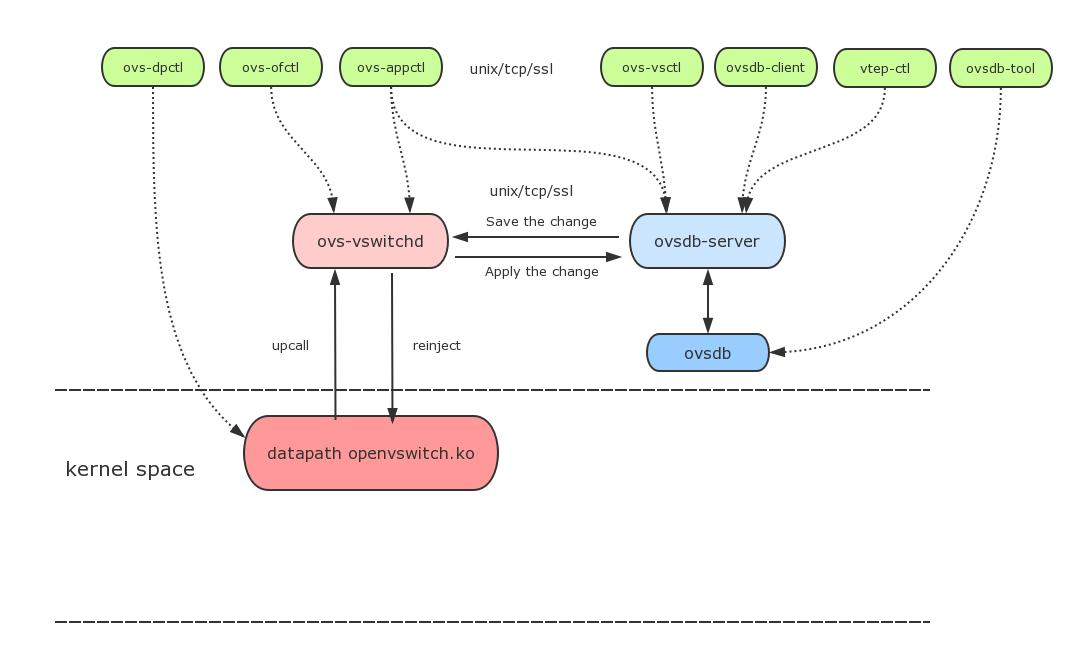
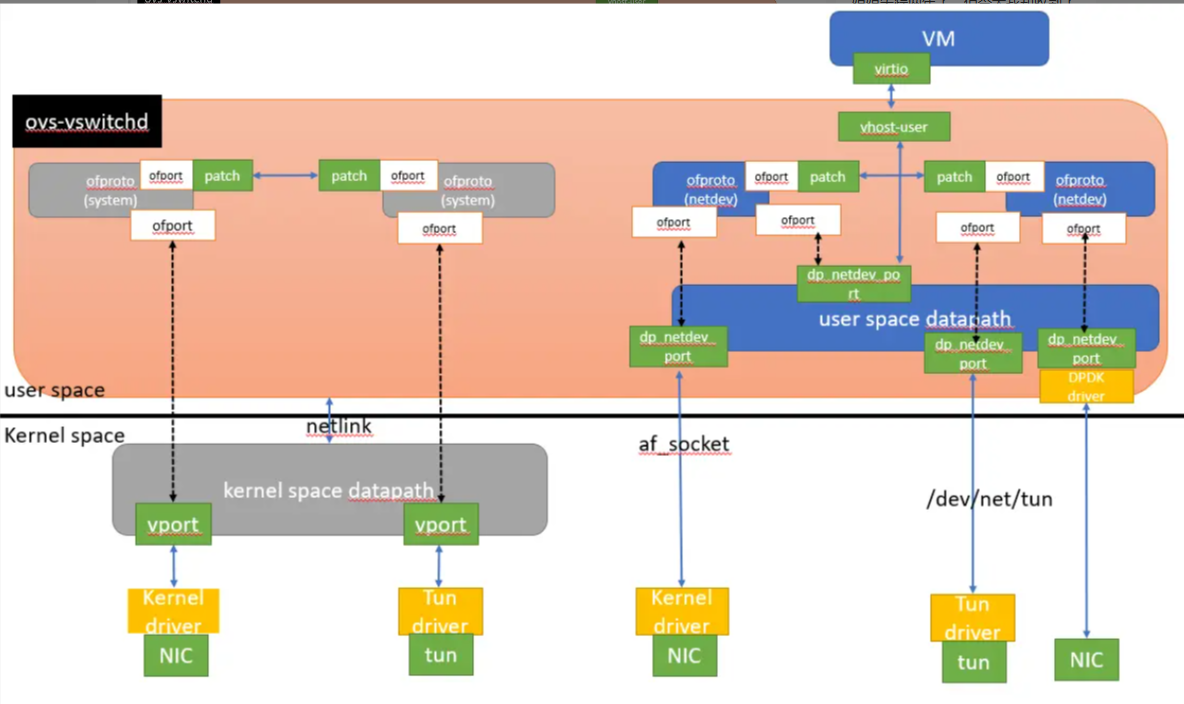
# ovs 的基本工作原理



# 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);  …  } |

## number\_handlers\_required()

指定 handler 线程的数目，参考 dpif\_number\_handlers\_required() 函数中的调用。

|  |
| --- |
| （lib/dpif-provider.h）  struct dpif\_class {  …  // 其中，n\_handlers 为返回值  bool (\*number\_handlers\_required)(struct dpif \*dpif, uint32\_t \*n\_handlers);  …  } |

dpif-netdev 没有实现这个函数，由 ovs 的配置指定。而 dpif-netlink 实现了，为 dpif\_netlink\_number\_handlers\_required()：

|  |
| --- |
| （lib/dpif-netlink.c）  static bool  dpif\_netlink\_number\_handlers\_required(struct dpif \*dpif\_, uint32\_t \*n\_handlers)  {  struct dpif\_netlink \*dpif = dpif\_netlink\_cast(dpif\_);  if (dpif\_netlink\_upcall\_per\_cpu(dpif)) {  \*n\_handlers = dpif\_netlink\_calculate\_n\_handlers();  return true;  }  return false;  } |

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);  }  } |

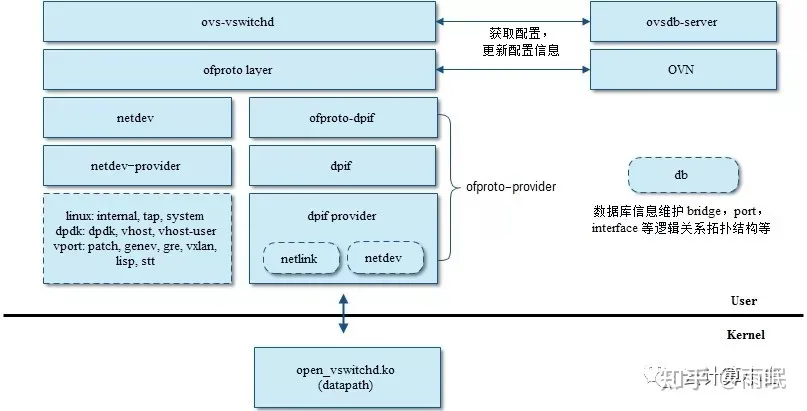
datapath\_create() 调用链条为：

* 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。

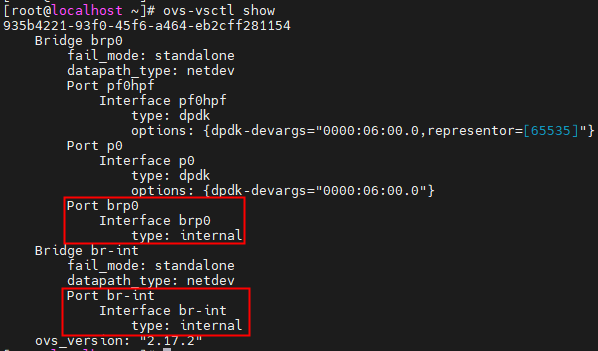
## vport

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);  } |

# Upcall

## ovs中handler和revalidator线程

参考：[ovs中handler和revalidator线程 - 简书 (jianshu.com)](https://www.jianshu.com/p/ab267654481e)



### handler和revalidator线程个数

线程个数由如下代码决定：

a. 如果配置了线程个数，则使用配置的值。使用如下命令配置，实时生效

ovs-vsctl --no-wait set Open\_vSwitch . other\_config:n-handler-threads=4

ovs-vsctl --no-wait set Open\_vSwitch . other\_config:n-revalidator-threads=4

b. 如果没有配置，则根据cpu个数来定。n\_revalidators 等于cpu个数除4后加1，n\_handlers 等于cpu个数减去n\_revalidators 个数。

上面的测试环境只有四个cpu，所以有两个revalidator线程和两个handler。

|  |
| --- |
| static void  bridge\_reconfigure(const struct ovsrec\_open\_vswitch \*ovs\_cfg)  ofproto\_set\_threads(  smap\_get\_int(&ovs\_cfg->other\_config, "n-handler-threads", 0),  smap\_get\_int(&ovs\_cfg->other\_config, "n-revalidator-threads", 0));  void  ofproto\_set\_threads(int n\_handlers\_, int n\_revalidators\_)  {  int threads = MAX(count\_cpu\_cores(), 2);  n\_revalidators = MAX(n\_revalidators\_, 0);  n\_handlers = MAX(n\_handlers\_, 0);  if (!n\_revalidators) {  n\_revalidators = n\_handlers  ? MAX(threads - (int) n\_handlers, 1)  : threads / 4 + 1;  }  if (!n\_handlers) {  n\_handlers = MAX(threads - (int) n\_revalidators, 1);  }  } |

### 线程的作用

#### handler 线程

我们知道，ovs包含三个流表：microflow，megaflow和openflow。其中openflow流表是由用户或者controller配置的，其他两个流表是报文触发创建，一条流的首包查找microflow，megaflow失败后，会走慢速路径查找openflow流表(流表规则为normal的话，就会变为mac学习，根据fdb转发)，然后将查找到的转发信息，再下发到microflow，megaflow，这样后面的报文就可以根据microflow，megaflow进行转发，这称为快速路径。这里的慢速路径的处理就是handler线程的工作内容。

有两点需要注意的是，一个是对于普通的ovs来说，microflow和megaflow存在于openvswitch.ko内核模块中，而对于ovs+dpdk来说，microflow和megaflow存在于用户态的ovs-vswitchd进程中。当然在ovs+dpdk模式下，也可以加载openvswitch.ko内核模块，这样两种模式可以共同存在，创建两种类型的桥。

另一个是handler线程只对普通ovs下，从内核upcall的消息进行处理。ovs+dpdk下，handler线程也存在，但是一直处于堵塞状态，实际上什么也没干。ovs+dpdk下的慢速路径的处理直接由收包线程执行，比如pmd线程。

#### revalidator线程

考虑下面这几种情况：

|  |
| --- |
| a. 比如已经通过handler线程下发了一条megaflow流表(可通过ovs-appctl dpctl/dump-flows进行查看)，如果一段时间内没有流量使用此流表，10s后就会被删除。  b. 又比如已经下发了一条megaflow流表，10s内将其对应的openflow流表删除，则这条megaflow流表也要立即被删除。  c. 再比如已经下发了一条megaflow流表，10s内修改其对应openflow流表的action后，则这条megaflow流表的action也要相应改变。 |

revalidator线程就是干上面的工作的，当然还有其他没考虑到的情况，总结来说就是revalidator线程用于megaflow流表的超时删除，响应openflow流表的改动，还有就是周期获取datapath流表的统计信息，一方面是为了确认流表是否被使用了，另一方面用来响应查看统计信息的命令(ovs-ofctl dump-flows br1)。

可以使用下面两条命令设置megaflow流表最大值(默认ofproto\_flow\_limit 20万)和流表超时时间(默认ofproto\_max\_idle 10s)

ovs-vsctl set Open\_vSwitch . other\_config:flow-limit=2

ovs-vsctl set Open\_vSwitch . other\_config:max-idle=1000000

revalidator线程在普通ovs和ovs+dpdk模式下都生效。

### 源码分析

#### handler线程

handler线程的主循环如下，只要exit\_latch不设置就一直存在。

|  |
| --- |
| static void \*  udpif\_upcall\_handler(void \*arg)  {  struct handler \*handler = arg;  struct udpif \*udpif = handler->udpif;  while (!latch\_is\_set(&handler->udpif->exit\_latch)) {  //recv\_upcalls用于接收upcall消息。只有kernel path提供了recv函数dpif\_netlink\_recv，所以说handler线程对dpdk path不生效。  //recv\_upcalls返回值大于0，说明已经处理过upcall消息，防止  //丢失后续的upcall消息，线程不能堵塞，需要再次调用recv\_upcalls  //尝试接收upcall消息，所以需要调用poll\_immediate\_wake将  //timeout\_when设置为最小值LLONG\_MIN，这样poll\_block调用poll函数就能立即返回，继续执行recv\_upcalls。  //recv\_upcalls返回值小于0，说明没有upcall消息，dpif\_recv\_wait调用等待upcall消息即可。  if (recv\_upcalls(handler)) {  poll\_immediate\_wake();  } else {  //等待upcall消息  dpif\_recv\_wait(udpif->dpif, handler->handler\_id);  //等待exit消息  latch\_wait(&udpif->exit\_latch);  }  //堵塞在poll函数上，超时时间timeout\_when为最大值，等待事件发生  poll\_block();  }  return NULL;  } |

主处理函数recv\_upcalls：

|  |
| --- |
| static size\_t  recv\_upcalls(struct handler \*handler)  {  struct udpif \*udpif = handler->udpif;  uint64\_t recv\_stubs[UPCALL\_MAX\_BATCH][512 / 8];  struct ofpbuf recv\_bufs[UPCALL\_MAX\_BATCH];  struct dpif\_upcall dupcalls[UPCALL\_MAX\_BATCH];  struct upcall upcalls[UPCALL\_MAX\_BATCH];  struct flow flows[UPCALL\_MAX\_BATCH];  size\_t n\_upcalls, i;  n\_upcalls = 0;  //每次最多处理50个upcall消息  while (n\_upcalls < UPCALL\_MAX\_BATCH) {  struct ofpbuf \*recv\_buf = &recv\_bufs[n\_upcalls];  struct dpif\_upcall \*dupcall = &dupcalls[n\_upcalls];  struct upcall \*upcall = &upcalls[n\_upcalls];  struct flow \*flow = &flows[n\_upcalls];  unsigned int mru;  int error;  ofpbuf\_use\_stub(recv\_buf, recv\_stubs[n\_upcalls],  sizeof recv\_stubs[n\_upcalls]);  //从datapath接收upcall消息存储在dupcall中  if (dpif\_recv(udpif->dpif, handler->handler\_id, dupcall, recv\_buf)) {  ofpbuf\_uninit(recv\_buf);  break;  }  //将dupcall消息中的key转换到flow中  if (odp\_flow\_key\_to\_flow(dupcall->key, dupcall->key\_len, flow)  == ODP\_FIT\_ERROR) {  goto free\_dupcall;  }  if (dupcall->mru) {  mru = nl\_attr\_get\_u16(dupcall->mru);  } else {  mru = 0;  }  error = upcall\_receive(upcall, udpif->backer, &dupcall->packet,  dupcall->type, dupcall->userdata, flow, mru,  &dupcall->ufid, PMD\_ID\_NULL);  if (error) {  if (error == ENODEV) {  /\* Received packet on datapath port for which we couldn't  \* associate an ofproto. This can happen if a port is removed  \* while traffic is being received. Print a rate-limited  \* message in case it happens frequently. \*/  dpif\_flow\_put(udpif->dpif, DPIF\_FP\_CREATE, dupcall->key,  dupcall->key\_len, NULL, 0, NULL, 0,  &dupcall->ufid, PMD\_ID\_NULL, NULL);  VLOG\_INFO\_RL(&rl, "received packet on unassociated datapath "  "port %"PRIu32, flow->in\_port.odp\_port);  }  goto free\_dupcall;  }  upcall->key = dupcall->key;  upcall->key\_len = dupcall->key\_len;  upcall->ufid = &dupcall->ufid;  upcall->out\_tun\_key = dupcall->out\_tun\_key;  upcall->actions = dupcall->actions;  pkt\_metadata\_from\_flow(&dupcall->packet.md, flow);  flow\_extract(&dupcall->packet, flow);  //开始处理，根据flow信息查找openflow流表  error = process\_upcall(udpif, upcall,  &upcall->odp\_actions, &upcall->wc);  if (error) {  goto cleanup;  }  n\_upcalls++;  continue;  cleanup:  upcall\_uninit(upcall);  free\_dupcall:  dp\_packet\_uninit(&dupcall->packet);  ofpbuf\_uninit(recv\_buf);  }  if (n\_upcalls) {  //如果查找openflow成功，则将相关信息安装到datapath中，后续的报文根据datapath中的流表转发  handle\_upcalls(handler->udpif, upcalls, n\_upcalls);  for (i = 0; i < n\_upcalls; i++) {  dp\_packet\_uninit(&dupcalls[i].packet);  ofpbuf\_uninit(&recv\_bufs[i]);  upcall\_uninit(&upcalls[i]);  }  }  return n\_upcalls;  } |

upcall消息有两种类型DPIF\_UC\_MISS和DPIF\_UC\_ACTION，前者表示查找流表失败，需要走慢速路径查找openflow流表，后者表示流表的action为将报文上送。这里只看一下DPIF\_UC\_MISS的情况。

|  |
| --- |
| static int  process\_upcall(struct udpif \*udpif, struct upcall \*upcall,  struct ofpbuf \*odp\_actions, struct flow\_wildcards \*wc)  {  const struct nlattr \*userdata = upcall->userdata;  const struct dp\_packet \*packet = upcall->packet;  const struct flow \*flow = upcall->flow;  size\_t actions\_len = 0;  enum upcall\_type upcall\_type = classify\_upcall(upcall->type, userdata);  //upcall类型为 MISS\_UPCALL  switch (upcall\_type) {  case MISS\_UPCALL:  upcall\_xlate(udpif, upcall, odp\_actions, wc);  return 0;  ...  }  }  static void  upcall\_xlate(struct udpif \*udpif, struct upcall \*upcall,  struct ofpbuf \*odp\_actions, struct flow\_wildcards \*wc)  {  struct dpif\_flow\_stats stats;  struct xlate\_in xin;  stats.n\_packets = 1;  stats.n\_bytes = dp\_packet\_size(upcall->packet);  stats.used = time\_msec();  stats.tcp\_flags = ntohs(upcall->flow->tcp\_flags);  xlate\_in\_init(&xin, upcall->ofproto,  ofproto\_dpif\_get\_tables\_version(upcall->ofproto),  upcall->flow, upcall->in\_port, NULL,  stats.tcp\_flags, upcall->packet, wc, odp\_actions);  if (upcall->type == DPIF\_UC\_MISS) {  xin.resubmit\_stats = &stats;  if (xin.frozen\_state) {  /\* We may install a datapath flow only if we get a reference to the  \* recirculation context (otherwise we could have recirculation  \* upcalls using recirculation ID for which no context can be  \* found). We may still execute the flow's actions even if we  \* don't install the flow. \*/  upcall->recirc = recirc\_id\_node\_from\_state(xin.frozen\_state);  upcall->have\_recirc\_ref = recirc\_id\_node\_try\_ref\_rcu(upcall->recirc);  }  } else {  /\* For non-miss upcalls, we are either executing actions (one of which  \* is an userspace action) for an upcall, in which case the stats have  \* already been taken care of, or there's a flow in the datapath which  \* this packet was accounted to. Presumably the revalidators will deal  \* with pushing its stats eventually. \*/  }  upcall->dump\_seq = seq\_read(udpif->dump\_seq);  upcall->reval\_seq = seq\_read(udpif->reval\_seq);  //关键函数，查找openflow流表  xlate\_actions(&xin, &upcall->xout);  if (wc) {  /\* Convert the input port wildcard from OFP to ODP format. There's no  \* real way to do this for arbitrary bitmasks since the numbering spaces  \* aren't the same. However, flow translation always exact matches the  \* whole thing, so we can do the same here. \*/  WC\_MASK\_FIELD(wc, in\_port.odp\_port);  }  upcall->xout\_initialized = true;  if (!upcall->xout.slow) {  ofpbuf\_use\_const(&upcall->put\_actions,  odp\_actions->data, odp\_actions->size);  } else {  uint32\_t smid = upcall->ofproto->up.slowpath\_meter\_id;  uint32\_t cmid = upcall->ofproto->up.controller\_meter\_id;  /\* upcall->put\_actions already initialized by upcall\_receive(). \*/  compose\_slow\_path(udpif, &upcall->xout, upcall->flow,  upcall->flow->in\_port.odp\_port,  &upcall->put\_actions, smid, cmid);  }  /\* This function is also called for slow-pathed flows. As we are only  \* going to create new datapath flows for actual datapath misses, there is  \* no point in creating a ukey otherwise. \*/  if (upcall->type == DPIF\_UC\_MISS) {  upcall->ukey = ukey\_create\_from\_upcall(upcall, wc);  }  } |

|  |
| --- |
| static void  handle\_upcalls(struct udpif \*udpif, struct upcall \*upcalls,  size\_t n\_upcalls)  {  //ops 为upcall个数的两倍，其中一个type为DPIF\_OP\_FLOW\_PUT，用于安装flow，  //另一个为DPIF\_OP\_EXECUTE，用于将upcall的报文安装action处理掉  struct dpif\_op \*opsp[UPCALL\_MAX\_BATCH \* 2];  struct ukey\_op ops[UPCALL\_MAX\_BATCH \* 2];  size\_t n\_ops, n\_opsp, i;  /\* Handle the packets individually in order of arrival.  \*  \* - For SLOW\_CFM, SLOW\_LACP, SLOW\_STP, SLOW\_BFD, and SLOW\_LLDP,  \* translation is what processes received packets for these  \* protocols.  \*  \* - For SLOW\_CONTROLLER, translation sends the packet to the OpenFlow  \* controller.  \*  \* - For SLOW\_ACTION, translation executes the actions directly.  \*  \* The loop fills 'ops' with an array of operations to execute in the  \* datapath. \*/  n\_ops = 0;  for (i = 0; i < n\_upcalls; i++) {  struct upcall \*upcall = &upcalls[i];  const struct dp\_packet \*packet = upcall->packet;  struct ukey\_op \*op;  //should\_install\_flow决定是否install flow到datapath，比如是否超了flow\_limit  if (should\_install\_flow(udpif, upcall)) {  struct udpif\_key \*ukey = upcall->ukey;  if (ukey\_install(udpif, ukey)) {  upcall->ukey\_persists = true;  put\_op\_init(&ops[n\_ops++], ukey, DPIF\_FP\_CREATE);  }  }  if (upcall->odp\_actions.size) {  op = &ops[n\_ops++];  op->ukey = NULL;  op->dop.type = DPIF\_OP\_EXECUTE;  op->dop.u.execute.packet = CONST\_CAST(struct dp\_packet \*, packet);  op->dop.u.execute.flow = upcall->flow;  odp\_key\_to\_dp\_packet(upcall->key, upcall->key\_len,  op->dop.u.execute.packet);  op->dop.u.execute.actions = upcall->odp\_actions.data;  op->dop.u.execute.actions\_len = upcall->odp\_actions.size;  op->dop.u.execute.needs\_help = (upcall->xout.slow & SLOW\_ACTION) != 0;  op->dop.u.execute.probe = false;  op->dop.u.execute.mtu = upcall->mru;  }  }  /\* Execute batch. \*/  n\_opsp = 0;  for (i = 0; i < n\_ops; i++) {  opsp[n\_opsp++] = &ops[i].dop;  }  //和datapath交互，将flow安装到datapath，并处理upcall报文  dpif\_operate(udpif->dpif, opsp, n\_opsp);  //设置 ukey 状态，ukey会在revalidator线程中使用  for (i = 0; i < n\_ops; i++) {  struct udpif\_key \*ukey = ops[i].ukey;  if (ukey) {  ovs\_mutex\_lock(&ukey->mutex);  if (ops[i].dop.error) {  transition\_ukey(ukey, UKEY\_EVICTED);  } else if (ukey->state < UKEY\_OPERATIONAL) {  transition\_ukey(ukey, UKEY\_OPERATIONAL);  }  ovs\_mutex\_unlock(&ukey->mutex);  }  }  } |

#### revalidator线程

revalidator线程在以下几种情况下才会将poll中从堵塞变为运行

a. need\_revalidate被设置，说明网桥配置或者流表发生变化，需要重新计算flow

b. pause\_latch被设置，说明此线程需要暂时停止运行

c. exit\_latch被设置，说明需要将此线程退出，比如将线程数量改小后，会有一部分线程需要退出

d. timeout 超时后运行，最小值为500ms

|  |
| --- |
| static void \*  udpif\_revalidator(void \*arg)  {  /\* Used by all revalidators. \*/  struct revalidator \*revalidator = arg;  struct udpif \*udpif = revalidator->udpif;  //第一个revalidator线程作为leader  bool leader = revalidator == &udpif->revalidators[0];  /\* Used only by the leader. \*/  long long int start\_time = 0;  uint64\_t last\_reval\_seq = 0;  size\_t n\_flows = 0;  revalidator->id = ovsthread\_id\_self();  for (;;) {  if (leader) {  uint64\_t reval\_seq;  recirc\_run(); /\* Recirculation cleanup. \*/  //获取 reval\_seq，只有 need\_revalidate 变为true后，reval\_seq才会加1，进而唤醒revalidator线程  reval\_seq = seq\_read(udpif->reval\_seq);  last\_reval\_seq = reval\_seq;  //获取datapath中flow个数  n\_flows = udpif\_get\_n\_flows(udpif);  udpif->max\_n\_flows = MAX(n\_flows, udpif->max\_n\_flows);  udpif->avg\_n\_flows = (udpif->avg\_n\_flows + n\_flows) / 2;  /\* Only the leader checks the pause latch to prevent a race where  \* some threads think it's false and proceed to block on  \* reval\_barrier and others think it's true and block indefinitely  \* on the pause\_barrier \*/  //在dp\_purge\_cb清理所有ukey时，会设置pause\_latch，暂停revalidator线程执行  udpif->pause = latch\_is\_set(&udpif->pause\_latch);  /\* Only the leader checks the exit latch to prevent a race where  \* some threads think it's true and exit and others think it's  \* false and block indefinitely on the reval\_barrier \*/  //如果不需要revalidator线程了，会在udpif\_stop\_threads设置exit\_latch，停止此线程  udpif->reval\_exit = latch\_is\_set(&udpif->exit\_latch);  start\_time = time\_msec();  if (!udpif->reval\_exit) {  bool terse\_dump;  //如果revalidator线程没有被停止，则创建dump，用来dump datapath中的flow  terse\_dump = udpif\_use\_ufid(udpif);  udpif->dump = dpif\_flow\_dump\_create(udpif->dpif, terse\_dump,  NULL);  }  }  /\* Wait for the leader to start the flow dump. \*/  //等待所有revalidator线程都运行到此处，才会开始向下执行  ovs\_barrier\_block(&udpif->reval\_barrier);  if (udpif->pause) {  revalidator\_pause(revalidator);  }  //如果线程停止了，跳出循环  if (udpif->reval\_exit) {  break;  }  //关键函数，下面会单独分析  revalidate(revalidator);  /\* Wait for all flows to have been dumped before we garbage collect. \*/  ovs\_barrier\_block(&udpif->reval\_barrier);  //删除状态为UKEY\_EVICTED的ukey  revalidator\_sweep(revalidator);  /\* Wait for all revalidators to finish garbage collection. \*/  ovs\_barrier\_block(&udpif->reval\_barrier);  if (leader) {  unsigned int flow\_limit;  long long int duration;  atomic\_read\_relaxed(&udpif->flow\_limit, &flow\_limit);  dpif\_flow\_dump\_destroy(udpif->dump);  seq\_change(udpif->dump\_seq);  duration = MAX(time\_msec() - start\_time, 1);  udpif->dump\_duration = duration;  //flow比较多的情况下，最大限制flow\_limit会发生变化  //如果dump时间大于2000ms，则flow\_limit需要除duration对于1000ms的倍数  //如果dump时间大于1300ms，小于2000ms，则flow\_limit变为当前值的四分之三  //如果dump时间小于1000ms，并且当前flow个数大于2000，并且xxx，则flow\_limit增加1000  if (duration > 2000) {  flow\_limit /= duration / 1000;  } else if (duration > 1300) {  flow\_limit = flow\_limit \* 3 / 4;  } else if (duration < 1000 && n\_flows > 2000  && flow\_limit < n\_flows \* 1000 / duration) {  flow\_limit += 1000;  }  //flow\_limit最大值不能超过ofproto\_flow\_limit，即20万  flow\_limit = MIN(ofproto\_flow\_limit, MAX(flow\_limit, 1000));  //保存flow\_limit，handler线程安装flow时会判断是否超过此值  atomic\_store\_relaxed(&udpif->flow\_limit, flow\_limit);  if (duration > 2000) {  VLOG\_INFO("Spent an unreasonably long %lldms dumping flows",  duration);  }  //用于计算poll函数超时时间  poll\_timer\_wait\_until(start\_time + MIN(ofproto\_max\_idle, 500));  //将seq插入poll\_loop节点，等待 seq\_change  seq\_wait(udpif->reval\_seq, last\_reval\_seq);  //将exit\_latch插入poll\_loop节点  latch\_wait(&udpif->exit\_latch);  //将pause\_latch插入poll\_loop节点  latch\_wait(&udpif->pause\_latch);  //调用poll函数，获取poll\_loop节点上发生的事件，如果有事件发生，则继续执行，否则就等待超时后再继续执行。  poll\_block();  if (!latch\_is\_set(&udpif->pause\_latch) &&  !latch\_is\_set(&udpif->exit\_latch)) {  long long int now = time\_msec();  /\* Block again if we are woken up within 5ms of the last start  \* time. \*/  start\_time += 5;  if (now < start\_time) {  poll\_timer\_wait\_until(start\_time);  latch\_wait(&udpif->exit\_latch);  latch\_wait(&udpif->pause\_latch);  poll\_block();  }  }  }  }  return NULL;  } |

|  |
| --- |
| static void  revalidate(struct revalidator \*revalidator)  {  uint64\_t odp\_actions\_stub[1024 / 8];  struct ofpbuf odp\_actions = OFPBUF\_STUB\_INITIALIZER(odp\_actions\_stub);  struct udpif \*udpif = revalidator->udpif;  struct dpif\_flow\_dump\_thread \*dump\_thread;  uint64\_t dump\_seq, reval\_seq;  unsigned int flow\_limit;  dump\_seq = seq\_read(udpif->dump\_seq);  reval\_seq = seq\_read(udpif->reval\_seq);  atomic\_read\_relaxed(&udpif->flow\_limit, &flow\_limit);  dump\_thread = dpif\_flow\_dump\_thread\_create(udpif->dump);  for (;;) {  struct ukey\_op ops[REVALIDATE\_MAX\_BATCH];  int n\_ops = 0;  struct dpif\_flow flows[REVALIDATE\_MAX\_BATCH];  const struct dpif\_flow \*f;  int n\_dumped;  long long int max\_idle;  long long int now;  size\_t n\_dp\_flows;  bool kill\_them\_all;  //从datapath dump flow信息  n\_dumped = dpif\_flow\_dump\_next(dump\_thread, flows, ARRAY\_SIZE(flows));  if (!n\_dumped) {  break;  }  now = time\_msec();  /\* In normal operation we want to keep flows around until they have  \* been idle for 'ofproto\_max\_idle' milliseconds. However:  \*  \* - If the number of datapath flows climbs above 'flow\_limit',  \* drop that down to 100 ms to try to bring the flows down to  \* the limit.  \*  \* - If the number of datapath flows climbs above twice  \* 'flow\_limit', delete all the datapath flows as an emergency  \* measure. (We reassess this condition for the next batch of  \* datapath flows, so we will recover before all the flows are  \* gone.) \*/  //再次获取datapath中flow个数  n\_dp\_flows = udpif\_get\_n\_flows(udpif);  //如果当前flow个数超过flow\_limit的两倍，则删除获取的所有flow  kill\_them\_all = n\_dp\_flows > flow\_limit \* 2;  //如果当前flow个数超过 flow\_limit，则max\_idle为100ms，否则为默认的10s.  //max\_idle为超时时间，如果在超时时间内，此flow没有被使用，则超时后就把flow删除。默认10s删除一次。  //如果flow个数超过最大限制flow\_limit了，超时时间修改为100ms，需要更快的删除。  max\_idle = n\_dp\_flows > flow\_limit ? 100 : ofproto\_max\_idle;  for (f = flows; f < &flows[n\_dumped]; f++) {  long long int used = f->stats.used;  struct recirc\_refs recircs = RECIRC\_REFS\_EMPTY\_INITIALIZER;  enum reval\_result result;  struct udpif\_key \*ukey;  bool already\_dumped;  int error;  //获取锁失败，说明正在被其他revalidate线程处理，继续处理下一个  if (ukey\_acquire(udpif, f, &ukey, &error)) {  if (error == EBUSY) {  /\* Another thread is processing this flow, so don't bother  \* processing it.\*/  COVERAGE\_INC(upcall\_ukey\_contention);  } else {  log\_unexpected\_flow(f, error);  if (error != ENOENT) {  delete\_op\_init\_\_(udpif, &ops[n\_ops++], f);  }  }  continue;  }  //dump\_seq 相等，说明已经被dump过，或者刚创建ukey，继续处理下一个  already\_dumped = ukey->dump\_seq == dump\_seq;  if (already\_dumped) {  /\* The flow has already been handled during this flow dump  \* operation. Skip it. \*/  if (ukey->xcache) {  COVERAGE\_INC(dumped\_duplicate\_flow);  } else {  COVERAGE\_INC(dumped\_new\_flow);  }  ovs\_mutex\_unlock(&ukey->mutex);  continue;  }  if (ukey->state <= UKEY\_OPERATIONAL) {  /\* The flow is now confirmed to be in the datapath. \*/  transition\_ukey(ukey, UKEY\_OPERATIONAL);  } else {  VLOG\_INFO("Unexpected ukey transition from state %d "  "(last transitioned from thread %u at %s)",  ukey->state, ukey->state\_thread, ukey->state\_where);  ovs\_mutex\_unlock(&ukey->mutex);  continue;  }  if (!used) {  used = ukey->created;  }  //used 表示flow最近被用的时间，如果超了max\_idle这么长时间不被使用就会被删除。  //比如只创建了一条flow，max\_idle就是默认的10s，如果10s内此flow没有流量，则会被删除。  if (kill\_them\_all || (used && used < now - max\_idle)) {  result = UKEY\_DELETE;  } else {  //验证ukey,比如只创建了一条flow，10s内openflow流表发生改变后，需要验证megaflow中flow是否需要改变，  //比如同一条流的action发生改变。  //udpif->reval\_seq和ukey->reval\_seq不一致，说明openflow流表发生变化  result = revalidate\_ukey(udpif, ukey, &f->stats, &odp\_actions,  reval\_seq, &recircs);  }  ukey->dump\_seq = dump\_seq;  //如果结果不是keep，则需要根据结果(删除/修改)初始化ops  if (result != UKEY\_KEEP) {  /\* Takes ownership of 'recircs'. \*/  reval\_op\_init(&ops[n\_ops++], result, udpif, ukey, &recircs,  &odp\_actions);  }  ovs\_mutex\_unlock(&ukey->mutex);  }  //和datapath交互，删除或者修改flow  if (n\_ops) {  /\* Push datapath ops but defer ukey deletion to 'sweep' phase. \*/  push\_dp\_ops(udpif, ops, n\_ops);  }  ovsrcu\_quiesce();  }  dpif\_flow\_dump\_thread\_destroy(dump\_thread);  ofpbuf\_uninit(&odp\_actions);  } |

revalidate\_ukey用来验证openflow流表是否发生改变，方法是使用datapath获取的flow信息重新查找openflow流表，根据结果判断是否和datapath的action是否一致，如果不一致说明说明需要修改或者删除datapath流表。

|  |
| --- |
| /\* Verifies that the datapath actions of 'ukey' are still correct, and pushes  \* 'stats' for it.  \*  \* Returns a recommended action for 'ukey', options include:  \* UKEY\_DELETE The ukey should be deleted.  \* UKEY\_KEEP The ukey is fine as is.  \* UKEY\_MODIFY The ukey's actions should be changed but is otherwise  \* fine. Callers should change the actions to those found  \* in the caller supplied 'odp\_actions' buffer. The  \* recirculation references can be found in 'recircs' and  \* must be handled by the caller.  \*  \* If the result is UKEY\_MODIFY, then references to all recirc\_ids used by the  \* new flow will be held within 'recircs' (which may be none).  \*  \* The caller is responsible for both initializing 'recircs' prior this call,  \* and ensuring any references are eventually freed.  \*/  static enum reval\_result  revalidate\_ukey(struct udpif \*udpif, struct udpif\_key \*ukey,  const struct dpif\_flow\_stats \*stats,  struct ofpbuf \*odp\_actions, uint64\_t reval\_seq,  struct recirc\_refs \*recircs)  OVS\_REQUIRES(ukey->mutex)  {  //最关键的判断，只有流表发生改变就会设置need\_revalidate  bool need\_revalidate = ukey->reval\_seq != reval\_seq;  enum reval\_result result = UKEY\_DELETE;  struct dpif\_flow\_stats push;  ofpbuf\_clear(odp\_actions);  push.used = stats->used;  push.tcp\_flags = stats->tcp\_flags;  push.n\_packets = (stats->n\_packets > ukey->stats.n\_packets  ? stats->n\_packets - ukey->stats.n\_packets  : 0);  push.n\_bytes = (stats->n\_bytes > ukey->stats.n\_bytes  ? stats->n\_bytes - ukey->stats.n\_bytes  : 0);  //need\_revalidate为true，说明openflow流表发生变化,需要验证mask/action是否改变  if (need\_revalidate) {  if (should\_revalidate(udpif, push.n\_packets, ukey->stats.used)) {  if (!ukey->xcache) {  ukey->xcache = xlate\_cache\_new();  } else {  xlate\_cache\_clear(ukey->xcache);  }  result = revalidate\_ukey\_\_(udpif, ukey, push.tcp\_flags,  odp\_actions, recircs, ukey->xcache);  } /\* else delete; too expensive to revalidate \*/  } else if (!push.n\_packets || ukey->xcache  || !populate\_xcache(udpif, ukey, push.tcp\_flags)) {  result = UKEY\_KEEP;  }  /\* Stats for deleted flows will be attributed upon flow deletion. Skip. \*/  if (result != UKEY\_DELETE) {  xlate\_push\_stats(ukey->xcache, &push);  ukey->stats = \*stats;  ukey->reval\_seq = reval\_seq;  }  return result;  } |

配置流表后，会将 ofproto->backer->need\_revalidate 设置为 REV\_FLOW\_TABLE，这样在 type\_run 中就可以将reval\_seq加1，唤醒revalidator线程进行处理(修改datapath流表)

|  |
| --- |
| handle\_flow\_mod\_\_  ofproto\_bump\_tables\_version(ofproto);  ++ofproto->tables\_version;  //set\_tables\_version  ofproto->ofproto\_class->set\_tables\_version(ofproto, ofproto->tables\_version);  struct ofproto\_dpif \*ofproto = ofproto\_dpif\_cast(ofproto\_);  /\* Use memory\_order\_release to signify that any prior memory accesses can  \* not be reordered to happen after this atomic store. This makes sure the  \* new version is properly set up when the readers can read this 'version'  \* value. \*/  atomic\_store\_explicit(&ofproto->tables\_version, version,  memory\_order\_release);  /\* 'need\_revalidate' can be reordered to happen before the atomic\_store  \* above, but it does not matter as this variable is not accessed by other  \* threads. \*/  ofproto->backer->need\_revalidate = REV\_FLOW\_TABLE;  static int  type\_run(const char \*type)  if (backer->need\_revalidate) {  udpif\_revalidate(backer->udpif);  //seq加1，并且唤醒正在等待seq的线程  seq\_change(udpif->reval\_seq);  } |

## Upcall 的流程

### 1 handler 线程的启动

Handler 线程和 revalidator 线程的作用前面一节已经说过了，这里不再赘述。

在 open\_dpif\_backer() 中，会调用 udpif\_set\_threads() 启动 handler 和 revalidator 线程：

|  |
| --- |
| （ofproto/ofproto-dpif.c）  static int  open\_dpif\_backer(const char \*type, struct dpif\_backer \*\*backerp)  {  …  if (backer->recv\_set\_enable) {  udpif\_set\_threads(backer->udpif, n\_handlers, n\_revalidators);  }  …  } |

udpif\_set\_threads()：

|  |
| --- |
| （ofproto/ofproto-dpif-upcall.c）  // 参数中，n\_handlers\_ 为配置的 handler 线程的数目  // n\_revalidators\_ 为配置的 revalidator 线程的数目  void  udpif\_set\_threads(struct udpif \*udpif, uint32\_t n\_handlers\_,  uint32\_t n\_revalidators\_)  {  …  // 调用 dpif\_class->number\_handlers\_required() 函数看对应的 dpif\_class 是否指定了 handler 线程的数目  if (dpif\_number\_handlers\_required(udpif->dpif, &n\_handlers\_requested)) {  forced = true;  if (!n\_revalidators\_) {  n\_revalidators\_requested = n\_handlers\_requested / 4 + 1;  } else {  n\_revalidators\_requested = n\_revalidators\_;  }  } else {  …  // 检查是否需要调整线程数目  // 需要调整时，先调用 udpif\_stop\_threads() 把当前线程停掉  // 停掉后 udpif->n\_handlers 和 udpif->revalidators 都为 0  if (udpif->n\_handlers != n\_handlers\_requested  || udpif->n\_revalidators != n\_revalidators\_requested) {  …  udpif\_stop\_threads(udpif, true);  }  // 启动新线程  if (!udpif->handlers && !udpif->revalidators) {  VLOG\_INFO("Starting %u threads", n\_handlers\_requested +  n\_revalidators\_requested);  int error;  // 调用 dpif\_class->handlers\_set() 通知其 handler 变化  error = dpif\_handlers\_set(udpif->dpif, n\_handlers\_requested);  if (error) {  VLOG\_ERR("failed to configure handlers in dpif %s: %s",  dpif\_name(udpif->dpif), ovs\_strerror(error));  return;  }  // 启动 handler 和 revalidator 线程  udpif\_start\_threads(udpif, n\_handlers\_requested,  n\_revalidators\_requested);  }  } |

在其中判断线程数由变化时，先调用 udpif\_stop\_threads() 停止老线程，然后调用 udpif\_start\_threads() 启动新线程。

线程的启动在 udpif\_start\_threads() 中，调用 ovs\_thread\_create() 创建线程：

|  |
| --- |
| （ofproto/ofproto-dpif-upcall.c）  static void  udpif\_start\_threads(struct udpif \*udpif, uint32\_t n\_handlers\_,  uint32\_t n\_revalidators\_)  {  …  for (size\_t i = 0; i < udpif->n\_handlers; i++) {  struct handler \*handler = &udpif->handlers[i];    handler->udpif = udpif;  handler->handler\_id = i;  handler->thread = ovs\_thread\_create(  "handler", udpif\_upcall\_handler, handler);  }  …  for (size\_t i = 0; i < udpif->n\_revalidators; i++) {  struct revalidator \*revalidator = &udpif->revalidators[i];    revalidator->udpif = udpif;  revalidator->thread = ovs\_thread\_create(  "revalidator", udpif\_revalidator, revalidator);  }  …  } |

这里可以看到，hander 线程运行的是 udpif\_upcall\_handler() 函数，revalidator 线程运行的是 udpif\_revalidator() 函数。

可以看到，udpif->handlers 数组中保存了所有的 handler 线程信息。

以上是 handler 线程启动的大致流程，对于不同的 dpif provider，是有一些地方是特殊的，我们跳过了，下面以 dpif-netlink 看下跳过的这些地方的特殊逻辑。

#### 1 handler 线程的数目（dpif\_number\_handlers\_required）

在 udpif\_set\_threads() 中调用了 dpif\_number\_handlers\_required() 获取 dpif\_class 指定的 handler 线程的数目，其实际调用了 dpif\_class->number\_handlers\_required()。对于 dpif-netlink，为 dpif\_netlink\_number\_handlers\_required()：

|  |
| --- |
| （lib/dpif-netlink.c）  static bool  dpif\_netlink\_number\_handlers\_required(struct dpif \*dpif\_, uint32\_t \*n\_handlers)  {  struct dpif\_netlink \*dpif = dpif\_netlink\_cast(dpif\_);  if (dpif\_netlink\_upcall\_per\_cpu(dpif)) {  \*n\_handlers = dpif\_netlink\_calculate\_n\_handlers();  return true;  }  return false;  } |

可以看到，对于 per-cpu 模式的 dispatch 方式，是由 dpif\_netlink\_calculate\_n\_handlers() 重新计算了 handler 线程的数目的。

#### 2 dpif\_class->handlers\_set()(dpif\_handlers\_set)

在调用 udpif\_start\_threads() 创建 handler 线程之前，还调用了 dpif\_class->handlers\_set()。这里就涉及到 dpif\_netlink 中也有一些自定义的 handler 线程的信息了。如前面所述，udpif 中使用 struct handler 保存 handler 线程的信息，dpif\_netlink 中定义了 struct dpif\_handler 保存 handler 线程的信息。

|  |
| --- |
| （lib/dpif-netlink.c）  struct dpif\_netlink {  …  struct dpif\_handler \*handlers; // handler 线程数组  uint32\_t n\_handlers; /\* Num of upcall handlers. \*/  } |

dpif\_netlink\_handlers\_set()：

|  |
| --- |
| （lib/dpif-netlink.c）  static int  dpif\_netlink\_handlers\_set(struct dpif \*dpif\_, uint32\_t n\_handlers)  {  struct dpif\_netlink \*dpif = dpif\_netlink\_cast(dpif\_);  int error = 0;  #ifdef \_WIN32  /\* Multiple upcall handlers will be supported once kernel datapath supports  \* it. \*/  if (n\_handlers > 1) {  return error;  }  #endif  fat\_rwlock\_wrlock(&dpif->upcall\_lock);  // 如果 dpif->handlers 不为空，调用 dpif\_netlink\_refresh\_handlers\_cpu\_dispatch() 刷新handler 信息  if (dpif->handlers) {  if (dpif\_netlink\_upcall\_per\_cpu(dpif)) {  error = dpif\_netlink\_refresh\_handlers\_cpu\_dispatch(dpif);  } else {  error = dpif\_netlink\_refresh\_handlers\_vport\_dispatch(dpif,  n\_handlers);  }  }  fat\_rwlock\_unlock(&dpif->upcall\_lock);  return error;  } |

dpif\_netlink\_refresh\_handlers\_cpu\_dispatch() 和 dpif\_netlink\_refresh\_handlers\_vport\_dispatch() 是否则创建和更新 handler 信息的函数，在 dpif\_netlink\_recv\_set() 中也会调用。

下面以 dpif\_netlink\_refresh\_handlers\_cpu\_dispatch() 为例：

|  |
| --- |
| （lib/dpif-netlink.c）  static int  dpif\_netlink\_refresh\_handlers\_cpu\_dispatch(struct dpif\_netlink \*dpif)  OVS\_REQ\_WRLOCK(dpif->upcall\_lock)  {  int handler\_id;  int error = 0;  uint32\_t n\_handlers;  uint32\_t \*upcall\_pids;  // 重新计算 handlers 的数目  n\_handlers = dpif\_netlink\_calculate\_n\_handlers();  if (dpif->n\_handlers != n\_handlers) {  VLOG\_DBG("Dispatch mode(per-cpu): initializing %d handlers",  n\_handlers);  // 先把现有的 handler 信息销毁  destroy\_all\_handlers(dpif);  upcall\_pids = xzalloc(n\_handlers \* sizeof \*upcall\_pids);  // 分配 dpif\_handlers 数组  dpif->handlers = xzalloc(n\_handlers \* sizeof \*dpif->handlers);  for (handler\_id = 0; handler\_id < n\_handlers; handler\_id++) {  struct dpif\_handler \*handler = &dpif->handlers[handler\_id];  error = create\_nl\_sock(dpif, &handler->sock);  if (error) {  VLOG\_ERR("Dispatch mode(per-cpu): Cannot create socket for"  "handler %d", handler\_id);  continue;  }  upcall\_pids[handler\_id] = nl\_sock\_pid(handler->sock);  VLOG\_DBG("Dispatch mode(per-cpu): "  "handler %d has Netlink PID of %u",  handler\_id, upcall\_pids[handler\_id]);  }  dpif->n\_handlers = n\_handlers;  error = dpif\_netlink\_set\_handler\_pids(&dpif->dpif, upcall\_pids,  n\_handlers);  free(upcall\_pids);  }  return error;  } |

### 2 线程函数 udpif\_upcall\_handler()

|  |
| --- |
| （ofproto/ofproto-dpif-upcall.c）  static void \*  udpif\_upcall\_handler(void \*arg)  {  struct handler \*handler = arg;  struct udpif \*udpif = handler->udpif;  // 只要exit\_latch不设置就一直存在  while (!latch\_is\_set(&handler->udpif->exit\_latch)) {  //recv\_upcalls用于接收upcall消息。只有kernel path提供了recv函数dpif\_netlink\_recv，所以说handler线程对dpdk path不生效。  //recv\_upcalls返回值大于0，说明已经处理过upcall消息，防止  //丢失后续的upcall消息，线程不能堵塞，需要再次调用recv\_upcalls  //尝试接收upcall消息，所以需要调用poll\_immediate\_wake将  //timeout\_when设置为最小值LLONG\_MIN，这样poll\_block调用poll函数就能立即返回，继续执行recv\_upcalls。  //recv\_upcalls返回值小于0，说明没有upcall消息，dpif\_recv\_wait调用等待upcall消息即可。  if (recv\_upcalls(handler)) {  poll\_immediate\_wake();  } else {  //等待upcall消息  dpif\_recv\_wait(udpif->dpif, handler->handler\_id);  //等待exit消息  latch\_wait(&udpif->exit\_latch);  }  //堵塞在poll函数上，超时时间timeout\_when为最大值，等待事件发生  poll\_block();  }  return NULL;  } |

## OVS\_DP\_F\_DISPATCH\_UPCALL\_PER\_CPU& per-cpu dispatch mode& per-vport dispatch mode

参考：<https://dev.openvswitch.narkive.com/AU2urfzA/ovs-rfc-net-next-openvswitch-introduce-per-cpu-upcall-dispatch>

The Open vSwitch kernel module uses the upcall mechanism to send

packets from kernel space to user space when it misses in the kernel

space flow table. The upcall sends packets via a Netlink socket.

Currently, a Netlink socket is created for every vport. In this way,

there is a 1:1 mapping between a vport and a Netlink socket.

When a packet is received by a vport, if it needs to be sent to

user space, it is sent via the corresponding Netlink socket.

This mechanism, with various iterations of the corresponding user

space code, has seen some limitations and issues:

\* On systems with a large number of vports, there is a correspondingly

large number of Netlink sockets which can limit scaling.

(https://bugzilla.redhat.com/show\_bug.cgi?id=1526306)

\* Packet reordering on upcalls.

(https://bugzilla.redhat.com/show\_bug.cgi?id=1844576)

\* A thundering herd issue.

(https://bugzilla.redhat.com/show\_bug.cgi?id=1834444)

This patch introduces an alternative, feature-negotiated, upcall

mode using a per-cpu dispatch rather than a per-vport dispatch.

In this mode, the Netlink socket to be used for the upcall is

selected based on the CPU of the thread that is executing the upcall.

In this way, it resolves the issues above as:

a) The number of Netlink sockets scales with the number of CPUs

rather than the number of vports.

b) Ordering per-flow is maintained as packets are distributed to

CPUs based on mechanisms such as RSS and flows are distributed

to a single user space thread.

c) Packets from a flow can only wake up one user space thread.

## Upcall id 的设置

内核模块发起 upcall 时需要知道 vswitchd 进程 socket 对应的 pid，这里看下是如何存储和设置的。

在 dpif\_netlink\_refresh\_handlers\_cpu\_dispatch() 中创建 handler 线程时，vswitchd 会创建 socket 获取 pid，然后通知给内核：

|  |
| --- |
| （lib/dpif-netlink.c）  static int  dpif\_netlink\_refresh\_handlers\_cpu\_dispatch(struct dpif\_netlink \*dpif)  OVS\_REQ\_WRLOCK(dpif->upcall\_lock)  {  int handler\_id;  int error = 0;  uint32\_t n\_handlers;  uint32\_t \*upcall\_pids;  // 重新计算 handlers 的数目  n\_handlers = dpif\_netlink\_calculate\_n\_handlers();  if (dpif->n\_handlers != n\_handlers) {  VLOG\_DBG("Dispatch mode(per-cpu): initializing %d handlers",  n\_handlers);  // 先把现有的 handler 信息销毁  destroy\_all\_handlers(dpif);  upcall\_pids = xzalloc(n\_handlers \* sizeof \*upcall\_pids);  // 分配 dpif\_handlers 数组  dpif->handlers = xzalloc(n\_handlers \* sizeof \*dpif->handlers);  for (handler\_id = 0; handler\_id < n\_handlers; handler\_id++) {  struct dpif\_handler \*handler = &dpif->handlers[handler\_id];  error = create\_nl\_sock(dpif, &handler->sock);  if (error) {  VLOG\_ERR("Dispatch mode(per-cpu): Cannot create socket for"  "handler %d", handler\_id);  continue;  }  upcall\_pids[handler\_id] = nl\_sock\_pid(handler->sock);  VLOG\_DBG("Dispatch mode(per-cpu): "  "handler %d has Netlink PID of %u",  handler\_id, upcall\_pids[handler\_id]);  }  dpif->n\_handlers = n\_handlers;  error = dpif\_netlink\_set\_handler\_pids(&dpif->dpif, upcall\_pids,  n\_handlers);  free(upcall\_pids);  }  return error;  } |

其在 create\_nl\_sock() 中创建 socket 并获取其 pid，然后把 pid 手机到一个 upcall\_pids 数组中，最后调用 dpif\_netlink\_set\_handler\_pids() 把 pid 设置给内核。

### 内核模块处理

|  |
| --- |
| （net/openvswitch/datapath.h）  struct datapath {  …  struct dp\_nlsk\_pids \_\_rcu \*upcall\_portids;  }  struct dp\_nlsk\_pids {  struct rcu\_head rcu;  u32 n\_pids; // upcall\_pid 的总数  u32 pids[]; // 动态分配的数组，包含 n\_pids 个 upcall\_pid  }; |

Vswitchd 调用 dpif\_netlink\_set\_handler\_pids() 函数设置 upcall\_pid 时，调用的是 netlink 的 OVS\_DP\_CMD\_SET 命令，在内核中经过 ovs\_dp\_cmd\_set() -> ovs\_dp\_change() -> ovs\_dp\_set\_upcall\_portids()，在其中分配并设置 dp\_nlsk\_pids：

|  |
| --- |
| （net/openvswitch/datapath.c）  static int ovs\_dp\_set\_upcall\_portids(struct datapath \*dp,  const struct nlattr \*ids)  {  struct dp\_nlsk\_pids \*old, \*dp\_nlsk\_pids;  if (!nla\_len(ids) || nla\_len(ids) % sizeof(u32))  return -EINVAL;  old = ovsl\_dereference(dp->upcall\_portids);  // 分配 dp\_nlsk\_pids 结构  dp\_nlsk\_pids = kmalloc(sizeof(\*dp\_nlsk\_pids) + nla\_len(ids),  GFP\_KERNEL);  if (!dp\_nlsk\_pids)  return -ENOMEM;  dp\_nlsk\_pids->n\_pids = nla\_len(ids) / sizeof(u32);  // 拷贝用户设置的 upcall\_pid  nla\_memcpy(dp\_nlsk\_pids->pids, ids, nla\_len(ids));  rcu\_assign\_pointer(dp->upcall\_portids, dp\_nlsk\_pids);  kfree\_rcu(old, rcu);  return 0;  } |

### Upcall\_id 的查询

当内核需要执行 upcall() 时，会调用 ovs\_dp\_get\_upcall\_portid() 查询 upcall\_pid：

|  |
| --- |
| （net/openvswitch/datapath.c）  u32 ovs\_dp\_get\_upcall\_portid(const struct datapath \*dp, uint32\_t cpu\_id)  {  struct dp\_nlsk\_pids \*dp\_nlsk\_pids;  dp\_nlsk\_pids = rcu\_dereference(dp->upcall\_portids);  if (dp\_nlsk\_pids) {  if (cpu\_id < dp\_nlsk\_pids->n\_pids) {  return dp\_nlsk\_pids->pids[cpu\_id];  } else if (dp\_nlsk\_pids->n\_pids > 0 &&  cpu\_id >= dp\_nlsk\_pids->n\_pids) {  /\* If the number of netlink PIDs is mismatched with  \* the number of CPUs as seen by the kernel, log this  \* and send the upcall to an arbitrary socket (0) in  \* order to not drop packets  \*/  pr\_info\_ratelimited("cpu\_id mismatch with handler threads");  return dp\_nlsk\_pids->pids[cpu\_id %  dp\_nlsk\_pids->n\_pids];  } else {  return 0;  }  } else {  return 0;  }  } |

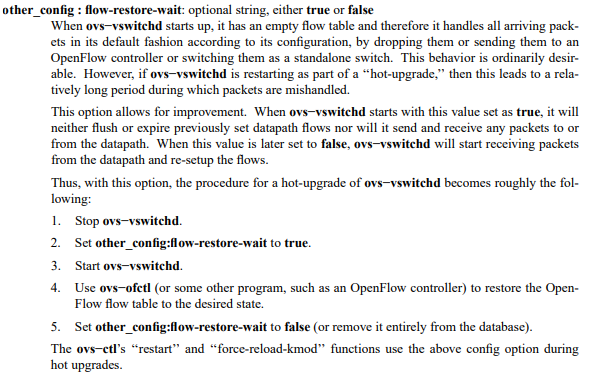
# 数据结构字段说明

## struct dpif\_backer

### recv\_set\_enable

|  |
| --- |
| bool recv\_set\_enable; /\* Enables or disables receiving packets. \*/ |

受 other\_config : flow-restore-wait 控制，参考 [ovs-vswitchd.conf.db.5.pdf (openvswitch.org)](http://www.openvswitch.org/ovs-vswitchd.conf.db.5.pdf)



用在 vswitchd 热升级时，这个字段设置为 true 时，就不处理 upcall 信息。具体可以看 open\_dpif\_backer() 函数中，这个字段为 true 时，才设置 upcall 线程：

|  |
| --- |
| （ofproto/ofproto-dpif.c）  static int  open\_dpif\_backer(const char \*type, struct dpif\_backer \*\*backerp)  {  …  if (backer->recv\_set\_enable) {  udpif\_set\_threads(backer->udpif, n\_handlers, n\_revalidators);  }  …  } |

## struct udpif

### n\_handlers& n\_revalidators

|  |
| --- |
| uint32\_t n\_handlers; // 当前实际的 handler 线程的数目  uint32\_t n\_revalidators; // 当前实际的 revalidator 线程的数目  struct handler \*handlers; /\* Upcall handlers. \*/  struct revalidator \*revalidators; /\* Flow revalidators. \*/ |

handlers 和 revalidators 都是数组，其中保存了hander 线程和 revalidator 线程的信息。

### exit\_latch

控制 handler 和 revalidator 线程的退出。

|  |
| --- |
| struct latch exit\_latch; /\* Tells child threads to exit. \*/ |

参考 udpif\_stop\_threads() 函数中，会设置该值：

|  |
| --- |
| （ofproto/ofproto-dpif-upcall.c）  static void  udpif\_stop\_threads(struct udpif \*udpif, bool delete\_flows)  {  if (udpif && (udpif->n\_handlers != 0 || udpif->n\_revalidators != 0)) {  …  /\* Tell the threads to exit. \*/  latch\_set(&udpif->exit\_latch);  }  } |

## struct handler

handler 结构保存 handler 线程的信息，参考 struct udpif 中的 handlers 数组。

|  |
| --- |
| /\* A thread that reads upcalls from dpif, forwards each upcall's packet,  \* and possibly sets up a kernel flow as a cache. \*/  struct handler {  struct udpif \*udpif; /\* Parent udpif. \*/  pthread\_t thread; /\* Thread ID. \*/  uint32\_t handler\_id; /\* Handler id. \*/  }; |

参考 udpif\_start\_threads() 中的 handler 线程初始化：

|  |
| --- |
| static void  udpif\_start\_threads(struct udpif \*udpif, uint32\_t n\_handlers\_,  uint32\_t n\_revalidators\_)  {  …  for (size\_t i = 0; i < udpif->n\_handlers; i++) {  struct handler \*handler = &udpif->handlers[i];    handler->udpif = udpif;  handler->handler\_id = i;  handler->thread = ovs\_thread\_create(  "handler", udpif\_upcall\_handler, handler);  }  …  } |

## struct dpif\_handler

如前文所属 dpif\_handler 是 dpif-netlink 中保存 handler 线程信息的结构。

### sock

|  |
| --- |
| (lib/dpif-netlink.c)  /\* per-cpu dispatch mode. \*/  struct nl\_sock \*sock; /\* Each handler thread holds one netlink  socket. \*/ |

per-cpu 模式下保存 netlink socket 信息。

在 dpif\_netlink\_refresh\_handlers\_cpu\_dispatch() 初始化时通过 create\_nl\_sock() 初始化：

|  |
| --- |
| （lib/dpif-netlink.c）  static int  create\_nl\_sock(struct dpif\_netlink \*dpif OVS\_UNUSED, struct nl\_sock \*\*sockp)  OVS\_REQ\_WRLOCK(dpif->upcall\_lock)  {  #ifndef \_WIN32  // 创建了一个 NETLINK\_GENERIC 的 socket  return nl\_sock\_create(NETLINK\_GENERIC, sockp);  #else  …  } |