第 12 章 网络-套接字与传输层

计算机网络用于将多个计算机通过网络介质连接在一起，以实现彼此之间的数据传输。计算机网络有多种类型，各种类型的网络通过物理介质传输数据，传输数据的方式格式，即传输协议各不相同。

网络中各主机之间的数据传输由进程发起和接收，也就是说进程是数据的发送者和接收者。数据在网络介质中传输需要遵循相同的规范，称为网络协议。发送进程按照网络协议要求封装好数据后通过网络设备发送到网络中，数据在网络中传输到达目的主机，由目的主机网络设备接收并通过网络协议解析后传递给接收进程。

Linux 内核虽然也将网络设备视为文件，但进程不是通过网络设备文件访问网络的。进程网络数据直接提交给网络协议，由网络协议通过路由选择发送的网络设备。网络设备接收到数据时，也是提交给网络协议，由网络协议传递给进程。用户进程实际是直接跟网络协议通信。内核支持的网络协议需要定义并注册协议操作接口 proto_ops 结构体实例，此结构体相当于文件操作中的 file_operations 结构体，是网络操作的接口。

Linux 内核中通过套接字访问网络，创建套接字时需要指明网络协议，并关联 proto_ops 实例，通过此实例中的函数访问网络。套接字创建时，被赋予一个文件描述符，进程通过文件描述符标识套接字。内核定义了访问网络的专用系统调用，也可以通过文件操作 file_operations 结构体接口访问套接字（网络）。套接字文件操作 file_operations 实例中的函数调用 proto_ops 实例中的函数通过网络协议访问网络。

目前，最知名的计算机网络当属因特网（Internet），它将世界上数以亿计的主机连接在一起，实现了远程主机之间的数据通信。因特网中每个主机被分配了一个 IP 地址，网络数据在传输过程中由路由器选择传输路径，最终到达目的主机。因特网网络协议采用了分层的结构，Linux 内核将网络协议分为应用层、传输层、网络层和数据链路层。

应用层位于用户空间，由用户进程实现，其余三层位于内核空间，由内核实现。传输层协议用于实现进程到进程的通信，网络层用于实现主机到主机的通信。网络层通过 IP 地址将数据传输到目的主机，而传输层通过进程端口号将数据传递给目的主机中的目的进程。

数据链路层对上接收网络层下发的数据，上传网络设备接收的数据。对下将数据发送到网络中相邻节点，以及从相邻节点接收数据提交到网络层。网络数据传输路径中各相邻节点之间的传输介质、传输协议可能不同，例如：主机与 WiFi 热点之间是无线传输，而 WiFi 热点与相邻路由器之间是有线传输。数据链路层通过网络设备实现数据在网络介质中的传输，网络设备驱动程序位于数据链路层，按照数据链路层传输协议发送和接收数据。

本章介绍套接字的定义及网络操作接口，netlink 套接字的实现，因特网的物理结构和分层协议，因特网传输层协议的实现等。下一章将介绍网络层协议（IPv4）、数据链路层协议在内核的实现，并简单介绍 IPv6 网络层协议在内核中的实现。

12.1 前言

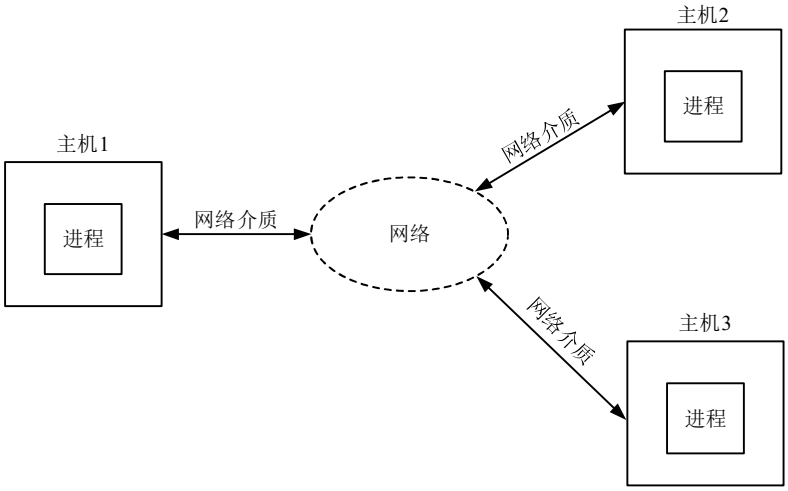
进程是网络数据传输的主体和客体，数据传输的两端都是进程，这不同于进程与文件（外部设备）之间的数据传输。进程与文件（外部设备）之间是多对一的关系，数据的传输、设备的控制都是由进程发起的。网络两端连接的都是进程，两端都是多个进程，通过网络链路需要实现多对多的关系，而且每个进程都可以是通信的主体（控制端）和客体。

网络数据传输会遇到许多新问题，例如：目的进程的寻址、网络通道的拥塞、数据的丢失等，因此内核使用了套接字来实现进程对网络的操作，而不是通过网络设备文件操作网络。内核定义了专门的网络操作接口 proto_ops 结构体，由各协议类型实现此实例。创建套接字需关联协议定义的 proto_ops 实例，由此实现对网络的操作，类似于文件操作 file_operations 结构体。

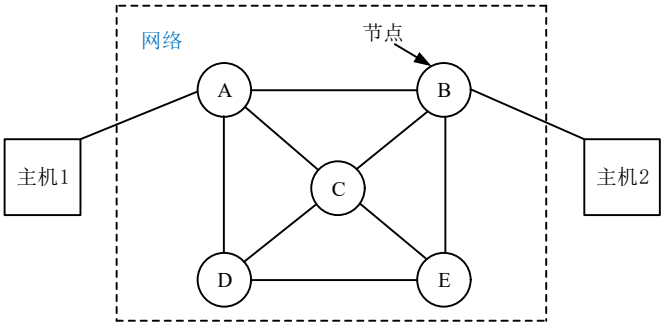
内核定义了网络命名空间，用于隔离网络资源。每个网络命名空间，有自己的网络设备、网络资源及配置参数等，以实现轻量级的虚拟化。

12.1.1 网络访问接口

计算机网络用于将不同的主机连接在一起，下图示意了一个简单的计算机网络：

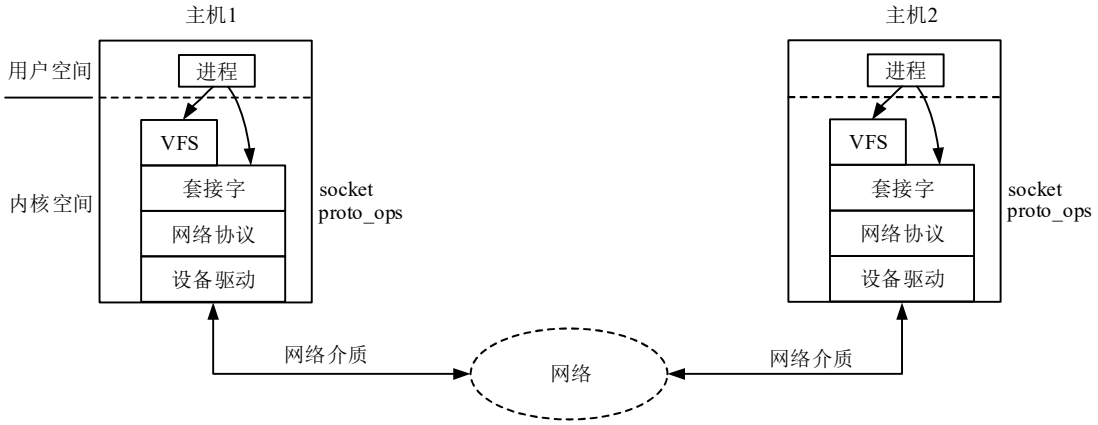


各主机通过网络介质，如网线、无线连接等，接入网络。网络中存在多个中间节点，用于中转转发数据，如下图所示。



如上图所示，主机 1 向主机 2 发送数据有多条路径，如：主机 1-A-B-主机 2，主机 1-A-C-B-主机 2 等。网络中的中间节点需要执行数据传输路径的选择，优先选择最佳路径，如果最佳路径中有节点拥塞、传输介质损坏等，节点可选择其它路径传输数据。例如：A 节点可选择通过 B、C 或 D 节点转发主机 1 的数据。

前面介绍过，进程通过套接字访问网络，各种类型网络协议需要实现网络操作接口 proto_ops 结构体实例。进程在创建套接字时需要指明访问网络的类型（网络协议），套接字由 socket 结构体表示，套接字将关联网络协议定义的 proto_ops 实例，进程对网络的操作将由 proto_ops 实例中的操作函数完成。进程访问网络的接口如下图所示：



进程创建套接字后，内核返回一个文件描述符，套接字 socket 实例关联到进程文件 file 实例，内核也

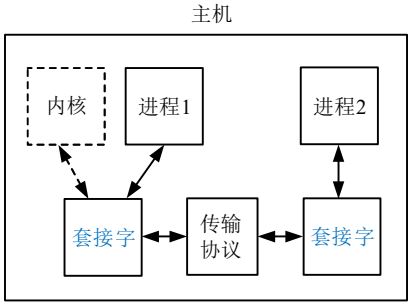
提供了通过文件操作 file_operations 接口访问网络的方式，file_operations 实例中操作函数调用 proto_ops 实例中的函数完成操作。套接字 socket 结构体可类比于 VFS 中的文件 inode 结构体。

另外，网络有些操作不能合并到 file_operations 接口，因此内核还提供了操作网络的专用系统调用，专用系统调用中直接调用网络协议定义的 proto_ops 实例中的函数，完成网络操作。套接字中通常维护着接收/发送数据的缓存队列，进程从缓存队列接收数据和发送数据。

网络设备驱动程序负责将网络协议下传的数据通过网络设备发送到网络介质，网络设备从网络介质接收数据，交由驱动程序将数据上传给网络协议。

网络中的中间节点设备只需要负责路径的选择和数据的转发即可，不需要对数据进行过多的处理。

套接字除了用于网络数据传输外，还可以用来实现同一主机内进程与进程、进程与内核之间的数据传输。此时，数据只在本机内存中移动，不需要外传，因此传输协议比较简单，如下图所示：



例如：netlink 套接字是一种在主机内进行数据传输的套接字，主要用于用户进程与内核之间的数据传输，后面将详细介绍。

12.1.2 网络命名空间

命名空间是一种轻量级的进程虚拟化，它提供了资源隔离功能。在命名空间内的进程只能看见和使用本命名空间内的资源。不同于 KVM 和 Xen 等虚拟化解决方案，使用命名空间时，不会在主机上创建额外的操作系统实例，而只使用一个操作系统实例。

Linux 内核支持网络命名空间，从逻辑上说，网络命名空间是网络栈的副本，每个网络命名空间有其自身的网络设备、路由选择表、邻居表、Netfilter 表、内核套接字、网络 procfs 表项和其它网络资源等。内核若要支持创建新网络命名空间，需要选择 NET_NS 配置选项，否则所有网络资源都属于初始网络命名空间。

请读者注意，说到网络不只就是 IPv4 和 IPv6 网络，还有其它类型的网络协议，网络命名空间管理多种类型网络协议的资源和参数。

网络命名空间由 net 结构体表示，定义如下（/include/net/net_namespace.h）：

```
struct net {
    atomic_t        passive;    /*确定网络命名空间何时释放*/
    atomic_t        count;      /*确定网络命名空间何时关闭*/
    spinlock_t          rules_mod_lock;
    atomic64_t          cookie_gen;

    struct list_head    list;       /*双链表成员，将实例添加到全局双链表 net_namespace_list*/
    struct list_head    cleanup_list;   /* namespaces on death row */
    struct list_head    exit_list;      /* Use only net_mutex */

    struct user_namespace   *user_ns;   /*创建网络命名空间的用户命名空间*/
```

```
    spinlock_t              nsid_lock;
    struct idr          netns_ids;
    struct ns_common    ns;     /*命名空间通用结构，详见第 5 章*/
    struct proc_dir_entry      *proc_net;     /*网络络命名空间在 procfs 中的表项*/
    struct proc_dir_entry      *proc_net_stat;

#ifdef CONFIG_SYSCTL
    struct ctl_table_set  sysctls;
#endif

    struct sock             *rtnl;              /*指向 rtnetlink 套接字 sock 实例，用于网络子系统*/
    struct sock             *genl_sock;     /*指向通用 netlink 套接字 sock 实例*/

    struct list_head        dev_base_head;     /*管理命名空间内的网络设备 net_device 实例*/
    struct hlist_head       *dev_name_head;   /*指向通过设备名称管理的网络设备散列链表*/
    struct hlist_head       *dev_index_head;   /*指向通过设备索引值管理的网络设备散列链表*/
    unsigned int            dev_base_seq;       /* protected by rtnl_mutex */
    int                 ifindex;                    /*网络命名空间中分配的最后一个设备索引值*/
    unsigned int            dev_unreg_count;

    struct list_head        rules_ops;          /*管理路由选择表操作结构实例，用于策略路由选择*/
    struct net_device       *loopback_dev;              /*指向环回网络设备*/

    struct netns_core       core;
    struct netns_mib        mib;
    struct netns_packet     packet;
    struct netns_unix       unx;
    struct netns_ipv4       ipv4;       /*IPv4 网络协议资源，路由选择表、邻居表、各种参数等*/

#if IS_ENABLED(CONFIG_IPV6)
    struct netns_ipv6       ipv6;         /*IPv6 网络协议资源*/
#endif
#if IS_ENABLED(CONFIG_IEEE802154_6LOWPAN)
    struct netns_ieee802154_lowpan    ieee802154_lowpan;
#endif
#if defined(CONFIG_IP_SCTP) || defined(CONFIG_IP_SCTP_MODULE)
    struct netns_sctp       sctp;       /*传输层 SCTP 协议*/
#endif
#if defined(CONFIG_IP_DCCP) || defined(CONFIG_IP_DCCP_MODULE)
    struct netns_dccp       dccp;         /*传输层 DCCP 协议*/
#endif

#ifdef CONFIG_NETFILTER      /*#if CONFIG_NETFILTER 开始*/
    struct netns_nf         nf;
    struct netns_xt         xt;         /*管理各网络层协议中的 xt_table 实例*/
  #if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
```

```
        struct netns_ct         ct;
    #endif
    #if defined(CONFIG_NF_TABLES) || defined(CONFIG_NF_TABLES_MODULE)
        struct netns_nftables       nft;
    #endif
    #if IS_ENABLED(CONFIG_NF_DEFRAG_IPV6)
        struct netns_nf_frag        nf_frag;
    #endif
        struct sock             *nfnl;          /*指向 nfnetlink 内核套接字*/
        struct sock             *nfnl_stash;    /*指向 nfnetlink 内核套接字*/
#endif     /*#if CONFIG_NETFILTER 结束*/

#ifdef CONFIG_WEXT_CORE
        struct sk_buff_head  wext_nlevents;
#endif

        struct net_generic __rcu *gen;   /*用于传递命名空间私有数据，/include/net/netns/generic.h*/

#ifdef CONFIG_XFRM
        struct netns_xfrm    xfrm;   /*xfrm 框架，用于 IPsec 子系统*/
#endif
#if IS_ENABLED(CONFIG_IP_VS)
        struct netns_ipvs    *ipvs;
#endif
#if IS_ENABLED(CONFIG_MPLS)
        struct netns_mpls    mpls;
#endif
        struct sock      *diag_nlsk;     /*监视套接字*/
        atomic_t         fnhe_genid;
};
```

网络命名空间 net 结构体中部分成员简介如下：

●**list**：双链表成员，用于将网络命名空间实例添加到全局双链表 **net_namespace_list**，在创建网络命名空间时添加。

●**rtnl**：指向网络命名空间的 rtnetlink 套接字 sock 实例，sock 结构体定义见下文。

●**genl_sock**：指向通用 netlink 套接字 sock 实例。

●**dev_base_head**：双链表头，管理命名空间内网络设备的 net_device 实例。

●**dev_name_head**：通过设备名称管理的网络设备散列链表。

●**dev_index_head**：通过设备索引值管理的网络设备散列链表。

●**ifindex**：网络命名空间中分配的最后一个网络设备索引值。

●**ipv4**：netns_ipv4 结构体成员，表示因特网 IPv4 网络协议的资源。

●**ipv6**：netns_ipv6 结构体成员，表示因特网 IPv6 网络协议的资源。

内核在/net/net_namespace.c 文件内定义了初始网络命名空间 init_net 实例：

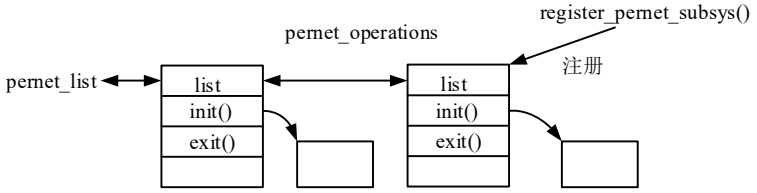
```
struct net init_net = {
    .dev_base_head = LIST_HEAD_INIT(init_net.dev_base_head),
```

};

在复制进程时将根据标记值，调用接口函数 copy_net_ns()复制（创建）网络命名空间。所有网络命名空间实例由全局双链表 net_namespace_list 管理。

在创建新网络命名空间时，需要初始化命名空间内各种网络资源。

内核在/include/net/net_namespace.h 头文件定义了 pernet_operations 结构体，用于向网络命名空间中注册资源、设置参数等。内核中所有 pernet_operations 实例由 pernet_list 双链表管理，如下图所示：



内核中需要向网络命名空间注册资源、设置参数等的子系统，需要定义并注册 pernet_operations 结构体实例（添加到全局双链表）。

在创建新网络命名空间的 copy_net_ns()函数中将调用 setup_net()函数，在 setup_net()函数中将遍历全局双链表 pernet_list 中的 pernet_operations 实例，对新网络命名空间调用各实例的 **init()函**数，完成各子系统在新网络命名空间内的初始化工作。在删除网络命名空间时将调用各实例中的 exit()函数，执行清理工作。pernet_list 双链表管理的 pernet_operations 实例适用于所有的网络命名空间，而不只在指定的网络命名空间中完成初始化工作。

pernet_operations 结构体定义如下：

```
struct pernet_operations {
    struct list_head list;        /*双链表节点，将实例链接到全局双链表*/
    int (*init)(struct net *net);     /*初始化函数，创建网络命名空间时调用*/
    void (*exit)(struct net *net);   /*删除网络命名空间时调用的函数*/
    void (*exit_batch)(struct list_head *net_exit_list);
    int *id;
    size_t size;
};
```

注册 pernet_operations 实例的接口函数为 **register_pernet_subsys**(struct pernet_operations *ops)，函数主要工作是将实例添加到全局双链表，并在所有现有网络命名空间中调用 init()函数，完成子系统在各网络命名空间中的初始化工作。

网络命名空间相关代码位于/net/core/net_namespace.c 文件内。

12.1.3 网络代码目录

内核中网络公共代码、各网络协议实现代码位于/net 目录下，网络设备驱动程序位于/drivers/net 目录下。下面简要列出/net 和/drivers/net 目录下的部分子目录和文件：

目录/文件	描述	备注
/net		
/socket.c	套接字相关代码，主要实现套接字与用户的接口等。	适用于所有网络协议（类型）。
/core	网络、各网络协议公用代码，sock、sk_buff 操作等。	适用于所有网络协议（类型）。
/netlink	netlink 套接字实现。	主要用于进程与内核通信。
/ipv4	IPv4 网络层协议实现代码。	因特网
/ipv6	IPv6 网络层协议实现代码。	因特网

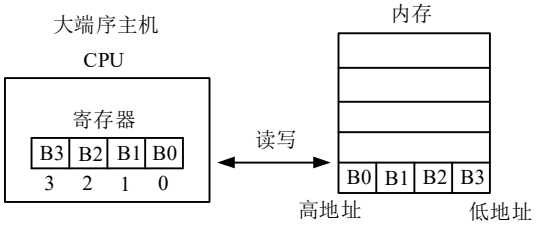
/netfilter	Netfilter 机制实现代码。	向网络栈中插入钩子函数。
/ethernet	以太网协议公共代码。	因特网数据链路层
/sched	实现发送数据包调度器，应用于数据链路层。	
/drivers/net		
/loopback.c	向网络命名空间注册环回网络设备。	
/mii.c	实现数据链路层 MAC 与 PHY 连接的 MII 接口的操作函数。	
/phy	PHY 层驱动代码。	
/ethernet	以太网网络设备驱动程序。	

12.1.4 网络字节序

计算机中的字节序是指 CPU 核从内存读取/写入半字（2 字节）、字（4 字节）和双字（8 字节）时，各字节在内存中的存放顺序。大端序表示高字节存放在内存中的低地址，低字节存放在内存中的高地址，小端序表示高字节存放在内存中的高地址，低字节存放在内存中的低地址。

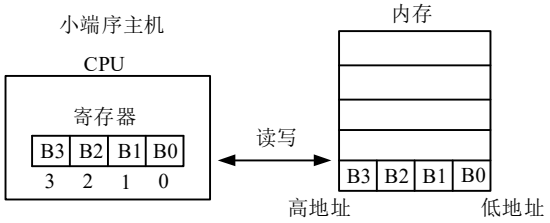
处理器体系结构会使用小端序或大端序其中的一种。有些体系结构能够支持两种字节序，但必须配置为使用其中的一种。

大端序（big endian）如下图所示：



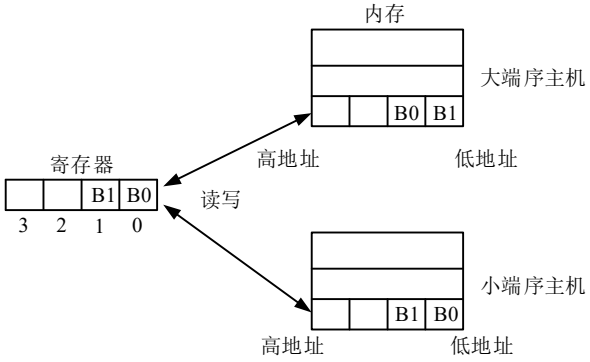
CPU 从物理内存中读取一个 32 位整数至寄存器时，位于内存低地址的字节加载到寄存器中高字节，内存高字节加载到寄存器中低字节。写操作也是按照相同的次序存放。半字、双字也类似，内存最低地址字节加载到寄存器中最高字节，内存中最高字节加载到寄存器中最低字节。

小端序（little endian）如下图所示：



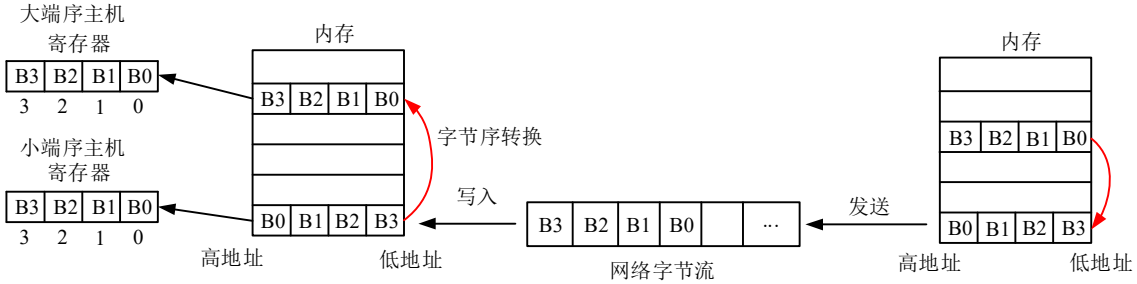
上图中示意了同一个整数在小端序体系结构中在内存中的保存次序。整数中低字节保存在内存中低地址，高字节保存在内存中高地址。

下图示意了同一个半字在大端序体系结构和小端系体系结构中的存放情形：



在网络数据传输中，数据视为一个字节流，主机接收到的字节流按顺序依次在物理内存中从低地址到高地址存放。发送数据时，从低地址到高地址依次发送字节流。

在因特网网络协议中，默认将所有主机都视为大端序主机。如下图所示，传输的 4 字节整数，先发送的是高字节，后面是低字节，保存到目的主机内存时，也是大端序的。



如果主机是大端序的，那么没有问题，数据可以直接发送和接收。如果主机是小端序的，则在发送前需要将小端序数据转换成大端序数据，接收后需要将大端序数据转换成小端序数据，否则数据就会出错。

字节序对字节流（字符串）的读写没有影响，只影响半字、字、双字。在 C 语言中定义的数据类型，默认为当前 CPU 主机使用的字节序。

内核定义了以下宏用于指明（短）整型数的字节序：

- **__le16**：小端序短整型数。
- **__le32**：小端序整型数。
- **__le64**：小端序（长）整型数。
- **__be16**：大端序短整型数。
- **__be32**：大端序整型数。
- **__be64**：大端序（长）整型数。

内核在/include/linux/byteorder/generic.h 头文件内定义了将主机字节序整数转为网络字节序（大端序）整数，以及将网络字节序整数转换成主机字节序的宏，例如：

- **htonl(x)**：将主机字节序整数（32 位）转为网络字节序整数。
- **ntohl(x)**：将网络字节序整数（32 位）转为主机字节序整数。
- **htons(x)**：将主机字节序短整数（16 位）转为网络字节序短整数。
- **ntohs(x)**：将网络字节序短整数（16 位）转为主机字节序短整数。

如果主机是大端序的，则以上函数直接返回 x 值即可，不需要处理。如果主机是小端序的，以上函数主要工作是对整型数字节位置进行交换。

由于网络字节序是大端的，所以内核中定义的用于保存网络数据的数据结构中，各成员声明指定为大端序，也就是说数据结构中保存的是大端序。

主机在使用数据结构中数据时需调用 ntohl(x)、ntohs(x)等函数，将成员值转换成主机字节序。在向数据结构中成员赋值时，需调用 htonl(x)、htons(x)等函数，将主机字节序整数转换成大端序后填充至数据结

构中。

12.2 套接字

套接字可视为一种进程间通信机制，可用于本机进程之间、进程与内核之间的通信。Linux 内核还使用套接字来访问网络，而不是通过网络设备文件访问网络。

12.2.1 概述

套接字适用于多种数据传输协议，包括网络传输协议。各数据传输协议需要定义操作接口 proto_ops 实例，由此实例执行具体的操作。

进程在创建套接字时需要指明使用的传输协议，从而关联到正确的 proto_ops 实例，进程进而通过此实例中的操作函数实现进程间的数据传输，以及访问网络等。

1 套接字框架

套接字的特性由 3 个属性确定：通信域（domain）、套接字类型（type）和协议类型（protocol），进程在创建套接字时需指明这 3 个属性。

通信域可理解成一个数据传输的领域，例如，本机进程与进程之间的通信可视为一个通信域，本机进程与内核之间的通信也可视为一个通信域，进程通过某种网络协议与网络中另一主机中的进程通信也视为一个通信域等。每个通信域需要定义一个数据传输的协议，这个协议可能由多个具体的协议类型组成，统称为协议簇。协议簇与通信域是一一对应的关系，也就是说某一通信域采用固定的协议簇，因此通信域与协议簇是等价的。

每个通信域需要使用某种固定的数据结构来标识主机和进程，这种标识称为地址簇。地址簇（主机和进程的标识方法）与通信域也是一一对应的关系。通信域、协议簇、地址簇三者之间是等价的，都唯一标识了套接字数据传输所采用的协议。协议类型是协议簇中某一具体的传输协议类型，协议类型可按传输数据的功能划分，也可以按传输数据的形式划分。

套接字类型表示数据的通信方式，主要有流套接字和数据报套接字。流套接字向进程提供一个有序、可靠、双向的字节流连接，发送的数据可以确保不会丢失、复制或乱序到达。在传输数据前，两个套接字之间需要建立连接，类似于生活中的打电话，通话前需要先拨通对方电话，并且要对方接听了才能进行通话。

数据报套接字不需要建立连接，它提供的是一种无序的不可靠的数据传输，数据可能丢失、复制或乱序到达。这类似于生活中的寄信件，只要在信件上写明地址和收件人投递出去即可，不需要与收件人取得联系，信件也可能到达不了或丢失。

内核套接字框架如下图所示，用户进程通过 socket()系统调用创建套接字，系统调用中需指明使用的通信域（地址簇）、套接字类型和协议类型。系统调用中将创建套接字 socket 实例，并关联到进程文件 file 实例，file 实例私有数据指针指向 socket 实例，file 实例关联 file_operations 结构体实例为 socket_file_ops（适用于所有套接字），用于进程通过 VFS 接口操作套接字，如 read()、write()系统调用。

socket()系统调用返回文件描述符，用户进程还可以此文件描述符通过专用系统调用访问套接字。

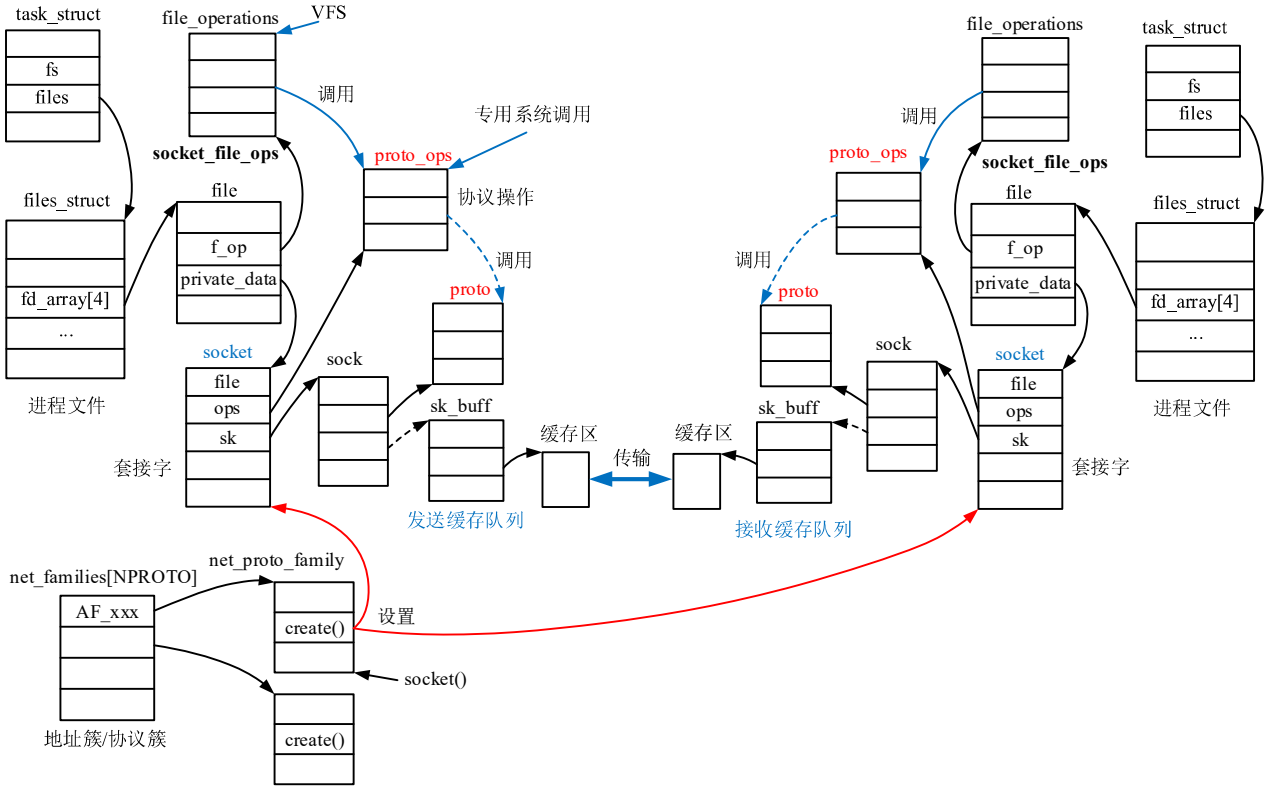
每个地址簇（协议簇）代码需要在内核定义并注册 net_proto_family 结构体实例，并定义协议操作结构 proto_ops 实例。在 socket()系统调用中，将根据参数指定的地址簇，调用对应 net_proto_family 实例中的 create()函数，在此函数中会将 proto_ops 实例赋予 socket 实例，并创建、初始化在协议中表示套接字的 sock 结构体实例等。

socket 是套接字在虚拟文件系统中的表示，实现与用户进程的接口。sock 结构体是套接字在协议中的表示，两者是一一对应的关系。socket 可以认为是套接字的通用表示，sock 结构体表示套接字特定于协议的信息。

sock 结构体通常嵌入到各传输协议定义的私有数据结构中，sock 结构体包含套接字更详细的信息，如协议类型、接收/发送数据缓存队列等等。

proto_ops 结构体是进程操作套接字的接口。例如，在同一协议簇下，流套接字对应一个 proto_ops 实例，数据报套接字对应一个 proto_ops 实例。如果同一类型套接字下还有不同的传输协议的话，每种协议再定义 proto 结构体实例，proto_ops 实例中函数调用 proto 实例中函数。

另外，proto 结构体中还包含对 sock 结构体实例本身的操作函数，如对 sock 实例的创建、初始化、管理等，即对本协议下套接字的管理和操作。



用户进程通过套接字传输数据时，数据被打包成数据包，一次传输可能需要多个数据包。数据包中数据保存在一个内存缓存区中，数据包头部需要加上一个报头，用于标识目的主机、进程、数据包类型等信息。如果传输协议采用了分层结构，则每一层都需要在数据包上添加一个报头。

数据包在内核中由 sk_buff 结构体表示，它关联到保存数据的内存缓存区。数据包在内核中的处理以 sk_buff 实例的形式进行。

sock 结构体中通常包括发送、接收数据包的缓存队列。发送数据包时，传输协议类型定义的发送操作函数（位于 proto_ops 实例或 proto 实例），可以将数据包直接发送到对方套接字，也可以将其添加到发送缓存队列，在合适的时机再发送出去。传输协议在接收数据包时，通常先将其添加到接收缓存队列，用户进程从此缓存队列读取数据。

套接字公共层代码主要在/net/socket.c 和/net/core/sock.c 文件内实现，在/net/core/skbuff.c 文件内实现数据包 sk_buff 实例的公共操作函数。

2 初始化

套接字初始化函数为 sock_init()，主要是为相关数据结构创建 slab 缓存，注册并挂载 sockfs 伪文件系统等，函数定义如下（/net/socket.c）：

```
static int __init sock_init(void)
{
    int err;
    err = net_sysctl_init();   /*网络系统控制参数初始化，/net/sysctl_net.c*/
```

```
    skb_init();   /*创建 sk_buff、sk_buff_fclones 结构体 slab 缓存，/net/core/skbuff.c*/

    init_inodecache(); /*创建 socket_alloc 结构体缓存，用于 sockfs，见下文，/net/socket.c*/

    err = register_filesystem(&sock_fs_type);     /*注册 sock_fs_type 伪文件系统类型，见下文*/
    ...
    sock_mnt = kern_mount(&sock_fs_type);   /*挂载 sock_fs_type 伪文件系统*/
    ...

#ifdef CONFIG_NETFILTER
    err = netfilter_init();     /*netfilter 子系统初始化，/net/netfilter/core.c*/
    ...
#endif
    ptp_classifier_init();     /*需选择 NET_PTP_CLASSIFY 选项，否则为空*/
out:
    return err;
    ...
}
core_initcall(sock_init);      /*内核启动阶段调用*/
```

内核通过 sockfs 伪文件系统管理套接字 socket 实例，类似于块设备层的 bdev 伪文件系统，sockfs 伪文件系统在后面将详细介绍。

12.2.2 地址簇

前面介绍过，每个通信域（协议簇）需要用一个固定的数据结构来标识主机和进程，以便双方之间通信，这个数据结构可理解成用于标识主机和进程的地址。由于每个协议簇采用固定的格式来表示地址，因此协议簇又称为地址簇。

1 地址簇

内核在/include/linux/socket.h 头文件中以整数的形式标识了套接字适用的地址簇（协议簇），如下所示：

```
#define   AF_UNSPEC      0    /*不表示特定的地址簇，也就可以表示所有的地址簇*/
#define   AF_UNIX        1    /* Unix 地址簇，用于同一主机中进程间通信*/
#define   AF_LOCAL       1    /* POSIX name for AF_UNIX*/
#define   AF_INET        2    /*IPv4 地址簇，用于因特网*/
#define   AF_AX25        3    /* Amateur Radio AX.25*/
#define   AF_IPX         4    /* Novell IPX */
#define   AF_APPLETALK   5    /* AppleTalk DDP*/
#define   AF_NETROM      6    /* Amateur Radio NET/ROM*/
#define   AF_BRIDGE      7    /* Multiprotocol bridge  */
#define   AF_ATMPVC      8    /* ATM PVCs*/
#define   AF_X25         9    /* Reserved for X.25 project  */
#define   AF_INET6       10   /* IPv6 地址簇，用于因特网*/
#define   AF_ROSE        11   /* Amateur Radio X.25 PLP  */
#define   AF_DECnet      12   /* Reserved for DECnet project*/
```

```
#define   AF_NETBEUI      13   /* Reserved for 802.2LLC project*/
#define   AF_SECURITY     14   /* Security callback pseudo AF */
#define   AF_KEY          15   /* PF_KEY key management API */
#define   AF_NETLINK      16   /*netlink 地址簇，用于主机内核与进程间通信*/
#define   AF_ROUTE    AF_NETLINK   /* Alias to emulate 4.4BSD */
#define   AF_PACKET       17   /* Packet family*/
#define   AF_ASH          18   /* Ash*/
#define   AF_ECONET       19   /* Acorn Econet*/
#define   AF_ATMSVC       20   /* ATM SVCs */
#define   AF_RDS          21   /* RDS sockets*/
#define   AF_SNA          22   /* Linux SNA Project (nutters!) */
#define   AF_IRDA         23   /* IRDA sockets*/
#define   AF_PPPOX        24   /* PPPoX sockets*/
#define   AF_WANPIPE      25   /* Wanpipe API Sockets */
#define   AF_LLC          26   /* Linux LLC  */
#define   AF_IB           27   /* Native InfiniBand address  */
#define   AF_MPLS         28   /* MPLS */
#define   AF_CAN          29   /* Controller Area Network，控制局域网*/
#define   AF_TIPC         30   /* TIPC sockets*/
#define   AF_BLUETOOTH    31   /* Bluetooth sockets，蓝牙*/
#define   AF_IUCV         32   /* IUCV sockets*/
#define   AF_RXRPC        33   /* RxRPC sockets */
#define   AF_ISDN         34   /* mISDN sockets */
#define   AF_PHONET       35   /* Phonet sockets，手机网络*/
#define   AF_IEEE802154   36   /* IEEE802154 sockets*/
#define   AF_CAIF         37   /* CAIF sockets，用于手机设备*/
#define   AF_ALG          38   /* Algorithm sockets*/
#define   AF_NFC          39   /* NFC sockets，近场通信协议*/
#define   AF_VSOCK        40   /* vSockets*/

#define   AF_MAX          41   /*地址簇最大数量*/
```

协议簇标识与地址簇标识取值相同，例如：

```
#define   PF_UNSPEC       AF_UNSPEC
#define   PF_UNIX         AF_UNIX
#define   PF_LOCAL        AF_LOCAL
#define   PF_INET         AF_INET      /*IPv4 协议簇*/
...
#define   PF_INET6        AF_INET6     /*IPv6 协议簇*/
...
#define   PF_NETLINK      AF_NETLINK     /*netlink 协议簇*/
...
```

在创建套接字的 socket()系统调用中需要指明套接字使用的地址簇和协议类型，例如：PF_INET 表示 IPv4 地址簇。协议类型表示地址簇（协议簇）下的子类型。

内核定义了通用 sockaddr 结构体，作为表示各地址簇地址值的通用模板（/include/linux/socket.h）：

```
struct sockaddr {
    sa_family_t    sa_family;      /*地址簇标识, AF_xxx  */
    char           sa_data[14];    /*地址值*/
};
```

各地址簇表示地址的方式各有不同，例如：AF_UNIX 地址簇用文件的路径名表示地址值，AF_INET 地址簇用 IP 地址加端口号表示地址值。各地址簇的地址值写入 sa_data[14]数组成员。

2 设置套接字

每种地址簇代码在初始化函数中需要定义并注册 net_proto_family 结构体实例，用于在创建本地址簇套接字时设置 socket 实例。

net_proto_family 结构体定义在/include/linux/net.h 头文件：

```
struct net_proto_family {
    int         family;     /*地址簇标识，AF_xxx*/
    int         (*create)(struct net *net, struct socket *sock,int protocol, int kern); /*设置 socket 实例函数*/
    struct module  *owner;
};
```

family 成员为地址簇标识，create()函数用于设置（初始化）套接字 socket 实例。

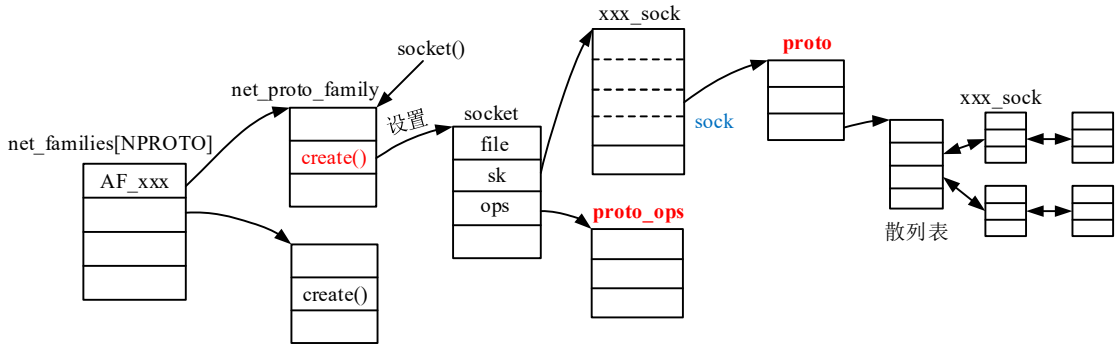
内核定义了全局指针数组用于管理 net_proto_family 实例（/net/socket.c）：

```
static const struct net_proto_family __rcu *net_families[NPROTO] __read_mostly;
```

数组项数 NPROTO 等于 AF_MAX，即内核支持的地址簇最大数量。

注册 net_proto_family 实例的函数 **sock_register**(const struct net_proto_family *ops)定义在/net/socket.c 文件内，主要工作是将 net_proto_family 实例关联到全局指针数组 **net_families[]**中的对应项，地址簇标识作为数组项索引值。

net_proto_family 结构体中 create()函数用于在创建套接字时，设置 socket 实例，主要是设置 socket.ops 成员（proto_ops 实例），为 socket 实例分配 sock 实例并初始化等，如下图所示。其中 sock 结构体通常嵌入到地址簇定义的私有数据结构中，sock 实例由 proto 实例中的散列表管理。

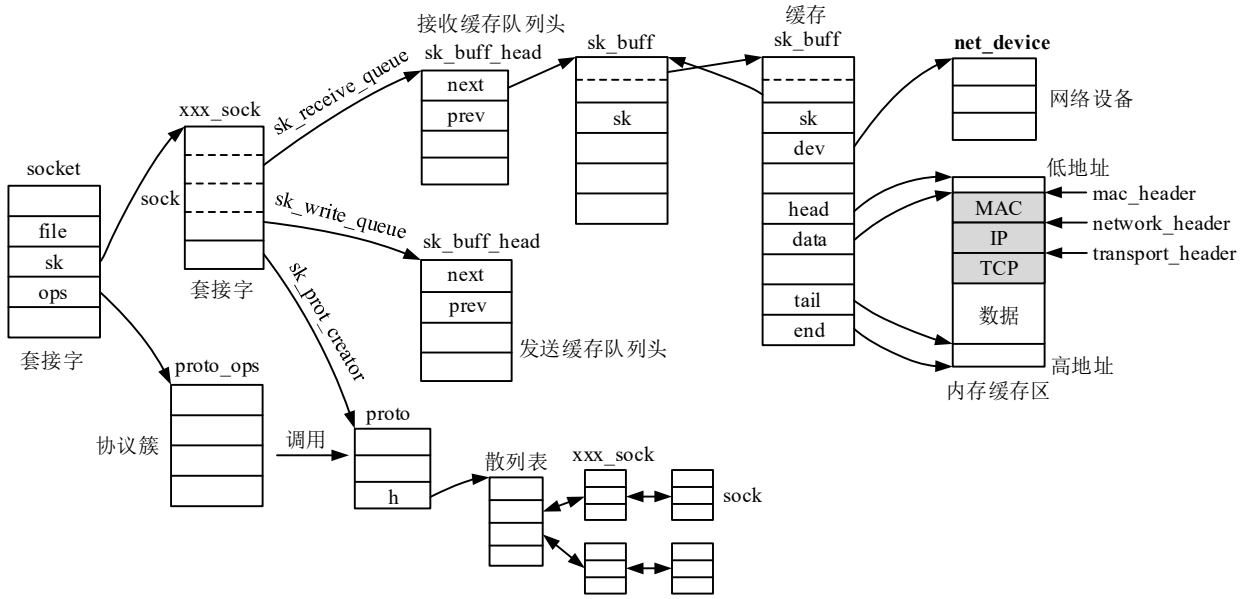


12.2.3 数据结构

套接字相关数据结构组织关系如下图所示。socket 结构体在 VFS 中表示套接字，proto_ops 表示协议簇定义的操作结构，实现特定于协议的操作。

sock 结构体是在传输协议中套接字的表示，其中包含两个缓存队列，分别是接收和发送数据包缓存队列。缓存队列头由 sk_buff_head 结构体表示，队列成员为数据包 sk_buff 实例。sk_buff 结构体中包含指向

内存缓存区的指针，内存缓存区用于保存发送/接收的数据，以及各层协议的报头。



1 socket

socket 结构体定义如下（/include/linux/net.h）：

```
struct socket {
    socket_state        state;          /*状态*/
    kmemcheck_bitfield_begin(type);
    short               type;           /*类型*/
    kmemcheck_bitfield_end(type);
    unsigned long       flags;          /*标记*/
    struct socket_wq __rcu  *wq; /*指向 socket_wq 实例，等待进程队列，创建 socket 实例时分配*/
    struct file         *file;          /*指向文件 file 实例*/
    struct sock         *sk;        /*指向 sock 实例*/
    const struct proto_ops  *ops;       /*指向 proto_ops 实例*/
};
```

socket 结构体主要成员简介如下：

●**sk**：指向 sock 实例，套接字在协议层的表示，包含套接字的详细信息。

●**ops**：指向传输协议操作 proto_ops 实例，见下文。

●**file**：指向 file 实例，用户进程创建套接字与创建普通文件类似，也会为其分配文件描述符及相应的 file 实例。

●**state**：套接字（与其它套接字）的连接状态，状态值由枚举类型表示（/include/upai/linux/net.h）：

```
typedef enum {
    SS_FREE = 0,            /*没有被分配？ */
    SS_UNCONNECTED,     /*没有连接到任何套接字*/
    SS_CONNECTING,          /*正在连接*/
    SS_CONNECTED,           /*已经建立连接*/
    SS_DISCONNECTING        /*正在断开连接*/
} socket_state;
```

●**type**：套接字类型，定义如下（/include/linux/net.h）：

```
enum sock_type {
    SOCK_STREAM   = 1,      /*流套接字*/
    SOCK_DGRAM    = 2,      /*数据报套接字*/
    SOCK_RAW  = 3,          /*原始套接字，数据区不加协议报头*/
    SOCK_RDM  = 4,          /*可靠传输消息*/
    SOCK_SEQPACKET    = 5,    /*顺序数据包套接字*/
    SOCK_DCCP = 6,          /*数据报拥塞控制协议套接字*/
    SOCK_PACKET   = 10,     /*已废弃*/
};
```

●**flags**：标记成员，取值定义如下（/include/linux/net.h）：

```
  #define   SOCK_ASYNC_NOSPACE        0
  #define   SOCK_ASYNC_WAITDATA       1
  #define   SOCK_NOSPACE              2
  #define   SOCK_PASSCRED             3
  #define   SOCK_PASSSEC              4
```

●**wq**：指向 socket_wq 结构体实例，定义如下（/include/linux/net.h）：

```
struct socket_wq {
    wait_queue_head_t wait;       /*等待进程队列头，必须是第一个成员*/
    struct fasync_struct *fasync_list;    /*异步操作链表*/
    struct rcu_head         rcu;    /*rcu 链表头*/
} ____cacheline_aligned_in_smp;
```

socket_wq 实例在创建套接字分配 socket 实例时分配。socket_wq 用于管理在套接字上睡眠等待的进程。

socket 实例由内核 sockfs 伪文件系统管理，后面将介绍 sockfs 伪文件系统和 socket 实例的创建。

2 proto_ops

proto_ops 结构体是进程操作套接字的接口，由传输协议代码实现，并在创建套接字时赋予 socket 实例，结构体中主要包含套接字操作函数指针成员。套接字 VFS 系统调用以及专用系统调用都调用此结构体中的函数完成操作。

proto_ops 结构体定义如下（/include/linux/net.h）：

```
struct proto_ops {
    int   family;     /*地址簇标识*/
    struct module  *owner;          /*模块指针*/
    int   (*release)    (struct socket *sock);    /*释放套接字*/
    int   (*bind)       (struct socket *sock,struct sockaddr *myaddr,int sockaddr_len);   /*绑定地址*/
    int   (*connect)    (struct socket *sock,struct sockaddr *vaddr,int sockaddr_len, int flags); /*连接*/
    int   (*socketpair) (struct socket *sock1,struct socket *sock2);
    int   (*accept)     (struct socket *sock,struct socket *newsock, int flags);    /*接受连接*/
    int   (*getname)    (struct socket *sock,struct sockaddr *addr,int *sockaddr_len, int peer);
    unsigned int    (*poll)    (struct file *file, struct socket *sock,struct poll_table_struct *wait);
    int   (*ioctl)      (struct socket *sock, unsigned int cmd,unsigned long arg);   /*套接字控制*/
#ifdef CONFIG_COMPAT
    ...
#endif
    int   (*listen)     (struct socket *sock, int len);    /*监听操作*/
```

```
    int    (*shutdown)    (struct socket *sock, int flags);      /*关闭套接字*/
    int    (*setsockopt)(struct socket *sock, int level,int optname, char __user *optval, unsigned int optlen);
    int    (*getsockopt)(struct socket *sock, int level,int optname, char __user *optval, int __user *optlen);
                                                                  /*设置/获取套接字参数*/
#ifdef CONFIG_COMPAT
    ...
#endif
    int    (*sendmsg)     (struct socket *sock, struct msghdr *m,size_t total_len);   /*发送消息*/
    int    (*recvmsg)     (struct socket *sock, struct msghdr *m,size_t total_len, int flags);     /*接收消息*/
    int    (*mmap)         (struct file *file, struct socket *sock,struct vm_area_struct * vma);   /*映射操作*/
    ssize_t    (*sendpage)    (struct socket *sock, struct page *page,int offset, size_t size, int flags);
    ssize_t    (*splice_read)(struct socket *sock,    loff_t *ppos,
                              struct pipe_inode_info *pipe, size_t len, unsigned int flags);
    int        (*set_peek_off)(struct sock *sk, int val);
};
```

proto_ops 结构体中主要函数指针成员语义如下：

●**bind**：绑定操作，给本套接字设置一个地址，以便通信对方能寻址到本套接字。

●**connect**：与目的套接字建立连接，适用于流套接字。

●**listen**：监听操作，表示本套接字可以接受其它套接字的连接请求。

●**accept**：接受其它套接字的连接请求。

●**sendmsg**：通过套接字向对方发送消息。

●**recvmsg**：从套接字接收消息。

●**ioctl**：套接字控制操作。

●**setsockopt/getsockopt**：设置/获取套接字参数。

3 sock

sock 是一个冗长的数据结构，是套接字在传输协议中的表示，包含了对内核有意义的套接字附加信息，其中最重要的是接收/发送数据包缓存队列。

在地址簇定义的 net_proto_family 实例中的 create()函数将为 socket 实例创建并初始化 sock 实例。sock 结构体通常嵌入到地址簇定义的表示私有数据结构中。

■定义

sock 结构体定义在/include/net/sock.h 头文件，下面简要列出其部分成员：

```
struct sock {
    struct sock_common      __sk_common;      /*/include/net/sock.h*/
                                /*内核管理 sock 实例比较重要的成员，表示套接字通用信息*/

    ...                         /*表示__sk_common 实例中成员的宏定义*/

    socket_lock_t       sk_lock;       /*套接字锁*/
    struct sk_buff_head  sk_receive_queue;    /*接收数据包缓存队列*/

    struct {                        /*后备数据包队列*/
        atomic_t  rmem_alloc;
```

```
            int         len;
            struct sk_buff  *head;    /*记录最先接收到的数据包*/
            struct sk_buff  *tail;    /*记录最后接收到的数据包*/
        } sk_backlog;

#define sk_rmem_alloc sk_backlog.rmem_alloc
        int         sk_forward_alloc;
#ifdef CONFIG_RPS
        __u32       sk_rxhash;
#endif

        u16  sk_incoming_cpu;
        __u32       sk_txhash;

#ifdef CONFIG_NET_RX_BUSY_POLL
        unsigned int        sk_napi_id;
        unsigned int        sk_ll_usec;
#endif
        atomic_t        sk_drops;     /*丢弃数据包数量*/
        int             sk_rcvbuf;    /*接收缓存区长度（字节数）*/

        struct sk_filter __rcu      *sk_filter;    /*数据包过滤器，/include/linux/filter.h*/
        struct socket_wq __rcu  *sk_wq;      /*在套接字上睡眠等待的进程，指向 socket->wq 成员*/

#ifdef CONFIG_XFRM
        struct xfrm_policy  *sk_policy[2];
#endif
        unsigned long       sk_flags;      /*套接字标记*/
        struct dst_entry        *sk_rx_dst;      /*接收路由选择查找结果*/
        struct dst_entry __rcu      *sk_dst_cache;   /*目的路由缓存，路由选择查找结果*/
        spinlock_t          sk_dst_lock;
        atomic_t        sk_wmem_alloc;      /*发送队列的字节数*/
        atomic_t        sk_omem_alloc;      /*可选择的字节数*/
        int             sk_sndbuf;          /*发送缓存区总长度（字节数）*/
        struct sk_buff_head  sk_write_queue;       /*发送数据包缓存队列*/
        kmemcheck_bitfield_begin(flags);
        unsigned int    sk_shutdown   : 2,
                        sk_no_check_tx : 1,     /*发送数据包是否没有校验和*/
                        sk_no_check_rx : 1,     /*接收数据包是否没有校验和*/
                        sk_userlocks : 4,
                        sk_protocol   : 8,      /*协议类型*/
                        sk_type       : 16;   /*套接字类型*/
        kmemcheck_bitfield_end(flags);
        int             sk_wmem_queued;     /*全部数据包占用内存计数*/
        gfp_t           sk_allocation;        /*分配模式*/
        u32             sk_pacing_rate;     /* bytes per second */
```

```
    u32             sk_max_pacing_rate;
    netdev_features_t   sk_route_caps;   /*路由的兼容性标志位*/
    netdev_features_t   sk_route_nocaps;
    int             sk_gso_type;        /*GSO 通用分段类型*/
    unsigned int    sk_gso_max_size;   /*用于建立通用分段 GSO 的最大长度*/
    u16             sk_gso_max_segs;      /*最大段数*/
    int             sk_rcvlowat;
    unsigned long           sk_lingertime;    /*停留时间，确定关闭时间*/
    struct sk_buff_head  sk_error_queue;
    struct proto            *sk_prot_creator;      /*指向 proto 实例，见下文*/
    rwlock_t        sk_callback_lock;
    int             sk_err,     /*出错码*/
                    sk_err_soft;   /*持续出现的错误*/
    u32             sk_ack_backlog;     /*当前监听到的连接数量*/
    u32             sk_max_ack_backlog;
    __u32           sk_priority;     /*SO_PRIORITY 参数设置的优先级*/
#if IS_ENABLED(CONFIG_CGROUP_NET_PRIO)
    __u32           sk_cgrp_prioidx;
#endif
    struct pid      *sk_peer_pid;
    const struct cred   *sk_peer_cred;
    long            sk_rcvtimeo;     /*SO_RCVTIMEO 参数设置接收超时时间*/
    long            sk_sndtimeo;     /*SO_SNDTIMEO 参数设置发送超时时间*/
    struct timer_list   sk_timer;     /*sock 冲刷定时器*/
    ktime_t         sk_stamp;     /*最后接收数据包的时间戳*/
    u16             sk_tsflags;     /*SO_TIMESTAMPING 套接字参数*/
    u32             sk_tskey;
    struct socket   *sk_socket;       /*指向表示套接字的 socket 实例*/
    void            *sk_user_data;   /*RPC 提供的数据*/
    struct page_frag    sk_frag;     /*缓存页段*/
    struct sk_buff      *sk_send_head;     /*发送数据包队列头*/
    __s32               sk_peek_off;
    int             sk_write_pending;     /*等待发送的数量*/
#ifdef CONFIG_SECURITY
    void            *sk_security;
#endif
    __u32           sk_mark;      /*通用数据包掩码*/
    u32             sk_classid;     /*组控制 classid*/
    struct cg_proto     *sk_cgrp;     /*组控制数据*/
    /*回调函数*/
    void            (*sk_state_change)(struct sock *sk);   /*数据包状态改变后调用的函数*/
    void            (*sk_data_ready)(struct sock *sk);     /*通知套接字有新数据包到达*/
    void            (*sk_write_space)(struct sock *sk);     /*当发送空间可以使用后，调用的函数*/
    void            (*sk_error_report)(struct sock *sk);     /*错误处理函数*/
    int             (*sk_backlog_rcv)(struct sock *sk,struct sk_buff *skb);   /*处理库存数据包函数*/
    void           (*sk_destruct)(struct sock *sk);     /*sock 销毁函数*/
```

};

sock 结构体部分成员简介如下：

●**sk_receive_queue、sk_write_queue**：sk_buff_head 结构体成员，表示接收、发送数据包缓存队列头，用于管理 sk_buff 实例。

sk_buff_head 结构体定义如下（/include/linux/skbuff.h）：

```
struct sk_buff_head {
    struct sk_buff * next;   /*指向下一个 sk_buff 实例*/
    struct sk_buff * prev;   /*指向前一个 sk_buff 实例*/

    __u32          qlen;     /*队列数据包数量*/
    spinlock_t     lock;
};
```

●**__sk_common**：sock_common 结构体成员，表示套接字通用属性，结构体定义见下文。

●**sk_prot_creator**：proto 结构体指针，包含传输协议层定义的套接字操作函数，以及 sock 实例的操作函数等，见下文。

●**sk_flags**：套接字标志，取值定义在/include/net/sock.h 头文件：

```
enum sock_flags {
    SOCK_DEAD,
    SOCK_DONE,
    SOCK_URGINLINE,
    SOCK_KEEPOPEN,
    SOCK_LINGER,
    SOCK_DESTROY,
    SOCK_BROADCAST,
    SOCK_TIMESTAMP,
    SOCK_ZAPPED,
    SOCK_USE_WRITE_QUEUE,       /* whether to call sk->sk_write_space in sock_wfree */
    SOCK_DBG,       /* %SO_DEBUG setting */
    SOCK_RCVTSTAMP,       /* %SO_TIMESTAMP setting */
    SOCK_RCVTSTAMPNS,         /* %SO_TIMESTAMPNS setting */
    SOCK_LOCALROUTE,            /* route locally only, %SO_DONTROUTE setting */
    SOCK_QUEUE_SHRUNK,          /* write queue has been shrunk recently */
    SOCK_MEMALLOC,              /* VM depends on this socket for swapping */
    SOCK_TIMESTAMPING_RX_SOFTWARE,    /* %SOF_TIMESTAMPING_RX_SOFTWARE */
    SOCK_FASYNC,       /* fasync() active */
    SOCK_RXQ_OVFL,
    SOCK_ZEROCOPY,     /* buffers from userspace */
    SOCK_WIFI_STATUS,    /* push wifi status to userspace */
    SOCK_NOFCS,
    SOCK_FILTER_LOCKED,    /* Filter cannot be changed anymore */
    SOCK_SELECT_ERR_QUEUE,    /* Wake select on error queue */
};
```

●sock_common

sock_common 结构体表示各传输协议套接字的通用属性，定义如下（/include/net/sock.h）：

```
struct sock_common {
```

```
    union {
        __addrpair      skc_addrpair;
        struct {
            __be32      skc_daddr;      /*目的主机 IP 地址（IPv4）*/
            __be32      skc_rcv_saddr;     /*绑定的本地（主机）IP 地址*/
        };
    };
    union   {
        unsigned int    skc_hash;       /*散列值*/
        __u16           skc_u16hashes[2];     /*两个散列值，用于 UDP 套接字*/
    };

    union {
        __portpair      skc_portpair;
        struct {
            __be16      skc_dport;      /*目的端口号，网络字节序*/
            __u16       skc_num;        /*端口号，本机字节序*/
        };
    };

    unsigned short      skc_family;       /*地址簇*/
    volatile unsigned char    skc_state;    /*连接状态*/
    unsigned char       skc_reuse:4;      /*确定复用地址，SO_REUSEADDR 参数设置*/
    unsigned char       skc_reuseport:1;   /*SO_REUSEPORT 参数设置*/
    unsigned char       skc_ipv6only:1;
    unsigned char       skc_net_refcnt:1;
    int             skc_bound_dev_if;       /*绑定网络设备 ID*/
    union {
        struct hlist_node     skc_bind_node;   /*将 sock 实例插入 proto 实例中散列表*/
        struct hlist_nulls_node skc_portaddr_node; /*用于 UDP/UDP-Lite 套接字插入第二个散列表*/
    };
    struct proto        *skc_prot;     /*指向 proto 实例*/
    possible_net_t      skc_net;      /*所属网络命名空间*/

#if IS_ENABLED(CONFIG_IPV6)
    struct in6_addr         skc_v6_daddr;
    struct in6_addr         skc_v6_rcv_saddr;
#endif

    atomic64_t          skc_cookie;
    int             skc_dontcopy_begin[0];

    union {
        struct hlist_node     skc_node;     /*用于将 sk_buff 插入主散列表*/
        struct hlist_nulls_node skc_nulls_node;   /*将 TCP/UDP/UDP-Lite  套接字链入第一个散列表*/
    };
```

```
    int             skc_tx_queue_mapping;    /*本连接发送队列数量*/
    atomic_t        skc_refcnt;      /*引用计数*/
    int             skc_dontcopy_end[0];
};
```

sock_common 结构体主要还是包含 IPv4 和 IPv6 套接字的信息。

●proto

proto 结构体表示子传输协议的操作接口，以及 sock 实例的操作函数和管理散列表。

proto 结构体定义如下（/include/net/sock.h）：

```
struct proto {
    /*套接字操作接口函数，对应 proto_ops 结构体中函数*/
    void   (*close)(struct sock *sk,long timeout);        /*关闭套接字*/
    int    (*connect)(struct sock *sk,struct sockaddr *uaddr,int addr_len);   /*连接操作*/
    int    (*disconnect)(struct sock *sk, int flags);      /*断开连接*/
    struct sock *    (*accept)(struct sock *sk, int flags, int *err);     /*接受连接请求*/
    int         (*ioctl)(struct sock *sk, int cmd,unsigned long arg);   /*套接字控制*/
    int         (*init)(struct sock *sk);        /*初始化 sock 实例*/
    void        (*destroy)(struct sock *sk);     /*销毁套接字*/
    void        (*shutdown)(struct sock *sk, int how);
    int    (*setsockopt)(struct sock *sk, int level,int optname, char __user *optval,unsigned int optlen);
    int    (*getsockopt)(struct sock *sk, int level,int optname, char __user *optval,int __user *option);
                                                          /*设置/获取套接字参数*/
#ifdef CONFIG_COMPAT
    ...
#endif
    int    (*sendmsg)(struct sock *sk, struct msghdr *msg,size_t len);     /*发送消息*/
    int    (*recvmsg)(struct sock *sk, struct msghdr *msg,size_t len, int noblock, int flags,int *addr_len);
    int    (*sendpage)(struct sock *sk, struct page *page,int offset, size_t size, int flags);
    int    (*bind)(struct sock *sk,struct sockaddr *uaddr, int addr_len);     /*绑定操作*/
    int    (*backlog_rcv) (struct sock *sk,struct sk_buff *skb);
    void   (*release_cb)(struct sock *sk);     /*释放控制块*/

    /*sock 实例操作函数，如插入散列表、移出散列表等*/
    void        (*hash)(struct sock *sk);        /*插入散列表*/
    void        (*unhash)(struct sock *sk);    /*从散列表移出*/
    void        (*rehash)(struct sock *sk);
    int         (*get_port)(struct sock *sk, unsigned short snum);     /*在散列表中查找 xxx_sock 实例*/
    void        (*clear_sk)(struct sock *sk, int size);

    /* Keeping track of sockets in use */
  #ifdef CONFIG_PROC_FS
    unsigned int            inuse_idx;
  #endif

    bool                (*stream_memory_free)(const struct sock *sk);
    void                (*enter_memory_pressure)(struct sock *sk);
```

```
    atomic_long_t       *memory_allocated;      /* Current allocated memory. */
    struct percpu_counter    *sockets_allocated;/* Current number of sockets. */
    int                 *memory_pressure;
    long                *sysctl_mem;
    int                 *sysctl_wmem;
    int                 *sysctl_rmem;
    int                 max_header;
    bool                no_autobind;

    struct kmem_cache   *slab;          /*xxx_sock 结构体 slab 缓存*/
    unsigned int        obj_size;     /*xxx_sock 结构体大小*/
    int                 slab_flags;         /*slab 缓存分配标记*/

    struct percpu_counter    *orphan_count;
    struct request_sock_ops *rsk_prot;      /*连接请求操作结构*/
    struct timewait_sock_ops *twsk_prot;

    union {             /*管理套接字 xxx_sock 实例的散列表*/
        struct inet_hashinfo  *hashinfo;      /*指向管理 TCP 套接字 tcp_sock 实例散列表*/
        struct udp_table      *udp_table;     /*指向管理 UDP 套接字散列表*/
        struct raw_hashinfo  *raw_hash;       /*指向管理原始套接字散列表*/
    } h;

    struct module       *owner;
    char                name[32];       /*套接字名称，用于 sockfs 中的目录项名称*/
    struct list_head    node;           /*将 proto 实例添加到全局双链表 proto_list*/
  #ifdef SOCK_REFCNT_DEBUG
    atomic_t        socks;
  #endif
  #ifdef CONFIG_MEMCG_KMEM
    ...
  #endif
};
```

proto 结构体主要实现两个功能：一是实现传输协议层中对套接字的操作，上接 proto_ops 结构体，二是对 sock 实例进行管理。

proto_ops 结构体中套接字操作函数将会调用 proto 结构体中的对应函数完成操作。通常 sock 被嵌入到传输协议定义的私有数据结构中，形如 xxx_sock，slab 成员指向的缓存用于分配 xxx_sock 实例。联合体成员 h 表示散列表，用于管理 xxx_sock 实例。

传输协议相关代码需要定义并注册 proto 实例，注册函数为 **proto_register(struct proto *prot, int alloc_slab)**，函数定义在/net/core/sock.c 文件内，alloc_slab 参数表示是否为私有数据创建 slab 缓存。注册函数主要工作是为私有数据结构创建 slab 缓存并赋予 slab 成员，缓存对象大小为 obj_size，并将 proto 实例插入到全局双链表 **proto_list** 等。

创建套接字时，在协议簇定义的 net_proto_family 实例中的 create()函数中，将从 proto->slab 指向的 slab 缓存中分配 xxx_sock 实例，并调用 proto->hash()函数将实例插入散列表，调用 proto->init()函数对实例进行初始化。

另外，内核通过 procfs 向用户导出 proto 实例的信息，相关代码在/net/core/sock.c 文件内实现，请读者自行阅读。

■分配 sock 实例

在地址簇定义的 net_proto_family 实例中的 create()函数中将为 socket 实例创建关联的 sock 实例，创建 sock 实例的接口函数为 **sk_alloc()**，定义如下（/net/core/sock.c）：

```
struct sock *sk_alloc(struct net *net, int family, gfp_t priority,struct proto *prot, int kern)
/*
*net：指向网络命名空间，family：地址簇标识，prot：指向 proto 实例，
*priority：内存分配标记，kern：是否是内核套接字。
*/
{
    struct sock *sk;

    sk = sk_prot_alloc(prot, priority | __GFP_ZERO, family);
                                    /*从 prot->slab 缓存中分配特定于传输协议的 xxx_sock 实例*/
    if (sk) {           /*初始化 sock 实例*/
        sk->sk_family = family;      /*地址簇标识*/

        sk->sk_prot = sk->sk_prot_creator = prot;     /*指向 proto 实例*/
        sock_lock_init(sk);
        sk->sk_net_refcnt = kern ? 0 : 1;     /*是否增加网络命名空间引用计数，只适用于用户套接字*/
        if (likely(sk->sk_net_refcnt))
            get_net(net);
        sock_net_set(sk, net);     /*设置网络命名空间*/
        atomic_set(&sk->sk_wmem_alloc, 1);

        sock_update_classid(sk);
        sock_update_netprioidx(sk);
    }
    return sk;     /*返回 sock 实例指针*/
}
```

在创建 sock 实例时，需要传递关联的 proto 实例指针。sk_alloc()函数从 proto 实例中的 slab 缓存中分配 xxx_sock 实例，此实例中包含 sock 实例（结构体中第一个成员），然后对 sock 实例进行初始化，返回 sock 实例指针。

■初始化 sock 实例

sock_init_data(sock, sk)函数用于对 sock 实例进行进一步的初始化，net_proto_family 实例中 create()函数需要调用此函数，函数定义在/net/core/sock.c 文件内，代码如下：

```
void sock_init_data(struct socket *sock, struct sock *sk)
{
    skb_queue_head_init(&sk->sk_receive_queue);     /*初始化 sk_buff 队列头*/
    skb_queue_head_init(&sk->sk_write_queue);
    skb_queue_head_init(&sk->sk_error_queue);
```

```
sk->sk_send_head  =    NULL;

init_timer(&sk->sk_timer);    /*初始化套接字定时器*/

sk->sk_allocation  =    GFP_KERNEL;
sk->sk_rcvbuf      =    sysctl_rmem_default;
                          /*接收缓存区长度，初始值为 64KB*/
                        /*套接字可通过 SO_RCVBUFFORCE 参数（SOL_SOCKET 级别）修改*/
sk->sk_sndbuf      =    sysctl_wmem_default;    /*发送缓存区长度，初始值 64KB*/
                                /*套接字可通过 SO_SNDBUFFORCE 参数修改*/
sk->sk_state       =    TCP_CLOSE;          /*TCP 套接字，初始为关闭状态*/
sk_set_socket(sk, sock);    /*关联 socket 实例*/

sock_set_flag(sk, SOCK_ZAPPED);        /*设置标记*/

if (sock) {
      sk->sk_type   =    sock->type;
      sk->sk_wq     =    sock->wq;
      sock->sk =    sk;      /*socket 关联 sock 实例*/
} else
      sk->sk_wq     =    NULL;

spin_lock_init(&sk->sk_dst_lock);
rwlock_init(&sk->sk_callback_lock);
lockdep_set_class_and_name(&sk->sk_callback_lock,
      af_callback_keys + sk->sk_family,
      af_family_clock_key_strings[sk->sk_family]);

sk->sk_state_change    =    sock_def_wakeup;
sk->sk_data_ready =    sock_def_readable;
sk->sk_write_space=    sock_def_write_space;
sk->sk_error_report    =    sock_def_error_report;
sk->sk_destruct        =    sock_def_destruct;

sk->sk_frag.page   =    NULL;
sk->sk_frag.offset  =    0;
sk->sk_peek_off        =    -1;

sk->sk_peer_pid    =    NULL;
sk->sk_peer_cred   =    NULL;
sk->sk_write_pending   =    0;
sk->sk_rcvlowat        =    1;
sk->sk_rcvtimeo        =    MAX_SCHEDULE_TIMEOUT;    /*接收超时时间*/
sk->sk_sndtimeo        =    MAX_SCHEDULE_TIMEOUT;    /*发送超时时间*/
```

```
    sk->sk_stamp = ktime_set(-1L, 0);

#ifdef CONFIG_NET_RX_BUSY_POLL
    sk->sk_napi_id          =    0;
    sk->sk_ll_usec          =    sysctl_net_busy_read;
#endif

    sk->sk_max_pacing_rate = ~0U;
    sk->sk_pacing_rate = ~0U;

    smp_wmb();
    atomic_set(&sk->sk_refcnt, 1);
    atomic_set(&sk->sk_drops, 0);
}
```

4 数据包

数据包由 sk_buff 结构体表示，sk_buff 关联一个内存缓存区，用于保存数据和报头信息。套接字中接收/发送数据包缓存队列由 sk_buff 实例组成，数据以 sk_buff 实例的形式在协议栈中传输。

■sk_buff

sk_buff 结构体定义如下（/include/linux/skbuff.h）：

```
struct sk_buff {
    union {      /*联合体*/
        struct {
            struct sk_buff  *next;      /*前两成员用于构成 sk_buff 实例队列*/
            struct sk_buff  *prev;

            union {
                ktime_t         tstamp;
                struct skb_mstamp skb_mstamp;
            };
        };

        struct rb_node rbnode;   /*红黑树节点，用于插入管理红黑树*/
    };    /*联合体结束*/

    struct sock         *sk;        /*指向 sock 实例*/
    struct net_device   *dev;       /*指向网络设备 net_device 实例*/
    char            cb[48]   __aligned(8);      /*数据包控制块，由传输协议使用*/

    unsigned long   _skb_refdst;   /*指向路由查找结果 dst_entry 实例*/
    void            (*destructor)(struct sk_buff *skb);     /*释放 sk_buff 实例时调用的函数，析构函数*/
#ifdef CONFIG_XFRM
    struct      sec_path  *sp;
```

```
#endif
#if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
    struct nf_conntrack *nfct;
#endif
#if IS_ENABLED(CONFIG_BRIDGE_NETFILTER)
    struct nf_bridge_info    *nf_bridge;
#endif
    unsigned int        len,    /*数据包的总长度，含分散数据块和分段数据包*/
                data_len;    /*只用非线性数据包，表示分散数据块或分段数据包中数据长度*/
    __u16           mac_len,    /*数据链路层报头长度*/
                hdr_len;    /*克隆 sk_buff 实例时可写的报头长度*/

    kmemcheck_bitfield_begin(flags1);
    __u16           queue_mapping;    /*用于多队列设备的队列映射*/
    __u8            cloned:1,   /*在被克隆和克隆得到的数据包中设置本标记位*/
                nohdr:1,     /*只参考有效载荷，禁止修改报头*/
                fclone:2,
                peeked:1,   /*这个数据包以前遇到过，已经根据它更新过统计信息，不再更新*/
                head_frag:1,
                xmit_more:1;
    kmemcheck_bitfield_end(flags1);

    __u32    headers_start[0];

#ifdef __BIG_ENDIAN_BITFIELD
    #define PKT_TYPE_MAX    (7 << 5)
#else
    #define PKT_TYPE_MAX    7
#endif
    #define PKT_TYPE_OFFSET()    offsetof(struct sk_buff, __pkt_type_offset)

    __u8    __pkt_type_offset[0];
    __u8    pkt_type:3;         /*数据包类型*/
    __u8    pfmemalloc:1;       /*从 PFMEMALLOC 保留区分配 SKB*/
    __u8    ignore_df:1;    /*是否允许本地分段*/
    __u8    nfctinfo:3;

    __u8    nf_trace:1;    /*Netfilter 数据包跟踪标志*/
    __u8    ip_summed:2;   /*IP（第 3 层）校验和指示器*/
    __u8    ooo_okay:1;    /*设置此位，以避免数据包不按顺序排列*/
    __u8    l4_hash:1;     /*设置此位，表示由四元组计算得散列值*/
    __u8    sw_hash:1;
    __u8    wifi_acked_valid:1;
    __u8    wifi_acked:1;

    __u8    no_fcs:1;    /*让网络接口将最后 4 个字节视为以太网帧在校验序列*/
```

```
    /* Indicates the inner headers are valid in the skbuff. */
    __u8        encapsulation:1;     /*指出 SKB 是用于封装的*/
    __u8        encap_hdr_csum:1;
    __u8        csum_valid:1;
    __u8        csum_complete_sw:1;
    __u8        csum_level:2;
    __u8        csum_bad:1;

#ifdef CONFIG_IPV6_NDISC_NODETYPE
    __u8        ndisc_nodetype:2;
#endif
    __u8        ipvs_property:1;
    __u8        inner_protocol_type:1;
    __u8        remcsum_offload:1;

#ifdef CONFIG_NET_SCHED
    __u16       tc_index; /* traffic control index */
  #ifdef CONFIG_NET_CLS_ACT
    __u16       tc_verd;   /* traffic control verdict */
  #endif
#endif

    union {
        __wsum  csum;       /*校验和*/
        struct {
            __u16       csum_start;
            __u16       csum_offset;
        };
    };
    __u32           priority;       /*数据包排队优先级*/
    int             skb_iif;        /*网络设备索引值*/
    __u32           hash;           /*散列值*/
    __be16          vlan_proto;
    __u16           vlan_tci;
#if defined(CONFIG_NET_RX_BUSY_POLL) || defined(CONFIG_XPS)
    union {
        unsigned int    napi_id;
        unsigned int    sender_cpu;
    };
#endif
#ifdef CONFIG_NETWORK_SECMARK
    __u32               secmark;
#endif
    union {
        __u32           mark;     /*通过标记来标识 SKB*/
        __u32           reserved_tailroom;
```

```
    };

    union {
        __be16          inner_protocol;
        __u8            inner_ipproto;
    };

    __u16           inner_transport_header;
    __u16           inner_network_header;
    __u16           inner_mac_header;

    __be16          protocol;       /*协议类型，如 ETH_P_IP*/
    __u16           transport_header;   /*传输层协议报头相对于 head 的偏移量*/
    __u16           network_header;     /*网络层协议报头相对于 head 的偏移量*/
    __u16           mac_header;         /*数据链路层协议报头相对于 head 的偏移量*/
                        /*以上三个成员初始值为（data-head），即报头指向数据区开头*/
    __u32           headers_end[0];

    sk_buff_data_t      tail;     /*缓存区数据末尾地址*/
    sk_buff_data_t      end;      /*缓存区的结束位置地址*/
    unsigned char       *head,    /*指向缓存区的起始位置*/
                        *data;    /*指向缓存区数据的起始位置*/
    unsigned int        truesize;     /*内存缓存区实际总大小*/
    atomic_t        users;      /*引用计数*/
};
```

sk_buff 结构体部分成员简介如下（许多成员都是用于 IPv4 和 IPv6 协议）：

●**sk**：指向 sock 实例。对于本地生成的流量或发送给当前主机的流量，sk 为拥有 SKB（指内存缓存区）的套接字，对于需要转发的数据包，sk 为 NULL。

●**cb[48]**：控制块，由传输协议使用，用于存储专用信息。例如：TCP 协议使用以下宏获取控制块：

```
#define   TCP_SKB_CB(__skb)   ((struct tcp_skb_cb *)&((__skb)->cb[0]))       /*/include/net/tcp.h*/
                                                        /*专用信息由 tcp_skb_cb 结构体表示*/
```

●**_skb_refdst**：路由选择表项（dst_entry）地址。dst_entry 表示给定目的地的路由选择表项。对于每个数据包，都需要执行路由选择表查找。这种查找有时被称为 FIB 查找。查找结果决定了应如何处理数据包，如是否需要转发、是否应该丢弃、是否要发送 ICMP 错误消息等。

struct dst_entry *skb_dst(const struct sk_buff *skb)函数用于获取 sk_buff 实例关联的 dst_entry 实例。

void skb_dst_set(struct sk_buff *skb, struct dst_entry *dst)函数用于设置 sk_buff 实例关联的 dst_entry 实例，实例内引用计数被使用。解除关联时需要调用 skb_dst_drop()函数释放。

void skb_dst_set_noref(struct sk_buff *skb, struct dst_entry *dst)函数用于设置 sk_buff 关联的 dst_entry 实例，实例内引用计数不使用。

●**pkt_type**：数据包类型，取值定义如下（/include/uapi/linux/if_packet.h）：

```
#define   PACKET_HOST           0       /*传递给当前主机的数据包，不外发*/
#define   PACKET_BROADCAST      1       /*广播数据包*/
#define   PACKET_MULTICAST      2       /*组播数据包*/
#define   PACKET_OTHERHOST      3       /*上述条件都不满足*/
#define   PACKET_OUTGOING       4       /* Outgoing of any type */
#define   PACKET_LOOPBACK       5       /* MC/BRD frame looped back，环回数据包*/
```

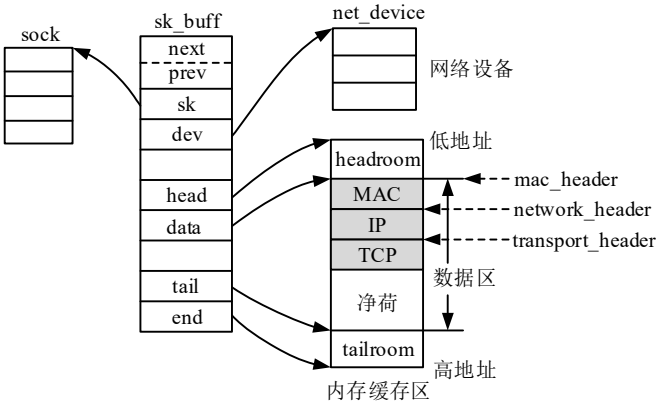
```
#define   PACKET_USER          6          /*传递给用户空间*/
#define   PACKET_KERNEL        7          /*传递给内核空间*/
```

●**users**：引用计数值，初始值为 1。skb_get()函数将其加 1，而 kfree_skb()和 consume_skb()函数将其值减 1。kfree_skb()函数在计数值为 0 时将释放 SKB。

int **skb_shared**(const struct sk_buff *skb)：如果 users 值不为 1 则返回 true，表明数据包被共享。

struct sk_buff ***skb_share_check**(struct sk_buff *skb, gfp_t pri)：检查数据包是否被共享，如果不共享则返回原数据包 skb 实例，否则克隆数据包，返回克隆数据包 skb_buff 实例。克隆与被克隆数据包共用内存缓存区，原内存缓存区引用计数加 1。在中断上下文中调用或持有自旋锁时，参数 pri 必须为 GFP_ATOMIC。如果内存分配失败返回 NULL。

在创建 sk_buff 时，sk_buff 实例和内存缓存区是分开分配的。内存缓存区中包含数据区，数据区是各层协议的报头和净荷。内存缓存区头尾还有预留空间，称为 headroom 和 tailroom，head 成员指向缓存区开头，end 指向缓存区末尾，data 指向数据区开头，tail 指向数据区末尾，如下图所示：



内核在/include/linux/skbuff.h 头文件内定义了 sk_buff 实例中成员的操作接口函数，例如：

●unsigned char ***skb_transport_header**(const struct sk_buff *skb)：返回传输层报头指针（地址）。

●void **skb_set_transport_header**(struct sk_buff *skb,const int offset)：设置传输层报头指针，offset 为相对于数据区开头（data）的偏移量。只设置指针，没有填充数据。

●unsigned char ***skb_network_header**(const struct sk_buff *skb)：返回网络层报头指针（地址）。

●void **skb_set_network_header**(struct sk_buff *skb, const int offset)：设置网络层报头指针，offset 为相对于数据区开头（data）的偏移量。

●unsigned char ***skb_mac_header**(const struct sk_buff *skb)：返回数据链路层报头指针（地址）。

●void **skb_set_mac_header**(struct sk_buff *skb, const int offset)：设置数据链路层报头指针，offset 为相对于数据区开头（data）的偏移量。

●unsigned int **skb_headroom**(const struct sk_buff *skb)：返回 headroom 区域大小，字节数。

●int **skb_tailroom**(const struct sk_buff *skb)：返回 tailroom 区域大小，字节数。

●unsigned char ***skb_push**(struct sk_buff *skb, unsigned int len)：将 skb->data 向前（低地址）移 len 字节，数据区长度增加 len 字节，返回新 skb->data 值，即在数据区头部扩展新空间的开头。只是增加空间，没有填充数据。

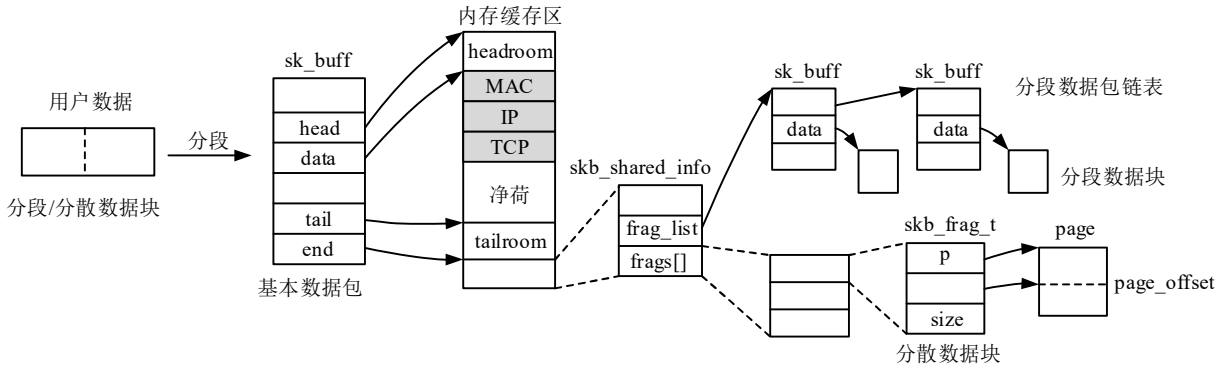
●unsigned char ***skb_pull**(struct sk_buff *skb, unsigned int len)：从数据区头部 skb->data 缩减 len 字节空间，返回新的 skb->data 值。

●unsigned char ***skb_put**(struct sk_buff *skb, unsigned int len)：将 skb->tail 向后（高地址）移 len 字节，数据区长度增加 len 字节，返回新扩展区域开头地址，即原 skb->tail 值。

●void **skb_trim**(struct sk_buff *skb, unsigned int len)：设置新 skb->tail 值，len 为相对于 data 的偏移量。

■内存缓存区

内核在发送、接收数据时需要为其创建 sk_buff 实例，并分配内存缓存区，用于保存传输的数据和报头信息。内核在分配内存缓存区的最后面会附上共享结构体 skb_shared_info 实例，如下图所示：



用户数据如果需要发送到网络上，网络接口（设备）对数据包的长度有限制。如在 10/100/1000Mb/s 以太网中，通常为 1500B（MTU），但有些网络接口允许 9KB 的巨型帧。发送长于 MTU 的数据时，就需要对数据进行分段。

在对用户数据分段时，将根据网络接口是否支持分散/聚集（Scatter/Gather，是否设置 NETIF_F_SG 标记），采用两种不同的分段处理方式。普通网络设备 DMA 要求物理内存中存放数据的区域必需是连续的，支持分散/聚集功能的网络设备，其 DMA 可从分散（不连续）的物理内存中读取/写入数据，这可以大大地提高发送效率。

如果支持分散/聚集功能，用户数据将保存在同一个 sk_buff 实例的分散数据块中，skb_shared_info 结构体中 frags[]数组指示了分散数据块。如果不支持分散/聚集功能，用户数据将划分成分段数据块，每个分段数据块由数据包 sk_buff 实例表示，由 skb_shared_info 结构体中 frag_list 链表管理。

skb_shared_info 结构体定义在/include/linux/skbuff.h 头文件，表示内存缓存区的共享信息：

```
struct skb_shared_info {
    unsigned char  nr_frags;   /*分散数据块数量，即 frags[]数组项数*/
    __u8           tx_flags;   /*标志*/
    unsigned short   gso_size;   /*分散数据块总长度*/
    unsigned short   gso_segs;   /**/
    unsigned short   gso_type;   /**/
    struct sk_buff  *frag_list;   /*分段数据包队列*/
    struct skb_shared_hwtstamps   hwtstamps;      /*硬件时间戳*/
    u32        tskey;
    __be32     ip6_frag_id;
    atomic_t  dataref;   /*引用计数，初始值为 1*/
    void *     destructor_arg;
    skb_frag_t     frags[MAX_SKB_FRAGS];   /*分散数据块*/
                        /*如果网络设备支持分散/聚集，/include/linux/skbuff.h*/
};
```

skb_shared_info 结构体部分成员简介如下：

●**nr_frags**：frags[]数组中表示分散数据块的数量。

●**tx_flags**：标记，取值定义如下（/include/linux/skbuff.h）：

```
enum {
    SKBTX_HW_TSTAMP = 1 << 0,      /*生成一个硬件时间戳*/
    SKBTX_SW_TSTAMP = 1 << 1,      /*生成一个软件时间戳*/
```

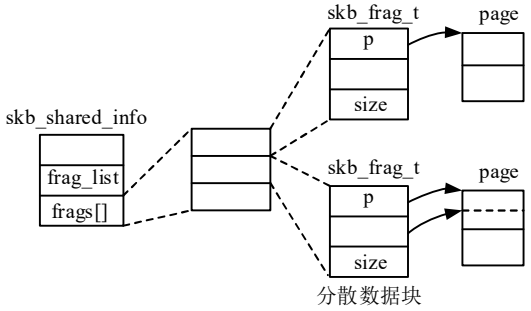
```
    SKBTX_IN_PROGRESS = 1 << 2,   /*设备驱动程序将提供一个硬件时间戳*/
    SKBTX_DEV_ZEROCOPY = 1 << 3,  /*设备驱动程序在发送路径上支持零拷贝缓存区*/
    SKBTX_WIFI_STATUS = 1 << 4,       /*生成 WiFi 状态信息*/
    SKBTX_SHARED_FRAG = 1 << 5,   /*至少有一个分段可能被覆盖*/
    SKBTX_SCHED_TSTAMP = 1 << 6,  /*当进入数据包调度时，产生一个软件时间戳*/
    SKBTX_ACK_TSTAMP = 1 << 7,
};
```

●**frag_list**：分段数据包队列，sk_buff 实例链表。

●**frags**[]：skb_frag_t 结构体数组，skb_frag_t 结构体表示分散数据块，定义如下（/include/linux/skbuff.h）：

```
typedef struct skb_frag_struct    skb_frag_t;
struct skb_frag_struct {
    struct {
        struct page *p;     /*指向页 page 实例*/
    } page;
  #if (BITS_PER_LONG > 32) || (PAGE_SIZE >= 65536)
    __u32 page_offset;
    __u32 size;
  #else
    __u16 page_offset;      /*分散数据块当前结束字节（加 1）在页内的起始偏移量，其后是空闲区*/
    __u16 size;             /*分散数据块中已写入数据长度*/
  #endif
};
```

MAX_SKB_FRAGS 表示 frags[]数组项数，即分散数据块最大数量，由物理页大小决定，保证能容下 64KB 的用户数据。



每个分散数据块由若干个连续的物理内存页组成，32 位系统，为 32KB（8 个内存页，若页大小为 4KB），由参数 SKB_FRAG_PAGE_ORDER 定义（/net/core/sock.c）。用户数据填入分散数据块时，将连续的填充，不留空区域。

使用分散数据块的数据包，或者 frag_list 链表不空的数据包，称为非线性数据包（skb->data_len 非零）。

skb_shared_info 结构体相关辅助方法定义在/include/linux/skbuff.h 头文件，例如：

skb_shinfo(SKB)：返回内存缓存区 skb_shared_info 实例指针，SKB 为 sk_buff 实例指针。

bool **skb_is_gso**(const struct sk_buff *skb)：gso_size 成员值为不为 0 时，返回 true，是否有分散数据块。

bool **skb_has_frag_list**(const struct sk_buff *skb)：返回 frag_list 列表是否不为空，是否有分段数据包。

■**分配 sk_buff 实例**

分配 sk_buff 实例的接口函数为 **alloc_skb()**，定义如下（/include/linux/skbuff.h）：

```
static inline struct sk_buff *alloc_skb(unsigned int size,gfp_t priority)
/*size：内存缓存区大小，实际分配大小要加上 skb_shared_info 实例大小，priority：内存分配掩码*/
{
    return __alloc_skb(size, priority, 0, NUMA_NO_NODE);
}
```

__alloc_skb()函数定义如下（/net/core/skbuff.c）：

```
struct sk_buff *__alloc_skb(unsigned int size, gfp_t gfp_mask,int flags, int node)
/*flags：标记，这里为 0（指示从 sk_buff 结构体 slab 缓存分配），node：NUMA 节点*/
{
    struct kmem_cache *cache;
    struct skb_shared_info *shinfo;
    struct sk_buff *skb;
    u8 *data;
    bool pfmemalloc;

    cache = (flags & SKB_ALLOC_FCLONE)? skbuff_fclone_cache : skbuff_head_cache; /*slab 缓存*/

    if (sk_memalloc_socks() && (flags & SKB_ALLOC_RX))
        gfp_mask |= __GFP_MEMALLOC;

    /*从 slab 缓存分配 sk_buff 实例*/
    skb = kmem_cache_alloc_node(cache, gfp_mask & ~__GFP_DMA, node);
    ...
    prefetchw(skb);

    size = SKB_DATA_ALIGN(size);     /*缓存区大小对齐*/
    size += SKB_DATA_ALIGN(sizeof(struct skb_shared_info));   /*加上 skb_shared_info 结构体大小*/
    data = kmalloc_reserve(size, gfp_mask, node, &pfmemalloc);   /*分配内存缓存区*/
    ...
    size = SKB_WITH_OVERHEAD(ksize(data));     /*缓存区减去 skb_shared_info 实例的大小*/
    prefetchw(data + size);
    memset(skb, 0, offsetof(struct sk_buff, tail));       /*内存缓存区清零*/
    skb->truesize = SKB_TRUESIZE(size);     /*内存缓存区真实大小，不含 skb_shared_info 实例*/
    skb->pfmemalloc = pfmemalloc;
    atomic_set(&skb->users, 1);     /*缓存区引用计数初始化为 1*/
    skb->head = data;     /*指向内存缓存区开始处*/
    skb->data = data;       /*指向内存缓存区开始处*/
    skb_reset_tail_pointer(skb);       /*skb->tail = skb->data*/
    skb->end = skb->tail + size;       /*指向缓存区末尾，后面还有 skb_shared_info 实例*/
    skb->mac_header = (typeof(skb->mac_header))~0U;
    skb->transport_header = (typeof(skb->transport_header))~0U;
```

```
    shinfo = skb_shinfo(skb);     /*skb_shared_info 实例清零*/
    memset(shinfo, 0, offsetof(struct skb_shared_info, dataref));
    atomic_set(&shinfo->dataref, 1);       /*引用计数初始化为 1*/
    kmemcheck_annotate_variable(shinfo->destructor_arg);

    if (flags & SKB_ALLOC_FCLONE) {     /*此处 flags 为 0*/
        ...
    }
out:
    return skb;     /*返回 sk_buff 实例指针*/
    ...
}
```

内核在/net/core/skbuff.c 文件内还定义了其它 sk_buff 实例的操作接口函数，例如：

●struct sk_buff ***skb_copy**(const struct sk_buff *skb, gfp_t gfp_mask)：复制 sk_buff 实例及内存缓存区，新实例是旧实例的完全拷贝。

●struct sk_buff ***skb_clone_sk**(struct sk_buff *skb)：创建 sk_buff 副本（克隆），新旧实例共用内存缓存，关联到共同的 sock 实例。

●struct sk_buff ***skb_clone**(struct sk_buff *skb, gfp_t gfp_mask)：创建 sk_buff 副本，新实例是独立的，不关联到旧实例关联的 sock 实例。

●void **skb_queue_head**(struct sk_buff_head *list, struct sk_buff *newsk)：将 sk_buff 实例添加到队列头部。

●void **skb_queue_tail**(struct sk_buff_head *list, struct sk_buff *newsk)：将 sk_buff 实例添加到队列末尾。

12.2.4 创建套接字

在了解了套接字相关的数据结构后，本小节介绍套接字的创建。

套接字在 VFS 中由 socket 结构体表示，sock 结构体是套接字在协议层的表示。在协议簇定义并注册的 net_proto_family 实例的 create()函数中创建并设置 sock 实例（嵌入到 xxx_sock 结构体中）。sock 实例通常由协议类型定义的 proto 实例管理。

内核通过 sockfs 伪文件系统管理 socket 实例，用户进程通过 socket()系统调用创建套接字，系统调用将从 sockfs 伪文件系统中分配套接字 socket 实例，调用地址簇注册的 net_proto_family 实例中的 create()函数创建并设置 socket 实例，最后为套接字分配 file 实例和文件描述符，返回表示套接字的文件描述符给用户进程。

1 socket 实例管理

内核通过 sockfs 伪文件系统来管理 socket 实例，类似于块设备层通过 bdev 伪文件系统管理表示块设备的 block_device 实例。sockfs 伪文件系统类型定义在/net/socket.c 文件内。

■sockfs 伪文件系统

sockfs 伪文件系统由 socket_alloc 结构体实例组成，结构体中封装了 socket 和 inode 结构体成员，结构体定义如下（/include/net/sock.h）：

```
struct socket_alloc {
    struct socket socket;       /*套接字 socket 结构体*/
    struct inode vfs_inode;     /*VFS 中文件节点 inode 结构体*/
};
```

内核在套接字初始化函数 sock_init()中，将为 socket_alloc 结构体创建了 slab 缓存。

sockfs 伪文件系统类型定义如下：

```
static struct file_system_type sock_fs_type = {
    .name =         "sockfs",       /*伪文件系统名称*/
    .mount =  sockfs_mount,    /*挂载函数*/
    .kill_sb = kill_anon_super,
};
```

挂载函数 sockfs_mount()定义如下（/net/socket.c）：

```
static struct dentry *sockfs_mount(struct file_system_type *fs_type,int flags, const char *dev_name,
                                                                                    void *data)
{
    return mount_pseudo(fs_type, "socket:", &sockfs_ops,&sockfs_dentry_operations, SOCKFS_MAGIC);
                                    /*挂载伪文件系统，文件系统超级块操作结构实例为 sockfs_ops。*/
}
```

sockfs 伪文件系统超级块操作结构实例 sockfs_ops 定义如下：

```
static const struct super_operations sockfs_ops = {
    .alloc_inode    = sock_alloc_inode,       /*分配 socket_alloc 实例的函数*/
    .destroy_inode= sock_destroy_inode,
    .statfs             = simple_statfs,
};
```

超级块操作结构中的分配节点函数 alloc_inode()为**sock_alloc_inode()**函数，用于分配 socket_alloc 实例，函数定义如下：

```
static struct inode *sock_alloc_inode(struct super_block *sb)
{
    struct socket_alloc *ei;
    struct socket_wq *wq;

    ei = kmem_cache_alloc(sock_inode_cachep, GFP_KERNEL); /*从 slab 缓存分配 socket_alloc 实例*/
    ...
    wq = kmalloc(sizeof(*wq), GFP_KERNEL);       /*分配 socket_wq 实例*/
    ...
    init_waitqueue_head(&wq->wait);       /*初始化等待队列头*/
    wq->fasync_list = NULL;
    RCU_INIT_POINTER(ei->socket.wq, wq);     /*socket->wq 指向 socket_wq 实例*/

    ei->socket.state = SS_UNCONNECTED;     /*设置套接字状态（未连接）*/
    ei->socket.flags = 0;
    ei->socket.ops = NULL;
    ei->socket.sk = NULL;
    ei->socket.file = NULL;

    return &ei->vfs_inode;     /*返回 inode 结构体成员指针*/
```

}

分配 inode 实例后，由于其与 socket 实例共同组成 socket_alloc 实例，由容器机制，即可由 inode 实例指针获取 socket 实例指针。

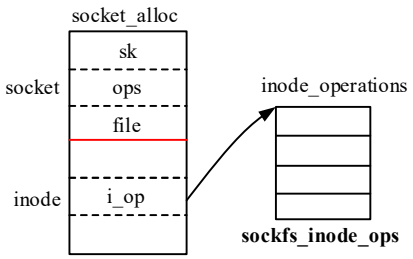
内核在初始化函数 sock_init()中为 socket_alloc 结构体创建了 slab 缓存，注册并挂载了 sockfs 伪文件系统，见上文。

■分配 socket 实例

sock_alloc()接口函数用于从 sockfs 伪文件系统中分配 socket_alloc 实例，返回内嵌的 socket 结构体成员指针，函数定义如下（/net/socket.c）：

```
static struct socket *sock_alloc(void)
{
    struct inode *inode;
    struct socket *sock;

    inode = new_inode_pseudo(sock_mnt->mnt_sb);   /*返回 vfs_inode 成员指针*/
                        /*调用超级块操作结构中的 sock_alloc_inode()函数分配 socket_alloc 实例*/
    ...
    sock = SOCKET_I(inode);     /*返回 socket 结构体成员指针*/

    kmemcheck_annotate_bitfield(sock, type);
    inode->i_ino = get_next_ino();          /*初始化 inode 结构体成员，inode 编号*/
    inode->i_mode = S_IFSOCK | S_IRWXUGO;          /*套接字文件*/
    inode->i_uid = current_fsuid();       /*文件系统 UID*/
    inode->i_gid = current_fsgid();       /*文件系统组 ID*/
    inode->i_op = &sockfs_inode_ops;     /*套接字节点文件操作结构，/net/socket.c*/

    this_cpu_add(sockets_in_use, 1);     /*增加当前 CPU 核套接字计数*/
    return sock;
}
```

sock_alloc()函数创建的 socket_alloc 实例如下图所示，内嵌 inode 结构体成员节点操作结构指针成员 i_op 指向 **sockfs_inode_ops** 实例，如下图所示。



2 socket()

用户进程通过 socket()系统调用创建套接字，系统调用中需要指明地址簇，套接字类型和协议类型，由于地址簇与协议簇是一一对应的关系，所以指定了地址簇即确定了协议簇。

socket()系统调用实现如下（/net/socket.c）：

```
SYSCALL_DEFINE3(socket, int, family, int, type, int, protocol)
/*family：地址簇标识，type：套接字类型，protocol：协议类型*/
{
    int retval;
    struct socket *sock;
    int flags;
    ...     /*有效性检查*/
    flags = type & ~SOCK_TYPE_MASK;
    if (flags & ~(SOCK_CLOEXEC | SOCK_NONBLOCK))
        return -EINVAL;
    type &= SOCK_TYPE_MASK;

    if (SOCK_NONBLOCK != O_NONBLOCK && (flags & SOCK_NONBLOCK))
        flags = (flags & ~SOCK_NONBLOCK) | O_NONBLOCK;

    retval = sock_create(family, type, protocol, &sock);    /*创建套接字*/
    ...
    retval = sock_map_fd(sock, flags & (O_CLOEXEC | O_NONBLOCK));  /*关联进程文件*/
    ...
out:
    return retval;
    ...
}
```

socket()系统调用主要分两步，一是创建并设置 socket 实例，二是建立 socket 与进程文件之间的关联，以便进程能通过文件描述符操作套接字。

■创建 socket 实例

socket()系统调用中通过 sock_create()函数创建 socket 实例，函数定义如下（/net/socket.c）：

```
int sock_create(int family, int type, int protocol, struct socket **res)
{
    return __sock_create(current->nsproxy->net_ns, family, type, protocol, res, 0);
}
```

__sock_create()函数定义如下：

```
int __sock_create(struct net *net, int family, int type, int protocol,struct socket **res, int kern)
/* *res：保存 socket 实例地址（指针），kern：是否是内核套接字，此处为 0*/
{
    int err;
    struct socket *sock;
    const struct net_proto_family *pf;
    ...     /*有效性检查*/

    if (family == PF_INET && type == SOCK_PACKET) {   /*过时的组合*/
        ...
```

```
    }

    err = security_socket_create(family, type, protocol, kern);     /*安全性检查*/
    ...
    sock = sock_alloc();     /*分配 socket 实例，见上文，/net/socket.c*/
    ...
    sock->type = type;     /*套接字类型，流套接字、数据报套接字等*/

#ifdef CONFIG_MODULES          /*支持模块*/
    if (rcu_access_pointer(net_families[family]) == NULL)
        request_module("net-pf-%d", family);   /*没找到对应 net_proto_family 实例，加载模块*/
#endif

    rcu_read_lock();
    pf = rcu_dereference(net_families[family]);     /*pf 指向协议簇注册的 net_proto_family 实例*/
    err = -EAFNOSUPPORT;
    ...
    rcu_read_unlock();

    err = pf->create(net, sock, protocol, kern);       /*调用 net_proto_family 实例中的 create()函数*/
    ...
    err = security_socket_post_create(sock, family, type, protocol, kern);       /*安全性检查*/
    ...
    *res = sock;     /*返回 socket 实例指针*/
    return 0;
    ...
}
```

__sock_create()函数中分配 socket 实例后，需要调用地址簇注册的 net_proto_family 实例中的 create()函数对 socket 实例进行设置，主要是关联协议簇定义的 **proto_ops** 实例，分配设置 **sock** 实例等。

■关联进程文件

socket()系统调用创建套接字后，需要将实例与进程文件建立关联，以便进程通过文件描述符操作套接字，**sock_map_fd()**函数用于完成这项工作，函数定义如下（/net/socket.c）：

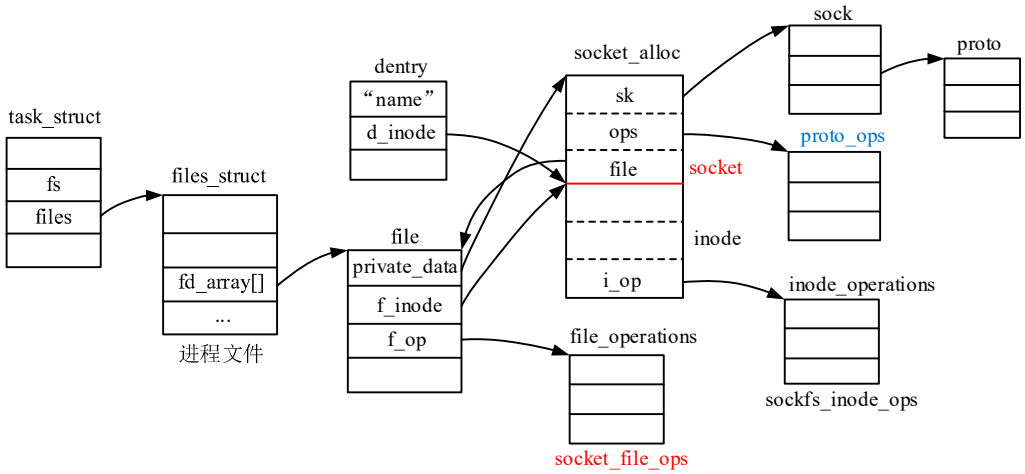
```
static int sock_map_fd(struct socket *sock, int flags)
{
    struct file *newfile;
    int fd = get_unused_fd_flags(flags);     /*获取进程最小未使用的文件描述符*/
    ...
    newfile = sock_alloc_file(sock, flags, NULL);     /*创建 file 实例等*/
    if (likely(!IS_ERR(newfile))) {
        fd_install(fd, newfile);     /*file 实例关联文件描述符*/
        return fd;     /*返回文件描述符*/
    }
    put_unused_fd(fd);
    return PTR_ERR(newfile);
```

```
}
```

sock_map_fd()函数调用 sock_alloc_file()函数创建 file 实例，并在 sockfs 伪文件系统中创建对应的目录项 dentry 实例，函数定义如下：

```
struct file *sock_alloc_file(struct socket *sock, int flags, const char *dname)
/*dname：此处为 NULL*/
{
    struct qstr name = { .name = "" };
    struct path path;
    struct file *file;

    if (dname) {    /*如果指定了目录项名称*/
        name.name = dname;
        name.len = strlen(name.name);
    } else if (sock->sk) {          /*参数没有指定目录项名称*/
        name.name = sock->sk->sk_prot_creator->name;   /*设为 proto->name[]*/
        name.len = strlen(name.name);
    }
    path.dentry = d_alloc_pseudo(sock_mnt->mnt_sb, &name);     /*分配 dentry 实例*/
    ...
    path.mnt = mntget(sock_mnt);     /*挂载结构*/

    d_instantiate(path.dentry, SOCK_INODE(sock));     /*dentry 关联 socket_alloc.vfs_inode 实例*/

    file = alloc_file(&path, FMODE_READ | FMODE_WRITE,&socket_file_ops);   /*分配 file 实例*/
    ...
    sock->file = file;
    file->f_flags = O_RDWR | (flags & O_NONBLOCK);     /*可读写，非阻塞*/
    file->private_data = sock;     /*指向 socket 实例*/
    return file;
}
```

socket()系统调用最终创建的套接字数据结构实例如下图所示，系统调用返回进程文件描述符：



注意，文件 file 实例中 f_op 成员指向的 file_operations 实例为 **socket_file_ops**，此实例用于实现进程通过普通文件操作接口操作套接字。

3 内核创建套接字

内核（内核线程）也可以创建（内核）套接字，但是以函数调用的方式而不是系统调用，例如：

●int **sock_create_kern**(struct net *net, int family, int type, int protocol, struct socket **res)：函数内直接调用__sock_create()函数创建内核套接字，内核套接字不需要关联 file 实例，内核直接通过指针访问 socket 实例。

●int **sock_create_lite**(int family, int type, int protocol, struct socket **res)：调用 sock_alloc()函数创建套接字 socket 实例，但是没有调用 net_proto_family 实例中的 create()函数，因此没有设置 proto_ops 结构体指针成员，也没有创建 sock 实例。

12.2.5 套接字操作

套接字将“万物皆文件”的 UNIX 哲学应用到了网络连接上。由前一小节我们知道，用户进程在创建套接字后，返回的是文件描述符，套接字关联到 file 实例，套接字也实现了文件操作结构 file_operations 实例，用户进程可以通过此实例以文件操作的形式操作套接字，如读写数据等。但是，套接字又是一种特殊的文件，内核将其不能纳入到普通文件操作结构中的操作用专用的系统调用来实现。

本小节介绍用户进程操作套接字的接口，包括专用系统调用和文件操作接口（其实都是系统调用）。另外，内核也会创建套接字，并对套接字进行操作，内核通过内核函数操作套接字，本小节最后将列出内核操作套接字的接口函数。

套接字专用系统调用、文件操作结构实例中的函数，以及内核接口函数都定义在/net/socket.c 文件内。

1 专用系统调用

虽然可以通过普通文件操作接口操作套接字，但是套接字是一种特殊的文件，有些操作不能纳入到普通文件操作接口中。因此，内核为套接字定义了专用的系统调用，例如：bind()、connect()等。另外，内核还实现了 socketcall()系统调用，它可以作为所有套接字专用系统调用的统一入口，它充当了一个分配器的角色。

■socketcall()

内核为每个套接字专用系统调用设置了标识编号，定义如下（/include/uapi/linux/net.h）：

```
#define   SYS_SOCKET      1          /* sys_socket(2)，创建套接字*/
#define   SYS_BIND        2          /* sys_bind(2)，为套接字绑定一个地址（服务器进程）*/
#define   SYS_CONNECT     3          /* sys_connect(2)，将套接字连接到另一套接字*/
#define   SYS_LISTEN      4          /* sys_listen(2)，打开被动连接，在套接字上监听，可接受连接*/
#define   SYS_ACCEPT      5          /* sys_accept(2)，接受一个进入的连接请求*/
#define   SYS_GETSOCKNAME      6     /* sys_getsockname(2)，返回本套接字地址*/
#define   SYS_GETPEERNAME      7     /* sys_getpeername(2)，返回对方套接字地址*/
#define   SYS_SOCKETPAIR       8     /* sys_socketpair(2)，创建一对套接字，用于双向通信*/
#define   SYS_SEND        9          /* sys_send(2)，以数据报发送数据*/
#define   SYS_RECV        10         /* sys_recv(2)，以数据报接收数据*/
#define   SYS_SENDTO      11         /* sys_sendto(2)，向明确指定的目标地址发送数据（UDP）*/
#define   SYS_RECVFROM    12   /*sys_recvfrom(2)，从一个数据报套接字接收数据，返回源地址*/
#define   SYS_SHUTDOWN      13         /* sys_shutdown(2)，关闭连接*/
#define   SYS_SETSOCKOPT    14         /* sys_setsockopt(2)，设置套接字参数*/
#define   SYS_GETSOCKOPT    15         /* sys_getsockopt(2)，获取套接字参数*/
#define   SYS_SENDMSG       16         /* sys_sendmsg(2)，以 BSD 风格发送消息*/
```

```
#define   SYS_RECVMSG        17        /* sys_recvmsg(2)，以 BSD 风格接收消息*/
#define   SYS_ACCEPT4        18        /* sys_accept4(2)，同 accept()*/
#define   SYS_RECVMMSG       19        /* sys_recvmmsg(2)，接收消息，内存映射方式*/
#define   SYS_SENDMMSG       20        /* sys_sendmmsg(2)，发送消息，内存映射方式*/
```

socketcall()系统调用参数中需要指明系统调用编号及参数集合，系统调用内部根据编号值调用相应系统调用的实现函数。socketcall()系统调用实现函数如下：

```
SYSCALL_DEFINE2(socketcall, int, call, unsigned long __user *, args)
/*call：专用系统调用编号，args：指向连续保存了专用系统调用参数的用户内存区*/
{
    unsigned long a[AUDITSC_ARGS];    /*保存系统调用参数*/
    unsigned long a0, a1;
    int err;
    unsigned int len;

    if (call < 1 || call > SYS_SENDMMSG)
        return -EINVAL;

    /*nargs[]全局数组记录了每个系统调用参数的总大小*/
    len = nargs[call]; /*系统调用所有参数大小，nargs[]数组项保存了专用系统调用所有参数的大小*/
    ...
    if (copy_from_user(a, args, len))    /*复制系统调用参数至内核空间*/
        return -EFAULT;

    err = audit_socketcall(nargs[call] / sizeof(unsigned long), a);
    ...
    a0 = a[0];    /*第一个参数，文件描述符*/
    a1 = a[1];    /*指向后面的参数*/

    switch (call) {      /*分配器，不同的专用系统调用调用不同的实现函数*/
    case SYS_SOCKET:
        err = sys_socket(a0, a1, a[2]);   /*创建套接字系统调用的实现函数*/
        break;
    case SYS_BIND:
        err = sys_bind(a0, (struct sockaddr __user *)a1, a[2]);
        break;
    case SYS_CONNECT:
        err = sys_connect(a0, (struct sockaddr __user *)a1, a[2]);
        break;
    case SYS_LISTEN:
        err = sys_listen(a0, a1);
        break;
    case SYS_ACCEPT:
        err = sys_accept4(a0, (struct sockaddr __user *)a1,(int __user *)a[2], 0);
        break;
    case SYS_GETSOCKNAME:
```

```
        err =sys_getsockname(a0, (struct sockaddr __user *)a1,(int __user *)a[2]);
        break;
case SYS_GETPEERNAME:
        err =sys_getpeername(a0, (struct sockaddr __user *)a1,(int __user *)a[2]);
        break;
case SYS_SOCKETPAIR:
        err = sys_socketpair(a0, a1, a[2], (int __user *)a[3]);
        break;
case SYS_SEND:
        err = sys_send(a0, (void __user *)a1, a[2], a[3]);
        break;
case SYS_SENDTO:
        err = sys_sendto(a0, (void __user *)a1, a[2], a[3],(struct sockaddr __user *)a[4], a[5]);
        break;
case SYS_RECV:
        err = sys_recv(a0, (void __user *)a1, a[2], a[3]);
        break;
case SYS_RECVFROM:
        err = sys_recvfrom(a0, (void __user *)a1, a[2], a[3],(struct sockaddr __user *)a[4],
                                                                (int __user *)a[5]);
        break;
case SYS_SHUTDOWN:
        err = sys_shutdown(a0, a1);
        break;
case SYS_SETSOCKOPT:
        err = sys_setsockopt(a0, a1, a[2], (char __user *)a[3], a[4]);
        break;
case SYS_GETSOCKOPT:
        err =sys_getsockopt(a0, a1, a[2], (char __user *)a[3],(int __user *)a[4]);
        break;
case SYS_SENDMSG:
        err = sys_sendmsg(a0, (struct user_msghdr __user *)a1, a[2]);
        break;
case SYS_SENDMMSG:
        err = sys_sendmmsg(a0, (struct mmsghdr __user *)a1, a[2], a[3]);
        break;
case SYS_RECVMSG:
        err = sys_recvmsg(a0, (struct user_msghdr __user *)a1, a[2]);
        break;
case SYS_RECVMMSG:
        err = sys_recvmmsg(a0, (struct mmsghdr __user *)a1, a[2], a[3],(struct timespec __user *)a[4]);
        break;
case SYS_ACCEPT4:
        err = sys_accept4(a0, (struct sockaddr __user *)a1,(int __user *)a[2], a[3]);
        break;
default:
```

```
            err = -EINVAL;
            break;
        }
        return err;
    }
```

socketcall()系统调用充当一个分配器，根据 call 参数调用专用系统调用的实现函数，参数 args 指向用户内存区，此内存区保存了 call 系统调用的所有参数，所有参数按顺序存放。

■专用系统调用简介

下面简要列出几个常用套接字专用系统调用：

●**socket**(int family, int type, int, protocol)：创建套接字。

●**bind**(int fd, struct sockaddr __user * umyaddr, int addrlen)：将 umyaddr 指向的地址绑定到套接字，主要用于服务器进程。

●**listen**(int fd, int backlog)：在套接字上监听，允许一个流 socket 接受来自其它 socket 的接入连接请求，主要用于服务器进程。

●**accept**(int fd, struct sockaddr __user * upeer_sockaddr,int __user * upeer_addrlen)：在一个监听流 socket 上接受来自一个对等应用程序的连接，并可选地返回对等 socket 地址，主要用于服务器进程。

●**connect**(int fd, struct sockaddr __user * uservaddr,int addrlen)：建立与另一个 socket 之间的连接，主要用于客户进程。

以下三个系统调用用于接收数据：

●**recv**(int fd, void __user * ubuf, size_t size,unsigned int flags)　　/*不返回对方套接字地址*/

●**recvfrom**(int fd, void __user * ubuf, size_t size,unsigned int flags, struct sockaddr __user * addr,int __user * addr_len)　　/*addr 中返回对方套接字地址*/

●**recvmsg**(int fd, struct user_msghdr __user * msg,unsigned int flags)

以下三个系统调用用于发送数据：

●**send**(int fd, void __user * buff, size_t len,unsigned int flags)　　/*未指定目的套接字地址*/

●**sendto**(int fd, void __user * buff, size_t len,unsigned int flags, struct sockaddr __user * addr,int addr_len)
/*addr 中指定目的套接字地址*/

●**sendmsg**(int fd, struct user_msghdr __user * msg, unsigned int flags)

●**setsockopt**(int fd, int level, int optname,char __user * optval, int optlen)：设置套接字、传输协议参数，有点类似于普通文件的 ioctl()系统调用，实现对套接字的控制或设置某项参数。

●**getsockopt**(int fd, int level, int optname,char __user * optval, int __user * optlen)：获取套接字、传输协议参数。

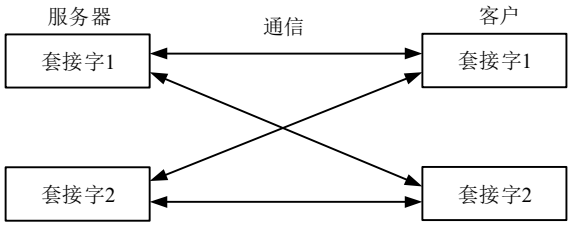
以上系统调用主要是通过调用 proto_ops 实例中的同名函数完成操作。

套接字有两个基本的类型：流套接字和数据报套接字。

流套接字（SOCK_STREAM）提供了一个可靠的双向字节流通信信道。流套接字保证发送者发送的数据会完整无缺陷、正确按序地到达接收方（接收套接字）。流套接字在通信前需要在一对相互连接的套接字之间建立连接，一个流套接字只能与一个已连接对等套接字进行通信，如下图所示。

服务器进程首先创建的是一个欢迎套接字，用于接收客户进程的连接请求，欢迎套接字接收了连接请求后，为每个连接请求创建一个套接字，用于与提交请求的客户套接字通信，也就是说通信双方套接字是一对一的关系。

数据报套接字（SOCK_DGRAM）允许数据以被称为数据报的消息形式进行交换，数据的传输是不可靠的，消息可能无序的、重复的到达接收方，甚至可能丢失而不能到达接收方。数据报套接字是无连接的套接字，通信前双方不需要建立连接。通信双方都可以接收任意套接字发送的数据，也可以向任意套接字发送数据，如下图所示。

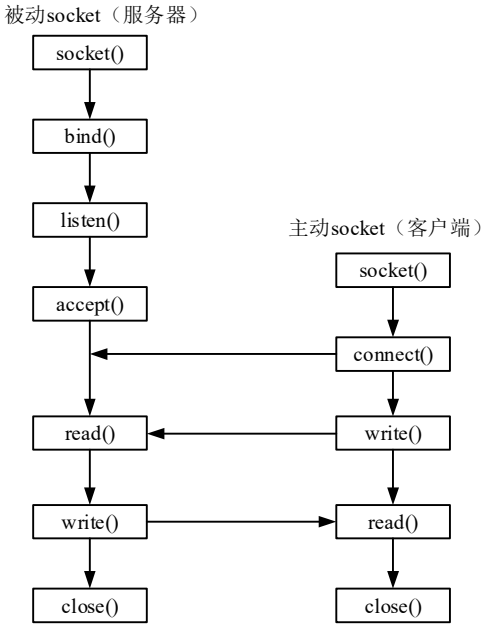


在上图中，客户套接字 1 可与服务器套接字 1 通信，也可与服务器套接字 2 通信。数据报套接字在发送数据时，可通过参数指定目的套接字。在接收数据时，可同时返回对方套接字地址，以便识别数据来自哪个套接字。另外，数据报套接字也可以执行连接操作，执行此操作后只接收连接套接字发送的数据，在不指定目的套接字时，默认将数据发给连接的套接字，也可以在发送操作中指定目的套接字而将数据发给其它套接字。

下面简要介绍流套接字和数据报套接字中执行的专用系统调用。

●流套接字操作

下图示意了在流套接字上常用到的系统调用：



流socket上用到的系统调用

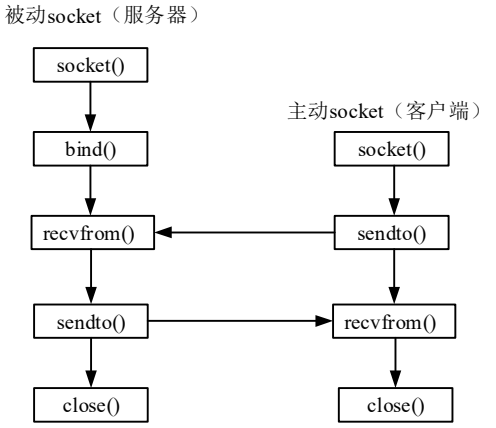
socket()用于创建套接字，被动连接套接字调用 bind()给自身绑定一个众所周知的地址，以便主动连接套接字寻址到它，然后调用 listen()通知内核它可以接受接入连接请求了。accept()用于接受一个连接请求，为连接请求创建一个套接字，用于与客户提交连接请求的套接字通信，返回文件描述符，以便服务器进程通过文件描述符访问新创建的套接字。accept()在没有连接请求时，将会阻塞直到有客户提交连接请求。

主动套接字调用 connect()向被动连接套接字发送连接请求，建立与被动套接字的连接（需要指定被动套接字地址）。

连接成功后，通信双方可通过 read()、recv()/write()、send()等系统调用（或文件操作系统调用）接收和发送数据，close()用于关闭套接字。

●数据报套接字操作

下图示意了在数据报套接字上常用的系统调用：



数据报socket上用到的系统调用

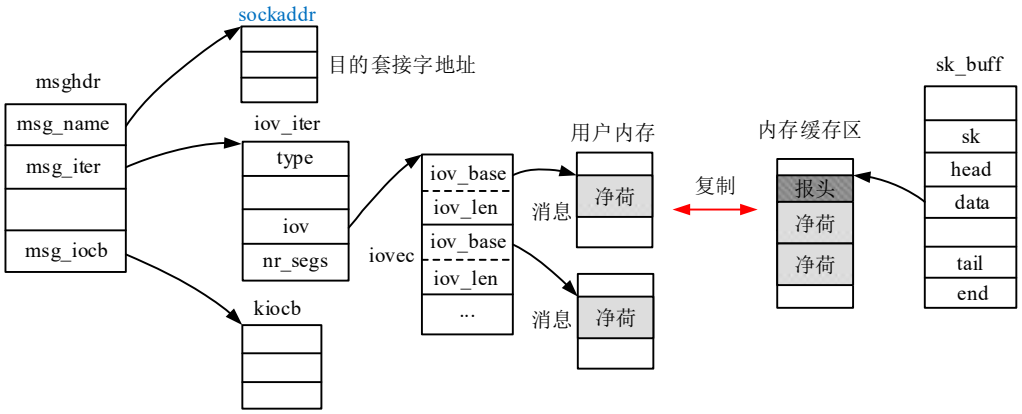
套接字在数据传输前不需要建立连接，被动连接套接字通常调用 bind()给自身绑定一个众所周知的地址，以便主动套接字寻址到它。通信双方通过 recvfrom()、sendto()等系统调用接收和发送消息，发送消息时需指定目的套接字地址，接收消息时可返回对方套接字地址，close()用于关闭套接字。

另外，数据报套接字也可以执 connect()系统调用，实现默认只与指定的套接字通信。执行连接操作后，应用进程也可以通过 read()、recv()/write()、send()等系统调用（不需要指定对方地址）接收和发送数据。

■发送数据

下面以发送数据为例，介绍套接字专用系统调用的实现。

套接字操作的读写函数中需要构建 msghdr 结构体实例，并将其作为调用 proto_ops 实例中接收发送消息函数的参数。msghdr 结构体如下图所示：



msghdr 结构体定义如下（/include/linux/socket.h）：

```
struct msghdr {
    void        *msg_name;     /*指向目的套接字地址结构*/
    int         msg_namelen;   /*地址结构长度*/
    struct iov_iter   msg_iter;    /*用户数据，包含 iovec 数组，可包含多个数据块*/
    void        *msg_control;      /* ancillary data 用于消息控制  */
    __kernel_size_t     msg_controllen;     /* ancillary data buffer length */
    unsigned int    msg_flags;     /*发送/接收数据标记*/
    struct kiocb    *msg_iocb;     /*指向 kiocb 结构体*/
};
```

msghdr 结构体主要成员简介如下：

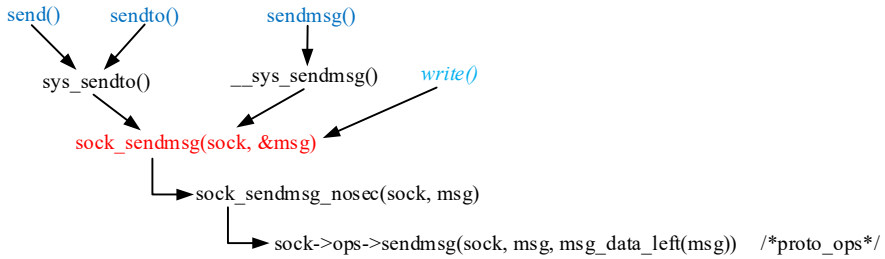
●**msg_name**：指向表示目的套接字地址的数据结构，如 sockaddr。

●**msg_flags**：发送/接收数据标记，标记值定义在/include/linux/socket.h 头文件，例如：

```
#define   MSG_OOB          1
#define   MSG_PEEK         2              /*用户进程读取套接字缓存队列中数据后，不释放数据包*/
#define   MSG_DONTROUTE    4              /*不执行路由选择查找*/
#define   MSG_TRYHARD      4              /* Synonym for MSG_DONTROUTE for DECnet */
#define   MSG_CTRUNC       8
#define   MSG_PROBE        0x10           /*只探测部分消息（如 MTU），不发送数据*/
#define   MSG_TRUNC        0x20
#define   MSG_DONTWAIT     0x40           /*非阻塞操作*/
#define   MSG_EOR          0x80           /* End of record */
#define   MSG_WAITALL      0x100          /* Wait for a full request */
#define   MSG_FIN          0x200          /*结束标记，用于关闭连接*/
#define   MSG_SYN          0x400          /*同步标记，用于连接建立*/
#define   MSG_CONFIRM      0x800          /* Confirm path validity */
#define   MSG_RST          0x1000         /*连接复位标记*/
#define   MSG_ERRQUEUE     0x2000         /* Fetch message from error queue */
#define   MSG_NOSIGNAL     0x4000         /* Do not generate SIGPIPE */
#define   MSG_MORE         0x8000         /*发送者还有更多数据，合并后再发送*/
#define   MSG_WAITFORONE   0x10000  /* recvmmsg(): block until 1+ packets avail */
#define   MSG_SENDPAGE_NOTLAST   0x20000 /* sendpage() internal : not the last page */
#define   MSG_EOF          MSG_FIN        /*结束（关闭）连接标记*/

#define   MSG_FASTOPEN         0x20000000  /* Send data in TCP SYN */
#define   MSG_CMSG_CLOEXEC     0x40000000     /**/
...
```

以上标记位许多是用于网络套接字的，后面介绍网络套接字、网络层协议实现时，将会讲解标记位的含义。

套接字发送消息的专用系统调用有 send()、sendto()、sendmsg()等，函数调用关系简列如下图所示：



下面以 **sendto()**系统调用为例（发送数据报到指定地址套接字），简要介绍其实现代码（/net/socket.c）：

```
SYSCALL_DEFINE6(sendto, int, fd, void __user *, buff, size_t, len,
                unsigned int, flags, struct sockaddr __user *, addr,int, addr_len)
/*
*fd：套接字文件描述符，buff：指向发送数据缓存区，len：发送数据长度，
*addr：指向通用地址表示 sockaddr 实例（目的套接字地址），addr_len：地址长度，
*flags：发送数据标记，/include/linux/socket.h。
*/
{
    struct socket *sock;
    struct sockaddr_storage address;    /*暂存目的套接字地址*/
    int err;
    struct msghdr msg;      /*msghdr 实例，见上文*/
    struct iovec iov;       /*关联数据缓存区*/
    int fput_needed;

    err = import_single_range(WRITE, buff, len, &iov, &msg.msg_iter);
                            /*由参数设置 msghdr 实例，/lib/iov_iter.c*/
    ...
    sock = sockfd_lookup_light(fd, &err, &fput_needed);  /*由文件描述符获取发送消息 socket 实例*/
    ...
    msg.msg_name = NULL;    /*初始化 msghdr 实例成员*/
    msg.msg_control = NULL;
    msg.msg_controllen = 0;
    msg.msg_namelen = 0;
    if (addr) {
        err = move_addr_to_kernel(addr, addr_len, &address);  /*复制目的套接字地址到内核空间*/
        if (err < 0)
            goto out_put;
        msg.msg_name = (struct sockaddr *)&address;  /*指向地址结构*/
        msg.msg_namelen = addr_len;     /*地址结构长度*/
    }
    if (sock->file->f_flags & O_NONBLOCK)    /*如果文件设置了非阻塞操作标记（默认）*/
        flags |= MSG_DONTWAIT;
    msg.msg_flags = flags;      /*设置标记*/
    err = sock_sendmsg(sock, &msg);    /*发送数据*/

out_put:
    fput_light(sock->file, fput_needed);
```

```
out:
    return err;
}
```

sendto()系统调用内通过传递的参数设置 msghdr 实例 msg，包括数据缓存区，目的套接字地址等信息，然后调用 sock_sendmsg(sock, &msg)函数完成数据发送，此函数是所有套接字发送数据系统调用的执行函数。

sock_sendmsg(sock, &msg)函数定义如下（/net/socket.c）：

```
int sock_sendmsg(struct socket *sock, struct msghdr *msg)
{
    int err = security_socket_sendmsg(sock, msg,msg_data_left(msg));   /*安全性检查*/

    return err ?: sock_sendmsg_nosec(sock, msg);
}
```

sock_sendmsg_nosec()函数定义如下（/net/socket.c）：

```
static inline int sock_sendmsg_nosec(struct socket *sock, struct msghdr *msg)
{
    int ret = sock->ops->sendmsg(sock, msg, msg_data_left(msg));     /*proto_ops->sendmsg()*/
    BUG_ON(ret == -EIOCBQUEUED);
    return ret;
}
```

sock_sendmsg_nosec()函数内调用 socket 实例关联的 proto_ops 实例中的 sendmsg()函数完成特定于传输协议的数据发送操作，参数 msg_data_left(msg)表示发送数据的长度。

套接字其它专用系统调用与发送数据系统调用类似，最后基本都是调用 proto_ops 实例中同名的函数完成操作，请读者自行阅读源代码（/net/socket.c）。

■设置参数

setsockopt()/getsockopt()系统调用于设置/获取套接字、传输协议参数。参数分不同的级别（level，或者说分类），每个级别中又包含多个参数（由整型数表示）。

内核在/include/uapi/asm-generic/socket.h 头文件中定义了 SOL_SOCKET 参数级别及其参数名称，例如：

```
#define   SOL_SOCKET      1      /*参数级别（通用参数）*/
/*以下是参数名称*/
#define   SO_DEBUG        1
#define   SO_REUSEADDR    2
#define   SO_TYPE         3
#define   SO_ERROR        4
#define   SO_DONTROUTE    5     /*不执行路由选择*/
#define   SO_BROADCAST    6
#define   SO_SNDBUF       7     /*发送缓存区*/
#define   SO_RCVBUF       8      /*接收缓存区*/
#define   SO_SNDBUFFORCE    32     /*强制设置发送缓存区大小（字节数）*/
#define   SO_RCVBUFFORCE    33     /*强制设置接收缓存区大小（字节数）*/
#define   SO_KEEPALIVE      9
...
```

内核在/include/linux/socket.h 头文件还定义了其它的套接字参数级别：

```
#define   SOL_IP          0       /*IPv4 参数，参数名称定义在/include/uapi/linux/in.h 头文件*/
#define   SOL_TCP         6       /*TCP 参数*/
#define   SOL_UDP         17      /*UDP 参数*/
#define   SOL_IPV6        41      /*IPv6 参数*/
#define   SOL_ICMPV6      58
#define   SOL_SCTP        132
#define   SOL_UDPLITE     136         /* UDP-Lite (RFC 3828) */
...
```

以上参数级别下的参数名称通常定义在/include/uapi/linux/xxx.h 头文件，请读者自行查阅。

下面以设置参数的 setsockopt()系统调用为例，介绍其实现，源代码位于/net/socket.c 文件内：

```
SYSCALL_DEFINE5(setsockopt, int, fd, int, level, int, optname,char __user *, optval, int, optlen)
/*
*fd：套接字文件描述符，level：参数级别，optname：参数名称（整型数表示），
*optval：用户空间内存地址，保存参数值，通常是某一结构体指针，optlen：参数值长度。
*/
{
    int err, fput_needed;
    struct socket *sock;

    if (optlen < 0)
        return -EINVAL;

    sock = sockfd_lookup_light(fd, &err, &fput_needed);   /*文件描述符关联的 socket 实例*/
    if (sock != NULL) {
        err = security_socket_setsockopt(sock, level, optname);
        if (err)
            goto out_put;

        if (level == SOL_SOCKET)      /*最高级别（通用）的参数*/
            err =sock_setsockopt(sock, level, optname, optval,optlen);   /*处理通用参数，/net/socket.c*/
        else
            err =sock->ops->setsockopt(sock, level, optname, optval,optlen);   /*处理私有参数*/
out_put:
        fput_light(sock->file, fput_needed);
    }
    return err;
}
```

setsockopt()系统调用中对于 SOL_SOCKET 级别的参数（通用参数），由 **sock_setsockopt()**函数处理，对于其它级别的（私有）参数由套接字关联 proto_ops 实例中的 setsockopt()函数处理。

获取参数值的系统调用为 getsockopt()，它与 setsockopt()系统调用类似，只不过数据传输的方向不同，源代码请读者自行阅读。

2 文件操作接口

在用户进程创建套接字的socket()系统调用中，其关联的file实例中文件操作结构实例为socket_file_ops，此实例是进程通过文件操作系统调用操作套接字的接口，实例定义如下（/net/socket.c）：

```
static const struct file_operations socket_file_ops = {
    .owner =  THIS_MODULE,
    .llseek =  no_llseek,
    .read_iter =    sock_read_iter,      /*读操作函数*/
    .write_iter =   sock_write_iter,     /*写操作函数*/
    .poll =         sock_poll,
    .unlocked_ioctl = sock_ioctl,       /*IO 控制*/
#ifdef CONFIG_COMPAT
    .compat_ioctl = compat_sock_ioctl,
#endif
    .mmap =  sock_mmap,
    .release = sock_close,       /*关闭套接字*/
    .fasync = sock_fasync,
    .sendpage =sock_sendpage,
    .splice_write = generic_splice_sendpage,
    .splice_read =  sock_splice_read,
};
```

用户进程可以通过 read()、write()、ioctl()等系统调用操作套接字，这些系统调用将调用 socket_file_ops 实例中的相应函数完成操作。

下面将简要介绍其中的套接字读、写、关闭、IO 控制操作函数的实现。

■读操作

在 read()和 write()系统调用中将构建 kiocb 和 iov_iter 结构体实例，然后调用 socket_file_ops 实例中的 read_iter()和 write_iter()函数完成操作。在这两个函数内将定义 msghdr 结构体实例（见上文），并将系统调用内构建的 kiocb 和 iov_iter 实例指针赋予 msghdr 实例。

socket_file_ops 实例中读操作函数 sock_read_iter()定义如下：

```
static ssize_t sock_read_iter(struct kiocb *iocb, struct iov_iter *to)
{
    struct file *file = iocb->ki_filp;
    struct socket *sock = file->private_data;     /*指向 socket 实例*/
    struct msghdr msg = {.msg_iter = *to,        /*定义 msghdr 实例并初始化*/
                         .msg_iocb = iocb};
    ssize_t res;

    if (file->f_flags & O_NONBLOCK)   /*创建套接字时，设置为可读写、非阻塞属性，见上文*/
        msg.msg_flags = MSG_DONTWAIT;     /*设置标记成员*/

    if (iocb->ki_pos != 0)
        return -ESPIPE;

    if (!iov_iter_count(to))   /* Match SYS5 behaviour */
```

```
        return 0;

    res = sock_recvmsg(sock, &msg, iov_iter_count(to), msg.msg_flags);
                                                    /*调用 proto_ops 实例中接收消息函数*/
    *to = msg.msg_iter;
    return res;     /*返回读取字节数*/
}
```

sock_recvmsg()函数最终调用 socket->**ops->recvmsg**(sock, msg, size, flags)函数（proto_ops->sendmsg()）完成接收消息的操作，通常是从套接字接收数据包缓存队列中的 sk_buff 实例关联的内存缓存区复制数据至用户内存。

传输协议接收到数据包（消息）后，通常将其添加到目的套接字的接收数据包缓存队列，等待用户进程读取。

■写操作

socket_file_ops 实例中写操作函数 sock_write_iter()定义如下：

```
static ssize_t sock_write_iter(struct kiocb *iocb, struct iov_iter *from)
{
    struct file *file = iocb->ki_filp;
    struct socket *sock = file->private_data;
    struct msghdr msg = {.msg_iter = *from,     /*定义 msghdr 实例，并初始化*/
                         .msg_iocb = iocb};
    ssize_t res;

    if (iocb->ki_pos != 0)
        return -ESPIPE;

    if (file->f_flags & O_NONBLOCK)
        msg.msg_flags = MSG_DONTWAIT;

    if (sock->type == SOCK_SEQPACKET)
        msg.msg_flags |= MSG_EOR;

    res = sock_sendmsg(sock, &msg);   /*调用 proto_ops 实例中发送消息函数*/

    *from = msg.msg_iter;
    return res;
}
```

写操作与读操作类似，sock_sendmsg(sock, &msg)函数最终调用 socket->ops->**sendmsg**(sock, msg, msg_data_left(msg))函数（proto_ops->sendmsg()）完成消息发送操作。

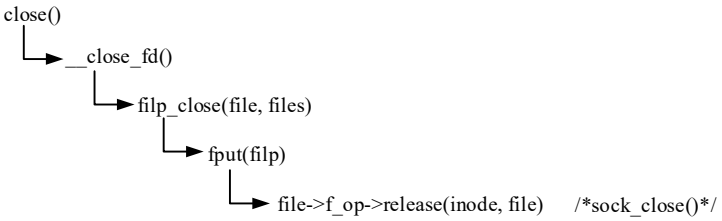
proto_ops->sendmsg()函数的主要工作是创建 sk_buff 实例，将用户空间数据复制到其内存缓存区，如果数据是发送到本主机中的其它套接字，则只需要将 sk_buff 实例绑定到目的套接字的接收消息缓存队列即可。如果消息需要外发，则将消息插入到套接字发送消息缓存队列，由传输协议将消息发送出去，或直接将数据通过传输协议发送出去。

在 read()、write()系统调用中，没有返回、指定对方套接字的目的地址。如果是流套接字，这没有问题，因为在通信前双方已经建立了连接，套接字之间是一对一的通信。如果是数据报套接字，通常需要先执行

connect()系统调用，事先指定通信对方套接字的地址。

■关闭套接字

用户进程通过 close()系统调用关闭套接字。close()系统调用在/fs/open.c 文件内实现，函数调用关系简列如下图所示：



close()系统调用中调用套接字文件操作结构 socket_file_ops 实例中的 release()函数释放套接字，此函数为 **sock_close()**，函数定义如下（/net/socket.c）：

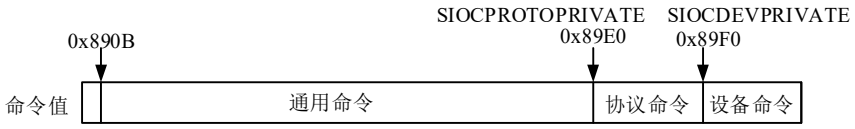
```
static int sock_close(struct inode *inode, struct file *filp)
{
    sock_release(SOCKET_I(inode));      /*/net/socket.c*/
    return 0;
}

sock_release()函数定义如下：
void sock_release(struct socket *sock)
{
    if (sock->ops) {
        struct module *owner = sock->ops->owner;

        sock->ops->release(sock);       /*调用 proto_ops 实例中的 release()函数*/
        sock->ops = NULL;
        module_put(owner);
    }
    ...
    this_cpu_sub(sockets_in_use, 1);
    if (!sock->file) {
        iput(SOCK_INODE(sock));
        return;
    }
    sock->file = NULL;
}
```

sock_release()函数最后调用 proto_ops 实例中的 release()函数释放套接字。

■套接字控制

用户进程可通过 ioctl()系统调用对套接字进行控制操作，如对网络参数进行设置，对网络设备进行控制等。ioctl()系统调用需传递命令值（cmd）和用于存储信息的数据结构实例指针。

套接字命令值布局如下图所示：



其中[0x890B,0x89E0)命令值范围可视为通用命令，适用于所有协议簇套接字，[0x89E0,0x89EF]为特定于协议的命令（16 个），[0x89F0,0x89FF]为设备私有的命令（16 个）。

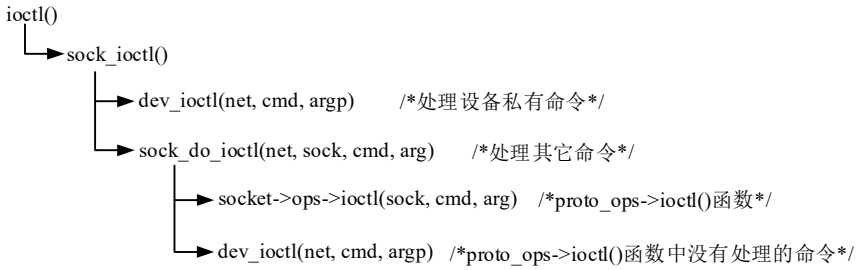
套接字命令 cmd 值定义在/include/uapi/linux/sockios.h 头文件内，如下所示：

```
/* Linux 特有命令*/
#define   SIOCINQ          FIONREAD
#define   SIOCOUTQ         TIOCOUTQ          /* output queue size (not sent + not acked) */

/*路由选择表命令*/
#define   SIOCADDRT        0x890B      /*添加路由选择表项*/
#define   SIOCDELRT        0x890C      /*删除路由选择表项*/
#define   SIOCRTMSG        0x890D      /* call to routing system */

/*网络设备配置命令*/
#define   SIOCGIFNAME      0x8910      /* get iface name*/
#define   SIOCSIFLINK      0x8911      /* set iface channel */
#define   SIOCGIFCONF      0x8912      /* get iface list*/
#define   SIOCGIFFLAGS     0x8913      /* get flags*/
#define   SIOCSIFFLAGS     0x8914      /* set flags*/
#define   SIOCGIFADDR      0x8915      /* get PA address，获取物理地址*/
#define   SIOCSIFADDR      0x8916      /* set PA address，设置物理地址*/
#define   SIOCGIFDSTADDR   0x8917        /* get remote PA address      */
#define   SIOCSIFDSTADDR   0x8918        /* set remote PA address*/
#define   SIOCGIFBRDADDR   0x8919        /* get broadcast PA address，获取广播物理地址*/
#define   SIOCSIFBRDADDR   0x891a        /* set broadcast PA address，设置广播物理地址*/
#define   SIOCGIFNETMASK   0x891b        /* get network PA mask */
#define   SIOCSIFNETMASK   0x891c        /* set network PA mask*/
#define   SIOCGIFMETRIC    0x891d        /* get metric*/
#define   SIOCSIFMETRIC    0x891e        /* set metric*/
#define   SIOCGIFMEM       0x891f      /* get memory address (BSD)*/
#define   SIOCSIFMEM       0x8920      /* set memory address (BSD) */
#define   SIOCGIFMTU       0x8921      /* get MTU size，获取 MTU 值*/
#define   SIOCSIFMTU       0x8922      /* set MTU size，设置 MTU 值*/
#define   SIOCSIFNAME      0x8923      /* set interface name */
#define   SIOCSIFHWADDR    0x8924        /* set hardware address  */
#define   SIOCGIFENCAP     0x8925        /* get/set encapsulations*/
#define   SIOCSIFENCAP     0x8926
#define   SIOCGIFHWADDR    0x8927        /* Get hardware address       */
#define   SIOCGIFSLAVE     0x8929        /* Driver slaving support     */
#define   SIOCSIFSLAVE     0x8930
#define   SIOCADDMULTI     0x8931        /* Multicast address lists*/
#define   SIOCDELMULTI     0x8932
#define   SIOCGIFINDEX     0x8933        /* name -> if_index mapping */
```

```
#define   SIOGIFINDEX          SIOCGIFINDEX   /* misprint compatibility :-)   */
#define   SIOCSIFPFLAGS        0x8934         /* set/get extended flags set   */
#define   SIOCGIFPFLAGS        0x8935
#define   SIOCDIFADDR          0x8936         /* delete PA address            */
#define   SIOCSIFHWBROADCAST   0x8937   /* set hardware broadcast addr      */
#define   SIOCGIFCOUNT         0x8938         /* get number of devices */

#define   SIOCGIFBR      0x8940          /* Bridging support        */
#define   SIOCSIFBR      0x8941          /* Set bridging options    */

#define   SIOCGIFTXQLEN      0x8942        /* Get the tx queue length      */
#define   SIOCSIFTXQLEN      0x8943        /* Set the tx queue length      */
#define   SIOCETHTOOL        0x8946        /* Ethtool interface */

#define   SIOCGMIIPHY        0x8947        /* Get address of MII PHY in use. */
#define   SIOCGMIIREG        0x8948        /* Read MII PHY register.     */
#define   SIOCSMIIREG        0x8949        /* Write MII PHY register.    */
#define   SIOCWANDEV         0x894A        /* get/set netdev parameters   */
#define   SIOCOUTQNSD        0x894B        /* output queue size (not sent only) */

#define   SIOCDARP     0x8953        /* delete ARP table entry，删除邻居表项*/
#define   SIOCGARP     0x8954        /* get ARP table entry，获取邻居表项*/
#define   SIOCSARP     0x8955        /* set ARP table entry，设置邻居表项*/

/* RARP cache control calls. */
#define   SIOCDRARP      0x8960        /* delete RARP table entry    */
#define   SIOCGRARP      0x8961        /* get RARP table entry       */
#define   SIOCSRARP      0x8962        /* set RARP table entry       */

/* Driver configuration calls */
#define   SIOCGIFMAP     0x8970        /* Get device parameters*/
#define   SIOCSIFMAP     0x8971        /* Set device parameters */

/* DLCI configuration calls */
#define   SIOCADDDLCI    0x8980        /* Create new DLCI device   */
#define   SIOCDELDLCI    0x8981        /* Delete DLCI device       */

#define   SIOCGIFVLAN    0x8982        /* 802.1Q VLAN support          */
#define   SIOCSIFVLAN    0x8983        /* Set 802.1Q VLAN options      */

/* bonding calls */
#define   SIOCBONDENSLAVE      0x8990          /* enslave a device to the bond */
#define   SIOCBONDRELEASE      0x8991          /* release a slave from the bond*/
#define   SIOCBONDSETHWADDR          0x8992/* set the hw addr of the bond   */
#define   SIOCBONDSLAVEINFOQUERY   0x8993    /* rtn info about slave state     */
#define   SIOCBONDINFOQUERY            0x8994    /* rtn info about bond state       */
```

```
#define    SIOCBONDCHANGEACTIVE       0x8995     /* update to a new active slave */

/* bridge calls */
#define    SIOCBRADDBR         0x89a0         /* create new bridge device         */
#define    SIOCBRDELBR         0x89a1         /* remove bridge device             */
#define    SIOCBRADDIF         0x89a2         /* add interface to bridge          */
#define    SIOCBRDELIF         0x89a3         /* remove interface from bridge */

/* hardware time stamping: parameters in linux/net_tstamp.h */
#define    SIOCSHWTSTAMP       0x89b0         /* set and get config*/
#define    SIOCGHWTSTAMP       0x89b1         /* get config*/


#define    SIOCPROTOPRIVATE    0x89E0     /*0x89E0 之后（含）的 16 个命令由协议定义*/
#define    SIOCDEVPRIVATE      0x89F0
                /*0x89F0 之后（含）的 16 个命令由网络设备定义，由 dev_ioctl()函数处理*/
```

ioctl()系统调用将调用 socket_file_ops 实例中 unlocked_ioctl()函数 **sock_ioctl()**完成操作，函数调用关系如下图所示：



对于设备私有命令将调用 dev_ioctl()函数处理，其它命令基本调用 sock_do_ioctl()函数处理，此函数内将调用 proto_ops->ioctl()函数处理命令，对 proto_ops->ioctl()函数中没有实现处理函数的命令，继续调用函数 dev_ioctl()进行处理。

sock_ioctl()函数定义如下，参数 cmd 表示命令值，arg 通常是一个地址值（指针），表示一个数据结构实例的基地址，用于传递命令参数（/net/socket.c）。

```
static long sock_ioctl(struct file *file, unsigned cmd, unsigned long arg)
{
    struct socket *sock;
    struct sock *sk;
    void __user *argp = (void __user *)arg;
    int pid, err;
    struct net *net;

    sock = file->private_data;
    sk = sock->sk;
    net = sock_net(sk);
    if (cmd >= SIOCDEVPRIVATE && cmd <= (SIOCDEVPRIVATE + 15)) {
        err = dev_ioctl(net, cmd, argp);   /*处理设备私有命令，/net/core/dev_ioctl.c*/
    } else
#ifdef CONFIG_WEXT_CORE
```

```
        if (cmd >= SIOCIWFIRST && cmd <= SIOCIWLAST) {
            err = dev_ioctl(net, cmd, argp);
        } else
#endif
            switch (cmd) {
            ...     /*部分命令处理函数*/
            default:    /*其它命令处理函数*/
                err = sock_do_ioctl(net, sock, cmd, arg);
                        /*调用 proto_ops->ioctl()或 dev_ioctl()函数，/net/socket.c*/
                break;
            }
        return err;
}
```

3 内核操作函数

用户进程通过系统调用操作套接字，内核则通过函数调用操作套接字，相关函数声明如下：

```
/*/include/linux/net.h*/
int kernel_sendmsg(struct socket *sock, struct msghdr *msg, struct kvec *vec,size_t num, size_t len);
int kernel_recvmsg(struct socket *sock, struct msghdr *msg, struct kvec *vec,
                                                    size_t num, size_t len, int flags);

int kernel_bind(struct socket *sock, struct sockaddr *addr, int addrlen);
int kernel_listen(struct socket *sock, int backlog);
int kernel_accept(struct socket *sock, struct socket **newsock, int flags);
int kernel_connect(struct socket *sock, struct sockaddr *addr, int addrlen,int flags);
int kernel_getsockname(struct socket *sock, struct sockaddr *addr,int *addrlen);
int kernel_getpeername(struct socket *sock, struct sockaddr *addr,int *addrlen);
int kernel_getsockopt(struct socket *sock, int level, int optname, char *optval,int *optlen);
int kernel_setsockopt(struct socket *sock, int level, int optname, char *optval,unsigned int optlen);
int kernel_sendpage(struct socket *sock, struct page *page, int offset,size_t size, int flags);
int kernel_sock_ioctl(struct socket *sock, int cmd, unsigned long arg);
int kernel_sock_shutdown(struct socket *sock, enum sock_shutdown_cmd how);
```

以上函数在/net/socket.c 文件内实现，函数实现与对应的系统调用实现类似，只不过内核不是用文件描述符来寻址套接字，而是直接采用 socket 实例指针寻址套接字。

套接字作为一种进程间通信机制，可应用于不同的通信域，例如：内核与本机用户进程间的通信，本机用户进程间的通信，本机进程与其它主机进程之间的网络通信等。内核通过地址簇（协议簇）来标识通信域，每个通信域具有自身的数据传输协议（协议簇），协议簇相关代码需要实现网络操作接口实例。

用户进程在创建套接字时需指明使用的地址簇（协议簇）、套接字类型和协议类型，套接字将关联到协议簇实现的操作接口实例。套接字关联到进程文件描述符，用户进程通过文件描述符标识套接字。

内核提供了操作套接字的专用系统调用，用户进程也可以通过普通文件操作接口操作套接字，不管何种方式，最终都通过套接字关联的协议簇实现的操作接口实例实现对套接字的操作。

12.3 netlink 套接字

套接字可用于多个通信域，本节介绍的 netlink 套接字用于本机进程与内核之间的通信。内核网络子系统大量地采用了 netlink 套接字与用户进程通信，用户通过 netlink 套接字设置/获取网络参数、控制网络设备等。因此，在介绍网络传输套接字前，先介绍 netlink 套接字的实现，对 netlink 套接字的学习也有助于对网络套接字的学习。

netlink 套接字实现代码位于/net/netlink/目录下，主要功能在 af_netlink.c 文件内实现。

12.3.1 概述

netlink 套接字主要用于进程与内核之间的通信，也可以用于进程与进程之间通信。netlink 可视为一个协议簇（地址簇、通信域）的名称。netlink 套接字数据包只在本机内存中传递而不需要外传，因此不需要网络设备，传输协议也较简单。

netlink 套接字在内核中广泛使用，套接字定义了多个协议类型，每种协议类型适用于一个内核子系统（或者说用于实现某项内核功能）。每种协议类型下的内核套接字和用户套接字是一对多的关系，即内核为每种协议类型创建一个内核套接字，用于接收用户消息并发送应答消息。多个用户进程可同时通过同协议类型套接字与同协议类型的内核套接字通信。

同一协议类型的 netlink 套接字，不管是内核套接字还是用户套接字，在内核中都由同一个链表管理。内核建立了链表数组，每个协议类型对应一个数组项（链表），用于管理同一协议类型下的套接字。

为了寻址套接字，每个套接字被赋予一个编号（地址值）。每种协议类型的内核套接字编号固定为 0，用户套接字编号可以通过 bind()系统调用绑定，也可以由内核自动分配。

netlink 套接字之间传输数据包时，以 sk_buff 实例的形式在源套接字和目的套接字之间传递。用户套接字向内核套接字发送数据时，通过协议类型在套接字链表中查找到内核套接字（编号为 0），并调用内核套接字注册的接收（处理）消息函数接收 sk_buff 实例。如果需要发送应答消息，内核套接字将在接收函数中向用户套接字发送应答消息。netlink 套接字还支持组播操作，即将同一消息发送给指定组内的所有套接字。

netlink 套接字采用数据报类型套接字。只要内核配置选择了支持网络（NET 配置选项），netlink 套接字相关代码将自动编译入内核。netlink 套接字用于内核多个子系统，例如：网络子系统、uevent、审记子系统等，每种应用对应一个协议类型。

netlink 套接字地址簇标识为 AF_NETLINK：

```
#define   AF_NETLINK     16       /*netlink 套接字*/
```

netlink 套接字协议簇最多支持 32 个协议类型（功能应用），定义如下（/include/uapi/linux/netlink.h）：

```
#define   NETLINK_ROUTE          0 /*Routing/device hook，用于网络子系统，适用于所有协议簇*/
#define   NETLINK_UNUSED         1    /* Unused number  */
#define   NETLINK_USERSOCK       2    /*用于用户进程间通信 */
#define   NETLINK_FIREWALL       3    /* Unused number, formerly ip_queue*/
#define   NETLINK_SOCK_DIAG      4    /*套接字监视接口*/
#define   NETLINK_NFLOG          5    /* netfilter/iptables ULOG */
#define   NETLINK_XFRM           6    /*用于网络 IPsec 子系统*/
#define   NETLINK_SELINUX        7    /* SELinux event notifications，安全子系统*/
#define   NETLINK_ISCSI          8    /* Open-iSCSI */
#define   NETLINK_AUDIT          9    /* auditing，审计子系统*/
#define   NETLINK_FIB_LOOKUP     10   /*用于网络路由选择表*/
#define   NETLINK_CONNECTOR      11
#define   NETLINK_NETFILTER      12   /*用于 netfilter 子系统（网络过滤）*/
#define   NETLINK_IP6_FW         13
```

```
#define   NETLINK_DNRTMSG          14   /* DECnet routing messages */
#define   NETLINK_KOBJECT_UEVENT    15   /*内核向用户空间发送事件信息，uevent 子系统*/
#define   NETLINK_GENERIC           16   /*通用 netlink 套接字*/

#define   NETLINK_SCSITRANSPORT   18   /* SCSI Transports */
#define   NETLINK_ECRYPTFS        19
#define   NETLINK_RDMA            20
#define   NETLINK_CRYPTO          21   /* Crypto layer */
#define   NETLINK_INET_DIAG     NETLINK_SOCK_DIAG
#define   MAX_LINKS               32   /*最大协议类型数量*/
```

协议类型在用户进程创建 netlink 套接字的 socket()系统调用中设置在 protocol 参数，每种协议类型可认为是 netlink 套接字的一个具体应用，适用于一个内核子系统。

下面是利用 socket()系统调用创建 netlink 套接字的一个例子：

socket(**AF_NETLINK**,SOCK_RAW,**NETLINK_ROUTE**);

netlink 套接字地址值由 sockaddr_nl 结构体表示，定义如下（/include/upai/linux/netlink.h）：

```
struct sockaddr_nl {
    __kernel_sa_family_t    nl_family;     /*始终为 AF_NETLINK*/
    unsigned short      nl_pad;         /*始终为 0*/
    __u32           nl_pid;         /*内核套接字为 0，用户套接字可为进程 pid（是唯一值即可）*/
    __u32           nl_groups;      /*组播组掩码*/
};
```

sockaddr_nl 结构体成员简介如下：

●**nl_family**：地址簇标识，始终为 AF_NETLINK。

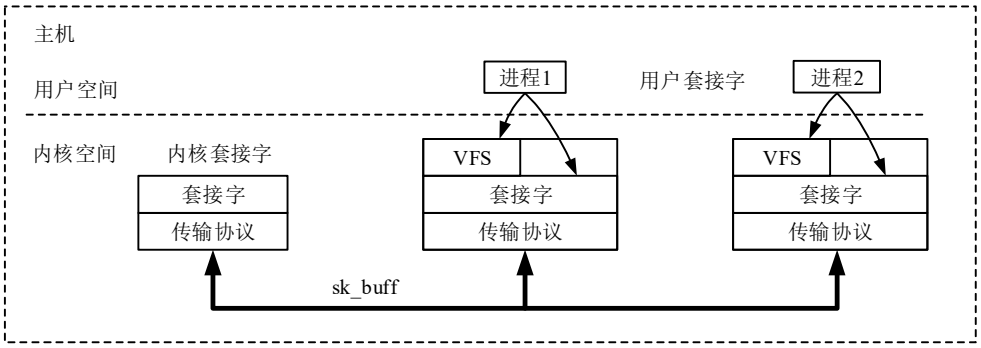
●**nl_pid**：端口 id，netlink 套接字单播地址，用于寻址套接字。对于内核套接字固定为 0，用户套接字可设置为进程 pid 值，或由进程 pid 自动生成一个值，确保此值是唯一的即可。

●**nl_groups**：组播组掩码，同一协议类型的多个 netlink 套接字可组成组播组，位图中每一位表示一个组播组编号，因此位图最多可表示 32 个组播组，组播组编号为 1~32。例如：bit0 表示组播组 1，bit31 表示组播组 32。

在绑定操作中位图中置 1 的位表示套接字监听该组播组（接受组播消息）。在发送消息的操作中，掩码表示消息要发送到哪些组播组。发送组播消息时，监听本组播组消息的套接字都可以收到发送的消息。

1 套接字框架

用户进程与内核通过 netlink 套接字通信流程，如下图所示：



内核套接字和用户套接字都由内核管理，数据以 sk_buff 实例的形式在套接字之间传递。使用 netlink

套接字的内核子系统在内核中创建一个套接字（每个网络命名空间中一个实例），用户可创建多个套接字与之通信。套接字之间通过协议类型和编号寻址。传输协议需要将传递的数据格式化，添加上 netlink 消息报头，详见下文。

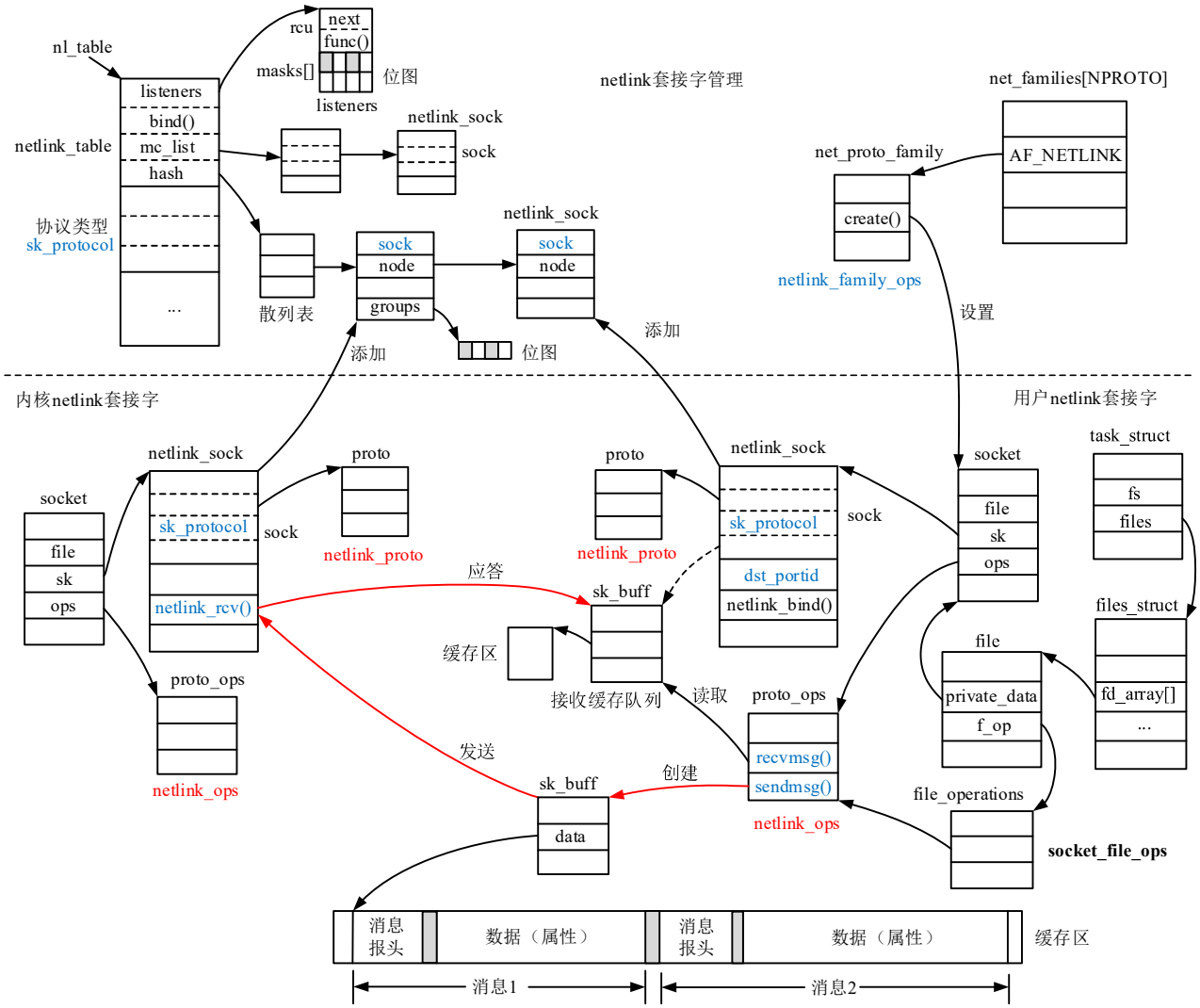
netlink 套接字数据结构组织关系如下图所示。netlink 套接字传输协议中私有数据结构为 netlink_sock，其中内嵌 sock 结构体成员，在传输协议中表示套接字。

内核建立了 netlink_table 结构体数组，每个数组项对应一个协议类型。netlink_table 实例中包含一个散列表，用于管理同一协议类型下的所有 netlink_sock 实例。另外，netlink_table 实例中还包含一个 mc_list 链表，用于管理监听了组播消息的套接字，在发送组播消息时将扫描此链表中的实例，对监听了本组播消息的套接字发送一个消息副本。

内核通过接口函数创建 netlink 套接字，用户进程通过 socket()系统调用创建套接字。netlink 套接字注册的 net_proto_family 实例中的 create()函数将为用户套接字创建 netlink_sock 实例并添加到 netlink_table 实例中的管理结构。套接字关联的 proto_ops 实例为 netlink_ops。

如果内核套接字需要接收用户套接字发送的消息，需定义接收函数，在创建套接字时赋予 netlink_sock 实例中的 netlink_rcv()函数指针成员。用户进程向内核套接字发送消息时，将调用内核套接字 netlink_sock 实例中的 netlink_rcv()函数，由内核处理用户发送的消息，如果需要发送应答消息，也将在 netlink_rcv()函数中发送。应答消息 sk_buff 实例将添加到用户进程套接字接收消息缓存队列，用户进程从中读取消息。

netlink 套接字支持组播功能。协议类型对应的 netlink_table 实例中包含一个监听位图，位图的大小表示协议类型支持的最大组播组数量，每个比特位表示一个组播组。比特位的位置加 1 表示组播组编号。置位的组播组表示有套接字在监听本组播组的消息。在表示 netlink 套接字的 netlink_sock 实例中也包含一个位图，表示本套接字监听的组播组。套接字需要组播消息时，在表示目的套接字地址的 sockaddr_nl 实例的 nl_groups 位图成员中指明组播到哪些组。内核组播消息时会扫描协议类型 netlink_table 实例 mc_list 链表中的 netlink_sock 实例，如果其监听位图中本组播组标记位置位了，则复制一个 sk_buff 实例添加到其接收缓存队列中，以发送组播消息。用户进程从用户套接字接收缓存队列中读取消息数据。



2 数据结构

下面介绍 netlink 套接字中主要的数据结构。

■netlink_sock

在 netlink 套接字传输协议中，套接字由 netlink_sock 结构体表示，定义如下（/net/netlink/af_netlink.h）：

```
struct netlink_sock {
    struct sock     sk;     /*sock 结构体成员，必须是第一个成员*/
    u32             portid;     /*源（本）套接字 id，内核套接字为 0，在创建套接字时赋值*/
    u32             dst_portid;  /*目的套接字 id，用于查找目的套接字 netlink_sock 实例*/
    u32             dst_group;   /*组播组位图中第一个置位的位（位置）*/
    u32             flags;  /*标记，NETLINK_F_KERNEL_SOCKET 等，/net/netlink/af_netlink.c*/
    u32             subscriptions;   /*将实例添加到 mc_list 链表时赋值*/
    u32             ngroups;        /*监听组播组的最大编号*/
    unsigned long       *groups;  /*指向监听组播组位图，位图 4 字节对齐，每位代表一个组*/
    unsigned long       state;      /*状态，NETLINK_S_CONGESTED，/net/netlink/af_netlink.c*/
    size_t              max_recvmsg_len;
    wait_queue_head_t wait;   /*等待队列，管理因拥塞而在套接字上睡眠的发送消息进程*/
```

```
        bool            bound;    /*是否已绑定地址*/
        bool            cb_running;
        struct netlink_callback   cb;    /*/include/linux/netlink.h*/
        struct mutex        *cb_mutex;
        struct mutex        cb_def_mutex;
        void        (*netlink_rcv)(struct sk_buff *skb);      /*内核套接字接收消息时调用此函数*/
        int         (*netlink_bind)(struct net *net, int group);   /*绑定组播组，标记组播组被监听*/
        void        (*netlink_unbind)(struct net *net, int group);   /*解绑组，标记组播组没有被监听*/
        struct module       *module;
    #ifdef CONFIG_NETLINK_MMAP
        struct mutex        pg_vec_lock;
        struct netlink_ring  rx_ring;
        struct netlink_ring  tx_ring;
        atomic_t        mapped;
    #endif

        struct rhash_head   node;    /*将实例添加到 rhashtable 散列表，见下文*/
        struct rcu_head     rcu;
    };
```

netlink_sock 结构体主要成员简介如下：

●**sk**：sock 结构体成员，必须是第一个成员。

●**portid**：本套接字 id，**内核套接字为 0，在创建套接字时赋值**。

●**dst_portid**：目的套接字 id，用于查找目的套接字 netlink_sock 实例。

●**netlink_rcv()**：内核套接字接收消息的处理函数。

●**node**：将实例添加到 rhashtable 散列表，见下文。

●**flags**：标记，标记位语义定义如下（/net/netlink/af_netlink.c）：

```
#define   NETLINK_F_KERNEL_SOCKET          0x1        /*内核套接字*/
#define   NETLINK_F_RECV_PKTINFO           0x2
#define   NETLINK_F_BROADCAST_SEND_ERROR   0x4
#define   NETLINK_F_RECV_NO_ENOBUFS        0x8
#define   NETLINK_F_LISTEN_ALL_NSID        0x10
```

●**state**：状态，比特位语义如下（/net/netlink/af_netlink.c）：

```
#define   NETLINK_S_CONGESTED      0x0        /*bit0，表示套接字是否拥塞*/
```

■netlink_table

netlink_table 结构体用于构成全局散列表，每个协议类型对应一个实例，用于管理同一协议类型下的套接字 netlink_sock 实例。结构体中还定义了某些统一的属性，定义如下（/net/netlink/af_netlink.h）：

```
    struct netlink_table {
        struct rhashtable    hash;    /*散列表，管理 netlink_sock 实例，/include/linux/rhashtable.h*/
        struct hlist_head    mc_list;   /*监听组播消息的 netlink_sock 实例链表*/
        struct listeners __rcu     *listeners;    /*指向 listeners 实例（链表），/net/netlink/af_netlink.c*/
        unsigned int         flags;    /*标记，来自 netlink_kernel_cfg.flags*/
        unsigned int         groups;    /*组播组编号最大值*/
        struct mutex         *cb_mutex;
```

```
    struct module       *module;
    int             (*bind)(struct net *net, int group);   /*来自 netlink_kernel_cfg 实例，绑定组*/
    void            (*unbind)(struct net *net, int group);    /*来自 netlink_kernel_cfg 实例，解绑组*/
    bool            (*compare)(struct net *net, struct sock *sock);    /*来自 netlink_kernel_cfg 实例*/
    int             registered;      /*netlink_table 实例是否已注册（初始化）*/
};
```

netlink_table 结构体主要成员简介如下：

●**hash**：rhashtable 结构体成员，rhashtable 散列表用于管理同一协议类型的 netlink_sock 实例（包含所有网络命名空间中同一协议类型下的套接字），详见下文。

●**mc_list**：用于管理监听了组播消息的 netlink_sock 实例（netlink_sock.sk），见下文。

●**groups**：被监听组播组的最大编号，决定位图的大小。

●**listeners**：指向 listeners 结构体成员（链表），结构体定义在/net/netlink/af_netlink.c 文件内：

```
struct listeners {
    struct rcu_head         rcu;   /*包含 next 指针，回调函数 func()指针成员，/include/linux/types.h*/
    unsigned long       masks[0];
                          /*位图（4 字节对齐），每位标记一个组播组，置位表示组播组被监听*/
};
```

listeners 结构体中主要包含一个位图，位图用于标记所有被监听的组播组，比特位的位置（加 1）表示组播组编号，例如：bit0 表示组播组 1，bit1 表示组播组 2。

●**flags**：标记成员，取值定义在/include/linux/netlink.h 头文件，表示非超级用户是否有向该协议类型套接字发送组播消息、接收组播消息的权限：

```
#define   NL_CFG_F_NONROOT_RECV   (1 << 0)    /*具有发送组播消息权限*/
#define   NL_CFG_F_NONROOT_SEND   (1 << 1)    /*具有接收组播消息权限*/
```

●**bind()**：标记某个组播组被监听，例如置位 listeners 指向位图中该组的标记位。

●**unbind()**：解绑组播组，例如清零 listeners 指向位图中该组的标记位。

●**registered**：netlink_table 实例是否已注册（初始化）。

内核在 netlink 套接字初始化时创建全局 netlink_table 结构体实例数组。在创建该协议类型的第一个**内核**套接字时，将注册（初始化）协议类型对应的 netlink_table 实例，详见下一小节。

netlink_table 结构体中有的成员值来自于 netlink_kernel_cfg 实例，netlink_kernel_cfg 实例用于内核初始化 netlink_table 实例时传递参数使用，详见下文。

3 初始化

内核在/net/netlink/af_netlink.c 文件内定义了 netlink 套接字的初始化函数，代码如下：

```
static int __init netlink_proto_init(void)
{
    int i;
    int err = proto_register(&netlink_proto, 0);       /*注册 proto 结构体实例 netlink_proto*/
    ...
    /*分配 netlink_table 实例数组*/
    nl_table = kcalloc(MAX_LINKS, sizeof(*nl_table), GFP_KERNEL);
    ...
    for (i = 0; i < MAX_LINKS; i++) {     /*初始化 netlink_table 数组*/
      if (rhashtable_init(&nl_table[i].hash,&netlink_rhashtable_params) < 0) {
                                  /*初始化 netlink_table 实例中散列表，见下文，/lib/rhashtable.c*/
```

```
            while (--i > 0)
                rhashtable_destroy(&nl_table[i].hash);
            kfree(nl_table);
            goto panic;
        }
    }

    INIT_LIST_HEAD(&netlink_tap_all);

    netlink_add_usersock_entry();  /*设置 NETLINK_USERSOCK 协议类型对应的 netlink_table 实例*/

    sock_register(&netlink_family_ops);     /*注册 netlink 地址簇 net_proto_family 实例*/
    register_pernet_subsys(&netlink_net_ops);
                /*注册 pernet_operations 实例，其初始化函数在 proc 文件系统内创建 netlink 目录项*/

    rtnetlink_init();     /*创建 NETLINK_ROUTE 内核套接字，用于网络子系统，/net/core/rtnetlink.c*/
out:
    return err;
    ...
}
core_initcall(netlink_proto_init);      /*内核启动阶段调用此函数*/
```

初始化函数中创建并初始化了 netlink_table 实例数组，注册了 netlink 地址簇的 net_proto_family 实例，创建了 NETLINK_ROUTE 协议类型的内核 netlink 套接字等。

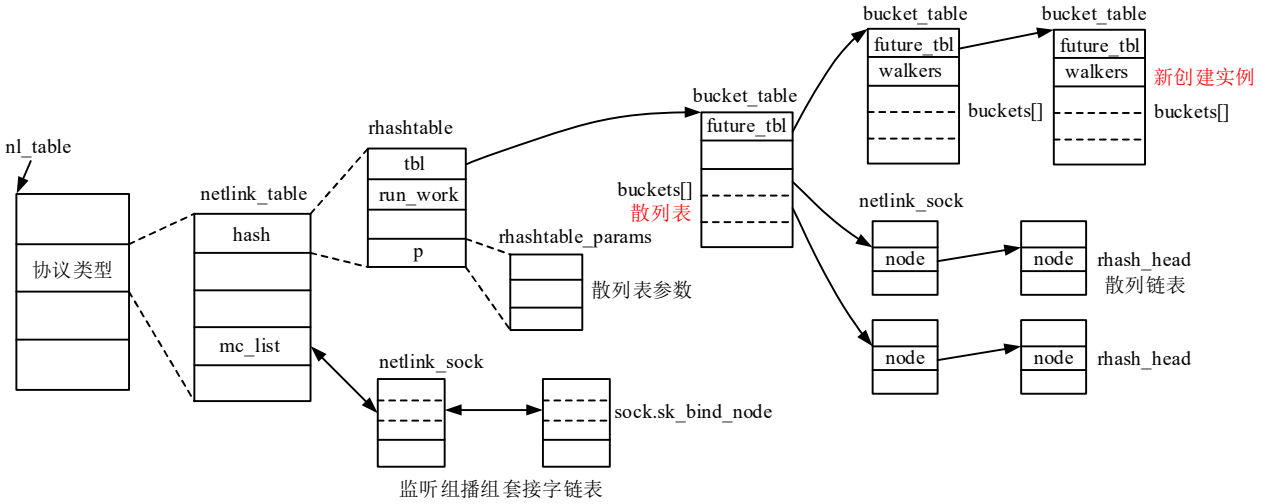
初始化函数中注册了 netlink 套接字关联的 proto 结构体实例 **netlink_proto**，定义如下：

```
static struct proto netlink_proto = {
    .name       = "NETLINK",
    .owner      = THIS_MODULE,
    .obj_size = sizeof(struct netlink_sock),      /*私有数据结构大小（netlink_sock）*/
};
```

在分配 sock 实例的接口函数 sk_alloc()中将分配 obj_size 大小的空间，即 netlink_sock 结构体大小，结构体中内嵌 sock 结构体成员。

4 管理 netlink_sock 实例

每种协议类型的 netlink 套接字 netlink_sock 实例由 netlink_table 实例中的散列表管理，如下图所示：



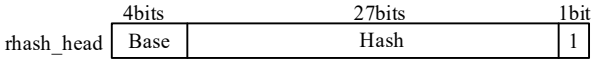
netlink_table 结构体中通过 rhashtable 散列表（hash 成员）管理 netlink_sock 实例。rhashtable 结构体中 p 成员为 rhashtable_params 结构体，包含散列表的配置参数，配置参数可由使用 rhashtable 散列表的子系统设置。rhashtable 结构体中包含一个 bucket_table 结构体实例单链表，此结构体中 buckets[]数组表示真正的散列表。

buckets[]数组表示的散列链表，是 rhash_head 结构体指针数组，结构体定义如下：

```
struct rhash_head {
    struct rhash_head __rcu *next;     /*指向下一个节点，组成单链表*/
};
```

散列链表由 rhash_head 实例构成，netlink_sock 结构体中 node 成员（rhash_head 结构体）插入到散列链表中。散列表 buckets[]数组项为指向 rhash_head 结构体的指针。

新添加的实例添加到散列链表的头部，末尾节点 rhash_head 实例赋值如下图所示：



bit0 为 1 表示此节点是链表末尾节点。bit[1,27]表示散列值，即 buckets[]数组索引值。bit[31...28]为基准值，用于区分多个散列表，其值来源于配置参数 rhashtable_params.nulls_base 成员。

netlink_sock 实例以其所在的网络命名空间和 id 值作为键值，计算散列值，插入散列值对应的散列链表头部。查找实例时，也以网络命名空间和 id 值作为键值，在散列表中查找。

rhashtable 结构体中包含一个 bucket_table 实例链表（tbl 为链表头指针），当需要扩展或收缩散列表时，将创建新 bucket_table 实例，添加到 bucket_table 实例链表末尾，并激活 run_work 工作（rhashtable 结构体成员）。

run_work 工作将迭代执行，其处理函数主要工作是将 bucket_table 实例链表第一个成员散列表中管理对象迁移到第二个实例的散列表中，从链表中移除第一个成员，此时第二个成员成了第一个成员。也就是说 run_work 工作将迭代执行至 bucket_table 实例链表中只剩一个成员。

netlink_table 结构体中 mc_list 成员（双链表）管理监听了组播消息的套接字 netlink_sock 实例。在绑定套接字时，将调用 netlink_update_subscriptions()函数将 netlink_sock 实例添加到 **mc_list** 双链表，实例通过 netlink_sock.sk.**sk_bind_node** 成员添加到 mc_list 双链表。

下面主要介绍 rhashtable 散列表的实现，内核其他地方也有使用 rhashtable 散列表的情形。

■散列表

下面先介绍 rhashtable 散列表使用的数据结构，散列值的计算，散列表的初始化，然后将介绍利用 rhashtable 散列表管理 netlink_sock 实例的操作函数。

rhashtable 散列表操作函数主要定义在/lib/rhashtable.c 文件内和/include/linux/rhashtable.h 头文件。

rhashtable 结构体表示散列表句柄，结构体定义如下（/include/linux/rhashtable.h）：

```
struct rhashtable {
    struct bucket_table __rcu    *tbl;    /*指向 bucket_table 结构体，内含散列表*/
    atomic_t             nelems;          /*散列表中管理对象数量*/
    unsigned int         key_len;         /*散列函数中键值（参数）长度*/
    unsigned int         elasticity;      /**/
    struct rhashtable_params   p;    /*散列表配置参数*/
    struct work_struct         run_work;   /*工作队列，用于重构散列表*/
    struct mutex               mutex;
    spinlock_t                 lock;
};
```

rhashtable 结构体中主要成员简介如下：

●**tbl**：指向 bucket_table 结构体成员，结构体定义如下：

```
struct bucket_table {
    unsigned int         size;    /*buckets[]成员中所含散列链表数量，即数组项数*/
    unsigned int         rehash;    /*当前散列表已迁移散列链表数量，在 run_work 工作中增加此值*/
    u32              hash_rnd;       /*随机产生的键值，在调用 obj_hashfn()函数时使用*/
    unsigned int         locks_mask;
    spinlock_t           *locks;
    struct list_head     walkers;    /*活跃的 walkers*/
    struct rcu_head            rcu;
    struct bucket_table __rcu *future_tbl;
                                    /*指向扩展的 bucket_table 实例，组成 bucket_table 实例链表*/
    struct rhash_head __rcu  *buckets[] ____cacheline_aligned_in_smp;
                                        /*表示散列表，rhash_head 实例指针数组*/
};
```

buckets[]成员是一个指针数组，数组项指向 rhash_head 实例组成的单链表。

●**p**：配置参数 rhashtable_params 结构体成员，结构体定义如下：

```
struct rhashtable_params {
    size_t               nelem_hint;    /*暗示管理对象数量，应设为散列链表数量的 75%*/
    size_t               key_len;      /*键值长度*/
    size_t               key_offset;   /*表示键值的成员在其所在结构体内的偏移量（字节数）*/
    size_t               head_offset;  /*rhash_head 成员在管理对象结构体内的偏移量（字节数）*/
    unsigned int         insecure_max_entries;      /*散列表管理对象最大数量*/
    unsigned int         max_size;     /*扩展散列表时 buckets[]项数最大值*/
    unsigned int         min_size;     /*收缩散列表时 buckets[]项数最小值*/
    u32              nulls_base;       /*基准值，赋予散列链表中末尾节点，设置 bit[30...27]*/
    bool             insecure_elasticity;      /*是否禁止散列链表长度检查*/
    bool             automatic_shrinking;   /*是否自动收缩散列表*/
    size_t               locks_mul;
    rht_hashfn_t         hashfn;      /*计算散列值函数，即 buckets[]数组项索引值*/
    rht_obj_hashfn_t     obj_hashfn;   /*散列函数*/
    rht_obj_cmpfn_t      obj_cmpfn;   /*通过键值比对对象函数*/
```

};

由 rhashtable 散列表管理的对象数据结构中，需要由一个 rhash_head 结构体成员用于插入到散列链表。key_offset、head_offset 成员分别表示这两个成员在其所在数据结构中的偏移量（字节数）。

rht_head_hashfn()接口函数用于计算散列值，即管理对象实例所在链表在 buckets[]数组中索引值，函数定义如下（/include/linux/rhashtable.h）：

```
static inline unsigned int rht_head_hashfn(struct rhashtable *ht, const struct bucket_table *tbl,
                                           const struct rhash_head *he, const struct rhashtable_params params)
/*he：管理对象中内嵌的 rhash_head 结构体成员，params：参数传递的配置参数，非 ht 中配置参数*/
{
    const char *ptr = rht_obj(ht, he);     /*管理对象实例指针*/

    return likely(params.obj_hashfn) ?
           rht_bucket_index(tbl, params.obj_hashfn(ptr, params.key_len ?:ht->p.key_len,tbl->hash_rnd)) :
           rht_key_hashfn(ht, tbl, ptr + params.key_offset, params);
}
```

如果配置参数 params 中定义了 obj_hashfn()函数，则依此函数返回值计算散列值，若没有定义此函数则调用 rht_key_hashfn()函数计算散列值。rht_key_hashfn()函数将依据 params.hashfn()或 jhash()或 jhash2()函数返回值计算散列值。以上函数都定义在/include/linux/rhashtable.h 头文件内，请读者自行阅读。

■初始化散列表

初始化散列表句柄 rhashtable 实例的函数 **rhashtable_init()**定义如下（/lib/rhashtable.c）：

```
int rhashtable_init(struct rhashtable *ht,const struct rhashtable_params *params)
{
    struct bucket_table *tbl;
    size_t size;     /*散列链表数量，即 tbl->buckets[]数组项数*/

    size = HASH_DEFAULT_SIZE;     /*散列链表默认数量为 64，/lib/rhashtable.c*/

    if ((!params->key_len && !params->obj_hashfn) ||
        (params->obj_hashfn && !params->obj_cmpfn))
        return -EINVAL;

    if (params->nulls_base && params->nulls_base < (1U << RHT_BASE_SHIFT))    /*27*/
        return -EINVAL;

    if (params->nelem_hint)       /*暗示散列链表数量*/
        size = rounded_hashtable_size(params);

    memset(ht, 0, sizeof(*ht));
    mutex_init(&ht->mutex);
    spin_lock_init(&ht->lock);
    memcpy(&ht->p, params, sizeof(*params));     /*复制配置参数至 rhashtable 实例*/

    /*以下是设置 rhashtable 实例中配置参数（p 成员）*/
```

```
    if (params->min_size)
        ht->p.min_size = roundup_pow_of_two(params->min_size);     /*散列表最小项数，2 的对数*/

    if (params->max_size)
        ht->p.max_size = rounddown_pow_of_two(params->max_size);   /*散列表最大项数*/

    if (params->insecure_max_entries)       /*最大散列表项数*/
        ht->p.insecure_max_entries =rounddown_pow_of_two(params->insecure_max_entries);
    else
        ht->p.insecure_max_entries = ht->p.max_size * 2;

    ht->p.min_size = max(ht->p.min_size, HASH_MIN_SIZE);    /*HASH_MIN_SIZE 值为 4*/
    if (!params->insecure_elasticity)
        ht->elasticity = 16;

    if (params->locks_mul)
        ht->p.locks_mul = roundup_pow_of_two(params->locks_mul);
    else
        ht->p.locks_mul = BUCKET_LOCKS_PER_CPU;

    ht->key_len = ht->p.key_len;     /*键值长度*/
    if (!params->hashfn) {         /*如果配置参数没有定义散列函数*/
        ht->p.hashfn = jhash;

        if (!(ht->key_len & (sizeof(u32) - 1))) {
            ht->key_len /= sizeof(u32);
            ht->p.hashfn = rhashtable_jhash2;
        }
    }

    tbl = bucket_table_alloc(ht, size, GFP_KERNEL);    /*/lib/rhashtable.c*/
          /*分配并初始化 bucket_table 实例，size 表示 buckets[]数组项数，即散列链表数量*/
    if (tbl == NULL)
        return -ENOMEM;

    atomic_set(&ht->nelems, 0);     /*散列链表中实例数量初始值为 0*/
    RCU_INIT_POINTER(ht->tbl, tbl);   /*bucket_table 实例指针赋予 ht->tbl*/
    INIT_WORK(&ht->run_work, rht_deferred_worker);     /*/lib/rhashtable.c*/
                                /*初始化工作实例，执行函数为 rht_deferred_worker()*/
    return 0;
}
```

rhashtable_init()函数设置 rhashtable 实例（主要是参数成员），创建并初始化 bucket_table 实例（内含散列表），初始化工作队列 ht->run_work，其执行函数为 rht_deferred_worker()，用于重构散列表。

bucket_table_alloc()函数用于分配并初始化 bucket_table 实例，实例中 buckets[]数组项数为 size，数组项值设为散列链表末尾节点的值（链表为空）。

■插入散列表

下面结合此处 rhashtable 的管理对象 netlink_sock 实例，看一下如何向散列表插入对象。

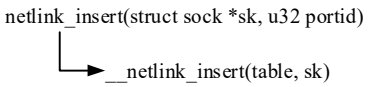
在前面介绍的 netlink 套接字初始化函数中，将对每个 netlink_table 实例 hash 成员调用以下函数，初始化散列表成员：

rhashtable_init(&nl_table[i].hash,**&netlink_rhashtable_params**)

散列表配置参数 rhashtable_params 结构体实例为 netlink_rhashtable_params，定义如下：

```
static const struct rhashtable_params netlink_rhashtable_params = {
    .head_offset = offsetof(struct netlink_sock, node),   /*rhash_head 结构体成员偏移量（字节数）*/
    .key_len = netlink_compare_arg_len,   /*键值长度，netlink_compare_arg 结构体长度*/
    .obj_hashfn = netlink_hash,          /*散列函数，调用 jhash2()函数*/
    .obj_cmpfn = netlink_compare,        /*对象比对函数*/
    .automatic_shrinking = true,   /*允许自动收缩散列表*/
};
```

在创建内核 netlink 套接字和绑定用户套接字（含自动绑定）时，调用 netlink_insert()函数将 netlink_sock 实例插入到 netlink_table.hash 的散列表中，函数调用关系如下图所示：



__netlink_insert()函数定义如下（/net/netlink/af_netlink.c）：

```
static int __netlink_insert(struct netlink_table *table, struct sock *sk)
/*table：协议类型对应的 netlink_table 实例，sk：指向 netlink_sock 实例中内嵌 sock 结构体成员*/
{
    struct netlink_compare_arg arg;      /*比较参数结构体，见下文*/

    netlink_compare_arg_init(&arg, sock_net(sk), nlk_sk(sk)->portid);   /*初始化比较参数*/
    return rhashtable_lookup_insert_key(&table->hash, &arg,&nlk_sk(sk)->node,
                                        netlink_rhashtable_params);
}
```

__netlink_insert()函数内 arg 变量为比较参数 netlink_compare_arg 结构体实例，结构体定义如下：

```
struct netlink_compare_arg          /*/net/netlink/af_netlink.c*/
{
    possible_net_t pnet;      /*网络命名空间指针*/
    u32 portid;               /*套接字 id*/
};
```

netlink_compare_arg 结构体表示比较（查找）套接字的键值，包含网络命名空间和套接字 id 值。因为不同的网络命名空间中可以有相同 id 值的套接字，因此需要加上网络命名空间参数。

netlink_compare_arg_init()函数用于初始化 arg 变量，变量 pnet 成员为套接字所在网络命名空间，portid 为套接字 id 值。

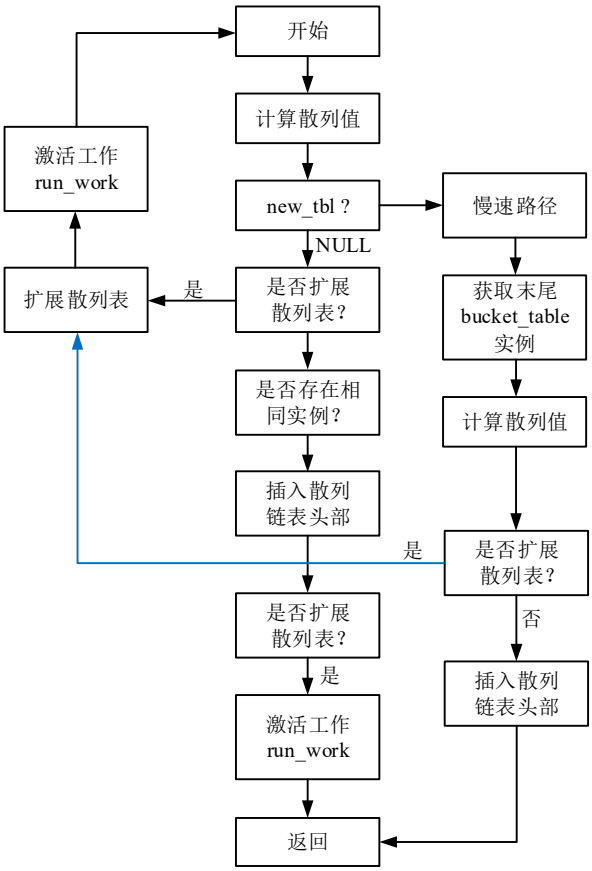
rhashtable_lookup_insert_key()函数定义在/include/linux/rhashtable.h 头文件：

```
static inline int rhashtable_lookup_insert_key(struct rhashtable *ht, const void *key, struct rhash_head *obj,
                                               const struct rhashtable_params params)
```

```
/*
*ht：指向 rhashtable 实例，key：比对结构指针，此处指向 netlink_compare_arg 实例，
*obj：指向对象实例（netlink_sock）中 rhash_head 成员，params：配置参数。
*/
{
    BUG_ON(!ht->p.obj_hashfn || !key);

    return __rhashtable_insert_fast(ht, key, obj, params);      /*/include/linux/rhashtable.h*/
}

__rhashtable_insert_fast()函数定义如下：
static inline int __rhashtable_insert_fast(struct rhashtable *ht, const void *key, struct rhash_head *obj,
                                           const struct rhashtable_params params)
{
    struct rhashtable_compare_arg arg = {           /*通用的比较参数结构*/
        .ht = ht,           /*指向 rhashtable 实例*/
        .key = key,         /*指向 netlink_compare_arg 实例*/
    };
    struct bucket_table *tbl, *new_tbl;
    struct rhash_head *head;
    spinlock_t *lock;
    unsigned int elasticity;
    unsigned int hash;     /*散列表索引值，即 buckets[]数组项索引值*/
    int err;

restart:
    rcu_read_lock();
    tbl = rht_dereference_rcu(ht->tbl, ht);     /*bucket_table 实例*/

    for (;;) {     /*计算散列值*/
        hash = rht_head_hashfn(ht, tbl, obj, params);   /*计算散列值，/include/linux/rhashtable.h*/
        lock = rht_bucket_lock(tbl, hash);
        spin_lock_bh(lock);

        if (tbl->rehash <= hash)   /*已迁移散列链表值小于等于 hash，表示当前散列链表没有迁移*/
            break;

        spin_unlock_bh(lock);
        tbl = rht_dereference_rcu(tbl->future_tbl, ht);
                                    /*当前散列链表已迁移，获取 tbl->future_tbl 实例*/
    }

    new_tbl = rht_dereference_rcu(tbl->future_tbl, ht);   /*新构建的 bucket_table 实例*/
    if (unlikely(new_tbl)) {        /*如果存在新构建 bucket_table 实例，执行慢操作*/
        err = rhashtable_insert_slow(ht, key, obj, new_tbl);   /*/lib/rhashtable.c*/
        if (err == -EAGAIN)       /*返回-EAGAIN 需要扩展散列表*/
```

```
                goto slow_path;     /*跳转至 slow_path 处执行*/
            goto out;
        }

        err = -E2BIG;
        if (unlikely(rht_grow_above_max(ht, tbl)))     /*ht->nelems 值超过最大值，插入失败*/
            goto out;                                  /*ht->nelems>= ht->p.insecure_max_entries*/

        if (unlikely(rht_grow_above_100(ht, tbl))) {   /*ht->nelems > tbl->size，散列表扩展一倍*/
slow_path:
            spin_unlock_bh(lock);
            err = rhashtable_insert_rehash(ht);
                /*分配新 bucket_table 实例（扩展散列表），激活 ht->run_work 工作迁移散列表*/
            rcu_read_unlock();
            if (err)
                return err;
            goto restart;     /*重新执行插入操作*/
        }

        err = -EEXIST;
        elasticity = ht->elasticity;     /*散列链表中对象成员最大值*/
        rht_for_each(head, tbl, hash) {       /*在散列表中查找是否存在相同实例*/
            if (key &&unlikely(!(params.obj_cmpfn ?params.obj_cmpfn(&arg, rht_obj(ht, head)) :
                        rhashtable_compare(&arg, rht_obj(ht, head)))))
                goto out;
            if (!--elasticity)
                goto slow_path;
        }
        /*不存在相同实例，将实例插入 tbl->buckets[hash]散列链表头部，增加对象数量计数*/
        err = 0;
        head = rht_dereference_bucket(tbl->buckets[hash], tbl, hash);
        RCU_INIT_POINTER(obj->next, head);     /*初始化对象实例中 rhead_head 成员*/
        rcu_assign_pointer(tbl->buckets[hash], obj);
        atomic_inc(&ht->nelems);         /*增加对象数量计数值*/
        if (rht_grow_above_75(ht, tbl))       /*激活工作 ht->run_work（ht->nelems >0.75 * tbl->size）*/
            schedule_work(&ht->run_work);
out:
        spin_unlock_bh(lock);
        rcu_read_unlock();
        return err;
}
```

__rhashtable_insert_fast()函数执行流程如下图所示：



__rhashtable_insert_fast()函数首先计算散列值时，然后判断 new_tbl 是否为 NULL，为 NULL 表示没有新创建的 bucket_table 实例，进入快速路径，不为 NULL 则进入慢速路径。

在快速路径中将判断是否要扩展散列表，如果需要则创建新 bucket_table 实例（散列表扩展一倍），新实例添加到 **future_tbl** 单链表末尾，激活 run_work 工作，然后回到函数开始处。在 run_work 工作中将迁移旧 bucket_table 实例中散列表中的对象实例迁移到新 bucket_table 实例。

如果不需要扩展散列表，则将对象实例添加到指定散列链表头部，然后判断是否要扩展散列表，如果需要则激活 run_work 工作，不需要则直接返回。

在慢速路径中将获取 bucket_table 实例单链表末尾实例，依此重新计算散列值，然后判断是否要扩展散列表，需要则扩展散列表，重新回到函数开始处。如果不需要扩展散列表，则将对象实例添加到此 bucket_table 实例的散列表中，函数返回。

以下接口函数用于从散列表查找查找和移除实现，源代码请读者自行阅读：

●void ***rhashtable_lookup_fast**(struct rhashtable *ht, const void *key,const struct rhashtable_params params)：快速查找管理对象实例。

●int **rhashtable_remove_fast**(struct rhashtable *ht, struct rhash_head *obj,const struct rhashtable_params params)：移除管理对象实例。

■重构散列表

初始化散列表句柄 rhashtable 实例的 **rhashtable_init()**函数中，rhashtable 实例 run_work 工作处理函数设为 rht_deferred_worker()，用于重构散列表（扩展、收缩散列表，迁移对象），函数定如下（/lib/rhashtable.c）：

```
static void rht_deferred_worker(struct work_struct *work)
{
    struct rhashtable *ht;
    struct bucket_table *tbl;
```

```
    int err = 0;

    ht = container_of(work, struct rhashtable, run_work);   /*rhashtable 实例*/
    mutex_lock(&ht->mutex);

    tbl = rht_dereference(ht->tbl, ht);     /*bucket_table 实例*/
    tbl = rhashtable_last_table(ht, tbl);   /*bucket_table 实例链表中末尾成员*/

    if (rht_grow_above_75(ht, tbl))     /*nelems > 0.75 * tbl->size*/
        rhashtable_expand(ht);      /*扩展散列表，创建新 rhashtable 实例，散列链表数量翻倍*/
    else if (ht->p.automatic_shrinking && rht_shrink_below_30(ht, tbl))     /*nelems <0.3 * tbl->size*/
        rhashtable_shrink(ht);      /*收缩散列表*/

    err = rhashtable_rehash_table(ht);
                    /*重构散列表，迁移散列表等，返回 -EAGAIN 表示还要继续执行迁移*/
    mutex_unlock(&ht->mutex);

    if (err)    /*还要继续迁移，再激活 run_work 工作*/
        schedule_work(&ht->run_work);
}
```

rht_deferred_worker()函数中首先执行扩展、收缩散列表，然后调用 rhashtable_rehash_table(ht)函数迁移管理对象到新 rhashtable 实例散列表中。

rhashtable_rehash_table(ht)函数定义如下：

```
static int rhashtable_rehash_table(struct rhashtable *ht)
{
    struct bucket_table *old_tbl = rht_dereference(ht->tbl, ht);
    struct bucket_table *new_tbl;
    struct rhashtable_walker *walker;
    unsigned int old_hash;

    new_tbl = rht_dereference(old_tbl->future_tbl, ht);     /*bucket_table 实例链表中第二个成员*/
    if (!new_tbl)
        return 0;

    for (old_hash = 0; old_hash < old_tbl->size; old_hash++)     /*迁移散列表*/
        rhashtable_rehash_chain(ht, old_hash);   /*迁移散列链表中对象，需要重新计算散列值*/

    rcu_assign_pointer(ht->tbl, new_tbl);     /*第二 bucket_table 实例设为第一个成员*/

    spin_lock(&ht->lock);
    list_for_each_entry(walker, &old_tbl->walkers, list)
        walker->tbl = NULL;
    spin_unlock(&ht->lock);
    call_rcu(&old_tbl->rcu, bucket_table_free_rcu);     /*释放旧 bucket_table 实例*/

    return rht_dereference(new_tbl->future_tbl, ht) ? -EAGAIN : 0;
```

/*返回 -EAGAIN 表示需要再执行一遍*/

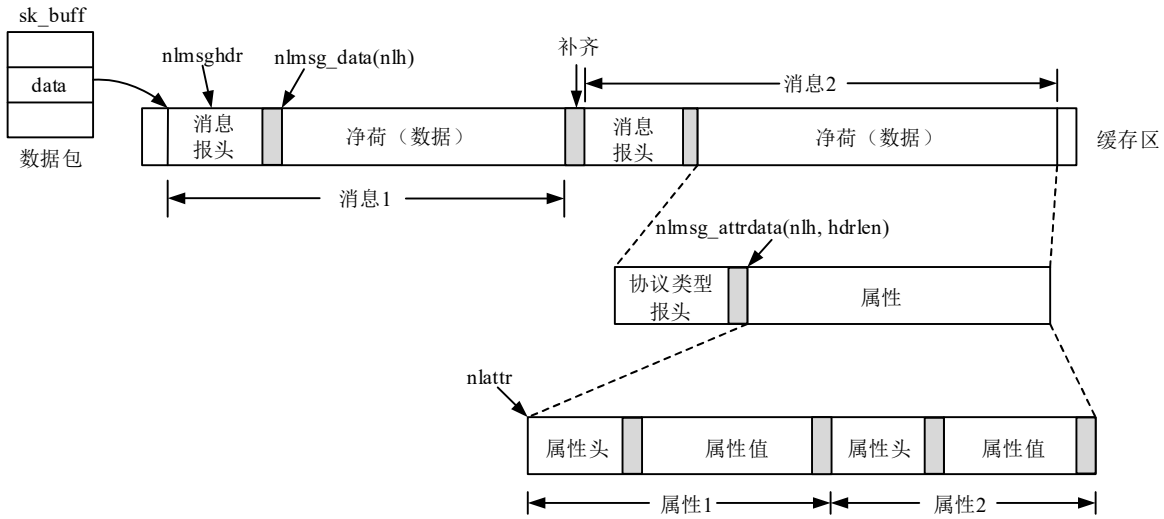
}

rhashtable_rehash_table(ht)函数的主要工作就是将 rhashtable 实例 bucket_table 实例链表中第一个成员中散列表中的管理对象迁移到第二个成员中，释放第一个成员 bucket_table 实例，而第二个成员此时成为第一个成员。如果此时 bucket_table 实例链表中还有第二个成员，则返回-EAGAIN 值。

如果 rhashtable_rehash_table(ht)函数返回-EAGAIN 值，将再次激活 run_work 工作，rht_deferred_worker()函数将再次执行。也就是说，run_work 工作将重复地执行，直到 rhashtable 实例 bucket_table 实例链表中只剩一个成员为止。

5 消息格式

netlink 套接字通过消息的形式传递数据，消息保存在 sk_buff 实例关联的内存缓存区中，一个缓存区可包含多个消息。消息格式如下图所示（/include/net/netlink.h）：



每个消息由两部分组成：消息报头和净荷（数据），每个部分都要满足对齐要求，通常是 4 字节对齐（/include/uapi/linux/netlink.h）。消息报头是所有协议类型公用的 netlink 消息报头。

数据区包含特定于协议类型的报头和属性，属性区域中可能包含多个属性，每个属性又包含属性头和属性值，各区域都要满足数据对齐要求。下面将详细介绍各区域的数据格式定义。

■消息报头

netlink 消息报头由 nlmsghdr 结构体表示，sk_buff 实例中的 data 成员指向消息报头的开始处。

nlmsghdr 结构体定义如下（/include/upai/linux/netlink.h）：

```
struct nlmsghdr {
    __u32       nlmsg_len;    /*加上报头的（单个）消息长度，不含补齐字节*/
    __u16       nlmsg_type;   /*消息类型*/
    __u16       nlmsg_flags;  /*标记，取值定义在/include/upai/linux/netlink.h 头文件*/
    __u32       nlmsg_seq;    /*序列号，用于排列消息，不要求必须使用*/
    __u32       nlmsg_pid;    /*发送套接字编号，内核为 0 */
};
```

nlmsghdr 结构体主要成员简介如下：

●**nlmsg_type**：协议类型下的消息类型，定义在/include/uapi/linux/netlink.h 头文件：

```
#define   NLMSG_NOOP        0x1   /* Nothing.*/
#define   NLMSG_ERROR       0x2   /* Error*/
```

```
#define   NLMSG_DONE          0x3    /* End of a dump*/
#define   NLMSG_OVERRUN       0x4    /* Data lost*/
#define   NLMSG_MIN_TYPE      0x10   /*编号小于 16 的消息类型为控制消息预留*/
```

每种协议类型可以定义自己的消息类型，类型值应大于 16（0x10），小于 16 的消息类型是公用的控制消息类型。

●**nlmsg_flags**：标记成员，取值定义在/include/upai/linux/netlink.h 头文件，例如：

```
#define   NLM_F_REQUEST    1    /*请求消息，需要内核处理的消息必须设置*/
#define   NLM_F_MULTI      2    /*消息由多部分组成*/
#define   NLM_F_ACK        4    /*希望接收方进行应答*/
#define   NLM_F_ECHO       8    /*回应当前请求*/
#define   NLM_F_DUMP_INTR       16   /* Dump was inconsistent due to sequence change */

/* Modifiers to GET request */
#define   NLM_F_ROOT     0x100   /*指定树根*/
#define   NLM_F_MATCH    0x200   /*返回所有匹配的表项*/
#define   NLM_F_ATOMIC   0x400   /* atomic GET*/
#define   NLM_F_DUMP     (NLM_F_ROOT|NLM_F_MATCH)

/* Modifiers to NEW request */
#define   NLM_F_REPLACE     0x100    /*覆盖现有表项*/
#define   NLM_F_EXCL        0x200    /*保留现有表项不动*/
#define   NLM_F_CREATE      0x400    /*创建表项（如果不存在）*/
#define   NLM_F_APPEND      0x800    /*在列表末尾添加表项*/
```

在 netlink 消息报头中有两个重要的信息没有指明，一是套接字的协议类型，二是目的套接字的 id 值。那么在发送消息时如何寻找到正确的套接字呢？

在创建套接字时，将为套接字创建 netlink_sock 实例，netlink_sock->sk->**sk_protocol** 成员保存了协议类型。用户进程通过 connect()系统调用可以将目的套接字 id 值设置到 netlink_sock->sk->**dst_portid** 成员，用户向内核发送消息时，目的套接字 id 值设为 0。

发送消息时，由于只能向同一协议类型的套接字发送消息，由本套接字 netlink_sock->sk->sk_protocol 和 netlink_sock->sk->dst_portid 成员值，即可在 netlink_table 实例数组中查找到目的套接字 netlink_sock 实例。内核套接字发送应答消息时，可从接收到的 sk_buff 实例中获取发送套接字的 netlink_sock 实例，从其 **portid** 成员获取用户套接字的 id 值（创建套接字分配 id）。因此在 netlink 消息报头中不需要指明协议类型和目的套接字 id 值。

内核在/include/net/netlink.h 头文件中定义了 netlink 消息操作的接口函数，例如：

●static inline struct sk_buff ***nlmsg_new**(size_t payload, gfp_t flags)：创建 sk_buff 实例，分配内存缓存区，缓存区大小为 payload（数据区长度）加上消息报头的长度（要求对齐）。

●static inline struct nlmsghdr ***nlmsg_put**(struct sk_buff *skb, u32 portid, u32 seq,int type, int payload, int flags)：在缓存区末尾添加新消息（添加设置消息报头，为数据区分配空间并清零等），portid 源套接字 id，seq 序列号，type 消息类型，paylaod 数据区长度，flags 消息标记（没有向数据区写入数据）。

●static inline struct nlmsghdr ***nlmsg_next**(const struct nlmsghdr *nlh, int *remaining)：获取内存缓存区中下一个消息报头。

●static inline void ***nlmsg_data**(const struct nlmsghdr *nlh)：返回消息净荷（数据）区指针。

■属性

netlink 消息的数据区（净荷）通常包含协议类型报头和属性区域，协议类型报头由特定的协议类型定义，这里暂不介绍。属性区域由一个或多个属性组成，属性又包含属性头和属性值。

通用的属性头由 nlattr 结构体表示，定义如下（/include/uapi/linux/netlink.h）：

```
struct nlattr {
    __u16       nla_len;    /*属性长度，含属性头和属性值的长度（不包含属性值的对齐字节）*/
    __u16       nla_type;   /*属性类型，bit15 表示是否包含嵌套属性，bit14 表示是否是网络字节序*/
};
```

nlattr 结构体成员简介如下：

●**nla_len**：属性长度（字节数），含属性头和属性值，不含属性值后面的对齐字节。

●**nla_type**：属性类型，bit15 表示是否嵌套属性，bit14 表示属性值是否是按网络字节序（大端）保存。这里的属性类型类似于 C 语言的数据类型，包括字节、字符串等。

内核定义了标准的属性类型（/include/net/netlink.h）：

```
enum {
    NLA_UNSPEC,
    NLA_U8,
    NLA_U16,
    NLA_U32,
    NLA_U64,
    NLA_STRING,
    NLA_FLAG,
    NLA_MSECS,
    NLA_NESTED,
    NLA_NESTED_COMPAT,
    NLA_NUL_STRING,
    NLA_BINARY,
    NLA_S8,
    NLA_S16,
    NLA_S32,
    NLA_S64,
    __NLA_TYPE_MAX,
};
```

内核在/include/net/netlink.h 头文件中定义了向属性区域写入/提取属性的接口函数，例如：

●static inline struct nlattr ***nla_next**(const struct nlattr *nla, int *remaining)：获取下一属性头指针。

●static inline int **nla_put_u8**(struct sk_buff *skb, int attrtype, u8 value)：写入字节属性。

●static inline int **nla_put_string**(struct sk_buff *skb, int attrtype,const char *str)：写入字符串属性。

●static inline u8 **nla_get_u8**(const struct nlattr *nla)：读取字节属性值。

●static inline int **nlmsg_parse**(const struct nlmsghdr *nlh, int hdrlen,struct nlattr *tb[], int maxtype,const struct nla_policy *policy)：解析消息中的属性，tb[]数组项指向各属性头，maxtype 表示期望的最大属性类型值，policy 指向 nla_policy 实例用于检查属性区传递属性的有效性，函数定义见下文。

在以上填充属性的函数中将最终调用/lib/nlattr.c 文件内的 nla_put()函数将属性写入消息末尾。

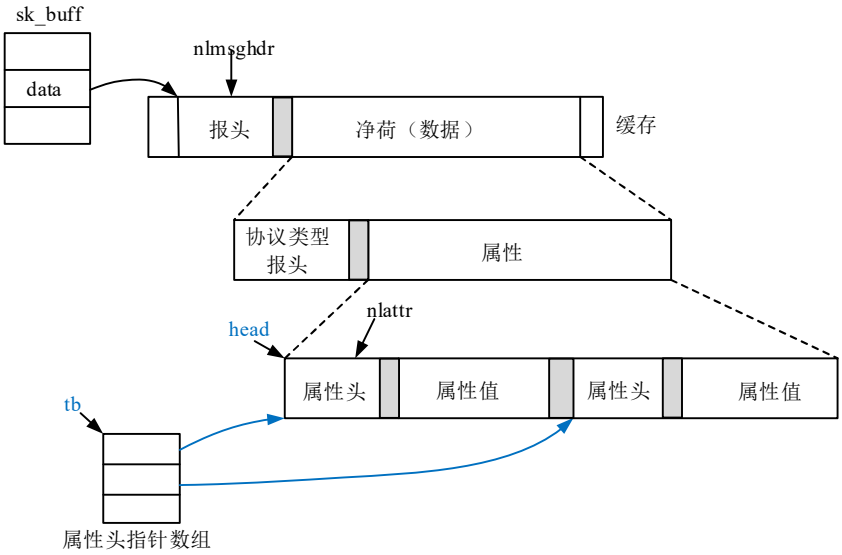
■**解析属性**

内核套接字（包括用户套接字）在收到 netlink 消息时，需要对属性进行解析，以便对属性进行正确的使用。netlink 套接字属性的解析、属性有效性判断等操作函数定义在/lib/nlattr.c 文件内。

nla_parse()函数用于解析属性，函数定义如下：

```
int nla_parse(struct nlattr **tb, int maxtype, const struct nlattr *head, int len, const struct nla_policy *policy)
/*
*tb：指向属性头指针数组，maxtype：属性类型最大值，head：属性区开始位置，len：属性区长度，
*policy：属性策略，用于属性有效性检查。
*/
{
    const struct nlattr *nla;
    int rem, err;

    memset(tb, 0, sizeof(struct nlattr *) * (maxtype + 1));   /*属性头指针数组清零*/

    nla_for_each_attr(nla, head, len, rem) {     /*遍历消息中的属性*/
        u16 type = nla_type(nla);       /*属性类型*/

        if (type > 0 && type <= maxtype) {
            if (policy) {
                err = validate_nla(nla, maxtype, policy);     /*属性有效性检查*/
                ...
            }
            tb[type] = (struct nlattr *)nla;   /*属性头指针数组项指向属性头实例*/
        }
    }
    ...
    err = 0;
errout:
    return err;
}
```

nla_parse()函数对属性进行解析的结果如下图所示（数据区含两个属性）：



内核在/include/net/netlink.h 头文件定义了 **nla_parse()**函数的包装函数 **nlmsg_parse()**，定义如下：

```
static inline int nlmsg_parse(const struct nlmsghdr *nlh, int hdrlen,struct nlattr *tb[], int maxtype,
                              const struct nla_policy *policy)
/*nlh：指向 netlink 消息报头，hdrlen：表示协议类型报头的长度*/
{
    if (nlh->nlmsg_len < nlmsg_msg_size(hdrlen))
        return -EINVAL;

    return nla_parse(tb, maxtype, nlmsg_attrdata(nlh, hdrlen),nlmsg_attrlen(nlh, hdrlen), policy);
}
```

nlmsg_parse()调用 nla_parse()函数时，第三个参数表示属性开始位置，需要跳过协议类型报头（长度为 hdrlen），第四个参数表示属性长度时，需要减去协议类型报头的长度。

12.3.2 创建套接字

内核通过接口函数 netlink_kernel_create()创建 netlink 套接字，用户进程通过 socket()系统调用创建套接字。内核套接字在启动阶段或加载模块时创建，它要先于用户套接字创建。

1 内核创建套接字

在介绍内核创建 netlink 套接字前，先看一下 netlink_kernel_cfg 结构体的定义，它在创建内核套接字时用于传递可选的信息。netlink_kernel_cfg 实例主要用于设置协议类型对应的 netlink_table 实例和表示内核套接字的 netlink_sock 实例。

netlink_kernel_cfg 结构体定义如下（/include/linux/netlink.h）：

```
struct netlink_kernel_cfg {
    unsigned int    groups;   /*协议类型支持的组播组最大数量（编号）*/
    unsigned int    flags;    /*标记，赋予 netlink_table 实例*/
    void        (*input)(struct sk_buff *skb);    /*接收用户消息函数，赋予内核 netlink_sock 实例*/
    struct mutex    *cb_mutex;
    int         (*bind)(struct net *net, int group);      /*赋予 netlink_table 实例，绑定组操作*/
    void        (*unbind)(struct net *net, int group);    /*赋予 netlink_table 实例，解绑组操作*/
    bool        (*compare)(struct net *net, struct sock *sk);    /*赋予 netlink_table 实例*/
};
```

netlink_kernel_cfg 结构体主要成员简介如下：

●**groups**：协议类型支持的组播组最大数量（最大编号值）。

●**flags**：标记，赋予协议类型 netlink_table.flags 成员，用于标记是否允许非超级用户发送和接收组播消息，取值如下：

```
#define   NL_CFG_F_NONROOT_RECV  (1 << 0)       /*bit0*/
#define   NL_CFG_F_NONROOT_SEND  (1 << 1)       /*bit1*/
```

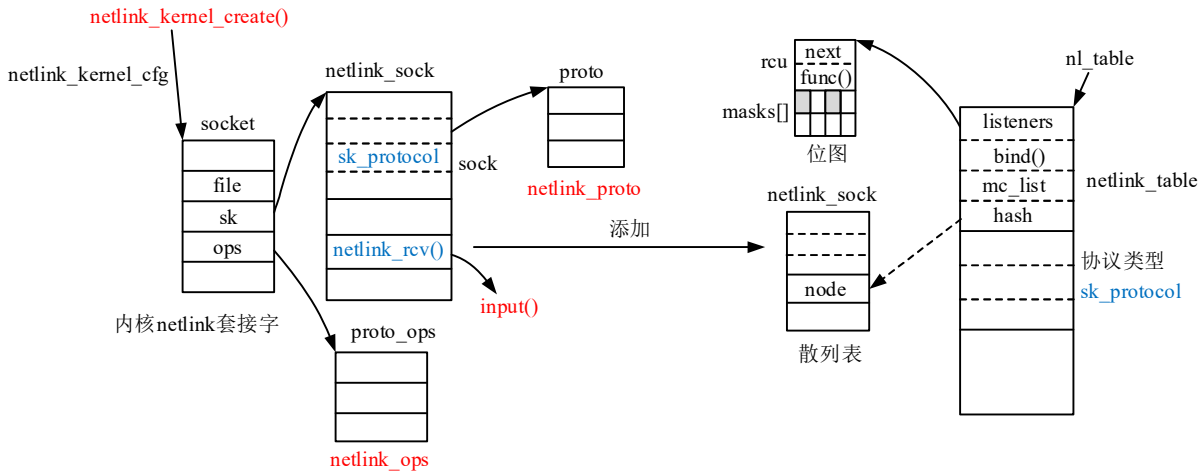
设置 NL_CFG_F_NONROOT_RECV 标记位表示非超级用户可以加入到组播组，接收组播消息。

设置 NL_CFG_F_NONROOT_SEND 标记位表示非超级用户可以发送组播消息。

●**input**：赋予内核套接字 netlink_sock 实例的 netlink_rcv()成员，用户套接字向内核套接字发送消息时，将调用此函数接收并处理消息，需要时发送应答消息。

●**bind()、unbind()、compare()**：函数指针成员，赋予协议类型对应的 netlink_table 实例中的对应成员。

内核创建 netlink 套接字的接口函数为 **netlink_kernel_create()**，函数执行结果如下图所示：



netlink_kernel_create()函数定义在/include/linux/netlink.h 头文件：

```
static inline struct sock *netlink_kernel_create(struct net *net, int unit, struct netlink_kernel_cfg *cfg)
/*net：网络命名空间，unit：协议类型，cfg：指向 netlink_kernel_cfg 实例*/
{
    return __netlink_kernel_create(net, unit, THIS_MODULE, cfg);
}
```

__netlink_kernel_create()函数定义在/net/netlink/af_netlink.c 文件内：

```
struct sock *__netlink_kernel_create(struct net *net, int unit, struct module *module,
                                     struct netlink_kernel_cfg *cfg)
{
    struct socket *sock;
    struct sock *sk;
    struct netlink_sock *nlk;
    struct listeners *listeners = NULL;   /*监听组位图，/net/netlink/af_netlink.c*/
    struct mutex *cb_mutex = cfg ? cfg->cb_mutex : NULL;
    unsigned int groups;

    BUG_ON(!nl_table);

    if (unit < 0 || unit >= MAX_LINKS)
```

```
        return NULL;

if (sock_create_lite(PF_NETLINK, SOCK_DGRAM, unit, &sock))
                                        /*只分配 socket 实例，套接字为数据报类型*/
        return NULL;

if (__netlink_create(net, sock, cb_mutex, unit, 1) < 0)     /*创建 netlink_sock 实例等，见下文*/
        goto out_sock_release_nosk;

sk = sock->sk;   /*netlink_sock.sk（sock 实例）*/

if (!cfg || cfg->groups < 32)
        groups = 32;                /*如果 groups 小于 32，则设置为 32*/
else                                /*groups 大于等于 32 的情形*/
        groups = cfg->groups;     /*groups 值不变*/

listeners = kzalloc(sizeof(*listeners) + NLGRPSZ(groups), GFP_KERNEL);
                        /*分配 listeners 实例，后接位图（位图 4 字节对齐），位图清零*/
if (!listeners)
        goto out_sock_release;

sk->sk_data_ready = netlink_data_ready;     /*netlink_data_ready()函数为空操作*/
if (cfg && cfg->input)
        nlk_sk(sk)->netlink_rcv = cfg->input;     /*设置 netlink_sock 实例接收消息回调函数*/

if (netlink_insert(sk, 0))     /*netlink_sock 实例插入 netlink_table 实例散列链表，注意其 id 值为 0*/
        goto out_sock_release;

nlk = nlk_sk(sk);       /*返回 netlink_sock 实例指针*/
nlk->flags |= NETLINK_F_KERNEL_SOCKET;     /*标记为内核 netlink 套接字*/

netlink_table_grab();
if (!nl_table[unit].registered) {   /*如果对应 netlink_table 实例未注册，则对其进行设置*/
        nl_table[unit].groups = groups;       /*支持的组播组数量*/
        rcu_assign_pointer(nl_table[unit].listeners, listeners);   /*指向 listeners 实例*/
        nl_table[unit].cb_mutex = cb_mutex;
        nl_table[unit].module = module;
        if (cfg) {                  /*由 netlink_kernel_cfg 实例设置 netlink_table 实例，设置回调函数等*/
                nl_table[unit].bind = cfg->bind;
                nl_table[unit].unbind = cfg->unbind;
                nl_table[unit].flags = cfg->flags;
                if (cfg->compare)
                        nl_table[unit].compare = cfg->compare;
        }
        nl_table[unit].registered = 1;       /*标记 netlink_table 实例已注册*/
} else {
```

```
            kfree(listeners);
            nl_table[unit].registered++;
        }
        netlink_table_ungrab();
        return sk;
        ...
    }
```

__netlink_kernel_create()函数内调用 sock_create_lite()函数只创建 socket 实例，再调用__netlink_create()函数创建 netlink_sock 实例，利用 cfg->input()设置 netlink_sock->netlink_rcv()成员，将 netlink_sock 实例（id 值设为 0）添加到 netlink_table 实例散列表。为 netlink_table 实例 listeners 成员分配空间，用于标记所有被监听的组，利用 netlink_kernel_cfg 实例设置 netlink_table 实例（如果实例尚未注册），最后标记 netlink_table 实例已经注册。

注意，netlink_kernel_cfg 实例中指定的最大组播组数量可以超过 32，内核最小将其设置为 32，表示套接字地址的 sockaddr_nl 结构体中最大只能支持 32 个组播组。如果用户套接字要监听编号大于 32 的组播组，需要通过 setsockopt()系统调用（NETLINK_ADD_MEMBERSHIP 参数）设置（扩展）netlink_sock 实例中的监听组位图。

下面看一下__netlink_kernel_create()函数中调用的**__netlink_create()**函数，它用于创建 netlink_sock 实例等，下文介绍的用户创建 netlink 套接字的操作中也将调用此函数，函数代码如下：

```
static int __netlink_create(struct net *net, struct socket *sock,
                            struct mutex *cb_mutex, int protocol,int kern)
/*protocol：协议类型，kern：是否是内核套接字，此处为 1*/
{
    struct sock *sk;
    struct netlink_sock *nlk;

    sock->ops = &netlink_ops;      /*socket 关联 proto_ops 实例*/

    sk = sk_alloc(net, PF_NETLINK, GFP_KERNEL, &netlink_proto, kern); /*创建 netlink_sock 实例*/
    ...
    sock_init_data(sock, sk);      /*初始化 sock 实例，关联 socket 实例等，/net/core/sock.c*/

    nlk = nlk_sk(sk);      /*netlink_sock 实例指针*/
    ...
    init_waitqueue_head(&nlk->wait);      /*初始化等待队列头*/
#ifdef CONFIG_NETLINK_MMAP
    mutex_init(&nlk->pg_vec_lock);
#endif

    sk->sk_destruct = netlink_sock_destruct;    /*析构函数*/
    sk->sk_protocol = protocol;        /*设置 netlink 套接字协议类型*/
    return 0;
}
```

__netlink_create()函数中有一项重要的工作就是对 socket.ops 成员赋值，netlink 套接字 proto_ops 实例为 **netlink_ops**，此实例后面将做介绍。netlink 套接字关联的 proto 结构体实例为 **netlink_proto**，在创建套接字时，将从其中的 slab 缓存中分配 netlink_sock 实例（内嵌 sock 实例）。

2 用户创建套接字

用户进程通过 socket()系统调用创建 netlink 套接字，例如：

socket(**AF_NETLINK**,SOCK_RAM|SOCK_CLOEXEC,**NETLINK_ROUTE**);

socket()系统调用中地址簇参数设为 AF_NETLINK，协议类型为 netlink 套接字协议类型。

前面介绍的 socket()系统调用中，在创建 socket 实例后，将调用地址簇关联的 net_proto_family 实例中的 create()函数设置 socket 实例并创建关联的 sock 实例等。

在前面介绍的 netlink 套接字初始化函数中注册了 netlink 地址簇对应的 net_proto_family 实例，实例定义如下（/net/netlink/af_netlink.c）：

```
static const struct net_proto_family netlink_family_ops = {
        .family = PF_NETLINK,
        .create = netlink_create,          /*设置用户 netlink 套接字*/
        .owner     = THIS_MODULE,
};
```

netlink_create()函数用于设置用户套接字 socket 实例，函数内执行的操作与创建内核 netlink 套接字的接口函数类似，代码如下：

```
static int netlink_create(struct net *net, struct socket *sock, int protocol,int kern)
{
        struct module *module = NULL;
        struct mutex *cb_mutex;
        struct netlink_sock *nlk;
        int (*bind)(struct net *net, int group);
        void (*unbind)(struct net *net, int group);
        int err = 0;

        sock->state = SS_UNCONNECTED;

        if (sock->type != SOCK_RAW && sock->type != SOCK_DGRAM)     /*只能是原始或数据报类型*/
                return -ESOCKTNOSUPPORT;

        if (protocol < 0 || protocol >= MAX_LINKS)
                return -EPROTONOSUPPORT;

        netlink_lock_table();
#ifdef CONFIG_MODULES
        if (!nl_table[protocol].registered) {     /*加载模块，注册 netlink_table 实例（如果未注册）*/
                netlink_unlock_table();
                request_module("net-pf-%d-proto-%d", PF_NETLINK, protocol);
                netlink_lock_table();
        }
#endif
        if (nl_table[protocol].registered &&try_module_get(nl_table[protocol].module))
                module = nl_table[protocol].module;
        else
```

```
        err = -EPROTONOSUPPORT;

    cb_mutex = nl_table[protocol].cb_mutex;
    bind = nl_table[protocol].bind; /*获取 netlink_table 实例中回调函数用于设置 netlink_sock 实例*/
    unbind = nl_table[protocol].unbind;
    netlink_unlock_table();
    ...
    err = __netlink_create(net, sock, cb_mutex, protocol, kern);     /*创建 netlink_sock 实例*/
    ...
    local_bh_disable();
    sock_prot_inuse_add(net, &netlink_proto, 1);     /*增加使用计数，/net/core/sock.c*/
    local_bh_enable();

    nlk = nlk_sk(sock->sk);     /*netlink_sock 实例*/
    nlk->module = module;
    nlk->netlink_bind = bind;       /*回调函数来自于 netlink_table 实例*/
    nlk->netlink_unbind = unbind;
out:
    return err;
    ...
}
```

用户创建 netlink 套接字时，内核为其创建并初始化了 netlink_sock 实例，但没有为 netlink_sock 实例设置编号 id 值，也没有将实例插入到 netlink_sock 实例散列表，这些操作将在后面介绍的 bind()（绑定）系统调用中执行。

12.3.3 套接字操作

本小节从用户进程的角度介绍 netlink 套接字的操作，主要是系统调用的实现。

在创建 netlink 套接字调用的__netlink_create()函数中，socket 实例关联的 proto_ops 实例为 netlink_ops，实例中函数负责实现的套接字操作接口。

netlink_ops 实例定义如下：

```
static const struct proto_ops netlink_ops = {
    .family = PF_NETLINK,           /*协议簇标识*/
    .owner =  THIS_MODULE,
    .release = netlink_release,
    .bind =    netlink_bind,      /*绑定操作*/
    .connect =      netlink_connect,     /*连接操作，设置目的套接字地址和组播组，用于发送消息*/
    .socketpair =   sock_no_socketpair,
    .accept =  sock_no_accept,
    .getname =netlink_getname,
    .poll =     netlink_poll,
    .ioctl =    sock_no_ioctl,
    .listen =   sock_no_listen,
    .shutdown =    sock_no_shutdown,
    .setsockopt =   netlink_setsockopt,     /*设置参数*/
    .getsockopt =   netlink_getsockopt,     /*获取参数*/
```

```
    .sendmsg =      netlink_sendmsg,     /*发送消息*/
    .recvmsg =      netlink_recvmsg,     /*接收消息*/
    .mmap =         netlink_mmap,
    .sendpage =     sock_no_sendpage,
};
```

下面简要介绍一下 netlink_ops 实例中绑定操作、连接操作、设置参数、发送消息和接收消息等函数的实现。

1 绑定操作

前面介绍过 netlink 套接字地址由 sockaddr_nl 结构体表示，定义如下（/include/upai/linux/netlink.h）：

```
struct sockaddr_nl {
    __kernel_sa_family_t    nl_family;      /*始终为 AF_NETLINK*/
    unsigned short      nl_pad;         /*始终为 0*/
    __u32           nl_pid;         /*内核套接字为 0，用户套接字可为进程 pid（是唯一值就行）*/
    __u32           nl_groups;      /*监听的组播组位图*/
};
```

绑定操作用于将一个地址绑定到当前套接字，在绑定操作中 sockaddr_nl.nl_groups 成员是一个位图，表示套接字监听的组播组，每个比特位代表一个组，比特位位置加 1 表示组播组编号。netlink 套接字通过 bind()系统调用最多可监听 32 个组播组，且进程需要超级用户权限或者 netlink_table 实例中 flags 成员设置了标记位 NL_CFG_F_NONROOT_RECV。如果需要监听更多的组播组，需要使用 setsockopt()系统调用。

bind()系统调用内将调用 proto_ops 实例中的 bind()函数，执行绑定操作。netlink_ops 实例中绑定操作函数为 netlink_bind()，定义如下：

```
static int netlink_bind(struct socket *sock, struct sockaddr *addr,int addr_len)
/*addr：指向地址值模板 sockaddr 实例，addr_len：地址实例长度*/
{
    struct sock *sk = sock->sk;
    struct net *net = sock_net(sk);
    struct netlink_sock *nlk = nlk_sk(sk);      /*指向 netlink_sock 实例*/
    struct sockaddr_nl *nladdr = (struct sockaddr_nl *)addr; /*sockaddr 指针转换成 sockaddr_nl 指针*/
    int err;
    long unsigned int groups = nladdr->nl_groups;   /*监听的组播组位图*/
    bool bound;
    ...
    /*超级用户或设置了 NL_CFG_F_NONROOT_RECV 标记位，才可监听组播消息*/
    if (groups) {
        if (!netlink_allowed(sock, NL_CFG_F_NONROOT_RECV))
            return -EPERM;
        err = netlink_realloc_groups(sk);      /*为 netlink_sock->groups 分配位图（4 字节对齐）*/
                    /*设置 netlink_sock->ngroups = groups，位图大小由 netlink_table.groups 确定*/
        ...
    }

    bound = nlk->bound;      /*netlink_sock 是否已绑定地址*/
    if (bound) {
```

```
            smp_rmb();
            if (nladdr->nl_pid != nlk->portid)
                return -EINVAL;
        }

        if (nlk->netlink_bind && groups) {     /*绑定监听的组播组*/
            int group;

            for (group = 0; group < nlk->ngroups; group++) {
                if (!test_bit(group, &groups))     /*检测监听的组，groups 为位图*/
                    continue;
                err = nlk->netlink_bind(net, group + 1);     /*绑定组，例如在全局位图中设置标记位*/
                                /*调用 netlink_sock 实例中 netlink_bind()函数，group + 1 为组编号*/
                if (!err)
                    continue;
                netlink_undo_bind(group, groups, sk);
                return err;
            }
        }

        if (!bound) {   /*自动设置 netlink_sock 实例 id 值，并将实例插入管理散列表*/
            err = nladdr->nl_pid ?netlink_insert(sk, nladdr->nl_pid) :netlink_autobind(sk);
            ...
        }

        if (!groups && (nlk->groups == NULL || !(u32)nlk->groups[0]))
            return 0;

        netlink_table_grab();
        netlink_update_subscriptions(sk, nlk->subscriptions +hweight32(groups) -hweight32(nlk->groups[0]));
                /*将 sock 实例（netlink_sock.sk.sk_bind_node）添加到 netlink_table 实例 mc_list 链表*/
        nlk->groups[0] = (nlk->groups[0] & ~0xffffffffUL) | groups;     /*设置监听组位图*/
        netlink_update_listeners(sk);       /*更新 listeners->masks[]成员，设置被监听组标记位*/
        netlink_table_ungrab();

        return 0;
    }
```

绑定操作中通过 sockaddr_nl 结构体实例传递套接字需要绑定的 id 值和监听组位图。如果设置了监听组，则需要为 netlink_sock 实例分配位图并设置监听组的标记位。另外，还需要通知协议类型某个组已被用户套接字监听，需设置 listeners->masks[]全局位图中的标记位等。

如果地址 sockaddr_nl 实例中指定了端口 id 值（nl_pid），则调用 netlink_insert()函数将 nl_pid 值赋予 nlk_sk(sk)->**portid** 成员，并将 netlink_sock 实例插入管理散列表（netlink_table.hash 成员内的散列表）。

如果 nl_pid 值为 0，则调用 **netlink_autobind(sock)**函数自动为套接字分配一个 id 值（通常由进程 pid 生成），然后再调用 netlink_insert()函数将 netlink_sock 实例插入管理散列表。

如果套接字监听了某个组播组消息，netlink_sock 实例还将添加到对应 netlink_table 实例的 **mc_list** 链表（netlink_sock.sk）。其它套接字发送组播消息时，将扫描此链表下的 netlink_sock 实例，如果套接字监

听了本组播组，则复制一份消息副本发送给此套接字。

2 连接操作

连接操作用于建立本套接字与通信对方套接字之间的关联，即设置目的套接字的 id 值，以实现数据的传输。用户套接字通过系统调用 connect()建立与对方套接字的关联，系统调用内将调用 proto_ops->connect()函数，对于 netlink 套接字此函数为 **netlink_connect()**，定义如下：

```
static int netlink_connect(struct socket *sock, struct sockaddr *addr,int alen, int flags)
/*sock：本套接字 socket 实例指针，addr：目的套接字地址，即 sockaddr_nl 实例，flags：标记*/
{
    int err = 0;
    struct sock *sk = sock->sk;
    struct netlink_sock *nlk = nlk_sk(sk);
    struct sockaddr_nl *nladdr = (struct sockaddr_nl *)addr;   /*转为 sockaddr_nl 实例指针*/
    /*sockaddr_nl 实例中 nl_pid 表示目的套接字编号，nl_groups 表示消息发送到的组播组（位图）*/

    ...  /*地址有效性检查*/

    if ((nladdr->nl_groups || nladdr->nl_pid) &&!netlink_allowed(sock, NL_CFG_F_NONROOT_SEND))
        return -EPERM;          /*如果进程没有发送组播消息权限，返回错误码*/

    if (!nlk->bound)    /*本套接字尚未绑定地址，则自动绑定*/
        err = netlink_autobind(sock);

    if (err == 0) {
        sk->sk_state   = NETLINK_CONNECTED;      /*设置已连接状态*/
        nlk->dst_portid = nladdr->nl_pid;            /*目的套接字编号（端口 ID）*/
        nlk->dst_group   = ffs(nladdr->nl_groups);      /*组播位图中第一个置位的位*/
    }

    return err;
}
```

连接操作主要是将套接字状态设置为 NETLINK_CONNECTED，将目的套接字编号 id 值赋予本套接字 netlink_sock->dst_portid 成员，将组播组位图中第一个置位的位（位置）赋予 netlink_sock->dst_group 成员。在发送组播消息时，只能向第一个置位的组播组发送组播消息。

3 设置/获取参数

netlink 套接字操作结构 proto_ops 实例 netlink_ops 中设置参数的函数为 **netlink_setsockopt()**，获取参数的函数为 **netlink_getsockopt()**。下面以设置参数的 netlink_setsockopt()函数为例介绍其实现。

netlink 套接字参数名称（整型数）定义在/include/uapi/linux/netlink.h 头文件：

```
#define   NETLINK_ADD_MEMBERSHIP          1    /*添加到组播组*/
#define   NETLINK_DROP_MEMBERSHIP         2
#define   NETLINK_PKTINFO                 3
#define   NETLINK_BROADCAST_ERROR         4
#define   NETLINK_NO_ENOBUFS              5
```

```
#define   NETLINK_RX_RING                6
#define   NETLINK_TX_RING                7
#define   NETLINK_LISTEN_ALL_NSID        8
#define   NETLINK_LIST_MEMBERSHIPS       9
```

netlink_setsockopt()函数定义如下：

```
static int netlink_setsockopt(struct socket *sock, int level, int optname,
                                                    char __user *optval, unsigned int optlen)
/*level：参数级别，必须为 SOL_NETLINK，表示 netlink 套接字，
*optname：参数名称，optval：指向用户内存区，保存参数值，optlen：表示参数长度。
*/
{
    struct sock *sk = sock->sk;
    struct netlink_sock *nlk = nlk_sk(sk);
    unsigned int val = 0;
    int err;

    if (level != SOL_NETLINK)      /*非 SOL_NETLINK 参数级别，返回错误码*/
        return -ENOPROTOOPT;

    if (optname != NETLINK_RX_RING && optname != NETLINK_TX_RING &&
        optlen >= sizeof(int) &&get_user(val, (unsigned int __user *)optval))
        return -EFAULT;          /*有效性检查*/

    switch (optname) {       /*参数名称*/
    case NETLINK_PKTINFO:
        if (val)          /*设置标记位*/
            nlk->flags |= NETLINK_F_RECV_PKTINFO;
        else
            nlk->flags &= ~NETLINK_F_RECV_PKTINFO;
        err = 0;
        break;
    case NETLINK_ADD_MEMBERSHIP:     /*加入组播组*/
    case NETLINK_DROP_MEMBERSHIP: {
        if (!netlink_allowed(sock, NL_CFG_F_NONROOT_RECV))
            return -EPERM;
        err = netlink_realloc_groups(sk);     /*扩展组播组位图*/
        if (err)
            return err;
        if (!val || val - 1 >= nlk->ngroups)
            return -EINVAL;
        if (optname == NETLINK_ADD_MEMBERSHIP && nlk->netlink_bind) {
            err = nlk->netlink_bind(sock_net(sk), val);     /*绑定组操作*/
            if (err)
                return err;
        }
```

```
            netlink_table_grab();
            netlink_update_socket_mc(nlk, val,optname == NETLINK_ADD_MEMBERSHIP);
            netlink_table_ungrab();
            if (optname == NETLINK_DROP_MEMBERSHIP && nlk->netlink_unbind)
                nlk->netlink_unbind(sock_net(sk), val);

            err = 0;
            break;
        }
        case NETLINK_BROADCAST_ERROR:
            if (val)
                nlk->flags |= NETLINK_F_BROADCAST_SEND_ERROR;
            else
                nlk->flags &= ~NETLINK_F_BROADCAST_SEND_ERROR;
            err = 0;
            break;
        case NETLINK_NO_ENOBUFS:
            if (val) {
                nlk->flags |= NETLINK_F_RECV_NO_ENOBUFS;
                clear_bit(NETLINK_S_CONGESTED, &nlk->state);
                wake_up_interruptible(&nlk->wait);
            } else {
                nlk->flags &= ~NETLINK_F_RECV_NO_ENOBUFS;
            }
            err = 0;
            break;
    #ifdef CONFIG_NETLINK_MMAP
        ...
    #endif   /* CONFIG_NETLINK_MMAP */
        case NETLINK_LISTEN_ALL_NSID:
            if (!ns_capable(sock_net(sk)->user_ns, CAP_NET_BROADCAST))
                return -EPERM;

            if (val)
                nlk->flags |= NETLINK_F_LISTEN_ALL_NSID;
            else
                nlk->flags &= ~NETLINK_F_LISTEN_ALL_NSID;
            err = 0;
            break;
        default:
            err = -ENOPROTOOPT;
        }
        return err;
}
```

netlink_setsockopt()函数根据不同的参数名称，进行不同的处理，源码请读者自行阅读。

netlink_ops 中实例中 netlink 套接字获取参数的函数为 **netlink_getsockopt()**，源代码请读者自行阅读。

4 单播与组播消息

在介绍 netlink 套接字发送和接收消息前，先介绍一下套接字单播消息和发送组播消息的接口函数。

■单播消息

netlink 套接字发送单播消息的接口函数为 **netlink_unicast()**，函数定义如下：

```
int netlink_unicast(struct sock *ssk, struct sk_buff *skb,u32 portid, int nonblock)
/*ssk：源套接字，skb：发送的 sk_buff 实例，portid：目的套接字 id 值， nonblock：是否非阻塞*/
{
    struct sock *sk;
    int err;
    long timeo;

    skb = netlink_trim(skb, gfp_any()); /*对 sk_buff 实例进行处理，如果共享实例则克隆一个副本等*/

    timeo = sock_sndtimeo(ssk, nonblock);  /*返回 0 或 sk->sk_sndtimeo（非阻塞时为 0）*/
retry:
    sk = netlink_getsockbyportid(ssk, portid);
                    /*由端口 id 值在管理散列表中查找目的套接字，ssk 中指定了协议类型*/
    ...
    if (netlink_is_kernel(sk))      /*目的套接字是内核套接字*/
        return netlink_unicast_kernel(sk, skb, ssk);
                                        /*调用 netlink_sock->netlink_rcv(skb)函数接收消息*/

    if (sk_filter(sk, skb)) {    /*数据包过滤，调用 sock->sk_filter 中函数*/
        ...
        return err;
    }

    err = netlink_attachskb(sk, skb, &timeo, ssk);  /*判断是否可将数据包发送到目的套接字*/
          /*返回 0 表示可以发送，1 表示重试，<0 表示出错（失败）sk_buff 被释放*/
    if (err == 1)
        goto retry;
    if (err)
        return err;
    return netlink_sendskb(sk, skb);  /*将 sk_buff 实例添加到目的套接字接收缓存队列末尾*/
}
```

单播消息函数根据目的套接字 id 值和源套接字中指定的协议类型，在管理散列表中查找其 netlink_sock 实例。如果目的套接字是内核套接字则调用其中的 **netlink_rcv(skb)**函数接收消息。如果目的套接字是用户套接字则先对数据包进行过滤操作，再将 sk_buff 实例添加到目的套接字接收数据包缓存队列末尾，用户进程可从此队列读取数据。

■组播消息

netlink 套接字发送组播消息的接口函数为 **netlink_broadcast()**，函数定义如下：

```
int netlink_broadcast(struct sock *ssk, struct sk_buff *skb, u32 portid,u32 group, gfp_t allocation)
```

```
/*
*ssk：源套接字，skb：发送数据包 sk_buff 实例，portid：不发送的套接字 id 值，
*group：组播组编号，allocation：内存分配标记。
*/
{
    return netlink_broadcast_filtered(ssk, skb, portid, group, allocation,NULL, NULL);
}
```

netlink_broadcast()函数用于将消息发送到监听了 group 组播组的套接字，注意 group 表示组播组编号，而不是位图，也就是说只能将消息发送到一个组里，并且会过滤掉 id 值为 portid 的套接字，即不将消息发送到 portid 表示的套接字（监听了 group 组播组也不发送）。

netlink_broadcast()函数内调用的 netlink_broadcast_filtered()函数定义如下：

```
int netlink_broadcast_filtered(struct sock *ssk, struct sk_buff *skb, u32 portid,u32 group, gfp_t allocation,
    int (*filter)(struct sock *dsk, struct sk_buff *skb, void *data),void *filter_data)
/*filter：数据包过滤函数，filter_data：数据包过滤函数参数，此处这两个参数为 NULL*/
{
    struct net *net = sock_net(ssk);
    struct netlink_broadcast_data info;     /*组播信息*/
    struct sock *sk;

    skb = netlink_trim(skb, allocation);

    info.exclude_sk = ssk;      /*netlink_broadcast_data 初始化*/
    info.net = net;
    info.portid = portid;
    info.group = group;
    info.failure = 0;
    info.delivery_failure = 0;
    info.congested = 0;
    info.delivered = 0;
    info.allocation = allocation;
    info.skb = skb;
    info.skb2 = NULL;
    info.tx_filter = filter;
    info.tx_data = filter_data;

    netlink_lock_table();

    /*扫描执行了监听组播消息的 netlink_sock 实例，netlink_table.mc_list 链表*/
    sk_for_each_bound(sk, &nl_table[ssk->sk_protocol].mc_list)
        do_one_broadcast(sk, &info);        /*复制 sk_buff 副本发送给监听 group 组播消息的套接字*/

    consume_skb(skb);     /*释放 sk_buff 实例*/

    netlink_unlock_table();

    if (info.delivery_failure) {
```

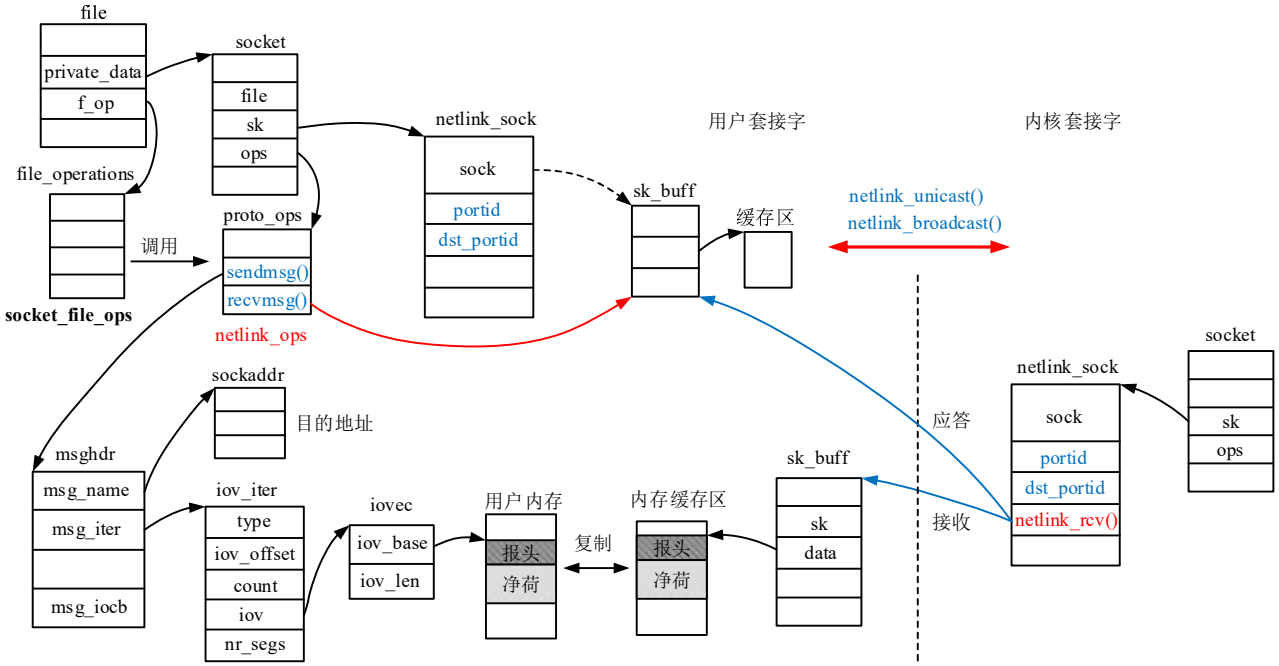
```
            kfree_skb(info.skb2);
            return -ENOBUFS;
        }
        consume_skb(info.skb2);

        if (info.delivered) {
            if (info.congested && (allocation & __GFP_WAIT))
                yield();
            return 0;
        }
        return -ESRCH;
    }
```

netlink_broadcast_filtered()函数定义了 netlink_broadcast_data 结构体实例用于暂存组播发送信息，函数内将扫描协议类型 netlink_table.mc_list 链表下的 netlink_sock 实例（sock 实例），调用 do_one_broadcast(sk, &info)函数对其发送组播消息。

do_one_broadcast(sk, &info)函数检查套接字组播监听位图，判断套接字是否监听了本组播组，并跳过 id 值为 portid 的套接字。对监听了本组播组消息的套接字复制一个 sk_buff 实例副本，添加到其接收数据包缓存队列，完成消息的发送。

5 消息传输

用户进程通过 netlink 套接字与内核套接字通信的流程如下图所示：



用户套接字需要按消息格式构建消息并存入用户内存，调用发送消息的 write()、sendto()等系统调用发送消息。系统调用中将调用 netlink_ops 实例中的 sendmsg()函数，在此函数内将创建 sk_buff 实例，并将用户内存中消息复制到 sk_buff 实例内存缓存区中，最后调用接口函数 **netlink_unicast()/netlink_broadcast()** 单播/组播消息。

在系统调用参数中（如 sendto()）需要指明目的套接字的地址和组播组，或事先（如执行 write()系统调用前）执行 connect()系统调用设置目的套接字地址和组播组。由于内核套接字端口 id 值固定为 0，因此与内核通信时需将目的套接字 id 值设为 0。

用户套接字向内核套接字单播消息时，将调用内核套接字 netlink_sock 实例中的 netlink_rcv()函数接收、处理消息，如果需要向用户套接字发送应答消息，将在此函数内调用 **netlink_unicast()/netlink_broadcast()** 接口函数发送应答消息。

用户套接字读取消息的系统调用将从套接字接收数据包缓存队列中 sk_buff 实例关联的内存缓存区中读取消息数据。

■用户发送消息

用户套接字发送消息的系统调用中将调用 proto_ops 实例中的 sendmsg()函数发送消息，对于 netlink 套接字，此函数为 netlink_sendmsg()，函数定义如下：

```
static int netlink_sendmsg(struct socket *sock, struct msghdr *msg, size_t len)
/*msg：指向 msghdr 实例（包含目的套接字地址和消息信息），len：发送数据长度*/
{
    struct sock *sk = sock->sk;     /*源 socket 关联的 sock 实例*/
    struct netlink_sock *nlk = nlk_sk(sk);      /*指向发送套接字 netlink_sock 实例*/
    DECLARE_SOCKADDR(struct sockaddr_nl *, addr, msg->msg_name); /*构建目的地址结构实例*/
    u32 dst_portid;
    u32 dst_group;
    struct sk_buff *skb;
    int err;
    struct scm_cookie scm;     /*用于消息控制，/include/net/scm.h*/
    u32 netlink_skb_flags = 0;

    if (msg->msg_flags&MSG_OOB)
        return -EOPNOTSUPP;

    err = scm_send(sock, msg, &scm, true);     /*/include/net/scm.h*/
    ...
    if (msg->msg_namelen) {     /*msg->msg_name 不为空，表示指明了目的套接字地址*/
        err = -EINVAL;
        if (addr->nl_family != AF_NETLINK)
            goto out;
        dst_portid = addr->nl_pid;     /*目的套接字端口 ID*/
        dst_group = ffs(addr->nl_groups);     /*第一个组播组编号*/
        err =    -EPERM;
        if ((dst_group || dst_portid) &&!netlink_allowed(sock, NL_CFG_F_NONROOT_SEND))
            goto out;                          /*发送消息权限检查*/
        netlink_skb_flags |= NETLINK_SKB_DST;
    } else {        /*没有指明目的地址，发送消息前需通过 connect()系统调用设置目的地址、组播组*/
        dst_portid = nlk->dst_portid;
        dst_group = nlk->dst_group;
    }

    if (!nlk->bound) {   /*如果发送消息的套接字没有绑定地址，自动绑定地址（自动分配 id 值）*/
        err = netlink_autobind(sock);
        ...
```

```
        } else {
            smp_rmb();
        }

        if (netlink_tx_is_mmaped(sk) &&iter_is_iovec(&msg->msg_iter) &&
            msg->msg_iter.nr_segs == 1 &&msg->msg_iter.iov->iov_base == NULL) {
            err = netlink_mmap_sendmsg(sk, msg, dst_portid, dst_group,&scm);
            goto out;
        }

        err = -EMSGSIZE;
        if (len > sk->sk_sndbuf - 32)
            goto out;
        err = -ENOBUFS;
        skb = netlink_alloc_large_skb(len, dst_group);     /*分配 sk_buff 实例，分配内存缓存区等*/
        ...
        /*设置 sk_buff 实例中控制块，由 netlink_skb_parms 结构体表示，/include/linux/netlink.h*/
        NETLINK_CB(skb).portid    = nlk->portid;       /*源套接字 id 值*/
        NETLINK_CB(skb).dst_group = dst_group;         /*第一个组播组编号*/
        NETLINK_CB(skb).creds     = scm.creds;
        NETLINK_CB(skb).flags     = netlink_skb_flags;   /*标记*/

        err = -EFAULT;
        if (memcpy_from_msg(skb_put(skb, len), msg, len)) {
                            /*复制数据到 sk_buff 关联内存缓存区，/include/linux/skbuff.h*/
            ...
        }
        ...

        if (dst_group) {     /*需要组播消息*/
            atomic_inc(&skb->users);
            netlink_broadcast(sk, skb, dst_portid, dst_group, GFP_KERNEL);   /*组播消息*/
        }
        err = netlink_unicast(sk, skb, dst_portid, msg->msg_flags&MSG_DONTWAIT); /*单播消息*/
    out:
        scm_destroy(&scm);
        return err;
    }
```

netlink_sendmsg()函数执行主要工作如下：

（1）从 msghdr 实例中获取目的套接字 id 值、组播组及发送数据等信息，若 msghdr 实例未指定地址则从 netlink_sock 实例中获取目的套接字 id 值和组播组（connect()系统调用中设置）。

（2）分配 sk_buff 实例及内存缓存区，将需要发送的消息复制到此缓存区。

（3）调用接口函数 netlink_unicast()和 netlink_broadcast()将消息发送出去。

■内核接收消息

内核 netlink 套接字 netlink_sock.netlink_rcv()函数用于接收用户消息，在创建内核套接字时设置，主要是对消息进行解析并处理，如果需要应答则向用户套接字发送应答消息。

netlink_rcv()函数内通常调用 **netlink_rcv_skb()**函数接收消息，函数定义如下（/net/netlink/af_netlink.c）：

```
int netlink_rcv_skb(struct sk_buff *skb, int (*cb)(struct sk_buff *,struct nlmsghdr *))
/*skb：指向接收 sk_buff 实例，cb()：处理消息的回调函数*/
{
    struct nlmsghdr *nlh;     /*指向 netlink 消息报头*/
    int err;

    while (skb->len >= nlmsg_total_size(0)) {     /*遍历内存缓存区中消息*/
        int msglen;

        nlh = nlmsg_hdr(skb);     /*指向消息报头*/
        ...
        /*用户消息需要内核处理时，必须设置 NLM_F_REQUEST 标记*/
        if (!(nlh->nlmsg_flags & NLM_F_REQUEST))
            goto ack;

        /*跳过控制消息*/
        if (nlh->nlmsg_type < NLMSG_MIN_TYPE)     /*消息类型*/
            goto ack;

        err = cb(skb, nlh);       /*调用接收消息回调函数*/
        ...
ack:    /*发送出错应答消息*/
        if (nlh->nlmsg_flags & NLM_F_ACK || err)
            netlink_ack(skb, nlh, err);       /*创建 sk_buff 实例，发送（出错）应答消息*/

skip:
        msglen = NLMSG_ALIGN(nlh->nlmsg_len);   /*消息长度*/
        if (msglen > skb->len)
            msglen = skb->len;
        skb_pull(skb, msglen);   /*指向下一个消息*/
    }    /*遍历内存缓存区中消息结束*/

    return 0;
}
```

netlink_rcv_skb()函数遍历 sk_buff 实例内存缓存区中的消息，对设置了 NLM_F_REQUEST 标记位的消息调用回调用函数 **cb()**进行处理。如果需要应答或处理过程中出错，将调用 netlink_ack()函数发送应答消息。

■用户读取消息

用户 netlink 套接字读取消息的系统调用将调用 proto_ops 实例中的 recvmsg()函数读取消息，对于 netlink

套接字，此函数为 **netlink_recvmsg**()，函数定义如下：

```
static int netlink_recvmsg(struct socket *sock, struct msghdr *msg, size_t len,int flags)
/*msg：用于接收数据，获取发送套接字地址信息*/
{
    struct scm_cookie scm;
    struct sock *sk = sock->sk;
    struct netlink_sock *nlk = nlk_sk(sk);
    int noblock = flags&MSG_DONTWAIT;        /*非阻塞读*/
    size_t copied;
    struct sk_buff *skb, *data_skb;
    int err, ret;

    if (flags&MSG_OOB)
        return -EOPNOTSUPP;

    copied = 0;

    skb = skb_recv_datagram(sk, flags, noblock, &err);     /*从接收缓存队列取下一个 sk_buff 实例*/
                /*接收一个数据报 sk_buff 实例，如果没有数据包则睡眠等待，/net/core/datagram.c*/
    if (skb == NULL)
        goto out;
    data_skb = skb;
#ifdef CONFIG_COMPAT_NETLINK_MESSAGES
    ...
#endif
    nlk->max_recvmsg_len = max(nlk->max_recvmsg_len, len);
    nlk->max_recvmsg_len = min_t(size_t, nlk->max_recvmsg_len,16384);     /*最大接收数据长度*/

    copied = data_skb->len;       /*复制数据长度*/
    if (len < copied) {       /*读取数据长度小于数据包中数据长度，截短数据*/
        msg->msg_flags |= MSG_TRUNC;
        copied = len;
    }

    skb_reset_transport_header(data_skb);       /*skb->transport_header = skb->data - skb->head*/
                                                /*传输层报头指向 netlink 消息报头*/
    err = skb_copy_datagram_msg(data_skb, 0, msg, copied);
                                /*从 sk_buff 复制数据到用户空间，/include/linux/skbuff.h*/
    if (msg->msg_name) {       /*获取发送套接字地址信息*/
        DECLARE_SOCKADDR(struct sockaddr_nl *, addr, msg->msg_name);
        addr->nl_family = AF_NETLINK;
        addr->nl_pad       = 0;
        addr->nl_pid   = NETLINK_CB(skb).portid;    /*发送套接字 id 值*/
        addr->nl_groups     = netlink_group_mask(NETLINK_CB(skb).dst_group);
        msg->msg_namelen = sizeof(*addr);
    }
```

```
        if (nlk->flags & NETLINK_F_RECV_PKTINFO)
            netlink_cmsg_recv_pktinfo(msg, skb);
        if (nlk->flags & NETLINK_F_LISTEN_ALL_NSID)
            netlink_cmsg_listen_all_nsid(sk, msg, skb);

        memset(&scm, 0, sizeof(scm));
        scm.creds = *NETLINK_CREDS(skb);
        if (flags & MSG_TRUNC)
            copied = data_skb->len;

        skb_free_datagram(sk, skb);     /*释放 sk_buff 实例等*/

        if (nlk->cb_running &&atomic_read(&sk->sk_rmem_alloc) <= sk->sk_rcvbuf / 2) {
            ret = netlink_dump(sk);
            if (ret) {
                sk->sk_err = -ret;
                sk->sk_error_report(sk);
            }
        }
        scm_recv(sock, msg, &scm, flags);
    out:
        netlink_rcv_wake(sk);
                /*唤醒在 netlink_sock->wait 睡眠等待的进程等（因拥塞而睡眠的发送消息进程）*/
        return err ? : copied;
    }
```

读取消息函数中调用 skb_recv_datagram()函数在接收缓存队列中获取一个 sk_buff 实例，如果没有可用的 sk_buff 实例，则当前进程睡眠等待。接收消息函数随后从 sk_buff 实例关联的内存缓存区复制数据到用户内存，并唤醒因拥塞而在套接字上睡眠的发送消息进程。消息内容的解析和处理由用户进程负责，内核只负责将内存缓存区中数据原原本本地导出给用户进程。

12.3.4 uevent 套接字

uevent 子系统利用 netlink 套接字由内核向用户空间传递事件信息，主要用于通用驱动模型向用户空间传递设备、驱动事件信息等，套接字协议类型为 NETLINK_KOBJECT_UEVENT。

uevent 子系统在/lib/kobject_uevent.c 文件内实现。

1 创建 uevent 套接字

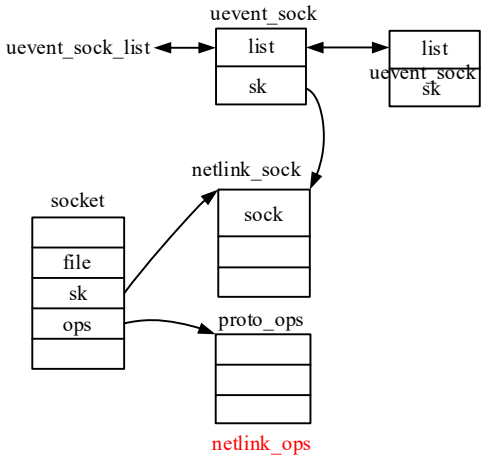
在 uevent 子系统初始化函数 kobject_uevent_init()中注册了 pernet_operations 实例：

```
static int __init kobject_uevent_init(void)
{
    return register_pernet_subsys(&uevent_net_ops);
}
postcore_initcall(kobject_uevent_init);
```

pernet_operations 实例定义如下：

```
static struct pernet_operations uevent_net_ops = {
    .init = uevent_net_init,     /*初始化函数*/
    .exit = uevent_net_exit,
};
```

初始化函数 uevent_net_init()在每次创建新网络命名空间时被调用，函数内将创建内核 netlink 套接字，也就是说每个网络命名空间内都有一个用于 uevent 子系统的内核 netlink 套接字。除了创建内核 netlink 套接字，还将创建一个 uevent_sock 实例，实例关联内核套接字 netlink_sock 实例。uevent_sock 实例由全局双链表 uevent_sock_list 管理，如下图所示：



uevent_net_init()函数定义如下：

```
static int uevent_net_init(struct net *net)
{
    struct uevent_sock *ue_sk;
    struct netlink_kernel_cfg cfg = {
        .groups    = 1,
        .flags     = NL_CFG_F_NONROOT_RECV,     /*非超级用户可接收消息*/
    };    /*内核套接字不接收用户套接字发送的消息*/

    ue_sk = kzalloc(sizeof(*ue_sk), GFP_KERNEL);       /*创建 uevent_sock 实例*/
    if (!ue_sk)
        return -ENOMEM;

    ue_sk->sk = netlink_kernel_create(net, NETLINK_KOBJECT_UEVENT, &cfg);
                                                        /*创建内核 netlink 套接字*/
    ...
    mutex_lock(&uevent_sock_mutex);
    list_add_tail(&ue_sk->list, &uevent_sock_list);     /*uevent_sock 实例插入全局双链表末尾*/
    mutex_unlock(&uevent_sock_mutex);
    return 0;
}
```

2 发送消息

NETLINK_KOBJECT_UEVENT 内核套接字只向用户空间发送消息，不接收用户套接字发送的消息，非超级用户进程可接收内核套接字发送的消息。

uevent 子系统一个重要的应用就是在向内核添加设备、驱动时，或设备状态改变时，向用户空间发送事件通知，在用户空间一般由 udev（mdev）用户进程处理事件，例如：创建设备文件等。

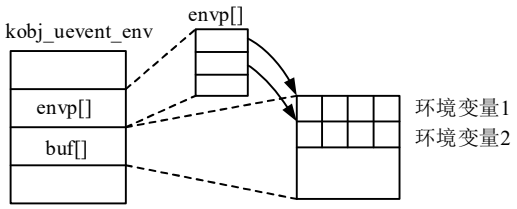
uevent 子系统发送的事件类型定义在/include/linux/kobject.h 头文件内：

```
enum kobject_action {
    KOBJ_ADD,              /*添加对象*/
    KOBJ_REMOVE,           /*移除对象*/
    KOBJ_CHANGE,           /*对象改变*/
    KOBJ_MOVE,             /*移动对象*/
    KOBJ_ONLINE,           /*上线*/
    KOBJ_OFFLINE,          /*离线*/
    KOBJ_MAX               /*事件类型数量*/
};
```

kobj_uevent_env 结构体用于缓存事件信息，定义如下（/include/linux/kobject.h）：

```
struct kobj_uevent_env {
    char *argv[3];
    char *envp[UEVENT_NUM_ENVP];        /*环境变量指针，数组项为 32*/
    int envp_idx;
    char buf[UEVENT_BUFFER_SIZE];       /*缓存环境变量，2048 字节，由套接字发送出去*/
    int buflen;                 /*实际缓存环境变量长度*/
};
```

事件信息是由形如“环境变量名称=环境变量值”的字符串组成，字符串保存在 buf[]数组，envp[]指针数组项指向各环境变量的开始位置，如下图所示，环境变量数量最大为 32 个。环境变量名称通常为事件名称。



kobject_uevent()函数是内核向用户空间发送内核事件的接口函数，函数定义在/lib/kobject_uevent.c 文件内，主要工作就是创建 kobj_uevent_env 实例，填充环境变量到 buff[]缓存区后，通过套接字将 buff[]中数据发送到用户空间，函数代码如下：

```
int kobject_uevent(struct kobject *kobj, enum kobject_action action)
{
    return kobject_uevent_env(kobj, action, NULL);     /*/lib/kobject_uevent.c*/
}
```

kobject_uevent()函数内直接调用 kobject_uevent_env()函数，代码简列如下：

```
int kobject_uevent_env(struct kobject *kobj, enum kobject_action action,char *envp_ext[])
{
    struct kobj_uevent_env *env;
    const char *action_string = kobject_actions[action];      /*事件名称字符串，/lib/kobject_uevent.c*/
    const char *devpath = NULL;
    const char *subsystem;
    struct kobject *top_kobj;
    struct kset *kset;
```

```
    const struct kset_uevent_ops *uevent_ops;
    int i = 0;
    int retval = 0;
#ifdef CONFIG_NET
    struct uevent_sock *ue_sk;
#endif
    ...
    kset = top_kobj->kset;
    uevent_ops = kset->uevent_ops;          /*uevent 操作结构实例*/

    if (kobj->uevent_suppress) {        /*如果 kobject 实例设置了不传递事件，直接返回 0*/
        ...
        return 0;
    }

    if (uevent_ops && uevent_ops->filter)
        if (!uevent_ops->filter(kset, kobj)) {     /*uevent_ops->filter()函数返回 0，则不发送事件*/
            ...
            return 0;
        }

    if (uevent_ops && uevent_ops->name)         /*获取子系统名称*/
         subsystem = uevent_ops->name(kset, kobj);
    else
         subsystem = kobject_name(&kset->kobj);       /*获取 kobject 名称*/
    ...
    env = kzalloc(sizeof(struct kobj_uevent_env), GFP_KERNEL);   /*创建 kobj_uevent_env 实例*/
    ...
    devpath = kobject_get_path(kobj, GFP_KERNEL);
                                /*获取 kobject 实例在层次结构中的路径字符串，/lib/kobject.c*/
    ...
    /*添加环境变量至 kobj_uevent_env 实例 buff[]缓存区*/
    retval = add_uevent_var(env, "ACTION=%s", action_string);
                                                /*向 kobj_uevent_env 实例添加环境变量的接口函数*/
    ...
    retval = add_uevent_var(env, "DEVPATH=%s", devpath);
    ...
    retval = add_uevent_var(env, "SUBSYSTEM=%s", subsystem);
    ...
    /* kset 添加环境变量*/
    if (uevent_ops && uevent_ops->uevent) {
        retval = uevent_ops->uevent(kset, kobj, env);        /*添加环境变量*/
        ...
    }

    if (action == KOBJ_ADD)             /*设置 kobject 状态*/
```

```
        kobj->state_add_uevent_sent = 1;
    else if (action == KOBJ_REMOVE)
        kobj->state_remove_uevent_sent = 1;

    mutex_lock(&uevent_sock_mutex);
    retval = add_uevent_var(env, "SEQNUM=%llu", (unsigned long long)++uevent_seqnum);
    ...

#if defined(CONFIG_NET)     /*通过 netlink 套接字发送信息*/
    list_for_each_entry(ue_sk, &uevent_sock_list, list) {     /*遍历各网络命名空间对应的 sock 实例*/
        struct sock *uevent_sock = ue_sk->sk;     /*sock 实例指针*/
        struct sk_buff *skb;
        size_t len;

        if (!netlink_has_listeners(uevent_sock, 1))     /*检测组播组 1 是否有监听者*/
            continue;

        /*有监听者，需要组播消息*/
        len = strlen(action_string) + strlen(devpath) + 2;
        skb = alloc_skb(len + env->buflen, GFP_KERNEL);     /*分配 sk_buff 实例和内存缓存*/
        if (skb) {
            char *scratch;

            scratch = skb_put(skb, len);     /*skb->tail 增加 len，增加数据区长度*/
            sprintf(scratch, "%s@%s", action_string, devpath);
                                            /*action_string, devpath 添加到内存缓存*/

            for (i = 0; i < env->envp_idx; i++) {        /*遍历环境变量*/
                len = strlen(env->envp[i]) + 1;
                scratch = skb_put(skb, len);
                strcpy(scratch, env->envp[i]);     /*复制环境变量至 sk_buff 内存缓存区*/
            }

            NETLINK_CB(skb).dst_group = 1;       /*发送到组播组 1*/
            retval = netlink_broadcast_filtered(uevent_sock, skb,
                                    0, 1, GFP_KERNEL,kobj_bcast_filter,kobj);   /*组播消息*/
            ...
        } else
            retval = -ENOMEM;
    }
#endif
    mutex_unlock(&uevent_sock_mutex);

#ifdef CONFIG_UEVENT_HELPER
    ...          /*老的向用户空间传递信息的机制*/
#endif
```

```
exit:
        kfree(devpath);
        kfree(env);          /*释放 kobj_uevent_env 实例*/
        return retval;
}
```

kobject_uevent_env()函数主要工作是将环境变量写入 kobj_uevent_env 实例的 buff[]缓存区，然后分配 sk_buff 实例和内存缓存区，将 buff[]缓存中环境变量复制到 sk_buff 实例关联的内存缓存区，最后将 sk_buff 实例发送到 uevent 套接字组播组 1。内存缓存区中没有通过消息的类型传递数据，而是直接将 buff[]缓存中数据复制到 sk_buff 关联的内存缓存区。

用户进程套接字需要接收 uevent 套接字消息时，先创建 NETLINK_KOBJECT_UEVENT 协议类型的 netlink 数据报类型套接字，然后调用 bind()系统调用绑定地址，重点是设置 sockaddr_nl.**nl_groups** 监听组播组位图，必须设置组播组 1 的 bit 位（bit0），以接收组播组 1 的组播消息。用户进程通过 recv()、read()等系统调用读取内核发送的事件消息，如果没有收到消息，进程将进入睡眠等待。

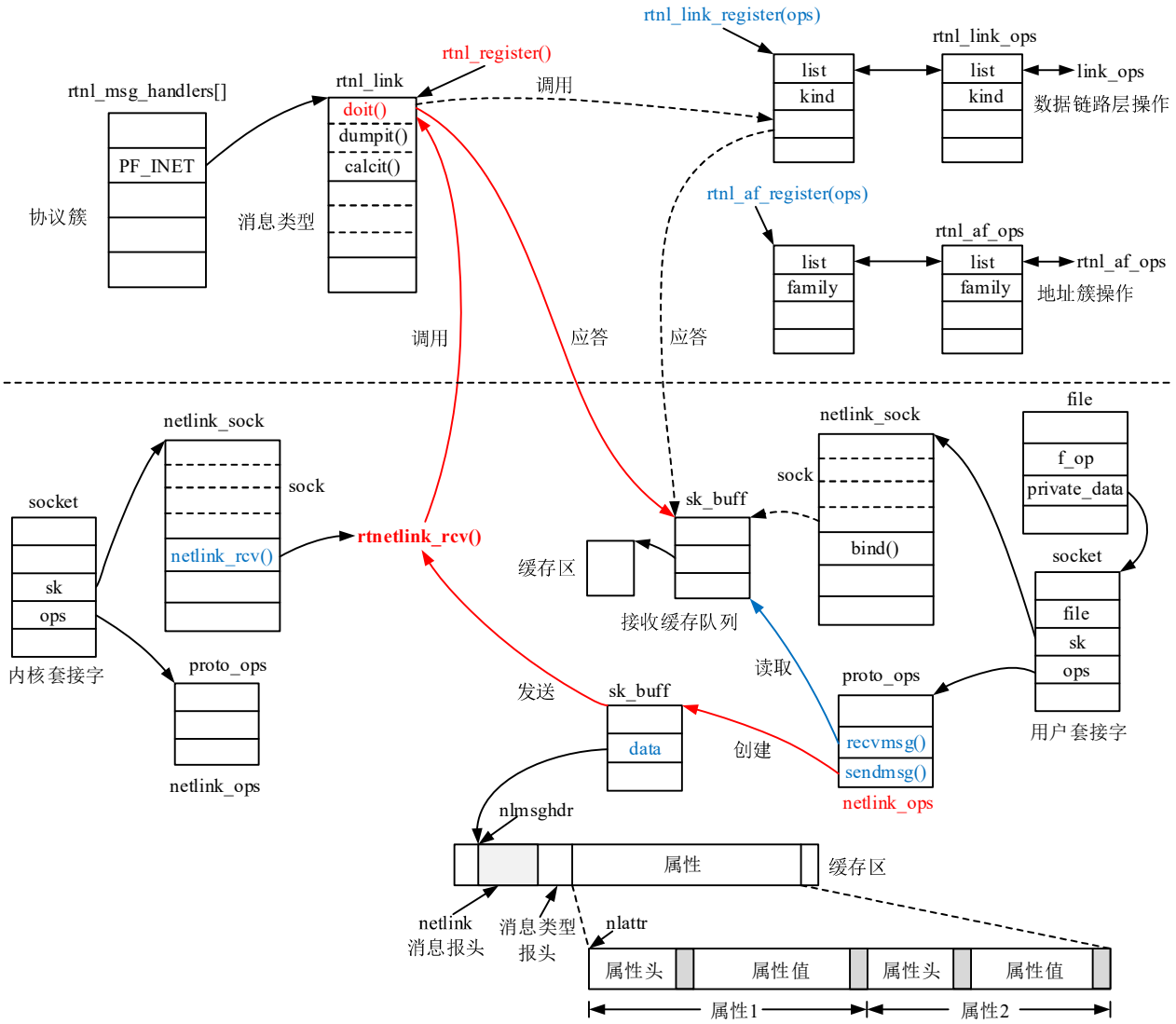
12.3.5 NETLINK_ROUTE 套接字

NETLINK_ROUTE（0）是 netlink 套接字的一种协议类型，此协议类型主要用于用户与网络子系统通信，如设置/获取网络路由选择子系统消息、邻居子系统消息、网络接口消息、防火墙消息、策略路由消息等。常用的用户工具 ip 就是通过 NETLINK_ROUTE 套接字与内核通信。读者可先学习完第 13 章因特网网络层知识后再回过头来学习这一小节。

NETLINK_ROUTE 套接字（rtnetlink）实现代码在/net/core/rtnetlink.c 文件内。

1 概述

NETLINK_ROUTE 套接字框架如下图所示：



NETLINK_ROUTE 套接字主要用于设置/获取网络参数、网络设备参数，它适用于内核支持的所有网络类型（通信域）。

NETLINK_ROUTE 套接字为每个协议簇创建 rtnl_link 结构体数组，全局指针数组 rtnl_msg_handlers[]各数组项指向各协议簇的 rtnl_link 数组。每个 rtnl_link 实例用于处理一个类型的消息，rtnl_link 结构体中主要包含处理消息的函数指针成员。所有协议簇共用一套消息类型。

内核在初始化时将创建内核 rtnetlink 套接字。各协议簇协议实现代码中需要定义各类型消息的处理函数，并调用 **rtnl_register()**函数将处理函数指针填充到对应 rtnl_link 实例中（如果 rtnl_link 数组尚未创建则先创建数组）。

用户进程通过向内核套接字发送某一类型的消息来与内核套接字通信。用户进程发送消息时，需要在 netlink 消息报头中写入消息类型（**nlmsghdr.nlmsg_type**），在数据区的开始处写入特定于消息类型的报头。所有消息类型报头第一个成员必须为协议簇标识，用于内核确定协议簇，在消息类型报头后面写入属性。

内核 rtnetlink 套接字接收到消息后，调用 **rtnetlink_rcv()**函数处理消息，此函数根据协议簇标识在指针数组 rtnl_msg_handlers[]中查找 rtnl_link 数组，根据消息类型在数组中查找对应的 rtnl_link 实例，并调用其中的消息处理函数处理消息。

NETLINK_ROUTE 套接字还定义了数据链路层操作 rtnl_link_ops 结构体和地址簇操作 rtnl_af_ops 结构体，用于实现消息处理操作中特定于数据链路层和地址簇的操作，在消息处理函数中可调用这两个结构体实例中的处理函数，执行特定的操作。协议簇实现代码可定义这两个结构体实例并注册，结构体实例分别由全局双链表管理。

总之，使用 NETLINK_ROUTE 套接字的协议簇，需要定义并注册各类型消息的处理函数。用户套接

字可通过协议簇（地址簇）、消息类型和消息数据与内核套接字通信，用于获取/设置网络子系统（网络设备）参数。

内核在/include/uapi/linux/rtnetlink.h 头文件定义了 NETLINK_ROUTE 套接字的消息类型，如下所示：

```
enum {
    RTM_BASE   = 16,          /*消息类型基数，小于 16 的消息类型为控制消息保留*/

    RTM_NEWLINK  = 16,    /*网络接口（网络设备）参数，如 MAC 地址等*/
    RTM_DELLINK,
    RTM_GETLINK,
    RTM_SETLINK,

    RTM_NEWADDR = 20,   /*网络层地址，如 IP 地址*/
    RTM_DELADDR,
    RTM_GETADDR,

    RTM_NEWROUTE     = 24,    /*添加、删除、获取路由*/
    RTM_DELROUTE,
    RTM_GETROUTE,

    RTM_NEWNEIGH     = 28,    /*邻居操作*/
    RTM_DELNEIGH,
    RTM_GETNEIGH,

    RTM_NEWRULE  = 32,
    RTM_DELRULE,
    RTM_GETRULE,

    RTM_NEWQDISC= 36,      /*发送数据包排队规则、调度器操作*/
    RTM_DELQDISC,
    RTM_GETQDISC,

    RTM_NEWTCLASS    = 40,
    RTM_DELTCLASS,
    RTM_GETTCLASS,

    RTM_NEWTFILTER   = 44,   /*Netfilter*/
    RTM_DELTFILTER,
    RTM_GETTFILTER,

    RTM_NEWACTION    = 48,
    RTM_DELACTION,
    RTM_GETACTION,

    RTM_NEWPREFIX    = 52,
```

```
    RTM_GETMULTICAST = 58,      /*组播*/

    RTM_GETANYCAST   = 62,

    RTM_NEWNEIGHTBL= 64,    /*邻居表*/
    RTM_GETNEIGHTBL  = 66,
    RTM_SETNEIGHTBL,

    RTM_NEWNDUSEROPT = 68,

    RTM_NEWADDRLABEL = 72,
    RTM_DELADDRLABEL,
    RTM_GETADDRLABEL,

    RTM_GETDCB = 78,
    RTM_SETDCB,

    RTM_NEWNETCONF = 80,   /*配置信息*/
    RTM_GETNETCONF = 82,

    RTM_NEWMDB = 84,
    RTM_DELMDB = 85,
    RTM_GETMDB = 86,

    RTM_NEWNSID = 88,
    RTM_DELNSID = 89,
    RTM_GETNSID = 90,

    __RTM_MAX,
    #define RTM_MAX      (((__RTM_MAX + 3) & ~3) - 1)     /*最大消息类型值*/
};
```

以上消息类型中，每 4 个为一组，分别代表添加、删除、获取、设置参数等。

2 数据结构

NETLINK_ROUTE 套接字中主要的数据结构有 rtnl_link、rtnl_link_ops、rtnl_af_ops 等。下面将介绍这几个数据结构，并介绍注册结构体实例的方法。

■rtnl_link

rtnl_link 结构体定义在/net/core/rtnetlink.c 文件内，如下所示：

```
struct rtnl_link {
    rtnl_doit_func      doit;       /*用于指定添加、删除、修改等操作*/
    rtnl_dumpit_func    dumpit;     /*用于获取信息（获取参数）*/
    rtnl_calcit_func    calcit;     /*用于计算缓存区大小*/
};
```

rtnl_link 结构体中包含 3 个函数指针，函数原型定义如下（/include/net/rtnetlink.h）：

```
typedef int (*rtnl_doit_func)(struct sk_buff *, struct nlmsghdr *);
typedef int (*rtnl_dumpit_func)(struct sk_buff *, struct netlink_callback *);
typedef u16 (*rtnl_calcit_func)(struct sk_buff *, struct nlmsghdr *);
```

注册 rtnl_link 结构体实例的函数为 **rtnl_register()**，函数参数中需要指明协议簇（网络协议类型）、消息类型、以及三个处理函数指针，函数定义如下：

```
void rtnl_register(int protocol, int msgtype,
                                        rtnl_doit_func doit, rtnl_dumpit_func dumpit,rtnl_calcit_func calcit)
{
    if (__rtnl_register(protocol, msgtype, doit, dumpit, calcit) < 0)
        panic(...);
}
```

rtnl_register()函数调用__rtnl_register()函数完成注册操作，此函数内根据协议簇在 rtnl_msg_handlers[]指针数组中查找协议类型关联的 rtnl_link 结构体数组（如果不存在则创建），通过消息类型在数组中查找对应的 rtnl_link 实例，并将参数传递的函数指针赋予此实例。

■rtnl_af_ops

rtnl_af_ops 结构体定义在/include/net/rtnetlink.h 头文件，表示协议簇操作，结构体定义如下：

```
struct rtnl_af_ops {
    struct list_head    list;    /*双链表成员，将实例添加到 rtnl_af_ops 双链表*/
    int         family;     /*地址簇，用于标识 rtnl_af_ops 实例（用于查找实例）*/

    int         (*fill_link_af)(struct sk_buff *skb,const struct net_device *dev); /*填充地址簇特有属性*/
    size_t      (*get_link_af_size)(const struct net_device *dev);   /*计算地址簇特有属性的大小*/

    int         (*validate_link_af)(const struct net_device *dev,const struct nlattr *attr);
                                                        /*特有属性有效性检查*/
    int         (*set_link_af)(struct net_device *dev,const struct nlattr *attr);    /*解析属性*/
};
```

内核定义了全局双链表 rtnl_af_ops 用于管理 rtnl_af_ops 结构体实例，**rtnl_af_register**(struct rtnl_af_ops *ops)为注册实例函数，函数内唯一工作就是将实例插入到 rtnl_af_ops 双链表末尾。

协议簇实现代码可定义并注册 rtnl_af_ops 实例，用于对消息进行特定于协议簇的处理。

■rtnl_link_ops

rtnl_link_ops 结构体定义在/include/net/rtnetlink.h 头文件，表示数据链路层操作结构，结构体定义如下：

```
struct rtnl_link_ops {
    struct list_head    list;       /*双链表成员*/
    const char          *kind;    /*名称，标识 rtnl_link_ops 实例，通过名称查找*/

    size_t      priv_size;    /*网络设备私有空间*/
    void        (*setup)(struct net_device *dev);    /*网络接口（网络设备）启动函数*/
```

```
    int         maxtype;      /*属性最大值*/
    const struct nla_policy   *policy;    /*属性有效性检查策略*/
    int   (*validate)(struct nlattr *tb[],struct nlattr *data[]);

    int   (*newlink)(struct net *src_net,struct net_device *dev,struct nlattr *tb[],struct nlattr *data[]);
                                                              /*配置和注册新网络设备*/
    int   (*changelink)(struct net_device *dev,struct nlattr *tb[],struct nlattr *data[]);
                                                          /*修改现有网络设备参数*/
    void  (*dellink)(struct net_device *dev,struct list_head *head);      /*移除网络设备*/

    size_t     (*get_size)(const struct net_device *dev);   /*计算转储网络设备属性所需空间*/
    int        (*fill_info)(struct sk_buff *skb,const struct net_device *dev);   /*转储网络设备属性*/
    size_t     (*get_xstats_size)(const struct net_device *dev);
    int        (*fill_xstats)(struct sk_buff *skb,const struct net_device *dev);
    unsigned int   (*get_num_tx_queues)(void);   /*确定新创建网络设备发送队列数量*/
    unsigned int   (*get_num_rx_queues)(void);    /*确定新创建网络设备接收队列数量*/


    int        slave_maxtype;
    const struct nla_policy   *slave_policy;
    int        (*slave_validate)(struct nlattr *tb[],struct nlattr *data[]);
    int        (*slave_changelink)(struct net_device *dev,struct net_device *slave_dev,
                                   struct nlattr *tb[],struct nlattr *data[]);
    size_t     (*get_slave_size)(const struct net_device *dev,const struct net_device *slave_dev);
    int        (*fill_slave_info)(struct sk_buff *skb,const struct net_device *dev,
                                                      const struct net_device *slave_dev);
    struct net   *(*get_link_net)(const struct net_device *dev);   /*网络设备所属网络命名空间*/
};
```

内核定义全局双链表 link_ops 用于管理 rtnl_link_ops 结构体实例，**rtnl_link_register**(struct rtnl_link_ops *ops)函数用于注册实例，函数内主要工作就是将实例插入到 link_ops 双链表末尾。

数据链路层协议可定义并注册 rtnl_link_ops 结构体实例，用于处理 RTM_NEWLINK 等类型消息，主要是创建网络接口，获取/设置网络接口参数等。

3 初始化

NETLINK_ROUTE 套接字初始化函数为 rtnetlink_init()，在 netlink 套接字初始化函数 netlink_proto_init()中将调用此函数，函数定义如下（/net/core/rtnetlink.c）：

```
void __init rtnetlink_init(void)
{
    if (register_pernet_subsys(&rtnetlink_net_ops))   /*注册 rtnetlink_net_ops 实例*/
        panic("rtnetlink_init: cannot initialize rtnetlink\n");

    register_netdevice_notifier(&rtnetlink_dev_notifier);
    /*向通知链 netdev_chain 注册通知 rtnetlink_dev_notifier 实例，发生设备事件时向用户发送消息*/
```

```
    /*注册 rtnl_link 实例，PF_UNSPEC 表示不特定于某个协议簇（适用于所有协议簇）*/
    rtnl_register(PF_UNSPEC, RTM_GETLINK, rtnl_getlink,rtnl_dump_ifinfo, rtnl_calcit);
    rtnl_register(PF_UNSPEC, RTM_SETLINK, rtnl_setlink, NULL, NULL);
    rtnl_register(PF_UNSPEC, RTM_NEWLINK, rtnl_newlink, NULL, NULL);
    rtnl_register(PF_UNSPEC, RTM_DELLINK, rtnl_dellink, NULL, NULL);

    rtnl_register(PF_UNSPEC, RTM_GETADDR, NULL, rtnl_dump_all, NULL);
    rtnl_register(PF_UNSPEC, RTM_GETROUTE, NULL, rtnl_dump_all, NULL);

    rtnl_register(PF_BRIDGE, RTM_NEWNEIGH, rtnl_fdb_add, NULL, NULL);
    rtnl_register(PF_BRIDGE, RTM_DELNEIGH, rtnl_fdb_del, NULL, NULL);
    rtnl_register(PF_BRIDGE, RTM_GETNEIGH, NULL, rtnl_fdb_dump, NULL);

    rtnl_register(PF_BRIDGE, RTM_GETLINK, NULL, rtnl_bridge_getlink, NULL);
    rtnl_register(PF_BRIDGE, RTM_DELLINK, rtnl_bridge_dellink, NULL, NULL);
    rtnl_register(PF_BRIDGE, RTM_SETLINK, rtnl_bridge_setlink, NULL, NULL);
}
```

rtnetlink_init()初始化函数中注册了 pernet_operations 结构体实例 **rtnetlink_net_ops**，此实例初始化函数为 **rtnetlink_net_init()**，其主要工作是创建 NETLINK_ROUTE 内核套接字，代码如下（/net/core/rtnetlink.c）：

```
static int __net_init rtnetlink_net_init(struct net *net)
{
    struct sock *sk;
    struct netlink_kernel_cfg cfg = {
        .groups     = RTNLGRP_MAX,
        .input      = rtnetlink_rcv,      /*接收用户套接字消息，后面再介绍*/
        .cb_mutex   = &rtnl_mutex,
        .flags      = NL_CFG_F_NONROOT_RECV,
    };

    sk = netlink_kernel_create(net, NETLINK_ROUTE, &cfg);
                                    /*创建内核 NETLINK_ROUTE 套接字，属于网络命名空间*/
    if (!sk)
        return -ENOMEM;
    net->rtnl = sk;     /*网络命名空间 net->rtnl 指向 rtnetlink 套接字*/
    return 0;
}
```

rtnetlink_init()函数内还注册 PF_UNSPEC 协议簇消息类型对应的 rtnl_link 实例，消息处理函数请读者自行阅读源代码。

在注册消息类型处理函数时，需要指明协议簇，如果协议簇指定为 PF_UNSPEC，表示此处理函数适用于所有协议簇。但是，内核处理消息时会优先调用消息类型报头中指定协议簇注册的处理函数，如果指定协议簇没有定义此类型消息的处理函数，再调用 PF_UNSPEC 协议簇注册的消息处理函数。

4 内核处理消息

NETLINK_ROUTE 套接字通常是用户进程向内核套接字发送消息，内核接收处理消息，并发送应答消息。由前面介绍的初始化函数可知，NETLINK_ROUTE 内核套接字通过 **rtnetlink_rcv()**函数接收用户套接字消息，并根据需要发送应答消息，函数定义如下：

```
static void rtnetlink_rcv(struct sk_buff *skb)
{
    rtnl_lock();
    netlink_rcv_skb(skb, &rtnetlink_rcv_msg);
    rtnl_unlock();
}
```

netlink_rcv_skb()函数在前面介绍过了，它将调用 rtnetlink_rcv_msg()函数接收消息，函数定义如下：

```
static int rtnetlink_rcv_msg(struct sk_buff *skb, struct nlmsghdr *nlh)
{
    struct net *net = sock_net(skb->sk);        /*网络命名空间*/
    rtnl_doit_func doit;        /*消息处理函数*/
    int sz_idx, kind;
    int family;       /*地址簇*/
    int type;         /*消息类型*/
    int err;

    type = nlh->nlmsg_type;       /*获取消息类型*/
    if (type > RTM_MAX)
        return -EOPNOTSUPP;

    type -= RTM_BASE;     /*消息对应 rtnl_link 数组的索引值，内核内部使用的消息索引值*/

    /* All the messages must have at least 1 byte length */
    if (nlmsg_len(nlh) < sizeof(struct rtgenmsg))
        return 0;

    family = ((struct rtgenmsg *)nlmsg_data(nlh))->rtgen_family;     /*获取协议簇标识*/
    sz_idx = type>>2;     /*消息类型分类，4 个一组，sz_idx 标识分组，RTM_NR_FAMILIES*/
    kind = type&3;        /*检测是否是获取信息消息（获取参数）*/

    if (kind != 2 && !netlink_net_capable(skb, CAP_NET_ADMIN))
        return -EPERM;

    if (kind == 2 && nlh->nlmsg_flags&NLM_F_DUMP) { /*调用 rtnl_link 实例中获取信息函数*/
        struct sock *rtnl;
        rtnl_dumpit_func dumpit;
        rtnl_calcit_func calcit;
        u16 min_dump_alloc = 0;

        dumpit = rtnl_get_dumpit(family, type);     /*获取消息函数*/
```

```
            if (dumpit == NULL)
                return -EOPNOTSUPP;
            calcit = rtnl_get_calcit(family, type);
            if (calcit)
                min_dump_alloc = calcit(skb, nlh);

            __rtnl_unlock();
            rtnl = net->rtnl;
            {
                struct netlink_dump_control c = {
                    .dump           = dumpit,
                    .min_dump_alloc    = min_dump_alloc,
                };
                err = netlink_dump_start(rtnl, skb, nlh, &c);
            }
            rtnl_lock();
            return err;
        }    /*处理获取信息消息*/
        /*处理其它消息*/
        doit = rtnl_get_doit(family, type);    /*family：协议簇，type：消息类型*/
                /*获取消息处理函数指针，如果指定协议簇中注册了处理函数，
                *则返回指定协议簇注册的处理函数，否则查找 PF_UNSPEC 协议簇中注册的处理函数，
                *有则返回，没有则返回 NULL。
                */
        if (doit == NULL)
            return -EOPNOTSUPP;

        return doit(skb, nlh);        /*调用消息处理函数*/
    }
```

netlink_rcv_skb()函数在用户消息报头中提取协议簇标识和消息类型，根据是否是检索（获取）参数的消息，查找对应的 rtnl_link 实例中的相应函数并调用，以实现消息的处理。

使用 rtnetlink 套接字的子系统需要定义消息类型对应的处理函数，并调用 **rtnl_register()**函数向内核注册。在后面介绍 IPv4 网络协议中路由选择和邻居子系统时将会介绍它们如何利用 rtnetlink 套接字传输消息。

12.3.6 NETLINK_NETFILTER 套接字

NETLINK_NETFILTER 协议类型的 netlink 套接字，用于 netfilter（网络过滤）子系统。netfilter 子系统提供了一个框架，它支持在数据包网络栈传输路径的各个地方（挂载点）注册回调函数，从而对数据包执行各种过滤操作，如修改地址或端口、丢弃数据包、定入日志等。读者可学习完第 13 章内容后再回过头来学习本小节内容。

NETLINK_NETFILTER 协议类型套接字用于实现 netfilter 子系统与用户之间的通信，此套接字适用于所有协议簇，并不专用于哪个网络协议簇。

NETLINK_NETFILTER 协议类型套接字实现代码位于/net/netfilter/nfnetlink.c 文件内。

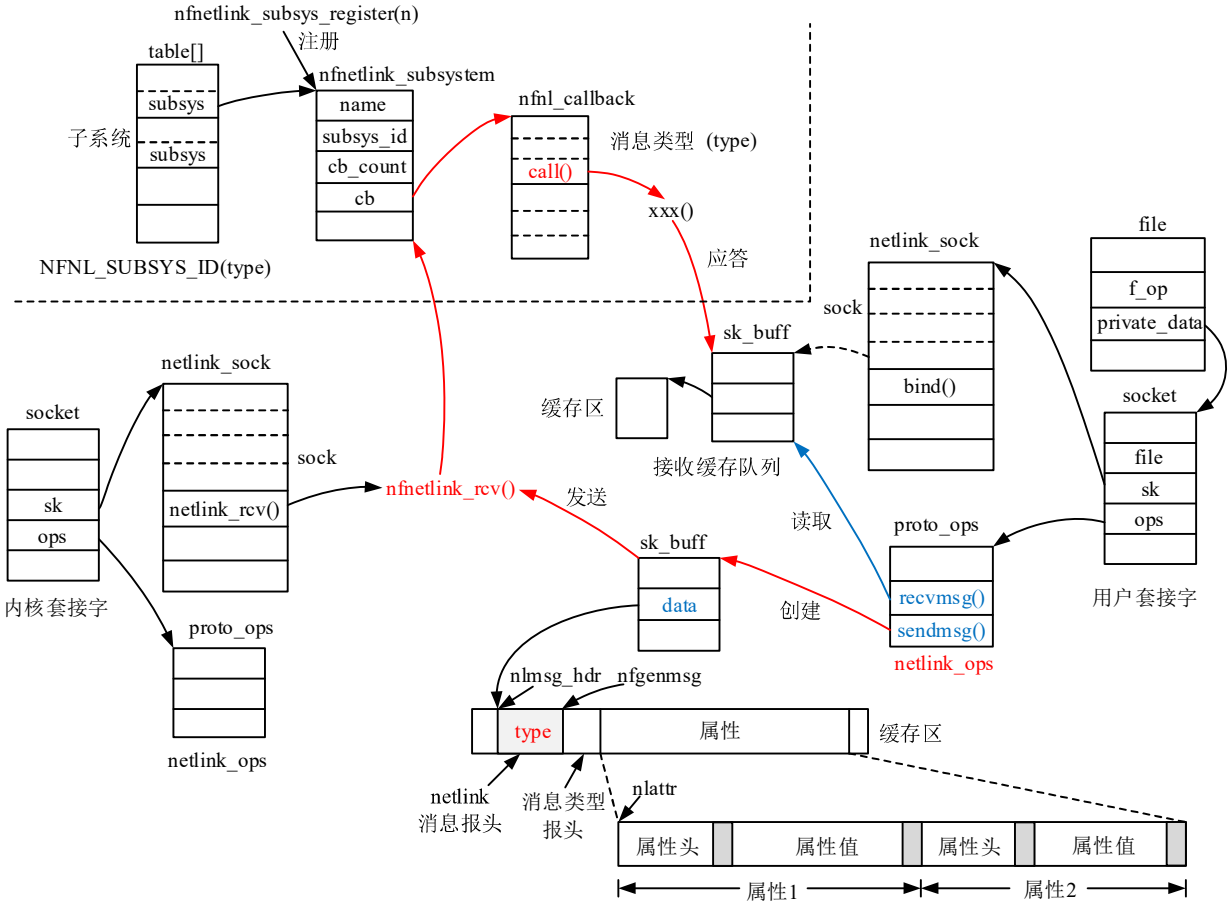
1 概述

内核中每个使用 netfilter 子系统实现某项功能的部分，称之为一个子系统，子系统标识定义如下：

```
/*/include/uapi/linux/netfilter/nfnetlink.h*/
#define   NFNL_SUBSYS_NONE                0
#define   NFNL_SUBSYS_CTNETLINK           1
#define   NFNL_SUBSYS_CTNETLINK_EXP       2
#define   NFNL_SUBSYS_QUEUE               3
#define   NFNL_SUBSYS_ULOG                4
#define   NFNL_SUBSYS_OSF           5
#define   NFNL_SUBSYS_IPSET         6
#define   NFNL_SUBSYS_ACCT          7
#define   NFNL_SUBSYS_CTNETLINK_TIMEOUT     8
#define   NFNL_SUBSYS_CTHELPER            9
#define   NFNL_SUBSYS_NFTABLES            10      /*数据包选择*/
#define   NFNL_SUBSYS_NFT_COMPAT          11
#define   NFNL_SUBSYS_COUNT               12      /*子系统数量*/
```

■套接字框架

NETLINK_NETFILTER 协议类型套接字实现框架如下图所示：



netfilter 各子系统若需要与用户进程通信，需要定义并注册 nfnetlink_subsystem 结构体实例，此实例由全局指针数组 table[]管理，子系统标识作为数组项索引值。nfnetlink_subsystem 结构体关联一个 nfnl_callback 结构体数组，nfnl_callback 结构体表示对一个类型消息的处理，在其处理函数中实现对消息的处理。

netlink 消息报头 **nlmsg_type** 成员，其高 8 位用于表示标识子系统，低 8 位表示消息类型标识。消息类型报头都由 nfgenmsg 结构体表示，结构体第一个成员标识了消息适用的协议簇（如 IPv4、IPv6 等）。

■数据结构

nfnetlink_subsystem 结构体由 netfilter 各子系统定义并注册，结构体定义如下：

```
struct nfnetlink_subsystem {     /*/include/linux/netfilter/nfnetlink.h*/
    const char *name;            /*名称*/
    __u8 subsys_id;              /*nfnetlink 子系统标识（id）*/
    __u8 cb_count;               /*cb 指向 nfnl_callback 数组项数*/
    const struct nfnl_callback *cb;    /*指向 nfnl_callback 数组*/
    int (*commit)(struct sk_buff *skb);
    int (*abort)(struct sk_buff *skb);
};
```

nfnetlink_subsystem 结构体主要成员简介如下：

●**subsys_id**：子系统标识。

●**cb_count**：cb 指向 nfnl_callback 数组项数。

●**cb**：指向 nfnl_callback 结构体数组，用于处理消息。

nfnl_callback 结构体用于消息处理，定义如下（/include/linux/netfilter/nfnetlink.h）：

```
struct nfnl_callback {
    int (*call)(struct sock *nl, struct sk_buff *skb, const struct nlmsghdr *nlh,
                                                    const struct nlattr * const cda[]);
                                                    /*处理单个消息函数*/
    int (*call_rcu)(struct sock *nl, struct sk_buff *skb, const struct nlmsghdr *nlh,
                                                    const struct nlattr * const cda[]);
    int (*call_batch)(struct sock *nl, struct sk_buff *skb,const struct nlmsghdr *nlh,
                                                    const struct nlattr * const cda[]);
                                                    /*批处理消息*/
    const struct nla_policy *policy;    /* netlink attribute policy ， 属性有效性检查*/
    const u_int16_t attr_count;         /* number of nlattr's ， 属性数量*/
};
```

使用 NETLINK_NETFILTER 套接字的子系统，需定义 nfnetlink_subsystem 实例及关联的 nfnl_callback 数组，调用 int nfnetlink_subsys_register(const struct nfnetlink_subsystem *n)函数注册 nfnetlink_subsystem 实例，其主要工作就是将实例关联到 table[]全局指针数组项。

2 初始化

NETLINK_NETFILTER 协议类型套接字初始化函数 nfnetlink_init()定义如下：

```
static int __init nfnetlink_init(void)
{
    int i;

    for (i = NFNLGRP_NONE + 1; i <= NFNLGRP_MAX; i++)
        BUG_ON(nfnl_group2type[i] == NFNL_SUBSYS_NONE);

    for (i=0; i<NFNL_SUBSYS_COUNT; i++)     /*初始化 table[]数组*/
        mutex_init(&table[i].mutex);
```

```
    pr_info("Netfilter messages via NETLINK v%s.\n", nfversion);
    return register_pernet_subsys(&nfnetlink_net_ops);
}

module_init(nfnetlink_init);      /*加载模块或内核初始化时调用此函数*/
```

初始化函数 nfnetlink_init()中将注册 pernet_operations 结构体实例 nfnetlink_net_ops，定义如下：

```
static struct pernet_operations nfnetlink_net_ops = {
    .init       = nfnetlink_net_init,
    .exit_batch     = nfnetlink_net_exit_batch,
};

nfnetlink_net_ops 实例初始化函数为 nfnetlink_net_init()，定义如下：
static int __net_init nfnetlink_net_init(struct net *net)
{
    struct sock *nfnl;
    struct netlink_kernel_cfg cfg = {
        .groups     = NFNLGRP_MAX,      /*监听组数量*/
        .input = nfnetlink_rcv,     /*接收用户消息函数*/
    #ifdef CONFIG_MODULES
        .bind  = nfnetlink_bind,
    #endif
    };

    nfnl = netlink_kernel_create(net, NETLINK_NETFILTER, &cfg);      /*创建内核套接字*/
    if (!nfnl)
        return -ENOMEM;
    net->nfnl_stash = nfnl;      /*关联网络命名空间*/
    rcu_assign_pointer(net->nfnl, nfnl);
    return 0;
}
```

nfnetlink_net_init()函数的主要是创建 NETLINK_NETFILTER 协议类型内核套接字，其接收用户消息的函数为 **nfnetlink_rcv()**。

NETLINK_NETFILTER 套接字监听组最大数量为 NFNLGRP_MAX，监听组标识定义如下：

```
enum nfnetlink_groups {     /*/include/uapi/linux/netfilter/nfnetlink.h*/
    NFNLGRP_NONE,
    NFNLGRP_CONNTRACK_NEW,
    NFNLGRP_CONNTRACK_UPDATE,
    NFNLGRP_CONNTRACK_DESTROY,
    NFNLGRP_CONNTRACK_EXP_NEW,
    NFNLGRP_CONNTRACK_EXP_UPDATE,
    NFNLGRP_CONNTRACK_EXP_DESTROY,
    NFNLGRP_NFTABLES,
    NFNLGRP_ACCT_QUOTA,
    __NFNLGRP_MAX,
```

```
};
#define NFNLGRP_MAX     (__NFNLGRP_MAX - 1)
```

3 消息传输

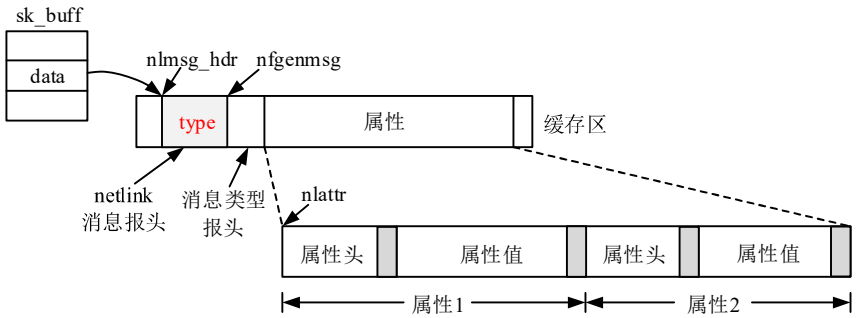
NETLINK_NETFILTER 消息报头由 nfgenmsg 结构体表示，消息的数据由属性组成。内核套接字通过函数 **nfnetlink_rcv()**函数接收处理用户消息。

■消息格式

NETLINK_NETFILTER 消息类型报头由 nfgenmsg 表示，定义如下：

```
struct nfgenmsg {                   /*/include/uapi/linux/netfilter/nfnetlink.h*/
    __u8    nfgen_family;           /* AF_xxx，地址簇*/
    __u8    version;            /* nfnetlink version，版本号*/
    __be16      res_id;         /*批处理消息时，表示子系统 id*/
};
```

消息格式如下图所示：



netlink 消息报头 **nlmsg_type** 成员，其高 8 位表示子系统标识，低 8 位表示消息类型标识。以下宏分别用于提取子系统标识和消息类型标识（/include/uapi/linux/netfilter/nfnetlink.h）：

```
#define   NFNL_SUBSYS_ID(x)   ((x & 0xff00) >> 8)
#define   NFNL_MSG_TYPE(x)    (x & 0x00ff)
```

消息类型小于 0x10（16）的值为控制消息预留，随后两个消息类型定义如下：

```
#define   NFNL_MSG_BATCH_BEGIN          NLMSG_MIN_TYPE     /*0x10，批处理消息*/
#define   NFNL_MSG_BATCH_END            NLMSG_MIN_TYPE+1
...                                     /*之后的消息类型由各子系统定义*/
```

■接收消息

内核套接字通过 **nfnetlink_rcv()**函数接收处理用户发送的消息，函数定义如下：

```
static void nfnetlink_rcv(struct sk_buff *skb)
{
    struct nlmsghdr *nlh = nlmsg_hdr(skb);      /*netlink 消息报头*/
    u_int16_t res_id;
    int msglen;

    if (nlh->nlmsg_len < NLMSG_HDRLEN ||skb->len < nlh->nlmsg_len)
        return;
```

```
    if (!netlink_net_capable(skb, CAP_NET_ADMIN)) {     /*能力检查*/
        netlink_ack(skb, nlh, -EPERM);
        return;
    }

    if (nlh->nlmsg_type == NFNL_MSG_BATCH_BEGIN) {
                                        /*批处理消息，内存缓存区中有多个消息*/
        struct nfgenmsg *nfgenmsg;      /*消息类型报头*/

        msglen = NLMSG_ALIGN(nlh->nlmsg_len);
        if (msglen > skb->len)
            msglen = skb->len;

        if (nlh->nlmsg_len < NLMSG_HDRLEN ||
            skb->len < NLMSG_HDRLEN + sizeof(struct nfgenmsg))
            return;

        nfgenmsg = nlmsg_data(nlh);
        skb_pull(skb, msglen);
        /* Work around old nft using host byte order */
        if (nfgenmsg->res_id == NFNL_SUBSYS_NFTABLES)
            res_id = NFNL_SUBSYS_NFTABLES;
        else
            res_id = ntohs(nfgenmsg->res_id);
        nfnetlink_rcv_batch(skb, nlh, res_id);     /*批处理消息*/
    } else {
        netlink_rcv_skb(skb, &nfnetlink_rcv_msg);   /*调用 nfnetlink_rcv_msg()函数处理消息*/
    }
}
```

nfnetlink_rcv()函数内判断消息类型是否是批处理消息，批处理消息表示内存缓存区中存在多个消息，此时调用 nfnetlink_rcv_batch()函数批处理消息，此函数内逐个取出消息调用 nfnl_callback->call_batch()函数处理消息。

如果内存缓存区中只有一个消息，则调用 nfnetlink_rcv_msg()函数处理消息，函数内根据子系统 id 和消息类型值（nfnl_callback 数组索引值）查找到对应的 nfnl_callback 实例，调用其中的 call_rcu()或 call()函数处理消息。

12.3.7 通用 netlink 套接字

netlink 套接字的一个缺点是协议类型数量不能超过 32（MAX_LINKS），而 netlink 套接字被内核各子系统广泛使用，32 个协议类型不够用，这是开发通用 netlink 套接字的主要原因之一。

通用 netlink 套接字协议类型为 NETLINK_GENERIC。通用 netlink 套接字下管理着多个通用簇，每个通用簇由名称和 id 值标识，通用簇中还关联了命令列表（含命令值和处理函数）。使用通用 netlink 套接字的子系统需要定义并注册一个通用簇，内含命令列表及处理函数。用户套接字通过名称寻址通用簇，通过向通用簇发送命令的形式，调用通用簇中的命令处理函数，实现与内核套接字的通信。

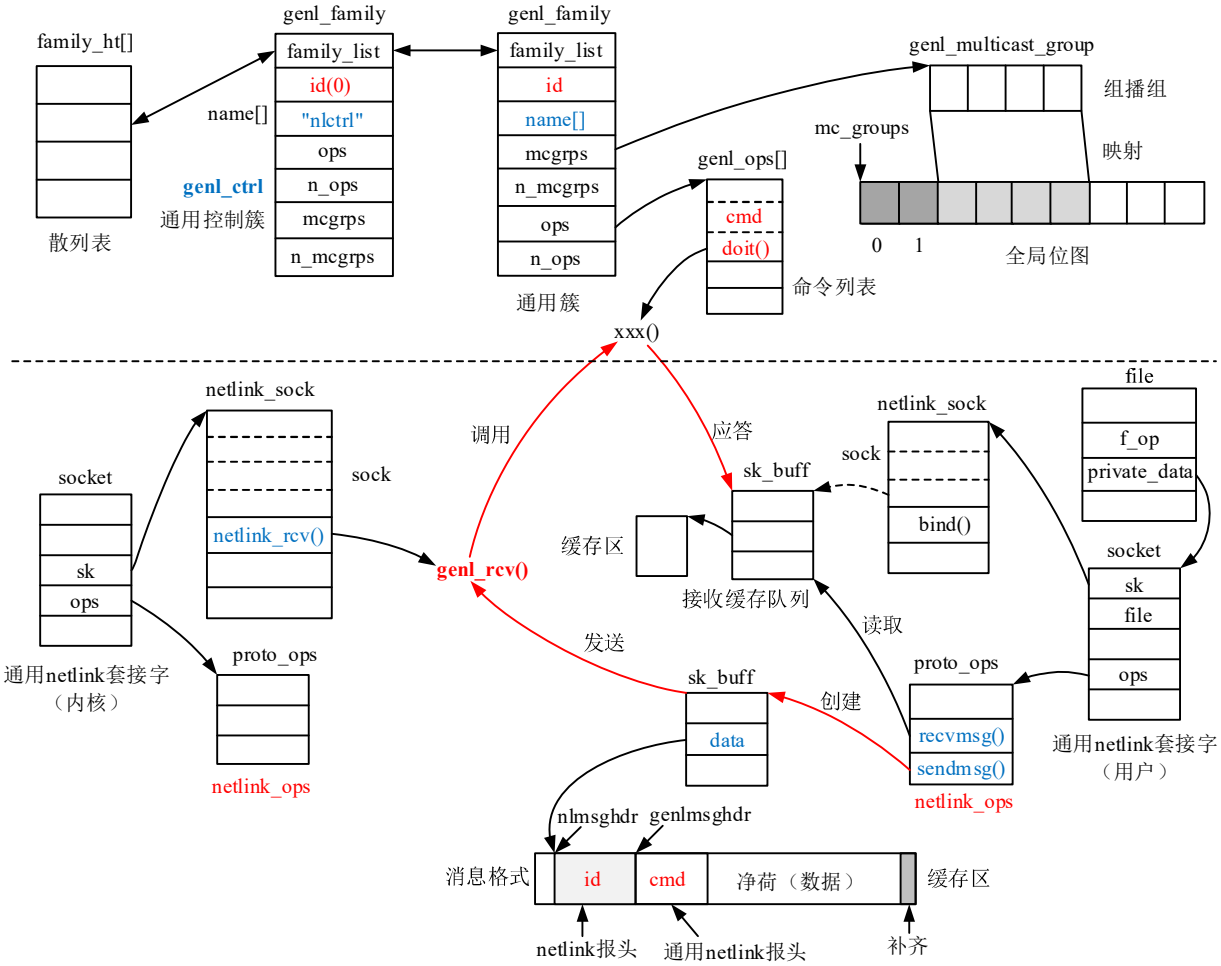
通用 netlink 套接字实现代码位于/net/netlink/genetlink.c 文件内。

1 概述

下面先介绍通用 netlink 套接字的框架及消息格式。

■套接字框架

通用 netlink 套接字框架如下图所示：



通用 netlink 套接字下管理着通用簇，通用簇由 genl_family 结构体表示，实例由全局散列表 family_ht[]管理。通用簇由名称和 id 值标识，其中包含命令（含命令处理函数）列表和组播组列表等。内核中使用通用 netlink 套接字的子系统需要定义并注册通用簇 genl_family 实例（含命令列表和组播组列表）。子系统定义通用簇时需指明名称，并且要求是唯一的，因为用户套接字通过名称寻址通用簇。

通用 netlink 套接字初始化函数中创建了 NETLINK_GENERIC 协议类型内核套接字，并注册了通用控制簇（通用簇的一个实例），通用控制簇用于实现对所有通用簇的管理和控制等。通用控制簇 id 值固定为 0，其它通用簇 id 值通常在注册时由内核自动分配。

通用 netlink 消息中包括 netlink 消息报头、通用 netlink 消息报头和净荷。在 netlink 消息报头中包含与之通信的通用簇 id 值，在通用 netlink 消息报头中包含命令值，净荷表示实际要传输的数据或者属性等。

内核套接字接收到用户套接字消息后，调用 **genl_rcv()**函数处理消息。genl_rcv()函数从报头中提取通用簇 id 值、命令值及属性等，通过 id 值查找通用簇及其命令列表，调用相应命令的处理函数处理消息。

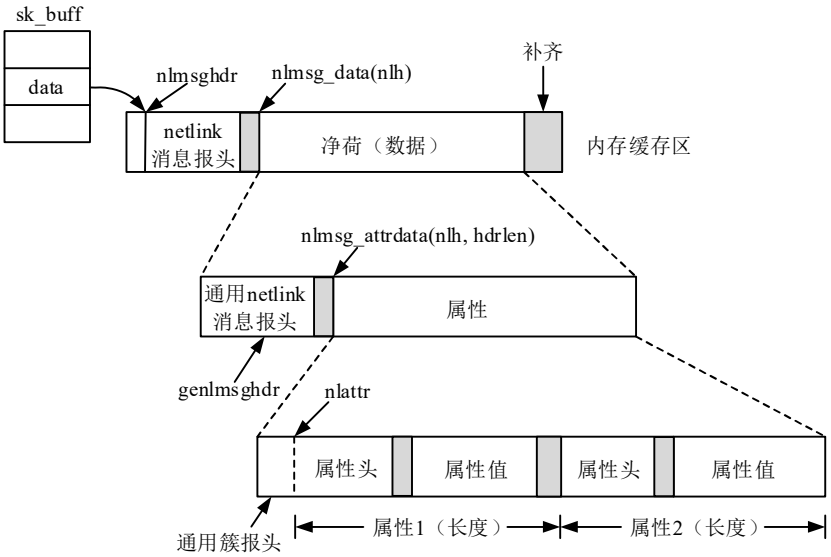
由于通用簇在注册时其 id 值是动态分配的，因此用户在与通用簇通信前需要获取其 id 值。用户套接字通过向通用控制簇（id 值固定为 0）发送 CTRL_CMD_GETFAMILY 命令消息来获取指定名称通用簇 id 值，在此消息的属性区需要指明查找通用簇的名称。CTRL_CMD_GETFAMILY 命令处理函数根据名称查找全局散列表，获取指定名称通用簇的 id 值，通过应答消息发送给用户套接字，随后用户套接字就可以向

此 id 值通用簇发送命令进行通信了。

通用 netlink 套接字也支持组播。通用簇通过名称来定义组播组（名称字符串列表）。在注册通用簇时，内核会将通用簇定义的组播组映到全局位图，并将映射的起始组播组编号赋予通用簇。这里的全局位图将与 netlink 套接字中 netlink_table.**listeners** 成员指向的全局位图同步。在注册通用簇时内核会将组播组名称和编号通过消息发送到用户空间。用户套接字可监听组播组消息以获取信息。

■消息格式

通用 netlink 套接字消息格式如下图所示：



通用 netlink 消息在 netlink 消息的数据区开头是 netlink 消息报头。通过 netlink 消息报头传递通用簇 id 值，而在通用 netlink 消息报头中传递命令值。净荷区包含传输的数据或属性，通用簇也可以添加通用簇报头，也可以不加。

如果通用簇是用属性来传递信息，则通用 netlink 消息报头后是属性区（否则直接是传输的数据）。属性区可以包含多个属性，每个属性包含属性头和属性值。

netlink 消息报头由 nlmsghdr 结构体表示，定义如下（/include/upai/linux/netlink.h）：

```
struct nlmsghdr {
    __u32       nlmsg_len;    /*加上首部的消息长度，不含补齐字节*/
    __u16       nlmsg_type;   /*消息类型，通用簇 id 值*/
    __u16       nlmsg_flags;  /*标记，取值定义在/include/upai/linux/netlink.h 头文件*/
    __u32       nlmsg_seq;    /*序列号，用于排列消息，不要求必须使用*/
    __u32       nlmsg_pid;    /*发送端口 ID，内核为 0 */
};
```

nlmsg_type 成员用于保存与之通信的通用簇 **id** 值。通用控制簇的 id 值固定为 GENL_ID_CTRL（0）。

通用 netlink 消息报头长度为 4 字节，由 genlmsghdr 结构体表示，定义如下：

```
struct genlmsghdr {      /*/include/uapi/linux/genetlink.h*/
    __u8      cmd;       /*命令值*/
    __u8      version;   /*版本号*/
    __u16     reserved;  /*保留*/
};
```

genlmsghdr 结构体成员简介如下：

●**cmd**：通用 netlink 命令值，每个注册的通用簇都将定义自己的命令。

●**version**：可用于提供版本控制支持，作用是在不破坏向后兼容性的情况下修改消息的格式。

●**reserved**：保留，供以后使用。

2 通用簇

通用簇由 genl_family 结构体表示，实例由全局散列表管理，通用簇中包含命令列表和组播组列表等信息。使用通用 netlink 套接字的内核子系统需要定义并注册通用簇 genl_family 实例。

■数据结构

通用簇 genl_family 结构体定义如下（/include/net/genetlink.h）：

```
struct genl_family {
    unsigned int        id;     /*通用簇标识，必须是全局唯一的，通常由内核分配*/
    unsigned int        hdrsize;    /*通用簇报头长度，字节数*/
    char            name[GENL_NAMSIZ];      /*名称，必须是全局唯一的*/
    unsigned int        version;    /*版本号*/
    unsigned int        maxattr;    /*支持的最大属性数量*/
    bool            netnsok;        /*是否支持网络命名空间*/
    bool            parallel_ops;   /*操作函数只能并行调用，不能同步调用*/
    int         (*pre_doit)(const struct genl_ops *ops,struct sk_buff *skb,struct genl_info *info);
                                                        /*处理命令前调用的函数*/
    void        (*post_doit)(const struct genl_ops *ops,struct sk_buff *skb,struct genl_info *info);
                                                        /*处理命令后调用的函数*/
    int         (*mcast_bind)(struct net *net, int group);      /*绑定组播组*/
    void        (*mcast_unbind)(struct net *net, int group);    /*解绑组播组*/
    struct nlattr ** attrbuf;           /*指向属性头指针数组，指向属性区中的属性头*/
    const struct genl_ops *   ops;      /*指向命令列表*/
    const struct genl_multicast_group *mcgrps;      /*组播组列表，与组播组位图对应*/
    unsigned int        n_ops;          /*命令列表数量*/
    unsigned int        n_mcgrps;   /*组播组数量*/
    unsigned int        mcgrp_offset;  /*组播组在全局位图中的起始偏移量*/
    struct list_head    family_list;    /*双链表成员，将实例链接到全局散列链表*/
    struct module       *module;
};
```

genl_family 结构体主要成员简介如下：

●**id**：通用簇 id 值，定义 genl_family 实例时通常赋予 GENL_ID_GENERATE，在注册实例时，由内核自动分配 id 值，并确保是全局唯一的。

●**hdrsize**：通用簇报头长度，通用簇还可以定义私有的报头，位于净荷区开头，也可以不定义。

●**name[GENL_NAMSIZ]**：通用簇名称，必须是全局唯一的。

●**ops**：指向 genl_ops 结构体数组，每个 genl_ops 结构体对应一个命令，包含命令值和处理函数等信息，genl_ops 结构体定义如下（/include/net/genetlink.h）：

```
struct genl_ops {
    const struct nla_policy   *policy;      /*属性有效性策略*/
    int   (*doit)(struct sk_buff *skb,struct genl_info *info);      /*处理命令的回调函数*/
    int   (*dumpit)(struct sk_buff *skb,struct netlink_callback *cb);   /*转储回调函数*/
    int   (*done)(struct netlink_callback *cb);     /*转储结束后执行的回调函数*/
```

```
    u8    cmd;              /*命令值*/
    u8    internal_flags;   /*通用簇定义和使用的私有标志*/
    u8    flags;            /*标记*/
};
```

genl_ops 结构体 policy 成员指向 nla_policy 结构体（数组），结构体定义在/include/net/netlink.h 头文件内：

```
struct nla_policy {
    u16    type;
    u16    len;
};
```

nla_policy 实例用于检查属性区传递属性的有效性。

genl_ops 结构体标记 flags 成员取值定义在/include/uapi/linux/genetlink.h：

```
#define  GENL_ADMIN_PERM        0x01    /*执行命令需要超级用户权限*/
#define  GENL_CMD_CAP_DO        0x02
#define  GENL_CMD_CAP_DUMP      0x04
#define  GENL_CMD_CAP_HASPOL    0x08
```

●**n_ops**：命令列表项数。

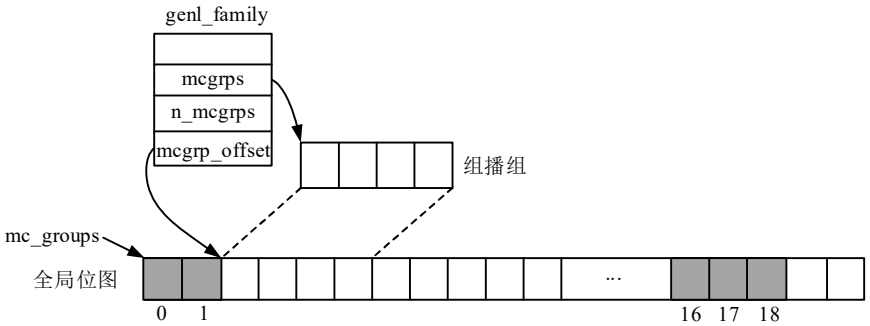
●**attrbuf**：指向 nlattr 结构体指针数组，数组项指向属性区的属性头。每个通用簇可以定义一组属性（用整数标识），每个属性值的类型可以指定为字符串、整型数等。在接收消息时将为通用簇分配属性头指针数组，用于关联消息中包含的属性头。

●**mcgrps**：指向 genl_multicast_group 结构体数组，表示组播组，结构体定义如下：

```
struct genl_multicast_group {
    char      name[GENL_NAMSIZ];    /*组播组名称*/
};
```

●**n_mcgrps**：组播组项数。

通用 netlink 套接字定义了全局位图用于映射所有通用簇实例定义的组播组，如下图所示：



在注册通用簇时，内核会将其组播组映射到全局位图，mcgrp_offset 表示在全局位图中的起始偏移量，由此可确定通用簇每个组播组的编号。通用簇通过名称来定义组播组，在注册通用簇时会向用户空间发送新组播组名称和编号的消息。全局位图中 bit0，bit1，bit16、bit17、bit18 已被内核占用。

内核定义了全局散列表用于管理 genl_family 实例：

```
static struct list_head family_ht[GENL_FAM_TAB_SIZE];   /*双链表数组项数为 16*/
```

散列函数比较简单，根据 genl_family 实例 id 成员低 4 值确定添加到的散列链表项索引值。

■初始化

通用 netlink 套接字初始化函数为 genl_init()，代码简列如下：

```
static int __init genl_init(void)
{
    int i, err;

    for (i = 0; i < GENL_FAM_TAB_SIZE; i++)     /*初始化通用簇散列表（16 项）*/
        INIT_LIST_HEAD(&family_ht[i]);

    err = genl_register_family_with_ops_groups(&genl_ctrl, genl_ctrl_ops,genl_ctrl_groups);
                                                    /*注册通用控制簇 genl_ctrl 实例，见下文*/
    ...
    err = register_pernet_subsys(&genl_pernet_ops);     /*注册 genl_pernet_ops 实例*/
    ...
    return 0;
    ...
}
subsys_initcall(genl_init);     /*内核初始化阶段调用 genl_init()函数*/
```

初始化函数中将初始化通用簇散列表，注册通用控制簇 **genl_ctrl** 实例，注册 pernet_operations 结构体实例 genl_pernet_ops，用于在网络命名空间中完成通用 netlink 套接字相关的初始化工作。

genl_pernet_ops 实例定义如下：

```
static struct pernet_operations genl_pernet_ops = {
    .init = genl_pernet_init,       /*初始化*/
    .exit = genl_pernet_exit,
};
```

初始化函数 genl_pernet_init()内主要是创建内核通用 netlink 套接字，函数定义如下：

```
static int __net_init genl_pernet_init(struct net *net)
{
    struct netlink_kernel_cfg cfg = {
        .input          = genl_rcv,   /*接收用户套接字消息的函数*/
        .flags          = NL_CFG_F_NONROOT_RECV,    /*非超级用户可接收组播消息*/
        .bind           = genl_bind,
        .unbind         = genl_unbind,
    };

    /*创建内核通用 netlink 套接字，每个网络命名空间对应一个内核 netlink 套接字实例*/
    net->genl_sock = netlink_kernel_create(net, NETLINK_GENERIC, &cfg);
                                /*网络命名空间 net 实例中 genl_sock 成员指向套接字 sock 实例*/
    ...
    return 0;
}
```

■注册通用簇

注册通用簇 genl_family 实例的接口函数定义如下（/include/net/genetlink.h）：

```
#define   genl_register_family_with_ops(family, ops)          \
      _genl_register_family_with_ops_grps((family),          \
                                  (ops), ARRAY_SIZE(ops),   \
                                  NULL, 0)
```

_genl_register_family_with_ops_grps()函数定义在/include/net/genetlink.h 头文件：

```
static inline int _genl_register_family_with_ops_grps(struct genl_family *family,
                              const struct genl_ops *ops, size_t n_ops,
                              const struct genl_multicast_group *mcgrps,size_t n_mcgrps)
/*
*family：指向 genl_family 实例，ops：指向 genl_ops 数组（命令列表），n_ops：genl_ops 数组项数，
*mcgrps：指向组播组列表，n_mcgrps：组播组数量。
*/
{
      family->module = THIS_MODULE;
      family->ops = ops;     /*指向 genl_ops 数组*/
      family->n_ops = n_ops;     /*genl_ops 数组项数*/
      family->mcgrps = mcgrps;     /*指向组播组*/
      family->n_mcgrps = n_mcgrps;     /*组播组数量*/
      return __genl_register_family(family);     /*/net/netlink/genetlink.c*/
}
```

__genl_register_family()函数定义如下：

```
int __genl_register_family(struct genl_family *family)
{
      int err = -EINVAL, i;

      if (family->id && family->id < GENL_MIN_ID)
            goto errout;

      if (family->id > GENL_MAX_ID)
            goto errout;

      err = genl_validate_ops(family);       /*检查 genl_ops 实例数组的有效性*/
      ...

      genl_lock_all();

      if (genl_family_find_byname(family->name)) {
            ...              /*在全局散列表中按名称查找通用簇实例，检查是否已存在同名实例*/
      }

      if (family->id == GENL_ID_GENERATE) {     /*id 值设为 GENL_ID_GENERATE，自动生成 id*/
```

```
        u16 newid = genl_generate_id();
        ...
        family->id = newid;
    } else if (genl_family_find_byid(family->id)) {   /*若指定了 id，查找是否已存在相同 id 的实例*/
        ...
    }

    if (family->maxattr && !family->parallel_ops) {     /*分配属性头指针数组*/
        family->attrbuf = kmalloc((family->maxattr+1) *sizeof(struct nlattr *), GFP_KERNEL);
        ...
    } else
        family->attrbuf = NULL;

    err = genl_validate_assign_mc_groups(family);
                /*将通用簇组播组映射到全局位图，同步 netlink_table 实例中全局监听位图等*/
    ...

    list_add_tail(&family->family_list, genl_family_chain(family->id)); /*将实例添加到全局散列表*/
    genl_unlock_all();

    /*向用户空间发送事件消息*/
    genl_ctrl_event(CTRL_CMD_NEWFAMILY, family, NULL, 0);     /*通过通用控制簇组播事件*/
    for (i = 0; i < family->n_mcgrps; i++)
        genl_ctrl_event(CTRL_CMD_NEWMCAST_GRP, family,
                &family->mcgrps[i], family->mcgrp_offset + i);
                        /*通过通用簇组播通用簇组播组的名称和在全局位图中的偏移量*/
    return 0;
    ...
}
```

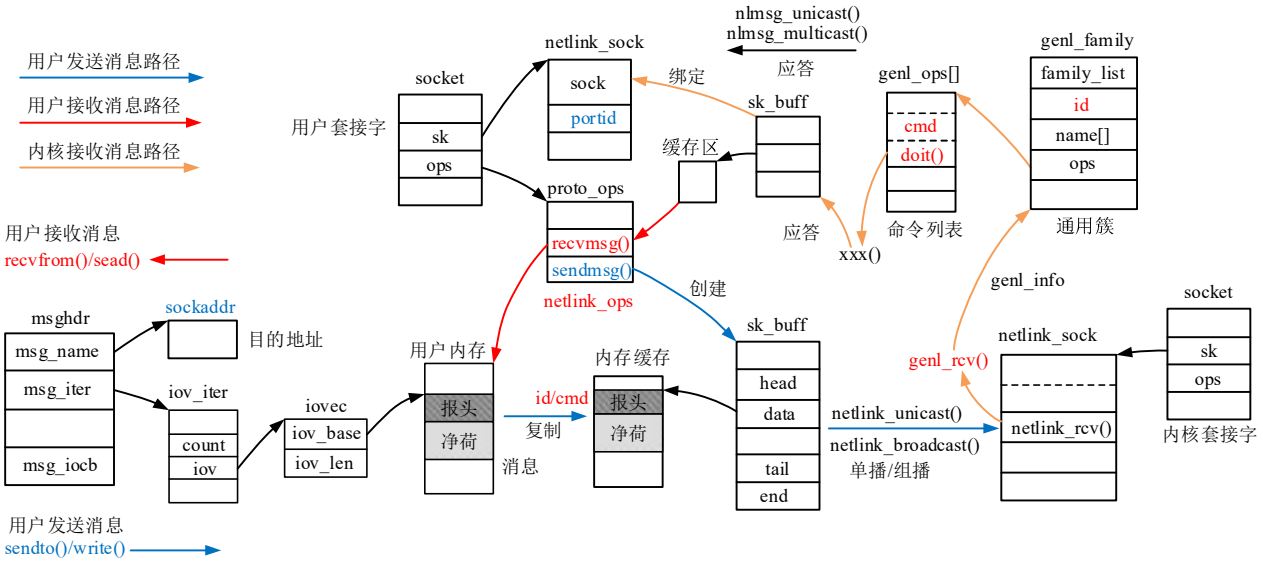
注册函数先检查 genl_family 实例、命令列表的有效性，如果实例 id 值为 GENL_ID_GENERATE，则为其动态生成 id 值，映射通用簇中组播组到全局位图，然后将实例添加到全局散列表，最后向用户空间发送事件消息，消息中包含通用簇名称，组播组编号等信息。

3 发送与接收消息

下面从用户套接字的角度介绍一下发送和接收消息的流程。用户套接字与内核套接字通信时，用户通过系统调用发送消息，内核套接字收到消息后提取 id 值和命令值后，查找通用簇并调用通用簇中命令处理函数，处理函数对消息进行处理并根据需要向用户套接字发送应答消息。

■用户发送接收消息

用户发送消息的流程如下图所示：



用户进程发送消息时，需要构建 netlink 消息并存入用户内存，并依此构建 msghdr 实例（或在系统调用内构建），其关联 sockaddr 实例在发送消息时表示目的套接字地址，在接收消息时保存发送套接字地址。

发送消息系统调用内将调用 netlink 套接字 proto_ops 实例中的 sendmsg()函数，即 netlink_sendmsg()函数，此函数内将构建 sk_buff 实例，复制用户空间消息至 sk_buff 实例内存缓存区，调用 netlink_unicast()和 netlink_broadcast()接口函数单播/组播消息。

用户向内核套接字发送消息时，netlink_unicast()函数将调用内核 netlink 套接字定义的 netlink_rcv()函数接收 sk_buff 实例，对于通用 netlink 套接字此函数为 **genl_rcv()**（见上文中的初始化函数）。genl_rcv()函数从消息报头中提取通用簇 id 值和命令值，由 id 值查找通用簇实例，最后调用通用簇中相应命令的处理函数，由处理函数处理消息并按需发送应答消息。

命令处理函数中通常调用 genlmsg_unicast()/genlmsg_multicast()等接口函数向用户套接字发送应答消息，这些接口函数最终调用 netlink 套接字的 netlink_unicast()/netlink_broadcast()接口函数完成消息的发送。

用户套接字读取消息时，从接收缓存队列 sk_buff 实例中复制消息数据到用户内存。

■内核接收消息

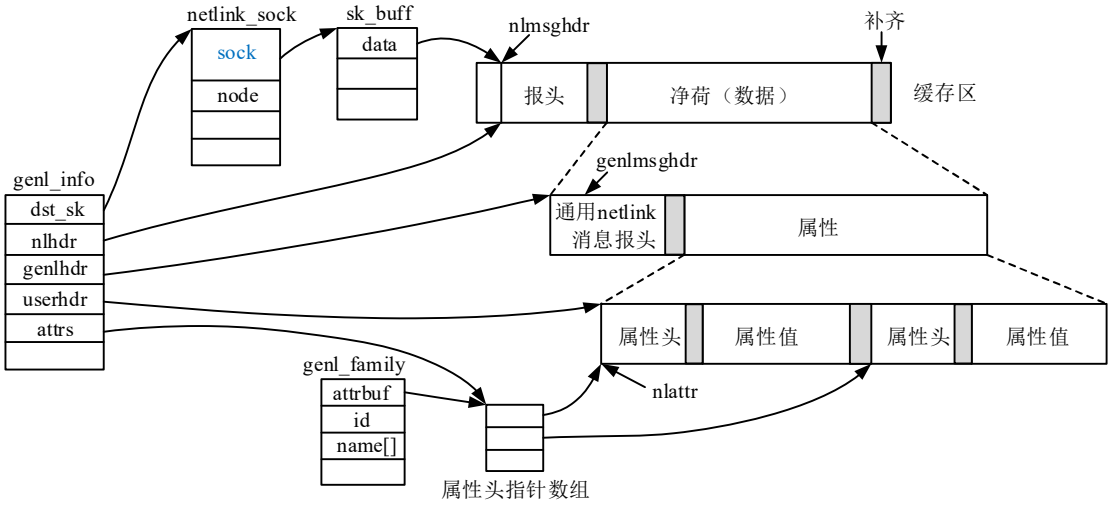
用户 netlink 套接字接收/发送消息的函数前面已经介绍过了，这里不再重复了。下面主要介绍一下内核套接字接收用户消息的处理函数。

内核通用 netlink 套接字接收消息的函数为 **genl_rcv()**，函数内利用 genl_info 结构体实例暂存消息处理过程中的相关信息，随后将结构体实例指针作为调用命令处理函数的参数，执行通用簇中定义的命令处理函数。

genl_info 结构体定义如下（/include/net/genetlink.h）：

```
struct genl_info {
    u32     snd_seq;        /*消息序列号*/
    u32     snd_portid;     /*发送消息套接字 id（用户套接字，在应答消息时作为目的套接字 id）*/
    struct nlmsghdr *       nlhdr;      /*指向 netlink 消息报头*/
    struct genlmsghdr *genlhdr;         /*指向通用 netlink 消息报头*/
    void *                  userhdr;    /*指向数据区*/
    struct nlattr **        attrs;      /*指向属性头指针数组*/
    possible_net_t          _net;
    void *                  user_ptr[2];
    struct sock *           dst_sk;     /*用户套接字（发送套接字）关联 sock 实例*/
};
```

genl_rcv()函数内将查找用户欲通信的通用簇实例，解析消息数据，并利用 genl_info 实例暂存这些信息，如下图所示：



genl_rcv()函数定义如下：

```
static void genl_rcv(struct sk_buff *skb)
{
    down_read(&cb_lock);
    netlink_rcv_skb(skb, &genl_rcv_msg);
    up_read(&cb_lock);
}
```

接收消息的 genl_rcv()函数中调用接口函数 netlink_rcv_skb()对消息进行处理，如果接收到的 netlink 消息设置了 NLM_F_REQUEST 标记，将进而调用 **genl_rcv_msg()**函数处理消息，函数定义如下：

```
static int genl_rcv_msg(struct sk_buff *skb, struct nlmsghdr *nlh)
/*nlh：指向内存缓存区 netlink 消息报头*/
{
    struct genl_family *family;
    int err;

    family = genl_family_find_byid(nlh->nlmsg_type);    /*nlh->nlmsg_type 保存通用簇 id 值*/
                                                        /*由 id 查找通用簇 genl_family 实例*/
    ...
    err = genl_family_rcv_msg(family, skb, nlh);       /*由查找到的通用簇处理命令*/
    ...
    return err;
}
```

genl_family_rcv_msg()函数中将消息交由查找到的通用簇处理，函数定义如下：

```
static int genl_family_rcv_msg(struct genl_family *family, struct sk_buff *skb, struct nlmsghdr *nlh)
/*nlh：指向 netlink 消息报头，非通用 netlink 套接字报头*/
{
    const struct genl_ops *ops;
    struct net *net = sock_net(skb->sk);     /*网络命名空间*/
    struct genl_info info;       /*genl_info 实例*/
    struct genlmsghdr *hdr = nlmsg_data(nlh);     /*指向通用 netlink 消息报头*/
```

```
struct nlattr **attrbuf;       /*指向属性头指针数组*/
int hdrlen, err;

/*如果通用簇不在此网络命名空间*/
if (!family->netnsok && !net_eq(net, &init_net))
        return -ENOENT;

hdrlen = GENL_HDRLEN + family->hdrsize;     /*通用 netlink 消息报头加通用簇报头长度*/
if (nlh->nlmsg_len < nlmsg_msg_size(hdrlen))
        return -EINVAL;

ops = genl_get_cmd(hdr->cmd, family);
                        /*从通用 netlink 消息报头提取命令值，在通用簇中查找 genl_ops 实例*/
...     /*ops 为 NULL，返回错误码*/

if ((ops->flags & GENL_ADMIN_PERM) &&!netlink_capable(skb, CAP_NET_ADMIN))
        return -EPERM;                                          /*权限检查*/

if ((nlh->nlmsg_flags & NLM_F_DUMP) == NLM_F_DUMP) {
                                /*处理 netlink 消息报头设置了 NLM_F_DUMP 标记位的情况*/
        ...
}

if (ops->doit == NULL)
        return -EOPNOTSUPP;

if (family->maxattr && family->parallel_ops) {       /*分配属性头指针数组*/
        attrbuf = kmalloc((family->maxattr+1) *sizeof(struct nlattr *), GFP_KERNEL);
        ...
} else
        attrbuf = family->attrbuf;

if (attrbuf) {
        err = nlmsg_parse(nlh, hdrlen, attrbuf, family->maxattr,ops->policy);
                                                        /*解析属性区属性，attrbuf 数组项指向属性头*/
        ...
}

info.snd_seq = nlh->nlmsg_seq;              /*设置 genl_info 实例*/
info.snd_portid = NETLINK_CB(skb).portid;       /*用户套接字 ID 值*/
info.nlhdr = nlh;
info.genlhdr = nlmsg_data(nlh);
info.userhdr = nlmsg_data(nlh) + GENL_HDRLEN;
info.attrs = attrbuf;
info.dst_sk = skb->sk;      /*发送消息套接字 sock 实例*/
genl_info_net_set(&info, net);
```

```
        memset(&info.user_ptr, 0, sizeof(info.user_ptr));

        if (family->pre_doit) {
            err = family->pre_doit(ops, skb, &info);     /*执行命令处理前函数*/
            ...
        }

        err = ops->doit(skb, &info);     /*调用命令处理函数*/

        if (family->post_doit)
            family->post_doit(ops, skb, &info);     /*执行命令处理后函数*/
    out:
        if (family->parallel_ops)
            kfree(attrbuf);

        return err;
    }
```

genl_family_rcv_msg()函数的主要工作是从通用 netlink 消息报头中提取命令值，解析消息中的属性，调用通用簇定义的 **genl_ops** 实例中的命令处理函数处理命令，genl_info 实例作为命令处理函数的参数。

4 通用控制簇

通用 netlink 套接字在初始化函数中注册了通用控制簇（通用簇的一个实例），用于管理和控制所有的通用簇，其 id 值固定为 0，用户套接字可直接与之通信。用户套接字在与其它通用簇通信前需要向通用控制簇发送命令，以获取欲通信通用簇的 id 值。

通用控制簇接收/发送的命令值定义如下（/include/upai/linux/netlink.h）：

```
enum {
    CTRL_CMD_UNSPEC,
    CTRL_CMD_NEWFAMILY,     /*注册了新通用簇*/
    CTRL_CMD_DELFAMILY,     /*删除了通用簇*/
    CTRL_CMD_GETFAMILY,     /*接收的命令，查找通用簇，其它为发送的命令*/
    CTRL_CMD_NEWOPS,
    CTRL_CMD_DELOPS,
    CTRL_CMD_GETOPS,
    CTRL_CMD_NEWMCAST_GRP,
    CTRL_CMD_DELMCAST_GRP,
    CTRL_CMD_GETMCAST_GRP, /* unused */
    __CTRL_CMD_MAX,         /*最大命令值*/
};
```

CTRL_CMD_GETFAMILY 命令用于执行用户套接字由名称查找通用簇 id 的操作，其它命令在有通用簇发生事件时组播消息。

■注册通用控制簇

通用 netlink 套接字初始化函数中注册了通用控制簇，代码如下：

```
static int __init genl_init(void)
```

```
{
    ...
    err = genl_register_family_with_ops_groups(&genl_ctrl, genl_ctrl_ops,genl_ctrl_groups);
                                                        /*注册通用控制簇，带组播组*/
    ...
}
```

注册函数参数中包含通用控制簇实例 genl_ctrl，命令列表 genl_ctrl_ops 和组播组 genl_ctrl_groups，下面分别对各参数做简要介绍。

通用控制簇 genl_family 实例为 genl_ctrl，定义如下：

```
static struct genl_family genl_ctrl = {
    .id = GENL_ID_CTRL,      /*id 值为 0，固定值*/
    .name = "nlctrl",        /*名称*/
    .version = 0x2,
    .maxattr = CTRL_ATTR_MAX,   /*最大属性数量*/
    .netnsok = true,     /*支持网络命名空间*/
};
```

通用控制簇消息中属性类型定义如下（/include/uapi/linux/genetlink.h）：

```
enum {
    CTRL_ATTR_UNSPEC,
    CTRL_ATTR_FAMILY_ID,        /*id 值属性*/
    CTRL_ATTR_FAMILY_NAME,   /*名称属性*/
    CTRL_ATTR_VERSION,
    CTRL_ATTR_HDRSIZE,          /*通用簇报头长度*/
    CTRL_ATTR_MAXATTR,
    CTRL_ATTR_OPS,
    CTRL_ATTR_MCAST_GROUPS,
    __CTRL_ATTR_MAX,
};

#define CTRL_ATTR_MAX    (__CTRL_ATTR_MAX - 1)
```

通用控制簇命令列表 genl_ctrl_ops[]实例定义如下，列表中只有一个命令：

```
static struct genl_ops genl_ctrl_ops[] = {
    {
        .cmd      = CTRL_CMD_GETFAMILY,      /*查找指定通用簇 id 值命令*/
        .doit     = ctrl_getfamily,      /*命令处理函数，查找通用簇，发送其 id 值给用户套接字*/
        .dumpit   = ctrl_dumpfamily,
        .policy   = ctrl_policy,
    },
};
```

通用控制簇只处理用户套接字发送的 CTRL_CMD_GETFAMILY 命令。

通用控制簇组播组 genl_ctrl_groups[]实例定义如下：

```
static struct genl_multicast_group genl_ctrl_groups[] = {       /*组播组名称*/
    { .name = "notify", },     /*用户可监听通用控制簇的此组播组获取所有通用簇事件消息*/
```

```
};
```

■查找通用簇

通用控制簇命令列表中只有一项，即 CTRL_CMD_GETFAMILY 命令项。用户进程与某一通用簇通信前，需要构建 CTRL_CMD_GETFAMILY 命令的消息，消息数据区需包含 **CTRL_ATTR_FAMILY_NAME** 属性，属性值为通用簇的名称，发往 id 值为 0 的通用控制簇。通用控制簇的应答消息中将包括指定通用簇的 id 值等信息，供用户读取。

内核套接字收到此消息后将调用通用控制簇中 CTRL_CMD_GETFAMILY 命令的 **ctrl_getfamily()**函数处理命令，函数定义如下。

```
static int ctrl_getfamily(struct sk_buff *skb, struct genl_info *info)
{
    struct sk_buff *msg;
    struct genl_family *res = NULL;
    int err = -EINVAL;
    /*处理由 id 查找通用簇的情况*/
    if (info->attrs[CTRL_ATTR_FAMILY_ID]) {     /*从 CTRL_ATTR_FAMILY_ID 属性获取 id 值*/
        u16 id = nla_get_u16(info->attrs[CTRL_ATTR_FAMILY_ID]);
        res = genl_family_find_byid(id);       /*由 id 值查找通用簇*/
        err = -ENOENT;
    }
    /*处理由名称查找通用簇的情况*/
    if (info->attrs[CTRL_ATTR_FAMILY_NAME]) {       /*通用簇名称属性*/
        char *name;

        name = nla_data(info->attrs[CTRL_ATTR_FAMILY_NAME]);     /*属性值，即通用簇名称*/
        res = genl_family_find_byname(name);     /*由名称查找通用簇*/
#ifdef CONFIG_MODULES
        if (res == NULL) {          /*加载模块*/
            genl_unlock();
            up_read(&cb_lock);
            request_module("net-pf-%d-proto-%d-family-%s",
                           PF_NETLINK, NETLINK_GENERIC, name);
            down_read(&cb_lock);
            genl_lock();
            res = genl_family_find_byname(name);
        }
#endif
        err = -ENOENT;
    }

    if (res == NULL)
        return err;

    if (!res->netnsok && !net_eq(genl_info_net(info), &init_net)) {
        ...
```

```
    }

    msg = ctrl_build_family_msg(res, info->snd_portid, info->snd_seq,CTRL_CMD_NEWFAMILY);
                                        /*构建 sk_buff 实例，info->snd_portid 为用户套接字 ID*/
    ...
    return genlmsg_reply(msg, info);      /*调用接口函数单播 sk_buff 实例，/include/net/genetlink.h*/
}
```

ctrl_getfamily()函数从消息属性区提取通用簇名称，在散列表查找通用簇，调用 ctrl_build_family_msg()函数构建 sk_buff 实例，将应答消息写入 sk_buff 实例关联的内存缓存区中（消息数据中包含通用簇许多的信息，不光只有 id 值），调用 genlmsg_reply()函数将消息发送给用户套接字（调用单播消息 nlmsg_unicast()函数）。

■发送控制事件

在注册通用簇的接口函数中，函数最后将调用 **genl_ctrl_event()**函数向通用控制簇组播组 1 组播消息，函数定义如下：

```
static int genl_ctrl_event(int event, struct genl_family *family,const struct genl_multicast_group *grp,
                                                                                    int grp_id)
/*
*event：事件类型（通用控制簇命令类型），family：指向产生事件的通用簇实例，
*grp：指向通用簇定义的组播组（组播组数组项），grp_id：通用簇定义组播组在全局位图中的编号。
*以上参数都是用来构建组播消息的，而不是将消息发送参数指定的组播组。
*/
{
    struct sk_buff *msg;      /*构建的 sk_buff 实例*/

    /* genl is still initialising */
    if (!init_net.genl_sock)
        return 0;

    switch (event) {        /*构建事件消息*/
    case CTRL_CMD_NEWFAMILY:
    case CTRL_CMD_DELFAMILY:
        WARN_ON(grp);
        msg = ctrl_build_family_msg(family, 0, 0, event);      /*增加/删除通用簇的消息*/
        break;
    case CTRL_CMD_NEWMCAST_GRP:
    case CTRL_CMD_DELMCAST_GRP:
        BUG_ON(!grp);
        msg = ctrl_build_mcgrp_msg(family, grp, grp_id, 0, 0, event);
        break;
    default:
        return -EINVAL;
    }
    ...
    /*将事件消发送到通用控制簇组播组 1，即通用控制簇"notify"*/
```

```
    if (!family->netnsok) {     /*通用簇不支持网络命名空间*/
        genlmsg_multicast_netns(&genl_ctrl, &init_net, msg, 0,0, GFP_KERNEL);
                                /*发送组播消息，/include/net/genetlink.h*/
    } else {      /*通用簇支持网络命名空间*/
        rcu_read_lock();
        genlmsg_multicast_allns(&genl_ctrl, msg, 0,0, GFP_ATOMIC);
                        /*在所有网络命名空间组播消息，/net/netlink/genetlink.c*/
        rcu_read_unlock();
    }

    return 0;
}
```

genl_ctrl_event()函数在注册通用簇时被调用，主要工作是向通用控制簇组播组 1 发送通用簇事件消息。在注册通用簇时，将组播增加新通用簇的消息，消息内容中包含通用簇的信息，详见 ctrl_build_family_msg()函数，还将组播注册通用簇定义的组播组名称，以及其在全局位图中的位置，即组播组编号。genl_ctrl_event()函数一次只能发送通用簇定义的一个组播组的名称和编号，如果定义了多个组，需要多次调用此函数。

用户套接字在向通用控制簇发送 CTRL_CMD_GETFAMILY 命令查询通用簇 id 值时，其内核套接字应答消息中也包含通用簇定义的组播组名称和编号信息。用户套接字可以监听通用控制簇组播组 1 的消息以获取通用簇事件消息。

通用控制簇应答消息中的属性标识定义在/include/uapi/linux/genetlink.h 头文件，ctrl_fill_info()函数会将各属性值填充到应答消息中，用户套接字可从中获取通用簇的信息，请读者自行阅读源代码。

12.3.8 小结

netlink 套接字主要用于内核与主机用户进程之间的通信，数据以 sk_buff 实例的形式在主机内存中传输，传输协议较简单。netlink 套接字传输的数据由消息组成，消息由报头和净荷（数据）组成，净荷通常通过属性来传递信息。

netlink 套接字被内核各子系统广泛使用，netlink 套接字最多可定义 32 个协议类型，每个协议类型适用于某个子系统（应用）。同一协议类型的 netlink 套接字由同一个散列表管理。不同协议类型的 netlink 套接字最大的区别在于消息数据区格式的不同，以及内核套接字接收消息的 netlink_rcv()函数不同。各子系统在创建协议类型对应的内核 netlink 套接字时，需要指明接收用户消息的 netlink_rcv()函数等信息。

用户套接字向内核套接字传递消息时，将调用内核套接字定义的 netlink_rcv()函数接收、处理消息，如果需要应答，则此函数内向用户套接字发送应答消息。含应答消息的 sk_buff 实例将添加到用户套接字接收缓存队列，等待用户进程读取。netlink 套接字还支持组播功能，用户套接字可加入内核套接字定义的组播组，接收内核套接字发送的组播消息。

本节介绍了协议类型为 NETLINK_KOBJECT_UEVENT、NETLINK_ROUTE、NETLINK_NETFILTER、NETLINK_GENERIC 等的 netlink 套接字实现。

NETLINK_KOBJECT_UEVENT 套接字主要用于通用驱动模型向用户空间发送设备、驱动事件消息。

NETLINK_ROUTE 套接字主要用于网络子系统设置/获取网络、网络设备参数等。

NETLINK_NETFILTER 套接字用于 netfilter 子系统。

NETLINK_GENERIC 套接字用于弥补 netlink 套接字 32 个协议类型的不足，此协议类型中管理着多个通用簇，每个通用簇可用于一个子系统。通用簇通过名称和 id 值标识，通用簇中包含命令列表（含命令标识和命令处理函数等）。用户套接字通过向指定 id 值的通用簇发送命令及数据的方式调用通用簇的命令处理函数，实现与内核子系统的交互。

netlink 其它协议类型套接字的实现请读者自行阅读源代码。

12.4 因特网及分层协议

因特网（Internet）是计算机网络的一个实例，是应用最广泛的计算机网络，它将分布于世界各地的数以亿计的计算机连接在一起，以实现相互之间通信。计算机按照网络协议的要求将数据拆分成一个个段的数据包（各层协议中称呼不同，如报文段、分组、帧等）发送到网络上，数据包在网络中接力传输最终到达目的计算机，目的计算机接收、处理数据包。

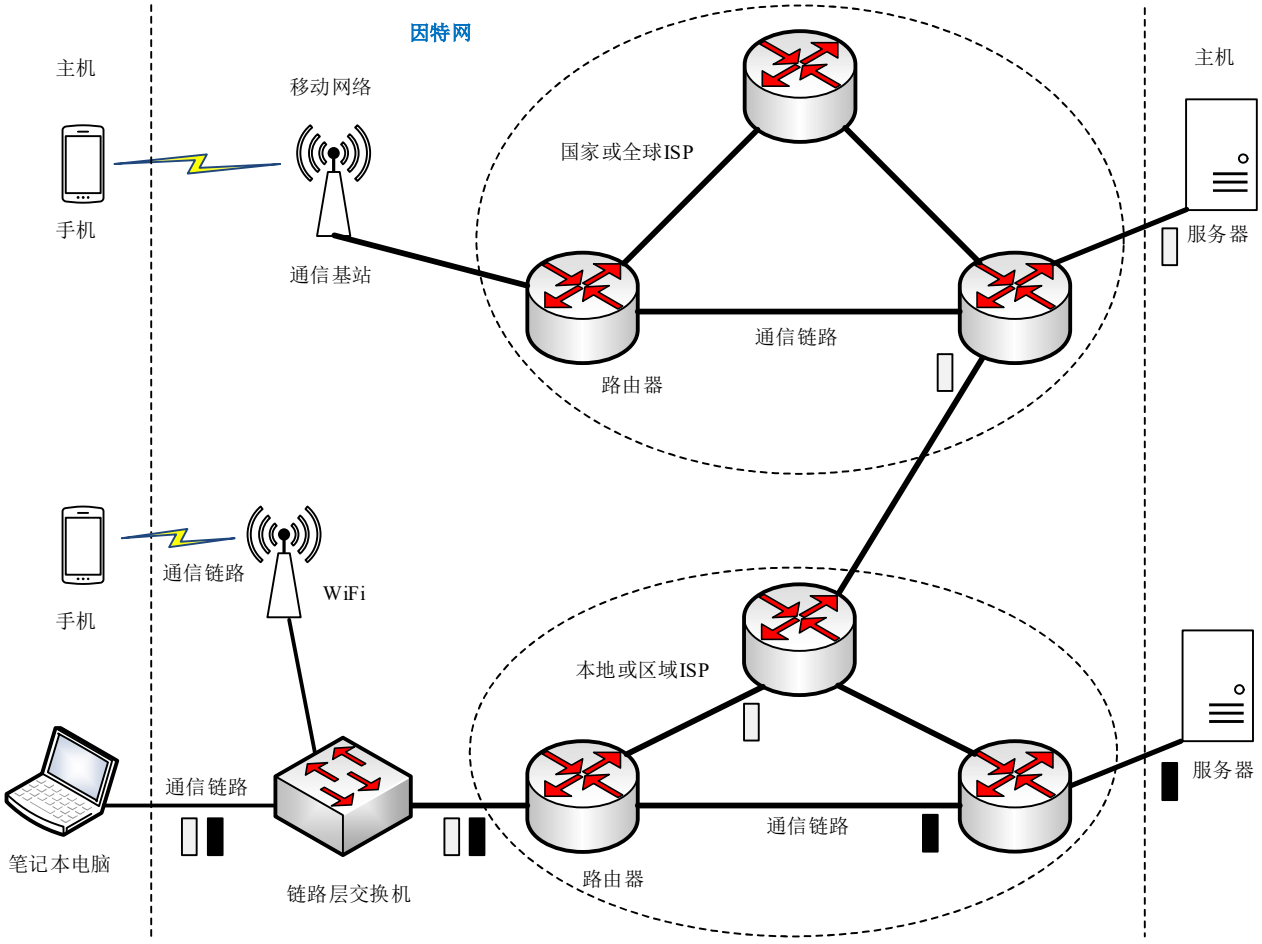
因特网传输协议采用了分层的结构，分别是应用层、传输层、网络层、数据链路层和物理层，每一层都有自身的协议类型。因特网传输协议称为 TCP/IP 协议簇，它是各层协议的集合。协议簇名称中取了传输层典型的 TCP 协议和网络层 IP 协议名称的组合。

TCP/IP 协议簇目前主要有两个版本，分别是 IPv4 和 IPv6（主要是网络层协议不同）。本章和下一章主要讲解 IPv4 协议的主要内容和在 Linux 内核中的实现，并简要介绍 IPv6 协议的内容。

本节简要介绍因特网的物理结构、网络协议分层，以及各层协议的主要内容（以 IPv4 为例），下一节开始将介绍各层网络协议在 Linux 内核中的实现。

12.4.1 物理结构

下图简要示意了因特网的物理结构：



因特网主要是由分组交换机和通信链路组成的网络，分组交换机通过通信链路连接在一起。当今因特网中，**分组交换机**最著名的两种类型是**链路层交换机**和**路由器**。分组交换机通常有多个入端口和多个出端口，任何一个入端口接收的数据包可通过任何一个出端口转发出去（路由选择），以此组成一个网络。分组交换机的主要工作就是转发数据包。数据包在网络中由分组交换机接力传输，最终到达目的主机。

因特网中的通信链路和分组交换机由因特网服务提供商（ISP）提供，每个 ISP 包含多台分组交换机和多段通信链路。各 ISP 在网络中组成层次结构，较低层的 ISP（本地或地区性的）通过国家的、较高层的 ISP 互联起来，较高层 ISP 又与其它高层的（国家级或全球级的）ISP 互联，如此使因特网成为一个全球的

计算机网络。

所有连接到因特网的设备称为主机或端系统，包括手机、电脑、服务器、数据中心等。主机划分成两类：客户和服务器，客户通常是手机、电脑、各种智能设备等，服务器通常是更强大的机器，用于存储 Web 页面、流视频、电子邮件等，现在这些服务器都属于大型数据中心。

因特网中的各机器，如主机、链路层交换机、路由器等，本书称之为网络节点，数据包在节点之间接力传输，传输路径中某一节点的下一个节点称之为下一跳。

主机通过通信链路与位于网络边缘的边缘路由器相连，它是主机到任何其它端系统路径上的第一台路由器。主机发送数据时，数据包被发送到边缘路由器，随后数据包通过通信链路在路由器之间传输，最终达到目的主机。

因特网中每台主机以及路由器的每个端口被赋予一个 IP 地址（也可以赋予多个 IP 地址），通过 IP 地址来标识主机和路由器端口。从发送主机到接收主机，一个数据包所经历的一系列通信链路和分组交换机称为通过的网络路径（route 或 path）。主机发送数据包时，需要在数据包中附上源主机和目的主机的 IP 地址，路由器通过目的 IP 地址来确定通过哪个输出端口转发数据包，各路由器选择最优的路径来转发数据包。

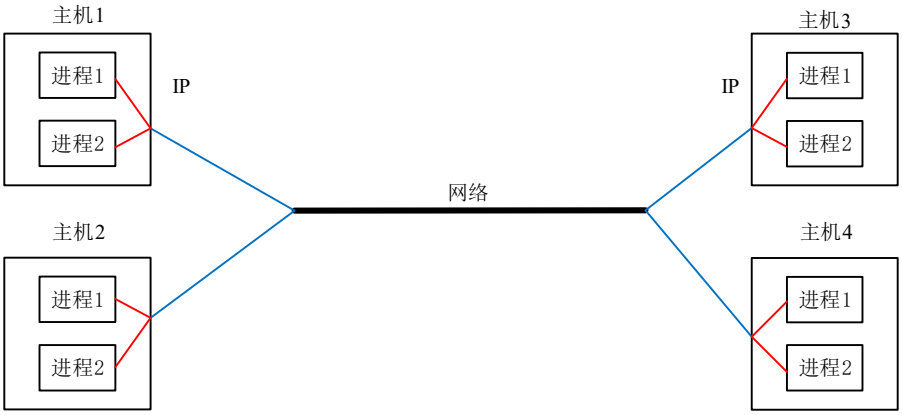
链路层交换机通常用于接入网中（局域网，如以太网、WiFi 等），交换机连接的主机之间可以通过物理地址（如 MAC 地址，而不是因特网中的 IP 地址）直接交换数据，即实现在局域网内交换数据。当需要将数据发送到外网时，数据包通过链路层交换机发送到与之相连的边缘路由器，由其向外网发送，接收数据时沿相反的路径进行。

12.4.2 分层协议模型

计算机网络各节点之间的数据传输协议非常复杂，因此网络协议采取了分层结构的设计。国际标准化组织（ISO）设计了一种参考模型，将计算机网络协议分为七层，称为 OSI（开放系统互连）模型，如下图所示。

OSI 模型中将网络协议划分成应用层、表示层、会话层、传输层、网络层、数据链路层和物理层。但是，七层的划分过于详细，实际上 Linux 内核将七层的网络协议进行了合并，将网络协议划分成五层，从高到低依次是：应用层、传输层、网络层、数据链路层和物理层，每层只与紧邻的层通信，其中应用层是最高层，物理层是最低层。

如下图所示，我们可以将网络简单地理解成一条总线，有多个主机连接到总线上，各个主机都可以向总线注入数据包和接收数据包。



网络的首要任务是对各主机进行标识，以使数据包能寻址到正确的目的主机，否则数据包将不知道发往何方。这个标识称为 IP 地址，在 IPv4 中 IP 地址为 32 位，IPv6 中 IP 地址为 128 位。网络是由分组交换机和通信链路组成的网状结构，IP 地址主要用于各分组交换机选择数据包传输的路径。端系统也需要通过 IP 地址确定数据包的输出端口。网络层协议的主要工作就是根据数据包的目的 IP 地址，选择转发的路径（输出端口），并将数据包转发给输出端口对应的网络设备。网络层协议需要在主机和分组交换机中实现。

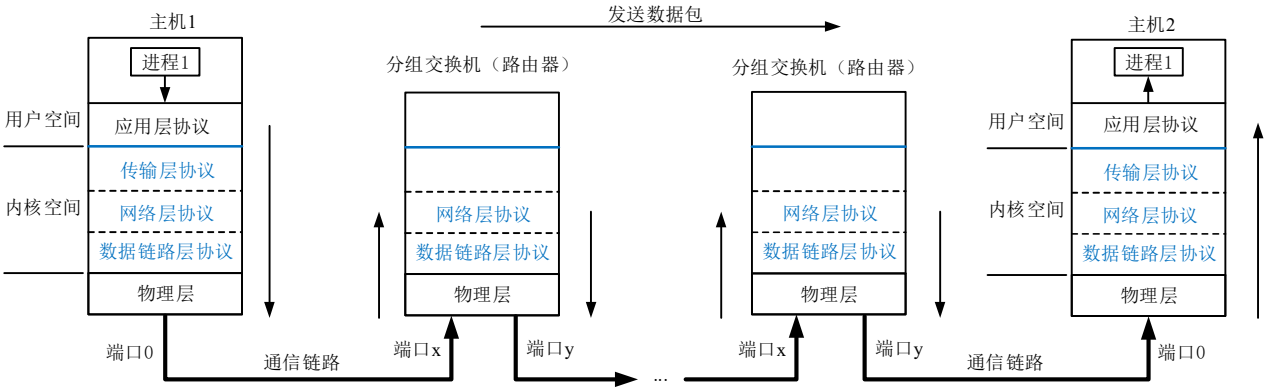
数据包到达目的主机后，需要确定数据包是传递给哪个进程的，主机通过端口号来标识进程。传输层协议主要工作是通过端口号来标识进程，为各进程建立发送、接收数据包的队列，对下与网络层进行数据包的交互，为进程屏蔽数据包发送、接收的细节，使进程访问网络犹如在访问本地文件。传输层协议只需要在主机（端系统）中实现，不需要在分组交换机中实现。因为中间节点的分组交换机只转发数据包，并不接收处理数据包。

网络中数据包通过各节点之间的接力传输，最终到达目的主机。数据链路层协议定义了数据包在相邻节点之间传输的形式和规则等。数据链路层协议有以太网、WiFi 等。

数据链路层定义了数据传输的格式，而物理层的任务是将数据链路层下传数据中的一个个比特从一个节点移动到下一个节点，实现电信号的传输。这一层的协议仍然是数据链路层相关的，例如，以太网具有许多物理层协议：一个是关于双绞铜线的，另一个是关于同轴电缆的，还有一个是关于光纤的，等等。

应用层在用户空间实现，用于进程之间定义网络数据的传输格式、使用规则等。其它层的协议并不关心进程传递数据包的内容，而只是将其从一个地方移动到另一个地方，数据的解析和使用由两端主机中的进程实现。由于应用层位于用户空间，不是位于内核空间，内核也没有实现应用层协议，因此本章不介绍应用层协议。

下图示意了主机 1 向主机 2 发送数据包时，数据包所经过的各节点及网络协议层（协议栈）：



主机 1 与主机 2 通信的进程必须使用相同的应用层协议。主机 1 进程按应用层协议要求格式化数据，将数据包传输给本机传输层，传输层在数据包头部添加上传输层报头，报头中包含通信双方进程的端口号，然后将数据包传递给网络层。

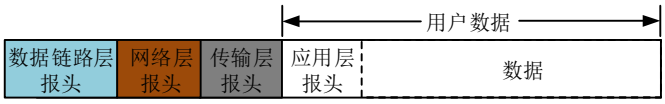
网络层需要在传输层报头前加上网络层报头，报头中包含两端主机的 IP 地址、传输层协议类型等信息，并按目的 IP 地址，确定数据包的输出端口，将数据包传递给输出端口对应的网络设备。

网络设备实现某一数据链路层协议，在网络层报头前加上数据链路层报头，报头中包含相邻节点网络

设备物理地址、网络层协议类型等信息，然后将数据包发送到通信链路上。

主机 1 连接的边缘路由器在收到数据包后，数据包从数据链路层传递到网络层。网络层通过目的 IP 地址确定转发输出端口，将数据包传递给输出端口网络设备，由其转发出去。数据包依此在网络各中间节点（分组交换机）之间接力转发，最终到达主机 2。

主机 1 进程发送的数据，在网络协议中从高层往低层传递时，每一层协议都会在数据包头部添加本层协议的报头，如下图所示：



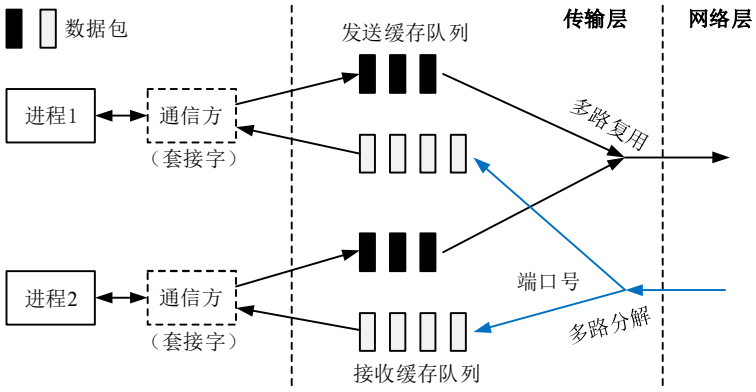
主机 2 在收到数据包后，在网络协议中从低层往高层传递，传递处理过程中将去除各层的报头。对两端主机通信的进程来说，它们看不见网络协议各层报头，网络协议对进程屏蔽了网络数据传输的细节，进程访问网络犹如在访问本地文件。

内核中实现的是因特网协议中的传输层、网络层和数据链路层，下面简要介绍一下这三层协议。

12.4.3 传输层协议简介

传输层协议为运行在不同主机上的进程之间提供逻辑通信功能。从进程的角度看，通过逻辑通信，运行不同进程的主机好像直接相连一样，传输层为进程屏蔽了网络传输的细节。

传输层上接用户进程，下接网络层。传输层协议通常为使用本协议的用户进程（通信双方）建立各自的发送、接收数据包缓存队列，如下图所示。用户进程发送数据包时，先将数据包添加到发送缓存队列，由传输层协议在合适的时机发送到网络层。传输层协议接收到发送给指定接收方的数据包时，将其添加到接收方的接收缓存队列，用户进程从接收缓存队列中读取数据包中的数据。



传输层协议通过端口号（id 值），标识通信的双方，通信双方需要知道对方的端口号。在这里现在只介绍协议的原理，而不是协议的具体实现，因此端口号暂时不对应任何实体，只是代表通信的一方。（在 Linux 内核中通信方由套接字表示，端口号被赋予套接字，而不是进程，因为一个进程可以有多个套接字，不能唯一表示通信方）。

发送数据时，传输层协议可能需要对进程发送的数据进行分段，以方便传输，并附上传输层报头。在因特网术语中传输层数据包称为报文段。传输层协议负责将数据包添加到进程发送数据包缓存队列，或直接传递给网络层。发送缓存队列中的数据包，由传输协议在合适的时机发送给网络层。

接收数据时，传输层从网络层接收数据包，传输层协议通过报头中指定的目的端口号，查找到接收方，将数据包添加到其接收缓存队列。用户进程读取网络数据时，从接收缓存队列数据包中读取数据。

因特网协议簇提供了两种主要的传输层协议（当然还有其它的传输层协议），一种是 UDP（用户数据报协议），另一种是 TCP（传输控制协议）。

在对 UDP 和 TCP 进行简要介绍之前，先简单介绍一下因特网的网络层。因特网网络层协议称为 IP，即网际协议。IP 为主机之间提供逻辑通信。IP 服务模型是尽力而为交付服务。这意味着 IP 尽它“最大努力”在主机之间交换数据包，但不做任何保证。它不确保数据包的交付，不保证数据包的按序交付，不保

证数据包中数据的完整性，因此 IP 被称为不可靠服务。也就是说，数据包在网络节点之间传输时，可能丢失、损坏、不按顺序送达等。

现在我们总结一下 UDP 和 TCP 所提供的服务模型。UDP 和 TCP 最基本的责任是将两个端系统间 IP 的交付服务扩展为运行在端系统上的两个进程之间的交付服务。由于 IP 传输的不可靠性，传输层协议需要增加功能，以提高数据传输的可靠性。UDP 和 TCP 可以通过在传输层报头中写入差错检查字段而提供完整性检查。数据交付服务和差错检查是两种最低限度的传输层服务。

UDP 为调用它的进程提供一种不可靠、无连接的服务。它只提供数据交付和差错检查这两种最低限度的传输层服务。此处的数据交付也是不可靠的，只是提供交付的机制，并不保证正确性和可靠性。用户进程通过 UDP 发送的数据包可能丢失、损坏、不按顺序到达目的进程接收缓存队列。UDP 提供的服务比较简单，因此协议实现也比较简单。

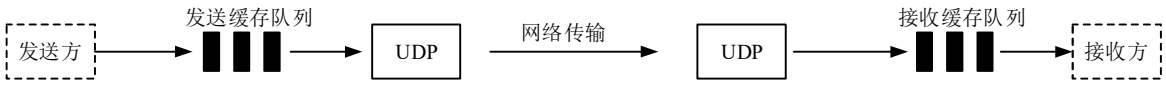
TCP 为进程提供一种可靠的、面向连接的服务。面向连接是指通信双方进程在通信前需要先建立连接，用户进程可同时与多个其它通信方建立连接。TCP 为每个连接建立一组接收、发送缓存队列，也就是说一个进程可以有多组接收、发送缓存队列。

可靠是指 TCP 保证发送数据包能够正确、完整、不丢失、按顺序到达接收通信方接收缓存队列。为了实现传输的可靠性，TCP 要复杂的多。为此 TCP 通过确认、重传、流量控制等机制实现可靠性。

UDP 类似于现实生活中的寄邮件，寄邮件前不需要联系收件人，只需写好收件人和地址交给邮政系统即可。邮件可能出现丢失、损坏、或延时送达等情形。TCP 类似于打电话，打电话前需要拨通对方电话，对方接听后才能通话，如果没听清对方的话，可以立即叫对方重复一遍，以保证正确接收信息。

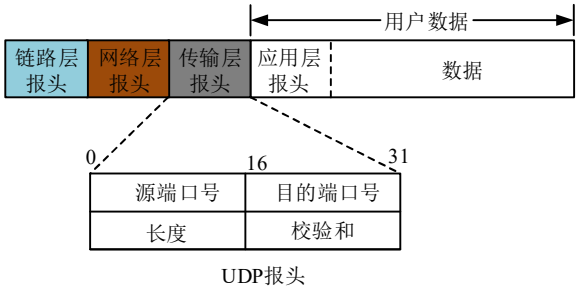
1 UDP

UDP 是一种无连接，不可靠的传输层协议，它仅提供数据交付和差错检查的功能。如下图所示，发送方将发送的数据交给 UDP，由其发送到网络上。到达接收方 UDP 后，UDP 根据端口号查找接收方，将数据交给接收方。



■UDP 报头

UDP 只是做了传输层能够做的最少的工作，即实现报文段的多路分解与复用以及差错检查。UDP 报头如下图所示，报头只有 4 个字段，分别是源端口号（发送方，2 字节）、目的端口号（接收方，2 字节）、长度和检验和。长度表示报文段的字节数（含报头和数据），校验和用于提供差错检查功能。



UDP报头

端口号用于标识通信双方，也用于实现多路分解与复用。UDP 为主机中使用 UDP 的通信方分配一个 16bit 的端口号。端口号大小在 0~65535 之间，0~1023 范围的端口号称为周知端口号，是受限制的，这是指它们保留给了诸如 HTTP 和 FTP 等周知的应用层协议使用。开发新的应用程序时，必须为其分配一个端口号。用户可编程指定本方的端口号（“周知协议”的服务器进程），也可由内核自动分配，自动分配的端口号范围是（1024~65535）。

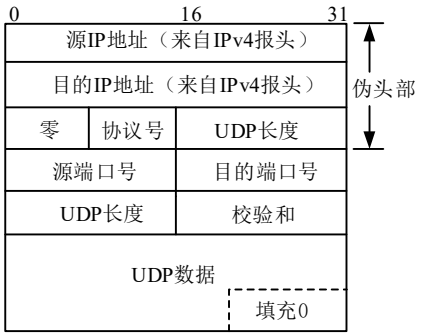
UDP 收到来自网络层的数据包后，从报头中提取目的端口号，在本机中查找目的方，将数据包绑定到

其接收数据包缓存队列，这被称为多路分解。

UDP 在发送数据包时，在源主机中从不同发送方缓存队列收集数据包，为每个数据包封装上报头信息生成报文段，然后传递给网络层协议的流程称为多路复用。

■UDP 校验和

UDP 校验和是我们遇到的第一个端到端的传输层校验和，它用于差错检查。它覆盖了 UDP 报头、UDP 数据和一个伪头部。它由初始的发送方计算得到，由最终的目的方校验。它在传送中不会被修改（除非它通过一个 NAT）。对于 UDP 来说，校验和是可选的（IPv6 中是强制的），通常都会选择。下图示意了计算 UDP 校验和时覆盖的字段，包含了伪头部以及 UDP 头部和负载：



计算校验和时数据长度必须偶数字节对齐，因此如果数据报长度为奇数，需要在末尾填充一个 0。伪头部只用于计算校验和，实际并不存在，也不会传送出去。这个伪头部包含来自 IPv4 报头的源 IP 地址和目的 IP 地址，以及协议类型值。它的目的是让 UDP 层验证数据是否已经到达正确的目的地。

如果计算出来的校验和值正好为 0x0000，它在头部中会被保存为 0xFFFF（等于算术反码）。报头中校验和值为 0x0000 表示发送方没有计算校验和。如果接收方检测到了一个校验差错，UDP 数据报会被毫无声息地丢弃，而不会产生差错信息（ICMP 报文）。

近来已有兴趣关注于对 UDP 校验和的松懈使用，主要是对一些对差错不完全看重的应用。这些讨论关系到是否有部分校验和。一个称为 UDP-Lite 或 UDPLite 的协议通过修改传统的 UDP 协议，提供了部分校验和来解决这个问题。这些校验和只覆盖每个 UDP 数据报里的一部分负载。UDP-Lite 有它自己的 IPv4 协议和 IPv6 下一个头部字段值，因此它可以认为是一种独立的传输层协议。UDP-Lite 用一个校验和覆盖范围字段取代了长度字段来修改 UDP 头部。UDP-Lite 报头如下图所示：

UDP-Lite报头

校验和覆盖范围字段是被校验和覆盖的字节数（从 UDP-Lite 头部第一个字节开始）。字段值 0 表示覆盖整个数据报，与 UDP 相同，最小值为 8（只覆盖报头）。

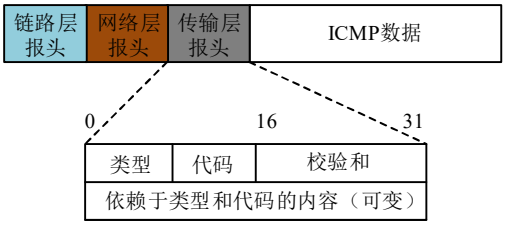
2 ICMP

网络层 IP 协议本身并没有为端系统提供直接的方法来发现那些发往目的地失败的 IP 数据包。IP 也没有提供直接的方式来获取诊断信息。为了解决这些不足之处，将一个特殊的 Internet 控制消息协议（Internet Control Message Protocol，ICMP）与 IP 结合使用，以便提供与 IP 协议层配置和 IP 数据包处置相关的诊断和控制信息。ICMP 通常被认为是 IP 层的一部分，它需要在所有 IP 实现中存在。它使用 IP 协议进行传输。因此，确切地说，它既不是一网络层协议，也不是一个传输层协议，而是位于两者之间。

ICMP 负责传递可能需要注意的差错和控制报文。ICMP 报文通常是由 IP 层本身、上层的传输协议（如 TCP 或 UDP），甚至某些情况下是用户进程触发发送的。ICMP 并不为 IP 网络提供可靠性，相反，它表

明了某些类别的故障和配置信息。ICMP 有 ICMPv4 和 ICMPv6 版本，分别用于 IPv4 和 IPv6。

ICMP 报头结构如下图所示：



ICMP报头

ICMP 报头字段描述如下：

●**类型**：8bit，表示 ICMP 报文的类型，ICMPv4 中，类型字段有 42 个不同的值，大概只有 8 个是经常使用的。ICMPv6 中类型值与 ICMPv4 不同。

●**代码**：8bit，代码，许多类型的 ICMP 报文使用不同的代码字段值进一步指定报文的含义。ICMPv4 和 ICMPv6 中代码值也不相同。

●**校验和**：涵盖整个 ICMP 报文段的校验和。

●**32 位的可变部分**：随后的 4 个字节的内容取决于 ICMP 中的类型和代码值。

ICMP 报文可分为两大类：有关 IP 数据报传递的 ICMP 报文（称为差错报文），以及有关信息采集和配置的 ICMP 报文（称为查询或信息类报文）。对于 ICMPv4，信息类报文包括回显请求和回显应答，以及路由器通告和路由请求。最常见的差错报文类型包括目的不可达、重定向、超时和参数问题。

下表列出了标准 ICMPv4 定义的报文类型：

类型	正式名称	差错/信息	用途
0	**回显应答**	**信息**	**回显（ping）应答，返回数据**
3	**目的不可达**	**差错**	**不可达的主机/协议**
4	源端抑制	差错	表示拥塞（弃用）
5	**重定向**	**差错**	**表示应该被使用的可选路由器**
8	**回显**	**信息**	**回显（ping）请求（数据可选）**
9	路由器通告	信息	指示路由器地址/优先级
10	路由器请求	信息	请求路由器通告
11	**超时**	**差错**	**资源耗尽（如 IPv4 中 TTL 为 0）**
12	**参数问题**	**差错**	**有问题的数据包或者头部**

（0、3、5、8、11、12 类型是最常用的报文类型）

IANA 维护了一个报文类型的正式列表，每种报文类型下又使用多个代码号，表示报文类型更详细的信息。ICMPv6 与 ICMPv4 的报文类型和代码不同。

ICMP 报文数据与报文类型和代码相关，不同的类型（代码）报文数据各不相同。

在 ICMP 中，对传入报文的处理随着系统的不同而不同。一般说来，传入的信息类请求将被操作系统自动处理，而差错类报文传递给用户进程（ping）或传输层协议。

3 TCP

TCP 是一种面向连接，提供可靠数据传输的传输层协议。在通信前，双方进程需要建立连接，在通信过程中需要维持连接状态及连接参数，通信结束后需要关闭连接。

TCP 协议除了提供前面介绍的多路分解与复用以及差错检查外，还提供了几种附加服务。

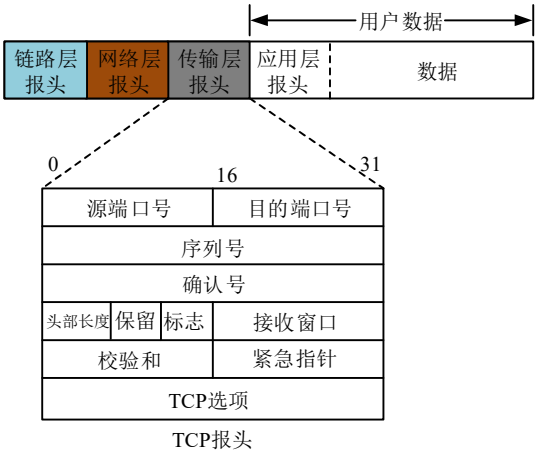
一是提供可靠数据传输服务：TCP 构建在不可靠的 IP 层上，通过使用应答、确认号、序列号、流量控制、重传等机制，TCP 确保正确、完整、按序地将数据包从发送方交付给接收方。

二是提供拥塞控制服务：TCP 感知到通信链路和交换设备拥塞时（分组缓存溢出），调节自身发送进网络的流量速率，以防止通信链路和交换设备因被过多流量淹没而产生分组丢失。拥塞控制更像是一种提供给整个因特网的服务。

TCP 要提供可靠数据传输和拥塞控制，其协议必然是复杂的，实际上 TCP 要比 UDP 复杂的多。下面先介绍 TCP 报头结构，然后介绍 TCP 连接的建立和关闭，可靠数据传输的实现，最后介绍拥塞控制原理。

■TCP 报头

TCP 的报头结构，如下图所示：



TCP报头

TCP 报头中各字段描述如下：

●**源端口号**：16bit，发送方端口号。

●**目的端口号**：16bit，接收方端口号。

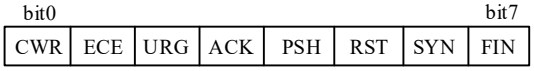
●**序列号、确认号**：32bit，序列号和确认号被发送方和接收方用来实现可靠数据传输。TCP 把发送的数据视为一个字节流，对字节流中每个字节依次编号（不一定从 0 开始）。如果进程发送的字节流较大则需要将数据拆分成多个段，序列号表示本数据段首字节在字节流中的编号，而不是数据段的编号。假设传输的字节流共 2000 字节，每个数据段 1000 字节，首字节编号从 0 开始（也可以不从 0 开始），则第一个数据段中的序列号为 0，第二个数据段的序列号为 1000。

确认号要比序列号难处理一些。TCP 是全双工通信的，在进程 A 向进程 B 发送数据时，进程 B 也向进程 A 发送数据。假设进程 B 收到了进程 A 发送的编号为 0~535 的所有字节，进程 B 期望接收下一个进程 A 报文段的序列号（起始字节编号）为 536，则在进程 B 向进程 A 发送的数据报中将确认号设置为 536。确认号即接收方期望接收字节流中下一字节的编号。确认号同时也通知发送方在确认号之前的所有数据接收方已正确接收。

●**头部长度**：4bit，表示以 4 字节（字）为单位的 TCP 报头的长度，因此 TCP 报头最长为 60 字节，没有选项时 TCP 报头为 20 字节。

●**保留**：4bit。

●**标志**：8bit，标记位定义如下图所示：



CWR：拥塞窗口缩小标志，提示发送方降低发送速率。

ECE：显式拥塞标志（ECN），它提供了一种发送有关网络拥塞的端到端的通知而不丢失数据包的机制。用于网络中间节点（路由器）发送给端系统显式拥塞标志。

URG：指示紧急指针是否有意义（很少使用）。

ACK：指示 TCP 报头中确认号是否有意义。

PSH：指出应尽快将数据交给用户进程。

RST：重置连接，在收到并非当前连接的数据段时使用。

SYN：在建立连接进行 3 次握手时发送 SYN 标志。

FIN：指示后面没有来自发送方的其他数据，用于关闭连接。

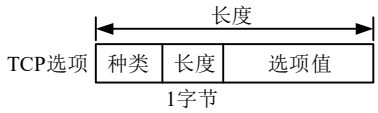
●**接收窗口**：16bit，接收方发送给发送方的接收方剩余缓存大小，用于流量控制（提示发送方限制发送速率）。

●**校验和**：16bit，用于差错检查。

●**紧急指针**：16bit，只有在 URG 标志置位时才有效。紧急指针表示紧急数据的最后一个字节地址（相对于序列的偏移量）。在实践中，PSH、URG 和紧急指针并没有使用。

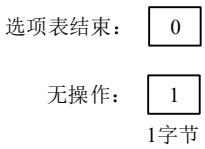
●**TCP 选项**：32bit，可选的选项，用于协商 MSS 值（数据报中用户数据字节数最大值），定义时间戳等。受限于 TCP 报头长度（共 60 字节），在一个 TCP 报头中选项总的长度不能超过 40 字节。

TCP 头部可以包含多个选项，每个选项的结构通常如下图所示：



一个 TCP 选项有三个字段，分别是：种类（kind，占一字节）、长度（len，占一字节）、选项值（若干字节）。种类用于标识选项的类别，由一个字节的整数表示。长度由一个字节表示，指示了选项总的长度。选项值的长度为长度字段表示的长度值减 2（种类、长度字段各占去一个字节）。

其中种类值为 0 和 1 的选项比较特殊，它们各只占一个字节，即只有种类字段，没有长度和选项值字段。种类 0 的选项表示选项列表结束，种类 1 的选项为空选项，用于补齐字节，因为 TCP 报头需要 4 字节对齐。种类 0 和 1 的选项如下图所示：



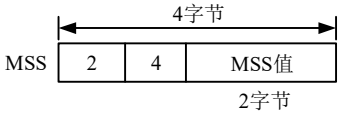
下表列举了一些目前较常用的选项：

种类	长度（字节数）	名称	描述
0	1	EOL	选项列表结束
1	1	NOP	无操作（用于填充）
2	4	**MSS**	最大报文段大小（用户数据大小）
3	3	**WSOPT**	窗口缩放因子（窗口值左移量）， 在 SYN 报文段中设置
4	2	**SACK-Permitted**	发送者支持 SACK 选项， 在 SYN 报文中设置
5	可变	**SACK**	SACK 阻塞（接收到乱序数据） 选择确认选项
8	10	TSOPT	时间戳选项
28	4	UTO	用户超时
29	可变	TCP-AO	认证选项
253	可变	Experimental	保留供实验所用
254	可变	Experimental	保留供实验所用

下面再介绍几个常用的 TCP 选项：

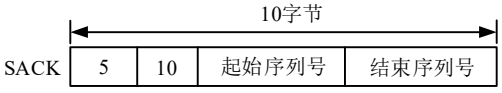
●**MSS**：最大报文段选项。指示本端可接收报文段的最大长度，即报文段中除 IP 报头、TCP 报头之外的用户数据长度。在建立 TCP 连接时，通信的每一方在 SYN 报文段中的 MSS 选项中说明自己允许的最大报文段大小。IPv4 中此值的典型值为 1460 字节，即 1500 字节减 20 字节的 IP 报头和 20 字节的 TCP 报头。

MSS 选项如下图所示：



●**SACK-Permitted**：选择确认允许选项，表示本端可接收“选择确认”选项。在建立连接时，在 SYN 或 SYN ACK 报文段中设置此选项，此选项没有选项值。

●**SACK**：选择确认选项。在收到乱序到达的数据包时，接收方可向发送方发送选择确认选项，指示接收方已接收数据块的序列号范围。每个范围称作一个 SACK 块，由一对 32 位的序列号表示。因此，如果一个 SACK 选项包含了 n 个 SACK 块，则选项长度为（8n+2）个字节，另外 2 个字节为种类和长度字段。SACK 选项如下图所示：

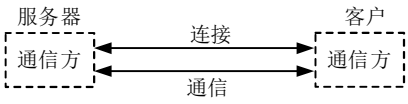


●**WSOPT**：接收窗口缩放因子选项。在 TCP 报头中窗口长度字段为 16 字节，能表示的接收窗口最大值为 64KB。WSOPT 选项可将窗口长度字段扩展为 30 位。WSOPT 选项值由一个字节表示，取值表示的是将 TCP 报头中接收窗口长度值（16 字节）左移的位数，取值范围[0,14]，用于扩展接收窗口长度。例如，假设接收窗口值为 100，WSOPT 选项值为 14，则接收窗口长度为 100×2^{14} 字节。

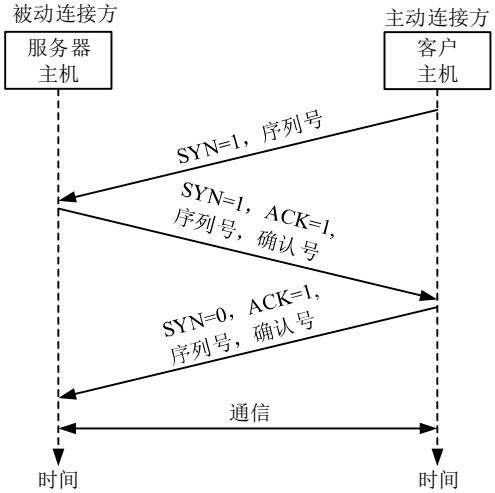
WSOPT 选项只能出现在一个 SYN 报文段中，当连接建立后比例因子与方向是绑定的，即双方的比例因子可以不同。

■连接管理

TCP 是面向连接的可靠传输层协议，在进行通信前发送方与接收方之间需要建立连接。这种连接属于逻辑上的连接，连接建立后，双方之间是一对一的通信，不需要再指明对方地址。通信结束后需要关闭连接，释放连接资源，如下图所示。



连接通常由客户端发起，称为主动连接方，服务器端为被动接收方。连接的建立一共要传输至少 3 个报文段，因此称为 3 次握手，如下图所示。



第一步：客户端向服务器端发送一个特殊的 SYN 报文段，称为连接请求报文段。该报文件段中不包含应用层数据，但报文段报头中的 SYN 标志位被置 1，并随机选择一个序列号写入报头中，此报文件段被发往服务器端。

第二步：服务器端接收到客户端发送的连接请求 SYN 报文段后，为该 TCP 连接创建连接请求，并向该客户端发送允许连接的应答 ACK 报文段。ACK 报文段也不包含应用层数据，SYN、ACK 标志位被置 1，

确认号为客户端发送的连接请求报文段中序列号加 1，服务器选择自己的序列号写入 ACK 报文段。

第三步：客户端收到允许连接 ACK 报文段后，也为连接分配缓存和变量。客户端向服务器端发送另一个特殊报文段，用于确认允许连接报文段。此报文件段报头 SYN 为 0，ACK 标志位置 1，确认号为服务器 ACK 报文段中序列号加 1，序列号为 SYN 报文段中序列号加 1。服务器端收到此报文段后，为连接分配缓存和变量等，连接建立。

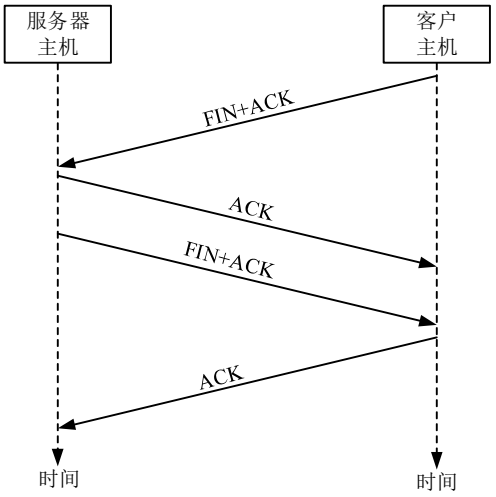
完成以上 3 个步骤后，客户与服务器就可以相互发送数据了，在这个过程中一共发送了 3 个报文段，因此称为 **3 次握手**。

通信结束后，需要关闭连接，通信双方都要执行本端的关闭操作，如下图所示。

第一步：假设关闭操作由客户端发起，客户端向服务器端发送一个终止报文段，发起本端的关闭操作，报头中 FIN、ACK 标志位置为 1（带确认号和序列号，下面的报文段中也带，不再重复了），服务器端收到后发回一个应答 ACK 报文段。

第二步：服务器端感知到客户端发起了关闭操作后，发送本端的终止报文段，FIN、ACK 设置为 1，客户端收到后发回一个应答 ACK 报文段，此时连接关闭，双方将释放各自的连接资源。

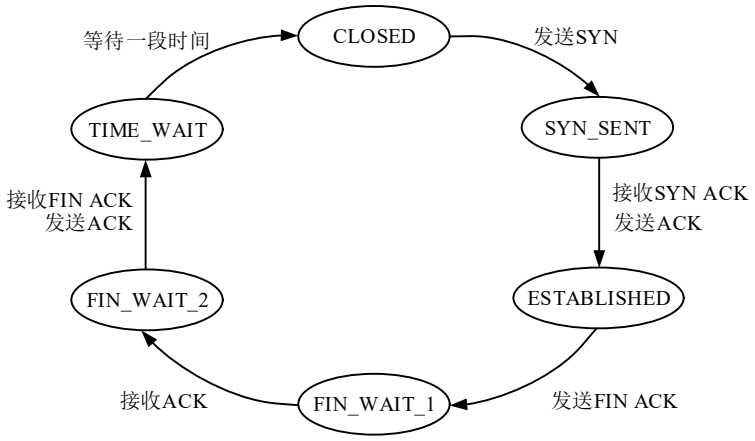
在关闭操作中一共传输了至少 4 个报文段，因此被称为 4 次握手，有时又被称为修改的 3 次握手。



■TCP 状态转换

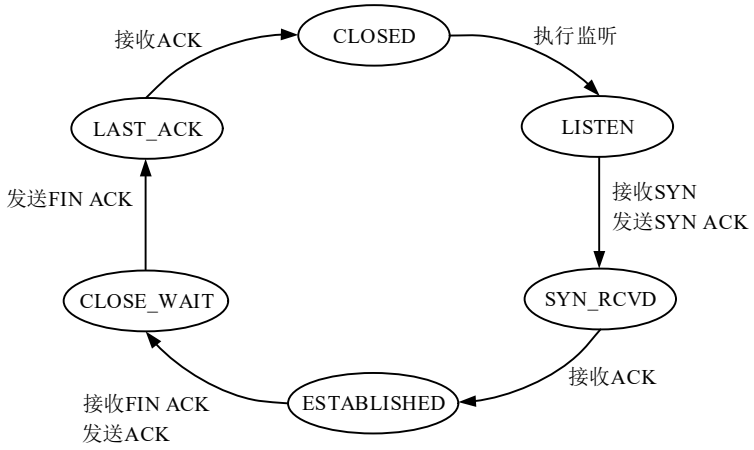
我们已经介绍了一个 TCP 连接的建立和关闭，也看到了在一个连接的不同阶段需要发送和各种类型的报文段。这些决定了 TCP 应该做什么的规则其实是由 TCP 通信方所属的状态决定的。在一个 TCP 连接的生命周期内，通信方在各种 TCP 状态之间变迁。

下图说明了客户端 TCP 通信方所经历的一系列典型 TCP 状态：



通信方开始处于关闭 CLOSED 状态，发起连接请求，发送 SYN 请求报文段后进入 SYN_SENT 状态。当接收到服务器发回的 SYN ACK 应答，并发送 ACK 应答后，进入 ESTABLISHED 状态，表示连接已经建立。发送 FIN ACK 终止报文段后，进入 FIN_WAIT_1 状态，等待服务器发回的 ACK 应答。接收到 FIN 的应答报文段后进入 FIN_WAIT_2 状态，等待服务器发送的 FIN ACK 报文段。当收到服务器端的 FIN ACK 报文段后，将发送应答 ACK 报文段，进入 TIME_WAIT 状态。在 TIME_WAIT 状态等待一段时间后，例如：30 称、1 分钟或 2 分钟等，连接正式关闭，通信方进入关闭 CLOSED 状态，随后释放连接资源。

下图说明了服务器端 TCP 通信方所经历的一系列典型 TCP 状态：



通信方开始处于 CLOSED 状态，执行监听操作（表示可以接收连接请求）后，进入 LISTEN 状态。当收到客户端连接请求 SYN 报文段，发回 SYN ACK 报文段后，进入 SYN_RCVD 状态。接收到客户端连接 ACK 报文段后，进入 ESTABLISHED 状态，表示连接已经建立。当收到客户端 FIN ACK 报文段（关闭请求）后，发送应答 ACK 报文段，并进入 CLOSE_WAIT 状态。当发送本端的关闭请求 FIN ACK 报文段后，进入 LAST_ACK 状态，收到客户端的 ACK 应答报文段后，进入 CLOSE 状态，释放连接资源，连接关闭。

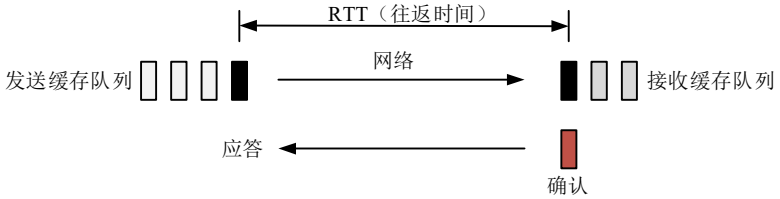
以上状态转换都是假设连接和关闭请求都是由客户端发起的，只考虑了状态的正常转换，没有考虑异常情况下的状态转换，更详细的内容请读者参考相关专著。

■可靠数据传输

TCP 建立在不可靠的 IP 层之上，它需要向用户进程提供可靠的数据传输服务。IP 层提供的是尽力而为的交付服务，不保证数据包的完整、正确和按序送达，数据包在网络传输过程中可能损坏、乱序到达，甚至丢失。TCP 要在此基础上向用户进程提供正确、完整、按序交付数据的服务，因此 TCP 必定是复杂的。下面简要介绍 TCP 实现可靠数据传输的原理。

●确认应答

TCP 与 UDP 一样，通过校验和来检测数据传输错误。接收方在正确接收到发送方发送的数据后，向发送方发送一个应答数据包，称为确认应答，以通知发送方数据包已正确接收，如下图所示。如果数据包丢失、检测出错误等，将不发送应答数据包。



数据包从发出到收到应答所经历的时间称为往返时间（RTT），在后面设置超时定时器时将用到此时间值。

为提高传输效率，发送方通常以流水线的方式，发送多个数据包，而不是等到收到一个数据包的确认

应答后，再发送下一个数据包。为了区分应答是对哪个数据包的应答，TCP 报头中增加了序列号和确认号。

TCP 是一个双向传输协议，每一方传输的数据视为一个无结构的、有序的字节流，并对字节流中每个字节赋予一个编号，称为序列号。序列号不一定就是从 0 开始的，在建立连接时，将会确定本方发送字节流的起始序列号。序列号是一个 32 位整数，当达到最大值时，将会回到 0，重新从 0 开始计数。

在发送数据包时，TCP 报头中的序列号表示用户数据起始字节在字节流中的编号。当接收方收到数据包后将发回应答数据包，其中的确认号表示接收方期望收到的下一个数据包的序列号。例如：假设发送方发送的数据包序列号为 0，用户数据长度为 100，此数据包中用户数据编号范围是 0-99，接收方在收到此数据包后，期望下次收到的是从编号 100 开始的用户数据，因此发回给发送方的确认号为 100。

接收方不一定对每个数据包都发送应答，而可能采用累积确认的方式。接收方发送给发送方的确认号表示，在确认号之前的数据接收方都已经正确接收。

●重传

TCP 发送到网络中的数据包可能出现损坏、丢失等，这时接收方将不发送确认应答，对传输不成功的数据包，发送方将执行重传。

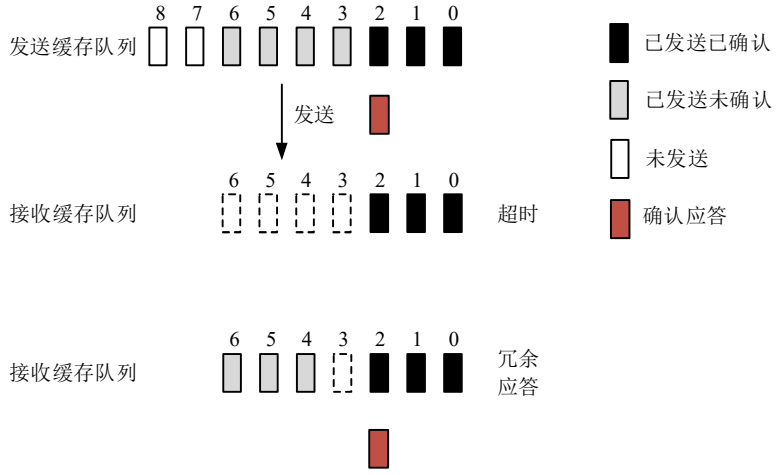
发送方维护了一个最小确认号，每次收确认应答时，将根据实际情况更新最小确认号数值。在此确认号之前的数据接收方都已正确接收，不必再理会。

发送方在发送数据包时，将启动一个定时器。如果在定时器到期时，发送方都没有收到确认应答，即认为数据包发生了丢失（也可能是确认应答丢失了）。此时，发送将重传最小确认号之后，已经发送但未收到确认应答的数据。

如下图所示，图中为了方便用数据包序号代替字节流序列号，0-2 号数据包已发送并确认，发送方的最小化确认号为 3。现在发送了 3-6 号数据包，并启动定时器，但是在定时器到期时，都没有收到确认应答，则发送方将重传 3-6 号数据包。这称为超时重传。

如果在定时器到期前收到了确认应答，则将更新发送方最小确认号，并重新启动定时器。定时器的作用是判断在指定的时间内是否收到了确认应答，如果收到了，则定时器重新启动，重新开始计时（复位），如果没有收到确认应答，定时器超时，就指示发送方需要重传已发送但未确认的数据包。

定时器超时时间的设置是一个值得研究的问题，如果设的过小，将执行不必要的重传，如果设的过大，将不能及时检测到数据包的丢失，通常根据往返时间 RTT 计算得到。



同一个 TCP 连接发送的数据包，可能通过为不同的网络路径到达接收方，因此数据包可能不按序到达接收方，或接收数据包出现空洞（某一路径上的数据包丢失了）。如上图中所示，发送方发送了 3-6 号数据包，4-6 号数据包到达了接收方，3 号数据包丢失了。接收方对 4-6 号数据包的应答中，其确认号依然是 3（同 2 号数据包应答），表示接收方还是只正确接收到了 3 号之前的数据包。这种应答称为冗余应答。发送方接收到冗余应答时，表示发生了数据包丢失，也将执行重传，重传 3-6 号数据包，这种重传称为快速重传。

在上面的例子中，接收方 4-6 号数据包已正确到达，对这些数据包的处理由具体的 TCP 实现决定，可

能直接将其丢弃，等待重传数据包，也可能缓存起来，等待 3 号数据包的到达。

另外，TCP 可通过选择确认 SACK 选项，指示丢失的数据包，使发送方执行选择重传，而不需要重传已经到达的数据包。

●流量控制

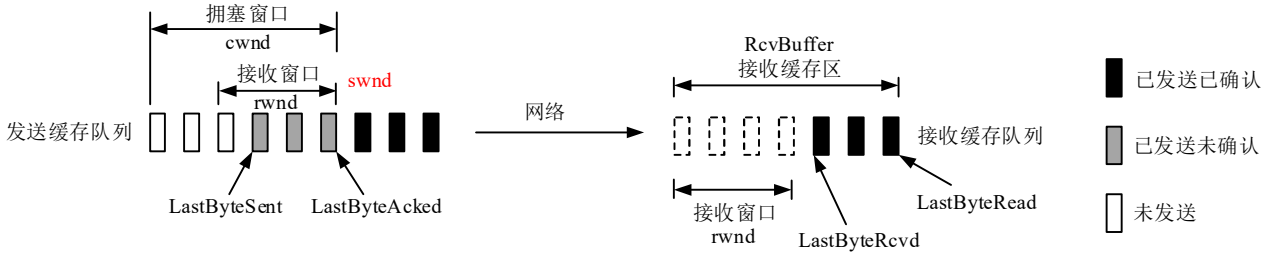
数据包在网络中传输可能出现丢失。丢失通常发生在两个点，一个是接收方的接收缓存溢出，丢弃数据包，另一个是中间路由器缓存溢出。发送方都是通过控制发送到网络中的数据包数量，来避免缓存溢出造成的数据包丢失。

为了防止接收方缓存溢出而采取的措施称为流量控制，为防止路由器缓存溢出而采取的措施称为拥塞控制。下面先介绍流量控制的实现，后面将介绍拥塞控制的实现。

如下图所示，接收方为连接分配了一个接收缓存区，其大小由 RcvBuffer 表示（字节数）。接收方跟踪两个变量：

LastByteRead：接收方应用进程从接收缓存区中读出数据流的最后一个字节编号。LastByteRead 之前（含）的数据进程已读取，已从接收缓存区中移出。

LastByteRcvd：接收方从网络中接收到的数据流中最后字节的编号。



为了防止缓存溢出，必须满足以下条件：

LastByteRcvd-LastByteRcvd≦RcvBuffer　/*缓存的数据不能超过缓存区大小*/

接收缓存区中的空闲区域大小为：

rwnd=RcvBuffer-(LastByteRcvd-LastByteRcvd)

rwnd 表示接收方的接收窗口，接收方在发送给发送方的确认应答中，将 rwnd 值写入 TCP 报头的**接收窗口**字段中。

发送方同样跟踪两个变量：

LastByteAcked：最后确认数据的字节编号。

LastByteSent：已发送数据最后的字节编号。

（LastByteSent-LastByteAcked）表示已发送，但尚未到达接收方的数据大小。为了防止接收方缓存溢出，必须满足以下条件：

LastByteSent-LastByteAcked≦**rwnd**

由上式可知， 发送方已发送，但尚未到达的数据长度不能超过接收窗口，即在网络中传输的数据长度不能超过接收窗口。

发送方在收到确认应答时，将从中提取确认号和接收窗口值 rwnd，更新 LastByteAcked 值。发送方下次发送的数据最大字节编号必须满足以下条件：

LastByteSent≦LastByteAcked+**rwnd**

●拥塞控制

网络中的路由器也会因其接收缓存区溢出而丢弃数据包，这称为网络拥塞。IP 层没有向端系统提供显

式的网络拥塞反馈。端系统（TCP）必须自己感知网络拥塞，并降低数据发送速率，以避免数据丢失，这称为拥塞控制。

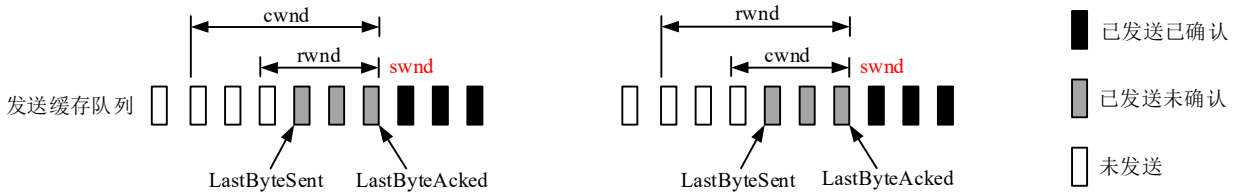
运行在发送方的 TCP 拥塞控制机制跟踪一个额外的变量，即**拥塞控制窗口**，由 **cwnd** 表示。它对 TCP 发送方发送到网络中的数据量进行控制。拥塞控制窗口可理解为某一时刻发送方能发送到网络中的最大数据量（字节数），即在网络中传输的数据量（不含已确认的数据）。

在前面介绍的流量控制中，我们知道（LastByteSent-LastByteAcked）表示某一时刻发送方已经发出的，但尚未确认的数据量。为了实现流量控制和拥塞控制，必须满足以下条件：

（LastByteSent-LastByteAcked）≦min(cwnd,rwnd)

也就是说发送的数据量不能超过接收窗口和拥塞窗口中的最小值。min(cwnd,rwnd)表示发送方的发送窗口，本书用 swnd 表示。

cwnd、rwnd、swnd 都起始于最后确认的字节编号（第一个尚未确认的字节编号），如下图所示，swnd 取 cwnd、rwnd 中的最小值，它是实际控制发送速率的量。



发送方在收到确认应答时，将会更新 LastByteAcked 值，更新 rwnd 以及 cwnd 值，各窗口将会向后移动并更新大小值。发送方以流水线方式发送数据时，最后发送的数据字节编号 LastByteSent 不能超过（LastByteAcked+swnd）值，以此来控制发送速率。

下面考虑另一个问题，TCP 发送方如何感知到网络拥塞了，又如何调节拥塞窗口 cwnd 值呢？

TCP 认为如果数据包发生了丢失，就认为网络拥塞了。如果出现了定时器超时或收到了冗余确认应答，就认为发生了数据包丢失，这也是网络拥塞的指示。

TCP 如果感知到了网络拥塞就应该减小拥塞窗口 cwnd 值，如果收到了确认应答，表示网络正常，可增大拥塞窗口，以充分利用网络带宽。

TCP 拥塞控制算法就是感知网络拥塞程度，调节拥塞窗口的机制。TCP 拥塞控制算法包括 3 个主要部分：（1）慢启动；（2）拥塞避免；（3）快速恢复。慢启动和拥塞避免是 TCP 的强制部分，快速恢复是推荐部分。

（1）慢启动：TCP 连接开始时，cwnd 设为较小的值，即 MSS 值（一个报文段中用户数据最大长度）。在慢启动状态，cwnd 值以 1 个 MSS 开始，每当确认一个报文段，cwnd 值就增加一个 MSS 值。TCP 开始时发送一个报文段，等待确认，收到确认后 cwnd 值增加 1 个 MSS 值，为 2MSS。然后，发送两个报文段，收到确认后，cwnd 值再增加 2 个 MSS 值，为 4MSS。依此类推，cwnd 以 2 的指数级增长。

那么，何时结束增长呢？慢启动提供了几种答案：

1.如果发生了由超时指示的丢包事件，将设置另一个状态变量 ssthresh（慢启动阀值）值为 cwnd/2（当前拥塞窗口值的一半），然后将 cwnd 值重置为 1MSS（最小值），重新开始慢启动。

2.当 cwnd 达到或超过 ssthresh 值时，将结束 cwnd 值的指数增长，而进入拥塞避免模式，即更为谨慎地增加 cwnd 值，见下文。

3.当收到 3 个冗余确认应答时（拥塞没有超时丢包严重），TCP 执行快速重传并进入快速恢复状态，见下文。

（2）拥塞避免模式：在慢启动过程中，当 cwnd 值达到或超过 ssthresh 值时，说明离拥塞已经不远了，再以指数级的形式增加 cwnd 值已经不合适了。此时，将结束慢启动，进入拥塞避免模式。

拥塞避免模式将更谨慎地增加 cwnd 值（线性增长），即减小 cwnd 的增加量。那么何时结束增长呢？

当出现超时丢包事件时，ssthresh 值更新为当前 cwnd 值的一半，cwnd 值重置为 MSS 值，与慢启动相

同。

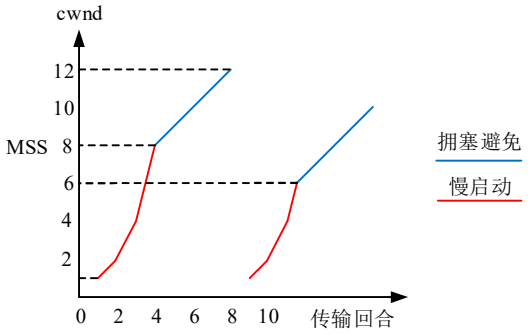
当收到 3 个冗余确认应答时，ssthresh 值更新为当前 cwnd 值的一半，将 cwnd 值减半，接下来进入快速恢复状态。

（3）快速恢复状态：当收到 3 个冗余确认应答时，ssthresh 值更新为当前 cwnd 值的一半，将 cwnd 值减半，此时进入快速恢复状态。

在快速恢复中，对收到的每个冗余的 ACK，cwnd 值增加 1 个 MSS。最终，当丢失报文段的 ACK 到达时，在降低 cwnd 后进入拥塞避免状态。

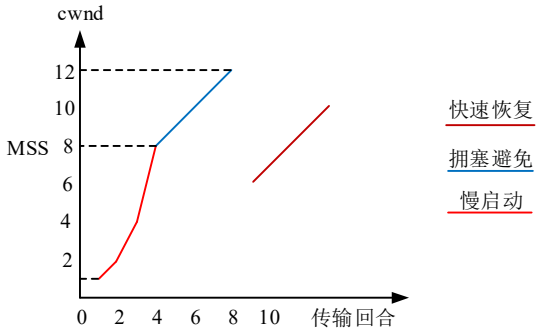
快速恢复是 TCP 推荐的而不是必需的构件。一种称为 TCP Tahoe 的 TCP 早期版本，不管发生何种丢包事件，都将 cwnd 减至 1 个 MSS 值，重新进入慢启动。TCP 的较新版本 TCP Reno，则综合了快速恢复。

下图示意了由超时引发丢包时，cwnd 值的变化情况：



假设 ssthresh 值初始值为 8MSS。在慢启动阶段，当 cwnd 值到达 ssthresh 值（8MSS）时，进入拥塞避免状态，cwnd 值线性增长。达到拥塞值时（此处假设为 12MSS），超时丢包事件发生，ssthresh 值更新为当前 cwnd 值的一半，即 6MSS，cwnd 值设为 MSS，重新开始慢启动。

下图示意了由 3 个冗余确认应答指示丢包事件发时，cwnd 值的变化情况：



假设 ssthresh 值初始值为 8MSS。在慢启动阶段，当 cwnd 值到达 ssthresh 值（8MSS）时，进入拥塞避免状态，cwnd 值线性增长。达到拥塞值时（此处假设为 12MSS），收到 3 个冗余确认应答。此时，进入快速恢复状态，ssthresh 值更新为当前 cwnd 值的一半，即 6MSS，cwnd 值减半设为 6MSS，cwnd 值的增长与拥塞避免模式相似，呈线性增长。

另外，具体 TCP 实现会实现许多其它的拥塞控制算法，供用户选择。

4 其它传输层协议

下面对其它几种传输层协议作简要介绍：

●**IGMP**：Internet 组控制协议，它是组播组成员关系管理的基础，添加和删除组播成员的工作都由它完成。

●**SCTP**：流控制传输协议，此协议设计用于通过 IP 网络传输公共交换电话网络信令，也可用于其他应用。

●**DCCP**：数据报拥塞控制协议，它是一种不可靠的拥塞控制传输协议，它借鉴了 UDP 和 TCP，并添加了新的功能。DCCP 适合用于要求延迟较短但允许少量数据丢失的应用程序，如电话应用程序和流媒体应用程序。

12.4.4 网络层协议简介

本小节以 IPv4 协议为例，简要介绍网络层协议的内容。

1 IP 地址

因特网中通过 IP 地址来唯一标识主机和路由器网络端口，由于路由器每个端口都能接收和发送数据，因此每个端口都需要一个 IP 地址进行标识。每个主机和路由器端口可以设置多个 IP 地址。

在 IPv4 协议中用 32 位的无符号整数表示 IP 地址，而在 IPv6 协议中 IP 地址为 128 位。IPv4 中 32 比特的 IP 地址分为 4 个字节，通常按所谓**点分十进制记法**书写，即地址中的每个字节用它的十进制数形式书写，字节间用点隔开。例如，IP 地址 192.168.1.20，192 是该地址的第一个 8 比特的十进制等价数，依此类推。

IP 地址可根据类型和大小分组。大多数 IPv4 地址最终被细分为一个地址，用于识别连接因特网或某些专用的内联网的计算机接口。这些地址称为单播地址。IPv4 地址空间中大部分是单播地址空间。除了单播地址，其它类型的地址包括广播、组播和任播地址，还有一些特殊用途的地址。

■分类寻址

当最初定义因特网地址结构时，每个单播 IP 地址都有一个网络部分，用于识别接口使用的 IP 地址在哪个网络中被发现；以及一个主机地址，用于识别由网络部分给出的网络中的特定主机。因此，地址中的一些连续位称为网络号，其余位称为主机号。当时，大多数主机只有一个网络接口，因此术语接口地址和主机地址有时交替使用。

现实中的不同网络可能有不同数量的主机，每台主机都需要一个唯一的 IP 地址。一种划分方法是基于当前或预计的主机数量，将不同大小的 IP 地址空间分配给不同的站点。地址空间划分涉及五大类。每类都基于网络中可容纳的主机数量，确定在一个 32 位的 IPv4 地址中分配给网络号和主机号的位数。下图显示了这个基本思路：

5 个类被命名为 A、B、C、D 和 E。A、B 和 C 类空间用于单播地址。从上图可以看到不同类的相对大小，以及在实际使用中的地址范围。下表列出了各类地址范围以及网络数、每个网络中主机数量：

类	地址范围	高序位	用途	网络数	主机数
A	0.0.0.0~127.255.255.255	0	单播/特殊	128	16 777 216
B	128.0.0.0~191.255.255.255	10	单播/特殊	16 384	65 536
C	192.0.0.0~223.255.255.255	110	单播/特殊	2 097 152	256

D	224.0.0.0~239.255.255.255	1110	组播	--	--
E	240.0.0.0~255.255.255.255	1111	保留	--	--

该表显示了分类地址结构的主要使用方式，如何将不同大小的单播地址块分配给用户。A 类地址只有 128 个网络数，每个网络中可以有 2^{24} 个地址分配给主机。C 类地址共有 2^{21} 个网络号，但每个网络只能容纳 256 台主机。

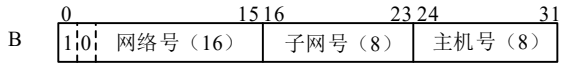
因特网地址分类方法在经历因特网增长第一个十年中没有变化，此后开始出现规模问题。当每个新的网段被添加到因特网时，集中协调为其分配一个新的 A、B 或 C 类地址变得很不方便。另外，A 和 B 类网络号通常浪费太多的主机号，而 C 类网络号不能为很多站点提供足够的主机号。

另外，IPv4 和 IPv6 地址空间中都包含了几个地址范围，它们被用于特殊用途。对于 IPv4，下表列出了部分特殊用途地址：

前缀	特殊用途
0.0.0.0/8	本地网络中的主机（前 8 位为 0），仅作为源 IP 地址使用。
10.0.0.0/8	专用网络（内联网）的地址，不会出现在公共因特网。
127.0.0.0/8	因特网主机回送地址（同一计算机），通常只有 **127.0.0.1。**
172.16.0.0/12	专用网络（内联网）地址，不会出现在公共因特网。
192.168.0.0/16	专用网络（内联网）地址，不会出现在公共因特网。
224.0.0.0/4	IPv4 组播地址，仅作为目的 IP 地址使用。
255.255.255.255/32	本地网络（受限）广播地址。

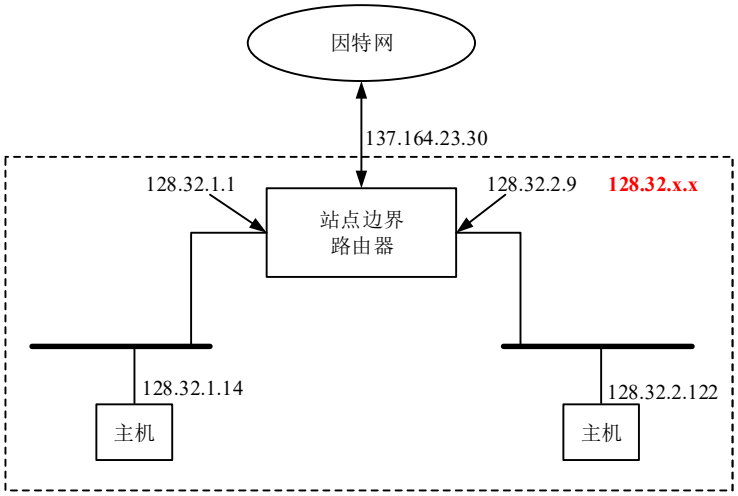
■子网寻址

为了解决为新接入因特网的网段分配一个新的网络号的难题，人们很自然想到的一个方式是对 A、B 和 C 类地址再进行进一步的划分。将其中的主机号进一步划分成子网号和主机号，下图示意了将一个 B 类地址划分子网的结构：



假设因特网某个站点（或 ISP）被分配了一个 B 类网络号。该站点将每个地址的前 16 位固定为分配的网络号，后 16 位由站点管理员按需分配。在这个例子中 8 位被选为子网号，剩下的 8 位为主机号，这个配置允许站点支持 256 个子网，每个子网最多可包含 254 台主机（0 和 255 主机号无效）。

下图示意了某个站点分配了一个 B 类网络号 128.32，站点划分了 8 位的主机号用于表示子网号，图中示意了 128.32.1.x 子网和 128.32.2.x 子网。



子网掩码是由一台主机或路由器使用的分配位，以确定如何从一台主机对应的 IP 地址中获得网络和子网信息。IP 子网掩码与对应的 IP 地址长度相同。它们通常在一台主机或路由器中以 IP 地址相同的方式配

置，既可以是动态分配的（DHCP），也可以是静态分配的。子网掩码由一些 1 后跟一些 0 构成，网络号和子网号对应的位用 1 标记，主机号用 0 标记，1 的位数表示前缀长度。

例如：128.32.1.x 子网的掩码为 255.255.255.0，前缀长度为 24，此子网记为 128.32.1.0/24，24 表示网络号加上子网号的长度，网络号、子网号为 128.32.1，主机号用 0 表示。某一 IP 地址与子网掩码相与的结果可确定此 IP 地址是否属于此子网。在路由选择查找中，可通过目的 IP 地址与子网掩码相与的结果确定数据包是否是发往本网络的数据包。

■CIDR

20 世纪 90 年代初，在采用子网录址缓解增长带来的痛苦后，因特网开始面临更严重的规模问题。为了帮助缓解 IPv4 地址（特别是 B 类地址）的压力，使用了被称为**无类别域间路由（CIDR）**的分配方案。这提供了一种方便的分配连续地址范围的方式。使用 CIDR，未经过预定义的任何地址范围可作为一个类的一部分，但需要一个类似于子网掩码的掩码，有时称为 CIDR 掩码。CIDR 掩码不再局限于一个站点，而对全球性路由系统都是可见的。因此，除了网络号之外，核心因特网路由器必须解释和处理掩码。这个数字组合称为网络前缀。

CIDR 消除了分类地址中的网络号（不按类分配地址）和主机号的预定义分隔，将更细粒度的 IP 地址分配范围成为可能。与分类寻址类似，地址空间分割成块最容易通过数值连续的地址来实现，以便用于某种类型或某些特殊用途。目前，这些分组普遍使用地址空间的前缀表示。一个 n 位的前缀是一个地址的前 n 个位的预定义值。对于 IPv4，n 的值通常在范围 0~32，它通常被追加到基本 IP 地址，并且后面跟着一上字“/”字符。例如：192.125.3.0/24 表示前缀为 24，基地址为 192.125.3.0。

通过取消分类结构的 IP 地址，能分配各种尺寸的 IP 地址块。但是，这增加了路由器中路由选择表项。在因特网中，可采用分层的路由思想以一种特定方式减少因特网路由表项数。这通过一个称为路由聚合的过程实现。通过将相邻的多个 IP 前缀合并成一个短前缀，可以覆盖更多地址空间。

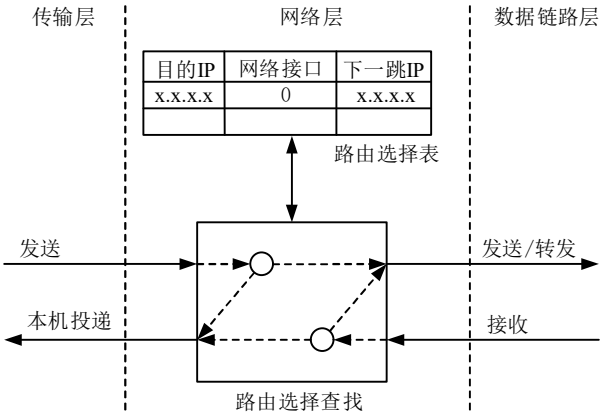
为了获取一块 IP 地址用于一个组织的子网内，某网络管理员也许首先会与他的 ISP 联系，该 ISP 可能会从已分给它的更大地址块中得到一个地址块，然后再从地址块中为主机分配一个地址。ISP 地址块可从非营利的 ICANN 组织申请分配。

某一组织一旦获得了一块地址，它就可以为本组织内的主机与路由器端口逐个分配 IP 地址。系统管理员通常手工配置路由器中的 IP 地址。主机地址也能手动配置，但是这项任务目前更多的是使用**动态主机配置协议（DHCP）**来完成，DHCP 允许主机自动获取一个 IP 地址。简单地说，就是子网中存在一个 DHCP 服务器负责管理和分配子网内主机的 IP 地址，新主机到达时向 DHCP 服务器发送消息，服务器为其动态分配 IP 地址。

2 功能概述

在因特网中，每台主机（路由器接口）至少需要分配一个 IP 地址，用于寻址主机（接口）。

网络层协议需要在主机和路由器中实现。网络层的主要工作是对到达网络层的数据包，包括本机传输层发送的数据包和数据链路层上传的数据包，根据其目的 IP 地址进行路由选择查找，确定数据包的下一步走向。经过路由选择查找，数据包可能需要投递到本机传输层（应用进程）或通过本机某个网络接口（网络设备）转发出去，又或者是将数据包丢弃等，如下图所示。



网络层需要配置一个路由选择表，每条路由选择表项中应当包含以下内容：

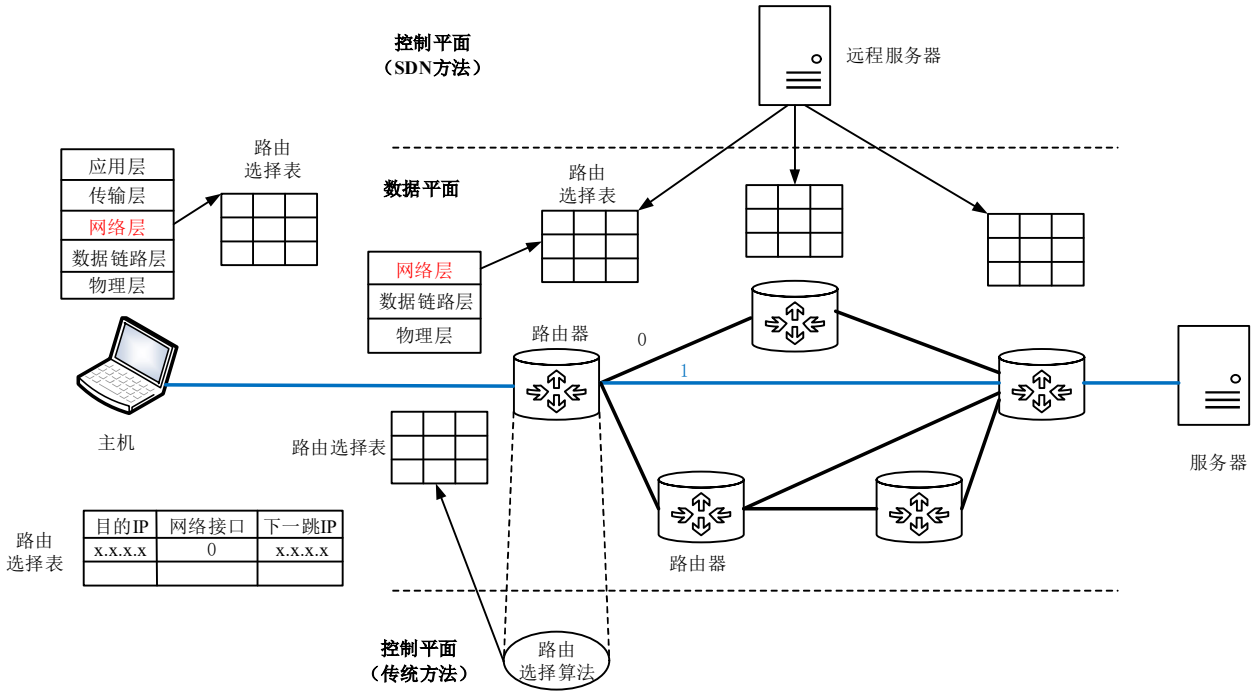
●**目的 IP 地址范围**：本表项匹配的目的 IP 地址范围，通常由一个 IP 地址和掩码组成。掩码由两部分组成，前半部分是若干个 1（前缀），后半部分是若干个 0。IP 地址与掩码相与的结果（忽略后半部分的 0）表示匹配前缀。如果数据包目的 IP 地址和掩码相与的结果同匹配前缀相等，则表示此表项匹配数据包目的 IP 地址。也就是说数据包目的 IP 地址前缀与表项的 IP 地址前缀相同，则表示是数据包匹配的表项，前缀的位数由掩码（前半部分 1 的个数）表示。

●**网络接口**：表示网络设备（接口）编号，匹配本表项的数据包由本接口发送出去。

●**下一跳 IP 地址**：表示与网络接口相连的下一跳机器的 IP 地址。

路由选择查找即根据数据包目的 IP 地址在路由选择表中查找匹配的表项，匹配表项指示了数据包下一个目的地。本机 IP 地址在路由选择表中也有相应的表项，匹配此表项的数据包投递到本主机传输层。

网络层协议需要在端系统（主机）和路由器中实现，如下图所示：



主机（端系统）网络层路由选择表通常由用户配置。

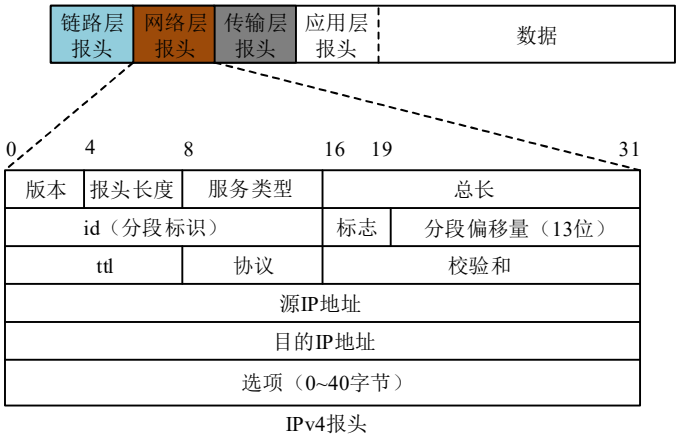
路由器中路由选择表设置要复杂一些，复杂性主要表现在路由选择表转发策略的确定。路由器网络层的控制平面用于确定路由选择表。控制平面有两种方法，传统方法和 SDN 方法。传统方法是在每个路由器中运行一个路由选择算法，路由器只与相邻的路由器通信，确定数据包转发策略，以此填充本路由器的路由选择表。路由器以此路由选择表执行数据包的转发操作。

SDN 方法是指网络中存在一个远程服务器，各路由器与此远程服务器相连，由远程服务器确定各路由器的转发策略，并填充各路由器的路由选择表。

3 网络层报头

IPv4 数据包的开头是一个 IP 报头，其长度不少于 20 字节，如果使用了 IP 选项，IPv4 报头最长可达 60 字节。要理解网络层协议，必须对网络层报头各字段语义了解清楚。

IPv4 网络层报头格式如下图所示：



IPv4报头

报头中各字段语义如下：

●**版本**：版本号，4bit，此处必须为 4，IPv6 报头格式与此不同。

●**报头长度**：4bit，表示报头以 4 字节为单位的长度，如果没有使用 IP 选项，则报头长度为 20 字节，此字段值为 5。报头长度最大值为 60 字节（报头长度值为 15）。

●**服务类型**：8bit，服务类型（TOS）表示 IP 数据包的类型，如实时数据包、非实时数据包等。

●**总长**：16bit，数据包总长度（首部加上数据），以字节计。由于其长度为 16bit，因此数据包的理论最大长度为 65535 字节。然而，数据包很少有超过 1500 字节的，因为数据链路层数据包的最大长度值（MTU）通常不会超过 1500 字节。

●**标识（id）**：标识，对于分段来说，id 字段很重要。分段是指网络层数据包的长度大于数据链路层帧的最大长度，网络层需要在发往数据链路层前需要将数据包进行分段，在接收端的网络层需要对分段进行重组。对于分段后的数据包，用 id 值标识属于同一个原数据包的分段。

●**标志**：3bit，标志字段各标志位语义如下：

001：表示本分段后面还有其它的分段（More Fragments，MF），除最后一个分段外，其它的分段都必须设置这个标志。最后一个分段此标志设为 000。

010：表示不可以分段（Don’t Fragments，DF）。

100：表示拥塞（Congestion，CE）。

●**分段偏移量**：13bit，表示分段在原始数据包中的偏移量，偏移量以 8 字节为单位，第一个分段偏移量为 0。

●**ttl**：8bit，寿命（存活时间），用来确保数据包不会被无休止地传输。在主机发送数据包时会对该字段设置一个初始值，数据包每经过一个路由器，该字段值会减 1，当值为 0 时，此数据包将会被丢弃，并向发送端发回一条 ICMPv4“超时”信息（在网络层协议中实现）。

●**协议**：8bit，数据包使用的传输层协议，如 IPPROTO_TCP、IPPROTO_UDP 等。

●**校验和**：16bit，仅根据报头计算的校验和，每次报头数据改变后需要重新计算校验和。

●**源 IP 地址**：32bit，发送主机的 IP 地址。

●**目的 IP 地址**：32bit，接收主机 IP 地址。发送主机通常通过 DNS 查找来决定目的主机，DNS 查找是通过 DNS 服务器将域名转换成 IP 地址。

源 IP 地址和目的 IP 地址是指末端主机的 IP 地址，而不是中间路由器接口的 IP 地址，这两个 IP 地址在数据包转发过程中是不会改变的。

●**选项**：0~40 字节，选项字段允许 IP 报头被扩展，选项字段很少使用，因此本书暂不介绍。在 IPv6 报头中不支持选项字段。

4 分段与重组

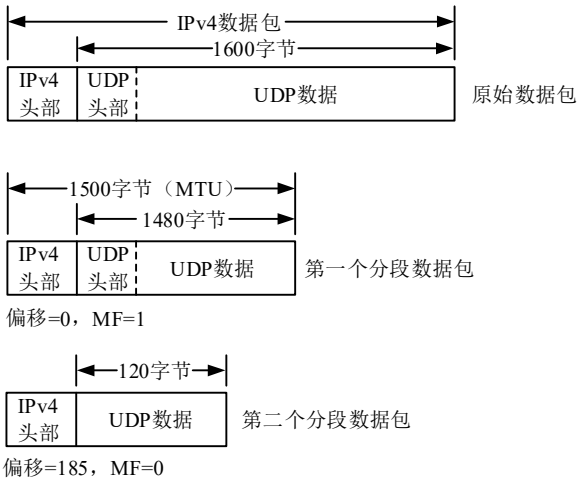
链路层通常对可传输的每个帧的最大长度（MTU）有一个上限。为了保持 IP 数据包抽象与链路层细节的一致和分离，IP 引入了数据包分段与重组。当 IP 层接收到一个需要发送的 IP 数据包时，它会判断该数据包应该从哪个本地接口发送以及要求的 MTU 是多少。IP 比较外出接口的 MTU 值和数据包的长度值，如果数据包长度大于 MTU 值，则需要对数据包进行分段。IPv4 中的分段可以在原始的发送方主机和路径中任一路由器中进行，但是只在最终目的端系统中才会被重组，也就是说在路由器中不会进行重组。IPv6 中只允许在源主机中分段，不允许在路由器中分段。

只允许在最终端系统中重组的原因有二个，一是在网络中不进行重组要比重组更能减轻路由器转发软件的负担，二是分段后的数据包可能经过不同的路径（路由器）到达目的地，因此中间路由器通常没有能力来重组数据包。

分段只对 IP 数据包的数据进行分段，也就是对数据包中除 IP 报头的数据进行分段，每个分段都要添加上一个 IP 报头，形成一个完整的 IP 数据包。

原始 IP 报头中的标识字段（由原始发送方设置）将被复制到同一个数据包的各个分段数据包报头中，用于标识属于同一个数据包的分段，以便于重组。分段偏移量给出该分片负载字节中的第一个字节在原始 IPv4 数据包中的偏移量（8 字节对齐），第一个分段偏移量为 0。除最后一个分段数据包外，所有分段数据包报头会设置标记位 MF，表示后面还有分段数据包，最后一个数据包 MF 标记位为 0，表示后面没有数据包了。

下图示意了网络层协议对一个数据长度为 1600 字节的原始数据包的分段结果，MTU 为 1500 字节：



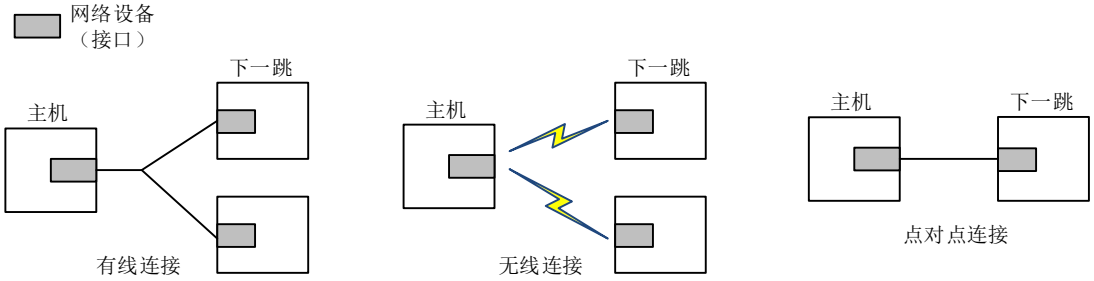
由于 IPv4 报头占用了 20 字节（没有选项），因此第一个分段数据包的数据长度只有 1480 字节，第二个分段数据包的偏移量为 185（185*8=1480）。UDP 头部只存在于第一个分段数据包中，每个分段数据包都有 IPv4 报头（来源于原始数据包，长度有变化，需要重新计算校验和）。

端系统在重组数据包前会将接收到的数据包按序排列（偏移量）。端系统在收到第一个分段数据包时，将会启动一个定时器，如果定时器超时还没有收到所有分段数据包，将触发超时重传。如果有一个分段数据包丢失了，整个数据包将被丢弃，数据包传输失败，需要重传整个数据包。

当有所有的分段数据包都正确接收后，IP 将会重组原始数据包，发送给传输层。

12.4.5 数据链路层协议简介

主机通过网络接口（网络设备）、通信链路与下一跳设备连接，以实现相邻节点之间的数据传输。如下图所示，主机网络接口可通过有线、无线通信链路同时与多个下一跳（机器）相连，或点对点与下一跳相连。

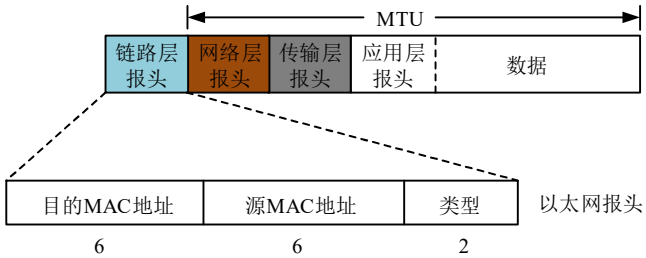


相邻节点之间通过不同的通信介质和形式相连，数据在相邻节点之间的传输方式，传输协议也必然不同。数据链路层协议就是用于定义相邻节点之间的通信协议，不同的传输介质，传输形式有不同的通信协议。

常用的数据链路层协议有以太网、WiFi 等。以太网是有线传输、WiFi 是无线传输。数据链路层协议定义了数据包在链路上传输的数据格式，节点之间数据传输的协调方式等。通信链路上传输的数据包通常称为链路层帧。除此之外，如果多个节点共用传输链路，还需要协议节点之间的数据包传输，以避免相互干扰和碰撞。

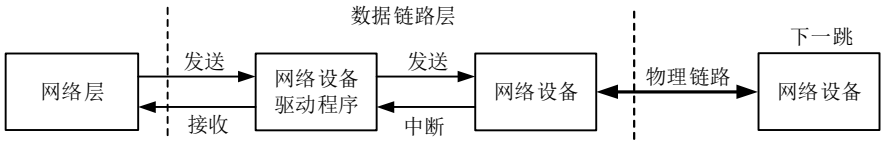
数据链路层协议主要由网络设备实现。主机只需要将要发送的数据包交给网络设备即可，由其发送到通信链路上。网络设备接收到数据包后，产生中断，主机从指定地址读取数据即可。

数据链路层中的网络设备具有唯一的硬件地址（物理地址，MAC 地址）用于标识网络设备，相当于总线上的设备地址。在数据链路层报头中需要加入源主机和目的主机网络设备的物理地址。例如，下图为以太网报头（基本型）：



以太网设备物理地址由 6 字节表示，称为 MAC 地址。报头中包含目的 MAC 地址、源 MAC 地址和类型（2 字节），类型表示网络层协议类型。

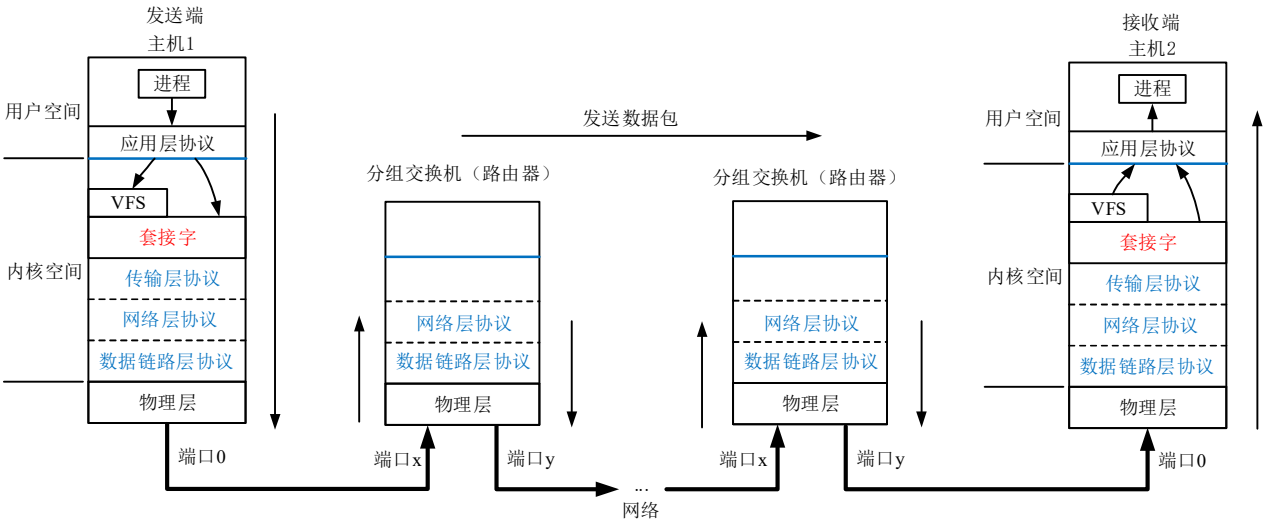
网络设备驱动程序在数据链路层实现，主要是实现对网络设备的控制，以及数据的传输等。网络设备负责将网络设备驱动程序下发的数据包按照数据链路层协议格式要求将数据包发送到物理链路上，并从物理链路上接收数据包，以中断的方式通知网络设备驱动程序接收数据包。



12.5 IPv4 套接字

从本节开始，本章将介绍 TCP/IP 协议簇在 Linux 内核中的实现。在 Linux 内核中用户进程通过套接字访问网络，套接字是进程访问网络的接口，套接字关联到网络协议，由网络协议实现数据包的传输。用户进程并不需要关注网络操作的细节。

下图示意了用户数据包在内核网络协议层（网络协议栈）中的传输流程：



图中所示主机 1 进程在向主机 2 进程发送数据，主机 1 和主机 2 通过套接字关联网络协议，通过网络协议传输网络数据。

12.5.1 地址与协议类型

TCP/IP 协议簇作为套接字的一个通信域（协议簇、地址簇），用于网络通信。因特网 IPv4 地址簇（协议簇）标识为 **AF_INET**（IPv6 为 AF_INET6），地址值由 sockaddr_in 结构体表示（/include/uapi/linux/in.h）：

```
struct sockaddr_in {
    __kernel_sa_family_t    sin_family;    /*AF_INET，地址簇标识*/
    __be16                  sin_port;      /*传输层中的端口号（标识套接字）*/
    struct in_addr          sin_addr;      /*网络层中的 IP 地址，32bit（标识主机）*/

    /*补齐*/
    unsigned char    __pad[__SOCK_SIZE__ - sizeof(short int) -
                           sizeof(unsigned short int) - sizeof(struct in_addr)];
};
```

IPv4 地址值主要由 32 位的 IP 地址和 16 位的端口号组成，IP 地址表示主机在网络中的编址，端口号表示主机内通信方（套接字）的编号，例如：192.168.1.8：80 表示 IP 地址为 192.168.1.8 计算机上的 Web 服务器（80 为周知端口号）。

在创建网络套接字时（IPv4），传输层协议作为 IPv4 协议簇下的协议类型，定义如下：

```
enum {                  /*/include/uapi/linux/in.h*/
  IPPROTO_IP = 0,           /* Dummy protocol for TCP*/
#define IPPROTO_IP          IPPROTO_IP
  IPPROTO_ICMP = 1,         /* Internet Control Message Protocol，Internet 控制消息协议*/
#define IPPROTO_ICMP        IPPROTO_ICMP
  IPPROTO_IGMP = 2,         /* Internet Group Management Protocol，Internet 组管理协议*/
#define IPPROTO_IGMP        IPPROTO_IGMP
  IPPROTO_IPIP = 4,         /* IPIP tunnels (older KA9Q tunnels use 94) */
#define IPPROTO_IPIP        IPPROTO_IPIP
  IPPROTO_TCP = 6,          /*TCP 协议*/
#define IPPROTO_TCP IPPROTO_TCP
  IPPROTO_EGP = 8,          /* Exterior Gateway Protocol */
```

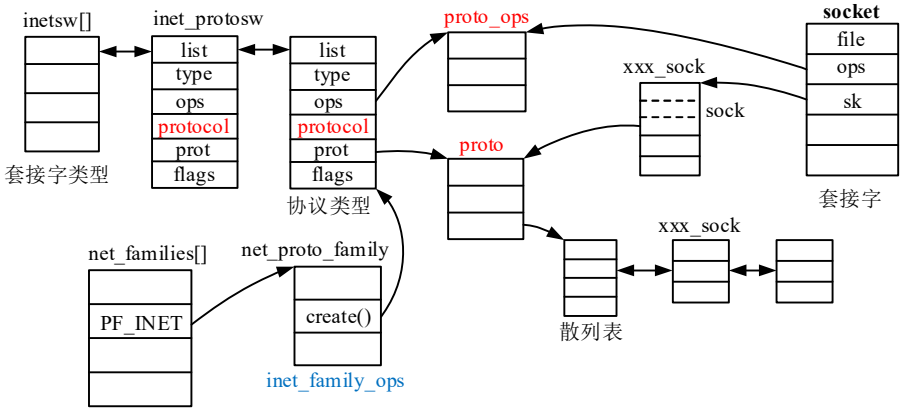
```
#define IPPROTO_EGP       IPPROTO_EGP
  IPPROTO_PUP = 12,       /* PUP protocol*/
#define IPPROTO_PUP       IPPROTO_PUP
  IPPROTO_UDP = 17,       /*UDP 协议*/
#define IPPROTO_UDP  IPPROTO_UDP
  IPPROTO_IDP = 22,       /* XNS IDP protocol*/
#define IPPROTO_IDP       IPPROTO_IDP
  IPPROTO_TP = 29,        /* SO Transport Protocol Class 4   */
#define IPPROTO_TP        IPPROTO_TP
  IPPROTO_DCCP = 33,      /*数据报拥塞控制协议*/
#define IPPROTO_DCCP      IPPROTO_DCCP
  IPPROTO_IPV6 = 41,      /* IPv6-in-IPv4 tunnelling*/
#define IPPROTO_IPV6      IPPROTO_IPV6
  IPPROTO_RSVP = 46,      /* RSVP Protocol*/
#define IPPROTO_RSVP      IPPROTO_RSVP
  IPPROTO_GRE = 47,       /* Cisco GRE tunnels (rfc 1701,1702)*/
#define IPPROTO_GRE       IPPROTO_GRE
  IPPROTO_ESP = 50,       /* Encapsulation Security Payload protocol */
#define IPPROTO_ESP       IPPROTO_ESP
  IPPROTO_AH = 51,        /* Authentication Header protocol */
#define IPPROTO_AH        IPPROTO_AH
  IPPROTO_MTP = 92,       /* Multicast Transport Protocol*/
#define IPPROTO_MTP       IPPROTO_MTP
  IPPROTO_BEETPH = 94,    /* IP option pseudo header for BEET*/
#define IPPROTO_BEETPH    IPPROTO_BEETPH
  IPPROTO_ENCAP = 98,     /* Encapsulation Header */
#define IPPROTO_ENCAP     IPPROTO_ENCAP
  IPPROTO_PIM = 103,      /* Protocol Independent Multicast  */
#define IPPROTO_PIM       IPPROTO_PIM
  IPPROTO_COMP = 108,     /* Compression Header Protocol*/
#define IPPROTO_COMP      IPPROTO_COMP
  IPPROTO_SCTP = 132,     /*流控制传输协议*/
#define IPPROTO_SCTP      IPPROTO_SCTP
  IPPROTO_UDPLITE = 136,  /* UDP-Lite (RFC 3828)*/
#define IPPROTO_UDPLITE   IPPROTO_UDPLITE
  IPPROTO_MPLS = 137,     /* MPLS in IP (RFC 4023)    */
#define IPPROTO_MPLS      IPPROTO_MPLS
  IPPROTO_RAW = 255,      /*原始 IP 数据包，没有传输层报头*/
#define IPPROTO_RAW       IPPROTO_RAW
  IPPROTO_MAX             /*256，最大传输层协议类型*/
};
```

IPv4 协议簇最多支持 256 种传输层协议类型，本章主要以 UDP、TCP 为例介绍传输层协议的实现。

12.5.2 套接字框架

在网络套接字中（IPv4），多种传输层协议（协议类型）可使用同一种类型的套接字（如流套接字、

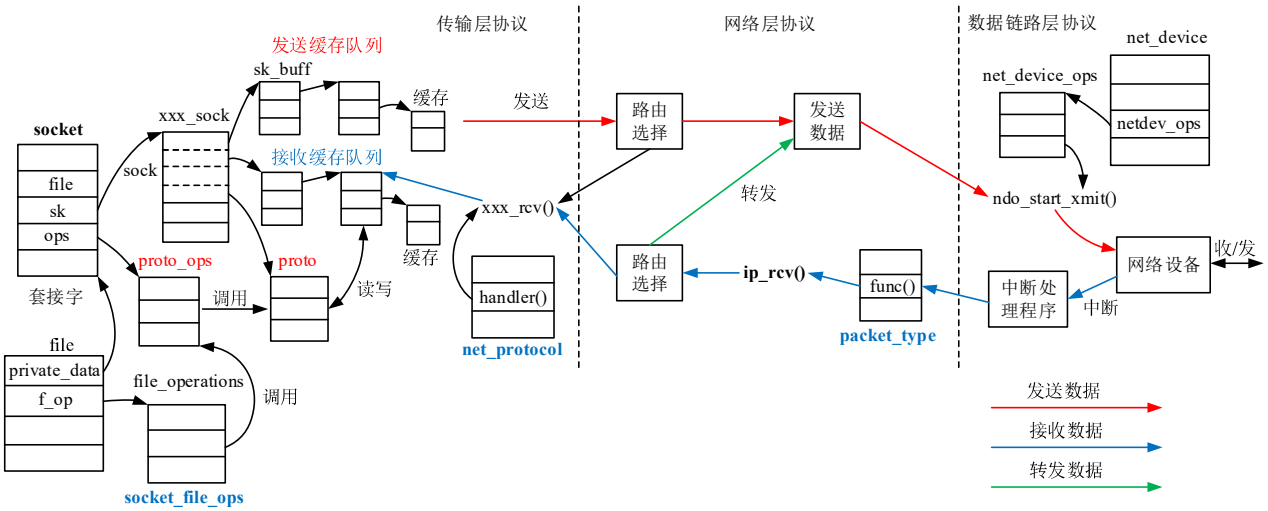
数据报套接字等）。每种传输层协议类型需要定义并注册 inet_protosw 结构体实例，此实例需关联 proto_ops 和 proto 结构体实例，这两个实例也由传输层协议实现，用于设置套接字。内核为每种套接字类型定义了双链表（双链表数组），用于管理属于同一套接字类型下的传输层协议注册的 inet_protosw 实例，如下图所示。



IPv4 协议簇对应的 net_proto_family 结构体实例 inet_family_ops 中的 create()函数，在创建套接字时用于设置套接字，此函数内将查找协议类型对应的 inet_protosw 实例，将实例关联的 proto_ops、proto 实例赋予 socket 及其关联的 sock 实例。

在传输层协议中套接字通常由 xxx_sock 结构体表示，结构体定义取决于具体的传输层协议类型（传输层协议定义的 proto 实例中指定了 xxx_sock 结构体大小），xxx_sock 结构体中内嵌 sock 结构体成员。传输层协议类型定义的 proto 实例中通过散列表等数据结构管理 xxx_sock 实例。

IPv4 网络套接字数据包发送接收流程简列如下图所示：



传输层表示套接字的 xxx_sock 结构体（内嵌的 sock 结构体成员）中包含发送和接收缓存队列。缓存队列中数据包由 sk_buff 结构体表示，其关联的内存缓存区保存了数据包的数据。

进程通过网络套接字发送数据包时，将会依次调用 proto_ops 和 proto 结构体中的 sendmsg()函数，此函数最终调用传输层协议定义的函数将数据包发往网络层。网络层根据目的 IP 地址执行路由选择，确定数据包的下一步走向，如果是发送回本机，则回传给传输层。如果需要外发，则调用网络设备操作结构中的发送函数，将数据包传递给网络设备，由网络设备将数据包发送到通信链路上。

在接收路径上，网络层协议需要定义并注册 packet_type 结构体实例，传输层协议需要定义并注册 net_protocol 结构体实例，其中的操作函数用于接收数据包。网络设备从通信链路上接收到数据包后，将产生中断，在中断处理函数中将构建 sk_buff 实例，根据数据链路层报头中指示的网络层协议类型调用对应 packet_type 实例中的 func()函数接收数据包。对于 IPv4 协议簇此函数为 **ip_rcv()**，此函数是一个非常复杂的函数，函数内需要根据目的 IP 地址执行路由选择查找，如果需要转发数据包则从查找结果中指示的网络设备将数据包转发出去。如果是发送给本机的数据包，则调用传输层协议类型注册的 net_protocol 实例中

的 handler()函数接收数据包，此函数内将根据传输层报头中目的端口号等信息查找目的进程套接字（xxx_sock 实例），将 sk_buff 实例添加到此套接字接收缓存队列，进程可从此接收缓存队列读取数据。

12.5.3 数据结构

本小节介绍 IPv4 网络套接字及网络协议中主要数据结构的定义。

1 inet_protosw

每种传输层协议需要定义并注册 inet_protosw 结构体实例，用于在创建套接字时设置 socket 和 xxx_sock 实例，结构体定义如下（/include/net/protocol.h）：

```
struct inet_protosw {
    struct list_head    list;     /*双链表成员*/
    unsigned short   type;        /*套接字类型（socket()系统调用参数）*/
    unsigned short   protocol;    /*协议类型（传输层协议类型）*/
    struct proto        *prot;        /*指向 proto 实例，赋予 sock 实例*/
    const struct proto_ops *ops;      /*指向 proto_ops 实例，赋予 socket 实例*/
    unsigned char    flags;         /*标记，INET_PROTOSW_ICSK 等，/include/net/protocol.h*/
};
```

各传输层协议代码中需要定义 inet_protosw 实例及其关联的 proto_ops 和 proto 实例，并调用接口函数 **inet_register_protosw**(struct inet_protosw *p)注册实例。

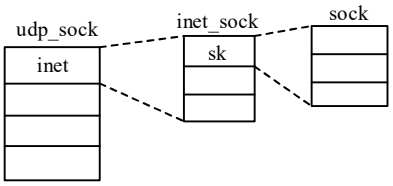
内核定义了全局双链表数组用于管理 inet_protosw 实例（/ipv4/af_inet.c）：

```
static struct list_head inetsw[SOCK_MAX];
```

每种套接字类型对应双链表数组中一个数组项（双链表），SOCK_MAX 为套接字类型最大值，也就是说同一套接字类型下的传输层协议定义的 inet_protosw 实例由同一个双链表管理。

2 inet_sock

各传输层协议实现代码中定义了表示本协议类型套接字的私有数据结构，通常形式为 xxx_sock，此结构体中内嵌 inet_sock 结构体，这是一个通用的数据结构，表示 IPv4 套接字的一些通用信息。例如，UDP 套接字由 udp_sock 结构体表示，结构体中第一个成员为 inet_sock 结构体，如下图所示：



inet_sock 结构体中内嵌了表示套接字的通用 sock 结结体成员、源套接字地址信息、目的套接字地址信息等。inet_sock 结构体定义如下（/include/net/inet_sock.h）：

```
struct inet_sock {
    struct sock          sk;        /*sock 结构体成员*/
#if IS_ENABLED(CONFIG_IPV6)
    struct ipv6_pinfo    *pinet6;    /*指向 IPv6 控制块*/
#endif
    #define   inet_daddr          sk.__sk_common.skc_daddr     /*目的主机 IP 地址，网络字节序*/
    #define   inet_rcv_saddr      sk.__sk_common.skc_rcv_saddr     /*本主机 IP 地址，网络字节序*/
    #define   inet_dport          sk.__sk_common.skc_dport      /*目的端口号，网络字节序*/
```

```
    #define   inet_num          sk.__sk_common.skc_num      /*本套接字端口号，本机字节序*/

    __be32        inet_saddr;     /*本主机 IP 地址，网络字节序*/
    __s16         uc_ttl;         /*单播 ttl 值*/
    __u16         cmsg_flags;     /*控制消息标记*/
    __be16        inet_sport;      /*源端口号，网络字节序*/
    __u16         inet_id;        /*分段数据包 id 值，初始值为 0，发送数据包时会重新赋值*/

    struct ip_options_rcu __rcu   *inet_opt;      /*指向 IP 选项*/
    int           rx_dst_ifindex;
    __u8          tos;        /*服务类型*/
    __u8          min_ttl;
    __u8          mc_ttl;       /*组播 ttl*/
    __u8          pmtudisc;     /*路径 MTU 探测标记，如 IP_PMTUDISC_DO，/include/uapi/linux/in.h*/
    __u8          recverr:1,
                  is_icsk:1,    /*是否是已连接套接字*/
                  freebind:1,
                  hdrincl:1,
                  mc_loop:1,
                  transparent:1,
                  mc_all:1,
                  nodefrag:1;
    __u8          bind_address_no_port:1;
    __u8          rcv_tos;
    __u8      convert_csum;
    int       uc_index; /*单播网络设备编号，可通过 setsockopt()系统调用设置，其它成员也可设置*/
    int       mc_index;     /*组播网络设备编号*/
    __be32    mc_addr;      /*组播源地址*/
    struct ip_mc_socklist __rcu   *mc_list;     /*组播组*/
    struct inet_cork_full      cork;   /*由用户数据构建数据包队列时所需的信息（启用抑制功能时）*/
};
```

inet_sock 结构体部分成员简介如下：

●**sk**：内嵌 sock 结构体成员。

●**cork**：inet_cork_full 结构体成员，结构体定义如下（/include/net/inet_sock.h）：

```
struct inet_cork_full {
    struct inet_cork      base;      /*将用户数据转成数据包队列时，保存过程信息的结构体*/
    struct flowi          fl;     /*路由查找结果*/
};
```

inet_cork 结构体在将用户数据转换成数据包队列的过程中使用，定义如下（/include/net/inet_sock.h）：

```
struct inet_cork {
    unsigned int          flags;
    __be32                addr;      /*目的 IP 地址*/
    struct ip_options     *opt;      /*IP 选项*/
    unsigned int          fragsize;   /*分段长度*/
    int                   length;       /*累积的用户数据长度，不能超过数据报长度（64KB）*/
```

```
    struct dst_entry      *dst;      /*路由选择查找结果*/
    u8                tx_flags;      /*发送标记*/
    __u8                  ttl;       /*ttl*/
    __s16                 tos;       /*服务类型*/
    char              priority;      /*优先级*/
};
```

在启用了抑制功能时，在将用户数据转换成数据包队列并添加到套接字发送缓存队列过程中，需要使用此处 inet_cork 实例中的参数。

3 net_protocol

每种**传输层协议**需要定义并注册 net_protocol 结构体实例，用于接收数据包，结构体定义如下：

```
struct net_protocol {                /*/include/net/protocol.h*/
    void            (*early_demux)(struct sk_buff *skb);
    int             (*handler)(struct sk_buff *skb);       /*接收处理数据包函数*/
    void            (*err_handler)(struct sk_buff *skb, u32 info);     /*处理 ICMP 错误消息*/
    unsigned int    no_policy:1,
                    netns_ok:1,
                    icmp_strict_tag_validation:1;
};
```

net_protocol 结构体主要成员简介如下：

●**handler**：传输层协议接收处理网络层数据包的函数。

●**err_handler**：在接收到 ICMP 错误信息并需要将错误信息传递到更高层时调用此函数。

●**no_policy**：标记设置表示不需要执行 IPsec 策略检查。

●**netns_ok**：标记设置表示协议支持网络命名空间。

内核定义了全局指针数组 inet_protos[MAX_INET_PROTOS]，用于管理 net_protocol 实例，数组项数为 256，数组项索引值表示传输层协议类型。

注册 net_protocol 实例的接口函数为 **inet_add_protocol**(const struct net_protocol *prot, unsigned char protocol)，函数内只是简单地将实例关联到协议类型对应的 inet_protos[]数组项。

4 packet_type

每种**网络层协议**需要定义并注册 packet_type 结构体实例，用于接收数据链路层上传的数据包，结构体定义如下（/include/linux/netdevice.h）：

```
struct packet_type {
    __be16              type;          /*以太网协议类型，大端序*/
    struct net_device   *dev;          /*指向网络设备 net_device 实例*/
    int    (*func) (struct sk_buff *,struct net_device *,struct packet_type *,struct net_device *);
                                                                  /*接收处理数据包函数*/
    bool   (*id_match)(struct packet_type *ptype,struct sock *sk);
    void   *af_packet_priv;
    struct list_head    list;          /*双链表成员*/
};
```

packet_type 结构体主要成员简介如下：

●**type**：以太网协议类型，取值定义在/include/uapi/linux/if_ether.h 头文件内，例如：

```
#define   ETH_P_LOOP      0x0060      /* Ethernet Loopback packet，环回数据包*/
#define   ETH_P_PUP       0x0200      /* Xerox PUP packet*/
#define   ETH_P_PUPAT     0x0201      /* Xerox PUP Addr Trans packet*/
#define   ETH_P_IP        0x0800      /*以太网 IPv4 数据包*/
...
#define   ETH_P_IPV6      0x86DD      /*以太网 IPv6 数据包*/
...
```

以上宏表示数值是主机字节序，要写入数据包报头，或与数据包报头中字段比较时，需要转为大端序。

●**dev**：指向网络设备 net_device 实例，将 packet_type 绑定到特定的网络设备，为 NULL 表示 packet_type 可用于所有网络设备。

●**func**：网络层协议接收处理数据链路层上传数据包的函数，IPv4 为 **ip_rcv()**。

●**list**：双链表成员，用于将实例插入到管理结构中。

注册 packet_type 实例的接口函数为 **dev_add_pack**()，定义如下（/net/core/dev.c）：

```
void dev_add_pack(struct packet_type *pt)
{
    struct list_head *head = ptype_head(pt);      /*确定添加的双链表头*/

    spin_lock(&ptype_lock);
    list_add_rcu(&pt->list, head);      /*添加到 head 双链表中*/
    spin_unlock(&ptype_lock);
}
```

内核定义了全局双链表 ptype_all、散列表 ptype_base[]用于管理 packet_type 实例。网络设备 net_device 结构体中 ptype_all 和 ptype_specific 双链表成员也用于管理 packet_type 实例。

ptype_head(pt)函数用于确定将 packet_type 实例添加到哪个双链表，函数定义如下（/net/core/dev.c）：

```
static inline struct list_head *ptype_head(const struct packet_type *pt)
{
    if (pt->type == htons(ETH_P_ALL))     /*packet_type 实例适用于所有以太网协议类型*/
        return pt->dev ? &pt->dev->ptype_all : &ptype_all;
    else
        return pt->dev ? &pt->dev->ptype_specific :
                    &ptype_base[ntohs(pt->type) & PTYPE_HASH_MASK];
}
```

由以上函数可知，全局双链表 ptype_all 和 net_device->ptype_all 双链表用于管理适用于所有以太网协议类型的 packet_type 实例。如果 packet_type 实例指定了绑定的网络设备则将实例添加到 net_device 实例中的 ptype_all 双链表，否则添加到全局双链表 ptype_all。

ptype_base[]散列表和 net_device->ptype_specific 双链表用于管理特定于以太网协议的 packet_type 实例。如果 packet_type 实例指定了绑定的网络设备则添加到 net_device 实例中的 ptype_specific 双链表，否则添加到 ptype_base[]散列表。

5 netns_ipv4

在网络命名空间 net 结构体中的 netns_ipv4 结构体成员记录了 IPv4 网络资源（参数），如下所示：

```
struct net {
    ...
```

```
    struct netns_ipv4    ipv4;      /*IPv4 网络协议资源，路由选择表、邻居表等*/
    ...
}
```

netns_ipv4 结构体定义如下（/include/net/netns/ipv4.h）：

```
struct netns_ipv4 {
  #ifdef CONFIG_SYSCTL
    struct ctl_table_header   *forw_hdr;
    struct ctl_table_header   *frags_hdr;
    struct ctl_table_header   *ipv4_hdr;
    struct ctl_table_header   *route_hdr;
    struct ctl_table_header   *xfrm4_hdr;
  #endif
    struct ipv4_devconf  *devconf_all;     /*指向网络设备配置信息*/
    struct ipv4_devconf  *devconf_dflt;    /*指向网络设备配置信息*/
  #ifdef CONFIG_IP_MULTIPLE_TABLES     /*支持策略路由选择，支持多个路由选择表*/
    struct fib_rules_ops*rules_ops;
    bool              fib_has_custom_rules;
    struct fib_table __rcu     *fib_local;      /*路由选择表*/
    struct fib_table __rcu     *fib_main;
    struct fib_table __rcu     *fib_default;
  #endif
  #ifdef CONFIG_IP_ROUTE_CLASSID
    int          fib_num_tclassid_users;
  #endif
    struct hlist_head     *fib_table_hash;     /*指向管理路由选择表的散列表*/
    bool              fib_offload_disabled;
    struct sock       *fibnl;         /*路由选择表的 netlink 套接字*/

    struct sock    * __percpu *icmp_sk;     /*指向 ICMP 套接字，percpu 变量*/
    struct sock    *mc_autojoin_sk;

    struct inet_peer_base     *peers;
    struct sock    * __percpu   *tcp_sk;
    struct netns_frags    frags;       /*数据包分段参数，/include/net/inet_frag.h*/
  #ifdef CONFIG_NETFILTER        /*支持网络过滤*/
    struct xt_table        *iptable_filter;
    struct xt_table        *iptable_mangle;
    struct xt_table        *iptable_raw;
    struct xt_table        *arptable_filter;
    #ifdef CONFIG_SECURITY
      struct xt_table     *iptable_security;
    #endif
    struct xt_table        *nat_table;
  #endif

    int sysctl_icmp_echo_ignore_all;
```

```
    int sysctl_icmp_echo_ignore_broadcasts;
    int sysctl_icmp_ignore_bogus_error_responses;
    int sysctl_icmp_ratelimit;
    int sysctl_icmp_ratemask;
    int sysctl_icmp_errors_use_inbound_ifaddr;

    struct local_ports ip_local_ports;

    int sysctl_tcp_ecn;
    int sysctl_tcp_ecn_fallback;

    int sysctl_ip_no_pmtu_disc;
    int sysctl_ip_fwd_use_pmtu;
    int sysctl_ip_nonlocal_bind;

    int sysctl_fwmark_reflect;
    int sysctl_tcp_fwmark_accept;
    int sysctl_tcp_mtu_probing;
    int sysctl_tcp_base_mss;
    int sysctl_tcp_probe_threshold;
    u32 sysctl_tcp_probe_interval;

    struct ping_group_range    ping_group_range;

    atomic_t dev_addr_genid;       /*网络设备随机数*/

#ifdef CONFIG_SYSCTL
    unsigned long *sysctl_local_reserved_ports;
#endif

#ifdef CONFIG_IP_MROUTE     /*当前机器配置为组播路由器*/
    #ifndef CONFIG_IP_MROUTE_MULTIPLE_TABLES
        struct mr_table         *mrt;
    #else
        struct list_head  mr_tables;
        struct fib_rules_ops  *mr_rules_ops;
    #endif
#endif

    atomic_t  rt_genid;
};
```

网络命名空间 net 实例中的 netns_ipv4 成员在 IPv4 协议簇初始化，以及 IPv4 协议簇各子系统初始化时设置。

12.5.4 IPv4 协议簇初始化

本小节介绍 IPv4 协议簇中各层协议在内核实现的初始化函数，以及相关系统控制参数的定义和注册。

1 初始化函数

IPv4 协议簇初始化函数为 inet_init()，初始化函数中包含了 IPv4 协议簇传输层、网络层协议的初始化工作等，函数代码简列如下（/net/ipv4/af_inet.c）：

```
static int __init inet_init(void)
{
    struct inet_protosw *q;
    struct list_head *r;
    int rc = -EINVAL;

    sock_skb_cb_check_size(sizeof(struct inet_skb_parm));

    rc = proto_register(&tcp_prot, 1);   /*注册 TCP 对应 proto 实例，/net/ipv4/tcp_ipv4.c*/
    ...
    rc = proto_register(&udp_prot, 1);    /*注册 UDP 对应 proto 实例，/net/ipv4/udp.c*/
    ...
    rc = proto_register(&raw_prot, 1);   /*注册原始套接字对应 proto 实例，/net/ipv4/raw.c*/
    ...
    rc = proto_register(&ping_prot, 1);   /*注册 ping 套接字对应 proto 实例，/net/ipv4/ping.c*/
    ...

    (void)sock_register(&inet_family_ops);    /*注册 IPv4 协议簇 net_proto_family 实例*/

#ifdef CONFIG_SYSCTL
    ip_static_sysctl_init();    /*注册控制参数 ipv4_route_table 列表，/net/ipv4/route.c*/
#endif

    /*注册传输层协议对应的 net_protocol 实例，用于接收数据包*/
    if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)      /*ICMP，/net/ipv4/af_inet.c*/
        pr_crit("%s: Cannot add ICMP protocol\n", __func__);
    if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)      /*UDP*/
        pr_crit("%s: Cannot add UDP protocol\n", __func__);
    if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)      /*TCP*/
        pr_crit("%s: Cannot add TCP protocol\n", __func__);
#ifdef CONFIG_IP_MULTICAST          /*如果支持组播*/
    if (inet_add_protocol(&igmp_protocol, IPPROTO_IGMP) < 0)      /*IGMP*/
        pr_crit("%s: Cannot add IGMP protocol\n", __func__);
#endif

    for (r = &inetsw[0]; r < &inetsw[SOCK_MAX]; ++r)
                                        /*初始化 inetsw[]数组项，双链表数组*/
        INIT_LIST_HEAD(r);
```

```
    for (q = inetsw_array; q < &inetsw_array[INETSW_ARRAY_LEN]; ++q) /*inet_protosw 实例数组*/
        inet_register_protosw(q);          /*注册 inet_protosw 实例，UDP、TCP 等协议对应的实例*/

    arp_init();     /*邻居子系统初始化，/net/ipv4/arp.c*/
    ip_init();      /*网络层协议初始化，路由选择表初始化等，/net/ipv4/ip_output.c*/
    tcp_v4_init();  /*为每个 CPU 核创建原始 TCP 套接字，设置 TCP 参数等，/net/ipv4/tcp_ipv4.c*/
    tcp_init();
          /*TCP 初始化，初始化 tcp_port 实例中（proto 结构体）inet_hashinfo 散列表等，/net/ipv4/tcp.c*/
    udp_init();     /*UDP 初始化，/net/ipv4/udp.c*/
    udplite4_register();     /*UDP-Lite 初始化，/net/ipv4/udplite.c*/
    ping_init();    /*ping 套接字初始化（初始化散列表），/net/ipv4/ping.c*/

    if (icmp_init() < 0)     /*ICMP（控制信息协议）初始化，/net/ipv4/icmp.c*/
        panic("Failed to create the ICMP control socket.\n");

#if defined(CONFIG_IP_MROUTE)       /*机器配置成组播路由器*/
    if (ip_mr_init())     /*初始化网络层组播路由，/net/ipv4/ipmr.c*/
        pr_crit("%s: Cannot init ipv4 mroute\n", __func__);
#endif

    if (init_inet_pernet_ops())        /*网络资源初始化，/net/ipv4/af_inet.c*/
          /*注册 pernet_operations 实例 af_inet_ops，实例初始化函数中设置 net->ipv4 成员*/
        pr_crit("%s: Cannot init ipv4 inet pernet ops\n", __func__);

    if (init_ipv4_mibs())         /*/net/ipv4/af_inet.c*/
          /*注册 pernet_operations 实例 ipv4_mib_ops，实例初始化函数中设置 net->mib 成员*/
        pr_crit("%s: Cannot init ipv4 mibs\n", __func__);

    ipv4_proc_init(); /*各传输层协议在 procfs 中的初始化，用于导出/设置参数等，/net/ipv4/af_inet.c*/
    ipfrag_init();          /*网络层数据包分段操作初始化，/net/ipv4/ip_fragment.c*/
    dev_add_pack(&ip_packet_type);       /*注册 IPv4 网络层协议 packet_type 实例*/
    rc = 0;
out:
    return rc;
    ...
}
fs_initcall(inet_init);       /*内核初始化子系统时调用此函数*/
```

初始化函数中注册了部分传输层协议对应的 proto 实例和 net_protocol 实例，注册了 inetsw_array[]数组中定义的 inet_protosw 实例（用于创建套接字），还完成了其它传输层协议的初始化，最后注册了网络层 IPv4 协议 packet_type 实例 ip_packet_type。

■初始 inet_protosw 实例

内核在/net/ipv4/af_inet.c 文件内为 UDP、TCP 等传输层协议定义了 inet_protosw 实例数组，也就是说

内核默认支持这几种传输层协议，在上面介绍的初始化函数中注册了这些实例。

```
static struct inet_protosw inetsw_array[] =
{
    {
        .type =       SOCK_STREAM,    /*流套接字，套接字类型*/
        .protocol =   IPPROTO_TCP,    /*TCP，传输层协议*/
        .prot =       &tcp_prot,      /*proto 实例，/net/ipv4/tcp_ipv4.c*/
        .ops =        &inet_stream_ops,  /*proto_ops 实例，/net/ipv4/af_inet.c*/
        .flags =      INET_PROTOSW_PERMANENT |INET_PROTOSW_ICSK,
    },

    {
        .type =       SOCK_DGRAM,     /*数据报套接字*/
        .protocol =   IPPROTO_UDP,    /*UDP*/
        .prot =       &udp_prot,
        .ops =        &inet_dgram_ops,
        .flags =      INET_PROTOSW_PERMANENT,
    },

    {
        .type =       SOCK_DGRAM,   /*数据报套接字*/
        .protocol =   IPPROTO_ICMP,  /*ICMP，ping 套接字*/
        .prot =       &ping_prot,
        .ops =        &inet_dgram_ops,
        .flags =      INET_PROTOSW_REUSE,
    },

    {
        .type =       SOCK_RAW,     /*原始套接字，无传输层协议，直接与网络层交互*/
        .protocol =   IPPROTO_IP,   /*匹配所有的协议类型*/
        .prot =       &raw_prot,
        .ops =        &inet_sockraw_ops,
        .flags =      INET_PROTOSW_REUSE,
    }
};
```

初始化函数 inet_init()中调用 inet_register_protosw()函数注册了 inetsw_array[]数组中实例。

■初始化网络资源

初始化函数 inet_init()中调用 init_inet_pernet_ops()函数注册了 pernet_operations 实例 af_inet_ops，实例初始化函数为 **inet_init_net()**，用于初始化网络命名空间中的 IPv4 网络资源，即 net->ipv4 成员。

inet_init_net()函数定义如下（/net/ipv4/af_inet.c）：

```
static __net_init int inet_init_net(struct net *net)
{
    /*设置默认的本地端口号范围*/
    seqlock_init(&net->ipv4.ip_local_ports.lock);
```

```
    net->ipv4.ip_local_ports.range[0] =    32768;
    net->ipv4.ip_local_ports.range[1] =    60999;

    seqlock_init(&net->ipv4.ping_group_range.lock);
    net->ipv4.ping_group_range.range[0] = make_kgid(&init_user_ns, 1);
    net->ipv4.ping_group_range.range[1] = make_kgid(&init_user_ns, 0);
    return 0;
}
```

2 系统控制参数

IPv4 协议簇实现中定义了许多控制参数，用于控制各层协议的行为（需选择 SYSCTL 配置选项）。控制参数定义在各层协议的实现代码中，以 sysctl_字符开头。例如：设置 IP 中默认的 ttl 值的控制参数：

```
int sysctl_ip_default_ttl __read_mostly = IPDEFTTL;     /*/net/ipv4/ip_output.c*/
EXPORT_SYMBOL(sysctl_ip_default_ttl);
```

有的控制参数导出到用户空间，可由用户设置/修改，此类参数需要定义并注册 ctl_table 实例。

内核在/net/ipv4/sysctl_net_ipv4.c 文件内定义了 ctl_table 实例数组：

```
static struct ctl_table ipv4_table[] = {       /*只适用于初始网络命名空间的控制参数*/
    {
        .procname       = "tcp_timestamps",
        .data       = &sysctl_tcp_timestamps,
        .maxlen         = sizeof(int),
        .mode           = 0644,
        .proc_handler = proc_dointvec     /*参数处理函数*/
    },
    ...
}

static struct ctl_table ipv4_net_table[] = { /*各网络命名空间（含初始网络命名空间）专有的控制参数*/
    {
        .procname       = "icmp_echo_ignore_all",
        .data       = &init_net.ipv4.sysctl_icmp_echo_ignore_all,
        .maxlen         = sizeof(int),
        .mode           = 0644,
        .proc_handler = proc_dointvec
    },
    ...
}
```

初始化函数 sysctl_ipv4_init()用于注册控制参数，函数定义如下：

```
static __init int sysctl_ipv4_init(void)
{
    struct ctl_table_header *hdr;

    hdr = register_net_sysctl(&init_net, "net/ipv4", ipv4_table);     /*注册 ipv4_table 列表中参数*/
```

```
    ...

    if (register_pernet_subsys(&ipv4_sysctl_ops)) {
        ...
    }
    return 0;
}
__initcall(sysctl_ipv4_init);    /*内核初始化阶段调用*/
```

sysctl_ipv4_init()函数在初始化网络命名空间中注册 pv4_table[]列表中控制参数，注册 pernet_operations 实例 **ipv4_sysctl_ops**，其初始化函数为在网络命名空间中注册 ipv4_net_table[]列表中控制参数。

另外，各层协议的实现中还可以定义自身的控制参数，并调用 register_net_sysctl()函数注册参数列表。例如：inet_init()函数中调用的 ip_static_sysctl_init()函数注册了路由选择相关的控制参数。

12.5.5 创建套接字

用户进程通过 socket()系统调用创建套接字，例如，下面是创建 ICMPv4 套接字的系统调用：

```
socket(PF_INET,SOCK_DGRAM,IPPROTO_ICMP);
```

socket()系统调用在前面介绍过，系统调用内创建 socket 实例后，将调用 net_proto_family 结构体实例 inet_family_ops 中的 create()函数为 socket 实例创建并设置 sock 实例，设置 socket.ops 成员（proto_ops 实例）。

create()函数根据参数传递的套接字类型和协议类型在 inetsw[]双链表数组中查找对应的 inet_protosw 实例，将实例关联的 proto_ops 实例赋予 socket 实例，依据 inet_protosw 实例关联的 proto 实例创建 xxx_sock 实例，调用 proto 实例中的 init()函数初始化 xxx_sock 实例。

PF_INET 协议簇注册的 net_proto_family 实例定义如下（/net/ipv4/af_inet.c）：

```
static const struct net_proto_family inet_family_ops = {
    .family = PF_INET,
    .create = inet_create,    /*设置 socket 实例函数*/
    .owner    = THIS_MODULE,
};
```

在创建套接字时将调用其中的 inet_create()函数，设置 socket 实例，创建并初始化 xxx_sock 实例等，函数定义在/net/ipv4/af_inet.c 文件内，代码如下：

```
static int inet_create(struct net *net, struct socket *sock, int protocol,int kern)
/*protocol：传输层协议，kern：是否是内核套接字*/
{
    struct sock *sk;
    struct inet_protosw *answer;
    struct inet_sock *inet;    /*指向 inet_sock 结构体，内嵌 sock 结构体成员，/include/net/inet_sock.h*/
    struct proto *answer_prot;        /*指向 proto 实例*/
    unsigned char answer_flags;
    int try_loading_module = 0;
    int err;

    sock->state = SS_UNCONNECTED;    /*未连接状态*/
```

```
    /*查找协议类型对应的 inet_protosw 实例*/
lookup_protocol:
    err = -ESOCKTNOSUPPORT;
    rcu_read_lock();
    list_for_each_entry_rcu(answer, &inetsw[sock->type], list) {
                                            /*遍历套接字类型 inet_protosw 实例双链表*/
        err = 0;
        if (protocol == answer->protocol) {       /*answer 指向协议类型对应的 inet_protosw 实例*/
            if (protocol != IPPROTO_IP)
                break;
        } else {
            if (IPPROTO_IP == protocol) {   /*如果 protocol 为 IPPROTO_IP，则返回第一个实例*/
                protocol = answer->protocol;
                break;
            }
            if (IPPROTO_IP == answer->protocol)
                break;
        }
        err = -EPROTONOSUPPORT;
    }

    if (unlikely(err)) {     /*未找到 inet_protosw 实例，加载模块后再查找*/
        if (try_loading_module < 2) {
            rcu_read_unlock();
            if (++try_loading_module == 1)
                request_module("net-pf-%d-proto-%d-type-%d",PF_INET, protocol, sock->type);
            else
                request_module("net-pf-%d-proto-%d",PF_INET, protocol);
            goto lookup_protocol;
        } else
            goto out_rcu_unlock;
    }

    err = -EPERM;
    if (sock->type == SOCK_RAW && !kern &&!ns_capable(net->user_ns, CAP_NET_RAW))
        goto out_rcu_unlock;

    sock->ops = answer->ops;          /*inet_protosw 实例关联 proto_ops 实例赋予 socket 实例*/
    answer_prot = answer->prot;      /*inet_protosw 实例关联 proto 实例*/
    answer_flags = answer->flags;     /*标记*/
    rcu_read_unlock();

    ...
    sk = sk_alloc(net, PF_INET, GFP_KERNEL, answer_prot, kern);       /*分配 xxx_sock 实例*/
    ...
    err = 0;
```

```
if (INET_PROTOSW_REUSE & answer_flags)
      sk->sk_reuse = SK_CAN_REUSE;

inet = inet_sk(sk);       /*指向 xxx_sock 实例中 inet_sock 结构体成员*/
inet->is_icsk = (INET_PROTOSW_ICSK & answer_flags) != 0;     /*是否是已连接套接字*/

inet->nodefrag = 0;

if (SOCK_RAW == sock->type) {        /*原始套接字*/
      inet->inet_num = protocol;          /*对原始套接字赋予端口号*/
      if (IPPROTO_RAW == protocol)
            inet->hdrincl = 1;
}

if (net->ipv4.sysctl_ip_no_pmtu_disc)      /*设置路径 MTU 探测标记*/
      inet->pmtudisc = IP_PMTUDISC_DONT;
else
      inet->pmtudisc = IP_PMTUDISC_WANT;

inet->inet_id = 0;     /*分段 id 值，初始化为 0*/

sock_init_data(sock, sk);        /*初始化 sock 实例，状态设为 TCP_CLOSE，/net/core/sock.c*/

sk->sk_destruct         = inet_sock_destruct;     /*析构函数*/
sk->sk_protocol         = protocol;               /*传输层协议类型*/
sk->sk_backlog_rcv = sk->sk_prot->backlog_rcv;

inet->uc_ttl     = -1;
inet->mc_loop= 1;
inet->mc_ttl    = 1;
inet->mc_all    = 1;
inet->mc_index      = 0;
inet->mc_list   = NULL;
inet->rcv_tos   = 0;       /*服务类型*/

sk_refcnt_debug_inc(sk);

if (inet->inet_num) {       /*只有原始套接字才调用以下函数，其它套接字此时端口号为 0*/
      inet->inet_sport = htons(inet->inet_num);
      sk->sk_prot->hash(sk);         /*将 sock 实例插入 proto 实例中管理链表中*/
}

if (sk->sk_prot->init) {
      err = sk->sk_prot->init(sk);       /*调用 proto 实例中初始化函数，初始化 sock 实例*/
      ...
}
```

```
out:
    return err;
    ...
}
```

inet_create()函数根据套接字类型和传输层协议查找 inet_protosw 实例，将其关联的 proto_ops 实例赋予 socket 实例，依据 inet_protosw 实例关联的 proto 实例创建表示套接字的 xxx_sock 实例，初始化 xxx_sock 实例内嵌的 inet_sock 结构体成员，调用 proto->init()函数初始化 sock 实例（内嵌在 inet_sock 结构体）。

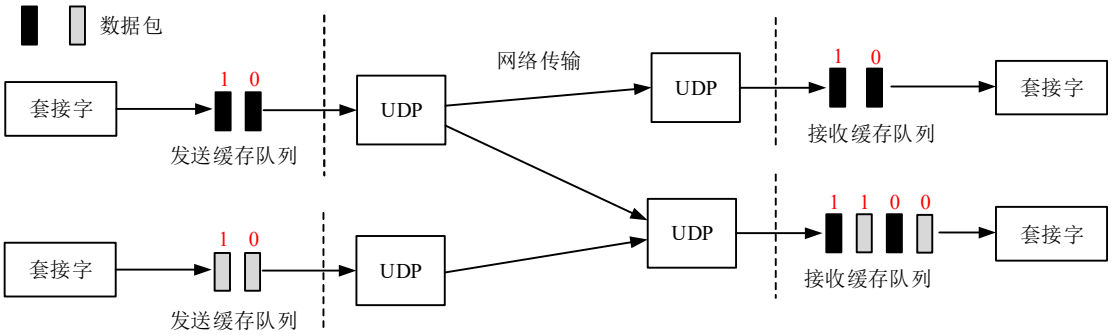
套接字操作（系统调用）将调用套接字关联的 proto_ops 实例中的函数，对于 IPv4 网络套接字又将进而调用传输层协议关联的 proto 实例中的函数完成操作。proto 实例是传输层协议相关的，在后面介绍具体传输层协议时将介绍套接字操作函数的实现。

12.6 UDP 实现

本节介绍 UDP 在 Linux 内核中的实现，IPv4 协议簇 UDP 实现代码主要在/net/ipv4/udp.c 文件内。

12.6.1 概述

UDP（用户数据报协议）是一种无连接的、不可靠的传输层协议，仅提供数据交付和差错检查功能。如下图所示，一个接收方可接收多个发送方发送的数据，发送方也可以向多个接收方发送数据。



每个通信方被赋予一个端口号，发送数据包时，通过目的 IP 地址寻址到主机，通过端口号寻址到接收方，并将发送的数据添加到接收方缓存队列。UDP 不保证数据的完整、正确和按序到达，如果要实现可靠性，需要用户进程在用户空间实现。

在 Linux 内核中，通信方由套接字表示。下面介绍 UDP 套接字的创建、管理和操作接口函数。

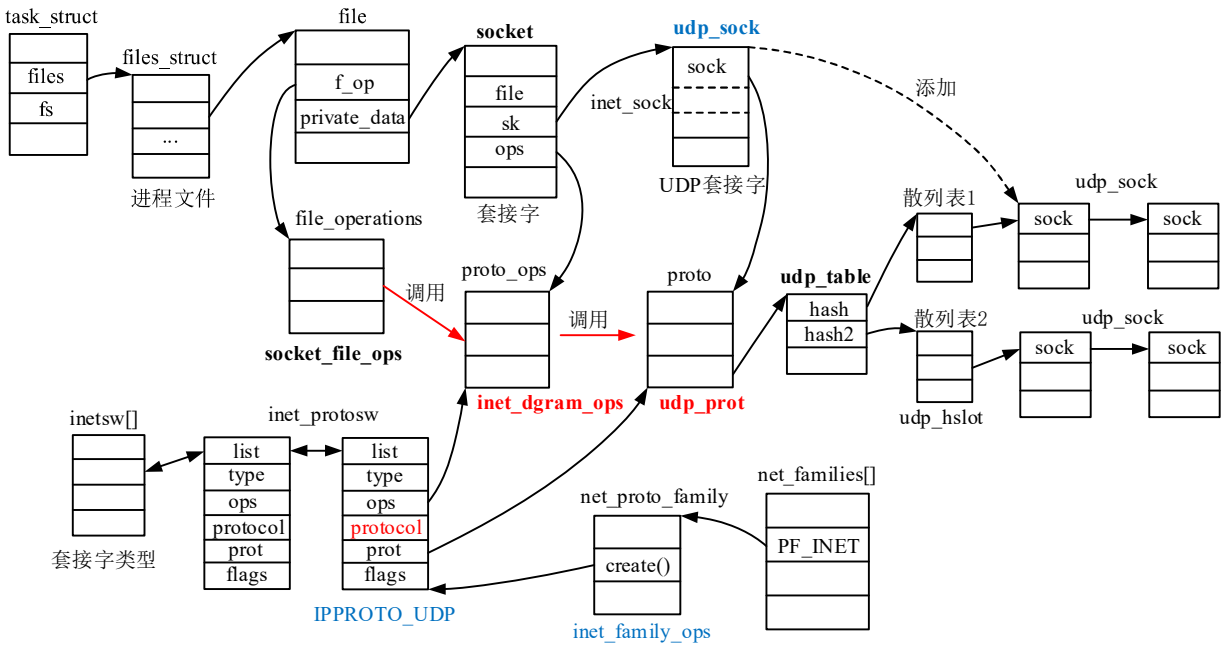
1 UDP 套接字

用户进程通过 socket()系统调用创建 UDP 套接字，如下所示：

socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);

AF_INET 表示 IPv4 地址簇，SOCK_DGRAM 表示数据报类型套接字，IPPROTO_UDP 表示传输层协议为 UDP。

socket()系统调用创建的 UDP 套接字如下图所示：



内核定义并注册了 UDP 对应的 inet_protosw 实例，在创建 UDP 套接字将实例关联的 proto_ops 实例 inet_dgram_ops 赋予 socket 实例。依据 inet_protosw 实例关联的 proto 实例 udp_prot 创建表示 UDP 套接字的 udp_sock 实例。udp_sock 实例将由 udp_prot 实例中的散列表管理（2 个散列表）。

内核在 inetsw_array[]数组中定义了 UDP 套接字对应的 inet_protosw 结构体实例（/net/ipv4/af_inet.c）：

```
static struct inet_protosw inetsw_array[] =
{
    ...
    {
        .type =       SOCK_DGRAM,     /*数据报类型套接字*/
        .protocol =   IPPROTO_UDP,    /*UDP*/
        .prot =       &udp_prot,      /*UDP 套接字关联的 proto 实例*/
        .ops =        &inet_dgram_ops,  /*UDP 套接字关联的 proto_ops 实例*/
        .flags =      INET_PROTOSW_PERMANENT,
     },
     ...
}
```

UDP 套接字关联的 proto 实例为 **udp_prot**，套接字操作结构 proto_ops 实例为 **inet_dgram_ops**。

UDP 套接字 proto 实例 udp_prot 定义如下（/net/ipv4/udp.c）：

```
struct proto udp_prot = {       /*没有定义 init()、bind()函数等*/
    .name               = "UDP",
    ...
    .obj_size       = sizeof(struct udp_sock),    /*表示 UDP 套接字的 udp_sock 结构体大小*/
    .slab_flags         = SLAB_DESTROY_BY_RCU,
    .h.udp_table        = &udp_table,    /*管理 udp_sock 实例的 udp_table 结构体实例，见下文*/
    ...
};
```

■udp_sock

由 udp_prot 实例可知，UDP 套接字由 **udp_sock** 结构体表示，定义如下（/include/linux/udp.h）：

```
struct udp_sock {
    struct inet_sock inet;        /*inet_sock 结构体成员，必须是第一个成员*/
  #define udp_port_hash          inet.sk.__sk_common.skc_u16hashes[0]    /*第 1 个散列表散列值*/
  #define udp_portaddr_hash      inet.sk.__sk_common.skc_u16hashes[1]    /*第 2 个散列表散列值*/
  #define udp_portaddr_node      inet.sk.__sk_common.skc_portaddr_node   /*第 2 个散列表节点*/
    int            pending;            /*是否有挂起的数据包*/
    unsigned int   corkflag;     /*可通过 setsockopt()系统调用启动/关闭抑制功能*/
                  /*抑制功能，启用后发送数据累积为一个数据包，直到标记取消后才进行传输*/
    __u8           encap_type;       /* Is this an Encapsulation socket? */
    unsigned char  no_check6_tx:1,
                   no_check6_rx:1;

    __u16          len;      /*发送缓存队列长度（字节数）*/

    /* UDP-Lite 私有成员*/
    __u16          pcslen;
    __u16          pcrlen;

    /*pcflag 成员值*/
  #define  UDPLITE_BIT         0x1          /* set by udplite proto init function */
  #define  UDPLITE_SEND_CC     0x2          /* set via udplite setsockopt          */
  #define  UDPLITE_RECV_CC     0x4          /* set via udplite setsocktopt         */
    __u8           pcflag;                 /* marks socket as UDP-Lite if > 0     */
    __u8           unused[3];

    int (*encap_rcv)(struct sock *sk, struct sk_buff *skb);
    void (*encap_destroy)(struct sock *sk);
};
```

udp_sock 结构体主要成员简介如下：

●**inet**：inet_sock 结构体成员，这是 IPv4 套接字中一个通用的数据结构（前一节介绍过），必须是第一个成员。inet_sock 结构体中第一个成员为 sock 结构体成员。

●**pending**：标识发送缓存队列中是否有等待发送的数据包。

●**corkflag**：标记是否启用了抑制功能，见下文发送数据操作中的介绍。

●**udp_port_hash**：在 UDP 关联的 proto 实例 udp_prot 中创建了两个散列表，用于管理 udp_sock 实例。此成员表示 udp_sock 实例在第 1 个散列表中的散列值，由网络命名空间和端口号计算得到。

●**udp_portaddr_hash**：表示 udp_sock 实例在第 2 个散列表中的散列值，由网络命名空间、端口号和 inet_sk(sk)->**inet_rcv_saddr** 成员值（源 IP 地址）计算得到。

2 udp_sock 实例管理

UDP 关联的 proto 实例 udp_prot 中建立了 2 个散列表，用于管理 udp_prot 实例。udp_sock 实例执行绑定操作时，将会插入到散列表中。当本机 IP 地址改变时，udp_sock 实例需要重新插入散列表。

■**散列表**

在 UDP 套接字 proto 实例 udp_prot 中 h.udp_table 成员指向 **udp_table** 实例（udp_table 结构体），用于管理 udp_sock 实例，udp_table 结构体定义如下（/include/net/udp.h）：

```
struct udp_table {
    struct udp_hslot    *hash;    /*指向散列表 1，以本地端口号计算散列值*/
    struct udp_hslot    *hash2;   /*指向散列表 2，以本地端口号和本地 IP 地址计算散列值*/
    unsigned int        mask;     /*散列表中 udp_hslot 结构体数组项索引值最大值*/
    unsigned int        log;      /*分配端口号时，表示端口号对应位图中的位置*/
};
```

hash 和 hash2 成员指向的散列表由 udp_hslot 结构体数组构成，(mask+1)表示数组项数。
udp_hslot 结构体定义在/include/linux/list_nulls.h 头文件，表示散列链表头：

```
struct hlist_nulls_head {
    struct hlist_nulls_node *first;    /*指向散列链表中第一个成员*/
};
```

散列链表中成员由 hlist_nulls_node 结构体表示，定义如下：

```
struct hlist_nulls_node {
    struct hlist_nulls_node *next, **pprev;
};
```

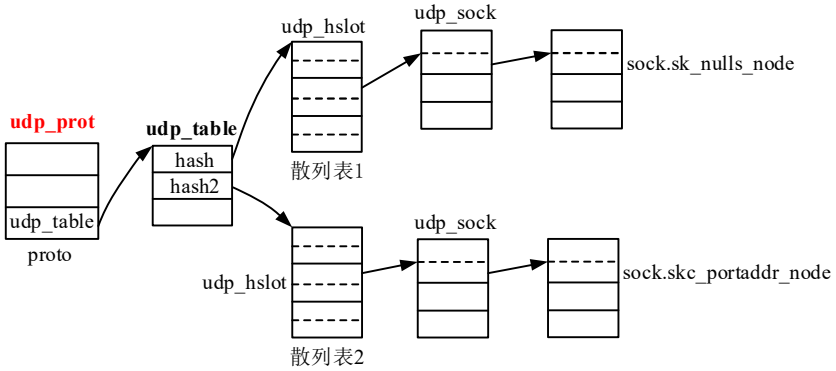
内核在/net/ipv4/udp.c 文件内定义了 udp_table 结构体实例，并赋予了 udp_prot 实例：

```
struct udp_table    udp_table __read_mostly;
```

内核在 UDP 协议初始化函数 udp_init()中将创建并初始化散列表，函数定义如下（/net/ipv4/udp.c）：

```
void __init udp_init(void)
{
    unsigned long limit;

    udp_table_init(&udp_table, "UDP");    /*创建并初始化散列表，/net/ipv4/udp.c*/
    limit = nr_free_buffer_pages() / 8;
    limit = max(limit, 128UL);
    sysctl_udp_mem[0] = limit / 4 * 3;
    sysctl_udp_mem[1] = limit;
    sysctl_udp_mem[2] = sysctl_udp_mem[0] * 2;

    sysctl_udp_rmem_min = SK_MEM_QUANTUM;
    sysctl_udp_wmem_min = SK_MEM_QUANTUM;
}
```

udp_table_init()函数用于创建并初始化 2 个散列表，函数源代码请读者自行阅读。
udp_prot 实例中散列表结构如下图所示：



udp_sock 通过内嵌 sock 结构体成员的 sk_null_node 成员链入到第 1 个散列表，通过 skc_portaddr_node 成员链入到第 2 个散列表。

udp_sock 实例在第 1 个散列表中的散列值，由网络命名空间和端口号计算得到；在第 2 个散列表中的散列值，由网络命名空间、inet_sk(sk)->**inet_rcv_saddr** 和端口号计算得到。散列值分别保存在 udp_sock 实例 udp_port_hash 和 udp_portaddr_hash 成员中。

■散列表操作

散列表操作主要包括将 udp_sock 实例插入到散列表，以及在散列表中查找指定实例。

●插入散列表

UDP 套接字关联的 proto 实例 udp_prot 中，与散列表操作相关的函数指针成员如下所示：

```
struct proto udp_prot = {        /*没有定义 init()、bind()函数等*/
    .name               = "UDP",
    ...
    .hash               = udp_lib_hash,      /*空操作*/
    .unhash             = udp_lib_unhash,    /*将 udp_sock 实例从散列表中移除*/
    .rehash             = udp_v4_rehash,
                /*inet_sk(sk)->inet_rcv_saddr 改变时，将 udp_sock 实例重新插入散列表 2*/
    .get_port       = udp_v4_get_port,     /*在绑定套接字时，为套接字赋予端口号，并插入散列表*/
    ...
}
```

套接字在绑定地址时，含手工绑定（bind()系统调用）和自动绑定，将调用 udp_prot 实例中的 get_port()函数，即 udp_v4_get_port()函数，为套接字设置/分配端口号，计算 2 个散列值，将 udp_sock 实例插入 2 个散列表中。如果套接字尚未指定源 IP 地址，计算散列值值时源 IP 地址设为 0。

在 connect()系统调用中，如果 inet_sock->inet_rcv_saddr 成员值为 0，将设置此值，并调用 udp_prot 实例中的 rehash()函数，即 udp_v4_rehash()函数，将 udp_sock 实例重新插入第 2 个散列表。

下面介绍 udp_v4_get_port()函数的实现，此函数为套接字设置或分配端口号，并将 udp_sock 实例插入到管理散列表中。

udp_v4_get_port()函数定义如下（/net/ipv4/udp.c）：

```
int udp_v4_get_port(struct sock *sk, unsigned short snum)
/*sk：指向 udp_sock 内嵌的 sock 结构体成员，snum：绑定端口号，0 表示由内核分配端口号*/
{
    unsigned int hash2_nulladdr =
        udp4_portaddr_hash(sock_net(sk), htonl(INADDR_ANY), snum);     /*第 1 个散列值*/
```

```
                                        /*由网络命名空间、全零 IP 地址、端口号计算散列值*/
    unsigned int hash2_partial =
        udp4_portaddr_hash(sock_net(sk), inet_sk(sk)->inet_rcv_saddr, 0);     /*第 2 个散列值*/
                                /*由网络命名空间、inet_rcv_saddr 地址和 0 端口号计算散列值*/
    udp_sk(sk)->udp_portaddr_hash = hash2_partial;       /*保存第 2 个散列表散列值*/
    return udp_lib_get_port(sk, snum, ipv4_rcv_saddr_equal, hash2_nulladdr);   /*插入散列表*/
}
```

udp4_portaddr_hash()函数中首先计算 2 个散列值，hash2_nulladdr 由网络命名空间、全零 IP 地址值和端口号 snum 计算得来，hash2_partial 由网络命名空间、源 IP 地址值和 0 端口号计算得来，并赋予 udp_sock 实例中 udp_portaddr_hash 成员。这 2 个散列值都是用于第 2 个散列表的。

udp4_portaddr_hash()函数然后调用 **udp_lib_get_port**()函数为套接字设置/分配端口号，并将套接字实例插入 2 个散列表中，函数定义如下（/net/ipv4/udp.c）：

```
int udp_lib_get_port(struct sock *sk, unsigned short snum,int (*saddr_comp)(const struct sock *sk1,
                                const struct sock *sk2),unsigned int hash2_nulladdr)
/*
*sk：sock 实例指针；snum：端口号，saddr_comp()：xxx_sock 实例比较函数（两者是否相同），
*此处为 ipv4_rcv_saddr_equal()（比较套接字源 IP 地址）；
*hash2_nulladdr：在 snum 参数非零时才会使用（计算散列值时 IP 地址为 0）。
*/
{
    struct udp_hslot *hslot, *hslot2;     /*指向散列表 1 和散列表 2 中的 udp_hslot 实例*/
    struct udp_table *udptable = sk->sk_prot->h.udp_table;     /*udp_prot 实例中 udp_table 实例*/
    int       error = 1;
    struct net *net = sock_net(sk);       /*网络命名空间*/

    if (!snum) {       /*snum 参数传递的端口号为 0*/
        int low, high, remaining;
        unsigned int rand;
        unsigned short first, last;
        /*下面是获取可用的端口号*/
        DECLARE_BITMAP(bitmap, PORTS_PER_CHAIN);
        inet_get_local_port_range(net, &low, &high);
        remaining = (high - low) + 1;

        rand = prandom_u32();
        first = reciprocal_scale(rand, remaining) + low;
        rand = (rand | 1) * (udptable->mask + 1);
        last = first + udptable->mask + 1;
        do {
            hslot = udp_hashslot(udptable, net, first);   /*由网络命名空间和端口号计算散列值*/
                                        /*指向第 1 个散列表中 udp_hslot 实例，first 表示端口号*/
            bitmap_zero(bitmap, PORTS_PER_CHAIN);
            spin_lock_bh(&hslot->lock);
            udp_lib_lport_inuse(net, snum, hslot, bitmap, sk,saddr_comp, udptable->log);
                                                /*在 bitmap 位图中标记 snum 端口号已使用*/
```

```
                snum = first;
                do {
                    if (low <= snum && snum <= high &&!test_bit(snum >> udptable->log, bitmap)
                                &&!inet_is_local_reserved_port(net, snum))
                        goto found;       /*查找到了可用的端口号，跳转到 found 处*/
                    snum += rand;
                } while (snum != first);
                spin_unlock_bh(&hslot->lock);
            } while (++first != last);
            goto fail;
        } else {        /*snum 参数传递的端口号不为 0*/
            hslot = udp_hashslot(udptable, net, snum);
                                        /*第 1 个散列表，由网络命名空间和端口号计算散列值*/
            spin_lock_bh(&hslot->lock);
            if (hslot->count > 10) {     /*链表中实例大于 10，查找散列表 2 中是否有相同的实例*/
                int exist;
                unsigned int slot2 = udp_sk(sk)->udp_portaddr_hash ^ snum; /*在第 2 个散列表中散列值*/
                slot2                &= udptable->mask;       /*计算散列表 2 中散列值*/
                hash2_nulladdr &= udptable->mask;        /*全零 IP 地址和 snum 端口号计算得散列值*/

                hslot2 = udp_hashslot2(udptable, slot2);     /*第 2 个散列表，slot2 为散列值*/
                if (hslot->count < hslot2->count)
                    goto scan_primary_hash;

                exist = udp_lib_lport_inuse2(net, snum, hslot2,sk, saddr_comp);
                                                /*检查散列表 2 中是否有相同的 udp_sock 实例*/
                if (!exist && (hash2_nulladdr != slot2)) {     /*snum 端口号未被使用*/
                    hslot2 = udp_hashslot2(udptable, hash2_nulladdr);
                    exist = udp_lib_lport_inuse2(net, snum, hslot2,sk, saddr_comp);
                                                /*检查 snum 端口号套接字是否已经存在*/
                }
                if (exist)
                    goto fail_unlock;     /*udp_sock 已经存在，返回 1，表示错误*/
                else
                    goto found;        /*没有相同的 udp_sock 实例，跳转至 found 处*/
            }
scan_primary_hash:        /*链表中实例不大于 10，只在第 1 个散列表中查找是否有相同的实例*/
            if (udp_lib_lport_inuse(net, snum, hslot, NULL, sk,saddr_comp, 0))
                goto fail_unlock;
        }

found:     /*snum 表示的端口号（不为 0）可用，往下执行，端口号可以是指定的或分配的*/
        inet_sk(sk)->inet_num = snum;         /*端口号赋予 inet_sock 实例*/
        udp_sk(sk)->udp_port_hash = snum;     /*端口号设为第 1 个散列表散列值*/
        udp_sk(sk)->udp_portaddr_hash ^= snum;    /*第 2 个散列表中散列值*/
                                    /*udp_portaddr_hash 为由源 IP 值和 0 端口号计算的散列值*/
```

```
    if (sk_unhashed(sk)) {              /*sock.skc_node 为 NULL*/
        sk_nulls_add_node_rcu(sk, &hslot->head);     /*将 udp_sock 实例添加到第 1 个散列表*/
        hslot->count++;          /*实例数量加 1*/
        sock_prot_inuse_add(sock_net(sk), sk->sk_prot, 1);

        hslot2 = udp_hashslot2(udptable, udp_sk(sk)->udp_portaddr_hash);
                                                    /*在第 2 个散列表中对应的散列链表*/
        spin_lock(&hslot2->lock);
        hlist_nulls_add_head_rcu(&udp_sk(sk)->udp_portaddr_node,&hslot2->head);
                                                     /*将 udp_sock 实例添加到第 2 个散列表*/
        hslot2->count++;     /*链表中实例数量加 1*/
        spin_unlock(&hslot2->lock);
    }
    error = 0;
fail_unlock:
    spin_unlock_bh(&hslot->lock);
fail:
    return error;     /*返回 1 表示出错，返回 0 表示成功*/
}
```

对 udp_sock 实例调用 **udp_lib_get_port**()函数前，udp_sk(sk)->udp_portaddr_hash 保存由源主机 IP 地址和 0 端口号计算的散列值，参数 hash2_nulladdr 是由全零 IP 地址和 snum 端口号（可能为 0）计算的散列值。

udp_lib_get_port()函数执行流程如下：

（1）判断参数 snum 是否为 0，不为 0 则执行步骤 2。snum 为 0，则自动分配端口号，赋予 snum，并计算在散列表 1 中的散列值，跳至步骤 3。

（2）由给定 snum 值（非零）计算在散列表 1 中的散列值。如果散列表 1 中散列值对应链表中的实例数量大于 10，则执行下面的步骤 a，否则执行步骤 b：

a：依据 udp_sk(sk)->udp_portaddr_hash 和 snum 计算在散列表 2 中的散列值 slot2。如果散列表 1 对应链表中的实例数量大于散列表 2 中对应链表中实例的数量，则跳转至步骤 b，否则执行以下操作。

检果 udp_sock 实例在散列表 2 对应链表中是否存在，存在则函数返回错误。不存在，则判断散列值 slot2 与 hash2_nulladdr 是否相等，相等则跳至步骤 3。不相等则以 hash2_nulladdr 作为散列表 2 中的散列值，检查是否存在相同的 udp_sock 实例，存在则返回错误，不存在则跳转至步骤 3。

b：检查散列表 1 散列值对应链表中是否存在相同的 udp_sock 实例，如果存在则函数返回错误，不存在则执行步骤（3）。

（3）将 udp_sock 实例插入到第 1 个散列表，散列值由网络命名空间和端口号计算得来；由端口号 snum 和 udp_sk(sk)->udp_portaddr_hash 值计算在散列表 2 中的散列值，将 udp_sock 实例插入到第 2 个散列表。

在手工绑定或自动绑定中，可能没有设置 udp_sock 实例的源 IP 地址，在 connect()系统调用中将设置或修改 udp_sock 实例源 IP 地址（inet_rcv_saddr 值），此时需要调用 udp_prot 实例中的 rehash()函数，即 **udp_v4_rehash()**函数，将 udp_sock 实例重新插入第 2 个散列表，此函数定义在/net/ipv4/udp.c 文件内，源代码请读者自行阅读。

另外，udp_lib_unhash(struct sock *sk)函数用于将 udp_sock 实例从两个散列表中移除。

●查找实例

UDP 在接收到网络层上传的数据包时，需要根据报头信息，在 udp_prot 实例散列表中查找目的套接字

udp_sock 实例，并将数据包添加到其接收缓存队列，以等待用户进程的读取。

udp4_lib_lookup()函数可视为查找操作的入口函数，定义如下（/net/ipv4/udp.c）：

```
struct sock *udp4_lib_lookup(struct net *net, __be32 saddr, __be16 sport,
                                                    __be32 daddr, __be16 dport, int dif)
/*
*net：网络命名空间，saddr：数据报报头中指示的源 IP 地址，sport：源端口号，
*daddr：目的 IP 地址，dport：目的端口号，dif：网络设备编号。
*/
{
    return __udp4_lib_lookup(net, saddr, sport, daddr, dport, dif, &udp_table);
}
```

__udp4_lib_lookup()函数定义如下（/net/ipv4/udp.c）：

```
struct sock *__udp4_lib_lookup(struct net *net, __be32 saddr,
        __be16 sport, __be32 daddr, __be16 dport,int dif, struct udp_table *udptable)
{
    struct sock *sk, *result;
    struct hlist_nulls_node *node;
    unsigned short hnum = ntohs(dport);     /*目的端口号*/
    unsigned int hash2, slot2, slot = udp_hashfn(net, hnum, udptable->mask); /*第 1 个散列表中散列值*/
    struct udp_hslot *hslot2, *hslot = &udptable->hash[slot];     /*第 1 个散列链表指针*/
    int score, badness, matches = 0, reuseport = 0;
    u32 hash = 0;

    rcu_read_lock();
    if (hslot->count > 10) {     /*第 1 个散列链表中成员数大于 10*/
        hash2 = udp4_portaddr_hash(net, daddr, hnum);     /*第 2 个散列表中散列值*/
        slot2 = hash2 & udptable->mask;
        hslot2 = &udptable->hash2[slot2];
        if (hslot->count < hslot2->count)
            goto begin;

        result = udp4_lib_lookup2(net, saddr, sport,daddr, hnum, dif,hslot2, slot2);
                                        /*在第 2 个散列表中查找 udp_sock 实例*/
        if (!result) {      /*如果未找到，将目的 IP 地址设为 0，计算散列值后再查找*/
            hash2 = udp4_portaddr_hash(net, htonl(INADDR_ANY), hnum);
            slot2 = hash2 & udptable->mask;
            hslot2 = &udptable->hash2[slot2];
            if (hslot->count < hslot2->count)
                goto begin;

            result = udp4_lib_lookup2(net, saddr, sport,htonl(INADDR_ANY), hnum, dif,hslot2, slot2);
        }
        rcu_read_unlock();
        return result;     /*返回找到的 sock 实例，如果未找到返回 NULL*/
    }
```

```
begin:      /*第 1 个散列链表中成员数小于 10，在第一个散列表中查找*/
    result = NULL;
    badness = 0;
    sk_nulls_for_each_rcu(sk, node, &hslot->head) {     /*遍历散列链表中 udp_sock 实例*/
        score = compute_score(sk, net, saddr, hnum, sport,daddr, dport, dif);
                                                /*检查 udp_sock 实例是否是需要查找的实例*/
        if (score > badness) {
            result = sk;
            badness = score;
            reuseport = sk->sk_reuseport;
            if (reuseport) {
                hash = udp_ehashfn(net, daddr, hnum,saddr, sport);
                matches = 1;
            }
        } else if (score == badness && reuseport) {
            matches++;
            if (reciprocal_scale(hash, matches) == 0)
                result = sk;
            hash = next_pseudo_random32(hash);
        }
    }
    if (get_nulls_value(node) != slot)
        goto begin;

    if (result) {
        if (unlikely(!atomic_inc_not_zero_hint(&result->sk_refcnt, 2)))
            result = NULL;
        else if (unlikely(compute_score(result, net, saddr, hnum, sport,daddr, dport, dif) < badness)) {
            sock_put(result);
            goto begin;
        }
    }
    rcu_read_unlock();
    return result;
}
```

__udp4_lib_lookup()函数中由网络命名空间、目的端口号计算在第 1 个散列表中的散列值。如果第 1 个散列链表中实例数量大于 10，则调用 **udp4_lib_lookup2()**函数在第 2 个散列表中查找匹配的 udp_sock 实例，否则在第 1 个散列表中查找匹配的 udp_sock 实例，如果找到返回其中的 sock 成员指针，没有找到返回 NULL。

需要注意的是，在 udp4_lib_lookup2()和__udp4_lib_lookup()函数内，将遍历散列链表中的 udp_sock 实例，并分别调用 **compute_score2**()和 **compute_score**()函数，检查 udp_sock 实例是否是需要查找的实例。在这两个函数中，将比对 udp_sock 实例所在网络命名空间、源 IP 地址、源端口号、目的 IP 地址、目的端口号是否与函数参数传递的数值相等，相等才表示匹配成功。

特别地，如果套接字执行了 connect()系统调用，即 udp_sock 实例设置了目的 IP 地址、目的端口号，它们必须与__udp4_lib_lookup()函数参数中的源 IP 地址、源端口号（发送方的地址、端口号）相等，才能匹配成功。也就是说如果套接字执行 connect()系统调用，它将只接收指定目的套接字发送的数据报，而不

接收其它套接字发送的数据报。

3 套接字操作

由前面的介绍可知 UDP 套接字操作结构 proto_ops 实例为 **inet_dgram_ops**，定义如下：

```
const struct proto_ops inet_dgram_ops = { /*/net/ipv4/af_inet.c，inet_xxx()函数适用于所有 IPv4 套接字*/
    .family          = PF_INET,
    .owner    = THIS_MODULE,
    .release        = inet_release,
    .bind        = inet_bind,     /*绑定操作*/
    .connect        = inet_dgram_connect,   /*连接操作*/
    .socketpair     = sock_no_socketpair,
    .accept         = sock_no_accept,
    .getname        = inet_getname,
    .poll           = udp_poll,
    .ioctl          = inet_ioctl,     /*IO 控制，/net/ipv4/af_inet.c*/
    .listen         = sock_no_listen,
    .shutdown       = inet_shutdown,
    .setsockopt     = sock_common_setsockopt,       /*设置参数*/
    .getsockopt     = sock_common_getsockopt,       /*获取参数*/
    .sendmsg        = inet_sendmsg,     /*发送数据*/
    .recvmsg        = inet_recvmsg,     /*接收数据*/
    .mmap       = sock_no_mmap,
    .sendpage       = inet_sendpage,
  #ifdef CONFIG_COMPAT
    ...
  #endif
};
```

UDP 套接字关联 proto 结构体实例 udp_prot 定义如下：

```
struct proto udp_prot = {        /*没有定义 init()、bind()函数等*/
    .name           = "UDP",
    .owner          = THIS_MODULE,
    .close          = udp_lib_close,
    .connect        = ip4_datagram_connect,     /*连接函数*/
    .disconnect     = udp_disconnect,
    .ioctl          = udp_ioctl,        /*处理特定于 UDP 的命令*/
    .destroy        = udp_destroy_sock,
    .setsockopt     = udp_setsockopt,     /*读取参数*/
    .getsockopt     = udp_getsockopt,     /*设置参数*/
    .sendmsg        = udp_sendmsg,     /*发送消息*/
    .recvmsg        = udp_recvmsg,     /*接收消息*/
    ...
    .hash           = udp_lib_hash,       /*空操作*/
    .unhash         = udp_lib_unhash,    /*从散列表中移除 udp_sock 实例*/
    .rehash         = udp_v4_rehash,    /*源 IP 地址修改后，重新插入散列表*/
```

```
    .get_port       = udp_v4_get_port,     /*在散列表中查找 udp_sock 实例*/
    ...
    .obj_size       = sizeof(struct udp_sock),     /*表示 UDP 套接字的 udp_sock 结构体*/
    .slab_flags             = SLAB_DESTROY_BY_RCU,
    .h.udp_table            = &udp_table,     /*管理 udp_sock 实例的散列表*/
    ...
};
```

套接字操作结构 inet_dgram_ops 实例中的函数将调用 udp_prot 实例中的相应函数完成操作。本节的后面小节将介绍 UDP 套接字操作函数的实现。

12.6.2 绑定操作

绑定操作是指给当前套接字赋予地址值，包括 IP 地址和端口号。通常服务器用户进程可以通过 bind()系统调用进行手动绑定，将套接字绑定到指定地址。UDP 套接字发送数据时，若未绑定端口号，将由内核自动为其绑定地址。

1 手工绑定

bind()系统调用将调用套接字关联 proto_ops 实例中的 bind()函数执行绑定操作。

UDP 套接字操作结构 inet_dgram_ops 实例中，绑定操作函数为 inet_bind()，这是一个 IPv4 套接字公用的函数，定义如下（/net/ipv4/af_inet.c）：

```
int inet_bind(struct socket *sock, struct sockaddr *uaddr, int addr_len)
/*uaddr：指向通用地址结构，addr_len：地址长度*/
{
    struct sockaddr_in *addr = (struct sockaddr_in *)uaddr;   /*转换成 IPv4 地址结构*/
    struct sock *sk = sock->sk;
    struct inet_sock *inet = inet_sk(sk);     /*指向 inet_sock 实例*/
    struct net *net = sock_net(sk);         /*所属网络命名空间*/
    unsigned short snum;     /*端口号*/
    int chk_addr_ret;
    int err;

    /*udp_prot 实例没有定义 bind()函数*/
    if (sk->sk_prot->bind) {
        err = sk->sk_prot->bind(sk, uaddr, addr_len);       /*调用 proto 实例中的绑定函数*/
        goto out;
    }
    ...  /*地址有效性检查*/

    chk_addr_ret = inet_addr_type(net, addr->sin_addr.s_addr);
                                        /*检查 IP 地址类型，/net/ipv4/fib_frontend.c*/

    err = -EADDRNOTAVAIL;
    if (!net->ipv4.sysctl_ip_nonlocal_bind &&!(inet->freebind || inet->transparent) &&
        addr->sin_addr.s_addr != htonl(INADDR_ANY) &&chk_addr_ret != RTN_LOCAL &&
        chk_addr_ret != RTN_MULTICAST &&chk_addr_ret != RTN_BROADCAST)
        goto out;        /*有效性检查*/
```

```
        snum = ntohs(addr->sin_port);     /*地址参数中指定的端口号*/
        err = -EACCES;
        if (snum && snum < PROT_SOCK &&!ns_capable(net->user_ns, CAP_NET_BIND_SERVICE))
            goto out;

        lock_sock(sk);

        /* Check these errors (active socket, double bind). */
        err = -EINVAL;
        if (sk->sk_state != TCP_CLOSE || inet->inet_num)       /*须为 TCP_CLOSE 状态，inet_num 为 0*/
            goto out_release_sock;                              /*初始化 sock 实例时，设为 TCP_CLOSE 状态*/

        inet->inet_rcv_saddr = inet->inet_saddr = addr->sin_addr.s_addr;   /*绑定 IP 地址*/
        if (chk_addr_ret == RTN_MULTICAST || chk_addr_ret == RTN_BROADCAST)
            inet->inet_saddr = 0;                /*广播和组播地址，设为 0*/

        /*udp_prot 实例 get_port()函数将 snum 端口号赋予 udp_sock 实例，如果为 0，则自动分配端口号，
        *然后将 udp_sock 实例插入到散列表 1 和散列表 2，成功返回 0，出错返回 1。
        */
        if ((snum || !inet->bind_address_no_port) &&sk->sk_prot->get_port(sk, snum)) {
            ...
        }

        if (inet->inet_rcv_saddr)     /*本机源 IP 地址*/
            sk->sk_userlocks |= SOCK_BINDADDR_LOCK;
        if (snum)
            sk->sk_userlocks |= SOCK_BINDPORT_LOCK;
        inet->inet_sport = htons(inet->inet_num);       /*本机字节序转网络字节序*/
                                    /*源端口号，在 get_port()函数中对 inet->inet_num 赋值*/
        inet->inet_daddr = 0;       /*目的 IP 地址、端口号赋予 0*/
        inet->inet_dport = 0;
        sk_dst_reset(sk);    /*sk->sk_dst_cache 设为 NULL，/include/net/sock.h*/
        err = 0;
    out_release_sock:
        release_sock(sk);
    out:
        return err;
    }
```

绑定操作的主要工作是将参数 uaddr 指向的 sockaddr_in 实例中的 IP 地址和端口号，赋予 inet_sock 实例，并将 udp_sock 实例插入到管理散列表。其中 **inet->inet_rcv_saddr、inet->inet_saddr** 成员保存绑定的 IP 地址，**inet->inet_sport、inet->inet_num** 成员保存端口号，目的 IP 地址和目的端口号设为 0。

2 自动绑定

在发送消息时，如果 UDP 套接字尚未绑定地址，内核将调用 **inet_autobind()**函数，自动为套接字绑定

地址，函数定义如下（/net/ipv4/af_inet.c）：

```
static int inet_autobind(struct sock *sk)
{
    struct inet_sock *inet;
    lock_sock(sk);
    inet = inet_sk(sk);       /*inet_sock 实例*/
    if (!inet->inet_num) {       /*创建套接字时 inet->inet_num 通常为 0*/
        if (sk->sk_prot->get_port(sk, 0)) {     /*调用 proto 实例中的 get_port()函数*/
            release_sock(sk);
            return -EAGAIN;
        }
        inet->inet_sport = htons(inet->inet_num);     /*端口号赋值，本机字节序转网络字节序*/
    }
    release_sock(sk);
    return 0;
}
```

inet_autobind()函数内主要是调用 UDP 套接字关联的 proto 实例中的 get_port()函数为套接字自动分配端口号，并将 udp_sock 实例插入散列表。UDP 套接字 proto 实例 udp_port 中 get_port()函数 **udp_v4_get_port()** 前面介绍过了，这里就不再重复了。

12.6.3 连接操作

UDP 虽然是无连接的协议，但是连接操作也是有意义的。UDP 连接操作的作用是将目的套接字地址、端口号写入本套接字。发送数据时，若不指定目的套接字，则默认发送到连接指定的套接字，指定目的套接字时也可以发给其它套接字。接收数据时，只接收连接指定套接字发送的数据。

1 连接函数

用户进程通过 connect()系统调用为套接字执行连接操作，系统调用内将调用套接字关联 proto_ops 实例中的 connect()函数执行连接操作。

UDP 套接字关联 proto_ops 实例 inet_dgram_ops 中连接函数 connect()为 **inet_dgram_connect()**，定义如下（/net/ipv4/af_inet.c）：

```
int inet_dgram_connect(struct socket *sock, struct sockaddr *uaddr,int addr_len, int flags)
/*uaddr：指向目的地址实例，addr_len：目的地址结构长度*/
{
    struct sock *sk = sock->sk;

    if (addr_len < sizeof(uaddr->sa_family))
        return -EINVAL;
    if (uaddr->sa_family == AF_UNSPEC)
        return sk->sk_prot->disconnect(sk, flags);        /*地址有效性判断*/

    if (!inet_sk(sk)->inet_num && inet_autobind(sk))
        return -EAGAIN;        /*如果本套接字没有绑定端口号，则执行自动绑定*/

    return sk->sk_prot->connect(sk, uaddr, addr_len);     /*调用 udp_prot 实例中的 connect()函数*/
```

```
}
```

inet_dgram_connect()函数中首先判断地址的有效性，然后判断当前套接字是否绑定了地址，如果没有则自动绑定地址，最后调用 proto 实例中的 connect()函数执行连接操作。

UDP 套接字关联 proto 实例中的连接函数为 **ip4_datagram_connect()**，定义如下（/net/ipv4/datagram.c）：

```
int ip4_datagram_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
{
    int res;

    lock_sock(sk);
    res = __ip4_datagram_connect(sk, uaddr, addr_len);    /*/net/ipv4/datagram.c*/
    release_sock(sk);
    return res;
}
```

__ip4_datagram_connect()函数定义如下：

```
int __ip4_datagram_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
{
    struct inet_sock *inet = inet_sk(sk);
    struct sockaddr_in *usin = (struct sockaddr_in *) uaddr;
    struct flowi4 *fl4;    /*指向 flowi4 实例，保存路由选择查找过程中的重要字段*/
    struct rtable *rt;    /*指向 rtable 实例，表示路由选择查找结果*/
    __be32 saddr;
    int oif;
    int err;
    ...   /*地址有效性检查*/
    sk_dst_reset(sk);    /*sk->sk_dst_cache 设为 NULL，dst_entry 结构体指针*/

    oif = sk->sk_bound_dev_if;    /*绑定的网络设备编号*/
    saddr = inet->inet_saddr;    /*源 IP 地址*/
    if (ipv4_is_multicast(usin->sin_addr.s_addr)) {    /*目的 IP 地址是否是组播地址*/
        if (!oif)
            oif = inet->mc_index;    /*组播网络设备编号*/
        if (!saddr)
            saddr = inet->mc_addr;    /*组播源地址*/
    }
    fl4 = &inet->cork.fl.u.ip4;
    rt = ip_route_connect(fl4, usin->sin_addr.s_addr, saddr,
                  RT_CONN_FLAGS(sk), oif,
                  sk->sk_protocol,
                  inet->inet_sport, usin->sin_port, sk);
                                    /*以源地址、目的地址执行路由选择查找，/include/net/route.h*/
    ...    /*错误处理*/

    if ((rt->rt_flags & RTCF_BROADCAST) && !sock_flag(sk, SOCK_BROADCAST)) {
        ip_rt_put(rt);
        err = -EACCES;
        goto out;
```

```
    }
    if (!inet->inet_saddr)     /*没有指定源地址*/
        inet->inet_saddr = fl4->saddr;   /*更新源 IP 地址，来自路由选择查找结果*/
    if (!inet->inet_rcv_saddr) {          /*如果没有指定接收 IP 地址*/
        inet->inet_rcv_saddr = fl4->saddr;
        if (sk->sk_prot->rehash)
            sk->sk_prot->rehash(sk);     /*inet_rcv_saddr 修改时，重新插入散列表*/
    }
    inet->inet_daddr = fl4->daddr;         /*目的 IP 地址*/
    inet->inet_dport = usin->sin_port;     /*目的端口号*/
    sk->sk_state = TCP_ESTABLISHED;     /*套接字状态设为 TCP_ESTABLISHED*/
    inet_set_txhash(sk);
    inet->inet_id = jiffies;     /*分段 id 值*/

    sk_dst_set(sk, &rt->dst);     /*设置 sk->sk_dst_cache 成员，dst_entry 实例指针*/
    err = 0;
out:
    return err;
}
```

连接函数根据源 IP 地址、目的 IP 地址执行路由选择查找，查找结果为 rtable 结构体实例，rtable.dst 成员（dst_entry 结构体）被赋予 sock 实例。目的套接字 IP 地址、端口号将保存在 inet_sock 结构体成员中。

2 路由选择

连接操作中将调用 ip_route_connect()函数执行路由选择，路由选择以目的 IP 地址在路由选择表中查找匹配的表项，返回结果为 rtable 结构体实例，实例中的 dst 成员（即 dst_entry 实例）指针将赋予套接字 sock 实例 sk_dst_cache 成员。其中 dst_entry 实例中包含数据包下一步操作的函数指针。

下面先简单看一下 ip_route_connect()函数的定义（/include/net/route.h），在第 13 章将详细介绍路由选择子系统：

```
static inline struct rtable *ip_route_connect(struct flowi4 *fl4,__be32 dst, __be32 src, u32 tos,
                                              int oif, u8 protocol,__be16 sport, __be16 dport,struct sock *sk)
/*fl4：指向 flowi4 实例，保存查找参数，dst：目的 IP 地址，src：源（本机）IP 地址，tos：服务类型，
*oif：输出网络接口编号，protocol：传输层协议类型，sport：源端口号，dport：目的端口号，
*sk：指向套接字 sock 实例。
*/
{
    struct net *net = sock_net(sk);
    struct rtable *rt;

    ip_route_connect_init(fl4, dst, src, tos, oif, protocol,sport, dport, sk);        /*初始化 flowi4 实例*/

    if (!dst || !src) {          /*如果目的 IP 地址，或源 IP 地址为 0*/
        rt = __ip_route_output_key(net, fl4);      /*输出通道路由选择核心函数，见第 13 章*/
        if (IS_ERR(rt))
            return rt;
        ip_rt_put(rt);
```

```
            flowi4_update_output(fl4, oif, tos, fl4->daddr, fl4->saddr);
                                                    /*更新 flowi4 实例，再执行路由选择*/
        }
        security_sk_classify_flow(sk, flowi4_to_flowi(fl4));
        return ip_route_output_flow(net, fl4, sk);          /*路由选择，/net/ipv4/route.c*/
    }
```

如果函数参数中传递的目的 IP 地址或本机 IP 地址为 0，将先调用__ip_route_output_key()函数确定目的或源 IP 地址，更新 flowi4 实例后，再调用 ip_route_output_flow()函数执行路由选择查找。

如果函数参数中传递的目的 IP 地址和本机 IP 地址都不为 0，则直接调用 ip_route_output_flow()函数执行路由选择查找。

ip_route_output_flow()函数定义如下：

```
struct rtable *ip_route_output_flow(struct net *net, struct flowi4 *flp4,struct sock *sk)
{
    struct rtable *rt = __ip_route_output_key(net, flp4);          /*输出通道路由选择核心函数*/

    if (IS_ERR(rt))
        return rt;
    if (flp4->flowi4_proto)
        rt = (struct rtable *)xfrm_lookup_route(net, &rt->dst,flowi4_to_flowi(flp4),sk, 0);
    return rt;      /*返回 rtable 实例指针*/
}
```

__ip_route_output_key()函数是网络层协议中输出通道路由选择的核心函数，详情见第 13 章。

12.6.4 参数与 IO 控制

用户进程可通过 setsockopt()/getsockopt()系统调用设置/获取套接字参数（含传输协议参数），通过 ioctl()系统调用设置/获取网络层协议、数据链路层协议（含网络设备）参数等。本小节介绍相关系统调用在 UDP 套接字中的实现。

1 设置/获取参数

setsockopt()/getsockopt()系统调用调用套接字关联 proto_ops 实例中的对应函数完成非 SOL_SOCKET 级别参数的设置/获取。UDP 套接字操作结构 inet_dgram_ops 实例中的 setsockopt()和 getsockopt()函数如下所示：

```
const struct proto_ops inet_dgram_ops = {
    .family             = PF_INET,
    ...
    .setsockopt         = sock_common_setsockopt,   /*设置参数，处理非 SOL_SOCKET 级别参数*/
    .getsockopt         = sock_common_getsockopt,   /*获取参数，处理非 SOL_SOCKET 级别参数*/
    ...
}
```

proto_ops 实例中的 setsockopt()和 getsockopt()函数用于设置/获取 SOL_SOCKET 级别（类型）以外的参数。下面以设置参数为例，介绍 setsockopt()函数 sock_common_setsockopt()的实现。

sock_common_setsockopt()函数定义如下（/net/core/sock.c）：

```
int sock_common_setsockopt(struct socket *sock, int level, int optname,char __user *optval,
                                                        unsigned int optlen)
```

```
/*level：参数级别，optname：参数名称（由整数标识），optval：参数值，optlen：参数长度*/
{
    struct sock *sk = sock->sk;

    return sk->sk_prot->setsockopt(sk, level, optname, optval, optlen);
}
```

参数 level 级别用于对参数进行分类，例如：按传输层协议分类参数，取值如下（/include/linux/socket.h）：

```
#define   SOL_IP          0       /*IPv4 参数*/
#define   SOL_TCP         6       /*TCP 参数*/
#define   SOL_UDP         17      /*UDP 参数*/
#define   SOL_IPV6        41      /*IPv6 参数*/
#define   SOL_ICMPV6      58
#define   SOL_SCTP        132
#define   SOL_UDPLITE     136       /* UDP-Lite (RFC 3828) */
...
```

参数 optname 表示参数名称，由一个整型数标识，optval 指向一个用户内存区，用于传递参数值，optlen 表示参数值长度。

sock_common_setsockopt()函数调用套接字 sock 关联 proto 实例中的 setsockopt()函数设置参数，对于 UDP 套接字此函数为 udp_setsockopt()（见 udp_prot 实例），函数定义如下（/net/ipv4/udp.c）：

```
int udp_setsockopt(struct sock *sk, int level, int optname,char __user *optval, unsigned int optlen)
{
    if (level == SOL_UDP   ||   level == SOL_UDPLITE)       /*设置 UDP/UDP-Lite 级别参数*/
        return udp_lib_setsockopt(sk, level, optname, optval, optlen,udp_push_pending_frames);
    return ip_setsockopt(sk, level, optname, optval, optlen);     /*设置 SOL_IP 级别参数*/
}
```

对于特定于 UDP 或 UDP-Lite 的参数由 udp_lib_setsockopt()函数处理，ip_setsockopt()函数处理 SOL_IP 级别参数。

udp_lib_setsockopt()函数定义在/net/ipv4/udp.c 文件内，函数源代码请读者自行阅读。UDP 私有参数名称定义在/include/uapi/linux/udp.h 头文件：

```
#define   UDP_CORK          1       /*抑制功能*/
#define   UDP_ENCAP         100     /* Set the socket to accept encapsulated packets */
#define   UDP_NO_CHECK6_TX        101       /* Disable sending checksum for UDP6X */
#define   UDP_NO_CHECK6_RX        102       /* Disable accpeting checksum for UDP6 */

/* UDP encapsulation types */
#define   UDP_ENCAP_ESPINUDP_NON_IKE      1 /* draft-ietf-ipsec-nat-t-ike-00/01 */
#define   UDP_ENCAP_ESPINUDP          2 /* draft-ietf-ipsec-udp-encaps-06 */
#define   UDP_ENCAP_L2TPINUDP         3 /* rfc2661 */
```

ip_setsockopt()函数定义在/net/ipv4/ip_sockglug.c 文件内，源代码请读者自行阅读。参数名称定义在头文件/include/uapi/linux/in.h，表示 IPv4 参数，例如：

```
#define   IP_TOS          1     /*服务类型*/
#define   IP_TTL          2     /*ttl 值*/
```

```
#define   IP_HDRINCL   3     /*用户设置 IP 报头*/
#define   IP_OPTIONS  4     /*IP 选项*/
#define   IP_ROUTER_ALERT   5
#define   IP_RECVOPTS     6
#define   IP_RETOPTS       7
#define   IP_PKTINFO       8
...
```

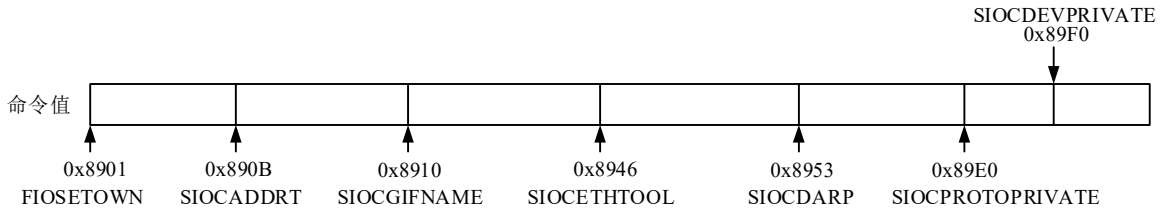
UDP 套接字 proto_ops 实例 inet_dgram_op 中获取参数的函数为 sock_common_getsockopt()，用于将套接字参数值传递到用户空间，源代码请读者自行阅读。

2 IO 控制

用户可通过 **ifconfig、route** 等命令设置/获取网络、网络设备参数，如路由选择表项、本机 IP 地址等等，这些用户工具通过 ioctl()系统调用实现。

ioctl()系统调用最终将调用套接字操作 proto_ops 实例中的 ioctl()函数完成操作，对于 UDP 套接字，此函数为 **inet_ioctl()**（见 inet_dgram_ops 实例），这个函数适用于 IPv4 协议簇中的多个套接字类型，不特于哪种传输层协议，因为它主要用于设置/获取网络层、数据链路层参数。

inet_ioctl()函数中命令参数取值定义在/include/uapi/linux/sockios.h 和/include/uapi/asm-generic/sockios.h 头文件，命令值按功能进行了分类，下图示意了几个命令分类的起始值。最后两个命令段分别是特定于传输层协议的命令和特定于网络设备的命令。



inet_ioctl()函数定义如下（/net/ipv4/af_inet.c）：

```
int inet_ioctl(struct socket *sock, unsigned int cmd, unsigned long arg)
/*sock：套接字 sock 实例，cmd：命令值，arg：命令参数值指针（用户空间内存地址）*/
{
    struct sock *sk = sock->sk;
    int err = 0;
    struct net *net = sock_net(sk);

    switch (cmd) {
    case SIOCGSTAMP:     /*获取时间戳*/
        err = sock_get_timestamp(sk, (struct timeval __user *)arg);
        break;
    case SIOCGSTAMPNS:
        err = sock_get_timestampns(sk, (struct timespec __user *)arg);
        break;
    case SIOCADDRT:
    case SIOCDELRT:
    case SIOCRTMSG:
        err = ip_rt_ioctl(net, cmd, (void __user *)arg);
```

```
                                        /*处理路由选择相关命令，/net/ipv4/fib_frontend.c*/
            break;
        case SIOCDARP:
        case SIOCGARP:
        case SIOCSARP:
            err = arp_ioctl(net, cmd, (void __user *)arg);   /*处理 ARP 相关命令，/net/ipv4/arp.c*/
            break;
        case SIOCGIFADDR:
        case SIOCSIFADDR:
        case SIOCGIFBRDADDR:
        case SIOCSIFBRDADDR:
        case SIOCGIFNETMASK:
        case SIOCSIFNETMASK:
        case SIOCGIFDSTADDR:
        case SIOCSIFDSTADDR:
        case SIOCSIFPFLAGS:
        case SIOCGIFPFLAGS:
        case SIOCSIFFLAGS:
            err = devinet_ioctl(net, cmd, (void __user *)arg);
                                        /*处理网络设备相关命令，/net/ipv4/devinet.c*/
            break;
        default:
            if (sk->sk_prot->ioctl)
                err = sk->sk_prot->ioctl(sk, cmd, arg);     /*udp_ioctl()，用于处理 SIOCOUTQ 等命令*/
            else
                err = -ENOIOCTLCMD;
            break;
        }
        return err;
    }
```

inet_ioctl()函数中具体命令的处理函数请读者自行阅读。

12.6.5 发送数据

UDP 报头在 Linux 内核中由 udphdr 结构体表示，结构体定义如下（/include/uapi/linux/udp.h）：

```
struct udphdr {
    __be16    source;     /*源端口号，网络字节序*/
    __be16    dest;       /*目的端口号*/
    __be16    len;        /*数据包总长度，不能超过 0xFFFF*/
    __sum16 check;        /*校验和*/
};
```

UDP 表示用户数据报传输协议，数据报传输是指数据是以数据报为单位传输的（具有消息边界，类似于 netlink 消息），而不像流套接字一样，是以字节流的形式传输的。用户每次传输的数据不能超过数据报的最大长度，对于 UDP 来说，数据报最大长度为 64KB（含 UDP 报头）。

由于 UDP 数据报需要外发，受数据链路层帧大小的限制（MTU），可能需要对数据报进行分段，生

成多个数据包（sk_buff 实例），以数据包的形式发送数据，在接收端再重组成数据报。

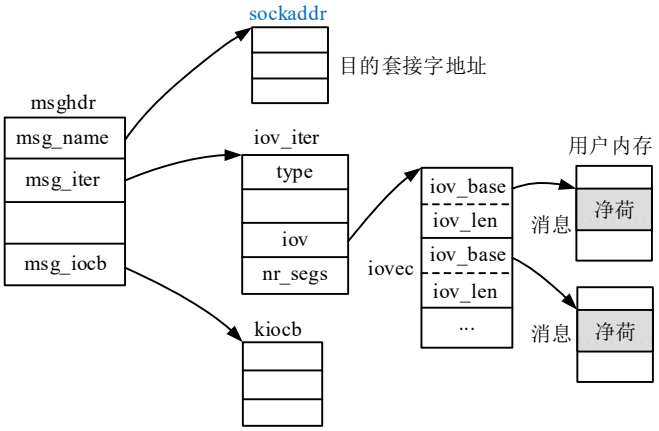
用户发送数据时，可通过一次写/发送系统调用产生一个数据报，也可以通过合并多次写/发送系统调用的数据累积生成一个数据报。但是，不管是通过一次还是多次系统调用产生数据报，数据报总长度都不能超过最大值（64KB）。

如果用户要将多次系统调用发送的数据合并成一个数据报，有两种方法：

（1）通过 setsockopt()系统调用 **UDP_CORK** 参数名称（SOL_UDP 级别），启用抑制功能，执行写/发送系统调用，最后取消抑制功能。抑制功能启用时发送数据包缓存在发送队列，抑制功能取消时，将发送缓存队列中数据包发送到网络层。

（2）在除最后一个写/发送系统调用中设置发送标记位 MSG_MORE（msghdr->msg_flags），在最后一个写/发送系统调用中清零此标记位。执行的效果与启用/取消抑制功能相同。

在前面介绍的套接字操作中，写操作 write()、send()、sendto()等系统调用最终调用 proto_ops 实例中的 sendmsg()函数发送数据。sendmsg()函数需要通过 msghdr 结构体传递发送的数据以及目的套接字地址，结构体如下所示，msghdr 结构体可以同时传递多个用户数据块。



UDP 套接字关联 proto_ops 实例 inet_dgram_ops 中 sendmsg()函数为 inet_sendmsg()，函数定义如下：

```
int inet_sendmsg(struct socket *sock, struct msghdr *msg, size_t size)      /*/net/ipv4/af_inet.c*/
{
    struct sock *sk = sock->sk;

    sock_rps_record_flow(sk);

    if (!inet_sk(sk)->inet_num && !sk->sk_prot->no_autobind &&inet_autobind(sk))
        return -EAGAIN;                          /*套接字未执行绑定则自动绑定*/

    return sk->sk_prot->sendmsg(sk, msg, size);     /*调用 proto 实例中的 sendmsg()函数*/
}
```

inet_sendmsg()函数内调用套接字关联 proto 实例中的 sendmsg()函数发送消息。UDP 套接字 udp_prot 实例中的 sendmsg()函数为 **udp_sendmsg()。**

1 发送函数

udp_sendmsg()函数定义如下（/net/ipv4/udp.c）：

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, size_t len)
/*sk：套接字 sock 实例，msg：指向 msghdr 结构体，指定发送数据和目的地址，len：发送数据长度*/
```

```
{
    struct inet_sock *inet = inet_sk(sk);
    struct udp_sock *up = udp_sk(sk);
    struct flowi4 fl4_stack;     /*flowi4 实例，保存路由选择查找过程中的信息*/
    struct flowi4 *fl4;
    int ulen = len;
    struct ipcm_cookie ipc;          /*保存数据包 IP 层信息（控制信息），/include/net/ip.h*/
    struct rtable *rt = NULL;       /*路由选择查找结果*/
    int free = 0;
    int connected = 0;
    __be32 daddr, faddr, saddr;     /*IP 地址*/
    __be16 dport;       /*端口号*/
    u8    tos;
    int err, is_udplite = IS_UDPLITE(sk);
    int corkreq = up->corkflag || msg->msg_flags&MSG_MORE;
                          /*是否启用了抑制（或合并）功能，累积数据成一个数据报后再发送*/
    int (*getfrag)(void *, char *, int, int, int, struct sk_buff *);        /*函数指针，复制用户数据到内核*/
    struct sk_buff *skb;
    struct ip_options_data opt_copy;     /*IP 选项*/

    if (len > 0xFFFF)        /*数据长度不能超过 0xFFFF，因为 UDP 数据包最大长度为 64KB*/
        return -EMSGSIZE;

    /*检查标记*/
    if (msg->msg_flags & MSG_OOB)     /* Mirror BSD error message compatibility */
        return -EOPNOTSUPP;

    ipc.opt = NULL;    /*IP 选项*/
    ipc.tx_flags = 0;     /*发送标记*/
    ipc.ttl = 0;         /*ttl*/
    ipc.tos = -1;       /*服务类型*/

    getfrag = is_udplite ? udplite_getfrag : ip_generic_getfrag;
                          /*将用户数据复制到内核空间的回调函数，/net/ipv4/ip_output.c*/

    fl4 = &inet->cork.fl.u.ip4;
    if (up->pending) { /*udp_sock 发送队列有挂起的数据包，合并后再发送（启用了抑制或合并）*/
        lock_sock(sk);
        if (likely(up->pending)) {
            if (unlikely(up->pending != AF_INET)) {
                release_sock(sk);
                return -EINVAL;
            }
            goto do_append_data;     /*持有锁，跳转至 do_append_data 处，进入慢速发送路径*/
        }
        release_sock(sk);
```

```
}

/*没有挂起的数据包，开启一个新的数据报，需要做一些准备工作*/
ulen += sizeof(struct udphdr);      /*数据包总长度（加上一个 UDP 报头）*/

/*获取并认证目的地址*/
if (msg->msg_name) {     /*msg 参数指定了目的地址*/
      DECLARE_SOCKADDR(struct sockaddr_in *, usin, msg->msg_name);
      if (msg->msg_namelen < sizeof(*usin))
            return -EINVAL;
      if (usin->sin_family != AF_INET) {
            if (usin->sin_family != AF_UNSPEC)
                  return -EAFNOSUPPORT;
      }

      daddr = usin->sin_addr.s_addr;     /*目的 IP 地址*/
      dport = usin->sin_port;        /*目的端口号*/
      if (dport == 0)       /*目的端口号不能为 0*/
            return -EINVAL;
} else {     /*msg 参数中未指定目的地址，采用 connect()操作中设置的地址*/
      if (sk->sk_state != TCP_ESTABLISHED)    /*connect()操作后设为 TCP_ESTABLISHED 状态*/
            return -EDESTADDRREQ;
      daddr = inet->inet_daddr;       /*inet_sock 中的目的 IP 地址和端口号*/
      dport = inet->inet_dport;
      connected = 1;
}

ipc.addr = inet->inet_saddr;          /*源 IP 地址*/
ipc.oif = sk->sk_bound_dev_if;     /*网络设备编号*/
sock_tx_timestamp(sk, &ipc.tx_flags);       /*发送时间戳*/

if (msg->msg_controllen) {     /*如果存在控制消息*/
      ...
}

if (!ipc.opt) {    /*如果不存在 IP 选项*/
     ...
}

saddr = ipc.addr;       /*源 IP 地址*/
ipc.addr = faddr = daddr;     /*目的 IP 地址*/

if (ipc.opt && ipc.opt->opt.srr) {    /*处理 IP 选项*/
      ...
}
tos = get_rttos(&ipc, inet);     /*服务类型*/
```

```
    if (sock_flag(sk, SOCK_LOCALROUTE) ||(msg->msg_flags & MSG_DONTROUTE) ||
        (ipc.opt && ipc.opt->opt.is_strictroute)) {
        tos |= RTO_ONLINK;     /*已连接上，/include/net/route.h*/
        connected = 0;
    }

    if (ipv4_is_multicast(daddr)) {     /*组播地址（224.0.0.0~239.255.255.255），/include/linux/in.h*/
        if (!ipc.oif)
            ipc.oif = inet->mc_index;     /*组播网络设备*/
        if (!saddr)
            saddr = inet->mc_addr;        /*组播源地址*/
        connected = 0;
    } else if (!ipc.oif)
        ipc.oif = inet->uc_index;     /*单播网络设备编号*/

    if (connected)     /*是否执了 connect()操作，此操作中将执行路由选择查找*/
        rt = (struct rtable *)sk_dst_check(sk, 0);
                                    /*如果执行了连接操作，将赋值 sock.sk_dst_cache 成员*/

    if (!rt) {     /*如果没有路由选择查找结果，执行路由选择查找*/
        struct net *net = sock_net(sk);

        fl4 = &fl4_stack;
        flowi4_init_output(fl4, ipc.oif, sk->sk_mark, tos,
                RT_SCOPE_UNIVERSE, sk->sk_protocol,
                inet_sk_flowi_flags(sk),
                faddr, saddr, dport, inet->inet_sport);       /*设置 fl4 指向的实例，/include/net/flow.h*/
        security_sk_classify_flow(sk, flowi4_to_flowi(fl4));
        rt = ip_route_output_flow(net, fl4, sk);     /*路由选择查找，见第 13 章，/net/ipv4/route.c*/
        ...     /*错误处理*/

        err = -EACCES;
        if ((rt->rt_flags & RTCF_BROADCAST) &&!sock_flag(sk, SOCK_BROADCAST))
            goto out;
        if (connected)
            sk_dst_set(sk, dst_clone(&rt->dst)); /*路由选择查找结果 dst_entry 实例赋予 sock 实例*/
    }   /*if (!rt)结束*/

    if (msg->msg_flags&MSG_CONFIRM)
        goto do_confirm;

back_from_confirm:
    saddr = fl4->saddr;     /*路由选择查找结果中的源 IP 地址*/
    if (!ipc.addr)       /*如果没有目的 IP 地址，使用路由选择查找的目的 IP 地址*/
        daddr = ipc.addr = fl4->daddr;
```

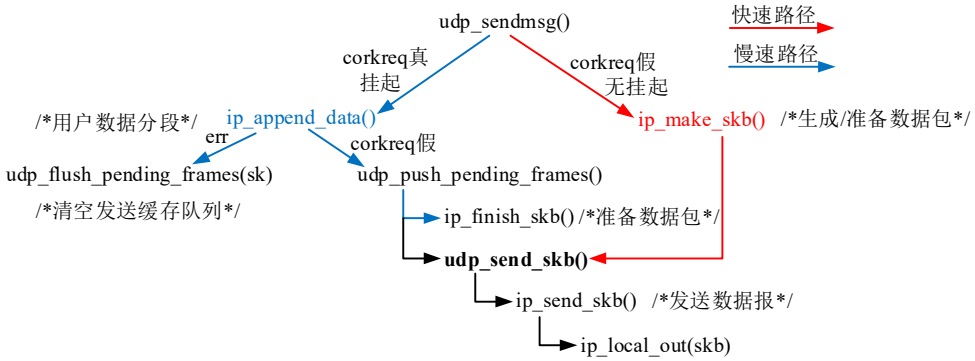
```
        /*corkreq 为 0，没有启用抑制（或合并）功能，进入快速路径，不需要持有套接字锁*/
        if (!corkreq) {
            skb = ip_make_skb(sk, fl4, getfrag, msg, ulen,sizeof(struct udphdr), &ipc, &rt,msg->msg_flags);
                                        /*生成发送数据包 sk_buff 实例（队列），/net/ipv4/ip_output.c*/
            err = PTR_ERR(skb);
            if (!IS_ERR_OR_NULL(skb))
                err = udp_send_skb(skb, fl4);     /*发送数据报，调用 ip_send_skb()函数，/net/ipv4/udp.c*/
            goto out;     /*跳至 out 处*/
        }

        /*corkreq 不为 0，启用了抑制（或合并）功能，持有锁进入慢速路径*/
        lock_sock(sk);
        if (unlikely(up->pending)) {
            ...   /*错误处理*/
        }
        fl4 = &inet->cork.fl.u.ip4;       /*设置 fl4 实例*/
        fl4->daddr = daddr;
        fl4->saddr = saddr;
        fl4->fl4_dport = dport;
        fl4->fl4_sport = inet->inet_sport;
        up->pending = AF_INET;       /*设置 up->pending 成员*/

    do_append_data:       /*慢速路径*/
        up->len += ulen;       /*增加发送缓存队列数据长度*/
        err = ip_append_data(sk, fl4, getfrag, msg, ulen,sizeof(struct udphdr), &ipc, &rt,
                             corkreq ? msg->msg_flags|MSG_MORE : msg->msg_flags);
                                        /*构建数据包并添加到发送缓存队列，/net/ipv4/ip_output.c*/
        if (err)        /*出错时*/
            udp_flush_pending_frames(sk);
                /*释放 sk->sk_write_queue 队列数据包，up->pending = 0，/net/ipv4/udp.c*/
        else if (!corkreq)       /*抑制（和合并）功能取消时，发送数据包*/
            err = udp_push_pending_frames(sk);       /*/net/ipv4/udp.c*/
        else if (unlikely(skb_queue_empty(&sk->sk_write_queue)))     /*发送缓存队列为空*/
            up->pending = 0;
        release_sock(sk);     /*释放套接字锁*/

    out:
        ...
        return err;
        ...
    }
```

udp_sendmsg()函数发送用户 UDP 数据包含两条路径，一是快速路径，二是慢速路径。

在发送数据时，如果 UDP 套接字发送缓存队列为空（没有挂起的数据包）并且没有启用抑制或合并功能时，将进入快速路径。快速路径中将对用户本次发送的数据进行分段（如果需要），生成一个临时数据包（队列），表示一个数据报，并立即发送到网络层。在此之前要做一些准备工作，如检查、设置目的地址，执行路由选择查找等，然后对用户数据进行分段，最后发送数据报。

在发送数据时，如果 UDP 套接字发送缓存队列不为空（有挂起的数据包）或者启用了抑制或合并功能时，将进入慢速路径。如果启用了抑制或合并功能，且发送缓存队列为空，则需要执行发送前的准备工作，然后对本次用户数据分段，将数据包添加到套接字发送缓存队列（不是快速路径中的临时队列），发送操作返回，不立即发送数据。下次用户发送数据时，发送缓存队不为空，将进入慢速路径，将本次数据与上次数据合并，生成数据包将添加到发送缓存队列。当用户取消抑制和合并功能时，发送缓存队列中的数据包将视为一个数据报，发送到网络层。

udp_sendmsg()函数发送用户数据路径简列如下图所示：

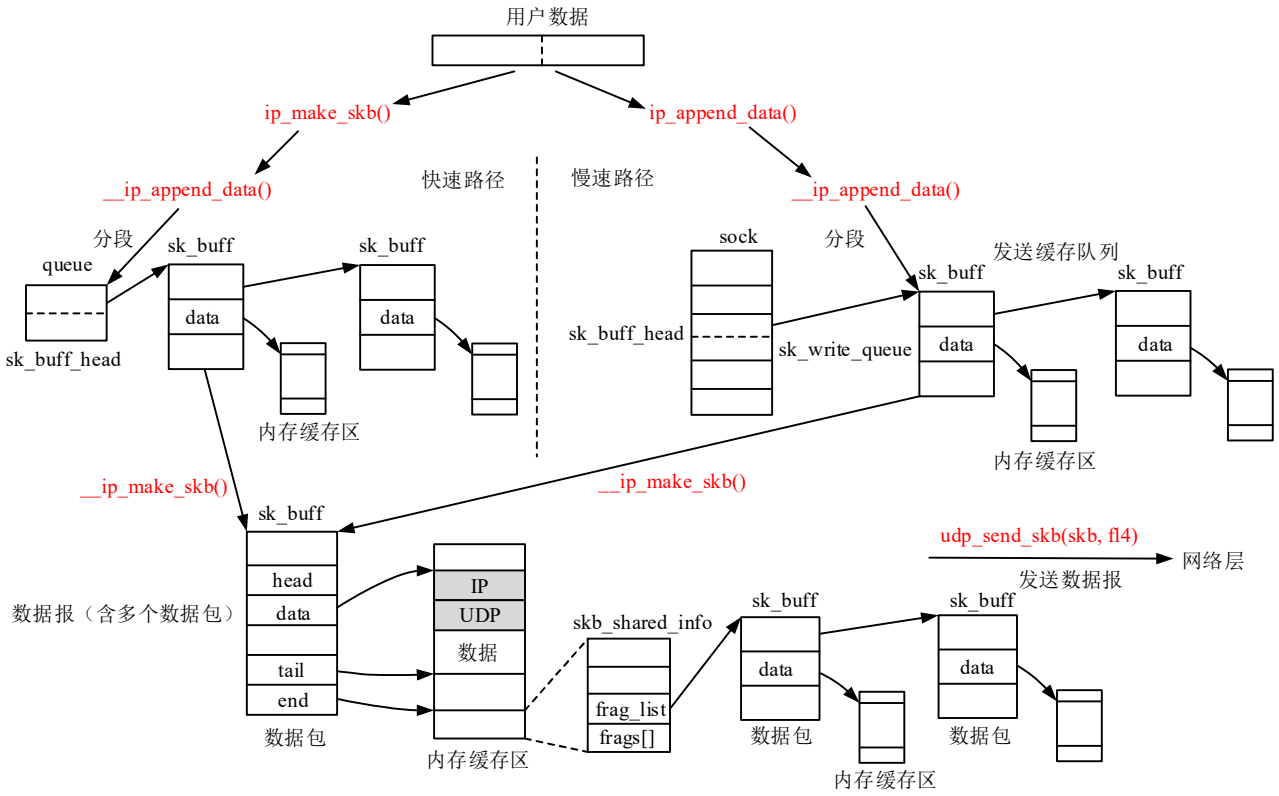


udp_sendmsg()函数中，当发送缓存队列没有挂起数据包且 corkreq 为假时（没有启用抑制和合并功能），发送路径进入快速路径，调用 **ip_make_skb()**函数对用户数据进行分段，生成发送数据包 sk_buff 实例（队列，保存在临时队列），然后调用函数__ip_make_skb()完成发送前的准备工作，最后调用 **udp_send_skb()** 函数（内部调用 ip_send_skb()函数）将本次发送数据视为一个数据报立即发送到网络层。

当发送缓存队列有挂起数据包或 corkreq 为真时，进入慢速路径。慢速路径调用 **ip_append_data**()函数将此次发送数据与发送缓存队列中数据合并，如果需要分段，则执行分段，分段数据包（队列）保存至发送缓存队列。当 corkreq 为假时，调用 **udp_push_pending_frames(sk)**函数将此时发送缓存队列中所有数据包视为同一个数据报，发送到网络层。

ip_append_data()函数返回错误码时，将调用 **udp_flush_pending_frames(sk)**函数清空套接字发送缓存队列。

下图示意了快速路径和慢速路径所执行的工作：



在快速路径和慢速路径中都调用__ip_append_data()函数对用户数据进行分段，构建一个 sk_buff 实例（或队列）。快速路径中 sk_buff 实例保存在临时队列中（不需要持有套接字锁），慢速路径中 sk_buff 实例保存在发送缓存队列中，如果原发送缓存队列不为空，则需要先填满队列中最后一个数据包，然后再创建新数据包。

__ip_make_skb()函数用于对 sk_buff 队列做发送前的准备工作，如将第一个数据包之后的数据包添加到第一个数据包 skb_shinfo(skb)->frag_list 链表，对第一个数据包写入 IP 报头等，将队列中的数据包视为一个数据报。udp_send_skb()函数用于最后将数据报发送到网络层。

下面先介绍快速路径和慢速路径中共用的用户数据分段函数和准备数据包函数的实现，后面再分别介绍快速路径和慢速路径中实现函数的定义。

■数据分段

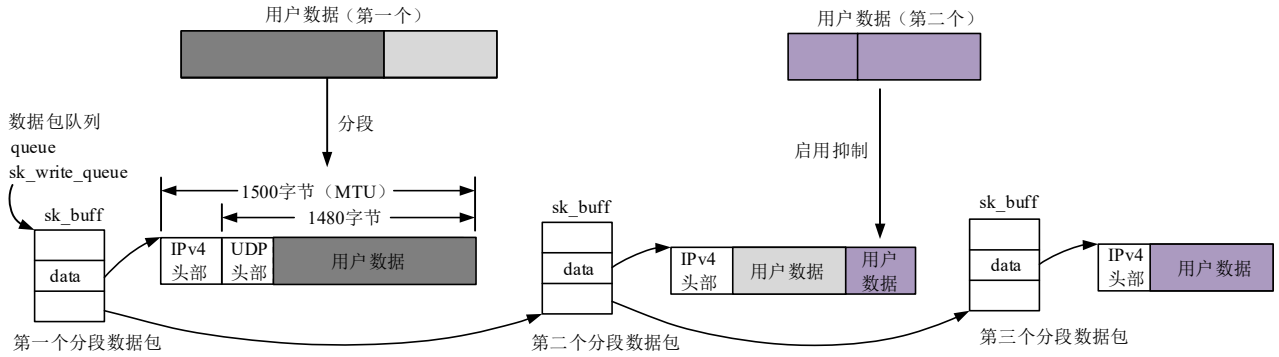
在对用户数据进行分段时，将根据网络接口（设备）是否设置 NETIF_F_UFO 标记位（UDPv4 分段），采用不同分段方式。

如没有设置 NETIF_F_UFO 标记位将采用常规分段方式，即将用户数据按 MTU 限制的大小分段，为每个段创建一个 sk_buff 实例，添加到数据包队列中。

如果设置了 NETIF_F_UFO 标记位，用户数据报将只创建一个 sk_buff 实例，其内存缓存区中只保存报头信息，所有用户数据保存在分散数据块中，这里称之为 UDPv4 分段。

●常规分段

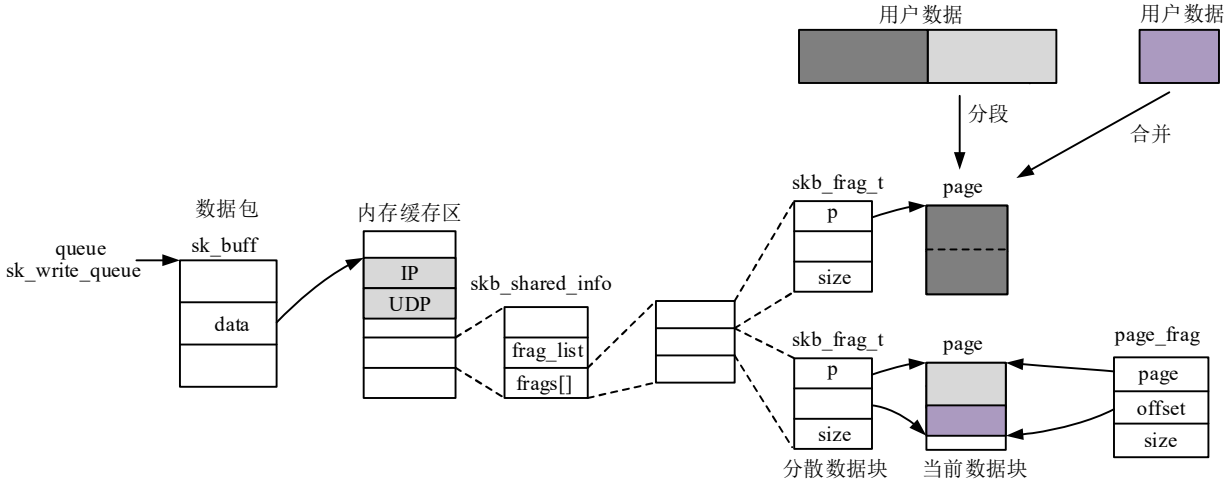
常规分段如下图所示，用户数据按 MTU 限制进行分段，每个段对应一个 sk_buff 实例，组成一个队列。在第一个数据包中需要包含 UDP 报头，而在第一个之后的数据包中不需要含有 UDP 报头，在接收端的 IP 层会重组数据包，然后再传递给传输层。



如果启用了抑制或合并功能，用户数据将先填满发送缓存队列中最后一个数据包，然后才创建新数据包，如上图中第二个用户数据所示。

●UDPv4 分段

如果网络接口（设备）设置了 NETIF_F_UFO 标记位，需同时设置 NETIF_F_SG（支持 Scatter/Gatter）、NETIF_F_GEN_CSUM（或 NETIF_F_IP_CSUM）标记位，所有用户数据将保存在同一个 sk_buff 实例的分散数据块中，如下图所示。



使用分散数据块时，在数据包内存缓区中只保存数据包的报头，而不保存用户数据。用户数据保存在 skb_shared_info 实例 frags[]数组中指示的分散数据块中。每个分散数据块由若干个连续（8 个 4KB 内存页）物理内存页组成，用户数据填入分散数据块中，也就是说用户数据按分散数据块大小进行划分。如果启用了抑制或合并功能，用户数据将与发送缓存队列中数据包现有分散数据块合并。

在分散数据块中保存用户数据时，将按顺序依次填充 frags[]数组中指示的分散数据块，一块填满后再创建新数据块，将用户数据填入其中。当前正在填充的分散数据块，称之为数据包的当前数据块。

在创建分散数据块的过程中内核通过 page_frag 结构体保存最近一次分配的分散数据块，并赋予 frags[]数组项。

page_frag 结构体定义如下（/include/linux/mm_types.h）：

```
struct page_frag {
    struct page *page;      /*物理页基指针*/
  #if (BITS_PER_LONG > 32) || (PAGE_SIZE >= 65536)
    __u32 offset;
    __u32 size;
  #else
    __u16 offset;      /*分散数据块空闲区域的起始偏移量（可变的）*/
    __u16 size;        /*分散数据块总大小*/
  #endif
```

```
};
```

sock、task_struct 结构体中内嵌 page_frag 结构体成员，用于保存当前分散数据块信息：

```
sock{
    ...
    struct page_frag     sk_frag;     /*内嵌 page_frag 结构体实例*/
    ...
}

task_struct{
    ...
    struct page_frag task_frag;
    ...
}
```

在将用户数据写入分散数据块时，将检查 page_frag 指向的当前分散数据块是否已填满，如果没有填满则填满此分散数据块，然后再分配新分散数据块，以容纳用户数据。当前分散数据块信息将写入对应 skb_shinfo(skb)->frags[]数组项中。

●分段函数

__ip_append_data()函数是一个非常重要的函数，它用于对用户数据进行分段。在快速路径中分段数据包保存在临时队列，在慢速路径中分段数据包保存在套接字发送缓存队列，并需要与队列中最末尾数据包合并。__ip_append_data()函数根据网络设备是否设置了 NETIF_F_UFO 标记位采用常规分段或 UDPv4 分段方式。

__ip_append_data()函数代码如下（/net/ipv4/ip_output.c）：

```
static int __ip_append_data(struct sock *sk,struct flowi4 *fl4,
                    struct sk_buff_head *queue,struct inet_cork *cork,struct page_frag *pfrag,
                    int getfrag(void *from, char *to, int offset,int len, int odd, struct sk_buff *skb),
                    void *from, int length, int transhdrlen,unsigned int flags)
/*queue：数据包队列头，快速路径中为临时队列头，慢速路径中为套接字发送缓存队列头，
*getfrag：复制用户数据到内核的接口函数，此处为 ip_generic_getfrag()，from：指向 msghdr 实例，
*length：用户数据长度（加上 UDP 报头），transhdrlen：UDP 报头长度，cork：指向 inet_cork 实例，
*flags：msg->msg_flags，发送接收标记，pfrag：指向 page_frag 结构体。
*/
{
    struct inet_sock *inet = inet_sk(sk);
    struct sk_buff *skb;

    struct ip_options *opt = cork->opt;     /*IP 选项*/
    int hh_len;
    int exthdrlen;
    int mtu;
    int copy;
    int err;
    int offset = 0;       /*用户数据偏移量，在分段过程中使用*/
    unsigned int maxfraglen, fragheaderlen, maxnonfragsize;
    int csummode = CHECKSUM_NONE;
```

```
struct rtable *rt = (struct rtable *)cork->dst;     /*路由选择查找结果*/
u32 tskey = 0;

skb = skb_peek_tail(queue);
                    /*指向 queue 队列最末尾 sk_buff 实例，可能为 NULL，/include/linux/skbuff.h*/

exthdrlen = !skb ? rt->dst.header_len : 0;       /*额外数据只在第一个数据包中存在*/
                                        /*数据链路层报头与网络层报头之间的额外数据*/
mtu = cork->fragsize;       /*分段长度*/
if (cork->tx_flags & SKBTX_ANY_SW_TSTAMP &&
                            sk->sk_tsflags & SOF_TIMESTAMPING_OPT_ID)
    tskey = sk->sk_tskey++;

hh_len = LL_RESERVED_SPACE(rt->dst.dev);     /*数据链路层报头长度*/

fragheaderlen = sizeof(struct iphdr) + (opt ? opt->optlen : 0);   /*分段数据包中 IP 报头长度*/
maxfraglen = ((mtu - fragheaderlen) & ~7) + fragheaderlen;
                                                /*分段最大长度（含 IP 报头），8 字节对齐*/
maxnonfragsize = ip_sk_ignore_df(sk) ? 0xFFFF : mtu;       /*由 inet_sk(sk)->pmtudisc 确定*/
                                    /*用户发送数据报最大长度（累积长度）*/

if (cork->length + length > maxnonfragsize - fragheaderlen) {
                                                    /*累积的发送数据长度不能超过最大值*/
    ip_local_error(sk, EMSGSIZE, fl4->daddr, inet->inet_dport,mtu - (opt ? opt->optlen : 0));
    return -EMSGSIZE;
}

/*传输层报头长度不为 0，开启一个新的数据报（快速路径或慢速路径中发送缓存队列为空）*/
if (transhdrlen &&length + fragheaderlen <= mtu &&
                        rt->dst.dev->features & NETIF_F_V4_CSUM &&!exthdrlen)
    csummode = CHECKSUM_PARTIAL;

cork->length += length;     /*累积发送数据报长度*/
/*网络设备支持 UDPv4 分段，只生成一个 sk_buff 实例，所有用户数据保存在分散数据块中*/
if (((length > mtu) || (skb && skb_is_gso(skb))) &&(sk->sk_protocol == IPPROTO_UDP)
                                &&(rt->dst.dev->features & NETIF_F_UFO) && !rt->dst.header_len
                                &&(sk->sk_type == SOCK_DGRAM)) {
    err = ip_ufo_append_data(sk, queue, getfrag, from, length,
                            hh_len, fragheaderlen, transhdrlen,maxfraglen, flags); /*/net/ipv4/ip_output.c*/
    if (err)
        goto error;
    return 0;
}

if (!skb)     /*queue 队列为空*/
    goto alloc_new_skb;     /*跳转到 alloc_new_skb 处执行，直接创建新 sk_buff 实例*/
```

```
    /*queue 队列不为空时，继续执行以下代码*/
    while (length > 0) {        /*循环对用户数据进行分段*/
        copy = mtu - skb->len;
        if (copy < length)
            copy = maxfraglen - skb->len;
        if (copy <= 0) {
            /*copy 表示在当前 sk_buff 实例中还能填入的数据长度，
            *小于等于 0 表示需要创建新 sk_buff 实例，以容纳用户数据。
            */
            char *data;   /*用户数据区指针*/
            unsigned int datalen;
            unsigned int fraglen;
            unsigned int fraggap;
            unsigned int alloclen;
            struct sk_buff *skb_prev;
alloc_new_skb:    /*分配新 sk_buff 实例*/
            skb_prev = skb;       /*skb 可能为 NULL*/
            if (skb_prev)
                fraggap = skb_prev->len - maxfraglen;
            else
                fraggap = 0;

            datalen = length + fraggap;         /*用户数据长度*/
            if (datalen > mtu - fragheaderlen)     /*用户数据需要分段*/
                datalen = maxfraglen - fragheaderlen; /*一个分段中用户数据长度（不含 IP 报头）*/
            fraglen = datalen + fragheaderlen;        /*分段长度（含 IP 报头）*/

            if ((flags & MSG_MORE) &&!(rt->dst.dev->features&NETIF_F_SG))
                alloclen = mtu;                       /*网络设备不支持分散/聚集，且启用合并*/
            else
                alloclen = fraglen;       /*网络设备支持分散/聚集，fraglen≦mtu*/

            alloclen += exthdrlen;       /*mtu 加上额外的长度，用于分配内存缓存区*/

            if (datalen == length + fraggap)
                alloclen += rt->dst.trailer_len;

            if (transhdrlen) {     /*数据报中第一个数据包*/
                skb = sock_alloc_send_skb(sk,alloclen + hh_len + 15,
                                          (flags & MSG_DONTWAIT), &err);
                                          /*分配 sk_buff 实例及缓存区，/net/core/sock.c*/
            } else {   /*不是数据报中第一个数据包*/
                skb = NULL;
                if (atomic_read(&sk->sk_wmem_alloc) <=2 * sk->sk_sndbuf)
                    skb = sock_wmalloc(sk,alloclen + hh_len + 15, 1,sk->sk_allocation);
```

```
            /*分配 sk_buff 实例及缓存区，受限于套接字发送缓存区大小，/net/core/sock.c*/
        ...  /*错误处理*/
    }
    ...  /*错误处理*/
    skb->ip_summed = csummode;
    skb->csum = 0;
    skb_reserve(skb, hh_len);       /*skb->data 和 skb->tail 都后移 hh_len 字节*/

    /* only the initial fragment is time stamped */
    skb_shinfo(skb)->tx_flags = cork->tx_flags;
    cork->tx_flags = 0;
    skb_shinfo(skb)->tskey = tskey;
    tskey = 0;

    data = skb_put(skb, fraglen + exthdrlen);     /*在内存缓存区后面增加空间，增加 skb->len*/
    skb_set_network_header(skb, exthdrlen);     /*设置 IP 报头指针*/
    skb->transport_header = (skb->network_header +fragheaderlen);     /*设置传输层报头*/
    data += fragheaderlen + exthdrlen;   /*data 指向 IP 层报头之后的 UDP 报头或用户数据*/

    if (fraggap) {   /*fraggap 区用于保存校验和？？ */
        skb->csum = skb_copy_and_csum_bits(skb_prev, maxfraglen,data + transhdrlen,
                                                                    fraggap, 0);
        skb_prev->csum = csum_sub(skb_prev->csum,skb->csum);
        data += fraggap;
        pskb_trim_unique(skb_prev, maxfraglen);
    }

    copy = datalen - transhdrlen - fraggap;         /*复制用户数据长度*/
    /*复制数据到 sk_buff 内存缓存区，getfrag()此处为 ip_generic_getfrag()函数*/
    if (copy > 0 && getfrag(from, data + transhdrlen, offset, copy, fraggap, skb) < 0) {
        ... /*错误处理*/
    }

    offset += copy;             /*下次分段数据的起始偏移量（用户数据）*/
    length -= datalen - fraggap;     /*剩余没有分段的数据长度*/
    transhdrlen = 0;     /*第二个数据包开始，不需要传输层报头和额外数据*/
    exthdrlen = 0;
    csummode = CHECKSUM_NONE;

    __skb_queue_tail(queue, skb);     /*将 sk_buff 实例添加到 queue 队列末尾*/
    continue;         /*回到 while (length > 0)循环开始处*/
}      /*if (copy <= 0)结束，数据分段结束*/

/*下面是处理 copy>0 的情形，即将用户数据开头的 copy 字节填充到队列最末尾 sk_buff 实例，
*然后，再跳转到  while ()循环处对剩下的用户数据进行分段。
*/
```

```
        if (copy > length)
            copy = length;

        if (!(rt->dst.dev->features&NETIF_F_SG)) {      /*网络设备不支持分散/聚集功能*/
            unsigned int off;                           /*复制 copy 数据至最末尾数据包*/
            off = skb->len;
            if (getfrag(from, skb_put(skb, copy),offset, copy, off, skb) < 0) {
                                        /*getfrag()此处为 ip_generic_getfrag()函数*/
                __skb_trim(skb, off);
                err = -EFAULT;
                goto error;
            }
        } else {      /*网络设备支持分散/聚集功能，复制用户数据至当前分散数据块*/
            int i = skb_shinfo(skb)->nr_frags;      /*数据包中现有分散数据块数量*/

            err = -ENOMEM;
            if (!sk_page_frag_refill(sk, pfrag))       /*/net/core/sock.c*/
                        /*确保 page_frag 指向分散数据块中有至少 32 字节的空闲区域，
                        *如果没有则分配新分散数据块，赋予 page_frag 实例*/
                goto error;

            if (!skb_can_coalesce(skb, i, pfrag->page,pfrag->offset)) {
                  /*是否创建了新分散数据块，或数据包中分散数据块为空，返回假*/
                err = -EMSGSIZE;
                if (i == MAX_SKB_FRAGS)
                    goto error;
                /*创建了新分散数据块，将信息赋予 skb_shinfo(skb)->frags[i]数组项*/
                __skb_fill_page_desc(skb, i, pfrag->page,pfrag->offset, 0);
                skb_shinfo(skb)->nr_frags = ++i;
                get_page(pfrag->page);
            }
            /*复制用户数据到分散数据块*/
            copy = min_t(int, copy, pfrag->size - pfrag->offset);     /*复制用户数据长度*/
            if (getfrag(from,page_address(pfrag->page) + pfrag->offset,offset, copy, skb->len, skb) < 0)
                goto error_efault;

            pfrag->offset += copy;   /*空闲区域起始偏移量*/
            skb_frag_size_add(&skb_shinfo(skb)->frags[i - 1], copy);      /*更新数据块中数据长度*/
            skb->len += copy;          /*调整数据包长度*/
            skb->data_len += copy;     /*分散数据块中数据长度*/
            skb->truesize += copy;
            atomic_add(copy, &sk->sk_wmem_alloc);
        }
        offset += copy;   /*处理 copy 字节以后的用户数据*/
        length -= copy;
    }   /*while (length > 0)  循环结束*/
```

```
    return 0;     /*成功返回 0*/
    ...
}
```

在__ip_append_data()函数中，如果网络接口设置了 NETIF_F_UFO 标记位，将调用 ip_ufo_append_data()函数，将所有用户数据填充到队列中数据包（或新创建数据包）的分散数据块中。ip_ufo_append_data()函数定义在/net/ipv4/ip_output.c 文件内，源代码请读者自行阅读。

如果网络接口没有设置 NETIF_F_UFO 标记位，将按 MTU 限制对用户数据进行分段，对每个分段创建 sk_buff 实例，添加到 queue 队列。如果 queue 队列起初不为空（启用了抑制或合并功能），将检查 queue 队列最末尾 sk_buff 实例内存缓存区是否已填满，如果未填满则先从用户数据划出部分数据填充满最末尾 sk_buff 实例，然后对剩下的用户数据进行分段，创建新 sk_buff 实例，并添加到 queue 队列。

■准备数据包

对用户数据进行分段后，用户数据将由 sk_buff 实例或队列表示。__ip_make_skb()函数用于对包含用户数据的 sk_buff 实例或队列执行一些发送前的准备工作。

__ip_make_skb()函数执行的主要工作有：将数据包队列中第一个 sk_buff 实例之后的实例添加到第一个 sk_buff 实例内存缓存区 skb_shinfo(skb)->frag_list 链表，对第一个数据包写入 IP 报头等。

__ip_make_skb()函数定义在/net/ipv4/ip_output.c 文件内，代码如下：

```
struct sk_buff *__ip_make_skb(struct sock *sk,struct flowi4 *fl4,
                              struct sk_buff_head *queue,struct inet_cork *cork)
{
    struct sk_buff *skb, *tmp_skb;
    struct sk_buff **tail_skb;     /**tail_skb 指向 frag_list 链表末尾 sk_buff 实例*/
    struct inet_sock *inet = inet_sk(sk);
    struct net *net = sock_net(sk);
    struct ip_options *opt = NULL;
    struct rtable *rt = (struct rtable *)cork->dst;     /*路由查找结果*/
    struct iphdr *iph;
    __be16 df = 0;
    __u8 ttl;

    skb = __skb_dequeue(queue);     /*取 queue 队列中第一个 sk_buff 实例，/include/linux/skbuff.h*/
    if (!skb)
        goto out;
    tail_skb = &(skb_shinfo(skb)->frag_list);   /*内存缓存区 skb_shared_info 实例中 frag_list 链表*/

    /*移动 skb->data 指针，为 IP 报头预留空间*/
    if (skb->data < skb_network_header(skb))
        __skb_pull(skb, skb_network_offset(skb));
    while ((tmp_skb = __skb_dequeue(queue)) != NULL) {    /*逐个取出 queue 队列 sk_buff 实例*/
            /*将 sk_buff 实例移入(skb_shinfo(skb)->frag_list 链表，为各实例 IP 报头预留空间*/
        __skb_pull(tmp_skb, skb_network_header_len(skb));     /*预留 IP 报头空间*/
        *tail_skb = tmp_skb;          /*将 sk_buff 实例添加到 frag_list 链表末尾*/
        tail_skb = &(tmp_skb->next);
        skb->len += tmp_skb->len;          /*所有数据包数据长度值，记录在第一个 sk_buff 实例中*/
```

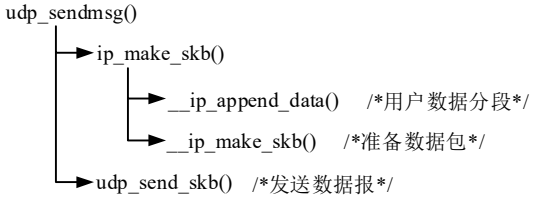
```
        skb->data_len += tmp_skb->len;      /*分段数据包中数据长度*/
        skb->truesize += tmp_skb->truesize;
        tmp_skb->destructor = NULL;
        tmp_skb->sk = NULL;
}

skb->ignore_df = ip_sk_ignore_df(sk);      /*是否忽略分段，/include/net/ip.h*/
if (inet->pmtudisc == IP_PMTUDISC_DO ||
        inet->pmtudisc == IP_PMTUDISC_PROBE ||
        (skb->len <= dst_mtu(&rt->dst) &&ip_dont_fragment(sk, &rt->dst)))
        df = htons(IP_DF);      /*满足以上条件，设置禁止分段标志*/

if (cork->flags & IPCORK_OPT)
        opt = cork->opt;        /*IP 选项*/

if (cork->ttl != 0)    /*确定 IP 报头中 ttl 值*/
        ttl = cork->ttl;
else if (rt->rt_type == RTN_MULTICAST)
        ttl = inet->mc_ttl;
else
        ttl = ip_select_ttl(inet, &rt->dst);

iph = ip_hdr(skb);      /*第一个数据包中 IP 报头指针，写入报头*/
iph->version = 4;
iph->ihl = 5;
iph->tos = (cork->tos != -1) ? cork->tos : inet->tos;      /*服务类型*/
iph->frag_off = df;      /*禁止分段标志*/
iph->ttl = ttl;
iph->protocol = sk->sk_protocol;    /*传输层协议类型*/
ip_copy_addrs(iph, fl4);      /*从 fl4 中复制源 IP 地址和目的 IP 地址，/net/ipv4/ip_output.c*/
ip_select_ident(net, skb, sk);      /*生成分段标识 id 值，/include/net/ip.h*/

if (opt) {      /*如果存在 IP 选项*/
        iph->ihl += opt->optlen>>2;
        ip_options_build(skb, opt, cork->addr, rt, 0);    /*构建选项*/
}

skb->priority = (cork->tos != -1) ? cork->priority: sk->sk_priority;      /*设置优先级*/
skb->mark = sk->sk_mark;

cork->dst = NULL;
skb_dst_set(skb, &rt->dst);    /*设置 sk_buff 实例 dst_entry 实例指针为 rt->dst*/

if (iph->protocol == IPPROTO_ICMP)      /*如果传输层协议为 ICMP*/
        icmp_out_count(net, ((struct icmphdr *)skb_transport_header(skb))->type);
```

```
    ip_cork_release(cork);
out:
    return skb;     /*返回第一个 sk_buff 实例*/
}
```

__ip_make_skb()函数取出 queue 队列中第一个 sk_buff 实例，随后取出 queue 队列的其它 sk_buff 实例，添加到第一个实例 skb_shinfo(skb)->frag_list 链表中，填充第一个 sk_buff 实例中的 IP 报头，其它 sk_buff 实例中预留 IP 报头空间，最后返回第一个 sk_buff 实例指针。

2 快速路径

在快速发送路径中，udp_sendmsg()函数将调用 **ip_make_skb()**函数，由用户数据生成发送数据包（队列），然后调用函数 **udp_send_skb()**将 UDP 数据报发送到网络层，函数调用关系简列如下。



■生成数据报

ip_make_skb()函数用于由用户数据生成发送数据报 sk_buff 实例（队列），此函数内分两步：

一是对用户数据进行分段，生成多个 sk_buff 实例，并添加到临时队列；二是准备数据包，将数据包从临时队列中取出，将第一个之后的数据包添加到第一个数据包 **skb_shinfo(skb)->frag_list** 链表，第一个数据包写入 IP 报头等，等待发送。

ip_make_skb()函数定义如下（/net/ipv4/ip_output.c）：

```
struct sk_buff *ip_make_skb(struct sock *sk,struct flowi4 *fl4,
                            int getfrag(void *from, char *to, int offset,int len, int odd, struct sk_buff *skb),
                            void *from, int length, int transhdrlen,struct ipcm_cookie *ipc, struct rtable **rtp,
                            unsigned int flags)
/*sk：套接字，fl4：flowi4 结构体，保存路由查找中间结果，getfrag：复制用户数据到内核的函数，
*from：指向 msghdr 实例，length：数据长度（加上 UDP 报头），transhdrlen：UDP 报头长度，
*ipc：指向 ipcm_cookie 实例，*rtp：指向路由选择查找结果，flags：msg->msg_flags。
*/
{
    struct inet_cork cork;      /*inet_cork 结构体，局部变量*/
    struct sk_buff_head queue;    /*sk_buff 队列头结构，临时数据包队列*/
    int err;

    if (flags & MSG_PROBE)    /*不需要发送数据包，调用者只是为了获取部分信息，如 MTU*/
        return NULL;

    __skb_queue_head_init(&queue);  /*初始化临时队列头*/

    cork.flags = 0;
    cork.addr = 0;
    cork.opt = NULL;
```

```
        err = ip_setup_cork(sk, &cork, ipc, rtp);     /*由 ipc、rtp 设置 cork 实例，/net/ipv4/ip_output.c*/
        if (err)
            return ERR_PTR(err);

        err = __ip_append_data(sk, fl4, &queue, &cork,&current->task_frag, getfrag,
                               from, length, transhdrlen, flags);
                    /*将用户数据转成 sk_buff 实例，添加到临时队列 queue，/net/ipv4/ip_output.c*/
        if (err) {     /*出错*/
            __ip_flush_pending_frames(sk, &queue, &cork);     /*清空发送缓存队列，/net/ipv4/ip_output.c*/
            return ERR_PTR(err);
        }

        return    __ip_make_skb(sk, fl4, &queue, &cork);       /*准备发送数据包*/
                        /*填充数据包 IP 报头，返回首个 sk_buff 实例指针等，/net/ipv4/ip_output.c*/
    }
```

ip_make_skb()函数内调用的__ip_append_data()和__ip_make_skb()函数，前面都介绍过了，不再重复了。

■发送数据包

快速路径中调用 ip_make_skb()函数生成 sk_buff 实例（链表）后，调用 **udp_send_skb(skb, fl4)**函数发送数据报，函数代码如下（/net/ipv4/udp.c）：

```
    static int udp_send_skb(struct sk_buff *skb, struct flowi4 *fl4)
    /*skb：sk_buff 实例，其 skb_shinfo(skb)->frag_list 链表可能包含分段 sk_buff 实例*/
    {
        struct sock *sk = skb->sk;
        struct inet_sock *inet = inet_sk(sk);
        struct udphdr *uh;
        int err = 0;
        int is_udplite = IS_UDPLITE(sk);
        int offset = skb_transport_offset(skb);
        int len = skb->len - offset;
        __wsum csum = 0;

        /*填充 UDP 报头*/
        uh = udp_hdr(skb);
        uh->source = inet->inet_sport;
        uh->dest = fl4->fl4_dport;
        uh->len = htons(len);
        uh->check = 0;

        if (is_udplite)                          /* UDP-Lite*/
            csum = udplite_csum(skb);

        else if (sk->sk_no_check_tx) {     /*是否关闭了校验和*/
            skb->ip_summed = CHECKSUM_NONE;
            goto send;
```

```
    } else if (skb->ip_summed == CHECKSUM_PARTIAL) { /*硬件实现校验和*/

        udp4_hwcsum(skb, fl4->saddr, fl4->daddr);
        goto send;

    } else
        csum = udp_csum(skb);     /*计算校验和*/

    /*添加伪头部计算得校验和*/
    uh->check = csum_tcpudp_magic(fl4->saddr, fl4->daddr, len,sk->sk_protocol, csum);
    if (uh->check == 0)
        uh->check = CSUM_MANGLED_0;

send:
    err = ip_send_skb(sock_net(sk), skb);     /*发送数据报，/net/ipv4/ip_output.c*/
    if (err) {
        ...  /*错误处理，增加统计量*/
    } else
        UDP_INC_STATS_USER(sock_net(sk),UDP_MIB_OUTDATAGRAMS, is_udplite);
                                              /*增加统计量*/
    return err;
}
```

udp_send_skb()函数比较简单，主要工作是填充第一个 sk_buff 实例中的 UDP 报头，计算校验和，然后交由 **ip_send_skb()**函数将数据报发送到网络层。

ip_send_skb()函数定义如下（/net/ipv4/ip_output.c）：

```
int ip_send_skb(struct net *net, struct sk_buff *skb)
{
    int err;

    err = ip_local_out(skb);      /*网络层协议发送数据包函数*/
    if (err) {
        ...    /*错误处理*/
    }
    return err;
}
```

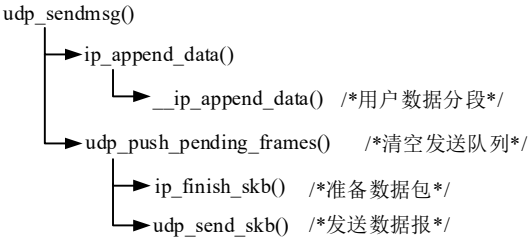
ip_local_out(skb)函数是网层发送数据包的函数，详见第 13 章。

3 慢速路径

慢速路径是指用户启用了抑制或合并功能，将用户多次发送的数据合并成一个数据报，在抑制或合并功能取消时，发送数据报。快速路径是为用户的每次数据发送构建一个单独的数据报，并立即发送。

慢速路径将用户发送数据构建的 sk_buff 实例（队列），缓存到套接字发送缓存队列，在取消抑制或合并功能时，再发送缓存队列中的数据包（数据报）。

慢速路径中函数调用关系简列如下图所示：



■用户数据分段

ip_append_data()函数对用户数据进行分段，并将分段数据包添加到套接字发送缓存队列，函数代码如下（/net/ipv4/ip_output.c）：

```
int ip_append_data(struct sock *sk, struct flowi4 *fl4,
        int getfrag(void *from, char *to, int offset, int len,int odd, struct sk_buff *skb),
        void *from, int length, int transhdrlen,
        struct ipcm_cookie *ipc, struct rtable **rtp,unsigned int flags)
{
    struct inet_sock *inet = inet_sk(sk);
    int err;

    if (flags&MSG_PROBE)
        return 0;

    if (skb_queue_empty(&sk->sk_write_queue)) {     /*如果发送缓存队列为空*/
        err = ip_setup_cork(sk, &inet->cork.base, ipc, rtp); /*设置 inet->cork.base 实例（inet_cork）*/
        if (err)
            return err;
    } else {
        transhdrlen = 0;
    }

  return    __ip_append_data(sk, fl4, &sk->sk_write_queue, &inet->cork.base,sk_page_frag(sk),
                                        getfrag,from, length, transhdrlen, flags);
}
```

此处在调用__ip_append_data()函数时，数据包队列头 queue 为套接字发送缓存队列 **sk->sk_write_queue**，inet_cork 实例为 udp_sock 内嵌 inet_sock 结构体 cork.base 成员，这两个都是属于套接字的实例。因此在调用 ip_append_data()函数时需要持有套接字锁。

__ip_append_data()函数在前面介绍过了，主要工作就是对用户数据进行分段，构建 sk_buff 实例（队列），并添加到 sk->sk_write_queue 队列。

■发送数据包

在慢速发送路径中，如果 corkreq 清零（抑制或合并功能取消了），将调用 **udp_push_pending_frames(sk)** 函数将套接字发送缓存队列中所有数据包视为一个数据报，发送到网络层，函数定义如下（/net/ipv4/udp.c）：

```
int udp_push_pending_frames(struct sock *sk)
{
```

```
    struct udp_sock    *up = udp_sk(sk);
    struct inet_sock *inet = inet_sk(sk);
    struct flowi4 *fl4 = &inet->cork.fl.u.ip4;
    struct sk_buff *skb;
    int err = 0;

    skb = ip_finish_skb(sk, fl4);    /*准备数据包，/include/net/ip.h*/
    if (!skb)
        goto out;

    err = udp_send_skb(skb, fl4);     /*发送数据报，见上文*/

out:
    up->len = 0;          /*udp_send_skb()函数会清空发送缓存队列*/
    up->pending = 0;
    return err;
}
```

udp_push_pending_frames()函数调用 ip_finish_skb(sk, fl4)函数完成发送缓存队列 sk->sk_write_queue 中数据包的准备工作，然后调用 udp_send_skb(skb, fl4)函数发送数据包，这个函数前面介绍过了。

ip_finish_skb(sk, fl4)函数调用前面介绍的__ip_make_skb()函数，定义如下（/include/net/ip.h）：

```
static inline struct sk_buff *ip_finish_skb(struct sock *sk, struct flowi4 *fl4)
{
    return __ip_make_skb(sk, fl4, &sk->sk_write_queue, &inet_sk(sk)->cork.base);
}
```

■清空发送队列

在慢速发送路径中，当 ip_append_data()返回非零值时（出错），将调用 udp_flush_pending_frames()函数清空发送缓存队列（直接释放 sk_buff 实例，不发送到网络层），将 up->pending 清零。

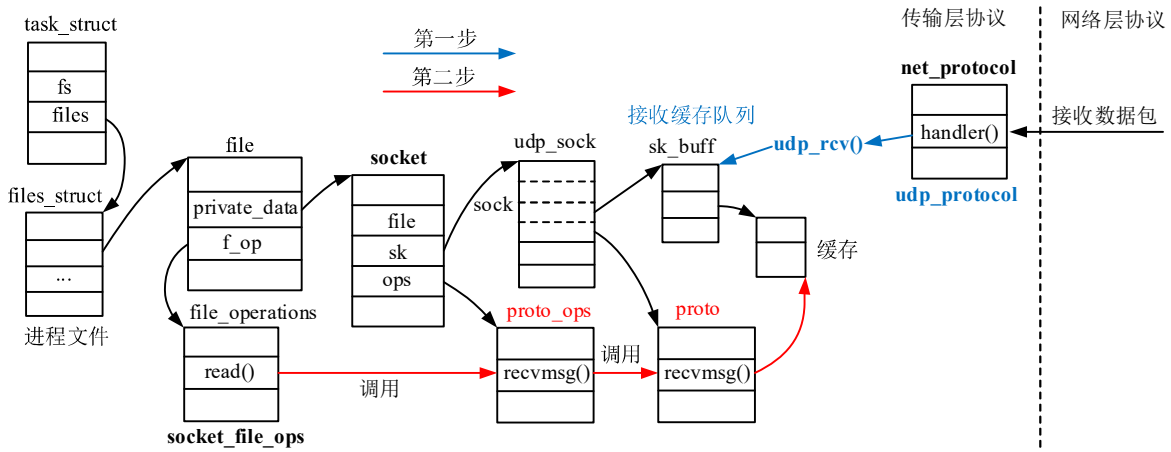
udp_flush_pending_frames()函数定义如下（/net/ipv4/udp.c）：

```
void udp_flush_pending_frames(struct sock *sk)
{
    struct udp_sock *up = udp_sk(sk);

    if (up->pending) {              /*up->pending 非 0*/
        up->len = 0;                /*发送缓存队列数据长度清零*/
        up->pending = 0;            /*up->pending 设为 0*/
        ip_flush_pending_frames(sk);    /*清空发送缓存队列，  /net/ipv4/ip_output.c*/
    }
}
```

ip_flush_pending_frames(sk)函数逐个取出发送缓存队列中 sk_buff 实例，并释放，源代码请读者自行阅读。

12.6.6 接收数据

用户接收数据过程主要分两步，一是 UDP 从 IP 层接收数据报，添加到接收方套接字接收缓存队列，二是用户进程从套接字接收缓存队列中读取数据，如下图所示。



传输层协议需要定义 net_protocol 结构体实例，网络层接收到数据包后根据报头中传输层协议类型，查找对应的 net_protocol 实例，并调用其中的 handler()函数接收数据包。对于 UDP，net_protocol 实例为 udp_protocol，接收数据包函数为 **udp_rcv()**。

用户进程通过系统调用读取接收缓存队列数据包中的数据，函数调用关系为：

系统调用->proto_ops.recvmsg()->proto.recvmsg()。

1 UDP 接收数据包

内核在/net/ipv4/af_inet.c 文件内定义了 UDP 协议对应的 net_protocol 实例：

```
static const struct net_protocol udp_protocol = {
        .early_demux =      udp_v4_early_demux,
        .handler =      udp_rcv,        /*UDP 接收数据包函数*/
        .err_handler = udp_err,
        .no_policy =    1,
        .netns_ok =     1,
};
```

内核在初始化函数 inet_init()中注册了 net_protocol 实例，实例中接收数据包的函数为 **udp_rcv()**，函数定义如下（/net/ipv4/udp.c）：

```
int udp_rcv(struct sk_buff *skb)
{
        return __udp4_lib_rcv(skb, &udp_table, IPPROTO_UDP);
}
```

__udp4_lib_rcv()函数定义如下：

```
int __udp4_lib_rcv(struct sk_buff *skb, struct udp_table *udptable,int proto)
{
        struct sock *sk;
        struct udphdr *uh;
        unsigned short ulen;
        struct rtable *rt = skb_rtable(skb);
        __be32 saddr, daddr;
        struct net *net = dev_net(skb->dev);

        /*数据包有效性检查*/
        if (!pskb_may_pull(skb, sizeof(struct udphdr)))
```

```
        goto drop;              /* No space for header. */

uh      = udp_hdr(skb);       /*UDP 报头*/
ulen = ntohs(uh->len);       /*数据报长度*/
saddr = ip_hdr(skb)->saddr;         /*发送数据包的主机（源）IP 地址*/
daddr = ip_hdr(skb)->daddr;       /*接收数据包主机（目的）IP 地址*/

if (ulen > skb->len)
        goto short_packet;

if (proto == IPPROTO_UDP) {
        if (ulen < sizeof(*uh) || pskb_trim_rcsum(skb, ulen))
                goto short_packet;
        uh = udp_hdr(skb);          /*UDP 报头*/
}

if (udp4_csum_init(skb, uh, proto))       /*校验和初始化*/
        goto csum_error;

sk = skb_steal_sock(skb);       /*返回 sk_buff 实例绑定的 sock 实例，/include/net/sock.h*/
if (sk) {       /*如果 sk_buff 已绑定了 sock 实例，这里为 NULL（未绑定）*/
        struct dst_entry *dst = skb_dst(skb);
        int ret;

        if (unlikely(sk->sk_rx_dst != dst))
                udp_sk_rx_dst_set(sk, dst);

        ret = udp_queue_rcv_skb(sk, skb);
        sock_put(sk);
        if (ret > 0)
                return -ret;
        return 0;
}

if (rt->rt_flags & (RTCF_BROADCAST|RTCF_MULTICAST))     /*组播或广播*/
        return __udp4_lib_mcast_deliver(net, skb, uh,saddr, daddr, udptable, proto);

sk = __udp4_lib_lookup_skb(skb, uh->source, uh->dest, udptable);       /*/net/ipv4/udp.c*/
                                /*调用__udp4_lib_lookup()函数在散列表中查找匹配的 udp_sock 实例*/
if (sk) {       /*查找到了 udp_sock 实例*/
        int ret;

        if (inet_get_convert_csum(sk) && uh->check && !IS_UDPLITE(sk))
                skb_checksum_try_convert(skb, IPPROTO_UDP, uh->check,inet_compute_pseudo);

        ret = udp_queue_rcv_skb(sk, skb);
```

```
                        /*将 sk_buff 实例添加到套接字接收缓存队列等，/net/ipv4/udp.c*/
        sock_put(sk);
        if (ret > 0)
            return -ret;
        return 0;        /*函数返回 0*/
    }

    /*处理没有找到 udp_sock 实例的情况*/
    if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
        goto drop;
    nf_reset(skb);

    /*校验和错误，丢弃数据包*/
    if (udp_lib_checksum_complete(skb))
        goto csum_error;

    UDP_INC_STATS_BH(net, UDP_MIB_NOPORTS, proto == IPPROTO_UDPLITE);
    icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0); /*发送目的不可达消息*/

    kfree_skb(skb);
    return 0;
    ... /*错误处理，丢弃数据包*/
}
```

__udp4_lib_rcv()函数的主要工作是调用__udp4_lib_lookup_skb()函数，根据报头中的 IP 地址和端口号，在散列表中查找匹配的 udp_sock 实例，调用 udp_queue_rcv_skb(sk, skb)函数将数据包添加到目的 udp_sock 实例的接收缓存队列中，等待用户进程读取。

需要注意的是，如果套接字执行了 connect()系统调用，在查找 udp_sock 实例时，还需要检查发送端的 IP 地址和端口号是否与 udp_sock 中保存的目的套接字 IP 地址和端口号相同，套接字只接收连接指定发送方套接字发送的数据包。如果没有查找到匹配的 udp_sock 实例，则丢弃数据包，并向发送方发送目的不可达 ICMP 消息。

2 用户进程读取数据

用户进程读取 UDP 套接字数据的系统调用将调用 proto_ops 实例 inet_dgram_ops 中的 **inet_recvmsg()** 函数读取接收缓存队列中数据包中的数据，函数定义如下（/net/ipv4/af_inet.c）：

```
int inet_recvmsg(struct socket *sock, struct msghdr *msg, size_t size,int flags)
/*msg：保存读取的数据以及发送方套接字地址*/
{
    struct sock *sk = sock->sk;
    int addr_len = 0;
    int err;
    sock_rps_record_flow(sk);
    err = sk->sk_prot->recvmsg(sk, msg, size, flags & MSG_DONTWAIT,
                                        flags & ~MSG_DONTWAIT, &addr_len);
    if (err >= 0)
        msg->msg_namelen = addr_len;
```

```
    return err;
}
```

inet_recvmsg()函数调用套接字关联 proto 实例中的 recvmsg()函数，UDP 套接字即 **udp_recvmsg()**函数，代码如下：

```
int udp_recvmsg(struct sock *sk, struct msghdr *msg, size_t len, int noblock,int flags, int *addr_len)
{
    struct inet_sock *inet = inet_sk(sk);
    DECLARE_SOCKADDR(struct sockaddr_in *, sin, msg->msg_name);
    struct sk_buff *skb;
    unsigned int ulen, copied;
    int peeked, off = 0;
    int err;
    int is_udplite = IS_UDPLITE(sk);
    bool slow;

    if (flags & MSG_ERRQUEUE)
        return ip_recv_error(sk, msg, len, addr_len);

try_again:
    skb = __skb_recv_datagram(sk, flags | (noblock ? MSG_DONTWAIT : 0),&peeked, &off, &err);
                                /*从接收缓存队列获取一个数据报，可能进入睡眠，/net/core/datagram.c*/
    ...   /*错误处理*/

    ulen = skb->len - sizeof(struct udphdr);   /*用户数据长度*/
    copied = len;
    if (copied > ulen)       /*读取数据长度不能超过数据报中用户数据长度*/
        copied = ulen;
    else if (copied < ulen)
        msg->msg_flags |= MSG_TRUNC;       /*数据报中用户数据被截短*/

    if (copied < ulen || UDP_SKB_CB(skb)->partial_cov) {
        if (udp_lib_checksum_complete(skb))        /*检查校验和*/
            goto csum_copy_err;
    }

    if (skb_csum_unnecessary(skb))        /*复制数据，不带校验和，/include/linux/skbuff.h*/
        err = skb_copy_datagram_msg(skb, sizeof(struct udphdr),msg, copied);
    else {                /*复制数据，带校验和，/net/core/datagram.c*/
        err = skb_copy_and_csum_datagram_msg(skb, sizeof(struct udphdr),msg);

        ...     /*错误处理*/
    }
    ...     /*错误处理*/

    if (!peeked)
```

```
            UDP_INC_STATS_USER(sock_net(sk),UDP_MIB_INDATAGRAMS, is_udplite);

        sock_recv_ts_and_drops(msg, sk, skb);

        /*获取发送方 IP 地址、端口号，保存到 msghdr 参数指定的地址结构中*/
        if (sin) {
            sin->sin_family = AF_INET;
            sin->sin_port = udp_hdr(skb)->source;        /*源端口号*/
            sin->sin_addr.s_addr = ip_hdr(skb)->saddr;    /*源 IP 地址*/
            memset(sin->sin_zero, 0, sizeof(sin->sin_zero));
            *addr_len = sizeof(*sin);       /*地址长度*/
        }
        if (inet->cmsg_flags)
            ip_cmsg_recv_offset(msg, skb, sizeof(struct udphdr));

        err = copied;        /*真实复制的字节数*/
        if (flags & MSG_TRUNC)
            err = ulen;

    out_free:
        skb_free_datagram_locked(sk, skb);    /*读取完之后，释放数据报*/
    out:
        return err;      /*返回真实读取的字节数*/
        ...    /*错误处理，校验和错误等*/
    }
```

udp_recvmsg()函数调用__skb_recv_datagram()函数从套接字接收缓存队列中获取一个数据报（sk_buff 实例），如果缓存队列中没有数据报，读进程可能进入睡眠（阻塞操作），直到接收到一个数据包，然后调用 skb_copy_datagram_msg()或 skb_copy_and_csum_datagram_msg()函数复制数据报中用户数据至读用户内存空间，最后返回真实读取的数据长度（字节数）。

需要注意的是：复制的数据长度不会超过参数 len 指定的字节数，如果数据报中用户数据长度超过此值，数据报中用户数据将会被截短；参数 msg 指定的 msghdr 结构体中（地址结构）将保存发送端套接字的源 IP 地址和端口号，用于识别数据来自哪个发送方；数据报中数据被读取后，数据报将会被释放。

12.7 ICMP 实现

Internet 控制消息协议（Internet Control Message Protocol，ICMP）用于提供与 IP 协议层配置和 IP 数据包处置相关的诊断和控制信息。ICMP 通常被认为是 IP 层的一个部分，它需要在所有 IP 实现中存在。它使用 IP 协议进行传输。因此，确切地说，它既不是一网络层协议，也不是一个传输层协议，而是位于两者之间。ICMP 实现比较简单，与 UDP 类似，因此在讲解完 UDP 之后，介绍 ICMP 的实现。

ICMP 负责传递可能需要注意的差错和控制报文。ICMP 报文通常是由 IP 层本身、上层的传输协议（如 TCP 或 UDP），甚至某些情况下是用户进程触发发送的。ICMP 并不为 IP 网络提供可靠性，相反，它表明了某些类别的故障和配置信息。ICMP 有 ICMPv4 和 ICMPv6 版本，分别用于 IPv4 和 IPv6。

ICMP 报文可分为两大类：有关 IP 数据报传递的 ICMP 报文（称为差错报文），以及有关信息采集和配置的 ICMP 报文（称为查询或信息类报文）。对于 ICMPv4，信息类报文包括回显请求和回显应答，以及路由器通告和路由请求。最常见的差错报文类型包括目的不可达、重定向、超时和参数问题。

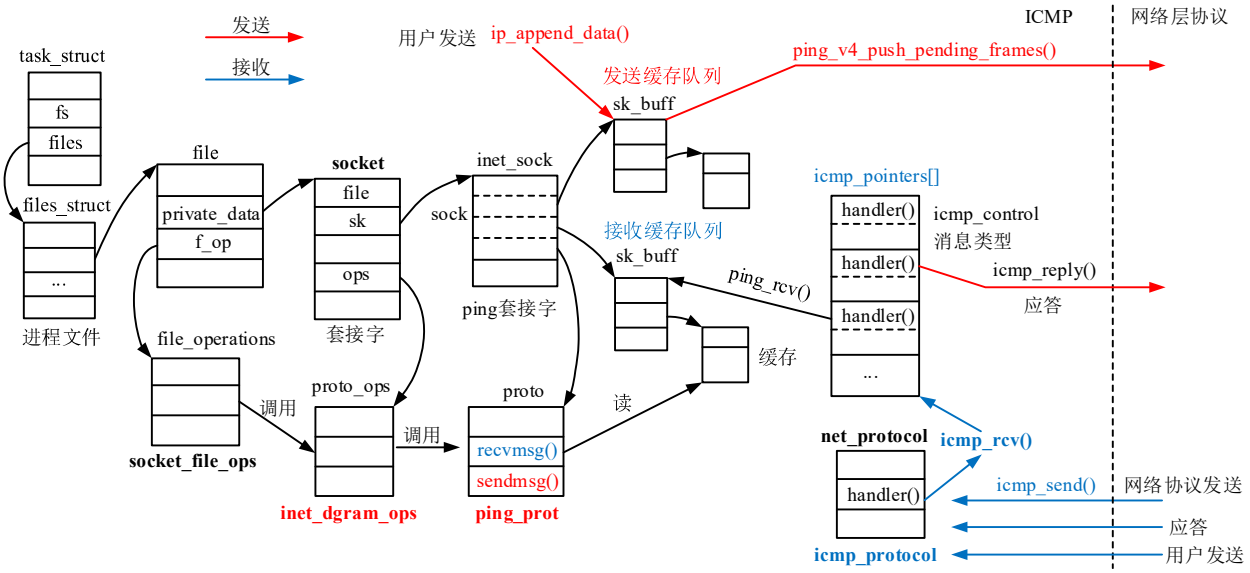
ICMP 数据报通常由网络协议产生和处理，部分类型报文也可以由用户进程通过 ping 套接字产生和接

收。ICMP 实现代码位于/net/ipv4/icmp.c 文件内，用户 ping 套接字相关代码位于/net/ipv4/ping.c 文件内。

12.7.1 概述

在 Linux 内核中 ICMP 协议类型由 IPPROTO_ICMP（1）标识。ICMP 消息通常由网络协议产生和处理，但是用户进程也可以通过 ping 套接字发送和接收部分 ICMP 消息。

ICMP 与 ping 套接字框架如下图所示：



ICMP 定义了 **icmp_control** 结构体数组 icmp_pointers[]，每个结构体代表一个 ICMP 消息类型，其中的 handler()函数用于处理所代表类型的消息。ICMP 中定义了各类型消息的处理函数 handler()。

ICMP 定义并注册了 net_protocol 实例 **icmp_protocol**，接收数据报函数为 **icmp_rcv()**，此函数将根据消息类型，查找到对应的 icmp_control 实例，调用其中的 handler()函数处理消息。对于需要发送应答（回显）的消息将调用 **icmp_reply()**函数发送应答消息。

网络层协议中通过接口函数 **icmp_send()**发送 ICMP 消息。

用户进程可通过 ping 套接字发送和接收部分 ICMP 消息，套接字关联 proto_ops 实例为 inet_dgram_ops，同 UDP 套接字，套接字关联的 proto 实例为 **ping_port**。套接字在传输层协议中由通用的 inet_sock 结构体表示。

对于需要传递给 ping 套接字的消息，icmp_control 实例中的处理函数将消息发送到用户 ping 套接字接收缓存队列。用户进程可读取 ping 套接字接收缓存队列数据报中的数据。

用户 ping 套接字发送 ICMP 消息的流程（ping_prot->sendmsg）与 UDP 套接字发送数据报类似。网络协议中发送 ICMP 消息的接口函数 **icmp_send()**操作流程与用户 ping 套接字发送消息操作类似。

12.7.2 协议实现

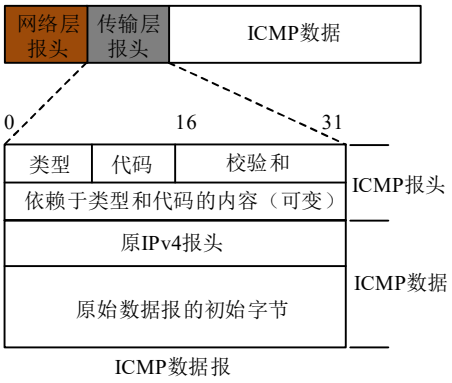
本小节介绍 ICMPv4 在 Linux 内核中的实现。

1 数据结构

ICMP 实现中主要的数据结构有 icmphdr、icmp_control 和 icmp_bxm 等。icmphdr 表示 ICMP 消息报头，icmp_control 用于定义 ICMP 消息的处理函数，icmp_bxm 用于构建 ICMP 消息时暂存消息数据。后两个数据结构都定义在/net/ipv4/icmp.c 文件内。

■icmphdr

ICMP 数据报结构如下图所示：



ICMP 报头由 icmphdr 结构体表示，定义如下（/include/uapi/linux/icmp.h）：

```
struct icmphdr {
  __u8          type;       /*消息类型*/
  __u8          code;       /*代码*/
  __sum16       checksum;   /*涵盖整个 ICMP 报文段的校验和*/
  union {         /*第 2 个 32 位字，依消息类型而不同*/
    struct {
        __be16    id;       /*回显应答，目的套接字端口号*/
        __be16    sequence;
    } echo;       /*回显消息*/
    __be32    gateway;      /*网关地址*/
    struct {
        __be16    __unused;
        __be16    mtu;
    } frag;       /*分段*/
  } un;
};
```

ICMP 数据报中的数据来源于原始 sk_buff 实例，即传输过程中触发发送 ICMP 消息的数据包。

■icmp_control

icmp_control 结构体定义如下（/net/ipv4/icmp.c）：

```
struct icmp_control {
    bool (*handler)(struct sk_buff *skb);      /*消息处理函数*/
    short     error;      /*ICMP 消息错误*/
};
```

内核定义了 icmp_control 结构体数线 icmp_pointers[NR_ICMP_TYPES+1]。NR_ICMP_TYPES 宏表示 ICMP 消息类型数量，消息类型定义在/include/uapi/linux/icmp.h 头文件内：

```
#define   ICMP_ECHOREPLY          0    /*回显应答*/
#define   ICMP_DEST_UNREACH       3    /*目的不可达*/
#define   ICMP_SOURCE_QUENCH      4    /*源端抑制，表示拥塞（弃用）*/
```

```
#define   ICMP_REDIRECT          5    /*重定向*/
#define   ICMP_ECHO              8    /*回显请求*/
#define   ICMP_TIME_EXCEEDED     11   /*超时*/
#define   ICMP_PARAMETERPROB     12   /*参数问题*/
#define   ICMP_TIMESTAMP         13   /*时间戳请求（弃用）*/
#define   ICMP_TIMESTAMPREPLY    14   /*时间戳应答（弃用）*/
#define   ICMP_INFO_REQUEST      15   /*信息请求（弃用）*/
#define   ICMP_INFO_REPLY        16   /*信息应答（弃用）*/
#define   ICMP_ADDRESS           17   /*地址掩码请求（弃用）*/
#define   ICMP_ADDRESSREPLY      18   /*地址掩码应答（弃用）*/
#define   NR_ICMP_TYPES          18    /*消息类型最大数量*/
```

消息类型代码也定义在/include/uapi/linux/icmp.h 头文件内，请读者自行阅读。

内核在/net/ipv4/icmp.c 文件内，对 icmp_pointers[]数组进行了初始化，设置各类型消息的处理函数，如下所示：

```
static const struct icmp_control icmp_pointers[NR_ICMP_TYPES + 1] = {
    [ICMP_ECHOREPLY] = {      /*回显应答*/
        .handler = ping_rcv,      /*将消息添加到 ping 套接字接收缓存队列*/
    },
    ...
    [ICMP_DEST_UNREACH] = {    /*目的不可达，最常用的 ICMP 消息*/
        .handler = icmp_unreach,
        .error = 1,
    },
    [ICMP_SOURCE_QUENCH] = {
        .handler = icmp_unreach,
        .error = 1,
    },
    [ICMP_REDIRECT] = {       /*路由重定向*/
        .handler = icmp_redirect,
        .error = 1,
    },
    ...
    [ICMP_ECHO] = {       /*回显应答*/
        .handler = icmp_echo,
    },
    ...
    [ICMP_TIME_EXCEEDED] = {
        .handler = icmp_unreach,
        .error = 1,
    },
    [ICMP_PARAMETERPROB] = {
        .handler = icmp_unreach,
        .error = 1,
    },
```

```
    ...        /*省略的消息类型都是已弃用的*/
};
```

■icmp_bxm

icmp_bxm 结构体用于构建 ICMP 消息数据报，在发送操作中使用，定义如下（/net/ipv4/icmp.c）：

```
struct icmp_bxm {
    struct sk_buff *skb;       /*指向承载消息的 sk_buff 实例*/
    int offset;
    int data_len;       /*数据长度*/

    struct {
        struct icmphdr    icmph;       /*ICMP 报头*/
        __be32            times[3];
    } data;
    int head_len;       /*ICMP 报头长度*/
    struct ip_options_data replyopts;     /*IP 选项*/
};
```

3 初始化

内核在 IPv4 协议簇初始化函数 inet_init()中调用了 ICMP 初始化函数 icmp_init()，函数定义如下：

```
int __init icmp_init(void)       /*/net/ipv4/icmp.c*/
{
    return register_pernet_subsys(&icmp_sk_ops);     /*注册 pernet_operations 结构体实例*/
}
```

icmp_init()函数内注册 pernet_operations 结构体实例 **icmp_sk_ops**，其初始化函数为 **icmp_sk_init()**，定义如下：

```
static int __net_init icmp_sk_init(struct net *net)
{
    int i, err;

    net->ipv4.icmp_sk = alloc_percpu(struct sock *); /*指向 sock 实例的指针（数组），percpu 变量*/
    if (!net->ipv4.icmp_sk)
        return -ENOMEM;

    for_each_possible_cpu(i) {       /*对每个 CPU 核创建内核 ICMP 套接字（原始套接字）*/
        struct sock *sk;

        err = inet_ctl_sock_create(&sk, PF_INET,SOCK_RAW, IPPROTO_ICMP, net);
                                    /*创建内核 ICMP 套接字（原始套接字），/net/socket.c*/
        if (err < 0)
            goto fail;

        *per_cpu_ptr(net->ipv4.icmp_sk, i) = sk;
```

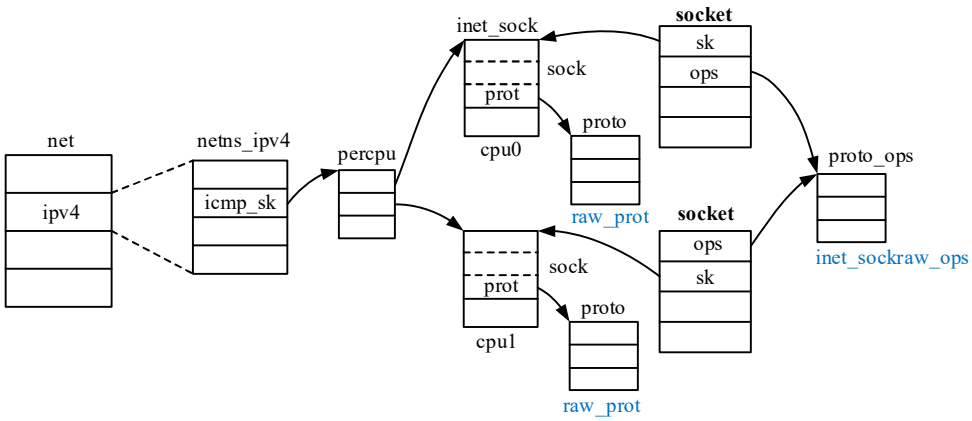
```
        sk->sk_sndbuf =    2 * SKB_TRUESIZE(64 * 1024);    /*设置发送缓存区大小*/
        sock_set_flag(sk, SOCK_USE_WRITE_QUEUE);
        inet_sk(sk)->pmtudisc = IP_PMTUDISC_DONT;
    }

    /*回显应答控制参数*/
    net->ipv4.sysctl_icmp_echo_ignore_all = 0;
    net->ipv4.sysctl_icmp_echo_ignore_broadcasts = 1;
    net->ipv4.sysctl_icmp_ignore_bogus_error_responses = 1;

    /*配置全局的速率控制参数*/
    net->ipv4.sysctl_icmp_ratelimit = 1 * HZ;
    net->ipv4.sysctl_icmp_ratemask = 0x1818;
    net->ipv4.sysctl_icmp_errors_use_inbound_ifaddr = 0;

    return 0;
    ...    /*错误处理*/
}
```

icmp_sk_init()函数主要工作是为每个 CPU 核创建原始 ICMP 内核套接字，由 net.ipv4.icmp（指针，percpu 变量）管理，设置 ICMP 控制参数。

inet_ctl_sock_create()函数用于创建内核套接字，由前面介绍的创建 IPv4 套接字函数可知，内核 ICMP 套接字匹配的 inet_protosw 实例如下所示：

```
static struct inet_protosw inetsw_array[] =
{
    ...
    {
        .type =       SOCK_RAW,    /*原始套接字，无传输层协议，直接与网络层交互*/
        .protocol =   IPPROTO_IP,  /*匹配所有的协议类型*/
        .prot =       &raw_prot,       /*/net/ipv4/raw.c*/
        .ops =        &inet_sockraw_ops,    /*/net/ipv4/af_inet.c*/
        .flags =      INET_PROTOSW_REUSE,
    }
};
```

初始化函数中创建的内核 ICMP 套接字如下图所示，原始套接字实现代码在/net/ipv4/raw.c 文件内，请读者自行阅读。



4 接收消息

内核在/net/ipv4/af_inet.c 文件内定义了 ICMP 对应的 net_protocol 实例，如下所示：

```
static const struct net_protocol icmp_protocol = {
    .handler =      icmp_rcv,     /*接收 ICMP 消息函数*/
    .err_handler = icmp_err,
    .no_policy =   1,
    .netns_ok =    1,
};
```

内核在 IPv4 协议簇初始化函数 inet_init()中，将注册 icmp_protocol 实例。接收 ICMP 消息的函数定义如下（/net/ipv4/icmp.c）：

```
int icmp_rcv(struct sk_buff *skb)
{
    struct icmphdr *icmph;
    struct rtable *rt = skb_rtable(skb);
    struct net *net = dev_net(rt->dst.dev);
    bool success;

    if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
        struct sec_path *sp = skb_sec_path(skb);
        int nh;

        if (!(sp && sp->xvec[sp->len - 1]->props.flags &XFRM_STATE_ICMP))
            goto drop;

        if (!pskb_may_pull(skb, sizeof(*icmph) + sizeof(struct iphdr)))
            goto drop;

        nh = skb_network_offset(skb);
        skb_set_network_header(skb, sizeof(*icmph));

        if (!xfrm4_policy_check_reverse(NULL, XFRM_POLICY_IN, skb))
            goto drop;

        skb_set_network_header(skb, nh);
    }

    ICMP_INC_STATS_BH(net, ICMP_MIB_INMSGS);

    if (skb_checksum_simple_validate(skb))
        goto csum_error;

    if (!pskb_pull(skb, sizeof(*icmph)))
        goto error;

    icmph = icmp_hdr(skb);      /*ICMP 报头指针*/
```

```
        ICMPMSGIN_INC_STATS_BH(net, icmph->type);
        if (icmph->type > NR_ICMP_TYPES)
            goto error;

        if (rt->rt_flags & (RTCF_BROADCAST | RTCF_MULTICAST)) {       /*组播或广播消息*/
            if ((icmph->type == ICMP_ECHO ||icmph->type == ICMP_TIMESTAMP) &&
                net->ipv4.sysctl_icmp_echo_ignore_broadcasts) {
                goto error;
            }
            if (icmph->type != ICMP_ECHO &&
                icmph->type != ICMP_TIMESTAMP &&
                icmph->type != ICMP_ADDRESS &&
                icmph->type != ICMP_ADDRESSREPLY) {
                goto error;
            }
        }

        success = icmp_pointers[icmph->type].handler(skb);       /*调用类型消息对应的处理函数*/

        if (success)  {     /*处理成功，释放 sk_buff 实例，函数返回 0*/
            consume_skb(skb);
            return 0;
        }
        ...     /*错误处理*/
    }
```

icmp_rcv()函数的主要工作是调用相应类型消息的处理函数，处理消息。各类型消息的处理函数请读者自行阅读。

5 发送消息

网络节点在接收数据包出错时，将向发送主机发送 ICMP 消息。网络协议中通常调用 **icmp_send()**接口函数向上层传递 ICMP 消息，函数定义如下（/net/ipv4/icmp.c）：

```
    void icmp_send(struct sk_buff *skb_in, int type, int code, __be32 info)
    /*skb_in：传递的 sk_buff 实例，type：消息类型，code：消息代码，info：信息，大部分时机为 0*/
    {
        struct iphdr *iph;
        int room;
        struct icmp_bxm *icmp_param;       /*指向 icmp_bxm 实例*/
        struct rtable *rt = skb_rtable(skb_in);     /*路由查找结果*/
        struct ipcm_cookie ipc;
        struct flowi4 fl4;
        __be32 saddr;       /*源 IP 地址*/
        u8    tos;        /*服务类型*/
        u32 mark;
        struct net *net;
```

```
struct sock *sk;

if (!rt)
    goto out;
net = dev_net(rt->dst.dev);        /*网络命名空间*/

iph = ip_hdr(skb_in);      /*IP 报头指针*/

...    /*IP 报头有效性检查*/

if (skb_in->pkt_type != PACKET_HOST)    /*不是传递给本机的数据包，不需要发送 ICMP 消息*/
    goto out;

if (rt->rt_flags & (RTCF_BROADCAST | RTCF_MULTICAST))
    goto out;            /*组播或广播数据包，不发送 ICMP 消息*/


/*只对第一个分段发送 ICMP 消息*/
if (iph->frag_off & htons(IP_OFFSET))
    goto out;

if (icmp_pointers[type].error) {     /*icmp_control.error*/
    if (iph->protocol == IPPROTO_ICMP) {        /*如果传递的是 ICMP 消息*/
        u8 _inner_type, *itp;

        itp = skb_header_pointer(skb_in,skb_network_header(skb_in) +(iph->ihl << 2) +
                    offsetof(struct icmphdr,type) -skb_in->data,
                    sizeof(_inner_type),
                    &_inner_type);
        if (!itp)
            goto out;
        if (*itp > NR_ICMP_TYPES ||icmp_pointers[*itp].error)
            goto out;
    }
}

icmp_param = kmalloc(sizeof(*icmp_param), GFP_ATOMIC);     /*分配 icmp_param 实例*/
if (!icmp_param)
    return;

sk = icmp_xmit_lock(net);
if (!sk)
    goto out_free;

/*构建源 IP 地址和 IP 选项*/
saddr = iph->daddr;      /*sk_buff 中目的 IP 地址为 ICMP 消息源地址*/
```

```
if (!(rt->rt_flags & RTCF_LOCAL)) {     /*如果路由查找结果指示不是传递给本机的数据包*/
        struct net_device *dev = NULL;

        rcu_read_lock();
        if (rt_is_input_route(rt) &&net->ipv4.sysctl_icmp_errors_use_inbound_ifaddr)
                dev = dev_get_by_index_rcu(net, inet_iif(skb_in));

        if (dev)
                saddr = inet_select_addr(dev, 0, RT_SCOPE_LINK);
        else
                saddr = 0;
        rcu_read_unlock();
}

tos = icmp_pointers[type].error ? ((iph->tos & IPTOS_TOS_MASK) |
                                   IPTOS_PREC_INTERNETCONTROL) :
                                   iph->tos;
mark = IP4_REPLY_MARK(net, skb_in->mark);

if (ip_options_echo(&icmp_param->replyopts.opt.opt, skb_in))
        goto out_unlock;

/*准备 ICMP 报头数据，写入 icmp_param->data 成员*/
icmp_param->data.icmph.type        = type;       /*消息类型*/
icmp_param->data.icmph.code        = code;       /*代码*/
icmp_param->data.icmph.un.gateway = info;     /*网关地址*/
icmp_param->data.icmph.checksum       = 0;
icmp_param->skb      = skb_in;
icmp_param->offset = skb_network_offset(skb_in);     /*网络层报头偏移量*/
inet_sk(sk)->tos = tos;
sk->sk_mark = mark;
ipc.addr = iph->saddr;
ipc.opt = &icmp_param->replyopts.opt;
ipc.tx_flags = 0;
ipc.ttl = 0;
ipc.tos = -1;

rt = icmp_route_lookup(net, &fl4, skb_in, iph, saddr, tos, mark,type, code, icmp_param);
                                                              /*路由选择查找*/
...     /*错误处理*/

if (!icmpv4_xrlim_allow(net, rt, &fl4, type, code))
        goto ende;

room = dst_mtu(&rt->dst);
if (room > 576)
```

```
        room = 576;         /*ICMP 消息不超过 576 字节*/
    room -= sizeof(struct iphdr) + icmp_param->replyopts.opt.opt.optlen;
    room -= sizeof(struct icmphdr);

    icmp_param->data_len = skb_in->len - icmp_param->offset;     /*数据长度*/
    if (icmp_param->data_len > room)
        icmp_param->data_len = room;
    icmp_param->head_len = sizeof(struct icmphdr);       /*ICMP 报头长度*/

    icmp_push_reply(icmp_param, &fl4, &ipc, &rt);     /*构建并发送 ICMP 消息*/
ende:
    ip_rt_put(rt);
out_unlock:
    icmp_xmit_unlock(sk);
out_free:
    kfree(icmp_param);
out::;
}
```

icmp_send()函数内分配 icmp_bxm 实例，并填充实例，主要是 sk_buff 实例指针成员和 ICMP 报头数据等。最后调用 icmp_push_reply()函数构建 ICMP 消息 sk_buff 实例，并发送。

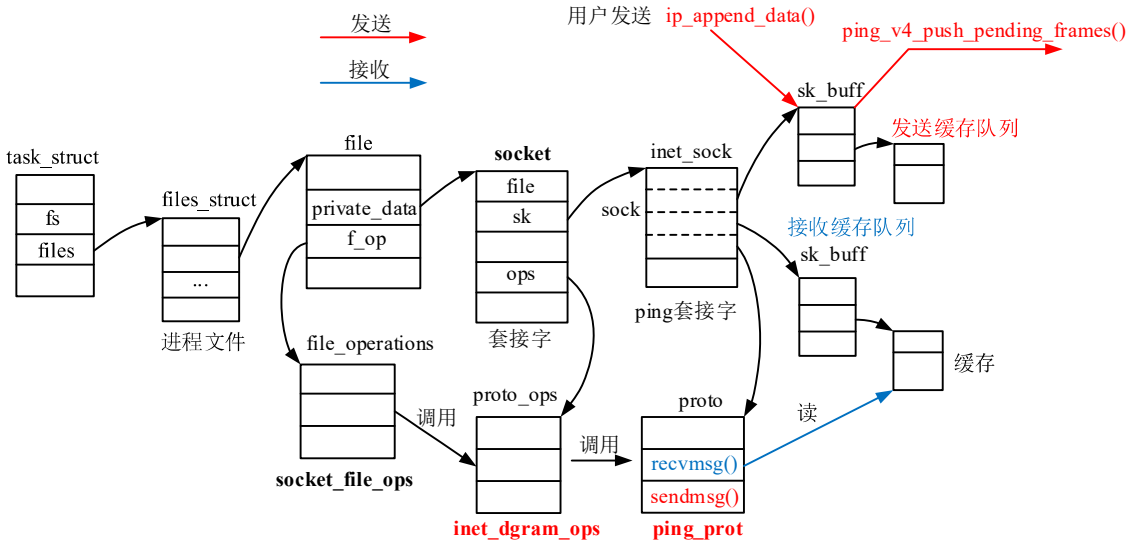
icmp_push_reply()函数调用 UDP 实现中介绍过的 ip_append_data()和 ip_flush_pending_frames()函数构建数据报并发送。ip_append_data()函数内将 icmp_bxm 实例中的 ICMP 报头和原始 sk_buff 实例中的部分数据复制到 ICMP 消息 sk_buff 实例中，函数源代码请读者自行阅读。

12.7.3 ping 套接字

用户进程可通过 ping 套接字与 ICMP 通信，本小节介绍 ping 套接字的实现。

1 套接字概述

ping 套接字框架如下图所示：



创建 ping 套接字时，关联的 inet_protosw 实例如下所示：

```
static struct inet_protosw inetsw_array[] =
```

```
{
    ...
    {
        .type =          SOCK_DGRAM,   /*数据报套接字*/
        .protocol =    IPPROTO_ICMP,   /*ICMP，ping 套接字*/
        .prot =          &ping_prot,      /*proto 实例，/net/ipv4/ping.c*/
        .ops =           &inet_dgram_ops,      /*proto_ops 实例，同 UDP 套接字*/
        .flags =         INET_PROTOSW_REUSE,
    },
    ...
}
```

ping 套接字关联的 proto_ops 实例为 **inet_dgram_ops**（同 UDP 套接字），proto 实例为 **ping_prot**，详见下文。

■套接字操作

ping 套接字关联 proto 实例为 ping_prot，定义如下（/net/ipv4/ping.c）：

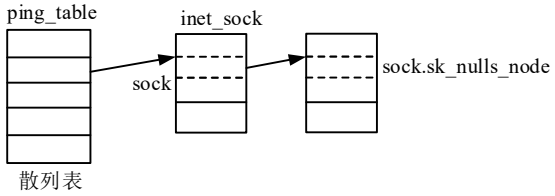
```
struct proto ping_prot = {
    .name =         "PING",
    .owner =  THIS_MODULE,
    .init =     ping_init_sock,     /*初始化函数，在创建套接字时调用*/
    .close =    ping_close,
    .connect =      ip4_datagram_connect,      /*连接函数，同 UDP 套接字*/
    .disconnect =  udp_disconnect,
    .setsockopt =  ip_setsockopt,        /*设置参数*/
    .getsockopt =  ip_getsockopt,        /*获取参数*/
    .sendmsg =     ping_v4_sendmsg,     /*发送消息函数*/
    .recvmsg =     ping_recvmsg,         /*接收消息函数*/
    .bind =        ping_bind,        /*绑定操作*/
    .backlog_rcv =      ping_queue_rcv_skb,
    .release_cb =  ip4_datagram_release_cb,
    .hash =        ping_hash,       /*空操作*/
    .unhash =ping_unhash,
    .get_port =     ping_get_port,     /*获取端口号，并将 inet_sock 实例插入全局散列表*/
    .obj_size =     sizeof(struct inet_sock),     /*套接字由 inet_sock 实例表示*/
};
```

■套接字管理

内核通过全局散列表管理表示 ping 套接字的 inet_sock 实例，全局散列表定义如下：

```
struct ping_table {
    struct hlist_nulls_head    hash[PING_HTABLE_SIZE];     /*PING_HTABLE_SIZE 为 64*/
    rwlock_t        lock;
};

static struct ping_table ping_table;
```

初始化函数 ping_init()（在 inet_init()中调用）用于初始化全局散列表 ping_table。全局散列表结构如下图所示：



套接字 inet_sock 实例通过网络命名空间和端口号计算散列值，通过 inet_sock 实例插入到散列表。

2 绑定操作

ping 套接字 bind()系统调用内调用 inet_dgram_ops 实例中的 inet_bind()函数完成绑定操作。inet_bind()函数内将调用 ping_prot 实例中的 bind()函数完成绑定操作，即 ping_bind()函数，定义如下。

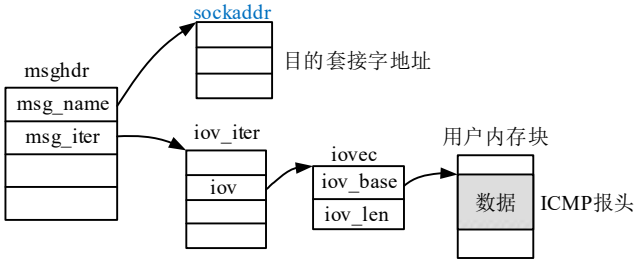
```
int ping_bind(struct sock *sk, struct sockaddr *uaddr, int addr_len)
{
    struct inet_sock *isk = inet_sk(sk);
    unsigned short snum;     /*端口号*/
    int err;
    int dif = sk->sk_bound_dev_if;

    err = ping_check_bind_addr(sk, isk, uaddr, addr_len);
    ...
    lock_sock(sk);

    err = -EINVAL;
    if (isk->inet_num != 0)     /*原端口号须为 0*/
        goto out;

    err = -EADDRINUSE;
    ping_set_saddr(sk, uaddr);      /*设置套接字源 IP 地址，/net/ipv4/ping.c*/
    snum = ntohs(((struct sockaddr_in *)uaddr)->sin_port);      /*源端口号*/
    if (ping_get_port(sk, snum) != 0) { /*将 inet_sock 实例插入散列表，snum 为 0，则先分配端口号*/
        ...
    }
    ...
    err = 0;
    if (sk->sk_family == AF_INET && isk->inet_rcv_saddr)
        sk->sk_userlocks |= SOCK_BINDADDR_LOCK;
#if IS_ENABLED(CONFIG_IPV6)
    ...
#endif

    if (snum)
        sk->sk_userlocks |= SOCK_BINDPORT_LOCK;
    isk->inet_sport = htons(isk->inet_num);     /*端口号*/
    isk->inet_daddr = 0;        /*目的 IP 地址、端口号，设为 0*/
```

```
        isk->inet_dport = 0;

    #if IS_ENABLED(CONFIG_IPV6)
        ...
    #endif

        sk_dst_reset(sk);
out:
        release_sock(sk);
        pr_debug("ping_v4_bind -> %d\n", err);
        return err;
}
```

ping_bind()函数中调用 **ping_get_port()**函数，此函数由网络命名空间和端口号 snum 计算散列值，将 inet_sock 实例插入到全局散列表 ping_table，并将端口号赋予 inet_sock->inet_num 成员。如果参数传递的端口号为 0，在 ping_get_port()函数将为套接字分配端口号，然后将 inet_sock 实例插入到 ping_table 全局散列表，将分配的端口号赋予 inet_sock->inet_num 成员。

3 发送消息

用户进程发送 ICMP 消息时，将 ICMP 报头写入用户内存，然后通过系统调用发送 ICMP 消息。



发送 ICMP 消息的系统调用最终调用 ping_prot 实例中的 sendmsg()函数发送 ICMP 消息，此函数为 ping_v4_sendmsg()，定义如下：

```
static int ping_v4_sendmsg(struct sock *sk, struct msghdr *msg, size_t len)
{
    struct net *net = sock_net(sk);
    struct flowi4 fl4;
    struct inet_sock *inet = inet_sk(sk);
    struct ipcm_cookie ipc;
    struct icmphdr user_icmph;     /*来自用户空间的 ICMP 报头*/
    struct pingfakehdr    pfh;        /*包含 ICMP 报头，msghdr 指针成员等，/include/net/ping.h*/
    struct rtable *rt = NULL;
    struct ip_options_data opt_copy;
    int free = 0;
    __be32 saddr, daddr, faddr;
    u8    tos;
    int err;
    ...
    err = ping_common_sendmsg(AF_INET, msg, len, &user_icmph,sizeof(user_icmph));
                                    /*将 msg 参数中用户 ICMP 报头复制到 user_icmph 实例*/
```

```
...
/*获取或验证地址*/
if (msg->msg_name) {       /*如果指定了目的地址*/
      DECLARE_SOCKADDR(struct sockaddr_in *, usin, msg->msg_name);
      if (msg->msg_namelen < sizeof(*usin))
            return -EINVAL;
      if (usin->sin_family != AF_INET)
            return -EAFNOSUPPORT;
      daddr = usin->sin_addr.s_addr;       /*目的 IP 地址*/
} else {
      if (sk->sk_state != TCP_ESTABLISHED)
            return -EDESTADDRREQ;
      daddr = inet->inet_daddr;         /*使用连接操作中的目的 IP 地址*/
}

ipc.addr = inet->inet_saddr;       /*源 IP 地址*/
ipc.opt = NULL;
ipc.oif = sk->sk_bound_dev_if;     /*输出网络设备编号*/
ipc.tx_flags = 0;
ipc.ttl = 0;
ipc.tos = -1;

sock_tx_timestamp(sk, &ipc.tx_flags);

if (msg->msg_controllen) {       /*如果是控制消息*/
      err = ip_cmsg_send(sock_net(sk), msg, &ipc, false);
      if (err)
            return err;
      if (ipc.opt)
            free = 1;
}
if (!ipc.opt) {         /*如果 ipc 不存在 IP 选项*/
      struct ip_options_rcu *inet_opt;

      rcu_read_lock();
      inet_opt = rcu_dereference(inet->inet_opt);
      if (inet_opt) {         /*复制 inet_sock 中 IP 选项到 ipc*/
            memcpy(&opt_copy, inet_opt,sizeof(*inet_opt) + inet_opt->opt.optlen);
            ipc.opt = &opt_copy.opt;
      }
      rcu_read_unlock();
}

saddr = ipc.addr;         /*源 IP 地址*/
ipc.addr = faddr = daddr;     /*目的 IP 地址*/
```

```
    if (ipc.opt && ipc.opt->opt.srr) {
        if (!daddr)
            return -EINVAL;
        faddr = ipc.opt->opt.faddr;
    }
    tos = get_rttos(&ipc, inet);       /*服务类型*/
    if (sock_flag(sk, SOCK_LOCALROUTE) ||
        (msg->msg_flags & MSG_DONTROUTE) ||
        (ipc.opt && ipc.opt->opt.is_strictroute)) {
        tos |= RTO_ONLINK;
    }

    if (ipv4_is_multicast(daddr)) {     /*目的地址为组播地址*/
        if (!ipc.oif)
            ipc.oif = inet->mc_index;
        if (!saddr)
            saddr = inet->mc_addr;
    } else if (!ipc.oif)
        ipc.oif = inet->uc_index;

    flowi4_init_output(&fl4, ipc.oif, sk->sk_mark, tos,
            RT_SCOPE_UNIVERSE, sk->sk_protocol,
            inet_sk_flowi_flags(sk), faddr, saddr, 0, 0);       /*初始化 fl4 实例*/

    security_sk_classify_flow(sk, flowi4_to_flowi(&fl4));
    rt = ip_route_output_flow(net, &fl4, sk);         /*路由选择查找*/
    ...     /*错误处理*/
back_from_confirm:

    if (!ipc.addr)
        ipc.addr = fl4.daddr;

    lock_sock(sk);

    pfh.icmph.type = user_icmph.type;   /*消息类型*/
    pfh.icmph.code = user_icmph.code;   /*代码*/
    pfh.icmph.checksum = 0;
    pfh.icmph.un.echo.id = inet->inet_sport;     /*端口号*/
    pfh.icmph.un.echo.sequence = user_icmph.un.echo.sequence;
    pfh.msg = msg;     /*指向 msghdr 实例*/
    pfh.wcheck = 0;
    pfh.family = AF_INET;     /*地址簇*/

    err = ip_append_data(sk, &fl4, ping_getfrag, &pfh, len,0, &ipc, &rt, msg->msg_flags);
                                /*创建 sk_buff 实例（队列），添加到发送缓存队列*/
    if (err)
```

```
        ip_flush_pending_frames(sk);
    else
        err = ping_v4_push_pending_frames(sk, &pfh, &fl4);    /*发送发送缓存队列中数据包*/
    release_sock(sk);

out:
    ip_rt_put(rt);
    if (free)
        kfree(ipc.opt);
    ...    /*错误处理*/
    return err;
    ...
}
```

ICMP 报头写入到 msghdr 结构体中指示的用户内存。ping_v4_sendmsg()函数通过 pingfakehdr 结构体暂存构建 ICMP 消息的信息（包括来自用户内存的 ICMP 报头），函数内调用 **ip_append_data()**函数构建发送数据包 sk_buff 实例（队列），并添加到套接字发送缓存队列，最后调用 **ping_v4_push_pending_frames()**函数发送数据包。

此处 **ip_append_data()**函数内调用 **ping_getfrag()**函数从 pingfakehdr 实例中复制数据至 ICMP 消息数据包 sk_buff 实例中（内存缓存区）。

4 读取消息

ICMP 在收到需要传递给用户套接字的消息时，调用 ping_rcv(skb)函数将数据包添加到目的 ping 套接字接收缓存队列。ping_rcv()函数通过网络命名空间和目的套接字端口号查找 inet_sock 实例，并将 sk_buff 实例添加到其接收缓存队列。

用户进程通过读操作系统调用可从 ping 套接字接收缓存队列中读取数据报中数据。读操作系统调用最终调用 ping_prot 实例中的 recvmsg()函数，即 **ping_recvmsg()**函数，将接收缓存队列中数据报数据复制到用户内存。

ping_recvmsg()函数定义如下：

```
int ping_recvmsg(struct sock *sk, struct msghdr *msg, size_t len, int noblock,int flags, int *addr_len)
{
    struct inet_sock *isk = inet_sk(sk);
    int family = sk->sk_family;
    struct sk_buff *skb;
    int copied, err;

    pr_debug("ping_recvmsg(sk=%p,sk->num=%u)\n", isk, isk->inet_num);

    err = -EOPNOTSUPP;
    if (flags & MSG_OOB)
        goto out;

    if (flags & MSG_ERRQUEUE)
        return    inet_recv_error(sk, msg, len, addr_len);

    skb = skb_recv_datagram(sk, flags, noblock, &err);    /*从接收缓存队列获取数据报，可能睡眠*/
```

```
    if (!skb)
        goto out;

    copied = skb->len;      /*复制数据长度，字节数*/
    if (copied > len) {
        msg->msg_flags |= MSG_TRUNC;
        copied = len;
    }

    err = skb_copy_datagram_msg(skb, 0, msg, copied);    /*复制数据报数据至用户内存*/
    if (err)
        goto done;

    sock_recv_timestamp(msg, sk, skb);

    /*获取发送端地址，写入 msg->msg_name*/
    if (family == AF_INET) {
        DECLARE_SOCKADDR(struct sockaddr_in *, sin, msg->msg_name);

        if (sin) {
            sin->sin_family = AF_INET;
            sin->sin_port = 0         /* skb->h.uh->source */;
            sin->sin_addr.s_addr = ip_hdr(skb)->saddr;      /*发送端 IP 地址*/
            memset(sin->sin_zero, 0, sizeof(sin->sin_zero));
            *addr_len = sizeof(*sin);
        }

        if (isk->cmsg_flags)
            ip_cmsg_recv(msg, skb);

        #if IS_ENABLED(CONFIG_IPV6)
           ...
        #endif
    } else {
        BUG();
    }
    err = copied;       /*复制数据长度，字节数*/
done:
    skb_free_datagram(sk, skb);       /*释放数据报*/
out:
    pr_debug("ping_recvmsg -> %d\n", err);
    return err;
}
```

ping_recvmsg()函数与 UDP 中的 udp_recvmsg()函数类似，主要工作是从套接字接收缓存队列中获取一个数据报，从中复制数据到用户内存，如果需要的话将发送端地址写入 msghdr 参数指定的地址结构中，最后释放数据报。

12.8 TCP 实现

TCP（传输控制协议）是最重要最复杂的传输层协议，它传递了网络中大部分的流量。TCP 提供与 UDP 一样的数据交付和差错检查功能，它还是一种面向连接的，提供可靠数据传输和拥塞控制的传输层协议。TCP 要在不可靠的 IP 之上建立起可靠的数据传输，其实现必然是复杂的。

12.8.1 概述

本小节先简要回顾 TCP 原理，然后概述 TCP 套接字实现。

1 TCP 原理

在前面介绍了 TCP 原理，主是连接管理和可靠数据传输原理，下面再做简要的回顾。

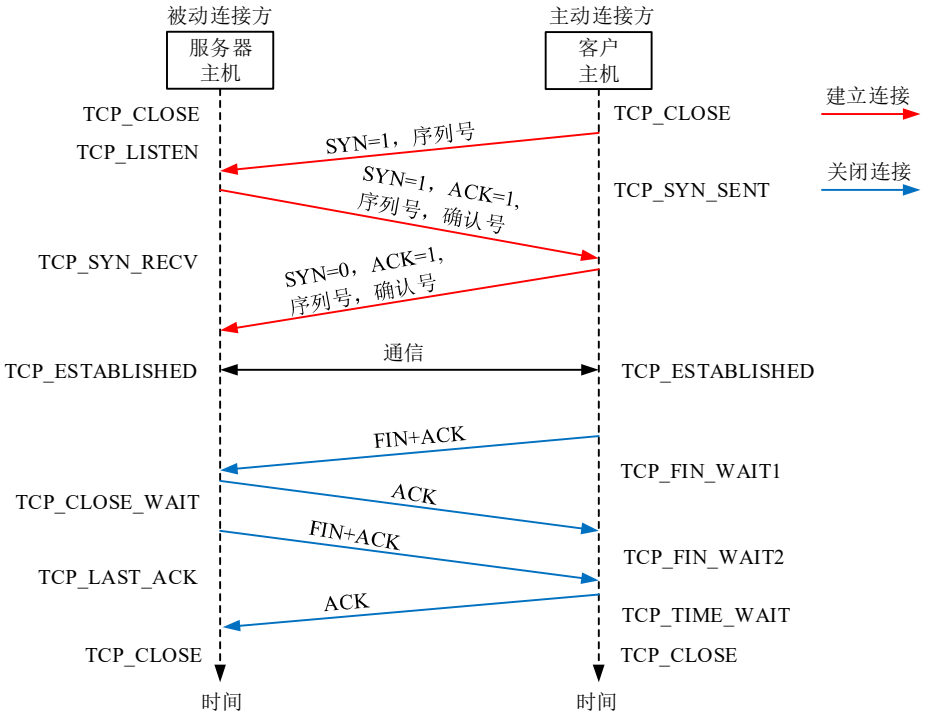
■TCP 连接管理

TCP 是面向连接的一对一的数据传输协议，通信双方在传输数据前需要建立连接，数据传输完成后需要关闭连接。TCP 通信双方根据连接的不同阶段，在各状态之间转换。在 Linux 内核中通信双方由套接字表示，因此 TCP 状态视为套接字的状态。

TCP 套接字状态值定义如下（/include/net/tcp_states.h）：

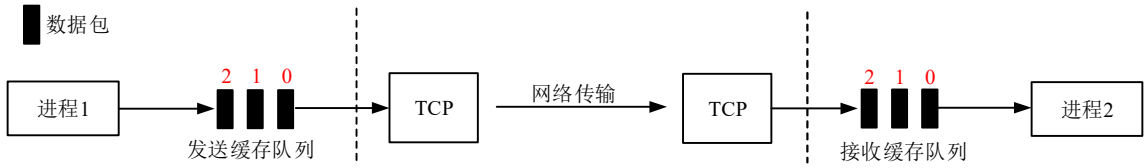
```
enum {
    TCP_ESTABLISHED = 1,    /*已连接状态*/
    TCP_SYN_SENT,
    TCP_SYN_RECV,
    TCP_FIN_WAIT1,
    TCP_FIN_WAIT2,
    TCP_TIME_WAIT,
    TCP_CLOSE,          /*关闭状态*/
    TCP_CLOSE_WAIT,
    TCP_LAST_ACK,
    TCP_LISTEN,         /*监听状态*/
    TCP_CLOSING,        /*正在关闭状态*/
    TCP_NEW_SYN_RECV,

    TCP_MAX_STATES    /*最大状态值*/
};
```

下图示意了连接的不同阶段服务器、客户套接字所处的状态，以及执行连接建立和关闭所进行操作：



■可靠传输原理

可靠数据传输是指传输协议保证发送方发送的数据能够正确、完整、按序地到达接收方，不会出现丢失、损坏和乱序到达。如下图所示，进程 1 向进程 2 发送 0、1、2 三个数据包，通信双方 TCP 保证这三个数据包能够正确、完整、按序地出现在接收方 2 的接收缓存队列中，进程 2 可从接收缓存队列中读取进程 1 发送的数据。



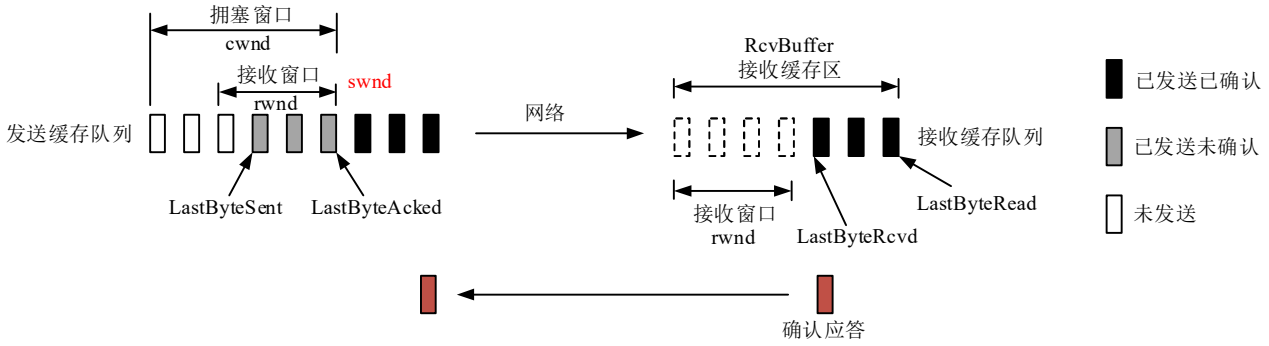
TCP 通过 IP 协议来传输数据包，而 IP 是不可靠的，因此 TCP 需要通过一些措施来达到可靠数据传输的目的。TCP 通过重传损坏、丢失的数据包来达到可靠数据传输的目的。TCP 通过接收方的确认应答，通知发送方数据包已正确送达，确认号、序列号用于区分哪些数据已经正确接收，哪些数据丢失。TCP 还设置了重传定时器，如果发出数据包后在定时器到期时还没有收到确认应答，则认为数据包丢失了，发送方需要重传数据包。

TCP 通过流量控制来防止接收方接收缓存溢出，以避免数据包丢失。通过拥塞控制来防止中间路由器缓存溢出，以避免数据包丢失。

接收方在确认应答中包含接收缓存区空闲区域大小的信息（接收窗口 rwnd），发送方发送到网络中，但尚未到达的数据量不能超过接收缓存空闲区域大小，这称为流量控制。

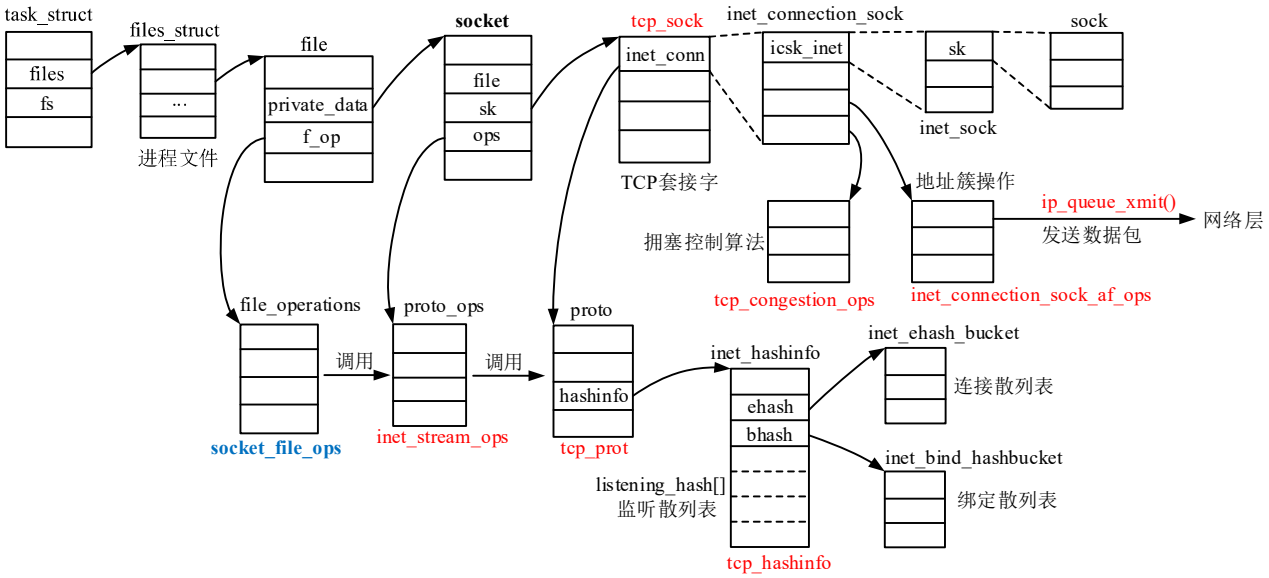
网络中间路由器缓存溢出也会导致数据包丢失，但 IP 层没有向端系统显式提供网络拥塞的信息。TCP 通过定时器超时和收到冗余应答来感知网络拥塞。与流量控制相同，TCP 也通过控制发送到网络中的数据量来达到拥塞控制，避免数据丢失的目的。TCP 发送方维护着拥塞窗口 cwnd 值，发送方发送到网络中，但尚未到达的数据量不能超过拥塞窗口值，这称为拥塞控制。有许多拥塞控制算法用于确定 cwnd 值。

TCP 实际的发送窗口（发送到网络上，但尚示到达的数据量）由 rwnd 和 cwnd 中的最小值决定，如下图所示，因为 TCP 需要同时满足流量控制和拥塞控制的条件。



2 套接字框架

TCP 套接字框架如下图所示：



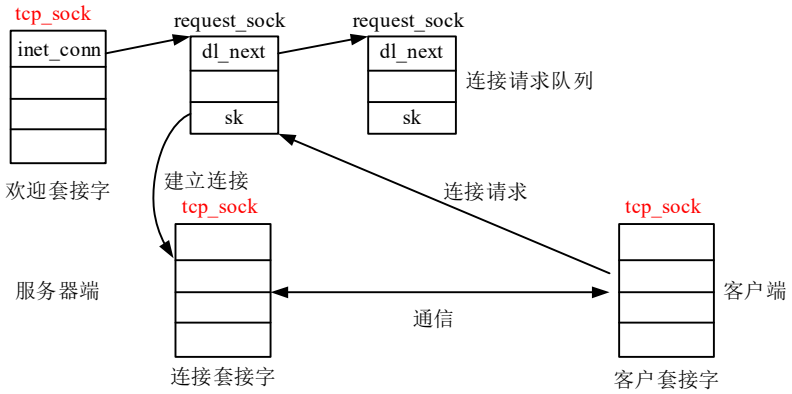
由前面介绍的初始化函数 inet_init()中注册的 inet_protosw 实例可知，TCP 套接字关联的 proto_ops 实例为 **inet_stream_ops**，proto 实例为 **tcp_prot**。

tcp_congestion_ops 结构体表示拥塞控制算法，inet_connection_sock_af_ops 结构体表示特定于网络层协议的操作结构，如发送数据包至网络层等。

TCP 套接字由 tcp_sock 结构体表示，tcp_prot 实例中通过散列表管理 tcp_sock 实例。

在使用 TCP 进行通信前，双方需要先建立连接。前面介绍过 TCP 通过 3 次握手建立连接。如下图所示，假设连接操作由客户发起。服务器端首先创建的是一个欢迎套接字，执行监听和接受连接请求操作。客户端提交的连接请求由欢迎套接字接收，连接请求添加到请求队列。3 次握手完成后，欢迎套接字将为连接请求创建连接套接字，客户套接字与连接套接字建立连接，相互通信。

服务器端执行接受连接请求操作后，连接套接字被赋予一个文件描述符，服务器端可通过文件描述符操作套接字，与客户端通信。



3 数据结构

下面介绍 TCP 实现中相关的数据结构。

■tcphdr

TCP 报头由 tcphdr 结构体表示，定义如下（/include/uapi/linux/tcp.h）：

```
struct tcphdr {
    __be16  source;     /*源端口号，大端序*/
    __be16  dest;       /*目的端口号*/
    __be32  seq;        /*序列号*/
    __be32  ack_seq;    /*确认号*/
#if defined(__LITTLE_ENDIAN_BITFIELD)    /*小端体系结构*/
    __u16   res1:4,     /*保留*/
        doff:4,     /*头部长度，4 字节为单位*/
        fin:1,      /*指示后面没有来自发送方的其他数据，用于关闭连接*/
        syn:1,      /*在建立连接进行 3 次握手时发送 SYN 标志*/
        rst:1,      /*复位标志*/
        psh:1,      /*指出应尽快将数据交给用户进程*/
        ack:1,      /*指示 TCP 报头中确认号是否有意义*/
        urg:1,      /*指示紧急指针是否有意义（很少使用）*/
        ece:1,      /*显式拥塞标志*/
        cwr:1;      /*拥塞窗口缩小标志，提示发送方降低发送速率*/
#elif defined(__BIG_ENDIAN_BITFIELD)    /*大端体系结构*/
    ...
#else
    ...
#endif
    __be16  window;     /*接收窗口长度（空闲字节数）*/
    __sum16 check;      /*校验和*/
    __be16  urg_ptr;    /*紧急指针*/
};
```

tcphdr 结构体表示的 TCP 报头结构与前面介绍的 TCP 报头结构相同，即 TCP 报头在内核中的表示，报头字段含义请读者参考前面的内容。

■**tcp_sock**

在 TCP 实现中，TCP 套接字由 tcp_sock 结构体表示，定义如下（/include/linux/tcp.h）：

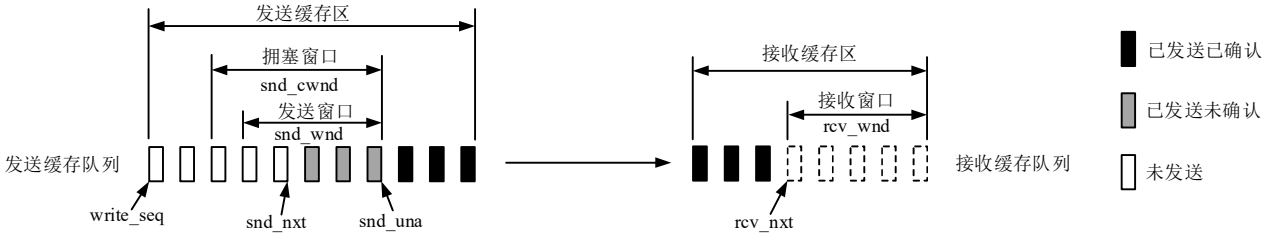
```
struct tcp_sock {
    struct inet_connection_sock  inet_conn;       /*必须是第一个成员*/
    u16  tcp_header_len;          /*TCP 报头长度（字节数）*/
    u16  gso_segs;                /*GSO 数据包中分段数量*/

    __be32   pred_flags;   /*结构等同于 TCP 报头中第 4 个 32 位字，含报头长度、标志和接收窗口*/

    u64  bytes_received;
    u32  segs_in;
    u32  rcv_nxt;        /*下一个期望接收的字节编号（确认号）*/
    u32  copied_seq;     /*暂未读取字节序列号*/
    u32  rcv_wup;        /* rcv_nxt on last window update sent   */
    u32  snd_nxt;        /*下一次发送数据的序列号*/
    u32  segs_out;
    u64  bytes_acked;
    struct u64_stats_sync syncp;    /* protects 64bit vars (cf tcp_get_info()) */

    u32  snd_una;        /*等待确认的第一个字节（序列号）*/
    u32  snd_sml;       /* Last byte of the most recently transmitted small packet */
    u32  rcv_tstamp;     /* timestamp of last received ACK (for keepalives) */
    u32  lsndtime;      /* timestamp of last sent data packet (for restart window) */
    u32  last_oow_ack_time;    /* timestamp of last out-of-window ACK */

    u32  tsoffset;       /* timestamp offset */

    struct list_head    tsq_node;     /* anchor in tsq_tasklet.head list */
    unsigned long   tsq_flags;

    /*表示在等待接收缓存队列中数据的进程，初始化为 NULL*/
    struct {
        struct sk_buff_head  prequeue;       /*预备队列*/
        struct task_struct    *task;
        struct msghdr         *msg;
        int                   memory;
        int                   len;
    } ucopy;

    u32  snd_wl1;        /* Sequence for window update*/
    u32  snd_wnd;            /*发送窗口，接收方通告的接收窗口*/
    u32  max_window;        /* Maximal window ever seen from peer*/
    u32  mss_cache;         /*缓存的 MSS 值，来自路由选择查找结果*/

    u32  window_clamp;      /* Maximal window to advertise*/
```

```
u32  rcv_ssthresh;         /* Current window clamp*/

u16  advmss;               /* Advertised MSS*/
u8   unused;
u8   nonagle        : 4,   /*是否禁止 Nagle 算法？ */
     thin_lto       : 1,    /* Use linear timeouts for thin streams */
     thin_dupack : 1,    /* Fast retransmit on first dupack */
     repair         : 1,
     frto           : 1;    /* F-RTO (RFC5682) activated in CA_Loss */
u8   repair_queue;
u8   do_early_retrans:1,      /* Enable RFC5827 early-retransmit    */
     syn_data:1,               /* SYN includes data */
     syn_fastopen:1,           /* SYN includes Fast Open option */
     syn_fastopen_exp:1,    /* SYN includes Fast Open exp. option */
     syn_data_acked:1,      /* data in SYN is acked by SYN-ACK */
     save_syn:1,           /* Save headers of SYN packet */
     is_cwnd_limited:1;    /* forward progress limited by snd_cwnd? */
u32  tlp_high_seq;         /* snd_nxt at the time of TLP retransmit. */

/*RTT 测量*/
u32  srtt_us;       /* smoothed round trip time << 3 in usecs */
u32  mdev_us;       /* medium deviation*/
u32  mdev_max_us;       /* maximal mdev for the last rtt period  */
u32  rttvar_us;       /* smoothed mdev_max*/
u32  rtt_seq;          /* sequence number to update rttvar*/

u32  packets_out;   /*发出但尚未确认数据包数量（还在传输的数据包）*/
u32  retrans_out;    /* Retransmitted packets out*/
u32  max_packets_out;   /* max packets_out in last window */
u32  max_packets_seq;   /* right edge of max_packets_out flight */

u16  urg_data;      /* Saved octet of OOB data and control flags */
u8   ecn_flags;      /* ECN 标志位*/
u8   keepalive_probes; /* num of allowed keep alive probes*/
u32  reordering;     /* Packet reordering metric.*/
u32  snd_up;         /* Urgent pointer*/

struct tcp_options_received    rx_opt;       /*接收时的 TCP 选项*/

/*慢启动和拥塞控制参数*/
u32  snd_ssthresh;     /*拥塞窗口减速阀值*/
u32  snd_cwnd;         /*拥塞窗口*/
u32  snd_cwnd_cnt;      /* Linear increase counter，线性增加计数值*/
u32  snd_cwnd_clamp;   /* Do not allow snd_cwnd to grow above this */
u32  snd_cwnd_used;
u32  snd_cwnd_stamp;
```

```
u32  prior_cwnd;          /* Congestion window at start of Recovery.  重置时拥塞窗口初始值*/
u32  prr_delivered;       /* Number of newly delivered packets to receiver in Recovery. */
u32  prr_out;      /* Total number of pkts sent during Recovery. */

u32  rcv_wnd;             /*本端接收窗口长度，接收缓存区的空闲区域大小，通告给发送方*/
u32  write_seq;           /*发送缓存队列中最后数据的序列号*/
u32  notsent_lowat;       /* TCP_NOTSENT_LOWAT */
u32  pushed_seq;         /* Last pushed seq, required to talk to windows */
u32  lost_out;          /* Lost packets*/
u32  sacked_out;      /* SACK'd packets*/
u32  fackets_out;     /* FACK'd packets*/

/* from STCP, retrans queue hinting */
struct sk_buff* lost_skb_hint;
struct sk_buff *retransmit_skb_hint;

struct sk_buff_head  out_of_order_queue;       /*失序到达数据包队列*/

/* SACKs data, these 2 need to be together (see tcp_options_write)，选择确认数据*/
struct tcp_sack_block duplicate_sack[1];   /* D-SACK block */
struct tcp_sack_block selective_acks[4];   /* The SACKS themselves*/

struct tcp_sack_block    recv_sack_cache[4];
struct sk_buff    *highest_sack;     /* */
int       lost_cnt_hint;
u32     retransmit_high;   /* L-bits may be on up to this seqno */
u32  lost_retrans_low;    /* Sent seq after any rxmit (lowest) */
u32  prior_ssthresh;     /* ssthresh saved at recovery start，拥塞阀值*/
u32  high_seq;       /* snd_nxt at onset of congestion   */

u32  retrans_stamp;    /* */
u32  undo_marker;      /*snd_una upon a new recovery episode. */
int    undo_retrans;      /* number of undoable retransmissions. */
u32  total_retrans;      /* Total retransmits for entire connection */

u32  urg_seq;         /* Seq of received urgent pointer */
unsigned int          keepalive_time;         /* time before keep alive takes place */
unsigned int          keepalive_intvl;        /* time interval between keep alive probes */

int    linger2;

/*接收方 RTT 估值*/
struct {
      u32  rtt;
      u32  seq;
      u32  time;
```

```
    } rcv_rtt_est;

    /*接收队列空间*/
    struct {
        int   space;
        u32  seq;
        u32  time;
    } rcvq_space;

    /* TCP-specific MTU probe information，MTU 探测信息*/
    struct {
        u32          probe_seq_start;
        u32          probe_seq_end;
    } mtu_probe;
    u32  mtu_info; /* */

#ifdef CONFIG_TCP_MD5SIG
    ...
#endif

    struct tcp_fastopen_request *fastopen_req;
    struct request_sock *fastopen_rsk;
    u32  *saved_syn;
};
```

tcp_sock 结构体中第一个成员为 inet_connection_sock 结构体，表示通用的面向连接套接字的连接信息。tcp_sock 结构体剩余成员主要记录套接字发送、接收通道的参数，如发送窗口、接收窗口、拥塞控制参数等等。下面简要列出发送和接收通道中的几个控制参数：



上图中画出了一个连接的发送方和接收方，发送方和接收方都定义了一个缓存区，用于限制缓存队列（缓存内存）的长度。接收方和发送方缓存队列中参数简介如下：

接收方：

●**rcv_nxt**：表示下一个期望收到字节的序列号，也就是应答给发送方的确认号。

●**rcv_wnd**：表示应答给发送方的接收窗口，即接收缓存中的空闲区域大小。

发送方：

●**write_seq**：发送缓存队列中最后字节的序列号。

●**snd_una**：最早的未确认过的序列号。

●**snd_wnd**：发送窗口，即接收方应答的接收窗口。已发送但尚未确认的数据长度不能超过此值。

●**snd_nxt**：下一个发送字节的序列号。snd_nxt 并不一定是最后发送字节的下一个字节，因为在重传操作中，snd_nxt 需要返回到之前发送过的字节。

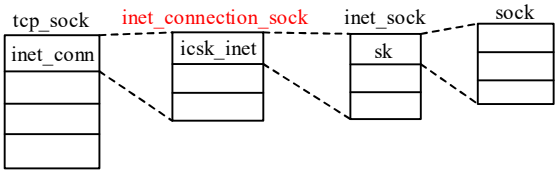
●**snd_cwnd**：拥塞窗口，发送方已发送到网络但尚未确认的数据长度不能超过拥塞窗口，也就是说发送操作需要同时满足发送窗口和拥塞窗口的限制。

由于 TCP 是全双工通信，套接字同时存在发送缓存队列和接收缓存队列，因此套接字同时具有发送方和接收方的参数，上图中只标识了一个通信方向中双方的参数。

发送方在接收到带应答 ACK 的数据包时，将调整发送窗口、拥塞窗口的位置（左移）和长度，因此以上参数是动态变化的。

■inet_connection_sock

tcp_sock 结构体第一个成员 inet_conn 为 inet_connection_sock 结构体，结构体组织关系如下图所示：



inet_connection_sock 结构体内嵌 inet_sock 结构体成员，定义如下（/include/net/inet_connection_sock.h）：

```
struct inet_connection_sock {
    struct inet_sock        icsk_inet;      /*inet_sock 结构体成员，必须是第一个成员*/
    struct request_sock_queue   icsk_accept_queue;    /*连接请求队列结构*/
    struct inet_bind_bucket     *icsk_bind_hash;      /*添加到 tcp_proto.hashinfo.bhash 散列表*/
    unsigned long           icsk_timeout;             /*超时时间*/
    struct timer_list       icsk_retransmit_timer;    /*重传定时器（没有收到应答）*/
    struct timer_list       icsk_delack_timer;        /*确定删除的定时器*/
    __u32                   icsk_rto;                 /*重传超时时间*/
    __u32                   icsk_pmtu_cookie;         /*最近探测的 MTU*/
    const struct tcp_congestion_ops *icsk_ca_ops;     /*拥塞控制算法回调函数，/include/net/tcp.h*/
    const struct inet_connection_sock_af_ops *icsk_af_ops;    /*IPv4、IPv6 网络层定义的操作结构*/
    unsigned int            (*icsk_sync_mss)(struct sock *sk, u32 pmtu);    /*同步 MSS 的函数指针*/
    __u8                    icsk_ca_state:6,          /*拥塞控制状态*/
                            icsk_ca_setsockopt:1,
                            icsk_ca_dst_locked:1;
    __u8                    icsk_retransmits;         /*重发数量*/
    __u8                    icsk_pending;             /*挂起*/
    __u8                    icsk_backoff;             /*允许连接的数量*/
    __u8                    icsk_syn_retries;         /*允许重新 SYN 的数量*/
    __u8                    icsk_probes_out;          /*探测到的未应答的窗口*/
    __u16                   icsk_ext_hdr_len;         /*网络层报头选项长度*/
    struct {
        __u8            pending;        /*ACK is pending*/
        __u8            quick;          /*Scheduled number of quick acks*/
        __u8            pingpong;       /* The session is interactive*/
        __u8            blocked;        /* Delayed ACK was blocked by socket lock */
        __u32           ato;            /* Predicted tick of soft clock*/
        unsigned long   timeout;        /* Currently scheduled timeout*/
        __u32           lrcvtime;       /* timestamp of last received data packet */
```

```
        __u16           last_seg_size; /* Size of last incoming segment*/
        __u16           rcv_mss;    /* MSS used for delayed ACK decisions*/
    } icsk_ack;     /*延时应答控制参数*/
    struct {
        int         enabled;

        /* Range of MTUs to search */
        int         search_high;
        int         search_low;

        /* Information on the current probe. */
        int         probe_size;
        u32         probe_timestamp;
    } icsk_mtup;    /*MTU 探测控制参数*/

    u32             icsk_user_timeout;
    u64             icsk_ca_priv[64 / sizeof(u64)];     /*私有数据*/
#define ICSK_CA_PRIV_SIZE       (8 * sizeof(u64))
};
```

inet_connection_sock 结构体表示面向连接的套接字，主要成员简介如下：

●**icsk_inet**：inet_sock 结构体成员，结构体定义在/include/net/inet_sock.h 头文件，见本章上文。

●**icsk_accept_queue**：request_sock_queue 结构体成员，表示连接请求队列，见下文。

●**icsk_bind_hash**：指向 inet_bind_bucket 结构体，结构体实例添加到 tcp_proto.hashinfo.bhash 散列表，结构体定义见下文。

●**icsk_ca_ops**：指向 tcp_congestion_ops 结构体，表示拥塞控制算法，定义如下（/include/net/tcp.h）：

```
struct tcp_congestion_ops {
    struct list_head    list;
    u32 key;        /*键值*/
    u32 flags;

    void (*init)(struct sock *sk);     /*初始化私有数据，可选*/
    void (*release)(struct sock *sk);      /*清除私有数据，可选*/

    u32 (*ssthresh)(struct sock *sk);       /*回到慢启动，必须实现*/
    void (*cong_avoid)(struct sock *sk, u32 ack, u32 acked);    /*计算新拥塞窗口，必须实现*/
    void (*set_state)(struct sock *sk, u8 new_state);    /*修改拥塞状态前调用，可选*/
    void (*cwnd_event)(struct sock *sk, enum tcp_ca_event ev);  /*拥塞窗口事件发生时调用，可选*/
    void (*in_ack_event)(struct sock *sk, u32 flags);    /*收到应答时调用，可选*/
    u32   (*undo_cwnd)(struct sock *sk);   /*发生丢失时，新拥塞窗口，可选*/
    void (*pkts_acked)(struct sock *sk, u32 num_acked, s32 rtt_us);  /*应答说明，可选*/

    size_t (*get_info)(struct sock *sk, u32 ext, int *attr,union tcp_cc_info *info);/*获取诊断信息，可选*/

    char            name[TCP_CA_NAME_MAX];      /*拥塞控制算法名称，用于标识算法*/
    struct module  *owner;      /*模块指针*/
};
```

tcp_congestion_ops 结构体用于实现一个拥塞控制算法，用户可通过模块的方式加载拥塞控制算法。算法实例由全局双链表 **tcp_cong_list** 管理。

●icsk_ca_state：拥塞控制状态，取值定义在/include/uapi/linux/tcp.h 头文件：

```
enum tcp_ca_state {
    TCP_CA_Open = 0,
#define TCPF_CA_Open    (1<<TCP_CA_Open)
    TCP_CA_Disorder = 1,
#define TCPF_CA_Disorder (1<<TCP_CA_Disorder)
    TCP_CA_CWR = 2,
#define TCPF_CA_CWR    (1<<TCP_CA_CWR)
    TCP_CA_Recovery = 3,
#define TCPF_CA_Recovery   (1<<TCP_CA_Recovery)
    TCP_CA_Loss = 4
#define TCPF_CA_Loss     (1<<TCP_CA_Loss)
};
```

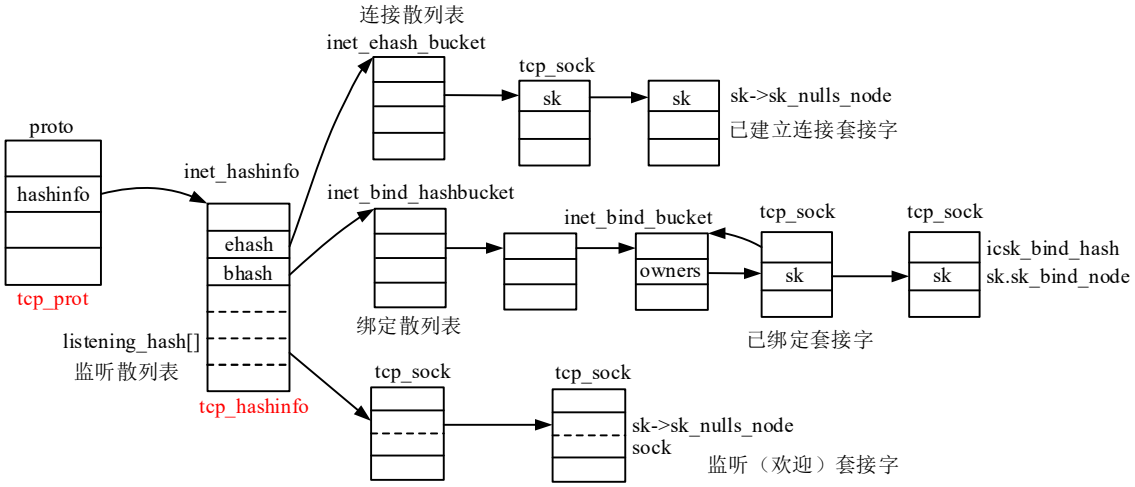
●icsk_af_ops：inet_connection_sock_af_ops 结构体指针，表示网络层协议定义的操作（如 IPv4、IPv6），结构体定义如下（/include/net/inet_connection_sock.h）：

```
struct inet_connection_sock_af_ops {
    int         (*queue_xmit)(struct sock *sk, struct sk_buff *skb, struct flowi *fl);
                                                        /*发送数据包至网络层函数*/
    void        (*send_check)(struct sock *sk, struct sk_buff *skb);
    int         (*rebuild_header)(struct sock *sk);
    void        (*sk_rx_dst_set)(struct sock *sk, const struct sk_buff *skb);
    int         (*conn_request)(struct sock *sk, struct sk_buff *skb);     /*接收连接 SYN 请求，并响应*/
    struct sock *(*syn_recv_sock)(struct sock *sk, struct sk_buff *skb,struct request_sock *req,
                                                        struct dst_entry *dst);

    u16         net_header_len;      /*网络层报头长度*/
    u16         net_frag_header_len;
    u16         sockaddr_len;
    int         (*setsockopt)(struct sock *sk, int level, int optname, char __user *optval, unsigned int optlen);
    int         (*getsockopt)(struct sock *sk, int level, int optname, char __user *optval, int __user *optlen);
                                        /*处理非 SOL_SOCKET、SOL_TCP 级别的参数*/
  #ifdef CONFIG_COMPAT
    ...
  #endif
    void        (*addr2sockaddr)(struct sock *sk, struct sockaddr *);
    int         (*bind_conflict)(const struct sock *sk,const struct inet_bind_bucket *tb, bool relax);
                                                        /*检查套接字冲突*/
    void        (*mtu_reduced)(struct sock *sk);
};
```

inet_connection_sock_af_ops 结构体中函数表示特定于网络层协议的操作函数，例如：**queue_xmit()**表示将数据包发送到网络层的函数。

4 套接字管理

TCP 套接字关联的 proto 实例 tcp_prot 中，其 hashinfo 成员指向 inet_hashinfo 结构体实例用于管理套接字 tcp_sock 实例，如下图所示。



inet_hashinfo 结构体中包含 3 个散列表，分别是绑定散列表、监听散列表和连接散列表。已绑定端口号的套接字添加到绑定散列表，已建立连接处于 TCP_ESTABLISHED 状态的套接字添加到连接散列表，处于 TCP_LISTEN 状态的监听套接字添加到监听散列表。

服务器进程创建的欢迎套接字既处于监听状态，又绑定了端口号，因此同时位于监听散列表和绑定散列表。服务器创建的连接套接字已建立连接并绑定了端口号，因此同时位于绑定散列表和连接散列表。客户进程创建的已建立连接的套接字添加到连接散列表和绑定散列表。

服务器欢迎套接字及其创建的连接套接字具有相同的端口号，在绑定散列表中由 inet_bind_bucket 实例管理，位于其下的 owners 链表中。

■数据结构

inet_hashinfo 结构体用于管理 tcp_sock 实例，定义在/include/net/inet_hashtables.h 头文件：

```
struct inet_hashinfo {
    struct inet_ehash_bucket*ehash;     /*指向 inet_ehash_bucket 数组组成的散列表*/
                    /*管理已建立连接的套接字，TCP_ESTABLISHED <= sk->sk_state < TCP_CLOSE*/

    spinlock_t              *ehash_locks;       /*锁列表*/
    unsigned int            ehash_mask;
    unsigned int            ehash_locks_mask;
    struct inet_bind_hashbucket   *bhash;     /*指向 inet_bind_hashbucket 数组表示的散列表*/
                                    /*管理已绑定端口号的套接字*/
    unsigned int            bhash_size;
    /* 4 bytes hole on 64 bit */
    struct kmem_cache       *bind_bucket_cachep;       /*inet_bind_bucket 结构体 slab 缓存*/
    struct inet_listen_hashbucket listening_hash[INET_LHTABLE_SIZE]       /*监听散列表*/
                                    ____cacheline_aligned_in_smp;     /*TCP_LISTEN 状态套接字*/

};
```

inet_hashinfo 结构体主要成员简介如下：

●**ehash**：指向 inet_ehash_bucket 结构体数组，表示散列表，用于管理已经建立连接的 tcp_sock 实例。

inet_ehash_bucket 结构体定义如下：

```
struct inet_ehash_bucket {
    struct hlist_nulls_head chain;    /*散列链表头*/
};
```

●**bhash**：指向 inet_bind_hashbucket 结构体数组，表示散列表，用于管理已绑定端口号 tcp_sock 实例，结构体定义如下（/include/net/inet_hashtables.h）：

```
struct inet_bind_hashbucket {
    spinlock_t          lock;     /*带自旋锁的链表*/
    struct hlist_head   chain;    /*散列链表头*/
};
```

inet_bind_hashbucket 结构体数组表示的散列表，管理的是 inet_bind_bucket 结构体实例，结构体实例中管理了同一端口号的套接字。套接字通过所属网络命名空间和端口号计算散列值。

●**bind_bucket_cachep**：指向 inet_bind_bucket 结构体 slab 缓存。

inet_bind_bucket 结构体定义如下（/include/net/inet_hashtables.h）：

```
struct inet_bind_bucket {
    possible_net_t      ib_net;       /*网络命名空间*/
    unsigned short      port;         /*管理套接字端口号*/
    signed char         fastreuse;
    signed char         fastreuseport;
    kuid_t              fastuid;
    int                 num_owners;   /*具有此端口号的套接字数量*/
    struct hlist_node   node;         /*添加到 bhash 散列表*/
    struct hlist_head   owners;       /*管理相同端口号的 tcp_sock 实例*/
};
```

服务器欢迎套接字与其创建的连接套接字具有相同的端口号，由同一个 inet_bind_bucket 实例管理，由其 owners 链表管理。

●**listening_hash[]**：inet_listen_hashbucket 结构体数组， 表示散列表，管理 TCP_LISTEN 状态套接字，结构体定义如下（/include/net/inet_hashtables.h）：

```
struct inet_listen_hashbucket {
    spinlock_t          lock;
    struct hlist_nulls_head   head;    /*散列链表头*/
};
```

内核在/net/ipv4/tcp_ipv4.c 文件内定义了 inet_hashinfo 结构体实例，并赋予 tcp_prot.h.hashinfo 成员：

```
struct inet_hashinfo tcp_hashinfo;
```

■接口函数

下面简要介绍将 TCP 套接字 tcp_sock 实例添加到以上各散列表的接口函数。

●插入绑定散列表

在绑定操作及为套接字分配端口号后，需要将套接字添加到绑定散列表。以下接口函数用于创建 inet_bind_bucket 结构体实例，并插入到绑定散列表：

```
struct inet_bind_bucket *inet_bind_bucket_create(struct kmem_cache *cachep,struct net *net,
                                                 struct inet_bind_hashbucket *head,const unsigned short snum);
```

/*cachep：slab 缓存，net：网络命名空间，head：bhash 散列链表头，snum：端口号*/

inet_bind_hash()函数用于建立 inet_bind_bucket 实例与 tcp_sock 实例之间的关联，函数定义如下：

```
void inet_bind_hash(struct sock *sk, struct inet_bind_bucket *tb,const unsigned short snum)
{
    inet_sk(sk)->inet_num = snum;       /*设置端口号*/
    sk_add_bind_node(sk, &tb->owners);
                /*将 tcp_sock 实例添加到 tb->owners 链表，sk->sk_bind_node*/
    tb->num_owners++;       /*套接字计数加 1*/
    inet_csk(sk)->icsk_bind_hash = tb;     /*指向 inet_bind_bucket 实例*/
}
```

●插入监听或连接散列表

TCP 套接字关联的 proto 实例 tcp_prot 中的 inet_hash()函数用于将套接字插入到监听散列表或连接散列表。inet_hash()函数定义如下（/net/ipv4/inet_hashtables.c）：

```
void inet_hash(struct sock *sk)
{
    if (sk->sk_state != TCP_CLOSE) {     /*套接字不能为 TCP_CLOSE 状态*/
        local_bh_disable();
        __inet_hash(sk, NULL);
        local_bh_enable();
    }
}
```

__inet_hash()函数定义如下：

```
int __inet_hash(struct sock *sk, struct inet_timewait_sock *tw)
/*sk：套接字，tw：NULL*/
{
    struct inet_hashinfo *hashinfo = sk->sk_prot->h.hashinfo;
    struct inet_listen_hashbucket *ilb;

    if (sk->sk_state != TCP_LISTEN)       /*套接字不处于监听状态*/
        return __inet_hash_nolisten(sk, tw);       /*将套接字添加到连接散列表*/

    /*以下处理处于监听状态的套接字*/
    WARN_ON(!sk_unhashed(sk));
    ilb = &hashinfo->listening_hash[inet_sk_listen_hashfn(sk)];     /*监听散列链表头*/

    spin_lock(&ilb->lock);
    __sk_nulls_add_node_rcu(sk, &ilb->head);       /*将套接字添加到监听散列链表*/
    sock_prot_inuse_add(sock_net(sk), sk->sk_prot, 1);     /*增加计数值*/
    spin_unlock(&ilb->lock);
    return 0;
}
```

5 TCP 初始化

在 IPv4 协议簇初始化函数 inet_init()将对 TCP 进行初始化，函数调用关系如下：

```
static int __init inet_init(void)
{
    ...
    tcp_v4_init();  /*为每个 CPU 核创建原始 TCP 套接字，设置 TCP 参数等，/net/ipv4/tcp_ipv4.c*/
    tcp_init();     /*TCP 初始化，/net/ipv4/tcp.c*/
    ...
}
```

tcp_v4_init()函数定义如下（/net/ipv4/tcp_ipv4.c）：

```
void __init tcp_v4_init(void)
{
    inet_hashinfo_init(&tcp_hashinfo);    /*/net/ipv4/inet_hashtables.c*/
                                          /*初始化 tcp_hashinfo->listening_hash[]散列表*/
    if (register_pernet_subsys(&tcp_sk_ops))
        panic("Failed to create the TCP control socket.\n");
}
```

tcp_v4_init()函数内对 tcp_hashinfo->listening_hash[]散列表进行初始化，注册 pernet_operations 结构体实例 **tcp_sk_ops**。tcp_sk_ops 实例中初始化函数为 tcp_sk_init()，定义如下：

```
static int __net_init tcp_sk_init(struct net *net)
{
    int res, cpu;

    net->ipv4.tcp_sk = alloc_percpu(struct sock *);    /*percpu 变量*/
    ...  /*错误处理*/

    for_each_possible_cpu(cpu) {       /*为每个 CPU 核创建 TCP 原始套接字*/
        struct sock *sk;
        res = inet_ctl_sock_create(&sk, PF_INET, SOCK_RAW,IPPROTO_TCP, net);
        ...
        *per_cpu_ptr(net->ipv4.tcp_sk, cpu) = sk;
    }

    /*以下是设置 TCP 系统参数*/
    net->ipv4.sysctl_tcp_ecn = 2;
    net->ipv4.sysctl_tcp_ecn_fallback = 1;
    net->ipv4.sysctl_tcp_base_mss = TCP_BASE_MSS;
    net->ipv4.sysctl_tcp_probe_threshold = TCP_PROBE_THRESHOLD;
    net->ipv4.sysctl_tcp_probe_interval = TCP_PROBE_INTERVAL;

    return 0;
    ...
}
```

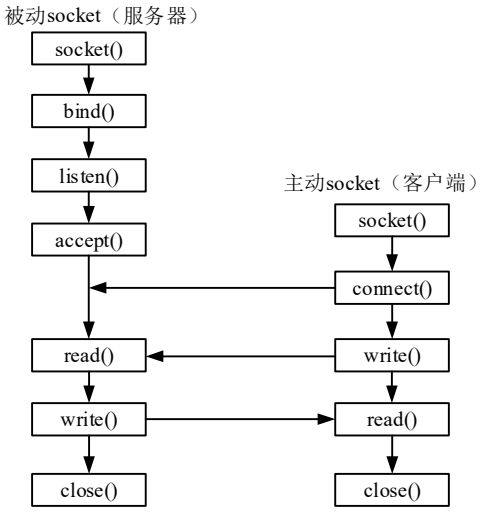
tcp_init()函数定义如下（/net/ipv4/tcp.c）：

```
void __init tcp_init(void)
{
    unsigned long limit;
    int max_rshare, max_wshare, cnt;
    unsigned int i;

    sock_skb_cb_check_size(sizeof(struct tcp_skb_cb));

    percpu_counter_init(&tcp_sockets_allocated, 0, GFP_KERNEL);
    percpu_counter_init(&tcp_orphan_count, 0, GFP_KERNEL);
    tcp_hashinfo.bind_bucket_cachep =
        kmem_cache_create("tcp_bind_bucket",sizeof(struct inet_bind_bucket), 0,
                    SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
                                    /*为 inet_bind_bucket 结构体创建 slab 缓存*/
    tcp_hashinfo.ehash =                /*创建 inet_ehash_bucket 散列表*/
        alloc_large_system_hash("TCP established",
                    sizeof(struct inet_ehash_bucket),
                    thash_entries,
                    17, /* one slot per 128 KB of memory */
                    0,
                    NULL,
                    &tcp_hashinfo.ehash_mask,
                    0,
                    thash_entries ? 0 : 512 * 1024);
    for (i = 0; i <= tcp_hashinfo.ehash_mask; i++)        /*初始化散列表*/
        INIT_HLIST_NULLS_HEAD(&tcp_hashinfo.ehash[i].chain, i);

    if (inet_ehash_locks_alloc(&tcp_hashinfo))     /*为 tcp_hashinfo->ehash_locks 创建锁列表*/
        panic("TCP: failed to alloc ehash_locks");
    tcp_hashinfo.bhash =              /*创建 inet_bind_hashbucket 散列表*/
        alloc_large_system_hash("TCP bind",
                    sizeof(struct inet_bind_hashbucket),
                    tcp_hashinfo.ehash_mask + 1,
                    17, /* one slot per 128 KB of memory */
                    0,
                    &tcp_hashinfo.bhash_size,
                    NULL,
                    0,
                    64 * 1024);
    tcp_hashinfo.bhash_size = 1U << tcp_hashinfo.bhash_size;
    for (i = 0; i < tcp_hashinfo.bhash_size; i++) {        /*初始化 bhash 散列表*/
        spin_lock_init(&tcp_hashinfo.bhash[i].lock);
        INIT_HLIST_HEAD(&tcp_hashinfo.bhash[i].chain);
    }
```

```
    cnt = tcp_hashinfo.ehash_mask + 1;

    tcp_death_row.sysctl_max_tw_buckets = cnt / 2;
    sysctl_tcp_max_orphans = cnt / 2;
    sysctl_max_syn_backlog = max(128, cnt / 256);

    tcp_init_mem();     /*初始化系统控制参数 sysctl_tcp_mem[3]，/net/ipv4/tcp.c*/
    /* Set per-socket limits to no more than 1/128 the pressure threshold */
    limit = nr_free_buffer_pages() << (PAGE_SHIFT - 7);
    max_wshare = min(4UL*1024*1024, limit);
    max_rshare = min(6UL*1024*1024, limit);

    sysctl_tcp_wmem[0] = SK_MEM_QUANTUM;       /*发送窗口长度*/
    sysctl_tcp_wmem[1] = 16*1024;              /*TCP 套接字初始值*/
    sysctl_tcp_wmem[2] = max(64*1024, max_wshare);

    sysctl_tcp_rmem[0] = SK_MEM_QUANTUM;       /*接收窗口长度*/
    sysctl_tcp_rmem[1] = 87380;                /*TCP 套接字初始值*/
    sysctl_tcp_rmem[2] = max(87380, max_rshare);
    ...

    tcp_metrics_init();     /*初始化 tcpm_hash_bucket 散列表，/net/ipv4/tcp_metrics.c*/
    BUG_ON(tcp_register_congestion_control(&tcp_reno) != 0);       /*注册 tcp_reno 拥塞控制算法*/
    tcp_tasklet_init();         /*初始化每个 CPU 核 tsq_tasklet 实例，/net/ipv4/tcp_output.c*/
}
```

6 套接字操作

下图示意了流套接字常用的系统调用：



流socket上用到的系统调用

socket()系统调用用于创建套接字，被动套接字（服务器）调用 bind()将自身绑定到一个众所周知的端口号，以便主动套接字（客户）寻址到它，然后调用 listen()通知内核它接受接入连接的意愿，调用 accept()接受连接，accept()在没有建立连接时会阻塞直到有客户连接，读写操作系统调用（如 read()、write()）用

于接收和发送数据，close()用于关闭套接字。

主动套接字调用 connect()建立与被动套接字的连接（需要指定被动套接字 IP 地址和端口号），建立连接后通过 read()、write()等系统调用接收和发送数据，close()用于关闭套接字。

套接字操作系统调用内调用 proto_ops 实例中的对应函数完成操作，TCP 套接字为 inet_stream_ops 实例，定义如下（/net/ipv4/af_inet.c）：

```
const struct proto_ops inet_stream_ops = {
    .family        = PF_INET,
    .owner         = THIS_MODULE,
    .release       = inet_release,       /*调用 proto->close()函数关闭连接*/
    .bind          = inet_bind,     /*绑定操作*/
    .connect       = inet_stream_connect,       /*主动连接操作*/
    .socketpair        = sock_no_socketpair,
    .accept            = inet_accept,    /*接受连接请求*/
    .getname       = inet_getname,       /*获取套接字目的或源地址（含端口号）*/
    .poll          = tcp_poll,
    .ioctl             = inet_ioctl,       /*IO 控制*/
    .listen            = inet_listen,     /*监听连接请求*/
    .shutdown          = inet_shutdown,
    .setsockopt        = sock_common_setsockopt,        /*设置参数*/
    .getsockopt        = sock_common_getsockopt,        /*获取参数*/
    .sendmsg       = inet_sendmsg,       /*发送消息*/
    .recvmsg       = inet_recvmsg,       /*接收消息*/
    .mmap          = sock_no_mmap,
    .sendpage      = inet_sendpage,
    .splice_read       = tcp_splice_read,
  #ifdef CONFIG_COMPAT
    ...
  #endif
};
```

inet_stream_ops 实例与 UDP 套接字 inet_dgram_ops 实例中的大部分函数相同，其中的函数主要调用 proto 实例中的对应函数完成操作。

TCP 套接字 proto 实例为 tcp_prot，定义如下（/net/ipv4/tcp_ipv4.c）：

```
struct proto tcp_prot = {        /*没有定义 bind()函数*/
    .name              = "TCP",
    .owner             = THIS_MODULE,
    .close             = tcp_close,     /*关闭套接字，调用 tcp_prot.disconnect()函数*/
    .connect       = tcp_v4_connect,       /*主动连接操作*/
    .disconnect        = tcp_disconnect,       /*断开连接*/
    .accept            = inet_csk_accept,     /*接受连接请求*/
    .ioctl             = tcp_ioctl,          /*IO 控制*/
    .init          = tcp_v4_init_sock,       /*创建套接字调用，用于初始化套接字*/
    .destroy       = tcp_v4_destroy_sock,
    .shutdown          = tcp_shutdown,
    .setsockopt        = tcp_setsockopt,       /*设置参数*/
    .getsockopt        = tcp_getsockopt,       /*获取参数*/
    .recvmsg       = tcp_recvmsg,        /*接收消息*/
```

```
    .sendmsg        = tcp_sendmsg,          /*发送消息*/
    .sendpage       = tcp_sendpage,
    .backlog_rcv        = tcp_v4_do_rcv,        /*接收后备队列中的数据包*/
    .release_cb         = tcp_release_cb,
    .hash               = inet_hash,        /*将 tcp_sock 实例插入监听/连接散列表*/
    .unhash             = inet_unhash,
    .get_port       = inet_csk_get_port,        /*绑定操作中调用，设置（分配）端口号，插入散表列*/
    .enter_memory_pressure= tcp_enter_memory_pressure,
    .stream_memory_free     = tcp_stream_memory_free,
    .sockets_allocated  = &tcp_sockets_allocated,
    .orphan_count       = &tcp_orphan_count,
    .memory_allocated = &tcp_memory_allocated,
    .memory_pressure  = &tcp_memory_pressure,
    .sysctl_mem         = sysctl_tcp_mem,
    .sysctl_wmem        = sysctl_tcp_wmem,
    .sysctl_rmem        = sysctl_tcp_rmem,
    .max_header         = MAX_TCP_HEADER,
    .obj_size       = sizeof(struct tcp_sock),      /*TCP 套接字由 tcp_sock 结构体表示*/
    .slab_flags         = SLAB_DESTROY_BY_RCU,
    .twsk_prot          = &tcp_timewait_sock_ops,
    .rsk_prot       = &tcp_request_sock_ops,        /*通用连接请求操作结构*/
    .h.hashinfo         = &tcp_hashinfo,        /*指向管理 tcp_sock 实例的数据结构*/
    .no_autobind        = true,
#ifdef CONFIG_COMPAT
    ...
#endif
#ifdef CONFIG_MEMCG_KMEM
    .init_cgroup        = tcp_init_cgroup,
    .destroy_cgroup     = tcp_destroy_cgroup,
    .proto_cgroup       = tcp_proto_cgroup,
#endif
};
```

本节后面小节将介绍 TCP 套接字操作的实现。

12.8.2 套接字初始化

在前面介绍的创建 IPv4 套接字的操作中，将调用套接字关联 proto 实例中的 init()函数对套接字进行初始化。TCP 套接字关联 tcp_prot 实例中，此函数为 **tcp_v4_init_sock()**，定义如下（/net/ipv4/tcp_ipv4.c）：

```
static int tcp_v4_init_sock(struct sock *sk)
{
    struct inet_connection_sock *icsk = inet_csk(sk);       /*指向 inet_connection_sock 实例*/
    tcp_init_sock(sk);      /*初始化 tcp_sock 实例，/net/ipv4/tcp.c*/
    icsk->icsk_af_ops = &ipv4_specific; /*设置 inet_connection_sock_af_ops 实例，/net/ipv4/tcp_ipv4.c*/

#ifdef CONFIG_TCP_MD5SIG
    tcp_sk(sk)->af_specific = &tcp_sock_ipv4_specific;
```

```
#endif
    return 0;
}
```

tcp_v4_init_sock()函数内调用 tcp_init_sock(sk)函数用于初始化 tcp_sock 实例，设置 icsk->icsk_af_ops 成员为 **ipv4_specific**。

1 初始化 tcp_sock

tcp_init_sock(sk)函数用于初始化 tcp_sock 实例，函数定义如下（/net/ipv4/tcp.c）：

```
void tcp_init_sock(struct sock *sk)
{
    struct inet_connection_sock *icsk = inet_csk(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    __skb_queue_head_init(&tp->out_of_order_queue);        /*初始化失序数据包队列头*/
    tcp_init_xmit_timers(sk);        /*初始化发送定时器，/net/ipv4/tcp_timer.c*/
    tcp_prequeue_init(tp);          /*初始化 tp->ucopy.task 为 NULL 等，/include/net/tcp.h*/
    INIT_LIST_HEAD(&tp->tsq_node);

    icsk->icsk_rto = TCP_TIMEOUT_INIT;    /*重传超时时间，（1 秒），/include/net/tcp.h*/
    tp->mdev_us = jiffies_to_usecs(TCP_TIMEOUT_INIT);
    tp->snd_cwnd = TCP_INIT_CWND;       /*初始拥塞窗口（10 个 MSS 值），/include/net/tcp.h*/
    tp->snd_ssthresh = TCP_INFINITE_SSTHRESH;    /*0x7fffffff，/include/net/tcp.h*/
    tp->snd_cwnd_clamp = ~0;
    tp->mss_cache = TCP_MSS_DEFAULT;    /*初始化 MSS 值（536），/include/uapi/linux/tcp.h*/
    u64_stats_init(&tp->syncp);

    tp->reordering = sysctl_tcp_reordering;
    tcp_enable_early_retrans(tp);       /*/include/net/tcp.h*/
    tcp_assign_congestion_control(sk);  /*设置拥塞控制算法（见下一小节），/net/ipv4/tcp_cong.c*/
    tp->tsoffset = 0;
    sk->sk_state = TCP_CLOSE;       /*sock 初始状态为关闭状态*/

    sk->sk_write_space = sk_stream_write_space;
    sock_set_flag(sk, SOCK_USE_WRITE_QUEUE);       /*使用发送队列*/

    icsk->icsk_sync_mss = tcp_sync_mss;

    sk->sk_sndbuf = sysctl_tcp_wmem[1];       /*初始发送窗口长度，16KB*/
    sk->sk_rcvbuf = sysctl_tcp_rmem[1];       /*初始接收窗口长度，87380 字节*/

    local_bh_disable();
    sock_update_memcg(sk);    /*函数定义在/mm/memcontrol.c 或为空操作*/
    sk_sockets_allocated_inc(sk);    /*增加计数值，/include/net/sock.h*/
    local_bh_enable();
}
```

tcp_init_sock()函数对 tcp_sock 实例进行部分初始化，许多的连接参数需要在后面介绍的建立连接操作中设置。下面看一下初始化发送定时器的函数。

■初始化发送定时器

初始化发送定时器的 tcp_init_xmit_timers()函数定义如下（/net/ipv4/tcp_timer.c）：

```
void tcp_init_xmit_timers(struct sock *sk)
{
    inet_csk_init_xmit_timers(sk, &tcp_write_timer, &tcp_delack_timer,&tcp_keepalive_timer);
                                        /*初始化发送定时器，/net/ipv4/inet_connection_sock.c*/
}
```

inet_csk_init_xmit_timers()函数定义如下：

```
void inet_csk_init_xmit_timers(struct sock *sk,
                               void (*retransmit_handler)(unsigned long),      /*重传定时器处理函数*/
                               void (*delack_handler)(unsigned long),          /*延时应答定时器处理函数*/
                               void (*keepalive_handler)(unsigned long))       /*保活定时器处理函数*/
{
    struct inet_connection_sock *icsk = inet_csk(sk);

    setup_timer(&icsk->icsk_retransmit_timer, retransmit_handler,(unsigned long)sk);
    setup_timer(&icsk->icsk_delack_timer, delack_handler,(unsigned long)sk);
    setup_timer(&sk->sk_timer, keepalive_handler, (unsigned long)sk);
    icsk->icsk_pending = icsk->icsk_ack.pending = 0;
}
```

由以上函数可知，这里设置了 3 个定时器超时处理函数，分别是重传定时器、延时应答定时器和保活定时器，处理函数分别为 tcp_write_timer()、tcp_delack_timer()、tcp_keepalive_timer()，这 3 个函数都定义在/net/ipv4/tcp_timer.c 文件内。

2 地址簇操作结构

在初始化 TCP 套接字的 tcp_v4_init_sock()函数中，TCP 套接字关联的 inet_connection_sock_af_ops 实例为 **ipv4_specific**，表示 TCP 套接字特定于网络层协议的操作，实例定义如下（/net/ipv4/tcp_ipv4.c）：

```
const struct inet_connection_sock_af_ops ipv4_specific = {
    .queue_xmit       = ip_queue_xmit,      /*向网络层发送数据包的函数*/
    .send_check       = tcp_v4_send_check,
    .rebuild_header   = inet_sk_rebuild_header,     /**/
    .sk_rx_dst_set    = inet_sk_rx_dst_set,
    .conn_request     = tcp_v4_conn_request,      /*接收连接 SYN 请求，发送 SYN ACK*/
    .syn_recv_sock    = tcp_v4_syn_recv_sock,
                                        /*3 次握手中，收到 ACK 后，创建连接套接字*/
    .net_header_len       = sizeof(struct iphdr),       /*网络层报头长度*/
    .setsockopt       = ip_setsockopt,      /*设置不属于 SOL_TCP 级别的参数*/
    .getsockopt       = ip_getsockopt,      /*获取不属于 SOL_TCP 级别的参数*/
    .addr2sockaddr    = inet_csk_addr2sockaddr,
```

```
    .sockaddr_len      = sizeof(struct sockaddr_in),      /*地址结构长度*/
    .bind_conflict     = inet_csk_bind_conflict,    /*绑定操作中检查地址冲突*/
#ifdef CONFIG_COMPAT
    ...
#endif
    .mtu_reduced       = tcp_v4_mtu_reduced,
};
```

12.8.3 设置/获取参数

用户进程可通过 setsockopt()/getsockopt()系统调用设置/获取套接字参数。setsockopt()/getsockopt()系统调用内将处理 SOL_SOCKET 级别的参数，然后调用 proto_ops 实例中的相应函数处理非 SOL_SOCKET 级别的参数。

1 概述

TCP 套接字操作结构 inet_stream_ops 实例中参数处理函数如下：

```
const struct proto_ops inet_stream_ops = {
    ....
    .setsockopt        = sock_common_setsockopt,        /*设置非 SOL_SOCKET 级别参数*/
    .getsockopt        = sock_common_getsockopt,        /*获取非 SOL_SOCKET 级别参数*/
    ...
}
```

sock_common_setsockopt()/sock_common_getsockopt()函数内将调用 proto 实例中的对应函数，完成参数的处理。

TCP 套接字 proto 实例为 tcp_prot，参数处理函数如下：

```
struct proto tcp_prot = {
    ....
    .setsockopt          = tcp_setsockopt,      /*设置参数*/
    .getsockopt          = tcp_getsockopt,      /*获取参数*/
    ...
}
```

下面以 tcp_setsockopt()函数为例，说明其实现，函数代码如下（/net/ipv4/tcp.c）：

```
int tcp_setsockopt(struct sock *sk, int level, int optname, char __user *optval,unsigned int optlen)
{
    const struct inet_connection_sock *icsk = inet_csk(sk);

    if (level != SOL_TCP)            /*处理非 SOL_TCP 级别参数*/
        return icsk->icsk_af_ops->setsockopt(sk, level, optname,optval, optlen);
    return do_tcp_setsockopt(sk, level, optname, optval, optlen);     /*处理 SOL_TCP 级别参数*/
}
```

tcp_setsockopt()函数中调用地址簇操作结构 inet_connection_sock_af_ops 中的 setsockopt()函数设置非 SOL_TCP 级别的参数，调用 **do_tcp_setsockopt()**函数处理 SOL_TCP 级别参数。

SOL_TCP 级别参数名称定义如下（/include/uapi/linux/tcp.h）：

```
#define    TCP_NODELAY          1     /* Turn off Nagle's algorithm.  关闭 Nagle 算法*/
```

```
#define   TCP_MAXSEG          2    /* Limit MSS */
#define   TCP_CORK            3    /* Never send partially complete segments，抑制功能*/
#define   TCP_KEEPIDLE        4    /* Start keeplives after this period */
#define   TCP_KEEPINTVL       5    /* Interval between keepalives */
#define   TCP_KEEPCNT         6    /* Number of keepalives before death */
#define   TCP_SYNCNT          7    /* Number of SYN retransmits，连接操作中 SYN 重发次数*/
#define   TCP_LINGER2         8    /* Life time of orphaned FIN-WAIT-2 state */
#define   TCP_DEFER_ACCEPT 9    /* Wake up listener only when data arrive */
#define   TCP_WINDOW_CLAMP    10     /* Bound advertised window */
#define   TCP_INFO            11     /* Information about this connection.  连接信息*/
#define   TCP_QUICKACK        12   /* Block/reenable quick acks */
#define   TCP_CONGESTION      13   /* Congestion control algorithm，拥塞控制算法*/
#define   TCP_MD5SIG          14   /* TCP MD5 Signature (RFC2385) */
#define   TCP_THIN_LINEAR_TIMEOUTS     16         /* Use linear timeouts for thin streams*/
#define   TCP_THIN_DUPACK              17         /* Fast retrans. after 1 dupack */
#define   TCP_USER_TIMEOUT      18     /* How long for loss retry before timeout */
#define   TCP_REPAIR            19     /* TCP sock is under repair right now */
#define   TCP_REPAIR_QUEUE      20
#define   TCP_QUEUE_SEQ         21
#define   TCP_REPAIR_OPTIONS    22
#define   TCP_FASTOPEN          23  /* Enable FastOpen on listeners，监听套接字支持快速打开*/
#define   TCP_TIMESTAMP         24
#define   TCP_NOTSENT_LOWAT     25  /* limit number of unsent bytes in write queue */
#define   TCP_CC_INFO           26  /* Get Congestion Control (optional) info */
#define   TCP_SAVE_SYN          27  /* Record SYN headers for new connections */
#define   TCP_SAVED_SYN         28  /* Get SYN headers recorded for connection */
```

处理 SOL_TCP 级别参数的 **do_tcp_setsockopt()**函数定义如下（/net/ipv4/tcp.c）：

```
static int do_tcp_setsockopt(struct sock *sk, int level,int optname, char __user *optval, unsigned int optlen)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct inet_connection_sock *icsk = inet_csk(sk);
    int val;        /*参数值，用户传递的*/
    int err = 0;
    switch (optname) {      /*设置拥塞控制算法*/
    case TCP_CONGESTION: {
        char name[TCP_CA_NAME_MAX];      /*拥塞控制算法名称（字符串），见下文*/
        ...
        val = strncpy_from_user(name, optval,min_t(long, TCP_CA_NAME_MAX-1, optlen));
        ...
        name[val] = 0;

        lock_sock(sk);
        err = tcp_set_congestion_control(sk, name);   /*设置塞控制算法，/net/ipv4/tcp_cong.c*/
        release_sock(sk);
        return err;
```

```
        }
        default:
            break;
        }

        /*以下是处理其它参数*/
        ...
        if (get_user(val, (int __user *)optval))       /*复制用户参数值*/
            return -EFAULT;

        lock_sock(sk);
        switch (optname) {     /*设置参数，其它参数值为整型数*/
        case TCP_MAXSEG:
            ...
            tp->rx_opt.user_mss = val;
            break;

        case TCP_NODELAY:
            if (val) {
                tp->nonagle |= TCP_NAGLE_OFF|TCP_NAGLE_PUSH;
                tcp_push_pending_frames(sk);        /*将发送缓存队列中数据包发送出去*/
            } else {
                tp->nonagle &= ~TCP_NAGLE_OFF;
            }
            break;
            ...
        }

        release_sock(sk);
        return err;
    }
```

TCP 参数中，除了拥塞控制算法，其它参数值都是一个整数，主要是对 TCP 参数进行设置，源代码请读者自行阅读，下面主要介绍拥塞控制算法的设置。

获取参数的 getsockopt()系统调用与 setsockopt()类似，源代码请读者自行阅读。

2 拥塞控制算法

前面介绍过，TCP 拥塞控制算法由 tcp_congestion_ops 结构体表示，TCP 拥塞控制算法公共层代码在/net/ipv4/tcp_cong.c 文件内实现。

■算法管理

内核实现了许多的拥塞控制算法，供用户 TCP 选择，各拥塞控制算法实现代码位于/net/ipv4/目录下，例如：tcp_bic.c，tcp_cdg.c，**tcp_cubic.c**，tcp_dctcp.c，tcp_westwood.c，tcp_highspeed.c，tcp_hybla.c，tcp_htcp.c，tcp_vegas.c，tcp_veno.c，tcp_scalable.c，tcp_lp.c，tcp_yeah.c，tcp_illinois.c。

内核在 tcp_cong.c 文件内定义了 tcp_reno 算法，并在 tcp_init()函数内注册。

如果用户选择了 TCP_CONG_ADVANCED 配置选项，用户可以通过配置选项选择默认的拥塞控制算法，如果不设置，内核默认使用 CUBIC 算法。如果没有选择 TCP_CONG_ADVANCED 配置选项，则使用在 tcp_init()函数中注册的 tcp_reno 算法。

内核定义了全局双链表 **tcp_cong_list** 管理表示拥塞控制算法的 tcp_congestion_ops 实例。实现拥塞算法的代码需要定义 tcp_congestion_ops 实例，并向内核注册。注册函数为 tcp_register_congestion_control()，代码如下（/net/ipv4/tcp_cong.c）：

```
int tcp_register_congestion_control(struct tcp_congestion_ops *ca)
{
    int ret = 0;
    if (!ca->ssthresh || !ca->cong_avoid) {
        ...
    }
    ca->key = jhash(ca->name, sizeof(ca->name), strlen(ca->name));      /*计算键值*/

    spin_lock(&tcp_cong_list_lock);
    if (ca->key == TCP_CA_UNSPEC || tcp_ca_find_key(ca->key)) {
        ...
    } else {
        list_add_tail_rcu(&ca->list, &tcp_cong_list);      /*将实例添加到 tcp_cong_list 双链表*/
        pr_debug("%s registered\n", ca->name);
    }
    spin_unlock(&tcp_cong_list_lock);
    return ret;
}
```

■设置算法

tcp_congestion_default()函数用于设置默认的拥塞控制算法，函数定义如下（/net/ipv4/tcp_cong.c）：

```
static int __init tcp_congestion_default(void)
{
    return tcp_set_default_congestion_control(CONFIG_DEFAULT_TCP_CONG);
}
late_initcall(tcp_congestion_default);    /*内核启动阶段调用*/
```

用户通过配置选项可设置默认的拥塞控制算法。CONFIG_DEFAULT_TCP_CONG 宏表示默认拥塞控制算法的名称，tcp_set_default_congestion_control()函数按名称查找 tcp_congestion_ops 实例，并将实例移动到全局双链表 tcp_cong_list 的头部，源代码请读者自行阅读。

在前面套接字初始化函数 tcp_v4_init_sock()中调用 **tcp_assign_congestion_control(sk)**函数设置套接字的初始拥塞控制算法，函数内获取 tcp_cong_list 双链表中的**第一个**实例赋予 icsk->icsk_ca_ops 成员。

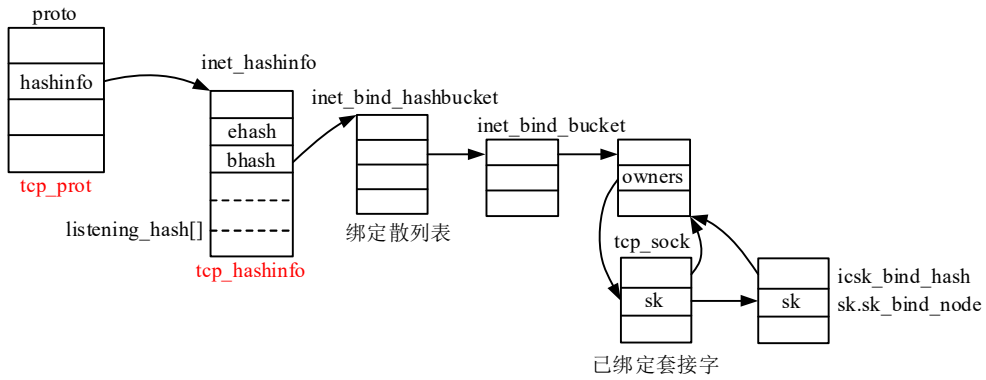
在设置参数的 do_tcp_setsockopt()函数中调用 **tcp_set_congestion_control(sk, name)**函数设置套接字的拥塞控制算法，函数内在 tcp_cong_list 双链表中根据名称查找 tcp_congestion_ops 实例，赋予套接字，并调用实例中的初始化函数 init()。

12.8.4 绑定

IPv4 地址簇地址值由 sockaddr_in 结构体表示，其中主要包含 IP 地址和端口号。bind()系统调用用于给

套接字绑定地址值，此系统调用内将调用 inet_stream_ops 实例中的 **inet_bind()**函数。

inet_bind()函数在 UDP 套接字的绑定操作中介绍过了，函数内将 IP 地址赋予套接字，调用套接字关联的 proto 实例中的 get_port()函数，检查地址结构中传递的端口号是否可用，如果可用则将端口号赋予套接字，并将套接字实例插入管理结构中。tcp_prot 实例中 get_port()函数为 **inet_csk_get_port()**，函数内将为套接字设置或分配端口号，并通过 inet_bind_bucket 实例将 tcp_sock 实例插入到绑定散列表，依网络命名空间和端口号计算列值，如下图所示。



tcp_hashinfo 实例中 bhash 散列表用于管理已绑定的套接字，散列表中对象为 inet_bind_bucket 实例，实例中 owners 链表管理的是相同端口号的 tcp_sock 实例。

1 设置端口号

tcp_prot 实例中的 get_port()函数为 inet_csk_get_port()，在绑定操作中被调用，用于设置套接字端口号，并将 tcp_sock 实例添加到绑定散列表中，函数定义如下（/net/ipv4/inet_connection_sock.c）：

```
int inet_csk_get_port(struct sock *sk, unsigned short snum)
/*sk：套接字 sock 实例，snum：端口号*/
{
    struct inet_hashinfo *hashinfo = sk->sk_prot->h.hashinfo;      /*TCP 套接字管理结构*/
    struct inet_bind_hashbucket *head;
    struct inet_bind_bucket *tb;      /*bhash 散列表中管理的实例*/
    int ret, attempts = 5;
    struct net *net = sock_net(sk);      /*网络命名空间*/
    int smallest_size = -1, smallest_rover;
    kuid_t uid = sock_i_uid(sk);     /*用户 ID 值*/
    int attempt_half = (sk->sk_reuse == SK_CAN_REUSE) ? 1 : 0;

    local_bh_disable();
    if (!snum) {      /*端口号为 0，需由内核分配端口号*/
        int remaining, rover, low, high;

again:
        inet_get_local_port_range(net, &low, &high);     /*可分配端口号范围*/
        if (attempt_half) {
            int half = low + ((high - low) >> 1);
            if (attempt_half == 1)
                high = half;
            else
```

```
                low = half;
        }
        remaining = (high - low) + 1;
        smallest_rover = rover = prandom_u32() % remaining + low;       /*分配的端口号*/

        smallest_size = -1;
        do {
            if (inet_is_local_reserved_port(net, rover))
                goto next_nolock;
            head = &hashinfo->bhash[inet_bhashfn(net, rover,hashinfo->bhash_size)];   /*散列链表头*/
            spin_lock(&head->lock);
            inet_bind_bucket_for_each(tb, &head->chain)
                            /*遍历链表中 inet_bind_bucket 实例，检查分配端口号是否可用*/
                if (net_eq(ib_net(tb), net) && tb->port == rover) {     /*命名空间和端口号都相同*/
                    if (((tb->fastreuse > 0 &&sk->sk_reuse &&sk->sk_state != TCP_LISTEN) ||
                         (tb->fastreuseport > 0 &&sk->sk_reuseport &&uid_eq(tb->fastuid, uid))) &&
                        (tb->num_owners < smallest_size || smallest_size == -1)) {
                        smallest_size = tb->num_owners;
                        smallest_rover = rover;
                    }
                    if (!inet_csk(sk)->icsk_af_ops->bind_conflict(sk, tb, false)) {
                        snum = rover;                   /*检查端口号是否有冲突*/
                        goto tb_found;
                    }
                    goto next;
                }
            break;
        next:
            spin_unlock(&head->lock);
        next_nolock:
            if (++rover > high)
                rover = low;
        } while (--remaining > 0);       /*分配端口号结束，rover 表示分配的端口号*/

        ret = 1;
        if (remaining <= 0) {       /*耗尽了可分配的端口号*/
          ...
        }
        snum = rover;         /*端口号*/
        /*内核分配端口号结束*/
    } else {         /*如果 sunm 不为 0，地址参数中指定了端口号*/
have_snum:         /*散列链表头，由网络命名空间和端口号计算散列值*/
        head = &hashinfo->bhash[inet_bhashfn(net, snum,hashinfo->bhash_size)];
        spin_lock(&head->lock);
        inet_bind_bucket_for_each(tb, &head->chain)       /*检查端口号是否可用*/
            if (net_eq(ib_net(tb), net) && tb->port == snum)       /*存在相同端口号*/
```

```
                goto tb_found;
        }
        /*运行至此表示端口号可用，并不存在相同端口号的套接字*/
        tb = NULL;
        goto tb_not_found;      /*跳转到 tb_not_found*/

tb_found:       /*已存在相同的端口号*/
        if (!hlist_empty(&tb->owners)) {        /*tb->owners 链表不为空，链接 tcp_sock 实例*/
                if (sk->sk_reuse == SK_FORCE_REUSE)
                        goto success;

                if (((tb->fastreuse > 0 &&sk->sk_reuse && sk->sk_state != TCP_LISTEN) ||
                     (tb->fastreuseport > 0 &&sk->sk_reuseport && uid_eq(tb->fastuid, uid))) &&
                     smallest_size == -1) {
                        goto success;
                } else {
                        ret = 1;
                        if (inet_csk(sk)->icsk_af_ops->bind_conflict(sk, tb, true)) {
                                if (((sk->sk_reuse && sk->sk_state != TCP_LISTEN) ||
                                     (tb->fastreuseport > 0 &&sk->sk_reuseport && uid_eq(tb->fastuid, uid))) &&
                                        smallest_size != -1 && --attempts >= 0) {
                                        spin_unlock(&head->lock);
                                        goto again;
                                }
                                goto fail_unlock;
                        }
                }
        }       /*if (!hlist_empty(&tb->owners))结束*/

tb_not_found:       /*不存在相同的端口号*/
        ret = 1;
        if (!tb && (tb = inet_bind_bucket_create(hashinfo->bind_bucket_cachep,net, head, snum)) == NULL)
                goto fail_unlock;                               /*分配 inet_bind_bucket 实例，并插入散列表*/
        if (hlist_empty(&tb->owners)) {     /*tb->owners 链表为空，设置 tb 实例*/
                if (sk->sk_reuse && sk->sk_state != TCP_LISTEN)
                        tb->fastreuse = 1;
                else
                        tb->fastreuse = 0;
                if (sk->sk_reuseport) {
                        tb->fastreuseport = 1;
                        tb->fastuid = uid;
                } else
                        tb->fastreuseport = 0;
        } else {        /*tb->owners 链表不为空*/
                if (tb->fastreuse &&(!sk->sk_reuse || sk->sk_state == TCP_LISTEN))
                        tb->fastreuse = 0;
```

```
            if (tb->fastreuseport &&(!sk->sk_reuseport || !uid_eq(tb->fastuid, uid)))
                tb->fastreuseport = 0;
        }
    success:
        if (!inet_csk(sk)->icsk_bind_hash)     /*icsk_bind_hash 为空，则关联新创建 inet_bind_bucket 实例*/
            inet_bind_hash(sk, tb, snum);      /*/net/ipv4/inet_hashtables.c*/
        WARN_ON(inet_csk(sk)->icsk_bind_hash != tb);
        ret = 0;
    fail_unlock:
        spin_unlock(&head->lock);
    fail:
        local_bh_enable();
        return ret;
    }
```

inet_csk_get_port()函数的主要工作是为套接字分配或设置套接字端口号，并检查端口号是否可用，如果可用则为套接字查找/分配 inet_bind_bucket 实例（新实例插入到绑定散列表），将 tcp_sock 实例关联到 inet_bind_bucket 实例，添加到其 owners 链表。

2 插入绑定散列表

inet_csk_get_port()函数内如果端口号未被使用，将调用 inet_bind_bucket_create()函数为端口号创建 inet_bind_bucket 实例，并插入到 bhash 散列表，然后调用 inet_bind_hash()建立 inet_bind_bucket 实例与套接字 tcp_sock 实例之间的关联。

inet_bind_bucket_create()函数定义如下（/net/ipv4/inet_hashtables.c）：

```
struct inet_bind_bucket *inet_bind_bucket_create(struct kmem_cache *cachep,struct net *net,
                                                 struct inet_bind_hashbucket *head,const unsigned short snum)
/*cachep：slab 缓存，net：网络命名空间，head：bhash 散列表头，snum：端口号*/
{
    struct inet_bind_bucket *tb = kmem_cache_alloc(cachep, GFP_ATOMIC);       /*分配实例*/

    if (tb) {            /*初始化 inet_bind_bucket 实例*/
        write_pnet(&tb->ib_net, net);
        tb->port       = snum;       /*端口号*/
        tb->fastreuse = 0;
        tb->fastreuseport = 0;
        tb->num_owners = 0;
        INIT_HLIST_HEAD(&tb->owners);
        hlist_add_head(&tb->node, &head->chain);          /*添加到 bhash 散列表*/
    }
    return tb;     /*返回 inet_bind_bucket 实例指针*/
}
```

inet_bind_hash()函数用于建立 inet_bind_bucket 实例与 tcp_sock 实例之间的关联，函数定义如下：

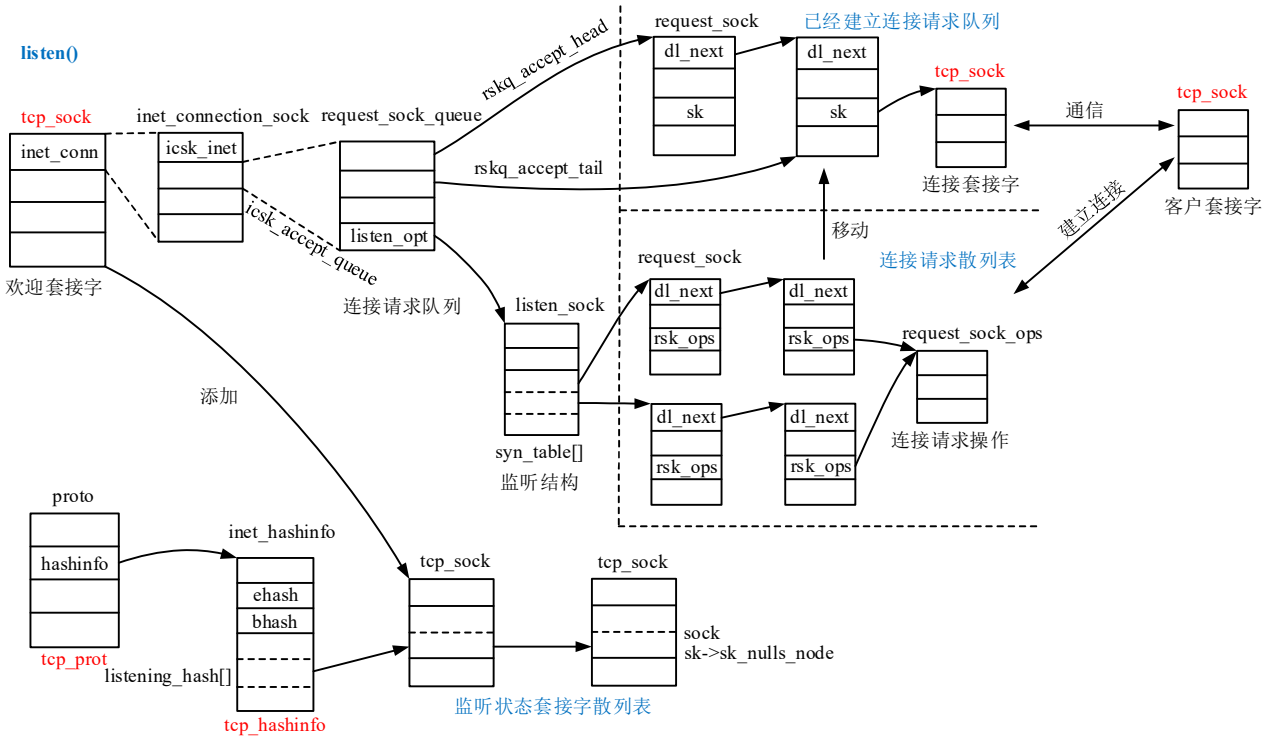
```
void inet_bind_hash(struct sock *sk, struct inet_bind_bucket *tb,const unsigned short snum)
{
    inet_sk(sk)->inet_num = snum;       /*设置端口号*/
```

```
    sk_add_bind_node(sk, &tb->owners);
                /*将 tcp_sock 实例添加到 tb->owners 链表，sk->sk_bind_node*/
    tb->num_owners++;       /*套接字计数加 1*/
    inet_csk(sk)->icsk_bind_hash = tb;     /*指向 inet_bind_bucket 实例*/
}
```

12.8.5 监听连接请求

服务器进程（被动连接套接字）在执行完套接字的创建和地址设置后，接下来执行 listen()系统调用监听客户进程（主动连接套接字）的连接请求。

服务器进程首先创建的是欢迎套接字。欢迎套接字通过 listen()系统调用为其创建监听结构 listen_sock 实例，将套接字设为 TCP_LISTEN 状态，并将套接字添加到 tcp_prot 实例中的监听状态套接字散列表中，如下图所示。



inet_connection_sock 结构体中 icsk_accept_queue 成员（request_sock_queue 结构体实例）表示连接请求队列，其中包含指向监听结构的指针成员和已建立连接的请求队列。连接请求由 request_sock 结构体表示。

欢迎套接字执行完 listen()系统调用后，就可以监听客户套接字的连接请求了。客户套接字通过连接系统调用 connect()向欢迎套接字发送连接请求，欢迎套接字接收到请求后将为其创建连接请求 request_sock 实例，插入到连接请求散列表，并向客户套接字发送请求应答。客户套接字接收到请求应答后，将向欢迎套接字发送应答。欢迎套接字收到应答后，将为连接请求创建并初始化连接套接字，并将连接请求移动到已建立连接请求队列。此后，客户套接字与服务器端连接套接字通信。这就是 TCP 连接中的 3 次握手，后面将详细介绍。

服务器进程执行 accept()系统调用，将创建套接字 socket 实例，分配文件描述符，从已建立连接请求队列中取出第一个请求实例。然后，将 socket 实例关联到连接请求中的连接套接字，释放连接请求实例。此后，服务器进程可通过连接套接字（分配了文件描述符）与客户套接字通信，后面将详细介绍 accept()系统调用的实现。

1 数据结构

在介绍 listen()系统调用实现前，先介绍连接相关数据结构的定义（/include/net/request_sock.h）。

（1）连接请求队列

面向连接的套接字由 inet_connection_sock 结构体表示，其中 icsk_accept_queue 为 **request_sock_queue** 结构体实例成员，用于管理连接请求，结构体定义如下：

```
struct request_sock_queue {
    struct request_sock *rskq_accept_head;      /*已建立连接的请求队列头*/
    struct request_sock *rskq_accept_tail;      /*已建立连接的请求队列尾*/
    u8              rskq_defer_accept;          /*默认重新 SYN 的数量*/
    struct listen_sock    *listen_opt;          /*指向监听结构*/
    struct fastopen_queue    *fastopenq;      /*/include/net/request_sock.h*/*/
                                /*由系统控制参数 sysctl_tcp_fastopen 控制，默认不开启*/
    spinlock_t          syn_wait_lock ____cacheline_aligned_in_smp;      /*同步锁*/
};
```

request_sock_queue 结构体中主要包含监听结构指针 listen_opt 成员和已建立连接请求队列。

（2）监听结构

监听结构由 listen_sock 结构体表示，在 listen()系统调用中将为欢迎套接字创建 listen_sock 实例，结构体定义如下：

```
struct listen_sock {
    int             qlen_inc;       /*散列表中连接请求数量*/
    int             young_inc;      /**/

    atomic_t        qlen_dec;    /* qlen = qlen_inc - qlen_dec */
    atomic_t        young_dec;

    u8              max_qlen_log    ____cacheline_aligned_in_smp;
                                            /*nr_table_entries 成员值，以 2 的幂次表示*/
    u8              synflood_warned;
    /* 2 bytes hole, try to use */
    u32             hash_rnd;       /*用于计算连接请求的散列值*/
    u32             nr_table_entries;     /*syn_table[]散列表的项数*/
    struct request_sock       *syn_table[0];        /*连接请求散列表，在创建 listen_sock 实例时分配*/
};
```

listen_sock 结构体中最后是一个散列表，管理主动连接套接字发送的连接请求。由目的套接字 IP 地址、端口号、listen_sock 结构体中 hash_rnd、nr_table_entries 成员值计算得散列值。内核在创建 listen_sock 实例时，同时为散列表分配空间。

系统控制参数 sysctl_max_syn_backlog 控制了散列表的最大项数（默认值为 256），最小值为 8。

（3）通用连接请求

连接请求由主动连接套接字发起，被动连接套接字收到请求后，创建由 request_sock 结构体表示的通用连接请求，并添加到连接请求散列表。request_sock 结构体定义如下：

```
struct request_sock {
    struct sock_common          __req_common;
  #define rsk_refcnt            __req_common.skc_refcnt
  #define rsk_hash              __req_common.skc_hash       /*在连接请求散列表中的散列值*/

    struct request_sock       *dl_next;       /*指向下一个连接请求（用于散列链表和队列）*/
```

```
    struct sock             *rsk_listener;    /*指向监听（欢迎）套接字*/
    u16     mss;            /*数据包中用户数据最大长度，不含传输层、网络层、数据链路层报头*/
    u8      num_retrans;     /* number of retransmits */
    u8      cookie_ts:1;     /* syncookie: encode tcpopts in timestamp */
    u8      num_timeout:7;   /* number of timeouts */

    u32     window_clamp;   /* window clamp at creation time */
    u32     rcv_wnd;        /*初始接收窗口长度*/
    u32     ts_recent;
    struct timer_list       rsk_timer;      /*定时器*/
    const struct request_sock_ops *rsk_ops;     /*连接请求操作结构*/
    struct sock     *sk;        /*连接套接字 sock 实例*/
    u32     *saved_syn;
    u32     secid;
    u32     peer_secid;
};
```

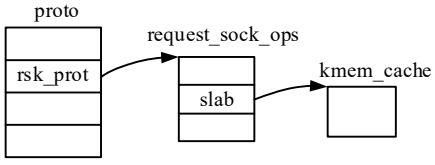
request_sock 表示通用的连接请求。在 TCP 中，连接请求由 tcp_request_sock 结构体表示，结构体中嵌入 request_sock 结构体成员，见下文。

（4）通用连接请求操作

通用连接请求操作结构定义如下：

```
struct request_sock_ops {
    int     family;     /*地址簇标识*/
    int     obj_size; /*传输层协议中表示连接请求数据结构大小，如 tcp_request_sock 结构体大小*/
    struct kmem_cache *slab;   /*连接请求数据结构 slab 缓存*/
    char    *slab_name;
    int     (*rtx_syn_ack)(struct sock *sk,struct request_sock *req);
    void    (*send_ack)(struct sock *sk, struct sk_buff *skb,struct request_sock *req);
                                                        /*发送请求应答*/
    void    (*send_reset)(struct sock *sk,struct sk_buff *skb);    /*发送复位信息*/
    void    (*destructor)(struct request_sock *req);
    void    (*syn_ack_timeout)(const struct request_sock *req);
};
```

在传输层协议关联的 proto 结构体中 **rsk_prot** 成员指向通用连接请求操作结构实例，在注册 proto 实例的 proto_register()函数中将调用 req_prot_init(prot)函数为 request_sock_ops->slab 成员创建 slab 缓存，如下图所示：



2 监听函数

监听连接请求 listen()系统调用原型如下所示：

```
listen(int fd, int backlog)
```

fd 表示套接字的文件描述符，backlog 表示连接请求散列表项数。

listen()系统调用（/net/socket.c）中将调用套接字 proto_ops 实例中的 listen()函数，对于 TCP 套接字此函数为 **inet_listen()**，函数定义如下（/net/ipv4/af_inet.c）：

```
int inet_listen(struct socket *sock, int backlog)
/*sock：服务器欢迎套接字，backlog：连接请求散列表项数*/
{
    struct sock *sk = sock->sk;
    unsigned char old_state;
    int err;

    lock_sock(sk);
    err = -EINVAL;
    if (sock->state != SS_UNCONNECTED || sock->type != SOCK_STREAM)
        goto out;                    /*套接字必须是未连接状态，且是流套接字*/

    old_state = sk->sk_state;       /*套接字原状态*/
    if (!((1 << old_state) & (TCPF_CLOSE | TCPF_LISTEN)))
        goto out;

    if (old_state != TCP_LISTEN) {
        if ((sysctl_tcp_fastopen & TFO_SERVER_ENABLE) != 0 &&
            !inet_csk(sk)->icsk_accept_queue.fastopenq) {
                                /*系统控制参数 sysctl_tcp_fastopen 默认关闭 fastopen*/
            if ((sysctl_tcp_fastopen & TFO_SERVER_WO_SOCKOPT1) != 0)
                err = fastopen_init_queue(sk, backlog);
            else if ((sysctl_tcp_fastopen &TFO_SERVER_WO_SOCKOPT2) != 0)
                err = fastopen_init_queue(sk,((uint)sysctl_tcp_fastopen) >> 16);
            else
                err = 0;
            if (err)
                goto out;

            tcp_fastopen_init_key_once(true);
        }
        err = inet_csk_listen_start(sk, backlog);     /*建立监听环境，/net/ipv4/inet_connection_sock.c*/
        if (err)
            goto out;
    }
    sk->sk_max_ack_backlog = backlog;     /*散列表项数*/
    err = 0;
out:
    release_sock(sk);
    return err;       /*成功返回 0*/
}
```

inet_listen()函数主要工作是调用 inet_csk_listen_start()函数为套接字创建 listen_sock 实例，设置套接字

状态为 **TCP_LISTEN**，并将套接字插入到 tcp_prot 实例中的监听连接套接字散列表中。

inet_csk_listen_start()函数定义如下（/net/ipv4/inet_connection_sock.c）：

```
int inet_csk_listen_start(struct sock *sk, const int nr_table_entries)
/*nr_table_entries：监听连接请求数量*/
{
    struct inet_sock *inet = inet_sk(sk);
    struct inet_connection_sock *icsk = inet_csk(sk);
    int rc = reqsk_queue_alloc(&icsk->icsk_accept_queue, nr_table_entries); /*/net/core/request_sock.c*/
                                        /*分配 listen_sock 实例，后附散列表*/

    if (rc != 0)
        return rc;

    sk->sk_max_ack_backlog = 0;
    sk->sk_ack_backlog = 0;
    inet_csk_delack_init(sk);

    sk->sk_state = TCP_LISTEN;      /*设置套接字为监听状态*/
    if (!sk->sk_prot->get_port(sk, inet->inet_num)) {    /*是否绑定了端口*/
        inet->inet_sport = htons(inet->inet_num);
        sk_dst_reset(sk);
        sk->sk_prot->hash(sk);      /*将套接字插入监听连接套接字散列表，inet_hash()*/
        return 0;
    }

    /*运行到这里表示出错了*/
    sk->sk_state = TCP_CLOSE;
    __reqsk_queue_destroy(&icsk->icsk_accept_queue);
    return -EADDRINUSE;
}
```

inet_csk_listen_start()函数首先调用 reqsk_queue_alloc()函数，为套接字分配 listen_sock 实例，后附连接请求散列表。然后，将套接字状态设为 **TCP_LISTEN**（监听状态），调用 tcp_prot 实例中 get_port()函数，确定套接字绑定了端口号。最后，调用 tcp_prot 实例中的 inet_hash()函数，对不是 TCP_CLOSE 状态的套接字，判断其状态是否为 TCP_LISTEN，如果不是则将套接字添加到已建立连接套接字散列表，如果是 TCP_LISTEN 状态，则将 tcp_sock 实例插入到 tcp_prot.hashinfo->listening_hash[]散列表（监听散列表），由网络命名空间和端口号计算得散列值。reqsk_queue_alloc()和 inet_hash()函数都比较简单，源代码请读者自行阅读。

在 listen()系统调用中只为套接字创建了监听结构 listen_sock 实例，而没有创建连接请求 request_sock 实例。在主动连接套接字发起连接请求时，在被动连接套接字（监听连接请求的套接字）内将创建连接请求 request_sock 实例，并添加到连接请求散列表。当连接建立后，将为连接请求创建连接套接字，并将请求移动到已建立连接请求队列中。主动连接套接字只需要发送连接请求，连接的建立由协议完成，不需要用户的操作。

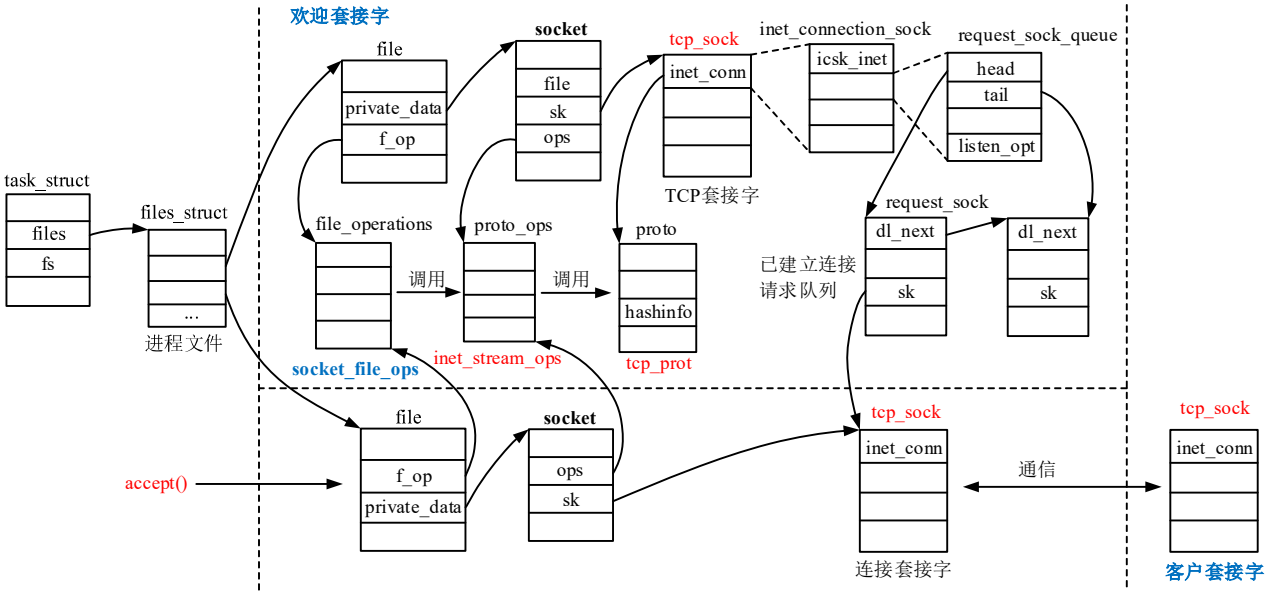
12.8.6 接受连接请求

当连接建立后（连接操作见下文），连接请求将移动到已建立连接请求队列，并为连接请求创建了连接套接字。

服务器进程（被动连接套接字）调用 accept()系统调用接受客户进程（主动连接套接字）的连接请求（已建立连接的请求）。accept()系统调用内将创建一个新的套接字 socket 实例，为其分配文件描述符，从已建立连接请求队列头部取出一个请求，将其连接套接字关联到新创建的 socket 实例。此后，服务器进程通过此（连接）套接字与客户套接字通信。

1 系统调用

服务器进程通过 accept()系统调用接受一个已建立连接的请求。系统调用内将为连接请求创建套接字 socket 实例，分配文件描述符，从已建立连接请求队列中取出一个 request_sock 实例（如果队列为空则睡眠等待），将新创建的 socket 实例关联到连接请求中的 tcp_sock 实例（连接套接字），释放此连接请求实例，返回连接套接字的文件描述符。



accept()系统调用定义如下（/net/socket.c）：

```
SYSCALL_DEFINE3(accept, int, fd, struct sockaddr __user *, upeer_sockaddr,int __user *, upeer_addrlen)
{
    return sys_accept4(fd, upeer_sockaddr, upeer_addrlen, 0);      /*返回连接套接字文件描述符*/
}
```

accept()系统调用内调用 accept4()系统调用的实现函数，定义如下：

```
SYSCALL_DEFINE4(accept4, int, fd, struct sockaddr __user *, upeer_sockaddr,
                                                    int __user *, upeer_addrlen, int, flags)
/*fd：欢迎套接字文件描述符，upeer_sockaddr：用户地址结构指针，upeer_addrlen：地址长度，
*flags：0。
*/
{
    struct socket *sock, *newsock;
    struct file *newfile;
    int err, len, newfd, fput_needed;
```

```
    struct sockaddr_storage address;

    if (flags & ~(SOCK_CLOEXEC | SOCK_NONBLOCK))
        return -EINVAL;

    if (SOCK_NONBLOCK != O_NONBLOCK && (flags & SOCK_NONBLOCK))
        flags = (flags & ~SOCK_NONBLOCK) | O_NONBLOCK;

    sock = sockfd_lookup_light(fd, &err, &fput_needed);     /*由文件描述符获取 socket 实例*/
    ...     /*错误处理*/

    err = -ENFILE;
    newsock = sock_alloc();     /*分配新 socket 实例（连接套接字），/net/socket.c*/
    ...     /*错误处理*/

    newsock->type = sock->type;       /*套接字类型*/
    newsock->ops = sock->ops;         /*套接字操作结构实例*/

    __module_get(newsock->ops->owner);

    newfd = get_unused_fd_flags(flags);         /*获取新文件描述符*/
    ...     /*错误处理*/
    newfile = sock_alloc_file(newsock, flags, sock->sk->sk_prot_creator->name);     /*分配 file 实例*/
    ...     /*错误处理*/
    err = security_socket_accept(sock, newsock);
    ...     /*错误处理*/
    err = sock->ops->accept(sock, newsock, sock->file->f_flags);     /*处理连接请求*/
                                                /*调用 proto_ops 实例中的 accept()函数*/
    ...     /*错误处理*/
    if (upeer_sockaddr) {       /*客户进程（主动连接套接字）地址返回给用户*/
        if (newsock->ops->getname(newsock, (struct sockaddr *)&address,&len, 2) < 0) {
            ...     /*错误处理*/
        }
        err = move_addr_to_user(&address,len, upeer_sockaddr, upeer_addrlen);
        ...     /*错误处理*/
    }

    fd_install(newfd, newfile);       /*文件描述符绑定 file 实例*/
    err = newfd;       /*文件描述符*/

out_put:
    fput_light(sock->file, fput_needed);
out:
    return err;     /*返回连接套接字描述符*/
    ...
}
```

accept()系统调用在为连接请求创建了 socket 实例和文件 file 实例后，将调用 proto_ops 实例中的 accept()函数处理连接请求，最后返回连接套接字的文件描述符。

2 处理连接请求

TCP 套接字关联 proto_ops 实例中 accept()函数为 inet_accept()，定义如下（/net/ipv4/af_inet.c）：

```
int inet_accept(struct socket *sock, struct socket *newsock, int flags)
/*sock：欢迎套接字，newsock：新创建的连接套接字 socket 实例，flags：0*/
{
    struct sock *sk1 = sock->sk;
    int err = -EINVAL;
    struct sock *sk2 = sk1->sk_prot->accept(sk1, flags, &err);    /*调用 proto 实例中的 accept()函数*/
                                /*从已建立连接请求队列中获取一个请求，返回其关联的 sock 实例等*/
    if (!sk2)
        goto do_err;
    lock_sock(sk2);
    sock_rps_record_flow(sk2);
    WARN_ON(!((1 << sk2->sk_state) &
            (TCPF_ESTABLISHED | TCPF_SYN_RECV |
            TCPF_CLOSE_WAIT | TCPF_CLOSE)));

    sock_graft(sk2, newsock);      /*newsock->sk=sk2，新 socket 实例关联 sock 实例*/

    newsock->state = SS_CONNECTED;      /*设置新套接字（socket）状态为已连接状态*/
    err = 0;
    release_sock(sk2);
do_err:
    return err;
}
```

inet_accept()函数内将调用套接字关联 proto 实例中的 accept()函数，从已建立连接请求队列中获取一个请求实例（如果队列为空则进入睡眠等待），返回连接请求关联的 sock 实例，最后释放连接请求实例，表示连接已完成，连接请求可以删除了。

对于 TCP 套接字，其关联的 proto 实例中的 accept()函数为 **inet_csk_accept()**，定义如下：

```
struct sock *inet_csk_accept(struct sock *sk, int flags, int *err)    /*/net/ipv4/inet_connection_sock.c*/
/*sk：欢迎套接字，flags：0*/
{
    struct inet_connection_sock *icsk = inet_csk(sk);
    struct request_sock_queue *queue = &icsk->icsk_accept_queue;      /*连接请求管理结构*/
    struct request_sock *req;
    struct sock *newsk;      /*连接请求关联的 sock 实例*/
    int error;

    lock_sock(sk);
    error = -EINVAL;
    if (sk->sk_state != TCP_LISTEN)      /*欢迎套接字必须为监听状态*/
```

```
            goto out_err;

        if (reqsk_queue_empty(queue)) {       /*已建立连接请求队列为空*/
            long timeo = sock_rcvtimeo(sk, flags & O_NONBLOCK);       /*等待超时时间值*/

            error = -EAGAIN;
            if (!timeo)
                goto out_err;

            error = inet_csk_wait_for_connect(sk, timeo);
                                        /*睡眠等待，已建立连接请求队列不为空或超时时唤醒*/
            ...     /*错误处理*/
        }
        req = reqsk_queue_remove(queue);     /*取出已建立连接请求队列中第一个连接请求*/
        newsk = req->sk;         /*连接请求关联的 sock 实例*/

        sk_acceptq_removed(sk);     /*sk->sk_ack_backlog--，/include/net/sock.h*/
        if (sk->sk_protocol == IPPROTO_TCP &&tcp_rsk(req)->tfo_listener &&queue->fastopenq) {
            spin_lock_bh(&queue->fastopenq->lock);
            if (tcp_rsk(req)->tfo_listener) {
                req->sk = NULL;
                req = NULL;
            }
            spin_unlock_bh(&queue->fastopenq->lock);
        }
    out:
        release_sock(sk);
        if (req)
            reqsk_put(req);       /*释放连接请求 request_sock 实例*/
        return newsk;       /*返回连接请求关联的 sock 实例*/
        ...
    }
```

inet_csk_accept()函数中从欢迎套接字已建立连接请求队列中取出第一个连接请求，取出其关联的连接套接字 sock 实例返回给调用者，释放连接请求实例。如果已建立连接请求队列为空，将睡眠等待（需是阻塞访问，否则不等待返回错误码）。

连接请求是在主动连接套接字（客户进程）发起连接请求，欢迎套接字接收到连接请求后创建的，随后传输协议将完成连接的建立，并将请求移动到欢迎套接字已建立连接请求队列，等待被动连接套接字处理（由 accept()系统调用处理）。下一小节将介绍主动连接套接字如何发起连接请求，欢迎套接字如何接收连接请求，以及 TCP 如何建立连接。

12.8.7 建立连接

TCP 是面向连接的传输层协议。在 Linux 内核实现中，客户进程套接字与服务器进程套接字通信前需要建立连接，由客户进程发出连接请求，服务器进程响应请求，并创建一个新连接套接字用于与用户进程套接字通信。本小节介绍客户进程如何发送连接请求，以及连接如何最终建立。

1 概述

TCP 连接操作由主动连接套接字（客户进程）发起，被动连接套接字（服务器进程）接受连接请求。在 TCP 客户端和 TCP 服务器之间，使用 3 次握手来建立 TCP 连接（一共需 3 个报文段）：

●客户端通过 connect()系统调用向服务器发送 SYN 请求报文段（SYN 标志位置 1，ACK 标志位清零，带序列号），客户套接字状态设为 TCP_SYN_SENT。

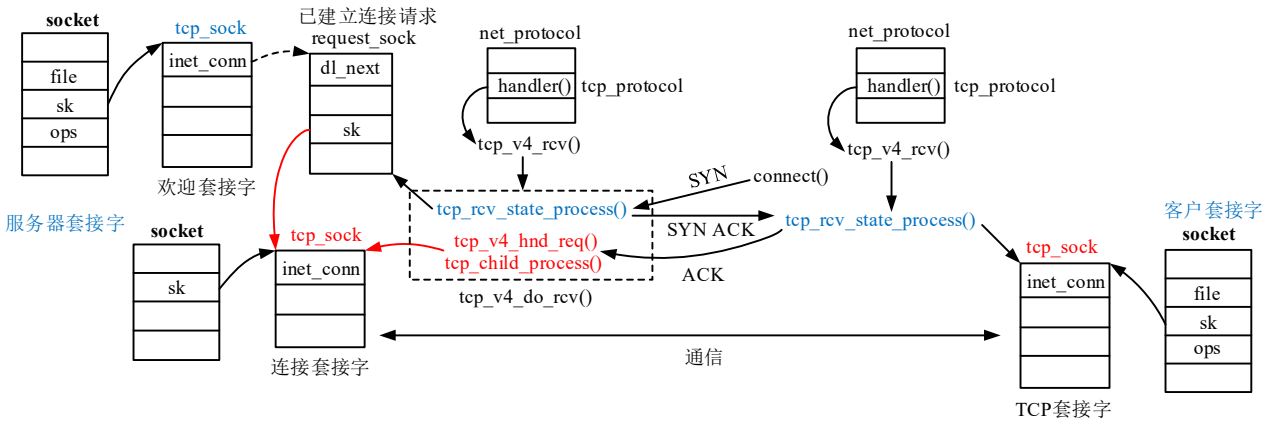
●服务器监听连接请求的套接字（状态为 TCP_LISTEN）收到 SYN 请求后，创建一个连接请求，添加到连接请求散列表，并向客户发回一个 SYN ACK 应答报文段（SYN 置 1，ACK 置 1，带序列号和确认号，确认号为 SYN 请求报文段中序列号加 1）。

●客户端收到 SYN ACK 报文段后，将套接字状态设为 TCP_ESTABLISHED，并向服务器发送一个 ACK 应答报文段（SYN 清零，ACK 置 1，带序列号和确认号，序列号为 SYN 请求中序列号加 1，确认号为 SYN ACK 中序列号加 1）。

●服务器收到 ACK 报文段后，创建连接套接字，其状态为 TCP_ESTABLISHED，并将连接请求移动到服务器欢迎套接字中已建立连接请求队列。

服务器进程 accept()系统调用将取出已建立连接请求中的连接套接字，关联到 socket 实例，以使服务器进程能通过文件描述符访问连接套接字，并释放连接请求。

在以上连接建立的过程中一共发送了 3 个报文段，分别是 SYN 请求、SYN ACK 和 ACK 报文段。其中 SYN 请求报文段由客户进程在 connect()系统调用中发送，SYN ACK 和 ACK 报文段由两端的 TCP 接收和发送，如下图所示。



前面介绍过，每个传输层协议需要定义并注册 net_potocol 实例，其中的 handler()函数用于接收数据包。对于 TCP 此函数为 tcp_v4_rcv()，在 tcp_v4_rcv()函数内将调用 tcp_v4_do_rcv()函数接收数据包。在连接操作中由 tcp_v4_do_rcv()函数内调用的 tcp_rcv_state_process()和 tcp_v4_hnd_req()、tcp_child_process()函数接收相关的数据包。

在客户端，通过 connect()系统调用向服务器发送 SYN 请求。服务器 TCP 收到 SYN 请求后，由函数 tcp_rcv_state_process()接收并处理，主要工作是创建连接请求，插入到连接请求散列表，向用户发回 SYN ACK 应答。

客户 TCP 收到 SYN ACK 后也由 tcp_rcv_state_process()函数处理（不是由 connect()系统调用处理），函数内查找到发出 SYN 请求的客户套接字，将其状态设为 TCP_ESTABLISHED，并向服务器发送 ACK 应答。

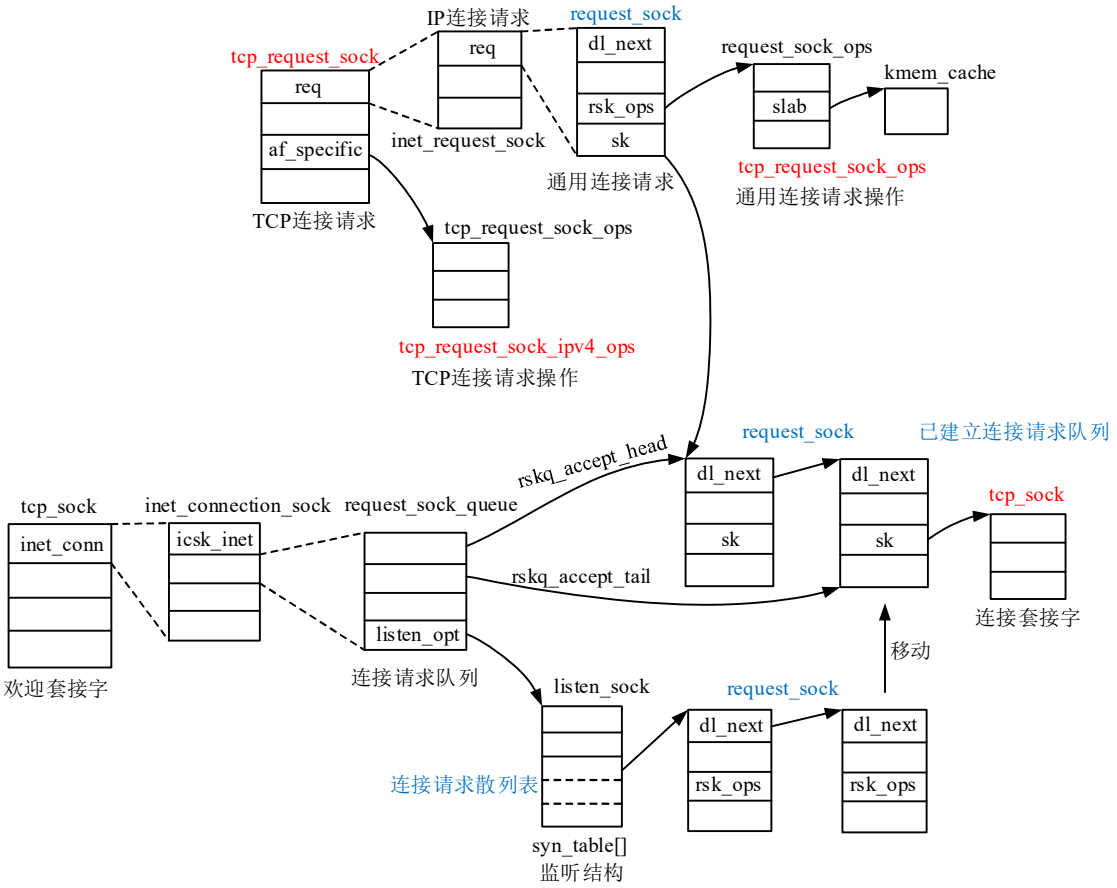
服务器 TCP 在收到 ACK 应答后，由 tcp_v4_hnd_req()和 tcp_child_process()函数处理。tcp_v4_hnd_req()函数为连接请求创建连接套接字，并初始化，套接字状态设为 TCP_SYN_RECV，将连接请求移动到已建立连接请求队列。

tcp_child_process()函数将 ACK 应答数据包交由连接套接字处理，函数内对连接套接字进行进一步的初始化，设置套接字状态为 TCP_ESTABLISHED，释放 ACK 应答数据包等。至此，3 次握手完成，连接正式建立。

建立连接操作中还有一项重要的任务时设置各连接参数、套接字参数等。有的参数来自于路由选择查找结果。

2 TCP 连接请求

TCP 连接请求由 tcp_request_sock 结构体表示，内嵌通用连接请求 request_sock 结构体成员，此成员添加到欢迎套接字中的连接请求散列表或已建立连接请求队列，如下图所示。



TCP 连接请求 tcp_request_sock 结构体定义如下（/include/linux/tcp.h）：

```
struct tcp_request_sock {
    struct inet_request_sock     req;        /*内嵌 IP 连接请求 inet_request_sock 结构体*/
    const struct tcp_request_sock_ops  *af_specific;   /*TCP 连接请求操作结构，/include/net/tcp.h*/
    bool                tfo_listener;
    u32                 rcv_isn;        /*接收端初始序列号*/
    u32                 snt_isn;        /*发送端初始序列号*/
    u32                 snt_synack;     /*SYN ACK 发送时间*/
    u32                 last_oow_ack_time;   /* last SYNACK */
    u32                 rcv_nxt;        /**/
};
```

tcp_request_sock 结构体主要成员简介如下：

●**req**：inet_request_sock 结构体成员，表示通用 IP 地址簇连接请求，定义如下（/include/net/inet_sock.h）：

```
struct inet_request_sock {
    struct request_sock     req;        /*通用连接请求 request_sock 结构体成员*/
    ...     /*宏定义，定义 req 结构体中成员*/
    kmemcheck_bitfield_begin(flags);
```

```
    u16             snd_wscale : 4,
                    rcv_wscale : 4,
                    tstamp_ok  : 1,
                    sack_ok    : 1,
                    wscale_ok  : 1,
                    ecn_ok     : 1,
                    acked      : 1,
                    no_srccheck: 1;
    kmemcheck_bitfield_end(flags);
    u32             ir_mark;
    union {
        struct ip_options_rcu   *opt;    /*IP 选项*/
        struct sk_buff  *pktopts;
    };
};
```

●**af_specific**：指向 TCP 连接请求操作结构 tcp_request_sock_ops，定义如下（/include/net/tcp.h）：

```
struct tcp_request_sock_ops {
    u16 mss_clamp;
  #ifdef CONFIG_TCP_MD5SIG
    ...
  #endif
    void (*init_req)(struct request_sock *req, struct sock *sk,struct sk_buff *skb);   /*初始化连接请求*/
  #ifdef CONFIG_SYN_COOKIES
    __u32 (*cookie_init_seq)(struct sock *sk, const struct sk_buff *skb,__u16 *mss);
  #endif
    struct dst_entry *(*route_req)(struct sock *sk, struct flowi *fl,
                                   const struct request_sock *req,bool *strict);    /*路由选择*/
    __u32 (*init_seq)(const struct sk_buff *skb);       /*初始化序列号*/
    int (*send_synack)(struct sock *sk, struct dst_entry *dst,struct flowi *fl, struct request_sock *req,
                                       u16 queue_mapping, struct tcp_fastopen_cookie *foc);
                                                        /*发送 SYN ACK 应答*/
    void (*queue_hash_add)(struct sock *sk, struct request_sock *req,const unsigned long timeout);
};
```

内核在/net/ipv4/tcp_ipv4.c 文件内定义了 TCP 连接请求操作结构 tcp_request_sock_ipv4_ops 实例，如下所示，将赋予 TCP 连接请求 tcp_request_sock 实例：

```
static const struct tcp_request_sock_ops tcp_request_sock_ipv4_ops = {
    .mss_clamp    =    TCP_MSS_DEFAULT,
  #ifdef CONFIG_TCP_MD5SIG
    ...
  #endif
    .init_req  =    tcp_v4_init_req,      /*初始化 TCP 连接请求 tcp_request_sock 实例*/
  #ifdef CONFIG_SYN_COOKIES
    .cookie_init_seq =  cookie_v4_init_sequence,
  #endif
```

```
    .route_req      =   tcp_v4_route_req,    /*路由选择查找结果，/net/ipv4/tcp_ipv4.c*/
    .init_seq  =    tcp_v4_init_sequence,    /*初始化序列号，/net/ipv4/tcp_ipv4.c*/
    .send_synack  =    tcp_v4_send_synack,      /*发送 SYN ACK，/net/ipv4/tcp_ipv4.c*/
    .queue_hash_add = inet_csk_reqsk_queue_hash_add,    /*/net/ipv4/inet_connection_sock.c*/
                                        /*将连接请求插入到欢迎套接字连接请求散列表*/
};
```

在建立连接的操作中（3 次握手）将调用 tcp_request_sock_ipv4_ops 实例中的相关函数。

内核在创建连接请求时，需要从 proto 实例中关联的通用连接请求操作结构 request_sock_ops 实例中的 slab 缓存中分配连接请求数据结构实例。

TCP 关联的 tcp_prot 实例中指定的 request_sock_ops 实例为 **tcp_request_sock_ops**，如下所示：

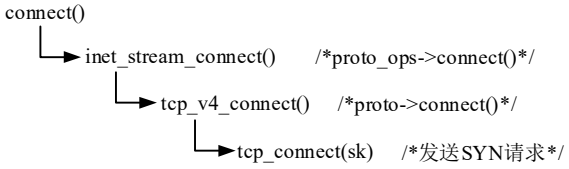
```
struct proto tcp_prot = {
    ...
    .rsk_prot       = &tcp_request_sock_ops,      /*通用连接请求操作结构实例*/
    ...
}
```

TCP 关联的通用连接请求操作 tcp_request_sock_ops 实例定义如下（/net/ipv4/tcp_ipv4.c）：

```
struct request_sock_ops tcp_request_sock_ops __read_mostly = {
    .family         =    PF_INET,
    .obj_size  =    sizeof(struct tcp_request_sock),
            /*TCP 连接请求由 tcp_request_sock 结构体表示，slab 缓存在注册 proto 实例时创建*/
    .rtx_syn_ack   =    tcp_rtx_synack,
    .send_ack=    tcp_v4_reqsk_send_ack,
    .destructor     =    tcp_v4_reqsk_destructor,
    .send_reset     =    tcp_v4_send_reset,
    .syn_ack_timeout =tcp_syn_ack_timeout,
};
```

3 发出连接请求

主动连接进程通过 connect()系统调用主动发起连接请求（SYN 请求），系统调用内函数调用关系简列如下图所示：



connect()系统调用将调用 proto_ops->connect()函数，此函数内又将调用 proto->connect()函数，TCP 套接字最终由函数 **tcp_connect()**发出 SYN 请求报文段。

connect()系统调用实现代码如下（/net/socket.c）：

```
SYSCALL_DEFINE3(connect, int, fd, struct sockaddr __user *, uservaddr,int, addrlen)
/*fd：文件描述符，uservaddr：被动连接套接字（服务器进程）地址，addrlen：地址长度*/
{
    struct socket *sock;
    struct sockaddr_storage address;    /*保存被连接套接字地址（对方套接字）*/
```

```
    int err, fput_needed;

    sock = sockfd_lookup_light(fd, &err, &fput_needed);     /*主动连接套接字 socket 实例*/
    ...       /*错误处理*/
    err = move_addr_to_kernel(uservaddr, addrlen, &address);     /*复制用户空间地址到内核*/
    ...       /*错误处理*/
    err =security_socket_connect(sock, (struct sockaddr *)&address, addrlen);   /*安全性检查*/
    ...       /*错误处理*/
    err = sock->ops->connect(sock, (struct sockaddr *)&address, addrlen,sock->file->f_flags);
                                                    /*调用 proto_ops 实例中的 connect()函数*/
out_put:
    fput_light(sock->file, fput_needed);
out:
    return err;       /*连接成功返回 0*/
}
```

connect()系统调用内调用 proto_ops 实例中的 connect()函数，对于流套接字为 inet_stream_connect()函数，定义如下（/net/ipv4/af_inet.c）：

```
int inet_stream_connect(struct socket *sock, struct sockaddr *uaddr,int addr_len, int flags)
/*sock：主动连接套接字，uaddr：对方地址，flags：文件 file 中标志成员*/
{
    int err;

    lock_sock(sock->sk);     /*持有套接字锁*/
    err = __inet_stream_connect(sock, uaddr, addr_len, flags);       /*/net/ipv4/af_inet.c*/
    release_sock(sock->sk);     /*释放套接字锁*/
    return err;       /*成功返回 0*/
}
```

__inet_stream_connect()函数定义如下（/net/ipv4/af_inet.c）：

```
int __inet_stream_connect(struct socket *sock, struct sockaddr *uaddr,int addr_len, int flags)
{
    struct sock *sk = sock->sk;
    int err;
    long timeo;

    if (addr_len < sizeof(uaddr->sa_family))
        return -EINVAL;

    if (uaddr->sa_family == AF_UNSPEC) {     /*地址簇为 AF_UNSPEC，断开连接*/
        err = sk->sk_prot->disconnect(sk, flags);
        sock->state = err ? SS_DISCONNECTING : SS_UNCONNECTED;
        goto out;
    }

    switch (sock->state) {        /*主动连接 socket 状态*/
```

```
    ...
    case SS_UNCONNECTED:      /*套接字 socket 处于 SS_UNCONNECTED 状态*/
        err = -EISCONN;
        if (sk->sk_state != TCP_CLOSE)     /*sock 初始为 TCP_CLOSE 状态*/
            goto out;

        err = sk->sk_prot->connect(sk, uaddr, addr_len);     /*调用 proto 实例中的 connect()函数*/
                                                             /*主要是发送 SYN 请求报文段*/
        ...
        sock->state = SS_CONNECTING;     /*设置 socket 状态为 SS_CONNECTING，正在连接*/
        err = -EINPROGRESS;
        break;
    }

    timeo = sock_sndtimeo(sk, flags & O_NONBLOCK);       /*超时时间，/include/net/sock.h*/
                                    /*返回 0 或  sk->sk_sndtimeo（阻塞访问则等待连接完成）*/
    if ((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV)) {
                                     /*sock 状态是 TCP_SYN_SENT 或 TCP_SYN_RECV*/
        int writebias = (sk->sk_protocol == IPPROTO_TCP) &&tcp_sk(sk)->fastopen_req &&
                                tcp_sk(sk)->fastopen_req->data ? 1 : 0;

        if (!timeo || !inet_wait_for_connect(sk, timeo, writebias))       /*等待连接完成或超时被唤醒*/
            goto out;

        err = sock_intr_errno(timeo);
        if (signal_pending(current))
            goto out;
    }

    /*如果连接已经断开（由于复位，超时，ICMP 错误等）*/
    if (sk->sk_state == TCP_CLOSE)
        goto sock_error;
    sock->state = SS_CONNECTED;      /*连接成功，设置 socket 为已连接状态*/
    err = 0;
out:
    return err;        /*成功返回 0*/
    ...      /*错误处理*/
}
```

由__inet_stream_connect()函数可知，发起连接请求的套接字 socket 状态需为 SS_UNCONNECTED，sock 状态需为 TCP_CLOSE，函数调用 proto 实例中的 connect()函数，执行主动连接操作，对于 TCP 来说就是发送 SYN 请求。然后，调用 inet_wait_for_connect()函数等待连接建立完成（需是阻塞访问，连接由传输层协议完成），进程在此函数中添加到套接字睡眠进程等待队列。当连接完成或超时时，将唤醒在套接字上睡眠等待的进程，__inet_stream_connect()函数继续往下执行，如果连接建立成功，将设置 socket 状态为 SS_CONNECTED，函数返回 0，connect()系统调用也返回 0。

小结：如果进程是以阻塞方式访问套接字，在 connect()系统调用中将调用 proto 实例中的 connect()函数发出连接请求（SYN 请求），并等待连接建立完成（非阻塞不等待），成功系统调用返回 0，出错返回

错误码。

■TCP 发送连接请求

TCP 套接字通过 proto->connect()函数发出 SYN 连接请求，此函数为 **tcp_v4_connect()**，函数定义如下：

```
int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)       /*/net/ipv4/tcp_ipv4.c*/
{
    struct sockaddr_in *usin = (struct sockaddr_in *)uaddr;
    struct inet_sock *inet = inet_sk(sk);
    struct tcp_sock *tp = tcp_sk(sk);
    __be16 orig_sport, orig_dport;
    __be32 daddr, nexthop;
    struct flowi4 *fl4;
    struct rtable *rt;
    int err;
    struct ip_options_rcu *inet_opt;

    ...    /*目的地址有效性检查*/

    nexthop = daddr = usin->sin_addr.s_addr;        /*目的 IP 地址*/
    inet_opt = rcu_dereference_protected(inet->inet_opt,sock_owned_by_user(sk));
    if (inet_opt && inet_opt->opt.srr) {       /*IP 选项*/
        if (!daddr)
            return -EINVAL;
        nexthop = inet_opt->opt.faddr;
    }

    orig_sport = inet->inet_sport;       /*源端口号*/
    orig_dport = usin->sin_port;        /*目的端口号*/
    fl4 = &inet->cork.fl.u.ip4;
    rt = ip_route_connect(fl4, nexthop, inet->inet_saddr,
                          RT_CONN_FLAGS(sk), sk->sk_bound_dev_if,IPPROTO_TCP,
                          orig_sport, orig_dport, sk);        /*路由选择查找，见第 13 章*/
    ...    /*错误处理*/

    if (rt->rt_flags & (RTCF_MULTICAST | RTCF_BROADCAST)) {
        ip_rt_put(rt);                                  /*目的地址为广播和组播地址，返回错误码*/
        return -ENETUNREACH;
    }

    if (!inet_opt || !inet_opt->opt.srr)
        daddr = fl4->daddr;          /*目的 IP 地址*/

    if (!inet->inet_saddr)        /*若未指定源 IP 地址，使用路由选择查找结果中源 IP 地址*/
        inet->inet_saddr = fl4->saddr;
    sk_rcv_saddr_set(sk, inet->inet_saddr);       /*设置接收源 IP 地址，sk->sk_rcv_saddr*/
```

```
    if (tp->rx_opt.ts_recent_stamp && inet->inet_daddr != daddr) {
        tp->rx_opt.ts_recent          = 0;
        tp->rx_opt.ts_recent_stamp = 0;
        if (likely(!tp->repair))
            tp->write_seq       = 0;
    }

    if (tcp_death_row.sysctl_tw_recycle &&!tp->rx_opt.ts_recent_stamp && fl4->daddr == daddr)
        tcp_fetch_timewait_stamp(sk, &rt->dst);

    inet->inet_dport = usin->sin_port;     /*目的端口号*/
    sk_daddr_set(sk, daddr);       /*设置目的 IP 地址*/

    inet_csk(sk)->icsk_ext_hdr_len = 0;
    if (inet_opt)
        inet_csk(sk)->icsk_ext_hdr_len = inet_opt->opt.optlen;

    tp->rx_opt.mss_clamp = TCP_MSS_DEFAULT;

    tcp_set_state(sk, TCP_SYN_SENT);
    err = inet_hash_connect(&tcp_death_row, sk);       /*/net/ipv4/inet_hashtables.c*/
                    /*将套接字插入已连接套接字散列表，如果没有设置端口号则还需要分配端口号，
                    *执行绑定操作，将套接字插入绑定散列表。
                    */

    ...      /*错误处理*/
    inet_set_txhash(sk);     /*设置 sk->sk_txhash，/include/net/ip.h*/

    rt = ip_route_newports(fl4, rt, orig_sport, orig_dport,inet->inet_sport, inet->inet_dport, sk);
                                                    /*路由选择查找（设置了新端口号）*/
    ...   /*错误处理*/

    sk->sk_gso_type = SKB_GSO_TCPV4;     /*设置 GSO 类型*/
    sk_setup_caps(sk, &rt->dst);     /*/net/core/sock.c*/
      /*设置 sk->sk_dst_cache、sk->sk_route_caps（表示网络接口能力）及 sk->sk_gso_max_segs 等*/
    if (!tp->write_seq && likely(!tp->repair))
        tp->write_seq = secure_tcp_sequence_number(inet->inet_saddr,inet->inet_daddr,
                                                   inet->inet_sport,usin->sin_port);
                                                  /*确定发送序列号*/
    inet->inet_id = tp->write_seq ^ jiffies;       /*分段 id 值*/
    err = tcp_connect(sk);          /*发送 SYN 请求数据包，/net/ipv4/tcp_output.c*/
    ...
    return 0;
    ...
}
```

tcp_v4_connect()函数首先为发送 SYN 请求做一些准备工作，如设置套接字中的源、目的 IP 地址和端口号，将套接字插入已连接套接字散列表（没有绑定端口号需要先执行绑定操作），执行路由选择查找，设置序列号、分段 id 值等，最后调用 **tcp_connect()**函数发送 SYN 请求数据包。

需要注意的是在 **inet_hash_connect()**函数中，如果套接字没有绑定端口号，将为其自动分配端口号，执行绑定操作，将套接字插入绑定散列表。然后，以源 IP 地址、源端口号、目的 IP 地址、目的端口号计算散列值，将套接字 tcp_sock 实例插入到已连接套接字散列表中。

●发送 SYN 请求

TCP 通过 tcp_connect()函数发送 SYN 请求报文段，函数定义如下（/net/ipv4/tcp_output.c）：

```
int tcp_connect(struct sock *sk)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *buff;
    int err;

    tcp_connect_init(sk);     /*初始化套接字连接参数，/net/ipv4/tcp_output.c*/

    if (unlikely(tp->repair)) {       /*如果是配对套接字*/
        tcp_finish_connect(sk, NULL);
                        /*完成连接，设置 TCP_ESTABLISHED 状态等等，/net/ipv4/tcp_input.c*/
        return 0;
    }

    buff = sk_stream_alloc_skb(sk, 0, sk->sk_allocation, true); /*分配 sk_buff 实例（带内存缓存区）*/
    ...       /*错误处理*/
    tcp_init_nondata_skb(buff, tp->write_seq++, TCPHDR_SYN);     /*设置报头中 SYN 标志位等*/
                                        /*初始化 sk_buff 实例，无用户数据，/net/ipv4/tcp_output.c*/
    tp->retrans_stamp = tcp_time_stamp;
    tcp_connect_queue_skb(sk, buff); /*将 sk_buff 添加到发送缓存队列末尾等，/net/ipv4/tcp_output.c*/
    tcp_ecn_send_syn(sk, buff);       /*写入 ECN 状态，/net/ipv4/tcp_output.c*/

    err = tp->fastopen_req ? tcp_send_syn_data(sk, buff) :
                        tcp_transmit_skb(sk, buff, 1, sk->sk_allocation);     /*发送报文段*/
    ...       /*错误处理*/

    tp->snd_nxt = tp->write_seq;       /*下一个序列号*/
    tp->pushed_seq = tp->write_seq;
    TCP_INC_STATS(sock_net(sk), TCP_MIB_ACTIVEOPENS);       /*更新统计量*/

    /*重置发送定时器，如未收到 SYN ACK，则重新发送 SYN 请求，
    */include/net/inet_connection_sock.h。
    */
    inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,inet_csk(sk)->icsk_rto, TCP_RTO_MAX);
    return 0;
}
```

tcp_connect()函数的主要工作是初始化套接字的连接参数，分配构建不带用户数据的 sk_buff 实例，设

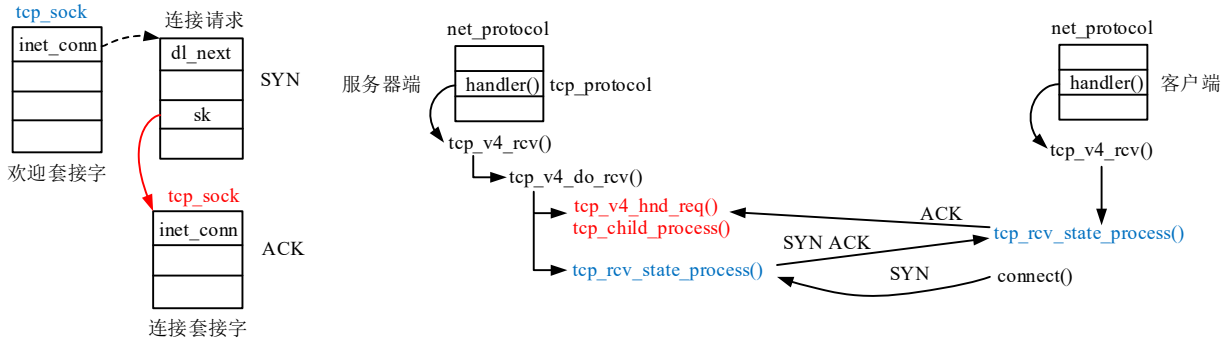
置 TCP 报头中的 SYN 标志位、序列号等，然后调用 **tcp_transmit_skb()**函数发送带 SYN 请求的 sk_buff 实例至网络层。**tcp_transmit_skb()**函数在后面介绍 TCP 发送数据时将会详细介绍。

4 完成连接

客户进程通过 connect()系统调用向服务器发送 SYN 请求数据包后，客户进程在套接字上睡眠等待，等待 TCP 连接建立完成后将进程唤醒，唤醒后从 connect()系统调用中返回（需是阻塞访问模式）。下面介绍通信双方 TCP 如何完成连接操作。

■概述

客户进程发送 SYN 请求后，剩下的连接操作由 TCP 完成。如下图所示，服务器端和客户端接收数据包的函数是相同的，都是 tcp_v4_rcv()函数，此函数内将调用 tcp_v4_do_rcv()接收数据包（详见下一节）。



tcp_v4_do_rcv()函数内将调用 tcp_rcv_state_process()和 tcp_v4_hnd_req()、tcp_child_process()函数。

服务器端通过 tcp_rcv_state_process()函数接收客户端发送的 SYN 请求，创建连接请求实例，插入连接请求散列表，并向客户端发送 SYN ACK 数据包。

客户端通过 tcp_rcv_state_process()函数接收服务器端发送的 SYN ACK 数据包，并向服务器发送 ACK 数据包。

服务器端通过 tcp_v4_hnd_req()函数接收 ACK 数据包，为连接请求创建连接套接字，并将连接请求移动到欢迎套接字已建立连接请求队列，连接套接字状态设为 TCP_SYN_RECV。tcp_child_process()函数将 ACK 数据包交由连接套接字处理，连接套接字将进一步初始化，状态设为 TCP_ESTABLISHED。此时，3 次握手完成，连接建立。

下面看一下 tcp_v4_do_rcv()函数的实现，代码简列如下（/net/ipv4/tcp_ipv4.c）：

```
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
    struct sock *rsk;

    if (sk->sk_state == TCP_ESTABLISHED) {      /*连接已建立时，接收数据包*/
        ...
        return 0;
    }
    ...     /*检查校验和*/

    if (sk->sk_state == TCP_LISTEN) {       /*欢迎套接字接收 ACK 应答*/
        struct sock *nsk = tcp_v4_hnd_req(sk, skb);     /*创建连接套接字（TCP_SYN_RECV）*/
```

```
            if (!nsk)                    /*未创建连接请求，未建立连接时，返回 sk，即 nsk==sk*/
                goto discard;

            if (nsk != sk) {      /*接收 ACK 时，nsk 为连接套接字 sock 实例指针，nsk!=sk*/
                sock_rps_save_rxhash(nsk, skb);
                sk_mark_napi_id(sk, skb);
                if (tcp_child_process(sk, nsk, skb)) {
                                        /*连接套接字处理 ACK 应答，TCP_ESTABLISHED*/
                    rsk = nsk;
                    goto reset;
                }
                return 0;      /*响应 ACK 结束，返回 0*/
            }
        } else      /*套接字状态不为 TCP_LISTEN*/
            sock_rps_save_rxhash(sk, skb);

        /*处理套接字状态不是 TCP_ESTABLISHED 时的情况（含 TCP_LISTEN 状态）*/
        if (tcp_rcv_state_process(sk, skb, tcp_hdr(skb), skb->len)) {   /*接收 SYN 和 SYN ACK*/
            ...
        }
        return 0;
        ...
    }
```

■接收 SYN 请求和 SYN ACK

tcp_rcv_state_process()函数需要负责接收 SYN 请求和 SYN ACK 应答，分别在欢迎套接字和客户套接字中实现，函数代码简列如下（/net/ipv4/tcp_input.c）：

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb,const struct tcphdr *th, unsigned int len)
/*sk：服务器欢迎套接字或客户套接字，th：指向 TCP 报头*/
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct inet_connection_sock *icsk = inet_csk(sk);
    struct request_sock *req;
    int queued = 0;
    bool acceptable;
    u32 synack_stamp;

    tp->rx_opt.saw_tstamp = 0;

    switch (sk->sk_state) {   /*欢迎套接字为 TCP_LISTEN，客户套接字为 TCP_SYN_SENT*/
    case TCP_CLOSE:
        goto discard;

    case TCP_LISTEN:   /*适用于欢迎套接字*/
        if (th->ack)
```

```
            return 1;
        if (th->rst)
            goto discard;

        if (th->syn) {      /*收到 SYN 请求*/
            if (th->fin)
                goto discard;
            if (icsk->icsk_af_ops->conn_request(sk, skb) < 0)      /*处理 SYN 请求*/
                                /*调用地址簇操作结构中的函数，创建请求、发送 SYN ACK 应答等*/
                return 1;
            kfree_skb(skb);
            return 0;      /*SYN 请求处理成功，返回 0*/
        }
        goto discard;

    case TCP_SYN_SENT:   /*适用客户套接字发送 SYN 请求后，接收 SYN ACK 并发送 ACK*/
        queued = tcp_rcv_synsent_state_process(sk, skb, th, len);     /*/net/ipv4/tcp_input.c*/
                                /*设置连接参数，发送 ACK 等，函数返回-1*/
        if (queued >= 0)
            return queued;

        tcp_urg(sk, skb, th);
        __kfree_skb(skb);       /*释放接收的 sk_buff 实例*/
        tcp_data_snd_check(sk);
        return 0;      /*响应 SYN ACK 成功，函数返回 0*/
    }

    /*以下处理套接字不是 TCP_CLOSE、TCP_LISTEN、TCP_SYN_SENT 状态的情况*/
    ...
    return 0;
}
```

●接收 SYN 请求

在 tcp_rcv_state_process()函数中，如果是欢迎套接字接收到 SYN 请求数据包，将调用地址簇操作结构 inet_connection_sock_af_ops 实例中的 **conn_request()**函数，接收并响应 SYN 请求。

对于 TCP 套接字，地址簇操作结构实例为 **ipv4_specific**，简列如下（/net/ipv4/tcp_ipv4.c）：

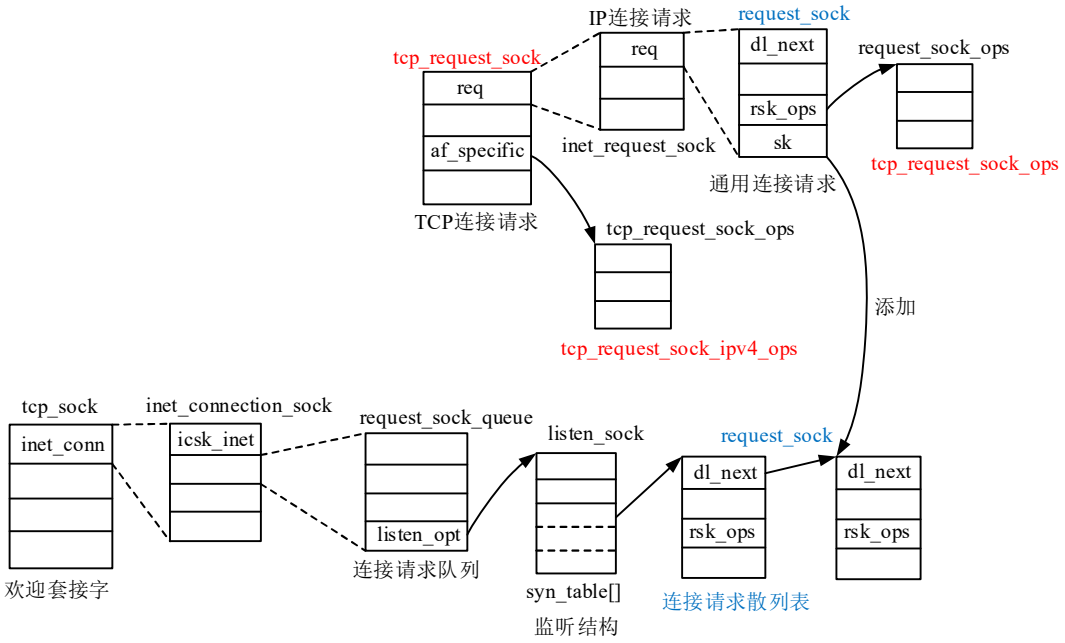
```
const struct inet_connection_sock_af_ops ipv4_specific = {
    .queue_xmit        = ip_queue_xmit,     /*向网络层发送数据包的函数*/
    ...
    .conn_request      = tcp_v4_conn_request,      /*接收连接 SYN 请求，并响应*/
    ...
}
```

ipv4_specific 实例中 conn_request()函数为 tcp_v4_conn_request()，定义如下（/net/ipv4/tcp_ipv4.c）：

```
int tcp_v4_conn_request(struct sock *sk, struct sk_buff *skb)
{
    /*不响应广播和组播的 SYN 请求*/
```

```
    if (skb_rtable(skb)->rt_flags & (RTCF_BROADCAST | RTCF_MULTICAST))
        goto drop;

    return tcp_conn_request(&tcp_request_sock_ops,&tcp_request_sock_ipv4_ops, sk, skb);
    ...
}
```

tcp_conn_request()函数内将创建表示 TCP 连接请求的 tcp_request_sock 实例，并将内嵌的通用连接请求 request_sock 实例添加到连接请求散列表，如下图所示：



在 tcp_conn_request()函数中指定了 TCP 连接请求操作结构实例为 **tcp_request_sock_ipv4_ops**，通用连接请求操作结构实例为 **tcp_request_sock_ops**。

tcp_conn_request()函数代码简列如下（/net/ipv4/tcp_input.c）：

```
int tcp_conn_request(struct request_sock_ops *rsk_ops,const struct tcp_request_sock_ops *af_ops,
                                                    struct sock *sk, struct sk_buff *skb)
/*rsk_ops：指向 tcp_request_sock_ops 实例，af_ops：指向 tcp_request_sock_ipv4_ops 实例
*sk：欢迎套接字，skb：SYN 请求数据包。
*/
{
    struct tcp_options_received    tmp_opt;      /*保存 SYN 数据包中的 TCP 选项*/
    struct request_sock *req;
    struct tcp_sock *tp = tcp_sk(sk);
    struct dst_entry *dst = NULL;
    __u32 isn = TCP_SKB_CB(skb)->tcp_tw_isn;      /*初始值为 0*/
    bool want_cookie = false, fastopen;
    struct flowi fl;
    struct tcp_fastopen_cookie foc = { .len = -1 };
    int err;

    ...      /*检查是否可接受连接请求*/

    req = inet_reqsk_alloc(rsk_ops, sk);     /*分配 TCP 连接请求实例，/net/ipv4/tcp_input.c*/
```

```
...   /*错误处理*/
tcp_rsk(req)->af_specific = af_ops; /*TCP 连接请求操作结构实例赋值 tcp_request_sock_ipv4_ops*/

tcp_clear_options(&tmp_opt);
tmp_opt.mss_clamp = af_ops->mss_clamp;
tmp_opt.user_mss    = tp->rx_opt.user_mss;
tcp_parse_options(skb, &tmp_opt, 0, want_cookie ? NULL : &foc);       /*解析 TCP 选项*/

if (want_cookie && !tmp_opt.saw_tstamp)
      tcp_clear_options(&tmp_opt);

tmp_opt.tstamp_ok = tmp_opt.saw_tstamp;
tcp_openreq_init(req, &tmp_opt, skb, sk);    /*初始化 inet_request_sock 实例，/net/ipv4/tcp_input.c*/

inet_rsk(req)->ir_iif = sk->sk_bound_dev_if;

af_ops->init_req(req, sk, skb);    /*初始化 TCP 连接请求 tcp_request_sock 实例，tcp_v4_init_req()*/

if (security_inet_conn_request(sk, skb, req))
      goto drop_and_free;

if (!want_cookie && !isn) {
      ...
      isn = af_ops->init_seq(skb);        /*初始化序列号*/
}
if (!dst) {
      dst = af_ops->route_req(sk, &fl, req, NULL);
      ...
}
tcp_ecn_create_request(req, skb, sk, dst);

if (want_cookie) {
      isn = cookie_init_sequence(af_ops, sk, skb, &req->mss);
      req->cookie_ts = tmp_opt.tstamp_ok;
      if (!tmp_opt.tstamp_ok)
            inet_rsk(req)->ecn_ok = 0;
}

tcp_rsk(req)->snt_isn = isn;       /*发送 SYN ACK 数据包中的序列号*/
tcp_openreq_init_rwin(req, sk, dst);     /*初始化窗口值，/net/ipv4/tcp_minisocks.c*/
fastopen = !want_cookie &&tcp_try_fastopen(sk, skb, req, &foc, dst);
err = af_ops->send_synack(sk, dst, &fl, req,skb_get_queue_mapping(skb), &foc);
                        /*发送 SYN ACK，tcp_v4_send_synack()，/net/ipv4/tcp_ipv4.c*/

if (!fastopen) {
      if (err || want_cookie)
```

```
            goto drop_and_free;

        tcp_rsk(req)->tfo_listener = false;
        af_ops->queue_hash_add(sk, req, TCP_TIMEOUT_INIT);
                                    /*inet_csk_reqsk_queue_hash_add()函数，
                                    *将连接请求插入套接字 icsk->icsk_accept_queue 连接请求散列表。
                                    */
    }
    tcp_reqsk_record_syn(sk, req, skb);      /*保存 SYN 请求 IP 报头，/net/ipv4/tcp_ipv4.c*/

    return 0;      /*返回 0*/
    ...
}
```

tcp_conn_request()函数内首先创建 TCP 连接请求 tcp_request_sock 实例，调用 TCP 连接请求操作结构 **tcp_request_sock_ipv4_ops** 实例中的函数，完成初始化连接请求实例，初始化序列号，发送 SYN ACK 应答，将请求实例插入连接请求散列表等工作。

tcp_request_sock_ipv4_ops 实例定义在/net/ipv4/tcp_ipv4.c 文件内，实例中函数源代码请读者自行阅读。

●接收 SYN ACK 应答

客户套接字在发送 SYN 请求时，其状态设为 **TCP_SYN_SENT**。由 tcp_rcv_state_process()函数可知，此时 TCP 调用 **tcp_rcv_synsent_state_process()**函数接收服务器欢迎套接字发送的 SYN ACK 应答。SYN ACK 中 TCP 报头 SYN 和 ACK 标志位置位，确认号为 SYN 请求中序列号加 1，序列号为服务器序列号。

tcp_rcv_synsent_state_process()函数代码简列如下（/net/ipv4/tcp_input.c）：

```
static int tcp_rcv_synsent_state_process(struct sock *sk, struct sk_buff *skb,
                                         const struct tcphdr *th, unsigned int len)
{
    struct inet_connection_sock *icsk = inet_csk(sk);
    struct tcp_sock *tp = tcp_sk(sk);
    struct tcp_fastopen_cookie foc = { .len = -1 };
    int saved_clamp = tp->rx_opt.mss_clamp;

    tcp_parse_options(skb, &tp->rx_opt, 0, &foc);      /*解析 TCP 选项*/
    if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr)
        tp->rx_opt.rcv_tsecr -= tp->tsoffset;

    if (th->ack) {
        ...  /*检查应答是否有效*/

        tcp_ecn_rcv_synack(tp, th);

        tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);
        tcp_ack(sk, skb, FLAG_SLOWPATH);     /*处理 ACK 信息*/

        tp->rcv_nxt = TCP_SKB_CB(skb)->seq + 1;     /*下一个接收字节序列号*/
        tp->rcv_wup = TCP_SKB_CB(skb)->seq + 1;
```

```
            tp->snd_wnd = ntohs(th->window);        /*设置接收窗口*/

            if (!tp->rx_opt.wscale_ok) {
                tp->rx_opt.snd_wscale = tp->rx_opt.rcv_wscale = 0;
                tp->window_clamp = min(tp->window_clamp, 65535U);
            }

            if (tp->rx_opt.saw_tstamp) {
                ...
            } else {
                tp->tcp_header_len = sizeof(struct tcphdr);
            }

            if (tcp_is_sack(tp) && sysctl_tcp_fack)
                tcp_enable_fack(tp);

            tcp_mtup_init(sk);
            tcp_sync_mss(sk, icsk->icsk_pmtu_cookie);
            tcp_initialize_rcv_mss(sk);     /*初始化 MSS 值，/net/ipv4/tcp_input.c*/
            tp->copied_seq = tp->rcv_nxt;

            smp_mb();

            tcp_finish_connect(sk, skb);        /*/net/ipv4/tcp_input.c*/
                        /*完成连接，设置连接参数，套接字状态设为 TCP_ESTABLISHED 等*/
            if ((tp->syn_fastopen || tp->syn_data) &&tcp_rcv_fastopen_synack(sk, skb, &foc))
                return -1;

            if (sk->sk_write_pending ||
                icsk->icsk_accept_queue.rskq_defer_accept ||
                icsk->icsk_ack.pingpong) {
                ...
            } else {
                tcp_send_ack(sk);       /*发送 ACK 等，/net/ipv4/tcp_output.c*/
            }
            return -1;          /*返回-1*/
        }       /*if (th->ack)结束*/

        ...     /*处理 ACK 标志位没有置位的数据包，忽略它*/
    }
```

tcp_rcv_synsent_state_process()函数调用 tcp_finish_connect(sk, skb)函数完成连接，主要是设置连接参数，设置套接字状态为 TCP_ESTABLISHED，唤醒在套接字上睡眠等待的进程（调用 connect()系统调用的进程）等；调用 tcp_send_ack(sk)函数向服务器发送 ACK 应答，最后函数返回-1。

■接收 ACK 应答

在服务器端，由 tcp_v4_do_rcv()函数接收客户端 ACK 应答，函数代码简列如下（/net/ipv4/tcp_ipv4.c）：

```
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
/*sk：欢迎套接字，skb：ACK 应答数据包*/
{
    struct sock *rsk;
    ...
    if (sk->sk_state == TCP_LISTEN) {        /*适用于欢迎套接字*/
        struct sock *nsk = tcp_v4_hnd_req(sk, skb);     /*接收客户端 ACK 应答，创建连接套接字*/
        if (!nsk)                                /*返回连接套接字 sock 实例指针，状态为 TCP_SYN_RECV*/
            goto discard;

        if (nsk != sk) {       /*接收 ACK 时，nsk 表示连接套接字，不等于 sk*/
            sock_rps_save_rxhash(nsk, skb);
            sk_mark_napi_id(sk, skb);
            if (tcp_child_process(sk, nsk, skb)) {    /*/net/ipv4/tcp_minisocks.c*/
                                /*连接套接字处理 ACK 应答，套接字状态设为 TCP_ESTABLISHED*/
                rsk = nsk;
                goto reset;
            }
            return 0;       /*响应 ACK 结束，返回 0*/
        }
    } else
    ...
}
```

tcp_v4_do_rcv()函数内对 ACK 应答的处理主要分两步：

（1）调用 tcp_v4_hnd_req(sk, skb)函数为连接请求创建连接套接字，套接字端口号与欢迎套接字相同，设置套接字连接参数，套接字状态设为 TCP_SYN_RECV，将连接套接字添加到绑定散列表和连接散列表，将连接请求移动到欢迎套接字已建立连接请求队列等。

（2）调用 tcp_child_process(sk, nsk, skb)函数，将 ACK 应答数据包交由连接套接字处理，此函数内调用前面介绍过的 tcp_rcv_state_process()函数，执行将连接套接字状态设为 TCP_ESTABLISHED，唤醒在欢迎套接字上睡眠等待进程（执行 accept()系统调用进程）等工作。

至此连接已经建立，连接请求关联连接套接字，连接请求添加到欢迎套接字已建立连接请求队列。欢迎套接字执行 accept()系统调用时，将从已建立连接请求队列取出一个请求，将其连接套接字关联到 socket 实例，释放连接请求，服务器进程可通过文件描述符访问连接套接字，与客户套接字通信。

●创建连接套接字

服务器端收到 ACK 时，此时已创建了连接请求实例，并添加到了连接请求散列表。tcp_v4_hnd_req()函数首先在连接请求散列表中查找连接请求实例，然后调用 tcp_check_req()函数为其创建连接套接字，函数代码简列如下（/net/ipv4/tcp_ipv4.c）：

```
static struct sock *tcp_v4_hnd_req(struct sock *sk, struct sk_buff *skb)
{
    const struct tcphdr *th = tcp_hdr(skb);
    const struct iphdr *iph = ip_hdr(skb);
    struct request_sock *req;       /*通用连接请求*/
```

```
    struct sock *nsk;

    req = inet_csk_search_req(sk, th->source, iph->saddr, iph->daddr);
                                    /*在连接请求散列表中查找连接请求实例，没找到返回 NULL*/
    if (req) {      /*此时连接请求已创建，req 指向通用连接请求*/
        nsk = tcp_check_req(sk, skb, req, false);      /*创建连接套接字，/net/ipv4/tcp_minisocks.c*/
        if (!nsk || nsk == sk)
            reqsk_put(req);
        return nsk;      /*返回连接套接字 sock 实例指针*/
    }
    ...
}
```

tcp_check_req()函数用于为连接请求创建连接套接字，函数代码简列如下（/net/ipv4/tcp_minisocks.c）：

```
struct sock *tcp_check_req(struct sock *sk, struct sk_buff *skb,struct request_sock *req,bool fastopen)
/*sock：欢迎套接字，skb：ACK 数据包，req：通用连接请求，fastopen：false*/
{
    struct tcp_options_received tmp_opt;
    struct sock *child;
    const struct tcphdr *th = tcp_hdr(skb);
    __be32 flg = tcp_flag_word(th) & (TCP_FLAG_RST|TCP_FLAG_SYN|TCP_FLAG_ACK);
    bool paws_reject = false;
    ...
    child = inet_csk(sk)->icsk_af_ops->syn_recv_sock(sk, skb, req, NULL);
                        /*调用地址簇操作结构中 syn_recv_sock()函数，创建连接套接字等*/
    ...
    inet_csk_reqsk_queue_drop(sk, req);     /*从连接请求散列表中移除连接请求*/
                                            /*/net/ipv4/inet_connection_sock.c*/
    inet_csk_reqsk_queue_add(sk, req, child);     /*/include/net/inet_connection_sock.h*/
        /*将连接请求移动到已建立连接请求队列，连接请求关联连接套接字（req->sk=child）*/
    return child;      /*返回连接套接字 sock 实例*/
    ...
}
```

tcp_check_req()函数比较复杂，上面只列出了与 ACK 应答创建连接套接字相关的代码。函数内调用套接字关联的地址簇操作结构 inet_connection_sock_af_ops 实例中的 syn_recv_sock()函数为连接请求创建连接套接字，然后将连接请求从连接请求散列表移动到已建立连接请求队列，最后返回连接套接字 sock 实例指针。

对于 TCP 套接字，关联的 inet_connection_sock_af_ops 实例定义如下：

```
const struct inet_connection_sock_af_ops ipv4_specific = {
    ...
    .syn_recv_sock     = tcp_v4_syn_recv_sock,
    ...
}
```

tcp_v4_syn_recv_sock()函数用于为 TCP 连接请求创建连接套接字，函数定义如下（/net/ipv4/tcp_ipv4.c）：

```
struct sock *tcp_v4_syn_recv_sock(struct sock *sk, struct sk_buff *skb,struct request_sock *req,
```

```
                                                    struct dst_entry *dst)
/*sk：欢迎套接字，skb：ACK 数据包，req：通用连接请求，dst：NULL*/
{
    struct inet_request_sock *ireq;
    struct inet_sock *newinet;
    struct tcp_sock *newtp;
    struct sock *newsk;
    ...
    struct ip_options_rcu *inet_opt;

    if (sk_acceptq_is_full(sk))
        goto exit_overflow;

    newsk = tcp_create_openreq_child(sk, req, skb);      /*/net/ipv4/minisocks.c*/
                        /*创建连接套接字，复制欢迎套接字参数，初始化连接套接字，
                        *套接字状态设为 TCP_SYN_RECV 等*/
    if (!newsk)
        goto exit_nonewsk;

    /*以下是初始化连接套接字*/
    newsk->sk_gso_type = SKB_GSO_TCPV4;
    inet_sk_rx_dst_set(newsk, skb);

    newtp              = tcp_sk(newsk);
    newinet            = inet_sk(newsk);
    ireq               = inet_rsk(req);
    sk_daddr_set(newsk, ireq->ir_rmt_addr);
    sk_rcv_saddr_set(newsk, ireq->ir_loc_addr);     /*设置接收 IP 地址*/
    newinet->inet_saddr      = ireq->ir_loc_addr;     /*源 IP 地址*/
    inet_opt           = ireq->opt;
    rcu_assign_pointer(newinet->inet_opt, inet_opt);
    ireq->opt          = NULL;
    newinet->mc_index        = inet_iif(skb);
    newinet->mc_ttl            = ip_hdr(skb)->ttl;
    newinet->rcv_tos         = ip_hdr(skb)->tos;
    inet_csk(newsk)->icsk_ext_hdr_len = 0;
    inet_set_txhash(newsk);
    if (inet_opt)
        inet_csk(newsk)->icsk_ext_hdr_len = inet_opt->opt.optlen;
    newinet->inet_id = newtp->write_seq ^ jiffies;       /*分段 id 值*/

    if (!dst) {
        dst = inet_csk_route_child_sock(sk, newsk, req);
        if (!dst)
            goto put_and_exit;
    } else {
```

```
        }
        sk_setup_caps(newsk, dst);

        tcp_ca_openreq_child(newsk, dst);       /*设置拥塞控制算法和状态，/net/ipv4/minisocks.c*/

        tcp_sync_mss(newsk, dst_mtu(dst));
        newtp->advmss = dst_metric_advmss(dst);
        if (tcp_sk(sk)->rx_opt.user_mss &&
            tcp_sk(sk)->rx_opt.user_mss < newtp->advmss)
            newtp->advmss = tcp_sk(sk)->rx_opt.user_mss;

        tcp_initialize_rcv_mss(newsk);
        ...
        if (__inet_inherit_port(sk, newsk) < 0)
                                /*连接套接字添加到绑定散列表，与欢迎套接字端口号相同*/
            goto put_and_exit;
        __inet_hash_nolisten(newsk, NULL);  /*将连接套接字添加到已连接套接字散列表*/
        return newsk;       /*返回连接套接字 sock 实例*/
        ...
    }
```

tcp_v4_syn_recv_sock()函数为连接请求创建连接套接字，初始化套接字，将套接字添加到绑定散列表和已建立连接套接字散列表。

●连接套接字处理 ACK

创建连接套接字后，tcp_v4_do_rcv()函数继续调用 **tcp_child_process(sk, nsk, skb)**函数，将 ACK 应答数据包交由连接套接字处理。tcp_child_process()函数定义如下（/net/ipv4/tcp_minisocks.c）：

```
    int tcp_child_process(struct sock *parent, struct sock *child,struct sk_buff *skb)
    /*parent：欢迎套接字，child：连接套接字，skb：ACK 应答数据包*/
    {
        int ret = 0;
        int state = child->sk_state;        /*连接套接字状态，此时为 TCP_SYN_RECV*/

        if (!sock_owned_by_user(child)) {       /*连接套接字未被用户锁定*/
            ret = tcp_rcv_state_process(child, skb, tcp_hdr(skb),skb->len);
            if (state == TCP_SYN_RECV && child->sk_state != state)
                parent->sk_data_ready(parent);      /*唤醒欢迎套接字，在 accept()系统调用中睡眠*/
        } else {
            __sk_add_backlog(child, skb);     /*连接套接字被锁定，ACK 数据包添加到后备队列*/
        }

        bh_unlock_sock(child);
        sock_put(child);
        return ret;
    }
```

tcp_child_process()函数内调用 **tcp_rcv_state_process**()函数由连接套接字处理 ACK 数据包，函数相关代码简列如下（/net/ipv4/tcp_input.c）：

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb,const struct tcphdr *th, unsigned int len)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct inet_connection_sock *icsk = inet_csk(sk);
    struct request_sock *req;
    int queued = 0;
    bool acceptable;
    u32 synack_stamp;
    ...
    req = tp->fastopen_rsk;
    ...
    switch (sk->sk_state) {     /*此时连接套接字状态为 TCP_SYN_RECV*/
    case TCP_SYN_RECV:
        if (!acceptable)
            return 1;

        if (req) {
            ...
        } else {
            synack_stamp = tp->lsndtime;
            icsk->icsk_af_ops->rebuild_header(sk);
            tcp_init_congestion_control(sk);     /*初始化拥塞控制算法*/

            tcp_mtup_init(sk);
            tp->copied_seq = tp->rcv_nxt;
            tcp_init_buffer_space(sk);
        }

        smp_mb();
        tcp_set_state(sk, TCP_ESTABLISHED);    /*连接套接字状态设为 TCP_ESTABLISHED*/
        sk->sk_state_change(sk);

        if (sk->sk_socket)
            sk_wake_async(sk, SOCK_WAKE_IO, POLL_OUT);

        tp->snd_una = TCP_SKB_CB(skb)->ack_seq;
        tp->snd_wnd = ntohs(th->window) << tp->rx_opt.snd_wscale;
        tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);
        tcp_synack_rtt_meas(sk, synack_stamp);

        if (tp->rx_opt.tstamp_ok)
            tp->advmss -= TCPOLEN_TSTAMP_ALIGNED;

        if (req) {
```

```
            ...
        } else
            tcp_init_metrics(sk);     /*初始化元参数*/

        tcp_update_pacing_rate(sk);
        tp->lsndtime = tcp_time_stamp;

        tcp_initialize_rcv_mss(sk);
        tcp_fast_path_on(tp);
        break;
        ...
    }
    ...
    return 0;
}
```

在 tcp_rcv_state_process()函数内，将套接字状态设为 TCP_ESTABLISHED，并对连接套接字参数进行进一步的初始化。此时 3 次握手完成，连接正式建立，套接字之间可以相互通信了。

12.8.8 发送数据

进程通过套接字发送数据的 write()、send()等系统调用内将调用 proto_ops 实例中的 sendmsg()函数发送数据，对于 TCP 套接字此函数为 inet_sendmsg()。inet_sendmsg()函数内判断当前套接字是否绑定了端口号，如果没有则执行自动绑定，为套接字分配端口号，并将 tcp_sock 实例插入管理散列表，最后调用 proto 实例中的 sendmsg()函数发送用户数据。

自动绑定函数为 inet_autobind()，同 UDP 中的绑定函数。此函数内将调用 tcp_prot 实例中的 get_port()函数 inet_csk_get_port()，为套接字自动分配端口号，并将套接字插入到管理散列表中。

TCP 套接字关联 proto 实例 tcp_prot 中 sendmsg()函数为**tcp_sendmsg()**，此函数执行特定于 TCP 的发送用户数据操作，本节主要介绍 tcp_sendmsg()函数的实现。

1 概述

TCP 发送用户数据时，TCP 首先对用户数据进行分段，生成多个数据包，并插入到套接字发送缓存队列。若网络接口不支持分散/聚集功能，此时用户数据分段的长度为 MSS 值，用户数据只保存在数据包内存缓存区中。若网络接口支持分散/聚集功能（来自连接操作中的路由选择查找结果），用户数据将保存在数据包内存缓存区和分散数据块中，内存缓存区中用户数据大小接近 2KB 或 1 个内存页，分散数据块的大小及数量与 UDP 中分散数据块相同。

分段后的数据包插入到套接字发送缓存队列，发送数据包时，TCP 遍历队列中尚未发送的数据包，逐个将数据包发送到网络层。在发送数据包时，发送用户数据量受对方接收窗口、拥塞窗口，以及 MSS 值的限制，因此 TCP 在将数据包发往网络层之前，可能还需要对数据包进行分段，以满足各种限制条件。分段后的数据包通过套接字关联的 inet_connection_sock_af_ops 实例中的 queue_xmit()函数发往网络层，TCP 套接字此函数为 **ip_queue_xmit()**。

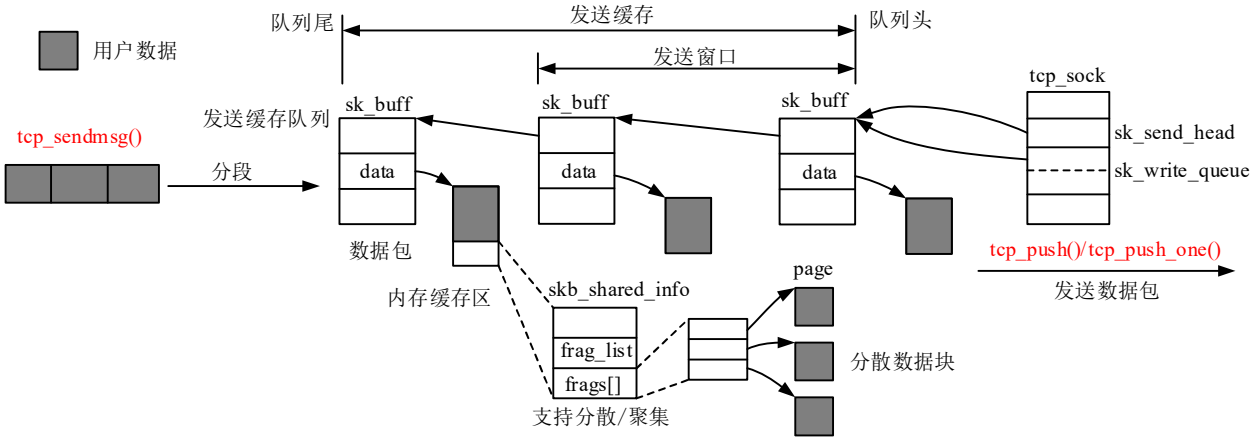
■发送流程

TCP 发送数据函数 tcp_sendmsg()与 UDP 的发送数据函数有点类似。函数内对用户数据进行分段，生成数据包，插入到套接字发送缓存队列，在适当的时候发送队列中的数据包到网络层。

TCP 发送函数 tcp_sendmsg()执行流程如下图所示，在对用户数据进行分段时与 UDP 有些区别。TCP

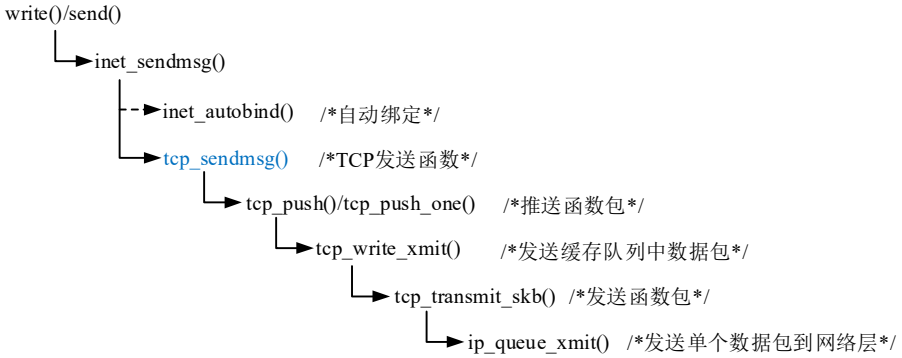
将用户数据视为字节流，而不是一个个的数据报，所有用户数据包都添加到发送缓存队列，然后以流水线的方式将队列中数据包发送到网络层。

在对用户数据是行分段时，将先填充数据包内存缓存区。如果网络接口支持分散/聚集功能，内存缓存区填满后，将填充数据包中的分散数据块，分散数据块填满后才会分配新数据包。如果网络设备不支持分散/聚集功能，将只填充数据包内存缓存区，不填充分散数据块，填满后分配新数据包。



TCP 套接字发送缓存区队列长度存在一个限制值（发送缓存内存长度），如果发送缓存队列达到限制值，TCP 将先发送缓存队列中数据包，腾出空间后，再对用户数据分段生成数据包，插入缓存队列。同时，套接字发送缓存队列中还存在一个发送窗口（滑动窗口），它受接收窗口、拥塞窗口等值限制，用于控制发送到网络，但还在网络上传输的数据量，以达到流量控制和拥塞控制的目的。

tcp_sendmsg()函数调用关系如下图所示：



tcp_sendmsg()函数中将调用 tcp_push()/tcp_push_one()等函数将发送缓存队列中数据包发送到网络层。

tcp_write_xmit()函数是发送队列中数据包的接口函数，函数内可能只发送一个数据包，也可能发送多个数据包，并且可能需要对数据包进行分段。tcp_transmit_skb()函数是发送单个已分段数据包的接口函数，**ip_queue_xmit()**函数是地址簇操作结构中指定的发送数据包函数，用于将数据包发送到网络层（IP 层）。

■控制结构

TCP 数据包控制块由 tcp_skb_cb 结构体表示，结构体实例由数据包 sk_buff 实例 cb[]成员表示，用于控制数据包的发送行为。

tcp_skb_cb 结构体定义如下（/include/net/tcp.h）：

```
struct tcp_skb_cb {
    __u32       seq;        /*用户数据起始序列号*/
    __u32       end_seq;    /*用户数据结束序列号*/
    union {
        __u32   tcp_tw_isn;    /*仅用于接收路径*/
        struct {
```

```
            u16  tcp_gso_segs;    /*发送前对数据包分段的数量，发送路径*/
            u16  tcp_gso_size;    /*分段长度*/
        };
    };
    __u8     tcp_flags;    /*TCP 报头标记，  (tcp[13]) */

    __u8     sacked;         /* State flags for SACK/FACK.*/
#define  TCPCB_SACKED_ACKED      0x01   /* SKB ACK'd by a SACK block*/
#define  TCPCB_SACKED_RETRANS    0x02   /* SKB retransmitted*/
#define  TCPCB_LOST              0x04   /* SKB is lost*/
#define  TCPCB_TAGBITS           0x07   /* All tag bits/
#define  TCPCB_REPAIRED          0x10   /* SKB repaired (no skb_mstamp) */
#define  TCPCB_EVER_RETRANS      0x80   /* Ever retransmitted frame    */
#define  TCPCB_RETRANS        (TCPCB_SACKED_RETRANS|TCPCB_EVER_RETRANS| \
                               TCPCB_REPAIRED)

    __u8          ip_dsfield;    /* IPv4 tos or IPv6 dsfield（服务类型）*/
    __u32         ack_seq;       /*确认号*/
    union {
        struct inet_skb_parm    h4;     /*IP 选项，/include/net/ip.h*/
#if IS_ENABLED(CONFIG_IPV6)
        struct inet6_skb_parm   h6;
#endif
    } header;   /* For incoming frames*/
};
```

tcp_skb_cb 结构体中 tcp_gso_segs、tcp_gso_size 成员表示的分段数量和分段长度，是指对 tcp_sendmsg()函数中已经创建的数据包，在发送到网络层前进行的分段。此分段操作在 tcp_write_xmit()函数中执行（对单个数据包再进行分段，前面的分段是直接对用户数据的分段，以生成数据包），因受限于接收窗口、拥塞窗口、MSS 等值的限制，需要对数据包分段，这称为 TCP 分段，是传输层的分段。

2 TCP 发送函数

TCP 通过 tcp_sendmsg()函数发送用户数据，此函数的首要任务是要对用户数据进行分段，以生成数据包。此函数内首先取出套接字发送缓存队列中最末尾数据包，检查能否在此数据包中再写入数据，如果能则写入数据，直到填充满此数据包，然后再创建新数据包填充用户数据，并将数据包插入发送缓存队列。TCP 在合适的时机将发送缓存队列中的数据包发送到网络层。

tcp_sendmsg()函数中通过一个 while 循环对用户数据进行分段，每次循环保证填满数据包中的内存缓区或一个分散数据块，而不是一次就填满一个数据包。如果队列中最末尾数据包内存缓存区或分散数据块还有空闲，则先填满它，然后再填充下一个内存缓存区或分散数据块。一个数据包的内存缓存区和分散数据块（如果使用）都填满后，再创建新数据包，插入发送缓存队列，填充用户数据。如此循环，直至用户数据分段完成。

另外，在用户数据分段、创建新数据包过程中需要满足一些限制条件，如发送缓存队列长度不能超过限制值等。用户数据分段结束后或达到限制条件时，将发送缓存队列中数据包。

tcp_sendmsg()函数定义如下（/net/ipv4/tcp.c）：

```
int tcp_sendmsg(struct sock *sk, struct msghdr *msg, size_t size)
/*sk：发送套接字，msg：用户数据、目的地址等，size：数据长度*/
```

```
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *skb;
    int flags, err, copied = 0;
    int mss_now = 0, size_goal, copied_syn = 0;
                        /*size_goal 表示一个数据包中用户数据长度，包括内存缓存区和分散数据块*/
    bool sg;
    long timeo;

    lock_sock(sk);

    flags = msg->msg_flags;         /*发送标记*/
    if (flags & MSG_FASTOPEN) {
        err = tcp_sendmsg_fastopen(sk, msg, &copied_syn, size);
        ...   /*错误处理*/
    }

    timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);       /*发送超时时间*/
    if (((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT)) &&
        !tcp_passive_fastopen(sk)) {
        if ((err = sk_stream_wait_connect(sk, &timeo)) != 0)
            goto do_error;
    }           /*等待连接完成*/

    if (unlikely(tp->repair)) {
        if (tp->repair_queue == TCP_RECV_QUEUE) {
            copied = tcp_send_rcvq(sk, msg, size);
            goto out_nopush;
        }

        err = -EINVAL;
        if (tp->repair_queue == TCP_NO_QUEUE)
            goto out_err;
    }

    clear_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);

    mss_now = tcp_send_mss(sk, &size_goal, flags);       /*返回当前 MSS 值，/net/ipv4/tcp.c*/
                        /*size_goal 记录一个数据包中用户数据最大长度，
                          *如果网络接口不支持 GSO，size_goal 等于 MSS 值，
                          *如果网络接口支持 GSO，数据包将使用分散数据块保存用户数据，
                          *size_goal 记录数据包中所有用户数据（含分散数据块）总长度的最大值。
                          */
    copied = 0;
    err = -EPIPE;
    if (sk->sk_err || (sk->sk_shutdown & SEND_SHUTDOWN))
```

```
        goto out_err;

sg = !!(sk->sk_route_caps & NETIF_F_SG); /*网络接口是否支持分散/聚集，来自路由选择查找*/

while (msg_data_left(msg)) {       /*while 循环开始，对用户数据进行分段，生成数据包*/
        int copy = 0;
        int max = size_goal;     /*数据包用户数据最大长度*/

        skb = tcp_write_queue_tail(sk);   /*指向发送缓存队列最末尾数据包，sk->sk_write_queue*/
        if (tcp_send_head(sk)) {       /*判断 sk->sk_send_head 值，指向发送头数据包*/
                if (skb->ip_summed == CHECKSUM_NONE)
                        max = mss_now;
                copy = max - skb->len;     /*最末尾数据包中还能写入数据的长度*/
        }

        if (copy <= 0) {     /*最末尾数据包中不能再写入数据，重新分配 sk_buff 实例*/
new_segment:
                if (!sk_stream_memory_free(sk))   /*如果发送队列长度超过发送窗口长度，发送数据包*/
                        goto wait_for_sndbuf;              /*(sk->sk_wmem_queued >= sk->sk_sndbuf)*/

                skb = sk_stream_alloc_skb(sk,select_size(sk, sg),sk->sk_allocation,
                                          skb_queue_empty(&sk->sk_write_queue)); /*分配 sk_buff 实例*/
                if (!skb)
                        goto wait_for_memory;
                if (sk->sk_route_caps & NETIF_F_ALL_CSUM)
                        skb->ip_summed = CHECKSUM_PARTIAL;

                skb_entail(sk, skb);
                /*将 skb 添加到发送缓存队列末尾，设置 TCPHDR_ACK 标记，
                *更新 sk->sk_wmem_queued 值等，/net/ipv4/tcp.c*/

                copy = size_goal;       /*数据包中用户数据最大长度*/
                max = size_goal;

                if (tp->repair)
                        TCP_SKB_CB(skb)->sacked |= TCPCB_REPAIRED;
        }       /*if (copy <= 0)  结束，已分配新数据包*/

        /*尝试将用户数据写入最末尾数据包或新创建数据包*/
        if (copy > msg_data_left(msg))
                copy = msg_data_left(msg);

        if (skb_availroom(skb) > 0) {
                /*分散数据块或分段数据包中有数据时，返回 0，
                *否则，返回内存缓存区中剩余空间大小，不含分散数据块/分段数据包。
                *大于 0 表示内存缓存区中还有空间，可以写入数据。*/
```

```
        copy = min_t(int, copy, skb_availroom(skb));     /*复制数据大小*/
        err = skb_add_data_nocache(sk, skb, &msg->msg_iter, copy);
                                        /*复制用户数据到 sk_buff 实例内存缓存区*/
        if (err)
            goto do_fault;
    } else {      /*内存缓存区中没有空闲空间了，将用户数据复制到分散数据块中*/
        bool merge = true;
        int i = skb_shinfo(skb)->nr_frags;
        struct page_frag *pfrag = sk_page_frag(sk);
                        /*sock 中 page_frag 结构体成员，指向最近分配的分散数据块*/
        if (!sk_page_frag_refill(sk, pfrag))
                        /*确保 pfrag 指向分散数据块中有空闲空间，如果没有则新分配*/
            goto wait_for_memory;

        if (!skb_can_coalesce(skb, i, pfrag->page,pfrag->offset)) {
            /*pfrag 指向的是新分配的分散数据块，或数据包中分散数据块为空，返回假*/
            if (i == MAX_SKB_FRAGS || !sg) {
                /*数据包中分散数据块数量达到最大值或不支持分散/聚集，创建新数据包*/
                tcp_mark_push(tp, skb);     /*设置 tcp_skb_cb 实例中 PUSH 控制标志*/
                goto new_segment;        /*跳转至 new_segment，创建新数据包*/
            }
            merge = false;       /*分配了新分散数据块，不与现有的合并*/
        }

        copy = min_t(int, copy, pfrag->size - pfrag->offset);   /*此次复制用户数据的长度*/

        if (!sk_wmem_schedule(sk, copy))
            goto wait_for_memory;

        err = skb_copy_to_page_nocache(sk, &msg->msg_iter, skb,pfrag->page,pfrag->offset,copy);
                        /*复制用户数据至新分散数据块，增加各长度计数，/include/net/sock.h*/
        if (err)
            goto do_error;

        if (merge) {       /*用户数据被复制到已有分散数据块，执行了合并*/
            skb_frag_size_add(&skb_shinfo(skb)->frags[i - 1], copy);
        } else {           /*用户数据被复制到新分散数据块，nr_frags = i + 1*/
            skb_fill_page_desc(skb, i, pfrag->page,pfrag->offset, copy);
                                    /*分散数据块关联到 skb_shinfo(skb)->nr_frags[i]数组项*/
            get_page(pfrag->page);
        }
        pfrag->offset += copy;       /*增加分散数据块中空闲区域起始偏移量*/
    }

    if (!copied)        /*copied 为 0*/
        TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_PSH;   /*清除 PUSH 控制标志*/
```

```
            tp->write_seq += copy;      /*发送队列中用户数据最后序列号*/
            TCP_SKB_CB(skb)->end_seq += copy;      /*数据包结束序列号*/
            tcp_skb_pcount_set(skb, 0);     /*TCP_SKB_CB(skb)->tcp_gso_segs 设为 0*/

            copied += copy;        /*已复制数据长度之和*/
            if (!msg_data_left(msg)) {          /*用户数据分段完成*/
                tcp_tx_timestamp(sk, skb);
                goto out;           /*跳转至 out 处，跳出 while 循环，发送数据包*/
            }

            if (skb->len < max || (flags & MSG_OOB) || unlikely(tp->repair))
                continue;

            if (forced_push(tp)) {      /*需要将数据尽快交给用户*/
                tcp_mark_push(tp, skb);       /*/net/ipv4/tcp.c*/
                __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
                                        /*发送数据包，/net/ipv4/tcp_output.c*/
            } else if (skb == tcp_send_head(sk))       /*skb 是当前发送头数据包*/
                tcp_push_one(sk, mss_now);      /*发送单个数据包*/
            continue;       /*继续分段，跳转至  while (msg_data_left(msg))  处*/

wait_for_sndbuf:          /*等待腾出发送窗口*/
            set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
wait_for_memory:          /*发送缓存队列达到最大值，发送数据包*/
            if (copied)
                tcp_push(sk, flags & ~MSG_MORE, mss_now,TCP_NAGLE_PUSH, size_goal);

            if ((err = sk_stream_wait_memory(sk, &timeo)) != 0)
                                        /*等待更多的发送缓存，/net/core/stream.c*/
                goto do_error;

            mss_now = tcp_send_mss(sk, &size_goal, flags); /*计算新 MSS 值和数据包用户数据最大长度*/
        }     /*while 循环结束，用户数据分段结束*/

    /*用户数据分段结束，发送队列中数据包*/
out:
        if (copied)       /*已复制数据不为 0*/
            tcp_push(sk, flags, mss_now, tp->nonagle, size_goal);      /*/net/ipv4/tcp.c*/
                                        /*调用__tcp_push_pending_frames()函数，发送数据包*/
out_nopush:
        release_sock(sk);
        return copied + copied_syn;       /*返回复制的数据*/
        ...       /*错误处理*/
}
```

上面函数代码中都添加了注释，请读者自行阅读。下面介绍创建新数据包、将数据包插入发送缓存队

列，以及发送队列数据包函数的实现。

■创建数据包

在 tcp_sendmsg()函数中调用 sk_stream_alloc_skb()函数为流套接字创建新数据包，其中 select_size(sk, **sg**)函数用于计算数据包中内存缓存区用户数据的长度，不含分散数据块。

select_size(sk, **sg**)函数定义如下（/net/ipv4/tcp.c）：

```
static inline int select_size(const struct sock *sk, bool sg)
/*sg：网络接口是否支持分散/聚集*/
{
    const struct tcp_sock *tp = tcp_sk(sk);
    int tmp = tp->mss_cache; /*套接字缓存的 MSS 值，在创建套接字时初始化，在连接操作中重置*/

    if (sg) {       /*网络设备支持分散/聚集*/
        if (sk_can_gso(sk)) {       /*网络设备是否支持 GSO，来自于连接操作中路由选择查找结果*/
            tmp = SKB_WITH_OVERHEAD(2048 - MAX_TCP_HEADER);       /*不超过 2KB*/
        } else {        /*不支持 GSO*/
            int pgbreak = SKB_MAX_HEAD(MAX_TCP_HEADER);     /*不超过一个内存页*/

            if (tmp >= pgbreak &&tmp <= pgbreak + (MAX_SKB_FRAGS - 1) * PAGE_SIZE)
                tmp = pgbreak;
        }
    }
    return tmp;
}
```

如果网络接口不支持分散/聚集，select_size()函数返回 MSS 值。如果网络接口支持分散/聚集功能，且支持 GSO，函数返回内存缓存区用户数据长度值，即内存缓存区总大小不超过 2KB。如果网络接口支持分散/聚集功能，但不支持 GSO，内存缓存区大小一般指定为一个物理内存页。

sk_stream_alloc_skb()函数用于为流套接字创建新数据包，函数定义如下（/net/ipv4/tcp.c）：

```
struct sk_buff *sk_stream_alloc_skb(struct sock *sk, int size, gfp_t gfp,bool force_schedule)
/*size：select_size(sk, sg)计算的大小，force_schedule：发送缓存队列是否为空*/
{
    struct sk_buff *skb;

    size = ALIGN(size, 4);        /*4 字节对齐*/

    if (unlikely(tcp_under_memory_pressure(sk)))
        sk_mem_reclaim_partial(sk);

    skb = alloc_skb_fclone(size + sk->sk_prot->max_header, gfp); /*创建 sk_buff 实例，带内存缓存区*/
    if (likely(skb)) {
        bool mem_scheduled;

        if (force_schedule) {       /*发送缓存队列为空*/
            mem_scheduled = true;
```

```
                sk_forced_mem_schedule(sk, skb->truesize);
            } else {                    /*发送缓存队列非空*/
                mem_scheduled = sk_wmem_schedule(sk, skb->truesize);
            }
            if (likely(mem_scheduled)) {
                skb_reserve(skb, sk->sk_prot->max_header);
                skb->reserved_tailroom = skb->end - skb->tail - size;
                return skb;             /*返回 sk_buff 实例*/
            }
            __kfree_skb(skb);
        } else {
            ...         /*分配失败*/
        }
        return NULL;
    }
```

sk_stream_alloc_skb()函数中 size 参数为 select_size(sk, **sg**)函数计算的数据包内存缓存区中用户数据大小。

■插入发送队列

tcp_sendmsg()函数中新创建的数据包直接插入到发送缓存队列末尾，填入用户数据后，等待 TCP 将其发送到网络层。skb_entail()函数用于将数据包添加到发送缓存队列末尾，定义如下（/net/ipv4/tcp.c）：

```
static void skb_entail(struct sock *sk, struct sk_buff *skb)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct tcp_skb_cb *tcb = TCP_SKB_CB(skb);        /*数据包控制结构*/

    skb->csum       = 0;
    tcb->seq        = tcb->end_seq = tp->write_seq;      /*数据包序列号*/
    tcb->tcp_flags = TCPHDR_ACK;        /*设置应答标志位*/
    tcb->sacked    = 0;       /*清除 SACK 标志*/
    __skb_header_release(skb);        /*设置数据包没有头部标志*/
    tcp_add_write_queue_tail(sk, skb);      /*将数据包添加到发送缓存队列末尾，/include/net/tcp.h*/
    sk->sk_wmem_queued += skb->truesize;      /*增加发送缓存队列（内存）长度*/
    sk_mem_charge(sk, skb->truesize);
    if (tp->nonagle & TCP_NAGLE_PUSH)
        tp->nonagle &= ~TCP_NAGLE_PUSH;     /*清除标志位*/
}
```

tcp_add_write_queue_tail()函数将数据包插入发送缓存队列末尾，函数定义如下：

```
static inline void tcp_add_write_queue_tail(struct sock *sk, struct sk_buff *skb)
{
    __tcp_add_write_queue_tail(sk, skb);      /*将数据包添加到发送缓存队列末尾，/include/net/tcp.h*/

    if (sk->sk_send_head == NULL) {         /*如果 sk->sk_send_head 为 NULL，发送头数据包*/
        sk->sk_send_head = skb;             /*数据包设为发送头数据包*/
```

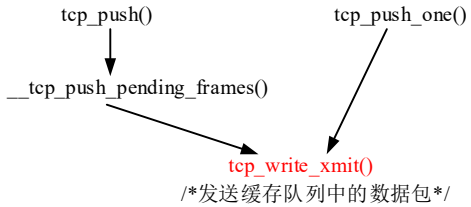
```
            if (tcp_sk(sk)->highest_sack == NULL)
                tcp_sk(sk)->highest_sack = skb;
        }
    }
```

__tcp_add_write_queue_tail(sk, skb)函数定义如下：

```
static inline void __tcp_add_write_queue_tail(struct sock *sk, struct sk_buff *skb)
{
    __skb_queue_tail(&sk->sk_write_queue, skb);     /*发送缓存队列*/
}
```

■推送数据包

在 tcp_sendmsg()函数中，如果用户数据分段完成或发送缓存队列中没有足够的内存将调用 **tcp_push()**函数发送缓存队列中的数据包，如果当前创建的数据包是套接字发送头数据包，将调用 **tcp_push_one()**函数发送单个数据包。函数调用关系简列如下图所示：



tcp_push()和 tcp_push_one()函数最终都是调用 **tcp_write_xmit()**函数发送缓存队列中的数据包。

下面先看一下 **tcp_push()**函数的实现，函数代码如下（/net/ipv4/tcp.c）：

```
static void tcp_push(struct sock *sk, int flags, int mss_now,int nonagle, int size_goal)
/*flags：发送消息标记，mss_now：MSS 值，nonagle：是否不支持 Nagle 算法，
*size_goal：数据包中用户数据最大长度。
*/
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *skb;

    if (!tcp_send_head(sk))       /*sk->sk_send_head == NULL，没有发送数据包，直接返回*/
        return;

    skb = tcp_write_queue_tail(sk);        /*发送队列最末尾数据包*/
    if (!(flags & MSG_MORE) || forced_push(tp))
        tcp_mark_push(tp, skb);
                                  /*设置控制结构 TCPHDR_PSH 标志，tp->pushed_seq = tp->write_seq*/
    tcp_mark_urg(tp, flags);

    if (tcp_should_autocork(sk, skb, size_goal)) {      /*是否自动启动抑制功能，/net/ipv4/tcp.c*/
        if (!test_bit(TSQ_THROTTLED, &tp->tsq_flags)) {
            NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPAUTOCORKING);
            set_bit(TSQ_THROTTLED, &tp->tsq_flags);
```

```
            }
            if (atomic_read(&sk->sk_wmem_alloc) > skb->truesize)   /*发送队列内存大于数据包大小*/
                return;          /*返回不发送*/
        }
        /*没启动抑制功能，发送数据包*/
        if (flags & MSG_MORE)
            nonagle = TCP_NAGLE_CORK;     /*TCP_NAGLE_CORK=2，/include/net/tcp.h*/

        __tcp_push_pending_frames(sk, mss_now, nonagle);
                                        /*发送缓存队列中的数据包，/net/ipv4/tcp_output.c*/
}
```

__tcp_push_pending_frames()函数定义如下：

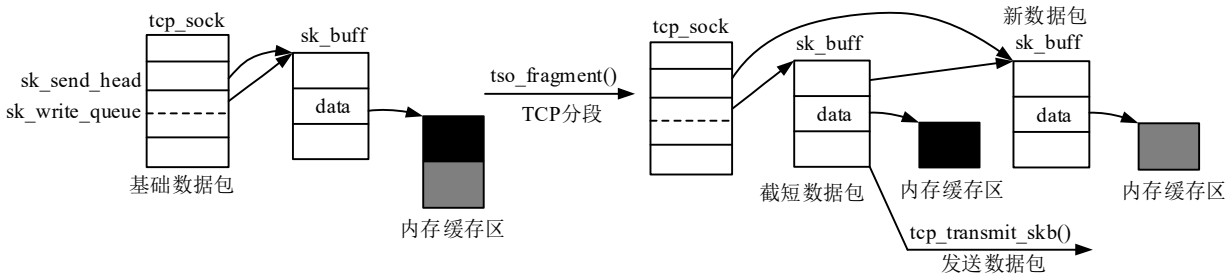
```
void __tcp_push_pending_frames(struct sock *sk, unsigned int cur_mss,int nonagle)
{
    if (unlikely(sk->sk_state == TCP_CLOSE))        /*连接关闭了，返回*/
        return;

    if (tcp_write_xmit(sk, cur_mss, nonagle, 0,sk_gfp_atomic(sk, GFP_ATOMIC)))
        tcp_check_probe_timer(sk);              /*推送队列中数据包，/net/ipv4/tcp_output.c*/
}
```

tcp_push_one()函数用于发送单个数据包，函数定义如下（/net/ipv4/tcp_output.c）：

```
void tcp_push_one(struct sock *sk, unsigned int mss_now)
{
    struct sk_buff *skb = tcp_send_head(sk);      /*发送头数据包*/
    BUG_ON(!skb || skb->len < mss_now);

    tcp_write_xmit(sk, mss_now, TCP_NAGLE_PUSH, 1, sk->sk_allocation);
}
```

3 接口函数

tcp_write_xmit()是发送套接字发送缓存队列中数据包的接口函数，函数内主要是一个循环，循环内从发送头数据包开始遍历队列中数据包（sk_buff 实例），检测接收窗口、拥塞窗口等值，以确定本次能发送数据的最大长度。如果数据包中数据长度超过了限制值，则需要对数据包中用户数据进行分段，超出部分数据放入新数据包，新数据包添加到原数据包之后（发送缓存队列），原数据包截短后发送到网络层。在下一次循环中再执行同样的操作。发送操作流程如下图所示：



如上图所示，假设数据包中内存缓存区用户数据超过了限制值，则在本次循环中按限制值截短内存缓

存区中数据，超出数据创建新数据包以容纳，新数据包添加到原数据包之后，本次循环只发送截短后的原数据包，下次循环将发送新数据包，执行同样的操作。如果数据包中存在分散数据块，分段操作将会更复杂一些，后面将详细介绍。如果数据包中用户数据较短，可在本次发送中一次性发出，则不需要进行分段。

在这里，分段前的数据包称之为基础数据包，截短后发送的数据包称为截短数据包，容纳本次不能发送数据的数据包称为新数据包。截短后数据包发送到网络层后，还存在于缓存队列中，只是发送头数据包指针指向新数据包，发送窗口后移。在收到确认应答后，才会释放发送过的数据包。

tcp_write_xmit()函数定义如下（/net/ipv4/tcp_output.c）：

```
static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now, int nonagle,int push_one, gfp_t gfp)
/*mss_now：MSS 值，分段数据长度，nonagle：是否禁止 Nagle 算法，gfp：分配标志，
*push_one：指示发送数据包数量（分段后），tcp_push()函数中为 0，tcp_push_one()函数中为 1。*/
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *skb;
    unsigned int tso_segs, sent_pkts;
    int cwnd_quota;          /*拥塞配额*/
    int result;
    bool is_cwnd_limited = false;        /*是否被拥塞窗口限制*/
    u32 max_segs;

    sent_pkts = 0;      /*记录发送数据包数量（TCP 分段后的数据包）*/

    if (!push_one) {        /*push_one 为 0 则执行 MTU 控测*/
        result = tcp_mtu_probe(sk);        /*执行 MTU 探测，/net/ipv4/tcp_output.c*/
        if (!result) {          /*返回 0 表示需要等待*/
            return false;
        } else if (result > 0) {     /*返回 1 表示发送了探测数据包*/
            sent_pkts = 1;
        }
    }

    max_segs = tcp_tso_autosize(sk, mss_now);
                            /*基础数据包 TCP 分段后的最大段数，/net/ipv4/tcp_output.c*/

    while ((skb = tcp_send_head(sk))) {        /*循环开始，skb 指向当前发送数据包*/
        unsigned int limit;

        tso_segs = tcp_init_tso_segs(skb, mss_now);    /*TCP_SKB_CB(skb)->tcp_gso_size 记录段数*/
                /*初始化数据包分段信息，按 mss_now 分段的段数，段长等，/net/ipv4/tcp_output.c*/
        BUG_ON(!tso_segs);

        if (unlikely(tp->repair) && tp->repair_queue == TCP_SEND_QUEUE) {
            skb_mstamp_get(&skb->skb_mstamp);
            goto repair;
        }

        cwnd_quota = tcp_cwnd_test(tp, skb);
```

```
                    /*拥塞窗口限制的本次可发送的分段数据包数量，/net/ipv4/tcp_output.c*/
if (!cwnd_quota) {    /*拥塞窗口限制不能发送数据包*/
    is_cwnd_limited = true;
    if (push_one == 2)
        cwnd_quota = 1;
    else
        break;
}

if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now)))
                        /*第一个分段是否在发送窗口内（可发送），/net/ipv4/tcp_output.c*/
    break;

if (tso_segs == 1) {      /*数据包分段数为 1（不分段）*/
    if (unlikely(!tcp_nagle_test(tp, skb, mss_now,(tcp_skb_is_last(sk, skb) ?
                                                    nonagle : TCP_NAGLE_PUSH))))
        break;
} else {          /*数据包分段数大于 1，需要分段*/
    if (!push_one &&tcp_tso_should_defer(sk, skb, &is_cwnd_limited,max_segs))
        /*检查拥塞窗口、发送窗口限制，是否需要延时发送，是则跳出循环，
        *否则发送数据，is_cwnd_limited 非 0 表示受拥塞窗口限制，/net/ipv4/tcp_output.c。
        */
        break;
}

limit = mss_now;          /*分段数据长度*/
if (tso_segs > 1 && !tcp_urg_mode(tp))        /*分段数大于 1，非紧急数据*/
    limit = tcp_mss_split_point(sk, skb, mss_now,min_t(unsigned int,cwnd_quota,max_segs),
                                                                    nonagle);
                                /*确定本次能发送的用户数据长度，/net/ipv4/tcp_output.c*/
if (skb->len > limit &&unlikely(tso_fragment(sk, skb, limit, mss_now, gfp)))
                    /*TCP 数据包分段，基础数据包一分为二，/net/ipv4/tcp_output.c*/
    break;

limit = max(2 * skb->truesize, sk->sk_pacing_rate >> 10);
limit = min_t(u32, limit, sysctl_tcp_limit_output_bytes);

if (atomic_read(&sk->sk_wmem_alloc) > limit) {
    set_bit(TSQ_THROTTLED, &tp->tsq_flags);
    smp_mb__after_atomic();
    if (atomic_read(&sk->sk_wmem_alloc) > limit)
        break;
}

if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp)))
                        /*发送单个分段后的数据包到网络层，/net/ipv4/tcp_output.c*/
```

```
                break;

repair:
        tcp_event_new_data_sent(sk, skb);          /*/net/ipv4/tcp_output.c*/
                /*更新发送头数据包指针（sk->sk_send_head）、统计发送数据包等*/
        tcp_minshall_update(tp, mss_now, skb);
        sent_pkts += tcp_skb_pcount(skb);      /*发送的 TCP 数据包数量，按 TCP 分段数据包*/

        if (push_one)      /*push_one 非 0，跳出循环，结束发送（只发送一个分段后的数据包）*/
            break;
    }     /*发送数据包循环结束*/

    if (likely(sent_pkts)) {        /*如果发送了数据包*/
        if (tcp_in_cwnd_reduction(sk))
            tp->prr_out += sent_pkts;

        /* Send one loss probe per tail loss episode. */
        if (push_one != 2)
            tcp_schedule_loss_probe(sk);
        tcp_cwnd_validate(sk, is_cwnd_limited);
                        /*设置套接字是否受拥塞窗口限制等参数，/net/ipv4/tcp_output.c*/
        return false;       /*返回假，表示发出了数据包*/
    }
    return (push_one == 2) || (!tp->packets_out && tcp_send_head(sk));
}
```

tcp_write_xmit()函数根据 mss_now 值确定基础数据包的分段数量，然后根据拥塞窗口、发送窗口确定本次能发送的数据长度，依此调用 **tso_fragment()**函数对基础数据包进行分段，对其一分为二，后一个数据包放在原数据包之后，原基础数据包截短后通过 tcp_transmit_skb()函数发送到网络层。如果基础数据包用户数据较短，在拥塞窗口和发送窗口内，则不需要对数据包进行分段，直接调用 tcp_transmit_skb()函数发送到网络层。

下面介绍基础数据包分段函数 tso_fragment()的实现，tcp_transmit_skb()函数在后面再介绍，其它拥塞窗口、发送窗口检测函数比较简单，请读者自行阅读。

■TCP 分段

tso_fragment()函数根据本次能发送的用户数据长度，对基础数据包一分为二，后一个数据包插入到基础数据包之后（缓存队列中）。tso_fragment()函数定义如下（/net/ipv4/tcp_output.c）：

```
static int tso_fragment(struct sock *sk, struct sk_buff *skb, unsigned int len,unsigned int mss_now,
                                                                    gfp_t gfp)
/*sk：套接字 sock 实例，skb：基础数据包，len：限制的发送数据长度，mss_now：MSS 值*/
{
    struct sk_buff *buff;
    int nlen = skb->len - len;     /*分段后剩下的用户数据长度（字节数）*/
    u8 flags;

    if (skb->len != skb->data_len)      /*如果数据包中存在分散数据块或分段数据包*/
```

```
        return tcp_fragment(sk, skb, len, mss_now, gfp);      /*TCP 分段，/net/ipv4/tcp_output.c*/

    /*数据包中不存在分散数据块或分段数据包，只对数据包内存缓存区中数据分段*/
    buff = sk_stream_alloc_skb(sk, 0, gfp, true);       /*创建新数据包，buff 指向新数据包*/
    if (unlikely(!buff))
        return -ENOMEM;

    sk->sk_wmem_queued += buff->truesize;       /*增加队列缓存长度*/
    sk_mem_charge(sk, buff->truesize);       /*减小 sk->sk_forward_alloc 值，/include/net/sock.h*/
    buff->truesize += nlen;       /*增加新数据包长度*/
    skb->truesize -= nlen;       /*减小原数据包长度*/

    /*设置新数据包中的序列号等*/
    TCP_SKB_CB(buff)->seq = TCP_SKB_CB(skb)->seq + len;
    TCP_SKB_CB(buff)->end_seq = TCP_SKB_CB(skb)->end_seq;
    TCP_SKB_CB(skb)->end_seq = TCP_SKB_CB(buff)->seq;

    /* PSH 和 FIN 标志设置到第 2 个数据包*/
    flags = TCP_SKB_CB(skb)->tcp_flags;
    TCP_SKB_CB(skb)->tcp_flags = flags & ~(TCPHDR_FIN | TCPHDR_PSH);
    TCP_SKB_CB(buff)->tcp_flags = flags;

    /* This packet was never sent out yet, so no SACK bits. */
    TCP_SKB_CB(buff)->sacked = 0;

    buff->ip_summed = skb->ip_summed = CHECKSUM_PARTIAL;
    skb_split(skb, buff, len);      /*拆分数据包，见下文，/net/core/skbuff.c*/
    tcp_fragment_tstamp(skb, buff);

    /* Fix up tso_factor for both original and new SKB. */
    tcp_set_skb_tso_segs(skb, mss_now);
    tcp_set_skb_tso_segs(buff, mss_now);

    /* Link BUFF into the send queue. */
    __skb_header_release(buff);       /*设置无报头标志*/
    tcp_insert_write_queue_after(skb, buff, sk);      /*新数据包添加到原数据包之后*/

    return 0;
}
```

tso_fragment()函数根据原数据包是否存在分散数据块，分别进行不同的处理。如果存在分散数据块，则调用 tcp_fragment()函数进行分段；如果不存在分散数据块，则创建新数据包，对新数据包进行初始化，然后调用 **skb_split()**函数对原数据包进行拆分，最后将新数据包添加到原数据包之后。

tcp_fragment()函数在对数据包进行拆分时，也会调用 skb_split()函数，因此下面先看 tcp_fragment()函数的实现，后面再介绍 skb_split()函数的实现。

tcp_fragment()函数用于对存在分散数据块的数据包进行分段，函数定义如下（/net/ipv4/tcp_output.c）：

```
int tcp_fragment(struct sock *sk, struct sk_buff *skb, u32 len,unsigned int mss_now, gfp_t gfp)
```

```
/*len：限制长度，mss_now：MSS 值*/
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *buff;
    int nsize, old_factor;
    int nlen;
    u8 flags;

    if (WARN_ON(len > skb->len))
        return -EINVAL;

    nsize = skb_headlen(skb) - len;     /*skb_headlen(skb)表示内存缓存区中数据长度，
                                        *不算分散数据块和分段数据块，
                                        *nsize 表示内存缓存区中数据截短后剩下的数据长度*/
    if (nsize < 0)      /*内存缓存区中数据没有超过限制值*/
        nsize = 0;

    if (skb_unclone(skb, gfp))
        return -ENOMEM;

    buff = sk_stream_alloc_skb(sk, nsize, gfp, true); /*分配新数据包，nsize 为内存缓存区中数据长度*/
    if (!buff)
        return -ENOMEM;    /* We'll just try again later. */

    sk->sk_wmem_queued += buff->truesize;     /*增加套接字发送队列长度*/
    sk_mem_charge(sk, buff->truesize);
    nlen = skb->len - len - nsize;
    buff->truesize += nlen;       /*新数据包数据长度*/
    skb->truesize -= nlen;        /*原数据包数据长度*/

    /*设置各序列号*/
    TCP_SKB_CB(buff)->seq = TCP_SKB_CB(skb)->seq + len;
    TCP_SKB_CB(buff)->end_seq = TCP_SKB_CB(skb)->end_seq;
    TCP_SKB_CB(skb)->end_seq = TCP_SKB_CB(buff)->seq;

    /* PSH and FIN should only be set in the second packet. */
    flags = TCP_SKB_CB(skb)->tcp_flags;
    TCP_SKB_CB(skb)->tcp_flags = flags & ~(TCPHDR_FIN | TCPHDR_PSH);
    TCP_SKB_CB(buff)->tcp_flags = flags;
    TCP_SKB_CB(buff)->sacked = TCP_SKB_CB(skb)->sacked;

    if (!skb_shinfo(skb)->nr_frags && skb->ip_summed != CHECKSUM_PARTIAL) {
        /* Copy and checksum data tail into the new buffer. */
        buff->csum = csum_partial_copy_nocheck(skb->data + len,skb_put(buff, nsize),nsize, 0);
        skb_trim(skb, len);
        skb->csum = csum_block_sub(skb->csum, buff->csum, len);
```

```
    } else {
        skb->ip_summed = CHECKSUM_PARTIAL;
        skb_split(skb, buff, len);     /*拆分数据包，见下文，/net/core/skbuff.c*/
    }

    buff->ip_summed = skb->ip_summed;
    buff->tstamp = skb->tstamp;
    tcp_fragment_tstamp(skb, buff);

    old_factor = tcp_skb_pcount(skb);   /*截短数据包中分段数量*/

    /* Fix up tso_factor for both original and new SKB. */
    tcp_set_skb_tso_segs(skb, mss_now);     /*设置分段数量*/
    tcp_set_skb_tso_segs(buff, mss_now);

    /*数据包已发出，调整数据包计数值*/
    if (!before(tp->snd_nxt, TCP_SKB_CB(buff)->end_seq)) {
        int diff = old_factor - tcp_skb_pcount(skb) -tcp_skb_pcount(buff);
        if (diff)
            tcp_adjust_pcount(sk, skb, diff);
    }

    __skb_header_release(buff);
    tcp_insert_write_queue_after(skb, buff, sk);       /*新数据包添加到原数据包之后*/

    return 0;
}
```

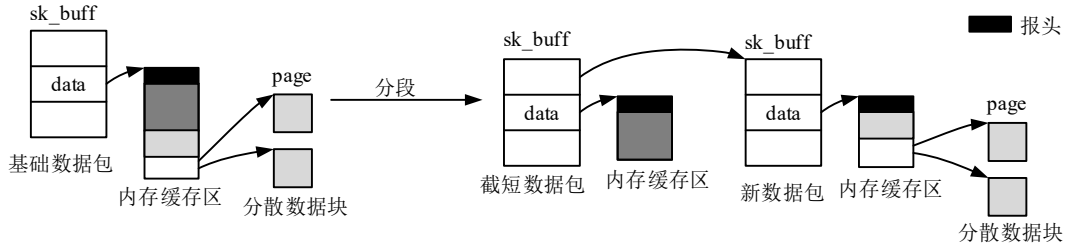
tcp_fragment()函数的主要工作也是对基础数据包进行分段，调用 skb_split()函数对数据包进行拆分，最后将新数据包添加到基础数据包之后。下面将简要介绍 skb_split()函数的实现。

■拆分数据包

skb_split()函数用于拆分数据包，函数定义如下（/net/core/skbuff.c）：

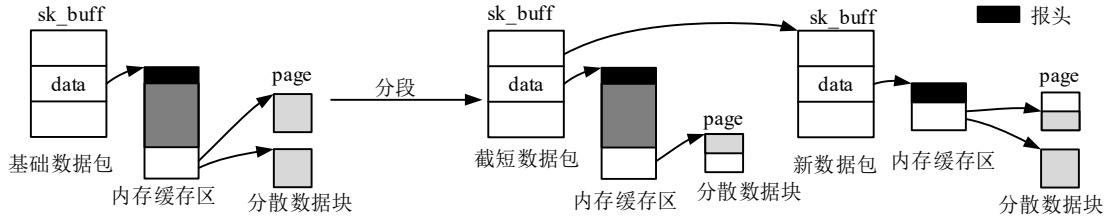
```
void skb_split(struct sk_buff *skb, struct sk_buff *skb1, const u32 len)
/*skb：基础数据包，skb1：新数据包，len：数据限制长度*/
{
    int pos = skb_headlen(skb);     /*基础数据包内存缓存区中用户数据长度*/

    skb_shinfo(skb1)->tx_flags = skb_shinfo(skb)->tx_flags & SKBTX_SHARED_FRAG;
    if (len < pos)     /*需要对内存缓存区进行拆分*/
        skb_split_inside_header(skb, skb1, len, pos);      /*/net/core/skbuff.c*/
    else          /*不需要对内存缓存区进行拆分*/
        skb_split_no_header(skb, skb1, len, pos);          /*/net/core/skbuff.c*/
}
```

skb_split()函数根据基础数据包内存缓存区中数据是否超过限制值，调用不同的函数对数据包进行拆分。当需要对基础数据包内存缓存区进行拆分时，调用 skb_split_inside_header()函数，拆分结果如下图所示：



基础数据包内存缓存区中超出限制值部分和分散数块（如果存在）被转移到新数据包。

当基础数据包内存缓存区中数据没有超过限制值或为 0 时，调用 skb_split_no_header()函数，拆分结果如下图所示：



截短数据包中包含内存缓存区中数据，并截取了分散数据块（可能是多个数据块）中若干的数据，以使数据总长度达到限制值，分散数据块中其它数据转移到新数据包（不是复制，只是调整内存缓存区中分散数据块信息）。

skb_split_inside_header()和 skb_split_no_header()函数比较简单，请读者自行阅读源代码。

4 发送单个数据包

tcp_transmit_skb()函数用于将一个已分段好的 TCP 数据包发送到网络层，这是一个接口函数，不止是在 tcp_write_xmit()函数中被调用。tcp_transmit_skb()函数定义如下（/net/ipv4/tcp_output.c）：

```
static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int clone_it,gfp_t gfp_mask)
/*skb：发送数据包，clone_it：此处为 1*/
{
    const struct inet_connection_sock *icsk = inet_csk(sk);
    struct inet_sock *inet;
    struct tcp_sock *tp;
    struct tcp_skb_cb *tcb;
    struct tcp_out_options opts;       /*TCP 选项*/
    unsigned int tcp_options_size, tcp_header_size;
    struct tcp_md5sig_key *md5;
    struct tcphdr *th;
    int err;

    BUG_ON(!skb || !tcp_skb_pcount(skb));

    if (clone_it) {       /*是否使用克隆数据包，此处使用，原数据包要留在发送队列*/
        skb_mstamp_get(&skb->skb_mstamp);
        if (unlikely(skb_cloned(skb)))       /*已经是克隆的，复制*/
            skb = pskb_copy(skb, gfp_mask);       /*复制数据包*/
        else
            skb = skb_clone(skb, gfp_mask);       /*克隆数据包*/
```

```
        if (unlikely(!skb))
            return -ENOBUFS;
}

inet = inet_sk(sk);
tp = tcp_sk(sk);          /*tcp_sock*/
tcb = TCP_SKB_CB(skb);        /*控制结构*/
memset(&opts, 0, sizeof(opts));

if (unlikely(tcb->tcp_flags & TCPHDR_SYN))      /*是 SYN 数据包，设置 MSS 选项值等*/
        tcp_options_size = tcp_syn_options(sk, skb, &opts, &md5);        /*/net/ipv4/tcp_output.c*/
else                    /*设置选择确认选项等，/net/ipv4/tcp_output.c*/
        tcp_options_size = tcp_established_options(sk, skb, &opts,&md5);    /*设置 TCP 选项*/
tcp_header_size = tcp_options_size + sizeof(struct tcphdr);          /*TCP 报头长度*/

if (tcp_packets_in_flight(tp) == 0)          /*还在网络中传输的数据包数量为 0*/
        tcp_ca_event(sk, CA_EVENT_TX_START);          /*调用拥塞控制算法 cwnd_event()函数*/

skb->ooo_okay = sk_wmem_alloc_get(sk) < SKB_TRUESIZE(1);

skb_push(skb, tcp_header_size);        /*开辟 TCP 头部空间*/
skb_reset_transport_header(skb);        /*记录 TCP 报头位置*/

skb_orphan(skb);
skb->sk = sk;
skb->destructor = skb_is_tcp_pure_ack(skb) ? sock_wfree : tcp_wfree;      /*设置析构函数*/
skb_set_hash_from_sk(skb, sk);
atomic_add(skb->truesize, &sk->sk_wmem_alloc);      /*增加套接字内存值*/

/*写入 TCP 报头和校验和，每个分段后的数据包都有 TCP 报头，不同于 UDP*/
th = tcp_hdr(skb);
th->source      = inet->inet_sport;          /*源端口号*/
th->dest        = inet->inet_dport;         /*目的端口号*/
th->seq         = htonl(tcb->seq);          /*序列号*/
th->ack_seq     = htonl(tp->rcv_nxt);       /*确认号*/
*(((__be16 *)th) + 6)     = htons(((tcp_header_size >> 2) << 12) |tcb->tcp_flags);      /*标志*/

if (unlikely(tcb->tcp_flags & TCPHDR_SYN)) {
        th->window    = htons(min(tp->rcv_wnd, 65535U));      /*在 SYN 数据包中设置接收窗口值*/
} else {
        th->window    = htons(tcp_select_window(sk));       /*其它数据包设置接收窗口值*/
                                                            /*/net/ipv4/tcp_output.c*/
}
th->check               = 0;
th->urg_ptr             = 0;
```

```
        if (unlikely(tcp_urg_mode(tp) && before(tcb->seq, tp->snd_up))) {
            ...
        }

        tcp_options_write((__be32 *)(th + 1), tp, &opts);        /*写入 TCP 选项*/
        skb_shinfo(skb)->gso_type = sk->sk_gso_type;          /*写入 GSO 类型*/
        if (likely((tcb->tcp_flags & TCPHDR_SYN) == 0))       /*SYN 数据包*/
            tcp_ecn_send(sk, skb, tcp_header_size);     /*根据情况设置 ECE 标志位*/

    #ifdef CONFIG_TCP_MD5SIG
        ...
    #endif

        icsk->icsk_af_ops->send_check(sk, skb);        /*地址簇操作结构定义的检查函数*/

        if (likely(tcb->tcp_flags & TCPHDR_ACK))        /*带应答*/
            tcp_event_ack_sent(sk, tcp_skb_pcount(skb));   /*停止 icsk->icsk_delack_timer 定时器*/
                                                           /*/include/net/inet_connection_sock.h*/
        if (skb->len != tcp_header_size)        /*发送了用户数据*/
            tcp_event_data_sent(tp, sk);     /*复位拥塞窗口，/net/ipv4/tcp_output.c*/

        if (after(tcb->end_seq, tp->snd_nxt) || tcb->seq == tcb->end_seq)
            TCP_ADD_STATS(sock_net(sk), TCP_MIB_OUTSEGS,tcp_skb_pcount(skb));

        tp->segs_out += tcp_skb_pcount(skb);       /*发出数据包段数*/
        skb_shinfo(skb)->gso_segs = tcp_skb_pcount(skb);       /*设置到共享结构中成员*/
        skb_shinfo(skb)->gso_size = tcp_skb_mss(skb);
        skb->tstamp.tv64 = 0;
        memset(skb->cb, 0, max(sizeof(struct inet_skb_parm),sizeof(struct inet6_skb_parm)));

        err = icsk->icsk_af_ops->queue_xmit(sk, skb, &inet->cork.fl);   /*调用地址簇操作结构中的函数*/

        ...      /*错误处理*/
        tcp_enter_cwr(sk);       /*拥塞状态进入 CWR 状态，/net/ipv4/tcp_input.c*/
        return net_xmit_eval(err);        /*是否通知拥塞，拥塞返回 0，/include/linux/netdevice.h*/
    }
```

以上在 tcp_transmit_skb()函数代码中添加了注释，调用的函数请读者自行阅读源代码，函数最后调用地址簇操作结构中的 queue_xmit()函数发送数据包。

在创建 TCP 套接字 tcp_sock 实例时，将调用 proto 实例的 init()函数初始化实例，对于 TCP 套接字此函数为 tcp_v4_init_sock()，定义如下（/net/ipv4/tcp_ipv4.c）：

```
static int tcp_v4_init_sock(struct sock *sk)
{
    struct inet_connection_sock *icsk = inet_csk(sk);

    tcp_init_sock(sk);
```

```
    icsk->icsk_af_ops = &ipv4_specific;      /*地址簇操作结构实例*/
            /*赋值 inet_connection_sock_af_ops 实例，/net/ipv4/tcp_ipv4.c*/
    ...
    return 0;
}
```

ipv4_specific 为 IPv4 地址簇操作结构实例 inet_connection_sock_af_ops 结构体实例，定义如下：

```
const struct inet_connection_sock_af_ops ipv4_specific = {
    .queue_xmit        = ip_queue_xmit,        /*向网络层发送数据包的函数*/
    ...
};
```

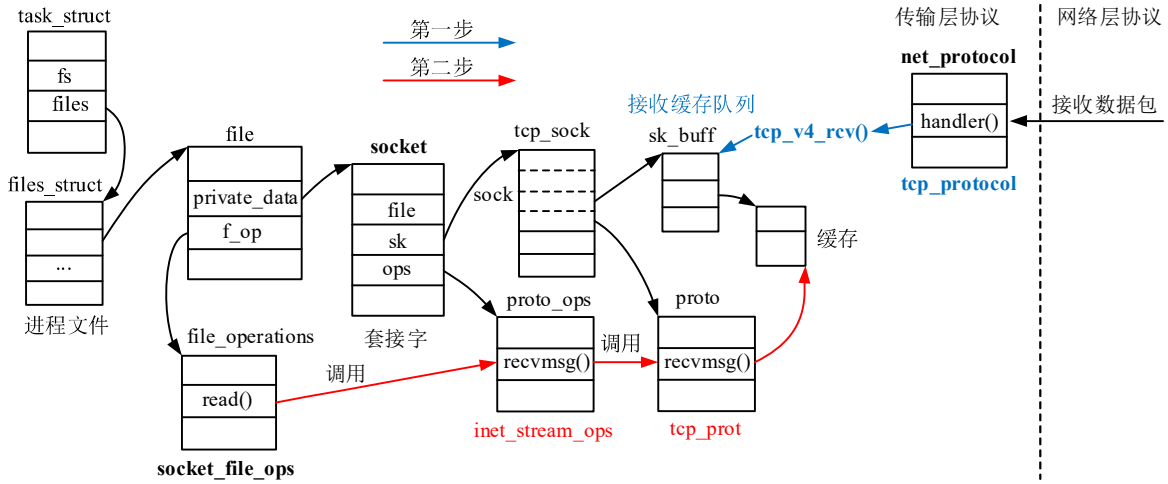
以上实例中 queue_xmit()函数为 ip_queue_xmit()，此函数为数据包进入网络层的入口函数，在下一节讲 IPv4 网络层代码时，将介绍此函数。

12.8.9 接收数据

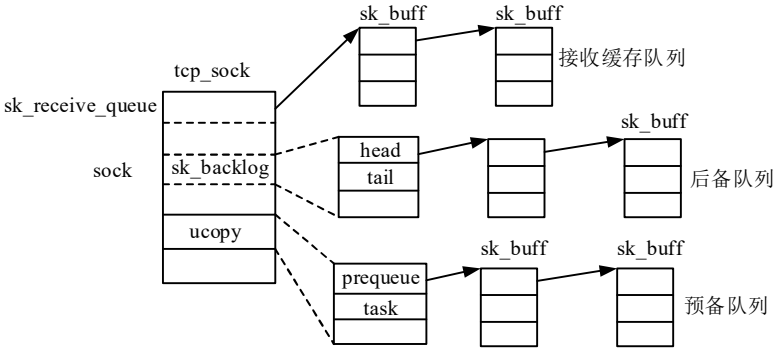
TCP 从网络层接收数据包，通过确认应答、重传、流量控制、拥塞控制等机制，保证套接字接收缓存队列中是完整正确按序排列的数据包。用户进程通过 read()、recv()等系统调用从接收缓存队列中读取数据。

1 概述

用户进程接收 TCP 数据与 UDP 类似，也分两步，如下图所示，一是 TCP 将接收数据包添加到套接字接收缓存队列，二是用户进程从接收缓存队列中读取数据。



在套接字 tcp_sock 实例中，存在 3 个与接收相关的数据包队列，分别是接收缓存队列，预备队列和后备队列，如下图所示。



在套接字未被锁定，用户进程没有读取套接字中数据时，TCP 接收到的数据包将添加到接收队列。用

户进程在读取套接字数据时，将锁定套接字，读取完成后解锁套接字。在套接字被锁定期间，TCP 接收到的数据包将添加到后备队列。

用户进程在读取数据时，当接收缓存队列中数据读完后，会将后备队列中数据包移入接收队列，然后再从接收队列读取数据包。

（1）如果 sysctl_tcp_low_latency 系统控制参数非 0（初始值为 0），或者 tcp_sock.ucopy.task 为 NULL，即没有指定复制数据的用户进程，数据包将直接添加到接收缓存队列。

（2）如果（1）中条件不成立，数据包将先添加到预备队列，然后添加到接收缓存队列。数据包添加到预备队列再添加到接收缓存队列，是 TCP 在接收数据包时连续完成的动作，而不像后备队列，需要解锁套接字时才移动。

2 用户读取数据

用户进程读取套接字数据的 read()、recv()等系统调用将调用套接字关联 proto_ops 实例中的 recvmsg()函数。TCP 套接字关联 proto_ops 实例 inet_stream_ops 中的 recvmsg()函数为 inet_recvmsg()，此函数内将调用套接字关联 proto 实例中的 recvmsg()函数。

TCP 套接字关联 proto 实例 tcp_prot 中的 recvmsg()函数为 tcp_recvmsg()，此函数负责从 TCP 套接字接收缓存队列数据包中读取数据返回给用户进程。

tcp_recvmsg()函数定义如下（/net/ipv4/tcp.c）：

```
int tcp_recvmsg(struct sock *sk, struct msghdr *msg, size_t len, int nonblock,int flags, int *addr_len)
/*nonblock：是否非阻塞读操作*/
{
    struct tcp_sock *tp = tcp_sk(sk);
    int copied = 0;       /*记录复制的数据量*/
    u32 peek_seq;
    u32 *seq;
    unsigned long used;
    int err;
    int target;           /*本次最少需要读取的字节数*/
    long timeo;
    struct task_struct *user_recv = NULL;        /*接收数据进程，初始为 NULL*/
    struct sk_buff *skb, *last;
    u32 urg_hole = 0;

    if (unlikely(flags & MSG_ERRQUEUE))
        return inet_recv_error(sk, msg, len, addr_len);

    if (sk_can_busy_loop(sk) && skb_queue_empty(&sk->sk_receive_queue) &&
                                        (sk->sk_state == TCP_ESTABLISHED))
                        /*如果接收队列为空，且可轮询网络接口，选择 NET_RX_BUSY_POLL 选项*/
        sk_busy_loop(sk, nonblock);       /*轮询网络接口，接收数据，/include/net/busy_poll.h*/

    lock_sock(sk);     /*锁定套接字*/

    err = -ENOTCONN;
    if (sk->sk_state == TCP_LISTEN)
        goto out;
```

```
timeo = sock_rcvtimeo(sk, nonblock);     /*读操作超时时间*/

if (flags & MSG_OOB)
      goto recv_urg;

if (unlikely(tp->repair)) {
      ...
}

seq = &tp->copied_seq;       /*指向已复制最后数据（加 1）的序列号*/
if (flags & MSG_PEEK) {
      peek_seq = tp->copied_seq;
      seq = &peek_seq;
}

target = sock_rcvlowat(sk, flags & MSG_WAITALL, len); /*至少需要读取的字节数，目标字节数*/

do {              /*do 循环开始，从接收队列数据包中复制数据*/
      u32 offset;
      if (tp->urg_data && tp->urg_seq == *seq) {       /*处理紧急数据*/
            ...
      }

      last = skb_peek_tail(&sk->sk_receive_queue);        /*指向接收队列末尾数据包*/
      skb_queue_walk(&sk->sk_receive_queue, skb) {
                                          /*从队列头遍历数据包，查找可复制数据的数据包*/
            /* *seq 记录的是本次复制数据起始字节的序列号，本次复制可能从一个新的数据包中
            *复制数据，也可能从上一个数据没有全部复制完的数据包中复制数据。
            *因此，*seq 应大于或等于当前复制数据包的序列号（起始字节的序号）。
            */
            last = skb;
            ...      /*检查序列号*/
            offset = *seq - TCP_SKB_CB(skb)->seq;       /*本次复制数据在数据包中偏移量*/
            if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_SYN)
                  offset--;          /*设置了 SYN 标志，偏移位置要减 1*/
            if (offset < skb->len)     /*可在当前数据包中复制数据*/
                  goto found_ok_skb;        /*跳转至 found_ok_skb 处复制数据*/
            if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN)
                  goto found_fin_ok;            /*设置了 FIN 标志的数据包*/
            ...
      }

      /*复制的数据不够，但接收缓存队列中已没有可复制的数据包，执行以下操作*/
      if (copied >= target && !sk->sk_backlog.tail)       /*复制数据数量达到最小值且后备队列空*/
            break;            /*跳出循环，不复制了*/
```

```
/*未达到最小复制量，或后备队列非空，需要继续想办法复制数据*/
if (copied) {        /*已复制了一些数据，满足下面条件时，结束复制，不满足继续往下执行*/
    if (sk->sk_err ||sk->sk_state == TCP_CLOSE ||
        (sk->sk_shutdown & RCV_SHUTDOWN) ||!timeo ||signal_pending(current))
        break;
} else {      /*没有复制到任何数据*/
    ...    /*满足某些条件，不复制了，跳出循环*/
}

/*继续想办法复制数据*/
tcp_cleanup_rbuf(sk, copied);      /*清空接收缓存，发送 ACK 应答等，/net/ipv4/tcp.c*/

if (!sysctl_tcp_low_latency && tp->ucopy.task == user_recv) {      /*安装等待接收进程*/
    if (!user_recv && !(flags & (MSG_TRUNC | MSG_PEEK))) {     /*user_recv 为 NULL*/
        user_recv = current;        /*设为当前进程*/
        tp->ucopy.task = user_recv;
        tp->ucopy.msg = msg;
    }

    tp->ucopy.len = len;     /*还需要读取的数据量*/

    WARN_ON(tp->copied_seq != tp->rcv_nxt &&!(flags & (MSG_PEEK | MSG_TRUNC)));

    if (!skb_queue_empty(&tp->ucopy.prequeue))        /*预备队列非空*/
        goto do_prequeue;      /*跳转至 do_prequeue 处，将预备队列数据包移至接收队列*/
}

/*预备队列为空*/
if (copied >= target) {      /*复制数据量达到了最小值，将后备队列中数据包移入接收队列*/
    release_sock(sk);        /*/net/core/sock.c*/
            /*解除套接字锁定，调用 sk_backlog_rcv(sk, skb)接收数据包，唤醒等待进程等*/
    lock_sock(sk);        /*继续锁定套接字*/
} else {       /*未达到复制数据量最小值，且预备队列为空，睡眠等待*/
    sk_wait_data(sk, &timeo, last);     /*睡眠等待接收缓存队列中数据包，/net/core/sock.c*/
                                        /*接收队列收到数据包或超时，将唤醒进程*/
}

if (user_recv) {        /*有进程在等待接收数据，将预备队列中数据包移入接收队列*/
    int chunk;
    if ((chunk = len - tp->ucopy.len) != 0) {
        ...       /*更新统计量*/
        len -= chunk;
        copied += chunk;
    }
```

```
            if (tp->rcv_nxt == tp->copied_seq &&!skb_queue_empty(&tp->ucopy.prequeue)) {
        do_prequeue:
                tcp_prequeue_process(sk);     /*将预备队列中数据包移入接收队列，/net/ipv4/tcp.c*/
                                              /*对预备队列中数据包调用 sk_backlog_rcv(sk, skb)*/
                if ((chunk = len - tp->ucopy.len) != 0) {
                    ...         /*更新统计量*/
                    len -= chunk;
                    copied += chunk;
                }
            }
        }

        if ((flags & MSG_PEEK) &&(peek_seq - copied - urg_hole != tp->copied_seq)) {
            ...  /*输出信息*/
            peek_seq = tp->copied_seq;
        }
        continue;       /*经过一番处理后，继续循环，从接收缓存列中取数据包，复制数据*/

    found_ok_skb:       /*从数据包中复制数据*/
        used = skb->len - offset;       /*数据包中可复制数据长度（字节数）*/
        if (len < used)
            used = len;

        if (tp->urg_data) {         /*处理紧急数据*/
            ...
        }

        if (!(flags & MSG_TRUNC)) {
            err = skb_copy_datagram_msg(skb, offset, msg, used);          /*复制数据到用户空间*/
            ...  /*错误处理*/
        }

        *seq += used;       /*更新 tp->copied_seq 值*/
        copied += used;     /*已复制数据长度*/
        len -= used;        /*还需要复制数据长度*/

        tcp_rcv_space_adjust(sk);     /*重新计算接收缓存空间 tp->rcvq_space，/net/ipv4/tcp_input.c*/

skip_copy:
        if (tp->urg_data && after(tp->copied_seq, tp->urg_seq)) {
            ...
        }

        if (used + offset < skb->len)       /*当前数据包中数据没有复制完，继续循环*/
            continue;
```

```
            /*当前数据包中数据复制完了，都交给用户进程了，需要释放数据包*/
            if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN)
                goto found_fin_ok;
            if (!(flags & MSG_PEEK))
                sk_eat_skb(sk, skb);   /*释放数据包，从队列移除，释放 sk_buff 实例，/include/net/sock.h*/
            continue;      /*继续循环，遍历下一个数据包*/

        found_fin_ok:
            ++*seq;
            if (!(flags & MSG_PEEK))
                sk_eat_skb(sk, skb);        /*收到带 FIN 标志的数据包，释放它*/
            break;        /*跳出循环*/
        } while (len > 0);     /*len 为 0 时，  do 循环结束，复制操作完成*/

        if (user_recv) {
            if (!skb_queue_empty(&tp->ucopy.prequeue)) {       /*预备队列不为空*/
                int chunk;

                tp->ucopy.len = copied > 0 ? len : 0;
                tcp_prequeue_process(sk);        /*将预备队列中数据包移入接收队列*/

                if (copied > 0 && (chunk = len - tp->ucopy.len) != 0) {
                    ...     /*更新统计量*/
                    len -= chunk;
                    copied += chunk;
                }
            }

            tp->ucopy.task = NULL;       /*设为初始值*/
            tp->ucopy.len = 0;
        }
        tcp_cleanup_rbuf(sk, copied);       /*清空接收缓存区*/

        release_sock(sk);       /*解锁套接字*/
        return copied;          /*返回复制的字节数*/
        ...
    }
```

tcp_recvmsg()函数参数 len 表示用户进程希望读取到的字节数，局部变量 target 表示最少需要读取的字节数。tcp_recvmsg()函数中首先锁定套接字，在读取过程中到达的数据包将添加到后备队列。函数内是一个循环，循环体遍历接收队列中数据包，从中复制数据到用户空间，数据包中数据复制完后，将被释放，随后遍历下一个数据包，复制到足够数据后，将跳出循环，解锁套接字，返回读取的字节数。

在循环体内，如果接收缓存队列中数据包遍历完后，还没有读取到足够的数据，当前进程将设置到 tp->ucopy.task 成员。如果读取数据量达到了最小值，且后备队列不为空，则将后备队列中数据移入接收队列，解锁套接字，随后再锁定套接字（在此解锁与锁定之间接收到的数据包将添加到预备队列）。循环继续从接收队列中读取数据。如果读取数据量未达到最小值，进程将进入睡眠，当接收队列收到数据包或超时，进程将被唤醒，继续循环读取数据。

在将后备队列中数据包移入接收队列时，添加到预备队列中的数据包，在下次循环前也将添加到接收队列。总之，后备队列和预备队列中数据包交替添加到接收队列，并保证数据包的按序。

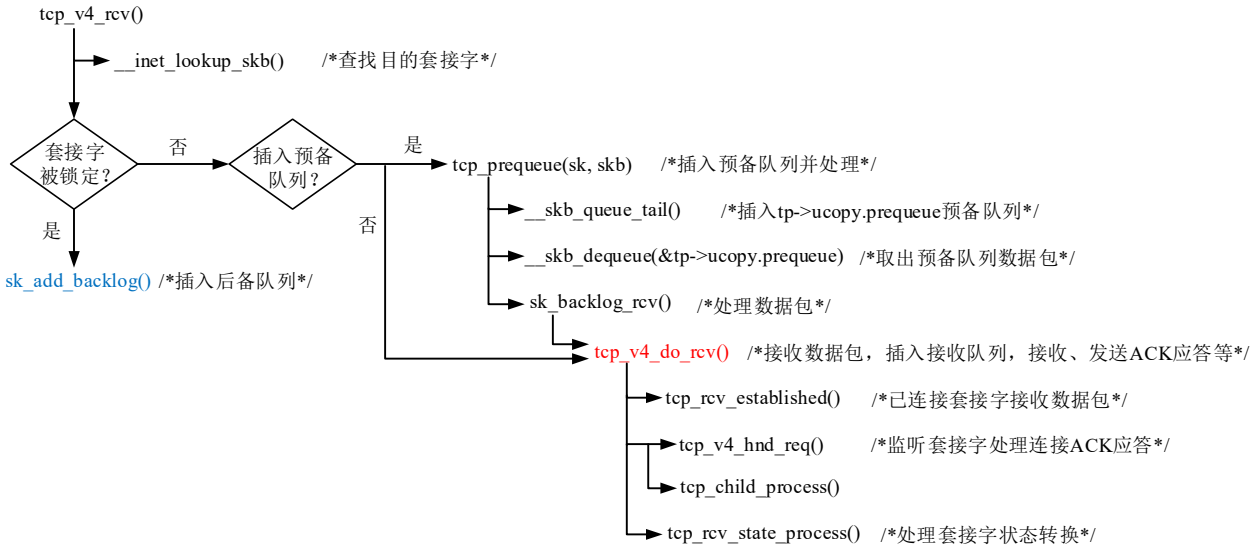
处理后备队列和预备队列时，都是调用 sk_backlog_rcv(sk, skb)函数将数据包添加到接收队列，后面将介绍此函数的实现。

3 TCP 接收函数

TCP 协议定义了 net_protocol 实例 tcp_protocol 并在初始化函数 inet_init()中注册，实例简列如下：

```
static const struct net_protocol tcp_protocol = {      /*/net/ipv4/af_inet.c*/
    .early_demux  =    tcp_v4_early_demux,
    .handler  =    tcp_v4_rcv,    /*接收数据包函数*/
    .err_handler    =    tcp_v4_err,
    .no_policy      =    1,
    .netns_ok=     1,
    .icmp_strict_tag_validation = 1,
};
```

tcp_v4_rcv()函数用于 TCP 接收网络层传递的数据包，函数调用关系简列如下：



函数内首先调用__inet_lookup_skb()函数以报头中指示的 IP 地址和端口号查找目的套接字，然后判断套接字是否被用户进程锁定。如果被锁定则调用 sk_add_backlog()函数，将数据包添加到后备队列末尾。如果未被锁定，则调用 tcp_prequeue()函数检查是否将数据包添加到预备队列并处理，如果是则数据包在此函数中处理，否则交由 tcp_v4_do_rcv()函数处理。在 tcp_prequeue()函数中先将数据包插入到预备队列，然后也是调用 tcp_v4_do_rcv()函数处理预备队列中的数据包。

tcp_v4_do_rcv()函数根据套接字所处的状态，对数据包调用不同的处理函数。对于已建立连接的套接字调用 tcp_rcv_established()函数进行处理。对于监听套接字，调用 tcp_v4_hnd_req()函数接收连接 ACK 应答，创建连接套接字，然后调用 tcp_child_process()函数由连接套接字处理连接 ACK 应答。套接字处于其它状态时，调用 tcp_rcv_state_process()函数接收数据包，完成套接字状态的转换。

tcp_v4_rcv()函数定义在/net/ipv4/tcp_ipv4.c 文件内，代码如下：

```
int tcp_v4_rcv(struct sk_buff *skb)
{
    const struct iphdr *iph;
    const struct tcphdr *th;
    struct sock *sk;
```

```
    int ret;
    struct net *net = dev_net(skb->dev);        /*网络命名空间*/

    if (skb->pkt_type != PACKET_HOST)        /*不是传递给本机的数据包，丢弃*/
        goto discard_it;

    TCP_INC_STATS_BH(net, TCP_MIB_INSEGS);

    if (!pskb_may_pull(skb, sizeof(struct tcphdr)))     /*检查、调整数据包的 TCP 报头*/
        goto discard_it;

    th = tcp_hdr(skb);        /*指向 TCP 报头*/

    if (th->doff < sizeof(struct tcphdr) / 4)
        goto bad_packet;
    if (!pskb_may_pull(skb, th->doff * 4))
        goto discard_it;

    if (skb_checksum_init(skb, IPPROTO_TCP, inet_compute_pseudo))
        goto csum_error;

    th = tcp_hdr(skb);          /*重获 TCP 报头*/
    iph = ip_hdr(skb);        /*指向 IP 报头*/
    memmove(&TCP_SKB_CB(skb)->header.h4, IPCB(skb),sizeof(struct inet_skb_parm));
    barrier();

    TCP_SKB_CB(skb)->seq = ntohl(th->seq);     /*序列号*/
    TCP_SKB_CB(skb)->end_seq = (TCP_SKB_CB(skb)->seq + th->syn + th->fin +
                                                    skb->len - th->doff * 4);       /*结束序列号*/
    TCP_SKB_CB(skb)->ack_seq = ntohl(th->ack_seq);       /*确认号*/
    TCP_SKB_CB(skb)->tcp_flags = tcp_flag_byte(th);       /*标记字段*/
    TCP_SKB_CB(skb)->tcp_tw_isn = 0;         /*起始序列号*/
    TCP_SKB_CB(skb)->ip_dsfield = ipv4_get_dsfield(iph);
    TCP_SKB_CB(skb)->sacked   = 0;

    sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th->dest);       /*查找目的 TCP 套接字*/
    if (!sk)
        goto no_tcp_socket;

process:
    if (sk->sk_state == TCP_TIME_WAIT)
        goto do_time_wait;

    if (unlikely(iph->ttl < inet_sk(sk)->min_ttl)) {
        NET_INC_STATS_BH(net, LINUX_MIB_TCPMINTTLDROP);
        goto discard_and_relse;
```

```
    }

    if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb))
        goto discard_and_relse;

#ifdef CONFIG_TCP_MD5SIG
    ...
#endif

    nf_reset(skb);

    if (sk_filter(sk, skb))        /*调用套接字定义的过滤函数*/
        goto discard_and_relse;

    sk_incoming_cpu_update(sk);
    skb->dev = NULL;

    bh_lock_sock_nested(sk);
    tcp_sk(sk)->segs_in += max_t(u16, 1, skb_shinfo(skb)->gso_segs);      /*数据段数*/
    ret = 0;
    if (!sock_owned_by_user(sk)) {      /*如果进程没有锁定套接字，/include/net/sock.h*/
        if (!tcp_prequeue(sk, skb))      /*插入预备队列并处理，/net/ipv4/tcp_ipv4.c*/
            ret = tcp_v4_do_rcv(sk, skb);      /*接收数据包函数，/net/ipv4/tcp_ipv4.c*/
    } else if (unlikely(sk_add_backlog(sk, skb,sk->sk_rcvbuf + sk->sk_sndbuf))) {
        bh_unlock_sock(sk);      /*套接字被锁定，数据包添加到后备队列末尾，/include/net/sock.h*/
        NET_INC_STATS_BH(net, LINUX_MIB_TCPBACKLOGDROP);
        goto discard_and_relse;
    }
    bh_unlock_sock(sk);
    sock_put(sk);
    return ret;
    ...
}
```

■预备队列

数据包添加到预备队列并处理的 tcp_prequeue()函数定义如下（/net/ipv4/tcp_ipv4.c）：

```
bool tcp_prequeue(struct sock *sk, struct sk_buff *skb)
{
    struct tcp_sock *tp = tcp_sk(sk);

    if (sysctl_tcp_low_latency || !tp->ucopy.task)
        return false;          /*sysctl_tcp_low_latency 非 0 或 tp->ucopy.task 为 NULL 将返回 flase*/

    if (skb->len <= tcp_hdrlen(skb) &&skb_queue_len(&tp->ucopy.prequeue) == 0)
        return false;
```

```
        if (likely(sk->sk_rx_dst))
            skb_dst_drop(skb);
        else
            skb_dst_force(skb);

        __skb_queue_tail(&tp->ucopy.prequeue, skb);       /*数据包添加到预备队列末尾*/
        tp->ucopy.memory += skb->truesize;
        if (tp->ucopy.memory > sk->sk_rcvbuf) {
            struct sk_buff *skb1;
            BUG_ON(sock_owned_by_user(sk));
            /*遍历预备队列中数据包*/
            while ((skb1 = __skb_dequeue(&tp->ucopy.prequeue)) != NULL) {
                sk_backlog_rcv(sk, skb1);     /*调用 sk->sk_backlog_rcv(sk, skb)函数，/include/net/sock.h*/
                ...          /*更新统计量*/
            }
            tp->ucopy.memory = 0;
        } else if (skb_queue_len(&tp->ucopy.prequeue) == 1) {
            wake_up_interruptible_sync_poll(sk_sleep(sk),POLLIN | POLLRDNORM | POLLRDBAND);
                                                        /*唤醒在套接字上睡眠进程*/
            if (!inet_csk_ack_scheduled(sk))
                inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,(3 * tcp_rto_min(sk)) / 4,
                                                                    TCP_RTO_MAX);
                                                        /*复位发送定时器*/
        }
        return true;      /*返回真，数据包由预备队列处理*/
    }
```

数据包添加到预备队列末尾后，如果 tp->ucopy.memory > sk->sk_rcvbuf 则遍历预备队列，对队列中每个数据包调用 sk_backlog_rcv()函数，函数定义如下（/include/net/sock.h）：

```
    static inline int sk_backlog_rcv(struct sock *sk, struct sk_buff *skb)
    {
        if (sk_memalloc_socks() && skb_pfmemalloc(skb))
            return   __sk_backlog_rcv(sk, skb);        /*/net/core/sock.c*/

        return sk->sk_backlog_rcv(sk, skb);
    }
```

__sk_backlog_rcv(sk, skb)函数定义如下（/net/core/sock.c）：

```
    int __sk_backlog_rcv(struct sock *sk, struct sk_buff *skb)
    {
        int ret;
        unsigned long pflags = current->flags;
        BUG_ON(!sock_flag(sk, SOCK_MEMALLOC));

        current->flags |= PF_MEMALLOC;
        ret = sk->sk_backlog_rcv(sk, skb);
```

```
    tsk_restore_flags(current, pflags, PF_MEMALLOC);

    return ret;
}
```

由以上函数可知，预备队列中的数据包将由 sk->sk_backlog_rcv()函数处理。在创建 IPv4 套接字时，将对此函数指针赋值，如下所示：

```
tatic int inet_create(struct net *net, struct socket *sock, int protocol,int kern)
{
    ...
    sk->sk_backlog_rcv = sk->sk_prot->backlog_rcv;
    ...
}
```

sk->sk_backlog_rcv()函数被赋值为 proto 实例中的 backlog_rcv()函数。tcp_prot 实例中指定的函数如下：

```
struct proto tcp_prot = {
    ...
    .backlog_rcv   = tcp_v4_do_rcv,        /*接收后备队列中的数据包*/
    ...
}
```

最终，不管是添加到预备队列的数据包，还是直接处理的数据包，都由 tcp_v4_do_rcv()函数接收，下面将介绍此函数的实现。

■接收函数

tcp_v4_do_rcv()函数根据套接字当前状态，调用不同的函数接收数据包，定义如下（/net/ipv4/tcp_ipv4.c）：

```
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
    struct sock *rsk;

    if (sk->sk_state == TCP_ESTABLISHED) {     /*快速路径，套接字已建立连接*/
        struct dst_entry *dst = sk->sk_rx_dst;

        sock_rps_save_rxhash(sk, skb);
        sk_mark_napi_id(sk, skb);
        if (dst) {
            if (inet_sk(sk)->rx_dst_ifindex != skb->skb_iif ||!dst->ops->check(dst, 0)) {
                dst_release(dst);
                sk->sk_rx_dst = NULL;
            }
        }
        tcp_rcv_established(sk, skb, tcp_hdr(skb), skb->len);
                                    /*已连接套接字接收数据包，/net/ipv4/tcp_input.c*/
        return 0;
    }

    if (tcp_checksum_complete(skb))
```

```
            goto csum_err;

        if (sk->sk_state == TCP_LISTEN) {     /*欢迎套接字*/
            struct sock *nsk = tcp_v4_hnd_req(sk, skb);        /*处理连接 ACK 报文段，创建连接套接字*/
            if (!nsk)
                goto discard;

            if (nsk != sk) {
                sock_rps_save_rxhash(nsk, skb);
                sk_mark_napi_id(sk, skb);
                if (tcp_child_process(sk, nsk, skb)) {     /*连接套接字处理连接 ACK 报文段*/
                    rsk = nsk;
                    goto reset;
                }
                return 0;
            }
        } else
            sock_rps_save_rxhash(sk, skb);
        /*套接字处于其它状态时，接收数据包*/
        if (tcp_rcv_state_process(sk, skb, tcp_hdr(skb), skb->len)) {
                                                    /*套接字状态转换，/net/ipv4/tcp_input.c*/
            rsk = sk;
            goto reset;
        }
        return 0;
        ...
    }
```

如果套接字已建立连接，调用 **tcp_rcv_established()**函数接收数据包。监听状态套接字接收数据包的处理在讲解连接操作时已经介绍过了，这里不再重复了。

套接字处于其它状态时，调用 **tcp_rcv_state_process()**函数接收数据包，主要完成套接字状态转换。下面介绍以上两个函数的实现。

4 已连接套接字接收数据包

tcp_rcv_established()函数定义如下（/net/ipv4/tcp_input.c）：

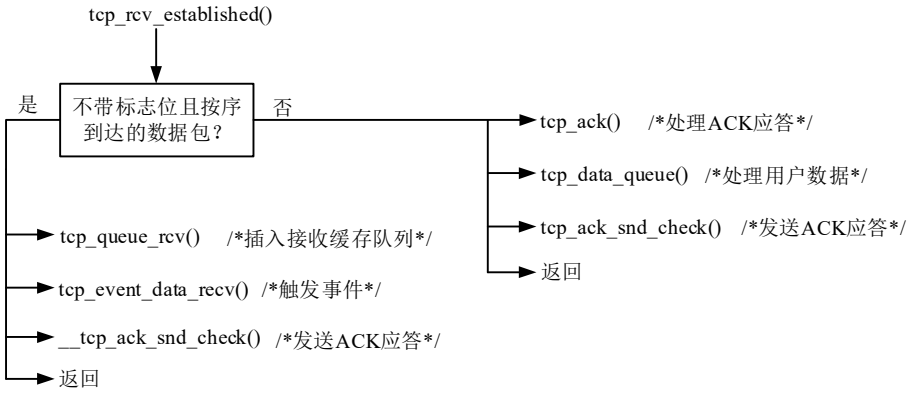
```
void tcp_rcv_established(struct sock *sk, struct sk_buff *skb,const struct tcphdr *th, unsigned int len)
{
    struct tcp_sock *tp = tcp_sk(sk);

    if (unlikely(!sk->sk_rx_dst))
        inet_csk(sk)->icsk_af_ops->sk_rx_dst_set(sk, skb);

    tp->rx_opt.saw_tstamp = 0;

    /*tcp_flag_word(th)表示 TCP 报头中第 4 个 32 位字（包含标志字段），TCP_HP_BITS 表示屏蔽
    *TCP_FLAG_PSH 标志位等，tp->pred_flags 通常为 0。
```

```
*下面 if 语句表示收到按序到达的数据包，报头不带标志位，且报头中确认号有效。
*/
if ((tcp_flag_word(th) & TCP_HP_BITS) == tp->pred_flags &&
    TCP_SKB_CB(skb)->seq == tp->rcv_nxt &&
    !after(TCP_SKB_CB(skb)->ack_seq, tp->snd_nxt)) {      /*收到按序到达的有效数据包*/

    int tcp_header_len = tp->tcp_header_len;        /*TCP 报头长度*/
    if (tcp_header_len == sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED) {
        if (!tcp_parse_aligned_timestamp(tp, th))        /*带时间截选项*/
            goto slow_path;

        if ((s32)(tp->rx_opt.rcv_tsval - tp->rx_opt.ts_recent) < 0)
            goto slow_path;
    }

    if (len <= tcp_header_len) {      /*数据包不带用户数据，只有报头信息*/
        if (len == tcp_header_len) {      /*只含报头的有效数据包*/
            if (tcp_header_len ==(sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED) &&
                                                    tp->rcv_nxt == tp->rcv_wup)
                tcp_store_ts_recent(tp);

            tcp_ack(sk, skb, 0);
            __kfree_skb(skb);
            tcp_data_snd_check(sk);
            return;
        } else {        /*数据包无效，丢弃*/
            TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_INERRS);
            goto discard;
        }
    } else {        /*数据包含用户数据*/
        int eaten = 0;
        bool fragstolen = false;

        if (tp->ucopy.task == current &&tp->copied_seq == tp->rcv_nxt &&
            len - tcp_header_len <= tp->ucopy.len &&sock_owned_by_user(sk)) {
            ...        /*当前进程是预处理进程*/
        }
        if (!eaten) {
            if (tcp_checksum_complete_user(sk, skb))       /*检查校验和*/
                goto csum_error;

            if ((int)skb->truesize > sk->sk_forward_alloc)     /*数据包长度大于已分配内存长度*/
                goto step5;

            if (tcp_header_len ==(sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED) &&
                                                    tp->rcv_nxt == tp->rcv_wup)
```

```
                tcp_store_ts_recent(tp);

                tcp_rcv_rtt_measure_ts(sk, skb);     /*更新 RTT 时间，/net/ipv4/tcp_input.c*/
                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPHPHITS);
                eaten = tcp_queue_rcv(sk, skb, tcp_header_len,&fragstolen);   /*/net/ipv4/tcp_input.c*/
                                        /*将数据包添加到接收缓存队列 sk->sk_receive_queue 末尾*/
            }

            tcp_event_data_recv(sk, skb);
                                /*接收到数据事件，更新 MSS、RTT 值等，/net/ipv4/tcp_input.c*/

            if (TCP_SKB_CB(skb)->ack_seq != tp->snd_una) {
                                                /*收到累积应答的确认号，用于发送缓存队列*/
                tcp_ack(sk, skb, FLAG_DATA);
                tcp_data_snd_check(sk);
                if (!inet_csk_ack_scheduled(sk))
                    goto no_ack;
            }

            __tcp_ack_snd_check(sk, 0); /*确定是否要发送应答，或延时应答，/net/ipv4/tcp_input.c*/
        no_ack:
            if (eaten)
                kfree_skb_partial(skb, fragstolen);
            sk->sk_data_ready(sk);      /*唤醒在套接字上睡眠的进程*/
            return;      /*返回*/
        }    /*带用户数据的数据包处理完毕*/
    }   /*if 语句结束*/

    /*以下处理报头带标志位、非按序到达等数据包*/
slow_path:      /*慢速路径*/
    if (len < (th->doff << 2) || tcp_checksum_complete_user(sk, skb))
        goto csum_error;

    if (!th->ack && !th->rst && !th->syn)   /*需带有 ACK、RST 或 FIN 标志位*/
        goto discard;

    if (!tcp_validate_incoming(sk, skb, th, 1))     /*检查数据包有效性，/net/ipv4/tcp_input.c*/
        return;
step5:
    if (tcp_ack(sk, skb, FLAG_SLOWPATH | FLAG_UPDATE_TS_RECENT) < 0)
                                /*报头中应答信息，/net/ipv4/tcp_input.c*/
        goto discard;

    tcp_rcv_rtt_measure_ts(sk, skb);

    /*处理紧急数据*/
```

```
        tcp_urg(sk, skb, th);

        /*处理用户数据*/
        tcp_data_queue(sk, skb);    /*/net/ipv4/tcp_input.c*/
        tcp_data_snd_check(sk);    /*调用 tcp_push_pending_frames()发送数据等，/net/ipv4/tcp_input.c*/
        tcp_ack_snd_check(sk);    /*调用__tcp_ack_snd_check()发送 ACK 应答等，/net/ipv4/tcp_input.c*/
        return;
        ...    /*校验和出错，释放数据包*/
    }
```

tcp_rcv_established()函数执行流程简列如下图所示：



tcp_rcv_established()函数通过一个 if 语句判断接收到的数据包是否是不带标志位且按序到达的，根据判断结果执行不同的操作。

对不带标志位且是按序到达的数据包的主要操作有：

（1）调用 tcp_queue_rcv()函数将数据包插入到接收缓存队列末尾。

（2）调用 tcp_event_data_recv()函数触发接收数据事件，如计算 RTT 值、MSS 值、更新拥塞阀值等。

（3）调用__tcp_ack_snd_check()函数确认是否要发送 ACK 应答，或延迟应答。

对带标志位或非按序到达的数据包的主要操作有：

（1）调用 tcp_ack()函数处理 ACK 应答。

（2）调用 tcp_data_queue()函数处理用户数据，如将按序到达的数据包插入接收缓存队列，处理带 FIN 标志位数据包等。

（3）调用 tcp_ack_snd_check()函数发达 ACK 应答，此函数内将调用__tcp_ack_snd_check()函数。

下面介绍 tcp_rcv_established()函数中调用的主要函数的实现。

■插入接收队列

如果接收到的是不带标志位且按序到达的数据包，将调用 tcp_queue_rcv()函数将数据包插入到套接字接收缓存队列末尾，函数定义如下（/net/ipv4/tcp_input.c）：

```
static int __must_check tcp_queue_rcv(struct sock *sk, struct sk_buff *skb, int hdrlen,bool *fragstolen)
{
    int eaten;
    struct sk_buff *tail = skb_peek_tail(&sk->sk_receive_queue);    /*接收缓存队列末尾数据包*/

    __skb_pull(skb, hdrlen);    /*跳过 TCP 报头*/
    eaten = (tail &&tcp_try_coalesce(sk, tail, skb, fragstolen)) ? 1 : 0;
                                        /*尝试合并到最末尾数据包，成功返回 1，否则返回 0*/
    tcp_rcv_nxt_update(tcp_sk(sk), TCP_SKB_CB(skb)->end_seq);
```

```
                                                    /*更新 tp->rcv_nxt 值等，/net/ipv4/tcp_input.c*/
        if (!eaten) {           /*不能合并，添加到接收缓存队列末尾*/
            __skb_queue_tail(&sk->sk_receive_queue, skb);
            skb_set_owner_r(skb, sk);
        }
        return eaten;       /*返回是否合并到接收队列末尾数据包*/
    }
```

■接收数据事件

在接收到用户数据后，tcp_rcv_established()函数将调用 tcp_event_data_recv()函数触发接收数据事件，函数定义如下（/net/ipv4/tcp_input.c）：

```
    static void tcp_event_data_recv(struct sock *sk, struct sk_buff *skb)
    {
        struct tcp_sock *tp = tcp_sk(sk);
        struct inet_connection_sock *icsk = inet_csk(sk);
        u32 now;

        inet_csk_schedule_ack(sk);      /*设置 inet_csk(sk)->icsk_ack.pending 的 ICSK_ACK_SCHED 标记*/
        tcp_measure_rcv_mss(sk, skb);    /*设置 icsk->icsk_ack.rcv_mss 值，/net/ipv4/tcp_input.c*/
        tcp_rcv_rtt_measure(tp);    /*计算 RTT 值，设置 tp->rcv_rtt_est 成员，/net/ipv4/tcp_input.c*/

        now = tcp_time_stamp;

        if (!icsk->icsk_ack.ato) {      /*ACK 应答时间为 0*/
            tcp_incr_quickack(sk);
            icsk->icsk_ack.ato = TCP_ATO_MIN;    /*最小延时应答时间，/include/net/tcp.h*/
        } else {      /*重置 ACK 应答延时时间*/
            int m = now - icsk->icsk_ack.lrcvtime;

            if (m <= TCP_ATO_MIN / 2) {
                icsk->icsk_ack.ato = (icsk->icsk_ack.ato >> 1) + TCP_ATO_MIN / 2;
            } else if (m < icsk->icsk_ack.ato) {
                icsk->icsk_ack.ato = (icsk->icsk_ack.ato >> 1) + m;
                if (icsk->icsk_ack.ato > icsk->icsk_rto)
                    icsk->icsk_ack.ato = icsk->icsk_rto;
            } else if (m > icsk->icsk_rto) {
                tcp_incr_quickack(sk);
                sk_mem_reclaim(sk);
            }
        }

        icsk->icsk_ack.lrcvtime = now;
        tcp_ecn_check_ce(tp, skb);

        if (skb->len >= 128)
```

```
        tcp_grow_window(sk, skb);     /*更新 tp->rcv_ssthresh 值，/net/ipv4/tcp_input.c*/
}
```

■处理应答信息

在 tcp_rcv_established()函数调用 tcp_ack()函数处理数据包报头中的应答信息，函数定义如下：

```
static int tcp_ack(struct sock *sk, const struct sk_buff *skb, int flag)       /*/net/ipv4/tcp_input.c*/
/*flag：标记，取值如 FLAG_DATA（包含用户数据），/net/ipv4/tcp_input.c*/
{
    struct inet_connection_sock *icsk = inet_csk(sk);
    struct tcp_sock *tp = tcp_sk(sk);
    struct tcp_sacktag_state sack_state;
    u32 prior_snd_una = tp->snd_una;          /*最早尚未确认的序列号*/
    u32 ack_seq = TCP_SKB_CB(skb)->seq;     /*数据包序列号*/
    u32 ack = TCP_SKB_CB(skb)->ack_seq;     /*确认号*/
    bool is_dupack = false;
    u32 prior_fackets;
    int prior_packets = tp->packets_out;
    const int prior_unsacked = tp->packets_out - tp->sacked_out;
    int acked = 0;     /*确认的数据包数量*/

    sack_state.first_sackt.v64 = 0;
    prefetchw(sk->sk_write_queue.next);

    if (before(ack, prior_snd_una)) {     /*收到过时的确认*/
        if (before(ack, prior_snd_una - tp->max_window)) {
            tcp_send_challenge_ack(sk, skb);
            return -1;
        }
        goto old_ack;
    }

    /*如果确认号是尚未发送的数据，忽略它*/
    if (after(ack, tp->snd_nxt))
        goto invalid_ack;

    if (icsk->icsk_pending == ICSK_TIME_EARLY_RETRANS ||
        icsk->icsk_pending == ICSK_TIME_LOSS_PROBE)
        tcp_rearm_rto(sk);          /*重置重传定时器，/net/ipv4/tcp_input.c*/

    if (after(ack, prior_snd_una)) {       /*收到的是有效的应答*/
        flag |= FLAG_SND_UNA_ADVANCED;
        icsk->icsk_retransmits = 0;          /*不需要重传*/
    }

    prior_fackets = tp->fackets_out;
```

```
if (flag & FLAG_UPDATE_TS_RECENT)
    tcp_replace_ts_recent(tp, TCP_SKB_CB(skb)->seq);

if (!(flag & FLAG_SLOWPATH) && after(ack, prior_snd_una)) {        /*非慢速路径，应答有效*/
    tcp_update_wl(tp, ack_seq);       /*tp->snd_wl1 = ack_seq，/include/net/tcp.h*/
    tcp_snd_una_update(tp, ack);      /*tp->snd_una = ack，/net/ipv4/tcp_input.c*/
    flag |= FLAG_WIN_UPDATE;

    tcp_in_ack_event(sk, CA_ACK_WIN_UPDATE);       /*/net/ipv4/tcp_input.c*/
                                /*调用拥塞控制算法 icsk->icsk_ca_ops->in_ack_event()函数*/
    NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPHPACKS);
} else {
    u32 ack_ev_flags = CA_ACK_SLOWPATH;

    if (ack_seq != TCP_SKB_CB(skb)->end_seq)
        flag |= FLAG_DATA;
    else
        NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPPUREACKS);

    flag |= tcp_ack_update_window(sk, skb, ack, ack_seq);     /*/net/ipv4/tcp_input.c*/
                                                        /*更新 snd_wnd、rcv_nxt 等值*/
    if (TCP_SKB_CB(skb)->sacked)     /*带选项确认信息*/
        flag |= tcp_sacktag_write_queue(sk, skb, prior_snd_una,&sack_state);

    if (tcp_ecn_rcv_ecn_echo(tp, tcp_hdr(skb))) {
        flag |= FLAG_ECE;
        ack_ev_flags |= CA_ACK_ECE;
    }

    if (flag & FLAG_WIN_UPDATE)
        ack_ev_flags |= CA_ACK_WIN_UPDATE;

    tcp_in_ack_event(sk, ack_ev_flags);
}

sk->sk_err_soft = 0;
icsk->icsk_probes_out = 0;
tp->rcv_tstamp = tcp_time_stamp;
if (!prior_packets)
    goto no_queue;

acked = tp->packets_out;
flag |= tcp_clean_rtx_queue(sk, prior_fackets, prior_snd_una,&sack_state);   /*/net/ipv4/tcp_input.c*/
                                                /*移除重传队列中已经应答的数据包*/
acked -= tp->packets_out;
```

```
    if (tcp_may_raise_cwnd(sk, flag))       /*是否执行增加拥塞窗口函数，/net/ipv4/tcp_input.c*/
        tcp_cong_avoid(sk, ack, acked);

    if (tcp_ack_is_dubious(sk, flag)) {
        is_dupack = !(flag & (FLAG_SND_UNA_ADVANCED | FLAG_NOT_DUP));
        tcp_fastretrans_alert(sk, acked, prior_unsacked,is_dupack, flag);
    }
    if (tp->tlp_high_seq)
        tcp_process_tlp_ack(sk, ack, flag);

    if ((flag & FLAG_FORWARD_PROGRESS) || !(flag & FLAG_NOT_DUP)) {
        struct dst_entry *dst = __sk_dst_get(sk);
        if (dst)
            dst_confirm(dst);
    }

    if (icsk->icsk_pending == ICSK_TIME_RETRANS)
        tcp_schedule_loss_probe(sk);
    tcp_update_pacing_rate(sk);
    return 1;

no_queue:
    if (flag & FLAG_DSACKING_ACK)
        tcp_fastretrans_alert(sk, acked, prior_unsacked,is_dupack, flag);

    if (tcp_send_head(sk))
        tcp_ack_probe(sk);

    if (tp->tlp_high_seq)
        tcp_process_tlp_ack(sk, ack, flag);
    return 1;

invalid_ack:
    SOCK_DEBUG(sk, "Ack %u after %u:%u\n", ack, tp->snd_una, tp->snd_nxt);
    return -1;

old_ack:        /*处理过时的应答*/
    if (TCP_SKB_CB(skb)->sacked) {
        flag |= tcp_sacktag_write_queue(sk, skb, prior_snd_una,&sack_state);
        tcp_fastretrans_alert(sk, acked, prior_unsacked,is_dupack, flag);
    }

    SOCK_DEBUG(sk, "Ack %u before %u:%u\n", ack, tp->snd_una, tp->snd_nxt);
    return 0;
}
```

■处理带标志或乱序到达数据包

tcp_data_queue()函数用于处理带标志位的数据包或非按序到达的数据包，函数定义如下：

```
static void tcp_data_queue(struct sock *sk, struct sk_buff *skb)      /*/net/ipv4/tcp_input.c*/
{
    struct tcp_sock *tp = tcp_sk(sk);
    int eaten = -1;
    bool fragstolen = false;

    if (TCP_SKB_CB(skb)->seq == TCP_SKB_CB(skb)->end_seq)      /*起始结束序列号相同，丢弃*/
        goto drop;

    skb_dst_drop(skb);
    __skb_pull(skb, tcp_hdr(skb)->doff * 4);

    tcp_ecn_accept_cwr(tp, skb);

    tp->rx_opt.dsack = 0;

    if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {      /*按序到达的数据包*/
        if (tcp_receive_window(tp) == 0)      /*检查接收窗口是否为 0，/include/net/tcp.h*/
            goto out_of_window;

        /*按序且接收窗口非零*/
        if (tp->ucopy.task == current &&tp->copied_seq == tp->rcv_nxt && tp->ucopy.len &&
                              sock_owned_by_user(sk) && !tp->urg_data) {
            ...          /*当前进程是预处理进程*/
        }

        if (eaten <= 0) {
queue_and_out:
            if (eaten < 0) {
                if (skb_queue_len(&sk->sk_receive_queue) == 0)
                    sk_forced_mem_schedule(sk, skb->truesize);
                else if (tcp_try_rmem_schedule(sk, skb, skb->truesize))
                    goto drop;
            }
            eaten = tcp_queue_rcv(sk, skb, 0, &fragstolen); /*将数据包添加到接收队列末尾或合并*/
        }

        tcp_rcv_nxt_update(tp, TCP_SKB_CB(skb)->end_seq);      /*设置 tp->rcv_nxt = seq 等*/
        if (skb->len)
            tcp_event_data_recv(sk, skb);      /*触发接收数据事件*/
        if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN)
            tcp_fin(sk);      /*处理 FIN 数据包，见下一节*/
```

```
        if (!skb_queue_empty(&tp->out_of_order_queue)) {     /*失序到达数据包队列非空*/
            tcp_ofo_queue(sk);
                        /*是否可以从失序数据包队列中移动数据包到接收队列，/net/ipv4/tcp_input.c*/
            if (skb_queue_empty(&tp->out_of_order_queue))
                inet_csk(sk)->icsk_ack.pingpong = 0;
        }

        if (tp->rx_opt.num_sacks)
            tcp_sack_remove(tp);

        tcp_fast_path_check(sk);

        if (eaten > 0)
            kfree_skb_partial(skb, fragstolen);
        if (!sock_flag(sk, SOCK_DEAD))
            sk->sk_data_ready(sk);       /*唤醒在套接字睡眠进程*/
        return;
    }    /*处理按序到达且在接收窗口内的数据包结束*/

    /*下面处理非按序到达或超出接收窗口的数据包*/
    if (!after(TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt)) {       /*收到重复数据包*/
        NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_DELAYEDACKLOST);
        tcp_dsack_set(sk, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq);

out_of_window:     /*超出接收窗口*/
        tcp_enter_quickack_mode(sk);
        inet_csk_schedule_ack(sk);
drop:
        __kfree_skb(skb);
        return;
    }

    if (!before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt + tcp_receive_window(tp)))
        goto out_of_window;

    tcp_enter_quickack_mode(sk);       /*进入快速应答模式*/

    if (before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {         /*收到跨越  rcv_next  的部分数据包*/
        /* Partial packet, seq < rcv_next < end_seq */
        SOCK_DEBUG(sk, "partial packet: rcv_next %X seq %X - %X\n",
                   tp->rcv_nxt, TCP_SKB_CB(skb)->seq,TCP_SKB_CB(skb)->end_seq);

        tcp_dsack_set(sk, TCP_SKB_CB(skb)->seq, tp->rcv_nxt);
        if (!tcp_receive_window(tp))
            goto out_of_window;
        goto queue_and_out;
```

```
        }

        tcp_data_queue_ofo(sk, skb);        /*将失序到达的数据包添加到失序队列，/net/ipv4/tcp_input.c*/
    }
```

■发送 ACK 应答

tcp_ack_snd_check()函数用于确定是否发送 ACK 应答或延时应答，定义如下（/net/ipv4/tcp_input.c）：

```
static inline void tcp_ack_snd_check(struct sock *sk)
{
    if (!inet_csk_ack_scheduled(sk)) {     /*已经发送了带用户数据的数据包，数据包中带应答，返回*/
        return;
    }
    __tcp_ack_snd_check(sk, 1);        /*发送专门的应答数据包，/net/ipv4/tcp_input.c*/
}
```

__tcp_ack_snd_check()函数定义如下：

```
static void __tcp_ack_snd_check(struct sock *sk, int ofo_possible)
{
    struct tcp_sock *tp = tcp_sk(sk);
    if (((tp->rcv_nxt - tp->rcv_wup) > inet_csk(sk)->icsk_ack.rcv_mss &&
         __tcp_select_window(sk) >= tp->rcv_wnd) ||tcp_in_quickack_mode(sk) ||
        (ofo_possible && skb_peek(&tp->out_of_order_queue))) {
        tcp_send_ack(sk);       /*立即发送 ACK 应答，/net/ipv4/tcp_output.c*/
    } else {
        tcp_send_delayed_ack(sk);       /*延时发送 ACK 应答，/net/ipv4/tcp_output.c*/
    }
}
```

tcp_send_ack(sk)函数用于发送不包含用户数据，只含应答信息的数据包，源代码请读者自行阅读。tcp_send_delayed_ack(sk)函数用于重置延时应答定时器 icsk->icsk_delack_timer，如果必要也会立即发送 ACK 应答，源代码请读者自行阅读。

5 套接字状态转换

如果套接字不是处于已建立连接状态或监听状态，tcp_v4_do_rcv()函数内将调用 tcp_rcv_state_process()函数接收数据包，主要工作是实现套接字状态的转换，函数定义如下（/net/ipv4/tcp_input.c）：

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb,const struct tcphdr *th, unsigned int len)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct inet_connection_sock *icsk = inet_csk(sk);
    struct request_sock *req;
    int queued = 0;
    bool acceptable;
    u32 synack_stamp;

    tp->rx_opt.saw_tstamp = 0;
```

```
switch (sk->sk_state) {
case TCP_CLOSE:
        goto discard;

case TCP_LISTEN:        /*监听套接字*/
        if (th->ack)
                return 1;

        if (th->rst)
                goto discard;

        if (th->syn) {        /*处理 SYN 连接请求*/
                if (th->fin)
                        goto discard;
                if (icsk->icsk_af_ops->conn_request(sk, skb) < 0)
                        return 1;
                kfree_skb(skb);
                return 0;
        }
        goto discard;

case TCP_SYN_SENT:        /*套接字发送 SYN 请求后，接收 SYN ACK 应答*/
        queued = tcp_rcv_synsent_state_process(sk, skb, th, len);
        if (queued >= 0)
                return queued;

        tcp_urg(sk, skb, th);
        __kfree_skb(skb);
        tcp_data_snd_check(sk);
        return 0;
}

req = tp->fastopen_rsk;
if (req) {
        ...
}

if (!th->ack && !th->rst && !th->syn)
        goto discard;

if (!tcp_validate_incoming(sk, skb, th, 0))
        return 0;

/*处理 ACK 应答信息*/
acceptable = tcp_ack(sk, skb, FLAG_SLOWPATH |FLAG_UPDATE_TS_RECENT) > 0;
```

```
switch (sk->sk_state) {
case TCP_SYN_RECV:      /*连接套接字接收对方的 ACK 应答，改变连接套接字状态*/
    if (!acceptable)
        return 1;
    if (req) {
        ...
    } else {
        synack_stamp = tp->lsndtime;
        icsk->icsk_af_ops->rebuild_header(sk);
        tcp_init_congestion_control(sk);          /*初始化拥塞控制算法*/

        tcp_mtup_init(sk);
        tp->copied_seq = tp->rcv_nxt;
        tcp_init_buffer_space(sk);
    }
    /*下面是初始化连接套接字*/
    smp_mb();
    tcp_set_state(sk, TCP_ESTABLISHED);       /*设为已建立连接状态*/
    sk->sk_state_change(sk);
    if (sk->sk_socket)
        sk_wake_async(sk, SOCK_WAKE_IO, POLL_OUT);       /*唤醒套接字上睡眠的进程*/

    tp->snd_una = TCP_SKB_CB(skb)->ack_seq;
    tp->snd_wnd = ntohs(th->window) << tp->rx_opt.snd_wscale;     /*发送窗口*/
    tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);
    tcp_synack_rtt_meas(sk, synack_stamp);

    if (tp->rx_opt.tstamp_ok)
        tp->advmss -= TCPOLEN_TSTAMP_ALIGNED;

    if (req) {
        tcp_rearm_rto(sk);
    } else
        tcp_init_metrics(sk);

    tcp_update_pacing_rate(sk);
    tp->lsndtime = tcp_time_stamp;

    tcp_initialize_rcv_mss(sk);
    tcp_fast_path_on(tp);
    break;

case TCP_FIN_WAIT1: {      /*主动关闭套接字发送 FIN 后，接收到对方发送的 ACK 应答*/
    struct dst_entry *dst;
    int tmo;
```

```
        if (req) {
                ...
        }
        if (tp->snd_una != tp->write_seq)
                break;

        tcp_set_state(sk, TCP_FIN_WAIT2);       /*状态设为 TCP_FIN_WAIT2*/
        sk->sk_shutdown |= SEND_SHUTDOWN;

        dst = __sk_dst_get(sk);
        if (dst)
                dst_confirm(dst);

        if (!sock_flag(sk, SOCK_DEAD)) {        /*套接字没有设置 SOCK_DEAD 标志*/
                sk->sk_state_change(sk);        /*唤醒套接字上睡眠的进程*/
                break;
        }

        /*套接字设置了 SOCK_DEAD 标志*/
        if (tp->linger2 < 0 ||(TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq &&
                               after(TCP_SKB_CB(skb)->end_seq - th->fin, tp->rcv_nxt))) {
                tcp_done(sk);
                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPABORTONDATA);
                return 1;
        }

        tmo = tcp_fin_time(sk);
        if (tmo > TCP_TIMEWAIT_LEN) {
                inet_csk_reset_keepalive_timer(sk, tmo - TCP_TIMEWAIT_LEN);
        } else if (th->fin || sock_owned_by_user(sk)) {
                inet_csk_reset_keepalive_timer(sk, tmo);
        } else {
                tcp_time_wait(sk, TCP_FIN_WAIT2, tmo);
                goto discard;
        }
        break;
}

case TCP_CLOSING:
        if (tp->snd_una == tp->write_seq) {
                tcp_time_wait(sk, TCP_TIME_WAIT, 0);
                goto discard;
        }
        break;

case TCP_LAST_ACK:      /*被动关闭套接字发送 FIN 后，接收到对方发送的 ACK 应答*/
```

```
            if (tp->snd_una == tp->write_seq) {
                tcp_update_metrics(sk);
                tcp_done(sk);       /*关闭套接字，释放套接字资源*/
                goto discard;
            }
            break;
        }

        tcp_urg(sk, skb, th);

        switch (sk->sk_state) {
        case TCP_CLOSE_WAIT:
        case TCP_CLOSING:
        case TCP_LAST_ACK:
            if (!before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt))
                break;
        case TCP_FIN_WAIT1:
        case TCP_FIN_WAIT2:
            if (sk->sk_shutdown & RCV_SHUTDOWN) {
                if (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq &&
                    after(TCP_SKB_CB(skb)->end_seq - th->fin, tp->rcv_nxt)) {
                    NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPABORTONDATA);
                    tcp_reset(sk);
                    return 1;
                }
            }

        case TCP_ESTABLISHED:
            tcp_data_queue(sk, skb);       /*接收带用户数据的数据包*/
            queued = 1;
            break;
        }

        /*主动关闭套接字处于 TIME-WAIT 状态，发送 ACK 应答*/
        if (sk->sk_state != TCP_CLOSE) {
            tcp_data_snd_check(sk);        /*清空发送缓存队列（发送出去）*/
            tcp_ack_snd_check(sk);         /*发送 ACK 应答*/
        }

    if (!queued) {
discard:
            __kfree_skb(skb);
        }
        return 0;
}
```

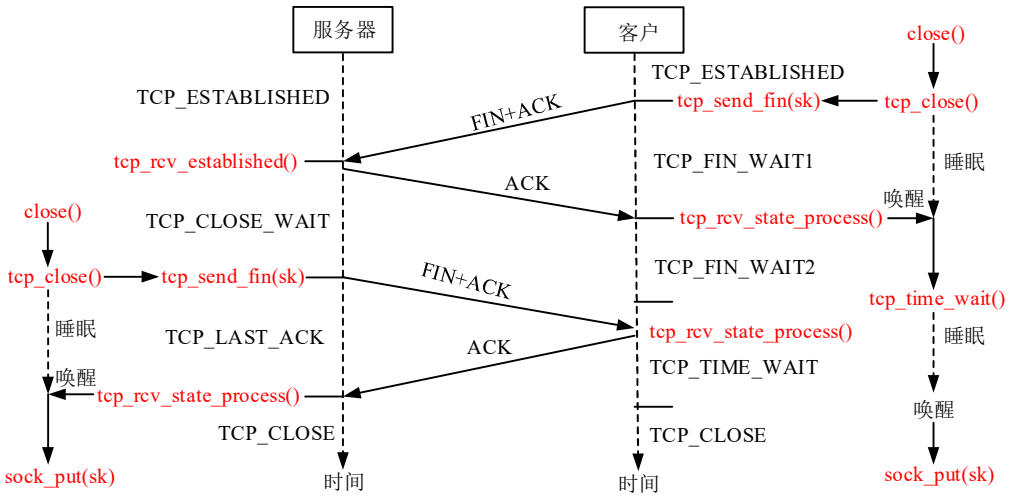
12.8.10 关闭连接

前面介绍了 TCP 通信双方连接的建立。连接建立后，双方套接字的状态都为 TCP_ESTABLISHED，此时双方可正常地进行数据传输了。通信结束后需要关闭连接，释放双方套接字连接资源。

本小节介绍连接的关闭，后面小节将介绍连接双方数据的传输。

1 概述

如下图所示，在关闭连接前客户和服务器套接字都处于 TCP_ESTABLISHED 状态。假设关闭连接操作由客户发起。



客户通过 close()系统调用关闭连接，系统调用中调用 tcp_close()函数完成关闭操作。tcp_close()函数内将套接字状态设为 TCP_FIN_WAIT1，通过 tcp_send_fin()函数发送 FIN 报文段（TCP 报头中 FIN、ACK 标志位置位，带序列号和确认号），随后进程进入睡眠状态。客户收到服务器发回的 ACK 报文段后，套接字设为 TCP_FIN_WAIT2 状态，唤醒睡眠进程。close()系统调用继续执行 tcp_time_wait()函数，设置套接字状态为 TCP_TIME_WAIT，等待一段时间后（睡眠），进程被唤醒调用 sock_put()函数释放套接字。客户套接字在 TCP_TIME_WAIT 状态接收响应服务器 FIN 报文段（主要是发送 ACK 报文段）。

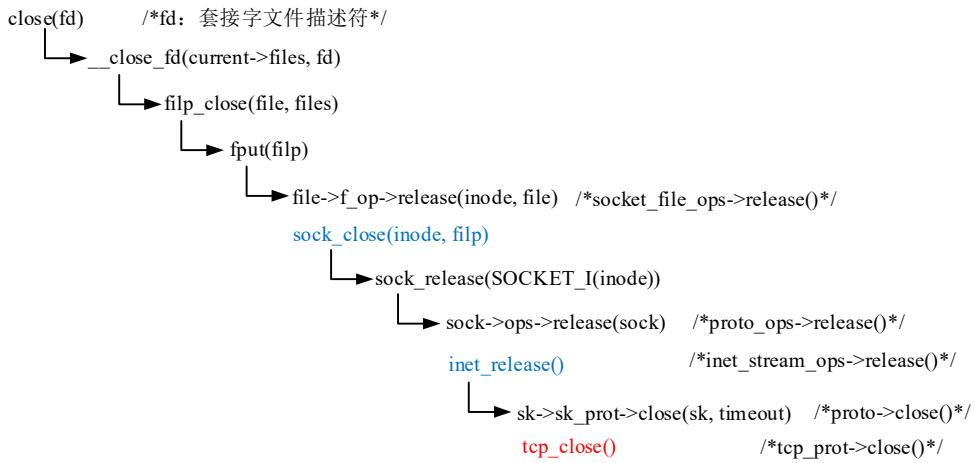
服务器套接字处于 TCP_ESTABLISHED 状态时，收到 FIN 报文段后，进入 TCP_CLOSE_WAIT 状态，向客户发回 ACK 报文段。服务器套接字收到客户发送的 FIN 报文段后，服务器端应用进程对套接字的读操作（read()）将返回 0，表示文件结尾，此时服务器端应用进程执行 close()系统调用。

close()系统调用内将服务器套接字设为 TCP_LAST_ACK 状态，并向客户套接字发送 FIN 报文段，随后进入睡眠等待，等待客户发回 ACK 报文段。服务器收到 ACK 报文段后，设置套接字状态为 TCP_CLOSE，唤醒服务器进程，继续执行 close()系统调用，在系统调用内释放套接字（连接）资源。

客户、服务器进程如果要在 close()系统调用内等待关闭连接操作完成，需通过 SO_LINGER 参数设置套接字(sock 标记成员）SOCK_LINGER 标志位，否则 close()系统调用内将不等待，立即释放本端连接资源后返回。

2 close()系统调用

用户进程通过 close()系统调用发起关闭连接的操作。close()系统调用在/fs/open.c 文件内实现，函数调用关系简列如下图所示：



close()系统调用中调用套接字文件操作结构 socket_file_ops 实例中的 release()函数，即 sock_close()函数。sock_close()调用 proto_ops 实例中的 release()函数，即 **inet_release()**函数。inet_release()函数最终调用 proto 实例中的 close()函数完成关闭套接字操作。

对于 TCP 套接字，proto 实例中的 close()函数为 **tcp_close()**，函数定义如下（/net/ipv4/tcp.c）：

```
void tcp_close(struct sock *sk, long timeout)
/*sk：待关闭套接字，timeout：等待时间，为 0 或 sk->sk_lingertime，见 inet_release()函数*/
{
    struct sk_buff *skb;
    int data_was_unread = 0;
    int state;

    lock_sock(sk);
    sk->sk_shutdown = SHUTDOWN_MASK;

    if (sk->sk_state == TCP_LISTEN) {       /*关闭欢迎套接字*/
        ...
    }

    while ((skb = __skb_dequeue(&sk->sk_receive_queue)) != NULL) {       /*清空接收缓存区*/
        u32 len = TCP_SKB_CB(skb)->end_seq - TCP_SKB_CB(skb)->seq;

        if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN)
            len--;
        data_was_unread += len;
        __kfree_skb(skb);       /*释放数据包*/
    }
    ...

    if (unlikely(tcp_sk(sk)->repair)) {
        sk->sk_prot->disconnect(sk, 0);
    } else if (data_was_unread) {
        ...
    } else if (tcp_close_state(sk)) {     /*根据套接字的当前状态设置下一个状态*/
                    /*此处设置状态为 TCP_FIN_WAIT1 或 TCP_LAST_ACK，/net/ipv4/tcp.c*/
```

```
        tcp_send_fin(sk);     /*清空发送缓存队列并发送 FIN 报文段，/net/ipv4/tcp_output.c*/
    }

    /*客户、服务器端在发送 FIN 报文段后进入睡眠，收到对方 ACK 报文段后被唤醒*/
    sk_stream_wait_close(sk, timeout);     /*睡眠等待，/net/core/stream.c*/

    /*以下是释放套接字（连接）资源*/
adjudge_to_death:
    state = sk->sk_state;     /*套接字当前状态*/
    sock_hold(sk);
    sock_orphan(sk);

    release_sock(sk);     /*释放套接字后备数据包，/net/core/sock.c*/

    local_bh_disable();
    bh_lock_sock(sk);
    WARN_ON(sock_owned_by_user(sk));

    percpu_counter_inc(sk->sk_prot->orphan_count);

    if (state != TCP_CLOSE && sk->sk_state == TCP_CLOSE)
        goto out;

    /*客户套接字在收到 ACK 后，进入 TCP_FIN_WAIT2 状态，进程被唤醒*/
    if (sk->sk_state == TCP_FIN_WAIT2) {
        struct tcp_sock *tp = tcp_sk(sk);
        if (tp->linger2 < 0) {       /*由 TCP_LINGER2 参数设置*/
            tcp_set_state(sk, TCP_CLOSE);           /*设置状态为 TCP_CLOSE*/
            tcp_send_active_reset(sk, GFP_ATOMIC);     /*向服务器发送复位报文段*/
            NET_INC_STATS_BH(sock_net(sk),
                    LINUX_MIB_TCPABORTONLINGER);
        } else {
            const int tmo = tcp_fin_time(sk);       /*/include/net/tcp.h*/

            if (tmo > TCP_TIMEWAIT_LEN) {
                inet_csk_reset_keepalive_timer(sk,tmo - TCP_TIMEWAIT_LEN);
            } else {
                tcp_time_wait(sk, TCP_FIN_WAIT2, tmo);    /*/net/ipv4/tcp_minisocks.c*/
                 /*套接字设为 TCP_TIME_WAIT 状态，
                   *睡眠等待一段时间后被唤醒，调用 tcp_done()，设状态为 TCP_CLOSE。
                   */
                goto out;       /*跳转到 out 处，释放套接字*/
            }
        }
    }
```

```
        if (sk->sk_state != TCP_CLOSE) {     /*套接字不是处于关闭状态*/
            sk_mem_reclaim(sk);
            if (tcp_check_oom(sk, 0)) {
                tcp_set_state(sk, TCP_CLOSE);
                tcp_send_active_reset(sk, GFP_ATOMIC);
                NET_INC_STATS_BH(sock_net(sk),LINUX_MIB_TCPABORTONMEMORY);
            }
        }

        if (sk->sk_state == TCP_CLOSE) {
                    /*服务器套接字收到 ACK 后，设为 TCP_CLOSE 状态，进程被唤醒*/
            struct request_sock *req = tcp_sk(sk)->fastopen_rsk;
            if (req)
                reqsk_fastopen_remove(sk, req, false);
            inet_csk_destroy_sock(sk);       /*销毁套接字，调用 sk->sk_prot->destroy(sk)函数等*/
                                             /*/net/ipv4/inet_connection_sock.c*/
        }
    out:
        bh_unlock_sock(sk);
        local_bh_enable();
        sock_put(sk);     /*释放套接字*/
    }
```

客户进程和服务器进程都通过 close()系统调用来关闭本端的套接字。close()系统调用中最终调用函数 tcp_close()执行关闭操作，主要工作是将套接字状态设为 TCP_FIN_WAIT1 或 TCP_LAST_ACK，发送 FIN 报文段，然后进程进入睡眠，等待对方发回的 ACK 报文段（timeout 需不为 0，为 0 则不等待），收到 ACK 后，进程被唤醒，在 tcp_close()函数内最后释放套接字。

tcp_close_state(sk)函数根据套接字当前状态，设置套接字的下一个状态。tcp_send_fin(sk)函数用于发送 FIN 报文段，如果套接字发送数据包缓存队列不为空，则在队列中最后一个数据包 TCP 报头中设置 FIN 标志位，然后将缓存队列中数据包全部发送出去。如果发送缓存队列为空，则创建一个数据包，TCP 报头中设置 FIN 标志位，然后将数据包发送出去。

进程在 sk_stream_wait_close(sk, timeout)函数中进入睡眠等待，timeout 为超时时间。套接字被唤醒时状态需不为(TCPF_FIN_WAIT1 | TCPF_CLOSING | TCPF_LAST_ACK)其中之一，超时时将被直接唤醒。进程唤醒后继续执行 tcp_close()函数，完成套接字释放工作。

close()系统调用与前面介绍的 connect()系统调用相似，系统调用内只发送第一个报文段，然后等待 TCP 完成剩下的工作，完成后再进行系统调用剩下的工作。对于 close()系统调用主要是要完成套接字资源的释放。

3 客户端工作

客户端进程通过 close()系统调用发起关闭操作，系统调用内将套接字状态设为 TCP_FIN_WAIT1，然后进程在套接字上睡眠等待，收到服务器端发回的 ACK 报文段后套接字进入 TCP_FIN_WAIT2 状态，唤醒睡眠进程，继续执行 close()系统调用，释放套接字。

close()系统调用前面介绍过了，下面介绍客户端接收服务器端 ACK 和 FIN 报文段的处理。

■接收 ACK 报文段

服务器套接字在收到 FIN 报文段后，发回 ACK 报文段，此时客户端套接字状态为 **TCP_FIN_WAIT1**。由 tcp_v4_do_rcv()函数可知，此 ACK 报文段由 tcp_rcv_state_process()函数处理，相关代码简列如下：

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb,const struct tcphdr *th, unsigned int len)
{
    ...
    switch (sk->sk_state) {
    ...
    case TCP_FIN_WAIT1: {
        struct dst_entry *dst;
        int tmo;
        ...
        if (tp->snd_una != tp->write_seq)
            break;

        tcp_set_state(sk, TCP_FIN_WAIT2);      /*设置套接字状态为 TCP_FIN_WAIT2*/
        sk->sk_shutdown |= SEND_SHUTDOWN;

        dst = __sk_dst_get(sk);
        if (dst)
            dst_confirm(dst);

        if (!sock_flag(sk, SOCK_DEAD)) {      /*套接字未设置 SOCK_DEAD 标志位*/
            sk->sk_state_change(sk);
                                /*sock_def_wakeup()，唤醒在套接字上睡眠进程，/net/core/sock.c*/

            break;
        }
        ...
    }    /*case TCP_FIN_WAIT1 结束*/
    ...
    }   /*switch (sk->sk_state)结束*/
    ...
}
```

客户端在收到 ACK 报文段后，将套接字状态设为 TCP_FIN_WAIT2，并唤醒在套接字上睡眠的进程。客户进程被唤醒后在 tcp_close()函数中继续执行，将套接字状态设为 TCP_TIME_WAIT，等待一段时间后，将套接字状态设为 TCP_CLOSE，而后释放套接字。

■接收 FIN 报文段

客户套接字在进入 TCP_TIME_WAIT 状态后，在此状态下将等待一段时间，在等待时将收到服务器发送的 FIN 报文段。此报文段由客户端 tcp_rcv_state_process()函数处理，主要是向服务器发送 ACK 报文段，相关代码简列如下：

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb,const struct tcphdr *th, unsigned int len)
{
```

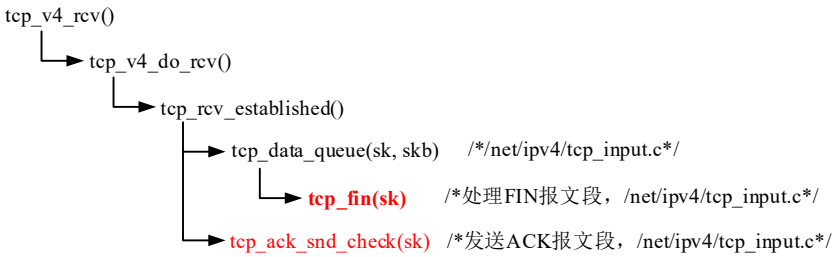
```
    ...
    if (sk->sk_state != TCP_CLOSE) {
        tcp_data_snd_check(sk);
        tcp_ack_snd_check(sk);      /*发送应答 ACK 报文段*/
    }
    ...
    return 0;
}
```

4 服务器端工作

服务器套接字在收到客户 FIN 报文段前处于 TCP_ESTABLISHED 状态，在收到客户 FIN 报文段后，将进入 TCP_CLOSE_WAIT 状态，并向客户发送 ACK 报文段。服务器端应用进程检测到客户端发起了关闭操作后（read()系统调用返回 0），执行 close()系统调用，关闭服务器端的连接。

■接收客户 FIN 报文段

服务器套接字在 TCP_ESTABLISHED 状态，收到客户端 FIN 报文段时，由 tcp_v4_do_rcv()函数可知，此 FIN 报文段由 **tcp_rcv_established()**函数处理，函数调用关系如下图所示：



tcp_fin()函数用于接收 FIN 报文段，**tcp_ack_snd_check()**函数用于发送 ACK 报文段。下面简要看一下 tcp_fin()函数实现，tcp_ack_snd_check()函数源代码请读者自行阅读。

```
static void tcp_fin(struct sock *sk)
{
    struct tcp_sock *tp = tcp_sk(sk);
    const struct dst_entry *dst;

    inet_csk_schedule_ack(sk);

    sk->sk_shutdown |= RCV_SHUTDOWN;
    sock_set_flag(sk, SOCK_DONE);       /*设置 sock 标志*/

    switch (sk->sk_state) {
    case TCP_SYN_RECV:
    case TCP_ESTABLISHED:       /*服务器套接字此时处于 TCP_ESTABLISHED 状态*/
        tcp_set_state(sk, TCP_CLOSE_WAIT);      /*套接字设为 TCP_CLOSE_WAIT 状态*/
        dst = __sk_dst_get(sk);
        if (!dst || !dst_metric(dst, RTAX_QUICKACK))
            inet_csk(sk)->icsk_ack.pingpong = 1;
        break;
```

```
        ...    /*处理套接字处于其它状态的情况*/
        }
        ...
    }
```

服务器套接字收到 FIN 报文段后，设置状态为 TCP_CLOSE_WAIT 状态，随后由服务器进程通过 close()系统调用发起关闭操作。

服务器套接字在 close()系统调用中进入 **TCP_LAST_ACK** 状态，向客户端发送 FIN 报文段，随后进入睡眠，等待客户发回的 ACK 报文段。

■接收客户 ACK 报文段

服务器进程在睡眠状态时，TCP 将接收客户端发回的 ACK 报文段。由前面介绍的 tcp_v4_do_rcv()函数可知，套接字不处于 TCP_ESTABLISHED 状态时，接收数据包都由函数 tcp_rcv_state_process()处理，函数代码简列如下（/net/ipv4/tcp_input.c）：

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb,const struct tcphdr *th, unsigned int len)
{
    ...
    switch (sk->sk_state) {
    ...
    case TCP_LAST_ACK:
        if (tp->snd_una == tp->write_seq) {
            tcp_update_metrics(sk);
            tcp_done(sk);    /*释放套接字，/net/ipv4/tcp.c*/
            goto discard;
        }
        break;
    }
    ...
    return 0;
}
```

tcp_rcv_state_process()函数内对处于 TCP_LAST_ACK 状态的套接字调用 tcp_done()函数，代码如下：

```
void tcp_done(struct sock *sk)
{
    struct request_sock *req = tcp_sk(sk)->fastopen_rsk;

    if (sk->sk_state == TCP_SYN_SENT || sk->sk_state == TCP_SYN_RECV)
        TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_ATTEMPTFAILS);

    tcp_set_state(sk, TCP_CLOSE);      /*设置状态为 TCP_CLOSE*/
    tcp_clear_xmit_timers(sk);      /*清除定时器*/
    if (req)
        reqsk_fastopen_remove(sk, req, false);

    sk->sk_shutdown = SHUTDOWN_MASK;
```

```
        if (!sock_flag(sk, SOCK_DEAD))
            sk->sk_state_change(sk);    /*sock_def_wakeup()，唤醒在套接字上睡眠进程，/net/core/sock.c*/
        else
            inet_csk_destroy_sock(sk);
    }
```

tcp_done()函数主要是将套接字状态设为 TCP_CLOSE，唤醒在套接字上睡眠等待的进程。服务器进程被唤醒后继续执行 close()系统调用，在 tcp_close()函数内将释放套接字（连接）资源。

12.9 小结

本章主要介绍了套接字的实现，netlink 套接字的实现，简要介绍了因特网各层协议，IPv4 套接字的实现，以及因特网传输层协议在内核中的实现，下一章将介绍因特网网络层和数据链路层协议在内核中的实现。