第16章 网 络 系 统

网络和Linux系统几乎可以说是同义词，因为Linux系统是WWW的产物。下面我们讨论一下Linux系统是如何支持TCP/IP协议的。

TCP/IP协议最初用来支持计算机和ARPANET网络之间通信。现在广泛使用的World Wide Web就是从ARPANET中发展来的，并且World Wide Web使用的也是TCP/IP协议。在UNIX系统中，首先带有网络功能的版本是4.3 BSD。Linux系统的网络功能就是以UNIX 4.3 BSD为模型发展起来的，它支持BSD套接口和全部的TCP/IP功能。这样UNIX系统中的软件就可以十分方便地移植到Linux 系统中了。

16.1 TCP/IP 网络简介

现在简单地介绍TCP/IP网络的主要原理。

在一个IP(Internet Protocol)网络中，每一台计算机都有一个32位的IP地址。每台计算机的IP地址都是唯一的。WWW是一个范围十分大，并且不断增长的IP网络，所以网络上的每台计算机都必须有一个唯一的IP地址。IP地址是用 . 分隔开的4个十进制数，例如16.42.0.9。实际上IP地址可以分为两部分：一部分是网络地址，另一部分是主机地址，例如，在 16.42.0.9中，16.42是网络地址，0.9则为主机地址。而主机地址又可以分为子网地址和主机地址。计算机的IP地址很不容易记忆，如果使用一个名字就可以方便得多。如果使用名字，则必须有某一种机制将名字转化为IP地址。这些名字可以静态地保存在 /etc/hosts文件中，或者Linux系统请求域名服务器（DNS服务器）来转换名字。如果使用DNS服务器的话，本地的主机则必须知道一个或者多个DNS服务器的IP地址，这些信息保存在/etc/resolv.conf文件中。

当你和其他计算机相连时，系统要使用IP地址和其他计算机交换数据。数据保存在IP数据包中。每一个IP数据包都有一个IP数据头，其中包括源地址和目的地址，一个数据校验和以及其他一些有关的信息。IP数据包的大小随传输介质的不同而不同，例如，以太网的数据包要大于PPP的数据包。目的地址的主机在接收数据包后，必须再将数据装配起来，然后传送给接收的应用程序。

连接在同一个IP子网上的主机之间可以直接传送IP数据包，而在不同子网之间的主机却要使用网关。网关用来在不同的子网之间传送数据包。

IP协议是一个传输层的协议，其他的协议可以利用 IP协议来传输数据。TCP(Transmission Control Protocol)协议是一个可靠的点到点之间的协议，它使用IP协议来传送和接收自己的数据包，如图16-1所示。TCP协议是基于连接的协议。需要通信的两个应用程序之间将建立起一条虚拟的连接线路，即使其中要经过很多子网、网关和路由器。 TCP协议保证在两个应用程序之间可靠地传送和接收数据，并且可以保证没有丢失的或者重复的数据包。当TCP协议使用IP协议传送它自己的数据包时，IP数据包中的数据就是TCP数据包本身。相互通信的主机中的IP协议层负责传送和接收IP数据包。每一个IP数据头中都包括一个字节的协议标识符。当TCP协议请求IP协议层传送一个IP数据包时，IP数据头中的协议标识符指明其中的数据包是一个TCP数据包。接收端的IP层则可以使用此协议标识符来决定将接收到的数据包传送到那一层, 在这里是TCP协议层。当应用程序使用TCP/IP通信时，它们不仅要指明目标计算机的IP地址，也要指明应用程序使用

的端口地址。端口地址可以唯一地表示一个应用程序，标准的网络应用程序使用标准的端口地址，例如，web服务器使用端口80。你可以在/etc/services中查看已经登记的端口地址。

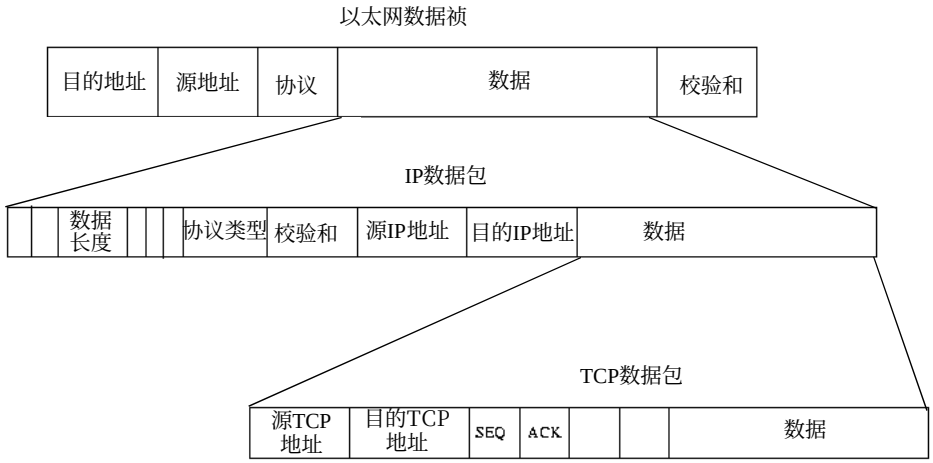

图16-1 网络协议示意图

IP 协议层也可以使用不同的物理介质来传送IP数据包到其他的IP地址主机。这些介质可以自己添加协议头。例如以太网协议层、 PPP协议层或者SLIP协议层。以太网可以同时连接很多个主机，每一个主机上都有一个以太网的地址。这个地址是唯一的，并且保存在以太网卡中。所以在以太网上传输IP数据包时，必须将IP数据包中的IP地址转换成主机的以太网卡中的物理地址。Linux系统使用地址解决协议（ARP）来把IP地址翻译成主机以太网卡中的物理地址。希望把IP地址翻译成硬件地址的主机使用广播地址向网络中的所有节点发送一个包括 IP地址的ARP请求数据包。拥有此 IP地址的目的计算机接收到请求以后，返回一个包括其物理地址的ARP应答。ARP协议不仅仅限于以太网，它还可以用于其他的物理介质，例如 FDDI等。那些不能使用ARP的网络设备可以标记出来，这样 Linux系统就不会试图使用 ARP。系统中也有一个反向的翻译协议，叫做RARP，用来将主机的物理地址翻译成 IP地址。网关可以使用此协议来代表远程网络中的IP地址回应ARP请求。

16.2 TCP/IP网络的分层

图16-2显示了Linux系统网络实现的分层结构。 BSD 套接口是最早的网络通信的实现，它由一个只处理BSD 套接口的管理软件支持。其下面是 INET套接口层，它管理TCP协议和UDP协议的通信末端。UDP(User Datagram Protocol)是无连接的协议，而TCP则是一个可靠的端到端协议。当网络中传送一个 UDP数据包时，Linux系统不知道也不关心这些 UDP数据包是否安全地到达目的节点。TCP数据包是编号的，同时TCP传输的两端都要确认数据包的正确性。 IP协议层是用来实现网间协议的，其中的代码要为上一层数据准备 IP数据头，并且要决定如何把接收到的IP数据包传送到TCP协议层或者UDP协议层。在IP协议层的下方是支持整个 Linux 网络系统的网络设备，例如 PPP和以太网。网络设备并不完全等同于物理设备，因为一些网络设备，例如回馈设备是完全由软件实现的。和其他那些使用 mknod命令创建的Linux系统的标准设备不同，网络设备只有在软件检测到和初始化这些设备时才在系统中出现。当你构建系统内核时，即使系统中有相应的以太网设备驱动程序，你也只能看到 /dev/eth0。ARP协议在IP协议

层和支持ARP翻译地址的协议之间。

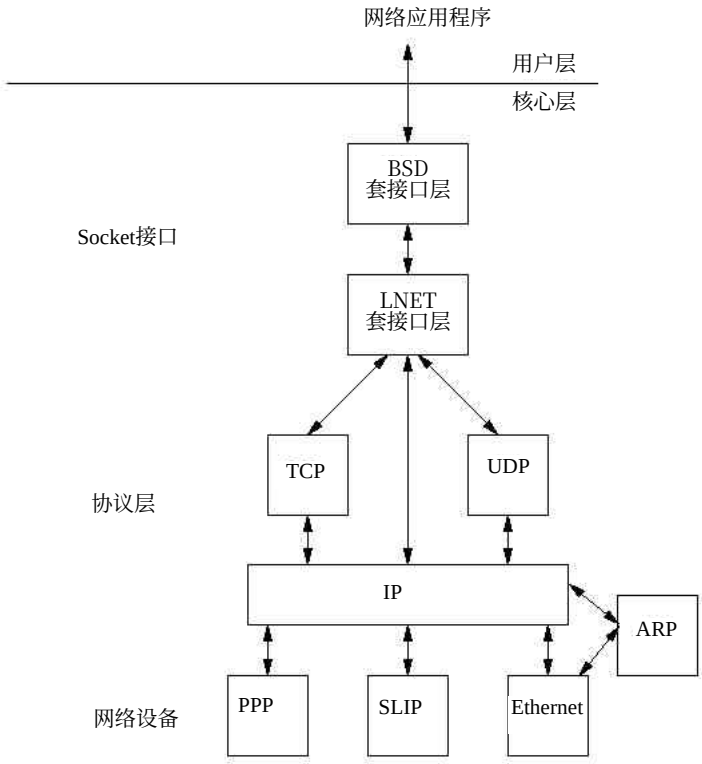

图16-2 TCP/ IP结构分层示意图

16.3 BSD套接口

BSD是UNIX系统中通用的网络接口，它不仅支持各种不同的网络类型，而且也是一种内部进程之间的通信机制。两个通信进程都用一个套接口来描述通信链路的两端。套接口可以认为是一种特殊的管道，但和管道不同的是，套接口对于可以容纳的数据的大小没有限制。Linux支持多种类型的套接口，也叫做套接口寻址族，这是因为每种类型的套接口都有自己的寻址方法。Linux支持以下的套接口类型：

UNIX	UNIX域套接口
INET	Internet地址族TCP/IP协议支持通信。
AX25	Amateur radio X25
IPX	Novell IPX
APPLETALK	Appletalk DDP
X25	X25

这些类型的套接口代表各种不同的连接服务。

Linux的BSD 套接口支持下面的几种套接口类型：

1. 流式（stream）

这些套接口提供了可靠的双向顺序数据流连接。它们可以保证数据传输中的完整性、正确

性和单一性。INET寻址族中的TCP协议支持这种类型的套接口。

2. 数据报（Datagram）

这种类型的套接口也可以像流式套接口一样提供双向的数据传输，但它们不能保证传输的数据一定能够到达目的节点。即使数据能够到达，也无法保证数据以正确的顺序到达以及数据的单一性、正确性。UDP协议支持这种类型的套接口。

3. 原始（Raw）

这种类型的套接口允许进程直接存取下层的协议。

4. 可靠递送消息（Reliable Delivered Messages）

这种套接口和数据报套接口一样，只能保证数据的到达。

5. 顺序数据包（Sequenced Packets）

这种套接口和流式套接口相同，除了数据包的大小是固定的。

6. 数据包（Packet）

这不是标准的BSD 套接口类型，而是Linux 中的一种扩展。它允许进程直接存取设备层的数据包。

利用套接口进行通信的进程使用的是客户机 /服务器模式。服务器用来提供服务，而客户机可以使用服务器提供的服务，就像一个提供 web页服务的Web服务器和一个读取并浏览 web页的浏览器。服务器首先创建一个套接口，然后给它指定一个名字。名字的形式取决于套接口的地址族，事实上也就是服务器的当地地址。系统使用数据结构 sockaddr来指定套接口的名字和地址。一个INET 套接口可以包括一个IP端口地址。你可以在/etc/services中查看已经注册的端口号，例如，一个web页面服务器的端口号是80。在服务器指定套接口的地址以后，它将监听和此地址有关的连接请求。请求的发起者，也就是客户机，将会创建一个套接口，然后再创建连接请求，并指定服务器的目的地址。对于一个 INET 套接口来说，服务器的地址就是它的IP地址和端口号。这些连接请求必须通过各种协议层，然后等待服务器的监听套接口。一旦服务器接收到了连接请求，它将接受或者拒绝这个请求。如果服务器接受了连接请求，它将创建一个新的套接口。一旦服务器使用一个套接口来监听连接请求，它就不能使用同样的套接口来支持连接。当连接建立起来以后，连接的两端都可以发送和接收数据。最后，当不再需要此连接时，可以关闭此连接。

使用BSD套接口的确切含义在于套接口所使用的地址族。设置一个 TCP/IP连接就和设置一个业余无线电X.25连接有很大的不同。和VFS一样，Linux从BSD 套接口协议层中抽象出了套接口界面，此界面负责和各种不同的应用程序之间进行通信。内核初始化时，内核中的各个不同的地址族将会在BSD 套接口界面中登记。稍后当应用程序创建和使用 BSD 套接口时，就将会在BSD 套接口和它支持的地址族之间建立一个连接。此连接是通过交叉关联的数据结构和地址族表建立的。例如，当一个应用程序创建一个新的套接口时，将产生一个可以被 BSD 套接口使用的与特定的地址族有关的套接口创建子过程。

设置系统内核时，一系列的地址族和协议将会保存在协议向量中。每一个协议都由它的名字代表，例如，INET和其初始化进程的地址。当系统启动并初始化套接口界面时，将会调用每一个协议的初始化进程。对于套接口地址族来说，这意味着它们注册的一系列有关协议操作。这是一系列的子程序，每一个都执行一个和特定的地址族有关的操作。已经注册的和协议相关的操作保存在pops向量中，而此向量由一系列指向数据结构proto_ops的指针组成。

数据结构proto_ops包括地址族的类型以及指向与特定地址族有关的套接口操作程序的指针。Pops向量用地址族标识符作为索引。

16.4 INET套接口层

INET 套接口层包括支持TCP/IP协议的Internet地址族。正如上面提到的，这些协议是分层的，每一个协议都使用另一个协议的服务。 Linux系统中的TCP/IP代码和数据结构也反映了这种分层的思想。它和BSD 套接口层的接口是通过一系列与 Internet地址族有关的套接口操作来实现的，而这些套接口操作是在网络初始化的过程中由 INET 套接口层在BSD 套接口层中注册的。这些操作和其他地址族的操作一样保存在 pops向量中。 BSD 套接口层通过 INET的proto_ops数据结构来调用与INET 层有关的套接口子程序来实现有关INET层的服务。例如，当BSD 套接口创建一个发送给INET地址族的请求时将会使用INET的套接口创建功能。BSD 套接口层将会把套接口数据结构传递给每一个操作中的 INET层。INET 套接口层在它自己的数据结构sock中而不是在BSD 套接口的数据结构中插入有关TCP/IP的信息，但sock数据结构是和BSD套接口的数据结构有关的，通过图16 - 3可以看出这种相关关系。它使用 BSD 套接口中的数据指针来连接sock数据结构和BSD 套接口数据结构，这意味着以后的 INET 套接口调用可以十分方便地得到sock数据结构。数据结构sock中的协议操作指针也会在创建时设置好，并且此指针是和所需要的协议有关的。如果需要的是 TCP协议，那么数据结构sock中的协议操作指针将会

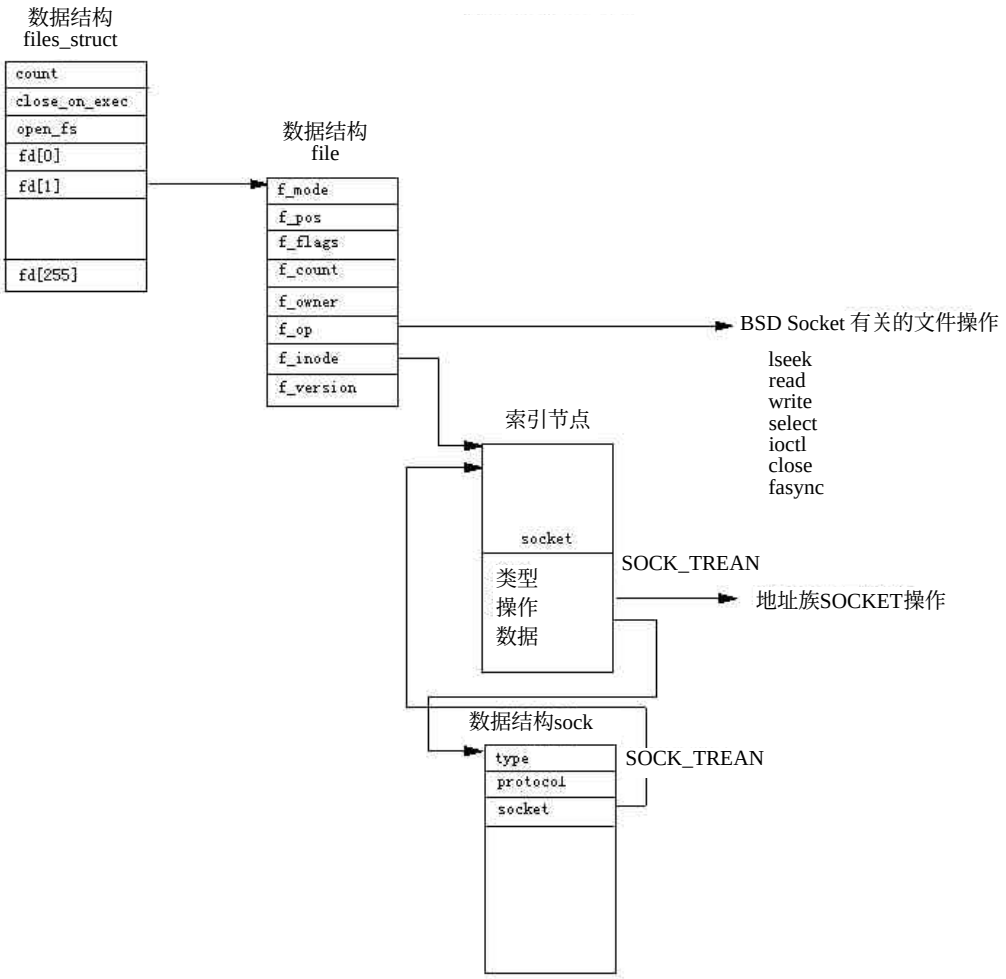

图16-3 Socket 数据结构示意图

指向一系列的TCP协议操作。

16.4.1 创建BSD 套接口

创建一个新套接口的系统调用将传递有关地址族、套接口类型和协议的标识符。

首先，系统调用使用需要的地址族来搜索pops向量以便找到一个匹配的地址族。也许某一个地址族是作为系统内核模块实现的，这时kerneld守护进程将会调入此模块以便继续运行。实际上，套接口数据结构物理上是VFS索引节点数据结构的一部分，所以分配一个套接口就等于分配一个VFS索引节点，这样就可以和使用普通文件一样来操作套接口。因为所有的文件都有一个VFS索引节点，故为了和文件操作之间的相互兼容，BSD 套接口也有一个代表它的VFS索引节点。

新创建的BSD 套接口数据结构包括一个指向特定地址族的套接口子过程的指针，它被设置为指向数据结构proto_ops,该数据结构从pops向量中获取。BSD 套接口数据结构的类型为SOCK_STREAM、SOCK_DGRAM或其他的套接口类型。与特定的地址族相关的创建进程通过保存在数据结构proto_ops中的地址来调用。

系统将会从当前进程的fd向量中分配一个空闲的文件描述符，同时也将初始化它所指向的文件的数据结构。这包括将文件操作指针设置到 BSD 套接口支持的文件操作中。任何以后的操作都将使用套接口，同时套接口将会通过调用相应的地址族操作子程序把这些操作传递到所支持的地址族中。

16.4.2 给INET BSD 套接口指定地址

为了能够监听到网络中的连接请求，每一个服务器都必须创建一个 INET BSD 套接口，并且指定一个地址。指定地址的操作一般都在 INET 套接口层中处理，但可能需要TCP层和UDP层的支持。已经指定了地址的套接口不能用于其他任何的通信，这意味着套接口的状态必须是TCP_CLOSE。传递给指定地址操作的sockaddr包括指定的IP地址和一个可选的端口号。一般情况下，指定的IP地址将会和指定给网络设备的IP地址相同，并且此网络设备支持INET地址族。你可以使用ifconfig命令查看系统中当前活动的网络接口。IP地址也可以是全为1或者全为0的IP广播地址。如果机器是作为一个透明的代理服务器或者防火墙，那么它就可以指定为任意的 IP地址，但只有具有超级用户优先权的进程可以指定它的 IP地址。指定的IP地址保存在数据结构sock中的recv_addr 和 saddr字段中，其中一个用于散列表的查找，另一个用于发送 IP地址。端口号是可选的，并且如果没有指定端口号的话，系统将要求网络选择一个空闲的端口号。按照惯例，没有超级用户权限的进程不能使用小于1024的端口号。如果网络确实分配了一个端口号，那么通常是大于1024的。

当系统中的网络设备接收到数据包时，这些数据包必须传送到正确的INET 和 BSD 套接口，以便进行下一步处理。正因为这样，UDP层和 TCP层提供一个用于查找接收的IP信息的地址的散列表，并且将数据包发送到正确的 socket/sock对中。TCP是以连接为导向的协议，所以在处理TCP数据包时要比处理UDP数据包涉及更多的信息。

UDP层中也有一个保存已分配的UDP端口的散列表 —— udp_散列表，表中包含指向sock数据结构的指针，并按照端口号作为索引。因为 UDP的散列表要比可以使用的端口数目少很多，所以散列表中的一些入口指向由多个数据结构sock组成的链表。

TCP层的情况较为复杂，因为它提供了几个散列表。尽管如此，TCP层在指定地址操作时也并不将数据结构sock添加到它的散列表中，而只是仅仅检查要求的端口号是否已经被占用。

数据结构sock反在监听操作时才被添加到TCP的散列表中。

16.4.3 在INET BSD套接口上创建连接

系统创建了一个套接口以后，如果此套接口没有用于监听进入的连接请求，那么它就可以用于向外发送连接请求。对于无连接的协议，例如 UDP，这种套接口操作并无太大的作用。但对于已连接为导向的协议，例如 TCP，套接口操作将在两个应用程序之间建立一个虚拟的链路。

INET BSD 套接口只有处于正确的状态时才能用于建立向外的连接，也就是说，此套接口没有用于连接，也没有用于监听进入的连接。这意味着 BSD 套接口数据结构必须处于SS_UNCONNECTED状态。UDP协议不需要在两个应用程序之间建立虚拟连接，在此协议中信息是以数据报的形式传送的。但它同样支持连接的 BSD 套接口操作。UDP INET BSD 套接口上的连接操作仅仅只是设置远程应用程序的地址，即它的 IP地址和IP端口地址。另外，它还建立一个路由表入口缓冲区，这样发送到此 BSD 套接口的UDP数据包则无须再次检查路由数据库。INET sock数据结构中的ip_route_cache指针指向缓冲区的路由信息。如果没有指定地址信息，此缓冲区路由和IP地址信息将会自动用于使用此BSD 套接口发送的信息，同时UDP的状态变为TCP_ESTABLISHED。

对于一个在TCP BSD 套接口上的连接操作，TCP必须创建一个包含连接信息的TCP信息块，并把它发送到指定的IP目的地址。TCP信息块包含关于连接的信息，包括一个唯一的起始信息序列号、信息的最大长度以及传送和接收的窗口大小等等。在 TCP协议中，所有的信息都是按顺序排好的，其中的起始序列号用于识别第一个信息块。 Linux系统会选择一个合理的随机值来避免恶意的协议的攻击。从TCP连接的一端传送的信息要经过连接的另一端的确认，这样才能保证信息传送的正确性，没有经过确认的信息将重发。发送和接收窗口的大小等于没有发送确认信息之前的信息个数。信息的最大长度是由用于初始化端的网络设备决定的。如果接收端的网络设备支持更小的信息长度，那么它们之间的连接将会使用两者之间较小的数值。创建向外的TCP连接请求的应用程序必须等待目标程序接收或者拒绝连接请求的回应。因为 TCP sock处于等待信息进入的状态，它将会添加到 tcp_listening_散列表中以便进入的TCP信息可以直接发送到此sock 数据结构中。TCP协议也将启动一个计时器，如果目标程序没有回应连接请求的话，向外的连接请求将会自动失效。

16.4.4 监听INET BSD 套接口

给套接口指定了地址以后，它将监听此指定地址的进入的连接请求。网络应用程序可以使用某一个套接口而不必首先指定一个地址，在这种情况下， INET 套接口层将会自动为套接口指定一个空闲的端口号。监听功能使得套接口进入到 TCP_LISTEN状态，同时做一些与特定的网络有关的允许进入连接的工作。

对于UDP 套接口来说，只须改变它的状态就已经足够了。但 TCP需要把套接口的sock 数据结构添加到tcp_bound_hash 和 tcp_listening_散列表中。

对于一个处于活动状态的正在监听的套接口来说，每当接收一个进入的 TCP连接请求，TCP都将建立一个新的 sock数据结构来代表此连接请求。它还将复制进入的包含连接信息的sk_buff，并将其插入到receive_queue队列中。在复制的sk_buff中包含一个指向新创建的sock数据结构的指针。

16.4.5 接收连接请求

UDP并不支持连接的概念，INET 套接口连接请求只适用于TCP协议。接收操作将会被传送到所支持的协议层，在这里是INET接收任何到来的连接请求。如果INET协议层下面的协议不支持连接，例如UDP，那么INET协议层将会放弃接收操作。否则，接收操作将会传送到真正的协议中，在这里为TCP协议。接收操作可以是阻塞的和非阻塞的。在非阻塞的情况下，如果没有到来的连接请求，接收操作将会失败，同时新创建的套接口数据结构也会被放弃。在阻塞的情况下，执行接收操作的网络应用程序将会添加到等待队列中，然后挂起直到接收到 TCP连接请求。一旦接收到连接请求以后，包含请求的sk_buff将会被扔掉，同时sock数据结构将返回到INET 套接口层。新的套接口文件描述符（fd）返回到网络应用程序中，这样应用程序就可以在套接口操作中使用此文件描述符。

16.5 IP 层

16.5.1 套接口缓冲区

在网络环境中，每一层的协议都可以使用其下面一层的协议。但这也同时产生一个问题，那就是每一个协议都需要在传送时数据上添加协议头和协议尾，而在接收处理数据的时候移走协议头和协议尾。这样就使得在协议之间传递数据缓冲区变得十分的困难，因为每一个协议层都需要了解其特殊的协议头和协议尾的位置。一个解决办法就是在每一个协议层中复制缓冲区，但这样做的效率是非常低的。Linux使用套接口缓冲区或者sk_buffs来在协议层中间和网络设备中间传递数据。数据结构sk_buffs包括指针和长度字段，这样就允许每一个协议层通过标准的功能或者方法来处理应用程序的数据。

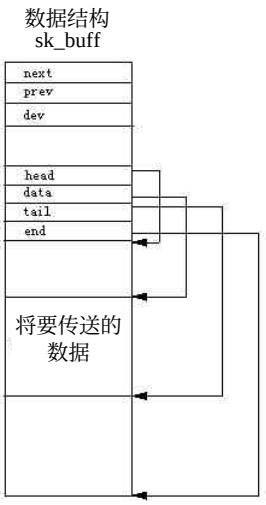

图16-4 SOCKET 缓冲区数据结构示意图

图16-4显示了数据结构sk_buff。每一个sk_buff都有一个数据块，并包括 4个数据指针，用来处理和管理套接口缓冲区的数据。

1. head（首部）

指向内存中数据区起始位置的指针。当sk_buff和它的数据块分配完毕以后，此指针就变为固定的值。

2. data（数据）

指向协议中数据当前起始位置的指针。它依据当前拥有sk_buff的协议层的不同而不同。

3. tail（尾部）

指向协议中数据当前末尾的指针。它同样依据当前拥有sk_buff的协议层的不同而不同。

4. end（结束）

指向内存中数据区的结束位置的指针。当sk_buff分配好了以后，此指针就为固定的值。

数据结构sk_buff中还包含两个长度字段，len 和 truesize，分别用于描述当前协议包的长度和整个数据缓冲区的长度。sk_buff的处理程序提供了标准机制，用于在应用程序的数据中添加

和移走协议头和协议尾。这些机制可以安全地处理数据结构 sk_buff中的数据、协议尾和长度字段：

1. push

此操作将data指针往数据区的起始位置移动，同时增加 len字段的值。往数据区的起始位置添加数据或协议头的时候可以使用它。

2. pull

此操作将 data 指针向数据区尾部的方向移动，同时减少 len字段的值。当把接收到的数据或协议头从数据的起始位置移走的时候可以使用它

3. put

此操作将tail指针向数据区的末端移动，同时增加 len字段的值。当往要传送的数据的末端添加数据或协议信息时可以使用它。

4. trim

此操作将tail指针向数据区的起始位置移动，同时减少 len字段的值。当从接收的数据包中移走数据或协议尾时可以使用它。

16.5.2 接收IP数据包

Linux系统内核初始化网络设备时，将产生一系列的 device数据结构，它们组成了dev_base链表。数据结构device 描述了其设备的状态并提供一系列可以调用的程序，在网络协议层需要网络设备工作时调用。这些程序大多涉及传送数据和网络设备的地址。当一个网络设备从网络中接收数据包时，它必须将接收到的数据转换成数据结构 sk_buff。这些sk_buff将会在接收时被网络设备添加到backlog队列中。如果backlog队列变得太大，那么系统将会扔掉一些接收到的sk_buff'。

当Linux网络层初始化时，每一个协议都将通过在 ptype_all链表或ptype_base 散列表中添加数据结构packet_type的方法登记。数据结构packet_type包含协议类型、一个指向网络设备的指针、一个指向接收数据处理过程的指针以及一个指向下一个 packet_type数据结构的指针。

16.5.3 发送IP数据包

数据产生时，将创建一个 sk_buff数据结构，它包含数据和协议层添加的各种协议头。sk_buff需要传送给网络设备以后才能发送。首先由协议，例如IP协议，决定要使用的网络设备，这取决于数据包的最佳路由。对于使用调制解调器，通过 PPP协议连接的单个网络来说，路由的选择相对简单。但对于以太网连接的多个计算机的网络，路由的选择则相对复杂。

对于每一个传送的IP数据包，IP协议使用路由表解决路由问题。在路由表中可以查找到的IP目的地址将会返回一个描述可以使用的路由 rtable数据结构，它包括一个源IP地址、device数据结构的地址以及一个预先创建的硬件头。这个硬件头与网络设备有关，并包含源和目的物理地址以及其他一些与特定介质有关的信息。如果网络设备是以太网设备，源和目的物理地址将会是以太网设备的物理地址。硬件头中也许包含需要使用 ARP协议解释的物理地址。硬件头将会保存在缓冲区中，以加快以后的查找速度。

16.5.4 数据碎片

每一个网络设备都有一个最大的数据包大小，它不能接收和发送大于此大小的数据包。 IP

协议遵守此规定，它将大的数据块分成较小的数据块，以便网络设备处理。IP协议头包括一个碎片字段、一个标志和一个碎片位移。

当一个IP数据包等待传送时，IP要找到发送数据的网络设备。IP通过路由表来决定使用哪一个设备发送IP数据包。每一个设备都包括一个描述设备可以传送的最大数据单位的字段，也就是mtu字段。如果设备的mtu字段比等待传送的IP数据包的大小的话，那么IP数据包就必须分割成mtu大小的碎片。每一个碎片都有一个sk_buff来代表，sk_buff的IP头可以表明这是一个IP数据包碎片，以及此碎片在IP数据包中的位移。如果在分割IP数据包的过程中，IP无法分配sk_buff,那么这个传送将会失败。

接收IP数据包碎片要比发送更为困难，因为接收端可以以任意的顺序接收到数据包碎片。而只有在接收了所有的数据包碎片以后才能将数据包碎片重新组装起来。每次接收一个IP数据包的时候，接收端都要查看此数据包是否为IP数据包碎片。第一次接收数据包碎片时，IP协议将创建一个新的ipq数据结构，并将此结构插入到ipqueue链表中。当接收到更多的IP数据包碎片时，接收端找到相应的ipq数据结构，同时创建一个新的数据结构ipfrag来描述此数据包碎片。每一个ipq数据结构都使用源和目的IP地址，上层协议的标识符以及此IP数据包的标识符用来唯一地描述接收的IP碎片。当接收到所有的碎片以后，它们组合成一个sk_buff数据结构，并将其传送到下一层的协议进一步处理。每一个ipq数据结构中都包含一个计时器，每当接收到一个有效的IP碎片以后，计时器都将重新工作。如果计时器失效，那么说明此IP数据包接收失败，上一层的协议将会负责重新传送此数据包。

16.6 地址解析协议

地址解析协议（ARP）负责将IP地址翻译成网络物理地址，例如以太网地址。IP协议在将数据发送到相应的网络设备前需要进行此种翻译。

ARP将会检查设备是否需要一个硬件头以及如果需要的话，此数据包的硬件头是否需要重新创建。Linux将硬件头保存在缓冲区中，以此避免频繁地重建硬件头。如果硬件头确实需要重建，ARP将调用和网络设备有关的硬件头重建子过程。所有的以太网设备都使用同样的硬件头重建子过程，而此子过程反过来使用ARP服务将目的IP地址翻译成物理地址。

ARP协议本身十分简单，它由两种类型的信息组成：ARP请求和ARP应答。ARP请求中包括需要翻译的IP地址，而ARP应答中包含的是已经翻译过的IP地址，也就是硬件地址。ARP请求将传送到网络中的所有主机中，所以对于一个以太网来说，连接在以太网上的所以机器都将接收到ARP请求。但只有包含此IP地址的计算机才作出应答，返回一个包含自己的物理地址的ARP应答。

在Linux中，ARP协议主要是一个由数据结构arp_table构成的表，表中的每一个数据结构都描述了IP地址到物理地址的翻译。arp_table入口在IP地址需要翻译时创建，并且在一段时间不用后将会从表中移走。数据结构arp_table中包含以下的字段：

last used	ARP入口最后被使用的时间。
last updated	ARP入口最新更新的时间。
flags	描述入口的状态。
IP address	入口描述的IP 地址。
hardware address	翻译以后的硬件地址 。
hardware header	指向硬件头的指针。
timer	ARP请求没有得到回应的失效的计时器。

retries	ARP请求重试的次数。
sk_buff queue	等待翻译的sk_buff 的入口。

当一个IP地址需要翻译时，通常在ARP表中并没有相应的arp_table入口，所以ARP将发送一个ARP请求信息，同时在ARP表中创建一个新的arp_table入口，并将需要地址翻译的sk_buff数据结构插入到新的入口的sk_buff queue等待队列中。在ARP发送出ARP请求以后，计时器就开始工作。如果在计时器失效以后还没有回应，那么 ARP将会重复在retries字段中设置的次数的请求。如果还是没有回应， ARP 将会把此arp_table入口从ARP表中移走。这时，等待翻译的队列中所有的sk_buff数据结构都将得到通知，同时 ARP告诉上一层的协议地址翻译请求失败。如果ARP请求得到了应答，那么arp_table入口就可以标记为complete，所有等待的sk_buff'都将从等待队列中移走，继续传送。翻译后的硬件地址将会保存在每一个 sk_buff的硬件头内。

经过一段时间，网络的拓扑结构可能会发生变化， IP地址也可能重新指定给其他的硬件地址。例如，拨号服务就是在连接时才指定 IP地址。为了保证ARP表保存最新的入口信息， ARP定时检查ARP表中的arp_table，以便移走已经失效的入口。有些入口是永久性的入口，它们不能被移走。