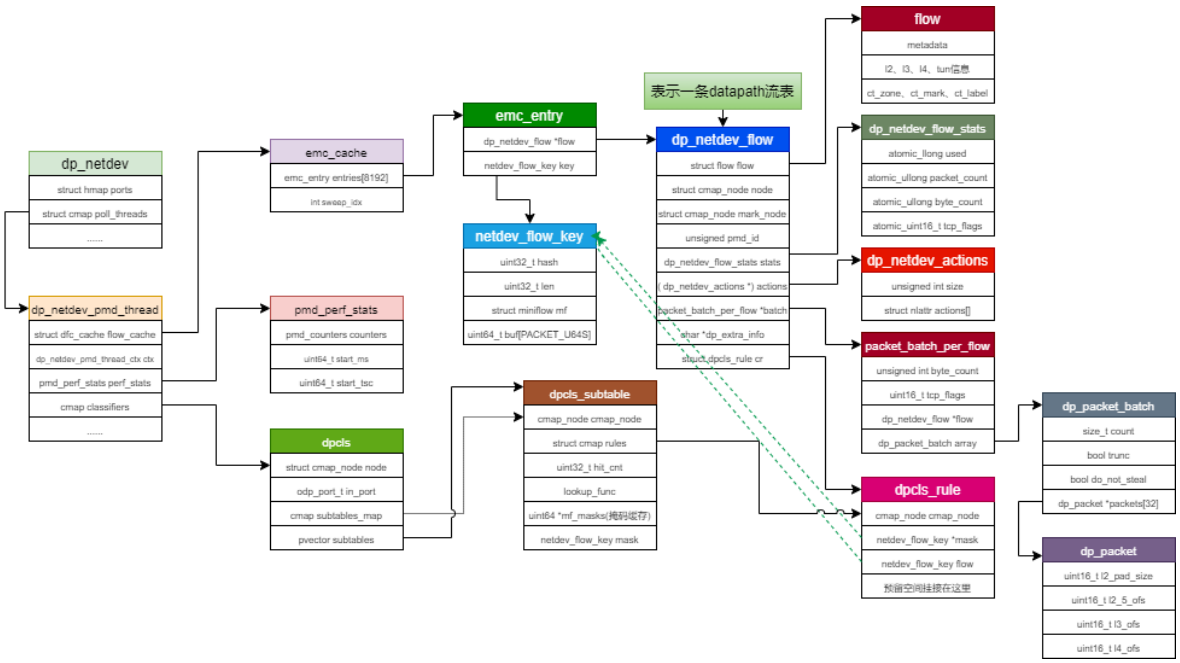
OVS收发包和流表卸载源码学习

流表模块数据结构与概要流程

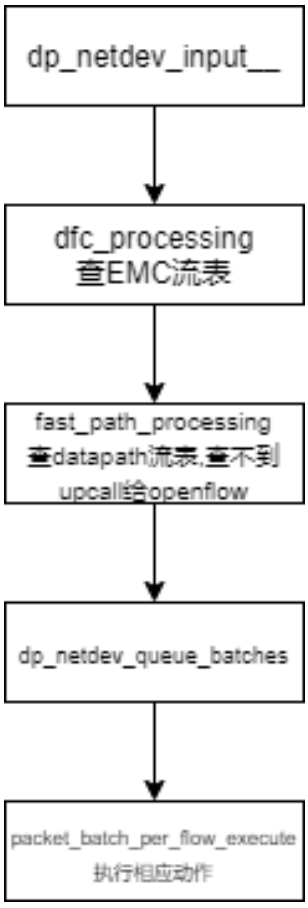
数据结构

看源码之前我们先学习下OVS的核心结构体之间的关系：



概要流程

其主要流程如下图所示：



再明确一个结论：EMC是以PMD线程为边界的，每个PMD线程拥有自己的EMC；dpcls是以端口为边界的，每个端口拥有自己的dpcls；ofproto classifier是以桥为边界的，每个桥拥有自己的ofproto classifier。（后文代码中均会有说明）

收包模块

收包模块概要流程

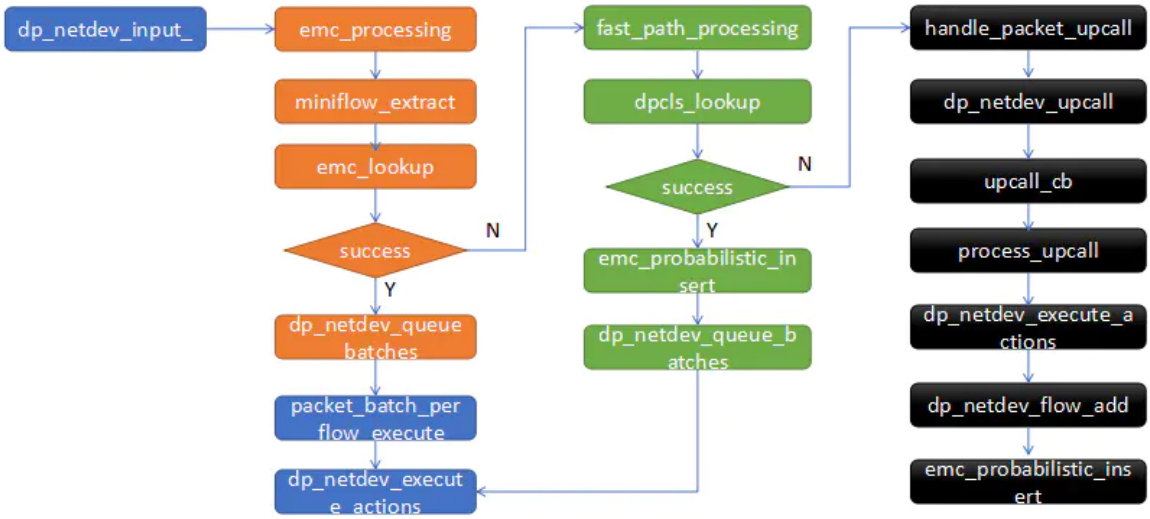
```
用户态Datapath处理packet流程：
        pmd_thread_main           dpif_netdev_run
                  \                     /
                    \                 /
                dp_netdev_process_rxq_port
                           |
                      dp_netdev_input
                           |
                     fast_path_processing
                           |
                    handle_packet_upcall
                       /             \
                 (2) /                 \(1)
                   /                     \
  dp_netdev_execute_actions       dp_netdev_upcall
                  |                          |
         odp_execute_actions             upcall_cb(调用注册的回调函数)
                  |
            dp_execute_cb
                  |
          dp_execute_output_action
                  |
    dp_netdev_pmd_flush_output_on_port
                  |
             netdev_send


内核态Datapath处理packet流程：
             netdev_frame_hook
                     |
             netdev_port_receive
                     |
             ovs_vport_receive
                     |
           ovs_dp_process_packet
                /             \
 (Normal Match)/               \ (Non-normal Match)
             /                   \
   ovs_execute_actions         ovs_dp_upcall
                                   |
                            queue_userspace_packet
                                   |
                           genlmsg_unicast
```

用户态收包模块详细流程

```
//主线程从非pmd类型端口收包
dpif_netdev_run
//pmd线程中从pmd类型端口收包
pmd_thread_main  //会调用dfc_cache_init，这也就是为什么emc是以pmd线程为边界的原因
  dp_netdev_process_rxq_port ->dp_netdev_input -> dp_netdev_input__
```

收到包之后主要做三件事：`emc_processing`,查emc流表；`fast_path_processing`查datapath流表和`handle_packet_upcall`查openflow流表已经后续的执行action。



flow到miniflow

flow和miniflow结构体介绍

介绍两个核心结构体：

```
//8字节对齐，总大小672个字节，负责存储流的所有信息。
struct flow {
    /* Metadata */
    struct flow_tnl tunnel;     /* Encapsulating tunnel parameters. */
    ovs_be64 metadata;          /* OpenFlow Metadata. */
    uint32_t regs[FLOW_N_REGS]; /* Registers. */
    uint32_t skb_priority;      /* Packet priority for QoS. */
    uint32_t pkt_mark;          /* Packet mark. */
    uint32_t dp_hash;           /* Datapath computed hash value. The exact
                                 * computation is opaque to the user space.
*/
    union flow_in_port in_port; /* Input port.*/
    uint32_t recirc_id;         /* Must be exact match. */
    uint8_t ct_state;           /* Connection tracking state. */
    uint8_t ct_nw_proto;        /* CT orig tuple IP protocol. */
    uint16_t ct_zone;           /* Connection tracking zone. */
    uint32_t ct_mark;           /* Connection mark.*/
    ovs_be32 packet_type;       /* OpenFlow packet type. */
    ovs_u128 ct_label;          /* Connection label. */
    uint32_t conj_id;           /* Conjunction ID. */
    ofp_port_t actset_output;   /* Output port in action set. */

    /* L2, Order the same as in the Ethernet header! (64-bit aligned) */
    struct eth_addr dl_dst;     /* Ethernet destination address. */
    struct eth_addr dl_src;     /* Ethernet source address. */
    ovs_be16 dl_type;           /* Ethernet frame type.
                                   Note: This also holds the Ethertype for
L3
                                   packets of type PACKET_TYPE(1,
Ethertype) */
    uint8_t pad1[2];            /* Pad to 64 bits. */
    union flow_vlan_hdr vlans[FLOW_MAX_VLAN_HEADERS]; /* VLANs */
    ovs_be32 mpls_lse[ROUND_UP(FLOW_MAX_MPLS_LABELS, 2)]; /* MPLS label
stack
```

```
                                                        (with
padding). */
    /* L3 (64-bit aligned) */
    ovs_be32 nw_src;            /* IPv4 source address or ARP SPA. */
    ovs_be32 nw_dst;            /* IPv4 destination address or ARP TPA. */
    ovs_be32 ct_nw_src;         /* CT orig tuple IPv4 source address. */
    ovs_be32 ct_nw_dst;         /* CT orig tuple IPv4 destination address.
*/
    struct in6_addr ipv6_src;   /* IPv6 source address. */
    struct in6_addr ipv6_dst;   /* IPv6 destination address. */
    struct in6_addr ct_ipv6_src; /* CT orig tuple IPv6 source address. */
    struct in6_addr ct_ipv6_dst; /* CT orig tuple IPv6 destination address.
*/
    ovs_be32 ipv6_label;        /* IPv6 flow label. */
    uint8_t nw_frag;            /* FLOW_FRAG_* flags. */
    uint8_t nw_tos;             /* IP ToS (including DSCP and ECN). */
    uint8_t nw_ttl;             /* IP TTL/Hop Limit. */
    uint8_t nw_proto;           /* IP protocol or low 8 bits of ARP opcode.
*/
    struct in6_addr nd_target;  /* IPv6 neighbor discovery (ND) target. */
    struct eth_addr arp_sha;    /* ARP/ND source hardware address. */
    struct eth_addr arp_tha;    /* ARP/ND target hardware address. */
    ovs_be16 tcp_flags;         /* TCP flags/ICMPv6 ND options type.
                                 * With L3 to avoid matching L4. */
    ovs_be16 pad2;              /* Pad to 64 bits. */
    struct ovs_key_nsh nsh;     /* Network Service Header keys */

    /* L4 (64-bit aligned) */
    ovs_be16 tp_src;            /* TCP/UDP/SCTP source port/ICMP type. */
    ovs_be16 tp_dst;            /* TCP/UDP/SCTP destination port/ICMP code.
*/
    ovs_be16 ct_tp_src;         /* CT original tuple source port/ICMP type.
*/
    ovs_be16 ct_tp_dst;         /* CT original tuple dst port/ICMP code. */
    ovs_be32 igmp_group_ip4;    /* IGMP group IPv4 address/ICMPv6 ND
reserved
                                 * field.
                                 * Keep last for BUILD_ASSERT_DECL below.
*/
    ovs_be32 pad3;              /* Pad to 64 bits. */
};
```

下面介绍下miniflow，miniflow可以理解成压缩版的flow：

```
struct miniflow {
    struct flowmap map;
    /* flowmap用来记录和flow的对应关系，flowmap中的每一个bit对应struct flow中的一个
uint64_t字段，如果bit为1，则flow中对应的uint64_t字段为非0值，如果bit为0，则flow中对应
的uint64_t字段为0；另一部分是flowmap后面的内存，由调用者根据flowmap中bit为1的个数*8字
节申请内存，用来保存flow中非0的uint64_t。*/

    /* 柔性数组后面接上:
     *     uint64_t values[n];
     * where 'n' is miniflow_n_values(miniflow). */
};
```

flow压缩流程

miniflow_extract用来从报文中提取flow信息，并保存到miniflow中。

```
/* Caller is responsible for initializing 'dst' with enough storage for
 * FLOW_U64S * 8 bytes. */
void
miniflow_extract(struct dp_packet *packet, struct miniflow *dst)
{
    const struct pkt_metadata *md = &packet->md;
    const void *data = dp_packet_data(packet);
    size_t size = dp_packet_size(packet);
    ovs_be32 packet_type = packet->packet_type;
    uint64_t *values = miniflow_values(dst);
    struct mf_ctx mf = { FLOWMAP_EMPTY_INITIALIZER, values,
                         values + FLOW_U64S };
    const char *frame;
    ovs_be16 dl_type = OVS_BE16_MAX;
    uint8_t nw_frag, nw_tos, nw_ttl, nw_proto;
    uint8_t *ct_nw_proto_p = NULL;
    ovs_be16 ct_tp_src = 0, ct_tp_dst = 0;
    ...
    ...
    //保存metadata信息到miniflow
    if (md->skb_priority || md->pkt_mark) {
        miniflow_push_uint32(mf, skb_priority, md->skb_priority);
        miniflow_push_uint32(mf, pkt_mark, md->pkt_mark);
    }
    //保存md->dp_hash到miniflow
    miniflow_push_uint32(mf, dp_hash, md->dp_hash);
    //保存报文入端口到miniflow
    miniflow_push_uint32(mf, in_port, odp_to_u32(md->in_port.odp_port));
    ...
    ...
    //保存二层信息到miniflow
    /* Link layer. */
    ASSERT_SEQUENTIAL(dl_dst, dl_src);
    miniflow_push_macs(mf, dl_dst, data);

    /* VLAN */
    union flow_vlan_hdr vlans[FLOW_MAX_VLAN_HEADERS];
    size_t num_vlans = parse_vlan(&data, &size, vlans);

    dl_type = parse_ethertype(&data, &size);
    miniflow_push_be16(mf, dl_type, dl_type);
    miniflow_pad_to_64(mf, dl_type);
    if (num_vlans > 0) {
        miniflow_push_words_32(mf, vlans, vlans, num_vlans);
    }
    ...
    ...
    //保存三层信息到miniflow
    /* Push both source and destination address at once. */
    miniflow_push_words(mf, nw_src, &nh->ip_src, 1);
    ...
    ...
    //保存四层信息到miniflow
```

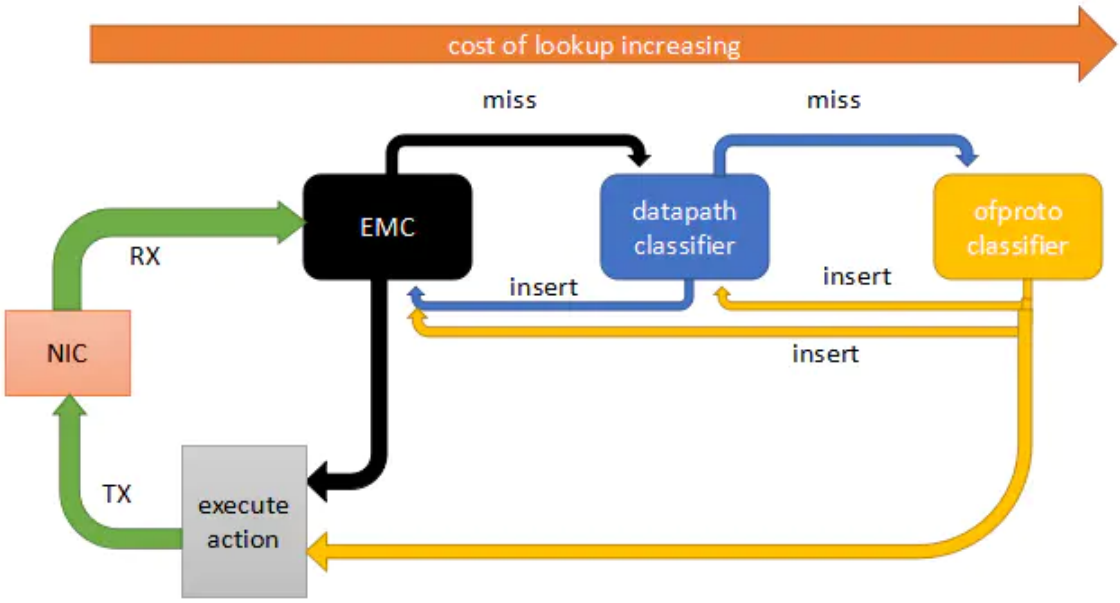
```
    if (OVS_LIKELY(nw_proto == IPPROTO_TCP)) {
        if (OVS_LIKELY(size >= TCP_HEADER_LEN)) {
            const struct tcp_header *tcp = data;

            miniflow_push_be32(mf, arp_tha.ea[2], 0);
            miniflow_push_be32(mf, tcp_flags,
                               TCP_FLAGS_BE32(tcp->tcp_ctl));
            miniflow_push_be16(mf, tp_src, tcp->tcp_src);
            miniflow_push_be16(mf, tp_dst, tcp->tcp_dst);
            miniflow_push_be16(mf, ct_tp_src, ct_tp_src);
            miniflow_push_be16(mf, ct_tp_dst, ct_tp_dst);
        }
    }
```

miniflow_expand用来将miniflow中的值恢复到flow结构体中,此处不做详细展开了。

三级流表

三级流表分别为EMC、datapath classifier、ofproto classifier。microflow在ovs+dpdk代码中，又被称为EMC(exact match cache)。megaflow在ovs+dpdk代码中，又被称为dpcls(datapath classifer)。从网卡接收到报文后，首先查找EMC表项，如果命中则直接执行action，如果miss则查找dpcls。如果查找dpcls命中，则将规则插入EMC，并且执行action。还是miss的话，就查找openflow流表。如果查找openflow命中，则将规则插入dpcls和EMC，并且执行action。还是miss的话，就丢包或者发给controller。



三级流表中所使用到的hash、TSS算法我会抽时间单独写篇文章来介绍。本文主要介绍大致查找流程。

emc流表查找流程

```
static inline size_t
emc_processing(struct dp_netdev_pmd_thread *pmd,
               struct dp_packet_batch *packets_,
               struct netdev_flow_key *keys,
               struct packet_batch_per_flow batches[], size_t *n_batches,
               bool md_is_valid, odp_port_t port_no)
{
    struct emc_cache *flow_cache = &pmd->flow_cache;
    struct netdev_flow_key *key = &keys[0];
    size_t n_missed = 0, n_dropped = 0;
```

```
    struct dp_packet *packet;
    const size_t size = dp_packet_batch_size(packets_);
    uint32_t cur_min;
    int i;
    //获取插入emc的可能性，如果为0，说明emc流表被关闭
    //此方案是为了解决EMC抖动问题，会在流表算法文章中单独阐述这一内容。
    //简而言之就是一个随机数获取函数 来实现概率性emc流表插入动作
    atomic_read_relaxed(&pmd->dp->emc_insert_min, &cur_min);

    //遍历报文
    DP_PACKET_BATCH_REFILL_FOR_EACH (i, size, packet, packets_) {
        struct dp_netdev_flow *flow;
        //报文长度太小，丢包
        if (OVS_UNLIKELY(dp_packet_size(packet) < ETH_HEADER_LEN)) {
            dp_packet_delete(packet);
            n_dropped++;
            continue;
        }

        if (i != size - 1) {
            struct dp_packet **packets = packets_->packets;
            /* 预取下一个包数据和metadata. */
            OVS_PREFETCH(dp_packet_data(packets[i+1]));
            pkt_metadata_prefetch_init(&packets[i+1]->md);
        }
        //初始化metadata
        if (!md_is_valid) {
            pkt_metadata_init(&packet->md, port_no);
        }
        //从报文提前flow信息，保存到key->mf，可以参考上面章节flow到miniflow
        miniflow_extract(packet, &key->mf);
        key->len = 0; /* Not computed yet. */
        //获取hash值，如果通过网卡的RSS收包则直接从mbuf获取hash值，否则需要软件计算
hash
        key->hash = dpif_netdev_packet_get_rss_hash(packet, &key->mf);
        //cur_min为0，说明EMC是关闭的，不用查询
        flow = (cur_min == 0) ? NULL: emc_lookup(flow_cache, key);
        if (OVS_LIKELY(flow)) {
            //查到了EMC表项，将报文插入flow的batch，方便后面批量处理
            dp_netdev_queue_batches(packet, flow, &key->mf, batches,
                                    n_batches);
        } else {
            /* Exact match cache missed. Group missed packets together at
             * the beginning of the 'packets' array. */
            //没查到emc，将报文重新插入packets_
            dp_packet_batch_refill(packets_, packet, i);
            /* 'key[n_missed]' contains the key of the current packet and
it
             * must be returned to the caller. The next key should be
extracted
             * to 'keys[n_missed + 1]'. */
            key = &keys[++n_missed];
        }
    }
    //增加命中emc的计数
    dp_netdev_count_packet(pmd, DP_STAT_EXACT_HIT,
                           size - n_dropped - n_missed);
    //返回未命中emc并且没有被丢弃的报文个数
```

```
    return dp_packet_batch_size(packets_);
}
```

datapath查找流程

下面再介绍下datapath的大致查找流程。

```
static inline void
fast_path_processing(struct dp_netdev_pmd_thread *pmd,
                     struct dp_packet_batch *packets_,
                     struct netdev_flow_key *keys,
                     struct packet_batch_per_flow batches[], size_t
*n_batches,
                     odp_port_t in_port,
                     long long now)
{
    int cnt = packets_->count;
#if !defined(__CHECKER__) && !defined(_WIN32)
    const size_t PKT_ARRAY_SIZE = cnt;
#else
    /* Sparse or MSVC doesn't like variable length array. */
    enum { PKT_ARRAY_SIZE = NETDEV_MAX_BURST };
#endif
    struct dp_packet **packets = packets_->packets;
    struct dpcls *cls;
    struct dpcls_rule *rules[PKT_ARRAY_SIZE];
    struct dp_netdev *dp = pmd->dp;
    int miss_cnt = 0, lost_cnt = 0;
    int lookup_cnt = 0, add_lookup_cnt;
    bool any_miss;
    size_t i;

    for (i = 0; i < cnt; i++) {
        /* Key length is needed in all the cases, hash computed on demand.
*/
        keys[i].len =
netdev_flow_key_size(miniflow_n_values(&keys[i].mf));
    }
    /* Get the classifier for the in_port */
    //根据入端口找到它的dpcls,这里也就是说为什么datapath是以端口为边界的
    cls = dp_netdev_pmd_lookup_dpcls(pmd, in_port);
    if (OVS_LIKELY(cls)) {
        //调用dpcls_lookup进行匹配
        any_miss = !dpcls_lookup(cls, keys, rules, cnt, &lookup_cnt);
    } else {
        any_miss = true;
        memset(rules, 0, sizeof(rules));
    }
    //如果有miss的，则需要进行openflow流表查询
    if (OVS_UNLIKELY(any_miss) && !fat_rwlock_tryrdlock(&dp-
>upcall_rwlock)) {
        uint64_t actions_stub[512 / 8], slow_stub[512 / 8];
        struct ofpbuf actions, put_actions;

        ofpbuf_use_stub(&actions, actions_stub, sizeof actions_stub);
        ofpbuf_use_stub(&put_actions, slow_stub, sizeof slow_stub);

```

```
        for (i = 0; i < cnt; i++) {
            struct dp_netdev_flow *netdev_flow;

            if (OVS_LIKELY(rules[i])) {
                continue;
            }

            /* It's possible that an earlier slow path execution installed
             * a rule covering this flow.  In this case, it's a lot
cheaper
             * to catch it here than execute a miss. */
            netdev_flow = dp_netdev_pmd_lookup_flow(pmd, &keys[i],
                                                    &add_lookup_cnt);
            if (netdev_flow) {
                lookup_cnt += add_lookup_cnt;
                rules[i] = &netdev_flow->cr;
                continue;
            }

            miss_cnt++;
            //开始查找openflow流表。
            如果查找openflow流表成功并需要下发到dpcls时，需要判断是否超出最大流表限
制
            handle_packet_upcall(pmd, packets[i], &keys[i], &actions,
                                 &put_actions, &lost_cnt, now);
        }

        ofpbuf_uninit(&actions);
        ofpbuf_uninit(&put_actions);
        fat_rwlock_unlock(&dp->upcall_rwlock);
    } else if (OVS_UNLIKELY(any_miss)) {
        for (i = 0; i < cnt; i++) {
            if (OVS_UNLIKELY(!rules[i])) {
                dp_packet_delete(packets[i]);
                lost_cnt++;
                miss_cnt++;
            }
        }
    }

    for (i = 0; i < cnt; i++) {
        struct dp_packet *packet = packets[i];
        struct dp_netdev_flow *flow;

        if (OVS_UNLIKELY(!rules[i])) {
            continue;
        }

        flow = dp_netdev_flow_cast(rules[i]);
        //查找dpcls成功的，需要将相关rule下发到emc表项
        emc_probabilistic_insert(pmd, &keys[i], flow);
        dp_netdev_queue_batches(packet, flow, &keys[i].mf, batches,
n_batches);
    }
    //统计信息
    dp_netdev_count_packet(pmd, DP_STAT_MASKED_HIT, cnt - miss_cnt);
    dp_netdev_count_packet(pmd, DP_STAT_LOOKUP_HIT, lookup_cnt);
    dp_netdev_count_packet(pmd, DP_STAT_MISS, miss_cnt);
```

```
    dp_netdev_count_packet(pmd, DP_STAT_LOST, lost_cnt);
}
```

openflow查找流程

如果EMC流表、datapath都查不到，则需要upcall调用查询openflow流表，内核情况下netlink实现，OVS+DPDK情况下，PMD线程中调用upcall_cb，通过回调函数调用的upcall_cb。

```
static int
upcall_cb(const struct dp_packet *packet, const struct flow *flow, ovs_u128
*ufid,
          unsigned pmd_id, enum dpif_upcall_type type,
          const struct nlattr *userdata, struct ofpbuf *actions,
          struct flow_wildcards *wc, struct ofpbuf *put_actions, void *aux)
{
    static struct vlog_rate_limit rl = VLOG_RATE_LIMIT_INIT(1, 1);
    struct udpif *udpif = aux;
    struct upcall upcall;
    bool megaflow;
    int error;
    //读取enable_megaflows，用来控制是否开启megaflow。
    //可以通过命令开启 "ovs-appctl upcall/enable-megaflows"
    atomic_read_relaxed(&enable_megaflows, &megaflow);

    error = upcall_receive(&upcall, udpif->backer, packet, type, userdata,
                           flow, 0, ufid, pmd_id);
    if (error) {
        return error;
    }
    //查找openflow流表，又是很长的函数调用
    //后面单独讲解
    error = process_upcall(udpif, &upcall, actions, wc);
    if (error) {
        goto out;
    }

    if (upcall.xout.slow && put_actions) {
        ofpbuf_put(put_actions, upcall.put_actions.data,
                   upcall.put_actions.size);
    }

    //如果关闭了megaflow，则将flow信息转换到wc，即megaflow也将变成精确匹配。
    //如果开启megaflow，则wc查找openflow流表的通配符集合，小于等于flow信息。
    if (OVS_UNLIKELY(!megaflow && wc)) {
        flow_wildcards_init_for_packet(wc, flow);
    }

    //如果超过了最大流表限制，则返回ENOSPC
    if (!should_install_flow(udpif, &upcall)) {
        error = ENOSPC;
        goto out;
    }

    if (upcall.ukey && !ukey_install(udpif, upcall.ukey)) {
        VLOG_WARN_RL(&rl, "upcall_cb failure: ukey installation fails");
        error = ENOSPC;
    }
```

```
out:
    if (!error) {
        upcall.ukey_persists = true;
    }
    upcall_uninit(&upcall);
    return error;
}
```

这里单独介绍下process_upcall的主要流程：

```
//主要函数调用流程如下：
process_upcall -> upcall_xlate -> xlate_actions ->
rule_dpif_lookup_from_table
//xbridge_lookup函数中就可以看出ofproto是基于bridge为边界的。
//xlate_actions函数就是先找到对应的bridge再去查流表

    miss_config = OFPUTIL_TABLE_MISS_CONTINUE;
    //从指定的table_id开始查找
    for (next_id = *table_id;
         next_id < ofproto->up.n_tables;
         next_id++, next_id += (next_id == TBL_INTERNAL))
    {
        *table_id = next_id;
        //到table的分类器中查找流表，并设置wc
        rule = rule_dpif_lookup_in_table(ofproto, version, next_id, flow,
wc);
        //如果查找到了，则返回
        if (rule) {
            goto out;   /* Match. */
        }
        //如果没有查找，则根据配置选择继续查找下一个table还是结束查找
        if (honor_table_miss) {
            miss_config = ofproto_table_get_miss_config(&ofproto->up,
                                                        *table_id);
            if (miss_config == OFPUTIL_TABLE_MISS_CONTINUE) {
                continue;
            }
        }
        break;
    }
```

流表模块卸载流程

```
流表卸载到驱动流程：
                            dpif_operate
           (dpif-netdev) /              \(dpif-netlink)
                        /                \
           dpif_netdev_operate         dpif_netlink_operate
                     | 用户态                        |内核态
           dpif_netdev_flow_put           try_send_to_netdev

                     |                              |
                     |                          parse_flow_put
          queue_netdev_flow_put                     |
                     |（回调）                  netdev_flow_put
         netdev_offload_dpdk_flow_put               |
```

```
                    |                          netdev_tc_flow_put
        netdev_offload_dpdk_flow_create
                    |
        netdev_offload_dpdk_mark_rss
                    |
        netdev_offload_dpdk_flow_create
                    |
        netdev_dpdk_rte_flow_create
                    |
             rte_flow_create
                    |各厂商驱动回调
             sfc_flow_create

内核态：
当报文不匹配的时候，会将报文上报，会调用udpif_upcall_handler
udpif_upcall_handler-->recv_upcalls-->handle_upcalls-->dpif_operate-->dpif_netlink_operate-->try_send_to_netdev-->parse_flow_put-->netdev_flow_put-->netdev_tc_flow_put

用户态：
fast_path_processing->handle_packet_upcall->dp_netdev_flow_add->queue_netdev_flow_put->dp_netdev_flow_offload_main->dp_netdev_flow_offload_put--callback-->netdev_offload_dpdk_flow_put

内核态卸载是由handler线程处理，而用户态OVS+DPDK是由dp_netdev_flow_offload线程处理，此时的handler线程处于阻塞状态。
```

OVS Conntrack和NAT

基础须知

对于kernel datapath来说，使用kernel的conntrack来实现，对于userspace datapath来说，ovs本身来实现，可参考 lib/conntrack.c 文件。
ct_state的可能值如下：

```
new 通过ct action指定报文经过conntrack模块处理，不一定有commit。通常是数据流的第一个数据包
est 表示conntrack模块看到了报文双向数据流，一定是在commit 的conntrack后
rel 表示和已经存在的conntrack相关，比如icmp不可达消息或者ftp的数据流
rpl 表示反方向的报文
inv 无效的，表示conntrack模块没有正确识别到报文，比如L3/L4 protocol handler没有加载，或者L3/L4 protocol handler认为报文错误
trk 表示报文经过了conntrack模块处理，如果这个flag不设置，其他flag都不能被设置
snat 表示报文经过了snat，源ip或者port
dnat 表示报文经过了dnat，目的ip或者port
```

这些flag得结合"+"或者"-"来使用，"+"表示必须匹配，"-"表示必须不匹配。可以同时指定多个flag，比如 ct_state=+trk+new。

key

匹配域和flow 中下的以下字段对应，用来匹配流表：

```
struct flow {
    ...
```

```
    uint8_t ct_state;           /* Connection tracking state. */
    uint8_t ct_nw_proto;        /* CT orig tuple IP protocol. */
    uint16_t ct_zone;           /* Connection tracking zone. */
    uint32_t ct_mark;           /* Connection mark.*/
    ovs_be32 ct_nw_src;         /* CT orig tuple IPv4 source address. */
    ovs_be32 ct_nw_dst;         /* CT orig tuple IPv4 destination address.
*/
    struct in6_addr ct_ipv6_src; /* CT orig tuple IPv6 source address. */
    struct in6_addr ct_ipv6_dst; /* CT orig tuple IPv6 destination address.
*/
    ovs_be16 ct_tp_src;         /* CT original tuple source port/ICMP type.
*/
    ovs_be16 ct_tp_dst;         /* CT original tuple dst port/ICMP code. */
    ...
}
```

action

ovs通过ct action实现conntrack,ct会将报文送到conntrack模块进行处理,发送格式 `ct([argument][,argument…])`
支持参数如下:

```
commit //只有执行了commit，才会在conntrack模块创建conntrack表项
force //强制删除已存在的conntrack表项
table //跳转到指定的table执行
zone //设置zone，隔离conntrack
exec //执行其他action，目前只支持设置ct_mark和ct_label，比如exec(set_field: 1->ct_mark)
    //alg=<ftp/tftp> 指定alg类型，目前只支持ftp和tftp
nat //指定ip和port
```

命令行配置NAT

```
#添加nat表项
ovs-ofctl add-flow br0 "table=0, priority=50, ct_state=-trk, tcp,
in_port=veth_l0, actions=ct(commit,nat(src=10.1.1.240-10.2.2.2:2222-3333))"

//在一个ct里指定多次nat，只有最后一个nat生效，可参考do_xlate_actions中，ctx->ct_nat_action = ofpact_get_NAT(a)只有一个ctx->ct_nat_action
ovs-ofctl add-flow br0 "table=0, priority=50, ct_state=-trk, tcp,
actions=ct(commit,nat(src=10.1.1.240-10.2.2.2:2222-3333),
nat(dst=10.1.1.240-10.2.2.2:2222-3333)), veth_r0"

//可以通过指定多个ct，实现fullnat，即同时转换源目的ip。
//但是这两个ct必须指定不同的zone，否则只有第一个ct生效。因为在 handle_nat 中，判断只有zone不一样才会进行后续的nat操作
//错误方式，指定了src和dst nat，但是zone相同，只有前面的snat生效
ovs-ofctl add-flow br0 "table=0, priority=50, ct_state=-trk, tcp,
actions=ct(commit,nat(src=10.1.1.240-10.2.2.2:2222-3333)),
ct(commit,nat(dst=10.1.1.240-10.2.2.2:2222-3333)), veth_r0"

//正确方式，使用不同zone，指定fullnat
ovs-ofctl add-flow br0 "table=0, priority=50, ct_state=-trk, tcp,
actions=ct(commit,zone=100, nat(src=10.1.1.240-10.2.2.2:2222-3333)),
ct(commit, zone=200, nat(dst=10.1.1.240-10.2.2.2:2222-3333)), veth_r0"
```

源码分析

数据结构

主要存储CT信息的两个结构体如下：

```
/* OFPACT_CT.
 *
 * Used for NXAST_CT. */
struct ofpact_conntrack {
    OFPACT_PADDED_MEMBERS(
        struct ofpact ofpact; /*eg:{ofpact = {type = OFPACT_CT, raw = 255
    '\377', len为带上柔性数组后面内容的总长度}  */
        uint16_t flags;   //NX_CT_F_COMMIT和NX_CT_F_FORCE
        uint16_t zone_imm;  //zone
        struct mf_subfield zone_src;
        uint16_t alg;  //算法类型
        uint8_t recirc_table;//跳转到指定table
    );
    struct ofpact actions[0];
    /* struct ofpact_conntrack用来保存ct后面的参数，并使用另一个结构体struct
ofpact_nat专门保存ct的nat信息。 */
};

struct ofpact_nat {
    OFPACT_PADDED_MEMBERS(
        struct ofpact ofpact; //类型为 OFPACT_NAT
        uint8_t range_af; /* AF_UNSPEC, AF_INET, or AF_INET6 */
        uint16_t flags;  /* NX_NAT_F_* */
        struct {
            struct {
                uint16_t min;
                uint16_t max;
            } proto;
            union {
                struct {
                    ovs_be32 min;
                    ovs_be32 max;
                } ipv4;
                struct {
                    struct in6_addr min;
                    struct in6_addr max;
                } ipv6;
            } addr;
        } range;
    );
};
```

源码举例-命令行配置

下面结合具体函数来进行代码流程分析,业务场景为命令行配置：

```
static char*
parse_CT(char *arg, const struct ofpact_parse_params *pp)
{
    const size_t ct_offset = ofpacts_pull(pp->ofpacts);
```

```
    struct ofpact_conntrack *oc;
    char *error = NULL;
    char *key, *value;

    //ofpact_put_CT可以参考
https://cloud.tencent.com/developer/article/1082684，代码抽象的一种用法
    //设置 OFPACT_CT
    oc = ofpact_put_CT(pp->ofpacts);
    oc->flags = 0;
    oc->recirc_table = NX_CT_RECIRC_NONE;
    while (ofputil_parse_key_value(&arg, &key, &value)) {
        if (!strcmp(key, "commit")) {
            oc->flags |= NX_CT_F_COMMIT;
        } else if (!strcmp(key, "force")) {
            oc->flags |= NX_CT_F_FORCE;
        } else if (!strcmp(key, "table")) {
            if (!ofputil_table_from_string(value, pp->table_map,
                                           &oc->recirc_table)) {
                error = xasprintf("unknown table %s", value);
            } else if (oc->recirc_table == NX_CT_RECIRC_NONE) {
                error = xasprintf("invalid table %#"PRIx8, oc-
>recirc_table);
            }
        } else if (!strcmp(key, "zone")) {
            error = str_to_u16(value, "zone", &oc->zone_imm);

            if (error) {
                free(error);
                error = mf_parse_subfield(&oc->zone_src, value);
                if (error) {
                    return error;
                }
            }
        } else if (!strcmp(key, "alg")) {
            error = str_to_connhelper(value, &oc->alg);
        } else if (!strcmp(key, "nat")) {
            const size_t nat_offset = ofpacts_pull(pp->ofpacts);
            //解析NAT信息
            error = parse_NAT(value, pp);
            /* Update CT action pointer and length. */
            pp->ofpacts->header = ofpbuf_push_uninit(pp->ofpacts,
nat_offset);
            oc = pp->ofpacts->header;
        } else if (!strcmp(key, "exec")) {
            /* Hide existing actions from ofpacts_parse_copy(), so the
             * nesting can be handled transparently. */
            enum ofputil_protocol usable_protocols2;
            const size_t exec_offset = ofpacts_pull(pp->ofpacts);

            /* Initializes 'usable_protocol2', fold it back to
             * '*usable_protocols' afterwards, so that we do not lose
             * restrictions already in there. */
            struct ofpact_parse_params pp2 = *pp;
            pp2.usable_protocols = &usable_protocols2;
            /* 解析 exec 参数，比如 set_field ct(commit,exec(set_field:1-
>ct_mark)) (->后面的ct_mark为key，前面的1为value) */
            error = ofpacts_parse_copy(value, &pp2, false, OFPACT_CT);
            *pp->usable_protocols &= usable_protocols2;
```

```
            pp->ofpacts->header = ofpbuf_push_uninit(pp->ofpacts,
exec_offset);
            oc = pp->ofpacts->header;
        } else {
            error = xasprintf("invalid argument to \"ct\" action: `%s'",
key);
        }
        if (error) {
            break;
        }
    }
    if (!error && oc->flags & NX_CT_F_FORCE && !(oc->flags &
NX_CT_F_COMMIT)) {
        error = xasprintf("\"force\" flag requires \"commit\" flag.");
    }

    if (ofpbuf_oversized(pp->ofpacts)) {
        free(error);
        return xasprintf("input too big");
    }
    //更新 struct ofpact_conntrack->ofpact.len，包含nat的长度
    ofpact_finish_CT(pp->ofpacts, &oc);
    ofpbuf_push_uninit(pp->ofpacts, ct_offset);
    return error;
}

```

该流程结束后大概内存格式如下：`struct ofpact_conntrack(OFPACT_CT) + struct ofpact_nat(OFPACT_NAT) + struct ofpact_set_field(OFPACT_SET_FIELD)`

源码举例-包触发

看到这里我们继续来分析下包触发CT流表的流程。ovs接收到报文后，查找fastpath失败，继续slowpath查找，如果匹配到的流表的action为ct，处理流程如下：

```
do_xlate_actions(const struct ofpact *ofpacts, size_t ofpacts_len,
                 struct xlate_ctx *ctx, bool is_last_action,
                 bool group_bucket_action)
{
    struct flow_wildcards *wc = ctx->wc;
    struct flow *flow = &ctx->xin->flow;
    const struct ofpact *a;

    OFPACT_FOR_EACH (a, ofpacts, ofpacts_len) {
        struct ofpact_controller *controller;
        const struct ofpact_metadata *metadata;
        const struct ofpact_set_field *set_field;
        const struct mf_field *mf;
        bool last = is_last_action && ofpact_last(a, ofpacts, ofpacts_len)
                    && ctx->action_set.size;

        //如果动作类型为CT，调用对应处理函数
        switch (a->type) {
            case OFPACT_CT:
                //ofpact_get_CT获取struct ofpact_conntrack及其后面嵌套的action
                compose_conntrack_action(ctx, ofpact_get_CT(a), last);
        }
```

```
    }
}
```

接下来是将struct ofpact_conntrack中的action信息转换到datapath能识别的aciton结构体。

```
static void
compose_conntrack_action(struct xlate_ctx *ctx, struct ofpact_conntrack
*ofc)
    //内部再次调用do_xlate_actions，解析nat和ct_mark,ct_label信息
    do_xlate_actions(ofc->actions, ofpact_ct_get_action_len(ofc), ctx);
        //获取nat信息，保存到 ctx->ct_nat_action，如果指定了多次nat，只有最后一次
会生效
        case OFPACT_NAT:
            /* This will be processed by compose_conntrack_action(). */
            ctx->ct_nat_action = ofpact_get_NAT(a);
            break;

        //解析 ct_mark 或者 ct_label 并保存到 flow->ct_mark和 flow->ct_label
        case OFPACT_SET_FIELD:
            set_field = ofpact_get_SET_FIELD(a);
            mf = set_field->field;

            /* Set the field only if the packet actually has it. */
            if (mf_are_prereqs_ok(mf, flow, wc)) {
                mf_mask_field_masked(mf, ofpact_set_field_mask(set_field),
wc);
                mf_set_flow_value_masked(mf, set_field->value,
                                         ofpact_set_field_mask(set_field),
                                         flow);
    if (ofc->zone_src.field) {
        zone = mf_get_subfield(&ofc->zone_src, &ctx->xin->flow);
    } else {
        zone = ofc->zone_imm;
    }

    //添加第一个 datapath action OVS_ACTION_ATTR_CT
    //OVS_ACTION_ATTR_CT 开始
    ct_offset = nl_msg_start_nested(ctx->odp_actions, OVS_ACTION_ATTR_CT);
    if (ofc->flags & NX_CT_F_COMMIT) {
        nl_msg_put_flag(ctx->odp_actions, ofc->flags & NX_CT_F_FORCE ?
OVS_CT_ATTR_FORCE_COMMIT : OVS_CT_ATTR_COMMIT);
        if (ctx->xbridge->support.ct_eventmask) {
            nl_msg_put_u32(ctx->odp_actions, OVS_CT_ATTR_EVENTMASK,
OVS_CT_EVENTMASK_DEFAULT);
        }
    }

    nl_msg_put_u16(ctx->odp_actions, OVS_CT_ATTR_ZONE, zone);
    put_ct_mark(&ctx->xin->flow, ctx->odp_actions, ctx->wc);
        if (wc->masks.ct_mark) {
            struct {
                uint32_t key;
                uint32_t mask;
            } *odp_ct_mark;

            odp_ct_mark = nl_msg_put_unspec_uninit(odp_actions,
OVS_CT_ATTR_MARK, sizeof(*odp_ct_mark));
```

```
47              odp_ct_mark->key = flow->ct_mark & wc->masks.ct_mark;
48              odp_ct_mark->mask = wc->masks.ct_mark;
49          }
50      put_ct_label(&ctx->xin->flow, ctx->odp_actions, ctx->wc);
51
52      put_ct_helper(ctx, ctx->odp_actions, ofc);
53          if (ofc->alg) {
54              switch(ofc->alg) {
55              case IPPORT_FTP:
56                  nl_msg_put_string(odp_actions, OVS_CT_ATTR_HELPER, "ftp");
57                  break;
58              case IPPORT_TFTP:
59                  nl_msg_put_string(odp_actions, OVS_CT_ATTR_HELPER,
"tftp");
60                  break;
61              default:
62                  xlate_report_error(ctx, "cannot serialize ct_helper %d",
ofc->alg);
63                  break;
64              }
65          }
66
67      put_ct_nat(ctx);
68          struct ofpact_nat *ofn = ctx->ct_nat_action;
69          nat_offset = nl_msg_start_nested(ctx->odp_actions,
OVS_CT_ATTR_NAT);
70          if (ofn->flags & NX_NAT_F_SRC || ofn->flags & NX_NAT_F_DST) {
71              nl_msg_put_flag(ctx->odp_actions, ofn->flags & NX_NAT_F_SRC
72                              ? OVS_NAT_ATTR_SRC : OVS_NAT_ATTR_DST);
73              if (ofn->flags & NX_NAT_F_PERSISTENT) {
74                  nl_msg_put_flag(ctx->odp_actions,
OVS_NAT_ATTR_PERSISTENT);
75              }
76              if (ofn->flags & NX_NAT_F_PROTO_HASH) {
77                  nl_msg_put_flag(ctx->odp_actions,
OVS_NAT_ATTR_PROTO_HASH);
78              } else if (ofn->flags & NX_NAT_F_PROTO_RANDOM) {
79                  nl_msg_put_flag(ctx->odp_actions,
OVS_NAT_ATTR_PROTO_RANDOM);
80              }
81              ...
82          }
83          nl_msg_end_nested(ctx->odp_actions, nat_offset);
84
85      ctx->ct_nat_action = NULL;
86      //OVS_ACTION_ATTR_CT 结束
87      nl_msg_end_nested(ctx->odp_actions, ct_offset);
88          }
89      }
90
91
92      //如果配置 ct(table=x) 则需要添加第二个 datapath action
OVS_ACTION_ATTR_RECIRC
93      //recirc_table 值为table id，表示需要转到其他table继续执行，比如
actions=ct(table=0)
94      //值为 NX_CT_RECIRC_NONE，说明不需要
95      //节点最后存储在ctx->xin->recirc_queue中
96      if (ofc->recirc_table == NX_CT_RECIRC_NONE) {
```

```
97          ctx->conntracked = true;
98          compose_recirculate_and_fork(ctx, ofc->recirc_table);
99              uint32_t recirc_id;
100             ctx->freezing = true;
101             recirc_id = finish_freezing__(ctx, table);
102                 struct frozen_state state = {
103                     //保存需要跳转到的 table id，即 recirc_table
104                     .table_id = table,
105                     .ofproto_uuid = ctx->xbridge->ofproto->uuid,
106                     .stack = ctx->stack.data,
107                     .stack_size = ctx->stack.size,
108                     .mirrors = ctx->mirrors,
109                     .conntracked = ctx->conntracked,
110                     .xport_uuid = ctx->xin->xport_uuid,
111                     .ofpacts = ctx->frozen_actions.data,
112                     .ofpacts_len = ctx->frozen_actions.size,
113                     .action_set = ctx->action_set.data,
114                     .action_set_len = ctx->action_set.size,
115                 };
116                 frozen_metadata_from_flow(&state.metadata, &ctx->xin-
>flow);
117
118                 //获取 recirc_id，保存到 odp_actions，作为datapath的其中一个
action
119                 id = recirc_alloc_id_ctx(&state);
120                     uint32_t hash = frozen_state_hash(state);
121                     struct recirc_id_node *node = recirc_ref_equal(state,
hash);
122                     node = recirc_alloc_id__(state, hash);
123                         struct recirc_id_node *node = xzalloc(sizeof
*node);
124                         node->hash = hash;
125                         ovs_refcount_init(&node->refcount);
126                         frozen_state_clone(CONST_CAST(struct frozen_state
*, &node->state), state);
127                         cmap_insert(&id_map, &node->id_node, node->id);
128                         cmap_insert(&metadata_map, &node->metadata_node,
node->hash);
129                         return node;
130                     node->id;
131                 nl_msg_put_u32(ctx->odp_actions, OVS_ACTION_ATTR_RECIRC,
id);
132         ctx->conntracked = false;
133     }
```

综上可知openflow中的action 可能包含两种action: OVS_ACTION_ATTR_CT和OVS_ACTION_ATTR_RECIRC，前者又包含了commit(OVS_CT_ATTR_FORCE_COMMIT), ct_mark(OVS_CT_ATTR_MARK), ct_label和nat(OVS_CT_ATTR_NAT)等信息，后者仅仅包含了recirc_id，用来重新注入datapath后查看到table id，即用来跳转到指定table执行。将action添加到datapath后，对当前报文执行action流程如下：

```
1   static inline void
2   packet_batch_per_flow_execute(struct packet_batch_per_flow *batch,
3                                 struct dp_netdev_pmd_thread *pmd)
4       {
5           struct dp_netdev_actions *actions;
6           struct dp_netdev_flow *flow = batch->flow;
```

```
        //获取action
        actions = dp_netdev_flow_get_actions(flow);

        //执行对应action
        dp_netdev_execute_actions(pmd, &batch->array, true, &flow->flow,
actions->actions, actions->size, now);
    }

    //odp_execute_actions函数是具体执行流程函数
        //遍历执行匹配流表的所有 actions
        NL_ATTR_FOR_EACH_UNSAFE (a, left, actions, actions_len)
            int type = nl_attr_type(a);
            //调用dp_execute_cb，如果是最后一个batch或者最后一个action 则执行完直
接返回
            //具体操作见下述流程
            dp_execute_action(dp, batch, a, may_steal);
                //执行 ct action
                case OVS_ACTION_ATTR_CT: {
                    NL_ATTR_FOR_EACH_UNSAFE (b, left, nl_attr_get(a),
nl_attr_get_size(a)) {
                        enum ovs_ct_attr sub_type = nl_attr_type(b);
                        switch(sub_type) {
                        case OVS_CT_ATTR_FORCE_COMMIT:
                            force = true;
                            /* fall through. */
                        case OVS_CT_ATTR_COMMIT:
                            commit = true;
                            break;
                        case OVS_CT_ATTR_ZONE:
                            zone = nl_attr_get_u16(b);
                            break;
                        case OVS_CT_ATTR_HELPER:
                            helper = nl_attr_get_string(b);
                            break;
                        case OVS_CT_ATTR_MARK:
                            setmark = nl_attr_get(b);
                            break;
                        case OVS_CT_ATTR_LABELS:
                            setlabel = nl_attr_get(b);
                            break;
                        case OVS_CT_ATTR_EVENTMASK:
                            /* Silently ignored, as userspace datapath
does not generate
                                * netlink events. */
                            break;
                        case OVS_CT_ATTR_NAT: {
                            const struct nlattr *b_nest;
                            unsigned int left_nest;
                            bool ip_min_specified = false;
                            bool proto_num_min_specified = false;
                            bool ip_max_specified = false;
                            bool proto_num_max_specified = false;
                            memset(&nat_action_info, 0, sizeof
nat_action_info);
                            nat_action_info_ref = &nat_action_info;

                            NL_NESTED_FOR_EACH_UNSAFE (b_nest, left_nest,
b) {
```

```
                                enum ovs_nat_attr sub_type_nest =
nl_attr_type(b_nest);

                                switch (sub_type_nest) {
                                case OVS_NAT_ATTR_SRC:
                                case OVS_NAT_ATTR_DST:
                                    nat_config = true;
                                    nat_action_info.nat_action |=
                                        ((sub_type_nest ==
OVS_NAT_ATTR_SRC)
                                            ? NAT_ACTION_SRC :
NAT_ACTION_DST);
                                    break;
                                case OVS_NAT_ATTR_IP_MIN:
                                    memcpy(&nat_action_info.min_addr,
                                            nl_attr_get(b_nest),
                                            nl_attr_get_size(b_nest));
                                    ip_min_specified = true;
                                    break;
                                case OVS_NAT_ATTR_IP_MAX:
                                    memcpy(&nat_action_info.max_addr,
                                            nl_attr_get(b_nest),
                                            nl_attr_get_size(b_nest));
                                    ip_max_specified = true;
                                    break;
                                case OVS_NAT_ATTR_PROTO_MIN:
                                    nat_action_info.min_port =
                                        nl_attr_get_u16(b_nest);
                                    proto_num_min_specified = true;
                                    break;
                                case OVS_NAT_ATTR_PROTO_MAX:
                                    nat_action_info.max_port =
                                        nl_attr_get_u16(b_nest);
                                    proto_num_max_specified = true;
                                    break;
                                //persistent，hash和random在 userspace
datapath中没用到
                                case OVS_NAT_ATTR_PERSISTENT:
                                case OVS_NAT_ATTR_PROTO_HASH:
                                case OVS_NAT_ATTR_PROTO_RANDOM:
                                    break;
                                case OVS_NAT_ATTR_UNSPEC:
                                case __OVS_NAT_ATTR_MAX:
                                    OVS_NOT_REACHED();
                                }
                            }

                            if (ip_min_specified && !ip_max_specified) {
                                nat_action_info.max_addr =
nat_action_info.min_addr;
                            }
                            if (proto_num_min_specified &&
!proto_num_max_specified) {
                                nat_action_info.max_port =
nat_action_info.min_port;
                            }
                            if (proto_num_min_specified ||
proto_num_max_specified) {
```

```
                                if (nat_action_info.nat_action &
NAT_ACTION_SRC) {
                                    nat_action_info.nat_action |=
NAT_ACTION_SRC_PORT;
                                } else if (nat_action_info.nat_action &
NAT_ACTION_DST) {
                                    nat_action_info.nat_action |=
NAT_ACTION_DST_PORT;
                                }
                            }
                            break;
                        }
                        }
                    }

                    /* We won't be able to function properly in this case,
hence
                     * complain loudly. */
                    if (nat_config && !commit) {
                        static struct vlog_rate_limit rl =
VLOG_RATE_LIMIT_INIT(5, 5);
                        VLOG_WARN_RL(&rl, "NAT specified without
commit.");
                    }
                    //struct dp_netdev 是全局的，所以 dp->conntrack 也是全局
的，多个pmd共享dp->conntrack
                    struct dp_netdev *dp = pmd->dp;
                    conntrack_execute(&dp->conntrack, packets_, aux->flow-
>dl_type, force, commit, zone, setmark, setlabel, helper,
nat_action_info_ref);
                //跳转到其他table执行
                case OVS_ACTION_ATTR_RECIRC:
                    if (*depth < MAX_RECIRC_DEPTH) {
                        ........
                    }
```

清除conntrack表项

创建datapath时，会启动专门的线程clean_thread_main清除超期的conntrack表项：`ct->clean_thread = ovs_thread_create("ct_clean", clean_thread_main, ct);`
主要处理流程如下：

```
int
conntrack_execute(struct conntrack *ct, struct dp_packet_batch *pkt_batch,
                  ovs_be16 dl_type, bool force, bool commit, uint16_t
zone,
                  const uint32_t *setmark,
                  const struct ovs_key_ct_labels *setlabel,
                  const char *helper,
                  const struct nat_action_info_t *nat_action_info)
    for (size_t i = 0; i < cnt; i++) {
        //从 pkts 中提取报文信息到 ct->key，并判断报文是否合法
        if (!conn_key_extract(ct, pkts[i], dl_type, &ctx, zone))
            ctx->key.zone = zone;
            ctx->key.dl_type = dl_type;
```

```
            extract_l3_ipv4(&ctx->key, l3, tail - (char *) l3, NULL,
!hwol_good_l3_csum);
                key->src.addr.ipv4 = ip->ip_src;
                key->dst.addr.ipv4 = ip->ip_dst;
                key->nw_proto = ip->ip_proto;
            extract_l4(&ctx->key, l4, tail - l4, &ctx->icmp_related, l3,
!hwol_good_l4_csum);
            //计算 hash 值
            ctx->hash = conn_key_hash(&ctx->key, ct->hash_basis);
        {
            //如果报文不合法，则设置 CS_INVALID 后，继续处理下一个报文
            pkts[i]->md.ct_state = CS_INVALID;
            write_ct_md(pkts[i], zone, NULL, NULL, NULL);
            continue;
        }
        //开始处理合法报文
        process_one(ct, pkts[i], &ctx, zone, force, commit, now, setmark,
setlabel, nat_action_info, helper);
            struct conn *conn;
            //根据 hash 值，得出一个 hash 桶
            unsigned bucket = hash_to_bucket(ctx->hash);
                #define CONNTRACK_BUCKETS_SHIFT 8
                #define CONNTRACK_BUCKETS (1 << CONNTRACK_BUCKETS_SHIFT)
                //hash 桶大小 256
                return (hash >> (32 - CONNTRACK_BUCKETS_SHIFT)) %
CONNTRACK_BUCKETS;

            //根据 ctx->key 查找 conn，如果是reply方向数据流，则设置reply标志
            conn_key_lookup(&ct->buckets[bucket], ctx, now);
                uint32_t hash = ctx->hash;
                struct conn *conn;
                HMAP_FOR_EACH_WITH_HASH (conn, node, hash, &ctb-
>connections) {
                    if (!conn_key_cmp(&conn->key, &ctx->key)
                            && !conn_expired(conn, now)) {
                        ctx->conn = conn;
                        ctx->reply = false;
                        break;
                    }
                    if (!conn_key_cmp(&conn->rev_key, &ctx->key)
                            && !conn_expired(conn, now)) {
                        ctx->conn = conn;
                        ctx->reply = true;
                        break;
                    }
                }
            conn = ctx->conn;

            /* Delete found entry if in wrong direction. 'force' implies
commit. */
            if (conn && force && ctx->reply) {
                conn_clean(ct, conn, &ct->buckets[bucket]);
                conn = NULL;
            }

            bool create_new_conn = false;
            struct conn conn_for_un_nat_copy;
            conn_for_un_nat_copy.conn_type = CT_CONN_TYPE_DEFAULT;
```

```
65              bool ftp_ctl = is_ftp_ctl(pkt);
66
67              if (OVS_LIKELY(conn)) {
68                  if (ftp_ctl) {
69                      /* Keep sequence tracking in sync with the source of
    the
70                       * sequence skew. */
71                      if (ctx->reply != conn->seq_skew_dir) {
72                          handle_ftp_ctl(ct, ctx, pkt, conn, now,
    CT_FTP_CTL_OTHER,
73                                         !!nat_action_info);
74                          create_new_conn = conn_update_state(ct, pkt, ctx,
    &conn, now,
75                                                              bucket);
76                      } else {
77                          create_new_conn = conn_update_state(ct, pkt, ctx,
    &conn, now,
78                                                              bucket);
79                          handle_ftp_ctl(ct, ctx, pkt, conn, now,
    CT_FTP_CTL_OTHER,
80                                         !!nat_action_info);
81                      }
82                  } else {
83                      create_new_conn = conn_update_state(ct, pkt, ctx,
    &conn, now,
84                                                          bucket);
85                  }
86                  if (nat_action_info && !create_new_conn) {
87                      handle_nat(pkt, conn, zone, ctx->reply, ctx-
    >icmp_related);
88                  }
89
90              }else if (check_orig_tuple(ct, pkt, ctx, now, &bucket, &conn,
91                                         nat_action_info)) {
92                  create_new_conn = conn_update_state(ct, pkt, ctx, &conn,
    now,
93                                                      bucket);
94              } else {
95                  if (ctx->icmp_related) {
96                      /* An icmp related conn should always be found; no new
97                         connection is created based on an icmp related
    packet. */
98                      pkt->md.ct_state = CS_INVALID;
99                  } else {
100                     create_new_conn = true;
101                 }
102             }
103
104             if (OVS_UNLIKELY(create_new_conn)) {
105                 conn = conn_not_found(ct, pkt, ctx, commit, now,
    nat_action_info, &conn_for_un_nat_copy, helper, alg_exp);
106                 unsigned bucket = hash_to_bucket(ctx->hash);
107                 struct conn *nc = NULL;
108
109                 //四层协议判断报文是否有效
110                 if (!valid_new(pkt, &ctx->key))
111                     return l4_protos[key->nw_proto]->valid_new(pkt);
112                 {
```

```
113                         pkt->md.ct_state = CS_INVALID;
114                         return nc;
115                     }
116 
117                     //设置 CS_NEW
118                     pkt->md.ct_state = CS_NEW;
119 
120                     //只有设置了 commit，才会将conn添加到hash表
121                     if (commit) {
122                         //判断是否超过 conn 表项最大限制
123                         unsigned int n_conn_limit;
124                         atomic_read_relaxed(&ct->n_conn_limit,
    &n_conn_limit);
125                         if (atomic_count_get(&ct->n_conn) >= n_conn_limit)
    {
126                             COVERAGE_INC(conntrack_full);
127                             return nc;
128                         }
129 
130                         //创建新表项
131                         nc = new_conn(&ct->buckets[bucket], pkt, &ctx-
    >key, now);
132                             struct conn *newconn;
133                             //tcp_new_conn
134                             newconn = l4_protos[key->nw_proto]-
    >new_conn(ctb, pkt, now);
135                             newconn->key = *key;
136                             return newconn;
137 
138                         ctx->conn = nc;
139                         nc->rev_key = nc->key;
140                         //翻转key
141                         conn_key_reverse(&nc->rev_key);
142 
143                         if (nat_action_info) {
144                             nc->nat_info = xmemdup(nat_action_info, sizeof
    *nc->nat_info);
145                             if (alg_exp) {
146                             } else {
147                                 *conn_for_un_nat_copy = *nc;
148                                 ct_rwlock_wrlock(&ct->resources_lock);
149                                 //根据nat配置，选择合适的ip和port
150                                 bool nat_res = nat_select_range_tuple(ct,
    nc, conn_for_un_nat_copy);
151                                     bool new_insert =
    nat_conn_keys_insert(&ct->nat_conn_keys, nat_conn, ct->hash_basis);
152                                         //将 nat 的conn插入 nat_conn_keys
153                                         hmap_insert(nat_conn_keys,
    &nat_conn_key->node, nat_conn_key_hash);
154                                 if (!nat_res) {
155                                     goto nat_res_exhaustion;
156                                 }
157                                 /* Update nc with nat adjustments made to
158                                  * conn_for_un_nat_copy by
    nat_select_range_tuple(). */
159                                 *nc = *conn_for_un_nat_copy;
160                                 ct_rwlock_unlock(&ct->resources_lock);
161                             }
```

```
                            //设置 conn_type 为 CT_CONN_TYPE_UN_NAT，表示此表项需要nat
                            conn_for_un_nat_copy->conn_type = CT_CONN_TYPE_UN_NAT;
                            conn_for_un_nat_copy->nat_info = NULL;
                            conn_for_un_nat_copy->alg = NULL;
                            //将报文做nat转换
                            nat_packet(pkt, nc, ctx->icmp_related);
                                if (conn->nat_info->nat_action & NAT_ACTION_SRC) {
                                    pkt->md.ct_state |= CS_SRC_NAT;
                                    if (conn->key.dl_type == htons(ETH_TYPE_IP)) {
                                        struct ip_header *nh = dp_packet_l3(pkt);
                                        packet_set_ipv4_addr(pkt, &nh->ip_src, conn->rev_key.dst.addr.ipv4_aligned);
                                    }
                                    if (!related) {
                                        pat_packet(pkt, conn);
                                    }
                                } else if (conn->nat_info->nat_action & NAT_ACTION_DST) {
                                    pkt->md.ct_state |= CS_DST_NAT;
                                }
                        }
                        //将新建表项插入hash表
                        hmap_insert(&ct->buckets[bucket].connections, &nc->node, ctx->hash);
                        //增加表项个数
                        atomic_count_inc(&ct->n_conn);
                    }
                    return nc;
            }

            write_ct_md(pkt, zone, conn, &ctx->key, alg_exp);
                pkt->md.ct_state |= CS_TRACKED;
                pkt->md.ct_zone = zone;
                pkt->md.ct_mark = conn ? conn->mark : 0;
                pkt->md.ct_label = conn ? conn->label : OVS_U128_ZERO;

                pkt->md.ct_orig_tuple_ipv6 = false;
                if (key) {
                    if (key->dl_type == htons(ETH_TYPE_IP)) {
                        //ct_orig_tuple 保存原始报文(第一次进ct模块时)的五元组信息
                        pkt->md.ct_orig_tuple.ipv4 = (struct ovs_key_ct_tuple_ipv4) {
                            key->src.addr.ipv4_aligned,
                            key->dst.addr.ipv4_aligned,
                            key->nw_proto != IPPROTO_ICMP
                            ? key->src.port : htons(key->src.icmp_type),
                            key->nw_proto != IPPROTO_ICMP
                            ? key->dst.port : htons(key->src.icmp_code),
                            key->nw_proto,
                        };
                    }
                }
```

```
210
211             if (conn && setmark) {
212                 set_mark(pkt, conn, setmark[0], setmark[1]);
213             }
214
215             if (conn && setlabel) {
216                 set_label(pkt, conn, &setlabel[0], &setlabel[1]);
217             }
218     }
219 }
```

发包模块

概要流程

```
1  kernel:
2  netdev_frame_hook-->netdev_port_receive-->ovs_vport_receive--
   >ovs_dp_process_packet-->ovs_execute_actions-->do_execute_actions--
   >do_output
3  这里会根据dev的mtu和报文的mru比较需要不需要分片，通常情况下报文的mru是0，不需要分片，但
   是因为重组函数handle_fragments重组时设置了报文的mru，所以分片时也根据此项来判定。调用
   ovs_fragment分片完成之后会调用ovs_vport_send-->dev_queue_xmit--
   >dev_hard_start_xmit-->ops->ndo_start_xmit-->bond_start_xmit进行报文发送。
4
5  另一种的报文分片是VM中设置了TSO、GSO、UFO等features，导致报文比较大，这个时候需要根据标
   志判定是否需要分片。
6
7  netdev_frame_hook-->netdev_port_receive-->ovs_vport_receive--
   >ovs_dp_process_packet-->ovs_execute_actions-->do_execute_actions--
   >do_output-->ovs_vport_send-->dev_queue_xmit-->dev_hard_start_xmit这里会根据
   gso进行分片，分片完成后继续调用ops->ndo_start_xmit-->bond_start_xmit进行报文发送。
   其实此处也会根据其他的一些特性来判定，一般比如带了TSO之类的需要网卡去分片。
8
9  OVS-DPDK:
10 pmd_thread_main-->dp_netdev_process_rxq_port-->dp_netdev_input--
   >dp_netdev_input__-->packet_batch_per_flow_execute--
   >dp_netdev_execute_actions-->dp_execute_cb-->netdev_send--
   >netdev_dpdk_eth_send
```

日志模块

相关配置命令

三种输出方式: syslog, console, file
查看log使能情况命令：`ovs-appctl -t ovsdb-server vlog/list`
设置所有模块log等级命令：`ovs-appctl -t ovsdb-server vlog/set dbg`
设置指定模块log等级命令，如jsonrpc：`ovs-appctl -t ovsdb-server vlog/set jsonrpc,dbg`

后续计划

写后续计划的目的是为了让大家督促我继续总结学习文档，毕竟话撂这了，不做到也挺打脸。后面计划是更新一篇DPDK相关内容和一篇流表算法内容。

路由转发

push eth

在函数`odp_execute_actions`中，如果需要执行push eth操作，他的eth信息是从哪里获取的呢?

```
    fast_path_processing

    dp_netdev_execute_actions

    odp_execute_actions

         |
    dpif_execute



    dpif_get_actions
         /* Lookup actions in userspace cache. */
         //upcall->ufid  流唯一的id
         //upcall->pmd_id  datapath pmd id
            struct udpif_key *ukey = ukey_lookup(udpif, upcall->ufid,
                                                 upcall->pmd_id);
            if (ukey) {
                ukey_get_actions(ukey, actions, &actions_len);
            }

        udpif_revalidator
            |
        revalidate
            |
        ukey_acquire
            |
ukey_create_from_dpif_flow
            |
            ukey_create__


xlate_actions


```

```
虚拟化笔记，内核态：
https://zhuanlan.zhihu.com/p/337143779

main
 └─bridge_run
    └─bridge_reconfigure
       └─ofproto_create
          └─construct
             └─open_dpif_backer
                ├─udpif_create
                |   └─dpif_register_upcall_cb//kernel为NULL，用于DPDK
                ├─udpif_set_threads
                |  └─dpif_handlers_set
                └─udpif_start_threads
```

```
                    └─udpif_upcall_handler
                       ├─dpif_recv
                       |  └─dpif_netlink_recv
                       |     └─dpif_netlink_recv__
                       |        └─parse_odp_packet
                       └─recv_upcalls

#linux kernel datapath
ovs_vport_receive
  ├─ovs_flow_key_extract
  |   └─key_extract
  └─ovs_dp_process_packet
      ├─if ovs_flow_tbl_lookup_stats
      |   └─ovs_dp_upcall
      |      └─queue_userspace_packet
      |          └─ovs_nla_put_key
      |              └─__ovs_nla_put_key
      └─else ovs_execute_actions
          ├─case OVS_ACTION_ATTR_OUTPUT
          └─do_output
              └─ovs_vport_send


recv_upcalls
 ├─odp_flow_key_to_flow
 ├─upcall_receive
 |  └─xlate_lookup
 |     └─xlate_lookup_ofproto_
 ├─process_upcall
 | └─upcall_xlate
 |   ├─xlate_in_init
 |   ├─xlate_actions
 |   |  ├─rule_dpif_lookup_from_table
 |   |  |  └─rule_dpif_lookup_in_table
 |   |  ├─rule_get_actions
 |   |  └─do_xlate_actions
 |   |     ├─freeze_unroll_actions
 |   |     |  └─freeze_put_unroll_xlate
 |   |     ├─xlate_group_action
 |   |     |  ├─pick_select_group//报文第一次找不到bucket
 |   |     |  |  └─pick_dp_hash_select_group
 |   |     |  |     └─ctx_trigger_recirculate_with_hash
 |   |     |  └─xlate_group_bucket//第二次找到bucket才执行这里
 |   |     |     ├─ofpacts_execute_action_set
 |   |     |     └─do_xlate_actions//相当于递归了
 |   |     └─finish_freezing//报文第一次来执行这里
 |   |        ├─xlate_commit_actions
 |   |        |  └─commit_odp_actions
 |   |        └─finish_freezing__
 |   |           ├─recirc_alloc_id_ctx
 |   |           ├─把OVS_ACTION_ATTR_HASH和OVS_ACTION_ATTR_RECIRC下给datapath
 |   |           └─ctx_cancel_freeze
 |   └─ukey_create_from_upcall
 └─handle_upcalls
    └─dpif_operate
       ├─ukey_install
       └─dpif_operate//给datapath安装流表和把skb发回内核
          ├─if dpif_netlink_operate
```

```
          └─else dpif_execute_with_help
             └─odp_execute_actions
                ├─计算hash
                └─dpif_execute_helper_cb
                   └─dpif_operate


内核datapath流表老化：
udpif_start_threads
  └─udpif_revalidator
     ├─revalidate
     |   ├─dpif_flow_dump_next
     |   ├─udpif_get_n_flows
     |   ├─delete_op_init__
     |   └─push_dp_ops
     └─revalidator_sweep
```

```
用户态：
https://blog.csdn.net/qq_20817327/article/details/106761936

dp_execute_cb
如果是OVS_ACTION_ATTR_OUTPUT，调用dp_netdev_lookup_port查找端口，
然后调用netdev_send进行报文发送。

如果是OVS_ACTION_ATTR_TUNNEL_PUSH，调用push_tnl_action进行tunnel封装，然后调用
dp_netdev_recirculate->dp_netdev_input__重新查表操作。

如果是OVS_ACTION_ATTR_TUNNEL_POP，调用netdev_pop_header解封装，
然后调用dp_netdev_recirculate->dp_netdev_input__重新查表操作。
```

```c
struct nm_rule_action {
    uint64_t flag; /* action flag, eg:set ipv4 src/redirect */
    uint32_t drop_flag; /* drop or forward */
    uint32_t soft_id; /* an unique value for this rule todo*/
    uint32_t mark_id;
    uint32_t counter_id;
    uint8_t outer_type; //outer_tnl_type,eg:tunnel
    uint8_t queue; /* assigned rx queue */
    uint8_t queues_cnt;
    uint8_t action_cnt;  /* different action type total cnt */

    uint16_t metadata;  /* secondary flow table need */

    /* set ops */
    struct nm_fdir_l4 l4_outer;
    struct nm_fdir_l2 l2_data_outer;
    struct nm_fdir_l3 ip_outer;

    /* push ops */
    struct nm_flow_item_vlan vlan;
    struct nm_flow_item_vxlan vxlan;
    uint8_t set_flag;

```

```
    uint8_t buf[NM_FLOW_ACTION_SET_OUTER_HEADER_MAX_LEN];
    uint32_t l2_cnt:2;
    uint32_t l3_cnt:2;
    uint32_t l4_cnt:2;
    uint32_t tnl_cnt:2;
    uint32_t vlan_cnt:2;
    uint32_t rsv:22;
};


/* SPDX-License-Identifier: BSD-3-Clause
 * Copyright(c) 2021-2024 Nebula Limited.
 */

#ifndef _NM_FLOW_H_
#define _NM_FLOW_H_

#define DESC (x)                        1
#define FALSE                           0
#define TRUE                            1
#define ERR             4
#define WARN            3
#define NOTICE          2
#define INFO            1

#define NM_RTE_ETHER_TYPE_IPV4          4
#define NM_RTE_ETHER_TYPE_IPV6          6

/* l4 outer tnl hash tab entries: 1024 */
#define NM_MAX_L4_OUTER_TNL_FILTER_NUM      (1<<10)
#define NM_MAX_L2_ETHTYPE_TNL_FILTER_NUM    (1<<10)
#define NM_MAX_L3_L2_TNL_FILTER_NUM    (1<<10)
#define NM_MAX_ACT_CNT_MAX_NUM    (1<<17)   /* 131072 */


#define NM_FLOW_TABLE_LENGTH            3
#define NM_FLOW_TABLE_VXLAN_LENGTH      3

#define NM_FLOW_TAB_PATTERN_TIMES   1
#define NM_FLOW_TABLE_IPV4_DEFAULT_MASK       0xFFFFFFFF
#define NM_FLOW_TABLE_L4_PORT_DEFAULT_MASK    0xFFFF
#define NM_MAX_FILTER_ID                      0x0FFF
#define NM_FLOW_TABLE_FULL_MASK_AS_U8         0xFF
#define NM_FLOW_ACTION_SET_OUTER_HEADER_MAX_LEN 128
#define NM_FLOW_ACTION_VLAN_CNT         2
#define NM_ETH_MAC_LEN               6
#define NM_ACTION_DEL_MIN_REF_CNT  1

/* input set */
#define NM_INSET_NONE               0ULL

#define nm_malloc(h, s)    rte_zmalloc(NULL, s, 0)
#define nm_calloc(h, c, s) rte_zmalloc(NULL, (c) * (s), 0)
#define nm_free(h, m)         rte_free(m)

#define NM_IPV6_ADDR_LEN_AS_U32    4
#define NM_IPV6_ADDR_LEN_AS_U8     16

```

```
/* Field range used to indicate pattern */
#define NM_PROT_MAC_INNER           (1ULL << 1)
#define NM_PROT_MAC_OUTER           (1ULL << 2)
#define NM_PROT_VLAN_INNER          (1ULL << 3)
#define NM_PROT_VLAN_OUTER          (1ULL << 4)
#define NM_PROT_IPV4_INNER          (1ULL << 5)
#define NM_PROT_IPV4_OUTER          (1ULL << 6)
#define NM_PROT_IPV6_INNER          (1ULL << 7)
#define NM_PROT_IPV6_OUTER          (1ULL << 8)
#define NM_PROT_TCP_INNER           (1ULL << 9)
#define NM_PROT_TCP_OUTER           (1ULL << 10)
#define NM_PROT_UDP_INNER           (1ULL << 11)
#define NM_PROT_UDP_OUTER           (1ULL << 12)
#define NM_PROT_SCTP_INNER          (1ULL << 13)
#define NM_PROT_SCTP_OUTER          (1ULL << 14)
#define NM_PROT_VXLAN               (1ULL << 15)
#define NM_PROT_NVGRE               (1ULL << 16)
#define NM_PROT_GTPU                (1ULL << 17)
#define NM_PROT_PPPOE_S             (1ULL << 18)
#define NM_PROT_ESP                 (1ULL << 19)
#define NM_PROT_AH                  (1ULL << 20)
#define NM_PROT_L2TPV3OIP           (1ULL << 21)
#define NM_PROT_PFCP                (1ULL << 22)

//action flag
#define NM_FLOW_ACTION_DROP                     (1ULL << 1)
#define NM_FLOW_ACTION_REDIRECT                 (1ULL << 2)
#define NM_FLOW_ACTION_MIRRED                   (1ULL << 3)
#define NM_FLOW_ACTION_VLAN_PUSH                (1ULL << 4)
#define NM_FLOW_ACTION_VLAN_POP                 (1ULL << 5)
#define NM_FLOW_ACTION_TUNNEL_ENCAP             (1ULL << 5)
#define NM_FLOW_ACTION_TUNNEL_DECAP             (1ULL << 6)
#define NM_FLOW_ACTION_COUNTER                  (1ULL << 7)
#define NM_FLOW_ACTION_SET_IPV4_SRC_IP          (1ULL << 8)
#define NM_FLOW_ACTION_SET_IPV4_DST_IP          (1ULL << 9)
#define NM_FLOW_ACTION_SET_IPV6_SRC_IP          (1ULL << 10)
#define NM_FLOW_ACTION_SET_IPV6_DST_IP          (1ULL << 11)
#define NM_FLOW_ACTION_SET_SRC_MAC              (1ULL << 12)
#define NM_FLOW_ACTION_SET_DST_MAC              (1ULL << 13)
#define NM_FLOW_ACTION_SET_SRC_PORT             (1ULL << 14)
#define NM_FLOW_ACTION_SET_DST_PORT             (1ULL << 15)
#define NM_FLOW_ACTION_SET_TTL                  (1ULL << 16)
#define NM_FLOW_ACTION_SET_DSCP                 (1ULL << 17)
#define NM_FLOW_ACTION_RSS                      (1ULL << 18)
#define NM_FLOW_ACTION_QUEUE                    (1ULL << 19)
#define NM_FLOW_ACTION_MARK                     (1ULL << 20)
#define NM_FLOW_ACTION_PUSH_VLAN                (1ULL << 21)
#define NM_FLOW_ACTION_POP_VLAN                 (1ULL << 22)
#define NM_FLOW_ACTION_METADATA_FLAG            (1ULL << 23)


// action:vxlan encap, set_flag
#define NM_FLOW_ACTION_ETH_FLAG                     (1ULL << 1)
#define NM_FLOW_ACTION_IPV4_FLAG                    (1ULL << 2)
#define NM_FLOW_ACTION_IPV6_FLAG                    (1ULL << 3)
#define NM_FLOW_ACTION_TCP_FLAG                     (1ULL << 4)
#define NM_FLOW_ACTION_UDP_FLAG                     (1ULL << 5)
#define NM_FLOW_ACTION_VLAN_FLAG                    (1ULL << 7)
```

```
#define NM_FLOW_ACTION_VXLAN_FLAG                    (1ULL << 8)


enum nm_flow_service_type {
    NM_FLOW_FILTER_NONE = 0,
    NM_L4_UPPER_TUNNEL_OUTER,
    NM_L4_UPPER_OUTER_AND_INNER_ETHTYPE,
    NM_DOWNER_NON_IP,
    NM_DOWNER_IP,
    NM_DOWNER_IP_AND_SECGRP,
    NM_SECURITY_GROUP,
    NM_UPPER_L2_MULTICAST_VLAN,
    NM_UPPER_L2_MULTICAST,
    NM_UPPER_L3_MULTICAST,
    NM_DOWNER_MULTICAST,
};

enum nm_fltr_ptype {
    /* NONE - used for undef/error */
    NM_FLTR_PTYPE_NONF_NONE = 0,
    NM_FLTR_PTYPE_NONF_IPV4_UDP,
    NM_FLTR_PTYPE_NONF_IPV4_TCP,
    NM_FLTR_PTYPE_NONF_IPV4_SCTP,
    NM_FLTR_PTYPE_NONF_IPV4_OTHER,
    NM_FLTR_PTYPE_NONF_IPV4_GTPU_IPV4_UDP,
    NM_FLTR_PTYPE_NONF_IPV4_GTPU_IPV4_TCP,
    NM_FLTR_PTYPE_NONF_IPV4_GTPU_IPV4_ICMP,
    NM_FLTR_PTYPE_NONF_IPV4_GTPU_IPV4_OTHER,
    NM_FLTR_PTYPE_NONF_IPV6_GTPU_IPV6_OTHER,
    NM_FLTR_PTYPE_NONF_IPV4_GTPU_EH_IPV4_OTHER,
    NM_FLTR_PTYPE_NONF_IPV6_GTPU_EH_IPV6_OTHER,
    NM_FLTR_PTYPE_NONF_IPV4_L2TPV3,
    NM_FLTR_PTYPE_NONF_IPV6_L2TPV3,
    NM_FLTR_PTYPE_NONF_IPV4_ESP,
    NM_FLTR_PTYPE_NONF_IPV6_ESP,
    NM_FLTR_PTYPE_NONF_IPV4_AH,
    NM_FLTR_PTYPE_NONF_IPV6_AH,
    NM_FLTR_PTYPE_NONF_IPV4_NAT_T_ESP,
    NM_FLTR_PTYPE_NONF_IPV6_NAT_T_ESP,
    NM_FLTR_PTYPE_NONF_IPV4_PFCP_NODE,
    NM_FLTR_PTYPE_NONF_IPV4_PFCP_SESSION,
    NM_FLTR_PTYPE_NONF_IPV6_PFCP_NODE,
    NM_FLTR_PTYPE_NONF_IPV6_PFCP_SESSION,
    NM_FLTR_PTYPE_NON_IP_L2,
    NM_FLTR_PTYPE_FRAG_IPV4,
    NM_FLTR_PTYPE_NONF_IPV6_UDP,
    NM_FLTR_PTYPE_NONF_IPV6_TCP,
    NM_FLTR_PTYPE_NONF_IPV6_SCTP,
    NM_FLTR_PTYPE_NONF_IPV6_OTHER,
    NM_FLTR_PTYPE_MAX,
};

struct rte_flow {
    uint8_t filter_type;
    void *rule;
    uint32_t counter_id;
};

```

```c
struct nm_fdir_l2 {
    uint8_t dst_mac[NM_ETH_MAC_LEN];  /* dest MAC address */
    uint8_t src_mac[NM_ETH_MAC_LEN];  /* src MAC address */
    uint16_t ether_type;        /* for NON_IP_L2 */
};

struct nm_fdir_session {
    uint32_t session_id;
};

struct nm_fdir_l4 {
    uint16_t dst_port;
    uint16_t src_port;
};

struct nm_fdir_l3{
    uint32_t dst_ip[NM_IPV6_ADDR_LEN_AS_U32];
    uint32_t src_ip[NM_IPV6_ADDR_LEN_AS_U32];
    uint8_t ip_ver;
    uint8_t tos;
    uint8_t ttl;
    uint32_t l4_header; /* next header */
    uint32_t sec_parm_idx; /* security parameter index */
    uint8_t tc;
    uint8_t proto;
    uint8_t hlim;
    uint16_t hdr_checksum;
};

/* upper outer tunnel tabl */

struct nm_l4_outer_tunnel_conf {
    union {
        uint32_t tunnel_id;
        uint8_t vni[NM_FLOW_TABLE_LENGTH];
    };

    uint16_t dst_port;
    uint8_t  dip[NM_IPV6_ADDR_LEN_AS_U8];   /**< IP address of destination host(s). */
};

/*  upper tnl inner tab  */
struct nm_l2_inner_tunnel_conf {
    uint32_t port;   /* packet forward port */
    uint16_t metadata;
    struct nm_fdir_l2 l2;
};

/* downer  non ip */
struct nm_l2_tunnel_conf {
    uint32_t port;
    struct nm_fdir_l2 l2;
};

/* down ip */
struct nm_l3_tunnel_conf {
    uint32_t port;
```

```
    struct nm_fdir_l2 l2;
    uint8_t  proto;
    uint8_t  sip[16];
    uint8_t  dip[16];
};

/* security grooup */
struct nm_l4_sec_group_conf {
    uint16_t metadata;
    uint16_t src_port;
    uint16_t dst_port;
};

struct nm_l4_outer_tunnel_conf_ele {
    TAILQ_ENTRY(nm_l4_outer_tunnel_conf_ele) entries;
    struct nm_l4_outer_tunnel_conf filter_info;
    void *child;
    void*  act;
};

struct nm_l2_inner_tunnel_conf_ele {
    TAILQ_ENTRY(nm_l2_inner_tunnel_conf_ele) entries;
    struct nm_l2_inner_tunnel_conf filter_info;
    void *child;
    void *act;
};

struct nm_ethertype_filter_ele {
    TAILQ_ENTRY(nm_ethertype_filter_ele) entries;
    struct nm_l2_tunnel_conf filter_info;
    void *child;
    void *act;
};

struct nm_l3_tunnel_conf_ele {
    TAILQ_ENTRY(nm_l3_tunnel_conf_ele) entries;
    struct nm_l3_tunnel_conf filter_info;
    void *child;
    void *act;
};

struct nm_l4_secgrp_conf_ele {
    TAILQ_ENTRY(nm_l4_secgrp_conf_ele) entries;
    struct nm_l4_sec_group_conf filter_info;
    void *parent;
    void *act;
};


/* rte_flow store */
struct nm_flow_mem {
    TAILQ_ENTRY(nm_flow_mem) entries;
    struct rte_flow *flow;
};

TAILQ_HEAD(nm_l4_outer_tun_filter_list, nm_l4_outer_tunnel_conf_ele);
TAILQ_HEAD(nm_l2_inner_tun_filter_list, nm_l2_inner_tunnel_conf_ele);
TAILQ_HEAD(nm_ethertype_filter_list, nm_ethertype_filter_ele);
```

```
313 TAILQ_HEAD(nm_l3_tunnel_filter_list, nm_l3_tunnel_conf_ele);
314 TAILQ_HEAD(nm_l4_secgrp_filter_list, nm_l4_secgrp_conf_ele);
315 TAILQ_HEAD(nm_flow_mem_list, nm_flow_mem);
316
317 /* hash-list struct  */
318 struct nm_l4_outer_tnl_filter {
319     TAILQ_ENTRY(nm_l4_outer_tnl_filter)    entries;
320     struct nm_l4_outer_tunnel_conf          *key;
321 };
322
323 TAILQ_HEAD(nm_l4_outer_tnl_hash_list, nm_l4_outer_tnl_filter);
324
325 struct nm_l4_outer_tnl_hash_info {
326     struct nm_l4_outer_tnl_hash_list  l4_outer_tnl_hash_list;
327     struct nm_l4_outer_tnl_filter    **hash_map;
328     struct rte_hash                  *hash_handle;
329 };
330
331 struct nm_l2_ethtype_tnl_filter {
332     TAILQ_ENTRY(nm_l2_ethtype_tnl_filter)    entries;
333     struct nm_l2_tunnel_conf          *key;
334 };
335
336 TAILQ_HEAD(nm_l2_ethtype_tnl_hash_list, nm_l2_ethtype_tnl_filter);
337
338 struct nm_l2_ethtype_tnl_hash_info {
339     struct nm_l2_ethtype_tnl_hash_list  l2_ethtype_tnl_hash_list;
340     struct nm_l2_ethtype_tnl_filter    **hash_map;
341     struct rte_hash                  *hash_handle;
342 };
343
344 struct nm_l3_tnl_filter {
345     TAILQ_ENTRY( nm_l3_tnl_filter)    entries;
346     struct nm_l3_tunnel_conf          *key;
347 };
348
349 TAILQ_HEAD(nm_l3_hash_list, nm_l3_tnl_filter);
350
351 struct nm_l3_hash_info {
352     struct nm_l3_hash_list           l3_hash_list;
353     struct nm_l3_tnl_filter          **hash_map;
354     struct rte_hash                  *hash_handle;
355 };
356
357 struct nm_flow_item_vxlan {
358     uint8_t flags;
359     uint8_t vni[NM_FLOW_TABLE_VXLAN_LENGTH];
360 };
361
362 struct nm_flow_item_vlan{
363     uint16_t vlan_tci;  /**< Priority (3) + CFI (1) + Identifier Code (12)
    */
364     uint16_t eth_proto;
365 };
366
367
368 struct nm_rule_action {
369     uint64_t flag; /* action flag, eg:set ipv4 src/redirect */
```

```
370     uint32_t drop_flag; /* drop or forward */
371     uint32_t soft_id; /* an unique value for this rule todo*/
372     uint32_t mark_id;
373     uint32_t counter_id;
374     uint8_t outer_type; //outer_tnl_type,eg:tunnel
375     uint8_t queue; /* assigned rx queue */
376     uint8_t queues_cnt;
377     uint8_t action_cnt;  /* different action type total cnt */
378
379     uint16_t metadata;  /* secondary flow table need */
380
381     /* set ops */
382     struct nm_fdir_l4 l4_outer;
383     struct nm_fdir_l2 l2_data_outer;
384     struct nm_fdir_l3 ip_outer;
385
386     /* push ops */
387     struct nm_flow_item_vlan vlan;
388     struct nm_flow_item_vxlan vxlan;
389     uint8_t set_flag;
390
391     uint8_t buf[NM_FLOW_ACTION_SET_OUTER_HEADER_MAX_LEN];
392     uint32_t l2_cnt:2;
393     uint32_t l3_cnt:2;
394     uint32_t l4_cnt:2;
395     uint32_t tnl_cnt:2;
396     uint32_t vlan_cnt:2;
397     uint32_t rsv:22;
398 };
399
400 struct nm_fdir_fltr {
401     struct nm_fdir_l3 ip;
402     struct nm_fdir_l3 ip_mask;
403
404     struct nm_fdir_l3 ip_outer;
405     struct nm_fdir_l3 ip_mask_outer;
406
407     struct nm_fdir_l4 l4;
408     struct nm_fdir_l4 l4_mask;
409
410     struct nm_fdir_l4 l4_outer;
411     struct nm_fdir_l4 l4_mask_outer;
412
413     struct nm_fdir_l2 l2_data_outer;
414     struct nm_fdir_l2 l2_mask_outer;
415
416     struct nm_fdir_l2 l2_data;
417     struct nm_fdir_l2 l2_mask;
418
419     uint16_t vlan_type;         /* VLAN ethertype */
420     uint16_t vlan_tag;          /* VLAN tag info */
421
422     struct nm_fdir_session l2tpv3_data;
423     struct nm_fdir_session l2tpv3_mask;
424
425     union{
426         uint32_t tunnel_id;
427         struct nm_flow_item_vxlan vxlan;
```

```
428     }tnl,tnl_mask;
429
430     uint32_t l2_cnt:1;
431     uint32_t l3_cnt:1;
432     uint32_t l4_cnt:1;
433     uint32_t tnl_cnt:1;
434     uint32_t rsv:28;
435
436     uint32_t port;
437 };
438
439 struct nm_acl_conf {
440     struct nm_fdir_fltr input;
441     uint64_t input_set;   //destroy rule by input-set
442 };
443
444 struct nm_action_filter {
445     TAILQ_ENTRY( nm_action_filter)    entries;
446     struct nm_rule_action          key;
447     uint32_t ref_cnt;  //ref times by different key type
448 };
449
450 TAILQ_HEAD(nm_action_hash_list, nm_action_filter);
451
452 struct nm_action_hash_info {
453     struct nm_action_hash_list        action_hash_list;
454     struct nm_action_filter          **hash_map;
455     struct rte_hash                  *hash_handle;
456 };
457
458 struct nm_filter_info {
459     struct nm_flow_mem_list nm_flow_list;
460     struct nm_l4_outer_tun_filter_list filter_l4_outer_list;
461     struct nm_l2_inner_tun_filter_list filter_l2_inner_list;
462     struct nm_ethertype_filter_list filter_ethtype_list;
463     struct nm_l3_tunnel_filter_list filter_l3_tunnel_list;
464     struct nm_l4_secgrp_filter_list filter_l4_tunnel_list;
465
466     struct nm_l4_outer_tnl_hash_info l4_outer_hash;
467     struct nm_l2_ethtype_tnl_hash_info l2_eth_hash;
468     struct nm_l3_hash_info l3_hash;
469     struct nm_action_hash_info act;
470 };
471
472 #define NM_DEV_PRIVATE_TO_FILTER_INFO(adapter) \
473                         (&(struct nm_adapter *)adapter->filter_info)
474
475 struct nm_flow_counter {
476     uint32_t shared:1;   /* Share counter ID with other flow rules. */
477     uint32_t ref_cnt:31; /* Reference counter. */
478     uint16_t id;   /* Counter ID. */
479     uint64_t hits; /* Number of packets matched by the rule. */
480 };
481
482 void nm_flow_init(struct rte_eth_dev *dev);
483 void nm_flow_fini(struct rte_eth_dev *dev);
484
485 #endif
```

```
486
```