OpenvSwitch 代码分析

最新版: yeasy@github

更新历史:
V0.7: 2012-10-18
　完成全部章节。

1. 源代码结构

1.1. 配置相关

acinclude.m4

宏定义文件，供 aclocal/automake 使用。

configure.ac

autoconf 的宏文件

boot.sh

执行 autoreconf 命令

子目录

build-aux/

m4/

Makefile.am

整体的 automake 配置文件

manpages.mk

自动生成的 man 配置文件

1.2. Install 相关

对应到各种应用场景下的安装指南

INSTALL

INSTALL.bridge

INSTALL.KVM

INSTALL.Libvirt

INSTALL.RHEL

INSTALL.SSL

INSTALL.userspace

INSTALL.XenServer

rhel/

Red Hat 系统集成

1.3. 核心代码

datapath/

ovs datapath 代码目录

vswitchd/

ovs-switchd 程序代码

ovsdb/

ovs 数据库管理代码

include/

头文件代码目录

lib/

库文件目录

ofproto/

解析 openflow 协议

1.4. 说明文件等

AUTHORS

作者信息

CodingStyle

编程风格建议

COPYING

许可说明

DESIGN

设计原则，处理 openflow 协议相关细节和考虑

FAQ

NEWS

NOTICE

README

README-gcov

REPORTING-BUGS

WHY-OVS

注：1.8.90 版本，所有 C 源码行数为 189964，其中 datapath 模块 28131，vswitchd 模块 5888，ovsdb 模块 10549，lib 为 101592，ofproto 为 18571。

find openvswitch -name "*.[ch]" | xargs cat | wc –l

1.5. 其他文件

debian/

IntegrationGuide

集成到其他 hypervisor

PORTING

移植说明

python/

SubmittingPatches

补丁提交

tests/

测试代码

third-party/

支持第三方的插件，包括让 tcpdump 支持解析 of 协议的补丁。

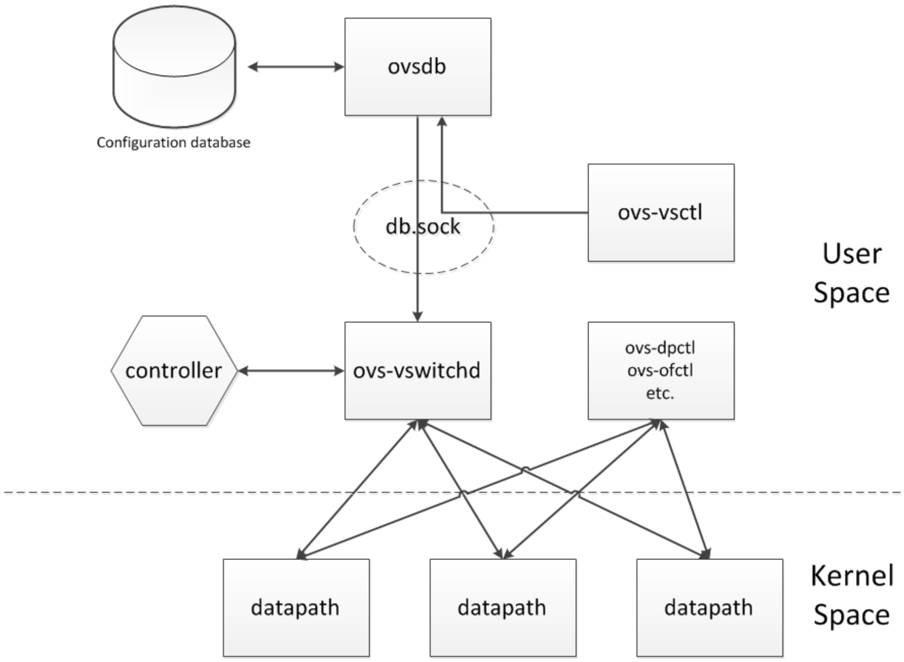
utilities/

小工具，包括用户操作命令，例如 ovs-dpctl、ovs-ofctl、ovs-controller、ovs-vsctl 等等。其中 ovs-vsctl 是主要的对配置数据库进行交互的接口。

xenserver/

xenserver 集成信息。

1.6. 整体功能逻辑



2. datapath 模块

2.1. 整体分析

datapath 模块的代码主要包括如下几个关键子模块：主文件 datapath.h/c，vport 的实现 vport-(generic/gre/capwap/netdev/internal/patch)，genl 子模块。

2.1.1. action 模块

action.c 中定义了对网包执行操作的各个接口。

包括对 vlan 头的处理，对 skb 执行一系列给定的操作，发出网包，发 skb 给用户态（ovsd），采样，设置包头各个域的属性等。

2.1.2. flow 模块

包括 flow.h 和 flow.c。

定义维护交换机本地流表相关的数据结构和操作，包括流表结构的创建、更新、删除，对每条流的管理等。

2.1.3. genl 模块

genl-exec.h 中定义了对 genl 的相关操作，包括

```
typedef int (*genl_exec_func_t)(void *data);
int genl_exec(genl_exec_func_t func, void *data);
int genl_exec_init(void);
void genl_exec_exit(void);
```

genl_exec_family 的定义为

```
static struct genl_family genl_exec_family = {
  .id = GENL_ID_GENERATE, //channel number: will be assigned by the controller
  .name = "ovs_genl_exec",
  .version = 1,
};
```

genl_exec_ops[]的定义为

```
static struct genl_ops genl_exec_ops[] = {
  {
   .cmd = GENL_EXEC_RUN, //reference the operation
   .doit = genl_exec_cmd, //the callback function
   .flags = CAP_NET_ADMIN,
  },
};
```

2.1.4. vport 和 vport_ops

关键的逻辑实现在 vport 子模块中。在 vport.h/c 中定义了抽象的 vport 结构。对外的对 vport 进行操作的接口如下

```
int ovs_vport_init(void);
void ovs_vport_exit(void);

struct vport *ovs_vport_add(const struct vport_parms *);
void ovs_vport_del(struct vport *);

struct vport *ovs_vport_locate(struct net *net, const char *name);

int ovs_vport_set_addr(struct vport *, const unsigned char *);
void ovs_vport_set_stats(struct vport *, struct ovs_vport_stats *);
void ovs_vport_get_stats(struct vport *, struct ovs_vport_stats *);

int ovs_vport_set_options(struct vport *, struct nlattr *options);
int ovs_vport_get_options(const struct vport *, struct sk_buff *);

int ovs_vport_send(struct vport *, struct sk_buff *);
```

这些接口对外提供统一的用户操作界面。部分并没有立刻定义，即使定义的接口中，大部分依次或者单独调用某种类型的 vport 上绑定的 vport_ops 中提供的接口，对所有支持的 vport 进行操作。例如，初始化过程中，实际上是初始化了一个 vport_ops_list[]，依次初始化不同类型的 vport，并放到该 list 中。再比如 ovs_vport_add 接口实际上先进行查找，找到给定的类型之后，进行对应的操作。

而具体到某个 vport，其能进行操作的接口在 vport_ops 结构体中声明，为

```
struct vport_ops {
  enum ovs_vport_type type;
  u32 flags;

  /* Called at module init and exit respectively. */
  int (*init)(void);
  void (*exit)(void);

  /* Called with RTNL lock. */
  struct vport *(*create)(const struct vport_parms *);
  void (*destroy)(struct vport *);

  int (*set_options)(struct vport *, struct nlattr *);
  int (*get_options)(const struct vport *, struct sk_buff *);

  int (*set_addr)(struct vport *, const unsigned char *);

  /* Called with rcu_read_lock or RTNL lock. */
  const char *(*get_name)(const struct vport *);
  const unsigned char *(*get_addr)(const struct vport *);
  void (*get_config)(const struct vport *, void *);
  struct kobject *(*get_kobj)(const struct vport *);

  unsigned (*get_dev_flags)(const struct vport *);
  int (*is_running)(const struct vport *);
  unsigned char (*get_operstate)(const struct vport *);

  int (*get_ifindex)(const struct vport *);

  int (*get_mtu)(const struct vport *);

  int (*send)(struct vport *, struct sk_buff *);
};
```

目前，vport_ops 支持 5 种类型，分别为

```
static const struct vport_ops *base_vport_ops_list[] = {
  &ovs_netdev_vport_ops,
  &ovs_internal_vport_ops,
  &ovs_patch_vport_ops,
  &ovs_gre_vport_ops,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,26)
  &ovs_capwap_vport_ops,
#endif
};
```

以 ovs_netdev_vport_ops 为例，定义了一系列的函数指针。

```
const struct vport_ops ovs_netdev_vport_ops = {
  .type            = OVS_VPORT_TYPE_NETDEV,
  .flags      =    VPORT_F_REQUIRED,
  .init            = netdev_init,
  .exit            = netdev_exit,
  .create          = netdev_create,
  .destroy         = netdev_destroy,
  .set_addr        = ovs_netdev_set_addr,
  .get_name        = ovs_netdev_get_name,
  .get_addr        = ovs_netdev_get_addr,
  .get_kobj        = ovs_netdev_get_kobj,
  .get_dev_flags   = ovs_netdev_get_dev_flags,
  .is_running      = ovs_netdev_is_running,
  .get_operstate   = ovs_netdev_get_operstate,
  .get_ifindex     = ovs_netdev_get_ifindex,
  .get_mtu         = ovs_netdev_get_mtu,
  .send            = netdev_send,
};
```

2.1.5. netdev_vport

绑定到具体网络设备上的 vport 的结构。

该结构的定义十分简单，封装了一个 net_device 结构，定义如下。

```
struct netdev_vport {
  struct net_device *dev;
};
```

2.2. datapath.c

模块的主文件。实现了一个简单的交换机。

2.2.1. 注册和回收

```
module_init(dp_init); //call when this module is loaded into kernel
module_exit(dp_cleanup); //call when the module is removed from kernel
```

注册后，调用 dp_init 来完成各项初始化工作，dp_init 也是模块的主函数。

当模块被卸载时，调用 dp_cleanup 完成清理工作，回收各项数据结构。

dp_init 主要过程为：

```
genl_exec_init()
ovs_workqueues_init()
ovs_tnl_init()
ovs_flow_init()
ovs_vport_init()
register_pernet_device(&ovs_net_ops)
register_netdevice_notifier(&ovs_dp_device_notifier)
dp_register_genl()
schedule_delayed_work(&rehash_flow_wq, REHASH_FLOW_INTERVAL)
```

2.2.2. GENL_EXEC_RUN 初始化

genl_exec_init()，完成 genl_exec_run 的相关注册，仅在 KERNEL_VERSION <2.6.35 时生效。主要代码为

```
err = genl_register_family_with_ops(&genl_exec_family,
                                    genl_exec_ops, ARRAY_SIZE(genl_exec_ops));
```

注册的 generic netlink family：genl_exec_family。

```
static struct genl_family genl_exec_family = {
  .id = GENL_ID_GENERATE, //new family, will be assigned by the controller
  .name = "ovs_genl_exec",
  .version = 1,
};
```

对应操作的结构为

```
static struct genl_ops genl_exec_ops[] = {
  {
   .cmd = GENL_EXEC_RUN,
   .doit = genl_exec_cmd,
   .flags = CAP_NET_ADMIN,
  },
};
```

2.2.3. 工作队列初始化

ovs_workqueues_init()

创建 worker_thread，初始化 more_work 工作队列，轮询检测执行队列中的任务。

主要代码为

```
workq_thread = kthread_create(worker_thread, NULL, "ovs_workq"); //just create
wake_up_process(workq_thread); //run the created thread
```

2.2.4. port_table 初始化

ovs_tnl_init()，初始化 port_table 数据结构

主要代码为

```
port_table = kmalloc(PORT_TABLE_SIZE * sizeof(struct hlist_head *), GFP_KERNEL);
for (i = 0; i < PORT_TABLE_SIZE; i++)
		INIT_HLIST_HEAD(&port_table[i]);
```

2.2.5. flow_cache 初始化

ovs_flow_init()，申请 flow_cache，主要代码为

```
flow_cache = kmem_cache_create("sw_flow", sizeof(struct sw_flow), 0,0, NULL);
```

2.2.6. *vport 数据结构初始化

ovs_vport_init()，初始化 dev_table 和 vport_op 的 list，按照 base_vport_ops_list 模板，创建一个新的 vport_ops_list，并执行各个 vport_ops 实例模板的初始化函数。目前 vport_ops 实例模板包括

```
static const struct vport_ops *base_vport_ops_list[] = {
  &ovs_netdev_vport_ops,
  &ovs_internal_vport_ops,
  &ovs_patch_vport_ops,
  &ovs_gre_vport_ops,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,26)
  &ovs_capwap_vport_ops,
#endif
};
```

2.2.7. 注册网络名字空间设备

register_pernet_device(&ovs_net_ops)，注册网络名字空间设备，仅在 KERNEL_VERSION<2.6.32 时生效。主要代码为

```
void *ovs_net = kzalloc(pnet->size, GFP_KERNEL);
err = net_assign_generic(net, *pnet->id, ovs_net);
err = pnet->init(net);
```

2.2.8. 注册设备通知事件

register_netdevice_notifier(&ovs_dp_device_notifier)，注册设备通知事件，包括 NETDEV_UNREGISTER, NETDEV_CHANGENAME。该通知的数据结构为

```
struct notifier_block ovs_dp_device_notifier = {
  .notifier_call = dp_device_event
};
```

2.2.9. *dp 相关的 netlink 族和命令注册

dp_register_genl()，位于 datapath/datapath.c，负责注册 dp_genl_families 中的 family 和 op。dp_genl_families 数据结构的定义为

```
static const struct genl_family_and_ops dp_genl_families[] = {
  { &dp_datapath_genl_family,
    dp_datapath_genl_ops, ARRAY_SIZE(dp_datapath_genl_ops),
    &ovs_dp_datapath_multicast_group }, //datapath
  { &dp_vport_genl_family,
    dp_vport_genl_ops, ARRAY_SIZE(dp_vport_genl_ops),
    &ovs_dp_vport_multicast_group }, //vport
  { &dp_flow_genl_family,
    dp_flow_genl_ops, ARRAY_SIZE(dp_flow_genl_ops),
    &ovs_dp_flow_multicast_group }, //flow
  { &dp_packet_genl_family,
    dp_packet_genl_ops, ARRAY_SIZE(dp_packet_genl_ops),
    NULL },//packet
};
```

这部分实现主要的对交换机、端口、网流和网包的各个事件的处理机制的注册。各个消息总结如下：

表格 1 datapath 注册的 netlink 消息总结

类型	消息前缀	消息
datapath	OVS_DP	NEW、DEL、GET、SET
vport	OVS_VPORT	NEW、DEL、GET、SET
flow	OVS_FLOW	NEW、DEL、GET、SET
packet	OVS_PACKET	EXECUTE

注册过程主要代码为

```
const struct genl_family_and_ops *f = &dp_genl_families[i];
err = genl_register_family_with_ops(f->family, f->ops,f->n_ops);
```

2.2.10. 定期 rehash

schedule_delayed_work(&rehash_flow_wq, REHASH_FLOW_INTERVAL)，主要实现一个工作队列，定期执行 rehash。

3. vswitchd 模块

生成 ovs-vswitchd 文件，主文件为 ovs-vswitchd.c。

3.1. 整体分析

Vswitchd 模块主要包括 bridge、ofproto 等模块。作为主模块，负责解析和执行其他各个 ovs 命令。

3.1.1. bridge 模块

负责所管理的所有 datapath，对外的接口很简单，包括

```
void bridge_init(const char *remote);
void bridge_exit(void);

void bridge_run(void);
void bridge_run_fast(void);
void bridge_wait(void);

void bridge_get_memory_usage(struct simap *usage);
```

数据结构主要在 bridge.c 中定义了 bridge 结构，定义为

```
struct bridge {
    struct hmap_node node;      /* In 'all_bridges'. */
    char *name;                 /* User-specified arbitrary name. */
    char *type;                 /* Datapath type. */
    uint8_t ea[ETH_ADDR_LEN];   /* Bridge Ethernet Address. */
    uint8_t default_ea[ETH_ADDR_LEN]; /* Default MAC. */
    const struct ovsrec_bridge *cfg;

    /* OpenFlow switch processing. */
    struct ofproto *ofproto;    /* OpenFlow switch. */

    /* Bridge ports. */
    struct hmap ports;          /* "struct port"s indexed by name. */
    struct hmap ifaces;         /* "struct iface"s indexed by ofp_port. */
    struct hmap iface_by_name;  /* "struct iface"s indexed by name. */

    struct list ofpp_garbage;   /* "struct ofpp_garbage" slated for removal. */
    struct hmap if_cfg_todo;    /* "struct if_cfg"s slated for creation.
                                   Indexed on 'cfg->name'. */

    /* Port mirroring. */
    struct hmap mirrors;        /* "struct mirror" indexed by UUID. */

    /* Synthetic local port if necessary. */
    struct ovsrec_port synth_local_port;
    struct ovsrec_interface synth_local_iface;
    struct ovsrec_interface *synth_local_ifacep;
};
```

其中，最重要的是 ofproto 指针，指向一个 openflow switch，负责进行 openflow switch 的所有处理。实际上，vswitchd 的主要功能就是不断检测并调用所有 bridge 上的 ofproto，执行其上的处理函数。

3.1.2. ofproto

类型定义在 ofproto/ofproto-provider.h 中。

```
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. */
    unsigned flow_eviction_threshold; /* Threshold at which to begin flow
                                 * table eviction. Only affects the
                                 * ofproto-dpif implementation */
    bool forward_bpdu;          /* Option to allow forwarding of BPDU frames
                              * when NORMAL action is invoked. */
    char *mfr_desc;             /* Manufacturer. */
    char *hw_desc;              /* Hardware. */
    char *sw_desc;              /* Software version. */
    char *serial_desc;          /* Serial number. */
    char *dp_desc;              /* Datapath description. */
    enum ofp_config_flags frag_handling; /* One of OFPC_*.  */

    /* Datapath. */
    struct hmap ports;          /* Contains "struct ofport"s. */
    struct shash port_by_name;

    /* Flow tables. */
    struct oftable *tables;
    int n_tables;

    /* OpenFlow connections. */
    struct connmgr *connmgr;

    /* Flow table operation tracking. */
```

```
    int state;                  /* Internal state. */
    struct list pending;        /* List of "struct ofopgroup"s. */
    unsigned int n_pending;     /* list_size(&pending). */
    struct hmap deletions;      /* All OFOPERATION_DELETE "ofoperation"s. */

    /* Flow table operation logging. */
    int n_add, n_delete, n_modify; /* Number of unreported ops of each kind. */
    long long int first_op, last_op; /* Range of times for unreported ops. */
    long long int next_op_report;    /* Time to report ops, or LLONG_MAX. */
    long long int op_backoff;        /* Earliest time to report ops again. */

    /* Linux VLAN device support (e.g. "eth0.10" for VLAN 10.)
     *
     * This is deprecated.  It is only for compatibility with broken device
     * drivers in old versions of Linux that do not properly support VLANs when
     * VLAN devices are not used.  When broken device drivers are no longer in
     * widespread use, we will delete these interfaces. */
    unsigned long int *vlan_bitmap; /* 4096-bit bitmap of in-use VLANs. */
    bool vlans_changed;             /* True if new VLANs are in use. */
    int min_mtu;                    /* Current MTU of non-internal ports. */
};
```

其中最关键的是 ofproto_class，是 ofproto 交换机的具体实现，定义了对于 of 协议的处理（包括 run 和 run_fast 函数，前者处理更为全面，调用了后者）。处理函数的实现在 ofproto/ofproto-dpif.c 中。

3.1.3. run_fast()

位于 ofproto-dpif.c 中。

分析 run_fast 函数，主要完成了两个需要周期性及时完成的事情。

首先对各个 port 上调用 port_run_fast，检查是否要发送连续性检查的网包消息（CCM，参考 IEEE 802.1aq），如果是，则发出。

```
HMAP_FOR_EACH (ofport, up.hmap_node, &ofproto->up.ports) {
      port_run_fast(ofport);
  }
```

然后，检查是否有 upcall，对所有的来自 datapath 的 upcall 进行处理。

```
while (work < FLOW_MISS_MAX_BATCH) {
      int retval = handle_upcalls(ofproto, FLOW_MISS_MAX_BATCH - work);
      if (retval <= 0) {
         return -retval;
      }
      work += retval;
  }
```

3.1.4. handle_upcalls()

位于 ofproto-dpif.c 中。

该函数从对应的 dpif 中获取到 upcalls 后，对 upcalls 进行类型检查。对于 SFLOW_UPCALL 和 BAD_UPCALL，进行对应处理后释放存有 upcall 消息的 buf，而对于 MISS_UPCALL 类型，则调用 handle_miss_upcalls 进行后续的处理。

其中，upcall 的类型为 dpif_upcall（lib/dpif.h），定义为

```
/* A packet passed up from the datapath to userspace.
 *
 * If 'key' or 'actions' is nonnull, then it points into data owned by
 * 'packet', so their memory cannot be freed separately.  (This is hardly a
 * great way to do things but it works out OK for the dpif providers and
 * clients that exist so far.)
 */
struct dpif_upcall {
    /* All types. */
    enum dpif_upcall_type type;
    struct ofpbuf *packet;      /* Packet data. */
    struct nlattr *key;         /* Flow key. */
    size_t key_len;             /* Length of 'key' in bytes. */

    /* DPIF_UC_ACTION only. */
    uint64_t userdata;          /* Argument to OVS_ACTION_ATTR_USERSPACE. */
};
```

3.1.5. handle_miss_upcalls ()

位于 ofproto-dpif.c 中。

该函数从 upcall 中提取相关的流信息，把属于同一个 key 的网包放到一起，最后放进 todo list 中。最后检查 todo list 中的每个元素，调用 handle_flow_miss()进行处理。

```
HMAP_FOR_EACH (miss, hmap_node, &todo) {
        handle_flow_miss(ofproto, miss, flow_miss_ops, &n_ops);
    }
```

处理完毕后，调用 dpif_operate()（位于 vswitchd/dpif.c）执行查找到的行动。

```
for (i = 0; i < n_ops; i++) {
        dpif_ops[i] = &flow_miss_ops[i].dpif_op;
    }
```

dpif_operate(ofproto->dpif, dpif_ops, n_ops);

3.1.6. handle_miss_upcall ()

位于 ofproto/ofproto-dpif.c 中。

该函数处理给定的某个 miss_upcall。首先，先判断是否发生了精确匹配，如果发生了，则直接按照匹配结果调用 handle_flow_miss_with_facet()；如果没有精确匹配结果，则调用 handle_flow_miss_without_facet()。

3.1.7. unixctl_server 相关

主循环中调用了 unixctl_server_run()函数。该函数首先获取到远端 server 的连接，然后，执行连接中的命令，代码为。

```
LIST_FOR_EACH_SAFE (conn, next, node, &server->conns) {
        int error = run_connection(conn);
        if (error && error != EAGAIN) {
            kill_connection(conn);
        }
    }
```

3.2. ovs-vswitchd.c

主文件，其中 main()为入口主函数，执行一系列的初始化，并配置各个队列，最后是主循环，进行任务处理。分析主要代码如下

```
Int main(int argc, char *argv[])
{
    char *unixctl_path = NULL;
    struct unixctl_server *unixctl;
    struct signal *sighup;
    char *remote;
    bool exiting;
    int retval;

    proctitle_init(argc, argv); //backup orignal argvs
    set_program_name(argv[0]);
    stress_init_command(); //register stress cmds to the commands
    remote = parse_options(argc, argv, &unixctl_path);
    signal(SIGPIPE, SIG_IGN); //ignore the pipe read end signal
    sighup = signal_register(SIGHUP); //register the SIGHUP signal handler
    process_init(); //create notification pipe and register signal for child process exit
    ovsrec_init(); //todo: make clear here
    daemonize_start(); //daemonize the process

    if (want_mlockall) {
#ifdef HAVE_MLOCKALL
        if (mlockall(MCL_CURRENT | MCL_FUTURE)) {
            VLOG_ERR("mlockall failed: %s", strerror(errno));
        }
#else
        VLOG_ERR("mlockall not supported on this system");
#endif
    }
    worker_start(); //start a worker subprocess, call worker_main (receive data and
process)
    retval = unixctl_server_create(unixctl_path, &unixctl);//create a unix domain socket
    if (retval) {
        exit(EXIT_FAILURE);
```

```
    }
    unixctl_command_register("exit", "", 0, 0, ovs_vswitchd_exit, &exiting);


    bridge_init(remote);//ini the bridge, configure from the ovsdb server, register ctrl commands
    free(remote);
    exiting = false;
    while (!exiting) {
        worker_run(); //reply with the worker subprocess
        if (signal_poll(sighup)) {
            vlog_reopen_log_file();
        }
        memory_run();//monitor the memory
        if (memory_should_report()) {
            struct simap usage;
            simap_init(&usage);
            bridge_get_memory_usage(&usage);
            memory_report(&usage);
            simap_destroy(&usage);
        }
        bridge_run_fast(); //check each bridge and run it's handler
        bridge_run(); //main process part, process of pkts
        bridge_run_fast();
        unixctl_server_run(unixctl);
        netdev_run(); //run periodic functions by all network devices.
        worker_wait();
        signal_wait(sighup);
        memory_wait();
        bridge_wait();
        unixctl_server_wait(unixctl);
        netdev_wait();
        if (exiting) {
            poll_immediate_wake();
```

```
        }
        poll_block();
    }
    bridge_exit();
    unixctl_server_destroy(unixctl);
    signal_unregister(sighup);
    return 0;
}
```

3.2.1. proctitle_init(argc, argv)

复制出输入的参数列表到新的存储中，让 argv 指向这块内存，主要是为了后面的 proctitle_set()函数（在 daemonize_start()->monitor_daemon()中调用，可能修改原 argv 存储）做准备。

3.2.2. set_program_name(argv[0])

设置程序名称、版本、编译日期等信息。

3.2.3. stress_init_command()

注册 stress 相关命令（list、set、enable、disable）到 commands 结构。

3.2.4. remote = parse_options(argc, argv, &unixctl_path)

解析参数，其中 unixctl_path 存储 unixctrl 域的 sock 名，作为接受外部控制命令的渠道；而 remote 存储连接到 ovsdb 的信息，即连接到配置数据库的 sock 名。

3.2.5. signal(SIGPIPE, SIG_IGN)

忽略 pipe 读结束的信号。

3.2.6. sighup = signal_register(SIGHUP)

注册对 SIGHUP 信号（终端挂起）的处理函数。处理函数为写到 fds[1]中空字符。

3.2.7. process_init()

注册对 SIGCHLD 信号（子进程结束）的处理函数。处理函数为执行 all_process 上的所有进程。

3.2.8. ovsrec_init()

数据表结构初始化。包括 13 张数据表。表的具体结构请参考 ovsdb 的相关文档。

3.2.9. daemonize_start()

让进程变成守护程序。

3.2.10. worker_start()

开启一个 worker 子进程。子进程与主进程交互数据。

3.2.11. unixctl_server_create(unixctl_path, &unixctl)

创建一个 unixctl server（存放在 unixctl），并监听在 unixctl_path 指定的 punix 路径。该路径作为 ovs-appctl 发送命令给 ovsd 的通道。

3.2.12. unixctl_command_register("exit", "", 0, 0, vs_vswitchd_exit, &exiting)

注册 unixctl 命令。

3.2.13. bridge_init(remote)

从 remote 数据库获取配置信息，并初始化 bridge。

3.2.14. 主循环

```
exiting = false;
  while (!exiting) {
    worker_run(); //reply with the worker subprocess
    if (signal_poll(sighup)) {
      vlog_reopen_log_file();
    }
    memory_run();//monitor the memory
    if (memory_should_report()) {
      struct simap usage;

      simap_init(&usage);
      bridge_get_memory_usage(&usage);
      memory_report(&usage);
      simap_destroy(&usage);
    }
    bridge_run_fast(); //check each bridge and run it's handler
    bridge_run(); //main process part, process of pkts
    bridge_run_fast();
    unixctl_server_run(unixctl);
    netdev_run(); //run periodic functions by all network devices.

    worker_wait();
    signal_wait(sighup);
    memory_wait();
    bridge_wait();
    unixctl_server_wait(unixctl);
    netdev_wait();
    if (exiting) {
      poll_immediate_wake();
    }
    poll_block();
  }
```

3.2.15. worker_run()

执行从 worker 子进程中获取的 RPC reply，执行其中的 cb_reply 回调函数。主要过程为

```
rxbuf_run(&client_rx, client_sock, sizeof(struct worker_reply));
reply->reply_cb(&client_rx.payload, client_rx.fds,  client_rx.n_fds, reply->reply_aux);
```

3.2.16. bridge_run_fast()

执行在 all_bridge 上的每个 bridge 的 ofproto 上的 run_fast。主要是监听和处理来自 datapath 的 upcall，主要过程为

```
HMAP_FOR_EACH (br, node, &all_bridges) {
      ofproto_run_fast(br->ofproto);
  }
```

3.2.17. bridge_run()

主要对网包进行完整处理过程。包括完成必要的配置更新（在配置更新中会从数据库读入配置信息，生成必要的 bridge 和 dp 等数据结构）。以及对 all_bridge 上的每个 bridge 的 ofproto 执行 ofproto_run()。

ofproto_run()

ofproto_run()位于 ofproto.c 中，核心代码为

```
int
ofproto_run(struct ofproto *p)
{
   error = p->ofproto_class->run(p);
   if (p->ofproto_class->port_poll) {
      char *devname;

      while ((error = p->ofproto_class->port_poll(p, &devname)) != EAGAIN) {
         process_port_change(p, error, devname);
      }
   }

   /* Update OpenFlow port status for any port whose netdev has changed.
    *
    * Refreshing a given 'ofport' can cause an arbitrary ofport to be
    * destroyed, so it's not safe to update ports directly from the
    * HMAP_FOR_EACH loop, or even to use HMAP_FOR_EACH_SAFE.  Instead, we
    * need this two-phase approach. */
   sset_init(&changed_netdevs);
   HMAP_FOR_EACH (ofport, hmap_node, &p->ports) {
      unsigned int change_seq = netdev_change_seq(ofport->netdev);
      if (ofport->change_seq != change_seq) {
         ofport->change_seq = change_seq;
         sset_add(&changed_netdevs, netdev_get_name(ofport->netdev));
      }
   }
   SSET_FOR_EACH (changed_netdev, &changed_netdevs) {
      update_port(p, changed_netdev);
   }
   sset_destroy(&changed_netdevs);

   switch (p->state) {
   case S_OPENFLOW: //of commands, run its parser
```

```
        connmgr_run(p->connmgr, handle_openflow);
        break;

    case S_EVICT: //evict flow from over-limit tables
        connmgr_run(p->connmgr, NULL);
        ofproto_evict(p);
        if (list_is_empty(&p->pending) && hmap_is_empty(&p->deletions)) {
            p->state = S_OPENFLOW;
        }
        break;

    case S_FLUSH: //delete all flow table rules
        connmgr_run(p->connmgr, NULL);
        ofproto_flush__(p);
        if (list_is_empty(&p->pending) && hmap_is_empty(&p->deletions)) {
            connmgr_flushed(p->connmgr);
            p->state = S_OPENFLOW;
        }
        break;

    default:
        NOT_REACHED();
    }
}
```

首先是 p->ofproto_class->run(p)执行给定的 ofproto 中的 ofproto_class 上的 run 函数，该函数依次调用了如下函数：

调用 dpif_run()处理所有注册的 netlink notifier 的汇报事件。

调用 run_fast()处理常见的周期性事件，包括对 upcalls 的处理等。

可选调用 netflow_run()和 sflow_run()，进行对 netflow 和 sflow 的支持。

可选调用 port_run()进行发送 CCM。

可选调用 bundle_run()处理 LACP、bonding 等杂项。

可选调用 stp_run()进行 STP 支持。

mac_learning_run()获取超时的 mac 表项，并将其删除掉。

可选调用 governor_run()进行限速处理。

之后，对控制器 ofproto 消息进行处理：

```
switch (p->state) {
  case S_OPENFLOW: //of commands, run its parser
    connmgr_run(p->connmgr, handle_openflow);
    break;

  case S_EVICT: //evict flow from over-limit tables
    connmgr_run(p->connmgr, NULL);
    ofproto_evict(p);
    if (list_is_empty(&p->pending) && hmap_is_empty(&p->deletions)) {
      p->state = S_OPENFLOW;
    }
    break;

  case S_FLUSH: //delete all flow table rules
    connmgr_run(p->connmgr, NULL);
    ofproto_flush__(p);
    if (list_is_empty(&p->pending) && hmap_is_empty(&p->deletions)) {
      connmgr_flushed(p->connmgr);
      p->state = S_OPENFLOW;
    }
    break;

  default:
    NOT_REACHED();
  }
```

根据输入参数的状态，分别执行相应的 openflow 操作。

其中，connmgr_run()函数处理与控制器的周期性交互，代码为

```
void
connmgr_run(struct connmgr *mgr,
            bool (*handle_openflow)(struct ofconn *, struct ofpbuf *ofp_msg))
{
    struct ofconn *ofconn, *next_ofconn;
    struct ofservice *ofservice;
    size_t i;

    if (handle_openflow && mgr->in_band) {
        if (!in_band_run(mgr->in_band)) {
            in_band_destroy(mgr->in_band);
            mgr->in_band = NULL;
        }
    }

    LIST_FOR_EACH_SAFE (ofconn, next_ofconn, node, &mgr->all_conns) {
        ofconn_run(ofconn, handle_openflow);
    }
    ofmonitor_run(mgr);

    /* Fail-open maintenance.  Do this after processing the ofconns since
     * fail-open checks the status of the controller rconn. */
    if (handle_openflow && mgr->fail_open) {
        fail_open_run(mgr->fail_open);
    }

    HMAP_FOR_EACH (ofservice, node, &mgr->services) {
        struct vconn *vconn;
        int retval;

        retval = pvconn_accept(ofservice->pvconn, OFP10_VERSION, &vconn);
        if (!retval) {
            struct rconn *rconn;
```

```
        char *name;

        /* Passing default value for creation of the rconn */
        rconn = rconn_create(ofservice->probe_interval, 0, ofservice->dscp);
        name = ofconn_make_name(mgr, vconn_get_name(vconn));
        rconn_connect_unreliably(rconn, vconn, name);
        free(name);

        ofconn = ofconn_create(mgr, rconn, OFCONN_SERVICE,
                               ofservice->enable_async_msgs);
        ofconn_set_rate_limit(ofconn, ofservice->rate_limit,
                              ofservice->burst_limit);
      } else if (retval != EAGAIN) {
        VLOG_WARN_RL(&rl, "accept failed (%s)", strerror(retval));
      }
    }

    for (i = 0; i < mgr->n_snoops; i++) {
      struct vconn *vconn;
      int retval;

      retval = pvconn_accept(mgr->snoops[i], OFP10_VERSION, &vconn);
      if (!retval) {
        add_snooper(mgr, vconn);
      } else if (retval != EAGAIN) {
        VLOG_WARN_RL(&rl, "accept failed (%s)", strerror(retval));
      }
    }
}
```

connmgr_run()（ofproto/connmgr.c）首先检查是否存在 in_band 的控制器，之后调用 ofconn_run()处理对 ofproto 的协议解析和行动，并进一步检查 fail-open 模式和接受新的连接。ofconn_run()代码为

```
static void
ofconn_run(struct ofconn *ofconn,
           bool (*handle_openflow)(struct ofconn *, struct ofpbuf *ofp_msg))
{
   struct connmgr *mgr = ofconn->connmgr;
   size_t i;

   for (i = 0; i < N_SCHEDULERS; i++) {
      pinsched_run(ofconn->schedulers[i], do_send_packet_in, ofconn);
   }

   rconn_run(ofconn->rconn);

   if (handle_openflow) {
      /* Limit the number of iterations to avoid starving other tasks. */
      for (i = 0; i < 50 && ofconn_may_recv(ofconn); i++) {
         struct ofpbuf *of_msg;

         of_msg = (ofconn->blocked
                  ? ofconn->blocked
                  : rconn_recv(ofconn->rconn));
         if (!of_msg) {
            break;
         }
         if (mgr->fail_open) {
            fail_open_maybe_recover(mgr->fail_open);
         }

         if (handle_openflow(ofconn, of_msg)) {
            ofpbuf_delete(of_msg);
            ofconn->blocked = NULL;
         } else {
            ofconn->blocked = of_msg;
```

```
            ofconn->retry = false;
          }
        }
      }

      if (!rconn_is_alive(ofconn->rconn)) {
        ofconn_destroy(ofconn);
      } else if (!rconn_is_connected(ofconn->rconn)) {
        ofconn_flush(ofconn);
      }
    }
```

其中，rconn_run(ofconn->rconn)负责连接到 controller，rconn_recv(ofconn->rconn)负责从 controller 收取消息。handle_openflow()最终调用 handle_openflow__()（ofproto/ofproto.c）来完成对各个 of 消息的处理，代码为

```
static enum ofperr
handle_openflow__(struct ofconn *ofconn, const struct ofpbuf *msg)
{
    const struct ofp_header *oh = msg->data;
    enum ofptype type;
    enum ofperr error;

    error = ofptype_decode(&type, oh);
    if (error) {
        return error;
    }

    switch (type) {
        /* OpenFlow requests. */
    case OFPTYPE_ECHO_REQUEST:
        return handle_echo_request(ofconn, oh);

    case OFPTYPE_FEATURES_REQUEST:
        return handle_features_request(ofconn, oh);

    case OFPTYPE_GET_CONFIG_REQUEST:
        return handle_get_config_request(ofconn, oh);

    case OFPTYPE_SET_CONFIG:
        return handle_set_config(ofconn, oh);

    case OFPTYPE_PACKET_OUT:
        return handle_packet_out(ofconn, oh);

    case OFPTYPE_PORT_MOD:
        return handle_port_mod(ofconn, oh);

    case OFPTYPE_FLOW_MOD:
```

```
    return handle_flow_mod(ofconn, oh);

  case OFPTYPE_BARRIER_REQUEST:
    return handle_barrier_request(ofconn, oh);

    /* OpenFlow replies. */
  case OFPTYPE_ECHO_REPLY:
    return 0;

    /* Nicira extension requests. */
  case OFPTYPE_ROLE_REQUEST:
    return handle_role_request(ofconn, oh);

  case OFPTYPE_FLOW_MOD_TABLE_ID:
    return handle_nxt_flow_mod_table_id(ofconn, oh);

  case OFPTYPE_SET_FLOW_FORMAT:
    return handle_nxt_set_flow_format(ofconn, oh);

  case OFPTYPE_SET_PACKET_IN_FORMAT:
    return handle_nxt_set_packet_in_format(ofconn, oh);

  case OFPTYPE_SET_CONTROLLER_ID:
    return handle_nxt_set_controller_id(ofconn, oh);

  case OFPTYPE_FLOW_AGE:
    /* Nothing to do. */
    return 0;

  case OFPTYPE_FLOW_MONITOR_CANCEL:
    return handle_flow_monitor_cancel(ofconn, oh);

  case OFPTYPE_SET_ASYNC_CONFIG:
```

```
    return handle_nxt_set_async_config(ofconn, oh);

    /* Statistics requests. */
  case OFPTYPE_DESC_STATS_REQUEST:
    return handle_desc_stats_request(ofconn, oh);

  case OFPTYPE_FLOW_STATS_REQUEST:
    return handle_flow_stats_request(ofconn, oh);

  case OFPTYPE_AGGREGATE_STATS_REQUEST:
    return handle_aggregate_stats_request(ofconn, oh);

  case OFPTYPE_TABLE_STATS_REQUEST:
    return handle_table_stats_request(ofconn, oh);

  case OFPTYPE_PORT_STATS_REQUEST:
    return handle_port_stats_request(ofconn, oh);

  case OFPTYPE_QUEUE_STATS_REQUEST:
    return handle_queue_stats_request(ofconn, oh);

  case OFPTYPE_PORT_DESC_STATS_REQUEST:
    return handle_port_desc_stats_request(ofconn, oh);

  case OFPTYPE_FLOW_MONITOR_STATS_REQUEST:
    return handle_flow_monitor_request(ofconn, oh);

  case OFPTYPE_HELLO:
  case OFPTYPE_ERROR:
  case OFPTYPE_FEATURES_REPLY:
  case OFPTYPE_GET_CONFIG_REPLY:
  case OFPTYPE_PACKET_IN:
  case OFPTYPE_FLOW_REMOVED:
```

```
    case OFPTYPE_PORT_STATUS:
    case OFPTYPE_BARRIER_REPLY:
    case OFPTYPE_DESC_STATS_REPLY:
    case OFPTYPE_FLOW_STATS_REPLY:
    case OFPTYPE_QUEUE_STATS_REPLY:
    case OFPTYPE_PORT_STATS_REPLY:
    case OFPTYPE_TABLE_STATS_REPLY:
    case OFPTYPE_AGGREGATE_STATS_REPLY:
    case OFPTYPE_PORT_DESC_STATS_REPLY:
    case OFPTYPE_ROLE_REPLY:
    case OFPTYPE_FLOW_MONITOR_PAUSED:
    case OFPTYPE_FLOW_MONITOR_RESUMED:
    case OFPTYPE_FLOW_MONITOR_STATS_REPLY:
    default:
        return OFPERR_OFPBRC_BAD_TYPE;
    }
}
```

这些处理，大部分都是在本地数据结构完成整理，并完成相应的 reply 结构之后，通过调用 ofconn_send_replies()发出去。

最后，做相应的信息统计。

以 PACKET_OUT 消息为例进行分析，调用的是 handle_packet_out()函数。该函数首先调用 ofputil_decode_packet_out()对 of 消息进行解析。之后调用 ofconn_pktbuf_retrieve()获取 payload 信息，最后利用 ofproto_class->packet_out()将网包发出。

分析 packet_out()（位于 ofproto/ofproto-dpif.c）函数，代码如下：

```
static enum ofperr
packet_out(struct ofproto *ofproto_, struct ofpbuf *packet,
        const struct flow *flow,
        const struct ofpact *ofpacts, size_t ofpacts_len)
{
    struct ofproto_dpif *ofproto = ofproto_dpif_cast(ofproto_);
    enum ofperr error;

    if (flow->in_port >= ofproto->max_ports && flow->in_port < OFPP_MAX) {
        return OFPERR_NXBRC_BAD_IN_PORT;
    }

    error = ofpacts_check(ofpacts, ofpacts_len, flow, ofproto->max_ports);
    if (!error) {
        struct odputil_keybuf keybuf;
        struct dpif_flow_stats stats;

        struct ofpbuf key;

        struct action_xlate_ctx ctx;
        uint64_t odp_actions_stub[1024 / 8];
        struct ofpbuf odp_actions;

        ofpbuf_use_stack(&key, &keybuf, sizeof keybuf);
        odp_flow_key_from_flow(&key, flow);

        dpif_flow_stats_extract(flow, packet, time_msec(), &stats);

        action_xlate_ctx_init(&ctx, ofproto, flow, flow->vlan_tci, NULL,
                        packet_get_tcp_flags(packet, flow), packet);
        ctx.resubmit_stats = &stats;

        ofpbuf_use_stub(&odp_actions,
```

```
                    odp_actions_stub, sizeof odp_actions_stub);
        xlate_actions(&ctx, ofpacts, ofpacts_len, &odp_actions);
        dpif_execute(ofproto->dpif, key.data, key.size,
                     odp_actions.data, odp_actions.size, packet);
        ofpbuf_uninit(&odp_actions);
    }
    return error;
}
```

Packet_out()函数首先检查 action 的格式是否正确。

之后调用 xlate_actions()将 ofpacts 转化为 dp 的行动格式 odp_actions。

调用 dpif_execute()函数让 dpif 执行给定的 action，构建 OVS_PACKET_CMD_EXECUTE netlink 消息并发给 datapath。

而 datapath 中将对应调用 ovs_packet_cmd_execute()函数（datapath/datapath.c）处理收到的 nlmsg。

ovs_packet_cmd_execute()的调用过程为

ovs_packet_cmd_execute()→ovs_execute_actions()→do_execute_actions()

3.2.18. unixctl_server_run(unixctl)

从 unixctl 指定的 server 中获取来自 ovs-appctl 发出的命令数据，并执行对应的命令。主要过程为

```
struct unixctl_conn *conn = xzalloc(sizeof *conn);
        list_push_back(&server->conns, &conn->node);
        conn->rpc = jsonrpc_open(stream);
```

在实现上（参考 lib/stream-*.c 文件），该通道采用函数指针，可以支持包括 tcp、unix、文件等多种通讯类型，这些类型也被 ovsd 跟 ovsdb 之间通信所使用。

目前采用的是文件（/usr/local/var/run/openvswtich/ovs-vswitchd.pid.ctl），ovs-appctl 可以通过该通道向 ovsd 发出相关的命令，包括

```
bond/disable-slave      port slave
 bond/enable-slave       port slave
 bond/hash               mac [vlan] [basis]
 bond/list
 bond/migrate            port hash slave
 bond/set-active-slave   port slave
 bond/show               [port]
 bridge/dump-flows       bridge
 bridge/reconnect        [bridge]
 cfm/set-fault           [interface] normal|false|true
 cfm/show                [interface]
 coverage/show
 exit
 fdb/flush               [bridge]
 fdb/show                bridge
 help
 lacp/show               [port]
 memory/show
 ofproto/clog
 ofproto/list
 ofproto/self-check      [bridge]
 ofproto/trace           bridge {tun_id in_port packet | odp_flow [-generate]}
 ofproto/unclog
 qos/show                interface
 stp/tcn                 [bridge]
 stress/disable
 stress/enable
 stress/list
 stress/set              option period [random | periodic]
 version
 vlog/list
 vlog/reopen
 vlog/set                {spec | PATTERN:facility:pattern}
```

3.2.19. netdev_run()

如果打开了一些 netdev，则执行对应在 netdev_classes 上定义的每个 netdev_class 实体，调用它们的 run()。主要过程为

```
SHASH_FOR_EACH(node, &netdev_classes) {
      const struct netdev_class *netdev_class = node->data;
      if (netdev_class->run) {
         netdev_class->run();
      }
   }
```

包括处理网卡注册的各个通知事件，获取网卡的最新的 mii 信息等。

netdev_class 的抽象声明在 netdev_provider.h 中，而各类 netdev_class 的具体定义在 lib/netdev-*.c，包括 bsd、linux、dummy、vport。

3.2.20. 循环等待事件处理

包括 woker、signal、memory、bridge、unixctl_server、netdev 等事件，被 poll_fd_wait()注册。

3.2.21. poll_block(void)

阻塞，直到之前被 poll_fd_wait()注册过的事件发生，或者等待时间超过 poll_timer_wait()注册的最短时间。

3.2.22. 清理工作

退出 bridge，关闭 unixctl 连接，取消对 sighup 信号的处理注册。

```
bridge_exit();
unixctl_server_destroy(unixctl);
signal_unregister(sighup);
```

3.3. 通用类型

3.3.1. 基础宏定义

首先分析下几个常见的基础宏。

CONTAINER_OF 宏：返回拥有某个给定 member 的 struct 结构的起始地址。其中，struct 为所定义的数据结构，其中有一个变量名字为 member，pointer 为指向 member 变

量的一个指针。该宏返回包含有 pointer 作为 member 变量地址的 struct 数据结构的起始地址。定义为

```
#define CONTAINER_OF(POINTER, STRUCT, MEMBER)                        \
        ((STRUCT *) (void *) ((char *) (POINTER) - offsetof (STRUCT, MEMBER)))
```

类似的，OBJECT_CONTAINING 宏：返回含有某个给定 member（ponter 指向该 member）的对象 object 的数据结构地址。

```
#define OBJECT_CONTAINING(POINTER, OBJECT, MEMBER)                    \
    ((OVS_TYPEOF(OBJECT)) (void *)                                 \
     ((char *) (POINTER) - OBJECT_OFFSETOF(OBJECT, MEMBER)))
```

ASSIGN_CONTAINER 宏：返回含有某个给定 member 的对象 object 和 1。

```
#define ASSIGN_CONTAINER(OBJECT, POINTER, MEMBER) \
    ((OBJECT) = OBJECT_CONTAINING(POINTER, OBJECT, MEMBER), 1)
```

3.3.2. 普通列表

ovs 代码中大量使用了列表的结构。列表的声明在 lib/list.h 中。列表的抽象结构用户不必关心，用户只需要维护好自己关心的节点的数据结构即可。

使用一个 list： struct list L = LIST_INITIALIZER(&L)。

正向遍历：

```
#define LIST_FOR_EACH(ITER, MEMBER, LIST)                          \
    for (ASSIGN_CONTAINER(ITER, (LIST)->next, MEMBER);             \
         &(ITER)->MEMBER != (LIST);                                \
         ASSIGN_CONTAINER(ITER, (ITER)->MEMBER.next, MEMBER))
```

逆向遍历

```
#define LIST_FOR_EACH_REVERSE(ITER, MEMBER, LIST)                  \
    for (ASSIGN_CONTAINER(ITER, (LIST)->prev, MEMBER);             \
         &(ITER)->MEMBER != (LIST);                                \
         ASSIGN_CONTAINER(ITER, (ITER)->MEMBER.prev, MEMBER))
```

安全遍历

```
#define LIST_FOR_EACH_SAFE(ITER, NEXT, MEMBER, LIST)          \
   for (ASSIGN_CONTAINER(ITER, (LIST)->next, MEMBER);         \
        (&(ITER)->MEMBER != (LIST)                           \
         ? ASSIGN_CONTAINER(NEXT, (ITER)->MEMBER.next, MEMBER) \
         : 0);                                               \
        (ITER) = (NEXT))
```

3.3.3. Hash 列表

列表的声明在 lib/hmap.h 中。

Hmap 列表，包含两个指针，指向其中含有的节点，定义为

```
/* A hash map. */
struct hmap {
   struct hmap_node **buckets; /* Must point to 'one' iff 'mask' == 0. */
   struct hmap_node *one;
   size_t mask;
   size_t n;
};
```

而 hmap_node 类型，包括一个 hash 值和一个后继指针，定义为

```
struct hmap_node {
   size_t hash;            /* Hash value. */
   struct hmap_node *next;    /* Next in linked list. */
};
```

对 hmap 的列表遍历似乎通过如下宏来实现的

```
#define HMAP_FOR_EACH(NODE, MEMBER, HMAP)                         \
   for (ASSIGN_CONTAINER(NODE, hmap_first(HMAP), MEMBER);            \
        &(NODE)->MEMBER != NULL;                                  \
        ASSIGN_CONTAINER(NODE, hmap_next(HMAP, &(NODE)->MEMBER), MEMBER))
```

其中 ASSIGN_CONTAINER 有三个参数，分别是一个输出指针 NODE，一个输入指针 HMAP 和一个成员 MEMBER。输入指针指向该成员，而输出指针获取指向包含有该成员的类型的地址。

所以遍历宏实际上，遍历了给定 hmap 变量的每个节点（在 hmap 上的成员其实是 member），并将包含该节点的数据结构 NODE 逐个返回。或者说，NODE 类型含有一个

member 成员，这些成员作为 hmap 上的节点实际链在一起。该遍历返回的是各个被逻辑链在一起的对应的 NODE 类型的变量。

3.3.4. ofproto

3.3.5. ofproto_class

3.3.6. log 消息

```
#define VLOG_FATAL(...) vlog_fatal(THIS_MODULE, __VA_ARGS__)
#define VLOG_ABORT(...) vlog_abort(THIS_MODULE, __VA_ARGS__)
#define VLOG_EMER(...) VLOG(VLL_EMER, __VA_ARGS__)
#define VLOG_ERR(...) VLOG(VLL_ERR, __VA_ARGS__)
#define VLOG_WARN(...) VLOG(VLL_WARN, __VA_ARGS__)
#define VLOG_INFO(...) VLOG(VLL_INFO, __VA_ARGS__)
#define VLOG_DBG(...) VLOG(VLL_DBG, __VA_ARGS__)
```

4. 动态过程

4.1. datapath 收到网包

在 dp_init 中通过调用 dp_register_genl()注册了对于 dp,vport,flow,packet 四种类型事件的 netlink family 和 ops。

当内核中的 openvswitch.ko 收到一个添加网桥的指令时候，即接收到 OVS_DATAPATH_FAMILY 通道的 OVS_DP_CMD_NEW 命令。该命令绑定的回调函数为 ovs_dp_cmd_new。相关实现在 datapath/datapath.c 文件中，关键代码如下：

```
static struct genl_ops dp_datapath_genl_ops[] = {
  { .cmd = OVS_DP_CMD_NEW,
    .flags = GENL_ADMIN_PERM, /* Requires CAP_NET_ADMIN privilege. */
    .policy = datapath_policy,
    .doit = ovs_dp_cmd_new
  },
  { .cmd = OVS_DP_CMD_DEL,
    .flags = GENL_ADMIN_PERM, /* Requires CAP_NET_ADMIN privilege. */
    .policy = datapath_policy,
    .doit = ovs_dp_cmd_del
  },
  { .cmd = OVS_DP_CMD_GET,
    .flags = 0,               /* OK for unprivileged users. */
    .policy = datapath_policy,
    .doit = ovs_dp_cmd_get,
    .dumpit = ovs_dp_cmd_dump
  },
  { .cmd = OVS_DP_CMD_SET,
    .flags = GENL_ADMIN_PERM, /* Requires CAP_NET_ADMIN privilege. */
    .policy = datapath_policy,
    .doit = ovs_dp_cmd_set,
  },
};
```

ovs_dp_cmd_new 函数除了初始化 dp 结构外，还调用 new_vport()函数来生成一个新的 vport。而 new_port 函数则调用 ovs_vport_add()函数，来尝试生成一个新的 vport。关键代码如下：

```
static struct vport *new_vport(const struct vport_parms *parms)
{
  struct vport *vport;

  vport = ovs_vport_add(parms);
  if (!IS_ERR(vport)) {
          struct datapath *dp = parms->dp;
          struct hlist_head *head = vport_hash_bucket(dp, vport->port_no);

          hlist_add_head_rcu(&vport->dp_hash_node, head);
          dp_ifinfo_notify(RTM_NEWLINK, vport);
  }
  return vport;
}
```

ovs_vport_add()函数会检查 vport 类型，并调用相关的 create()函数来生成 vport 结构。关键代码为

```
struct vport *ovs_vport_add(const struct vport_parms *parms)
{
  struct vport *vport;
  int err = 0;
  int i;

  ASSERT_RTNL();

  for (i = 0; i < n_vport_types; i++) {
        if (vport_ops_list[i]->type == parms->type) {
              struct hlist_head *bucket;

              vport = vport_ops_list[i]->create(parms);
              if (IS_ERR(vport)) {
                    err = PTR_ERR(vport);
                    goto out;
              }

              bucket = hash_bucket(ovs_dp_get_net(vport->dp),
                                   vport->ops->get_name(vport));
              hlist_add_head_rcu(&vport->hash_node, bucket);
              return vport;
        }
  }

  err = -EAFNOSUPPORT;

out:
  return ERR_PTR(err);
}
```

其中 vport_ops_list[]在 ovs_vport_init()的初始化过程中，被定义为与 base_vport_ops_list 相同。关键代码如下

```
int ovs_vport_init(void)
{
   int err;
   int i;

   dev_table = kzalloc(VPORT_HASH_BUCKETS * sizeof(struct hlist_head),
                       GFP_KERNEL);
   if (!dev_table) {
          err = -ENOMEM;
          goto error;
   }

   vport_ops_list = kmalloc(ARRAY_SIZE(base_vport_ops_list) *
                            sizeof(struct vport_ops *), GFP_KERNEL);
   if (!vport_ops_list) {
          err = -ENOMEM;
          goto error_dev_table;
   }

    /* create a vport_ops_list, templated from base_vport_ops_list.*/
   for (i = 0; i < ARRAY_SIZE(base_vport_ops_list); i++) {
          const struct vport_ops *new_ops = base_vport_ops_list[i]; //check each
vport_ops instance

          if (new_ops->init)
                 err = new_ops->init(); //init each vport_ops
          else
                 err = 0;

          if (!err)
                 vport_ops_list[n_vport_types++] = new_ops;
          else if (new_ops->flags & VPORT_F_REQUIRED) {
                 ovs_vport_exit();
```

```
                goto error;
        }
  }

  return 0;

error_dev_table:
  kfree(dev_table);
error:
  return err;
}
```

而 base_vport_ops_list[]变量的成员目前有 5 种，分别为

```
/* List of statically compiled vport implementations.  Don't forget to also
 * add yours to the list at the bottom of vport.h. */
static const struct vport_ops *base_vport_ops_list[] = {
  &ovs_netdev_vport_ops, //netdev instance
  &ovs_internal_vport_ops,
  &ovs_patch_vport_ops,
  &ovs_gre_vport_ops,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,26)
  &ovs_capwap_vport_ops,
#endif
};
```

因此，当 vport 定义为网络类型时，会执行 ovs_netdev_vport_ops 中定义的相关函数，包括 init、create 等，函数列表如下：

```
const struct vport_ops ovs_netdev_vport_ops = {
  .type             = OVS_VPORT_TYPE_NETDEV,
  .flags        = VPORT_F_REQUIRED,
  .init             = netdev_init,
  .exit             = netdev_exit,
  .create           = netdev_create,
  .destroy          = netdev_destroy,
  .set_addr         = ovs_netdev_set_addr,
  .get_name         = ovs_netdev_get_name,
  .get_addr         = ovs_netdev_get_addr,
  .get_kobj         = ovs_netdev_get_kobj,
  .get_dev_flags    = ovs_netdev_get_dev_flags,
  .is_running       = ovs_netdev_is_running,
  .get_operstate    = ovs_netdev_get_operstate,
  .get_ifindex      = ovs_netdev_get_ifindex,
  .get_mtu          = ovs_netdev_get_mtu,
  .send             = netdev_send,
};
```

可见，当 dp 是网络设备时（vport-netdev.c），最终由 ovs_vport_add()函数调用的是 netdev_create()函数，而 netdev_create()函数中最关键的一步是注册了收到网包时的回调函数。

```
err = netdev_rx_handler_register(netdev_vport->dev, netdev_frame_hook, vport);
```

该操作将 netdev_vport->dev 收到网包时的相关数据由, netdev_frame_hook()函数来处理。后面都是进行一些辅助处理后，依次调用各处理函数，中间从 ovs_vport_receive()回到 vport.c，从 ovs_dp_process_received_packet()回到 datapath.c，进行统一处理。

```
netdev_frame_hook()→netdev_port_receive()→ovs_vport_receive()→ovs_dp_process_received_packet()
```

在 ovs_dp_process_received_packet()（datapath/datapath.c）中进行复杂的包处理过程，进行流查表，查表后执行对应的行为。当查找失败时候，使用 ovs_dp_upcall()发送 upcall 到用户空间（ovs-vswitchd）。此后处理过程交给 ovsd 处理。

```
ovs_dp_process_received_packet()→ovs_dp_upcall()→ queue_userspace_packet()
```

4.2. ovs-vswitchd 快速处理

ovs-vswitchd 利用 bridge_run()/bridge_fast_run()（vswitchd/bridge.c）不断轮询各个 bridge，执行相应的操作。

主要通过调用 ofproto_run_fast()和 ofproto_run()来运行各个 bridge 上的 ofproto 的处理过程。其中 run_fast()函数简化了一些不必要的处理，主要处理 upcall，运行速度更快。

4.2.1. bridge_run_fast()

下面我们首先以 run_fast()函数为例进行分析。

```
void
bridge_run_fast(void)
{
    struct bridge *br;

    HMAP_FOR_EACH (br, node, &all_bridges) {
        ofproto_run_fast(br->ofproto);
    }
}
```

ofproto_run_fast()调用 struct ofproto_class {}中对应的 run_fast()。其中 struct ofproto_class（ofproto/ofproto-provider.h）是 j 个抽象类。run_fast()是个函数指针，实际上，在 bridge_run()中进行了赋值。

可能的 ofproto_class 类型在 ofproto_classes[]变量中声明。而 ofproto_classes[]变量是通过 ofproto_initialize()来进行初始化的。在 ofproto/ofproto.c 中有如下的代码

可见，ofproto_classes 在初始化 ofproto_initialize()（该初始化函数多次被调用，但仅执行一次）后仅含有一个变量，即 ofproto_dpif_class，而 ofproto_dpif_class 的定义在 ofproto/ofproto-dpif.c 中，声明了各个变量和操作函数，如下

```
static void
ofproto_initialize(void)
{
    static bool inited;

    if (!inited) {
        inited = true;
        ofproto_class_register(&ofproto_dpif_class);
    }
}
```

其中，ofproto_class_register()定义如下。

```
int
ofproto_class_register(const struct ofproto_class *new_class)
{
    size_t i;

    for (i = 0; i < n_ofproto_classes; i++) {
        if (ofproto_classes[i] == new_class) {
            return EEXIST;
        }
    }

    if (n_ofproto_classes >= allocated_ofproto_classes) {
        ofproto_classes = x2nrealloc(ofproto_classes,
                                     &allocated_ofproto_classes,
                                     sizeof *ofproto_classes);
    }
    ofproto_classes[n_ofproto_classes++] = new_class;
    return 0;
}
```

可见，ofproto_classes 在初始化 ofproto_initialize()（该初始化函数多次被调用，但仅执行一次）后仅含有一个变量，即 ofproto_dpif_class，而 ofproto_dpif_class 的定义在 ofproto/ofproto-dpif.c 中，声明了各个变量和操作函数，如下

```
const struct ofproto_class ofproto_dpif_class = {
    enumerate_types,
    enumerate_names,
    del,
    alloc,
    construct,
    destruct,
    dealloc,
    run,
    run_fast,
    wait,
    get_memory_usage,
    flush,
    get_features,
    get_tables,
    port_alloc,
    port_construct,
    port_destruct,
    port_dealloc,
    port_modified,
    port_reconfigured,
    port_query_by_name,
    port_add,
    port_del,
    port_get_stats,
    port_dump_start,
    port_dump_next,
    port_dump_done,
    port_poll,
    port_poll_wait,
    port_is_lacp_current,
    NULL,                   /* rule_choose_table */
    rule_alloc,
```

```
    rule_construct,
    rule_destruct,
    rule_dealloc,
    rule_get_stats,
    rule_execute,
    rule_modify_actions,
    set_frag_handling,
    packet_out,
    set_netflow,
    get_netflow_ids,
    set_sflow,
    set_cfm,
    get_cfm_fault,
    get_cfm_opup,
    get_cfm_remote_mpids,
    get_cfm_health,
    set_stp,
    get_stp_status,
    set_stp_port,
    get_stp_port_status,
    set_queues,
    bundle_set,
    bundle_remove,
    mirror_set,
    mirror_get_stats,
    set_flood_vlans,
    is_mirror_output_bundle,
    forward_bpdu_changed,
    set_mac_idle_time,
    set_realdev,
};
```

Ofproto_class 的初始化多次被调用，一个可能的流程如下：

```
bridge_run()→ bridge_reconfigure()→ bridge_update_ofprotos()→
ofproto_create()→ofproto_initialize()
```

除了这个过程以外，在 ofproto_class_find__()中也都调用了 ofproto_initialize()。

因此，ofproto_class 中的成员的函数指针实际上指向了 ofproto_dpif_class 中的各个函数。

4.2.2. run_fast()

我们来看 ofproto_dpif_class 中的 run_fast(struct ofproto *ofproto)（ofproto/ofproto-dpif.c）的代码。

```
static int
run_fast(struct ofproto *ofproto_)
{
    struct ofproto_dpif *ofproto = ofproto_dpif_cast(ofproto_);
    struct ofport_dpif *ofport;
    unsigned int work;

    HMAP_FOR_EACH (ofport, up.hmap_node, &ofproto->up.ports) {
        port_run_fast(ofport);
    }

    /* Handle one or more batches of upcalls, until there's nothing left to do
     * or until we do a fixed total amount of work.
     *
     * We do work in batches because it can be much cheaper to set up a number
     * of flows and fire off their patches all at once.  We do multiple batches
     * because in some cases handling a packet can cause another packet to be
     * queued almost immediately as part of the return flow.  Both
     * optimizations can make major improvements on some benchmarks and
     * presumably for real traffic as well. */
    work = 0;
    while (work < FLOW_MISS_MAX_BATCH) {
        int retval = handle_upcalls(ofproto, FLOW_MISS_MAX_BATCH - work);
        if (retval <= 0) {
            return -retval;
        }
        work += retval;
    }
    return 0;
}
```

实际上主要是做了对 handle_upcalls()的调用。这也可以理解，因为 ovs-vswitchd 在各个 bridge 上很重要的一个操作就是要监听和处理来自各个 bridge 上的请求。

4.2.3. handle_upcalls()

代码如下：

```
static int
handle_upcalls(struct ofproto_dpif *ofproto, unsigned int max_batch)
{
    struct dpif_upcall misses[FLOW_MISS_MAX_BATCH];
    struct ofpbuf miss_bufs[FLOW_MISS_MAX_BATCH];
    uint64_t miss_buf_stubs[FLOW_MISS_MAX_BATCH][4096 / 8];
    int n_processed;
    int n_misses;
    int i;

    assert(max_batch <= FLOW_MISS_MAX_BATCH);

    n_misses = 0;
    for (n_processed = 0; n_processed < max_batch; n_processed++) {
        struct dpif_upcall *upcall = &misses[n_misses];
        struct ofpbuf *buf = &miss_bufs[n_misses];
        int error;

        ofpbuf_use_stub(buf, miss_buf_stubs[n_misses],
                        sizeof miss_buf_stubs[n_misses]);
        error = dpif_recv(ofproto->dpif, upcall, buf);
        if (error) {
            ofpbuf_uninit(buf);
            break;
        }

        switch (classify_upcall(upcall)) {
        case MISS_UPCALL:
            /* Handle it later. */
            n_misses++;
            break;

        case SFLOW_UPCALL:
```

```
        if (ofproto->sflow) {
            handle_sflow_upcall(ofproto, upcall);
        }
        ofpbuf_uninit(buf);
        break;

    case BAD_UPCALL:
        ofpbuf_uninit(buf);
        break;
    }
  }

  /* Handle deferred MISS_UPCALL processing. */
  handle_miss_upcalls(ofproto, misses, n_misses);
  for (i = 0; i < n_misses; i++) {
    ofpbuf_uninit(&miss_bufs[i]);
  }

  return n_processed;
}
```

在这部分代码中，完成对相关的 upcall 的处理。包括 datapath 找不到流表时的 MISS_UPCALL 和 SFLOW 相关流量处理等。对流表 MISS_UPCALL 部分的处理主要在 handle_miss_upcalls()（ofproto/ofproto-dpif.c）。其中，handle_miss_upcalls()的主要过程为执行 handle_flow_miss()和 dpif_operate()。handle_flow_miss()负责查找出对 upcall 的对应行动，后者根据行动执行操作。

```
HMAP_FOR_EACH (miss, hmap_node, &todo) {
        handle_flow_miss(ofproto, miss, flow_miss_ops, &n_ops);
    }
    assert(n_ops <= ARRAY_SIZE(flow_miss_ops));

    /* Execute batch. */
    for (i = 0; i < n_ops; i++) {
        dpif_ops[i] = &flow_miss_ops[i].dpif_op;
    }
    dpif_operate(ofproto->dpif, dpif_ops, n_ops);
```

handle_flow_miss()

位于 ofproto/ofproto-dpif.c，主要过程如下

```
static void
handle_flow_miss(struct ofproto_dpif *ofproto, struct flow_miss *miss,
            struct flow_miss_op *ops, size_t *n_ops)
{
   struct facet *facet;
   uint32_t hash;

   /* The caller must ensure that miss->hmap_node.hash contains
    * flow_hash(miss->flow, 0). */
   hash = miss->hmap_node.hash;

   facet = facet_lookup_valid(ofproto, &miss->flow, hash);
   if (!facet) {
      struct rule_dpif *rule = rule_dpif_lookup(ofproto, &miss->flow);

      if (!flow_miss_should_make_facet(ofproto, miss, hash)) {
         handle_flow_miss_without_facet(miss, rule, ops, n_ops);
         return;
      }

      facet = facet_create(rule, &miss->flow, hash);
   }
   handle_flow_miss_with_facet(miss, facet, ops, n_ops);
}
```

该过程主要包括两部分，首先是在本地通过 facet_lookup_valid()函数查找流表，看是否有与 flow 精确匹配的规则。

如果不存在 facet，则通过 rule_dpif_lookup()函数来查找匹配的规则，并利用 flow_miss_should_make_facet ()测试是否值得在 ovsd 的 ofproto 中添加相应规则并写到 datapath 中（多数情况下）。如果为是，handle_flow_miss_without_facet()将结果 rule 写入到 ops 或者利用 facet_create()在 ofproto 内添加新的 facet。

如果存在 facet，则利用 handle_flow_miss_with_facet()更新 ops。其中，handle_flow_miss_with_facet()调用 handle_flow_miss_common()进行状态测试：如果在 fail mod，则发送 miss 消息（ofproto/ofproto-dpif.c 文件中，调用

send_packet_in_miss()→connmgr_send_packet_in()）给 controller。之后检查是否创建 slow flow 的标记等。

以 rule_dpif_lookup()为例，该函数进一步的调用 rule_dpif_lookup__()函数。其代码为

```
static struct rule_dpif *
rule_dpif_lookup__(struct ofproto_dpif *ofproto, const struct flow *flow,
               uint8_t table_id)
{
    struct cls_rule *cls_rule;
    struct classifier *cls;

    if (table_id >= N_TABLES) {
        return NULL;
    }

    cls = &ofproto->up.tables[table_id].cls;
    if (flow->nw_frag & FLOW_NW_FRAG_ANY
        && ofproto->up.frag_handling == OFPC_FRAG_NORMAL) {
        /* For OFPC_NORMAL frag_handling, we must pretend that transport ports
         * are unavailable. */
        struct flow ofpc_normal_flow = *flow;
        ofpc_normal_flow.tp_src = htons(0);
        ofpc_normal_flow.tp_dst = htons(0);
        cls_rule = classifier_lookup(cls, &ofpc_normal_flow);
    } else {
        cls_rule = classifier_lookup(cls, flow);
    }
    return rule_dpif_cast(rule_from_cls_rule(cls_rule));
}
```

其中，classifier_lookup()通过对 ofproto 中存储的规则表进行查找，找到最高优先级的匹配规则，并返回该规则。

dpif_operate()

dpif_operate(ofproto->dpif, dpif_ops, n_ops)主要根据 handle_flow_miss()后确定的行动，执行相关的操作。

主要代码如下：

```
/* Execute batch. */
    for (i = 0; i < n_ops; i++) {
        dpif_ops[i] = &flow_miss_ops[i].dpif_op;
    }
    dpif_operate(ofproto->dpif, dpif_ops, n_ops);
```

其中 dpif_operate()首先判断具体的 dpif_class 中是否存在 operate()函数，如果有则调用执行。否则就根据 op 的类型调用具体的 dpif_class 中的 flow_put()、flow_del()、或 execute()函数。

operate()函数存在

先分析存在 operate()函数的情况。

```
dpif->dpif_class->operate(dpif, ops, n_ops);
```

operate()需要根据具体的类型来定。dpif_class 类型仍然在 dpif.c（lib/dpif.c）中通过 base_dpif_classes[]来声明。

```
static const struct dpif_class *base_dpif_classes[] = {
#ifdef HAVE_NETLINK
    &dpif_linux_class,
#endif
    &dpif_netdev_class,
};
```

其中 dpif_linux_class 通过 netlink 跟本地的 datapath 通信，而 dpif_netdev_class 则意味着通过网络协议跟远程的 datapath 通信。此处分析以常见的 dpif-linux 为例，其 operate()实际为 dpif_linux_operate()（lib/dpif-linux.c）。

dpif_linux_operate()实际上主要就是调用了 dpif_linux_operate__()，在 dpif_linux_operate__()中，首先利用传入的 dpif_op **ops 对不同类型（PUT、DEL、EXECUTE）的行动分别创建相应的多个 aux->request 消息，之后调用 nl_sock_transact_multiple()函数（lib/netlink-socket.c）来发出 request，并收回 reply。代码为

```
    nl_sock_transact_multiple(genl_sock, txnsp, n_ops);
```

注意，txnsp 中同时包括发出的 request 和收回的 reply。

之后，检查 reply 函数，更新相关的统计变量。

operate()函数不存在

如果对于某个具体的 dpif_class，并没有提供 operate()，则需要分别处理不同类型的行为，代码为

```
for (i = 0; i < n_ops; i++) {
        struct dpif_op *op = ops[i];

        switch (op->type) {
        case DPIF_OP_FLOW_PUT:
            op->error = dpif_flow_put__(dpif, &op->u.flow_put);
            break;

        case DPIF_OP_FLOW_DEL:
            op->error = dpif_flow_del__(dpif, &op->u.flow_del);
            break;

        case DPIF_OP_EXECUTE:
            op->error = dpif_execute__(dpif, &op->u.execute);
            break;

        default:
            NOT_REACHED();
        }
```

可以处理 DPIF_OP_FLOW_PUT、DPIF_OP_FLOW_DEL 和 DPIF_OP_EXECUTE 三种类型的情况（均在 lib/dpif.h 中定义）。

```
enum dpif_op_type {
    DPIF_OP_FLOW_PUT = 1,
    DPIF_OP_FLOW_DEL,
    DPIF_OP_EXECUTE,
};
```

值得注意的是第三种情况，DPIF_OP_EXECUTE，需要将执行命令发回 datapath。dpif_execute__()调用了 dpif 结构中的 dpif_class 抽象类型中的 execute()函数。

仍然以 dpif-linux（lib/dpif-linux.c）为例，定义为

```
const struct dpif_class dpif_linux_class = {
   "system",
   dpif_linux_enumerate,
   dpif_linux_open,
   dpif_linux_close,
   dpif_linux_destroy,
   dpif_linux_run,
   dpif_linux_wait,
   dpif_linux_get_stats,
   dpif_linux_port_add,
   dpif_linux_port_del,
   dpif_linux_port_query_by_number,
   dpif_linux_port_query_by_name,
   dpif_linux_get_max_ports,
   dpif_linux_port_get_pid,
   dpif_linux_port_dump_start,
   dpif_linux_port_dump_next,
   dpif_linux_port_dump_done,
   dpif_linux_port_poll,
   dpif_linux_port_poll_wait,
   dpif_linux_flow_get,
   dpif_linux_flow_put,
   dpif_linux_flow_del,
   dpif_linux_flow_flush,
   dpif_linux_flow_dump_start,
   dpif_linux_flow_dump_next,
   dpif_linux_flow_dump_done,
   dpif_linux_execute,
   dpif_linux_operate,
   dpif_linux_recv_set,
   dpif_linux_queue_to_priority,
   dpif_linux_recv,
   dpif_linux_recv_wait,
```

```
    dpif_linux_recv_purge,
};
```

所以，执行函数的为 dpif_linux_execute()（lib/dpif-linux.c），该函数首先调用的是 dpif_linux_execute__()函数。

```
static int
dpif_linux_execute__(int dp_ifindex, const struct dpif_execute *execute)
{
    uint64_t request_stub[1024 / 8];
    struct ofpbuf request;
    int error;

    ofpbuf_use_stub(&request, request_stub, sizeof request_stub);
    dpif_linux_encode_execute(dp_ifindex, execute, &request);
    error = nl_sock_transact(genl_sock, &request, NULL);
    ofpbuf_uninit(&request);

    return error;
}
```

该函数创建一个 OVS_PACKET_CMD_EXECUTE 类型的 nlmsg，并利用 nl_sock_transact()将它发出给 datapath。

4.3. ovs-vswitchd 完整处理

主要通过调用 bridge_run()来完成完整的配置和处理。

该函数执行完整的 bridge 的操作，包括对 of 命令的操作和网桥的维护，数据库信息同步、维护到 controller 的连接等。之后调用 ofproto_run()处理相关的 ofproto 消息（参考 3.2.17）。

最后判断是否到了周期 log 的时间，进行 log 记录（但默认的周期为 LLONG_MAX），并刷新各个连接和状态信息等。

5. datapath 和 ovsd 的通信机制

datapath 运行在内核态，ovsd 运行在用户态，两者通过 netlink 通信。

何为 netlink？netlink 是一种灵活和强大的进程间通信机制（socket），甚至可以沟通用户态和内核态。并且，netlink 是全双工的。作为 socket，netlink 的地址族是 AF_NETLINK（TCP/IP socket 的地址族是 AF_INET）。官方的介绍材料可以参考 http://www.infradead.org/~tgr/libnl/。

目前有大量的通信场景应用了 netlink，这些特定扩展和设计的 netlink 通信 bus，被定义为 family。比如 NETLINK_ROUTE、NETLINK_FIREWALL、NETLINK_ARPD 等。

因为大量的专用 family 会占用了 family id，而 family id 数量自身有限（kernel 允许 32 个）；同时为了方便用户扩展使用，一个通用的 netlink family 被定义出来，这就是 generic netlink family。

5.1. generic netlink 简介

generic netlink 的架构如下图所示。

```
+---------------------+     +---------------------+
  | (3) application "A" |     | (3) application "B" |
  +------+--------------+     +--------------+------+
         |                                |
         \                               /
          \                             /
           |                           |
   +-------+-------------------------------+-------+
   |       :                           :       |  user-space
=====+       :   (5)  kernel socket API     :       +================
   |       :                           :       |  kernel-space
   +--------+-----------------------------+-------+
            |                          |
      +-----+-----------------------------+----+
      |        (1)  Netlink subsystem           |
      +---------------------+--------------------+
```

```
                    |
     +--------------------+-------------------+
     |      (2) Generic Netlink bus           |
     +--+-------------------------+-------+----+
        |                         |       |
  +-------+---------+             |       |
  |  (4) controller |            /         \
  +----------------+            /           \
                              |           |
            +-----------------+--+     +--+-----------------+
            | (3) kernel user "X" |     | (3) kernel user "Y" |
            +--------------------+     +--------------------+
```

而 generic netlink 的消息结构如下所示。

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Netlink message header (nlmsghdr)              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Generic Netlink message header (genlmsghdr)         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Optional user specific message header             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Optional Generic Netlink message payload            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

要使用 generic netlink，需要熟悉的数据结构包括 genl_family、genl_ops 等。

5.1.1. genl_family

该结构体负责新的 socket bus，声明为

```
struct genl_family
{
    unsigned int        id;
    unsigned int        hdrsize;
    char                name[GENL_NAMSIZ];
    unsigned int        version;
    unsigned int        maxattr;
    struct nlattr **    attrbuf;
    struct list_head    ops_list;
    struct list_head    family_list;
};
```

其中，各个成员的含义为

- id，新 family 的信道号码，一般复制为 GENL_ID_GENERATE（值为 0），让 controller 来自动赋值。
- hdrsize，如果新 family 指定特殊的包头，则此处为该包头长度，否则一般设置为 0。
- name，family 的名字，该字符串必须唯一，以便 controller 查找信道号。
- version，版本。
- maxattr，generic netlink family 的属性（attribute）个数上限。
- attrbuf，私有数据，不能修改。
- ops_list，私有数据，不能修改。
- family_list，私有数据，不能修改。

5.1.2. genl_ops

该结构负责 family 的操作，对每一个 family，最多可以定义 255 个操作。

```
struct genl_ops
{
    u8                  cmd;
    unsigned int        flags;
    struct nla_policy   *policy;
    int                 (*doit)(struct sk_buff *skb,
                             struct genl_info *info);
    int                 (*dumpit)(struct sk_buff *skb,
                              struct netlink_callback *cb);
    struct list_head    ops_list;
};
```

其中，各个成员的含义为

- cmd，在定义 family 中值唯一，用于索引 operation。
- flags，用于指定该操作的一些属性，例如 GENL_ADMIN_PERM 表示执行该操作需要 CAP_NET_ADMIN 权限。
- policy，用于检查在 operation request 消息中的属性的格式。
- doit()，即该操作的回调函数。第一个参数为触发该操作的消息 buffer，第二个为 request 消息一些可能的附加信息。
- dumpit()，也是一个回调函数。如果收到的消息中含有 NLM_F_DUMP 标志位，则该函数被调用。其中第一个参数为预分配的 buffer，函数在 buffer 中写入回复消息，并返回消息长度。该回复消息会自动返回请求端。返回值大于 0，说明还有剩余数据需要回复，该函数被重新调用，并分配新的 buffer；当返回值为 0，表明发送完毕。第二个参数可以用来保存状态信息。
- ops_list，私有数据，不能进行修改。

5.1.3. genl_info

用于描述消息的一些相关信息，被用在 genl_ops 中 doit()回调函数的第二个参数。该结构定义为

```
struct genl_info
  {
    u32              snd_seq;
    u32              snd_pid;
    struct nlmsghdr *     nlhdr;
    struct genlmsghdr *    genlhdr;
    void *              userhdr;
    struct nlattr **      attrs;
  };
```

其中，各个成员的含义为

- snd_seq，request 消息的序列号。
- snd_pid，发送端的 pid（port id），用于标识发送端，并非系统中的进程号。
- nlhdr，指针指向 request 消息的 netlink 消息头。
- genlhdr，指针指向 request 消息的 generic netlink 消息头。
- userhdr，如果某个 family 有自定义的头，则该指针指向自定义头。
- attrs，request 消息中可能带有的属性信息。

5.1.4. nla_policy

负责检查 request 消息中属性是否符合标准。定义为

```
struct nla_policy
  {
    u16        type;
    u16        len;
  };
```

其中

type 为要检查属性的类型，可以为

- NLA_UNSPEC：未定义
- NLA_U8：8 位无符号整数
- NLA_U16：16 位无符号整数

- NLA_U32：32 位无符号整数
- NLA_U64：64 位无符号整数
- NLA_FLAG：bool 类型，用作标志
- NLA_MSECS：64 位的时间值，单位是毫秒
- NLA_STRING：变长字符串
- NLA_NUL_STRING：以 null 结尾的变长字符串
- NLA_NESTED：属性流（stream）。

len 为属性的最大长度，如果定义为 0，则不进行检查。

5.2. generic netlink 简单示例

首先，给出一个简单的示例。

5.2.1. 定义 family

```
/* attributes */
 enum {
     DOC_EXMPL_A_UNSPEC,
     DOC_EXMPL_A_MSG,
     __DOC_EXMPL_A_MAX,
 };
 #define DOC_EXMPL_A_MAX (__DOC_EXMPL_A_MAX - 1)

 /* attribute policy */
 static struct nla_policy doc_exmpl_genl_policy[DOC_EXMPL_A_MAX + 1] = {
     [DOC_EXMPL_A_MSG] = { .type = NLA_NUL_STRING },
 };

 /* family definition */
 static struct genl_family doc_exmpl_gnl_family = {
     .id = GENL_ID_GENERATE,
     .hdrsize = 0,
     .name = "DOC_EXMPL",
     .version = 1,
     .maxattr = DOC_EXMPL_A_MAX,
 };
```

5.2.2. 定义操作

```
/* handler */
 int doc_exmpl_echo(struct sk_buff *skb, struct genl_info *info)
 {
     /* message handling code goes here; return 0 on success, negative
      * values on failure */
 }

 /* commands */
 enum {
```

```
    DOC_EXMPL_C_UNSPEC,
    DOC_EXMPL_C_ECHO,
    __DOC_EXMPL_C_MAX,
};
#define DOC_EXMPL_C_MAX (__DOC_EXMPL_C_MAX - 1)

/* operation definition */
struct genl_ops doc_exmpl_gnl_ops_echo = {
    .cmd = DOC_EXMPL_C_ECHO,
    .flags = 0,
    .policy = doc_exmpl_genl_policy,
    .doit = doc_exmpl_echo,
    .dumpit = NULL,
};
```

5.2.3. 注册 family 到 generic netlink 机制

```
int rc;

 rc = genl_register_family(&doc_exmpl_gnl_family);
 if (rc != 0)
   goto failure;
```

5.2.4. 将操作注册到 family

```
int rc;

 rc = genl_register_ops(&doc_exmpl_gnl_family, &doc_exmpl_gnl_ops_echo);
 if (rc != 0)
   goto failure;
```

5.2.5. 内核端代码

包含头文件

```
#include/net/netlink.h
 #include/net/genetlink.h
```

发送消息

包括三步：分配存储 buffer，创建消息，发送消息。

分配 buffer 可以用 nlsmsg_new()函数。

```
struct sk_buff *skb;

 skb = genlmsg_new(NLMSG_GOODSIZE, GFP_KERNEL);
 if (skb == NULL)
     goto failure;
```

当不知道消息的大小的时候，可以用 NLMSG_GOODSIZE。genlmsg_new()会自动添加 netlink 消息头和 generic netlink 消息头。

创建消息主要需要填充消息的 payload。

```
int rc;
 void *msg_head;

 /* create the message headers */
 msg_head = genlmsg_put(skb, pid, seq, type, 0, flags, DOC_EXMPL_C_ECHO, 1);
 if (msg_head == NULL) {
     rc = -ENOMEM;
     goto failure;
 }
 /* add a DOC_EXMPL_A_MSG attribute */
 rc = nla_put_string(skb, DOC_EXMPL_A_MSG, "Generic Netlink Rocks");
 if (rc != 0)
     goto failure;
 /* finalize the message */
 genlmsg_end(skb, msg_head);
```

其中，genlmsg_put()函数利用给定的数值创建消息头。

nla_put_string()函数添加一个字符串属性到消息最后。

genlmsg_end()函数则在添加完 payload 后更新消息头。

最后一步，是发出消息，代码为

```
int rc;


  rc = genlmsg_unicast(skb, pid);
  if (rc != 0)
      goto failure;
```

5.3. datapath 使用 generic netlink

在 dp_init()函数（datapath.c）中，调用 dp_register_genl()完成对四种类型的 family 以及相应操作的注册，包括 datapath、vport、flow 和 packet。前三种 family，都对应四种操作都包括 NEW、DEL、GET、SET，而 packet 的操作仅为 EXECUTE。

这些 family 和操作的定义均在 datapath.c 中。

以 flow family 为例。代码为

```
static const struct nla_policy flow_policy[OVS_FLOW_ATTR_MAX + 1] = {
   [OVS_FLOW_ATTR_KEY] = { .type = NLA_NESTED },
   [OVS_FLOW_ATTR_ACTIONS] = { .type = NLA_NESTED },
   [OVS_FLOW_ATTR_CLEAR] = { .type = NLA_FLAG },
};


static struct genl_family dp_flow_genl_family = {
   .id = GENL_ID_GENERATE,
   .hdrsize = sizeof(struct ovs_header),
   .name = OVS_FLOW_FAMILY,
   .version = OVS_FLOW_VERSION,
   .maxattr = OVS_FLOW_ATTR_MAX,
    SET_NETNSOK
};
```

而绑定的 ops 的定义为

```
static struct genl_ops dp_flow_genl_ops[] = {
  { .cmd = OVS_FLOW_CMD_NEW,
    .flags = GENL_ADMIN_PERM, /* Requires CAP_NET_ADMIN privilege. */
    .policy = flow_policy,
    .doit = ovs_flow_cmd_new_or_set
  },
  { .cmd = OVS_FLOW_CMD_DEL,
    .flags = GENL_ADMIN_PERM, /* Requires CAP_NET_ADMIN privilege. */
    .policy = flow_policy,
    .doit = ovs_flow_cmd_del
  },
  { .cmd = OVS_FLOW_CMD_GET,
    .flags = 0,               /* OK for unprivileged users. */
    .policy = flow_policy,
    .doit = ovs_flow_cmd_get,
    .dumpit = ovs_flow_cmd_dump
  },
  { .cmd = OVS_FLOW_CMD_SET,
    .flags = GENL_ADMIN_PERM, /* Requires CAP_NET_ADMIN privilege. */
    .policy = flow_policy,
    .doit = ovs_flow_cmd_new_or_set,
  },
};
```

可见，dp 定义的 nlmsg 类型除了 genl 头和 nl 头之外，还有自定义的 ovs_header。

5.4. ovsd 使用 netlink

ovsd 对于 netlink 的实现，主要在 lib/netlink-socket.c 文件中。而对这些 netlink 操作的调用，主要在 lib/dpif-linux.c 文件（以 dpif_linux_class 为例）中对于各个行为的处理，各种可能的消息类型在 datapath 模块中事先进行了内核注册。

datapath 中对 netlink family 类型进行了注册，ovsd 在使用这些 netlink family 之前需要获取它们的信息，这一过程主要在 lib/dpif-linux.c 文件（以 dpif_linux_class 为例），dpif_linux_init()函数。代码为

```
static int
dpif_linux_init(void)
{
    static int error = -1;

    if (error < 0) {
        unsigned int ovs_vport_mcgroup;

        error = nl_lookup_genl_family(OVS_DATAPATH_FAMILY,
                                      &ovs_datapath_family);
        if (error) {
            VLOG_ERR("Generic Netlink family '%s' does not exist. "
                     "The Open vSwitch kernel module is probably not loaded.",
                     OVS_DATAPATH_FAMILY);
        }
        if (!error) {
            error = nl_lookup_genl_family(OVS_VPORT_FAMILY, &ovs_vport_family);
        }
        if (!error) {
            error = nl_lookup_genl_family(OVS_FLOW_FAMILY, &ovs_flow_family);
        }
        if (!error) {
            error = nl_lookup_genl_family(OVS_PACKET_FAMILY,
                                          &ovs_packet_family);
        }
        if (!error) {
            error = nl_sock_create(NETLINK_GENERIC, &genl_sock);
        }
        if (!error) {
            error = nl_lookup_genl_mcgroup(OVS_VPORT_FAMILY, OVS_VPORT_MCGROUP,
                                           &ovs_vport_mcgroup,
                                           OVS_VPORT_MCGROUP_FALLBACK_ID);
        }
```

```
        if (!error) {
            static struct dpif_linux_vport vport;
            nln = nln_create(NETLINK_GENERIC, ovs_vport_mcgroup,
                             dpif_linux_nln_parse, &vport);
        }
    }

    return error;
}
```

完成这些查找后，ovsd 即可利用 dpif 中的 api，通过发出这些 netlink 消息给 datapath 实现对 datapath 的操作。

相关的中间层 API 定义主要在 dpif_class（位于 lib/dpif-provider.h）的抽象类型中。

代码为

```
struct dpif_class {
   /* Type of dpif in this class, e.g. "system", "netdev", etc.
    *
    * One of the providers should supply a "system" type, since this is
    * the type assumed if no type is specified when opening a dpif. */
   const char *type;

   /* Enumerates the names of all known created datapaths, if possible, into
    * 'all_dps'.  The caller has already initialized 'all_dps' and other dpif
    * classes might already have added names to it.
    *
    * This is used by the vswitch at startup, so that it can delete any
    * datapaths that are not configured.
    *
    * Some kinds of datapaths might not be practically enumerable, in which
    * case this function may be a null pointer. */
   int (*enumerate)(struct sset *all_dps);

   /* Attempts to open an existing dpif called 'name', if 'create' is false,
    * or to open an existing dpif or create a new one, if 'create' is true.
    *
    * 'dpif_class' is the class of dpif to open.
    *
    * If successful, stores a pointer to the new dpif in '*dpifp', which must
    * have class 'dpif_class'.  On failure there are no requirements on what
    * is stored in '*dpifp'. */
   int (*open)(const struct dpif_class *dpif_class,
             const char *name, bool create, struct dpif **dpifp);

   /* Closes 'dpif' and frees associated memory. */
   void (*close)(struct dpif *dpif);

   /* Attempts to destroy the dpif underlying 'dpif'.
```

```
 *
 * If successful, 'dpif' will not be used again except as an argument for
 * the 'close' member function. */
int (*destroy)(struct dpif *dpif);

/* Performs periodic work needed by 'dpif', if any is necessary. */
void (*run)(struct dpif *dpif);

/* Arranges for poll_block() to wake up if the "run" member function needs
 * to be called for 'dpif'. */
void (*wait)(struct dpif *dpif);

/* Retrieves statistics for 'dpif' into 'stats'. */
int (*get_stats)(const struct dpif *dpif, struct dpif_dp_stats *stats);

/* Adds 'netdev' as a new port in 'dpif'.  If '*port_no' is not
 * UINT16_MAX, attempts to use that as the port's port number.
 *
 * If port is successfully added, sets '*port_no' to the new port's
 * port number.  Returns EBUSY if caller attempted to choose a port
 * number, and it was in use. */
int (*port_add)(struct dpif *dpif, struct netdev *netdev,
            uint16_t *port_no);

/* Removes port numbered 'port_no' from 'dpif'. */
int (*port_del)(struct dpif *dpif, uint16_t port_no);

/* Queries 'dpif' for a port with the given 'port_no' or 'devname'.  Stores
 * information about the port into '*port' if successful.
 *
 * The caller takes ownership of data in 'port' and must free it with
 * dpif_port_destroy() when it is no longer needed. */
int (*port_query_by_number)(const struct dpif *dpif, uint16_t port_no,
```

```
                   struct dpif_port *port);
int (*port_query_by_name)(const struct dpif *dpif, const char *devname,
                  struct dpif_port *port);

/* Returns one greater than the largest port number accepted in flow
 * actions. */
int (*get_max_ports)(const struct dpif *dpif);

/* Returns the Netlink PID value to supply in OVS_ACTION_ATTR_USERSPACE
 * actions as the OVS_USERSPACE_ATTR_PID attribute's value, for use in
 * flows whose packets arrived on port 'port_no'.
 *
 * A 'port_no' of UINT16_MAX should be treated as a special case.  The
 * implementation should return a reserved PID, not allocated to any port,
 * that the client may use for special purposes.
 *
 * The return value only needs to be meaningful when DPIF_UC_ACTION has
 * been enabled in the 'dpif''s listen mask, and it is allowed to change
 * when DPIF_UC_ACTION is disabled and then re-enabled.
 *
 * A dpif provider that doesn't have meaningful Netlink PIDs can use NULL
 * for this function.  This is equivalent to always returning 0. */
uint32_t (*port_get_pid)(const struct dpif *dpif, uint16_t port_no);

/* Attempts to begin dumping the ports in a dpif.  On success, returns 0
 * and initializes '*statep' with any data needed for iteration.  On
 * failure, returns a positive errno value. */
int (*port_dump_start)(const struct dpif *dpif, void **statep);

/* Attempts to retrieve another port from 'dpif' for 'state', which was
 * initialized by a successful call to the 'port_dump_start' function for
 * 'dpif'.  On success, stores a new dpif_port into 'port' and returns 0.
 * Returns EOF if the end of the port table has been reached, or a positive
```

```
 * errno value on error.  This function will not be called again once it
 * returns nonzero once for a given iteration (but the 'port_dump_done'
 * function will be called afterward).
 *
 * The dpif provider retains ownership of the data stored in 'port'.  It
 * must remain valid until at least the next call to 'port_dump_next' or
 * 'port_dump_done' for 'state'. */
int (*port_dump_next)(const struct dpif *dpif, void *state,
                      struct dpif_port *port);

/* Releases resources from 'dpif' for 'state', which was initialized by a
 * successful call to the 'port_dump_start' function for 'dpif'.  */
int (*port_dump_done)(const struct dpif *dpif, void *state);

/* Polls for changes in the set of ports in 'dpif'.  If the set of ports in
 * 'dpif' has changed, then this function should do one of the
 * following:
 *
 * - Preferably: store the name of the device that was added to or deleted
 *   from 'dpif' in '*devnamep' and return 0.  The caller is responsible
 *   for freeing '*devnamep' (with free()) when it no longer needs it.
 *
 * - Alternatively: return ENOBUFS, without indicating the device that was
 *   added or deleted.
 *
 * Occasional 'false positives', in which the function returns 0 while
 * indicating a device that was not actually added or deleted or returns
 * ENOBUFS without any change, are acceptable.
 *
 * If the set of ports in 'dpif' has not changed, returns EAGAIN.  May also
 * return other positive errno values to indicate that something has gone
 * wrong. */
int (*port_poll)(const struct dpif *dpif, char **devnamep);
```

```
/* Arranges for the poll loop to wake up when 'port_poll' will return a
 * value other than EAGAIN. */
void (*port_poll_wait)(const struct dpif *dpif);

/* Queries 'dpif' for a flow entry.  The flow is specified by the Netlink
 * attributes with types OVS_KEY_ATTR_* in the 'key_len' bytes starting at
 * 'key'.
 *
 * Returns 0 if successful.  If no flow matches, returns ENOENT.  On other
 * failure, returns a positive errno value.
 *
 * If 'actionsp' is nonnull, then on success '*actionsp' must be set to an
 * ofpbuf owned by the caller that contains the Netlink attributes for the
 * flow's actions.  The caller must free the ofpbuf (with ofpbuf_delete())
 * when it is no longer needed.
 *
 * If 'stats' is nonnull, then on success it must be updated with the
 * flow's statistics. */
int (*flow_get)(const struct dpif *dpif,
            const struct nlattr *key, size_t key_len,
            struct ofpbuf **actionsp, struct dpif_flow_stats *stats);

/* Adds or modifies a flow in 'dpif'.  The flow is specified by the Netlink
 * attributes with types OVS_KEY_ATTR_* in the 'put->key_len' bytes
 * starting at 'put->key'.  The associated actions are specified by the
 * Netlink attributes with types OVS_ACTION_ATTR_* in the
 * 'put->actions_len' bytes starting at 'put->actions'.
 *
 * - If the flow's key does not exist in 'dpif', then the flow will be
 *   added if 'put->flags' includes DPIF_FP_CREATE.  Otherwise the
 *   operation will fail with ENOENT.
 *
```

```
 *   If the operation succeeds, then 'put->stats', if nonnull, must be
 *   zeroed.
 *
 * - If the flow's key does exist in 'dpif', then the flow's actions will
 *   be updated if 'put->flags' includes DPIF_FP_MODIFY.  Otherwise the
 *   operation will fail with EEXIST.  If the flow's actions are updated,
 *   then its statistics will be zeroed if 'put->flags' includes
 *   DPIF_FP_ZERO_STATS, and left as-is otherwise.
 *
 *   If the operation succeeds, then 'put->stats', if nonnull, must be set
 *   to the flow's statistics before the update.
 */
int (*flow_put)(struct dpif *dpif, const struct dpif_flow_put *put);

/* Deletes a flow from 'dpif' and returns 0, or returns ENOENT if 'dpif'
 * does not contain such a flow.  The flow is specified by the Netlink
 * attributes with types OVS_KEY_ATTR_* in the 'del->key_len' bytes
 * starting at 'del->key'.
 *
 * If the operation succeeds, then 'del->stats', if nonnull, must be set to
 * the flow's statistics before its deletion. */
int (*flow_del)(struct dpif *dpif, const struct dpif_flow_del *del);

/* Deletes all flows from 'dpif' and clears all of its queues of received
 * packets. */
int (*flow_flush)(struct dpif *dpif);

/* Attempts to begin dumping the flows in a dpif.  On success, returns 0
 * and initializes '*statep' with any data needed for iteration.  On
 * failure, returns a positive errno value. */
int (*flow_dump_start)(const struct dpif *dpif, void **statep);

/* Attempts to retrieve another flow from 'dpif' for 'state', which was
```

```
 * initialized by a successful call to the 'flow_dump_start' function for
 * 'dpif'.  On success, updates the output parameters as described below
 * and returns 0.  Returns EOF if the end of the flow table has been
 * reached, or a positive errno value on error.  This function will not be
 * called again once it returns nonzero within a given iteration (but the
 * 'flow_dump_done' function will be called afterward).
 *
 * On success, if 'key' and 'key_len' are nonnull then '*key' and
 * '*key_len' must be set to Netlink attributes with types OVS_KEY_ATTR_*
 * representing the dumped flow's key.  If 'actions' and 'actions_len' are
 * nonnull then they should be set to Netlink attributes with types
 * OVS_ACTION_ATTR_* representing the dumped flow's actions.  If 'stats'
 * is nonnull then it should be set to the dumped flow's statistics.
 *
 * All of the returned data is owned by 'dpif', not by the caller, and the
 * caller must not modify or free it.  'dpif' must guarantee that it
 * remains accessible and unchanging until at least the next call to
 * 'flow_dump_next' or 'flow_dump_done' for 'state'. */
int (*flow_dump_next)(const struct dpif *dpif, void *state,
                      const struct nlattr **key, size_t *key_len,
                      const struct nlattr **actions, size_t *actions_len,
                      const struct dpif_flow_stats **stats);

/* Releases resources from 'dpif' for 'state', which was initialized by a
 * successful call to the 'flow_dump_start' function for 'dpif'.  */
int (*flow_dump_done)(const struct dpif *dpif, void *state);

/* Performs the 'execute->actions_len' bytes of actions in
 * 'execute->actions' on the Ethernet frame specified in 'execute->packet'
 * taken from the flow specified in the 'execute->key_len' bytes of
 * 'execute->key'.  ('execute->key' is mostly redundant with
 * 'execute->packet', but it contains some metadata that cannot be
 * recovered from 'execute->packet', such as tun_id and in_port.) */
```

```
int (*execute)(struct dpif *dpif, const struct dpif_execute *execute);

/* Executes each of the 'n_ops' operations in 'ops' on 'dpif', in the order
 * in which they are specified, placing each operation's results in the
 * "output" members documented in comments.
 *
 * This function is optional.  It is only worthwhile to implement it if
 * 'dpif' can perform operations in batch faster than individually. */
void (*operate)(struct dpif *dpif, struct dpif_op **ops, size_t n_ops);

/* Enables or disables receiving packets with dpif_recv() for 'dpif'.
 * Turning packet receive off and then back on is allowed to change Netlink
 * PID assignments (see ->port_get_pid()).  The client is responsible for
 * updating flows as necessary if it does this. */
int (*recv_set)(struct dpif *dpif, bool enable);

/* Translates OpenFlow queue ID 'queue_id' (in host byte order) into a
 * priority value used for setting packet priority. */
int (*queue_to_priority)(const struct dpif *dpif, uint32_t queue_id,
                        uint32_t *priority);

/* Polls for an upcall from 'dpif'.  If successful, stores the upcall into
 * '*upcall', using 'buf' for storage.  Should only be called if 'recv_set'
 * has been used to enable receiving packets from 'dpif'.
 *
 * The implementation should point 'upcall->packet' and 'upcall->key' into
 * data in the caller-provided 'buf'.  If necessary to make room, the
 * implementation may expand the data in 'buf'.  (This is hardly a great
 * way to do things but it works out OK for the dpif providers that exist
 * so far.)
 *
 * This function must not block.  If no upcall is pending when it is
 * called, it should return EAGAIN without blocking. */
```

```
    int (*recv)(struct dpif *dpif, struct dpif_upcall *upcall,
                struct ofpbuf *buf);

    /* Arranges for the poll loop to wake up when 'dpif' has a message queued
     * to be received with the recv member function. */
    void (*recv_wait)(struct dpif *dpif);

    /* Throws away any queued upcalls that 'dpif' currently has ready to
     * return. */
    void (*recv_purge)(struct dpif *dpif);
};
```

各个抽象 API 具体的实现，仍然分为 dpif_linux_class（lib/dpif-linux.c）和 dpif_netdev_class（lib/dpif-netdev.c）两个具体类型。这些 api 作为中间层，对上层则被 lib/dpif.c 文件中的高级 api 所封装使用。

这里以 dpif_flow_put()（lib/dpif.c）为例分析 netlink 消息的构建和发出过程。该函数试图对绑定的 datapath 中的 flow 进行设置。其主要调用了函数 dpif_flow_put__()，而 dpif_flow_put__()的代码为

```
static int
dpif_flow_put__(struct dpif *dpif, const struct dpif_flow_put *put)
{
    int error;

    COVERAGE_INC(dpif_flow_put);
    assert(!(put->flags & ~(DPIF_FP_CREATE | DPIF_FP_MODIFY
                            | DPIF_FP_ZERO_STATS)));

    error = dpif->dpif_class->flow_put(dpif, put);
    if (error && put->stats) {
        memset(put->stats, 0, sizeof *put->stats);
    }
    log_flow_put_message(dpif, put, error);
    return error;
}
```

可见，其调用了具体的 dpif_class 中的抽象接口，以 dpif_linux_class 为例，该接口实际为 dpif_linux_flow_put()，其代码如下

```
static int
dpif_linux_flow_put(struct dpif *dpif_, const struct dpif_flow_put *put)
{
    struct dpif_linux_flow request, reply;
    struct ofpbuf *buf;
    int error;

    dpif_linux_init_flow_put(dpif_, put, &request);
    error = dpif_linux_flow_transact(&request,
                                put->stats ? &reply : NULL,
                                put->stats ? &buf : NULL);
    if (!error && put->stats) {
        dpif_linux_flow_get_stats(&reply, put->stats);
        ofpbuf_delete(buf);
    }
    return error;
}
```

从代码中，可见主要执行两个过程，利用 dpif_linux_init_flow_put()进行初始化，和利用 dpif_linux_flow_transact()发送 nlmsg。

dpif_linux_init_flow_put()利用 put 构建了 request 消息，代码为

```
static void
dpif_linux_init_flow_put(struct dpif *dpif_, const struct dpif_flow_put *put,
                         struct dpif_linux_flow *request)
{
    static struct nlattr dummy_action;

    struct dpif_linux *dpif = dpif_linux_cast(dpif_);

    dpif_linux_flow_init(request);
    request->cmd = (put->flags & DPIF_FP_CREATE
                    ? OVS_FLOW_CMD_NEW : OVS_FLOW_CMD_SET);
    request->dp_ifindex = dpif->dp_ifindex;
    request->key = put->key;
    request->key_len = put->key_len;
    /* Ensure that OVS_FLOW_ATTR_ACTIONS will always be included. */
    request->actions = put->actions ? put->actions : &dummy_action;
    request->actions_len = put->actions_len;
    if (put->flags & DPIF_FP_ZERO_STATS) {
        request->clear = true;
    }
    request->nlmsg_flags = put->flags & DPIF_FP_MODIFY ? 0 : NLM_F_CREATE;
}
```

而 dpif_linux_flow_transact()则具体负责发出 nlmsg，代码为

```c
static int
dpif_linux_flow_transact(struct dpif_linux_flow *request,
                         struct dpif_linux_flow *reply, struct ofpbuf **bufp)
{
    struct ofpbuf *request_buf;
    int error;

    assert((reply != NULL) == (bufp != NULL));

    if (reply) {
        request->nlmsg_flags |= NLM_F_ECHO;
    }

    request_buf = ofpbuf_new(1024);
    dpif_linux_flow_to_ofpbuf(request, request_buf);
    error = nl_sock_transact(genl_sock, request_buf, bufp);
    ofpbuf_delete(request_buf);

    if (reply) {
        if (!error) {
            error = dpif_linux_flow_from_ofpbuf(reply, *bufp);
        }
        if (error) {
            dpif_linux_flow_init(reply);
            ofpbuf_delete(*bufp);
            *bufp = NULL;
        }
    }
    return error;
}
```

dpif_linux_flow_transact()的第一个参数 request 为发送消息的相关数据，第二个参数和第三个参数如果给定非空，则用来存储可能收到的回复信息。

该函数首先调用 dpif_linux_flow_to_ofpbuf()函数，利用 request 中的 attrs 信息创建一个 struct ofpbuf *request_buf=ovs_header+attrs。其中 ovs_header 结构（include/linux/openvswitch.h）中仅保存了一个 dp_ifindex 信息。之后则调用 nl_sock_transact()函数发出消息。

nl_sock_transact()函数代码为

```
int
nl_sock_transact(struct nl_sock *sock, const struct ofpbuf *request,
            struct ofpbuf **replyp)
{
   struct nl_transaction *transactionp;
   struct nl_transaction transaction;

   transaction.request = CONST_CAST(struct ofpbuf *, request);
   transaction.reply = replyp ? ofpbuf_new(1024) : NULL;
   transactionp = &transaction;

   nl_sock_transact_multiple(sock, &transactionp, 1);

   if (replyp) {
      if (transaction.error) {
         ofpbuf_delete(transaction.reply);
         *replyp = NULL;
      } else {
         *replyp = transaction.reply;
      }
   }

   return transaction.error;
}
```

nl_sock_transact()函数进一步调用了 nl_sock_transact_multiple()函数发出消息。

nl_sock_transact_multiple()函数第一个参数为要发送消息的 socket，第二个参数存储了需要发送的消息，第三个参数指定发出消息的数目。函数中主要调用了 nl_sock_transact_multiple__()函数构建 nlmsg 并发出。

其他类型的行为的处理过程类似，都遵循如图 1 所示的过程。

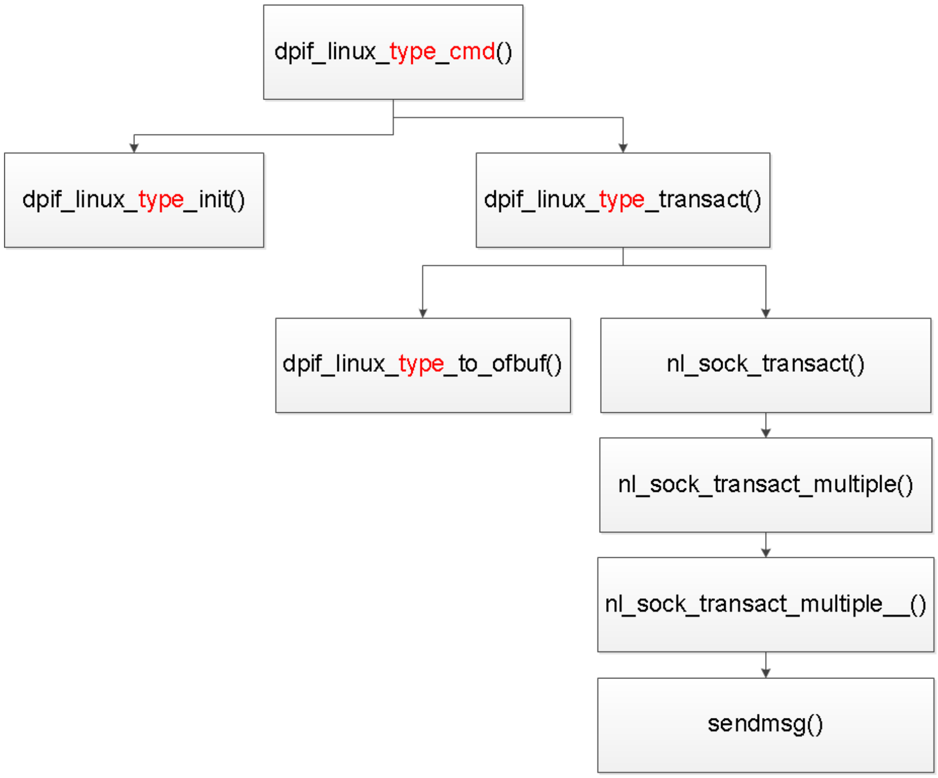


图 1 ovsd 利用 netlink 发送消息流程