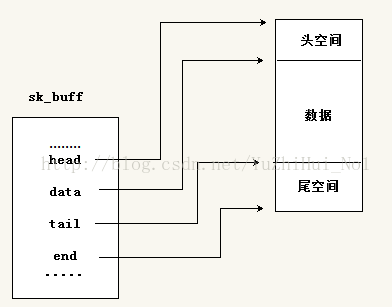
## sk\_buff的介绍

**结构体源码以及解释：**

|  |
| --- |
| struct sk\_buff {  /\* These two members must be first. \*/  struct sk\_buff \*next; // 因为sk\_buff结构体是双链表，所以有前驱后继。这是个指向后面的sk\_buff结构体指针  struct sk\_buff \*prev; // 这是指向前一个sk\_buff结构体指针  //老版本（2.6以前）应该还有个字段： sk\_buff\_head \*list //即每个sk\_buff结构都有个指针指向头节点  struct sock \*sk; // 指向拥有此缓冲的套接字sock结构体，即：宿主传输控制模块  ktime\_t tstamp; // 时间戳，表示这个skb的接收到的时间，一般是在包从驱动中往二层发送的接口函数中设置  struct net\_device \*dev; // 表示一个网络设备，当skb为输出/输入时，dev表示要输出/输入到的设备  unsigned long \_skb\_dst; // 主要用于路由子系统，保存路由有关的东西  char cb[48]; // 保存每层的控制信息,每一层的私有信息  unsigned int len, // 表示数据区的长度(tail - data)与分片结构体数据区的长度之和。其实这个len中数据区长度是个有效长度，  // 因为不删除协议头，所以只计算有效协议头和包内容。如：当在L3时，不会计算L2的协议头长度。  data\_len; // 只表示分片结构体数据区的长度，所以len = (tail - data) + data\_len；  \_\_u16 mac\_len, // mac报头的长度  hdr\_len; // 用于clone时，表示clone的skb的头长度  // 接下来是校验相关域，这里就不详细讲了。  \_\_u32 priority; // 优先级，主要用于QOS  kmemcheck\_bitfield\_begin(flags1);  \_\_u8 local\_df:1, // 是否可以本地切片的标志  cloned:1, // 为1表示该结构被克隆，或者自己是个克隆的结构体；同理被克隆时，自身skb和克隆skb的cloned都要置1  ip\_summed:2,  nohdr:1, // nohdr标识payload是否被单独引用，不存在协议首部。 // 如果被引用，则决不能再修改协议首部，也不能通过skb->data来访问协议首部。</span></span>  nfctinfo:3;  \_\_u8 pkt\_type:3, // 标记帧的类型  fclone:2, // 这个成员字段是克隆时使用，表示克隆状态  ipvs\_property:1,  peeked:1,  nf\_trace:1;  \_\_be16 protocol:16; // 这是包的协议类型，标识是IP包还是ARP包或者其他数据包。  kmemcheck\_bitfield\_end(flags1);  void (\*destructor)(struct sk\_buff \*skb); // 这是析构函数，后期在skb内存销毁时会用到  #if defined(CONFIG\_NF\_CONNTRACK) || defined(CONFIG\_NF\_CONNTRACK\_MODULE)  struct nf\_conntrack \*nfct;  struct sk\_buff \*nfct\_reasm;  #endif  #ifdef CONFIG\_BRIDGE\_NETFILTER  struct nf\_bridge\_info \*nf\_bridge;  #endif  int iif; // 接受设备的index  #ifdef CONFIG\_NET\_SCHED  \_\_u16 tc\_index; /\* traffic control index \*/  #ifdef CONFIG\_NET\_CLS\_ACT  \_\_u16 tc\_verd; /\* traffic control verdict \*/  #endif  #endif  kmemcheck\_bitfield\_begin(flags2);  \_\_u16 queue\_mapping:16;  #ifdef CONFIG\_IPV6\_NDISC\_NODETYPE  \_\_u8 ndisc\_nodetype:2;  #endif  kmemcheck\_bitfield\_end(flags2);  /\* 0/14 bit hole \*/  #ifdef CONFIG\_NET\_DMA  dma\_cookie\_t dma\_cookie;  #endif  #ifdef CONFIG\_NETWORK\_SECMARK  \_\_u32 secmark;  #endif  \_\_u32 mark;  \_\_u16 vlan\_tci;  sk\_buff\_data\_t transport\_header; // 指向四层帧头结构体指针  sk\_buff\_data\_t network\_header; // 指向三层IP头结构体指针  sk\_buff\_data\_t mac\_header; // 指向二层mac头的头  /\* These elements must be at the end, see alloc\_skb() for details. \*/  sk\_buff\_data\_t tail; // 指向数据区中实际数据结束的位置  sk\_buff\_data\_t end; // 指向数据区中结束的位置（非实际数据区域结束位置）  unsigned char \*head, // 指向数据区中开始的位置（非实际数据区域开始位置）  \*data; // 指向数据区中实际数据开始的位置  unsigned int truesize; // 表示总长度，包括sk\_buff自身长度和数据区以及分片结构体的数据区长度  atomic\_t users; // skb被克隆引用的次数，在内存申请和克隆时会用到  }; //end sk\_buff |

**数据区结构：**

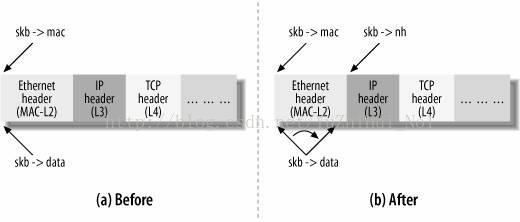
****

(1)sk\_buff->data\_len：只计算分片中数据的长度，即是分片结构体中page指向的数据区长度。

(2)sk\_buff->len：表示当前缓冲区中数据块的大小的总长度。它包括主缓冲中（即是sk\_buff结构中指针data指向）的数据区的实际长度（data-tail）和分片中的数据长度。这个长度在数据包在各层间传输时会改变，因为分片数据长度不变，从L2到L4时，则len要减去帧头大小和网络头大小；从L4到L2则相反，要加上帧头和网络头大小。所以：len = (data - tail) + data\_len；

(3)sk\_buff->truesize：这是缓冲区的总长度，包括sk\_buff结构和数据部分。如果申请一个len字节的缓冲区，alloc\_skb函数会把它初始化成len+sizeof(sk\_buff)。当skb->len变化时，这个变量也会变化。所以：truesize = len + sizeof(sk\_buff) = (data - tail) + data\_len + sizeof(sk\_buff)；

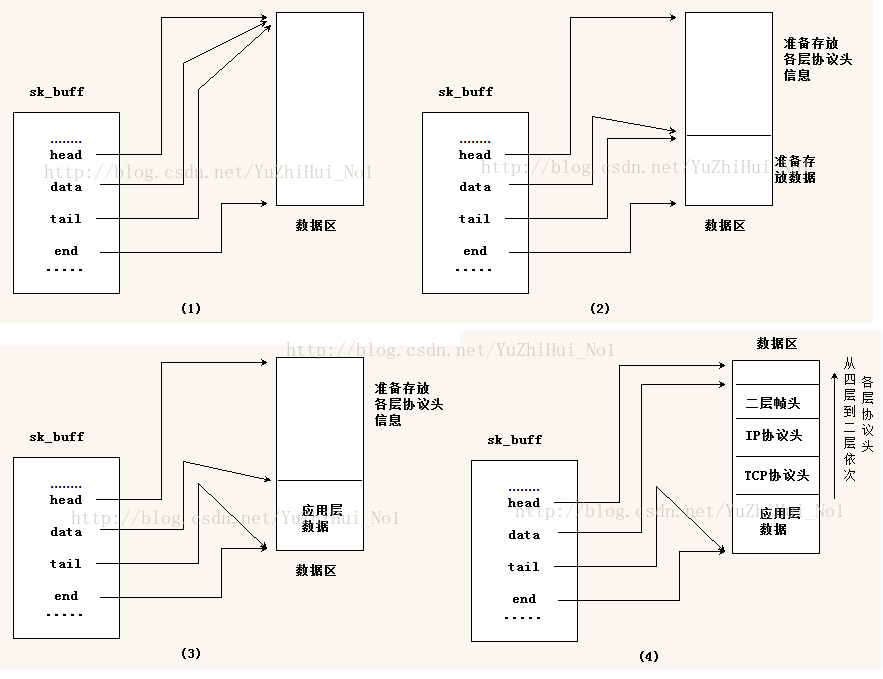
**数据区指针移动示例：**

****

（1）开始进入第二层时，这时data指针指向帧头。mac = data，然后操作mac指针已经数据包。当二层操作完后把包往三层传送时，会让data指针指向三层的IP头；

（2）当包进入第三层时，这时data指针已经指向了IP头，让nh = data，然后操作nh指针已经数据包，当三层操作完后把包往四层传送时，同样把data指向四层的TCP头；同理，四层也是一样处理的，只移动指针，不删除协议头。发包时就相反了，只是变成了为每一层添加协议头了。

**数据区各个指针移动过程：**

****

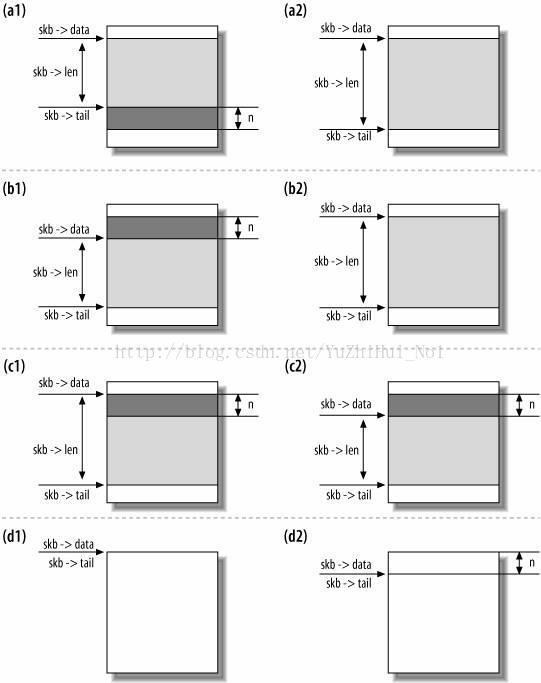
（1）初始状态，刚刚申请内存

（2）准备存应用层数据

（3）移动tail，添加数据

（4）添加协议头

**常用操作数据空间函数：**

****

（a）skb\_put()：向后扩大数据区空间，headroom空间不变，tailroom空间减少，skb->data指针不变，skb->tail指针下移；

（b）skb\_push()：向前扩大数据区空间，headroom空间减少，tailroom空间不变，skb->tail指针不变，skb->data指针上移；

（c）skb\_pull()：缩小数据区空间，headroom空间增大，tailroom空间不变，skb->data指针下移，skb->tail指针不变；

（d）skb\_reserve()：数据区不变，headroom空间增大，tailroom空间减少，skb->data和skb->tail同时下移；

## Generic Netlink中sk\_buff的填充

netlink这个协议在使用的sk\_buff结构体中，所有的数据都会填充到数据空间中，头空间不变（netlink的头也会放到数据空间中）。

**步骤如下：**

通过enlmsg\_new新建一个sk\_buff结构体，

|  |
| --- |
| /\*\*  \* genlmsg\_new - Allocate a new generic netlink message  \* @payload: size of the message payload  \* @flags: the type of memory to allocate.  \*/  static inline struct sk\_buff \*genlmsg\_new(size\_t payload, gfp\_t flags)  生成的sk\_buff包括了netlink头和generic netlink头的大小 |

通过genlmsg\_new新建一个sk\_buff结构体，

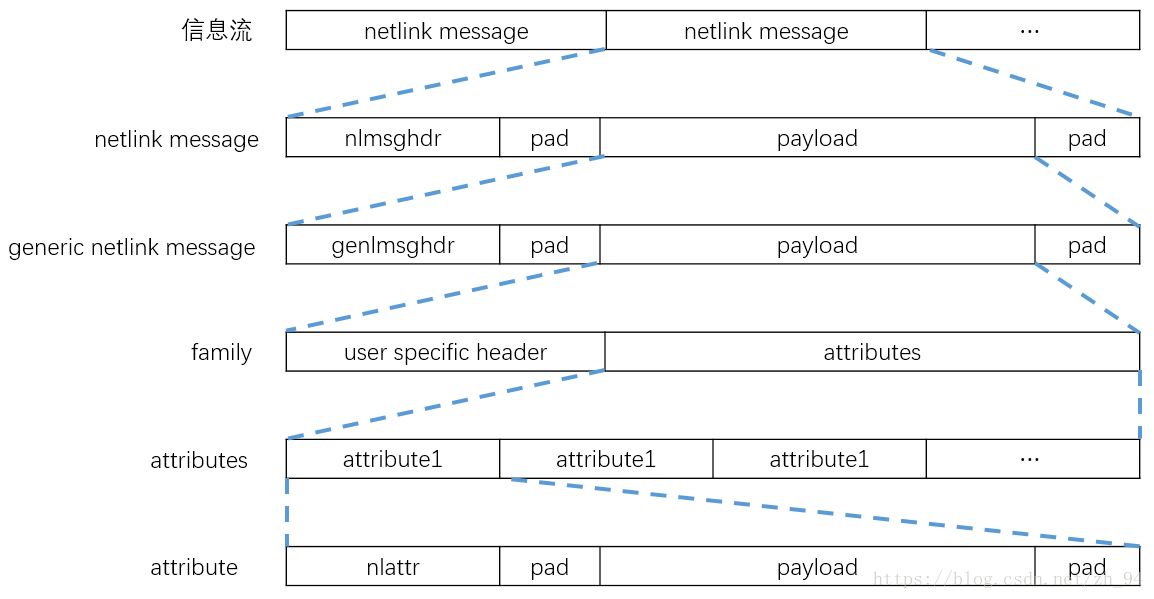
|  |
| --- |
| /\*\*  \* genlmsg\_put - Add generic netlink header to netlink message，和用户自定义头  \* @skb: socket buffer holding the message  \* @portid: netlink portid the message is addressed to  \* @seq: sequence number (usually the one of the sender)  \* @family: generic netlink family  \* @flags: netlink message flags  \* @cmd: generic netlink command  \* Returns pointer to user specific header  \*/添加netlink头（已经填写内容）、 generic netlink head（已经填写内容），以及用户自定义头  void \*genlmsg\_put(struct sk\_buff \*skb, u32 portid, u32 seq,  const struct genl\_family \*family, int flags, u8 cmd) |

通过\_\_nla\_put添加netlink attribute，

|  |
| --- |
| /\*\*  \* \_\_nla\_put - Add a netlink attribute to a socket buffer  \* @skb: socket buffer to add attribute to  \* @attrtype: attribute type  \* @attrlen: length of attribute payload  \* @data: head of attribute payload  \*  \* The caller is responsible to ensure that the skb provides enough  \* tailroom for the attribute header and payload.  \*/  void \_\_nla\_put(struct sk\_buff \*skb, int attrtype, int attrlen,  const void \*data) |

## Generic Netlink介绍

**数据包格式介绍:**



1. nlmsghdr是netlink包头，记录长度、类型等
2. payload是真正的数据部分
3. pad是数据填充，目的是为了对齐
4. genlmasghdr是扩展的头，记录相应的命令号
5. family是用来区分不同的Generic Netlink

**使用流程：**

kernel端程序的准备：

1、定义想要传送的消息种类

2、定义一组命令

3、为每个命令定义一个响应函数

4、将命令与相应函数关联起来

5、创建一个命令族

6、向内核注册新建的命令族

用户端程序的使用：

1、新建一个套接字，并绑定上 netlink address

2、向 GENL\_ID\_CRTL 询问想通信的命令族的 id

3、等待接收 GENL\_ID\_CRTL 的返回的查询结果

4、拆分读取接收到的信息

5、与想要通信的命令族通信

以下介绍以ovs中源码作为例子。

**定义想要传送的消息种类**

在openvswithch.h文件下

|  |
| --- |
| enum ovs\_packet\_attr {  OVS\_PACKET\_ATTR\_UNSPEC,  OVS\_PACKET\_ATTR\_PACKET, /\* Packet data. \*/  OVS\_PACKET\_ATTR\_KEY, /\* Nested OVS\_KEY\_ATTR\_\* attributes. \*/  OVS\_PACKET\_ATTR\_ACTIONS, /\* Nested OVS\_ACTION\_ATTR\_\* attributes. \*/  OVS\_PACKET\_ATTR\_USERDATA, /\* OVS\_ACTION\_ATTR\_USERSPACE arg. \*/  OVS\_PACKET\_ATTR\_JIFFIES, /\*OVS\_PACKET\_ATTR\_JIFFIES Gary's code\*/  OVS\_PACKET\_ATTR\_EGRESS\_TUN\_KEY, /\* Nested OVS\_TUNNEL\_KEY\_ATTR\_\*  attributes. \*/  OVS\_PACKET\_ATTR\_UNUSED1,  OVS\_PACKET\_ATTR\_UNUSED2,  OVS\_PACKET\_ATTR\_PROBE, /\* Packet operation is a feature probe,  error logging should be suppressed. \*/  OVS\_PACKET\_ATTR\_MRU, /\* Maximum received IP fragment size. \*/  OVS\_PACKET\_ATTR\_LEN, /\* Packet size before truncation. \*/  \_\_OVS\_PACKET\_ATTR\_MAX  }; |

**定义一组命令**

|  |
| --- |
| enum ovs\_vport\_cmd {  OVS\_VPORT\_CMD\_UNSPEC,  OVS\_VPORT\_CMD\_NEW,  OVS\_VPORT\_CMD\_DEL,  OVS\_VPORT\_CMD\_GET,  OVS\_VPORT\_CMD\_SET  }; |

**为每个命令定义一个响应函数，并将命令与相应函数关联起来**

|  |
| --- |
| static struct genl\_ops dp\_vport\_genl\_ops[] = {  { .cmd = OVS\_VPORT\_CMD\_NEW,//新建vport  .flags = GENL\_UNS\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN privilege. \*/  .policy = vport\_policy,  .doit = ovs\_vport\_cmd\_new  },  { .cmd = OVS\_VPORT\_CMD\_DEL,//删除vport  .flags = GENL\_UNS\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN privilege. \*/  .policy = vport\_policy,  .doit = ovs\_vport\_cmd\_del  },  { .cmd = OVS\_VPORT\_CMD\_GET,//获取vport  .flags = 0, /\* OK for unprivileged users. \*/  .policy = vport\_policy,  .doit = ovs\_vport\_cmd\_get,  .dumpit = ovs\_vport\_cmd\_dump  },  { .cmd = OVS\_VPORT\_CMD\_SET,//修改vport  .flags = GENL\_UNS\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN privilege. \*/  .policy = vport\_policy,  .doit = ovs\_vport\_cmd\_set,  },  };  static const struct nla\_policy vport\_policy[OVS\_VPORT\_ATTR\_MAX + 1] = {  [OVS\_VPORT\_ATTR\_NAME] = { .type = NLA\_NUL\_STRING, .len = IFNAMSIZ - 1 },  [OVS\_VPORT\_ATTR\_STATS] = { .len = sizeof(struct ovs\_vport\_stats) },  [OVS\_VPORT\_ATTR\_PORT\_NO] = { .type = NLA\_U32 },  [OVS\_VPORT\_ATTR\_TYPE] = { .type = NLA\_U32 },  [OVS\_VPORT\_ATTR\_UPCALL\_PID] = { .type = NLA\_U32 },  [OVS\_VPORT\_ATTR\_OPTIONS] = { .type = NLA\_NESTED },  [OVS\_VPORT\_ATTR\_IFINDEX] = { .type = NLA\_U32 },  [OVS\_VPORT\_ATTR\_NETNSID] = { .type = NLA\_S32 },  }; |

* cmd: 命令名。用于识别各genl\_ops
* flag: 各种设置属性，以“或”连接。在需要admin特权级别时，使用GENL\_ADMIN\_PERM
* policy：定义了attr规则。如果此指针非空，genl在触发事件处理程序之前，会使用这个字段来对帧中的attr做校验（见nlmsg\_parse函数）。该字段可以为空，表示在触发事件处理程序之前，不做校验。定义所能处理的数据类型
* doit：这是一个回调函数。在generic netlink收到数据时触发，运行在进程上下文。doit传入两个参数，skb为触发此回调函数的socket buffer。第二个参数是一个genl\_info结构体
* dumpit：这是一个回调函数，当genl\_ops的flag标志被添加了NLM\_F\_DUMP以后，每次收到genl消息即会回触发这个函数。 dumpit与doit的区别是:dumpit的第一个参数skb不会携带从客户端发来的数据。相反地，开发者应该在skb中填入需要传给客户端的数据， 然后，并skb的数据长度（可以用skb->len）return。skb中携带的数据会被自动送到客户端。

nla\_policy结构体是定义相应的属性是什么样的数据类型（可以不指定），以及每一个属性的长度。

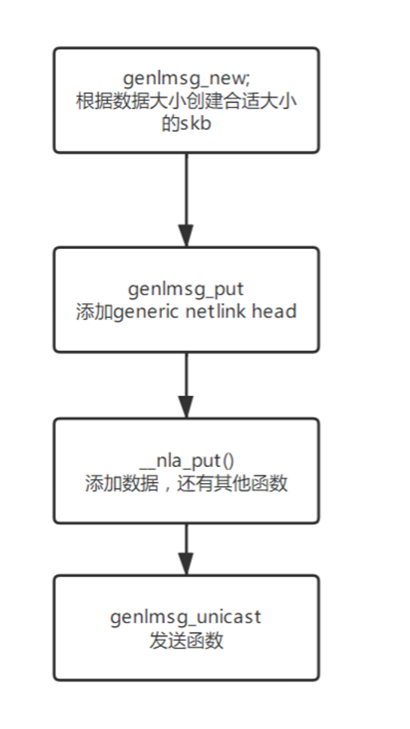
**创建一个命令族**

|  |
| --- |
| struct genl\_family dp\_vport\_genl\_family \_\_ro\_after\_init = {  .hdrsize = sizeof(struct ovs\_header), //用户自定义头部长度  .name = OVS\_VPORT\_FAMILY,  .version = OVS\_VPORT\_VERSION,  .maxattr = OVS\_VPORT\_ATTR\_MAX, //最大的attribute数量  .netnsok = true, //是否支持处理network namespace  .parallel\_ops = true, //操作是否可以并行调用  .ops = dp\_vport\_genl\_ops, //支持的操作  .n\_ops = ARRAY\_SIZE(dp\_vport\_genl\_ops), //支持的操作数  .mcgrps = &ovs\_dp\_vport\_multicast\_group, //该family使用的多播组  .n\_mcgrps = 1,  .module = THIS\_MODULE, //所属于的模块  }; |

**向内核注册新建的命令族**

|  |
| --- |
| static struct genl\_family \*dp\_genl\_families[] = {  &dp\_datapath\_genl\_family,  &dp\_vport\_genl\_family,  &dp\_flow\_genl\_family,  &dp\_packet\_genl\_family,  &dp\_meter\_genl\_family,  #if IS\_ENABLED(CONFIG\_NETFILTER\_CONNCOUNT)  &dp\_ct\_limit\_genl\_family,  #endif  };  //注册定义的genl\_family  static int \_\_init dp\_register\_genl(void)  {  int err;  int i;  for (i = 0; i < ARRAY\_SIZE(dp\_genl\_families); i++) {  err = genl\_register\_family(dp\_genl\_families[i]);  if (err)  goto error;  }  return 0;  error:  dp\_unregister\_genl(i);  return err;  } |

**内核中数据填充和发送过程：**

****

## dp\_genl\_families涉及操作

|  |
| --- |
| static struct genl\_family \*dp\_genl\_families[] = {  &dp\_datapath\_genl\_family,  &dp\_vport\_genl\_family,  &dp\_flow\_genl\_family,  &dp\_packet\_genl\_family,  &dp\_meter\_genl\_family,  #if IS\_ENABLED(CONFIG\_NETFILTER\_CONNCOUNT)  &dp\_ct\_limit\_genl\_family,  #endif  }; |

dp\_datapath\_genl\_family涉及对datapath的一些控制。

dp\_vport\_genl\_family涉及对vport的一些控制。

dp\_flow\_genl\_family涉及对流表的一些控制。

dp\_packet\_genl\_family用户态流表查询结束后，发出的对数据包的操作。

dp\_meter\_genl\_family涉及对meter表的一些操作。

dp\_ct\_limit\_genl\_family这个没看懂。

operation并没有包括所有的netlink，只是包含了所有接收用户态的netlink命令，因为这些operation都有一个回调函数，显然内核态发给用户态的回调函数是没有的，因此，operation没有包含内核态发送给用户态的netlink命令，可以通过openvswitch.h文件，该文件中有着所有的netlink命令。

1. **dp\_datapath\_gel\_ops:**

|  |
| --- |
| static struct genl\_ops dp\_datapath\_genl\_ops[] = {  { .cmd = OVS\_DP\_CMD\_NEW,//新建datapath  .flags = GENL\_UNS\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN privilege. \*/  .policy = datapath\_policy,  .doit = ovs\_dp\_cmd\_new  },  { .cmd = OVS\_DP\_CMD\_DEL,//删除datapath  .flags = GENL\_UNS\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN privilege. \*/  .policy = datapath\_policy,  .doit = ovs\_dp\_cmd\_del  },  { .cmd = OVS\_DP\_CMD\_GET,//获取datapath  .flags = 0, /\* OK for unprivileged users. \*/  .policy = datapath\_policy,  .doit = ovs\_dp\_cmd\_get,  .dumpit = ovs\_dp\_cmd\_dump  },  { .cmd = OVS\_DP\_CMD\_SET,//修改datapath  .flags = GENL\_UNS\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN privilege. \*/  .policy = datapath\_policy,  .doit = ovs\_dp\_cmd\_set,  },  }; |

datapath结构体代表一个网桥

1. **dp\_vport\_gel\_ops：**

|  |
| --- |
| static struct genl\_ops dp\_vport\_genl\_ops[] = {  { .cmd = OVS\_VPORT\_CMD\_NEW,//新建vport  .flags = GENL\_UNS\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN privilege. \*/  .policy = vport\_policy,  .doit = ovs\_vport\_cmd\_new  },  { .cmd = OVS\_VPORT\_CMD\_DEL,//删除vport  .flags = GENL\_UNS\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN privilege. \*/  .policy = vport\_policy,  .doit = ovs\_vport\_cmd\_del  },  { .cmd = OVS\_VPORT\_CMD\_GET,//获取vport  .flags = 0, /\* OK for unprivileged users. \*/  .policy = vport\_policy,  .doit = ovs\_vport\_cmd\_get,  .dumpit = ovs\_vport\_cmd\_dump  },  { .cmd = OVS\_VPORT\_CMD\_SET,//修改vport  .flags = GENL\_UNS\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN privilege. \*/  .policy = vport\_policy,  .doit = ovs\_vport\_cmd\_set,  },  }; |

vport代表网桥中各个端口的结构体

1. **dp\_flow\_gel\_ops：**

|  |
| --- |
| static struct genl\_ops dp\_flow\_genl\_ops[] = {  { .cmd = OVS\_FLOW\_CMD\_NEW,//添加新的流表项  .flags = GENL\_UNS\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN privilege. \*/  .policy = flow\_policy,  .doit = ovs\_flow\_cmd\_new  },  { .cmd = OVS\_FLOW\_CMD\_DEL,//删除流表项  .flags = GENL\_UNS\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN privilege. \*/  .policy = flow\_policy,  .doit = ovs\_flow\_cmd\_del  },  { .cmd = OVS\_FLOW\_CMD\_GET,//获取流表  .flags = 0, /\* OK for unprivileged users. \*/  .policy = flow\_policy,  .doit = ovs\_flow\_cmd\_get,  .dumpit = ovs\_flow\_cmd\_dump  },  { .cmd = OVS\_FLOW\_CMD\_SET,//修改流表  .flags = GENL\_UNS\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN privilege. \*/  .policy = flow\_policy,  .doit = ovs\_flow\_cmd\_set,  },  }; |

1. **dp\_packet\_genl\_ops**

|  |
| --- |
| static struct genl\_ops dp\_packet\_genl\_ops[] = {  { .cmd = OVS\_PACKET\_CMD\_EXECUTE,  .flags = GENL\_UNS\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN privilege. \*/  .policy = packet\_policy,  .doit = ovs\_packet\_cmd\_execute  }  }; |

回调函数ovs\_packet\_cmd\_execute最终会调用ovs\_execute\_actions函数。

ovs\_execute\_actions函数在内核态中，还被查询流表结束后进行调用处理数据包。

这是在用户态流表查询完后发出的命令

1. **dp\_meter\_gel\_ops:**

|  |
| --- |
| static struct genl\_ops dp\_meter\_genl\_ops[] = {  { .cmd = OVS\_METER\_CMD\_FEATURES,//请求内核态中meter表的属性比如MAX\_METERS、MAX\_BANDS、bands的类型  .flags = 0, /\* OK for unprivileged users. \*/  .policy = meter\_policy,  .doit = ovs\_meter\_cmd\_features  },  { .cmd = OVS\_METER\_CMD\_SET,//修改meter表  .flags = GENL\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN  \* privilege.  \*/  .policy = meter\_policy,  .doit = ovs\_meter\_cmd\_set,  },  { .cmd = OVS\_METER\_CMD\_GET,//获取meter表  .flags = 0, /\* OK for unprivileged users. \*/  .policy = meter\_policy,  .doit = ovs\_meter\_cmd\_get,  },  { .cmd = OVS\_METER\_CMD\_DEL,//删除meter表  .flags = GENL\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN  \* privilege.  \*/  .policy = meter\_policy,  .doit = ovs\_meter\_cmd\_del  },  }; |

meter表可以用来实现简单的Qos功能，比如速率限制等

1. **ct\_limit\_genl\_ops：**

|  |
| --- |
| static struct genl\_ops ct\_limit\_genl\_ops[] = {  { .cmd = OVS\_CT\_LIMIT\_CMD\_SET,  .flags = GENL\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN  \* privilege. \*/  .policy = ct\_limit\_policy,  .doit = ovs\_ct\_limit\_cmd\_set,  },  { .cmd = OVS\_CT\_LIMIT\_CMD\_DEL,  .flags = GENL\_ADMIN\_PERM, /\* Requires CAP\_NET\_ADMIN  \* privilege. \*/  .policy = ct\_limit\_policy,  .doit = ovs\_ct\_limit\_cmd\_del,  },  { .cmd = OVS\_CT\_LIMIT\_CMD\_GET,  .flags = 0, /\* OK for unprivileged users. \*/  .policy = ct\_limit\_policy,  .doit = ovs\_ct\_limit\_cmd\_get,  },  };  - OVS\_CT\_LIMIT\_CMD\_SET:  \* set default connection limit for all zones  \* set the connection limit for a particular zone  - OVS\_CT\_LIMIT\_CMD\_DEL:  \* remove the connection limit for a particular zone  - OVS\_CT\_LIMIT\_CMD\_GET:  \* get the default connection limit for all zones  \* get the connection limit for a particular zone |

没看懂要干嘛

## 内核态流表的查询过程

**TTS算法的主要组成部分：（参考：**[**https://github.com/yearsj/ovs\_notes**](https://github.com/yearsj/ovs_notes)**）**

**Rule : 单条的包过滤规则+动作**

以下为具体的例子：

1 Rule #1: ip\_src=192.168.0.0/16 ip\_dst=0/0 protocol=0/0 port\_src=0/0 port\_dst=0/0

2 Rule #2: ip\_src=0/0 ip\_dst=23.23.233.0/24 protocol=6/8(TCP) port\_src=0/0 port\_dst=23/16

3 Rule #3: ip\_src=0/0 ip\_dst=11.11.233.0/24 protocol=17/8(UDP) port\_dst=0/0 port\_dst=4789/16

4 Rule #4: ip\_src=10.10.0.0/16 ip\_dst=0/0 protocol=0/0 port\_src=0/0 port\_dst=0/0

可以看到一个rule中有多个字段，每个字段的形式为 ：**字段值/掩码前缀**

**Tuple : 使用相同的匹配字段+每个匹配字段都使用相同的掩码长度**

以下为具体的例子：

1 Tuple #1: ip\_src\_mask=16 ip\_dst\_mask=0 protocol\_mask=0 port\_src\_mask=0 port\_dst\_mask=0

2 Tuple #2: ip\_src\_mask=0 ip\_dst\_mask=24 protocol\_mask=8 port\_src\_mask=0 port\_dst\_mask=16

tuple是将有**相同规则**的rule进行合并，例如上述rule #1和rule #4可以看成是同一个tuple #1，因为其每个字段的掩码都相同，所以tuple有如下特点：

1. 使用相同的匹配字段
2. 每个匹配字段都使用相同的掩码长度

**Key：用于hash**

以Tuple #2中的Rule #2为例说明一下，首先用tuple的掩码去**与**rule中的各个**字段值**，丢弃tuple不关心的位，得到：

ip\_src=\_ ip\_dst=23.23.233 protocol=6 port\_src=\_ port\_dst=23

然后把这些位拼接起来，就是哈希表的key，转换为二进制如下：

key = 0001 0111(23) 0001 0111(23) 1110 1001(233) 0000 0110(6) 0000 0000 0001 0111(23)

最后，用这个key去做散列，即是哈希表的索引

**匹配过程：**

* 所有的rule都被分成了多个tuple，并存储在相应tuple下的哈希表中
* 当要对一个包进行匹配时，将遍历这多个tuple下的哈希表，一个一个查过去，查出所有匹配成功的结果，然后按一定策略在匹配结果中选出最优的一个。

**OVS：**

ovs中内核态流表使用的就是前面介绍的TTS算法的形式。所以查找流表过程：

for 每一个过滤规则：

for 相应过滤规则对应的元组中的每一个rule：

if（rule与当前的数据包匹配）：

返回查找到的流表

现在ovs流表中有一个microflow cache（别人是那么叫的，程序里面是一个struct mask\_cache\_entry），存放着前几次的过滤规则的索引，这样如果命中那么就可以减少最外层的循环。

**涉及相关查表函数的解释：**

**ovs\_flow\_tbl\_lookup\_stats：**

|  |
| --- |
| struct sw\_flow \*ovs\_flow\_tbl\_lookup\_stats(struct flow\_table \*tbl,  const struct sw\_flow\_key \*key,  u32 skb\_hash,  u32 \*n\_mask\_hit)  {  struct mask\_array \*ma = rcu\_dereference(tbl->mask\_array);  struct table\_instance \*ti = rcu\_dereference(tbl->ti);//这个是干啥用的  struct mask\_cache\_entry \*entries, \*ce;  struct sw\_flow \*flow;  u32 hash;  int seg;  \*n\_mask\_hit = 0;  if (unlikely(!skb\_hash)) {  u32 mask\_index = 0;  return flow\_lookup(tbl, ti, ma, key, n\_mask\_hit, &mask\_index);  }  /\* Pre and post recirulation flows usually have the same skb\_hash  \* value. To avoid hash collisions, rehash the 'skb\_hash' with  \* 'recirc\_id'. \*/  if (key->recirc\_id)//从数据包中提取key的时候该值设置为0  skb\_hash = jhash\_1word(skb\_hash, key->recirc\_id);  ce = NULL;  hash = skb\_hash;  entries = this\_cpu\_ptr(tbl->mask\_cache);//具体参见per\_cpu变量  /\* Find the cache entry 'ce' to operate on. \*/  for (seg = 0; seg < MC\_HASH\_SEGS; seg++) {  int index = hash & (MC\_HASH\_ENTRIES - 1);//后八位全是1，前面全是0，取出hash后八位的值作为index  struct mask\_cache\_entry \*e;  e = &entries[index];  if (e->skb\_hash == skb\_hash) {//如果在cache entry找到报文hash相同项，则根据该entry指定的mask查表  atomic\_inc(&hit\_cache);  flow = flow\_lookup(tbl, ti, ma, key, n\_mask\_hit,  &e->mask\_index);  if (!flow)//没找到，就让cache中相应的mask无效  e->skb\_hash = 0;  return flow;  }  if (!ce || e->skb\_hash < ce->skb\_hash)  ce = e; /\* A better replacement cache candidate. \*/  hash >>= MC\_HASH\_SHIFT;  }  /\* Cache miss, do full lookup. \*/  flow = flow\_lookup(tbl, ti, ma, key, n\_mask\_hit, &ce->mask\_index);  if (flow)  ce->skb\_hash = skb\_hash;  return flow;  } |

\*

@ 参数：

tbl：相应datapath中的流表

key：数据包提出出的特征

skb\_hash:数据包的hash值，根据数据包源IP、目的IP、源端口和目的端口计算

n\_mask\_hit:记录查找mask的次数

@ 返回值：返回查找到的流表，失败返回NULL

@ 描述：通过skb\_hash寻找到最有可能的mask（匹配规则）,然后用得到的mask传入flow\_lookup函数进行流表的查询

\*/

**flow\_lookup：**

|  |
| --- |
| static struct sw\_flow \*flow\_lookup(struct flow\_table \*tbl,  struct table\_instance \*ti,  const struct mask\_array \*ma,  const struct sw\_flow\_key \*key,  u32 \*n\_mask\_hit,  u32 \*index)  {  struct sw\_flow\_mask \*mask;  struct sw\_flow \*flow;  int i;  if (\*index < ma->max) {//如果index有效，那么只需要查询一个元组内的rule  mask = rcu\_dereference\_ovsl(ma->masks[\*index]);  if (mask) {  flow = masked\_flow\_lookup(ti, key, mask, n\_mask\_hit);  if (flow)  return flow;  }  }  for (i = 0; i < ma->max; i++) {  if (i == \*index)  continue;  mask = rcu\_dereference\_ovsl(ma->masks[i]);  if (!mask)  continue;  flow = masked\_flow\_lookup(ti, key, mask, n\_mask\_hit);  if (flow) { /\* Found \*/  \*index = i;  return flow;  }  }  return NULL;  } |

/\*

@ 参数：

tbl：相应datapath中的流表

ti：流表实例

ma：mask数组

key:数据包提取出的key值

n\_mask\_hit:记录查找mask的次数

index：ma[index]是通过sk\_hash查找到的mask

@ 返回值：返回查找到的流表，失败返回NULL

@ 描述：如果传入的index是有效的那么只需要查找一个桶内的flow（一个元组内的rule），如果不是有效的那么需要遍历所有的桶

\*/

参数中的相关结构体：

|  |
| --- |
| struct flow\_table {  struct table\_instance \_\_rcu \*ti;  struct table\_instance \_\_rcu \*ufid\_ti;  struct mask\_cache\_entry \_\_percpu \*mask\_cache;  struct mask\_array \_\_rcu \*mask\_array;  unsigned long last\_rehash;  unsigned int count;  unsigned int ufid\_count;  };  struct table\_instance {  struct flex\_array \*buckets;  unsigned int n\_buckets;  struct rcu\_head rcu;  int node\_ver;  u32 hash\_seed;  bool keep\_flows;  }; |

**masked\_flow\_lookup：**

|  |
| --- |
| static struct sw\_flow \*masked\_flow\_lookup(struct table\_instance \*ti,  const struct sw\_flow\_key \*unmasked,  const struct sw\_flow\_mask \*mask,  u32 \*n\_mask\_hit)  {  struct sw\_flow \*flow;  struct hlist\_head \*head;  u32 hash;  struct sw\_flow\_key masked\_key;  ovs\_flow\_mask\_key(&masked\_key, unmasked, false, mask);  hash = flow\_hash(&masked\_key, &mask->range);  head = find\_bucket(ti, hash);  (\*n\_mask\_hit)++;  hlist\_for\_each\_entry\_rcu(flow, head, flow\_table.node[ti->node\_ver]) {//遍历元组中的rule  if (flow->mask == mask && flow->flow\_table.hash == hash &&  flow\_cmp\_masked\_key(flow, &masked\_key, &mask->range))  return flow;  }  return NULL;  } |

/\*

@ 参数：

ti：流表实例

unmasked：数据包提取出的key

mask:相应的掩码，掩码与key相与之后，获取hash值（与之前的hash不一样，之前的hash是为了寻找mask），通过获取的hash值寻找元组所在的链表头

n\_mask\_hit:记录查找mask的次数

@ 返回值：返回查找到的流表，失败则返回NULL

@ 描述：查找到相应元组的链表头部，然后遍历该元组查找相匹配的rule

\*/

## Epoll介绍

**I/O模式介绍:**

I/O过程主要是这两个阶段

等待数据准备 (Waiting for the data to be ready)，将数据放到内核缓冲区（这一步数据生成都是操作系统来进行）

将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)（异步是操作系统来进行拷贝）

linux系统有了下面五种网络模式的方案。  
- 阻塞 I/O（blocking IO）  
- 非阻塞 I/O（nonblocking IO）  
- I/O 多路复用（ IO multiplexing）  
- 信号驱动 I/O（ signal driven IO）  
- 异步 I/O（asynchronous IO）

blocking IO和non-blocking I/O的区别是否block住I/O过程的第1个阶段（缓冲区数据准备），而第2阶段（拷贝到用户进程）都是block。

synchronous IO和asynchronous IO的区别是否block住I/O过程的第2个阶段，对于异步IO来说第一个过程和第二个过程都是操作都是操作系统来做。

I/O 多路复用（ IO multiplexing）实际上是阻塞IO和非阻塞IO方式的一种折中，阻塞方式中一个用户线程只能监听一个文件描述符，非阻塞方式可以通过循环方式监听多个文件描述符，但是如果文件描述符过少会浪费cpu资源。

多路复用是将系统中所有需要监听的文件描述符进行集中管理，操作系统负责轮询整个系统的文件描述符，而用户只需要将需要监听的文件描述符交给操作系统，这样一个用户线程就可以监听多个文件描述符，并且不会浪费CPU资源。具体的机制有select、poll、和epoll

select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

（1）select：

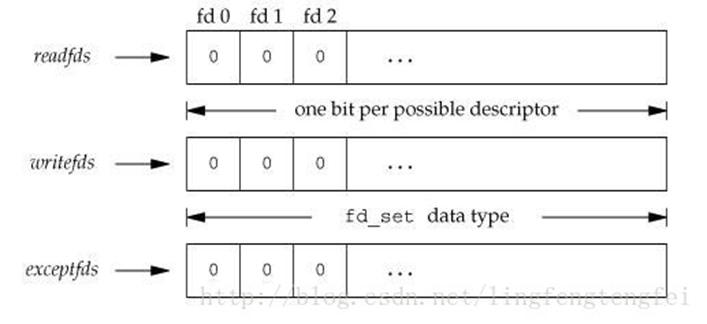
|  |
| --- |
| int select(int maxfdp1,fd\_set \*readset,fd\_set \*writeset,fd\_set \*exceptset,const struct timeval \*timeout) |

返回值：就绪描述符的数目，超时返回0，出错返回-1

函数参数介绍如下：

第一个参数maxfdp指定待测试的描述字个数，

中间的三个参数readset、writeset和exceptset指定我们要让内核测试读、写和异常条件的描述字。readset, writset, exceptset,指向描述符集。这些参数指明了我们关心哪些描述符，和需要满足什么条件(可写，可读，异常)。一个文件描述集保存在 fd\_set 类型中。fd\_set类型变量每一位代表了一个描述符。我们也可以认为它只是一个由很多二进制位构成的数组。如下图所示：



timeout告知内核等待所指定描述字中的任何一个就绪可花多少时间。其timeval结构用于指定这段时间的秒数和微秒数

（2）poll：

|  |
| --- |
| int poll(struct pollfd \*ufds, unsigned int nfds, int timeout); 2  struct pollfd {  int fd; //文件描述符  short events; //要求查询的事件掩码  short revents; //返回的事件掩码  }; |

返回值：就绪描述符的数目，超时返回0，出错返回-1

函数参数介绍如下：

ufds是要监控的文件句柄集合

nfds是监控的文件句柄数量

timeout是等待的毫秒数，这段时间内无论I/O是否准备好，poll都会返回

select和poll都需要在返回后，通过遍历文件描述符来获取已经就绪的socket。事实上，同时连接的大量客户端在一时刻可能只有很少的处于就绪状态，因此随着监视的描述符数量的增长，其效率也会线性下降。

（3）epoll：

|  |
| --- |
| int epoll\_create(int size)；//创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大 2  int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)； 3  int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout); |

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)；

函数是对指定描述符fd执行op操作。

- epfd：是epoll\_create()的返回值。

- op：表示op操作，用三个宏来表示：添加EPOLL\_CTL\_ADD，删除EPOLL\_CTL\_DEL，修改EPOLL\_CTL\_MOD。分别添加、删除和修改对fd的监听事件。

- fd：是需要监听的fd（文件描述符）

- epoll\_event：是告诉内核需要监听什么事

int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

函数返回已经处理的事件数目；

- epoll\_event：内核返回就绪的事件（该结构体中包含着文件描述符）

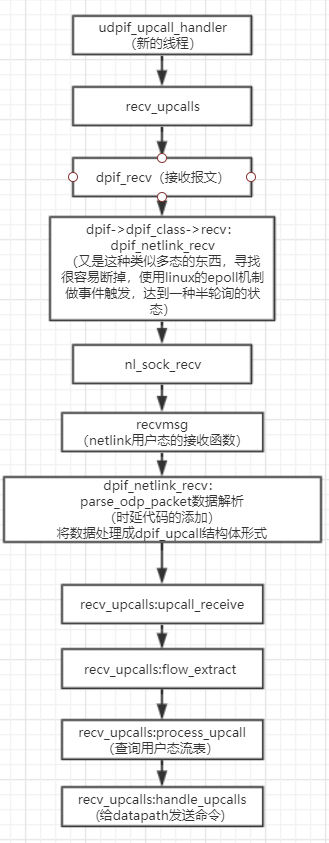
- timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）

**select/poll和epoll的区别**：

在 select/poll中，进程只有在调用一定的相应方法后，内核才对所有监视的文件描述符进行扫描，更新相应文件描述符的状态，而epoll事先通过epoll\_ctl()来注册一 个文件描述符，一旦基于某个文件描述符就绪时，内核会采用类似callback的回调机制，更新这个文件描述符的状态，当进程调用epoll\_wait() 时便得到通知。(此处去掉了遍历文件描述符，而是通过监听回调的的机制。)

## 用户态接收upcall和查流表过程

函数流程图：



**对涉及函数的解释：**

|  |
| --- |
| static void udpif\_start\_threads(struct udpif \*, size\_t n\_handlers,  size\_t n\_revalidators); |

参数：

udpif：处理upcall和更新内核流表的集合

n\_handlers\_:需要开启处理upcall的线程数量

n\_revalidators\_:需要开启处理revalidators的线程数量

描述：启动处理upcall和revalidator的线程

|  |
| --- |
| static void \*  udpif\_upcall\_handler(void \*arg) |

参数：

arg：传入一个handler结构体

|  |
| --- |
| /\* 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. \*/  }; |

描述：不断循环读取来自datapath的Generic Netlink消息，一次读取一组消息，并进行处理，下发相应的流到datapath中。

|  |
| --- |
| static size\_t  recv\_upcalls(struct handler \*handler) |

参数：

handler:

|  |
| --- |
| /\* 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. \*/  }; |

描述：读取一组来自datapath的Generic Netlink消息，并进行处理，下发相应的流到datapath中。

调用者：udpif\_upcall\_handler

|  |
| --- |
| int  dpif\_recv(struct dpif \*dpif, uint32\_t handler\_id, struct dpif\_upcall \*upcall,  struct ofpbuf \*buf)  {  int error = EAGAIN;  if (dpif->dpif\_class->recv) {  error = dpif->dpif\_class->recv(dpif, handler\_id, upcall, buf);  if (!error) {  dpif\_print\_packet(dpif, upcall);//将数据包以字符串输出  } else if (error != EAGAIN) {  log\_operation(dpif, "recv", error);  }  }  return error;  } |

参数：

dpif:数据包对应datapath的类型，使用对应的接收函数

|  |
| --- |
| /\* Open vSwitch datapath interface.  \*  \* This structure should be treated as opaque by dpif implementations. \*/  struct dpif {  const struct dpif\_class \*dpif\_class;//所属于的datapath接口的类型，"system", "netdev", etc.  char \*base\_name;  char \*full\_name;  uint8\_t netflow\_engine\_type;  uint8\_t netflow\_engine\_id;  long long int current\_ms;  }; |

handler\_id:区分不同的处理线程

upcall：需要将接收的数据解析到此结构体（有相应的Generic Netlink数据包的attribute成员指针，如key）

buf：需要将接收的数据填首先填到此结构体，（此结构体一般用来作为openflow数据包载体）

返回：0表示成功，其他值表示错误，如EAGAIN（linux编程的错误类型）

描述：接收一个Generic Netlink消息，并填写dpif\_upcall和ofbuf结构体。

|  |
| --- |
| static int  dpif\_netlink\_recv(struct dpif \*dpif\_, uint32\_t handler\_id,  struct dpif\_upcall \*upcall, struct ofpbuf \*buf)  {  struct dpif\_netlink \*dpif = dpif\_netlink\_cast(dpif\_);//将dpif转化为dpif\_netlink,dpif是dpif\_netlink的一个成员  int error;  fat\_rwlock\_rdlock(&dpif->upcall\_lock);  #ifdef \_WIN32  error = dpif\_netlink\_recv\_windows(dpif, handler\_id, upcall, buf);  #else  error = dpif\_netlink\_recv\_\_(dpif, handler\_id, upcall, buf);  #endif  fat\_rwlock\_unlock(&dpif->upcall\_lock);  return error;  } |

@ 参数：

dpif\_:表明datapath Interface的类型

handler\_id:区分不同的处理线程

upcall：需要将接收的数据解析到此结构体（有相应的Generic Netlink数据包的attribute成员指针，如key）

buf：需要将接收的数据填首先填到此结构体，（此结构体一般用来作为openflow数据包载体）

@ 描述：接收一个Generic Netlink消息，并填写dpif\_upcall和ofbuf结构体，通过dpif成员变量，获取dpif\_netlink结构体指针

@ 返回值：返回0表示成功

|  |
| --- |
| static int  dpif\_netlink\_recv\_\_(struct dpif\_netlink \*dpif, uint32\_t handler\_id,  struct dpif\_upcall \*upcall, struct ofpbuf \*buf) |

@ 参数：

dpif\_:netlink类型的dapath接口，其中包含了epoll中监听的事件（epoll事件从哪来的）

handler\_id:区分不同的处理线程，记录相应线程处理的事件

upcall：需要将接收的数据解析到此结构体（有相应的Generic Netlink数据包的attribute成员指针，如key）

buf：需要将接收的数据填首先填到此结构体，（此结构体一般用来作为openflow数据包载体）

@ 描述：接收一个Generic Netlink消息，并填写dpif\_upcall和ofbuf结构体。更新dpif->handlers中相应线程处理的epoll事件

@ 返回值：返回0表示成功

|  |
| --- |
| int nl\_sock\_recv(struct nl\_sock \*, struct ofpbuf \*, int \*nsid, bool wait); |

@ 参数：

sock:保存端口号，文件描述符

buff:信息存储位置

nsid：没看懂是什么

wait：是否阻塞接收

@ 描述：调用socket中的recvmsg函数，接收一个Generic Netlink消息，并存放到ofbuff中。

@ 返回值：返回0表示成功

|  |
| --- |
| static int  parse\_odp\_packet(const struct dpif\_netlink \*dpif, struct ofpbuf \*buf,  struct dpif\_upcall \*upcall, int \*dp\_ifindex) |

@ 参数：

dpif: netlink类型的dapath接口，其中包含了epoll中监听的事件（epoll事件从哪来的）

buff：存储数据的地方

ucall：将buff的数据解析成upcall的形式

dp\_ifindex: 取出Generic Netlink的用户自定义头中的变量

@ 描述：将ofbuff中的数据解析成upcall的形式，本质就是将各个attribute起始地址赋值到upcall中相应的指针。upcall时延获取的代码就是在这个函数上面添加。

@ 返回值：返回0表示成功

|  |
| --- |
| static int  process\_upcall(struct udpif \*udpif, struct upcall \*upcall,  struct ofpbuf \*odp\_actions, struct flow\_wildcards \*wc) |

@ 参数：

udpif：处理upcall和更新内核流表的集合

upcall:代表需要处理的upcall，保存upcall的相关信息，比如flow key等

odp\_actions:

wc:

@ 描述：根据不同的upcall类型执行不同的操作

@ 返回值：返回0表示成功

@ 调用者：recv\_upcalls函数

|  |
| --- |
| static void  upcall\_xlate(struct udpif \*udpif, struct upcall \*upcall,  struct ofpbuf \*odp\_actions, struct flow\_wildcards \*wc) |

@ 参数：

udpif：处理upcall和更新内核流表的集合

upcall:代表需要处理的upcall，保存upcall的相关信息，比如flow key等

odp\_actions:

wc:

@ 返回值：返回0表示成功

@ 调用者：process\_upcall函数

@ 描述：这个函数没这么看懂，只能了解到它总是会调用xlate\_action 函数。

|  |
| --- |
| enum xlate\_error  xlate\_actions(struct xlate\_in \*xin, struct xlate\_out \*xout) |

@ 参数：

xin：

xout:

@ 返回值：返回0表示成功

@ 描述: 该函数会调用查找流表函数，并生成相应的action，生成action部分比较复杂还有待研究。

@ 调用者：process\_upcall函数

|  |
| --- |
| static void  handle\_upcalls(struct udpif \*udpif, struct upcall \*upcalls,  size\_t n\_upcalls) |

@ 参数：

@ 返回值：返回0表示成功

@ 描述：process\_upcall函数已经将相应的动作都存在upcall结构体中，handle\_upcalls执行相应的操作,向内核态发送命令，一共涉及四个命令DPIF\_OP\_FLOW\_PUT、 DPIF\_OP\_FLOW\_DEL、 DPIF\_OP\_EXECUTE、 DPIF\_OP\_FLOW\_GET，这就跟内核态中的ovs\_flow\_cmd\_set() 、ovs\_flow\_cmd\_get()、ovs\_flow\_cmd\_del()和ovs\_execute\_action()联系起来了。

@ 调用者：recv\_upcalls函数

## 在OVS源码中用户态记录的信息如何输出

对于用户态数据，我使用linux的syslog（Ubuntu为rsyslog）日志系统进行输出。步骤为：

1. 修改/etc/rsyslog.conf配置文件，添加规则

2. 在程序中添加输出代码

3. 重启日志系统 /etc/init.d/rsyslog restart

**修改/etc/rsyslog.conf配置文件，添加规则：**

|  |
| --- |
| Local4.\* /home/gary/gary.log |

让日志朝指定路径输出，可以通过添加一条新的规则实现。

每个规则行由两部分组成，selector部分和action部分，这两部分由一个或多个空格或tab分隔，selector部分指定源和日志等级，action部分指定对应的操作。selector也由两部分组成，设施和优先级，由点号.分隔。第一部分为消息源或称为日志设施，第二部分为日志级别。action是规则描述的一部分，规则用于处理消息。总的来说，消息内容被写到一种日志文件上，但也可以执行其他动作，比如写到数据库表中或转发到其他主机。

日志设施有：

auth(security), authpriv: 授权和安全相关的消息

kern: 来自Linux内核的消息

mail: 由mail子系统产生的消息

cron: cron守护进程相关的信息

daemon: 守护进程产生的信息

news: 网络消息子系统

lpr: 打印相关的日志信息

user: 用户进程相关的信息

local0 to local7: 保留，本地使用

日志级别有(升序)：

debug：包含详细的开发情报的信息，通常只在调试一个程序时使用。

info：情报信息，正常的系统消息，比如骚扰报告，带宽数据等，不需要处理。

notice： 不是错误情况，也不需要立即处理。

warning： 警告信息，不是错误，比如系统磁盘使用了85%等。

err：错误，不是非常紧急，在一定时间内修复即可。

crit：重要情况，如硬盘错误，备用连接丢失。

alert：应该被立即改正的问题，如系统数据库被破坏，ISP连接丢失。

emerg：紧急情况，需要立即通知技术人员。

**在程序中添加输出代码：**

|  |
| --- |
| openlog("vSwitch",LOG\_PID,LOG\_LOCAL4);  syslog(LOG\_DEBUG, ""); |

void openlog(const char \*ident, int option, int facility);

描述：此函数用来打开一个到系统日志记录程序的连接，打开之后就可以用syslog或vsyslog函数向系统日志里添加信息了（进程之间通信的通道）

参数：

ident ：表示信息的来源，会被加至每则记录消息中。ident 一般是程序的名称（例如 ，cron ，ine 等）

option：用于指定openlog函数和接下来调用的syslog函数的控制标志。

facility:这个要与rsyslogd守护进程的配置文件中的日志设施对应，日志信息会写入syslog.conf文件指定的位置。

void syslog(int priority, const char \*format, ...);

描述：通过连接向日志系统添加消息

参数：

priority：日志等级。如LOG\_DEBUG, LOG\_INFO(对应配置文件中的日志等级)

format：要输出到日志的消息。

void closelog(void);

描述：关闭openlog打开的连接。