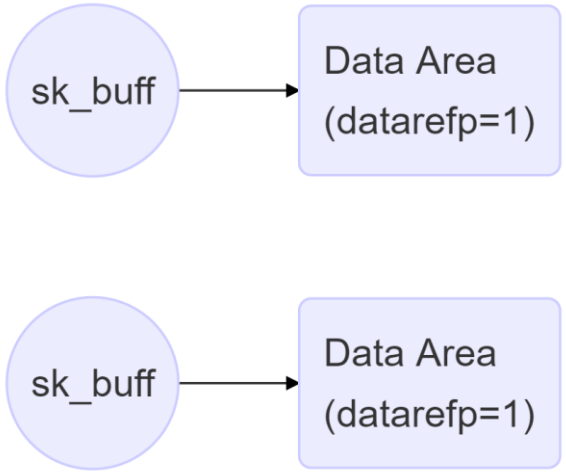


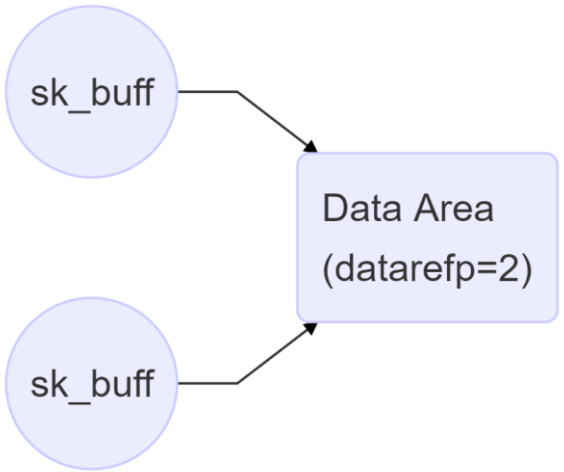
知识讲解

sk_buff中的管理函数字段

- 这些函数都只是修改sk_buff中特定指针字段的值，并不会引起数据区中数据的增加或减少
- sk_buff的复制与克隆：
  - skb_copy函数连同数据区一起复制，深拷贝
  - skb_clone函数只复制sk_buff不复制数据区，浅拷贝，更新数据区的引用计数(datarefp)

知识讲解





批量传输

批量传输

- sk_buff只是Linux网络协议栈内部用来表示报文的私有数据结构，其中既包含各层通信协议的包头，也包含对最终用户来说真正有意义的有效载荷。利用通常所说的套接字应用编程接口，即Socket API，编写网络应用程序，程序员只需要关心报文中的有效载荷即可，从传输层到数据链路层的各级包头完全由系统内核中的网络协议栈负责处理，除非使用了比较特殊的协议族(如PF_PACKET)和套接字类型(如SOCK_RAW)等。如下表所示：

知识讲解

批量传输

知识讲解

协议族	套接字类型	L2 数据链路层	L3 网络层	L4 传输层	L5 应用层
		以太网包头	IP包头	TCP/UDP包头	有效载荷
PF_INET	SOCK_STREAM SOCK_DGRAM	内核空间			用户空间
PF_INET	SOCK_RAW	内核空间		用户空间	
PF_PACKET	SOCK_DGRAM	内核空间	用户空间		
PF_PACKET	SOCK_RAW	用户空间			

批量传输

- 通过套接字应用编程接口收发报文，除了较为常用的recv和send函数外，还有一组基于消息的可以实现批量化数据传输的编程接口：

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recvmsg(int sockfd, struct msghdr* msg, int flags);       // 接收消息
ssize_t sendmsg(int sockfd, const struct msghdr* msg, int flags); // 发送消息
```

- 其中表示消息的msghdr结构：

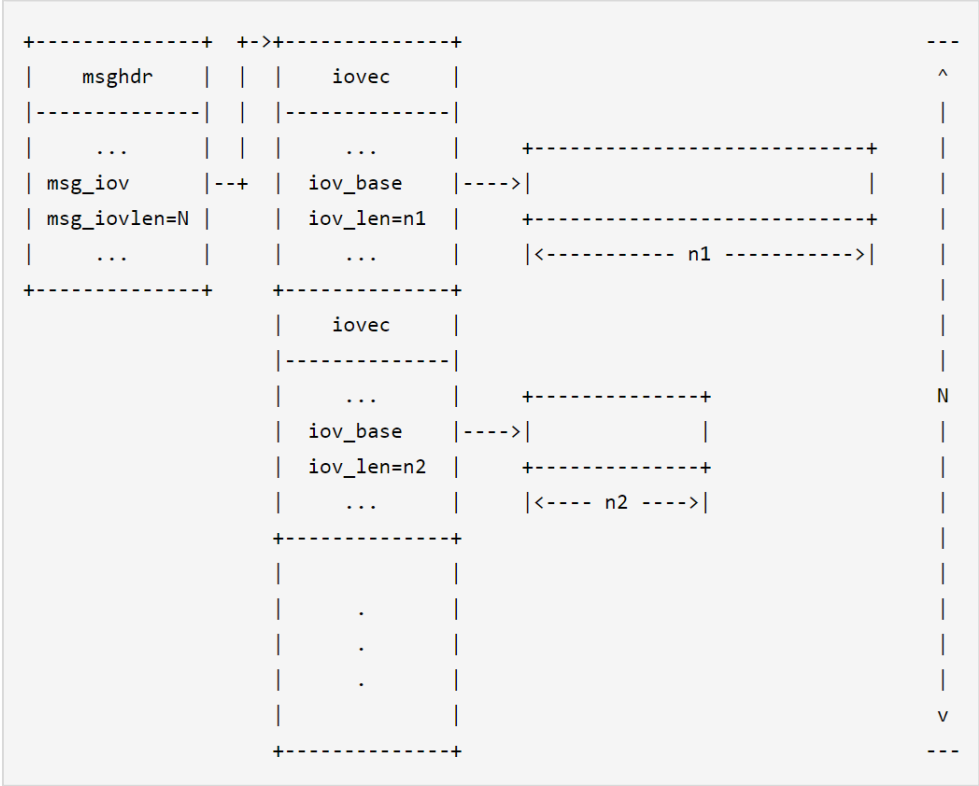
```
#include<sys/socket.h>

struct msghdr {
    void         * msg_name;       // 可选地址
    socklen_t      msg_namelen;    // 可选地址字节数
    struct iovec * msg_iov;        // I/O缓冲区数组
    int            msg_iovlen;     // I/O缓冲区数组元素数
    void         * msg_control;    // 附加数据
    socklen_t      msg_controllen; // 附加数据字节数
    int            msg_flags;      // 接收标志
    ...
};
```

知识讲解

批量传输

- msghdr结构的msg_iov字段指向一个包含msg_iovlen个元素的数组，该数组的每个元素都是iovec结构，该结构的iov_base字段指向真正的数据缓冲区，iov_len字段表示该缓冲区的长度。如下图所示：
- 基于这样的msghdr结构，可将总共msg_iovlen块数据作为一个整体，以批量的方式交给套接字，大大提高了收发报文的效率。事实上，不但在应用程序中可以这种方式实现批量传输，在Linux网络协议栈中同样可以采用这种方式收发报文



知识讲解

报文发送

套接字函数层(L5-用户态应用层)

- 以下是一个TCP客户端应用程序向服务器发送并接收数据的示意代码：
- 对于套接字而言，总共有四个函数可用于发送数据：
  - write
    - Linux系统将套接字表示为一个文件描述符，因此可以使用文件系统提供的write函数发送数据
    - write函数是为写文件而设计的通用接口，无法支持网络通信的所有功能，如发送带外数据等

知识讲解

```
// 创建套接字
int sockfd = socket(PF_INET, SOCK_STREAM, 0);

// 获取服务器地址
struct hostent* server = gethostbyname(SERVER_NAME);

// 填写地址结构
struct sockaddr_in address;
address.sin_family = AF_INET;
address.sin_port = htons(PORT_NUM);
memcopy(&address.sin_addr, server->h_addr, server->h_length);

// 连接服务器
connect(sockfd, &address, sizeof(address));

// 发送数据
write(sockfd, "Hello World !", 13);
// 接收数据
read(sockfd, buf, BUFLEN);

// 关闭套接字
close(sockfd);
```

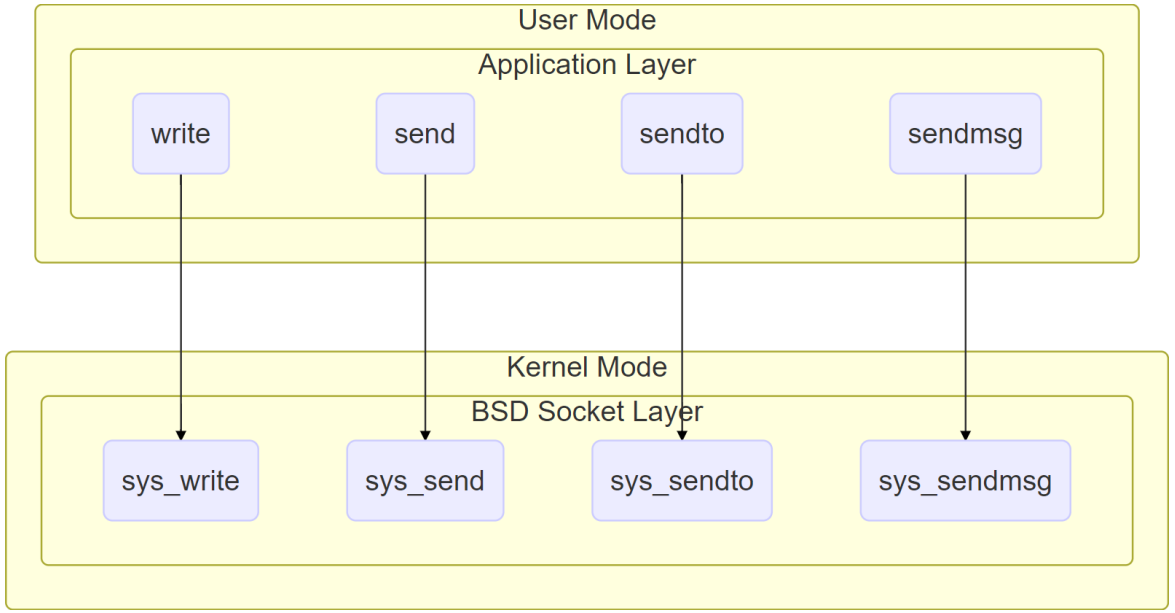
套接字函数层(L5-用户态应用层)

- 对于套接字而言，总共有四个函数可用于发送数据：
  - send
    - 与write函数十分相似，前三个参数与write函数完全相同
    - 增加了第四个参数flags，以实现一些套接字所特有的功能，如用MSG_OOB表示带外数据等
  - sendto
    - sendto函数最大的特点在于提供了设定目的地址和目的端口的能力
    - 对于面向连接的套接字(SOCK_STREAM)，协议栈会使用既有的连接发送数据，而忽略该函数指定的目的地址和端口号
    - 对于面向无连接的套接字(SOCK_DGRAM)，只能使用sendto函数发送数据，否则会因为缺少目的信息而产生错误

知识讲解

套接字函数层(L5-用户态应用层)

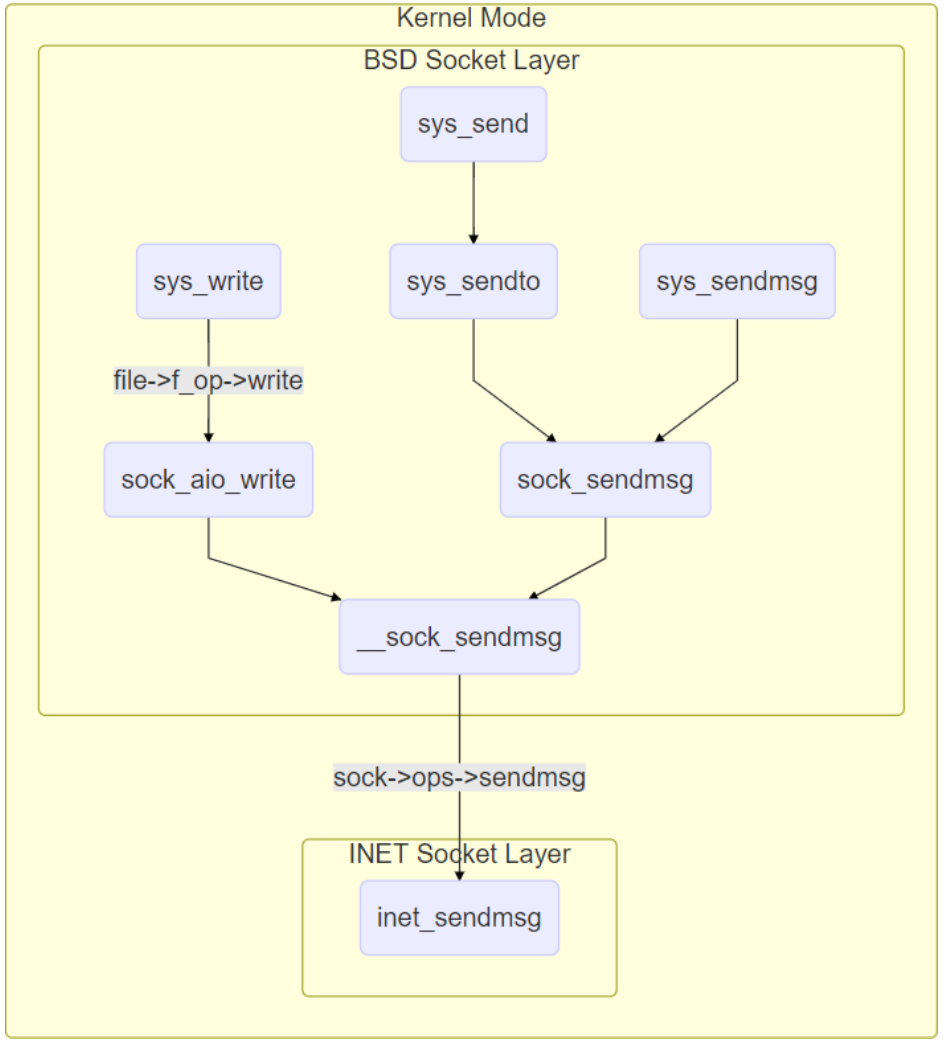
- 对于套接字而言，总共有四个函数可用于发送数据：
  - sendmsg
    - sendmsg函数的功能与send函数类似
    - 在参数中使用了msghdr结构，可以一次性发送由多个数据缓冲区组成的批量报文
- 套接字函数层的四个发送函数都运行于用户态，它们会通过BSD套接字层的四个对应的函数进入内核态。如下图所示：



知识讲解

BSD套接字层(L5-内核态应用层)

- BSD套接字层所有带sys_前缀的函数都是系统调用。执行系统调用将使程序运行于内核态。四个发送函数最终都会调用__sock_sendmsg函数，该函数通过msghdr结构管理待发送数据。msghdr结构中保存的仅仅是数据缓冲区的指针，真正的数据仍然存放在应用程序指定的用户态缓存中，直到tcp_sendmsg函数准备实际发送报文时，才会将数据拷贝进内核态。如下图所示：



知识讲解

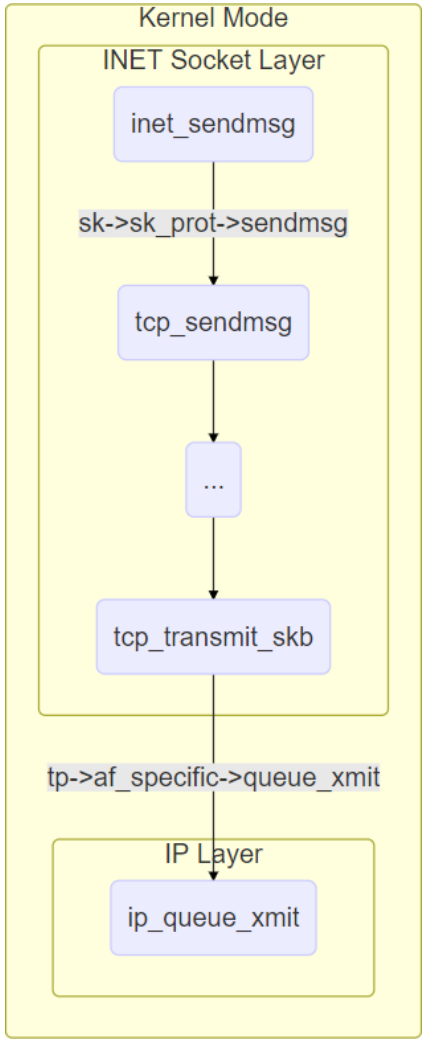
BSD套接字层(L5-内核态应用层)

- 函数指针sock->ops->sendmsg的作用是将BSD套接字接口映射到不同协议族的具体实现上。对于PF_INET协议族，该函数指针指向INET套接字层的inet_sendmsg函数。对于其它协议族，如ATM、IPX/SPX等，该函数指针也可指向其它实现函数

知识讲解

INET套接字层(L4-传输层)

- 在INET套接字层，inet_sendmsg函数的主要工作是通过函数指针sk->sk_prot->sendmsg调用tcp_sendmsg函数。如果使用其它类型的套接字，如SOCK_DGRAM、SOCK_RAW等，该函数指针也可指向其它实现相应服务的函数。如下图所示：



知识讲解

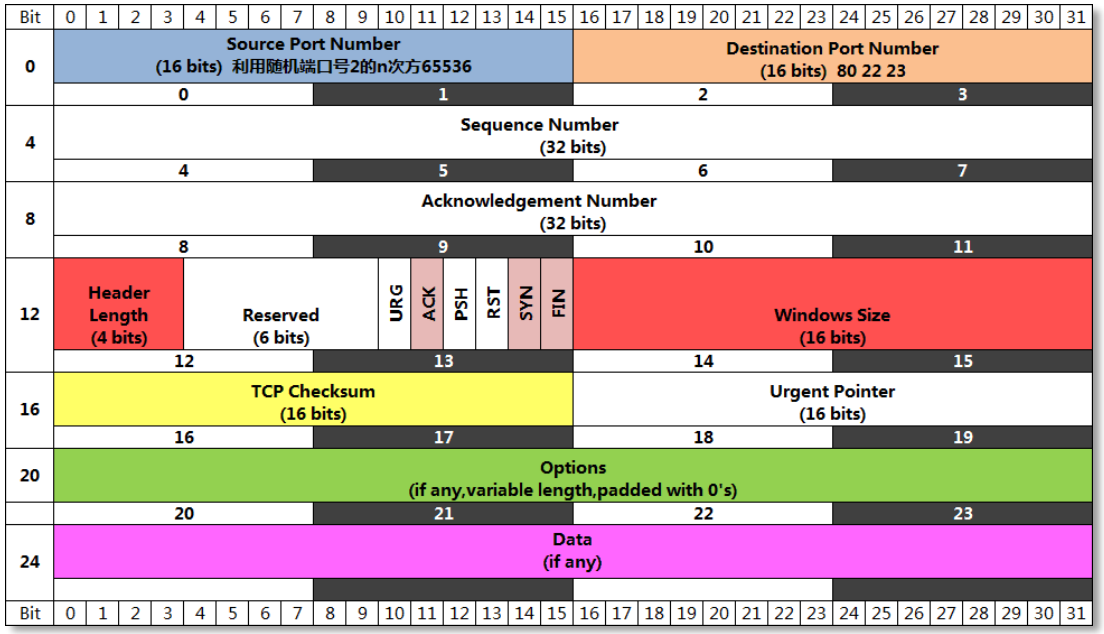
INET套接字层(L4-传输层)

- tcp_sendmsg函数是上层协议发送TCP报文的接口，它的主要任务是创建sk_buff结构，并将用户态缓存中的待发送数据拷贝到内核由sk_buff管理的数据区中。生成报文的大小受TCP最大报文长度(Maximum Segment Size, MSS)的限制，因此一次发送的数据可能需要分配到几个不同的sk_buff中。TCP是一种流式传输服务，不需要在报文中体现数据边界，可以在填满前一个sk_buff的数据区后再创建并填充下一个sk_buff
- TCP协议在发送报文时还需要考虑诸如流量控制、拥塞控制、超时重传以及避免糊涂窗口综合症等因素，因此报文的实际发送过程，在经调度后可能会被异步化，具体实现细节在上图中用省略号表示

知识讲解

INET套接字层(L4-传输层)

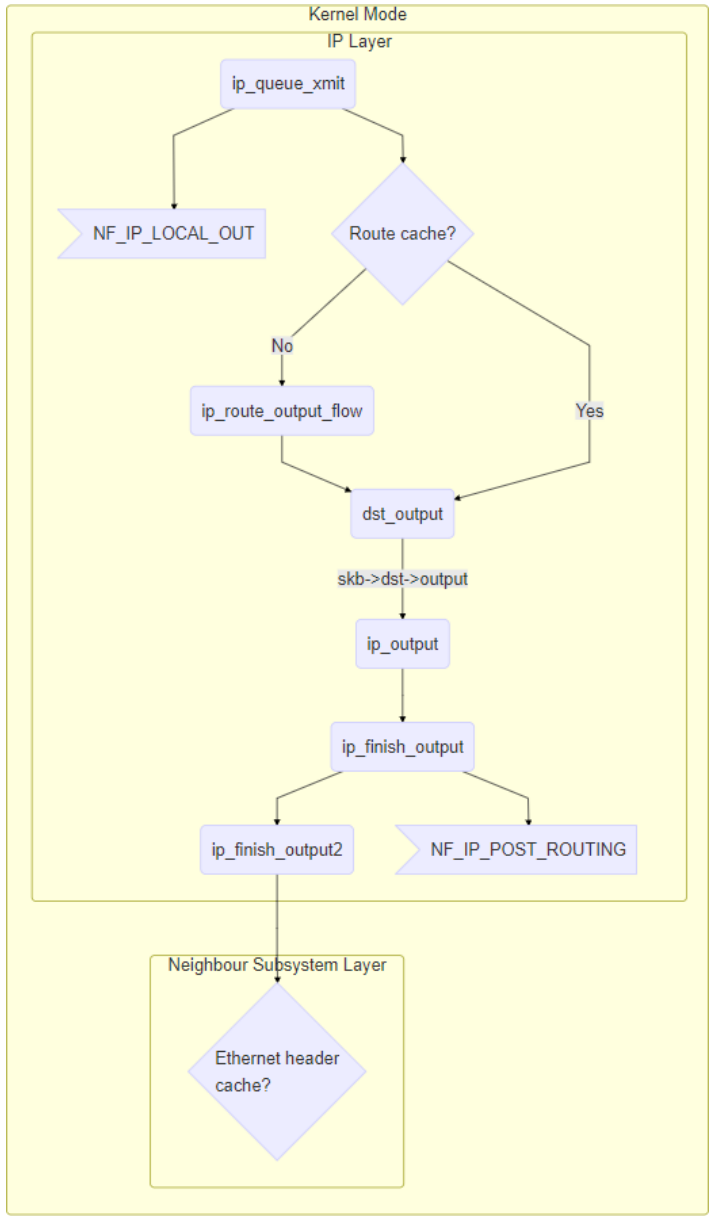
- TCP报文的发送流程最终要由tcp_transmit_skb函数构造TCP包头，并借助函数指针tp->af_specific->queue_xmit调用IP层的接口函数。如果是IPv4协议，该函数指针指向ip_queue_xmit函数，IPv6等其它IP层协议也有相对应的函数



知识讲解

IP层(L3-网络层)

- ip_queue_xmit函数会为本机发送的报文查找合适的路由信息，然后构造IP包头，并交由下一层继续处理。如下图所示：



知识讲解

IP层(L3-网络层)

- 路由缓存
  - 因为从同一个套接字发送的报文都具有完全相同的下一跳地址，所以只有连接过程中发送的第一个报文需要通过查找路由表确定下一跳地址，其余报文利用之前的路由缓存即可
  - Linux网络协议栈定义了与协议无关的和针对具体协议的多种路由缓存结构：
    - 与协议无关的路由缓存结构：

```
struct dst_entry {
    struct dst_entry* next;
    ...
};
```

    - 针对IP协议的路由缓存结构：

```
struct rtable {
    union {
        struct dst_entry dst;
        struct rtable* rt_next;
    }   u;
    ...
};
```

    - 以面向对象的观点来看，可将rtable视作继承自dst_entry的子类：
      - 任何时候，将一个rtable*转换为dst_entry*总是安全的
      - 在确悉一个dst_entry*指向的是一个rtable时，将其转换为rtable*也是安全的

知识讲解

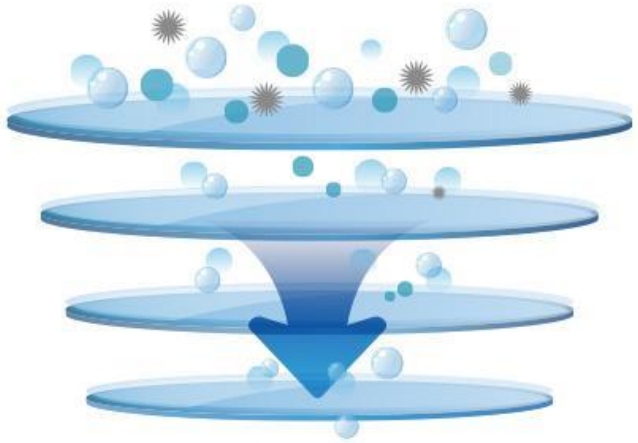
IP层(L3-网络层)

- 路由缓存
  - tcp_sendmsg函数在创建sk_buff结构时：
    - 会令其sk字段指向拥有该sk_buff的套接字，其类型为sock，其中dst_entry*类型的字段sk_dst_cache即指向与该套接字相关联的路由缓存。对于IP协议，路由缓存的实际类型为rtable
    - 若该套接字的sk_dst_cache字段非空，即已存在路由缓存，tcp_sendmsg函数会用该字段的值初始化所创建sk_buff结构中表示路由缓存且同为dst_entry*类型的字段dst，并以此作为报文完成路由的标志
  - ip_queue_xmit函数会根据sk_buff中的dst字段是否为空，判断报文是否已完成路由：
    - 已完成路由，直接通过sk_buff中的dst字段从缓存中获取路由
    - 未完成路由，调用ip_route_output_flow函数选择路由，将其缓存结构指针保存在拥有该sk_buff的套接字的sk_dst_cache字段中。之后在此套接字上创建的sk_buff都会通过其dst字段持有该指针

知识讲解

IP层(L3-网络层)

- 报文过滤
  - ip_queue_xmit函数是从本机发送IP报文的必由之路，非常适合安排对待发送报文做检查和过滤工作，因此Netfilter框架在该函数中设置了钩子函数的挂接点NF_IP_LOCAL_OUT，用来检查从本机发送的所有IP报文



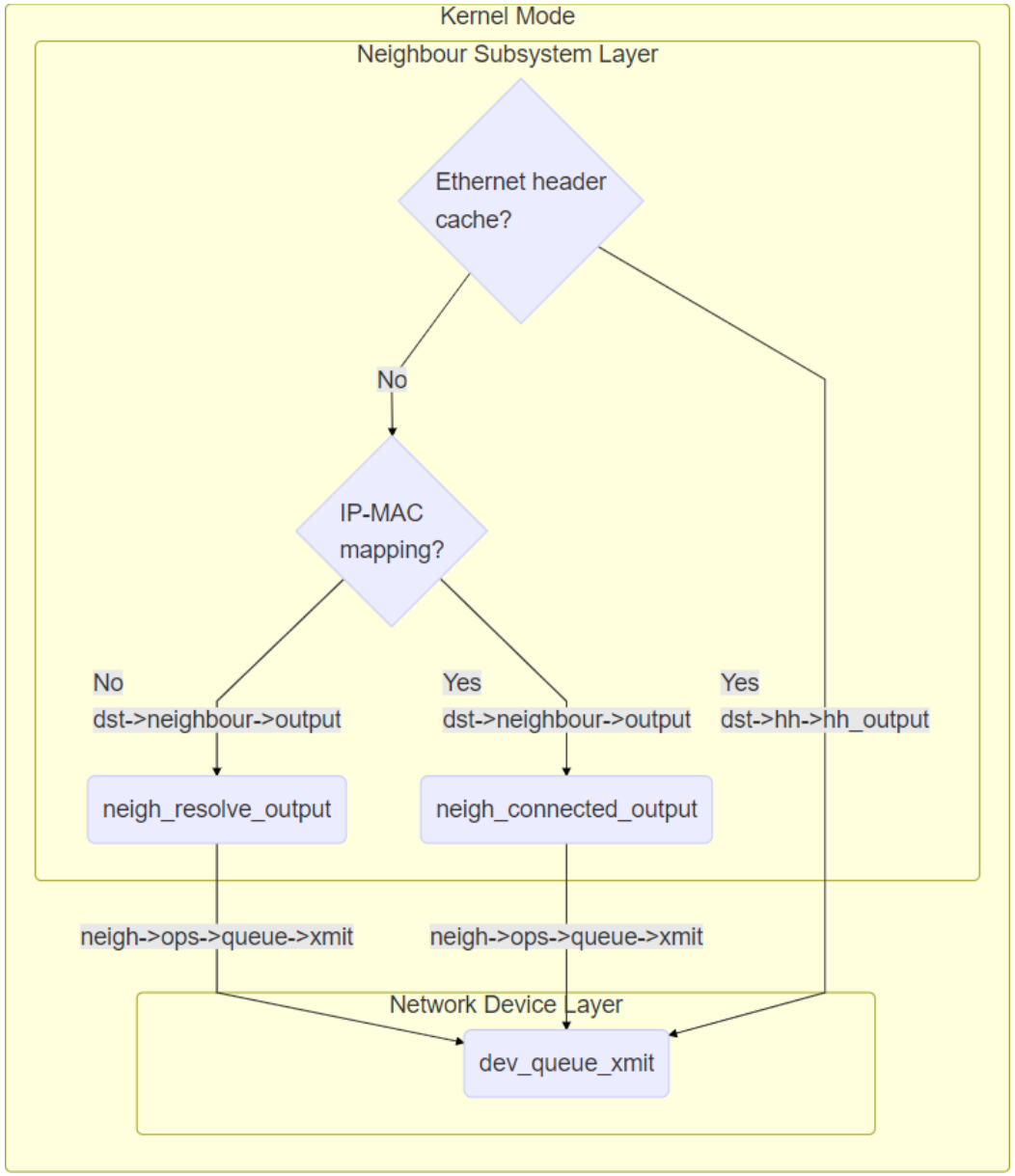
IP层(L3-网络层)

- 额外处理
  - 在某些情况下，协议栈需要在IP层完成一些额外的处理，如添加新的IP包头实现隧道功能，或添加IP安全协议(IP Security Protocol, IPSec)包头等。Linux网络协议栈将这些可能需要的额外处理组织成一个函数指针链表，skb->dst->output指针即指向该链表，而dst_output函数的本质就是遍历这个函数指针链表，通过函数指针调用完成上述额外处理任务的函数。函数指针链表的最后一个函数指针指向ip_output函数，该函数负责将报文传递给下一层继续处理
  - 接下来被ip_output函数调用的是ip_finish_output函数，该函数在完成对Netfilter框架NF_IP_POST_ROUTING钩子函数的调用后，通过ip_finish_output2函数进入邻居子系统层

知识讲解

邻居子系统层(L2-数据链路层)

- 邻居子系统层在将报文交给网络设备层之前需要完成以下两项任务：
  - 将网络层地址(如IP地址)转换为数据链路层地址(如MAC地址)
  - 创建数据链路包头(如以太网包头)
- 邻居子系统的执行过程如下图所示：



知识讲解

邻居子系统层(L2-数据链路层)

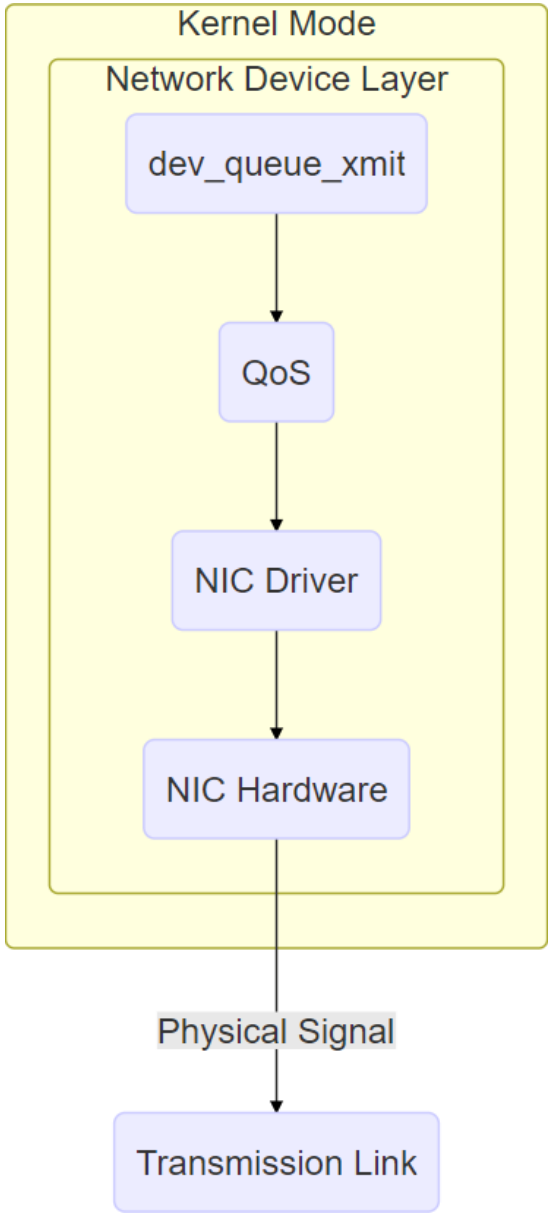
- 进入邻居子系统后，首先检查是否存在以太网包头的缓存：
  - 若存在，直接将缓存中的以太网包头拷贝到sk_buff中，完成以太网包的构建，通过函数指针dst->hh->hh_output调用dev_queue_xmit函数，进入网络设备层，发送报文
  - 不存在，先检查是否存在下一跳IP地址到MAC地址的映射：
    - 若存在，通过函数指针dst->neighbour->output调用neigh_connected_output函数，直接从映射中获取与下一跳IP地址相对应的MAC地址，创建以太网包头，形成以太网包
    - 不存在，通过函数指针dst->neighbour->output调用neigh_resolve_output函数，借助地址解析协议(如ARP)，获得与下一跳IP地址相对应的MAC地址，创建以太网包头，形成以太网包
    - 最后通过函数指针neigh->ops->queue->xmit调用dev_queue_xmit函数，进入网络设备层，发送报文

知识讲解

网络设备层(L1-物理层)

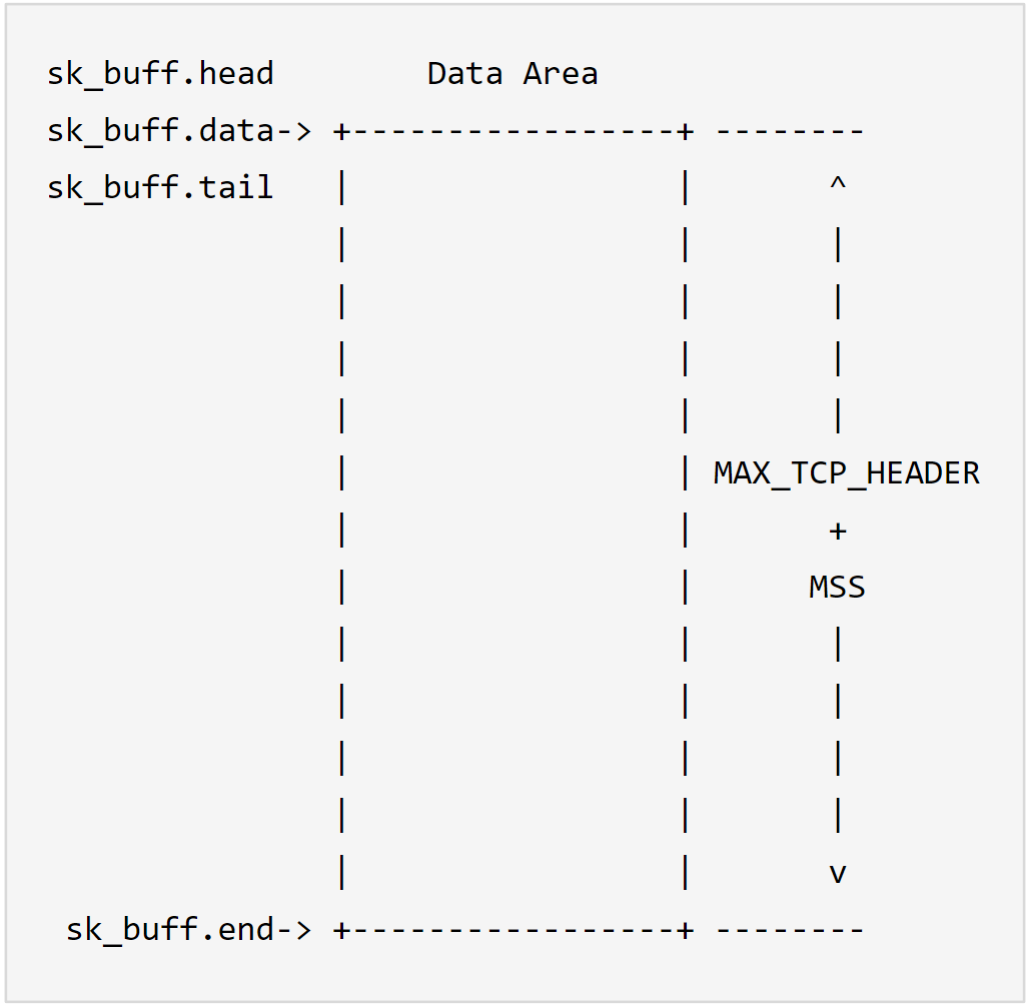
- 无论走哪一条路，最终都会调用dev_queue_xmit函数，它是进入网络设备层的唯一入口。如图所示：
- 在网络设备层，经过必要的流量控制，报文通过网卡驱动被发送到网卡硬件，最终以物理信号的形式进入网络传输线路

知识讲解



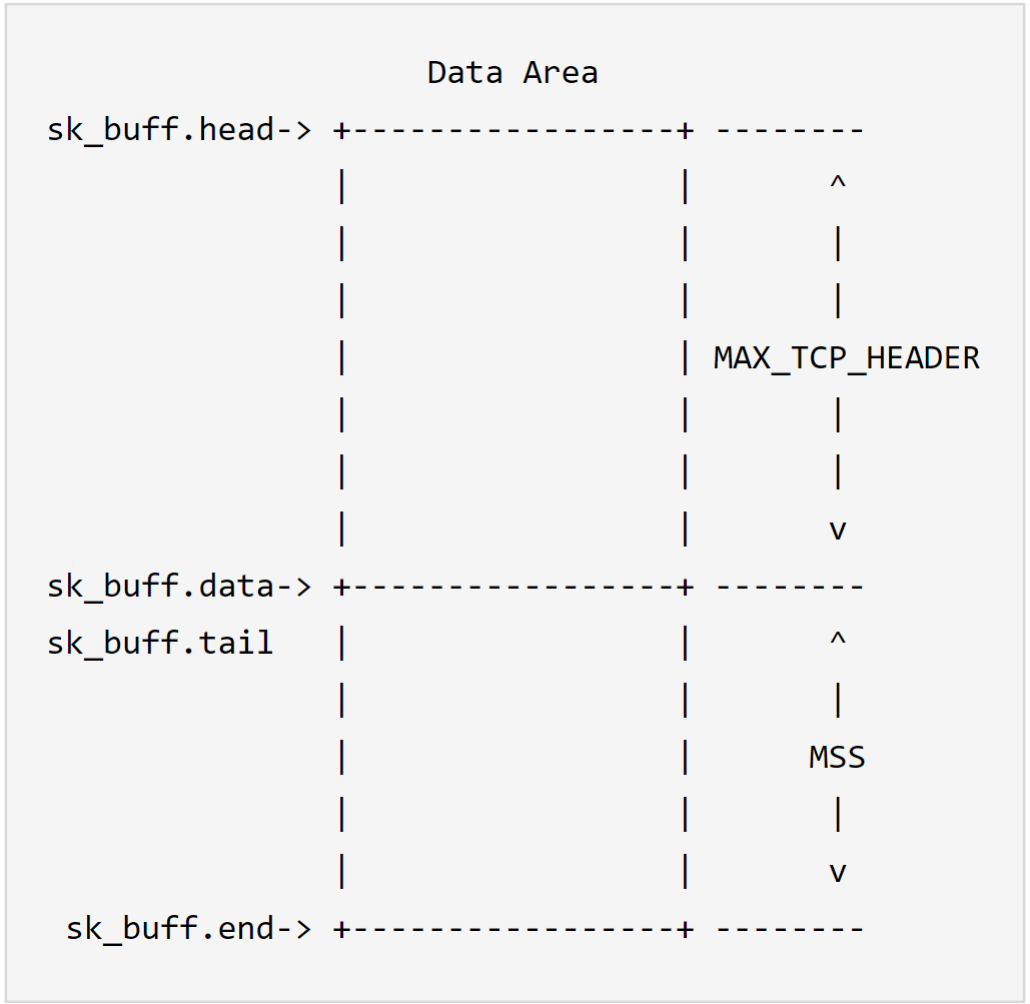
sk_buff的变化过程

- 在TCP报文的发送过程中，sk_buff要依次经历如下变化过程：
  - 当套接字发现有数据需要发送时，首先在tcp_sendmsg函数中向内核申请创建sk_buff结构来表示报文。用于存放报文的数据区至少大于TCP最大报文长度(Maximum Segment Size, MSS)与各层包头长度之和：



sk_buff的变化过程

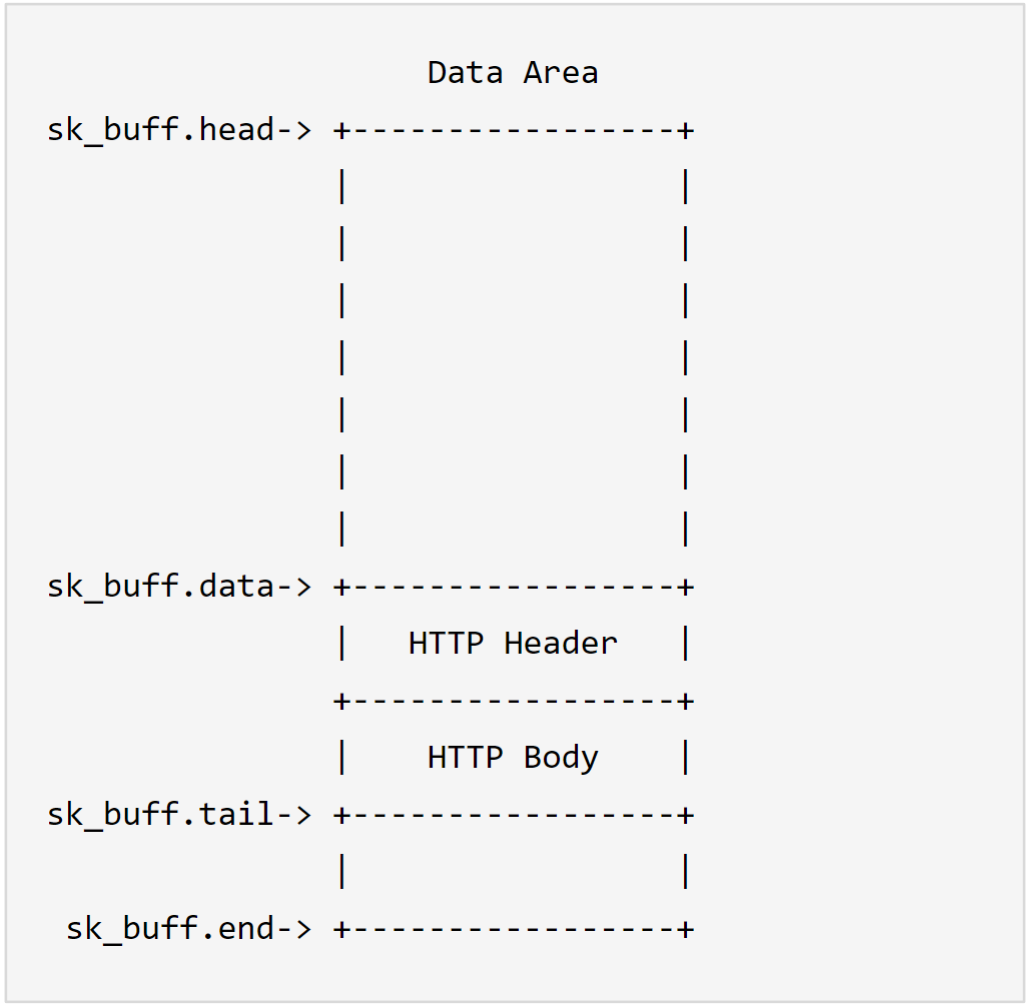
- 在TCP报文的发送过程中，sk_buff要依次经历如下变化过程：
  - tcp_sendmsg函数通过skb_reserve函数在数据区中为各层协议包头预留总共MAX_TCP_HEADER字节空间：



知识讲解

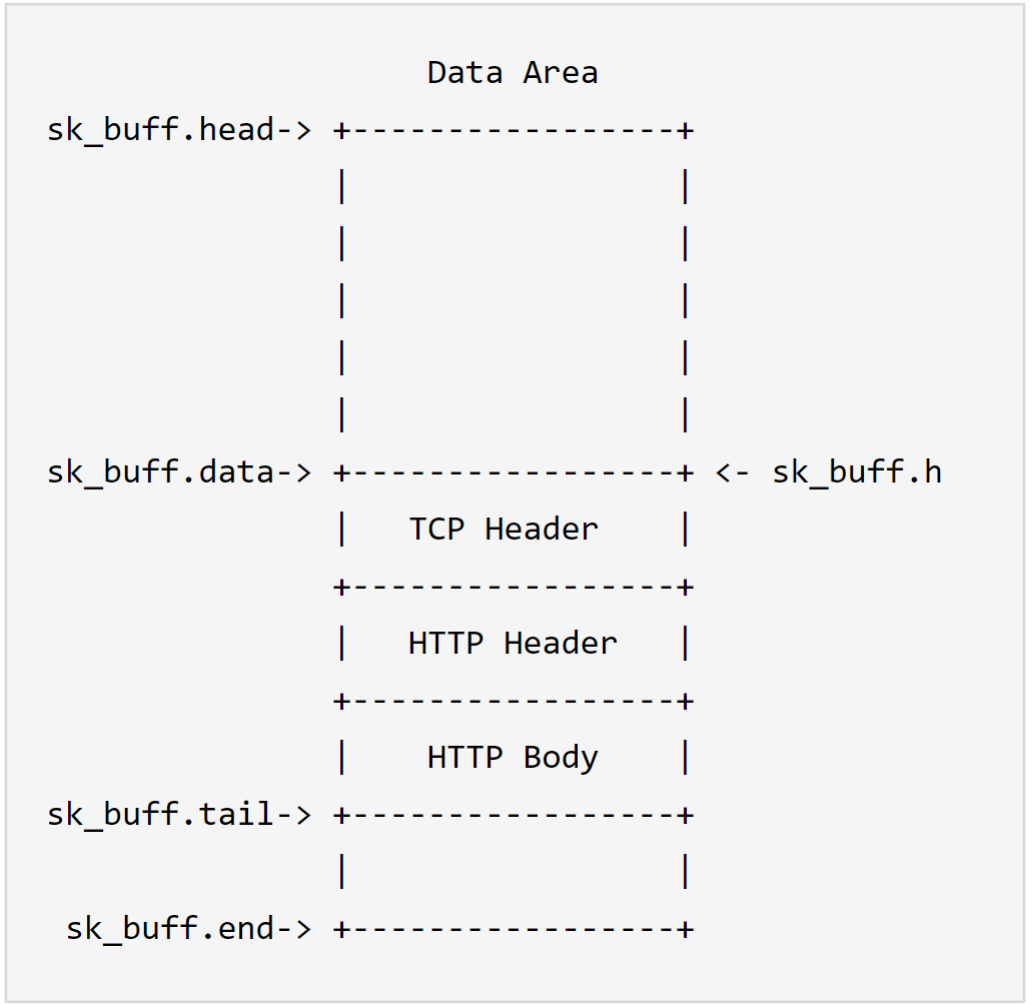
sk_buff的变化过程

- 在TCP报文的发送过程中，sk_buff要依次经历如下变化过程：
  - tcp_sendmsg函数通过skb_put函数调整sk_buff.tail指针，为报文有效载荷划定存储空间，并将其从用户态缓存拷贝到数据区中：



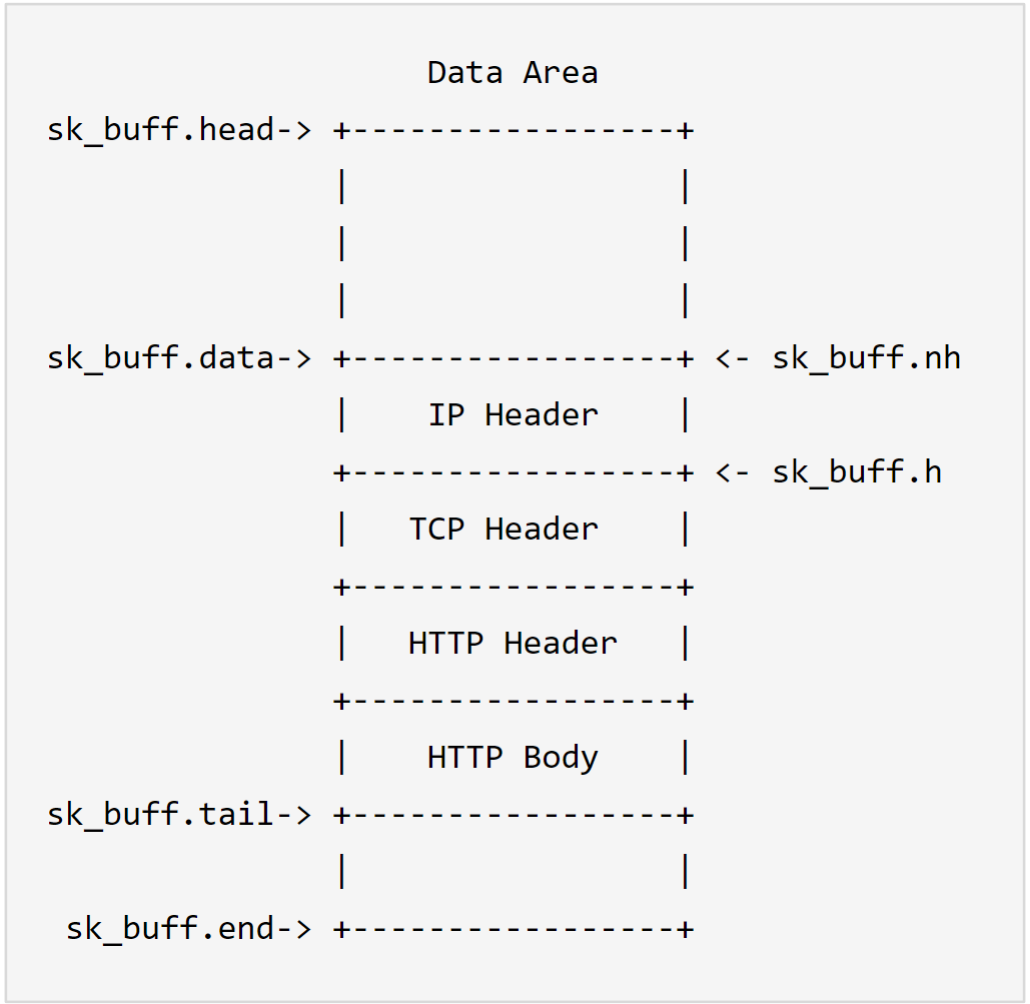
sk_buff的变化过程

- 在TCP报文的发送过程中，sk_buff要依次经历如下变化过程：
  - tcp_transmit_skb函数通过skb_push函数上推sk_buff.data指针，构造TCP包头：



sk_buff的变化过程

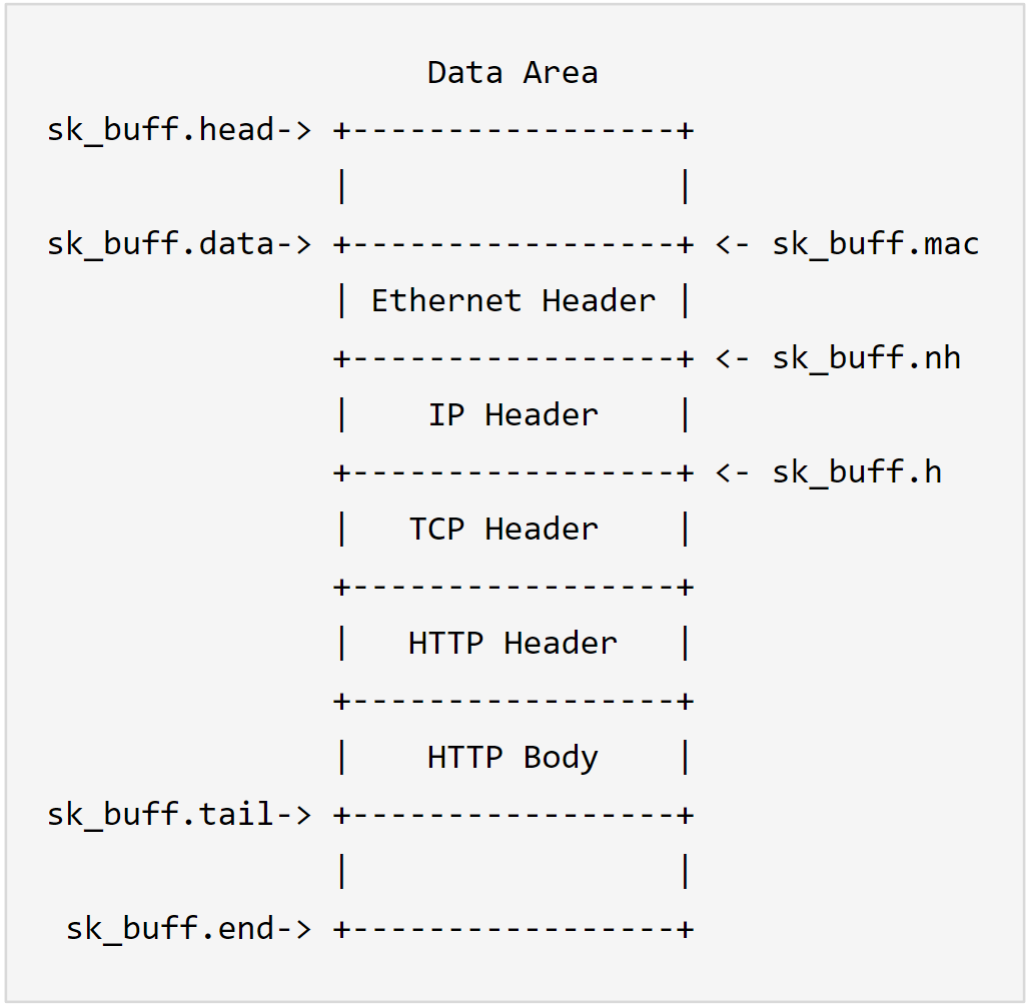
- 在TCP报文的发送过程中，sk_buff要依次经历如下变化过程：
  - ip_queue_xmit函数通过skb_push函数上推sk_buff.data指针，并构造IP包头：



sk_buff的变化过程

- 在TCP报文的发送过程中，sk_buff要依次经历如下变化过程：
  - ip_finish_output2函数通过skb_push函数上推sk_buff.data指针，并构造以太网包头：
  - 最后dev_queue_xmit函数将[data, tail)内的完整以太网包交给网卡驱动

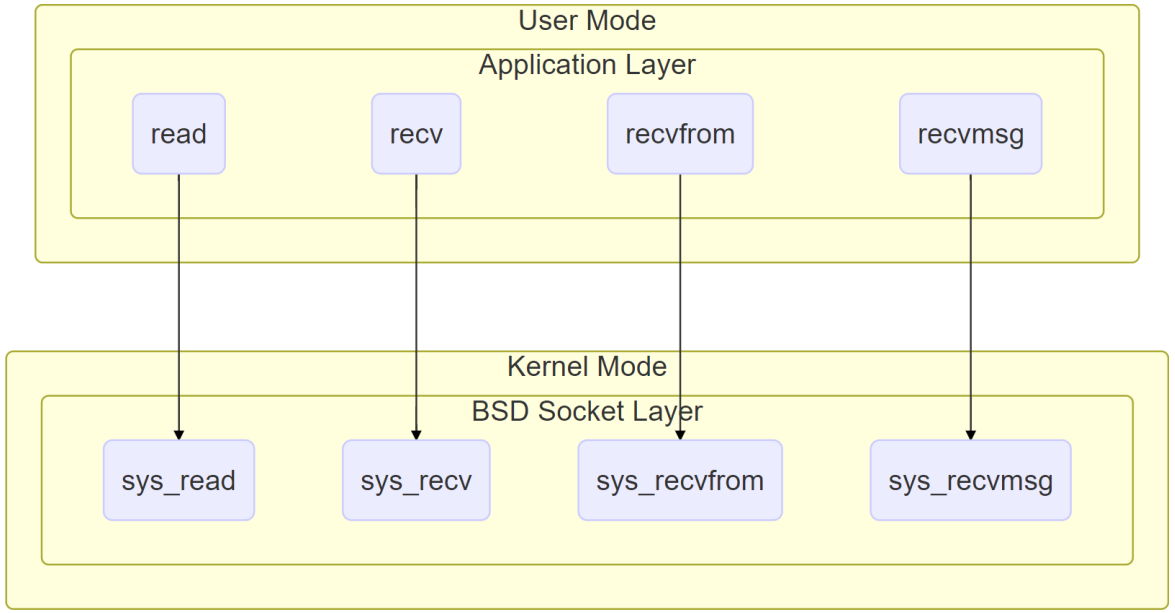
知识讲解



报文接收

套接字函数层(L5-用户态应用层)

- 对于套接字而言，总共有四个函数可用于接收数据：
  - read
  - recv
  - recvfrom
  - recvmsg
- 套接字函数层的四个接收函数都运行于用户态，它们会通过BSD套接字层的四个对应的函数进入内核态。如下图所示：

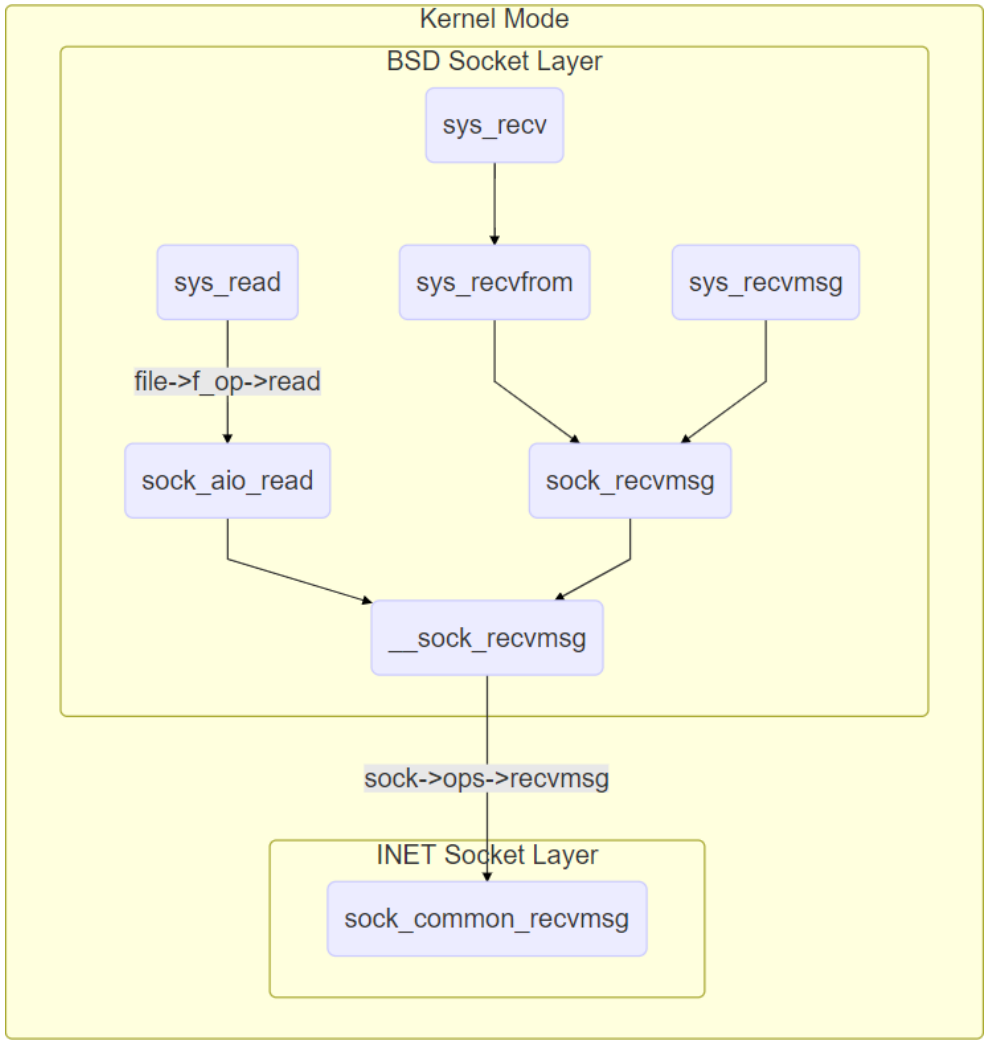


知识讲解

BSD套接字层(L5-内核态应用层)

知识讲解

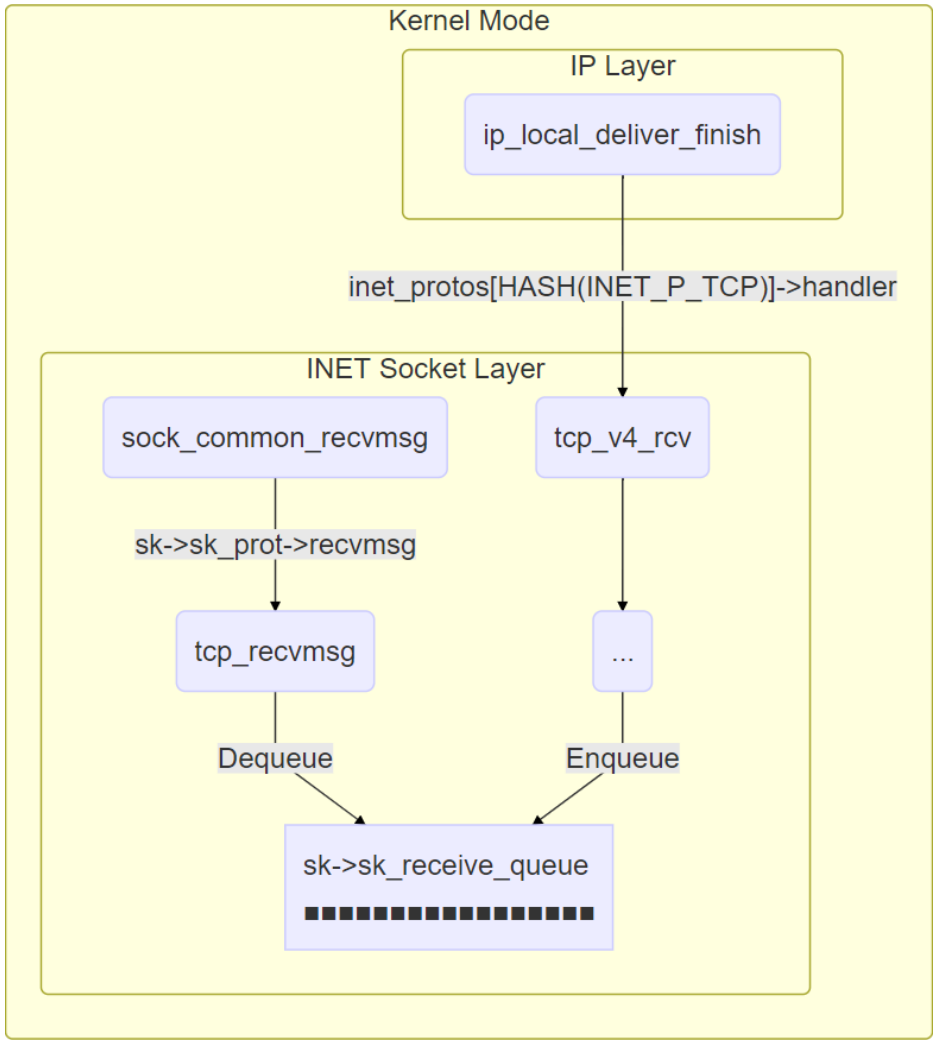
- BSD套接字层所有带sys_前缀的函数都是系统调用。执行系统调用将使程序运行于内核态。四个接收函数最终都会调用__sock_recvmsg函数，该函数通过msghdr结构管理被接收数据。如下图所示：



- 函数指针sock->ops->recvmsg的作用是将BSD套接字接口映射到不同协议族的具体实现上。对于PF_INET协议族，该函数指针指向INET套接字层的sock_common_recvmsg函数。对于其它协议族，如ATM、IPX/SPX等，该函数指针也可指向其它实现函数

INET套接字层(L4-传输层)

- 在INET套接字层，sock_common_recvmsg函数的主要工作是通过函数指针sk->sk_prot->recvmsg调用tcp_recvmsg函数。如果使用其它类型的套接字，如SOCK_DGRAM、SOCK_RAW等，该函数指针也可指向其它实现相应服务的函数。如下图所示：

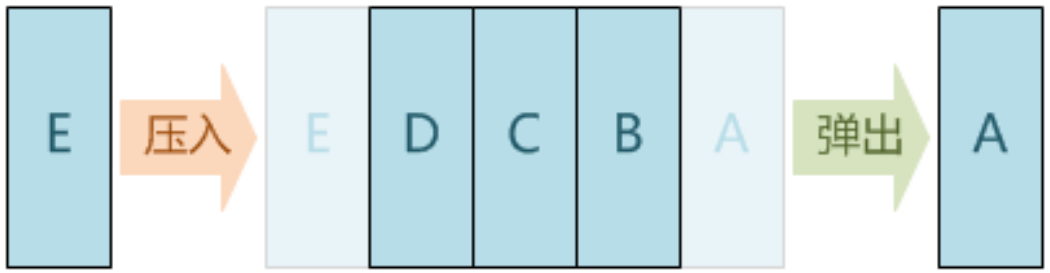


知识讲解

INET套接字层(L4-传输层)

- tcp_recvmsg函数会检查sk->sk_receive_queue指针指向的套接字接收队列：
  - 如果队列中有待接收报文，则将其中的数据复制到用户通过接收函数指定的缓冲区中，立即返回
  - 如果队列中无待接收报文：
    - 对于非阻塞套接字，立即返回
    - 对于阻塞套接字，先使调用线程进入睡眠，直到有新的可接收报文进入队列，将其复制到用户通过接收函数指定的缓冲区中，再唤醒调用线程，使其从接收函数返回

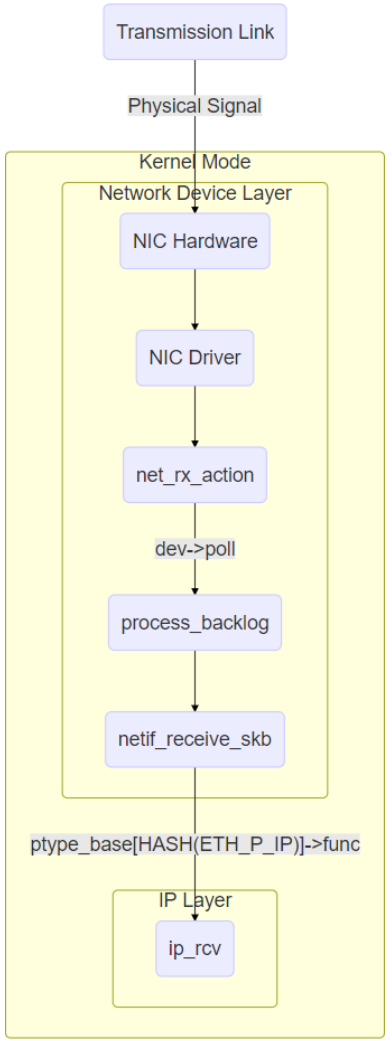
知识讲解



网络设备层(L1-物理层)

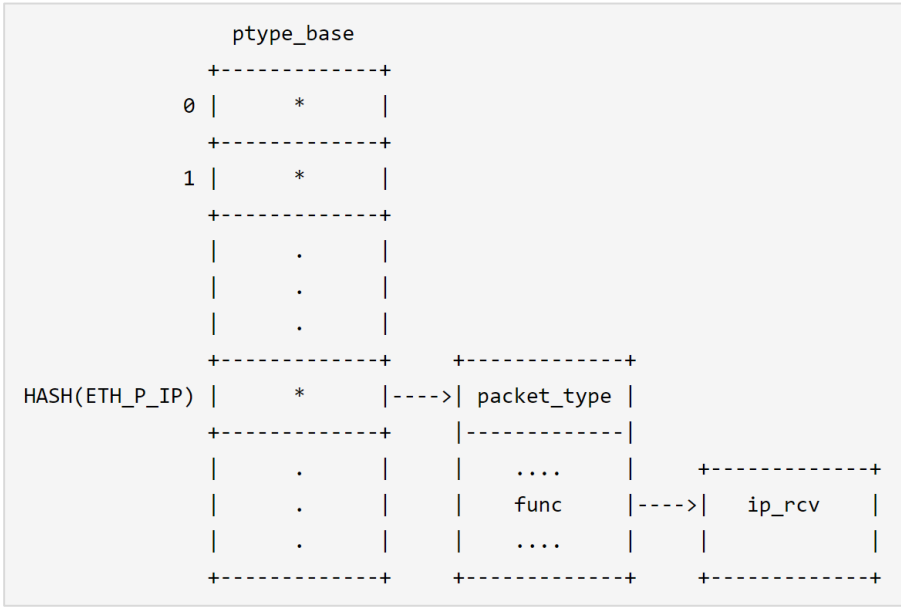
知识讲解

- 以以太网为例，网络传输线路中的物理信号进入网卡硬件，触发特定的硬件中断。网卡驱动会在硬件中断的处理过程中将网卡缓冲区中的报文分组以sk_buff的形式复制到网络协议栈中，同时触发软件中断。对该软件中断的处理，从调用net_rx_action函数开始。如下图所示：
- net_rx_action函数通过函数指针dev->poll调用网络设备轮询函数。每种网络设备都有自己的轮询函数，因此这里使用函数指针来调用。事实上只有最新支持NAPI的网卡才支持轮询操作，其它大部分网卡都是使用内核提供的缺省处理函数process_backlog



网络设备层(L1-物理层)

- 在process_backlog函数中最重要的一步就是调用netif_receive_skb函数，该函数负责处理那些已被网卡接收，但还在等待网络协议栈处理的报文分组
- 以太网包头在网卡驱动中已经处理完毕，因此报文接收过程中无需数据链路层介入，而且netif_receive_skb函数可以从以太网包头中解析出网络层的协议类型，如ETH_P_IP表示IP协议。以该协议类型的哈希值为索引从ptype_base数组中检索得到一个指向packet_type结构的指针，该结构中的func字段是一个函数指针，指向针对该协议的处理函数，如ip_rcv即为针对IP协议的处理函数。调用该函数，即进入IP层。如下图所示：



知识讲解

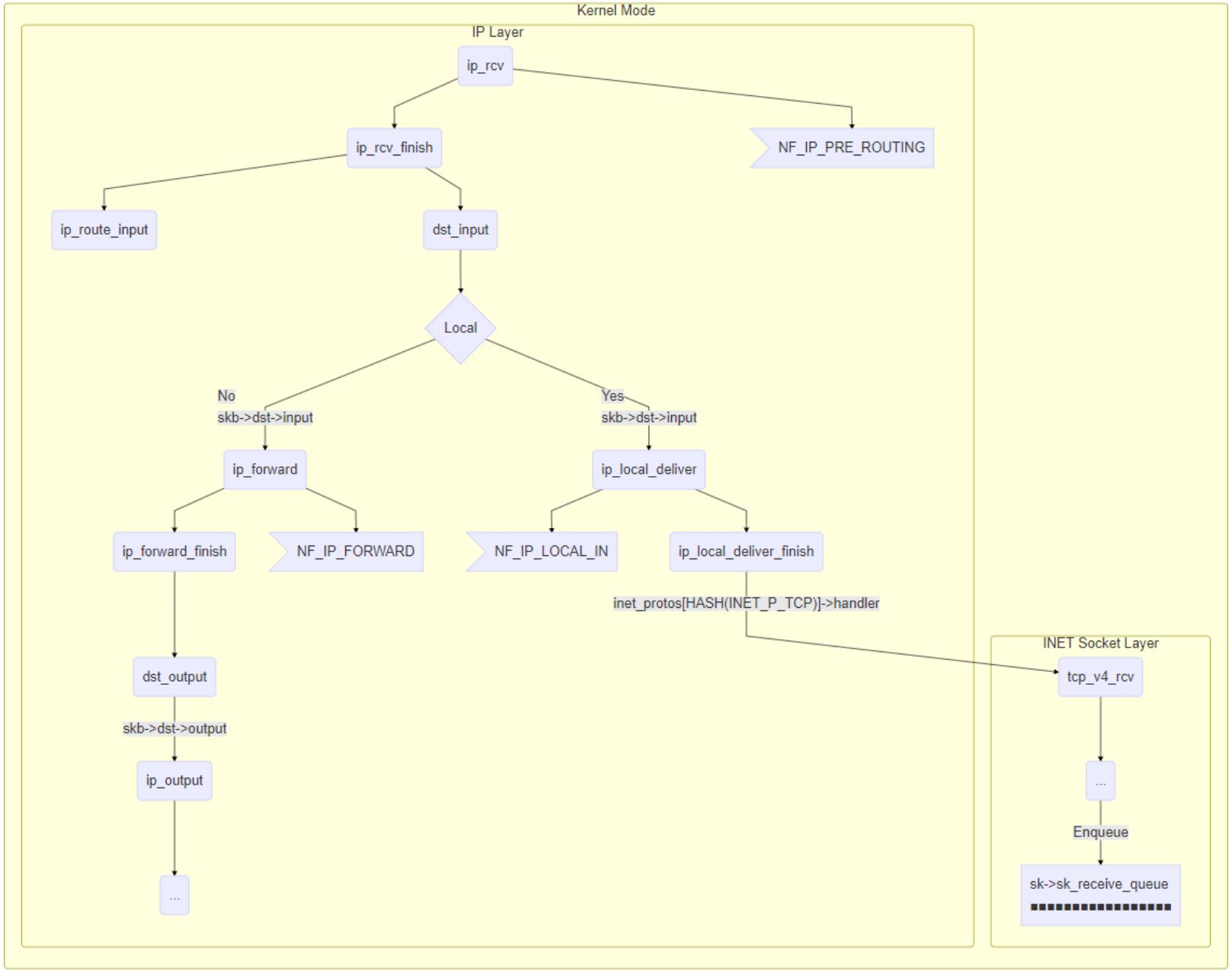
IP层(L3-网络层)

- ip_rcv函数是针对接收到的IP报文进行处理的起点，这里的报文尚未经过路由处理，既可能是转发报文也可能是本机报文，因此Netfilter框架在该函数中设置了钩子函数的挂接点NF_IP_PRE_ROUTING，用来检查主机接收到的所有IP报文
- 在调用完钩子函数以后，ip_rcv函数会调用ip_rcv_finish函数，它的主要工作是通过ip_route_input函数判断所接收的报文到底是转发报文还是本机报文：
  - 转发报文
    - 在skb->dst->input所指向的函数指针链表末尾追加ip_forward函数指针
    - 在skb->dst->output所指向的函数指针链表末尾追加ip_output函数指针
  - 本机报文
    - 在skb->dst->input所指向的函数指针链表末尾追加ip_local_deliver函数指针

知识讲解

IP层(L3-网络层)

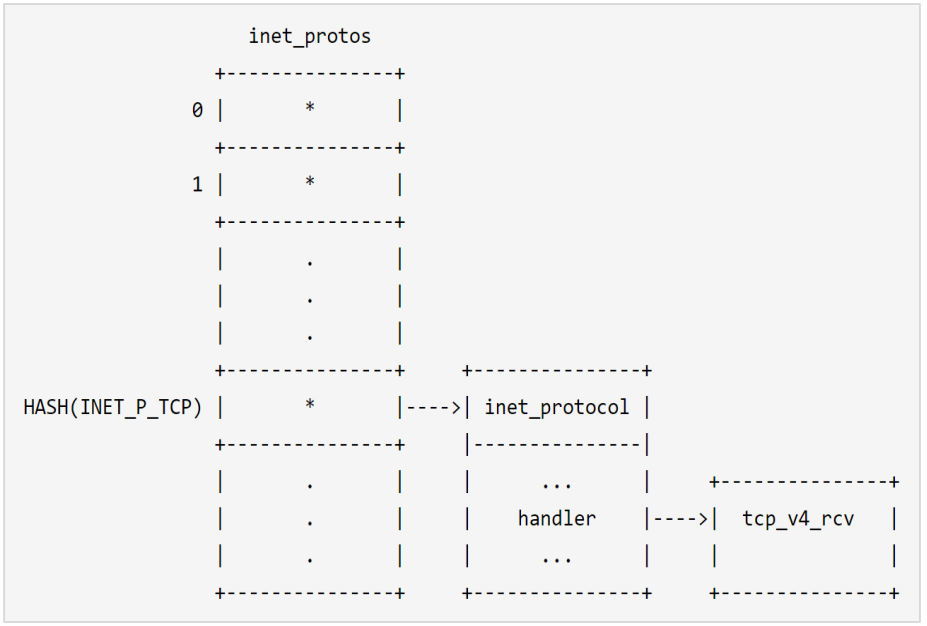
- dst_input函数负责遍历skb->dst->input所指向的函数指针链表，依次调用其中每个函数指针所指向的函数，最后调用ip_forward函数(转发报文)或者ip_local_deliver函数(本机报文)。如下图所示：



知识讲解

IP层(L3-网络层)

- ip_forward函数在执行完用于监控转发报文的NF_IP_FORWARD钩子以后，会依次调用ip_forward_finish函数和dst_output函数，而后者又会通过skb->dst->output指向的函数指针链表调用ip_output函数，汇入报文发送流程
- ip_local_deliver函数在执行完用于监控本机报文的NF_IP_LOCAL_IN钩子以后，会调用ip_local_deliver_finish函数。该函数可以从IP包头中解析出传输层的协议类型，如INET_P_TCP表示TCP协议。以该协议类型的哈希值为索引从inet_protos数组中检索得到一个指向inet_protocol结构的指针，该结构中的handler字段是一个函数指针，指向针对该协议的处理函数，如tcp_v4_rcv即为针对TCP协议的处理函数。如下图所示：

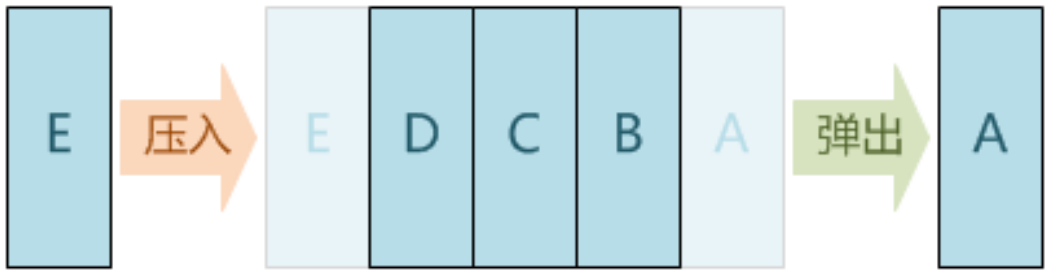


知识讲解

IP层(L3-网络层)

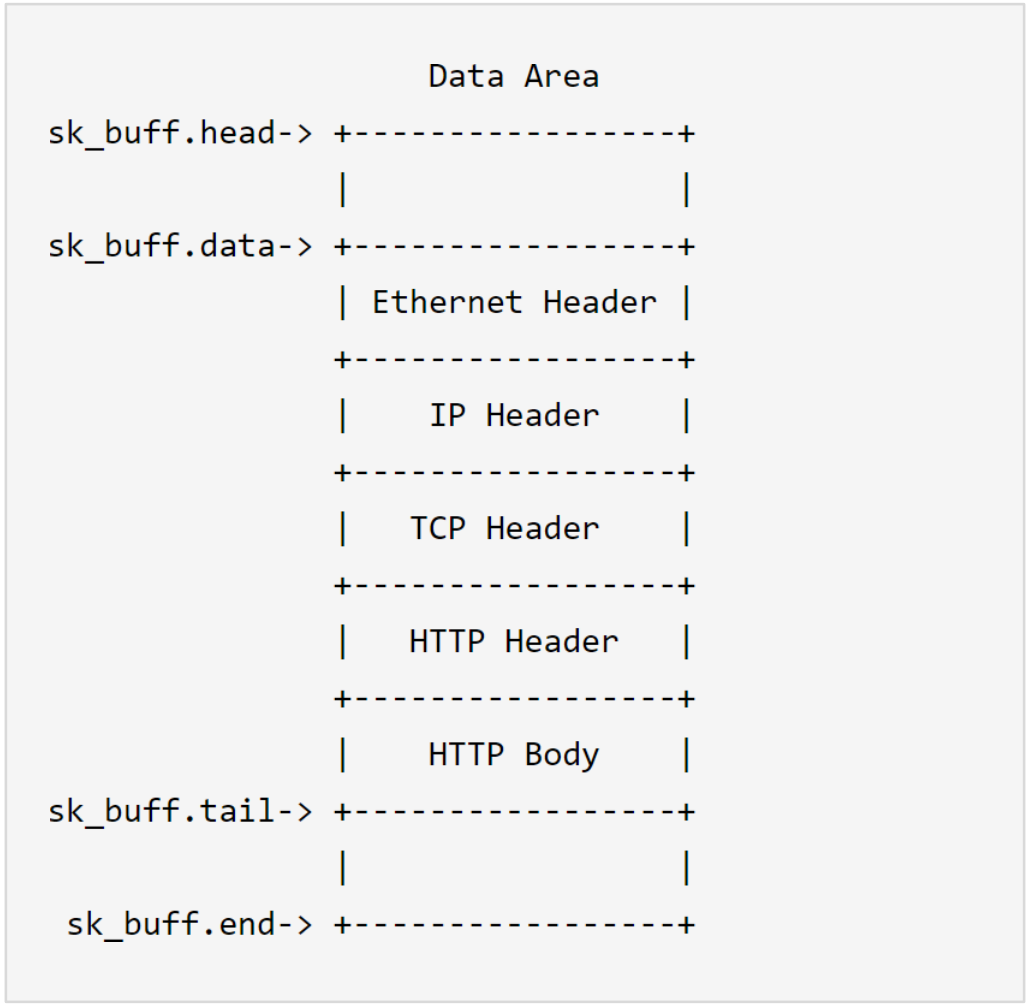
- tcp_v4_rcv函数是针对接收到的本机TCP报文进行处理的起点。该函数通过对IP包头和TCP包头的解析构建由源IP地址、源端口号、目的IP地址和目的端口号组成的四元组，据此找到建立TCP连接时所创建的套接字，将其sock结构的指针赋予表示该报文的sk_buff中的sk字段，并将该sk_buff加入到由sk->sk_receive_queue指针所指向的套接字接收队列中。鉴于TCP协议处理过程的复杂性，具体实现细节在上图中用省略号表示

知识讲解



sk_buff的变化过程

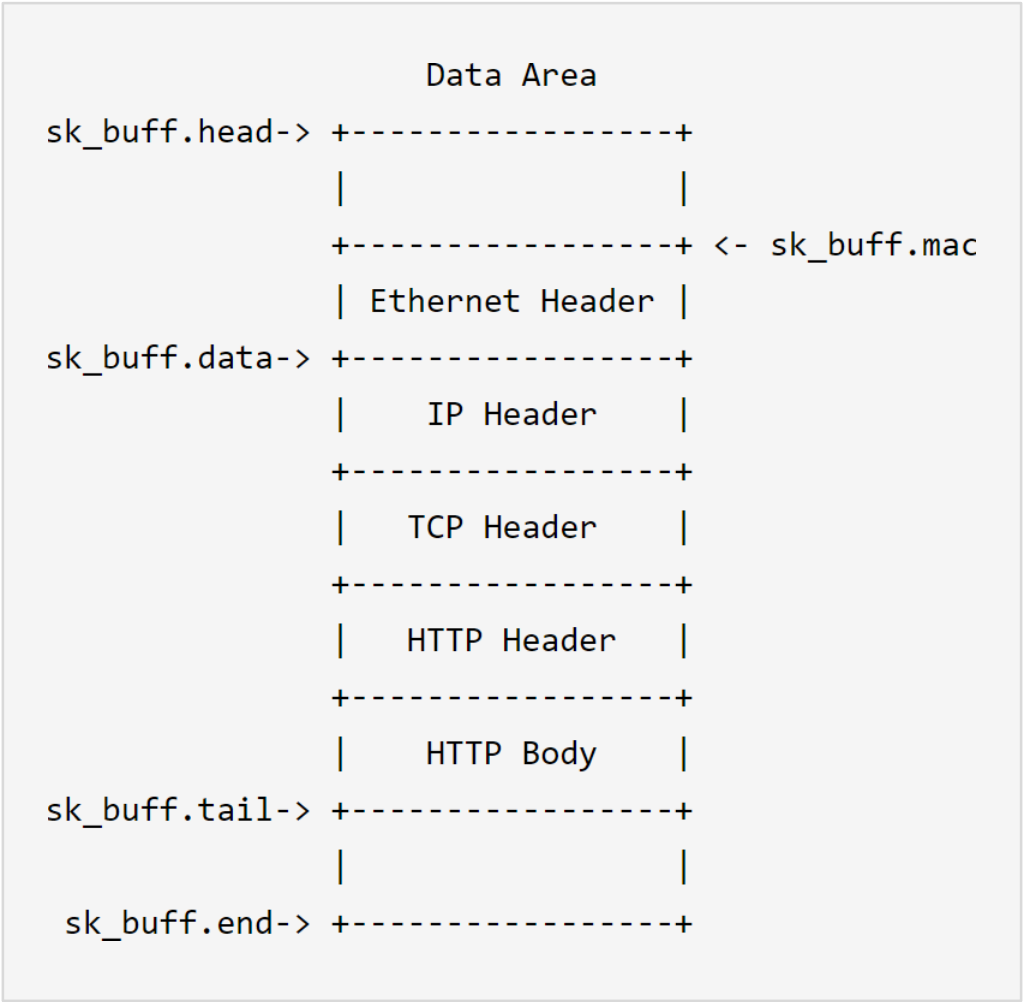
- 在TCP报文的接收过程中，sk_buff要依次经历如下变化过程：
  - 网卡硬件从网络传输线路中接收到物理信号并形成网卡缓冲区中的报文分组，网卡驱动在硬件中断的处理过程中将该报文分组以sk_buff的形式复制到网络协议栈中：



知识讲解

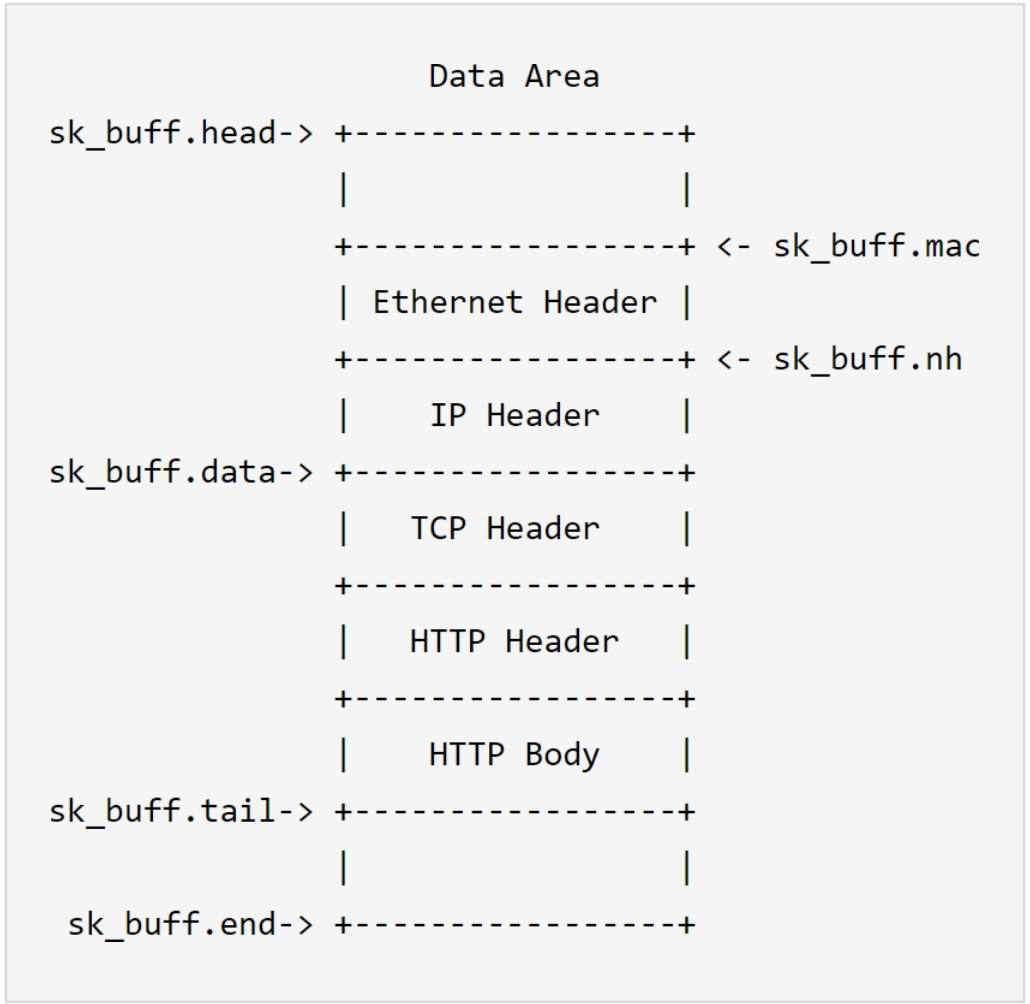
sk_buff的变化过程

- 在TCP报文的接收过程中，sk_buff要依次经历如下变化过程：
  - 网卡驱动通过eth_type_trans函数从报文分组中解析出以太网包头，并通过skb_pull函数下拉sk_buff.data指针：



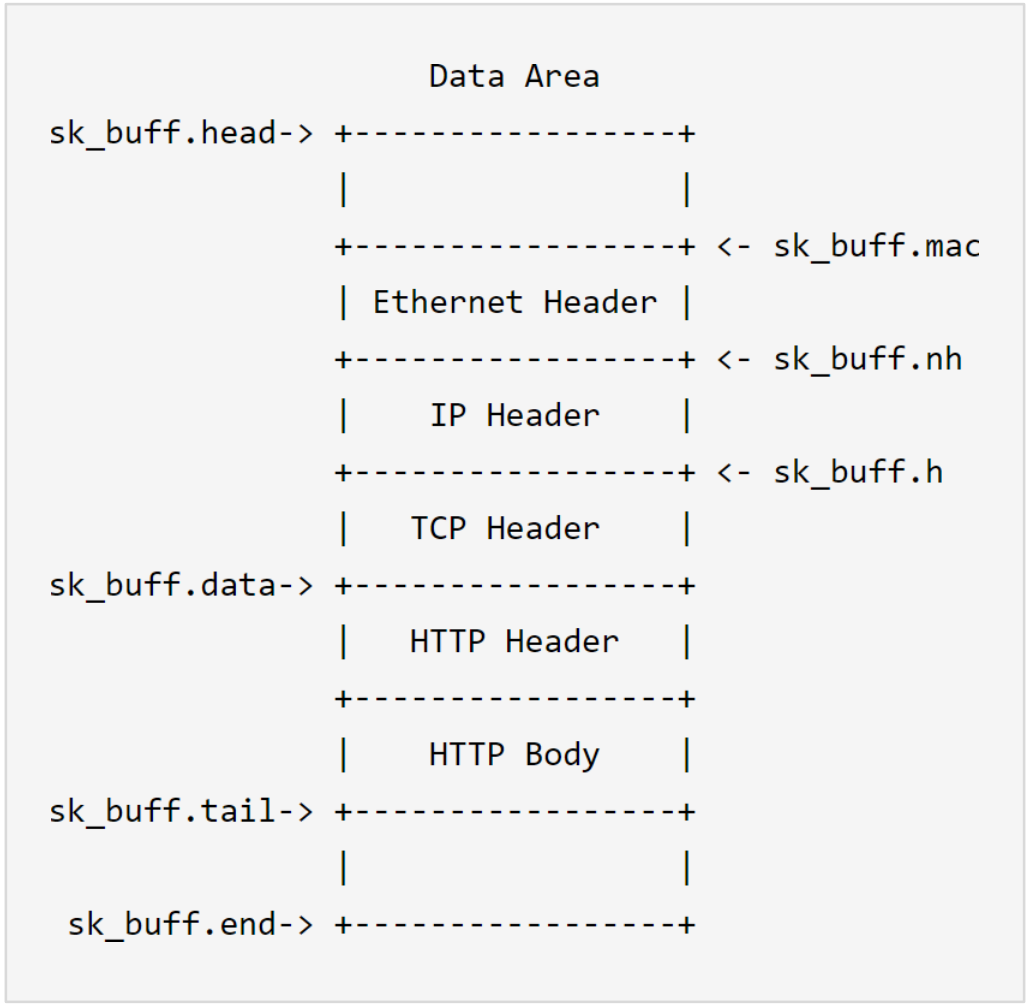
sk_buff的变化过程

- 在TCP报文的接收过程中，sk_buff要依次经历如下变化过程：
  - ip_local_deliver函数负责本机报文IP包头的解析，并通过skb_pull函数下拉sk_buff.data指针：



sk_buff的变化过程

- 在TCP报文的接收过程中，sk_buff要依次经历如下变化过程：
  - tcp_v4_rcv函数负责TCP包头的解析，并通过skb_pull函数下拉sk_buff.data指针：
  - 当表示接收报文的sk_buff被加入到套接字接收队列中时，其data字段刚好指向报文中的有效载荷部分。tcp_recvmsg函数负责将接收报文中的有效载荷拷贝到用户程序的缓冲区中，然后清除该sk_buff结构



总结和答疑