OVS源码阅读--datapath新增流表项流程分析

基本流程

当用户空间向内核新增一个流表项时，主要涉及到以下步骤：

1. 内核接受来自用户空间的消息请求，判断请求类型是新增流表项，先将消息解析并封装成genl_info结构，然后调用ovs_flow_cmd_new回调函数进行具体的处理；
2. 提取OVS_FLOW_ATTR_KEY类型的消息，将其填充到sw_flow_key结构体中；
3. 提取OVS_FLOW_ATTR_MASK类型的消息，将其填充到sw_flow_mask结构体中；
4. 提取OVS_FLOW_ATTR_ACTION类型的消息，将其填充到sw_flow_action结构体中；
5. 获得对应的datapath，查询该datapath中的流表，判断要新增的流表项是否已经存在，如果存在则更新该流表项的动作，否则将该流表项插入到对应的流表中

下面按照上述的流程一步步进行分析。

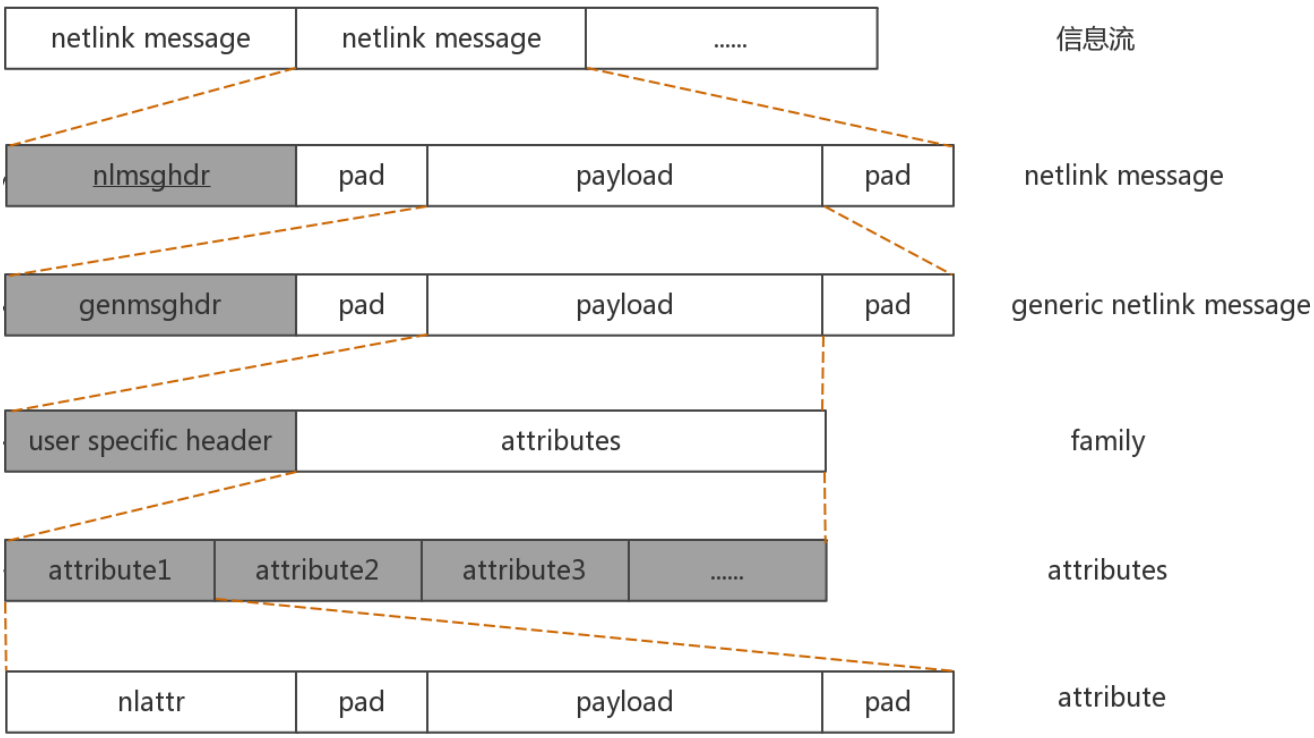
> 有关netlink和ovs数据结构之间的关系，可以参考我之前的博客

封装genl_info

在ovs中，用户态和内核态是通过generic netlink机制进行通信，首先简答介绍一下netlink的消息结构体。

netlink message 结构

一个netlink消息的结构如下：



该消息结构主要分为上面六层：

信息流

最上一层为信息流，一个netlink的消息中可能含有多个netlink message。

netlink message

每一个netlink message中包含一个nlmsghdr结构体，该结构体中记录了netlink消息的长度、类型等信息，payload域为具体的消息内容。

```
struct nlmsghdr {
    __u32       nlmsg_len;      // 长度（包括haeder）
    __u16       nlmsg_type;   // 类型
    __u16       nlmsg_flags;  // 标签
    __u32       nlmsg_seq;    // 序号
    __u32       nlmsg_pid;    // 发送进程的id
}
```

generic netlink message

generic netlink message 放在netlink message 的playload域中，该协议对netlink的协议进行了扩展，其中genmsghdr结构体记录了generic netlink message的命令号等信息，在generic netlink中，使用cmd来标识一条消息。另外payload域中存在具体的消息的内容。

```

```

user space header

在generic netlink message的payload中，如果用户有自定义的内容，可以将该内容存放在user space header中，在ovs中，该字段存放的是ovs_header结构体的内容，下面会具体的介绍，紧接在user space header之后是具体的属性信息；如果用户为定义user space header，则generic netlink message的payload中存放的就是属性信息，user space header的长度为0。

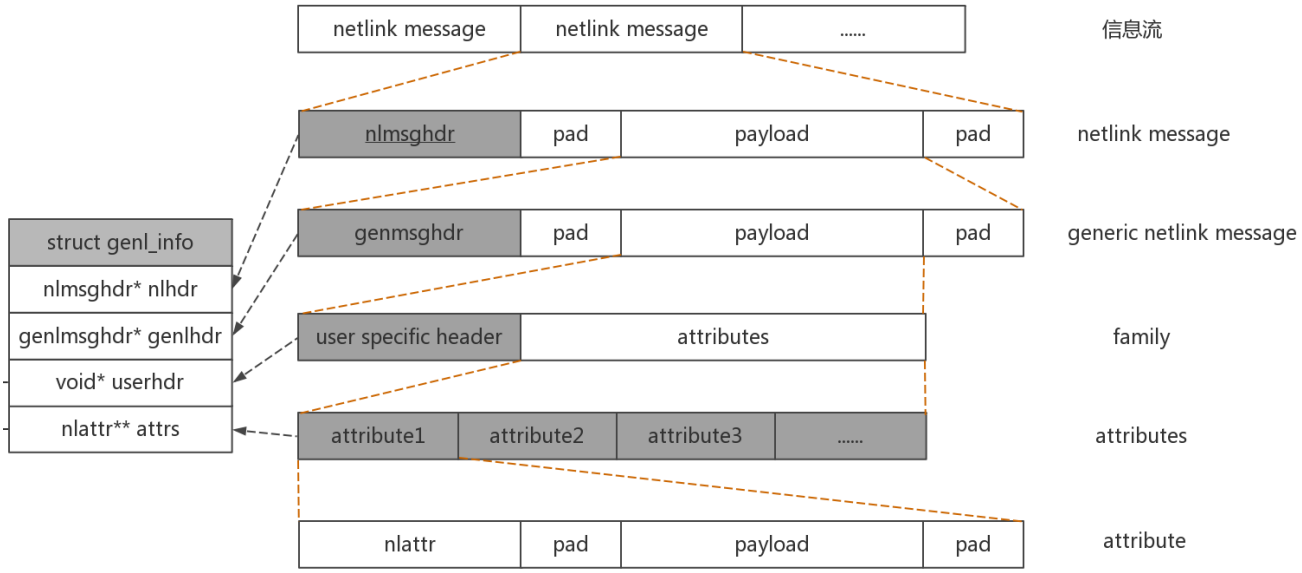
attributes

接下来就是具体的属性信息，从上图中可以看到，一个payload中含有多个不同类型的属性信息，每个属性信息包含一个nalttr结构体和payload，其中nalttr结构体包含属性的长度和类型，payload中为具体的属性的信息。

```
struct nlattr {
    __u16 nla_len; //属性长度
    __u16 nla_type; //属性类型
};
```

genl_info结构

当消息传送到内核空间，内核会将netlink message中的消息封装入一个genl_info结构体中，下面是genl_info结构体和netlink消息的具体字段对应情况：

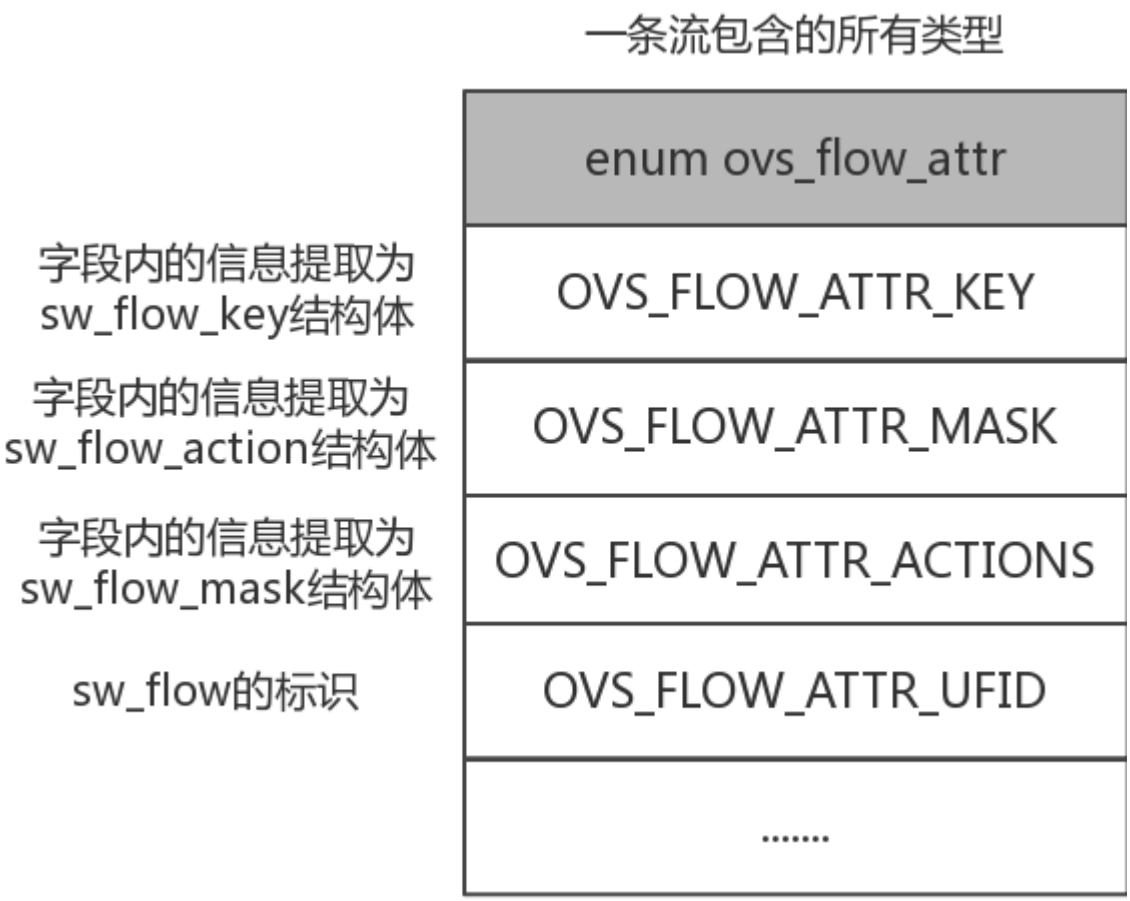


内核将一个netlink message中的信息提取到genl_info结构体中，在新增流表项的回调函数ovs_flow_cmd_new中，主要使用到gen_info中的attrs字段，该字段对应一个netlink message中所有属性的信息。

属性的类型

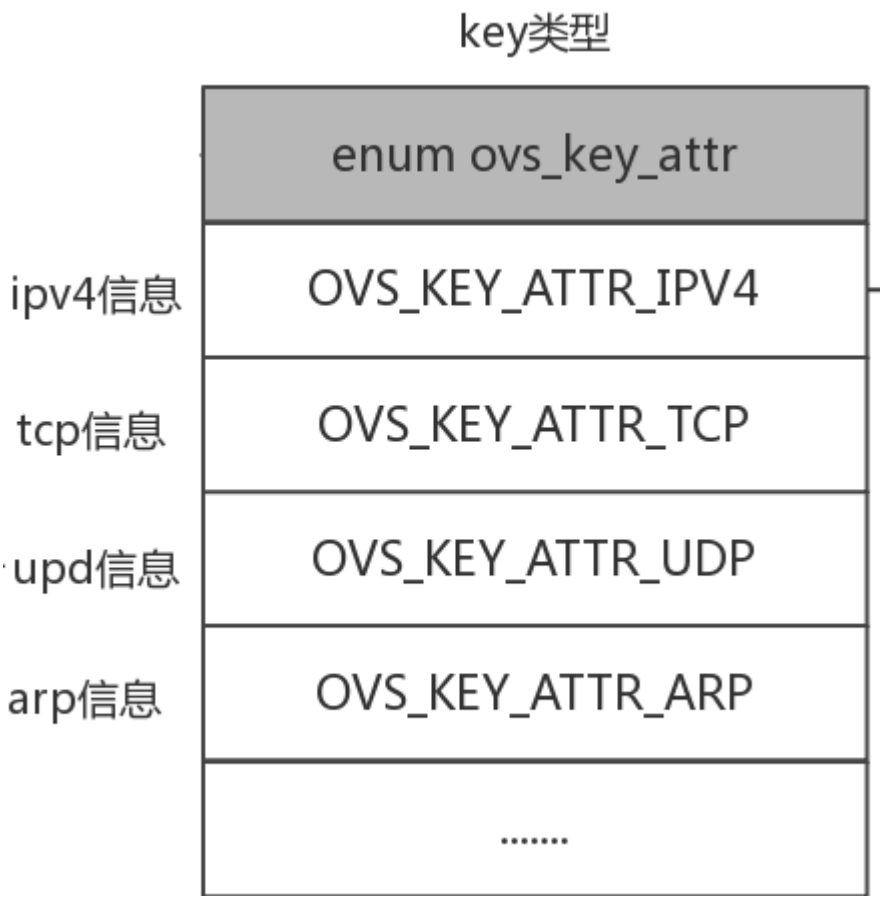
在具体解析之前，首先得介绍一下具体属性的类型，在新增流表项的消息中，由于用户空间需要向内核空间增加一个流表项，流表项的结构为sw_flow，具体参见之前的博客，sw_flow中重要的结构体为sw_flow_key、sw_flow_mask、sw_flow_acton，这些结构体包含具体的ip、tcp、udp、动作等信息，所以用户发送的属性内容中，需要填充下面的字段。

在openvswitch.h中定义了具体属性的类型：

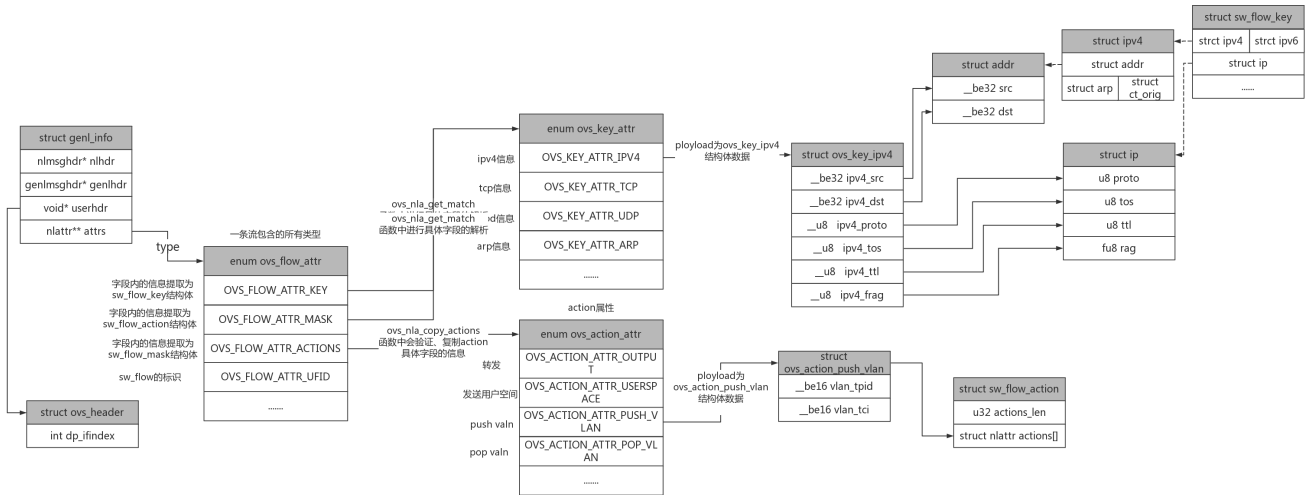


嵌套属性

值得注意的是，在不同类型的属性中，不全都是直接简单的将内容放在payload字段，有的属性中含有嵌套的属性，也就是paylaod字段中提取出来的仍然是一个nlattr数据。以具体的OVS_FLOW_ATTR_KEY类型为例，该类型的属性信息解析完之后需要将其塞入一个sw_flow_key结构体，当解析该类型的属性时，将对应的payload字段内容提出出来后成为一个子nlattr，然后该子nlattr对应的属性类型如下：



其中不同的类型对应着sw_flow_key中不同的结构体，同理ovs_flow_attr_mask结构体和ovs_flow_attr_action结构体，完整的一张包含关系图如下：



在上图中，首先在ovs中genl_info->userhdr字段（user space header）对应的是一个ovs_header结构体，该结构体中只含有一个dp_ifindex整形变量，内核通过该变量定位用户发送流表信息的目的datapath；接着就是刚才上述提到的嵌套属性，其中OVS_FLOW_ATTR_MASK和OVS_FLOW_ATTR_KEY中的属性类型相同，这是因为ovs_flow_mask会对ovs_flow_key进行掩码操作，具体可以参考博客和博客，同样的OVS_FLOW_ATTR_ACTIONS类型的属性中也包含嵌套属性，其对应的嵌套属性类型为ovs_action_attr下面的类型，最终会对应到sw_flow_action结构体中。

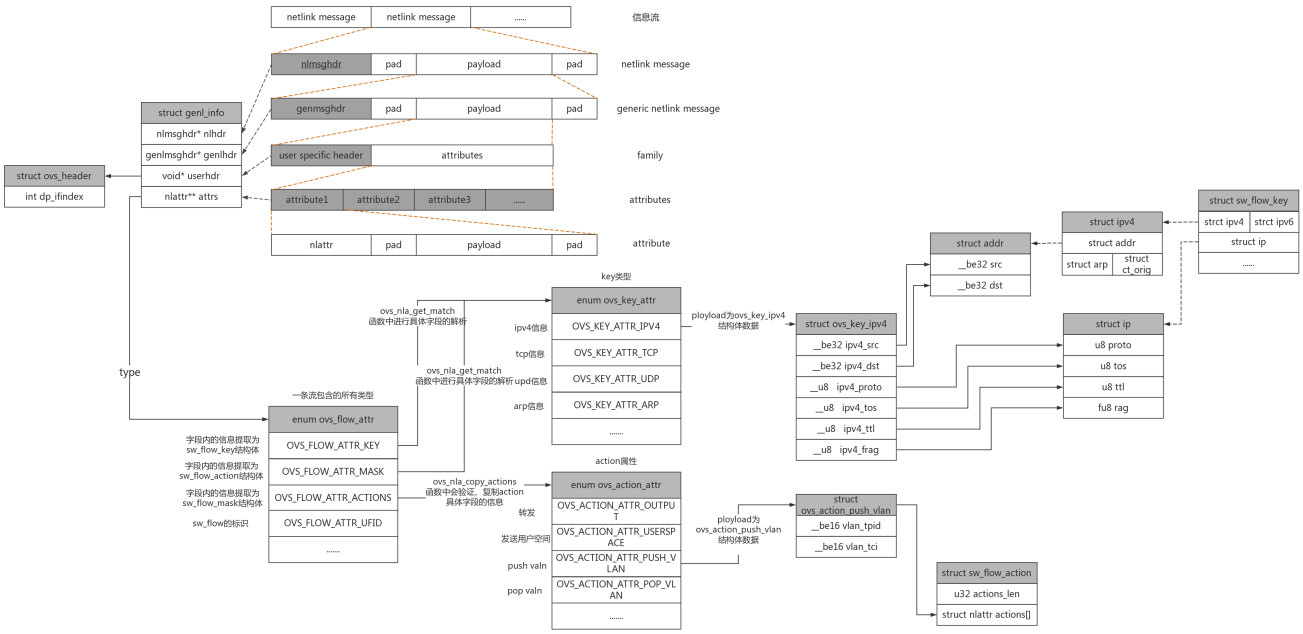
以OVS_FLOW_ATTR_KEY类型的属性进行具体的说明，其他的属性就不再此赘述：

1. 首先提取OVS_FLOW_ATTR_KEY属性之后，将对应的payload转化成一个新的nlattr，这个子nlattr中所有的类型为枚举ovs_key_attr；

2. 判断子nlattr的类型，这里以OVS_KEY_ATTR_IPV4进行说明，如果为OVS_KEY_ATTR_IPV4，将其payload字段的内容转化为ovs_key_ipv4结构体，该结构体中的内容如上图所示，也就是说，该类型的属性内容即为ovs_key_ipv4结构体中的内容；
3. 然后在上图的右上方，显示了ovs_key_ipv4结构体中的字段对应到sw_flow_key中的字段，这里不再详细介绍sw_flow_key的字段信息，具体参考博文，这个解析的过程会在接下来的文章中进行介绍

小结

综上，我们可以得到ovs的netlink信息提取中，字段信息的对应，如下图（只显示部分属性类型，点击可查看大图）：



新增流表流程

接下来根据代码进行具体的分析

ovs_flow_cmd_new

首先贴上ovs_flow_cmd_new的源代码：

```
/**
    新增一个流表项
    info中包含了一个流表项所需的信息（sw_flow_key、sw_flow_action、sw_flow_mask）；创建一个流表项（对应TTS中的rule，包括action），可能会新建一个mask（查询是否存在现有的mask）
**/
static int ovs_flow_cmd_new(struct sk_buff *skb, struct genl_info *info)
{
    struct net *net = sock_net(skb->sk);
    struct nlattr **a = info->attrs;
    struct ovs_header *ovs_header = info->userhdr; //来自于info中的自定义字段，用户发送信息的时候，会选择填充该字段，如果有则含有用户自定义的数据
    struct sw_flow *flow = NULL, *new_flow;
    struct sw_flow_mask mask;
    struct sk_buff *reply;
```

```
struct datapath *dp;
struct sw_flow_actions *acts;
struct sw_flow_match match;
u32 ufid_flags = ovs_nla_get_ufid_flags(a[OVS_FLOW_ATTR_UFID_FLAGS]);
int error;
bool log = !a[OVS_FLOW_ATTR_PROBE];

/* Must have key and actions. */
error = -EINVAL;
if (!a[OVS_FLOW_ATTR_KEY]) {
    OVS_NLERR(log, "Flow key attr not present in new flow.");
    goto error;
}
if (!a[OVS_FLOW_ATTR_ACTIONS]) {
    OVS_NLERR(log, "Flow actions attr not present in new flow.");
    goto error;
}

/* Most of the time we need to allocate a new flow, do it before
 * locking.
 */
new_flow = ovs_flow_alloc(); //给sw_flow结构体分配空间
if (IS_ERR(new_flow)) {
    error = PTR_ERR(new_flow);
    goto error;
}

/* Extract key. */
ovs_match_init(&match, &new_flow->key, false, &mask); //初始化sw_flow_match

//从nlattr中提取信息到match的sw_flow_key和sw_flow_mask结构体中
error = ovs_nla_get_match(net, &match, a[OVS_FLOW_ATTR_KEY],
              a[OVS_FLOW_ATTR_MASK], log);
if (error)
    goto err_kfree_flow;

//获得flow的标识（ufid或者unmasked_key）
error = ovs_nla_get_identifier(&new_flow->id, a[OVS_FLOW_ATTR_UFID],
                   &new_flow->key, log);
if (error)
    goto err_kfree_flow;

//判断sw_flow的标识是否为key，如果是则match.key赋值为未掩码的unmasked_key
if (ovs_identifier_is_key(&new_flow->id))
    match.key = new_flow->id.unmasked_key;

//将new_flow->key与mask->key做‘与’操作，然后将结果保存到new_flow->key
ovs_flow_mask_key(&new_flow->key, &new_flow->key, true, &mask);

/* Validate actions. */
//验证actions
error = ovs_nla_copy_actions(net, 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;
    }

    //reply为回复给用户空间的消息，申请reply所需空间
    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ovs_header->dp_ifindex获得对应的datapath
    dp = get_dp(net, ovs_header->dp_ifindex);
    if (unlikely(!dp)) {
        error = -ENODEV;
        goto err_unlock_ovs;
    }

    //如果一个flow的标识为ufid，这个flow既会放在dp->table->ti表中，也会放入dp->table->uti表中
    if (ovs_identifier_is_ufid(&new_flow->id))
        //查找是否存在对应的流表项，这里面查找的是table->uti表格
        flow = ovs_flow_tbl_lookup_ufid(&dp->table, &new_flow->id);
    if (!flow) //如果不存在
        //查找table，这里面查找的是table->ti表格
        flow = ovs_flow_tbl_lookup(&dp->table, &new_flow->key);
    if (likely(!flow)) { //如果还是找不到，说明dp中没有该流表项
        rcu_assign_pointer(new_flow->sf_acts, acts); //赋值action

        /* Put flow in bucket. */
        error = ovs_flow_tbl_insert(&dp->table, new_flow, &mask); //插入新的表项
        if (unlikely(error)) {
            acts = NULL;
            goto err_unlock_ovs;
        }

        if (unlikely(reply)) {
            //构造回复给用户空间的信息
            error = ovs_flow_cmd_fill_info(new_flow,
                                ovs_header->dp_ifindex,
                                reply, info->snd_portid,
                                info->snd_seq, 0,
                                OVS_FLOW_CMD_NEW,
                                ufid_flags);
            BUG_ON(error < 0);
        }
        ovs_unlock();
    } else { //如果找到了对应的流表项，则应该更新对应的流表项
        struct sw_flow_actions *old_acts;

        /* Bail out if we're not allowed to modify an existing flow.
```

```
         * We accept NLM_F_CREATE in place of the intended NLM_F_EXCL
         * because Generic Netlink treats the latter as a dump
         * request.  We also accept NLM_F_EXCL in case that bug ever
         * gets fixed.
         */
        if (unlikely(info->nlhdr->nlmsg_flags & (NLM_F_CREATE
                                 | NLM_F_EXCL))) {
            error = -EEXIST;
            goto err_unlock_ovs;
        }
        /* The flow identifier has to be the same for flow updates.
         * Look for any overlapping flow.
         */
         //如果找到的流表项与要插入的不能匹配（这是在已经确定找到的流表项跟要插入的流表项是相同的，所以一般不会出现不能匹配的情况）
        if (unlikely(!ovs_flow_cmp(flow, &match))) {
            if (ovs_identifier_is_key(&flow->id))
                //遍历所有的mask，对流表项再次查找
                flow = ovs_flow_tbl_lookup_exact(&dp->table,
                                 &match);
            else /* UFID matches but key is different */
                flow = NULL;
            if (!flow) {
                error = -ENOENT;
                goto err_unlock_ovs;
            }
        }
        /* Update actions. */
        old_acts = ovsl_dereference(flow->sf_acts);
        rcu_assign_pointer(flow->sf_acts, acts); //更新动作

        if (unlikely(reply)) {
            //构造回复的信息
            error = ovs_flow_cmd_fill_info(flow,
                               ovs_header->dp_ifindex,
                               reply, info->snd_portid,
                               info->snd_seq, 0,
                               OVS_FLOW_CMD_NEW,
                               ufid_flags);
            BUG_ON(error < 0);
        }
        ovs_unlock();

        ovs_nla_free_flow_actions_rcu(old_acts); //释放action
        ovs_flow_free(new_flow, false); //释放流表项
    }

    if (reply)
        //发送信息到用户空间
        ovs_notify(&dp_flow_genl_family, &ovs_dp_flow_multicast_group, reply, info);
    return 0;

err_unlock_ovs:
```

```
    ovs_unlock();
    kfree_skb(reply);
err_kfree_acts:
    ovs_nla_free_flow_actions(acts);
err_kfree_flow:
    ovs_flow_free(new_flow, false);
error:
    return error;
}
```

可以看到在该函数中，主要的步骤为：

1. 调用ovs_nla_get_match函数对genl_info中的nlattr信息进行解析，放入到sw_flow_key、sw_flow_mask结构体中；
2. 调用ovs_flow_mask_key将上一部获得的key和mask进行掩码操作；
3. 调用ovs_nla_copy_actions，对gen_info中的action信息进行验证；
4. 调用get_dp函数，获得要处理的datapath；
5. 调用ovs_flow_tbl_lookup_ufid或者ovs_flow_tbl_lookup等函数，查询上一步获得的datapath对应的流表是否含有该流表项；
6. 判断是否对流表项进行插入，如果是则调用ovs_flow_tbl_insert插入到对应的流表中，如果不是则上一部查询到的流表项的动作信息
7. 发送信息返回到用户空间

ovs_nla_get_match

这个函数为解析netlink消息的主要函数，源代码如下：

```
/**
 * ovs_nla_get_match - parses Netlink attributes into a flow key and
 * mask. In case the 'mask' is NULL, the flow is treated as exact match
 * flow. Otherwise, it is treated as a wildcarded flow, except the mask
 * does not include any don't care bit.（如果mask == NULL，则该流为完全匹配流，否则为模糊匹配（也就是通配符匹配，在匹配的时候只关心流的一部分信息，其他的信息不进行匹配））
 * @net: Used to determine per-namespace field support.
 * @match: receives the extracted flow match information.
 * @key: Netlink attribute holding nested %OVS_KEY_ATTR_* Netlink attribute
 * sequence. The fields should of the packet that triggered the creation
 * of this flow.
 * @mask: Optional. Netlink attribute holding nested %OVS_KEY_ATTR_* Netlink
 * attribute specifies the mask field of the wildcarded flow.
 * @log: Boolean to allow kernel error logging.  Normally true, but when
 * probing for feature compatibility this should be passed in as false to
 * suppress unnecessary error logging.
 */
int ovs_nla_get_match(struct net *net, struct sw_flow_match *match,
                      const struct nlattr *nla_key,
                      const struct nlattr *nla_mask,
                      bool log)
{
    const struct nlattr *a[OVS_KEY_ATTR_MAX + 1];
    struct nlattr *newmask = NULL;
    u64 key_attrs = 0;
```

```
    u64 mask_attrs = 0;
    int err;

    //将nla_key中的信息，一个个遍历之后放入a中，key_attr的每个bit记录是否存在对应的OVS_KEY_ATTR_*
类型
    err = parse_flow_nlattrs(nla_key, a, &key_attrs, log);
    if (err)
        return err;

    //解析vlan信息
    err = parse_vlan_from_nlattrs(match, &key_attrs, a, false, log);
    if (err)
        return err;

    //解析sw_flow_key中的其他字段
    err = ovs_key_from_nlattrs(net, match, key_attrs, a, false, log);
    if (err)
        return err;

    if (match->mask) {
        if (!nla_mask) {
            //nla_mask==NULL 则为完全匹配，则将sw_flow_key对应的位置在mask->key中全部置为0xff
            /* Create an exact match mask. We need to set to 0xff
             * all the 'match->mask' fields that have been touched
             * in 'match->key'. We cannot simply memset
             * 'match->mask', because padding bytes and fields not
             * specified in 'match->key' should be left to 0.
             * Instead, we use a stream of netlink attributes,
             * copied from 'key' and set to 0xff.
             * ovs_key_from_nlattrs() will take care of filling
             * 'match->mask' appropriately.
             */
            newmask = kmemdup(nla_key,
                      nla_total_size(nla_len(nla_key)),
                      GFP_KERNEL);  //复制一份新的
            if (!newmask)
                return -ENOMEM;

            mask_set_nlattr(newmask, 0xff); //将newmask中的信息都设置为0xff

            /* The userspace does not send tunnel attributes that
             * are 0, but we should not wildcard them nonetheless.
             */
            if (match->key->tun_proto)
                SW_FLOW_KEY_MEMSET_FIELD(match, tun_key,
                             0xff, true);

            nla_mask = newmask;
        }

        //将nlattr* attr中的信息，一个个遍历之后放入a中，attr的每个bit记录是否存在对应的
OVS_KEY_ATTR_*类型
        err = parse_flow_mask_nlattrs(nla_mask, a, &mask_attrs, log);
```

```
        if (err)
            goto free_newmask;

        SW_FLOW_KEY_PUT(match, eth.vlan.tci, htons(0xffff), true);
        SW_FLOW_KEY_PUT(match, eth.cvlan.tci, htons(0xffff), true);

        //解析VLAN，填充到match->mask
        err = parse_vlan_from_nlattrs(match, &mask_attrs, a, true, log);
        if (err)
            goto free_newmask;

        //解析key，填充到match->mask
        err = ovs_key_from_nlattrs(net, match, mask_attrs, a, true,
                       log);
        if (err)
            goto free_newmask;
    }

    //字段匹配的信息进行校验
    if (!match_validate(match, key_attrs, mask_attrs, log))
        err = -EINVAL;

free_newmask:
    kfree(newmask);
    return err;
}
```

注意到这个函数可以分为两部分，一部分是对sw_flow_key信息的解析，一部分是对sw_flow_mask信息的解析，传入的参数中，nla_key为含有sw_flow_key信息的nlattr结构体，nla_mask为含有sw_flow_mask信息的nlattr结构体。

以下为具体的步骤：

1. 提取nla_key中的信息，放到一个nlattr结构体数组a中，数组中的每个元素对应为一类属性信息；
2. 调用parse_vlan_from_nlattrs，解析a中vlan信息填充到sw_flow_key；
3. 调用ovs_key_from_nlattrs，将剩余的其他信息填充到sw_flow_key；
4. 提取nla_mask中的信息，放到一个nlattr结构体数组a中，数组中的每个元素对应为一类属性信息；
5. 调用parse_vlan_from_nlattrs，解析a中vlan信息填充到sw_flow_mask；
6. 调用ovs_key_from_nlattrs，将剩余的其他信息填充到sw_flow_mask；
7. 调用match_validate，对上述提取的信息进行验证；

ovs_key_from_nlattrs

ovs_nla_get_match函数中，不同类型的属性的解析稍微有点不同，其中VLAN信息的解析较为复杂，需要学习相关VLAN的知识，其他属性的解析较为简单，但是基本的逻辑是没有多大的差距的，在这里具体分析一下ovs_key_from_nlattrs函数的处理过程。

```
//从nlattr中提取信息填充到ovs_flow_key中
static int ovs_key_from_nlattrs(struct net *net, struct sw_flow_match *match,
                u64 attrs, const struct nlattr **a,
                bool is_mask, bool log)
{
    int err;
```

```
    err = metadata_from_nlattrs(net, match, &attrs, a, is_mask, log); //解析元数据
    if (err)
        return err;

    // 解析以太网信息
    if (attrs & (1ULL << OVS_KEY_ATTR_ETHERNET)) {
        const struct ovs_key_ethernet *eth_key;

        //解析以太网源和目标地址
        eth_key = nla_data(a[OVS_KEY_ATTR_ETHERNET]);
        SW_FLOW_KEY_MEMCPY(match, eth.src,
                eth_key->eth_src, ETH_ALEN, is_mask);
        SW_FLOW_KEY_MEMCPY(match, eth.dst,
                eth_key->eth_dst, ETH_ALEN, is_mask);
        attrs &= ~(1ULL << OVS_KEY_ATTR_ETHERNET);

        if (attrs & (1ULL << OVS_KEY_ATTR_VLAN)) {
            /* VLAN attribute is always parsed before getting here since it
             * may occur multiple times.
             */
            OVS_NLERR(log, "VLAN attribute unexpected.");
            return -EINVAL;
        }

        if (attrs & (1ULL << OVS_KEY_ATTR_ETHERTYPE)) {
            err = parse_eth_type_from_nlattrs(match, &attrs, a, is_mask, //解析以太网类型
                                log);
            if (err)
                return err;
        } else if (!is_mask) {
            SW_FLOW_KEY_PUT(match, eth.type, htons(ETH_P_802_2), is_mask);
        }
    } else if (!match->key->eth.type) {
        OVS_NLERR(log, "Either Ethernet header or EtherType is required.");
        return -EINVAL;
    }

    // 解析IPV4
    if (attrs & (1 << OVS_KEY_ATTR_IPV4)) {
        const struct ovs_key_ipv4 *ipv4_key;

        ipv4_key = nla_data(a[OVS_KEY_ATTR_IPV4]);
        if (!is_mask && ipv4_key->ipv4_frag > OVS_FRAG_TYPE_MAX) { //OVS_FRAG_TYPE_MAX
ip分片类型的数量
            OVS_NLERR(log, "IPv4 frag type %d is out of range max %d",
                  ipv4_key->ipv4_frag, OVS_FRAG_TYPE_MAX);
            return -EINVAL;
        }
        SW_FLOW_KEY_PUT(match, ip.proto,
                ipv4_key->ipv4_proto, is_mask);
        SW_FLOW_KEY_PUT(match, ip.tos,
                ipv4_key->ipv4_tos, is_mask);
```

```
        SW_FLOW_KEY_PUT(match, ip.ttl,
                ipv4_key->ipv4_ttl, is_mask);
        SW_FLOW_KEY_PUT(match, ip.frag,
                ipv4_key->ipv4_frag, is_mask);
        SW_FLOW_KEY_PUT(match, ipv4.addr.src,
                ipv4_key->ipv4_src, is_mask);
        SW_FLOW_KEY_PUT(match, ipv4.addr.dst,
                ipv4_key->ipv4_dst, is_mask);
        attrs &= ~(1 << OVS_KEY_ATTR_IPV4);
    }

    // 解析IPV6
    if (attrs & (1ULL << OVS_KEY_ATTR_IPV6)) {
        const struct ovs_key_ipv6 *ipv6_key;

        ipv6_key = nla_data(a[OVS_KEY_ATTR_IPV6]);
        if (!is_mask && ipv6_key->ipv6_frag > OVS_FRAG_TYPE_MAX) {
            OVS_NLERR(log, "IPv6 frag type %d is out of range max %d",
                  ipv6_key->ipv6_frag, OVS_FRAG_TYPE_MAX);
            return -EINVAL;
        }

        if (!is_mask && ipv6_key->ipv6_label & htonl(0xFFF00000)) {
            OVS_NLERR(log, "IPv6 flow label %x is out of range (max=%x)",
                  ntohl(ipv6_key->ipv6_label), (1 << 20) - 1);
            return -EINVAL;
        }

        SW_FLOW_KEY_PUT(match, ipv6.label,
                ipv6_key->ipv6_label, is_mask);
        SW_FLOW_KEY_PUT(match, ip.proto,
                ipv6_key->ipv6_proto, is_mask);
        SW_FLOW_KEY_PUT(match, ip.tos,
                ipv6_key->ipv6_tclass, is_mask);
        SW_FLOW_KEY_PUT(match, ip.ttl,
                ipv6_key->ipv6_hlimit, is_mask);
        SW_FLOW_KEY_PUT(match, ip.frag,
                ipv6_key->ipv6_frag, is_mask);
        SW_FLOW_KEY_MEMCPY(match, ipv6.addr.src,
                ipv6_key->ipv6_src,
                sizeof(match->key->ipv6.addr.src),
                is_mask);
        SW_FLOW_KEY_MEMCPY(match, ipv6.addr.dst,
                ipv6_key->ipv6_dst,
                sizeof(match->key->ipv6.addr.dst),
                is_mask);

        attrs &= ~(1ULL << OVS_KEY_ATTR_IPV6);
    }

    // 解析arp
    if (attrs & (1ULL << OVS_KEY_ATTR_ARP)) {
        const struct ovs_key_arp *arp_key;
```

```
        arp_key = nla_data(a[OVS_KEY_ATTR_ARP]);
        if (!is_mask && (arp_key->arp_op & htons(0xff00))) {
            OVS_NLERR(log, "Unknown ARP opcode (opcode=%d).",
                  arp_key->arp_op);
            return -EINVAL;
        }

        SW_FLOW_KEY_PUT(match, ipv4.addr.src,
                arp_key->arp_sip, is_mask);
        SW_FLOW_KEY_PUT(match, ipv4.addr.dst,
            arp_key->arp_tip, is_mask);
        SW_FLOW_KEY_PUT(match, ip.proto,
                ntohs(arp_key->arp_op), is_mask);
        SW_FLOW_KEY_MEMCPY(match, ipv4.arp.sha,
                arp_key->arp_sha, ETH_ALEN, is_mask);
        SW_FLOW_KEY_MEMCPY(match, ipv4.arp.tha,
                arp_key->arp_tha, ETH_ALEN, is_mask);

        attrs &= ~(1ULL << OVS_KEY_ATTR_ARP);
    }

    //解析nsh
    if (attrs & (1 << OVS_KEY_ATTR_NSH)) {
        if (nsh_key_put_from_nlattr(a[OVS_KEY_ATTR_NSH], match,
                        is_mask, false, log) < 0)
            return -EINVAL;
        attrs &= ~(1 << OVS_KEY_ATTR_NSH);
    }

    //解析MPLS协议
    if (attrs & (1ULL << OVS_KEY_ATTR_MPLS)) {
        const struct ovs_key_mpls *mpls_key;

        mpls_key = nla_data(a[OVS_KEY_ATTR_MPLS]);
        SW_FLOW_KEY_PUT(match, mpls.top_lse,
                mpls_key->mpls_lse, is_mask);

        attrs &= ~(1ULL << OVS_KEY_ATTR_MPLS);
    }

    //解析TCP协议
    if (attrs & (1ULL << OVS_KEY_ATTR_TCP)) {
        const struct ovs_key_tcp *tcp_key;

        tcp_key = nla_data(a[OVS_KEY_ATTR_TCP]);
        SW_FLOW_KEY_PUT(match, tp.src, tcp_key->tcp_src, is_mask);
        SW_FLOW_KEY_PUT(match, tp.dst, tcp_key->tcp_dst, is_mask);
        attrs &= ~(1ULL << OVS_KEY_ATTR_TCP);
    }

    //解析TCP flag
    if (attrs & (1ULL << OVS_KEY_ATTR_TCP_FLAGS)) {
```

```
        SW_FLOW_KEY_PUT(match, tp.flags,
                nla_get_be16(a[OVS_KEY_ATTR_TCP_FLAGS]),
                is_mask);
        attrs &= ~(1ULL << OVS_KEY_ATTR_TCP_FLAGS);
    }

    //解析UDP
    if (attrs & (1ULL << OVS_KEY_ATTR_UDP)) {
        const struct ovs_key_udp *udp_key;

        udp_key = nla_data(a[OVS_KEY_ATTR_UDP]);
        SW_FLOW_KEY_PUT(match, tp.src, udp_key->udp_src, is_mask);
        SW_FLOW_KEY_PUT(match, tp.dst, udp_key->udp_dst, is_mask);
        attrs &= ~(1ULL << OVS_KEY_ATTR_UDP);
    }

    //解析SCTP
    if (attrs & (1ULL << OVS_KEY_ATTR_SCTP)) {
        const struct ovs_key_sctp *sctp_key;

        sctp_key = nla_data(a[OVS_KEY_ATTR_SCTP]);
        SW_FLOW_KEY_PUT(match, tp.src, sctp_key->sctp_src, is_mask);
        SW_FLOW_KEY_PUT(match, tp.dst, sctp_key->sctp_dst, is_mask);
        attrs &= ~(1ULL << OVS_KEY_ATTR_SCTP);
    }

    //解析ICMP
    if (attrs & (1ULL << OVS_KEY_ATTR_ICMP)) {
        const struct ovs_key_icmp *icmp_key;

        icmp_key = nla_data(a[OVS_KEY_ATTR_ICMP]);
        SW_FLOW_KEY_PUT(match, tp.src,
                htons(icmp_key->icmp_type), is_mask);
        SW_FLOW_KEY_PUT(match, tp.dst,
                htons(icmp_key->icmp_code), is_mask);
        attrs &= ~(1ULL << OVS_KEY_ATTR_ICMP);
    }

    //解析ICMPV6
    if (attrs & (1ULL << OVS_KEY_ATTR_ICMPV6)) {
        const struct ovs_key_icmpv6 *icmpv6_key;

        icmpv6_key = nla_data(a[OVS_KEY_ATTR_ICMPV6]);
        SW_FLOW_KEY_PUT(match, tp.src,
                htons(icmpv6_key->icmpv6_type), is_mask);
        SW_FLOW_KEY_PUT(match, tp.dst,
                htons(icmpv6_key->icmpv6_code), is_mask);
        attrs &= ~(1ULL << OVS_KEY_ATTR_ICMPV6);
    }

    //解析OVS_KEY_ATTR_ND
    if (attrs & (1ULL << OVS_KEY_ATTR_ND)) {
        const struct ovs_key_nd *nd_key;
```

```
        nd_key = nla_data(a[OVS_KEY_ATTR_ND]);
        SW_FLOW_KEY_MEMCPY(match, ipv6.nd.target,
            nd_key->nd_target,
            sizeof(match->key->ipv6.nd.target),
            is_mask);
        SW_FLOW_KEY_MEMCPY(match, ipv6.nd.sll,
            nd_key->nd_sll, ETH_ALEN, is_mask);
        SW_FLOW_KEY_MEMCPY(match, ipv6.nd.tll,
                nd_key->nd_tll, ETH_ALEN, is_mask);
        attrs &= ~(1ULL << OVS_KEY_ATTR_ND);
    }

    if (attrs != 0) {
        OVS_NLERR(log, "Unknown key attributes %llx",
              (unsigned long long)attrs);
        return -EINVAL;
    }

    return 0;
}
```

上述代码是具体的过程，可以看到流程很简单，首先调用metadata_from_nlattrs对元数据进行解析，这里的元数据指的是流的进入端口、skb的掩码信息等，这些信息也会被放在nlattr中，作为不同的属性，可以类比sw_flow_key中的tcp、ip信息，在逻辑处理上是一样的；然后就是按照网络的层次，对ip、tcp、udp等这些信息进行解析。

> 注意到SW_FLOW_KEY_PUT和SW_FLOW_KEY_MEMCPY函数，这两个函数是具体的把nlattr中的信息放入sw_flow_key中对应的字段中，在填充key和mask的时候都会调用该函数，不同的是，这两个函数中有一个很重要的参数is_mask，如果是解析mask这个值为true，然后将信息填充到match->sw_flow_mask->sw_flow_key中，否则将信息填充到match->sw_flow_key中。
>
> sw_flow_match包装了我们要提取的sw_flow_key和sw_flow_mask

ovs_flow_mask_key

再回到ovs_flow_cmd_new函数中，填充完信息之后，一个很重要的步骤是调用ovs_nla_get_identifier判断flow（流表项）的标识，一条flow的标识有两种，一种是通过ufid进行标识，一种是没有ufid则使用未经掩码过的sw_flow_key进行标识，具体的不同的flow处理会在下面的步骤中进行分析。

如果一条flow不是通过ufid进行标识，则会复制一份sw_flow_key（这个时候的key是刚刚提取出来的，未经过任何的处理，也就是未掩码），将复制后的结果复制给flow->sw_flow_id->sw_flow_key这个结构体下，也就是flow->id->unmasked_key成员。

接下来就是对flow->sw_flow_key结构体内容调用ovs_flow_mask_key进行掩码操作，具体代码如下：

```
//掩码操作，其实就是key和mask的内容进行'与'操作
//dst存储掩码结果的key值，src是要进行掩码操作的key值，mask是掩码的信息
//当full为true时，dst会完全的初始化，当full为false时，dst只会初始化（mask->start，mask->end）之间的部分
void ovs_flow_mask_key(struct sw_flow_key *dst, const struct sw_flow_key *src,
               bool full, const struct sw_flow_mask *mask)
{
```

```
    int start = full ? 0 : mask->range.start;
    int len = full ? sizeof *dst : range_n_bytes(&mask->range);  //mask->range的长度或者
dst的长度
    const long *m = (const long *)((const u8 *)&mask->key + start);
    const long *s = (const long *)((const u8 *)src + start);
    long *d = (long *)((u8 *)dst + start);
    int i;

    /* If 'full' is true then all of 'dst' is fully initialized. Otherwise,
     * if 'full' is false the memory outside of the 'mask->range' is left
     * uninitialized. This can be used as an optimization when further
     * operations on 'dst' only use contents within 'mask->range'.
     */
    for (i = 0; i < len; i += sizeof(long))
        *d++ = *s++ & *m++;
}
```

经过处理之后接下来flow->sw_flow_key就是经过sw_flow_mask掩码处理之后的key内容了

ovs_nla_copy_actions

接下来就是对nlattr中OVS_FLOW_ATTR_ACTIONS类型的属性信息进行分析，也就是对应到sw_flow_action结构体，在ovs_nla_copy_actions函数中，主要对函数该类属性的内容进行验证，并进行提取，具体的源代码如下：

```
```

__ovs_nla_copy_actions函数中的内容十分复杂，其中涉及到对不同类型的action进行验证，在这里不具体展开。

get_dp

接下来，调用get_dp函数，根据dp_ifindex在内核中查找对应的datapath，这个函数中涉及到具体的设备处理，其中包括通过网络命名空间和接口索引ifindex寻找对应的网络设备，在这里不具体展开，将会在之后的博客中介绍相关的内容。

ovs_flow_tbl_lookup_ufid、ovs_flow_tbl_lookup

接下来就是查找datapath中对应的流表，判断是否已经存在该流表项，这个涉及到具体的流表项是怎么存储的，可以参看博文，了解相关的ovs结构体知识。

在上文中也提到过，在ovs中，ufid标识的flow会存放在datapath的ti流表和uti流表中，而未经ufid标识的flow只会存放在uti流表中，下面分别介绍：

```
```

上面代码是查找uti流表，可以看到经过ufid标识的流表项查找过程，是直接将ufid经过hash之后，查找uti中对应的哈希桶的，然后直接在这个hash桶中进行查找。

```
```

```
// ------------------ovs_flow_tbl_lookup-------------------------
//查找table，这里面查找的是table->ti表格
struct sw_flow *ovs_flow_tbl_lookup(struct flow_table *tbl,
                    const struct sw_flow_key *key)
{
    struct table_instance *ti = rcu_dereference_ovsl(tbl->ti);
    struct mask_array *ma = rcu_dereference_ovsl(tbl->mask_array);
    u32 __always_unused n_mask_hit;
    u32 index = 0;

    return flow_lookup(tbl, ti, ma, key, &n_mask_hit, &index);
}

//------------------------flow_lookup--------------------------
/***
 主要的流程：其中参数index输入的是最有可能直接成功匹配的mask的索引（在flow_table->mask_arry中，也就是ma）
 1. 获得index对应的mask，调用masked_flow_lookup获得匹配的流表项
 2. 如果上一步没有返回结果，则从头遍历flow_table->mask_arry，每次调用masked_flow_lookup，直到找到流表项为止
 ***/
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)      //index应该是最终返回的能够匹配上的匹配域（ma）的索引
{
    struct sw_flow_mask *mask;
    struct sw_flow *flow;
    int i;

    if (*index < ma->max) {
        mask = rcu_dereference_ovsl(ma->masks[*index]);     //取出index的匹配域
        if (mask) {
            flow = masked_flow_lookup(ti, key, mask, n_mask_hit);       //匹配流表项
            if (flow)
                return flow;            //index中刚好找到对应的流表项就直接返回
        }
    }

    //否则遍历table中的匹配域【之前的版本是没有上面一步，直接遍历table中的匹配域】
    for (i = 0; i < ma->max; i++)  {

        if (i == *index)        //上面已经操作过了
            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;
            return flow;
        }
    }

    return NULL;
}

//--------------------------masked_flow_lookup------------------
/***
 查看当前的匹配域mask中的流表项是否能匹配unmasked
 1. 调用ovs_flow_mask_key，进行‘与’操作，获得key中对应range的结果
 2. 利用上一步返回的结果做hash
 3. 调用find_bucket，该函数中会再做一次hash，返回对应的桶（哈希头）--- 其实也就是sw_flow流表项链表的头指针
 4. 遍历流表项链表，判断每个流表项是否和unmasked匹配，返回第一个被匹配到的流表项
 ***/
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;

    ovs_flow_mask_key(&masked_key, unmasked, false, mask);  //unmasked 与 mask->key做‘与’操作，返回在masked_key中
    hash = flow_hash(&masked_key, &mask->range);        //利用masked_key做一个hash

    // 调用find_bucket通过hash值查找hash所在的哈希头
    // 因为openVswitch中有多条流表项链表，所以要先查找出要匹配的流表在哪个链表中，然后再去遍历该链表【一个流表中有多条流表项链表】
    head = find_bucket(ti, hash);       //找到ti表中对应的哈希头
    (*n_mask_hit)++;
    hlist_for_each_entry_rcu(flow, head, flow_table.node[ti->node_ver]) {   //这个flow_table就是sw_flow->flow_table，可以指向两个流表项链表的表头
        if (flow->mask == mask && flow->flow_table.hash == hash &&
            flow_cmp_masked_key(flow, &masked_key, &mask->range))
            return flow;
    }
    return NULL;
}
```

上述代码是具体的ti表查找过程，在ovs_flow_tbl_lookup函数中，先调用flow_lookup函数，然后flow_lookup函数最终调用masked_flow_lookup函数，流程较为复杂，具体的过程不再详细介绍，在这里只介绍跟本博文相关的部分。 在masked_flow_lookup函数中，可以看到与查找uti表不同的是，hash的内容不同，在查找ti标时，首先对key和mask进行掩码操作（按照上述的流程，这里的key应该是已经进行了掩码操作了，这个步骤感觉有点多余，因为这个函数在其他代码中被使用到，可能就是直接这么写的，经过掩码之后的key再经过掩码，其key的内容也不会变化），然后对经过掩码的key的内容做一个hash，用这个hash找到ti中的一个哈希桶，最后在这个哈希桶中进行查找。

> 关于masked_flow_lookup查找的原理，可以参考我的博文ovs源码阅读--元组空间搜索算法，这里面用到了TTS算法，其中flow就是TTS中的rule，mask对应tuple，而掩码之后经过hash的值就是TTS中的key值。

ovs_flow_tbl_insert

如果上一步找到了对应的流表，则调用ovs_flow_tbl_insert函数，将flow插入到对应的流表中，具体代码如下：

```
//插入新的表项
int ovs_flow_tbl_insert(struct flow_table *table, struct sw_flow *flow,
            const struct sw_flow_mask *mask)
{
    int err;

    err = flow_mask_insert(table, flow, mask); //将mask插入mask_array中
    if (err)
        return err;
    flow_key_insert(table, flow); //将flow插入table中，这里插入的是在flow_table->ti内
    if (ovs_identifier_is_ufid(&flow->id))
        flow_ufid_insert(table, flow); //将flow插入table中，这里插入的是在flow_table->uti内

    return 0;
}
```

在ovs_flow_tbl_insert代码中，先调用flow_mask_insert将mask的信息保存到table的mask_array表中，然后调用flow_key_insert将key的信息保存到ti表，最后判断flow是否为ufid标识，如果是还要调用flow_ufid_insert函数将该flow保存到uti表。

flow_mask_insert

```
//插入mask
static int flow_mask_insert(struct flow_table *tbl, struct sw_flow *flow,
                const struct sw_flow_mask *new)
{
    struct sw_flow_mask *mask;

    mask = flow_mask_find(tbl, new); //查找是否已经存在该mask
    if (!mask) {    //如果不存在，则插入
        struct mask_array *ma;
        int i;

        /* Allocate a new mask if none exsits. */
        mask = mask_alloc(); //申请一个新的mask
        if (!mask)
            return -ENOMEM;

        mask->key = new->key;
        mask->range = new->range;

        //获得mask_array
        ma = ovsl_dereference(tbl->mask_array);
        //mask_array中实际长度大于最大长度，则重新分配mask_array空间（扩容）
        if (ma->count >= ma->max) {
            int err;
```

```
            err = tbl_mask_array_realloc(tbl, ma->max +
                                MASK_ARRAY_SIZE_MIN); // 给tbl->mask_array重新分配
            if (err) {
                kfree(mask);
                return err;
            }
            ma = ovsl_dereference(tbl->mask_array);
        }

        //找到一个空位插入
        for (i = 0; i < ma->max; i++) {
            struct sw_flow_mask *t;

            t = ovsl_dereference(ma->masks[i]);
            if (!t) {
                rcu_assign_pointer(ma->masks[i], mask);
                ma->count++;
                break;
            }
        }

    } else {
        BUG_ON(!mask->ref_count);
        mask->ref_count++;
    }

    flow->mask = mask;
    return 0;
}
```

该流程较为简单，就是从mask_array数组中获得一个空的位置，将mask信息写入即可

flow_key_insert

```
//将flow插入table中，这里插入的是在flow_table->ti内
static void flow_key_insert(struct flow_table *table, struct sw_flow *flow)
{
    struct table_instance *new_ti = NULL;
    struct table_instance *ti;

    flow->flow_table.hash = flow_hash(&flow->key, &flow->mask->range); //生成hash
    ti = ovsl_dereference(table->ti);
    table_instance_insert(ti, flow); //将flow插入ti对应位置的链表中
    table->count++;

    /* Expand table, if necessary, to make room. */
    //表空间不够或者超过一定的时间，则就重新扩容
    if (table->count > ti->n_buckets)
        new_ti = table_instance_expand(ti, false); //扩容
    else if (time_after(jiffies, table->last_rehash + REHASH_INTERVAL))
        new_ti = table_instance_rehash(ti, ti->n_buckets, false);
```

```
    if (new_ti) {
        rcu_assign_pointer(table->ti, new_ti);
        call_rcu(&ti->rcu, flow_tbl_destroy_rcu_cb);
        table->last_rehash = jiffies;
    }
}

//将flow插入ti对应位置的链表中
static void table_instance_insert(struct table_instance *ti,
                  struct sw_flow *flow)
{
    struct hlist_head *head;

    head = find_bucket(ti, flow->flow_table.hash); //通过hash找到对应的位置
    hlist_add_head_rcu(&flow->flow_table.node[ti->node_ver], head);
}
```

flow_key_insert函数中，将flow插入到ti表，与查询ti表的步骤类似，先对key经过掩码操作，然后对掩码后的内容进行hash，利用调用table_instance_insert利用hash查找要插入的位置并插入。

```
```

flow_ufid_insert

如果flow是ufid标识的，还要将该flow插入到uti表中，具体代码如下：

```
//将flow插入table中，这里插入的是在flow_table->uti内
static void flow_ufid_insert(struct flow_table *table, struct sw_flow *flow)
{
    struct table_instance *ti;

    flow->ufid_table.hash = ufid_hash(&flow->id);
    ti = ovsl_dereference(table->ufid_ti);
    ufid_table_instance_insert(ti, flow);
    table->ufid_count++;

    /* Expand table, if necessary, to make room. */
    //同样的，如果需要就扩容
    if (table->ufid_count > ti->n_buckets) {
        struct table_instance *new_ti;

        new_ti = table_instance_expand(ti, true);
        if (new_ti) {
            rcu_assign_pointer(table->ufid_ti, new_ti);
            call_rcu(&ti->rcu, flow_tbl_destroy_rcu_cb);
        }
    }
}
```

跟查找uti表的逻辑类似，不再赘述

action更新

回到ovs_flow_tbl_lookup_ufid或者ovs_flow_tbl_lookup处理之后，如果查到对应的flow，表明datapath中存在对应的flow，则直接将查到的flow中sw_flow_action结构体中的指针指向经过netlink消息填充之后的action结构体，对action进行更新。

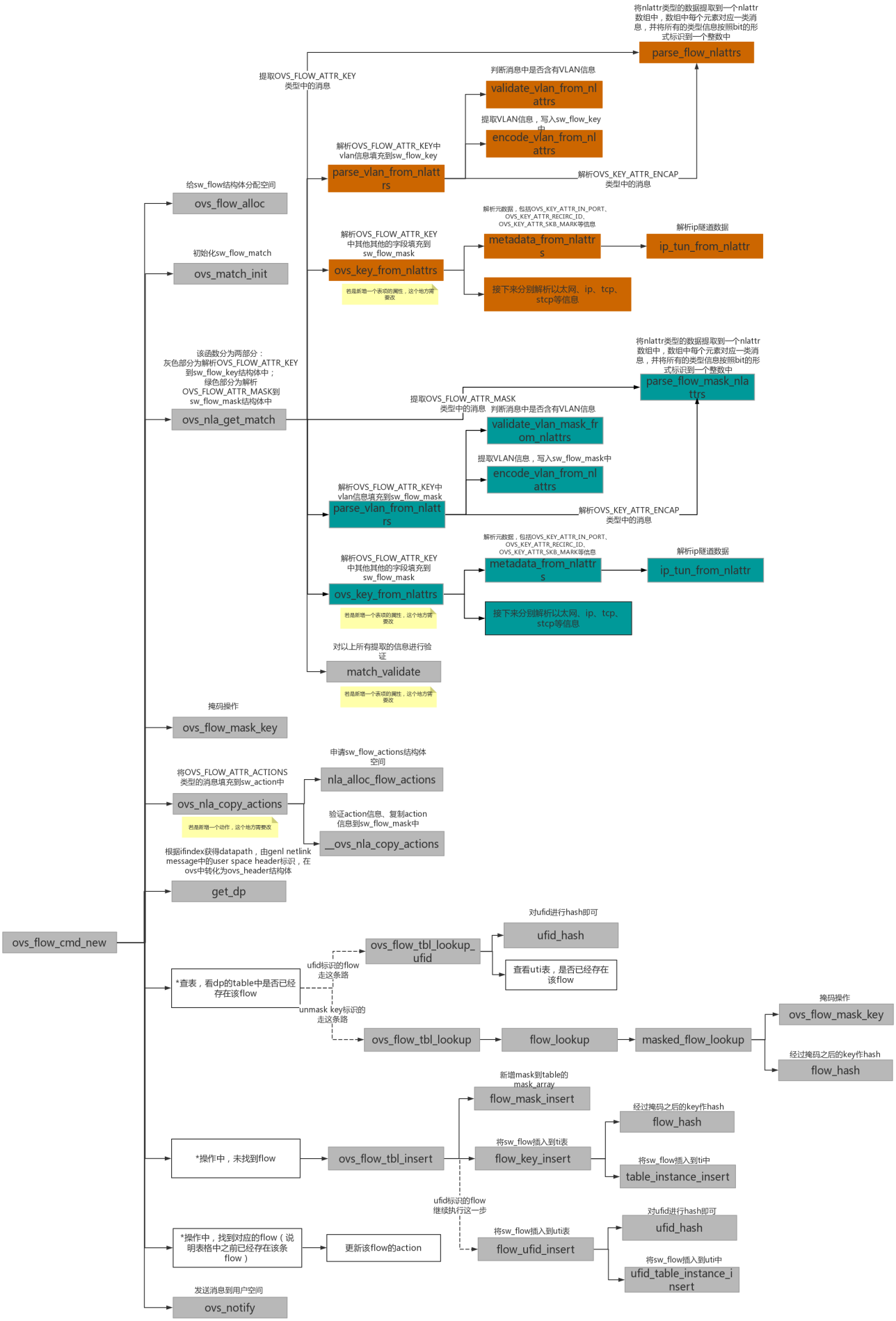
在ovs_flow_cmd_new代码中，更新action之前还经过一些判断，对不能确定的情况进行确认，可以参看代码注释，在这里就不详细赘述。

ovs_notify

最终，构造要返回给用户空间的信息，通过ovs_notify函数进行发送，关于这一方面的内容，将在后续的博文中进行更新。

总结

在这里通过一张图总结新增流表项的过程，如下图（点击可查看大图）


ovs_flow_cmd_new
get_dp
根据ifindex获得datapath，由genl netlink message中的user space header标识，在ovs中转化为ovs_header结构体
ovs_flow_alloc
给sw_flow结构体分配空间
ovs_match_init
初始化sw_flow_match
ovs_nla_get_match
该函数分为两部分：
灰色部分为解析OVS_FLOW_ATTR_KEY到sw_flow_key结构体中；
绿色部分为解析OVS_FLOW_ATTR_MASK到sw_flow_mask结构体中
ovs_flow_mask_key
掩码操作
ovs_nla_copy_actions
将OVS_FLOW_ATTR_ACTIONS类型的消息填充到sw_action中
若是新增一个动作，这个地方需要改
nla_alloc_flow_actions
申请sw_flow_actions结构体空间
__ovs_nla_copy_actions
验证action信息、复制action信息到sw_flow_mask中
ovs_notify
发送消息到用户空间
提取OVS_FLOW_ATTR_KEY类型中的消息
parse_flow_nlattrs
将nlattr类型的数据提取到一个nlattr数组中，数组中每个元素对应一类消息，并将所有的类型信息按照bit的形式标识到一个整数中
parse_vlan_from_nlattrs
解析OVS_FLOW_ATTR_KEY中vlan信息填充到sw_flow_key
validate_vlan_from_nlattrs
判断消息中是否含有VLAN信息
encode_vlan_from_nlattrs
提取VLAN信息，写入sw_flow_key中
解析OVS_KEY_ATTR_ENCAP类型中的消息
ovs_key_from_nlattrs
解析OVS_FLOW_ATTR_KEY中其他其他的字段填充到sw_flow_mask
若是新增一个表项的属性，这个地方需要改
metadata_from_nlattrs
解析元数据，包括OVS_KEY_ATTR_IN_PORT、OVS_KEY_ATTR_RECIRC_ID、OVS_KEY_ATTR_SKB_MARK等信息
ip_tun_from_nlattr
解析ip隧道数据
接下来分别解析以太网、ip、tcp、stcp等信息
match_validate
对以上所有提取的信息进行验证
若是新增一个表项的属性，这个地方需要改
提取OVS_FLOW_ATTR_MASK类型中的消息
parse_flow_mask_nlattrs
将nlattr类型的数据提取到一个nlattr数组中，数组中每个元素对应一类消息，并将所有的类型信息按照bit的形式标识到一个整数中
parse_vlan_from_nlattrs
解析OVS_FLOW_ATTR_KEY中vlan信息填充到sw_flow_mask
validate_vlan_mask_from_nlattrs
判断消息中是否含有VLAN信息
encode_vlan_from_nlattrs
提取VLAN信息，写入sw_flow_mask中
解析OVS_KEY_ATTR_ENCAP类型中的消息
ovs_key_from_nlattrs
解析OVS_FLOW_ATTR_KEY中其他其他的字段填充到sw_flow_mask
若是新增一个表项的属性，这个地方需要改
metadata_from_nlattrs
解析元数据，包括OVS_KEY_ATTR_IN_PORT、OVS_KEY_ATTR_RECIRC_ID、OVS_KEY_ATTR_SKB_MARK等信息
ip_tun_from_nlattr
解析ip隧道数据
接下来分别解析以太网、ip、tcp、stcp等信息
*查表，看dp的table中是否已经存在该flow
ufid标识的flow走这条路
ovs_flow_tbl_lookup_ufid
对ufid进行hash即可
ufid_hash
查看uti表，是否已经存在该flow
unmask key标识的走这条路
ovs_flow_tbl_lookup
flow_lookup
masked_flow_lookup
掩码操作
ovs_flow_mask_key
经过掩码之后的key作hash
flow_hash
*操作中，未找到flow
ovs_flow_tbl_insert
新增mask到table的mask_array
flow_mask_insert
将sw_flow插入到ti表
flow_key_insert
经过掩码之后的key作hash
flow_hash
将sw_flow插入到ti中
table_instance_insert
ufid标识的flow继续执行这一步
*操作中，找到对应的flow（说明表格中之前已经存在该条flow）
更新该flow的action
将sw_flow插入到uti表
flow_ufid_insert
对ufid进行hash即可
ufid_hash
将sw_flow插入到uti中
ufid_table_instance_insert

> 该流程图中，上下表示调用的先后关系（如在ovs_flow_cmd_new中，ovs_nla_get_match比ovs_flow_mask_key先调用，则ovs_nla_get_match在ovs_flow_mask_key上面），箭头表示调用关系