# 算法采用

将匹配域中的每一个字段又被称为一个维度，多字段的软件查表算法大体上可以分为两类：单维组合分类算法和多维联合分类算法。单维组合查找算法的主要思想是：单独地对数据包每个字段进行匹配，并对每个字段的匹配结果进行合并从而找到最终匹配的规则，其代表包括递归流分类（RFC）、位向量（BV）等。多维联合分类查找算法的大致思想是不单独地考虑每个字段内部特点，而是简单地把包头的所有字段看作一个维度，进行联合查找，其代表包括决策树（Decision tree）、元组空间搜索（TSS）等。而OVS流表采用的算法就是TSS，TSS的一个最大优点是，当所有规则的各字段掩码长度的组合相对较少时，TSS算法很高效，比较适合ovs流表实际的掩码情况。

## 什么是TSS

举一个比较好理解的例子：

> 1. Rule #1: ip_src=192.168.0.0/16 ip_dst=0/0 protocol=0/0 port_src=0/0 port_dst=0/0
> 2. Rule #2: ip_src=0/0 ip_dst=23.23.233.0/24 protocol=6/8(TCP) port_src=0/0 port_dst=23/16
> 3. Rule #3: ip_src=0/0 ip_dst=11.11.233.0/24 protocol=17/8(UDP) port_dst=0/0 port_dst=4789/16
> 4. Rule #4: ip_src=10.10.0.0/16 ip_dst=0/0 protocol=0/0 port_src=0/0 port_dst=0/0

可以看到一个rule中有多个字段，每个字段的形式为 ： 字段值/掩码前缀;
使用相同的匹配字段+每个字段相同的掩码长度，可得两个tuple如下：

> 1. Tuple #1: ip_src_mask=16 ip_dst_mask=0 protocol_mask=0 port_src_mask=0 port_dst_mask=0
> 2. Tuple #2: ip_src_mask=0 ip_dst_mask=24 protocol_mask=8 port_src_mask=0 port_dst_mask=16

tuple将有相同掩码的rule进行合并，每个tuple下面的字段的掩码都相同。用tuple的掩码与上rule中的字段值，得到:

> ip_src=_ ip_dst=23.23.233 protocol=6 port_src=_ port_dst=23

然后将这些位拼接起来得到哈希表的key用于hash查找。

## 匹配过程

1. 所有的rule都被分成了多个tuple，并存储在相应tuple下的哈希表中
2. 当要对一个包进行匹配时，将遍历这多个tuple下的哈希表，一个一个查过去，查出所有匹配成功的结果，然后按一定策略在匹配结果中选出最优的一个。

在ovs流表中采用的是EMC和megaflow table结合的方式，sw_flow_mask结构体相当于是TTS中的tuple，sw_flow相当于是rule。其中EMC流表中存放的是上次访问的sw_flow_mask索引。

# 实际应用

## miss报文处理

```
/*
 * 允许用户临时改变flow中的内容
 */
static struct rule_dpif *
rule_dpif_lookup_in_table(struct ofproto_dpif *ofproto, ovs_version_t version,
                          uint8_t table_id, struct flow *flow,
                          struct flow_wildcards *wc)
{
    struct classifier *cls = &ofproto->up.tables[table_id].cls;
    return rule_dpif_cast(rule_from_cls_rule(classifier_lookup(cls, version,
                                                               flow, wc)));
}
```

## underlay路由表查找

```
/* ovs路由查找，得到网关的IP地址和出接口的源IP地址 */
bool
ovs_router_lookup(const struct in6_addr *ip6_dst, char output_bridge[],
                  struct in6_addr *src, struct in6_addr *gw)
{
    const struct cls_rule *cr;
    struct flow flow = {.ipv6_dst = *ip6_dst};

    cr = classifier_lookup(&cls, OVS_VERSION_MAX, &flow, NULL);
    if (cr) {
        struct ovs_router_entry *p = ovs_router_entry_cast(cr);

        ovs_strlcpy(output_bridge, p->output_bridge, IFNAMSIZ);
        *gw = p->gw;
        if (src) {
            *src = p->src_addr;
        }
        return true;
    }
    return ovs_router_lookup_fallback(ip6_dst, output_bridge, src, gw);
}
```

## 隧道终结表查找

```
/* 'flow' is non-const to allow for temporary modifications during the
lookup.
 * Any changes are restored before returning.
 * flow参数允许临时修改一些值，但是在返回之前需要被恢复原来的值。
 */
odp_port_t
tnl_port_map_lookup(struct flow *flow, struct flow_wildcards *wc)
{
    //进行分类器规则查找
    const struct cls_rule *cr = classifier_lookup(&cls, OVS_VERSION_MAX,
flow,
                                                  wc);

    return (cr) ? tnl_port_cast(cr)->portno : ODPP_NONE;
}
```

# 优化策略

对于流表匹配OVS主要做了如下三种优化策略

1. subtable之间还做了优先级排序，需要优先级向量来标识。这样的话从高优先级的subtable先开始，一旦匹配就可以跳过不少低优先级的subtable。
2. 分阶段匹配，对于一个subtable还可以再继续拆分多个hashtable，因为如果某个subtabl同时要匹配的项比较多，包含了metadata，L2, L3，L4的匹配项，那么就按这四个子类来分，第一阶段先匹配metadata；然后再匹配metadata,L2；继续metadata,L2，L3；最后才是metadata,L2，L3，L4。对于能匹配的rule其实并不能增加效率，但是对于不匹配的情况却是可以增加很大效率。
3. 前缀追踪，前缀跟踪允许分类器跳过比必要的前缀更长的rule。例如，当一个流表中包含一个完整的IP地址匹配和一个地址前缀匹配，完整的地址匹配通常会导致此表的该字段非通配符全0（取决于rule优先级）。在这种情况下，每个有不同的地址的数据包只能被交给用户空间处理并生成自己的数据流。在前缀跟踪启用后用户空间会为问题Packet生成更短的前缀地址匹配，而把无关的地址位置成通配，可以使用一个rule来处理所有的问题包。在这种情况下，可以避免许多的用户上调，这样整体性能可以更好。当然这仅仅是性能优化，因为不管是否有前缀跟踪数据包将得到相同的处理。另外前缀跟踪是可以和分阶段匹配配套使用，Trie树会追踪整个Classifier中所有Rule的地址前缀的数量。更神奇的是通过维护一个在Trie树遍历中遇到的最长前缀的列表或者维护通过不同Metadata分开的规则子集独立的Trie树可以实现表跳跃。前缀追踪是通过OVSDB“Flow_Table”表”fieldspec” 列来配置的，”fieldspec” 列是用string map这里前缀的key值告诉哪些字段可以被用来做前缀追踪。

# 流表数据结构

## 数据结构示意图

## 核心流表数据结构分析

EMC(即为Microflow Cache)的数据结构

```
struct mask_cache_entry {
    u32 skb_hash;
    u32 mask_index;
};

```

他其实本质就是个地址，存放的是上一次匹配命中的Megaflow中某个元组表的索引值。
Mask_array是个指针数组，数组中每个元素存放的是掩码，每个掩码就代表一个元组表。每条流的首报文都会遍历这个数组，相当于遍历所有的元组表。每个元组表采用hash的方式，用链式解决Hash冲突。

```
struct mask_array {
    struct rcu_head rcu;
    int count, max;
    struct sw_flow_mask __rcu *masks[];
};

```

dpcls(即为megaflow cache)的数据结构

```
struct sw_flow {
    struct rcu_head rcu;
    struct {
        struct hlist_node node[2];
        u32 hash;
    } flow_table, ufid_table;
    int stats_last_writer;      /* CPU id of the last writer on
                    * 'stats[0]'.
                    */
    struct sw_flow_key key;
    struct sw_flow_id id;
    struct cpumask cpu_used_mask;
    struct sw_flow_mask *mask;
    struct sw_flow_actions __rcu *sf_acts;
    struct sw_flow_stats __rcu *stats[]; /* One for each CPU.  First one
                      * is allocated at flow creation time,
                      * the rest are allocated on demand
                      * while holding the 'stats[0].lock'.
                      */
};

```

sw_flow通过hash_list串接起来。

```
struct flow_table {
    struct table_instance __rcu *ti;
    struct table_instance __rcu *ufid_ti;
    struct mask_cache_entry __percpu *mask_cache;
    struct mask_array __rcu *mask_array;
    unsigned long last_rehash;
    unsigned int count;
    unsigned int ufid_count;
};

```

openflow流表的主要核心结构如下

```
struct cls_subtable {
    struct cmap_node cmap_node;    /* 连接到分类器的子表hash表中，根据掩码进行hash
*/

```

```
    /* These fields are only used by writers. */
    int max_priority;              /* Max priority of any rule in subtable.
*/
    unsigned int max_count;        /* Count of max_priority rules. */

    /* Accessed by iterators. */
    struct rculist rules_list;              /* 无序规则链表. */

    /* Identical, but lower priority rules are not inserted to any of the
     * following data structures. */

    /* These fields are accessed by readers who care about wildcarding.
       这个值为3，不包括最后一个段 */
    const uint8_t n_indices;                    /* 存在掩码有效段个数 */
    const struct flowmap index_maps[CLS_MAX_INDICES + 1]; /* 阶段索引映射 */

    //前缀树的前缀长度，所有前缀有相同的前缀长度
    unsigned int trie_plen[CLS_MAX_TRIES];  /* 子表前缀树的长度 */
    const int ports_mask_len;

    //阶段查询hash map，保存的是前三个段的hash值，可以通过hash进行快速过滤
    struct ccmap indices[CLS_MAX_INDICES];  /* Staged lookup indices. */
    rcu_trie_ptr ports_trie;                /* NULL if none. */

    /* 根据最后一个段进行hash，确定hash桶，然后进行匹配 */
    struct cmap rules;                      /* Contains 'cls_match'es. */
    const struct minimask mask;             /* Wildcards for fields. */
    /* 'mask' must be the last field. */
};

struct classifier {
    int n_rules;                    /* 总共的规则数目 */
    uint8_t n_flow_segments;
    /*段匹配器的段的个数，  一般是四个段：  metadata，l2,l3,l4，但是这个值只能是3，最后
一个段单独处理 */

    //存储每个段的起始地址的64字节偏移
    uint8_t flow_segments[CLS_MAX_INDICES];

    /* for staged lookup. 段匹配器的结束边界,总共有四个段，三个边界
    * |---段1----|-----段2----|----段三----|----段4-----|
    *           边界1         边界2           边界3
    */

    struct cmap subtables_map;      /* 子表hash表，分类器下，按照flow的掩码不同
                                     * 划分成不同的子表，每个子表的规则的掩码是一样
的，一个流表有多个子表  */
    struct pvector subtables;       /* 子表数组，按照表格的优先级进行排列，这样的
话，可以跳过低优先级的表格  */
    struct cmap partitions;         /* Contains "struct cls_partition"s. */
    struct cls_trie tries[CLS_MAX_TRIES]; /*  前缀树，用于过滤一些规则 */
    unsigned int n_tries;           /* 前缀树的个数 */
    bool publish;                   /* 表示是否可以查询 */
};

const uint8_t flow_segment_u64s[4] = {
    FLOW_SEGMENT_1_ENDS_AT / sizeof(uint64_t),/* 元数据匹配结束地址 */
    FLOW_SEGMENT_2_ENDS_AT / sizeof(uint64_t),/* 链路层匹配结束    地址 */
```

```
    FLOW_SEGMENT_3_ENDS_AT / sizeof(uint64_t),/* 网络层匹配结束地址 */
    FLOW_U64S                                 /* 传输层匹配结束地址 */
};
```

cls_subtable主要是openflow的单个结构体，classifier是用来管理openflow流表的结构体。
在分类器中，规则使用struct cls_rule进行组织，一个规则有一个掩码域和一个值域。

```
/* A rule to be inserted to the classifier. */
/* 一个规则被插入分类器 */
struct cls_rule {
    struct rculist node;          /* In struct cls_subtable 'rules_list'. 双链表，连接到对应的子表中 */
    const int priority;           /* Larger numbers are higher priorities.值越大其优先级越高 */
    OVSRCU_TYPE(struct cls_match *) cls_match;  /* NULL if not in a
                                                 * classifier. */
    const struct minimatch match; /* Matching rule. 匹配器，由两个miniflow构成，一个位掩码一个值 */
};

/* Internal representation of a rule in a "struct cls_subtable".
 * 子表中的规则表示。next成员构成单链表，该链表表示相同的匹配域的不同规则。按照优先级
 * 高低顺序进行排列。
 */
struct cls_match {
    /* Accessed by everybody. */
    OVSRCU_TYPE(struct cls_match *) next; /* Equal, lower-priority matches. 相同的匹配条件，根据优先级进行排序 */
    OVSRCU_TYPE(struct cls_conjunction_set *) conj_set;

    /* Accessed by readers interested in wildcarding. */
    /* 优先级，值越大优先级越高 */
    const int priority;         /* Larger numbers are higher priorities. */

    /* Accessed by all readers. */
    struct cmap_node cmap_node; /* Within struct cls_subtable 'rules'. h挂到子表的rules链表中 */

    /* Rule versioning. */
    /* 规则版本号 */
    struct versions versions;

    const struct cls_rule *cls_rule;/* 指向其所属的规则 */
    const struct miniflow flow; /* Matching rule. Mask is in the subtable. 匹配域，其掩码在subtable中 */
    /* 'flow' must be the last field. */
};
```

## Trie树匹配优化

一个分类器classifier可以有多个Trie，典型的是两个：IPV4源目的IP。

```
typedef OVSRCU_TYPE(struct trie_node *) rcu_trie_ptr;

/* A longest-prefix match tree. */
/* 一个最长匹配的前缀树，前缀树节点      */
```

```
struct trie_node {
    uint32_t prefix;            /* Prefix bits for this node, MSB first. 前缀，msb优先*/
    uint8_t  n_bits;            /* Never zero, except for the root node. 前缀长度，用于限制上面的prefix */
    unsigned int n_rules;       /* Number of rules that have this prefix. 拥有这个前缀的规则数 */
    rcu_trie_ptr edges[2];      /* 叶子的话，两边的都为空，
                                   树的两个节点，左右节点，因为一个bit只有两个值，
                                   要么0，要么1，所以是二叉树 */
};
/* Prefix trie for a 'field' */
/* 为field域构建前缀树 */
struct cls_trie {
    const struct mf_field *field; /* Trie field, or NULL. 构成该树的报文域 */
    rcu_trie_ptr root;            /* NULL if none. 树根 */
};
```

## trie树具体流程

1. trie_init

```
/* 进行分类器的前缀树的初始化：1计算前缀长度，2将已经存在的规则加入该前缀树 */
static void
trie_init(struct classifier *cls, int trie_idx, const struct mf_field *field)
{
    struct cls_trie *trie = &cls->tries[trie_idx];/* 从前缀树数组中获取指定的树描述控制块 */
    struct cls_subtable *subtable;/* 子表 */

    /* 先将以前的树进行删除 */
    if (trie_idx < cls->n_tries) {
        trie_destroy(&trie->root);
    } else {
        ovsrcu_set_hidden(&trie->root, NULL);
    }
    //设置该前缀树对应的域
    trie->field = field;

    /* 添加已经存在的规则到新树中 */
    CMAP_FOR_EACH (subtable, cmap_node, &cls->subtables_map) {/* 遍历每一个子表 */
        unsigned int plen;/* 前缀长度局部变量 */
        //计算前缀树的前缀长度
        plen = field ? minimask_get_prefix_len(&subtable->mask, field) : 0;/* 获取匹配域的掩码的长度 */
        if (plen) {/* 存在前缀长度，那么将该子表下的规则插入该前缀树 */
            struct cls_match *head;

            CMAP_FOR_EACH (head, cmap_node, &subtable->rules) {/* 遍历每一个规则 */
                trie_insert(trie, head->cls_rule, plen);/* 将规则插入前缀树 */
            }
        }
        /* Initialize subtable's prefix length on this field.  This will
         * allow readers to use the trie. */
```

```
        atomic_thread_fence(memory_order_release);
        subtable->trie_plen[trie_idx] = plen; /* 设置子表的前缀树的前缀长度，子表的前缀长度是确定的 */
    }
}
```

2. trie_insert

```
/* 插入的是前缀掩码 */
/* edge为当前节点 */
static void
trie_insert_prefix(rcu_trie_ptr *edge, const ovs_be32 *prefix, int mlen)/* 前缀与前缀长度 */
{
    struct trie_node *node;
    int ofs = 0;

    /* Walk the tree. */
    /* 遍历整棵树 */
    for (; (node = ovsrcu_get_protected(struct trie_node *, edge));/* 从根节点开始 */
         edge = trie_next_edge(node, prefix, ofs)) {/* 根据前缀bit的值为1还是0获取下一个节点 */
        unsigned int eqbits = trie_prefix_equal_bits(node, prefix, ofs, mlen);/*  */
        ofs += eqbits;/* 得到需要插入的前缀与本节点相同的bit个数 */
        if (eqbits < node->n_bits) {/* bit相等的个数比该节点的前缀bit个数要小，说明两者不相等 */
            /* Mismatch, new node needs to be inserted above. 所以需要新建一个节点*/
            int old_branch = get_bit_at(node->prefix, eqbits);/* 从相等的位置开始一个新的分支 */
            struct trie_node *new_parent;
            /* 创建一个新的分支 */
            new_parent = trie_branch_create(prefix, ofs - eqbits, eqbits,
                                            ofs == mlen ? 1 : 0);
            /* Copy the node to modify it. */
            /* 修改一个节点 */
            node = trie_node_rcu_realloc(node);
            /* Adjust the new node for its new position in the tree. */
            node->prefix <<= eqbits; /* 去掉相同的几个bit，因为这几个bit分给新的父节点 */
            node->n_bits -= eqbits;  /* 前缀个数减小，因为新的节点已经占用了一部分 */
            ovsrcu_set_hidden(&new_parent->edges[old_branch], node);/* 原来的节点作为这个新的节点的子节点 */

            /* Check if need a new branch for the new rule. */
            if (ofs < mlen) {
                ovsrcu_set_hidden(&new_parent->edges[!old_branch],
                                  trie_branch_create(prefix, ofs, mlen - ofs,
                                                     1));
            }
            ovsrcu_set(edge, new_parent); /* Publish changes. */
            return;
        }
```

```
        /* Full match so far. */

        if (ofs == mlen) {
            /* Full match at the current node, rule needs to be added here.
*/
            node->n_rules++;
            return;
        }
    }
    /* Must insert a new tree branch for the new rule. */
    ovsrcu_set(edge, trie_branch_create(prefix, ofs, mlen - ofs, 1));
}
/*
 * This is called only when mask prefix is known to be CIDR and non-zero.
 * Relies on the fact that the flow and mask have the same map, and since
 * the mask is CIDR, the storage for the flow field exists even if it
 * happened to be zeros.
 * 获取指定的前缀。
 */
static const ovs_be32 *
minimatch_get_prefix(const struct minimatch *match, const struct mf_field
*mf)
{
    size_t u64_ofs = mf->flow_be32ofs / 2;/* 4字节单位偏移转换成8个字节单位偏移
*/

    return (OVS_FORCE const ovs_be32 *)miniflow_get__(match->flow, u64_ofs)
        + (mf->flow_be32ofs & 1);
}
/* Insert rule in to the prefix tree.
 * 'mlen' must be the (non-zero) CIDR prefix length of the 'trie->field'
mask
 * in 'rule'.
 * 插入一个规则到前缀树，前缀掩码必须不为0
 */
static void
trie_insert(struct cls_trie *trie/* 需要插入的树 */, const struct cls_rule
*rule/* 需要插入的规则 */, int mlen/* 掩码长度 */)
{
    trie_insert_prefix(&trie->root,/* 根节点 */
                       minimatch_get_prefix(&rule->match, trie->field)/* 获
取匹配域的掩码起始地址 */, mlen/* 掩码长度 */);
}
```

3. trie_remove

```
/* 'mlen' must be the (non-zero) CIDR prefix length of the 'trie->field'
mask
 * in 'rule'. 移除规则cls_rule对应的前缀*/
static void
trie_remove(struct cls_trie *trie,/* 规则的前缀树 */
                const struct cls_rule *rule, /* 要移除的规则描述 */
                int mlen)/* 掩码长度 */
{
    trie_remove_prefix(&trie->root,
                       minimatch_get_prefix(&rule->match, trie->field),
mlen);
```

```
}

/* 'mlen' must be the (non-zero) CIDR prefix length of the 'trie->field' mask
 * in 'rule'. */
static void
trie_remove_prefix(rcu_trie_ptr *root, const ovs_be32 *prefix, int mlen)
{
    struct trie_node *node;
    rcu_trie_ptr *edges[sizeof(union trie_prefix) * CHAR_BIT];
    int depth = 0, ofs = 0;

    /* Walk the tree. */
    for (edges[0] = root;
         (node = ovsrcu_get_protected(struct trie_node *, edges[depth]));
         edges[++depth] = trie_next_edge(node, prefix, ofs)) {/* 根据前缀获取下一个节点 */
        unsigned int eqbits = trie_prefix_equal_bits(node, prefix, ofs, mlen);/* 判断两个节点前缀相等的位置 */

        if (eqbits < node->n_bits) {/* 不相等，说明不能命中 */
            /* Mismatch, nothing to be removed.  This should never happen, as
             * only rules in the classifier are ever removed. */
            break; /* Log a warning. */
        }
        /* Full match so far. 全匹配，到目前位置 */
        ofs += eqbits;

        if (ofs == mlen) {/* 全匹配了 */
            /* Full prefix match at the current node, remove rule here. */
            if (!node->n_rules) {/* 没有规则，直接退出，一般是不可能的 */
                break; /* Log a warning. */
            }
            node->n_rules--;/* 规则数减掉1 */

            /* Check if can prune the tree. 检查是否能阶段这个树，一般来说，最后一个规则被删除的话，是可以截断这棵树的 */
            while (!node->n_rules) {
                struct trie_node *next,/* 获取当前节点的左右分支 */
                    *edge0 = ovsrcu_get_protected(struct trie_node *,
                                                  &node->edges[0]),
                    *edge1 = ovsrcu_get_protected(struct trie_node *,
                                                  &node->edges[1]);

                if (edge0 && edge1) {/* 左右分支都存在不能截断 */
                    break; /* A branching point, cannot prune. */
                }

                /* Else have at most one child node, remove this node. 只有一个分支，截断他，这个分支就是最后一个规则的分支 */
                next = edge0 ? edge0 : edge1;

                if (next) {
                    if (node->n_bits + next->n_bits > TRIE_PREFIX_BITS) {
                        break;    /* Cannot combine. */
                    }
                    next = trie_node_rcu_realloc(next); /* Modify. */
```

```
                    /* Combine node with next. */
                    next->prefix = node->prefix | next->prefix >> node->n_bits;
                    next->n_bits += node->n_bits;
                }
                /* Update the parent's edge. */
                ovsrcu_set(edges[depth], next); /* Publish changes. 更新变化 */
                trie_node_destroy(node);/* 销毁当前节点 */

                if (next || !depth) {
                    /* Branch not pruned or at root, nothing more to do. */
                    break;
                }
                node = ovsrcu_get_protected(struct trie_node *,
                                            edges[--depth]);
            }
            return;
        }
    }
    /* Cannot go deeper. This should never happen, since only rules
     * that actually exist in the classifier are ever removed. */
}
```

4. trie_lookup

```
/* 前缀树查找函数
 * trie:树根
 * value:需要匹配的内容，一般是ip地址(ipv4或者ipv6)
 * plens:命中结果存储的位置，它跟value是等长的，用于记录存在规则的bit，即多长的前缀
 * 存在规则的命中。
 * n_bits:本域的匹配长度(bit数)，要么是32要么是128。
 * 返回值为匹配过的bit数，如果最后一个node miss-match，返回值为match_len + 1
 */
static unsigned int
trie_lookup_value(const rcu_trie_ptr *trie, const ovs_be32 value[],
                  ovs_be32 plens[]/* 返回存在命中规则的bit，用于记录命中 */, unsigned int n_bits/* 本区域最长的bit数 */)
{
    const struct trie_node *prev = NULL;
    const struct trie_node *node = ovsrcu_get(struct trie_node *, trie);
    unsigned int match_len = 0; /* Number of matching bits. */

    /* 从树根开始遍历节点 */
    for (; node; prev = node, node = trie_next_node(node, value, match_len)) {
        unsigned int eqbits;
        /* Check if this edge can be followed. 判断下一个节点的前缀与需要查找的前缀相等的个数 */
        eqbits = prefix_equal_bits(node->prefix, node->n_bits, value,
                                   match_len);
        match_len += eqbits;/* 得到相等的个数 */
        if (eqbits < node->n_bits) { /* Mismatch, nothing more to be found. */
            /* Bit at offset 'match_len' differed. 如果相等的bit个数不一致的话，说明这两个节点代表的掩码不一样 */
```

```
            return match_len + 1; /* Includes the first mismatching bit. 返回的长度还包含了第一个不匹配的bit */
        }
        /* Full match, check if rules exist at this prefix length.
         * 如果两者的前缀掩码完全一样，判断该节点是否存在规则 */
        /* 如果存在的话，说明已经命中了规则，设置当前位置命中了规则
         */
        if (node->n_rules > 0) {
            be_set_bit_at(plens, match_len - 1);
        }
        if (match_len >= n_bits) {/* 全匹配了 */
            return n_bits; /* Full prefix. */
        }
    }
    /* node == NULL.  Full match so far, but we tried to follow an
     * non-existing branch.  Need to exclude the other branch if it exists
     * (it does not if we were called on an empty trie or 'prev' is a leaf
     * node).
     * 能走到这里说明整个前缀树都匹配完毕了，即node == NULL跳出了循环。
     * 没有全前缀的命中。
     * prev为NULL，说明整棵树为空。
     * prev为leaf，说明整棵树匹配完毕。
     * 其它情况应该不会走到这里，所以不会存在返回match_len + 1的情况。
     * match_len + 1表示包含了第一个不匹配的bit。
     */
    return !prev || trie_is_leaf(prev) ? match_len : match_len + 1;
}

/* 使用分类器的前缀树进行查找flow */
/* trie:分类器前缀树
** flow:需要匹配的报文的flow
** plens:返回的匹配的前缀
 */
static unsigned int
trie_lookup(const struct cls_trie *trie, const struct flow *flow,
            union trie_prefix *plens)
{
    const struct mf_field *mf = trie->field;/* 树的匹配元数据 */

    /* 检查当前流的匹配先决条件，一些匹配域可能用于多个目的。比如我们需要查找的
     * ipv4的话，那么在Ethernet头部的ethertype字段必须要等于ipv4
     */
    if (mf_are_prereqs_ok(mf, flow, NULL)) {/* 进行查找 */
        return trie_lookup_value(&trie->root,/* 树根节点 */
                                 &((ovs_be32 *)flow)[mf->flow_be32ofs],/* 报文在该域的值 */
                                 &plens->be32, mf->n_bits);//该域的长度
    }
    //不满足条件，说明本报文不需要参与前缀树过滤，所以对任何subtable都是符合条件的。
    //不需要跳过，继续查找subtable。
    memset(plens, 0xff, sizeof *plens); /* All prefixes, no skipping. */
    return 0; /* Value not used in this case. 这种情况下，该报文不需要参与Trie过滤 */
}
```

5. 过滤器check_tries

```c
/* 基于前缀树查找结果，前缀树一般用于匹配源目的ip字段。用于快速过滤。
 */
static inline bool
check_tries(struct trie_ctx trie_ctx[CLS_MAX_TRIES], unsigned int n_tries,
            const unsigned int field_plen[CLS_MAX_TRIES],
            const struct flowmap range_map, const struct flow *flow,//报文内容
            struct flow_wildcards *wc)//报文域bit位
{
    int j;

    /* Check if we could avoid fully unwildcarding the next level of
     * fields using the prefix tries.  The trie checks are done only as
     * needed to avoid folding in additional bits to the wildcards mask.
     * */
    for (j = 0; j < n_tries; j++) {/* 遍历每一棵树 */
        /* Is the trie field relevant for this subtable, and
           is the trie field within the current range of fields? */
        if (field_plen[j] &&/* 前缀长度存在并且在当前的stage中，则需要进行匹配 */
            flowmap_is_set(&range_map, trie_ctx[j].be32ofs / 2)) {
            struct trie_ctx *ctx = &trie_ctx[j];/* 树匹配上下文 */

            /* 对需要查找的树进行查找，整个报文每个trie只需要查找一次，后续子表直接拿前面的结果来用即可 */
            if (!ctx->lookup_done) {/* 树还没查找过，进行查找 */
                memset(&ctx->match_plens, 0, sizeof ctx->match_plens);/* 清空匹配了的长度为0 */
                /* 返回的是参与了匹配的bit数，命中 */
                ctx->maskbits = trie_lookup(ctx->trie, flow, &ctx->match_plens);
                ctx->lookup_done = true;/* 设置本树已经查找完了 */
            }
            /* 查看本subtable的前缀长度是否有规则命中，没有的话，说明该子表不存在
             * 命中的可能，直接跳过。没看明白里面的两个if的作用。
             */
            if (!be_get_bit_at(&ctx->match_plens.be32, field_plen[j] - 1)) {
                /* Trie是整个分类器共享的。它是可能的maskbits可能包含一些与这个报文不相关的部分。
                 * 因此需要进行下面的检查。
                 */

                /* Check that the trie result will not unwildcard more bits
                 * than this subtable would otherwise.
                 * 这棵树最长*/
                if (ctx->maskbits <= field_plen[j]) {//没有全匹配
                    /* Unwildcard the bits and skip the rest. */
                    /* 设置非通配bit，用于跳过其子表其他 */
                    mask_set_prefix_bits(wc, ctx->be32ofs, ctx->maskbits);
                    /* Note: Prerequisite already unwildcarded, as the only
                     * prerequisite of the supported trie lookup fields is
                     * the ethertype, which is always unwildcarded. */
                    return true;
                }
                /* Can skip if the field is already unwildcarded. */
                if (mask_prefix_bits_set(wc, ctx->be32ofs, ctx->maskbits)) {
                    return true;
```

```
                }
            }
        }
    }
    return false;
}
```

## 关联匹配规则

关联匹配即多个规则同时命中后才算命中规则，即叫关联匹配。命中关联匹配中的一个规则叫soft-match，关联匹配所有规则都soft-match后，那么整体规则才算命中，由soft-match转换成hard-match。

```
/* A collection of "struct cls_conjunction"s currently embedded into a
 * cls_match.一个cls_conjunction集合用于嵌入到cls_match中的
 * 一个关联规则，可以关联多个关联id
 */
struct cls_conjunction_set {
    /* Link back to the cls_match.
    反向指向其所属的匹配器 */
    struct cls_match *match;
    int priority;               /* Cached copy of match->priority.缓存其所属的匹配器的优先级 */

    /* Conjunction information.
     * 连接点信息
     * 'min_n_clauses' allows some optimization during classifier lookup. */
    unsigned int n;             /* Number of elements in 'conj'. 关联规则个数 */
    unsigned int min_n_clauses; /* Smallest 'n' among elements of 'conj'.最小的n在关联集合数组中 */
    struct cls_conjunction conj[];/* 关联的规则数组 */
};

struct cls_conjunction {
    uint32_t id;/* 关联动作id */
    uint8_t clause;/* 流的bit位置,即关联的维度，从0开始 */
    uint8_t n_clauses;/* 关联的流的个数，多少个维度 */
};

//关联id结构，一个关联id，clauses个维度
//该结构是一个内部结构，用于关联匹配时进行暂存，等待所有维度都命中后，转换为hard-match。
struct conjunctive_match {
    struct hmap_node hmap_node;/* 用于连接到hash表中，一般用于暂存表matches中 */
    uint32_t id;/* 关联id */
    uint64_t clauses;/* 对应的流表位掩码，64个bit，也就是说ovs支持64个维度 */
};
```

关联规则匹配流程

```
/* 遍历hash桶，在暂存hash表中找到关联id相同的项，没找到返回NULL */
static struct conjunctive_match *
find_conjunctive_match__(struct hmap *matches, uint64_t id, uint32_t hash)
{
    struct conjunctive_match *m;
```

```
6
7       HMAP_FOR_EACH_IN_BUCKET (m, hmap_node, hash, matches) {/* 在一个hash表中进行匹配 */
8           if (m->id == id) {/* 遍历hash桶的每一个元素，比较id是否相等 */
9               return m;
10          }
11      }
12      return NULL;
13  }
14
15  /* 查找关联匹配 */
16  static bool
17  find_conjunctive_match(const struct cls_conjunction_set *set, /* 本规则对应的关联id集合 */
18                         unsigned int max_n_clauses, /* 本table所命中该set对应的规则优先的规则个数 */
19                         struct hmap *matches, /* 关联id暂存hash表*/
20                         /* conjunctive_match局部数组，避免动态分配内存 */
21                         struct conjunctive_match *cm_stubs, size_t n_cm_stubs,
22                         uint32_t *idp)//返回命中的关联id
23  {
24      const struct cls_conjunction *c;
25
26      if (max_n_clauses < set->min_n_clauses) {/* 命中对应优先级的规则个数小于该规则所有的关联id的维度 */
27          return false;/* 直接返回，不可能命中 */
28      }
29
30      /* 遍历该规则对应的每一个关联id */
31      for (c = set->conj; c < &set->conj[set->n]; c++) {/* 遍历每一个关联匹配 */
32          struct conjunctive_match *cm;/* 关联匹配 */
33          uint32_t hash;
34          /* 关联规则的维度大于对应优先级命中的规则，那么不可能命中，直接跳过 */
35          if (c->n_clauses > max_n_clauses) {
36              continue;
37          }
38
39          hash = hash_int(c->id, 0);/* 对id进行hash，然后用来根据关联id查找关联匹配 */
40          /* matches为暂存表，即同一个关联id的第一个处理的c会添加到该hash表中 */
41          cm = find_conjunctive_match__(matches, c->id, hash);
42          if (!cm) {/* 没有找到，说明需要新增 */
43              /* 计算已经暂存的关联id的个数，如果大于我们的局部数组，那么需要动态分配内存用于暂存hash表 */
44              size_t n = hmap_count(matches);
45              cm = n < n_cm_stubs ? &cm_stubs[n] : xmalloc(sizeof *cm);
46              hmap_insert(matches, &cm->hmap_node, hash);/* 将新建的cm插入到hash表中 */
47              cm->id = c->id;/* 记录id */
48              /* 有几个维度就将0xffffffffffffffff向左移动多少位，即将最右边几个bit清零，然后在命中的时候，
49              ** 命中一个维度就将该bit置1.一旦cm->clauses == 0xffffffffffffffff 时表示该关联匹配命中了。
50              */
51              cm->clauses = UINT64_MAX << (c->n_clauses & 63);
52          }
53          /* 设置本关联匹配维度对应的bit位 */
```

```
        cm->clauses |= UINT64_C(1) << c->clause;
        if (cm->clauses == UINT64_MAX) {/* 一旦等于全ff，则说明关联匹配全命中了 */
            *idp = cm->id;/* 返回关联匹配id，说明同一个优先级不能设置两个关联id */
            return true;
        }
    }
    return false;
}

/* 释放关联匹配的内存，即释放掉暂存表matches，分为两部分，一部分是cm_stubs中的内容不用在这里释放
** 需要释放的是动态申请的内存
*/
static void
free_conjunctive_matches(struct hmap *matches,
                         struct conjunctive_match *cm_stubs, size_t n_cm_stubs)
{
    if (hmap_count(matches) > n_cm_stubs) {/* 只有大于我们的局部数组的时候，才需要进行内存的释放 */
        struct conjunctive_match *cm, *next;

        HMAP_FOR_EACH_SAFE (cm, next, hmap_node, matches) {
            /* 不在数组内的内存是动态申请的，进行释放 */
            if (!(cm >= cm_stubs && cm < &cm_stubs[n_cm_stubs])) {
                free(cm);
            }
        }
    }
    hmap_destroy(matches);
}
```

## subtable接口分析

1. insert_subtable

```
/* 该函数新建一个子表，插入到分类器中。当一个规则的掩码与现有的所有子表的掩码都不相同时，会调用新建一个子表。 */
/* 在这个函数后，新的子表可以被读者看到 */
static struct cls_subtable *
insert_subtable(struct classifier *cls, const struct minimask *mask)
{
    uint32_t hash = minimask_hash(mask, 0);/* 根据掩码计算hash值 */
    struct cls_subtable *subtable;
    int i, index = 0;
    struct flowmap stage_map;
    uint8_t prev;
    size_t count = miniflow_n_values(&mask->masks);/* 计算掩码中1的个数 */

    subtable = xzalloc(sizeof *subtable + MINIFLOW_VALUES_SIZE(count));/* 分配子表 */
    cmap_init(&subtable->rules);/* 初始化规则hash表 */
    miniflow_clone(CONST_CAST(struct miniflow *, &subtable->mask.masks),
                   &mask->masks, count);/* 将掩码拷贝到子表中 */

    /* Init indices for segmented lookup, if any. */
```

```
    /* 初始化段查询索引 */
    prev = 0;
    for (i = 0; i < cls->n_flow_segments; i++) {/* 遍历每一个段，所有的子表都跟分类器拥有一样的段匹配器 */
        /* 构建段映射map表，prev为段起始地址，cls->flow_segments[i]为段的结束地址 */
        stage_map = miniflow_get_map_in_range(&mask->masks, prev,
                                              cls->flow_segments[i]);
        /* Add an index if it adds mask bits. */
        /* 如果非空，即存在有效值域 */
        if (!flowmap_is_empty(stage_map)) {/* 如果该子表的掩码在这个段不为空 */
            ccmap_init(&subtable->indices[index]);/* 初始化该段匹配hash表 */
            *CONST_CAST(struct flowmap *, &subtable->index_maps[index])/* 将该段的掩码赋值给段掩码数组 */
                = stage_map;
            index++;/* 统计存在掩码段的个数 */
        }
        prev = cls->flow_segments[i];/* 获取下一个段的边界 */
    }
    /* Map for the final stage. */
    *CONST_CAST(struct flowmap *, &subtable->index_maps[index])
        = miniflow_get_map_in_range(&mask->masks, prev, FLOW_U64S);/* 进行最后一个段的处理 */

    /* Check if the final stage adds any bits. */
    /* 检查最后一个段是否为空 */
    if (index > 0) {
        if (flowmap_is_empty(subtable->index_maps[index])) {/* 最后一个段没有掩码，则统计减掉1 */
            /* Remove the last index, as it has the same fields as the rules
             * map. */
            --index;
            ccmap_destroy(&subtable->indices[index]);
        }
    }
    *CONST_CAST(uint8_t *, &subtable->n_indices) = index;/* 存在有效掩码段的个数 */
    //计算前缀树的前缀长度
    for (i = 0; i < cls->n_tries; i++) {/* 遍历该分类器支持的前缀树，初始化该子表相应的前缀树掩码的长度 */
        subtable->trie_plen[i] = minimask_get_prefix_len(mask,
                                                         cls->tries[i].field);/* 获取前缀长度 */
    }

    /* Ports trie. 端口前缀树 */
    ovsrcu_set_hidden(&subtable->ports_trie, NULL);
    *CONST_CAST(int *, &subtable->ports_mask_len)/* 得到该位置1的个数 */
        = 32 - ctz32(ntohl(MINIFLOW_GET_BE32(&mask->masks, tp_src)));

    /* List of rules. 初始化子表规则链表 */
    rculist_init(&subtable->rules_list);

    /* 将子表插入到分类器的子表hash表中 */
    cmap_insert(&cls->subtables_map, &subtable->cmap_node, hash);

    return subtable;
```

```
}
```

2. find_match

```
/* 查找匹配域 */
static inline const struct cls_match *
find_match(const struct cls_subtable *subtable, ovs_version_t version,
           const struct flow *flow, uint32_t hash)
{
    const struct cls_match *head, *rule;

    CMAP_FOR_EACH_WITH_HASH (head, cmap_node, hash, &subtable->rules) {/* 遍历规则hash表 */
        if (OVS_LIKELY(miniflow_and_mask_matches_flow(&head->flow,/* 匹配域 */
                                                      &subtable->mask,/* 子表掩码 */
                                                      flow))) {/* 进行掩码匹配 */
            /* Return highest priority rule that is visible. */
            /* 返回本设备支持的最高优先级的规则 */
            CLS_MATCH_FOR_EACH (rule, head) {/* 遍历hash桶链表的每一个规则 */
                if (OVS_LIKELY(cls_match_visible_in_version(rule, version))) {
                    return rule;
                }
            }
        }
    }

    return NULL;
}
```

3. find_match_wc

```
/* 进行通配符的查找 */
static const struct cls_match *
find_match_wc(const struct cls_subtable *subtable, ovs_version_t version,
              const struct flow *flow, struct trie_ctx trie_ctx[CLS_MAX_TRIES],
              unsigned int n_tries, struct flow_wildcards *wc)
{
    if (OVS_UNLIKELY(!wc)) {/* 对于不需要生成缓存的匹配器，wc为null，比如underlay路由表 */
        return find_match(subtable, version, flow,
                          flow_hash_in_minimask(flow, &subtable->mask, 0));
    }

    uint32_t basis = 0, hash;
    const struct cls_match *rule = NULL;
    struct flowmap stages_map = FLOWMAP_EMPTY_INITIALIZER;
    unsigned int mask_offset = 0;
    int i;

    /* 在段里面尝试完成早期的校验 */
    for (i = 0; i < subtable->n_indices; i++) {/* 进行segments查找 */
```

```
        if (check_tries(trie_ctx, n_tries, subtable->trie_plen/* 前缀长度，匹配，查看是否可以跳过这个子表 */,
                        subtable->index_maps[i], flow, wc)) {
            /* 'wc' bits for the trie field set, now unwildcard the preceding
             * bits used so far. */
            goto no_match;
        }

        /* Accumulate the map used so far. */
        /* 累积到目前为止的掩码           */
        stages_map = flowmap_or(stages_map, subtable->index_maps[i]);
        /* 计算当前的hash值 */
        hash = flow_hash_in_minimask_range(flow, &subtable->mask,
                                           subtable->index_maps[i],
                                           &mask_offset, &basis);

        /* 段匹配，没有命中，跳出 */
        if (!ccmap_find(&subtable->indices[i], hash)) {
            goto no_match;
        }
    }
    /* Trie check for the final range. */
    /* 为最后的范围做树校验 */
    if (check_tries(trie_ctx, n_tries, subtable->trie_plen,
                    subtable->index_maps[i], flow, wc)) {
        goto no_match;
    }
    /* 对指定range进行hash */
    hash = flow_hash_in_minimask_range(flow, &subtable->mask,
                                       subtable->index_maps[i],
                                       &mask_offset, &basis);
    /* 进行子表匹配，找到最高优先级的 */
    rule = find_match(subtable, version, flow, hash);
    if (!rule && subtable->ports_mask_len) {/* 没有找到，并且有传输层src ports引擎树 */
        /* The final stage had ports, but there was no match.  Instead of
         * unwildcarding all the ports bits, use the ports trie to figure out a
         * smaller set of bits to unwildcard.
         * 最后阶段存在端口，但是这儿没有匹配，代替非通配所有的端口，使用ports tree去计算最小的通配端口集合 */
        unsigned int mbits;
        ovs_be32 value, plens, mask;
        /* 获取四层端口的掩码 */
        mask = MINIFLOW_GET_BE32(&subtable->mask.masks, tp_src);/* 获取四层源端口掩码 */
        /* 两者相与 */
        value = ((OVS_FORCE ovs_be32 *)flow)[TP_PORTS_OFS32] & mask;/* 与流进行掩码与操作，得到我们需要查找的值 */
        /* 对端口树进行查找 */
        mbits = trie_lookup_value(&subtable->ports_trie, &value, &plens, 32);

        ((OVS_FORCE ovs_be32 *)&wc->masks)[TP_PORTS_OFS32] |=
            mask & be32_prefix_mask(mbits);

        goto no_match;
```

```
    }

    /* Must unwildcard all the fields, as they were looked at. */
    /* 必须非通配所有的域，因为他们需要被查找匹配，这里采用的是或操作
    ** 查询多个表格后会命中多个规则，mask进行或
    */
    flow_wildcards_fold_minimask(wc, &subtable->mask);
    //返回掩码
    return rule;

no_match:
    /* Unwildcard the bits in stages so far, as they were used in
determining
     * there is no match. */
    flow_wildcards_fold_minimask_in_map(wc, &subtable->mask, stages_map);
    return NULL;
}
```

## classifier流程分析

从cls->subtables第一个子表开始循环遍历每一个符合要求的子表，对子表调用函数find_match_wc进行处理。
函数find_match_wc进行如下处理：
对flow在子表的前三个段中进行trie过滤，不符合命中条件的话，开始下一个子表。
计算flow在子表在前三个段中的hash值，在subtable->indices[i]中查找是否有对应的hash值，前三个段就是通过hash值进行过滤。

前面两条都满足要求后，对最后一个段进行trie过滤，不符合命中条件的话，开始下一个子表。对最后一个段进行hash，使用hash值在子表中调用函数find_match进行精确匹配。所有子表遍历完毕后，开始对匹配中命中的规则进行关联匹配处理，找出最高优先级的规则。

### 相关函数

classifier_init

```
/* Initializes 'cls' as a classifier that initially contains no
classification
 * rules. 初始化一个分类器
 * 参数flow_segments为段边界数组，即flow_segment_u64s */
void
classifier_init(struct classifier *cls, const uint8_t *flow_segments)
{
    cls->n_rules = 0;/* 规则个数为0 */
    cmap_init(&cls->subtables_map);/* 初始化子表map */
    pvector_init(&cls->subtables);//初始化子表数组
    cls->n_flow_segments = 0;/* 初始化段为0 */
    if (flow_segments) {/* 如果存在段匹配器，那么初始化段匹配器的结束地址，这里只初始
化了三个段 */
        while (cls->n_flow_segments < CLS_MAX_INDICES
               && *flow_segments < FLOW_U64S) {
            cls->flow_segments[cls->n_flow_segments++] = *flow_segments++;
        }
    }
    cls->n_tries = 0;/* 默认前缀树匹配器的个数为0 */
    for (int i = 0; i < CLS_MAX_TRIES; i++) {/* 初始化前缀树 */
        trie_init(cls, i, NULL);
```

```
    }
    cls->publish = true; //是否对用户可见
}
```

classifier_insert

```
/* Inserts 'rule' into 'cls'.  Until 'rule' is removed from 'cls', the caller
 * must not modify or free it.
 *
 * 'cls' must not contain an identical rule (including wildcards, values of
 * fixed fields, and priority).  Use classifier_find_rule_exactly() to find
 * such a rule.
 * 插入规则到分类器，直到规则从分类器中删除，调用者不能修改和释放该规则*/
void
classifier_insert(struct classifier *cls,/* 分类器 */
                        const struct cls_rule *rule,/* 规则 */
                        ovs_version_t version, /* 规则版本 */
                        const struct cls_conjunction conj[],
                        size_t n_conj)
{
    const struct cls_rule *displaced_rule
        = classifier_replace(cls, rule, version, conj, n_conj);
    ovs_assert(!displaced_rule);
}

/* Inserts 'rule' into 'cls' in 'version'.  Until 'rule' is removed from 'cls',
 * the caller must not modify or free it.
 *
 * If 'cls' already contains an identical rule (including wildcards, values of
 * fixed fields, and priority) that is visible in 'version', replaces the old
 * rule by 'rule' and returns the rule that was replaced.  The caller takes
 * ownership of the returned rule and is thus responsible for destroying it
 * with cls_rule_destroy(), after RCU grace period has passed (see
 * ovsrcu_postpone()).
 *
 * Returns NULL if 'cls' does not contain a rule with an identical key, after
 * inserting the new rule.  In this case, no rules are displaced by the new
 * rule, even rules that cannot have any effect because the new rule matches a
 * superset of their flows and has higher priority.
 *
 * 如果不包含一个一样的规则，则返回空。
 */
const struct cls_rule *
classifier_replace(struct classifier *cls, const struct cls_rule *rule,
                   ovs_version_t version,
                   const struct cls_conjunction *conjs, size_t n_conjs)
```

```
{
    struct cls_match *new;
    struct cls_subtable *subtable;
    uint32_t ihash[CLS_MAX_INDICES];
    struct cls_match *head;
    unsigned int mask_offset;
    size_t n_rules = 0;
    uint32_t basis;
    uint32_t hash;
    unsigned int i;

    /* 'new' is initially invisible to lookups. */
    new = cls_match_alloc(rule, version, conjs, n_conjs);/* 分配一个新的匹配器 */
    ovsrcu_set(&CONST_CAST(struct cls_rule *, rule)->cls_match, new);/* 设置规则的匹配器 */

    /* 根据规则的掩码进行子表的查找 */
    subtable = find_subtable(cls, rule->match.mask);/* 根据掩码查找子表 */
    if (!subtable) {/* 没有找到子表，新建一个子表，将规则插入新的子表  */
        subtable = insert_subtable(cls, rule->match.mask);
    }

    /* Compute hashes in segments. */
    /* 计算在段里面的hash值 */
    basis = 0;
    mask_offset = 0;
    for (i = 0; i < subtable->n_indices; i++) {/*遍历每一个段，计算每一个段的hash值，这些hash值可以用来过滤 */
        /* 对规则的每一个段进行hash */
        ihash[i] = minimatch_hash_range(&rule->match, subtable->index_maps[i],
                                        &mask_offset, &basis);
    }
    /* 对最后一个段进行hash */
    hash = minimatch_hash_range(&rule->match, subtable->index_maps[i],
                                &mask_offset, &basis);

    /* 根据hash值对子表中的规则进行查找，最后一个段的hash值作为hash桶的key */
    head = find_equal(subtable, rule->match.flow, hash);
    if (!head) {/* 没有找到，添加规则 */
        /* Add rule to tries.
         *
         * Concurrent readers might miss seeing the rule until this update,
         * which might require being fixed up by revalidation later. */
        for (i = 0; i < cls->n_tries; i++) {/* 遍历分类器的每一个前缀树 */
            if (subtable->trie_plen[i]) {/* 如果该子表的对应的掩码长度存在，插入该前缀树 */
                trie_insert(&cls->tries[i], rule, subtable->trie_plen[i]);/* 这个是前缀长度 */
            }
        }

        /* Add rule to ports trie. */
        /* 添加规则到端口树中，只对传输层源端口进行树插入 */
        if (subtable->ports_mask_len) {
```

```
91              /* We mask the value to be inserted to always have the
    wildcarded
92               * bits in known (zero) state, so we can include them in
    comparison
93               * and they will always match (== their original value does
    not
94               * matter). */
95              ovs_be32 masked_ports = minimatch_get_ports(&rule->match);
96
97              trie_insert_prefix(&subtable->ports_trie, &masked_ports,
98                                 subtable->ports_mask_len);
99          }
100
101         /* Add new node to segment indices. 将hash值插入子表段hash过滤表 */
102         for (i = 0; i < subtable->n_indices; i++) {
103             ccmap_inc(&subtable->indices[i], ihash[i]);
104         }
105         n_rules = cmap_insert(&subtable->rules, &new->cmap_node, hash);
106     } else {   /* Equal rules exist in the classifier already. */
107         struct cls_match *prev, *iter;
108
109         /* Scan the list for the insertion point that will keep the list
    in
110          * order of decreasing priority.  Insert after rules marked
    invisible
111          * in any version of the same priority. */
112         FOR_EACH_RULE_IN_LIST_PROTECTED (iter, prev, head) {
113             if (rule->priority > iter->priority
114                 || (rule->priority == iter->priority
115                     && !cls_match_is_eventually_invisible(iter))) {
116                 break;
117             }
118         }
119
120         /* Replace 'iter' with 'new' or insert 'new' between 'prev' and
121          * 'iter'. */
122         if (iter) {
123             struct cls_rule *old;
124
125             if (rule->priority == iter->priority) {
126                 cls_match_replace(prev, iter, new);
127                 old = CONST_CAST(struct cls_rule *, iter->cls_rule);
128             } else {
129                 cls_match_insert(prev, iter, new);
130                 old = NULL;
131             }
132
133             /* Replace the existing head in data structures, if rule is
    the new
134              * head. */
135             if (iter == head) {
136                 cmap_replace(&subtable->rules, &head->cmap_node,
137                              &new->cmap_node, hash);
138             }
139
140             if (old) {
141                 struct cls_conjunction_set *conj_set;
142
```

```
143                 conj_set = ovsrcu_get_protected(struct cls_conjunction_set
    *,
144                                                 &iter->conj_set);
145                 if (conj_set) {
146                     ovsrcu_postpone(free, conj_set);
147                 }
148
149                 ovsrcu_set(&old->cls_match, NULL); /* Marks old rule as
    removed
150                                                     * from the classifier.
    */
151                 ovsrcu_postpone(cls_match_free_cb, iter);
152
153                 /* No change in subtable's max priority or max count. */
154
155                 /* Make 'new' visible to lookups in the appropriate
    version. */
156                 cls_match_set_remove_version(new,
    OVS_VERSION_NOT_REMOVED);
157
158                 /* Make rule visible to iterators (immediately). */
159                 rculist_replace(CONST_CAST(struct rculist *, &rule->node),
160                                 &old->node);
161
162                 /* Return displaced rule.  Caller is responsible for
    keeping it
163                  * around until all threads quiesce. */
164                 return old;
165             }
166         } else {
167             /* 'new' is new node after 'prev' */
168             cls_match_insert(prev, iter, new);
169         }
170     }
171
172     /* Make 'new' visible to lookups in the appropriate version. */
173     cls_match_set_remove_version(new, OVS_VERSION_NOT_REMOVED);
174
175     /* Make rule visible to iterators (immediately). */
176     rculist_push_back(&subtable->rules_list,
177                       CONST_CAST(struct rculist *, &rule->node));
178
179     /* Rule was added, not replaced.  Update 'subtable's 'max_priority'
    and
180      * 'max_count', if necessary.
181      *
182      * The rule was already inserted, but concurrent readers may not see
    the
183      * rule yet as the subtables vector is not updated yet.  This will
    have to
184      * be fixed by revalidation later. */
185     if (n_rules == 1) {
186         subtable->max_priority = rule->priority;
187         subtable->max_count = 1;
188         pvector_insert(&cls->subtables, subtable, rule->priority);
189     } else if (rule->priority == subtable->max_priority) {
190         ++subtable->max_count;
191     } else if (rule->priority > subtable->max_priority) {
```

```
        subtable->max_priority = rule->priority;
        subtable->max_count = 1;
        pvector_change_priority(&cls->subtables, subtable, rule->priority);
    }

    /* Nothing was replaced. */
    cls->n_rules++;

    if (cls->publish) {/* 更新子表 */
        pvector_publish(&cls->subtables);
    }

    return NULL;
}
```

classifier_lookup

```
/* Finds and returns the highest-priority rule in 'cls' that matches 'flow' and
 * that is visible in 'version'.  Returns a null pointer if no rules in 'cls'
 * match 'flow'.  If multiple rules of equal priority match 'flow', returns one
 * arbitrarily.
 *
 * If a rule is found and 'wc' is non-null, bitwise-OR's 'wc' with the
 * set of bits that were significant in the lookup.  At some point
 * earlier, 'wc' should have been initialized (e.g., by
 * flow_wildcards_init_catchall()).
 *
 * 'flow' is non-const to allow for temporary modifications during the lookup.
 * Any changes are restored before returning.
 * 从cls分类器中找到一个能够匹配flow的最高优先级的规则。
 * 如果在分类器cls没有规则匹配这个flow，则返回空，如果多个规则匹配该流，则返回一个最高
 * 优先级的。
 * 可以临时修改flow
 */
const struct cls_rule *
classifier_lookup(const struct classifier *cls, /* 分类器 */ovs_version_t version,/* 版本 */
                  struct flow *flow,/* 需要查找的flow */ struct flow_wildcards *wc/* 通配符 */)
{
    return classifier_lookup__(cls, version, flow, wc, true);
}

/* Like classifier_lookup(), except that support for conjunctive matches can be
 * configured with 'allow_conjunctive_matches'.  That feature is not exposed
 * externally because turning off conjunctive matches is only useful to avoid
 * recursion within this function itself.
 *
```

```
 * 'flow' is non-const to allow for temporary modifications during the
lookup.
 * Any changes are restored before returning. */
static const struct cls_rule *
classifier_lookup__(const struct classifier *cls, ovs_version_t version,/*
分类器及其版本号 */
                    struct flow *flow, struct flow_wildcards *wc,/* 匹配流
和通配符 */
                    bool allow_conjunctive_matches)/* 是否允许conjunctive匹
配 */
{
    struct trie_ctx trie_ctx[CLS_MAX_TRIES];
    const struct cls_match *match;
    /* Highest-priority flow in 'cls' that certainly matches 'flow'. */
    /* 指向最高优先级的flow在cls中 */
    const struct cls_match *hard = NULL;
    int hard_pri = INT_MIN;     /* hard ? hard->priority : INT_MIN. */

    /* Highest-priority conjunctive flows in 'cls' matching 'flow'.  Since
     * these are (components of) conjunctive flows, we can only know
whether
     * the full conjunctive flow matches after seeing multiple of them.
Thus,
     * we refer to these as "soft matches". 我们只有在查看了所有的关联匹配后
     * 我们才能知道是全匹配了，因此我们将其称之为软匹配
     * 最高优先级的关联匹配，在分类器中，使用最高优先级的关联flows匹配flow
     * */
    struct cls_conjunction_set *soft_stub[64];/* 关联匹配集合，我们默认64个集合
*/
    struct cls_conjunction_set **soft = soft_stub;/* 软匹配集合 */
    size_t n_soft = 0, allocated_soft = ARRAY_SIZE(soft_stub);/* 64个元素
*/
    int soft_pri = INT_MIN;    /* n_soft ? MAX(soft[*]->priority) :
INT_MIN. */

    /* Synchronize for cls->n_tries and subtable->trie_plen.  They can
change
     * when table configuration changes, which happens typically only on
     * startup. */
    atomic_thread_fence(memory_order_acquire);

    /* Initialize trie contexts for find_match_wc(). */
    /* 初始化trie上下文用于查找wc */
    for (int i = 0; i < cls->n_tries; i++) {
        trie_ctx_init(&trie_ctx[i], &cls->tries[i]);
    }

    /* Main loop. */
    /* 主循环，遍历所有的子表，从这里的循环可以看出，每一次循环处理一个子表，由于子表掩
码相同，所以一次返回
    ** 与掩码区域相同的多个规则的公用match，该match是一个单链表，按照优先级高低进行排
列。
    */
    struct cls_subtable *subtable;
    PVECTOR_FOR_EACH_PRIORITY (subtable, hard_pri + 1, 2, sizeof
*subtable,
                               &cls->subtables) {
        struct cls_conjunction_set *conj_set;/* 关联器集合 */
```

```
75
76          /* Skip subtables with no match, or where the match is lower-
priority
77           * than some certain match we've already found.
78           * 跳过没有相同的match的子表，或者匹配的优先级比先前匹配的较低
79           */
80          match = find_match_wc(subtable, version, flow, trie_ctx, cls-
>n_tries,
81                                wc);
82          if (!match || match->priority <= hard_pri) {/* 过滤掉有先级比已经匹配
的规则低或者相等的 */
83              continue;
84          }
85
86          /* 获取规则的关联匹配 */
87          conj_set = ovsrcu_get(struct cls_conjunction_set *, &match-
>conj_set);
88          if (!conj_set) {/* 该匹配项没有关联匹配项，直接作为硬匹配 */
89              /* 'match' isn't part of a conjunctive match.  It's the best
90               * certain match we've got so far, since we know that it's
91               * higher-priority than hard_pri.
92               *
93               * (There might be a higher-priority conjunctive match.  We
can't
94               * tell yet.) */
95              hard = match;
96              hard_pri = hard->priority;/* 当前命中的最高优先级的硬匹配 */
97          } else if (allow_conjunctive_matches) {/* 存在关联匹配，并且允许关联匹
配 */
98              /* 'match' is part of a conjunctive match.  Add it to the
list. */
99              /* 该match是关联匹配的一部分，将其添加到链表中 */
100             if (OVS_UNLIKELY(n_soft >= allocated_soft)) {/* 超过了我们能够容
纳的关联个数，需要进行重新分配 */
101                 struct cls_conjunction_set **old_soft = soft;/* 先保存老的匹
配 */
102
103                 allocated_soft *= 2;/* 进行指数扩展 */
104                 soft = xmalloc(allocated_soft * sizeof *soft);/* 分配软匹配
数组 */
105                 memcpy(soft, old_soft, n_soft * sizeof *soft);/* 将原来的内
容拷贝进去 */
106                 if (old_soft != soft_stub) {/* 如果更换了软匹配，释放以前的内容
*/
107                     free(old_soft);
108                 }
109             }
110             /* 将新增加的软匹配加入 */
111             soft[n_soft++] = conj_set;
112
113             /* Keep track of the highest-priority soft match. */
114             /* 如果原来的有先级小于现在的这个规则，更新优先级 */
115             if (soft_pri < match->priority) {
116                 soft_pri = match->priority;
117             }
118         }
119     }
120
```

```
121     /* In the common case, at this point we have no soft matches and we
    can
122      * return immediately.  (We do the same thing if we have potential
    soft
123      * matches but none of them are higher-priority than our hard match.)
124      * 在通常情况下：到这里的时候，我们没有软匹配，我们将会立即返回。
125      * 如果我们有潜在的软匹配，我们会做同样的事情*/
126     if (hard_pri >= soft_pri) {/* 如果硬匹配的优先级大于等于软匹配，说明我们将会选
    择硬匹配规则，软匹配如果分配了内存则需要释放 */
127         if (soft != soft_stub) {
128             free(soft);/* 释放我们分配的内存 */
129         }
130         return hard ? hard->cls_rule : NULL;/* 返回命中的规则 */
131     }
132
133     /* At this point, we have some soft matches.  We might also have a
    hard
134      * match; if so, its priority is lower than the highest-priority soft
135      * match.
136      * 能够走到这里，说明我们有多个软匹配，我们需要逐个将软匹配的关联匹配变成新的
137      * 全关联匹配 */
138
139     /* Soft match loop.
140      * 外层循环，遍历每一个优先级，通过第一个for语句选出本次循环的最高优先级。
141      * Check whether soft matches are real matches. 检查软匹配是否是一个真实的
    全匹配，即关联规则的每一个维度都命中了，则软匹配升级为硬匹配 */
142     for (;;) {
143         /* Delete soft matches that are null.  This only happens in second
    and
144          * subsequent iterations of the soft match loop, when we drop back
    from
145          * a high-priority soft match to a lower-priority one.
146          *
147          * Also, delete soft matches whose priority is less than or equal
    to
148          * the hard match's priority.  In the first iteration of the soft
149          * match, these can be in 'soft' because the earlier main loop
    found
150          * the soft match before the hard match.  In second and later
    iteration
151          * of the soft match loop, these can be in 'soft' because we
    dropped
152          * back from a high-priority soft match to a lower-priority soft
    match.
153          *
154          * It is tempting to delete soft matches that cannot be satisfied
155          * because there are fewer soft matches than required to satisfy
    any of
156          * their conjunctions, but we cannot do that because there might
    be
157          * lower priority soft or hard matches with otherwise identical
158          * matches.  (We could special case those here, but there's no
159          * need--we'll do so at the bottom of the soft match loop anyway
    and
160          * this duplicates less code.)
161          *
162          * It's also tempting to break out of the soft match loop if
    'n_soft ==
```

```
         * 1' but that would also miss lower-priority hard matches.  We
could
         * special case that also but again there's no need. */
        /* 过滤掉所有的优先级比硬匹配低的，空的软匹配。 */
        for (int i = 0; i < n_soft; ) {
            if (!soft[i] || soft[i]->priority <= hard_pri) {
                soft[i] = soft[--n_soft];
            } else {
                i++;
            }
        }
        /* 如果所有的软匹配都不合格的话，直接跳出 */
        if (!n_soft) {
            break;
        }

        /* Find the highest priority among the soft matches.  (We know
this
         * must be higher than the hard match's priority; otherwise we
would
         * have deleted all of the soft matches in the previous loop.)
Count
         * the number of soft matches that have that priority.
         * 从所有的软匹配中找到最高优先级的软匹配，这个软匹配必须高于硬匹配。同时计算出
这个优先级的软匹配的个数
         */
        soft_pri = INT_MIN;
        int n_soft_pri = 0;/* 获取一个最高的软匹配优先级及其个数 */
        for (int i = 0; i < n_soft; i++) {/* 遍历剩下的合规的软匹配 */
            if (soft[i]->priority > soft_pri) {
                soft_pri = soft[i]->priority;/* 更新软匹配的有先级 */
                n_soft_pri = 1;
            } else if (soft[i]->priority == soft_pri) {
                n_soft_pri++;/* 计算同一最高优先级的软匹配的个数 */
            }
        }
        ovs_assert(soft_pri > hard_pri);

        /* Look for a real match among the highest-priority soft matches.
         *
         * It's unusual to have many conjunctive matches, so we use stubs
to
         * avoid calling malloc() in the common case.  An hmap has a
built-in
         * stub for up to 2 hmap_nodes; possibly, we would benefit a
variant
         * with a bigger stub. */
        struct conjunctive_match cm_stubs[16];
        struct hmap matches;

        hmap_init(&matches);/* 初始暂存hash，同一个优先级的所有软匹配都使用该hash
表，每次循环清零。这样可以组合
                             * 多个子表中的软匹配形成硬匹配。
                             */
        for (int i = 0; i < n_soft; i++) {/* 遍历每一个软匹配，只处理我们需要的
优先级，即本次选出的最高优先级
                                           * 一次循环处理一个该优先级的软匹配。
设置一个维度bit。
```

```
                                            */
            uint32_t id;

            if (soft[i]->priority == soft_pri/* 如果是我们需要的优先级 */
                && find_conjunctive_match(soft[i], n_soft_pri, &matches,/*
根据关联id，查找关联匹配 */
                                          cm_stubs, ARRAY_SIZE(cm_stubs),
                                          &id)) {/* 将该软匹配的每一个关联匹配
进行过滤，放到matches杂凑表中 */
                uint32_t saved_conj_id = flow->conj_id;/* 记录流的关联动作id
*/
                const struct cls_rule *rule;

                flow->conj_id = id; /* 最终的关联id，根据关联id进行第二步查找 */
                rule = classifier_lookup__(cls, version, flow, wc,
false);/* 进行流表查找，不允许处理关联动作，因为前面已经处理过了 */
                flow->conj_id = saved_conj_id;

                if (rule) {/* 找到规则 */
                    free_conjunctive_matches(&matches,
                                             cm_stubs,
ARRAY_SIZE(cm_stubs));
                    if (soft != soft_stub) {
                        free(soft);
                    }
                    return rule;
                }
            }
        }
        free_conjunctive_matches(&matches, cm_stubs,
ARRAY_SIZE(cm_stubs));

        /* There's no real match among the highest-priority soft matches.
         * However, if any of those soft matches has a lower-priority but
         * otherwise identical flow match, then we need to consider those
for
         * soft or hard matches.
         *
         * The next iteration of the soft match loop will delete any null
         * pointers we put into 'soft' (and some others too).
         * 走到这里说明，本优先级没有形成硬匹配
         */
        for (int i = 0; i < n_soft; i++) {
            if (soft[i]->priority != soft_pri) {
                continue;
            }

            /* Find next-lower-priority flow with identical flow match. */
            /* 从同样的匹配条件中找到下一个优先级的匹配，如果该匹配是关联匹配，则
            ** 准备下一轮。否则作为硬匹配。
            */
            match = next_visible_rule_in_list(soft[i]->match, version);
            if (match) {
                soft[i] = ovsrcu_get(struct cls_conjunction_set *,
                                     &match->conj_set);
                if (!soft[i]) {//该规则不是关联匹配，作为硬匹配与原来的硬匹配比较优
先级。
                    /* The flow is a hard match; don't treat as a soft
```

```
260                      * match. */
261                     if (match->priority > hard_pri) {
262                         hard = match;
263                         hard_pri = hard->priority;
264                     }
265                 }
266             } else {
267                 /* No such lower-priority flow (probably the common case).
    */
268                 /* 为空，需要清除，在下一次循环中会进行计数修改 */
269                 soft[i] = NULL;
270             }
271         }
272     }
273
274     /* 如果是新分配的软匹配保存控制块，则释放 */
275     if (soft != soft_stub) {
276         free(soft);
277     }
278     return hard ? hard->cls_rule : NULL;
279 }
```