Dsr.c
Ack 发送、选项添加、选项接收
Ack-request 创建、添加、发送、接收
Dsr-dev.c

sk_buff（socket buffer）结构是 linux 网络代码中重要的数据结构，它管理和控制接收或发送数据包的信息。

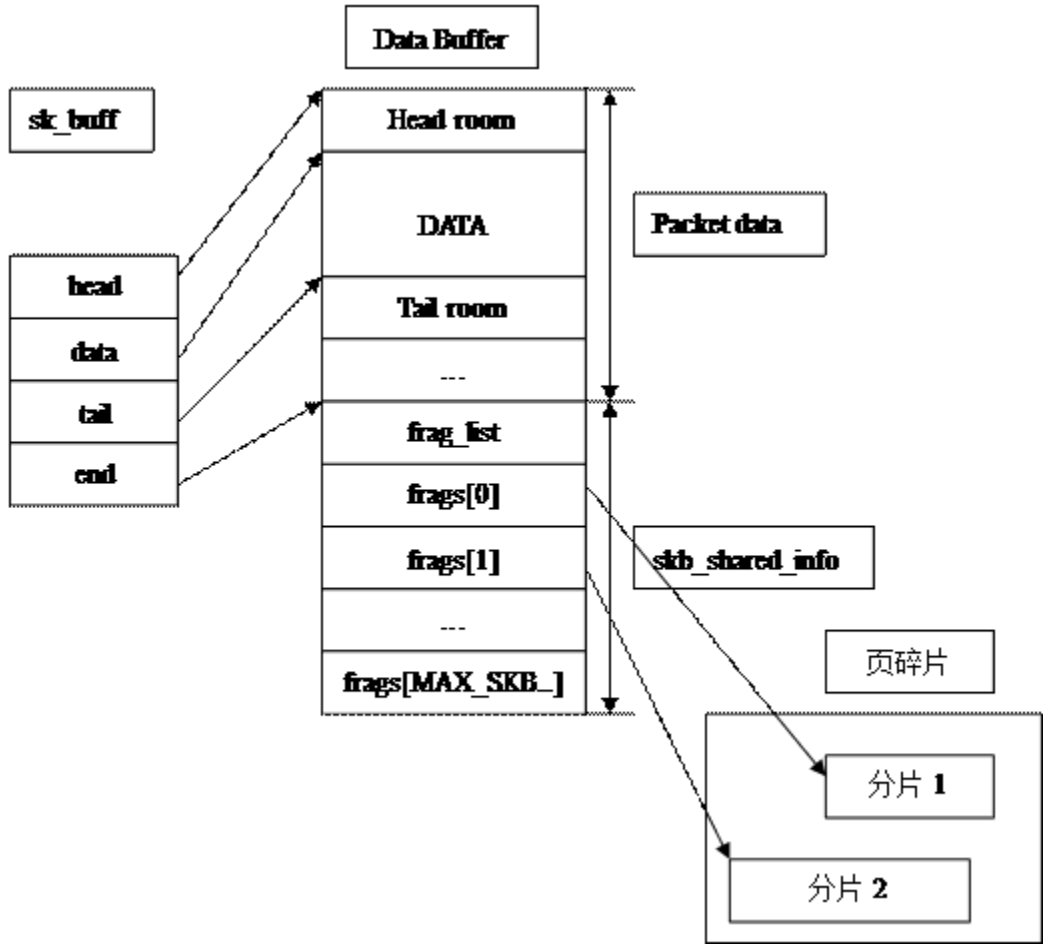
sk_buff 组成

Packet data：通过网卡收发的报文，包括链路层、网络层、传输层的协议头和携带的应用数据，包括 head room,data,tail room 三部分。

skb_shared_info 作为 packet data 的补充，用于存储 ip 分片，其中 sk_buff *frag_list 是一系列子 skbuff 链表，而 frag[]是由一组单独的 page 组成的数据缓冲区。

Data buffer：用于存储 packet data 的缓冲区，分为以上两部分。

Sk_buff：缓冲区控制结构 sk_buff。



```c
struct sk_buff *dsr_skb_create(struct dsr_pkt *dp, struct net_device *dev)
{
    struct sk_buff *skb;
    char *buf;
    int ip_len;
    int tot_len;
    int dsr_opts_len = dsr_pkt_opts_len(dp);

    ip_len = dp->nh.iph->ihl << 2;

    tot_len = ip_len + dsr_opts_len + dp->payload_len;

    DEBUG("ip_len=%d dsr_opts_len=%d payload_len=%d tot_len=%d\n",
          ip_len, dsr_opts_len, dp->payload_len, tot_len);
#ifdef KERNEL26
    skb = alloc_skb(tot_len + LL_RESERVED_SPACE(dev), GFP_ATOMIC);
#else
    skb = alloc_skb(dev->hard_header_len + 15 + tot_len, GFP_ATOMIC);
#endif
    if (!skb) {
        DEBUG("alloc_skb failed\n");
        return NULL;
    }

    /* We align to 16 bytes, for ethernet: 2 bytes + 14 bytes header */
#ifdef KERNEL26
    skb_reserve(skb, LL_RESERVED_SPACE(dev));
#else
    skb_reserve(skb, (dev->hard_header_len + 15) & ~15);
#endif
    skb->mac.raw = skb->data - 14;
    skb->nh.raw = skb->data;
    skb->dev = dev;
    skb->protocol = htons(ETH_P_IP);

    /* Copy in all the headers in the right order */
    buf = skb_put(skb, tot_len);

    memcpy(buf, dp->nh.raw, ip_len);

    /* For some reason the checksum has to be recalculated here, at least
     * when there is a record route IP option */
    ip_send_check((struct iphdr *)buf);

    buf += ip_len;

    /* Add DSR header if it exists */
    if (dsr_opts_len) {
        memcpy(buf, dp->dh.raw, dsr_opts_len);
        buf += dsr_opts_len;
    }

    /* Add payload */
    if (dp->payload_len && dp->payload)
        memcpy(buf, dp->payload, dp->payload_len);

    return skb;
} « end dsr_skb_create »
```

添加 packet 内容到 socket_buffer

Dsr_hardware_header_create

*int dsr_hw_header_create(struct dsr_pkt *dp, struct sk_buff *skb)*

{

```
    struct sockaddr broadcast =
        { AF_UNSPEC, {0xff, 0xff, 0xff, 0xff, 0xff, 0xff} };
    struct neighbor_info neigh_info;

    if (dp->dst.s_addr == DSR_BROADCAST)                    packet 的目的地址为广播
        memcpy(neigh_info.hw_addr.sa_data, broadcast.sa_data, ETH_ALEN);
    else {
        /* Get hardware destination address */
        if (neigh_tbl_query(dp->nxt_hop, &neigh_info) < 0) {
            DEBUG
                ("Could not get hardware address for next hop %s\n",
                 print_ip(dp->nxt_hop));
            return -1;
        }
    }

    if (skb->dev->hard_header) {
        skb->dev->hard_header(skb, skb->dev, ETH_P_IP,
                              neigh_info.hw_addr.sa_data, 0, skb->len);
    } else {
        DEBUG("Missing hard_header\n");
        return -1;
    }
    return 0;
}
```

网络设备地址事件处理

```
static int dsr_dev_inetaddr_event(struct notifier_block *this,
                                  unsigned long event, void *ptr)
{
    struct in_ifaddr *ifa = (struct in_ifaddr *)ptr;
    struct in_device *indev;

    if (!ifa)
        return NOTIFY_DONE;

    indev = ifa->ifa_dev;

    if (!indev)
        return NOTIFY_DONE;

    switch (event) {
    case NETDEV_UP:
        DEBUG("inetdev UP\n");
```

```
            if (indev->dev == dsr_dev) {
                struct dsr_node *dnode;
                struct in_addr addr, bc;

                dnode = (struct dsr_node *)indev->dev->priv;

                dsr_node_lock(dnode);
                dnode->ifaddr.s_addr = ifa->ifa_address;
                dnode->bcaddr.s_addr = ifa->ifa_broadcast;

                dnode->slave_indev = in_dev_get(dnode->slave_dev);

                                /* Disable rp_filter and enable forwarding */
                                if (dnode->slave_indev) {
                                        rp_filter = dnode->slave_indev->cnf.rp_filter;
                                        forwarding   =   dnode->slave_indev->cnf.forwarding;
dnode->slave_indev->cnf.rp_filter = 0;
                                        dnode->slave_indev->cnf.forwarding = 1;
                                }
                dsr_node_unlock(dnode);

                addr.s_addr = ifa->ifa_address;
                bc.s_addr = ifa->ifa_broadcast;

                DEBUG("New ip=%s broadcast=%s\n",
                        print_ip(addr), print_ip(bc));
            }
            break;
        default:
            break;
        };
        return NOTIFY_DONE;
}
```

Dsr-device-netdevice-event 处理

Dsr-device-netdevice 启动、状态改变、关闭

```
static int dsr_dev_netdev_event(struct notifier_block *this,
                        unsigned long event, void *ptr)
{
        struct net_device *dev = (struct net_device *)ptr;
        struct dsr_node *dnode = (struct dsr_node *)dsr_dev->priv;
        int slave_change = 0;
```

```
if (!dev)
        return NOTIFY_DONE;

switch (event) {
case NETDEV_REGISTER:
        DEBUG("Netdev register %s\n", dev->name);
        if (dnode->slave_dev == NULL &&
            strcmp(dev->name, dnode->slave_ifname) == 0) {     当前无且 name 相同

                DEBUG("Slave dev %s up\n", dev->name);

                dsr_node_lock(dnode);
                dnode->slave_dev = dev;
                dev_hold(dev);
                dsr_node_unlock(dnode);

                /* Reduce the MTU to allow DSR options of 100
                 * bytes. If larger, drop or implement
                 * fragmentation... ;-) Alternatively find a
                 * way to dynamically reduce the data size of
                 * packets depending on the size of the DSR
                 * header. */
                dsr_dev->mtu = dev->mtu - DSR_OPTS_MAX_SIZE;

                DEBUG("Registering packet type\n");
                dsr_packet_type.func = dsr_dev_llrecv;
                dsr_packet_type.dev = dev;
                dev_add_pack(&dsr_packet_type);

                slave_change = 1;
        }

        if (slave_change)
                DEBUG("New DSR slave interface %s\n", dev->name);
        break;
case NETDEV_CHANGE:
        DEBUG("Netdev change\n");
        break;
case NETDEV_UP:
        DEBUG("Netdev up %s\n", dev->name);
        if (ConfVal(PromiscOperation) &&
            dev == dsr_dev && dnode->slave_dev)
                dev_set_promiscuity(dnode->slave_dev, +1);
```

```
            break;
        case NETDEV_UNREGISTER:
            DEBUG("Netdev unregister %s\n", dev->name);

            dsr_node_lock(dnode);
            if (dev == dnode->slave_dev) {
                dev_remove_pack(&dsr_packet_type);
                dsr_packet_type.func = NULL;
                slave_change = 1;
                dev_put(dev);
                dnode->slave_dev = NULL;
            }
            dsr_node_unlock(dnode);

            if (slave_change)
                DEBUG("DSR slave interface %s unregisterd\n",
                      dev->name);
            break;
        case NETDEV_DOWN:
            DEBUG("Netdev down %s\n", dev->name);
            if (dev == dsr_dev) {
                if (dnode->slave_dev && ConfVal(PromiscOperation))
                    dev_set_promiscuity(dnode->slave_dev, -1);

                dsr_node_lock(dnode);
                                if (dnode->slave_indev) {
                                        dnode->slave_indev->cnf.rp_filter = rp_filter;
                                        dnode->slave_indev->cnf.forwarding   =   forwarding;
in_dev_put(dnode->slave_indev);
                                        dnode->slave_indev = NULL;
                                }
                                dsr_node_unlock(dnode);
            } else if (dev == dnode->slave_dev && dnode->slave_indev) {
                dsr_node_lock(dnode);
                dnode->slave_indev->cnf.rp_filter = rp_filter;
                dnode->slave_indev->cnf.forwarding                  =                  forwarding;
in_dev_put(dnode->slave_indev);
                dnode->slave_indev = NULL;
                                dsr_node_unlock(dnode);
            }
            break;
        default:
            break;
        };
```

return NOTIFY_DONE;
}

Dsr_dev_start_xmit 发射 dev 开始
Dsr_dev_get_stats 获取状态
Dsr_dev_set_address 设置 dev 的地址
Dsr_dev_accept_fastpath 允许 dev fastpath 可以依据已有状态直接转发的路径 slowpath 需要寻找路由、解析 MAC
以及 open、stop、uninit device

Dsr-io.c
Dsr 接收、开始发送

```
int NSCLASS dsr_recv(struct dsr_pkt *dp)
{
    int i = 0, action;
    int mask = DSR_PKT_NONE;

    /* Process DSR Options */
    action = dsr_opt_recv(dp);

    /* Add mac address of previous hop to the neighbor table */

    if (dp->flags & PKT_PROMISC_RECV) {
        dsr_pkt_free(dp);
        return 0;
    }
    for (i = 0; i < DSR_PKT_ACTION_LAST; i++) {

        switch (action & mask) {
        case DSR_PKT_NONE:
            break;
        case DSR_PKT_DROP:
        case DSR_PKT_ERROR:
            DEBUG("DSR_PKT_DROP or DSR_PKT_ERROR\n");
            dsr_pkt_free(dp);
            return 0;
        case DSR_PKT_SEND_ACK:
            /* Moved to dsr-ack.c */
            break;
        case DSR_PKT_SRT_REMOVE:
            //DEBUG("Remove source route\n");
            // Hmm, we remove the DSR options when we deliver a
            //packet
            //dsr_opt_remove(dp);
            break;
        case DSR_PKT_FORWARD:

#ifdef NS2
            if (dp->nh.iph->ttl() < 1)
#else
            if (dp->nh.iph->ttl < 1)
#endif
            {
```

Action 为收到的 dsr 报文的选项

添加 MAC 地址到前一跳到相邻节点表中
如果 dp 不为零 收到 则返回
I=0; i<12;switch (action&mask mask=1) 处理报文

```
        case DSR_PKT_FORWARD:

#ifdef NS2
            if (dp->nh.iph->ttl() < 1)
#else
            if (dp->nh.iph->ttl < 1)
#endif
            {
                DEBUG("ttl=0, dropping!\n");
                dsr_pkt_free(dp);
                return 0;
            } else {
                DEBUG("Forwarding %s %s nh %s\n",
                      print_ip(dp->src),
                      print_ip(dp->dst), print_ip(dp->nxt_hop));
                XMIT(dp);
                return 0;
            }
            break;
```

主要 case DSR_PKT_FORWARD 2^7
若 ttl<1 则丢弃
否则转发

```
case DSR_PKT_FORWARD_RREQ:
    XMIT(dp);
    return 0;
case DSR_PKT_SEND_RREP:
    /* In dsr-rrep.c */
    break;
case DSR_PKT_SEND_ICMP:
    DEBUG("Send ICMP\n");
    break;
case DSR_PKT_SEND_BUFFERED:
    if (dp->rrep_opt) {
        struct in_addr rrep_srt_dst;
        int i;

        for (i = 0; i < dp->num_rrep_opts; i++) {
            rrep_srt_dst.s_addr = dp->rrep_opt[i]->addrs[DSR_RREP_ADDRS_LEN(dp->rrep_opt[i]) / sizeof(struct in_addr)];

            send_buf_set_verdict(SEND_BUF_SEND, rrep_srt_dst);
        }
    }
        break;
case DSR_PKT_DELIVER:
    DEBUG("Deliver to DSR device\n");
    DELIVER(dp);
    return 0;
case 0:
    break;
default:
    DEBUG("Unknown pkt action\n");
```

DSR-FORWARD-RREQ 转发
DSR-SEND-RREP 路由应答
DSR_PKT_SEND_BUFFERED
发送 buffer 详细在 send-buf.c
DSR-PKT-DELIVER deliver 到 DSR device 函数在 dsr-dev.h

Dsr_start_xmit

```
void NSCLASS dsr_start_xmit(struct dsr_pkt *dp)
{
    int res;

    if (!dp) {
        DEBUG("Could not allocate DSR packet\n");
        return;
    }

    dp->srt = dsr_rtc_find(dp->src, dp->dst);

    if (dp->srt) {

        if (dsr_srt_add(dp) < 0) {
            DEBUG("Could not add source route\n");
            goto out;
        }
        /* Send packet */

        XMIT(dp);

        return;

    } else {
#ifdef NS2
        res = send_buf_enqueue_packet(dp, &DSRUU::ns_xmit);
#else
        res = send_buf_enqueue_packet(dp, &dsr_dev_xmit);
#endif
```

无 packet DEBUG

dp->srt = dsr_rtc_find(dp->src, dp->dst); 寻找源路由

判断能否添加源路由

发送 dp

发送 buffer 入列的 packet

Res<0 buffer full

Res <0 无 route request table entry RREQ transmission failed

```
        if (res < 0) {
            DEBUG("Queueing failed!\n");
            goto out;
        }
        res = dsr_rreq_route_discovery(dp->dst);

        if (res < 0)
            DEBUG("RREQ Transmission failed...");

        return;
    }
  out:
    dsr_pkt_free(dp);
} « end dsr_start_xmit »
```

Dsr-opt.c

```
struct dsr_opt_hdr *dsr_opt_hdr_add(char *buf, unsigned int len,
                    unsigned int protocol)
{
    struct dsr_opt_hdr *opt_hdr;

    if (len < DSR_OPT_HDR_LEN)
        return NULL;

    opt_hdr = (struct dsr_opt_hdr *)buf;

    opt_hdr->nh = protocol;
    opt_hdr->f = 0;
    opt_hdr->res = 0;
    opt_hdr->p_len = htons(len - DSR_OPT_HDR_LEN);

    return opt_hdr;
}
```

添加 dsr-header option

```
struct iphdr *dsr_build_ip(struct dsr_pkt *dp, struct in_addr src,
               struct in_addr dst, int ip_len, int tot_len,
               int protocol, int ttl)
{
    struct iphdr *iph;

    dp->nh.iph = iph = (struct iphdr *)dp->ip_data;

    if (dp->skb && dp->skb->nh.raw) {
        memcpy(dp->ip_data, dp->skb->nh.raw, ip_len);
    } else {
        iph->version = IPVERSION;
        iph->ihl = 5;
        iph->tos = 0;
        iph->id = 0;
        iph->frag_off = 0;
        iph->ttl = (ttl ? ttl : IPDEFTTL);
        iph->saddr = src.s_addr;
        iph->daddr = dst.s_addr;
    }

    iph->tot_len = htons(tot_len);
    iph->protocol = protocol;

    ip_send_check(iph);

    return iph;
} « end dsr_build_ip »
```

构造 IP 报文

```c
struct dsr_opt *dsr_opt_find_opt(struct dsr_pkt *dp, int type)
{
    int dsr_len, l;
    struct dsr_opt *dopt;

    dsr_len = dsr_pkt_opts_len(dp);

    l = DSR_OPT_HDR_LEN;
    dopt = DSR_GET_OPT(dp->dh.opth);

    while (l < dsr_len && (dsr_len - l) > 2) {
        if (type == dopt->type)
            return dopt;

        l += dopt->length + 2;
        dopt = DSR_GET_NEXT_OPT(dopt);
    }
    return NULL;
}
```

发现选项　-2 大概因为选项一个字段为 kind 一个字段为 length

```c
int NSCLASS dsr_opt_remove(struct dsr_pkt *dp)
{
    int len, ip_len, prot, ttl;

    if (!dp || !dp->dh.raw)
        return -1;

    prot = dp->dh.opth->nh;
#ifdef NS2
    ip_len = 20;
    ttl = dp->nh.iph->ttl();
#else
    ip_len = (dp->nh.iph->ihl << 2);
    ttl = dp->nh.iph->ttl;
#endif
    dsr_build_ip(dp, dp->src, dp->dst, ip_len,
                 ip_len + dp->payload_len, prot, ttl);

    len = dsr_pkt_free_opts(dp);

    /* Return bytes removed */
    return len;
} « end dsr_opt_remove »
```

移除选项　--通过截断 IP 前 20 字节

Dsr-opt-parse 解析 dsr 的 opt　服务器接收 SYN 包时

```
        case DSR_OPT_RREQ:
            if (dp->num_rreq_opts == 0)
                dp->rreq_opt = (struct dsr_rreq_opt *)dopt;
#ifndef NS2
            else
                DEBUG("ERROR: More than one RREQ option!!\n");
#endif
            break;
        case DSR_OPT_RREP:
            if (dp->num_rrep_opts < MAX_RREP_OPTS)
                dp->rrep_opt[dp->num_rrep_opts++] = (struct dsr_rrep_opt *)
#ifndef NS2
            else
                DEBUG("Maximum RREP opts in one packet reached\n");
#endif
            break;
        case DSR_OPT_RERR:
            if (dp->num_rerr_opts < MAX_RERR_OPTS)
                dp->rerr_opt[dp->num_rerr_opts++] = (struct dsr_rerr_opt *)
#ifndef NS2
            else
                DEBUG("Maximum RERR opts in one packet reached\n");
#endif
```

不确定 in one packet reached 是指在这之前已经到达还是这个 packet 包含

Dsr-opt-recv

```
int NSCLASS dsr_opt_recv(struct dsr_pkt *dp)
{
    int dsr_len, l;
    int action = 0;
    struct dsr_opt *dopt;
    struct in_addr myaddr;

    if (!dp)
        return DSR_PKT_ERROR;

    myaddr = my_addr();

    /* Packet for us ? */
#ifdef NS2
    //DEBUG("Next header=%s\n", packet_info.name((packet_t)dp->dh.opth->nh));

    if (dp->dst.s_addr == myaddr.s_addr &&
        (DATA_PACKET(dp->dh.opth->nh) || dp->dh.opth->nh == PT_PING))
        action |= DSR_PKT_DELIVER;
#else
    if (dp->dst.s_addr == myaddr.s_addr && dp->payload_len != 0)
        action |= DSR_PKT_DELIVER;
#endif
    dsr_len = dsr_pkt_opts_len(dp);

    l = DSR_OPT_HDR_LEN;
    dopt = DSR_GET_OPT(dp->dh.opth);
```

Myaddr 由 my_addr()获取 dsr_node 的 ifaddr 即 ipaddr
如果 datapacket 的目的地址与 destnode 的 addr 相同且 datapacket 的 header 满足要求 (DATA_PACKET()与 PT_PING 均没找到)
或 datapacket 的目的地址与 destnode 的 addr 相同且 dp 的负载 len 不为 0
Action 为 DSR_PKT_DELIVER

```
while (l < dsr_len && (dsr_len - l) > 2) {
    //DEBUG("dsr_len=%d l=%d\n", dsr_len, l);
    switch (dopt->type) {
    case DSR_OPT_PADN:
        break;
    case DSR_OPT_RREQ:
        if (dp->flags & PKT_PROMISC_RECV)
            break;

        action |= dsr_rreq_opt_recv(dp, (struct dsr_rreq_opt *)dopt);
        break;
    case DSR_OPT_RREP:
        if (dp->flags & PKT_PROMISC_RECV)
            break;

        action |= dsr_rrep_opt_recv(dp, (struct dsr_rrep_opt *)dopt);
        break;
    case DSR_OPT_RERR:
        if (dp->flags & PKT_PROMISC_RECV)
            break;
        if (dp->num_rerr_opts < MAX_RERR_OPTS) {
            action |=
                dsr_rerr_opt_recv(dp, (struct dsr_rerr_opt *)dopt);
        }

        break;
    case DSR_OPT_PREV_HOP:
        break;
    case DSR_OPT_ACK:
        if (dp->flags & PKT_PROMISC_RECV)
            break;
```

判断 action

```
            if (dp->flags & PKT_PROMISC_RECV)
                break;

            if (dp->num_ack_opts < MAX_ACK_OPTS) {
                dp->ack_opt[dp->num_ack_opts++] =
                    (struct dsr_ack_opt *)dopt;
                action |=
                    dsr_ack_opt_recv((struct dsr_ack_opt *)
                             dopt);
            }
            break;
        case DSR_OPT_SRT:
            action |= dsr_srt_opt_recv(dp, (struct dsr_srt_opt *)dopt);
            break;
        case DSR_OPT_TIMEOUT:
            break;
        case DSR_OPT_FLOWID:
            break;
        case DSR_OPT_ACK_REQ:
            action |=
                dsr_ack_req_opt_recv(dp, (struct dsr_ack_req_opt *)
                         dopt);
            break;
        case DSR_OPT_PAD1:
            l++;
            dopt++;
            continue;
        default:
            DEBUG("Unknown DSR option type=%d\n", dopt->type);
        } « end switch dopt->type »
        l += dopt->length + 2;
        dopt = DSR_GET_NEXT_OPT(dopt);
    } « end while l<dsr_len&&(dsr_len-l... »
    return action;
} « end dsr_opt_recv »
```

中间的 while 循环应该是用来读取包内的下一个 opt 结束时减去一个 opt 的大小，进入下一个解析

Dsr-pkt.c

Dsr_pkt_alloc_opts 和 dsr_pkt_alloc_opts_expand

添加选项和扩展选项（增加选项）

Dsr_pkt_free_opts

释放/删除选项

Dsr_pkt_alloc （重载）

创建 pkt 填充字段

Dsr_pkt_free

释放 pkt