# 第13章 网络通信（3）

前一章我们完成了数据包发送流程的分析学习，本章则要学习数据包的接收过程，我们同样也从应用层开始入手，然后深入到内核的实现代码，从而真正理解接收数据接口。本章也是网络通信的最后一章。

## 13.1 系统调用接口

与发送类似，内核也提供了多个发送数据的系统调用接口。

#include <sys/types.h>

#include <sys/socket.h>

ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags,

struct sockaddr \*src\_addr, socklen\_t \*addrlen);

ssize\_t recvmsg(int sockfd, struct msghdr \*msg, int flags);

与send类似，recv一般也是用于面向连接的套接字。原因在于，对于非连接的套接字来说，使用recv接收数据，通过接口不能获得发送端的地址，也就是说不知道这个数据是谁发过来的。所以，如果使用者不关心发送端信息，或者该信息可以从数据中获得，那么recv接口同样也可以用于非连接的套接字。而recvfrom通过额外的参数src\_addr和addrlen，来获得发送方的地址，其中需要注意的是addrlen，它既是输入值又是输出值。recvmsg与sendmsg一样，把接收到的数据和地址都保存在msg中。其中msg.msg\_name和msg.msg\_len用于保存接收端地址，而msg.msg\_iov用于保存接收到的数据。这三个系统调用与对应的发送接口一样，都支持设置标志位flags——都是比较现代的接口设计方法。

## 13.2 数据包内核空间到用户空间的流程

前一章中，几个不同的发送数据包的系统调用，最终都是通过公共的函数sock\_sendmsg来完成的。那么对于接收数据包的系统调用，我们相信最后也是殊途同归，进入到一个公共的函数中。接下来，我们跟踪一下上一节三个系统调用的实现，来证明我们的猜想。

首先是recv的源码，

asmlinkage long sys\_recv(int fd, void \_\_user \*ubuf, size\_t size,

unsigned flags)

{

return sys\_recvfrom(fd, ubuf, size, flags, NULL, NULL);

}

代码很简单，recv完全是通过调用sys\_recvfrom来实现的，仅是将sys\_recvfrom的最后两个参数设置为0。

那么接下来就进入recvfrom的源码，

SYSCALL\_DEFINE6(recvfrom, int, fd, void \_\_user \*, ubuf, size\_t, size,

unsigned, flags, struct sockaddr \_\_user \*, addr,

int \_\_user \*, addr\_len)

{

struct socket \*sock;

struct iovec iov;

struct msghdr msg;

struct sockaddr\_storage address;

int err, err2;

int fput\_needed;

/\* 限制最大的读取字节长度为整数的最大值INT\_MAX \*/

if (size > INT\_MAX)

size = INT\_MAX;

/\* 从文件描述符得到套接字结构 \*/

sock = sockfd\_lookup\_light(fd, &err, &fput\_needed);

if (!sock)

goto out;

/\* 控制信息为0 \*/

msg.msg\_control = NULL;

msg.msg\_controllen = 0;

/\* 设置消息的数据段信息 \*/

msg.msg\_iovlen = 1;

msg.msg\_iov = &iov;

iov.iov\_len = size;

iov.iov\_base = ubuf;

/\* 设置消息存储地址信息 \*/

msg.msg\_name = (struct sockaddr \*)&address;

msg.msg\_namelen = sizeof(address);

/\* 如果套接字设置O\_NONBLOCK标志，即非阻塞标志。则设置MSG\_DONTWAIT标志，表示此次接收消息，无需等待 \*/

if (sock->file->f\_flags & O\_NONBLOCK)

flags |= MSG\_DONTWAIT;

/\* 调用sock\_recvmsg接收数据 \*/

err = sock\_recvmsg(sock, &msg, size, flags);

/\* 将地址信息复制到用户空间 \*/

if (err >= 0 && addr != NULL) {

err2 = move\_addr\_to\_user((struct sockaddr \*)&address,

msg.msg\_namelen, addr, addr\_len);

if (err2 < 0)

err = err2;

}

fput\_light(sock->file, fput\_needed);

out:

return err;

}

然后进入sock\_recvmsg->\_\_sock\_recvmsg->\_\_sock\_recvmsg\_nosec

static inline int \_\_sock\_recvmsg\_nosec(struct kiocb \*iocb, struct socket \*sock,

struct msghdr \*msg, size\_t size, int flags)

{

struct sock\_iocb \*si = kiocb\_to\_siocb(iocb);

sock\_update\_classid(sock->sk);

/\* 设置套接字异步IO信息 \*/

si->sock = sock;

si->scm = NULL;

si->msg = msg;

si->size = size;

si->flags = flags;

/\* 根据不同套接字类型，调用不同的数据接收函数 \*/

return sock->ops->recvmsg(iocb, sock, msg, size, flags);

}

下面我们跟踪第三个接收数据包系统调用recvmsg。

SYSCALL\_DEFINE3(recvmsg, int, fd, struct msghdr \_\_user \*, msg,

unsigned int, flags)

{

int fput\_needed, err;

struct msghdr msg\_sys;

/\* 从文件描述符fd获得套接字 \*/

struct socket \*sock = sockfd\_lookup\_light(fd, &err, &fput\_needed);

if (!sock)

goto out;

/\* \_\_sys\_recvmsg用于实现接收数据 \*/

err = \_\_sys\_recvmsg(sock, msg, &msg\_sys, flags, 0);

/\* 释放fd引用（如需要），这也是fput\_light与fput的区别 \*/

fput\_light(sock->file, fput\_needed);

out:

return err;

}

进入\_\_sys\_recvmsg，

static int \_\_sys\_recvmsg(struct socket \*sock, struct msghdr \_\_user \*msg,

struct msghdr \*msg\_sys, unsigned flags, int nosec)

{

struct compat\_msghdr \_\_user \*msg\_compat =

(struct compat\_msghdr \_\_user \*)msg;

struct iovec iovstack[UIO\_FASTIOV];

struct iovec \*iov = iovstack;

unsigned long cmsg\_ptr;

int err, iov\_size, total\_len, len;

/\* kernel mode address \*/

struct sockaddr\_storage addr;

/\* user mode address pointers \*/

struct sockaddr \_\_user \*uaddr;

int \_\_user \*uaddr\_len;

/\* 将消息头从用户空间复制到内核空间 \*/

if (MSG\_CMSG\_COMPAT & flags) {

if (get\_compat\_msghdr(msg\_sys, msg\_compat))

return -EFAULT;

} else if (copy\_from\_user(msg\_sys, msg, sizeof(struct msghdr)))

return -EFAULT;

err = -EMSGSIZE;

/\* 检查数据段个数 \*/

if (msg\_sys->msg\_iovlen > UIO\_MAXIOV)

goto out;

/\* Check whether to allocate the iovec area \*/

/\*

为了避免频繁申请内存，内核在栈上申请了UIO\_FASTIOV大小的iovec数组以供iov使用。当数据段个数超过UIO\_FASTIOV时，就需要动态申请内存。\*/

err = -ENOMEM;

iov\_size = msg\_sys->msg\_iovlen \* sizeof(struct iovec);

if (msg\_sys->msg\_iovlen > UIO\_FASTIOV) {

iov = sock\_kmalloc(sock->sk, iov\_size, GFP\_KERNEL);

if (!iov)

goto out;

}

/\* 验证用户传递的数据段参数和地址参数 \*/

uaddr = (\_\_force void \_\_user \*)msg\_sys->msg\_name;

uaddr\_len = COMPAT\_NAMELEN(msg);

if (MSG\_CMSG\_COMPAT & flags) {

err = verify\_compat\_iovec(msg\_sys, iov,

(struct sockaddr \*)&addr,

VERIFY\_WRITE);

} else

err = verify\_iovec(msg\_sys, iov,

(struct sockaddr \*)&addr,

VERIFY\_WRITE);

if (err < 0)

goto out\_freeiov;

total\_len = err;

cmsg\_ptr = (unsigned long)msg\_sys->msg\_control;

/\* 确保消息标志只保护这两个标志 \*/

msg\_sys->msg\_flags = flags & (MSG\_CMSG\_CLOEXEC|MSG\_CMSG\_COMPAT);

/\* 如果套接字为非阻塞，则设置标志位为不等待（非阻塞） \*/

if (sock->file->f\_flags & O\_NONBLOCK)

flags |= MSG\_DONTWAIT;

/\* 根据安全检查标志，调用不同的接收函数，但最终都会调用到sock\_recvmsg \*/

err = (nosec ? sock\_recvmsg\_nosec : sock\_recvmsg)(sock, msg\_sys,

total\_len, flags);

if (err < 0)

goto out\_freeiov;

len = err;

/\* 将发送端的地址复制到用户空间 \*/

if (uaddr != NULL) {

err = move\_addr\_to\_user((struct sockaddr \*)&addr,

msg\_sys->msg\_namelen, uaddr,

uaddr\_len);

if (err < 0)

goto out\_freeiov;

}

…… ……

}

由上面的代码可以看出，内核提供的三个接收数据包系统调用，也是殊途同归，最终都会走到\_\_sock\_recvmsg\_nosec，后面的调用就要依赖于具体的协议实现了。

## 13.3 UDP数据包的接收流程

首先我们分析一下相对简单的协议UDP的数据包接收流程。

int udp\_recvmsg(struct kiocb \*iocb, struct sock \*sk, struct msghdr \*msg,

size\_t len, int noblock, int flags, int \*addr\_len)

{

struct inet\_sock \*inet = inet\_sk(sk);

/\* 让sin指向msg\_name，用于保存发送端地址 \*/

struct sockaddr\_in \*sin = (struct sockaddr\_in \*)msg->msg\_name;

struct sk\_buff \*skb;

unsigned int ulen, copied;

int peeked;

int err;

int is\_udplite = IS\_UDPLITE(sk);

bool slow;

/\* addr\_len不为NULL，即用户传递了地址长度参数。进入了具体的协议层，已经可以明确地址长度信息了。 \*/

if (addr\_len)

\*addr\_len = sizeof(\*sin);

/\* 用户设置了MSG\_ERRQUEUE标志，用于接收错误消息。这个应用并不广泛，在此忽略。 \*/

if (flags & MSG\_ERRQUEUE)

return ip\_recv\_error(sk, msg, len);

try\_again:

/\* 接收了一个数据报文 \*/

skb = \_\_skb\_recv\_datagram(sk, flags | (noblock ? MSG\_DONTWAIT : 0),

&peeked, &err);

/\* 没有收到报文，则直接退出 \*/

if (!skb)

goto out;

/\* 得到UDP的数据长度 \*/

ulen = skb->len - sizeof(struct udphdr);

/\* 要复制的长度初始化为用户指定长度 \*/

copied = len;

/\* 复制长度大于UDP的数据长度，则调整复制长度为数据长度。若复制长度小于数据长度，则设置标志MSG\_TRUNC，表示数据发生了截断。 \*/

if (copied > ulen)

copied = ulen;

else if (copied < ulen)

msg->msg\_flags |= MSG\_TRUNC;

/\*

\* If checksum is needed at all, try to do it while copying the

\* data. If the data is truncated, or if we only want a partial

\* coverage checksum (UDP-Lite), do it before the copy.

\*/

/\* 如果发生了数据截断，或者我们只需要部分部分覆盖的校验和，那么就在复制前进行校验。 \*/

if (copied < ulen || UDP\_SKB\_CB(skb)->partial\_cov) {

/\* 进行UDP校验和校验 \*/

if (udp\_lib\_checksum\_complete(skb))

goto csum\_copy\_err;

}

/\* 判断是否需要进行校验和校验 \*/

if (skb\_csum\_unnecessary(skb)) {

/\* 复制数据包内容到msg\_iov中 \*/

err = skb\_copy\_datagram\_iovec(skb, sizeof(struct udphdr),

msg->msg\_iov, copied);

}

else {

/\* 复制数据包内容的同时，进行校验和校验 \*/

err = skb\_copy\_and\_csum\_datagram\_iovec(skb,

sizeof(struct udphdr),

msg->msg\_iov);

if (err == -EINVAL)

goto csum\_copy\_err;

}

/\* 复制错误检查 \*/

if (err)

goto out\_free;

/\* 如果不是peek动作，则增加相应的统计计数 \*/

if (!peeked)

UDP\_INC\_STATS\_USER(sock\_net(sk),

UDP\_MIB\_INDATAGRAMS, is\_udplite);

/\* 更新套接字的最新接收数据包时间戳以及丢包消息 \*/

sock\_recv\_ts\_and\_drops(msg, sk, skb);

/\* 如果用户指定了保存对端地址的参数，则从数据包中复制地址和端口信息 \*/

if (sin) {

sin->sin\_family = AF\_INET;

sin->sin\_port = udp\_hdr(skb)->source;

sin->sin\_addr.s\_addr = ip\_hdr(skb)->saddr;

memset(sin->sin\_zero, 0, sizeof(sin->sin\_zero));

}

/\* 设置了接收控制消息 \*/

if (inet->cmsg\_flags) {

/\* 接收控制消息如TTL、TOS等 \*/

ip\_cmsg\_recv(msg, skb);

}

/\* 设置已复制的字节长度 \*/

err = copied;

if (flags & MSG\_TRUNC)

err = ulen;

out\_free:

/\* 释放接收的这个数据包 \*/

skb\_free\_datagram\_locked(sk, skb);

out:

/\* 返回读取的字节数 \*/

return err;

/\* 错误处理 \*/

…… ……

}

再进入\_\_skb\_recv\_datagram，查看UDP如何接收的报文。

struct sk\_buff \*\_\_skb\_recv\_datagram(struct sock \*sk, unsigned flags,

int \*peeked, int \*err)

{

struct sk\_buff \*skb;

long timeo;

/\* 检查套接字是否出错 \*/

int error = sock\_error(sk);

if (error)

goto no\_packet;

/\* 得到超时时间，如果设置了MSG\_DONTWAIT，则超时为0。 \*/

timeo = sock\_rcvtimeo(sk, flags & MSG\_DONTWAIT);

do {

unsigned long cpu\_flags;

spin\_lock\_irqsave(&sk->sk\_receive\_queue.lock, cpu\_flags);

/\* 得到接收队列的第一个数据包 \*/

skb = skb\_peek(&sk->sk\_receive\_queue);

if (skb) {

\*peeked = skb->peeked;

/\* 如果只是查看动作，则仅是增加数据包的引用计数 \*/

if (flags & MSG\_PEEK) {

skb->peeked = 1;

atomic\_inc(&skb->users);

} else {

/\* 将数据包从接收队列中删除 \*/

\_\_skb\_unlink(skb, &sk->sk\_receive\_queue);

}

}

spin\_unlock\_irqrestore(&sk->sk\_receive\_queue.lock, cpu\_flags);

/\* 得到了数据包，直接返回 \*/

if (skb)

return skb;

/\* 如已经没有了剩余的超时时间，则跳转到no\_packet返回NULL \*/

error = -EAGAIN;

if (!timeo)

goto no\_packet;

/\* 使task在套接字上等待 \*/

} while (!wait\_for\_packet(sk, err, &timeo));

return NULL;

no\_packet:

\*err = error;

return NULL;

}

至此，UDP数据包的接收流程已经跟踪完毕。但是我们遗留了一个问题：在上面的代码中，报文是从接收队列中获得的，但是数据包又是如何保存到套接字的接收队列中的呢？这个问题留到后面再做分解。

## 13.4 TCP数据包的接收流程

前一节我们跟踪了UDP数据包的接收流程，本节我们来分析一下TCP数据包的接收流程。由于TCP协议的复杂性，其接受流程自然也比UDP要繁琐的多。

int tcp\_recvmsg(struct kiocb \*iocb, struct sock \*sk, struct msghdr \*msg,

size\_t len, int nonblock, int flags, int \*addr\_len)

{

struct tcp\_sock \*tp = tcp\_sk(sk);

int copied = 0;

u32 peek\_seq;

u32 \*seq;

unsigned long used;

int err;

int target; /\* Read at least this many bytes \*/

long timeo;

struct task\_struct \*user\_recv = NULL;

int copied\_early = 0;

struct sk\_buff \*skb;

u32 urg\_hole = 0;

/\* 对套接字上锁 \*/

lock\_sock(sk);

err = -ENOTCONN;

/\* 如果套接字为监听状态，则跳转到退出分支 \*/

if (sk->sk\_state == TCP\_LISTEN)

goto out;

/\* 与UDP类似，得到超时时间 \*/

timeo = sock\_rcvtimeo(sk, nonblock);

/\* 设置了MSG\_OOB标志，即带外数据，对于TCP来说，就是接收紧急数据 \*/

if (flags & MSG\_OOB)

goto recv\_urg;

seq = &tp->copied\_seq;

if (flags & MSG\_PEEK) {

peek\_seq = tp->copied\_seq;

seq = &peek\_seq;

}

/\*

计算最小的目标数据长度。

1. 当设置了MSG\_WAITALL，则目标为用户指定长度；

2. 不然，则选择套接字的低水线和用户指定长度的最小值；

3. 如果第二种情况的最小值为0，则目标长度为1字节；

\*/

target = sock\_rcvlowat(sk, flags & MSG\_WAITALL, len);

/\*

CONFIG\_NET\_DMA编译选项含义为TCP接收复制卸载。

利用DMA来将接收到的数据复制到用户空间，来节省CPU。

\*/

#ifdef CONFIG\_NET\_DMA

tp->ucopy.dma\_chan = NULL;

preempt\_disable();

skb = skb\_peek\_tail(&sk->sk\_receive\_queue);

{

int available = 0;

/\* 接收队列中有未读的数据包 \*/

if (skb) {

/\* 计算可读的数据量 \*/

available = TCP\_SKB\_CB(skb)->seq + skb->len - (\*seq);

}

if ((available < target) &&

(len > sysctl\_tcp\_dma\_copybreak) && !(flags & MSG\_PEEK) &&

!sysctl\_tcp\_low\_latency &&

dma\_find\_channel(DMA\_MEMCPY)) {

preempt\_enable\_no\_resched();

/\* 确定DMA要使用的数据段 \*/

tp->ucopy.pinned\_list =

dma\_pin\_iovec\_pages(msg->msg\_iov, len);

} else {

preempt\_enable\_no\_resched();

}

}

#endif

do {

u32 offset;

/\* 判断是否在读取紧急数据 \*/

if (tp->urg\_data && tp->urg\_seq == \*seq) {

/\* 如果已经读取了一定数据，则结束读取 \*/

if (copied)

break;

/\* 如果有未处理的信号，也结束读取 \*/

if (signal\_pending(current)) {

copied = timeo ? sock\_intr\_errno(timeo) : -EAGAIN;

break;

}

}

/\* 遍历接收队列 \*/

skb\_queue\_walk(&sk->sk\_receive\_queue, skb) {

/\* Now that we have two receive queues this

\* shouldn't happen.

\*/

if (WARN(before(\*seq, TCP\_SKB\_CB(skb)->seq),

"recvmsg bug: copied %X seq %X rcvnxt %X fl %X\n",

\*seq, TCP\_SKB\_CB(skb)->seq, tp->rcv\_nxt,

flags))

break;

/\* 取得在数据包中的偏移，即上次没有将这个数据包读取完毕 \*/

offset = \*seq - TCP\_SKB\_CB(skb)->seq;

/\* syn标志会占用一个sequence，所以偏移减一 \*/

if (tcp\_hdr(skb)->syn)

offset--;

/\* 偏移小于数据包长度，则这个数据包为要接收的 \*/

if (offset < skb->len)

goto found\_ok\_skb;

/\* 如果当前数据包包含FIN标志，则跳转到fin处理处 \*/

if (tcp\_hdr(skb)->fin)

goto found\_fin\_ok;

WARN(!(flags & MSG\_PEEK),

"recvmsg bug 2: copied %X seq %X rcvnxt %X fl %X\n",

\*seq, TCP\_SKB\_CB(skb)->seq, tp->rcv\_nxt, flags);

}

/\* 已经复制了超过目标的数据并且有积压的数据，则立刻跳出，尝试处理积压数据。 \*/

if (copied >= target && !sk->sk\_backlog.tail)

break;

/\*

这里针对是否已经复制了部分数据做了条件判断，而每个分支中都有相似的条件判断，为什么要分两种情况呢？因为在读取过程中，发生了同样的错误时，读取了部分数据，系统调用的返回值要返回成功读取的字节数，而未读取任何数据时，则返回-1错误。

\*/

if (copied) {

/\*

已复制了部分数据，检查下面几个条件：

1. 套接字出错；

2. 连接已经关闭；

3. 套接字关闭了接收端；

4. 已经超时；

5. 有待处理的信号；

则跳出接收数据循环。

\*/

if (sk->sk\_err ||

sk->sk\_state == TCP\_CLOSE ||

(sk->sk\_shutdown & RCV\_SHUTDOWN) ||

!timeo ||

signal\_pending(current))

break;

} else {

/\*

当套接字设置了SOCK\_DONE标志，则跳出循环。

对于TCP来说，被动关闭时，套接字会被设置这个标志。这时，意味着对端已经关闭，所以不可能再有新的数据了。

\*/

if (sock\_flag(sk, SOCK\_DONE))

break;

/\* 判断套接字是否出错 \*/

if (sk->sk\_err) {

copied = sock\_error(sk);

break;

}

/\* 套接字关闭了接收端 \*/

if (sk->sk\_shutdown & RCV\_SHUTDOWN)

break;

/\* 套接字状态为关闭状态但又没有设置SOCK\_DONE标志，这种情况只发生在用户企图从一个未连接的套接字读取数据。 \*/

if (sk->sk\_state == TCP\_CLOSE) {

if (!sock\_flag(sk, SOCK\_DONE)) {

copied = -ENOTCONN;

break;

}

break;

}

/\* 已经超时 \*/

if (!timeo) {

copied = -EAGAIN;

break;

}

/\* 有未处理的信号 \*/

if (signal\_pending(current)) {

copied = sock\_intr\_errno(timeo);

break;

}

}

/\* 清除已经读取的数据包 \*/

tcp\_cleanup\_rbuf(sk, copied);

if (!sysctl\_tcp\_low\_latency && tp->ucopy.task == user\_recv) {

/\* 保存用户进程地址 \*/

if (!user\_recv && !(flags & (MSG\_TRUNC | MSG\_PEEK))) {

user\_recv = current;

tp->ucopy.task = user\_recv;

tp->ucopy.iov = msg->msg\_iov;

}

tp->ucopy.len = len;

WARN\_ON(tp->copied\_seq != tp->rcv\_nxt &&

!(flags & (MSG\_PEEK | MSG\_TRUNC)));

/\*

处理完receive queue，需要处理prequeue 。

TCP套接字有三个队列，需要按照以下顺序处理

1. receive\_queue;

2. prequeue；

3. backlog；

\*/

if (!skb\_queue\_empty(&tp->ucopy.prequeue))

goto do\_prequeue;

/\* \_\_ Set realtime policy in scheduler \_\_ \*/

}

#ifdef CONFIG\_NET\_DMA

if (tp->ucopy.dma\_chan) {

if (tp->rcv\_wnd == 0 &&

!skb\_queue\_empty(&sk->sk\_async\_wait\_queue)) {

/\*

接收窗口已经为0并且有进程在等待数据，这时要尽快接收数据。所以这里的dma操作为同步的。

\*/

tcp\_service\_net\_dma(sk, true);

tcp\_cleanup\_rbuf(sk, copied);

} else

dma\_async\_memcpy\_issue\_pending(tp->ucopy.dma\_chan);

}

#endif

if (copied >= target) {

/\* 已复制了超过目标的数据量 \*/

release\_sock(sk);

lock\_sock(sk);

} else {

/\* 等待更多的数据 \*/

sk\_wait\_data(sk, &timeo);

}

#ifdef CONFIG\_NET\_DMA

tcp\_service\_net\_dma(sk, false); /\* Don't block \*/

tp->ucopy.wakeup = 0;

#endif

if (user\_recv) {

int chunk;

/\* \_\_ Restore normal policy in scheduler \_\_ \*/

if ((chunk = len - tp->ucopy.len) != 0) {

NET\_ADD\_STATS\_USER(sock\_net(sk), LINUX\_MIB\_TCPDIRECTCOPYFROMBACKLOG, chunk);

len -= chunk;

copied += chunk;

}

/\* 处理完receive\_queue，继续处理prequeue \*/

if (tp->rcv\_nxt == tp->copied\_seq &&

!skb\_queue\_empty(&tp->ucopy.prequeue)) {

do\_prequeue:

tcp\_prequeue\_process(sk);

/\* 计算从prequeue中读取的数据长度，并调整相应的len和copied。 \*/

if ((chunk = len - tp->ucopy.len) != 0) {

NET\_ADD\_STATS\_USER(sock\_net(sk), LINUX\_MIB\_TCPDIRECTCOPYFROMPREQUEUE, chunk);

len -= chunk;

copied += chunk;

}

}

}

if ((flags & MSG\_PEEK) &&

(peek\_seq - copied - urg\_hole != tp->copied\_seq)) {

if (net\_ratelimit())

printk(KERN\_DEBUG "TCP(%s:%d): Application bug, race in MSG\_PEEK.\n",

current->comm, task\_pid\_nr(current));

peek\_seq = tp->copied\_seq;

}

continue;

found\_ok\_skb:

/\* Ok so how much can we use? \*/

/\* 找到了正确的skb，计算该skb未读的可用数据长度 \*/

used = skb->len - offset;

/\* 如果用户要读取的长度小于当前剩余长度，则调整可用长度 \*/

if (len < used)

used = len;

/\* 判断是否有紧急数据 \*/

if (tp->urg\_data) {

/\* 得到紧急数据的偏移 \*/

u32 urg\_offset = tp->urg\_seq - \*seq;

if (urg\_offset < used) {

if (!urg\_offset) {

/\* 判断紧急数据是否在普通数据流中 \*/

if (!sock\_flag(sk, SOCK\_URGINLINE)) {

++\*seq;

urg\_hole++;

offset++;

used--;

if (!used)

goto skip\_copy;

}

} else

used = urg\_offset;

}

}

/\* 没有设置截断标志 \*/

if (!(flags & MSG\_TRUNC)) {

/\* 先尝试使用DMA来将数据复制到用户空间 \*/

#ifdef CONFIG\_NET\_DMA

if (!tp->ucopy.dma\_chan && tp->ucopy.pinned\_list)

tp->ucopy.dma\_chan = dma\_find\_channel(DMA\_MEMCPY);

if (tp->ucopy.dma\_chan) {

tp->ucopy.dma\_cookie = dma\_skb\_copy\_datagram\_iovec(

tp->ucopy.dma\_chan, skb, offset,

msg->msg\_iov, used,

tp->ucopy.pinned\_list);

if (tp->ucopy.dma\_cookie < 0) {

printk(KERN\_ALERT "dma\_cookie < 0\n");

/\* Exception. Bailout! \*/

if (!copied)

copied = -EFAULT;

break;

}

dma\_async\_memcpy\_issue\_pending(tp->ucopy.dma\_chan);

if ((offset + used) == skb->len)

copied\_early = 1;

} else

#endif

{

/\* 复制数据到用户空间 \*/

err = skb\_copy\_datagram\_iovec(skb, offset,

msg->msg\_iov, used);

if (err) {

/\* Exception. Bailout! \*/

if (!copied)

copied = -EFAULT;

break;

}

}

}

/\* 调整序列号\*seq,已复制长度，剩余长度 \*/

\*seq += used;

copied += used;

len -= used;

/\* 因为成功读取了数据，所有要调整TCP套接字的接收缓存 \*/

tcp\_rcv\_space\_adjust(sk);

skip\_copy:

/\* 如果正在读取，并且已读取的序列号大于紧急数据，这意味着已经读取完紧急数据。那么就重置urg\_data，并且进行TCP快速路径检查（通过检查，则打开快速路径开关。打开快速路径的时候，TCP接收数据包时会做比较少的检查，接收更为快速）。 \*/

if (tp->urg\_data && after(tp->copied\_seq, tp->urg\_seq)) {

tp->urg\_data = 0;

tcp\_fast\_path\_check(sk);

}

/\* 使用的数据长度加上偏移小于数据包的长度，则该数据包可以继续使用 \*/

if (used + offset < skb->len)

continue;

/\* 如果该数据包有FIN标志，则跳转到found\_fin\_ok \*/

if (tcp\_hdr(skb)->fin)

goto found\_fin\_ok;

/\* 如果没设置MSG\_PEEK标志，则需要从接收队列中消耗掉这个数据包，并根据copied\_early标志，将其直接释放或者放置其异步队列 \*/

if (!(flags & MSG\_PEEK)) {

sk\_eat\_skb(sk, skb, copied\_early);

copied\_early = 0;

}

continue;

found\_fin\_ok:

/\* 这里开始处理FIN数据包 \*/

/\* FIN标志也占用一个序列号，因此给序列号加一 \*/

++\*seq;

/\* 与前文相同，不再重复注释 \*/

if (!(flags & MSG\_PEEK)) {

sk\_eat\_skb(sk, skb, copied\_early);

copied\_early = 0;

}

/\* 接收到FIN标志，表示对端已经关闭写通道，那么对于本端来说，这是这是最后一个可读数据包，因此退出循环。 \*/

break;

} while (len > 0);

if (user\_recv) {

/\* prequeue队列中仍然有未读取的数据包 \*/

if (!skb\_queue\_empty(&tp->ucopy.prequeue)) {

int chunk;

/\* 设置要读取的长度 \*/

tp->ucopy.len = copied > 0 ? len : 0;

/\* 处理prequeue队列 \*/

tcp\_prequeue\_process(sk);

if (copied > 0 && (chunk = len - tp->ucopy.len) != 0) {

NET\_ADD\_STATS\_USER(sock\_net(sk), LINUX\_MIB\_TCPDIRECTCOPYFROMPREQUEUE, chunk);

len -= chunk;

copied += chunk;

}

}

tp->ucopy.task = NULL;

tp->ucopy.len = 0;

}

#ifdef CONFIG\_NET\_DMA

tcp\_service\_net\_dma(sk, true); /\* Wait for queue to drain \*/

tp->ucopy.dma\_chan = NULL;

if (tp->ucopy.pinned\_list) {

dma\_unpin\_iovec\_pages(tp->ucopy.pinned\_list);

tp->ucopy.pinned\_list = NULL;

}

#endif

/\* 释放已经读取的数据包 \*/

tcp\_cleanup\_rbuf(sk, copied);

/\* 释放套接字控制权 \*/

release\_sock(sk);

return copied;

out:

release\_sock(sk);

return err;

recv\_urg:

/\* 接收紧急数据 \*/

err = tcp\_recv\_urg(sk, msg, len, flags);

goto out;

}

上面的tcp\_recvmsg即使已经加了大量的注释，但是由于这个函数逻辑过于复杂，以及TCP接收队列的多样性，即使我们已经看完了这个函数实现，却仍然无法清楚的掌握这个函数的整体脉络。后面，我会通过进一步分析TCP套接字几个队列的用途，尽力理清TCP数据包的接收流程。到时候，读者可以前后对照来增强理解。

## 13.5从网卡到套接字

对于一般的套接字编程来说，大多是应用编程，所以套接字一般都是UDP或者TCP协议的套接字。前面两章是从应用层次的角度，自上而下的分析了UDP和TCP数据包的接收流程。同时，也有了一个新的问题，数据包是如何进入对应套接字的接收缓冲的呢？本章将从网卡接收到数据包开始，一直跟踪到内核将数据包放入到对应的套接字缓冲为止。

### 13.5.1 从硬中断到软中断

对于网卡来说，数据包的到达是一个无法预料的事件，系统需要某种手段来得知该事件。一般来说，有两种方式：轮询和中断。用直白的语言来描述，轮询就是CPU不断地问网卡：“你那有准备好的数据包吗？”。如果网卡回答有数据包的话，CPU就进行处理，不然要么干点别的，要么继续问。后者呢，则是CPU该干嘛干嘛。当网卡收到数据包后，直接喊话“喂，有活干了”。于是CPU赶紧把手头工作保存一下，尽快响应任务。第一种方式，毫无疑问会造成CPU的浪费。因为在网卡没有数据包的时候，CPU还是要浪费计算周期来询问网卡。第二种中断方式呢，看上去很美，网卡没有数据的时候，CPU可以做其它事情，有数据的时候，则可以及时处理。然而在实际应用中，中断方式也有很大的问题。在CPU影响中断时，为了不影响当前工作，需要将当前工作的上下文保存起来，然后再进行中断处理。试想，当前千兆、万兆网卡已经非常普遍，当网卡满负载的时候，每秒钟会产生大量中断。除了切换过程带来的计算代价，上下文的切换还会导致cache的失效——这对高性能设备来说，是不可忽视的问题。于是，Linux对这两种方式进行了折中，引入了一个New API，缩写为NAPI。简单来说，在CPU响应网卡中断时，不再仅仅处理一个数据包就退出，而是使用轮询的方式继续尝试处理新数据包，直到没有新包到来或者到达本次中断的最多处理数据包个数。

网卡硬中断的处理是在网卡驱动中进行的，这个与硬件联系过于紧密，我们可以忽略细节。只需要知道对于支持NAPI的网卡来说，其读取数据包的硬中断处理函数会调用\_\_napi\_schedule，将网卡加入NAPI的poll list中。

void \_\_napi\_schedule(struct napi\_struct \*n)

{

unsigned long flags;

/\* 禁止本地中断，保护添加poll list的临界区 \*/

local\_irq\_save(flags);

/\* 加入到当前CPU的poll列表中 \*/

\_\_\_\_napi\_schedule(&\_\_get\_cpu\_var(softnet\_data), n);

local\_irq\_restore(flags);

}

进入\_\_\_\_napi\_schedule,

static inline void \_\_\_\_napi\_schedule(struct softnet\_data \*sd,

struct napi\_struct \*napi)

{

/\* 将napi加入对队尾 \*/

list\_add\_tail(&napi->poll\_list, &sd->poll\_list);

/\* 触发接收当前CPU接收软中断（实际上设置一个标志位） \*/

\_\_raise\_softirq\_irqoff(NET\_RX\_SOFTIRQ);

}

关于中断处理为什么要分为硬中断和软中断（也经常被称为上下部分）的解释已经很多了。简单地说：硬中断处理是一个特殊的上下文，CPU会屏蔽掉绝大部分中断，并且有不少限制。所以硬中断应尽可能的快速处理，来增加系统的响应，因此要将具体的处理工作放到软中断中。

### 13.5.2 软中断处理

前一节，我们看到硬中断通过设置标志位，“触发”了软中断。那么内核又是何时处理软中断的呢？内核目前在以下几种条件下，会检查是否需要处理软中断

1. 退出硬中断上下文；
2. 重新enable软中断时；
3. 每个cpu都有一个ksoftirqd的内核线程。当内核的软中断数量过多时，就会唤醒该线程循环处理软中断；

接收数据包的软中断处理函数为net\_rx\_action

static void net\_rx\_action(struct softirq\_action \*h)

{

/\* 接收数据包的per cpu队列 \*/

struct softnet\_data \*sd = &\_\_get\_cpu\_var(softnet\_data);

/\* 最长的运行时间限制为2个jiffies \*/

unsigned long time\_limit = jiffies + 2;

/\*

一次软中断最多处理的包个数。

netdev\_budget的值为/proc/sys/net/core/netdev\_budget

\*/

int budget = netdev\_budget;

void \*have;

/\* 因为网卡驱动会访问poll list，因此需要禁止本地硬中断 \*/

local\_irq\_disable();

/\* 遍历所有加入到poll链表的网卡 \*/

while (!list\_empty(&sd->poll\_list)) {

struct napi\_struct \*n;

int work, weight;

/\* 已经消耗掉最大的数据包或者超出了允许时间限制，则退出此次处理 \*/

if (unlikely(budget <= 0 || time\_after(jiffies, time\_limit))) {

/\* 这时退出此次收包软中断只为了避免过长的占用CPU，所以跳到softnet\_break，再次触发一次收包软中断 \*/

goto softnet\_break;

}

/\* 打开本地硬中断 \*/

local\_irq\_enable();

/\* 虽然这里在打开硬中断时，访问了poll\_list，但仍然是安全的。因为硬中断只是在往poll\_list的末尾插入，不会影响第一个元素。 \*/

n = list\_first\_entry(&sd->poll\_list, struct napi\_struct, poll\_list);

/\* 获得该设备的netpoll锁 \*/

have = netpoll\_poll\_lock(n);

/\* 得到该网卡的权重，其意义一般为在这个网卡接收几个数据包 \*/

weight = n->weight;

/\* This NAPI\_STATE\_SCHED test is for avoiding a race

\* with netpoll's poll\_napi(). Only the entity which

\* obtains the lock and sees NAPI\_STATE\_SCHED set will

\* actually make the ->poll() call. Therefore we avoid

\* accidentally calling ->poll() when NAPI is not scheduled.

\*/

work = 0;

/\* 再次检查该网卡是否有NAPI调用（因为与netpoll有竞争） \*/

if (test\_bit(NAPI\_STATE\_SCHED, &n->state)) {

/\* 对网卡进行查询操作，work值为读取的数据包个数 \*/

work = n->poll(n, weight);

trace\_napi\_poll(n);

}

WARN\_ON\_ONCE(work > weight);

/\* 更新包预算即还可以读取数据包个数 \*/

budget -= work;

local\_irq\_disable();

/\* 从该网卡读取的数据包达到预算个数 \*/

if (unlikely(work == weight)) {

/\* 判断该网卡的NAPI是否被禁止了 \*/

if (unlikely(napi\_disable\_pending(n))) {

/\* NAPI已经被禁止了，则执行NAPI的完成处理 \*/

local\_irq\_enable();

napi\_complete(n);

local\_irq\_disable();

} else {

/\* 该设备仍然要继续进行NAPI操作，则将其移至队尾 \*/

list\_move\_tail(&n->poll\_list, &sd->poll\_list);

}

}

/\* 释放netpoll锁 \*/

netpoll\_poll\_unlock(have);

}

out:

/\* 执行RPS处理并打开本地硬中断 \*/

net\_rps\_action\_and\_irq\_enable(sd);

#ifdef CONFIG\_NET\_DMA

/\* 启动未处理的DMA操作 \*/

dma\_issue\_pending\_all();

#endif

return;

softnet\_break:

/\* 本次没有接收完所有的数据包，再次触发一次软中断 \*/

sd->time\_squeeze++;

\_\_raise\_softirq\_irqoff(NET\_RX\_SOFTIRQ);

goto out;

}

在这个收包软中断处理函数中，CPU会遍历poll列表，调用挂载到NAPI列表上的网卡回调函数poll，来轮询接收数据包。所以，我们还需要进入驱动代码，来跟踪收包流程。在此，以Intel的e1000网卡驱动为例，在使用NAPI的情况下，其收包流程为net\_rx\_action-> e1000\_clean-> e1000\_clean\_rx\_irq-> e1000\_receive\_skb-> napi\_gro\_receive-> netif\_receive\_skb。在这个调用链上，有一个函数是napi\_gro\_receive，其是用于支持GRO（Generic Receive Offload）。这个GRO是用于减轻CPU的处理压力而诞生的。大家可以计算一下，对于10G，100G网卡来说，即使每个数据包都是1500字节（以太网的最大MTU，暂不考虑Jumbo帧），每秒钟系统需要处理多少个数据包？因此，为了减轻CPU的负担，Linux内核在驱动层引入了GRO，其会将符合条件的数据包合并为一个数据包再传递给系统协议栈。在此，我们只关注数据包的接收流程，就不研究GRO的实现了。有兴趣的同学可以自行阅读源码。

### 13.5.3 传递给协议栈流程

数据包在脱离驱动层后，进入了netif\_receive\_skb：

int netif\_receive\_skb(struct sk\_buff \*skb)

{

/\* 判断是否在入队前给数据包打时间戳 \*/

if (netdev\_tstamp\_prequeue)

net\_timestamp\_check(skb);

if (skb\_defer\_rx\_timestamp(skb))

return NET\_RX\_SUCCESS;

/\* 是否打开了RPS（Receive Packet Steering）编译开关，其根据IP地址，端口号进行hash运算，将其发送给对应CPU。一方面保证了CPU间的负载均衡，另一方面将同一特征的数据包发给同一个CPU，提高了cache的命中率。 \*/

#ifdef CONFIG\_RPS

{

struct rps\_dev\_flow voidflow, \*rflow = &voidflow;

int cpu, ret;

rcu\_read\_lock();

/\* 根据RPS算法，计算得到处理这个数据包的CPU \*/

cpu = get\_rps\_cpu(skb->dev, skb, &rflow);

/\* 当cpu大于等于0时，表示RPS计算得到了正确的CPU \*/

if (cpu >= 0) {

/\* 向其它cpu的队列追加这个数据包 \*/

ret = enqueue\_to\_backlog(skb, cpu, &rflow->last\_qtail);

rcu\_read\_unlock();

} else {

/\* 由本CPU处理该数据包 \*/

rcu\_read\_unlock();

ret = \_\_netif\_receive\_skb(skb);

}

return ret;

}

#else

/\* 本CPU继续处理该数据包 \*/

return \_\_netif\_receive\_skb(skb);

#endif

}

这里可以看出netif\_receive\_skb只是对\_\_netif\_receive\_skb的封装，增加了对RPS的支持。继续跟进\_\_netif\_receive\_skb：

static int \_\_netif\_receive\_skb(struct sk\_buff \*skb)

{

struct packet\_type \*ptype, \*pt\_prev;

rx\_handler\_func\_t \*rx\_handler;

struct net\_device \*orig\_dev;

struct net\_device \*null\_or\_dev;

bool deliver\_exact = false;

int ret = NET\_RX\_DROP;

\_\_be16 type;

/\* 如果没有打开入队前采样数据包时间戳功能，则需要在这里进行数据包时间戳采样 \*/

if (!netdev\_tstamp\_prequeue)

net\_timestamp\_check(skb);

trace\_netif\_receive\_skb(skb);

/\* 判断是否由netpoll处理 \*/

if (netpoll\_receive\_skb(skb))

return NET\_RX\_DROP;

/\* 设置网卡的入口网卡 \*/

if (!skb->skb\_iif)

skb->skb\_iif = skb->dev->ifindex;

orig\_dev = skb->dev;

/\* 初始化数据包的网络层首部，传输层首部，以及二层MAC首部长度 \*/

skb\_reset\_network\_header(skb);

skb\_reset\_transport\_header(skb);

skb\_reset\_mac\_len(skb);

pt\_prev = NULL;

rcu\_read\_lock();

another\_round:

\_\_this\_cpu\_inc(softnet\_data.processed);

/\* 如果是802.1Q协议的 数据包 \*/

if (skb->protocol == cpu\_to\_be16(ETH\_P\_8021Q)) {

/\* 去掉vlan tag \*/

skb = vlan\_untag(skb);

if (unlikely(!skb))

goto out;

}

/\* 内核打开了包分类编译选项 \*/

#ifdef CONFIG\_NET\_CLS\_ACT

/\* 如果数据包被设置了流控结果，则跳过后面的流控处理 \*/

if (skb->tc\_verd & TC\_NCLS) {

skb->tc\_verd = CLR\_TC\_NCLS(skb->tc\_verd);

goto ncls;

}

#endif

/\* 遍历注册在ptype\_all上所有节点。ptype\_all上的节点需要处理所有收到的以太网数据包 \*/

list\_for\_each\_entry\_rcu(ptype, &ptype\_all, list) {

/\* 如果注册节点没有绑定网卡，或者绑定的网卡与数据包接收网卡相同，则这个节点符合接收数据包条件 \*/

if (!ptype->dev || ptype->dev == skb->dev) {

/\* 将数据包传递给对应的处理函数 \*/

if (pt\_prev)

ret = deliver\_skb(skb, pt\_prev, orig\_dev);

pt\_prev = ptype;

}

}

/\* 内核打开了包分类编译选项 \*/

#ifdef CONFIG\_NET\_CLS\_ACT

skb = handle\_ing(skb, &pt\_prev, &ret, orig\_dev);

if (!skb)

goto out;

ncls:

#endif

/\* 如果这个数据包带有vlan标签 \*/

if (vlan\_tx\_tag\_present(skb)) {

if (pt\_prev) {

/\* 将数据包传递给之前确定的上层协议 \*/

ret = deliver\_skb(skb, pt\_prev, orig\_dev);

pt\_prev = NULL;

}

/\* 进行vlan的处理 \*/

if (vlan\_do\_receive(&skb))

goto another\_round;

else if (unlikely(!skb))

goto out;

}

/\*

判断该设备是否注册了接收处理函数。

何时设备上回注册了接收处理函数呢？netdev\_rx\_handler\_register是注册设备接收处理函数的接口。通过搜索netdev\_rx\_handler\_register的调用者，可以发现当网卡作为bond、加入桥接，或者创建macvlan时，会注册网卡的处理函数。使用这种方式，就做到了网卡接收处理函数与接收框架的解耦。对于框架来说，通过这个回调函数（用函数指针实现，内核中充斥着这样的代码），可以完全不用了解具体细节。未来增加更多的网卡处理函数时，只需要在该具体实现上，调用注册函数，而不用更改接收框架的代码。

\*/

rx\_handler = rcu\_dereference(skb->dev->rx\_handler);

if (rx\_handler) {

if (pt\_prev) {

/\* 将数据包传递给之前确定的上层协议 \*/

ret = deliver\_skb(skb, pt\_prev, orig\_dev);

pt\_prev = NULL;

}

/\* 调用在设备上注册的处理函数 \*/

switch (rx\_handler(&skb)) {

case RX\_HANDLER\_CONSUMED:

/\* 处理函数已经消耗了这个数据包，直接跳至退出 \*/

ret = NET\_RX\_SUCCESS;

goto out;

case RX\_HANDLER\_ANOTHER:

/\* 跳至another\_round，即跳至函数开头，重新处理 \*/

goto another\_round;

case RX\_HANDLER\_EXACT:

/\* 指示必须严格匹配接收网卡 \*/

deliver\_exact = true;

case RX\_HANDLER\_PASS:

/\* 继续后面的处理 \*/

break;

default:

BUG();

}

}

/\* 如果数据包还带有vlan tag，则证明该数据包是发给其他终端的 \*/

if (vlan\_tx\_nonzero\_tag\_present(skb))

skb->pkt\_type = PACKET\_OTHERHOST;

/\* deliver only exact match when indicated \*/

null\_or\_dev = deliver\_exact ? skb->dev : NULL;

/\* 根据数据包类型，遍历对应的处理函数 \*/

type = skb->protocol;

list\_for\_each\_entry\_rcu(ptype,

&ptype\_base[ntohs(type) & PTYPE\_HASH\_MASK], list) {

/\* 如果数据包类型匹配，并且接收接口设备也匹配，则证明这是正确的协议处理函数。则调用前面的响应处理函数，将数据包传递给上层协议 \*/

if (ptype->type == type &&

(ptype->dev == null\_or\_dev || ptype->dev == skb->dev ||

ptype->dev == orig\_dev)) {

if (pt\_prev)

ret = deliver\_skb(skb, pt\_prev, orig\_dev);

pt\_prev = ptype;

}

}

/\*

最后检查pt\_prev是否为真。为真，则表示前面有匹配的处理函数，然后进行调用。如果为假，表示对于这个数据包，内核没有对应的处理函数，则直接释放这个数据包。 \*/

if (pt\_prev) {

ret = pt\_prev->func(skb, skb->dev, pt\_prev, orig\_dev);

} else {

atomic\_long\_inc(&skb->dev->rx\_dropped);

kfree\_skb(skb);

/\* Jamal, now you will not able to escape explaining

\* me how you were going to use this. :-)

\*/

ret = NET\_RX\_DROP;

}

out:

rcu\_read\_unlock();

return ret;

}

在上面的代码中，有一个地方，大家一定觉得会有点奇怪。为什么每次匹配了协议时，内核都是检查上一次的结构pt\_prev。如果有，就调用上次匹配的结构；如果没有，则使用pt\_prev保存本次的处理函数，而不是直接调用呢？这样让代码看上去有点奇怪。内核之所以这样处理，是处于性能上面的考虑。如果不利用pt\_prev，而是直接调用当前匹配的处理函数，那么内核在第一次找到对应的处理函数时，就把数据包传递给上层应用。这时，内核不知道后面是否还有应该调用的处理函数，需要保证这个数据包的内容不能被修改。因此当这个数据包传递给上层应用时，如果该处理函数有可能修改数据包的话，就必须重新申请内存复制这个数据包，才能保证原数据包内容不变。然而更多的时候，对于一个数据包，可能只会有一个处理函数被调用。那么这个内存申请和复制，就是白白浪费掉的。因此内核在这里利用一个变量pt\_prev来保存上次的匹配结果，暂时不做任何处理。这样，当有仅有一个匹配处理函数时，内核就避免了无谓的内存申请和复制了。

### 13.5.4 IP协议处理流程

以IPv4的协议栈处理为例，我们首先看IPv4协议如何注册的处理回调函数。在inet\_init中，调用了dev\_add\_pack(&ip\_packet\_type);进行了IPv4协议的注册。ip\_packet\_type的定义为：

static struct packet\_type ip\_packet\_type \_\_read\_mostly = {

.type = cpu\_to\_be16(ETH\_P\_IP),

.func = ip\_rcv,

.gso\_send\_check = inet\_gso\_send\_check,

.gso\_segment = inet\_gso\_segment,

.gro\_receive = inet\_gro\_receive,

.gro\_complete = inet\_gro\_complete,

};

因此，ip\_rcv为IPv4协议数据包的入口函数。

int ip\_rcv(struct sk\_buff \*skb, struct net\_device \*dev, struct packet\_type \*pt, struct net\_device \*orig\_dev)

{

const struct iphdr \*iph;

u32 len;

/\*

如果该数据是发给其它终端的，则丢弃这个数据包。

这里的pkt\_type是根据二层地址判断是否发给本机的。

\*/

if (skb->pkt\_type == PACKET\_OTHERHOST)

goto drop;

IP\_UPD\_PO\_STATS\_BH(dev\_net(dev), IPSTATS\_MIB\_IN, skb->len);

/\* 对数据包进行共享检查，保证IP协议处理独享一个skb。 \*/

if ((skb = skb\_share\_check(skb, GFP\_ATOMIC)) == NULL) {

IP\_INC\_STATS\_BH(dev\_net(dev), IPSTATS\_MIB\_INDISCARDS);

goto out;

}

/\* 对IP首部检查，如果数据包小于IP首部大小，则出错 \*/

if (!pskb\_may\_pull(skb, sizeof(struct iphdr)))

goto inhdr\_error;

/\* 得到IP首部地址 \*/

iph = ip\_hdr(skb);

/\* 根据RFC标准，IP首部小于20字节或者版本号不是4的，就报错丢弃 \*/

if (iph->ihl < 5 || iph->version != 4)

goto inhdr\_error;

/\* 根据IP首部指定的长度，再次检查数据包的大小 \*/

if (!pskb\_may\_pull(skb, iph->ihl\*4))

goto inhdr\_error;

/\* 重新获取IP首部地址。之所以要重新获取，因为pskb\_may\_pull可能重新申请skb \*/

iph = ip\_hdr(skb);

/\* 校验IPv4的校验和 \*/

if (unlikely(ip\_fast\_csum((u8 \*)iph, iph->ihl)))

goto inhdr\_error;

/\* 对数据包长度进行检查 \*/

len = ntohs(iph->tot\_len);

if (skb->len < len) {

IP\_INC\_STATS\_BH(dev\_net(dev), IPSTATS\_MIB\_INTRUNCATEDPKTS);

goto drop;

} else if (len < (iph->ihl\*4))

goto inhdr\_error;

/\* Our transport medium may have padded the buffer out. Now we know it

\* is IP we can trim to the true length of the frame.

\* Note this now means skb->len holds ntohs(iph->tot\_len).

\*/

/\* 因为传输媒介可能会给数据包进行补齐，现在我们已经根据IP首部明确了数据包的长度，因此需要将数据包长度变为真正的长度并改变 \*/

if (pskb\_trim\_rcsum(skb, len)) {

IP\_INC\_STATS\_BH(dev\_net(dev), IPSTATS\_MIB\_INDISCARDS);

goto drop;

}

/\* 重置数据包的控制块信息 \*/

memset(IPCB(skb), 0, sizeof(struct inet\_skb\_parm));

/\* 重置数据包的套接字 \*/

skb\_orphan(skb);

/\* 遍历执行netfilter在PREROUTING点上的规则，如果数据包没有被丢弃，则进入ip\_rcv\_finish \*/

return NF\_HOOK(NFPROTO\_IPV4, NF\_INET\_PRE\_ROUTING, skb, dev, NULL,

ip\_rcv\_finish);

inhdr\_error:

IP\_INC\_STATS\_BH(dev\_net(dev), IPSTATS\_MIB\_INHDRERRORS);

drop:

kfree\_skb(skb);

out:

return NET\_RX\_DROP;

}

本文不分析netfilter的相关代码，那么数据包经过netfilter的PREROUTING处的规则后，进入了ip\_rcv\_finish。

static int ip\_rcv\_finish(struct sk\_buff \*skb)

{

const struct iphdr \*iph = ip\_hdr(skb);

struct rtable \*rt;

/\* 如果数据包没有设置路由信息，则进行路由查询 \*/

if (skb\_dst(skb) == NULL) {

int err = ip\_route\_input\_noref(skb, iph->daddr, iph->saddr,

iph->tos, skb->dev);

if (unlikely(err)) {

/\* 查找路由失败，增加相应的错误计数，并丢弃数据包 \*/

if (err == -EHOSTUNREACH)

IP\_INC\_STATS\_BH(dev\_net(skb->dev),

IPSTATS\_MIB\_INADDRERRORS);

else if (err == -ENETUNREACH)

IP\_INC\_STATS\_BH(dev\_net(skb->dev),

IPSTATS\_MIB\_INNOROUTES);

else if (err == -EXDEV)

NET\_INC\_STATS\_BH(dev\_net(skb->dev),

LINUX\_MIB\_IPRPFILTER);

goto drop;

}

}

#ifdef CONFIG\_IP\_ROUTE\_CLASSID

if (unlikely(skb\_dst(skb)->tclassid)) {

struct ip\_rt\_acct \*st = this\_cpu\_ptr(ip\_rt\_acct);

u32 idx = skb\_dst(skb)->tclassid;

st[idx&0xFF].o\_packets++;

st[idx&0xFF].o\_bytes += skb->len;

st[(idx>>16)&0xFF].i\_packets++;

st[(idx>>16)&0xFF].i\_bytes += skb->len;

}

#endif

/\* iph大于5，即首部长度大于了固定长度20字节。因此说明该IP报文还有IP选项，那么调用ip\_rcv\_options处理IP选项 \*/

if (iph->ihl > 5 && ip\_rcv\_options(skb))

goto drop;

/\* 根据路由类型，增加相应的计数 \*/

rt = skb\_rtable(skb);

if (rt->rt\_type == RTN\_MULTICAST) {

IP\_UPD\_PO\_STATS\_BH(dev\_net(rt->dst.dev), IPSTATS\_MIB\_INMCAST,

skb->len);

} else if (rt->rt\_type == RTN\_BROADCAST)

IP\_UPD\_PO\_STATS\_BH(dev\_net(rt->dst.dev), IPSTATS\_MIB\_INBCAST,

skb->len);

/\* 调用路由的输入出入函数 \*/

return dst\_input(skb);

drop:

kfree\_skb(skb);

return NET\_RX\_DROP;

}

对于发往本机的数据包来说，其路由输入函数为ip\_local\_deviver。

int ip\_local\_deliver(struct sk\_buff \*skb)

{

/\* 该数据包是一个IP分片数据包 \*/

if (ip\_is\_fragment(ip\_hdr(skb))) {

if (ip\_defrag(skb, IP\_DEFRAG\_LOCAL\_DELIVER))

return 0;

}

/\* 遍历执行netfilter在LOCAL\_IN上的规则，如为丢弃，则进入ip\_local\_deliver\_finish \*/

return NF\_HOOK(NFPROTO\_IPV4, NF\_INET\_LOCAL\_IN, skb, skb->dev, NULL,

ip\_local\_deliver\_finish);

}

进入ip\_local\_deliver\_finish，

static int ip\_local\_deliver\_finish(struct sk\_buff \*skb)

{

struct net \*net = dev\_net(skb->dev);

/\* 拉出IP报文首部 \*/

\_\_skb\_pull(skb, ip\_hdrlen(skb));

/\* 设置传输层首部地址 \*/

skb\_reset\_transport\_header(skb);

rcu\_read\_lock();

{

/\* 得到传输层协议 \*/

int protocol = ip\_hdr(skb)->protocol;

int hash, raw;

const struct net\_protocol \*ipprot;

resubmit:

/\* 将数据包传递给对应的原始套接字 \*/

raw = raw\_local\_deliver(skb, protocol);

/\* 根据传输协议确定对应的inet协议 \*/

hash = protocol & (MAX\_INET\_PROTOS - 1);

ipprot = rcu\_dereference(inet\_protos[hash]);

if (ipprot != NULL) {

/\* 找到了匹配传输层协议 \*/

int ret;

/\* 检查名称空间是否匹配 \*/

if (!net\_eq(net, &init\_net) && !ipprot->netns\_ok) {

if (net\_ratelimit())

printk("%s: proto %d isn't netns-ready\n",

\_\_func\_\_, protocol);

kfree\_skb(skb);

goto out;

}

/\* 协议的安全策略检查 \*/

if (!ipprot->no\_policy) {

if (!xfrm4\_policy\_check(NULL, XFRM\_POLICY\_IN, skb)) {

kfree\_skb(skb);

goto out;

}

nf\_reset(skb);

}

/\* 将数据包传递给传输层处理 \*/

ret = ipprot->handler(skb);

if (ret < 0) {

protocol = -ret;

goto resubmit;

}

IP\_INC\_STATS\_BH(net, IPSTATS\_MIB\_INDELIVERS);

} else {

/\* 没有对应的传输层协议 \*/

if (!raw) {

/\* 没有匹配的原始套接字，进行安全策略检查 \*/

if (xfrm4\_policy\_check(NULL, XFRM\_POLICY\_IN, skb)) {

/\* 没有对应的安全策略，则使用ICMP返回不可达错误 \*/

IP\_INC\_STATS\_BH(net, IPSTATS\_MIB\_INUNKNOWNPROTOS);

icmp\_send(skb, ICMP\_DEST\_UNREACH,

ICMP\_PROT\_UNREACH, 0);

}

} else

IP\_INC\_STATS\_BH(net, IPSTATS\_MIB\_INDELIVERS);

kfree\_skb(skb);

}

}

out:

rcu\_read\_unlock();

return 0;

}

### 13.5.5 大师的错误？原始套接字的接收

在UNP1的28.4 Raw Socket Input一节中，Stevens大师是这样说的：

Received UDP packets and received TCP packets are never passed to a raw socket. If a process wants to read IP datagrams containing UDP or TCP packets, the packets must be read at the datalink layer, as described in Chapter 29.

中文版的翻译是这样的

接收到UDP分组和TCP分组绝不传递到任何原始套接口。如果一个进程想要读取含有UDP分组或TCP分组的IP数据报，它就必须在数据链路层读取这些分组。

这里得吐槽一下中文版的翻译，上文中的“分组”实在是不专业，因为这不是一个准确的术语。当读者看到这个部分后，绝对会很疑惑。分组？何谓分组？是分片的笔误还是组播？我自己也是对照了英文原版后，才明白中文版的意思。与其用一个模糊的“分组”，还不如直接用“报文”更直截了当呢。

回到正题，根据UNP1的说法，raw socket是无法收到TCP和UDP的数据包的。而实际上Linux内核的实际行为却不是这样的。下面让我们用代码来说明：

int raw\_local\_deliver(struct sk\_buff \*skb, int protocol)

{

int hash;

struct sock \*raw\_sk;

/\* 根据传输层协议确定hash桶索引 \*/

hash = protocol & (RAW\_HTABLE\_SIZE - 1);

/\* 获得该桶的头结点 \*/

raw\_sk = sk\_head(&raw\_v4\_hashinfo.ht[hash]);

/\* 当头结点不为空时，才进入raw\_v4\_input做进一步检查 \*/

if (raw\_sk && !raw\_v4\_input(skb, ip\_hdr(skb), hash)) {

/\* 如果没有找到匹配的原始套接字，则重置raw\_sk为NULL。 \*/

raw\_sk = NULL;

}

return raw\_sk != NULL;

}

然后进入raw\_v4\_input，

static int raw\_v4\_input(struct sk\_buff \*skb, const struct iphdr \*iph, int hash)

{

struct sock \*sk;

struct hlist\_head \*head;

int delivered = 0;

struct net \*net;

/\* 与raw\_local\_deliver不同，因为需要使用到头结点中的内容，所以需要对这个桶上锁，才能保证在处理这个桶的过程中，所有节点都是有效的。\*/

read\_lock(&raw\_v4\_hashinfo.lock);

/\* 再次检查头结点 \*/

head = &raw\_v4\_hashinfo.ht[hash];

if (hlist\_empty(head))

goto out;

/\* 获得网络名称空间 \*/

net = dev\_net(skb->dev);

/\* 查询匹配的原始套接字 \*/

sk = \_\_raw\_v4\_lookup(net, \_\_sk\_head(head), iph->protocol,

iph->saddr, iph->daddr,

skb->dev->ifindex);

/\* 找到了匹配的原始套接字，则继续处理 \*/

while (sk) {

delivered = 1;

/\* 如果数据包不是ICMP数据包或者不是被指定要过滤的ICMP类型 \*/

if (iph->protocol != IPPROTO\_ICMP || !icmp\_filter(sk, skb)) {

/\* 数据包要发给该套接字，需要clone一个新的skb \*/

struct sk\_buff \*clone = skb\_clone(skb, GFP\_ATOMIC);

/\* clone成功，则调用原始套接字的接收函数 \*/

if (clone)

raw\_rcv(sk, clone);

}

/\* 继续查询后面的套接字 \*/

sk = \_\_raw\_v4\_lookup(net, sk\_next(sk), iph->protocol,

iph->saddr, iph->daddr,

skb->dev->ifindex);

}

out:

read\_unlock(&raw\_v4\_hashinfo.lock);

return delivered;

}

进入\_\_raw\_v4\_lookup，

static struct sock \*\_\_raw\_v4\_lookup(struct net \*net, struct sock \*sk,

unsigned short num, \_\_be32 raddr, \_\_be32 laddr, int dif)

{

struct hlist\_node \*node;

/\* 遍历套接字 \*/

sk\_for\_each\_from(sk, node) {

struct inet\_sock \*inet = inet\_sk(sk);

/\*

检查如下几个条件：

1. 检查名称空间；

2. 比较协议号

3. 如果套接字设置了目的地址且地址相同；

4. 如果套接字设置了源地址且地址相同

5. 如果套接字绑定了网卡，且网卡相同

只有当以上五个条件都匹配的时候，该套接字则匹配。

\*/

if (net\_eq(sock\_net(sk), net) && inet->inet\_num == num &&

!(inet->inet\_daddr && inet->inet\_daddr != raddr) &&

!(inet->inet\_rcv\_saddr && inet->inet\_rcv\_saddr != laddr) &&

!(sk->sk\_bound\_dev\_if && sk->sk\_bound\_dev\_if != dif))

goto found; /\* gotcha \*/

}

sk = NULL;

found:

return sk;

}

上面的匹配条件中，源地址、目的地址和绑定网卡是通过原始套接字调用connect、bind等系统调用设置的过滤条件。增加这些过滤条件，一般是应用层为了减少不必要的消耗，过滤不需要的数据包。我们可以发现，在原始套接字的接收流程中，没有对TCP和UDP进行的任何的限制。也就是说在Linux环境下，原始套接字完全可以接受TCP和UDP数据包，这与UNP的描述不符。那这是怎么回事呢？因为Stevens大师的UNP针对的是Unix环境的网络编程，而Linux虽然是与Unix兼容，但在细节的实现上必然与Unix有所不同。需要注意的是Stevens大师的另一本经典书籍APUE，也是针对Unix环境的介绍。

### 13.5.6 注册传输层协议

在13.5.4节中的ip\_local\_deliver\_finish函数中，内核通过调用ipprot->handler(skb)将数据包传递给正确的传输层协议。对于IPv4协议来说，其传输层协议的处理函数的handler是在inet\_init中添加的。下面是inet\_init中的部分代码：

/\* 添加ICMP协议 \*/

if (inet\_add\_protocol(&icmp\_protocol, IPPROTO\_ICMP) < 0)

printk(KERN\_CRIT "inet\_init: Cannot add ICMP protocol\n");

/\* 添加UDP协议 \*/

if (inet\_add\_protocol(&udp\_protocol, IPPROTO\_UDP) < 0)

printk(KERN\_CRIT "inet\_init: Cannot add UDP protocol\n");

/\* 添加TCP协议 \*/

if (inet\_add\_protocol(&tcp\_protocol, IPPROTO\_TCP) < 0)

printk(KERN\_CRIT "inet\_init: Cannot add TCP protocol\n");

#ifdef CONFIG\_IP\_MULTICAST

/\* 添加IGMP协议 \*/

if (inet\_add\_protocol(&igmp\_protocol, IPPROTO\_IGMP) < 0)

printk(KERN\_CRIT "inet\_init: Cannot add IGMP protocol\n");

#endif

通过调用inet\_add\_protocol，传输层将自己的处理函数添加到了inet\_protos中，这样就可以在ip\_local\_deliver\_finish中调用对应的传输层的处理函数。

在inet\_init中的另一部分代码：

for (q = inetsw\_array; q < &inetsw\_array[INETSW\_ARRAY\_LEN]; ++q)

inet\_register\_protosw(q);

这部分代码是用于注册AF\_INET的各种协议，如UDP，TCP等。那么为什么inet会使用两种不同的方式来支持传输层协议的注册呢？何不合并为一个结构呢？在我看来，inet\_add\_protocol面向的底层接口，而inet\_register\_protosw面向的是上层应用，所以将其分为了两个结构。

### 13.5.7 确定UDP套接字

UDP协议的面向底层接口的处理结构为

static const struct net\_protocol udp\_protocol = {

.handler = udp\_rcv,

.err\_handler = udp\_err,

.gso\_send\_check = udp4\_ufo\_send\_check,

.gso\_segment = udp4\_ufo\_fragment,

.no\_policy = 1,

.netns\_ok = 1,

};

因此，如果是UDP数据包，会进入udp\_rcv-> \_\_udp4\_lib\_rcv。

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);

/\* 校验数据包至少要有UDP首部大小 \*/

if (!pskb\_may\_pull(skb, sizeof(struct udphdr)))

goto drop; /\* No space for header. \*/

/\* 得到UDP首部指针 \*/

uh = udp\_hdr(skb);

/\* 得到UDP数据包长度 \*/

ulen = ntohs(uh->len);

saddr = ip\_hdr(skb)->saddr;

daddr = ip\_hdr(skb)->daddr;

/\* 如果UDP数据包长度超过数据包的实际长度，则出错 \*/

if (ulen > skb->len)

goto short\_packet;

/\*

判断协议是否为UDP协议。

也许有的读者会觉得奇怪，为什么在UDP的接收函数中还要判断协议是否为UDP？

因为这个函数还用于处理UDPLITE协议。

\*/

if (proto == IPPROTO\_UDP) {

/\* 如果是UDP协议，则将数据包的长度更新为UDP指定长度，并更新校验和 \*/

if (ulen < sizeof(\*uh) || pskb\_trim\_rcsum(skb, ulen))

goto short\_packet;

/\* 因为前面的操作可能会导致skb内存变化，所以需要重新得到UDP首部指针 \*/

uh = udp\_hdr(skb);

}

/\* 初始化UDP校验和 \*/

if (udp4\_csum\_init(skb, uh, proto))

goto csum\_error;

/\* 如果路由标志位广播或多播，表明该UDP数据包为广播或者多播 \*/

if (rt->rt\_flags & (RTCF\_BROADCAST|RTCF\_MULTICAST))

return \_\_udp4\_lib\_mcast\_deliver(net, skb, uh,

saddr, daddr, udptable);

/\* 确定匹配的UDP套接字 \*/

sk = \_\_udp4\_lib\_lookup\_skb(skb, uh->source, uh->dest, udptable);

if (sk != NULL) {

/\* 找到了匹配的套接字 \*/

/\* 将数据包加入到UDP的接收队列 \*/

int ret = udp\_queue\_rcv\_skb(sk, skb);

sock\_put(sk);

/\* a return value > 0 means to resubmit the input, but

\* it wants the return to be -protocol, or 0

\*/

if (ret > 0)

return -ret;

return 0;

}

/\* 进行xfrm策略检查 \*/

if (!xfrm4\_policy\_check(NULL, XFRM\_POLICY\_IN, skb))

goto drop;

/\* 重置netfilter信息 \*/

nf\_reset(skb);

/\* 检查UDP检验和 \*/

if (udp\_lib\_checksum\_complete(skb))

goto csum\_error;

/\* 没有知道匹配的UDP套接字，则发送ICMP错误消息 \*/

UDP\_INC\_STATS\_BH(net, UDP\_MIB\_NOPORTS, proto == IPPROTO\_UDPLITE);

icmp\_send(skb, ICMP\_DEST\_UNREACH, ICMP\_PORT\_UNREACH, 0);

/\*

\* Hmm. We got an UDP packet to a port to which we

\* don't wanna listen. Ignore it.

\*/

kfree\_skb(skb);

return 0;

/\* 错误处理 \*/

…… ……

}

下面看一下如何匹配的UDP套接字，请看\_\_udp4\_lib\_lookup\_skb->\_\_udp4\_lib\_lookup函数：

static 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);

/\* 使用目的端口确定hash桶索引 \*/

unsigned int hash2, slot2, slot = udp\_hashfn(net, hnum, udptable->mask);

struct udp\_hslot \*hslot2, \*hslot = &udptable->hash[slot];

int score, badness;

rcu\_read\_lock();

/\* 该桶的套接字个数多于10个，则需要再次定位 \*/

if (hslot->count > 10) {

/\* 使用目的地址和目的端口确定hash桶索引 \*/

hash2 = udp4\_portaddr\_hash(net, daddr, hnum);

slot2 = hash2 & udptable->mask;

/\*

UDP套接字表维护了两个hash表。

第一个hash表，使用端口来索引；

第二个hash表，使用地址+端口来索引；

在进行UDP套接字匹配的时候，优先使用第一个hash表，因为第一个hash表中使用的是端口进行散列索引，那么只要端口相同，无论是监听的指定IP还是任意IP，都可以在一个桶中进行匹配。但是由于端口只有65535种可能，所以可能导致不够散列，一个桶的套接字个数会比较多。而第二个hash表是使用地址+端口来索引，因此理论上套接字的分布比第一个hash表更加分散。

因此当第一个hash表对应桶的套接字多于10个时，内核会尝试去第二个hash进行匹配查找。

\*/

hslot2 = &udptable->hash2[slot2];

/\* 尽管第二个hash表理论上会比第一个hash表分散，但是如果实际上还是可能第二个表的桶中套接字个数还是大于第一个表的桶中套接字个数，则这时还是利用第一个hash表进行匹配 \*/

if (hslot->count < hslot2->count)

goto begin;

/\* 在第二个hash表的桶中匹配查找套接字 \*/

result = udp4\_lib\_lookup2(net, saddr, sport,

daddr, hnum, dif,

hslot2, slot2);

if (!result) {

/\* 利用指定IP和端口没有在该桶中找到匹配的套接字，则常使用任意IP+端口来进行散列索引 \*/

hash2 = udp4\_portaddr\_hash(net, htonl(INADDR\_ANY), hnum);

slot2 = hash2 & udptable->mask;

hslot2 = &udptable->hash2[slot2];

/\* 还是要与第一个hash桶中的个数进行比较 \*/

if (hslot->count < hslot2->count)

goto begin;

/\* 在第二个hash表中使用任意IP+端口进行匹配查找 \*/

result = udp4\_lib\_lookup2(net, saddr, sport,

htonl(INADDR\_ANY), hnum, dif,

hslot2, slot2);

}

rcu\_read\_unlock();

return result;

}

begin:

result = NULL;

badness = -1;

/\* 在第一个hash表的桶中进行查找 \*/

sk\_nulls\_for\_each\_rcu(sk, node, &hslot->head) {

/\* 计算该套接字的匹配得分 \*/

score = compute\_score(sk, net, saddr, hnum, sport,

daddr, dport, dif);

/\* 保证匹配得分最高的套接字为最终结果 \*/

if (score > badness) {

result = sk;

badness = score;

}

}

/\*

检查是否在查找的过程中，我们遇到了某个套接字被移到另外一个桶内。

这时，需要重新进行匹配。

\*/

if (get\_nulls\_value(node) != slot)

goto begin;

/\* 找到了匹配的套接字 \*/

if (result) {

/\* 增加套接字引用计数 \*/

if (unlikely(!atomic\_inc\_not\_zero\_hint(&result->sk\_refcnt, 2)))

result = NULL;

/\* 再次计算套接字得分，如小于最大分数，则重新匹配查找 \*/

else if (unlikely(compute\_score(result, net, saddr, hnum, sport,

daddr, dport, dif) < badness)) {

sock\_put(result);

goto begin;

}

}

rcu\_read\_unlock();

return result;

}

从上面的代码中，可以看到匹配UDP套接字的关键在于对应套接字的匹配得分。第一个hash表的得分计算函数为compute\_score：

static inline int compute\_score(struct sock \*sk, struct net \*net, \_\_be32 saddr,

unsigned short hnum,

\_\_be16 sport, \_\_be32 daddr, \_\_be16 dport, int dif)

{

int score = -1;

/\* 比较名称空间，端口等 \*/

if (net\_eq(sock\_net(sk), net) && udp\_sk(sk)->udp\_port\_hash == hnum &&

!ipv6\_only\_sock(sk)) {

struct inet\_sock \*inet = inet\_sk(sk);

/\* 套接字指明为PF\_INET，加1分 \*/

score = (sk->sk\_family == PF\_INET ? 1 : 0);

/\* 套接字绑定了接收地址 \*/

if (inet->inet\_rcv\_saddr) {

/\* 如果数据包的目的地址与绑定接收地址不符，则分数为-1，相同则增加2分。 \*/

if (inet->inet\_rcv\_saddr != daddr)

return -1;

score += 2;

}

/\* 套接字设置了对端目的地址 \*/

if (inet->inet\_daddr) {

/\* 如果数据包的源地址与设置的目的地址不同，则分数为-1，相同则增加2分 \*/

if (inet->inet\_daddr != saddr)

return -1;

score += 2;

}

/\* 套接字设置了对端目的端口 \*/

if (inet->inet\_dport) {

/\* 如果数据包的源端口与设置的目的端口不同，则分数为-1，相同则增加2分 \*/

if (inet->inet\_dport != sport)

return -1;

score += 2;

}

/\* 套接字绑定了网卡 \*/

if (sk->sk\_bound\_dev\_if) {

/\* 如果接受数据包的网卡与绑定网卡不同，则分数为-1，相同则增加2分 \*/

if (sk->sk\_bound\_dev\_if != dif)

return -1;

score += 2;

}

}

return score;

}

对应第二个hash来说，其匹配分数计算函数为compute\_score2，算法与compute\_score基本相同。总的来说UDP的套接字匹配有以下几个条件：

1. 接收端口：必须匹配；

2. 接收地址：如绑定了则必须匹配，分值2分；

3. 对端目的地址：如设置了则必须匹配，分值2分；

4. 对端目的端口：如设置了则必须匹配，分值2分；

5. 网卡：如绑定了则必须匹配，分值2分；

6. 套接字设置了PF\_INET协议族，则分值1分；

根据上面的规则，匹配分值最高的套接字为选中的UDP套接字，然后内核会将这个数据包加入到该UDP套接字的接收队列中。也就是说，及时数据包可以匹配多个UDP套接字（这是很有可能的），那么最终也只有一个最匹配的套接字被选中，只有这个套接字可以收到数据包。

### 13.5.8 确定TCP套接字

TCP面向底层接口的处理结构是

static const struct net\_protocol tcp\_protocol = {

.handler = tcp\_v4\_rcv,

.err\_handler = tcp\_v4\_err,

.gso\_send\_check = tcp\_v4\_gso\_send\_check,

.gso\_segment = tcp\_tso\_segment,

.gro\_receive = tcp4\_gro\_receive,

.gro\_complete = tcp4\_gro\_complete,

.no\_policy = 1,

.netns\_ok = 1,

};

那么，如果是TCP数据包，则会进入tcp\_v4\_rcv：

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);

/\* 丢弃不是发给自己的数据包 \*/

if (skb->pkt\_type != PACKET\_HOST)

goto discard\_it;

/\* Count it even if it's bad \*/

TCP\_INC\_STATS\_BH(net, TCP\_MIB\_INSEGS);

/\* 检查数据包至少要有TCP固定首部的大小 \*/

if (!pskb\_may\_pull(skb, sizeof(struct tcphdr)))

goto discard\_it;

/\* 获得TCP首部地址 \*/

th = tcp\_hdr(skb);

/\* 检查TCP的数据偏移量是否合法 ，不能小于TCP首部固定大小 \*/

if (th->doff < sizeof(struct tcphdr) / 4)

goto bad\_packet;

/\* 检查数据包的大小是否满足TCP指定的数据偏移位置 \*/

if (!pskb\_may\_pull(skb, th->doff \* 4))

goto discard\_it;

/\* 如果需要检查检验和，则进行校验和初始化 \*/

if (!skb\_csum\_unnecessary(skb) && tcp\_v4\_checksum\_init(skb))

goto bad\_packet;

/\* 因为skb可能会重新申请，所以需要重新得到TCP首部 \*/

th = tcp\_hdr(skb);

iph = ip\_hdr(skb);

/\* 根据TCP报文信息，设置skb的TCP控制块 \*/

TCP\_SKB\_CB(skb)->seq = ntohl(th->seq);

TCP\_SKB\_CB(skb)->end\_seq = (TCP\_SKB\_CB(skb)->seq + th->syn + th->fin +

skb->len - th->doff \* 4);

TCP\_SKB\_CB(skb)->ack\_seq = ntohl(th->ack\_seq);

TCP\_SKB\_CB(skb)->when = 0;

TCP\_SKB\_CB(skb)->ip\_dsfield = ipv4\_get\_dsfield(iph);

TCP\_SKB\_CB(skb)->sacked = 0;

/\* 查找匹配的TCP套接字 \*/

sk = \_\_inet\_lookup\_skb(&tcp\_hashinfo, skb, th->source, th->dest);

if (!sk)

goto no\_tcp\_socket;

/\* 找到匹配的套接字开始处理，不是本文重点，因此省略后面的分析 \*/

process:

…… ……

}

进入\_\_inet\_lookup\_skb->\_\_inet\_lookup，

static inline struct sock \*\_\_inet\_lookup(struct net \*net,

struct inet\_hashinfo \*hashinfo,

const \_\_be32 saddr, const \_\_be16 sport,

const \_\_be32 daddr, const \_\_be16 dport,

const int dif)

{

u16 hnum = ntohs(dport);

/\* 先在已连接的套接字进行查找 \*/

struct sock \*sk = \_\_inet\_lookup\_established(net, hashinfo,

saddr, sport, daddr, hnum, dif);

/\* 优先使用已连接的套接字，若没有则在监听的套接字中进行查找 \*/

return sk ? : \_\_inet\_lookup\_listener(net, hashinfo, daddr, hnum, dif);

}

那么，我们先看\_\_inet\_lookup\_established，

struct sock \* \_\_inet\_lookup\_established(struct net \*net,

struct inet\_hashinfo \*hashinfo,

const \_\_be32 saddr, const \_\_be16 sport,

const \_\_be32 daddr, const u16 hnum,

const int dif)

{

INET\_ADDR\_COOKIE(acookie, saddr, daddr)

const \_\_portpair ports = INET\_COMBINED\_PORTS(sport, hnum);

struct sock \*sk;

const struct hlist\_nulls\_node \*node;

/\* Optimize here for direct hit, only listening connections can

\* have wildcards anyways.

\*/

/\* 根据目的地址，目的端口，源地址，源端口计算得到已连接hash表的桶索引 \*/

unsigned int hash = inet\_ehashfn(net, daddr, hnum, saddr, sport);

unsigned int slot = hash & hashinfo->ehash\_mask;

struct inet\_ehash\_bucket \*head = &hashinfo->ehash[slot];

rcu\_read\_lock();

begin:

/\* 遍历该桶节点 \*/

sk\_nulls\_for\_each\_rcu(sk, node, &head->chain) {

/\*

比较源地址，目的地址，源端口，目的端口以及接收网卡。

细心的读者会发现这里有两个参数acookie和ports，这两个参数用于加速匹配。在64位机器上，acookie为源地址和目的地址合成的64位整数，32位机器上acookie无意义。ports为源端口和目的端口合成的32位整数。通过直接比较组合的整数，可以加速匹配。

\*/

if (INET\_MATCH(sk, net, hash, acookie,

saddr, daddr, ports, dif)) {

/\* 增加套接字引用计数 \*/

if (unlikely(!atomic\_inc\_not\_zero(&sk->sk\_refcnt)))

goto begintw;

/\* 再次检测，防止套接字被改变 \*/

if (unlikely(!INET\_MATCH(sk, net, hash, acookie,

saddr, daddr, ports, dif))) {

sock\_put(sk);

goto begin;

}

goto out;

}

}

/\*

\* if the nulls value we got at the end of this lookup is

\* not the expected one, we must restart lookup.

\* We probably met an item that was moved to another chain.

\*/

if (get\_nulls\_value(node) != slot)

goto begin;

begintw:

/\* 如果在已连接的hash表中找不到对应的套接字，需要到TIME\_WAIT状态连接的hash表中查找套接字，原理与上面相同。这说明TIME\_WAIT状态的连接如果依然存在，会优先于监听套接字。 \*/

sk\_nulls\_for\_each\_rcu(sk, node, &head->twchain) {

if (INET\_TW\_MATCH(sk, net, hash, acookie,

saddr, daddr, ports, dif)) {

if (unlikely(!atomic\_inc\_not\_zero(&sk->sk\_refcnt))) {

sk = NULL;

goto out;

}

if (unlikely(!INET\_TW\_MATCH(sk, net, hash, acookie,

saddr, daddr, ports, dif))) {

sock\_put(sk);

goto begintw;

}

goto out;

}

}

/\* 省略 \*/

…… ……

}

如果在已连接和TIME\_WAIT状态的hash表中，都没有找到匹配的套接字。这时就需要到监听hash表中查找匹配的套接字。

struct sock \*\_\_inet\_lookup\_listener(struct net \*net,

struct inet\_hashinfo \*hashinfo,

const \_\_be32 daddr, const unsigned short hnum,

const int dif)

{

struct sock \*sk, \*result;

struct hlist\_nulls\_node \*node;

/\* 根据源端口计算监听hash表对应的桶索引 \*/

unsigned int hash = inet\_lhashfn(net, hnum);

struct inet\_listen\_hashbucket \*ilb = &hashinfo->listening\_hash[hash];

int score, hiscore;

rcu\_read\_lock();

begin:

result = NULL;

hiscore = -1;

/\* 遍历该桶中的套接字，与UDP相似，得分最高的套接字为匹配套接字 \*/

sk\_nulls\_for\_each\_rcu(sk, node, &ilb->head) {

/\* 计算套接字的得分 \*/

score = compute\_score(sk, net, hnum, daddr, dif);

if (score > hiscore) {

result = sk;

hiscore = score;

}

}

/\*

\* if the nulls value we got at the end of this lookup is

\* not the expected one, we must restart lookup.

\* We probably met an item that was moved to another chain.

\*/

if (get\_nulls\_value(node) != hash + LISTENING\_NULLS\_BASE)

goto begin;

/\* 找到了匹配的套接字 \*/

if (result) {

/\* 增加套接字引用计数 \*/

if (unlikely(!atomic\_inc\_not\_zero(&result->sk\_refcnt)))

result = NULL;

/\* 需要再次检查该套接字的得分 \*/

else if (unlikely(compute\_score(result, net, hnum, daddr,

dif) < hiscore)) {

sock\_put(result);

goto begin;

}

}

rcu\_read\_unlock();

return result;

}

匹配TCP监听套接字的流程与UDP基本相同，都是在与套接字的计算得分上。下面我们看一下TCP监听套接字的得分计算函数compute\_score。

static inline int compute\_score(struct sock \*sk, struct net \*net,

const unsigned short hnum, const \_\_be32 daddr,

const int dif)

{

int score = -1;

struct inet\_sock \*inet = inet\_sk(sk);

/\* 必须匹配名称空间，目的端口 \*/

if (net\_eq(sock\_net(sk), net) && inet->inet\_num == hnum &&

!ipv6\_only\_sock(sk)) {

\_\_be32 rcv\_saddr = inet->inet\_rcv\_saddr;

/\* 协议族为PF\_INET，得分加1 \*/

score = sk->sk\_family == PF\_INET ? 1 : 0;

/\*

套接字指定了接收地址，则接收地址必须与目的地址相同。

不同则不匹配，相同则得分加2。

\*/

if (rcv\_saddr) {

if (rcv\_saddr != daddr)

return -1;

score += 2;

}

/\*

如果套接字绑定了网卡，则接收网卡必须相同。

不同则不匹配，相同则得分加2。

\*/

if (sk->sk\_bound\_dev\_if) {

if (sk->sk\_bound\_dev\_if != dif)

return -1;

score += 2;

}

}

return score;

}

这个函数名字和功能均与UDP的相同，但是其实现代码却略有不同。大家可以发现，TCP的compute\_score不会对源端信息进行任何检查，如源地址、源端口。为什么会这样呢？原因在于，TCP的compute\_score是用于监听套接字的匹配。这意味着这是TCP连接的第一个数据包即SYN包，属于连接初始化阶段，这时自然无需对源端信息进行任何检查和匹配。当本机回复了SYN+ACK后，本机会创建已连接套接字插入到已连接hash表中。这样该连接后面的数据包，就会在已连接hash表中找到匹配的套接字，而不会再进入监听套接字的匹配。