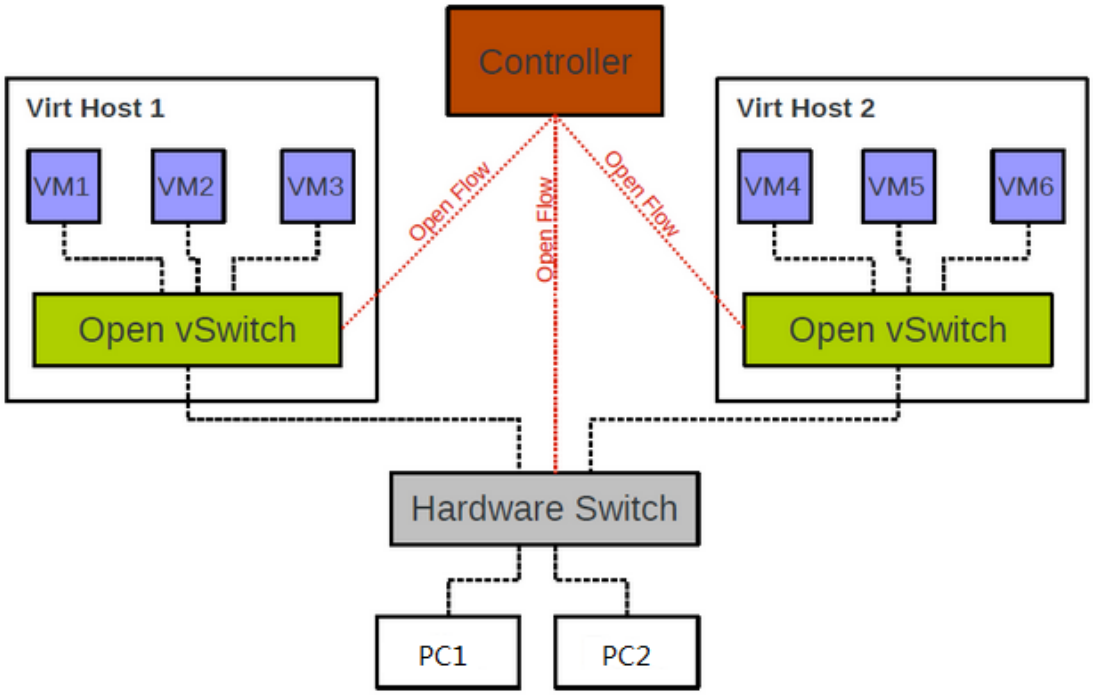
Open vSwitch 源码阅读笔记

引言

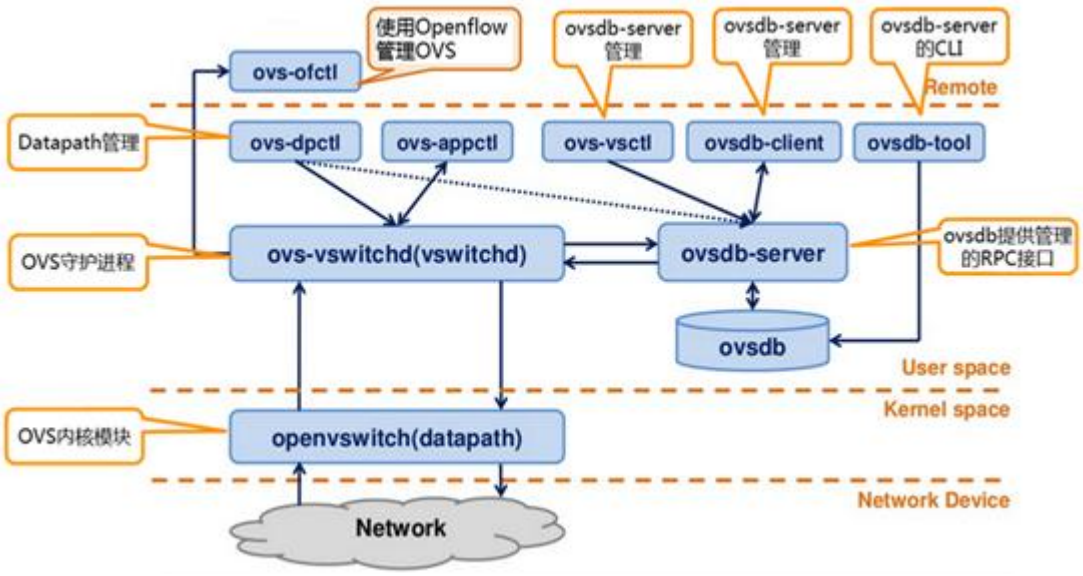
本文主要对 OpenvSwitch（基于 2.3.90 版本）重点模块的源码实现流程做了简要的阅读记录，适合阅读 OpenvSwitch 源码的初级读者参考使用，任何错误和建议欢迎加作者 QQ 号 38293996 沟通交流。

1. OVS 网络架构

Openvswitch 是一个虚拟交换机，支持 Open Flow 协议（也有一些硬件交换机支持 Open Flow），他们被远端的 controller 通过 Open Flow 协议统一管理着，从而实现对接入的虚拟机（或设备）进行组网和互通，整体组网结构如下图：

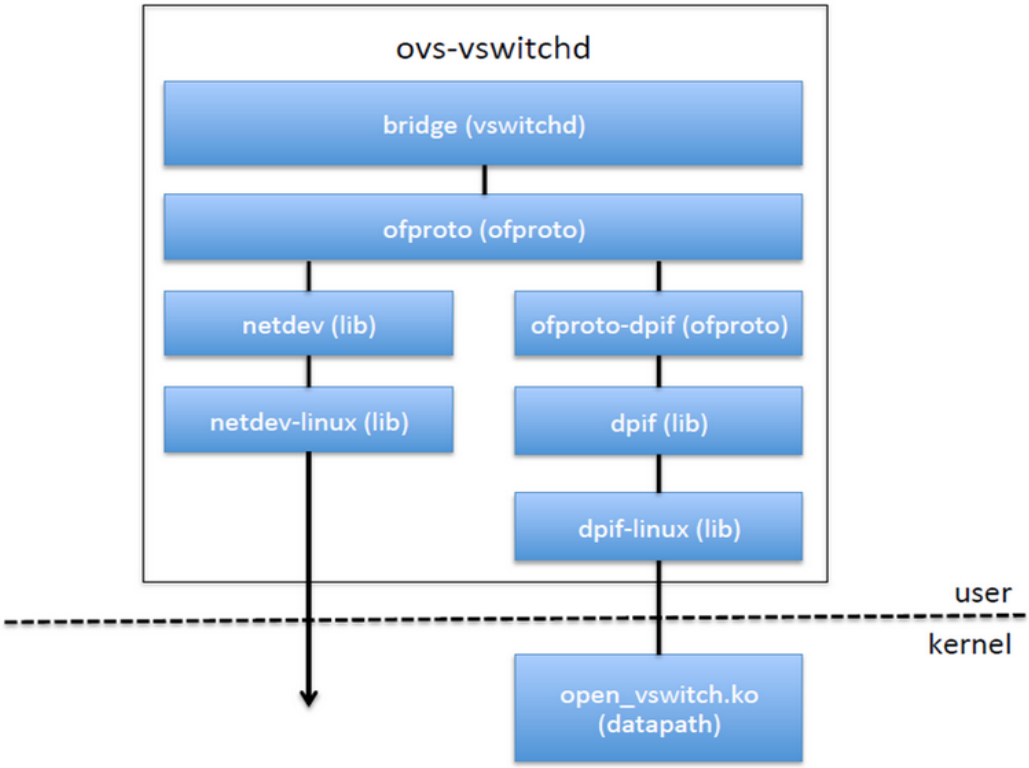


2. OVS 内部架构



- ovs-vswitchd 主要模块，实现 vswitch 的守候进程 daemon；
- ovsdb-server 轻量级数据库服务器,用于 ovs 的配置信息；
- ovs-vsctl 通过和 ovsdb-server 通信，查询和更新 vswitch 的配置；
- ovs-dpctl 用来配置 vswitch 内核模块的一个工具；
- ovs-appctl 发送命令消息到 ovs 进程；
- ovs-ofctl 查询和控制 OpenFlow 虚拟交换机的流表；
- datapath 内核模块，根据流表匹配结果做相应处理；

3. OVS 代码架构



- **vswitchd** 是 ovs 主要的用户态程序，它从 ovsdb-server 读取配置并发送到 ofproto 层，也从 ofproto 读取特定的状态和统计信息并发送到数据库；

- **ofproto** 是 openflow 的接口层，负责和 Openflow controller 通信并通过 ofproto_class 与 ofproto provider 底层交互；

- **ofproto-dpif** 是 ofproto 接口类的具体实现；

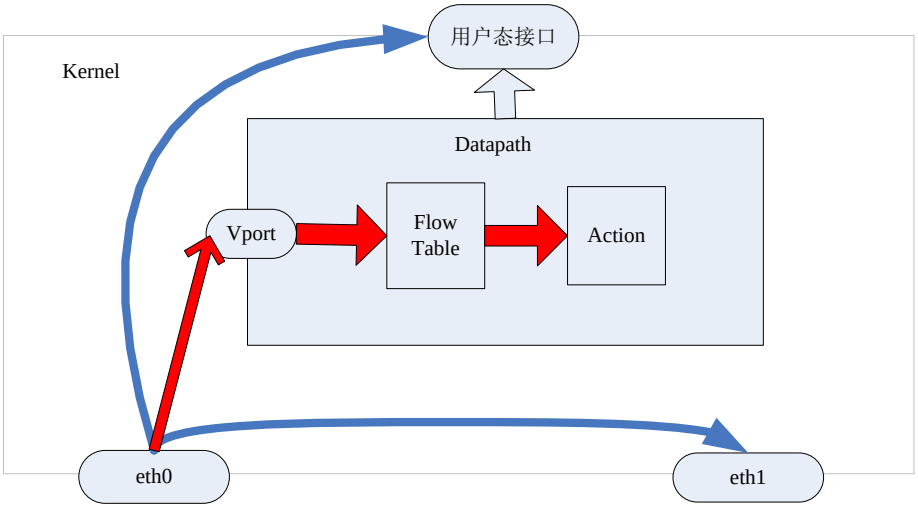
- **netdev** 是 ovs 系统的网络设备抽象（比如 linux 的 net_device 或交换机的 port），netdev_class 定义了 netdev-provider 的具体实现需要的接口，具体的平台实现需要支持这些统一的接口，从而完成 netdev 设备的创建、销毁、打开、关闭等一系列操作；

3.1 datapath

由于 openvswitch 用户态代码相对复杂，首先从内核模块入手分析。

datapath 为 ovs 内核模块，负责执行数据处理，也就是把从接收端口收到的数据包在流表中进行匹配，并执行匹配到的动作。一个 datapath 可以对应多个 vport，一个 vport 类似物理交换机的端口概念。一个 datapth 关联一个 flow table，一个 flow table 包含多个条目，每个条目包括两个内容：一个 match/key 和一个 action。

3.1.1 数据流向



一般的数据包在 Linux 网络协议中的流向为上图中的蓝色箭头流向：网卡 eth0 收到数据包后判断报文走向，如果是本地报文把数据传送到用户态，如果是转发报文根据选路（二层交换或三层路由）把报文送到另一个网卡如 eth1。当有 OVS 时，数据流向如红色所示：从网卡 eth0 收到报文后进入 ovs 的端口，根据 key 值进行流表匹配，如果匹配成功执行流表对应的 action；如果失败通过 upcall 送入用户态处理。

3.1.2 模块初始化

内核模块采用 module_init(dp_init)进行 datapath 的初始化，代码如下：

```
/* 初始化action_fifos空间 */
err = action_fifos_init();
if (err)
    goto error;

/* 注册ovs的internal设备的rtnl */
err = ovs_internal_dev_rtnl_link_register();
if (err)
    goto error_action_fifos_exit;

/* 初始化flow模块，分配相关cache */
err = ovs_flow_init();
if (err)
    goto error_unreg_rtnl_link;

/* 初始化vport，分配hash节点空间 */
err = ovs_vport_init();
if (err)
    goto error_flow_exit;

/* 注册ovs网络空间类型设备 */
err = register_pernet_device(&ovs_net_ops);
if (err)
    goto error_vport_exit;

/* 注册dp的internal类型设备的通知链 */
err = register_netdevice_notifier(&ovs_dp_device_notifier);
if (err)
    goto error_netns_exit;

/* 注册dp的netlink family*/
err = dp_register_genl();
if (err < 0)
    goto error_unreg_notifier;
```

其中 dp 的 genl_family 注册了如下四个类型：

```
static struct genl_family * const dp_genl_families[] = {
    &dp_datapath_genl_family,
    &dp_vport_genl_family,
    &dp_flow_genl_family,
    &dp_packet_genl_family,
};
```

3.1.3 收包处理

通过 vport 注册的回调函数 netdev_frame_hook()->netdev_frame_hook()->netdev_port_receive()->ovs_vport_receive()处理接收报文，ovs_flow_key_extract()函数生成 flow 的 key 内容用以接下来进行流表匹配，最后调用 ovs_dp_process_packet()函数进入真正的 ovs 数据包处理,代码流程如下：

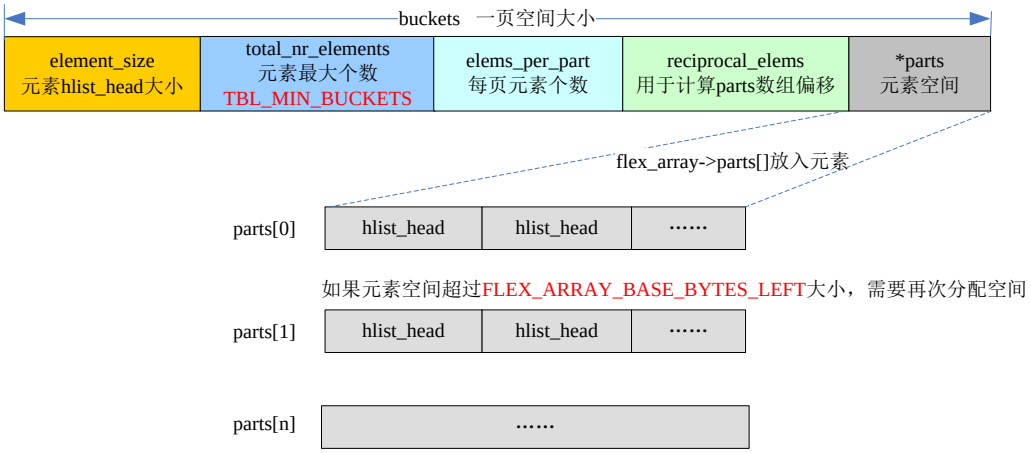
```
    stats = this_cpu_ptr(dp->stats_percpu);
    /* Look up flow. */
    flow = ovs_flow_tbl_lookup_stats(&dp->table, key, skb_get_hash(skb),
                     &n_mask_hit);/* 根据mask和key进行匹配查找*/
    if (unlikely(!flow)) {/* 没有找到flow，需要发送到用户态进行慢速匹配 */
        struct dp_upcall_info upcall;
        int error;

        upcall.cmd = OVS_PACKET_CMD_MISS;
        upcall.userdata = NULL;
        upcall.portid = ovs_vport_find_upcall_portid(p, skb);
        upcall.egress_tun_info = NULL;
        error = ovs_dp_upcall(dp, skb, key, &upcall);/* 发送到用户态 */
        if (unlikely(error))
            kfree_skb(skb);
        else
            consume_skb(skb);
        stats_counter = &stats->n_missed;/* 未匹配的包数 */
        goto out;
    }
    ovs_flow_stats_update(flow, key->tp.flags, skb);/* 更新流表状态信息 */
    sf_acts = rcu_dereference(flow->sf_acts);
    ovs_execute_actions(dp, skb, sf_acts, key);/* 执行action */

    stats_counter = &stats->n_hit;/* 匹配的包数 */

out:
    /* Update datapath statistics. */
    u64_stats_update_begin(&stats->syncp);
    (*stats_counter)++;//收包总数
    stats->n_mask_hit += n_mask_hit;//流表查询次数
    u64_stats_update_end(&stats->syncp);
```

3.1.3 流表哈希桶

流表采用 hash 的方式排列存放，流表的 hash 头结点存储数据结构如下：

```
struct flex_array {
    union {
        struct {
            int element_size;//每个元素大小
            int total_nr_elements;//元素总个数
            int elems_per_part;//每个part包含的元素个数
            struct reciprocal_value reciprocal_elems;
            /* 当元素占用的空间(element_size * total_nr_elements)
               大于FLEX_ARRAY_BASE_BYTES_LEFT(一页减去上面字段占用的空间大小)时，
               parts包含多个分页空间，否则只包含一页空间即可
            */
            struct flex_array_part *parts[];
        };
        /*
         * This little trick makes sure that
         * sizeof(flex_array) == PAGE_SIZE
         */
        char padding[FLEX_ARRAY_BASE_SIZE];//一个页的大小PAGE_SIZE
    };
};
```

该 hash 桶的初始化函数 alloc_buckets ()，生成的数据格式可参考如下：

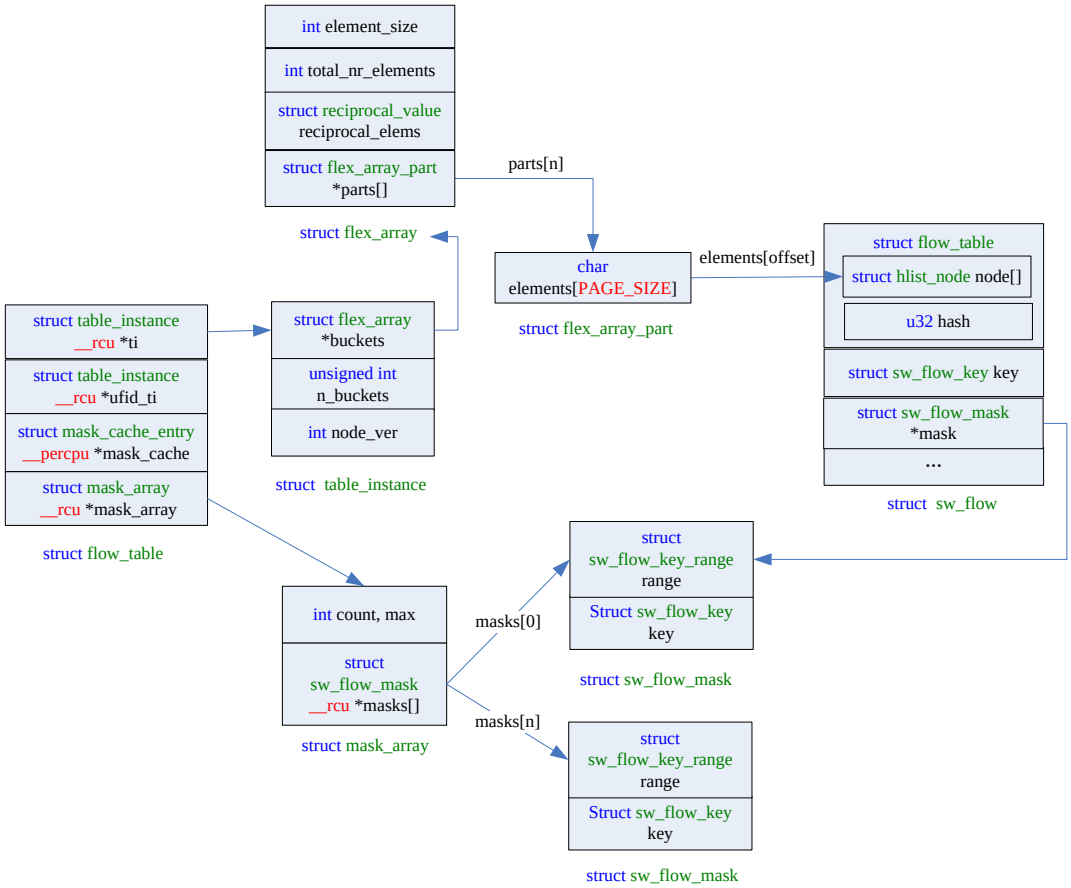


3.1.4 流表创建

用户态通过 netlink 进行 datapath 流表更新的入口函数都定义在 dp_flow_genl_ops 中，流表创建的入口函数是 ovs_flow_cmd_new 函数，代码分析如下：

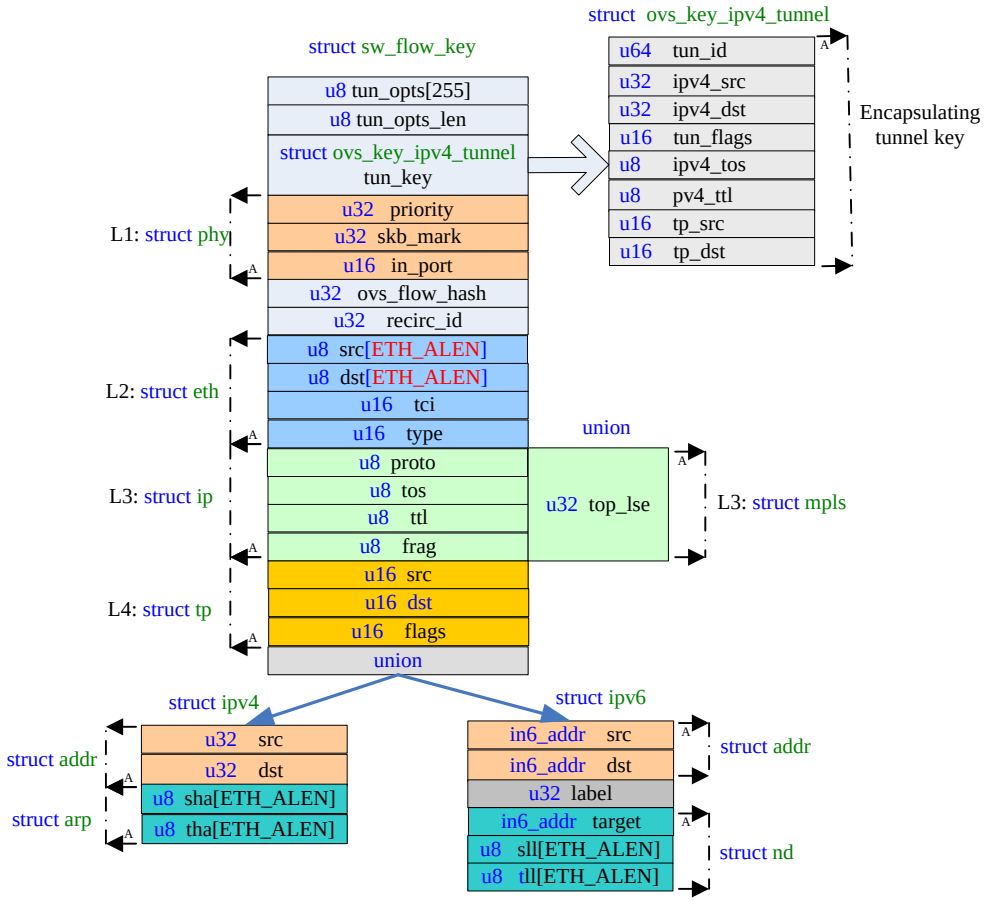
```
/* Most of the time we need to allocate a new flow, do it before locking.*/
new_flow = ovs_flow_alloc();//分配sw_flow并初始化
if (IS_ERR(new_flow)) {
    error = PTR_ERR(new_flow);
    goto error;
}
/* Extract key. */
ovs_match_init(&match, &key, &mask);
/* 解析key和mask，放入sw_flow_match中 */
error = ovs_nla_get_match(&match, a[OVS_FLOW_ATTR_KEY],
              a[OVS_FLOW_ATTR_MASK], log);
if (error)
    goto err_kfree_flow;
/* 根据key和mask->range重新生成掩码后的key放入new_flow->key*/
ovs_flow_mask_key(&new_flow->key, &key, &mask);
/* Extract flow identifier. */
error = ovs_nla_get_identifier(&new_flow->id, a[OVS_FLOW_ATTR_UFID],
                   &key, log);
if (error)
    goto err_kfree_flow;
/* Validate actions. 获取action并放入sw_flow_actions*/
error = ovs_nla_copy_actions(a[OVS_FLOW_ATTR_ACTIONS], &new_flow->key,
                 &acts, log);
if (error) {
    OVS_NLERR(log, "Flow actions may not be safe on all matching packets.");
    goto err_kfree_flow;
}
/* 分配发往用户空间的数据区*/
reply = ovs_flow_cmd_alloc_info(acts, &new_flow->id, info, false,
                ufid_flags);
if (IS_ERR(reply)) {
    error = PTR_ERR(reply);
    goto err_kfree_acts;
}

ovs_lock();
/* 根据dp索引获取dp */
dp = get_dp(sock_net(skb->sk), ovs_header->dp_ifindex);
if (unlikely(!dp)) {
    error = -ENODEV;
    goto err_unlock_ovs;
}
/* Check if this is a duplicate flow 检查flow是否存在*/
if (ovs_identifier_is_ufid(&new_flow->id))
    flow = ovs_flow_tbl_lookup_ufid(&dp->table, &new_flow->id);
if (!flow)
    flow = ovs_flow_tbl_lookup(&dp->table, &key);
if (likely(!flow)) {
    rcu_assign_pointer(new_flow->sf_acts, acts);

    /* Put flow in bucket. */
    /*1) mask插入到table_instance->mask_array
      2) flow插入到table_instance->ti
      3) flow插入到table_instance->ufid_ti*/
    error = ovs_flow_tbl_insert(&dp->table, new_flow, &mask);
    if (unlikely(error)) {
        acts = NULL;
        goto err_unlock_ovs;
    }
```

根据上述流程给出流表的主要数据结构如下：



3.1.6 流表查询

流表查找主要是查表关键字的匹配，关键字数据结构如下，根据 skb 中的 Ethernet 帧生成 key 的函数为 ovs_flow_key_extract()：



流表查询的入口函数 ovs_flow_tbl_lookup_stats(), flow 的匹配策略是和流表中所有 mask 和所有 key 进行匹配处理，为了加速查询效率，在调用真正的流表查询函数 flow_lookup()之前，对于 mask 的查询采用了缓存机制，实现原理是首先查询缓存的 mask_cache_entry，这些 cache 是查询成功后形成的 cache，并针对 cache 采用分段查询的方式，代码如下：

```
ce = NULL;
hash = skb_hash;
/* mask_cache 存放匹配命中的cache，该指针空间在ovs_flow_tbl_init()初始化
   一共分配了MC_HASH_ENTRIES(256)个mask_cache_entry对象的空间*/
entries = this_cpu_ptr(tbl->mask_cache);
/* Find the cache entry 'ce' to operate on. */
/* mask_cache的hash链表被分成MC_HASH_SEGS(4)部分以实现更快速的查找*/
for (seg = 0; seg < MC_HASH_SEGS; seg++) {
    int index = hash & (MC_HASH_ENTRIES - 1);//取hash的最低一个字节值
    struct mask_cache_entry *e;
    e = &entries[index];//获取entries指向的第index-1个mask_cache_entry地址值
    if (e->skb_hash == skb_hash) {//通过skb_hash作为唯一索引
        flow = flow_lookup(tbl, ti, ma, key, n_mask_hit,
                   &e->mask_index);
        if (!flow)//flow没有找到，当前cache无效
            e->skb_hash = 0;
        return flow;
    }
    /* 因为hash是从地位到高位分4段搜索的，所以hash值小优先级高*/
    if (!ce || e->skb_hash < ce->skb_hash)
        ce = e;  /* A better replacement cache candidate. */
    /* hash是4个字节，每个字节单独作为索引使用 */
    hash >>= MC_HASH_SHIFT;
}
/* Cache miss, do full lookup. */
flow = flow_lookup(tbl, ti, ma, key, n_mask_hit, &ce->mask_index);
if (flow)//流找到，更新ce，以便下次cache快速查找
    ce->skb_hash = skb_hash;

return flow;
```

flow_lookup()函数的处理流程如下：

```
static struct sw_flow *flow_lookup(struct flow_table *tbl,
                   struct table_instance *ti,
                   const struct mask_array *ma,
                   const struct sw_flow_key *key,
                   u32 *n_mask_hit,
                   u32 *index)
{
    struct sw_flow_mask *mask;
    struct sw_flow *flow;
    int i;

    if (*index < ma->max) {/* index有效性判断 */
        mask = rcu_dereference_ovsl(ma->masks[*index]);
        if (mask) {/* cache的mask，针对该mask进行key匹配 */
            flow = masked_flow_lookup(ti, key, mask, n_mask_hit);
            if (flow)
                return flow;
        }
    }
    /* 1、当前的cache mask没有匹配到flow，index指向找到的cache
       2、cache mask没有找到，index默认为0
       上述情况会进入下面的全局所有匹配*/
    for (i = 0; i < ma->max; i++)  {

        if (i == *index)/*index指向的cache上面已经匹配过，直接跳过*/
            continue;
        mask = rcu_dereference_ovsl(ma->masks[i]);
        if (!mask)
            continue;
        flow = masked_flow_lookup(ti, key, mask, n_mask_hit);
        if (flow) { /* Found */
            *index = i;/* 生成cache的mask_index */
            return flow;
        }
    }
    return NULL;
}
```

masked_flow_lookup()函数处理如下：

```
static struct sw_flow *masked_flow_lookup(struct table_instance *ti,
                      const struct sw_flow_key *unmasked,
                      const struct sw_flow_mask *mask,
                      u32 *n_mask_hit)
{
    struct sw_flow *flow;
    struct hlist_head *head;
    u32 hash;
    struct sw_flow_key masked_key;
    /* unmasked为数据包提取的key，和mask进行与运算生成masked_key，
       与运算的范围是range.start到range.end*/
    ovs_flow_mask_key(&masked_key, unmasked, mask);
    hash = flow_hash(&masked_key, &mask->range); /* hash计算 */
    head = find_bucket(ti, hash);/* 根据hash获取sw_flow在表中的hash链表节点头*/
    (*n_mask_hit)++;
    /* 遍历hash链表中的flow节点并进行匹配检测 */
    hlist_for_each_entry_rcu(flow, head, flow_table.node[ti->node_ver]) {
        if (flow->mask == mask && flow->flow_table.hash == hash &&
            flow_cmp_masked_key(flow, &masked_key, &mask->range))
            return flow;
    }
    return NULL;
}
```

3.1.7 action 处理

ovs 的 action 类型如下，使用 nla_type()函数获取 nl_type 的值，入口处理函数为 do_execute_actions()。

```
enum ovs_action_attr {
    OVS_ACTION_ATTR_UNSPEC,
    OVS_ACTION_ATTR_OUTPUT,       /* u32 port number. */
    OVS_ACTION_ATTR_USERSPACE,    /* Nested OVS_USERSPACE_ATTR_*. */
    OVS_ACTION_ATTR_SET,          /* One nested OVS_KEY_ATTR_*. */
    OVS_ACTION_ATTR_PUSH_VLAN,    /* struct ovs_action_push_vlan. */
    OVS_ACTION_ATTR_POP_VLAN,     /* No argument. */
    OVS_ACTION_ATTR_SAMPLE,       /* Nested OVS_SAMPLE_ATTR_*. */
    OVS_ACTION_ATTR_RECIRC,       /* u32 recirc_id. */
    OVS_ACTION_ATTR_HASH,         /* struct ovs_action_hash. */
    OVS_ACTION_ATTR_PUSH_MPLS,    /* struct ovs_action_push_mpls. */
    OVS_ACTION_ATTR_POP_MPLS,     /* __be16 ethertype. */
    OVS_ACTION_ATTR_SET_MASKED,   /* One nested OVS_KEY_ATTR_* including
                                   * data immediately followed by a mask.
                                   * The data must be zero for the unmasked
                                   * bits. */

#ifndef __KERNEL__
    OVS_ACTION_ATTR_TUNNEL_PUSH,   /* struct ovs_action_push_tnl*/
    OVS_ACTION_ATTR_TUNNEL_POP,    /* u32 port number. */
#endif
    __OVS_ACTION_ATTR_MAX
};
```

- OVS_ACTION_ATTR_OUTPUT：获取 port 号，调用 do_output()发送报文到该 port；
- OVS_ACTION_ATTR_USERSPACE：调用 output_userspace()发送到用户态；
- OVS_ACTION_ATTR_HASH：调用 execute_hash()获取 skb 的 hash 赋值到 ovs_flow_hash
- OVS_ACTION_ATTR_PUSH_VLAN：调用 push_vlan()增加 vlan 头部

```
int skb_vlan_push(struct sk_buff *skb, __be16 vlan_proto, u16 vlan_tci)
{
    if (skb_vlan_tag_present(skb)) {/* CFI等于0是以太网，1是令牌环 */
        unsigned int offset = skb->data - skb_mac_header(skb);/* mac头长度 */
        int err;

        /* __vlan_insert_tag expect skb->data pointing to mac header.
         * So change skb->data before calling it and change back to
         * original position later
         */
        __skb_push(skb, offset);/* data指向mac头*/
        err = __vlan_insert_tag(skb, skb->vlan_proto,
                skb_vlan_tag_get(skb));/* packet增加vlan头*/
        if (err)
            return err;
        skb->mac_len += VLAN_HLEN;
        __skb_pull(skb, offset);

        if (skb->ip_summed == CHECKSUM_COMPLETE)/*data改变，重新计算校验和*/
            skb->csum = csum_add(skb->csum, csum_partial(skb->data
                    + (2 * ETH_ALEN), VLAN_HLEN, 0));
    }
    /* 增加vlan_proto和vlan_tci到skb结构中 */
    __vlan_hwaccel_put_tag(skb, vlan_proto, vlan_tci);
    return 0;
}
```

- OVS_ACTION_ATTR_POP_VLAN：调用 pop_vlan()移除 vlan 头

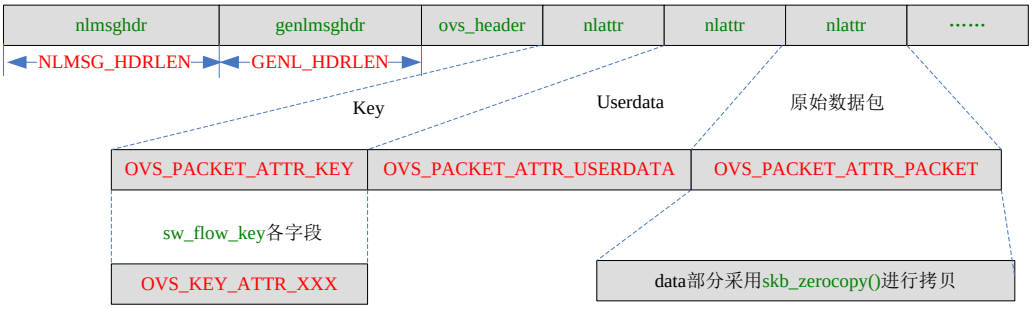
```
int skb_vlan_pop(struct sk_buff *skb)
{
    u16 vlan_tci;
    __be16 vlan_proto;
    int err;

    if (likely(skb_vlan_tag_present(skb))) {/* FDDI、令牌环等网络*/
        skb->vlan_tci = 0;
    } else {/* 以太网 */
        if (unlikely((skb->protocol != htons(ETH_P_8021Q) &&
                    skb->protocol != htons(ETH_P_8021AD)) ||
                skb->len < VLAN_ETH_HLEN))
            return 0;
        /* 移除packet中的vlan头，更新skb中的protocol */
        err = __skb_vlan_pop(skb, &vlan_tci);
        if (err)
            return err;
    }
    /* move next vlan tag to hw accel tag */
    if (likely((skb->protocol != htons(ETH_P_8021Q) &&
            skb->protocol != htons(ETH_P_8021AD)) ||
           skb->len < VLAN_ETH_HLEN))
        return 0;
    /* 双vlan的情况 */
    vlan_proto = htons(ETH_P_8021Q);
    err = __skb_vlan_pop(skb, &vlan_tci);
    if (unlikely(err))
        return err;

    __vlan_hwaccel_put_tag(skb, vlan_proto, vlan_tci);
    return 0;
}
```

- OVS_ACTION_ATTR_RECIRC:在 action_fifos 全局数组中添加一个 deferred_action；

- OVS_ACTION_ATTR_SET:调用 execute_set_action()设置相关参数；
- OVS_ACTION_ATTR_SAMPLE:概率性的发送报文到用户态（详见 sflow 章节）。

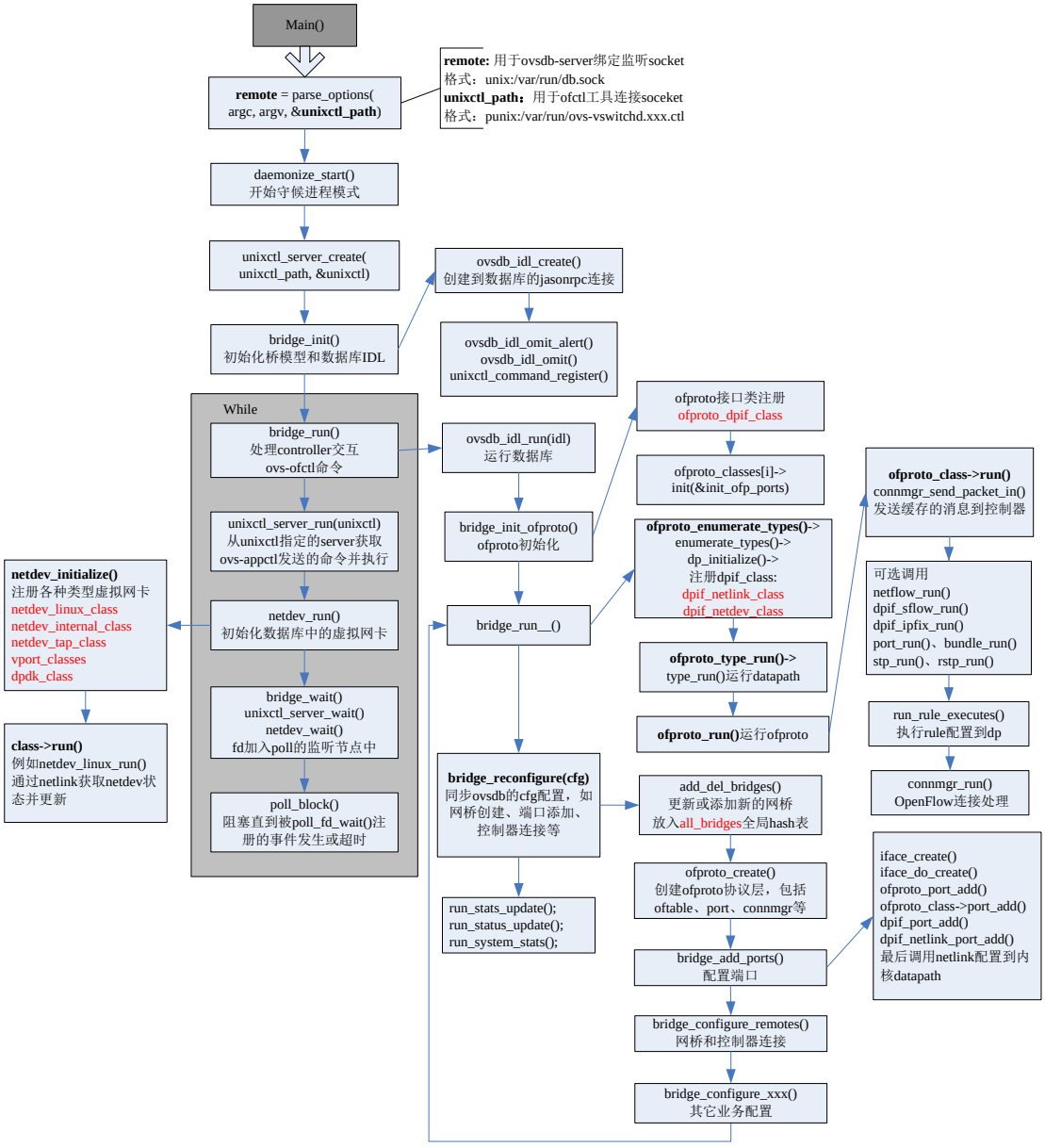
3.1.8 upcall 处理

当没有找到匹配的流表时，内核通过 netlink 发送报文到用户层处理，入口函数 ovs_dp_upcall()，该函数调用 queue_userspace_packet()构造发往用户层的 skb，通过 netlink 通信机制发送到用户层，其中形成的主要数据格式如下：



3.2 ovs-vswitchd

vswitchd 作为守护进程和 ovsdb 通信以及和 controller 进行 openflow 通信，并完成和底层内核的交互。代码在 vswitchd/目录下面，可以从 main 函数入口分析，整体处理流程如下：

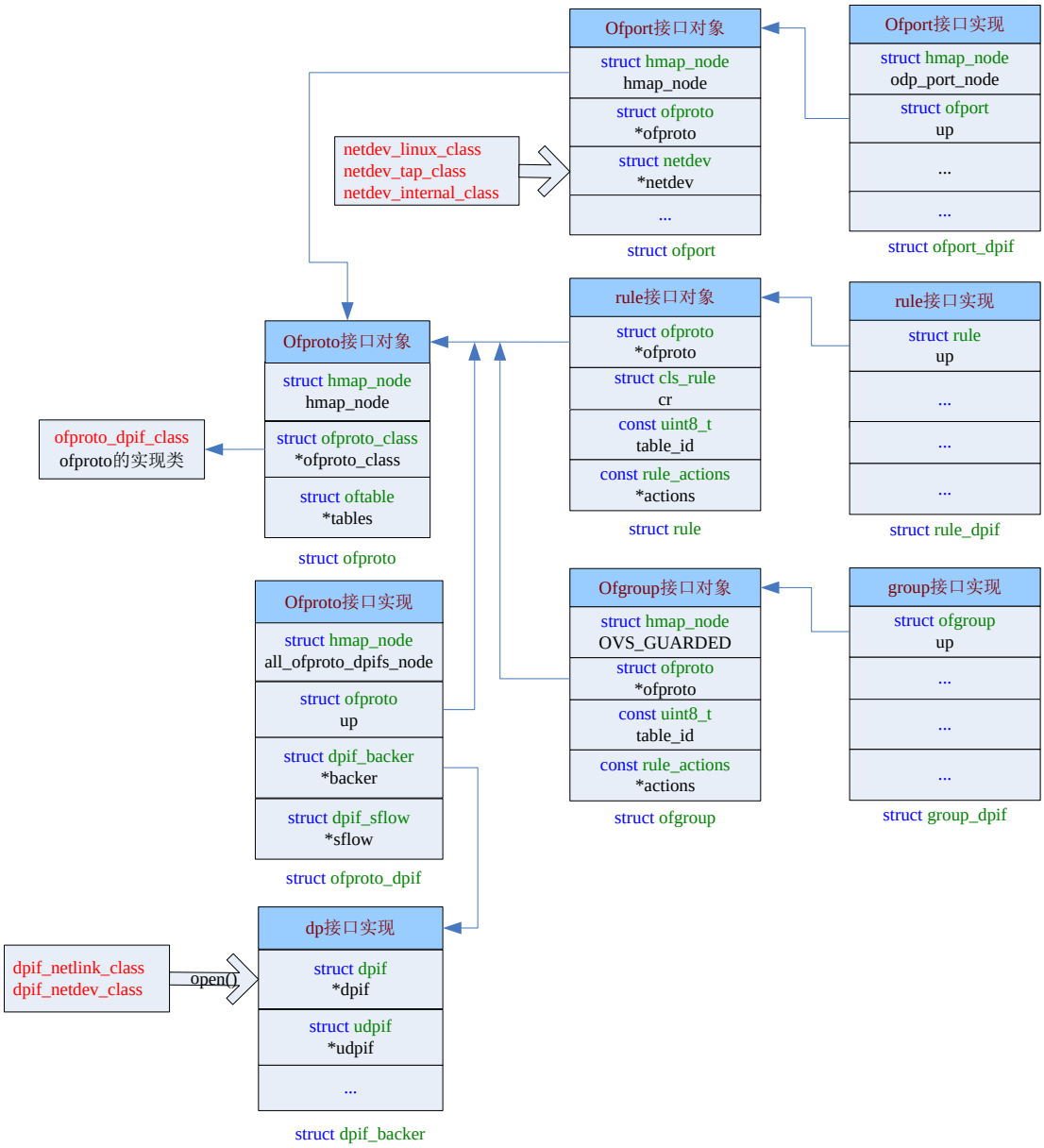


3.3 ofproto

ofproto 层通过 ofproto_class 类（实现是 ofproto_dpif_class）实现了 openflow 的接口，它主要包括如下几个接口类对象：

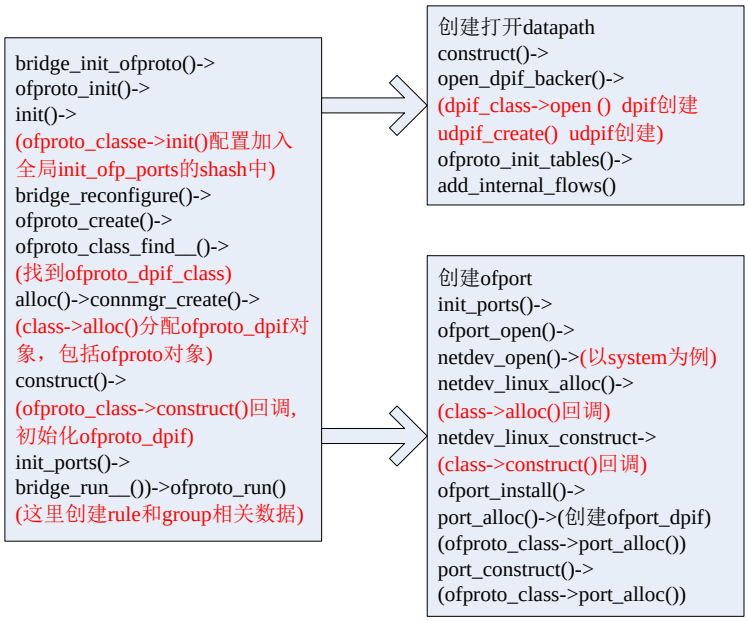
- ofproto 代表了一个 openflow switch 的具体实现，是 ofproto 层的整体结构体；
- ofport 代表了一个 openflow switch 的端口，关联一个 netdev 设备；
- ofrule 代表了一条 openflow 规则，rule 里面包含一组 actions；
- ofgroup 代表了一个 openflow 的行为组合，openflow 1.1+以上版本支持；

3.3.1 ofproto 数据结构



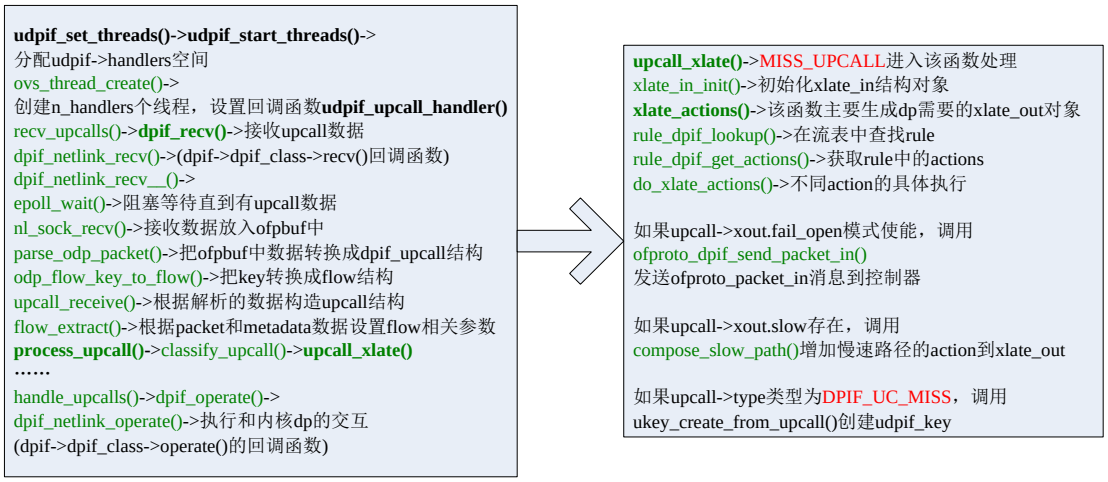
3.3.2 ofproto 创建流程

其中 rule 和 group 的创建流程不在本节列出



3.3.3 udpif

udpif 接口层采用多个线程处理内核发往用户层的 upcall 请求，入口函数为 udpif_set_threads(),主要处理流程如下：

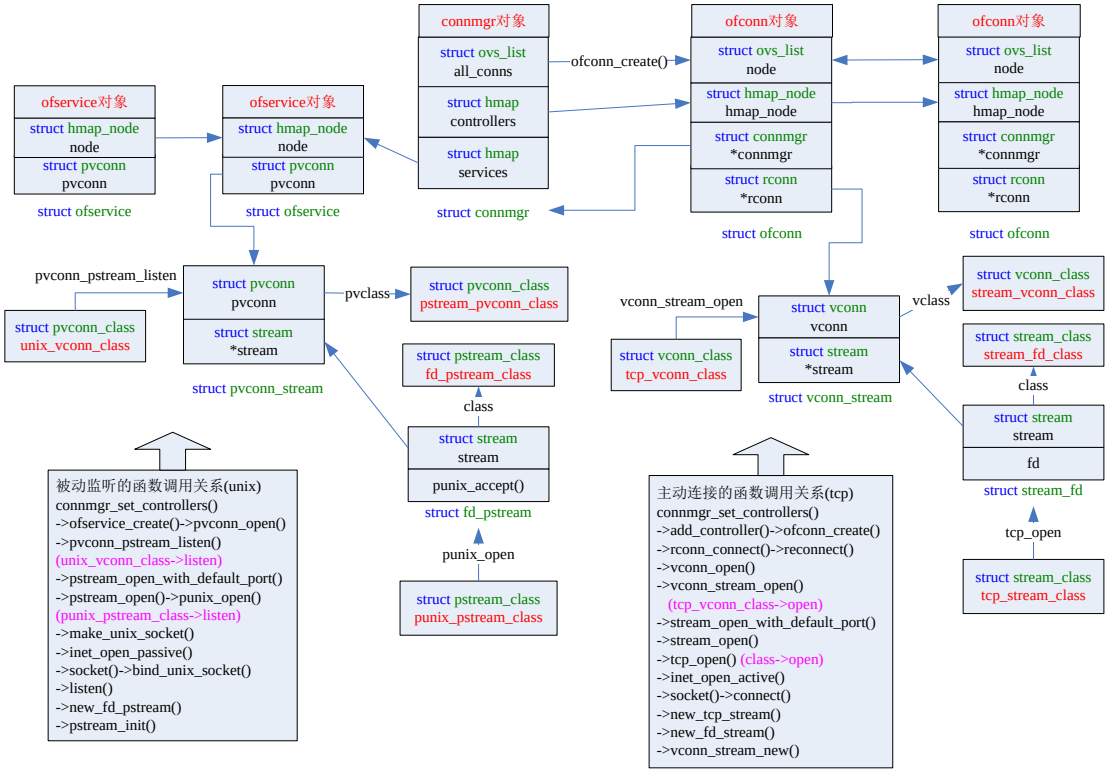


3.4 openflow

OpenFlow 是用于管理交换机流表的协议，ovs-ofctl 则是 OVS 提供的命令行工具。在没有配置 OpenFlow controller 的模式下，用户可以使用 ovs-ofctl 命令通过 OpenFlow 协议去连接 OVS，创建、修改或删除 OVS 中的流表项，并对 OVS 的运行状况进行动态监控。

3.4.1 openflow 连接建立

在 bridge_reconfigure()函数中调用 bridge_configure_remotes 进行 openflow 连接的相关处理，主要创建两个对象：ofconn 作为客户端负责和远端 conntroller 主动建立连接；ofservice 作为服务器提供被动式的监听服务，主要数据结构及流程如下图：



3.4.2 ofconn 报文处理

入口函数 ofproto_run()，主要调用流程如下：

ofproto_run()
->connmgr_run()->ofconn_run()
->rconn_run()->vconn_run()
->vconn_stream_run()
(vconn->vclass->run)
->vconn_stream_send()->stream_send()
->rconn_recv()->vconn_recv()
->do_recv()->vconn_stream_recv()
(vconn->vclass->recv)
->vconn_stream_recv__()
->stream_recv()->fd_recv()
(stream->class->recv)
->recv()系统调用

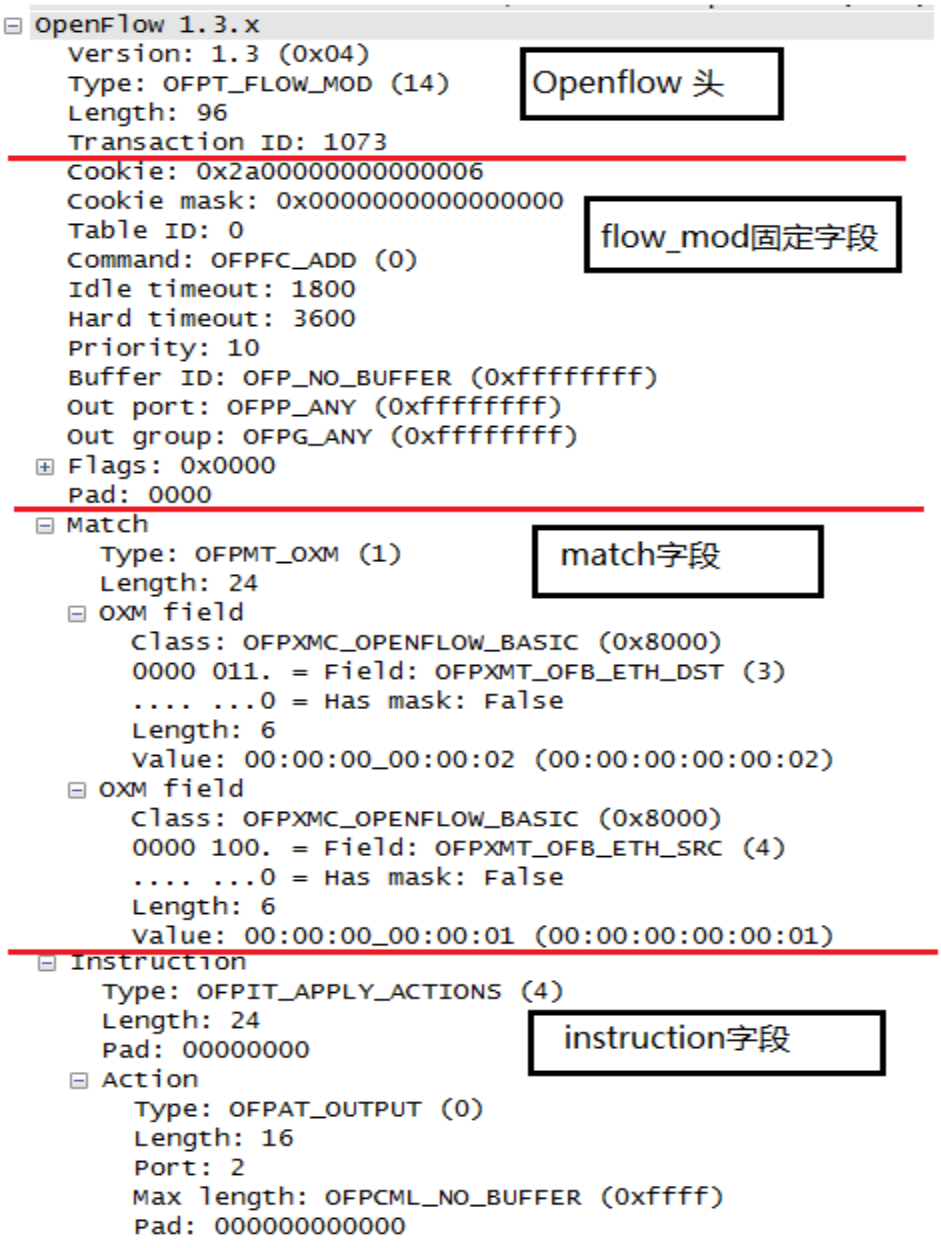
3.4.3 ofservice 报文处理

ofproto_run()
->connmgr_run()->pvconn_accept()
->pvconn_pstream_accept()
(pvconn->pvclass->accept)
->pstream_accept()
->pfd_accept()
(pstream->class->accept)
->accept()->punix_accept()
->new_fd_stream()
->stream_init()(stream_fd_class)
->rconn_create()->ofconn_create()
(OFCONN_SERVICE)
创建ofconn插入到all_conns链表中，下一次执行connmgr_run函数时，会调用ofconn_run，完成报文处理

3.4.4 flow_mod 消息格式

openflow 协议消息处理入口函数是 handle_openflow()，其中最重要的是 flow_mod 流表项的处理，flow_mod 流表的报文格式主要有四部分组成：openflow 头部、flow_mod 固定字段、match 字段和 instruction 字段。

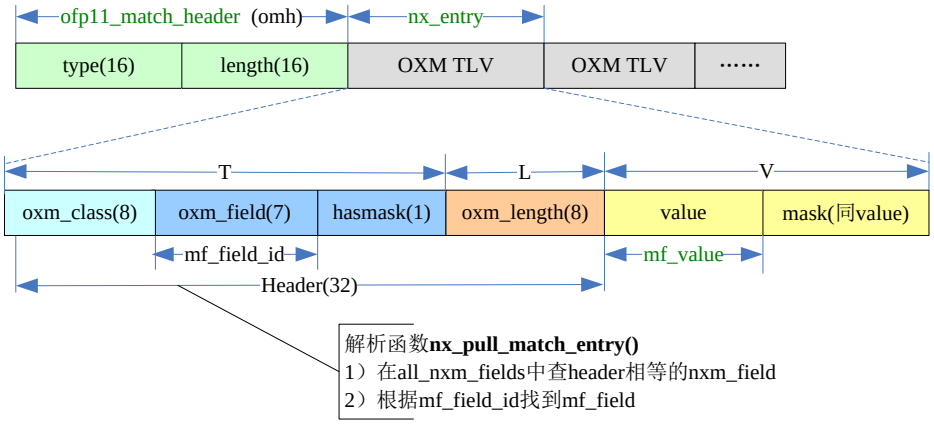
match 分为 OFPMT_STANDARD 和 OFPMT_OXM 两种类型，可以包含多个 oxm，instruction 可以包含多个 action，也可以没有。抓包示例可参考如下：



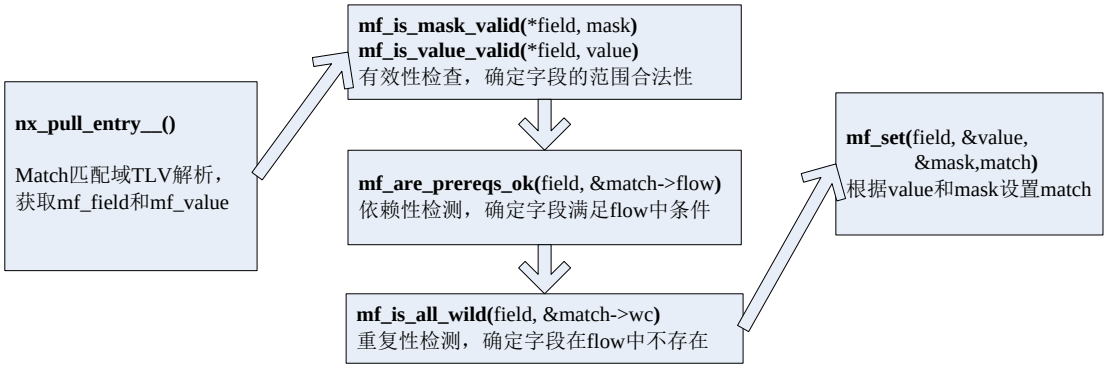
3.4.5 match 字段处理

match 字段的解析处理入口函数为 ofputil_pull_ofp11_match(),其中的核心处理函数为 nx_pull_raw(),主要流程是解析出 flow_mod 的 match 字段，和 flow 中的 match 相关参数做一些合法性检测，最后使用解析出的 value 更新 flow 中的 match。

目前 match 匹配域用的较多的是 OXM 即 TLV 格式，字段解析结构示意图如下：



核心流程处理如下：



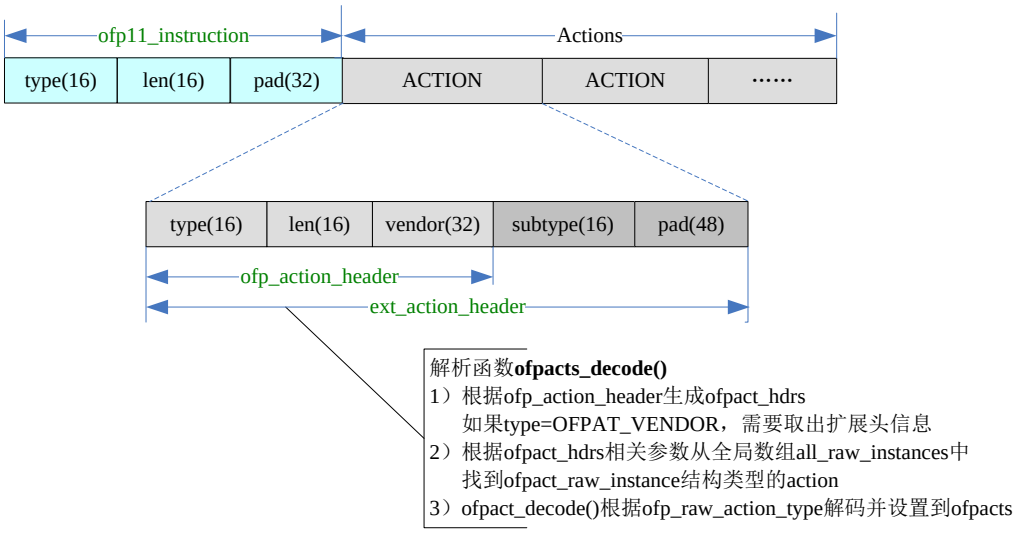
3.4.6 instruction 字段处理

instruction 字段的解析处理入口函数 ofpacts_pull_openflow_instructions(),主要流程是解析出 flow_mod 的 instruction 字段，根据不同的 instruction 做不同的处理，其中函数 decode_openflow11_instructions()解析出所有 instruction 并按照不同类型放入 ofp11_instruction *insts[N_OVS_INSTRUCTIONS]数组中，N_OVS_INSTRUCTIONS 根据 OVS_INSTRUCTIONS 定义推导出值为 6(即 instruction 支持的所有类型)。

其中最重要的宏 OVS_INSTRUCTIONS 完成了主要的数据生成和转换，根据它的定义可推导出 instruction 的类型和后续主要的处理函数对应关系：

Instruction类型	Instruction处理
OVSINST_OFPIT13_METER	ofpact_put_METER(ofpacts)增加meter
OVSINST_OFPIT11_APPLY_ACTIONS	ofpacts_decode()解析actions放入ofpacts
OVSINST_OFPIT11_CLEAR_ACTIONS	ofpact_put_CLEAR_ACTIONS(ofpacts)清空action
OVSINST_OFPIT11_WRITE_ACTIONS	ofpacts_decode_for_action_set()重写action
OVSINST_OFPIT11_WRITE_METADATA	ofpact_put_WRITE_METADATA(ofpacts)重写元数据
OVSINST_OFPIT11_GOTO_TABLE	ofpact_put_GOTO_TABLE(ofpacts)跳转流表

ofpacts_decode()函数完成 actions 的解析，字段解析结构示意图如下：



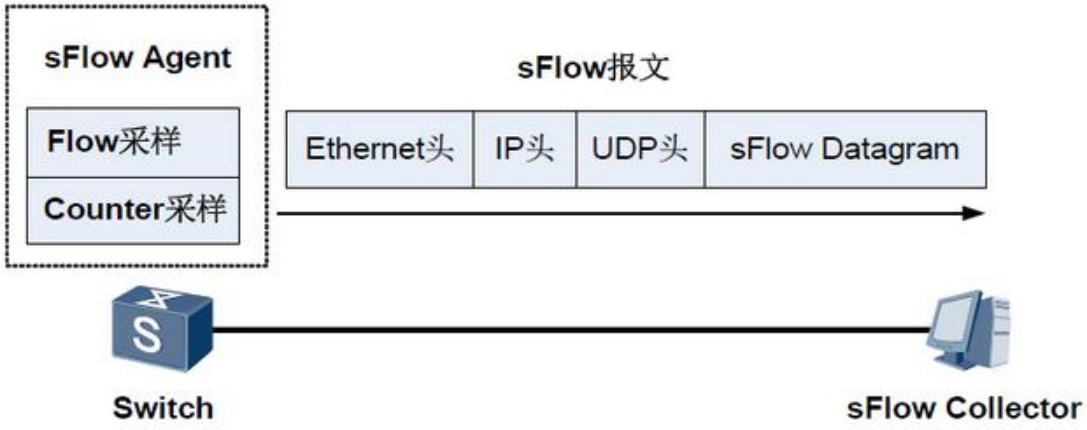
最后调用 ofpacts_check_consistency()进行参数的有效性检查。

3.4.7 flow_mod 处理流程

以增加流表 OFPTYPE_FLOW_MOD 为例整理函数处理流程如下：

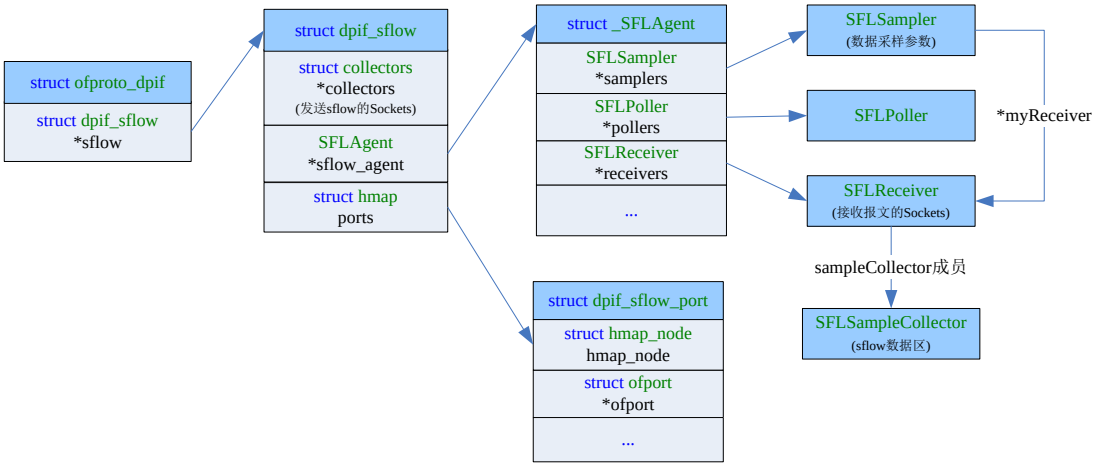
3.5 sflow

采样流 sFlow（Sampled Flow）是一种基于报文采样的网络流量监控技术，主要用于对网络流量进行统计分析。sFlow 系统包含一个嵌入在设备中的 sFlow Agent 和远端的 sFlow Collector。其中，sFlow Agent 通过 sFlow 采样获取本设备上的接口统计信息和数据信息，将信息封装成 sFlow 报文，当 sFlow 报文缓冲区满或是在 sFlow 报文缓存时间超时后，sFlow Agent 会将 sFlow 报文发送到指定的 sFlow Collector。sFlow Collector 对 sFlow 报文进行分析，并显示分析结果，组网图如下：



3.5.1 sflow 初始化

SFlow 的配置入口函数是 bridge_configure_sflow(),该函数创建 SFlow 相关的数据区并初始化相应配置，形成的主要数据结构如下：



3.5.2 sflow 流表生成

sflow 的实现是在用户态生成 sflow 的流表并配置到内核 datapath，内核完成数据采样，发送到用户态，然后有上层 sflow agent 发送到 collector。sflow 在用户态生成 sflow 的流表。流表生成的代码调用流程 add_sflow_action()->compose_sflow_cookie()->compose_sample_action(),生成 OVS_ACTION_ATTR_SAMPLE 类型的 action，主要代码如下：

```
/* Compose SAMPLE action for sFlow or IPFIX.  The given probability is
 * the number of packets out of UINT32_MAX to sample.  The given
 * cookie is passed back in the callback for each sampled packet.
 */
static size_t
compose_sample_action(const struct xbridge *xbridge,
                      struct ofpbuf *odp_actions,
                      const struct flow *flow,
                      const uint32_t probability,
                      const union user_action_cookie *cookie,
                      const size_t cookie_size,
                      const odp_port_t tunnel_out_port)
{
    size_t sample_offset, actions_offset;
    odp_port_t odp_port;
    int cookie_offset;
    uint32_t pid;
//netlink增加OVS_ACTION_ATTR_SAMPLE头
    sample_offset = nl_msg_start_nested(odp_actions, OVS_ACTION_ATTR_SAMPLE);
//netlink增加OVS_SAMPLE_ATTR_PROBABILITY参数
    nl_msg_put_u32(odp_actions, OVS_SAMPLE_ATTR_PROBABILITY, probability);
//netlink增加OVS_SAMPLE_ATTR_ACTIONS
    actions_offset = nl_msg_start_nested(odp_actions, OVS_SAMPLE_ATTR_ACTIONS);
//通过openflow端口号获取datapath端口号
    odp_port = ofp_port_to_odp_port(xbridge, flow->in_port.ofp_port);
//netlink msg pid
    pid = dpif_port_get_pid(xbridge->dpif, odp_port,
                            flow_hash_5tuple(flow, 0));
//增加发送到内核的用户空间数据
//数据在compose_sflow_cookie函数中组成，数据类型USER_ACTION_COOKIE_SFLOW
    cookie_offset = odp_put_userspace_action(pid, cookie, cookie_size,
                                             tunnel_out_port, odp_actions);

    nl_msg_end_nested(odp_actions, actions_offset);
    nl_msg_end_nested(odp_actions, sample_offset);
    return cookie_offset;
}
```

3.5.3 sflow 内核处理

sflow 在内核执行 action 的时候处理，入口函数 do_execute_actions()->sample()，代码如下，如果是最后一个 action，调用 output_userspace()发送数据到用户空间；否则把该 action 加入到队列中等待执行，代码如下：

```
static int sample(struct datapath *dp, struct sk_buff *skb,
          struct sw_flow_key *key, const struct nlattr *attr)
{
    const struct nlattr *acts_list = NULL;
    const struct nlattr *a;
    int rem;

    for (a = nla_data(attr), rem = nla_len(attr); rem > 0;
         a = nla_next(a, &rem)) {
        switch (nla_type(a)) {
        case OVS_SAMPLE_ATTR_PROBABILITY:
            //随机数比较，实现随机概率上报数据，如果随机数大于用户态的参数值，直接退出
            if (prandom_u32() >= nla_get_u32(a))
                return 0;
            break;

        case OVS_SAMPLE_ATTR_ACTIONS:
            acts_list = a;
            break;
        }
    }
```

```
    /* The only known usage of sample action is having a single user-space
     * action. Treat this usage as a special case.
     * The output_userspace() should clone the skb to be sent to the
     * user space. This skb will be consumed by its caller.
     */
    if (likely(nla_type(a) == OVS_ACTION_ATTR_USERSPACE &&
           nla_is_last(a, rem)))
        return output_userspace(dp, skb, key, a);

    skb = skb_clone(skb, GFP_ATOMIC);
    if (!skb)
        /* Skip the sample action when out of memory. */
        return 0;

    if (!add_deferred_actions(skb, key, a)) {
        if (net_ratelimit())
            pr_warn("%s: deferred actions limit reached, dropping sample action\n",
                ovs_dp_name(dp));

        kfree_skb(skb);
    }
    return 0;
}
```

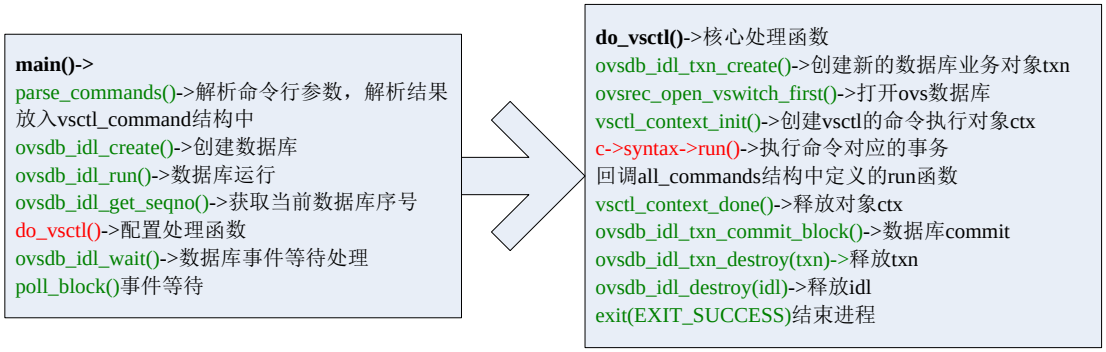
3.5.4 sflow 消息处理

内核 datapath 采样的数据通过 netlink 发送到用户空间的 vswithd 进程，接收函数为 recv_upcalls()，调用 process_upcall()函数进入核心处理流程，其中对 SFLOW_UPCALL 分支的处理就是 sflow 的入口，处理函数 dpif_sflow_received()，核心流程如下：

dpif_sflow_received()->
(创建SFLFLOW_HEADER和
SFLFLOW_EX_SWITCH数据
添加到SFLFlow_sample结构中)
sfl_sampler_writeFlowSample()->
sfl_receiver_writeFlowSample()->
(填充SFLReceiver结构内容)
sendSample()->
receiver->agent->sendFn()->
(sflow_agent_send_packet_cb())
collectors_send()->
(发送给所有的collectors)
Send()系统调用

3.6 ovs-vsctl

ovs-vsctl 根据用户的命令和 ovsdb-server 通信，用于查询和更新数据库配置。而 vswithcd 会在需要重新更新配置的时候和 ovsdb 交互，然后和内核 dp 模块通过 netlink 消息执行真正的操作。本节以添加网桥、端口、vxlan 端口为例分析主要实现流程，其中 ovsctl 进程的主要处理流程如下：



3.6.1 添加网桥

用户态 shell 键入命令 ovs-vsctl add-br br0，启动 vsctl 进程用户完成数据库配置，流程如上面所述，最后调用 add-br 对应的 run 函数 cmd_add_br(),流程如下：

```
cmd_add_br()->
vsctl_context_populate_cache(ctx)->填充配置到ctx中
check_conflicts()->判断配置是否重复

iface = ovsrec_interface_insert(ctx->txn);//Interface表插入行
ovsrec_interface_set_name(iface, br_name);//设置name列
ovsrec_interface_set_type(iface, "internal");//设置type列

port = ovsrec_port_insert(ctx->txn);//Port表插入行
ovsrec_port_set_name(port, br_name);//设置name列
ovsrec_port_set_interfaces(port, &iface, 1);//设置interfaces列

aa = ovsrec_autoattach_insert(ctx->txn);//AutoAttach插入行

br = ovsrec_bridge_insert(ctx->txn);//Bridge插入行
ovsrec_bridge_set_name(br, br_name);//设置name列
ovsrec_bridge_set_ports(br, &port, 1);//设置ports列
ovsrec_bridge_set_auto_attach(br, aa);//设置auto_attach列

ovs_insert_bridge(ctx->ovs, br);//写入数据库

vsctl_context_invalidate_cache(ctx);//释放ctx
```

vswithcd 检测到数据库的设置后完成业务配置，流程如下：

```
bridge_run()->ofproto_init()->bridge_run__()->

bridge_reconfigure()->add_del_bridges()->
bridge_create()->增加桥到all_bridge链表
bridge_delete_ofprotos()->删除错误的ofprotos
ofproto_create()->创建桥的ofproto
ofproto_class_find__()->(class=ofproto_dpif_class)
class->alloc()回调alloc(),分配ofproto_dpif结构
ofproto_class->construct()回调construct()
open_dpif_backer()->所有dp的基础结构函数
……
ofproto_init_tables()->创建流表
add_internal_flows()->增加内部流表

init_ports()->
ofport_open()->桥默认创建internel类型的netdev端口
ofport_install()->创建ofport结构并初始化
ofproto_class->port_alloc()
ofproto_class->port_construct()

bridge_run__()
```

```
open_dpif_backer()->
dpif_create_and_open()->创建并打开一个dp
dpif_create()->do_open()->
dp_initialize()->dp_register_provider()->
注册dpif_classes = dpif_netlink_class和dpif_netdev_class
dp_class_lookup()->(registered_class=dpif_netlink_class)
dpif_class->open()回调dpif_netlink_open()
dpif_netlink_init()->dpif_netlink_dp_transact()->
dpif_netlink_dp_to_ofpbuf()->填充dp的netlink消息
nl_transact()->发送OVS_DP_CMD_NEW到dp
open_dpif()->创建dpif_netlink和dpif，dpif_class=dpif_netlink_class

udpif_create()->dpif_register_upcall_cb()->
创建udpif，注册dpif_class->enable_upcall = upcall_cb()
shash_add(&all_dpif_backers, type, backer)->增加到全局链表
dpif_recv_set()->dpif_netlink_recv_set()->
(dpif_class->recv_set()的回调函数)
dpif_netlink_refresh_channels()->
创建dpif_handler结构作为处理dp的消息通道，发送
OVS_VPORT_CMD_SET消息到dp进行相关配置
```

内核 datapath 通过 OVS_DATAPATH_FAMILY 通道收到一个 OVS_DP_CMD_NEW 类型的添加网桥的命令，该命令绑定的回调函数为 ovs_dp_cmd_new(),处理流程如下：

```
ovs_dp_cmd_new()
ovs_vport_cmd_alloc_info()分配响应消息的skb
ovs_flow_tbl_init()初始化dp的流表
new_vport()新建vport类型OVS_VPORT_TYPE_INTERNAL
ovs_vport_add()
port_ops_list[i]->create()
回调函数ovs_internal_vport_ops结构中的create()
internal_dev_create()
ovs_vport_alloc()分配vport
alloc_netdev()分配netdev
register_netdevice()
ovs_dp_cmd_fill_info()填充响应消息
ovs_notify()发送消息
```

3.6.2 添加端口

shell 键入命令 ovs-vsctl add-port br0 eth0（这里分析 netdev 类型的 vport 端口，vxlan 在下一章节单独分析），vsctl 调用 add-port 对应的函数 cmd_add_port()配置数据库，流程如下：

```
cmd_add_port()->add_port()->
vsctl_context_populate_cache(ctx);填充配置到ctx中
check_conflicts();判断配置端口是否重复
find_bridge();根据bridge名字找到vsctl_bridge

ifaces=xmalloc(n_ifaces * sizeof *ifaces);//分配n_ifaces个ifaces
iface[i]=ovsrec_interface_insert(ctx->txn);//interface表插入行
ovsrec_interface_set_name(iface, br_name);//设置name列

port = ovsrec_port_insert(ctx->txn);//Port表插入行
ovsrec_port_set_name(port, br_name);//设置name列
ovsrec_port_set_interfaces(port, &iface, 1);//设置interfaces列
ovsrec_port_set_bond_fake_iface(port, fake_iface);//设置bond_fake_iface列
set_column(get_table("Port"),...);//port表的列配置写入数据库

bridge_insert_port();//ports插入bridge
ovsrec_bridge_set_ports();//bridge表的ports列写入数据库

add_port_to_cache(ctx, bridge, port);//port增加到ctx
add_iface_to_cache();//iface增加到ctx
```

vswitchd 调用 dpif_netlink_port_add()通过 netlink 发送对应消息到内核，流程和上一节所述的添加网桥类似，如下所示：

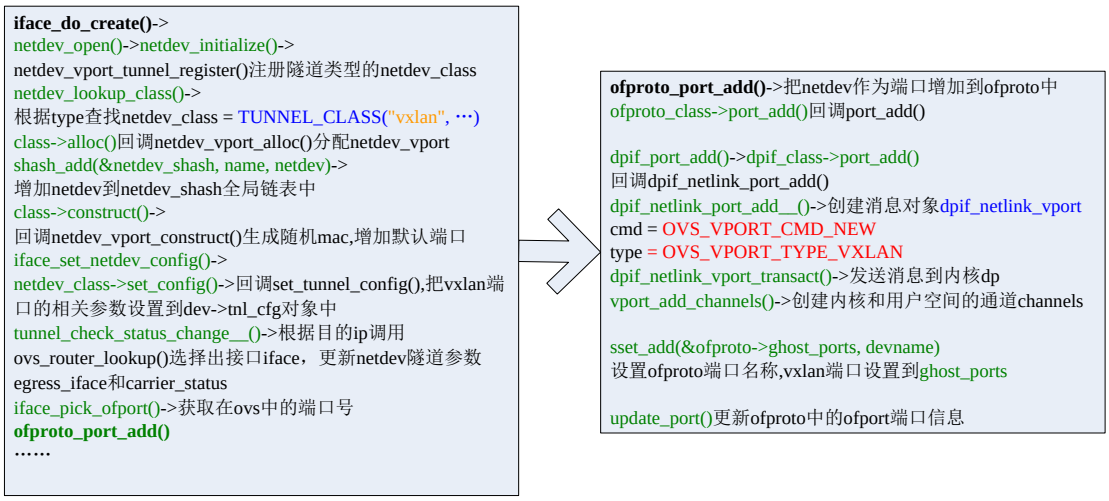
内核 datapath 通过 OVS_VPORT_FAMILY 通道收到一个类型为 OVS_VPORT_CMD_NEW 的添加端口的命令，该命令绑定的回调函数为 ovs_vport_cmd_new ()，处理流程如下：

3.7 vxlan

vxlan 端口是 ovs 的 OVS_VPORT_TYPE_VXLAN 类型的隧道端口，用户态 netdev 库通过 netdev_vport_tunnel_register()注册 vport_class 结构，它包含如 vxlan、gre 等各种类型隧道的相关处理函数。

3.7.1 添加 vxlan 端口

添加命令为 ovs-vsctl add-port br0 vxlan -- set interface vxlan type=vxlan，用户态处理流程和上节的添加端口相同，不同时的是对 vxlan 端口的参数配置，发往内核 dp 的消息类型为 OVS_VPORT_TYPE_VXLAN，流程如下：



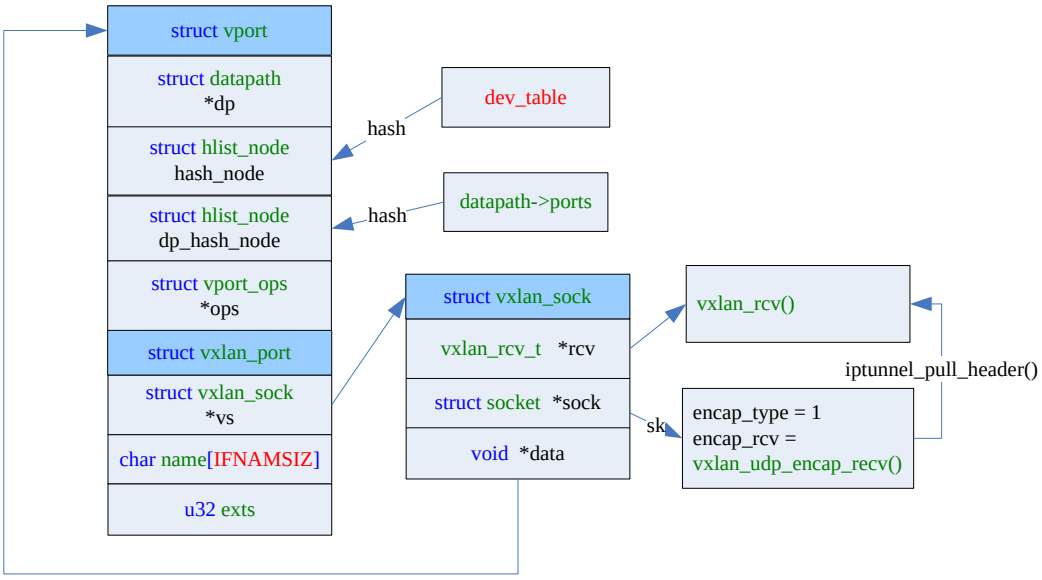
3.7.2 内核 vxlan 创建

内核 ovs_vport_cmd_new ()函数中 ovs_vport_add()调用 ovs_vxlan_vport_ops 对应的操作函数，其中创建函数 vxlan_tnl_create()流程如下：

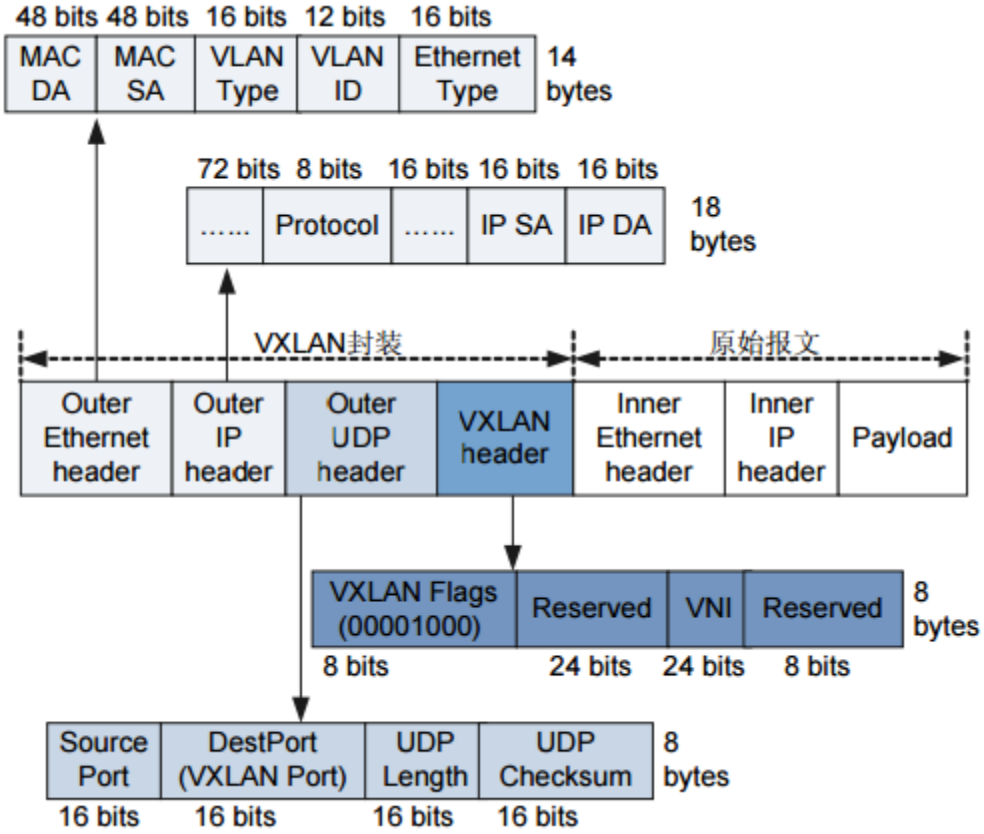
```
a = nla_find_nested(options, OVS_TUNNEL_ATTR_DST_PORT);
if (a && nla_len(a) == sizeof(u16)) {
    dst_port = nla_get_u16(a);//L4 隧道使用的目的端口
} else {
    /* Require destination port from userspace. */
    err = -EINVAL;
    goto error;
}
//创建vport对象
vport = ovs_vport_alloc(sizeof(struct vxlan_port),
            &ovs_vxlan_vport_ops, parms);
if (IS_ERR(vport))
    return vport;
//vxlan_port位置在vport地址后面
vxlan_port = vxlan_vport(vport);
strncpy(vxlan_port->name, parms->name, IFNAMSIZ);

a = nla_find_nested(options, OVS_TUNNEL_ATTR_EXTENSION);
if (a) {//vxlan的exts配置
    err = vxlan_configure_exts(vport, a);
    if (err) {
        ovs_vport_free(vport);
        goto error;
    }
}
/* 内核创建udp sock，用于接收和发送vxlan隧道报文，
   sock接收函数设置为vxlan_udp_encap_recv()
   该函数去除隧道头部后调用vxlan_rcv()进入dp的处理*/
vs = vxlan_sock_add(net, htons(dst_port), vxlan_rcv, vport, true,
            vxlan_port->exts);
if (IS_ERR(vs)) {
    ovs_vport_free(vport);
    return (void *)vs;
}
vxlan_port->vs = vs;
```

3.7.3 vport 数据结构

根据上述流程分析，vxlan 类型的 vport 数据结构如下：



3.7.4 vxlan 报文格式



3.7.5 vxlan 收包处理

vxlan 隧道报文的处理的入口函数是 udp sock 监听函数 vxlan_udp_encap_recv(),处理流程如下：

vxlan_udp_encap_recv()->
获取vxlan头部的VXLAN Flags和VNI值
iptunnel_pull_header()->
获取隧道内部eth头部信息并设置skb->protocol
vs->rcv->回调vxlan_rcv()
ovs_flow_tun_info_init()->设置ovs_tunnel_info对象信息
ovs_vport_receive(vport, skb, &tun_info)进入ovs的处理
ovs_flow_key_extract()->设置key->tun_key和key
ovs_dp_process_packet()进入dp的处理，其中tun的处理
ovs_execute_actions()->do_execute_actions()->
do_execute_actions()->execute_set_action()
OVS_KEY_ATTR_TUNNEL_INFO类型的处理中设置
OVS_CB(skb)->egress_tun_info参数

3.7.6 vxlan 发包处理

vlan 在执行 action 时，判断需要发送数据的时候调用发送函数，函数调用 do_execute_actions()->do_output()->ovs_vport_send()->vxlan_tnl_send(),这里 vxlan_tnl_send()函数即为创建 vport 端口是注册 ovs_vxlan_vport_ops 的 send 函数，主要处理流程如下：

vxlan_tnl_send()->
tun_key = &OVS_CB(skb)->egress_tun_info->tunnel
获取隧道信息
find_route()->查询路由
udp_flow_src_port()->选择源端口

vxlan_xmit_skb()->
udp_tunnel_handle_offloads()->
设置inner相关的head，设置encapsulation
vlan_hwaccel_push_inside()->如果存在vlan，增加vlan头到skb
vxh = (struct vxlanhdr *) __skb_push(skb, sizeof(*vxh))->
增加vxlan头部并进行赋值
vxlan_set_owner()->设置发包sock
ovs_skb_set_inner_protocol(skb, htons(ETH_P_TEB))->
设置内部协议

udp_tunnel_xmit_skb()->
__skb_push(skb, sizeof(*uh))->增加udp头部并进行赋值
udp_set_csum()->计算校验和
iptunnel_xmit()->(rpl_iptunnel_xmit())调用发送函数
__skb_push(skb, sizeof(struct iphdr))->增加ip头并进行赋值
ip_local_out()->(rpl_ip_local_out())->output_ip()->
#undef ip_local_out 取消vxlan函数定义
ip_local_out(skb)调用内核本地发包接口函数