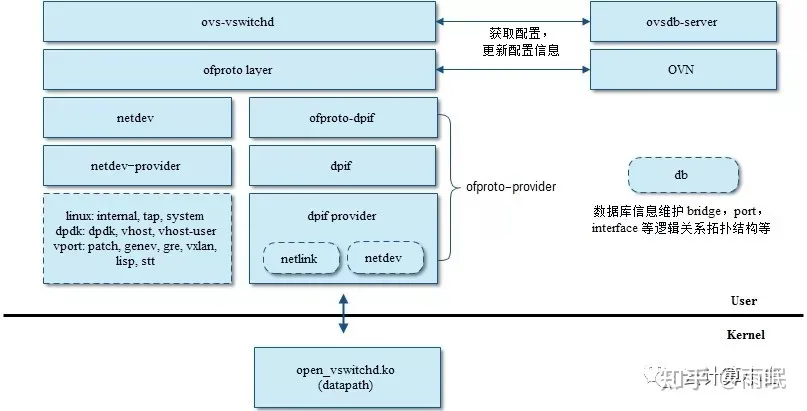
# Vswitchd



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；

**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。

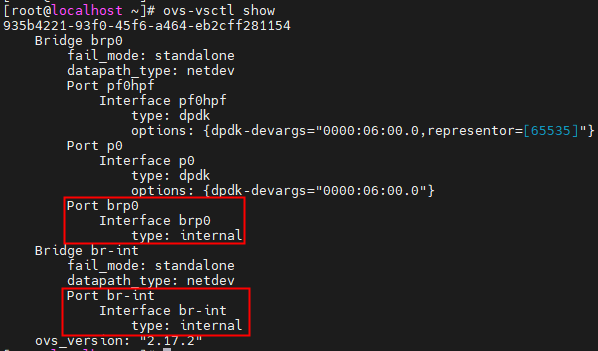
## vport

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);  } |