Openvswitch(OVS)源代码分析系列

转载请注明原文出处，原文地址为：
http://blog.csdn.net/yuzhihui_no1/article/details/39161515

openVswitch（OVS）源代码分析之简介

云计算是现在 IT 行业比较流行的，但真正什么是云计算业界也没有个什么统一的定义（很多公司都是根据自己的利益狭隘的定义云计算），更别说什么标准规范了。所以现在就有很多人说云计算只不过是个幌子，是个噱头，没点实用的，嘴上说说而已，虽然我也不太清楚什么叫做云计算，云计算的定义究竟是什么，但我根据我公司现在做的云计算产品来说，对于云计算服务还是懂些的。我觉得那并不是什么幌子、噱头，但如果说这云计算技术还不太成熟，我倒还勉强认可的。若把云计算比作一个人的话，我个人觉得现在它正是二十岁的样子，到三十多岁就算是比较成熟了，所以大概就能想象的到云计算现在的境况了。下面就来简介下实现云计算的一些技术，我对云计算并没有什么研究，也没能达到从全局的角度来分析云计算技术，更别说从一些更高的位置来分析问题，我所能介绍的仅仅是我一个小程序员在工作中所遇到的一些和云计算有关的技术，日积月累，希望终有一天能成为云计算“砖家”。

云计算是个全世界的话题，所以也有全世界的能人异士来为实现这个云计算而奋斗。我现阶段遇到的有关云计算的技术就是 openVswitch、openStack 技术和 Docker 技术。那就先从 openVswitch 开始介绍起，我会用一系列 blog 来分析 openVswitch 的相关数据结构和工作流程，以及各个重要模块的分析。所有的介绍都是基于源码的分析，希望对初学着有点用。

openVswitch，根据其名就可以知道这是一个开放的虚拟交换机（open virtual switch）；它是实现网络虚拟化 SDN 的基础，它是在开源的 Apache2.0 许可下的产品级质量的多层虚拟交换标准。设计这个 openVswitch 的目的是为了解决物理交换机存在的一些局限性：openVswitch 较物理交换机而言有着更低的成本和更高的工作效率；一个虚拟交换机可以有几十个端口来连接虚拟机，而 openVswitch 本身占用的资源也非常小；可以根据自己的选

择灵活的配置，可以对数据包进行接收分析处理；同时还支持标准的管理接口和协议，如 NetFlow， sFlow， SPAN， RSPAN 等。

Open vSwtich 模块介绍

当前最新代码包主要包括以下模块和特性：

ovs-vswitchd 主要模块，实现 switch 的 daemon，包括一个支持流交换的 Linux 内核模块；

ovsdb-server 轻量级数据库服务器，提供 ovs-vswitchd 获取配置信息；

ovs-brcompatd 让 ovs-vswitch 替换 Linux bridge，包括获取 bridge ioctls 的 Linux 内核模块；

ovs-dpctl 用来配置 switch 内核模块；

一些 Scripts and specs 辅助 OVS 安装在 Citrix XenServer 上，作为默认 switch；

ovs-vsctl 查询和更新 ovs-vswitchd 的配置；

ovs-appctl 发送命令消息，运行相关 daemon；

ovsdbmonitor GUI 工具，可以远程获取 OVS 数据库和 OpenFlow 的流表。

ovs-openflowd：一个简单的 OpenFlow 交换机；

ovs-controller：一个简单的 OpenFlow 控制器；

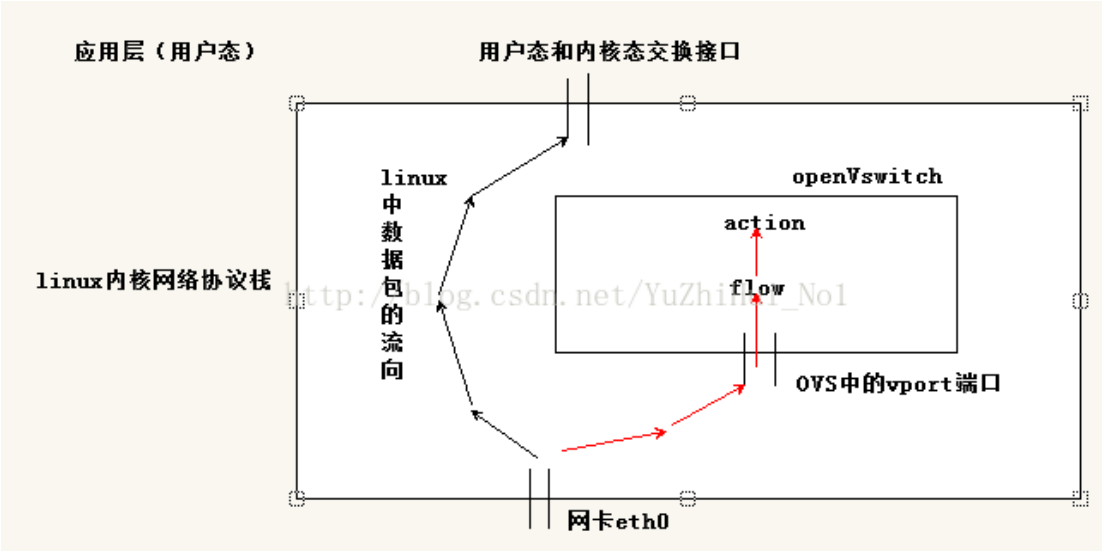
ovs-ofctl 查询和控制 OpenFlow 交换机和控制器；

ovs-pki ：OpenFlow 交换机创建和管理公钥框架；

ovs-tcpundump：tcpdump 的补丁，解析 OpenFlow 的消息；

上面是网上提到的一些 openVswitch 的主要模块。其实 openVswitch 中最主要的还是 datapath 目录下的一些文件。有端口模块 vport 等，还有关键的逻辑处理模块 datapath 等，以及 flow 等流表模块，最后的还有 action 动作响应模块，通道模块等等。

下面来介绍下其工作流程：



一般的数据包在 linux 网络协议栈中的流向为黑色箭头流向：从网卡上接受到数据包后层层往上分析，最后离开内核态，把数据传送到用户态。当然也有些数据包只是在内核网络协议栈中操作，然后再从某个网卡发出去。

但当其中有 openVswitch 时，数据包的流向就不一样了。首先是创建一个网桥：ovs-vsctl add-br br0；然后是绑定某个网卡：绑定网卡：ovs-vsctl add-port br0 eth0；这里默认为绑定了 eth0 网卡。数据包的流向是从网卡 eth0 上然后到 openVswitch 的端口 vport 上进入 openVswitch 中，然后根据 key 值进行流表的匹配。如果匹配成功，则根据流表中对应的 action 找到其对应的操作方法，完成相应的动作（这个动作有可能是把一个请求变成应答，也有可能是直接丢弃，也可以自己设计自己的 action）；如果匹配不成功，则执行默认的动作，有可能是放回内核网络协议栈中去处理（在创建网桥时就会相应的创建一个端口连接内核协议栈的）。

其大概工作流程就是这样了，在工作中一般在这几个地方来修改内核代码以达到自己的目的：第一个是在 datapath.c 中的 ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb)函数内添加相应的代码来达到自己的目的，因为对于每个数据包来说这个函数都是必经之地；第二个就是自己去设计自己的流表了；第三个和第二个是相关联的，就是根据流表来设计自己的 action，完成自己想要的功能。

openVswitch（OVS）源代码分析之数据结构

记得 Pascal 之父、结构化程序设计的先驱 Niklaus Wirth 最著名的一本书，书名叫作《算法 + 数据结构 = 程序》。还有位传奇的软件工程师 Frederick P. Brooks 曾经说过："给我看你的数据"。因此可见数据结构对于一个程序来说是多么的重要，如果你不了解程序中的数据结构，你根本就无法去理解整个程序的工作流程。所以在分析 openVswitch（OVS）源代码之前先来了解下 openVswitch 中一些重要的数据结构，这将对你分析后面的源代码起着至关重要的作用。

按照数据包的流向来分析下涉及到一些重要的数据结构。

第一、vport 端口模块中涉及到的一些数据结构：

[cpp] view plain copy

```cpp
//  这是表示网桥中各个端口结构体
struct  vport  {
        struct  rcu_head  rcu;  //  一种锁机制
        struct  datapath  *dp;  //  网桥结构体指针，表示该端口是属于哪个网桥的
        u32  upcall_portid;  //  Netlink 端口收到的数据包时使用的端口 id
        u16  port_no;  //  端口号，唯一标识该端口

//  因为一个网桥上有多个端口，而这些端口都是用哈希链表来存储的，
//  所以这是链表元素（里面没有数据，只有 next 和 prev 前驱后继指针，数据部分就是 vport 结构体中的其他成员）
        struct  hlist_node  hash_node;
        struct  hlist_node  dp_hash_node;  //  这是网桥的哈希链表元素
        const  struct  vport_ops  *ops;  //  这是端口结构体的操作函数指针结构体，结构体里面存放了很多操作函数的函数指针

        struct  pcpu_tstats  __percpu  *percpu_stats;//  vport 指向每个 cpu 的统计数据使用和维护

        spinlock_t  stats_lock;  //  自旋锁，防止异步操作，保护下面的两个成员
        struct  vport_err_stats  err_stats;  //  错误状态（错误标识）指出错误 vport 使用和维护的统计数字
        struct  ovs_vport_stats  offset_stats;  //  添加到实际统计数据，部分原因是为了兼容
```

```
};

//  端口参数，当创建一个新的 vport 端口是要传入的参数
struct  vport_parms  {
        const  char  *name;  //  新端口的名字
        enum  ovs_vport_type  type;  //  新端口的类型（端口不仅仅只有一种类型，后面会分析到）
        struct  nlattr  *options;  //  这个没怎么用到过，好像是从 Netlink 消息中得到的 OVS_VPORT_ATTR_OPTIONS 属性

        /*  For  ovs_vport_alloc().  */
        struct  datapath  *dp;  //  新的端口属于哪个网桥的
        u16  port_no;  //  新端口的端口号
        u32  upcall_portid;  //  和 Netlink 通信时使用的端口 id
};

//  这是端口 vport 操作函数的函数指针结构体，是操作函数的集合，里面存放了所有有关 vport 操作函数的函数指针
struct  vport_ops  {
        enum  ovs_vport_type  type;  //  端口的类型
        u32  flags;  //  标识符

        //  vport 端口模块的初始化加载和卸载函数
        int  (*init)(void);  //  加载模块函数，不成功则 over
        void  (*exit)(void);  //  卸载端口模块函数

        //  新 vport 端口的创建函数和销毁端口的函数
        struct  vport  *(*create)(const  struct  vport_parms  *);  //  根据指定的参数配置创建个新的 vport，成功返回新端口指针
        void  (*destroy)(struct  vport  *);  //  销毁端口函数

        //  得到和设置 option 成员函数
        int  (*set_options)(struct  vport  *,  struct  nlattr  *);
        int  (*get_options)(const  struct  vport  *,  struct  sk_buff  *);

        //  得到端口名称和配置以及发送数据包函数
        const  char  *(*get_name)(const  struct  vport  *);  //  获取指定端口的名称
        void  (*get_config)(const  struct  vport  *,  void  *);//  获取指定端口的配置信息
        int  (*get_ifindex)(const  struct  vport  *);//  获取系统接口和设备间的指数
```

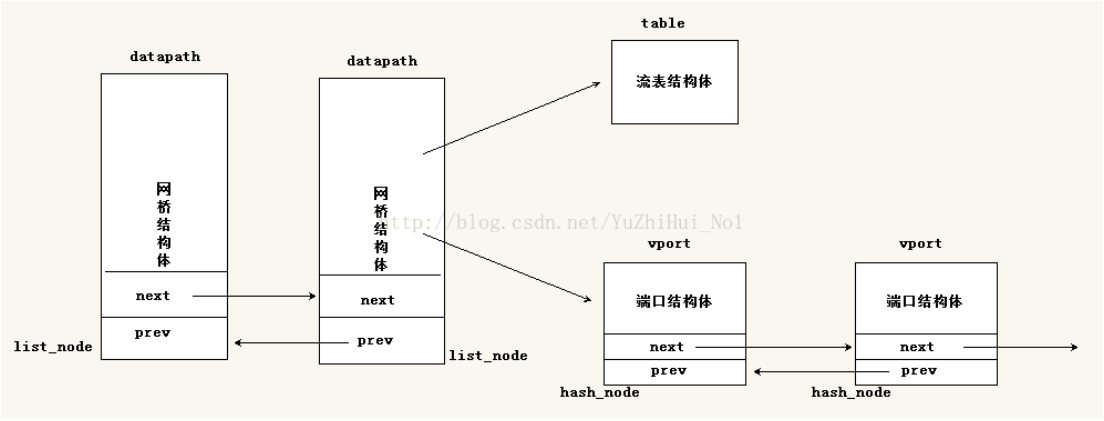
```
54.         int  (*send)(struct  vport  *,  struct  sk_buff  *);  //  发送数据包到设备上
55. };
56.
57. //  端口 vport 的类型，枚举类型存储
58. enum  ovs_vport_type{
59.         OVS_VPORT_TYPE_UNSPEC,
60.         OVS_VPORT_TYPE_NETDEV,
61.                 OVS_VPORT_TYPE_INTERNAL,
62.                 OVS_VPORT_TYPE_GRE,
63.                 OVS_VPORT_TYPE_VXLAN,
64.                 OVS_VPORT_TYPE_GRE64  =  104,
65.                 OVS_VPORT_TYPE_LISP  =  105,
66.         _OVS_VPORT_TYPE_MAX
67. };
```

第二、网桥模块 datapath 中涉及到的一些数据结构：

[cpp] view plain copy

```
1.  //  网桥结构体
2.  struct  datapath  {
3.          struct  rcu_head  rcu;  //  RCU 调延迟破坏。
4.          struct  list_head  list_node;  //  网桥哈希链表元素，里面只有 next 和 prev 前驱后继指针，数据时该结构体其他成员
5.
6.          /*  Flow  table.  */
7.          struct  flow_table  __rcu  *table;//  这是哈希流表，里面包含了哈希桶的地址指针。该哈希表受_rcu 机制保护
8.
9.          /*  Switch  ports.  */
10.         struct  hlist_head  *ports;//  一个网桥有多个端口,这些端口都是用哈希链表来链接的
11.
12.         /*  Stats.  */
13.         struct  dp_stats_percpu  __percpu  *stats_percpu;
14.
15. #ifdef  CONFIG_NET_NS
16.         /*  Network  namespace  ref.  */
17.         struct  net  *net;
18. #endif
19. };
```

其实上面的网桥结构也表示了整个 openVswitch（OVS）的结构，如果能捋顺这些结构的关系，那就对分析 openVswitch 源代码有很多帮助，下面来看下这些结构的关系图：



第三、流表模块 flow 中涉及到的一些数据结构：

[cpp] view plain copy

```
1.  //  可以说这是 openVswitch 中最重要的结构体了（个人认为）
2.  //  这是 key 值，主要是提取数据包中协议相关信息，这是后期要进行流表匹配的关键结构
3.  struct  sw_flow_key  {
4.              //  这是隧道相关的变量
5.          struct  ovs_key_ipv4_tunnel  tun_key;    /*  Encapsulating  tunnel  key.  */
6.          struct  {
7.              //  包的优先级
8.                  u32  priority;  //  包的优先级
9.                  u32  skb_mark;  //  包的 mark 值
10.                 u16  in_port;  //  包进入的端口号
11.         }  phy;  //  这是包的物理层信息结构体提取到的
12.         struct  {
13.                 u8          src[ETH_ALEN];  //  源 mac 地址
14.                 u8          dst[ETH_ALEN];  //  目的 mac 地址
15.                 __be16  tci;  //  这好像是局域网组号
16.                 __be16  type;  //  包的类型，即：是 IP 包还是 ARP 包
17.         }  eth;  //  这是包的二层帧头信息结构体提取到的
18.         struct  {
19.                 u8          proto;  //  协议类型  TCP：6；UDP：17；ARP 类型用低 8 位表示
20.                 u8          tos;  //  服务类型
21.                 u8          ttl;  //  生存时间，经过多少跳路由
22.                 u8          frag;  //  一种 OVS 中特有的 OVS_FRAG_TYPE_*.
23.         }  ip;  //  这是包的三层 IP 头信息结构体提取到的
```

```
24.             //  下面是共用体，有 IPV4 和 IPV6 两个结构，为了后期使用 IPV6 适应
25.         union  {
26.                 struct  {
27.                         struct  {
28.                                 __be32  src;  //  源 IP 地址
29.                                 __be32  dst;  //  目标 IP 地址
30.                         }  addr;  //  IP 中地址信息
31.                                                 //  这又是个共用体，有 ARP 包和 TCP 包（包含 UDP）两种
32.                         union  {
33.                                 struct  {
34.                                         __be16  src;  //  源端口，应用层发送数据的端口
35.                                         __be16  dst;  //  目的端口，也是指应用层传输数据端口
36.                                 }  tp;  //  TCP（包含 UDP）地址提取
37.                                 struct  {
38.                                         u8  sha[ETH_ALEN];  //  ARP 头中源 Mac 地址
39.                                         u8  tha[ETH_ALEN];  //  ARP 头中目的 Mac 地址
40.                                 }  arp;ARP 头结构地址提取
41.                         };
42.                 }  ipv4;
43.                                 //  下面是 IPV6 的相关信息，基本和 IPV4 类似，这里不讲
44.                 struct  {
45.                         struct  {
46.                                 struct  in6_addr  src;          /*  IPv6  source  address.  */
47.                                 struct  in6_addr  dst;          /*  IPv6  destination  address.  */
48.                         }  addr;
49.                         __be32  label;                      /*  IPv6  flow  label.  */
50.                         struct  {
51.                                 __be16  src;            /*  TCP/UDP  source  port.  */
52.                                 __be16  dst;            /*  TCP/UDP  destination  port.  */
53.                         }  tp;
54.                         struct  {
```

```
55.                             struct  in6_addr  target;  /*  ND  targe
   t  address.  */
56.                             u8  sll[ETH_ALEN];      /*  ND  source
   link  layer  address.  */
57.                             u8  tll[ETH_ALEN];      /*  ND  target
   link  layer  address.  */
58.                         }  nd;
59.                 }  ipv6;
60.         };
61. };
```

接下来要分析的数据结构是在网桥结构中涉及的的：struct flow_table __rcu *table;

[cpp] view plain copy

```
1.  //流表
2.  struct  flow_table  {
3.          struct  flex_array  *buckets;  //哈希桶地址指针
4.          unsigned  int  count,  n_buckets;  //  哈希桶个数
5.          struct  rcu_head  rcu;  //  rcu 包含机制
6.          struct  list_head  *mask_list;  //  struct  sw_flow_mask 链表头
   指针
7.          int  node_ver;
8.          u32  hash_seed;  //哈希算法需要的种子，后期匹配时要用到
9.          bool  keep_flows;  //是否保留流表项
10. };
```

顺序分析下去，应该是分析哈希桶结构体了，因为这个结构体设计的实在是太巧妙了。所以应该仔细的分析下。

这是一个共用体，是个设计非常巧妙的共用体。因为共用体的特点是：整个共用体的大小是其中最大成员变量的大小。也就是说 共用体成员中某个最大的成员的大小就是共用体的大小。正是利用这一点特性，最后一个 char padding[FLEX_ARRAY_BASE_SIZE]其实是没有用的，仅仅是起到一个占位符的作用了。让整个共用体的大小为 FLEX_ARRAY_BASE_SIZE（即是一个页的大小：4096），那为什么要这么费劲心机去设计呢？是因为 struct flex_array_part *parts[]; 这个结构，这个结构并不多见，因为在标准的 c/c++代码中是无效的，只有在 GNU 下才是合法的。这个称为弹性数组，或者可变数组，和常规的数组不一样。这里这个弹性数组的大小是一个页大小减去前面几个整型成员变量后所剩的大小。

[cpp] view plain copy

```
1.  //  哈希桶结构
2.  struct  flex_array  {
3.          //  共用体，第二个成员为占位符，为共用体大小
4.          union  {
5.                  //  对于这个结构体的成员数据含义，真是花了我不少时间来研究，发现有歧义，（到后期流表匹配时会详细分析）。现在就我认为最正确的理解来分析
6.                  struct  {
7.                          int  element_size;  //  无疑这是数组元素的大小
8.                          int  total_nr_elements;  //  这是数组元素的总个数
9.                          int  elems_per_part;  //  这是每个 part 指针指向的空间能存储多少元素
10.                         u32  reciprocal_elems;
11.                         struct  flex_array_part  *parts[];  //  结构体指针数组，里面存放的是 struct  flex_array_part 结构的指针
12.                 };
13.                 /*
14.                   *  This  little  trick  makes  sure  that
15.                   *  sizeof(flex_array)  ==  PAGE_SIZE
16.                   */
17.                 char  padding[FLEX_ARRAY_BASE_SIZE];
18.         };
19. };
20.
21. //  其实 struct  flex_array_part  *parts[];中的结构体只是一个数组而已
22. struct  flex_array_part  {
23.         char  elements[FLEX_ARRAY_PART_SIZE];  //  里面是一个页大小的字符数组
24. };
25.
26. //  上面的字符数组中存放的就是流表项头指针，流表项也是用双链表链接而成的
27. //流表项结构体
28. struct  sw_flow  {
29.         struct  rcu_head  rcu;  //  rcu 保护机制
30.         struct  hlist_node  hash_node[2];  //  两个节点指针，用来链接作用，前驱后继指针
31.         u32  hash;  //  hash 值
32.
33.         struct  sw_flow_key  key;  //  流表中的 key 值
34.         struct  sw_flow_key  unmasked_key;  //  也是流表中的 key
```

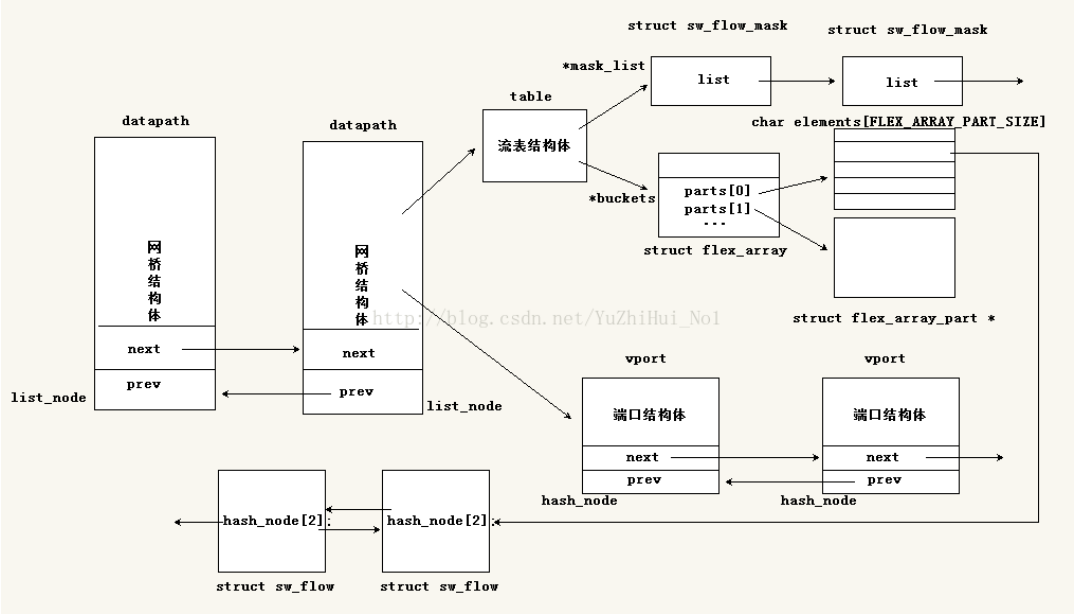
```
35.         struct  sw_flow_mask  *mask;  //  要匹配的 mask 结构体
36.         struct  sw_flow_actions  __rcu  *sf_acts;  //  相应的 action 动
    作
37.
38.         spinlock_t  lock;  //  保护机制自旋锁
39.         unsigned  long  used;  //  最后使用的时间
40.         u64  packet_count;  //  匹配过的数据包数量
41.         u64  byte_count;  //  匹配字节长度
42.         u8  tcp_flags;  //  TCP 标识
43. };
```

顺序下来，应该轮到分析 mask 结构体链表了：

[cpp] view plain copy

```
1.  //  这个 mask 比较简单，就几个关键成员
2.  struct  sw_flow_mask  {
3.          int  ref_count;
4.          struct  rcu_head  rcu;
5.          struct  list_head  list;//  mask 链表元素，因为 mask 结构是个双链
    表结构体
6.          struct  sw_flow_key_range  range;//  操作范围结构体，因为 key 值中
    有些数据时不要用来匹配的
7.          struct  sw_flow_key  key;//  要和数据包操作的 key，将要被用来匹配
    的 key 值
8.  };
9.
10. //  key 的匹配范围，因为 key 值中有一部分的数据时不用匹配的
11. struct  sw_flow_key_range  {
12.         size_t  start;  //  key 值匹配数据开始部分
13.         size_t  end;  //  key 值匹配数据结束部分
14. };
```

下面是整个 openVswitch 中数据结构所构成的图示，也是整个 openVswitch 中主要结构：



openVswitch（OVS）源代码分析之工作流程（收发数据包）

前面已经把分析 openVswitch 源代码的基础（openVswitch（OVS）源代码分析之数据结构）写得非常清楚了，虽然访问的人比较少，也因此让我看到了一个现象：第一篇，openVswitch（OVS）源代码分析之简介其实就是介绍了下有关于云计算现状和 openVswitch 的各个组成模块，还有笼统的介绍了下其工作流程，个人感觉对于学习 openVswitch 源代码来说没有多大含金量。云计算现状是根据公司发展得到的个人体会，对学习 openVswitch 源代码其实没什么帮助；openVswitch 各个组成模块到网上一搜一大堆，更别说什么含金量了；最后唯一一点还算过的去的就是 openVswitch 工作流程图，对从宏观方面来了解整个 openVswitch 来说还算是有点帮助的。但整体感觉对于学 openVswitch 源代码没有多少实质性的帮助，可是访问它的人就比较多。相反，第二篇，openVswitch（OVS）源代码分析之数据结构分析了整个 openVswitch 源代码中涉及到的主要数据结构，这可是花了我不少精力。它也是分析整个源代码的重要基础，更或者说可以把它当做分析 openVswitch 源代码的字典工具。可是访问它的人数却是少的可怜，为什么会这样呢？

网上有很多 blog 写有关于 openVswitch 的，但是绝大部分只是介绍 openVswitch 以及怎么安装配置它，或者是一些命令的解释。对于源代码的分析是非常少的，至少我开始学习 openVswitch 时在网上搜资料那会是这样的。因此对于一个开始接触学习 openVswitch 源代码的初学者来说是非常困难的，什么资料都没有（当然官网上还是有些资料得，如果你英文够好，看官网的资料也是个不错的选择），只得从头开始去分析，可是要想想 openVswitch 是由一个世界级的杰出团队花几年的时间设计而成的，如果要从零开始学习分析它，要到猴

年马月。所幸的是我开始学的时候，公司前辈们提供了些学习心得以及结构资料，所以在此我也把自己的学习心得和一些理解和大家分享。如有不正确之处，望大家指正，谢谢！！！

言归正传，基础已经学习过了，下面来正真分析下 openVswitch 的工作流程源代码。

首先是数据包的接受函数，这是在加载网卡时把网卡绑定到 openVswitch 端口上（ovs-vsctl add-port br0 eth0），绑定后每当有数据包过来时，都会调用该函数，把数据包传送给这个函数去处理。而不是像开始那样（未绑定前）把数据包往内核网络协议栈中发送，让内核协议栈去处理。openVswitch 中数据包接受函数为：void ovs_vport_receive(struct vport *vport, struct sk_buff *skb);函数，该函数所在位置为：datapath/vport.c 中。实现如下：

[cpp] view plain copy

```cpp
// 数据包接受函数，绑定网卡后，所有数据包都是从这个函数作为入口传入到 openVswitch 中去处理的，
// 可以说这是 openVswitch 的入口点。参数 vport：数据包从哪个端口进来的；参数 skb：数据包的地址指针
void ovs_vport_receive(struct vport *vport, struct sk_buff *skb)

{
        struct pcpu_tstats *stats; // 其实这个东西一直没弄明白，大概作用是维护 CPU 的锁状态

        stats = this_cpu_ptr(vport->percpu_stats); // 开始获取到 CPU 的锁状态，这和 linux 内核中的自旋锁类似
        u64_stats_update_begin(&stats->syncp); // 开始上锁
        stats->rx_packets++; // 统计数据包的个数
        stats->rx_bytes += skb->len; // 记录数据包中数据的大小
        u64_stats_update_end(&stats->syncp);// 结束锁状态

        if (!(vport->ops->flags & VPORT_F_TUN_ID)) // 这是种状态处理
                OVS_CB(skb)->tun_key = NULL;

// 其实呢这个函数中下面这行代码才是关键，如果不是研究 openVswitch 而是为了工作，个人觉得没必要（估计也不可能）
// 去弄清楚每条代码的作用。只要知道大概是什么意思，关键代码有什么作用，如果要添加自己的代码时，该往哪个地方添加就可以了。
// 下面这行代码是处理数据包的函数调用，是整个 openVswitch 的核心部分，传入的参数和接受数据包函数是一样的。
        ovs_dp_process_received_packet(vport, skb);
}
```

俗话说有接必有还，有进必有出嘛。上面的是数据包进入 openVswitch 的函数，那一定有其对应的出 openVswitch 的函数。数据包进入 openVswitch 后会调用函数

ovs_dp_process_received_packet(vport,skb);对数据包进行处理，到后期会分析到，这个函数对数据包进行流表的匹配，然后执行相应的 action。其中 action 动作会操作对数据包进行一些修改，然后再把数据包发送出去，这时就会调用 vport.c 中的数据包发送函数： ovs_vport_send(struct vport *vport, struct sk_buff *skb);来把数据包发送到端口绑定的网卡设备上去，然后网卡驱动就好把数据包中的数据发送出去。当然也有些 action 会把数据包直接向上层应用发送。下面来分析下数据包发送函数的实现，函数所在位置为：datapath/vport.c 中。

[cpp] view plain copy

```
1. //  这是数据包发送函数。参数 vport：指定由哪个端口发送出去；参数 skb：指定把哪个数据包发送出去
2. int  ovs_vport_send(struct  vport  *vport,  struct  sk_buff  *skb)
3. {
4. //  这是我自己加的代码，为了过滤掉 ARP 数据包。这里额外的插一句，不管在什么源代码中添加自己的代码时
5. //  都要在代码开头处做上自己的标识,因为这样不仅便于自己修改和调试、维护，而且也让其他人便于理解
6.                 /*==========yuzhihui:==========*/
7.         if  (0x806  ==  ntohs(skb->protocol))  {
8.                         arp_proc_send(vport,skb);//  自定义了一个函数处理了 ARP 数据包
9.         }
10.         //  在前篇数据结构中讲了 ops 是 vport 结构中的一些操作函数的函数指针集合结构体
11.                 //  所以 vport->ops->send() 是函数指针来调用函数，把数据包发送出去
12.         int  sent  =  vport->ops->send(vport,  skb);
13.
14.         if  (likely(sent))  {  //  定义了一个判断宏 likely()，如果发送成功执行下面
15.                 struct  pcpu_tstats  *stats;  //  下面的这些代码是不是觉得非常眼熟，没错就是接受函数中的那些代码
16.
17.                 stats  =  this_cpu_ptr(vport->percpu_stats);
18.
19.                 u64_stats_update_begin(&stats->syncp);
20.                 stats->tx_packets++;
21.                 stats->tx_bytes  +=  sent;
22.                 u64_stats_update_end(&stats->syncp);
23.         }
24.         return  sent;  //  返回的 sent 是已经发送成功的数据长度
25. }
```

这两个函数就是 openVswitch 中收发数据包函数了，对这两个函数没有完全去分析它的所有代码，这也不是我的本意，我只是想让初学者知道这是数据包进入和离开 openVswitch 的函数。其实知道了这个是非常有用的，因为不管你是什么数据包，只要是到该主机的（当然了包括主机内的各种虚拟机及服务器），全部都会经过这两个函数（对于接受的数据时一定要进过接受函数的，但是发送数据包有时候到不了发送函数的），那么要对数据包进行怎么样的操作那就全看你想要什么操作了。

在这两个函数中对数据包操作举例：

数据包接受函数中操作：如果你要阻断和某个 IP 主机间的通信（或者对某个 IP 主机数据包进行特殊处理），那么你可以在数据进入 openVswitch 的入口函数（ovs_vport_receive(struct vport *vport, struct sk_buff *skb)；）中进行处理，判断数据包中提取到的 IP 对比，如果是指定 IP 则把这个数据包直接销毁掉（也可以自己定义函数做些特殊操作）。这样就可以对整个数据进行控制。

数据包发送函数中操作：就像上面的函数中我自己写的那些代码一样，提取数据包中数据包类型进行判断，当判断如果是 ARP 数据包时，则调用我自定义的 arp_proc_send(vport,skb)；函数进行去处理，而不是贸然的直接把它发送出去，因为你不知道该数据包发送的端口是什么类型的。如果是公网 IP 端口，那么就在自定义函数中直接把这个数据包掐死掉（ARP 数据包是在局域网内作用的，就算发到公网上也会被处理掉的）；如果是发送到外层局域网中或者是相连的服务器中，则修改数据包中的目的 Mac 地址进行洪发；又如果是个 ARP 请求数据包，则把该数据包修改为应答包，再原路发送回去，等等情况；这些操作控制都是在发送数据包函数中做的手脚。

以上就是 openVswitch（OVS）工作流程中的数据包收发函数，经过大概的分析和应用举例说明，我想对于初学者来说应该知道大概在哪个地方添加自己的代码，实现自己的功能要求了。

openVswitch（OVS）源代码分析之工作流程（数据包处理）

上篇分析到数据包的收发，这篇开始着手分析数据包的处理问题。在 openVswitch 中数据包的处理是其核心技术，该技术分为三部分来实现：第一、根据 skb 数据包提取相关信息封装成 key 值；第二、根据提取到 key 值和 skb 数据包进行流表的匹配；第三、根据匹配到的流表做相应的 action 操作（若没匹配到则调用函数往用户空间传递数据包）；其具体的代码实现在 datapath/datapath.c 中的，函数为： void ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb);当接受到一个数据包后，自然而然的就应该是开始对其进行处理了。所以其实在上篇的 openVswitch（OVS）源代码分析之工作流程（收发数据包）中的接受数据包函数：void ovs_vport_receive(struct vport *vport, struct sk_buff *skb)中已有体现，该函数在最后调用了 ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb);来把数据包传递到该函数中去进行处理。也由此可见所有进入到 openVswitch 的数据包都必须经过 ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb);函数的处理。所以说 ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb);是整个 openVswitch 的中间枢纽，是 openVswitch 的核心部分。

对于 ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb);的重要性已经解释的非常清楚，紧接着就应该分析该函数源代码了，在分析源代码之前还是得提醒下，其中涉及到很多数据结构，如果有些陌生可以到 openVswitch（OVS）源代码分析之数据结构中进行查阅，最好能先大概的看下那文章，了解下其中的数据结构，对以后分析源代码有很大的帮助。

下面来分析几个 ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb);函数中涉及到但在 openVswitch（OVS）源代码分析之数据结构又没有分析到的数据结构：

第一个、是数据包的统计结构体，是 CPU 用来对所有数据的一个统计作用：

[cpp] view plain copy

```cpp
// CPU 对给定的数据包处理统计
struct dp_stats_percpu {
        u64 n_hit; // 匹配成功的数据包个数
```

```
4.          u64   n_missed;  //  匹配失败的数据包个数，n_hit  +  n_missed 就是接受到的数据包总和
5.          u64   n_lost;  //  丢失的数据包个数（可能是 datapath 队列溢出导致）
6.          struct  u64_stats_sync  sync;
7.  };
```

第二、是数据包发送到用户空间的参数结构体，在匹配流表没有成功时，数据将发送到用户空间。而内核空间和用户空间进行数据交互是通过 netLinks 来实现的，所以这个函数就是为了实现 netLink 通信而设置的一些参数：

[cpp] view plain copy

```
1.  //  把数据包传送给用户空间所需的参数结构体
2.  struct  dp_upcall_info  {
3.          u8   cmd;  //  命令，OVS_PACKET_CMD_  *之一
4.          const  struct  sw_flow_key  *key;  //  key 值，不能为空
5.          const  struct  nlattr  *userdata;  //  数据的大小，若为空，OVS_PACKET_ATTR_USERDATA 传送到用户空间
6.          u32   portid;  //  发送数据包的 Netlink 的 PID，其实就是 netLink 通信的 id 号
7.  };
```

下面是来分析 ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb);函数的实现源代码：

[cpp] view plain copy

```
1.  //  数据包的处理函数，openVswitch 的核心部分
2.  void  ovs_dp_process_received_packet(struct  vport  *p,  struct  sk_buff  *skb)
3.  {
4.          struct  datapath  *dp  =  p->dp;  //  定义网桥变量，得到端口所在的网桥指针
5.          struct  sw_flow  *flow;  //  流表
6.          struct  dp_stats_percpu  *stats;  //  cpu 中对数据包的统计
7.          struct  sw_flow_key  key;  //  skb 中提取到的 key 值
8.
9.
10.         u64  *stats_counter;
11.         int  error;
12.           //  这应该是 linux 内核中的，openVswitch 中很多是根据 linux 内核设计而来的
13.           //  这里应该是对网桥中的数据包统计属性进行初始化
```

```
14.         stats = this_cpu_ptr(dp->stats_percpu);
15.
16.           // 根据端口和 skb 数据包的相关值进行提取，然后封装成 key 值
17.         error = ovs_flow_extract(skb, p->port_no, &key);
18.         if (unlikely(error)) { // 定义宏来判断 key 值得提取封装是否成功
19.                 kfree_skb(skb);// 如果没有成功，则销毁掉 skb 数据包，然后直接退出
20.                 return;
21.         }
22.
23.           // 调用函数根据 key 值对流表中所有流表项进行匹配，把结果返回到 flow 中
24.         flow = ovs_flow_lookup(rcu_dereference(dp->table), &key);
25.         if (unlikely(!flow)) { // 定义宏判断是否匹配到相应的流表项，如没有，执行下面代码
26.                 struct dp_upcall_info upcall; // 定义一个结构体，设置相应的值，然后把数据包发送到用户空间
27.                 // 下面是根据 dp_upcall_info 数据结构，对其成员进行填充
28.                 upcall.cmd = OVS_PACKET_CMD_MISS;// 命令
29.                 upcall.key = &key;// key 值
30.                 upcall.userdata = NULL;// 数据长度
31.                 upcall.portid = p->upcall_portid;// netLink 通信时的 id 号
32.                 ovs_dp_upcall(dp, skb, &upcall);// 把数据包发送到用户空间
33.                 consume_skb(skb);// 销毁 skb
34.                 stats_counter = &stats->n_missed;// 用未匹配到流表项的包数，给计数器赋值
35.                 goto out;// goto 语句，内部跳转，跳转到 out 处
36.         }
37.               // 在分析下面的代码时，先看下 OVS_CB()这个宏：
#define OVS_CB(skb) ((struct ovs_skb_cb *)(skb)->cb)
38.               // 这个宏如果知道 skb 数据结构的话，就好理解。大概的意思是把 skb 中保存的当前层协议信息的数据强转为 ovs_skb_cb*数据指针
39.         OVS_CB(skb)->flow = flow;// 能够执行到这里，说明匹配到了流表。把匹配到的流表想 flow 赋值给结构体中成员
40.         OVS_CB(skb)->pkt_key = &key;// 同上，把相应的 key 值赋值到结构体变量中
41.             //    这是匹配成功的，用匹配成功的数据包数赋值于计数器变量
42.         stats_counter = &stats->n_hit;
```

```
43.         ovs_flow_used(OVS_CB(skb)->flow,  skb);//  调用函数调整流表项成员变量（也许是用来流表项的更新）
44.         ovs_execute_actions(dp,  skb);  //  根据匹配到的流表项（已经在skb 中的 cb）执行相应的 action 操作
45.
46. out:
47.           //  这是流表匹配失败，数据包发到用户空间后，跳转到该处，
48.           //  对处理过的数据包数进行调整（虽然没匹配到流表，但也算是处理掉了一个数据包，所以计数器变量应该增加 1）
49.         u64_stats_update_begin(&stats->sync);
50.         (*stats_counter)++;
51.         u64_stats_update_end(&stats->sync);
52. }
```

上面就是 openVswitch 的核心部分，所有的数据包都要经过此函数进行逻辑处理。这只是一个逻辑处理的大体框架，还有一些细节（key 值得提取，流表的匹配查询，数据传输到用户空间，根据流表执行相应 action）将在后面分析。当把整个 openVswitch 的工作流程梳理清晰，会发现这其实就是 openVswitch 的头脑部分，所有的逻辑处理都在里实现，所以我们自己添加代码时，这里往往也是个不错的选择。

如果看了前面那篇 openVswitch（OVS）源代码分析之工作流程（收发数据包），那么应该记得其中也说到了可以在收发函数中添加自己代码，因为一般来说收发函数也是数据包的必经之地（发送函数可能不是）。那么怎么区分在哪里添加自己代码合适呢？

其实在接受数据包函数中添加自己代码和在这里的逻辑处理函数中添加自己代码，没有多大区别，因为接受函数中没有做什么处理就把数据包直接发送打逻辑处理函数中去了，所以这两个地方添加自己代码其实是没什么区别的。但是从习惯和规范来说，数据包接受函数只是根据条件控制数据包的接受，并不对数据包进行逻辑上的处理，也不会对数据包进行修改等操作。而逻辑处理函数是会对数据包进行某些逻辑上的处理。（最明显的是修改数据包内的数据，一般来说接受数据包函数中是不会对数据包内容修改的，但逻辑处理函数则有可能会去修改的）。

而在数据包发送函数中添加自己代码和逻辑函数中添加自己代码也有些区别，数据包发送函数其性质和接受函数一样，一般不会去修改数据包，而仅仅是根据条件判断该数据包是否发送而已。

那下面就逻辑处理函数中添加代码来举例：

假若要把某个指定的 IP 主机上发来的 ARP 数据包进行处理，把所有的请求数据包变成应答数据包，原路返回。这里最好就是把自己的代码添加到逻辑处理函数中去（如果你要强制的添加到数据包接受函数中去也可以），因为这里要修改数据包的内容，是一个逻辑处理。具体实现：可以在 key 值提取前对数据包进行判断，看是否是 ARP 数据包，并且是否是指定 IP 主机发来的。若不是，交给系统去处理；若是，则对 Mac 地址和 IP 地址进行交换，并且把请求标识变成应答标识；最后调用发送函数从原来的端口直接发送出去。这只是一个简单的应用，旨在说明逻辑处理代码最好添加到逻辑处理函数中去。如果要处理复杂的操作也是可以的，比如定义自己的流表，然后然后屏蔽掉系统的流表查询，按自己的流表来操作。这就是一个对 openVswitch 比较大的改造了，流表、action、流表匹配等这些 openVswitch 主要功能和结构都要自己去定义实现完成。

以上分析得就是 openVswitch 的核心部分，当然了只是一个大体框架而已，后续将会逐步完善。

openVswitch（OVS）源代码分析之工作流程（key 值得提取）

其实想了很久要不要去分析下 key 值得提取，因为 key 值的提取是比较简单的，而且没多大实用。因为你不可能去修改 key 的结构，也不可能去修改 key 值得提取函数（当然了除非你想重构 openVswitch 整个项目），更不可能在 key 提取函数中添加自己的代码。因此对于分析 key 值没有多大的实用性。但我依然去简单分析 key 值得提取函数，有两个原因：第一、key 值作为数据结构在 openVswitch 中是非常重要的，后期的一些流表查询和匹配都要用到 key 值；第二、想借机复习下内核网络协议栈的各层协议信息；

首先来看下各层协议的协议信息：

第一、二层帧头信息

[cpp] view plain copy

```
1. struct ethhdr {
2.         unsigned char      h_dest[ETH_ALEN];      /*目标 Mac 地址  6 个字节*/
3.         unsigned char      h_source[ETH_ALEN];  /*源 Mac 地址*/
4.         __be16             h_proto;               /*包的协议类型  IP 包：0x800；ARP 包：0x806；IPV6：0x86DD*/
5. } __attribute__((packed));
6. /*从 skb 网络数据包中获取到帧头*/
7. static inline struct ethhdr *eth_hdr(const struct sk_buff *skb)
8. {
9.         return (struct ethhdr *)skb_mac_header(skb);
10. }
```

第二、三层网络层 IP 头信息

[cpp] view plain copy

```
1. /*IPV4 头结构体*/
2. struct iphdr {
3. #if defined(__LITTLE_ENDIAN_BITFIELD)
4.         __u8        ihl:4,  //  报文头部长度
5.                     version:4;  //  版本 IPv4
6. #elif defined (__BIG_ENDIAN_BITFIELD)
7.         __u8        version:4,
8.                     ihl:4;
9. #else
```

```
#error    "Please  fix  <asm/byteorder.h>"
#endif
        __u8        tos;                //  服务类型
        __be16    tot_len;          //  报文总长度
        __be16    id;                 //  标志符
        __be16    frag_off;       //  片偏移量
        __u8        ttl;                //  生存时间
        __u8        protocol;       //  协议类型  TCP：6；UDP：17
        __sum16  check;            //  报头校验和
        __be32    saddr;            //  源 IP 地址
        __be32    daddr;            //  目的 IP 地址
        /*The  options  start  here.  */
};

#ifdef  __KERNEL__
#include  <linux/skbuff.h>
/*通过数据包 skb 获取到 IP 头部结构体指针*/
static  inline  struct  iphdr  *ip_hdr(const  struct  sk_buff  *skb)
{
        return  (struct  iphdr  *)skb_network_header(skb);
}
/*通过数据包 skb 获取到二层帧头结构体指针*/
static  inline  struct  iphdr  *ipip_hdr(const  struct  sk_buff  *skb)

{
        return  (struct  iphdr  *)skb_transport_header(skb);
}
```

第三、ARP 协议头信息

[cpp] view plain copy

```
struct  arphdr
{
        __be16          ar_hrd;         /*  format  of  hardware  address 硬件类型      */
        __be16          ar_pro;         /*  format  of  protocol  address 协议类型      */
        unsigned  char      ar_hln;          /*  length  of  hardware  address 硬件长度      */
        unsigned  char      ar_pln;          /*  length  of  protocol  address 协议长度      */
        __be16          ar_op;          /*  ARP  opcode  (command)操作，请求：1；应答：2；*/

#if  0  //下面被注释掉了，使用时要自己定义结构体
```

```
10.          /*
11.            *    Ethernet  looks  like  this  :  This  bit  is  variable
     sized  however...
12.            */
13.          unsigned  char              ar_sha[ETH_ALEN];      /*  sender  h
   ardware  address 源 Mac    */
14.          unsigned  char              ar_sip[4];            /*  sender  IP
     address 源 IP          */
15.          unsigned  char              ar_tha[ETH_ALEN];      /*  target  h
   ardware  address 目的 Mac  */
16.          unsigned  char              ar_tip[4];            /*  target  IP
     address        目的 IP        */
17. #endif
18.
19. };
```

对于传输层协议信息 TCP/UDP 协议头信息比较多，这里就不分析了。下面直接来看 key 值提取代码：

[cpp] view plain copy

```
1.  int  ovs_flow_extract(struct  sk_buff  *skb,  u16  in_port,  struct  sw_
    flow_key  *key)
2.  {
3.          int  error;
4.          struct  ethhdr  *eth;  //帧头协议结构指针
5.
6.          memset(key,  0,  sizeof(*key));//  初始化 key 为 0
7.
8.          key->phy.priority  =  skb->priority;//赋值 skb 数据包的优先级
9.          if  (OVS_CB(skb)->tun_key)
10.                 memcpy(&key->tun_key,  OVS_CB(skb)->tun_key,  sizeof(key
    ->tun_key));
11.         key->phy.in_port  =  in_port;//  端口成员的设置
12.         key->phy.skb_mark  =  skb_get_mark(skb);//默认为 0
13.
14.         skb_reset_mac_header(skb);//该函数的实现
    skb->mac_header  =  skb->data;
15.
16.         /*  Link  layer.    We  are  guaranteed  to  have  at  least  th
    e  14  byte  Ethernet
17.           *  header  in  the  linear  data  area.
18.           */
19.         eth  =  eth_hdr(skb);  //获取到以太网帧头信息
```

```
20.         memcpy(key->eth.src,  eth->h_source,  ETH_ALEN);//  源地址成员赋值
21.         memcpy(key->eth.dst,  eth->h_dest,  ETH_ALEN);//  目的地址成员赋值
22.
23.         __skb_pull(skb,  2  *  ETH_ALEN);//这是移动 skb 结构中指针
24.
25.         if  (vlan_tx_tag_present(skb))//  数据包的类型判断设置
26.                 key->eth.tci  =  htons(vlan_get_tci(skb));
27.         else  if  (eth->h_proto  ==  htons(ETH_P_8021Q))//    协议类型设置
28.                 if  (unlikely(parse_vlan(skb,  key)))
29.                         return  -ENOMEM;
30.
31.         key->eth.type  =  parse_ethertype(skb);//包的类型设置，即是 IP 包还是 ARP 包
32.         if  (unlikely(key->eth.type  ==  htons(0)))
33.                 return  -ENOMEM;
34.
35.         skb_reset_network_header(skb);//  函数实现:skb->nh.raw  =  skb->data;
36.         __skb_push(skb,  skb->data  -  skb_mac_header(skb));//  移动 skb 中的指针
37.
38.         /*  Network  layer.  */
39.         //  判断是否是邋IP 数据包,如果是则设置 IP 相关字段
40.         if  (key->eth.type  ==  htons(ETH_P_IP))  {
41.                 struct  iphdr  *nh;//设置 IP 协议头信息结构体指针
42.                 __be16  offset;//  大端格式 short 类型变量
43.
44.                 error  =  check_iphdr(skb);//  检测 IP 协议头信息
45.                 if  (unlikely(error))  {
46.                         if  (error  ==  -EINVAL)  {
47.                                 skb->transport_header  =  skb->network_header;
48.                                 error  =  0;
49.                         }
50.                         return  error;
51.                 }
52.
53.                 nh  =  ip_hdr(skb);//  函数实现:return  (struct  iphdr  *)skb_network_header(skb);
54.                 //  下面就是 IP 协议头的一些字段的赋值
55.                 key->ipv4.addr.src  =  nh->saddr;
```

```
56.                 key->ipv4.addr.dst = nh->daddr;
57.
58.                 key->ip.proto = nh->protocol;
59.                 key->ip.tos = nh->tos;
60.                 key->ip.ttl = nh->ttl;
61.
62.                 offset = nh->frag_off & htons(IP_OFFSET);
63.                 if (offset) {
64.                         key->ip.frag = OVS_FRAG_TYPE_LATER;
65.                         return 0;
66.                 }
67.                 if (nh->frag_off & htons(IP_MF) ||
68.                           skb_shinfo(skb)->gso_type & SKB_GSO_UDP)
69.                         key->ip.frag = OVS_FRAG_TYPE_FIRST;
70.
71.                 /* Transport layer. */
72.                 if (key->ip.proto == IPPROTO_TCP) {
73.                         if (tcphdr_ok(skb)) {
74.                                 struct tcphdr *tcp = tcp_hdr(skb);
75.                                 key->ipv4.tp.src = tcp->source;
76.                                 key->ipv4.tp.dst = tcp->dest;
77.                         }
78.                 } else if (key->ip.proto == IPPROTO_UDP) {
79.                         if (udphdr_ok(skb)) {
80.                                 struct udphdr *udp = udp_hdr(skb);
81.                                 key->ipv4.tp.src = udp->source;
82.                                 key->ipv4.tp.dst = udp->dest;
83.                         }
84.                 } else if (key->ip.proto == IPPROTO_ICMP) {
85.                         if (icmphdr_ok(skb)) {
86.                                 struct icmphdr *icmp = icmp_hdr(skb)
    ;
87.                                 /* The ICMP type and code fields
    use the 16-bit
88.                                  * transport port fields, so we n
    eed to store
89.                                  * them in 16-bit network byte or
    der. */
90.                                 key->ipv4.tp.src = htons(icmp->type);
91.                                 key->ipv4.tp.dst = htons(icmp->code);
```

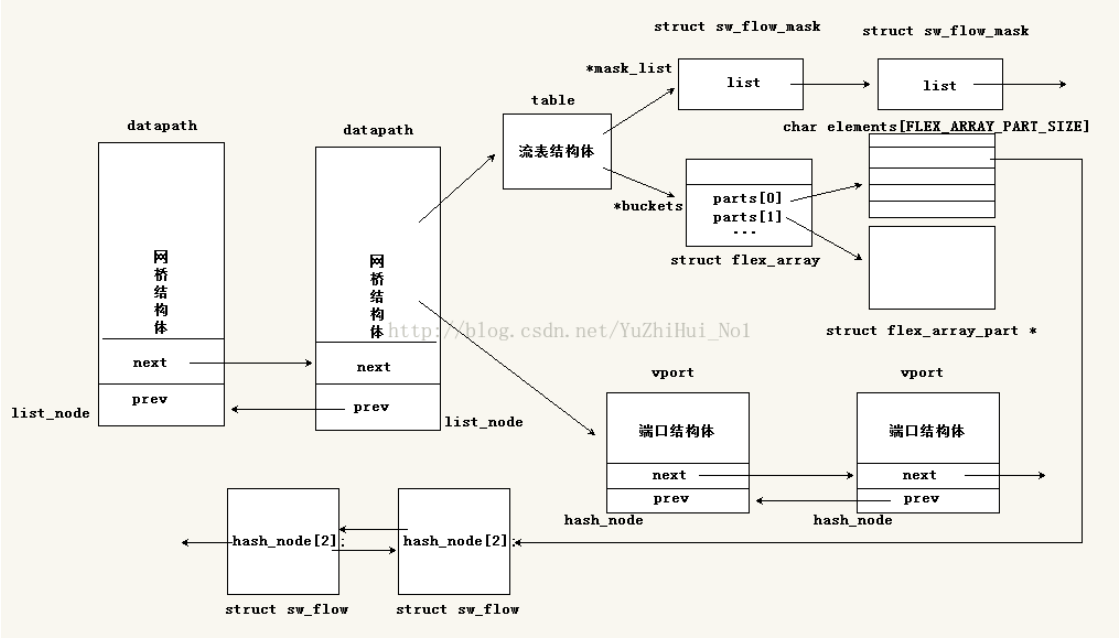
```
92.                         }
93.                 }
94.                 //  判断是否是 ARP 数据包，设置 ARP 数据包字段
95.         }  else  if  ((key->eth.type  ==  htons(ETH_P_ARP)  ||
96.                         key->eth.type  ==  htons(ETH_P_RARP))  &&  arphdr_
    ok(skb))  {
97.                 struct  arp_eth_header  *arp;  //  定义 ARP 协议头结构体
    指针
98.
99.
100.                 arp  =  (struct  arp_eth_header  *)skb_network_header(s
    kb);//  return  skb->nh.raw;
101.                 //  下面就是一些 ARP 数据包字段的设置
102.                 if  (arp->ar_hrd  ==  htons(ARPHRD_ETHER)
103.                                 &&  arp->ar_pro  ==  htons(ETH_P_IP)

104.                                 &&  arp->ar_hln  ==  ETH_ALEN
105.                                 &&  arp->ar_pln  ==  4)  {
106.
107.                         /*  We  only  match  on  the  lower  8  bits  o
    f  the  opcode.  */
108.                         if  (ntohs(arp->ar_op)  <=  0xff)
109.                                 key->ip.proto  =  ntohs(arp->ar_op);

110.                         memcpy(&key->ipv4.addr.src,  arp->ar_sip,  size
    of(key->ipv4.addr.src));
111.                         memcpy(&key->ipv4.addr.dst,  arp->ar_tip,  size
    of(key->ipv4.addr.dst));
112.                         memcpy(key->ipv4.arp.sha,  arp->ar_sha,  ETH_AL
    EN);
113.                         memcpy(key->ipv4.arp.tha,  arp->ar_tha,  ETH_AL
    EN);
114.                 }
115.                 //判断是否是 IPV6 数据包，设置 IPV6 数据包字段
116.         }  else  if  (key->eth.type  ==  htons(ETH_P_IPV6))  {
117.                 int  nh_len;                            /*  IPv6  Header
     +  Extensions  */
118.                 //  IPV6 就不分析了
119.                 nh_len  =  parse_ipv6hdr(skb,  key);
120.                 if  (unlikely(nh_len  <  0))  {
121.                         if  (nh_len  ==  -EINVAL)  {
122.                                 skb->transport_header  =  skb->network_
    header;
123.                                 error  =  0;
```

```
124.                        } else {
125.                                error = nh_len;
126.                        }
127.                        return error;
128.                }
129.
130.                if (key->ip.frag == OVS_FRAG_TYPE_LATER)
131.                        return 0;
132.                if (skb_shinfo(skb)->gso_type & SKB_GSO_UDP)
133.                        key->ip.frag = OVS_FRAG_TYPE_FIRST;
134.
135.                /* Transport layer. */
136.                if (key->ip.proto == NEXTHDR_TCP) {
137.                        if (tcphdr_ok(skb)) {
138.                                struct tcphdr *tcp = tcp_hdr(skb);
139.                                key->ipv6.tp.src = tcp->source;
140.                                key->ipv6.tp.dst = tcp->dest;
141.                        }
142.                } else if (key->ip.proto == NEXTHDR_UDP) {
143.                        if (udphdr_ok(skb)) {
144.                                struct udphdr *udp = udp_hdr(skb);
145.                                key->ipv6.tp.src = udp->source;
146.                                key->ipv6.tp.dst = udp->dest;
147.                        }
148.                } else if (key->ip.proto == NEXTHDR_ICMP) {
149.                        if (icmp6hdr_ok(skb)) {
150.                                error = parse_icmpv6(skb, key, nh_len);
151.                                if (error)
152.                                        return error;
153.                        }
154.                }
155.        }
156.
157.
158.        return 0;
159. }
```

openVswitch（OVS）源代码分析之工作流程（flow 流表查询）

前面分析了 openVswitch 几部分源代码，对于 openVswitch 也有了个大概的理解，今天要分析得代码将是整个 openVswitch 的重中之重。整个 openVswitch 的核心代码在 datapath 文件中；而 datapath 文件中的核心代码又在 ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb);函数中；而在 ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb);函数中的核心代码又是流表查询（流表匹配的）；有关于 ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb);核心代码的分析在前面的 openVswitch（OVS）源代码分析之工作流程（数据包处理）中。今天要分析得就是其核心中的核心：流表的查询（匹配流表项）源代码。我分析源代码一般是采用跟踪的方法，一步步的往下面去分析，只会跟着主线走（主要函数调用），对其他的分支函数调用只作大概的说明，不会进入其实现函数去分析。由于流表的查询设计到比较多的数据结构，所以建议对照着 openVswitch（OVS）源代码分析之数据结构去分析，我自己对数据结构已经大概的分析了遍，可是分析流表查询代码时还是要时不时的倒回去看看以前的数据结构分析笔记。

注：这是我写完全篇后补充的，我写完后自己阅读了下，发现如果就单纯的看源代码心里没有个大概的轮廓，不是很好理解，再个最后面的那个图，画的不是很好（我也不知道怎么画才能更好的表达整个意思，抱歉），所以觉得还是在这个位置（源代码分析前）先来捋下框架（也可以先看完源码分析再来看着框架总结，根据自己情况去学习吧）。上面已经说过了 openVswitch（OVS）源代码分析之数据结构的重要性，现在把里面最后那幅图拿来顺着图示来分析，会更好理解。（最后再来说下那幅图是真的非常有用，那相当于 openVswitch 的整个框架图了，如果你要分析源代码，有了那图绝对是事半功倍，希望阅读源代码的朋友重视起来，哈哈，绝不是黄婆卖瓜）



流表查询框架（或者说理论）：从 ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb)函数中开始调用函数查流表，怎么查呢？

第一步、它会根据网桥上的流表结构体（table）中的 mask_list 成员来遍历，这个 mask_list 成员是一条链表的头结点，这条链表是由 mask 元素链接组成（里面的 list 是没有数据的链表结构，作用就是负责链接多个 mask 结构，是 mask 的成员）；流表查询函数开始就是循环遍历这条链表，每遍历到得到一个 mask 结构体，就调用函数进入第二步。

第二步、是操作 key 值，调用函数让从数据包提取到的 key 值和第一步得到的 mask 中的 key 值，进行与操作，然后把结构存放到另外一个 key 值中（masked_key）。顺序执行第三步。

第三步、把第二步中得到的那个与操作后的 key 值（masked_key），传入 jhash2()算法函数中,该算法是经典的哈希算法,想深入了解可以自己查资料(不过都是些数学推理，感觉挺难的），linux 内核中也多处使用到了这个算法函数。通过这个函数把 key 值（masked_key）转换成 hash 关键字。

第四步、把第三步得到的 hash 值，传入 find_bucket()函数中，在该函数中再通过 jhash_1word()算法函数，把 hash 关键字再次哈希得到一个全新的 hash 关键字。这个函数和第三步的哈希算法函数类似，只是参数不同，多了一个 word。经过两个哈希算法函数的计算得到一个新的 hash 值。

第五步、 把第四步得到的 hash 关键字，传入到 flex_array_get()函数中，这个函数的作用就是找到对应的哈希头位置。具体的请看上面的图,流表结构(table)中有个 buckets

成员，该成员称作为哈希桶，哈希桶里面存放的是成员字段和弹性数组 parts[n]，而这个 parts[n]数组里面存放的就是所要找的哈希头指针,这个哈希头指针指向了一个流表项链表（在图中的最下面 struct sw_flow），所以这个才是我们等下要匹配的流表项。（这个哈希桶到弹性数组这一段，我有点疑问，不是很清楚，在下一篇 blog 中会分析下这个疑问，大家看到如果和源代码有出入，请按源代码来分析），这一步就是根据 hash 关键字查找到流表项的链表头指针。

第六步、由第五步得到的流表项链表头指针,根据这个指针遍历整个流表项节点元素(就是 struct sw_flow 结构体元素），每遍历得到一个流表项 sw_flow 结构体元素，就把流表项中的 mask 成员和第一步遍历得到的 mask 变量（忘记了可以重新回到第一步去看下）进行比较；比较完后还要让流表项 sw_flow 结构体元素中的 key 值成员和第二步中得到的 key 值（masked_key）进行比较；只有当上面两个比较都相等时，这个流表项才是我们要匹配查询的流表项了。然后直接返回该流表项的地址。查询完毕！！接下来分析源代码了。

在 ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb);函数中流表查询调用代码：

[cpp] view plain copy

```
1. //  ovs_flow_lookup()是流表查询实现函数；参数 rcu_dereference(dp->table)：网桥流表（不是流表项）；
2. //  参数&key：数据包各层协议信息提取封装的 key 地址;返回值 flow：查询到的或者说匹配到的流表项
3. flow  =  ovs_flow_lookup(rcu_dereference(dp->table),  &key);
```

这里要特别说明下：dp->table 是流表，是结构体 struct flow_table 的变量；而 flow 是流表项，是结构体 struct sw_flow 的变量;我们平常习惯性说的查询流表或者匹配流表，其实并不是说查询或者匹配 flow_table 结构体的变量(在 openVswitch 中 flow_table 没有链表，只有一个变量），而是 struct sw_flow 的结构体链表。所以确切的说应该是查询和匹配流表项。这两个结构是完全不同的，至于具体的是什么关系，有什么结构成员，可以查看下 openVswitch（OVS）源代码分析之数据结构。如果不想看那么繁琐的分析，也可以看下最后面的那张图，可以大概的了解下他们的关系和区别。

下面来着重的分析下 ovs_flow_lookup()函数，主要是循环遍历 mask 链表节点，和调用 ovs_masked_flow_lookup()函数。

[cpp] view plain copy

```
1.  //  tbl 是网桥结构中的 table,key 是包中提取的 key 的地址指针
2.  struct  sw_flow  *ovs_flow_lookup(struct  flow_table  *tbl,
3.                                       const  struct  sw_flow_key  *key)
4.  {
5.          struct  sw_flow  *flow  =  NULL;//  准备流表项，匹配如果匹配到流表项则返回这个变量
6.          struct  sw_flow_mask  *mask;//  了解 mask 结构就非常好理解 mask 的作用
7.                  //  下面是从网桥上的 table 结构中头 mask_list 指针开始遍历 mask 链表，
8.                  //  依次调用函数去查询流表，如果得到流表项则退出循环
9.          list_for_each_entry_rcu(mask,  tbl->mask_list,  list)  {
10.                 //  流表查询函数(其实是 key 的匹配),参数分别是:网桥的 table，和数据包的 key，以及网桥上的 mask 节点
11.                 flow  =  ovs_masked_flow_lookup(tbl,  key,  mask);
12.                 if  (flow)     /*  Found  */
13.                         break;
14.         }
15.         return  flow;
16. }
```

接下来是 flow = ovs_masked_flow_lookup(tbl, key, mask);函数的分析，该函数就是最后流表的比较了和查询了。主要实现功能是对 key 值得操作，和哈希得到哈希值，然后根据哈希值查找哈希头结点，最后在头结点链表中遍历流表项，匹配流表项。

[cpp] view plain copy

```
1.  //  table  是网桥结构体中成员，flow_key 是数据包提取到的 key 值，mask 是 table 中的 sw_flow_mask 结构体链表节点
2.  static  struct  sw_flow  *ovs_masked_flow_lookup(struct  flow_table  *table,
3.                                                  const  struct  sw_flow_key  *flow_key,
4.                                                  struct  sw_flow_mask  *mask)
5.  {
6.                  //  要返回的流表项指针，其实这里就可以断定后面如果查询成功的话，返回的是存储 flow 的动态申请好的内存空间地址，
7.                  //  因为这几个函数都是返回 flow 的地址的，如果不是动态申请的地址，返回的就是局部变量，那后面使用时就是非法的了。
8.                  //  它这样把地址一层一层的返回；若查询失败返回 NULL。因为这里不涉及到对流表的修改，只是查询而已，如果为了防止流表被更改，
9.          //  也可以自己动态的申请个 flow 空间，存储查询到的 flow 结构。
10.         struct  sw_flow  *flow;
11.         struct  hlist_head  *head;//  哈希表头结点
```

```
12.                 //  下面是 mask 结点结构体中成员变量：
    struct  sw_flow_key_range  range;
13.                 //  而该结构体 struct  sw_flow_key_range 有两个成员，分别为
    start 和 end
14.                 //  这两个成员是确定 key 值要匹配的范围，因为 key 值中的数据
    并不是全部都要进程匹配的
15.         int  key_start  =  mask->range.start;
16.         int  key_len  =  mask->range.end;
17.         u32  hash;
18.         struct  sw_flow_key  masked_key;//  用来得到与操作后的结果 key
    值
19.
20.         //  下面调用的 ovs_flow_key_mask()函数是让 flow_key(最开始的数据
    包中提取到的 key 值)，和 mask 中的 key 进行与操作(操作范围是
21.         //  在 mask 的 key 中 mask->range->start 开始，到 mask->range->end
    结束)  最后把与操作后的 key 存放在 masked_key 中
22.         ovs_flow_key_mask(&masked_key,  flow_key,  mask);
23.
24.         //  通过操作与得到的 key，然后再通过 jhash2 算法得到个 hash 值，其
    操作范围依然是  range->start  到 range->end
25.                 //  因为这个 key 只有在这段范围中数据时有效的，对于匹配操
    作来说。返回个哈希值
26.         hash  =  ovs_flow_hash(&masked_key,  key_start,  key_len);
27.
28.         //  调用 find_bucket 通过 hash 值查找 hash 所在的哈希头，为什么要查
    询链表头节点呢?
29.                 //  因为 openVswitch 中有多条流表项链表，所以要先查找出要
    匹配的流表在哪个链表中，然后再去遍历该链表
30.         head  =  find_bucket(table,  hash);
31.
32.                 //  重点戏来了，下面是遍历流表项节点。由开始获取到哈希链
    表头节点，依次遍历这个哈希链表中的节点元素，
33.                 //  把每一个节点元素都进行比较操作，如果成功，则表示查询
    成功，否则查询失败。
34.         hlist_for_each_entry_rcu(flow,  head,  hash_node[table->node_ver
    ])  {
35.                 //  下面都是比较操作了，如果成功则返回对应的 flow
36.                 if  (flow->mask  ==  mask  &&
37.                         __flow_cmp_key(flow,  &masked_key,  key_start,
    key_len))
38.                         //  上面的 flow 是流表项，masked_key 是与操作后的
    key 值，
39.                         return  flow;
40.         }
```

```
41.         return  NULL;
42. }
```

分析到上面主线部分其实已经分析完了，但其中有几个函数调用比较重要，不得不再来补充分析下。也是按顺序依次分析下来，

第一个调用函数：ovs_flow_key_mask(&masked_key, flow_key, mask);分析。这个函数主要功能是让数据包中提取到的 key 值和 mask 链表节点中 key 与操作，然后把结果存储到 masked_key 中。

[cpp] view plain copy

```
1.  //  要分析一个函数除了看实现代码外，分析传入的参数和返回的数据类型也是非常重要和有效的
2.  //  传入函数的参数要到调用该函数的地方去查找，返回值得类型可以在本函数内看到。
3.  //  上面调用函数：ovs_flow_key_mask(&masked_key,  flow_key,  mask);
4.  //  所以 dst 是要存储结构的 key 值变量地址，src 是最开始数据包中提取的 key 值，mask 是 table 中 mask 链表节点元素
5.  void  ovs_flow_key_mask(struct  sw_flow_key  *dst,  const  struct  sw_flow_key  *src,
6.                                 const  struct  sw_flow_mask  *mask)
7.  {
8.          u8  *m  =  (u8  *)&mask->key  +  mask->range.start;
9.          u8  *s  =  (u8  *)src  +  mask->range.start;
10.         u8  *d  =  (u8  *)dst  +  mask->range.start;
11.         int  i;
12.
13.         memset(dst,  0,  sizeof(*dst));
14.         //  ovs_sw_flow_mask_size_roundup()是求出 range->end  -  range->start 长度
15.         //  循环让最开始的 key 和 mask 中的 key 值进行与操作，放到目标 key 中
16.         for  (i  =  0;  i  <  ovs_sw_flow_mask_size_roundup(mask);  i++)  {
17.                 *d  =  *s  &  *m;
18.                 d++,  s++,  m++;
19.         }
20. }
```

第二个调用函数：hash = ovs_flow_hash(&masked_key, key_start, key_len);这个没什么好分析得，只是函数里面调用了个 jhash2 算法来获取一个哈希值。所以这个函数的整

体功能是：由数据包中提取到的 key 值和每个 mask 节点中的 key 进行与操作后得到的有效 masked_key，和 key 值的有效数据开始位置及其长度，通过 jhash2 算法得到一个 hash 值。

第三个调用函数：head = find_bucket(table, hash);分析，该函数实现的主要功能是对 hash 值调用 jhash_1word()函数再次对 hash 进行哈希，最后遍历哈希桶，查找对应的哈希链表头节点。

[cpp] view plain copy

```
1. //  参数 table:网桥上的流表;参数 hash:由上面调用函数获取到哈希值;返回
2. static struct hlist_head *find_bucket(struct flow_table *table, u3
   2 hash)
3. {
4.         //  由开始的 hash 值再和 table 结构中自带的哈希算法种子通过
   jhash_1word()算法，再次 hash 得到哈希值
5.                 //  不知道为什么要哈希两次，我个人猜测是因为第一次哈希的
   值，会有碰撞冲突。所以只能二次哈希来解决碰撞冲突
6.         hash = jhash_1word(hash, table->hash_seed);
7.
8.         //  传入的参数数:buckets 桶指针，哈希值 hash(相当于关键
   字)n_buckets 表示有多少个桶(其实相当于有多少个 hlist 头结点)
9.         //  hash&(table->n_buckets-1)表示根据 hash 关键字查找到第几个桶，
   其实相当于求模算法，找出关键字 hash 应该存放在哪个桶里面
10.        return flex_array_get(table->buckets,
11.                              (hash & (table->n_buckets - 1)));

12. }
```

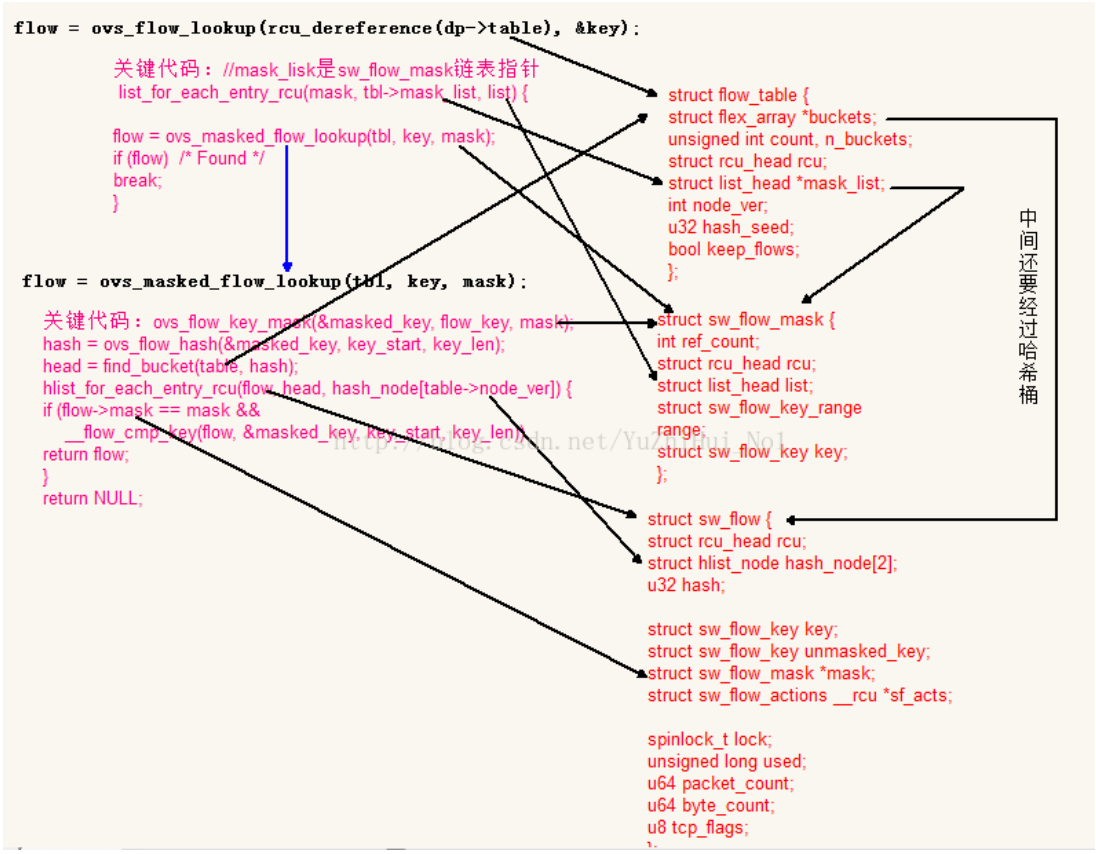
第四个调用函数： __flow_cmp_key(flow, &masked_key, key_start, key_len));分析，该函数实现的功能是对流表中的 key 值和与操作后的 key 值（masked_key）进行匹配。

[cpp] view plain copy

```
1. //  参数 flow：流表项链表节点元素；参数 key：数据包 key 值和 mask 链表节点
   key 值与操作后的 key 值；
2. //  参数 key_start:key 值得有效开始位置;参数 key_len:key 值得有效长度
3. static bool __flow_cmp_key(const struct sw_flow *flow,
4.                 const struct sw_flow_key *key, int key_start, int
     key_len)
5. {
6.         return __cmp_key(&flow->key, key, key_start, key_len);
7. }
8. //用数据包中提取到的key和mask链表节点中的key操作后的key值(masked_key)
   和流表项里面的 key 比较
```

```
9.  static  bool  __cmp_key(const  struct  sw_flow_key  *key1,
10.                 const  struct  sw_flow_key  *key2,    int  key_start,  i
    nt  key_len)
11. {
12.         return  !memcmp((u8  *)key1  +  key_start,
13.                         (u8  *)key2  +  key_start,  (key_len  -  key_sta
    rt));
14. }
```

上面就是整个流表的匹配实现源代码的分析，现在来整理下其流程，如下图所示（图有点丑见谅）：



到此流表查询就就算结束了，分析了遍自己也清晰了遍。

openVswitch（OVS）源代码的分析技巧（哈希桶结构体为例）

前言：

个人认为学习 openVswitch 源代码是比较困难的，因为该项目开发出来不久（当然了这是相对于其他开源项目），一些资料不完全，网上也比较少资料（一般都是说怎么使用，比如：怎么安装，怎么配置，以及一些命令等。对源码分析的资料比较少），如果遇到问题只有自己看源代码，分析源代码，然后各种假设，再带着假设去阅读分析源代码。而源代码中注释的也比较少（只有几部分函数和结构体有注释）。所以对源代码的分析是比较困难的。不像 linux（UNIX）方面的分析，因为 linux 项目已经开放出来好久了，资料漫天都是，各种内核源代码分析，只要你有耐心什么内核源代码分析，内核源代码全注释，都有各种详细版本供你参考。而我在学习 openVswitch 源代码的时候也遇到个比较大的问题，困扰了我比较久的时间,我也查看了几个模块的源代码发现都解决不了(因为各个模块中意思不能统一)，今天拿出来和大家一起来探讨下，也顺便分享下我分析源代码和处理问题的方法。

顺便说下，我原本是要顺着 ovs_dp_process_received_packet（）函数处理流程来分析所有的流程源代码，但分析完流表查询，觉得那个问题还是得处理下，所以在这里插入一篇 blog 来处理下那个问题。在之后会继续分析 upcall 和流表使用，action 动作等模块。

问题：

话不多说，还是看问题吧。其实在系列 blog 中的第二篇 openVswitch（OVS）源代码分析之数据结构已经说过。就是下面的结构体成员字段有歧义：

[cpp] view plain copy

```cpp
1.  //  哈希桶结构
2.  struct  flex_array  {
3.          //  共用体，第二个成员为占位符，为共用体大小
4.          union  {
5.                  //  对于这个结构体的成员数据含义，真是花了我不少时间来研究，发现有歧义，(到后期流表匹配时会详细分析)。现在就我认为最正确的理解来分析
6.                  struct  {
7.                          int  element_size;  //  无疑这是数组元素的大小
8.                          int  total_nr_elements;  //  这是数组元素的总个数
```

```
9.                          int  elems_per_part;  //  这是每个 part 指针指向的空间能存储多少元素
10.                         u32  reciprocal_elems;
11.                         struct  flex_array_part  *parts[];  //  结构体指针数组，里面存放的是 struct  flex_array_part 结构的指针
12.                 };
13.                 char  padding[FLEX_ARRAY_BASE_SIZE];
14.         };
15. };
16.
17. //  其实 struct  flex_array_part  *parts[];中的结构体只是一个数组而已
18. struct  flex_array_part  {
19.         char  elements[FLEX_ARRAY_PART_SIZE];  //  里面是一个页大小的字符数组
20. };
```

上面这个结构的分析还是引用下第二篇 blog 的原文吧：

“顺序分析下去，应该是分析哈希桶结构体了，因为这个结构体设计的实在是太巧妙了。所以应该仔细的分析下。

这是一个共用体，是个设计非常巧妙的共用体。因为共用体的特点是：整个共用体的大小是其中最大成员变量的大小。也就是说 共用体成员中某个最大的成员的大小就是共用体的大小。正是利用这一点特性，最后一个 char padding[FLEX_ARRAY_BASE_SIZE]其实是没有用的，仅仅是起到一个占位符的作用了。让整个共用体的大小为 FLEX_ARRAY_BASE_SIZE（即是一个页的大小：4096），那为什么要这么费劲心机去设计呢？是因为 struct flex_array_part *parts[]; 这个结构，这个结构并不多见，因为在标准的 c/c++代码中是无效的，只有在 GNU 下才是合法的。这个称为弹性数组，或者可变数组，和常规的数组不一样。这里这个弹性数组的大小是一个页大小减去前面几个整型成员变量后所剩的大小。”

分析：

这是前面 blog 中分析得，下面跟着源代码来分析下这个结构体成员变量的含义。

因为对这个哈希桶结构体有疑问，那么就到这个哈希桶结构内存申请的函数中去找，因为申请函数中肯定会有给结构体赋值的代码，只要弄清楚给结构体成员变量赋什么值就可以知道这个成员变量大概是什么意思了。

第一步

哈希桶结构的内存申请函数 struct flex_array *flex_array_alloc(int element_size, unsigned int total, gfp_t flags);

记得前面说过分析函数代码，除了看实现代码外，函数参数和返回值也是重要的线索.那么 element_size 和 total 究竟是什么意思呢？（gfp_t flags 这个参数学内核的大概都看过，就算没看过也能猜到是什么，就是在内核中内存申请的一些方式）， 要查看 element_size 和 total 是什么意思，就要查到谁（哪个函数）调用了 flex_array_alloc()函数。

第二步

于是在整个项目中进行搜索得，也可以理性的去分析下哪个函数会调用下面的 flex_array 申请函数，其实就是 buckets 内存申请函数（alloc_buckets）了；在该函数中有行代码为：buckets = flex_array_alloc(sizeof(struct hlist_head *),n_buckets, GFP_KERNEL);

至此已经找到了 element_size 的含义了，即：哈希头指针的大小。那么就剩下 total 参数了。

第三步

继续往上个查找，看看哪个函数调用了 alloc_buckets()函数，用搜索或者理性分析，哪个函数会调用 buckets 来申请函数，其实和上面类似，就是 table 内存申请函数（__flow_tbl_alloc）了。

在_flow_tbl_alloc()函数中有行调用代码为：table->buckets = alloc_buckets(new_size)；到这里还是没有查找到 total 究竟是什么意思，只知道是 new_size 传过来的。

第四步

不要放弃，继续往上查找调用函数中，也是类似的方法，整个文件去搜索 flow_tbl_alloc（_flow_tbl_alloc 是基础函数，被包装后为 flow_tbl_alloc）看看哪个函数调用了它。

最后搜索发现在 static int flush_flows(struct datapath *dp)函数有行调用代码：new_table = ovs_flow_tbl_alloc(TBL_MIN_BUCKETS);这里问题终于明了了，total 就是 TBL_MIN_BUCKETS，那么 TBL_MIN_BUCKETS 是什么意思呢？只能搜索了，结果会发现就是宏定义：#define TBL_MIN_BUCKETS 1024。到此参数的问题解决了。

第五步

上面是分析方法，总结下可以得出：参数 element_size 为 sizeof(struct hlist_head*)（其实这里暗示了 element_size 就是指哈希头的大小，后面会有用）；参数 total 为 1024，是从最开始宏定义过来的，这个参数倒推不出什么含义，只是个数值。

上面 element_size 和 total 的含义是有前面的源代码推到而来，而源代码自带的注释含义为，element_size：数组中单个元素的大小；total：应该被创建的元素总个数。

[cpp] view plain copy

```
1. //  参数 element_size：数组中单个元素大小，其实暗示了该数组存放了 struct  hlist_head
2. //  参数 total：没有别的意思，就是每次创建是最小的元素总是，默认设置为 1024
3. struct  flex_array  *flex_array_alloc(int  element_size,  unsigned  int  total,
4.                                       gfp_t  flags)
5. {
6.         struct  flex_array  *ret;
7.         int  elems_per_part  =  0;
8.         int  reciprocal_elems  =  0;
9.         int  max_size  =  0;
10. //  这里不懂为什么还要对 element_size 判空，因为源代码中已经写死了：sizeof(struct  hlist_head)
11. //  这里很明显恒成立。个人猜测是为了说明如果数组元素为空，则不能执行下面的操作
12.         if  (element_size)  {
13.
14. //  下面的代码让我很疑惑了，elems_per_part 按照变量名的意思：每个 part 数组元素中存储了多少个元素
15. //  根据赋值情况分析，后面的宏定义为：#define  FLEX_ARRAY_ELEMENTS_PER_PART(size)  (FLEX_ARRAY_PART_SIZE  /  size)
16. //  得:(页大小/  element_size)==  表示一个页(1024)中能存放多少哈希头
17.                 elems_per_part  =  FLEX_ARRAY_ELEMENTS_PER_PART(element_size);
18.                 reciprocal_elems  =  reciprocal_value(elems_per_part);
19.
20. //  max_size 也让我疑惑，按照变量名来说应该是，数组中存放了最大的元素数
21. //  根据赋值情况来分析，先分析等号右边的宏定义：
22. //  #define  FLEX_ARRAY_NR_BASE_PTRS      (FLEX_ARRAY_BASE_BYTES_LEFT  /  sizeof(struct  flex_array_part  *))
23.
```

```
24. //  offsetof()宏用来求一个成员在结构体  中的偏移量
25. //  #define  FLEX_ARRAY_BASE_BYTES_LEFT      (FLEX_ARRAY_BASE_SIZE  -
    offsetof(struct  flex_array,  parts))
26.
27. //  所以前面的  FLEX_ARRAY_NR_BASE_PTRS  为桶结构中弹性数组的大小，即：可
    以存放多少个 flex_array_part  *
28. //  后面的 elems_per_part 为每个数组元素中能存放多少哈希头，所以 max_size
    就是该结构中最多能存放的哈希头数
29.                 max_size  =  FLEX_ARRAY_NR_BASE_PTRS  *  elems_per_part;

30.         }
31.
32.         /*  max_size  will  end  up  0  if  element_size  >  PAGE_SIZE
    */
33. //  上面自带的注释意思为如果数组元素为一个页的大小，那么 max_size 将会趋近
    于 0
34.         if  (total  >  max_size)
35.                 return  NULL;
36.         ret  =  kzalloc(sizeof(struct  flex_array),  flags);//  为数组分
    配内存空间
37.         if  (!ret)
38.                 return  NULL;
39. //  下面一系列的代码就是为结构体成员变量赋值
40.         ret->element_size  =  element_size;
41.         ret->total_nr_elements  =  total;
42.         ret->elems_per_part  =  elems_per_part;
43.         ret->reciprocal_elems  =  reciprocal_elems;
44.         if  (elements_fit_in_base(ret)  &&  !(flags  &  __GFP_ZERO))
45.                 memset(&ret->parts[0],  FLEX_ARRAY_FREE,
46.                                                 FLEX_ARRAY_BASE_BYTES_LE
    FT);
47.         return  ret;
48. }
```

总结：

其实从上面的源代码已经可以初步的看出 flex_array 结构体中的各个成员变量的大概含义了，现来初步总结下各个成员的含义：

[cpp] view plain copy

```
1.  struct  {
2.                          int  element_size;  //  这是 flex_array_part 结构
    体存放的哈希头指针的大小
3.                          int  total_nr_elements;  //  这是所有
    flex_array_part 结构体中的哈希头指针的总个数
```

```
4.                          int  elems_per_part;  //  这是每个 part 指针指向的空间能存储多少个哈希头指针
5.                          u32  reciprocal_elems;
6.                          struct  flex_array_part  *parts[];  //  结构体指针数组，里面存放的是 struct  flex_array_part 结构的指针
7.                  };
```

矛盾：

按照上面的分析结论，发现在 static inline int elements_fit_in_base(struct flex_array *fa)函数中是不怎么好理解的。至少我开始是理解不了，下面请看下该函数的具体实现：

[cpp] view plain copy

```
1.  static  inline  int  elements_fit_in_base(struct  flex_array  *fa)
2.  {
3.  //  fa->element_size  根据上面的结论应该是哈希头的大小，flex_array_part 结构体中存放的哈希头大小
4.  //  fa->total_nr_elements  根据上面的结论应该是所有哈希头的总数
5.  //  那么 data_size  就是所有存储哈希头的空间大小了，矛盾来了
6.          int  data_size  =  fa->element_size  *  fa->total_nr_elements;

7.  //  FLEX_ARRAY_BASE_BYTES_LEFT 是什么意思呢？
8.  //  #define  FLEX_ARRAY_BASE_BYTES_LEFT    (FLEX_ARRAY_BASE_SIZE  -  offsetof(struct  flex_array,  parts))
9.  //  offsetof()宏用来求一个成员在结构体中的偏移量
10. //  所以所有存储哈希头空间的大小和  FLEX_ARRAY_BASE_BYTES_LEFT  比较是什么意思呢？我当时的判断就是element_size 和 total_nr_elements 这两个成员变量理解错了。
11.         if  (data_size  <=  FLEX_ARRAY_BASE_BYTES_LEFT)
12.                 return  1;
13.         return  0;
14. }
```

扩展：

为了解决上面这个问题，我对 flex_array.h 和 flex_array.c 文件进行了反复的分析，做各种假设最后终于解决了。但是还存在一个问题就是 flex_array_part 结构体：

[cpp] view plain copy

```
1.  struct  flex_array_part  {
2.          char  elements[FLEX_ARRAY_PART_SIZE];
3.  };
```

仔细的学者会发现这个 flex_array_part 结构体中就是一个页长大小的字符数组，注意是 char 型的字符数组，那怎么存放哈希头指针呢？这是个问题，我也正在研究，当然如果对于只是用来工作的，那么可以不必计较的这么仔细。

openVswitch（OVS）源代码分析之工作流程（哈希桶结构体的解释）

这篇 blog 是专门解决前篇 openVswitch（OVS）源代码分析之工作流程（哈希桶结构体的疑惑）中提到的哈希桶结构 flex_array 结构体成员变量含义的问题。

引用下前篇 blog 中分析讨论得到的 flex_array 结构体成员变量的含义结论：

[cpp] view plain copy

```
1. struct {
2.                         int element_size; // 这是 flex_array_part 结构体存放的哈希头指针的大小
3.                         int total_nr_elements; // 这是所有 flex_array_part 结构体中的哈希头指针的总个数
4.                         int elems_per_part; // 这是每个 part 指针指向的空间能存储多少个哈希头指针
5.                         u32 reciprocal_elems;
6.                         struct flex_array_part *parts[]; // 结构体指针数组，里面存放的是 struct flex_array_part 结构的指针
7.                 };
```

其实这个结论是正确的，这些结构体成员的含义就是这些意思。但前篇分析中这个结论和 static inline int elements_fit_in_base(struct flex_array *fa)函数产生矛盾，这里也看下该函数的具体实现：

[cpp] view plain copy

```
1. static inline int elements_fit_in_base(struct flex_array *fa)
2. {
3. // fa->element_size 根据上面的结论应该是哈希头的大小，flex_array_part 结构体中存放的哈希头大小
4. // fa->total_nr_elements 根据上面的结论应该是所有哈希头的总数
5. // 那么 data_size 就是所有存储哈希头的空间大小了，矛盾来了
6.         int data_size = fa->element_size * fa->total_nr_elements;
7. // FLEX_ARRAY_BASE_BYTES_LEFT 是什么意思呢？
8. // #define FLEX_ARRAY_BASE_BYTES_LEFT (FLEX_ARRAY_BASE_SIZE - offsetof(struct flex_array, parts))
9. // offsetof()宏用来求一个成员在结构体中的偏移量
10. // 所以所有存储哈希头空间的大小和 FLEX_ARRAY_BASE_BYTES_LEFT 比较是什么意思呢？
```

```
11. //  我当时的判断就是element_size和total_nr_elements这两个成员变量理解错了。
12.         if  (data_size  <=  FLEX_ARRAY_BASE_BYTES_LEFT)
13.                 return  1;
14.         return  0;
15. }
```

如果按照一般的思想来分析这个源代码真的有问题了，至少这个函数分析不下了。那么真正的原因是什么呢？

首先来看下哈希桶内存申请函数（在上篇中有分析）其中传过来的分别为：elements = sizeof(struct hlist*)和 total = 1024（宏定义而来）；

再看看上面这个函数的实现：data_size = element_size * total_nr_elements； 也即是 data_size = elements * total；带入数据得：data_size = 4 * 1024 = 4096（因为两个参数一个是宏定义的，对整个项目来说是不变的；另外一个也一样是不会变的。所以可以当做常量带入去应验下）；

那么现在来看看 if 判断语句：data_size <= (4096 - 4*4)；因为根据上面的 flex_array 结构体成员变量可以知道：有 3 个 int 型成员和一个 u32 类型的成员。所以得到 parts 前有 4*4 个字节。用一个页的大小减去到 parts 成员前的字节为：4096 - 4*4；

最后把所有数据带入可以得到：4096 <= (4096 - 4*4)；那么这个条件肯定是恒不成立的。所以这个函数就是多余的了，因为 data_size 的值是一定为 4096 的，不管 flex_array 结构中成员变量代表什么意思。而 FLEX_ARRAY_BASE_BYTES_LEFT 也是一定不变的。

得到上面的结论其实离真相就比较接近了，可以想象得到一个由这么多顶尖的程序员设计出来的项目，不太可能会出现一个冗余的函数，而且在 flex_array.c 中大量的使用。那么这个函数一定有其他用处，我想了很多种可能，也反复的分析 flex_array.c 和 flex_array.h 中的源代码，最后我得到一种猜想：就是当这个项目中所要的最大元素数非常小，就是说根据需求 total 不需要 1024，不要那么大呢？

猜想：需要的流表项链表头结点比较少（total_nr_elements < 1024），那么不需要分配一个 parts 指针（一个 parts 数组指针元素有一个页大小的空间）来存储，如果 total_nr_elements 不大于 1020，就没必要分配 parts 指针了，直接在 flex_array 结构体（该结构体的大小为一个页，有 3 个 int 型和 1 个 u32 成员，所以剩下的就是 1020 * 4 个字节了）中存储就得了。

下面来验证下这个猜想，来分析调用了 static inline int elements_fit_in_base(struct flex_array *fa)函数的各个代码：

[cpp] view plain copy

```
1. if (elements_fit_in_base(fa))
2.                 part = (struct flex_array_part *)&fa->parts[0];
3. else {
4.                 part_nr = fa_element_to_part_nr(fa, element_nr);
5.                 part = __fa_get_part(fa, part_nr, flags);
6.                 if (!part)
7.                         return -ENOMEM;
8.         }
```

这段代码在很多函数中都有，可以看 int flex_array_put(struct flex_array *fa, unsigned int element_nr, void *src,gfp_t flags)；数据拷贝函数的具体实现。该代码中调用了 elements_fit_in_base(fa)来判断，如果成立，也就是说 total_nr_elements 不大于 1020；那么直接用数组头元素的地址来强转为需要的结构体，即是直接在数组头元素存储的地方开始操作，而不是数组头元素指向的地方开始操作。说明了数据就是存储在 flex_array 结构体中。

下面来看另外一段代码：

[cpp] view plain copy

```
1. void flex_array_free_parts(struct flex_array *fa)
2. {
3.         int part_nr;
4.
5.         if (elements_fit_in_base(fa))
6.                 return;
7.         for (part_nr = 0; part_nr < FLEX_ARRAY_NR_BASE_PTRS; part
   _nr++)
8.                 kfree(fa->parts[part_nr]);
9. }
```

看这段代码大概就知道是用来释放 parts 内存的。该代码中调用了 elements_fit_in_base(fa)，如果成立，也就是说 total_nr_elements 不大于 1020；那么就直接返回，什么都不执行。这就暗示了这个项目中根本就没有申请 parts 内存，所有的流表项链表头结点都是存放在 flex_array 结构体中的。再看下行的 for 循环，是从 0 开始的，更能说明如果 total_nr_elements 大于 1020 就一定得申请 parts 内存。

还有其他代码中调用了该函数，就不一一列证了。就目前为止来说这个猜想还是比较符合源代码的，我不能百分百的说这个猜想是正确的，希望有兴趣的朋友可以分析下。当然我也在找各种途径去分析这个矛盾和猜想。

openVswitch（OVS）源代码分析 upcall 调用（之 linux 中的 NetLink 通信机制）

前面做了一大堆的准备就是为了分析下 upcall 调用，但是现在因为工作重心已经从 OpenVswitch 上转移到了 openstack，所以根本没时间去研究 OpenVswitch 了。（openstack 是用 Python 写的，我大学没接触过 Python，所以现在要一边学 Python 一边学 openstack）后面的 OpenVswitch 分析更新的时间可能会有点久。

由于前面做了很多准备，所以这里不能只分析 NetLink 通信机制（否则可能会感觉没意思了），首先来分析下 upcall 函数调用的原因。如果看了前面的源码分析的就会知道，在什么情况下会调用 upcall 函数呢？就是在一个数据包查找不到相应的流表项时，才会调用 upcall 函数（比如一个数据包第一次进入这个内核，里面没有为这个数据包设定相应的流表规则）。upcall 函数的调用其实就是把数据包的信息下发到用户 空间去，而由内核空间到用户空间的通信则要用到 linux 中的 NetLink 机制。所以熟悉下 NetLink 通信可以知道 upcall 函数调用需要什么样的参数以及整个函数的作用和功能。

现在来测试下 NetLink 的使用，NetLink 由两部分程序构成，一部分是用户空间的，另外一部分是内核空间的。用户空间的和大多数 socket 编程一样，只是用的协议时 AF_NETLINK，其他基本都市一样的步骤。

下面是 NetLine 程序中的用户代码 NetLinke_user.c：

[cpp] view plain copy

```cpp
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include  <sys/types.h>
#include<unistd.h>
#include<sys/stat.h>
#include<sys/socket.h>
#include<linux/netlink.h>

#define  NETLINK_TEST  30
#define  MAX_MSG  1024

int  main(void)
{
```

```
/*按代码规范所有变量都要定义在函数的开始部分，但为了便于理解，所以顺序定义变量*/
                /*
                  *        struct  sockaddr_nl  addr;
                  *        struct  nlmsghdr    *nlhdr;
                  *        struct  iovec                iov;
                  *        struct  msghdr              msg;
                  *        int                           sockId;
                  */

                //创建 socket 套接字
                int  socketId  =  socket(AF_NETLINK,SOCK_RAW,NETLINK_TEST);
                if  (0  >  socketId){
                                printf("The  error  in  socket_create!\n");
                                return  -1;
                }

                //套接字地址设置
                struct  sockaddr_nl  addr;
                memset(&addr,0,sizeof(struct  sockaddr_nl));
                addr.nl_family  =  AF_NETLINK;  //一定是这个协议
                addr.nl_pid  =  0;  //消息是发给内核的，所以为 0;或者内核多播数据给用户空间
                addr.nl_groups  =  0;  //  单播或者发给内核

                //将打开的套接字和 addr 绑定
                int  ret  =  bind(socketId,(struct  sockaddr*)(&addr),sizeof(addr));
                if  (0  >  ret){
                                printf("The  error  in  bind!\n");
                                close(socketId);
                                return  -1;
                }

                //NetLink 消息头设置
                struct  nlmsghdr  *nlhdr  =  NULL;
                nlhdr  =  (struct  nlmsghdr*)malloc(NLMSG_SPACE(MAX_MSG));
                if  (!nlhdr){
                                printf("The  error  in  nlmsghdr_malloc!\n");
                                close(socketId);
```

```
52.                                 return  -1;
53.             }
54.                     nlhdr->nlmsg_len  =  NLMSG_SPACE(MAX_MSG);
55.                     nlhdr->nlmsg_pid  =  getpid();//内核如果要返回消息会查找
    这个 pid
56.                     nlhdr->nlmsg_flags  =  0;
57.                     strcpy(NLMSG_DATA(nlhdr),"This  is  data  what  will  be
     sent!\n");
58.
59.                     //设置消息缓存指针
60.                     struct  iovec  iov;
61.                     memset(&iov,0,sizeof(struct  iovec));
62.                     iov.iov_base  =  (void*)nlhdr;
63.                     iov.iov_len    =  NLMSG_SPACE(MAX_MSG);
64.
65.                     //设置消息结构体
66.                     struct  msghdr  msg;
67.                     memset(&msg,0,sizeof(struct  msghdr));
68.                     msg.msg_iov  =  &iov;
69.                     msg.msg_iovlen  =  1;
70.
71.                     //发送消息给内核
72.                     ret  =  sendmsg(socketId,&msg,0);
73.                     if  (0  >  ret){
74.                                     printf("The  error  in  sendmsg!\n");

75.                                     close(socketId);
76.                                     free(nlhdr);
77.                                     return  -1;
78.                     }
79.
80.                     /********************接受消息部分
    **********************/
81.                     printf("begin  receive  message!\n");
82.
83.                     //对接受消息的字段进行清零，因为发送时，里面存储了发送数
    据
84.                     memset((char*)NLMSG_DATA(nlhdr),0,MAX_MSG);
85.                     recvmsg(socketId,&msg,0);
86.                     //打印接受到的消息
87.                     printf("receive  message:==========\n%s\n",NLMSG_DATA(n
    lhdr));
88.                     //收尾工作，对资源的处理
89.                     close(socketId);
```

```
90.                 free(nlhdr);
91.                 return  0;
92.
93. }
```

NetLink 程序内核代码本来有两种情况的：一、单播给某个指定的 pid；二、多播个 nl_gloups.下面的代码只有单播的功能，没有多播。多播实验了很久也没成功，所以就搁着了。

下面是 NetLink 程序中的 NetLink_kernel.c 内核代码：

[cpp] view plain copy

```
1.  /*=====================KERNEL=========================================
    =*/
2.  #include<linux/init.h>
3.  #include<linux/module.h>
4.  #include<linux/kernel.h>
5.  #include<linux/types.h>
6.  #include<net/sock.h>
7.  #include  <linux/skbuff.h>
8.  #include  <linux/ip.h>
9.  #include  <linux/sched.h>
10. #include<linux/netlink.h>
11.
12. #define  NETLINK_TEST  30
13. #define  MAX_MSG            1024
14.
15. //  内核 sock
16. struct  sock*  socketId  =  NULL;
17.
18. //  单播数据
19. char  kernel_to_user_msg_unicast[]  =  "hello  userspace  uncast!";
20. int    unicastMsgLen  =  NLMSG_SPACE(sizeof(kernel_to_user_msg_unicast))
    ;
21.
22. //  单播，groups 一定要为 0,pid 则为单播 pid
23. int  kernel_unicast_user(u32  pid)
24. {
25.                 struct  sk_buff  *skb_sent  =  NULL;
26.                 struct  nlmsghdr  *nlhdr      =  NULL;
27.
28.                 //  创建网络数据包结构体
29.                 skb_sent  =  alloc_skb(unicastMsgLen,GFP_KERNEL);
30.                 if  (!skb_sent){
```

```
31.                                 printk(KERN_ALERT"error in uncast all
   oc_skb!\n");
32.                                 return -1;
33.                 }
34.                 // nlhdr = NLMSG_NEW(skb_sent,0,0,NLMSG_DONE,unicastM
   sgLen,0);
35.                 nlhdr = nlmsg_put(skb_sent, 0, 0, 0, unicastMsgLen
   ,0);
36.
37.                 // 填充发送数据
38.                 memcpy(NLMSG_DATA(nlhdr),kernel_to_user_msg_unicast,unicas
   tMsgLen);
39.
40.                 // 设置控制块
41.                 NETLINK_CB(skb_sent).pid = 0;
42.                 NETLINK_CB(skb_sent).dst_group = 0;
43.
44.                 // 单播发送
45.                 if (0 > netlink_unicast(socketId,skb_sent,pid,0)){

46.                                 printk(KERN_ALERT"error in netlink_uni
   cast\n");
47.                                 return -1;
48.                 }
49.                 return 0;
50. }
51. EXPORT_SYMBOL(kernel_unicast_user);// 导出函数符号
52.
53. // 注册的回调函数，处理接受到数据
54. static void kernel_recv_user(struct sk_buff *__skb)
55. {
56.                 struct sk_buff *skb = NULL;
57.                 struct nlmsghdr *nlhdr = NULL;
58.
59.                 skb = skb_get(__skb);// 从一个缓冲区中引用指针出
   来
60.
61.                 if (skb->len >= NLMSG_SPACE(0)){    //其实就是判断是
   否为空
62.                                 nlhdr = nlmsg_hdr(skb);// 宏
   nlmsg_hdr(skb)的实现为(struct nlmsghdr*)skb->data
63.
64.                                 // 开始打印接受到的消息
```

```
65.                                 printk(KERN_INFO"base  info  =======\n
   len:%d,  type:%d,  flags:%d,  pid:%d\n",
66.                                                                 nlhdr->n
   lmsg_len,nlhdr->nlmsg_type,nlhdr->nlmsg_flags,nlhdr->nlmsg_pid);
67.                                 printk(KERN_INFO"data  info  =======\n
   data:%s\n",(char*)NLMSG_DATA(nlhdr));
68.                 }
69.                                 kernel_unicast_user(nlhdr->nlmsg_pid);

70. }
71.
72. //  模块插入时触发的函数，一般用来做初始化用，也是模块程序的入口
73. static  int  __init  netlink_init(void)
74. {
75.                 printk(KERN_ALERT"netlink_init()!\n");
76.
77.                 socketId  =  netlink_kernel_create(&init_net,NETLINK_TES
   T,0,kernel_recv_user,NULL,THIS_MODULE);
78.                 if  (!socketId){
79.                                 printk(KERN_ALERT"error  in  sock  creat
   e!\n");
80.                                 return  -1;
81.                 }
82.                 return  0;
83. }
84.
85. //  模块退出时触发的函数，目的一般是用来清理和收尾工作
86. static  void  __exit  netlink_exit(void)
87. {
88.                 printk(KERN_ALERT"netlink_exit()!\n");
89.                 netlink_kernel_release(socketId);
90. }
91.
92. MODULE_LICENSE("Dual  BSD/GPL");
93. MODULE_AUTHOR("yuzhihui");
94.
95. module_init(netlink_init);
96. module_exit(netlink_exit);
```

上面的两个程序就是 linux 中内核空间和用户空间通信的实例，其实这个 NetLink 通信实验并不是非常重要，对理解 OpenVswitch 来说，个人只是兴趣所好，特意去看了下 NetLink 的工作原理。至于多播的功能实现，如果有知道麻烦给个链接。

openVswitch（OVS）源代码之 linux RCU 锁机制分析

前言

本来想继续顺着数据包的处理流程分析 upcall 调用的，但是发现在分析 upcall 调用时必须先了解 linux 中内核和用户空间通信接口 Netlink 机制，所以就一直耽搁了对 upcall 的分析。如果对 openVswitch 有些了解的话，你会发现其实 openVswitch 是在 linux 系统上运行的，因为 openVswitch 中有很多的机制，模块等都是直接调用 linux 内核的。比如：现在要分析的 RCU 锁机制、upcall 调用、以及一些结构体的定义都是直接从 linux 内核中获取的。所以如果你在查看源代码的一些结构（或者模块，机制性代码）时，发现在 openVswitch 中没有定义（我用的是 Source Insight 来查看和分析源码，可以很好的查看是否定义过），那么很可能就是 openVswitch 包含了 linux 头文件引用了 linux 内核的一些定义。

RCU 是 linux 的新型锁机制（RCU 是在 linux 2.6 内核版本中开始正式使用的），本来一直纠结要不要用篇 blog 来说下这个锁机制。因为在 openVswitch 中有很多的地方用到了 RCU 锁，我开始分析的时候都是用一种锁机制一笔带过（可以看下 openVswitch（OVS）源代码分析之数据结构里面有很多地方都用到了 RCU 锁机制）。后来发现有很多地方还用到了该锁机制的链表插入和删除操作，而且后面分析的代码中也有 RCU 的出现，所以就稍微的说下这个锁机制的一些特性和操作。

RCU 运行原理

我们先来回忆下读写锁（rwlock）运行机制，这样可以分析 RCU 的时候可以对照着分析。读写锁分为读锁（也称共享锁），写锁（也称排他锁，或者独占锁）。分情况来分析下读写锁：

第一、要操作的数据区被上了读锁；1、若请求是读数据时，上读锁，多个读锁不排斥（即，在访问数据的读者上线未达到时，可以对该数据区再上读锁）；2、若请求是写数据，则不能马上上写锁，而是要等到数据区的所有锁（包括读锁和写锁）都释放掉后才能开始上写访问。

第二、要操作的数据区上了写锁；则不管是什么请求都必须等待数据区的写锁释放掉后才能上锁访问。

同理来分析下 RCU 锁机制： RCU 是 read copy udate 的缩写，按照单词意思就知道这是一种针对数据的读、复制、修改的保护锁机制。锁机制原理：

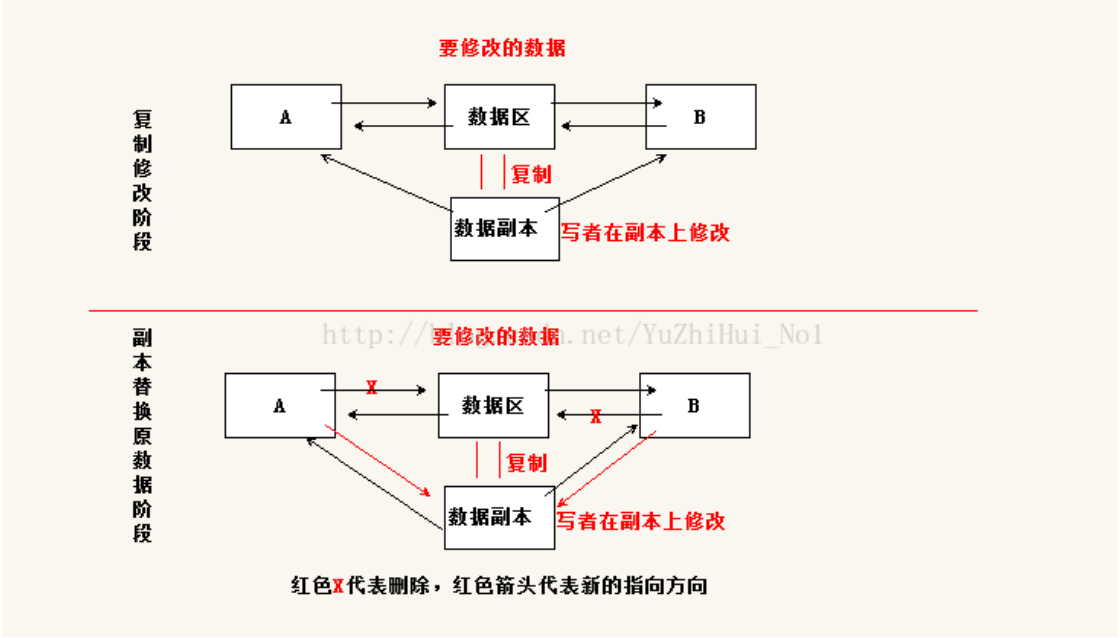
第一、写数据的时候，不需要像读写锁那样等待所有锁的释放。而是会拷贝一份数据区的副本，然后在副本中修改，等待修改完后。用这个副本替换原来的数据区，替换的时候就要像读写锁中上写锁那样，等到原数据区上所有访问者都退出后，才进行数据的替换；根据这种特性可以推断出，用 RCU 锁可以有多个写者，拷贝了多份数据区数据，修改后各个写着陆续的替换掉原数据区内容。

第二、读数据的时候，不需要上任何锁，也几乎不需要什么等待（读写锁中如果数据区有写锁则要等待）就可以直接访问数据。为什么说几乎不需要等待呢？因为写数据中替换原数据时，只要修改个指针就可以，消耗的时间可以说几乎不算，所以说读数据不需要其他额外开销。

总结下 RCU 锁机制特性，允许多个读者和多个写者同时访问共享数据区的内容。而且这种锁对多读少写的数据来说是非常高效的，可以让 CPU 减少些额外的开销。如果写得操作多了的话，这种机制就没读写锁那么好了。因为 RCU 写数据开销还是很大的，要拷贝数据，然后还要修改,最后还要等待替换。其实这个机制就好比我们在一台共享服务器上放了个文件，有很多个人一起使用。如果你只是看看这个文件内容,那么直接在服务器上 cat 查看就可以。但如果你要修改该文件，那么你不能直接在服务器上修改，因为你这样操作会影响到将要看这个文件或者写这个文件的人。所以你只能先拷贝到自己本机上修改，当最后确认保证正确时，然后就替换掉服务器上的原数据。

RCU 写者工作图示

下面看下 RUC 机制下修改数据（以链表为例）



根据上面的图会发现其实替换的时候只要修改下指针就可以，原数据区内容在被替换后，默认会被垃圾回收机制回收掉。

linux 内核 RCU 机制 API

了解了 RCU 的这些机制原理，下面来看下 linux 内核中常使用的一些和 RCU 锁有关的操作。注意，本 blog 并不会过多的去深究 RCU 最底层的实现机制，因为分享 RCU 工作机制的目的只是为了更好的了解 openVswitch 中使用到的那部分代码的理解，而不是为了分析 linux 内核源代码，不要本末倒置。如果遇到个知识点就拼命的深挖，那么你看一份源代码估计得几个月。

rcu_read_lock()；看到这里有人可能会觉得和上面有矛盾，不是说好的读者不需要锁吗？其实这不是和上读写锁的那种上锁，这仅仅只是标识了临界区的开始位置。表明在临界区内不能阻塞和休眠，也不能让写者进行数据的替换（其实这功能远不止这些）。rcu _read_unlock()则是和上面 rcu_read_lock()对应的，用来界定一个临界区（就是要用锁保护起来的数据区）。

synchronize_rcu()；当该函数被一个 CPU 调用时（一般是有写者替换数据时调用），而其他的 CPU 都在 RCU 保护的临界区读数据，那么 synchronize_rcu()将会保证阻塞写者，直到所有其它读数据的 CPU 都退出临界区时，才中止阻塞，让写着开始替换数据。该函数作用就是保证在替换数据前，所有读数据的 CPU 能够安全的退出临界区。同样，还有个 call_rcu()函数功能也是类似的。如果 call_rcu（）被一个 CPU 调用，而其他的 CPU 都在

RCU 保护的临界区内读数据，相应的 RCU 回调的调用将被推迟到其他读临界区数据的 CPU 全部安全退出后才执行（可以看 linux 内核源文件的注释，在 Rcupdate.h 文件中 rcu_read_look()函数前面的注释）。

rcu_dereference()； 获取在一个 RCU 保护的指针，指向 RCU 读端临界区。他的指针以后可能会被安全地解除引用。说到底就是一个 RCU 保护指针。

list_add_rcu()；往 RCU 保护的数据结构中添加一个数据节点进去。这个和一般的往链表中增加一个节点操作是类似的，唯一不同的是多了这条代码：
rcu_assign_pointer(prev->next, new)； 代码大概含义：分配指向一个新初始化的结构指针，将由 RCU 读端临界区被解除引用，返回指定的值。（说实话我也不太懂这个注释是什么意思）大概的解释下：就是让插入点的前一个节点的 next 指向新增加的 new 节点，为什么要单独用一条这个语句来实现，而不是用 prev->next = new；直接实现呢？这是因为 prev->next 本来是指向其他值得，有可能有 CPU 通过 prev->next 去访问其他 RCU 保护的数据了，所以如果你要插入一个 RCU 保护的数据结构中必要要调用这个语句，它里面会帮你处理好一些细节（比如有其他 CPU 使用后面的数据，直接使用 prev->next 可能会使读数据的 CPU 断开，产生问题），并且让刚加入的新节点也受到 RCU 的保护。这类的插入有很多，比如从头部插入，从尾 部插入等，实现都差不多，这里不一一细讲。

list_for_each_entry_rcu()；这是个遍历 RCU 链表的操作，和一般的链表遍历差不多。不同点就是必须要进入 RCU 保护的 CPU（即：调用了 rcu_read_lock()函数的 CPU）才能调用这个操作，可以和其他 CPU 共同遍历这个 RCU 链表。以此相同的还有其他变相的遍历及哈希链表的遍历，不细讲。

如果在 openVswitch 源代码分析中发现了有关 RCU 的分析和这里的矛盾，可以以这里为准，当然我也会校对下。

OVS datapath 模块分析：packet 处理流程

这来主要看看 ovs 从网络接口收到 packet 后的一系列操作。

在内核模块启动的时候会初始化 vport 子系统（ovs_vport_init），各种 vport 类型，那么什么时候会调用相应的函数与实际网络设备建立联系？其实当我们在为网桥增设端口的时候，就会进入 ovs_netdev_vport_ops 中的 create 方法，进而 注册网络设备。

看 ovs-vsctl add-port br0 eth1 实际做了什么？

```
struct  netdev_vport  {
         struct rcu_head rcu;
         struct net_device *dev;
};
const struct vport_ops  ovs_netdev_vport_ops  = {
         .type                   = OVS_VPORT_TYPE_NETDEV,
         .flags                   = VPORT_F_REQUIRED,
          .init                   = netdev_init,      //之后的内核版本，这里直接
return 0;
         .exit                   = netdev_exit,
          .create                   = netdev_create,
         .destroy         = netdev_destroy,
         .set_addr         = ovs_netdev_set_addr,
         .get_name         = ovs_netdev_get_name,
         .get_addr         = ovs_netdev_get_addr,
         .get_kobj         = ovs_netdev_get_kobj,
         .get_dev_flags          = ovs_netdev_get_dev_flags,
         .is_running         = ovs_netdev_is_running,
         .get_operstate          = ovs_netdev_get_operstate,
         .get_ifindex         = ovs_netdev_get_ifindex,
         .get_mtu         = ovs_netdev_get_mtu,
         .send                   = netdev_send,
```

```
};
--datapath/vport-netdev.c
static struct vport *netdev_create(const struct vport_parms *parms)
{
         struct vport *vport;
         struct netdev_vport *netdev_vport;
         int err;
                   vport           =           ovs_vport_alloc(sizeof(struct
netdev_vport),     &ovs_netdev_vport_ops, parms);
        //有 ovs_netdev_vport_ops 和 vport parameters 来构造初始化一个 vport;
        netdev_vport = netdev_vport_priv(vport);
        //获得 vport 私有区域? ?
          netdev_vport->dev     =     dev_get_by_name(ovs_dp_get_net(vport->dp),
parms->name);
         //通过 interface  name 比如 eth0 得到具体具体的 net_device 结构体,然后下
面注册 rx_handler;
                  if       (netdev_vport->dev->flags       &       IFF_LOOPBACK
||   netdev_vport->dev->type != ARPHRD_ETHER ||
                ovs_is_internal_dev(netdev_vport->dev)) {
                  err = -EINVAL;
                  goto error_put;
         }
         //不是环回接口；而且底层链路层是以太网；netdev->netdev_ops ==
&internal_dev_netdev_ops 显然为 false
          err                  =  netdev_rx_handler_register(netdev_vport->dev,
netdev_frame_hook,  vport);
          //核心，收到 packet 后会调用  netdev_frame_hook 处理;
         dev_set_promiscuity(netdev_vport->dev, 1);   //设置为混杂模式;
         netdev_vport->dev->priv_flags |= IFF_OVS_DATAPATH;   //设置 netdevice
私有区域的标识;
```

```
        return vport;
}
--datapath/vport.h 创建 vport 所需要的参数结构
struct  vport_parms  {
        const char *name;
        enum ovs_vport_type type;
        struct nlattr *options;     //利于必要的时候从 netlink msg 通过属性
OVS_VPORT_ATTR_OPTIONS 取得

        /* For ovs_vport_alloc(). */
         struct datapath *dp;   // 这个 vport 所从属的 datapath
        u16 port_no;    //端口号
         u32 upcall_portid;  // 如果从这个 vport 收到的包 在 flow table 没有得到
匹配就会从 netlink 端口 upcall_portid 发送到用户空间;
};
```

函数 netdev_rx_handler_register(struct net_device *dev,rx_handler_func_t *rx_handler, void *rx_handler_data) 定义在 linux/netdevice.h 实现在 net/core/dev.c 中，为网络设备 dev 注册一个 receive handler，rx_handler_data 指向的是这个 receive handler 是用的内存区域(这里存的是 vport,里面有 datapath 的相关信息)。这个 handler 以后会被 __netif_receive_skb() 呼叫，实际就是更新 netdevice 中的两个指针域,rcu_assign_pointer(dev->rx_handler_data, rx_handler_data), rcu_assign_pointer(dev->rx_handler, rx_handler) 。

netif_receive_skb(struct sk_buff *skb)从网络中接收数据,它是主要的接收数据处理函数，总是成功，这个 buffer 在拥塞处理或协议层的时候可能被丢弃。这个函数只能从软中断环境（softirq context）中调用，并且中断允许。返回值 NET_RX_SUCCESS 表示没有拥塞，NET_RX_DROP 包丢弃。（实现细节暂时没看）

接下来进入我们的钩子函数 netdev_frame_hook（datapath/vport-netdev.c）这里主要看内核版本>=2.6.39 的实现。

```
        static rx_handler_result_t  netdev_frame_hook(struct sk_buff **pskb)
{
```

```
        struct sk_buff *skb = *pskb;
        struct vport *vport;
        if (unlikely(skb->pkt_type == PACKET_LOOPBACK))
                return RX_HANDLER_PASS;
        vport = ovs_netdev_get_vport(skb->dev);
       //提携出前面存入的那个 vport 结构体，vport-netdev.c line 401;
         netdev_port_receive(vport, skb);
        return RX_HANDLER_CONSUMED;
}
```

函数 netdev_port_receive 首先得到一个 packet 的拷贝，否则会损坏先于我们而来的 packet 使用者 (e.g. tcpdump via AF_PACKET)，我们之后没有这种情况，因为会告知 handle_bridge()我们获得了那个 packet 。

skb_push 是将 skb 的数据区向后移动*_HLEN 长度，为了存入帧头；而 skb_put 是扩展数据区后面为存数据 memcpy 做准备。

```
 static void  netdev_port_receive(struct vport *vport, struct sk_buff *skb)
{
       skb = skb_share_check(skb, GFP_ATOMIC);
        //GFP_ATOMIC 用于在中断处理例程或其它运行于进程上下文之外的地方分配内存，不会休眠（LDD214）。
        skb_push(skb, ETH_HLEN);
       //疑问：刚接收到的 packet 应该是有 ether header 的，为何还执行这个操作？？
        if (unlikely(compute_ip_summed(skb, false)))
                goto error;
        vlan_copy_skb_tci(skb); // <2.6.27 版本的时候才需要 VLAN field;
         ovs_vport_receive(vport, skb);
        return;
        .................
}
```

接下来将收到的 packet 传给 datapath 处理（datapath/vport.c）,参数 vport 是收到这个包的 vport（表征物理接口和 datapath），skb 是收到的数据。读的时候要用

rcu_read_lock，这个包不能被共享而且 skb->data 应该指向以太网头域，而且调用者要确保已经执行过 compute_ip_summed() 初始化那些校验和域。

```
void  ovs_vport_receive(struct vport *vport, struct sk_buff *skb)
{
        struct vport_percpu_stats *stats;
        stats = per_cpu_ptr(vport->percpu_stats, smp_processor_id());
        //每当收发数据的时候更新这个 vport 的状态（包数，字节数），struct vport_percpu_stats 定义在 vport.h 中。
        u64_stats_update_begin(&stats->sync);
        stats->rx_packets++;
        stats->rx_bytes += skb->len;
        u64_stats_update_end(&stats->sync);
        if (!(vport->ops->flags & VPORT_F_FLOW))
                OVS_CB(skb)->flow = NULL;
       //vport->ops->flags （VPORT_F_*）影响的是这个通用 vport 层如何处理这个 packet;
        if (!(vport->ops->flags & VPORT_F_TUN_ID))
                OVS_CB(skb)->tun_key = NULL;
         ovs_dp_process_received_packet(vport, skb);
}
```

接下来我们的 datapath 模块来处理传上来的 packet（datapath/datapath.c），首先我们要判断如果存在 skb->cb 域中的 OVS data sw_flow 是空的话，就要从 packet 中提携构造；函数 ovs_flow_extract 从以太网帧中构造 sw_flow_key，为接下来的流表查询做准备；流表结构 struct flow_table 定义在 flow.h 中，流表实在 ovs_flow_init 的时候初始化的?? 如果没有 match 成功,就会 upcall 递交给用户空间处理(见 vswitchd 模块分析)，匹配成功的话执行 flow action（接下来就是 openflow 相关）。

```
void  ovs_dp_process_received_packet(struct vport *p, struct sk_buff *skb)
{
        struct datapath *dp = p->dp;
        struct sw_flow *flow;
```

```
        struct dp_stats_percpu *stats;
        u64 *stats_counter;
        int error;
        stats = per_cpu_ptr(dp->stats_percpu, smp_processor_id());
        if (!OVS_CB(skb)->flow) {
                struct sw_flow_key key;
                int key_len;
               /* Extract flow from 'skb' into 'key'. */
                error =  ovs_flow_extract(skb, p->port_no, &key, &key_len);
              /* Look up flow. */
                                                                        flow
=  ovs_flow_tbl_lookup(rcu_dereference(dp->table),   &key, key_len);
                if (unlikely(!flow)) {
                          struct dp_upcall_info upcall;
                          upcall.cmd = OVS_PACKET_CMD_MISS;
                          upcall.key = &key;
                          upcall.userdata = NULL;
                          upcall.portid = p->upcall_portid;
                           ovs_dp_upcall(dp, skb, &upcall);
                          consume_skb(skb);
                          stats_counter = &stats->n_missed;
                          goto out;
                }
                OVS_CB(skb)->flow = flow;
        }
stats_counter = &stats->n_hit;
        ovs_flow_used(OVS_CB(skb)->flow, skb);
         ovs_execute_actions(dp, skb);
out:
        /* Update datapath statistics. */
```

```
        u64_stats_update_begin(&stats->sync);
        (*stats_counter)++;
        u64_stats_update_end(&stats->sync);
}
```