大专栏

OpenvSwitch 流表转换

2019-10-01 | 前端 | 没有评论

“感谢打赏”

推荐看一下这篇文章，讲述了各个流表，我们这里着重讲流程和代码，对流表不再细说。

我们主要的关注点还是OVS-DPDK的流表转换，其实和OVS的转换差不多，只不过OVS的Datapath流表位于kernel，报文在Datapath找不到流表即通过netlink上传到Userspace，而OVS-DPDK则是Datapath流表依然位于Userspace，可以看做是一个缓存。查找不到的话直接继续调用其他接口查找Userspace的流表。

- controller会根据网络情况给ovs下发流表，或者命令ovs-ofctl，属于Userspace的流表(ofproto classifier)。
- 当报文来的时候会先提取key值，然后在Datapath的流表(EMC或者microflow)进行查找匹配，查找到之后会进行action操作
- 如果没有查找到，就会转而继续查找Datapath的流表(也叫dpcls、TSS classifier、megaflow)。
- dpcls有好多的子表，根据掩码来分类，需要挨个查找每个子表，如果查找到就会讲带掩码的流表转换成精确匹配的流表，然后匹配转发
- 如果匹配不到就会将key和报文传递给Userspace进行匹配
- Userspace的查找会根据优先级和对每个table进行查找，然后执行相应的操作，最后匹配的所有流表会进行组合生成更简单的一些流表，比如table 0有n0条流表，一直到table 24有n24条流表，那么最终生成的dpcls流表可能有n1 x n2 x … x n24中可能性
- 将生成的流表转换安装到dpcls，然后转换安装到EMC
- 如果匹配不到的话会丢弃或者上报packet in给controller

边界点

- EMC是以pmd为边界的，每个pmd都有自己的一套EMC

- dpcls是以端口边界的，每个端口都有自己的dpcls流表
- ofproto classifier是以桥为边界的，每个桥都有自己的流表

流表下发

其实前面我们说到了三种存在形式的流表，这里流表下发只是下到了ofproto classifier了，其他的都是需要报文去触发去上一级拉取相应的流表。

流表发送

流表下发一般是两种方式:

- controller，根据情况生成流表，通过openflow协议下发flow mod给ovs的Userspace流表。
- 命令ovs-ofctl，这个是根据命令情况生成流表，通过openflow协议下发flow mod给ovs的Userspace流表。

流表下发我们就先不去看了，因为我目前的原则是操作命令先不看，先看服务，controller后面会看ovn的，到时候单独来写。

流表接收

流表接收是指将命令行或者controller下发的流表接收，并且暂存。

服务启动

`ovs-vswitchd.c` 的路径为 `main-->bridge_run-->bridge_run__(针对每个桥运行ofproto)-->ofproto_run(信息量大，暂时忽略)-->handle_openflow`

handle_openflow-->handle_openflow__

- 提取openflow协议的类型
- 简单介绍下部分openflow的协议类型
  - ECHO是握手
  - FEATURES是同步版本和特性
  - CONFIG是同步配置
  - PACKET_OUT发包
  - PORT_MOD修改端口信息
  - **FLOW_MOD修改流表，这个是最关键的**
  - GROUP_MOD, TABLE_MOD, METER_MOD
- 最重要的 `OFPTYPE_FLOW_MOD` 的操作为 `handle_flow_mod` 主要是对流表的操作

handle_flow_mod

- `ofputil_decode_flow_mod` 主要是将FLOW_MOD信息解析到 `ofputil_flow_mod` 结构中
- `handle_flow_mod__-->ofproto_flow_mod_init` 主要是将上面 `ofputil_flow_mod` 结构的数据解析到结构 `ofproto_flow_mod` 中。
- `handle_flow_mod__-->ofproto_flow_mod_start` 支持了流表添加、删除、修改的操作，我们主要关注添加，即 `add_flow_start`

ofproto_flow_mod_init

`ofproto_flow_mod_init` 涉及到数据结构的转换，我们先看下两个数据结构，然后看一下怎么转换的

```
1  struct  {
2      struct ovs_list list_node;
3
4      //匹配项，支持standard和OXM两种类型
```

```
    struct match match;
    //流表优先级
    int priority;
    //添加流表时这两项为0
    ovs_be64 cookie;         /* Cookie bits to match. */
    ovs_be64 cookie_mask;    /* 1-bit in each 'cookie' bit to match. */
    ovs_be64 new_cookie;     /* New cookie to install or UINT64_MAX. */
    bool modify_cookie;      /* Set cookie of existing flow to 'new_cookie'? */
    //流表的table id，只有删除的时候不需要指定
    uint8_t table_id;
    //操作命令，是添加、删除还是修改
    uint16_t command;
    //空闲超时时间，即流表没有报文之后多长时间销毁流表
    uint16_t idle_timeout;
    //硬超时时间，从建立起多长时间销毁流表，不论有木有报文
    uint16_t hard_timeout;
    uint32_t buffer_id;
    //出端口
    ofp_port_t out_port;
    uint32_t out_group;
    //一些操作标识，比如要不要统计等等
    enum ofputil_flow_mod_flags flags;
    uint16_t importance;     /* Eviction precedence. */
    //操作
    struct ofpact *ofpacts;  /* Series of "struct ofpact"s. */
    size_t ofpacts_len;      /* Length of ofpacts, in bytes. */
};
```

将 ofputil_flow_mod 的数据转换为 ofproto_flow_mod 的数据

```
struct ofproto_flow_mod {
    //rculist链表，里面存储了match项，和上面的match项还是有差别的
    //上面的match结构主要是flow和mask两个结构
```

```
    //rule里面的是minimatch，里面包含了上面的两个flow，并且还有flowmap
    //rule的其他信息基本上是从上面拷贝过来的
    struct rule *temp_rule;
    struct rule_criteria criteria;
    //和actions相关的，不太明白
    struct cls_conjunction *conjs;
    size_t n_conjs;

    //以下两条直接从上面数据结构拷贝
    uint16_t command;
    bool modify_cookie;
    //允许添加流表
    bool modify_may_add_flow;
    //取自上面的flags，是否保持统计，false表示重新统计
    bool modify_keep_counts;
    enum nx_flow_update_event event;

    ovs_version_t version;

    bool learn_adds_rule;               /* Learn execution adds a rule. */
    struct rule_collection old_rules;   /* Affected rules. */
    struct rule_collection new_rules;   /* Replacement rules. */
};
```

- 首先直接拷贝过来的结构包括command、modify_cookie
- 然后根据操作的命令执行不同的函数，我先关注添加，即调用 `add_flow_init`
  - 没有指定table_id则指定table_id为0
  - 根据指定的table_id找到数据结构oftable
  - 调用 `cls_rule_init` 和 `ofproto_rule_create` 创建 `ofm->temp_rule` ，这个接下来详细说下，主要是priority和match的填充
  - 调用 `get_conjunctions` 获取根据上面的ofpacts填充下面的conjs，这块还是不太懂，是和action相关的信息

`rule`创建

要看rule创建，我们首先了解一下rule的数据结构，然后看一下当前填充的priority和match

```
struct rule {
    //包含自己的ofproto
    struct ofproto *const ofproto; /* The ofproto that contains this rule. */
    const struct cls_rule cr;      /* In owning ofproto's classifier. */
    //流表中看到的table_id
    const uint8_t table_id;        /* Index in ofproto's 'tables' array. */

    //流表状态，包括初始化、插入和删除
    enum rule_state state;

    //用于释放的引用计数
    struct ovs_refcount ref_count;

    const ovs_be64 flow_cookie; /* Immutable once rule is constructed. */
    struct hindex_node cookie_node OVS_GUARDED_BY(ofproto_mutex);

    enum ofputil_flow_mod_flags flags OVS_GUARDED;

    //对应上面的两个超时
    uint16_t hard_timeout OVS_GUARDED; /* In seconds from ->modified. */
    uint16_t idle_timeout OVS_GUARDED; /* In seconds from ->used. */

    /* Eviction precedence. */
    const uint16_t importance;

    uint8_t removed_reason;

    struct eviction_group *eviction_group OVS_GUARDED_BY(ofproto_mutex);
    struct heap_node evg_node OVS_GUARDED_BY(ofproto_mutex);

    //action操作
    const struct rule_actions * const actions;
```

```
33
34      /* In owning meter's 'rules' list.  An empty list if there is no meter. */
35      struct ovs_list meter_list_node OVS_GUARDED_BY(ofproto_mutex);
36
37      enum nx_flow_monitor_flags monitor_flags OVS_GUARDED_BY(ofproto_mutex);
38      uint64_t add_seqno OVS_GUARDED_BY(ofproto_mutex);
39      uint64_t modify_seqno OVS_GUARDED_BY(ofproto_mutex);
40
41      struct ovs_list expirable OVS_GUARDED_BY(ofproto_mutex);
42
43      long long int created OVS_GUARDED; /* Creation time. */
44
45      long long int modified OVS_GUARDED; /* Time of last modification. */
46  };
```

cls_rule_init

该函数主要是填充的ofm->temp_rule->cr

- `cls_rule_init__` 是将priority填充
- `minimatch_init` 主要是将match填充，主要将struct match中的flow和wc分别填充到struct minimatch的flow和mask

ofproto_rule_create

该函数主要是创建ofm->temp_rule，并且填充一系列的内容，包括上面的new_cookie、idle_timeout、hard_timeout、flags、importance、ofpacts等

- `ofproto->ofproto_class->rule_alloc` 申请rule空间
- rule->ofproto = ofproto指向自己的father
- 初始化引用计数ref_count
- 复制过来new_cookie、idle_timeout、hard_timeout、flags、importance

- 记录创建时间
- 调用 `rule_actions_create` 来填充之前的ofpacts信息
- 调用 `rule_construct` 进行一些信息的初始化

add_flow_start

- `classifier_find_rule_exactly` 从流表指定的table(table_id)的classifier中通过掩码找到子表，在子表中进行匹配，必须找到匹配项，并且优先级和rule版本匹配，则找到流表，否则都是不匹配返回NULL
- 没有匹配到规则，则需要判定规则总数是否超过最大值(UINT_MAX)，超过需要删除一条流表。实际操作就是用新的流表替换掉旧的流表
- 如果找到匹配的规则，说明已有规则，则需要用新的流表替换掉旧的流表
- `replace_rule_start` 主要操作就是新的流表替换旧的流表的操作，如果存在旧流表，则调用 `ofproto_rule_remove__` 删除，然后调用 `ofproto_rule_insert__` 和 `classifier_insert` 添加流表。其中 `classifier_insert` 主要是将rule->cr添加到table->cls中

ofproto_rule_insert__

该函数主要是将rule插入到ofproto中去，上面有了rule的数据结构，下面我们看一下ofproto数据结构存储了什么信息

```
struct ofproto {
    struct hmap_node hmap_node; /* In global 'all_ofprotos' hmap. */
    const struct ofproto_class *ofproto_class;
    char *type;                 /* Datapath type. */
    char *name;                 /* Datapath name. */

    /* Settings. */
    uint64_t fallback_dpid;     /* Datapath ID if no better choice found. */
    uint64_t datapath_id;       /* Datapath ID. */
    bool forward_bpdu;          /* Option to allow forwarding of BPDU frames
                                 * when NORMAL action is invoked. */
    char *mfr_desc;             /* Manufacturer (NULL for default). */
```

```
    char *hw_desc;              /* Hardware (NULL for default). */
    char *sw_desc;              /* Software version (NULL for default). */
    char *serial_desc;          /* Serial number (NULL for default). */
    char *dp_desc;              /* Datapath description (NULL for default). */
    enum ofputil_frag_handling frag_handling;

    /* Datapath. */
    struct hmap ports;          /* Contains "struct ofport"s. */
    struct shash port_by_name;
    struct simap ofp_requests;  /* OpenFlow port number requests. */
    uint16_t alloc_port_no;     /* Last allocated OpenFlow port number. */
    uint16_t max_ports;         /* Max possible OpenFlow port num, plus one. */
    struct hmap ofport_usage;   /* Map ofport to last used time. */
    uint64_t change_seq;        /* Change sequence for netdev status. */

    /* Flow tables. */
    long long int eviction_group_timer; /* For rate limited reheapification. */
    struct oftable *tables;
    int n_tables;
    ovs_version_t tables_version;  /* Controls which rules are visible to
                                    * table lookups. */

    /* Rules indexed on their cookie values, in all flow tables. */
    struct hindex cookies OVS_GUARDED_BY(ofproto_mutex);
    struct hmap learned_cookies OVS_GUARDED_BY(ofproto_mutex);

    /* List of expirable flows, in all flow tables. */
    struct ovs_list expirable OVS_GUARDED_BY(ofproto_mutex);

    /* Meter table.
     * OpenFlow meters start at 1.  To avoid confusion we leave the first
     * pointer in the array un-used, and index directly with the OpenFlow
     * meter_id. */
    struct ofputil_meter_features meter_features;
    struct meter **meters; /* 'meter_features.max_meter' + 1 pointers. */
```

```
48
49      /* OpenFlow connections. */
50      struct connmgr *connmgr;
51
52      int min_mtu;                    /* Current MTU of non-internal ports. */
53
54      /* Groups. */
55      struct cmap groups;               /* Contains "struct ofgroup"s. */
56      uint32_t n_groups[4] OVS_GUARDED; /* # of existing groups of each type. */
57      struct ofputil_group_features ogf;
58
59      /* Tunnel TLV mapping table. */
60      OVSRCU_TYPE(struct tun_table *) metadata_tab;
61
62      /* Variable length mf_field mapping. Stores all configured variable length
63       * meta-flow fields (struct mf_field) in a switch. */
64      struct vl_mff_map vl_mff_map;
65  };
```

- 如果有超时时间的设置，调用 `ovs_list_insert` 将rule->expirable添加到 `ofproto->expirable` 中
- 调用 `cookies_insert` 将 `rule->cookie_node` 插入 `ofproto->cookies`
- `eviction_group_add_rule` 先不管
- 如果有meter配置，调用 `meter_insert_rule`
- 有group的话，调用 `ofproto_group_lookup` 和 `group_add_rule`

EMC查找

首先是报文接收，路径之前我们写过， `pmd_thread_main-->dp_netdev_process_rxq_port-->netdev_rxq_recv-->netdev_dpdk_vhost_rxq_recv-->dp_netdev_input-->dp_netdev_input__` 进行报文的处理。查找到就直接进行操作即可，如果查找不到的话就需要去dpcls进行查找了，找到后调用 `emc_insert` 安装EMC流表。

key值提取

`emc_processing-->miniflow_extract` 会进行key值的提取。这块相对比较简单，我们就不看了，主要就是提取L2、L3、L4的报文协议头。

`emc_processing`

- 因为之前收取报文，一次最多NETDEV_MAX_BURST(32)个报文，所以是循环查表
- `miniflow_extract` 主要是讲报文的信息提取到 `key->mf`
- `dpif_netdev_packet_get_rss_hash` 是获取rss计算的hash值，如果没有计算，则调用 `miniflow_hash_5tuple` 计算出hash值
- `emc_lookup` 主要是在pmd的flowcache中查找表项，必须是hash值、key->mf、并且流表是alive的
- 如果匹配， `dp_netdev_queue_batches` 主要是将报文添加到批处理中
- 如果不匹配，记录下不匹配的报文
- 循环持续到处理完所有的报文
- `dp_netdev_count_packet` 主要是统计一下丢弃的报文、不匹配的报文、和EMC匹配的报文数

dpcls查找

上面EMC查找匹配的报文会放在批处理里面，还会剩下不匹配的报文，接下来会在dpcls中查找。

`fast_path_processing`

- dp_netdev_pmd_lookup_dpcls 根据报文入端口从pmd找出对应的classifier
- 如果找不到classifier，则记录为miss
- 如果找到则继续调用 dpcls_lookup 从各个子表找到合适的流表，只要有一个报文不匹配也记录为miss
- 如果都匹配则继续下面的操作，如果有不匹配的报文，尝试upcall的读锁
- 获取读锁失败，删掉不匹配的报文
- 获取读锁成功，则调用 dp_netdev_pmd_lookup_flow 重新查一下，以防意外收获，查找到了就继续
- 如果依然没有查询到，则调用 handle_packet_upcall 继续调用到ofproto classifier的流表查找。
- 接下来对报文检测已经匹配到流表的，调用 emc_insert 将流表插入EMC中
- 最后调用 dp_netdev_queue_batches 将报文加入批处理中
- 记录丢弃的报文、不匹配的报文、查找的报文和匹配掩码的报文

dpcls_lookup 比较复杂，主要是根据不同的掩码进行子表的区分，然后拿着报文分别去所有的子表用key和mask计算出hash，查看子表中有没有相应的node，如果有的话查看是否有hash冲突链，最终查看是否有匹配key值的表项。我们直接看一下代码

```
1  static bool
2  dpcls_lookup(struct dpcls *cls, const struct netdev_flow_key keys[],
3               struct dpcls_rule **rules, const size_t cnt,
4               int *num_lookups_p)
5  {
6      typedef uint32_t map_type;
7
8
9      struct dpcls_subtable *subtable;
10
11     //keys_map所有位都置1
12     map_type keys_map = TYPE_MAXIMUM(map_type); /* Set all bits. */
13     map_type found_map;
14     uint32_t hashes[MAP_BITS];
15     const struct cmap_node *nodes[MAP_BITS];
16
```

```
17      //清除多余的位，只记录跟报文一样多的位
18      if (cnt != MAP_BITS) {
19          keys_map >>= MAP_BITS - cnt; /* Clear extra bits. */
20      }
21      memset(rules, 0, cnt * sizeof *rules);
22
23      int lookups_match = 0, subtable_pos = 1;
24
25      //dpcls是由众多的subtables组成，当新的规则插入时，子表根据情况动态创建。
26      //每个子表都是根据掩码来区分的，我们通过key和子表的掩码进行计算，
27      //找到匹配的表项，因为不会重复，所以只要找到即可停止
28      //以下就是循环所有子表进行查找
29      PVECTOR_FOR_EACH (subtable, &cls->subtables) {
30          int i;
31
32          //这个循环是找到keys_map是1的最低位是多少，一开始的时候肯定全是1，就是从0开始
33          //然后根据报文的key和mask计算出hash存储起来，继续下一个1的位
34          //直到计算出所有报文hash值，下面会去匹配表项的
35          //hash值的计算可以通过cpu加速，需要cpu支持，并且编译时配置"-msse4.2"
36          ULLONG_FOR_EACH_1(i, keys_map) {
37              hashes[i] = netdev_flow_key_hash_in_mask(&keys[i],
38                          &subtable->mask);
39          }
40          //从子表中进行hash值的匹配，将匹配到node的报文的bit置1到found_map
41          found_map = cmap_find_batch(&subtable->rules, keys_map, hashes, nodes);
42          //在找到匹配node的报文的冲突hash链中继续详细匹配报文
43          ULLONG_FOR_EACH_1(i, found_map) {
44              struct dpcls_rule *rule;
45              //冲突链中继续检测key值是否匹配
46              CMAP_NODE_FOR_EACH (rule, cmap_node, nodes[i]) {
47                  if (OVS_LIKELY(dpcls_rule_matches_key(rule, &keys[i]))) {
48                      //找到匹配的规则，则记录一下，后面会用到
49                      rules[i] = rule;
50                      subtable->hit_cnt++;
51                      lookups_match += subtable_pos;
```

```
                    goto next;
                }
            }
            //不匹配则将该位设置为0
            ULLONG_SET0(found_map, i);  /* Did not match. */
next:
            ;                     /* Keep Sparse happy. */
        }
        //清除已经匹配流表的位
        keys_map &= ~found_map;             /* Clear the found rules. */
        if (!keys_map) {
            if (num_lookups_p) {
                *num_lookups_p = lookups_match;
            }
            return true;              /* All found. */
        }
        subtable_pos++;
    }
    if (num_lookups_p) {
        *num_lookups_p = lookups_match;
    }
    return false;                     /* Some misses. */
}
```

EMC流表安装

dpcls会查找到rules，然后rules转换成flow，最后调用 `emc_insert` 将流表插入到EMC中。

emc_insert

- 根据key->hash找到hash桶，并且进行轮询
- 查看是否有匹配的key值，有的话调用 `emc_change_entry` 修改流表。
- 如果没有匹配的就会根据算法记录一个entry，用来替代
- 循环完毕之后，调用 `emc_change_entry` 替代之前不用的流表

`emc_change_entry`

操作很简单，就是赋值netdev_flow_key和dp_netdev_flow

ofproto classifier查找

`handle_packet_upcall`

- `miniflow_expand` 讲key->mf解析到match.flow
- `dpif_flow_hash` 根据key值计算出hash
- `dp_netdev_upcall` 是进一步调用去ofproto classifier查表的接口，如果失败则删除报文
- `dp_netdev_execute_actions` 可能是直接执行action，后期需要看看为什么不能放入批处理，现在还不明白
- `dp_netdev_pmd_lookup_flow` 需要重新查找dpcls，没有查找到则调用 `dp_netdev_flow_add` 添加流表
- `emc_insert` 讲dpcls的流表插入EMC中

`dp_netdev_upcall-->upcall_cb`

- `upcall_receive` 主要是将一堆信息解析到upcall中
- `process_upcall` 根据upcall的类型MISS_UPCALL确定调用函数 `upcall_xlate`

upcall_xlate

- `xlate_in_init` 主要是将upcall的数据转给xlate_in
- `xlate_actions` 主要是进行流表查找

xlate_actions

- 调用 `xbridge_lookup` 查找对应的xbridge信息
- 根据当前掌握的一堆信息生成一个结构 `xlate_ctx`
- `xlate_wc_init` 主要是初始化通配符的一些已知的项
- `rule_dpif_lookup_from_table` 会查找指定table的流表，默认是table 0，用一个循环去遍历每一个table，然后知道找到匹配的rule
- `do_xlate_actions` 主要是执行所有的action，轮询所有的action，并且根据具体的情况进行相应的操作。
- `tun_metadata_to_geneve_udpif_mask` 给geneve封装metadata

rule_dpif_lookup_from_table

- 如果报文分片，默认是设置源目的端口都设置为0，其他情况下丢弃报文。
- 遍历所有的table，每次都会调用 `rule_dpif_lookup_in_table` 去查找rule，如果找到最终找到之后返回，找不到的话就会去下一个table找。
- 如果遍历完成都找不到，则返回miss_rule

do_xlate_actions

- 根据查找到的rule，遍历所有的action，支持的有OFPACT_OUTPUT、OFPACT_GROUP、OFPACT_CONTROLLER、OFPACT_ENQUEUE、OFPACT_SET_VLAN_VID、OFPACT_SET_VLAN_PCP、OFPACT_STRIP_VLAN、OFPACT_PUSH_VLAN、OFPACT_SET_ETH_SRC、OFPACT_SET_ETH_DST、OFPACT_SET_IPV4_SRC、OFPACT_SET_IPV4_DST、OFPACT_SET_IP_DSCP、OFPACT_SET_IP_ECN、OFPACT_SET_IP_TTL、OFPACT_SET_L4_SRC_PORT、OFPACT_SET_L4_DST_PORT、OFPACT_RESUBMIT、OFPACT_SET_TUNNEL、OFPACT_SET_QUEUE、

OFPACT_POP_QUEUE、OFPACT_REG_MOVE、OFPACT_SET_FIELD、OFPACT_STACK_PUSH、OFPACT_STACK_POP、OFPACT_PUSH_MPLS、OFPACT_POP_MPLS、OFPACT_SET_MPLS_LABEL、OFPACT_SET_MPLS_TC、OFPACT_SET_MPLS_TTL、OFPACT_DEC_MPLS_TTL、OFPACT_DEC_TTL、OFPACT_NOTE、OFPACT_MULTIPATH、OFPACT_BUNDLE、OFPACT_OUTPUT_REG、OFPACT_OUTPUT_TRUNC、OFPACT_LEARN、OFPACT_CONJUNCTION、OFPACT_EXIT、OFPACT_UNROLL_XLATE、OFPACT_FIN_TIMEOUT、OFPACT_CLEAR_ACTIONS、OFPACT_WRITE_ACTIONS、OFPACT_WRITE_METADATA、OFPACT_METER、OFPACT_GOTO_TABLE、OFPACT_SAMPLE、OFPACT_CLONE、OFPACT_CT、OFPACT_CT_CLEAR、OFPACT_NAT、OFPACT_DEBUG_RECIRC，因为action太多，我们先介绍几个常用的

- `OFPACT_OUTPUT`， `xlate_output_action` 会根据端口情况进行一些操作，这块不细看了
- `OFPACT_CONTROLLER`， `execute_controller_action` 生成一个packet_in报文，然后发送
- `OFPACT_SET_ETH_SRC`、`OFPACT_SET_ETH_DST`、`OFPACT_SET_IPV4_SRC`、`OFPACT_SET_IPV4_DST`、`OFPACT_SET_IP_DSCP`、`OFPACT_SET_IP_ECN`、`OFPACT_SET_IP_TTL`、`OFPACT_SET_L4_SRC_PORT`、`OFPACT_SET_L4_DST_PORT`，修改源目的mac、IP以及DSCP、ECN、TTL和L4的源目的端口
- `OFPACT_RESUBMIT`， `xlate_ofpact_resubmit` 会继续查找指定的table的流表
- `OFPACT_SET_TUNNEL`，设置tunnel id
- `OFPACT_CT`， `compose_conntrack_action` 执行完ct的设置之后回调 `do_xlate_actions` 执行其他的action

`rule_dpif_lookup_from_table-->rule_dpif_lookup_in_table-->classifier_lookup-->classifier_lookup__`

- 遍历所有子表，然后调用 `find_match_wc` 根据流表和掩码计算hash，然后进行对子表的各个rule进行匹配比较、
- 如果是严格匹配的话就直接返回，不是严格匹配的话还有一系列操作，暂时先不写了。

dpcls 流表安装

`handle_packet_upcall-->dp_netdev_flow_add-->dpcls_insert` 主要是根据掩码找到相应的子表，然后插入当前的流表

Kindle 笔记导出到印象笔记，分析与完全解决方法 (18.7.17 印象笔记新版分享不是全文解决办法)

spring mvc+ajax 实现json格式数据传递