流表配置

结合测试的文档我已经总结了个ovs-dpdk的常见命令行文档: http://10.10.3.15:8090/pages/viewpage.action?pageId=7910136

```
#正常转发（最简转发）
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,actions=output:dpdk1"
ovs-ofctl add-flow br0 "table=0,in_port=dpdk1,actions=output:dpdk0"

#2级流表转发
ovs-ofctl add-flow br0 "table=0,actions=goto_table=1"
ovs-ofctl add-flow br0 "table=1,in_port=dpdk0,actions=output:dpdk1"
ovs-ofctl add-flow br0 "table=1,in_port=dpdk1,actions=output:dpdk0"

#3级流表转发
ovs-ofctl add-flow br0 "table=0,actions=goto_table=1"
ovs-ofctl add-flow br0 "table=1,actions=goto_table=2"
ovs-ofctl add-flow br0 "table=2,in_port=dpdk0,actions=output:dpdk1"
ovs-ofctl add-flow br0 "table=2,in_port=dpdk1,actions=output:dpdk0"

#匹配SMAC流转发
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,ip,dl_src=B8:CE:01:01:00:33,actions=output:dpdk1"

#匹配DMAC流转发
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,ip,dl_dst=B8:EE:01:01:00:99,actions=output:dpdk1"

#匹配S+DMAC流转发
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,ip,ip,dl_src=B8:CE:01:01:00:33,dl_dst=B8:EE:01:01:00:99,actions=output:dpdk1"

#匹配VLAN流转发
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,dl_vlan=100,actions=output:dpdk1"

```

```
#匹配SMAC+SIP转发
vs-ofctl add-flow br0 "table=0,in_port=dpdk0,ip,nw_src=192.168.0.0/32,dl_src=B8:CE:01:01:00:33,actions=output:dpdk1"

#匹配DMAC+DIP转发
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,ip,nw_dst=3.3.3.3,dl_dst=B8:EE:01:01:00:99,actions=output:dpdk1"

#匹配SMAC+DMAC+SIP+DIP转发
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,ip,nw_src=192.168.0.0/32,dl_src=B8:CE:01:01:00:33,nw_dst=3.3.3.3,dl_dst=B8:EE:01:01:00:99,actions=output:dpdk1"

#匹配SMAC+DMAC+SIP+DIP+S_TCP+D_TCP转发
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,tcp,nw_src=192.168.0.0/32,dl_src=B8:CE:01:01:00:33,nw_dst=3.3.3.3,dl_dst=B8:EE:01:01:00:99,tp_src=1024,tp_dst=1024,actions=output:dpdk1"

#匹配SMAC+DMAC+SIP+DIP+S_UCP+D_UCP转发
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,udp,nw_src=192.168.0.0/32,dl_src=B8:CE:01:01:00:33,nw_dst=3.3.3.3,dl_dst=B8:EE:01:01:00:99,tp_src=1024,tp_dst=1024,actions=output:dpdk1"

#加一层vlan
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,actions=mod_vlan_vid:10,output:dpdk1"

#剥一层vlan
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,dl_vlan=100,actions=strip_vlan,output:dpdk1"

#修改单层vlan
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,dl_vlan=100,actions=mod_vlan_vid:10,output:dpdk1"

#修改源mac
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,actions=mod_dl_src:B8:CE:01:01:00:35,output:dpdk1"

#修改目的mac
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,actions=mod_dl_dst:B8:EE:01:01:00:88,output:dpdk1"

#修改源+目的mac
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,actions=mod_dl_src:B8:CE:01:01:00:35,mod_dl_dst:B8:EE:01:01:00:88,output:dpdk1"

#修改源mac+vlan
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,actions=mod_dl_src:B8:CE:01:01:00:35,mod_vlan_vid:10,output:dpdk1"

#修改源+目的mac+vlan
```

```
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,actions=mod_dl_src:B8:CE:01:01:00:35,mod_dl_dst:B8:EE:01:01:00:88,mod_vlan_vid:10,output:dpdk1"

#修改源ip
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,actions=mod_nw_src:192.169.0.2,output:dpdk1"

#修改目的ip
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,actions=mod_nw_dst:5.5.5.5,output:dpdk1"

#修改源+目的ip
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,actions=mod_nw_src:192.169.0.2,mod_nw_dst:5.5.5.5,output:dpdk1"

#修改源mac+目的mac+源IP+目的IP
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,actions=mod_dl_src:B8:CE:01:01:00:35,mod_dl_dst:B8:EE:01:01:00:88,mod_nw_src:192.169.0.2,mod_nw_dst:5.5.5.5,output:dpdk1"

#修改源mac+目的mac+源IP+目的IP+vlan
ovs-ofctl add-flow br0 "table=0,in_port=dpdk0,actions=mod_dl_src:B8:CE:01:01:00:35,mod_dl_dst:B8:EE:01:01:00:88,mod_nw_src:192.169.0.2,mod_nw_dst:5.5.5.5,mod_vlan_vid:10,output:dpdk1"

#修改源mac+目的mac+源IP+目的IP+vlan+tcp源端口+tcp目的端口
ovs-ofctl add-flow br0 "table=0,tcp,in_port=dpdk0,actions=mod_dl_src:B8:CE:01:01:00:35,mod_dl_dst:B8:EE:01:01:00:88,mod_nw_src:192.169.0.2,mod_nw_dst:5.5.5.5,mod_vlan_vid:10,mod_tp_src=1000,mod_tp_dst=1000,output:dpdk1"

#修改源mac+目的mac+目的IP+vlan+tcp源端口+tcp目的端口
ovs-ofctl add-flow br0 "table=0,tcp,in_port=dpdk0,actions=mod_dl_src:B8:CE:01:01:00:35,mod_dl_dst:B8:EE:01:01:00:88,mod_nw_dst:5.5.5.5,mod_vlan_vid:10,mod_tp_src=1000,mod_tp_dst=1000,output:dpdk1"

#修改源mac+目的mac+目的IP+vlan+tcp目的端口
ovs-ofctl add-flow br0 "table=0,tcp,in_port=dpdk0,actions=mod_dl_src:B8:CE:01:01:00:35,mod_dl_dst:B8:EE:01:01:00:88,mod_nw_dst:5.5.5.5,mod_vlan_vid:10,mod_tp_dst=1000,output:dpdk1"
```

rte_flow

模式条目

模式条目分成两类:

> 匹配协议头部及报文数据（ANY，RAW，ETH，VLAN，IPV4，IPV6，ICMP，UDP，TCP，SCTP,，VXLAN，MPLS，GRE等等）。
> 匹配元数据或影响模式处理（END，VOID，INVERT，PF，VF，PORT等等）。

常规流表举例:

index	Description
0	Eth
1	IPv4
2	TCP
3	VOID
4	END

常见的几个令人费解的匹配项

PORT:Matches packets coming from the specified physical port of the underlying device.匹配来自指定底层设备物理端口的数据包。
VF:Matches packets addressed to a virtual function ID of the device.
匹配寻址到设备虚拟功能ID的数据包。
PF:Matches packets addressed to the physical function of the device.
匹配寻址到设备物理功能的数据包。

RAW：Matches a byte string of a given length at a given offset.
在给定的偏移量处匹配给定长度的字节字符串。

其他好理解匹配项的举例如下，像eth、ipv4等不赘述：
Matches a VXLAN header (RFC 7348).

- flags: normally 0x08 (I flag).
- rsvd0: reserved, normally 0x000000.
- vni: VXLAN network identifier.
- rsvd1: reserved, normally 0x00.
- Default mask matches VNI only.

action项

主要包含以下三种：

- 终止类操作，如DROP、QUEUE、RSS、PF、VF，防止被后续流表处理。
- 非终止类操作，如PASSTHRU、DUPDUP，以供后续流表继续处理。
- 其他不影响数据包的非终止类操作，如END、VOID、MARK、FLAG、COUNT。

当多个动作组合在一起时，他们必须是不同的类型，这里还有个注意点：PASSTHRU动作是可以覆盖终止类动作的。
这里单独介绍下几个action：
mark：将整数值附加到数据包并设置PKT_RX_FDIR和PKT_RX_FDIR_ID的mbuf标志。
flag：和mark很像，但是flag只设置PKT_RX_FDIR的mbuf标志，不设置值。
DUP：复制包到给定的queue中。

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_actions/netdev_offload_dpdk_mark_rss
                        //组织pattern和action
                                |
                        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

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。
```

重点函数分析：
action信息组织

```c
static int
parse_flow_actions(struct netdev *netdev,
                   struct flow_actions *actions,
                   struct nlattr *nl_actions,
                   size_t nl_actions_len)
{
    struct nlattr *nla;
    size_t left;

    add_count_action(actions);
    NL_ATTR_FOR_EACH_UNSAFE (nla, left, nl_actions, nl_actions_len) {
        if (nl_attr_type(nla) == OVS_ACTION_ATTR_OUTPUT) {
            if (add_output_action(netdev, actions, nla)) {
                return -1;
            }
        } else if (nl_attr_type(nla) == OVS_ACTION_ATTR_DROP) {
            add_flow_action(actions, RTE_FLOW_ACTION_TYPE_DROP, NULL);
        } else if (nl_attr_type(nla) == OVS_ACTION_ATTR_SET ||
                   nl_attr_type(nla) == OVS_ACTION_ATTR_SET_MASKED) {
            const struct nlattr *set_actions = nl_attr_get(nla);
            const size_t set_actions_len = nl_attr_get_size(nla);
```

```
            bool masked = nl_attr_type(nla) == OVS_ACTION_ATTR_SET_MASKED;

            if (parse_set_actions(actions, set_actions, set_actions_len,
                                  masked)) {
                return -1;
            }
        } else if (nl_attr_type(nla) == OVS_ACTION_ATTR_PUSH_VLAN) {
            const struct ovs_action_push_vlan *vlan = nl_attr_get(nla);

            if (parse_vlan_push_action(actions, vlan)) {
                return -1;
            }
        ............................
    }

    if (nl_actions_len == 0) {
        VLOG_DBG_RL(&rl, "No actions provided");
        return -1;
    }

    add_flow_action(actions, RTE_FLOW_ACTION_TYPE_END, NULL);
    return 0;
}

```

pattern信息组织：

```
static int
parse_flow_match(struct netdev *netdev,
                 odp_port_t orig_in_port OVS_UNUSED,
                 struct flow_patterns *patterns,
                 struct match *match)
{
    struct flow *consumed_masks;
    uint8_t proto = 0;

    consumed_masks = &match->wc.masks;

    if (!flow_tnl_dst_is_set(&match->flow.tunnel)) {
        memset(&consumed_masks->tunnel, 0, sizeof consumed_masks->tunnel);
    }

    patterns->physdev = netdev;
    memset(&consumed_masks->in_port, 0, sizeof consumed_masks->in_port);
    /* recirc id must be zero. */
    if (match->wc.masks.recirc_id & match->flow.recirc_id) {
        return -1;
    }
    consumed_masks->recirc_id = 0;
    consumed_masks->packet_type = 0;

    /* Eth */
    if (match->wc.masks.dl_type ||
        .........................................

        add_flow_pattern(patterns, RTE_FLOW_ITEM_TYPE_ETH, spec, mask);
    }
```

```

    /* VLAN */
    if (match->wc.masks.vlans[0].tci && match->flow.vlans[0].tci) {
        .........................................

        add_flow_pattern(patterns, RTE_FLOW_ITEM_TYPE_VLAN, spec, mask);
    }
    /* For untagged matching match->wc.masks.vlans[0].tci is 0xFFFF and
     * match->flow.vlans[0].tci is 0. Consuming is needed outside of the if
     * scope to handle that.
     */
    memset(&consumed_masks->vlans[0], 0, sizeof consumed_masks->vlans[0]);

    /* IP v4 */
    if (match->flow.dl_type == htons(ETH_TYPE_IP)) {
        .........................................
    }
    /* If fragmented, then don't HW accelerate - for now. */
    if (match->wc.masks.nw_frag & match->flow.nw_frag) {
        return -1;
    }
    consumed_masks->nw_frag = 0;

    /* IP v6 */
    if (match->flow.dl_type == htons(ETH_TYPE_IPV6)) {
`       .........................................
        add_flow_pattern(patterns, RTE_FLOW_ITEM_TYPE_IPV6, spec, mask);

        /* Save proto for L4 protocol setup. */
        proto = spec->hdr.proto & mask->hdr.proto;
    }

    if (proto == IPPROTO_TCP) {
        .........................................
        add_flow_pattern(patterns, RTE_FLOW_ITEM_TYPE_TCP, spec, mask);
    } else if (proto == IPPROTO_UDP) {
        .........................................
        add_flow_pattern(patterns, RTE_FLOW_ITEM_TYPE_UDP, spec, mask);
    } else if (proto == IPPROTO_SCTP) {
        .........................................
        add_flow_pattern(patterns, RTE_FLOW_ITEM_TYPE_SCTP, spec, mask);
    } else if (proto == IPPROTO_ICMP) {
            .........................................
    }

    add_flow_pattern(patterns, RTE_FLOW_ITEM_TYPE_END, NULL, NULL);

    if (!is_all_zeros(consumed_masks, sizeof *consumed_masks)) {
        return -1;
    }
    return 0;
}
```

在下发到pmd流程结束后，调用函数`ufid_to_rte_flow_associate`,建立ufid到rte_flow的映射关系，`cmap_insert(&ufid_to_rte_flow,
                CONST_CAST(struct cmap_node *, &data->node), hash);`。

rte_flow在pmd中的大致流程

```
                    函数入口
                       |
                 解析pattern组织成流表key A
                       |
                  解析action组织成流表act B
                       |
        根据解析的pattern确定使用那种key-template, 如template c
                       |
        存储流表key A到template c的 c-hash-list
                       |
        存储流表act B到action专门的 act-hash-list
                       |
  如果带隧道, 下发隧道的hash-list和其action为metadata的ac-hash-list
                       |
        组织key与act为逻辑要求的格式下发软硬件通道

```

EAL初始化

核心函数分析:

```
/* Launch threads, called at application init(). */
int
rte_eal_init(int argc, char **argv)
{
        ...
        /* rte_eal_cpu_init() ->
                *     eal_cpu_core_id()
                *     eal_cpu_socket_id()
                * 读取/sys/devices/system/[cpu|node]
                * 设置lcore_config->[core_role|core_id|socket_id] */
        if (rte_eal_cpu_init() < 0) {
                rte_eal_init_alert("Cannot detect lcores.");
                rte_errno = ENOTSUP;
                return -1;
        }

        /* eal_parse_args() ->
                *     eal_parse_common_option() ->
                *         eal_parse_coremask()
                *         eal_parse_master_lcore()
                *         eal_parse_lcores()
                *     eal_adjust_config()
                * 解析-c、--master_lcore、--lcores参数
                * 在eal_parse_lcores()中确认可用的logical CPU
                * 在eal_adjust_config()中设置rte_config.master_lcore为0 (设置第一个lcore为MASTER lcore) */
        fctret = eal_parse_args(argc, argv);
        if (fctret < 0) {
                rte_eal_init_alert("Invalid 'command line' arguments.");
                rte_errno = EINVAL;
                rte_atomic32_clear(&run_once);
                return -1;
```

```
        }
        ...
        /* 初始化大页信息 */
        if (rte_eal_memory_init() < 0) {
                rte_eal_init_alert("Cannot init memory\n");
                rte_errno = ENOMEM;
                return -1;
        }
        ...
        /* eal_thread_init_master() ->
                *     eal_thread_set_affinity()
                * 设置当前线程为MASTER lcore
                * 在eal_thread_set_affinity()中绑定MASTER lcore到logical CPU */
        eal_thread_init_master(rte_config.master_lcore);
        ...
        /* rte_bus_scan() ->
                *     rte_pci_scan() ->
                *         pci_scan_one() ->
                *             pci_parse_sysfs_resource()
                *             rte_pci_add_device()
                * 遍历rte_bus_list链表，调用每个bus的scan函数，pci为rte_pci_scan()
                * 遍历/sys/bus/pci/devices目录，为每个DBSF分配struct rte_pci_device
                * 逐行读取并解析每个DBSF的resource，保存到dev->mem_resource[i]
                * 将dev插入rte_pci_bus.device_list链表 */
        if (rte_bus_scan()) {
                rte_eal_init_alert("Cannot scan the buses for devices\n");
                rte_errno = ENODEV;
                return -1;
        }

        /* pthread_create() ->
                *     eal_thread_loop() ->
                *         eal_thread_set_affinity()
                * 为每个SLAVE lcore创建线程，线程函数为eal_thread_loop()
                * 在eal_thread_set_affinity()中绑定SLAVE lcore到logical CPU */
        RTE_LCORE_FOREACH_SLAVE(i) {

                /*
                        * create communication pipes between master thread
                        * and children
                        */
                /* MASTER lcore创建pipes用于MASTER和SLAVE lcore间通信（父子线程间通信） */
                if (pipe(lcore_config[i].pipe_master2slave) < 0)
                        rte_panic("Cannot create pipe\n");
                if (pipe(lcore_config[i].pipe_slave2master) < 0)
                        rte_panic("Cannot create pipe\n");

                lcore_config[i].state = WAIT; /* 设置SLAVE lcore的状态为WAIT */

                /* create a thread for each lcore */
                ret = pthread_create(&lcore_config[i].thread_id, NULL,
                                        eal_thread_loop, NULL);
```

```
                ...
        }
        /*函数用于载入业务函数。函数流程如下:

        检查所有的副线程，是否都在WAIT状态。
        1.1 如果不是所有的副线程都在WAIT状态，则返回-EBUSY，跳出程序。
        1.2 如果所有的副线程都在WAIT状态，进行后续的步骤。
        遍历所有的副线程。
        2.1. 调用rte_eal_remote_launch()，为各个副线程载入业务函数f，并通知副线程执行。
        主线程按需执行业务函数f。
        3.1. 如果参数call_master设置为CALL_MASTER，则主线程需要执行业务函数f。
        3.2. 如果参数call_master设置为SKIP_MASTER，则主线程不用执行业务函数f。*/
        rte_eal_mp_remote_launch(sync_func, NULL, SKIP_MASTER);

        /* 用于等待所有副线程返回 */
        rte_eal_mp_wait_lcore();
        ...
        /* Probe all the buses and devices/drivers on them */
        /* rte_bus_probe() ->
     *     rte_pci_probe() ->
     *         pci_probe_all_drivers() ->
     *             rte_pci_probe_one_driver() ->
     *                 rte_pci_match()
     *                 rte_pci_map_device() ->
     *                     pci_uio_map_resource()
     *                 eth_ixgbe_pci_probe()
     * 遍历rte_bus_list链表，调用每个bus的probe函数，pci为rte_pci_probe()
     * rte_pci_probe()/pci_probe_all_drivers()分别遍历rte_pci_bus.device_list/driver_list链表，匹配设备和驱动
     * 映射BAR，调用驱动的probe函数，ixgbe为eth_ixgbe_pci_probe() */
        if (rte_bus_probe()) {
                rte_eal_init_alert("Cannot probe devices\n");
                rte_errno = ENOTSUP;
                return -1;
        }
        ...
}
```

rte以太网设备配置流程分析

`struct rte_eth_dev *dev = &rte_eth_devices[port_id];`
驱动probe阶段识别到网卡后，层层深入调用到rte_eth_dev_allocate，从rte_eth_devices数组中分配一个未使用的结构
后续API中的port_id其实访问的就是rte_eth_devices[port_id]

核心对象

port
端口对象，例如一个pcie网卡

rx_queue/tx_queue
端口收发队列对象
多核环境下，端口收到包后可指定响应的cpu来处理这个包。
通过增加收发队列，根据五元组哈希分配处理的core，实现计算资源的初步负载均衡

每个端口进来的包通过rss模块计算hash后，发送到对应cpu的queue上等待处理

tx_desc/rx_desc
网卡驱动中收发dma的队列数量。
收发desc中描述了dma收发需要的信息，如源/目的地址、长度等

核心接口

函数	功能
rte_eth_dev_count()	网卡数
rte_eth_dev_configure()	配置网卡
rte_eth_rx_queue_setup()	为网卡分配接收队列
rte_eth_tx_queue_setup()	为网卡分配发送队列
rte_eth_rx_burst()	网卡收包函数
rte_eth_tx_burst()	网卡发包函数
rte_eth_dev_start()	启动网卡

核心流程

```
                rte_pktmbuf_pool_create
                   创建mbuf_pool
                        |
                    port_init
                   端口设置队列数
                        |
                rte_eth_dev_configure
                        |---rte_eth_dev_rx_queue_config
        如果是第一次配置，那么就为每个发包队列分配一个指针。
        如果是重新配置，而且新的队列数量不为0，那么就释放老的队列
        如果是重新配置，而且新的队列数量为0，那么就释放所有的队列。
                        |---rte_eth_dev_tx_queue_config
                        |---nbl_repr_dev_configure
        配置队列的个数以及接口的配置信息（分配队列指针空间）
                        |
                rte_eth_rx_queue_setup
                        |---nbl_rx_queue_setup
        使用mbuf_pool数据初始化接收队列，分配desc空间
                        |
                rte_eth_tx_queue_setup
                设置以太网设备 的发包队列
                        |---nbl_tx_queue_setup
                            网口的发包队列的初始化
                        |
                rte_eth_dev_start
                        |---nbl_dev_start
                        |---nbl_tx_start
                        |---nbl_rx_start
1、启动设备，为每个队列设置dma寄存器，标识每个队列的描述符ring的地址和长度
2、启动设备的收发单元（设置寄存器，mbuf设置为网卡收包的dma地址，记录在desc上 启动引擎）
3、混杂、组播、链路状态更新
```

```
                           |
                        lcore_main
                     rx_burst/tx_burst收发包
                           |---nbl_repr_rx_burst
                           |---nbl_repr_tx_burst
```