删除流表主要有三种方式。

controller/ovs-ofctl主动删除流表

精确删除必须要匹配所有字段（包括优先级）才能删除对应单条流表，后者要求指定的字段是流表的一个子集。

```
//比如添加如下两条流表
ovs-ofctl add-flow br10 "priority=80, in_port=2,ip, nw_dst=1.2.0.0/16, action=1"
ovs-ofctl add-flow br10 "priority=100, in_port=2, ip, nw_dst=1.1.1.0/24, action=1"

//使用OFPFC_DELETE删除流表时，可以只指定in_port或者ip就可以将上面两条流表删除
ovs-ofctl del-flows br10 "in_port=2"
ovs-ofctl del-flows br10 "ip"

//使用OFPFC_DELETE_STRICT(加上选项 --strict)删除流表时，需要指定流表的所有内容，包括优先级
ovs-ofctl --strict del-flows br10 "priority=100, in_port=2, ip, nw_dst=1.1.1.0/24"
```

流表超时机制

添加流表时如果指定idle_timeout或者hard_timeout参数，则流表超时后将被删除。如果不指定这俩参数，则默认不会被超时机制删除。hard_timeout指定的超时时间是从创建流表，或者修改流表开始计时，超时时间到后，不管此流表有没有被使用，都会被删除。idle_timeout指定流表空闲超时时间，从最近流表被使用开始计时，如果指定时间内此流表没有被使用，则被删除。

强制回收机制

添加流表时，如果当前流表个数大于等于最大流表个数，则判断是否可以强制回收之前添加的流表。可以通过Flow_Table里的overflow_policy参数指定当前流表个数大于等于最大流表个数时的行为，如果为refuse则拒绝添加新流表，如果为evict则强制删除即将超时的流表。强制回收只考虑指定了超时时间的流表。

源码分析

主动删除流表

```
static enum ofperr
ofproto_flow_mod_start(struct ofproto *ofproto, struct ofproto_flow_mod *ofm)
```

```
    rule_collection_init(&ofm->old_rules);
    rule_collection_init(&ofm->new_rules);

    switch (ofm->command) {
    case OFPFC_ADD:
        error = add_flow_start(ofproto, ofm);
        break;
    case OFPFC_MODIFY:
        error = modify_flows_start_loose(ofproto, ofm);
        break;
    case OFPFC_MODIFY_STRICT:
        error = modify_flow_start_strict(ofproto, ofm);
        break;
    case OFPFC_DELETE:
        error = delete_flows_start_loose(ofproto, ofm);
        break;
    case OFPFC_DELETE_STRICT:
        error = delete_flow_start_strict(ofproto, ofm);
        break;
    default:
        OVS_NOT_REACHED();
    }


ofproto_flow_mod_start
    delete_flows_start_loose(ofproto, ofm);
        //先根据指定的字段ofm->criteria查找需要删除的rule，这是用的是loose方式查找
        collect_rules_loose(ofproto, &ofm->criteria, rules);
        delete_flows_start__(ofproto, ofm->version, rules);

    delete_flow_start_strict(ofproto, ofm);
        //先根据指定的字段ofm->criteria查找需要删除的rule，这是用的是strict方式查找，必须精确匹配
        collect_rules_strict(ofproto, &ofm->criteria, rules);
        delete_flows_start__(ofproto, ofm->version, rules);

//将rules从ofproto中删除
static void
delete_flows_start__(struct ofproto *ofproto, ovs_version_t version,
                     const struct rule_collection *rules)
    struct rule *rule;

    RULE_COLLECTION_FOR_EACH (rule, rules) {
        struct oftable *table = &ofproto->tables[rule->table_id];
        //流表个数减一
        table->n_flows--;
        //设置versions->remove_version为version，表示此流表从version开始就被删除了
        cls_rule_make_invisible_in_version(&rule->cr, version);
            struct cls_match *cls_match = get_cls_match_protected(rule);
            cls_match_set_remove_version(cls_match, remove_version);
                versions_set_remove_version(&rule->versions, version);
                    atomic_store_relaxed(&versions->remove_version, version);
        //一个流表会插入多个地方保存，这个只是将流表从ofproto中删除，后面会将流表从分类器中删除
```

```
        /* Removes 'rule' from the ofproto data structures.  Caller may
have deferred
         * the removal from the classifier. */
        /* Remove rule from ofproto data structures. */
        ofproto_rule_remove__(ofproto, rule);
    }

```

将流表从分类器中删除

```
ofproto_flow_mod_finish(ofproto, &ofm, req);
    delete_flows_finish(ofproto, ofm, req);
        delete_flows_finish__(ofproto, &ofm->old_rules, OFPRR_DELETE, req);
            remove_rules_postponed(rules);
                ovsrcu_postpone(remove_rules_rcu,
rule_collection_detach(rules));
                    remove_rule_rcu__(rule);
                        cls_rule_visible_in_version(&rule->cr,
OVS_VERSION_MAX)
                        //将rule从分类器删除
                        //回调调用
                        ofproto->ofproto_class->rule_delete(rule)
                        classifier_remove(&table->cls, &rule->cr)
```

再分析下`classifier_remove`的流程

```
classifier_remove(struct classifier *cls, const struct cls_rule *cls_rule)

    /* 获取对应rule和sub_table */
    rule = get_cls_match_protected(cls_rule);
    subtable = find_subtable(cls, cls_rule->match.mask);

    /* 根据分段的mask和miniflow获取ihash */
    for (i = 0; i < subtable->n_indices; i++) {
        ihash[i] = minimatch_hash_range(&cls_rule->match,
                                        subtable->index_maps[i],
                                        &mask_offset, &basis);
    }

    /* 找到对应的subtab */
    hash = minimatch_hash_range(&cls_rule->match, subtable->index_maps[i],
                                &mask_offset, &basis);

    head = find_equal(subtable, cls_rule->match.flow, hash);
    /* 如果不是head rule， 走check priority流程
      如果找到了replace rule，则覆盖，走check priority 流程
      */

    /* 移除对应前缀树 */
    trie_remove(&cls->tries[i], cls_rule, subtable->trie_plen[i]);

    /* 移除对应分段 */
    n_rules = cmap_remove(&subtable->rules, &rule->cmap_node, hash);

    /* 如果该subtable下面没rule了则销毁该subtable */
    if (n_rules == 0) {
```

```
        destroy_subtable(cls, subtable);
    }

```