

# 目錄

1. 前言	1.1
-------	-----

## 网络基础

2. 网络基础理论	2.1
-----------	-----

TCP/IP网络模型	2.1.1
ARP	2.1.2
ICMP	2.1.3
路由	2.1.4
交换机	2.1.5
UDP	2.1.6
DHCP/DNS	2.1.7
TCP	2.1.8
VLAN	2.1.9
Overlay	2.1.10

3. Linux网络	2.2
------------	-----

Linux网络配置	2.2.1
虚拟网络设备	2.2.1.1
iptables/netfilter	2.2.2
负载均衡	2.2.3
流量控制	2.2.4
SR-IOV	2.2.5
内核VRF	2.2.6
eBPF	2.2.7
bcc	2.2.7.1
故障排查	2.2.7.2
XDP	2.2.8
XDP架构	2.2.8.1
使用场景	2.2.8.2
常用工具	2.2.9

网络抓包tcpdump	2.2.9.1
scapy	2.2.9.2
内核网络参数	2.2.10
4. Open vSwitch	2.3
OVS介绍	2.3.1
OVS编译	2.3.2
OVS原理	2.3.3
OVN	2.3.4
OVN编译	2.3.4.1
OVN实践	2.3.4.2
OVN高可用	2.3.4.3
OVN Kubernetes插件	2.3.4.4
OVN Docker插件	2.3.4.5
OVN OpenStack	2.3.4.6
5. DPDK	2.4
DPDK简介	2.4.1
DPDK安装	2.4.2
报文转发模型	2.4.3
NUMA	2.4.4
Ring和共享内存	2.4.5
PCIe	2.4.6
网卡性能优化	2.4.7
多队列	2.4.8
硬件offload	2.4.9
虚拟化	2.4.10
OVS DPDK	2.4.11
SPDK	2.4.12
OpenFastPath	2.4.13

## SDN&NFV

6. SDN	3.1
SDN控制器	3.1.1
OpenDaylight	3.1.1.1

---

ONOS	3.1.1.2
Floodlight	3.1.1.3
Ryu	3.1.1.4
NOX/POX	3.1.1.5
南向接口	3.1.2
OpenFlow	3.1.2.1
OF-Config	3.1.2.2
NETCONF	3.1.2.3
P4	3.1.2.4
数据平面	3.1.3
7. NFV	3.2
8. SDWAN	3.3

---

## 容器网络

9. 容器网络	4.1
Host Network	4.1.1
CNI	4.1.2
CNI介绍	4.1.2.1
Flannel	4.1.2.2
Weave	4.1.2.3
Contiv	4.1.2.4
Calico	4.1.2.5
SR-IOV	4.1.2.6
Romana	4.1.2.7
OpenContrail	4.1.2.8
CNI Plugin Chains	4.1.2.9
CNM	4.1.3
CNM介绍	4.1.3.1
Calico	4.1.3.2
Contiv	4.1.3.3
Romana	4.1.3.4
SR-IOV	4.1.3.5
Kubernetes网络	4.1.4

---

# SDN实践

10. Mininet	5.1
11. SDN实践案例	5.2
Goolge网络	5.2.1

---

## 参考文档

12. FAQ	6.1
13. 参考文档	6.2

---

# SDN网络指南

SDN（Software Defined Networking）作为当前最重要的热门技术之一，目前已经普遍得到大家的共识。有关SDN的资料和书籍非常丰富，但入门和学习SDN依然非常困难。本书整理了SDN实践中的一些基本理论和实践案例心得，希望能给大家带来启发，也欢迎大家关注和贡献。

本书内容包括

- 网络基础
- SDN网络
- 容器网络
- Linux网络
- OVS以及DPDK
- SD-WAN
- NFV
- 实践案例

## 在线阅读

可以通过[GitBook](#)或者[Github](#)来在线阅读。

也可以下载[ePub](#)或者[PDF](#)版本。

## 项目源码

项目源码存放于[Github](#)上，见<https://github.com/feiskyer/sdn-handbook>。

## 网络基础理论

本章介绍计算机网络基础理论。

# TCP/IP网络模型

## TCP/IP网络模型

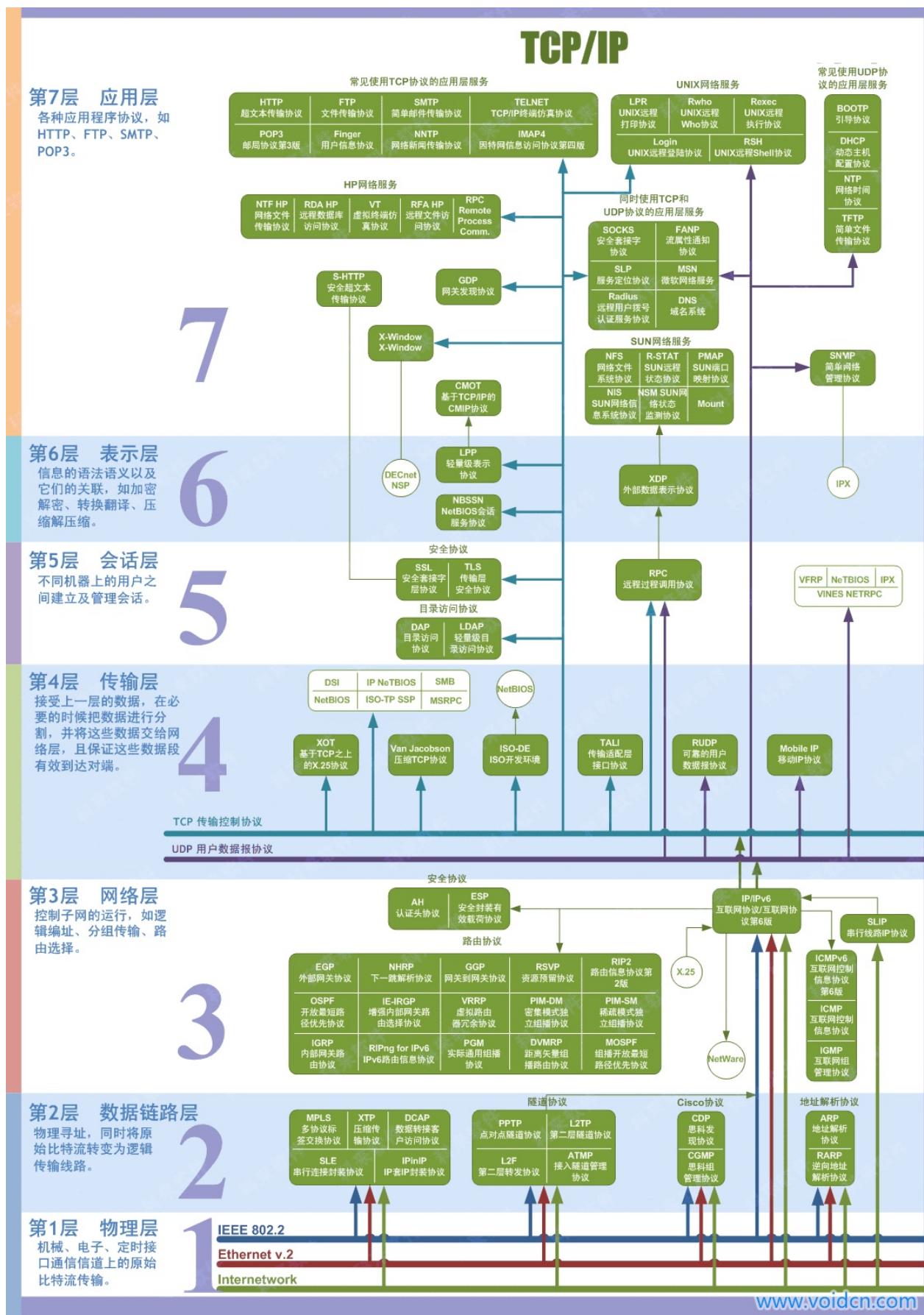
TCP/IP模型是互联网的基础，它是一系列网络协议的总称。这些协议可以划分为四层，分别为链路层、网络层、传输层和应用层。

- 链路层：负责封装和解封装IP报文，发送和接受ARP/RARP报文等。
- 网络层：负责路由以及把分组报文发送给目标网络或主机。
- 传输层：负责对报文进行分组和重组，并以TCP或UDP协议格式封装报文。
- 应用层：负责向用户提供应用程序，比如HTTP、FTP、Telnet、DNS、SMTP等。

在网络体系结构中网络通信的建立必须是在通信双方的对等层进行，不能交错。在整个数据传输过程中，数据在发送端时经过各层时都要附加上相应层的协议头和协议尾（仅数据链路层需要封装协议尾）部分，也就是要对数据进行协议封装，以标识对应层所用的通信协议。

## OSI七层模型

当然在理论上，还有一个OSI七层模型：物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。这是一个理想模型，由于其复杂性并没有被大家广泛采用。



链路层

## 1 以太网和802封装

以太网封装是以RFC894定义的 而802封装则是RFC1042定义的 主机需求RFC要求：（1）必须支持以太网封装 （2）应该支持与RFC894混合的RFC1042封装 （3）或许可以发送RFC1042封装的分组

## 2 SLIP

适用于RS-232和高速调制解调器接入网络 （1）以0xC0结束 （2）对报文中的0xC0和ESC字符进行转义 缺点：没有办法通知本端IP到对端；没有类型字段；没有校验和

## 3 CSLIP

将SLIP报文中的20字节IP首部和20字节TCP首部压缩为3或5字节

## 4 PPP协议

修正了SLIP协议的缺陷，支持多种协议类型；带数据校验和；报文首部压缩；双方可以进行IP地址动态协商（使用IP协议）；链路控制协议可以对多个链路选项进行设置。

## 5 环回接口

用于同一台主机上的程序通过TCP／IP通信。传给环回的数据均作为输入；传给该主机IP地址的数据也是送到环回接口；广播和多播数据先复制一份到环回接口，再送到以太网上。

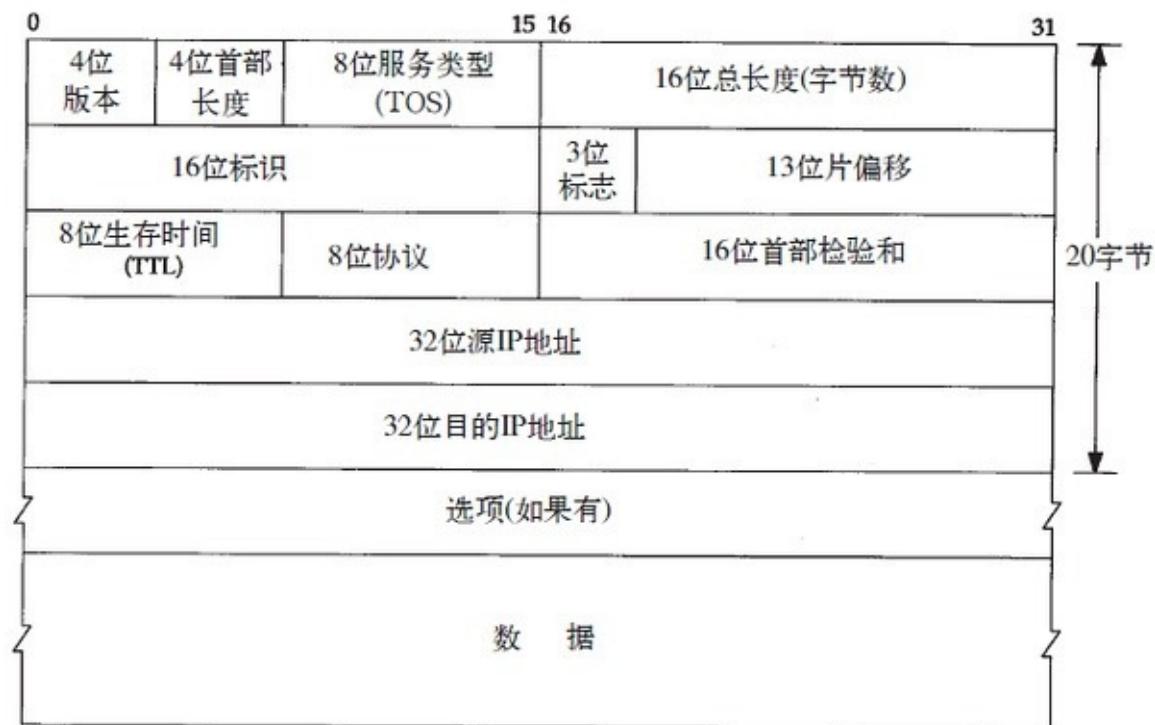
## 6 MTU

对数据帧长度的最大限制，如果数据分组长度大于这个数值，需要在IP层对其进行分片。注意：发往以太网的数据要考虑路径MTU

# IP网际协议

IP是TCP/IP中最为核心的协议，所有的TCP、UDP、ICMP等协议均以IP数据报的格式传输。IP协议提供不可靠、无连接的服务，它不保证数据报一定可以送达目的，也不保证数据报的先后次序。

IP头部格式为



注：网络字节序：32bit传输的次序为0-7bit, 8-15bit, 16-23bit, 24-31bit（即big endian字节序）

## IP路由

IP路由选择是逐跳进行的。IP并不知道到达任何目的的完整路径（当然，除了那些与主机直接相连的）。所有的IP路由选择只为数据报传输提供下一站路由器的IP地址。它假定下一站路由器比发送数据报的主机更接近目的，而且下一站路由器与该主机是直接相连的。

IP路由选择主要完成以下这些功能：

- 1) 搜索路由表，寻找能与目的IP地址完全匹配的表目（网络号和主机号都要匹配）。如果找到，则把报文发送给该表目指定的下一站路由器或直接连接的网络接口（取决于标志字段的值）。
- 2) 搜索路由表，寻找能与目的网络号相匹配的表目。如果找到，则把报文发送给该表目指定的下一站路由器或直接连接的网络接口（取决于标志字段的值）。目的网络上的所有主机都可以通过这个表目来处置。例如，一个以太网上的所有主机都是通过这种表目进行寻径的。这种搜索网络的匹配方法必须考虑可能的子网掩码。关于这一点我们在下一节中进行讨论。
- 3) 搜索路由表，寻找标为“默认”的表目。如果找到，则把报文发送给该表目指定的下一站路由器。

如果上面这些步骤都没有成功，那么该数据报就不能被传送。如果不能传送的数据报来自本机，那么一般会向生成数据报的应用程序返回一个“主机不可达”或“网络不可达”的错误。

IP路由选择是通过逐跳来实现的。数据报在各站的传输过程中目的IP地址始终不变，但是封装和目的链路层地址在每一站都可以改变。大多数的主机和许多路由器对于非本地网络的数据报都使用默认的下一站路由器。

IP路由选择机制的两个特征：（1）完整主机地址匹配在网络号匹配之前执行（2）为网络指定路由，而不必为每个主机指定路由

## IP地址和MAC地址分类

按IP地址范围划分

- A类：地址范围1.0.0.1-126.255.255.255，A类IP地址的子网掩码为255.0.0.0，每个网络支持的最大主机数为 $2^{24}-2=16777214$ 台。
- B类：地址范围128.0.0.1-191.255.255.255，B类IP地址的子网掩码为255.255.0.0，每个网络支持的最大主机数为 $2^{16}-2=65534$ 台
- C类：地址范围192.0.1.1-223.255.255.255，C类IP地址的子网掩码为255.255.255.0，每个网络支持的最大主机数为 $2^{8}-2=254$ 台
- D类：以1110开始的地址，多播地址
- E类：以11110开始的地址，保留地址

按照通讯模式划分

- 单播：目标是特定的主机，比如192.168.0.3
- 广播：目标IP地址的主机部分全为1，并且目的MAC地址为FF-FF-FF-FF-FF-FF。比如B类网络172.16.0.0的默认子网掩码为255.255.0.0，广播地址为172.16.255.255。
- 多播：目标为一组主机，IP地址范围为224.0.0.0～239.255.255.255。多播MAC地址以十六进制值01-00-5E打头，余下的6个十六进制位根据IP多播组地址的最后23位转换得到。

单播是对特定的主机进行数据传送。如给某一个主机发送IP数据包，链路层头部是非常具体的目的地址，对于以太网来说，就是网卡的MAC地址。广播和多播仅应用于UDP，它们对需将报文同时传往多个接收者的应用来说十分重要。

- 广播是针对某一个网络上的所有主机发包，这个网络可能是网络，可能是子网，还可能是所有的子网。如果是网络，例如A类网址的广播就是netid.255.255.255，如果是子网，则是netid.netid.subnetid.255；如果是所有的子网（B类IP）则是netid.netid.255.255。广播所用的MAC地址FF-FF-FF-FF-FF-FF。网络内所有的主机都会收到这个广播数据，网卡只要把MAC地址为FF-FF-FF-FF-FF-FF的数据交给内核就可以了。一般说来ARP，或者路由协议RIP应该是以广播的形式播发的。
- 多播就是给一组特定的主机（多播组）发送数据，这样，数据的播发范围会小一些。多播的MAC地址是最高字节的低位为一，例如01-00-00-00-00-00。多播组的地址是D类IP，规定是224.0.0.0-239.255.255.255。与IP多播相对应的以太网地址范围从01:00:0e:00:00:00到01:00:5e:ff:ff:ff。通过将其低位23 bit映射到相应以太网地址中便可实现多播组地址到以太网地址的转换。由于地址映射是不唯一的，因此要其他的协议实现额外的数据报过滤。

## 子网掩码

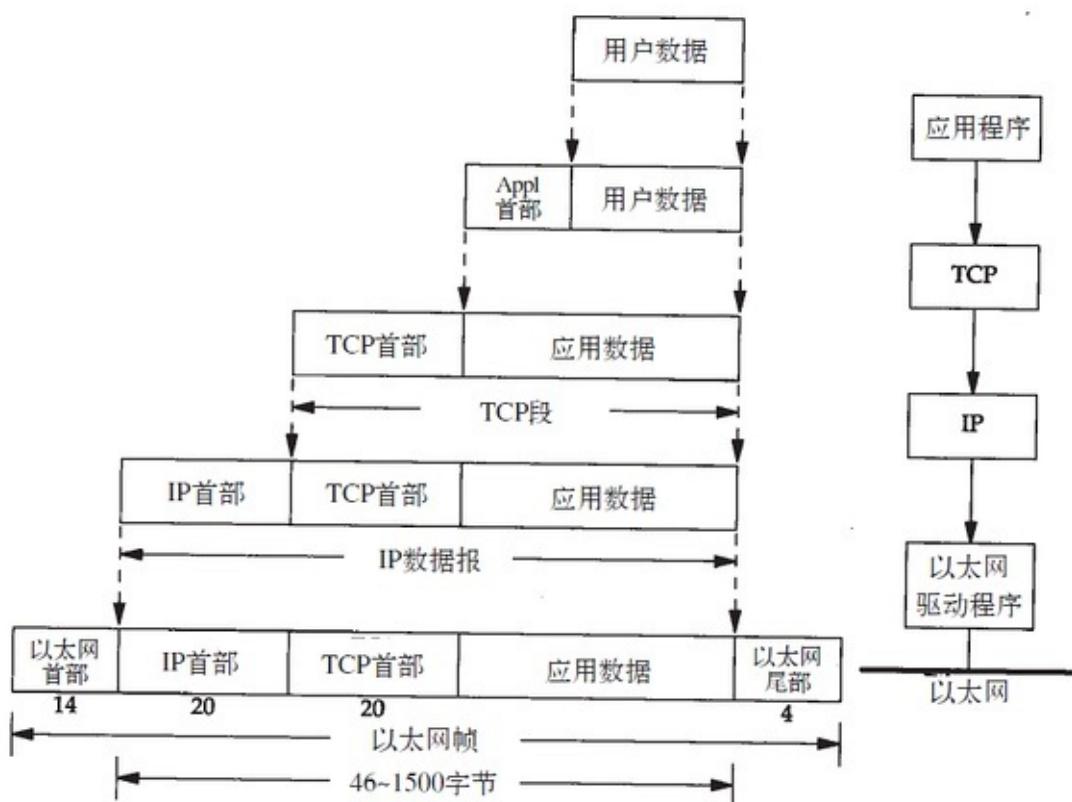
子网掩码用来确定多少bit用于网络号和多少bit用于主机号。

给定IP地址和子网掩码以后，主机就可以确定IP数据报的目的是：(1)本子网上的主机；(2)本网络中其他子网中的主机；(3)其他网络上的主机。

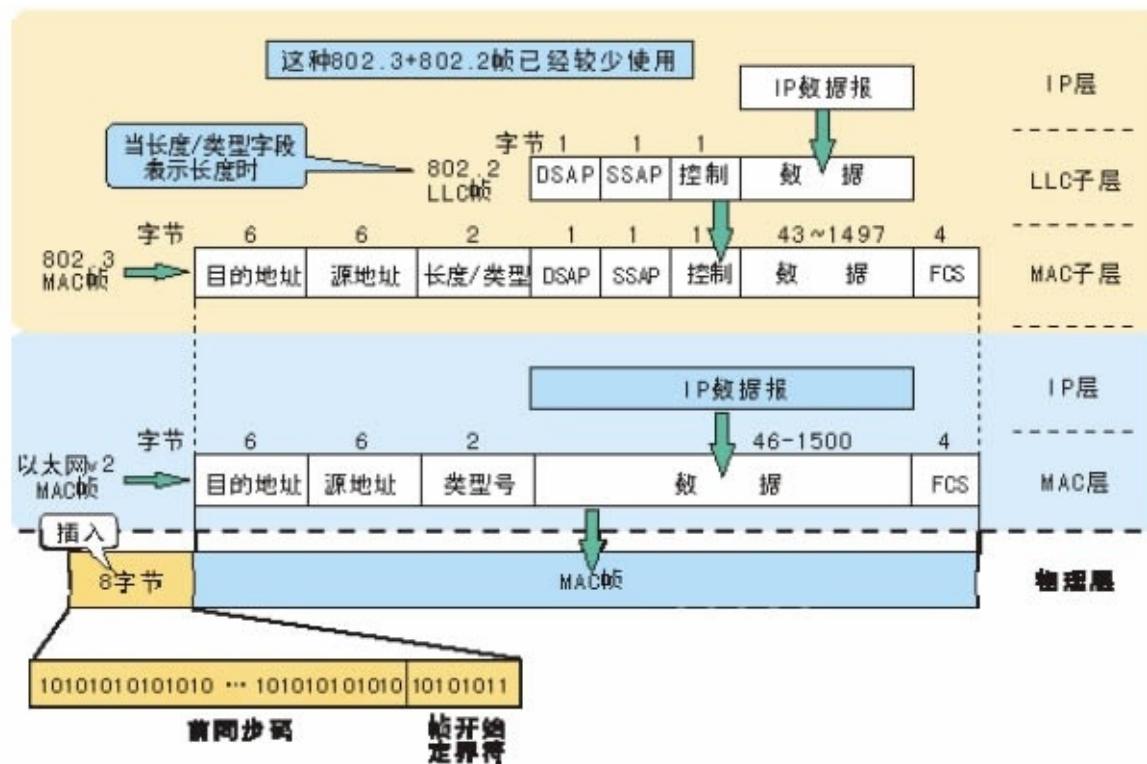
如果知道本机的IP地址，那么就知道它是否为A类、B类或C类地址(从IP地址的高位可以得知)，也就知道网络号和子网号之间的分界线。而根据子网掩码就可知道子网号与主机号之间的分界线。

## 封装

以太网数据帧的物理特性是其长度必须在46~1500字节之间，而数据帧在进入每一层协议栈的时候均会做一些封装。

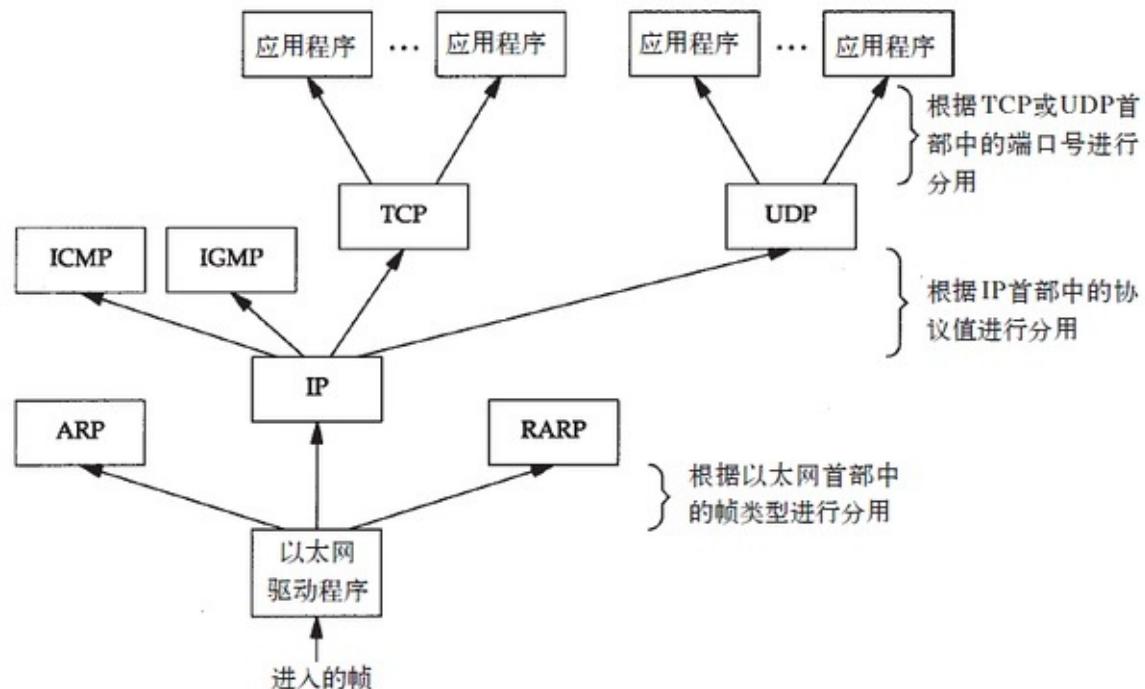


而更具体的以太网帧格式为



## 分用

当目的主机收到一个以太网帧时，就在协议栈中从底向上升，同时去掉各层协议加上的报文首部。每层协议盒都要去检查报文首部的协议标识，以确定接收数据的上层协议。这个过程称作分用。



## 分段 (fragmentation)

老的内核通常在IP层处理IP分段，IP层可以接收0~64KB的数据。因此，当数据IP packet大于PMTU时，就必须把数据分成多个IP分段。较新的内核中，L4会尝试进行分段：L4不会再把超过PMTU的缓冲区直接传给IP层，而是传递一组和PMTU相匹配的缓冲区。这样，IP层只需要给每个分段增加IP报头。但是这并不意味着IP层就不做分段的工作了，一些情况下，IP层还会进行分段操作。

- 分段是指将一个IP包分成多个传输，在接收端IP层重新组装
- 一个IP包能否分包，取决于它的DF标志位：DF bit (0 = "may fragment," 1 = "don't fragment")
- 分包后，每个分段有MF标志位：MF bit (0 = "last fragment," 1 = "more fragments")

### Original IP Datagram

Sequence	Identifier	Total Length	DF May / Don't	MF Last / More	Fragment Offset
0	345	5140	0	0	0

### IP Fragments (Ethernet)

Sequence	Identifier	Total Length	DF May / Don't	MF Last / More	Fragment Offset
0-0	345	1500	0	1	0
0-1	345	1500	0	1	185
0-2	345	1500	0	1	370
0-3	345	700	0	0	555

第一个表格中：

- IP包长度5140，包括5120 bytes的payload
- DF = 0，允许分包
- MF = 0，这是未分包

第二个表格中：

- 0-0 第一个分包：长度  $1500 = 1480$  (payload) + 20 (IP Header). Offset(起始偏移量): 0
- 0-1 第二个分包：长度  $1500 = 1480$  (payload) + 20 (IP Header). Offset:  $185 = 1480 / 8$
- 0-2 第三个分包：长度  $1500 = 1480$  (payload) + 20 (IP Header). Offset:  $370 = 185 + 1480 / 8$
- 0-3 第四个分包：长度  $700 = 680$  (payload, =  $(5140 - 20) - 1480 * 3$ ) + 20 (IP Header). Offset:  $555 = 370 + 1480 / 8$

需要注意的是，只有第一个包带有原始包的完整IPv4+TCP/UDP信息，后续的分包只有IPv4信息。

分包带来的问题：

- **sender overhead**：需要消耗 CPU 去分包，包括计算和数据拷贝。
- **receiver overhead**：重新组装多个分包。在路由器上组装非常低效率，因此组装往往在接收主机上进行。
- 重发 **overhead**：一个分包丢失，则整个包需要重传。
- 在多个分包出现顺序错开时，防火墙可能将分到当无效包处理而丢弃。

## MTU

一个网络接口的 MTU 是它一次所能传输的最大数据块的大小。任何超过MTU的数据块都会在传输前分成小的传输单元。MTU 有两个测量层次：网络层和链路层。比如，网络层上标准的因特网 MTU 是 1500 bytes，而在连接层上是 1518 字节。没有特别说的时候，往往指的是网络层的MTU。

要增加一个网络接口 MTU 的常见原因是增加高速因特网的吞吐量。标准因特网 MTU 使用 1500byte 是为了和 10M 和 100M 网络后向兼容，但是，在目前1G和 10G网络中远远不够。新式的网络设备可以处理更大的MTU，但是，MTU需要显式设置。这种更大MTU的帧叫做“巨帧”，通常 9000 byte 是比较普遍的。

相对地，一些可能得需要减少MTU的原因：

- 满足另一个网络的MTU的需要（为了消除UDP分包，以及需要TCP PMTU discover）
- 满足 ATM cell 的要求
- 在高出错率线路上提高吞吐量

MTU 不能和目前任何 Internet 网络协议混在一起，但是，可以使用一个路由器将不同 MTU 的网段连在一起。

## TCP fragmentation

每个TCP数据包（segment）的大小受MSS (TCP\_MAXSEG选项) 限制。最大报文段长度 (MSS) 表示 TCP 传往另一端的最大块数据的长度。当一个连接建立时 (SYN packet)，连接的双方都要通告各自的MSS。

一般说来,如果没有分段发生, MSS还是越大越好。报文段越大允许每个报文段传送的数据就越多,相对IP和TCP首部有更高的网络利用率。当TCP发送一个SYN时,或者是因为一个本地应用进程想发起一个连接,或者是因为另一端的主机收到了一个连接请求,它能将MSS值设置为外出接口上的MTU长度减去固定的IP首部(20 bytes)和TCP首部长度(20 bytes)。对于一个以太网，MSS值可达1460字节（详细参考tcp\_sendmsg）。

TCP/SCTP会将数据按MTU进行切片，然后3层的工作只需要给传递下来的切片加上ip头就可以了(也就是说调用这个函数的时候,其实4层已经切好片了)。

## Segmentation offload

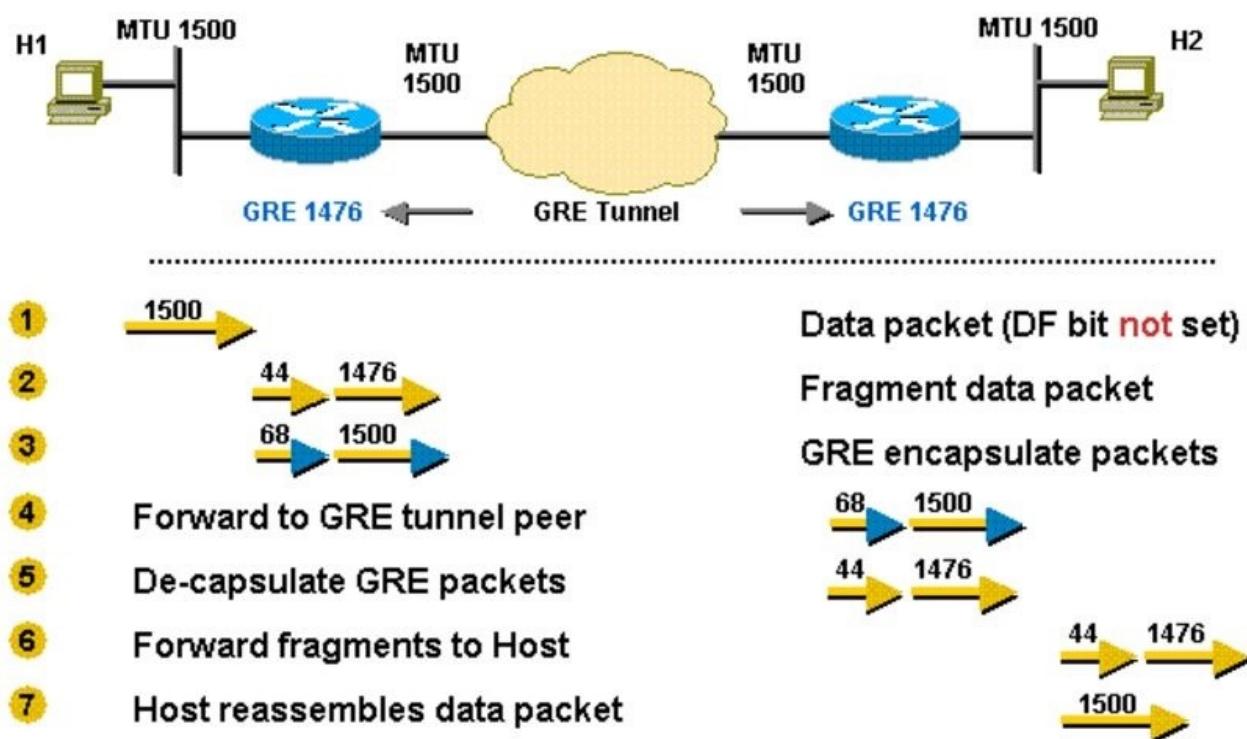
现在很多网卡本身支持数据分片，这样，上层L4/L3就可以不用进行分片(最大64KB)，而由NIC来完成，从而提高网络性能。

- Large Segment Offload (LSO)：使得网络协议栈能够将超过PMTU的数据包推送至网卡，然后网卡执行分片工作，这样减轻了CPU的负荷
- TCP Segmentation Offload (TSO)：类似于LSO，针对TCP协议包
- UDP Fragmentation Offload (UFO): 类似于TSO，针对UDP包
- Large Receive Offload (LRO): 将接收到的包聚合成一个大的数据包，然后再发给协议栈处理
- Generic Segmentation Offload (GSO): TSO/LSO的增强，同时支持TCP和UDP协议，负责把超过MTU的包分片
- Generic Receive Offload (GRO)：LRO的增强，负责将接收到的多个包聚合成一个大的数据包，然后再发给协议栈处理

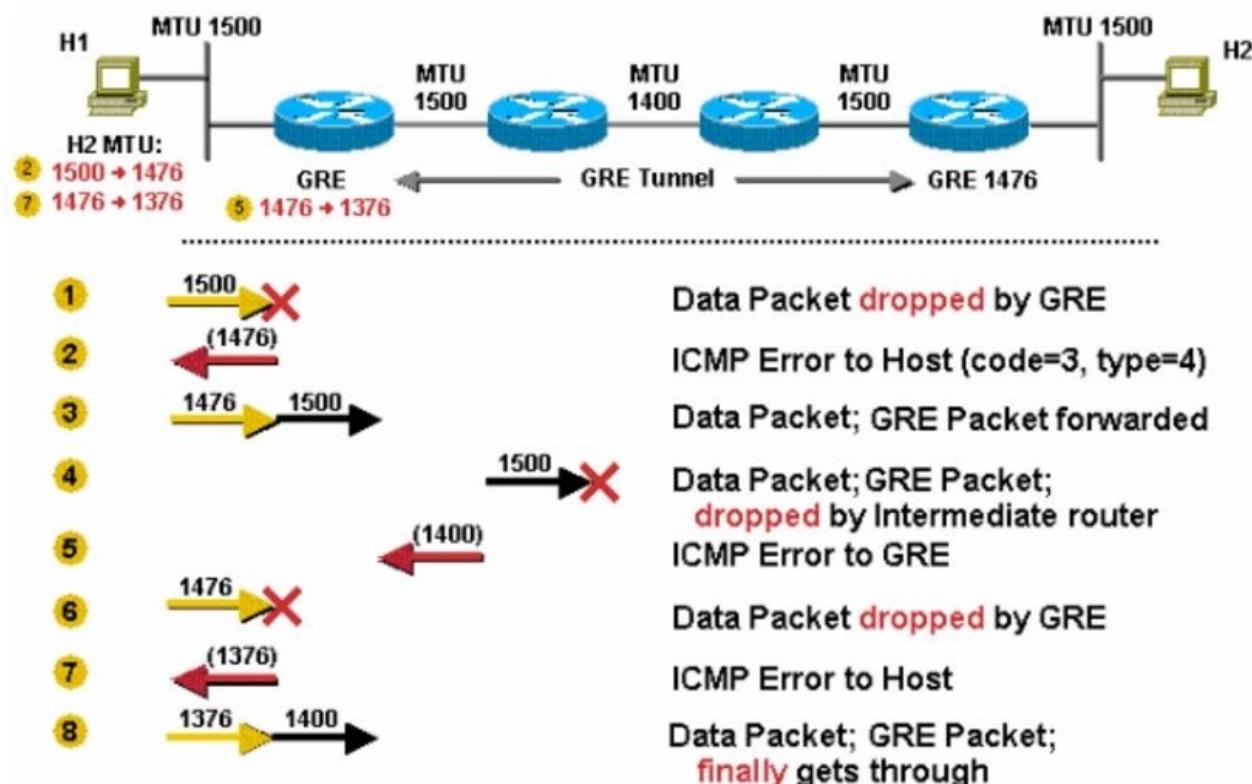
## PMTU (Path Maximum Transmission Unit Discovery)

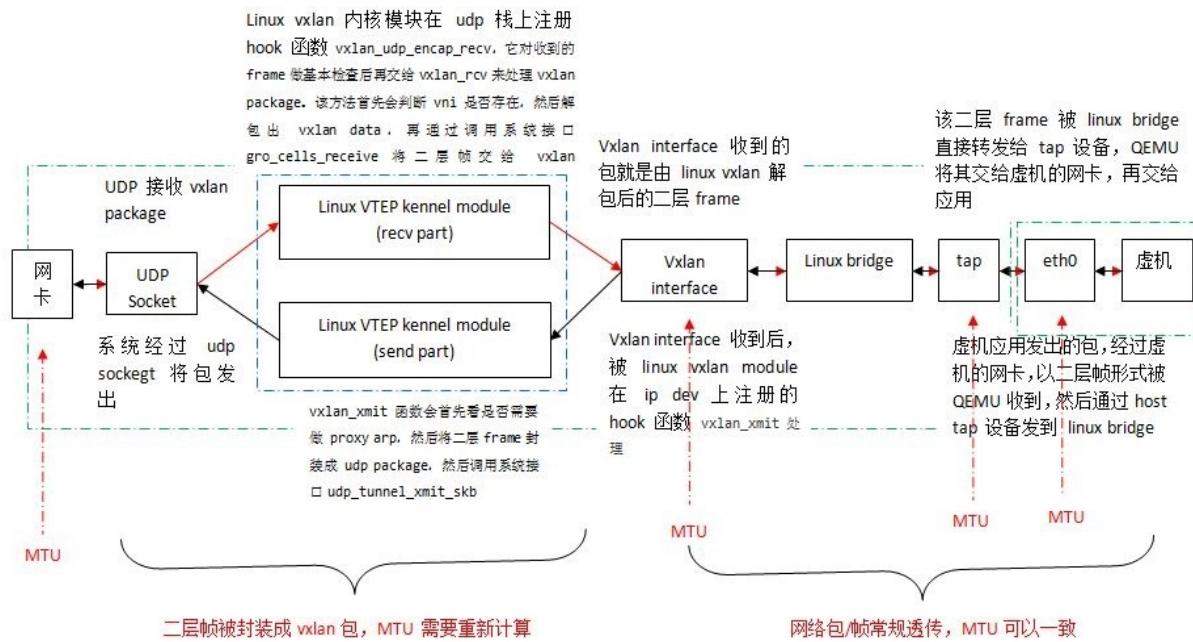
PMTU 的用途是动态的确定从发送端到接收端整个路径上的最小 MTU，从而避免分包。注意，PMTU 只支持 TCP，对其他协议比如 UDP 无效。而且，如果发送方已经开启了 PMTU，那么它发送的所有 TCP/IP 包的 DF 标志都被设置为 1 即不再允许分包。当网络路径上某个路由器发现发送者的包因为超过前面转发路径的 MTU 而无法发送时，它向发送者返回一个 ICMP "Destination Unreachable" 消息，其中包含了那个 MTU，然后发送者就会在它的路由表中将该mtu值保存下来，再使用较小的 MTU 重新发出新的较小的包。

例子1：超过 MTU，DF = 0 => 路由器分包、发送，接收主机组装



例子2：超过，DF = 1 => PMTU，发送者重新以小包发送





## 参考文档

- [1] <http://www.cnblogs.com/sammyliu/p/5079898.html>
- [2] <http://www.cisco.com/c/en/us/support/docs/ip/generic-routing-encapsulation-gre/25885-pmtud-ipfrag.html>
- [3] [http://blog.csdn.net/opens\\_tym/article/details/17658569](http://blog.csdn.net/opens_tym/article/details/17658569)

# ARP

链路层通信根据48bit以太网地址（硬件地址）来确定目的接口，而地址解析协议负责32bit IP地址与48bit以太网地址之间的映射：

- (1) ARP负责将IP地址映射到对应的硬件地址
- (2) RARP负责相反的过程，通常用于无盘系统。

## ARP高速缓存

ARP高效运行的关键是每台主机上都有一个ARP高速缓存，缓存中每一项的生存时间一般为20分钟，但不完整表项超时时间为3分钟（如192.168.13.254）。

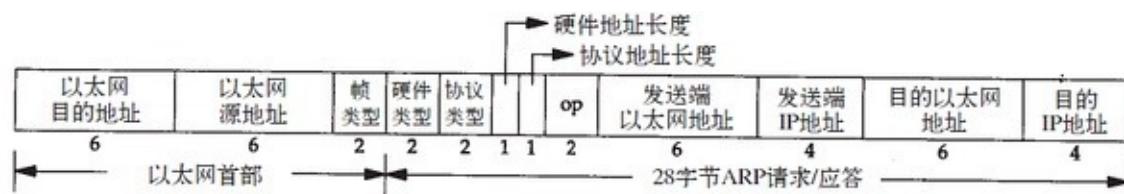
```
# arp -a

? (192.168.0.16) at 00:1b:21:b9:9f:d4 [ether] on eth0
? (192.168.0.6) at 00:1b:21:b9:9f:d4 [ether] on eth0
? (192.168.13.233) at 00:16:3e:01:7a:b2 [ether] on eth0
? (192.168.13.254) at <incomplete> on eth0
```

可以通过arp命令来操作ARP高速缓存：

- arp 显示当前的ARP缓存列表。
- arp -s ip mac 添加静态ARP记录，如果需要永久保存，应该编辑/etc/ethers文件。
- arp -f 使/etc/ethers中的静态ARP记录生效。

## ARP分组格式



其中：

- ARP协议的帧类型为0x0806
- 硬件类型：1表示以太网地址
- 协议类型：0x800表示IP协议
- 硬件地址长度：值为6
- 协议地址长度：值为4

- op：1为ARP请求，2为ARP应答，3为RARP请求，4为RARP应答
- 对于ARP请求来说，目的端硬件地址为广播地址f:ff:ff:ff:ff:ff），由ARP相应的主机填入。

一个完整ARP请求应答的抓包：

```
# tcpdump -e -p arp -n -vv
21:08:10.329163 00:16:3e:01:79:43 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 42: Ethernet (len 6), IPv4 (len 4), Request who-has 192.168.14.23 tell 192.168.13.43, length 28
21:08:10.329626 00:16:3e:01:7b:17 > 00:16:3e:01:79:43, ethertype ARP (0x0806), length 60: Ethernet (len 6), IPv4 (len 4), Reply 192.168.14.23 is-at 00:16:3e:01:7b:17, length 46
```

## ARP代理

当向位于不同网络的主机发送ARP请求时，由两个网络之间的路由器响应该ARP请求，这个过程称为ARP代理。

ARP代理也称为混合ARP：通过两个网络之间的路由器可以互相隐藏物理网络。

## 免费ARP

主机发送ARP请求查找自己的IP地址。通常有两个用途：

- (1) 确认网络中是否有其他主机设置了相同的IP地址；
- (2) 当主机的物理地址改变了，可以通过免费ARP更新更新路由器和其他主机中的高速缓存。

## RARP

1. RARP通常用于无盘系统，无盘系统从物理网卡上读到硬件地址后，发送一个RARP请求查询自己的IP地址。
2. RARP的协议格式：与ARP协议一致，只不过帧类型代码为0x8035
3. RARP使用链路层广播，这样阻止了大多数路由器转发ARAP请求，只返回很小的信息，即IP地址。

# ICMP

## ICMP协议格式

ICMP报文是在IP数据报内部传输的： | IP头部 | ICMP报文 |

### ICMP报文格式

Bits	0-7	8-15	16-23	24-31
0	Type	Code		Checksum
32	Rest of Header			

- Type – ICMP type as specified below.
- Code – Subtype to the given type.
- Checksum – Error checking data. Calculated from the ICMP header+data, with value 0 for this field. The checksum algorithm is specified in RFC 1071.
- Rest of Header – Four byte field. Will vary based on the ICMP type and code.

ICMP报文可以分为两类：查询报文和差错报文，具体报文类型如下图所示：

类 型	代 码	描 述	查 询	差 错
0	0	回显应答(Ping应答, 第7章)	•	
3		目的不可达: 0 网络不可达(9.3节) 1 主机不可达(9.3节) 2 协议不可达 3 端口不可达(6.5节) 4 需要进行分片但设置了不分片比特(11.6节) 5 源站选路失败(8.5节) 6 目的网络不认识 7 目的主机不认识 8 源主机被隔离(作废不用) 9 目的网络被强制禁止 10 目的主机被强制禁止 11 由于服务类型TOS, 网络不可达(9.3节) 12 由于服务类型TOS, 主机不可达(9.3节) 13 由于过滤, 通信被强制禁止 14 主机越权 15 优先权中止生效		• • • • • • • • • • • • • • • • •
4	0	源端被关闭(基本流控制, 11.11节)		•
5		重定向(9.5节): 0 对网络重定向 1 对主机重定向 2 对服务类型和网络重定向 3 对服务类型和主机重定向		• • • •
8	0	请求回显(Ping请求, 第7章)	•	
9	0	路由器通告(9.6节)	•	
10	0	路由器请求(9.6节)	•	
11		超时: 0 传输期间生存时间为0(Traceroute, 第8章) 1 在数据报组装期间生存时间为0(11.5节)		• •
12		参数问题: 0 坏的IP首部(包括各种差错) 1 缺少必需的选项		• •
13	0	时间戳请求(6.4节)	•	
14	0	时间戳应答(6.4节)	•	
15	0	信息请求(作废不用)	•	
16	0	信息应答(作废不用)	•	
17	0	地址掩码请求(6.3节)	•	
18	0	地址掩码应答(6.3节)	•	

下面各种情况都不会导致产生ICMP差错报文：

1) ICMP差错报文(但是, ICMP查询报文可能会产生ICMP差错报文)。2) 目的地是广播地址或多播地址的IP数据报。3) 作为链路层广播的数据报。4) 不是IP分片的第一片。5) 源地址为零地址、环回地址、广播地址或多播地址。

这些规则是为了防止过去允许ICMP差错报文对广播分组响应所带来的广播风暴。

## ICMP地址掩码请求

ICMP地址掩码请求用于无盘系统启动时获取自己的子网掩码。

构造一个ICMP Address Mask Request：

```
#We want to send an ICMP packet Address Mask Request and wait 10 seconds to see the replies. We mask the packet with source address of 10.2.3.4 and we send it to the address 10.0.1.255:
icmpush -mask -sp 10.2.3.4 -to 10 10.0.1.255
```

注意：ICMP地址掩码应答必须是收到请求接口的子网掩码

## ICMP时间戳请求与应答

ICMP时间戳请求允许系统向另一个系统查询当前的时间，返回的建议值是自午夜开始计算的毫秒数，协调的统一时间，可以达到毫秒的分辨率。

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31																				
Type = 14	Code = 0										Header Checksum																																								
Identifier										Sequence Number																																									
Originate Timestamp																																																			
Receive Timestamp																																																			
Transmit Timestamp																																																			

构造一个ICMP时间戳请求： icmpush -tstamp 192.168.3.255

## ICMP端口不可达差错

根据code的不同，共有15种类型的ICMP差错报文。

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31																					
Type = 3	Code										Header Checksum																																									
Empty										Next-Hop MTU																																										
IP Header + First 8 Bytes of Original Datagram's Data																																																				

注意，ICMP报文是在主机之间交换的，而不用目的端口号，而UDP数据报则是从一个特定端口发送到另一个特定端口。

ICMP的一个规则是，ICMP差错报文必须包括生成该差错报文的数据报IP首部（包含任何选项），还必须至少包括跟在该IP首部后面的前8个字节。导致差错的数据报中的IP首部要被退回的原因是因为IP首部中包含了协议字段，使得ICMP可以知道如何解释后面的8个字节。对于TCP和UDP协议来说，这8个字节正好是源端口号和目的端口号。

## ping

ping通过ICMP回显请求和应答实现。

```
# setup ping interval in seconds
ping -i 5 IP

# Check whether the local network interface is up and running
ping 0

# Set packet num
ping -c 5 google.com

# Set packet size
ping -s 100 localhost
```

## traceroute

ping程序提供一个记录路由选项，但并不是所有的路由器都支持这个选项，而且IP首部选项字段最多也只能存储9个IP地址，因此开发traceroute是必要的。traceroute利用了ICMP报文和IP首部的TTL字段。TTL是一个8bit的字段，为路由器的跳站计数器，也表示数据报的生存周期。每个处理数据报的路由器都需要将TTL减一。如果TTL为0或者1，则路由器不转发该数据报，如果TTL为1，路由器丢弃该包并给源地址发送一个ICMP超时报文（如果是主机接收到TTL为1的数据报可以交给上层应用程序）。

traceroute程序开始时发送一个TTL字段为1的UDP数据报（选择一个不可能的值作为UDP端口号），然后将TTL每次加1，以确定路径中每个路由器。每个路由器在丢弃UDP数据报的时候都返回一个ICMP超时报文（如：ICMP time exceeded in-transit, length 36），而最终主机则产生一个ICMP端口不可达报文（如： ICMP 74.125.128.103 udp port 33492 unreachable, length）。

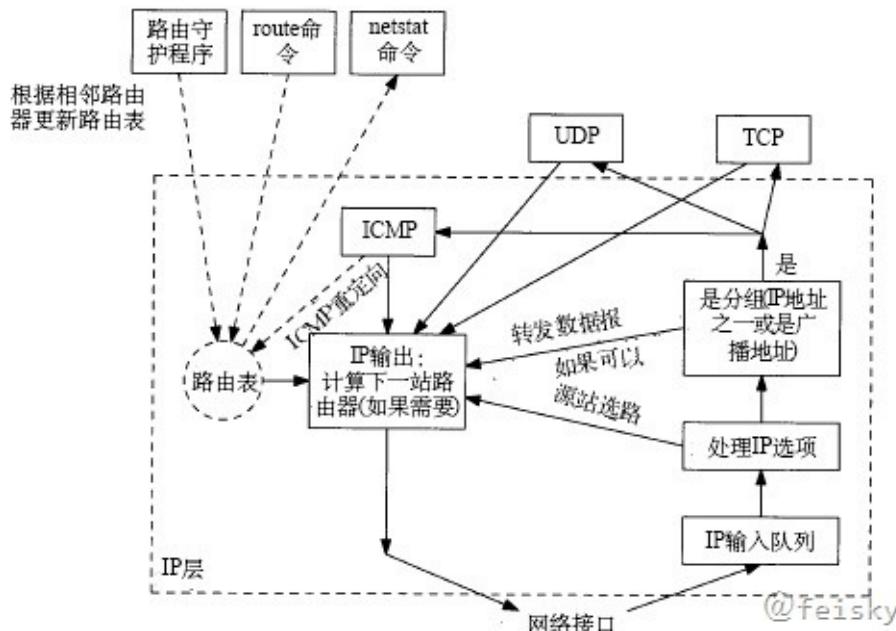
对每个TTL，发送3份数据报，并且计算打印出往返时间。如果5秒内未收到任意一份回应，则打印一个星号。

需要注意的是：

1. 并不能保证现在的路由就是将来所采用的路由；
2. 不能保证ICMP报文的路由与traceroute程序发出的UDP数据报采用同一路由；
3. 返回的ICMP报文中信源的IP地址是UDP数据报到达的路由器接口的IP地址。

# 路由

## IP选路



### 1. 搜索路由表的优先级

- 主机地址
- 网络地址
- 默认路由

### 2. 路由表

3. 如果找不到匹配的路由，则返回“主机不可达差错”或“网络不可达差错”

一个典型的路由表如下：

```
# route -n
Kernel IP routing table
Destination      Gateway          Genmask         Flags Metric Ref  Use Iface
192.168.0.0      0.0.0.0        255.255.192.0   U     0      0      0 eth0
0.0.0.0          192.168.0.1    0.0.0.0        UG    100    0      0 eth0
```

Flags各项的含义：

- U 该路由可用
- G 该路由是一个网关，如果没有该标志，则是直接路由
- H 该路由是一个主机，如果没有该标志，则是一个网络
- D 该路由是由重定向报文创建的

- M 该路由被ICMP重定向报文修改过

## 路由的修改

可以通过route命令来修改路由表，ICMP重定向报文也会修改路由表

一般在系统的配置文件中会设置默认路由。

## ICMP重定向差错

当IP数据报应该被发送到另一个路由器时，收到数据报的路由器就要给发送端回复一个ICMP重定向差错报文。

重定向一般用来让具有很少选路信息的主机逐渐建立更完善的路由表。

## ICMP路由器发现报文

一般来说，主机在引导以后要广播或多播一份路由器请求报文，一台或多台路由器响应一份路由器通告报文。路由器也会定期地广播或多播路由器通告报文。

## 路由协议

路由协议用来从多条路由路径中选择一条最佳的路径，并沿着这条路径将数据流送到目的设备。

- 路由信息协议（RIP）：采用距离向量算法，收集所有可到达目的地的不同路径，并且保存有关到达每个目的地的最少站点数的路径信息；同时路由器也把这些信息用RIP协议通知相邻路由器。RIP只适用于小型网络（最大15跳）。
- 开放式最短路径协议（OSPF）：基于链路状态，每个路由器向其同一管理域的所有其它路由器发送链路状态广播信息，并将自治域划分为区，并根据区的位置执行区内路由选择和区间路由选择。
- IS-IS：链路状态路由协议，和OSPF相同，IS-IS也使用了“区域”的概念，同样也维护着一份链路状态数据库，通过最短生成树算法（SPF）计算出最佳路径。
- 边界网关协议（BGP）：外部网关协议，用于与其它自治域的BGP交换网络可达信息（通过TCP确保可靠性）。

## STP和Trill

为了提高网络的可靠性，交换网络通常会使用冗余链路，这会带来环路的风险。而STP和Trill就是为了解决环路问题而生的。

STP（Spanning Tree Protocol）的基本原理是在交换机之间传输BPDU（Bridge Protocol Data Unit）报文，并使用生成树来确定网络拓扑：

- 生成树初始化，建立根网桥
- 根端口选举，选择的依据是端口到根网桥的路径开销最小，如果路径开销相同则使用端口ID最小的端口
- 网段指定端口选举，选择的依据也是到根网桥路径开销最小
- 网络收敛后，只有指定端口和根端口可以转发数据。其他端口为预备端口，被阻塞，不能转发数据

STP最大的问题是二层链路利用率不足，且收敛慢，不适合大型数据中心。IETF又提出了Trill技术来克服STP的种种缺陷。Trill（TRansparent Interconnection of Lots of Links）的核心思想是将成熟的三层路由的控制算法引入到二层交换中，将原先的L2报文加一个新的封装(隧道封装)，转换到新的地址空间上进行转发。而新的地址有与IP类似的路由属性，具备大规模组网、最短路径转发、等价多路径、快速收敛、易扩展等诸多优势，从而规避STP/MSTP等技术的缺陷，实现健壮的大规模二层组网。支持TRILL技术的以太网交换机被称为 RBridge。

## MPLS

MPLS（Multi-Protocol Label Switching）利用标签进行数据转发，而不是向传统路由决策那样每次数据包进行解包，大大减少了路由决策的时间。当分组进入MPLS网络时，为其分配固定长度的短标记，并将标记与分组封装在一起，在整个转发过程中，交换节点仅根据标记进行转发。

## 以太网交换机

交换机是最重要的信息交换网络设备，主要功能包括

- 学习设备MAC地址
- 二层转发
- 三层转发
- ACL
- QoS
- 消除回路

随着SDN和NFV的发展，现在已经有越来越多的功能都放到了虚拟交换机上来。最常见的虚拟交换机是[Open vSwitch](#)。

## 三层交换机与路由器

三层交换机也支持三层转发（即路由），解决了路由器带宽和性能受限的问题：交换机通过交换芯片转发数据，而路由器则是通过CPU转发的。那么它与路由器相比有什么不同呢

- 三层交换机同时支持二层和三层转发，而路由器则仅支持三层转发
- 交换机针对以太网研发，对其他网络类型支持较少；而路由器则支持较多的网络类型，更适合用在网络复杂的场景下

## 白牌交换机

随着SDN的兴起，白牌交换机（WhiteBox Switch）逐渐兴起。白牌交换机是指不贴标签的交换机，并且像PC一样，硬件和软件分离。用户从厂商拿到硬件后，可以安装自定义的软件（如OpenSwitch、OPX、Sonic等）。比如，用户可以选择

- 硬件使用Broadcom、Cavium、盛科等
- 软件使用Cumulus、BigSwitch、Pica8、Snaproute、OPX、OpenSwitch等

白牌交换机的优势

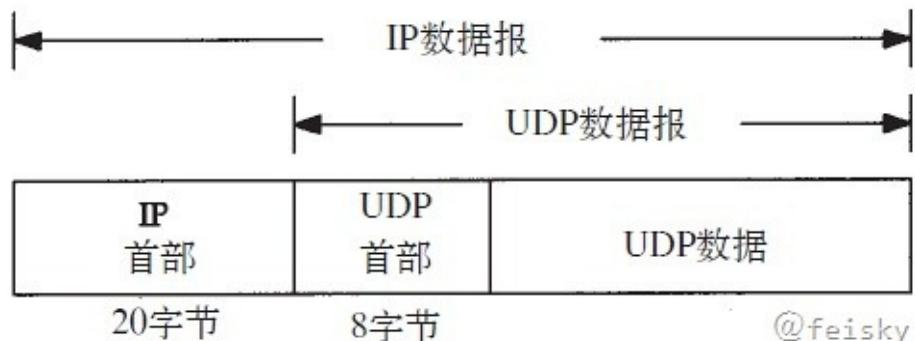
- 成本低，白牌交换机的总体成本低于品牌设备
- 更大的灵活性，软件可以自定义，方便针对特定需求和场景做定制开发
- 避免厂商绑定

当然了，并不是所有的软件都适用于所有的硬件，这还有待软硬件厂商进一步协作提升开放性。



# UDP

UDP是一种对象数据报的传输层协议，它不提供可靠性，其数据报被封装在IP数据报中，封装格式如下图所示：



首部格式为



- 源端口号和目的端口号分表表示了发送进程和接收进程
- UDP长度字段包括了UDP首部和UDP数据的字节长度
- UDP检验和覆盖了UDP首部和UDP数据（IP首部检验和只覆盖了IP首部，不覆盖数据报中的任何数据）
- UDP数据报的长度可以为奇数字节，但是检验和算法是把若干个16bit字相加。解决方法是必要时在最后增加填充字节0，这只是为了检验和的计算。UDP数据报和TCP段都包含一个12字节长的伪首部，它是为了计算检验和而设置的。伪首部包含IP首部一些字段。

# IP分片

以太网和802.3对数据帧的长度都有一个限制，其最大值分别是1500和1492个字节。链路层的这个特性称作MTU。不同类型的网络大多数都有一个上限。如果IP层有一个数据要传，且数据的长度比链路层的MTU还大，那么IP层就要进行分片（fragmentation），把数据报分成

若干片，这样每一个分片都小于MTU。当IP数据报被分片后，每一片都成为一个分组，具有自己的IP首部，并在选择路由时与其他分组独立。

把一份IP数据报进行分片以后，由到达目的端的IP层来进行重新组装，其目的是使分片和重新组装过程对运输层（TCP/UDP）是透明的。由于每一片都是一个独立的包，当这些数据报的片到达目的端时有可能会失序，但是在IP首部中有足够的信息让接收端能正确组装这些数据报片。

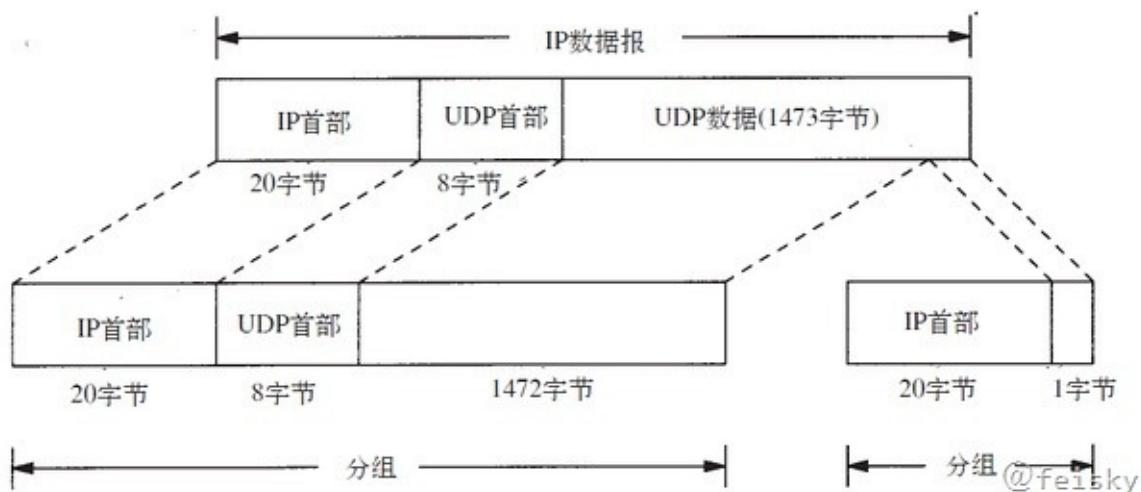
尽管IP分片过程看起来透明的，但有一点让人不想使用它：即使只丢失一片数据也要重新传整个数据报。

why？因为IP层本身没有超时重传机制——由更高层（比如TCP）来负责超时和重传。当来自TCP报文段的某一片丢失后，TCP在超时后会重发整个TCP报文段，该报文段对应于一份IP数据报（而不是一个分片），没有办法只重传数据报中的一个数据分片。

使用UDP很容易导致IP分片，TCP试图避免IP分片。那么TCP是如何试图避免IP分片的呢？其实说白了，采用TCP协议进行数据传输是不会造成IP分片的，因为一旦TCP数据过大，超过了MSS，则在传输层会对TCP包进行分段（如何分，见下文！），自然到了IP层的数据报肯定不会超过MTU，当然也就不用分片了。而对于UDP数据报，如果UDP组成的IP数据报长度超过了1500，那么IP数据报显然就要进行分片，因为UDP不能像TCP一样自己进行分段。

MSS（Maximum Segment Size）最大分段大小的缩写，是TCP协议里面的一个概念

- 1) MSS就是TCP数据包每次能够传输的最大数据分段。为了达到最佳的传输效能TCP协议在建立连接的时候通常要协商双方的MSS值，这个值TCP协议在实现的时候往往用MTU值代替（需要减去IP数据包包头的大小20Bytes和TCP数据段的包头20Bytes）所以往往MSS为1460。通讯双方会根据双方提供的MSS值得最小值确定为这次连接的最大MSS值。
- 2) 相信看到这里，还有最后一个问题：TCP是如何实现分段的呢？其实TCP无所谓分段，因为每个TCP数据报在组成前其大小就已经被MSS限制了，所以TCP数据报的长度是不可能大于MSS的，当然由它形成的IP包的长度也就不会大于MTU，自然也就不用IP分片了。



- 发生ICMP不可达差错的另一种情况是，当路由器收到一份需要分片的数据报，而在IP首部又设置了不分片（DF）的标志比特。如果某个程序需要判断到达目的端的路途中最小MTU是多少—称作路径MTU发现机制，那么这个差错就可以被该程序使用。
- 理论上，UDP数据的最大长度为：65535-20字节IP首部长度-8字节UDP首部长度=65507。但是大多是实现都比这个值小，主要是受限于socket接口以及TCP/IP内核的限制。大部分系统都默认提供了可读写大于8192字节的UDP数据报。
- 当目标主机的处理速度赶不上数据接收的速度，因为接受主机的IP层缓存会被占满，所以主机就会发出一个ICMP源站抑制差错报文。

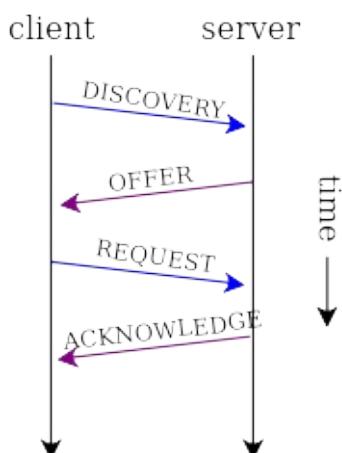
# DHCP和DNS

## DHCP

DHCP（Dynamic Host Configuration Protocol）是一个用于主机动态获取IP地址的配置解析，使用UDP报文传送，端口号为67何68。

DHCP使用了租约的概念，或称为计算机IP地址的有效期。租用时间是不定的，主要取决于用户在某地连接Internet需要多久，这对于教育行业和其它用户频繁改变的环境是很实用的。通过较短的租期，DHCP能够在一个计算机比可用IP地址多的环境中动态地重新配置网络。

DHCP支持为计算机分配静态地址，如需要永久性IP地址的Web服务器。



## DNS

DNS（Domain Name System）是一个解析域名和IP地址对应关系以及电子邮件选路信息的服务。它以递归的方式运行：首先访问最近的DNS服务器，如果查询到域名对应的IP地址则直接返回，否则的话再向上一级查询。DNS通常以UDP报文来传送，并使用端口号53。

从应用的角度来看，其实就是两个库函数`gethostbyname()`和`gethostbyaddr()`。

**FQDN：**全域名(FQDN，Fully Qualified Domain Name)是指主机名加上全路径，全路径中列出了序列中所有域成员(包括root)。全域名可以从逻辑上准确地表示出主机在什么地方，也可以说全域名是主机名的一种完全表示形式。

### 资源记录 (RR)

- A记录：用于查询IP地址
- PTR记录：逆向查询记录，用于从IP地址查询域名

- CNAME：表示“规范名字”，用来表示一个域名，也通常称为别名
- HINFO：表示主机信息，包括主机CPU和操作系统的两个字符串
- MX：邮件交换记录
- NS：名字服务器记录，即下一级域名信息的服务器地址，只能设置为域名，不能是IP

### 高速缓存

为了减少DNS的通信量，所有的名字服务器均使用高速缓存。在标准Unix实现中，高速缓存是由名字服务器而不是名字解释器来维护的。

### 用UDP还是TCP

DNS服务器支持TCP和UDP两种协议的查询方式，而且端口都是53。而大多数的查询都是UDP查询的，一般需要TCP查询的有两种情况：

1. 当查询数据过大以至于产生了数据截断(TC标志为1)，这时，需要利用TCP的分片能力来进行数据传输（看TCP的相关章节）。
2. 当主（master）服务器和辅（slave）服务器之间通信，辅服务器要拿到主服务器的zone信息的时候。

### 示例

```

$ dig k8s.io
; <>> DiG 9.10.3-P4-Ubuntu <>> k8s.io
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 37946
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;k8s.io.           IN      A

;; ANSWER SECTION:
k8s.io.        299     IN      A      23.236.58.218

;; Query time: 392 msec
;; SERVER: 169.254.169.254#53(169.254.169.254)
;; WHEN: Mon Sep 11 05:50:37 UTC 2017
;; MSG SIZE  rcvd: 51

# 反向查询
$ dig -x 23.236.58.218
; <>> DiG 9.10.3-P4-Ubuntu <>> -x 23.236.58.218
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 7130
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;218.58.236.23.in-addr.arpa.    IN      PTR

;; ANSWER SECTION:
218.58.236.23.in-addr.arpa. 119     IN      PTR      218.58.236.23.bc.googleusercontent.com
.

;; Query time: 158 msec
;; SERVER: 169.254.169.254#53(169.254.169.254)
;; WHEN: Mon Sep 11 05:50:45 UTC 2017
;; MSG SIZE  rcvd: 107

```

## FAQ

### **dnsmasq bad DHCP host name 问题**

这个问题是由于hostname是数字前缀，并且dnsmasq对版本低于2.67，这个问题在[2.67版本中修复](#)：

Allow hostnames to start with a number, as allowed in RFC-1123. Thanks to Kyle Mestery for the patch.

# TCP

TCP的特性

1. TCP提供面向连接的、可靠的字节流服务
2. 上层应用数据被TCP分割为TCP认为合适的报文段
3. TCP使用超时重传机制，而接收到一个TCP数据后需要发送一个确认
4. TCP使用包含了首部和数据的校验和来检查数据是否在传输过程中发生了差错
5. TCP可以将失序的报文重新排序
6. TCP连接的每一端都有固定大小的缓冲区，只允许另一端发送发送接收缓冲区所能接纳的数据
7. TCP提供面向字节流的服务，不在字节流中插入记录标识符，也不对字节流的内容作任何解释（由上层应用解释）

## TCP首部

TCP数据也是封装在IP数据报中，TCP首部格式如下图所示：



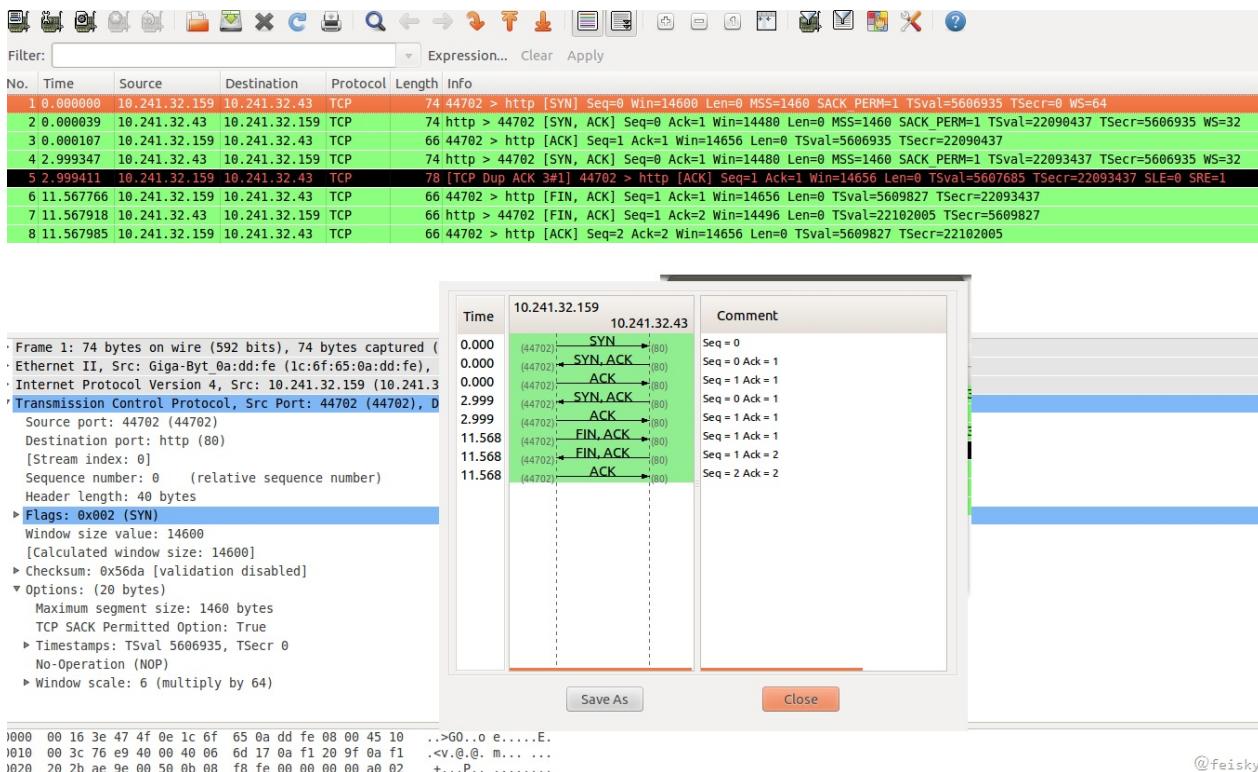
其中，

- 序列号：用于对报文进行计数（注SYN和FIN都会消耗一个序列号），TCP为应用层提供全双工服务，连接的每一端都要保持每个方向上的传输序列号
- SYN：用来发起一个连接，当新建一个链接时，SYN变为1
- ACK：确认序号有效，其序列号为上次接收的序号加1
- 首部长度：首部中32bit的长度（最多60字节），如果没有任选字段，长度为20字节

- URG：标志紧急指针有效
- PSH：接收方应该尽快将这个报文交给应用层
- RST：重建连接
- FIN：发端完成发送任务
- 窗口大小：用于TCP的流量控制，最大65535字节
- 检验和：覆盖首部和数据，由发端计算和存储，接收端验证
- 紧急指针：只有当URG为1时才有效，用于发送紧急数据
- 数据部分是可选的，在连接建立和终止时，双方交换的报文中只有TCP首部

TCP可以表述为一个没有选择确认或否认的滑动窗口协议（滑动窗口协议用于数据传输）。我们说TCP缺少选择确认是因为TCP首部中的确认序号表示发方已成功收到字节，但还不包含确认序号所指的字节。当前还无法对数据流中选定的部分进行确认。例如，如果1~1024字节已经成功收到，下一报文段中包含序号从2049~3072的字节，收端并不能确认这个新的报文段。它所能做的就是发回一个确认序号为1025的ACK。它也无法对一个报文段进行否认。例如，如果收到包含1025~2048字节的报文段，但它的检验和错，TCP接收端所能做的就是发回一个确认序号为1025的ACK。

## TCP连接过程



上图由wireshark抓取，并显示了TCP状态图（注意：由于网络阻塞，发生了丢包现象，4是对2的重发，而5是对4的响应（同3相同））。

根据上图可以看到建立一个TCP连接的过程为（三次握手的过程）：

1. 客户端向服务器端发送一个SYN请求，同时传送一个初始序列号（ISN）；
2. 服务器发回包含客户端初始序列号的SYN报文段作为应答，同时将ACK序号设置为ISN+1；
3. 客户端向服务器发送一个ACK确认，ACK序号为ISN+1.

终止一个TCP连接需要4次握手，这是由于TCP的半关闭（当一方调用shutdown关闭连接后，另一端还是可以发送数据，典型的例子为rsh）导致的：TCP连接是全双工的，连接的每一端在关闭连接时都向对方发送一个FIN来终止连接，同时对方会对其进行确认（回复ACK）。通常，都是一方完成主动关闭，另一方来完成被动关闭：

1. 以上面的抓包为例，客户端向服务器发送了一个FIN（NO. 6）；
2. 服务器端对上面的FIN进行确认（NO. 7），同时向客户端发送一个FIN（这儿其实是两个动作，一个是对上面FIN的ACK，另一个是发送一个FIN，但由于TCP的捎带ACK机制，两者放在一个包里发送了）；
3. 客户端对服务器端的FIN进行确认（NO. 8）。

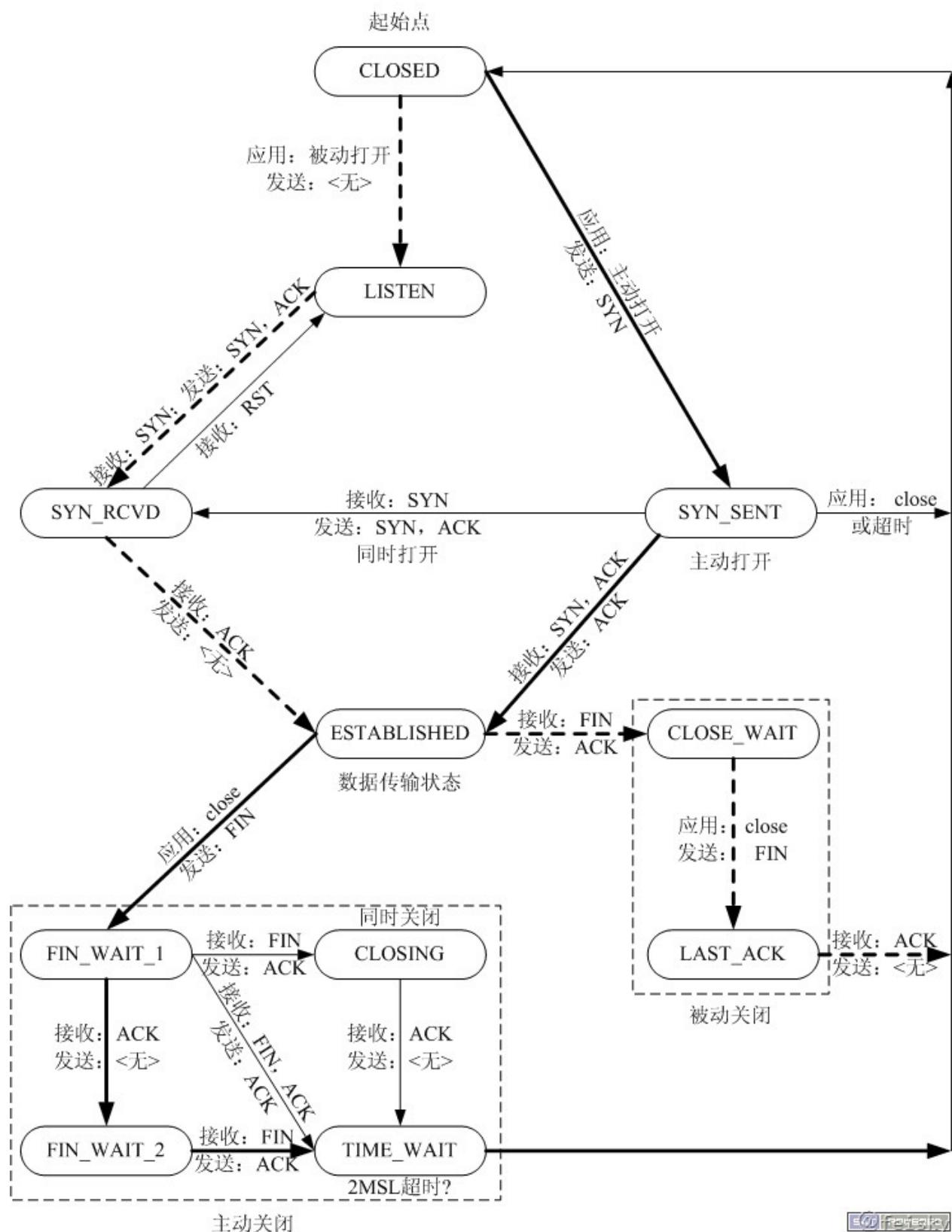
## MSS

最大报文长度（MSS）表示TCP传往另一端的最大块数据的长度。MSS在连接建立时传送给对方，只会出现在SYN报文段中。

MSS让主机限制另一端发送数据报的长度。

## TCP状态变迁图

# TCP 状态转换图



复位报文段

Reset Segment

1. 当连接到一个不在监听的端口时，客户端回收到一个RST响应（UDP连接到一个不存在的端口时会产生一个ICMP端口不可达的差错）。
2. 在连接终止时，也可以通过发送一个复位报文段而不是FIN来终止连接，可通过设置SO\_LINGER来这么做。
3. 可通过TCP的SO\_KEEPALIVE选项来检测半打开连接，当检测到这种连接时会发送一个RST报文。关于该选项更多的内容参见[http://www.tldp.org/HOWTO/html\\_single/TCP-Keepalive-HOWTO/](http://www.tldp.org/HOWTO/html_single/TCP-Keepalive-HOWTO/)。

### SO\_LINGER 选项

此选项指定函数close对面向连接的协议如何操作（如TCP）。内核缺省close操作是立即返回，如果有数据残留在套接口缓冲区中则系统将试着将这些数据发送给对方。

## Nagle 算法

前面可以看到，TCP交互的双方每次发送数据的时候（即便是只有一个字节的数据），都需要产生一个（数据长度+40字节）的分组。当数据的长度远小于40字节时，网络的实际利用率其实很低，并且大量的小分组也会增加拥塞的可能。

Nagle算法正是解决了该问题。它要求一个TCP连接上最多只能有一个未被确认的未完成的小分组，在该分组的确认到达之前不能发送其他的小分组。TCP收集这些小的分组，并在确认到来时以一个分组的形式发出去。其特点是：确认到达的越快，数据也就发送的越快，并可以发送更少的分组。

TCP链接的过程中，默认开启Nagle算法，进行小包发送的优化。优化网络传输，兼顾网络延时和网络拥塞。

Nagle虽然解决了小封包问题，但也导致了较高的不可预测的延迟，同时降低了吞吐量。这个时候可以置位TCP\_NODELAY关闭Nagle算法，有数据包的话直接发送保证网络时效性。

在进行大量数据发送的时候可以置位TCP\_CORK关闭Nagle算法保证网络利用性。尽可能的进行数据的组合，以最大mtu传输，如果发送的数据包大小过小则如果在0.6~0.8S范围内都没能组装成一个MTU时，直接发送。如果发送的数据包大小足够间隔在0.45内时，每次组装一个MTU进行发送。如果间隔大于0.4~0.8S则，每过来一个数据包就直接发送。

Nagle算法和CORK算法非常类似，但是它们的着眼点不一样，Nagle算法主要避免网络因为太多的小包（协议头的比例非常之大）而拥塞，而CORK算法则是为了提高网络的利用率，使得总体上协议头占用的比例尽可能的小。如此看来这二者在避免发送小包上是一致的，在用户控制的层面上，Nagle算法完全不受用户socket的控制，你只能简单的设置TCP\_NODELAY而禁用它，CORK算法同样也是通过设置或者清除TCP\_CORK使能或者禁用之，然而Nagle算法关心的是网络拥塞问题，只要所有的ACK回来则发包，而CORK算法却可以关心内容，

在前后数据包发送间隔很短的前提下（很重要，否则内核会帮你将分散的包发出），即使你是分散发送多个小数据包，你也可以通过使能CORK算法将这些内容拼接在一个包内，如果此时用Nagle算法的话，则可能做不到这一点。

## Keepalive

1. Keepalive定时器用于检测空闲连接的另一端是否崩溃或重启。
2. 设置SO\_KEEPALIVE选项后，如果2小时内在此套接口的任一方向都没有数据交换，TCP就自动给对方发一个保持存活探测报文段，客户主机处于以下4种状态之一：(1)客户主机接收一切正常，服务器收到期望的ACK响应，并将keepalive定时器复位。(2)客户主机已崩溃，并且关闭或者正在重启。此时，服务器无法收到相应，在75s后超时。服务器总共发出10个这样的探查，每个间隔75秒。如果一个响应都没有收到，则终止连接。(3)客户主机已重启，此时服务器将收到一个复位响应，终止连接。(4)客户主机正常运行，但服务不可达，同(2)。
3. keepalive定时器默认2小时的间隔备受争议，通常应用上需要的时间要比2小时短的多。并且，当系统关闭一个由KEEPALIVE机制检查出来的死连接时，是不会主动通知上层应用的，只有在调用相应的IO操作在返回值中检查出来。因此，如果上层应用需要保活机制，最好还是自己实现。

## TCP的路径MTU探测

- 1)根据自身MTU及对方SYN中携带的MSS确定发送报文数据部分的最大容量（如果对方没有指定MSS，则默认为536）；
- 2)在IP头部打开DF标志位；
- 3)如果收到ICMP错误信息告知需要分片，如果ICMP信息中包含下一跳MTU的信息，那么根据这个值调整数据的最大容量，如果ICMP信息中不支持这种新协议(下一跳MTU值为0)，那么调整数据的最大容量至下一个可能的大小；
- 4)DF标志位会一直打开，以保证能够测量得到正确的Path MTU；
- 5)超时后会重新探询Path MTU以保证链路改变也能用到正确的Path MTU.

TCP Path MTU探询的好处是：

- 1)避免在通过MTU小于576的中间链路时进行分片；
- 2)防止中部链路的某些网络的MTU小于通信两端所在网络的MTU时进行分片；
- 3)充分利用链路的吞吐量.

## 长肥管道

带宽延时积很大的网络叫做长肥网络(LFN, long fat network, 单位为字节), 在LFN上建立的TCP链接叫做长肥管.

长肥管道带来的一些问题：

- 1) 长肥管的带宽延时积很大, TCP头部的窗口大小字段只能最多声明65535( $2^{16}$ )字节大小的窗口, 因此不能充分利用网络, 由此提出了窗口扩大选项以声明更大的窗口.
- 2) 由于长肥管的延时较高, 出现丢包的情况会使得管道枯竭(即网络通信速度急剧下降), 快重传快恢复算法就是用以削弱这一问题的影响, SACK选项也有使用.
- 3) 为了提高长肥管的吞吐量, 长肥管一般声明很大的窗口值, 而这样不利于RTT的测量(因为TCP只有一个RTT计时器, 启动RTT计时的数据在没有被ACK前, TCP无法进行下一次RTT的测量, 而由于发送延时一般大于传播延时, 所以TCP往往是发送完一个窗口的数据计算一次RTT), 所以需要引入时间戳选项提高测试RTT的频率.
- 4) 由于长肥管的发送速度非常快, 所以导致很短时间内数据的序号就会重复(在gigabit网络只需要34秒就会出现序号重复). 因此引入PAWS算法应对这种情况.

## 超时重传

1. 对每个连接, TCP管理4个定时器：

(1) 重传定时器：用于等待另一端的确认； (2) persist定时器：用于使窗口大小信息保持不断流动，即使另一端关闭了其接收窗口； (3) keepalive定时器：用于检测空闲连接的另一端是否崩溃或重启； (4) 2MSL定时器：用于测量一个处于TIME\_WAIT状态连接的时间

1. 超时与重传递时间间隔

超时时间可以应用程序设置(SO\_RCVTIMEO, SO\_SNDFTIMEO), 而重试的时间采用指数退避的方式，即每次重试的时间间隔为上次的2倍。在目前的实现中，首次分组传输与复位信号传输的时间间隔为9分钟。

1. 往返时间RTT的测量

平滑的RTT估计器： $R = \alpha R + (1 - \alpha)M$ , 其中 $\alpha=0.9$ , M是ACK测量到的RTT

重传超时时间的计算:

最初RTO= $R * \beta$ ,  $\beta=2$ , 但该方法在RTT变化很大时会引起不必要的重传

使用均值和方差来计算RTO：

$Err = M - A$   $A = A + gErr$   $D = D + h * (|Err| - D)$   $RTO = A + 4D$  其中, A和D分别被初始化为0和3, RTO初始化为 $A+2D=6$  只有数据报文段才会被计时, 不对单纯的ACK计时

1. 重传的多义性问题 (Karn算法)

当一个超时和重传发生时，在重传数据的确认最后到达之前不能更新RTT估计器，因为我们不知道ACK对应哪次传输。并且，由于数据重传，RTO已经得到一个指数退避，下次传输的时候使用这个退避后的RTO。对一个没有被重传的报文段而言，除非收到了一个确认，否则不计算新的RTO。

### 1. 拥塞避免算法

慢启动算法是在一个连接上发起数据流的方法，但有时分组回达到中间路由的极限。拥塞避免算法是一种处理丢失分组的方法。该算法假设分组由于损坏引起的丢失是非常少的，因此分组丢失就意味着源主机和目的主机的某处网络发生了拥塞。

有两种分组丢失的指示：发生超时、接收到重复的确认。

拥塞避免算法通常与慢启动算法同时实现，它们需要对每个连接维护两个变量：拥塞窗口 cwnd 和慢启动门限 ssthresh。这样，算法的过程如下：1) 对一个给定的连接，初始化 cwnd 为 1 个报文段，ssthresh 为 65535 个字节。2) TCP 输出例程的输出不能超过 cwnd 和接收方通告窗口的大小。拥塞避免是发送方使用的流量控制，而通告窗口则是接收方进行的流量控制。前者是发送方感受到的网络拥塞的估计，而后者则与接收方在该连接上的可用缓存大小有关。3) 当拥塞发生时（超时或收到重复确认），ssthresh 被设置为当前窗口大小的一半（cwnd 和接收方通告窗口大小的最小值，但最少为 2 个报文段）。此外，如果是超时引起了拥塞，则 cwnd 被设置为 1 个报文段（这就是慢启动）。4) 当新的数据被对方确认时，就增加 cwnd，但增加的方法依赖于我们是否正在进行慢启动或拥塞避免。如果 cwnd 小于或等于 ssthresh，则正在进行慢启动，否则正在进行拥塞避免。启动一直持续到我们回到当拥塞发生时所处位置的半时候才停止（因为我们记录了在步骤 2 中给我们制造麻烦的窗口大小的一半），然后转为执行拥塞避免。

慢启动算法初始设置 cwnd 为 1 个报文段，此后每收到一个确认就加 1。这会使窗口按指数方式增长：发送 1 个报文段，然后是 2 个，接着是 4 个……。

拥塞避免算法要求每次收到一个确认时将 cwnd 增加  $1/cwnd$ 。与慢启动的指数增加比起来，这是一种加性增长(additive increase)。我们希望在一个往返时间内最多为 cwnd 增加 1 个报文段（不管在这个 RTT 中收到了多少个 ACK），然而慢启动将根据这个往返时间中所收到的确认的个数增加 cwnd。

术语“慢启动”并不完全正确。它只是采用了比引起拥塞更慢些的分组传输速率，但在慢启动期间进入网络的分组数增加的速率仍然是在增加的。只有在达到 ssthresh 拥塞避免算法起作用时，这种增加的速率才会慢下来。

### 1. 快速重传算法

拥塞避免算法的修改建议 1990 年提出 [Jacobson 1990b]。在介绍修改之前，我们认识到在收到一个失序的报文段时，TCP 立即需要产生一个 ACK（一个重复的 ACK）。这个重复的 ACK 不应该被迟延。该重复的 ACK 的目的在于让对方知道收到一个失序的报文段，并告诉对方自己希望收到的序号。由于我们不知道一个重复的 ACK 是由一个丢失的报文段引起的，还

是由于仅仅出现了几个报文段的重新排序，因此我们等待少量重复的ACK到来。假如这只是些报文段的重新排序，则在重新排序的报文段被处理并产生一个新的ACK之前，只可能产生1~2个重复的ACK。如果一连串收到3个或3个以上的重复ACK，就非常可能是一个报文段丢失了。于是我们就重传丢失的数据报文段，而无需等待超时定时器溢出。这就是快速重传算法。接下来执行的不是慢启动算法而是拥塞避免算法。这就是快速恢复算法。

### 1. TCP连接对ICMP差错的处理

(1) ICMP源站抑制差错：将拥塞窗口 $cwnd$ 设为1个报文段大小发起慢启动，但是慢启动门限 $ssthresh$ 没有变化； (2) 主机不可达或网络不可达差错：忽略，因为这两个差错通常被认为是暂时的。

#### 1. 重新分组

当tcp重传的时候，不一定要重新传输相同的报文段。实际上，TCP允许进行重新分组而发送一个较大的报文段，这有助于提升性能。

## 滑动窗口

1. 滑动窗口协议允许发送方在停止并等待确认前可以连续发送多个分组，由于发送方不必每发一个分组就停下来等待确认，因此该方法可以加速数据的传输。

#### 2. 滑动窗口

窗口大小表示接收端的TCP协议缓存中还有多少剩余空间，用于接收端的流量控制

特点：

(1) 发送方不必发送一个全窗口的大小 (2) 来自接收方的一个报文段确认数据并把窗口向右滑动（窗口大小是相对于确认序号的） (3) 窗口的大小可以减小，但窗口的右边不能向左移动 (4) 接收方在发送一个ACK前不必等待窗口被填满

窗口更新：一个ACK分组，但不确认任何数据（分组中的序号已被前面的ACK确认），只是通知对方窗口大小已变化

#### 1. PUSH标志

发送方使用该标志通知接收方将所收到的数据全部交给接收进程，这标志着发送方暂时没有更多的数据要发送了

#### 1. 慢启动

慢启动为TCP增加一个拥塞窗口（ $cwnd$ ），刚建立连接时 $cwnd$ 初始化为一个报文段的大小（由另一端通告），其后每收到一个ACK， $cwnd$ 就增加一个报文段大小。发送方取拥塞窗口与通告窗口中的最小值作为发送上限。

拥塞窗口是发送方使用的流量控制，而通告窗口是接收方使用的流量控制。

### 1. URG标志

紧急标志用于发送端通知接收端分组中包含了紧急数据，具体如何处理由接收方确定

紧急数据也被成为带外数据

### 1. 带宽时延积

带宽：单位时间内从发送端到接收端所能通过的“最高数据率” RTT：从发送端到接收端的一去一回需要的时间 带宽时延乘积：等于带宽\*RTT，实际上就是发送端到接收端单向通道的数据容积的两倍

设带宽为B，RTT为Tr，滑动窗口为W，则：

(1)  $WB \cdot Tr$  时，则影响速率的因素是带宽

## 参考文档

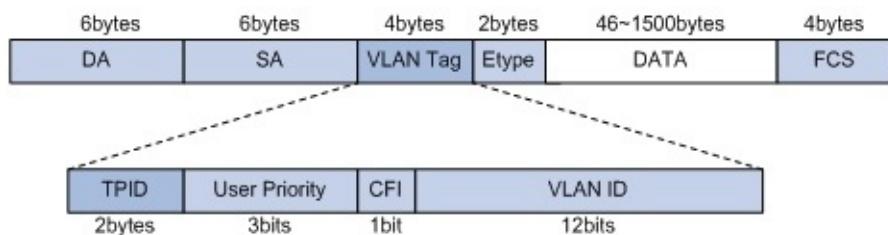
- [tcpip详解笔记](#)

# VLAN

LAN 表示 Local Area Network，本地局域网，通常使用 Hub 和 Switch 来连接 LAN 中的计算机。一个 LAN 表示一个广播域，它的意思是 LAN 中的所有成员都会收到 LAN 中一个成员发出的广播包。因此，LAN 的边界在路由器或者类似的三层设备。

VLAN 表示 Virtual LAN。一个带有 VLAN 功能的 Switch 能够同时处于多个 LAN 中。简单的说，VLAN 是一种将一个交换机分成多个交换机的一种方法。

IEEE 802.1Q 标准定义了 VLAN Header 的格式。它在普通以太网帧结构 SA (src address) 之后加入了 4bytes 的 VLAN Tag/Header 数据，其中包括 12bits 的 VLAN ID。VLAN ID 的最大值是 4096，但是有效值范围是 1- 4094。



## 交换机端口类型

以太网端口有三种链路类型：

- Access：只能属于一个 VLAN，一般用于连接计算机的端口
- Trunk：可以属于多个 VLAN，可以接收和发送多个 VLAN 的报文，一般用于交换机之间连接的接口
- Hybrid：属于多个 VLAN，可以接收和发送多个 VLAN 报文，既可以用于交换机之间的连接，也可以用户连接用户的计算机。Hybrid 端口和 Trunk 端口的不同之处在于 Hybrid 端口可以允许多个 VLAN 的报文发送时不打标签，而 Trunk 端口只允许缺省 VLAN 的报文发送时不打标签。

## VLAN 的不足

- VLAN 使用 12-bit 的 VLAN ID，因此第一个不足之处就是最多只支持 4096 个 VLAN 网络
- VLAN 是基于 L2 的，因此很难跨越 L2 的边界，限制了网络的灵活性
- VLAN 的配置需手动介入较多

## **QinQ**

QinQ是为了扩大VLAN ID的数量而提出的技术（IEEE 802.1ad），外层tag称为Service Tag，而内层tag则称为Customer Tag。

参考文档

- <http://www.cnblogs.com/sammyliu/p/4626419.html>
- <https://docs.ustack.com/unp/src/architecture/vlan.html>

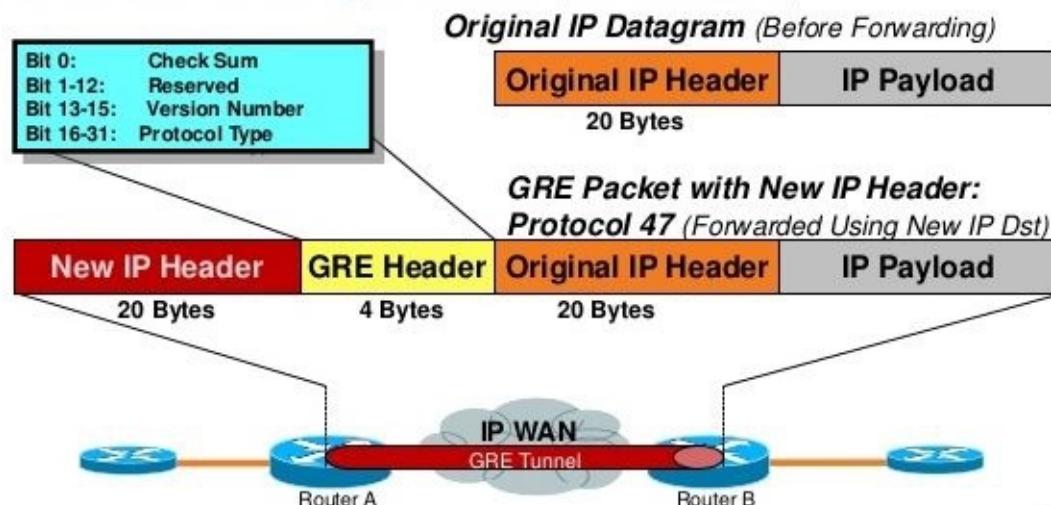
# Overlay

技术名称	支持者	支持方式	网络虚拟化方式	数据新增报文长度	链路HAS能力
VXLAN	Cisco/VMWARE/Citrix/Red Hat/Broadcom	L2 over UDP	VXLAN 报头 24 bit VNI	50Byte(+原数据)	现网可行L~L HAS
NVGRE	HP/Microsoft/Broadcom/Dell/Intel	L2 over GRE	NVGRE 报头 24 bit VSI	42Byte(+原数据)	GRE头需网线升级
STT	VMWare	无状态TCP，即L2在类似TCP的传输层	STT报头 64 bit Context ID	58 ~ 76Byte(+原数据)	现网可行L~L HAS

## Generic Routing Encapsulation (GRE)

GRE提供了IP in IP的封装技术:

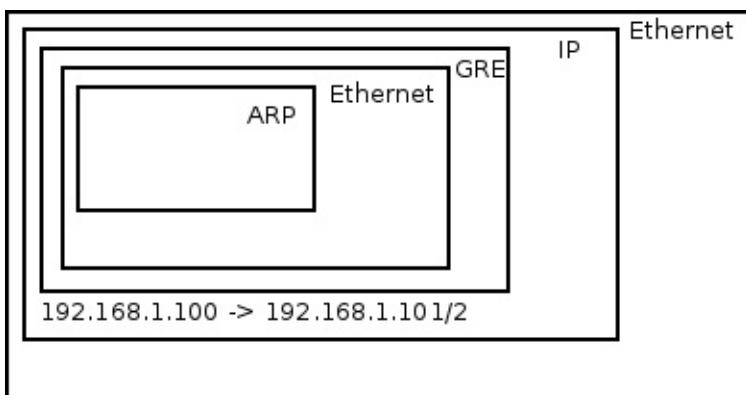
### GRE Tunnel Encapsulation (RFC 2784)



步骤	操作/封装	协议	长度	备注
1	ping -s 1448	ICMP	$1456 = 1448 + 8$ (ICMP header)	ICMP MSS
2	L3	IP	$1476 = 1456 + 20$ (IP header)	GRE Tunnel MTU
3	L2	Ethernet	$1490 = 1476 + 14$ (Ethernet header)	经过 bridge 到达 GRE
4	GRE	IP	$1500 = 1476 + 4$ (GRE header) + 20 (IP header)	物理网卡 (IP) MTU
5	L2	Ethernet	$1514 = 1500 + 14$ (Ethernet header)	最大可传输帧大小

因此，GRE 的 overhead 是  $1514 - 1490 = 24$  byte。

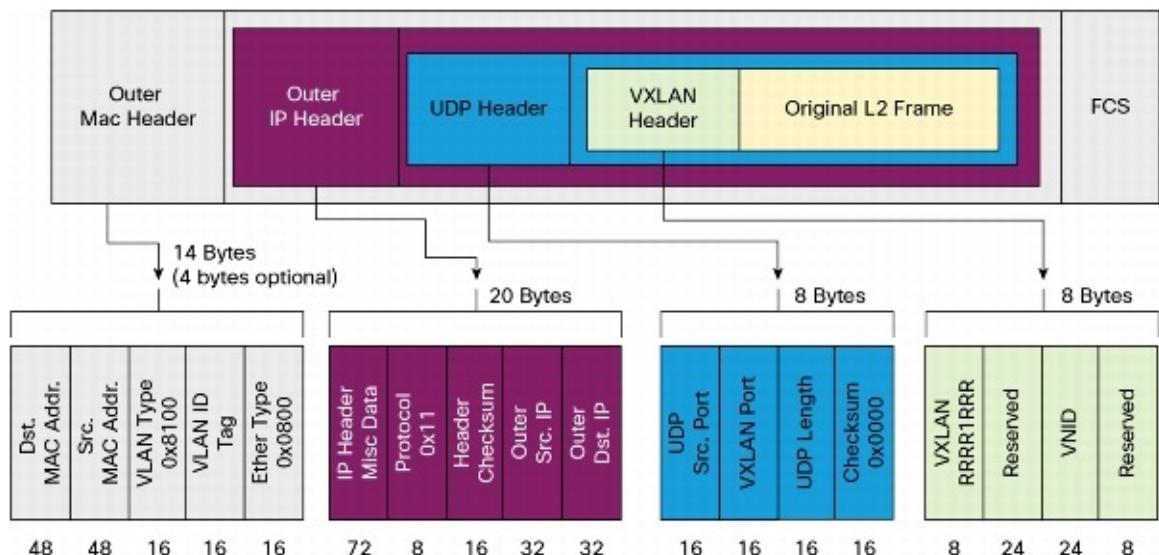
可见，使用 GRE 可以比使用 VxLAN 每次可以多传输  $1448 - 1422 = 26$  byte 的数据。



由于GRE没有提供加密和防止窃听的技术，故而经常跟IPSEC一起配合实现对数据的加密传输。

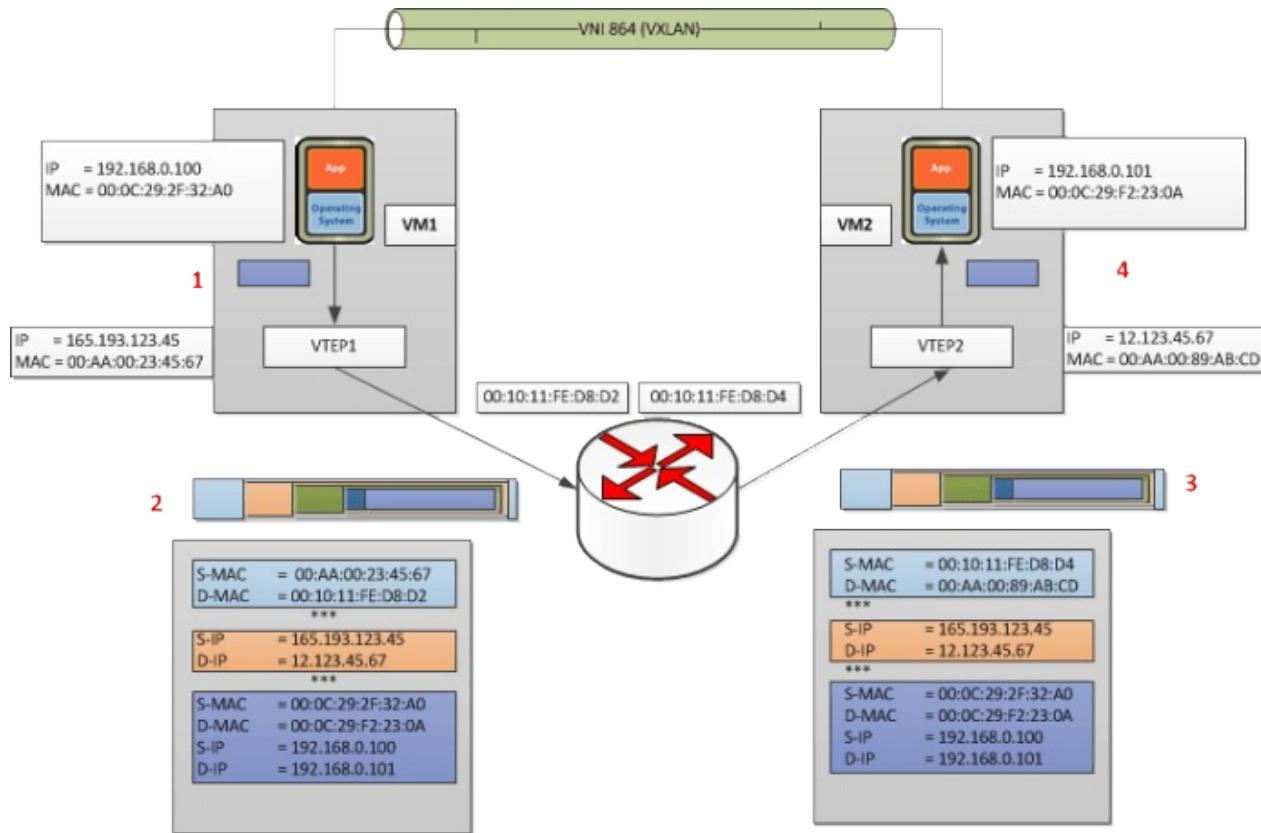
## VXLAN

Virtual eXtensible Local Area Network (VXLAN) 是一种将2层报文封装到UDP包(Mac in UDP)中进行传输的一种封装协议。VXLAN主要是由Cisco推出的，VXLAN的包头有一个24bit的ID段，即意味着1600万个独一无二的虚拟网段，这个ID通常是对UDP端口采取伪随机算法而生成的（UDP端口是由该帧中的原始MAC Hash生成的）。这样做的好处是可以保证基于5元组的负载均衡，保存VM之间数据包的顺序，具体做法是将数据包内部的MAC组映射到唯一的UDP端口组。将二层广播被转换成IP组播，VXLAN使用IP组播在虚拟网段中泛洪而且依赖于动态MAC学习。在VXLAN中，封装和解封的组件有个专有的名字叫做VTEP，VTEP之间通过组播发现对方。



步骤	操作/封包	协议	长度	MTU
1	ping -s 1422	ICMP	$1430 = 1422 + 8$ (ICMP header)	
2	L3	IP	$1450 = 1430 + 20$ (IP header)	VxLAN Interface 的 MTU
3	L2	Ethernet	$1464 = 1450 + 14$ (Ethernet header)	
4	VxLAN	UDP	$1480 = 1464 + 8$ (VxLAN header) + 8 (UDP header)	
5	L3	IP	$1500 = 1480 + 20$ (IP header)	物理网卡的 (IP) MTU，它不包括 Ethernet header 的长度
6	L2	Ethernet	$1514 = 1500 + 14$ (Ethernet header)	最大可传输帧大小

因此，VxLAN 的 overhead 是  $1514 - 1464 = 50$  byte。



基于组播的 VXLAN 网络其实是没有控制平面的，依赖于数据平面的 flood-and-learn，如果交换机不支持组播的话，将会退化到广播，目前这类的应用已经很少了。为了解决组播的依赖，一种方法是通过 HER 的方法复制报文成单播，这样组播报文或者广播报文可以通过单播复制的形式发送，这种方式被称为 Head-End Replication。Open vSwitch Driver 实现的 VXLAN 即使用类似这种方式避免组播的依赖。HER 在即使有控制平面的情况下依然具备价值，因为有可能有静默主机、MAC 表项老化、虚拟机需要使用组播或广播达成业务的需求。

## VXLAN Offload

一些新型号的网卡(Intel X540 or X710)，具备VXLAN硬件封包／解包能力。开启硬件VXLAN offload，并使用较大的MTU（如9000），可以明显提升虚拟网络的性能。

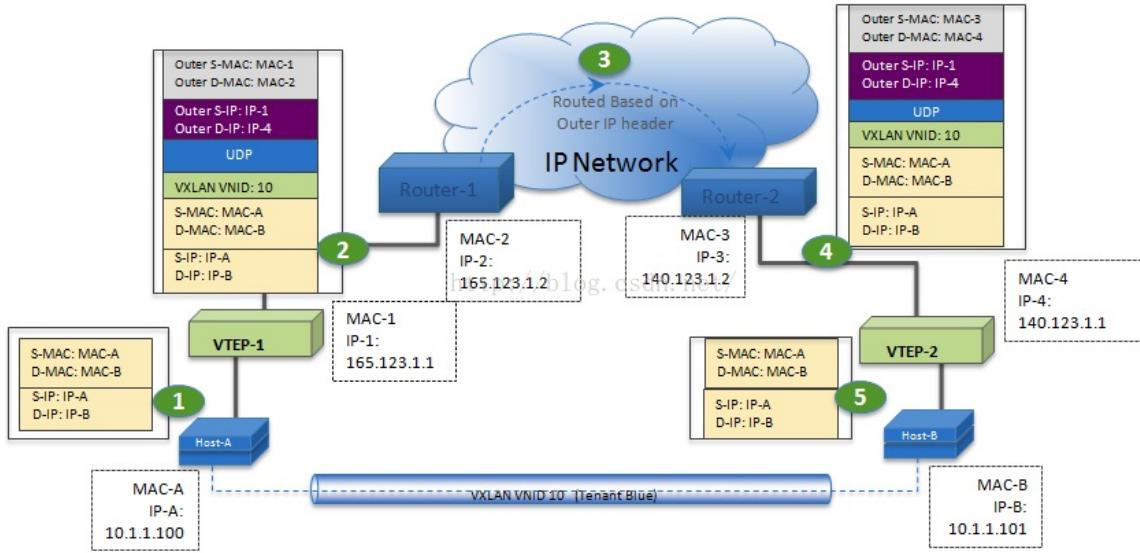
### 开启或关闭vxlan offload的方法

```
ethtool -k <eth0/eth1> tx-udp_tnl-segmentation <on/off>
```

## VXLAN转发过程

### 同VXLAN ID内转发

VXLAN最早依靠组播泛洪的方式来转发，但这会导致产生大量的组播流量。所以，在实际生产中，通常使用SDN控制器结合南向协议来避免组播问题。



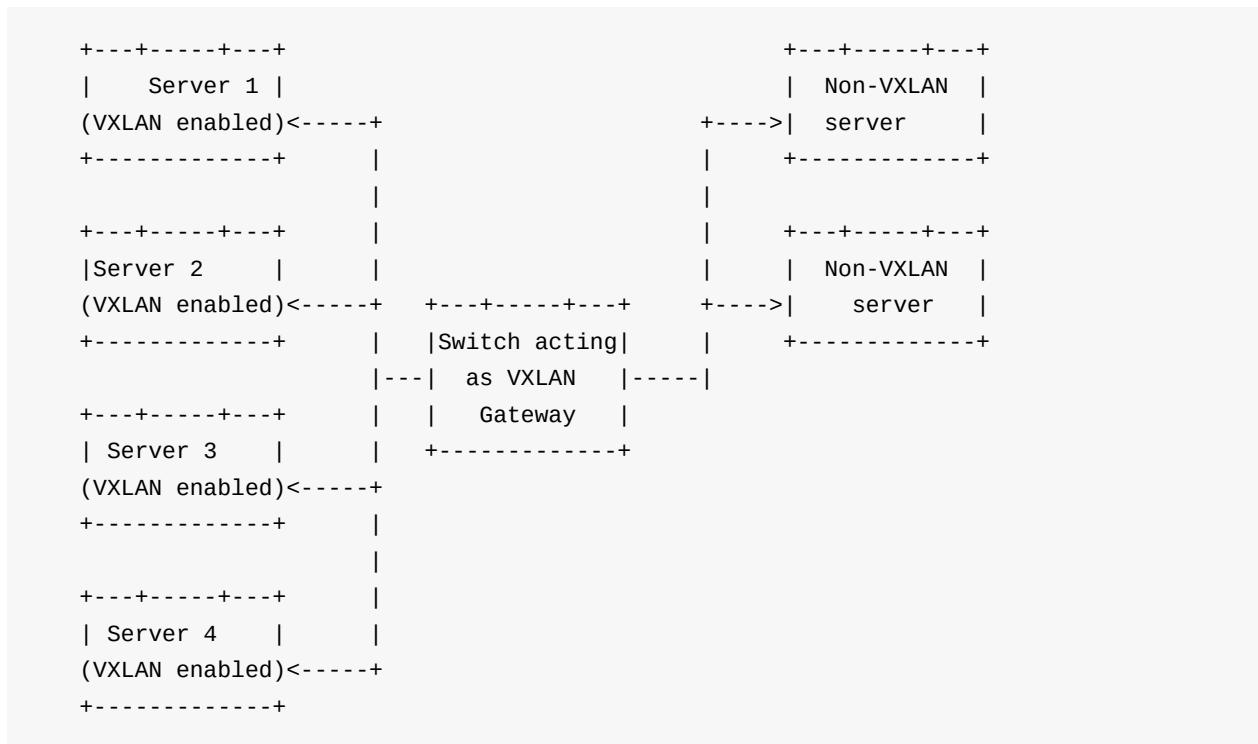
(图片来自[csdn](#))

## 不同VXLAN ID转发

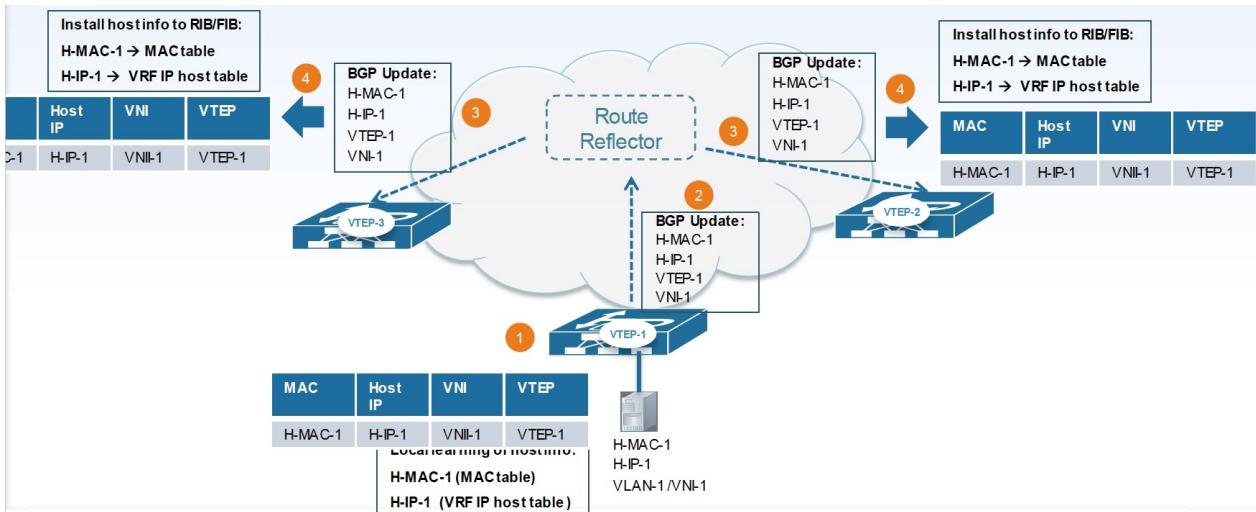
大致转发过程与上面类似，所不同的是需要报文在所属vtep或者vxlan gateway处来转换vxlan id（源和目的处都需要做这个转换）。

## VXLAN与非VXLAN（如VLAN）转发

需要VXLAN Gateway来转换vxlan vni和vlan id：

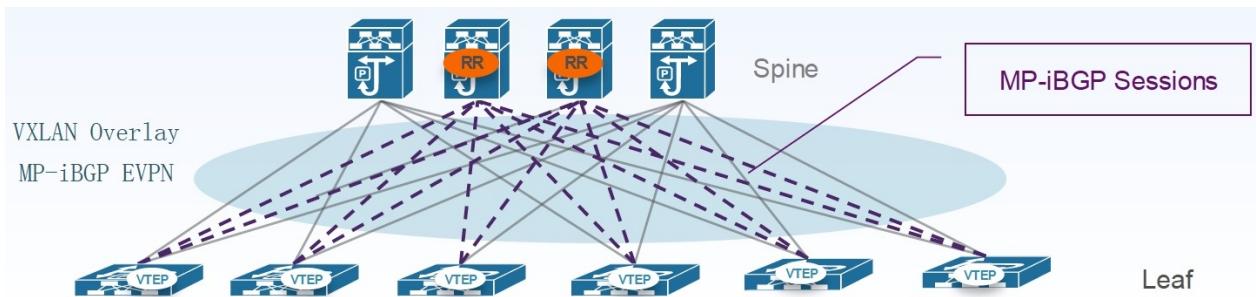


## MPBGP EVPN VXLAN



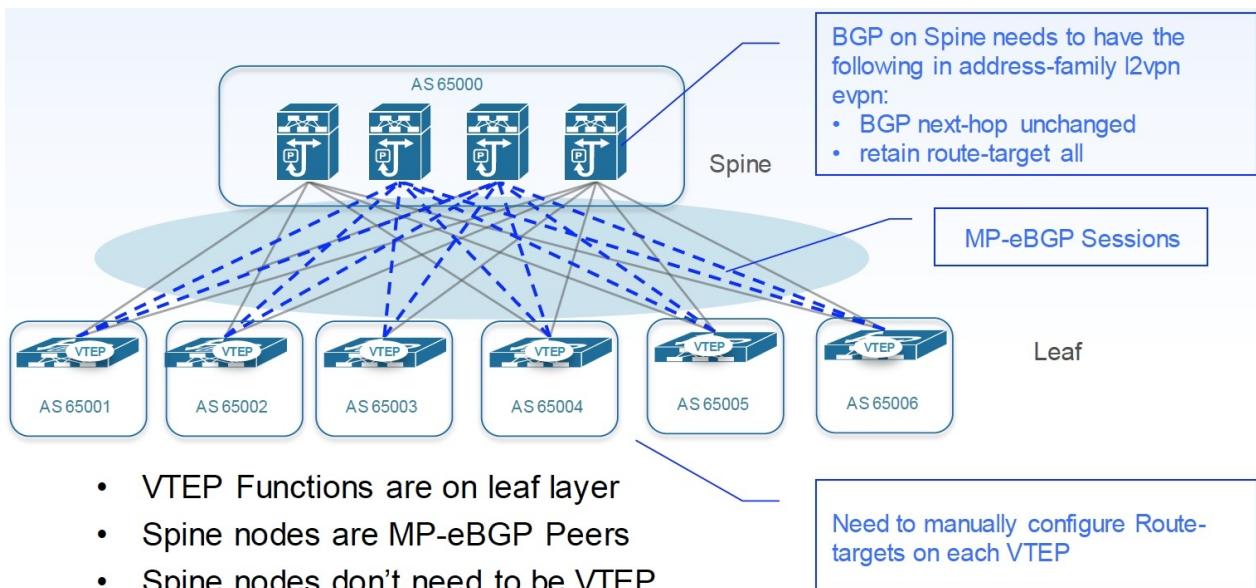
每一个 VTEP 将作为一个 BGP Speaker，向其他 VTEP 通过 EVPN 发送本地的 MAC、IP 信息，BGP RR 可以避免 BGP 的 Full-Mesh，提高通信效率。得益于控制平面，每个 VTEP 将可作为分布式网关、可以抑制 ARP 广播、可以将广播或组播通过单播复制来提升效率、可以对 VTEP 进行认证。

具体到 BGP 租网上，有几种选择，包括 iBGP、eBGP 的选择和外部网络通信。



这种模型下 VTEP 只在 Leaf 上，Spine 中选取两个作为 iBGP RR，Spine 不需要作为 VTEP。此外 RR 也可以有多种放置方法，例如在 Leaf 上，这样 Spine 不需要运行 MPBGP EVPN，或者在额外的专门网络设备。

如果是 eBGP，典型的部署方法如下图，好处是 Spine 作为 eBGP Peer，而不是 iBGP RR，Spine，Spine上的 BGP 需要有对 address-family l2vpn evpn 的转发能力，但不需要支持 VXLAN。所有 Leaf 可以设置各自的 AS，也可以设置为同一 AS，eBGP 运维难度较高，参考设计见 draft-ietf-rtgwg-bgp-routing-large-dc，目前一般较少采用。

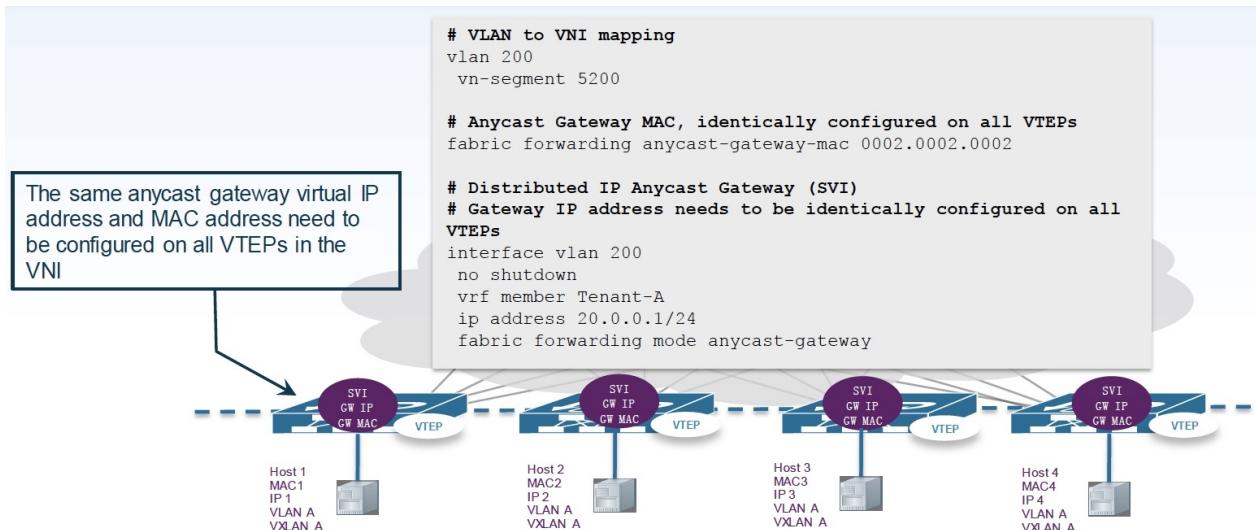


## Distributed Anycast Gateway

IETF 在 draft-ietf-bess-evpn-inter-subnet-forwarding 中对在 EVPN 中属于不同的 VxLan 下如何通过 Integrated Routing and Bridging (以下简称 IRB) 处理跨子网通信做了说明，换句话说，EVPN VxLan 提供了原生的基于 IRB 的分布式三层网关参考。

然而 EVPN VxLan 的实际路由过程可以分成两步来谈，第一部分是虚拟机的 First-hop 地址，即网关地址，第二部分是如何在不同 VxLan 间路由 (IRB)，本节会先谈网关地址的问题。

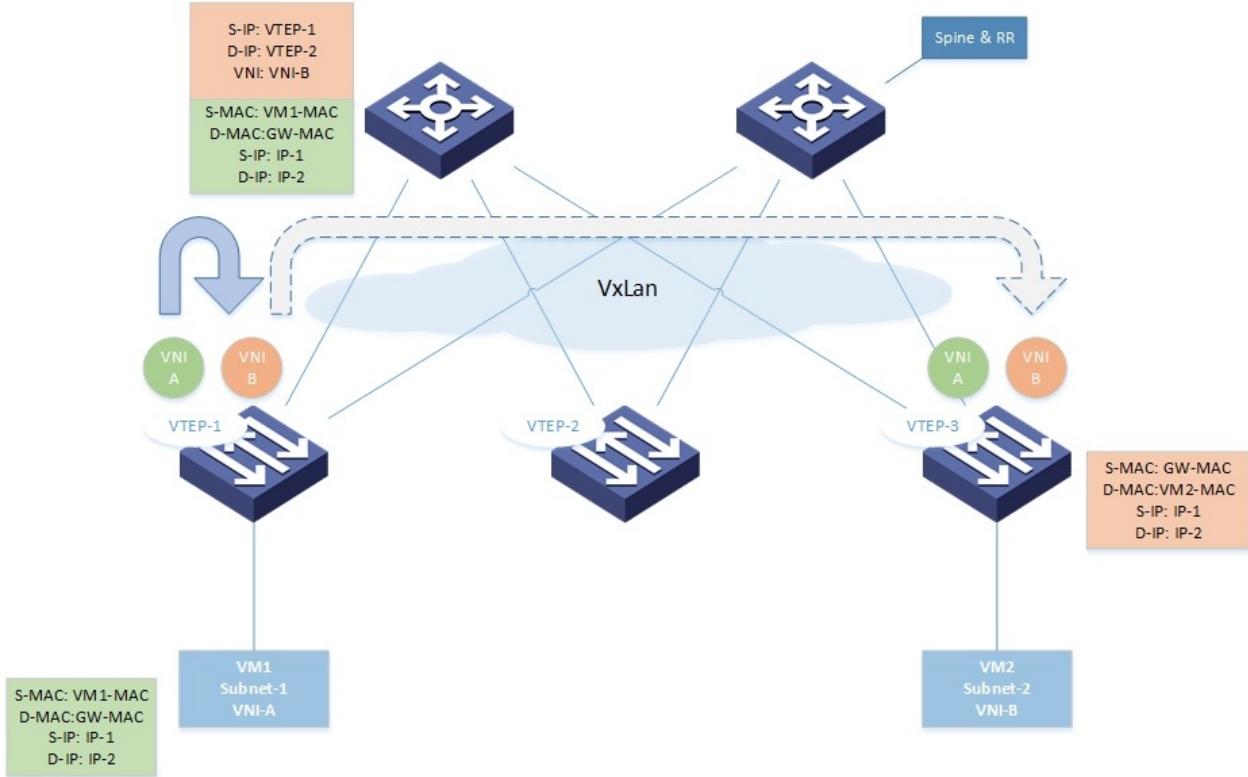
目前一种实践是使用 Anycast Gateway 技术，每个 VTEP 上均配置相同的 vIP 和 vMAC，如图：



这样首先每个虚拟机的网关都在最近的 VTEP 上，可以优化网络路径，其次当虚拟机发生迁移时，不会需要重新获取默认网关的 ARP。在一些厂商中，这项技术被称为 Static Anycast Gateway。

## Integrated Routing and Bridging

IRB 即 VTEP 提供三层和二层功能，但是对于具体如何路由，目前存在两种方法，分别为 Asymmetric IRB mode 和 Symmetric IRB mode。前者是非对称模式，后者是对称模式，对于 Asymmetric，结合 Anycast Gateway 后路径是这样的：



报文由虚拟机发出时，目的 MAC 是网关的虚拟 MAC，VTEP-1 收到报文后查询路由找到 IP-2 对应的虚拟机，查询到对应的 VTEP 为 VTEP-2 后，封上 VxLan 的头部发到 VTEP-2，并将 VNI 设置为对方的 VNI-B，VTEP-2 收到报文后，将 VxLan 头部剥掉换成 Vlan 并发往 VM-2。

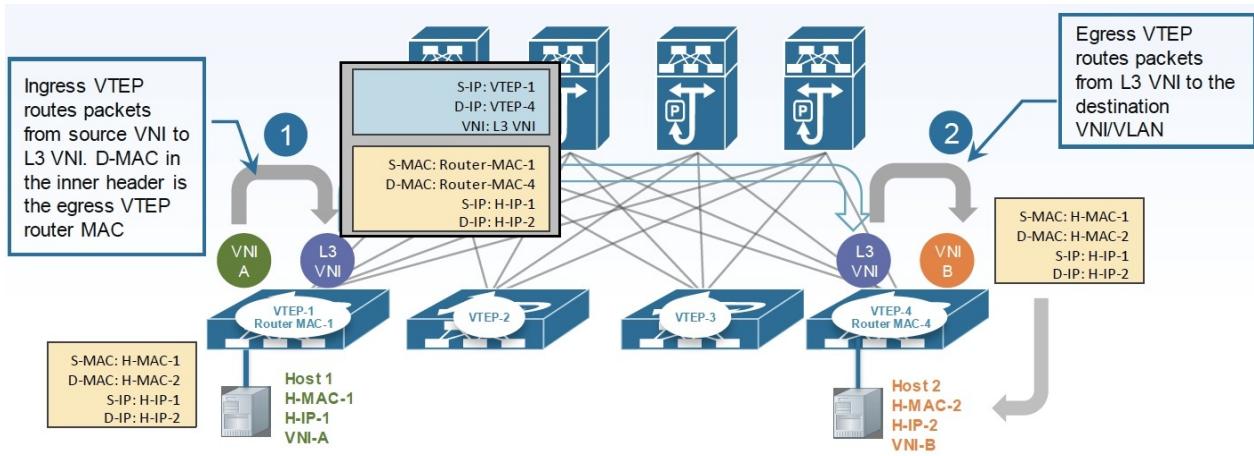
当虚拟机需要回复时，路径完全反过来，即在 VTEP-2 上完成 VXLAN 封包和设置 VNI 为 VNI-A。所以这个过程是非对称的。

这种实现存在一些显而易见的问题：

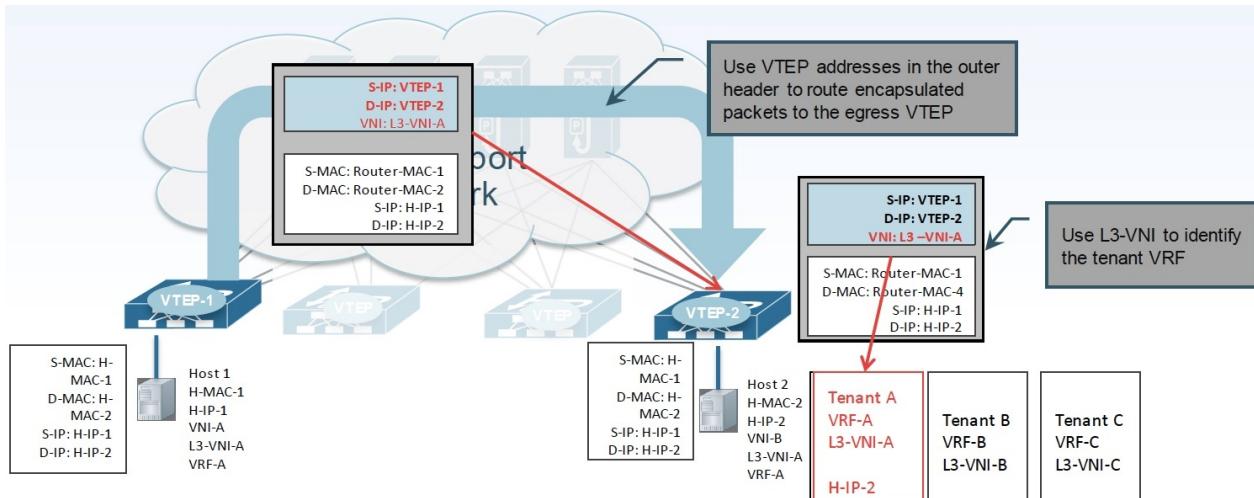
- 所有的 VTEP 必须配置上所有的 VXALN VNI，否则不同 VNI 通信会存在问题；
- 所有的 VTEP 必须获整个 Fabric 完整的 Host tables 信息，否则无法完成完路由。

另一种实现方法是 Symmetric IRB，其实现与 Asymmetric IRB 最显著的不同是源 VTEP 和目标 VTEP 都会承担三层和二层功能，而不像 Symmetric IRB 只在源 VTEP 做路由。这样最终实现是对称的，但前提是必须引入一个新的概念即 L3 VNI。

在 Symmetric IRB 中，每个租户的 VRF 会分配一个 L3 VNI，可达信息（NLDR）会在同一个 L3 VNI 下同步，这样每次路由需要将外层 VXLAN 目的地地址设置为目的 VTEP 的地址，将 VNI 设置为 L3 VNI。



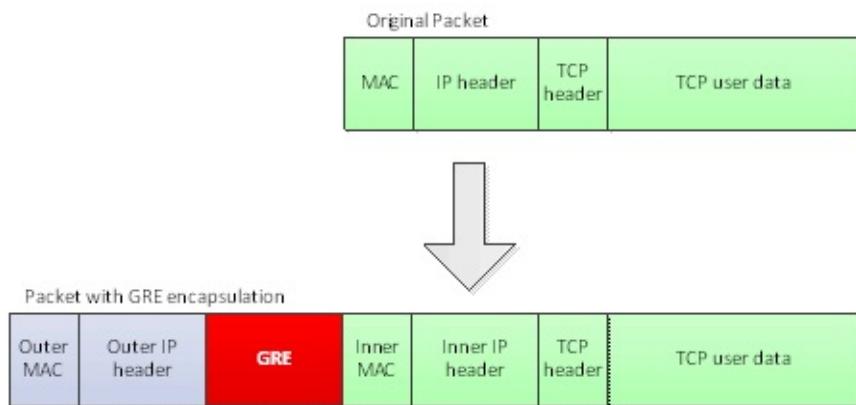
从上面的分析可以得知，Symmetric IRB 最大的好处一是不需要所有 VTEP 均配置所有 VNI，二是不需要所有的 VTEP 知道整个 Fabric 的完整 Host tables 信息。但是，这是建立在不是最差情况的前提，如果说恰好每个租户都在每个 VTEP 下具有虚拟机，或着整个网络只有一个租户，那么网络可能产生退化。Symmetric IRB 的主要优化场景是针对多租户的。



## NVGRE

NVGRE主要支持者是Microsoft。与VXLAN不同的是，NVGRE没有采用标准传输协议（TCP/UDP），而是借助通用路由封装协议（GRE）。NVGRE使用GRE头部的低24位作为租户网络标识符（TNI），与VXLAN一样可以支持1600个虚拟网络。为了提供描述带宽利用率粒度的流，传输网络需要使用GRE头，但是这导致NVGRE不能兼容传统负载均衡，这是NVGRE与VXLAN相比最大的区别也是最大的不足。为了提高负载均衡能力建议每个NVGRE主机使用多个IP地址，确保更多流量能够被负载均衡。

NVGRE不需要依赖泛洪和IP组播进行学习，而是以一种更灵活的方式进行广播，但是这需要依赖硬件/供应商。最后一个区别关于分片，NVGRE支持减小数据包最大传输单元以减小内部虚拟网络数据包大小，不要求传输网络支持传输大型帧。



## STT

STT（Stateless Transport Tunneling Protocol）是Nicira提交的隧道协议，类似于VXLAN和VGGRE，它也是把二层的帧封装在一个ip报文的payload中，并在前面增加了tcp头和STT头。注意，STT的tcp头是精心构造出来的，以便利用TSO、LRO、GRO等网卡特性。

## Geneve

Geneve（Generic Network Virtualization Encapsulation）旨在统一VXLAN、NVGRE等各种方案，提供更灵活且适用各种虚拟化场景的通用封装协议。

Geneve使用UDP封包，端口号为6081。

## 参考文档

- <https://docs.ustack.com/unp/src/architecture/vxlan.html>
- <http://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/guide-c07-734107.html>

# Linux网络

Linux网络知识简介。

# Linux网络配置

Linux网络配置方法简介。

## 配置IP地址

```
# 使用ifconfig
ifconfig eth0 192.168.1.3 netmask 255.255.255.0

# 使用ip命令增加一个IP
ip addr add 192.168.1.4/24 dev eth0

# 使用ifconfig增加网卡别名
ifconfig eth0:0 192.168.1.10
```

这样配置的IP地址重启机器后会丢失，所以一般应该把网络配置写入文件中。如Ubuntu可以将网卡配置写入 `/etc/network/interfaces` (Redhat和CentOS则需要写入 `/etc/sysconfig/network-scripts/ifcfg-eth0` 中)：

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
    address 192.168.1.3
    netmask 255.255.255.0
    gateway 192.168.1.1

auto eth1
iface eth1 inet dhcp
```

## 配置默认路由

```
# 使用route命令
route add default gw 192.168.1.1
# 也可以使用ip命令
ip route add default via 192.168.1.1
```

## 配置VLAN

```
# 安装并加载内核模块  
apt-get install vlan  
modprobe 8021q  
  
# 添加vlan  
vconfig add eth0 100  
ifconfig eth0.100 192.168.100.2 netmask 255.255.255.0  
  
# 删除vlan  
vconfig rem eth0.100
```

## 配置硬件选项

```
# 改变speed  
ethtool -s eth0 speed 1000 duplex full  
  
# 关闭GRO  
ethtool -K eth0 gro off  
  
# 开启网卡多队列  
ethtool -L eth0 combined 4  
  
# 开启vxlan offload  
ethtool -K ens2f0 rx-checksum on  
ethtool -K ens2f0 tx-udp_tnl-segmentation on  
  
# 查询网卡统计  
ethtool -S eth0
```

# 虚拟网络设备

Linux提供了许多虚拟设备，这些虚拟设备有助于构建复杂的网络拓扑，满足各种网络需求。

## 网桥（bridge）

网桥是一个二层设备，工作在链路层，主要是根据MAC学习来转发数据到不同的port。

```
# 创建网桥  
brctl addbr br0  
# 添加设备到网桥  
brctl addif br0 eth1  
# 查询网桥mac表  
brctl showmacs br0
```

## veth

veth pair是一对虚拟网络设备，一端发送的数据会由另外一端接受，常用于不同的网络命名空间。

```
# 创建veth pair  
ip link add veth0 type veth peer name veth1  
  
# 将veth1放入另一个netns  
ip link set veth1 netns newns
```

## TAP/TUN

TAP/TUN设备是一种让用户态程序向内核协议栈注入数据的设备，TAP等同于一个以太网设备，工作在二层；而TUN则是一个虚拟点对点设备，工作在三层。

```
ip tuntap add tap0 mode tap  
ip tuntap add tun0 mode tun
```

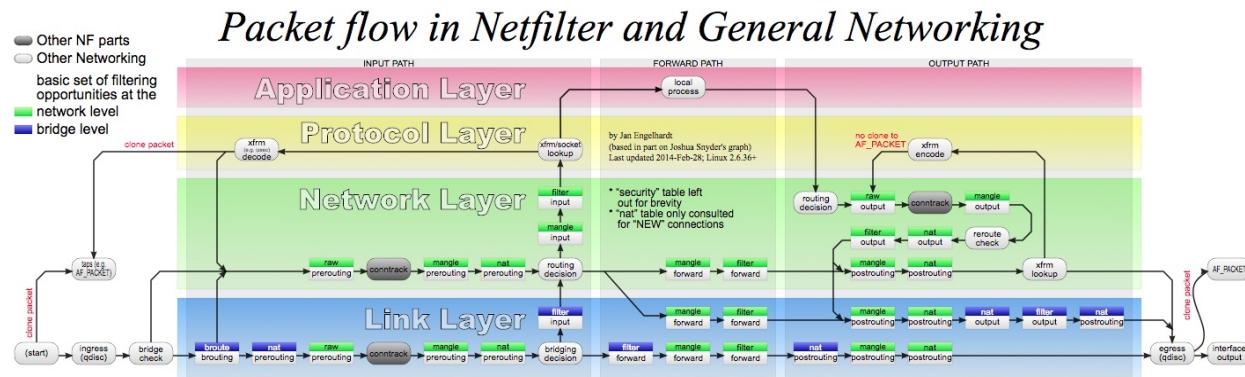
# iptables/netfilter

iptables是一个配置Linux内核防火墙的命令行工具，它基于内核的netfilter机制。新版本的内核（3.13+）也提供了nftables，用于取代iptables。

## netfilter

netfilter是Linux内核的包过滤框架，它提供了一系列的钩子（Hook）供其他模块控制包的流动。这些钩子包括

- `NF_IP_PRE_ROUTING`：刚刚通过数据链路层解包进入网络层的数据包通过此钩子，它在路由之前处理
- `NF_IP_LOCAL_IN`：经过路由查找后，送往本机（目的地址在本地）的包会通过此钩子
- `NF_IP_FORWARD`：不是本地产生的并且目的地不是本地的包（即转发的包）会通过此钩子
- `NF_IP_LOCAL_OUT`：所有本地生成的发往其他机器的包会通过该钩子
- `NF_IP_POST_ROUTING`：在包就要离开本机之前会通过该钩子，它在路由之后处理



## iptables

iptables通过表和链来组织数据包的过滤规则，每条规则都包括匹配和动作两部分。默认情况下，每张表包括一些默认链，用户也可以添加自定义的链，这些链都是顺序排列的。这些表和链包括：

- raw表用于决定数据包是否被状态跟踪机制处理，内建PREROUTING和OUTPUT两个链
- filter表用于过滤，内建INPUT（目的地是本地的包）、FORWARD（不是本地产生的并且目的地不是本地）和OUTPUT（本地生成的包）等三个链
- nat用表于网络地址转换，内建PREROUTING（在包刚刚到达防火墙时改变它的目的地）、INPUT、OUTPUT和POSTROUTING（要离开防火墙之前改变其源地址）等链
- mangle用于对报文进行修改，内建PREROUTING、INPUT、FORWARD、OUTPUT和

## POSTROUTING等链

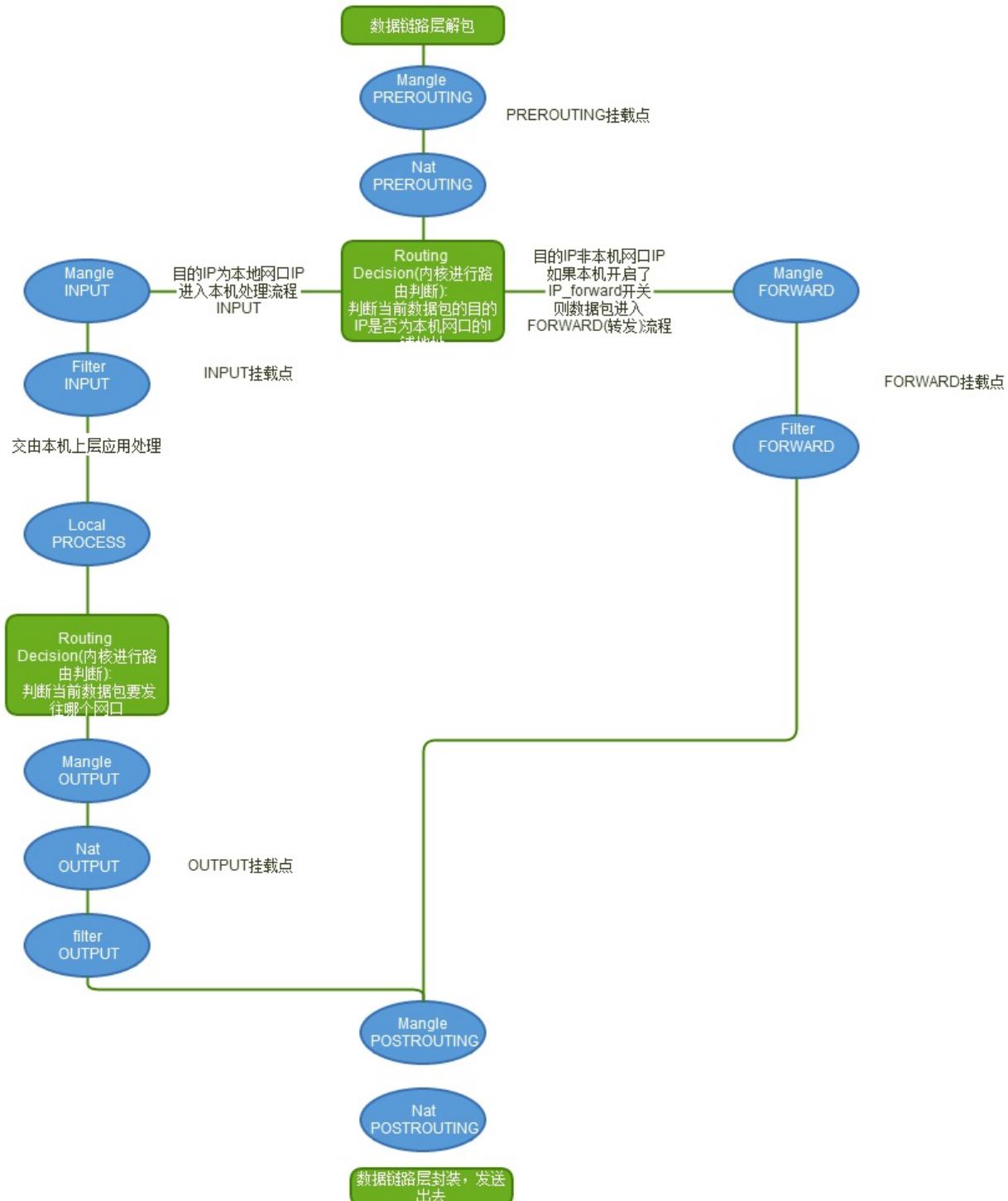
- security表用于根据安全策略处理数据包，内建INPUT、FORWARD和OUTPUT链

Tables↓/Chains→	PREROUTING	INPUT	FORWARD	OUTPUT	POSTROUTING
(routing decision)				✓	
<b>raw</b>	✓			✓	
(connection tracking enabled)	✓			✓	
<b>mangle</b>	✓	✓	✓	✓	✓
<b>nat (DNAT)</b>	✓			✓	
(routing decision)	✓			✓	
<b>filter</b>		✓	✓	✓	
<b>security</b>		✓	✓	✓	
<b>nat (SNAT)</b>		✓			✓

所有链默认都是没有任何规则的，用户可以按需要添加规则。每条规则都包括匹配和动作两部分：

- 匹配可以有多条，比如匹配端口、IP、数据包类型等。匹配还可以包括模块（如conntrack、recent等），实现更复杂的过滤。
- 动作只能有一个，通过 `-j` 指定，如ACCEPT、DROP、RETURN、SNAT、DNAT等

这样，网络数据包通过iptables的过程为



其规律为

1. 当一个数据包进入网卡时，数据包首先进入PREROUTING链，在PREROUTING链中我们有机会修改数据包的DestIP(目的IP)，然后内核的“路由模块”根据“数据包目的IP”以及“内核中的路由表”判断是否需要转送出去(注意，这个时候数据包的DestIP有可能已经被我们修改过了)
2. 如果数据包就是进入本机的(即数据包的目的IP是本机的网口IP)，数据包就会沿着图向下移动，到达INPUT链。数据包到达INPUT链后，任何进程都会收到它
3. 本机上运行的程序也可以发送数据包，这些数据包经过OUTPUT链，然后到达

POSTROUTING链输出(注意，这个时候数据包的SrcIP有可能已经被我们修改过了)

4. 如果数据包是要转发出去的(即目的IP地址不再当前子网中)，且内核允许转发，数据包就会向右移动，经过FORWARD链，然后到达POSTROUTING链输出(选择对应子网的网口发送出去)

## iptables示例

查看规则列表

```
iptables -nvL
```

允许22端口

```
iptables -A INPUT -p tcp --dport 22 -j ACCEPT
```

允许来自192.168.0.4的包

```
iptables -A INPUT -s 192.168.0.4 -j ACCEPT
```

允许现有连接或与现有连接关联的包

```
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
```

禁止ping包

```
iptables -A INPUT -p icmp --icmp-type echo-request -j DROP
```

禁止所有其他包

```
iptables -P INPUT DROP
iptables -P FORWARD DROP
```

## MASQUERADE

```
iptables -t nat -I POSTROUTING -s 10.0.0.30/32 -j MASQUERADE
```

## NAT

```
iptables -I FORWARD -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables -I INPUT    -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables -t nat -I OUTPUT -d 55.55.55.55/32 -j DNAT --to-destination 10.0.0.30
iptables -t nat -I PREROUTING -d 55.55.55.55/32 -j DNAT --to-destination 10.0.0.30
iptables -t nat -I POSTROUTING -s 10.0.0.30/32 -j SNAT --to-source 55.55.55.55
```

### 端口映射

```
iptables -t nat -I OUTPUT -d 55.55.55.55/32 -p tcp -m tcp --dport 80 -j DNAT --to-destination 10.10.10.3:80
iptables -t nat -I POSTROUTING -m conntrack ! --ctstate DNAT -j ACCEPT
iptables -t nat -I PREROUTING -d 55.55.55.55/32 -p tcp -m tcp --dport 80 -j DNAT --to-destination 10.10.10.3:80
```

### 重置所有规则

```
iptables -F
iptables -t nat -F
iptables -t mangle -F
iptables -X
```

## nftables

**nftables** 是从内核 3.13 版本引入的新的数据包过滤框架，旨在替代现用的 **iptables** 框架。  
**nftables** 引入了一个新的命令行工具 `nft`，取代了之前的 **iptables**、**ip6tables**、**ebtables** 等各种工具。

跟 **iptables** 相比，**nftables** 带来了一系列的好处

- 更易用易理解的语法
- 表和链是完全可配置的
- 匹配和目标之间不再有区别
- 在一个规则中可以定义多个动作
- 每个链和规则都没有内建的计数器
- 更好的动态规则集更新支持
- 简化 IPv4/IPv6 双栈管理
- 支持 `set/map` 等
- 支持级连（需要内核 4.1+）

跟 **iptables** 类似，**nftables** 也是使用表和链来管理规则。其中，表包括 `ip`、`arp`、`ip6`、`bridge`、`inet` 和 `netdev` 等 6 个类型。下面是一些简单的例子。

```
# 新建一个ip类型的表
nft add table ip foo

# 列出所有表
nft list tables

# 删除表
nft delete table ip foo

# 添加链
nft add table ip filter
nft add chain ip filter input { type filter hook input priority 0 \; }
nft add chain ip filter output { type filter hook output priority 0 \; }

# 添加规则
nft add rule filter output ip daddr 8.8.8.8 counter
nft add rule filter output tcp dport ssh counter
nft insert rule filter output ip daddr 192.168.1.1 counter

# 列出规则
nft list table filter

# 删除规则
nft list table filter -a # 查询handle是多少
nft delete rule filter output handle 5

# 删除链中所有规则
nft delete rule filter output

# 删除表中所有规则
nft flush table filter
```

## 参考文档

- [A Deep Dive into Iptables and Netfilter Architecture](#)
- <http://www.netfilter.org/>
- [iptables wiki](#)
- [nftables wiki](#)
- [Linux数据包路由原理、Iptables/netfilter入门学习](#)

# 负载均衡

## Ivs

Linux Virtual Server (Ivs) 是Linux内核自带的负载均衡器，也是目前性能最好的软件负载均衡器之一。Ivs 包括ipvs 内核模块和ipvsadm 用户空间命令行工具两部分。

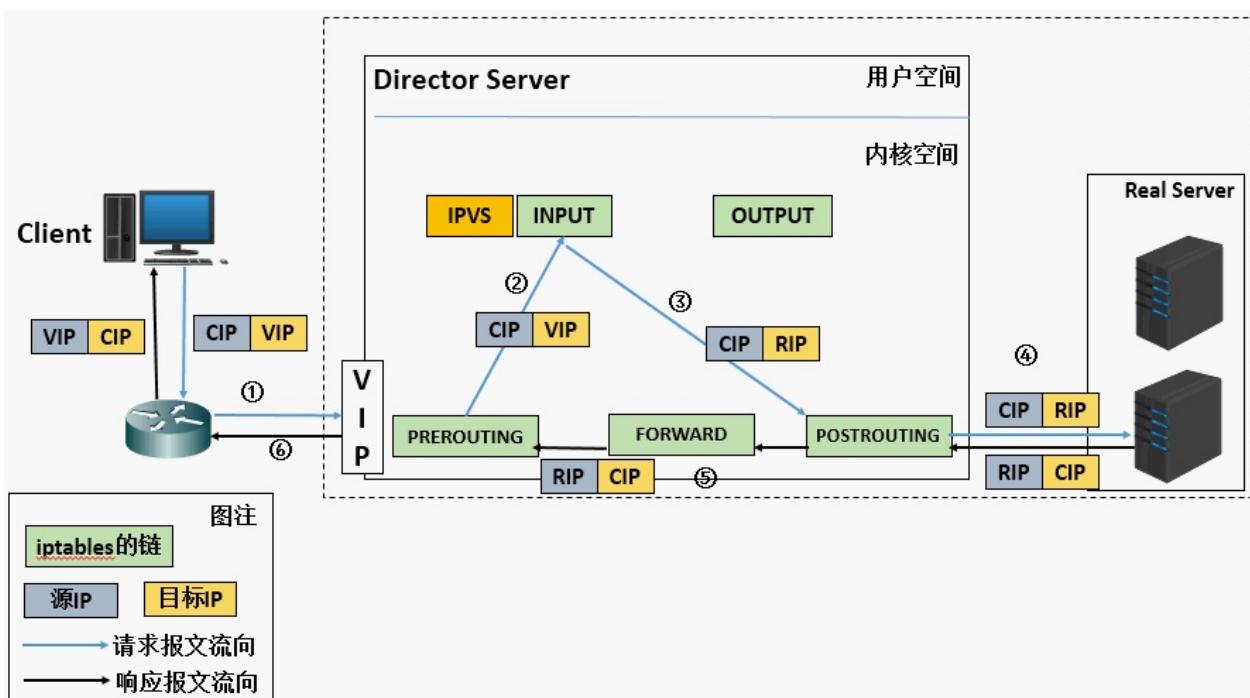
在Ivs中，节点分为Director Server和Real Server两个角色，其中Director Server是负载均衡器所在节点，而Real Server则是后端服务节点。当用户的请求到达Director Server时，内核 netfilter机制的PREROUTING链会将发往本地IP的包转发给INPUT链（也就是ipvs的工作链），在INPUT链上，ipvs根据用户定义的规则对数据包进行处理（如修改目的IP和端口等），并把新的包发送到POSTROUTING链，进而再转发给Real Server。

## 转发模式

### NAT

NAT模式通过修改数据包的目的IP和目的端口来将包转发给Real Server。它的特点包括

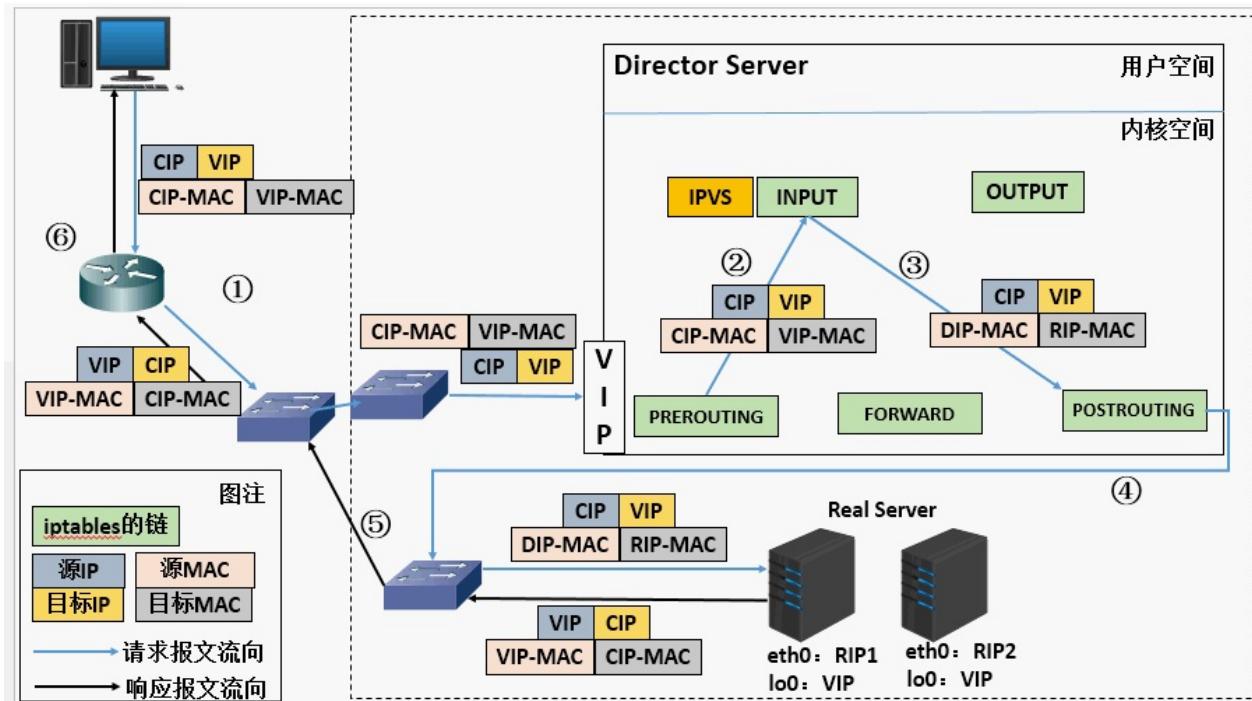
- Director Server必须作为Real Server的网关，并且它们必须处于同一个网段内
- 不需要Real Server做任何特殊配置
- 支持端口映射
- 请求和响应都需要经过Director Server，易称为性能瓶颈



## DR

DR (Direct Route) 模式通过修改数据包的目的MAC地址将包转发给Real Server。它的特点包括

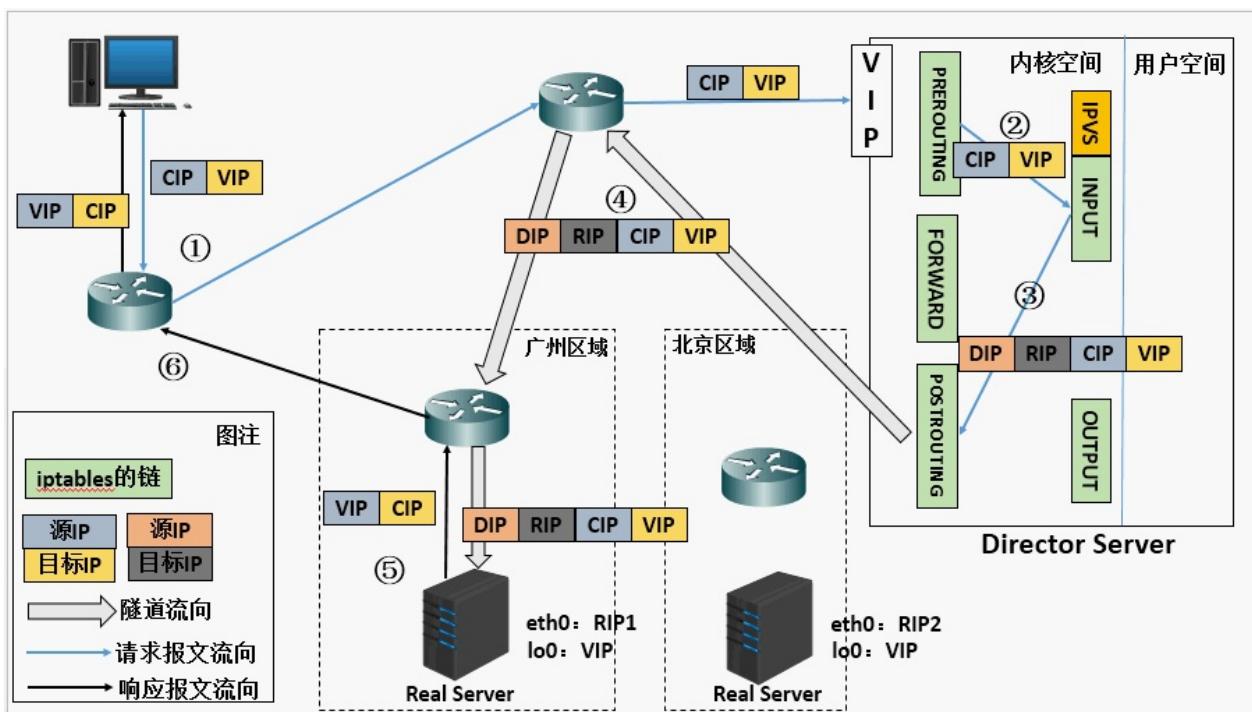
- 需要在Real Server的lo上配置vip，并配置arp\_ignore和arp\_announce忽略对vip的ARP解析请求
- Director Server和Real Server必须在同一个物理网络内，二层可达
- 虽然所有请求包都会经过Director Server，但响应报文不经过，有性能上的优势



## TUN

TUN模式通过将数据包封装在另一个IP包中（源地址为DIP，目的为RIP）将包转发给Real Server。它的特点包括

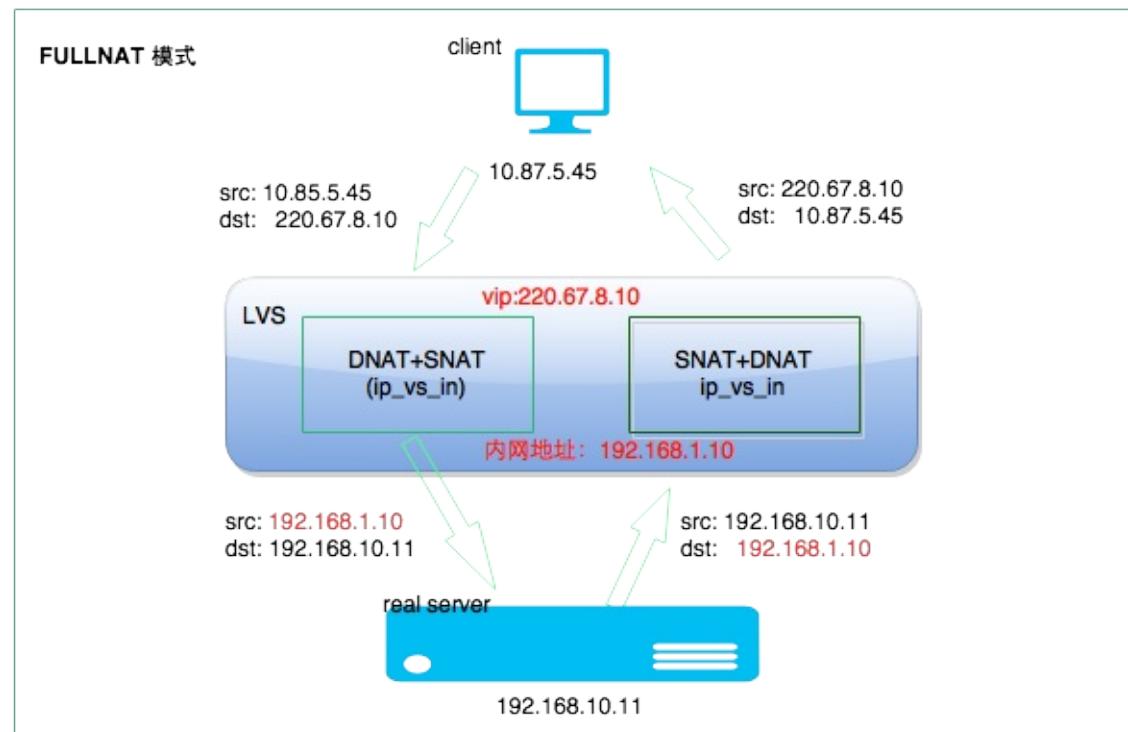
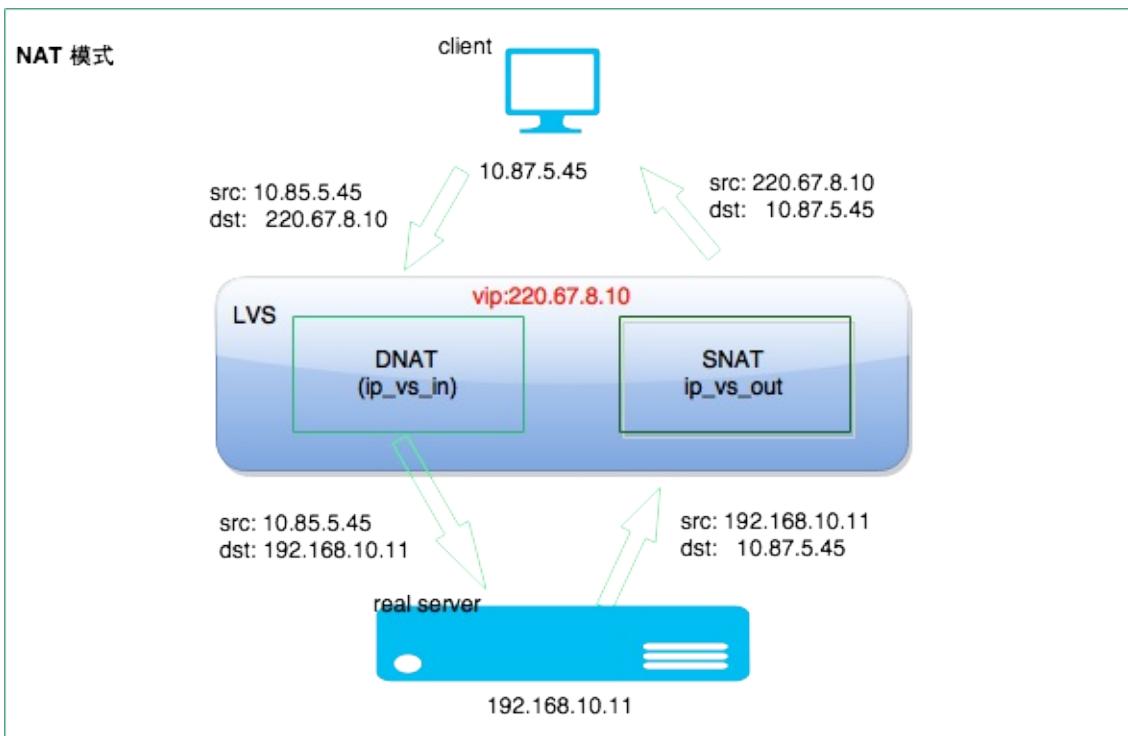
- Real Server需要在lo上配置vip，但不需要Director Server作为网关
- 不支持端口映射



## FULLNAT

FULLNAT是阿里在NAT基础上增加的一个新转发模式，通过引入local IP（CIP-VIP转换为LIP->RIP，而LIP和RIP均为IDC内网IP）使得物理网络可以跨越不同vlan，代码维护在<https://github.com/alibaba/LVS>上面。其特点是

- 物理网络仅要求三层可达
- Real Server不需要任何特殊配置
- SYNPROXY防止synflooding攻击
- 未进入内核主线，维护复杂



## 调度算法

- 轮叫调度 (Round-Robin Scheduling)
- 加权轮叫调度 (Weighted Round-Robin Scheduling)
- 最小连接调度 (Least-Connection Scheduling)
- 加权最小连接调度 (Weighted Least-Connection Scheduling)

- 基于局部性的最少链接（Locality-Based Least Connections Scheduling）
- 带复制的基于局部性最少链接（Locality-Based Least Connections with Replication Scheduling）
- 目标地址散列调度（Destination Hashing Scheduling）
- 源地址散列调度（Source Hashing Scheduling）
- 最短预期延时调度（Shortest Expected Delay Scheduling）
- 不排队调度（Never Queue Scheduling）

## Ivs配置示例

安装ipvs包并开启ip转发

```
yum -y install ipvsadm keepalived  
sysctl -w net.ipv4.ip_forward=1
```

修改/etc/keepalived/keepalived.conf，增加vip和lvs的配置

```

vrrp_instance VI_3 {
    state MASTER    # 另一节点为BACKUP
    interface eth0
    virtual_router_id 11
    priority 100    # 另一节点为50
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass PASSWORD
    }

    track_script {
        chk_http_port
    }

    virtual_ipaddress {
        192.168.0.100
    }
}

virtual_server 192.168.0.100 9696 {
    delay_loop 30
    lb_algo rr
    lb_kind DR
    persistence_timeout 30
    protocol TCP

    real_server 192.168.0.101 9696 {
        weight 3
        TCP_CHECK {
            connect_timeout 10
            nb_get_retry 3
            delay_before_retry 3
            connect_port 9696
        }
    }

    real_server 192.168.0.102 9696 {
        weight 3
        TCP_CHECK {
            connect_timeout 10
            nb_get_retry 3
            delay_before_retry 3
            connect_port 9696
        }
    }
}

```

重启 keepalived :

```
systemctl reload keepalived
```

最后在neutron-server所在机器上为lo配置vip，并抑制ARP响应：

```
vip=192.168.0.100
ifconfig lo:1 ${vip} broadcast ${vip} netmask 255.255.255.255
route add -host ${vip} dev lo:1
echo "1" >/proc/sys/net/ipv4/conf/lo/arp_ignore
echo "2" >/proc/sys/net/ipv4/conf/lo/arp_announce
echo "1" >/proc/sys/net/ipv4/conf/all/arp_ignore
echo "2" >/proc/sys/net/ipv4/conf/all/arp_announce
```

## LVS缺点

- Keepalived主备模式设备利用率低；不能横向扩展；VRRP协议，有脑裂的风险。
- ECMP的方式需要了解动态路由协议，LVS和交换机均需要较复杂配置；交换机的HASH算法一般比较简单，增加删除节点会造成HASH重分布，可能导致当前TCP连接全部中断；部分交换机的ECMP在处理分片包时会有BUG。

## HAProxy

HAProxy也是Linux最常用的负载均衡软件之一，兼具性能和功能的组合，同时支持TCP和HTTP负载均衡。

配置和使用方法请见[官网](#)。

## Nginx

Nginx也是Linux最常用的负载均衡软件之一，常用作反向代理和HTTP负载均衡（当然也支持TCP和UDP负载均衡）。

配置和使用方法请见[官网](#)。

## 自研负载均衡

## Google Maglev

Maglev是Google自研的负载均衡方案，在2008年就已经开始用于生产环境。Maglev安装后不需要预热5秒内就能处理每秒100万次请求。谷歌的性能基准测试中，Maglev实例运行在一个8核CPU下，网络吞吐率上限为12M PPS（数据包每秒）。如果Maglev使用Linux内核网络堆

栈则速度会慢下来，吞吐率小于4M PPS。

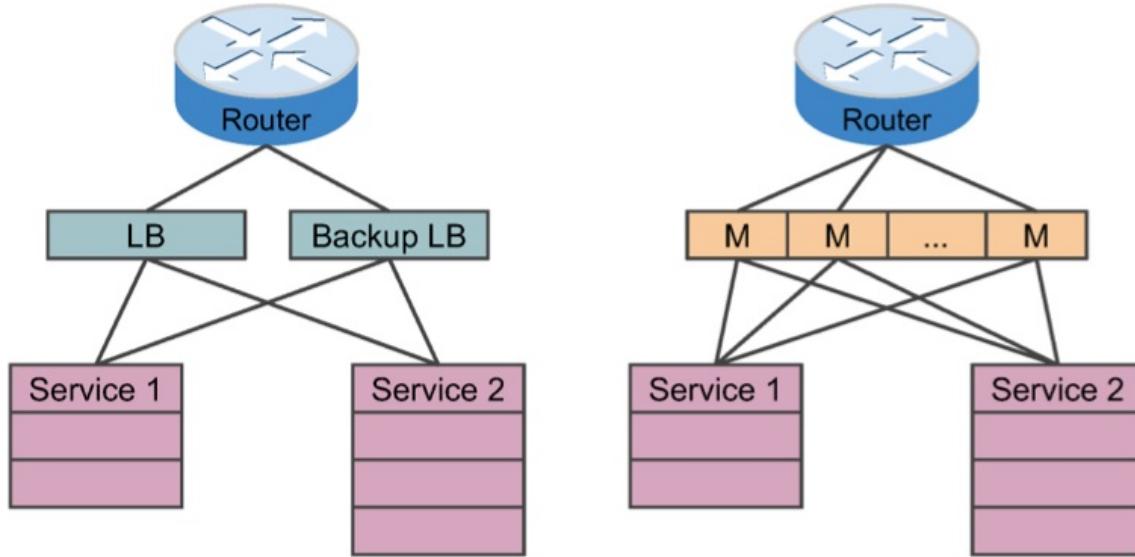


Figure 1: Hardware load balancer and Maglev.

- 路由器ECMP (Equal Cost Multipath) 转发包到Maglev (而不是传统的主从结构)
- Kernel Bypass, CPU绑定，共享内存
- 一致性哈希保证连接不中断

## UCloud Vortex

Vortex参考了Maglev，大致的架构和实现跟Maglev类似：

- ECMP实现集群的负载均衡
- 一致性哈希保证连接不中断
  - 即使是不同的Vortex服务器收到了数据包，仍然能够将该数据包转发到同一台后端服务器
  - 后端服务器变化时，通过连接追踪机制保证当前活动连接的数据包被送往之前选择的服务器，而所有新建连接则会在变化后的服务器集群中进行负载分担
- DPDK提升单机性能 (14M PPS, 10G, 64字节线速)
  - 通过RSS直接将网卡队列和CPU Core绑定，消除线程的上下文切换带来的开销
  - Vortex线程间采用高并发无锁的消息队列通信
- DR模式避免额外开销

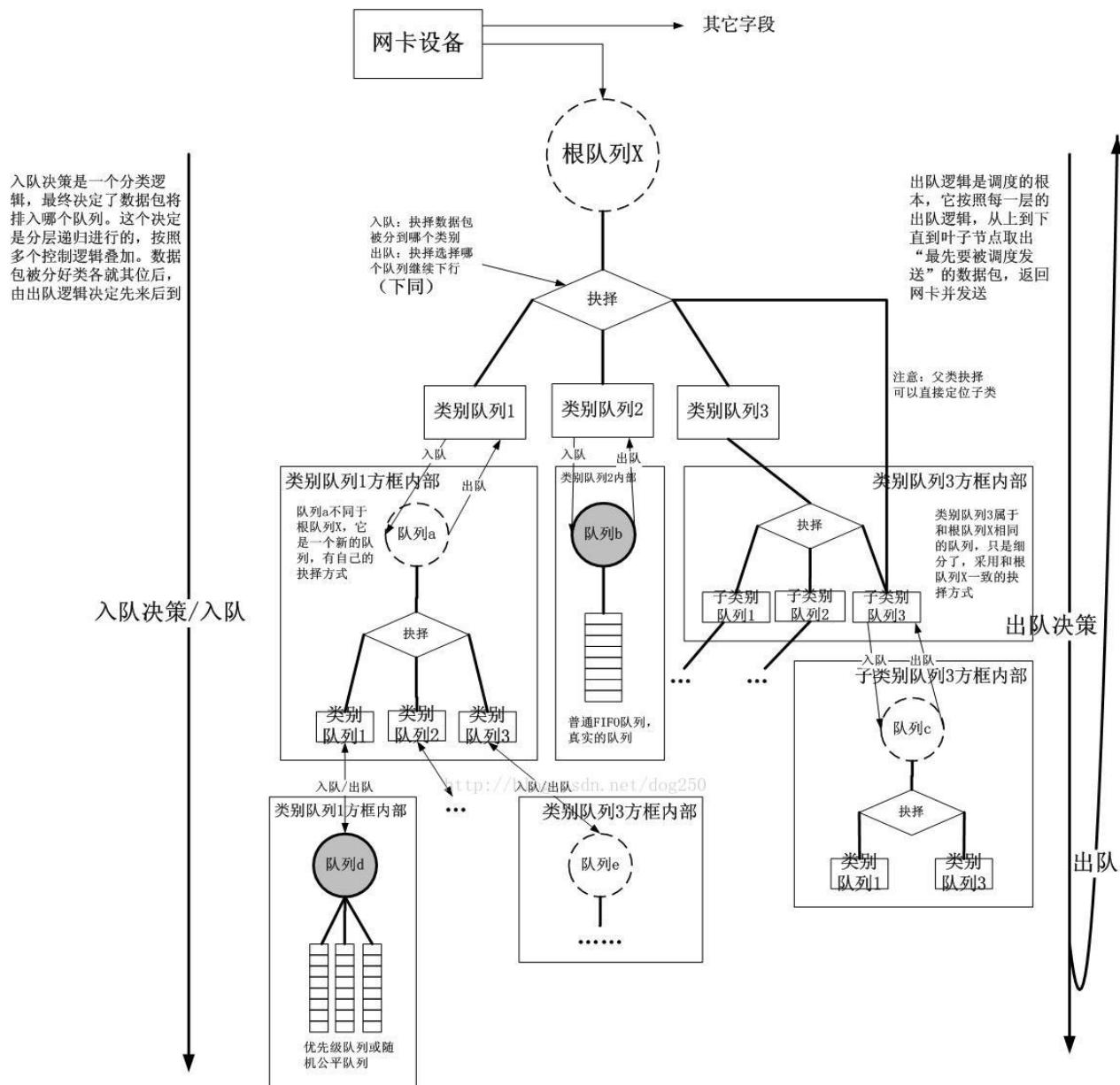
## 参考文档

- <http://www.linuxvirtualserver.org/>

- <http://www.haproxy.org/>
- 揭秘100G+线速云负载均衡的设计与实现：从Maglev到Vortex
- 你真的掌握lvs工作原理吗
- lvs 负载均衡fullnat 模式clientip 怎样传递给 realserver

## 流量控制

流量控制（Traffic Control，tc）是Linux内核提供的流量限速、整形和策略控制机制。它以 `qdisc-class-filter` 的树形结构来实现对流量的分层控制：



## 图示解释



不存在的队列，只是一个入队/出队回调函数的调用，具体实现不关心



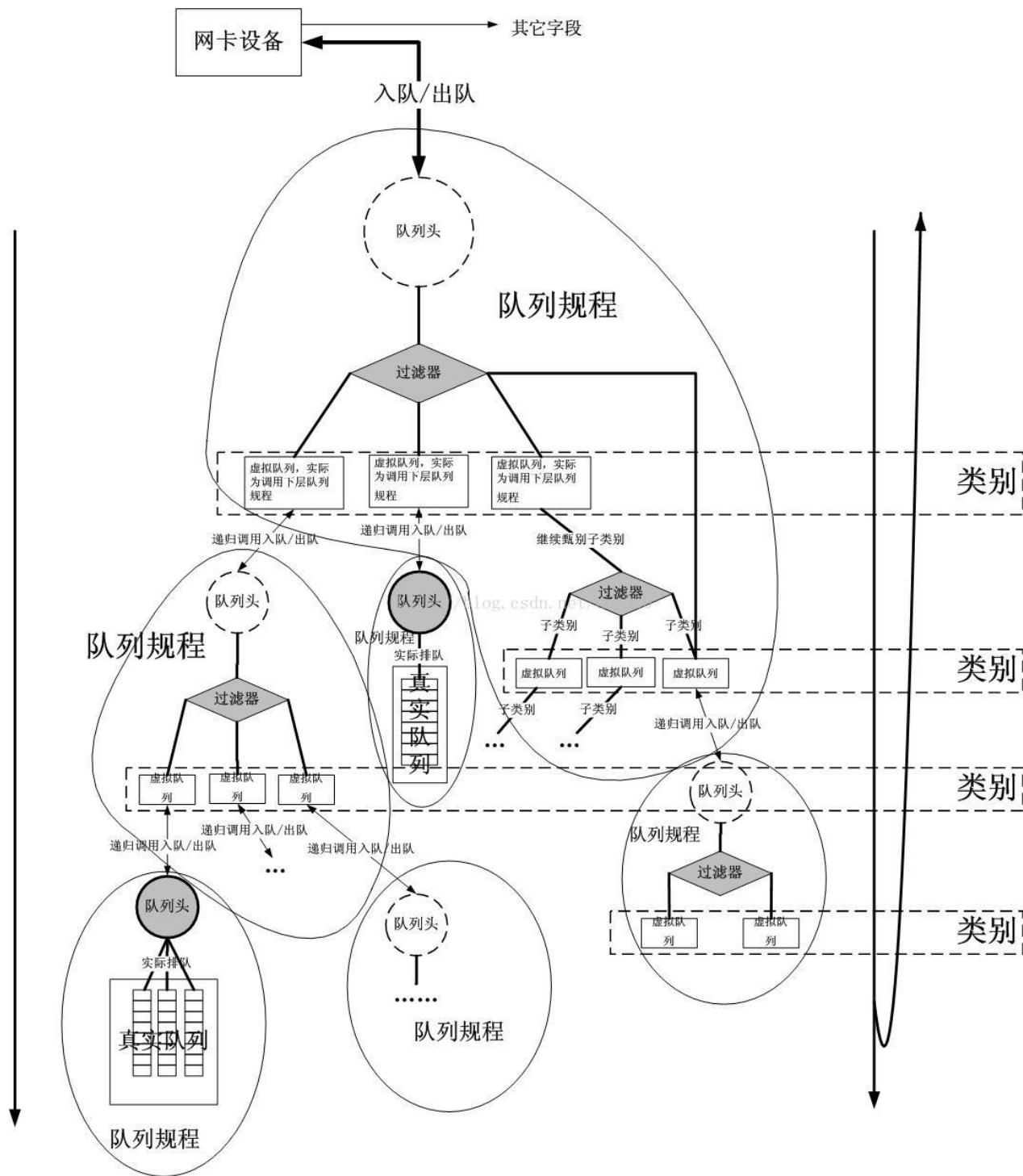
真实的队列，数据包将被排队



所谓的类别，只是一个递归队列的封装



入队：抉择数据包被分到哪个类别  
出队：抉择选择哪个队列继续下行



tc最佳的参考就是[Linux Traffic Control HOWTO](#)，详细介绍了tc的原理和使用方法。

## 基本组成

从上图中可以看到，tc由qdisc、filter和class三部分组成：

- **qdisc**通过队列将数据包缓存起来，用来控制网络收发的速度
- **class**用来表示控制策略
- **filter**用来将数据包划分到具体的控制策略中

## qdisc

qdisc通过队列将数据包缓存起来，用来控制网络收发的速度。实际上，每个网卡都有一个关联的qdisc。它包括以下几种：

- 无分类qdisc（只能应用于root队列）

- [p|b]fifo : 简单先进先出
- pfifo\_fast : 根据数据包的tos将队列划分到3个band，每个band内部先进先出
- red : Random Early Detection，带宽接近限制时随机丢包，适合高带宽应用
- sfq : Stochastic Fairness Queueing，按照会话对流量排序并循环发送每个会话的数据包
- tbf : Token Bucket Filter，只允许以不超过事先设定的速率到来的数据包通过，但可能允许短暂突发流量超过设定值

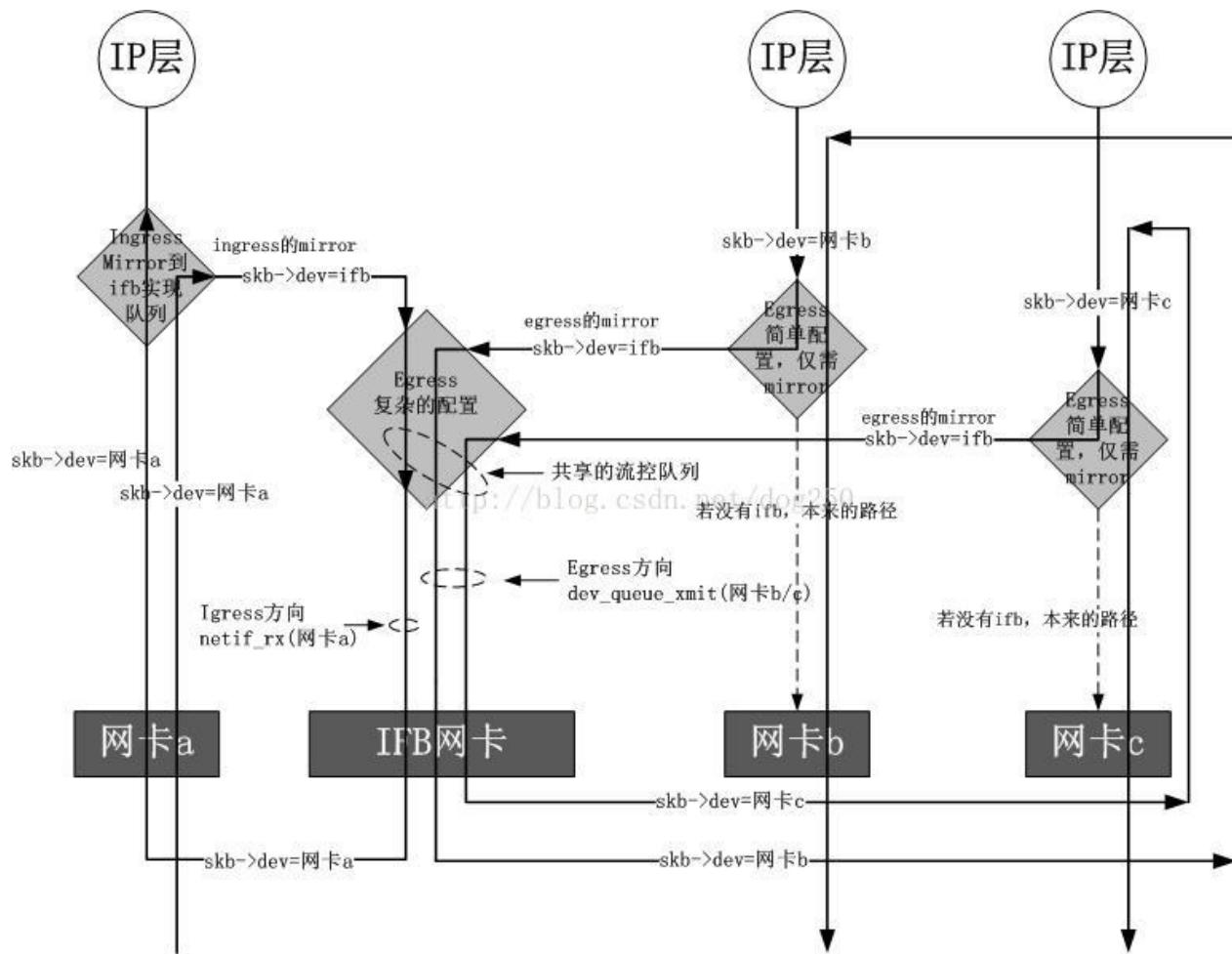
- 有分类qdisc（可以包括多个队列）

- cbq : Class Based Queueing，借助EWMA(exponential weighted moving average, 指数加权移动均值)算法确认链路的闲置时间足够长，以达到降低链路实际带宽的目的。如果发生越限，CBQ就会禁止发包一段时间。
- htb : Hierarchy Token Bucket，在tbf的基础上增加了分层
- prio : 分类优先算法并不进行整形，它仅仅根据你配置的过滤器把流量进一步细分。缺省会自动创建三个FIFO类。

注意，一般说到qdisc都是指egress qdisc。每块网卡实际上还可以添加一个ingress qdisc，不过它有诸多的限制

- ingress qdisc不能包含子类，而只能作过滤
- ingress qdisc只能用于简单的整形

如果相对ingress方向作流量控制的话，可以借助ifb（Intermediate Functional Block）内核模块。因为流入网络接口的流量是无法直接控制的，那么就需要把流入的包导入（通过tc action）到一个中间的队列，该队列在ifb设备上，然后让这些包重走tc层，最后流入的包再重新入栈，流出的包重新出栈。



## filter

filter用来将数据包划分到具体的控制策略中，包括以下几种：

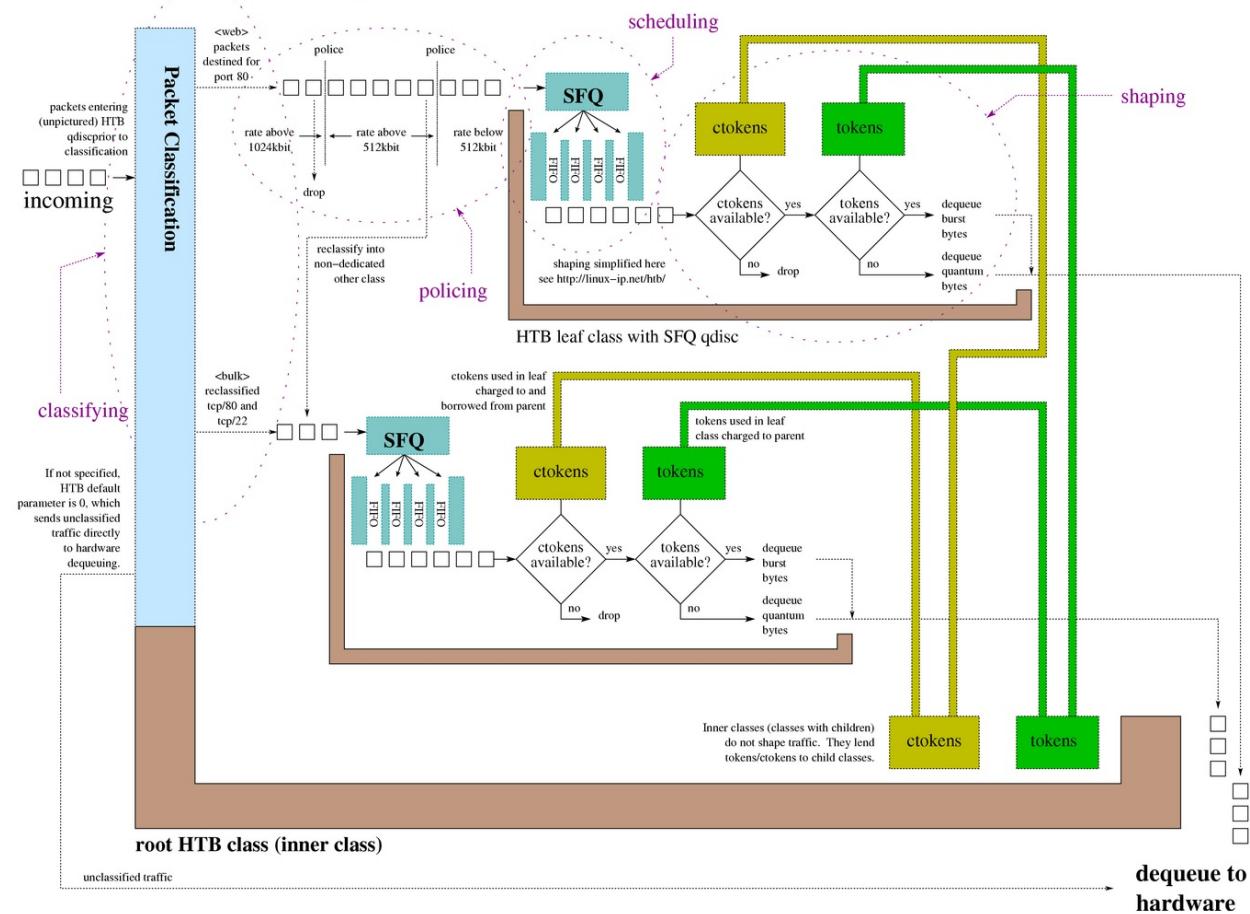
- u32：根据协议、IP、端口等过滤数据包
- fwmark：根据iptables MARK来过滤数据包
- tos：根据tos字段过滤数据包

## class

class用来表示控制策略，只用于有分类的qdisc上。每个class要么包含多个子类，要么只包含一个子qdisc。当然，每个class还包括一些列的filter，控制数据包流向不同的子类，或者是直接丢掉。

## htb示例

## Simplified Linux Traffic Control Scenario with HTB



```

# add qdisc
tc qdisc add dev eth0 root handle 1: htb default 2 r2q 100
# add default class
tc class add dev eth0 parent 1:0 classid 1:1 htb rate 1000mbit ceil 1000mbit
tc class add dev eth0 parent 1:1 classid 1:2 htb prio 5 rate 1000mbit ceil 1000mbit
tc qdisc add dev eth0 parent 1:2 handle 2: pfifo limit 500
# add default filter
tc filter add dev eth0 parent 1:0 prio 5 protocol ip u32
tc filter add dev eth0 parent 1:0 prio 5 handle 3: protocol ip u32 divisor 256
tc filter add dev eth0 parent 1:0 prio 5 protocol ip u32 ht 800:: match ip src 192.168
.0.0/16 hashkey mask 0x000000ff at 12 link 3:

# add egress rules for 192.168.0.9
tc class add dev eth0 parent 1:1 classid 1:9 htb prio 5 rate 3mbit ceil 3mbit
tc qdisc add dev eth0 parent 1:9 handle 9: pfifo limit 500
tc filter add dev eth0 parent 1: protocol ip prio 5 u32 ht 3:9: match ip src "192.168.
0.9" flowid 1:9

```

## ifb示例

```
# init ifb
modprobe ifb numifbs=1
ip link set ifb0 up
# redirect ingress to ifb0
tc qdisc add dev eth0 ingress handle ffff:
tc filter add dev eth0 parent ffff: protocol ip prio 0 u32 match u32 0 0 flowid ffff:
action mirred egress redirect dev ifb0
# add qdisc
tc qdisc add dev ifb0 root handle 1: htb default 2 r2q 100
# add default class
tc class add dev ifb0 parent 1:0 classid 1:1 htb rate 1000mbit ceil 1000mbit
tc class add dev ifb0 parent 1:1 classid 1:2 htb prio 5 rate 1000mbit ceil 1000mbit
tc qdisc add dev ifb0 parent 1:2 handle 2: pfifo limit 500
# add default filter
tc filter add dev ifb0 parent 1:0 prio 5 protocol ip u32
tc filter add dev ifb0 parent 1:0 prio 5 handle 4: protocol ip u32 divisor 256
tc filter add dev ifb0 parent 1:0 prio 5 protocol ip u32 ht 800:: match ip dst 192.168
.0.0/16 hashkey mask 0x000000ff at 16 link 4:

# add ingress rules for 192.168.0.9
tc class add dev ifb0 parent 1:1 classid 1:9 htb prio 5 rate 3mbit ceil 3mbit
tc qdisc add dev ifb0 parent 1:9 handle 9: pfifo limit 500
tc filter add dev ifb0 parent 1: protocol ip prio 5 u32 ht 4:9: match ip dst "192.168.
0.9" flowid 1:9
```

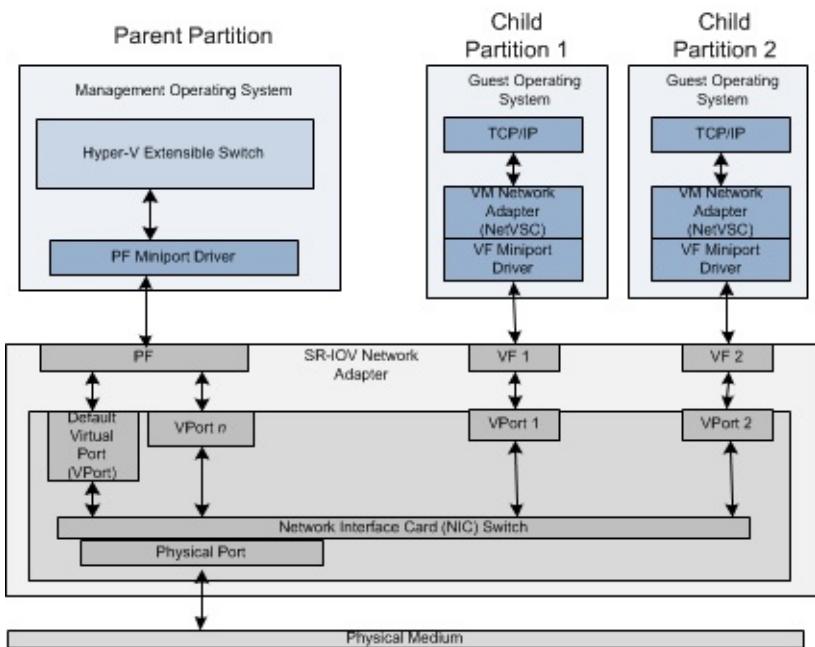
## 参考文档

- [Linux Traffic Control HOWTO](#)
- [ifb wiki](#)
- [Linux TC \(Traffic Control\) 框架原理解析](#)
- [Linux TC的ifb原理以及ingress流控](#)

# SR-IOV

SR-IOV (Single Root I/O Virtualization) 是一个将PCIe共享给虚拟机的标准，通过为虚拟机提供独立的内存空间、中断、DMA流，来绕过VMM实现数据访问。SR-IOV基于两种PCIe functions：

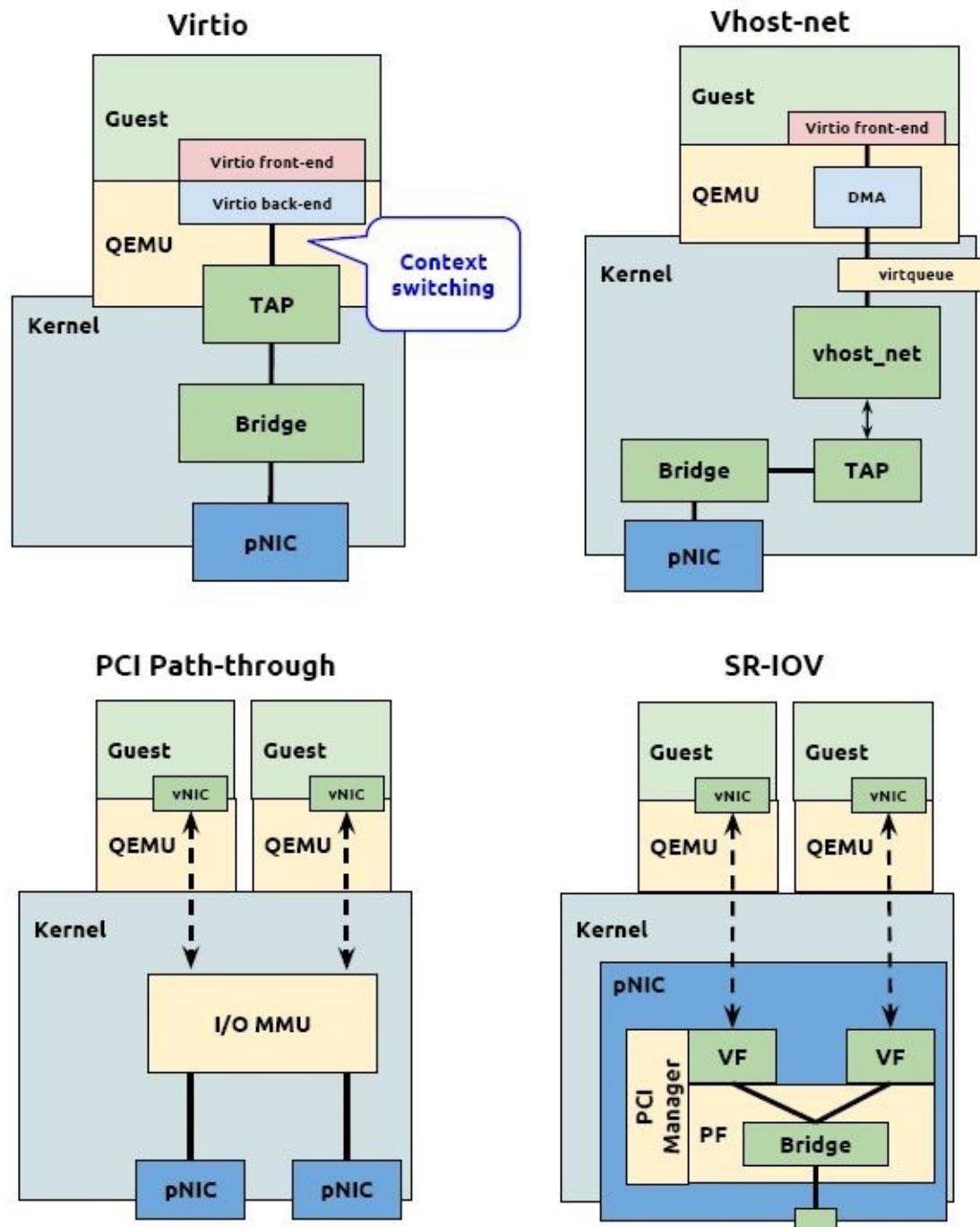
- PF (Physical Function)：包含完整的PCIe功能，包括SR-IOV的扩张能力，该功能用于SR-IOV的配置和管理。
- VF (Virtual Function)：包含轻量级的PCIe功能。每一个VF有它自己独享的PCI配置区域，并且可能与其他VF共享着同一个物理资源



## SR-IOV要求

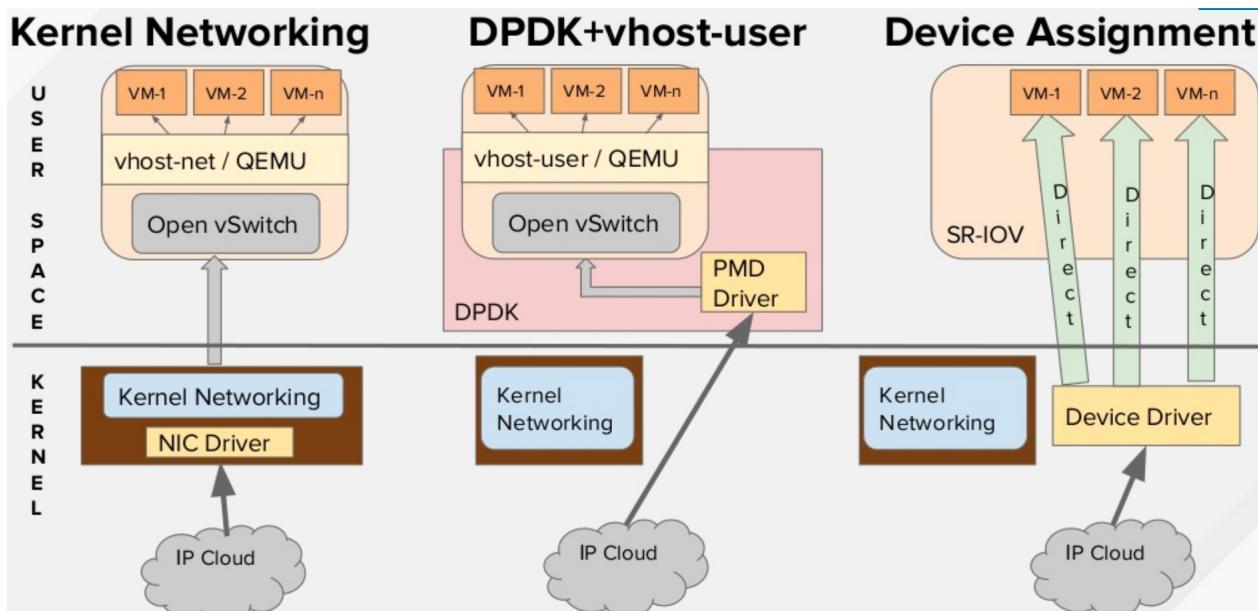
- CPU 必须支持IOMMU（比如英特尔的 VT-d 或者AMD的 AMD-Vi，Power8 处理器默认支持IOMMU）
- 固件Firmware 必须支持IOMMU
- CPU 根桥必须支持 ACS 或者ACS等价特性
- PCIe 设备必须支持ACS 或者ACS等价特性
- 建议根桥和PCIe 设备中间的所有PCIe 交换设备都支持ACS，如果某个PCIe交换设备不支持ACS，其后的所有PCIe设备只能共享某个IOMMU 组，所以只能分配给1台虚机。

## SR-IOV vs PCI path-through

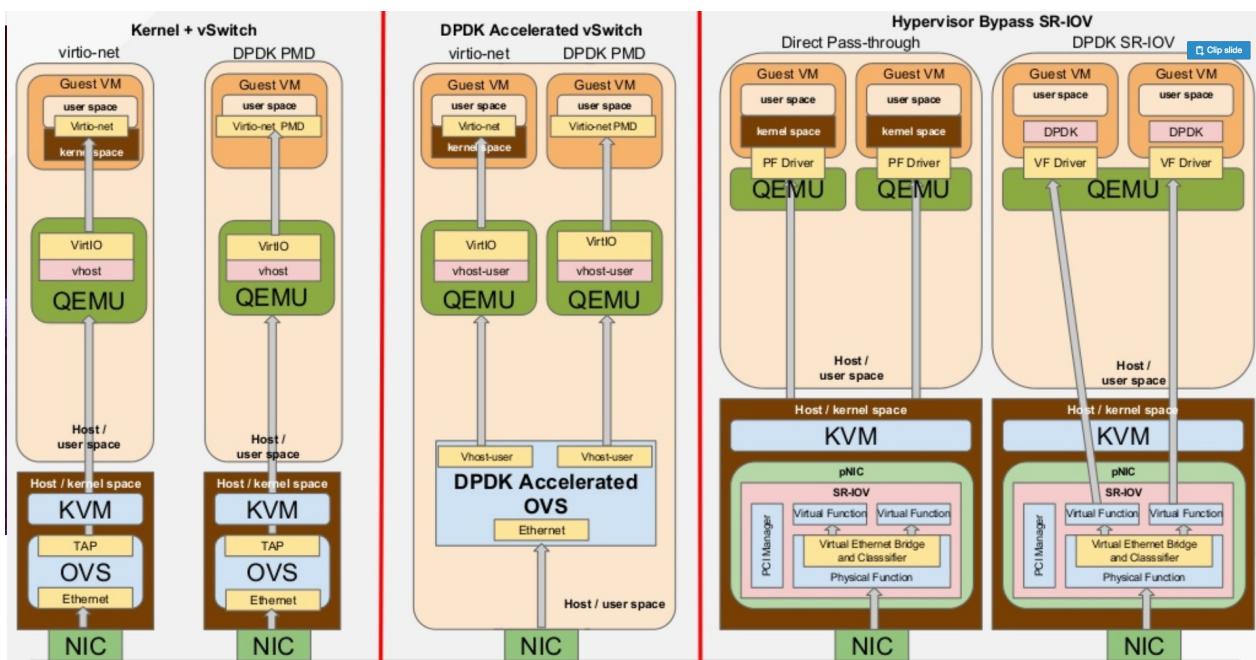


<b>Basic Operation</b>	<ul style="list-style-type: none"> <li>- Backend/Guest direct access to shared Vring buffers - PIO</li> <li>- Switching at software level</li> <li>- Management Flexibility – internal SDN support  <code>ovs-vsctl add-port br0 &lt;phys-intfc&gt;</code> - vSwitch  <code>ovs-ofctl</code> – control flows</li> <li>- IRQ bottleneck – QEMU – call into kvm inject Kernel – inject directly</li> </ul>	<ul style="list-style-type: none"> <li>- Direct access to hw memory regions</li> <li>- DMA Support</li> <li>- Switching at hw level – SR-IOV depends on #of Queues</li> <li>- Management Flexibility – external SDN capable</li> <li>- IRQ bottleneck – hw enhancements, posted interrupts, exitless EOI improve things – closer to native</li> </ul>
<b>Migration</b>	<ul style="list-style-type: none"> <li>- Virtio lockless</li> <li>- Saves device state, tracks dirty pages</li> </ul>	<ul style="list-style-type: none"> <li>- QEMU sets 'unmigratable', or installs migration blocker</li> <li>- Guest can be holding a lock – deadlock, hw state, ....</li> </ul>
<b>Scalability</b>	<ul style="list-style-type: none"> <li>- Practical limitations – primarily performance</li> </ul>	<ul style="list-style-type: none"> <li>- Number of Devices limited, limits #VMs</li> <li>- SR-IOV - #of VF - # of queues</li> </ul>
<b>Network Performance</b>	<ul style="list-style-type: none"> <li>- Soft switching – bridge, vSwitch</li> <li>- Several IO HOPS</li> <li>- Can approach near native – 10Ge for few bridged Guest</li> </ul>	<ul style="list-style-type: none"> <li>- Switching done at HW level – hw queues</li> <li>- Performance scales with # of Guests</li> <li>- DMA support</li> <li>- IRQ Passthrough still a problem</li> </ul>
<b>Host Performance</b>	<ul style="list-style-type: none"> <li>- PIO – takes cpu cycles</li> <li>- Exits – few but still</li> <li>- Guest pages swapable</li> </ul>	<ul style="list-style-type: none"> <li>- Guest pinned – can't swap</li> <li>- Fewer exits</li> <li>- Less PIO</li> </ul>
<b>Cloud Environment</b>	<ul style="list-style-type: none"> <li>- Cloud friendly – migration, SDN, paging</li> </ul>	<ul style="list-style-type: none"> <li>- Not Cloud friendly, great for NFV/RT DPDK, run to completion</li> </ul>

## SR-IOV vs DPDK



OVS+kernel	OVS+DPDK	SR-IOV
<b>Pros</b>	<b>Pros</b>	<b>Pros</b>
<ul style="list-style-type: none"> <li>• Feature rich / robust solution</li> <li>• Ultra flexible</li> <li>• Integration with SDN</li> <li>• Supports Live Migration</li> <li>• Supports overlay networking</li> <li>• Full Isolation support / Namespace / multi-tenancy</li> </ul>	<ul style="list-style-type: none"> <li>• Packets directly sent to user space</li> <li>• Line rate performance with tiny packets</li> <li>• Integration with SDN</li> <li>• CPU consumption</li> </ul>	<ul style="list-style-type: none"> <li>• Line rate performance</li> <li>• Packets directly sent to VMs</li> <li>• HW based isolation</li> <li>• No CPU burden</li> </ul>
<b>Cons</b>	<b>Cons</b>	<b>Cons</b>
<ul style="list-style-type: none"> <li>• CPU consumption</li> </ul>	<ul style="list-style-type: none"> <li>• More dependence on user space packages</li> </ul>	<ul style="list-style-type: none"> <li>• 10s of VMs or fewer</li> <li>• Not as flexible</li> <li>• No OVS</li> <li>• Need VF driver in the guest</li> <li>• Switching at ToR</li> <li>• No Live Migration</li> <li>• No overlays</li> </ul>
	<b>WIP</b>	
	<ul style="list-style-type: none"> <li>• Live Migration</li> <li>• IOMMU support</li> </ul>	



## SR-IOV 使用示例

开启VF：

```
modprobe -r igb
modprobe igb max_vfs=7
echo "options igb max_vfs=7" >>/etc/modprobe.d/igb.conf
```

查找Virtual Function：

```
# lspci | grep 82576
0b:00.0 Ethernet controller: Intel Corporation 82576 Gigabit Network Connection (rev 01)
0b:00.1 Ethernet controller: Intel Corporation 82576 Gigabit Network Connection (rev 01)
0b:10.0 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0b:10.1 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0b:10.2 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0b:10.3 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0b:10.4 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0b:10.5 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0b:10.6 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0b:10.7 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0b:11.0 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0b:11.1 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0b:11.2 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0b:11.3 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0b:11.4 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0b:11.5 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)

# virsh nodedev-list | grep 0b
pci_0000_0b_00_0
pci_0000_0b_00_1
pci_0000_0b_10_0
pci_0000_0b_10_1
pci_0000_0b_10_2
pci_0000_0b_10_3
pci_0000_0b_10_4
pci_0000_0b_10_5
pci_0000_0b_10_6
pci_0000_0b_11_7
pci_0000_0b_11_1
pci_0000_0b_11_2
pci_0000_0b_11_3
pci_0000_0b_11_4
pci_0000_0b_11_5
```

```
$ virsh nodedev-dumpxml pci_0000_0b_00_0
<device>
  <name>pci_0000_0b_00_0</name>
  <parent>pci_0000_00_01_0</parent>
  <driver>
    <name>igb</name>
  </driver>
  <capability type='pci'>
    <domain>0</domain>
    <bus>11</bus>
    <slot>0</slot>
    <function>0</function>
    <product id='0x10c9'>82576 Gigabit Network Connection</product>
    <vendor id='0x8086'>Intel Corporation</vendor>
  </capability>
</device>
```

通过**libvirt**绑定到虚拟机

```
$ cat >/tmp/interface.xml <<EOF
<interface type='hostdev' managed='yes'>
  <source>
    <address type='pci' domain='0' bus='11' slot='16' function='0' />
  </source>
</interface>
EOF
$ virsh attach-device MyGuest /tmp/interface. xml --live --config
```

当然也可以给网卡配置MAC地址和VLAN：

```
<interface type='hostdev' managed='yes'>
  <source>
    <address type='pci' domain='0' bus='11' slot='16' function='0' />
  </source>
  <mac address='52:54:00:6d:90:02'>
  <vlan>
    <tag id='42' />
  </vlan>
  <virtualport type='802.1Qbh'>
    <parameters profileid='finance' />
  </virtualport>
</interface>
```

通过**Qemu**绑定到虚拟机

```
/usr/bin/qemu-kvm -name vdisk -enable-kvm -m 512 -smp 2 \
-hda /mnt/nfs/vdisk.img \
-monitor stdio \
-vnc 0.0.0.0:0 \
-device pci-assign,host=0b:00.0
```

## 优缺点

Pros:

- More Scalable than Direct Assign
- Security through IOMMU and function isolation
- Control Plane separation through PF/VF notion
- High packet rate, Low CPU, Low latency thanks to Direct Pass through

Cons:

- Rigid: Composability issues
- Control plane is pass through, puts pressure on Hardware resources
- Parts of the PCIe config space are direct map from Hardware
- Limited scalability (16 bit)
- SR-IOV NIC forces switching features into the HW
- All the Switching Features in the Hardware or nothing

## 参考文档

- [Intel SR-IOV Configuration Guide](#)
- [OpenStack SR-IOV Passthrough for Networking](#)
- [Redhat OpenStack SR-IOV Configure](#)
- [SDN Fundamentals for NFV, Openstack and Containers](#)
- [I/O设备直接分配和SRIOV](#)
- [Libvirt PCI passthrough of host network devices](#)
- [Story of Network Virtualization and its future in Software and Hardware](#)

# Virtual Routing and Forwarding (VRF)

Linux内核的Virtual Routing and Forwarding (VRF) 是由路由表和一组网络设备组成的路由实例。

## VRF安装

Ubuntu默认不包括vrf内核模块，需要额外安装：

```
apt-get install linux-headers-4.10.0-14-generic linux-image-extra-4.10.0-14-generic
reboot
apt-get install linux-image-extra-$(uname -r)
modprobe vrf
```

## VRF示例

```
# create vrf device
ip link add vrf-blue type vrf table 10
ip link set dev vrf-blue up

# An l3mdev FIB rule directs lookups to the table associated with the device.
# A single l3mdev rule is sufficient for all VRFs.
# Prior to the v4.8 kernel iif and oif rules are needed for each VRF device:
ip ru add oif vrf-blue table 10
ip ru add iif vrf-blue table 10

#Set the default route for the table (and hence default route for the VRF).
ip route add table 10 unreachable default

# Enslave L3 interfaces to a VRF device.
# Local and connected routes for enslaved devices are automatically moved to
# the table associated with VRF device. Any additional routes depending on
# the enslaved device are dropped and will need to be reinserted to the VRF
# FIB table following the enslavement.
ip link set dev eth1 master vrf-blue

# The IPv6 sysctl option keep_addr_on_down can be enabled to keep IPv6 global
# addresses as VRF enslavement changes.
sysctl -w net.ipv6.conf.all.keep_addr_on_down=1

# Additional VRF routes are added to associated table.
ip route add table 10 ...
```

## 进程绑定VRF

Linux进程可以通过在VRF设备上监听socket来绑定VRF：

```
setsockopt(sd, SOL_SOCKET, SO_BINDTODEVICE, dev, strlen(dev)+1);
```

TCP & UDP services running in the default VRF context (ie., not bound to any VRF device) can work across all VRF domains by enabling the `tcp_13mdev_accept` and `udp_13mdev_accept` sysctl options:

```
sysctl -w net.ipv4.tcp_13mdev_accept=1
sysctl -w net.ipv4.udp_13mdev_accept=1
```

## VRF操作

### 创建VRF

```
ip link add dev NAME type vrf table ID
```

### 查询VRF列表

```
# ip -d link show type vrf
16: vrf-blue: <NOARP,MASTER,UP,LOWER_UP> mtu 65536 qdisc noqueue state UP mode DEFAULT
  group default qlen 1000
    link/ether 9e:9c:8e:7b:32:a4 brd ff:ff:ff:ff:ff:ff promiscuity 0
    vrf table 10 addrgenmode eui64
```

### 添加网卡到VRF

```
ip link set dev eth0 master vrf-blue
```

### 查询VRF邻接表和路由

```
ip neigh show vrf vrf-blue
ip addr show vrf vrf-blue
ip -br addr show vrf vrf-blue
ip route show vrf vrf-blue
```

## 从**VRF**中删除网卡

```
ip link set dev eth0 nomaster
```

参考文档

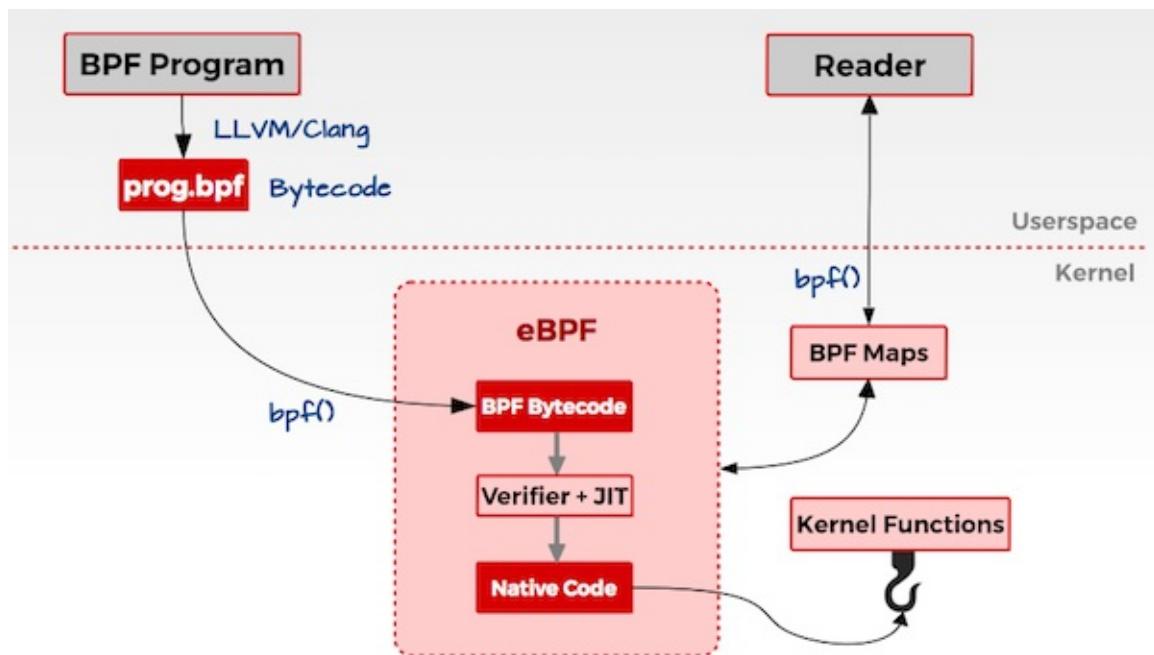
- [Linux kernel documentation](#)

# eBPF

eBPF (extended Berkeley Packet Filter) 起源于BPF，它提供了内核的数据包过滤机制。

BPF的基本思想是对用户提供两种SOCKET选项：`SO_ATTACH_FILTER` 和 `SO_ATTACH_BPF`，允许用户在socket上添加自定义的filter，只有满足该filter指定条件的数据包才会上发到用户空间。`SO_ATTACH_FILTER` 插入的是cBPF代码，`SO_ATTACH_BPF` 插入的是eBPF代码。eBPF是对cBPF的增强，目前用户端的tcpdump等程序还是用的cBPF版本，其加载到内核中后会被内核自动的转变为eBPF。

Linux 3.15 开始引入 eBPF。其扩充了 BPF 的功能，丰富了指令集。它在内核提供了一个虚拟机，用户态将过滤规则以虚拟机指令的形式传递到内核，由内核根据这些指令来过滤网络数据包。



BPF和eBPF的内核文档见[Documentation/networking/filter.txt](#)。

## 使用场景

eBPF使用场景包括

- XDP
- 流量控制
- 防火墙
- 网络包跟踪
- 内核探针

- cgroups
- bcc
- bpftools

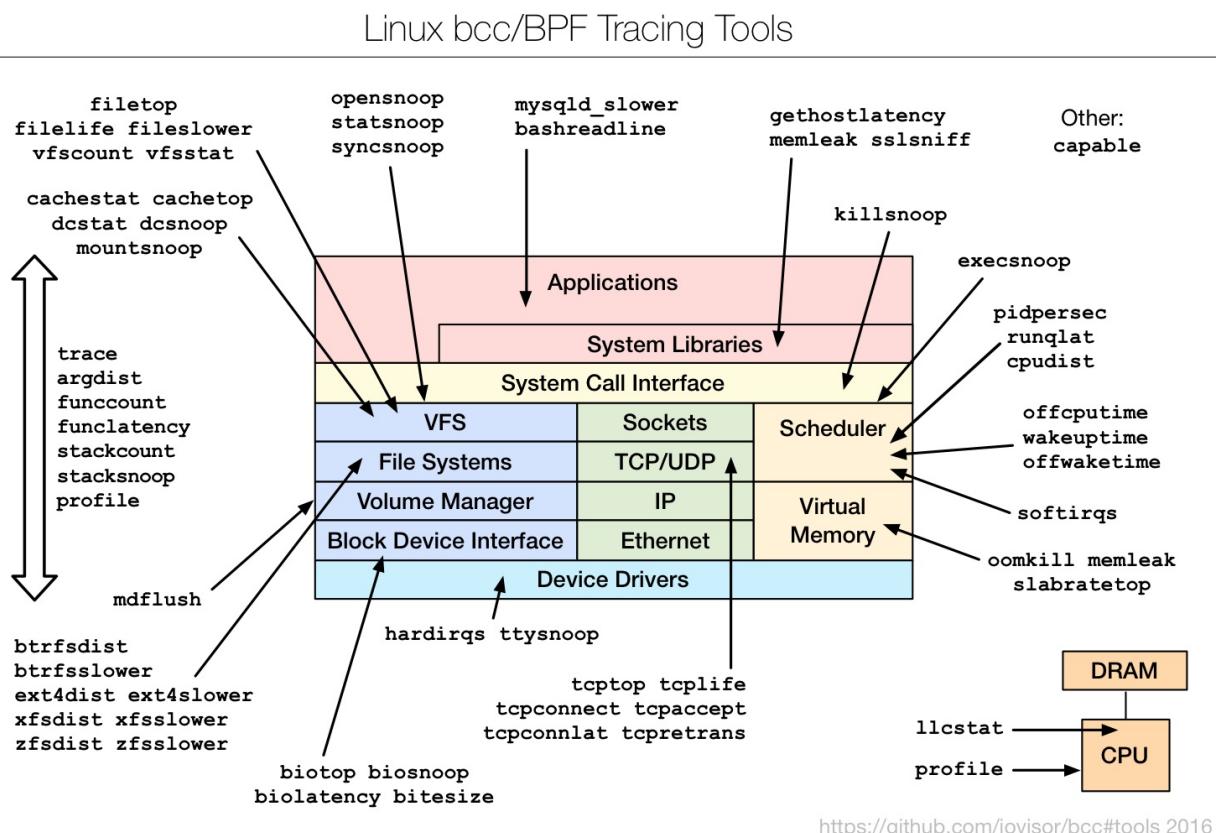
## 参考文档

- [Linux Kernel BPF documentation](#)
- [bcc](#)
- [The BSD Packet Filter: A New Architecture for User-level Packet Capture](#)
- [Notes on BPF & eBPF](#)

# BCC (BPF Compiler Collection)

BPF Compiler Collection (BCC)是基于eBPF的Linux内核分析、跟踪、网络监控工具。其源码存放于<https://github.com/iovisor/bcc>。

BCC包括一些列的工具



## 安装BCC

Ubuntu :

```
echo "deb [trusted=yes] https://repo.iovisor.org/apt/xenial xenial-nightly main" | sudo tee /etc/apt/sources.list.d/iovisor.list
sudo apt-get update
sudo apt-get install -y bcc-tools libbcc-examples python-bcc
```

CentOS :

```
echo -e '[iovisor]\nbaseurl=https://repo.iovisor.org/yum/nightly/f23/$basearch\nenable\n\nd=1\nngpgcheck=0' | sudo tee /etc/yum.repos.d/iovisor.repo
yum install -y bcc-tools
```

安装完成后，bcc工具会放到 /usr/share/bcc/tools 目录中

```
$ ls /usr/share/bcc/tools
argdist      cachestat  ext4dist          hardirqs        offwaketime  softirqs    tcpc
onnect      vfscount
bashreadline cachetop   ext4slower       killsnoop     old          solisten   tcpc
onlat      vfsstat
biolatency   capable    filelife         l1cstat       oomkill     ssldsniff  tcpl
ife      wakeuptime
biosnoop    cpudist    fileslower       mdflush      opensnoop   stackcount tcpr
etrans      xfsdist
biotop      dcsnoop    filetop          memleak      pidpersec   stacksnoop tcpt
op          xfsslower
bitesize    dcstat     funcount        mountsnoop  profile     statsnoop tcli
st          zfsdist
btrfsdist   doc       funclatency    mysqld_qslower runqlat     syncsnoop  trac
e          zfsslower
btrfsslower execsnoop  gethostlatency offcpuftime slabratetop  tcpaccept ttys
noop
```

## 常用工具示例

### capable

capable检查Linux进程的security capabilities：

```
$ capable
TIME      UID      PID      COMM           CAP      NAME          AUDIT
22:11:23  114      2676     snmpd          12      CAP_NET_ADMIN    1
22:11:23  0        6990     run            24      CAP_SYS_RESOURCE 1
22:11:23  0        7003     chmod          3       CAP_FOWNER      1
22:11:23  0        7003     chmod          4       CAP_FSETID      1
22:11:23  0        7005     chmod          4       CAP_FSETID      1
22:11:23  0        7005     chmod          4       CAP_FSETID      1
22:11:23  0        7006     chown          4       CAP_FSETID      1
22:11:23  0        7006     chown          4       CAP_FSETID      1
22:11:23  0        6990     setuidgid     6       CAP_SETGID      1
22:11:23  0        6990     setuidgid     6       CAP_SETGID      1
22:11:23  0        6990     setuidgid     7       CAP_SETUID      1
22:11:24  0        7013     run             24      CAP_SYS_RESOURCE 1
22:11:24  0        7026     chmod          3       CAP_FOWNER      1
[...]
```

## tcpconnect

tcpconnect检查活跃的TCP连接，并输出源和目的地址：

```
$ ./tcpconnect
PID      COMM          IP_SADDR           DADDR           DPORt
2462     curl          4 192.168.1.99       74.125.23.138   80
```

## tcptop

tcptop统计TCP发送和接受流量：

```
$ ./tcptop -C 1 3
Tracing... Output every 1 secs. Hit Ctrl-C to end

08:06:45 loadavg: 0.04 0.01 0.00 2/174 3099

PID      COMM          LADDR           RADDR           RX_KB  TX_KB
1740     sshd          192.168.1.99:22    192.168.0.29:60315 0        0

08:06:46 loadavg: 0.04 0.01 0.00 2/174 3099

PID      COMM          LADDR           RADDR           RX_KB  TX_KB
1740     sshd          192.168.1.99:22    192.168.0.29:60315 0        0

08:06:47 loadavg: 0.04 0.01 0.00 2/174 3099

PID      COMM          LADDR           RADDR           RX_KB  TX_KB
1740     sshd          192.168.1.99:22    192.168.0.29:60315 0        0
```

## 扩展工具

基于eBPF和bcc，可以很方便的扩展功能。bcc目前支持以下事件

- `kprobe__kernel_function_name ( BPF.attach_kprobe() )`
- `kretprobe__kernel_function_name ( BPF.attach_kretprobe() )`
- `TRACEPOINT_PROBE(category, event)`，支持的`event`列表参见 `/sys/kernel/debug/tracing/events/category/event/format`
- `BPF.attach_uprobe()` 和 `BPF.attach_uretprobe()`
- 用户自定义探针(USDT) `USDT.enable_probe()`

## 简单示例

```
#!/usr/bin/env python
from __future__ import print_function
from bcc import BPF

text='int kprobe__sys_sync(void *ctx) { bpf_trace_printk("Hello, World!\\n"); return 0
; }'
prog=""""
int hello(void *ctx) {
    bpf_trace_printk("Hello, World!\\n");
    return 0;
}
"""

b = BPF(text=prog)
b.attach_kprobe(event="sys_clone", fn_name="hello")

print("%-18s %-16s %-6s %s" % ("TIME(s)", "COMM", "PID", "MESSAGE"))
while True:
    try:
        (task, pid, cpu, flags, ts, msg) = b.trace_fields()
    except ValueError:
        continue
    print("%-18.9f %-16s %-6d %s" % (ts, task, pid, msg))
```

## 使用 **BPF\_PERF\_OUTPUT**

```

from __future__ import print_function
from bcc import BPF
import ctypes as ct

# load BPF program
b = BPF(text=""""
struct data_t {
    u64 ts;
};

BPF_PERF_OUTPUT(events);

void kprobe__sys_sync(void *ctx) {
    struct data_t data = {};
    data.ts = bpf_ktime_get_ns() / 1000;
    events.perf_submit(ctx, &data, sizeof(data));
}
""")

class Data(ct.Structure):
    _fields_ = [
        ("ts", ct.c_ulonglong)
    ]

# header
print("%-18s %s" % ("TIME(s)", "CALL"))

# process event
def print_event(cpu, data, size):
    event = ct.cast(data, ct.POINTER(Data)).contents
    print("%-18.9f sync()" % (float(event.ts) / 1000000))

# loop with callback to print_event
b["events"].open_perf_buffer(print_event)
while True:
    b.kprobe_poll()

```

更多的示例参

考[https://github.com/iovisor/bcc/blob/master/docs/tutorial\\_bcc\\_python\\_developer.md](https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_python_developer.md)。

## 用户自定义探针示例

```

from __future__ import print_function
from bcc import BPF
from time import strftime
import ctypes as ct

# load BPF program
bpf_text = """
#include <uapi/linux/ptrace.h>

struct str_t {
    u64 pid;
    char str[80];
};

BPF_PERF_OUTPUT(events);

int printret(struct pt_regs *ctx) {
    struct str_t data  = {};
    u32 pid;
    if (!PT_REGS_RC(ctx))
        return 0;
    pid = bpf_get_current_pid_tgid();
    data.pid = pid;
    bpf_probe_read(&data.str, sizeof(data.str), (void *)PT_REGS_RC(ctx));
    events.perf_submit(ctx,&data,sizeof(data));

    return 0;
};
"""
STR_DATA = 80

class Data(ct.Structure):
    _fields_ = [
        ("pid", ct.c_ulonglong),
        ("str", ct.c_char * STR_DATA)
    ]

b = BPF(text=bpf_text)
b.attach_uretprobe(name="/bin/bash", sym="readline", fn_name="printret")

# header
print("%-9s %-6s %s" % ("TIME", "PID", "COMMAND"))

def print_event(cpu, data, size):
    event = ct.cast(data, ct.POINTER(Data)).contents
    print("%-9s %-6d %s" % (strftime("%H:%M:%S"), event.pid, event.str))

b["events"].open_perf_buffer(print_event)
while 1:
    b.kprobe_poll()

```

```
# ./bashreadline
TIME      PID      COMMAND
08:22:44  2070    ls /
08:22:56  2070    ping -c3 google.com

# ./gethostlatency
TIME      PID      COMM          LATms   HOST
08:23:26  3370    ping           2.00    google.com
08:23:37  3372    ping           56.00   baidu.com
```

## 参考文档

- <https://www.iovisor.org/>
- <https://www.iovisor.org/technology/ebpf>
- <https://www.iovisor.org/technology/xdp>
- <https://github.com/iovisor/bpf-docs>
- <https://www.kernel.org/doc/Documentation/networking/filter.txt>
- <https://events.linuxfoundation.org/sites/events/files/slides/iovisor-lc-bof-2016.pdf>
- <https://suchakra.wordpress.com/2015/08/12/bpf-internals-ii/>
- <https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/>
- <https://github.com/cilium/cilium>

# eBPF故障排查

## 内存限制

eBPF map使用固定的内存（locked memory），但默认非常小，可以通过调用 `setrlimit(2)` 来增大 `RLIMIT_MEMLOCK`。如果内存不足，`bpf_create_map` 会返回 `EPERM (Operation not permitted)` 错误。

## 开启 BPF JIT

开启方法为

```
$ sysctl net/core/bpf_jit_enable=1
net.core.bpf_jit_enable = 1
```

## ELF二进制文件

eBPF通过LLVM编译器生成的程序就是一个普通的ELF二进制文件，可以使用 `readelf` 或者 `llvm-objdump` 分析该文件，如

```
$ llvm-objdump -h xdp_ddos01_blacklist_kern.o

xdp_ddos01_blacklist_kern.o:      file format ELF64-unknown

Sections:
Idx Name      Size      Address          Type
 0           00000000 00000000000000000000
 1 .strtab    00000072 00000000000000000000
 2 .text      00000000 00000000000000000000 TEXT DATA
 3 xdp_prog   000001b8 00000000000000000000 TEXT DATA
 4 .relxdp_prog 00000020 00000000000000000000
 5 maps       00000028 00000000000000000000 DATA
 6 license    00000004 00000000000000000000 DATA
 7 .symtab    000000d8 00000000000000000000
```

## 提取eBPF-JIT代码

在调试eBPF程序时，有时需要提取eBPF-JIT代码

```
$ sysctl net.core.bpf_jit_enable=2
```

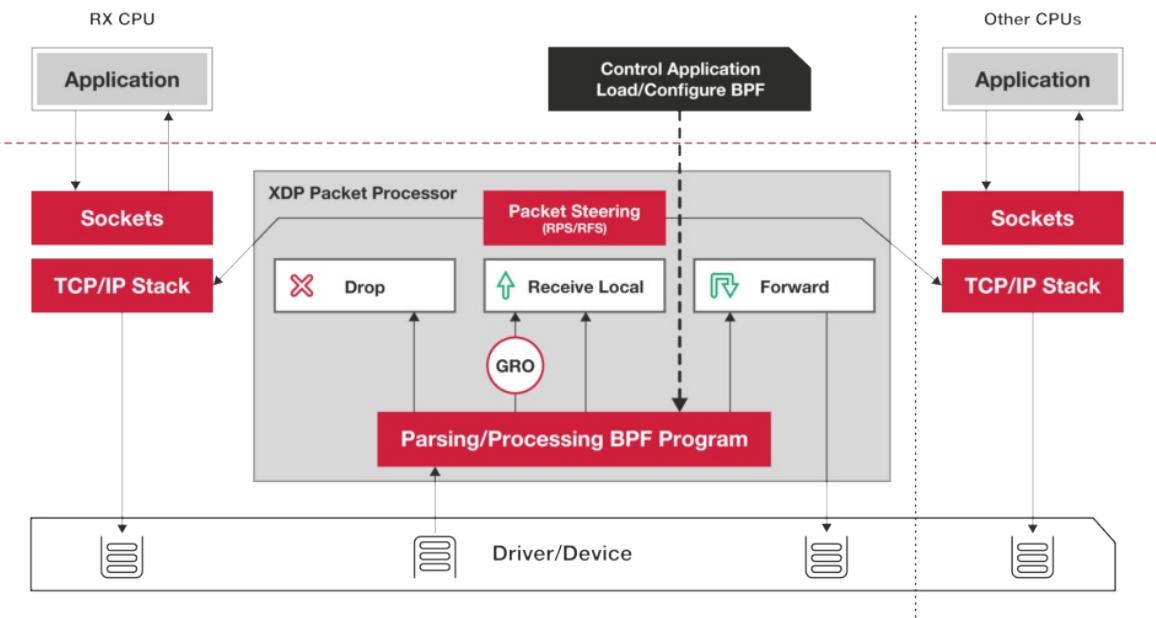
输出如下所示：

```
flen=55 proglen=335 pass=4 image=fffffffffa0006820 from=xdp_ddos01_blac pid=13333
JIT code: 00000000: 55 48 89 e5 48 81 ec 28 02 00 00 48 89 9d d8 fd
JIT code: 00000010: ff ff 4c 89 ad e0 fd ff ff 4c 89 b5 e8 fd ff ff
JIT code: 00000020: 4c 89 bd f0 fd ff ff 31 c0 48 89 85 f8 fd ff ff
JIT code: 00000030: bb 02 00 00 00 48 8b 77 08 48 8b 7f 00 48 89 fa
JIT code: 00000040: 48 83 c2 0e 48 39 f2 0f 87 e1 00 00 00 48 0f b6
JIT code: 00000050: 4f 0c 48 0f b6 57 0d 48 c1 e2 08 48 09 ca 48 89
JIT code: 00000060: d1 48 81 e1 ff 00 00 00 41 b8 06 00 00 00 49 39
JIT code: 00000070: c8 0f 87 b7 00 00 00 48 81 fa 88 a8 00 00 74 0e
JIT code: 00000080: b9 0e 00 00 00 48 81 fa 81 00 00 00 75 1a 48 89
JIT code: 00000090: fa 48 83 c2 12 48 39 f2 0f 87 90 00 00 00 b9 12
JIT code: 000000a0: 00 00 00 48 0f b7 57 10 bb 02 00 00 00 48 81 e2
JIT code: 000000b0: ff ff 00 00 48 83 fa 08 75 49 48 01 cf 31 db 48
JIT code: 000000c0: 89 fa 48 83 c2 14 48 39 f2 77 38 8b 7f 0c 89 7d
JIT code: 000000d0: fc 48 89 ee 48 83 c6 fc 48 bf 00 9c 24 5f 07 88
JIT code: 000000e0: ff ff e8 29 cd 13 e1 bb 02 00 00 00 48 83 f8 00
JIT code: 000000f0: 74 11 48 8b 78 00 48 83 c7 01 48 89 78 00 bb 01
JIT code: 00000100: 00 00 00 89 5d f8 48 89 ee 48 83 c6 f8 48 bf c0
JIT code: 00000110: 76 12 13 04 88 ff ff e8 f4 cc 13 e1 48 83 f8 00
JIT code: 00000120: 74 0c 48 8b 78 00 48 83 c7 01 48 89 78 00 48 89
JIT code: 00000130: d8 48 8b 9d d8 fd ff ff 4c 8b ad e0 fd ff ff 4c
JIT code: 00000140: 8b b5 e8 fd ff ff 4c 8b bd f0 fd ff ff c9 c3
```

其中，`proglen` 是opcode sequence的长度，`flen` 是bpf insns的个数。可以使用`bpf_jit_disasm` 工具来生成相关的opcodes。

# XDP

XDP（eXpress Data Path）为Linux内核提供了高性能、可编程的网络数据路径。由于网络包在还未进入网络协议栈之前就处理，它给Linux网络带来了巨大的性能提升（性能比DPDK还要高）。



XDP主要的特性包括

- 在网络协议栈前处理
- 无锁设计
- 批量I/O操作
- 轮询式
- 直接队列访问
- 不需要分配skbuff
- 支持网络卸载
- DDIO
- XDP程序快速执行并结束，没有循环
- Packeting steering

## 与DPDK对比

相对于DPDK，XDP具有以下优点

- 无需第三方代码库和许可

- 同时支持轮询式和中断式网络
- 无需分配大页
- 无需专用的CPU
- 无需定义新的安全网络模型

## 示例

- [Linux内核BPF示例](#)
- [prototype-kernel示例](#)
- [libbpf](#)

## 缺点

注意XDP的性能提升是有代价的，它牺牲了通用型和公平性

- XDP不提供缓存队列（qdisc），TX设备太慢时直接丢包，因而不要在RX比TX快的设备上使用XDP
- XDP程序是专用的，不具备网络协议栈的通用性

## 参考文档

- [Introduction to XDP](#)
- [Network Performance BoF](#)
- [XDP Introduction and Use-cases](#)
- [Linux Network Stack](#)
- [NetDev 1.2 video](#)

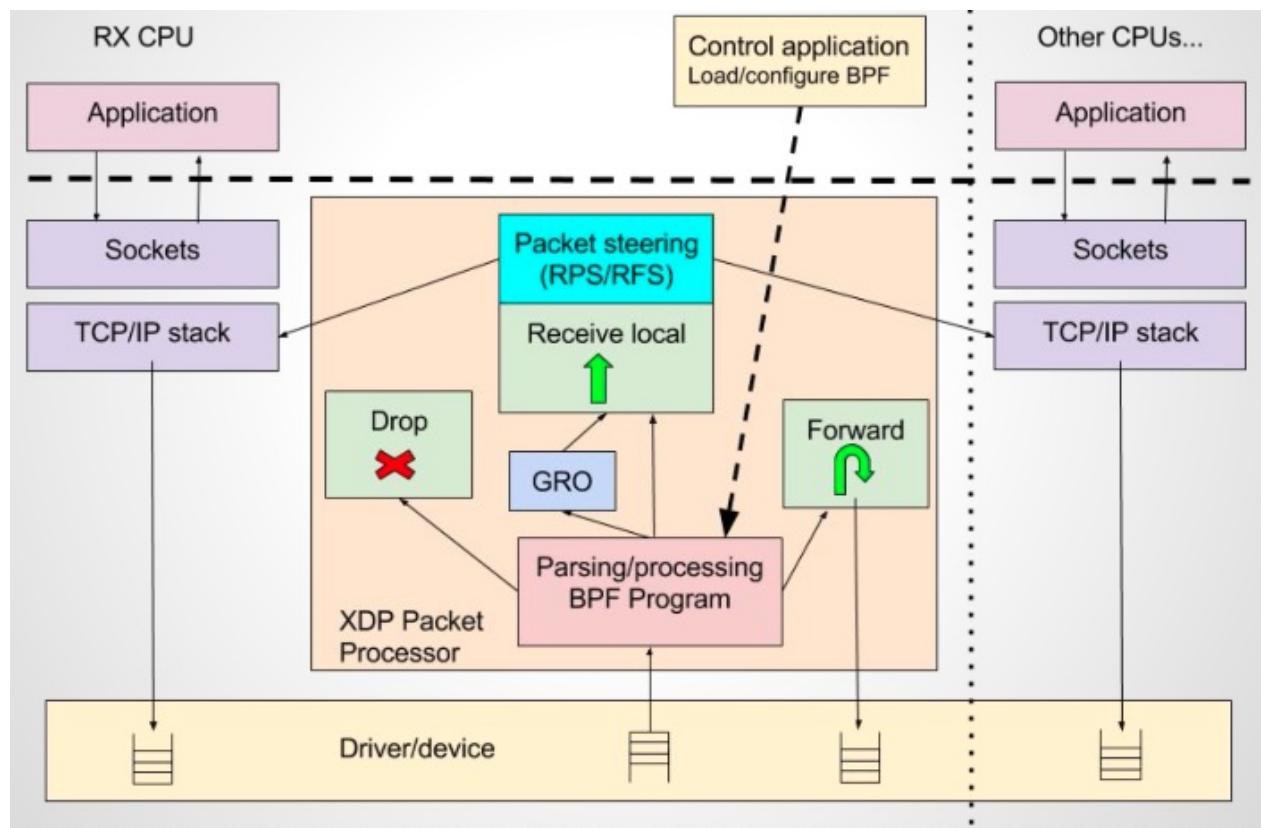
# XDP架构

XDP基于一系列的技术来实现高性能和可编程性，包括

- 基于eBPF
- Capabilities negotiation：通过协商确定网卡驱动支持的特性，XDP尽量利用新特性，但网卡驱动不需要支持所有的特性
- 在网络协议栈前处理
- 无锁设计
- 批量I/O操作
- 轮询式
- 直接队列访问
- 不需要分配skbuff
- 支持网络卸载
- DDIO
- XDP程序快速执行并结束，没有循环
- Packeting steering

## 包处理逻辑

如下图所示，基于内核的eBPF程序处理包，每个RX队列分配一个CPU，且以每个网络包一个Page（packet-page）的方式避免分配skbuff。



## XDP使用场景

XDP的使用场景包括

- DDoS防御
- 防火墙
- 基于 XDP\_TX 的负载均衡
- 网络统计
- 复杂网络采样
- 高速交易平台

## 网络常用工具

Linux网络常用工具介绍。

# tcpdump

注：本文大部分转自[细说tcpdump的妙用](#)，有删改。

tcpdump命令：

```
tcpdump -en -i p3p2 -vv      # show vlan
```

tcpdump选项可划分为四大类型：控制tcpdump程序行为，控制数据怎样显示，控制显示什么数据，以及过滤命令。

## 控制程序行为

这一类命令行选项影响程序行为，包括数据收集的方式。之前已介绍了两个例子：`-r`和`-w`。`-w`选项允许用户将输出重定向到一个文件，之后可通过`-r`选项将捕获数据显示出来。

如果用户知道需要捕获的报文数量或对于数量有一个上限，可使用`-c`选项。则当达到该数量时程序自动终止，而无需使用`Kill`命令或`Ctrl-C`。下例中，收集到100个报文之后tcpdump终止：

```
bsd1# tcpdump -c100
```

如果用户在多余一个网络接口上运行tcpdump，用户可以通过`-i`选项指定接口。在不确定的情况下，可使用`ifconfig -a`来检查哪一个接口可用及对应哪一个网络。例如，一台机器有两个C级接口，`xl0`接口IP地址`205.153.63.238`，`xl1`接口IP地址`205.153.61.178`。要捕捉`205.153.61.0`网络的数据流，使用以下命令：

```
bsd1# tcpdump -i xl1
```

没有指定接口时，tcpdump默认为最低编号接口。

`-p`选项将网卡接口设置为非混杂模式。这一选项理论上将限制为捕获接口上的正常数据流——来自或发往主机，多播数据，以及广播数据。

`-s`选项控制数据的截取长度。通常，tcpdump默认为一最大字节数量并只会从单一报文中截取到该数量长度。实际字节数取决于操作系统的设备驱动。通过默认值来截取合适的报文头，而舍弃不必要的报文数据。

如果用户需截取更多数据，通过`-s`选项来指定字节数。也可以用`-s`来减少截取字节数。对于少于或等于200字节的报文，以下命令会截取完整报文：

```
bsd1# tcpdump -s200
```

更长的报文会被缩短为200字节。

## 控制信息如何显示

`-a`，`-n`，`-N` 和 `-f` 选项决定了地址信息是如何显示的。`-a`选项强制将网络地址显示为名称，`-n`阻止将地址显示为名字，`-N`阻止将域名转换。`-f`选项阻止远端名称解析。下例中，从 `sloan.lander.edu (205.153.63.30)` 远程站点，分别不加选项，`-a`，`-n`，`-N`，`-f`。（选项`-c1`限制抓取1个报文）

```
bsd1# tcpdump -c1 host 192.31.7.130
tcpdump: listening on x10
14:16:35.897342 sloan.lander.edu > cio-sys.cisco.com: icmp: echo request
bsd1# tcpdump -c1 -a host 192.31.7.130
tcpdump: listening on x10
14:16:14.567917 sloan.lander.edu > cio-sys.cisco.com: icmp: echo request
bsd1# tcpdump -c1 -n host 192.31.7.130
tcpdump: listening on x10
14:17:09.737597 205.153.63.30 > 192.31.7.130: icmp: echo request
bsd1# tcpdump -c1 -N host 192.31.7.130
tcpdump: listening on x10
14:17:28.891045 sloan > cio-sys: icmp: echo request
bsd1# tcpdump -c1 -f host 192.31.7.130
tcpdump: listening on x10
14:17:49.274907 sloan.lander.edu > 192.31.7.130: icmp: echo request
```

默认为`-a`选项。

`-t` 和 `-tt` 选项控制时间戳的打印。`-t`选项不显示时间戳而`-tt`选项显示无格式的时间戳。以下命令显示了`tcpdump`命令无选项，`-t`选项，`-tt`选项的同一报文：

```
12:36:54.772066 sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 86
47 (DF)
sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 8647 (DF)
934303014.772066 sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 8
647 (DF)
```

## 控制显示什么数据

可以通过`-v`和`-vv`选项来打印更多详细信息。例如，`-v`选项将会打印TTL字段。要显示较少信息，使用`-q`，或`quiet`选项。一下为同一报文分别使用`-q`选项，无选项，`-v`选项，和`-vv`选项的输出。

```
12:36:54.772066 sloan.lander.edu.1174 > 205.153.63.238.telnet: tcp 0 (DF)
12:36:54.772066 sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 86
47 (DF)
12:36:54.772066 sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 86
47 (DF) (ttl 128, id 45836)
12:36:54.772066 sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 86
47 (DF) (ttl 128, id 45836)
```

-e 选项用于显示链路层头信息。上例中-e选项的输出为：

```
12:36:54.772066 0:10:5a:a1:e9:8 0:10:5a:e3:37:c ip 60:
sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 8647 (DF)
```

0:10:5a:a1:e9:8是sloan.lander.edu中3Com卡的以太网地址，0:10:5a:e3:37:c是205.153.63.238中3Com卡的以太网地址。

-x 选项将报文以十六进制形式dump出来，排除了链路层报文头。-x和-vv选项报文显示如下：

```
13:57:12.719718 bsd1.lander.edu.1657 > 205.153.60.5.domain: 11587+ A? www.microsoft.co
m. (35) (ttl 64, id 41353)
4500 003f a189 0000 4011 c43a cd99 3db2
cd99 3c05 0679 0035 002b 06d9 2d43 0100
0001 0000 0000 0000 0377 7777 096d 6963
726f 736f 6674 0363 6f6d 0000 0100 01
```

## 过滤

要有效地使用tcpdump，掌握过滤器非常必要的。过滤允许用户指定想要抓取的数据流，从而用户可以专注于感兴趣的数据。此外，ethereal这样的工具使用tcpdump过滤语法来抓取数据流。

如果用户很清楚对何种数据流不感兴趣，可以将这部分数据排除在外。如果用户不确定需要什么数据，可以将源数据收集到文件之后在读取时应用过滤器。实际应用中，需要经常在两种方式之间转换。

简单的过滤器是加在命令行之后的关键字。但是，复杂的命令是由逻辑和关系运算符构成的。对于这样的情况，通常最好用-F选项将过滤器存储在文件中。例如，假设testfilter是一个包含过滤主机205.153.63.30的文本文件，之后输入tcpdump -Ftestfilter等效于输入命令tcpdump host 205.153.63.30。通常，这一功能只在复杂过滤器时使用。但是，同一命令中命令行过滤器和文件过滤器不能混用。

## 地址过滤

过滤器可以按照地址选择数据流。例如，考虑如下命令：

```
bsd1# tcpdump host 205.153.63.30
```

该命令抓取所有来自以及发往IP地址205.153.63.30的主机。主机可以通过名称或IP地址来选定。虽然指定的是IP地址，但抓取数据流并不限于IP数据流，实际上，过滤器也会抓到ARP数据流。限定仅抓取特定协议的数据流要求更复杂的过滤器。

有若干种方式可以指定和限制地址，下例是通过机器的以太网地址来选择数据流：

```
bsd1# tcpdump ether host 0:10:5a:e3:37:c
```

数据流可进一步限制为单向，分别用**src**或**dst**指定数据流的来源或目的地。下例显示了发送到主机205.153.63.30的数据流：

```
bsd1# tcpdump dst 205.153.63.30
```

注意到本例中**host**被省略了。在某些例子中省略是没问题的，但添加这些关键字通常更安全些。

广播和多播数据相应可以使用**broadcast**和**multicast**。由于多播和广播数据流在链路层和网络层所指定的数据流是不同的，所以这两种过滤器各有两种形式。过滤器**ether multicast**抓取以太网多播地址的数据流，**ip multicast**抓取IP多播地址数据流。广播数据流也是类似的方法。注意多播过滤器也会抓到广播数据流。

除了抓取特定主机以外，还可以抓取特定网络。例如，以下命令限制抓取来自或发往205.153.60.0的报文：

```
bsd1# tcpdump net 205.153.60
```

以下命令也可以做同样的事情：

```
bsd1# tcpdump net 205.153.60.0 mask 255.255.255.0
```

而以下命令由于最后的.0就无法正常工作：

```
bsd1# tcpdump net 205.153.60.0
```

## 协议及端口过滤

限制抓取指定协议如IP，Appletalk或TCP。还可以限制建立在这些协议之上的服务，如DNS或RIP。这类抓取可以通过三种方式进行：使用tcpdump关键字，通过协议关键字proto，或通过服务使用port关键字。

一些协议名能够被tcpdump识别到因此可通过关键字来指定。以下命令限制抓取IP数据流：

```
bsd1# tcpdump ip
```

当然，IP数据流包括TCP数据流，UDP数据流，等等。

如果仅抓取TCP数据流，可以使用：

```
bsd1# tcpdump tcp
```

tcpdump可识别的关键字包括ip, igmp, tcp, udp, and icmp。

有很多传输层服务没有可以识别的关键字。在这种情况下，可以使用关键字proto或ip proto加上/etc/protocols能够找到的协议名或相应的协议编号。例如，以下两种方式都会查找OSPF报文：

```
bsd1# tcpdump ip proto ospf  
bsd1# tcpdump ip proto 89
```

内嵌的关键字可能会造成问题。下面的例子中，无法使用tcp关键字，或必须使用数字。例如，下面的例子是正常工作的：

```
bsd#1 tcpdump ip proto 6
```

另一方面，不能使用proto加上tcp:

```
bsd#1 tcpdump ip proto tcp
```

会产生问题。

对于更高层级的建立于底层协议之上的服务，必须使用关键字port。以下两者会采集DNS数据流：

```
bsd#1 tcpdump port domain  
bds#1 tcpdump port 53
```

第一条命令中，关键字domain能够通过查找/etc/services来解析。在传输层协议有歧义的情况下，可以将端口限制为指定协议。考虑如下命令：

```
bsd#1 tcpdump udp port domain
```

这会抓取使用UDP的DNS名查找但不包括使用TCP的DNS zone传输数据。而之前的两条命令会同时抓取这两种数据。

## 报文特征

过滤器也可以基于报文特征比如报文长度或特定字段的内容，过滤器必须包含关系运算符。要指定长度，使用关键字less或greater。如下例所示：

```
bsd1# tcpdump greater 200
```

该命令收集长度大于200字节的报文。

根据报文内容过滤更加复杂，因为用户必须理解报文头的结构。但是尽管如此，或者说正因如此，这一方式能够使用户最大限度的控制抓取的数据。

一般使用语法 proto [ expr : size ]。字段proto指定要查看的报文头——ip则查看IP头，tcp则查看TCP头，以此类推。expr字段给出从报文头索引0开始的位移。即：报文头的第一个字节为0，第二字节为1，以此类推。size字段是可选的，指定需要使用的字节数，1，2或4。

```
bsd1# tcpdump "ip[9] = 6"
```

查看第十字节的IP头，协议值为6。注意这里必须使用引号。撇号或引号都可以，但反引号将无法正常工作。

```
bsd1# tcpdump tcp
```

也是等效的，因为TCP协议编号为6。

这一方式常常作为掩码来选择特定比特位。值可以是十六进制。可通过语法&加上比特掩码来指定。下例提取从以太网头第一字节开始（即目的地址第一字节），提取低阶比特位，并确保该位不为0：

```
bsd1# tcpdump 'ether[0] & 1 != 0'
```

该条件会选取广播和多播报文。

以上两个例子都有更好的方法来匹配报文。作为一个更实际的例子，考虑以下命令：

```
bsd1# tcpdump "tcp[13] & 0x03 != 0"
```

该过滤器跳过TCP头的13个字节，提取flag字节。掩码0x03选择第一和第二比特位，即FIN和SYN位。如果其中一位不为0则报文被抓取。此命令会抓取TCP连接建立及关闭报文。

不要将逻辑运算符与关系运算符混淆。比如想 `tcp src port > 23` 这样的表达式就无法正常工作。因为 `tcp src port` 表达式返回值为 `true` 或 `false`，而不是一个数值，所以无法与数值进行比较。如果需要查找端口号大于23的所有TCP数据流，必须从报文头提取端口字段，使用表达式 `tcp[0:2] & 0xffff > 0x0017`。

## 参考文档

- 细说tcpdump的妙用

# scapy

scapy是一个强大的python网络数据包处理库，它可以生成或解码网络协议数据包，可以用来端口扫描、探测、网络测试等。

## scapy安装

```
pip install scapy
```

## 简单使用

scapy提供了一个简单的交互式界面，直接运行scapy命令即可进入。当然，也可以在python交互式命令行中导入scapy包进入

```
from scapy.all import *
```

查看所有支持的协议和预制工具：

```
ls()  
lsc()
```

## 构造IP数据包

```
pkt=IP(dst="8.8.8.8")  
pkt.show()  
print pkt.dst # 8.8.8.8  
str(pkt) # hex string  
hexdump(pkt) # hex dump
```

## 输出HEX格式的数据包

```
import binascii  
from scapy.all import *  
a=Ether(dst="02:ac:10:ff:00:22",src="02:ac:10:ff:00:11")/IP(dst="172.16.255.22",src="1  
72.16.255.11", ttl=10)/ICMP()  
print binascii.hexlify(str(a))
```

TCP/IP协议的四层模型都可以分别构造，并通过 / 连接

```
Ether()/IP()/TCP()
IP()/TCP()
IP()/TCP()/"GET / HTTP/1.0\r\n\r\n"
Ether()/IP()/IP()/UDP()
Ether()/IP(dst="www.slashdot.org")/TCP()/"GET /index.html HTTP/1.0 \n\n"
```

从PCAP文件读入数据

```
a = rdpcap("test.cap")
```

发送数据包

```
# 三层发送，不接收
send(IP(dst="8.8.8.8")/ICMP())
# 二层发送，不接收
sendp(Ether()/IP(dst="8.8.8.8", ttl=10)/ICMP())
# 三层发送并接收
# 二层可以用srp, srp1和srploop
result, unanswered = sr(IP(dst="8.8.8.8")/ICMP())
# 发送并只接收第一个包
sr1(IP(dst="8.8.8.8")/ICMP())
# 发送多个数据包
result=srloop(IP(dst="8.8.8.8")/ICMP(), inter=1, count=2)
```

嗅探数据包

```
sniff(filter="icmp", count=3, timeout=5, prn=lambda x:x.summary())
a=sniff(filter="tcp and ( port 25 or port 110 )",
prn=lambda x: x.sprintf("%IP.src%:%TCP.sport% -> %IP.dst%:%TCP.dport% %2s,TCP.flags% : %TCP.payload%"))
```

SYN扫描

```
sr1(IP(dst="172.217.24.14")/TCP(dport=80, flags="S"))
ans,unans = sr(IP(dst=["192.168.1.1", "yahoo.com", "slashdot.org"])/TCP(dport=[22, 80, 443 ], flags="S"))
```

TCP traceroute

```
ans,unans=sr(IP(dst="www.baidu.com",ttl=(2,25),id=RandShort())/TCP(flags=0x2),timeout=50)
for snd,rcv in ans:
    print snd.ttl,rcv.src, isinstance(rcv.payload,TCP)
```

## ARP Ping

```
ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst="192.168.1.0/24"),timeout=2)
ans.summary(lambda (s,r): r.sprintf("%Ether.src% %ARP.psrc%"))
```

## ICMP Ping

```
ans,unans=sr(IP(dst="192.168.1.1-254")/ICMP())
ans.summary(lambda (s,r): r.sprintf("%IP.src% is alive"))
```

## TCP Ping

```
ans,unans=sr( IP(dst="192.168.1.*")/TCP(dport=80,flags="S"))
ans.summary( lambda(s,r) : r.sprintf("%IP.src% is alive") )
```

# 内核中的网络参数

## nf\_conntrack

`nf_conntrack` 是Linux内核连接跟踪的模块，常用在`iptables`中，比如

```
-A INPUT -m state --state RELATED,ESTABLISHED -j RETURN
-A INPUT -m state --state INVALID -j DROP
```

可以通过 `cat /proc/net/nf_conntrack` 来查看当前跟踪的连接信息，这些信息以哈希形式（用链地址法处理冲突）存在内存中，并且每条记录大约占300B空间。

与 `nf_conntrack` 相关的内核参数有三个：

- `nf_conntrack_max`：连接跟踪表的大小，建议根据内存计算该值 `CONNTRACK_MAX = RAMSIZE (in bytes) / 16384 / (x / 32)`，并满足 `nf_conntrack_max=4*nf_conntrack_buckets`，默认 `262144`
- `nf_conntrack_buckets`：哈希表的大小，(`nf_conntrack_max/nf_conntrack_buckets` 就是每条哈希记录链表的长度)，默认 `65536`
- `nf_conntrack_tcp_timeout_established`：tcp会话的超时时间，默认是 `432000 (5天)`

比如，对64G内存的机器，推荐配置：

```
net.netfilter.nf_conntrack_max=4194304
net.netfilter.nf_conntrack_tcp_timeout_established=300
net.netfilter.nf_conntrack_buckets=1048576
```

## bridge-nf

`bridge-nf`使得`netfilter`可以对Linux网桥上的IPv4/ARP/IPv6包过滤。比如，设置 `net.bridge.bridge-nf-call-iptables=1` 后，二层的网桥在转发包时也会被`iptables`的 FORWARD规则所过滤，这样有时会出现L3层的`iptables rules`去过滤L2的帧的问题（见[这里](#)）。

常用的选项包括

- `net.bridge.bridge-nf-call-arptables`：是否在`arptables`的FORWARD中过滤网桥的ARP包
- `net.bridge.bridge-nf-call-ip6tables`：是否在`ip6tables`链中过滤IPv6包
- `net.bridge.bridge-nf-call-iptables`：是否在`iptables`链中过滤IPv4包

- `net.bridge.bridge-nf-filter-vlan-tagged`：是否在 `iptables/arptables` 中过滤打了 `vlan` 标签的包

当然，也可以通过 `/sys/devices/virtual/net/<bridge-name>/bridge/nf_call_iptables` 来设置，但要注意内核是取两者中大的生效。

有时，可能只希望部分网桥禁止 `bridge-nf`，而其他网桥都开启（比如 CNI 网络插件中一般要求 `bridge-nf-call-iptables` 选项开启，而有时又希望禁止某个网桥的 `bridge-nf`），这时可以改用 `iptables` 的方法：

```
iptables -t raw -I PREROUTING -i <bridge-name> -j NOTRACK
```

## 反向路径过滤

反向路径过滤可用于防止数据包从一接口传入，又从另一不同的接口传出（这有时被称为“非对称路由”）。除非必要，否则最好将其关闭，因为它可防止来自子网络的用户采用 IP 地址欺骗手段，并减少 DDoS（分布式拒绝服务）攻击的机会。

通过 `rp_filter` 选项启用反向路径过滤，比如 `sysctl -w net.ipv4.conf.default.rp_filter=INTEGER`。支持三种选项：

- 0 ——未进行源验证。
- 1 ——处于如 RFC3704 所定义的严格模式。
- 2 ——处于如 RFC3704 所定义的松散模式。

可以通过 `net.ipv4.interface.rp_filter` 可实现对每一网络接口设置的覆盖。

## TCP 相关

参数	描述	默认值	优化值
<code>net.core.rmem_default</code>	默认的 TCP 数据接收窗口大小（字节）	212992	
<code>net.core.rmem_max</code>	最大的 TCP 数据接收窗口（字节）。	212992	
<code>net.core.wmem_default</code>	默认的 TCP 数据发送窗口大小（字节）。	212992	
<code>net.core.wmem_max</code>	最大的 TCP 数据发送窗口（字节）。	212992	
	在每个网络接口接收数据包的速率比内核处理这些包的速率		

	快时，允许送到队列的数据包的最大数目。		
net.core.somaxconn	定义了系统中每一个端口最大的监听队列的长度，这是个全局的参数。	128	2048
net.core.optmem_max	表示每个套接字所允许的最大缓冲区的大小。	20480	81920
net.ipv4.tcp_mem	确定TCP栈应该如何反映内存使用，每个值的单位都是内存页（通常是4KB）。第一个值是内存使用的下限；第二个值是内存压力模式开始对缓冲区使用应用压力的上限；第三个值是内存使用的上限。在这个层次上可以将报文丢弃，从而减少对内存的使用。对于较大的BDP可以增大这些值（注意，其单位是内存页而不是字节）。	5814 7754 11628	
net.ipv4.tcp_rmem	为自动调优定义socket使用的内存。第一个值是为socket接收缓冲区分配的最少字节数；第二个值是默认值（该值会被 rmem_default 覆盖），缓冲区在系统负载不重的情况下可以增长到这个值；第三个值是接收缓冲区空间的最大字节数（该值会被 rmem_max 覆盖）。	4096 87380 3970528	
net.ipv4.tcp_wmem	为自动调优定义socket使用的内存。第一个值是为socket发送缓冲区分配的最少字节数；第二个值是默认值（该值会被 wmem_default 覆盖），缓冲区在系统负载不重的情况下可以增长到这个值；第三个值是发送缓冲区空间的最大字节数（该值会被 wmem_max 覆盖）。	4096 16384 3970528	
net.ipv4.tcp_keepalive_time	TCP发送keepalive探测消息的间隔时间（秒），用于确认TCP连接是否有效。	7200	1800
net.ipv4.tcp_keepalive_intvl	探测消息未获得响应时，重发该消息的间隔时间（秒）	75	30
net.ipv4.tcp_keepalive_probes	在认定TCP连接失效之前，最多发送多少个keepalive探测消息。	9	3

net.ipv4.tcp_sack	启用有选择的应答（1表示启用），通过有选择地应答乱序接收到的报文来提高性能，让发送者只发送丢失的报文段，（对于广域网通信来说）这个选项应该启用，但是会增加对CPU的占用。	1	1
net.ipv4.tcp_fack	启用转发应答，可以进行有选择应答（SACK）从而减少拥塞情况的发生，这个选项也应该启用。	1	1
net.ipv4.tcp_timestamps	TCP时间戳（会在TCP包头增加12个字节），以一种比重发超时更精确的方法（参考RFC 1323）来启用对RTT的计算，为实现更好的性能应该启用这个选项。	1	1
net.ipv4.tcp_window_scaling	启用RFC 1323定义的window scaling，要支持超过64KB的TCP窗口，必须启用该值（1表示启用），TCP窗口最大至1GB，TCP连接双方都启用时才生效。	1	1
net.ipv4.tcp_syncookies	表示是否打开TCP同步标签（syncookie），内核必须打开了 CONFIG_SYN_COOKIES 项进行编译，同步标签可以防止一个套接字在有过多试图连接到达时引起过载。	1	1
net.ipv4.tcp_tw_reuse	表示是否允许将处于TIME-WAIT状态的socket（TIME-WAIT的端口）用于新的TCP连接。	0	1
net.ipv4.tcp_tw_recycle	能够更快地回收TIME-WAIT套接字。	0	1
net.ipv4.tcp_fin_timeout	对于本端断开的socket连接，TCP保持在FIN-WAIT-2状态的时间（秒）。对方可能会断开连接或一直不结束连接或不可预料的进程死亡。	60	30
net.ipv4.ip_local_port_range	表示TCP/UDP协议允许使用的本地端口号	32768 60999	1024 65000
net.ipv4.tcp_max_syn_backlog	对于还未获得对方确认的连接请求，可保存在队列中的最大数目。如果服务器经常出现过载，可以尝试增加这个数字。	128	

net.ipv4.tcp_low_latency	允许TCP/IP栈适应在高吞吐量情况下低延时的情况，这个选项应该禁用。	0	0
--------------------------	-------------------------------------	---	---

## ARP相关

### ARP回收

- `gc_stale_time` 每次检查neighbour记录的有效性的周期。当neighbour记录失效时，将在给它发送数据前再解析一次。缺省值是60秒。
- `gc_thresh1` 存在于ARP高速缓存中的最少记录数，如果少于这个数，垃圾收集器将不会运行。缺省值是128。
- `gc_thresh2` 存在 ARP 高速缓存中的最多的记录软限制。垃圾收集器在开始收集前，允许记录数超过这个数字 5 秒。缺省值是 512。
- `gc_thresh3` 保存在 ARP 高速缓存中的最多记录的硬限制，一旦高速缓存中的数目高于此，垃圾收集器将马上运行。缺省值是1024。

比如可以增大为

```
net.ipv4.neigh.default.gc_thresh1=1024
net.ipv4.neigh.default.gc_thresh2=4096
net.ipv4.neigh.default.gc_thresh3=8192
```

### ARP过滤

#### arp\_filter - BOOLEAN

1 - Allows you to have multiple network interfaces on the same subnet, and have the ARPs for each interface be answered based on whether or not the kernel would route a packet from the ARP'd IP out that interface (therefore you must use source based routing for this to work). In other words it allows control of which cards (usually 1) will respond to an arp request.

0 - (default) The kernel can respond to arp requests with addresses from other interfaces. This may seem wrong but it usually makes sense, because it increases the chance of successful communication. IP addresses are owned by the complete host on Linux, not by particular interfaces. Only for more complex setups like load-balancing, does this behaviour cause problems.

arp\_filter for the interface will be enabled if at least one of conf/{all,interface}/arp\_filter is set to TRUE, it will be disabled otherwise

## arp\_announce - INTEGER

Define different restriction levels for announcing the local source IP address from IP packets in ARP requests sent on interface:

0 - (default) Use any local address, configured on any interface  
1 - Try to avoid local addresses that are not in the target's subnet for this interface. This mode is useful when target hosts reachable via this interface require the source IP address in ARP requests to be part of their logical network configured on the receiving interface. When we generate the request we will check all our subnets that include the target IP and will preserve the source address if it is from such subnet. If there is no such subnet we select source address according to the rules for level 2.

2 - Always use the best local address for this target.  
In this mode we ignore the source address in the IP packet and try to select local address that we prefer for talks with the target host. Such local address is selected by looking for primary IP addresses on all our subnets on the outgoing interface that include the target IP address. If no suitable local address is found we select the first local address we have on the outgoing interface or on all other interfaces, with the hope we will receive reply for our request and even sometimes no matter the source IP address we announce.

The max value from conf/{all,interface}/arp\_announce is used.

Increasing the restriction level gives more chance for receiving answer from the resolved target while decreasing the level announces more valid sender's information.

## arp\_ignore - INTEGER

Define different modes for sending replies in response to received ARP requests that resolve local target IP addresses:

0 - (default): reply for any local target IP address, configured on any interface  
1 - reply only if the target IP address is local address configured on the incoming interface  
2 - reply only if the target IP address is local address configured on the incoming interface and both with the sender's IP address are part from same subnet on this interface  
3 - do not reply for local addresses configured with scope host, only resolutions for global and link addresses are replied  
4-7 - reserved  
8 - do not reply for all local addresses

The max value from conf/{all,interface}/arp\_ignore is used when ARP request is received on the {interface}

## arp\_notify - BOOLEAN

```
Define mode for notification of address and device changes.  
0 - (default): do nothing  
1 - Generate gratuitous arp requests when device is brought up  
      or hardware address changes.
```

## arp\_accept - BOOLEAN

```
Define behavior for gratuitous ARP frames who's IP is not  
already present in the ARP table:  
0 - don't create new entries in the ARP table  
1 - create new entries in the ARP table
```

Both replies and requests type gratuitous arp will trigger the ARP table to be updated, if this setting is on.

If the ARP table already contains the IP address of the gratuitous arp frame, the arp table will be updated regardless if this setting is on or off.

## 参考文档

- [Linux Kernel ip sysctl documentation](#)

# Open vSwitch

## OVS安装

### CentOS

```
yum install centos-release-openstack-newton  
yum install openvswitch  
systemctl enable openvswitch  
systemctl start openvswitch
```

如果想要安装master版本，可以使用<https://copr.fedorainfracloud.org/coprs/leifmadsen/ovs-master/>的BUILD：

```
wget -o /etc/yum.repos.d/ovs-master.repo https://copr.fedorainfracloud.org/coprs/leifmadsen/ovs-master/repo/epel-7/leifmadsen-ovs-master-epel-7.repo  
yum install openvswitch openvswitch-ovn-*
```

## OVS常用命令参考

### 如何添加bridge和port

```
ovs-vsctl add-br br0  
ovs-vsctl del-br br0  
ovs-vsctl list-br  
ovs-vsctl add-port br0 eth0  
ovs-vsctl set port eth0 tag=1 #vlan id  
ovs-vsctl del-port br0 eth0  
ovs-vsctl list-ports br0  
ovs-vsctl show
```

### 给OVS端口配置IP

```
ovs-vsctl add-port br-ex port tag=10 -- set Interface port type=internal # default is access  
ifconfig port 192.168.100.1
```

### 如何配置流镜像

## 4. Open vSwitch

```
ovs-vsctl -- set Bridge br-int mirrors=@m -- --id=@tap6a094914-cd get Port tap6a094914  
-cd -- --id=@tap73e945b4-79 get Port tap73e945b4-79 -- --id=@tapa6cd1168-a2 get Port t  
apa6cd1168-a2 -- --id=@m create Mirror name=mymirror select-dst-port=@tap6a094914-cd,@  
tap73e945b4-79 select-src-port=@tap6a094914-cd,@tap73e945b4-79 output-port=@tapa6cd116  
8-a2  
  
# clear  
ovs-vsctl remove Bridge br0 mirrors mymirror  
ovs-vsctl clear Bridge br-int mirrors
```

利用**mirror**特性对**ovs**端口**patch-tun**抓包

```
ip link add name snooper0 type dummy  
ip link set dev snooper0 up  
ovs-vsctl add-port br-int snooper0  
ovs-vsctl -- set Bridge br-int mirrors=@m \  
-- --id=@snooper0 get Port snooper0 \  
-- --id=@patch-tun get Port patch-tun \  
-- --id=@m create Mirror name=mymirror select-dst-port=@patch-tun \  
select-src-port=@patch-tun output-port=@snooper0  
  
# capture  
tcpdump -i snooper0  
  
# clear  
ovs-vsctl clear Bridge br-int mirrors  
ip link delete dev snooper0
```

如何配置**QOS**，比如队列和限速

```
# egress
$ ovs-vsctl -- \
add-br br0 -- \
add-port br0 eth0 -- \
add-port br0 vif1.0 -- set interface vif1.0 ofport_request=5 -- \
add-port br0 vif2.0 -- set interface vif2.0 ofport_request=6 -- \
set port eth0 qos=@newqos -- \
--id=@newqos create qos type=linux-htb \
    other-config:max-rate=1000000000 \
    queues:123=@vif10queue \
    queues:234=@vif20queue -- \
--id=@vif10queue create queue other-config:max-rate=10000000 -- \
--id=@vif20queue create queue other-config:max-rate=20000000
$ ovs-ofctl add-flow br0 in_port=5,actions=set_queue:123,normal
$ ovs-ofctl add-flow br0 in_port=6,actions=set_queue:234,normal

# ingress
ovs-vsctl set interface vif1.0 ingress_policing_rate=10000
ovs-vsctl set interface vif1.0 ingress_policing_burst=8000

# clear
ovs-vsctl clear Port vif1.0 qos
ovs-vsctl list qos
ovs-vsctl destroy qos _uuid
ovs-vsctl list qos
ovs-vsctl destroy queue _uuid
```

### 如何配置流监控sflow

```
ovs-vsctl -- --id=@s create sFlow agent=vif1.0 target=\"10.0.0.1:6343\" header=128 sampling=64 polling=10 -- set Bridge br-int sflow=@s
ovs-vsctl -- clear Bridge br-int sflow
```

### 如何配置流规则

```
ovs-ofctl add-flow br-int idle_timeout=0,in_port=2,dl_type=0x0800,dl_src=00:88:77:66:55:44,dl_dst=11:22:33:44:55:66,nw_src=1.2.3.4,nw_dst=5.6.7.8,nw_proto=1,tp_src=1,tp_dst=2,actions=drop
ovs-ofctl del-flows br-int in_port=2 //in_port=2的所有流规则被删除
ovs-ofctl dump-ports br-int
ovs-ofctl dump-flows br-int
ovs-ofctl show br-int //查看端口号
```

- 支持字段还有nw\_tos,nw\_ecn,nw\_ttl,dl\_vlan,dl\_vlan\_pcp,ip\_frag ,arp\_sha,arp\_tha,ipv6\_src,ipv6\_dst等;
- 支持流动作还有output : port , mod\_dl\_src/mod\_dl\_dst, set field等;

### 如何查看OVS的配置

```
ovs-vsctl list/set/get/add/remove/clear/destroy table record column [VALUE]
```

其中，TABLE名支持

bridge,controller,interface,mirror,netflow,open\_vswitch,port,qos,queue,ssl,sflow

### 配置VXLAN/GRE

```
ovs-vsctl add-port br-ex port -- set interface port type=vxlan options:remote_ip=192.168.100.3  
ovs-vsctl add-port br-ex port -- set Interface port type=gre options:remote_ip=192.168.100.3  
ovs-vsctl set interface vxlan type=vxlan option:remote_ip=140.113.215.200 option:key=f  
low ofport_request=9
```

### 显示并学习MAC

```
ovs-appctl fdb/show br-ex
```

### 设置控制器地址

```
ovs-vsctl set-controller br-ex tcp:192.168.100.1:6633  
ovs-vsctl get-controller br0
```

## 流表管理

### 流规则组成

每条流规则由一系列字段组成，分为基本字段、条件字段和动作字段三部分：

- 基本字段包括生效时间duration\_sec、所属表项table\_id、优先级priority、处理的数据包数n\_packets，空闲超时时间idle\_timeout等，空闲超时时间idle\_timeout以秒为单位，超过设置的空闲超时时间后该流规则将被自动删除，空闲超时时间设置为0表示该流规则永不过期，idle\_timeout将不包含于ovs-ofctl dump-flows brname的输出中。
- 条件字段包括输入端口号in\_port、源目的mac地址dl\_src/dl\_dst、源目的ip地址nw\_src/nw\_dst、数据包类型dl\_type、网络层协议类型nw\_proto等，可以为这些字段的任意组合，但在网络分层结构中底层的字段未给出确定值时上层的字段不允许给确定值，即一条流规则中允许底层协议字段指定为确定值，高层协议字段指定为通配符(不指定即为匹配任何值)，而不允许高层协议字段指定为确定值，而底层协议字段却为通配符(不指定即为匹配任何值)，否则，ovs-vswitchd中的流规则将全部丢失，网络无法连接。
- 动作字段包括正常转发normal、定向到某交换机端口output : port、丢弃drop、更改源目的mac地址mod\_dl\_src/mod\_dl\_dst等，一条流规则可有多个动作，动作执行按指定的先

## 4. Open vSwitch

后顺序依次完成。

流规则中可包含通配符和简写形式，任何字段都可等于\*或ANY，如丢弃所有收到的数据包

```
ovs-ofctl add-flow xenbr0 dl_type=*,nw_src=ANY,actions=drop
```

简写形式为将字段组简写为协议名，目前支持的简写有ip，arp，icmp，tcp，udp，与流规则条件字段的对应关系如下：

```
dl_type=0x0800 <=> ip  
dl_type=0x0806 <=> arp  
dl_type=0x0800, nw_proto=1 <=> icmp  
dl_type=0x0800, nw_proto=6 <=> tcp  
dl_type=0x0800, nw_proto=17 <=> udp  
dl_type=0x86dd. <=> ipv6  
dl_type=0x86dd, nw_proto=6. <=> tcp6  
dl_type=0x86dd, nw_proto=17. <=> udp6  
dl_type=0x86dd, nw_proto=58. <=> icmp6
```

屏蔽某个IP

```
ovs-ofctl add-flow xenbr0 idle_timeout=0,dl_type=0x0800,nw_src=119.75.213.50,actions=drop
```

数据包重定向

```
ovs-ofctl add-flow xenbr0 idle_timeout=0,dl_type=0x0800,nw_proto=1,actions=output:4
```

去除VLAN tag

```
ovs-ofctl add-flow xenbr0 idle_timeout=0,in_port=3,actions=strip_vlan,normal
```

更改数据包源IP地址后转发

```
ovs-ofctl add-flow xenbr0 idle_timeout=0,in_port=3,actions=mod_nw_src:211.68.52.32,normal
```

注包

```
# 格式为：ovs-ofctl packet-out switch in_port actions packet  
# 其中，packet为hex格式数据包  
ovs-ofctl packet-out br2 none output:2 040815162342FFFFFFFFFFFF07C30000
```

## 流表常用字段

- **in\_port=port** 传递数据包的端口的 OpenFlow 端口编号
- **dl\_vlan=vlan** 数据包的 VLAN Tag 值，范围是 0-4095，0xffff 代表不包含 VLAN Tag 的数据包
- **dl\_src=** 和 **dl\_dst=** 匹配源或者目标的 MAC 地址 01:00:00:00:00:00/01:00:00:00:00:00 代表广播地址 00:00:00:00:00:00/01:00:00:00:00:00 代表单播地址
- **dl\_type=ethertype** 匹配以太网协议类型，其中：dl\_type=0x0800 代表 IPv4 协议  
dl\_type=0x086dd 代表 IPv6 协议 dl\_type=0x0806 代表 ARP 协议
- **nw\_src=ip[/netmask]** 和 **nw\_dst=ip[/netmask]** 当 dl\_type=0x0800 时，匹配源或者目标的 IPv4 地址，可以使 IP 地址或者域名
- **nw\_proto=proto** 和 **dl\_type** 字段协同使用。当 dl\_type=0x0800 时，匹配 IP 协议编号；当 dl\_type=0x086dd 代表 IPv6 协议编号
- **table=number** 指定要使用的流表的编号，范围是 0-254。在不指定的情况下，默认值为 0。通过使用流表编号，可以创建或者修改多个 Table 中的 Flow
- **reg=value[/mask]** 交换机中的寄存器的值。当一个数据包进入交换机时，所有的寄存器都被清零，用户可以通过 Action 的指令修改寄存器中的值

## 常见的操作

- **output:port:** 输出数据包到指定的端口。port 是指端口的 OpenFlow 端口编号
- **mod\_vlan\_vid:** 修改数据包中的 VLAN tag
- **strip\_vlan:** 移除数据包中的 VLAN tag
- **mod\_dl\_src/ mod\_dl\_dest:** 修改源或者目标的 MAC 地址信息
- **mod\_nw\_src/mod\_nw\_dst:** 修改源或者目标的 IPv4 地址信息
- **resubmit:port:** 替换流表的 in\_port 字段，并重新进行匹配
- **load:value->dst[start..end]:** 写数据到指定的字段

## 跟踪数据包的处理过程

```
ovs-appctl ofproto/trace br0 in_port=3,tcp,nw_src=10.0.0.2,tcp_dst=22

ovs-appctl ofproto/trace br-int \
    in_port=1,dl_src=00:00:00:00:00:01, \
    dl_dst=00:00:00:00:00:02 -generate
```

## Packet out (注包)

```
import binascii
from scapy.all import *
a=Ether(dst="02:ac:10:ff:00:22",src="02:ac:10:ff:00:11")/IP(dst="172.16.255.22",src="1
72.16.255.11", ttl=10)/ICMP()
print binascii.hexlify(str(a))

ovs-ofctl packet-out br-int 5 "normal" 02AC10FF002202AC10FF001108004500001C000100000A0
15A9DAC10FF0BAC10FF160800F7FF00000000
```

## OVS 文档链接

- <http://openvswitch.org/>
- <http://blog.scottlowe.org/>
- <https://blog.russellbryant.net/>
- Comparing OpenStack Neutron ML2+OVS and OVN – Control Plane

# Build OVS

## 直接源码编译安装

```

export OVS_VERSION="2.6.1"
export OVS_DIR="/usr/src/ovs"
export OVS_INSTALL_DIR="/usr"
curl -sS1 http://openvswitch.org/releases/openvswitch-${OVS_VERSION}.tar.gz | tar -xz
&& mv openvswitch-${OVS_VERSION} ${OVS_DIR}

cd ${OVS_DIR}
./boot.sh
# 如果启用DPDK，还需要加上--with-dpdk=/usr/local/share/dpdk/x86_64-native-linuxapp-gcc
./configure --prefix=${OVS_INSTALL_DIR} --localstatedir=/var --enable-ssl --with-linux
=/lib/modules/$(uname -r)/build
make -j `nproc`
make install
make modules_install

```

### 更新内核模块

```

cat > /etc/depmod.d/openvswitch.conf << EOF
override openvswitch * extra
override vport-* * extra
EOF

depmod -a
cp debian/openvswitch-switch.init /etc/init.d/openvswitch-switch
/etc/init.d/openvswitch-switch force-reload-kmod

```

## 编译RPM包

```
make rpm-fedor RPMBUILD_OPT="--without check"
```

### 启用DPDK

```
make rpm-fedor RPMBUILD_OPT="--with dpdk --without check"
```

### 编译内核模块

```
make rpm-fedorakmod
```

## 编译deb包

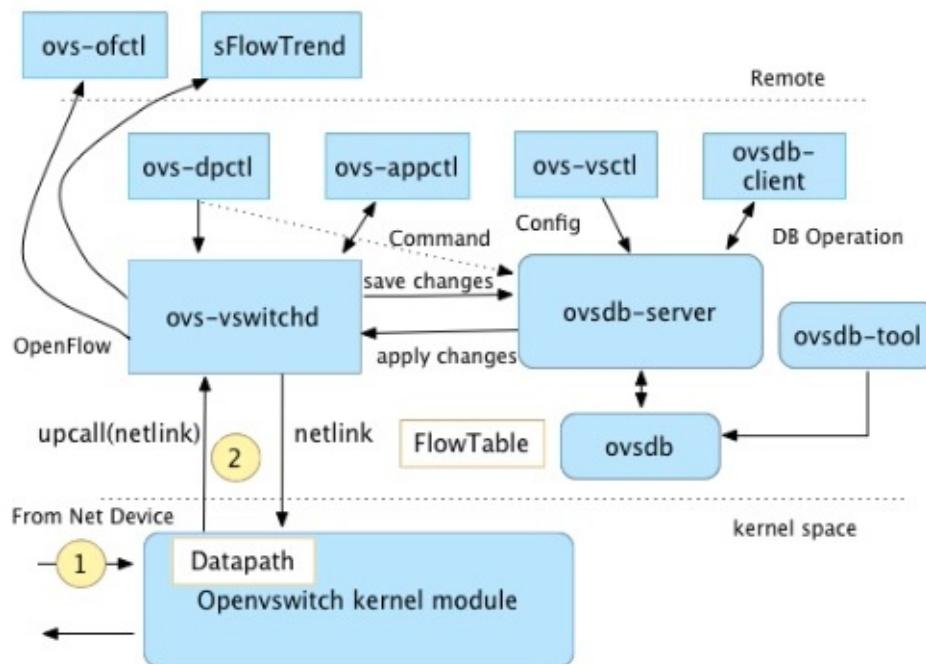
```
apt-get install build-essential fakeroot  
dpkg-checkbuilddeps  
# 已经编译过，需要首先clean  
# fakeroot debian/rules clean  
DEB_BUILD_OPTIONS='parallel=8 nocheck' fakeroot debian/rules binary
```

## 参考文档

- <http://docs.openvswitch.org/en/latest/intro/install/>
- <http://docs.openvswitch.org/en/latest/intro/install/debian/>
- <http://docs.openvswitch.org/en/latest/intro/install/dpdk/>
- <http://docs.openvswitch.org/en/latest/intro/install/general/#hot-upgrading>

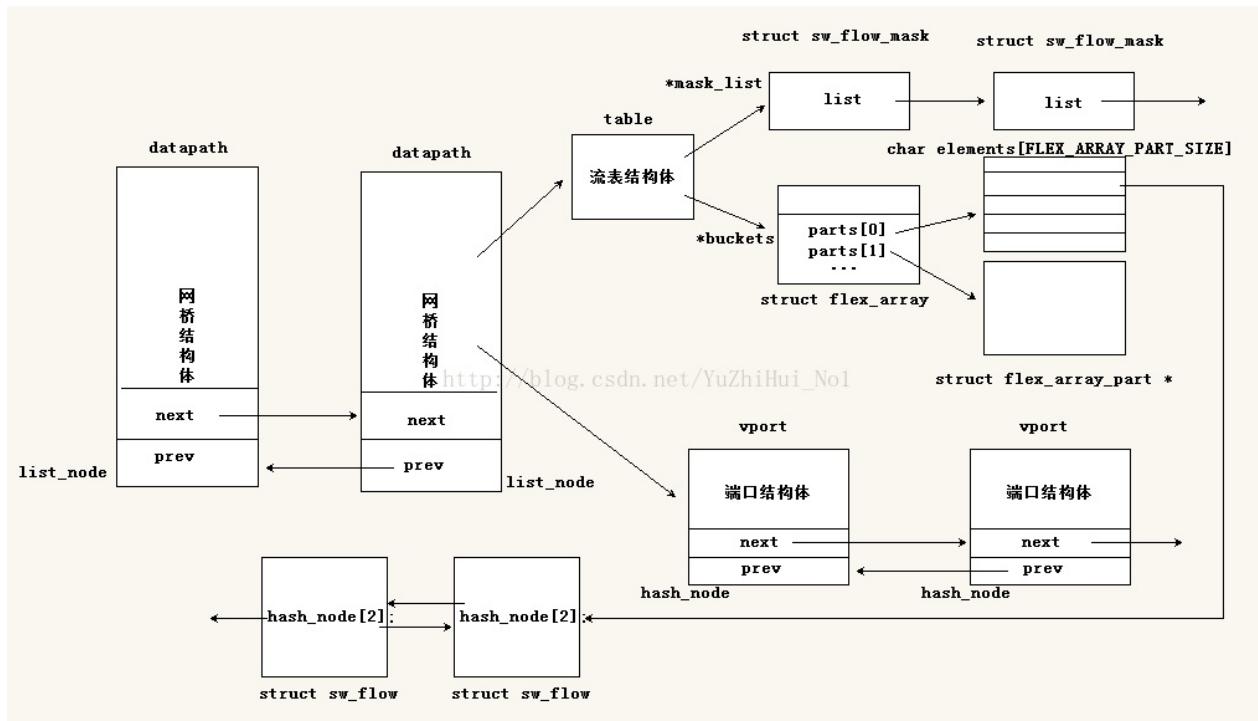
# OVS原理

ovs的架构如下图所示，主要由内核datapath、vswitchd、ovsdb以及用户空间的ovs-ofctl/ovs-ofctl/ovs-dpctl等组成。



- vswitchd是一个守护进程，是ovs的管理和控制服务，通过unix socket将配置信息保存到ovsdb，并通过netlink和内核模块交互
- ovsdb则是ovs的数据库，保存了ovs配置信息
- datapath是负责数据交换的内核模块，比如把接受端口收到的包放到流表中进行匹配，并执行匹配后的动作等。它在初始化和port binding的时候注册钩子函数，把端口的报文处理接管到内核模块

## 主要数据结构



(图片来自csdn)

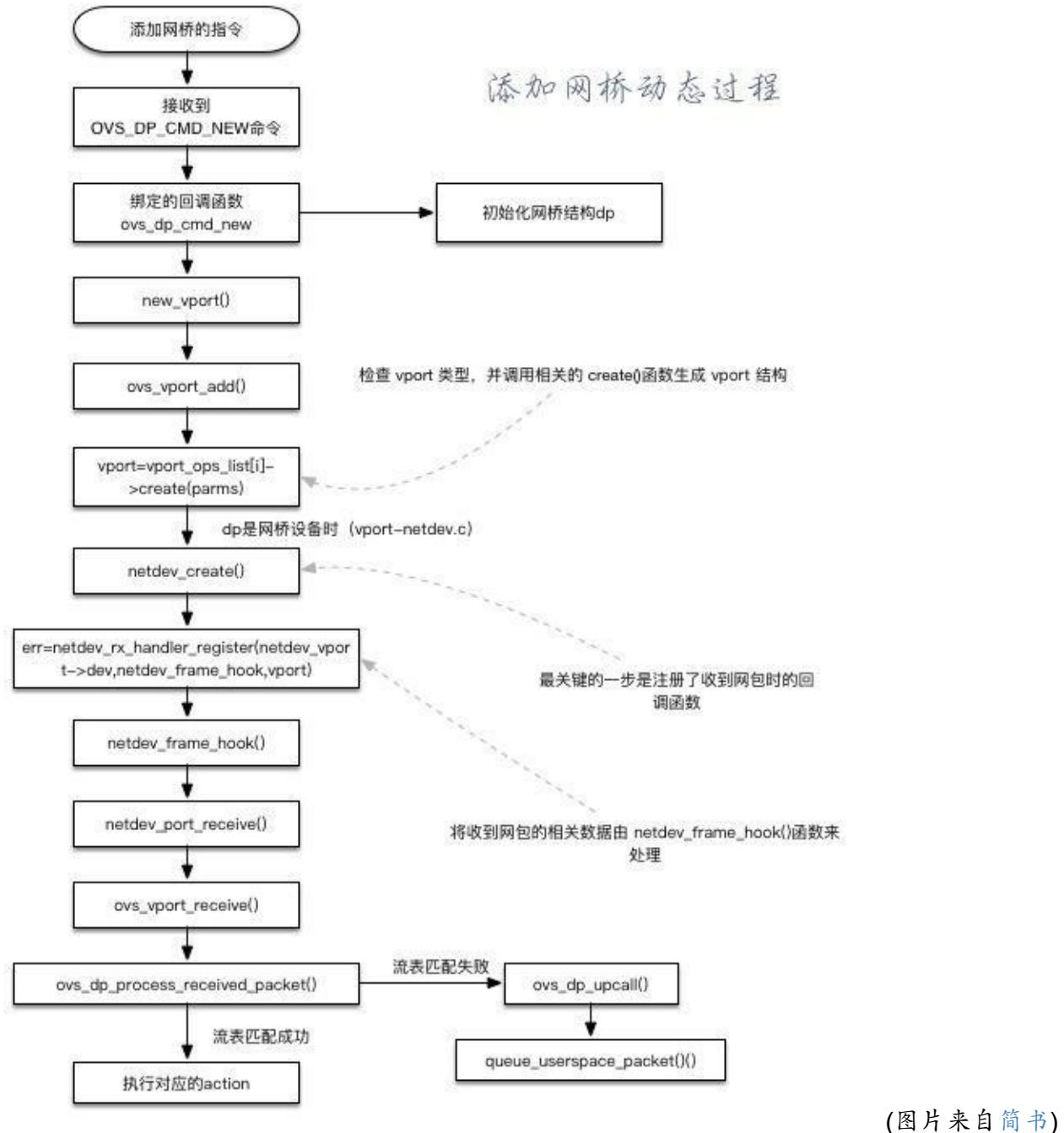
## 主要流程

注：部分转载自 [OVS 源码分析整理](#)

### 添加网桥

1. 键入命令 `ovs-vsctl add-br testBR`
2. 内核中的 `openvswitch.ko` 收到一个添加网桥的命令时候——即收到 `OVS_DATAPATH_FAMILY` 通道的 `OVS_DP_CMD_NEW` 命令。该命令绑定的回调函数为 `ovs_dp_cmd_new`
3. `ovs_dp_cmd_new` 函数除了初始化 `dp` 结构外，调用 `new_vport` 函数来生成新的 `vport`
4. `new_vport` 函数调用 `ovs_vport_add()` 来尝试生成一个新的 `vport`
5. `ovs_vport_add()` 函数会检查 `vport` 类型（通过 `vport_ops_list[]` 数组），并调用相关的 `create()` 函数来生成 `vport` 结构
6. 当 `dp` 是网络设备时 (`vport_netdev.c`)，最终由 `ovs_vport_add()` 函数调用的是 `netdev_create()` 【在 `vport_ops_list` 的 `ovs_netdev_ops` 中】
7. `netdev_create()` 函数最关键的一步是注册了收到网包时的回调函数
8. `err=netdev_rx_handler_register(netdev_vport->dev, netdev_frame_hook, vport);`
9. 操作是将 `netdev_vport->dev` 收到网包时的相关数据由 `netdev_frame_hook()` 函数来处理，都是些辅助处理，依次调用各处理函数，在 `netdev_port_receive()` 【这里会进行数据包的拷贝，避免损坏】进入 `ovs_vport_receive()` 回到 `vport.c`，从 `ovs_dp_process_receive_packet()` 回到 `datapath.c`，进行统一处理
10. 流程：`netdev_frame_hook() -> netdev_port_receive -> ovs_vport_receive ->`

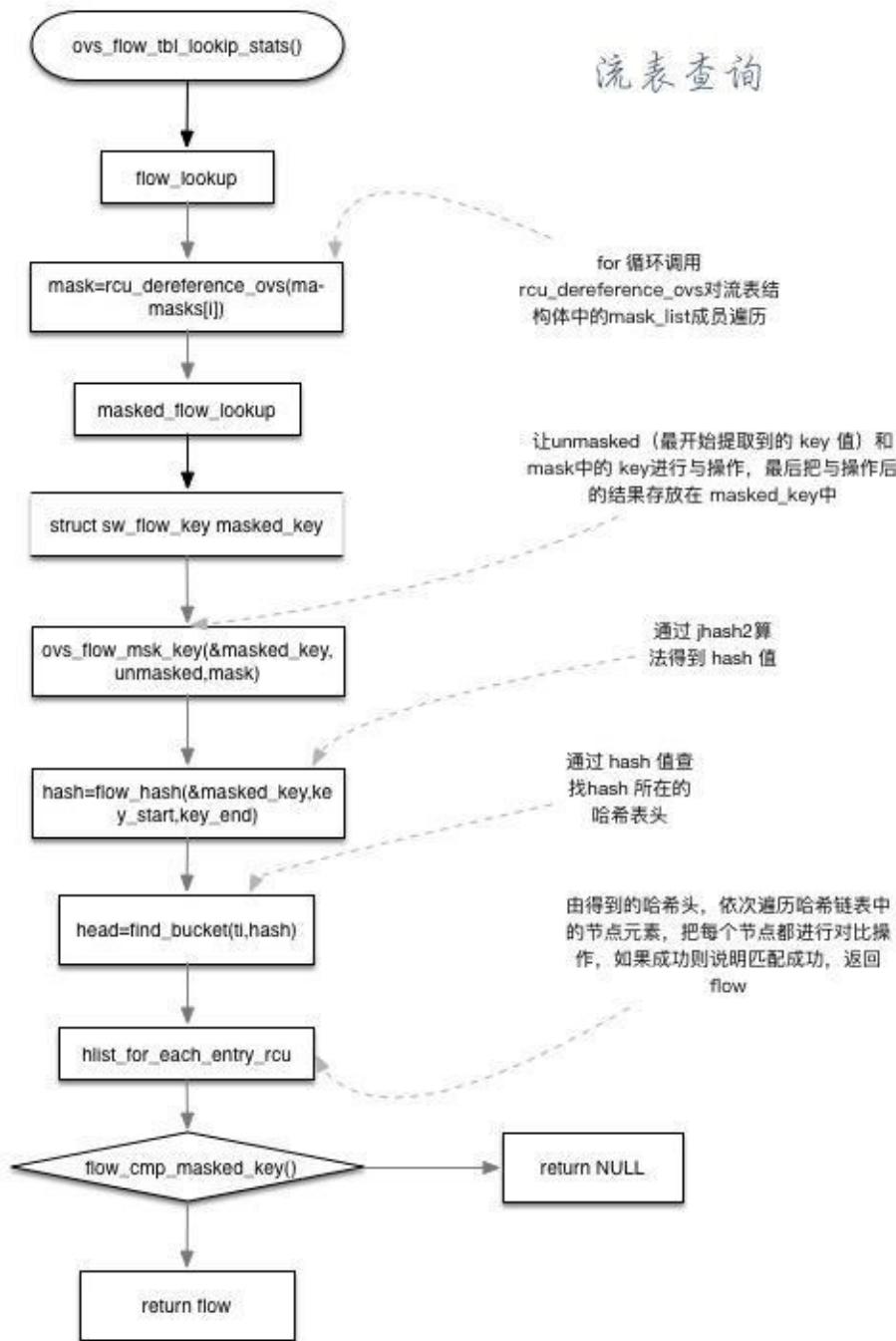
- >ovs\_dp\_process\_received\_packet()
11. net\_port\_receive()首先检测是否 skb 被共享，若是则得到 packet 的拷贝。
  12. net\_port\_receive()其调用ovs\_vport\_receive()，检查包的校验和，然后交付给我们的 vport通用层来处理。



## 流表匹配

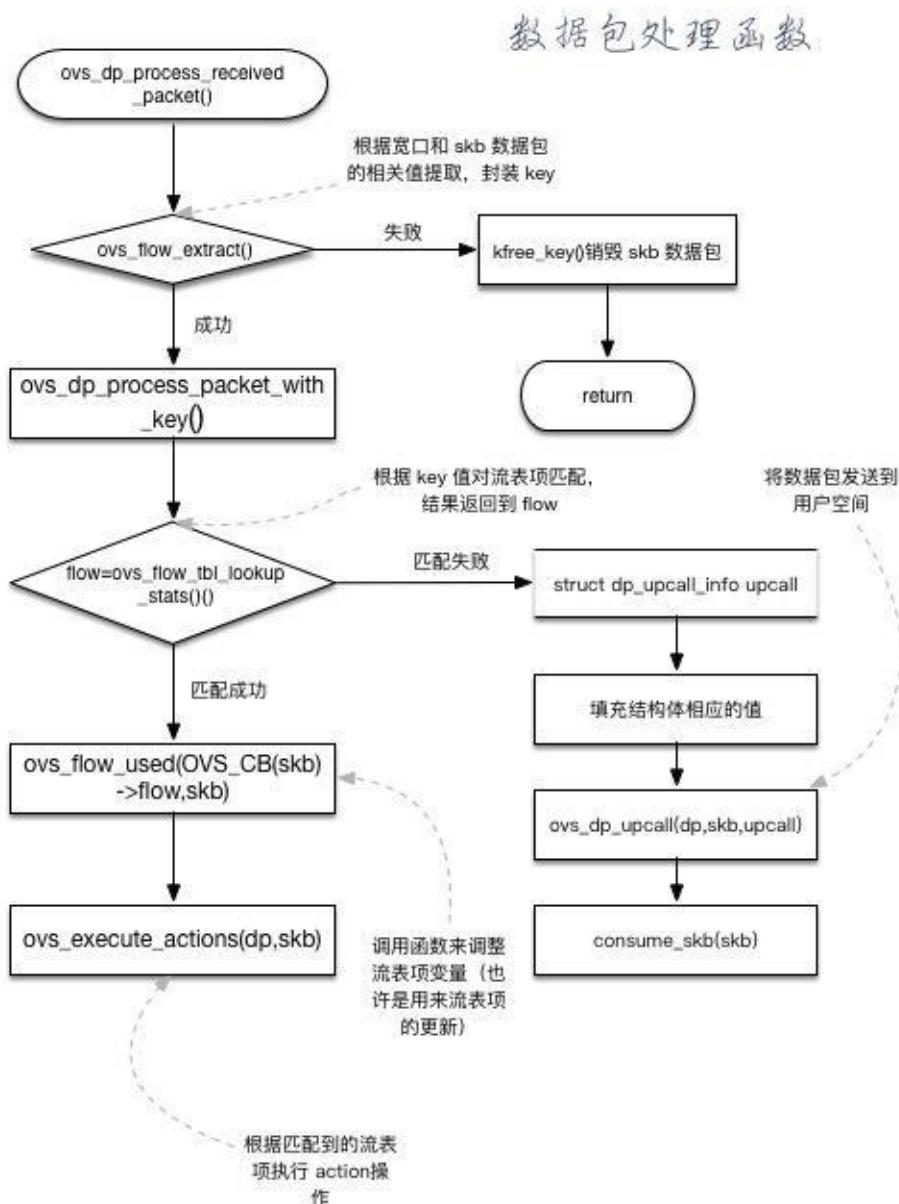
1. flow\_lookup()查找对应的流表项
2. for 循环调用 rcu\_dereference\_ovs 对流表结构体中的 mask\_list 成员遍历，找到对应的成员
3. flow=masked\_flow\_lookup()遍历进行下一级 hmap 查找，找到为止

4. 进入包含函数 `ovs_flow_mask_key(&masked_key,unmasked,mask)`，将最开始提取的 Key 值和 mask 的 key 值进行“与”操作，结果存放在 `masked_key` 中，用来得到后面的 Hash 值
5. `hash=flow_hash(&masked_key,key_start,key_end)`key 值的匹配字段只有部分
6. `ovs_vport_add()` 函数会检查 vport 类型（通过 `vport_ops_list[]` 数组），并调用相关的 `create()` 函数来生成 vport 结构
7. 可见，当 dp 时网络设备时（`vport_netdev.c`），最终由 `ovs_vport_add()` 函数调用的是 `netdev_create()` 【在 `vport_ops_list` 的 `ovs_netdev_ops` 中】
8. `netdev_vport->dev` 收到网包时的相关数据由 `netdev_frame_hook()` 函数来处理，都是些辅助处理，依次调用各处理函数，在 `netdev_port_receive()` 【这里会进行数据包的拷贝，避免损坏】进入 `ovs_vport_receive()` 回到 `vport.c`，从 `ovs_dp_process_receive_packet()` 回到 `datapath.c`，进行统一处理



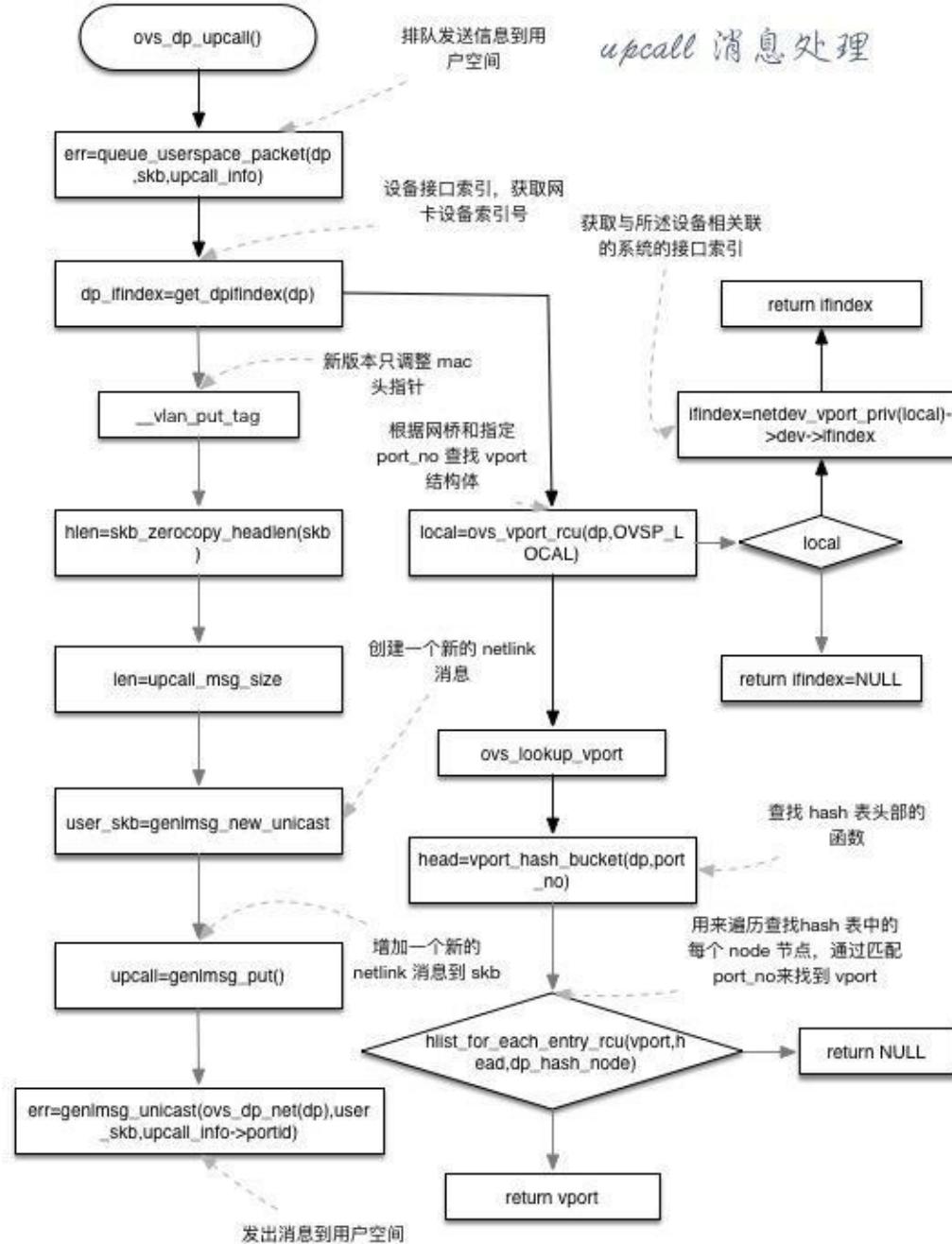
## 收包处理

1. `ovs_vport_receive_packets()` 调用 `ovs_flow_extract` 基于 `skb` 生成 `key` 值，并检查是否有错，然后调用 `ovs_dp_process_packet`。交付给 datapath 处理
2. `ovs_flow_tbl_lookup_stats`。基于前面生成的 `key` 值进行流表查找，返回匹配的流表项，结构为 `sw_flow`。
3. 若不存在匹配，则调用 `ovs_dp_upcall` 上传至 userspace 进行匹配。(包括包和 `key` 都要上传)
4. 若存在匹配，则直接调用 `ovs_execute_actions` 执行对应的 action，比如添加 `vlan` 头，转发到某个 `port` 等。



## upcall 消息处理

1. `ovs_dp_upcall()`首先调用 `err=queue_userspace_packet()`将信息排队发到用户空间去
2. `dp_ifindex=get_dpifindex(dp)`获取网卡设备索引号
3. 调整 VLAN 的 MAC 地址头指针
4. 网络链路属性，如果不需要填充则调用此函数
5. `len=upcall_msg_size()`，获得 upcall 发送消息的大小
6. `user_skb=genlmsg_new_unicast`，创建一个新的 netlink 消息
7. `upcall=genlmsg_put()`增加一个新的 netlink 消息到 skb
8. `err=genlmsg_unicast()`，发送消息到用户空间去处理



## 参考文档

- OVS 源码分析整理
- openvswitch源码分析
- OVS Deep Dive

# OVN

## OVN 简介

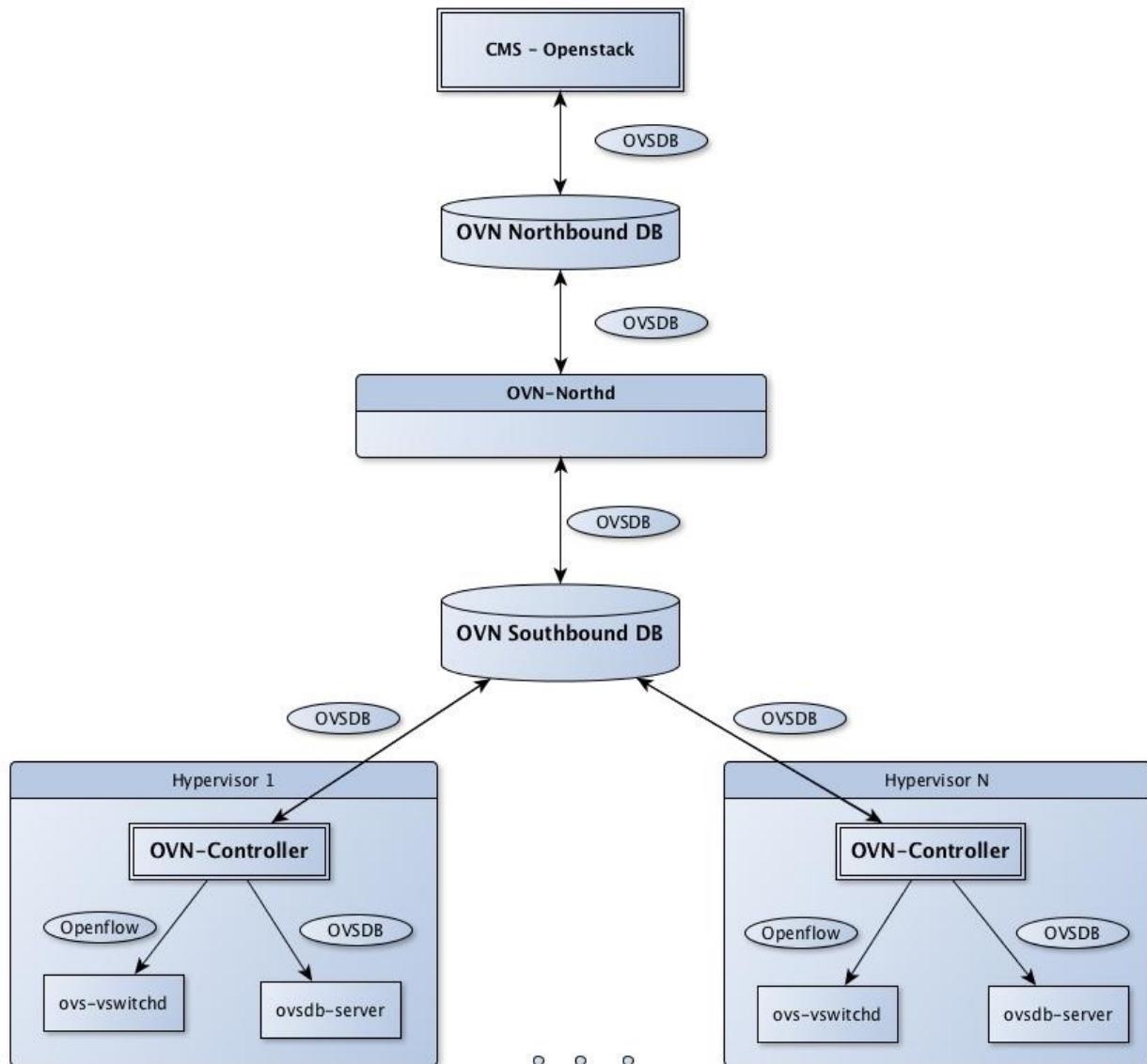
[OVN \(Open Virtual Network\)](#) 是OVS提供的原生虚拟化网络方案，旨在解决传统SDN架构（比如Neutron DVR）的性能问题。其主要功能包括

1. L2/L3虚拟网络以及逻辑交换机(logical switch)
2. L2/L3/L4 ACL
3. IPv4/IPv6分布式L3路由
4. ARP and IPv6 Neighbor Discovery suppression for known IP-MAC bindings
5. Native support for NAT and load balancing using OVS connection tracking
6. Native fully distributed support for DHCP
7. Works with any OVS datapath (such as the default Linux kernel datapath, DPDK, or Hyper-V) that supports all required features (namely Geneve tunnels and OVS connection tracking)
8. Supports L3 gateways from logical to physical networks
9. Supports software-based L2 gateways
10. Supports TOR (Top of Rack) based L2 gateways that implement the hardware\_vtep schema
11. Can provide networking for both VMs and containers running inside of those VMs, without a second layer of overlay networking

## OVN架构

OVN由以下组件构成：

- northbound database：存储逻辑交换机、路由器、ACL、端口等的信息，目前基于ovsdb-server，未来可能会支持etcd v3
- ovn-northd: 集中式控制器，负责把northbound database数据分发到各个ovn-controller
- ovn-controller: 运行在每台机器上的本地SDN控制器
- southbound database：基于ovsdb-server（未来可能会支持etcd v3），包含三类数据
  - 物理网络数据，比如VM的IP地址和隧道封装格式
  - 逻辑网络数据，比如报文转发方式
  - 物理网络和逻辑网络的绑定关系



补充说明

- Data Path : OVN的实现简单有效，都是基于OVS原生的功能特性来做的（由于OVN的实现不依赖于内核特性，这些功能在OVS+DPDK上也完全支持），比如
  - security policies 基于 OVS+conntrack 实现
  - 分布式L3路由基于OVS flow实现
- Logical Flows : 逻辑流表，会由ovn-northd分发给每台机器的ovn-controller，然后ovn-controller再把它们转换为物理流表
- 更多架构可以参考[Open Virtual Network architecture](#)

## OVN安装

如果想要安装master版本，可以使用<https://copr.fedorainfracloud.org/coprs/leifmadsen/ovs-master/>的BUILD：

## CentOS

```
wget -o /etc/yum.repos.d/ovs-master.repo https://copr.fedorainfracloud.org/coprs/leifmadsen/ovs-master/repo/epel-7/leifmadsen-ovs-master-epel-7.repo
yum install openvswitch openvswitch-ovn-*
```

## Ubuntu

```
apt-get install -y openvswitch-switch ovn-central ovn-common ovn-controller-vtep ovn-docker ovn-host
```

## 启动ovn

控制节点：

```
# start ovsdb-server
/usr/share/openvswitch/scripts/ovs-ctl start --system-id=random

# start ovn northd
/usr/share/openvswitch/scripts/ovn-ctl start_northd

export CENTRAL_IP=10.140.0.2
export LOCAL_IP=10.140.0.2
export ENCAP_TYPE=vxlan
ovs-vsctl set Open_vSwitch . external_ids:ovn-remote="tcp:$CENTRAL_IP:6642" external_ids:ovn-nb="tcp:$CENTRAL_IP:6641" external_ids:ovn-encap-ip=$LOCAL_IP external_ids:ovn-encap-type="$ENCAP_TYPE"
```

计算节点：

```
# start ovsdb-server
/usr/share/openvswitch/scripts/ovs-ctl start --system-id=random
# start ovn-controller and vtep
/usr/share/openvswitch/scripts/ovn-ctl start_controller
/usr/share/openvswitch/scripts/ovn-ctl start_controller_vtep

export CENTRAL_IP=10.140.0.2
export LOCAL_IP=10.140.0.2
export ENCAP_TYPE=vxlan
ovs-vsctl set Open_vSwitch . external_ids:ovn-remote="tcp:$CENTRAL_IP:6642" external_ids:ovn-nb="tcp:$CENTRAL_IP:6641" external_ids:ovn-encap-ip=$LOCAL_IP external_ids:ovn-encap-type="$ENCAP_TYPE"
```

对于ovs 1.7，还需要设置

```
ovn-nbctl set-connection ptcp:6641  
ovn-sbctl set-connection ptcp:6642
```

# Build OVN

## Update & install dependencies

```
apt-get update  
apt-get -y install build-essential fakeroot
```

## Install Build-Depends from debian/control file

```
apt-get -y install graphviz autoconf automake bzip2 debhelper dh-autoreconf libssl-dev  
libtool openssl  
apt-get -y install procps python-all python-twisted-conch python-zopeinterface python-six
```

## Check the working directory & build

```
curl -o openvswitch-2.7.0.tar.gz http://openvswitch.org/releases/openvswitch-2.7.0.tar.gz  
tar zxvf openvswitch-2.7.0.tar.gz  
cd openvswitch-2.7.0  
  
# if everything is ok then this should return no output  
dpkg-checkbuilddeps  
  
`DEB_BUILD_OPTIONS='parallel=8 nocheck' fakeroot debian/rules binary`
```

The .deb files for ovs will be built and placed in the parent directory (ie. in .../). The next step is to build the kernel modules.

## Install datapath sources

```
cd ..  
apt-get -y install module-assistant  
dpkg -i openvswitch-datapath-source_2.7.0-1_all.deb
```

## Build kernel modules using module-assistant

```
m-a prepare  
m-a build openvswitch-datapath
```

Copy the resulting deb package. Note that your version may differ slightly depending on your specific kernel version.

```
cp /usr/src/openvswitch-datapath-module-*.deb ./
apt-get -y install python-six python2.7
dpkg -i openvswitch-datapath-module-*.deb
dpkg -i openvswitch-common_2.7.0-1_amd64.deb openvswitch-switch_2.7.0-1_amd64.deb
dpkg -i ovn-central_2.7.0-1_amd64.deb ovn-common_2.7.0-1_amd64.deb ovn-controller-vtep
_2.7.0-1_amd64.deb ovn-docker_2.7.0-1_amd64.deb ovn-host_2.7.0-1_amd64.deb python-open
vswitch_2.7.0-1_all.deb
```

```
/usr/share/openvswitch/scripts/ovs-ctl start --system-id=random
/usr/share/openvswitch/scripts/ovn-ctl start_northd
/usr/share/openvswitch/scripts/ovn-ctl start_controller
/usr/share/openvswitch/scripts/ovn-ctl start_controller_vtep
export CENTRAL_IP=10.140.0.2
export LOCAL_IP=10.140.0.2
export ENCAP_TYPE=vxlan
ovs-vsctl set Open_vSwitch . external_ids:ovn-remote="tcp:$CENTRAL_IP:6642" external_i
ds:ovn-nb="tcp:$CENTRAL_IP:6641" external_ids:ovn-encap-ip=$LOCAL_IP external_ids:ovn-
encap-type="$ENCAP_TYPE"
```

# OVN 实践

## OVN Logical Flow

OVN逻辑流表会由ovn-northd分发给每台机器的ovn-controller，然后ovn-controller再把它们转换为物理流表。

更多参考

- <https://blog.russellbryant.net/2016/11/11/ovn-logical-flows-and-ovn-trace/>
- <https://blog.russellbryant.net/2015/10/22/openstack-security-groups-using-ovn-acls/>

## OVN安全组

使用OVN只需要把VM的tap直接连接到br-int（而不是现在需要多加一层Linux Bridge），并使用OVS conntrack根据连接状态进行匹配，提高了流表的查找速度，同时也支持有状态防火墙和NAT。

```
# allow all ip traffic from port "ls1-vm1" on switch "ls1" and allowing related connections back in
ovn-nbctl acl-add ls1 from-lport 1000 "inport == \"ls1-vm1\" && ip" allow-related

# allow ssh to ls1-vm1
ovn-nbctl acl-add ls1 to-lport 999 "outport == \"ls1-vm1\" && tcp.dst == 22" allow-related

# block all IPv4/IPv6 traffic to ls1-vm1
ovn-nbctl acl-add ls1 to-lport 998 "outport == \"ls1-vm1\" && ip" drop

# using address sets
ovn-nbctl create Address_Set name=wwwServers addresses=172.16.1.2,172.16.1.3
ovn-nbctl create Address_Set name=www6Servers addresses=\"fd00::1\", \"fd00::2\"
ovn-nbctl create Address_Set name=macs addresses=\"02:00:00:00:00:01\", \"02:00:00:00:00:02\"
ovn-nbctl create Address_Set name=dmz addresses=\"172.16.255.130/31\"
# allow from dmz on 3306
ovn-nbctl acl-add inside to-lport 1000 'outport == "inside-vm3" && ip4.src == $dmz && tcp.dst == 3306' allow-related

# clean up
ovn-nbctl acl-del dmz
ovn-nbctl acl-del inside
ovn-nbctl destroy Address_Set dmz
```

更多参考<https://blog.russellbryant.net/2015/10/22/openstack-security-groups-using-ovn-acls/>。

## OVN L2

OVN L2 功能包括

- L2 switch
- L2 ACL
- Supports software-based L2 gateways
- Supports TOR (Top of Rack) based L2 gateways that implement the hardware\_vtep schema
- Can provide networking for both VMs and containers running inside of those VMs, without a second layer of overlay networking

```

# Create the first logical switch and its two ports.
ovn-nbctl ls-add sw0

ovn-nbctl lsp-add sw0 sw0-port1
ovn-nbctl lsp-set-addresses sw0-port1 "00:00:00:00:00:01 10.0.0.51"
ovn-nbctl lsp-set-port-security sw0-port1 "00:00:00:00:00:01 10.0.0.51"

ovn-nbctl lsp-add sw0 sw0-port2
ovn-nbctl lsp-set-addresses sw0-port2 "00:00:00:00:00:02 10.0.0.52"
ovn-nbctl lsp-set-port-security sw0-port2 "00:00:00:00:00:02 10.0.0.52"

# Create the second logical switch and its two ports.
ovn-nbctl ls-add sw1

ovn-nbctl lsp-add sw1 sw1-port1
ovn-nbctl lsp-set-addresses sw1-port1 "00:00:00:00:00:03 192.168.1.51"
ovn-nbctl lsp-set-port-security sw1-port1 "00:00:00:00:00:03 192.168.1.51"

ovn-nbctl lsp-add sw1 sw1-port2
ovn-nbctl lsp-set-addresses sw1-port2 "00:00:00:00:00:04 192.168.1.52"
ovn-nbctl lsp-set-port-security sw1-port2 "00:00:00:00:00:04 192.168.1.52"

# Create a logical router between sw0 and sw1.
ovn-nbctl create Logical_Router name=lr0

ovn-nbctl lrp-add lr0 lrp0 00:00:00:00:ff:01 10.0.0.1/24
ovn-nbctl lsp-add sw0 sw0-lrp0 \
    -- set Logical_Switch_Port sw0-lrp0 type=router \
    options:router-port=lrp0 addresses='00:00:00:00:ff:01'

ovn-nbctl lrp-add lr0 lrp1 00:00:00:00:ff:02 192.168.1.1/24
ovn-nbctl lsp-add sw1 sw1-lrp1 \
    -- set Logical_Switch_Port sw1-lrp1 type=router \
    options:router-port=lrp1 addresses='00:00:00:00:ff:02'

# Create ovs port
# Create ports on the local OVS bridge, br-int. When ovn-controller
# sees these ports show up with an "iface-id" that matches the OVN
# logical port names, it associates these local ports with the OVN
# logical ports. ovn-controller will then set up the flows necessary
# for these ports to be able to communicate each other as defined by
# the OVN logical topology.
ovs-vsctl add-port br-int lport1 -- set interface lport1 type=internal \
    -- set Interface lport1 external_ids:iface-id=sw0-port1
ovs-vsctl add-port br-int lport2 -- set interface lport2 type=internal \
    -- set Interface lport2 external_ids:iface-id=sw0-port2

```

更多参考<http://galsagie.github.io/2015/05/30/ovn-deep-dive/>。

## OVN L3

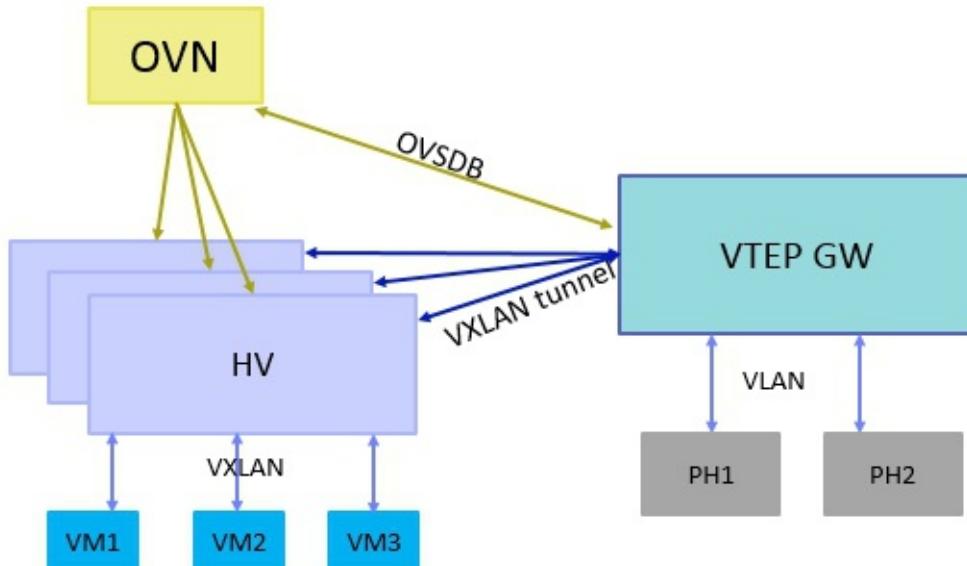
OVN L3 的功能包括

- IPv4/IPv6 分布式 L3 路由
- ARP and IPv6 Neighbor Discovery suppression for known IP-MAC bindings
- L3 ACL
- Native support for NAT and load balancing using OVS connection tracking
- Native fully distributed support for DHCP
- Supports L3 gateways from logical to physical networks

```
# SNAT
# create snat rule which will nat to the edge1-outside interface
ovn-nbctl -- --id=@nat create nat type="snat" logical_ip=172.16.255.128/25 \
external_ip=10.127.0.129 -- add logical_router edge1 nat @nat
```

更多参考<http://galsagie.github.io/2015/11/23/ovn-l3-deepdive/> 和  
<http://networkop.co.uk/blog/2016/12/10/ovn-part2/>。

## OVN VTEP



OVN 可以通过 VTEP 网关把物理网络和逻辑网络连接起来。VTEP 网关可以是 TOR (Top of Rack) switch，目前很多硬件厂商都支持，比如 Arista，Juniper，HP 等等；也可以是软件做的逻辑 switch，OVS 社区就做了一个简单的 VTEP 模拟器。

VTEP 网关需要遵守 VTEP OVSDB schema，它里面定义了 VTEP 网关需要支持的数据项和内容，VTEP 通过 OVSDB 协议与 OVN 通信，通信的流程 OVN 也有相关标准，VTEP 上需要一个 ovn-controller-vtep 来做 ovn-controller 所做的事情。VTEP 网关和 HV 之间常用 VXLAN 封装技术。

虽然 VTEP OVSDB schema 里面定义了三层的表项，但是目前没有硬件厂商支持，VTEP 模拟器也不支持，所以 VTEP 网络只支持二层的功能，也就是说只能连接物理网络的 VLAN 到逻辑网络的 VXLAN，如果 VTEP 上不同 VLAN 之间要做路由，需要 OVN 里面的路由器来做。

## OVN Chassis

Chassis 是 OVN 新增的概念，OVS 里面没有这个概念，Chassis 可以是 HV，也可以是 VTEP 网关。Chassis 的信息保存在 Southbound DB 里面，由 ovn-controller/ovn-controller-vtep 来维护。

以 ovn-controller 为例，当 ovn-controller 启动的时候，它去本地的数据库 Open\_vSwitch 表里面读取 external\_ids:system\_id，external\_ids:ovn-remote，external\_ids:ovn-encap-ip 和 external\_ids:ovn-encap-type 的值，然后它把这些值写到 Southbound DB 里面的表 Chassis 和表 Encap 里面：

- external\_ids:system\_id 表示 Chassis 名字
- external\_ids:ovn-remote 表示 Southbound DB 的 IP 地址
- external\_ids:ovn-encap-ip 表示 tunnel endpoint IP 地址，可以是 HV 的某个接口的 IP 地址
- external\_ids:ovn-encap-type 表示 tunnel 封装类型，可以是 VXLAN/Geneve/STT

external\_ids:ovn-encap-ip 和 external\_ids:ovn-encap-type 是一对，每个 tunnel IP 地址对应一个 tunnel 封装类型，如果 HV 有多个接口可以建立 tunnel，可以在 ovn-controller 启动之前，把每对值填在 table Open\_vSwitch 里面。

## OVN tunnel

OVN 支持的 tunnel 类型有三种，分别是 Geneve，STT 和 VXLAN。HV 与 HV 之间的流量，只能用 Geneve 和 STT 两种，HV 和 VTEP 网关之间的流量除了用 Geneve 和 STT 外，还能用 VXLAN，这是为了兼容硬件 VTEP 网关，因为大部分硬件 VTEP 网关只支持 VXLAN。

虽然 VXLAN 是数据中心常用的 tunnel 技术，但是 VXLAN header 是固定的，只能传递一个 VNID (VXLAN network identifier)，如果想在 tunnel 里面传递更多的信息，VXLAN 实现不了。所以 OVN 选择了 Geneve 和 STT，Geneve 的头部有个 option 字段，支持 TLV 格式，用户可以根据自己的需要进行扩展，而 STT 的头部可以传递 64-bit 的数据，比 VXLAN 的 24-bit 大很多。

OVN tunnel 封装时使用了三种数据，

- Logical datapath identifier (逻辑的数据通道标识符)：datapath 是 OVS 里面的概念，报文需要送到 datapath 进行处理，一个 datapath 对应一个 OVN 里面的逻辑交换机或者

逻辑路由器，类似于 tunnel ID。这个标识符有 24-bit，由 ovn-northd 分配的，全局唯一，保存在 Southbound DB 里面的表 Datapath\_Binding 的列 tunnel\_key 里。

- Logical input port identifier（逻辑的入端口标识符）：进入 logical datapath 的端口标识符，15-bit 长，由 ovn-northd 分配的，在每个 datapath 里面唯一。它可用范围是 1-32767，0 预留给内部使用。保存在 Southbound DB 里面的表 Port\_Binding 的列 tunnel\_key 里。
- Logical output port identifier（逻辑的出端口标识符）：出 logical datapath 的端口标识符，16-bit 长，范围 0-32767 和 logical input port identifier 含义一样，范围 32768-65535 给组播组使用。对于每个 logical port，input port identifier 和 output port identifier 相同。

如果 tunnel 类型是 Geneve，Geneve header 里面的 VNI 字段填 logical datapath identifier，Option 字段填 logical input port identifier 和 logical output port identifier，TLV 的 class 为 0xffff，type 为 0，value 为 1-bit 0 + 15-bit logical input port identifier + 16-bit logical output port identifier。

如果 tunnel 类型是 STT，上面三个值填在 Context ID 字段，格式为 9-bit 0 + 15-bit logical input port identifier + 16-bit logical output port identifier + 24-bit logical datapath identifier。

OVS 的 tunnel 封装是由 Openflow 流表来做的，所以 ovn-controller 需要把这三个标识符写到本地 HV 的 Openflow flow table 里面，对于每个进入 br-int 的报文，都会有这三个属性，logical datapath identifier 和 logical input port identifier 在入口方向被赋值，分别存在 openflow metadata 字段和 Nicira 扩展寄存器 reg6 里面。报文经过 OVS 的 pipeline 处理后，如果需要从指定端口发出去，只需要把 Logical output port identifier 写在 Nicira 扩展寄存器 reg7 里面。

OVN tunnel 里面所携带的 logical input port identifier 和 logical output port identifier 可以提高流表的查找效率，OVS 流表可以通过这两个值来处理报文，不需要解析报文的字段。

OVN 里面的 tunnel 类型是由 HV 上面的 ovn-controller 来设置的，并不是由 CMS 指定的，并且 OVN 里面的 tunnel ID 又由 OVN 自己分配的，所以用 neutron 创建 network 时指定 tunnel 类型和 tunnel ID（比如 vnid）是无用的，OVN 不做处理。

## OVN Northbound DB

Northbound DB 是 OVN 和 CMS 之间的接口，Northbound DB 里面的几乎所有的内容都是由 CMS 产生的，ovn-northd 监听这个数据库的内容变化，然后翻译，保存到 Southbound DB 里面。

Northbound DB 里面主要有如下几张表：

- Logical\_Switch：每一行代表一个逻辑交换机，逻辑交换机有两种，一种是 overlay logical switches，对应于 neutron network，每创建一个 neutron network，networking-

ovn 会在这张表里增加一行；另一种是 bridged logical switch，连接物理网络和逻辑网络，被 VTEP gateway 使用。Logical\_Switch 里面保存了它包含的 logical port（指向 Logical\_Port table）和应用在它上面的 ACL（指向 ACL table）。

- Logical\_Port：每一行代表一个逻辑端口，每创建一个 neutron port，networking-ovn 会在这张表里增加一行，每行保存的信息有端口的类型，比如 patch port，localnet port，端口的 IP 和 MAC 地址，端口的状态 UP/Down。
- ACL：每一行代表一个应用到逻辑交换机上的 ACL 规则，如果逻辑交换机上面的所有端口都没有配置 security group，那么这个逻辑交换机上不应用 ACL。每条 ACL 规则包含匹配的内容，方向，还有动作。
- Logical\_Router：每一行代表一个逻辑路由器，每创建一个 neutron router，networking-ovn 会在这张表里增加一行，每行保存了它包含的逻辑的路由器端口。
- Logical\_Router\_Port：每一行代表一个逻辑路由器端口，每创建一个 router interface，networking-ovn 会在这张表里加一行，它主要保存了路由器端口的 IP 和 MAC。

## OVN Southbound DB

Southbound DB 里面有如下几张表：

- Chassis：每一行表示一个 HV 或者 VTEP 网关，由 ovn-controller/ovn-controller-vtep 填写，包含 chassis 的名字和 chassis 支持的封装的配置（指向表 Encap），如果 chassis 是 VTEP 网关，VTEP 网关上和 OVN 关联的逻辑交换机也保存在这张表里。
- Encap：保存着 tunnel 的类型和 tunnel endpoint IP 地址。
- Logical\_Flow：每一行表示一个逻辑的流表，这张表是 ovn-northd 根据 Nourthbound DB 里面二三层拓扑信息和 ACL 信息转换而来的，ovn-controller 把这个表里面的流表转换成 OVS 流表，配到 HV 上的 OVS table。流表主要包含匹配的规则，匹配的方向，优先级，table ID 和执行的动作。
- Multicast\_Group：每一行代表一个组播组，组播报文和广播报文的转发由这张表决定，它保存了组播组所属的 datapath，组播组包含的端口，还有代表 logical egress port 的 tunnel\_key。
- Datapath\_Binding：每一行代表一个 datapath 和物理网络的绑定关系，每个 logical switch 和 logical router 对应一行。它主要保存了 OVN 给 datapath 分配的代表 logical datapath identifier 的 tunnel\_key。
- Port\_Binding：这张表主要用来确定 logical port 处在哪个 chassis 上面。每一行包含的内容主要有 logical port 的 MAC 和 IP 地址，端口类型，端口属于哪个 datapath binding，代表 logical input/output port identifier 的 tunnel\_key，以及端口处在哪个 chassis。端口所处的 chassis 由 ovn-controller/ovn-controller 设置，其余的值由 ovn-northd 设置。

表 Chassis 和表 Encap 包含的是物理网络的数据，表 Logical\_Flow 和表 Multicast\_Group 包含的是逻辑网络的数据，表 Datapath\_Binding 和表 Port\_Binding 包含的是逻辑网络和物理网络绑定关系的数据。

## OVN DB 汇总

```
ovn-nbctl list Logical_Switch  
ovn-nbctl list Logical_Switch_Port  
ovn-nbctl list ACL  
ovn-nbctl list Address_Set  
ovn-nbctl list Logical_Router  
ovn-nbctl list Logical_Router_Port  
  
ovn-sbctl list Chassis  
ovn-sbctl list Encap  
ovn-nbctl list Address_Set  
ovn-sbctl lflow-list  
ovn-sbctl list Multicast_Group  
ovn-sbctl list Datapath_Binding  
ovn-sbctl list Port_Binding  
ovn-sbctl list MAC_Binding
```

## OVN Load Balancer

OVN Load Balancer提供了一种基于hash的负载均衡机制，可以用在逻辑switch或者逻辑router上：

- 用在logical router上
  - 只能用在gateway router上
  - 集中式（而不是分布式的）
- 用在logical switch上
  - 分布式的
  - 由于OVN Load Balancer仅处理ingress，所以要把它用在client logical switch（而不是server logical switch）

```
uuid=`ovn-nbctl create load_balancer vips:10.127.0.254="172.16.255.130,172.16.255.131"
`  
  
# apply to logical router  
ovn-nbctl set logical_router edge1 load_balancer=$uuid  
  
# clean up  
ovn-nbctl clear logical_router edge1 load_balancer  
ovn-nbctl destroy load_balancer $uuid
```

```
uuid=`ovn-nbctl create load_balancer vips:172.16.255.62="172.16.255.130,172.16.255.131"
`  
  
# apply to logical switch  
ovn-nbctl set logical_switch inside load_balancer=$uuid  
  
# clean up  
ovn-nbctl clear logical_switch inside load_balancer  
ovn-nbctl destroy load_balancer $uuid
```

## DHCP

```
ovn-nbctl ls-add dmz

# add the router
ovn-nbctl lr-add tenant1

# create router port for the connection to dmz
ovn-nbctl lrp-add tenant1 tenant1-dmz 02:ac:10:ff:01:29 172.16.255.129/26

ovn-nbctl lsp-add dmz dmz-vm1
ovn-nbctl lsp-set-addresses dmz-vm1 "02:ac:10:ff:01:30 172.16.255.130"
ovn-nbctl lsp-set-port-security dmz-vm1 "02:ac:10:ff:01:30 172.16.255.130"

dmzDhcp=$(ovn-nbctl create DHCP_Options cidr=172.16.255.128/26 \
options=\"server_id\=\"172.16.255.129\" \"server_mac\=\"02:ac:10:ff:01:29\" \
\"lease_time\=\"3600\" \"router\=\"172.16.255.129\"")
echo $dmzDhcp

ovn-nbctl lsp-set-dhcpv4-options dmz-vm1 $dmzDhcp
ovn-nbctl lsp-get-dhcpv4-options dmz-vm1

ip netns add vm1
ovs-vsctl add-port br-int vm1 -- set interface vm1 type=internal
ip link set vm1 address 02:ac:10:ff:01:30
ip link set vm1 netns vm1
ovs-vsctl set Interface vm1 external_ids:iface-id=dmz-vm1
ip netns exec vm1 dhclient vm1
ip netns exec vm1 ip addr show vm1
ip netns exec vm1 ip route show
```

## OVN Trace

ovn-trace 是很好的辅助工具.

```

$ sudo ovn-trace --minimal sw0 'inport == "sw0-port1" && eth.src == 00:00:00:00:00:01
&& eth.dst == 00:00:00:00:00:02'
# reg14=0x1,vlan_tci=0x0000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,dl_type=
0x0000
output("sw0-port2");

$ ovn-trace --summary sw0 'inport == "sw0-port1" && eth.src == 00:00:00:00:00:01 && et
h.dst == 00:00:00:00:00:02'
# reg14=0x1,vlan_tci=0x0000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,dl_type=
0x0000
ingress(dp="sw0", inport="sw0-port1") {
    next;
    outport = "sw0-port2";
    output;
    egress(dp="sw0", inport="sw0-port1", outport="sw0-port2") {
        output;
        /* output to "sw0-port2", type "" */;
    };
};

$ ovn-trace --detailed sw0 'inport == "sw0-port1" && eth.src == 00:00:00:00:00:01 && e
th.dst == 00:00:00:00:00:02'
# reg14=0x1,vlan_tci=0x0000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,dl_type=
0x0000

ingress(dp="sw0", inport="sw0-port1")
-----
0. ls_in_port_sec_l2 (ovn-northd.c:2979): inport == "sw0-port1" && eth.src == {00:00:
00:00:00:01}, priority 50, uuid 50dd1db0
    next;
13. ls_in_l2_lkup (ovn-northd.c:3274): eth.dst == 00:00:00:00:00:02, priority 50, uuid
faab2844
    outport = "sw0-port2";
    output;

egress(dp="sw0", inport="sw0-port1", outport="sw0-port2")
-----
8. ls_out_port_sec_l2 (ovn-northd.c:3399): outport == "sw0-port2" && eth.dst == {00:0
0:00:00:00:02}, priority 50, uuid 4b4d798e
    output;
    /* output to "sw0-port2", type "" */

$ ovn-trace --detailed sw0 'inport == "sw0-port1" && eth.src == 00:00:00:00:00:ff && e
th.dst == 00:00:00:00:00:02'
# reg14=0x1,vlan_tci=0x0000,dl_src=00:00:00:00:00:ff,dl_dst=00:00:00:00:00:02,dl_type=
0x0000

ingress(dp="sw0", inport="sw0-port1")
-----
0. ls_in_port_sec_l2: no match (implicit drop)

```

更多OVN的使用方法可以参考[这里](#)。

# OVN HA

目前，OVS支持主从模式的高可用。

## Active-Backup

在启动ovsdb-server时，可以设置[主从同步选项](#)：

```
Syncing Options

The following options allow ovsdb-server to synchronize its databases
with another running ovsdb-server.

--sync-from=server
    Sets up ovsdb-server to synchronize its databases with the databases
    in server, which must be an active connection method in one of the forms
    documented in ovsdb-client(1). Every transaction committed by server
    will be replicated to ovsdb-server. This option makes ovsdb-server start
    as a backup server; add --active to make it start as an active server.

--sync-exclude-tables=db:table[,db:table]...
    Causes the specified tables to be excluded from replication.

--active
    By default, --sync-from makes ovsdb-server start up as a backup
    for server. With --active, however, ovsdb-server starts as an active
    server. Use this option to allow the syncing options to be specified
    using command line options, yet start the server, as the default, active
    server. To switch the running server to backup mode, use ovs-appctl(1) to
    execute the ovsdb-server/connect-active-ovsdb-server command.
```

注意，这里的配置是静态的，主ovsdb-server出现问题时，从并不会自动恢复。这时可以借助Pacemaker来实现自动故障恢复：

After creating a pacemaker cluster, use the following commands to create one active and multiple backup servers for OVN databases:

```
$ pcs resource create ovndb_servers ocf:ovn:ovndb-servers \
  master_ip=x.x.x.x \
  ovn_ctl=<path of the ovn-ctl script> \
  op monitor interval="10s" \
  op monitor role=Master interval="15s"
$ pcs resource master ovndb_servers-master ovndb_servers \
  meta notify="true"
```

The `master_ip` and `ovn_ctl` are the parameters that will be used by the OCF script.

- `ovn_ctl` is optional, if not given, it assumes a default value of `/usr/share/openvswitch/scripts/ovn-ctl`.
- `master_ip` is the IP address on which the active database server is expected to be listening, the slave node uses it to connect to the master node. You can add the optional parameters ‘`nb_master_port`’, ‘`nb_master_protocol`’, ‘`sb_master_port`’, ‘`sb_master_protocol`’ to set the protocol and port.

Whenever the active server dies, pacemaker is responsible to promote one of the backup servers to be active. Both ovn-controller and ovn-northd needs the ip-address at which the active server is listening. With pacemaker changing the node at which the active server is run, it is not efficient to instruct all the ovn-controllers and the ovn-northd to listen to the latest active server’s ip-address.

This problem can be solved by using a native ocf resource agent `ocf:heartbeat:IPAddr2`. The `IPAddr2` resource agent is just a resource with an ip-address. When we colocate this resource with the active server, pacemaker will enable the active server to be connected with a single ip-address all the time. This is the ip-address that needs to be given as the parameter while creating the `ovndb_servers` resource.

Use the following command to create the `IPAddr2` resource and colocate it with the active server:

```
$ pcs resource create VirtualIP ocf:heartbeat:IPAddr2 ip=x.x.x.x \
  op monitor interval=30s
$ pcs constraint order promote ovndb_servers-master then VirtualIP
$ pcs constraint colocation add VirtualIP with master ovndb_servers-master \
  score=INFINITY
```

主从同步的实现方法可见[OVSDP Replication Implementation](#)。

## Active-Active

OVN控制平面的Active-Active高可用还在开发中，预计会借鉴etcd的方式，基于Raft算法实现。

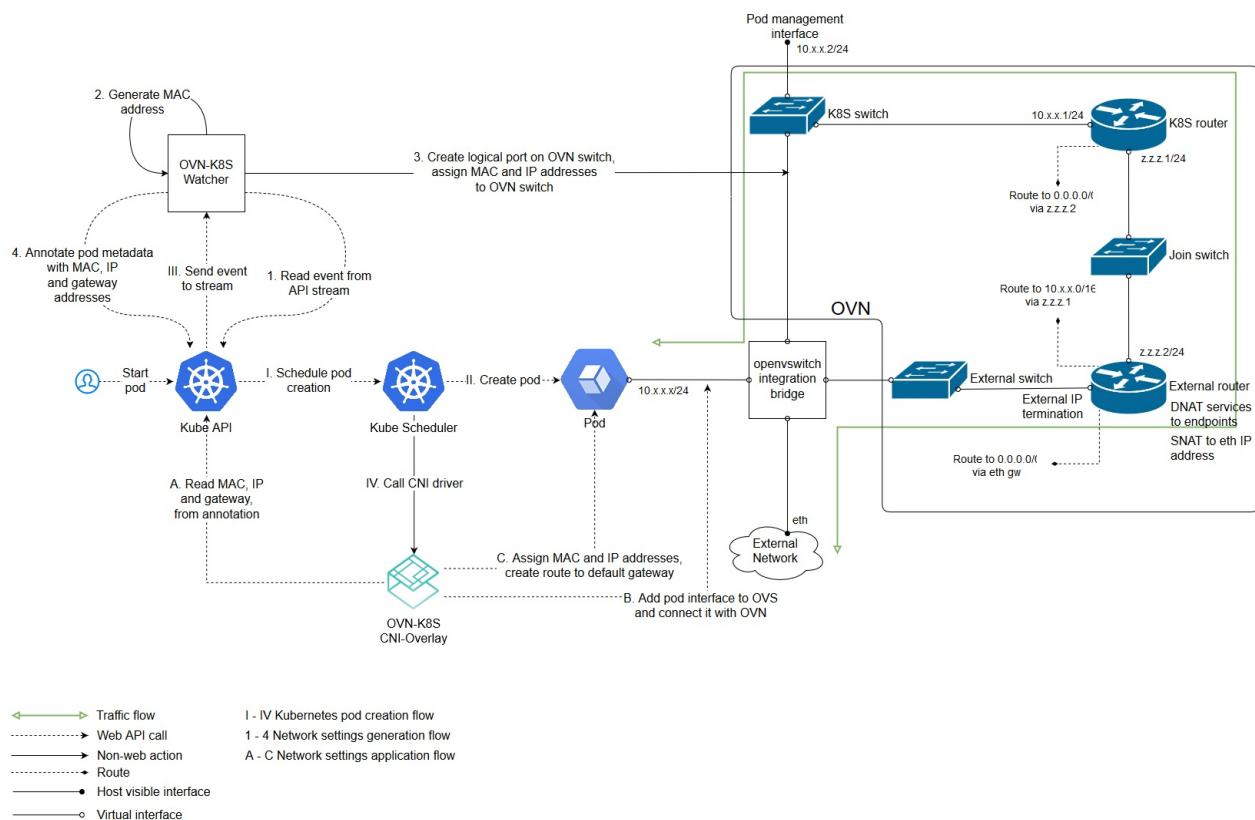
- <https://github.com/blp/ovs-reviews/tree/raft3>
- <http://docs.openvswitch.org/en/latest/topics/high-availability/>
- <http://galsagie.github.io/2015/08/03/df-distributed-db/>

# OVN Kubernetes插件

[ovn-kubernetes](#)提供了一个ovs OVN网络插件，支持underlay和overlay两种模式。

- underlay：容器运行在虚拟机中，而ovs则运行在虚拟机所在的物理机上，OVN将容器网络和虚拟机网络连接在一起
- overlay：OVN通过logical overlay network连接所有节点的容器，此时ovs可以直接运行在物理机或虚拟机上

## Overlay模式



(图片来自<https://imgur.com/i7sci9O>)

## 配置master

```
ovs-vsctl set Open_vSwitch . external_ids:k8s-api-server="127.0.0.1:8080"
ovn-k8s-overlay master-init \
--cluster-ip-subnet="192.168.0.0/16" \
--master-switch-subnet="192.168.1.0/24" \
--node-name="kube-master"
```

## 配置Node

```
ovs-vsctl set Open_vSwitch . \
    external_ids:k8s-api-server="$K8S_API_SERVER_IP:8080"

ovs-vsctl set Open_vSwitch . \
    external_ids:k8s-api-server="https://$K8S_API_SERVER_IP" \
    external_ids:k8s-ca-certificate="$CA_CRT" \
    external_ids:k8s-api-token="$API_TOKEN"

ovn-k8s-overlay minion-init \
    --cluster-ip-subnet="192.168.0.0/16" \
    --minion-switch-subnet="192.168.2.0/24" \
    --node-name="kube-minion1"
```

## 配置网关Node (可以用已有的Node或者单独的节点)

选项一：外网使用单独的网卡 eth1

```
ovs-vsctl set Open_vSwitch . \
    external_ids:k8s-api-server="$K8S_API_SERVER_IP:8080"
ovn-k8s-overlay gateway-init \
    --cluster-ip-subnet="192.168.0.0/16" \
    --physical-interface eth1 \
    --physical-ip 10.33.74.138/24 \
    --node-name="kube-minion2" \
    --default-gw 10.33.74.253
```

选项二：外网网络和管理网络共享同一个网卡，此时需要将该网卡添加到网桥中，并迁移IP和路由

```

# attach eth0 to bridge breth0 and move IP/routes
ovn-k8s-util nics-to-bridge eth0

# initialize gateway
ovs-vsctl set Open_vSwitch . \
    external_ids:k8s-api-server="$K8S_API_SERVER_IP:8080"
ovn-k8s-overlay gateway-init \
    --cluster-ip-subnet="$CLUSTER_IP_SUBNET" \
    --bridge-interface breth0 \
    --physical-ip "$PHYSICAL_IP" \
    --node-name="$NODE_NAME" \
    --default-gw "$EXTERNAL_GATEWAY"

# Since you share a NIC for both mgmt and North-South connectivity, you will
# have to start a separate daemon to de-multiplex the traffic.
ovn-k8s-gateway-helper --physical-bridge=breth0 --physical-interface=eth0 \
    --pidfile --detach

```

## 启动ovn-k8s-watcher

ovn-k8s-watcher监听Kubernetes事件，并创建逻辑端口和负载均衡。

```

ovn-k8s-watcher \
    --overlay \
    --pidfile \
    --log-file \
    -vfile:info \
    -vconsole:emer \
    --detach

```

## CNI插件原理

### ADD操作

- 从 ovn annotation 获取ip/mac/gateway
- 在容器netns中配置接口和路由
- 添加OVS端口

```

ovs-vsctl add-port br-int veth_outside \
    --set interface veth_outside \
        external_ids:attached_mac=mac_address \
        external_ids:iface-id=namespace_pod \
        external_ids:ip_address=ip_address

```

### DEL操作

```
ovs-vsctl del-port br-int port
```

## Underlay模式

暂未实现。

## 参考文档

- <https://github.com/openvswitch/ovn-kubernetes>

# OVN docker插件

```
# start docker
docker daemon --cluster-store=consul://127.0.0.1:8500 \
--cluster-advertise=$HOST_IP:0

# start north
/usr/share/openvswitch/scripts/ovn-ctl start_northd
ovn-nbctl set-connection ptcp:6641
ovn-sbctl set-connection ptcp:6642

# start south
ovs-vsctl set Open_vSwitch . \
    external_ids:ovn-remote="tcp:$CENTRAL_IP:6642" \
    external_ids:ovn-nb="tcp:$CENTRAL_IP:6641" \
    external_ids:ovn-encap-ip=$LOCAL_IP \
    external_ids:ovn-encap-type="$ENCAP_TYPE"
/usr/share/openvswitch/scripts/ovn-ctl start_controller

# start openvswitch plugin
pip install Flask
PYTHONPATH=$OVS_PYTHON_LIBS_PATH ovn-docker-overlay-driver --detach

# create docker network
docker network create -d openvswitch --subnet=192.168.1.0/24 foo
```

## Workflow

### Initialize ovn bridge

```
ovs-vsctl --timeout=5 -vconsole:off -- --may-exist add-br br-int \
-- set bridge br-int external_ids:bridge-id=br-int \
other-config:disable-in-band=true fail-mode=secure

ovs-vsctl --timeout=5 -vconsole:off -- get Open_vSwitch . external_ids:ovn-nb
ovs-vsctl --timeout=5 -vconsole:off -- set open_vswitch . external_ids:ovn-bridge=br-i
nt
```

### Create network

```

nid="red-net"
ovn-nbctl ls-add $nid -- set Logical_Switch $nid external_ids:subnet=10.160.0.0/24 ext
ernal_ids:gateway_ip=10.160.0.1
ovn-nbctl show

```

## Create container

```

nid="red-net"
eid="blue-container"
ip="10.160.0.2"
mac="02:38:e1:a2:28:38"
ovn-nbctl lsp-add $nid $eid
ovn-nbctl lsp-set-addresses $eid "$mac $ip"

ip netns add $eid
ip link add veth_inside type veth peer name veth_outside
ip link set dev veth_inside address $mac
ip link set veth_inside netns $eid
ip link set veth_outside up
ip netns exec $eid ip addr add 10.160.0.2/24 dev veth_inside
ip netns exec $eid ip route add default via 10.160.0.1

ovs-vsctl --timeout=5 -vconsole:off \
    -- add-port br-int veth_outside \
    -- set interface veth_outside \
        external_ids:attached-mac=$mac \
        external_ids:iface-id=$eid \
        external_ids:vm-id=$eid \
        external_ids:iface-status=active

```

## Get endpoint status

```
ovn-nbctl --if-exists get Logical_Switch_Port $eid addresses
```

## Delete container

```

ip netns del $eid
ip link delete veth_outside
ovs-vsctl --if-exists del-port veth_outside
ovn-nbctl lsp-del $eid

```

## Delete network

```
ovn-nbctl ls-del red-net
```

## 参考文档

- <http://docs.openvswitch.org/en/latest/howto/docker/>
- <http://dockone.io/article/1200>

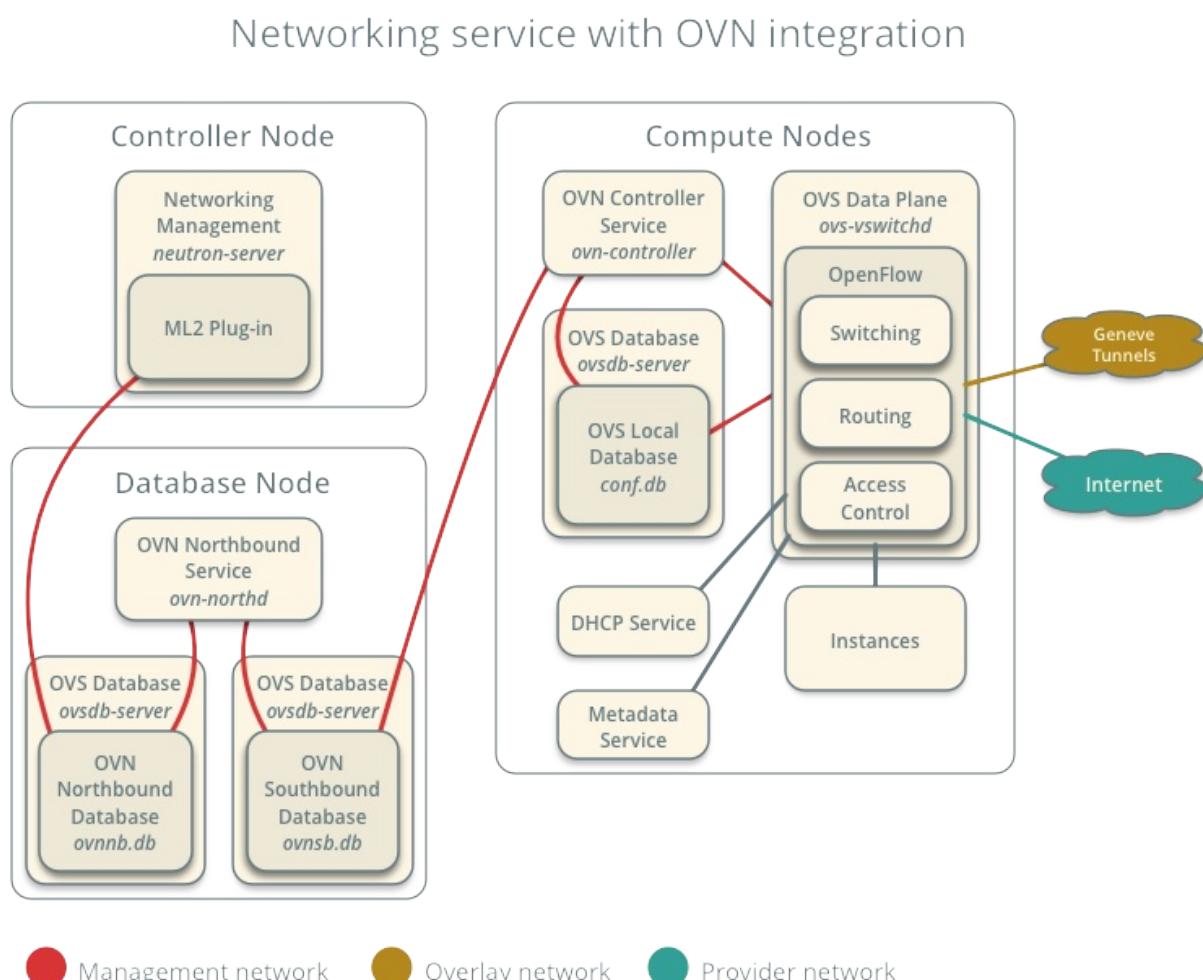
# OVN OpenStack

OpenStack [networking-ovn](#) 项目为 Neutron 提供了一个基于 ML2 的 OVN 插件，它使用 OVN 组件代替了各种 Neutron 的 Python agent，也不再使用 RabbitMQ，而是基于 OVN 数据库进行通信：使用 OVSDB 协议来把用户的配置写在 Northbound DB 里面，ovn-northd 监听到 Northbound DB 配置发生改变，然后把配置翻译到 Southbound DB 里面，ovn-controller 注意到 Southbound DB 数据的变化，然后更新本地的流表。

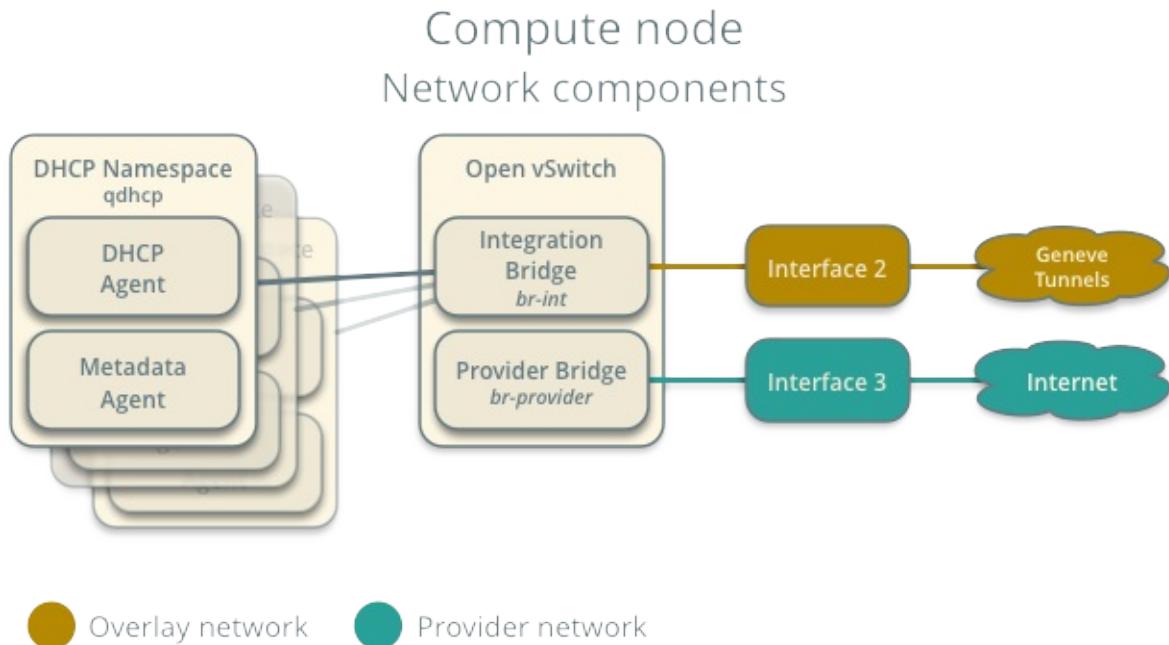
OVN 里面报文的处理都是通过 OVS OpenFlow 流表来实现的，而在 Neutron 里面二层报文处理是通过 OVS OpenFlow 流表来实现，三层报文处理是通过 Linux TCP/IP 协议栈来实现。

## 架构

网络节点



而计算节点包括以下服务



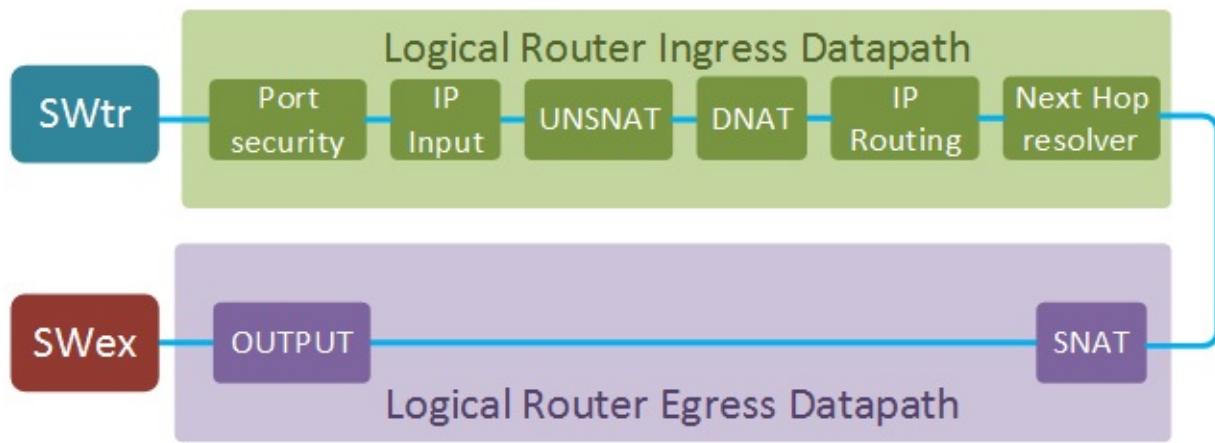
## 实现原理

### 安全组

OVN 的 security group 每创建一个 neutron port，只需要把 tap port 连到 OVS bridge（默认是 br-int），不用像现在 Neutron 那样创建那么多 network device，大大减少了跳数。更重要的是，OVN 的 security group 是用到了 OVS 的 conntrack 功能，可以直接根据连接状态进行匹配，而不是匹配报文的字段，提高了流表的查找效率，还可以做有状态的防火墙和 NAT。OVS 的 conntrack 是用 Linux kernel 的 netfilter 来做的，他调用 netfilter userspace netlink API 把来报文送给 Linux kernel 的 netfilter connection tracker 模块进行处理，这个模块给每个连接维护一个连接状态表，记录这个连接的状态，OVS 获取连接状态，Openflow flow 可以 match 这些连接状态。

## OVN L3

Neutron 的三层功能主要有路由，SNAT 和 Floating IP（也叫 DNAT），它是通过 Linux kernel 的 namespace 来实现的，每个路由器对应一个 namespace，利用 Linux TCP/IP 协议栈来做路由转发。OVN 支持原生的三层功能，不需要借助 Linux TCP/IP stack，用 OpenFlow 流表来实现路由查找，ARP 查找，TTL 和 MAC 地址的更改。OVN 的路由也是分布式的，路由器在每个计算节点上都有实例，有了 OVN 之后，不需要 Neutron L3 agent 了和 DVR 了。



比如SNAT和DNAT的流表为

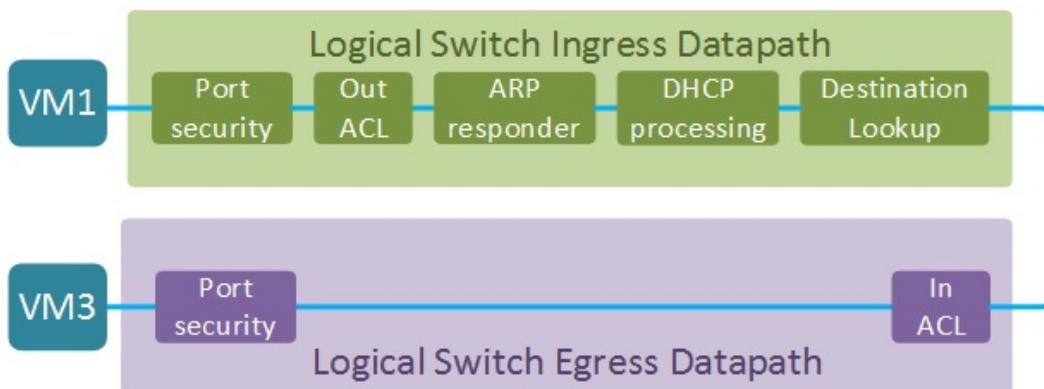
```
# SNAT
table=0 (lr_out_snat), priority=25, match=(ip && ip4.src == 10.0.0.0/24), action=(ct_snat(169.254.0.54);)

# UNSNAT
table=3 (lr_in_unsnat), priority=100, match=(ip && ip4.dst == 169.254.0.54), action=(ct_snat; next;)

# DNAT
table=4 (lr_in_dnac), priority=100, match=(ip && ip4.dst == 169.254.0.52), action=(flags.loopback = 1; ct_dnat(10.0.0.5));
```

## OVN L2

OVN的L2功能都是基于OpenFlow流表实现的，包括Port Security、Egress ACL、ARP Responder、DHCP、Destination Lookup、Ingress ACL等。



## 参考文档

- networking-ovn reference architecture
- 如何借助 OVN 来提高 OVS 在云计算环境中的性能

- [OpenStack SDN With OVN](#)

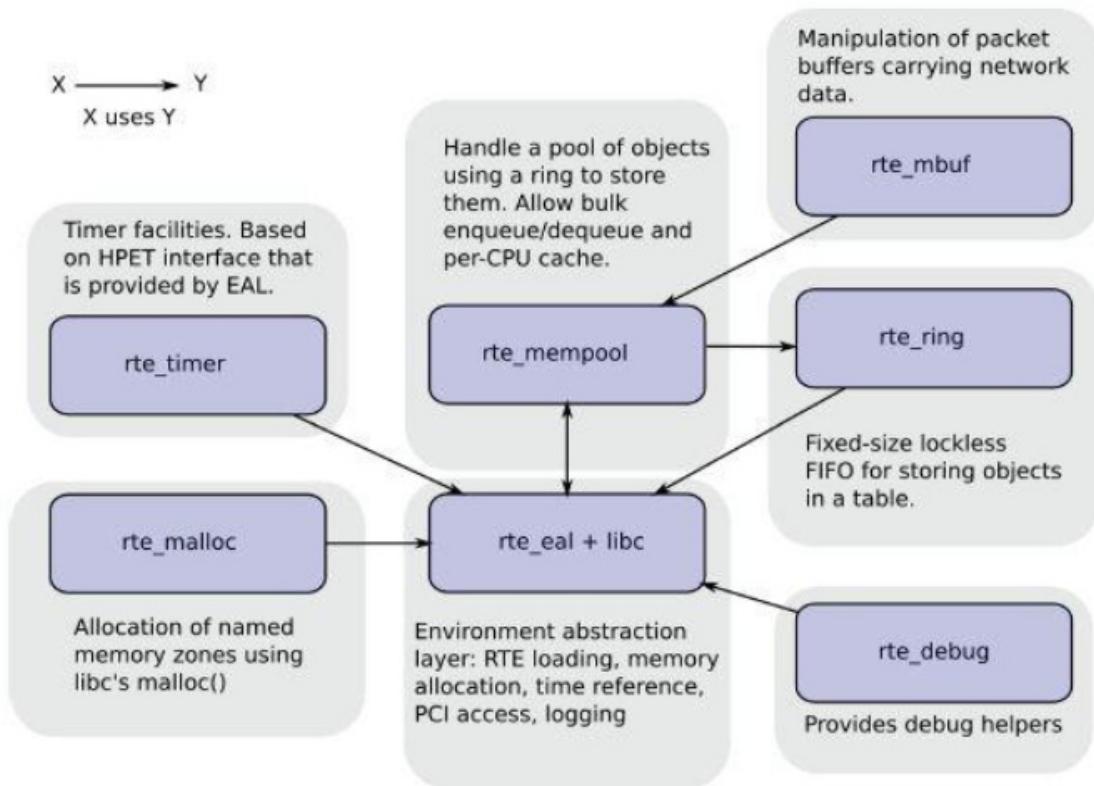
# DPDK

Intel DPDK全称Intel Data Plane Development Kit，是intel提供的数据平面开发工具集，为Intel architecture (IA) 处理器架构下用户空间高效的数据包处理提供库函数和驱动的支持，它不同于Linux系统以通用性设计为目的，而是专注于网络应用中数据包的高性能处理。

DPDK应用程序是运行在用户空间上利用自身提供的数据平面库来收发数据包，绕过了Linux内核协议栈对数据包处理过程。Linux内核将DPDK应用程序看作是一个普通的用户态进程，包括它的编译、连接和加载方式和普通程序没有什么两样。DPDK程序启动后只能有一个主线程，然后创建一些子线程并绑定到指定CPU核心上运行。

## 基本组件

### Relationship of DPDK Libraries



- EAL (Environment Abstraction Layer) 即环境抽象层，为应用提供了一个通用接口，隐藏了与底层库与设备打交道的相关细节。EAL实现了DPDK运行的初始化工作，基于大页表的内存分配，多核亲缘性设置，原子和锁操作，并将PCI设备地址映射到用户空间，方便应用程序访问。
- Buffer Manager API通过预先从EAL上分配固定大小的多个内存对象，避免了在运行过程

中动态进行内存分配和回收来提高效率，常常用作数据包buffer来使用。

- Queue Manager API以高效的方式实现了无锁的FIFO环形队列，适合与一个生产者多个消费者、一个消费者多个生产者模型来避免等待，并且支持批量无锁的操作。
- Flow Classification API通过Intel SSE基于多元组实现了高效的hash算法，以便快速的将数据包进行分类处理。该API一般用于路由查找过程中的最长前缀匹配中，安全产品中根据Flow五元组来标记不同用户的场景也可以使用。
- PMD则实现了Intel 1GbE、10GbE和40GbE网卡下基于轮询收发包的工作模式，大大加速网卡收发包性能。

DPDK核心思想：

- PMD: DPDK针对Intel网卡实现了基于轮询方式的PMD（Poll Mode Drivers）驱动，该驱动由API、用户空间运行的驱动程序构成，该驱动使用无中断方式直接操作网卡的接收和发送队列（除了链路状态通知仍必须采用中断方式以外）。目前PMD驱动支持Intel的大部分1G、10G和40G的网卡。PMD驱动从网卡上接收到数据包后，会直接通过DMA方式传输到预分配的内存中，同时更新无锁环形队列中的数据包指针，不断轮询的应用程序很快就能感知收到数据包，并在预分配的内存地址上直接处理数据包，这个过程非常简洁。如果要是让Linux来处理收包过程，首先网卡通过中断方式通知协议栈对数据包进行处理，协议栈先会对数据包进行合法性进行必要的校验，然后判断数据包目标是否本机的socket，满足条件则会将数据包拷贝一份向上递交给用户socket来处理，不仅处理路径冗长，还需要从内核到应用层的一次拷贝过程。
- hugetlbf: 这样有两个好处：第一是使用hugepage的内存所需的页表项比较少，对于需要大量内存的进程来说节省了很多开销，像oracle之类的大型数据库优化都使用了大页面配置；第二是TLB冲突概率降低，TLB是cpu中单独的一块高速cache，采用hugepage可以大大降低TLB miss的开销。DPDK目前支持了2M和1G两种方式的hugepage。通过修改默认/etc/grub.conf中hugepage配置为“default\_hugepagesz=1G hugepagesz=1G hugepages=32 isolcpus=0-22”，然后通过mount -t hugetlbf nodev /mnt/huge就将hugepage文件系统hugetlbf挂载在/mnt/huge目录下，然后用户进程就可以使用mmap映射hugepage目标文件来使用大页面了。测试表明应用使用大页表比使用4K的页表性能提高10%~15%。
- CPU亲缘性：多核则是每个CPU核一个线程，核心之间访问数据无需上锁。为了最大限度减少线程调度的资源消耗，需要将Linux绑定在特定的核上，释放其余核心来专供应用程序使用。同时还需要考虑CPU特性和系统是否支持NUMA架构，如果支持的话，不同插槽上CPU的进程要避免访问远端内存，尽量访问本端内存。
- 减少内存访问：少用数组和指针，多用局部变量；少用全局变量；一次多访问一些数据；自己管理内存分配；进程间传递指针而非整个数据块
- Cache有效性得益于空间局部性（附近的数据也会被用到）和时间局部性（今后一段时间内会被多次访问）原理，通过合理的使用cache，能够使得应用程序性能得到大幅提升
- 避免False Sharing: 多核CPU中每个核都拥有自己的L1/L2 cache，当运行多线程程序时，尽管算法上不需要共享变量，但实际执行中两个线程访问同一cache line的数据时就会引起冲突，每个线程在读取自己的数据时也会把别人的数据读进来，这时一个核

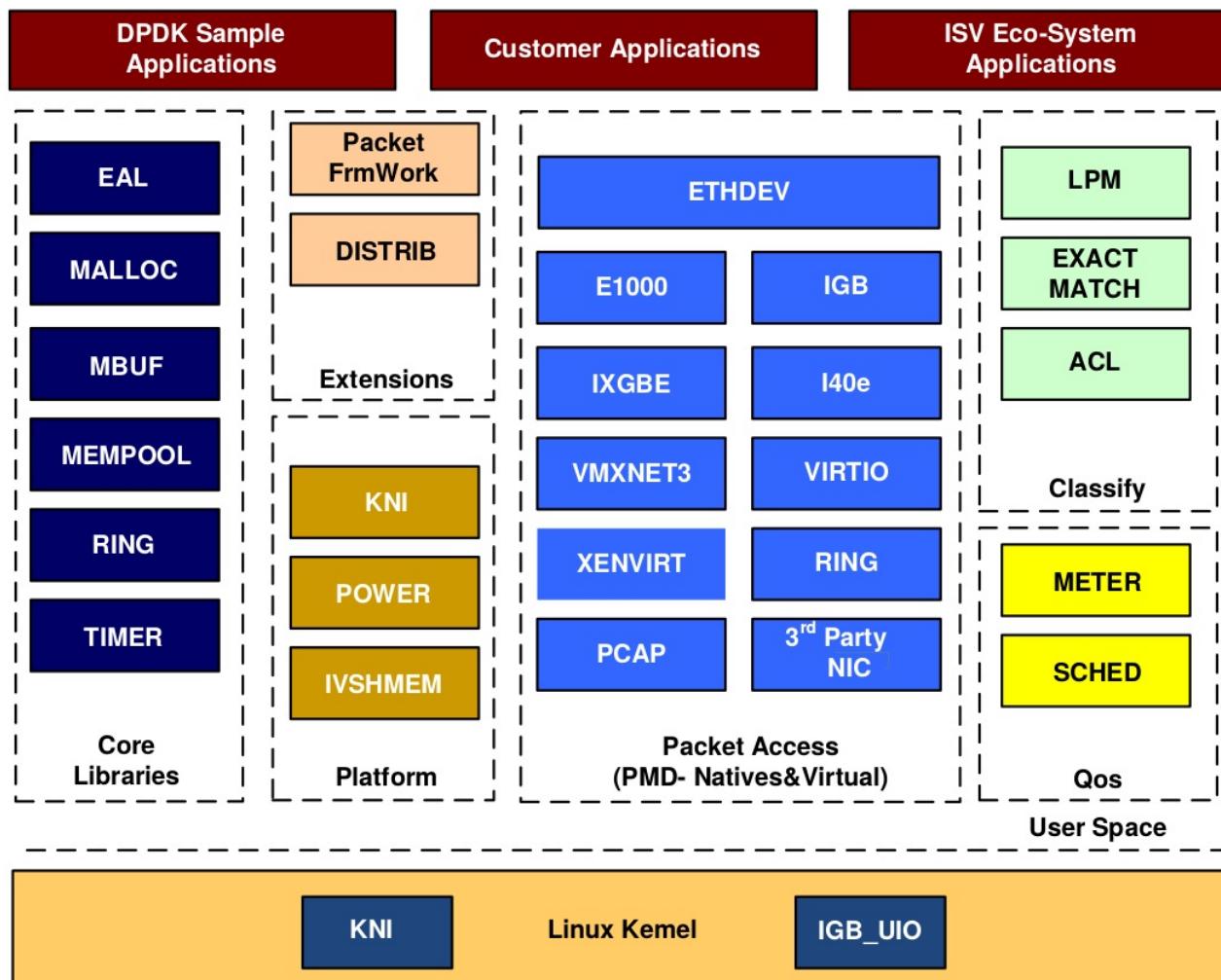
修改改变量，CPU的cache一致性算法会迫使另一个核的cache中包含该变量所在的cache line无效，这就产生了**false sharing**（伪共享）问题。**Falsing sharing**会导致大量的cache冲突，应该尽量避免。访问全局变量和动态分配内存是**false sharing**问题产生的根源，当然访问在内存中相邻的但完全不同的全局变量也可能导致**false sharing**，多使用线程本地变量是解决**false sharing**的根源办法。

- 内存对齐：根据不同存储硬件的配置来优化程序，性能也能够得到极大的提升。在硬件层次，确保对象位于不同channel和rank的起始地址，这样能保证对象并行加载。字节对齐：众所周知，内存最小的存储单元为字节，在32位CPU中，寄存器也是32位的，为了保证访问更加高效，在32位系统中变量存储的起始地址默认是4的倍数（64位系统则是8的倍数），定义一个32位变量时，只需要一次内存访问即可将变量加载到寄存器中，这些工作都是编译器完成的，不需人工干预，当然我们可以使用`__attribute__((aligned(n)))`来改变对齐的默认值。
- cache对齐，这也是程序开发中需要关注的。**Cache line**是CPU从内存加载数据的最小单位，一般L1 cache的cache line大小为64字节。如果CPU访问的变量不在**cache**中，就需要先从内存调入到**cache**，调度的最小单位就是**cache line**。因此，内存访问如果没有按照**cache line**边界对齐，就会多读写一次内存和**cache**了。
- NUMA: NUMA系统节点一般是由一组CPU和本地内存组成。NUMA调度器负责将进程在同一节点的CPU间调度，除非负载太高，才迁移到其它节点，但这会导致数据访问延时增大。
- 减少进程上下文切换：需要了解哪些场景会触发CS操作。首先就介绍的就是不可控的场景：进程时间片到期；更高优先级进程抢占CPU。其次是可控场景：休眠当前进程(`pthread_cond_wait`)；唤醒其它进程(`pthread_cond_signal`)；加锁函数、互斥量、信号量、`select`、`sleep`等非常多函数都是可控的。对于可控场景是在应用编程需要考虑的问题，只要程序逻辑设计合理就能较少CS的次数。对于不可控场景，首先想到的是适当减少活跃进程或线程数量，因此保证活跃进程数目不超过CPU个数是一个明智的选择；然后有些场景下，我们并不知道有多少个活跃线程的时候怎么来保证上下文切换次数最少呢？这是我们就要使用线程池模型：让每个线程工作前都持有带计数器的信号量，在信号量达到最大值之前，每个线程被唤醒时仅进行一次上下文切换，当信号量达到最大值时，其它线程都不会再竞争资源了。
- 分组预测机制，如果预测的一个分支指令加入流水线，之后却发现它是错误的分支，处理器要回退该错误预测执行的工作，再用正确的指令填充流水线。这样一个错误的预测会严重浪费时钟周期，导致程序性能下降。《计算机体系结构：量化研究方法》指出分支指令产生的性能影响为10%~30%，流水线越长，性能影响越大。Core i7和Xen等较新的处理器当分支预测失效时无需刷新全部流水，当错误指令加载和计算仍会导致一部分开销。分支预测中最核心的是分支目标缓冲区（Branch Target Buffer，简称BTB），每条分支指令执行后，都会BTB都会记录指令的地址及它的跳转信息。BTB一般比较小，并且采用Hash表的方式存入，在CPU取值时，直接将PC指针和BTB中记录对比来查找，如果找到了，就直接使用预测的跳转地址，如果没有记录，必须通过**cache**或内存取下一条指令。
- 利用流水线并发：像Pentium处理器就有UV两条流水，并且可以独自独立读写缓存，循环

2可以将两条指令安排在不同流水线上执行，性能得到极大提升。另外两条流水线是非对称的，简单指令（mpv,add,push,inc,cmp,lea等）可以在两条流水上并行执行、位操作和跳转操作并发的前提是在特定流水线上工作、而某些复杂指令却只能独占CPU。

- 为了利用空间局部性，同时也为了覆盖数据从内存传输到CPU的延迟，可以在数据被用到之前就将其调入缓存，这一技术称为预取Prefetch，加载整个cache即是一种预取。CPU在进行计算过程中可以并行的对数据进行预取操作，因此预取使得数据/指令加载与CPU执行指令可以并行进行。

## 架构

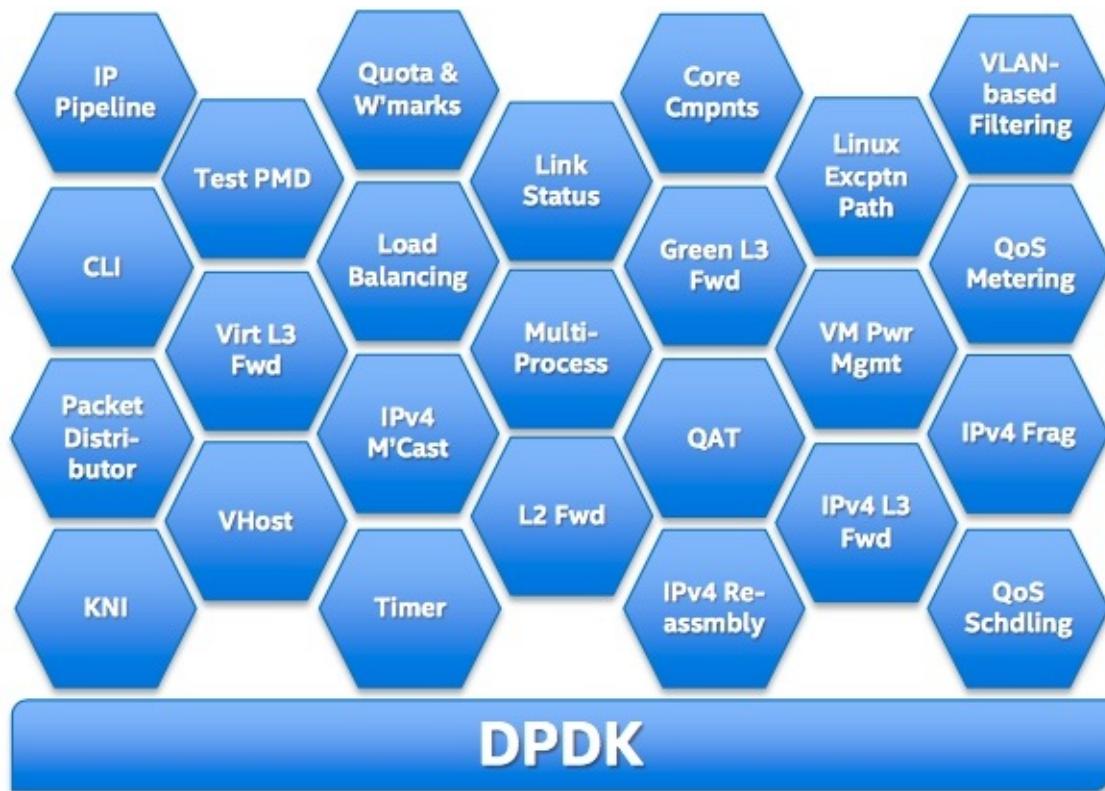


在最底部的内核态(Linux Kernel)DPDK 有两个模块:KNI 与 IGB\_UIO。其中,KNI 提供给用户一个使用 Linux 内核态的协议栈,以及传统的 Linux 网络工具(如ethtool, ifconfig)。

IGB\_UIO(igb\_uio.ko 和 kni.ko. IGB\_UIO)则借助了 UIO 技术,在初始化过程中将网卡硬件寄存器映射到用户态。

DPDK 的上层用户态由很多库组成,主要包括核心部件库(Core Libraries)、平台相关模块(Platform)、网卡轮询模式驱动模块(PMD-Natives& Virtual)、QoS 库、报文转发分类算法(Classify)等几大类,用户应用程序可以使用这些库进行二次开发.

## 应用



- SPDK
- OPNFV
- Open vSwitch for NFV
- Data Plane Acceleration (DPACC)
- OVS-DPDK
- VPP和TLDK
- Seastar : TCP/IP协议栈的实现只适合在内网运行，公网复杂的网络环境会导致它出现各种问题
- F-Stack : 粘合了DPDK,FreeBSD协议栈，POSIX API，同时支持coroutine

## 参考文档

注：本章内容大部分整理自《深入浅出DPDK》的读书笔记。

- 《深入浅出DPDK》
- <http://dpdk.org> and <http://dpdk.org/doc/guides/>
- <http://intel.com/go/dpdk>
- <https://fd.io> and [https://wiki.fd.io/view/Main\\_Page](https://wiki.fd.io/view/Main_Page)
- <https://github.com/lagopus/lagopus>
- Data Plane Performance Demonstrators (DPPD)



# DPDK简介

## 主流包处理硬件平台

- 硬件加速器：ASIC、FPGA
- 网络处理器
- 多核处理器

## 传统Linux网络驱动的问题

- 中断开销突出，大量数据到来会触发频繁的中断（softirq）开销导致系统无法承受
- 需要把包从内核缓冲区拷贝到用户缓冲区，带来系统调用和数据包复制的开销
- 对于很多网络功能节点来说，TCP/IP协议并非是数据转发环节所必需的
- NAPI/Netmap等虽然减少了内核到用户空间的数据拷贝，但操作系统调度带来的cache替换也会对性能产生负面影响

## dpdk最佳实践

- PMD用户态驱动: DPDK针对Intel网卡实现了基于轮询方式的PMD（Poll Mode Drivers）驱动，该驱动由API、用户空间运行的驱动程序构成，该驱动使用无中断方式直接操作网卡的接收和发送队列（除了链路状态通知仍必须采用中断方式以外）。目前PMD驱动支持Intel的大部分1G、10G和40G的网卡。PMD驱动从网卡上接收到数据包后，会直接通过DMA方式传输到预分配的内存中，同时更新无锁环形队列中的数据包指针，不断轮询的应用程序很快就能感知收到数据包，并在预分配的内存地址上直接处理数据包，这个过程非常简洁。如果要是让Linux来处理收包过程，首先网卡通过中断方式通知协议栈对数据包进行处理，协议栈先会对数据包进行合法性进行必要的校验，然后判断数据包目标是否本机的socket，满足条件则会将数据包拷贝一份向上递交给用户socket来处理，不仅处理路径冗长，还需要从内核到应用层的一次拷贝过程。
- hugetlbfs: 这样有两个好处：第一是使用hugepage的内存所需的页表项比较少，对于需要大量内存的进程来说节省了很多开销，像oracle之类的大型数据库优化都使用了大页面配置；第二是TLB冲突概率降低，TLB是cpu中单独的一块高速cache，采用hugepage可以大大降低TLB miss的开销。DPDK目前支持了2M和1G两种方式的hugepage。通过修改默认/etc/grub.conf中hugepage配置为“default\_hugepagesz=1G hugepagesz=1G hugepages=32 isolcpus=0-22”，然后通过mount -t hugetlbfs nodev /mnt/huge就将hugepage文件系统hugetlbfs挂在/mnt/huge目录下，然后用户进程就可以使用mmap映射hugepage目标文件来使用大页面了。测试表明应用使用大页表比使用4K的页表性能提高

10%~15%。

- CPU亲缘性和独占: 多核则是每个CPU核一个线程，核心之间访问数据无需上锁。为了最大限度减少线程调度的资源消耗，需要将Linux绑定在特定的核上，释放其余核心来专供应用程序使用。同时还需要考虑CPU特性和系统是否支持NUMA架构，如果支持的话，不同插槽上CPU的进程要避免访问远端内存，尽量访问本端内存。
  - 避免不同核之间的频繁切换，从而避免cache miss和cache write back
  - 避免同一个核内多任务切换开销
- 降低内存访问开销:
  - 借助大页降低TLB miss
  - 利用内存多通道交错访问提高内存访问的有效带宽
  - 利用内存非对称性感知避免额外的访存延迟
  - 少用数组和指针，多用局部变量
  - 少用全局变量
  - 一次多访问一些数据
  - 自己管理内存分配；进程间传递指针而非整个数据块
- Cache有效性得益于空间局部性（附近的数据也会被用到）和时间局部性（今后一段时间内会被多次访问）原理，通过合理的使用cache，能够使得应用程序性能得到大幅提升
- 避免False Sharing: 多核CPU中每个核都拥有自己的L1/L2 cache，当运行多线程程序时，尽管算法上不需要共享变量，但实际执行中两个线程访问同一cache line的数据时就会引起冲突，每个线程在读取自己的数据时也会把别人的cacheline读进来，这时一个核修改改变量，CPU的cache一致性算法会迫使另一个核的cache中包含该变量所在的cache line无效，这就产生了false sharing（伪共享）问题。Falsing sharing会导致大量的cache冲突，应该尽量避免。访问全局变量和动态分配内存是falsesharing问题产生的根源，当然访问在内存中相邻的但完全不同的全局变量也可能导致false sharing，多使用线程本地变量是解决false sharing的根源办法。
- 内存对齐：根据不同存储硬件的配置来优化程序，性能也能够得到极大的提升。在硬件层次，确保对象位于不同channel和rank的起始地址，这样能保证对象并行加载。字节对齐：众所周知，内存最小的存储单元为字节，在32位CPU中，寄存器也是32位的，为了保证访问更加高效，在32位系统中变量存储的起始地址默认是4的倍数（64位系统则是8的倍数），定义一个32位变量时，只需要一次内存访问即可将变量加载到寄存器中，这些工作都是编译器完成的，不需人工干预，当然我们可以使用attribute((aligned(n)))来改变对齐的默认值。
- cache对齐，这也是程序开发中需要关注的。Cache line是CPU从内存加载数据的最小单位，一般L1 cache的cache line大小为64字节。如果CPU访问的变量不在cache中，就需要先从内存调入到cache，调度的最小单位就是cache line。因此，内存访问如果没有按照cache line边界对齐，就会多读写一次内存和cache了。
- NUMA: NUMA系统节点一般是由一组CPU和本地内存组成。NUMA调度器负责将进程在同一节点的CPU间调度，除非负载太高，才迁移到其它节点，但这会导致数据访问延时增大。
- 减少进程上下文切换: 需要了解哪些场景会触发CS操作。首先就介绍的就是不可控的场

景：进程时间片到期；更高优先级进程抢占CPU。其次是可控场景：休眠当前进程(`pthread_cond_wait`)；唤醒其它进程(`pthread_cond_signal`)；加锁函数、互斥量、信号量、`select`、`sleep`等非常多函数都是可控的。对于可控场景是在应用编程需要考虑的问题，只要程序逻辑设计合理就能较少CS的次数。对于不可控场景，首先想到的是适当减少活跃进程或线程数量，因此保证活跃进程数目不超过CPU个数是一个明智的选择；然后有些场景下，我们并不知道有多少个活跃线程的时候怎么来保证上下文切换次数最少呢？这是我们就要使用线程池模型：让每个线程工作前都持有带计数器的信号量，在信号量达到最大值之前，每个线程被唤醒时仅进行一次上下文切换，当信号量达到最大值时，其它线程都不会再竞争资源了。

- 分组预测机制，如果预测的一个分支指令加入流水线，之后却发现它是错误的分支，处理器要回退该错误预测执行的工作，再用正确的指令填充流水线。这样一个错误的预测会严重浪费时钟周期，导致程序性能下降。《计算机体系结构：量化研究方法》指出分支指令产生的性能影响为10%~30%，流水线越长，性能影响越大。Core i7和Xen等较新的处理器当分支预测失效时无需刷新全部流水，当错误指令加载和计算仍会导致一部分开销。分支预测中最核心的是分支目标缓冲区（Branch Target Buffer，简称BTB），每条分支指令执行后，都会BTB都会记录指令的地址及它的跳转信息。BTB一般比较小，并且采用Hash表的方式存入，在CPU取值时，直接将PC指针和BTB中记录对比来查找，如果找到了，就直接使用预测的跳转地址，如果没有记录，必须通过cache或内存取下一条指令。
- 利用流水线并发：像Pentium处理器就有U/V两条流水，并且可以独自独立读写缓存，循环2可以将两条指令安排在不同流水线上执行，性能得到极大提升。另外两条流水线是非对称的，简单指令（`mpv,add,push,inc,cmp,lea`等）可以在两条流水上并行执行、位操作和跳转操作并发的前提是在特定流水线上工作、而某些复杂指令却只能独占CPU。
- 为了利用空间局部性，同时也为了覆盖数据从内存传输到CPU的延迟，可以在数据被用到之前就将其调入缓存，这一技术称为预取Prefetch，加载整个cache即是一种预取。CPU在进行计算过程中可以并行的对数据进行预取操作，因此预取使得数据/指令加载与CPU执行指令可以并行进行。
- 充分挖掘网卡的潜能：借助现代网卡支持的分流（RSS, FDIR）和卸载（TSO, checksum）等特性。

## Cache子系统

- 一级Cache：4个指令周期，分为数据cache和指令cache，一般只有几十KB
- 二级Cache：12个指令周期，几百KB到几MB
- 三级Cache：26-31个指令周期，几MB到几十MB
- TLB Cache：缓存内存中的页表项，减少CPU开销

如何把内存中的内容放到cache中呢？这里需要映射算法和分块机制。当今主流块大小是64字节。

硬件Cache预取（Netburst为例）：

- 只有两次cache miss才能激活预取机制，且2次的内存地址偏差不超过256或512字节
- 一个4KB的page内只定义一条stream
- 能同时独立的追踪8条stream
- 对4KB边界之外不进行预取
- 预取的数据放在二级或三级cache中
- 对strong uncacheable和write combining内存类型不预取

硬件预取不一定能够提升性能，所以DPDK还借助软件预取尽量将数据放到cache中。另外，DPDK在定义数据结构的时候还保证了cache line对齐。

cache一致性

- 原则是避免多个核访问同一个内存地址或数据结构
- 在数据结构上：每个核都有独立的数据结构
- 多个核访问同一个网卡：每个核都创建单独的接收队列和发送队列

## Huge Page

hugetlbfs有两个好处：

- 第一是使用hugepage的内存所需的页表项比较少，对于需要大量内存的进程来说节省了很多开销，像oracle之类的大型数据库优化都使用了大页面配置；
- 第二是TLB冲突概率降低，TLB是cpu中单独的一块高速cache，采用hugepage可以大大降低TLB miss的开销。

DPDK目前支持了2M和1G两种方式的hugepage。通过修改默认/etc/grub.conf中hugepage配置为 default\_hugepagesz=1G hugepagesz=1G hugepages=32 isolcpus=0-22，然后通过 mount -t hugetlbfs nodev /mnt/huge 就将hugepage文件系统hugetlbfs挂在/mnt/huge目录下，然后用户进程就可以使用mmap映射hugepage目标文件来使用大页面了。测试表明应用使用大页表比使用4K的页表性能提高10%-15%。

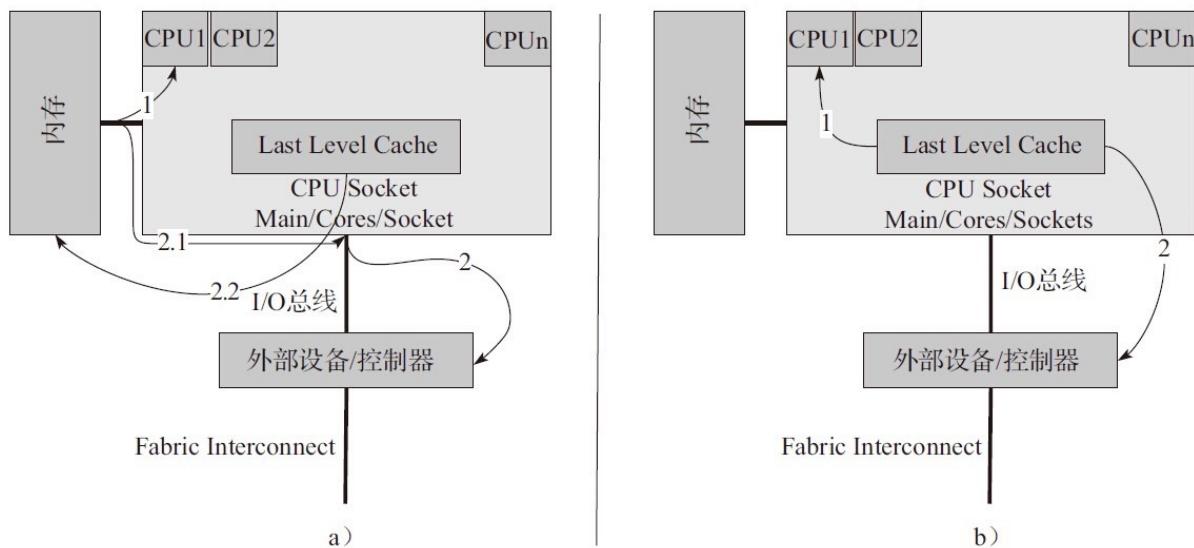
Linux系统启动后预留大页的方法

- 非NUMA系统： echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr\_hugepages
- NUMA系统： echo 1024 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr\_hugepages
- 对于1G的大页，必须在系统启动的时候指定，不能动态预留。

## Data Direct I/O (DDIO)

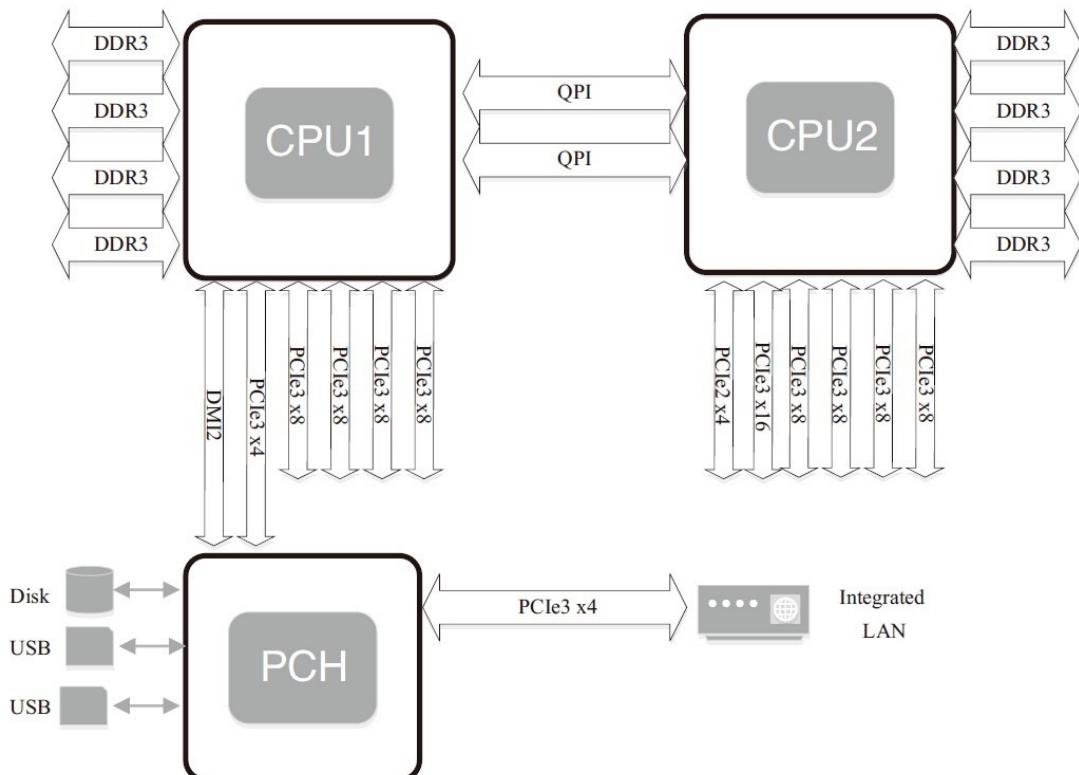
DDIO使得外部网卡和CPU通过LLC cache直接交换数据，绕过了内存，增加了CPU处理报文的速度。

在Intel E5系列产品中，LLC Cache的容量提高到了20MB。



## NUMA

NUMA来源于AMD Opteron微架构，处理器和本地内存之间有更小的延迟和更大的带宽；每个处理器还可以有自己的总线。处理器访问本地的总线和内存时延迟低，而访问远程资源时则要高。



DPDK充分利用了NUMA的特点

- Per-core memory，每个核都有自己的内存，一方面是本地内存的需要，另一方面也是为了cache一致性
- 用本地处理器和本地内存处理本地设备上产生的数据

```
q = rte_zmalloc_socket("fm10k", sizeof(*q), RTE_CACHE_LINE_SIZE, socket_id)
```

CPU核心的几个概念：

- 处理器核数（cpu cores）：每个物理CPU core的个数
- 逻辑处理器核心数（siblings）：单个物理处理器超线程的个数
- 系统物理处理器封装ID（physical id）：也称为socket插槽，物理机处理器封装个数，物理CPU个数
- 系统逻辑处理器ID（processor）：逻辑CPU数，是物理处理器的超线程技术

### CPU亲和性

将进程与CPU绑定，提高了Cache命中率，从而减少内存访问损耗。CPU亲和性的主要应用场景为

- 大量计算场景
- 运行时间敏感、决定性的线程，即实时线程

### 相关工具

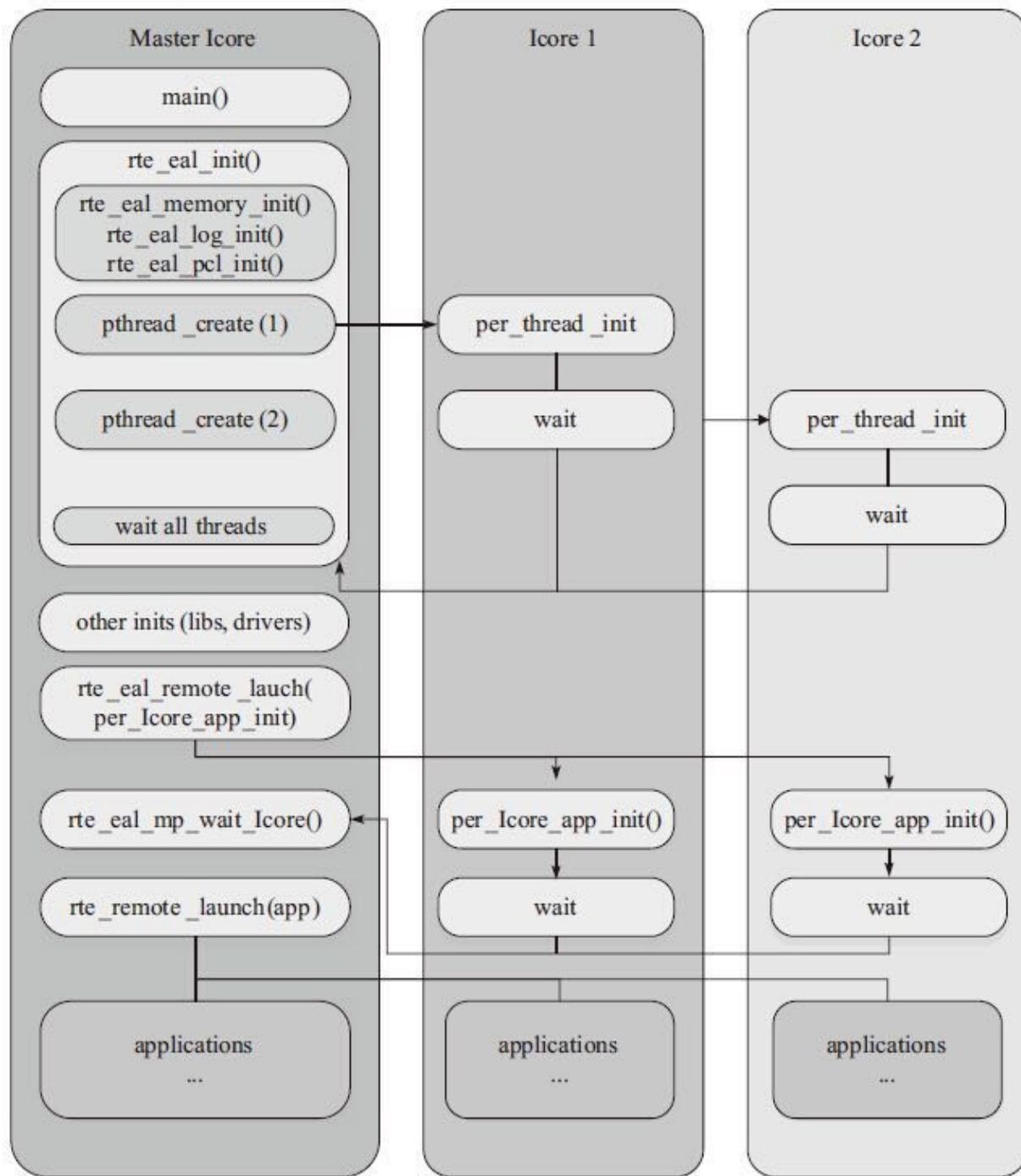
- `sched_set_affinity()`、`sched_get_affinity()` 内核函数
- `taskset`命令
- `isolcpus`内核启动参数：CPU绑定之后依然是有可能发生线程切换，可以借助 `isolcpus=2,3` 将cpu从内核调度系统中剥离。

### DPDK中的CPU亲和性

DPDK中lcore实际上是EAL pthread，每个EAL pthread都有一个Thread Local Storage的 `_lcore_id`，`_lcore_id` 与CPU ID是一致的。注意虽然默认是1:1关系，但可以通过 `--lcores='<lcore_set>@<cpu_set>'` 来指定lcore的CPU亲和性，这样可以不是1:1的，也就是多个lcore还是可以亲和到同一个的核，这就需要注意调度的情况（以非抢占式无锁 `rte_ring` 为例）：

- 单生产者、单消费者模式不受影响
- 多生产者、多消费者模式，调度策略为 `SCHED_OTHER` 时，性能会有所影响
- 多生产者、多消费者模式，调度策略为 `SCHED_FIFO/SCHED_RR`，会产生死锁

而在具体实现流程如下所示：



- DPDK通过读取 /sys/devices/system/cpu/cpuX/ 目录的信息获取CPU的分布情况，将第一核设置为MASTER，并通过 eal\_thread\_set\_affinity() 为每个SLAVE绑定CPU
- 不同模块要调用 rte\_eal\_mp\_remote\_launch() 将自己的回调函数注册到DPDK中  
( lcore\_config[].f )
- 每个核最终调用 eal\_thread\_loop()->回调函数 来执行真正的逻辑

### 指令并发

借助SIMD (Single Instruction Multiple Data，单指令多数据) 可以最大化的利用一级缓存访存的带宽，但对频繁的窄位宽数据操作就有比较大的副作用。DPDK中的 rte\_memcpy() 在 Intel处理器上充分利用了SSE/AVX的特点：优先保证Store指令存储的地址对齐，然后在每个

指令周期指令2条Load的特新弥补一部分非对齐Load带来的性能损失。

# DPDK安装

## 源码安装

```

export RTE_SDK="/usr/src/dpdk"
export DPDK_VERSION="2.2.0"
export RTE_TARGET="x86_64-native-linuxapp-gcc"

##### ubuntu #####
apt-get install -y vim gcc-multilib libfuse-dev linux-source libssl-dev llvm-dev python
autoconf libtool libpciaccess-dev make libcunit1-dev libaio-dev

##### centos #####
yum -y install git cmake gcc autoconf automake device-mapper-devel \
sqlite-devel pcre-devel libsepol-devel libselinux-devel \
automake autoconf gcc make glibc-devel glibc-devel.i686 kernel-devel \
fuse-devel pciutils libtool openssl-devel libpciaccess-devel CUnit-devel libaio-devel
mkdir -p /lib/modules/$(uname -r)
ln -sf /usr/src/kernels/$(uname -r) /lib/modules/$(uname -r)/build

# download dpdk
curl -sSL http://dpdk.org/browse/dpdk/snapshot/dpdk-${DPDK_VERSION}.tar.gz | tar -xz &
& mv dpdk-${DPDK_VERSION} ${RTE_SDK}

# install dpdk
cd ${RTE_SDK}
sed -ri 's,(CONFIG_RTE_BUILD_COMBINE_LIBS=).*,\1y,' config/common_linuxapp
sed -ri 's,(CONFIG_RTE_LIBRTE_VHOST=).*,\1y,' config/common_linuxapp
sed -ri 's,(CONFIG_RTE_LIBRTE_VHOST_USER=).*,\1n,' config/common_linuxapp
sed -ri '/CONFIG_RTE_LIBNAME/a CONFIG_RTE_BUILD_FPIC=y' config/common_linuxapp
make config CC=gcc T=${RTE_TARGET}
make -j `nproc` RTE_KERNELDIR=/lib/modules/$(uname -r)/build
make install
depmod

```

**配置Hugepages:** 添加 "default\_hugepagesz=1G hugepagesz=1G hugepages=4" 到 /etc/grub/default 的 GRUB\_CMDLINE\_LINUX，并执行（测试环境可以配置 GRUB\_CMDLINE\_LINUX="crashkernel=auto rhgb quiet transparent\_hugepage=never default\_hugepagesz=2MB hugepagesz=2MB hugepages=512"）

```

grub2-mkconfig -o /boot/grub2/grub.cfg
systemctl reboot

```

## 加载内核模块

```
modprobe uio  
insmod kmod/igb_uio.ko  
../tools/dpdk-devbind.py --bind=igb_uio <Device BDF>
```

## REHL 7.2

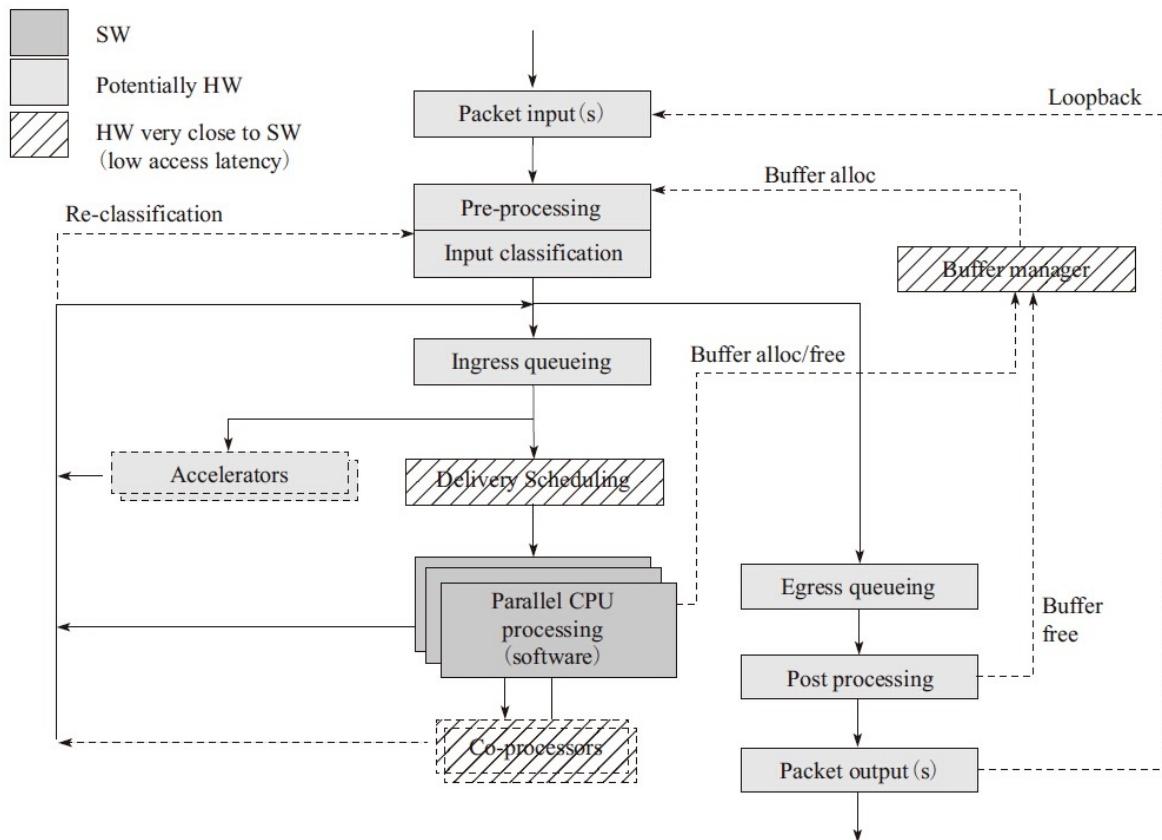
```
subscription-manager repos --enable rhel-7-server-extras-rpms  
yum install dpdk dpdk-tools
```

## Ubuntu 15+

```
apt-get install dpdk
```

# DPDK报文转发

基本的网络包处理主要包含：

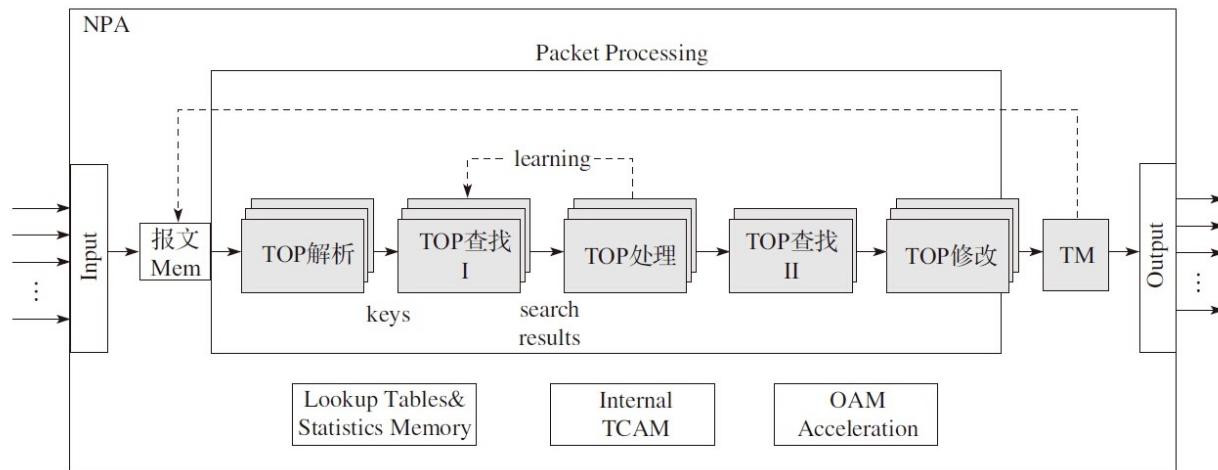


- **Packet input**：报文输入。
- **Pre-processing**：对报文进行比较粗粒度的处理。
- **Input classification**：对报文进行较细粒度的分流。
- **Ingress queueing**：提供基于描述符的队列FIFO。
- **Delivery/Scheduling**：根据队列优先级和CPU状态进行调度。
- **Accelerator**：提供加解密和压缩/解压缩等硬件功能。
- **Egress queueing**：在出口上根据QOS等级进行调度。
- **Post processing**：后期报文处理释放缓存。
- **Packet output**：从硬件上发送出去。

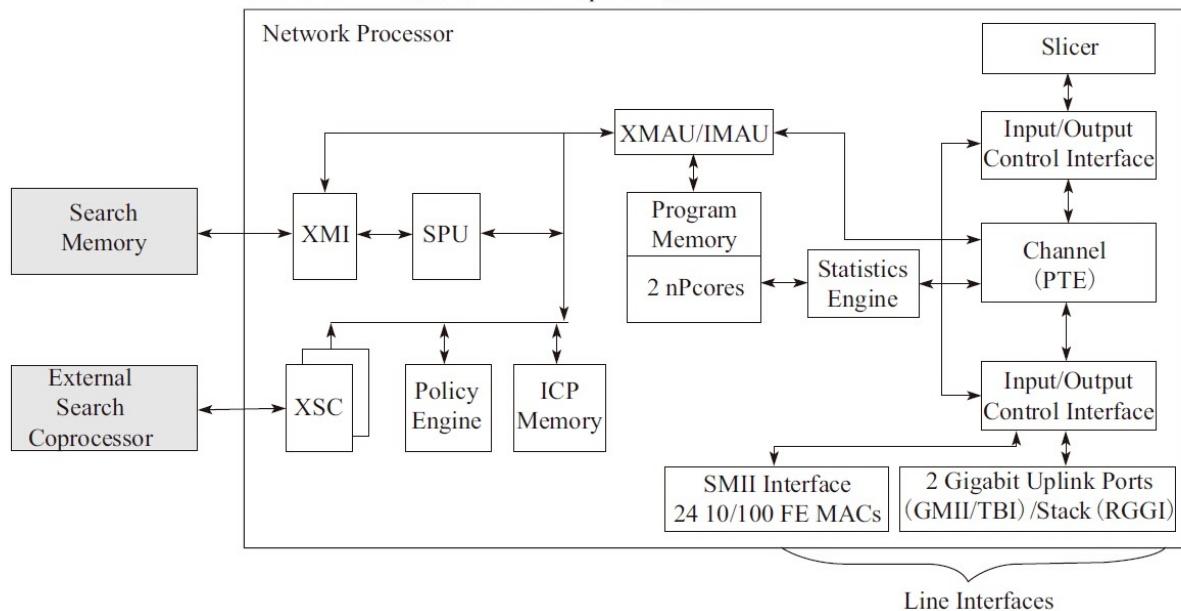
## 专用网络处理器转发模型

传统的Network Processor（专用网络处理器）转发的模型可以分为run to completion（运行至终结，简称RTC）模型和pipeline（流水线）模型。

Ezchip-传统的pipeline 模型

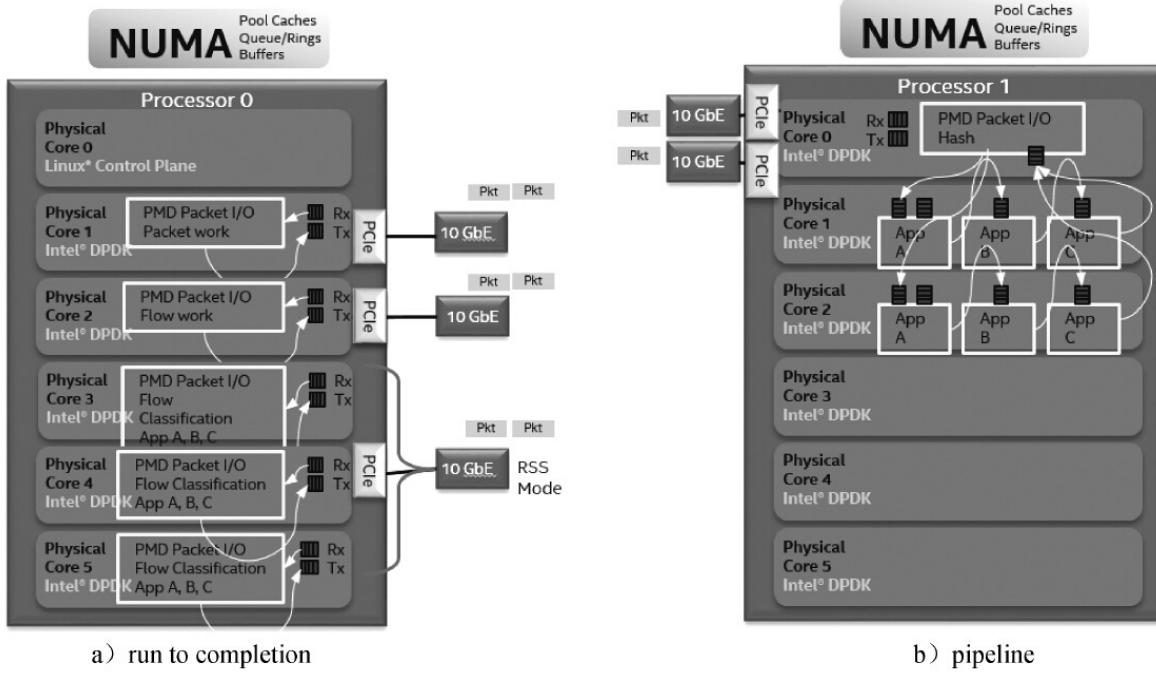


AMCC 345x—传统的 run to completion 模型



## DPDK 转发模型

从a的run to completion的模型中，我们可以清楚地看出，每个IA的物理核都负责处理整个报文的生命周期从RX到TX，这点非常类似前面所提到的AMCC的nP核的作用。在图b的pipeline模型中可以看出，报文的处理被划分成不同的逻辑功能单元A、B、C，一个报文需分别经历A、B、C三个阶段，这三个阶段的功能单元可以不止一个并且可以分布在不同的物理核上，不同的功能单元可以分布在相同的核上（也可以分布在不同的核上），从这一点可以看出，其对于模块的分类和调用比EZchip的硬件方案更加灵活。



优缺点	DPDK pipeline 模型	DPDK RTC 模型
开发	利用脚本方式可以快速配置需要的网络单元，迅速构建网络产品所需的功能	调用 DPDK 的 API 接口实现网络功能，需要二次开发较慢
性能	多核之间会有缓存一致性的问题	会有处理流程上的依赖问题
扩展性	不好	较好

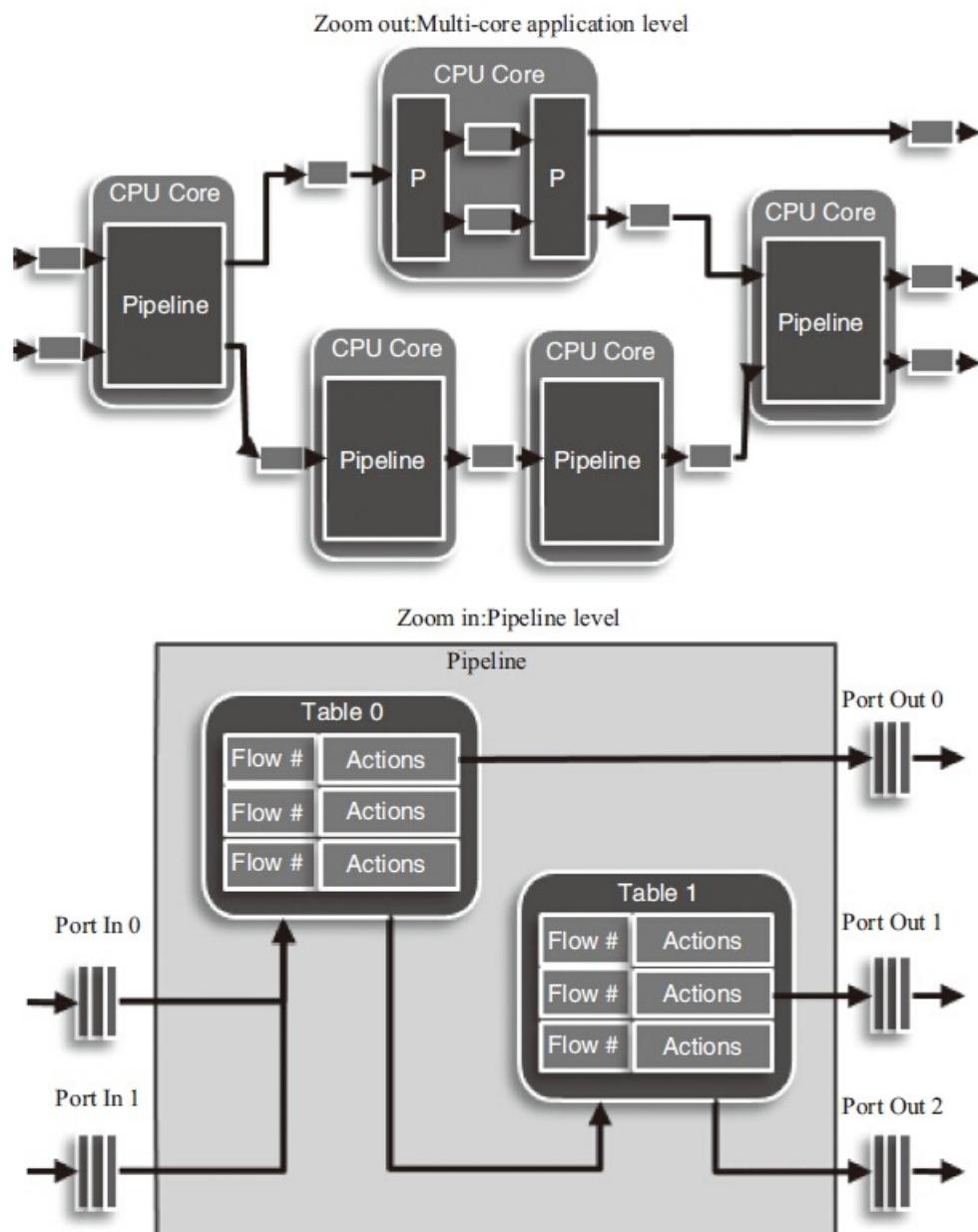
### DPDK run to completion模型

在DPDK的轮询模式中主要通过一些DPDK中eal中的参数-c、-l、-l core s来设置哪些核可以被DPDK使用，最后再把处理对应收发队列的线程绑定到对应的核上。每个报文的整个生命周期都只可能在其中一个线程中出现。和普通网络处理器的run to completion的模式相比，基于IA平台的通用CPU也有不少的计算资源，比如一个socket上面可以有独立运行的16运算单元（核），每个核上面可以有两个逻辑运算单元（thread）共享物理的运算单元。而多个socket可以通过QPI总线连接在一起，这样使得每一个运算单元都可以独立地处理一个报文并且通用处理器上的编程更加简单高效，在快速开发网络功能的同时，利用硬件AES-NI、SHA-NI等特殊指令可以加速网络相关加解密和认证功能。运行到终结功能虽然有许多优势，但是针对单个报文的处理始终集中在一个逻辑单元上，无法利用其他运算单元，并且逻辑的耦合性太强，而流水线模型正好解决了以上的问题。

### DPDK pipeline模型

pipeline的主要思想就是不同的工作交给不同的模块，而每一个模块都是一个处理引擎，每个处理引擎都只单独处理特定的事务，每个处理引擎都有输入和输出，通过这些输入和输出将不同的处理引擎连接起来，完成复杂的网络功能，DPDK pipeline的多处理引擎实例和每个处

理引擎中的组成框图可见图5-5中两个实例的图片：zoom out（多核应用框架）和zoom in（单个流水线模块）。



Zoom out的实例中包含了五个DPDK pipeline处理模块，每个pipeline作为一个特定功能的包处理模块。一个报文从进入到发送，会有两个不同的路径，上面的路径有三个模块（解析、分类、发送），下面的路径有四个模块（解析、查表、修改、发送）。Zoom in的图示中代表在查表的pipeline中有两张查找表，报文根据不同的条件可以通过一级表或两级表的查询从不同的端口发送出去。

DPDK的pipeline是由三大部分组成的：

Pipeline 要素	选 项
逻辑端口 (port)	硬件队列、软件队列、IP Fragmentation、IP Reassembly、发包器、内核网络接口、Source/Sink
查找表 (Table)	Exact Match、哈希、Access Control List (ACL)、Longest Prefix Match (LPM)、数组、Pattern Matching
处理逻辑 (action)	缺省处理逻辑：发送到端口，发送到查找表，丢弃 报文修改逻辑：压入队列，弹出队列，修改报文头 报文流：限速、统计、按照 APP ID 分类 报文加速：加解密、压缩 负载均衡

现在DPDK支持的pipeline有以下几种：

- Packet I/O
- Flow classification
- Firewall
- Routing
- Metering

## 转发算法

除了良好的转发框架之外，转发中很重要的一部分内容就是对于报文字段的匹配和识别，在DPDK中主要用到了精确匹配（Exact Match）算法和最长前缀匹配（Longest Prefix Matching，LPM）算法来进行报文的匹配从而获得相应的信息。

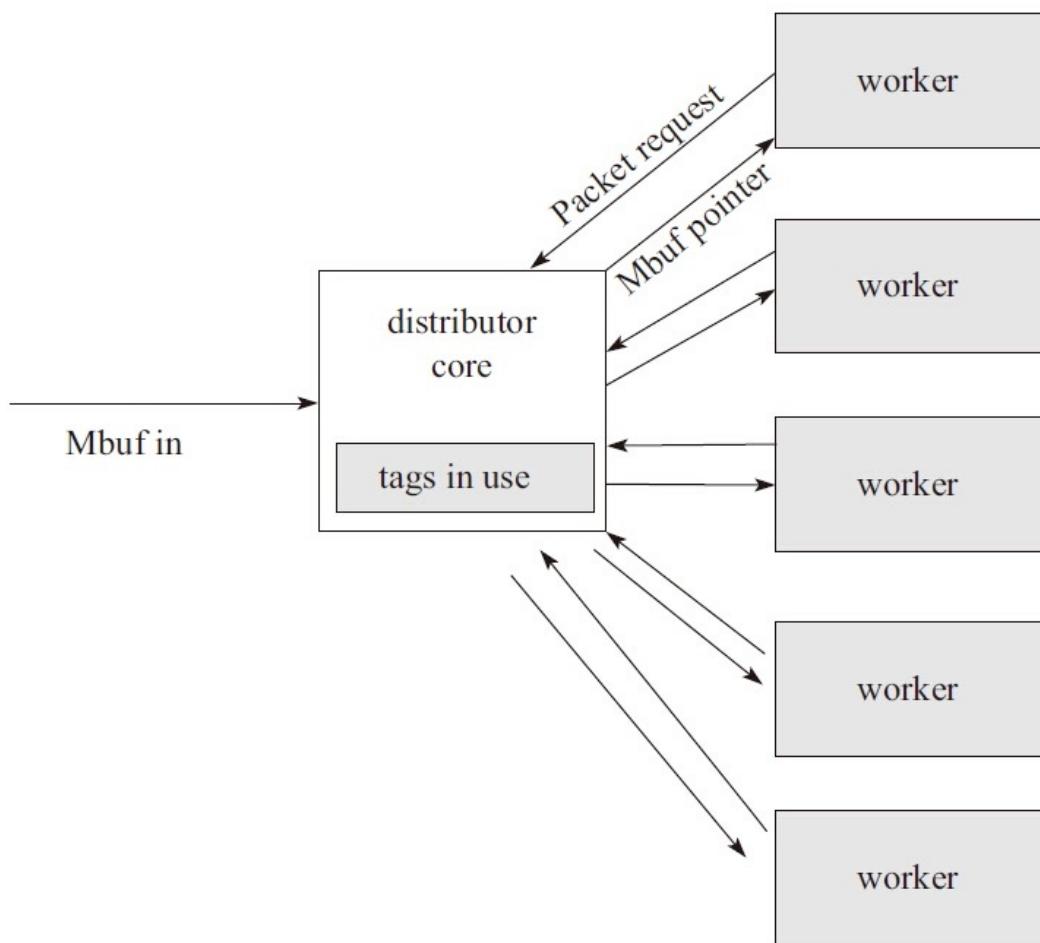
精确匹配主要需要解决两个问题：进行数据的签名（哈希），解决哈希的冲突问题，DPDK中主要支持CRC32和J hash。

最长前缀匹配（Longest Prefix Matching，LPM）算法是指在IP协议中被路由器用于在路由表中进行选择的一个算法。当前DPDK使用的LPM算法就利用内存的消耗来换取LPM查找的性能提升。当查找表条目的前缀长度小于24位时，只需要一次访存就能找到下一条，根据概率统计，这是占较大概率的，当前缀大于24位时，则需要两次访存，但是这种情况是小概率事件。

ACL库利用N元组的匹配规则去进行类型匹配，提供以下基本操作：

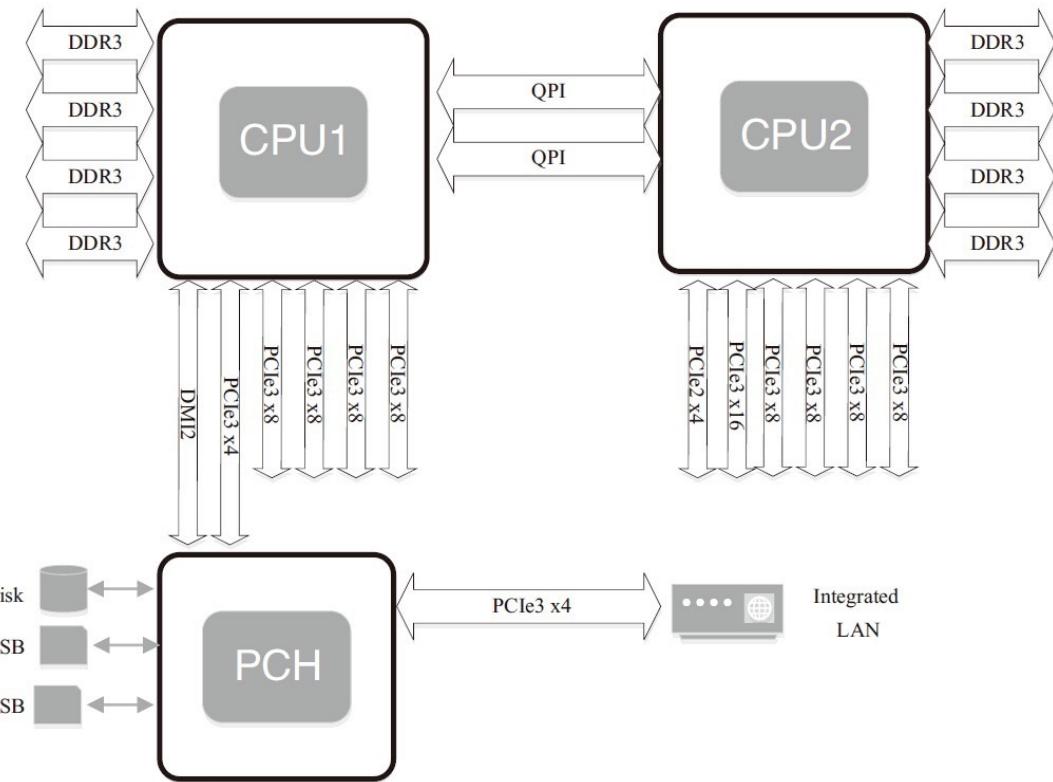
ACL API	说 明
rte_acl_create	创建 AC 的上下文
rte_acl_add_rules	增加规则到 AC 的上下文
rte_acl_build	创建 AC 的运行时结构体
rte_acl_classify	匹配 AC 的规则

Packet distributor（报文分发）是DPDK提供给用户的一个用于包分发的API库，用于进行包分发。主要功能可以用下图进行描述



# NUMA

NUMA来源于AMD Opteron微架构，处理器和本地内存之间有更小的延迟和更大的带宽；每个处理器还可以有自己的总线。处理器访问本地的总线和内存时延迟低，而访问远程资源时则要高。



DPDK充分利用了NUMA的特点

- Per-core memory，每个核都有自己的内存，一方面是本地内存的需要，另一方面也是为了cache一致性
- 用本地处理器和本地内存处理本地设备上产生的数据

```
q = rte_zmalloc_socket("fm10k", sizeof(*q), RTE_CACHE_LINE_SIZE, socket_id)
```

CPU核心的几个概念：

- 处理器核数（cpu cores）：每个物理CPU core的个数
- 逻辑处理器核心数（siblings）：单个物理处理器超线程的个数
- 系统物理处理器封装ID（physical id）：也称为socket插槽，物理机处理器封装个数，物理CPU个数
- 系统逻辑处理器ID（processor）：逻辑CPU数，是物理处理器的超线程技术

**CPU亲和性**

将进程与CPU绑定，提高了Cache命中率，从而减少内存访问损耗。CPU亲和性的主要应用场景为

- 大量计算场景
- 运行时间敏感、决定性的线程，即实时线程

## 相关工具

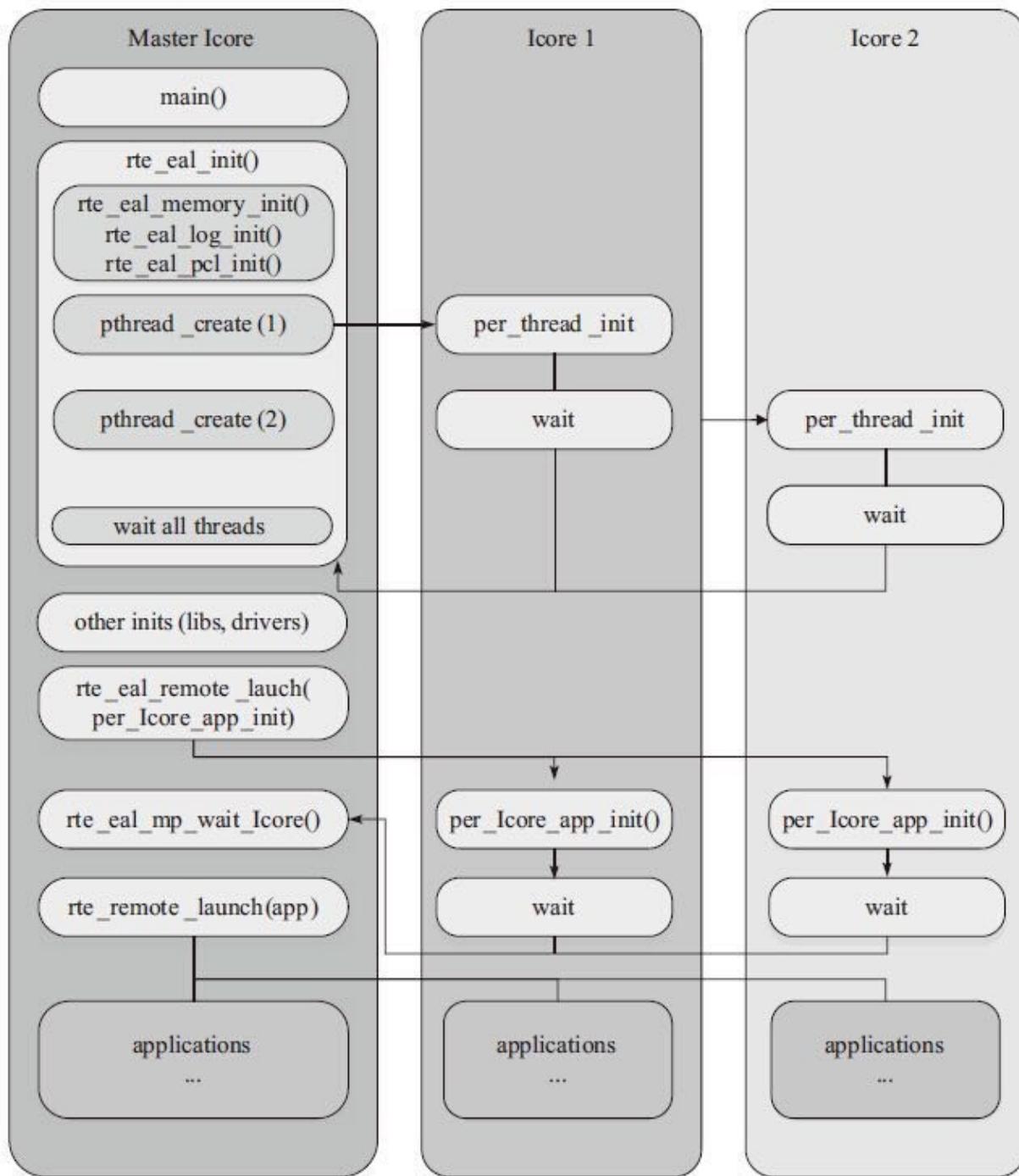
- `sched_set_affinity()`、`sched_get_affinity()` 内核函数
- `taskset`命令
- `isolcpus`内核启动参数：CPU绑定之后依然是有可能发生线程切换，可以借助 `isolcpus=2,3` 将cpu从内核调度系统中剥离。

## DPDK中的CPU亲和性

DPDK中lcore实际上是EAL pthread，每个EAL pthread都有一个Thread Local Storage的 `_lcore_id`，`_lcore_id` 与CPU ID是一致的。注意虽然默认是1:1关系，但可以通过 `--lcores='<lcore_set>@<cpu_set>'` 来指定lcore的CPU亲和性，这样可以不是1:1的，也就是多个lcore还是可以亲和到同一个核，这就需要注意调度的情况（以非抢占式无锁 `rte_ring` 为例）：

- 单生产者、单消费者模式不受影响
- 多生产者、多消费者模式，调度策略为 `SCHED_OTHER` 时，性能会有所影响
- 多生产者、多消费者模式，调度策略为 `SCHED_FIFO/SCHED_RR`，会产生死锁

而在具体实现流程如下所示：



- DPDK通过读取 /sys/devices/system/cpu/cpuX/ 目录的信息获取CPU的分布情况，将第一核设置为MASTER，并通过 eal\_thread\_set\_affinity() 为每个SLAVE绑定CPU
- 不同模块要调用 rte\_eal\_mp\_remote\_launch() 将自己的回调函数注册到DPDK中  
( lcore\_config[].f )
- 每个核最终调用 eal\_thread\_loop()->回调函数 来执行真正的逻辑

### 指令并发

借助SIMD (Single Instruction Multiple Data，单指令多数据) 可以最大化的利用一级缓存访存的带宽，但对频繁的窄位宽数据操作就有比较大的副作用。DPDK中的 rte\_memcpy() 在 Intel处理器上充分利用了SSE/AVX的特点：优先保证Store指令存储的地址对齐，然后在每个

指令周期指令2条Load的特新弥补一部分非对齐Load带来的性能损失。

## 数据同步与互斥

DPDK根据多核处理器的特点，遵循资源局部化的原则，解耦数据的跨核共享，使得性能可以有很好的水平扩展。但当面对实际应用场景，CPU核间的数据通信、数据同步、临界区保护等都是不得不面对的问题

### 原子操作

原子操作（atomic operation）：不可被中断的一个或一系列操作，多个线程执行一个操作时，其中任何一个线程要么完全执行完此操作，要么没有执行此操作的任何步骤，那么这个操作就是原子的。原子操作是其他内核同步方法的基石。CPU提供三种独立的原子锁机制：原子保证操作、加LOCK指令前缀和缓存一致性协议。

原子操作在DPDK代码中的定义都在rte\_atomic.h文件中，主要包含两部分：内存屏蔽和原16、32和64位的原子操作API。

- rte\_mb() : 内存屏障读写API
- rte\_wmb() : 内存屏障写API
- rte\_rmb() : 内存屏障读API
- rte\_atomic64\_add() : 原子操作API

### 读写锁

读写锁实际是一种特殊的自旋锁，它把对共享资源的访问操作划分成读操作和写操作，读操作只对共享资源进行读访问，写操作则需要对共享资源进行写操作。这种锁相对于自旋锁而言，能提高并发性，因为在多处理器系统中，它允许同时有多个读操作来访问共享资源，最大可能的读操作数为实际的逻辑CPU数。

1) 互斥。任意时刻读者和写者不能同时访问共享资源（即获得锁）；任意时刻只能有至多一个写者访问共享资源。2) 读者并发。在满足“互斥”的前提下，多个读者可以同时访问共享资源。3) 无死锁。如果线程A试图获取锁，那么某个线程必将获得锁，这个线程可能是A自己；如果线程A试图但是永远没有获得锁，那么某个或某些线程必定无限次地获得锁。

DPDK读写锁的定义在rte\_rwlock.h文件中，主要用于在查找空闲的memory segment的时候，使用读写锁来保护memseg结构；LPM表创建、查找和释放；Memory ring的创建、查找和释放；ACL表的创建、查找和释放；Memzone的创建、查找和释放等。

### 自旋锁

自旋锁必须基于CPU的数据总线锁定，它通过读取一个内存单元（spinlock\_t）来判断这个自旋锁是否已经被别的CPU锁住。如果不，它写进一个特定值，表示锁定了总线，然后返回。如果是，它会重复以上操作直到成功，或者spin次数超过一个设定值。锁定数据总线的指令只

能保证一个指令操作期间CPU独占数据总线。（自旋锁在锁定的时候，不会睡眠而是会持续地尝试）。其作用是为了解决某项资源的互斥使用。因为自旋锁不会引起调用者睡眠，所以自旋锁的效率远高于互斥锁。虽然自旋锁的效率比互斥锁高，但是它也有些不足之处：

1) 自旋锁一直占用CPU，它在未获得锁的情况下，一直运行——自旋，所以占用着CPU，如果不能在很短的时间内获得锁，这无疑会使CPU效率降低。2) 在用自旋锁时有可能造成死锁，当递归调用时有可能造成死锁，调用有些其他函数（如`copy_to_user()`、`copy_from_user()`、`kmalloc()`等）也可能造成死锁。

DPDK中自旋锁API的定义在`rte_spinlock.h`文件中，其中`rte_spinlock_lock()`，`rte_spinlock_unlock()`被广泛的应用在告警、日志、中断机制、内存共享和link bonding的代码中，用于临界资源的保护。

### 无锁机制

在多核环境下，需要把重要的数据结构从锁的保护下移到无锁环境，以提高软件性能。现在无锁机制变得越来越流行，在特定的场合使用不同的无锁队列，可以节省锁开销，提高程序效率。Linux内核中有无锁队列的实现，可谓简洁而不简单（`kfifo`是一种“First In First Out”数据结构，它采用了前面提到的环形缓冲区来实现，提供一个无边界的字节流服务。采用环形缓冲区的好处是，当一个数据元素被用掉后，其余数据元素不需要移动其存储位置，从而减少拷贝，提高效率。更重要的是，`kfifo`采用了并行无锁技术，`kfifo`实现的单生产/单消费模式的共享队列是不需要加锁同步的）。

无锁队列中单生产者——单消费者模型中不需要加锁，定长的可以通过读指针和写指针进行控制队列操作，变长的通过读指针、写指针、结束指针控制操作。

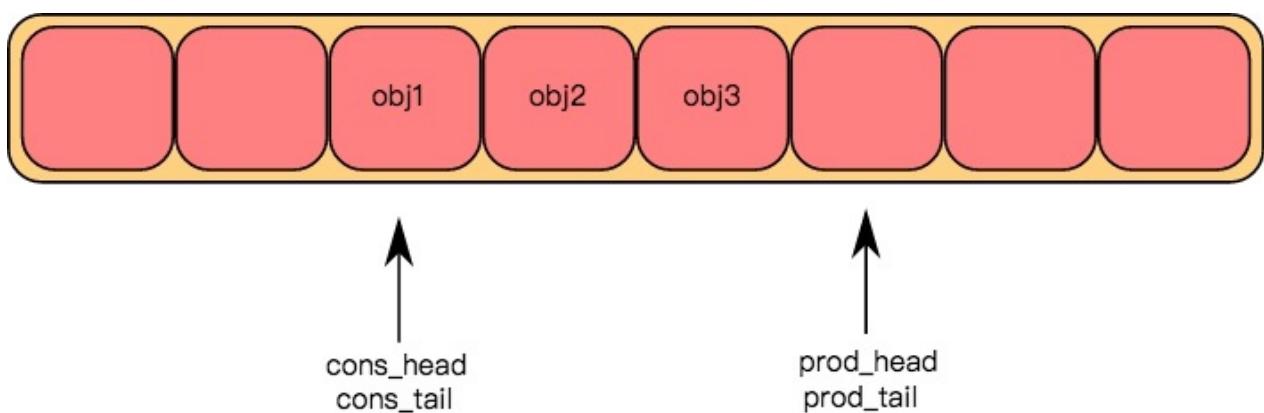
基于无锁环形缓冲的原理，Intel DPDK提供了一套无锁环形缓冲区队列管理代码，支持单生产者产品入列，单消费者产品出列；多名生产者产品入列，多名消费者出列操作。

# DPDK Ring and ivshmem

## DPDK Ring

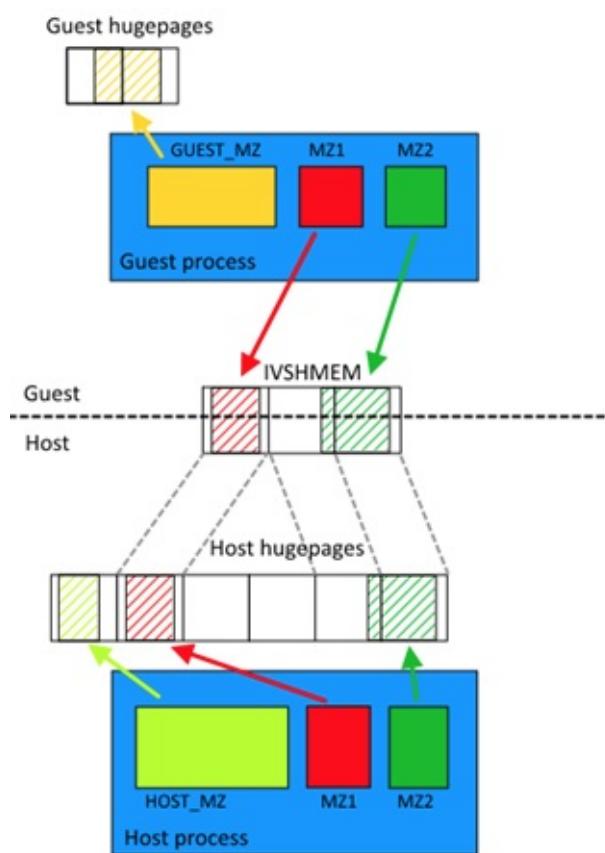
DPDK Ring提供了一个FIFO无锁队列，支持丰富的队列操作，比如

- Multi-consumer or single-consumer dequeue
- Multi-producer or single-producer enqueue
- Bulk dequeue - Dequeues the specified count of objects if successful; otherwise fails
- Bulk enqueue - Enqueues the specified count of objects if successful; otherwise fails
- Burst dequeue - Dequeue the maximum available objects if the specified count cannot be fulfilled
- Burst enqueue - Enqueue the maximum available objects if the specified count cannot be fulfilled

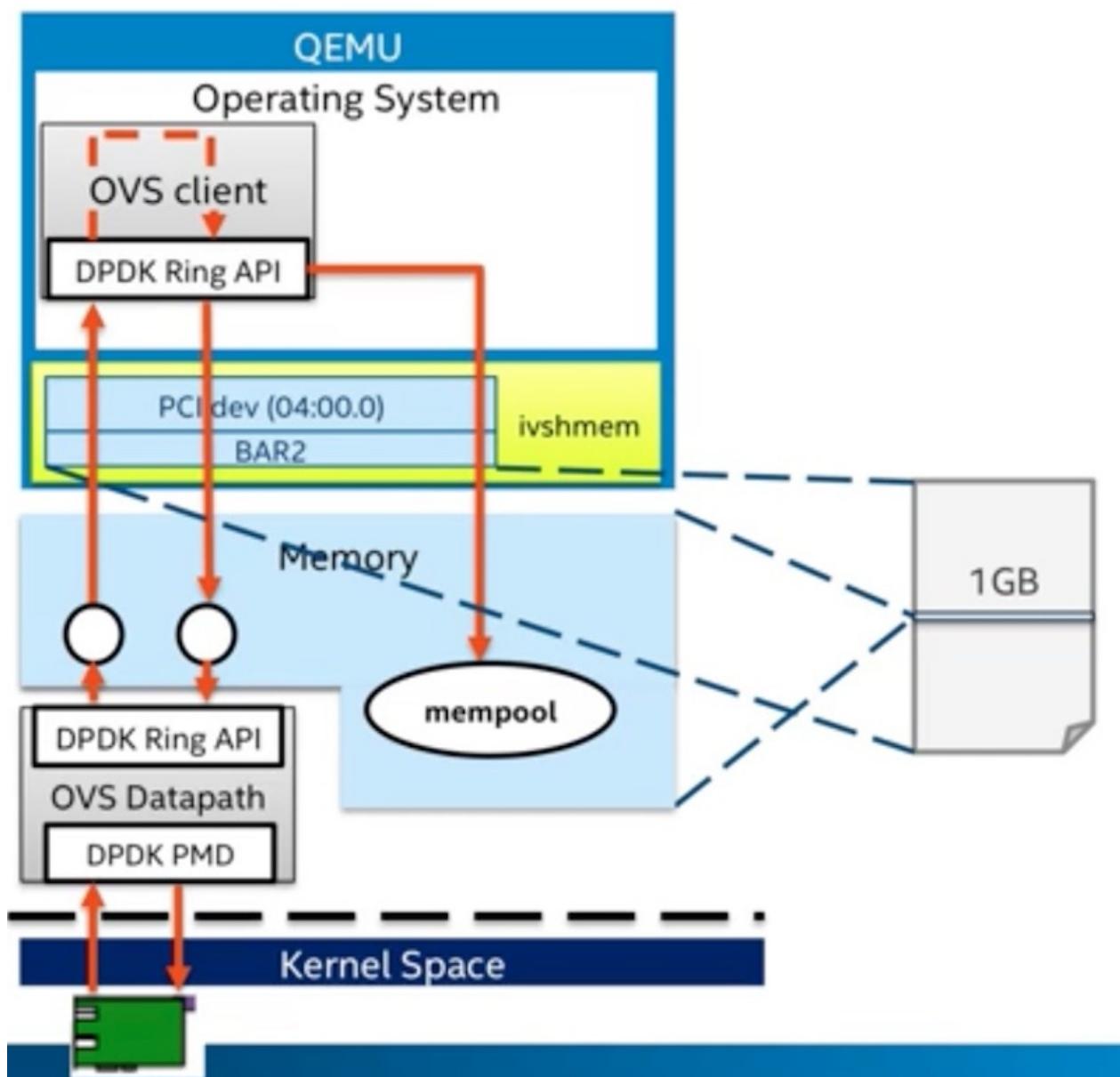


## ivshmem

ivshmem则通过把内存映射成虚拟机PCI设备提供了虚拟机间(host-to-guest or guest-to-guest)共享内存的机制。



DPDK ivshmem:



## ivshmem使用示例

```
# on the host
mount tmpfs /dev/shm -t tmpfs -osize=32m
ivshmem_server -m 64 -p/tmp/nahanni &

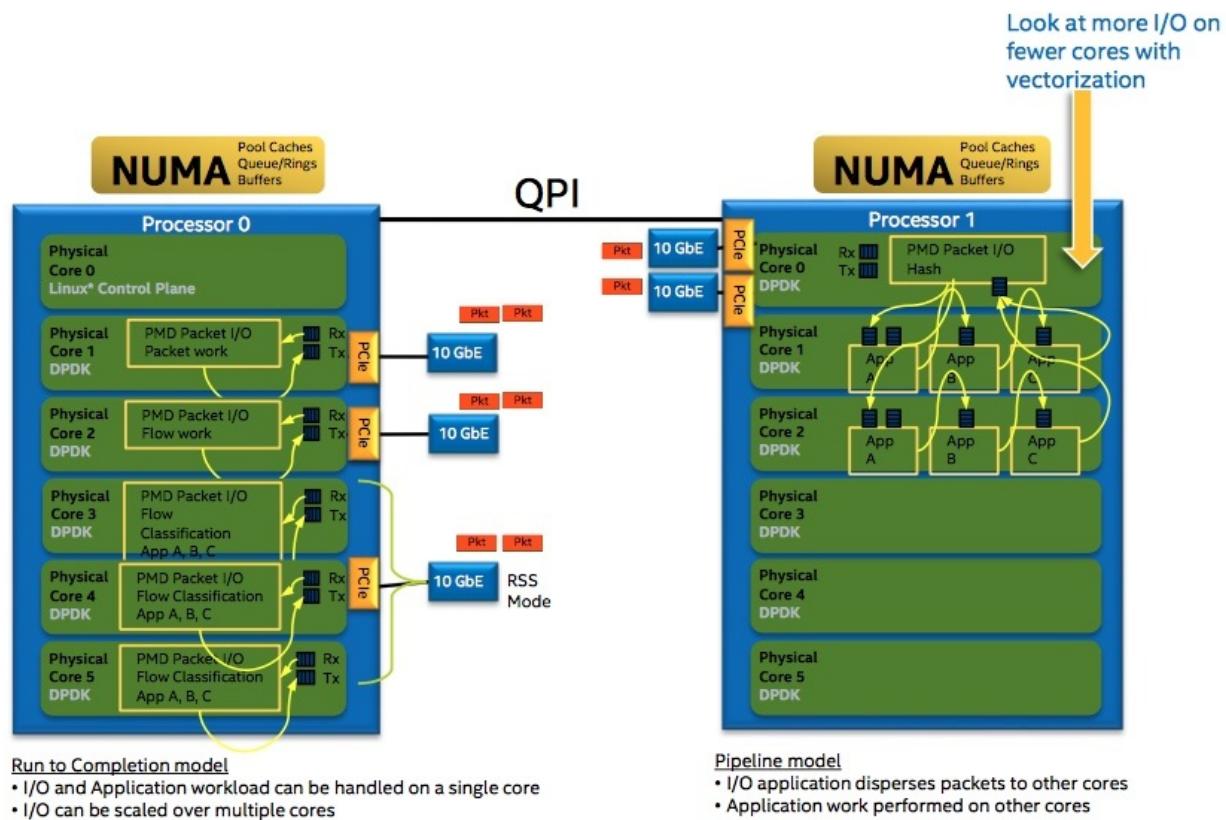
# start VM
qemu-system-x86_64 -hda mg -L pc-bios/ --smp 4 -chardev socket,path=/tmp/nahanni,id=nahanni-device ivshmem,chardev=nahanni,size=32m,msi=off -serial telnet:0.0.0.0:4000,server,nowait,nodelay-enable-kvm&

# inside VM
modprobe kvm_ivshmem
cat/proc/devices | grep kvm_ivshmem
mknod-mode=666 /dev/ivshmem c 245 0
```

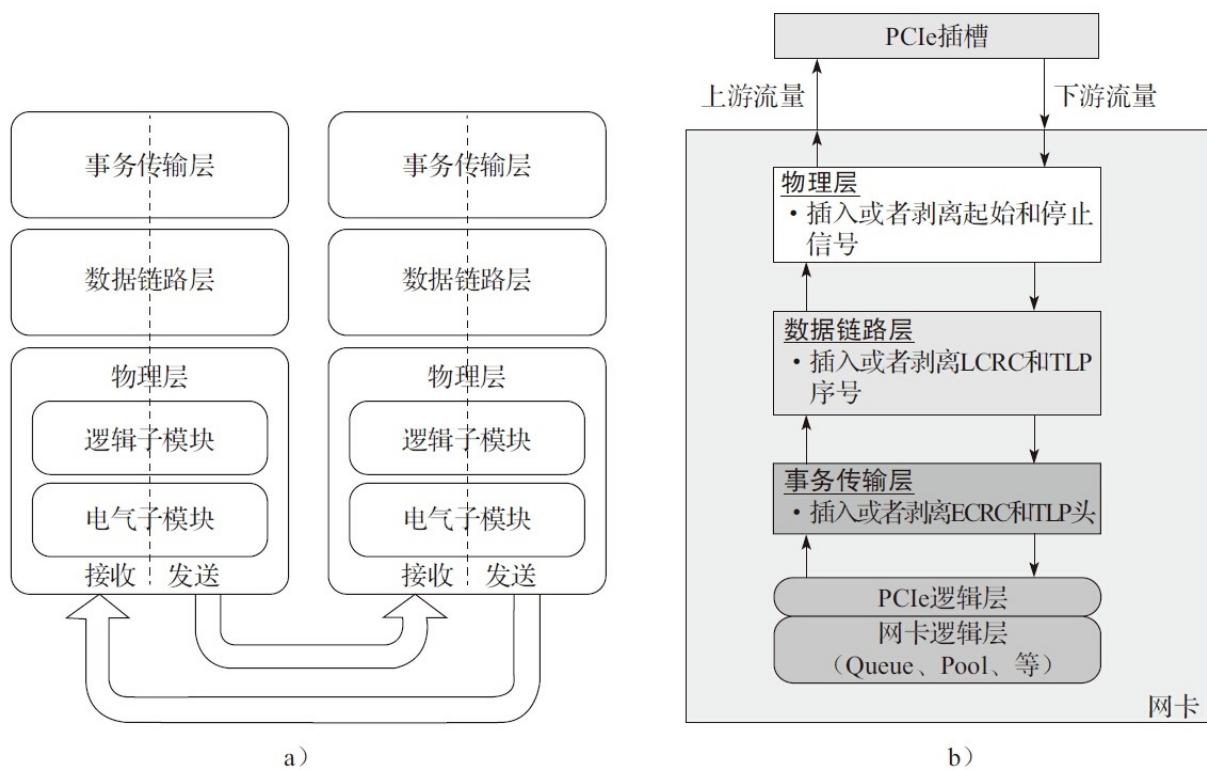
## 参考文档

- DPDK Ring Library
- DPDK IVSHMEM Library

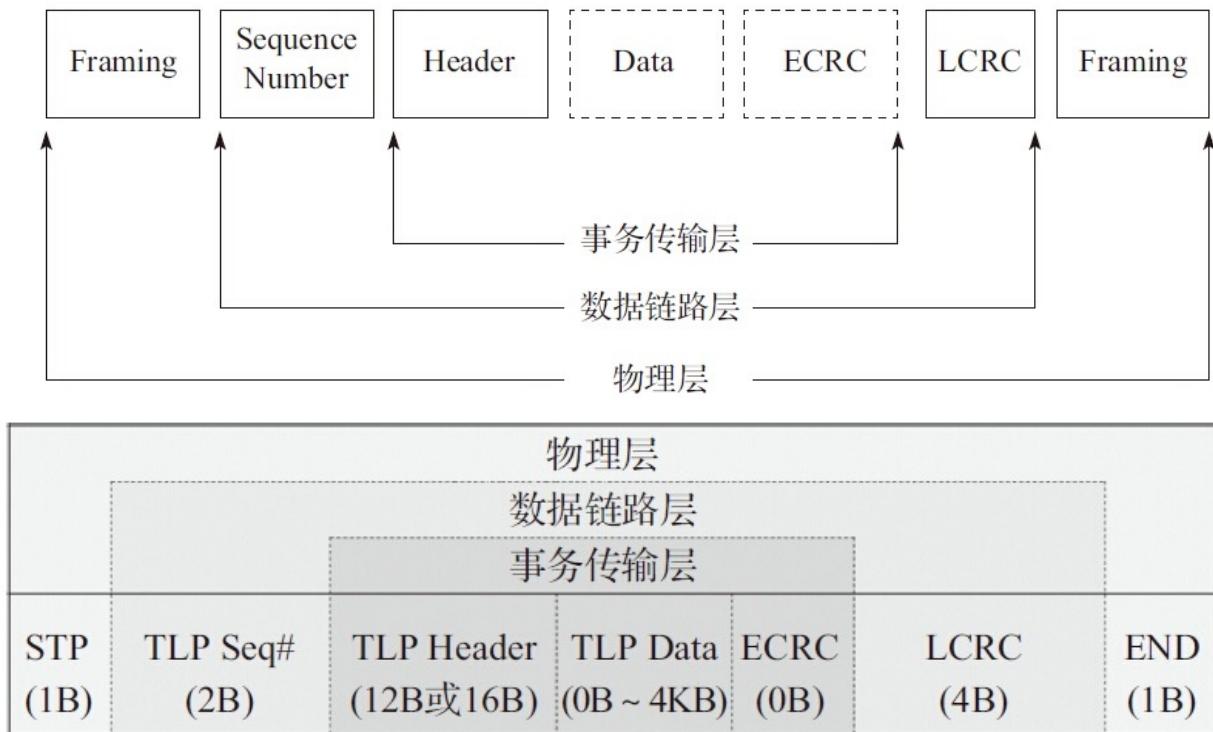
# PCIe



PCI Express (Peripheral Component Interconnect Express) 又称PCIe，它是一种高速串行通信互联标准。PCIe规范遵循开放系统互联参考模型（OSI），自上而下分为事务传输层、数据链路层、物理层。对于特定的网卡，PCIe一般作为处理器外部接口。一般网卡采用DMA控制器通过PCIe Bus访问内存，除了对以太网数据内容的读写外，还有DMA描述符操作相关的读写，这些操作也由MRd/MWr来完成。



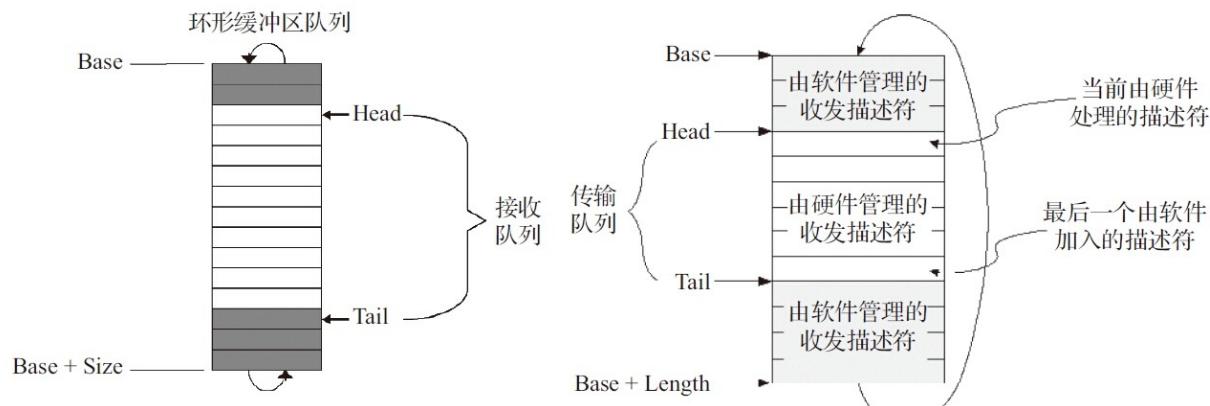
PCIe包格式示例，对于一个完整的TLP包来说，除去有效载荷，额外还有24B的开销（TLP头部以16B计算）。



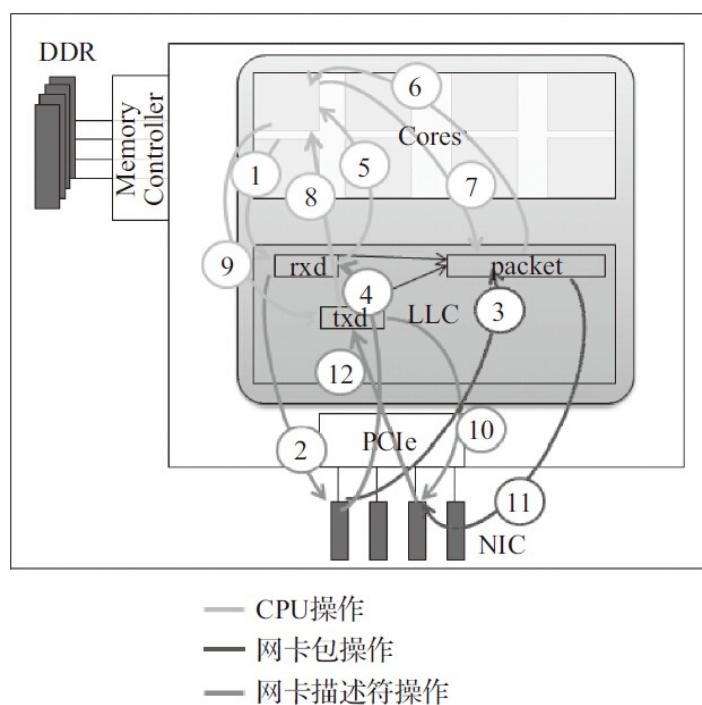
要查看特定PCIe设备的链路能力和当前速率，可以用Linux工具lspci读取PCIe的配置寄存器。

## 网卡DMA描述符环形队列

DMA (Direct Memory Access, 直接存储器访问) 是一种高速的数据传输方式，允许在外部设备和存储器之间直接读写数据。数据既不通过CPU，也不需要CPU干预。整个数据传输操作在DMA控制器的控制下进行。网卡DMA控制器通过环形队列与CPU交互。环形队列的内容部分位于主存中，控制部分通过访问外设寄存器的方式完成。



## 转发操作



1. CPU填充缓冲地址到接收侧描述符
2. 网卡读取接收侧描述符获取缓冲区地址
3. 网卡将包的内容写到缓冲区地址处
4. 网卡回写接收侧描述符更新状态 (确认包内容已写完)
5. CPU读取接收侧描述符以确定包接收完毕
6. CPU读取包内容做转发判断
7. CPU填充更改包内容，做发送准备
8. CPU读发送侧描述符，检查是否有发送完成标志
9. CPU将准备发送的缓冲区地址填充到发送侧描述符
10. 网卡读取发送侧描述符中地址
11. 网卡根据描述符中地址，读取缓冲区中数据内容
12. 网卡写发送侧描述符，更新发送已完成标记

## 优化的考虑

(1) 减少MMIO访问的频度。接收包时，尾寄存器 (tail register) 的更新发生在新缓冲区分配以及描述符重填之后。只要将每包分配并重填描述符的行为修改为滞后的批量分配并重填描述符，接收侧的尾寄存器更新次数将大大减少。DPDK是在判断空置率小于一定值后才触发重填来完成这个操作的。发送包时，就不能采用类似的方法。因为只有及时地更新尾寄存器，才会通知网卡进行发包。但仍可以采用批量发包接口的方式，填充一批等待发送的描述符后，统一更新尾寄存器。 (2) 提高PCIe传输的效率。如果能把4个操作合并成整Cache

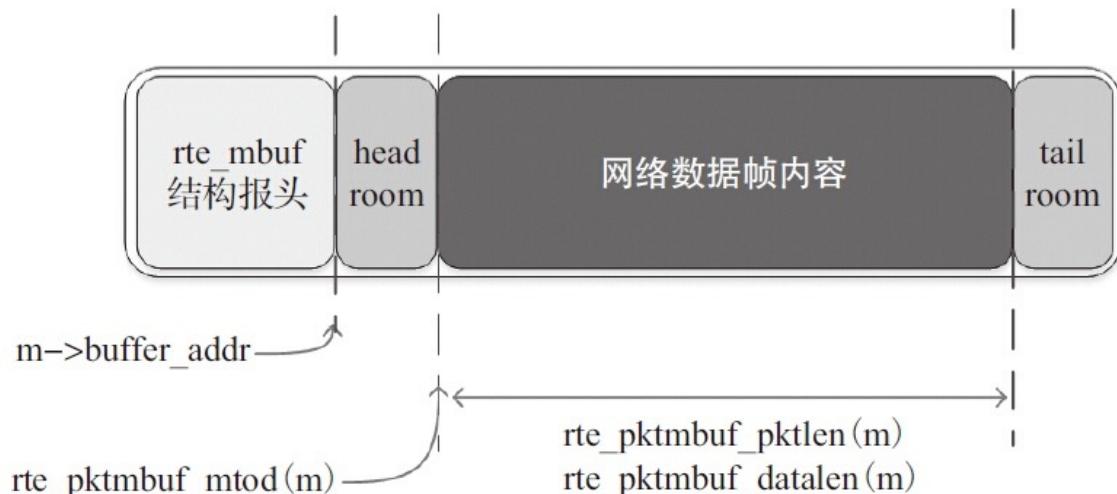
Line大小来作为PCIe的事务请求（PCIe净荷为64Byte），带宽利用率就能得到提升。（3）尽量避免Cache Line的部分写。Cache Line的部分写会引发内存访问read-modify-write的合并操作，增加额外的读操作，也会降低整体性能。所以，DPDK在Mempool中分配buffer的时候，会要求对齐到Cache Line大小。

每转发一个64字节的包的平均转发开销接近于168字节（96+24+8+8+32）。如果计算包转发率，就会得出64B报文的最大转发速率为 $4000\text{MB}/\text{s} / 168\text{B} = 23.8\text{Mp/s}$ 。

## Mbuf

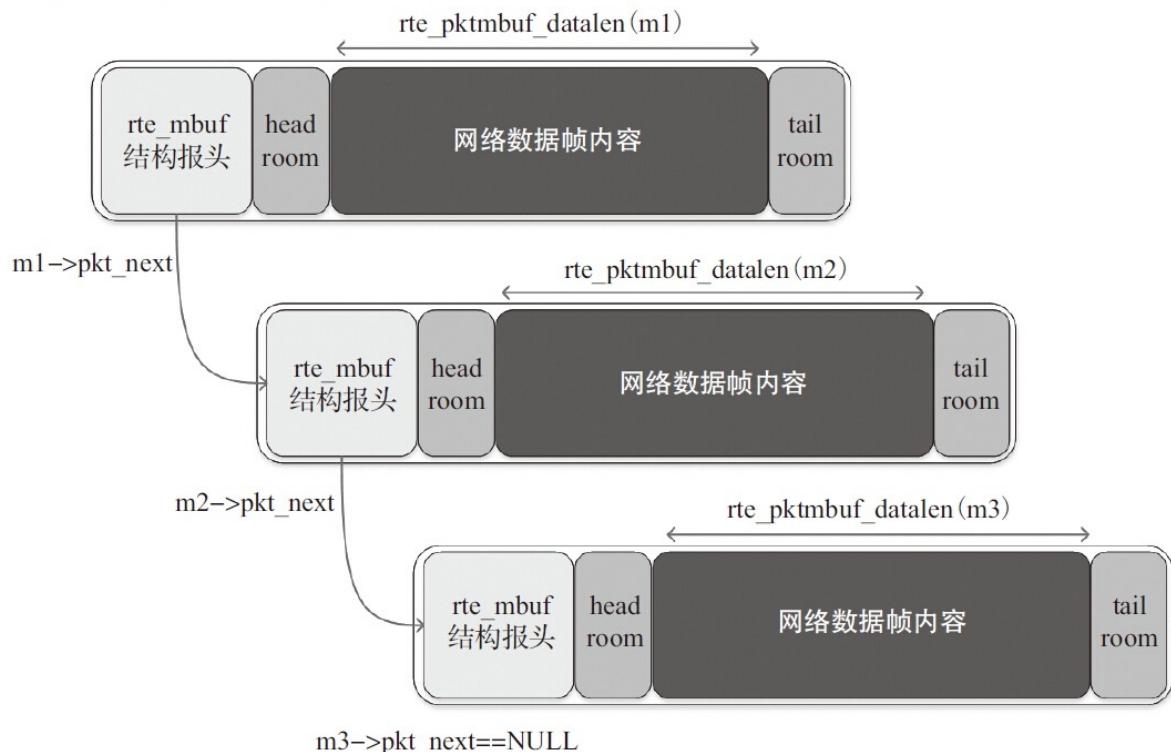
为了高效访问数据，DPDK将内存封装在Mbuf（`struct rte_mbuf`）结构体内。Mbuf主要用来封装网络帧缓存，也可用来封装通用控制信息缓存（缓存类型需使用`CTRL_MBUF_FLAG`来指定）。网络帧元数据的一部分内容由DPDK的网卡驱动写入。这些内容包括VLAN标签、RSS哈希值、网络帧入口端口号以及巨型帧所占的Mbuf个数等。对于巨型帧，网络帧元数据仅出现在第一个帧的Mbuf结构中，其他的帧该信息为空。

单帧结构



### 巨型帧结构

`rte_pktmbuf_pktnlen(m) = rte_pktmbuf_datalen(m1) + rte_pktmbuf_datalen(m2) + rte_pktmbuf_datalen(m3)`

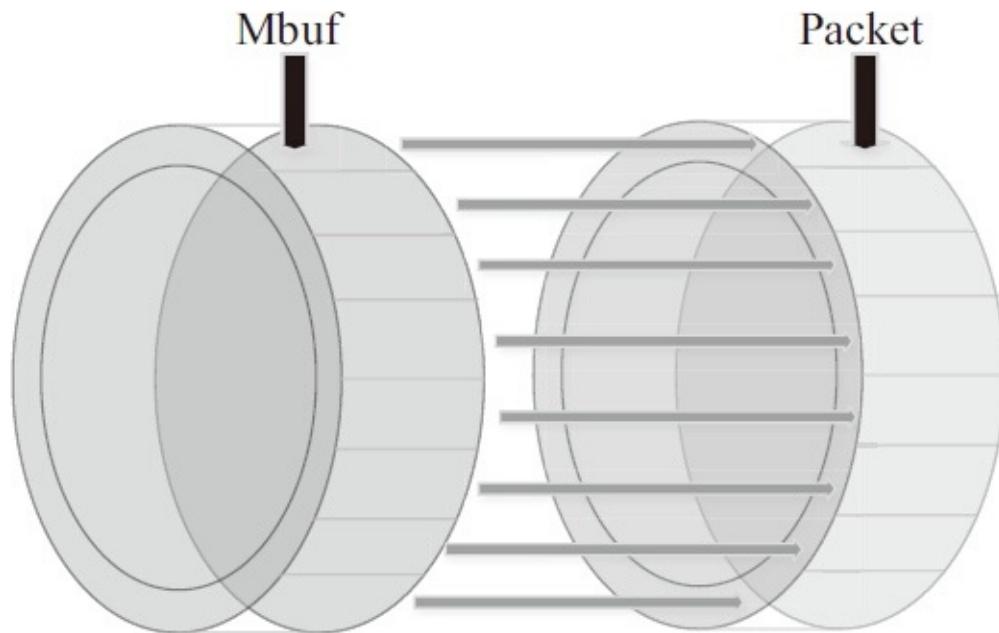


## Mempool

在DPDK中，数据包的内存操作对象被抽象化为Mbuf结构，而有限的rte\_mbuf结构对象则存储在内存池中。内存池使用环形缓存区来保存空闲对象。

当一个网络帧被网卡接收时，DPDK的网卡驱动将其存储在一个高效的环形缓存区中，同时在Mbuf的环形缓存区中创建一个Mbuf对象。当然，两个行为都不涉及向系统申请内存，这些内存已经在内存池被创建时就申请好了。Mbuf对象被创建好后，网卡驱动根据分析出的帧信息将其初始化，并将其和实际帧对象逻辑相连。对网络帧的分析处理都集中于Mbuf，仅在必要的时候访问实际网络帧。这就是内存池的双环形缓存区结构。为增加对Mbuf的访问效率，内

存池还拥有内存通道/Rank对齐辅助方法。内存池还允许用户设置核心缓存区大小来调节环形内存块读写的频率。



实践证明，在内存对象之间补零，以确保每个对象和内存的一个通道和Rank起始处对齐，能大幅减少未命中的发生概率且增加存取效率。在L3转发和流分类应用中尤为如此。内存池以更大内存占有量的代价来支持此项技术。在创建一个内存池时，用户可选择是否启用该技术。

多核CPU访问同一个内存池或者同一个环形缓存区时，因为每次读写时都要进行Compare-and-Set操作来保证期间数据未被其他核心修改，所以存取效率较低。DPDK的解决方法是使用单核本地缓存一部分数据，实时对环形缓存区进行块读写操作，以减少访问环形缓存区的次数。单核CPU对自己缓存的操作无须中断，访问效率因而得到提高。当然，这个方法也并非全是好处：该方法要求每个核CPU都有自己私用的缓存（大小可由用户定义，也可为0，或禁用该方法），而这些缓存在绝大部分时间都没有能得到百分之百运用，因此一部分内存空间将被浪费。

# 网卡性能优化

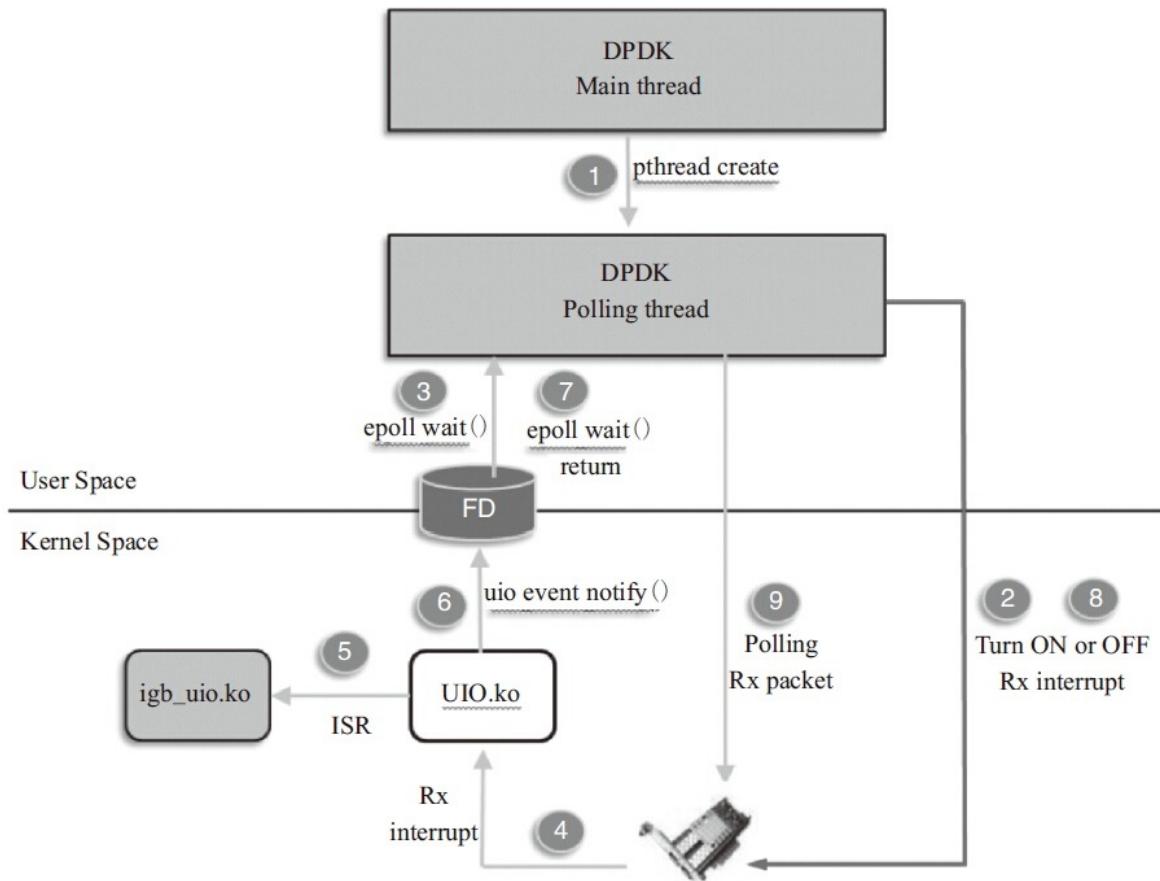
运行在操作系统内核态的网卡驱动程序基本都是基于异步中断处理模式，而DPDK采用了轮询或者轮询混杂中断的模式来进行收包和发包。DPDK起初的纯轮询模式是指收发包完全不使用任何中断，集中所有运算资源用于报文处理。但这不是意味着DPDK不可以支持任何中断。根据应用场景需要，中断可以被支持，最典型的就是链路层状态发生变化的中断触发与处理。

任何包进入到网卡，网卡硬件会进行必要的检查、计算、解析和过滤等，最终包会进入物理端口的某一个队列。物理端口上的每一个收包队列，都会有一个对应的由收包描述符组成的软件队列来进行硬件和软件的交互，以达到收包的目的。DPDK的轮询驱动程序负责初始化好每一个收包描述符，其中就包含把包缓冲内存块的物理地址填充到收包描述符对应的位置，以及把对应的收包成功标志复位。然后驱动程序修改相应的队列管理寄存器来通知网卡硬件队列里面的哪些位置的描述符是可以有硬件把收到的包填充进来的。网卡硬件会把收到的包一一填充到对应的收包描述符表示的缓冲内存块里面，同时把必要的信息填充到收包描述符里面，其中最重要的就是标记好收包成功标志。当一个收包描述符所代表的缓冲内存块大小不够存放一个完整的包时，这时候就可能需要两个甚至多个收包描述符来处理一个包。每一个收包队列，DPDK都会有一个对应的软件线程负责轮询里面的收包描述符的收包成功的标志。一旦发现某一个收包描述符的收包成功标志被硬件置位了，就意味着有一个包已经进入到网卡，并且网卡已经存储到描述符对应的缓冲内存块里面，这时候驱动程序会解析相应的收包描述符，提取各种有用的信息，然后填充对应的缓冲内存块头部。然后把收包缓冲内存块存放到收包函数提供的数组里面，同时分配好一个新的缓冲内存块给这个描述符，以便下一次收包。

每一个发包队列，DPDK都会有一个对应的软件线程负责设置需要发送出去的包，DPDK的驱动程序负责提取发包缓冲内存块的有效信息，例如包长、地址、校验和信息、VLAN配置信息等。DPDK的轮询驱动程序根据内存缓存块中的包的内容来负责初始化好每一个发包描述符，驱动程序会把每个包翻译成为一个或者多个发包描述符里能够理解的内容，然后写入发包描述符。发包的轮询就是轮询发包结束的硬件标志位。DPDK驱动程序根据需要发送的包的信息和内容，设置好相应的发包描述符，包含设置对应的RS标志，然后会在发包线程里不断查询发包是否结束。当驱动程序发现写回标志，意味着包已经发送完成，就释放对应的发包描述符和对应的内存缓冲块，这时候就全部完成了包的发送过程。

由于实际网络应用中可能存在的潮汐效应，在某些时间段网络数据流量可能很低，甚至完全没有需要处理的包，这样就会出现在高速端口下低负荷运行的场景，而完全轮询的方式会让处理器一直全速运行，明显浪费处理能力和不节能。因此在DPDK R2.1和R2.2陆续添加了收包中断与轮询的混合模式的支持。例子程序l3fwd-power，使用了DPDK支持的中断加轮询的混合模式。应用程序开始就是轮询收包，这时候收包中断是关闭的。但是当连续多次收到的包的个数为零的时候，应用程序定义了一个简单的策略来决定是否以及什么时候让对应的收

包线程进入休眠模式，并且在休眠之前使能收包中断。休眠之后对应的核的运算能力就被释放出来。当后续有任何包收到的时候，会产生一个收包中断，并且最终唤醒对应的应用程序收包线程。线程被唤醒后，就会关闭收包中断，再次轮询收包。



## 性能优化

- **Burst收发包**就是DPDK的优化模式，它把收发包复杂的处理过程进行分解，打散成不同的相对较小的处理阶段，把相邻的数据访问、相似的数据运算集中处理。这样就能尽可能减少对内存或者低一级的处理器缓存的访问次数，用更少的访问次数来完成更多次收发包运算所需要数据的读或者写（参考 `rte_eth_rx_burst()` 和 `rte_eth_tx_burst()`）。
- 利用CPU指令乱序多发的能力，批量处理无数据前后依赖关系的独立事务，可以掩藏指令延迟。对于重复事务执行，通常采用循环逐次操作。对于较复杂事务，编译器很难大量地去乱序不同迭代序列下的指令。为了达到批量处理下乱序时延隐藏的效果，常用的做法是在一个序列中铺开执行多个事务，以一个合理的步进迭代。
- 利用Intel SIMD指令进一步并行化包收发。
- 大页：例如，增加内核启动参数“`default_hugepagesz=1G hugepagesz=1G hugepages=8`”来配置好8个1G的大页。
- DPDK的软件线程一般都需要独占一些处理器的物理核或者逻辑核来完成稳定和高性能的包处理，如果硬件平台的处理器有足够的核，一般都会预留出一些核来给DPDK应用程序使用。例如，增加内核启动参数“`isolcpus=2,3,4,5,6,7,8`”，使处理器上ID为

2, 3, 4, 5, 6, 7, 8的逻辑核不被操作系统调度。

- 修改编译参数来使能“extended tag”：`CONFIG_RTE_PCI_CONFIG=y`，  
`CONFIG_RTE_PCI_EXTENDED_TAG="on"`。

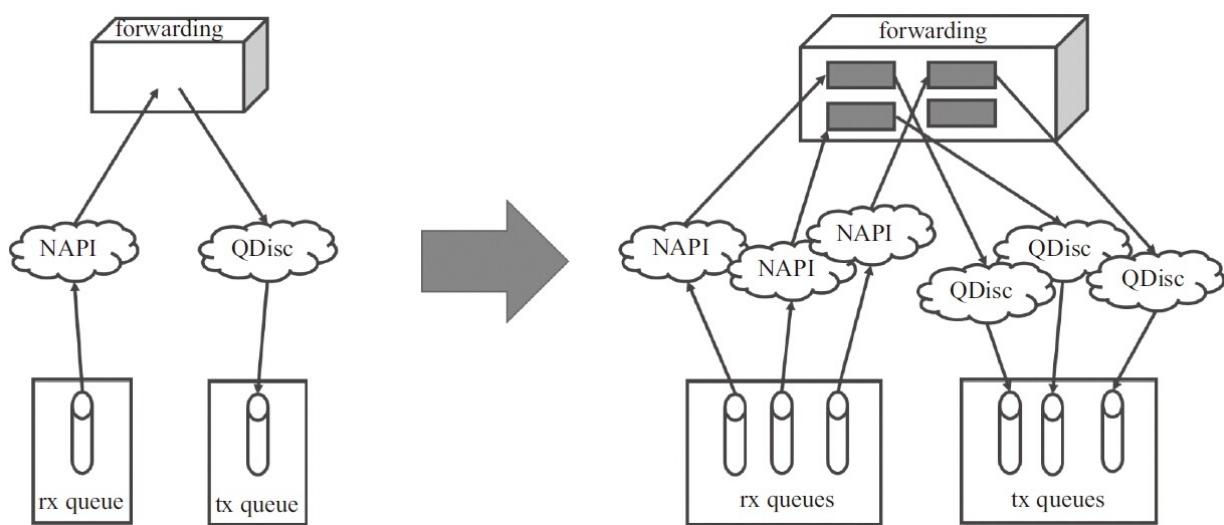
- DPDK参数

- 收包队列长度：DPDK很多示例程序里面默认的收包队列长度是128，这就是表示为每一个收包队列都分配128个收包描述符，这是一个适应大多数场景经验值。但是在某些更高速率的网卡收包的情况下，128就可能不一定够了，或者在某些场景下发现丢包现象比较容易的时候，就需要考虑使用更长的收包队列，例如可以使用512或者1024。
- 发包队列长度：DPDK的示例程序里面默认的发包队列长度使用的是512，这就表示为每一个发包队列都分配512个发包描述符，这是一个适用大部分场合经验值。当处理更高速率的网卡设备时，或者发现有丢包的时候，就应该考虑更长的发包队列，例如1024。
- 收包队列可释放描述符数量阈值（rx\_free\_thresh）：DPDK驱动程序并没有每次收包都更新收包队列尾部索引寄存器，而是在可释放的收包描述符数量达到一个阈值（rx\_free\_thresh）的时候才真正更新收包队列尾部索引寄存器。这个可释放收包描述符数量阈值在驱动程序里面的默认值一般都是32，在示例程序里面，有的会设置成用户可配参数，可能设置成不同的默认值，例如64或者其他。设置合适的可释放描述符数量阈值，可以减少没有必要的过多的收包队列尾部索引寄存器的访问，改善收包的性能。
- 发包队列发送结果报告阈值（tx\_rs\_thresh）：这个阈值的存在允许软件在配置发包描述符的同时设定一个回写标记，只有设置了回写标记的发包描述符硬件才会在发包完成后产生写回的动作，并且这个回写标记是设置在一定间隔（阈值）的发包描述符上。这个机制可以减少不必要的回写的次数，从而能够改善性能。
- 发包描述符释放阈值（tx\_free\_thresh）：在DPDK驱动程序里面，默认值是32，用户可能需要根据实际使用的队列长度来调整。发包描述符释放阈值设置得过大，则可能描述符释放的动作很频繁发生，影响性能；发包描述符释放阈值设置过小，则可能每一次集中释放描述符的时候耗时较多，来不及提供新的可用的发包描述符给发包函数使用，甚至造成丢包。

## 网卡多队列

网卡多队列，顾名思义，也就是传统网卡的DMA队列有多个，网卡有基于多个DMA队列的分配机制。多队列网卡已经是当前高速率网卡的主流。

Linux内核中，RPS（Receive Packet Steering）在接收端提供了这样的机制。RPS主要是把软中断的负载均衡到CPU的各个core上，网卡驱动对每个流生成一个hash标识，这个hash值可以通过四元组（源IP地址SIP，源四层端口SPORT，目的IP地址DIP，目的四层端口DPORT）来计算，然后由中断处理的地方根据这个hash标识分配到相应的core上去，这样就可以比较充分地发挥多核的能力了。



## DPDK多队列支持

DPDK Packet I/O机制具有与生俱来的多队列支持功能，可以根据不同的平台或者需求，选择需要使用的队列数目，并可以很方便地使用队列，指定队列发送或接收报文。由于这样的特性，可以很容易实现CPU核、缓存与网卡队列之间的亲和性，从而达到很好的性能。从DPDK的典型应用l3fwd可以看出，在某个核上运行的程序从指定的队列上接收，往指定的队列上发送，可以达到很高的cache命中率，效率也就会高。

除了方便地做到对指定队列进行收发包操作外，DPDK的队列管理机制还可以避免多核处理器中的多个收发进程采用自旋锁产生的不必要等待。

以run to completion模型为例，可以从核、内存与网卡队列之间的关系来理解DPDK是如何利用网卡多队列技术带来性能的提升。

- 将网卡的某个接收队列分配给某个核，从该队列中收到的所有报文都应当在该指定的核上处理结束。
- 从核对应的本地存储中分配内存池，接收报文和对应的报文描述符都位于该内存池。

- 为每个核分配一个单独的发送队列，发送报文和对应的报文描述符都位于该核和发送队列对应的本地内存池中。

可以看出不同的核，操作的是不同的队列，从而避免了多个线程同时访问一个队列带来的锁的开销。但是，如果逻辑核的数目大于每个接口上所含的发送队列的数目，那么就需要有机制将队列分配给这些核。不论采用何种策略，都需要引入锁来保护这些队列的数据。

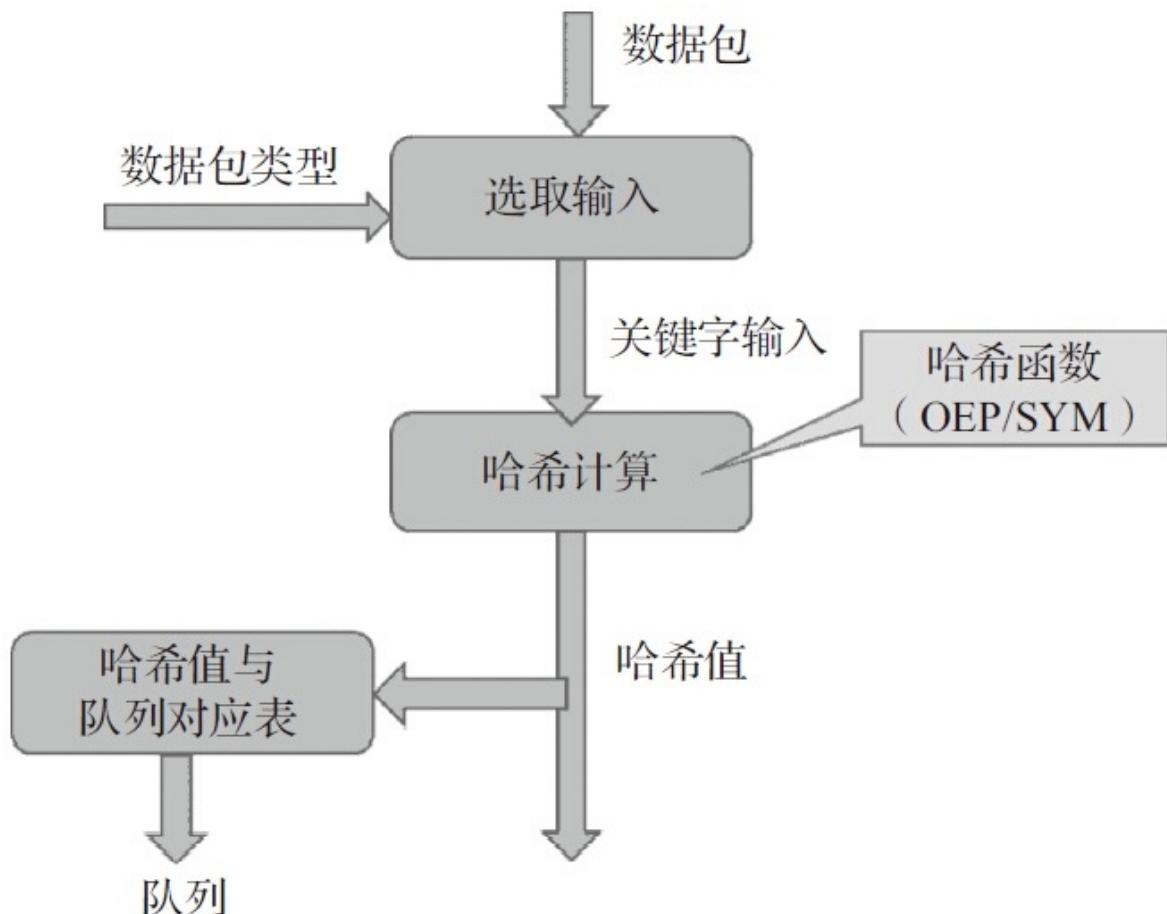
网卡是如何将网络中的报文分发到不同的队列呢？常用的方法有微软提出的RSS与英特尔提出的Flow Director技术，前者是根据哈希值希望均匀地将包分发到多个队列中。后者是基于查找的精确匹配，将包分发到指定的队列中。此外，网卡还可以根据优先级分配队列提供对QoS的支持。

## 流分类

高级的网卡设备（比如Intel XL710）可以分析出包的类型，包的类型会携带在接收描述符中，应用程序可以根据描述符快速地确定包是哪种类型的包。DPDK的Mbuf结构中含有相应的字段来表示网卡分析出的包的类型。

### RSS (Receive-Side Scaling，接收方扩展)

RSS就是根据关键字通过哈希函数计算出哈希值，再由哈希值确定队列。



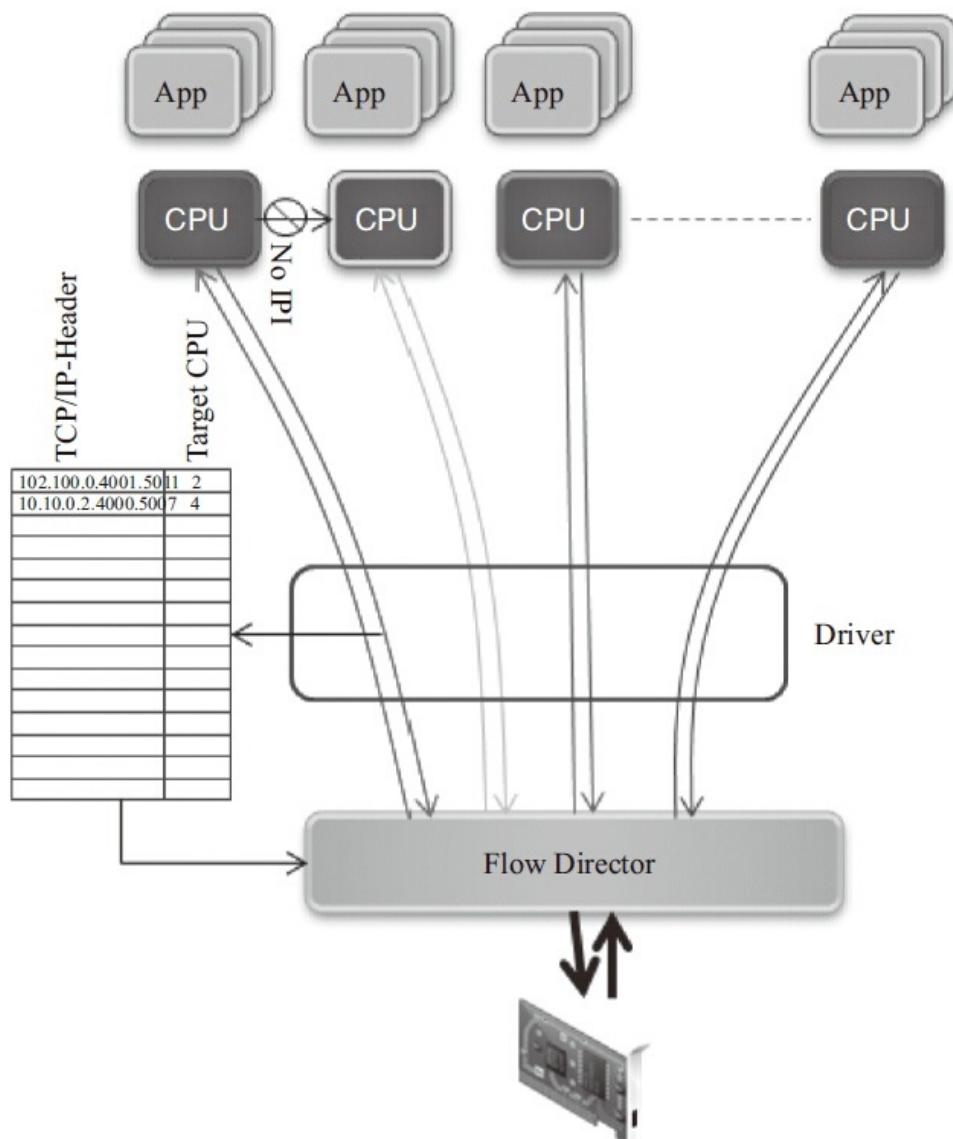
关键字是如何确定的呢？

数据包类型	哈希计算输入
IPV4 UDP	S-IP、D-IP、S-Port、D-Port
IPV4 TCP	S-IP、D-IP、S-Port、D-Port
IPV4 SCTP	S-IP、D-IP、S-Port、D-Port、Verification-Tag
IPV4 OTHER	S-IP、D-IP
IPV6 UDP	S-IP、D-IP、S-Port、D-Port
IPV6 TCP	S-IP、D-IP、S-Port、D-Port
IPV6 SCTP	S-IP、D-IP、S-Port、D-Port、Verification-Tag
IPV6 OTHER	S-IP、D-IP

哈希函数一般选取微软托普利兹算法（Microsoft Toeplitz Based Hash）或者对称哈希。

## Flow Director

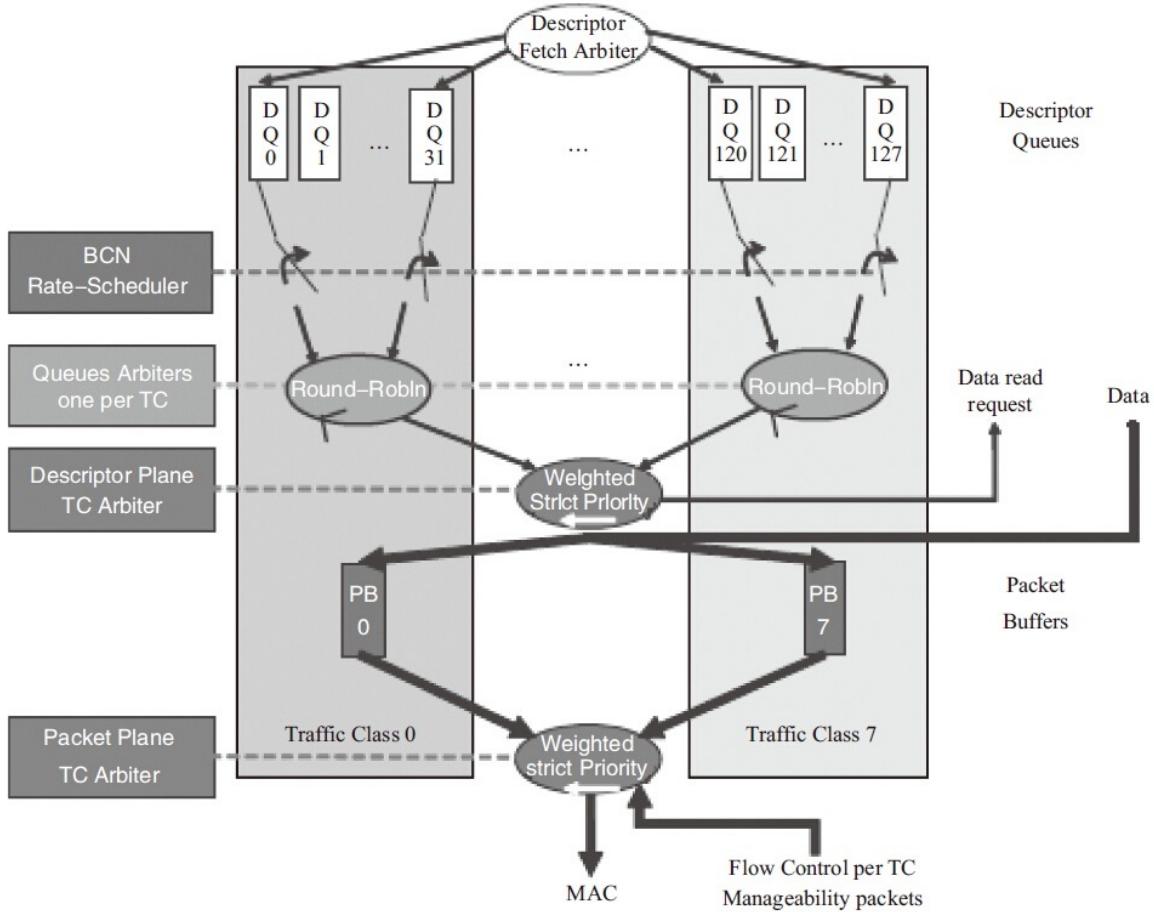
Flow Director技术是Intel公司提出的根据包的字段精确匹配，将其分配到某个特定队列的技术：网卡上存储了一个Flow Director的表，表的大小受硬件资源限制，它记录了需要匹配字段的关键字及匹配后的动作；驱动负责操作这张表，包括初始化、增加表项、删除表项；网卡从线上收到数据包后根据关键字查Flow Director的这张表，匹配后按照表项中的动作处理，可以是分配队列、丢弃等。



相比RSS的负载分担功能，它更加强调特定性。比如，用户可以为某几个特定的TCP对话（S-IP+D-IP+S-Port+D-Port）预留某个队列，那么处理这些TCP对话的应用就可以只关心这个特定的队列，从而省去了CPU过滤数据包的开销，并且可以提高cache的命中率。

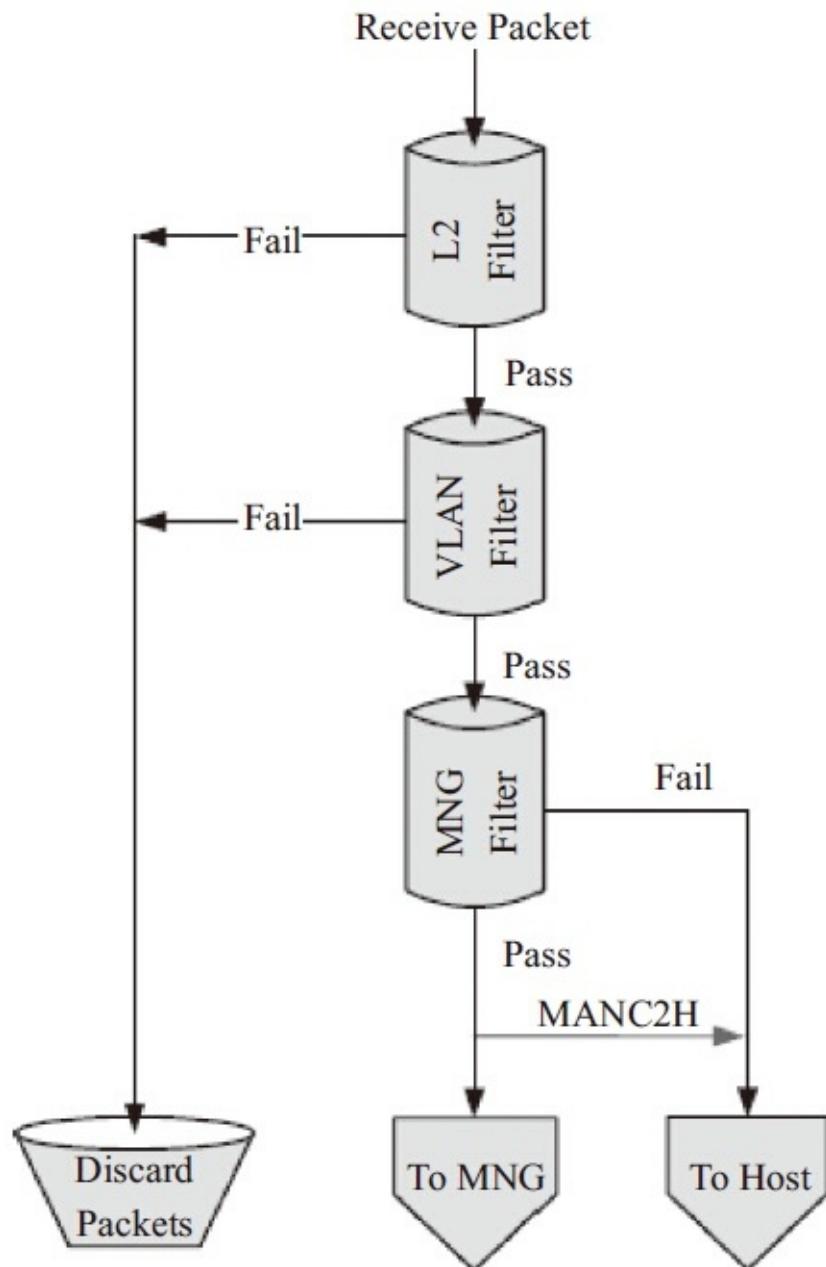
### 服务质量

多队列应用于服务质量（QoS）流量类别：把发送队列分配给不同的流量类别，可以让网卡在发送侧做调度；把收包队列分配给不同的流量类别，可以做到基于流的限速。



### 流过滤

来自外部的数据包哪些是本地的、可以被接收的，哪些是不可以被接收的？可以被接收的数据包会被网卡送到主机或者网卡内置的管理控制器，其过滤主要集中在以太网的二层功能，包括VLAN及MAC过滤。



## 应用

针对Intel®XL710网卡，PF使用i40e Linux Kernel驱动，VF使用DPDK i40e PMD驱动。使用Linux的Ethtool工具，可以完成配置操作cloud filter，将大量的数据包直接分配到VF的队列中，交由运行在VF上的虚机应用来直接处理。

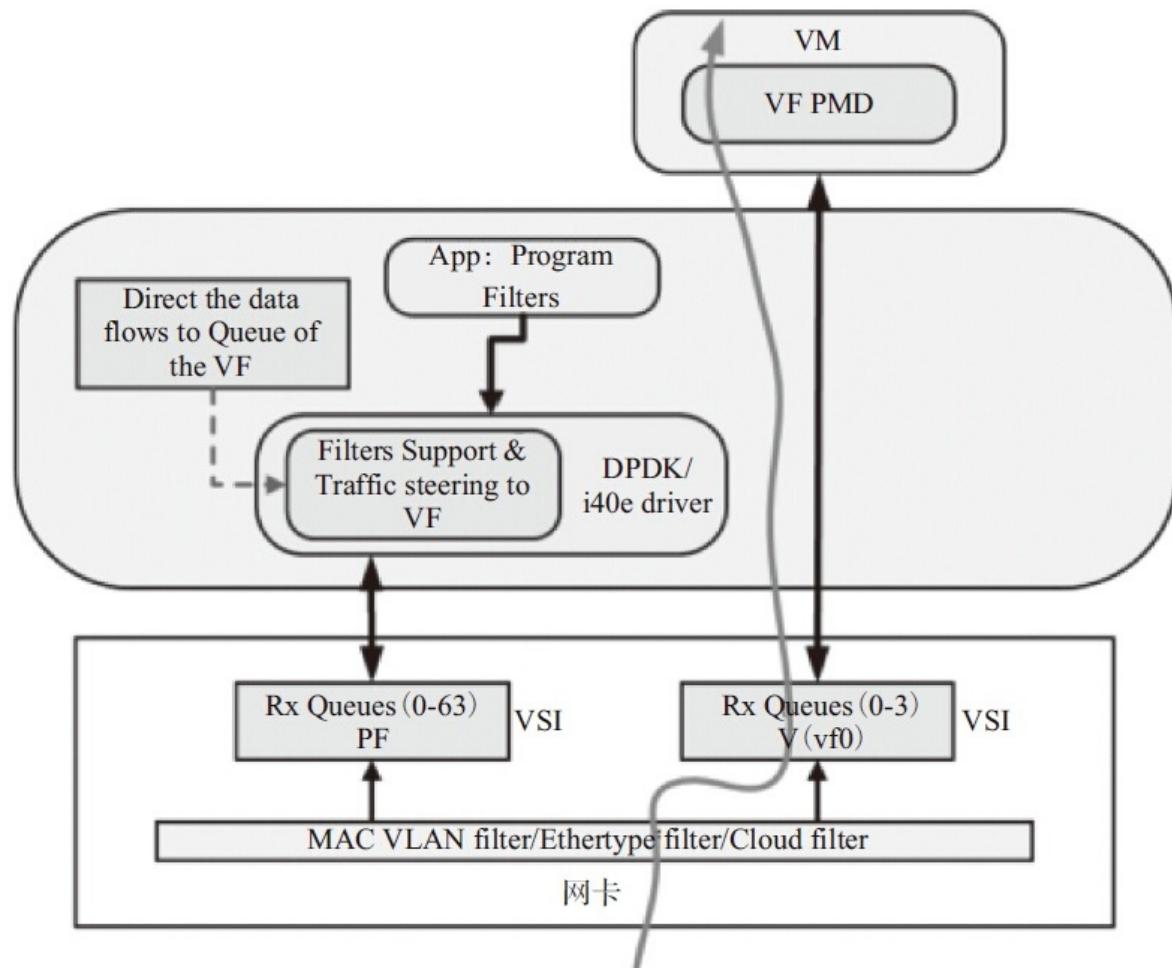
```

echo 1 > /sys/bus/pci/devices/0000:02:00.0/sriov_numvfs
modprobe pci-stub
echo "8086 154c" > /sys/bus/pci/drivers/pci-stub/new_id
echo 0000:02:02.0 > /sys/bus/pci/devices/0000:2:02.0/driver/unbind
echo 0000:02:02.0 > /sys/bus/pci/drivers/pci-stub/bind

qemu-system-x86_64 -name vm0 -enable-kvm -cpu host -m 2048 -smp 4 -drive file=dpdk-vm0
.img -vnc :4 -device pci-assign,host=02:02.0

ethtool -N ethx flow-type ip4 dst-ip 2.2.2.2 user-def 0xffffffff00000000 action 2 loc
1

```



# 硬件加速与功能卸载

## 网卡硬件卸载功能

各种网卡支持的硬件卸载的功能：

		i350	82599	x550	x1710
计算及更新	VLAN	✓	✓	✓	✓
	Double VLAN	✓	✓	✓	✓
	IEEE1588	✓	✓	✓	✓
	IP/TCP/UDP/SCTP 的 Checksum Offload	✓	✓	✓	✓
	VXLAN & NVGRE Support			✓	
分片	TCP Segmentation Offload	✓	✓	✓	✓
组包	RSC			✓	

DPDK提供了硬件卸载的接口，利用rte\_mbuf数据结构里的64位的标识（ol\_flags）来表征卸载与状态

接收时：

ol-flags 解码信息	功能解释
PKT_RX_VLAN_PKT	接收包带有 VLAN 信息，VLAN 标识被剥离到 Mbuf 中
PKT_RX_RSS_HASH	接收包带有 RSS 的哈希运算结果在 Mbuf 中
PKT_RX_FDIR	接收包带有 FDIR 的信息，在 Mbuf 中
PKT_RX_L4_CKSUM_BAD PKT_RX_IP_CKSUM_BAD	接收侧进行了 checksum 的检查，报文正确性在此显示
PKT_RX_IEEE1588_PTP; PKT_RX_IEEE1588_TMST	IEEE1588 卸载

发送时：

ol-flags 解码信息	功能解释
PKT_TX_VLAN_PKT	发送时插入 VLAN 标识，VLAN 标识已经在 Mbuf 中
PKT_TX_IP_CKSUM PKT_TX_TCP_CKSUM PKT_TX_SCTP_CKSUM PKT_TX_UDP_CKSUM PKT_TX_OUTER_IP_CKSUM PKT_TX_TCP_SEG PKT_TX_IPV4 PKT_TX_IPV6 PKT_TX_OUTER_IPV4 PKT_TX_OUTER_IPV6	发送时进行 checksum 计算，插入协议头部的 Checksum 字段。这些标志可以用在 TSO, VXLAN/NVGRE 协议的场景下
PKT_TX_IEEE1588_PTP;	IEEE1588 卸载

## VLAN硬件卸载

如果由软件完成VLAN Tag的插入将会给CPU带来额外的负荷，涉及一次额外的内存拷贝（报文内容复制），最坏场景下，这可能是上百周期的开销。大多数网卡硬件提供了VLAN卸载的功能。

接收侧针对VLAN进行包过滤

网卡最典型的卸载功能之一就是在接收侧针对VLAN进行包过滤，在DPDK中app/testpmd提供了测试命令与实现代码

Testpmd 的命令	功能解释
vlan set filter (on off) (port_id)	打开或者关闭端口的 VLAN 过滤功能。不匹配 VLAN 过滤表的 VLAN 包，会被丢弃。
rx_vlan set tpid (value) (port_id)	设置 VLAN 过滤的 TPID 选项。支持多个 TPID
rx_vlan add (vlan_id all) (port_id)	添加过滤的 VLAN ID，可以添加多个 VLAN, 最大支持 VLAN 的表现由网卡数据手册限定
rx_vlan rm (vlan_id all) (port_id)	删除单个或者所有 VLAN 过滤表项
rx_vlan add (vlan_id) port (port_id) vf (vf_mask)	添加 port/vf 设置 VLAN 过滤表
rx_vlan rm (vlan_id) port (port_id) vf (vf_mask)	删除 port/vf 设置 VLAN 过滤表

DPDK的app/testpmd提供了如何基于端口使能与去使能的测试命令。

```
testpmd> vlan set strip (on|off) (port_id)
testpmd> vlan set stripq (on|off) (port_id,queue_id)
```

发包时VLAN Tag的插入

在DPDK中，在调用发送函数前，必须提前设置mbuf数据结构，设置 `PKT_TX_VLAN_PKT` 位，同时将具体的Tag信息写入 `vlan_tci` 字段。

### 多层VLAN

现代网卡硬件大多提供对两层VLAN Tag进行卸载，如VLAN Tag的剥离、插入。DPDK的app/testapp应用中提供了测试命令。网卡数据手册有时也称VLAN Extend模式。

## IEEE588协议

DPDK提供的是打时间戳和获取时间戳的硬件卸载。需要注意，DPDK的使用者还是需要自己去管理IEEE1588的协议栈，DPDK并没有实现协议栈。

## IP/TCP/UDP/SCTP checksum硬件卸载功能

checksum在收发两个方向上都需要支持，操作并不一致，在接收方向上，主要是检测，通过设置端口配置，强制对所有达到的数据报文进行检测，即判断哪些包的checksum是错误的，对于这些出错的包，可以选择将其丢弃，并在统计数据中体现出来。在DPDK中，和每个数据包都有直接关联的是rte\_mbuf，网卡自动检测进来的数据包，如果发现checksum错误，就会设置错误标志。软件驱动会查询硬件标志状态，通过mbuf中的`ol_flags`字段来通知上层应用。

## Tunnel硬件卸载功能

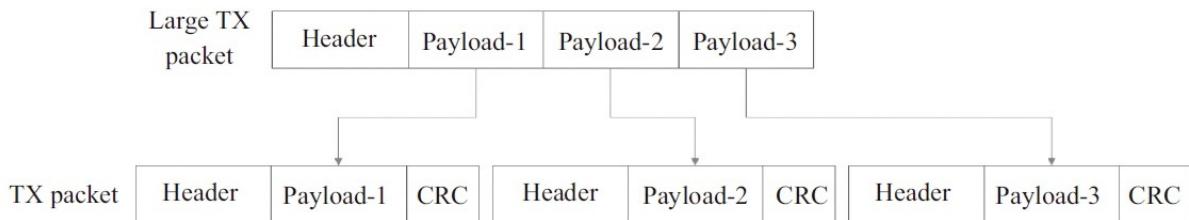
目前DPDK仅支持对VxLAN和NVGRE的流进行重定向：基于VxLAN和NVGRE的特定信息，TNI或VNI，以及内层的MAC或IP地址进行重定向。

在dpdk/testpmd中，可以使用相关的命令行来使用VxLAN和NVGRE的数据流重定向功能，如下所示：

```
flow_director_filter X mode Tunnel add/del/update mac XX:XX:XX:XX:XX:XX vlan XXXX tunnel NVGRE/VxLAN tunnel-id XXXX flexbytes (X,X) fwd/drop queue X fd_id X
```

## TSO

TSO (TCP Segment Offload) 是TCP分片功能的硬件卸载，显然这是发送方向的功能。硬件提供的TCP分片硬件卸载功能可以大幅减轻软件对TCP分片的负担。

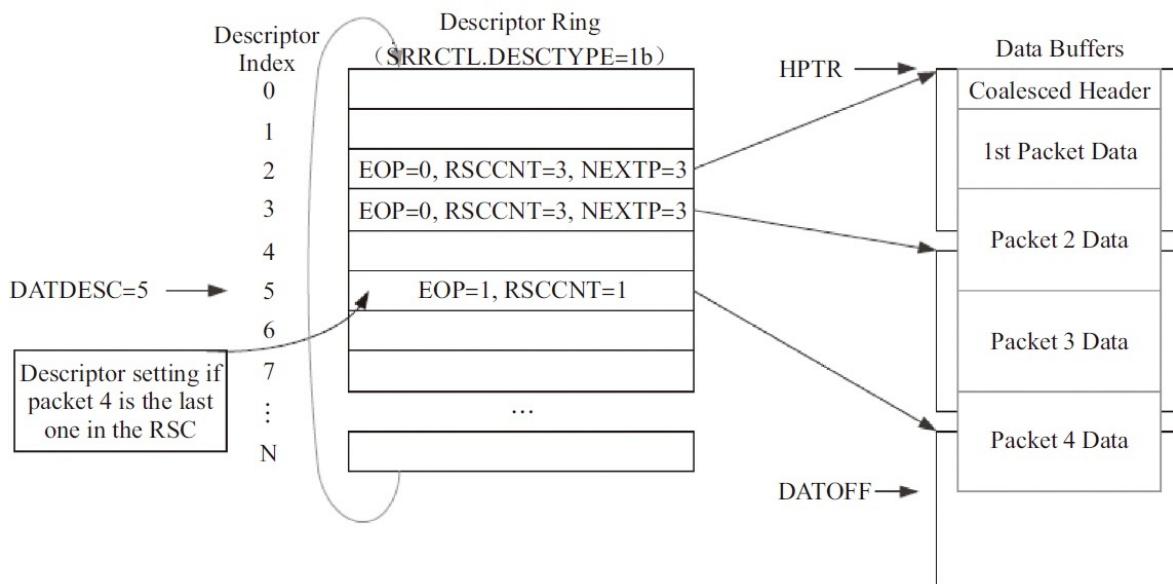


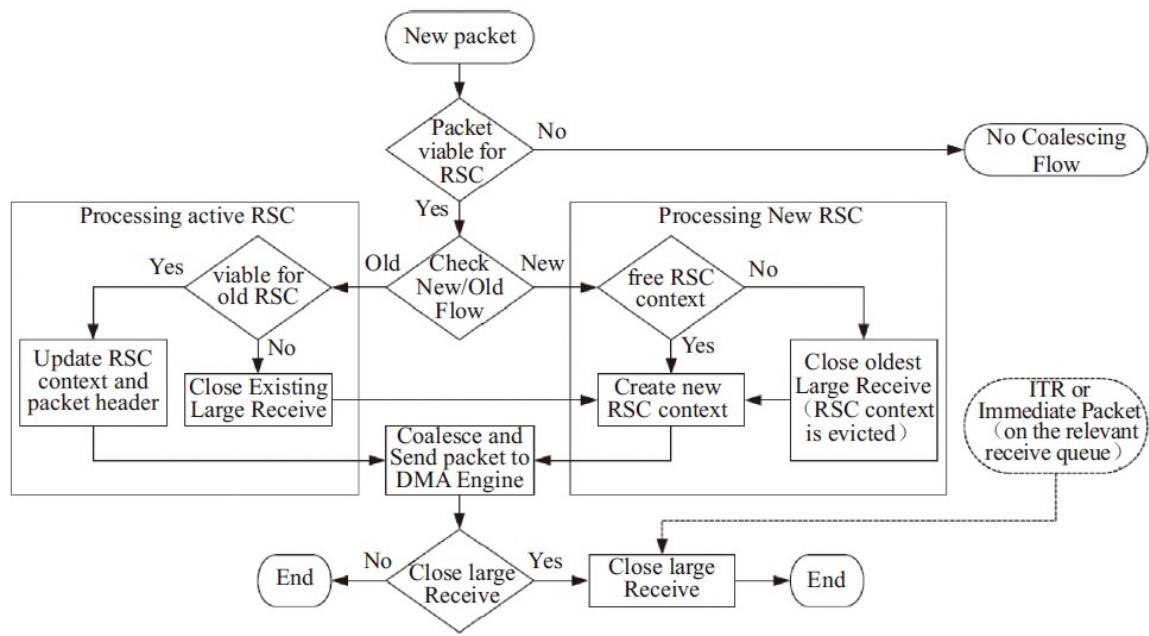
在dpdk/testpmd中提供了两条TSO相关的命令行：

- 1) tso set 14000：用于设置tso分片大小。
- 2) tso show 0：用于查看tso分片的大小。

## RSC

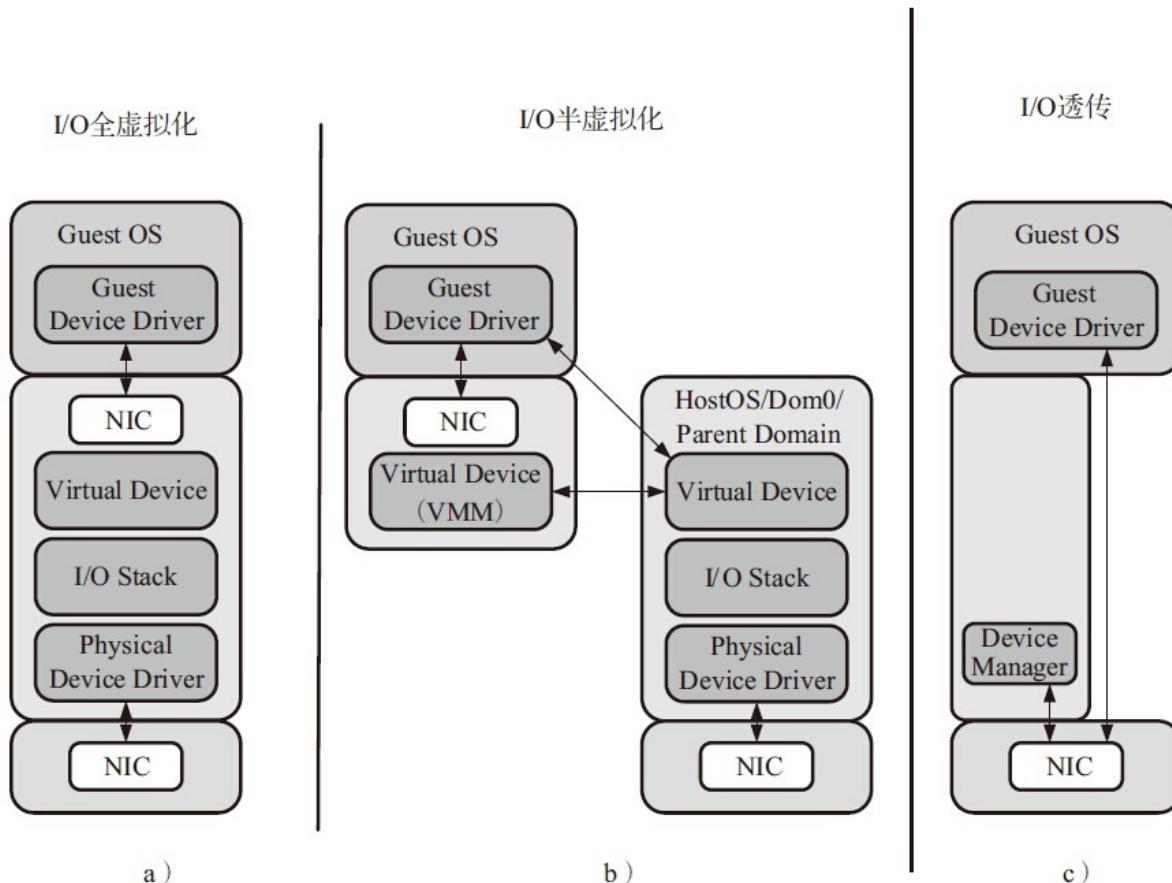
RSC (Receive Side Coalescing，接收方聚合) 是TCP组包功能的硬件卸载。硬件组包功能实际上是硬件拆包功能的逆向功能。硬件组包功能针对TCP实现，是接收方向的功能，可以将拆分的TCP分片聚合成一个大的分片，从而减轻软件的处理。





# 网络虚拟化

I/O虚拟化包括管理虚拟设备和共享的物理硬件之间I/O请求的路由选择。目前，实现I/O虚拟化有三种方式：I/O全虚拟化、I/O半虚拟化和I/O透传。



- 全虚拟化：宿主机截获客户机对I/O设备的访问请求，然后通过软件模拟真实的硬件。这种方式对客户机而言非常透明，无需考虑底层硬件的情况，不需要修改操作系统。
- 半虚拟化：通过前端驱动/后端驱动模拟实现I/O虚拟化。客户机中的驱动程序为前端，宿主机提供的与客户机通信的驱动程序为后端。前端驱动将客户机的请求通过与宿主机间的特殊通信机制发送给后端驱动，后端驱动在处理完请求后再发送给物理驱动。
- IO透传：直接把物理设备分配给虚拟机使用，这种方式需要硬件平台具备I/O透传技术，例如Intel VT-d技术。它能获得近乎本地的性能，并且CPU开销不高。

DPDK支持半虚拟化的前端virtio和后端vhost，并且对前后端都有性能加速的设计，这些将分别在后面两章介绍。而对于I/O透传，DPDK可以直接在客户机里使用，就像在宿主机里，直接接管物理设备，进行操作。

## I/O透传

I/O透传带来的好处是高性能，几乎可以获得本机的性能，这个主要是因为Intel®VT-d的技术支持，在执行IO操作时大量减少甚至避免VM-Exit陷入到宿主机中。目前只有PCI和PCI-e设备支持Intel®VT-d技术。可以配合SR-IOV使用，让一个网卡生成多个独立的虚拟网卡，把这些虚拟网卡分配给每一个客户机，可以获得相对好的性能。

## VT-d

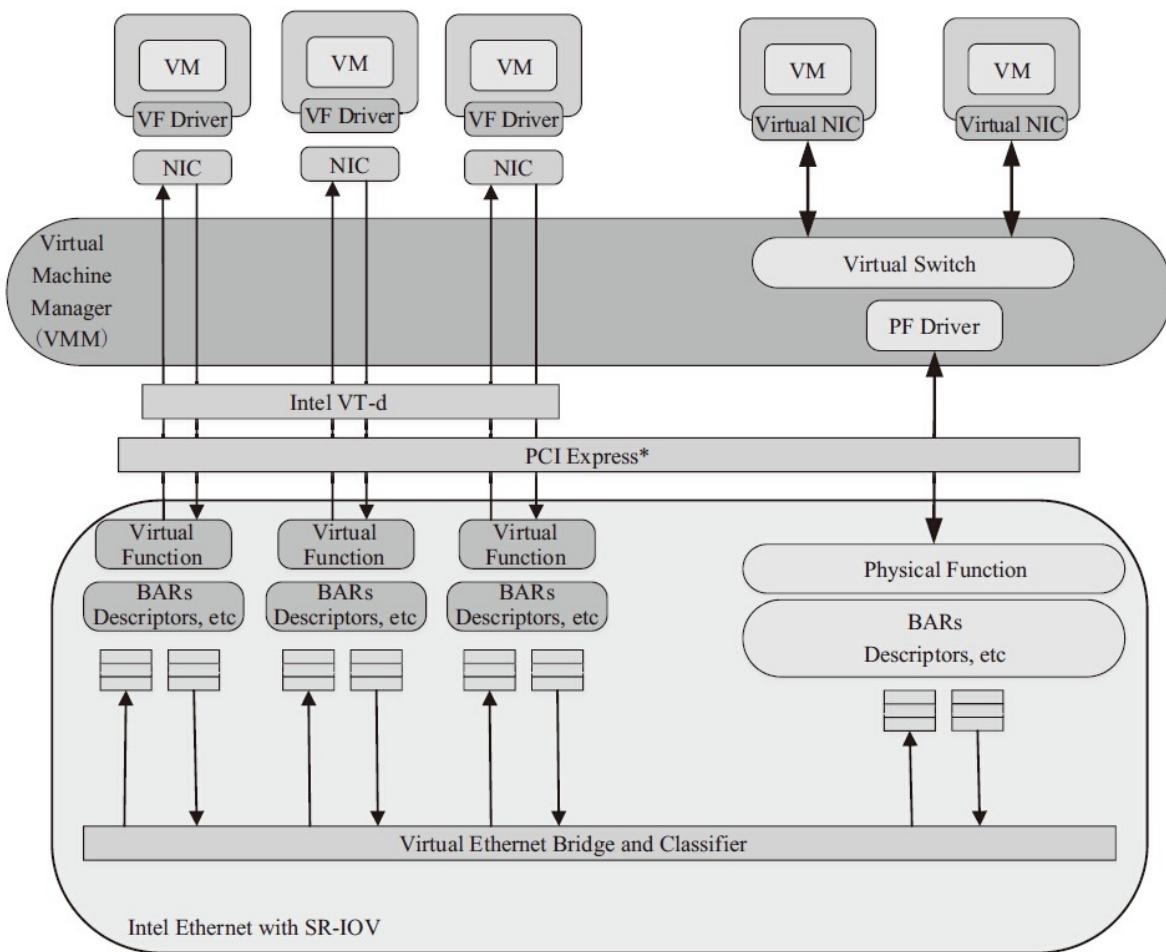
VT-d主要给宿主机软件提供了以下的功能：

- I/O设备的分配：可以灵活地把I/O设备分配给虚拟机，把对虚拟机的保护和隔离的特性扩展到IO的操作上来。
- DMA重映射：可以支持来自设备DMA的地址翻译转换。
- 中断重映射：可以支持来自设备或者外部中断控制器的中断的隔离和路由到对应的虚拟机。
- 可靠性：记录并报告DMA和中断的错误给系统软件，否则的话可能会破坏内存或影响虚拟机的隔离。

## SR-IOV

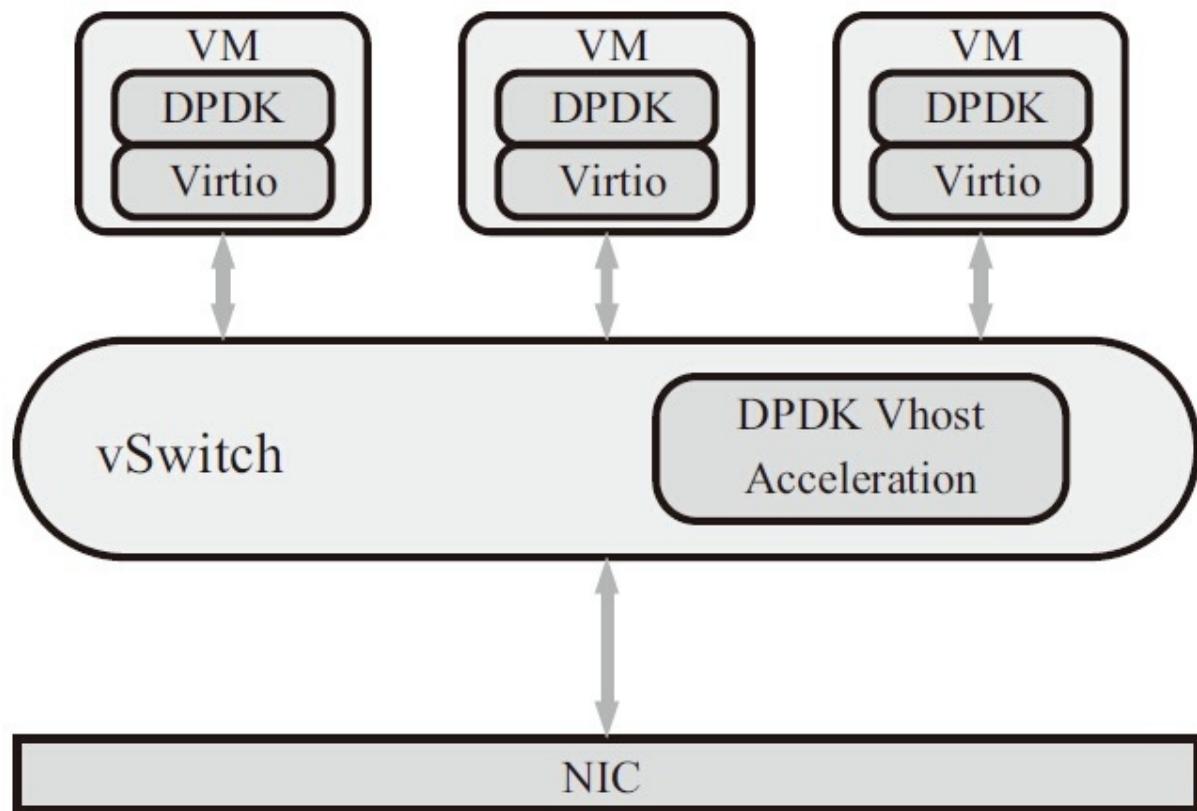
SR-IOV技术是由PCI-SIG制定的一套硬件虚拟化规范，全称是Single Root IO Virtualization（单根IO虚拟化）。SR-IOV规范主要用于网卡（NIC）、磁盘阵列控制器（RAID controller）和光纤通道主机总线适配器（Fibre Channel Host Bus Adapter，FC HBA），使数据中心达到更高的效率。SR-IOV架构中，一个I/O设备支持最多256个虚拟功能，同时将每个功能的硬件成本降至最低。SR-IOV引入了两个功能类型：

- PF（Physical Function，物理功能）：这是支持SR-IOV扩展功能的PCIe功能，主要用于配置和管理SR-IOV，拥有所有的PCIe设备资源。PF在系统中不能被动态地创建和销毁（PCI Hotplug除外）。
- VF（Virtual Function，虚拟功能）：“精简”的PCIe功能，包括数据迁移必需的资源，以及经过谨慎精简的配置资源集，可以通过PF创建和销毁。



## virtio

在客户机操作系统中实现的前端驱动程序一般直接叫Virtio，在宿主机实现的后端驱动程序目前常用的叫vhost。与宿主机纯软件模拟I/O（如e1000、rtl8139）设备相比，virtio可以获得很好的I/O性能。但其缺点是必须要客户机安装特定的virtio驱动使其知道是运行在虚拟化环境中。



### 常见的virtio设备

Virtio Device ID	Virtio Device
0	reserved (invalid)
1	network card
2	block device
3	console
4	entropy source
5	memory ballooning (traditional)
6	ioMemory
7	rpmsg
8	SCSI host
9	9P transport
10	mac80211 wlan
11	rproc serial
12	virtio CAIF
13	memory balloon
16	GPU device
17	Timer/Clock device
18	Input device

### Virtio网络设备Linux内核驱动设计

Virtio网络设备Linux内核驱动主要包括三个层次：底层PCI-e设备层，中间Virtio虚拟队列层，上层网络设备层。

## PCI-e设备层

drivers/virtio/virtio.c, virtio\_pci\_common.c,  
virtio\_pci\_legacy.c, virtio\_pci\_modern.c

virtio\_driver  
  
-<<probe>>  
-<<remove>>  
-register\_virtio\_driver  
-unregister\_virtio\_driver

virtio\_device  
  
-register\_virtio\_device  
-unregister\_virtio\_device

virtio\_pci\_device  
  
-<<setup\_vq>>  
-<<del\_vq>>  
-<<config\_vector>>  
-virtio\_pci\_probe

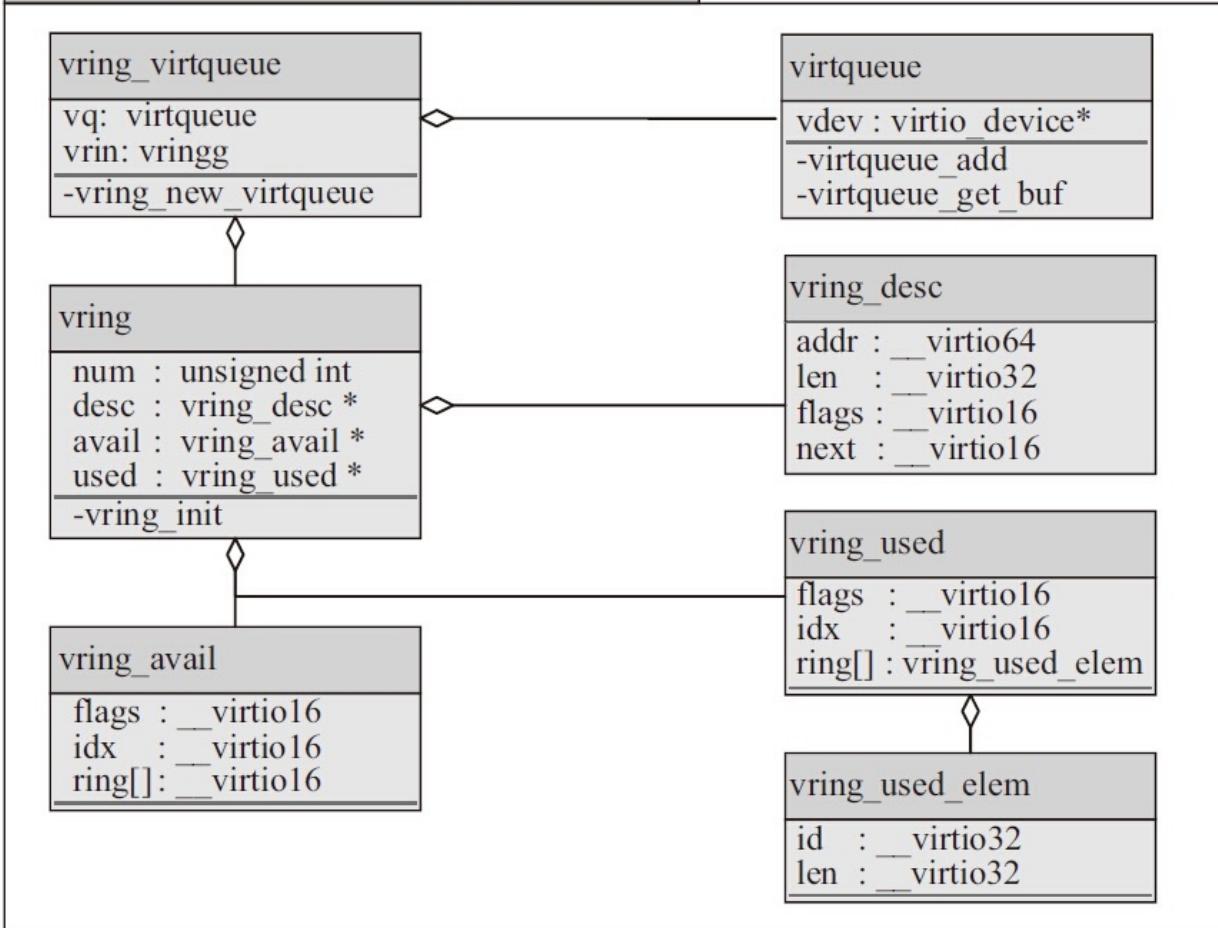
virtio\_pci\_modern\_device  
  
-setup\_vq  
-del\_vq  
-vq\_config\_vector  
-virtio\_pci\_modern\_probe

virtio\_pci\_legacy\_device  
  
-setup\_vq  
-del\_vq  
-vq\_config\_vector  
-virtio\_pci\_legacy\_probe



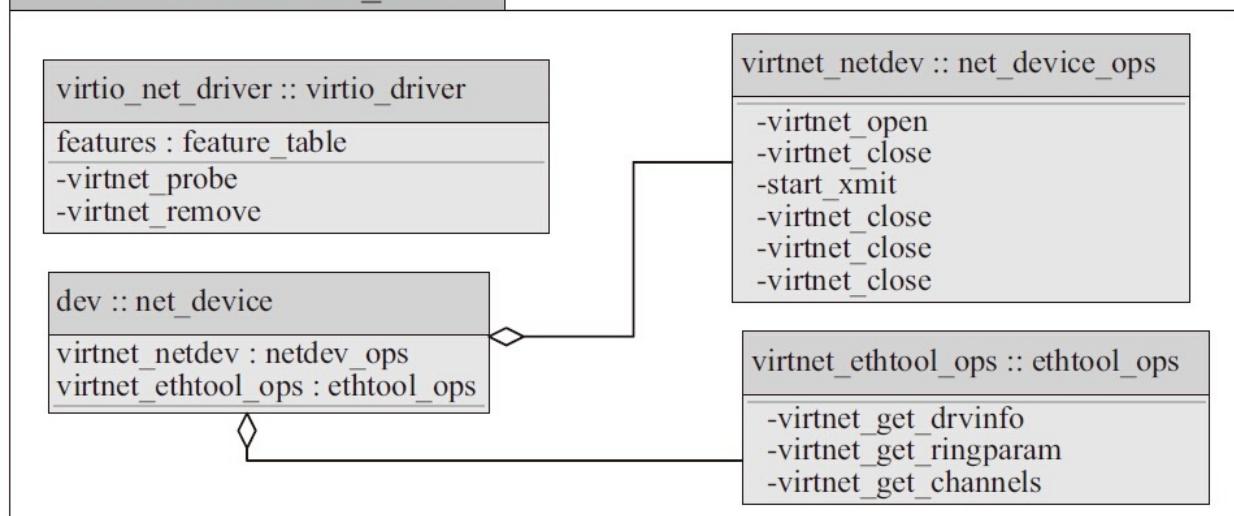
## 虚拟队列层

drivers/virtio/virtio\_ring.c



## 网络设备层

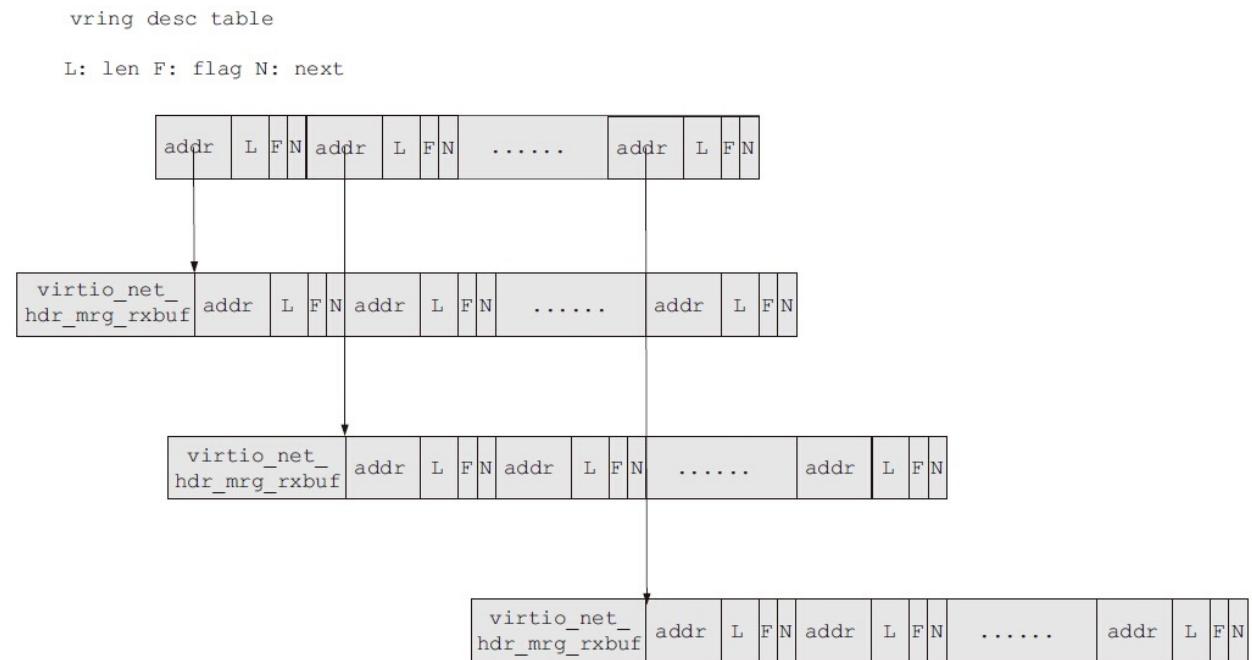
drivers/net/virtio\_net.c



## DPDK用户空间virtio设备的优化

DPDK用户空间驱动和Linux内核驱动相比，主要不同点在于DPDK只暂时实现了Virtio网卡设备，所以整个构架和优化上面可以暂时只考虑网卡设备的应用场景。

- 关于单帧mbuf的网络包收发优化：固定了可用环表表项与描述符表项的映射，即可用环表所有表项head\_idx指向固定的vring描述符表位置（对于接收过程，可用环表0->描述符表0，1->1，…，255->255的固定映射；对于发送过程，0->128，1->129，…127->255，128->128，129->129，…255->255的固定映射，描述符表0~127指向mbuf的数据区域，描述符表128~255指向virtio net header的空间），对可用环表的更新只需要更新环表自身的指针。固定的可用环表除了能够避免不同核之间的CACHE迁移，也节省了vring描述符的分配和释放操作，并为使用SIMD指令进行进一步加速提供了便利。
- Indirect特性在网络包发送中的支持：如前面介绍，发送的包至少需要两个描述符。通过支持indirect特性，任何一个要发送的包，无论单帧还是巨型帧（巨型帧的介绍见6.6.1节）都只需要一个描述符，该描述符指向一块驱动程序额外分配的间接描述符表的内存区域。

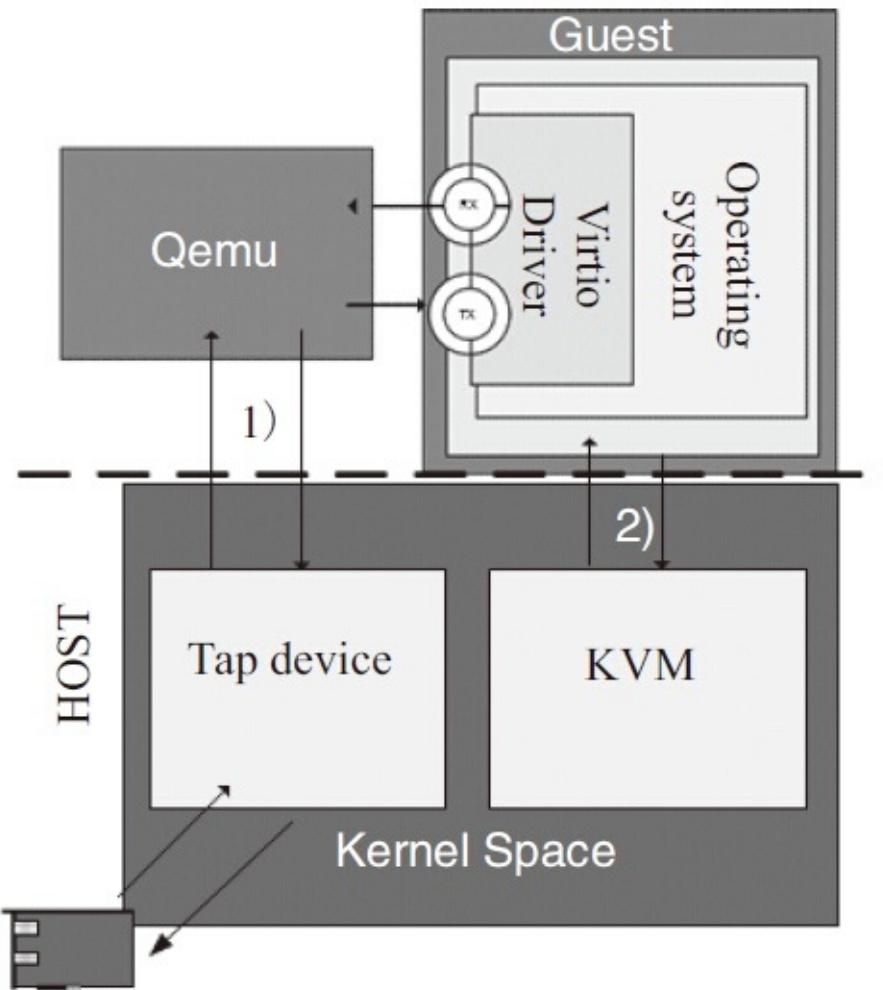


## vhost

virtio-net的后端驱动经历过从virtio-net后端，到内核态vhost-net，再到用户态vhost-user的演进过程。

## virtio-net

virtio-net后端驱动的最基本要素是虚拟队列机制、消息通知机制和中断机制。虚拟队列机制连接着客户机和宿主机的数据交互。消息通知机制主要用于从客户机到宿主机的消息通知。中断机制主要用于从宿主机到客户机的中断请求和处理。

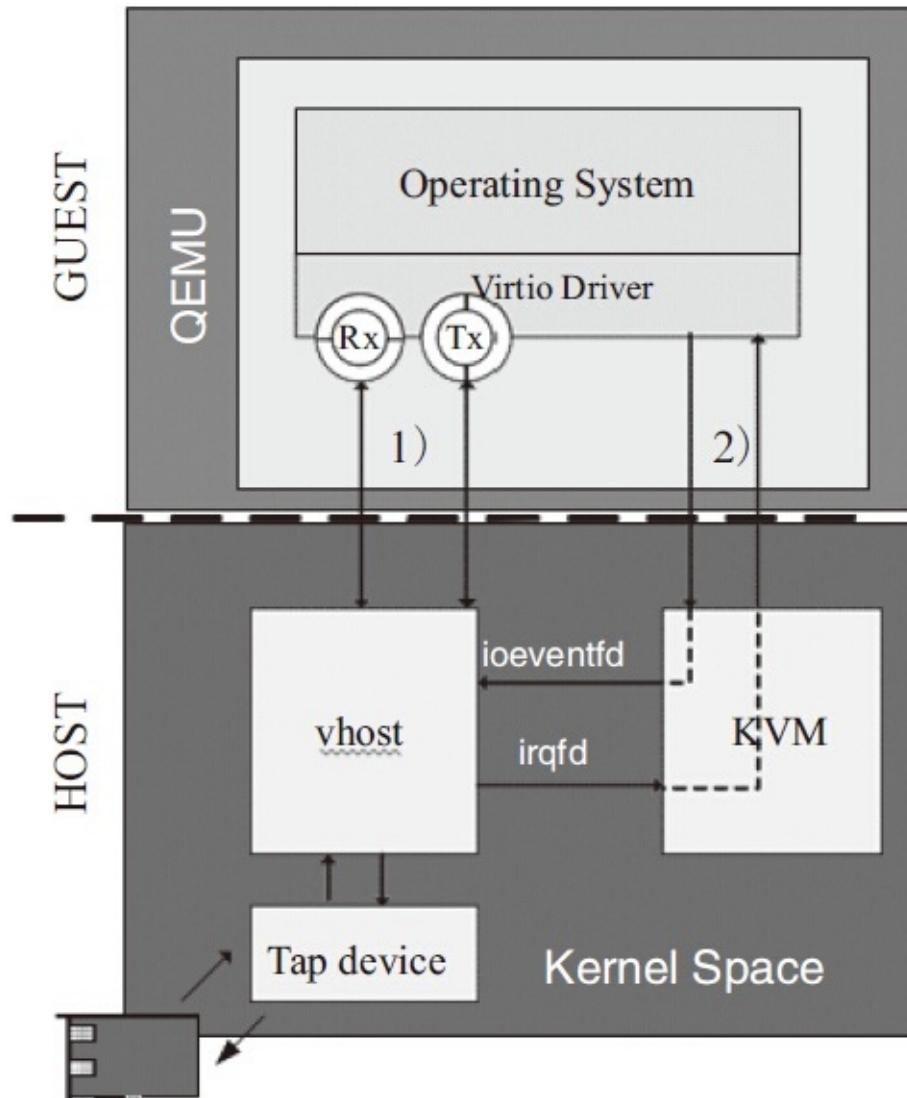


性能瓶颈主要存在于数据通道和消息通知路径这两块：

- 1) 数据通道是从Tap设备到Qemu的报文拷贝和Qemu到客户机的报文拷贝，两次报文拷贝导致报文接收和发送上的性能瓶颈。
- 2) 消息通知路径是当报文到达Tap设备时内核发出并送到Qemu的通知消息，然后Qemu利用IOCTL向KVM请求中断，KVM发送中断到客户机。

## Linux内核态vhost-net

vhost-net通过卸载virtio-net在报文收发处理上的工作，使Qemu从virtio-net的虚拟队列工作中解放出来，减少上下文切换和数据包拷贝，进而提高报文收发的性能。除此以外，宿主机上的vhost-net模块还需要承担报文到达和发送消息通知及中断的工作。



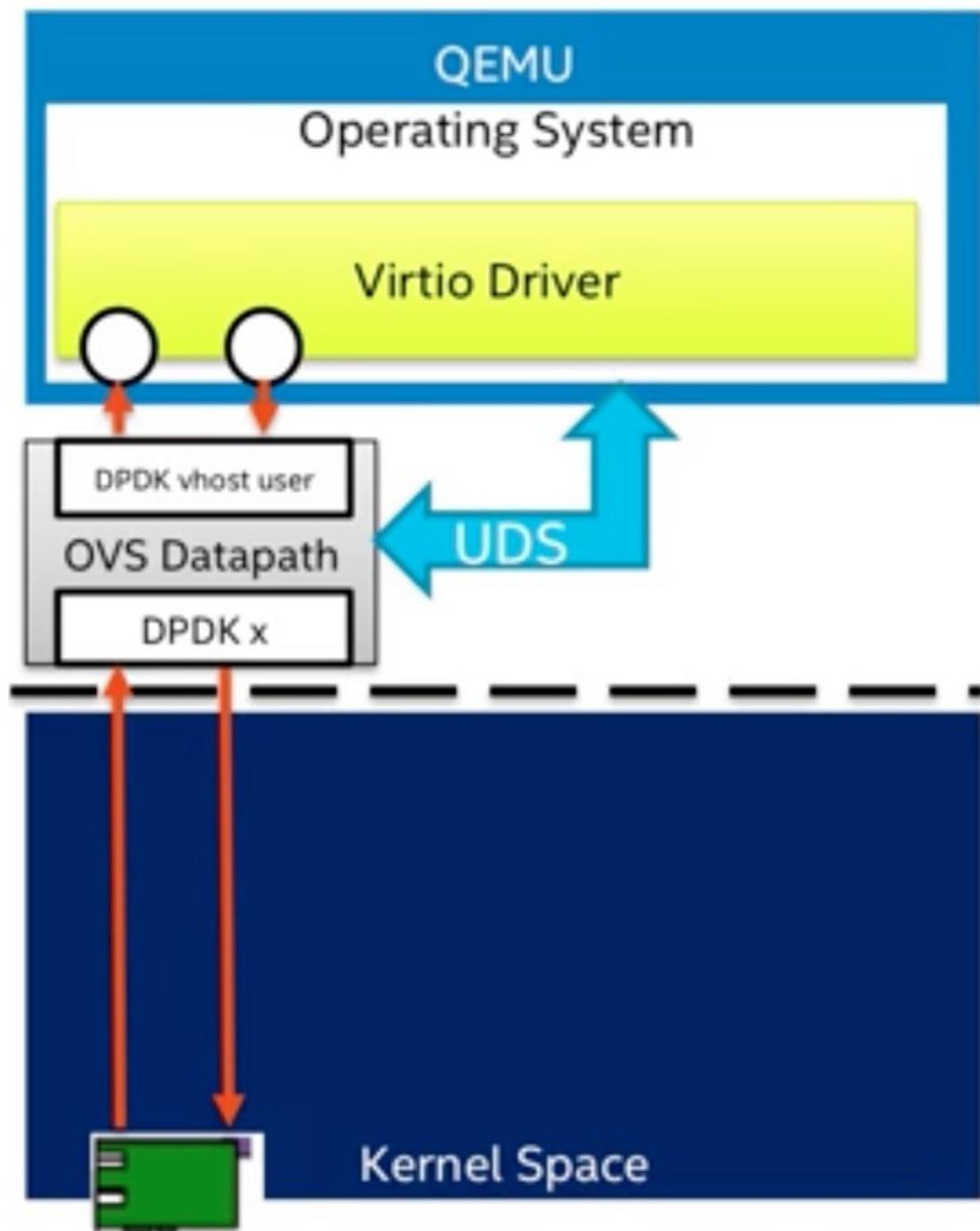
报文接收仍然包括数据通路和消息通知路径两个方面：

- 1) 数据通路是从Tap设备接收数据报文，通过vhost-net模块把该报文拷贝到虚拟队列中的数据区，从而使客户机接收报文。
- 2) 消息通路是当报文从Tap设备到达vhost-net时，通过KVM模块向客户机发送中断，通知客户机接收报文。

## 用户态vhost

Linux内核态的vhost-net模块需要在内核态完成报文拷贝和消息处理，这会给报文处理带来一定的性能损失，因此用户态的vhost应运而生。用户态vhost采用了共享内存技术，通过共享的虚拟队列来完成报文传输和控制，大大降低了vhost和virtio-net之间的数据传输成本。

- 1) 数据通路不再涉及内核，直接通过共享内存发送给用户态应用（如DPDK，OVS）
- 2) 消息通路通过Unix Domain Socket实现



## Userspace vHost Characteristics

<u>Current</u>	<u>Future</u>
Performance	Performance
▪ Less copies and context switches	▪ Bulk operations
Security	▪ Vectorisation
▪ Virtqueues mapped to vswitchd address space	Features
Live Migration	▪ Virtio-net backend enhancements
▪ Solution exists	QEMU
Compatibility	▪ Multi-queue
▪ Virtio-net standard	

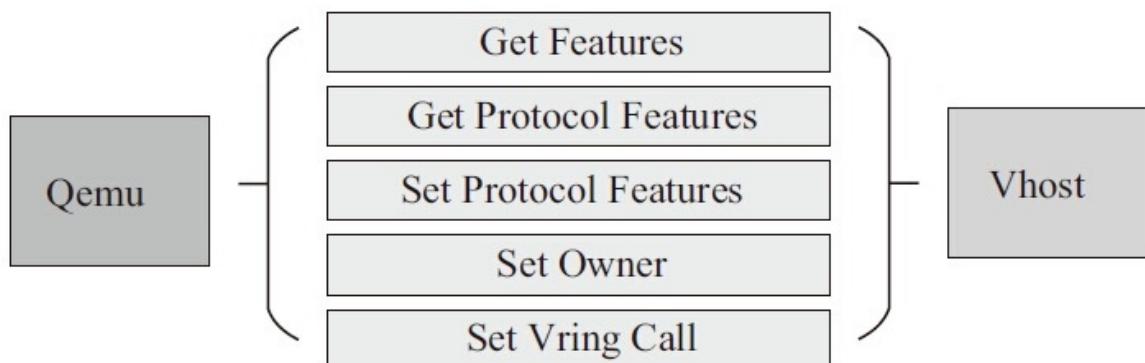
DPDK vhost同时支持Linux virtio-net驱动和DPDK virtio PMD驱动的前端。

## DPDK vhost

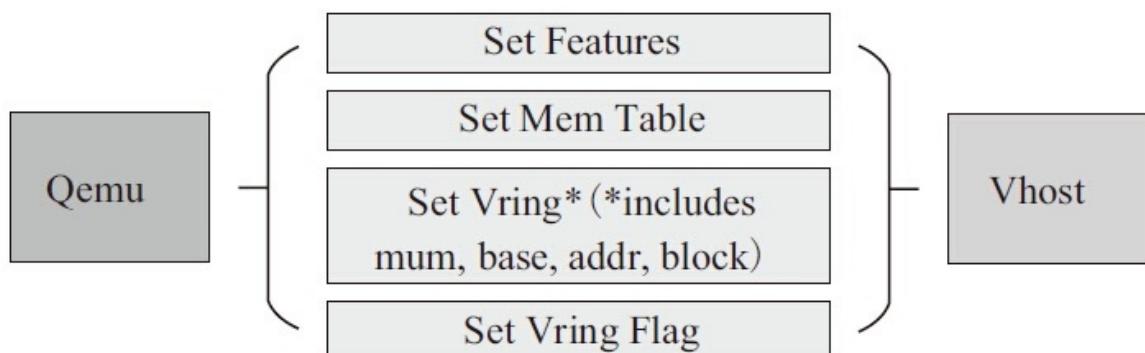
DPDK vhost支持vhost-cuse（用户态字符设备）和vhost-user（用户态socket服务）两种消息机制，它负责为客户机中的virtio-net创建、管理和销毁vhost设备。

当使用vhost-user时，首先需要在系统中创建一个Unix domain socket server，用于处理Qemu发送给vhost的消息，其消息机制如图所示。

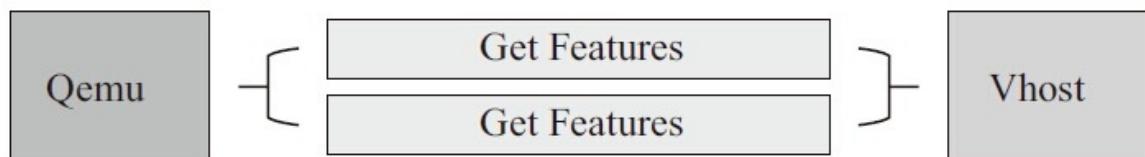
Vhost dev init:



Vhost dev start:



Vhost dev stop:

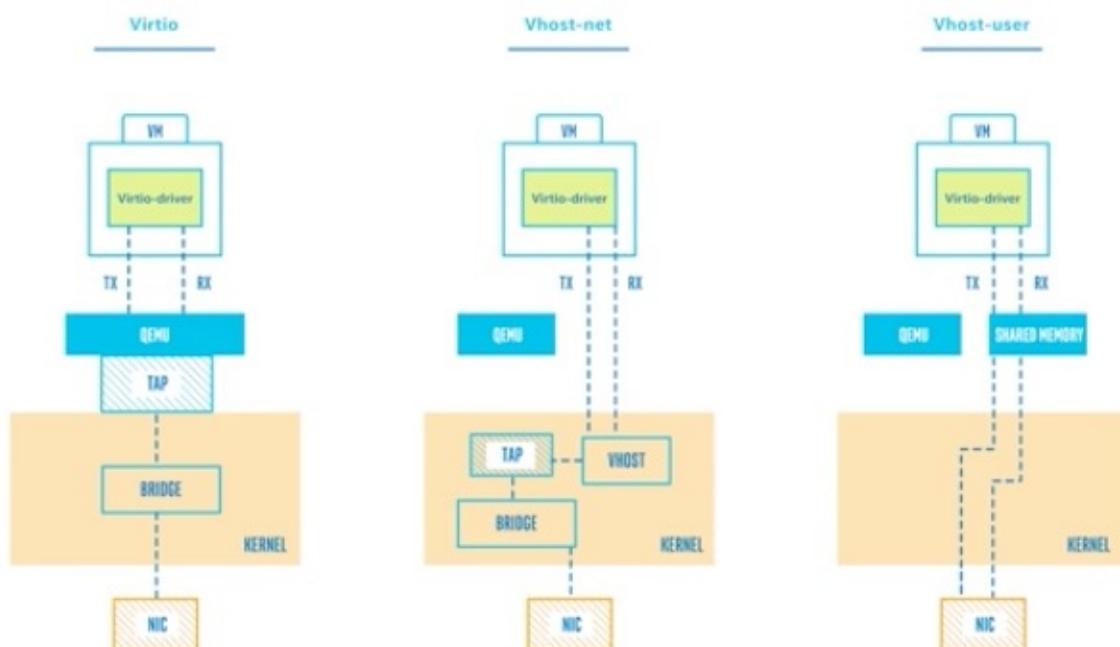


DPDK的vhost有两种封装形式：vhost lib和vhost PMD。vhost lib实现了用户态的vhost驱动供vhost应用程序调用，而vhost PMD则对vhost lib进行了封装，将其抽象成一个虚拟端口，可以使用标准端口的接口来进行管理和报文收发。

DPDK示例程序vhost-switch是基于vhost lib的一个用户态以太网交换机的实现，可以完成在virtio-net设备和物理网卡之间的报文交换。还使用了虚拟设备队列（VMDQ）技术来减少交换过程中的软件开销，该技术在网卡上实现了报文处理分类的任务，大大减轻了处理器的负担。

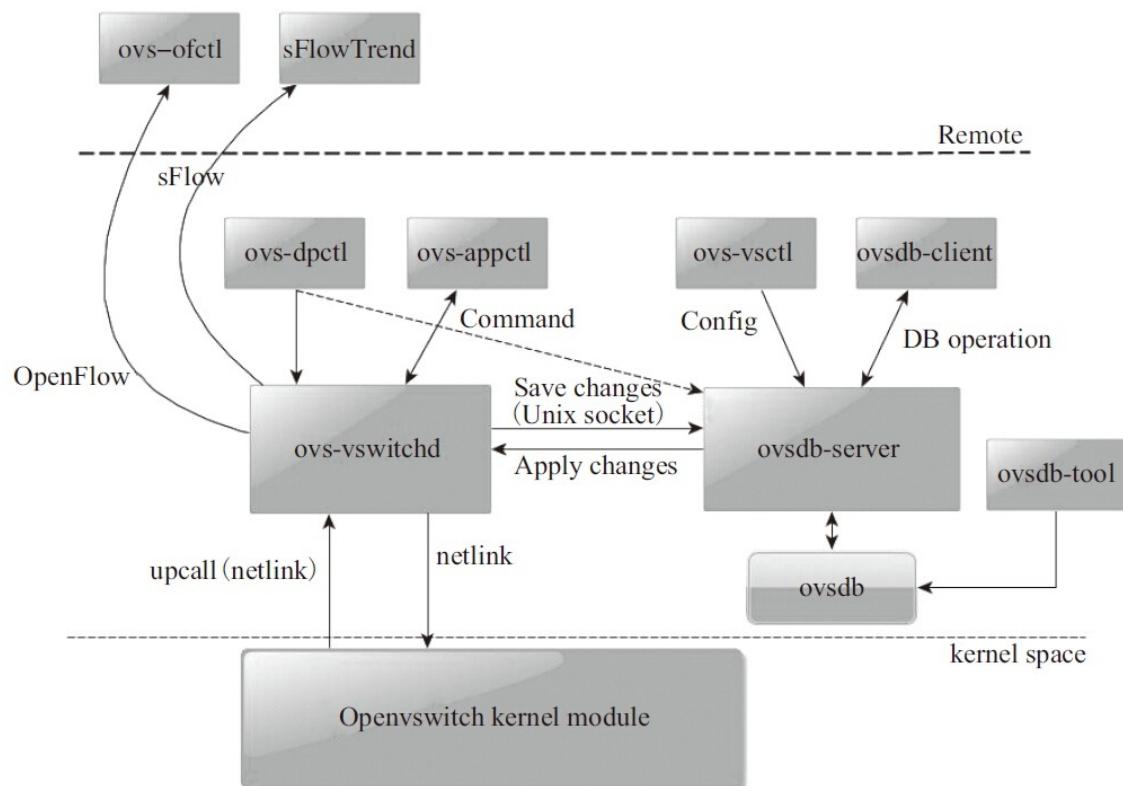
## 对比

接口	性能	操作性	维护性	安全性
IVSHMEM [ Ref13-7 ]	提供良好的性能	对共享内存的迁移需要考虑，目前没有支持	Qemu 需要打补丁来实现对 DPDK IVSHMEM 的支持；另外 Qemu 社区目前没有 IVSHMEM 的维护人员	安全性存在漏洞，需要对虚拟机可信赖
Virtio	标准的 Virtio 性能不好，DPDK 的版本提供了优化版本	DPDK 的 Vhost 可以兼容标准的 Linux Virtio 驱动；DPDK 的 Virtio 热迁移支持正在开发中；与主流的 vSwitch 软件都可对接，以保持与同一主机上的其他虚拟机连接	良好的维护性，对 Virtio 接口的优化工作持续进行，如多队列的支持	提供很好的安全性
SR-IOV VF 透传	接近于物理机上的网络性能	需要部署的网卡支持 SR-IOV 功能，并且 DPDK 的驱动也要支持；与 vSwitch 不可对接，只支持与同一主机上的其他虚拟机以基于 MAC 或 VLAN 的二层连接；基于 SR-IOV 的热迁移方案不是很成熟	良好的维护性，越来越多的网卡支持 DPDK 驱动	提供很好的安全性
物理网卡直通	接近于物理机上的网络性能	需要确定有 DPDK 的驱动；该物理网卡就不可以给同一主机上的其他虚拟机使用；基于网卡直通的热迁移方案不是很成熟	良好的维护性，越来越多的网卡支持 DPDK 驱动	提供很好的安全性

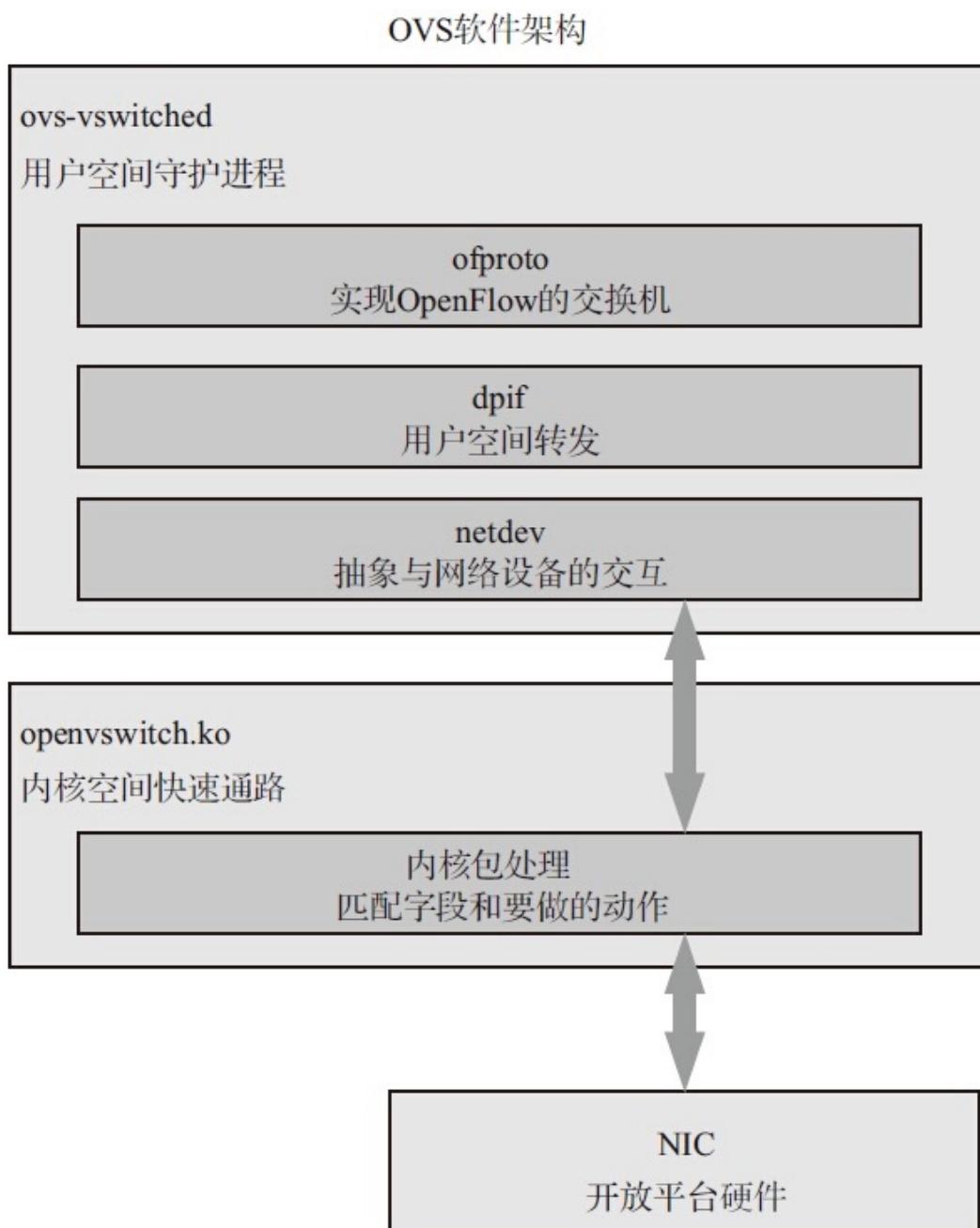


# OVS DPDK

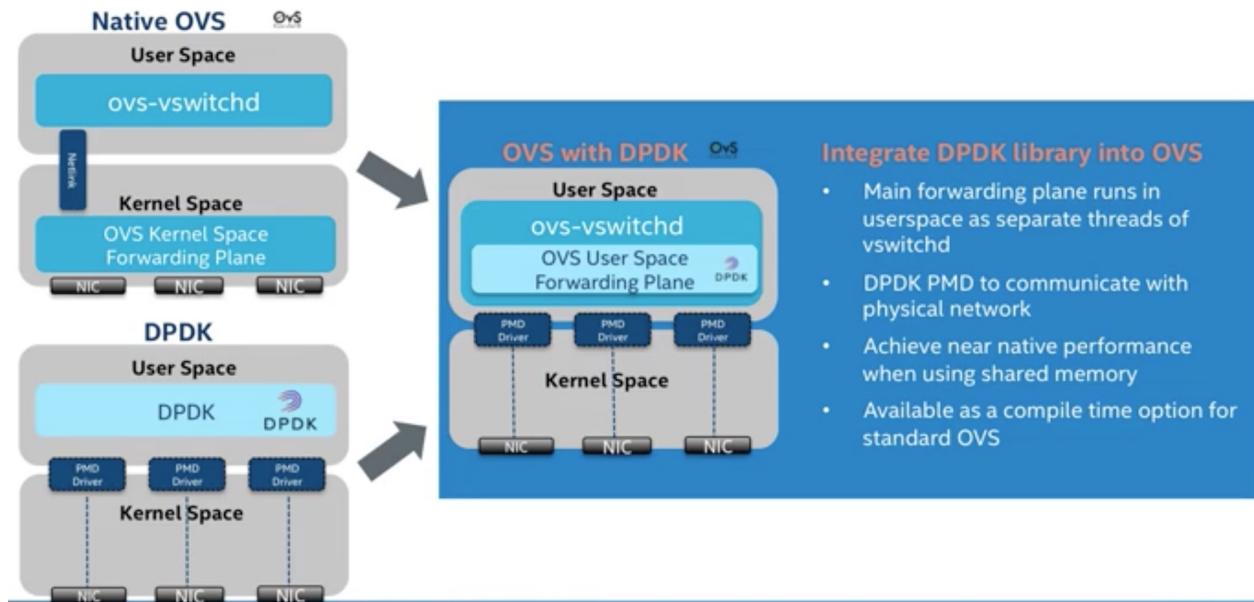
OVS在实现中分为用户空间和内核空间两个部分。用户空间拥有多个组件，它们主要负责实现数据交换和OpenFlow流表功能，还有一些工具用于虚拟交换机管理、数据库搭建以及和内核组件的交互。内核组件主要负责流表查找的快速通道。OVS的核心组件及其关联关系如图



下图显示了OVS数据通路的内部模块图



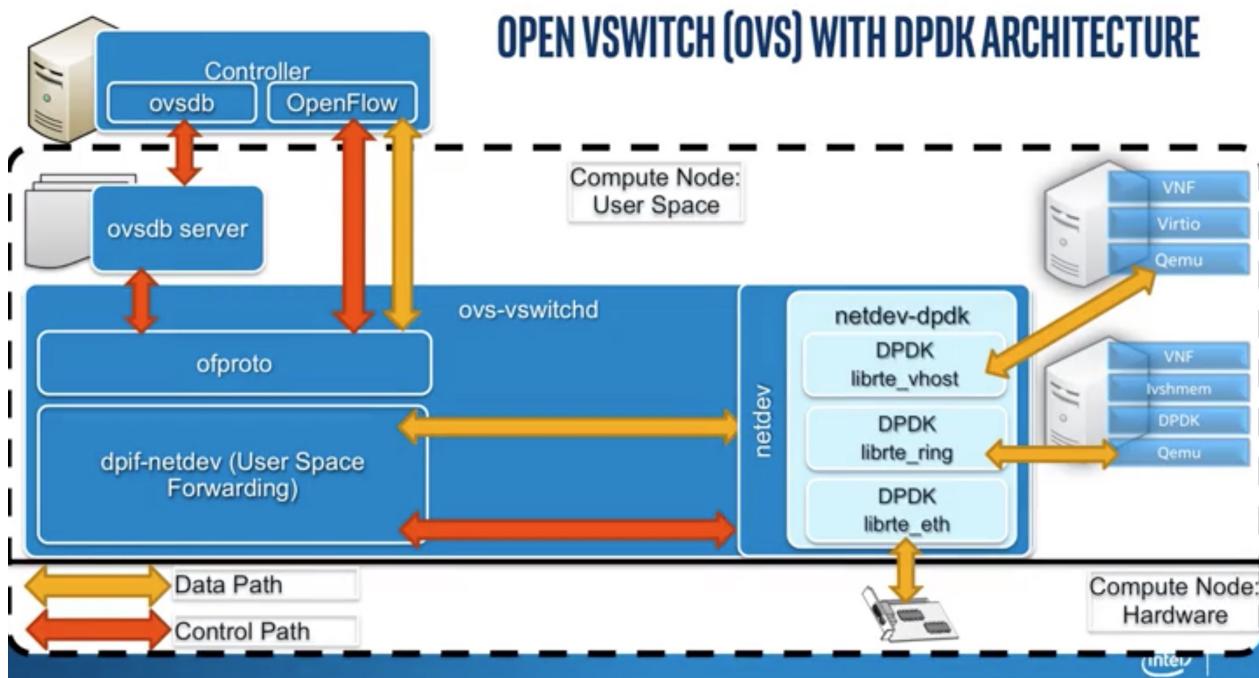
DPDK加速的思想就是专注在这个数据通路上

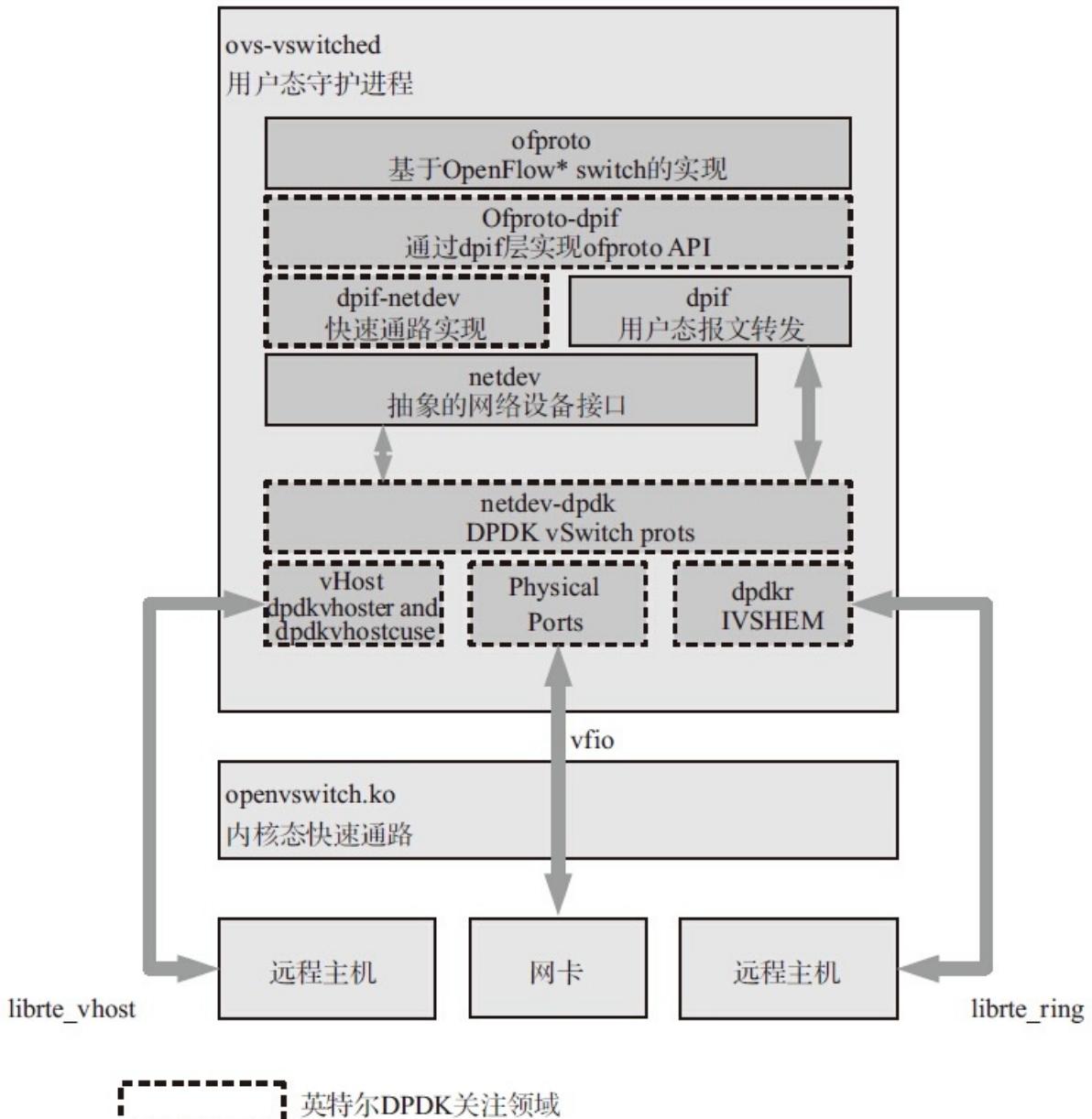


ovs-vswitchd 主要包含 ofproto、dpif、netdev 模块。ofproto 模块实现 openflow 的交换机；dpif 模块抽象一个单转发路径；netdev 模块抽象网络接口（无论物理的还是虚拟的）。

openvswitch.ko 主要由数据通路模块组成，里面包含着流表。流表中的每个表项由一些匹配字段和要做的动作组成。

## ovs with dpdk





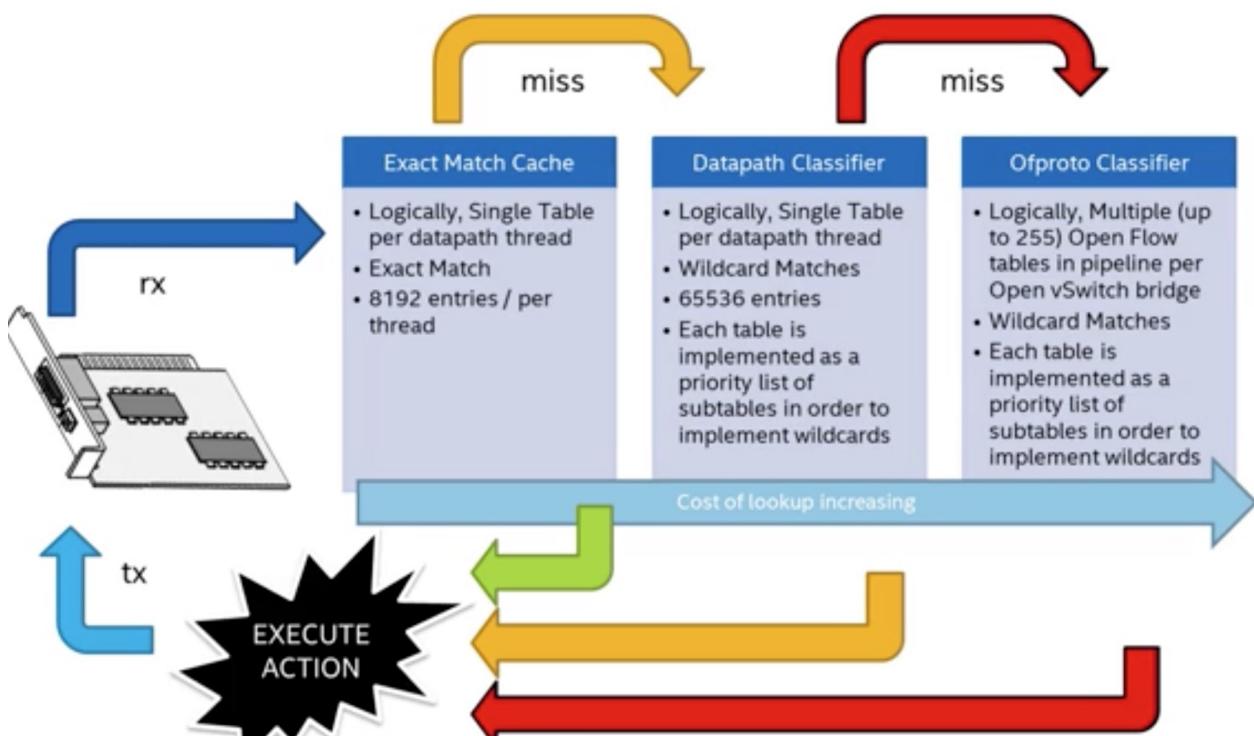
- dpif-netdev：用户态的快速通路，实现了基于netdev设备的dpif API。
- Ofproto-dpif：实现了基于dpif层的ofproto API。
- netdev-dpdk：实现了基于DPDK的netdev API，其定义的几种网络接口如下：
- dpdk物理网口：其实现是采用高性能向量化DPDK PMD的驱动。
- dpdkvhostuser与dpdkvhostcuse接口：支持两种DPDK实现的vhost优化接口：vhost-user和vhost-cuse。vhost-user或vhost-cuse可以挂接到用户态的数据通道上，与虚拟机的virtio网口快速通信。如第12章所说，vhost-cuse是一个过渡性技术，vhost-user是建议使用的接口。为了性能，在vhost burst收发包个数上，需要和dpdk物理网口设置的burst收发包个数相同。
- dpdkr：其实现是基于DPDK librte\_ring机制创建的DPDK ring接口。dpdkr接口挂接到用户态的数据通道上，与使用了IVSHMEM的虚拟机合作可以通过零拷贝技术实现高速通信。

DPDK加速的OVS数据流转发的大致流程如下：

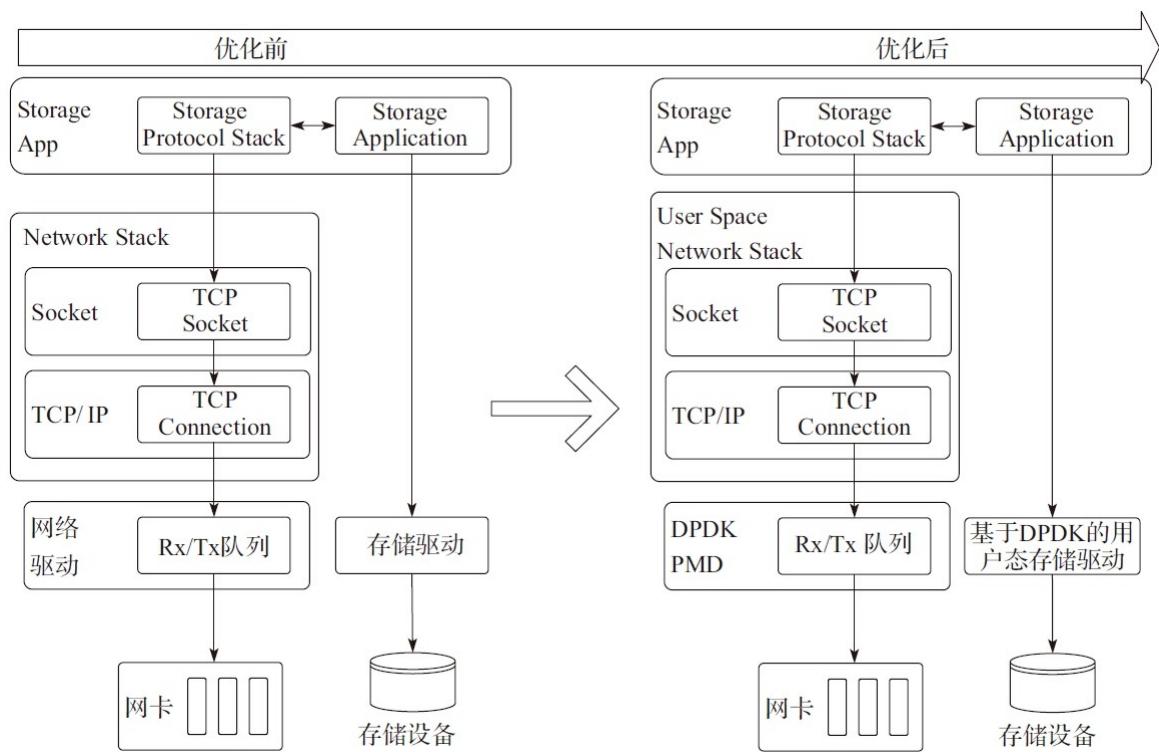
1) OVS的ovs-vswitchd接收到从OVS连接的某个网络端口发来的数据包，从数据包中提取源/目的IP、源/目的MAC、端口等信息。2) OVS在用户态查看精确流表和模糊流表，如果命中，则直接转发。3) 如果还不命中，在SDN控制器接入的情况下，经过OpenFlow协议，通告给控制器，由控制器处理。4) 控制器下发新的流表，该数据包重新发起选路，匹配；报文转发，结束。

DPDK加速的OVS与原始OVS的区别在于，从OVS连接的某个网络端口接收到的报文不需要openvswitch.ko内核态的处理，报文通过DPDK PMD驱动直接到达用户态ovs-vswitchd里。

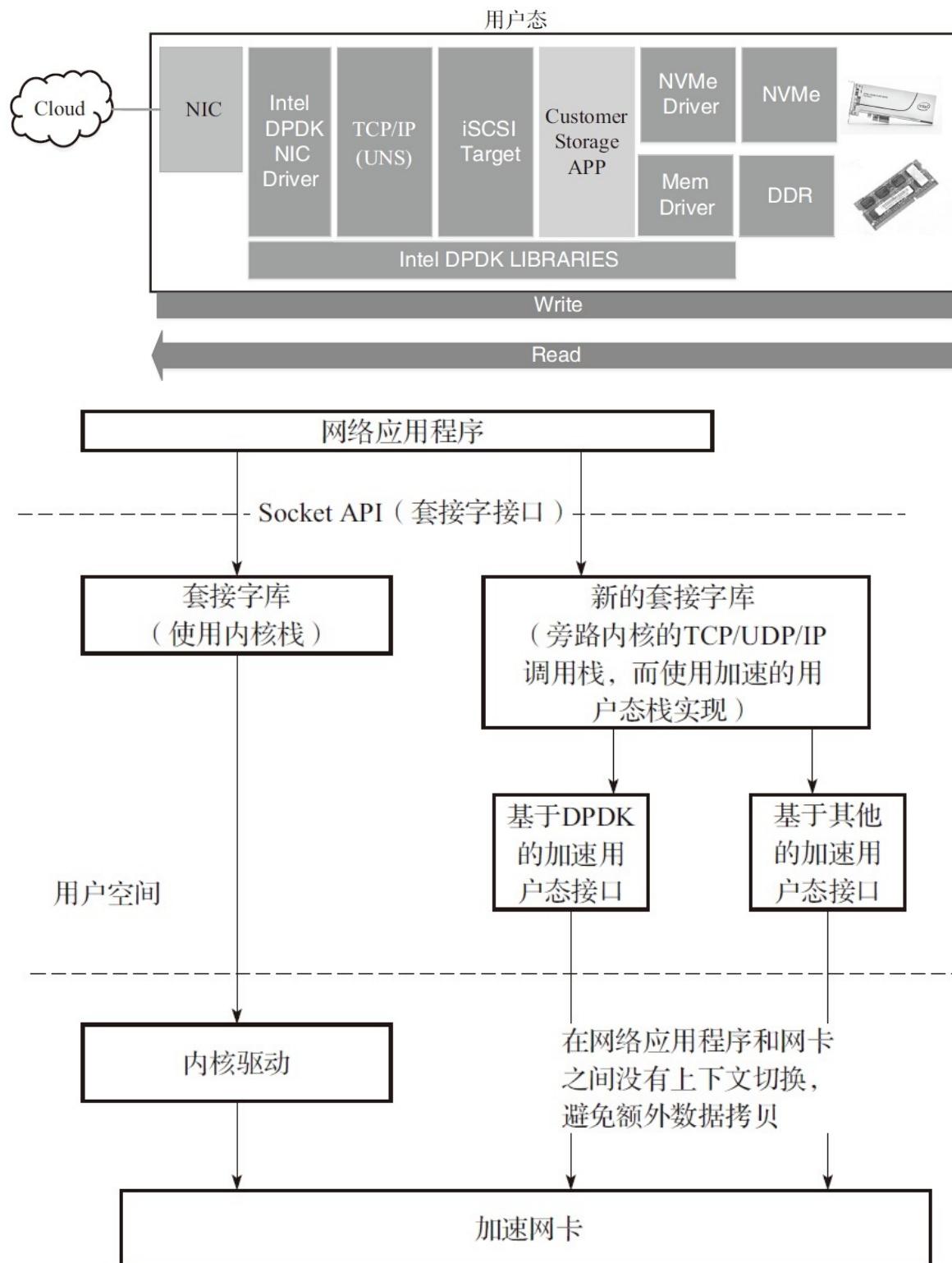
## Packet Flow



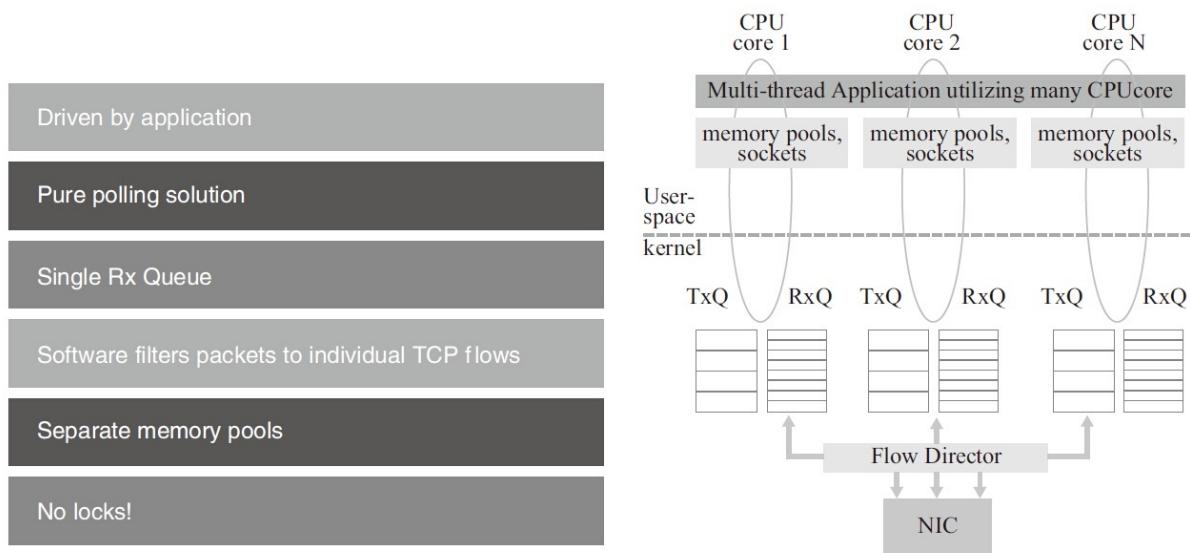
## 网络存储优化



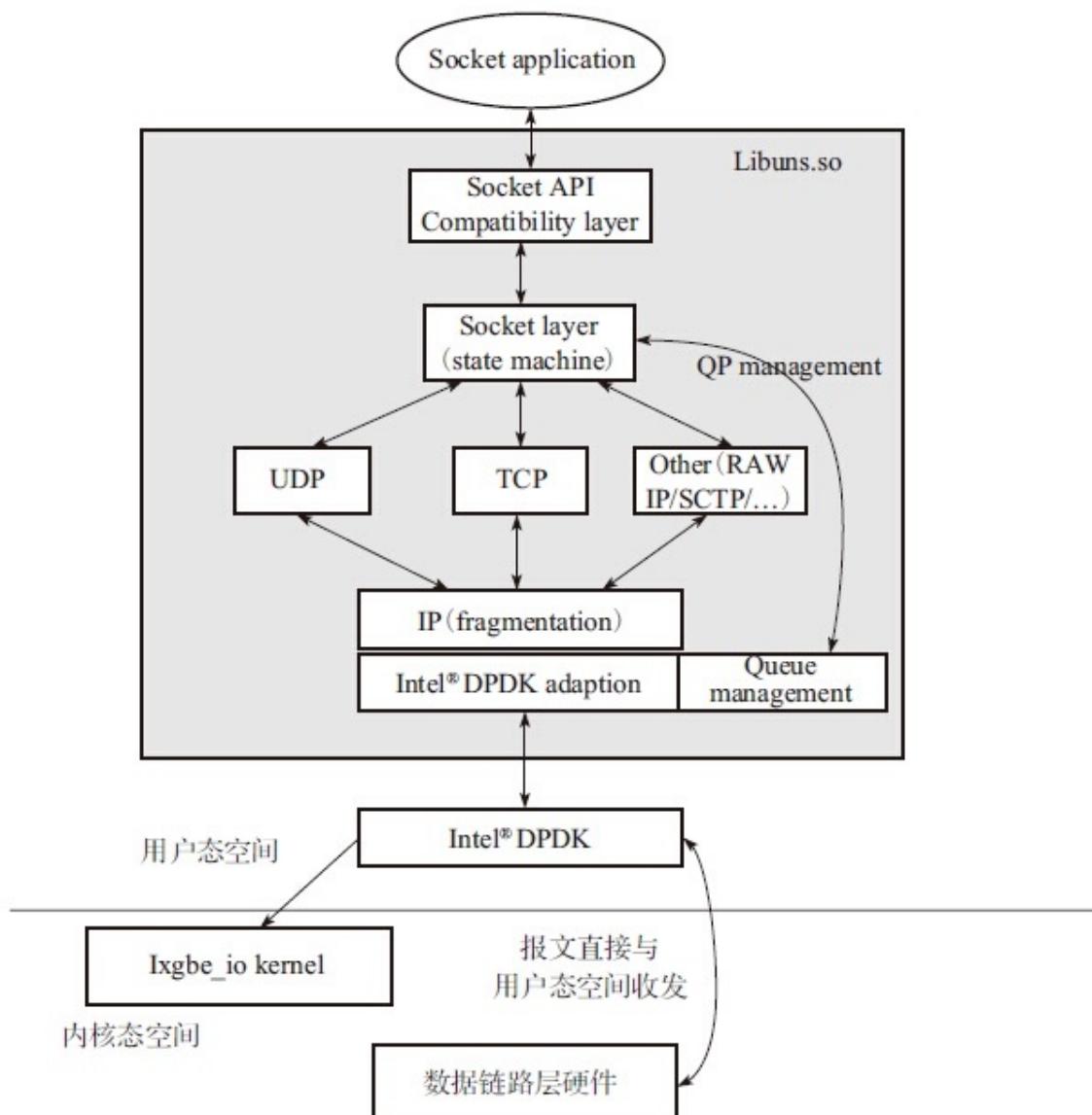
# SPDK

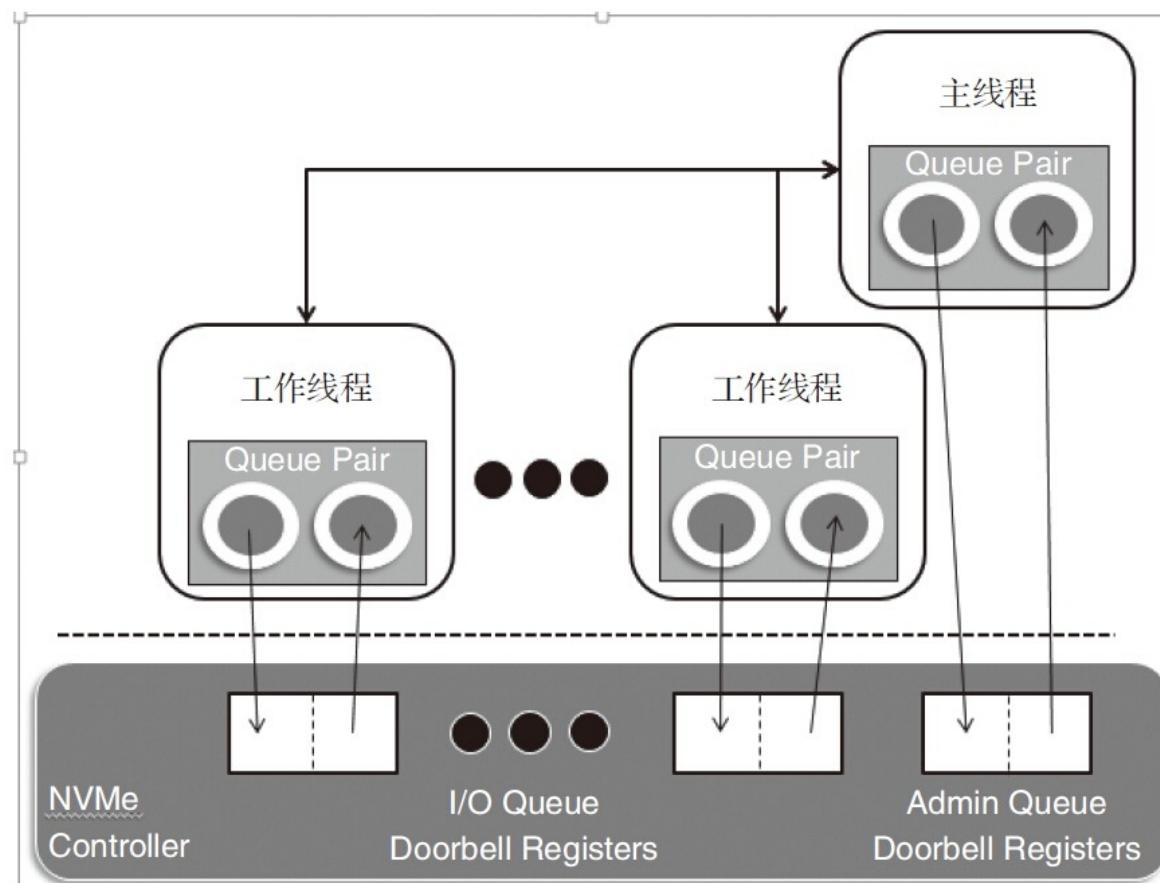
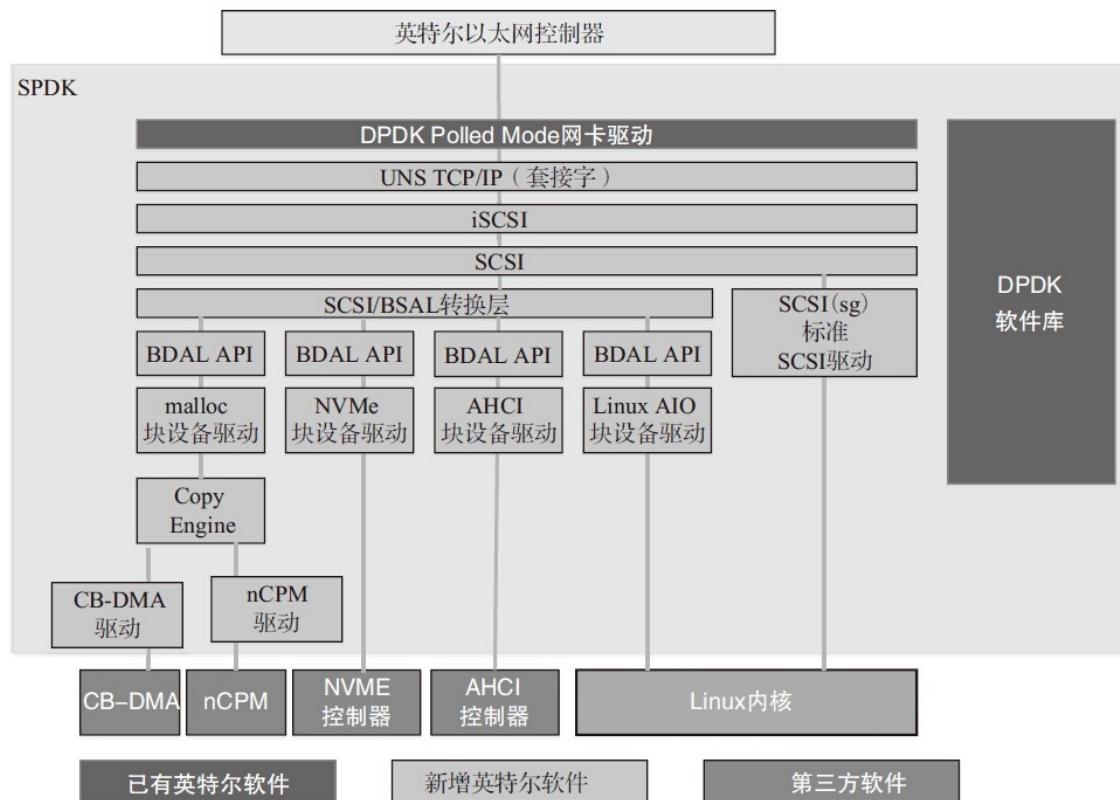


Libuns 系统架构



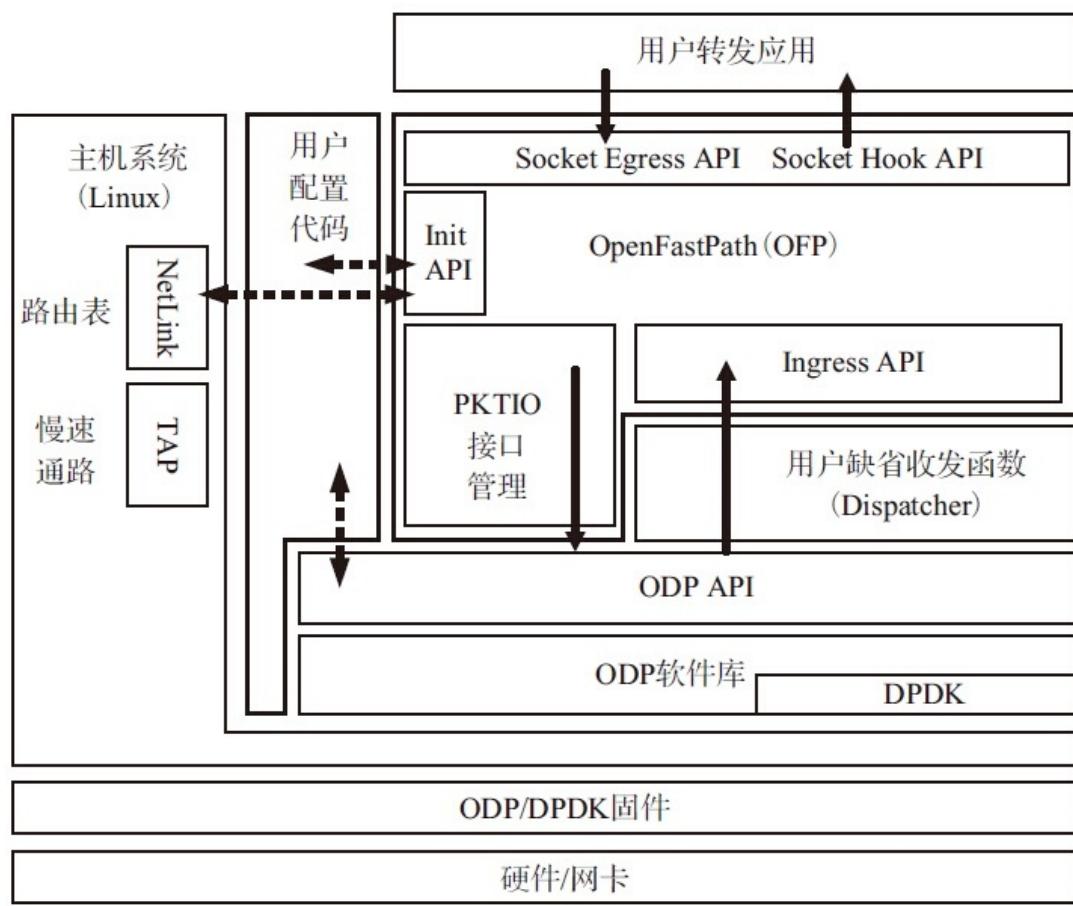
### Libuns的TCP/UDP/IP协议栈



**iSCSI target**



# OpenFastPath (OFP)



→ 数据流

→ 控制流

## SDN网络

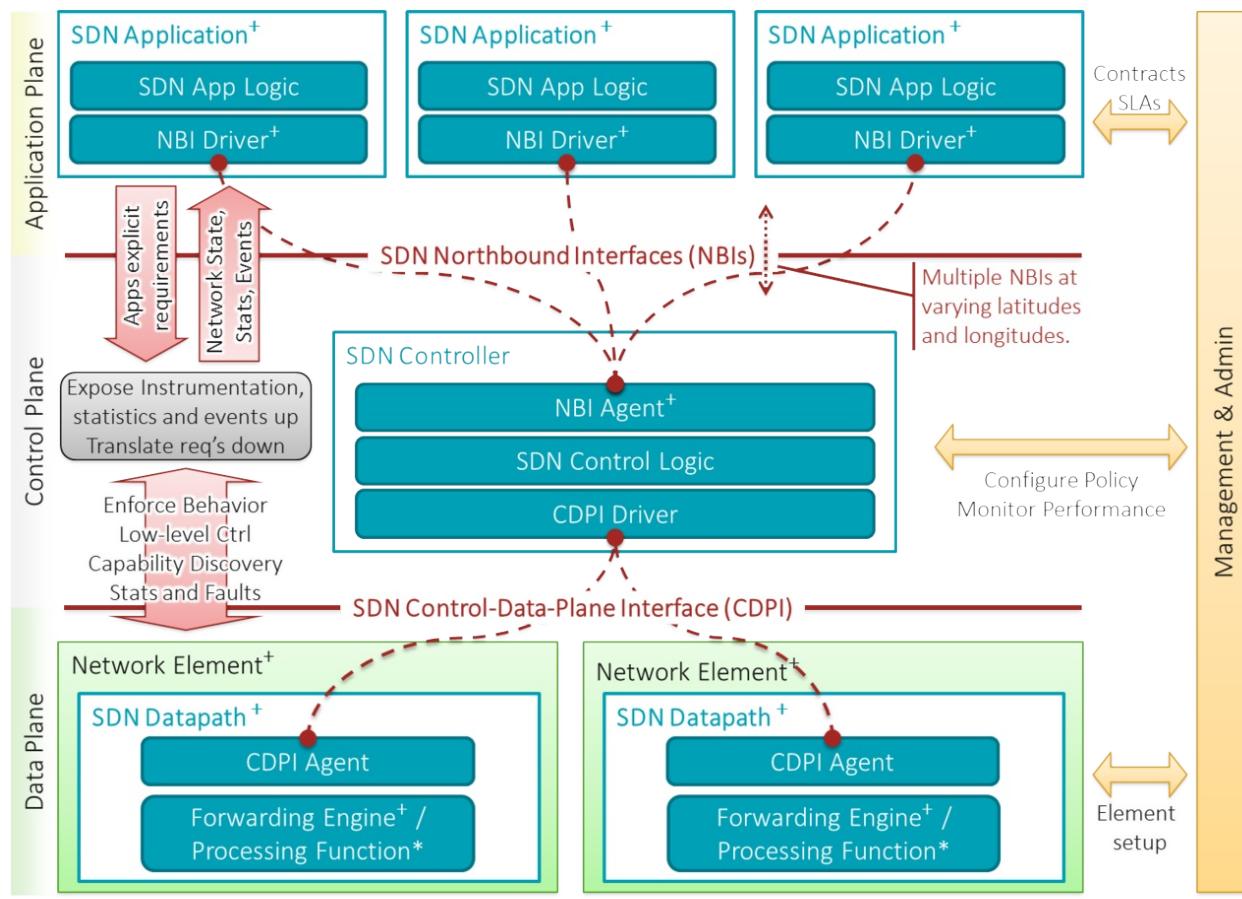
SDN（Software Defined Network）自诞生以来就非常火热，它是一种新的网络设计理念，即控制与转发分离、集中控制并且开放API。一般称控制器开放的API为北向接口，而控制器与底层网络之间的接口为南向接口。南北向接口目前都还没有统一的标准，但南向接口用的比较多的是OpenFlow，使其成为事实上的标准（曾经也有人认为SDN=OpenFlow）。

与SDN相辅相成的NFV，即网络功能虚拟化，最初主要是电信运营商在推动。NFV借助通用硬件以及虚拟化技术，实现专用网络设备的功能，从而降低网络建设的成本。对应的还有一个NV（网络虚拟化）的概念，这是在数据中心将各种不同软硬件网络资源整合为基于软件统一管理的过程，主要是用来虚拟化数据中心网络。

## SDN架构

ONF (Open Networking Foundation) 将SDN架构分为三层

- 应用层包括各种不同的业务应用
- 控制层负责数据平面资源的编排、维护网络拓扑和状态信息等
- 数据层负责数据处理、转发和状态收集



## SDN的基本特征

SDN具有三个基本特征

- 控制与转发分离：转发平面由受控转发的设备组成，转发方式以及业务逻辑由运行在分离出去的控制面上的控制应用所控制。
- 开放API：通过开放的南向和北向API，能够实现应用和网络的无缝集成，使得应用只需要关注自身逻辑，而不需要关注底层实现细节。
- 集中控制：逻辑上集中的控制平面能够获得网络资源的全局信息并根据业务需求进行全局调配和优化。

## SDN优势

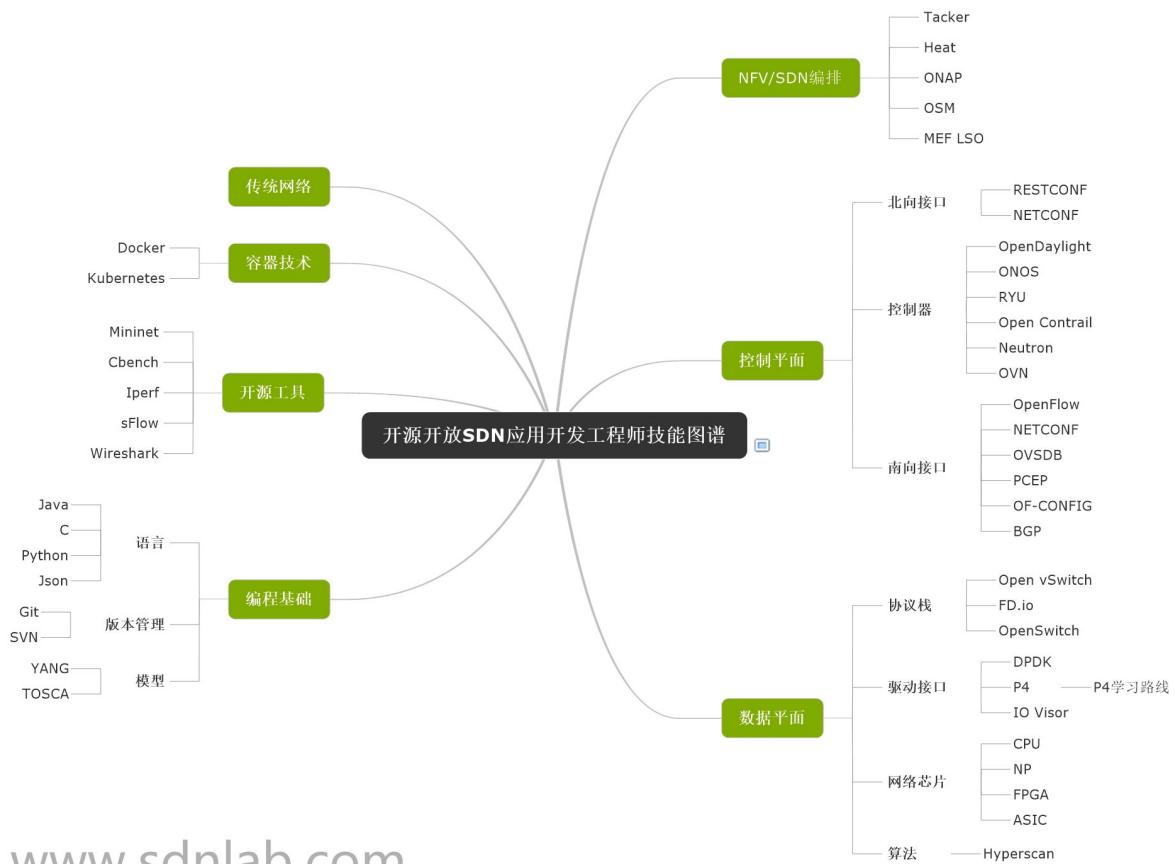
从SDN的特征出发，SDN的优势包括

- 灵活性，动态调整网络设备的配置，再也不需要人工去配置每台设备
- 网络硬件简化（如白牌交换机等），只需要关注数据的处理和转发，与业务特性解耦，加快了新业务的引入速度
- 网络的自动化部署和运维、故障诊断

## SDN发展历程

- 2006年，Martin Casado博士在RCP和4D论文基础上，提出了一个逻辑上集中控制的企业安全解决方案SANE，打开了集中控制解决安全问题的大门
- 2007年，Martin博士在SANE基础上实现了面向企业网管理的Ethane项目（SDN架构和OpenFlow的前身），Nick、Scott和Martin一起创建Nicira
- 2008年，网络领域八位学者联合发表了OpenFlow论文，Nick团队发布了第一个开源SDN控制器NOX
- 2009年，SDN入选麻省理工科技评论的“未来十大突破性技术”，Nick团队发布POX以及FlowVisor，Nicira发布Open vSwitch，James Liao和杜林共同创办了Pica8公司
- 2010年，Nick团队发布Mininet，Google发布分布式SDN控制器Onix，Nick的博士Guido等创办了BigSwitch，前思科员工JR等创办了Cumulus
- 2011年，开放网络基金会ONF诞生，第一届开放网络峰会ONS成功举办，Nick McKeown、Scott Shenker、Larry Peterson等创建了开放网络研究中心ONRC，这是ON.Lab的前身
- 2012年，Google发布B4，VMWare天价收购Nicira，BigSwitch发布Floodlight，NTT发布Ryu
- 2013年，OpenDaylight项目诞生，思科发布ACI产品方案，VMWare发布NSX产品
- 2014年，ONOS、P4等诞生。Facebook在OCP项目中开放发布Wedge交换机设计细节，白盒交换机成为这一年的主旋律。
- 2015年，ONF发布了一个开源SDN项目社区，SD-WAN成为第二个成熟的SDN应用市场。SDN与NFV融合成为趋势
- 2016年，国内的云杉网络、大河云联、盛科网络以及国外的VeloCloud、Plexxi、Cumulus 和BigSwitch都获得了新一轮融资
- 2017年，鹏博士正式发布运行国内首个运营商级SD-WAN

## SDN技能图谱



ONF整理的[SDN Reading List](#)是一个不错的SDN进阶资料库。

## 参考文档

- <https://www.opennetworking.org/index.php>
- 漫谈SDN大历史
- [SDN Reading List](#)

# SDN控制器

控制器是整个SDN网络的核心大脑，负责数据平面资源的编排、维护网络拓扑和状态信息等，并向应用层提供北向API接口。其核心技术包括

- 链路发现和拓扑管理
- 高可用和分布式状态管理
- 自动化部署以及无丢包升级

## 链路发现和拓扑管理

在SDN中通常使用LLDP发现其所控制的交换机并形成控制层面的网络拓扑。

LLDP（Link Layer Discovery Protocol，链路层发现协议）定义在802.1ab中，它是一个二层协议，提供了一种标准的链路层发现方式。LLDP协议使得接入网络的一台设备可以将其主要的能力，管理地址、设备标识、接口标识等信息发送给接入同一个局域网络的其它设备。当一个设备从网络中接收到其它设备的这些信息时，它就将这些信息以MIB的形式存储起来。这些MIB信息可用于发现设备的物理拓扑结构以及管理配置信息。需要注意的是LLDP仅仅被设计用于进行信息通告，它被用于通告一个设备的信息并可以获得其它设备的信息，进而得到相关的MIB信息。它不是一个配置、控制协议，无法通过该协议对远端设备进行配置，它只是提供了关于网络拓扑以及管理配置的信息。

发现过程为

- 控制器通过 `Packet_out` 包向所有与之相连的交换机发送LLDP包，同时要求交换机发出广播包（为了发现非OpenFlow交换机）
- 这些交换机收到消息后会向其所有端口发送LLDP数据包
- OpenFlow交换机收到LLDP数据包后通过 `Packet_in` 消息将链路信息发送给控制器
- 控制器根据收到的 `Packet_in` 消息创建网络拓扑

## 下发流表

SDN控制器通过南向接口（如OpenFlow）向SDN交换机下发流表，有两种方式

- 主动下发：控制器在数据包到达OpenFlow交换机之前就已经下发流表。这种方式不存在控制器的瓶颈问题，是主流的设计
- 被动下发：OpenFlow交换机收到第一个数据包并且没有发现与之匹配的流表项时发送给控制器处理，这种方式增加了流表设置的时间，也增加了控制器的处理负担，使控制器容易成为瓶颈

## 北向接口

北向接口是直接面向业务应用服务的，其设计密切联系业务应用需求，具有多样化的特征。北向接口目前还没有统一的标准，各家的实现均不相同，但共同点是都提供了一个开放的API，方便业务应用编程。

## 高可用

为了保证逻辑集中控制器的高可用性，需要多台控制器形成分布式集群，避免单点故障。由于控制器掌握着全网的网络设备，通过分布式协作的方式确保网络状态的一致性尤为重要。

## 开源控制器

- OpenDaylight
- ONOS
- NOX/POX
- OpenContrail
- Ryu
- Floodlight

## 参考文档

- 《SDN核心技术剖析和实战指南》
- 《重构网络-SDN架构与实现》

# OpenDaylight

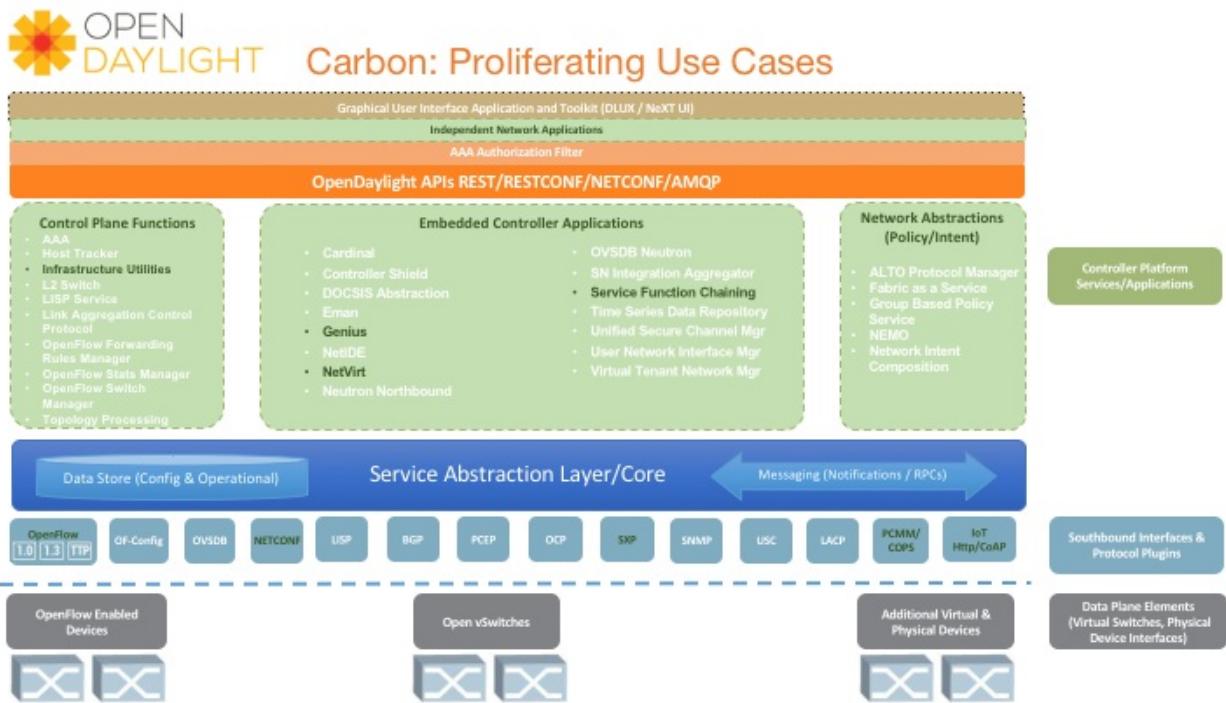
[OpenDaylight](#)是Linux基金会管理的开源SDN控制器，依托强大的社区支持和功能特性，已成为最受瞩目的开源SDN控制器。

OpenDaylight(ODL)高度模块化、可扩展、可升级、支持多协议。北向接口可扩展性强，REST型API用于松耦合应用，OSGI型用于紧耦合应用。引入SAL屏蔽不同协议的差异性。南向支持多种协议插件，如OpenFlow 1.0、OpenFlow 1.3、OVSDB、NETCONF、LISP、BGP、PCEP和SNMP等。底层支持传统交换机、纯Openflow交换机、混合模式的交换机。ODL控制平台采用了OSGI框架，实现了模块化和可扩展化，为OSGI模块和服务提供了版本和周期管理。ODL靠社区的力量驱动发，支持工业级最广的SDN和NFV使用用例。ODL每6个月推出一个版本，经历的版本为Hydrogen、Helium、Lithium、Beryllium、Boron、Carbon等。

## 架构

如下图所示，OpenDaylight架构分为南向接口层、控制平面层、北向接口层和网络应用层。

- 南向接口层：包含多种协议插件，如OpenFlow 1.0、OpenFlow 1.3、OVSDB、NETCONF、LISP、BGP、PCEP和SNMP等。南向接口层使用Netty来管理底层的并发IO
- 控制平面层：包含MD-SAL、网络功能、网络服务和网络抽象等模块，其中MD-SAL（Model Driven Service Abstraction Layer）是OpenDaylight的核心，所有模块都需要向其注册才可使用。MD-SAL也是整个控制器的管理中心，负责数据存储、请求路由、消息的订阅和发布等
- 北向接口层：包含开放API接口（包括REST API和OSGI）和认证模块
- 网络应用层：包含各种基于OpenDaylight北向接口层的各种应用集合，如Management GUI/CLI、VTN Coordinator、D4A Protection和OpenStack Neutron等



## 参考文档

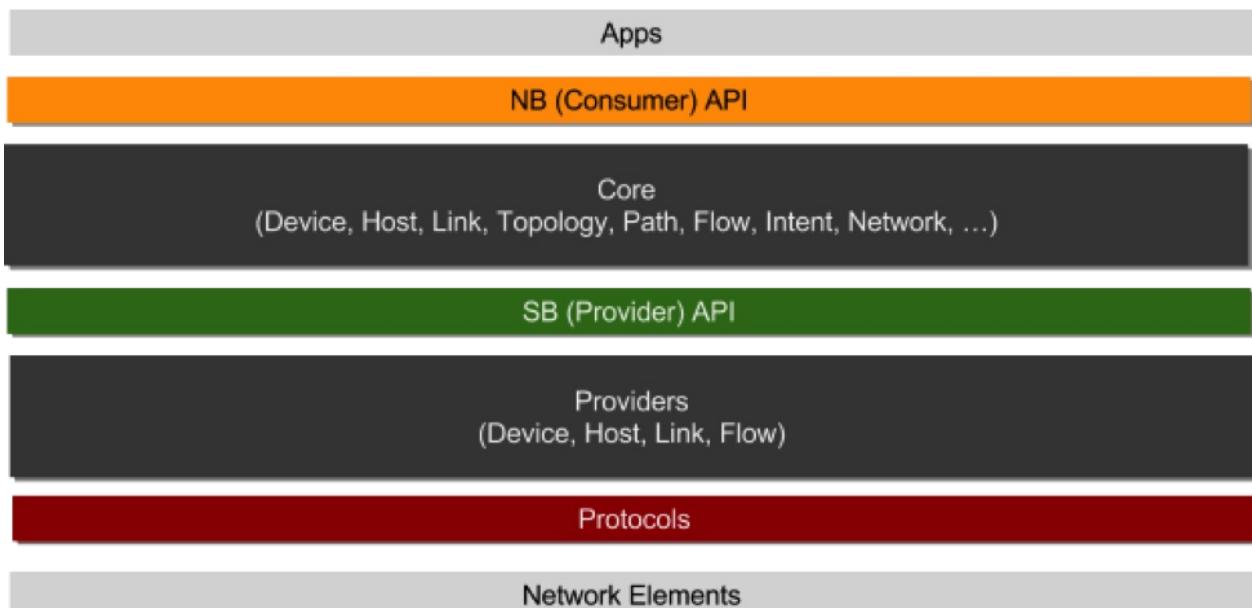
- [OpenDaylight官网](#)
- [OpenDaylight Handbook](#)
- [OpenDaylight Wiki](#)
- [OpenDaylight控制器架构分析](#)
- 《重构网络-SDN架构与实现》

# ONOS

[ONOS](#)是一个开源SDN网络操作系统，主要面向服务提供商和企业骨干网。ONOS的设计宗旨是满足网络需求实现可靠性强，性能好，灵活度高等特性。此外，ONOS的北向接口抽象层和API使得应用开发变得更加简单，而通过南向接口抽象层和接口则可以管控OpenFlow或者传统设备。ONOS集聚了知名的服务提供商（AT&T、NTT通信），高标准的网络供应商（Ciena、Ericsson、Fujitsu、Huawei、Intel、NEC），网络运营商（Internet2、CNIT、CREATE-NET），以及其他合作伙伴（SRI、Infoblox），并且获得ONF的鼎力支持，通过一些真实用例来验证其系统架构。

## ONOS架构

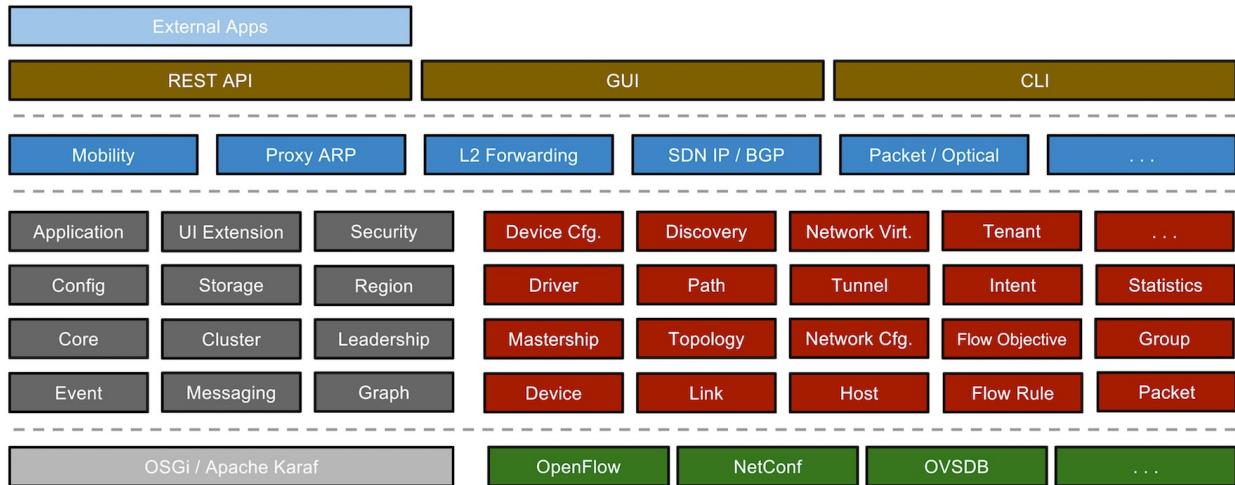
### 系统层次



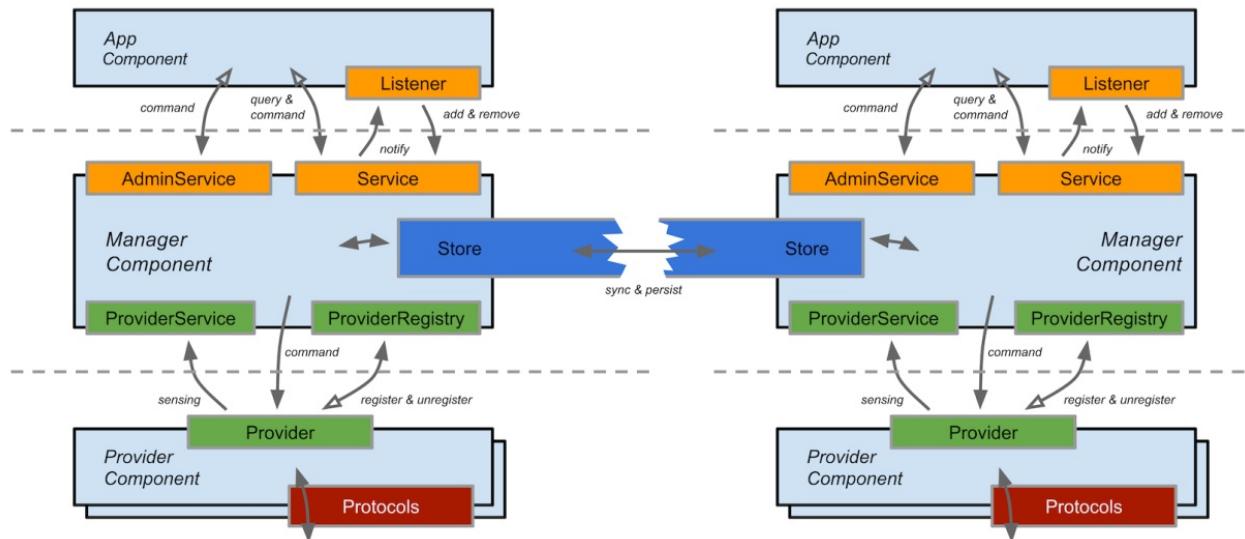
### 组件和服务

- **模块化**：ONOS由一系列功能模块组成，每个功能模块由一个或者多个组件组成，对外提供一种特定服务，这种基于SOA的框架同时支持对组件的全生命周期管理，支持动态加载、卸载组件。
- **开放**：ONOS提供开放的北向与南向API，使得用户能够很方便的基于ONOS开发应用以及南向插件。
- **抽象**：ONOS 抽象出了统一的网络资源和网元模型奠定了第三方SDN应用程序互通的基础，使得运营商可以做灵活的业务协同和低成本业务创新。
- **简单**：ONOS屏蔽了复杂的分布式等通用机制，对外只暴露业务接口，使得应用开发十

分简单。



## ONOS 集群



- **分布式**: 由多个实例组成一个集群
- **对称性**: 每一个实例运行相同的软件和配置
- **容错与弹性扩展**: 集群在面对节点故障时仍然可操作，支持新节点动态加入，轻松应对网络扩张
- **位置透明**: 一个客户端可以和任何实例打交道，集群要展现单个逻辑实例的抽象
- **ONOS 集群间通信**: 分为两种，一种基于 Gossip 协议，是数据弱一致性的通信方式；一种基于 Raft 算法，是保证数据强一致性的通信方式
- **高可靠**: ONOS 的 Cluster 机制能够保障节点失效对业务无影响，当 ONOS 节点宕机时，其他节点会接管该节点对网元的控制权，当节点恢复后，通过 loadbalance 命令恢复节点对网元的控制并使整体的控制达到负载均衡

## 参考文档

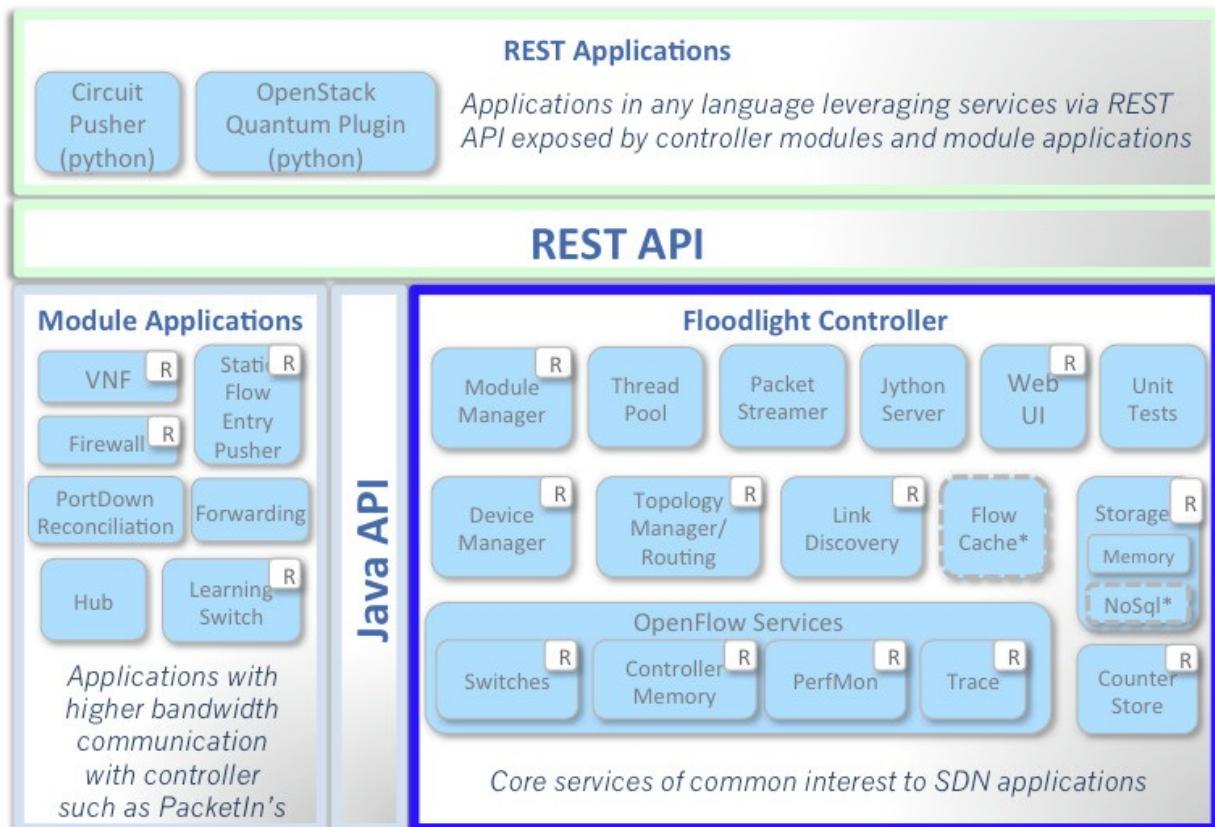
- ONOS Website
- ONOS Wiki
- ONOS架构分析
- ONOS白皮书上篇之ONOS简介
- ONOS白皮书中篇之ONOS架构

# Floodlight

Floodlight是BigSwitch在Beacon基础上开发的SDN控制器，它基于Java开发，具有良好的架构和性能，也是早期最流行的SDN控制器之一。

Floodlight的架构可以分为控制层和应用层，应用层通过北向API与控制层通信；控制层则通过南向接口控制数据平面。

Floodlight模块结果如下所示：



由于Floodlight更新迭代速度较慢，特别是OpenDaylight诞生以后，Floodlight已经丧失了

## 参考文档

- [Floodlight官网](#)
- [Project Floodlight](#)

# Ryu

Ryu是日本NTT公司推出的SDN控制器框架，它基于Python开发，模块清晰，可扩展性好，逐步取代了早期的NOX和POX。

- Ryu支持OpenFlow 1.0到1.5版本，也支持Netconf，OF-CONFIG等其他南向协议
- Ryu可以作为OpenStack的插件，见[Dragonflow](#)
- Ryu提供了丰富的组件，便于开发者构建SDN应用

## 示例

Ryu的安装非常简单，直接用pip就可以安装

```
pip install ryu
```

安装完成后，就可以用python来开发SDN应用了。比如下面的例子构建了一个L2Switch应用：

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_0

class L2Switch(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(L2Switch, self).__init__(*args, **kwargs)

        @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
        def packet_in_handler(self, ev):
            msg = ev.msg
            dp = msg.datapath
            ofp = dp.ofproto
            ofp_parser = dp.ofproto_parser

            actions = [ofp_parser.OFPActionOutput(ofp.OFPP_FLOOD)]
            out = ofp_parser.OFPPacketOut(
                datapath=dp, buffer_id=msg.buffer_id, in_port=msg.in_port,
                actions=actions)
            dp.send_msg(out)
```

最后可以使用 `ryu-manager` 启动应用：

```
ryu-manager L2Switch.py
```

## 参考文档

- [Ryu官网](#)
- [Ryu源码](#)
- [Ryu Book](#)
- [RYU 控制器性能测试报告](#)

# NOX/POX

[NOX](#)是第一个SDN控制器，由Nicira开发，并于2008年开源发布。NOX在2010年以前得到广泛应用，不过由于其基于C++开发，开发成本较高，逐渐在控制器竞争中没落。所以后来其兄弟版本[POX](#)面世。POX是完全基于Python的，适合SDN初学者。但POX也有其架构和性能的缺陷，逐渐也被新兴的控制器所取代。

目前，NOX/POX社区已不再活跃，其官网 <http://www.noxrepo.org> 也已废弃，不推荐在生产中继续使用它们。

## 参考文档

- [NOX源码](#)
- [POX Wiki](#)

# 南向接口

南向接口负责控制器与数据平面的通信，可以理解成数据平面的编程接口，直接决定了SDN架构的可编程能力。

主流的南向接口协议包括

- **OpenFlow**：第一个开放的南向接口协议，也是目前最流行的协议。它提出了控制与转发分离的架构，规定了SDN转发设备的基本组件和功能要求，以及与控制器通信的协议。
- **OF-Config**：提供开放接口用于控制和配置OpenFlow交换机，但不影响流表的内容和数据转发行为。OF-CONFIG在OpenFlow架构上增加了一个被称作OpenFlow Configuration Point的配置节点。这个节点既可以是控制器上的一个软件进程，也可以是传统的网管设备。OF-Config基于NET-CONF与设备通信。
- **P4**：协议无关的数据包处理编程语言，提供了比OpenFlow更出色的编程能力。它不仅可以指导数据流进行转发，还可以对交换机等转发设备的数据处理流程进行软件编程定义。
- **OVSDB**：Open vSwitch数据库协议。
- **NET-CONF**：用于替代CLI、SNMP等配置交换机。
- **OpFlex**：思科ACI使用的一种声明式南向接口协议。

## 参考文档

- <https://www.opennetworking.org/sdn-resources/openflow>
- <http://p4.org/>
- [P4:真正的SDN还遥远吗](#)

# OpenFlow

OpenFlow是第一个开放的南向接口协议，也是目前最流行的南向协议。它提出了控制与转发分离的架构，规定了SDN转发设备的基本组件和功能要求，以及与控制器通信的协议。

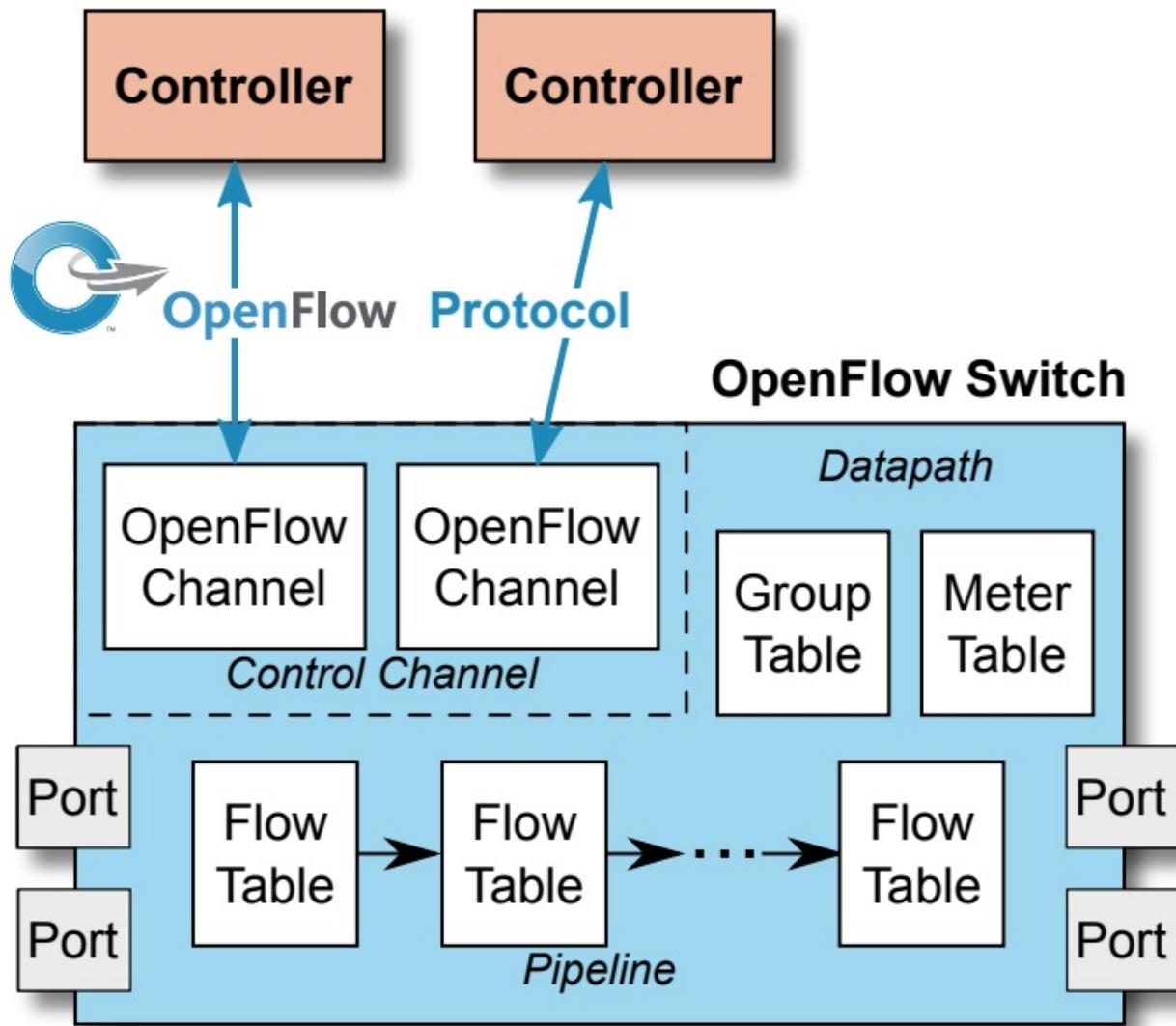
OpenFlow起源于Nick McKeown等在2008年发表的[OpenFlow: enabling innovation in campus networks](#)论文，并在次年发布了1.0版本协议。2011年又成立了Open Networking Foundation (ONF)进一步规范和推动OpenFlow的发展，并将OpenFlow的协议规范发布在[ONF网站](#)。

## OpenFlow原理

OpenFlow协议规范定义了OpenFlow交换机、流表、OpenFlow通道以及OpenFlow交换协议。

### OpenFlow交换机

一个典型的OpenFlow交换机如下图所示

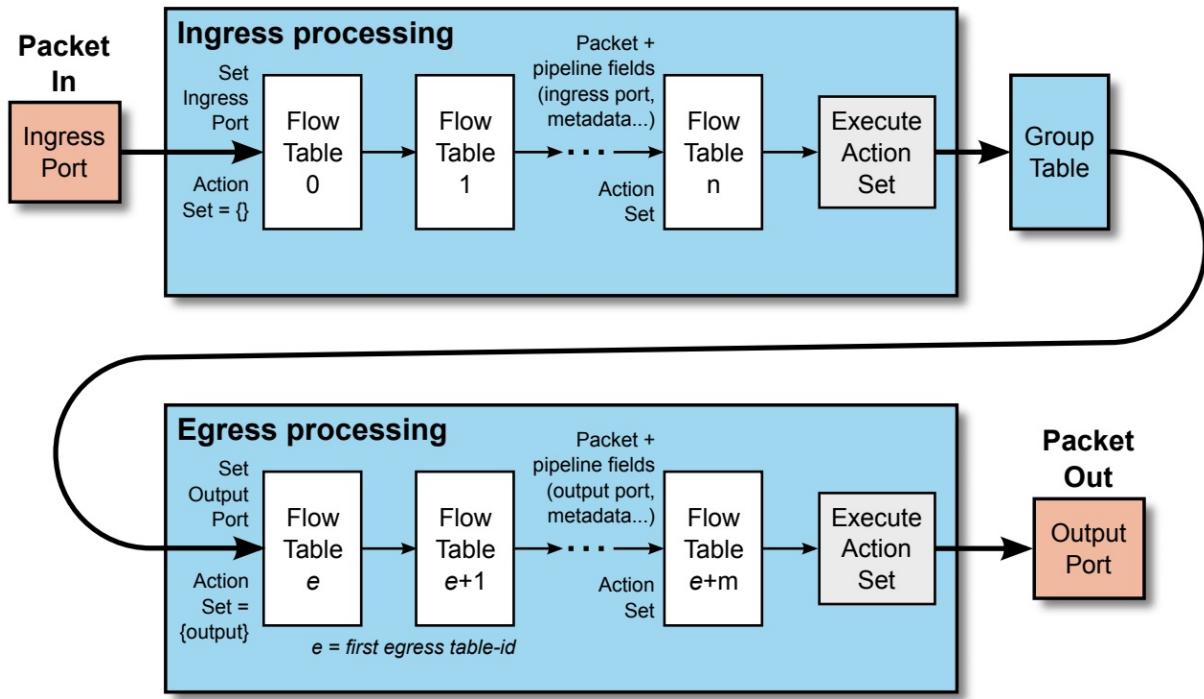


它主要由OpenFlow通道和数据平面组成，而数据平面又包括流表、端口、组表和Meter表等：

- OpenFlow通道用于交换机和控制器进行通信（基于OpenFlow交换协议）
- 流表即存放流表项的表
- 端口是OpenFlow与其他网络协议栈进行数据交换的网络接口，包括物理端口、逻辑端口以及预留端口等
- 组表用于定义一组可被多个流表项共同使用的动作
- Meter表用于计量和限速

## 流表

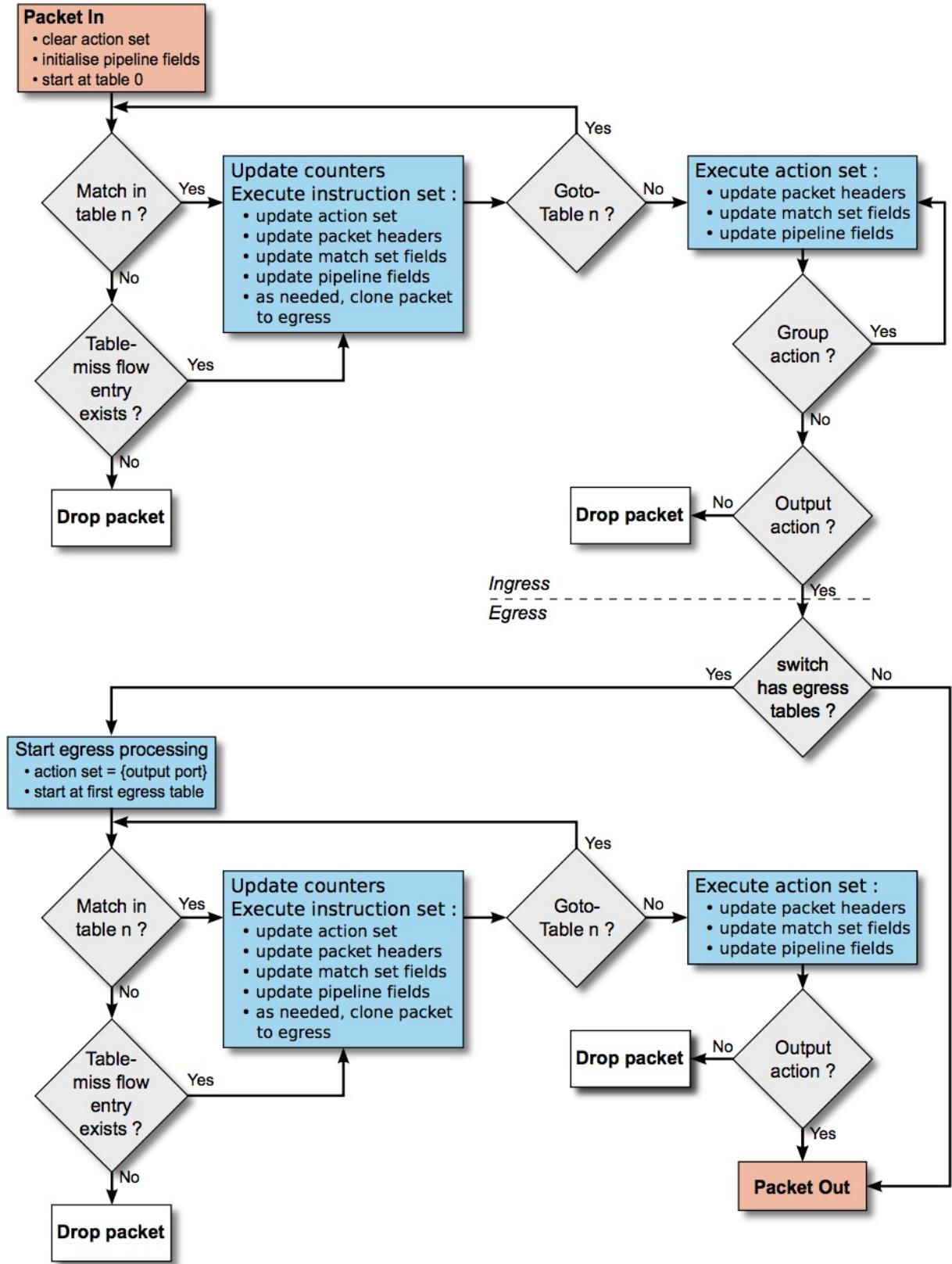
流表用于存储流表项，多级流表以流水线的方式处理。



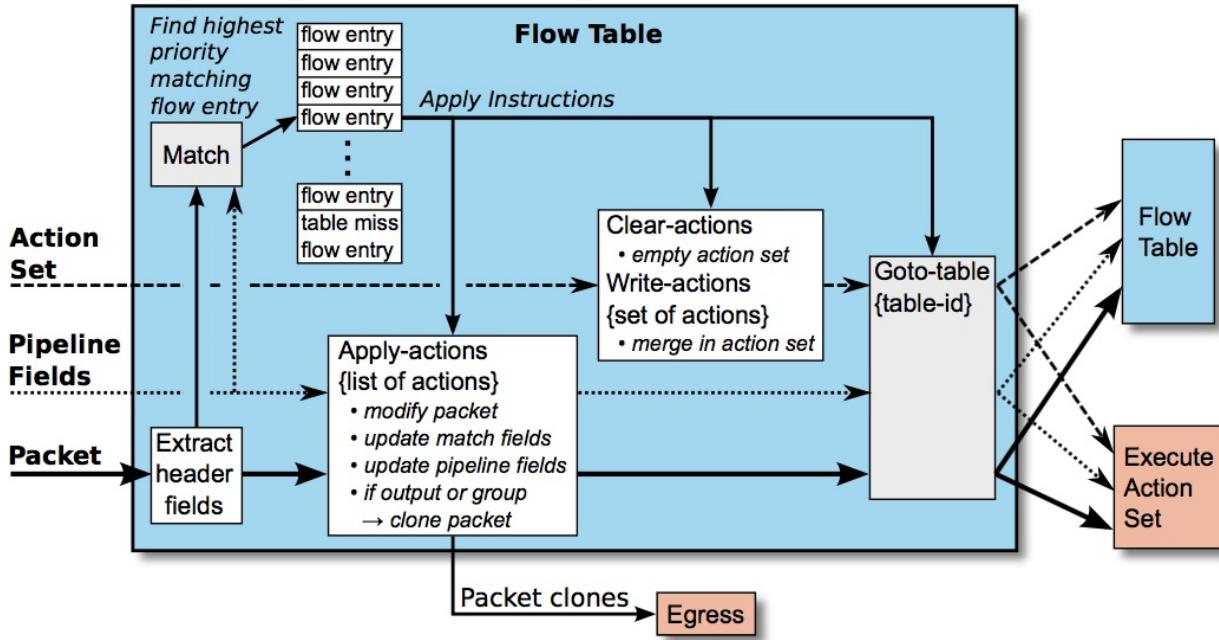
每个流表项由匹配域（包括输入端口、包头以及其他流表设置的元数据）、优先级、指令集、计数器、计时器、Cookie和用于管理流表项的flag组成：

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags
--------------	----------	----------	--------------	----------	--------	-------

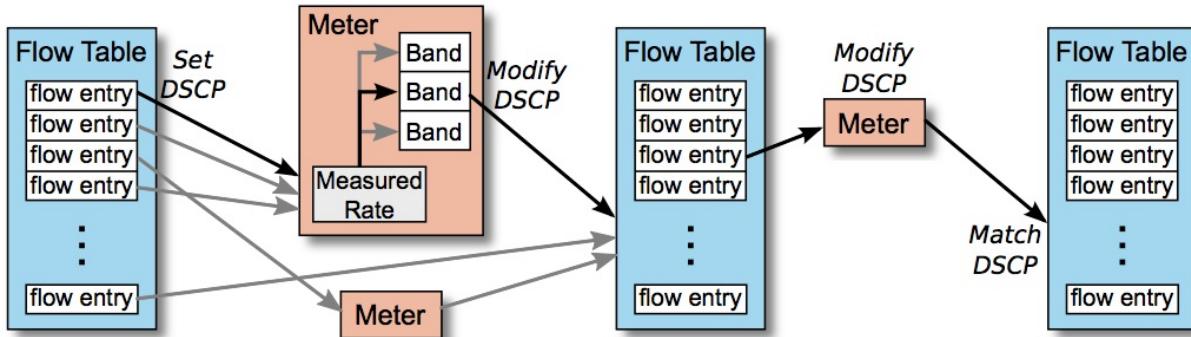
一个典型的流表匹配过程如下所示



而典型的指令执行过程如下所示



除了流表，还可以定义Meter表



## OpenFlow通道

OpenFlow通道是控制器和交换机通信的通道。控制器可以通过该通道来配置和管理交换机、接收交换机发出的事件等。OpenFlow通道使用OpenFlow交换协议（OpenFlow switch protocol），通常基于TLS通信，但也支持直接TCP通信。

OpenFlow交换协议支持三种类型的报文

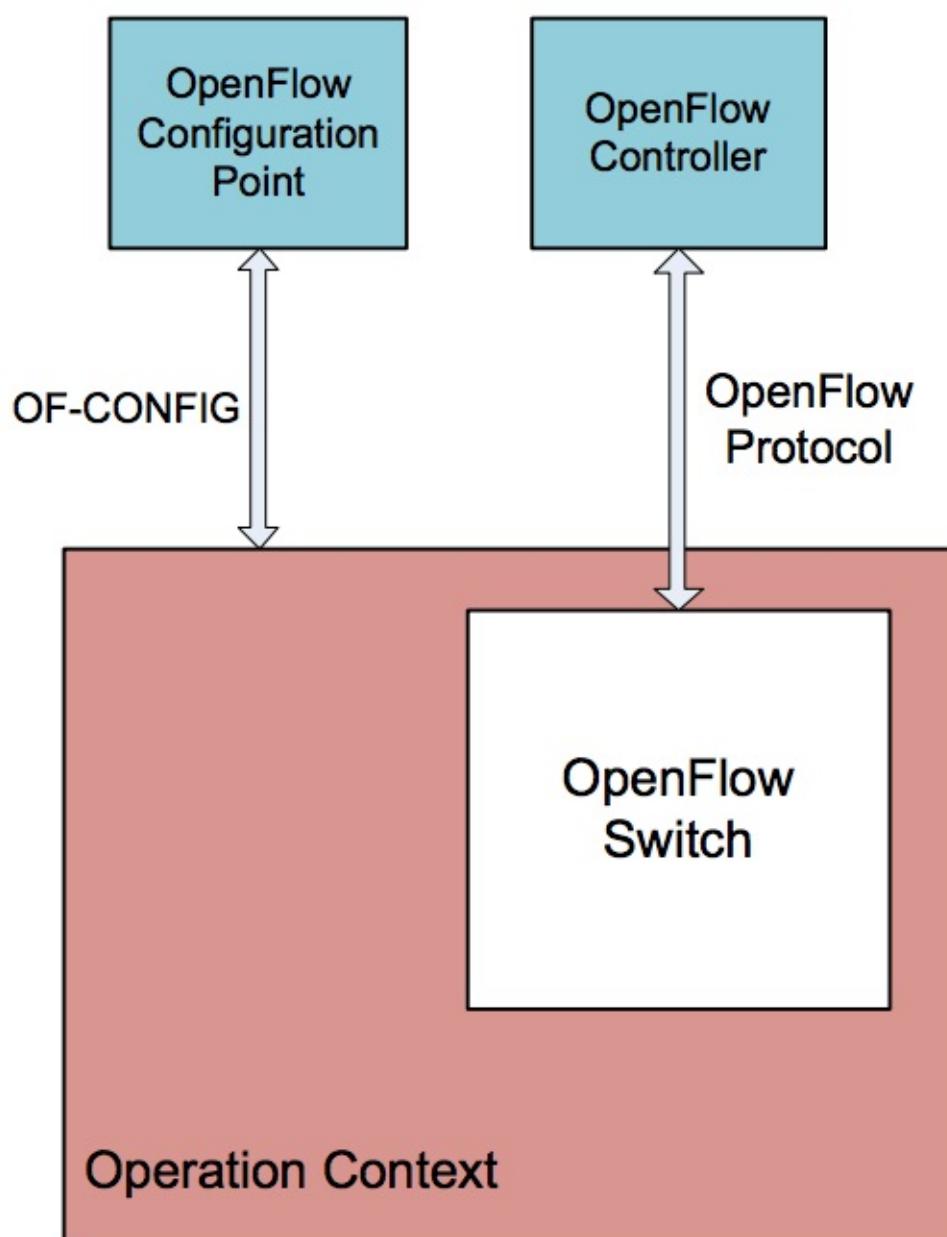
- controller-to-switch：控制器初始化并下发给交换机的报文，用于管理和查询交换机状态（如查询交换机特性，修改交换机流表、组表等）
- asynchronous：交换机异步发送给控制器的报文，用于更新网络事件和交换机状态的改变（如新报文到达、交换机端口变化等）
- symmetric：交换机或控制器发送，但无需对方许可，如Hello协商、Echo活性测试、Error错误报文等

## 参考文档

- [OpenFlow官方网站](#)
- [OpenFlow协议规范](#)

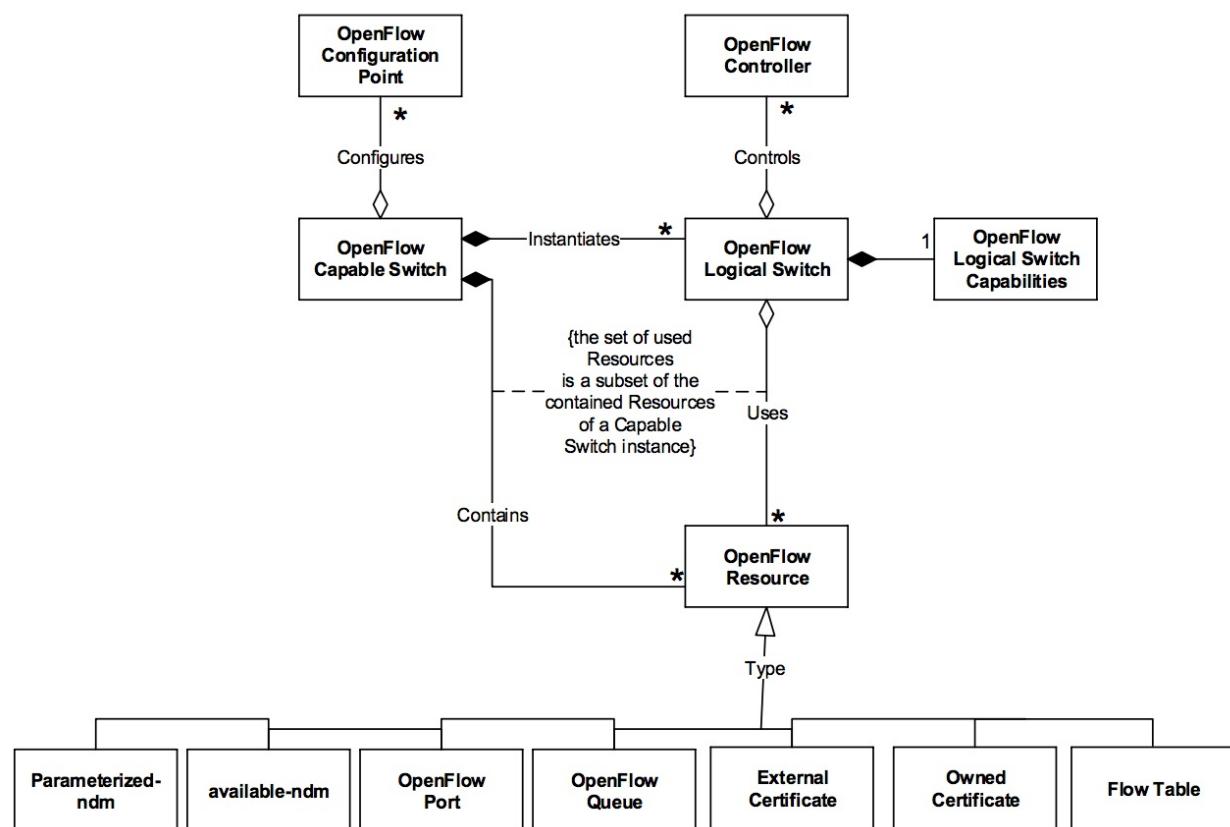
# OF-Config

OF-Config是一个OpenFlow交换机配置协议，也是ONF主推的OpenFlow补充协议，很好的填补了OpenFlow协议之外的交换机运维配置等内容。它提供了开放接口用于控制和配置OpenFlow交换机，但不影响流表的内容和数据转发行为。OF-CONFIG在OpenFlow架构上增加了一个被称作OpenFlow Configuration Point的配置节点。这个节点既可以是控制器上的一个软件进程，也可以是传统的网管设备。



OF-Config的协议规范也发布在[ONF官方网站](#)。

OF-Config基于NET-CONF与设备通信，其核心数据结构如下所示。



# NETCONF

NETCONF是一个基于XML的交换机配置接口，用于替代CLI、SNMP等配置交换机。

## 协议

NETCONF通过RPC与交换机通信，其协议包含四层



- (1) 安全传输层，用于跟交换机安全通信，NETCONF并未规定具体使用哪种传输层协议，所以可以使用SSH、TLS、HTTP等各种协议
- (2) 消息层，提供一种传输无关的消息封装格式，用于RPC通信
- (3) 操作层，定义了一系列的RPC调用方法，并可以通过Capabilities来扩展
- (4) 内容层，定义RPC调用的数据内容

## 参考文档

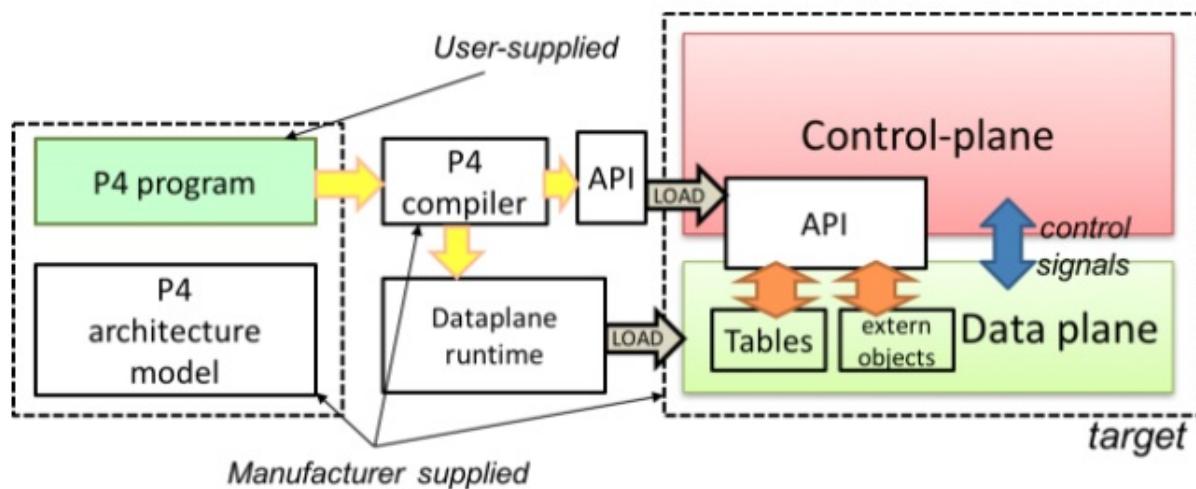
- [RFC6241](#)



# P4

P4是一个协议无关的数据包处理编程语言，提供了比OpenFlow更出色的编程能力。它不仅可以指导数据流进行转发，还可以对交换机等转发设备的数据处理流程进行编程。主要特点包括

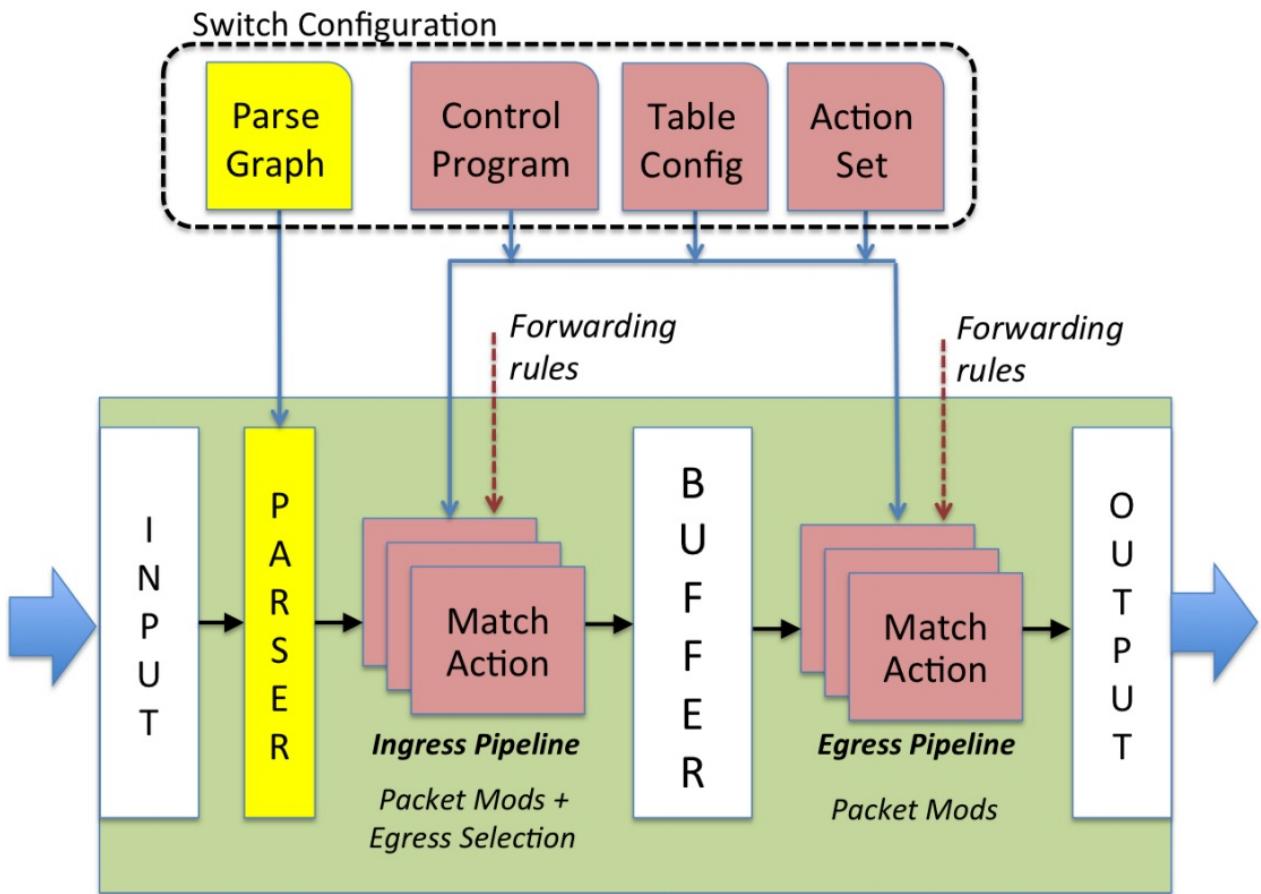
- 灵活定义数据转发流程，支持重新配置匹配域，并支持无中断的重配置
- 协议无关，只需要关注如何处理转发包，而不需要关注底层协议细节
- 设备无关，无需关注底层设备的具体信息



## 转发模型

如下图所示，数据包经过解析后，会被传递到一个“匹配-动作”表，并支持串行和并行操作。类似于OpenFlow流水线，这些表决定了数据包将被送往哪里，如丢弃或送到某个出端口。P4的流水线分为入口流水线和出口流水线：

- 入口流水线中，数据包可能会被转发、复制、丢弃或触发流量控制
- 而出口流水线可以对数据包作进一步的修改，并送到相应的出端口



P4交换机将流水线处理数据的过程进行抽象和重定义，数据处理单元对数据的处理抽象成匹配和执行匹配-动作表的过程，包头的解析抽象成P4中的解析器，数据处理流程抽象成流控制。P4基础数据处理单元是不记录数据的，所以就需要引入一个元数据总线，用来存储一条流水线处理过程中需要记录的数据。P4交换机的专用物理芯片Tofino，最高支持12个数据处理单元，可以覆盖传统交换机的所有功能。

## P4语言

每个P4程序包含如下的5个关键组件：

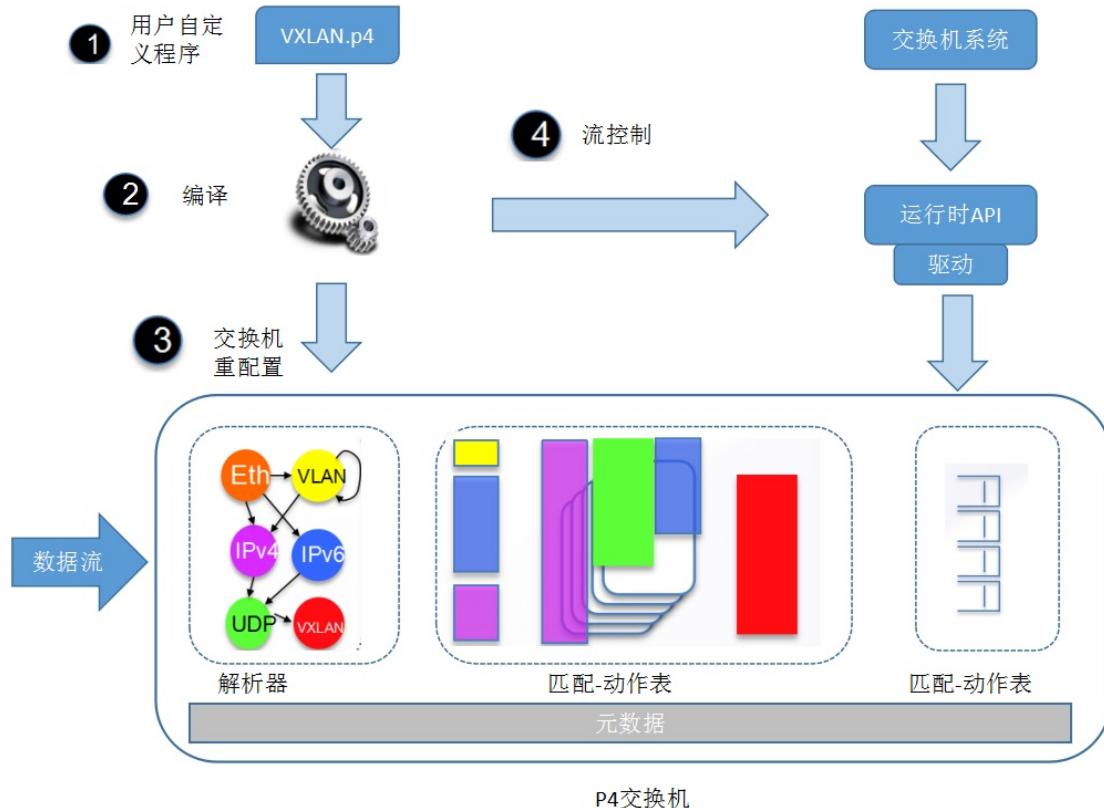
- Headers：定义报文头部格式，支持重定义的头部名称和任意长度字段
- Parses：定义数据包解析流程的有限状态机
- Tables：“匹配-动作”表，定义匹配域以及对应的执行动作
- Actions：动作指令集，包括构造查找键（Construct lookup keys）、根据查找键查表、执行动作等
- Control Flow：控制程序，决定了数据包处理的流程，比如如何在不同表之间跳转等

具体的编写方法可以参考[P4 Language Specification](#)。

## P4工作流程

P4的完整工作流程为：

- 首先用户需要自定义数据帧的解析器和流控制程序
- 然后，P4程序经过编译器编译后输出JSON格式的交换机配置文件和运行时的API
- 再次配置的文件载入到交换器中后更新解析起始和匹配-动作表
- 最后交换机操作系统按照流控制程序进行包的查表操作



以新增VLAN包解析为例，图中解析器除VXLAN以外的包解析是交换机中已有的，载入VXLAN.p4文件所得的配置文件的过程就是交换机的重配置过程。配置文件载入交换机后，解析器中会新增对VXLAN包解析，同时更新匹配-动作表，匹配成功后执行的动作也是在用户自定义的程序中指定。执行动作需要交换机系统调用执行动作对应的指令来完成，这时交换机系统调用的是经过P4编译器生成的统一的运行时API，这个API就是交换机系统调用芯片功能的驱动，流控制程序就是指定API对应的交换机指令。

## 参考文档

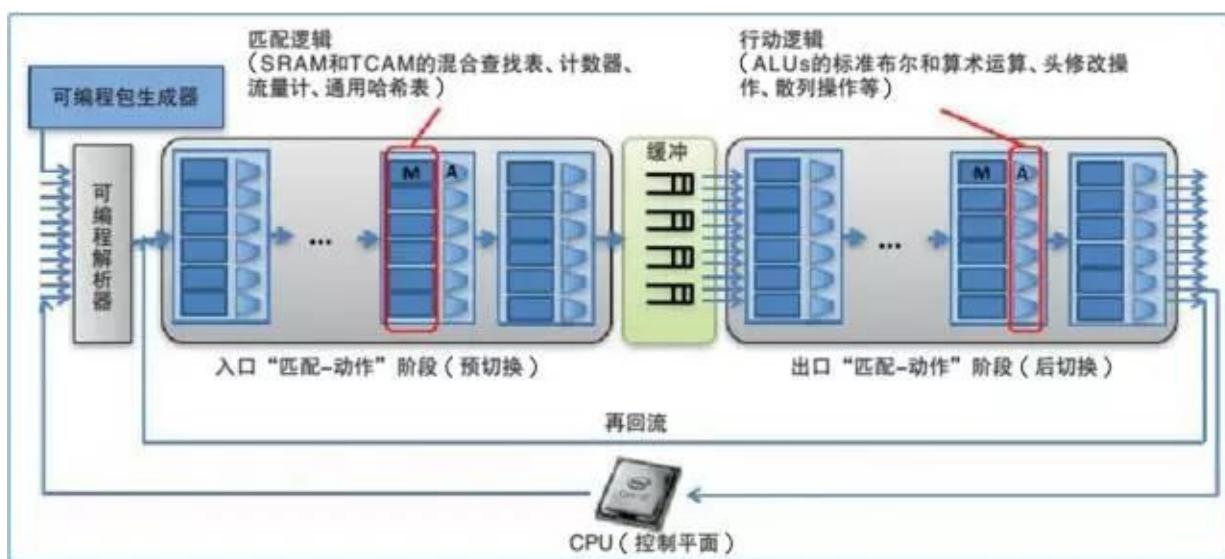
- [P4](#)
- [P4:开创数据平面可编程时代](#)

# SDN数据平面

数据平面负责数据处理、转发和状态收集等。其核心设备为交换机，可以是物理交换机，也可以是虚拟交换机。不同于传统网络转发设备，应用于SDN中的转发设备将数据平面与控制平面完全解耦，所有数据包的控制策略由远端的控制器通过南向接口协议下发，网络的配置管理同样也由控制器完成，这大大提高了网络管控的效率。交换设备只保留数据平面，专注于数据包的高速转发，降低了交换设备的复杂度。

本质上来说，决定SDN可编程能力的因素在于数据平面的可编程性，所以就有了通用可编程数据平面OpenFlow Switch。通用可编程数据平面支持用户通过软件编程的方式任意定义数据平面的能力，包括数据包的解析、处理等功能。

从OpenFlow Switch通用转发模型诞生至今，学术界和产业界在通用可编程数据平面领域做了很多努力，持续推动了SDN数据平面的发展。其中典型的通用可编程数据平面设计思路是The McKeown Group的可编程协议无关交换机架构PISA（Protocol-Independent Switch Architecture）。到达PISA系统的数据包先由可编程解析器解析，再通过入口侧一系列的Match-Action阶段，然后经由队列系统交换，由出口Match-Action阶段再次处理，最后重新组装发送到输出端口。



# SDN数据平面发展历史

- 早期的软件交换机，如Open vSwitch、Indigo等，一般存在性能问题
- 随后的可编程硬件，如基于NetFPGA的OpenFlow交换机，成本高，开发难度大，灵活性差
- 中期的OpenFlow网络设备，如Pica8、Cumulus、Big Switch等在现有网络交换机操作系统上增加了对OpenFlow南向接口的支持，还有博通、盛科等网络芯片厂商在网络交换机

芯片上实现了对OpenFlow南向接口和OpenFlow Switch通用转发模型的支持

- 现阶段通用可编程网络芯片和数据平面编程语言的出现进一步推动SDN数据平面的发展，如The McKeown Group的PISA (Protocol Independent Switch Architecture) 架构以及Barefoot发布的基于PISA的可编程网络芯片Tofino。现阶段数据平面的编程语言代表为P4。

## 参考文档

- [Barefoot Tofino](#)
- [P4 Language](#)
- [《重构网络-SDN架构与实现》](#)
- [基于SDN的数据中心网络技术研究](#)

# NFV

NFV (Network Functions Virtualization)是一种使用x86等通用硬件来承载传统网络设备功能的技术。它是通过用软件和自动化替代专用网络设备来定义、创建和管理网络的新方式。

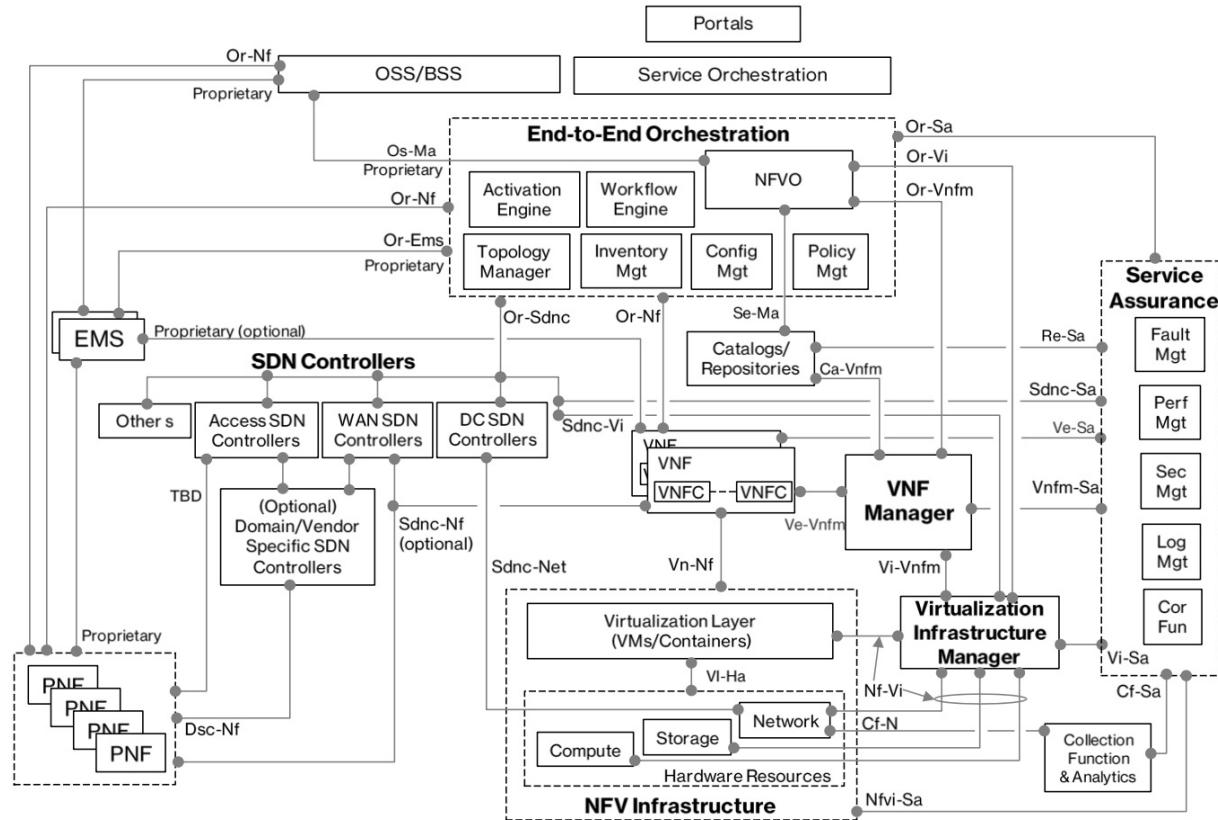
同SDN一样，NFV从根本上讲是从基于硬件的解决方案转向更开放的基于软件的解决方案。例如，取代专用防火墙设备，软件可以通过虚拟防火墙提供相同的功能。再如入侵检测和入侵防御、NAT、负载均衡、WAN加速、缓存、GGSN、会话边界控制器、DNS等等虚拟网络功能。有时，不同的子功能可以组合起来形成一个更高级的多组件VNF，如虚拟路由器。

正如SDN和NFV可以在廉价的裸机或白盒服务器上的实现方式，这些VNF可以运行在通用的商用硬件组件上，而不是成本高昂的专有设备。网络运营商通过NFV快速实现并应用VNF，并通过业务流程自动化服务交付。

ETSI列出了NFV为网络运营商及其客户提供的几点优势：

- 通过降低设备成本和降低功耗，减少运营商CAPEX和OPEX
- 缩短部署新网络服务的时间
- 提高新服务的投资回报率
- 更灵活的扩大，缩小或发展服务
- 开放虚拟家电市场和纯软件进入者
- 以较低的风险试用和部署新的创新服务

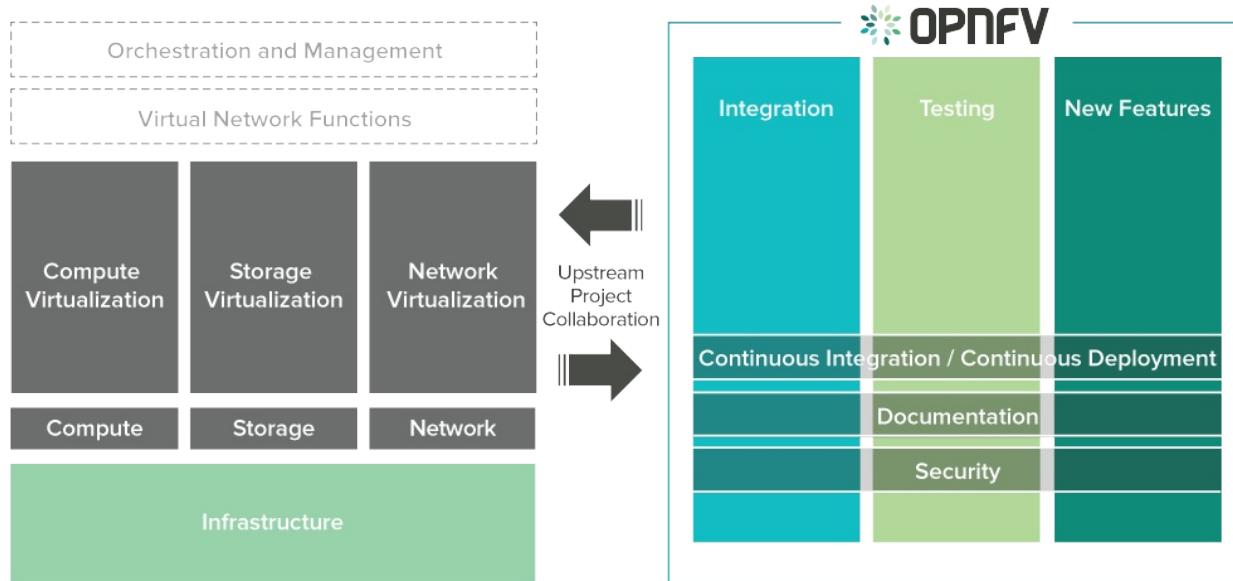
## NFV架构



- NFV VIM (Virtualised Infrastructure Manager), 包括虚拟化（hypervisor或container）以及物理资源（服务器、交换机、存储设备等）
- NFVO (Network Functions Virtualisation Orchestrator)，NFV的管理和编排，包括自动化交付、按需提供资源以及VNF配置（包括物理和虚拟资源）
- VNF (Virtual Network Function)

## OPNFV (Open Platform for NFV)

OPNFV是Linux基金会的开源NFV方案，致力于将其他开源项目通过集成、部署和测试进行系统级的整合，从而搭建一个基准的NFV平台。



## 开源项目

- ONAP (Open Network Automation Platform)
  - 由AT&T主导下的ECOMP（增强控制、编排、管理和策略）项目和中国三大运营商主导下的Open-O项目合并而成
  - 旨在帮助电信行业克服在MANO领域遇到的一些障碍，加快NFV的部署，降低将VNF纳入到网络中的时间和成本
  - 官方网站为<https://www.onap.org/>.
- ETSI OSM
  - ETSI开源的NFV MANO (Management and Orchestration)。
  - 官方网站为<https://osm.etsi.org/>
- OpenStack Tacker
  - 官方网站为<https://wiki.openstack.org/wiki/Tacker>
  - Redhat solution for network functions virtualization
- OpenDaylight: <https://www.opendaylight.org/>
- ONOS: <http://onosproject.org/>
- CORD: <http://opencord.org/>
- Openbaton
- Cloudify
- Clearwater vIMS
- Gohan

## 参考文档

- <http://www.etsi.org/>

- <https://www.onap.org/>
- NFV specifications by ETSI。
- 历数NFV的发展历程

# SD-WAN

SD-WAN (Software-Defined Wide-Area Networking) 是继SDN之后的又一个热门技术，将软件可编程与商业化硬件结合，通过集中管理和软件可编程方式自动部署和管理广域网，加速服务交付，并通过路径优化提升网络性能和可靠性。

SD-WAN服务是基于企业客户的需求，以革新方式亦或是升级方式支持的广域组网SDN解决方案。SDN支持的软件定义广域网络，不仅仅是硬件设备开放的南向接口和Openflow、NETCONF、PCEP等接口协议，也不仅仅是集中的SDN控制器加上业务编排系统，其核心的特点是对应用需求的感知和按需服务，以及以应用视角贯穿的网络构建、用户服务、持续交付、运维支撑、运营保障的全生命周期管理。

目前，全球各大运营商和设备制造商均在积极布局SD-WAN。

## SD-WAN特性

现在普遍认为，SD-WAN应该具有以下4个功能：

- 支持多种连接方式，MPLS，frame relay，LTE，Public Internet等等。SD-WAN将Virtual WAN与传统WAN结合，在这之上做overlay。对于应用程序来说，不需要清楚底层的WAN连接究竟是什么。在不需要传统WAN的场景下，SD-WAN就是Virtual WAN。
- 能够在多种连接之间动态选择链路，以达到负载均衡或者资源弹性。与Virtual WAN类似，动态选择多条路径。SD-WAN如果同时连接了MPLS和Internet，那么可以将一些重要的应用流量，例如VoIP，分流到MPLS，以保证应用的可用性。对于一些对带宽或者稳定性不太敏感的应用流量，例如文件传输，可以分流到Internet上。这样减轻了企业对MPLS的依赖。或者，Internet可以作为MPLS的备份连接，当MPLS出故障了，至少企业的WAN网络不至于也断连。
- 简单的WAN管理接口。凡是涉及网络的事物，似乎都存在管理和故障排查较为复杂的问题，WAN也不例外。SD-WAN通常也会提供一个集中的控制器，来管理WAN连接，设置应用流量policy和优先级，监测WAN连接可用性等等。基于集中控制器，可以再提供CLI或者GUI。以达到简化WAN管理和故障排查的目的。
- 支持VPN，防火墙，网关，WAN优化器等服务。SD-WAN在WAN连接的基础上，将提供尽可能多的，开放的和基于软件的技术。

除此之外，SD-WAN通常还具备以下功能

- 弹性，实时检测网络故障并自动切换链路
- QoS，自动链路选择，为关键应用优化最佳路径
- 安全
- 易部署，易管理，易调试

- 在线流量工程

在部署时，SD-WAN与WAN edge router放置在一起，用来增强WAN edge router甚至替代WAN edge router。

## 相关技术

### Hybrid WAN

Hybrid WAN是指采用同时采用多种WAN连接，通常就是私有MPLS连接和Internet连接。企业通过Hybrid WAN技术，可以将一些应用流量分流到Internet连接上来。毕竟，私有MPLS连接成本不低。从这点看，Hybrid WAN与前面描述的SD-WAN非常接近。实际InfoVista将hybrid WAN看作是SD-WAN的前身。不过Hybrid WAN只是强调同时使用多条WAN连接，SD-WAN在这之上加上了software-defined的概念，这包括了集中控制，智能分析和动态创建网络服务等。Hybrid WAN仍然占据了WAN市场较大一部分，当用户需要升级或者需要更灵活的WAN连接管理时，SD-WAN会是一个不错的替代。

### WAN Optimization

WAN Optimization是指提高数据在WAN上传输效率的技术的集合。SD-WAN关注的是使用低成本线路，以达到高性能线路传输效果。而WAN Optimization关注的是网络数据包如何更有效的在已有线路上传输。在实际中，SD-WAN可以配合WAN Optimization使用。在SD-WAN场景下，WAN Optimization通常是以虚拟的形式存在。

### WAN edge router

前面说过，SD-WAN实际上能增强WAN edge router甚至取而代之。传统的网络厂商一般是在自己的WAN edge device（路由器，NGFW）里集成SD-WAN功能，而新兴的SD-WAN创业公司，更倾向于专有的SD-WAN设备，或者虚拟的SD-WAN产品，来配合WAN edge router。

### MPLS

SD-WAN的倡导者通常会宣称SD-WAN是用来替代MPLS的。不过，只要对网络流量可靠的QoS还有需要，那么MPLS或者其他的传统WAN连接技术仍然是不能替代。现实中，SD-WAN厂商通常会建议MPLS和Virtual WAN一起部署。对于高优先级流量，仍然走MPLS。从技术的角度来看，MPLS，可以通过Traffic Engineering完全控制骨干网网络流量。而SD-WAN，其所有的控制都是在网络边缘。网络对于SD-WAN来说就是个黑盒子。所以总的来说，SD-WAN可以减轻企业对MPLS的依赖，但是不能完全消除MPLS。

### NFV

SD-WAN产品需要支持基于软件的VPN，防火墙，WAN Optimization等。这可以在SD-WAN上实现，也可以通过NFV技术向SD-WAN添加相应的VNF来实现。NFV和SD-WAN都是虚拟网络服务，两者并不互斥，可以配合工作。

## SDN

与SDN的联系更多是概念上。前面已经提过了SD-WAN与SDN的区别。这里再引用一个报告，Riverbed 2015年通过对260个样本调查发现，29%的用户正在研究SD-WAN，而已经有5%的用户在使用SD-WAN。相比之下，77%的用户在研究SDN，只有13%在使用。SD-WAN的先驱使用者是零售商和金融机构，他们都有大量的分支机构。SDN是对现有网络架构的更新，虽然说SDN架构优势明显，但是应用到实际中，因为企业现有网络架构在还能用之前，没人会提出更换成SDN架构，企业不会承担相应的成本和风险。而SD-WAN，最直接的效应就是减少企业在WAN上的投入，特别是分支机构较多的企业。设计到钱的问题的时候，总是比涉及技术更容易在企业推广。并且SD-WAN是增量变化，企业有时可以在原有WAN架构的基础上新增SD-WAN功能。因此，普遍认为SD-WAN的发展速度会更快。

## SD-WAN厂商

- Viptela
- VeloCloud
- Aryaka

更多SD-WAN厂商列表见[这里](#)。

国外运营商SDWAN服务及提供商：

NETMANIAS ONE-SHOT		Operator's Managed SD-WAN Service			Updated: May 24, 2017
Operator	SD-WAN service	Launched	Country	SD-WAN Vendor	
AT&T	FlexWare	2017.planned	US	Velocloud	
Sprint	Sprint SD-WAN	2017.05	US	Velocloud	
Comcast	Beta Trial	2017.05	US	Versa Networks	
GTT	Managed SD-WAN	2017.04	US, Global	Velocloud	
Global Capacity	Managed SD-WAN	2017.04	US, Global	Velocloud	
MegaPath	SD-WAN	2017.03	US	Velocloud	
Transbeam	Transbeam SD-WAN	2017.02	US	Velocloud	
Windstream	Windstream SD-WAN Concierge	2017.01	US	Velocloud	
Vonage	SmartWAN	2016.12	US	Velocloud	
Mitel	MiCloud Edge	2016.11	US	Velocloud	
TelePacific	SD-WAN	2016.10	US	Velocloud	
EarthLink	EarthLink SD-WAN Concierge	2016.09	US	Velocloud	
Verizon	VNS - SD WAN	2016.07	US, Europe, Asia	Viptela, Cisco	
CentryLink	CentryLink SD-WAN	2016.06	US	Versa Networks	
MetTel	SD-WAN	2016.05	US	Velocloud	
Sonera	SD-WAN	2017.03	Finland	Nuage (Nokia)	
Exponential-e	SD-WAN (Cloudnet)	2016.11	UK	Nuage (Nokia)	
Colt	Colt SD WAN	2016.10	Europe	Versa Networks	
Telefonica	SD-WAN	2016.07	Spain	Cisco, Nuage (Nokia)	
Deutsche Telekom	Beta Trial	2016.06	Germany	Velocloud	
BT	Connect Intelligence IWAN SD-WAN	2016.01	UK	Cisco, Nuage (Nokia)	
CHT	CHT Global SD-WAN	2017.03	Taiwan, Global	Velocloud	
NTTPC	Master'sONE	2017.01	Japan	Viptela	
Tata	IZO SDWAN	2016.11	India,Global	Cisco, Versa, Velocloud	
NTT	SD-WAN (SD-NS)	2016.10	Japan	Nuage (Nokia)	
Telstra	Managed SD-WAN	2016.05	Australia	Cisco, Velocloud	
Singtel	ConnectPlus SD-WAN	2015.01	Singapore	Viptela	

©2017 Netmanias • www.netmanias.com

(图片来自NET MANIAS)

注：部分转自SD-WAN和SD-WAN漫谈。



# 容器网络

本章介绍Kubernetes的网络模型以及常见插件的原理和使用方法，主要包括

- **Host Network**：最简单的网络模型就是让容器共享Host的network namespace，使用宿主机的网络协议栈。这样，不需要额外的配置，容器就可以共享宿主的各种网络资源。
- **共享容器网络**：多个容器共享同一个netns，只需要第一个容器配置网络。比如Kubernetes Pod就是所有容器共享同一个pause容器的网络。
- **Docker主推的CNM（Container network model）**：是Docker的网络模型，主要由Sandbox、Network以及Endpoint组成。
- **CNCF主推的CNI（Container Network Interface）**：CNI是由CoreOS发起的容器网络规范，是Kubernetes网络插件的基础。其基本思想为：Container Runtime在创建容器时，先创建好network namespace，然后调用CNI插件为这个netns配置网络，其后再启动容器内的进程。目前已经贡献给CNCF，将成为CNCF主推的网络模型。
- **Kubernetes网络模型**：IP-Per-Pod网络模型，基于CNI的网络插件，并基于CNI bridge实现了kubenet网络插件。

常见的容器网络插件对比：

Feature	Docker Overlay Network	Calico	Flannel	Weave Net	Canal (Calico + Flannel)	Romana	Apoereo/Trireme	Cisco Contiv	Covalent Cilium
Open Source	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Network Model	VXLAN overlay	L3 with optional encapsulation	VXLAN or UDP overlay	VXLAN or UDP overlay; IP routed for AWS VPC	VXLAN or UDP overlay	Layer 3	Layer 3 with TLS	Layer 2, Layer 3 (BGP) & VXLAN overlay	L3 with optional encapsulation
Application Isolation	CIDR Schema	Policy Schema based on labels, cidrs, ports and profiles	CIDR Schema	CIDR Schema, Profile Schema	Policy Schema based on labels, cidrs, ports and profiles	CIDR Schema	TLS-based	Both Label based as well as CIDR Schema	Policy based on labels
Isolation from Host Network NS	YES	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Egress policy									Artificially limited to IPv4/IPv6 & TCP/UDP/ICMPv6
Protocol Support	All except multicast	All	All	All	All	All	TCP only	All	
Built-in Name Service	YES	No	No	Yes	No	No	No	Yes	No
Built-in Service Load Balancer	YES	No	No	No	No	No	No	Yes	Yes
Cluster Store Requirements	None	etcd/k8s API	etcd/k8s API	None	etcd/k8s API	etcd/Consul Zookeeper	None	etcd/Consul	Yes (consul or etcd)
Encryption Channel	YES	No	No	NaCl Library	No	No	TLS	No	Yes (IPSec)
Partially Connected Network Support	NO	??	No	RR	No	No	N/A	??	No
Separate vNIC for Container	YES	Yes	No	Yes	No	No	No	Yes	Yes (shared logical routing table)
IP Overlap Support	YES	No	No	No	No	No	No	Yes, multiple VRFs	No
Container Subnet Restriction	YES	No	No	Yes, configurable after start	No	No	No	No restriction	No
Multicast support	NO	No	No	Yes	No	No	No	Yes	No
Pods routable from outside cluster	N/A	Yes	No	Yes	No	No	No	Yes	Yes
Container Networking Interface	N/A	Yes	Yes	Yes	Yes	Yes	?	Yes	Yes
Container Networking Model	YES	Yes	No	Yes	No	No	?	Yes	Yes
OpenStack Support	N/A	Yes	No	Yes	No	Yes	Yes	Yes	No
Kubernetes Support	N/A	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Mesos Support	N/A	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Docker Support	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes
rkt Support	N/A	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Cloud Foundry Support	NO	?	Yes	?	?	No	No	Yes	No
Nomad Support	N/A	No	No	No	No	No	No	Yes	No
URL to Networking Architecture		Calico Reference Architecture	Flannel	Introducing Weave	tigera/canal	Romana Basics	Trireme Architecture	contiv.io	<a href="https://github.com/cilium/cilium">https://github.com/cilium/cilium</a>

(来自 [Container Native Networking - Comparison](#))

# Host network

最简单的网络模型就是让容器共享**Host**的**network namespace**，使用宿主机的网络协议栈。这样，不需要额外的配置，容器就可以共享宿主的各种网络资源。

## 优点

- 简单，不需要任何额外配置
- 高效，没有NAT等额外的开销

## 缺点

- 没有任何的网络隔离
- 容器和**Host**的端口号容易冲突
- 容器内任何网络配置都会影响整个宿主机

# CNI (Container Network Interface)

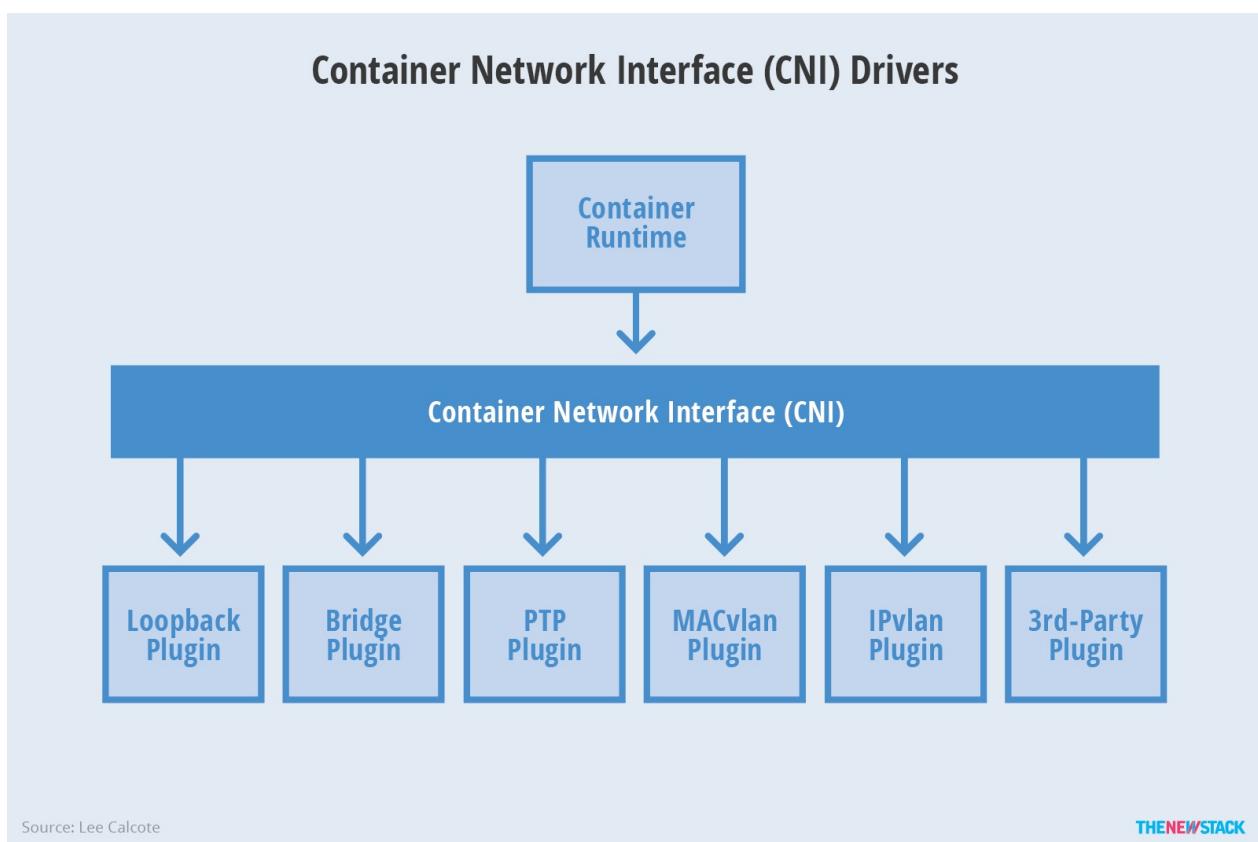
Container Network Interface (CNI) 最早是由CoreOS发起的容器网络规范，是Kubernetes网络插件的基础。其基本思想为：Container Runtime在创建容器时，先创建好network namespace，然后调用CNI插件为这个netns配置网络，其后再启动容器内的进程。现已加入CNCF，成为CNCF主推的网络模型。

CNI插件包括两部分：

- CNI Plugin负责给容器配置网络，它包括两个基本的接口
  - 配置网络: AddNetwork(*net NetworkConfig, rt RuntimeConf*) (*types.Result, error*)
  - 清理网络: DelNetwork(*net NetworkConfig, rt RuntimeConf*) *error*
- IPAM Plugin负责给容器分配IP地址，主要实现包括host-local和dhcp。

Kubernetes Pod 中的其他容器都是Pod所属pause容器的网络，创建过程为：

1. kubelet 先创建pause容器生成network namespace
2. 调用网络CNI driver
3. CNI driver 根据配置调用具体的cni 插件
4. cni 插件给pause 容器配置网络
5. pod 中其他的容器都使用 pause 容器的网络



所有CNI插件均支持通过环境变量和标准输入传入参数：

```
$ echo '{"cniVersion": "0.3.1", "name": "mynet", "type": "macvlan", "bridge": "cni0", "isGateway": true, "ipMasq": true, "ipam": {"type": "host-local", "subnet": "10.244.1.0/24", "routes": [{"dst": "0.0.0.0/0"}]} }' | sudo CNI_COMMAND=ADD CNI_NETNS=/var/run/netns/a CNI_PATH=./bin CNI_IFNAME=eth0 CNI_CONTAINERID=a CNI_VERSION=0.3.1 ./bin/bridge
```

```
$ echo '{"cniVersion": "0.3.1", "type": "IGNORED", "name": "a", "ipam": {"type": "host-local", "subnet": "10.1.2.3/24"} }' | sudo CNI_COMMAND=ADD CNI_NETNS=/var/run/netns/a CNI_PATH=./bin CNI_IFNAME=a CNI_CONTAINERID=a CNI_VERSION=0.3.1 ./bin/host-local
```

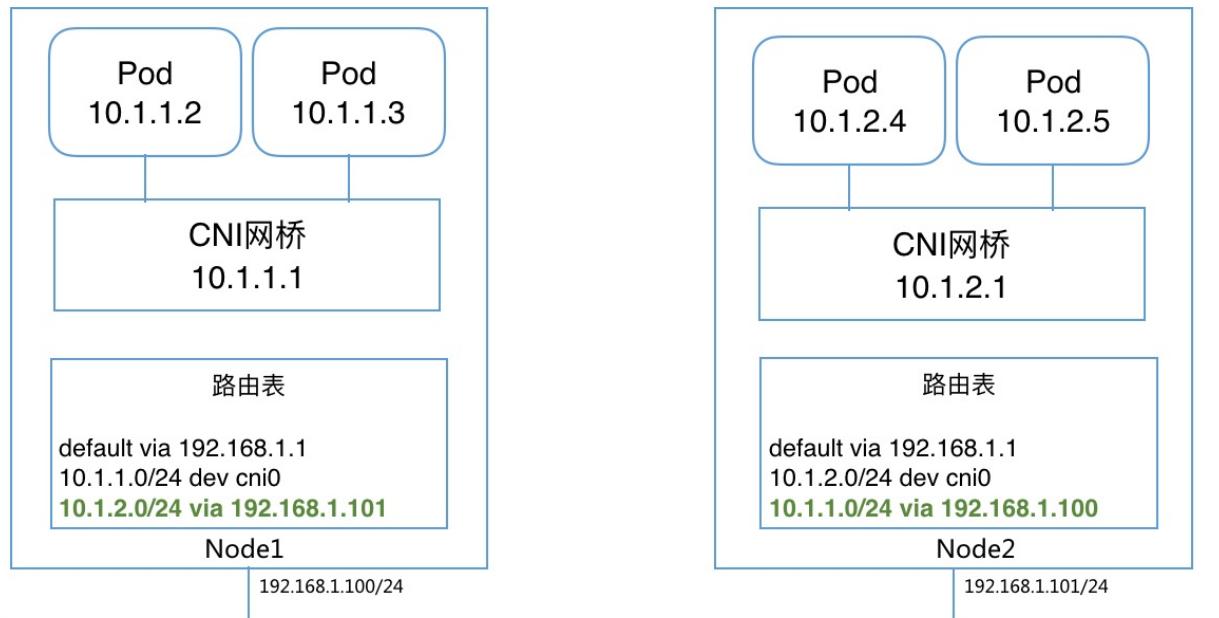
常见的CNI网络插件有

- Bridge
- PTP
- IPVLAN
- MACVLAN
- VLAN
- PORTMAP

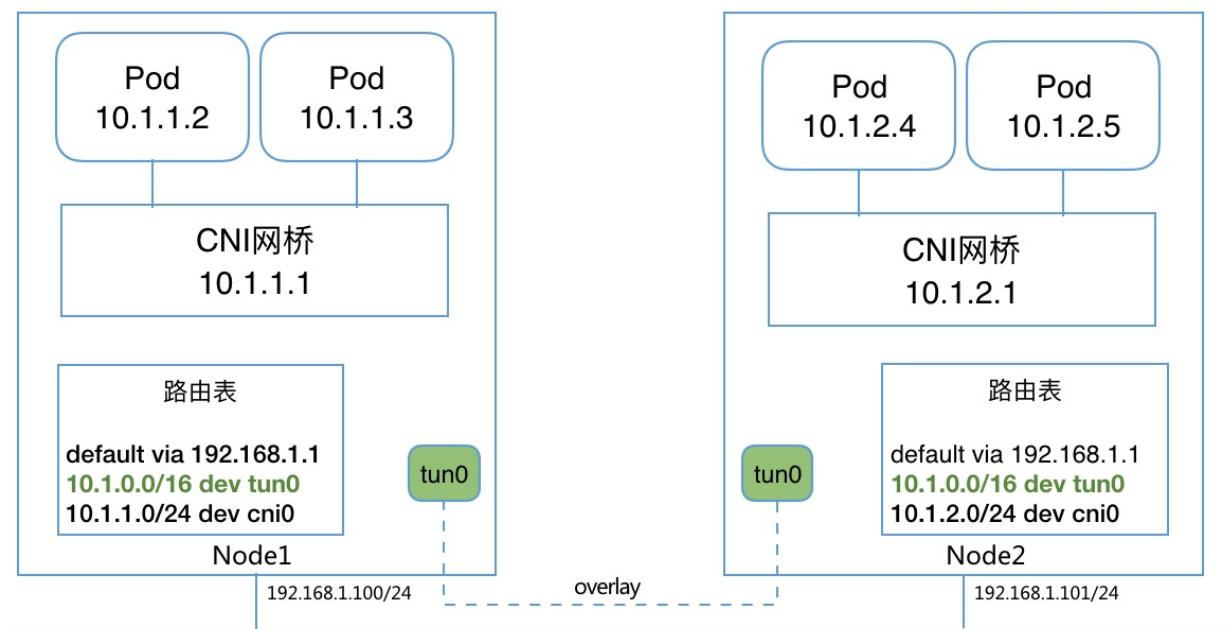


## Bridge

Bridge是最简单的CNI网络插件，它首先在Host创建一个网桥，然后再通过veth pair连接该网桥到container netns。



注意：**Bridge**模式下，多主机网络通信需要额外配置主机路由，或使用**overlay**网络。可以借助[Flannel](#)或者[Quagga](#)动态路由等来自动配置。比如**overlay**情况下的网络结构为



### 配置示例

```
{
  "cniVersion": "0.3.0",
  "name": "mynet",
  "type": "bridge",
  "bridge": "mynet0",
  "isDefaultGateway": true,
  "forceAddress": false,
  "ipMasq": true,
  "hairpinMode": true,
  "ipam": {
    "type": "host-local",
    "subnet": "10.10.0.0/16"
  }
}
```

```

# export CNI_PATH=/opt/cni/bin
# ip netns add ns
# /opt/cni/bin/cnitoold add mynet /var/run/netns/ns
{
    "interfaces": [
        {
            "name": "mynet0",
            "mac": "0a:58:0a:0a:00:01"
        },
        {
            "name": "vethc763e31a",
            "mac": "66:ad:63:b4:c6:de"
        },
        {
            "name": "eth0",
            "mac": "0a:58:0a:0a:00:04",
            "sandbox": "/var/run/netns/ns"
        }
    ],
    "ips": [
        {
            "version": "4",
            "interface": 2,
            "address": "10.10.0.4/16",
            "gateway": "10.10.0.1"
        }
    ],
    "routes": [
        {
            "dst": "0.0.0.0/0",
            "gw": "10.10.0.1"
        }
    ],
    "dns": {}
}
# ip netns exec ns ip addr
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
9: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 0a:58:0a:0a:00:04 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.10.0.4/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::8c78:6dff:fe19:f6bf/64 scope link tentative dadfailed
        valid_lft forever preferred_lft forever
# ip netns exec ns ip route
default via 10.10.0.1 dev eth0
10.10.0.0/16 dev eth0 proto kernel scope link src 10.10.0.4

```

## DHCP

DHCP插件是最主要的IPAM插件之一，用来通过DHCP方式给容器分配IP地址，在macvlan插件中也会用到DHCP插件。

在使用DHCP插件之前，需要先启动dhcp daemon:

```
/opt/cni/bin/dhcp daemon &
```

然后配置网络使用dhcp作为IPAM插件

```
{
  ...
  "ipam": {
    "type": "dhcp",
  }
}
```

## host-local

host-local是最常用的CNI IPAM插件，用来给container分配IP地址。

IPv4:

```
{
  "ipam": {
    "type": "host-local",
    "subnet": "10.10.0.0/16",
    "rangeStart": "10.10.1.20",
    "rangeEnd": "10.10.3.50",
    "gateway": "10.10.0.254",
    "routes": [
      { "dst": "0.0.0.0/0" },
      { "dst": "192.168.0.0/16", "gw": "10.10.5.1" }
    ],
    "dataDir": "/var/my-orchestrator/container-ipam-state"
  }
}
```

IPv6:

```
{
    "ipam": {
        "type": "host-local",
        "subnet": "3ffe:ffff:0:01ff::/64",
        "rangeStart": "3ffe:ffff:0:01ff::0010",
        "rangeEnd": "3ffe:ffff:0:01ff::0020",
        "routes": [
            { "dst": "3ffe:ffff:0:01ff::1/64" }
        ],
        "resolvConf": "/etc/resolv.conf"
    }
}
```

## ptp

ptp插件通过veth pair给容器和host创建点对点连接：veth pair一端在container netns内，另一端在host上。可以通过配置host端的IP和路由来让ptp连接的容器之前通信。

```
{
    "name": "mynet",
    "type": "ptp",
    "ipam": {
        "type": "host-local",
        "subnet": "10.1.1.0/24"
    },
    "dns": {
        "nameservers": [ "10.1.1.1", "8.8.8.8" ]
    }
}
```

## IPVLAN

IPVLAN 和 MACVLAN 类似，都是从一个主机接口虚拟出多个虚拟网络接口。一个重要的区别就是所有的虚拟接口都有相同的 mac 地址，而拥有不同的 ip 地址。因为所有的虚拟接口要共享 mac 地址，所以有些需要注意的地方：

- DHCP 协议分配 ip 的时候一般会用 mac 地址作为机器的标识。这个情况下，客户端动态获取 ip 的时候需要配置唯一的 ClientID 字段，并且 DHCP server 也要正确配置使用该字段作为机器标识，而不是使用 mac 地址

IPVLAN支持两种模式：

- L2 模式：此时跟macvlan bridge 模式工作原理很相似，父接口作为交换机来转发子接口的数据。同一个网络的子接口可以通过父接口来转发数据，而如果想发送到其他网络，

报文则会通过父接口的路由转发出去。

- L3 模式：此时 `ipvlan` 有点像路由器的功能，它在各个虚拟网络和主机网络之间进行不同网络报文的路由转发工作。只要父接口相同，即使虚拟机/容器不在同一个网络，也可以互相 `ping` 通对方，因为 `ipvlan` 会在中间做报文的转发工作。注意 L3 模式下的虚拟接口不会接收到多播或者广播的报文（这个模式下，所有的网络都会发送给父接口，所有的 ARP 过程或者其他多播报文都是在底层的父接口完成的）。另外外部网络默认情况下是不知道 `ipvlan` 虚拟出来的网络的，如果不在外部路由器上配置好对应的路由规则，`ipvlan` 的网络是不能被外部直接访问的。

创建 `ipvlan` 的简单方法为

```
ip link add link <master-dev> <slave-dev> type ipvlan mode { 12 | L3 }
```

cni 配置格式为

```
{
  "name": "mynet",
  "type": "ipvlan",
  "master": "eth0",
  "ipam": {
    "type": "host-local",
    "subnet": "10.1.2.0/24"
  }
}
```

需要注意的是

- `ipvlan` 插件下，容器不能跟 Host 网络通信
- 主机接口（也就是 master interface）不能同时作为 `ipvlan` 和 `macvlan` 的 master 接口

## MACVLAN

MACVLAN 可以从一个主机接口虚拟出多个 macvtap，且每个 macvtap 设备都拥有不同的 mac 地址（对应不同的 linux 字符设备）。MACVLAN 支持四种模式

- bridge 模式：数据可以在同一 master 设备的子设备之间转发
- vepa 模式：VEPA 模式是对 802.1Qbg 标准中的 VEPA 机制的软件实现，MACVTAP 设备简单的将数据转发到 master 设备中，完成数据汇聚功能，通常需要外部交换机支持 Hairpin 模式才能正常工作
- private 模式：Private 模式和 VEPA 模式类似，区别是子 MACVTAP 之间相互隔离
- passthrough 模式：内核的 MACVLAN 数据处理逻辑被跳过，硬件决定数据如何处理，从而释放了 Host CPU 资源

创建macvlan的简单方法为

```
ip link add link <master-dev> name macvtap0 type macvtap
```

cni配置格式为

```
{
    "name": "mynet",
    "type": "macvlan",
    "master": "eth0",
    "ipam": {
        "type": "dhcp"
    }
}
```

需要注意的是

- macvlan需要大量 mac 地址，每个虚拟接口都有自己的 mac 地址
- 无法和 802.11(wireless) 网络一起工作
- 主机接口（也就是master interface）不能同时作为ipvlan和macvlan的master接口

## Flannel

Flannel通过给每台宿主机分配一个子网的方式为容器提供虚拟网络，它基于Linux TUN/TAP，使用UDP封装IP包来创建overlay网络，并借助etcd维护网络的分配情况。

## Weave Net

Weave Net是一个多主机容器网络方案，支持去中心化的控制平面，各个host上的wRouter间通过建立Full Mesh的TCP链接，并通过Gossip来同步控制信息。这种方式省去了集中式的K/V Store，能够在一定程度上减低部署的复杂性，Weave将其称为“data centric”，而非RAFT或者Paxos的“algorithm centric”。

数据平面上，Weave通过UDP封装实现L2 Overlay，封装支持两种模式，一种是运行在user space的sleeve mode，另一种是运行在kernal space的 fastpath mode。Sleeve mode通过pcap设备在Linux bridge上截获数据包并由wRouter完成UDP封装，支持对L2 traffic进行加密，还支持Partial Connection，但是性能损失明显。Fastpath mode即通过OVS的odp封装VxLAN并完成转发，wRouter不直接参与转发，而是通过下发odp流表的方式控制转发，这种方式可以明显地提升吞吐量，但是不支持加密等高级功能。

## Contiv

Contiv是思科开源的容器网络方案，主要提供基于Policy的网络管理，并与主流容器编排系统集成。Contiv最主要的优势是直接提供了多租户网络，并支持L2(VLAN), L3(BGP), Overlay (VXLAN)以及思科自家的ACI。

## Calico

Calico是一个基于BGP的纯三层的数据中心网络方案（不需要Overlay），并且与OpenStack、Kubernetes、AWS、GCE等IaaS和容器平台都有良好的集成。

Calico在每一个计算节点利用Linux Kernel实现了一个高效的vRouter来负责数据转发，而每个vRouter通过BGP协议负责把自己上运行的工作负载（workload）的路由信息像整个Calico网络内传播——小规模部署可以直接互联，大规模下可通过指定的BGP route reflector来完成。这样保证最终所有的workload之间的数据流量都是通过IP路由的方式完成互联的。Calico节点组网可以直接利用数据中心的网络结构（无论是L2或者L3），不需要额外的NAT，隧道或者Overlay Network。

此外，Calico基于iptables还提供了丰富而灵活的网络Policy，保证通过各个节点上的ACLs来提供Workload的多租户隔离、安全组以及其他可达性限制等功能。

## OVN

OVN (Open Virtual Network) 是OVS提供的原生虚拟化网络方案，旨在解决传统SDN架构（比如Neutron DVR）的性能问题。

OVN为Kubernetes提供了两种网络方案：

- Overlay: 通过ovs overlay连接容器
- Underlay: 将VM内的容器连到VM所在的相同网络（开发中）

其中，容器网络的配置是通过OVN的CNI插件来实现。

## SR-IOV

Intel维护了一个SR-IOV的CNI插件，fork自，并扩展了DPDK的支持。

项目主页见<https://github.com/Intel-Corp/sriov-cni>。

## Romana

Romana是Panic Networks在2016年提出的开源项目，旨在借鉴 route aggregation的思路来解决Overlay方案给网络带来的开销。

## OpenContrail

OpenContrail是Juniper推出的开源网络虚拟化平台，其商业版本为Contrail。其主要由控制器和vRouter组成：

- 控制器提供虚拟网络的配置、控制和分析功能
- vRouter提供分布式路由，负责虚拟路由器、虚拟网络的建立以及数据转发

其中，vRouter支持三种模式

- Kernel vRouter：类似于ovs内核模块
- DPDK vRouter：类似于ovs-dpdk
- Netronome Agilio Solution (商业产品)：支持DPDK, SR-IOV and Express Virtio (XVIO)

[michaelhenkel/opencontrail-cni-plugin](#)提供了一个OpenContrail的CNI插件。

## CNI Plugin Chains

CNI还支持Plugin Chains，即指定一个插件列表，由Runtime依次执行每个插件。这对支持端口映射（portmapping）、虚拟机等非常有帮助。

使用方法请参考[这里](#)。

## 其他

### Canal

Canal是Flannel和Calico联合发布的一个统一网络插件，提供CNI网络插件，并支持network policy。

### kuryr-kubernetes

kuryr-kubernetes是OpenStack推出的集成Neutron网络插件，主要包括Controller和CNI插件两部分，并且也提供基于Neutron LBaaS的Service集成。

### Cilium

Cilium是一个基于eBPF和XDP的高性能容器网络方案，提供了CNI和CNM插件。

项目主页为<https://github.com/cilium/cilium>。

## CNI-Genie

**CNI-Genie**是华为PaaS团队推出的同时支持多种网络插件（支持calico, canal, romana, weave等）的CNI插件。

项目主页为<https://github.com/Huawei-PaaS/CNI-Genie>。

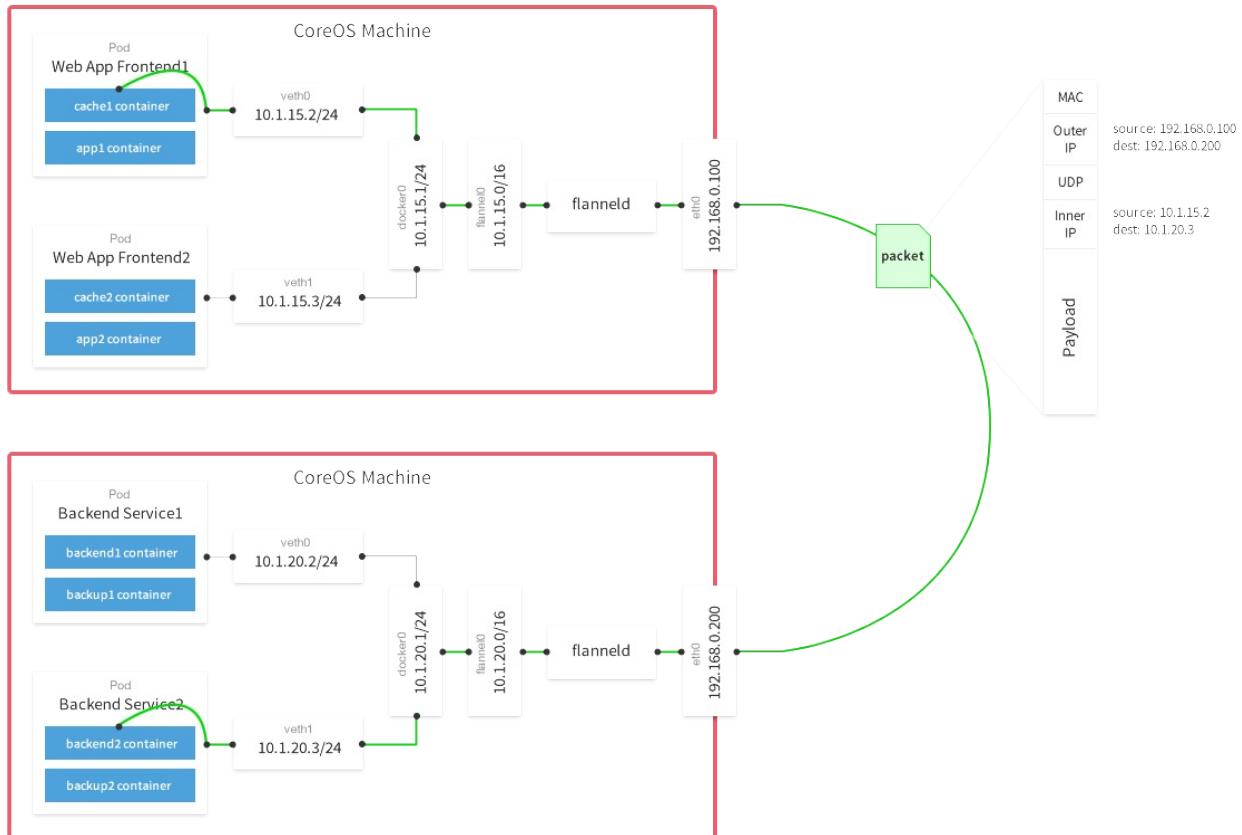
# Flannel

Flannel通过给每台宿主机分配一个子网的方式为容器提供虚拟网络，它基于Linux TUN/TAP，使用UDP封装IP包来创建overlay网络，并借助etcd维护网络的分配情况。

## Flannel原理

控制平面上host本地的flanneld负责从远端的ETCD集群同步本地和其它host上的subnet信息，并为POD分配IP地址。数据平面flannel通过Backend（比如UDP封装）来实现L3 Overlay，既可以选择一般的TUN设备又可以选择VxLAN设备。

```
{  
    "Network": "10.0.0.0/8",  
    "SubnetLen": 20,  
    "SubnetMin": "10.10.0.0",  
    "SubnetMax": "10.99.0.0",  
    "Backend": {  
        "Type": "udp",  
        "Port": 7890  
    }  
}
```



除了 UDP，Flannel 还支持很多其他的 Backend：

- **udp**：使用用户态 udp 封装，默认使用 8285 端口。由于是在用户态封装和解包，性能上有较大的损失
- **vxlan**：vxlan 封装，需要配置 VNI，Port（默认 8472）和 GBP
- **host-gw**：直接路由的方式，将容器网络的路由信息直接更新到主机的路由表中，仅适用于二层直接可达的网络
- **aws-vpc**：使用 Amazon VPC route table 创建路由，适用于 AWS 上运行的容器
- **gce**：使用 Google Compute Engine Network 创建路由，所有 instance 需要开启 IP forwarding，适用于 GCE 上运行的容器
- **ali-vpc**：使用阿里云 VPC route table 创建路由，适用于阿里云上运行的容器

## Docker 集成

```
source /run/flannel/subnet.env
docker daemon --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU} &
```

## CNI 集成

CNI flannel插件会将flannel网络配置转换为bridge插件配置，并调用bridge插件给容器netns配置网络。比如下面的flannel配置

```
{  
    "name": "mynet",  
    "type": "flannel",  
    "delegate": {  
        "bridge": "mynet0",  
        "mtu": 1400  
    }  
}
```

会被cni flannel插件转换为

```
{  
    "name": "mynet",  
    "type": "bridge",  
    "mtu": 1472,  
    "ipMasq": false,  
    "isGateway": true,  
    "ipam": {  
        "type": "host-local",  
        "subnet": "10.1.17.0/24"  
    }  
}
```

## Kubernetes集成

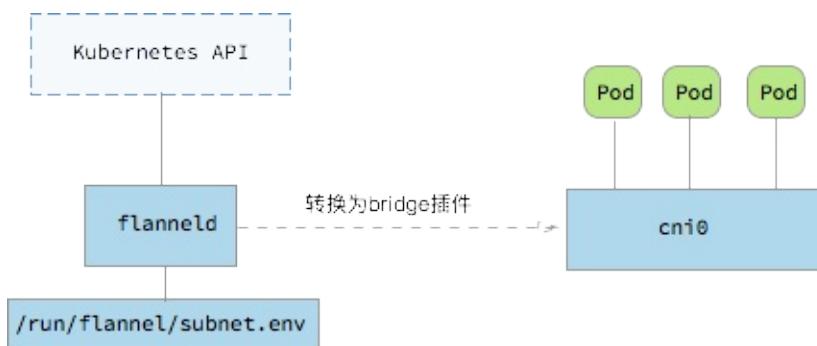
使用flannel前需要配置 `kube-controller-manager --allocate-node-cidrs=true --cluster-cidr=10.244.0.0/16`。

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```

这会启动flanneld容器，并配置CNI网络插件：

```
$ ps -ef | grep flannel | grep -v grep
root      3625  3610  0 13:57 ?        00:00:00 /opt/bin/flanneld --ip-masq --kube-sub
net-mgr
root      9640  9619  0 13:51 ?        00:00:00 /bin/sh -c set -e -x; cp -f /etc/kube-
flannel/cni-conf.json /etc/cni/net.d/10-flannel.conf; while true; do sleep 3600; done

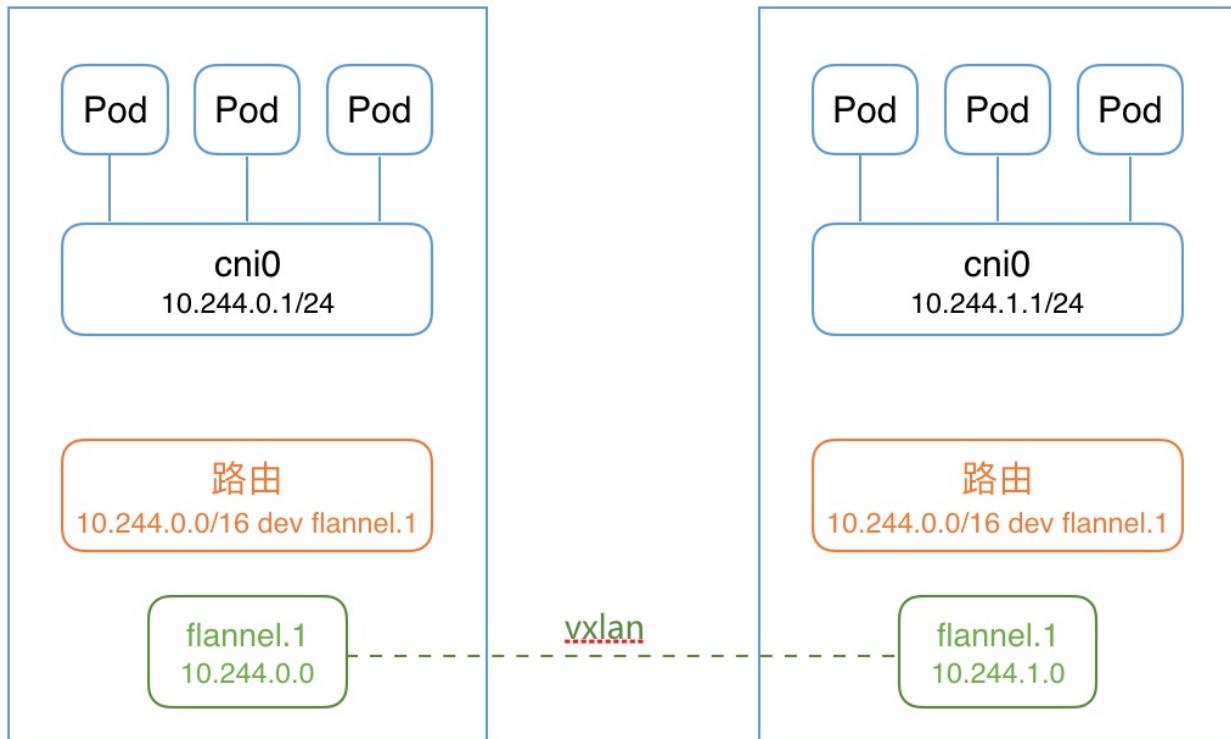
$ cat /etc/cni/net.d/10-flannel.conf
{
  "name": "cbr0",
  "type": "flannel",
  "delegate": {
    "isDefaultGateway": true
  }
}
```



flanneld 自动连接 kubernetes API，根据 node.Spec.PodCIDR 配置本地的 flannel 网络子网，并为容器创建 vxlan 和相关的子网路由。

```
$ cat /run/flannel/subnet.env
FLANNEL_NETWORK=10.244.0.0/16
FLANNEL_SUBNET=10.244.0.1/24
FLANNEL_MTU=1410
FLANNEL_IPMASQ=true

$ ip -d link show flannel.1
12: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc noqueue state UNKNOWN
mode DEFAULT group default
    link/ether 8e:5a:0d:07:0f:0d brd ff:ff:ff:ff:ff:ff promiscuity 0
    vxlan id 1 local 10.146.0.2 dev ens4 srcport 0 0 dstport 8472 nolearning ageing 30
    0 udpchecksum addrgenmode eui64
```



## 优点

- 配置安装简单，使用方便
- 与云平台集成较好，VPC的方式没有额外的性能损失

## 缺点

- VXLAN模式对zero-downtime restarts支持不好

When running with a backend other than udp, the kernel is providing the data path with flanneld acting as the control plane. As such, flanneld can be restarted (even to do an upgrade) without disturbing existing flows. However in the case of vxlan backend, this needs to be done within a few seconds as ARP entries can start to timeout requiring the flannel daemon to refresh them. Also, to avoid interruptions during restart, the configuration must not be changed (e.g. VNI, --iface values).

## 参考文档

- <https://github.com/coreos/flannel>
- <https://coreos.com/flannel/docs/latest/>

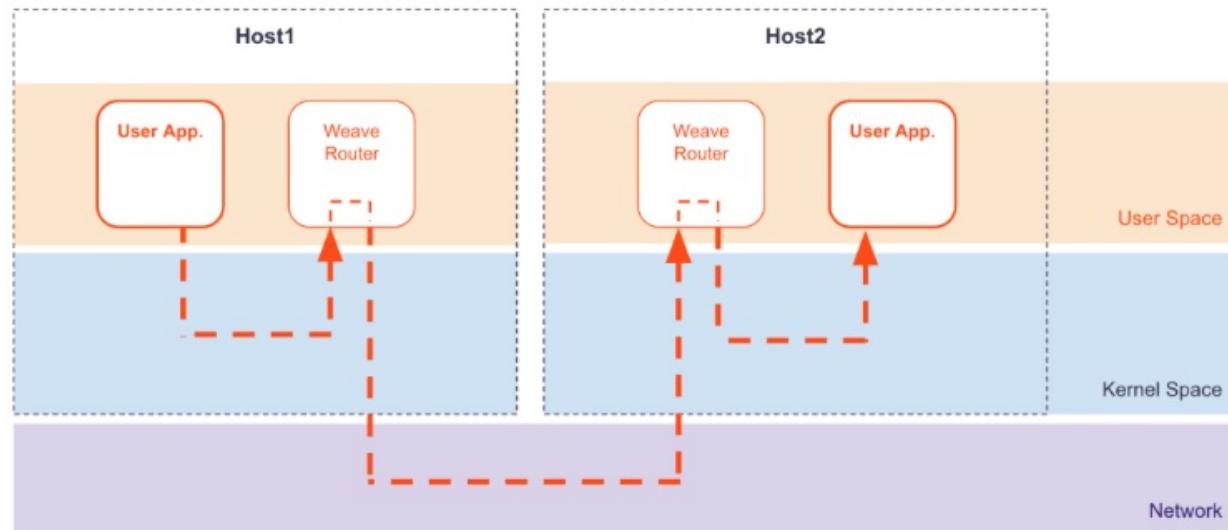
# Weave Net

Weave Net是一个多主机容器网络方案，支持去中心化的控制平面，各个host上的wRouter间通过建立Full Mesh的TCP链接，并通过Gossip来同步控制信息。这种方式省去了集中式的K/V Store，能够在一定程度上减低部署的复杂性，Weave将其称为“data centric”，而非RAFT或者Paxos的“algorithm centric”。

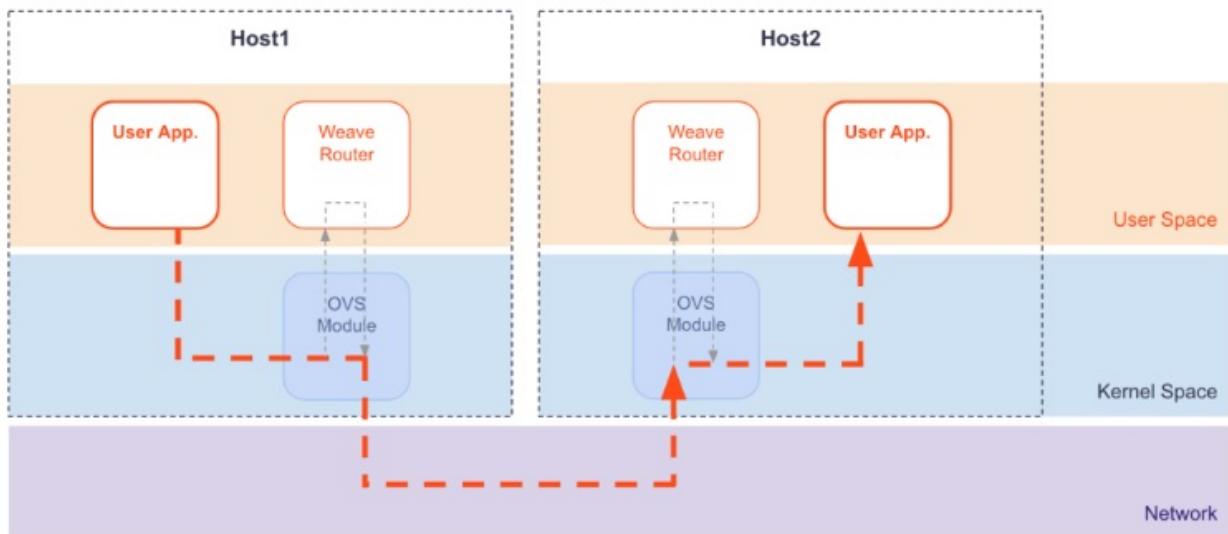
数据平面上，Weave通过UDP封装实现L2 Overlay，封装支持两种模式：

- 运行在user space的sleeve mode：通过pcap设备在Linux bridge上截获数据包并由wRouter完成UDP封装，支持对L2 traffic进行加密，还支持Partial Connection，但是性能损失明显。
- 运行在kernel space的 fastpath mode：即通过OVS的odp封装VxLAN并完成转发，wRouter不直接参与转发，而是通过下发odp流表的方式控制转发，这种方式可以明显地提升吞吐量，但是不支持加密等高级功能。

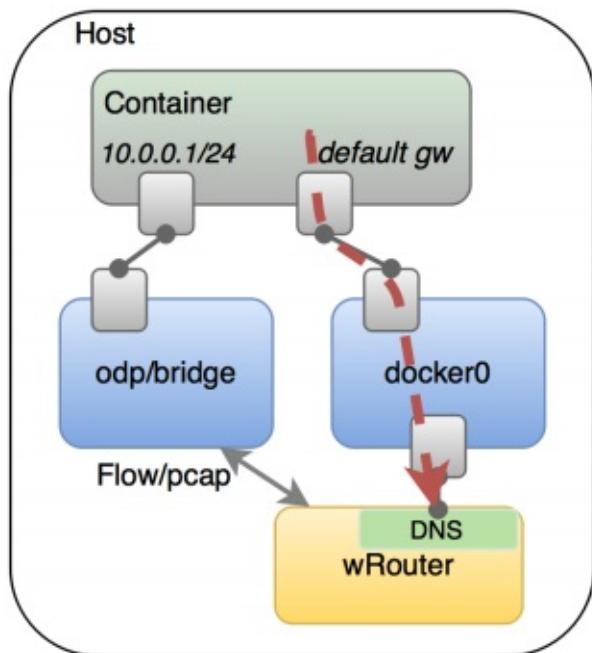
Sleeve Mode:



Fastpath Mode:



关于Service的发布，weave做的也比较完整。首先，wRouter集成了DNS功能，能够动态地进行服务发现和负载均衡，另外，与libnetwork 的overlay driver类似，weave要求每个POD有两个网卡，一个就连在lb/ovs上处理L2 流量，另一个则连在docker0上处理Service流量， docker0后面仍然是iptables作NAT。



Weave已经集成了主流的容器系统

- Docker: <https://www.weave.works/docs/net/latest/plugin/>
- Kubernetes: <https://www.weave.works/docs/net/latest/kube-addon/>
  - `kubectl apply -f https://git.io/weave-kube`
- CNI: <https://www.weave.works/docs/net/latest/cni-plugin/>
- Prometheus: <https://www.weave.works/docs/net/latest/metrics/>

# Weave Kubernetes

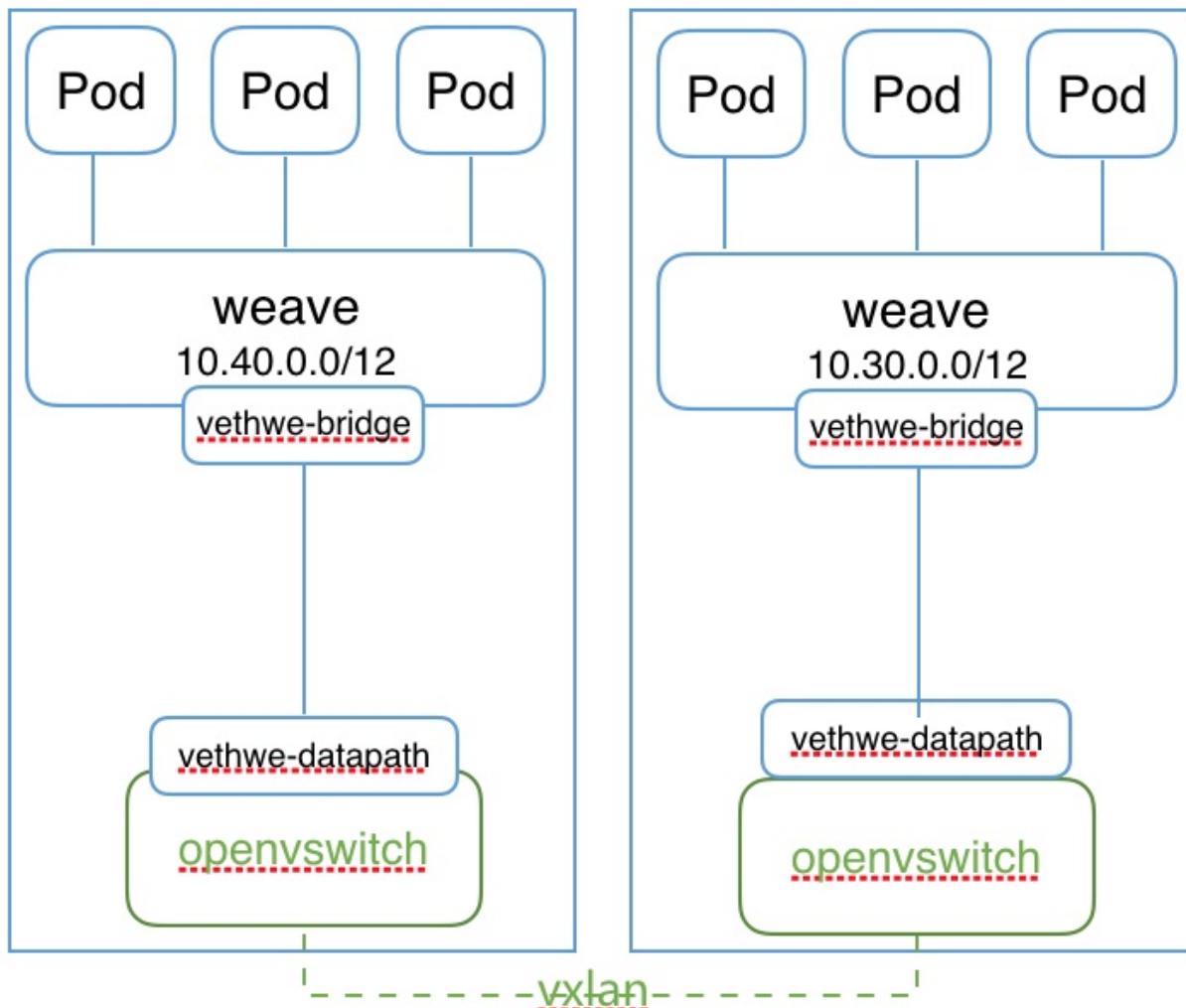
```
kubectl apply -n kube-system -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"
```

这会在所有Node上启动Weave插件以及Network policy controller：

```
$ ps -ef | grep weave | grep -v grep
root      25147 25131  0 16:22 ?          00:00:00 /bin/sh /home/weave/launch.sh
root      25204 25147  0 16:22 ?          00:00:00 /home/weave/weaver --port=6783 --datapath=datapath --host-root=/host --http-addr=127.0.0.1:6784 --status-addr=0.0.0.0:6782 --docker-api= --no-dns --db-prefix=/weavedb/weave-net --ipalloc-range=10.32.0.0/12 --nickname=ubuntu-0 --ipalloc-init consensus=2 --conn-limit=30 --expect-npc 10.146.0.2 10.146.0.3
root      25669 25654  0 16:22 ?          00:00:00 /usr/bin/weave-npc
```

这样，容器网络为

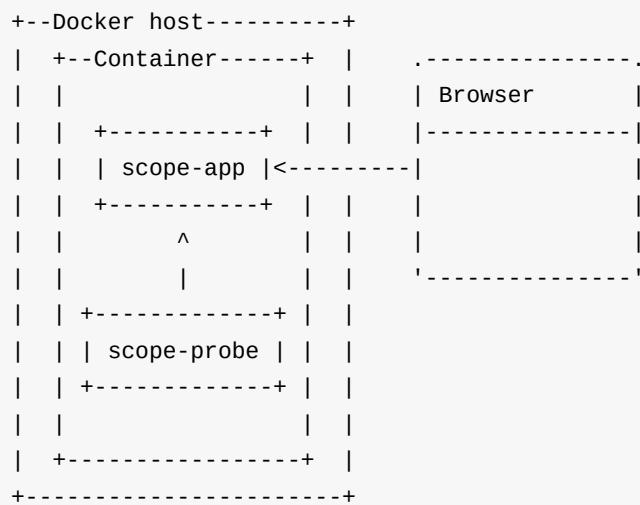
- 所有容器都连接到weave网桥
- weave网桥通过veth pair连到内核的openvswitch模块
- 跨主机容器通过openvswitch vxlan通信
- policy controller通过配置iptables规则为容器设置网络策略



## Weave Scope

Weave Scope是一个容器监控和故障排查工具，可以方便的生成整个集群的拓扑并智能分组（Automatic Topologies and Intelligent Grouping）。

Weave Scope主要由scope-probe和scope-app组成



## 优点

- 去中心化
- 故障自动恢复
- 加密通信
- Multicast networking

## 缺点

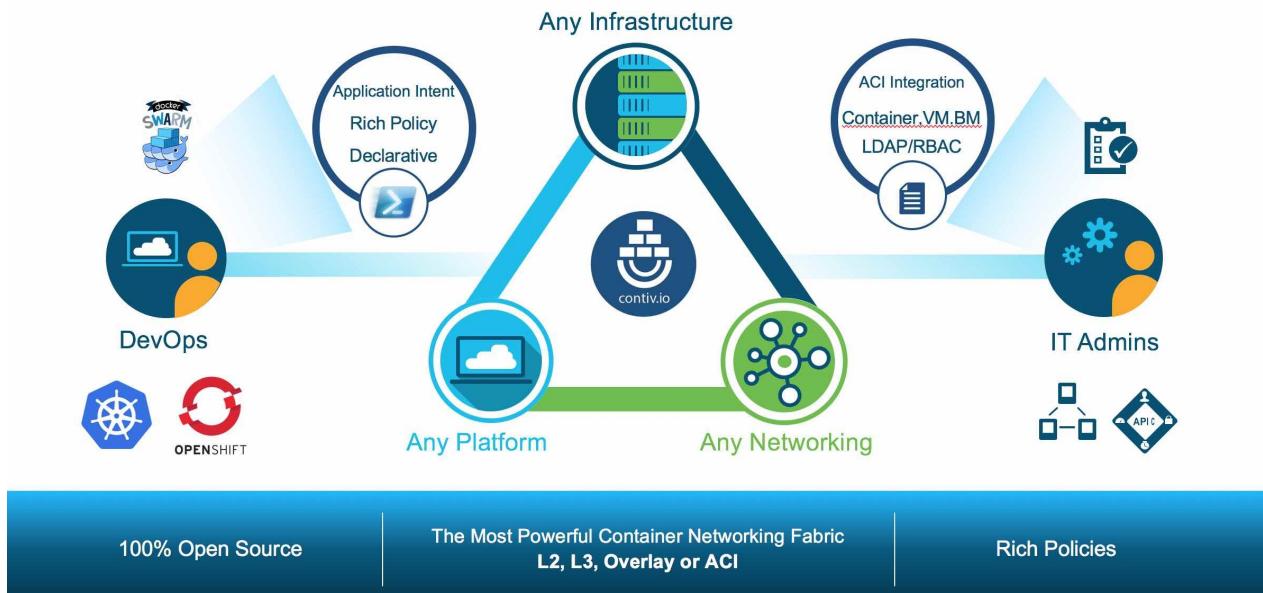
- UDP模式性能损失较大

### 参考文档

- <https://github.com/weaveworks/weave>
- <https://www.weave.works/products/weave-net/>
- <https://github.com/weaveworks/scope>
- <https://www.weave.works/guides/monitor-docker-containers/>
- <http://www.sdnlab.com/17141.html>

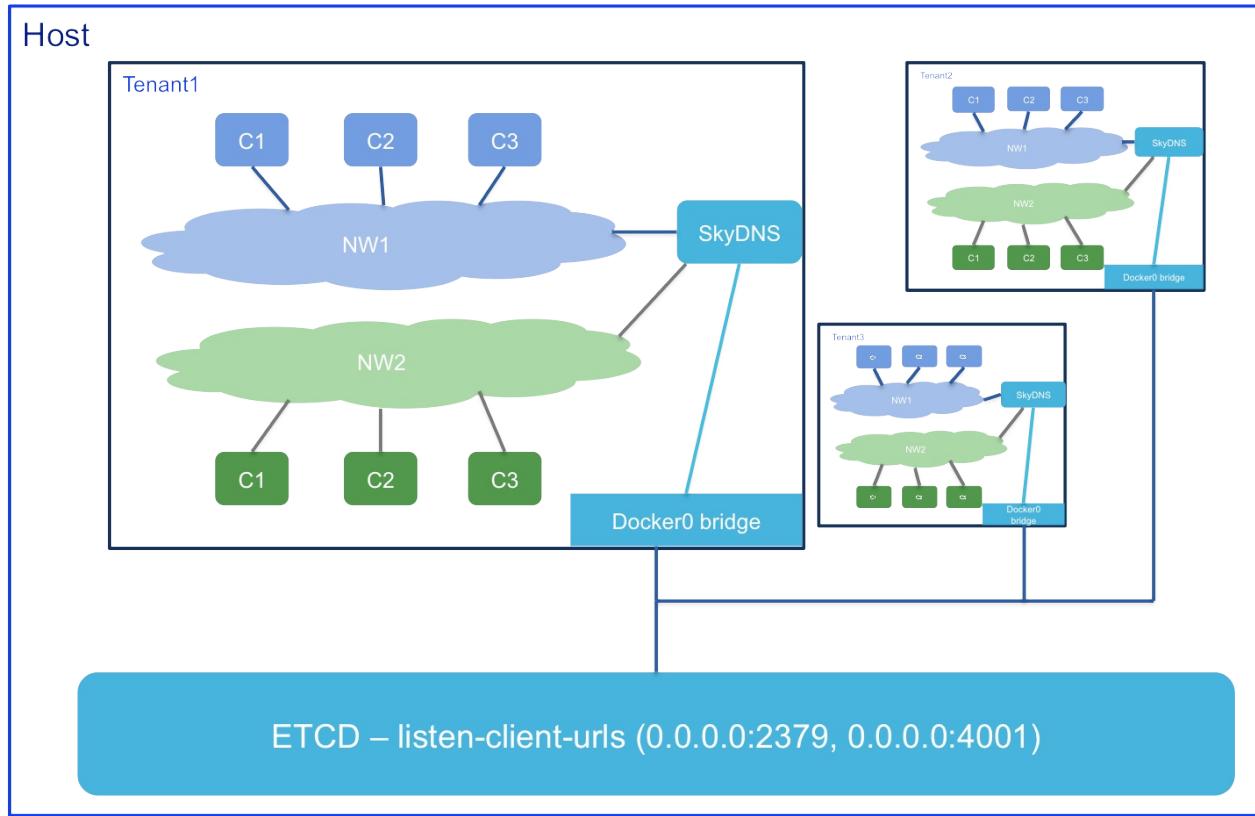
# Contiv

Contiv是思科开源的容器网络方案，是一个用于跨虚拟机、裸机、公有云或私有云的异构容器部署的开源容器网络架构，并与主流容器编排系统集成。Contiv最主要的优势是直接提供了多租户网络，并支持L2(VLAN), L3(BGP), Overlay (VXLAN)以及思科自家的ACI。

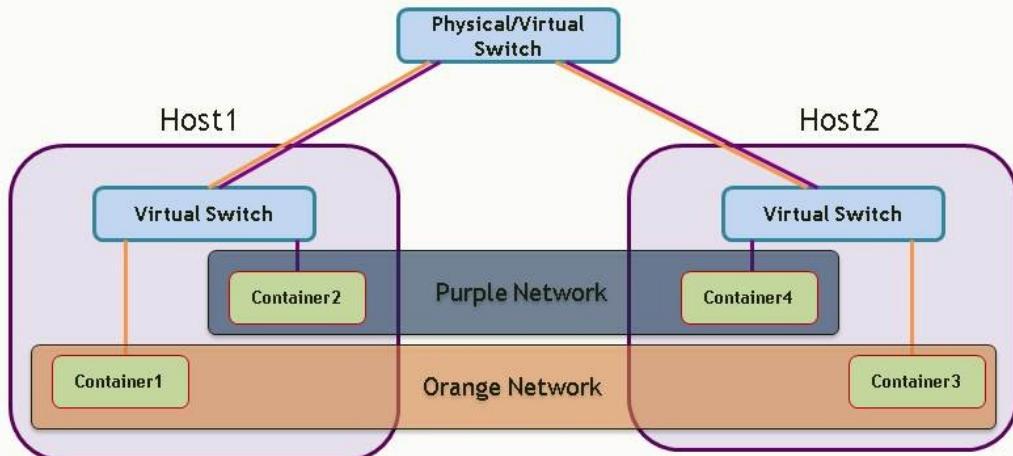


## 主要特征

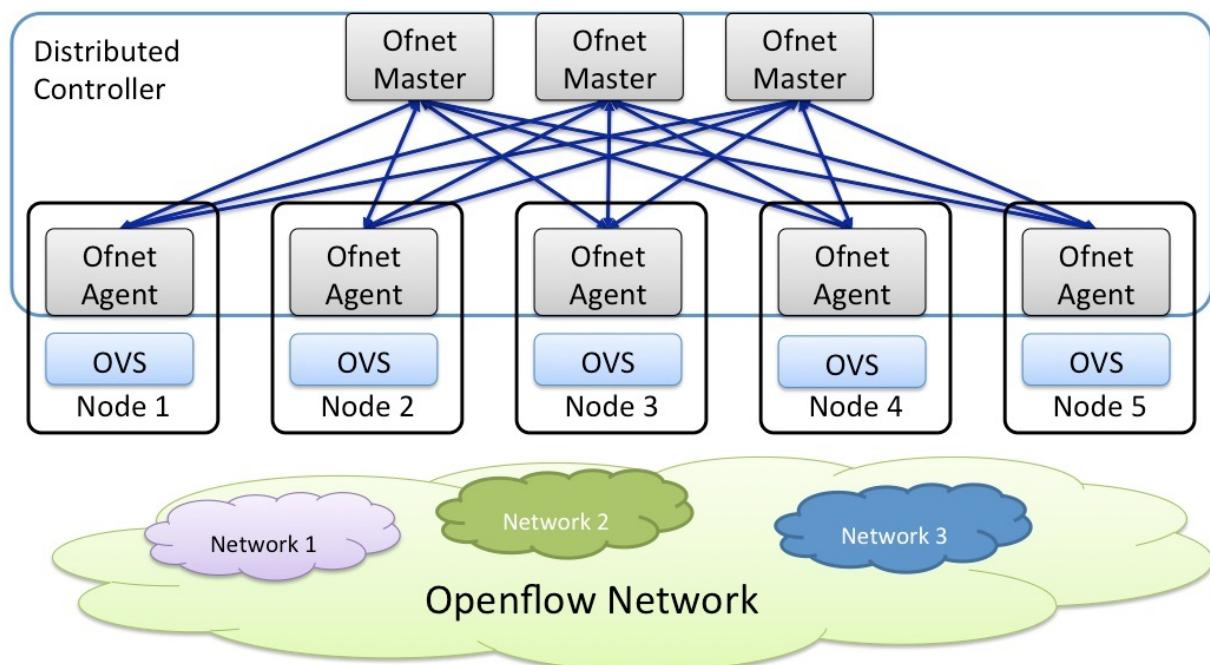
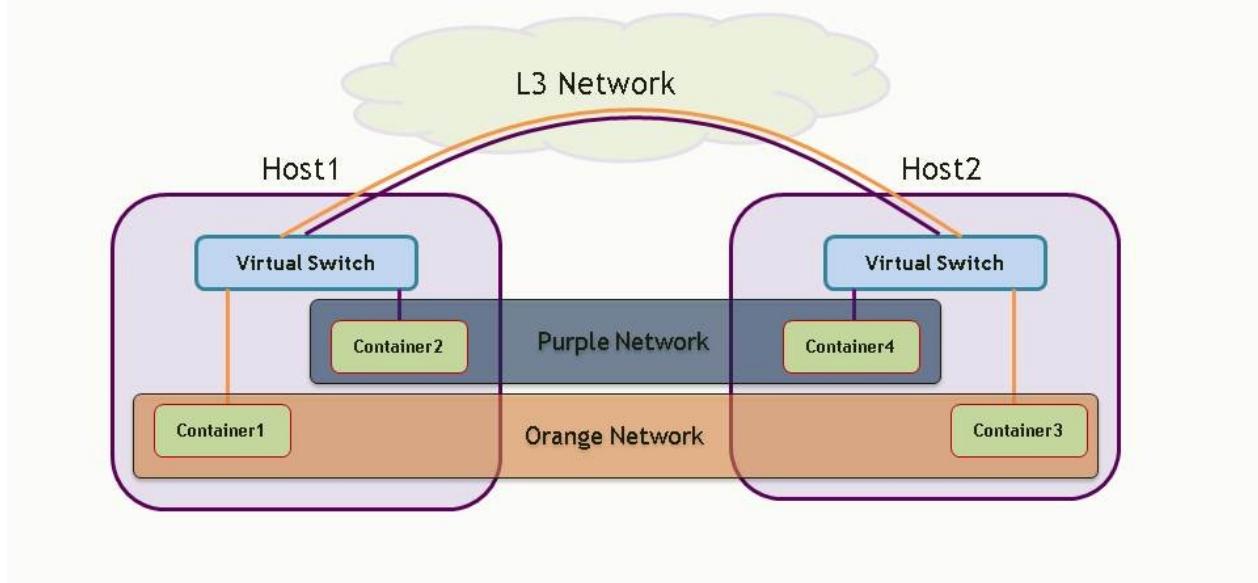
- 原生的Tenant支持，一个Tenant就是一个virtual routing and forwarding (VRF)
- 两种网络模式
  - L2 VLAN Bridged
  - Routed network, e.g. vxlan, BGP, ACI
- Network Policy，如Bandwidth, Isolation等



## VLAN Network



## VXLAN Network



## Kubernetes 集成

Ansible 部署见<https://github.com/kubernetes/contrib/tree/master/ansible/roles/contiv>。

```
export VERSION=1.0.0-beta.3
curl -L -o https://github.com/contiv/install/releases/download/$VERSION/contiv-$VERSION.tgz
tar xf contiv-$VERSION.tgz
cd ~/contiv/contiv-$VERSION/install/k8s
netctl --netmaster http://$netmaster:9999 global set --fwd-mode routing

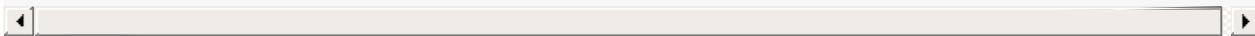
cd ~/contiv/contiv-$VERSION
install/k8s/install.sh -n 10.87.49.77 -v b -w routing

# check contiv pods
export NETMASTER=http://10.87.49.77:9999
netctl global info

# create a network
# netctl network create --encap=vlan --pkt-tag=3280 --subnet=10.100.100.215-10.100.100.220/27 --gateway=10.100.100.193 vlan3280
netctl net create -t default --subnet=20.1.1.0/24 default-net

# create BGP connections to each of the nodes
netctl bgp create devstack-77 --router-ip="30.30.30.77/24" --as="65000" --neighbor-as="65000" --neighbor="30.30.30.2"
netctl bgp create devstack-78 --router-ip="30.30.30.78/24" --as="65000" --neighbor-as="65000" --neighbor="30.30.30.2"
netctl bgp create devstack-71 --router-ip="30.30.30.79/24" --as="65000" --neighbor-as="65000" --neighbor="30.30.30.2"

# then create pod with label "io.contiv.network"
```



## 参考文档

- <http://contiv.github.io/>
- <https://github.com/contiv/netplugin>
- <http://blogs.cisco.com/cloud/introducing-contiv-1-0>
- [Kubernetes and Contiv on Bare-Metal with L3/BGP](#)

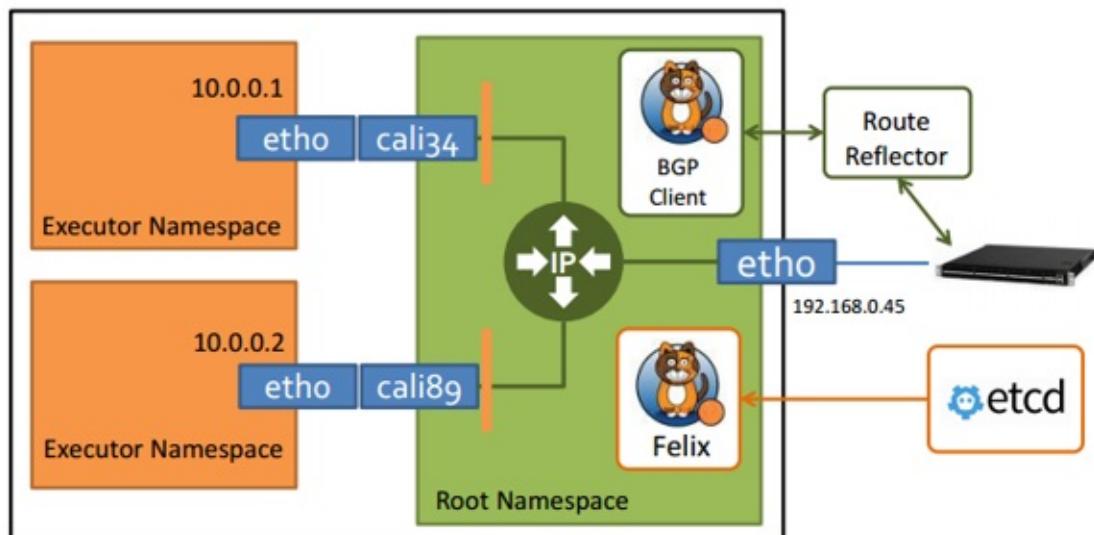
# Calico

Calico 是一个纯三层的数据中心网络方案（不需要Overlay），并且与OpenStack、Kubernetes、AWS、GCE等IaaS和容器平台都有良好的集成。

Calico在每一个计算节点利用Linux Kernel实现了一个高效的vRouter来负责数据转发，而每个vRouter通过BGP协议负责把自己上运行的workload的路由信息像整个Calico网络内传播——小规模部署可以直接互联，大规模下可通过指定的BGP route reflector来完成。这样保证最终所有的workload之间的数据流量都是通过IP路由的方式完成互联的。Calico节点组网可以直接利用数据中心的网络结构（无论是L2或者L3），不需要额外的NAT，隧道或者Overlay Network。

此外，Calico基于iptables还提供了丰富而灵活的网络Policy，保证通过各个节点上的ACLs来提供Workload的多租户隔离、安全组以及其他可达性限制等功能。

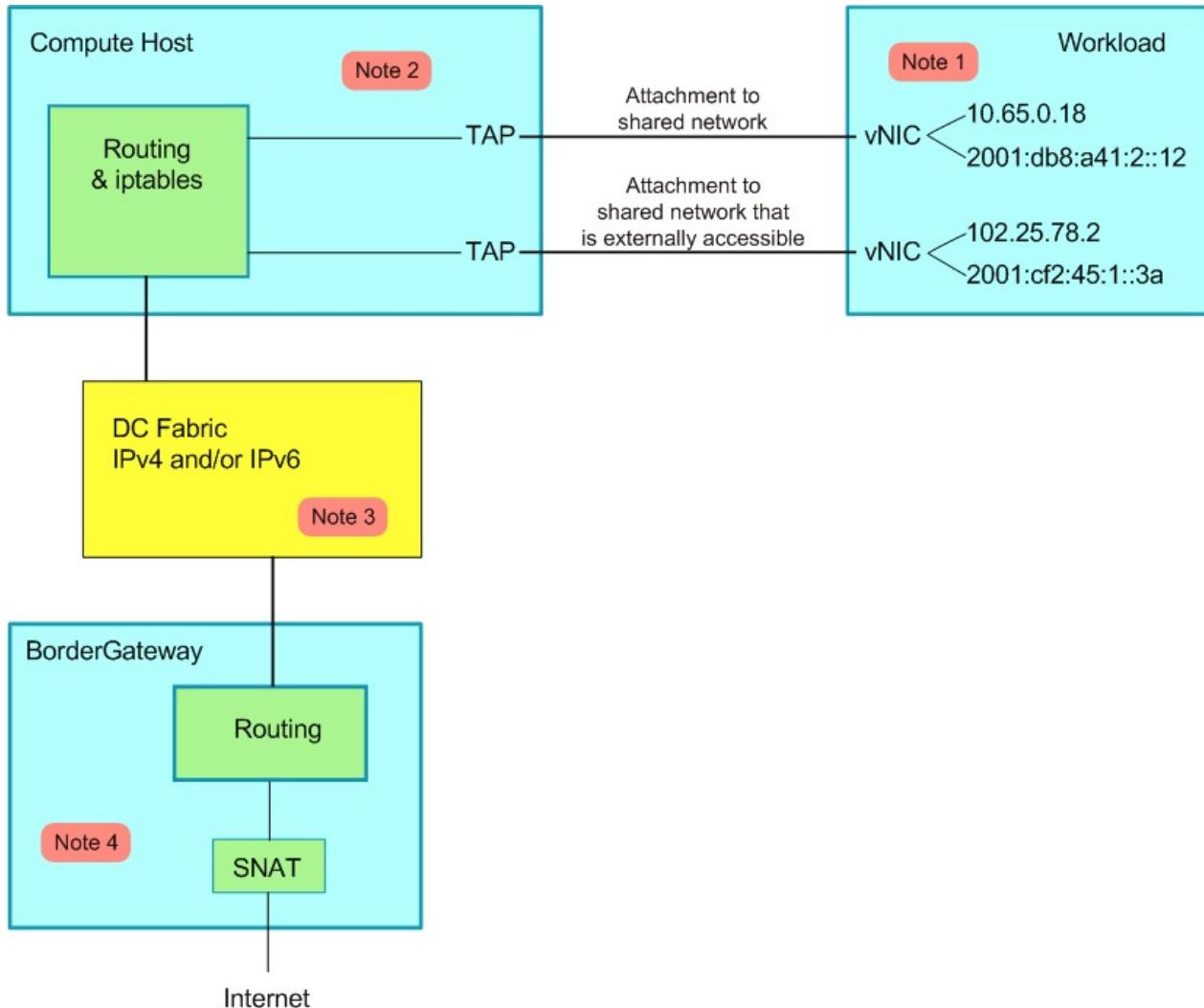
## Calico 架构



Calico主要由Felix、etcd、BGP client以及BGP Route Reflector组成

1. Felix，Calico Agent，跑在每台需要运行Workload的节点上，主要负责配置路由及ACLs等信息来确保Endpoint的连通状态；
2. etcd，分布式键值存储，主要负责网络元数据一致性，确保Calico网络状态的准确性；
3. BGP Client (BIRD)，主要负责把Felix写入Kernel的路由信息分发到当前Calico网络，确保Workload间的通信的有效性；
4. BGP Route Reflector (BIRD)，大规模部署时使用，摒弃所有节点互联的 mesh 模式，通过一个或者多个BGP Route Reflector来完成集中式的路由分发。

5. calico/calico-ipam，主要用作Kubernetes的CNI插件



## IP-in-IP

Calico控制平面的设计要求物理网络得是L2 Fabric，这样vRouter间都是直接可达的，路由不需要把物理设备当做下一跳。为了支持L3 Fabric，Calico推出了IPinIP的选项。

## Calico CNI

见<https://github.com/projectcalico/cni-plugin>。

## Calico CNM

Calico通过Pool和Profile的方式实现了docker CNM网络：

1. Pool，定义可用于Docker Network的IP资源范围，比如：10.0.0.0/8或者

192.168.0.0/16；

2. Profile，定义Docker Network Policy的集合，由tags和rules组成；每个Profile默认拥有一个和Profile名字相同的Tag，每个Profile可以有多个Tag，以List形式保存。

具体实现见<https://github.com/projectcalico/libnetwork-plugin>，而使用方法可以参考<http://docs.projectcalico.org/v2.1/getting-started/docker/>。

## Calico Kubernetes

见<http://docs.projectcalico.org/v2.1/getting-started/kubernetes/>.

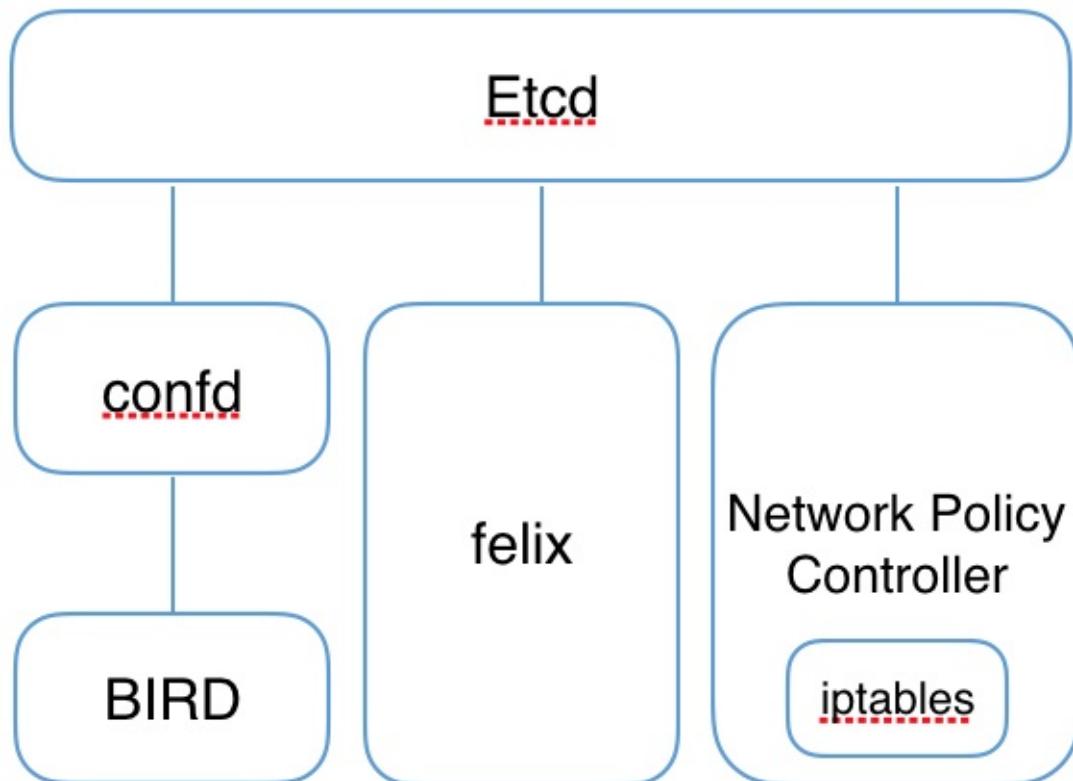
For Kubeadm 1.5 with Kubernetes 1.5.x:

```
kubectl apply -f http://docs.projectcalico.org/v2.3/getting-started/kubernetes/installation/hosted/kubeadm/1.5/calico.yaml
```

For Kubeadm 1.6 with Kubernetes 1.6.x:

```
kubectl apply -f http://docs.projectcalico.org/v2.3/getting-started/kubernetes/installation/hosted/kubeadm/1.6/calico.yaml
```

这会在Pod中启动Calico-etcd，在所有Node上启动bird6、felix以及confd，并配置CNI网络为calico插件：

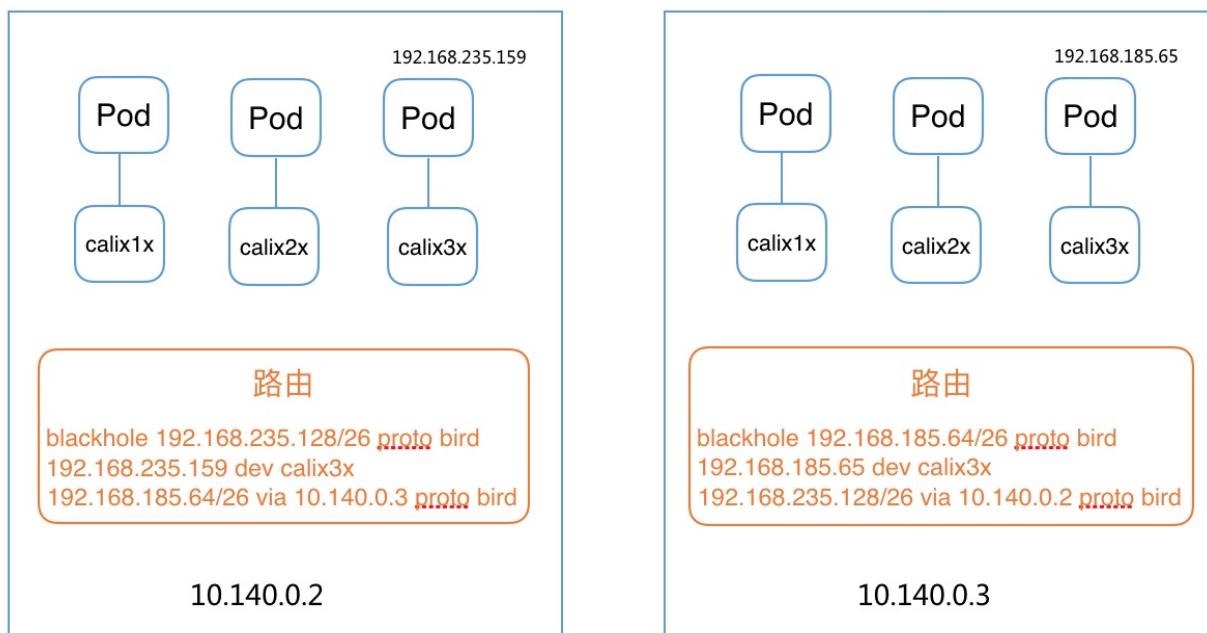


```

# Calico相关进程
$ ps -ef | grep calico | grep -v grep
root      9012  8995  0 14:51 ?        00:00:00 /bin/sh -c /usr/local/bin/etcd --name=calico --data-dir=/var/etc/calico-data --advertise-client-urls=http://$CALICO_ETCD_IP:6666 --listen-client-urls=http://0.0.0.0:6666 --listen-peer-urls=http://0.0.0.0:6667
root      9038  9012  0 14:51 ?        00:00:01 /usr/local/bin/etcd --name=calico --data-dir=/var/etc/calico-data --advertise-client-urls=http://10.146.0.2:6666 --listen-client-urls=http://0.0.0.0:6666 --listen-peer-urls=http://0.0.0.0:6667
root      9326  9325  0 14:51 ?        00:00:00 bird6 -R -s /var/run/calico/bird6.ctl -d -c /etc/calico/confd/config/bird6.cfg
root      9327  9322  0 14:51 ?        00:00:00 confd -confdir=/etc/calico/confd -interval=5 -watch --log-level=debug -node=http://10.96.232.136:6666 -client-key= -client-cert= -client-ca-keys=
root      9328  9324  0 14:51 ?        00:00:00 bird -R -s /var/run/calico/bird.ctl -d -c /etc/calico/confd/config/bird.cfg
root      9329  9323  1 14:51 ?        00:00:04 calico-felix
  
```

```
# CNI网络插件配置
$ cat /etc/cni/net.d/10-calico.conf
{
    "name": "k8s-pod-network",
    "cniVersion": "0.1.0",
    "type": "calico",
    "etcd_endpoints": "http://10.96.232.136:6666",
    "log_level": "info",
    "ipam": {
        "type": "calico-ipam"
    },
    "policy": {
        "type": "k8s",
        "k8s_api_root": "https://10.96.0.1:443",
        "k8s_auth_token": "<token>"
    },
    "kubernetes": {
        "kubeconfig": "/etc/cni/net.d/calico-kubeconfig"
    }
}

$ cat /etc/cni/net.d/calico-kubeconfig
# Kubeconfig file for Calico CNI plugin.
apiVersion: v1
kind: Config
clusters:
- name: local
  cluster:
    insecure-skip-tls-verify: true
users:
- name: calico
contexts:
- name: calico-context
  context:
    cluster: local
    user: calico
current-context: calico-context
```



## Calico的不足

- 既然是三层实现，当然不支持VRF
- 不支持多租户网络的隔离功能，在多租户场景下会有网络安全问题
- Calico控制平面的设计要求物理网络得是L2 Fabric，这样vRouter间都是直接可达的

### 参考文档

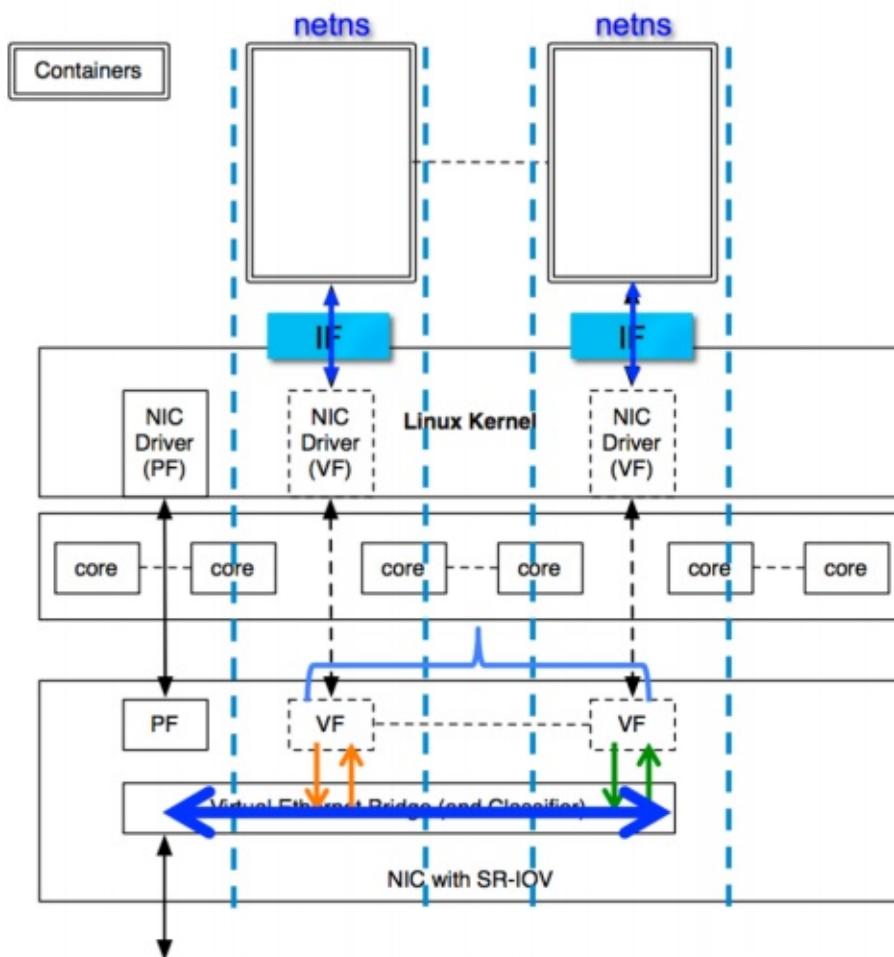
- <https://xuxinkun.github.io/2016/07/22/cni-cnm/>
- <https://www.projectcalico.org/>
- <http://blog.dataman-inc.com/shurenyun-docker-133/>

# SR-IOV

SR-IOV 技术是一种基于硬件的虚拟化解决方案，可提高性能和可伸缩性

SR-IOV 标准允许在虚拟机之间高效共享 PCIe (Peripheral Component Interconnect Express, 快速外设组件互连) 设备，并且它是在硬件中实现的，可以获得能够与本机性能媲美的 I/O 性能。SR-IOV 规范定义了新的标准，根据该标准，创建的新设备可允许将虚拟机直接连接到 I/O 设备 (SR-IOV 规范由 PCI-SIG 在 <http://www.pcisig.com> 上进行定义和维护)。单个 I/O 资源可由许多虚拟机共享。共享的设备将提供专用的资源，并且还使用共享的通用资源。这样，每个虚拟机都可访问唯一的资源。因此，启用了 SR-IOV 并且具有适当的硬件和 OS 支持的 PCIe 设备 (例如以太网端口) 可以显示为多个单独的物理设备，每个都具有自己的 PCIe 配置空间。

SR-IOV主要用于虚拟化中，当然也可以用于容器。



## SR-IOV配置

```
modprobe ixgbevf
lspci -Dvmm|grep -B 1 -A 4 Ethernet
echo 2 > /sys/bus/pci/devices/0000:82:00.0/sriov_numvfs
# check ifconfig -a. You should see a number of new interfaces created, starting with
"eth", e.g. eth4
```

## docker sriov plugin

Intel给docker写了一个SR-IOV network plugin，源码位于  
<https://github.com/clearcontainers/sriov>，同时支持runc和clearcontainer。

## CNI插件

Intel维护了一个SR-IOV的CNI插件，fork自[hustcat/sriov-cni](https://github.com/hustcat/sriov-cni)，并扩展了DPDK的支持。

项目主页见<https://github.com/Intel-Corp/sriov-cni>。

## 优点

- 性能好
- 不占用计算资源

## 缺点

- VF数量有限
- 硬件绑定，不支持容器迁移

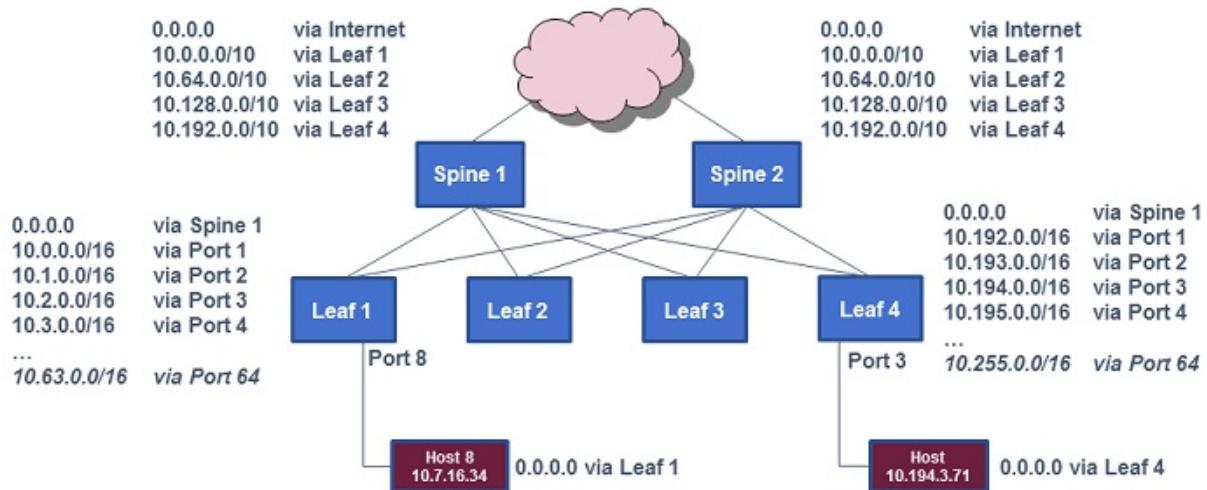
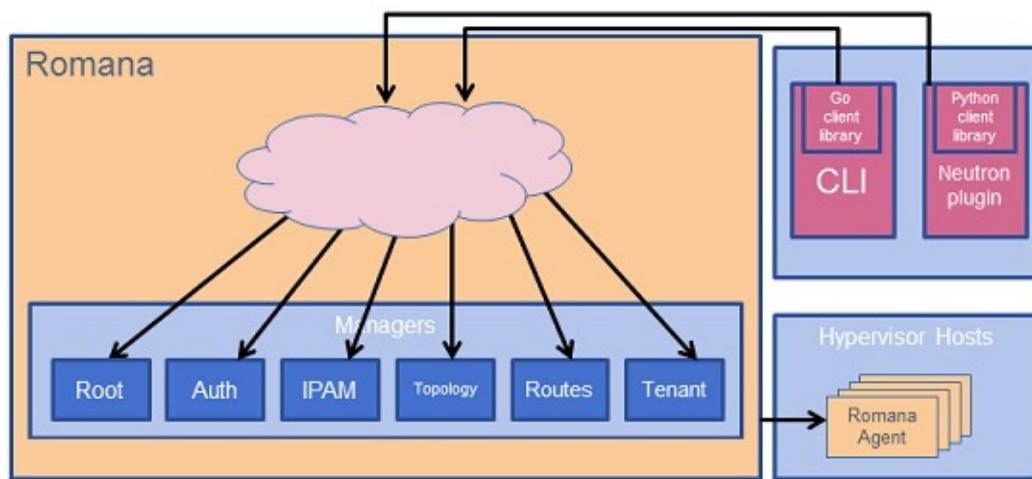
## 参考文档

- <http://blog.scottlowe.org/2009/12/02/what-is-sr-iov/>
- <https://github.com/clearcontainers/sriov>
- <https://software.intel.com/en-us/articles/single-root-inputoutput-virtualization-sr-iov-with-linux-containers>
- <http://jason.digitalinertia.net/exposing-docker-containers-with-sr-iov/>

# Romana

Romana是Panic Networks在2016年提出的开源项目，旨在解决Overlay方案给网络带来的开销。

## 工作原理



Spine splits full 10/8 CIDR across 4 Leaf devices (four /10 networks)  
 Leaf allocates /10 network across 64 ports assigning a /16 address block to each  
 Host on Leaf 1, Port 8 must get address within 10.7.0.0/16  
 Host on Leaf 4, Port 3 must get address within 10.194.0.0/16

- layer 3 networking，消除overlay带来的开销
- 基于iptables ACL的网络隔离
- 基于hierarchy CIDR管理Host/Tenant/Segment ID

Bit location	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Field	10/8 Net Mask								Host ID Bits (8)								Tenant ID Bits (8)								Segment ID and IID							
Capacity	0	0	0	0	1	0	1	0	Up to 255 Hosts								Up to 255 Tenants								255 Endpoints for each Tenant							

Example:

	Bits	Length	Location	Purpose
10/8 Network	8	8	1-8	10/8 Network
Hosts	8	16	9-16	Up to 255 Hosts
Tenants	8	24	17-24	Up to 255 Tenants
Segments	4	28	25-28	Up to 16 Segments per Tenant
Endpoints	4	32	29-32	Up to 16 Endpoints per Segment

Host 1	ID	CIDR or IP
Physical Addr		192.168.0.10
Host	1	10.1/16
Tenant	1	10.1.1/24
Segment	1	10.1.1.16/28
Pod 1	11	10.1.1.27
Pod 2	14	10.1.1.40
Tenant	2	10.1.2/24
Segment	1	10.1.2.16/28
Pod 1	4	10.1.2.20
Pod 2	8	10.1.2.24

Host 2	ID	CIDR or IP
Physical Addr		192.168.0.11
Host	2	10.2/16
Tenant	1	10.2.1/24
Segment	1	10.2.1.16/28
Pod 1	4	10.2.1.20
Pod 2	5	10.2.1.21
Tenant	1	10.2.1/24
Segment	2	10.2.1.32/28
Pod 1	9	10.2.1.41
Pod 2	12	10.2.1.44

Host 3	ID	CIDR or IP
Physical Addr		192.168.0.12
Host	3	10.3/16
Tenant	1	10.3.1/24
Segment	1	10.3.1.16/28
Pod 1	4	10.3.1.20
Pod 2	5	10.3.1.21
Tenant	2	10.3.2/24
Segment	1	10.3.2.32/28
Pod 1	9	10.3.2.25
Pod 2	12	10.3.2.28

## 优点

- 纯三层网络，性能好

## 缺点

- 基于IP管理租户，有规模上的限制
- 物理设备变更或地址规划变更麻烦

## 参考文档

- <http://romana.io/>
- <https://github.com/romana/romana>

# OpenContrail

OpenContrail是Juniper推出的开源网络虚拟化平台，其商业版本为Contrail。

## 架构

OpenContrail主要由控制器和vRouter组成：

- 控制器提供虚拟网络的配置、控制和分析功能
- vRouter提供分布式路由，负责虚拟路由器、虚拟网络的建立以及数据转发

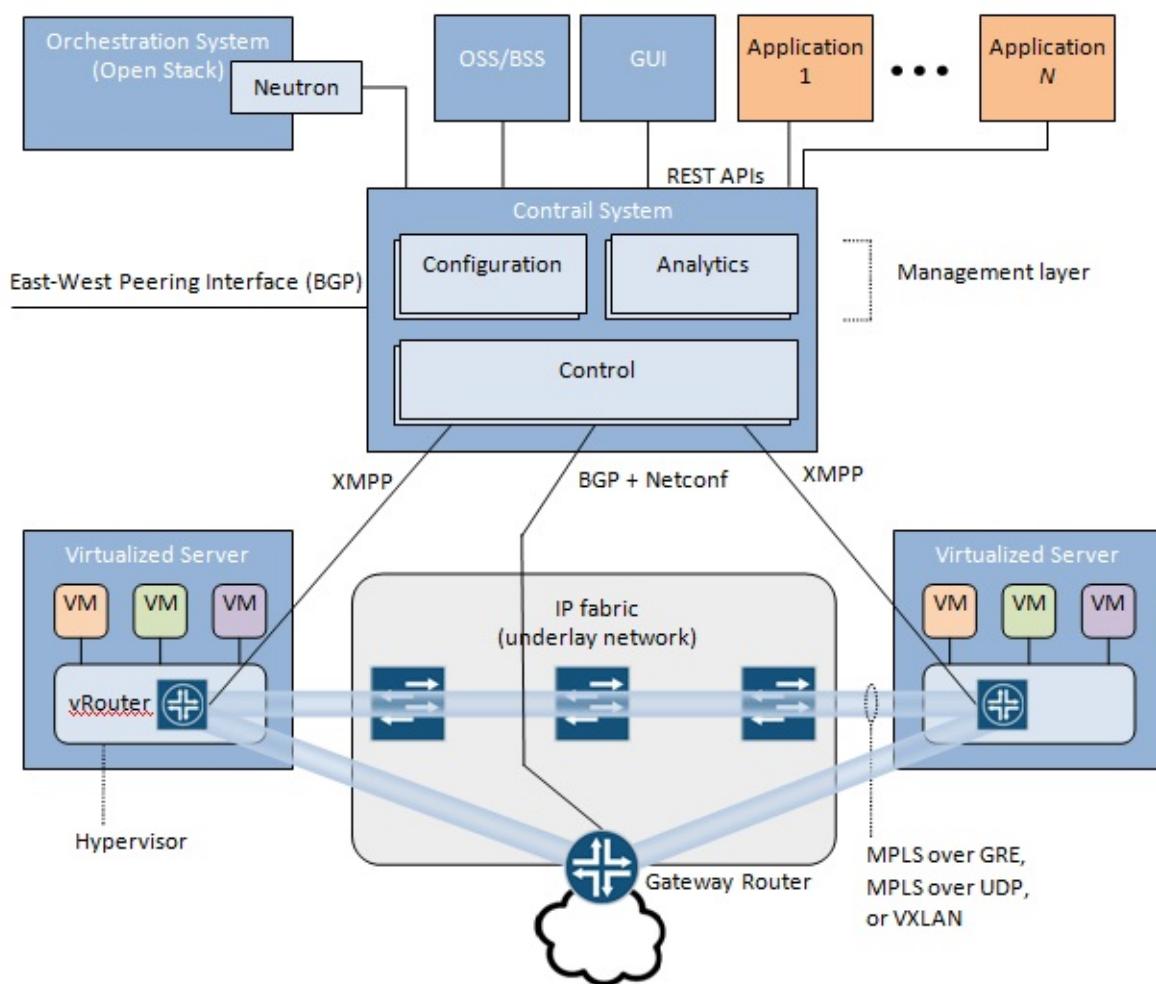
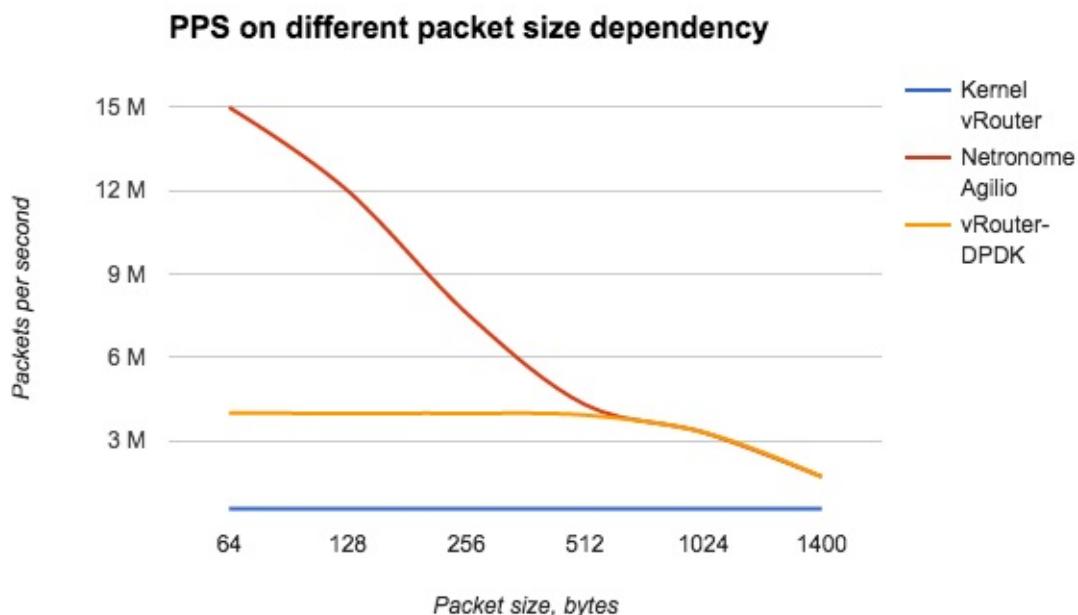


Figure 1: OpenContrail System Overview

vRouter支持三种模式

- Kernel vRouter：类似于ovs内核模块
- DPDK vRouter：类似于ovs-dpdk

- Netronome Agilio Solution (商业产品)：支持DPDK, SR-IOV and Express Virtio (XVIO)



## 参考文档

- <http://www.opencontrail.org/opencontrail-architecture-documentation/>
- <http://www.opencontrail.org/network-virtualization-architecture-deep-dive/>

# CNI Plugin Chains

CNI还支持Plugin Chains，即指定一个插件列表，由Runtime依次执行每个插件。这对支持portmapping、vm等非常有帮助。

## Network Configuration Lists

CNI SPEC支持指定网络配置列表，包含多个网络插件，由Runtime依次执行。注意

- ADD操作，按顺序依次调用每个插件；而DEL操作调用顺序相反
- ADD操作，除最后一个插件，前面每个插件需要增加 `prevResult` 传递给其后的插件
- 第一个插件必须要包含ipam插件

### 示例

下面的例子展示了bridge+portmap插件的用法。

首先，配置CNI网络使用bridge+portmap插件：

```
# cat /root/mynet.conflist
{
  "name": "mynet",
  "cniVersion": "0.3.0",
  "plugins": [
    {
      "type": "bridge",
      "bridge": "mynet",
      "ipMasq": true,
      "isGateway": true,
      "ipam": {
        "type": "host-local",
        "subnet": "10.244.10.0/24",
        "routes": [
          { "dst": "0.0.0.0/0" }
        ]
      }
    },
    {
      "type": "portmap",
      "capabilities": {"portMappings": true}
    }
  ]
}
```

然后通过 `CAP_ARGS` 设置端口映射参数：

```
# export CAP_ARGS='{
    "portMappings": [
        {
            "hostPort":      9090,
            "containerPort": 80,
            "protocol":      "tcp",
            "hostIP":        "127.0.0.1"
        }
    ]
}'
```

测试添加网络接口：

```
# ip netns add test
# CNI_PATH=/opt/cni/bin NETCONFPATH=/root ./cnitool add mynet /var/run/netns/test
{
    "interfaces": [
        {
            "name": "mynet",
            "mac": "0a:58:0a:f4:0a:01"
        },
        {
            "name": "veth2cfb1d64",
            "mac": "4a:dc:1f:b7:56:b1"
        },
        {
            "name": "eth0",
            "mac": "0a:58:0a:f4:0a:07",
            "sandbox": "/var/run/netns/test"
        }
    ],
    "ips": [
        {
            "version": "4",
            "interface": 2,
            "address": "10.244.10.7/24",
            "gateway": "10.244.10.1"
        }
    ],
    "routes": [
        {
            "dst": "0.0.0.0/0"
        }
    ],
    "dns": {}
}
```

可以从 `iptables` 规则中看到添加的规则：

```
# iptables-save | grep 10.244.10.7
-A CNI-DN-be1eedf7a76853f303ebd -d 127.0.0.1/32 -p tcp -m tcp --dport 9090 -j DNAT --to-destination 10.244.10.7:80
-A CNI-SN-be1eedf7a76853f303ebd -s 127.0.0.1/32 -d 10.244.10.7/32 -p tcp -m tcp --dport 80 -j MASQUERADE
```

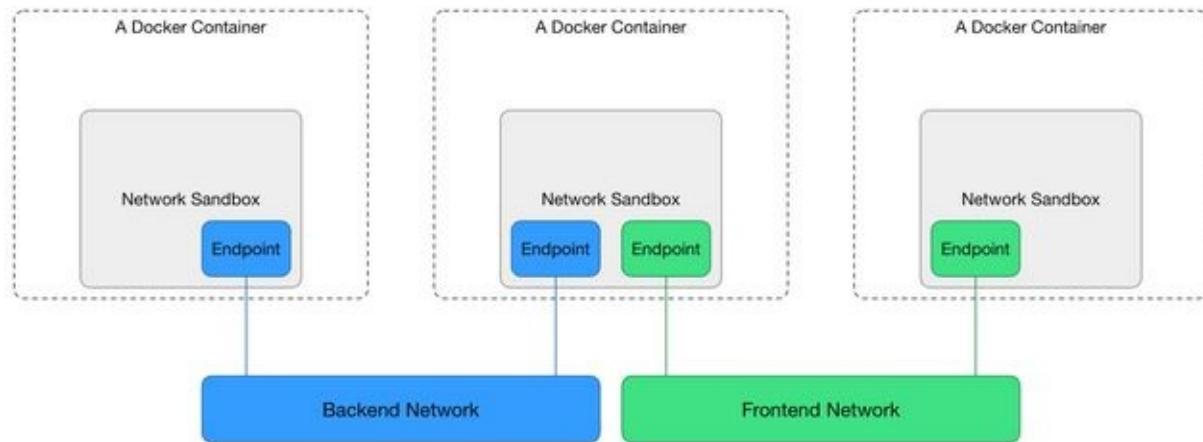
最后，清理网络接口：

```
# CNI_PATH=/opt/cni/bin NETCONFPATH=/root ./cni tool del mynet /var/run/netns/test
```

# CNM (Container Network Model)

[Container network model \(CNM\)](#)是Docker的网络模型，主要由Sandbox、Network以及Endpoint组成。

- **Sandbox**：一个Sandbox对应一个容器的网络栈，能够对该容器的interface、route、dns等参数进行管理。一个Sandbox中可以有多个Endpoint，这些Endpoint可以属于不同的Network。Sandbox的实现可以为linux network namespace、FreeBSD Jail或其他类似的机制。
- **Endpoint**：Sandbox通过Endpoint接入Network，一个Endpoint只能属于一个Network，but may only belong to one Sandbox（这句翻译不好）。Endpoint的实现可以是veth pair、Open vSwitch internal port或者其他类似的设备。
- **Network**：一个Network由一组Endpoint组成，这些Endpoint彼此间可以直接进行通信，不同的Network间Endpoint的通信彼此隔离。Network的实现可以是linux bridge、Open vSwitch等。



Libnetwork对于CNM的实现包括以下5类对象：

- **NetworkController**：每创建一个Network对象时，就会相应地生成一个NetworkController对象，NetworkController对象将Network对象的API暴露给用户，以便用户对libnetwork进行调用，然后驱动特定的Driver对象实现Network对象的功能。NetworkController允许用户绑定Network对象所使用的Driver对象。NetworkController对象可以看做是Network对象的分布式SDN控制器。
- **Network**：Network对象是CNM Network的一种实现。NetworkController对象通过提供API对Network对象进行创建和管理。NetworkController对象需要操作Network对象的时候，Network对象所对应的Driver对象会得到通知。一个Network对象能够包含多个Endpoint对象，一个Network对象中包含的各个Endpoint对象间可以通过Driver完成通信，这种通信支持可以是同一主机的，也可以是跨主机的。不同Network对象中的Endpoint对象间彼此隔离。

- Driver : Driver对象真正实现Network功能（包括通信和管理），它并不直接暴露API给用户。Libnetwork支持多种Driver，其中包括内置的bridge，host，container和overlay，也对remote driver（即第三方，或用户自定义的网络驱动）进行了支持。
- Endpoint : Endpoint对象是CNM Endpoint的一种实现。容器通过Endpoint对象接入Network，并通过Endpoint对象与其它容器进行通信。一个Endpoint对象只能属于一个Network对象，Network对象的API提供了对于Endpoint对象的创建与管理。
- Sandbox : Sandbox对象是CNM Sandbox的一种实现。Sandbox对象代表了一个容器的网络栈，拥有IP地址，MAC地址，routes，DNS等网络资源。一个Sandbox对象中可以有多个Endpoint对象，这些Endpoint对象可以属于不同的Network对象，Endpoint对象使用Sandbox对象中的网络资源与外界进行通信。Sandbox对象的创建发生在Endpoint对象的创建后，（Endpoint对象所属的）Network对象所绑定的Driver对象为该Sandbox对象分配网络资源并返回给libnetwork，然后libnetwork使用特定的机制（如linux netns）去配置Sandbox对象中对应的网络资源。

## Bridge

Bridge是docker默认的网络插件。在不指定network插件时，容器会通过veth pair接到宿主机的docker0网桥上，并通过iptables实现端口映射和外网访问。

## Overlay

Overlay是docker swarm默认的网络插件，基于linux内核vxlan实现跨主机的容器网络通信。

## MACVLAN

MACVLAN可以从一个主机接口虚拟出多个macvtap，且每个macvtap设备都拥有不同的mac地址（对应不同的linux字符设备）。

MACVLAN支持四种模式

- bridge模式：数据可以在同一master设备的子设备之间转发
- vepa模式：VEPA 模式是对 802.1Qbg 标准中的 VEPA 机制的软件实现，MACVTAP 设备简单的将数据转发到master设备中，完成数据汇聚功能，通常需要外部交换机支持 Hairpin 模式才能正常工作
- private模式：Private 模式和 VEPA 模式类似，区别是子 MACVTAP 之间相互隔离
- passthrough模式：内核的 MACVLAN 数据处理逻辑被跳过，硬件决定数据如何处理，从而释放了 Host CPU 资源

使用方法可以参考<https://docs.docker.com/engine/userguide/networking/get-started-macvlan/>。

## IPVLAN

IPVLAN 和 MACVLAN 类似，都是从一个主机接口虚拟出多个虚拟网络接口。一个重要的区别就是所有的虚拟接口都有相同的 mac 地址，而拥有不同的 ip 地址。

IPVLAN 支持两种模式：

- L2 模式：此时跟 macvlan bridge 模式工作原理很相似，父接口作为交换机来转发子接口的数据。同一个网络的子接口可以通过父接口来转发数据，而如果想发送到其他网络，报文则会通过父接口的路由转发出去。
- L3 模式：此时 ipvlan 有点像路由器的功能，它在各个虚拟网络和主机网络之间进行不同网络报文的路由转发工作。只要父接口相同，即使虚拟机/容器不在同一个网络，也可以互相 ping 通对方，因为 ipvlan 会在中间做报文的转发工作。注意 L3 模式下的虚拟接口不会接收到多播或者广播的报文（这个模式下，所有的网络都会发送给父接口，所有的 ARP 过程或者其他多播报文都是在底层的父接口完成的）。另外外部网络默认情况下是不知道 ipvlan 虚拟出来的网络的，如果不在外部路由器上配置好对应的路由规则，ipvlan 的网络是不能被外部直接访问的。

## OVS

<https://github.com/openvswitch/ovs/blob/master/utilities/ovs-docker>

[http://containertutorials.com/network/ovs\\_docker.html](http://containertutorials.com/network/ovs_docker.html)

## OVS-DPDK

<https://github.com/clearcontainers/ovsdpdk>

## OVN

OVS 官方实现了一个 OVN 的 CNM 插件，可以方便的通过 overlay 或者 underlay（需要配合 OpenStack Neutron）来给容器配置网络。具体使用可以参考[这里](#)。

## Calico

[Calico](#) 是一个基于BGP的纯三层的数据中心网络方案（不需要Overlay），并且与OpenStack、Kubernetes、AWS、GCE等IaaS和容器平台都有良好的集成。

Calico在每一个计算节点利用Linux Kernel实现了一个高效的vRouter来负责数据转发，而每个vRouter通过BGP协议负责把自己上运行的工作负载（workload）的路由信息像整个Calico网络内传播——小规模部署可以直接互联，大规模下可通过指定的BGP route reflector来完成。这样保证最终所有的workload之间的数据流量都是通过IP路由的方式完成互联的。Calico节点组网可以直接利用数据中心的网络结构（无论是L2或者L3），不需要额外的NAT，隧道或者Overlay Network。

此外，Calico基于iptables还提供了丰富而灵活的网络Policy，保证通过各个节点上的ACLs来提供Workload的多租户隔离、安全组以及其他可达性限制等功能。

## Contiv

[Contiv](#)是思科开源的容器网络方案，主要提供基于Policy的网络管理，并与主流容器编排系统集成。Contiv最主要的优势是直接提供了多租户网络，并支持L2(VLAN), L3(BGP), Overlay (VXLAN)以及思科自家的ACI。

## Romana

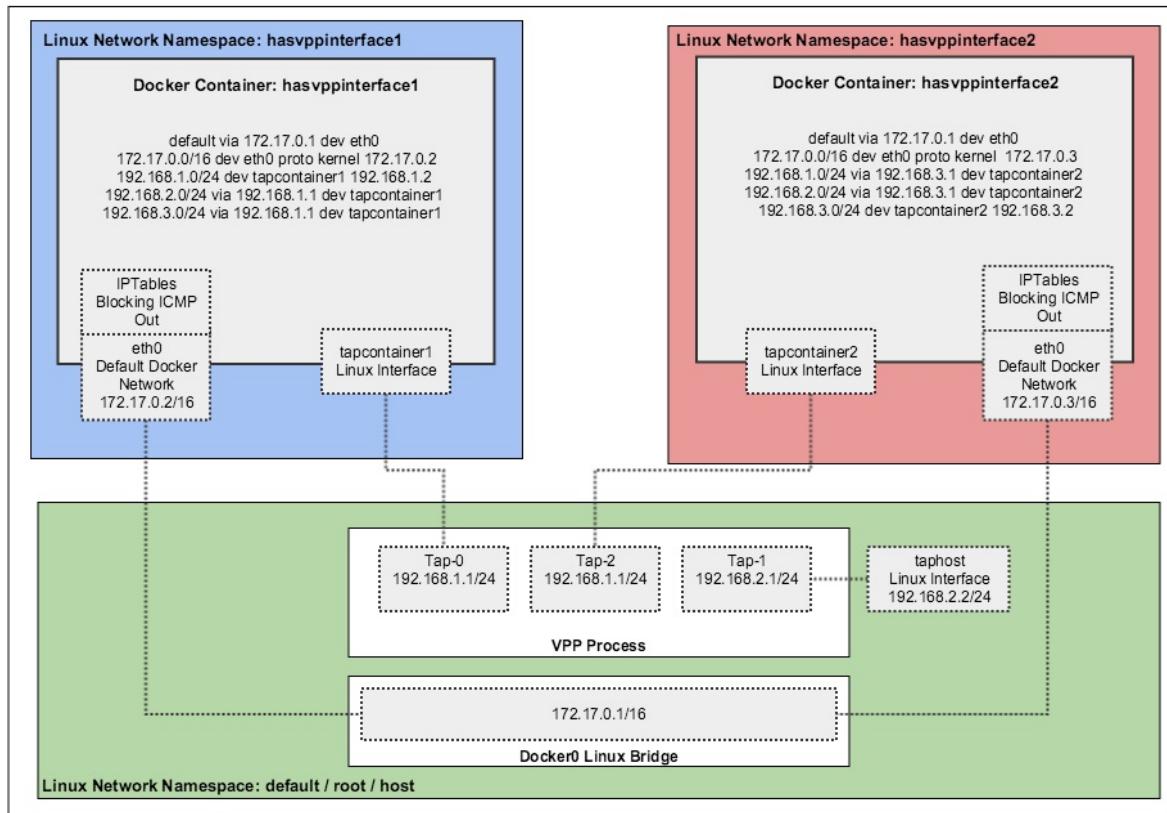
Romana是Panic Networks在2016年提出的开源项目，旨在借鉴route aggregation的思路来解决Overlay方案给网络带来的开销。

目前，Romana的docker plugin还在开发中。

## VPP

Intel clear container团队针对docker与clear container的集成开发了一个[FD.io VPP](#)（Vector Packet Processing）插件。

Vagrant Box (VPP Build Environment)



具体使用方法见<https://github.com/clearcontainers/vpp>。

## SR-IOV

SR-IOV 技术是一种基于硬件的虚拟化解决方案，允许在虚拟机之间高效共享 PCIe 设备，并获得能够与本机性能媲美的 I/O 性能。

# Kubernetes 网络

## Kubernetes 网络模型

- IP-per-Pod，每个Pod都拥有一个独立IP地址，Pod内所有容器共享一个网络命名空间
- 集群内所有Pod都在一个直接连通的扁平网络中，可通过IP直接访问
  - 所有容器之间无需NAT就可以直接互相访问
  - 所有Node和所有容器之间无需NAT就可以直接互相访问
  - 容器自己看到的IP跟其他容器看到的一样
- Service cluster IP尽可在集群内部访问，外部请求需要通过NodePort、LoadBalance或者Ingress来访问

## 官方插件

目前，Kubernetes 支持以下两种插件：

- kubenet：这是一个基于 CNI bridge 的网络插件（在 bridge 插件的基础上扩展了 port mapping 和 traffic shaping），是目前推荐的默认插件
- CNI：CNI 网络插件，需要用户将网络配置放到 /etc/cni/net.d 目录中，并将 CNI 插件的二进制文件放入 /opt/cni/bin
- exec：~~通过第三方的可执行文件来为容器配置网络，已在 v1.6 中移除，见 [kubernetes#39254](#)~~

## kubenet

kubenet 是一个基于 CNI bridge 的网络插件，它为每个容器建立一对 veth pair 并连接到 cbr0 网桥上。kubenet 在 bridge 插件的基础上拓展了很多功能，包括

- 使用 host-local IPAM 插件为容器分配 IP 地址，并定期释放已分配但未使用的 IP 地址
- 设置 sysctl net.bridge.bridge-nf-call-iptables = 1
- 为 Pod IP 创建 SNAT 规则
  - -A POSTROUTING ! -d 10.0.0.0/8 -m comment --comment "kubenet: SNAT for outbound traffic from cluster" -m addrtype ! --dst-type LOCAL -j MASQUERADE
- 开启网桥的 hairpin 和 promisc 模式，允许 Pod 访问它自己所在的 Service IP（即通过 NAT 后再访问 Pod 自己）

```
-A OUTPUT -j KUBE-DEDUP
-A KUBE-DEDUP -p IPv4 -s a:58:a:f4:2:1 -o veth+ --ip-src 10.244.2.1 -j ACCEPT
-A KUBE-DEDUP -p IPv4 -s a:58:a:f4:2:1 -o veth+ --ip-src 10.244.2.0/24 -j DROP
```

- HostPort 管理以及设置端口映射
- Traffic shaping , 支持通过 `kubernetes.io/ingress-bandwidth` 和 `kubernetes.io/egress-bandwidth` 等 Annotation 设置 Pod 网络带宽限制

未来 kubenet 插件会迁移到标准的 CNI 插件（如ptp）, 具体计划见[这里](#)。

## CNI plugin

安装CNI：

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=http://yum.kubernetes.io/repos/kubernetes-e17-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
      https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF

yum install -y kubernetes-cni
```

配置CNI brige插件：

```

mkdir -p /etc/cni/net.d
cat >/etc/cni/net.d/10-mynet.conf <<-EOF
{
  "cniVersion": "0.3.0",
  "name": "mynet",
  "type": "bridge",
  "bridge": "cni0",
  "isGateway": true,
  "ipMasq": true,
  "ipam": {
    "type": "host-local",
    "subnet": "10.244.0.0/16",
    "routes": [
      { "dst": "0.0.0.0/0" }
    ]
  }
}
EOF
cat >/etc/cni/net.d/99-loopback.conf <<-EOF
{
  "cniVersion": "0.3.0",
  "type": "loopback"
}
EOF

```

更多CNI网络插件的说明请参考[CNI 网络插件](#)。

## Flannel

[Flannel](#)是一个为Kubernetes提供overlay network的网络插件，它基于Linux TUN/TAP，使用UDP封装IP包来创建overlay网络，并借助etcd维护网络的分配情况。

```

kubectl create -f https://github.com/coreos/flannel/raw/master/Documentation/kube-flannel-rbac.yaml
kubectl create -f https://github.com/coreos/flannel/raw/master/Documentation/kube-flannel.yaml

```

## Weave Net

Weave Net是一个多主机容器网络方案，支持去中心化的控制平面，各个host上的wRouter间通过建立Full Mesh的TCP链接，并通过Gossip来同步控制信息。这种方式省去了集中式的K/V Store，能够在一定程度上减低部署的复杂性，Weave将其称为“data centric”，而非RAFT或者Paxos的“algorithm centric”。

数据平面上，Weave通过UDP封装实现L2 Overlay，封装支持两种模式，一种是运行在user space的sleeve mode，另一种是运行在kernal space的 fastpath mode。Sleeve mode通过pcap设备在Linux bridge上截获数据包并由wRouter完成UDP封装，支持对L2 traffic进行加密，还支持Partial Connection，但是性能损失明显。Fastpath mode即通过OVS的odp封装VxLAN并完成转发，wRouter不直接参与转发，而是通过下发odp流表的方式控制转发，这种方式可以明显地提升吞吐量，但是不支持加密等高级功能。

```
kubectl apply -f https://git.io/weave-kube
```

## Calico

Calico是一个基于BGP的纯三层的数据中心网络方案（不需要Overlay），并且与OpenStack、Kubernetes、AWS、GCE等IaaS和容器平台都有良好的集成。

Calico在每一个计算节点利用Linux Kernel实现了一个高效的vRouter来负责数据转发，而每个vRouter通过BGP协议负责把自己上运行的工作负载（workload）的路由信息像整个Calico网络内传播——小规模部署可以直接互联，大规模下可通过指定的BGP route reflector来完成。这样保证最终所有的workload之间的数据流量都是通过IP路由的方式完成互联的。Calico节点组网可以直接利用数据中心的网络结构（无论是L2或者L3），不需要额外的NAT，隧道或者Overlay Network。

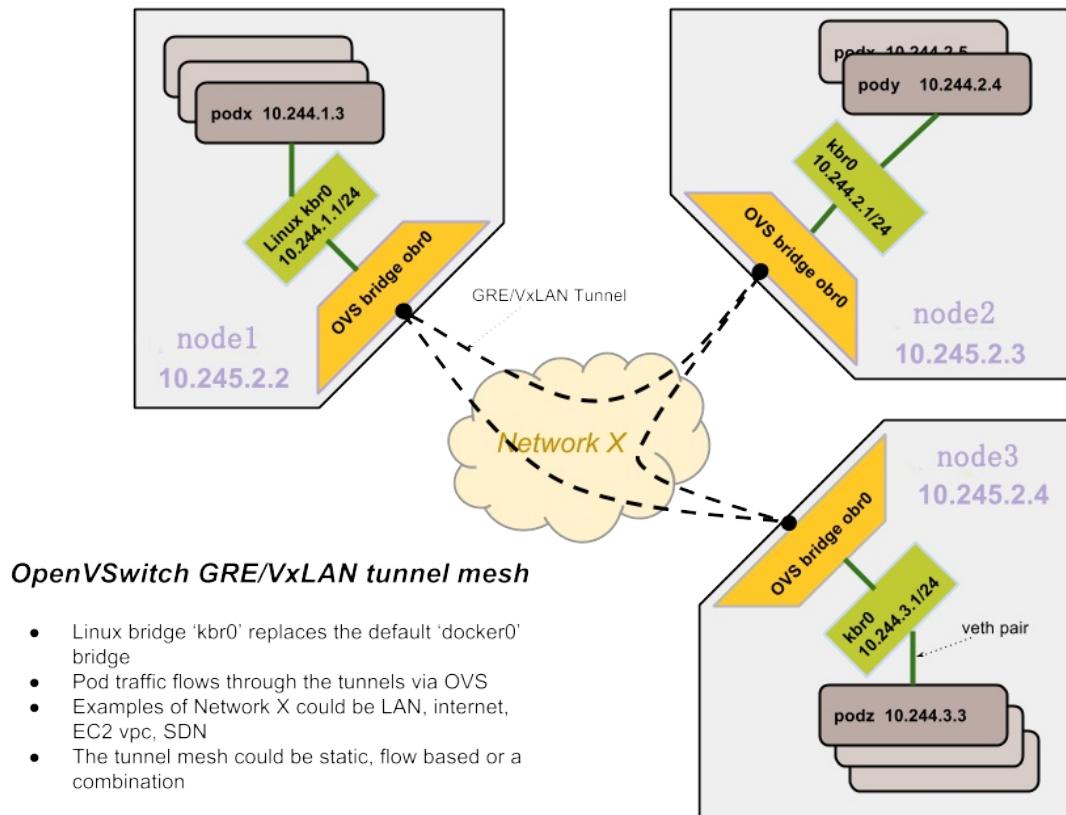
此外，Calico基于iptables还提供了丰富而灵活的网络Policy，保证通过各个节点上的ACLs来提供Workload的多租户隔离、安全组以及其他可达性限制等功能。

```
kubectl apply -f http://docs.projectcalico.org/v2.1/getting-started/kubernetes/installation/hosted/kubeadm/1.6/calico.yaml
```

## OVS

<https://kubernetes.io/docs/admin/ovs-networking/>提供了一种简单的基于OVS的网络配置方法：

- 每台机器创建一个Linux网桥kbr0，并配置docker使用该网桥（而不是默认的docker0），其子网为10.244.x.0/24
- 每台机器创建一个OVS网桥obr0，通过veth pair连接kbr0并通过GRE将所有机器互联
- 开启STP
- 路由10.244.0.0/16到OVS隧道



## OVN

OVN (Open Virtual Network) 是OVS提供的原生虚拟化网络方案，旨在解决传统SDN架构（比如Neutron DVR）的性能问题。

OVN为Kubernetes提供了两种网络方案：

- Overlay: 通过ovs overlay连接容器
- Underlay: 将VM内的容器连到VM所在的相同网络（开发中）

其中，容器网络的配置是通过OVN的CNI插件来实现。

## Contiv

Contiv是思科开源的容器网络方案，主要提供基于Policy的网络管理，并与主流容器编排系统集成。Contiv最主要的优势是直接提供了多租户网络，并支持L2(VLAN), L3(BGP), Overlay (VXLAN)以及思科自家的ACI。

## Romana

Romana是Panic Networks在2016年提出的开源项目，旨在借鉴 route aggregation的思路来解决Overlay方案给网络带来的开销。

## OpenContrail

OpenContrail是Juniper推出的开源网络虚拟化平台，其商业版本为Contrail。其主要由控制器和vRouter组成：

- 控制器提供虚拟网络的配置、控制和分析功能
- vRouter提供分布式路由，负责虚拟路由器、虚拟网络的建立以及数据转发

其中，vRouter支持三种模式

- Kernel vRouter：类似于ovs内核模块
- DPDK vRouter：类似于ovs-dpdk
- Netronome Agilio Solution (商业产品)：支持DPDK, SR-IOV and Express Virtio (XVIO)

[Juniper/contrail-kubernetes](#) 提供了Kubernetes的集成，包括两部分：

- kubelet network plugin基于kubernetes v1.6已经删除的exec network plugin
- kube-network-manager监听kubernetes API，并根据label信息来配置网络策略

## Midonet

Midonet是Midokura公司开源的OpenStack网络虚拟化方案。

- 从组件来看，Midonet以Zookeeper+Cassandra构建分布式数据库存储VPC资源的状态——Network State DB Cluster，并将controller分布在转发设备（包括vswitch和L3 Gateway）本地——Midolman（L3 Gateway上还有quagga bgpd），设备的转发则保留了ovs kernel作为fast datapath。可以看到，Midonet和DragonFlow、OVN一样，在架构的设计上都是沿着OVS-Neutron-Agent的思路，将controller分布到设备本地，并在neutron plugin和设备agent间嵌入自己的资源数据库作为super controller。
- 从接口来看，NSDB与Neutron间是REST API，Midolman与NSDB间是RPC，这俩没什么好说的。Controller的南向方面，Midolman并没有用OpenFlow和OVSDP，它干掉了user space中的vswitchd和ovsdb-server，直接通过linux netlink机制操作kernel space中的ovs datapath。

## Host network

最简单的网络模型就是让容器共享 Host 的 network namespace，使用宿主机的网络协议栈。这样，不需要额外的配置，容器就可以共享宿主的各种网络资源。

## 优点

- 简单，不需要任何额外配置
- 高效，没有NAT等额外的开销

## 缺点

- 没有任何的网络隔离
- 容器和Host的端口号容易冲突
- 容器内任何网络配置都会影响整个宿主机

注意：HostNetwork 是在 Pod 配置文件中设置的，kubelet 在启动时还是需要配置使用 CNI 或者 kubenet 插件（默认kubenet）。

## 其他

### ipvs

Kubernetes v1.8已经支持ipvs负载均衡模式（alpha版）。

### Canal

Canal是Flannel和Calico联合发布的一个统一网络插件，提供CNI网络插件，并支持network policy。

### kuryr-kubernetes

kuryr-kubernetes是OpenStack推出的集成Neutron网络插件，主要包括Controller和CNI插件两部分，并且也提供基于Neutron LBaaS的Service集成。

### Cilium

Cilium是一个基于eBPF和XDP的高性能容器网络方案，提供了CNI和CNM插件。

项目主页为<https://github.com/cilium/cilium>。

### kope

kope是一个旨在简化Kubernetes网络配置的项目，支持三种模式：

- Layer2：自动为每个Node配置路由
- Vxlan：为主机配置vxlan连接，并建立主机和Pod的连接（通过vxlan interface和ARP entry）

- **ipsec**：加密链接

项目主页为<https://github.com/kopeio/kope-routing>。

# Mininet

Mininet是一个由Stanford大学Nick研究小组开发的网络虚拟化平台，可以用来方便的测试、验证和研究OpenFlow和SDN网络。

Mininet使用Linux Network Namespaces来创建虚拟节点，默认情况下，Mininet会为每一个host创建一个新的网络命名空间，同时在root Namespace（根进程命名空间）运行交换机和控制器的进程，因此这两个进程就共享host网络命名空间。由于每个主机都有各自独立的网络命名空间，我们就可以进行个性化的网络配置和网络程序部署。

Mininet提供了命令行工具 `mn` 和Python API，用来构建网络拓扑。

## 安装Mininet

在Ubuntu系统上可以使用通过下面的命令来安装mininet：

```
sudo apt-get install -y mininet openvswitch-testcontroller
sudo /usr/bin/ovs-testcontroller /usr/bin/ovs-controller
sudo service openvswitch-switch start
```

然后运行下面的命令验证安装是否正常

```
sudo mn --test pingall
```

其他系统上，可以参考[这里](#)下载预置mininet的虚拟机镜像。

## `mn`命令行

直接运行`mn`命令可以进入mininet控制台，默认创建一个 `minimal` 拓扑，即一个控制器、一台OpenFlow交换机并连着两台host。

```
$ sudo mn
mininet> nodes
available nodes are:
h1 h2 s1
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=21291>
<Host h2: h2-eth0:10.0.0.2 pid=21293>
<OVSBridge s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=21298>
```

以节点名字开始的命令在节点内运行

```
mininet> h1 ifconfig -a
h1-eth0 Link encap:Ethernet HWaddr aa:7d:6a:7f:b5:52
          inet addr:10.0.0.1 Bcast:10.255.255.255 Mask:255.0.0.0
          inet6 addr: fe80::a87d:6aff:fe7f:b552/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
            RX packets:15 errors:0 dropped:0 overruns:0 frame:0
            TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:1206 (1.2 KB) TX bytes:648 (648.0 B)

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:65536 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

mininet> h1 ping -c1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.040 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.040/0.040/0.040/0.000 ms
```

## 命令和选项

### 命令列表

- **py** : 执行python命令，如 `py h1.IP()`
- **dump** : 查看各节点的信息
- **nodes** : 查看所有节点
- **net** : 查看链路信息
- **links** : 查看网络接口连接拓扑
- **link** : 开启或关闭网络接口，如 `link s1 h1 up`
- **xterm** : 开启终端
- **dpctl** : 操作OpenFlow流表
- **pingall** : 自动ping测试

### 选项列表

- **--topo** : 自定义拓扑，如 `linear|minimal|reversed|single|torus|tree`
- **--link** : 自定义网络参数，如 `default|ovs|tc`

- `--switch` : 自定义虚拟交换机，如 `default|ivs|lxbr|ovs|ovsbr|ovsk|user`
- `--controller` : 自定义控制器，如 `default|none|nox|ovsc|ref|remote|ryu`
- `--nat` : 自动设置NAT
- `--cluster` : 集群模式，将网络拓扑运行在多台机器上
- `--mac` : 自动设置主机mac
- `--arp` : 自动设置arp表项

## 自定义网络拓扑

默认的 `minimal` 拓扑比较简单，可以使用 `--topo` 选项设置网络拓扑。

```
$ sudo mn --topo single,3
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=22193>
<Host h2: h2-eth0:10.0.0.2 pid=22195>
<Host h3: h3-eth0:10.0.0.3 pid=22197>
<OVSBridge s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None pid=22202>
```

## 自定义网络参数

可以使用 `--link` 选项设置网络参数。

```
$ sudo mn --link tc,bw=10,delay=10ms
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['9.50 Mbits/sec', '12.0 Mbits/sec']
```

## 自定义独立命名空间

默认情况下，host在独立的netns中，而switch和controller都还是使用host netns，可以使用 `-innamespace` 选项将它们也放到独立的netns中。

```
$ sudo mn --innamespace --switch user
```

## Python API

对于复杂的网络，需要使用[Python API](#)来构建。

比如，使用python API构造一个单switch接4台虚拟节点的示例

```

#!/usr/bin/python
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.util import dumpNodeConnections
from mininet.log import setLogLevel
from mininet.node import OVSController

class SingleSwitchTopo(Topo):
    "Single switch connected to n hosts."
    def build(self, n=2):
        switch = self.addSwitch('s1')
        # Python's range(N) generates 0..N-1
        for h in range(n):
            host = self.addHost('h%s' % (h + 1))
            self.addLink(host, switch)

    def simpleTest():
        "Create and test a simple network"
        topo = SingleSwitchTopo(n=4)
        net = Mininet(topo=topo, controller = OVSController)
        net.start()
        print "Dumping host connections"
        dumpNodeConnections(net.hosts)
        print "Testing network connectivity"
        net.pingAll()
        net.stop()

topos = {"mytopo": SingleSwitchTopo}

if __name__ == '__main__':
    # Tell mininet to print useful information
    setLogLevel('info')
    simpleTest()

```

这个脚本可以直接运行，也可以使用mn命令启动

```
$ sudo mn --custom a.py --topo mytopo,3 --test pingall
```

Mininet v2.2.0+还提供了一个miniedit的可视化界面，更直观的编辑和查看网络拓扑。miniedit实际上是一个python脚本，存放在mininet安装目录的examples中，如 /usr/lib/python2.7/dist-packages/mininet/examples/miniedit.py 。

## 参考文档

- [Mininet官方网站](#)
- [Mininet Github](#)

- REPRODUCING NETWORK RESEARCH

## SDN实践案例

SDN网络实践案例。

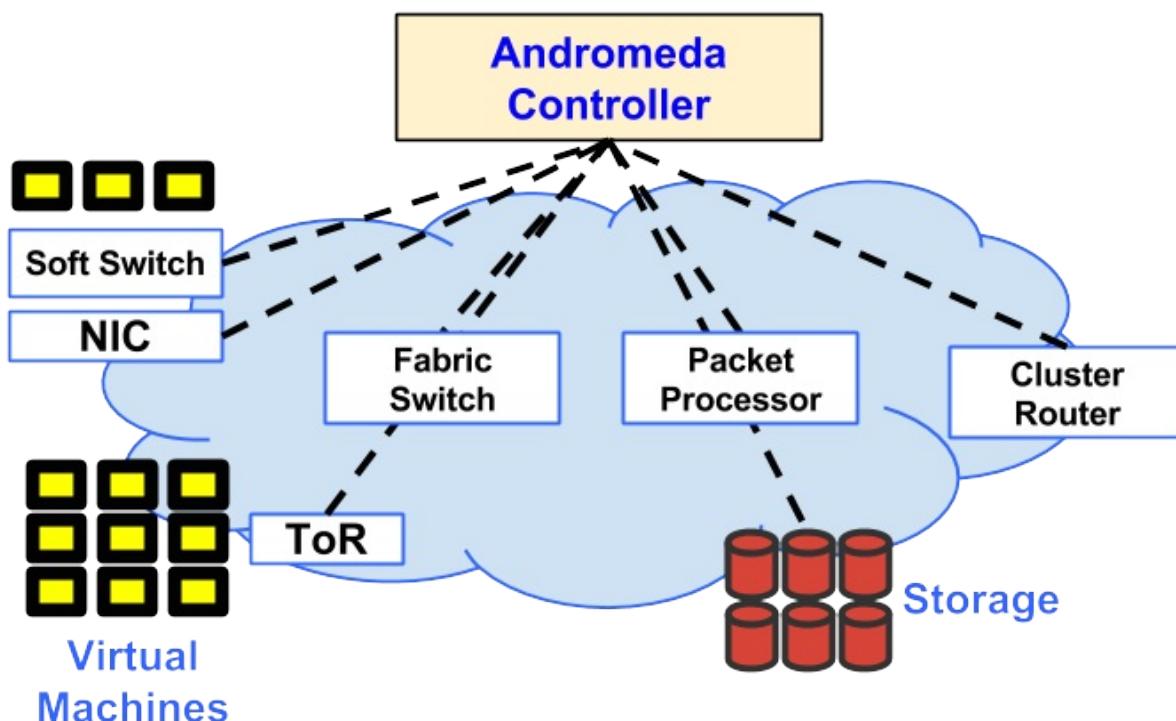
# Google Data Center Networks

## Jupiter

Google通过SDN的途径来构建Jupiter，Jupiter是一个能够支持超过10万台服务器规模的数据中心互联架构。它支持超过1 Pb/s的总带宽来承载其服务。

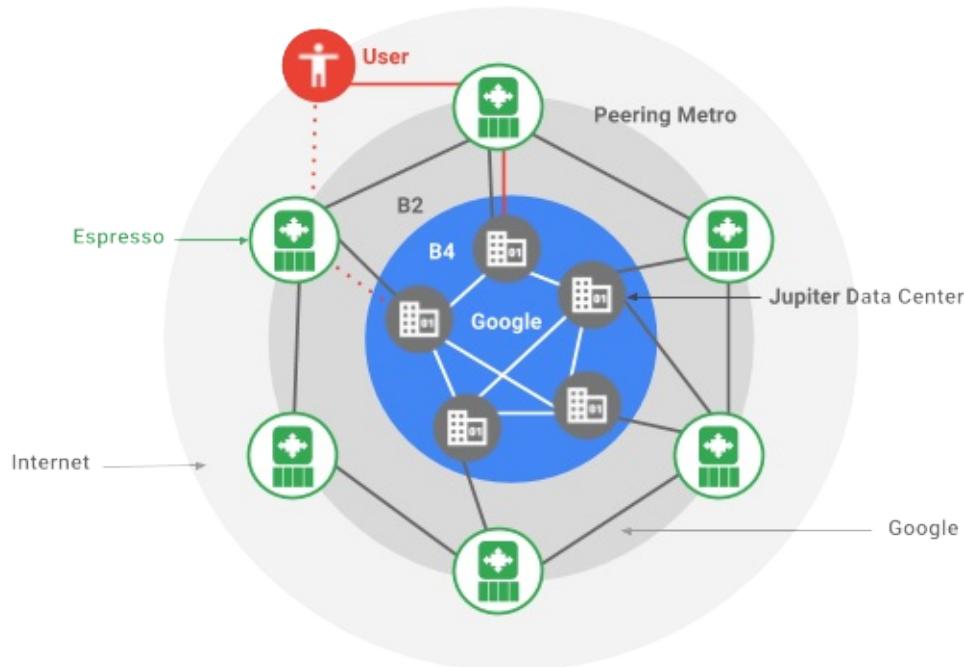
## Andromeda

Google Andromeda是一个网络功能虚拟化（NFV）堆栈，通过融合软件定义网络(SDN)和网络功能虚拟化(NFV)，Andromeda能够提供分布式拒绝服务(DDoS)攻击保护、透明的服务负载均衡、访问控制列表和防火墙。



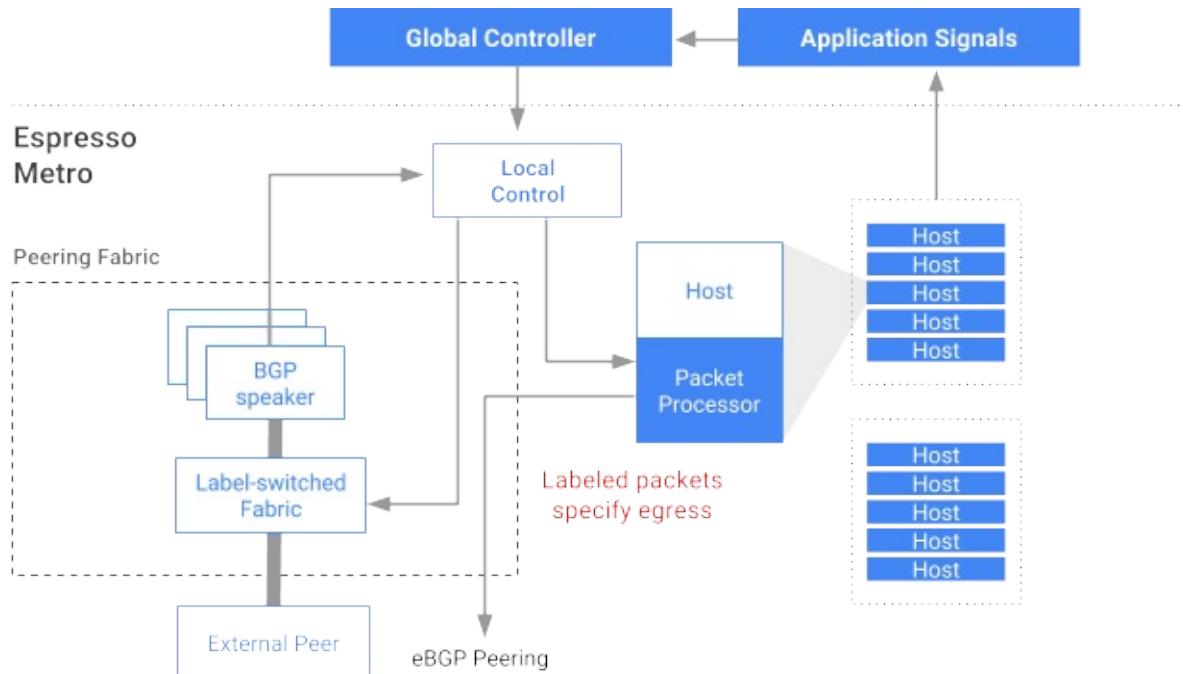
## Espresso

Espresso将SDN扩展到Google网络的对等边缘，连接到全球其他网络。Espresso使得Google根据网络连接实时性的测量动态智能化地为个人用户提供服务。



“根据其IP地址（或DNS解析器的IP地址），我们动态选择最佳网络接入点，并根据实际的性能数据重新平衡流量，而不是选择一个静态点。”

Expresso在与标签交换结构相同的服务器上运行边界网关协议（BGP）。分组处理器将标签插入每个数据包。路由器读取标签后，每个城市的本地控制器都可以编写标签交换结构。服务器向全局控制器发送流量实时的情况简报。通过这种途径，本地控制器可以实时更新，全球控制器可以集成所有区域。



## B4

Google构建B4网络实现了数据中心的连接，以便在各个校园网之间实时复制数据。B4是基于白盒交换机建立的，受谷歌开发的软件控制。Google的目标是建立一个复制网络，随着它的成长，逐渐承载关键的网络任务。B4的增长速度比我们的公共网络快。

#### 参考链接

- <https://cloudplatform.googleblog.com/2015/06/A-Look-Inside-Googles-Data-Center-Networks.html>
- <https://cloudplatform.googleblog.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html>
- <https://blog.google/topics/google-cloud/making-google-cloud-faster-more-available-and-cost-effective-extending-sdn-public-internet-espresso/>
- <https://people.eecs.berkeley.edu/~sylvia/cs268-2014/papers/b4-sigcomm13.pdf>

# FAQ

## 如何定位丢包问题

- 如何知道哪个网卡在丢包：`netstat -i`
- 如何知道什么时候丢包：`perf record -g -a -e skb:kfree_skb`
- 如何知道哪里丢包了：[https://github.com/pavel-odintsov/drop\\_watch](https://github.com/pavel-odintsov/drop_watch)

## 如何查看Linux系统的带宽流量

- 按网卡查看流量：`ifstat`、`dstat -nf` 或 `sar -n DEV 1 2`
- 按进程查看流量：`nethogs`
- 按连接查看流量：`iptraf`、`iftop` 或 `tcptrack`
- 查看流量最大的进程：`sysdig -c toprocs_net`
- 查看流量最大的端口：`sysdig -c topports_server`
- 查看连接最多的服务器端口：`sysdig -c fdbytes_by fd.sport`

## 参考文档

- [Monitoring and Tuning the Linux Networking Stack: Receiving Data](#)
- [Monitoring and Tuning the Linux Networking Stack: Sending Data](#)

# 参考文档

- 《计算机网络》
- 《TCP/IP详解》
- 《深入浅出DPDK》
- 《重构网络-SDN架构与实现》
- 《SDN核心技术剖析和实战指南》
- 《深入理解LINUX网络技术内幕》
- [Open Networking Foundation](#)
- [OpenStack Networking Guide](#)
- <https://www.openstack.org/>
- <https://www.opnfv.org/>
- <http://dpdk.org/>
- <http://openvswitch.org/>
- <https://kernelnewbies.org/>
- <https://www.opendaylight.org/>
- <https://www.docker.com/>
- <https://kubernetes.io/>
- <https://github.com/containernetworking/cni>