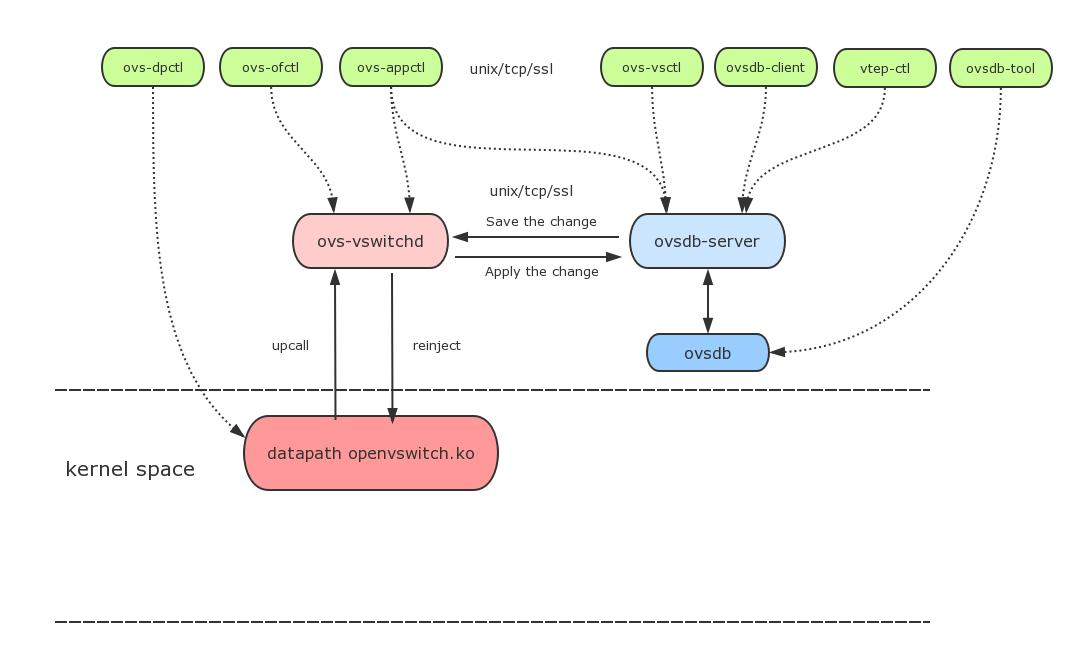
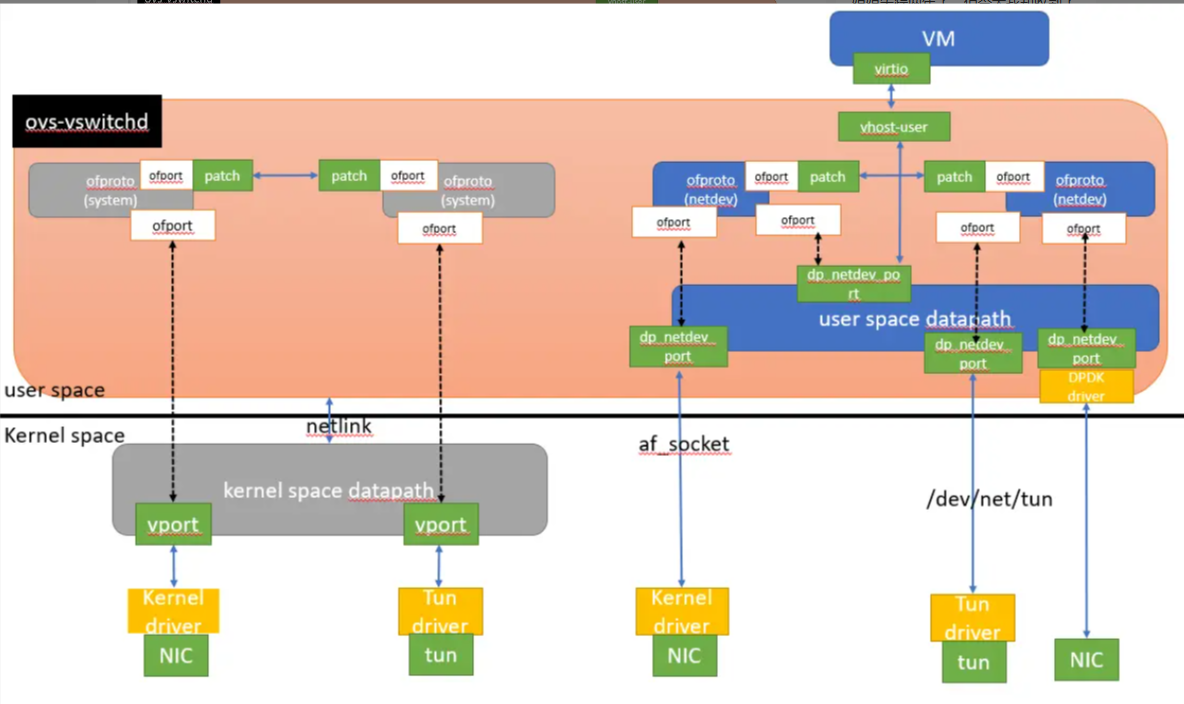
# ovs 的基本工作原理



# 基本概念

Ovs 中，重点要理解 datapath 的概念，datapath 负责执行数据交换，也就是把从接收端口收到的数据包在流表中进行匹配，并执行匹配到的动作。Datapath 是 ofproto（OpenFlow） 接口的唯一实现。Ovs 中有两种datapath：system 和 netdev。其中 system 是内核态的 datapath 实现，通过一个内核模块，报文交换逻辑在内核实现。Netdev 是用户态的 datapath，支持两种类型的端口: non-pmd/pmd，前者可以是linux系统上绑定在kernel driver的物理网卡，也可以是tap等虚拟网卡，后者需要dpdk的支持，所以需要在编译时，就指定dpdk提供的lib，其使用dpdk的pmd线程处理网卡，可以是物理网卡，也可以是vhost-user虚拟网卡。

kernel datapath和userspace datapath是可以共存的，下图为ovs集成dpdk后，两种datapath框架图。



a. 每种datapath可以支持多个网桥，不同datapath类型的网桥不能互通。

b. patch端口只存在于ofproto层面，不会下发到datapath中，所以通过 "ovs-appctl dpctl/show" 命令是看不到patch端口的。

c. kernel datapath在openvswitch.ko模块中直接调用网卡驱动进行收发包。

userspace datapath中，对于non-pmd端口，使用af\_socket或者read/write /dev/net/tun实现收发包，对于pmd类型端口，使用其dpdk driver在用户态实现收发包。

## Datapath 和网桥、端口的关系？

# 关于 bridge\_run\_\_()

bridge\_run\_\_() 调用了两个级别的 run 函数：

* Dapath\_type 级别的，实际调用的是 ofproto\_class->type\_run()
* Ofproto 级别的（网桥或者交换机，多个网桥或交换机属于同一个 datapath\_type），实际调用的是

要搞清楚这两个级别的run，需要搞清楚源码中哪些结构是属于 datapath\_type 级别的，哪些是属于 ofproto 级别的。

属于 datapath\_type 级别的对象：

* dpif\_backer

属于 ofproto 级别的对象：

# dpif

在 dpif\_backer 中有 dpif 和 udpif 两个结构：

|  |
| --- |
| （ofproto/ofproto-dpif.h）  struct dpif\_backer {  …  struct dpif \*dpif;  struct udpif \*udpif;  …  } |

如前文所述，ofproto 是 OpenFlow 交换机接口的抽象，dpif 也是一层抽象，是 ovs datapath 接口的抽象。对应的实现也是接口+数据，接口定义是 dpif\_class，数据是 dpif，依据于不同的实现，具体的数据定义在 dpif 结构上扩展。一个 dpif 结构对应的就是一个 datapath。目前 dpif 有两个实现（称为 dpif provider），dpif\_netdev\_class 和 dpif\_netlink\_class，对应的结构为 dpif\_netdev 和 dpif\_netlink：

|  |
| --- |
| （lib/dpif-netdev.c）  /\* Interface to netdev-based datapath. \*/  struct dpif\_netdev {  struct dpif dpif;  struct dp\_netdev \*dp;  uint64\_t last\_port\_seq;  };  （lib/dpif-netlink.c）  /\* Datapath interface for the openvswitch Linux kernel module. \*/  struct dpif\_netlink {  struct dpif dpif;  …  } |

# dpif\_class 相关实现

dpif\_class 抽象datapath 相关的接口，定义在：

|  |
| --- |
| （lib/dpif-provider.h）  struct dpif\_class {  // 对应 datapath 的类型，如 system 和 netdev，默认类型是 system  const char \*type;  …  } |

## 注册与 init()

dpif\_class 也需要注册。为此，源码中专门定义了一个 registered\_class 结构来表示注册的 dpif\_class：

|  |
| --- |
| （lib/dpif.c）  struct registered\_dpif\_class {  const struct dpif\_class \*dpif\_class; // 指向对应的 dpif\_class  int refcount;  }; |

所有注册的 registered\_class 保存在全局变量 dpif\_classes 中：

|  |
| --- |
| （lib/dpif.c）  static struct shash dpif\_classes = SHASH\_INITIALIZER(&dpif\_classes); |

这是一个哈希表，通过 dpif\_class type 来索引。

通过dp\_register\_provider() 函数注册 dpif\_class，系统预定了一些基础的 dpif\_class 实现，定义在 base\_dpif\_classes 全局数组中：

|  |
| --- |
| （lib/dpif.c）  static const struct dpif\_class \*base\_dpif\_classes[] = {  #if defined(\_\_linux\_\_) || defined(\_WIN32)  &dpif\_netlink\_class,  #endif  &dpif\_netdev\_class,  }; |

base\_dpif\_classes 数组中的 dpif\_class 在 dp\_initialize() 函数中调用 dp\_register\_provider() 进行注册：

|  |
| --- |
| （lib/dpif.c）  static int  dp\_register\_provider\_\_(const struct dpif\_class \*new\_class)  {  struct registered\_dpif\_class \*registered\_class;  int error;  // dpif\_disallowed 中是明确禁止的，不允许注册  if (sset\_contains(&dpif\_disallowed, new\_class->type)) {  VLOG\_DBG("attempted to register disallowed provider: %s",  new\_class->type);  return EINVAL;  }  // 已经注册过了的  if (shash\_find(&dpif\_classes, new\_class->type)) {  VLOG\_WARN("attempted to register duplicate datapath provider: %s",  new\_class->type);  return EEXIST;  }  // 如果 dpif\_class 提供了 init()，调用 init()  error = new\_class->init ? new\_class->init() : 0;  if (error) {  VLOG\_WARN("failed to initialize %s datapath class: %s",  new\_class->type, ovs\_strerror(error));  return error;  }  // 初始化一个 registered\_class 结构，保存注册的 dpif\_class 信息  registered\_class = xmalloc(sizeof \*registered\_class);  registered\_class->dpif\_class = new\_class;  registered\_class->refcount = 0;  // 保存到全局的 dpif\_classes 中  shash\_add(&dpif\_classes, new\_class->type, registered\_class);  return 0;  } |

dpif\_class 中 init 函数定义为：

|  |
| --- |
| （lib/dpif-provider.h）  struct dpif\_class {  …  int (\*init)(void);  …  } |

dpif\_netlink\_class 没有提供 init 函数，而 dpif\_netdev\_class 提供的 init 函数为 dpif\_netdev\_init，主要是在其中注册了一些控制接口，这里不再赘述。

## enumerate()

enumerate() 用于返回该 dpif\_class 下所有已创建的 datapath 名称，定义为：

|  |
| --- |
| （lib/dpif-provider.h）  struct dpif\_class {  …  // 参数 all\_dps 是返回值，保存获取到的该 dpif\_class 下所有的 datapath 名称集合  // dpif\_class 参数无用  int (\*enumerate)(struct sset \*all\_dps, const struct dpif\_class \*dpif\_class);  …  } |

dpif\_netlink\_class 的实现是 dpif\_netlink\_enumerate。它会向内核发送 OVS\_DP\_CMD\_GET 的netlink 消息，获取所有的 datapath。

## open()

open() 函数用于打开或创建一个指定name 的 dpif，定义为：

|  |
| --- |
| （lib/dpif-provider.h）  struct dpif\_class {  …  // 参数中，name 为 dpif 的名称（datapath的名称），如 ovs-system  // create：不存在时是否创建  // dpifp 返回值，返回创建的 dpif  int (\*open)(const struct dpif\_class \*dpif\_class,  const char \*name, bool create, struct dpif \*\*dpifp);  …  } |

dpif\_netlink\_class 的实现是 dpif\_netlink\_open，它会向内核发送 OVS\_DP\_CMD\_NEW 创建 datapath。

## run()

run() 函数在 bridge\_run\_\_() 中周期性调用，定义为：

|  |
| --- |
| （lib/dpif-provider.h）  struct dpif\_class {  …  /\* Performs periodic work needed by 'dpif', if any is necessary.  \* Returns true if need to revalidate. \*/  // 参数 dpif 为对应的 dpif 结构  bool (\*run)(struct dpif \*dpif);  …  } |

dpif 和 udpif 再 dpif\_backer 被初始化时初始化，具体是在 open\_dpif\_backer() 函数中:

|  |
| --- |
| （ofproto/ofproto-dpif.c）  static int  open\_dpif\_backer(const char \*type, struct dpif\_backer \*\*backerp)  {  …  error = dpif\_create\_and\_open(backer\_name, type, &backer->dpif);  free(backer\_name);  if (error) {  VLOG\_ERR("failed to open datapath of type %s: %s", type,  ovs\_strerror(error));  free(backer);  return error;  }  backer->udpif = udpif\_create(backer, backer->dpif);  …  } |

其中，dpif 通过 dpif\_create\_and\_open() 初始化，udpif 通过 udpif\_create() 初始化。

# ovs-ofctl add-flow 命令实现

（参考：[Openvswitch原理与代码分析(7): 添加一条流表flow - popsuper1982 - 博客园 (cnblogs.com)](https://www.cnblogs.com/popsuper1982/p/5904329.html)）

## 1 基本流程

Client 端的基本流程：

* 解析流表规则为 ofputil\_flow\_mod 结构，主要在 parse\_ofp\_flow\_mod\_str() 函数中
* 将ofputil\_flow\_mod 结构转换为 openflow 协议，通过 socket 发送给 ovs-vswitchd，这段逻辑在 ofctl\_flow\_mod\_\_() 函数中

### client 端

client 端对应的是 ovs-ofctl 工具。

|  |
| --- |
| ovs-ofctl add-flow br-test "in\_port=1,actions=drop" |

命令的实现函数为 ofctl\_add\_flow：

|  |
| --- |
| （utilities/ovs-ofctl.c）  static const struct ovs\_cmdl\_command all\_commands[] = {  …  { "add-flow", "switch flow",  2, 2, ofctl\_add\_flow, OVS\_RW },  …  } |

#### ofctl\_add\_flow

|  |
| --- |
| （utilities/ovs-ofctl.c）  static void  ofctl\_add\_flow(struct ovs\_cmdl\_context \*ctx)  {  ofctl\_flow\_mod(ctx->argc, ctx->argv, OFPFC\_ADD);  } |

ofctl\_flow\_mod 是一个公共函数，流的添加，修改，删除都由它实现。通过command 来指定操作，这里的 command 是 OFPFC\_ADD，表示添加 flow。这个 command 类型定义在：

|  |
| --- |
| （include/openflow/openflow-common.h）  enum ofp\_flow\_mod\_command {  OFPFC\_ADD, /\* New flow. \*/  OFPFC\_MODIFY, /\* Modify all matching flows. \*/  OFPFC\_MODIFY\_STRICT, /\* Modify entry strictly matching wildcards \*/  OFPFC\_DELETE, /\* Delete all matching flows. \*/  OFPFC\_DELETE\_STRICT /\* Strictly match wildcards and priority. \*/  }; |

ofctl\_flow\_mod：

|  |
| --- |
| // 参数中，argc 为命令行参数的数目，本例中为3  // argv 是命令行参数，本例中有3个，argv[0] 是 “add-flow” 这个子命令 // argv[1] 是网桥 br-test，argv[2] 是流表规则 “in\_port=1,actions=drop"  // command 上面已经说过  static void  ofctl\_flow\_mod(int argc, char \*argv[], uint16\_t command)  {  if (argc > 2 && !strcmp(argv[2], "-")) {  // 从文件导入流表规则  ofctl\_flow\_mod\_file(argc, argv, command);  } else {  struct ofputil\_flow\_mod fm;  char \*error;  enum ofputil\_protocol usable\_protocols;  error = parse\_ofp\_flow\_mod\_str(&fm, argc > 2 ? argv[2] : "",  ports\_to\_accept(argv[1]),  tables\_to\_accept(argv[1]), command,  &usable\_protocols);  if (error) {  ovs\_fatal(0, "%s", error);  }  ofctl\_flow\_mod\_\_(argv[1], &fm, 1, usable\_protocols);  }  } |

#### parse\_ofp\_flow\_mod\_str

解析流表规则字符串。

|  |
| --- |
| （lib/ofp-flow.c）  // 参数中，fm 是输出，保存解析结果，string 是流表规则的字符串  // command 表示执行动作（添加、删除流表），本例中是 OFPFC\_ADD  char \* OVS\_WARN\_UNUSED\_RESULT  parse\_ofp\_flow\_mod\_str(struct ofputil\_flow\_mod \*fm, const char \*string,  const struct ofputil\_port\_map \*port\_map,  const struct ofputil\_table\_map \*table\_map,  int command,  enum ofputil\_protocol \*usable\_protocols)  {  char \*error = parse\_ofp\_str(fm, command, string, port\_map, table\_map,  usable\_protocols);  if (!error) {  /\* Normalize a copy of the match. This ensures that non-normalized  \* flows get logged but doesn't affect what gets sent to the switch, so  \* that the switch can do whatever it likes with the flow. \*/  struct match match;  minimatch\_expand(&fm->match, &match);  ofputil\_normalize\_match(&match);  }  return error;  } |

#### parse\_ofp\_str

|  |
| --- |
| （lib/ofp-flow.c）  // 参数和 parse\_ofp\_flow\_mod\_str() 函数的一致  char \* OVS\_WARN\_UNUSED\_RESULT  parse\_ofp\_str(struct ofputil\_flow\_mod \*fm, int command, const char \*str\_,  const struct ofputil\_port\_map \*port\_map,  const struct ofputil\_table\_map \*table\_map,  enum ofputil\_protocol \*usable\_protocols)  {  // 复制流表规则的字符串  char \*string = xstrdup(str\_);  char \*error;  error = parse\_ofp\_str\_\_(fm, command, string, port\_map, table\_map,  usable\_protocols);  if (error) {  fm->ofpacts = NULL;  fm->ofpacts\_len = 0;  }  free(string);  return error;  } |

parse\_ofp\_str\_\_：

|  |
| --- |
| （lib/ofp-flow.c） |

### Server 端

首先，server端 ovs-vswitchd 会给每个网桥监听管理服务和监控服务，用于执行 OpenFlow 管理命令和监控命令，参考：[ovs-ofctl(8) - Linux manual page (man7.org)](https://www.man7.org/linux/man-pages/man8/ovs-ofctl.8.html)。其中，管理的 socket 名称为 <ovs\_run\_dir>/bridge.mgmt，监控的 docket 名称为 <ovs\_run\_dir>/bridge.snoop。例如：



这个服务的配置是在 bridge\_configure\_remotes() 中进行的。

# ovs-ofctl dump-flows 命令实现

命令：ovs-ofctl dump-flows br-int

## 命令行参数和子命令解析

命令行参数解析在 parse\_options() 中，我们没有指定额外的参数，所以没有啥参数解析。

dump-flows是子命令：

|  |
| --- |
| int  main(int argc, char \*argv[])  {  …  // 运行子命令  if (read\_only) {  // 只读方式运行，指定了 --read-only 参数  ovs\_cmdl\_run\_command\_read\_only(&ctx, get\_all\_commands());  } else {  // 正常运行  ovs\_cmdl\_run\_command(&ctx, get\_all\_commands());  }  return 0;  } |

所以的子命令定义在全局的 all\_commands 数组中，get\_all\_commands() 函数返回该数组：

|  |
| --- |
| （utilities/ovs-ofctl.c）  static const struct ovs\_cmdl\_command all\_commands[] = {  …  { "dump-flows", "switch",  1, 2, ofctl\_dump\_flows, OVS\_RO },  …  } |

Ofctl\_dupm\_flows 是 dump-flows 子命令的处理函数。

## ofctl\_dump\_flows

|  |
| --- |
| （utilities/ovs-ofctl.c）  static void  ofctl\_dump\_flows(struct ovs\_cmdl\_context \*ctx)  {  if (!n\_criteria && !should\_show\_names() && show\_stats) {  // 这里命中的是简单的这个分支  ofctl\_dump\_flows\_\_(ctx->argc, ctx->argv, false);  return;  } else {  …  }  } |

|  |
| --- |
| （utilities/ovs-ofctl.c）  static void  ofctl\_dump\_flows\_\_(int argc, char \*argv[], bool aggregate)  {  struct ofputil\_flow\_stats\_request fsr;  enum ofputil\_protocol protocol;  struct vconn \*vconn;  // 准备 dump flow 的请求参数，主要是 fsr 和 protocol 两个参数  vconn = prepare\_dump\_flows(argc, argv, aggregate, &fsr, &protocol);  dump\_transaction(vconn, ofputil\_encode\_flow\_stats\_request(&fsr, protocol));  vconn\_close(vconn);  } |

## prepare\_dump\_flows

|  |
| --- |
| （utilities/ovs-ofctl.c）  // 参数中，argc 为参数的个数  // argv 为参数，示例中是 “dump-flows br-int”，即 argv[0] 是子命令，argv[1]是网桥名称，argv[2]（如果有的话）是一些过滤条件，如 table=0  static struct vconn \*  prepare\_dump\_flows(int argc, char \*argv[], bool aggregate,  struct ofputil\_flow\_stats\_request \*fsr,  enum ofputil\_protocol \*protocolp)  {  const char \*vconn\_name = argv[1]; // 网桥名称  enum ofputil\_protocol usable\_protocols, protocol;  struct vconn \*vconn;  char \*error;  const char \*match = argc > 2 ? argv[2] : ""; // argc >2 说明指定了过滤条件  // 获取所有的 port name 和 id 的映射  const struct ofputil\_port\_map \*port\_map  = \*match ? ports\_to\_accept(vconn\_name) : NULL;  // 获取所有的 table name 和 id 的映射  const struct ofputil\_table\_map \*table\_map  = \*match ? tables\_to\_accept(vconn\_name) : NULL;  // 准备 ofputil\_flow\_stats\_request，保存到 fsr 变量中  error = parse\_ofp\_flow\_stats\_request\_str(fsr, aggregate, match,  port\_map, table\_map,  &usable\_protocols);  if (error) {  ovs\_fatal(0, "%s", error);  }  protocol = open\_vconn(vconn\_name, &vconn);  \*protocolp = set\_protocol\_for\_flow\_dump(vconn, protocol, usable\_protocols);  return vconn;  } |

# ovs 中的基本概念

todo

ofproto\_class 和 ofproto\_class

## dpif\_class 和 dpif

目前有两种类型的 dpif：dpif\_netlink 和 dpif\_netdev，它们都是在struct dpif 基础上扩展的。

dpif\_netlink 结构：

|  |
| --- |
| (lib/dpif-netlink.c)  /\* Datapath interface for the openvswitch Linux kernel module. \*/  struct dpif\_netlink {  struct dpif dpif;  int dp\_ifindex;  uint32\_t user\_features;  /\* Upcall messages. \*/  struct fat\_rwlock upcall\_lock;  struct dpif\_handler \*handlers;  uint32\_t n\_handlers; /\* Num of upcall handlers. \*/  /\* Per-vport dispatch mode. \*/  struct dpif\_channel \*channels; /\* Array of channels for each port. \*/  int uc\_array\_size; /\* Size of 'handler->channels' and \*/  /\* 'handler->epoll\_events'. \*/  /\* Change notification. \*/  struct nl\_sock \*port\_notifier; /\* vport multicast group subscriber. \*/  bool refresh\_channels;  }; |

dpif\_netdev 结构：

|  |
| --- |
| (lib/dpif-netdev.c)  /\* Interface to netdev-based datapath. \*/  struct dpif\_netdev {  struct dpif dpif;  struct dp\_netdev \*dp;  uint64\_t last\_port\_seq;  }; |

## dpif\_backer

dpif\_backer 表示交换机和 datapath 的接口，相同 type 的 datapath 共享一个 backer，Linux 内核支持两种 type 的 datapath ：”system” 和 “netdev”，所以就有两种类型的 datapath 接口：默认的 dpif\_netlink , 以及 DPDK 下使用的 dpif\_netdev。

在 open\_dpif\_backer() 中创建，在 close\_dpif\_backer() 中销毁。

|  |
| --- |
| （ofproto/ofproto-dpif.h）  /\* All datapaths of a given type share a single dpif backer instance. \*/  struct dpif\_backer {  char \*type;  int refcount;  struct dpif \*dpif;  struct udpif \*udpif;  struct ovs\_rwlock odp\_to\_ofport\_lock;  struct hmap odp\_to\_ofport\_map OVS\_GUARDED; /\* Contains "struct ofport"s. \*/  struct simap tnl\_backers; /\* Set of dpif ports backing tunnels. \*/  enum revalidate\_reason need\_revalidate; /\* Revalidate all flows. \*/  bool recv\_set\_enable; /\* Enables or disables receiving packets. \*/  /\* Meter. \*/  struct id\_pool \*meter\_ids; /\* Datapath meter allocation. \*/  /\* Connection tracking. \*/  struct id\_pool \*tp\_ids; /\* Datapath timeout policy id  \* allocation. \*/  struct cmap ct\_zones; /\* "struct ct\_zone"s indexed by zone  \* id. \*/  struct hmap ct\_tps; /\* "struct ct\_timeout\_policy"s indexed  \* by timeout policy (struct simap). \*/  struct ovs\_list ct\_tp\_kill\_list; /\* A list of timeout policy to be  \* deleted. \*/  /\* Version string of the datapath stored in OVSDB. \*/  char \*dp\_version\_string;  /\* Datapath feature support. \*/  struct dpif\_backer\_support bt\_support; /\* Boot time support. Set once  when vswitch starts up, then  it is read only through out  the life time of vswitchd. \*/  struct dpif\_backer\_support rt\_support; /\* Runtime support. Can be  set to a lower level in  feature than 'bt\_support'. \*/  struct atomic\_count tnl\_count;  }; |

#### all\_dpif\_backers

如前文所述，dpif\_backer 和 datapath type 对应，如果多个网桥使用相同的 datapath\_type，它们对应的是同一个 dpif\_backer 实例，根据 datapath\_type 来索引 dpif\_backer。所有的 dpif\_backer 实例保存在全局变量 all\_dpif\_backers 中：

|  |
| --- |
| （ofproto/ofproto-dpif.c）  /\* All existing ofproto\_backer instances, indexed by ofproto->up.type. \*/  struct shash all\_dpif\_backers = SHASH\_INITIALIZER(&all\_dpif\_backers); |

在 open\_dpif\_backer() 函数中，会先去查找，如果没有，则创建后会添加到 all\_dpif\_backers 中：

|  |
| --- |
| （ofproto/ofproto-dpif.c）  static int  open\_dpif\_backer(const char \*type, struct dpif\_backer \*\*backerp)  {  …  backer = shash\_find\_data(&all\_dpif\_backers, type);  if (backer) {  backer->refcount++;  \*backerp = backer;  return 0;  }  …  shash\_add(&all\_dpif\_backers, type, backer);  …  } |

# 问题：ovsdb 中的虚拟网桥和虚拟端口什么时候同步到内核？

ovs-vsctl 只会修改 ovsdb 中的信息，同步到内核是 vswitchd 做的。答案是在 vswitchd 中，会循环调用 bridge\_run() 和 netdev\_run()，这两个函数中会做同步：

|  |
| --- |
| （vswitchd/ovs-vswitchd.c）  int  main(int argc, char \*argv[])  {  …  while (!exiting) {  …  bridge\_run();  …  netdev\_run();  }  } |

# vswitchd 内存中网桥信息的更新(all\_bridges)

vswitchd 中使用的信息是内存中的副本，vswitchd 会从 ovsdb 从同步一份信息到内存中，然后使用内存中的信息。

vswitchd 在 bridge\_reconfigure() 函数中同步 ovsdb 的信息到内存。

## all\_bridges 与 bridge 同步

vswtichd 中把所有的 bridge 信息保存在 all\_bridges 中：

|  |
| --- |
| （vswitchd/bridge.c）  /\* All bridges, indexed by name. \*/  static struct hmap all\_bridges = HMAP\_INITIALIZER(&all\_bridges); |

这是一个哈希表，通过 bridge\_name 索引，struct bridge 结构通过 node 成员挂到哈希表上。

struct bridge 结构：

|  |
| --- |
| （vswitchd/bridge.c）  struct bridge {  struct hmap\_node node; /\* In 'all\_bridges'. \*/  char \*name; /\* User-specified arbitrary name. \*/  char \*type; /\* Datapath type. \*/  struct eth\_addr ea; /\* Bridge Ethernet Address. \*/  struct eth\_addr default\_ea; /\* Default MAC. \*/  // 指向 ovsdb 中对应的行  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. \*/  /\* Port mirroring. \*/  struct hmap mirrors; /\* "struct mirror" indexed by UUID. \*/  /\* Auto Attach \*/  struct hmap mappings; /\* "struct" indexed by UUID \*/  /\* Used during reconfiguration. \*/  struct shash wanted\_ports;  /\* Synthetic local port if necessary. \*/  struct ovsrec\_port synth\_local\_port;  struct ovsrec\_interface synth\_local\_iface;  struct ovsrec\_interface \*synth\_local\_ifacep;  }; |

Bridge 同步在 add\_del\_bridges() 函数中，对于 vswitchd 内存中存在的但 ovsdb 中不存在的，调用 bridge\_destroy() 从 vswitchd 的内存中删除，对于 vswitchd 内存中不存在但 ovsdb 中存在的，调用 bridge\_create() 添加到 vswitchd 内存中。

## port 同步

add\_del\_bridges() 函数只是同步了 bridge 信息，下面看下 port 信息的同步。

在 struct bridge 中用一个 hmap 保存了所有 port 信息：

|  |
| --- |
| （vswitchd/bridge.c）  struct bridge {  …  /\* Bridge ports. \*/  struct hmap ports; /\* "struct port"s indexed by name. \*/  …  } |

port 在 bridge 中的添加和删除分别由函数 port\_create() 和 port\_destroy() 中完成：

port\_create()：

|  |
| --- |
| （vswitchd/bridge.c）  static struct port \*  port\_create(struct bridge \*br, const struct ovsrec\_port \*cfg)  {  struct port \*port;  port = xzalloc(sizeof \*port);  port->bridge = br;  port->name = xstrdup(cfg->name);  port->cfg = cfg;  ovs\_list\_init(&port->ifaces);  // 插入 port 到 br->ports 中  hmap\_insert(&br->ports, &port->hmap\_node, hash\_string(port->name, 0));  return port;  } |

port\_destroy()：

|  |
| --- |
| （vswitchd/bridge.c）  static void  port\_destroy(struct port \*port)  {  if (port) {  struct bridge \*br = port->bridge;  struct iface \*iface;  if (br->ofproto) {  ofproto\_bundle\_unregister(br->ofproto, port);  }  // 遍历 destroy port 下的 interfaces  LIST\_FOR\_EACH\_SAFE (iface, port\_elem, &port->ifaces) {  iface\_destroy\_\_(iface);  }  // 从 br->ports 中删除  hmap\_remove(&br->ports, &port->hmap\_node);  free(port->name);  free(port);  }  } |

port\_create() 调用链条为：

* bridge\_reconfigure() （vswitchd/bridge.c）
* bridge\_add\_ports() （vswitchd/bridge.c）
* bridge\_add\_ports\_\_() （vswitchd/bridge.c）

port\_destroy() 调用链条为：

* bridge\_reconfigure() （vswitchd/bridge.c）
* bridge\_del\_ports() （vswitchd/bridge.c）

## datapath 同步

vswitchd 中所有的 datapath 也保存在一个名为 all\_datapaths 的全局变量中：

|  |
| --- |
| （vswitchd/bridge.c）  /\* All datapath configuartions, indexed by type. \*/  static struct hmap all\_datapaths = HMAP\_INITIALIZER(&all\_datapaths); |

这也是类似 all\_bridges 的一个哈希表，key 是 type。

Datapath 通过 datapath\_create() 和 datapath\_destroy() 进行同步。

datapath\_create()：

|  |
| --- |
| （vswitchd/bridge.c）  static struct datapath \*  datapath\_create(const char \*type)  {  struct datapath \*dp = xzalloc(sizeof \*dp);  dp->type = xstrdup(type);  hmap\_init(&dp->ct\_zones);  // 添加到 all\_datapaths  hmap\_insert(&all\_datapaths, &dp->node, hash\_string(type, 0));  smap\_init(&dp->caps);  return dp;  } |

datapath\_destroy()：

|  |
| --- |
| （vswitchd/bridge.c）  static void  datapath\_destroy(struct datapath \*dp)  {  if (dp) {  struct ct\_zone \*ct\_zone;  HMAP\_FOR\_EACH\_SAFE (ct\_zone, node, &dp->ct\_zones) {  ofproto\_ct\_del\_zone\_timeout\_policy(dp->type, ct\_zone->zone\_id);  ct\_zone\_remove\_and\_destroy(dp, ct\_zone);  }  // 从 all\_datapaths 中删除  hmap\_remove(&all\_datapaths, &dp->node);  hmap\_destroy(&dp->ct\_zones);  free(dp->type);  smap\_destroy(&dp->caps);  free(dp);  }  } |

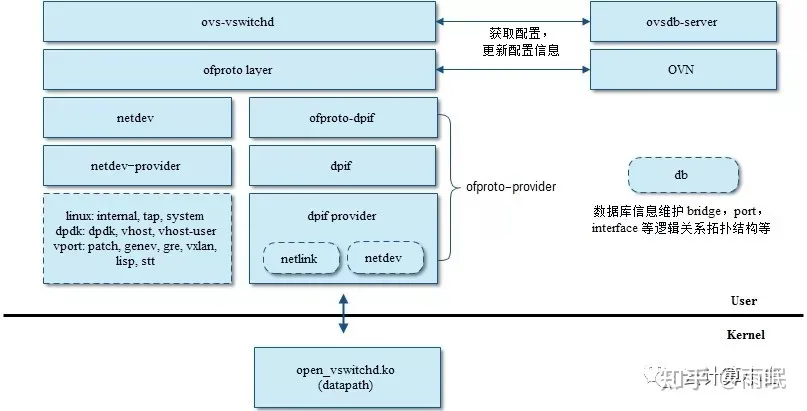
datapath\_create() 调用链条为：

* bridge\_reconfigure() （vswitchd/bridge.c）
* datapath\_reconfigure() （vswitchd/bridge.c）

datapath\_destroy() 调用链条为：

* bridge\_reconfigure() （vswitchd/bridge.c）
* datapath\_reconfigure() （vswitchd/bridge.c）

# Vswitchd



从图中可以看出 OVS 的分层结构，最上层 vswitchd 主要与 ovsdb 通信，做配置下发和更新等；中间层是 ofproto ，用于和 OpenFlow 控制器通信，并基于下层的 ofproto provider 提供的接口，完成具体的设备操作和流表操作等工作。netdev 层实现了对网络设备（如 Ethernet）的抽象，基于 netdev provider 接口实现多种不同平台的设备，如 Linux 内核的 system, tap, internal 等，dpdk 系的 vhost, vhost-user 等，以及隧道相关的 gre, vxlan 等。

vswitchd是ovs主要的用户态程序，它从ovsdb-server读取配置并发送到ofproto层，也从ofproto读取特定的状态和统计信息并发送到数据库；

ofproto是openflow的接口层，负责和Openflow controller通信并通过ofproto\_class与ofproto provider底层交互；

ofproto-dpif是ofproto接口类的具体实现；目前openvswitch只支持ofproto-dpif，但是用户可以很容易的实现其他ofproto provider。

dpif 层实现对流表的操作。

netdev是ovs系统的网络设备抽象（比如linux的net\_device或交换机的port），netdev\_class定义了netdev-provider的具体实现需要的接口，具体的平台实现需要支持这些统一的接口，从而完成netdev设备的创建、销毁、打开、关闭等一系列操作；

## ofproto\_class 、ofproto 和 ofproto\_dpif

在 struct bridge 中包含一个 ofproto 成员，其表示 OpenFlow 网桥的实现：

|  |
| --- |
| （vswitchd/bridge.c）  struct bridge {  ...  /\* OpenFlow switch processing. \*/  struct ofproto \*ofproto; /\* OpenFlow switch. \*/  ...  } |

ofproto 层可以有多个实际实现，代码中分为了接口和数据两部分，其中 struct ofproto\_class 对应接口，struct ofproto 对应数据。ofproto\_class 和 ofproto 都有具体的实现，目前只有一个实现，对应的接口是 ofproto\_dpif\_class，数据是 ofproto\_dpif。ofproto\_dpif 定义为：

|  |
| --- |
| （ofproto/ofproto-dpif.h）  struct ofproto\_dpif {  ...  struct ofproto up;  ...  } |

ofproto\_dpif 中封装了 ofproto，此外还封装了一些实现相关的内容。ofproto\_dpif 和 ofproto 的关系，类似内核中 inet\_sock 和 sock 结构的关系。ofproto 的具体实现，在源码中称为 ofproto provider，参考 ofproto/ofproto-provider.h 文件的命名。

由于这种关系，ofproto 实际是由 ofproto\_class 中的 alloc 函数进行分配的，如 ofproto\_dpif\_class 的实现：

|  |
| --- |
| （ofproto/ofproto-dpif.c）  static struct ofproto \*  alloc(void)  {  struct ofproto\_dpif \*ofproto = xzalloc(sizeof \*ofproto);  return &ofproto->up;  } |

可以看到，实际分配的是 ofproto\_dpif 结构，只不过返回时返回的是其中的 ofproto 地址。而且 ofproto\_class 的接口中也都是使用的 ofproto 结构。

### ofproto\_class 相关操作

#### 注册

ofproto\_class 需要进行注册，所有注册的 ofproto\_class 保存在一个全局数组 ofproto\_classes[] 中：

|  |
| --- |
| （ofproto/ofproto.c）  static const struct ofproto\_class \*\*ofproto\_classes; // 动态分配大小的数组  static size\_t n\_ofproto\_classes; // 数组中当前已注册的 ofproto\_class 数目  static size\_t allocated\_ofproto\_classes; // ofproto\_classes 数组的空间大小 |

通过 ofproto\_class\_register() 函数进行注册：

|  |
| --- |
| （ofproto/ofproto.c）  int  ofproto\_class\_register(const struct ofproto\_class \*new\_class)  {  size\_t i;  // 已经注册过  for (i = 0; i < n\_ofproto\_classes; i++) {  if (ofproto\_classes[i] == new\_class) {  return EEXIST;  }  }  // ofproto\_classes 数组扩容  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; // 保存  return 0;  } |

ofproto\_dpif\_class 是在 ofproto\_init() 中注册的：

|  |
| --- |
| （ofproto/ofproto.c）  void  ofproto\_init(const struct shash \*iface\_hints)  {  …  ofproto\_class\_register(&ofproto\_dpif\_class);  …  } |

#### 使用

注册后，ofproto\_class 通过 ofproto\_class\_find\_\_() 函数来查找使用，参考 ofproto\_create()：

|  |
| --- |
| （ofproto/ofproto.c）  int  ofproto\_create(const char \*datapath\_name, const char \*datapath\_type,  struct ofproto \*\*ofprotop)  OVS\_EXCLUDED(ofproto\_mutex)  {  …  datapath\_type = ofproto\_normalize\_type(datapath\_type);  class = ofproto\_class\_find\_\_(datapath\_type);  …  } |

|  |
| --- |
| （ofproto/ofproto.c）  // 其中，参数为网桥设置的 datapath\_type  static const struct ofproto\_class \*  ofproto\_class\_find\_\_(const char \*type)  {  size\_t i;  // 遍历已注册的 ofproto\_class  for (i = 0; i < n\_ofproto\_classes; i++) {  const struct ofproto\_class \*class = ofproto\_classes[i];  struct sset types;  bool found;  sset\_init(&types);  // 获取已注册的 ofproto\_class 支持的 datapath\_type 列表  class->enumerate\_types(&types);  // 检查是否包含了指定的 type，如果包含，则匹配  found = sset\_contains(&types, type);  sset\_destroy(&types);  if (found) {  return class;  }  }  VLOG\_WARN("unknown datapath type %s", type);  return NULL;  } |

所以这里涉及到的重点就是网桥和 ofproto\_class 的匹配，是通过 ofproto\_class 所支持的 datapath\_type 决定的，一个 ofproto\_class 可以支持多种 datapath type。这个逻辑与内核中 pci 驱动匹配非常的类似。

### ofproto\_class 中的函数说明

#### init()

使用 ofproto\_class 的第一步是注册，前面已经说过。第二步骤是调用其 init() 函数，在 ofproto\_init() 函数中：

|  |
| --- |
| （ofproto/ofproto.c）  void  ofproto\_init(const struct shash \*iface\_hints)  {  struct shash\_node \*node;  size\_t i;  ofproto\_class\_register(&ofproto\_dpif\_class);  /\* Make a local copy, since we don't own 'iface\_hints' elements. \*/  SHASH\_FOR\_EACH(node, iface\_hints) {  const struct iface\_hint \*orig\_hint = node->data;  struct iface\_hint \*new\_hint = xmalloc(sizeof \*new\_hint);  const char \*br\_type = ofproto\_normalize\_type(orig\_hint->br\_type);  new\_hint->br\_name = xstrdup(orig\_hint->br\_name);  new\_hint->br\_type = xstrdup(br\_type);  new\_hint->ofp\_port = orig\_hint->ofp\_port;  shash\_add(&init\_ofp\_ports, node->name, new\_hint);  }  for (i = 0; i < n\_ofproto\_classes; i++) {  ofproto\_classes[i]->init(&init\_ofp\_ports);  }  ofproto\_unixctl\_init();  } |

Init 的定义为：

|  |
| --- |
| （ofproto/ofproto-provider.h）  struct ofproto\_class {  …  void (\*init)(const struct shash \*iface\_hints);  …  } |

这里重点在参数 iface\_hints，这是一个哈希表，key 为 interface name，value 为 struct iface\_hint 结构，该结构中保存了一些用户指定的 interface 的信息，如通过 ofport\_request 指定的 openflow 的端口号，具体可参考 bridge\_init\_ofproto() 函数。

ofproto\_dpif\_class->init() 实现在：

|  |
| --- |
| （ofproto/ofproto-dpif.c）  static void  init(const struct shash \*iface\_hints)  {  struct shash\_node \*node;  /\* Make a local copy, since we don't own 'iface\_hints' elements. \*/  SHASH\_FOR\_EACH(node, iface\_hints) {  const struct iface\_hint \*orig\_hint = node->data;  struct iface\_hint \*new\_hint = xmalloc(sizeof \*new\_hint);  new\_hint->br\_name = xstrdup(orig\_hint->br\_name);  new\_hint->br\_type = xstrdup(orig\_hint->br\_type);  new\_hint->ofp\_port = orig\_hint->ofp\_port;  shash\_add(&init\_ofp\_ports, node->name, new\_hint);  }  ofproto\_unixctl\_init();  ofproto\_dpif\_trace\_init();  udpif\_init();  } |

#### alloc()

alloc() 用于分配 ofproto provider 自己的 struct ofproto 结构，在 ofproto\_create() 中调用：

|  |
| --- |
| (ofproto/ofproto.c)  int  ofproto\_create(const char \*datapath\_name, const char \*datapath\_type,  struct ofproto \*\*ofprotop)  OVS\_EXCLUDED(ofproto\_mutex)  {  …  ofproto = class->alloc();  …  } |

Alloc() 的定义为：

|  |
| --- |
| （ofproto/ofproto-provider.h）  struct ofproto\_class {  …  struct ofproto \*(\*alloc)(void);  …  } |

ofproto\_dpif\_class->alloc() 的实现在：

|  |
| --- |
| （ofproto/ofproto-dpif.c）  static struct ofproto \*  alloc(void)  {  // 分配 ofproto\_dpif 结构，返回其中的 ofproto 地址  struct ofproto\_dpif \*ofproto = xzalloc(sizeof \*ofproto);  return &ofproto->up;  } |

#### construct()

**bridge->ofproto 初始化**

bridge->ofproto 初始化在 bridge\_reconfigure() 函数中：

|  |
| --- |
| （vswitchd/bridge.c）  static void  bridge\_reconfigure(const struct ovsrec\_open\_vswitch \*ovs\_cfg)  {  …  /\* Finish pushing configuration changes to the ofproto layer:  \*  \* - Create ofprotos that are missing.  \*  \* - Add ports that are missing. \*/  HMAP\_FOR\_EACH\_SAFE (br, node, &all\_bridges) {  if (!br->ofproto) {  int error;  // 调用 ofproto\_create() 初始化 bridge->ofproto  error = ofproto\_create(br->name, br->type, &br->ofproto);  if (error) {  VLOG\_ERR("failed to create bridge %s: %s", br->name,  ovs\_strerror(error));  shash\_destroy(&br->wanted\_ports);  bridge\_destroy(br, true);  } else {  /\* Trigger storing datapath version. \*/  seq\_change(connectivity\_seq\_get());  }  }  }  …  } |

# 内核部分

## datapath

一个网桥对应一个内核态的 datapath 结构。

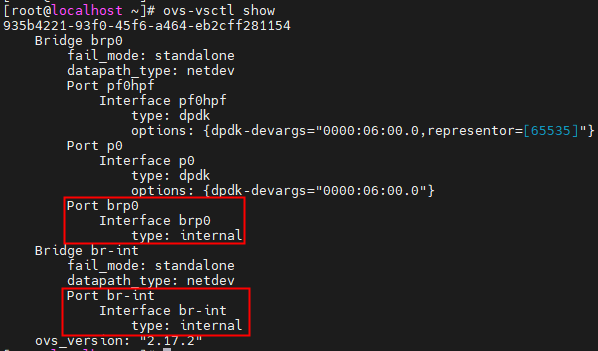
## vport

端口的类型定义在：

|  |
| --- |
| （linux/compat/include/linux/openvswitch.h）  enum ovs\_vport\_type {  OVS\_VPORT\_TYPE\_UNSPEC,  OVS\_VPORT\_TYPE\_NETDEV, /\* network device \*/  OVS\_VPORT\_TYPE\_INTERNAL, /\* network device implemented by datapath \*/  OVS\_VPORT\_TYPE\_GRE, /\* GRE tunnel. \*/  OVS\_VPORT\_TYPE\_VXLAN, /\* VXLAN tunnel. \*/  OVS\_VPORT\_TYPE\_GENEVE, /\* Geneve tunnel. \*/  OVS\_VPORT\_TYPE\_LISP = 105, /\* LISP tunnel \*/  OVS\_VPORT\_TYPE\_STT = 106, /\* STT tunnel \*/  \_\_OVS\_VPORT\_TYPE\_MAX  }; |

以下是各种类型端口的作用：

* OVS\_VPORT\_TYPE\_INTERNAL：每个OVS交换机有个可以用来处理数据报的本地端口，可以为这个网络设备配置 IP 地址。(比方在把eth0增加某个bridge的时候，它的IP地址就失效了。能够把IP地址赋给br，这就是internal port的地址)等等。每个网桥创建时会默认创建一个同名的 internal 端口，例如：



## OVS\_VPORT\_TYPE\_INTERNAL 相关逻辑

下面是 internal 类型的 vport 在内核模块的逻辑。

### 创建和删除

如前文所述，internal 类型的 port 是和网桥的生命周期绑定的，因此，其创建和删除是在 datapath（内核态表示网桥）的创建和删除中。

**创建**

ovs\_dp\_cmd\_new() 中（响应 OVS\_DP\_CMD\_NEW 的 netlink 消息）：

|  |
| --- |
| （datapath.c）  static int ovs\_dp\_cmd\_new(struct sk\_buff \*skb, struct genl\_info \*info)  {  …  /\* Set up our datapath device. \*/  // 创建 internal port  parms.name = nla\_data(a[OVS\_DP\_ATTR\_NAME]); // 名称和 datapath 相同  parms.type = OVS\_VPORT\_TYPE\_INTERNAL; // 类型是 internal  parms.options = NULL;  parms.dp = dp;  parms.port\_no = OVSP\_LOCAL; // 端口编号是特殊的 OVSP\_LOCAL  parms.upcall\_portids = a[OVS\_DP\_ATTR\_UPCALL\_PID];  …  vport = new\_vport(&parms); |

**删除**

ovs\_dp\_cmd\_del()->\_\_dp\_destroy() 中（响应 OVS\_DP\_CMD\_DEL 的 netlink 消息）：

|  |
| --- |
| （datapath.c）  /\* Called with ovs\_mutex. \*/  static void \_\_dp\_destroy(struct datapath \*dp)  {  int i;  // 先删除非 internal 的端口  for (i = 0; i < DP\_VPORT\_HASH\_BUCKETS; i++) {  struct vport \*vport;  struct hlist\_node \*n;  hlist\_for\_each\_entry\_safe(vport, n, &dp->ports[i], dp\_hash\_node)  if (vport->port\_no != OVSP\_LOCAL)  ovs\_dp\_detach\_port(vport);  }  list\_del\_rcu(&dp->list\_node);  /\* OVSP\_LOCAL is datapath internal port. We need to make sure that  \* all ports in datapath are destroyed first before freeing datapath.  \*/  // 删除 internal 端口  ovs\_dp\_detach\_port(ovs\_vport\_ovsl(dp, OVSP\_LOCAL));  /\* RCU destroy the flow table \*/  call\_rcu(&dp->rcu, destroy\_dp\_rcu);  } |