

# 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
SNMP	2.1.11
LLDP	2.1.12
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
Open Virtual Network	2.3.4
OVN在Ubuntu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
6. 安全设备	2.5
VPN	2.5.1
IPSec VPN	2.5.1.1
SSL VPN	2.5.1.2

---

ICG	2.5.2
Firewall	2.5.3
工作原理	2.5.3.1
常见分类	2.5.3.2
演进过程	2.5.3.3

---

# SDN&NFV

7. SDN	3.1
YANG Language	3.1.1
SDN控制器	3.1.2
OpenDaylight	3.1.2.1
OpenDaylight Projects	3.1.2.1.1
DataStore	3.1.2.1.2
ONOS	3.1.2.2
Floodlight	3.1.2.3
Ryu	3.1.2.4
NOX/POX	3.1.2.5
南向接口	3.1.3
OpenFlow	3.1.3.1
OF-Config	3.1.3.2
NETCONF	3.1.3.3
NETCONF Call Home	3.1.3.3.1
YANG Module for NETCONF Monitoring	3.1.3.3.2
NETCONF请求和响应中的标签	3.1.3.3.3
P4	3.1.3.4
AAA	3.1.3.5
Radius	3.1.3.5.1
数据平面	3.1.4
8. NFV	3.2
9. SDWAN	3.3

---

# SDN实践

---

11. Mininet	4.1
12. Neutron	4.2
13. SDN实践案例	4.3
Google数据中心网络	4.3.1

---

## SDN业务类型示例

14. 业务示例	5.1
SDN控制器应用场景	5.1.1
业务控制平台-SCP	5.1.2

---

## 参考文档

15. FAQ	6.1
16. 参考文档	6.2

---

## ChangeLog

ChangeLog	7.1
-----------	-----

---

# 前言

build error

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/tonydeng/sdn-handbook>。

# 网络基础理论

计算机网络理论知识

# TCP/IP网络模型

## TCP/IP网络模型

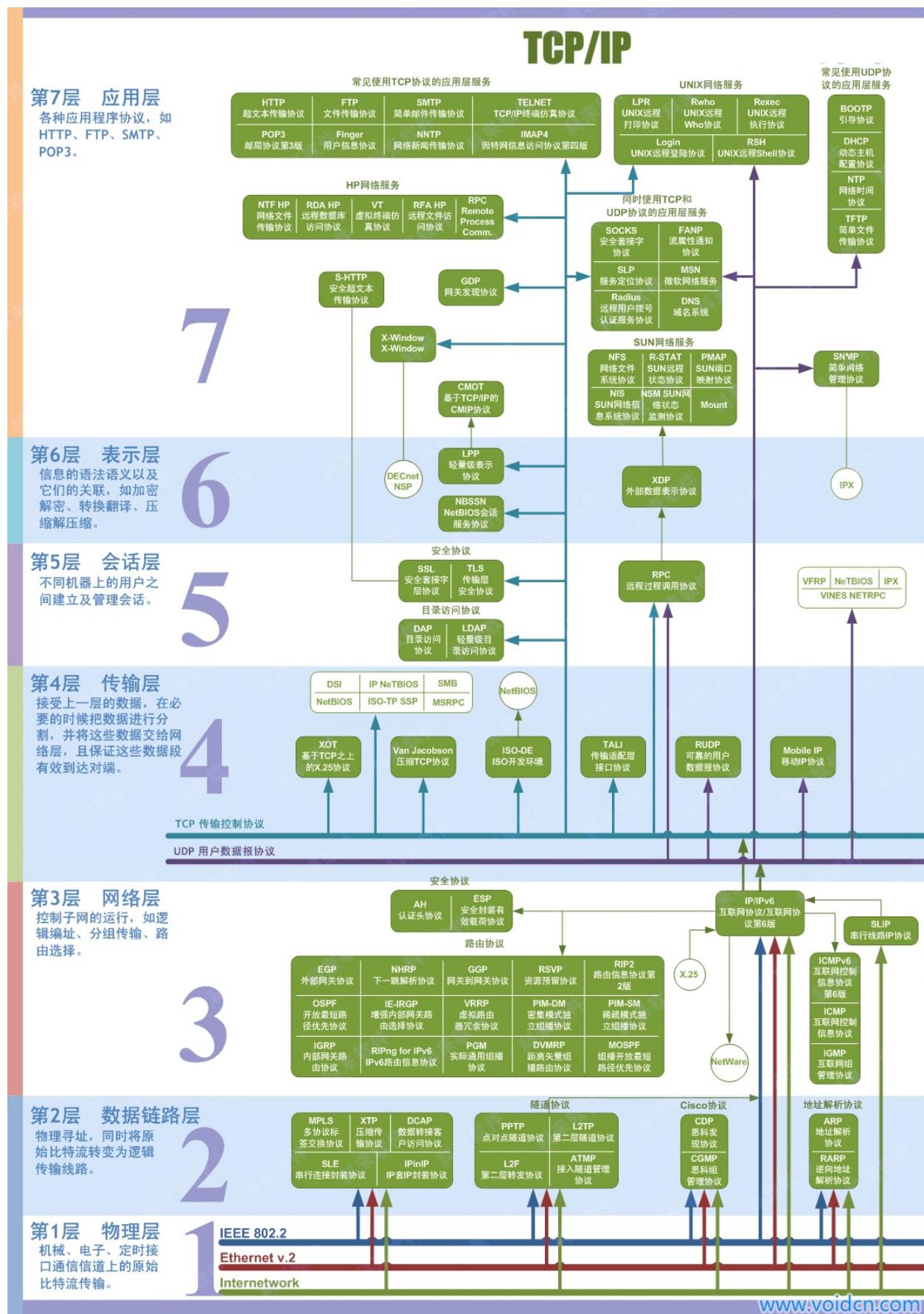
TCP/IP模型是互联网的基础，它是一系列网络协议的总称。这些协议可以划分为四层，分别为链路层、网络层、传输层和应用层。

- 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。

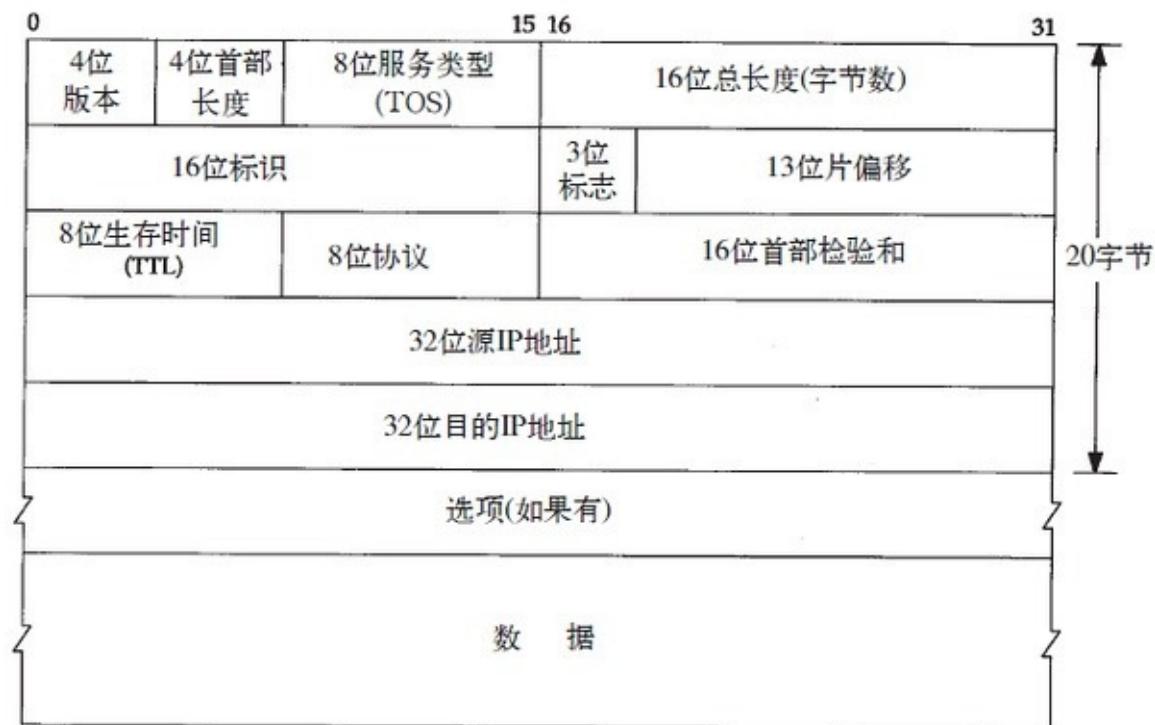
## 6 MTU

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

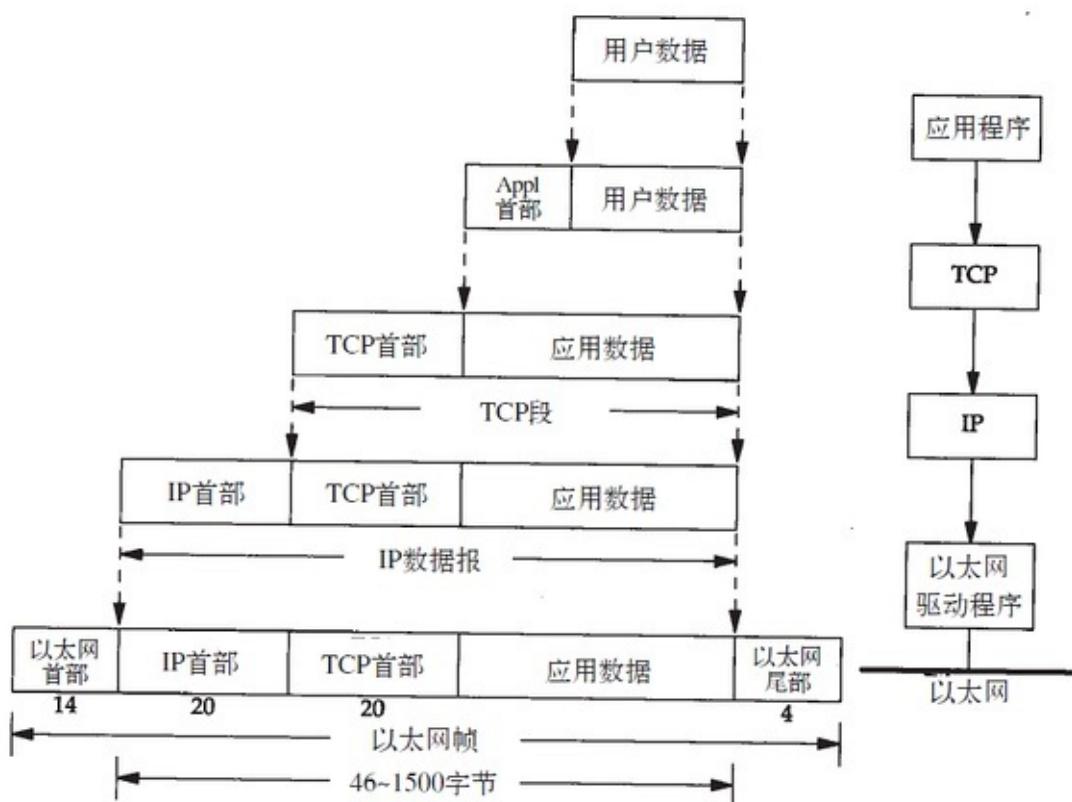
子网掩码用来确定多少bit用于网络号和多少bit用于主机号。

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。

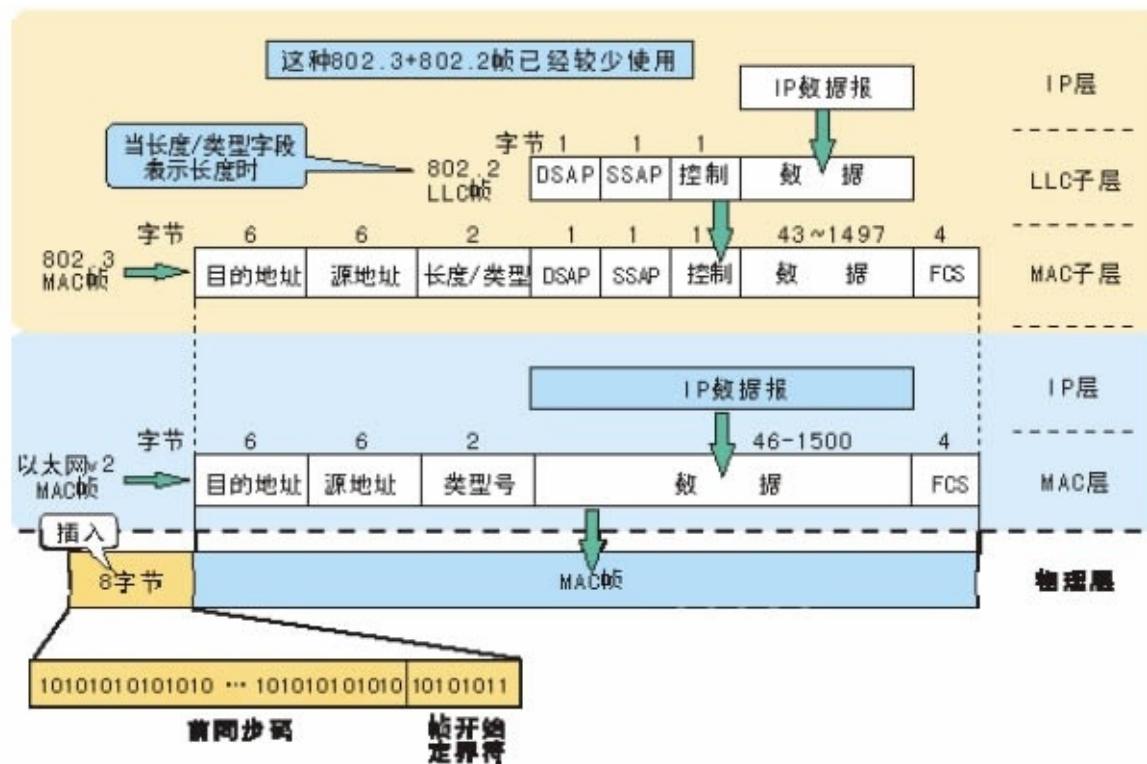
如果知道本机的IP地址，那么就知道它是否为A类、B类或C类地址(从IP地址的高位可以得知)，也就知道网络号和子网号之间的分界线。而根据子网掩码就可知道子网号与主机号之间的分界线。

## 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。

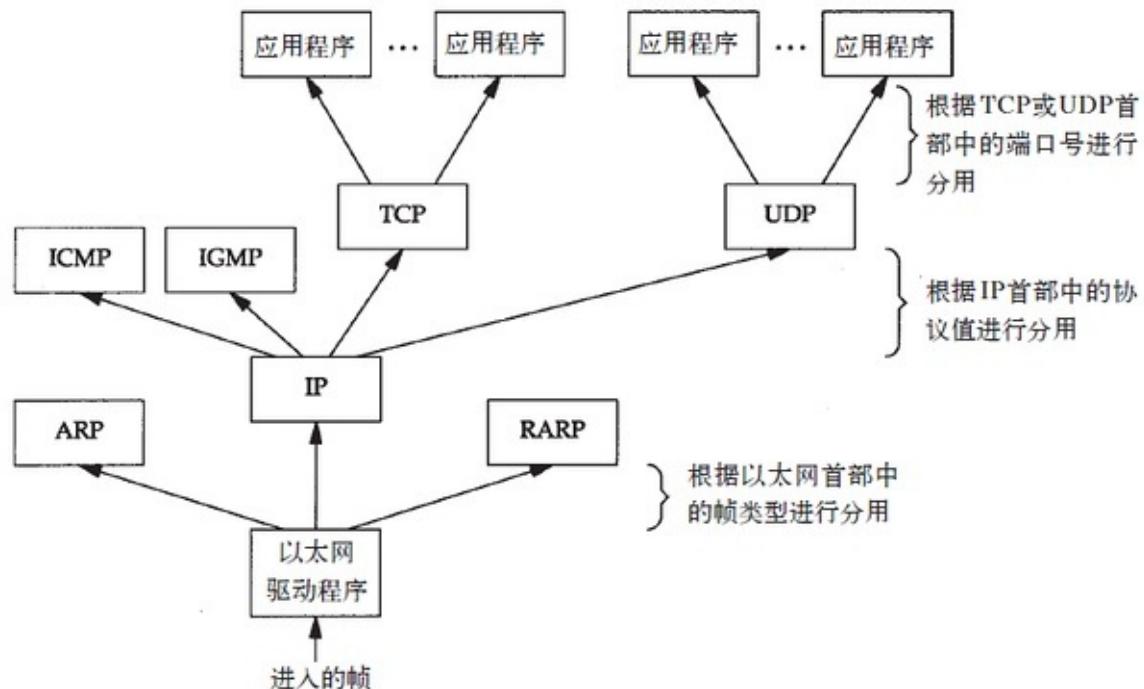


而更具体的以太网帧格式为



## 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

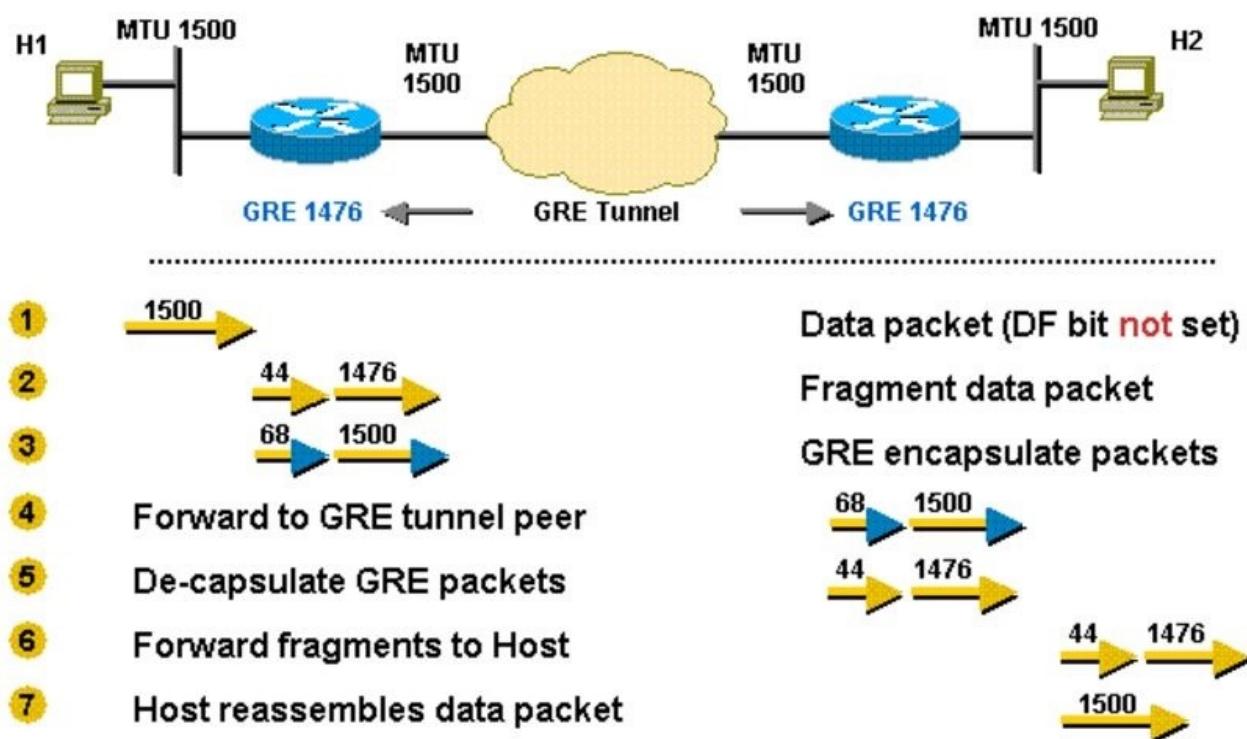
现在很多网卡本身支持数据分片，这样，上层L4/L3就可以不用进行分片(最大64KB)，而由NIC来完成，从而提高网络性能。

- 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

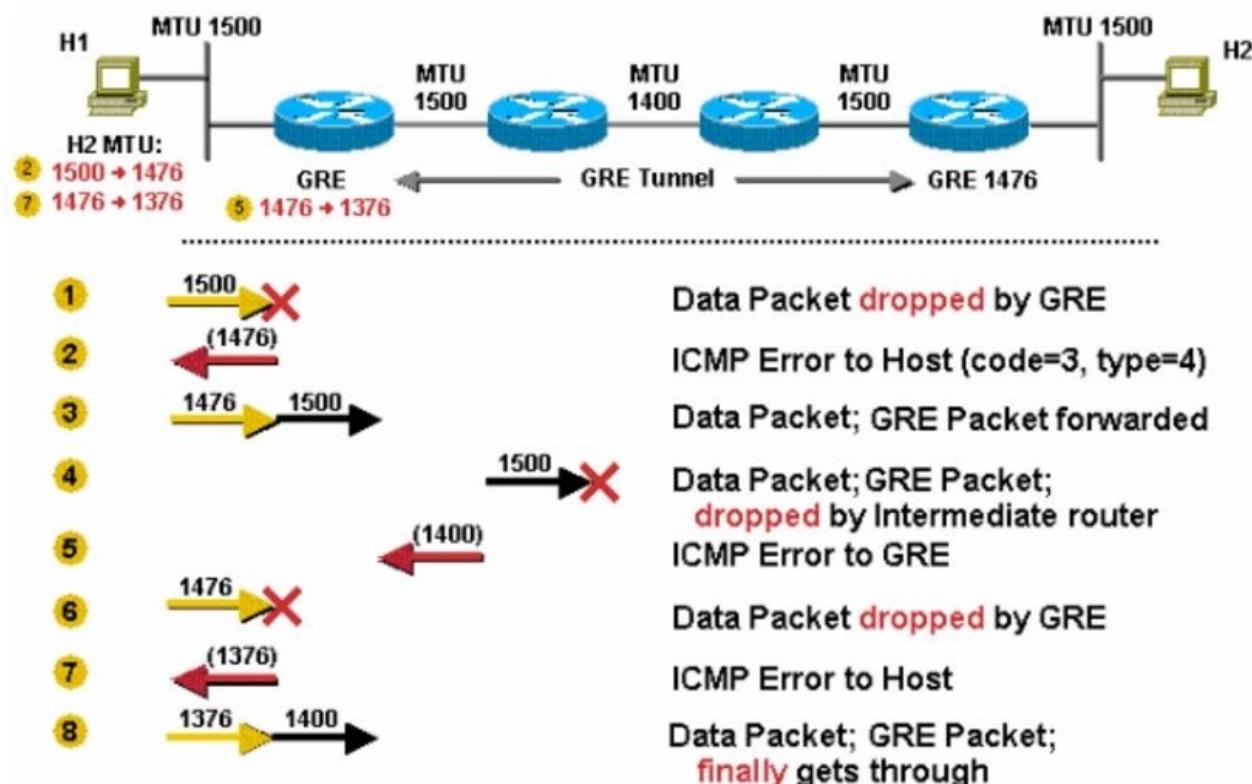
## PMTU (Path Maximum Transmission Unit Discovery)

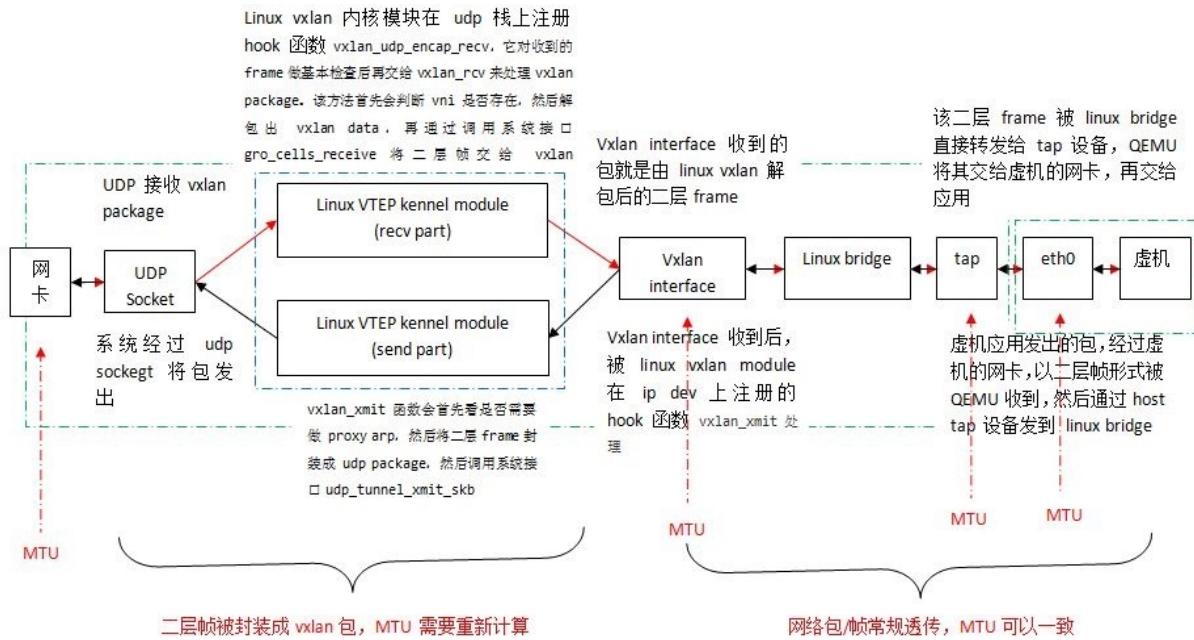
PMTU 的用途是动态的确定从发送端到接收端整个路径上的最小 MTU，从而避免分包。注意，PMTU 只支持 TCP，对其他协议比如 UDP 无效。而且，如果发送方已经开启了 PMTU，那么它发送的所有 TCP/IP 包的 DF 标志都被设置为 1 即不再允许分包。当网络路径上某个路由器发现发送者的包因为超过前面转发路径的 MTU 而无法发送时，它向发送者返回一个 ICMP "Destination Unreachable" 消息，其中包含了那个 MTU，然后发送者就会在它的路由表中将该mtu值保存下来，再使用较小的 MTU 重新发出新的较小的包。

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

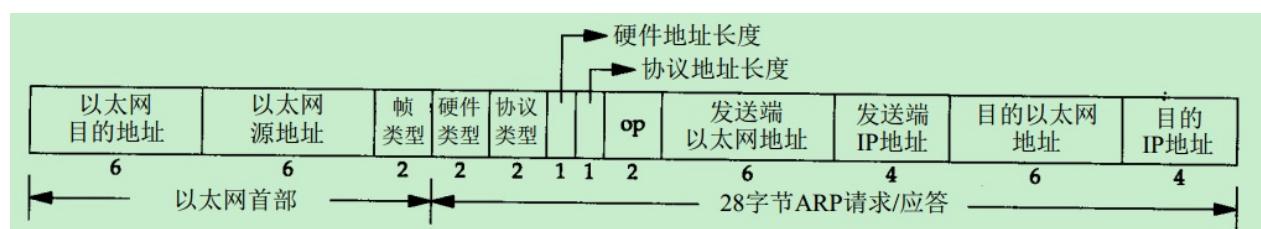
ARP 高效运行的关键是每台主机上都有一个 ARP 高速缓存，缓存中每一项的生存时间为 20 分钟，但不完整表项超时时间为 3 分钟（如 192.168.13.254）。

```
# 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 FF: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。

## 参考

- 地址解析协议 (ARP)
- ARP协议揭密
- TCP/IP协议详解卷一 -- ARP：地址解析协议

# 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			时间截请求(6.4节)		.			
14	0			时间截应答(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。

0 0   1   2   3   4   5   6   7   8   9	1 0   1   2   3   4   5   6   7   8   9	2 0   1   2   3   4   5   6   7   8   9	3 0   1
Type	Code	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。

类型(Type)	编码(Code)	description
0	0	回声应答 (ping)
3	0	目的网络不可达
3	1	目的主机不可达
3	2	目的协议不可达
3	3	目的端口不可达
3	6	目的网络未知
3	7	目的主机未知
4	0	源抑制(拥塞控制-未用)
8	0	回声请求(ping)
9	0	路由通告
10	0	路由发现
11	0	TTL超期
12	0	IP首部错误

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 首部要被送回的原因是因为 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。

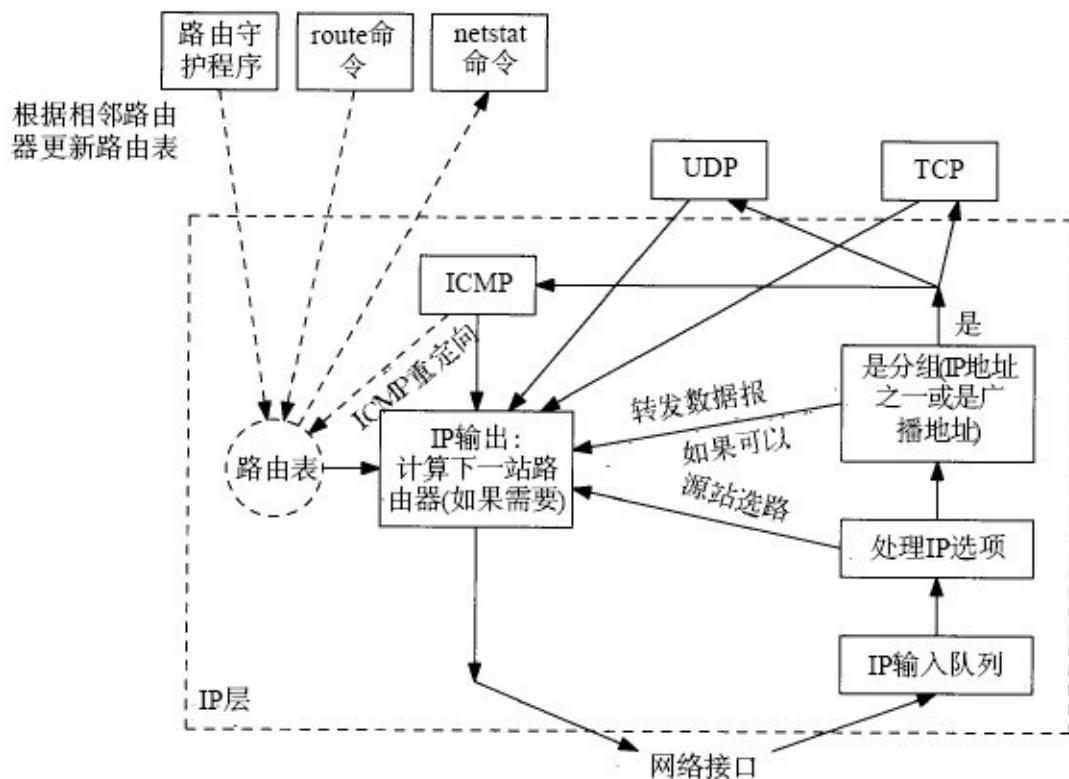
## 参考

- [TCP/IP协议详解卷一 -- ICMP : Internet控制报文协议](#)
- [Traceroute](#)



# 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 ( Multiprotocol 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。

## 参考

- What is IP routing?
- IP Routing
- 路由表
- RIP
- Open Shortest Path First (OSPF)
- Intermediate System to Intermediate System (IS-IS)
- Border Gateway Protocol (BGP)

- Spanning Tree Protocol (STP)
- TRILL ("TRansparent Interconnection of Lots of Links")
- Multiprotocol Label Switching (MPLS)
- RFC3031 - Multiprotocol Label Switching Architecture
- TCP/IP协议详解卷一 -- IP选路

# 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

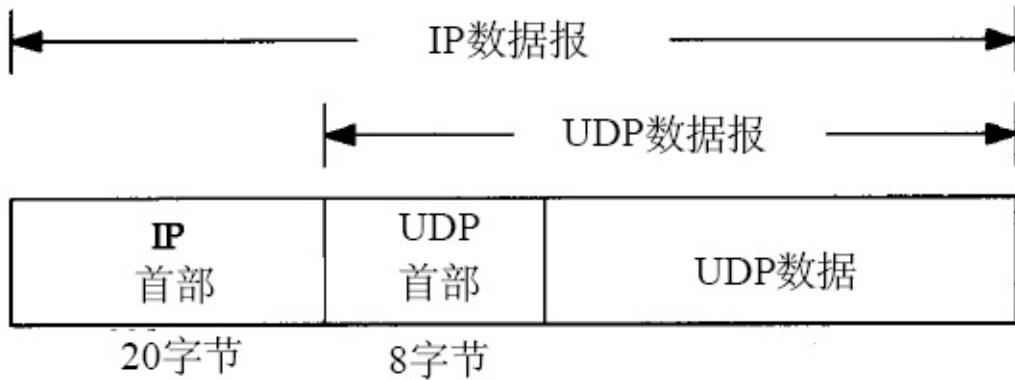
当然了，并不是所有的软件都适用于所有的硬件，这还有待软硬件厂商进一步协作提升开放性。

## 参考

- 网络交换机

# 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。由于每一分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。

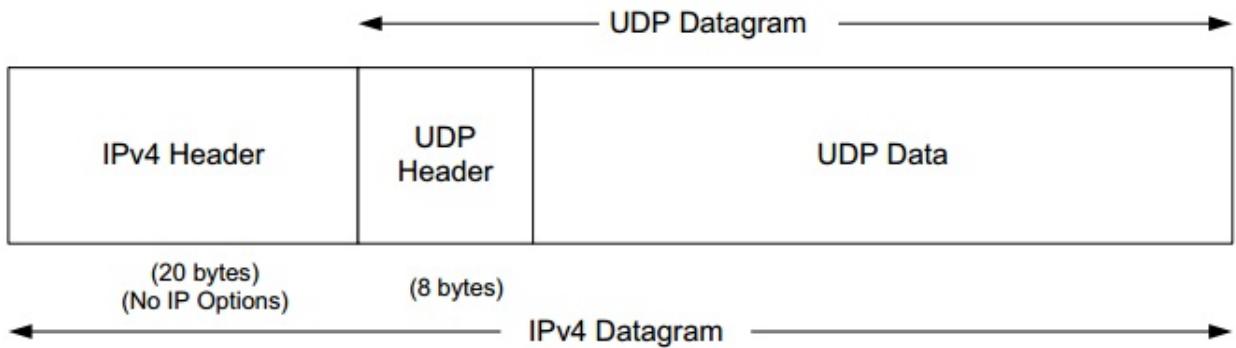
尽管 IP 分片过程看起来透明的，但有一点让人不想使用它：即使只丢失一片数据也要重新传整个数据报。

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： $65535 - 20 \text{ 字节 IP 首部长度} - 8 \text{ 字节 UDP 首部长度} = 65507$ 。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。

## 参考

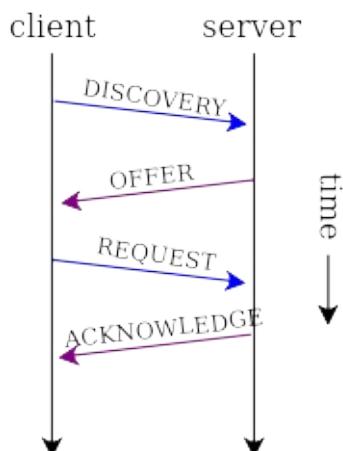
- [User Datagram Protocol](#)
- [UDP - IETF](#)

# 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 (server) 和 68 (client)。

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 是实现中，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.

## 参考

- [Dynamic Host Configuration Protocol](#)

# 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：



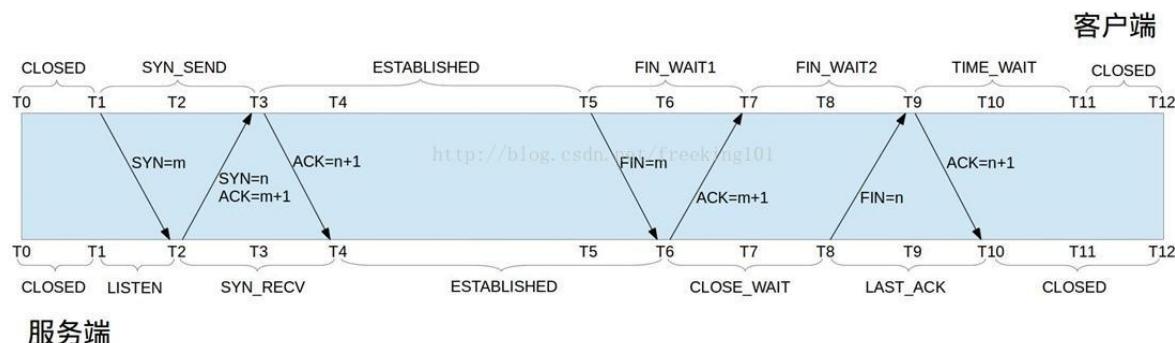
其中，

- 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

### TCP 三次握手、四次握手状态迁移



Time	172.20.15.114 121.201.0.74	Comment
1.033054000	SYN (54753) → (80)	Seq = 0
1.084770000	SYN, ACK (54753) ← (80)	Seq = 0 Ack = 1
1.084829000	ACK (54753) → (80)	Seq = 1 Ack = 1
1.085470000	PSH, ACK - ... (54753) → (80)	Seq = 1 Ack = 1
1.128582000	ACK (54753) ← (80)	Seq = 1 Ack = 1028
1.252997000	PSH, ACK - ... (54753) → (80)	Seq = 1 Ack = 1028
1.273069000	ACK (54753) → (80)	Seq = 1028 Ack = 226
1.320313000	PSH, ACK - ... (54753) → (80)	Seq = 1028 Ack = 226
1.363172000	ACK (54753) ← (80)	Seq = 226 Ack = 2100
1.370619000	PSH, ACK - ... (54753) → (80)	Seq = 226 Ack = 2100
1.374224000	PSH, ACK - ... (54753) → (80)	Seq = 2100 Ack = 472
1.422456000	ACK (54753) ← (80)	Seq = 472 Ack = 3063
1.422744000	PSH, ACK - ... (54753) → (80)	Seq = 472 Ack = 3063
1.422745000	FIN, ACK (54753) ← (80)	Seq = 1396 Ack = 3063
1.422805000	ACK (54753) → (80)	Seq = 3063 Ack = 1397
1.423852000	FIN, ACK (54753) → (80)	Seq = 3063 Ack = 1397
1.466871000	ACK (54753) ← (80)	Seq = 1397 Ack = 3064

上图由 wireshark 抓取，并显示了 TCP 状态图

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：

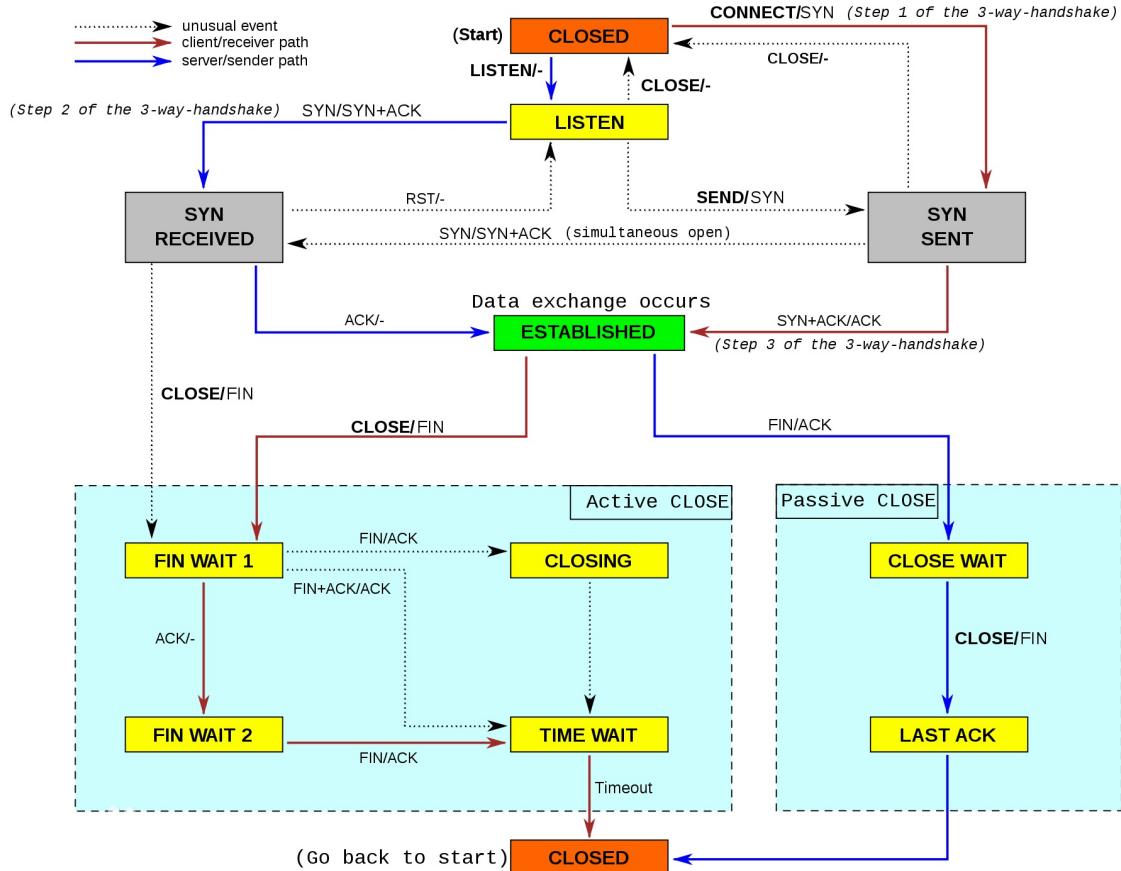
1. 以上面的抓包为例，客户端向服务器发送了一个 FIN （ NO. 6 ）；
2. 服务器端对上面的 FIN 进行确认（ NO. 7 ），同时向客户端发送一个 FIN （这儿其实是两个动作，一个是对上面 FIN 的 ACK ，另一个是发送一个 FIN ，但由于 TCP 的捎带 ACK 机制，两者放在一个包里发送了）；
3. 客户端对服务器端的 FIN 进行确认（ NO. 8 ）。

## 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



## 复位报文段

- 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）。
- 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。
- 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 连接的过程中，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  $RT0 = R * \beta$ ， $\beta = 2$ ，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 RT0：

```

Err=M-A
A=A+gErr
D=D+h*(|Err|-D)
RT0=A+4D

```

其中， $A$  和  $D$  分别被初始化为 0 和 3， $RT0$  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， $RT0$  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  $RT0$ 。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  $RT0$ 。

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 不应该被延迟。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)  $W < B^*Tr$  时，影响 TCP 发送数据速率的最直接的因素是滑动窗口的大小，TCP 的流量控制策略（比如超时时窗口设置为1，重复 ACK 时窗口减半）最终都是通过控制窗口大小来控制速率，而慢启动，拥塞避免这些流量控制算法实际上就是控制窗口增长方式的算法，也就是控制的是加速度大小。 (2) 当  $W > B^*Tr$  时，则影响速率的因素是带宽

## 参考文档

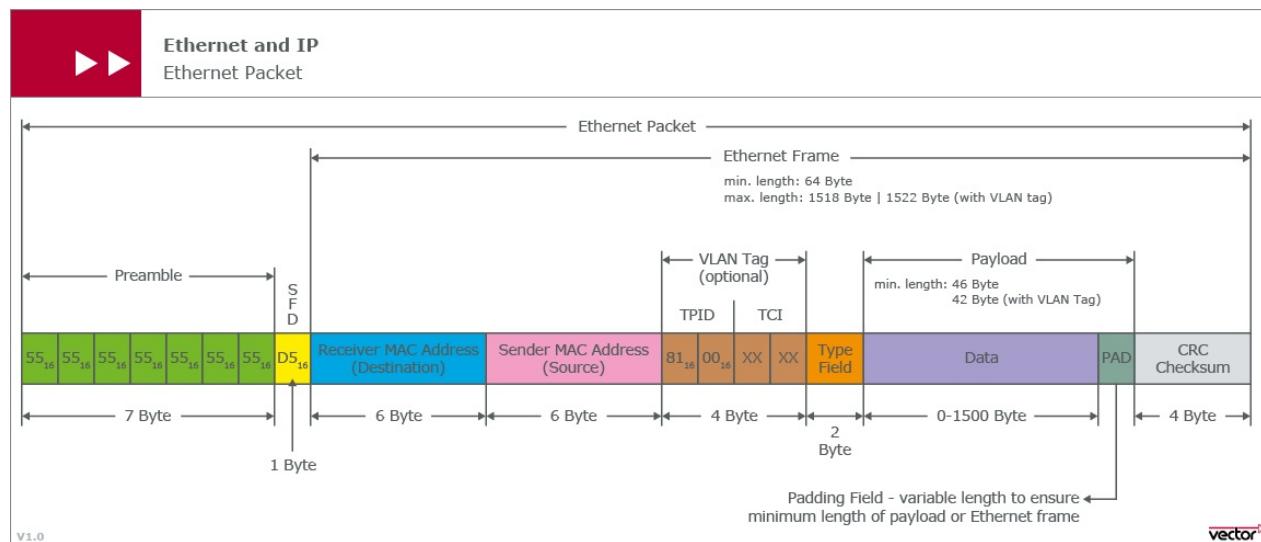
- [Transmission Control Protocol](#)
- [RFC 675 – Specification of Internet Transmission Control Program, December 1974 Version](#)
- [RFC 793 – TCP v4](#)
- [STD 7 – Transmission Control Protocol, Protocol specification](#)
- [RFC 1122 – includes some error corrections for TCP](#)
- [RFC 1323 – TCP Extensions for High Performance \[Obsoleted by RFC 7323\]](#)
- [RFC 1379 – Extending TCP for Transactions—Concepts \[Obsoleted by RFC 6247\]](#)
- [RFC 1948 – Defending Against Sequence Number Attacks](#)
- [RFC 2018 – TCP Selective Acknowledgment Options](#)
- [RFC 5681 – TCP Congestion Control](#)
- [RFC 6247 – Moving the Undeployed TCP Extensions RFC 1072, RFC 1106, RFC 1110, RFC 1145, RFC 1146, RFC 1379, RFC 1644, and RFC 1693 to Historic Status](#)
- [RFC 6298 – Computing TCP's Retransmission Timer](#)
- [RFC 6824 – TCP Extensions for Multipath Operation with Multiple Addresses](#)
- [RFC 7323 – TCP Extensions for High Performance](#)
- [RFC 7414 – A Roadmap for TCP Specification Documents](#)

# 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。



图片来源[vector.com](http://vector.com) - IEEE Ethernet MAC and VLAN

## 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，12位的值 0x000 和 0xFFFF 为保留值，其他的值都可用来做 VLAN 的识别符，因此第一个不足之处就是最多只支持 4094 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

- Neutron 理解 (2): 使用 Open vSwitch + VLAN 组网 [Netruon Open vSwitch + VLAN Virutal Network]
- UnitedStack® UNP 技术白皮书 - VLAN
- QinQ vs VLAN vs VXLAN
- IEEE 802.1Q - Wikipedia

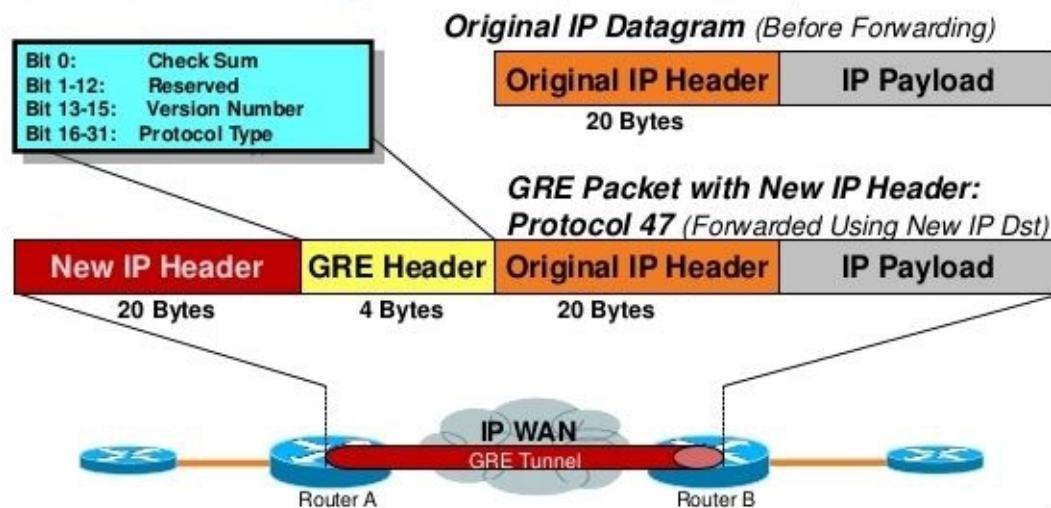
# Overlay

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 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 HAS

## Generic Routing Encapsulation (GRE)

GRE提供了IP in IP的封装技术:

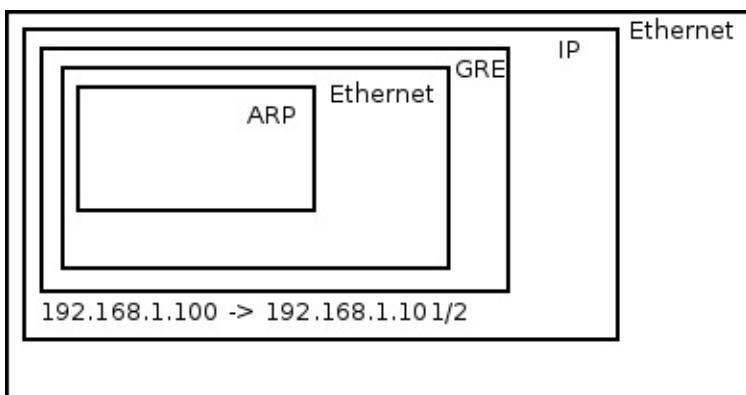
### GRE Tunnel Encapsulation (RFC 2784)



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。

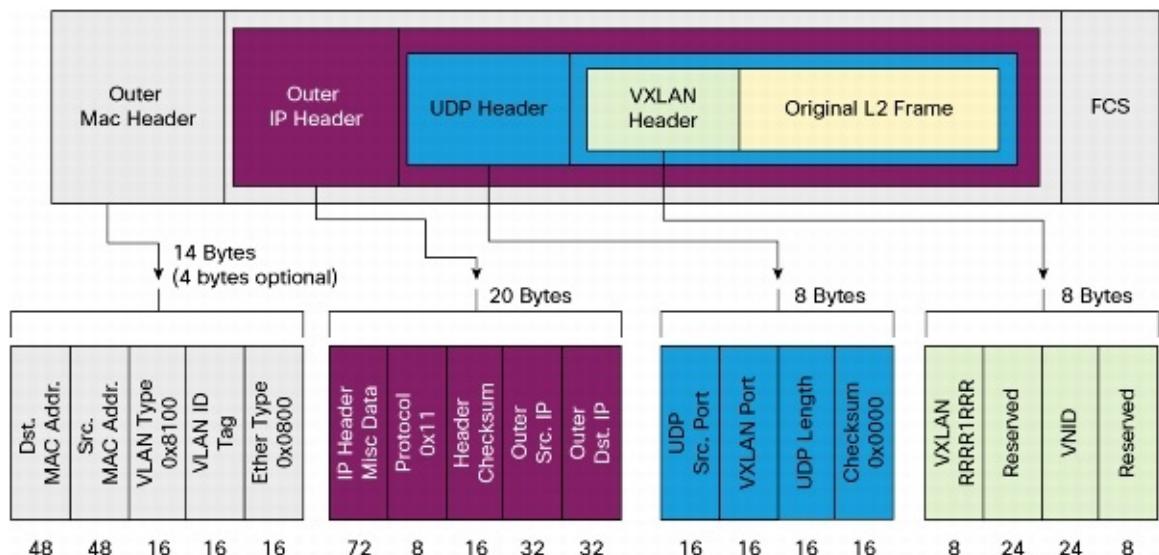
可见，使用 GRE 可以比使用 VxLAN 每次可以多传输  $1448 - 1422 = 26$  byte 的数据。



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。

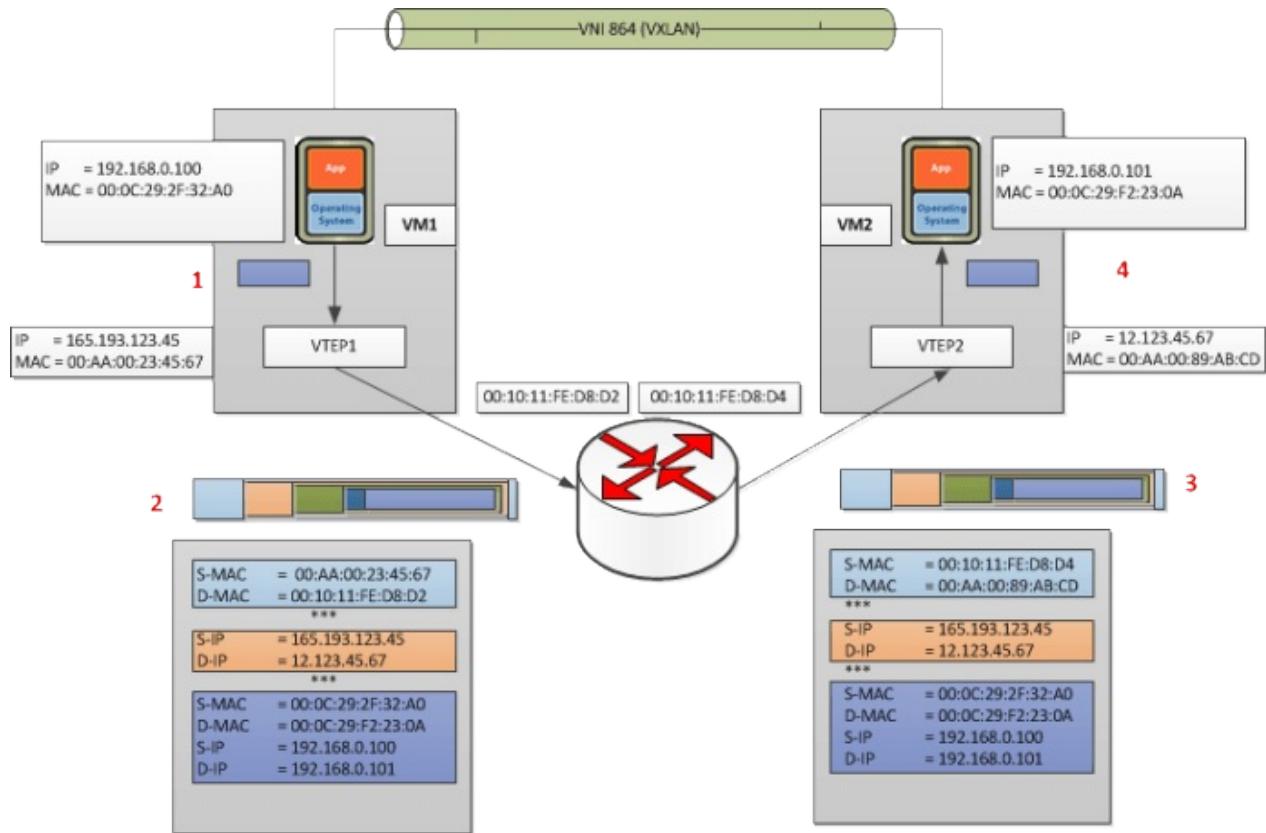
## VXLAN

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。

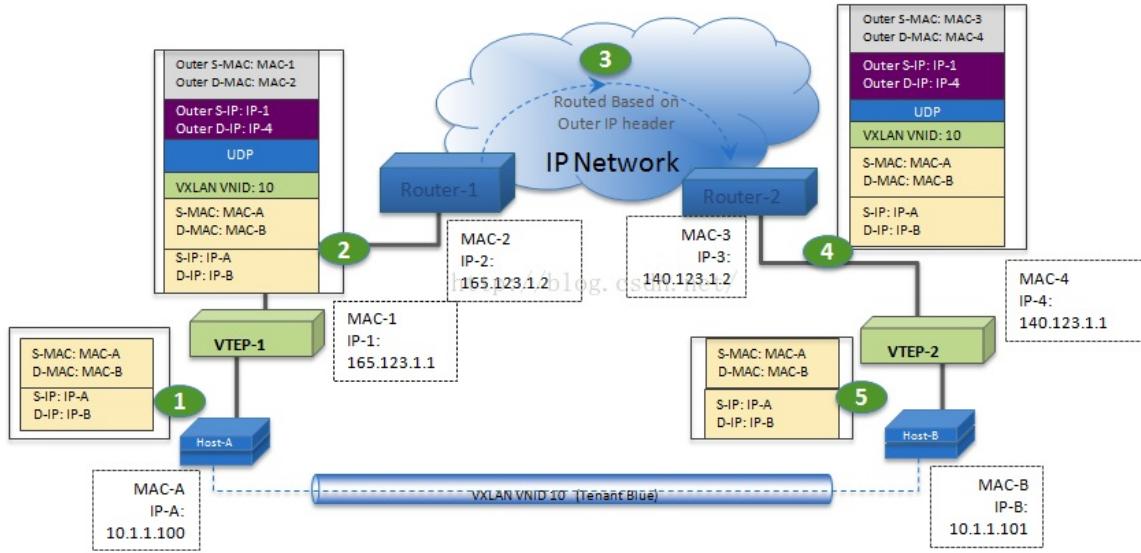
### 开启或关闭vxlan offload的方法

```
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。



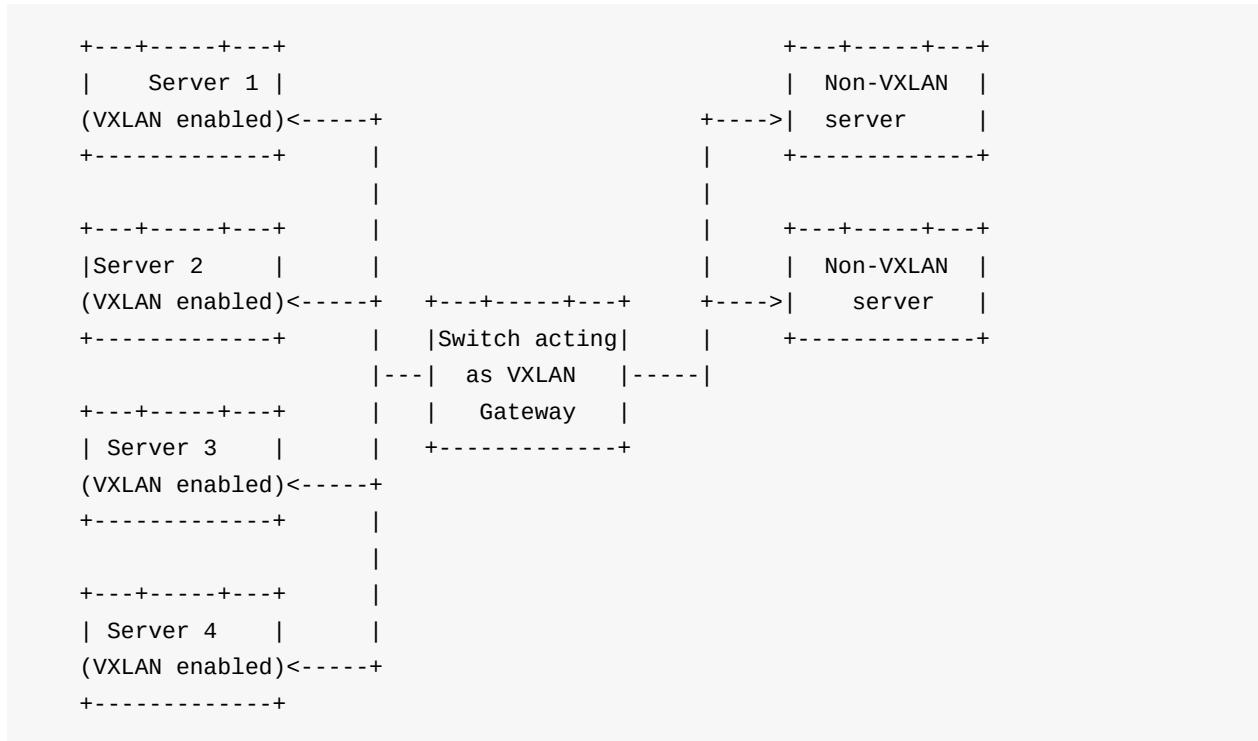
(图片来自[csdn](#))

## 不同VXLAN ID转发

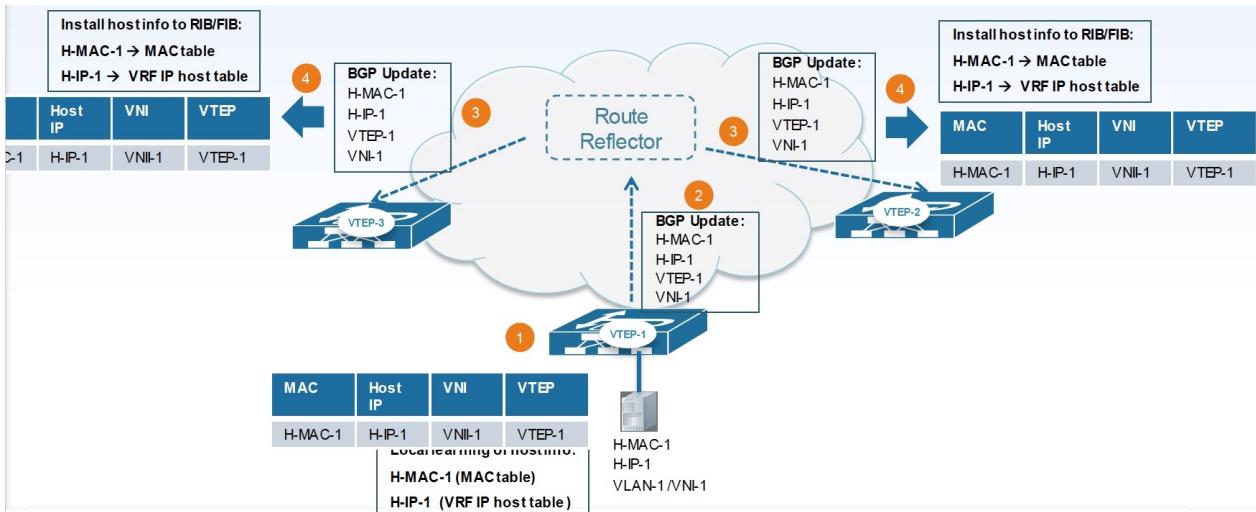
大致转发过程与上面类似，所不同的是需要报文在所属vtep或者vxlan gateway处来转换vxlan id（源和目的处都需要做这个转换）。

## VXLAN与非VXLAN（如VLAN）转发

需要VXLAN Gateway来转换vxlan vni和vlan id：

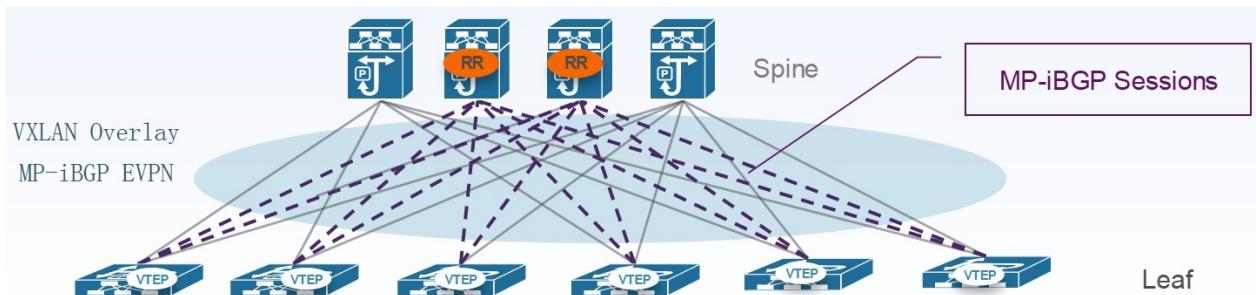


## MPBGP EVPN VXLAN



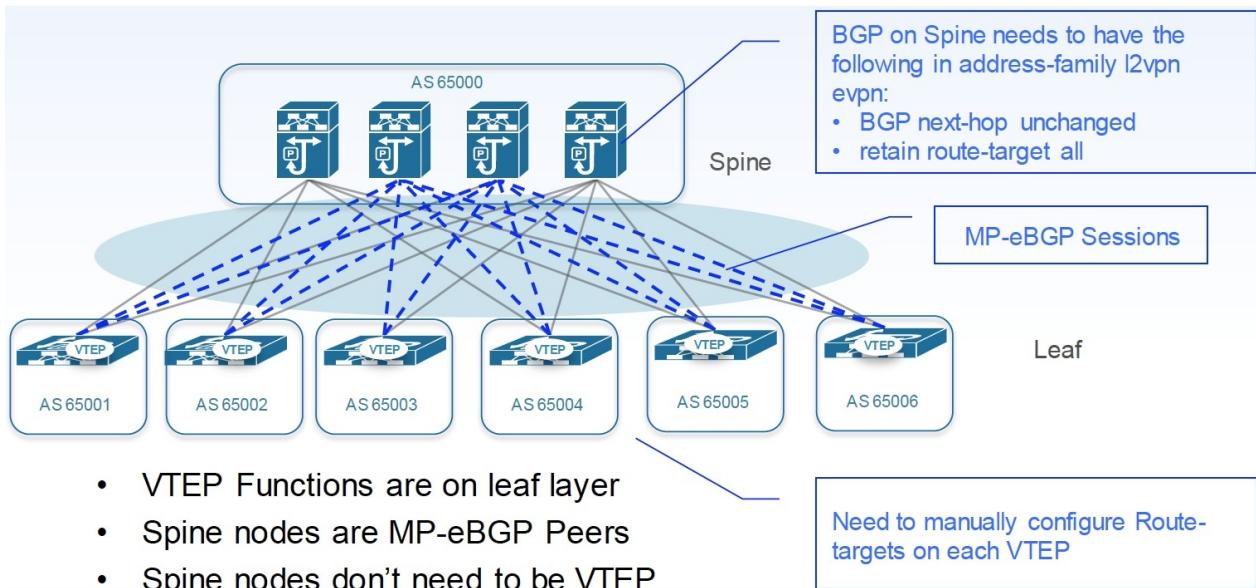
每一个 VTEP 将作为一个 BGP Speaker，向其他 VTEP 通过 EVPN 发送本地的 MAC、IP 信息，BGP RR 可以避免 BGP 的 Full-Mesh，提高通信效率。得益于控制平面，每个 VTEP 将可作为分布式网关、可以抑制 ARP 广播、可以将广播或组播通过单播复制来提升效率、可以对 VTEP 进行认证。

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。

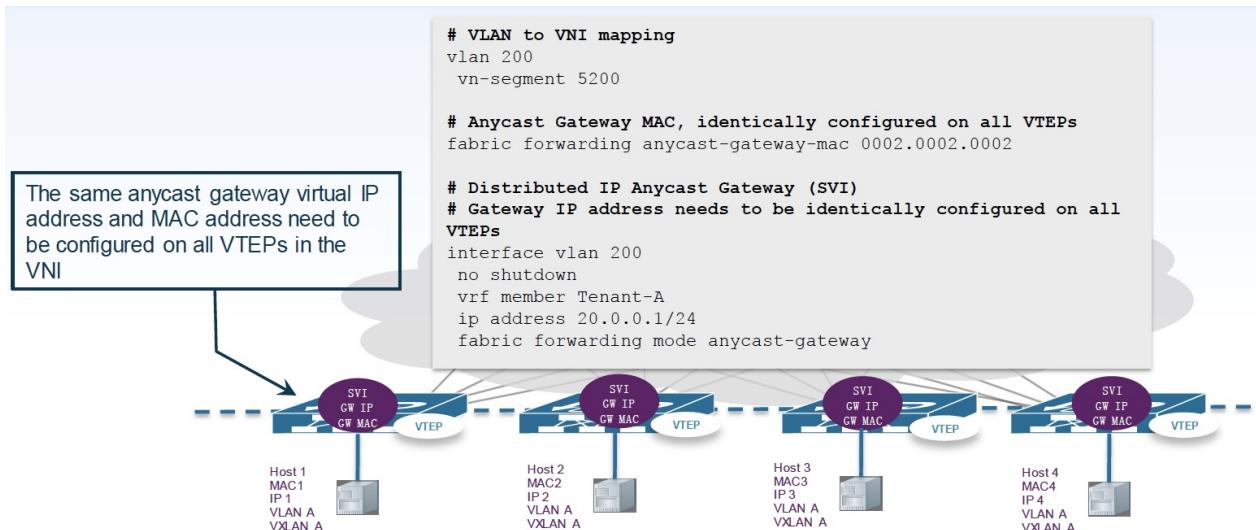


## Distributed Anycast Gateway

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。

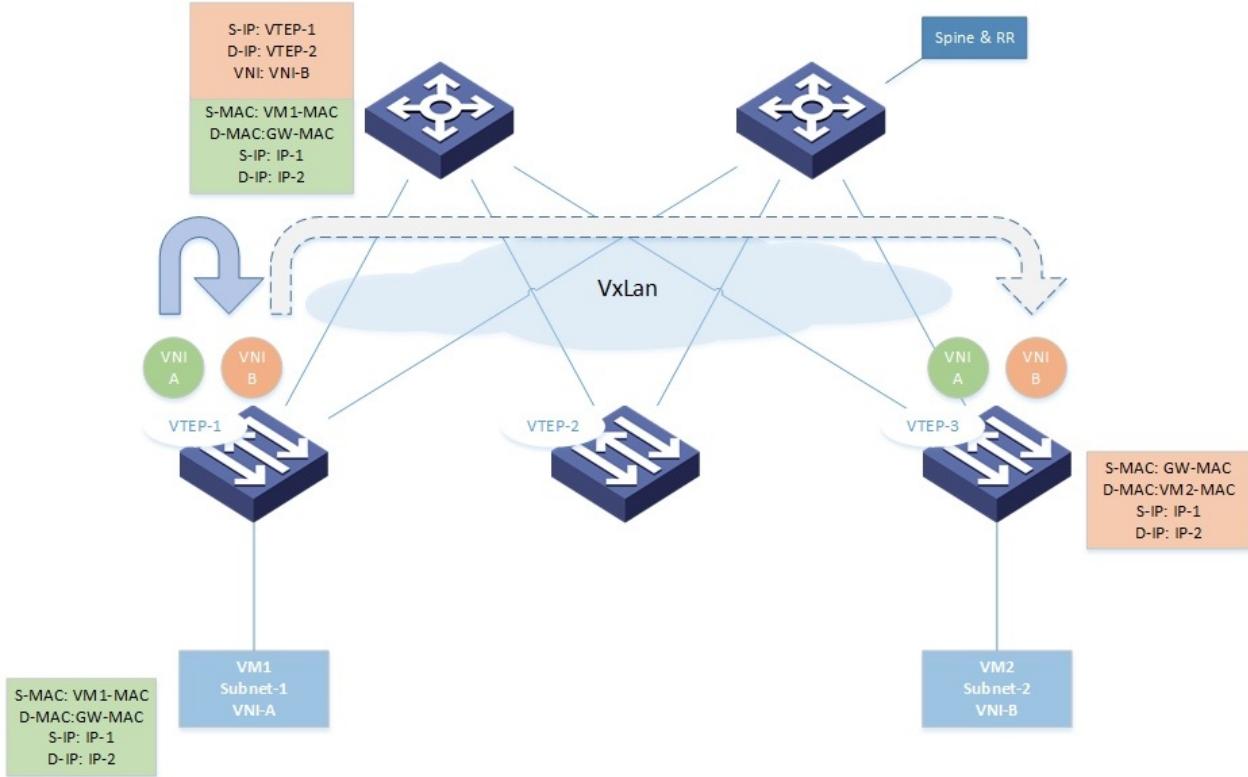
目前一种实践是使用 Anycast Gateway 技术，每个 VTEP 上均配置相同的 vIP 和 vMAC，如图：



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。

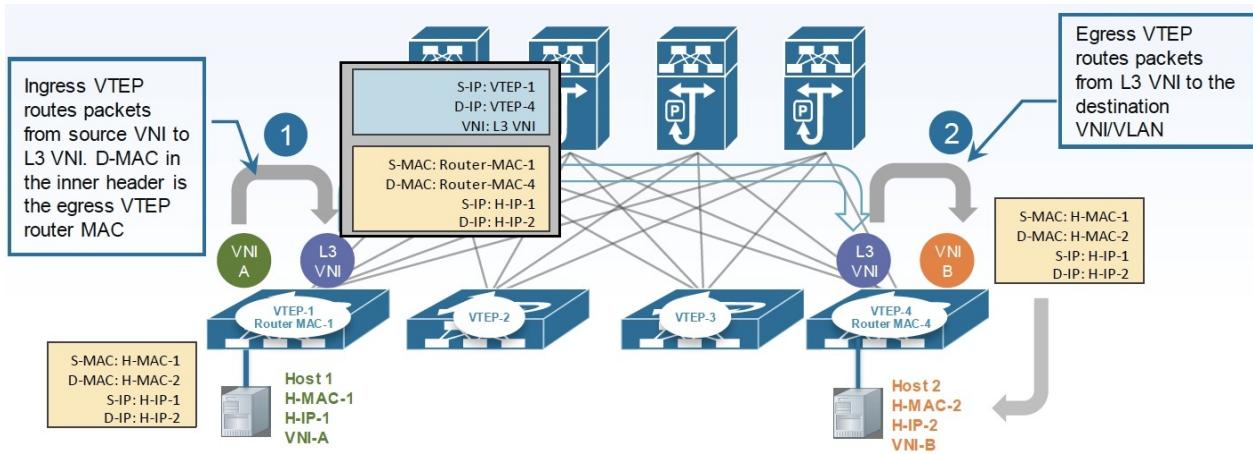
当虚拟机需要回复时，路径完全反过来，即在 VTEP-2 上完成 VXLAN 封包和设置 VNI 为 VNI-A。所以这个过程是非对称的。

这种实现存在一些显而易见的问题：

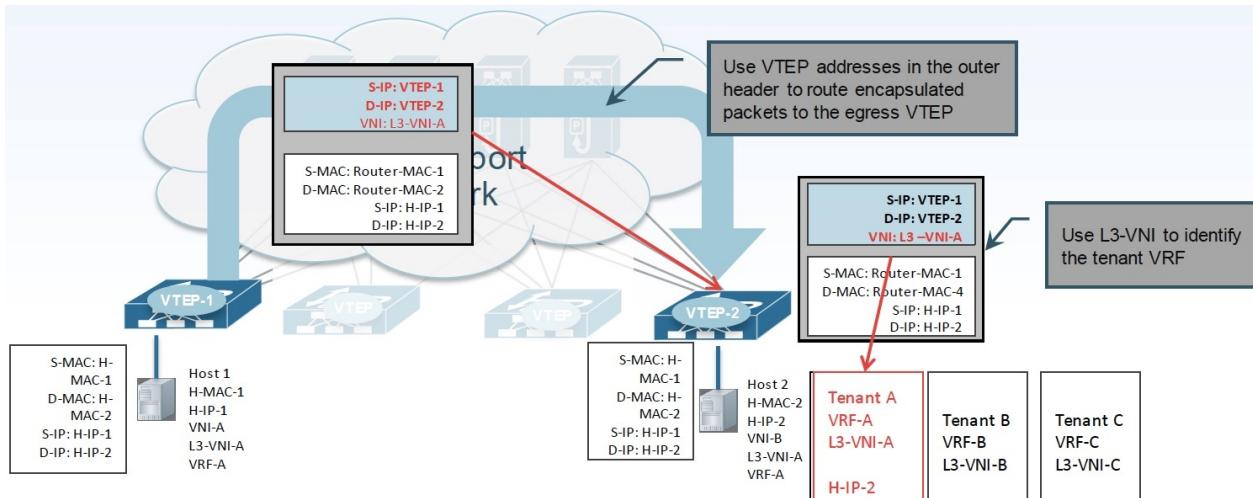
- 所有的 VTEP 必须配置上所有的 VXALN VNI，否则不同 VNI 通信会存在问题；
- 所有的 VTEP 必须获整个 Fabric 完整的 Host tables 信息，否则无法完成完路由。

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。



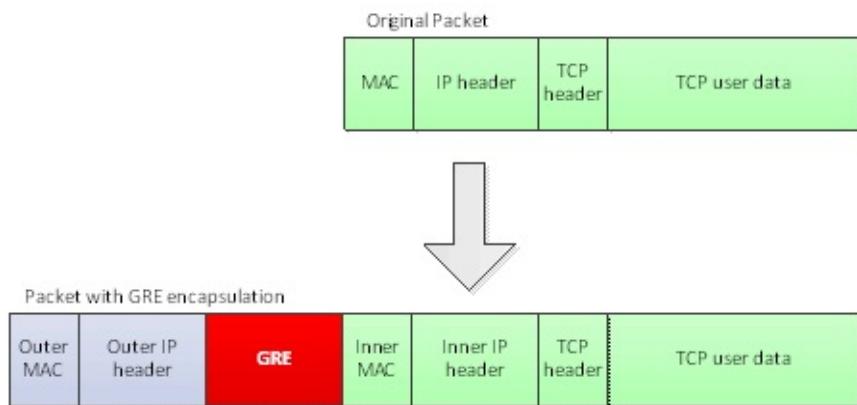
从上面的分析可以得知，Symmetric IRB 最大的好处一是不需要所有 VTEP 均配置所有 VNI，二是不需要所有的 VTEP 知道整个 Fabric 的完整 Host tables 信息。但是，这是建立在不是最差情况的前提，如果说恰好每个租户都在每个 VTEP 下具有虚拟机，或着整个网络只有一个租户，那么网络可能产生退化。Symmetric IRB 的主要优化场景是针对多租户的。



## 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>

# **SNMP**

# **LLDP**

# 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
```

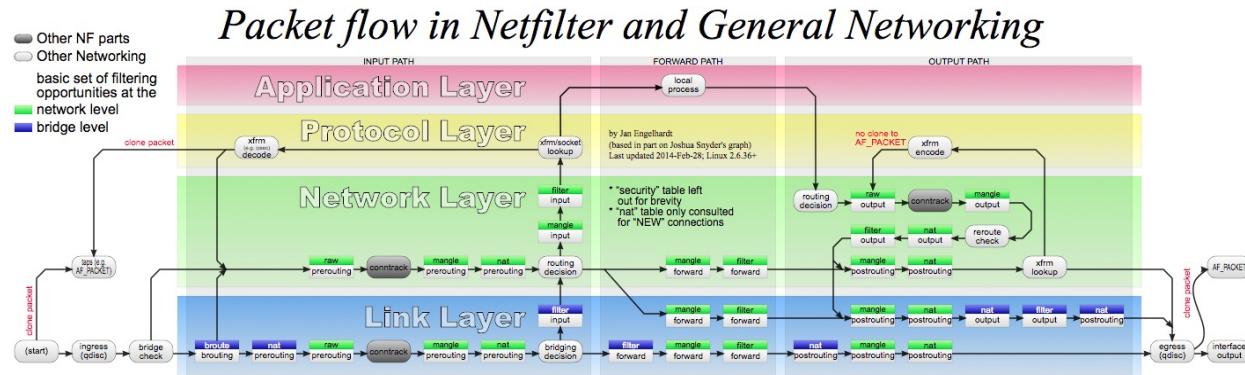
# iptables/netfilter

`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` 表 用于网络地址转换，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` 表 用于对报文进行修改，内

建 PREROUTING 、 INPUT 、 FORWARD 、 OUTPUT 和 POSTROUTING 等链

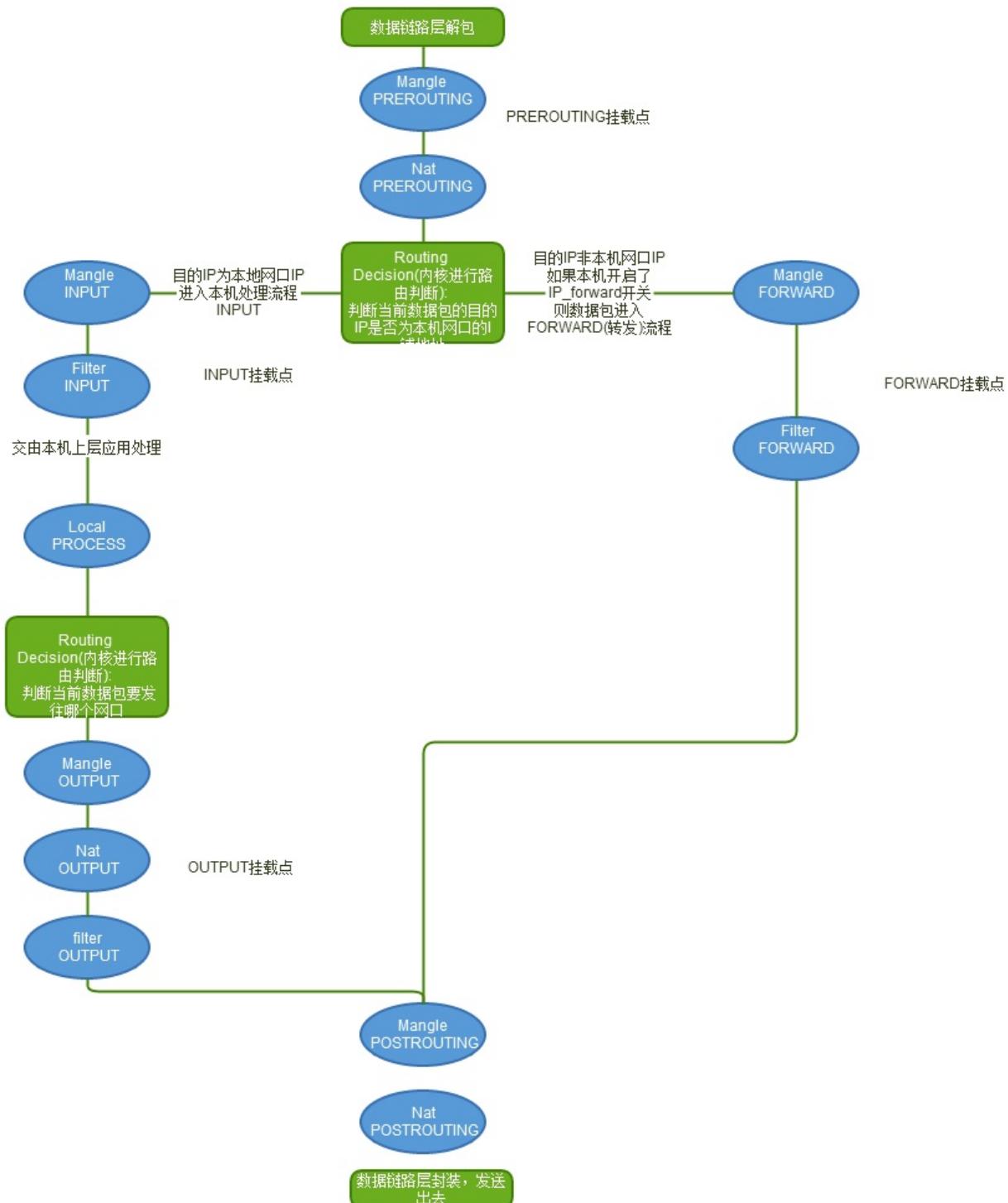
- security表 用于根据安全策略处理数据包，内建 INPUT 、 FORWARD 和 OUTPUT 链

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

- 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)
- 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
- 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](#)
- [netfilter.org](#)
- [archlinux wiki - iptables](#)
- [nftables wiki](#)
- [Linux数据包路由原理、Iptables/netfilter入门学习](#)

# 负载均衡

## Ivs

Linux Virtual Server ( lvs )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。 lvs 包括 ipvs 内核模块和 ipvsadm 用户空间命令行工具两部分。

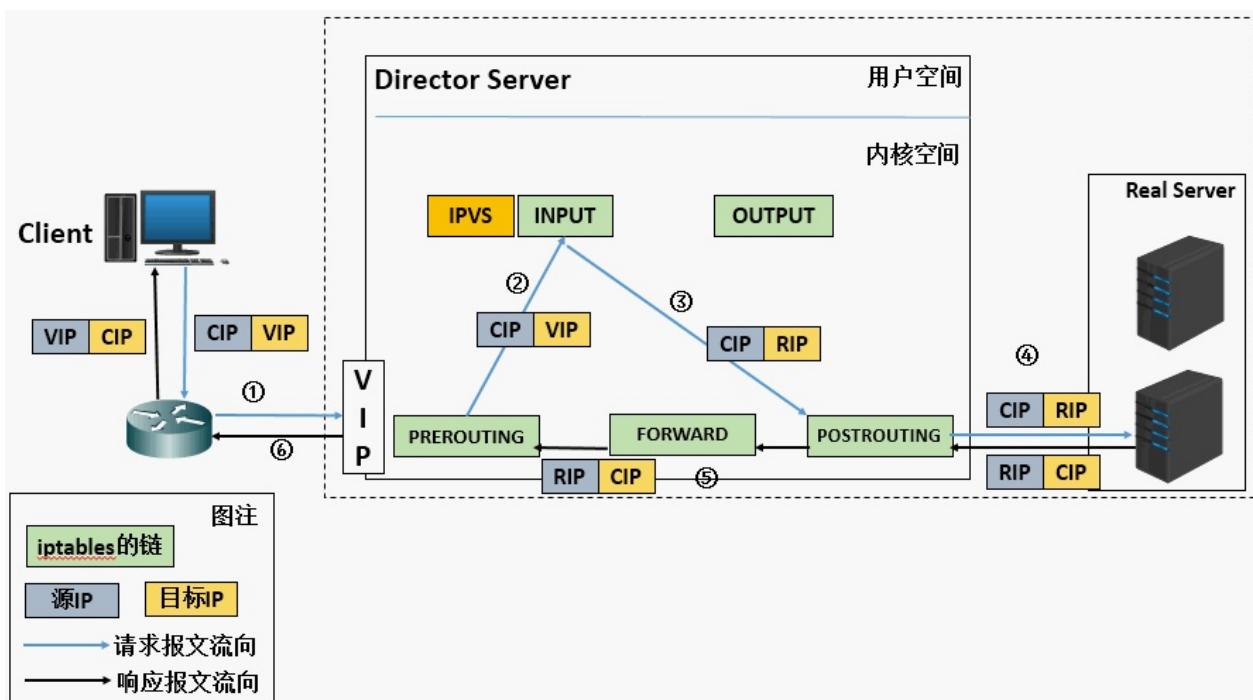
在 lvs 中，节点分为 Director Server 和 Real Server 两个角色，其中 Director Server 是负载均衡器所在节点，而 Real Server 则是后端服务节点。当用户的请求到达 Director Server 时，内核 netfilter 机制的 PREROUTING 链会将发往本地 IP 的包转发给 INPUT 链（也就是 ipvs 的工作链），在 INPUT 链上， ipvs 根据用户定义的规则对数据包进行处理（如修改目的IP和端口等），并把新的包发送到 POSTROUTING 链，进而再转发给 Real Server 。

## 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

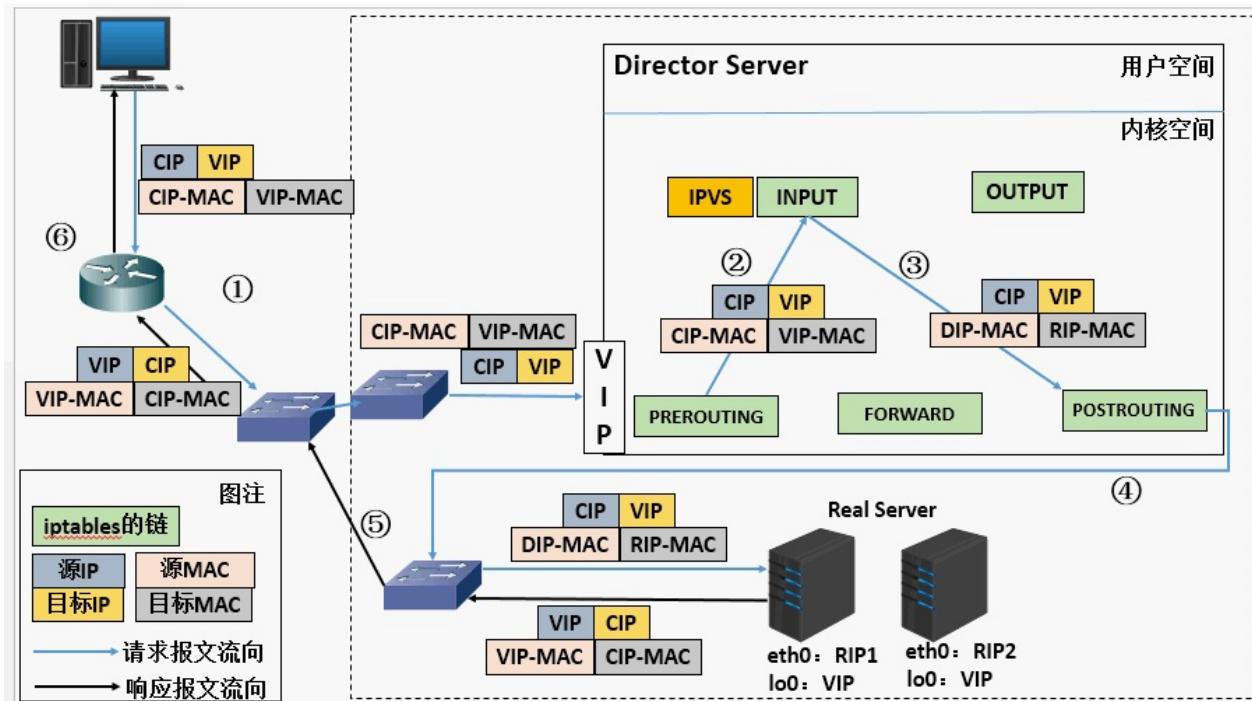
- Director Server 必须作为 Real Server 的网关，并且它们必须处于同一个网段内
- 不需要 Real Server 做任何特殊配置
- 支持端口映射
- 请求和响应都需要经过 Director Server ，易称为性能瓶颈



## 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

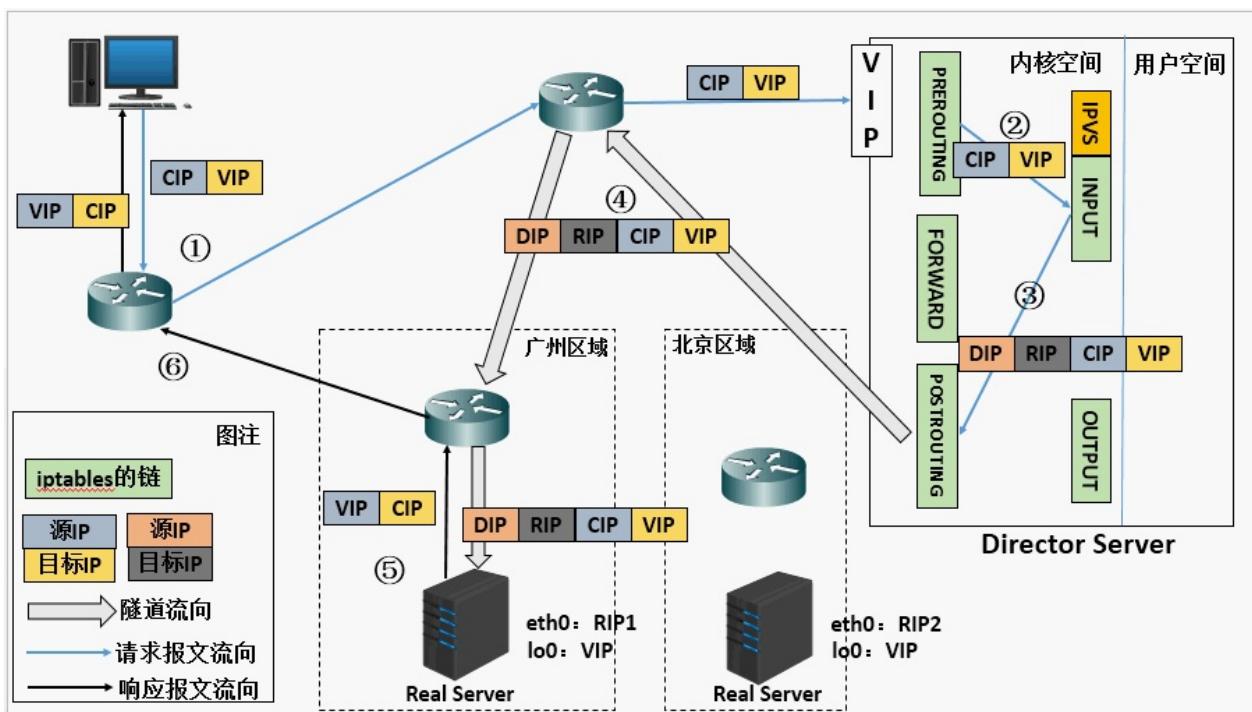
- 需要在 Real Server 的 lo 上配置 vip , 并配置 arp\_ignore 和 arp\_announce 忽略对 vip 的 ARP 解析请求
- Director Server 和 Real Server 必须在同一个物理网络内，二层可达
- 虽然所有请求包都会经过 Director Server , 但响应报文不经过 , 有性能上的优势



## 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

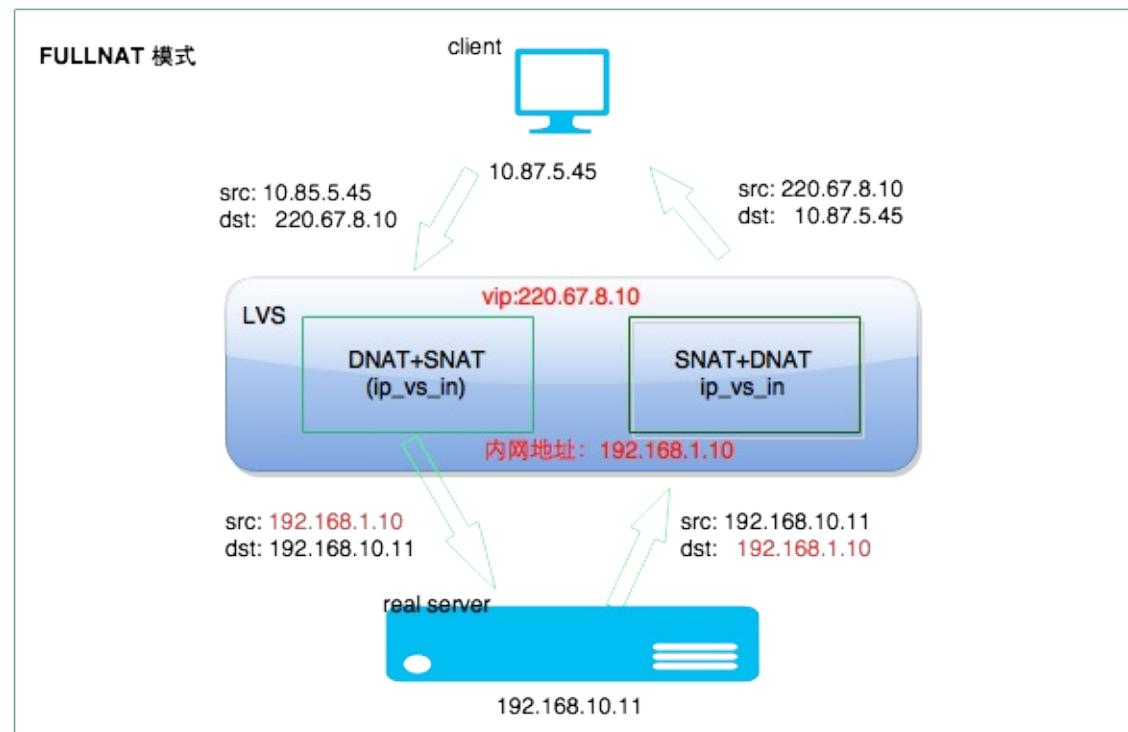
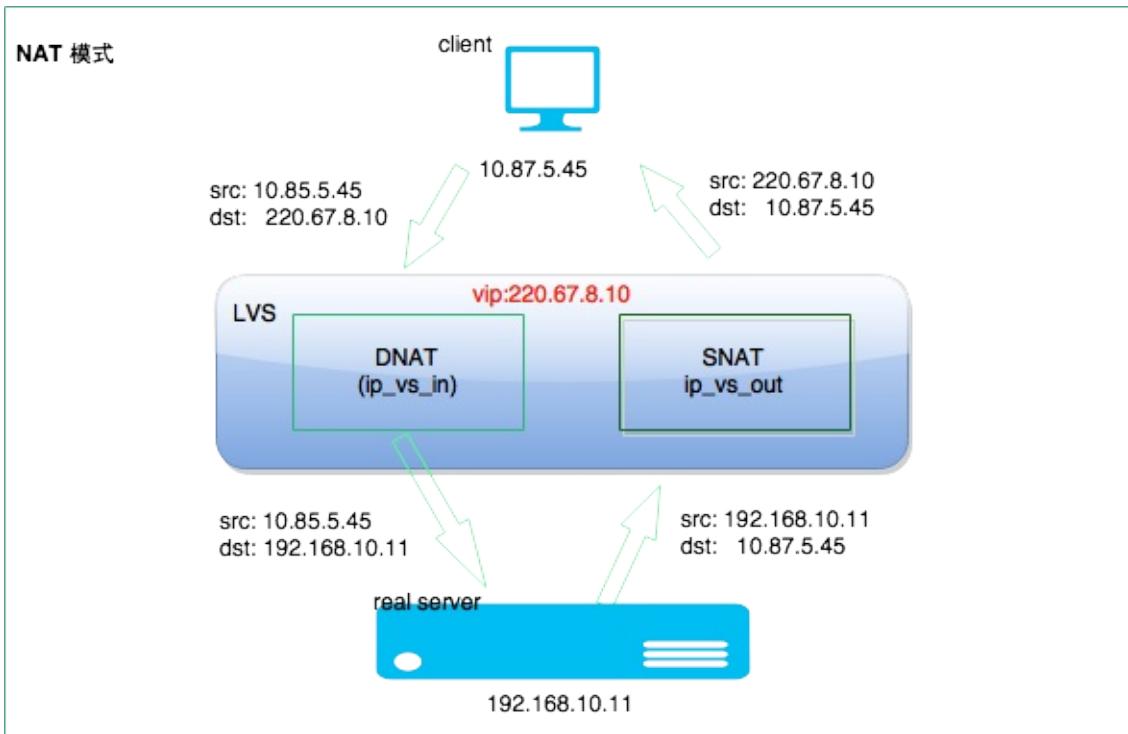
- Real Server 需要在 lo 上配置 vip , 但不需要 Director Server 作为网关
- 不支持端口映射



## 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 )

## lvs 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。

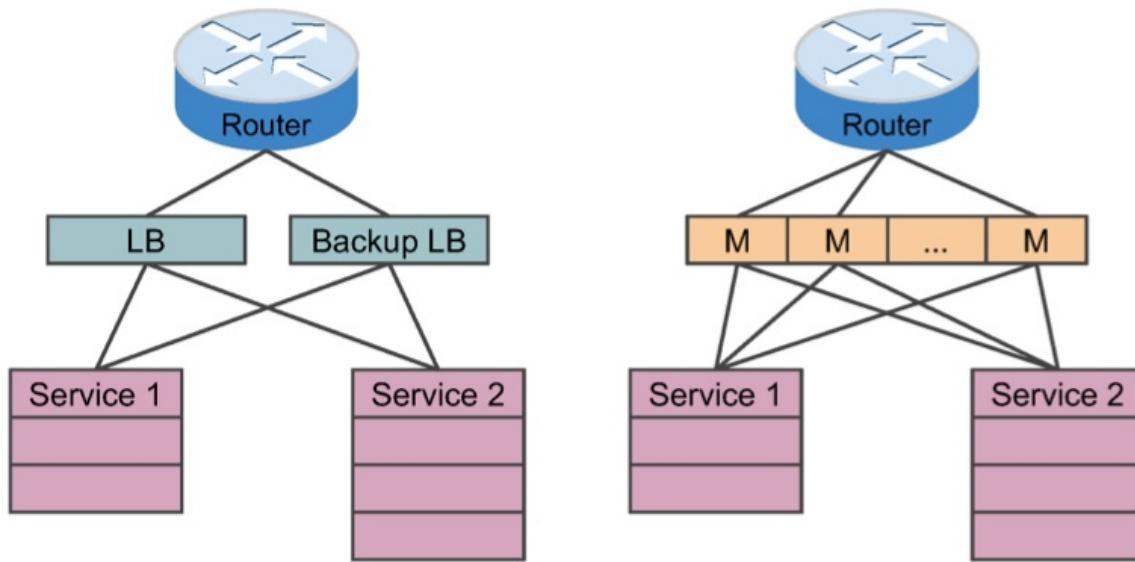


Figure 1: Hardware load balancer and Maglev.

- 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

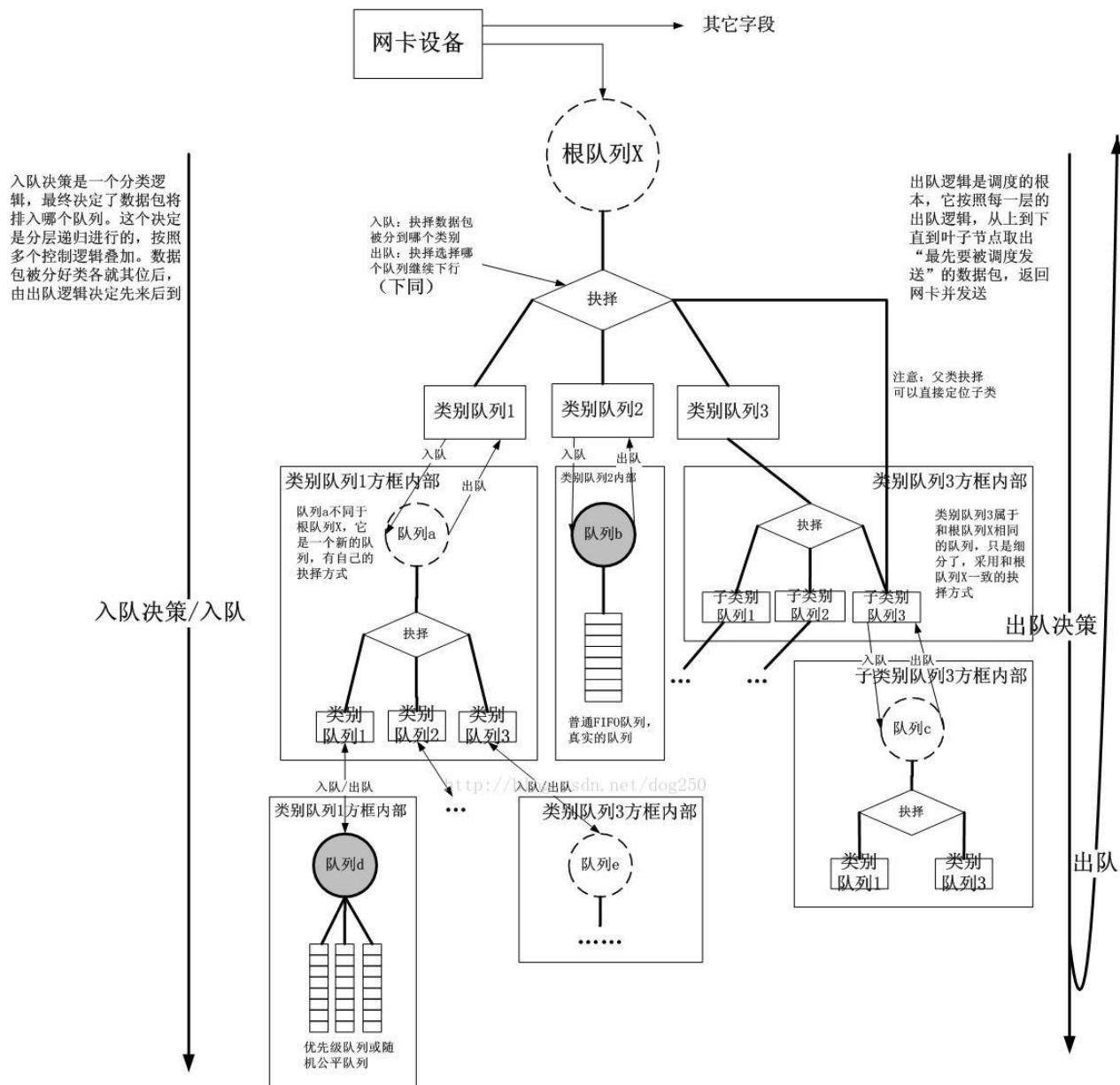
## 参考文档

- <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

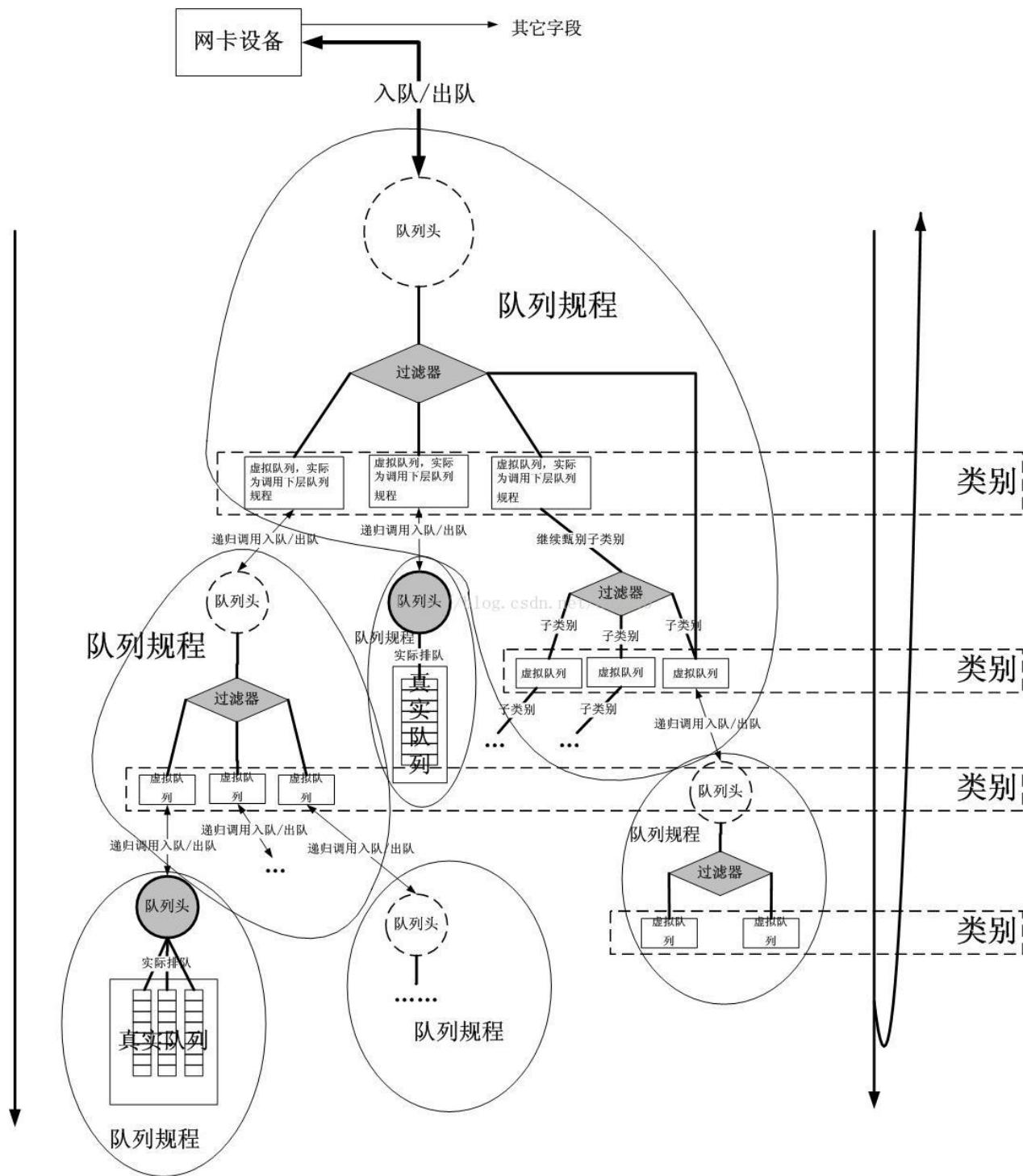


真实的队列，数据包将被排队



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

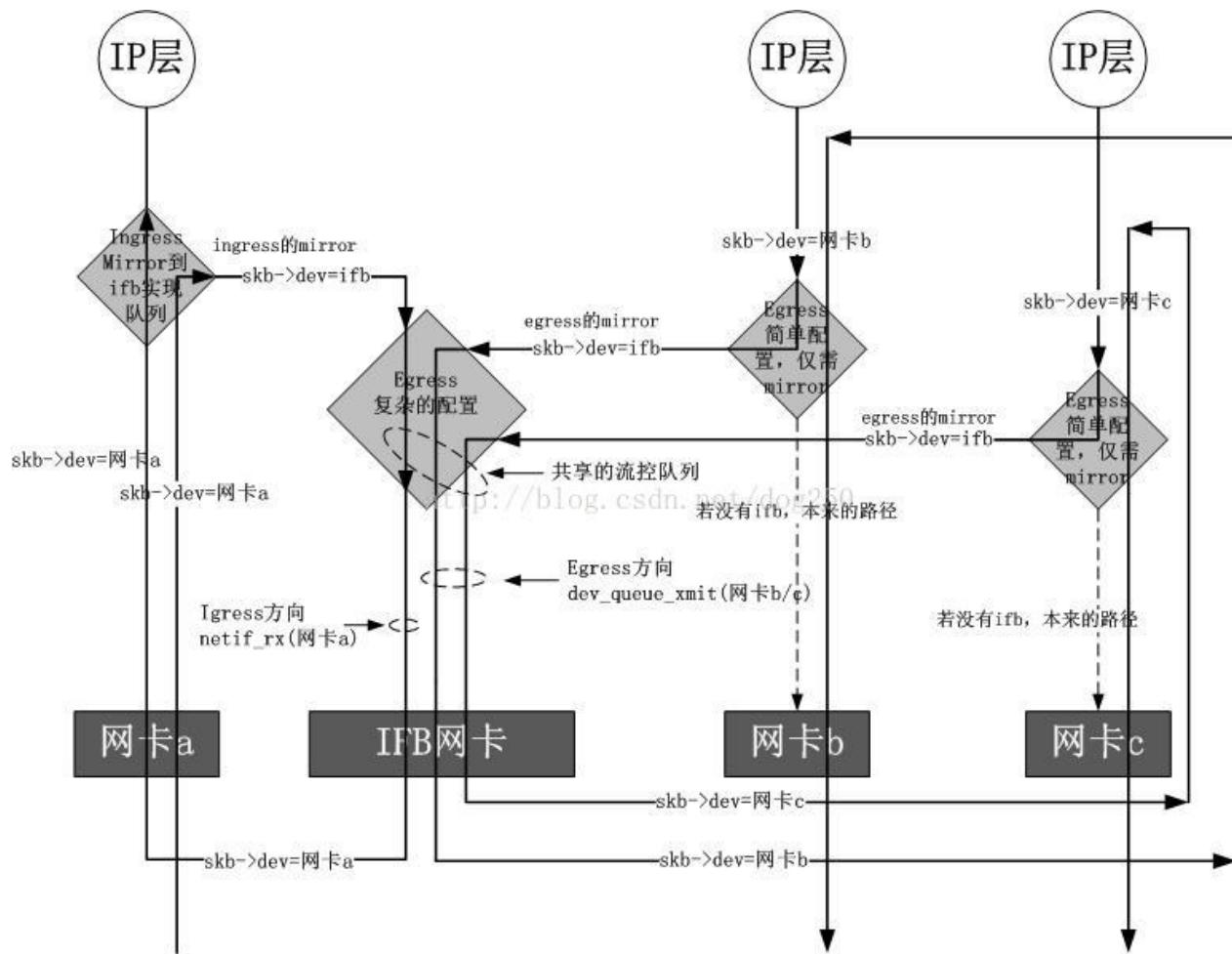
- 有分类`qdisc`（可以包括多个队列）

- `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：

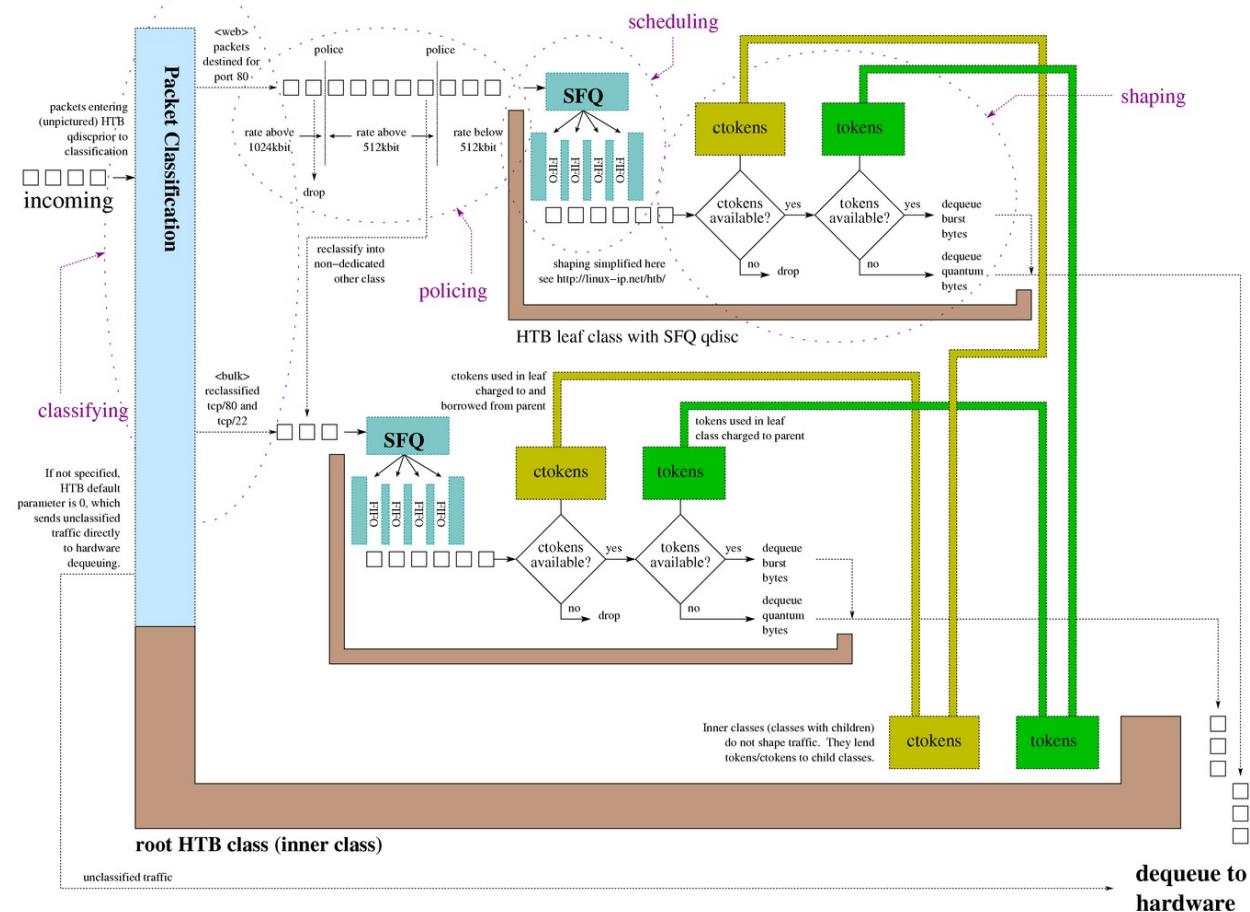
- `u32` : 根据协议、`IP`、端口等过滤数据包
- `fwmark` : 根据 `iptables MARK` 来过滤数据包
- `tos` : 根据 `tos` 字段过滤数据包

## 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
```

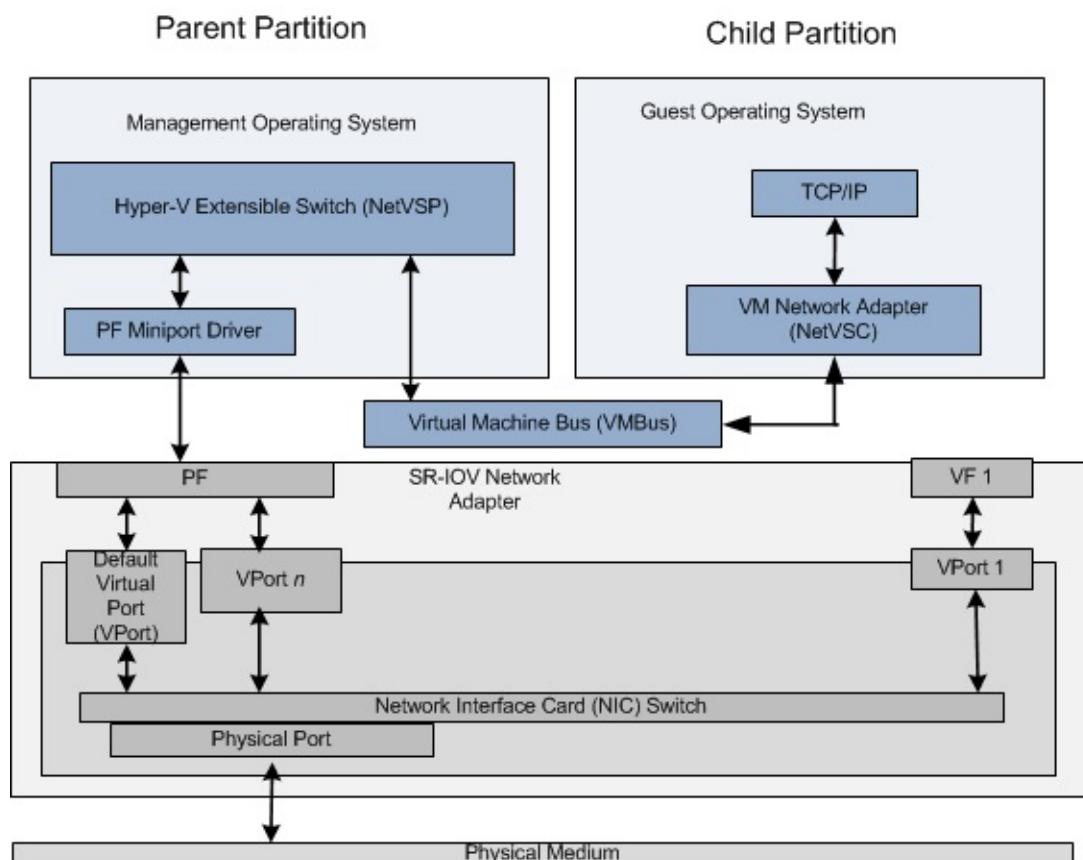
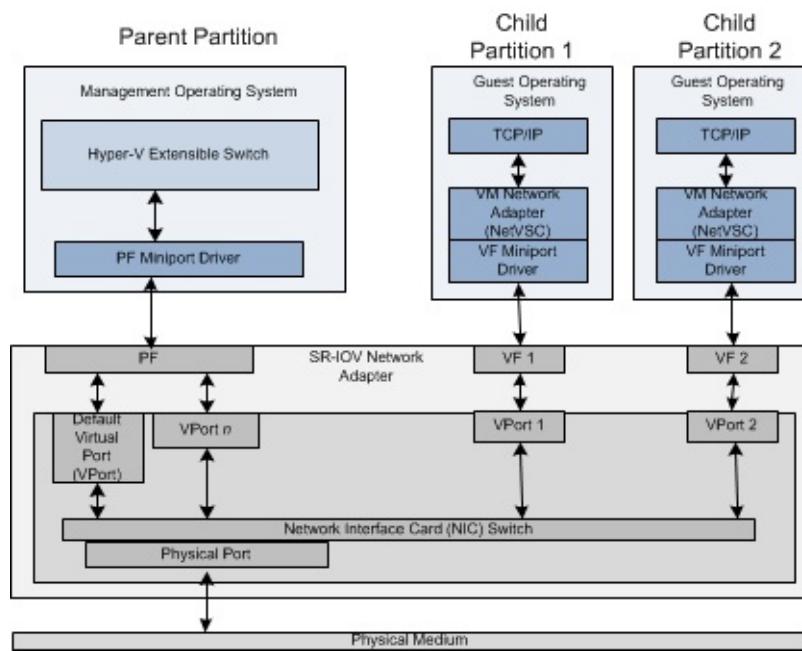
## 参考文档

- [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



图片来源 Microsoft - SR-IOV Architecture

## SR-IOV要求

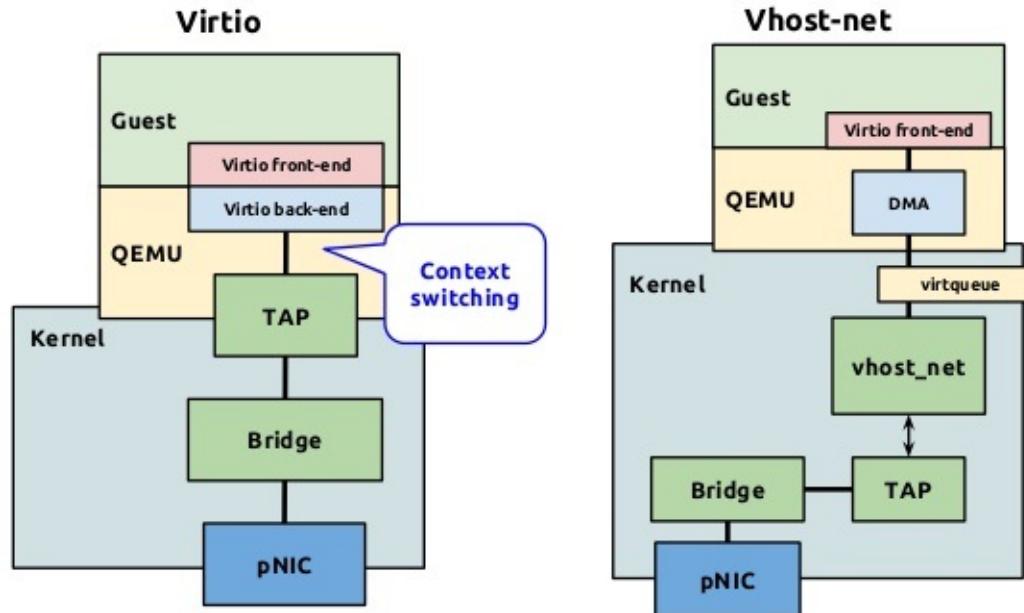
- CPU 必须支持 IOMMU (比如英特尔的 VT-d 或者 AMD 的 AMD-Vi , Power8 处理器默认支持 IOMMU )
- 固件 Firmware 必须支持 IOMMU

- 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

架构上的比较（以网卡为例）

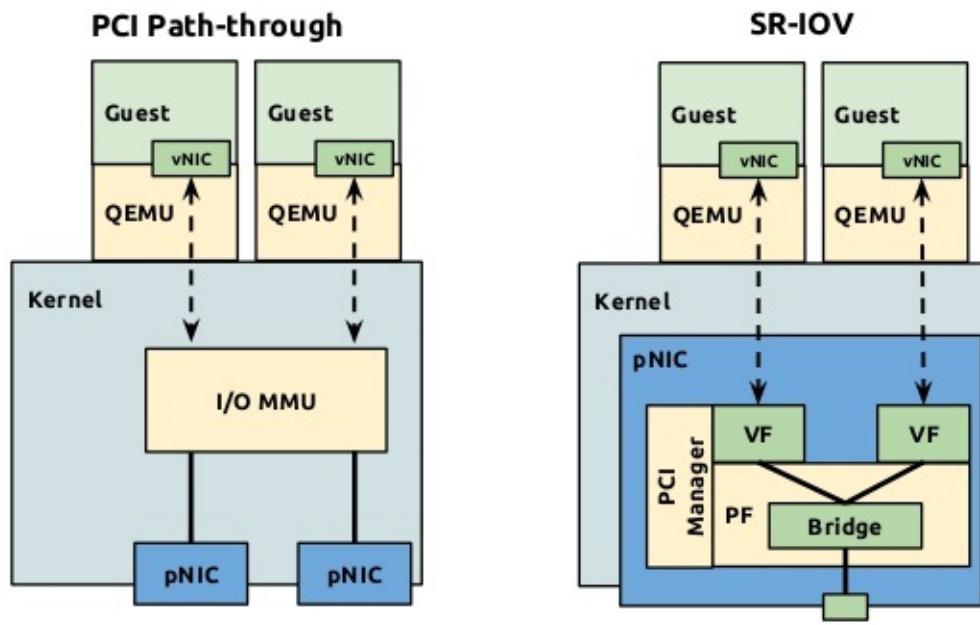
### ● Networking - vhost-net



CANONICAL



- Networking - vhost-net



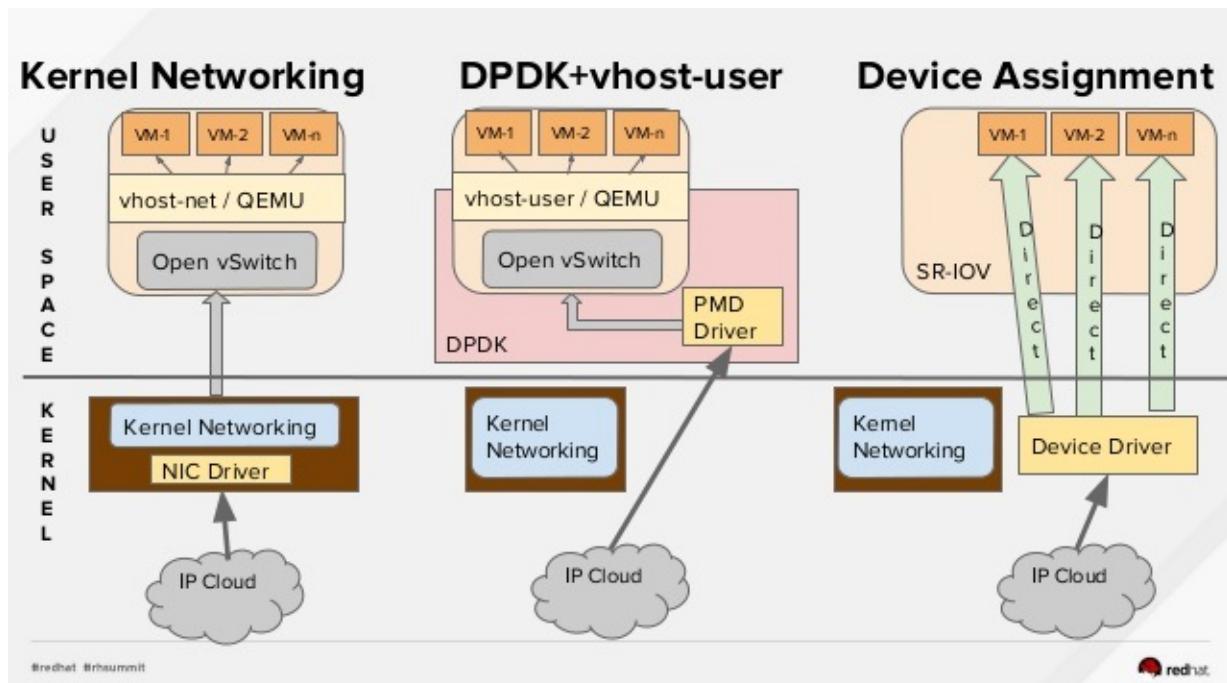
CANONICAL

## Virtio 和 Pass-Through 的详细比较

<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 ovs-vsctl add-port br0 &lt;phys-intfc&gt; -vSwitch ovs-ofctl – 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>

图片来源 [slideshare - Kvm performance optimization for ubuntu、KVM 介绍（4）：I/O 设备直接分配和 SR-IOV \[KVM PCI/PCIe Pass-Through SR-IOV\]](#)

## SR-IOV vs DPDK

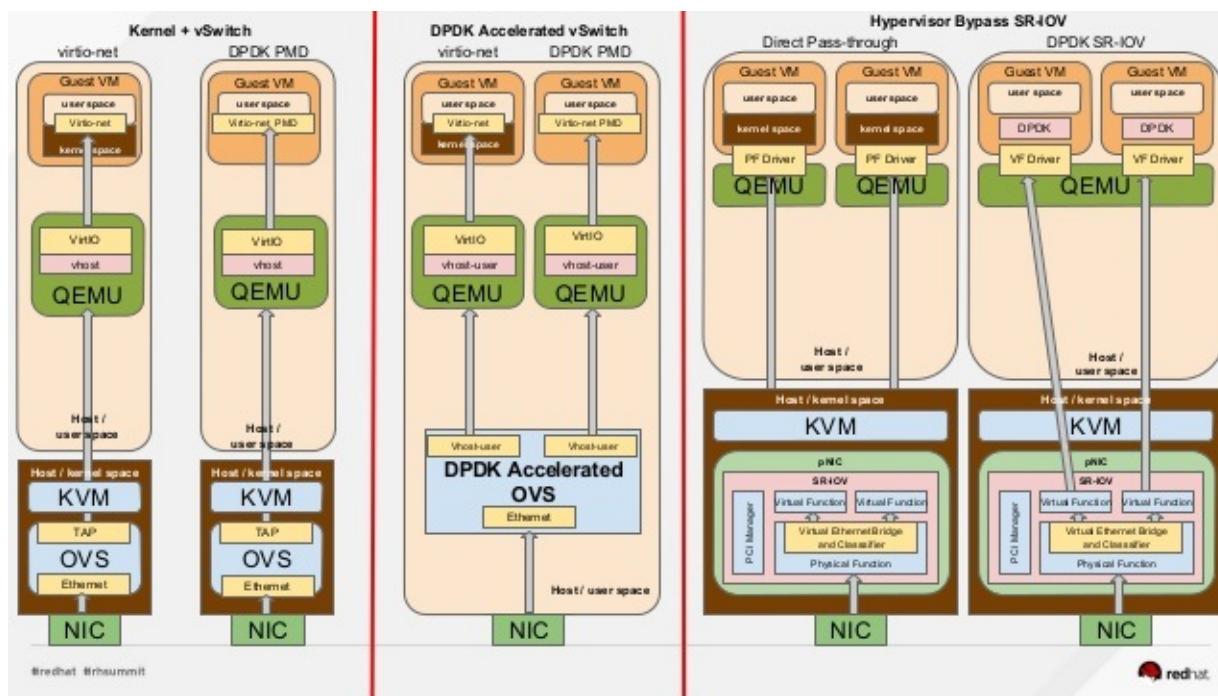


# Comparison

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>	

Bredhat #rhsummit

redhat



## 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>
  </mac>
</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

- [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_l3mdev_accept` and `udp_l3mdev_accept` sysctl options:

```
sysctl -w net.ipv4.tcp_l3mdev_accept=1
sysctl -w net.ipv4.udp_l3mdev_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

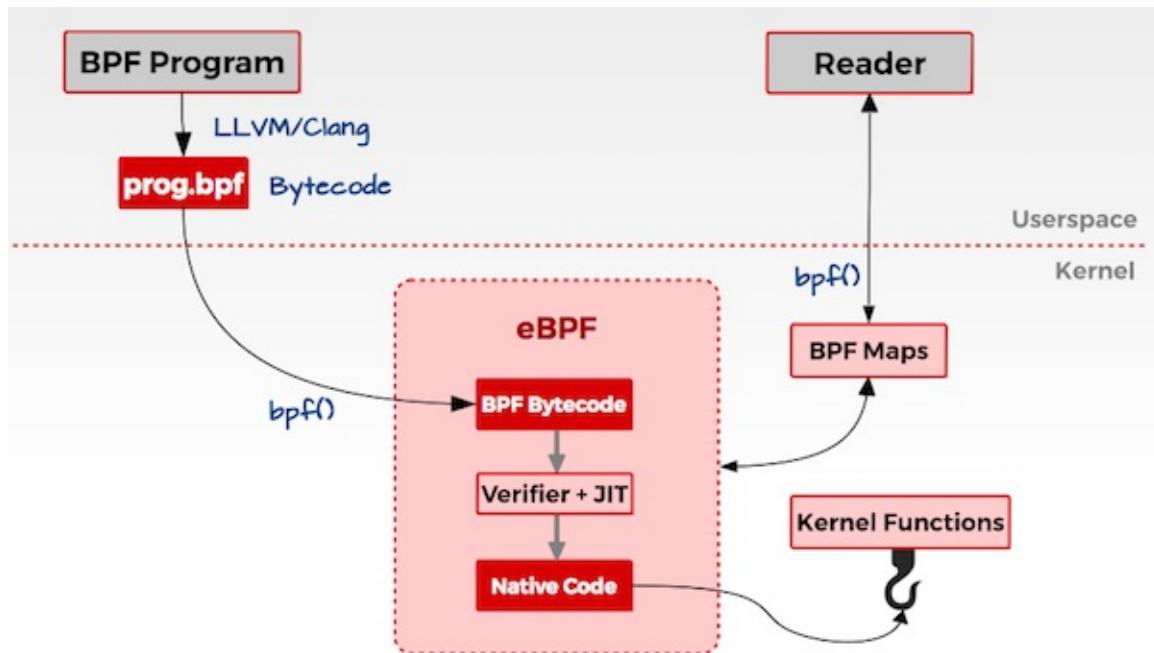
- [Linux kernel documentation](#)

# 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

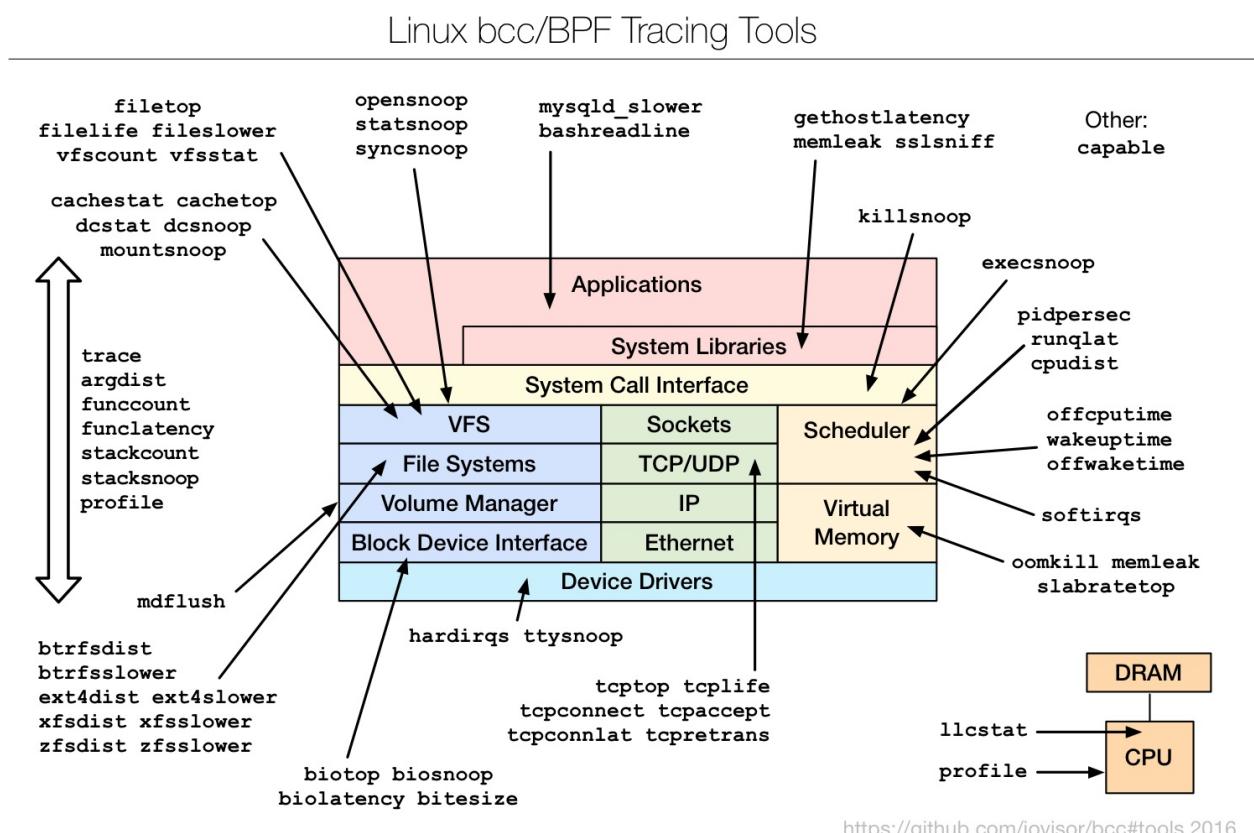
## 参考

- 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 [iovisor/bcc](#)。

## BCC包括的一些工具



## 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       llcstat       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 tpls
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[bbc/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

- [iovisor.org](#)
- [iovisor.org - ebpf](#)
- [iovisor.org - xdp](#)
- [iovisor/bpf-docs](#)
- [kernel.org - fitler](#)
- [iovisor-lc-bof-2016](#)
- [BPF Internals – II](#)
- [Dive into BPF: a list of reading material](#)
- [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
```

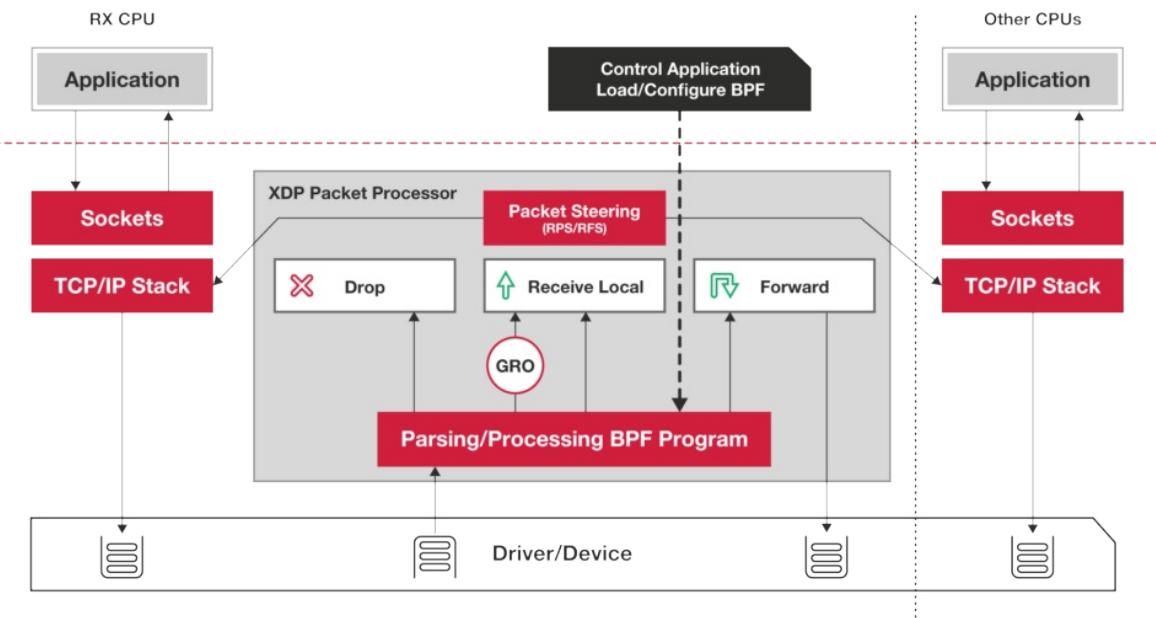
输出如下所示：

```
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

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

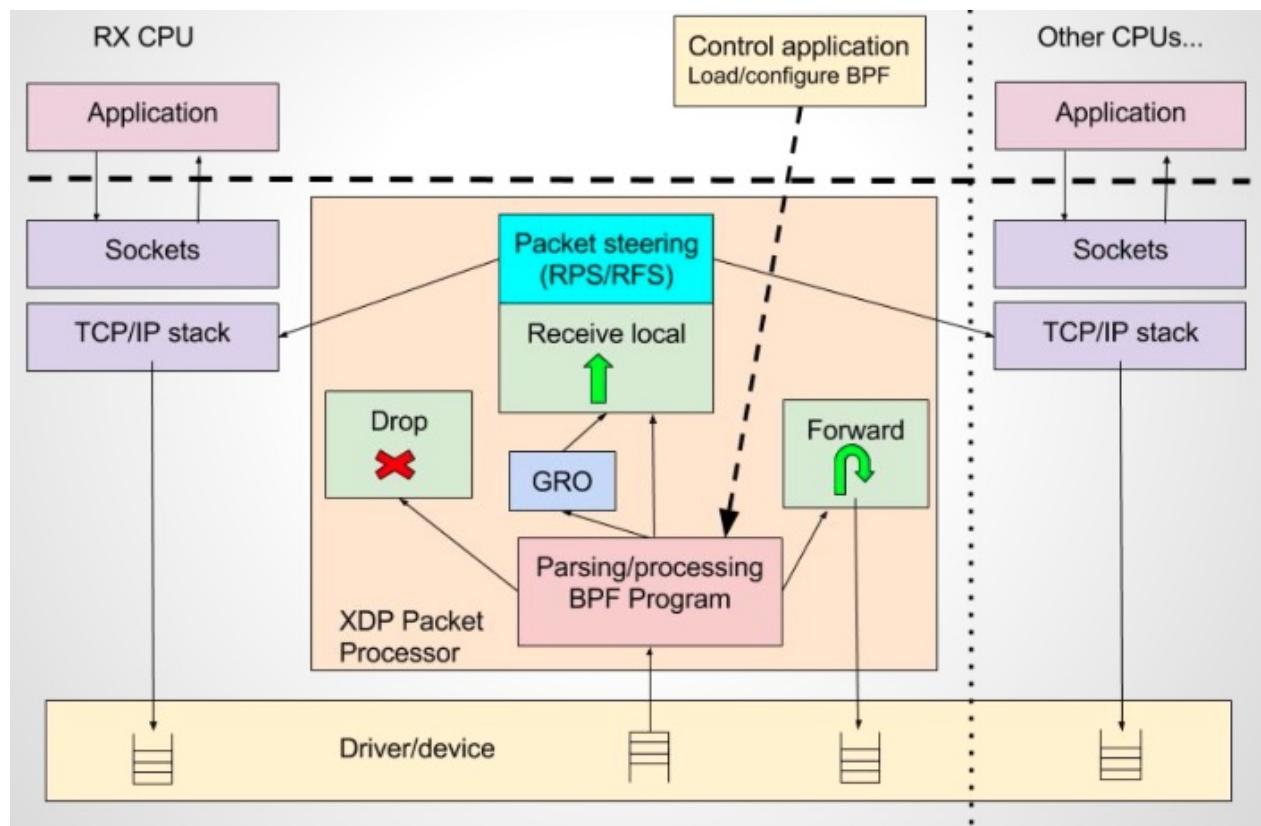
# XDP架构

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)` ing 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，可以用来自端口扫描、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="172.16.255.11", 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: xsprintf("%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		

net.core.netdev_max_backlog	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。	1000	10000
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

	息。		
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	128	

	载，可以尝试增加这个数字。		
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

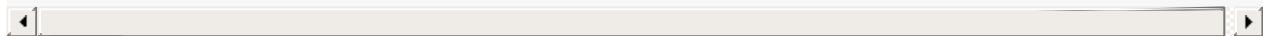
```
ovs-vsctl add-port br-ex port tag=10 -- set Interface port type=internal # default is access
ifconfig port 192.168.100.1
```

### 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=@tapa6cd1168  
-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=10000000000 \
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 sam
pling=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=00:88:77:66:5
5:44,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 , ipv  
6\_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 , ss1,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=flow 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

## 4. Open vSwitch

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=<MAC>` 和 `dl_dst=<MAC>` 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<idx> = 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
```

## 参考

- <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-fedora-kmod
```

## 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
```

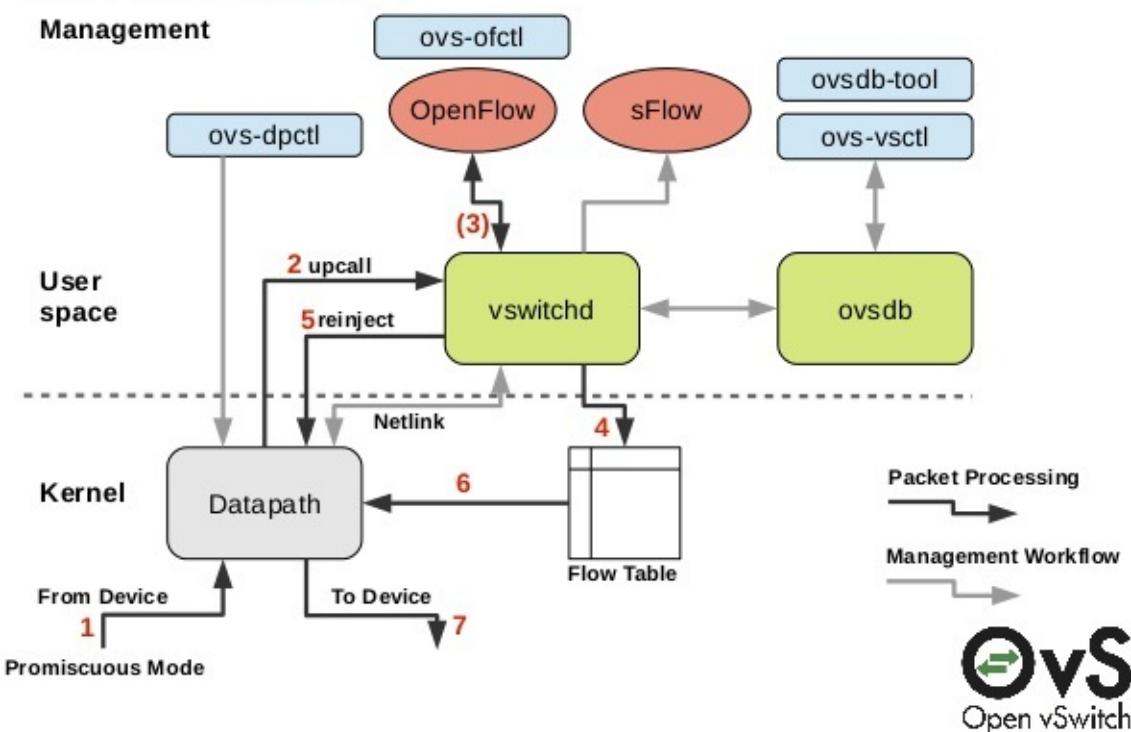
## 参考

- [Installing Open vSwitch](#)
- [Debian Packaging for Open vSwitch](#)
- [Open vSwitch with DPDK](#)
- [Open vSwitch on Linux, FreeBSD and NetBSD - Hot Upgrading](#)

# OVS原理

## OVS架构

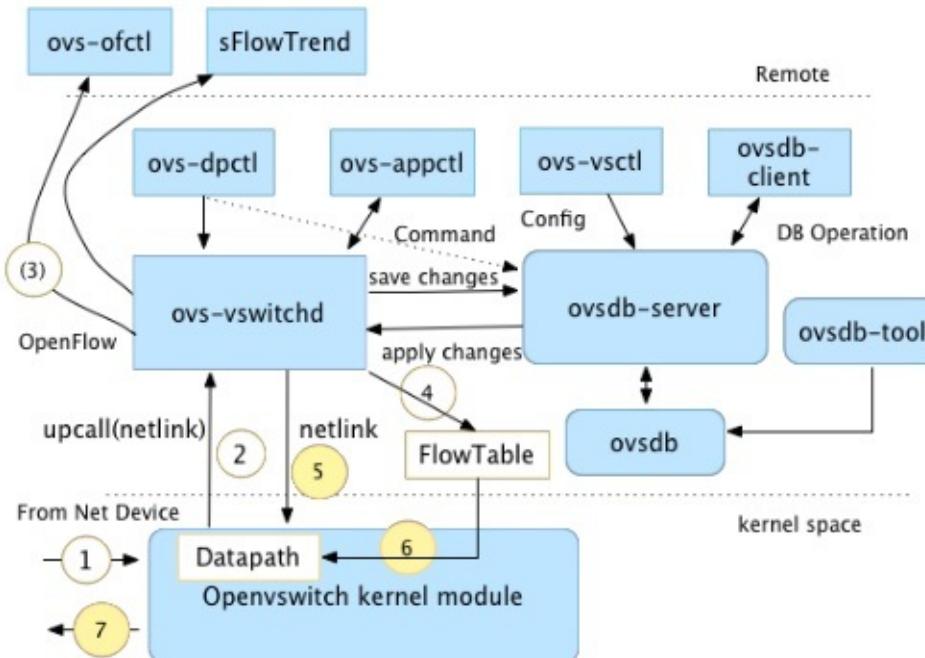
### Architecture



图片来源2015 FOSDEM - OVS Stateful Services

ovs 的架构如上图所示，主要由内核 datapath 和用户空间的 vswitchd 、 ovsdb 组成。

# OpenvSwitch Internals

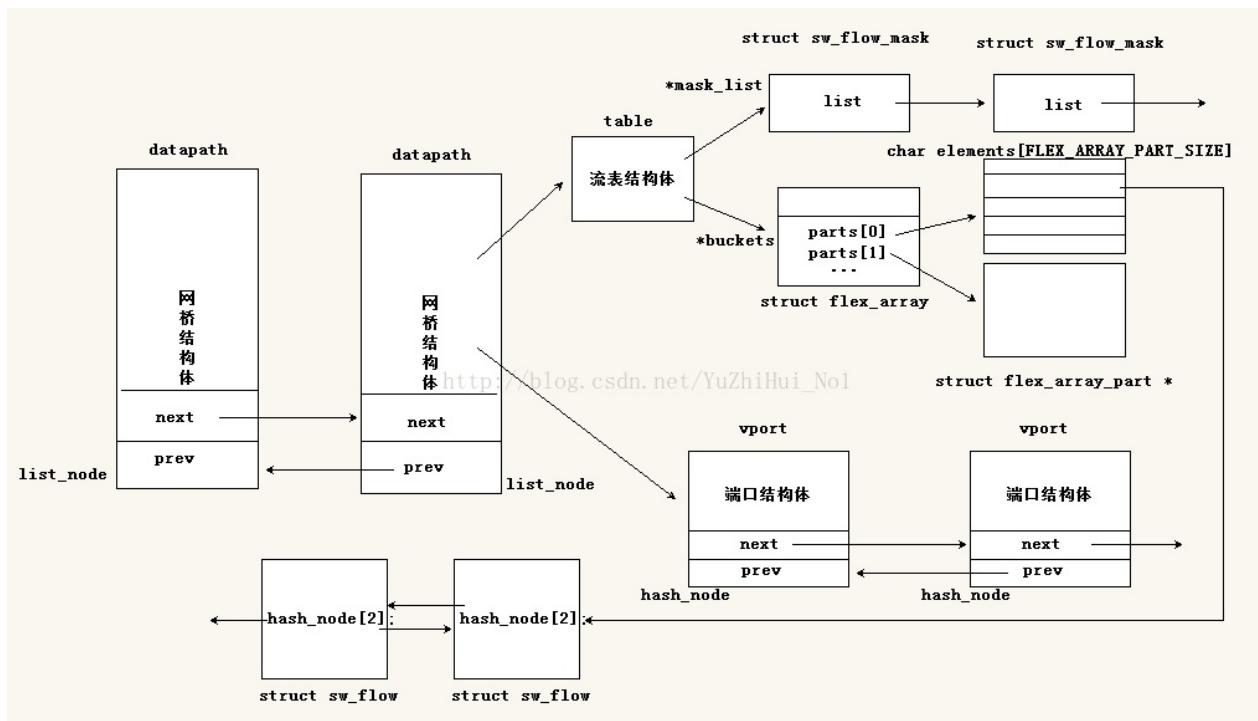


图片来源[OpenvSwitch Deep Dive](#)

## 主要模块职责

- **datapath** 是负责数据交换的内核模块，其从网口读取数据，并快速匹配**Flowtable**中的流表项，成功的直接转发，失败的上交 **vswitchd** 处理。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。
- **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

## 主要数据结构



(图片来自[csdn](http://blog.csdn.net/YuZhiHui_Noi))

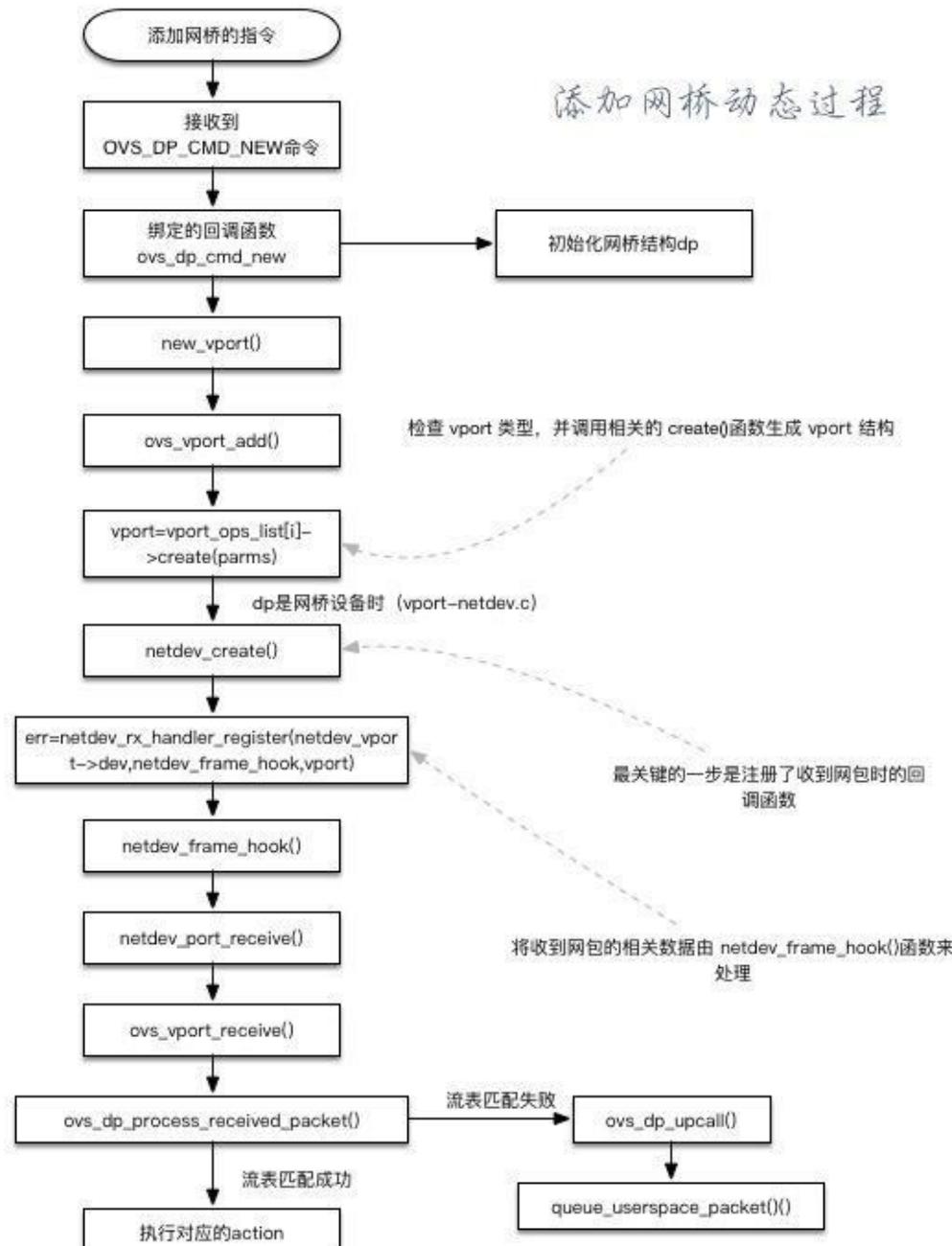
## 主要流程

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。

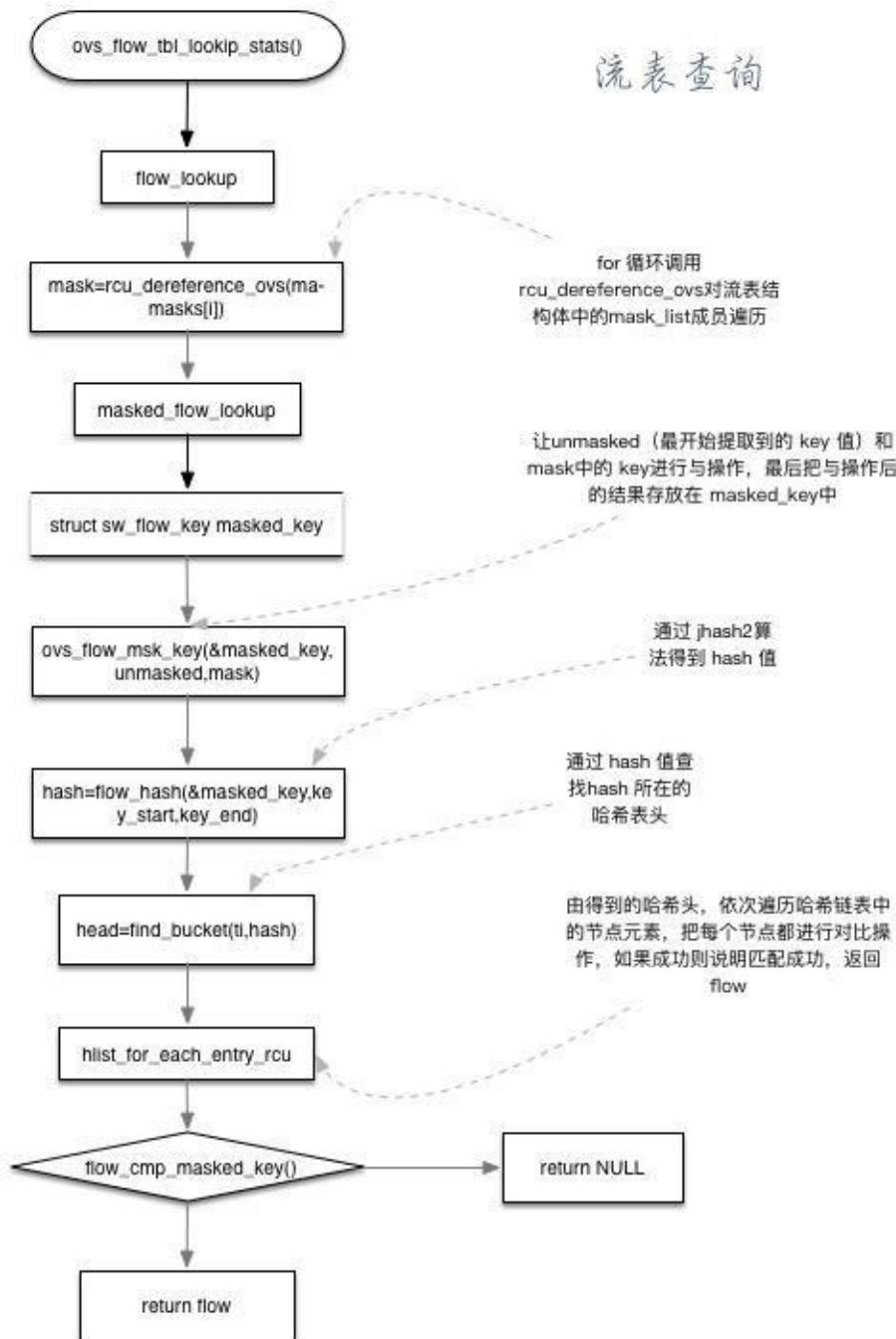


(图片来自简书)

## 流表匹配

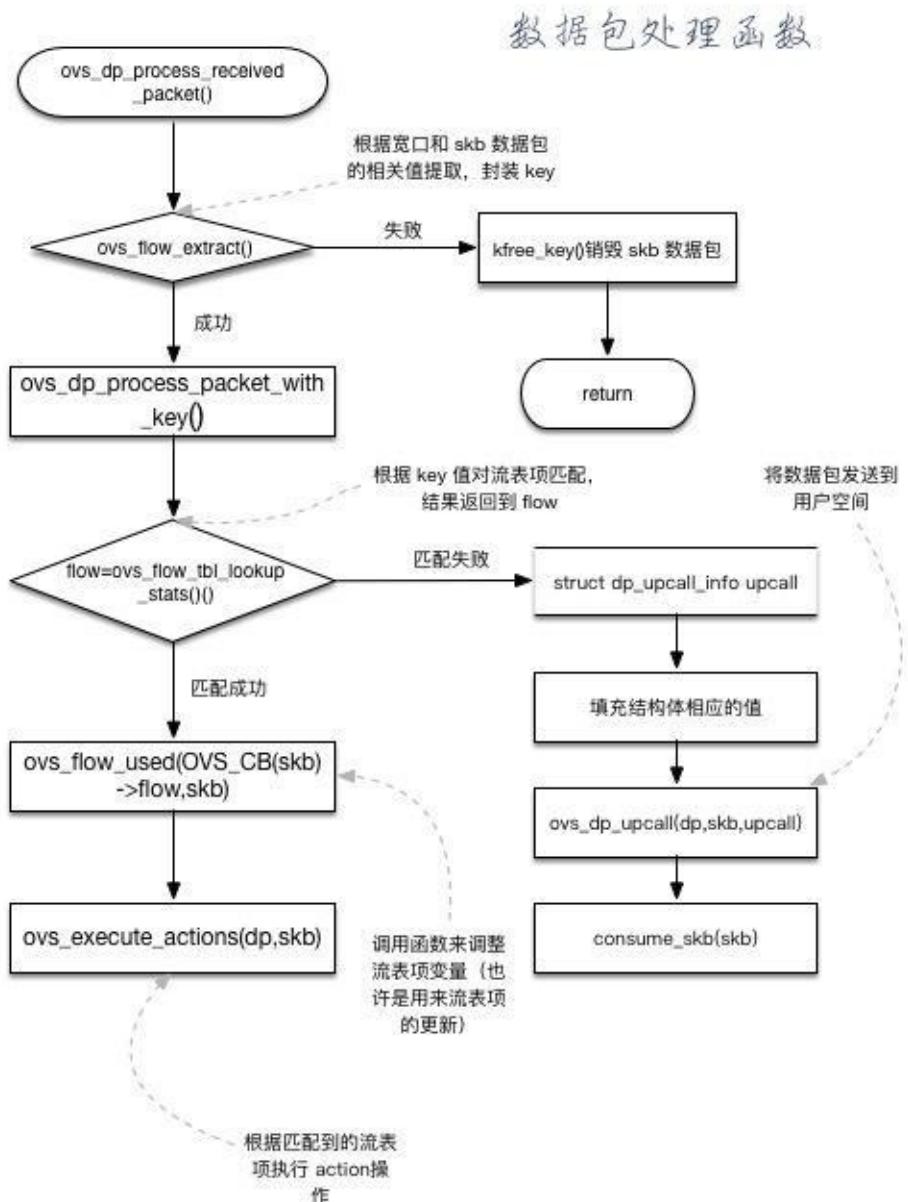
1. flow\_lookup() 查找对应的流表项
2. for 循环调用 rcu\_dereference\_ovs 对流表结构体中的 mask\_list 成员遍历，找到对应的成员
3. flow=masked\_flow\_lookup() 遍历进行下一级 hmap 查找，找到为止

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



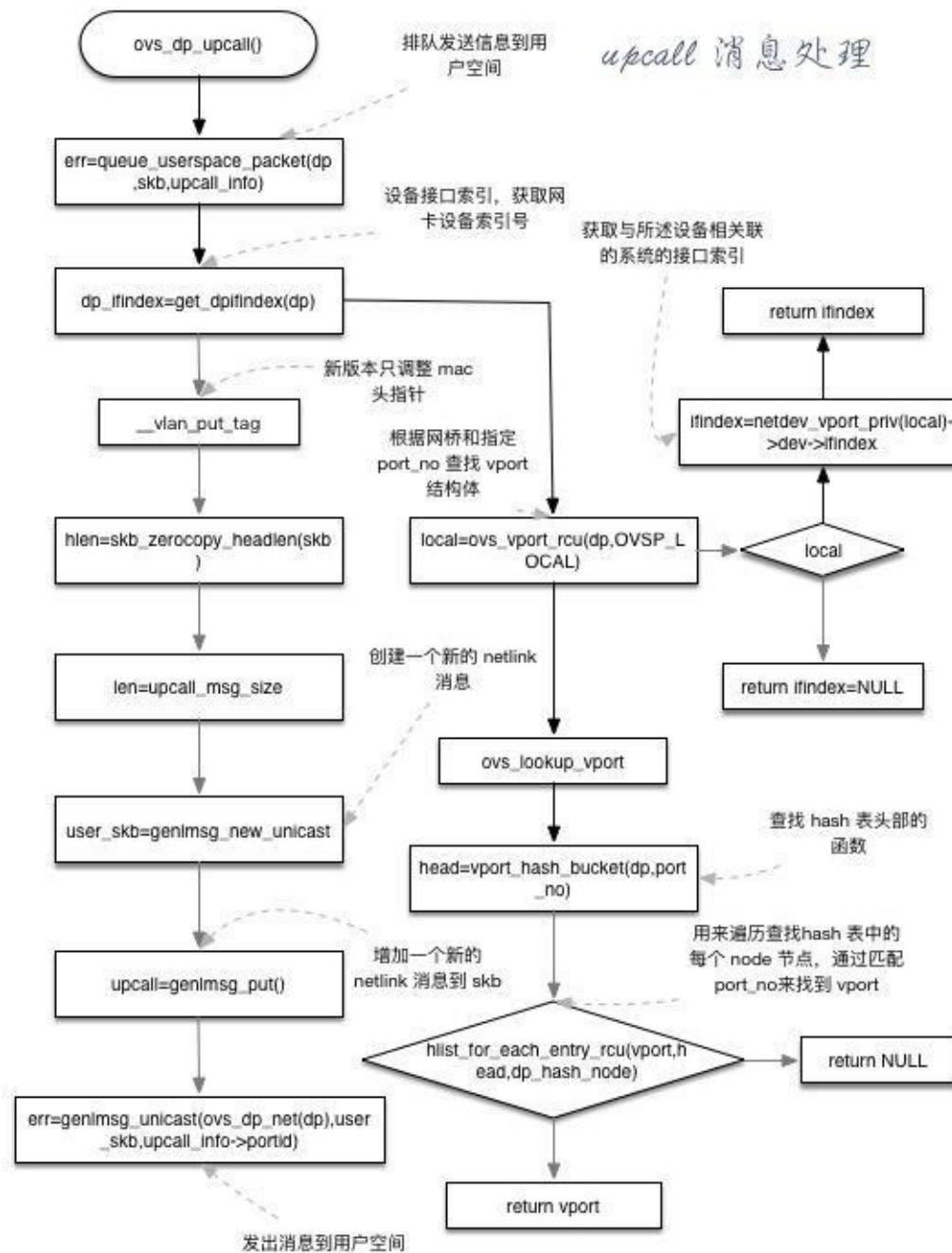
## 收包处理

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

- OpenvSwitch Deep Dive
- 2015 FOSDEM - OVS Stateful Services
- OVS 源码分析整理
- openvswitch源码分析
- OVS Deep Dive



# Open Virtual Network

## 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

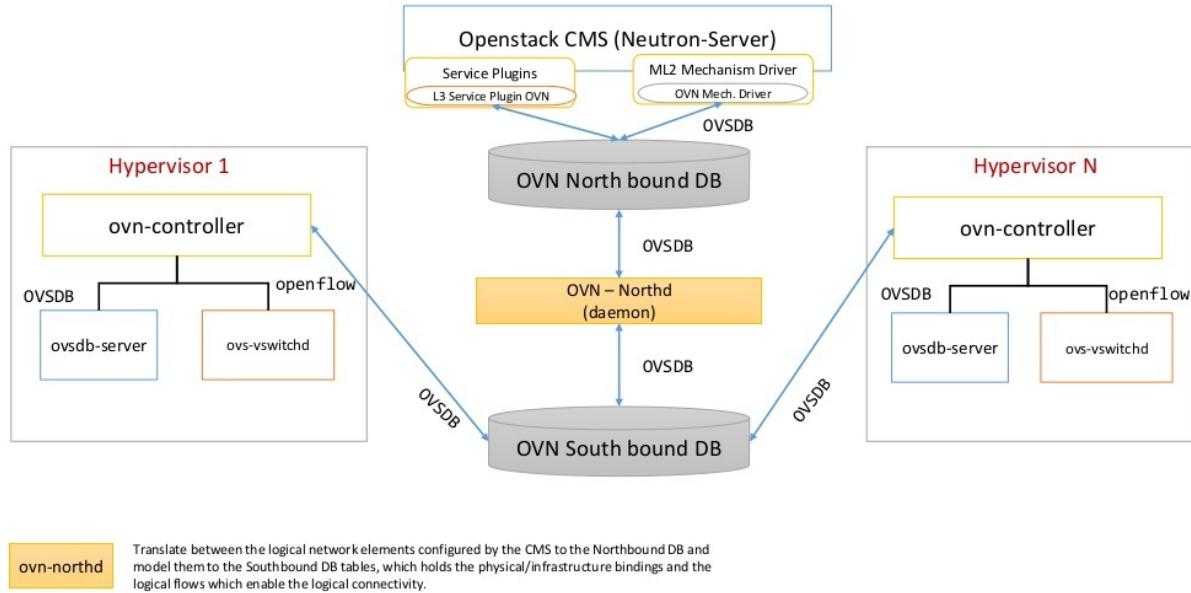
1. L2/L3 虚拟网络以及逻辑交换机( logical switch )
2. L2/L3/L4 ACL
3. IPv4/IPv6 分布式 L3 路由
4. ARP and IPv6 Neighbor Discovery suppression for known IP-MAC bindings
5. Native support for NAT and load balancing using OVS connection tracking
6. Native fully distributed support for DHCP
7. Works with any OVS datapath (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 ovn-northd，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 ovn-southd（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

## OVN – Architecture



## 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
```

## 参考

- [Open Virtual Network - architecture](#)

- OVN - Basics and deep dive
- Open Virtual Network (OVN) - sflow blog

# 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

- [OVN Logical Flows and ovn-trace](#)
- [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

- [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

- [OVN L2 Deep Dive](#)

# OVN L3

OVN L3 的功能包括

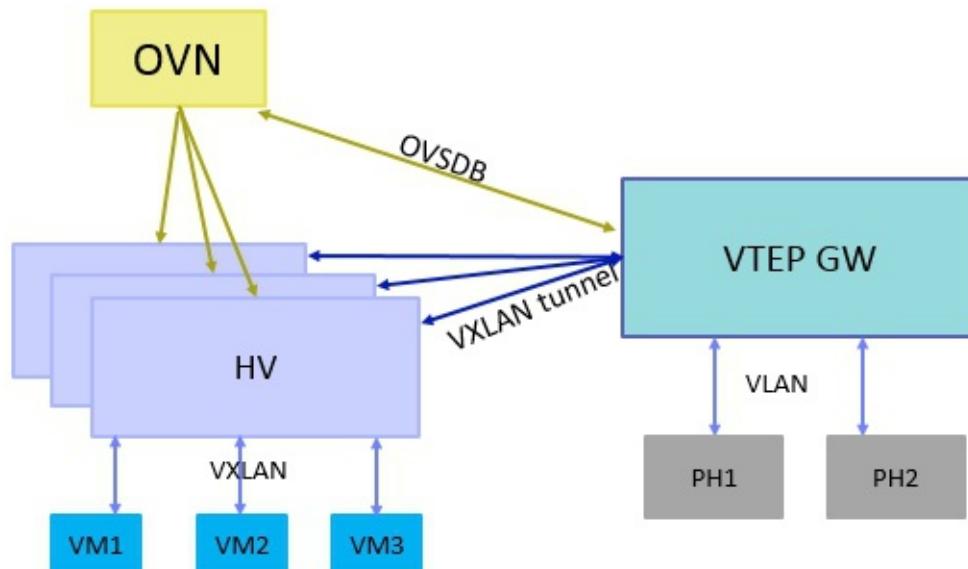
- IPv4/IPv6 分布式 L3 路由
- ARP and IPv6 Neighbor Discovery suppression for known IP-MAC bindings
- L3 ACL
- Native support for NAT and load balancing using OVS connection tracking
- Native fully distributed support for DHCP
- Supports L3 gateways from logical to physical networks

```
# 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

- [OVN L3 Deep Dive](#)
- [OpenStack SDN With OVN \(Part 2\) - Network Engineering Analysis](#)

# OVN VTEP



图片来源[如何借助 OVN 来提高 OVS 在云计算环境中的性能](#)

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表项和内容，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 Sounthbound 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](#)。

## 参考

- [OVN Logical Flows and ovn-trace](#)
- [OpenStack Security Groups using OVN ACLs](#)
- [OVN L2 Deep Dive](#)
- [如何借助 OVN 来提高 OVS 在云计算环境中的性能](#)
- [OVN Tutorial - feiskyer/ops](#)

# 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 data-
    bases in server, which must be an active connection method in
    one of the forms documented in ovsdb-client(1). Every transac-
    tion 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/con-
    nect-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。

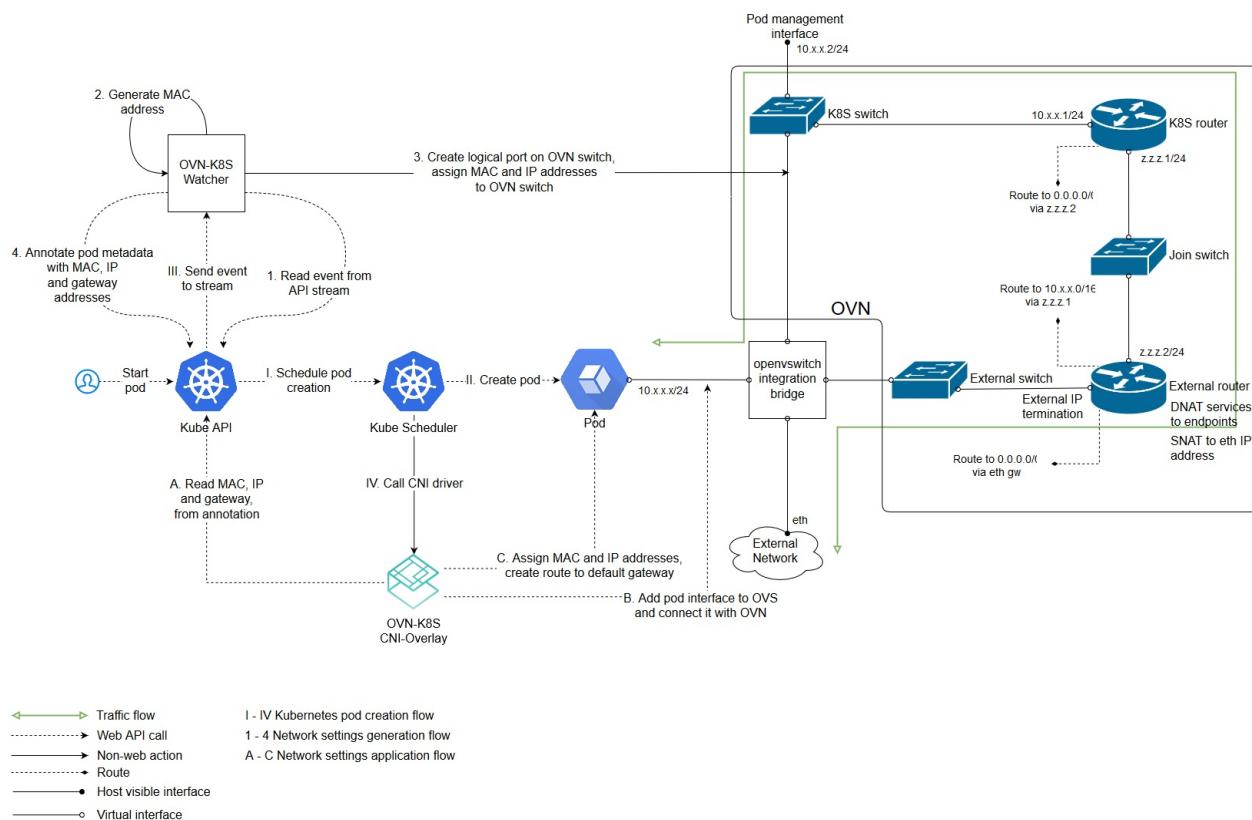
- <https://github.com/blp/ovs-reviews/tree/raft3>
- <http://docs.openvswitch.org/en/latest/topics/high-availability/>
- <http://galsagie.github.io/2015/08/03/df-distributed-db/>

# 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 [How to Use Open Virtual Networking With Kubernetes](#)

## 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

- [How to Use Open Virtual Networking With 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

- [Open Virtual Networking With Docker](#)
- [在Docker中使用Open vSwitch创建跨主机的容器网络](#)

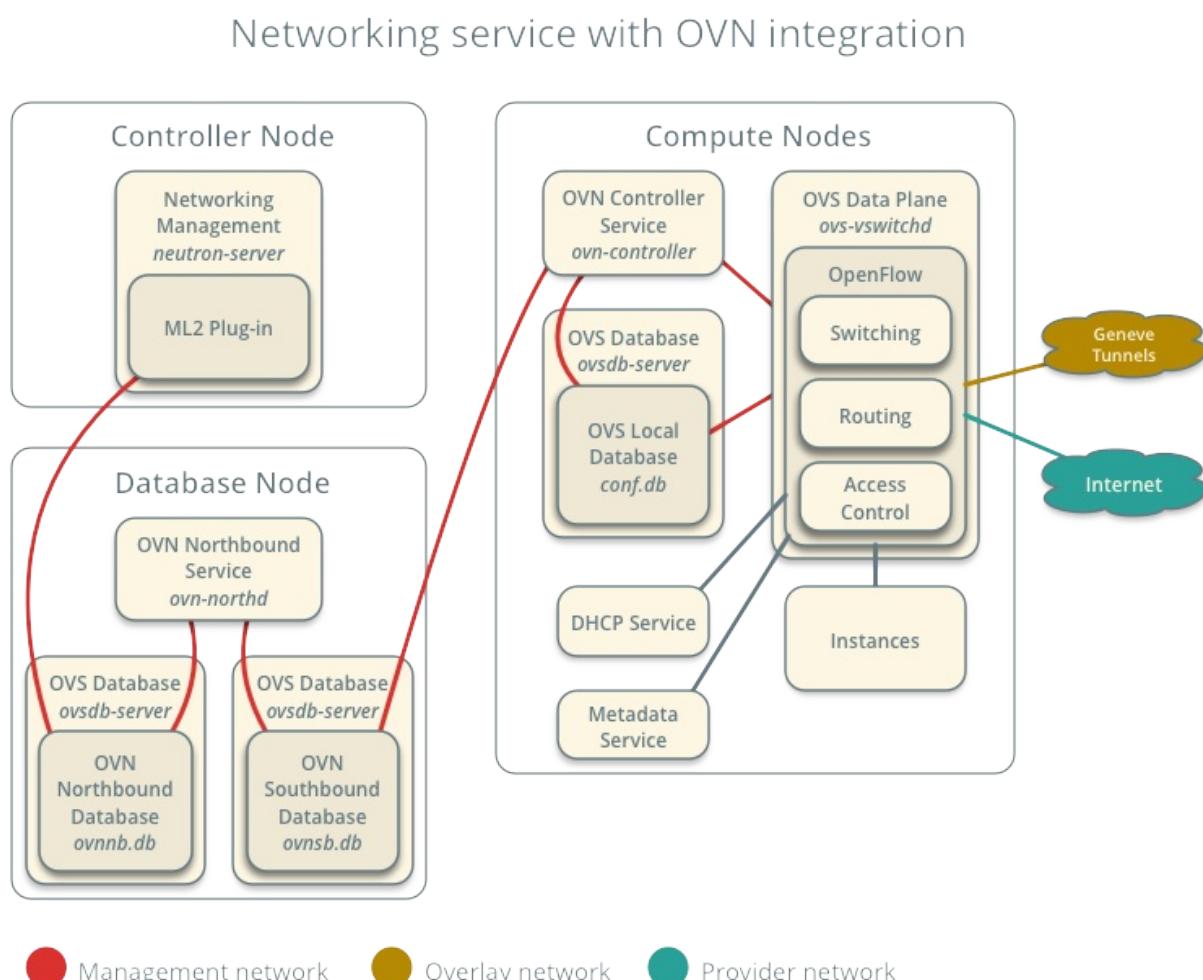
# OVN OpenStack

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。

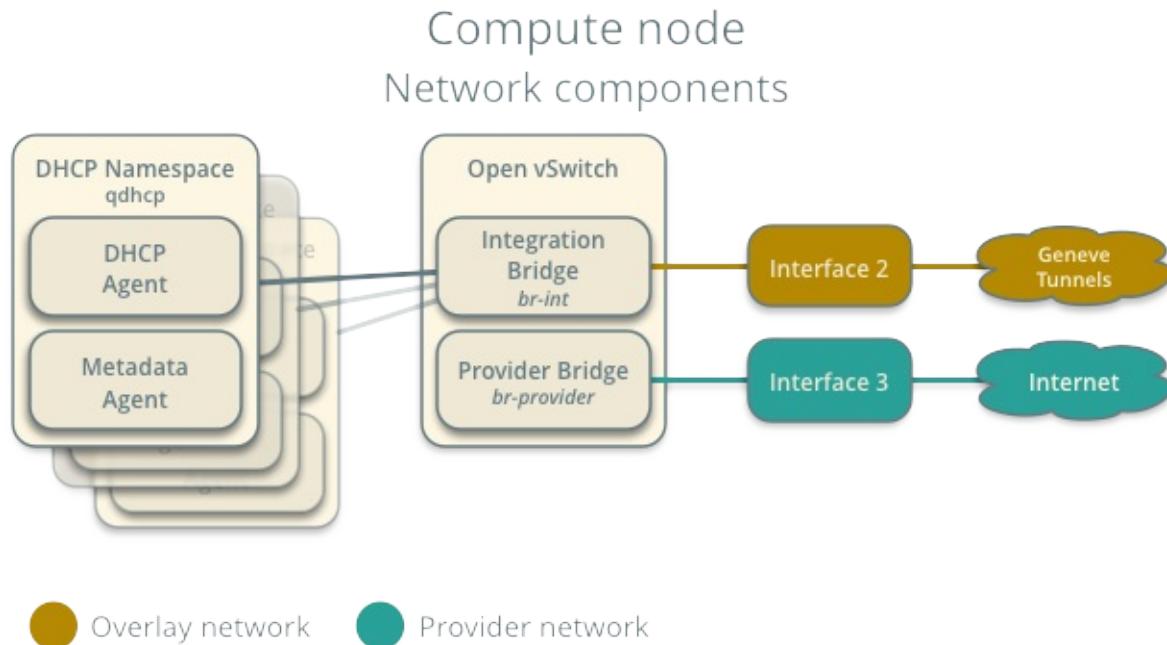
OVN 里面报文的处理都是通过 OVS OpenFlow 流表来实现的，而在 Neutron 里面二层报文处理是通过 OVS OpenFlow 流表来实现，三层报文处理是通过 Linux TCP/IP 协议栈来实现。

## 架构

网络节点



而计算节点包括以下服务



图片来源 [Reference architecture - OpenStack](#)

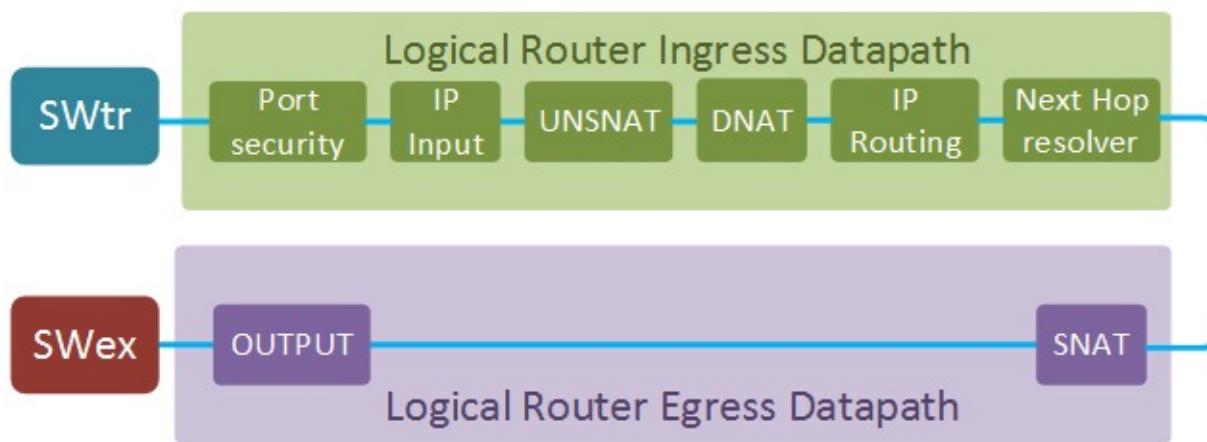
## 实现原理

### 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。



图片来源 [OpenStack SDN With OVN \(Part 2\) - Network Engineering Analysis](#)

比如SNAT和DNAT的流表为

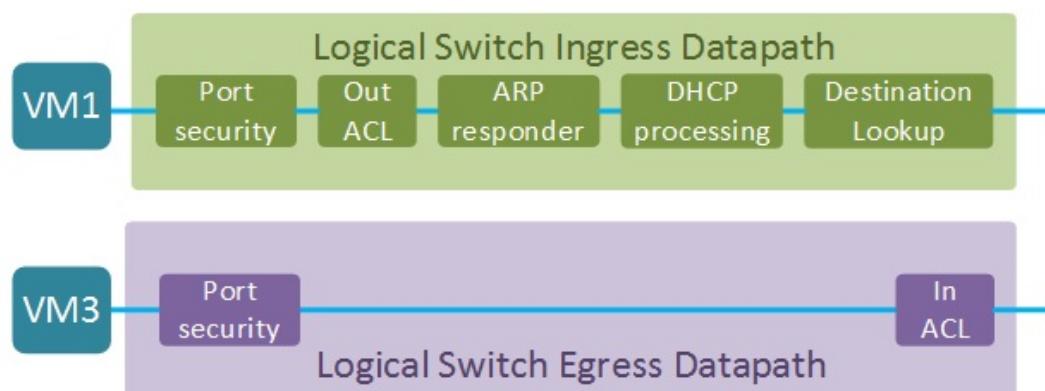
```
# SNAT
table=0 (lr_out_snat), priority=25, match=(ip && ip4.src == 10.0.0.0/24), action=(ct_snat(169.254.0.54);)

# 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

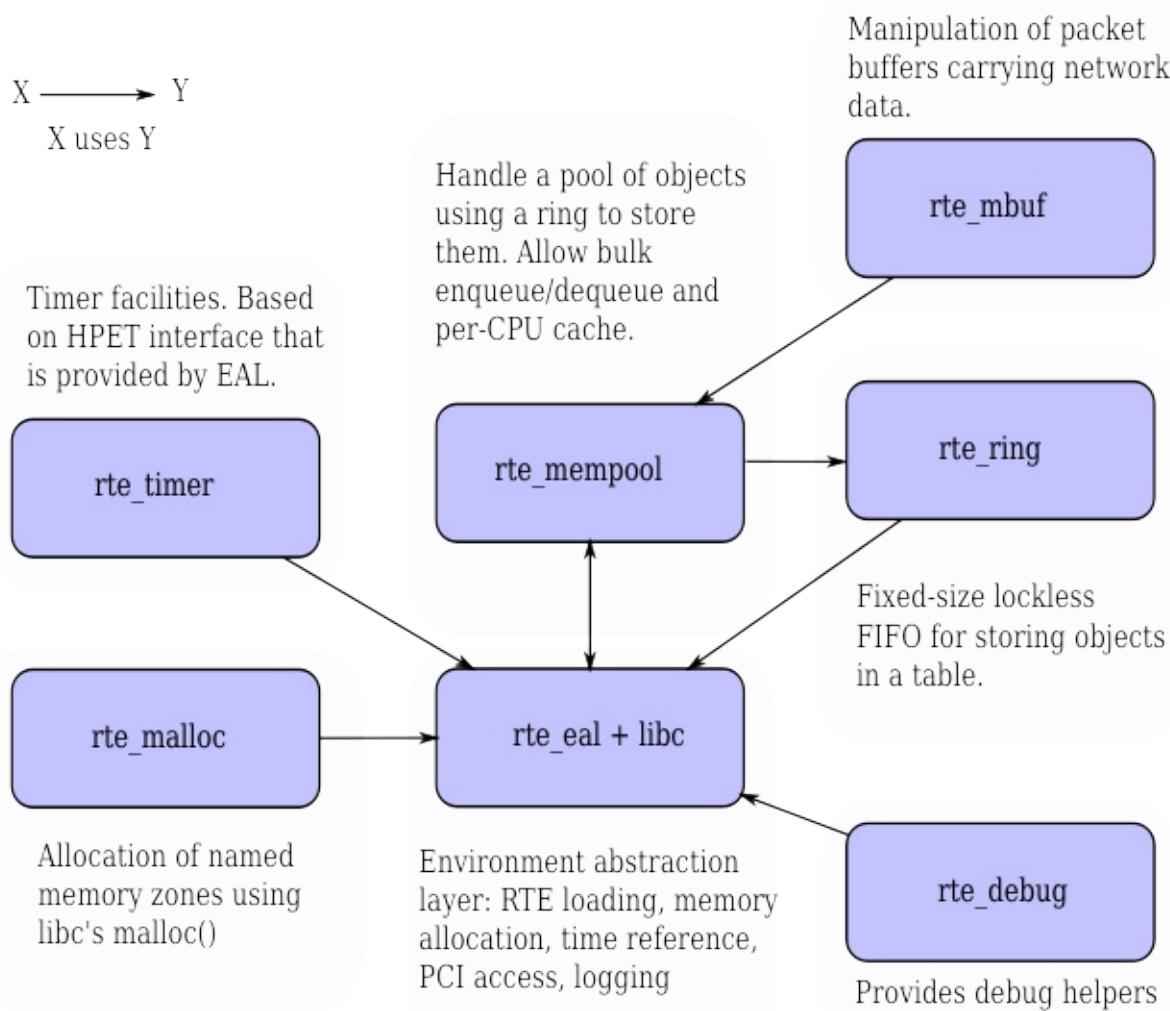
- networking-ovn reference architecture
- 如何借助 OVN 来提高 OVS 在云计算环境中的性能
- OpenStack SDN With OVN

# 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



图片来源 [Architecture Overview - dpdk.org](https://dpdk.org/doc/ArchitectureOverview.html)

- 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：

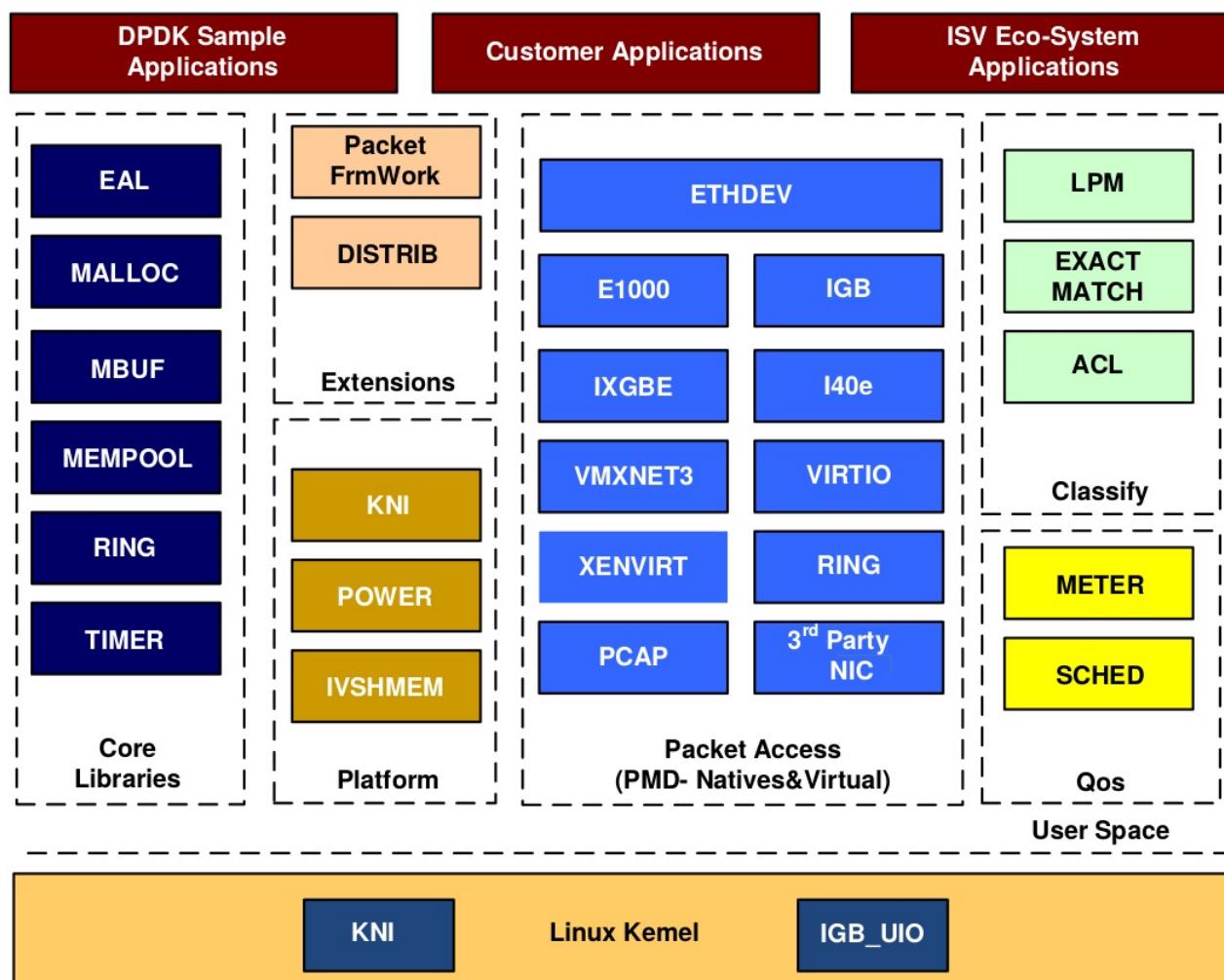
- PMD：DPDK 针对 Intel 网卡实现了基于轮询方式的 PMD ( Poll Mode Drivers ) 驱动，该驱动由 API 、用户空间运行的驱动程序构成，该驱动使用无中断方式直接操作网卡的接收和发送队列（除了链路状态通知仍必须采用中断方式以外）。目前 PMD 驱动支持 Intel 的大部分 1G 、 10G 和 40G 的网卡。PMD 驱动从网卡上接收到数据包后，会直接通过 DMA 方式传输到预分配的内存中，同时更新无锁环形队列中的数据包指针，不断轮询的应用程序很快就能感知收到数据包，并在预分配的内存地址上直接处理数据包，这个过程非常简洁。如果要是让 Linux 来处理收包过程，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。
- 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：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，尽量访问本地内存。
- 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



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。

## 架构

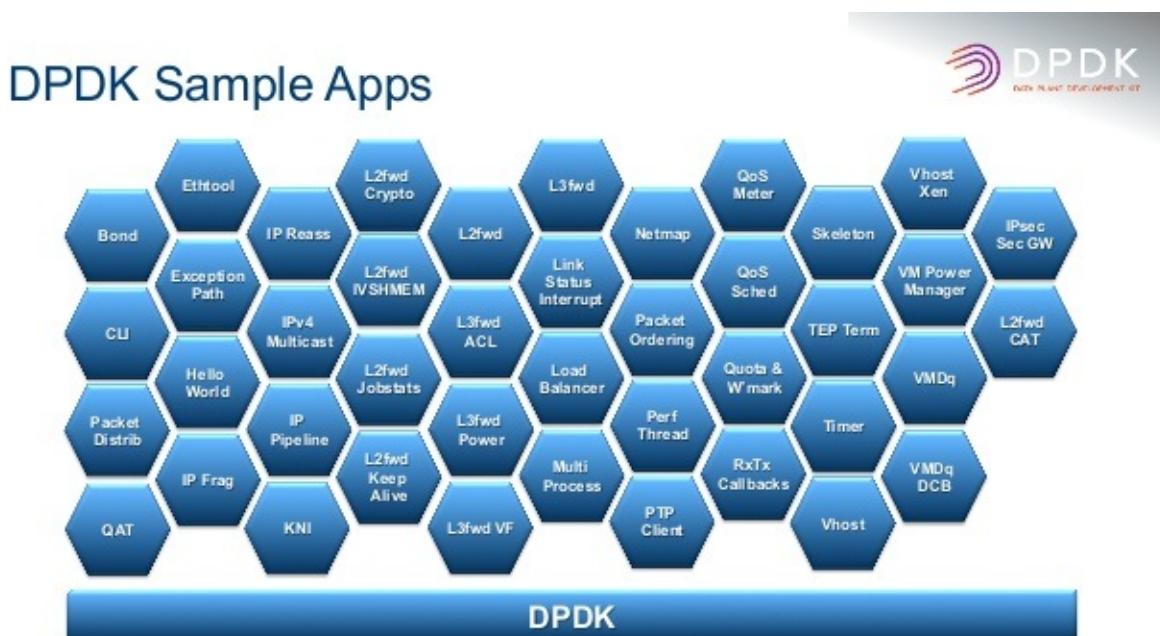


图片来源DPDK初探

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\uiuo.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



图片来源[DPDK: Multi Architecture High Performance Packet Processing](#)

- 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

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》
- dpdk.org
- DPDK documentation - dpdk.org
- Data Plane Development Kit (DPDK)
- FD.io - The Fast Data Project The Universal Dataplane
- fd.io Developer Wiki
- lagopus/lagopus
- Data Plane Performance Demonstrators (DPPD)
- DPDK Summit - 08 Sept 2014 - Intel - Networking Workloads on Intel Architecture
- Getting Started Guide for Linux
- DPDK: Multi Architecture High Performance Packet Processing

# 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 cache line 读进来，这时一个核修改变量，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。False 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会导致 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 `Xeon` 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%。

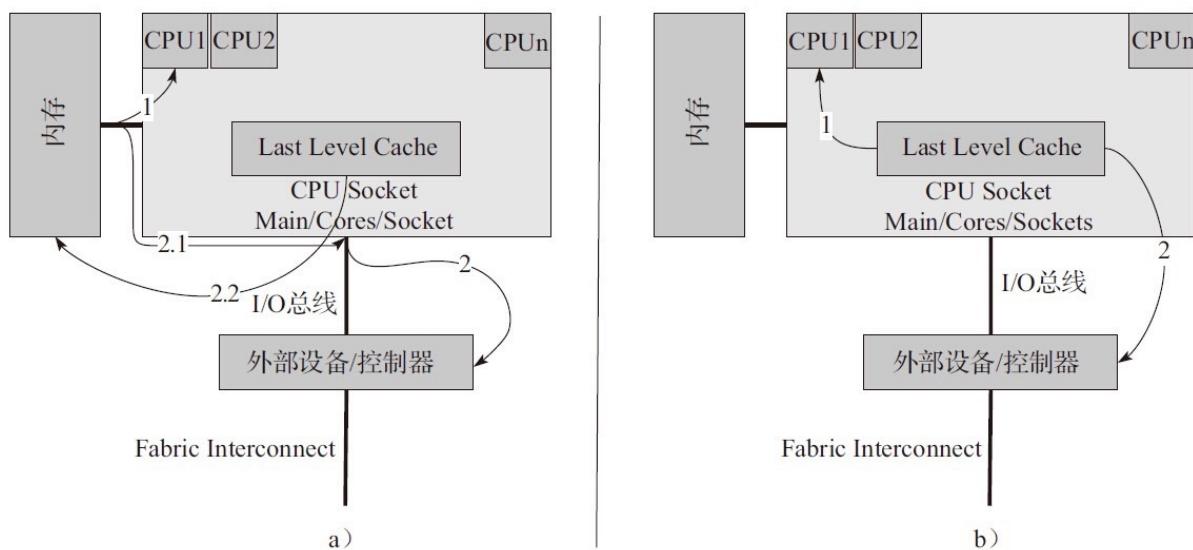
Linux系统启动后预留大页的方法

- 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)

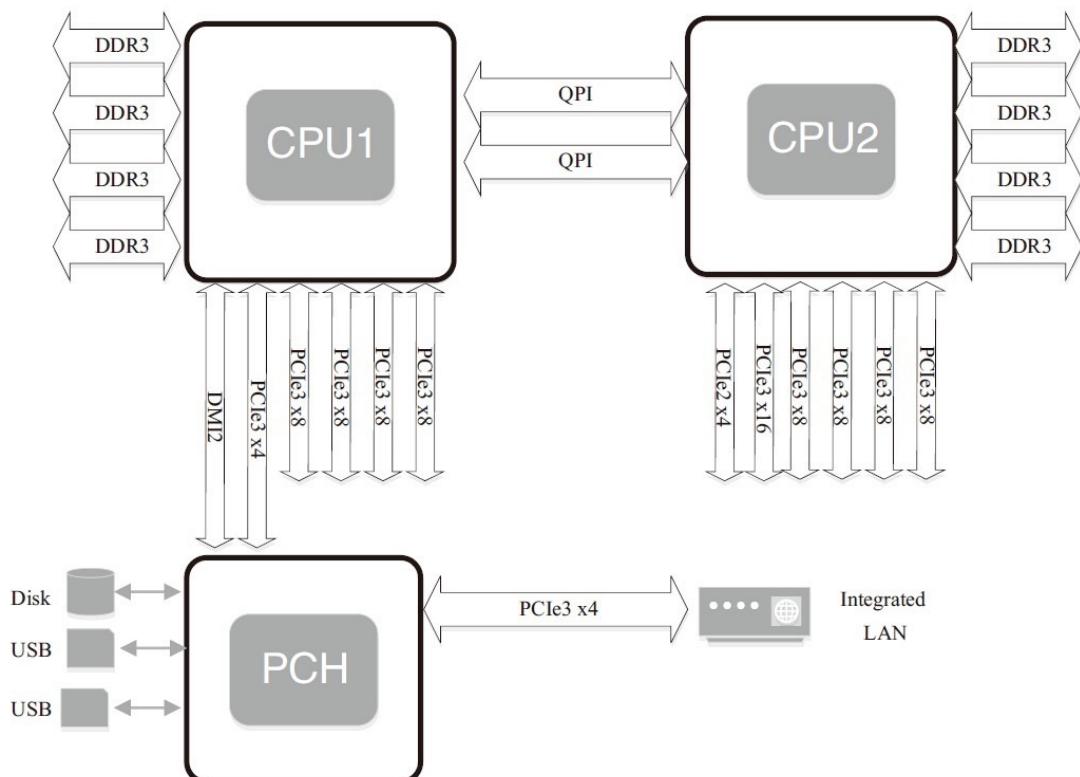
DDIO使得外部网卡和CPU通过LLC cache直接交换数据，绕过了内存，增加了CPU处理报文的速度。

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CPUcore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

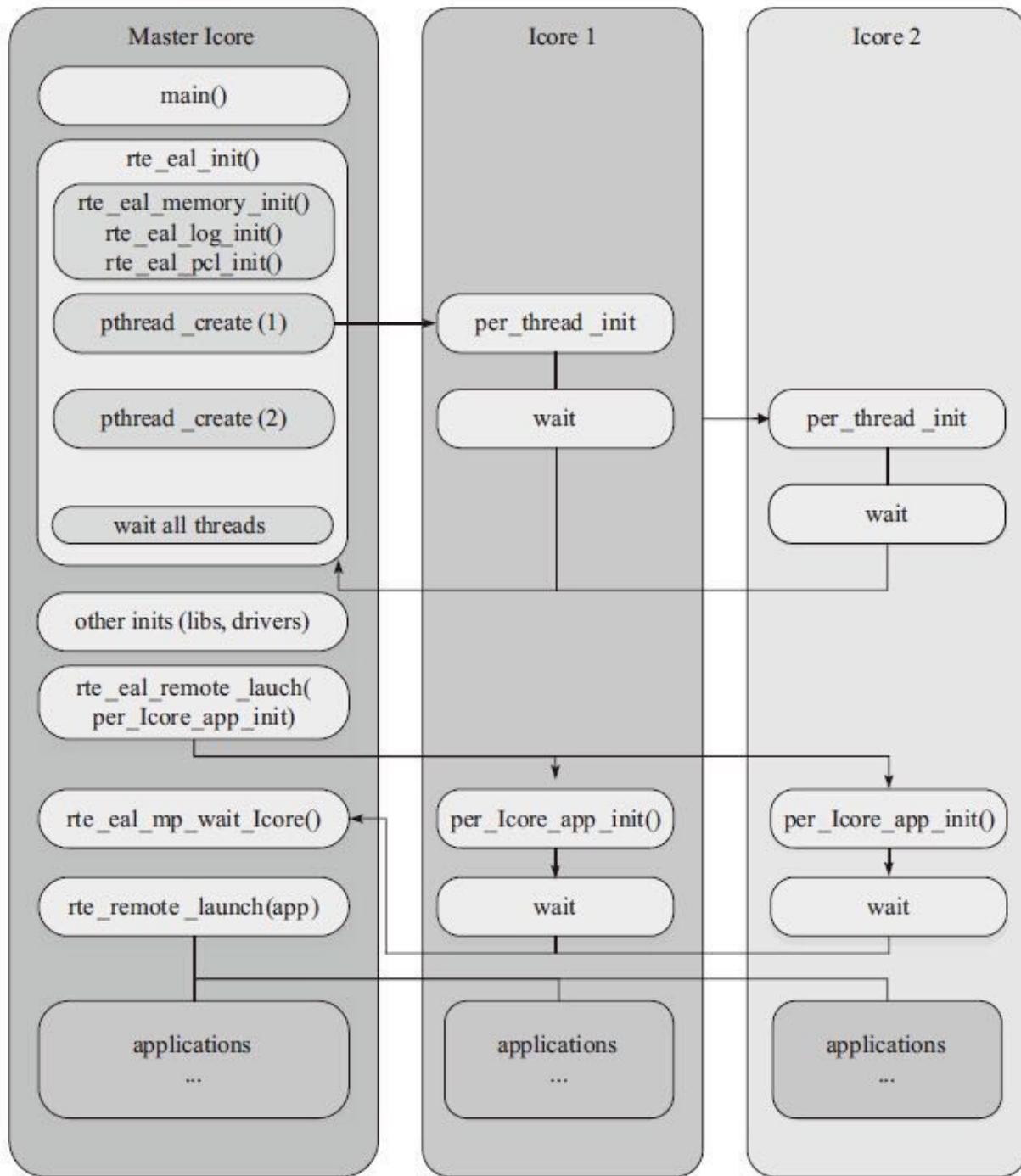
- sched\_set\_affinity()、sched\_get\_affinity() 内核函数
- taskset 命令
- isolcpus 内核启动参数：CPU绑定之后依然是有可能发生线程切换，可以借助 isolcpus=2,3 将cpu从内核调度系统中剥离。

## 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

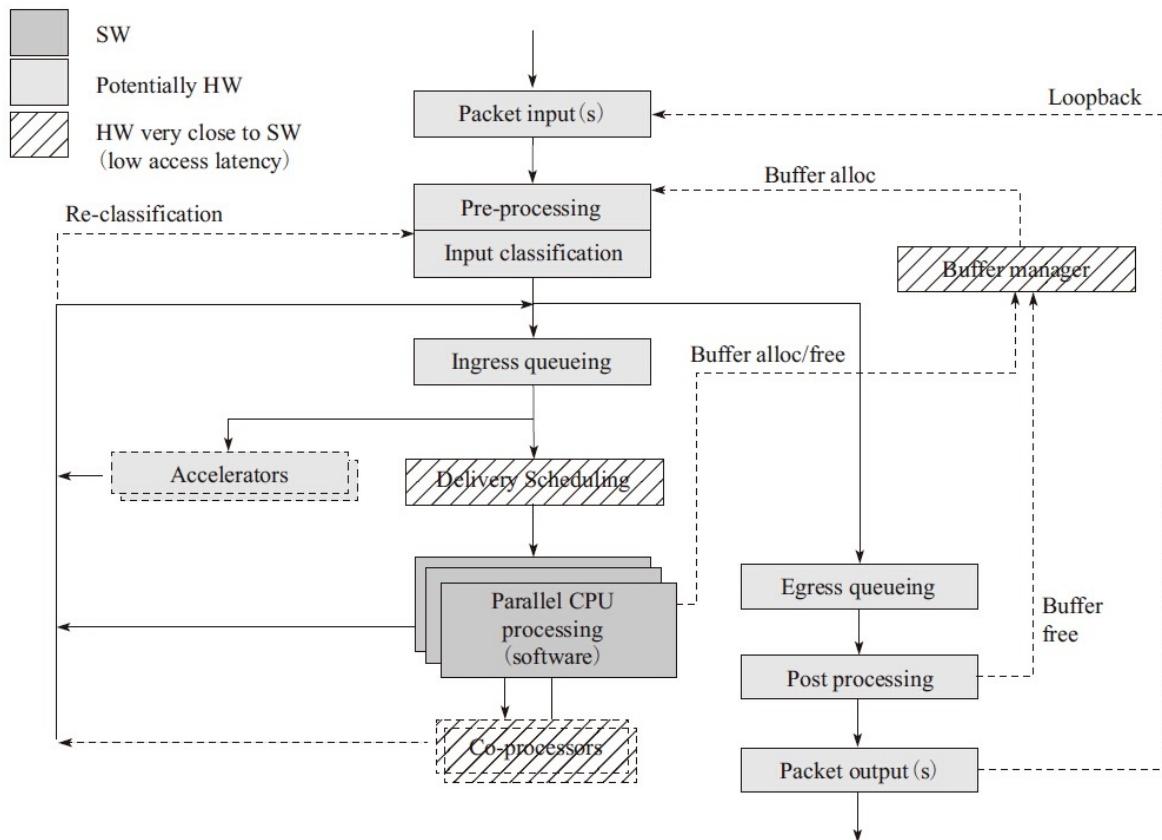
```
subscription-manager repos --enable rhel-7-server-extras-rpms  
yum install dpdk dpdk-tools
```

## Ubuntu 15+

```
apt-get install dpdk
```

# DPDK报文转发

基本的网络包处理主要包含：

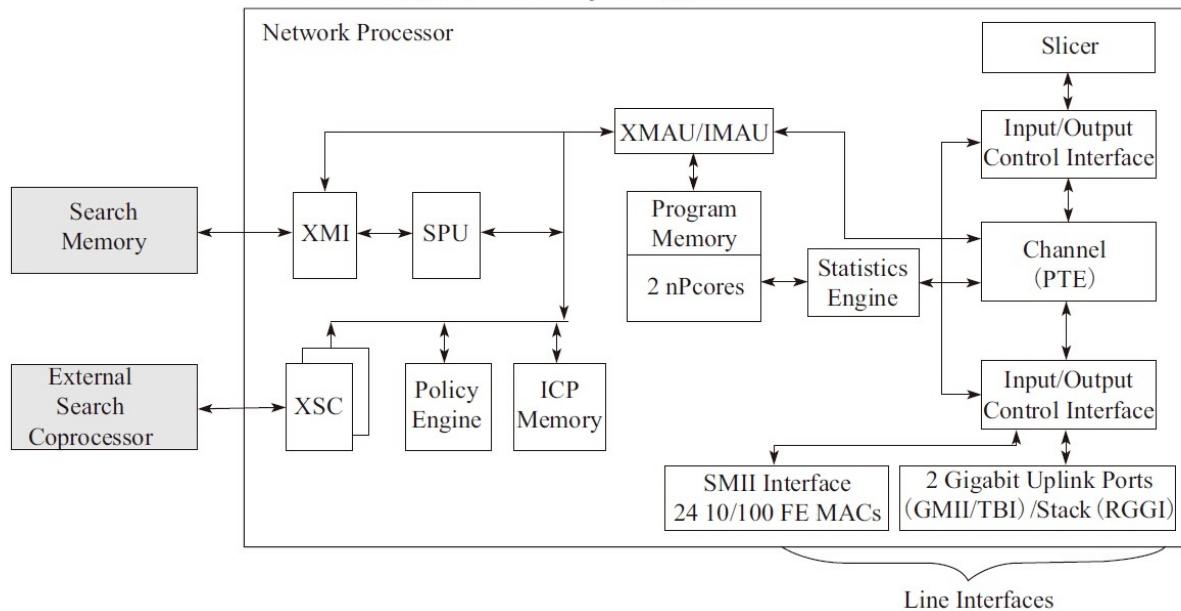


- `Packet input` : 报文输入。
- `Pre-processing` : 对报文进行比较粗粒度的处理。
- `Input classification` : 对报文进行较细粒度的分流。
- `Ingress queueing` : 提供基于描述符的队列 FIFO。
- `Delivery/Scheduling` : 根据队列优先级和 CPU 状态进行调度。
- `Accelerator` : 提供加解密和压缩/解压缩等硬件功能。
- `Egress queueing` : 在出口上根据 QOS 等级进行调度。
- `Post processing` : 后期报文处理释放缓存。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

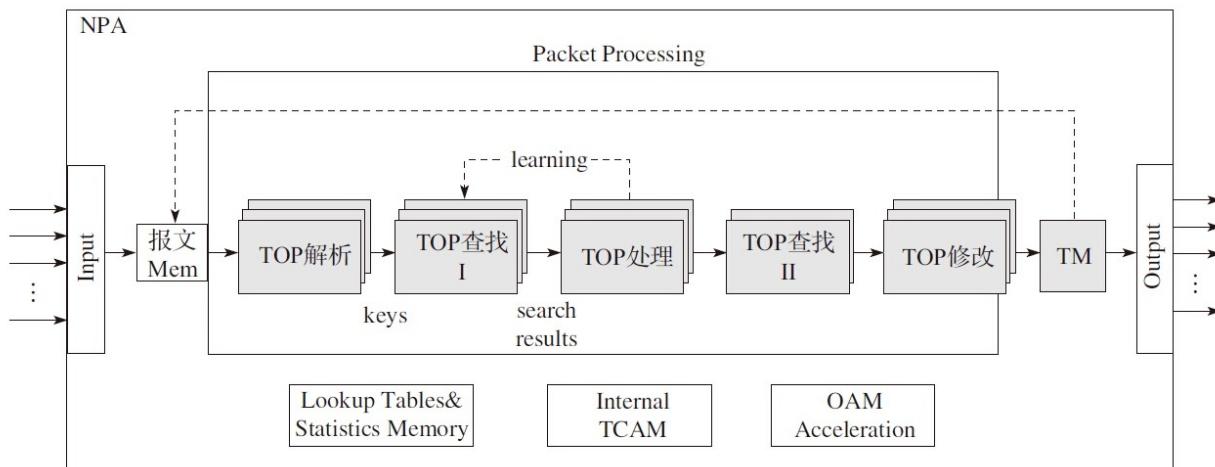
## 专用网络处理器转发模型

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。

AMCC 345x—传统的 run to completion 模型

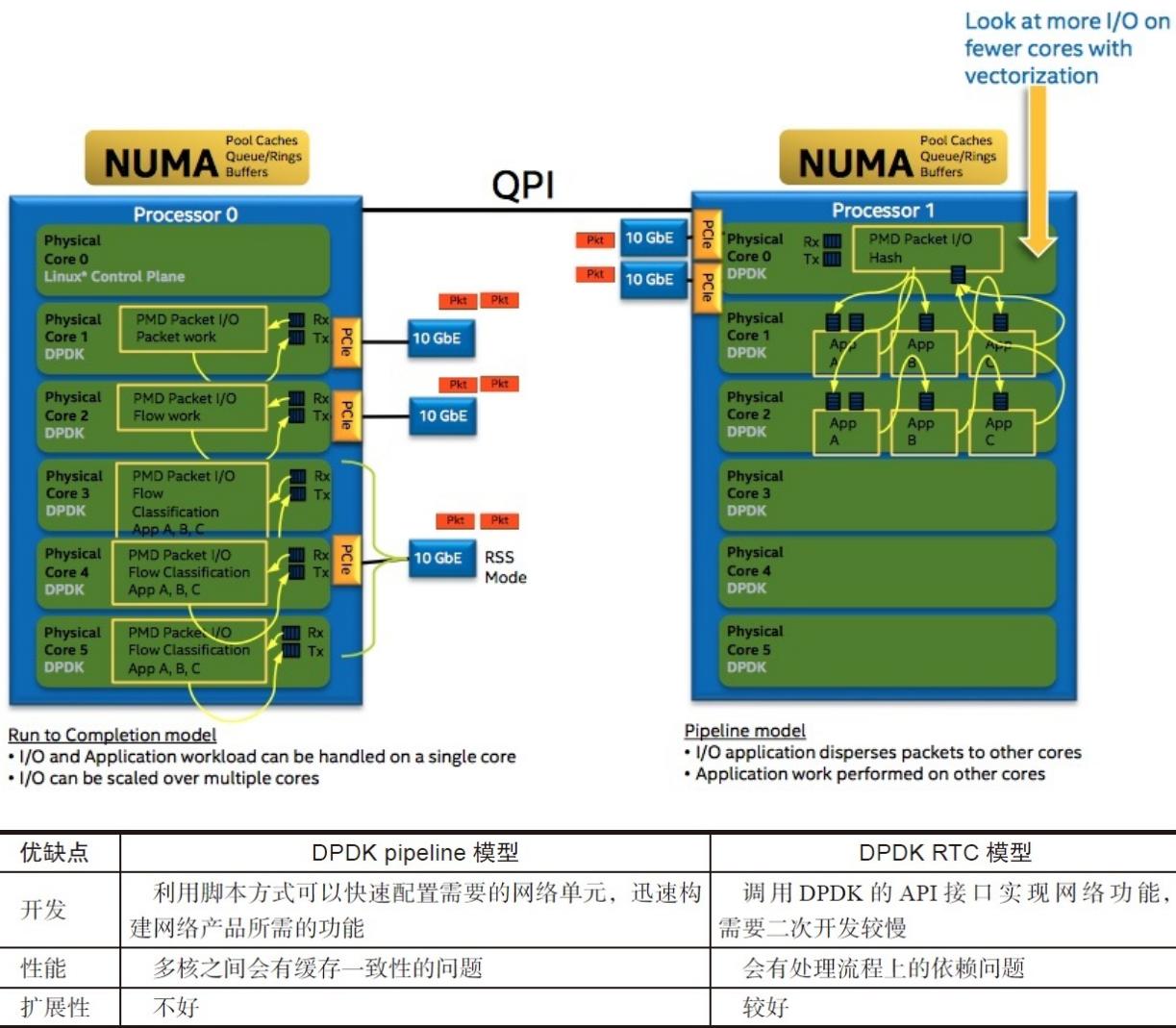


Ezchip-传统的pipeline 模型



## DPDK 转发模型

从 [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 [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。

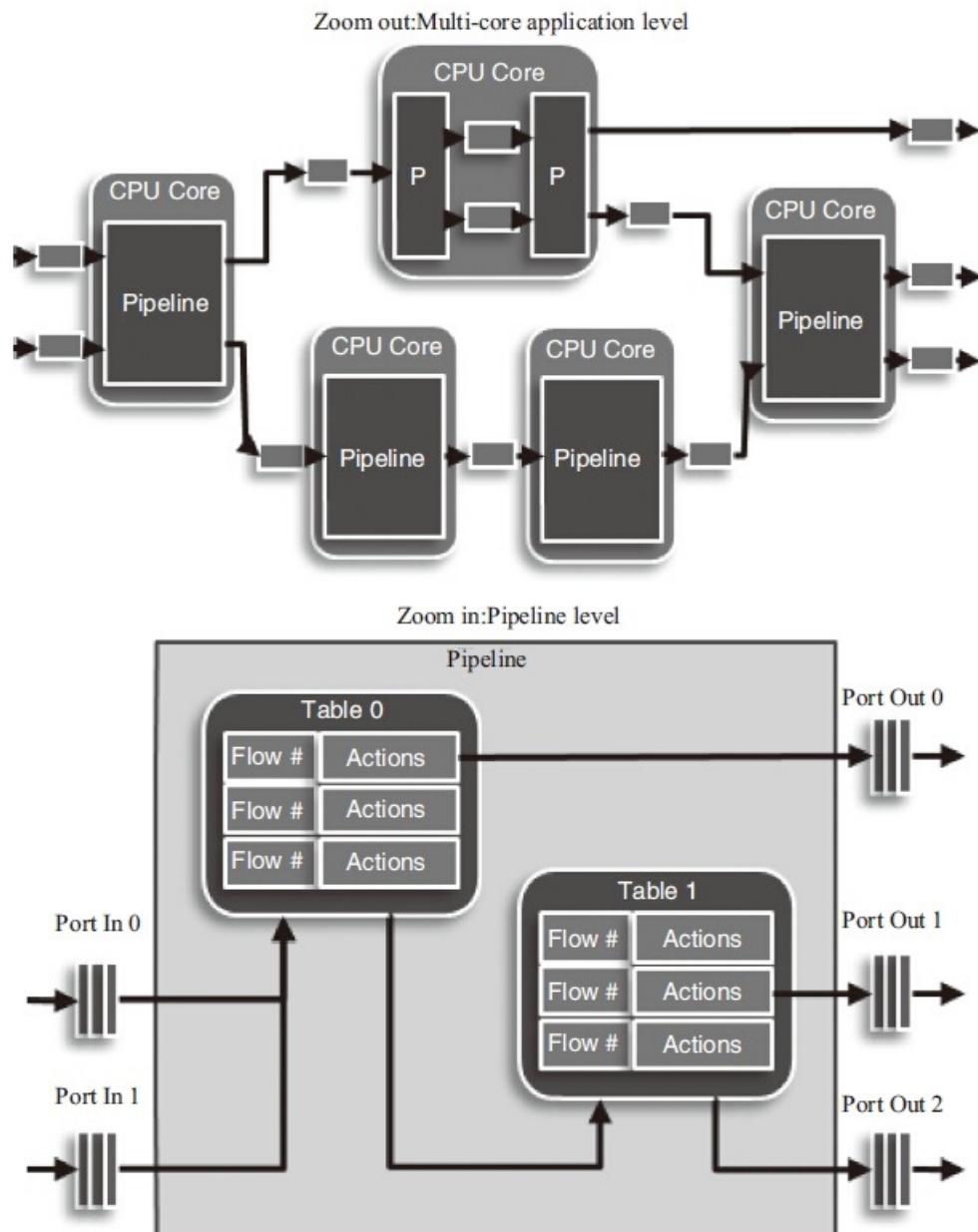


# DPDK run to completion 模型

在 DPDK 的轮询模式中主要通过一些 DPDK 中 eal 中的参数 -c、-l、-1 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。

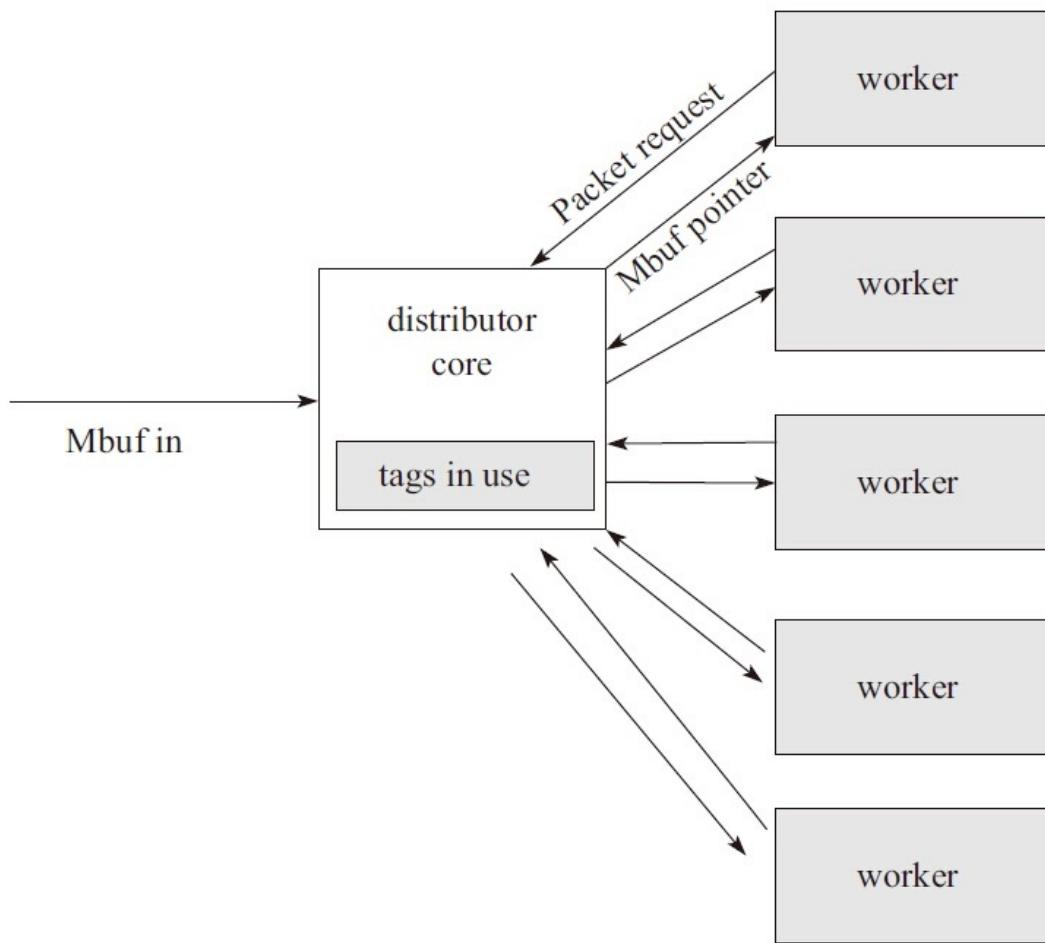
精确匹配主要需要解决两个问题：进行数据的签名（哈希），解决哈希的冲突问题，DPDK 中主要支持 CRC32 和 J hash。

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：

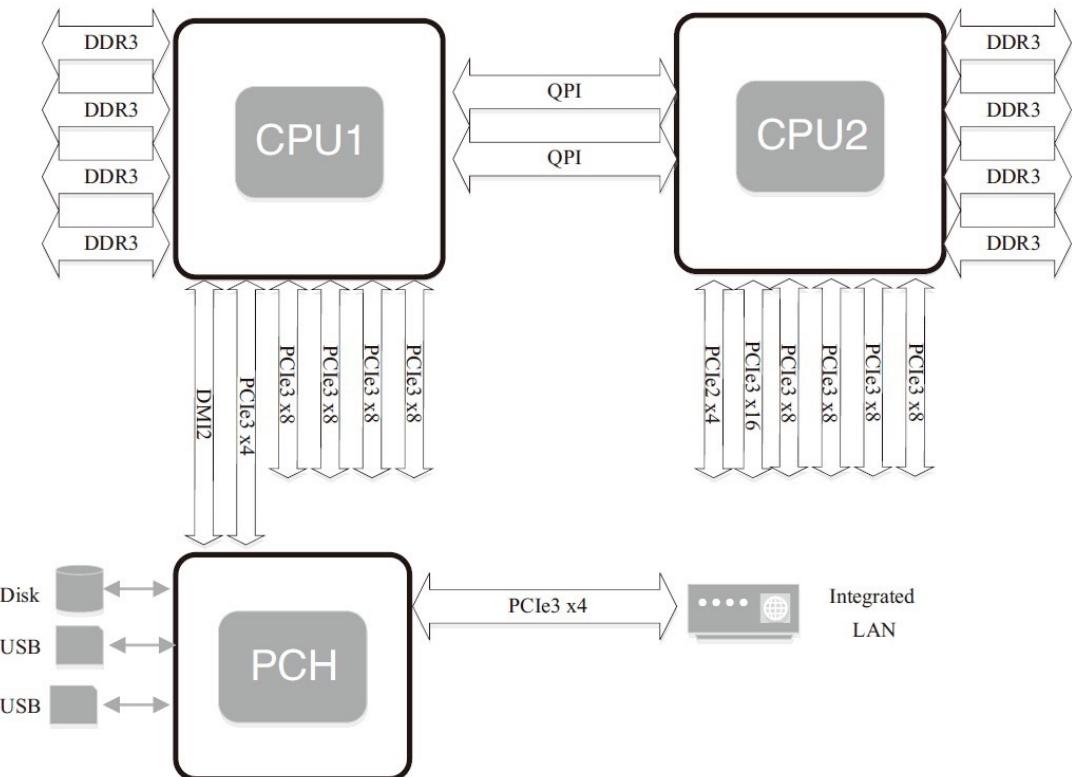
ACL API	说 明
rte_acl_create	创建 AC 的上下文
rte_acl_add_rules	增加规则到 AC 的上下文
rte_acl_build	创建 AC 的运行时结构体
rte_acl_classify	匹配 AC 的规则

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

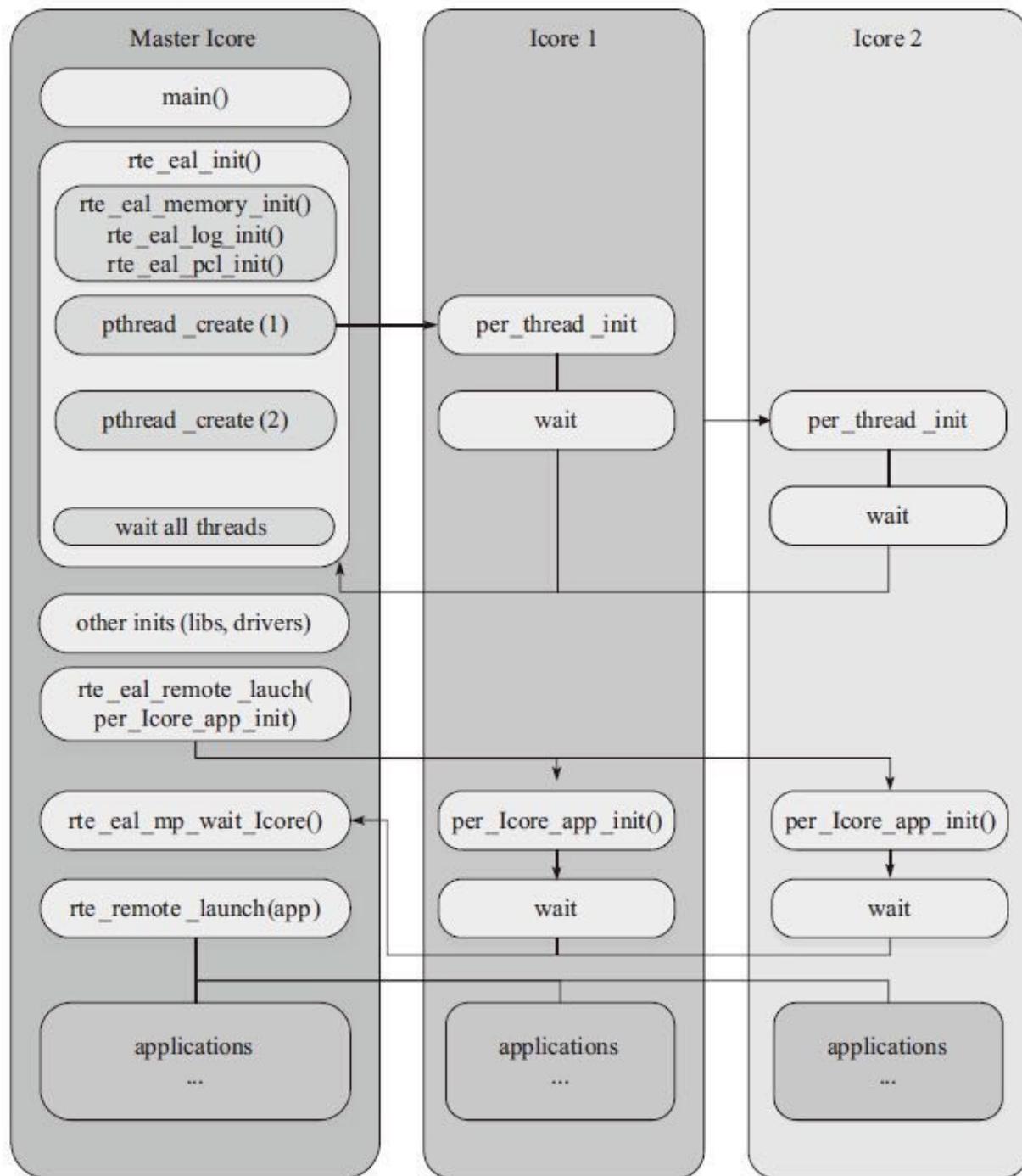
- `sched_set_affinity()`、`sched_get_affinity()` 内核函数
- `taskset` 命令
- `isolcpus` 内核启动参数：CPU绑定之后依然是有可能发生线程切换，可以借助 `isolcpus=2,3` 将cpu从内核调度系统中剥离。

## 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。

## 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。如果否，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。

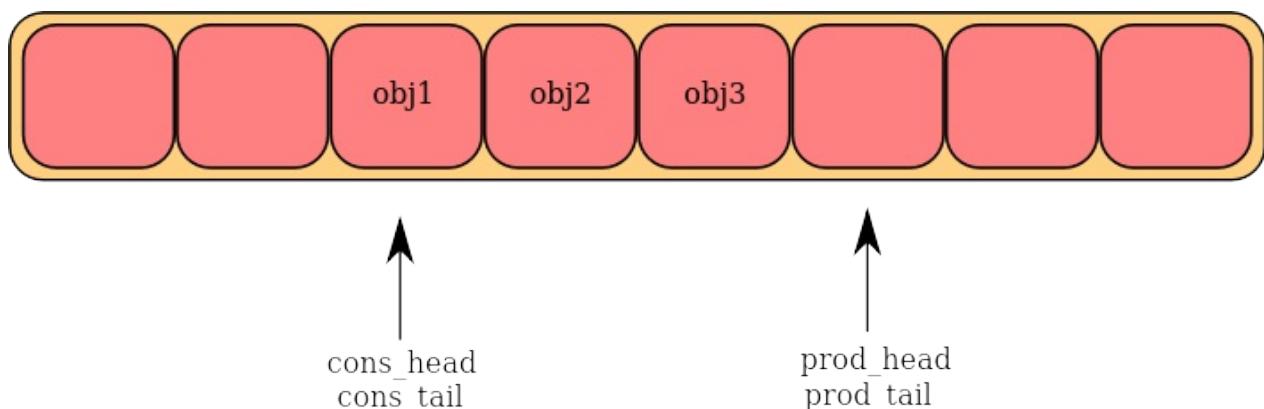
基于无锁环形缓冲的原理，Intel DPDK提供了一套无锁环形缓冲区队列管理代码，支持单生产者产品入列，单消费者产品出列；多名生产者产品入列，多名消费者出列操作。

# 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

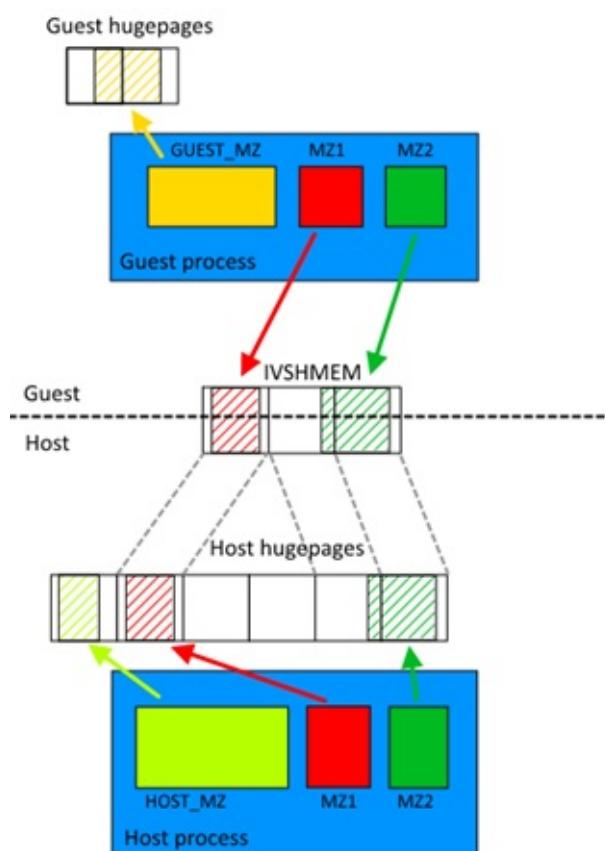
- Multi-consumer or single-consumer dequeue
- Multi-producer or single-producer enqueue
- Bulk dequeue - Dequeues the specified count of objects if successful; otherwise fails
- Bulk enqueue - Enqueues the specified count of objects if successful; otherwise fails
- Burst dequeue - Dequeue the maximum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



图片来源 [Ring Library - dpdk.org](https://ring.ringlib.org/)

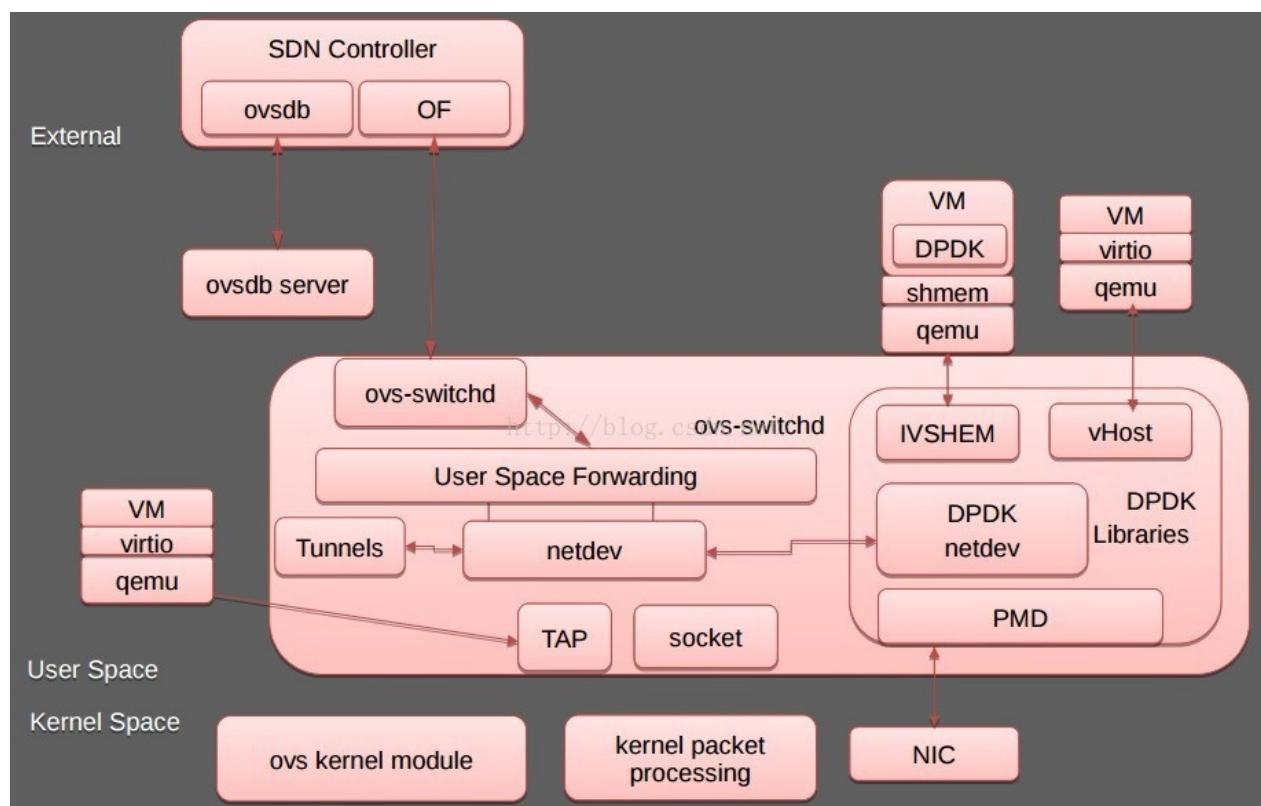
## ivshmem

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。



图片来源[DPDK IVSHMEM Library](#)

DPDK ivshmem：



图片来源[使用OVS DPDK \(by quqi99\)](#)

## ivshmem使用示例

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

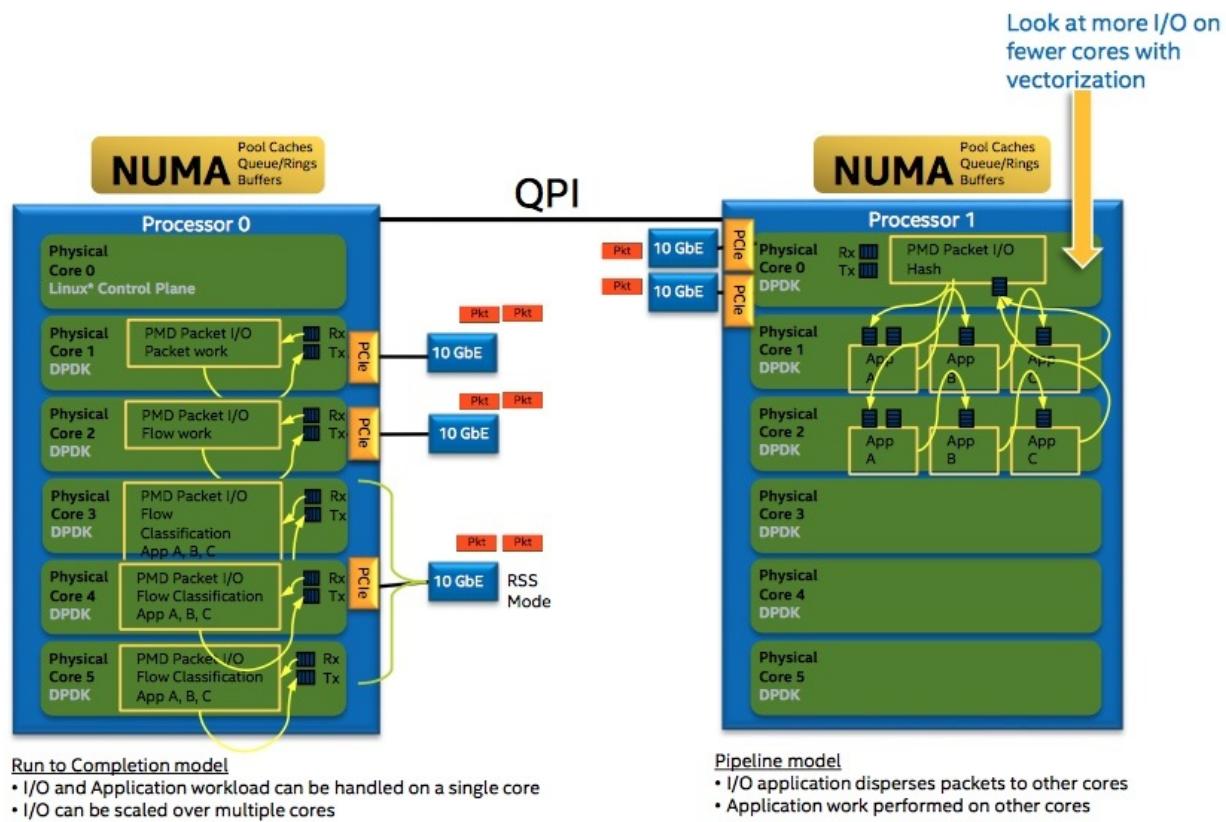
# start VM
qemu-system-x86_64 -hda mg -L pc-bios/ --smp 4 -chardev socket,path=/tmp/nahanni,id=nahanni-device ivshmem,chardev=nahanni,size=32m,msi=off -serial telnet:0.0.0.0:4000,server,nowait,nodelay-enable-kvm&

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

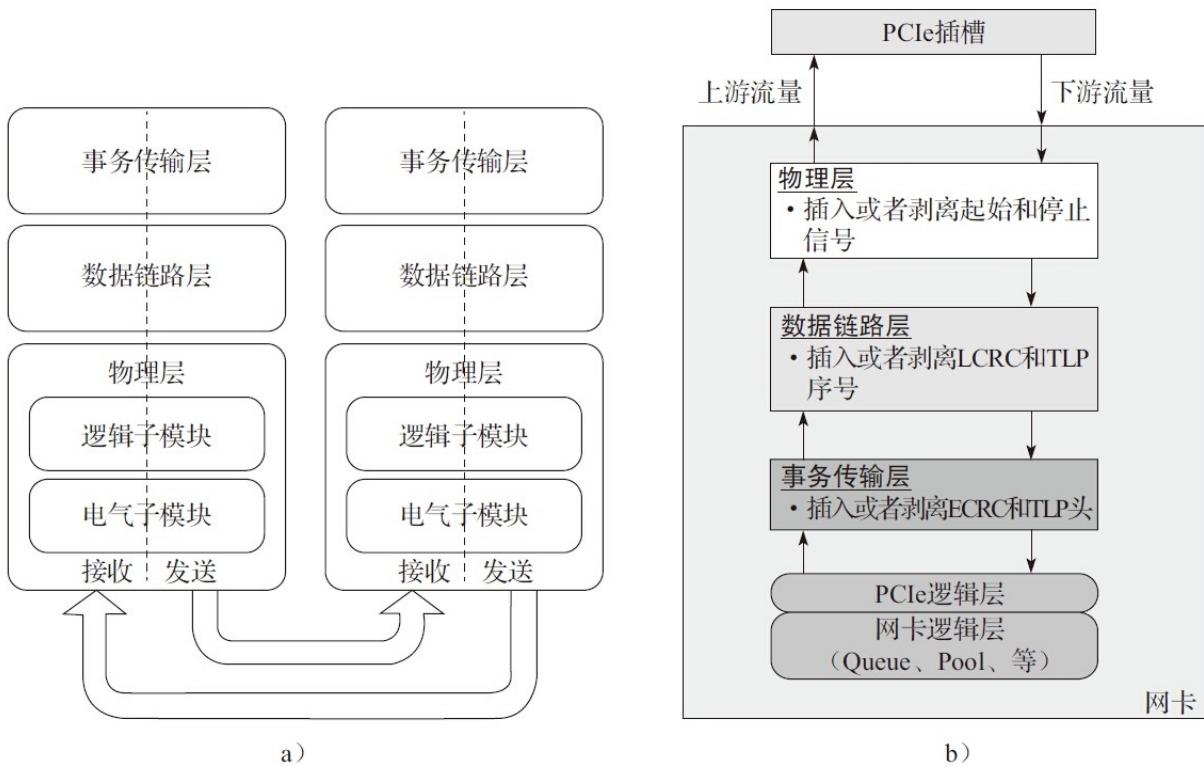
### 参考文档

- [DPDK Ring Library](#)
- [DPDK IVSHMEM Library](#)
- [Accelerating the Path to the Guest](#)
- [DPDK Summit - 08 Sept 2014 - VMware and Intel - Using DPDK In A Virtual World](#)

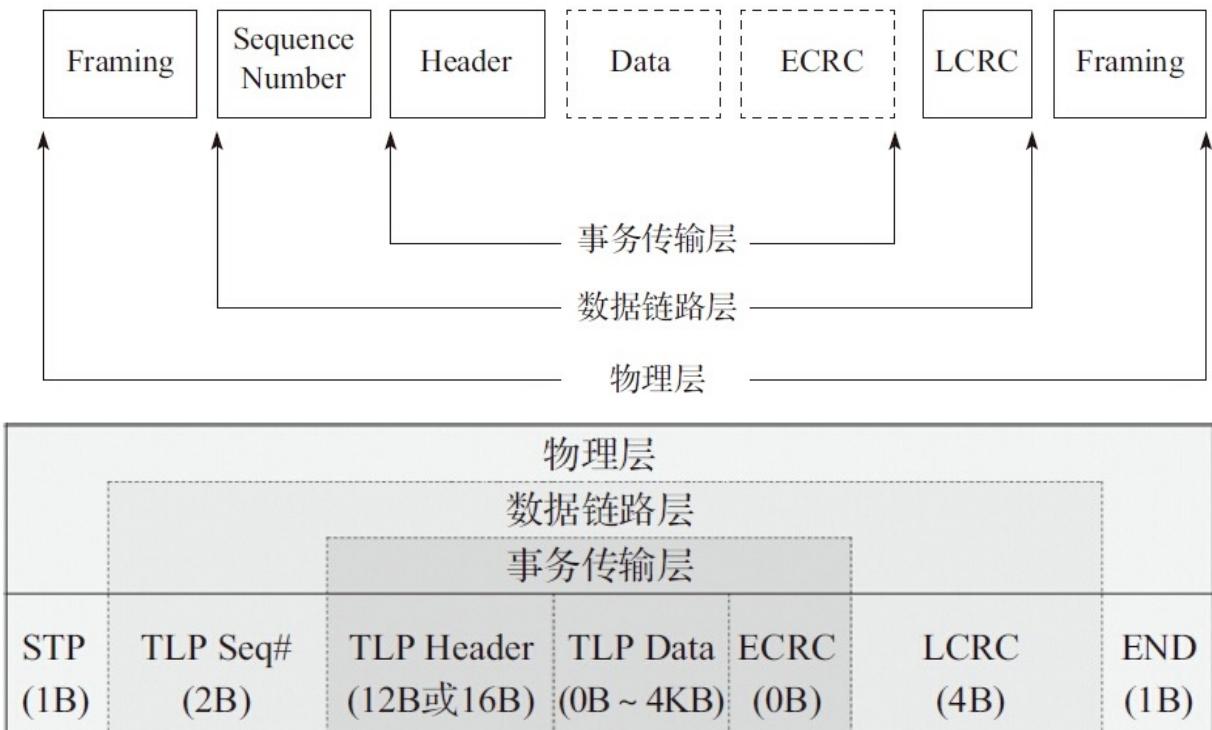
# PCIe



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。



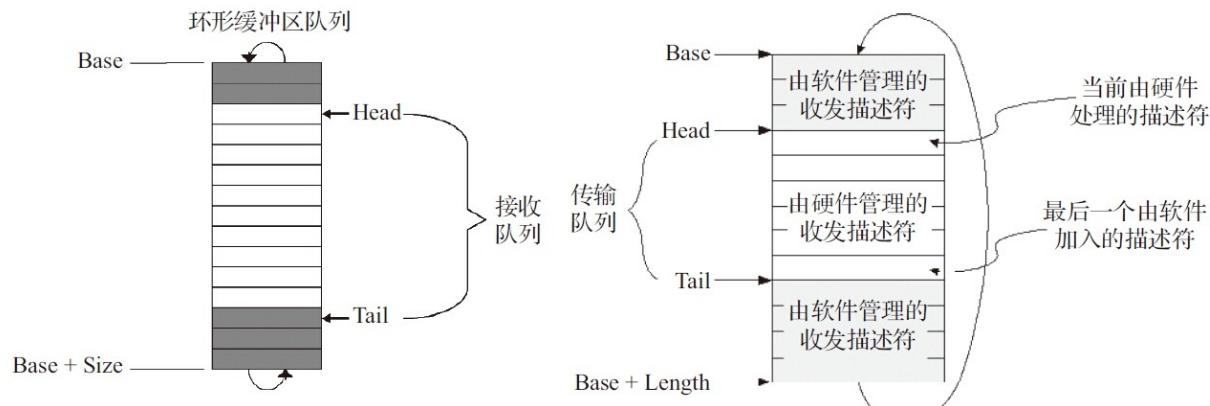
PCIe包格式示例，对于一个完整的TLP包来说，除去有效载荷，额外还有24B的开销（TLP头部以16B计算）。



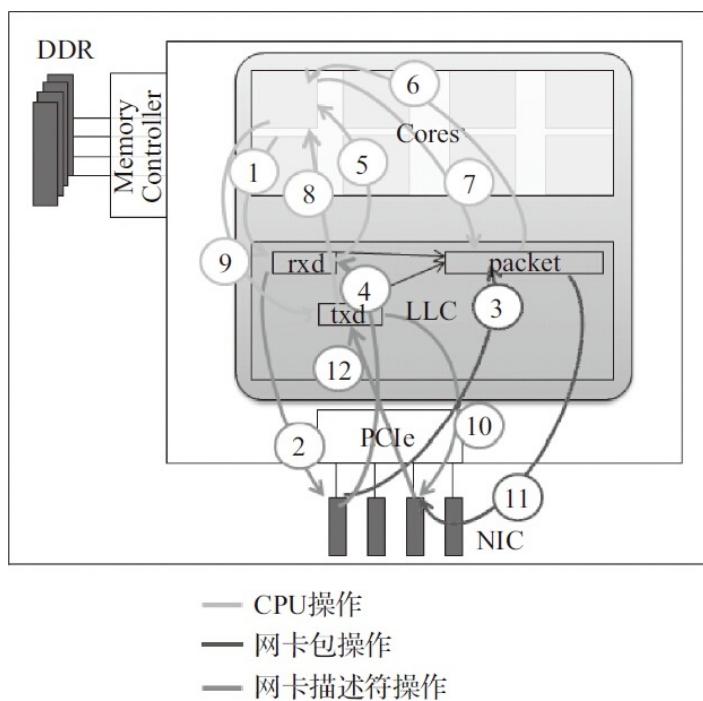
要查看特定 PCIe 设备的链路能力和当前速率，可以用 Linux 工具 `lspci` 读取 PCIe 的配置寄存器。

## 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。

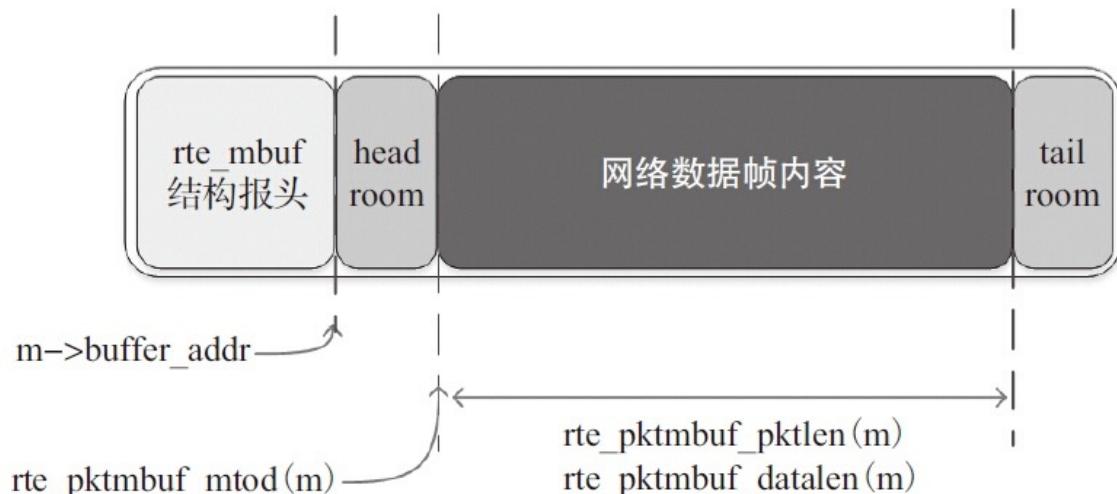
2. 提高 PCIe 传输的效率。如果能把4个操作合并成整 Cache Line 大小来作为 PCIe 的事务请求（PCIe 净荷为 64Byte），带宽利用率就能得到提升。
3. 尽量避免 Cache Line 的部分写。Cache Line 的部分写会引发内存访问 read-modify-write 的合并操作，增加额外的读操作，也会降低整体性能。所以，DPDK在 Mempool 中分配 buffer 的时候，会要求对齐到 Cache Line 大小。

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  $4000\text{MB/s}/168\text{B}=23.8\text{Mp/s}$ 。

## Mbuf

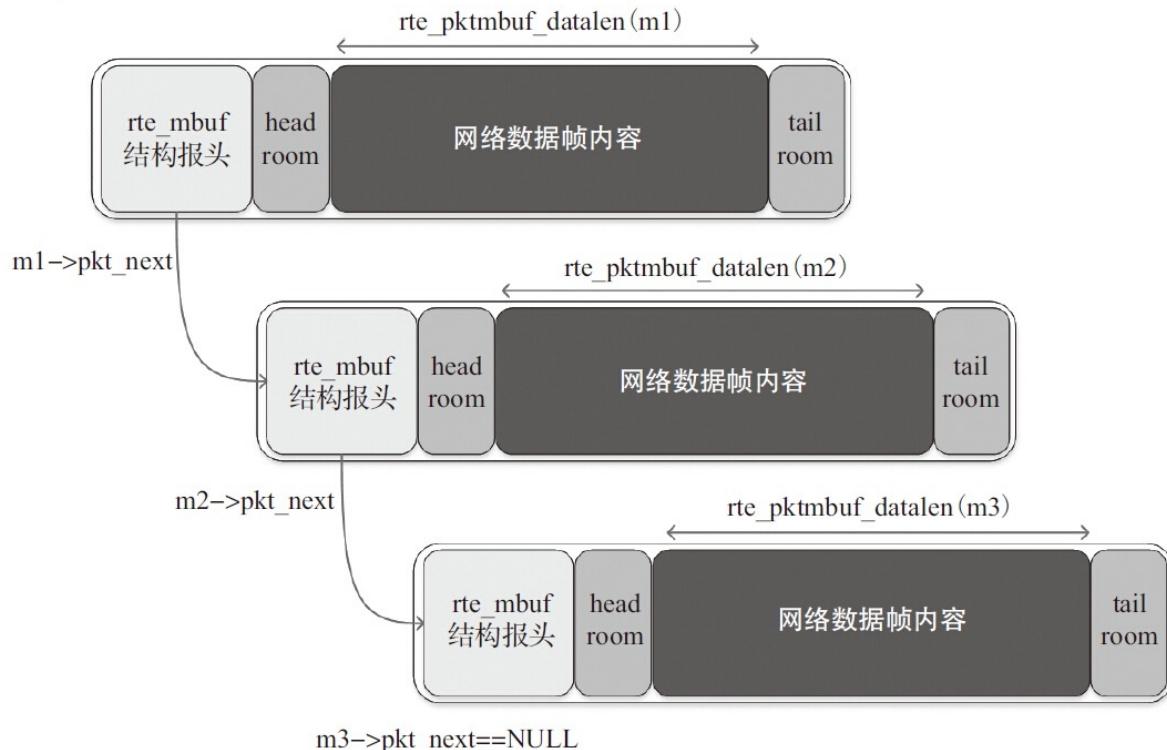
为了高效访问数据，DPDK 将内存封装在 Mbuf（`struct rte_mbuf`）结构体内。Mbuf 主要用来封装网络帧缓存，也可用来封装通用控制信息缓存（缓存类型需使用 `CTRL_MBUF_FLAG` 来指定）。网络帧元数据的一部分内容由 DPDK 的网卡驱动写入。这些内容包括 VLAN 标签、RSS 哈希值、网络帧入口端口号以及巨型帧所占的 Mbuf 个数等。对于巨型帧，网络帧元数据仅出现在第一个帧的 Mbuf 结构中，其他的帧该信息为空。

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)`

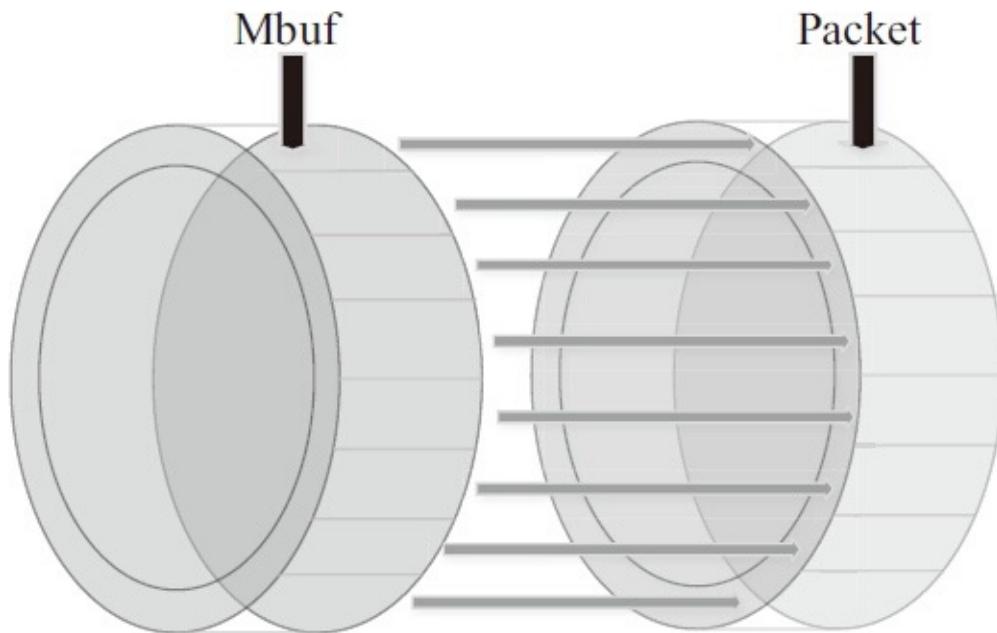


## Mempool

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。

## 参考

- [lspci](#)
- [Understanding DPDK](#)

# 网卡性能优化

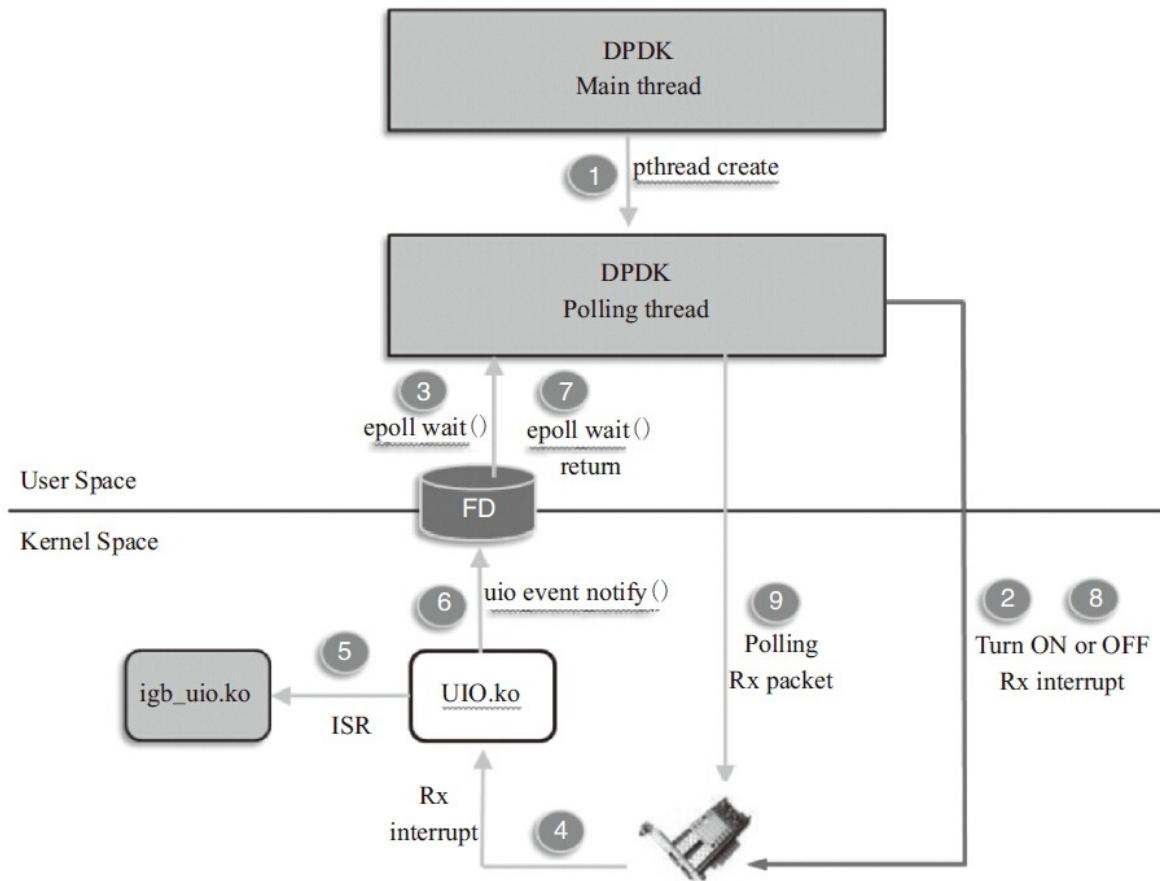
运行在操作系统内核态的网卡驱动程序基本都是基于异步中断处理模式，而DPDK采用了轮询或者轮询混杂中断的模式来进行收包和发包。DPDK起初的纯轮询模式是指收发包完全不使用任何中断，集中所有运算资源用于报文处理。但这不是意味着DPDK不可以支持任何中断。根据应用场景需要，中断可以被支持，最典型的就是链路层状态发生变化的中断触发与处理。

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

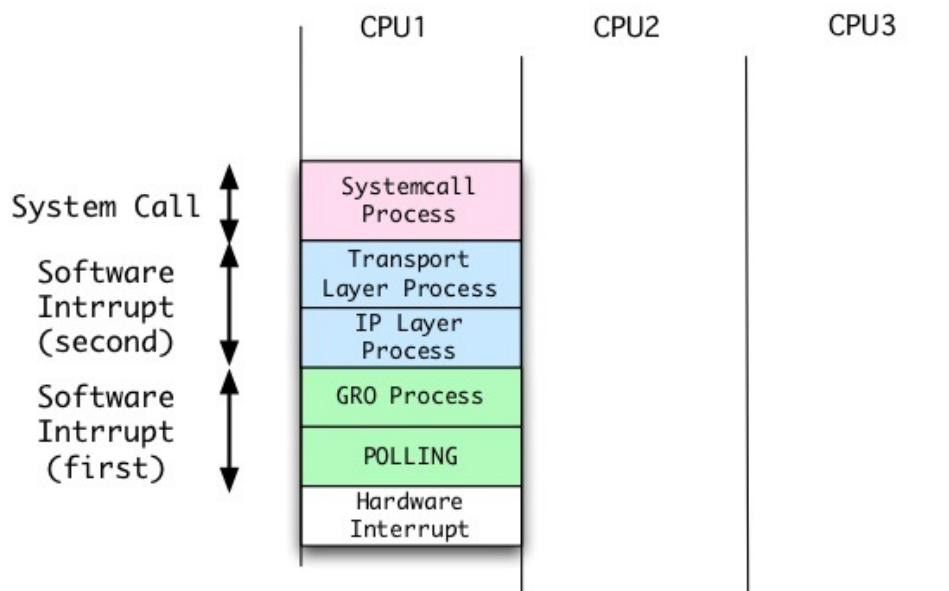
- 收包队列长度：DPDK很多示例程序里面默认的收包队列长度是128，这就是表示为每一个收包队列都分配128个收包描述符，这是一个适应大多数场景经验值。但是在某些更高速率的网卡收包的情况下，128就可能不一定够了，或者在某些场景下发现丢包现象比较容易的时候，就需要考虑使用更长的收包队列，例如可以使用512或者1024。
- 发包队列长度：DPDK的示例程序里面默认的发包队列长度使用的是512，这就表示为每一个发包队列都分配512个发包描述符，这是一个适用大部分场合经验值。当处理更高速率的网卡设备时，或者发现有丢包的时候，就应该考虑更长的发包队列，例如1024。
- 收包队列可释放描述符数量阈值（`rx_free_thresh`）：DPDK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

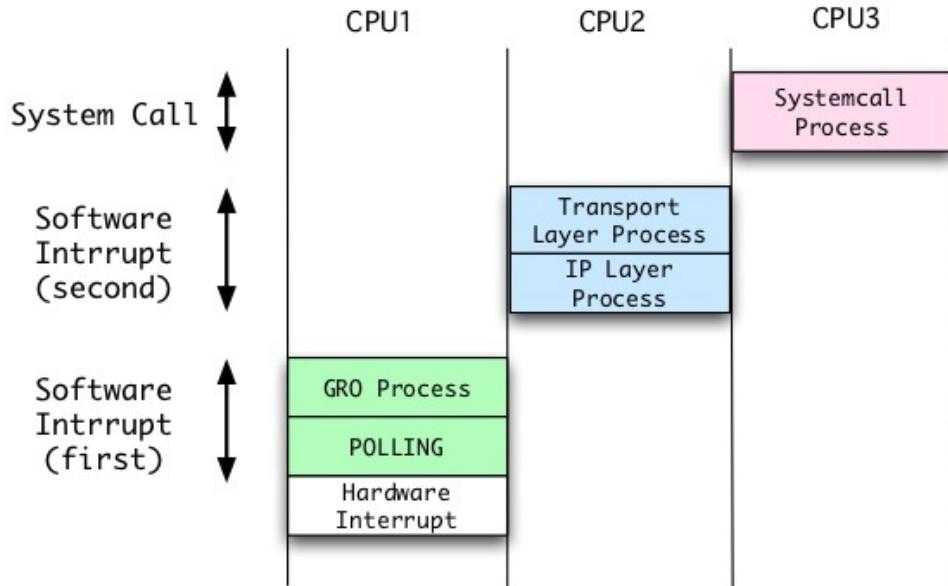
网卡多队列，顾名思义，也就是传统网卡的 DMA 队列有多个，网卡有基于多个 DMA 队列的分配机制。多队列网卡已经是当前高速率网卡的主流。

## RPS

### No RPS/RFS



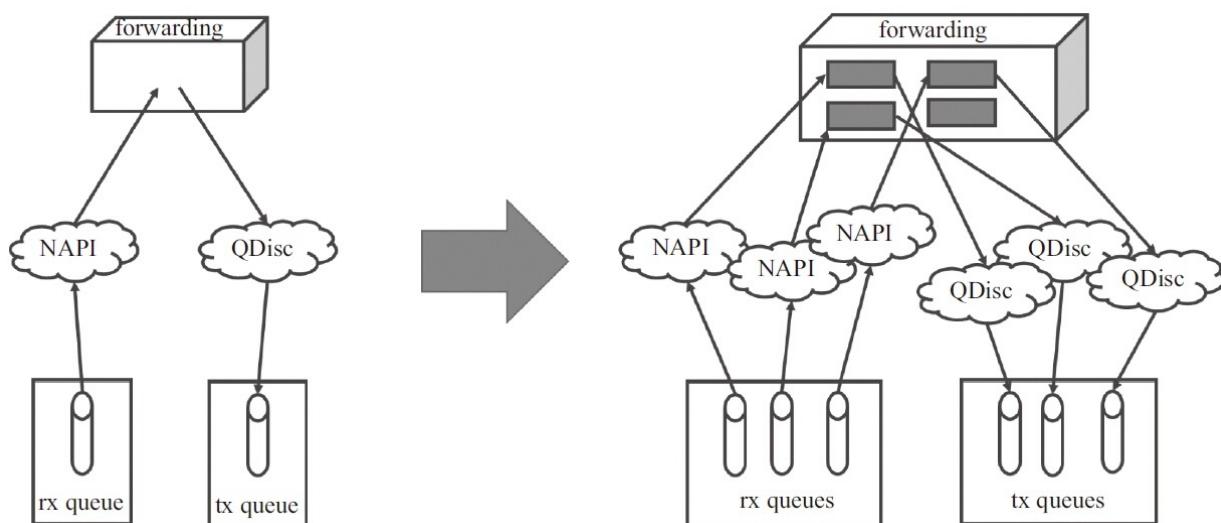
# RPS Only



4

图片来源 [RPS/RFS](#)

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 SPOR T，目的IP地址 DIP，目的四层端口 DPOR T）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。

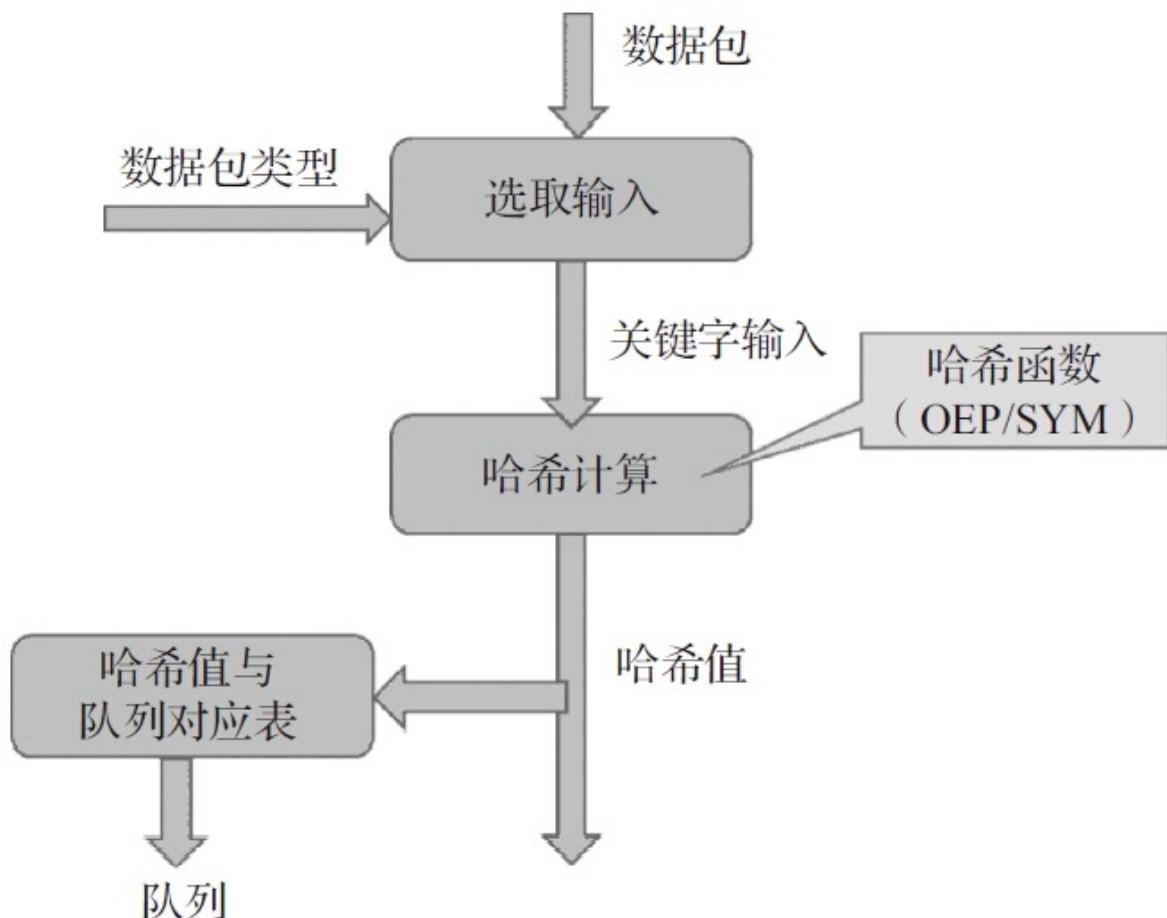
网卡是如何将网络中的报文分发到不同的队列呢？常用的方法有微软提出的RSS与英特尔提出的 Flow Director 技术，前者是根据哈希值希望均匀地将包分发到多个队列中。后者是基于查找的精确匹配，将包分发到指定的队列中。此外，网卡还可以根据优先级分配队列提供对 QoS 的支持。

## 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。



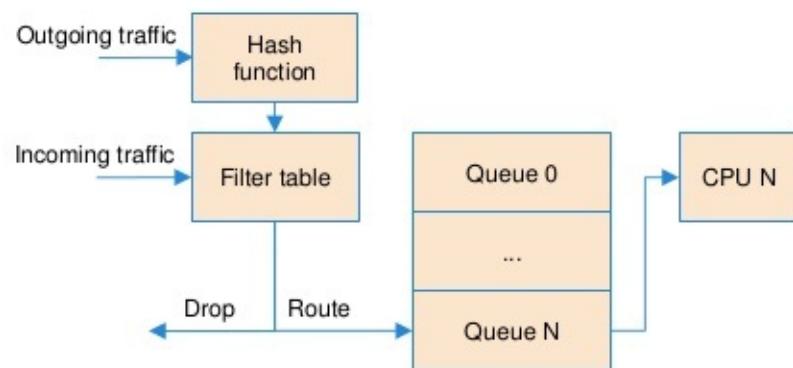
关键字是如何确定的呢？

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。

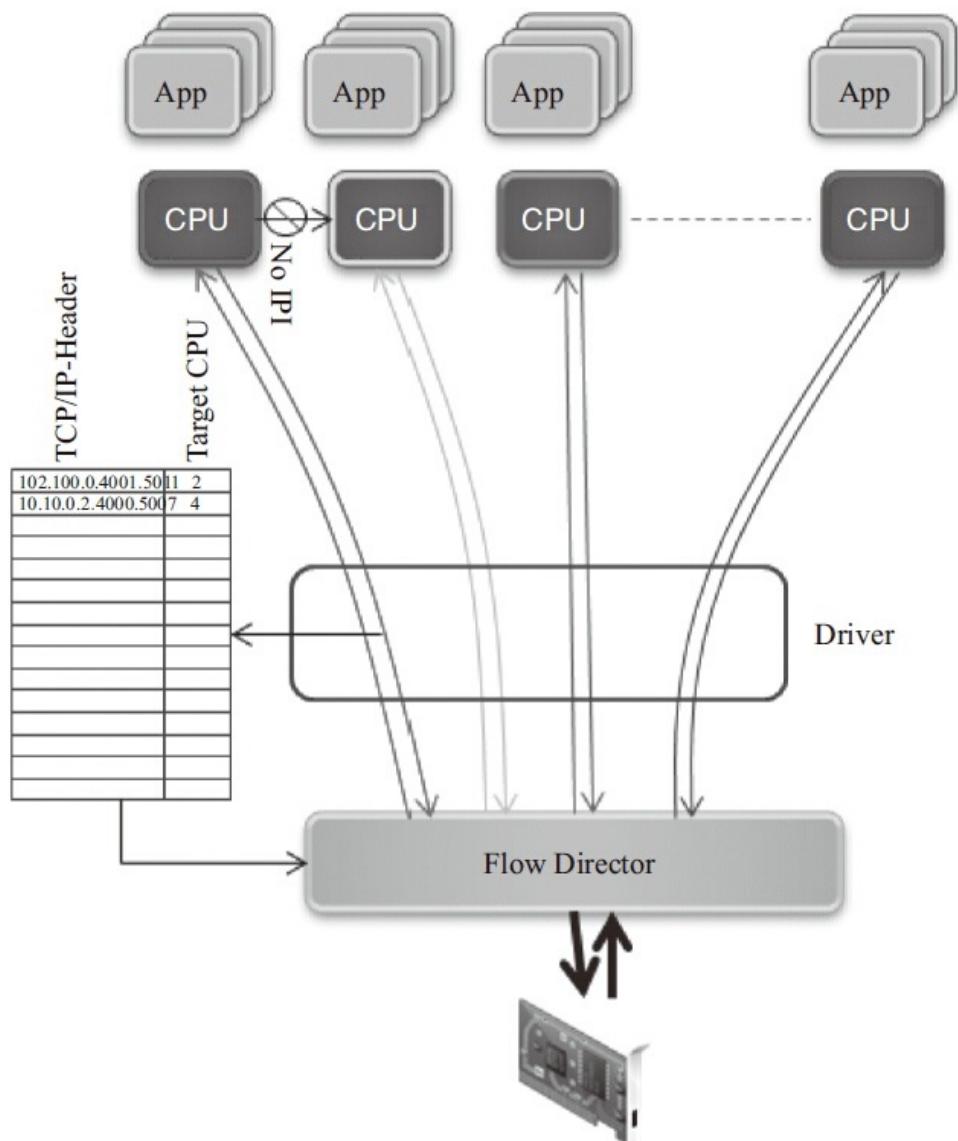
## Flow Director

# Flow director



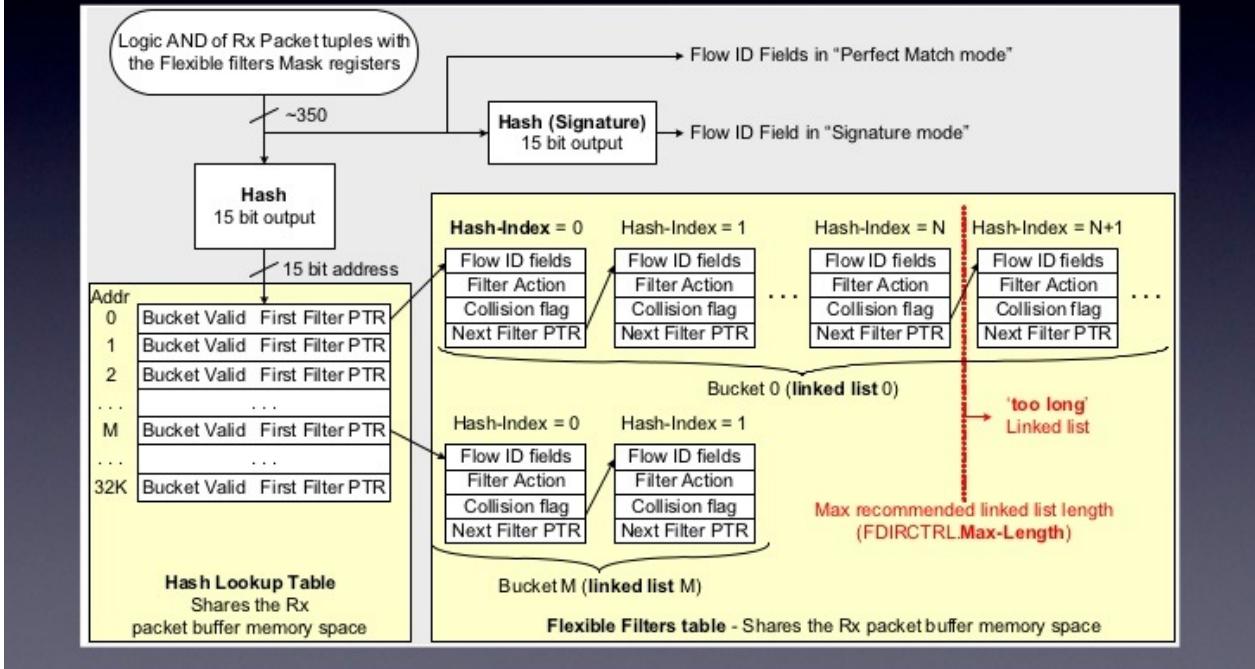
图片来源[Understanding DPDK](#)

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。

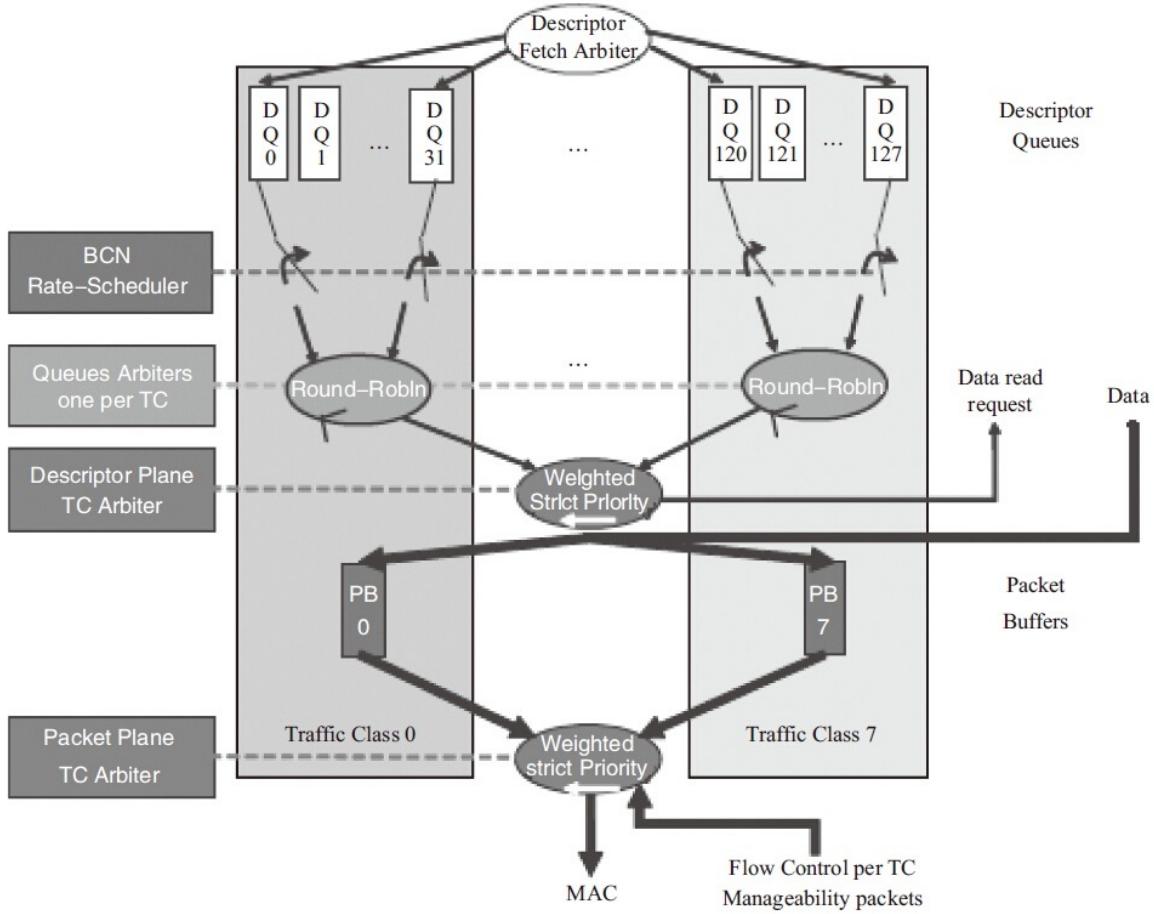
# Flow Director



图片来源Intel 82599 10GbE Controllerで遊ぼう

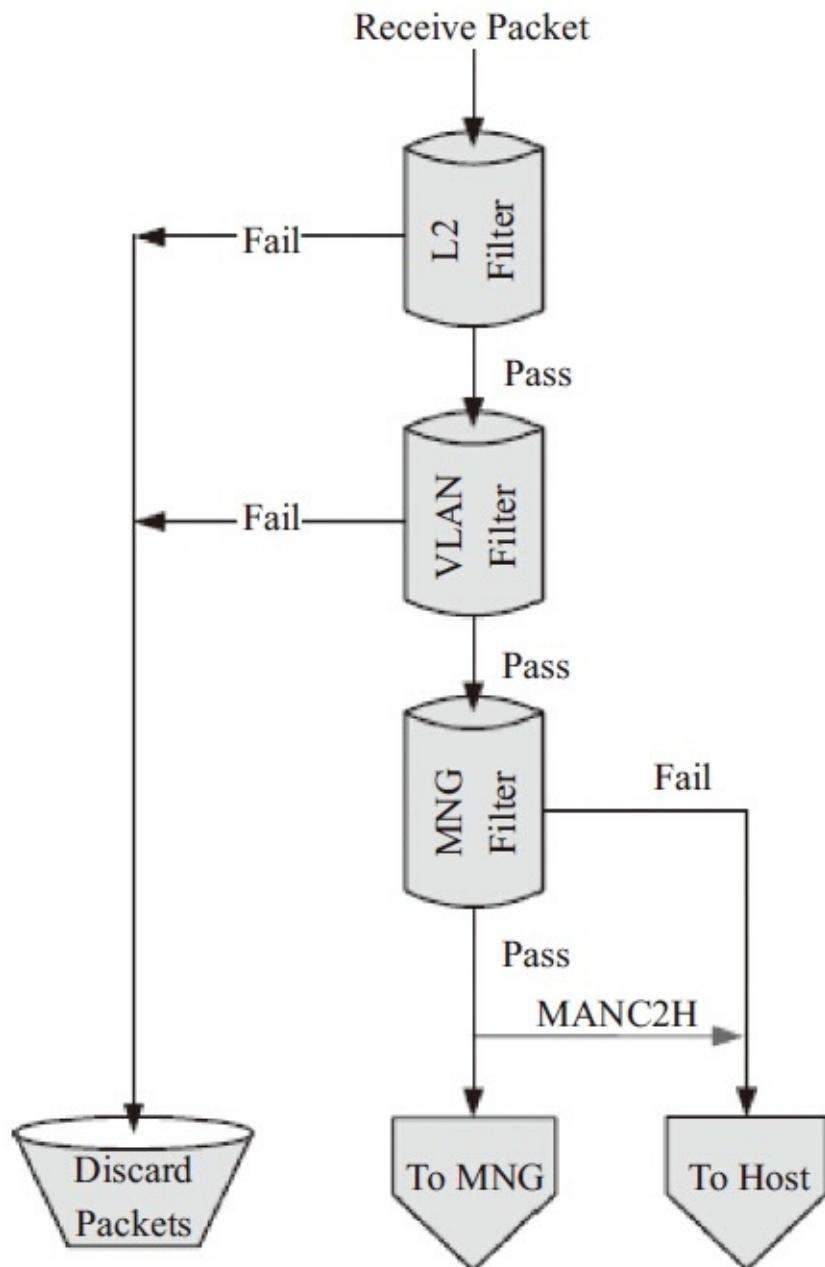
## 服务质量

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。

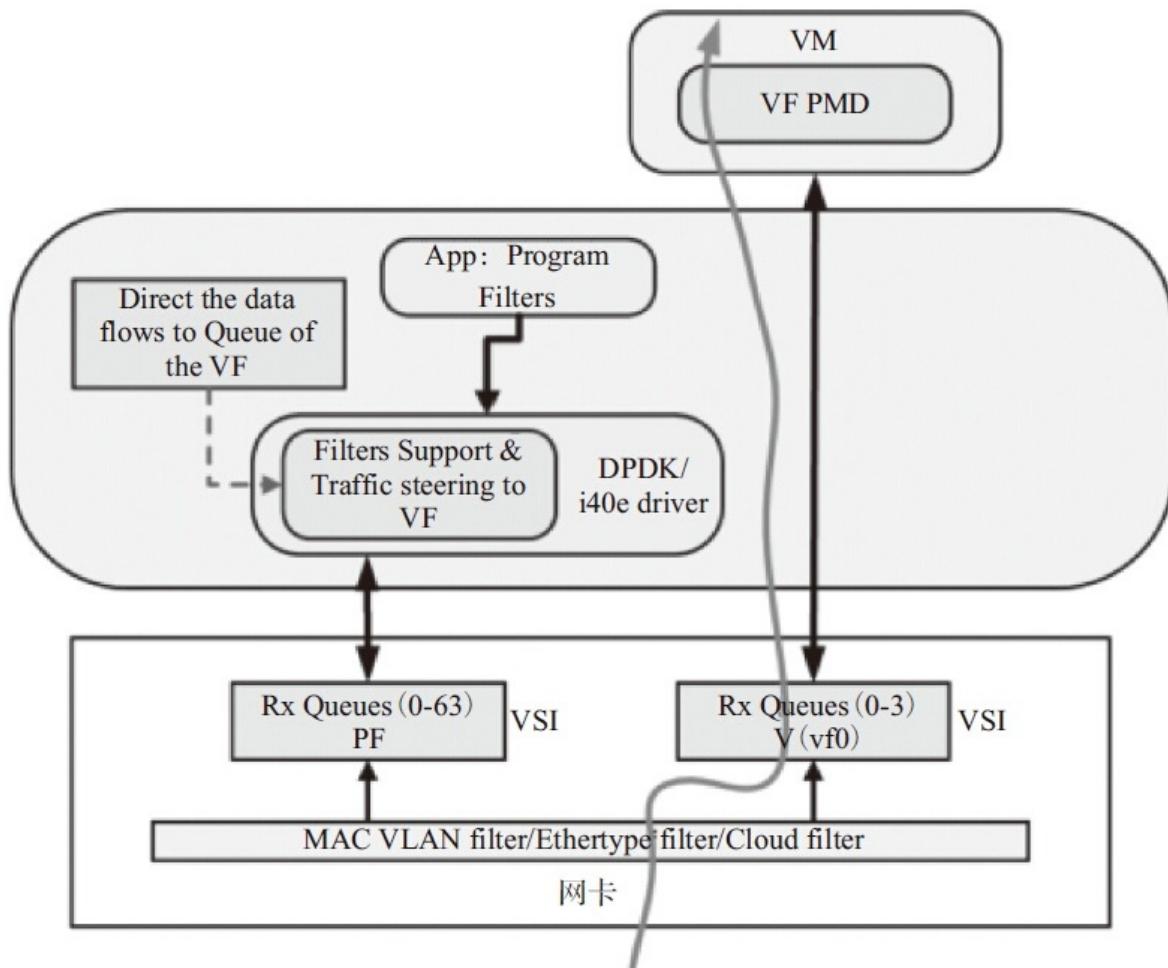
```

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

```



## 参考

- RPS/RFS
- Understanding DPDK
- Intel 82599 10GbE Controllerで遊ぼう



# 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。

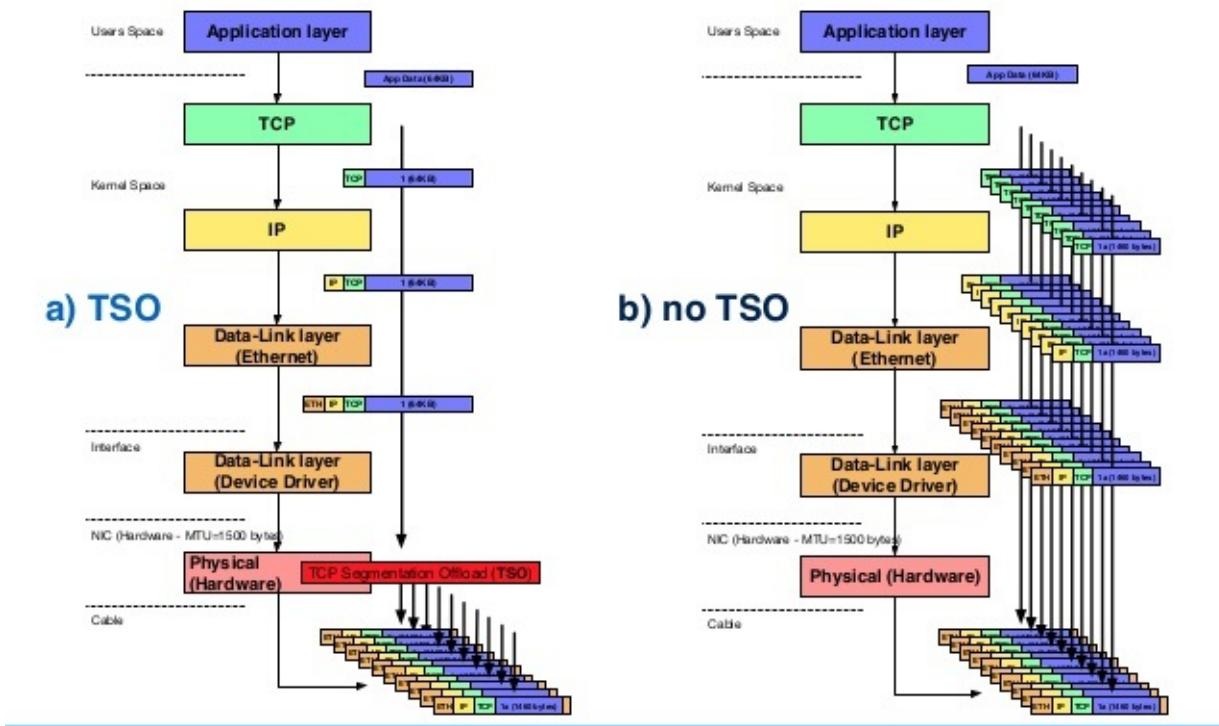
在 dpdk/testpmd 中，可以使用相关的命令行来使用 VXLAN 和 NVGRE 的数据流重定向功能，如下所示：

```
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



# TCP Segmentation Offload

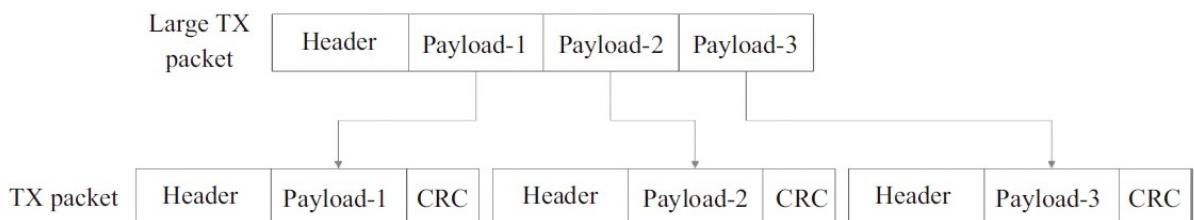


EuroBSDcon 2014 - Stefano Garzarella - [stefano.garzarella@gmail.com](mailto:stefano.garzarella@gmail.com)

5

图片来源 [TCP Segment Offload for DPDK vRouter](#)

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。



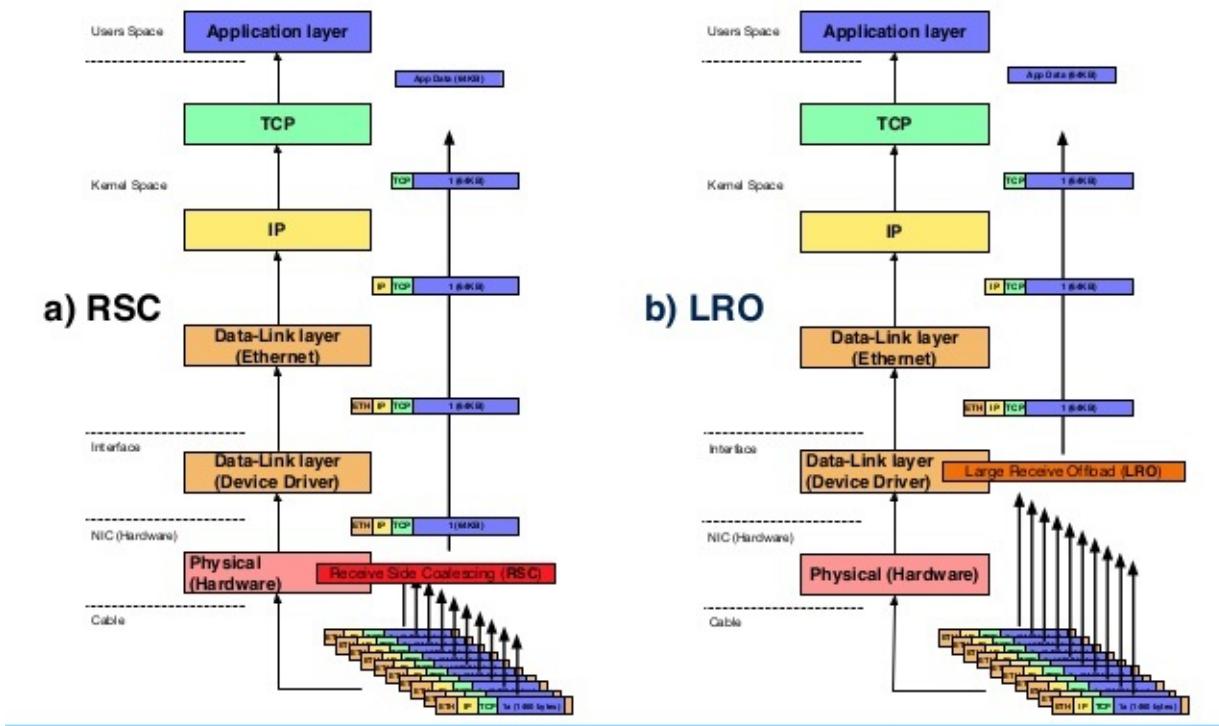
在 `dpdk/testpmd` 中提供了两条 tso 相关的命令行：

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



# Large Receive Offload

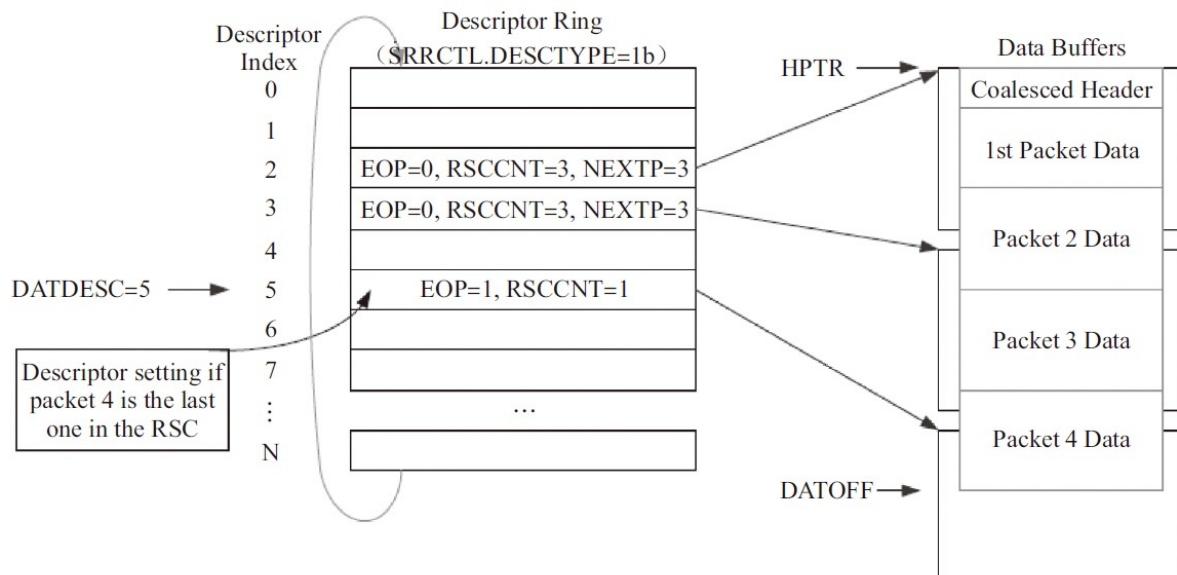


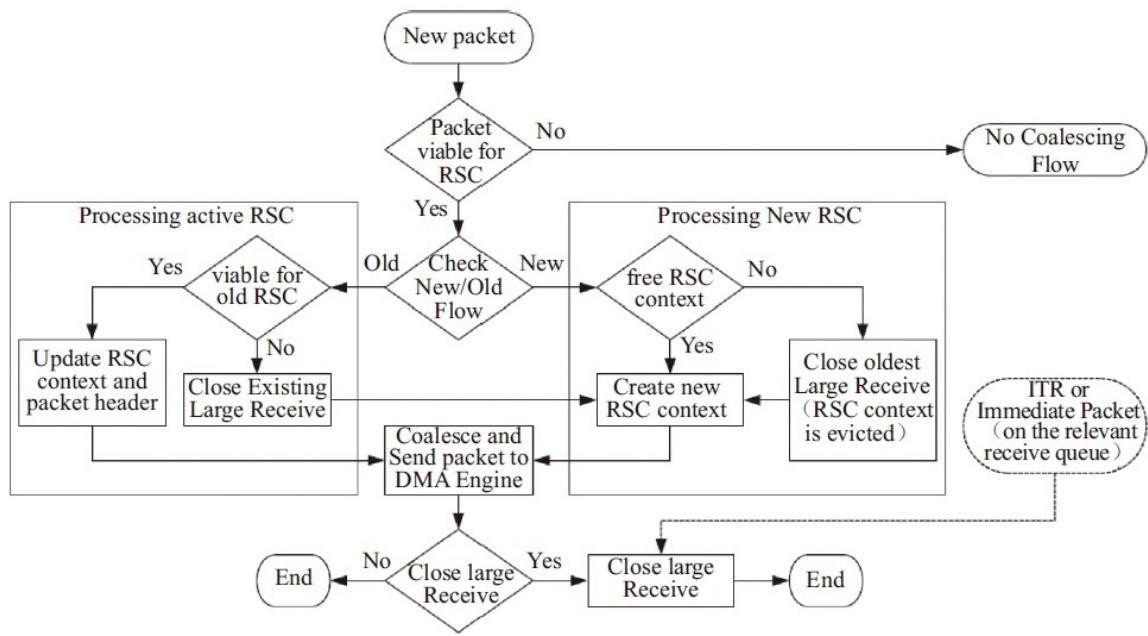
EuroBSDcon 2014 - Stefano Garzarella - [stefano.garzarella@gmail.com](mailto:stefano.garzarella@gmail.com)

10

图片来源 [Software segmentation offloading for FreeBSD by Stefano Garzarella](#)

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。



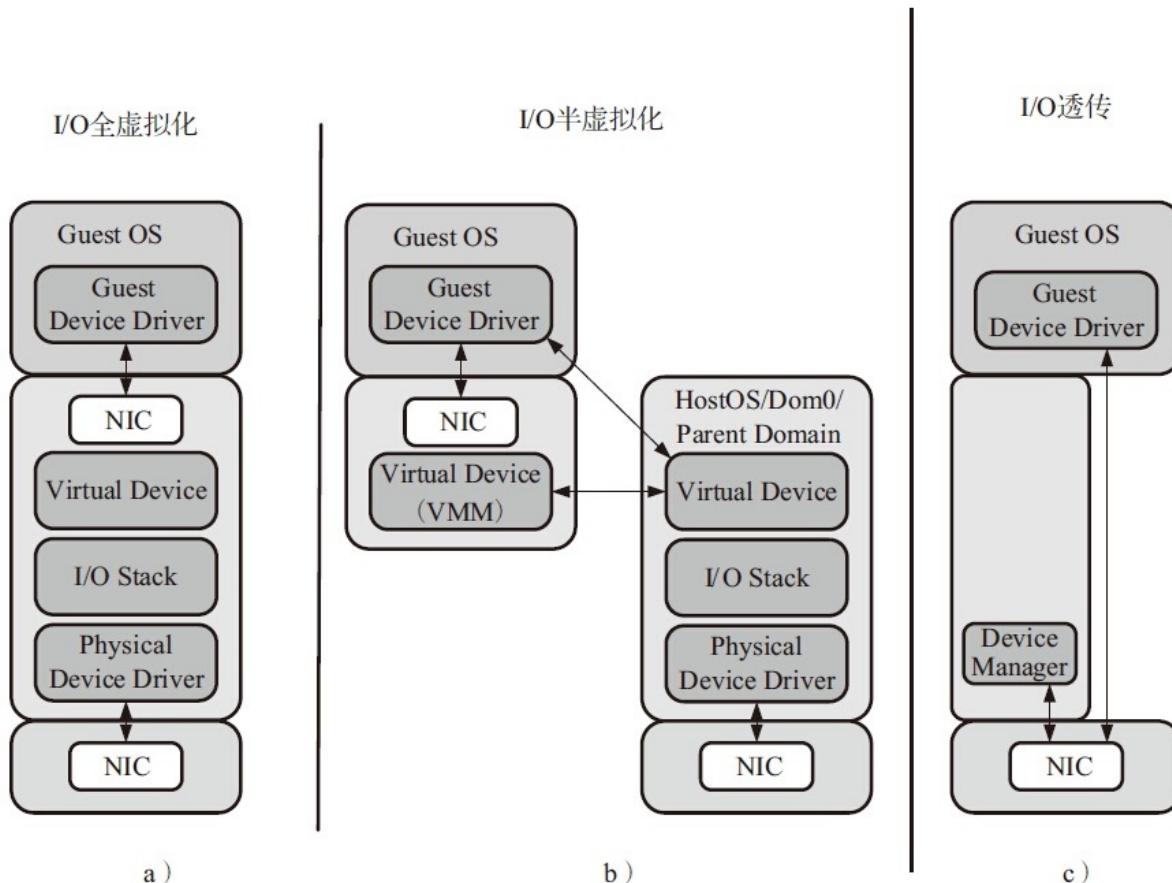


## 参考

- [TCP Segment Offload for DPDK vRouter](#)
- [Software segmentation offloading for FreeBSD by Stefano Garzarella](#)

# 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。
- I/O 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。

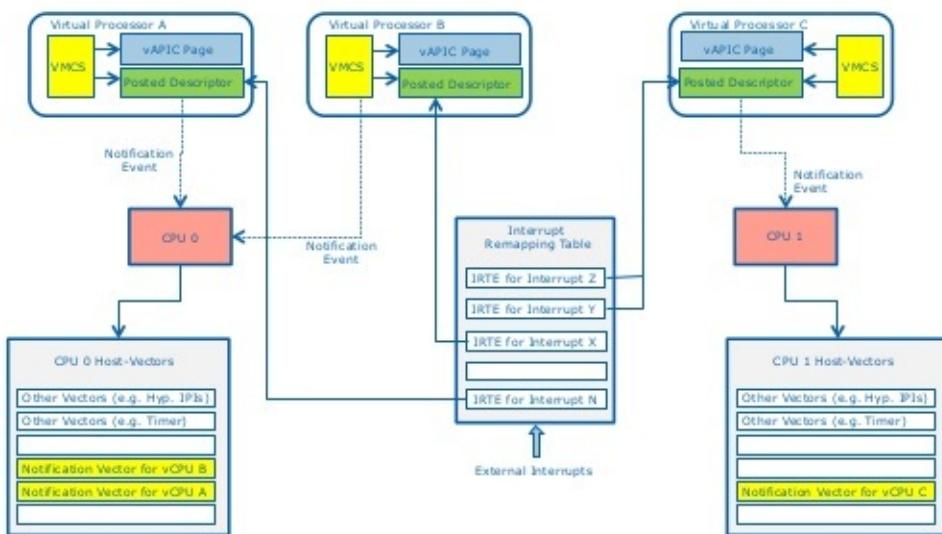
DPDK 支持半虚拟化的前端 virtio 和后端 vhost，并且对前后端都有性能加速的设计，这些将分别在后面两章介绍。而对于 I/O 透传，DPDK 可以直接在客户机里使用，就像在宿主机里，直接接管物理设备，进行操作。

## 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 I/O 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 Posted-Interrupts Architecture

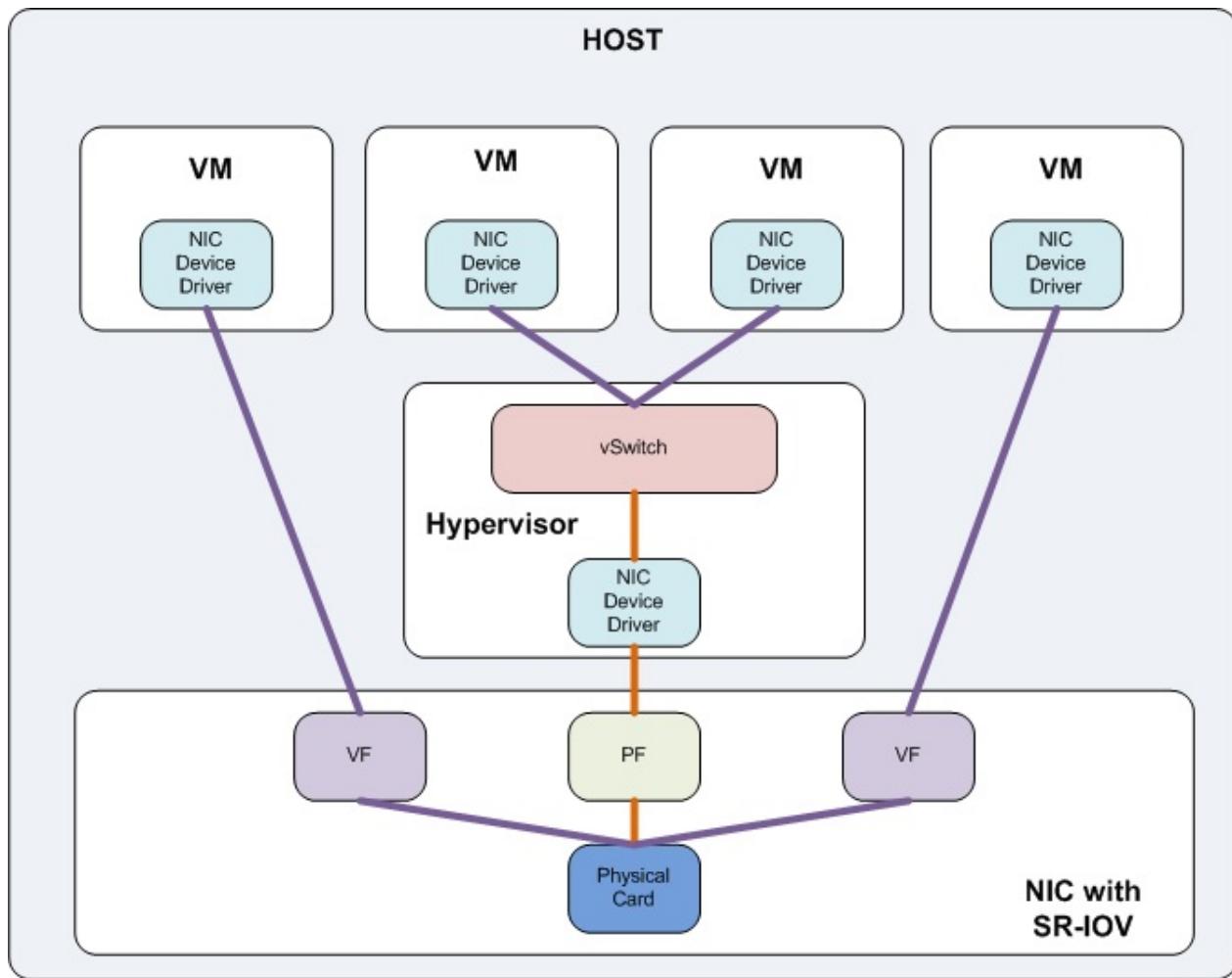


图片来源XPDS14 - Intel(r) Virtualization Technology for Directed I/O (VT-d) Posted Interrupts - Feng Wu, Intel

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 I/O 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。

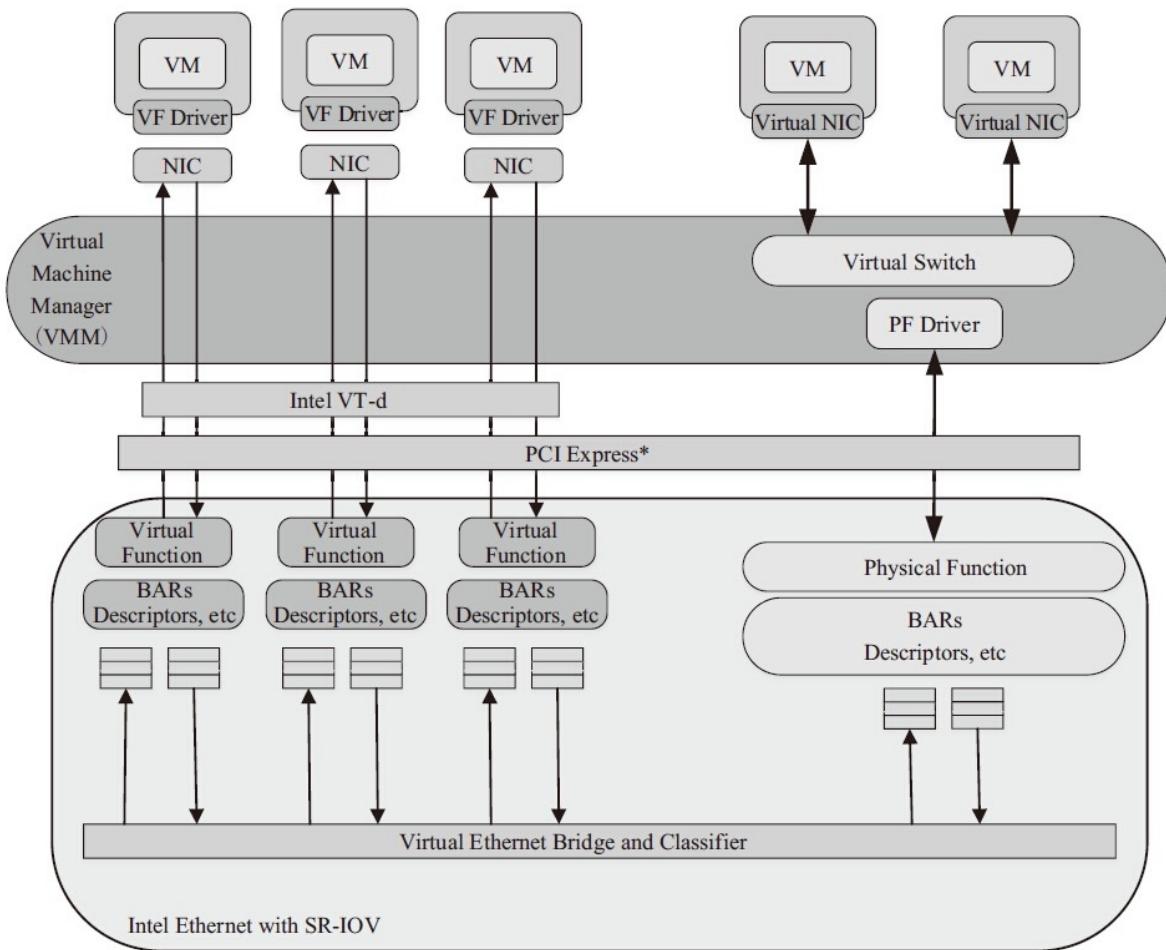
## SR-IOV



图片来源 [I/O Virtualization Overview: CNA, SR-IOV, VN-Tag and VEPA](#)

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：

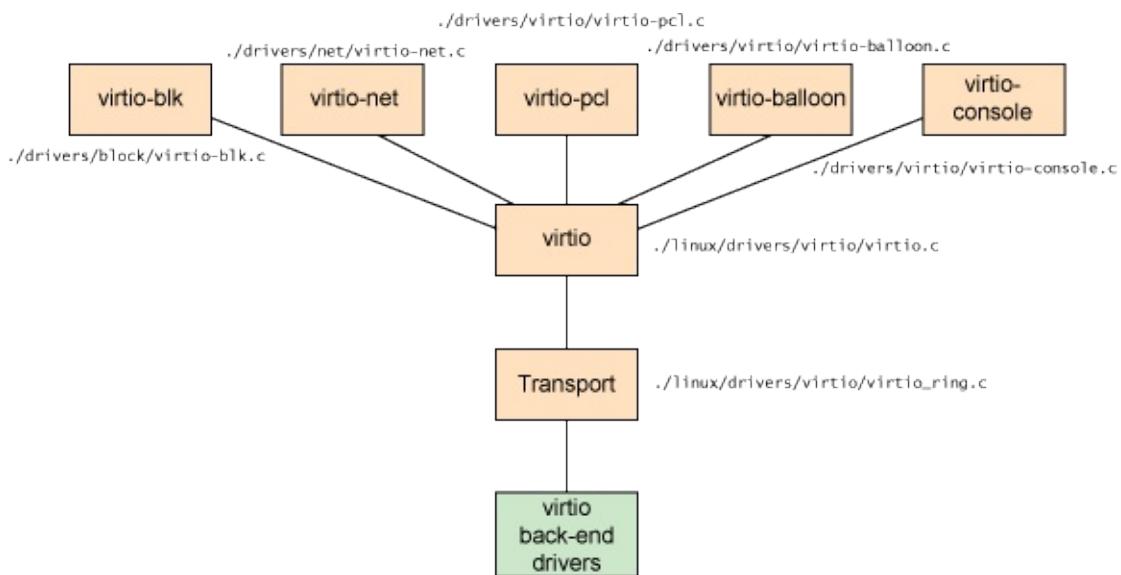
- PF (Physical Function, 物理功能)：这是支持 SR-IOV 扩展功能的 PCIe 功能，主要用于配置和管理 SR-IOV，拥有所有的 PCIe 设备资源。PF 在系统中不能被动态地创建和销毁（PCI Hotplug 除外）。
- VF (Virtual Function, 虚拟功能)：“精简”的 PCIe 功能，包括数据迁移必需的资源，以及经过谨慎精简的配置资源集，可以通过 PF 创建和销毁。



## VMware SR-IOV Architecture

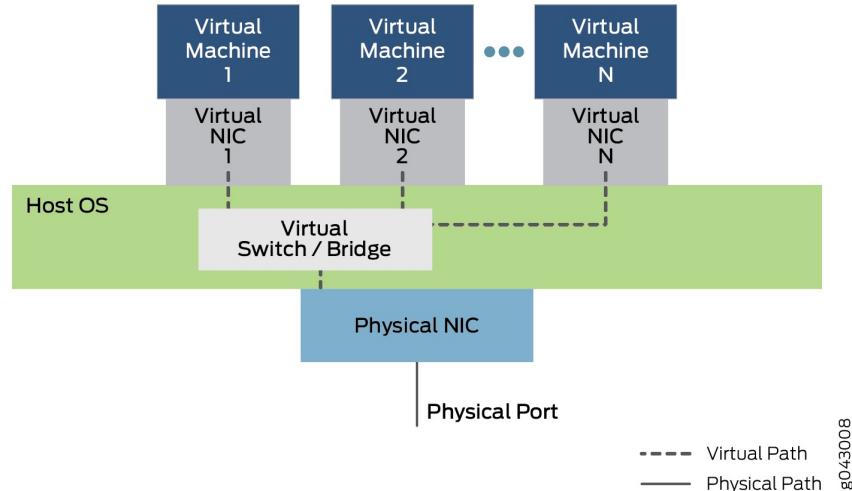
图片来源[SR-IOV Component Architecture and Interaction](#)

**virtio**



图片来源[Virtio: An I/O virtualization framework for Linux](#)

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。



图片来源[Understanding Virtio Usage - Juniper](#)

## 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	rpmmsg
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**

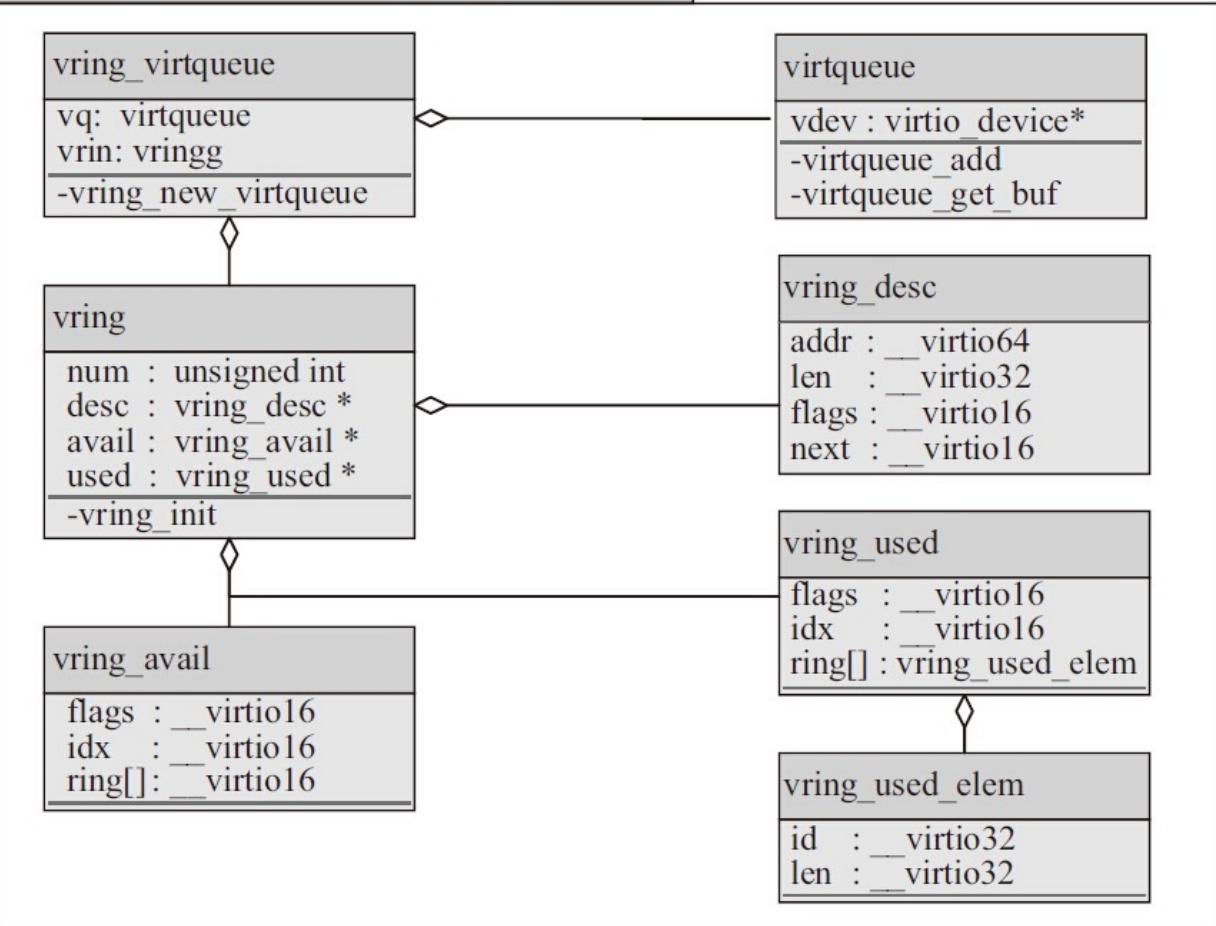
```
-setup_vq
-del_vq
-vq_config_vector
-virtio_pci_modern_probe
```

**virtio\_pci\_legacy\_device**

```
-setup_vq
-del_vq
-vq_config_vector
-virtio_pci_legacy_probe
```

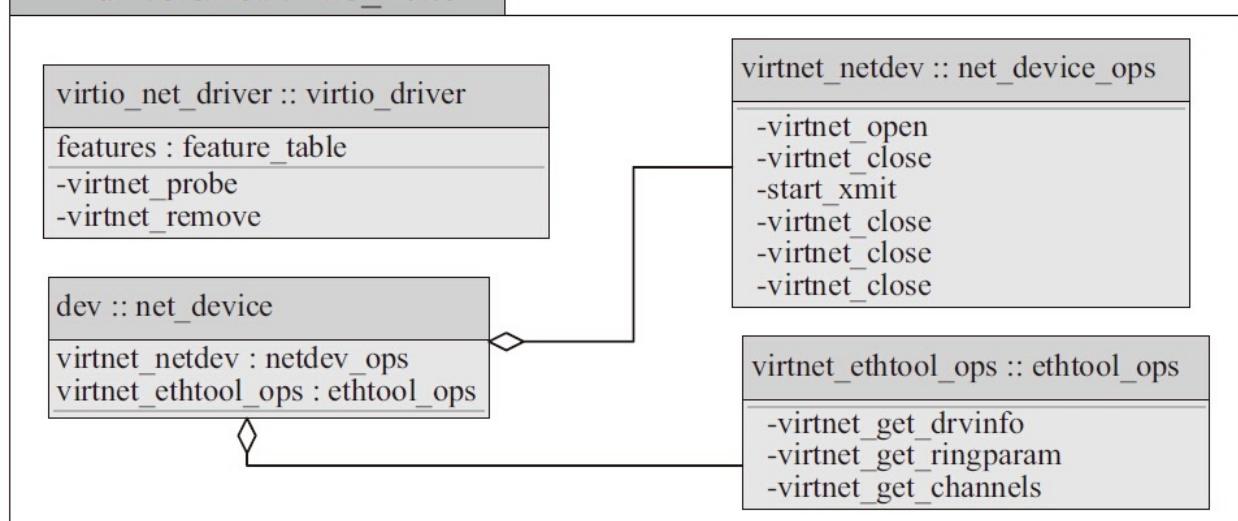
## 虚拟队列层

drivers/virtio/virtio\_ring.c



## 网络设备层

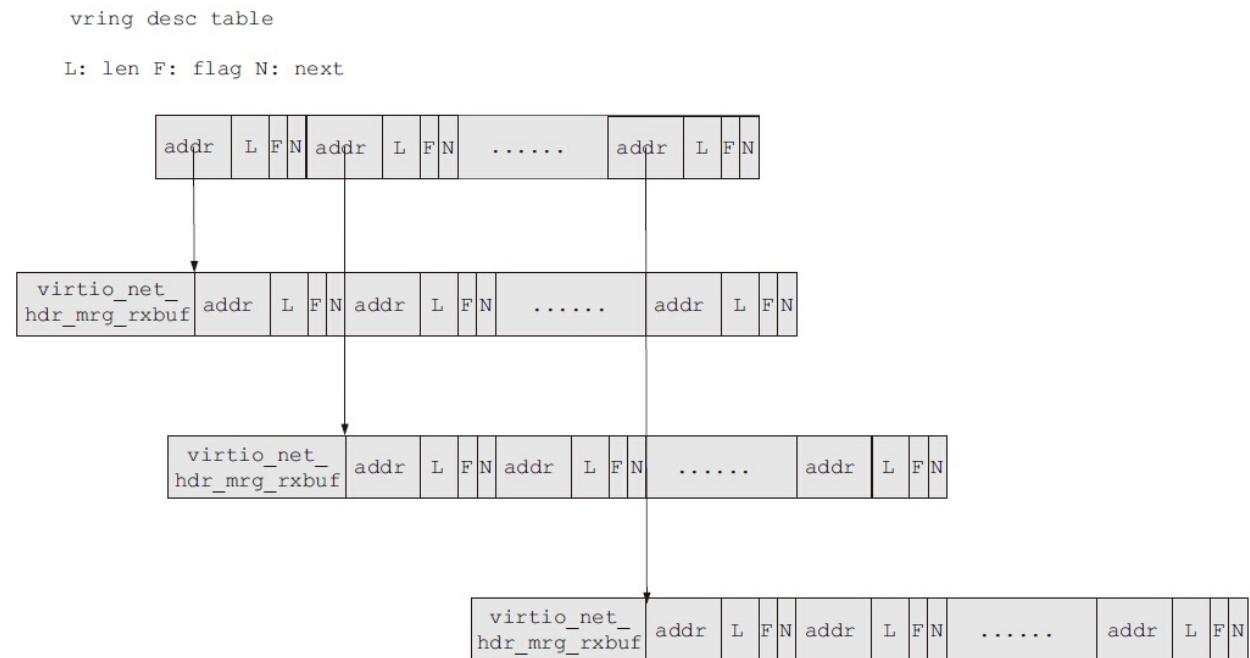
drivers/net/virtio\_net.c



## 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（相关的介绍见 PCIe）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。

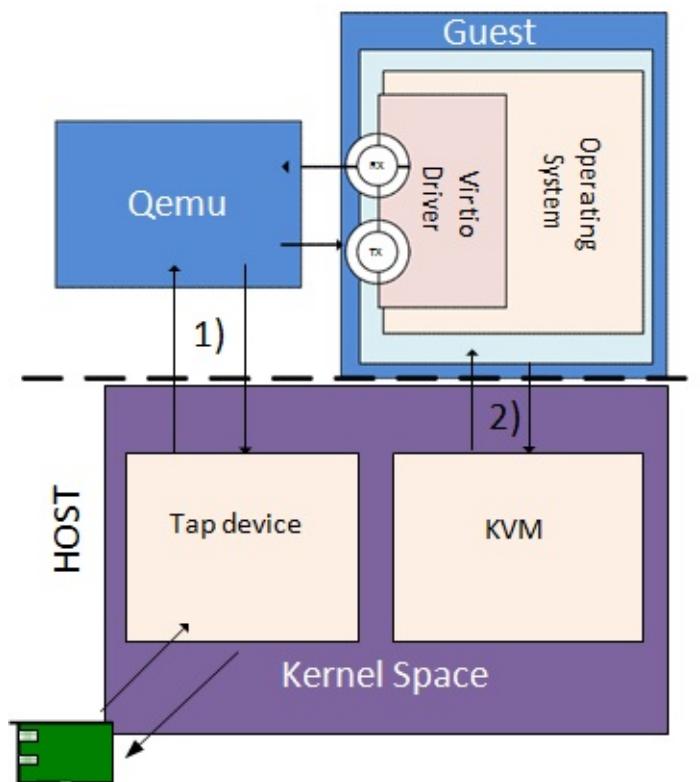


## vhost

`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。



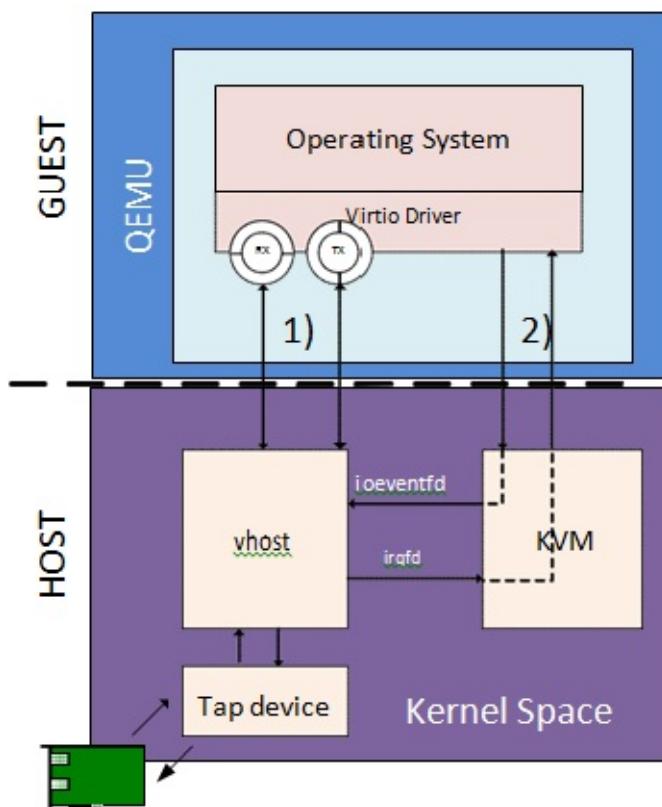
图片来源 [Vhost Sample Application](#)

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。



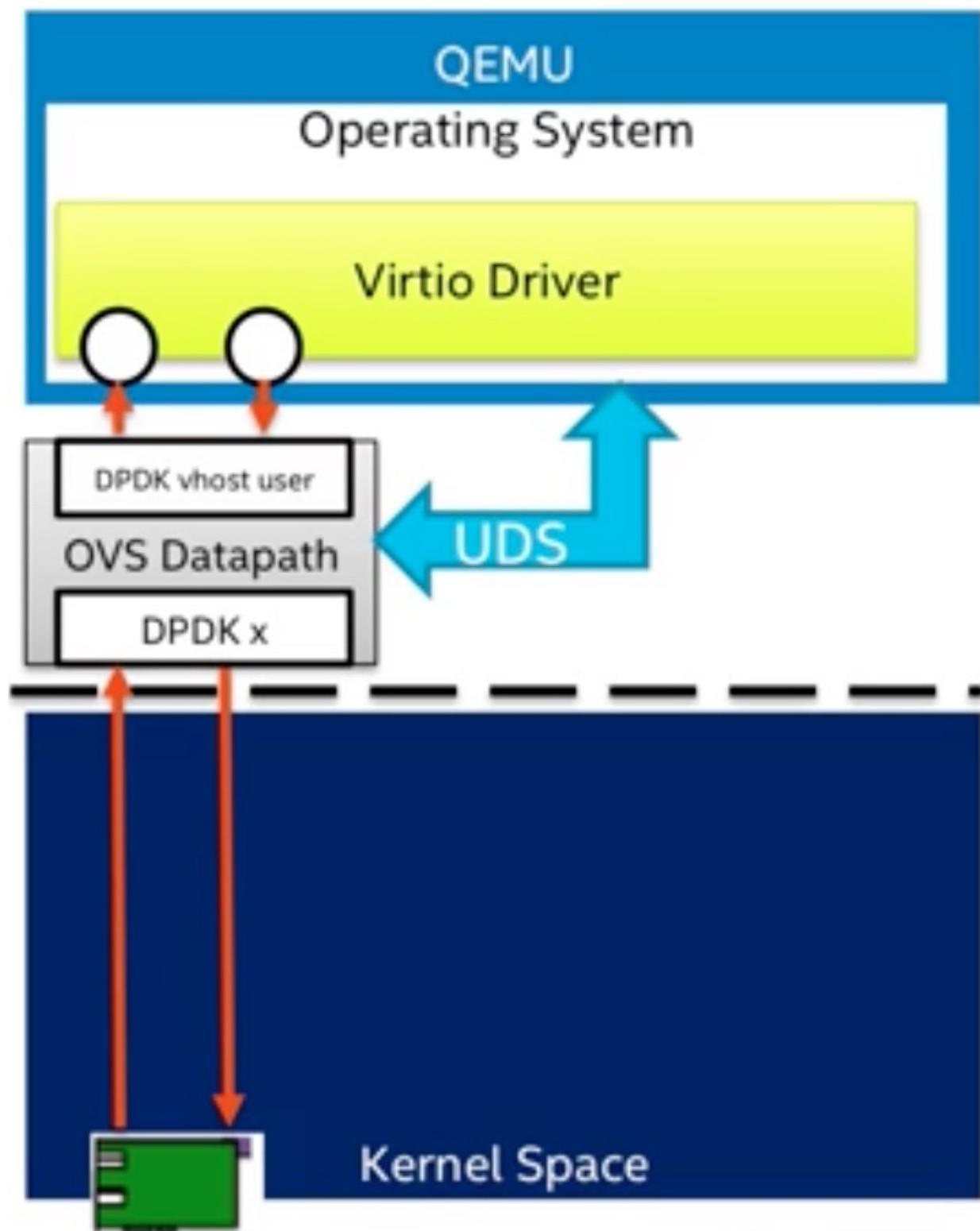
报文接收仍然包括数据通路和消息通知路径两个方面：

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>
<b>Performance</b>	<b>Performance</b>
▪ Less copies and context switches	▪ Bulk operations
<b>Security</b>	▪ Vectorisation
▪ Virtqueues mapped to vswitchd address space	<b>Features</b>
<b>Live Migration</b>	▪ Virtio-net backend enhancements
▪ Solution exists	<b>QEMU</b>
<b>Compatibility</b>	▪ Multi-queue
▪ Virtio-net standard	

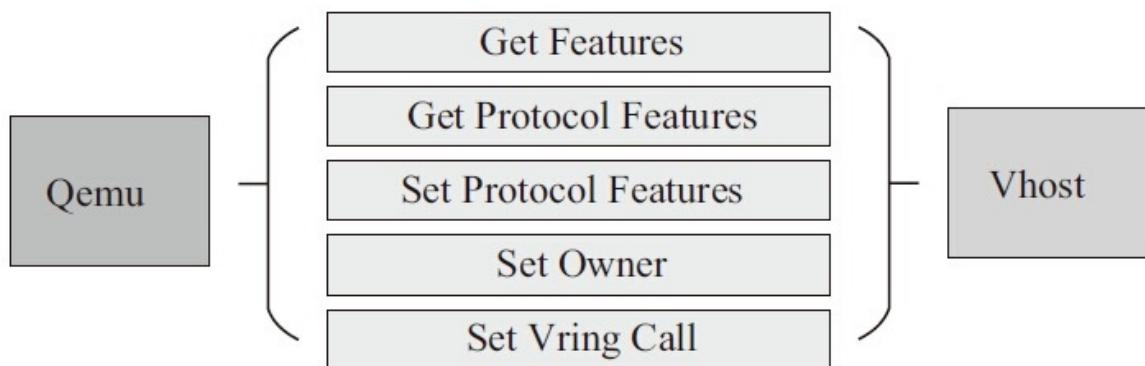
DPDK vhost 同时支持 Linux virtio-net 驱动和 DPDK virtio PMD 驱动的前端。

## DPDK vhost

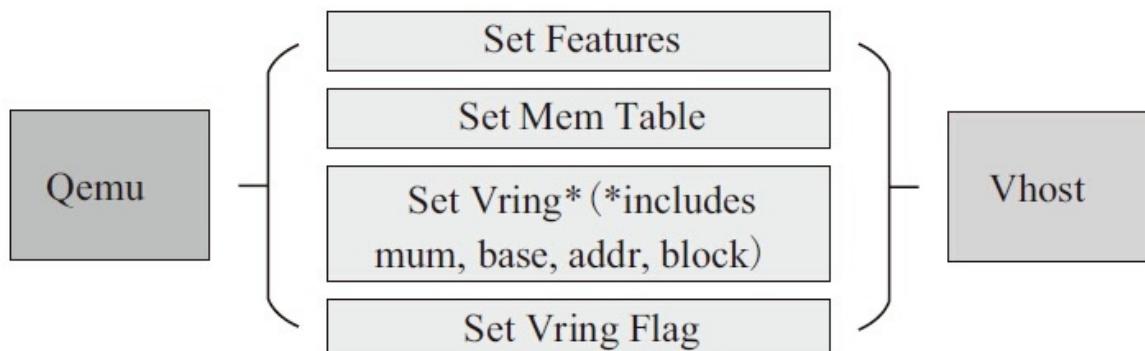
DPDK vhost 支持 vhost-cuse（用户态字符设备）和 vhost-user（用户态 socket 服务）两种消息机制，它负责为客户机中的 virtio-net 创建、管理和销毁 vhost 设备。

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。

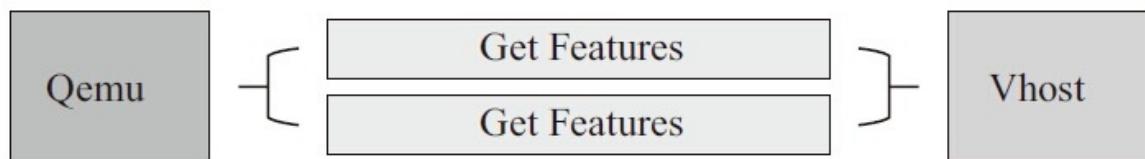
Vhost dev init:



Vhost dev start:



Vhost dev stop:

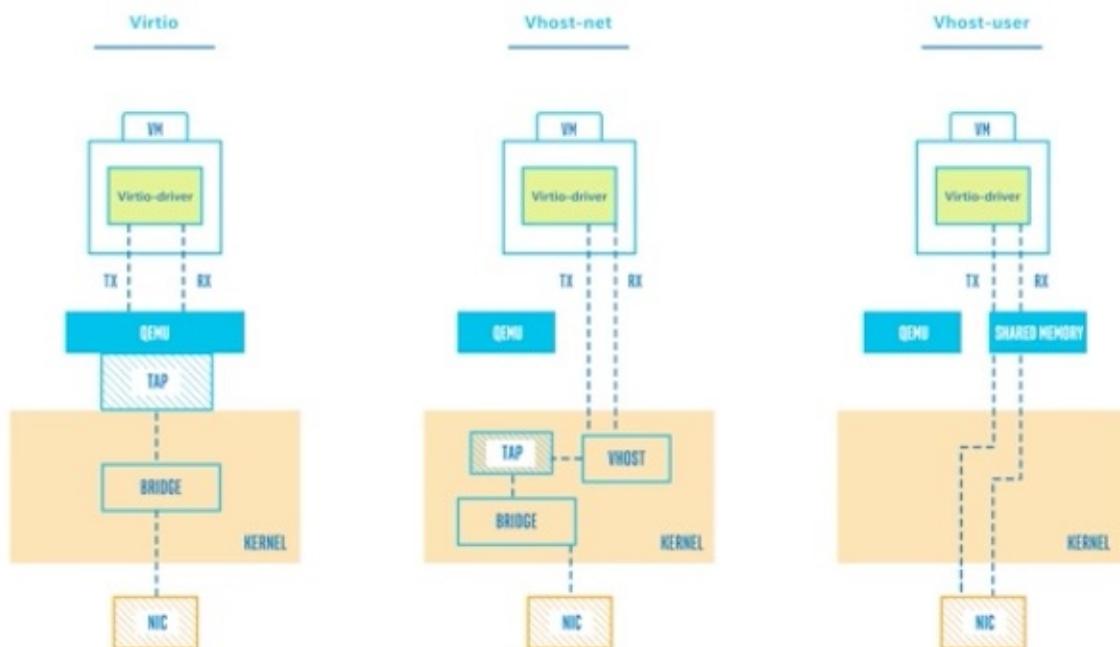


DPDK 的 vhost 有两种封装形式：`vhost lib` 和 `vhost PMD`。`vhost lib` 实现了用户态的 vhost 驱动供 vhost 应用程序调用，而 `vhost PMD` 则对 `vhost lib` 进行了封装，将其抽象成一个虚拟端口，可以使用标准端口的接口来进行管理和报文收发。

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



## 参考

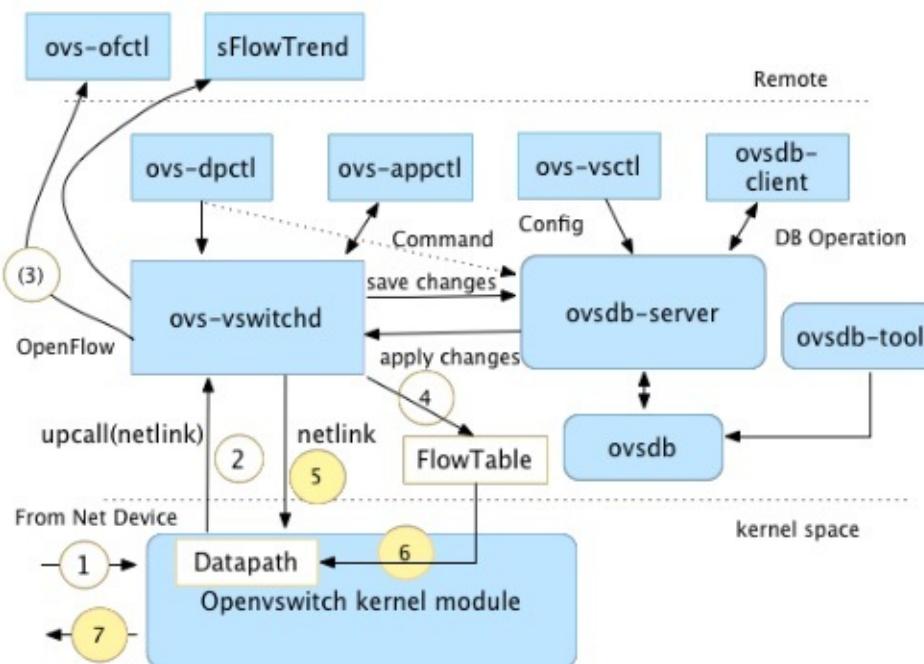
- What is I/O Virtualization (IOV)?
- IO virtualization
- XPDS14 - Intel(r) Virtualization Technology for Directed I/O (VT-d) Posted Interrupts - Feng Wu, Intel
- Virtio: An I/O virtualization framework for Linux

- Understanding Virtio Usage - Juniper
- Single Root I/O Virtualization (SR-IOV) – Part 1
- I/O Virtualization Overview: CNA, SR-IOV, VN-Tag and VEPA
- SR-IOV Component Architecture and Interaction
- Vhost Sample Application
- KVM I/O虚拟化分析

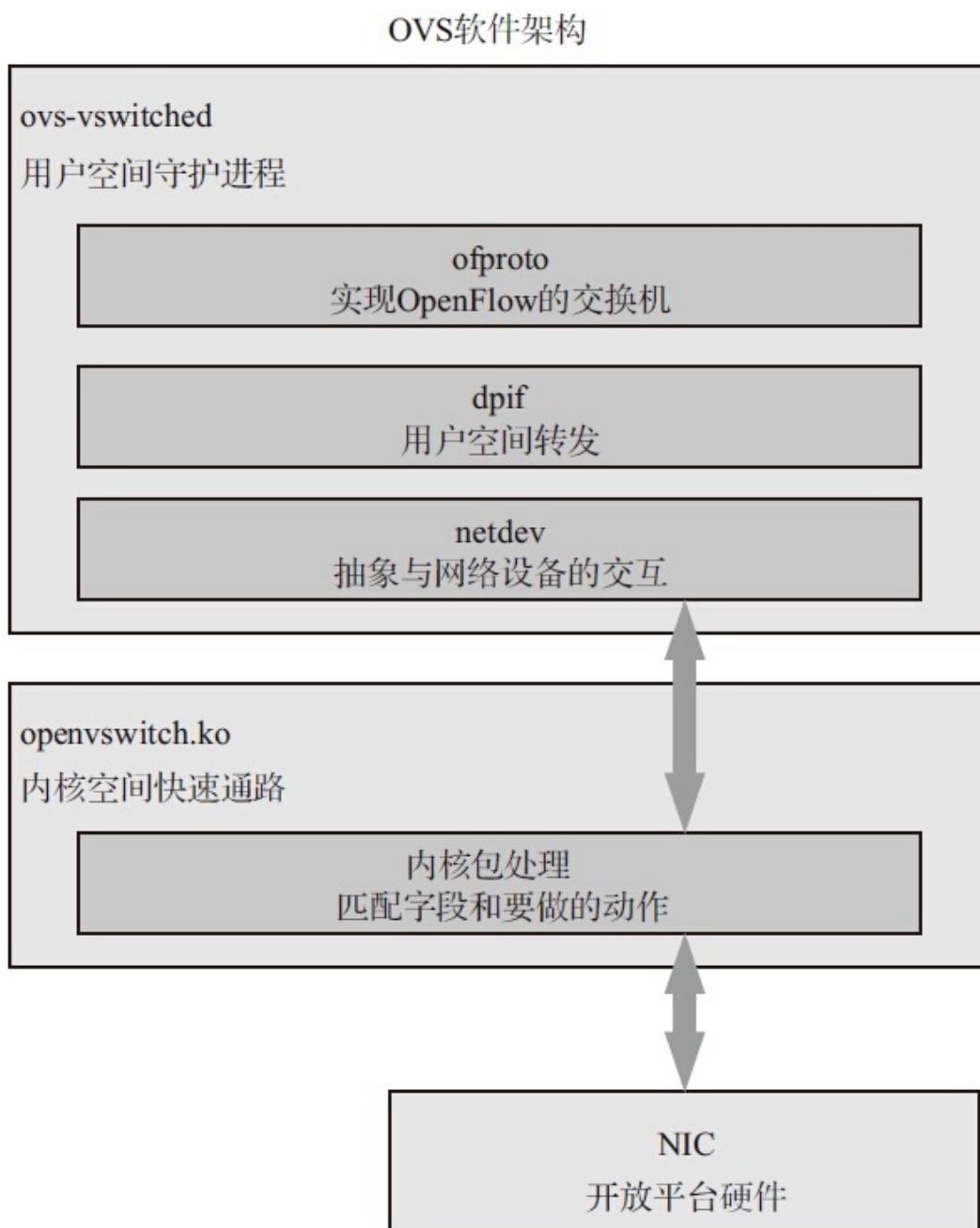
# OVS DPDK

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

## OpenvSwitch Internals

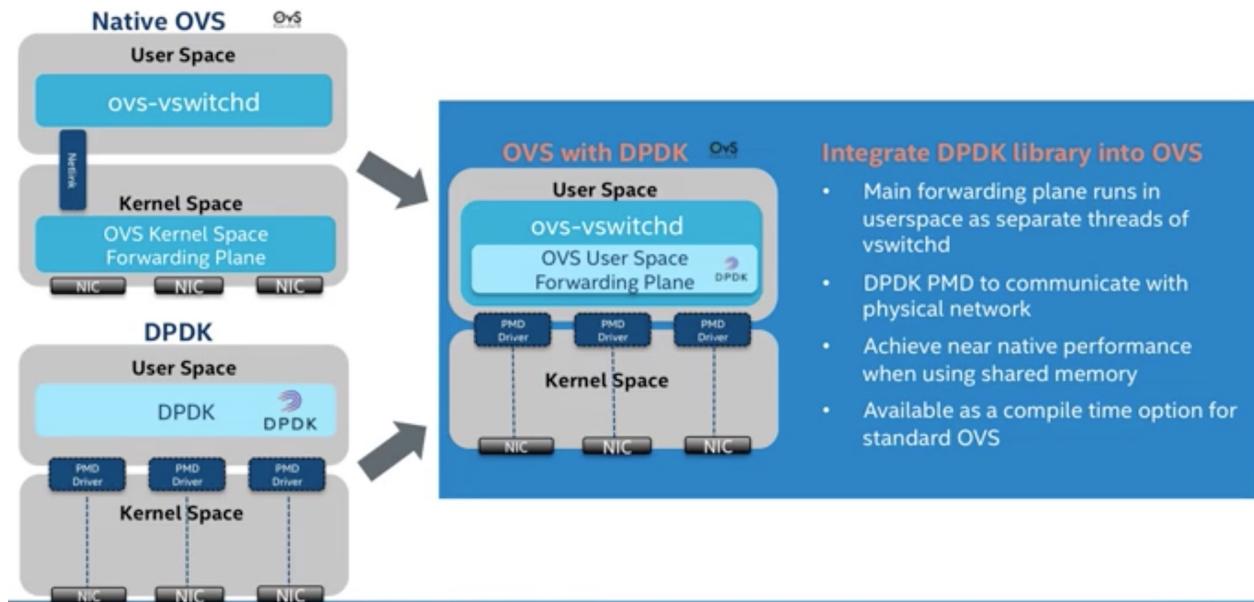


下图显示了 ovs 数据通路的内部模块图

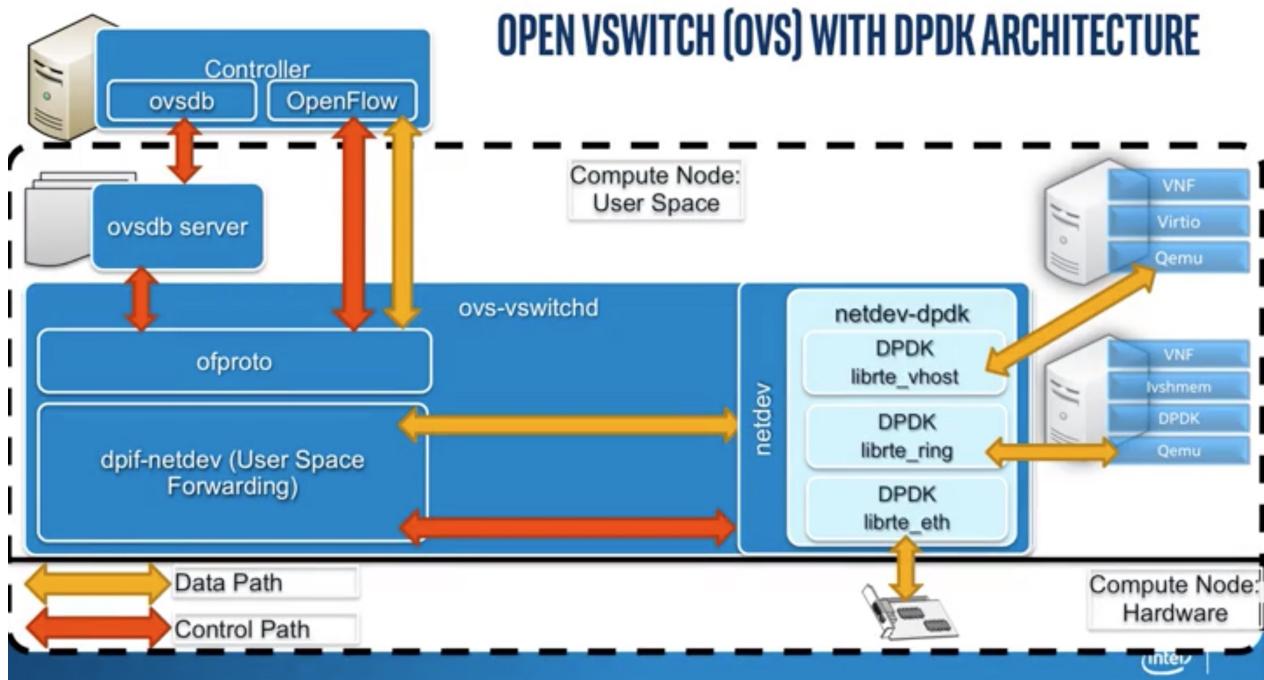


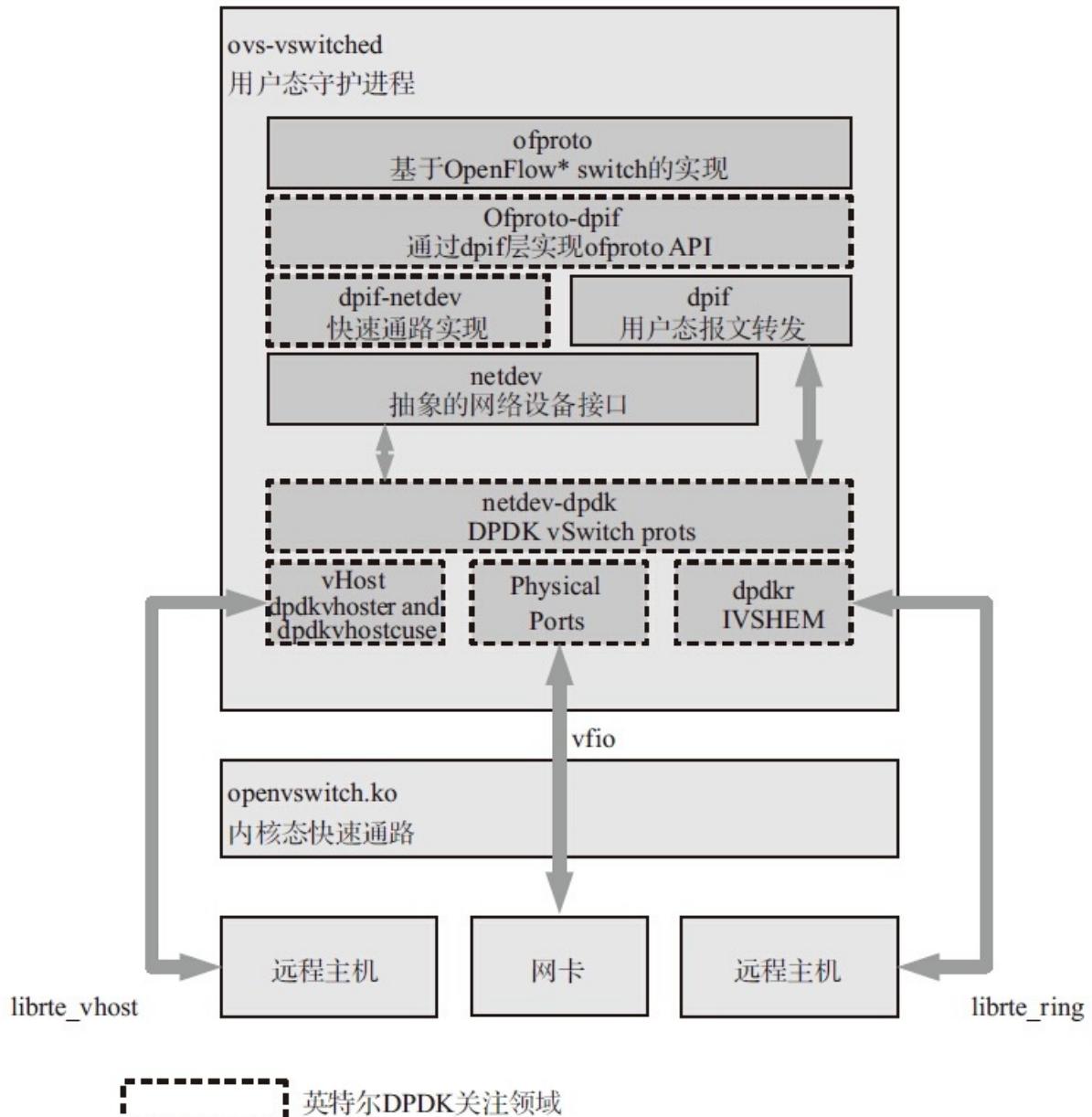
- `ovs-vswitchd` 主要包含 `ofproto`、`dpif`、`netdev` 模块
  - `ofproto` 模块实现 `openflow` 的交换机
  - `dpif` 模块抽象一个单转发路径
  - `netdev` 模块抽象网络接口（无论物理的还是虚拟的）
- `openvswitch.ko` 主要由数据通路模块组成，里面包含着流表。流表中的每个表项由一些匹配字段和要做的动作组成。

DPDK 加速的思想就是专注在这个数据通路上



## ovs with dpdk





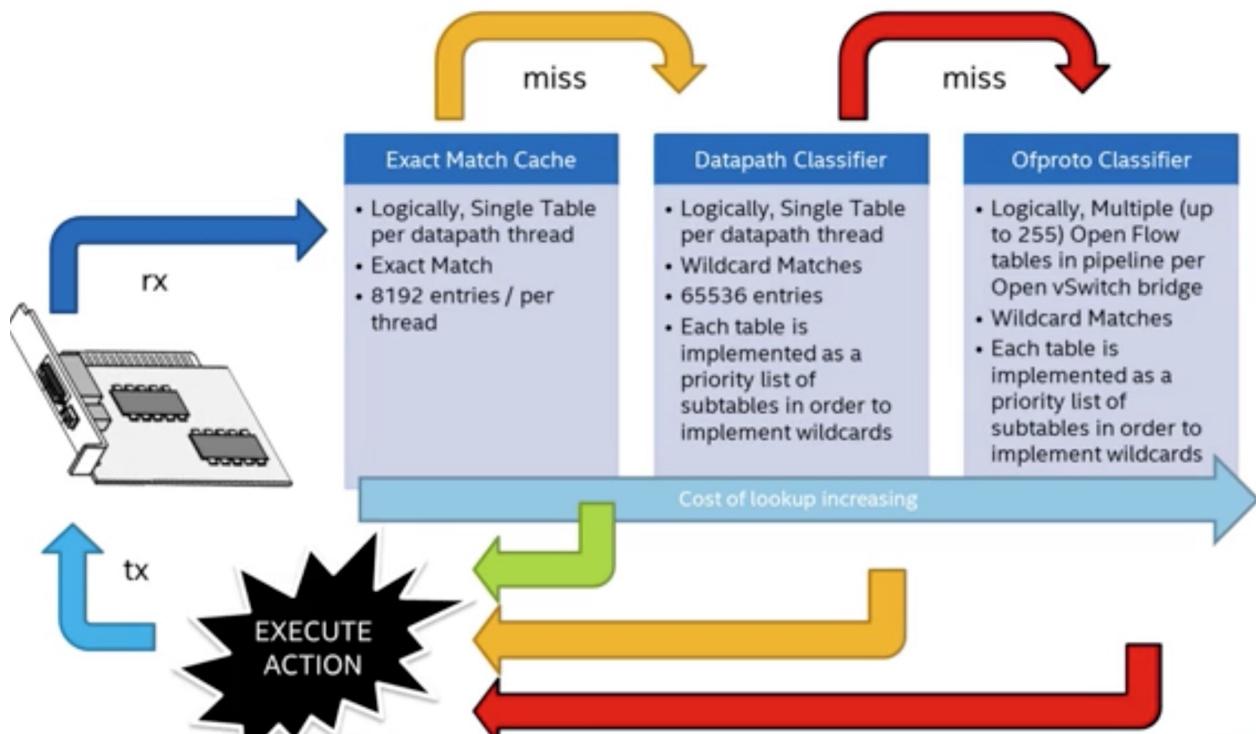
- `dpif-netdev` : 用户态的快速通路，实现了基于 `netdev` 设备的 `dpif API`。
- `Ofproto-dpif` : 实现了基于 `dpif` 层的 `ofproto API`。
- `netdev-dpdk` : 实现了基于 `DPDK` 的 `netdev API`，其定义的几种网络接口如下：
- `dpdk 物理网口`：其实现是采用高性能向量化 `DPDK PMD` 的驱动。
- `dpdkvhostuser` 与 `dpdkvhostcuse` 接口：支持两种 `DPDK` 实现的 `vhost` 优化接口：`vhost-user` 和 `vhost-cuse`。`vhost-user` 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：

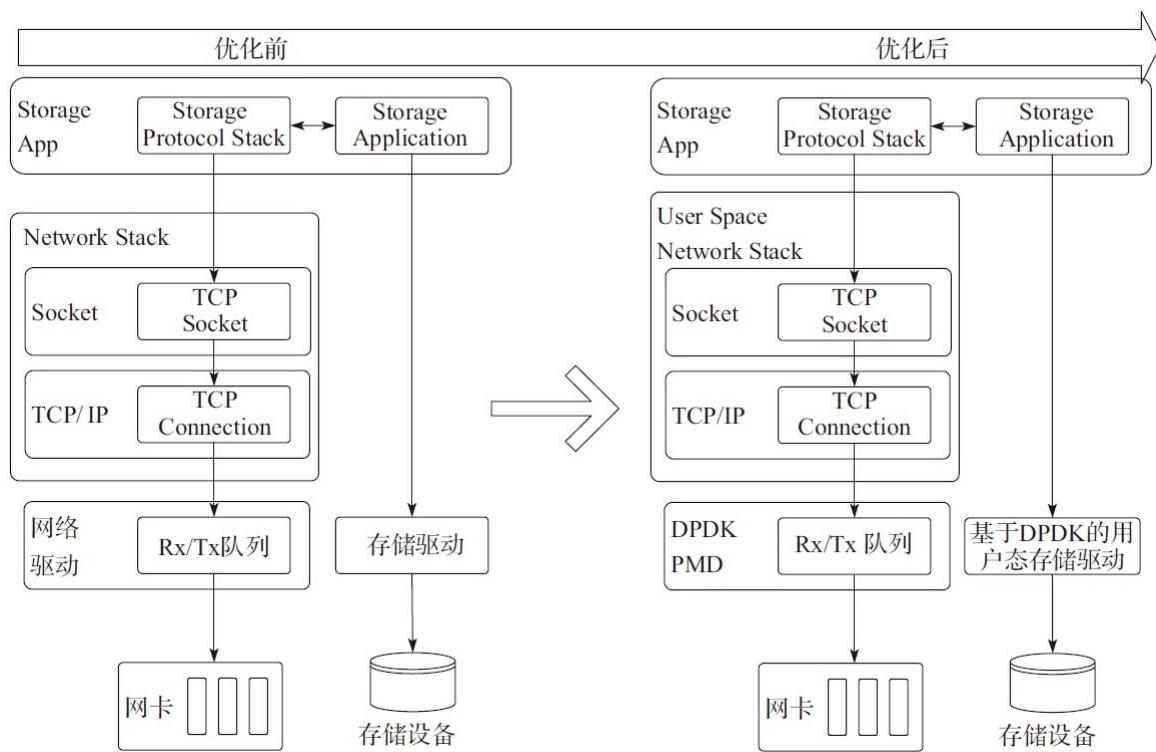
1. OVS 的 ovs-vswitchd 接收到从 OVS 连接的某个网络端口发来的数据包，从数据包中提取源/目的 IP、源/目的 MAC、端口等信息。
2. OVS 在用户态查看精确流表和模糊流表，如果命中，则直接转发。
3. 如果还不命中，在 SDN 控制器接入的情况下，经过 OpenFlow 协议，通告给控制器，由控制器处理。
4. 控制器下发新的流表，该数据包重新发起选路，匹配；报文转发，结束。

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

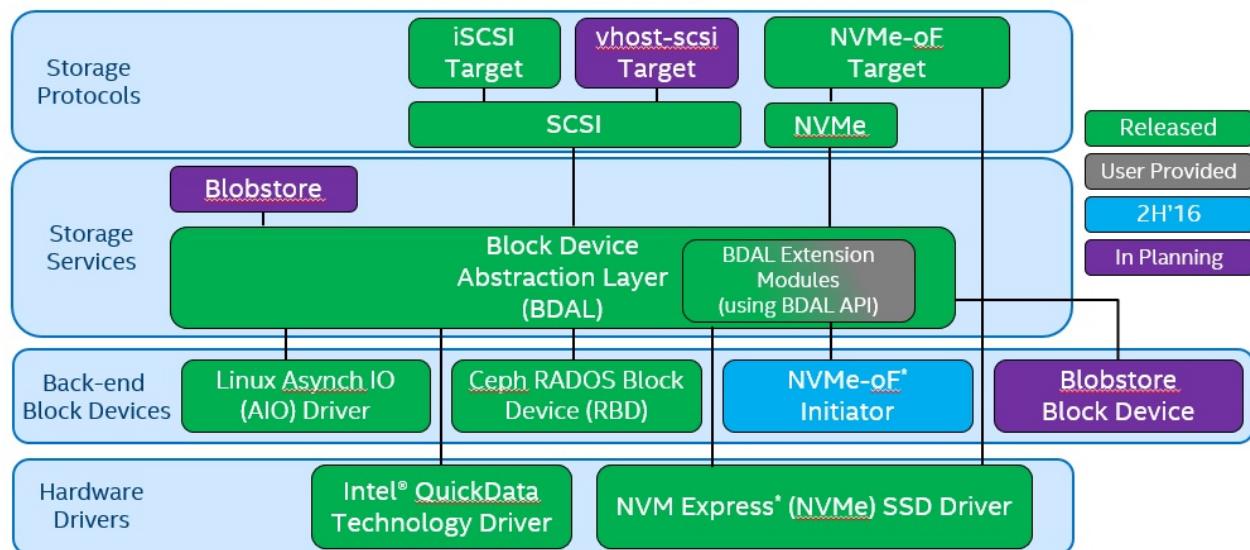


# SPDK

## 简介

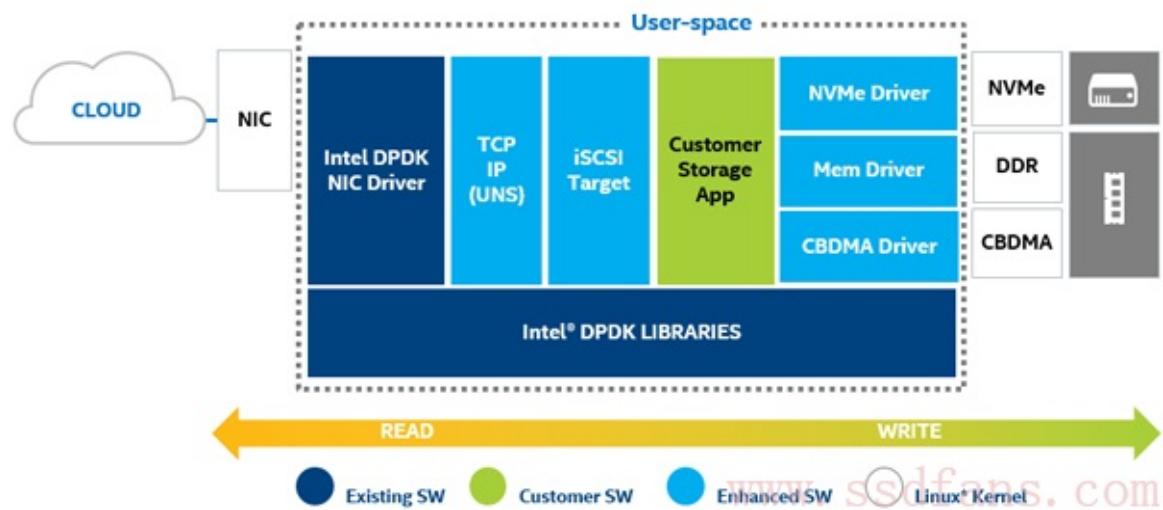
SPDK 全称 Storage Performance Development Kit (高性能存储开发包)，它把驱动程序尽可能都放到用户态，同时采用了轮询模式，这样消除了 Kernel 进程之间的切换和中断处理，用这种方法达到高性能。

### Storage Performance Development Kit (SPDK)

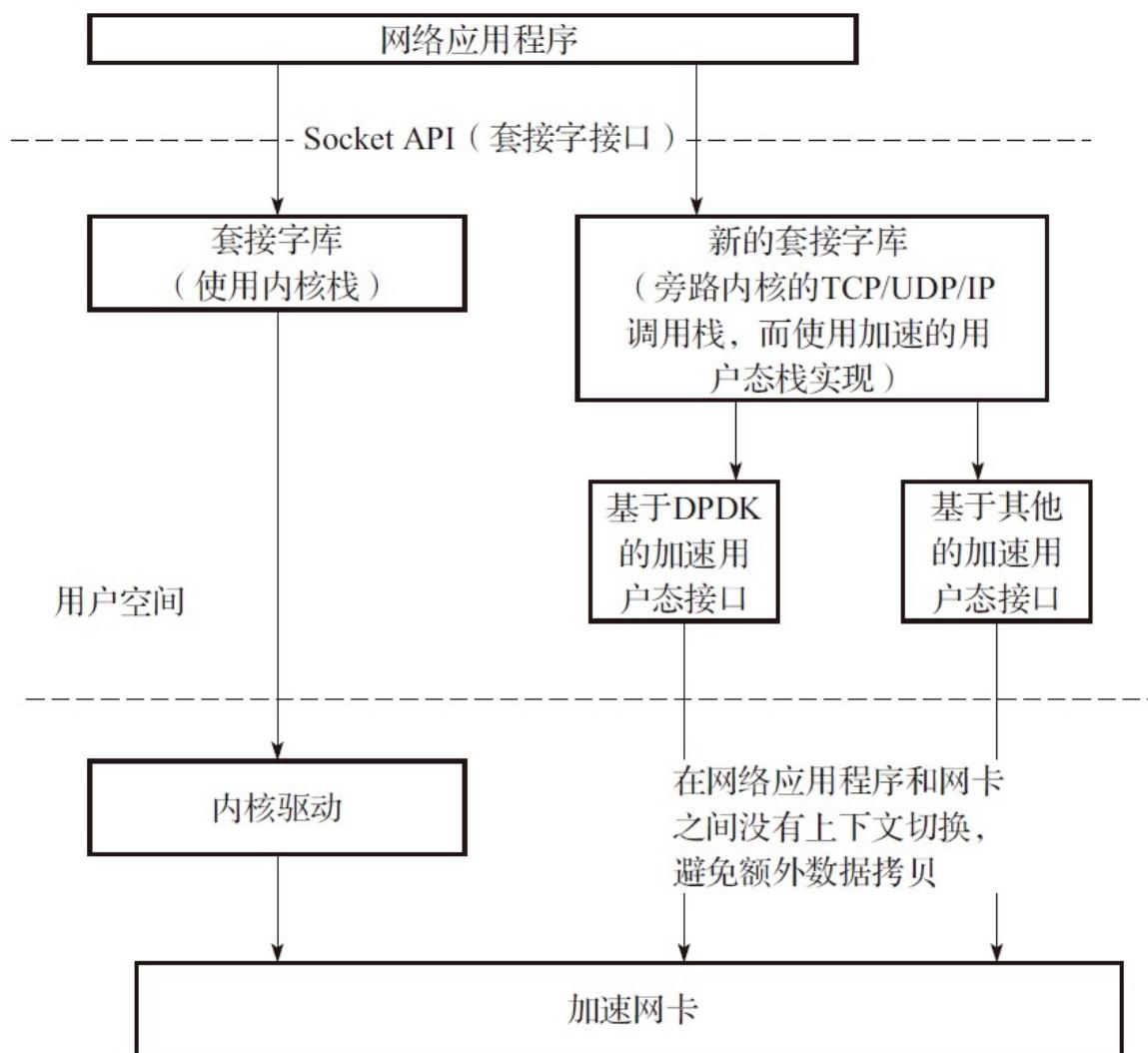


图片来源 [Introduction to the Storage Performance Development Kit \(SPDK\)](#)

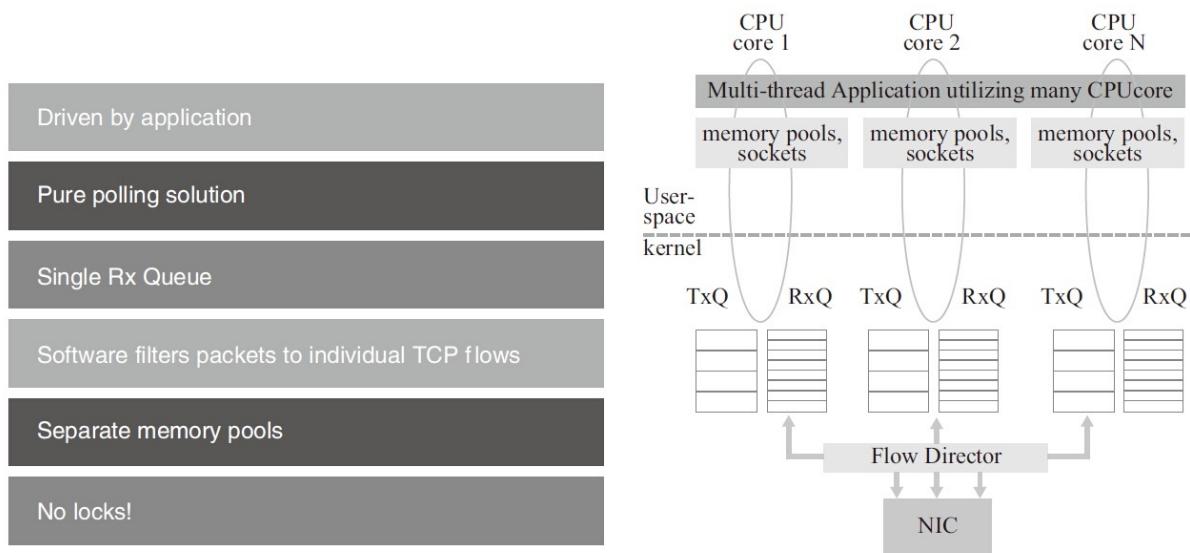
为了帮助存储 OEM（设备代工厂）和 ISV（独立软件开发商）整合硬件，Intel 构造了一系列驱动，以及一个完善的、端对端的参考存储体系结构，被命名为 Storage Performance Development Kit (SPDK)。SPDK 的目标是通过同时使用 Intel 的网络技术，处理技术和存储技术来提高突出显著的效率和性能。通过运行为硬件设计的软件，SPDK 已经证明很容易达到每秒钟数百万次 I/O 读取，通过使用许多处理器核心和许多 NVMe 驱动去存储，而不需要额外卸载硬件。Intel 在 [BSD license](#) 许可协议下通过 [Github](#) 分发提供其全部的 [Linux](#) 参考架构的源代码。博客、邮件列表和额外文档可以在 [spdk.io](#) 中找到。



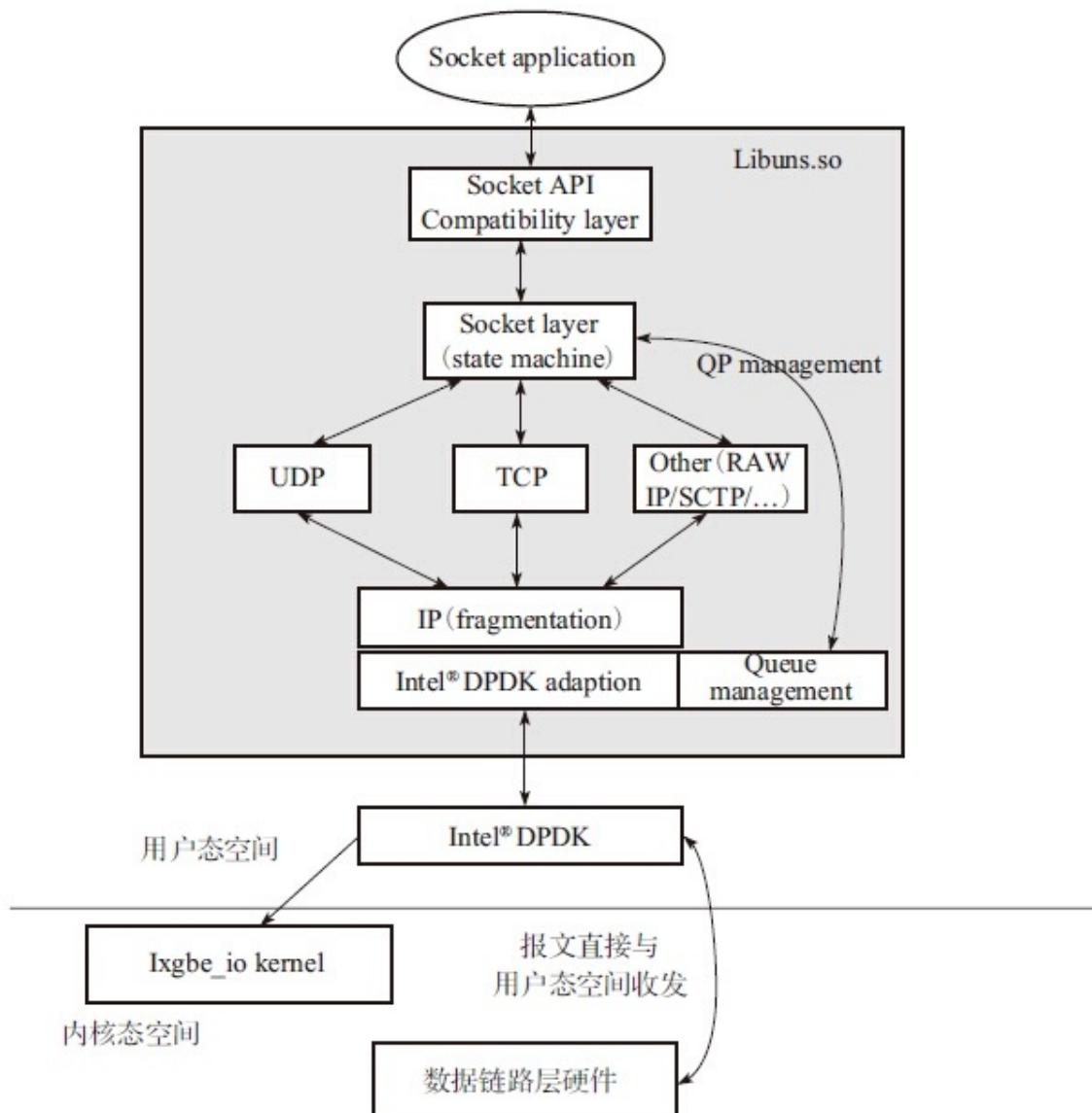
图片来源NVMeDirect：超越SPDK



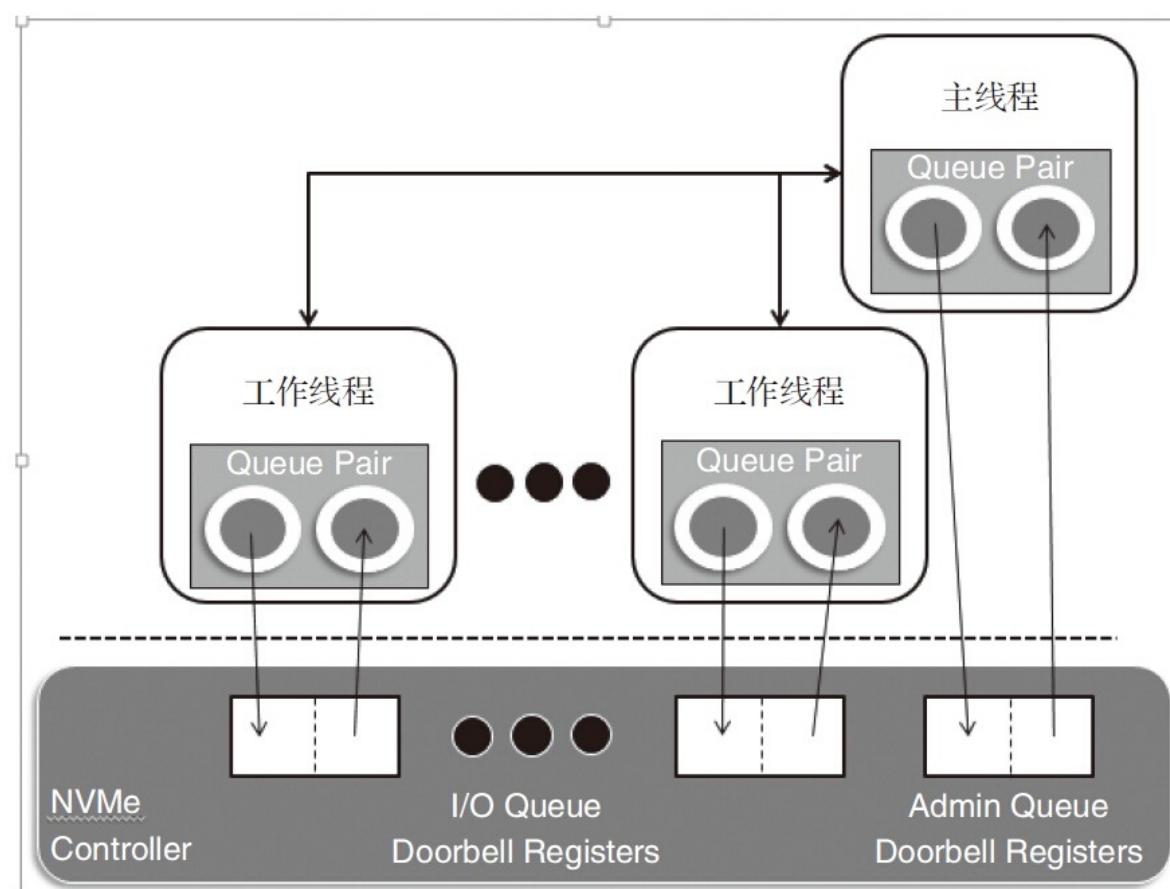
## Libuns 系统架构



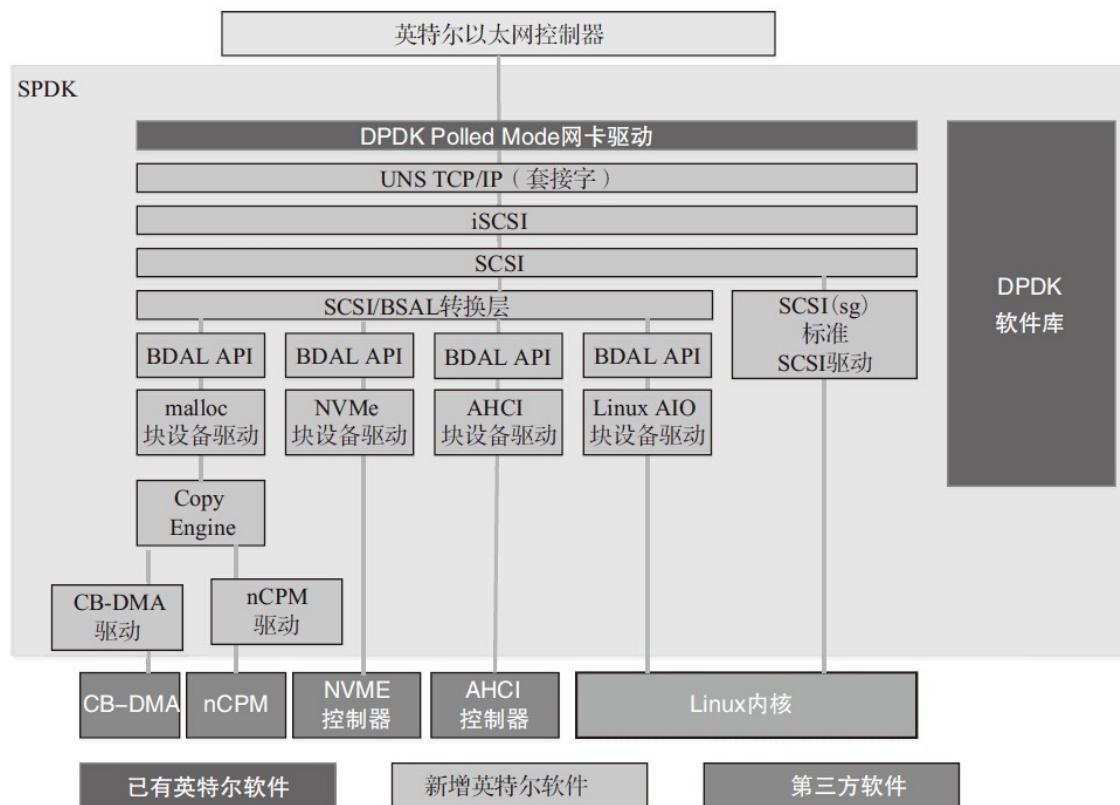
### Libuns的TCP/UDP/IP协议栈



## 用户态 NVMe



## iSCSI target



## 参考

- [Introduction to the Storage Performance Development Kit \(SPDK\)](#)
- [SPDK简介](#)
- [NVMeDirect : 超越SPDK](#)

# OpenFastPath

OpenFastPath 项目（即OFP）主页位于：

- <https://openfastpath.org/>
- <https://github.com/OpenFastPath/ofp/>

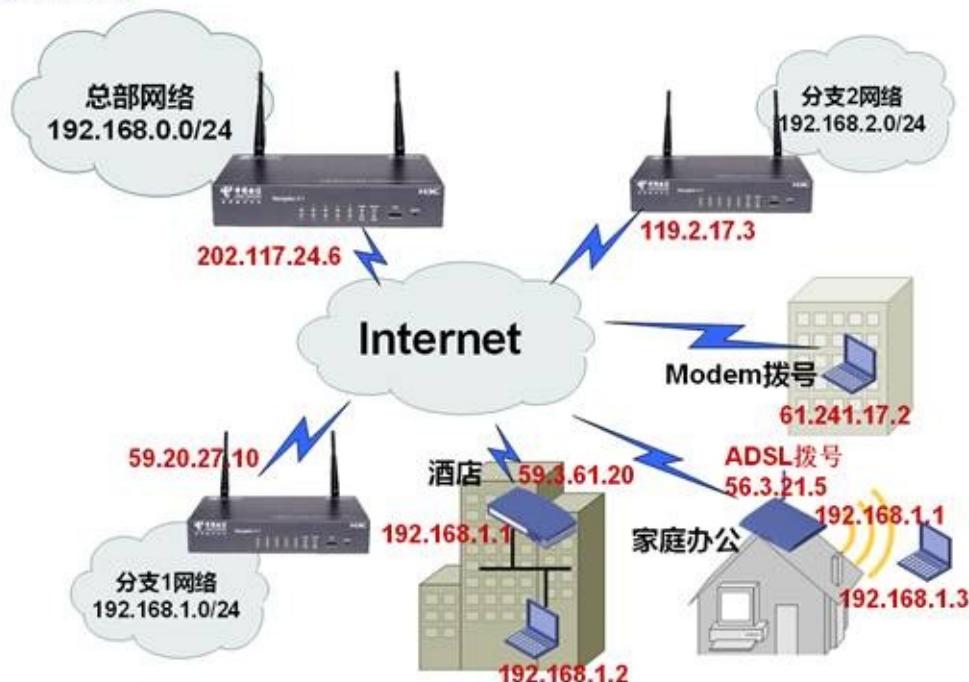
## 安全设备

待补充

# VPN(Virtual Private Network)概述

## 为什么需要VPN

### 远程访问



很多企业随着业务的发展，已经在异地建立分支机构，或者许多员工出差至外地开展工作，甚至需要回家继续办公，那么这些远程员工是否还能够连接到总部网络享受到统一的企业信息化管理呢？我想，大家必定会回答可以，既然可以，是通过什么技术实现的呢？或者说实现的时候要考虑哪些因素呢？从上面的图中我们可以看到似乎有很多种异地网络用户，布置这种技术似乎很困难，我们先把图中网络单元可以分为3类：

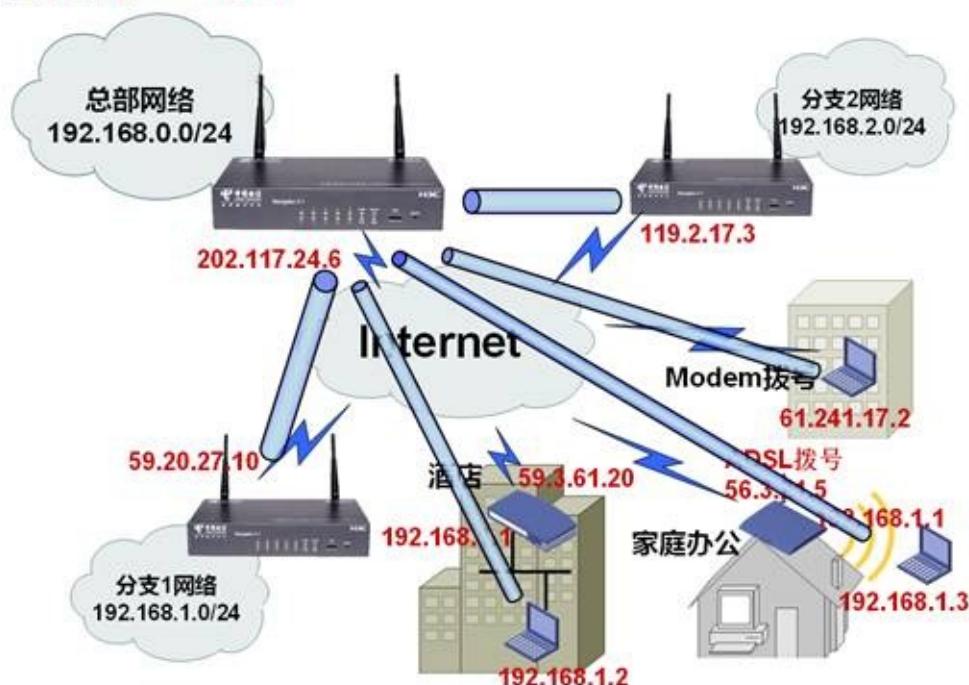
1. 总部机构，总部在连接互联网绝大多数情况下使用固定出口以及固定地址，在一些极端情况下可能会采用动态地址方式。
2. 远程分支机构，这类网络有固定的网络出口连接到互联网，互联网出口设备以及内部网络都是完全受企业管理的，在图中，分支1和分支2都是这类，虽然网络出口是固定的，但是出口地址是否固定和接入方式有关，比如租用光纤那么通常会从运营商获得一个固定地址，如果是ADSL则是动态的，远程分支机构的典型特征是以一个网络作为远程接入单位。
3. 出差员工，这类网络用户的特点是以用户PC为远程网络单元，为什么呢，因为这类用户的互联网出口通常不受企业管理，比如出差员工在酒店通过酒店网络接入互联网、员工在家上网、员工在酒店直接拨号到互联网等，这类用户的特征是以单台PC作为远程接入单位。

这些异地网络用户访问总部网络，可能大家会认为很简单，直接访问总部网络的网段就可以了。实际上，企业网络通常使用私有地址，是无法从互联网直接访问的，那么我们可以通过什么手段访问这些总部的私有网段吗？

1. 使用专线，即每个异地网络用户单元使用一条专线连接到总部，那么在上图中，我们至少需要5条专线，如果出差员工或者分支多起来需要更多专线，每条专线都价值不菲，而且专线只能连接总部，无法访问互联网，显然这种方式对于非常注重成本的企业而言是不可接受的。
2. 既然总部和各个异地网络都连接到了互联网，能不能通过互联网把大家连起来呢？我们可以看到各个网络单元的出口都是互联网公有地址，让这些地址互相访问不成问题，那么能不能把访问内部私有网络的连接建立在这些共有连接上呢？答案当然可以，这就是今天要讲的内容——虚拟私有网（Virtual Private Network），即在公共网络上建立虚拟的隧道，模拟成专线。

## VPN应用场景和隧道特点

### 远程访问——VPN



那么如何建立这些隧道呢？要建立什么样的隧道？我们可以通过这张表来描述异地网络用户和总部网络出口隧道的特点。

	隧道端点	隧道内流量	隧道发起	安全性需求	是否穿越NAT
异地分支	分支出口<->总部出口	分支内网<->总部内网	总部和分支都可以发起	少量身份认证和加密	不需要
酒店、家庭办公	异地PC<->总部出口	异地PC<->总部内网	只能从PC发起	大量动态身份认证和加密	需要
异地分支	异地PC<->总部出口	异地PC<->总部内网	只能从PC发起	大量动态身份认证和加密	不需要

## VPN基本功能

VPN属于远程访问技术，简单地说就是利用公用网络架设专用网络。例如某公司员工出差到外地，他想访问企业内网的服务器资源，这种访问就属于远程访问。

在传统的企业网络配置中，要进行远程访问，传统的方法是租用DDN（数字数据网）专线或帧中继，这样的通讯方案必然导致高昂的网络通讯和维护费用。对于移动用户（移动办公人员）与远端个人用户而言，一般会通过拨号线路（Internet）进入企业的局域网，但这样必然带来安全上的隐患。

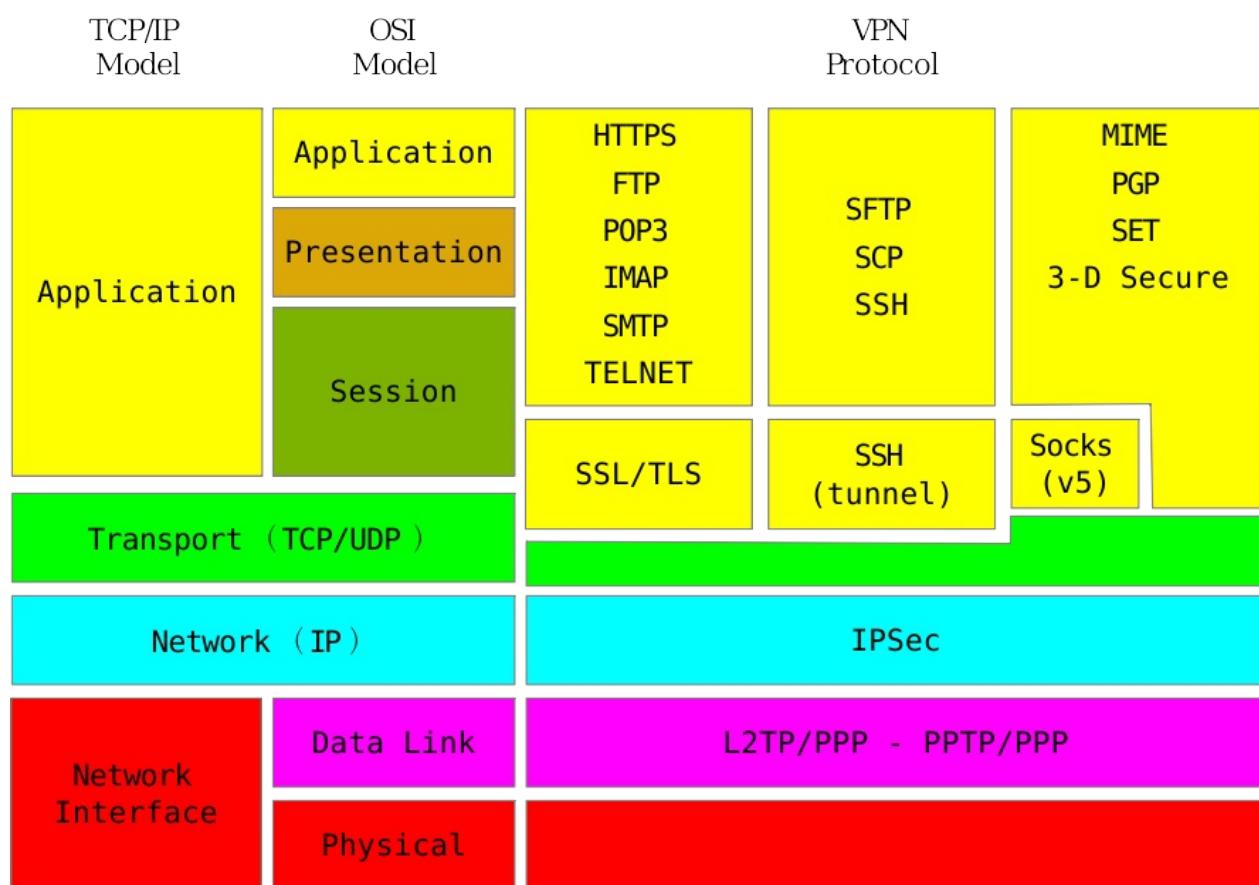
让外地员工访问到内网资源，利用VPN的解决方法就是在内网中架设一台VPN服务器。外地员工在当地连上互联网后，通过互联网连接VPN服务器，然后通过VPN服务器进入企业内网。为了保证数据安全，VPN服务器和客户机之间的通讯数据都进行了加密处理。有了数据加密，就可以认为数据是在一条专用的数据链路上进行安全传输，就如同专门架设了一个专用网络一样，但实际上VPN使用的是互联网上的公用链路，因此VPN称为虚拟专用网络，其实质上就是利用加密技术在公网上封装出一个数据通讯隧道。有了VPN技术，用户无论是在外地出差还是在家中办公，只要能上互联网就能利用VPN访问内网资源，这就是VPN在企业中应用得如此广泛的原因。

## VPN分类标准

根据不同的划分标准，VPN可以按几个标准进行分类划分：

### 1. 按VPN的协议分类：

VPN的隧道协议主要有四种，PPTP、L2TP、IPSec和SSL，其中 PPTP 和 L2TP 协议工作在 OSI模型的第二层，又称为二层隧道协议；IPSec 是第三层隧道协议；而 SSL 是工作在OSI会话层之上的协议，如果按照TCP/IP协议模型划分，即工作在应用层。



## 2. 按VPN的应用分类：

1. Access VPN (远程接入VPN)：客户端到网关，使用公网作为骨干网在设备之间传输VPN数据流量；
2. Intranet VPN (内联网VPN)：网关到网关，通过公司的网络架构连接来自同公司的资源；
3. Extranet VPN (外联网VPN)：与合作伙伴企业网构成Extranet，将一个公司与另一个公司的资源进行连接。

## 3. 按所用的设备类型进行分类：

网络设备提供商针对不同客户的需求，开发出不同的VPN网络设备，主要为交换机、路由器和防火墙：

1. 路由器式VPN：路由器式VPN部署较容易，只要在路由器上添加VPN服务即可；
2. 交换机式VPN：主要应用于连接用户较少的VPN网络；
3. 防火墙式VPN：防火墙式VPN是最常见的一种VPN的实现方式，许多厂商都提供这种配置类型

## 4. 按照实现原理划分：

1. 重叠VPN：此VPN需要用户自己建立端节点之间的VPN链路，主要包括：GRE、L2TP、IPSec等众多技术。
2. 对等VPN：由网络运营商在主干网上完成VPN通道的建立，主要包括MPLS、VPN技术。

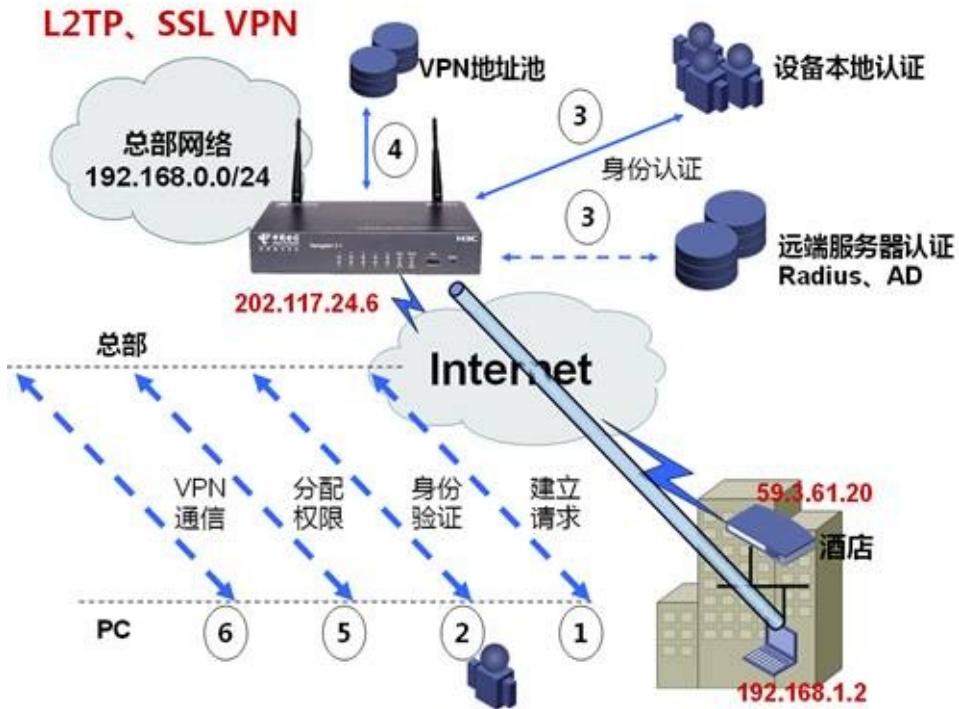
## VPN技术介绍

我们可以从上表中发现异地分支和总部、异地PC和总部之间隧道有较多不同，可以把VPN划分为这两类场景进行研究，有什么样的需求就有什么样的技术：

1. PPTP（点到点隧道协议）、L2TP（二层隧道协议）、SSL VPN（安全会话层VPN）：  
这两个技术主要用于异地PC向总部方向建立VPN，但PPTP、L2TP不支持加密（PPTP的兼容性没有L2TP好），SSL VPN则必须要求加密；这三类技术都有很灵活的动态身份认证机制，SSL VPN不但拥有灵活安全的认证机制，在用户角色权限控制上具备极强的扩展性，因此这3类技术非常适合远程PC拨号接入场景，随着SSL VPN技术成熟，部署成本下降，正在不断地侵占L2TP原有市场。
2. IPSec：通用性最强的VPN安全技术，能够适应异地分支和总部互联，也适用于异地PC向总部发起连接；可以单独使用，也可以和L2TP结合，保证L2TP的安全，但是IPSec的动态身份认证功能较弱，不太适用于大量动态用户拨号的场景，比较适合接入数量相对稳定的场景，此外IPSec功能复杂，PC上通常是通过各厂家专用客户端实现，因此IPSec技术更适合异地分支和总部网络互连的场景。
3. SSL VPN是以HTTPS（Secure HTTP，安全的HTTP，即支持SSL的HTTP协议）为基础的VPN技术，工作在传输层和应用层之间。**SSL VPN**充分利用了**SSL**协议提供的基于证书的身份认证、数据加密和消息完整性验证机制，可以为应用层之间的通信建立安全连接。**SSL VPN**广泛应用于基于Web的远程安全接入，为用户远程访问公司内部网络提供了安全保证。

下面对集中VPN技术和应用场景进行详细介绍。由于PPTP功能和L2TP重叠，且应用较窄，在此文中不作介绍。

## L2TP和SSLVPN

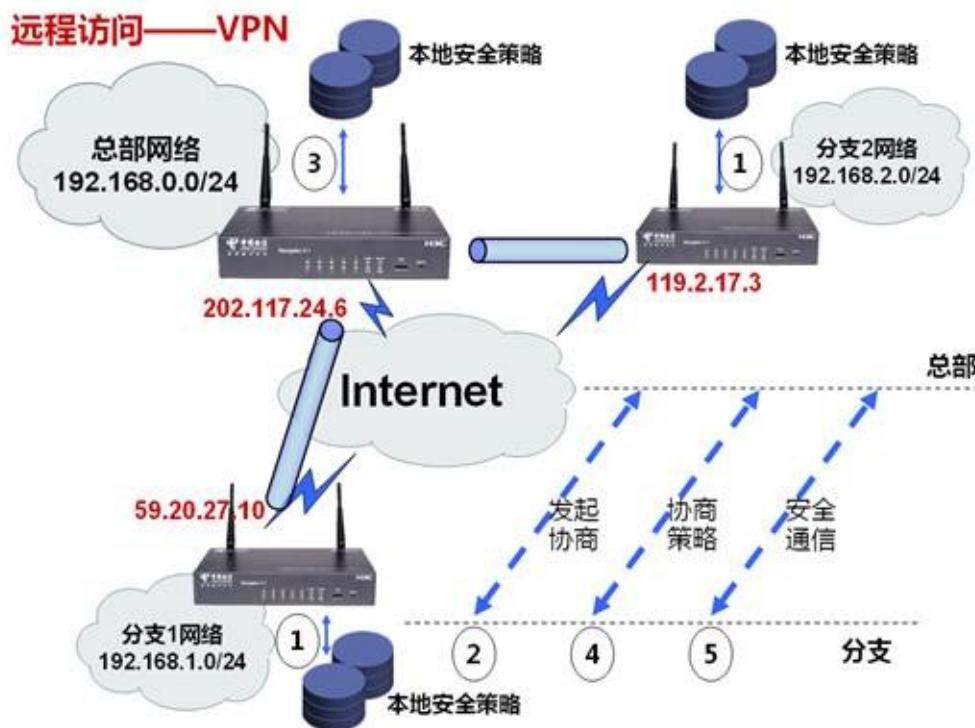


该类型的VPN可以分为如下几个阶段：

1. 分支向总部发出连接请求，要就建立VPN，如果是SSL VPN，该阶段还需要协商密钥，为后续所有通信进行加密保护，而L2TP则没有专用保护手段，除非借助IPSec的帮助。
2. 对接入请求进行身份验证，因为企业的VPN资源只允许对本企业员工开放，所以必须对接入者身份进行验证，身份验证通常分为身份确认和口令验证两部分组成，身份表明用户的角色，而口令则是进一步确认角色的准确性。
3. 身份验证可以有两种方式实现，一是本地认证，二是专用服务器认证，通常专用服务器认证能够有更好的扩展性和性能。认证的结果称之为授权，即授予一定的功能，授权内容由总部出口控制。
4. 授权中一个很重要内容是为远端PC的隧道接口分配一个属于企业内部的私有地址，表示该远端PC已经接入到企业内部网络了；地址授权依然可以由设备实现，也可以由认证服务器实现。为什么要分配地址呢？因为我们说了VPN是在互联网上模拟一条专线，物理专线需要IP地址，那么这个虚拟专线也需要IP地址。
5. 总部出口设备返回给远端PC的是认证和授权的报告，报告内容主要是通知接入者是否接入成功，如果认证失败则对PC提示接入失败，如果认证成功，则提示用户已经成功接入VPN。
6. VPN建立后，该远端PC就像是通过一个专用的线缆接入到企业出口设备上了，可以根据授权内容进行访问。

7. 这种类型的VPN连接和断开都是在外出员工在远端PC上主动操作，通常总部出口网关无法主动关闭VPN；为了避免总部出口长期没有收到拆除消息而维护大量VPN连接导致资源耗尽，VPN连接通常都是有计时机制的，如果总部出口设备长时间没有收到远端PC的任何VPN流量，那么总部出口会认为该PC已经下线，释放资源，这是很重要的安全考虑。

## IPSec VPN



在IPSec中，由于总部网络相对固定，而分支变动性较大，所以大部分情况下是也是由分支向总部发起连接：

1. IPsec协商并不需要像L2TP、SSL VPN那样需要每次人为主动去建立连接，IPsec需要通信双方提前做好配置，我们通常称这些配置为安全策略，主要内容包括远端地址、兴趣流、远端身份信息和预共享密钥（也可以采用数字证书）、阶段1安全提议、阶段2安全提议等，将本地安全策略制定好后，IPsec VPN就会根据兴趣流自动触发建立，因此特别适合分支网络和总部网络通信的场景。
2. 当分支检测到兴趣流后，就会根据安全策略配置向总部发起阶段1协商，在协商时需要同样需要进行验证，验证失败则退出协商，除了验证身份外还需要对阶段1安全参数进行协商，如果协商不出共同的安全参数，那么也是退出协商。
3. 总部通常处于被动响应模式，所以当需要协商时，也是从本地安全策略中找出匹配的参数。

4. 如果阶段1安全参数和验证成功，会进行2阶段的协商，阶段2协商的是兴趣流和阶段2安全参数，如果能够找到相匹配的参数，则协商成功，如果协商失败则有可能同时把阶段1协商结果删除。
5. 阶段2协商成功后，分支和总部就建立了安全隧道，可以正常的通信了。
6. IPSec默认周期性协商机制，如果下一周期协商失败，那么隧道拆除；此外还有对端激活检测，如果检测对端已经失效则自动拆除隧道。

## 不同VPN实现对比

	L2TP	SSL VPN	IPSec
发起方	远端PC	远端PC	总部、分支、远端PC
身份验证	基于PPP认证	用户名+口令+证书	地址或名字+口令或证书
加密保证	无	有	有
地址分配	有	有	无
客户端	XP自带或第三方	免安装式浏览器插件	通常为厂家设备内部实现， PC使用厂家专用客户端
兼容互通性	优良，基本上各个厂家都兼容XP客户端	PC使用浏览器打开VPN，与浏览器相关	优良，已经达到工业化标准成都，大部分厂家能够互通
资源消耗	轻	重	中
性能加速	无	加密卡硬件加速	加密卡硬件加速
隧道穿越NAT	可以	可以	可以

从上表可以看出，L2TP和SSL VPN确实更适合远端PC，特别是SSL VPN就是专门为远端PC开发的VPN技术。而IPSec更适合在分支和总部之间搭建VPN隧道。

## 参考

1. [VPN](#)
2. [PPTP\(Point to Point Tunneling Protocol\)](#)
3. [L2TP\(Layer 2 Tunneling Protocol\)](#)

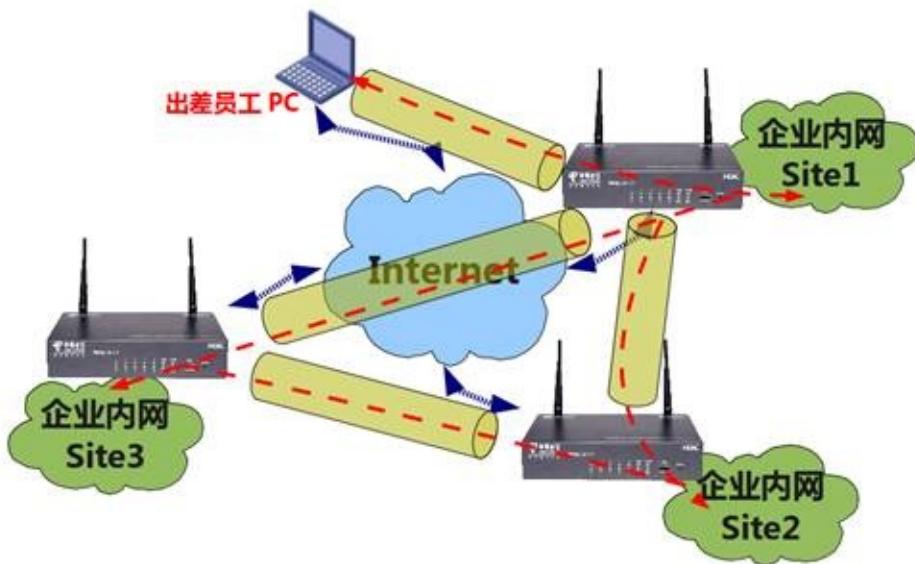
4. IPSec
5. SSL/TLS(Secure Sockets Layer / Transport Layer Security)
6. Client/Serveur SSL/TLS multiplateformes avec OpenSSL
7. OSI 7 LAYER MODEL

# IPSec VPN

IPSec VPN是目前VPN技术中点击率非常高的一种技术，同时提供VPN和信息加密两项技术。

## IPSec VPN应用场景

### IPSec VPN 应用需求



IPSec VPN的应用场景分为3种：

1. Site-to-Site (站点到站点或者网关到网关)：如弯曲评论的3个机构分布在互联网的3个不同的地方，各使用一个商务领航网关相互建立VPN隧道，企业内网（若干PC）之间的数据通过这些网关建立的IPSec隧道实现安全互联。
2. End-to-End (端到端或者PC到PC)：两个PC之间的通信由两个PC之间的IPSec会话保护，而不是网关。
3. End-to-Site (端到站点或者PC到网关)：两个PC之间的通信由网关和异地PC之间的IPSec进行保护。
4. VPN只是IPSec的一种应用方式，IPSec其实是 IP Security 的简称，它的目的是为IP提供高安全性特性，VPN则是在实现这种安全特性的方式下产生的解决方案。IPSec是一个框架性架构，具体由两类协议组成：

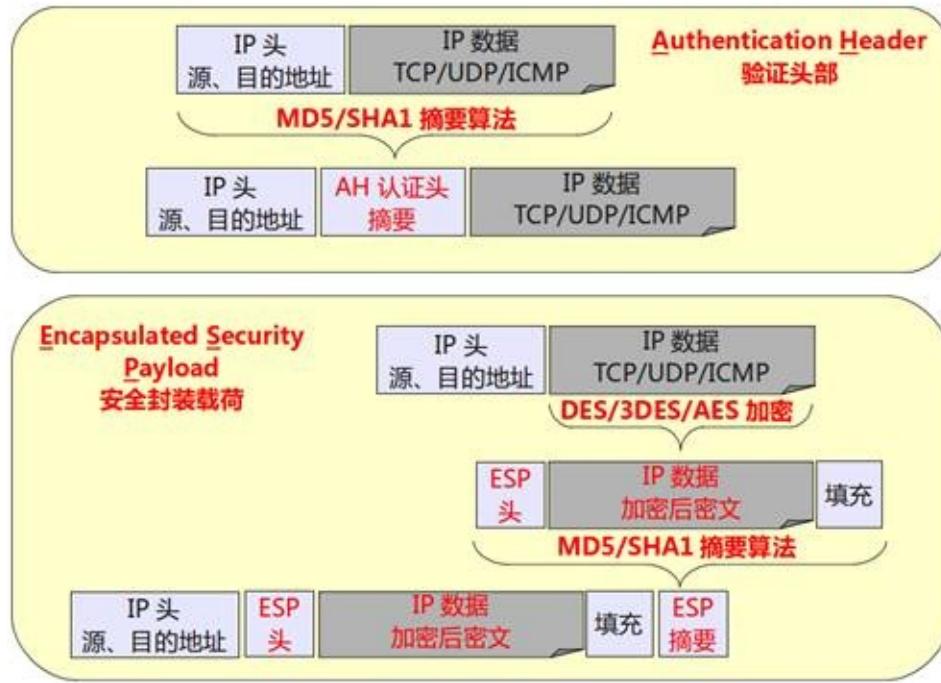
4. AH协议（ Authentication Header，使用较少）：可以同时提供数据完整性确认、数据来源确认、防重放等安全特性；AH常用摘要算法（单向Hash函数）MD5和SHA1实现该特性。
5. ESP协议（ Encapsulated Security Payload，使用较广）：可以同时提供数据完整性确认、数据加密、防重放等安全特性；ESP通常使用DES、3DES、AES等加密算法实现数据加密，使用MD5或SHA1来实现数据完整性。

为何AH使用较少呢？因为AH无法提供数据加密，所有数据在传输时以明文传输，而ESP提供数据加密；其次AH因为提供数据来源确认（源IP地址一旦改变，AH校验失败），所以无法穿越NAT。当然，IPSec在极端的情况下可以同时使用AH和ESP实现最完整的安全特性，但是此种方案极其少见。

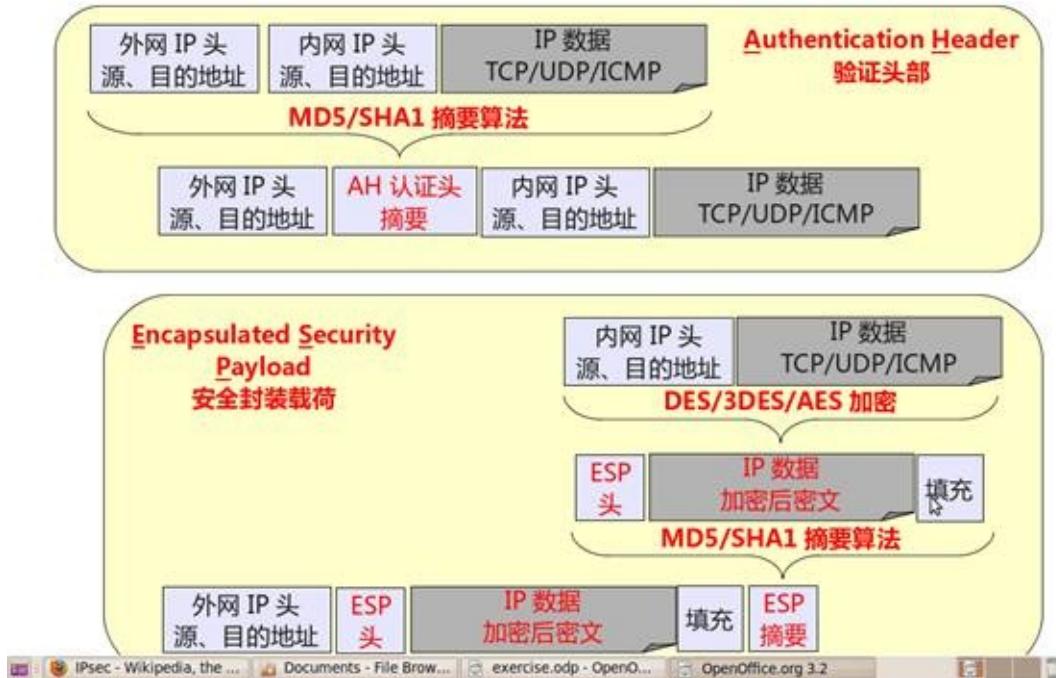
## IPSec封装模式

介绍完IPSec VPN的场景和IPSec协议组成，再来看一下IPSec提供的两种封装模式（传输 Transport 模式和隧道 Tunnel 模式）

### IPSec 的两种应用方式——传输模式



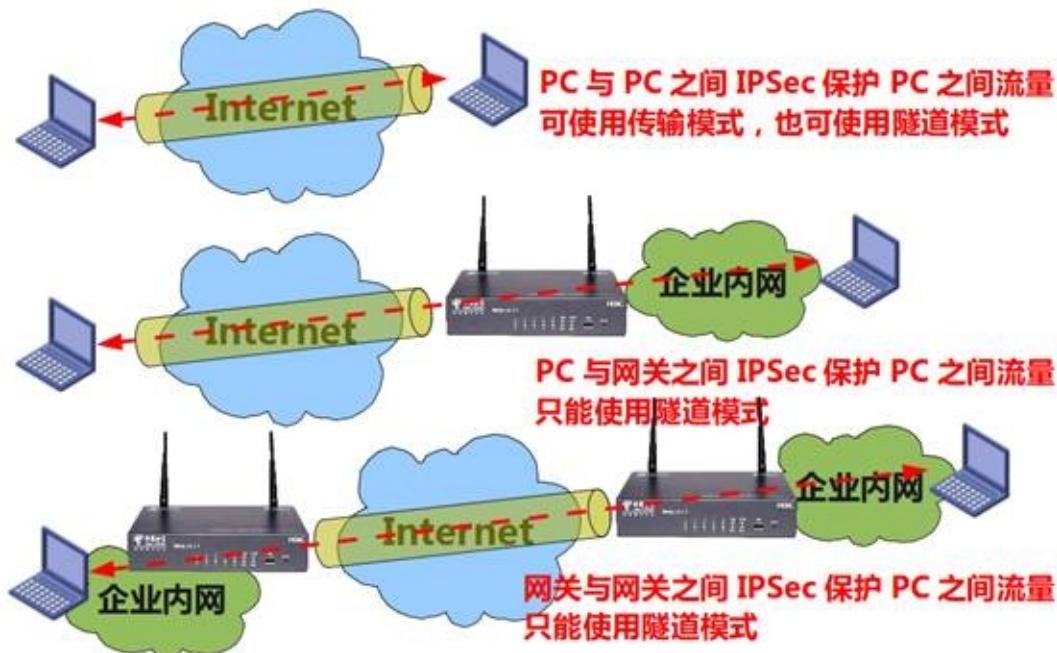
### IPSec 的两种应用方式——隧道模式



可以发现传输模式和隧道模式的区别：

1. 传输模式在AH、ESP处理前后IP头部保持不变，主要用于End-to-End的应用场景。
2. 隧道模式则在AH、ESP处理之后再封装了一个外网IP头，主要用于Site-to-Site的应用场景。

### IPSec 隧道模式和传输模式的适用场景

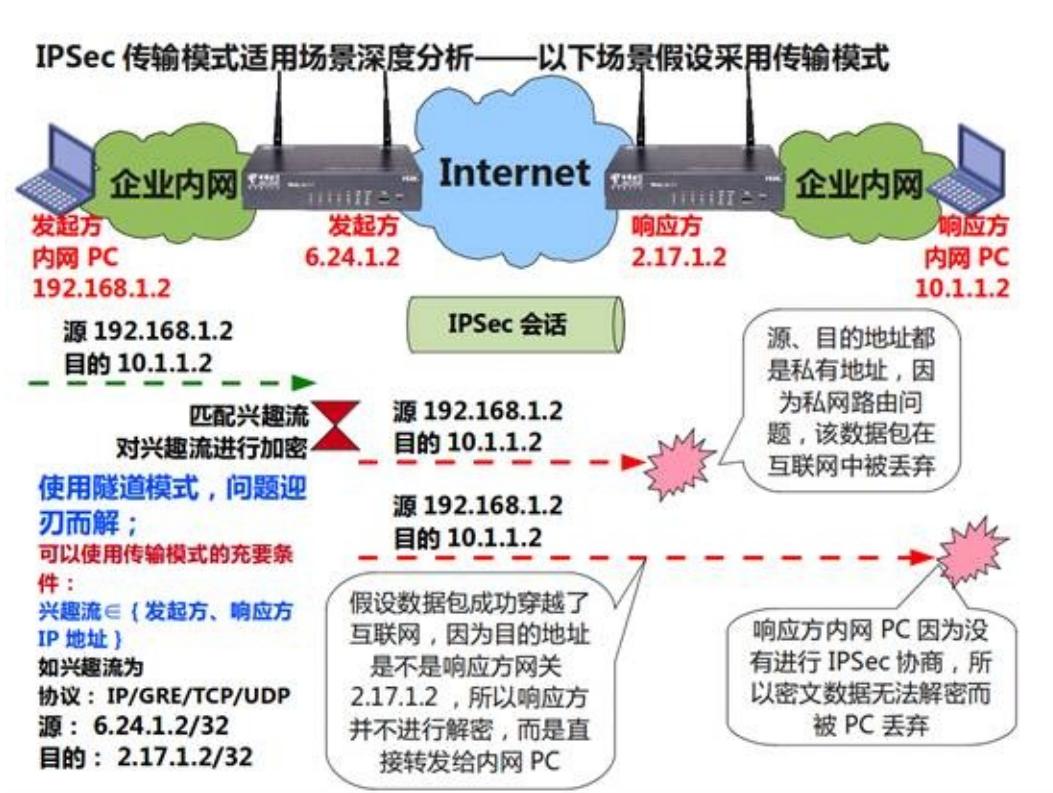


从这张图的对比可以看出：

1. 隧道模式可以适用于任何场景
  2. 传输模式只能适合PC到PC的场景

隧道模式虽然可以适用于任何场景，但是隧道模式需要多一层IP头（通常为20字节长度）开销，所以在PC到PC的场景，建议还是使用传输模式。

为了使大家有个更直观的了解，我们看看下图，分析一下为何在Site-to-Site场景中只能使用隧道模式：

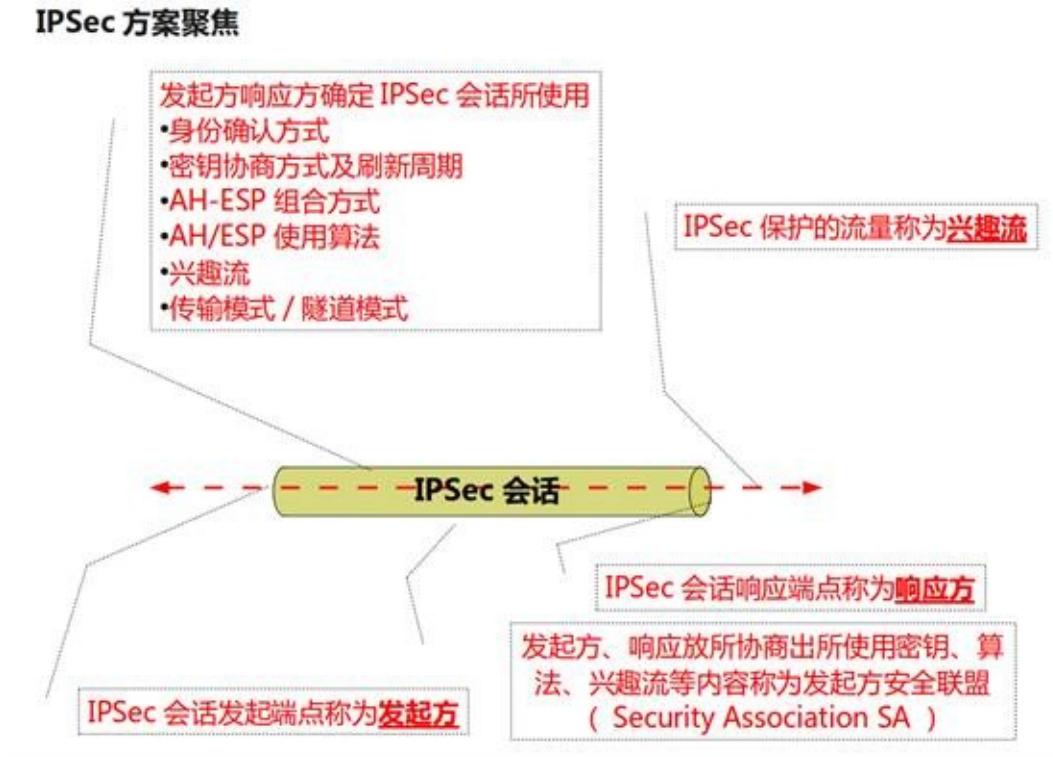


如上图所示，如果发起方内网PC发往响应方内网PC的流量满足网关的兴趣流匹配条件，发起方使用传输模式进行封装：

1. IPSec会话建立在发起方、响应方两个网关之间。
  2. 由于使用传输模式，所以IP头部并不会有任何变化，IP源地址是192.168.1.2，目的地址是10.1.1.2。
  3. 这个数据包发到互联网后，其命运注定是杯具的，为什么这么讲，就因为其目的地址是10.1.1.2吗？这并不是根源，根源在于互联网并不会维护企业网络的路由，所以丢弃的可能性很大。
  4. 即使数据包没有在互联网中丢弃，并且幸运地抵达了响应方网关，那么我们指望响应方网关进行解密工作吗？凭什么，的确没什么好的凭据，数据包的目的地址是内网PC的10.1.1.2，所以直接转发了事。
  5. 最杯具的是响应方内网PC收到数据包了，因为没有参与IPSec会话的协商会议，没有对应的SA，这个数据包无法解密，而被丢弃。

我们利用这个反证法，巧妙地解释了在Site-to-Site情况下不能使用传输模式的原因。并且提出了使用传输模式的充要条件：兴趣流必须完全在发起方、响应方IP地址范围内的流量。比如在图中，发起方IP地址为6.24.1.2，响应方IP地址为2.17.1.2，那么兴趣流可以是源6.24.1.2/32、目的是2.17.1.2/32，协议可以是任意的，倘若数据包的源、目的IP地址稍有不同，对不起，请使用隧道模式。

## IPSec 协商



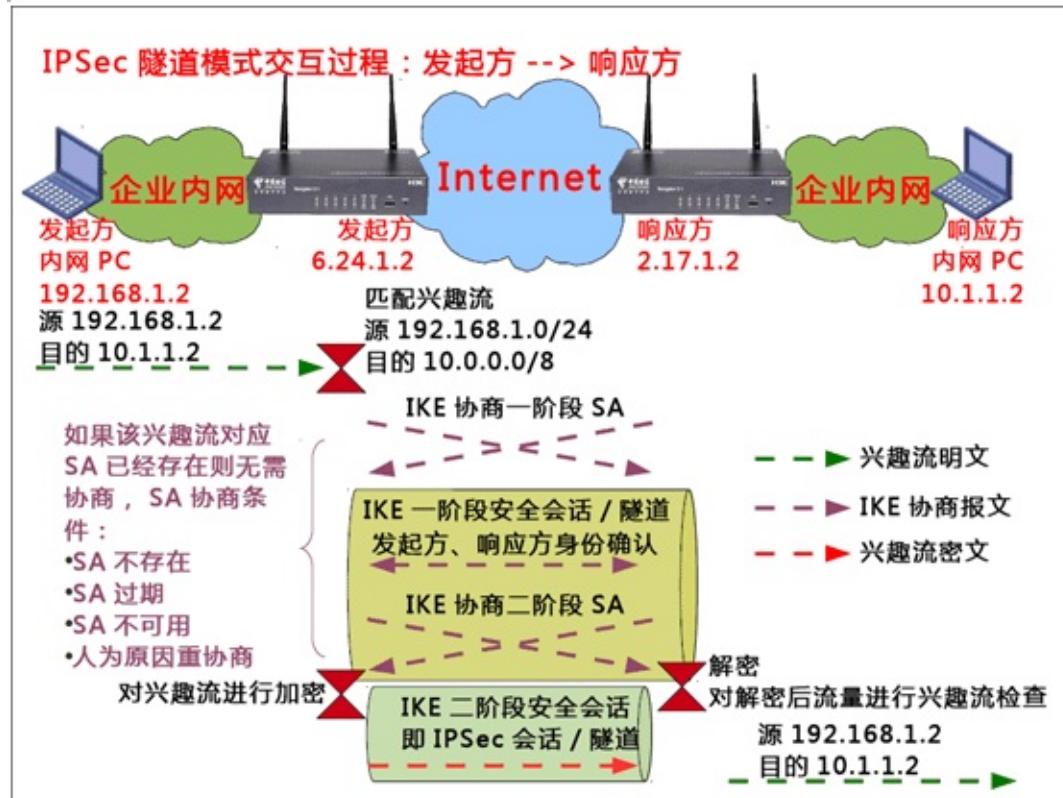
IPSec除了一些协议原理外，我们更关注的是协议中涉及到方案制定的内容：

1. 兴趣流：IPSec是需要消耗资源的保护措施，并非所有流量都需要IPSec进行处理，而需要IPSec进行保护的流量就称为兴趣流，最后协商出来的兴趣流是由发起方和响应方所指定兴趣流的交集，如发起方指定兴趣流为192.168.1.0/24->10.0.0.0/8，而响应方的兴趣流为10.0.0.0/8->192.168.0.0/16，那么其交集是192.168.1.0/24<-->10.0.0.0/8，这就是最后会被IPSec所保护的兴趣流。
2. 发起方：Initiator，IPSec会话协商的触发方，IPSec会话通常是由指定兴趣流触发协商，触发的过程通常是将数据包中的源、目的地址、协议以及源、目的口号与提前指定的IPSec兴趣流匹配模板如ACL进行匹配，如果匹配成功则属于指定兴趣流。指定兴趣流只是用于触发协商，至于是否会被IPSec保护要看是否匹配协商兴趣流，但是在通常实施方案过程中，通常会设计成发起方指定兴趣流属于协商兴趣流。
3. 响应方：Responder，IPSec会话协商的接收方，响应方是被动协商，响应方可以指定兴趣流，也可以不指定（完全由发起方指定）。
4. 发起方和响应方协商的内容主要包括：双方身份的确认和密钥种子刷新周期、AH/ESP的

组合方式及各自使用的算法，还包括兴趣流、封装模式等。

5. SA：发起方、响应方协商的结果就是曝光率很高的SA，SA通常包括密钥及密钥生存期、算法、封装模式、发起方、响应方地址、兴趣流等内容。

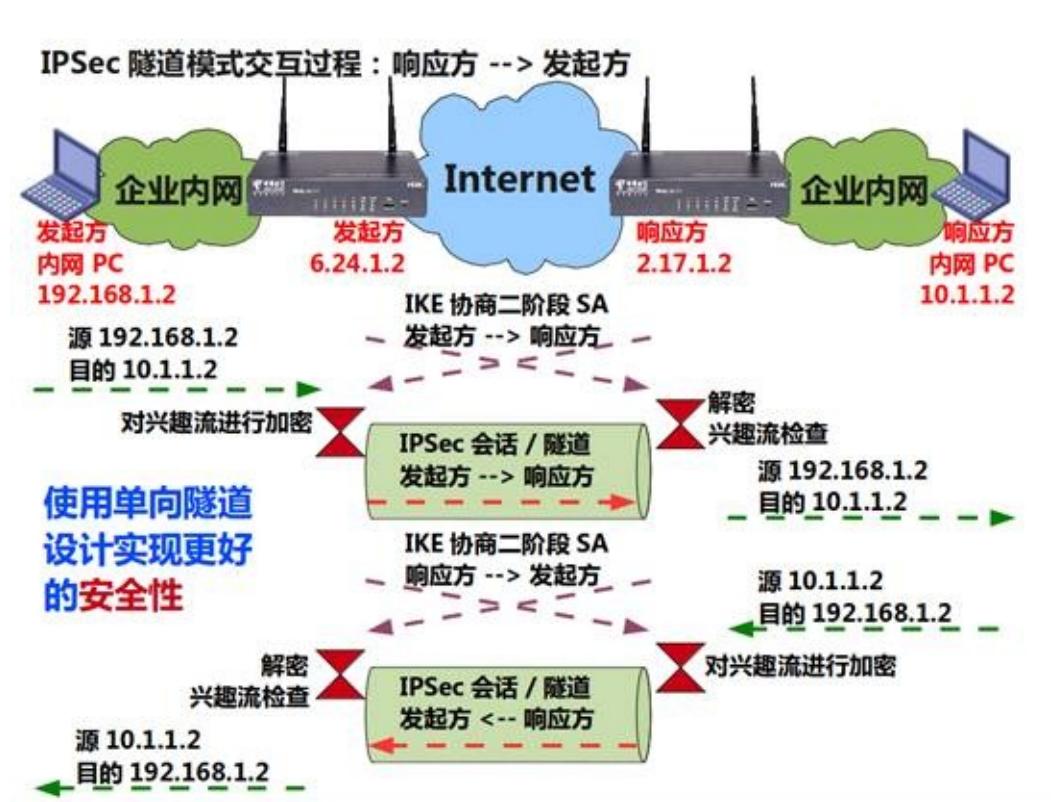
我们以最常见的IPSec隧道模式为例，解释一下**IPSec**的协商过程：



上图描述了由兴趣流触发的IPSec协商流程，原生IPSec并无身份确认等协商过程，在方案上存在诸多缺陷，如无法支持发起方地址动态变化情况下的身份确认、密钥动态更新等。伴随IPSec出现的IKE（Internet Key Exchange）协议专门用来弥补这些不足：

1. 发起方定义的兴趣流是源192.168.1.0/24目的10.0.0.0/8，所以在接口发送发起方内网PC发给响应方内网PC的数据包，能够得以匹配。
2. 满足兴趣流条件，在转发接口上检查SA不存在、过期或不可用，都会进行协商，否则使用当前SA对数据包进行处理。
3. 协商的过程通常分为两个阶段，第一阶段是为第二阶段服务，第二阶段是真正的为兴趣流服务的SA，两个阶段协商的侧重有所不同，第一阶段主要确认双方身份的正确性，第二阶段则是为兴趣流创建一个指定的安全套件，其最显著的结果就是第二阶段中的兴趣流在会话中是密文。

IPSec中安全性还体现在第二阶段**SA**永远是单向的：



从上图可以发现，在协商第二阶段SA时，SA是分方向性的，发起方到响应方所用SA和响应放到发起方SA是单独协商的，这样做好处在于即使某个方向的SA被破解并不会波及到另一个方向的SA。这种设计类似于双向车道设计。

# SSL VPN

SSL VPN 作为远程接入型的VPN，已经具备非常广阔前景，它的主要适应场景是取代 L2TP Over IPSec，但功能要比 L2TP Over IPSec 更丰富，方案也更加灵活。

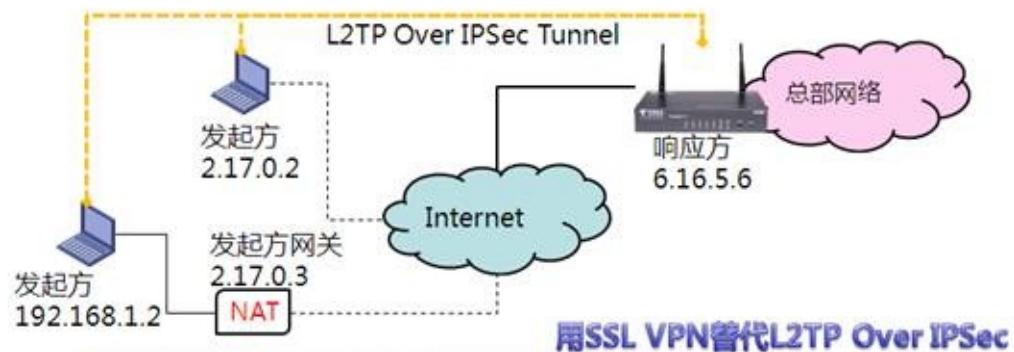
## SSL VPN 简介

何谓 SSL VPN，首先要从 SSL 谈起，使用网络不能不提的是各个网站，浏览网站使用浏览器，网络上传送网页的协议叫HTTP，它是明文传播的，传播内容可以被黑客读取。而SSL全名叫 Secure Session Layer（安全会话层），其最初目的是给 HTTP 加密使用的安全套件，使用 SSL 的 HTTP，也就摇身一变成了 HTTPS，端口也从 HTTP 的 80 变成了 443。由于 HTTPS 具备安全性，也具备传输数据的能力，也就被研究 VPN 技术的专家盯上了，觉得 HTTPS 可以用于组建 VPN 方案，于是乎 SSL VPN 技术就呼之欲出了。经过多年的发展，SSL 版本发展到了 3.0，也被标准组织采纳为 TLS（Transport Layer Security 传输层安全）1.0 之中，所以 SSL VPN 也叫 TLS VPN。下面是 SSL 与 SSL VPN、TLS 的区别：

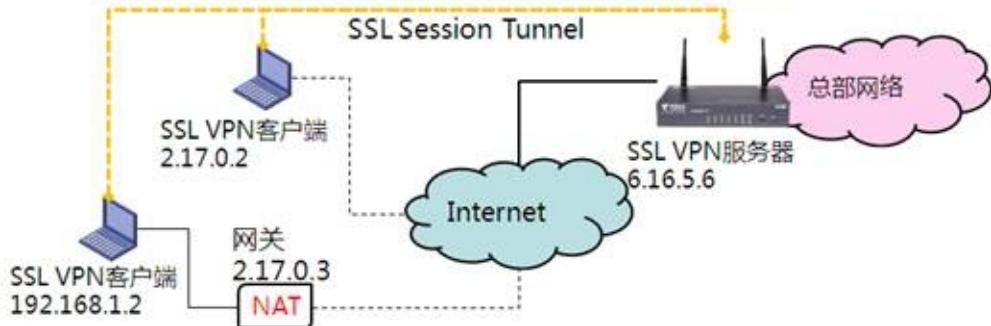
1. SSL：单纯的实现对某些 TCP 应用的保护，如 HTTPS 和 SFTP；
2. SSL VPN：利用 TCP 的传输作用以及 SSL 对 TCP 会话的保护，实现 VPN 业务，被保护的 VPN 业务可以是 TCP 的、也可以 UDP，纯 IP 的应用；
3. TLS：在 SSL 上进行扩展，能够直接实现对 UDP 应用的保护，这也是传输层安全的最佳注释。

## SSL VPN 的使用场景

### PC远程接入VPN的新贵——SSL VPN



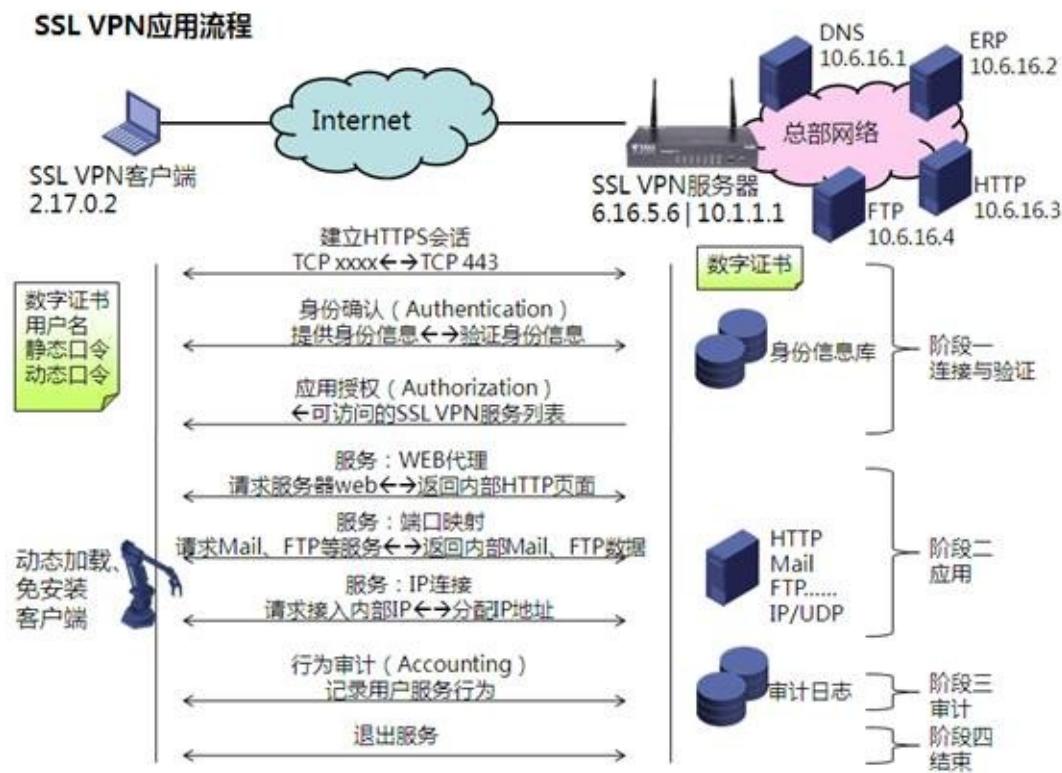
用SSL VPN替代L2TP Over IPSec



L2TP 实现的是远程接入VPN，而 IPSec 为 L2TP 提供安全保护，这种应用已经非常成熟，但属于两个协议的生硬组合，在方案上不是特别灵活。而 SSL VPN 是天然的安全远程接入，在方案上，特别是权限控制、应用粒度上有独到之处，成为目前远程接入领域的香饽饽，目前已经超越了技术范畴，而成为一个安全网络服务框架。

## SSL VPN应用

### SSL VPN应用整体流程

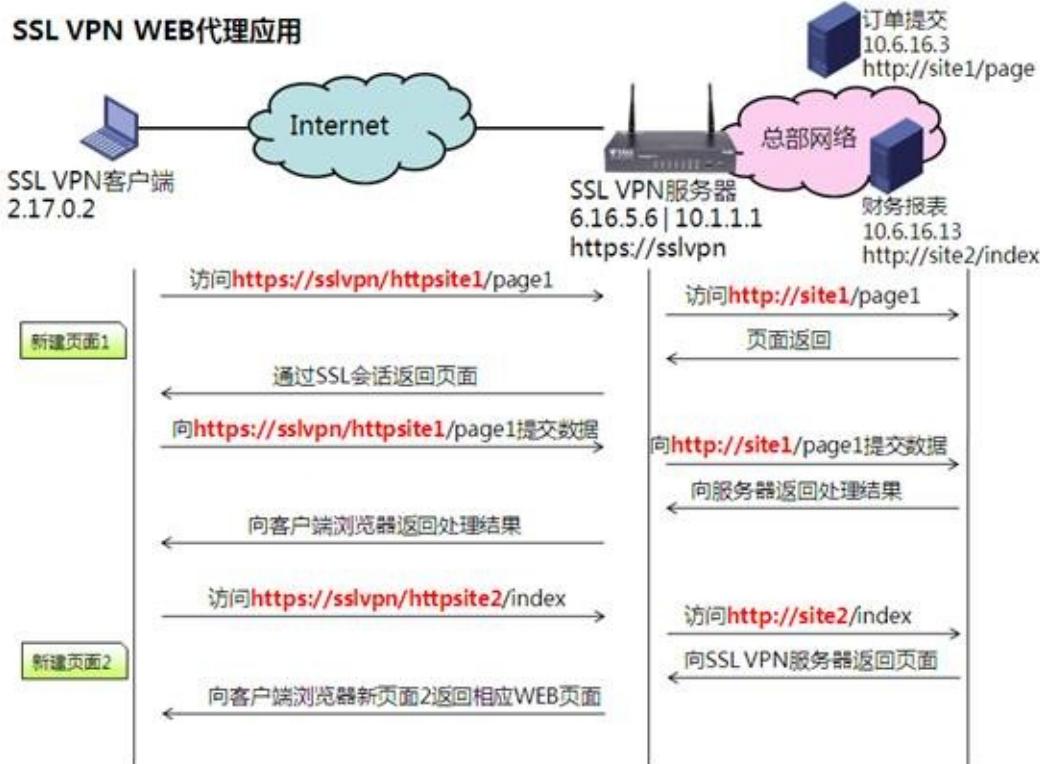


SSL VPN 最常见的入口还是网页，所以推广起来特别方便：

1. 使用者只需要记住VPN的网站（通常是HTTPS），用浏览器打开该网站；
2. 输入使用者的身份信息，身份信息可以是用户名、数字证书（如USB-Key）、静态口令、动态口令的至尊组合，确保身份不泄露、不假冒；
3. 选择服务种类，其中WEB代理是最为简单的应用，也是控制粒度最细的SSL VPN应用，可以精确地控制每个链接；
4. 端口映射是粒度仅次于WEB代理的应用，它通过TCP端口映射的方式（原理上类似于NAT内部服务器应用），为使用者提供远程接入TCP的服务，它需要专门的、与服务器配套的SSL VPN客户端程序帮忙；
5. IP连接是SSL VPN中粒度最粗的服务，但也是使用最广泛的，它实现了类似于L2TP的特性，所有客户端都可以从服务器获得一个VPN地址，然后直接访问内部服务器，它也需要专门的SSL VPN客户端程序帮忙；
6. SSL VPN由于处在TCP层，所以可以进行丰富的业务控制，如行为审计，可以记录每名用户的所有操作，为更好地管理VPN提供了有效统计数据；
7. 当使用者退出SSL VPN登陆页面时，所有上述安全会话会统统释放。

以上7个步骤可以划分为三个阶段：阶段一是连接与验证、阶段二是VPN应用、阶段三是审计与退出。连接与验证、审计与退出都是统一流程，也比较简单。

## Web代理

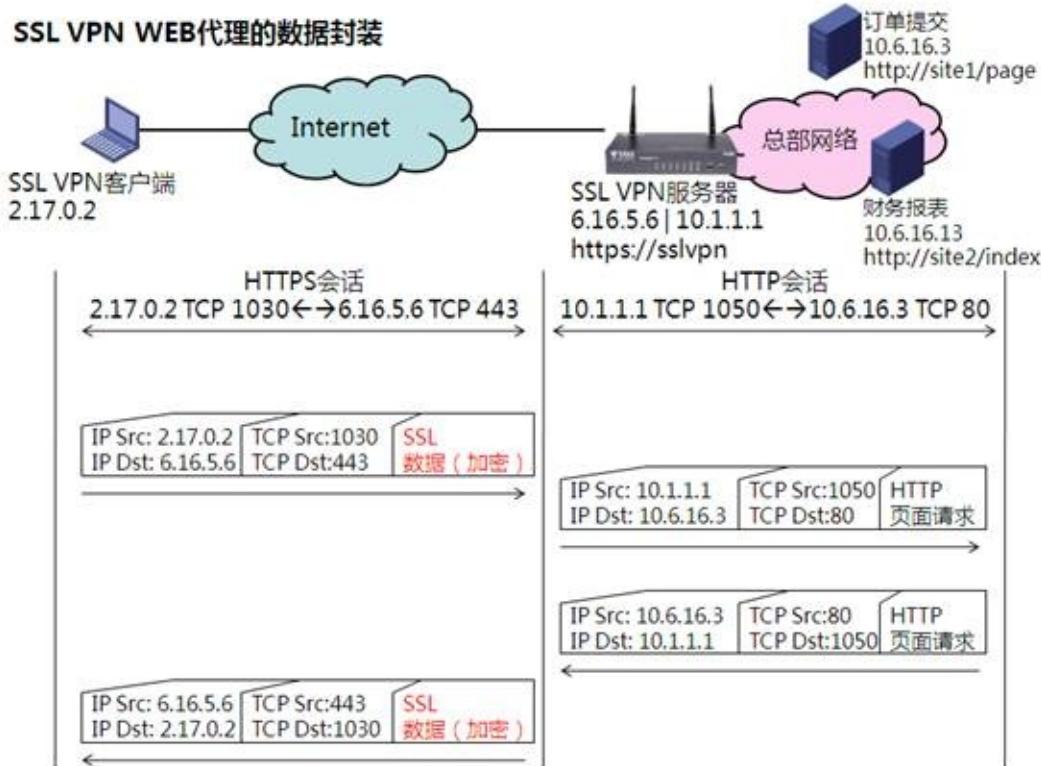


由于SSL是封装在TCP上的，穿越NAT不是问题，所以在示例中客户端使用公网地址进行介绍：

1. 假设SSL VPN的WEB站点的互联网域名是<https://sslvpn>，该WEB站点对应的主机则是SSL VPN服务器；
2. 使用者登陆SSL VPN的WEB页面后，WEB代理一栏会有许多链接，如内部财务报表、订单提交等内部网站；
3. 假设“订单提交”网站在单位内部私有URL是<http://site1/page>，那么在SSL VPN服务器上的订单提交链接URL则会进行相应的修改，变成<https://sslvpn/httpsite1/pate>，相当于SSL VPN站点的内部链接；
4. 使用者点击“订单提交”链接后，会新建一个浏览器窗口，打开链接<https://sslvpn/httpsite1/page>，也就是说对于使用者而言，订单提交像是SSL VPN站点的一个链接，而非另外一个站点，所有的访问都终结在SSL VPN站点；
5. SSL VPN站点的所有者SSL VPN服务器在接收到使用者对<https://sslvpn/httpsite1/page>的页面请求后，SSL VPN服务器会做WEB代理的工作，即以内部地址10.1.1.1向真正的“订单提交”站点10.6.16.3访问页面<http://site1/page>；
6. 可以发现整个页面访问是由使用者与服务器之间的HTTPS会话、服务器与“订单提交”站点的HTTP会话连接而成的，服务器在这个访问中起的是WEB代理作用，因为在“订单提交”站点看来，访问者IP是服务器，而不是最终用户IP；
7. 而使用者访问另外一个站点“财务报表”，也是类似过程。

WEB代理因为原理简单，实现起来较为容易，因为传统的**WEB Proxy**代理是两段**HTTP**会话的衔接，而**SSL VPN**的**WEB**代理则把用户与服务器的连接把**HTTP**换成了**HTTPS**、并对网站的**URL**进行了替换而已，从图中我们可以看到红色部分即为**URL**的替换。

## SSL VPN web代理的数据封装



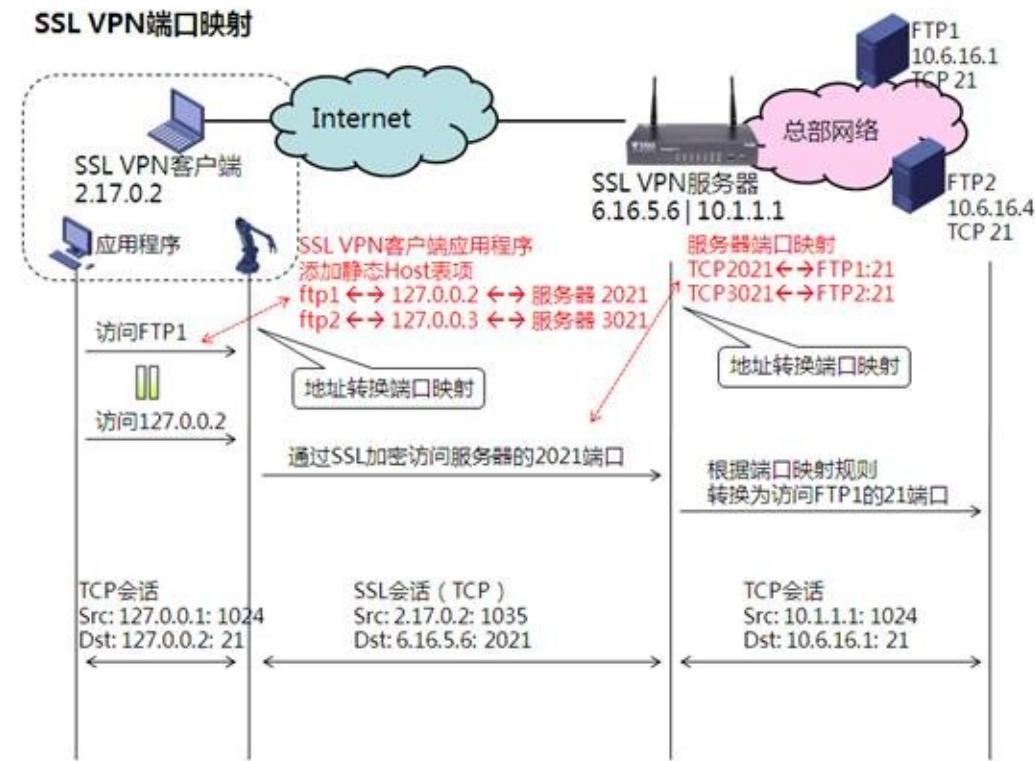
从这张封装原理图，我们可以比较清楚地看到**HTTPS**与**HTTP**会话在这个访问过程中的衔接。可能有人会问，让“订单提交”站点直接在互联网提供**HTTPS**服务，直接用一个**HTTPS**会话不是更好，原理上当然更好，但是有更多的现实问题：

1. 直接在互联网提供**HTTPS**服务，需要一个互联网地址和公共域名，这两样都是要花钱申请的，使用SSL VPN统一接入，这么多内部站点只需要一个公网地址、一个公网域名，多划算；
2. 订单提交真的需要开放到互联网吗？订单提交都是公司内部业务，访问量也不大，直接开放到互联网并不能有更多的提速效果；
3. 开放到互联网怎么保证安全，财务报表等信息都是公司机密，老老实实放在内网，前面通过SSL VPN服务器挡着，即使有攻击也只是攻击SSL VPN服务器，内网服务器还是很安全的。

综上所述对于一些内部站点，使用SSL VPN还是相当有好处的，特别是在拥有一款强大的SSL VPN服务器的时候。

## 端口映射

刚才讲的是WEB代理，对于一些内部服务器并不是WEB站点，那WEB代理还能使用吗？不能使用了，比如内部站点是FTP应用，那么访问不可能由HTTPS会话和FTP会话衔接而成，SSL VPN必须想其余办法。由于**SSL**只能封装在**TCP**之上，所以端口映射服务器只能针对内部的**TCP**应用，如FTP。



在端口映射中，SSL VPN的使用者会从SSL VPN页面自动加载一个客户端程序，我们姑且就叫它SSL VPN客户端程序吧，它是怎么使端口映射工作的呢，我们假设内部有两个FTP服务器，一个叫FTP1，内部地址10.6.16.1，另一个是FTP2，内部地址10.6.16.4，都是监听TCP 21端口：

1. SSL VPN服务器为这两个内部服务器做了端口映射，TCP 2021端口映射到FTP1的TCP 21，3021则映射到FTP2；
2. SSL VPN服务器会让使用者PC自动加载SSL VPN客户端程序，并根据这两个映射生成两个静态host映射表项，告诉使用者PC访问FTP1其实就是访问127.0.0.2，访问FTP2就是访问127.0.0.3，127.0.0.0/8称为环回地址，及该地址只能在PC内部使用，不可能被发出到PC之外，那么SSL VPN客户端程序就监听这两个内部地址；
3. 使用者访问FTP1，其实访问的是TCP 127.0.0.2:21，所有数据都会被SSL VPN客户端程序监听，客户端程序会进行代理，变成访问服务器TCP 6.16.5.6:2021，该TCP访问会使用SSL进行加密；
4. 大家可能会问，为何需要个客户端程序进行代理呢，使用者的应用程序不能直接和SSL VPN服务器建立SSL会话吗？这个问题很好，使用者的应用程序的确无法直接建立SSL会话，所以使用客户端程序代劳，这种方式可以让所有TCP应用都能够享用SSL VPN服

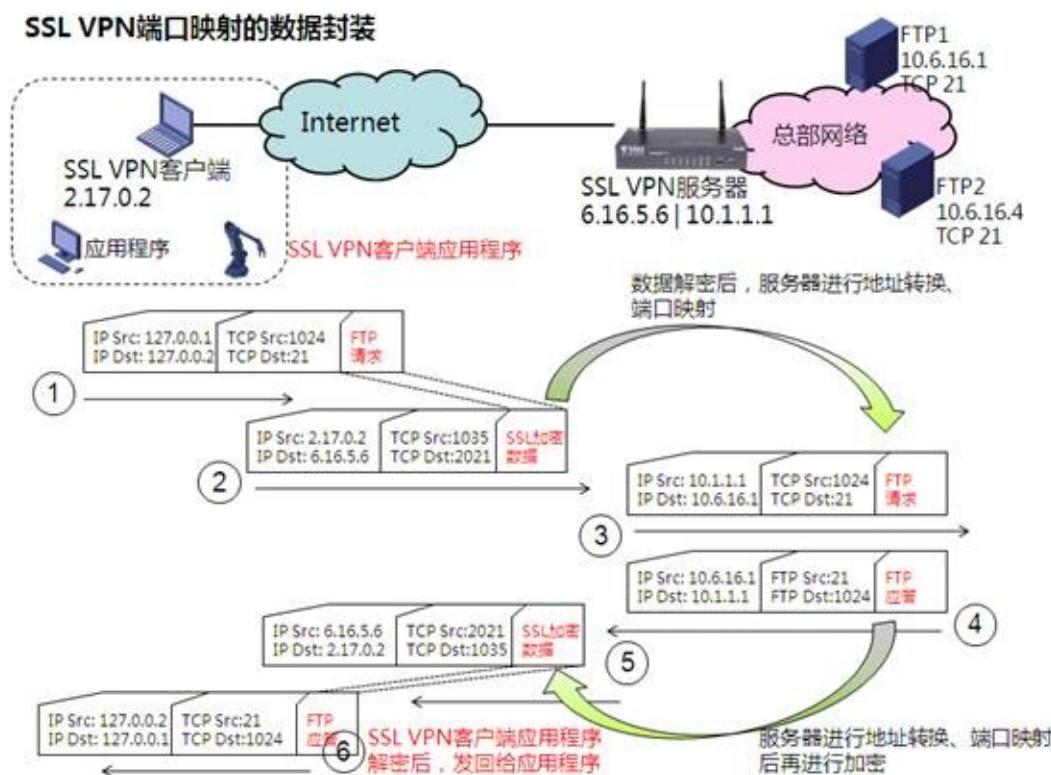
务；

5. 服务器接收到SSL加密的请求后，首先会进行解密，然后根据端口映射，会向内部FTP1站点TCP 10.6.16.1:21发起访问；
6. 反向转发以及访问FTP2类似。

我们可以发现这种端口映射使整个访问过程由三段会话组成：

使用者应用程序与**SSL VPN**客户端程序的普通**TCP**会话 **SSL VPN**客户端程序与服务器的**SSL**会话 服务器与内部站点的普通**TCP**会话。

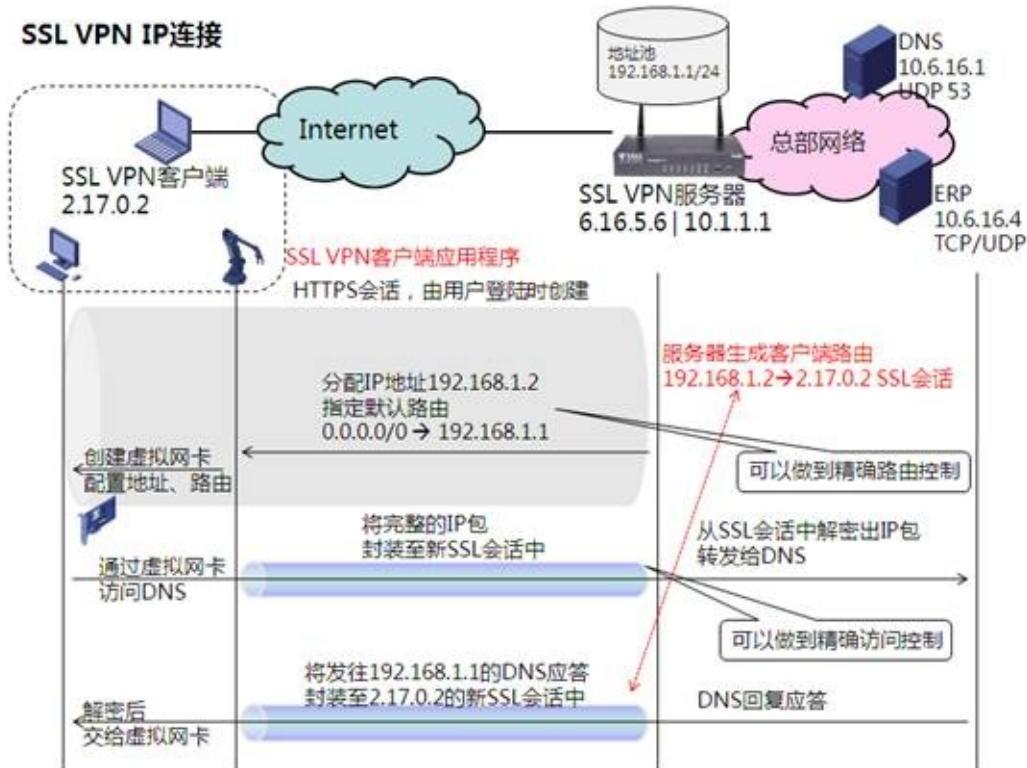
## SSL VPN端口映射的数据封装



从这张图可以看出更详细的数据封装过程以及会话衔接过程，使用**SSL VPN**服务器进行端口映射的好处与**WEB**代理类似，可以牺牲**SSL VPN**服务器，保护内部服务器。

## IP连接

**WEB**代理是专门针对**WEB**应用的，端口映射则受制于**SSL**只支持**TCP**应用，如果使用者要任意访问一个内部服务器的任意协议、端口，该如何是好呢？**IP**连接应运而生，**IP**连接可以完美地替代**L2TP**这种传统意义上的**VPN**：虚拟连接、内部地址、路由互联。

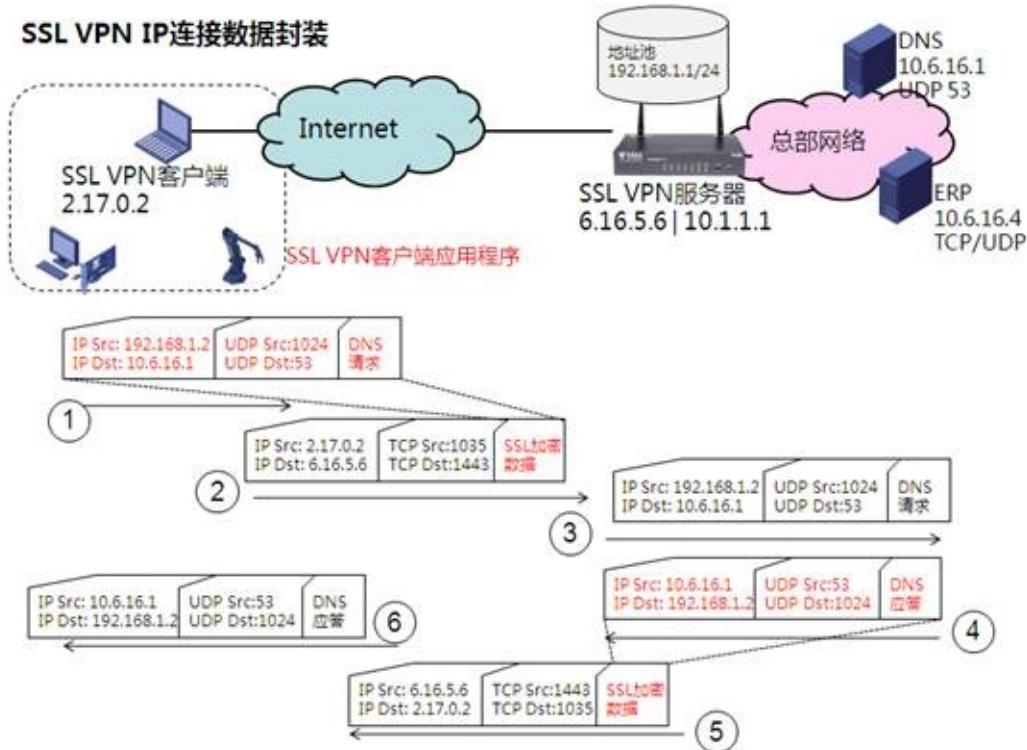


我们从上图来解释一下IP连接的原理：

1. 用户登陆SSL VPN页面后，会建立HTTPS会话，服务器通过这个会话给用户自动加载SSL VPN客户端程序；
2. 此时的SSL VPN客户端程序的目的是给用户PC创建一个虚拟网卡，以实现类似于L2TP那种到客户总部网络的VPN连接；
3. 虚拟网卡创建好后，服务器会给该用户从地址池中取一个地址分配给该用户，同时下发路由、DNS等信息，服务器针对该地址池也会有一个服务器地址192.168.1.1，作为所有客户端程序虚拟网卡的网关；
4. 此时SSL VPN客户端程序与服务器之间会建立一个全新的SSL会话，专门用来传输虚拟网卡与服务器之间的流量；
5. 假设用户要访问DNS 10.6.16.1，根据路由的关系，PC会通过虚拟网卡将DNS请求（源192.168.1.2目的10.6.16.1）转发给SSL VPN服务器192.168.1.1；
6. PC上的SSL VPN客户端程序会将虚拟网卡发出的IP包封装至新的SSL会话中，通过互联网传送到服务器；
7. 服务器进行解密，解封装后发现IP目的地址是10.6.16.1，那么就转发给DNS；
8. 反向过程以及访问ERP服务器10.6.16.4与此类似。

## SSL VPN IP连接数据封装

再来看一下数据封装过程，会有更加直观的认识：



在SSL VPN的IP连接中，客户端访问内部服务器不再像WEB代理、端口映射那应该多个会话衔接而成，而是一个内部地址端到端会话，穿越互联网的时候直接会话被封装至SSL会话中，和L2TP Over IPSec非常类似。

## SSL VPN总结

为什么说SSL VPN可以通过多粒度的服务呢？这是一个对比：

1. WEB代理，可以精确到对HTTP站点某些URL的控制；
2. 端口映射，可以精确到对某个端口的控制；
3. IP连接，可以精确到对某个IP地址的控制；

L2TP只能实现3，无法实现1和2，更是很难实现行为审计功能。

## SSL VPN局限性和代价

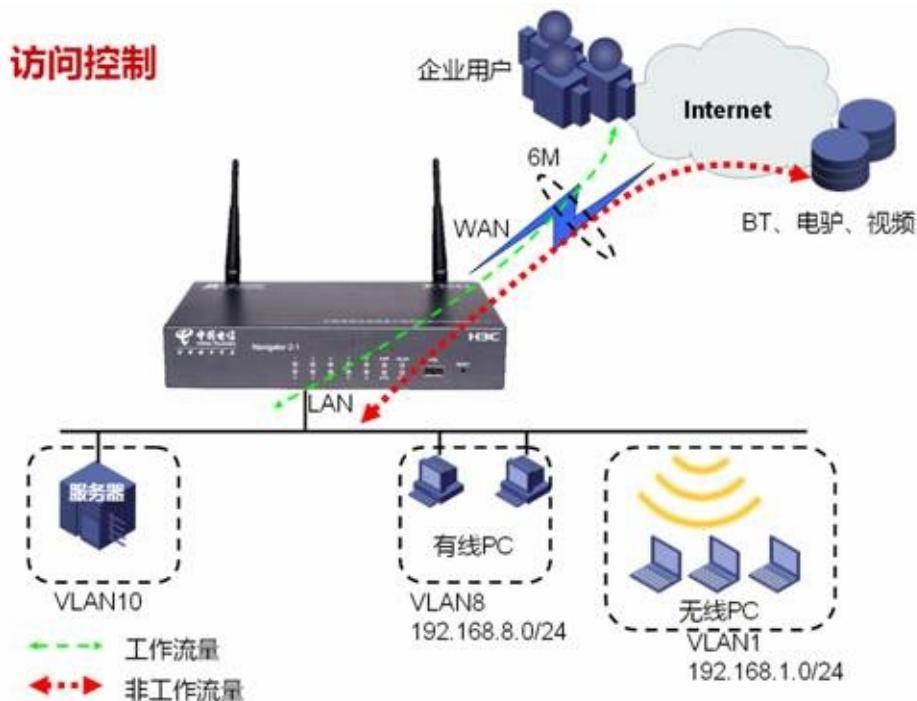
因此业界非常看好SSL VPN的前景，但使用SSL VPN必然也有一些局限性和代价：

1. 对于站点到站点的应用，效率上不如IPSec，所以通常用来取代L2TP方案，而不是IPSec方案；

2. SSL VPN的IP连接服务，在有连接的TCP中封装无连接的IP、UDP上效率不是很高，如果TCP中再封装TCP，在网络状况不稳定情况下，传输效率可能会急剧下降，但这个难题在被逐步功课之中，将来的TLS VPN可以实现在安全UDP会话，那情况就会好转很多；
3. SSL VPN客户端必须配合SSL VPN服务器，各个厂家的客户端都是自行开发的，无法互相兼容，由于客户端都是动态加载的，也就是说访问什么服务器，必然加载与之配套的客户端，不存在兼容性问题，但客户端是操作系统相关，甚至是浏览器相关的，很多厂家的SSL VPN只开发了基于Windows IE的客户端，使用Linux、BSD操作系统的使用者无疑就杯具了

# ICG(Internet Control Gateway)简介

## 自由互联和应用控制



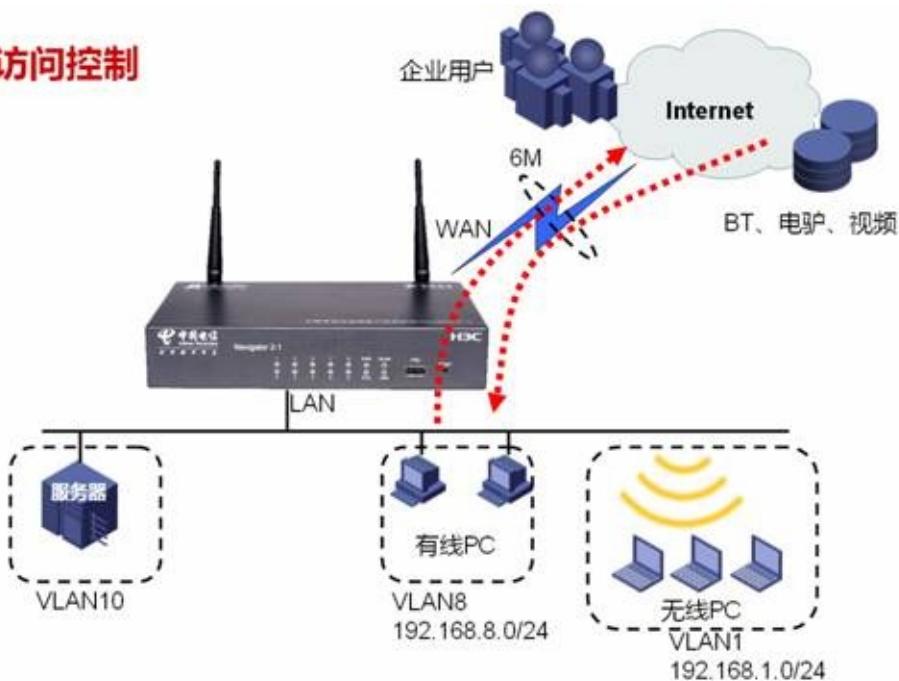
在网络带宽有限的环境下，带宽就变成一项很重要的资源，那么对于一个企业而言，有限的带宽如果被一些非工作流量所占据，而工作流量却因为带宽不够受到影响，这无疑是一件很糟糕的事情。今天我们就来讨论一下互联网访问控制。

为何事情会发展到如此地步呢？在互联网方兴未艾的时候根本就没人提要控制应用，提倡的都是自由互联，这是因为当互联网刚刚萌芽的时候，受到接入技术限制（以56K Modem为主），互联网上的应用基本上以静态的文本网页或者少量的图片为主，当时互联网是以吸引用户数为目的的，运营商则是通过提供接入服务数量为盈利点的。

随着互联网接入技术的革新（面向电话线路的xDSL和面向以太网的FTTx技术），用户数量的迅速膨胀，互联网应用开始井喷了，有人在互联网上发明了榨干每一滴带宽的P2P技术，有人则依赖高速带宽进行VPN互联或者企业信息发布，对于一般企业而言，后者能带来盈利，所以在一个有限的带宽上，必然要对类P2P技术进行限制，这就是互联网访问控制催生的重要原因。

## 基于网段的限速

## 访问控制



基于网段的限速是比较简单的，其原理在于对某一个网段每个IP地址进行平均带宽控制，而不是对应用进行控制，从逻辑的角度而言，对某网段的限制必然就是对其余网段的保证。它的生效要满足如下条件：

1. WAN口出现拥塞，以太网接口通过配置接口带宽和实时流量比较判断是否拥塞；xDSL则可以通过端口协商机制判断是否拥塞。
2. 对指定地址范围段内每一个IP所占用上下行带宽都限制在固定范围内，即这些IP都能得到公平的处理，不会出现某一个IP独占许多带宽的情况。

那么这里就有疑问了，假设互联网出口带宽是4M，内部PC有20台，我要对其中的16台进行限制，我应该怎么设置上下行流量应该限制在多少呢？

这里我们就要利用到TCP基础理论知识了，P2P只是改变了传统“客户-服务器”应用模式，传输层使用的还是TCP和UDP，我们知道TCP是通过滑动窗口机制来工作的，窗口越大，每秒发送的数据就越多，默认情况下窗口是根据传输时延情况不断增大的，也就是说TCP的速率会试探性地不断增长，但是一旦发生丢包（TCP通过计时和确认机制判断是否丢包）、重传或乱序，窗口都会以每次减半的方式缩小，所以TCP一旦出现丢包、重传、乱序是非常影响速率的，并且网络限速通常是随机丢弃方式，当到流量到达门限时丢包、重传、乱序的概率增大，窗口会迅速缩小，达到明显限速效果。那么UDP呢？UDP是一种发送后不管的传输协议，无法判断是否丢包，发送成功与否完全取决于应用层的实现，所以一般基于UDP的应用程序处于实现简单的原因，发送数据都不会很快，属于一种慢速传输技术。

基于对互联网应用和TCP理论的分析，针对目前互联网广泛使用应用进行试验，配置如下数值效果可以适配各种带宽情况，既能保证受限PC普通上网业务，又能避免他们滥用带宽，而不是采用平均速率（如4M带宽，内部20台PC，那么每台PC均享200k带宽）：

1. 下行1024Kbps，即1M带宽，通过各下载工具显示带宽大概在每秒100K字节（Byte）左右
2. 上行512Kbps，即0.5M带宽
3. 根据实际链路情况也可以将此数值放大1倍或缩小1半
4. 如果是高级网管人员，也可以自行设置数值

在这种情况下所有被限制的IP在网络拥塞情况下带宽都无法逾越这两个数值，比如在上图中，针对VLAN8的192.168.8.0/24和VLAN1的192.168.1.0/24进行限速，那么可以发现那些使用BT下载或看视频的PC，打开网页速度就很慢，根据用户心理，他也许会关闭BT以保证网页可以浏览。在VLAN8和VLAN1被限制的同时，VLAN9不受限制，各项应用运转顺畅。

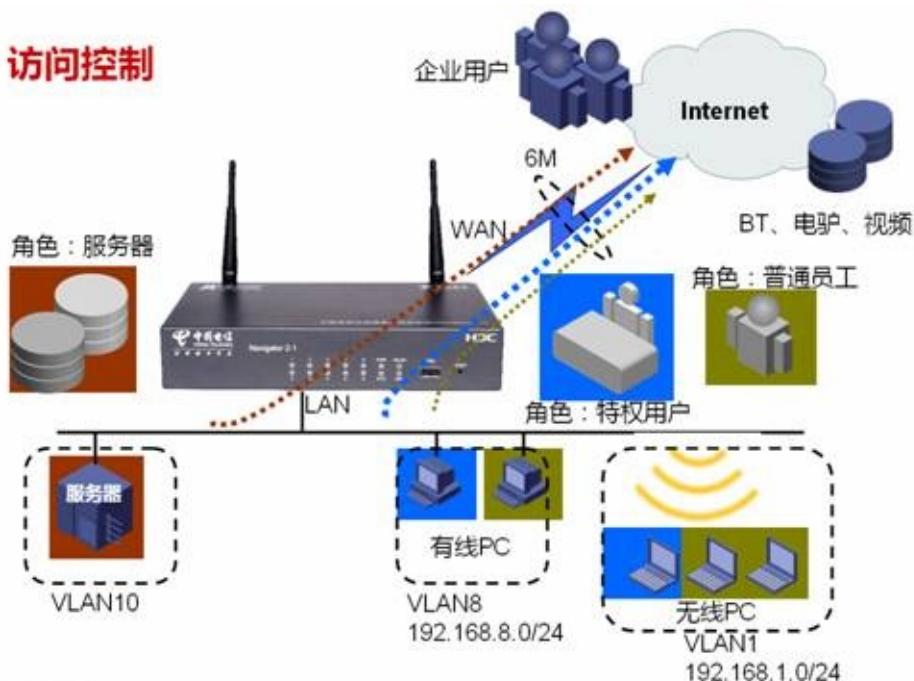
基于网段的限速要求提前把PC根据应用类型划分成不同的连续网段，这样有利于配置，比如上图就将需要受限的普通员工PC划分到VLAN8和VLAN1中，而服务器和主管使用PC则放入到不受限的VLAN9中。

## 基于角色的流量审计和应用控制

之前我们介绍的是比较简单的互联网控制策略，在拥塞发生时，对指定IP地址进行均等限速。随着网络应用丰富多样，用户产生了更加灵活的需求，对互联网控制策略也产生了基于用户角色进行控制和管理的理念，在桌面级网络设备领域，该理念已经被越来越多厂家所接受：

1. 内部用户将不再单纯根据连续网段区分，而是全面地通过主机名、IP地址、MAC地址、物理端口号灵活地确定一台内部主机的身份。但是真正在互联网访问控制中生效的主要还是IP地址，附加信息的作用是使用户能够更直观地掌握内部主机和IP地址的对应关系。
2. 预定义不同的角色，为每个角色制定不同的控制策略，比如普通用户不允许使用QQ、MSN、BT、电驴等工具，不允许访问开心网、优酷等视频网站；服务器用户只允许开放服务，不允许发起任何访问，对于开放服务流量进行带宽保证；特权用户不作任何限制，同时保证带宽。
3. 将不同的内部用户赋予不同的角色，因此互联网行为受到相应角色所控制。
4. 同时实时流量的审计，有助于网管人员能够及时了解各内网PC的应用动态，为每种角色接受哪些控制策略做好准备。

## 访问控制



基于用户角色的应用控制主要有两大类技术：

- 第一类是用户的识别，只有将用户识别后才能对用户分配角色，标识用户的信息包括主机名、IP地址、MAC地址、端口等信息，而实际上在WAN口真正生效的是IP地址，因为在IP网络中只有IP地址能够代表一个主机，引进主机名、MAC地址和端口等信息能够更直观地展现内部主机的信息，使普通用户也能对内部主机做到验明正身，是一种易用性的体现。下表展示了各协议或技术将这不同信息绑定在一起标识内部用户的：

信息 协议	主机名	IP 地址	MAC 地址	端口信息
主机名与 IP 对应	NetBIOS 协议			
IP 与 MAC 对应	ARP 协议			
MAC 与端口对应	MAC 地址学习			

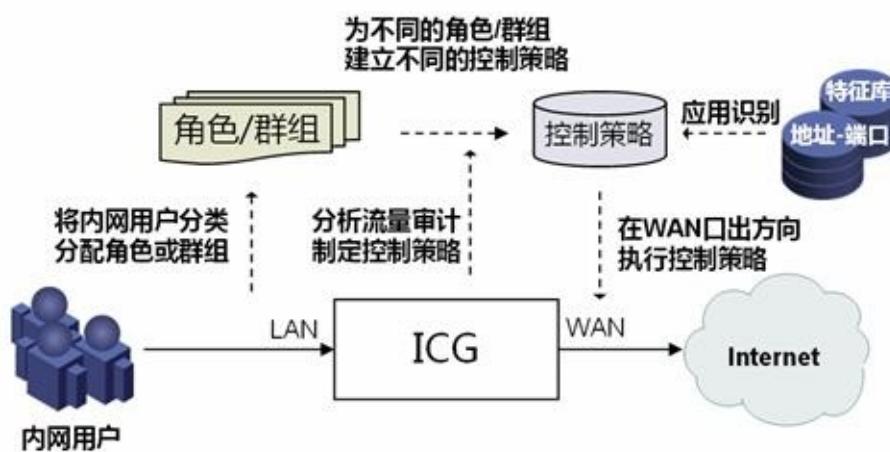
- 第二类是应用的识别，只有识别了应用才能正确的限制动作，现在互联网很多应用，如BT、QQ、炒股软件等都无法用传统的地址+端口来识别，这些应用使用的端口是动态的，必须使用特殊技术对应用层数据进行分析，这项技术被称为应用程序特征库，有点类似于杀毒软件的病毒库，应用程序特征库也需要持续更新以跟踪应用程序的变化，但往往滞后于应用程序的更新进度，因为特征库必须等到应用程序发布后才能进行分析更新，此外还有一个困难是很多软件如BT会对特征库进行加密，加密后的数据包就无法被特征库所识别了，俗话说道高一尺魔高一丈，加密是需要协商的，而协商过程是存在明文阶段的，因此可以在协商阶段进行识别，所以很多时候启用应用程序限制后，已经下载的BT和已经登录的QQ依然可以使用，新增的下载任务和新登录QQ才会失败，由于大

部分用户的BT和QQ都存在上上下下的过程，这种限制在实际环境的作用还是相当明显的；当然在应用识别中传统的地址+端口也保留下来，再新增一些URL过滤，基本上对各种网络应用达到比较完整的覆盖，可以说80%的网络流量都可以识别出来。

## 基于用户角色的应用控制实现原理

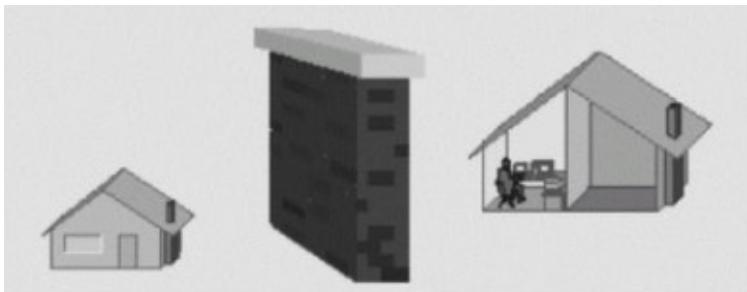
将上述涉及的几个技术要点结合起来，基于用户角色的应用控制实现原理可以用下面这张图来表示：

### 基于用户角色的应用控制——原理



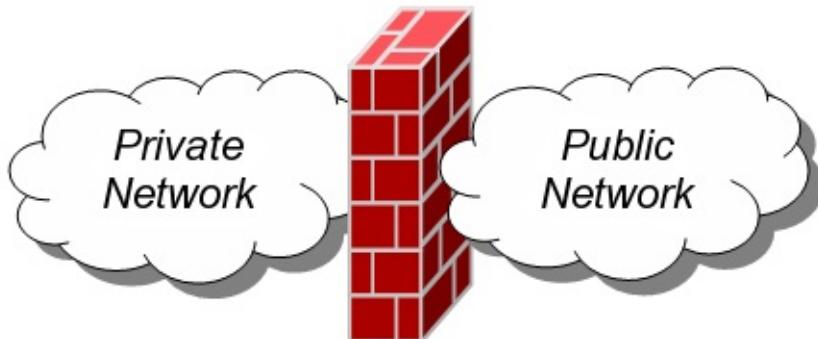
# 防火墙（ Firewall ）介绍

## 来源



防火墙的本义，是指古代构筑和使用木制结构房屋时，为防止火灾发生及蔓延，人们将坚固石块堆砌在房屋周围做为屏障，这种防护结构建筑就被称为防火墙。

现代网络时代引用此喻意，指隔离本地网络与外界网络或是局域网间与互联网或互联网的一道防御系统，借由控制过滤限制消息来保护内部网络数据的安全。

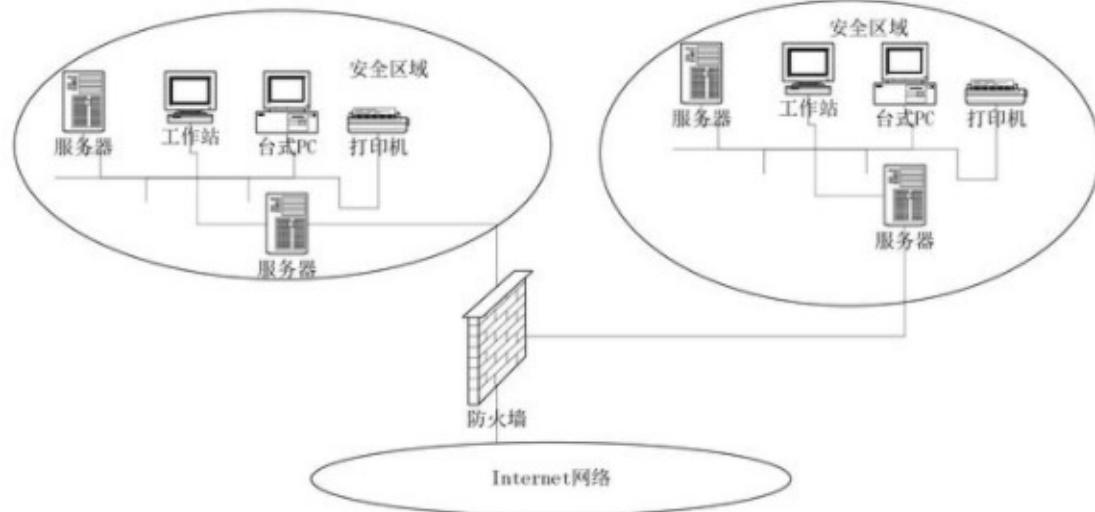


## 基础定义

传统意义上，防火墙（ Firewall ）是一个架设在互联网与企业内网之间的信息安全系统，根据企业预定的策略来监控往来的传输。

防火墙可能是一台专属的网络设备或是运行于主机上来检查各个网络接口上的网络传输。

防火墙是目前最重要的一种网络防护设备，从专业角度来说，防火墙是位于两个(或多个)网络间，实行网络间访问或控制的一组组件集合之硬件或软件。

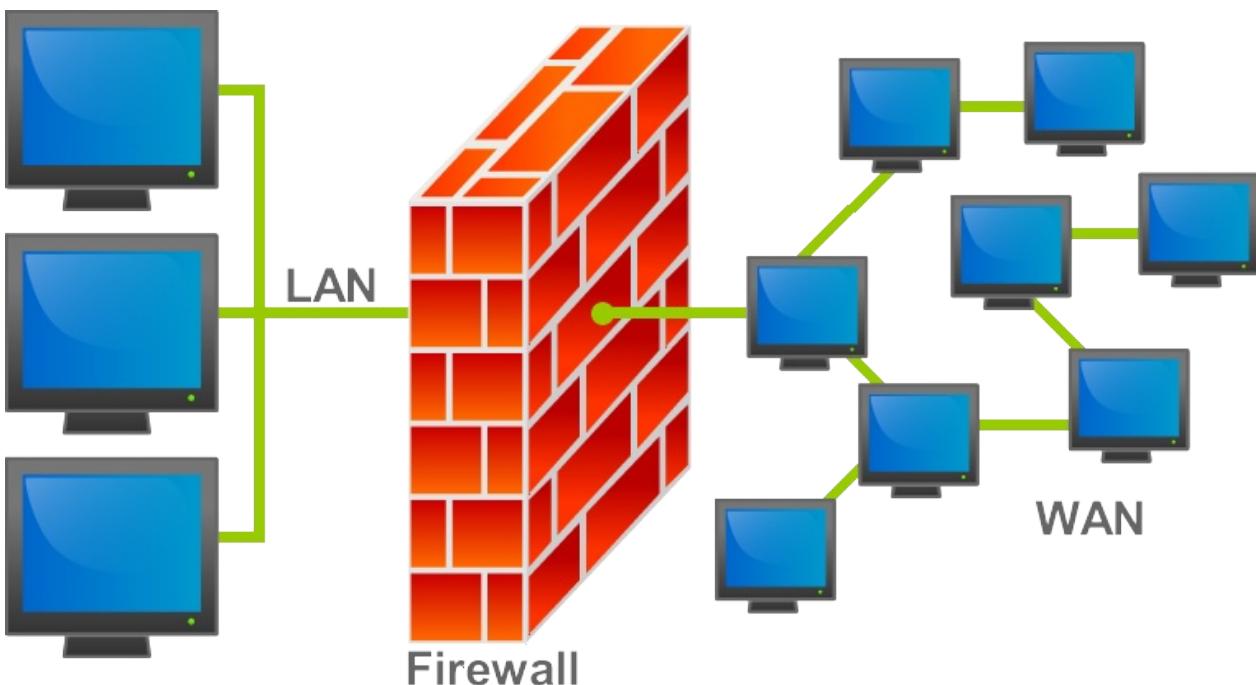


## 功能

防火墙最基本的功能就是隔离网络，通过将网络划分成不同的区域（通常情况下称为 **ZONE**），制定出不同区域之间的访问控制策略来控制不同信任程度区域间传送的数据流。以避免安全策略中禁止的一些通信。它有控制信息基本的任务在不同信任的区域。

例如互联网是不可信任的区域，而内部网络是高度信任的区域。

典型信任的区域包括互联网(一个没有信任的区域)和一个内部网络(一个高信任的区域)。



根据不同的需要，防火墙的功能有比较大的差异，但是一般都包含以下三种基本功能：

- 可以限制未授权的用户进入内部网络，过滤掉不安全的服务和非法用户
- 防止入侵者解决网络防御设施

- 限制内部用户访问特殊站点

## 目标

防火墙具有三个目标：

- 从外部到内部和从内部到外部的所有流量都通过防火墙。
- 仅被授权的流量（由安全策略定义）允许通过。
- 防火墙自身免于渗透。

## 最终目标

总结来说，防火墙的最终目标是根据模型之间最少特权原则，提供受控连通性在不同水平的信任区域通过安全政策的运行和连通性。

例如，TCP/IP Port 135~139是Microsoft Windows的【网上邻居】所使用的。如果电脑有使用【网上邻居】的【共享文件夹】，又没使用任何防火墙相关的防护措施的话，就等于把自己的【共享文件夹】公开到Internet，供不特定的任何人有机会浏览目录内的文件。

## 局限性

没有万能的网络安全技术，防火墙也不例外。防火墙主要有以下三方面的局限。

- 防火墙不能防范网络内部的攻击。比如：防火墙无法禁止内部人员将敏感数据拷贝到自己的存储介质上
- 防火墙也不能防范哪些伪装成超级用户或声称新员工的黑客们劝说没有防范心理的用户公开其口令，并授予其临时的网络访问权限。
- 防火墙不能防止传送已感染病毒的软件或文件，不能期望防火墙对每一个文件进行扫描，查出潜在的病毒。

## 类型

### 代理防火墙

代理防火墙是一种早期的防火墙设备类型，它针对特定应用充当从一个网络到另一个网络的网关。

代理服务器可以通过阻止来自网络外部的直接连接来提供其他功能（例如内容缓存和安全性）。但是，这可能会影响设备的吞吐量以及它们所能支持的应用。

## 状态检测防火墙

目前，状态检测防火墙普遍被视为“传统”防火墙，这种防火墙根据状态、端口和协议来允许或阻止流量。

它负责监控从连接打开到连接关闭期间的所有活动。过滤决策以管理员定义的规则以及情景为依据（情景是指以前的连接以及属于该连接的数据包的使用信息）。

## 统一威胁管理 (UTM) 防火墙

UTM 设备通常以一种松散耦合的方式，将状态检测防火墙的功能与入侵防御和防病毒功能结合到一起。

这类防火墙也可能包含其他服务，而且通常会包含云管理功能。UTM 主要侧重于简单性和易用性。

## 下一代防火墙 (NGFW)

如今，防火墙经过演变，已不再单纯提供数据包过滤和状态检测功能。大多数公司都在部署下一代防火墙，以求阻止高级恶意软件攻击和应用层攻击等现代威胁。

根据 Gartner 公司的定义，下一代防火墙必须包括以下要素：

- 标准防火墙功能（如状态检测）
- 集成入侵防御功能
- 应用感知和控制功能（可查看和阻止存在风险的应用）
- 能够吸纳未来出现的信息源的升级途径
- 能够应对不断演变的安全威胁的方法

这些功能已逐渐被大多数公司奉为标准，但实际上，下一代防火墙还可以提供更多功能。

## 专注于威胁的下一代防火墙

这类防火墙不仅包含常规下一代防火墙的所有功能，而且具备高级威胁检测和补救能力。利用专注于威胁的下一代防火墙，您可以：

- 借助全面的情景感知能力，了解哪些资产面临最高的风险
- 借助能够动态设置策略并加强防御的智能安全自动化功能，快速应对各种攻击
- 通过关联网络时间和终端事件，更好地发现逃避检测的活动或可疑活动
- 借助在初始检测后仍持续监控可疑活动和可疑行为的追溯性安全功能，极大地缩短从发现威胁到清除威胁的时间
- 借助涵盖整个攻击过程的统一策略，实现轻松管理并降低复杂性

## 解释

- [访问控制](#)：在信息安全领域中，访问控制包含了认证、授权以及审核。

## 参考

- [维基百科-防火墙](#)
- [Cisco-防火墙是什么](#)
- [ConnnetOS-Firewall配置](#)

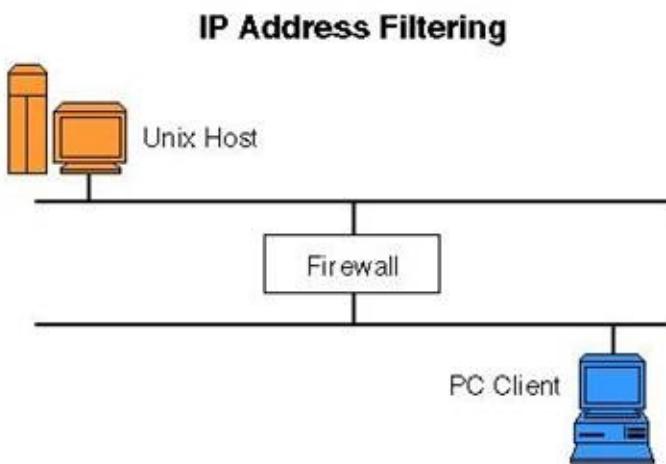
# 防火墙工作原理

防火墙的基本工作原理就是起到 **Filter** (过滤筛)的作用。你可以让你喜欢的东西通过这个过滤筛，别的东西统统过滤掉。

在网络的世界里，由防火墙过滤的就是承载通信数据的通信包。

## IP Address Filtering

所有的防火墙都有 IP 地址过滤功能。这个任务主要是检查 IP 包头，根据其IP源地址和目标地址做出 放行/丢弃 的决定。



当PC客户机向UNIX计算机发起telnet请求时，PC的telnet客户程序就产生一个TCP包并把它传给本地的协议栈准备发送。接下来，协议栈将这个TCP包“塞”到一个IP包里，然后通过PC机的TCP/IP栈所定义的路径将它发送给UNIX计算机。在这个例子里，这个IP包必须经过横在PC和UNIX计算机中的防火墙才能到达UNIX计算机

## 附录

- 防火墙的工作原理概述

## 防火墙的分类

传统防火墙总体上分为包过滤、应用级网关和代理服务器几大类型

### 数据包过滤

数据包过滤( **Packet Filtering** )技术是在网络层对数据包进行选择，选择的依据是系统内设置的过滤逻辑，被称为访问控制表( **Access Control Table** )。

通过检查数据流中每个数据包的源地址、目的地址、所用的端口号、协议状态等因素，或它们的组合来确定是否允许该数据包通过。

数据包过滤防火墙逻辑简单，价格便宜，易于安装和使用，网络性能和透明性好，它通常安装在路由器上。路由器是内部网络与Internet连接必不可少的设备，因此在原有网络上增加这样的防火墙几乎不需要任何额外的费用。

数据包过滤防火墙的缺点有二：

- 非法访问一旦突破防火墙，即可对主机上的软件和配置漏洞进行攻击；
- 数据包的源地址、目的地址以及IP的端口号都在数据包的头部，很有可能被窃听或假冒。

过滤规则考虑的因素如下：

- IP 源或目的地址
- 在 IP 数据包中协议类型字段： TCP 、 UDP 、 ICMP 、 OSPF 等
- TCP 或 UDP 的源和目的端口
- TCP 标识比特: SYN 、 ACK 等
- ICMP 报文类型
- 数据包离开和进入网络的不同规则
- 对不同路由器接口的不同规则

### 状态分组过滤器

状态分组过滤器会跟踪 TCP 连接，并使用这种知识做出过滤决定。

状态分组过滤器通过一张连接表来跟踪所有进行中的 TCP 连接来解决这个问题。

### 用于状态过滤器的连接表

源地址	目的地址	源端口	目的端口
222.22.1.7	37.96.87.123	12699	80
222.22.93.2	199.1.205.23	37654	80

## 用于专题过滤器的访问控制列表

动作	源地址	目的地址	协议	源端口	目的端口	标识比特	核对连接
允许	222.22、16	222.22/16的外部	TCP	>1023	80	任意	
允许	222.22/16的外部	222.22/16	TCP	80	>1023	ACK	X
允许	222.22/16	222.22/16的外部	UDP	>1023	53	-	
允许	222.22/16的外部	222.22/16	UDP	53	>1023	-	X
拒绝	全部	全部	全部	全部	全部	全部	

## 应用级网关

应用级网关( Application Level Gateways )是在网络应用层上建立协议过滤和转发功能。它针对特定的网络应用服务协议使用指定的数据过滤逻辑，并在过滤的同时，对数据包进行必要的分析、登记和统计，形成报告。实际中的应用网关通常安装在专用工作站系统上。数据包过滤和应用网关防火墙有一个共同特点，就是它们仅仅依靠特定的逻辑判定是否允许数据包通过。

一旦满足逻辑，则防火墙内外计算机系统建立直接联系，防火墙外部的用户便有可能直接了解防火墙内部网络结构和运行状态，这有利于实施非法访问和攻击。

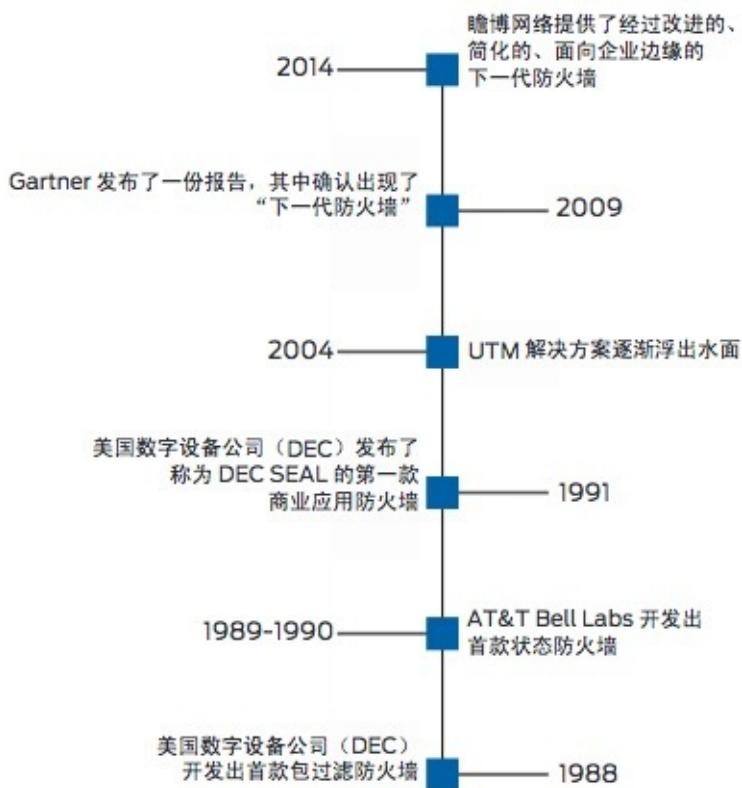
## 代理服务

代理服务( Proxy Service )也称链路级网关或 TCP 通道( Circuit Level Gateways or TCP Tunnels )，也有人将它归于应用级网关一类。

它是针对数据包过滤和应用网关技术存在的缺点而引入的防火墙技术，其特点是将所有跨越防火墙的网络通信链路分为两段。防火墙内外计算机系统间应用层的"链接"，由两个终止代理服务器上的"链接"来实现，外部计算机的网络链路只能到达代理服务器，从而起到了隔离防火墙内外计算机系统的作用。

此外，代理服务也对过往的数据包进行分析、注册登记，形成报告，同时当发现被攻击迹象时会向网络管理员发出警报，并保留攻击痕迹。

# 防火墙演进过程



防火墙现在可以为我们提供强大深厚的安全和高级功能，这是怎样的一个发展过程呢？这是一个很长的故事，但却发生在一段相对较短的时间内，很可能您也是这段故事的见证者。

二十世纪七十年代晚期，局域网( LAN )和连接系统面世，因此公司能让员工彼此通信，也能通过网络传送数据。

LAN 实现了局间进行电子通信，也就是电子邮件。对公司而言，该网络是本地的、私有且专属的，它支持有限数量的用户。 LAN 的安全原则形式是密码保护。

到了二十世纪八十年代初，推出了多协议路由器，用于互连不同的网络并定向这些网络之间的流量。有了路由器和其他 Internet 技术后，全球电子发展动向尽在计算机用户的掌握之中。路由器用来定向通过 Internet 的流量，还包括访问控制软件来保护业务传输和数据访问。

但是路由器无法提供必需的安全性，因此无法应对 Internet 连接带来的挑战。全球通信为黑客提供了入侵连接企业和个人电脑的机会，他们很快就会进入私人帐户。为找到这些黑客而执行的探测任务，迎来了计算机相关科学（称为数字取证）的发展。越来越多的安全问题推动了计算机网络安全的发展，促使产生了第一个网络防火墙。这些防火墙基于策略允许或阻

止流量。网络防火墙在用途上与物理防火墙相似，都是用来遏制攻击并防止它们蔓延。计算机网络防火墙在内部网络与外部网络之间竖起一道屏障，前者是在公司内部并被视为可信，后者被认为不可信，例如 Internet

## 第一代防火墙：包过滤防火墙

第一代防火墙是相对简单的过滤系统，被称为包过滤防火墙，但发展到今天，它们已成为高度复杂的计算机网络安全技术。

包过滤防火墙也称为无状态防火墙，基于过滤规则过滤出和丢弃流量。包过滤防火墙不保持连接状态。也就是说，将一个数据包作为一个原子单元处理，而不涉及相关的数据包。包过滤防火墙通常部署在路由器和交换机上。

包过滤的核心在于ACL的定义。而ACL的核心其实就是一个典型的五元组。通常在第3层和第4层执行数据包处理，方法就是对照五元组匹配数据包的标头字段：

- 源IP地址
- 目的IP地址
- 源端口
- 目的端口
- 协议号

不会将更高层的信息考虑在内。

包过滤防火墙的主要弱点是，黑客可以利用缺乏状态这一缺陷精心设计数据包以通过过滤。第一次使用包过滤防火墙时，操作系统堆栈易受攻击，单个数据包可能会导致系统崩溃，但在今天很少发生这种情况。

包过滤防火墙允许所有基于 web 的流量通过防火墙，包括基于 web 的攻击。这些防火墙无法区分有效的返回数据包和冒名顶替返回的数据包。如何处理这些及其他的问题，为防火墙的未来发展创造了条件。

(1988) 美国数字设备公司（DEC）开发出了首款无状态包过滤防火墙

## 第二代防火墙：状态防火墙

包过滤防火墙出现后不久就出现了状态防火墙。第二代防火墙具有与包过滤防火墙相同的功能，但它们可以监控和存储会话、连接状态。第二代防火墙基于源和目标IP地址、源和目标端口以及所用的协议将流量中的相关数据包关联起来。如果有数据包双向匹配这些信息，那么它就属于该流量。

随着 Internet 的广泛使用，企业逐渐高度网络化，他们可以有选择地为用户提供 Internet 服务，但前提是他们能够保护自己的资产免受从 LAN 外部发起的入侵和攻击。企业还想防止员工和其他企业人员（例如承包商及合作伙伴）尝试进入他们的网络，试图访问未经授权的网络资源。此外，他们还想保护自己的网络免受从 LAN 内部和整个 Internet 中发起的攻击，不管是有意还是无心为之，都要阻止。为了解决这些问题，企业转向部署状态防火墙。

无状态包过滤防火墙没有给管理员提供会话内和会话间的通信和连接状态所需的工具。状态防火墙解决了这个问题。

状态防火墙提供的安全性的核心在于根据会话连接决定是允许还是滤出流量。在它的状态表中，该防火墙维护所有允许的开放连接和会话的状态、源主机和目标主机之间的通信状态。这项技术为管理员提供了网络连接的智能视图，允许他们定义基于连接状态控制流量访问的规则。状态防火墙规则不仅包括四元组包过滤防火墙，还包括用来识别连接状态的第五元组。

状态防火墙解决了包过滤防火墙不能确定返回数据包是否来自合法连接的问题，但是其还不能区分**web**流量的安全与否。企业需要的是能够检测和阻止Web攻击的防火墙功能，这些功能随后不久也面世了

(1989-1990) AT&T Bell Laboratories 开发出第一款称作电路级网关的状态防火墙

## 有针对性的防火墙功能

随着安全问题日益突出而具体，与防火墙有关的功能强大的安全软件应运而生，解决了这些问题。

这些产品有防病毒( AV )应用程序、入侵防御系统( IPS )、 URL 内容过滤( URLF )和统一威胁管理应用程序。本节将介绍这些技术推动的网络安全性。

## 防病毒( AV )应用程序

了解防病毒软件有助于您了解计算机病毒。

早在计算机病毒出现之前，Jon von Neumann就在他的出版物“《Theory of Self-Reproducing Automata》”中假设可以设计自复制计算机程序。

虽然它并不是计算机病毒，而且也没有被用作病毒，但大家仍将 von Neumann 设计看作第一个计算机病毒。可以假设“《Theory of Self-Reproducing Automata》”和其他类似作品是计算机病毒设计者参考的文献。

计算机病毒属于自复制计算机程序，将自身嵌入到资源中，例如数据文件和其他计算机程序。它们可以在个别计算机和计算机网络上大肆破坏，干扰生产效率并造成数十亿美元的损失。

(1983) Peter Szor 将计算机病毒定义为“递归地复制可能是在自身基础上

据说，最具破坏性的病毒是“ ITW (In the Wild) ”。它们通常包含可以擦除所有计算文件的内容，以计算机的 BIOS 为目标。如果是“ ITW (In the Wild) ”，由于普通的日常操作，病毒肯定会在公共网络中被感染的计算机之间传播，无法遏制。在大约 50,000 种已知的计算机病毒中，只有不到 600 种已被确认为是“ ITW (in the wild) ”。

(1971) Creeper 病毒是确定的第一种计算机病毒。它感染大型机，最终被名为 Reaper 的第一款计算机防病毒软删除，Reaper 本身也是专门用来删除 Creeper 的病毒。

据说，第一种病毒是由朋友间交换的软盘等介质传播的。共享软件和盗版软件在人们中间传来传去。

电子邮件加快了计算机病毒的传播。病毒通常是通过即时消息和在 Internet 下载期间传播的。计算机病毒的发展策略依靠对安全漏洞的利用，变得越来越复杂，最终涉及利用社会工程。

今天，计算机病毒制造者经常使用脚本语言来创建利用社交媒体网站的宏病毒。

计算机病毒出现后，开发出了旨在检测和删除这些病毒的防病毒软件。防病毒程序对照已知病毒列表扫描可执行文件和引导块，尝试确定它们是否被感染，如果感染的话，就消除病毒和其他恶意软件。

防病毒软件也包含了动态功能，可用来检查 Internet 下载和不断扫描已知病毒类的活动。

防病毒程序依靠签名来检测病毒。签名是从唯一标识特定病毒的文本字符串中派生的算法或哈希数值。

第一个防病毒签名是代表特定恶意软件的整个文件的哈希值或字节序列。

渐渐地，先进的启发式技术开始发挥作用，该技术在它们的分析中使用可疑的节名称、不正确的标头大小、通配符和正则表达式。

为了与恶意软件的激增速度保持同步，防病毒软件的开发人员使用越来越复杂的算法。大多数的防病毒软件提供定期签名更新服务器，让客户实时了解最新的病毒。

(1987) 弗雷德·科恩提出，“没有任何算法可以完美地检测出所有可能的计算机病毒。”启发式防病毒实用程序出现：Ross Greenberg 的 FluShot Plus 和 Erwin Lanting 的 Anti4us 是最先出现的实用程序。(2013) IPS 强势复出。NSS Labs 研究总监 Rob Ayoub 预测 IPS 的回归时说：“……不要以为 IPS 技术翻身之后就会垮台。”

防病毒程序必须能够保护日益增多的文件类型，同时病毒检查程序必须更频繁地更新，这样才能持续有效。出现的防病毒程序可以防止、检测和删除病毒以及其他形式的恶意软件，例如 Rootkit 、劫持软件、木马、后门程序、拨号程序、造假工具、广告软件和间谍软件。

## 入侵防御系统

并非所有的网络攻击都是可以预防的，但成熟有效的预防系统（例如入侵防御系统）能够大大降低潜在伤害，安全专家称大约降低 90% 以上。这就是一些安全专家对入侵防御系统在 21 世纪初就要过时感到意外的原因。

结合使用传感器与分析器，依赖包括利用文件签名在内的各种手段，入侵防御系统监控网络流量并执行启发式流量分析。它们监控已视为知名攻击的利用签名的情况。它们还查找针对底层系统漏洞的漏洞签名。

入侵防御系统（IPS）的其中一个主要特征是即时性。识别潜在威胁后，IPS 可立即采取先发制人的措施，通常是在发动攻击之前应对威胁。它采取的措施由一组规则或策略决定，并由管理员基于组织的网络基础架构需求设置。这些规则可能会指示 IPS 触发警报以通知潜在威胁，也可能会指示 IPS 暂时或永久阻止流量（IP 源地址）来修复恶意流量。虽然 IPS 产品没有正式成为防火墙的一部分，但通常在防火墙后面排列成行，实现分层的安全保护。IPS 解决方案位于源和目标 IP 地址之间的直接通信路径中。

IPS 产品通常被称为设备，可在硬件或软件中实现，或者同时在硬件和软件中实现。它们保护从网络层直到应用层的主机免受已知和未知攻击。

它们是如何工作的？大多数的 IPS 设备使用以下三种检测方法中的一种或多种基于签名、基于统计异常和状态协议分析

- 基于签名的检测：监控知名攻击模式的流量数据包，依赖预先定义的签名数据库
- 基于资料或统计异常的检测：确定正常的网络活动。基于这些信息，IPS 检测异常行为或活动。异常检测技术并不一定针对恶意流量。
- 状态协议分析检测：其行为类似于基于签名的检测，但进行更深入的数据包检查。IPS 将观察到的事件和预先确定的良性行为资料进行对比，进而识别协议状态的偏差。

讨论 IPS 时，必须要考虑入侵检测系统（IDS）软件。IPS 有时被认为是下一代的 IDS，但这些技术在许多重要方面有所不同。IDS 应用程序属于被动监控系统，通常用来监控可疑活动和潜在入侵，它们警告管理员有此类入侵正在发生。IDS 仅告知潜在的攻击，由 IPS 阻止，或许还会发出警报。

## URLF 内容过滤

通过 URLF（URL 内容过滤）软件，用户可以控制到 Internet 网站的访问。通过阻止或允许从整个类别的网站到单个 URL 的任何链接，您可以控制员工、家庭成员及其他用户有权访问的网站。

## 统一威胁管理

虽然统一威胁管理( UTM )解决方案并不标志着防火墙演进的正式阶段，但它们汇集在一起，组成一套集成在一个单一平台中的安全服务。除了 AV 、 VPN 、内容过滤、负载均衡、报告等标准的防火墙功能外，它们还包含其他各种功能。它们提供了深度防御的安全性，但性能仍然是大型企业面临的一个问题。

(2004) 多种解决方案的出现，阻止了黑客、病毒和蠕虫攻击企业的机密数据系统。

## 第三代防火墙：应用防火墙

防火墙技术不断发展，快速推进；由于攻击者想法设法地寻找 OSI 模型层中的漏洞，防火墙技术发展也推动了 OSI 模型的发展，要求利用更高的层提供更多的流量和访问保护，以及对尝试性攻击的可视性。

(1991) DEC 基于 Marcus Ranum 的研究和设计发布了名为 DEC SEAL 的首款商业应用防火墙。

传统状态防火墙基于对协议和端口的控制提供保护，限制到达和来自特定 IP 地址的流量，被认为不足以抵御数量和种类均不断增长的基于 Web 的攻击。随着 Internet 日渐成熟，此类攻击也越来越普遍。

由于基于协议和端口的防火墙无法区分依赖这些协议和端口的合法应用程序与非法应用程序和攻击，基于Web的攻击很容易通过众所周知的端口： HTTP (端口 80 ) 、 HTTPS (端口 443 ) 和电子邮件 (端口 25 ) 。它们无法区分使用同一个端口的不同种类的 Web 流量。

应用防火墙解决了任何应用程序都可以在任何端口上运行这种情况产生的问题。它们能让管理员控制并保护直到到应用层的网络。它们识别应用程序试图突破或绕过允许或开放端口上的防火墙的情况。

借助应用防火墙，管理员可以识别并阻止可能存在恶意的应用程序和服务调用。

(1993) 根据 DARPA 合同，Marcus Ranum 、 Wei Xu 和 Peter Churchyard 共同开发出首款可免费使用的防火墙 Firewall Toolkit (FWTK) ，为其他产品提供了通用的构建基础。

由于大部分的应用程序在端口 80 上运行，因此能够区分不同类型的程序的安全软件是必不可少的。基于 Web 的攻击可以轻易通过已知端口： HTTP (端口 80 ) 、 HTTPS (端口 443 ) 和电子邮件 (端口 25 ) 。基于协议和端口的包过滤和状态防火墙，无法区分依赖这些协议和端口的合法应用程序与非法应用程序和攻击。

(1999) Perfecto Software 的 AppShield 率先提供 Web 应用防火墙 (WAF) 技术。

应用防火墙可以检测流量内容并阻止特定内容，例如 Web 服务和已知病毒。它们解决了许多与基于 Web 的攻击相关的问题。

基于主机的应用防火墙可以监控应用程序的输入、输出以及与应用程序相关的系统服务调用。**Web 应用防火墙( WAF )**是作为设备或服务器插件提供的，通过对 **HTTP** 会话应用一组规则来过滤流量。

**Web 应用防火墙**继续发展，再加上防火墙技术不断取得突破性的进步，不仅可以提供数据包及与其关联的应用程序的相关信息，还提供了对数据包内容本身的可视性。

## 下一代防火墙

2003年，Gartner 开始研究下一代防火墙(**NGFW**)的概念，并于 2004 年开始发布相关的注意事项。

2009 年，该公司在意识到企业 **NGFW** 的早期版本开始出现之后，发布了定义下一代防火墙的正式报告。**NGFW** 将许多现有和新开发的安全技术汇聚在一个合成结构中。应用可视性与可控性、深度包检测、高级威胁保护和服务质量共同构成了 **NGFW** 环境，实现了无中断、联机的网络嵌入式 (**bump-in-the-wire**) 配置、性能和管理改进。

**NGFW** 提供的服务产品组合包括一些传统的防火墙服务，但没有感知性能问题。这些解决方案还支持基于用户、用户组和用户角色的功能，将用户身份与分配给其系统的 IP 地址分别显示。除了解决剩余的一些网络安全问题，**NGFW** 还确保管理员能够将安全性落实到位，以便跟上计算机技术的飞跃发展和不断变化的 Internet 形势，进而让用户选择使用不受限制的计算机（系统不连接 USB 设备），支持设备发现技术和无线配置，提供自助服务门户并允许用户在网络中使用“自带设备”(**BYOD**)。

**NGFW** 既融合了上一代防火墙的优势，又扩大和深化了侦察和控制能力，无需牺牲性能。它们的深度包检测功能确保系统识别出尝试性攻击并采取补救措施。这些功能可以严格检查流量以确定它是否会引发攻击。

瞻博网络 **NGFW** 解决方案包括：

- 高性能深度（或全部）包检测 (**DPI**)，对产生有关特定流量（可用于规范化标准通信）的大量信息的数据包进行全面检查。它还提供了更快和更有效的异常检测。**DPI** 系统收集的数据为管理员提供了网络流量的全面视图。
- 端到端的网络访问控制 (**NAC**) 和应用感知，用于维护应用程序状态和资源需求信息。应用感知功能，支持智能流量分析，使管理员能够阻止旨在破坏可用性的分布式拒绝服务攻击(**DDOS**)和**WEB**应用程序执行的数据泄露。
- **Active Directory (AD)** 集成和用户身份管理。用户防火墙通过增加一个用户身份元组拓宽了安全策略的应用范围。可以根据用户的角色（基于用户角色的防火墙）、他们所属的组或个人用户标识（用户防火墙）来识别他们。
- 防止数据泄露或数据丢失的权威攻击分析保护，有助于在全球共享有关黑客的情报，从而加快检测和监控。

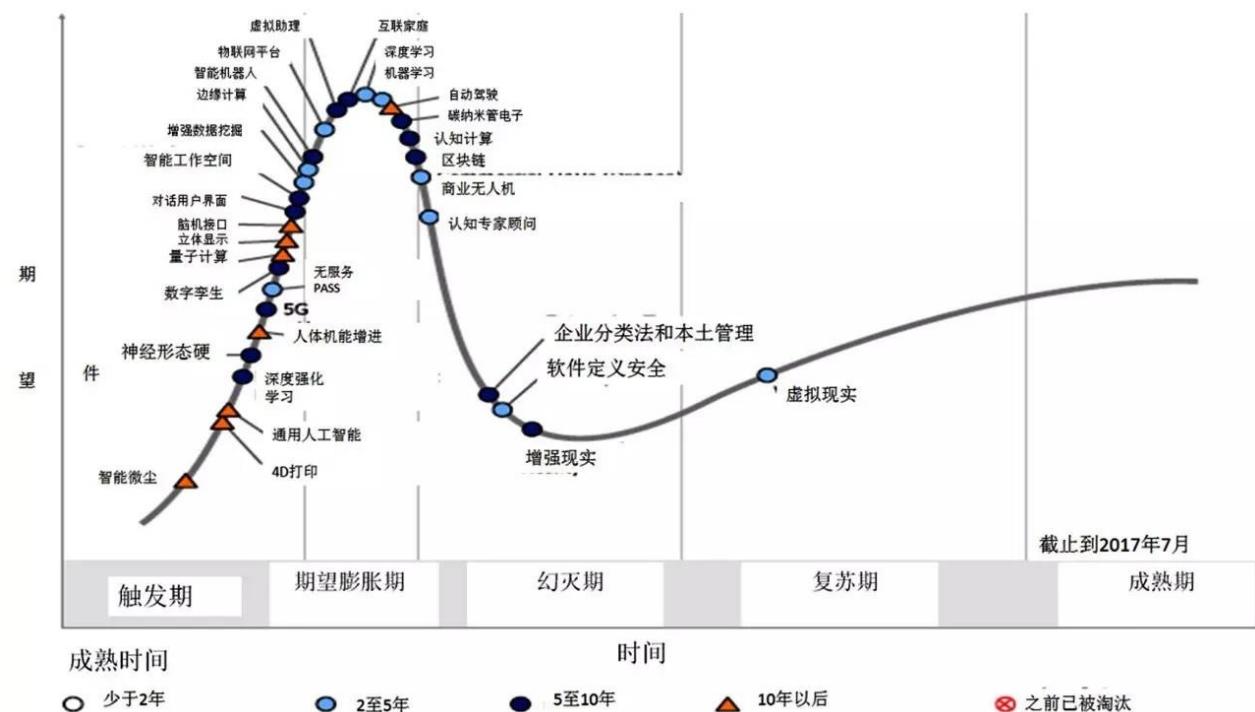


## 参考

- 从包过滤防火墙到下一代防火墙的演进过程
- 了解防火墙设计
- 下一代防火墙的几个思考
- 业内下一代防火墙 (NGFW) 对比
- 下一代防火墙 (NGFW) 与UTM有什么本质上的区别？
- 下一代防火墙产生的必然性
- 【企业研究报告】新一代防火墙的撕X大戏：Palo Alto Networks安全公司全解读

# SDN

2017年7月，Gartner公司发布了年度新兴技术成熟度曲线。Gartner认为，2017年技术成熟度曲线揭示了未来5-10年的三方面技术趋势。



## 参考

1. DNA开发者最佳实践
2. ODL深度开发

# YANG Language

## 参考

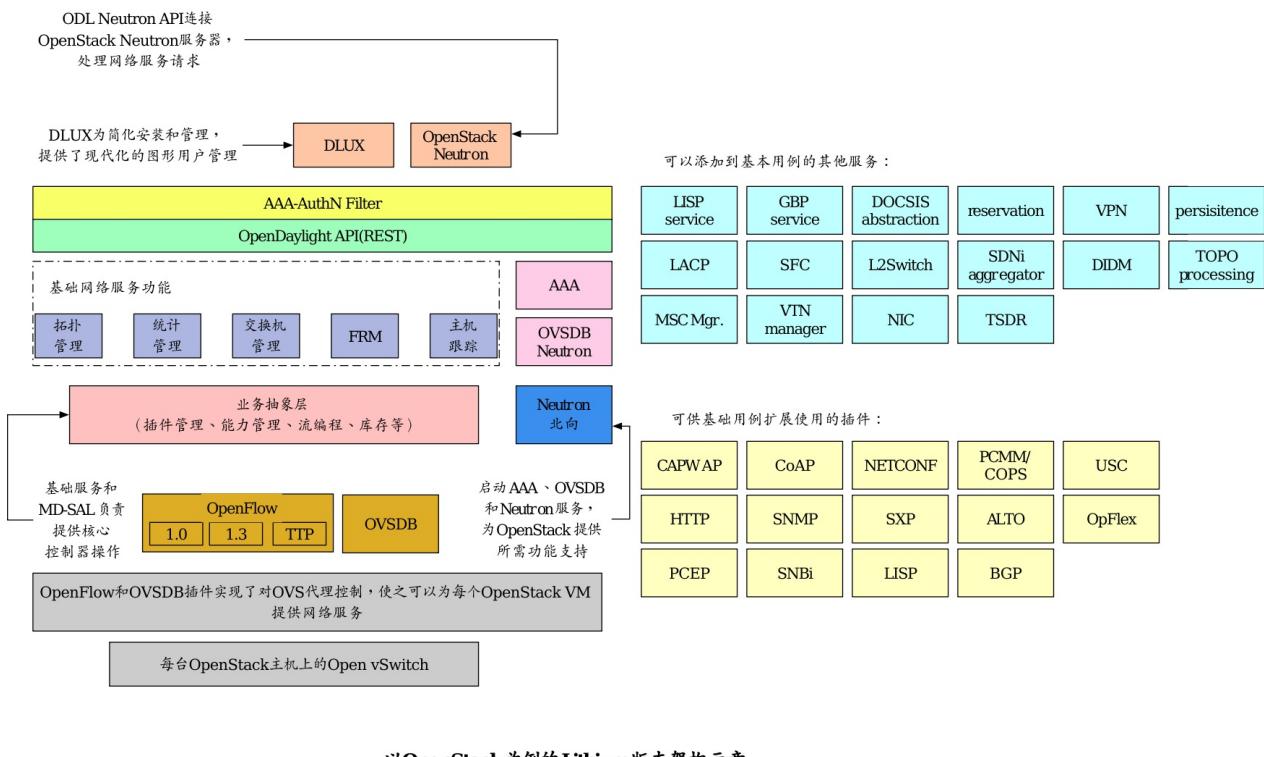
1. RFC6020 YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)
2. RFC6020 - YANG语言标准中文
3. ODL中Yang模型的通知(Notification)简介及报文接收方法
4. yang模型和openflow南北向接口

## SDN控制器

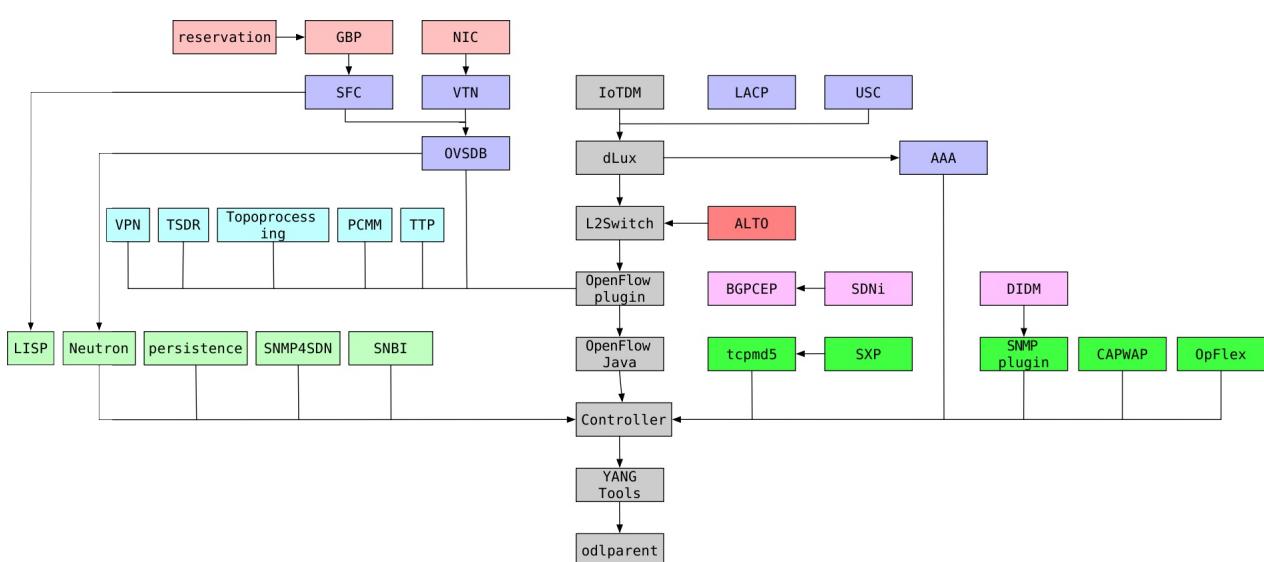
待补充

# OpenDaylight

# OpenDaylight Lithium 架构



## OpenDaylight项目相关性



OpenDaylight 版本项目相关性



# OpenDaylight Projects介绍

## Controller(控制器)项目

项目地址：[opendaylight/controller](#)

为多厂商网络的SDN部署提供一个高可用、模块化、可以扩展并可支持多协议的控制器基础框架。

在该项目中，模型驱动的业务抽象层使控制器支持多个南向协议插件，而面向应用的可扩展北向架构为控制器提供丰富的北向API：  
针对松耦合应用的RESTful Web服务和针对协作应用的OSGi服务。

另外，基于控制器平台的OSGi框架为控制器带来了模块化、可扩展的特性，还能为OSGi模型和服务提供版本控制和生命周期管理。

## OpenDaylight Root Parent项目

项目地址：[opendaylight/odlparent](#)

OpenDaylight中所有项目的Maven配置基础，包含外部依赖、默认版本、依赖管理、插件管理、库信息等所有共同信息。

它能为参与版本发布的所有项目提供统一的设置，其他项目的配置只需要继承 `odlparent` 即可获得ODL的统一设置，这在很大程度上能够帮助简化项目配置。

## YANG Tools项目

项目地址：[opendaylight/yangtools](#)

旨在开发必须的工具和库的基础设置项目，它能为Java项目（基于JVM语言）和应用提供NETCONF 和 YANG 支持，在OpenDaylight中，使用 YANG 作为模型化语言的应用有控制器 MD-SDL 和 NETCONF/OFConfig 插件。

## AAA项目

项目地址：[opendaylight/aaa](#)

为用户开发身份认证、授权、计费等功能。

包括为用户提供适用于多种身份认证、授权、计费机制的通用模型，提供可插拔的机制为通用系统提供插件。

## BGP LS PCEP（BGPCEP）项目

项目地址：[opendaylight/bgpcep](#)

为控制器提供两种南向接口插件--作为L3拓扑信息来源BGP（包括作为BGP扩展的BGP-LS、BGP-FlowSpec）和为底层网络提供实例化路径的PCEP（Path Computation Element Protocol）。

## DLUX项目

项目地址：[opendaylight/dlux](#)

为控制器的使用者提供新的交互式Web UI应用，它选择了AngularJS作为主要的前端技术，希望能够通过图形化的用户界面提高用户体验。

## L2Switch项目

项目地址：[opendaylight/l2switch](#)

该项目将L2的具体处理代码分离出来，组成一个独立的项目，提供基本的L2交换机功能并创建一些可重用的服务。

如提供模块化的事件驱动的数据包处理程序(packet handler)、地址跟踪、最优路径计算、基本的生成树协议等。

## LISP Flow Mapping Service项目

项目地址：[opendaylight/lispflowmapping](#)

该项目提供LISP映射系统服务，包括LISP Map-Server和LISP Map-Resolver服务，负责提供和存储数据到数据屏幕节点和OpenDaylight应用的映射。

LISP (Locator ID Separation Protocol) 是一个提供灵活的映射和封装框架的技术，它可用于数据中心网络虚拟化、网络功能虚拟化等 overlay 网络应用之中。

映射数据包括是虚拟化节点可达的虚拟地址到物理网络地址的映射以及流量工程、负载均衡等各种各样的路由策略。

OpenDaylight应用和服务可使用北向REST API在LISP映射服务中定义映射和策略，数据平面设备通过南向LISP插件具备LISP控制协议的能力。

## Neutron Northbound项目

项目地址：[opendaylight/neutron](#)

一个像OpenStack网络管理项目Neutron提供北街项目的插件项目，是使OpenDaylight和OpenStack协同工作的重要项目。

它提供网络、子网、端口、负载均衡、VPN、安全策略等REST API，并随着OpenDaylight的发展，不断的增加。

## ODL SDNi App项目

项目地址：[opendaylight/sdninterfaceapp](#)

OpenDaylight SDN接口应用项目旨在通过开发软件定义网络接口应用(Software Defined Networking interface, SDNi)保证SDN控制器之间的通信，该应用可在Helium版本的OpenDaylight上部署。

## OpenFlow Protocol Library项目

项目地址：[opendaylight/openflowjava](#)

OpenFlow协议库将会实施OpenFlow v1.3及后续版本协议。

该库是控制器的OpenFlow南向插件的基础，支持第三方供应商的扩展。

## OpenDaylight OpenFlow Plugin项目

项目地址：[opendaylight/openflowplugin](#)

OpenFlow是SDN架构中实现控制层和转发层之间交互的厂商中里的标准通信接口。

该项目旨在开发一个支持OpenFlow规范的插件，该插件将实现OF 1.0、OF 1.3、OF 1.4及后续版本的支持和整合。

## Persistence Store Service项目

项目地址：[opendaylight/persistence](#)

本项目是一个数据库统一操作服务框架，该框架的作用是实现营业在查询时的查询逻辑持久性，主要应用在需要连接数据库查询数据的应用中，例如 AAA 项目和 TSDR(Time Series Data Repository) 项目。

框架将不同的数据库查询方式封装起来，暴露统一的接口给ODL应用使用，提供数据查询、增加、删除等数据操作功能，并且维护查询操作的完整性。

框架可以支持不同类型的数据库，关系型数据库如 MySQL，非关系型数据库如 MongoDB 或者内存数据库。

## **SNBI(Secure Network Bootstrapping Infrastructure)项目**

项目地址：[opendaylight/snbi](#)

提供安全、自动、集成的网络设备和控制器。

通常，在安全通信建立之前，操作人员必须在一系列网络设备之间执行手动密钥分发过程。而SNBI使用零接触的方法即可引导安装 IEEE 802.1 AR 证书的产品进行安全认证。

SNBI设备和控制器能够自动发现对方，获取制定的IP地址并建立安全的IP连接。

另外，这个发现过程可展示网络的物理拓扑，识别每个设备的类型（可能是一个常规的网络设备或控制器），并为每个设备指定域，设备类型和域信息可用于控制器联合流程的初始化。

SNBI还包括控制器和转发单元上创建的组件和功能，这些组件和功能可包括单个的网络单元服务，如性能分析、流量探测、流量传送等。

## **SNMP4SDN项目**

项目地址：[opendaylight/snmp4sdn](#)

提供一个SNMP南向插件实现OpenDaylight控制器对现有商用以太网交换机的控制，该插件可提供管理配置的能力。

该项目使SDN不在局限于OpenFlow，支持以太网交换机作为SDN网络的数据面设备，它主要经历以下三个阶段的演进：

1. 创建一个SNMP南向插件配置通过SNMP配置以太网交换机
2. 插件通过CLI (Command Line Interface，命令行接口) 对以太网交换机做一些SNMP不能放松的设定

### 3. 实现SAL API的扩展

## SNMP Plugin项目

项目地址：[opendaylight/snmp](#)

SNMP是一个实现网络管理的协议。

它可用来手机信息、配置IP网络设备，如交换机、路由器、打印机、服务器等。

该项目致力于南向插件的需求，使应用和控制器服务能够使用SNMP与网络设备实现交互。

SNMP南向插件使应用充当SNMP管理者与支持SNMP代理的设备的交互。

## SXP(Source-Group Tag eXchange Protocol)项目

项目地址：[opendaylight/sxp](#)

SXP是一个传送IP地址和源组标签(Source Group Tag, SGT)之间绑定信息的控制协议。

在SXP中，源组是一系列具有共同网络策略的连接网络的端节点。每个源组通过一个特殊的SGT值(16字节)标识(大多数思科设备都支持SGT标识)。

该项目旨在ODL中实现SXP，使ODL获取大型已安装思科设备的绑定信息，并提供个应用和网络单元。

当前实施支持SXP协议的版本4，并提供对1~3版本的支持。此外，作为单向连接的扩展，版本4还增加了双向连接类型。

## TCP-MD5项目

项目地址：[opendaylight/archived-tcpmd5](#)

本项目库为操作系统提供TCP-MD5([RFC2385](#))支持，可用于保护BGP会话和PCEP会话。

该项目定义了一个使用Java本地接口库(Java Native Interface, JNI)实现简单API，可用于设置与TCP channel关联的MD5密钥。

虽然相应的代码已经存在于BGPCEP项目中，但该项目旨在将这些代码分离为一个独立的组件，使其生命周期独立与BGPCEP项目，这将带来以下益处：

不同提交者带来的无限潜力、明确的问题阐述以及更为稳定的版本。

## Topology Protocol Framework项目

项目地址：[opendaylight/topoprocessing](#)

旨在创建拓扑聚合的框架，提供统一的拓扑视图。

该项目主要提供量大产品特点：拓扑聚合和拓扑过滤。

1. 拓扑聚合指从数据库中获取 `underlay` 拓扑信息，然后将其聚合为一个或多个 `overlay` 拓扑，这些拓扑基于 `router ID`、管理IP和 `datapath ID` 等映射关系创建，提供多协议节点抽象(多协议节点抽象是通过RPC转发实现的)的统一视图，使应用无需关注多个拓扑和多个节点标识。

这个框架还提供过滤拓扑视图，该过滤可应用到交换机、交换机组、特定链接和其他对象。

## Use Case



## ALTO(Application-Layer Traffic Optimization)项目

项目地址：[opendaylight/alto](#)

ALTO是一个为应用提供网络信息的IETF协议，定义映射成本服务、过滤映射服务、端节点属性服务、端节点服务成本等网络服务，从而引导应用使用网络资源。

该项目致力于在OpenDaylight中实现ALTO。为了实现ALTO基础协议([RFC7285](#))，该项目将在OpenDaylight中实现这些服务并通过北向API开放给他人使用。

## CAPWAP(Cotrol And Provisioning of Wireless Accesss Points)项目

项目地址：[opendaylight/capwap](#)

该项目是为了“有线和无线网络通过适当抽象实现统一管理”的长远目标而提出的。

现阶段它致力于提供CAPWAP协议作为了一个南向接口。该项目的作用范围包括：

1. 为CAPWAP提供南向MD-SAL插件
2. CAPWAP协议库
3. CAPWAP和IEEE 802.11之间的绑定关系

- 
- 4. CAPWAP和IEEE 802.11间绑定关系的标准测试应用
  - 5. 本地映射支持

## Controller Core Functionality Tutorials 项目

项目地址：[opendaylight/coretutorials](https://opendaylight.org/project/coretutorials)

该项目面向开发者提供各种基础功能的教程，以期开发者快速理解OpenDaylight的项目结构、基础功能，从而能加入到OpenDaylight社区开发。

这些教程涉及的工程有 Controller 、 YANG Tools 、 OpenFlow Java 以及 OpenFlow Plugin ，其中包含各种基础操作有 ConfigSubsystem 、 NETCONF 、 RESTCONF 、 flow 以及 MD-SAL 的相关操作。

该项目提供的各个教程均是使用标准的项目结构(例如 ArcheType )，并且每个教程均采用分步式方法介绍。

## Defense4All 项目

项目地址：[opendaylight/archived-defense4all](https://opendaylight.org/project/archived-defense4all)

一个检测和缓解DDoS攻击的SDN应用。

在本项目定义的系统中，应用通过ODC北向REST API与OpenDaylight控制器之间实行交互，主要指向以下两大功能：监控被保护流量的行为并将攻击流量转移到被选攻击缓解系统(Attack Mitigation System, AMS)。

该项目的实现方法如下：

应用通过OpenDaylight控制器想每个被保护网( Protected Network, PN )的合适位置下发流条目，采集和聚合流量并利用流条目的 counter 字段做流量统计，识别异常流量，然后再想被选网络位置下发流条目，将异常流量转移到被选 AMS 。

当攻击消失时，应用将移除相应的流条目，使流量回归到正常的路径。

## DIDM(Device Identification and Driver Management) 项目

项目地址：[opendaylight/didm](https://opendaylight.org/project/didm)

提供一个可用于通知控制器发现它所控制的新设备、标识设备类型、将设备驱动注册为路由类型的RPC(Remote Procedure Call, 远程过程调用)、搜集设备数据、定义库存模型、调用设备驱动的框架。

该项目使用SNMP协议与设备进行交互时需要进行安全认证，因此它需要使用SNMP南向协议插件项目、AAA项目(进行认证管理)中的一些组件。

## Documentation项目

项目地址：[opendaylight/docs](#)

提供OpenDaylight项目群的文档。

该项目计划建立内容基础框架，用来改善各个OpenDaylight项目的用户说明以及确保不断增长的内容的质量。

整个项目管理包括内容编辑、内容发布以及设置内容格式等。

## Group Based Policy(GBP)项目

项目地址：[opendaylight/groupbasedpolicy](#)

定义以应用为中心的策略模型，将应用的网络连接从底层网络抽象出来。

该工程提供了一个简单的自我记录的机制来获取策略，而不需要关心底层网络信息的框架，另外通过将网络端点分组，可以同时操作多个节点，提升了策略下发的自动化，同时也可方便的统一和简洁的处理策略的变化。

## Integration Group项目

项目地址：[opendaylight/integration-test](#) [opendaylight/integration-packaging](#)

[opendaylight/integration-distribution](#)

提供协调、促进集成以及持续集成测试的基础框架，以期能够发布成功的OpenDaylight版本。

该项目服务内容有：

1. OpenDaylight整体测试策略制定
2. 持续系统集成测试
3. 发布版本协调
4. 社区实验室测试协调

- 
- 5. 自动化测试框架
  - 6. 测试用例管理
  - 7. 实验环境搭建

## IoTDM(Internet of Things Data Management)项目

项目地址：[opendaylight/iotdm](https://opendaylight.org/project/iotdm)

提供以数据为中心的中间件，运行通过验证的应用访问设备上传的物联网数据。

该项目使用ODL平台模型化 oneM2M 的 DataStore(分层容器树) ，其中书上每个节点均表示 oneM2M 的物联网某个资源。

物联网设备与应用通过资源树进行交互，其中网络协议包含 CoAP(Constrained Application Protocol) 、 MQTT(Message Queuing Telemetry Transport, 消息队列遥测传输) 以及 HTTP 。

## LACP(Link Aggregation Control Protocol)项目

项目地址：[opendaylight/lacp](https://opendaylight.org/project/lacp)

一种实现链路动态汇聚的协议。

在带宽比较紧张的情况下，该协议可通过逻辑聚合将带宽扩展到原链路的 n 倍，还可通过配置链路聚合实现同一聚合组的各个端口之间彼此动态备份。

在OpenDaylight中，LACP项目使用MD-SAL的方式实现了LACP协议，可用于ODL控制网络或支持LACP的交换机/设备的自动发现和多链路聚合。

## NIC(Network Intent Composition)项目

项目地址：[opendaylight/nic](https://opendaylight.org/project/nic)

使控制器能够按照网络行为和网络策略定义的“意图”对网络服务的网络资源进行调度和管理。

这个意图通过一个新的北向接口(Northbound Interface，NBI)描述，北向接口提供一个普遍的抽象的策略语义，用于代替类似OpenFlow的流表规则。

不同于描述如何提供不同服务的SDN接口，基于NBI的“意图”允许通过描述性的方式从基础设备获取需要的资源。

NBI负责协调编排服务和面向网络和业务的SDN应用，包括OpenStack Neutron、SFC项目和GBP项目。

NIC将使用现有OpenDaylight网络服务功能和南向插件来控制虚拟和物理的网络设备。

NIC被设计Wie协议无关的，可以使用OpenFlow、OVSDB、I2RS、NETCONFIG、SNMP等控制协议。

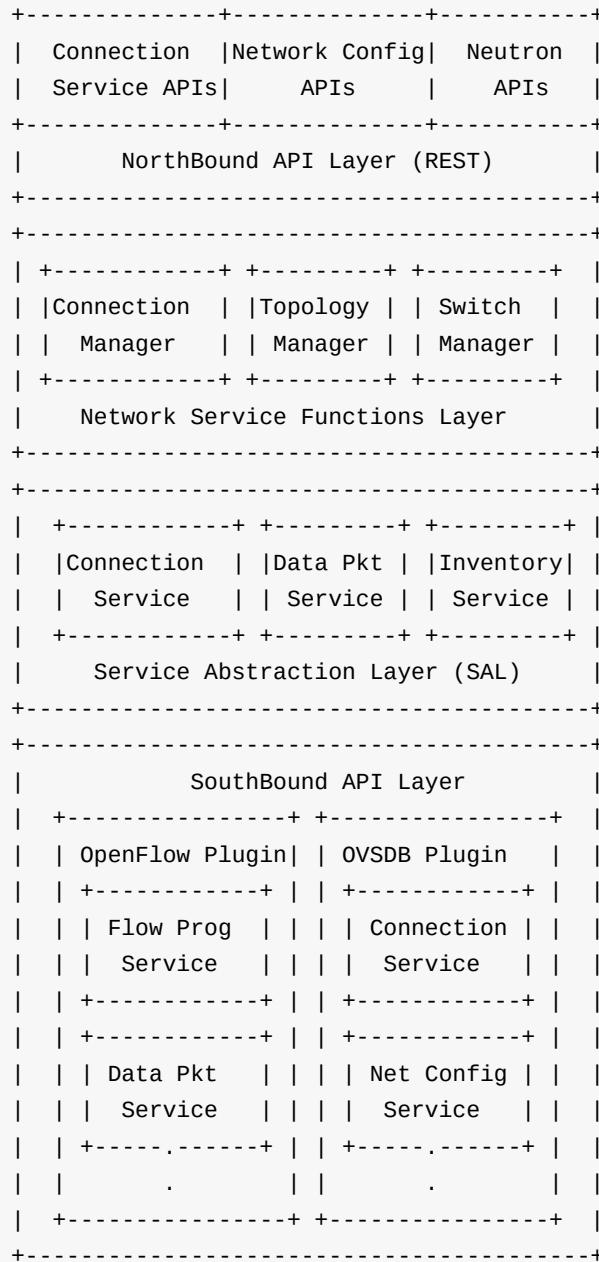
## OpenvSwitch Database Integration项目

项目地址：[opendaylight/ovsdb](#)

一个为OpenDaylight实施OpenvSwitch Database([RFC7047](#))管理协议的项目，可实现虚拟交换机的南向配置。

该项目包含一个OVSDB库及相关的各种插件的用法。

[OpenDaylight OVSDB Integration:Design](#)



## OpFlex项目

项目地址：[opendaylight/opflex](https://opendaylight/opflex)

主要提供相关OpFlex协议的实现，该协议基于策略模型实现一个分布式控制系统。

在OpFlex协议中，多数本地执行策略是由一个逻辑集中策略库提供。

项目主要分为三个部分：

1. `libopflex` 是一个在OpFlex协议框架下交互管理对象、解析策略、更新订阅的工具库
2. `genie` 是一个根据 `libopflex` 库中模型生成代码的框架
3. `agent-ovs` 是一个和OVS一起运行的策略代理，主要负责执行OpFlex的策略，这个策略

代理可以编排工具(如OpenStack)一同使用。

## PCMM(Packet Cable MultiMedia)项目

项目地址：[opendaylight/packetcable](#)

提供控制和管理 CMTS(Cable Modem Terminatin System, 线缆调制解调器终端系统) 网络单元服务流的接口。

PCMM架构包含以下组件：

1. 应用管理器，将QoS需求基于每应用发送给策略服务器。
2. 策略服务器，按每应用每用户分配网络资源，保证消耗量能满足(Multiple Service Operator)优先级。
3. CMTS，根据带宽容量执行策略。
4. 电缆调制解调器，位于客户端，负责帮助客户网络连接到电缆系统。

该项目的目标是利用OpenDaylight控制平台作为应用管理器和部分策略服务器，并最大程度的利用该平台提供的现有组件。

## Release Engineering-Autorelease项目

项目地址：[opendaylight/releng-autorelease](#)

该项目致力于对所有相关脚本进行版本管理和位置标记。

它负责创建、管理和提交OpenDaylight每个发布版本的相关标签，并制定按照设定的常规周期(如每天、每周、每月、里程碑时间)来生成候选版本。

## Reservation项目

项目地址：[opendaylight/reservation](#)

提供动态的低级别的资源预留，使用在指定时间段获得网络及服务、连通性、相应的资源池(端口、带宽)等。

## Service Function Chaining(SFC)项目

项目地址：[opendaylight/sfc](#)

定义有顺序的网络服务(如防火墙、负载均衡)链表的能力，这些网络服务按照一定顺序连接在一起称为业务链。

该项目为网络和终端用户应用提供定义业务链所需的基础框架(建逻辑链、API)。

## TTP(Table Type Patterns)项目

项目地址：[opendaylight/ttp](#)

ONF的转发抽象工作组(Forwarding Abstractions Working Group, FAWG)的第一个具体输出，其目标是使OpenFlow控制器和OpenFlow交换机对一系列功能进行协商，从而实现OF 1.1+版本的多样性管理。

TTP项目是在“数据路径协商模型”下提出的，当前主要聚焦于OpenFlow。

## TSDR(Time Series Data Repository)项目

项目地址：[opendaylight/tsdr](#)

目的是创建一个可伸缩、可扩展的时间序列采集数据的持久化框架，其中采集的数据保护 `DataStore` 的统计数据以及消息总线的消息。

该项目保护一个时间序列数据存储库以及一组时间序列数据MD-SAL服务(包括采集、存储、查询以及维护时间序列数据等服务)，并提供访问TDSR数据的北向接口。

## Unified Secure Channel项目

项目地址：[opendaylight/usc](#)

在企业网中，越来越多的控制器和网络管理系统正在远程部署(如在云端部署)，除此之外，企业网页越来越多样化，如分支、物联网、无线等。

因此，企业客户需要一个收敛的网络控制器和管理系统解决方案，该项目在网络单元和控制器之间建立一个统一的安全通信隧道：创建安全通道和支持多种通信协议的通用机制。

## VPN Service项目

项目地址：[opendaylight/vpnservice](#)

目标是提供建立基于BGP-MPLS([RFC4364](#))的L3 VPN服务所需的基础设施，以后的版本将提供基于EVPN的L2 VPN服务。

该项目与OVSDDB Integration项目、BGP-LS、SDNi的相关组件具有一定的依赖关系。

## VTN(Virtual Tenant Network)项目

项目地址：[opendaylight/vtn](#)

为用户提供多租户级别的虚拟网络。

在传统网络中，如果要为企业用户配置和部署某个部门或某个应用系统的网络，需要对每个租户进行安装和配置，而且不同租户间的网络是不可共享的。而VTN将网络分割为逻辑平面和物力平面，将逻辑平面提供给用户，用户可以自己设计和部署网络，不需要了解物力网络的拓扑结构和带宽限制。

VTN允许用户定义类似传统的二层和三层网络，并自动将设计的网络映射到物理网络上。逻辑平面的定义不仅可以隐藏底层网络的复杂性，同时也能更好的管理网络资源。

这不仅可以减少配置网络服务的时间，还能减少网络配置错误。

# OpenDaylight DataStore

## 参考

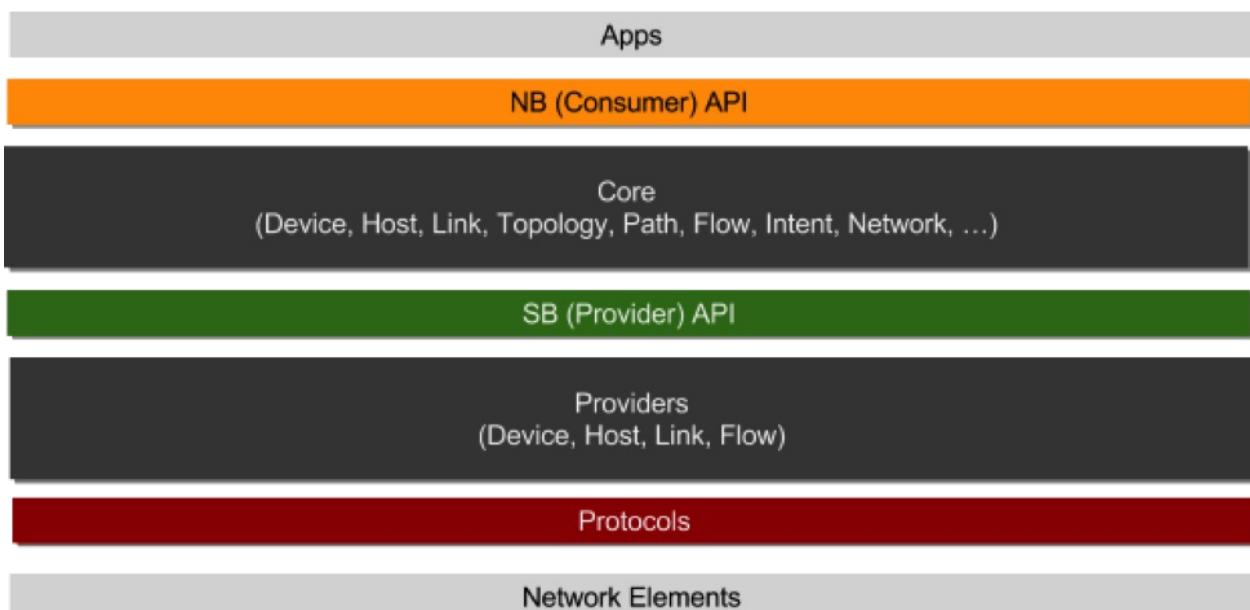
1. [OpenDaylight DataStore分析](#)
2. [OpenDaylight中DataStore Tree Notification示例](#)

# 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

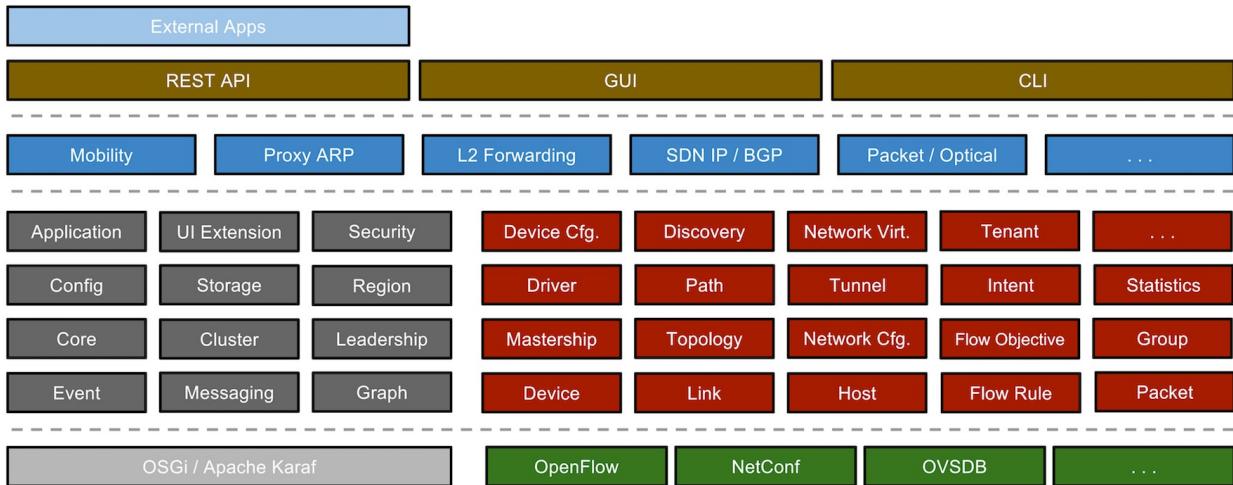
### 系统层次



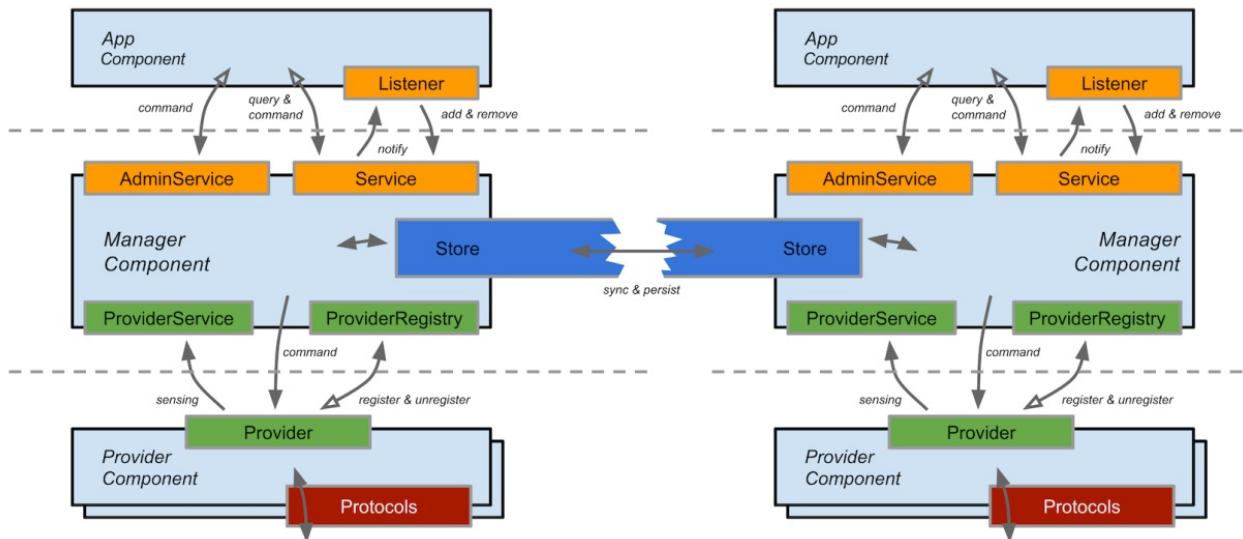
### 组件和服务

- **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

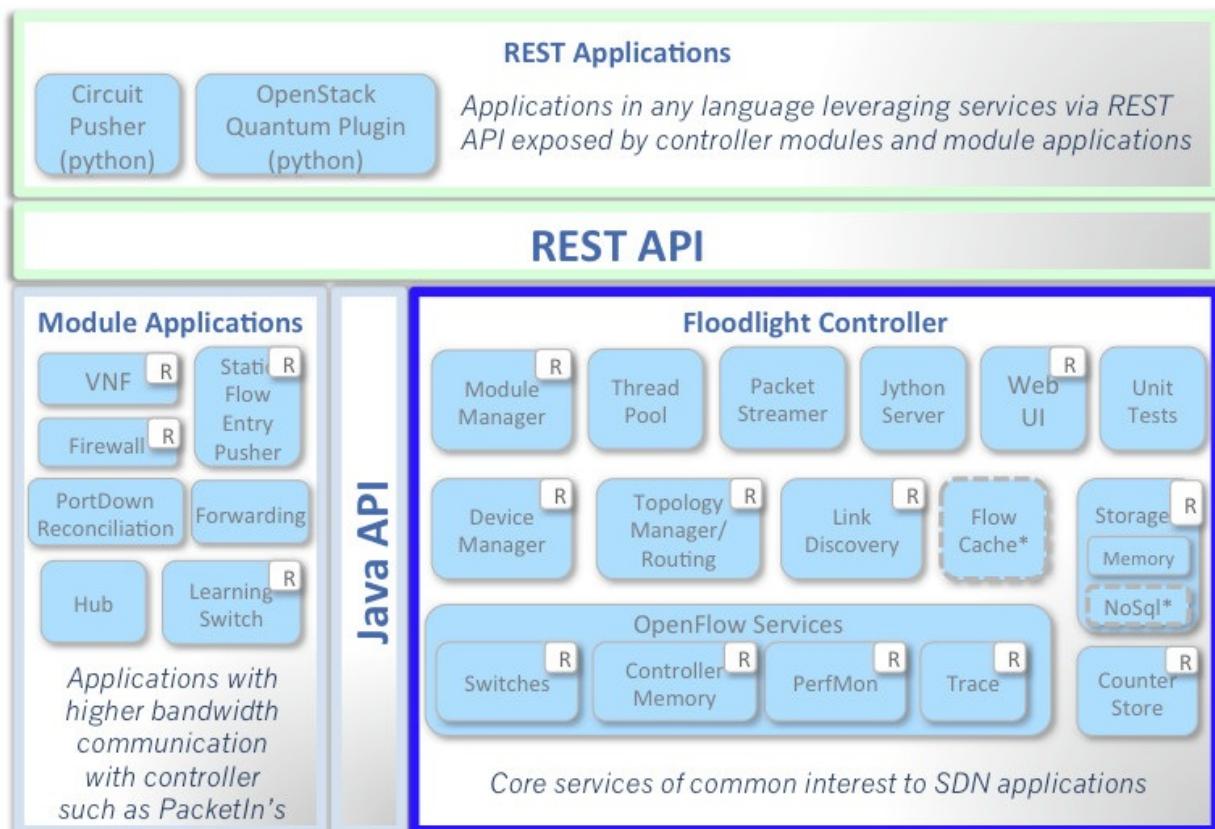
- ONOS Website
- ONOS Wiki
- ONOS架构分析
- ONOS白皮书上篇之ONOS简介
- ONOS白皮书中篇之ONOS架构

# 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](#)

# 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。

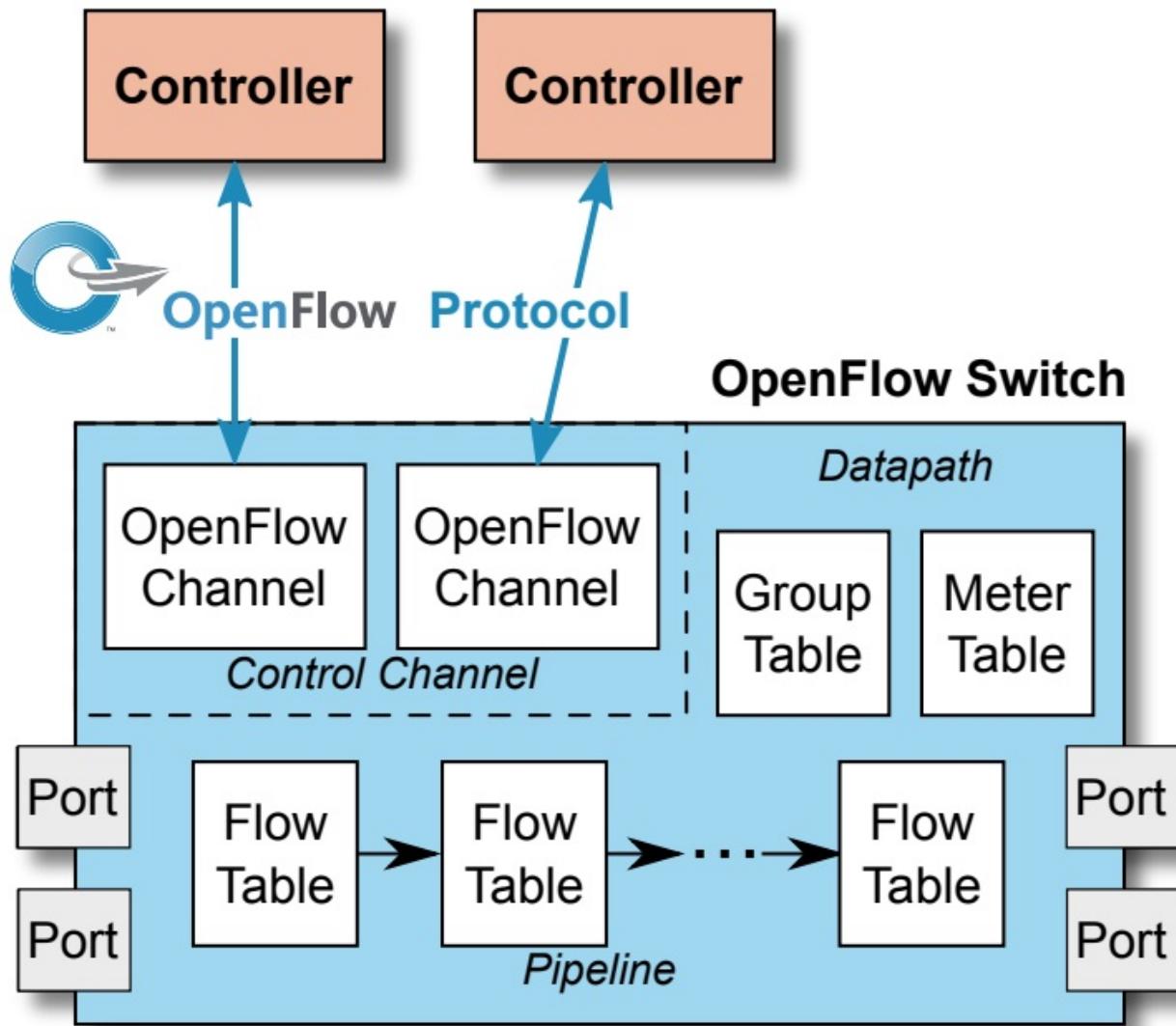
OpenFlow起源于Nick McKeown等在2008年发表的[OpenFlow: enabling innovation in campus networks](#)论文，并在次年发布了1.0版本协议。2011年又成立了Open Networking Foundation (ONF)进一步规范和推动OpenFlow的发展，并将OpenFlow的协议规范发布在[ONF网站](#)。

## 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

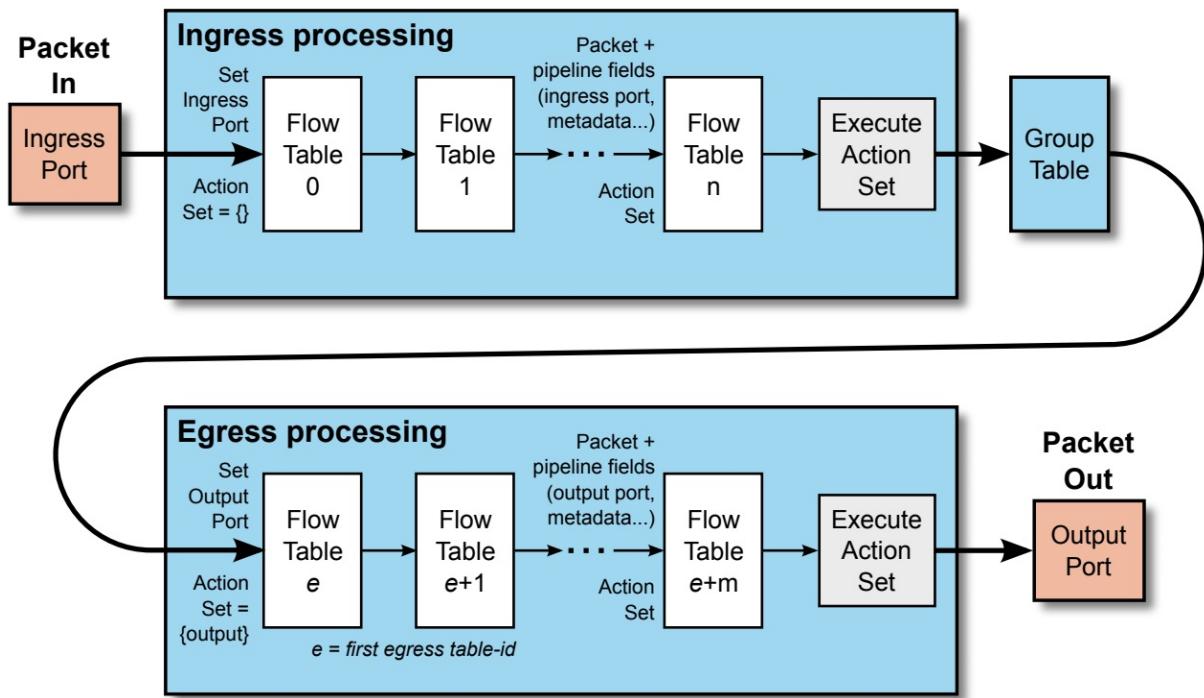


它主要由OpenFlow通道和数据平面组成，而数据平面又包括流表、端口、组表和Meter表等：

- 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

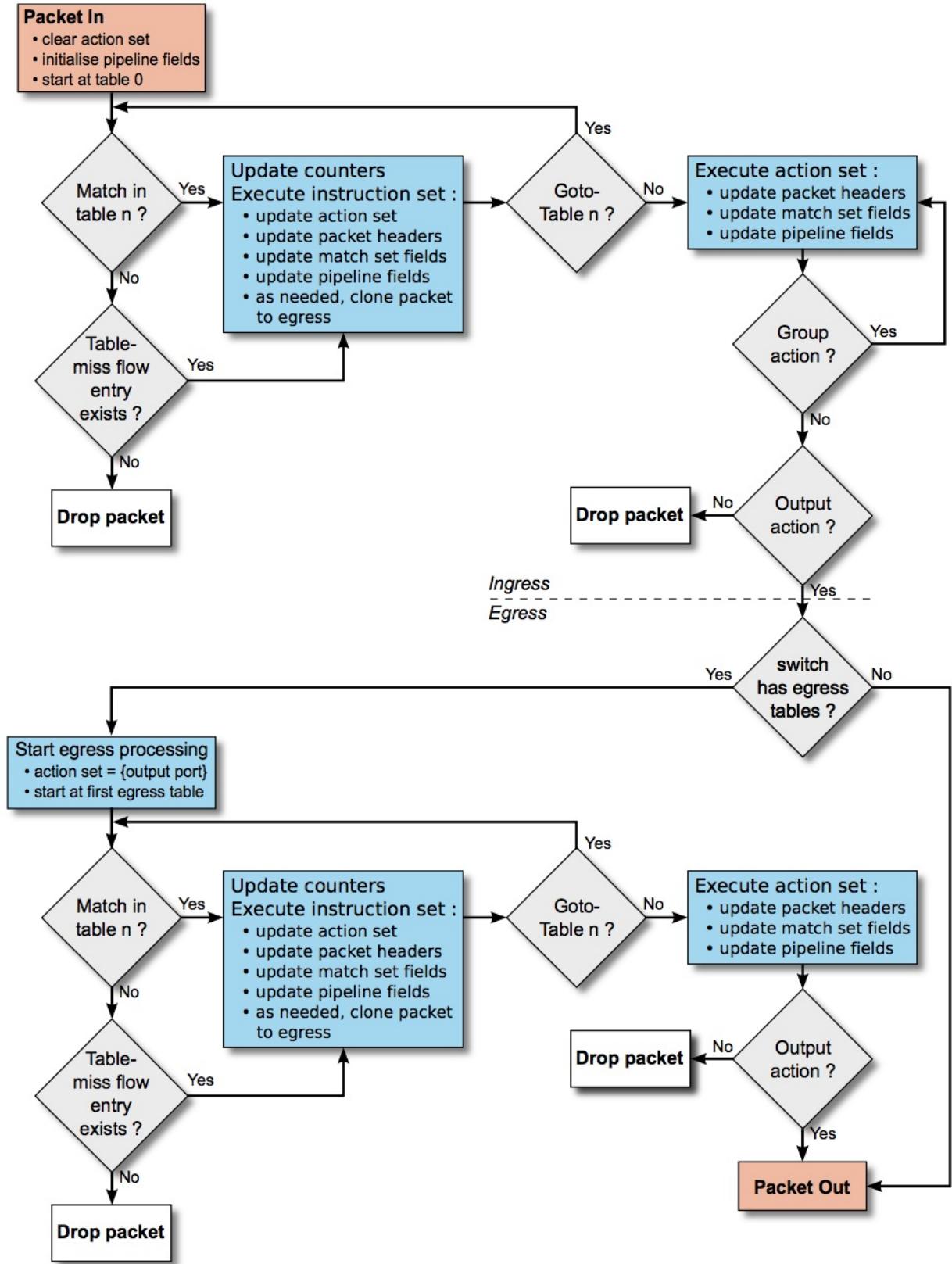
流表用于存储流表项，多级流表以流水线的方式处理。



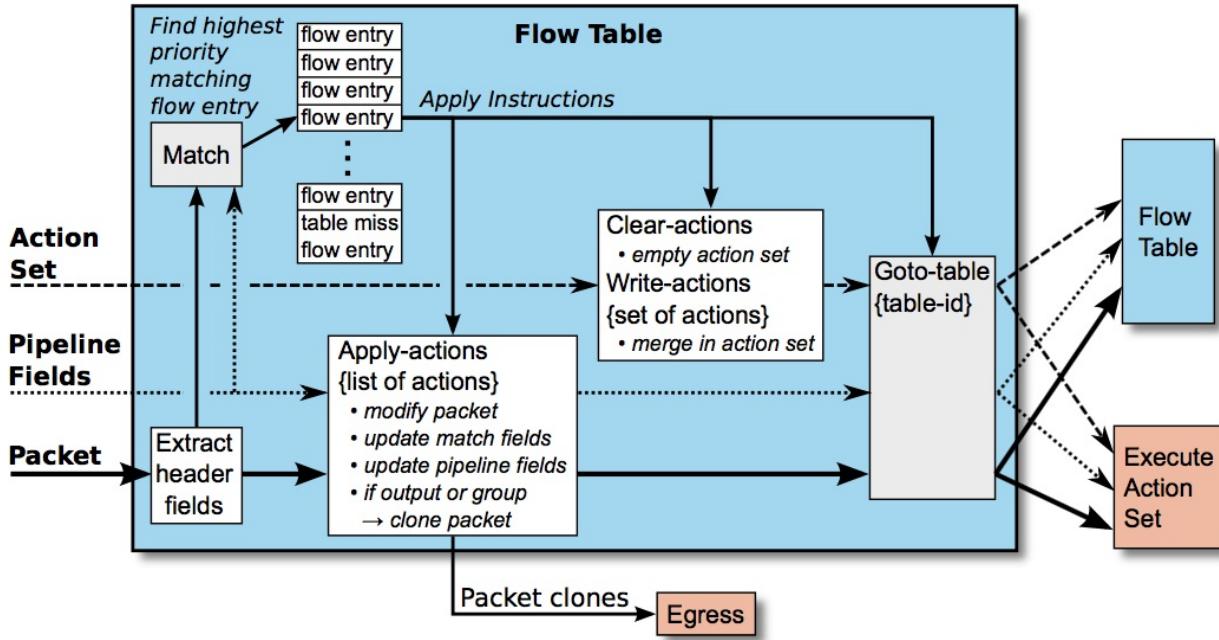
每个流表项由匹配域（包括输入端口、包头以及其他流表设置的元数据）、优先级、指令集、计数器、计时器、Cookie和用于管理流表项的flag组成：

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags
--------------	----------	----------	--------------	----------	--------	-------

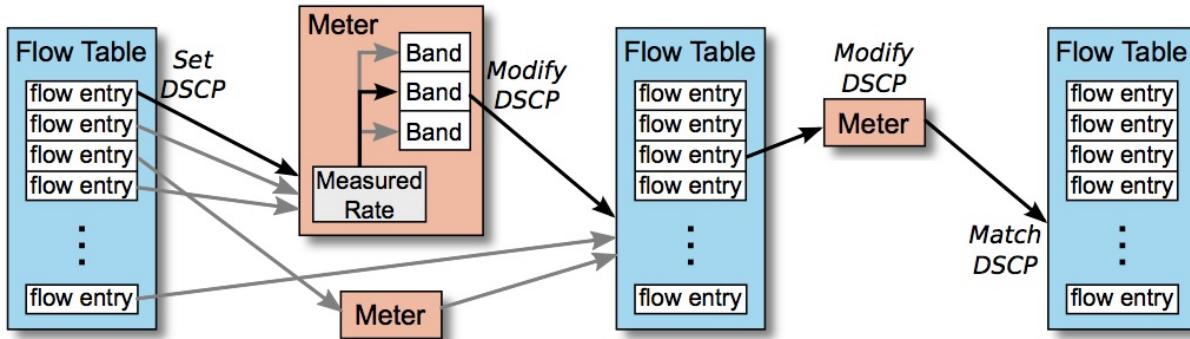
一个典型的流表匹配过程如下所示



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

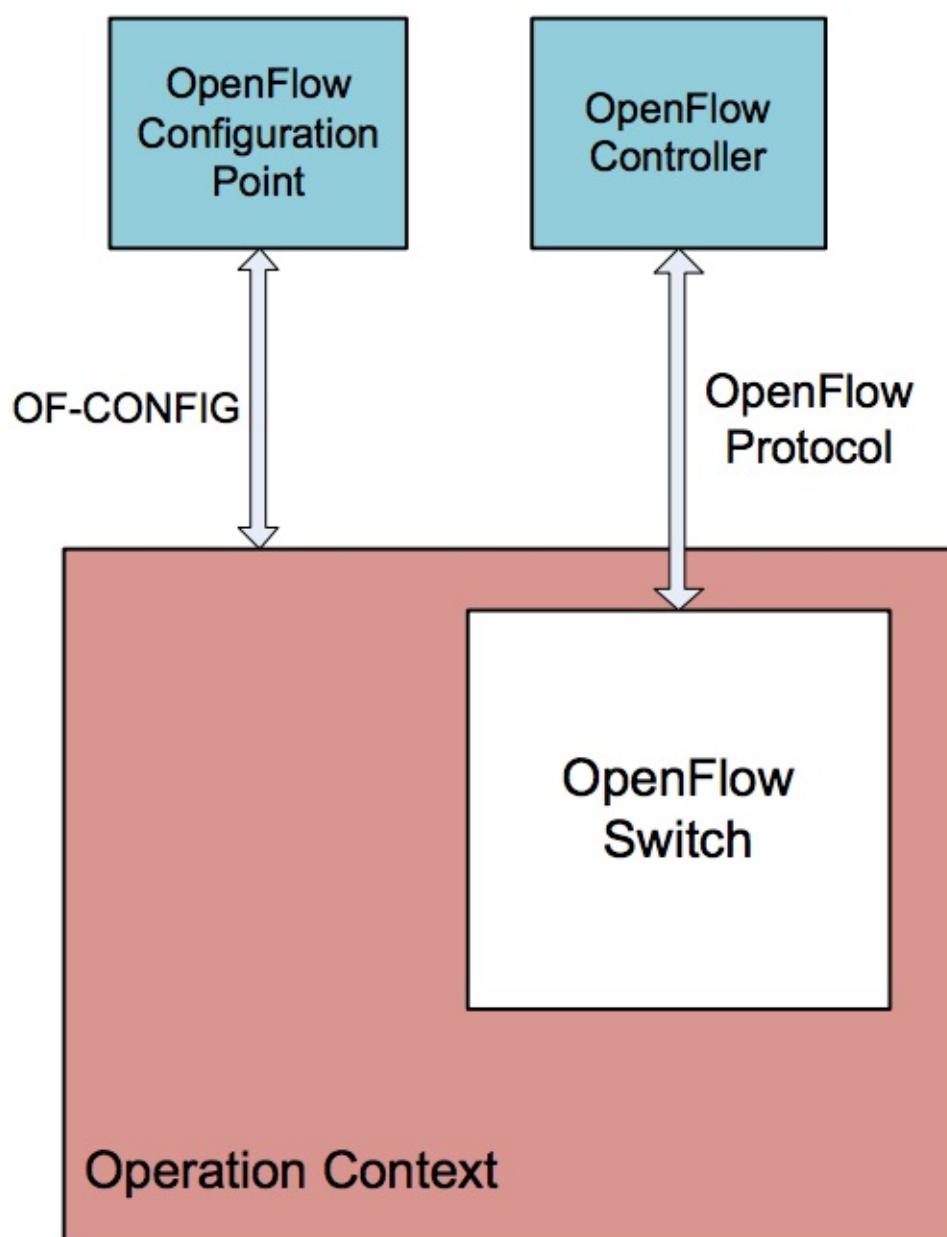
- **controller-to-switch**：控制器初始化并下发给交换机的报文，用于管理和查询交换机状态（如查询交换机特性，修改交换机流表、组表等）
- **asynchronous**：交换机异步发送给控制器的报文，用于更新网络事件和交换机状态的改变（如新报文到达、交换机端口变化等）
- **symmetric**：交换机或控制器发送，但无需对方许可，如Hello协商、Echo活性测试、Error错误报文等

## 参考文档

- [OpenFlow官方网站](#)
- [OpenFlow协议规范](#)

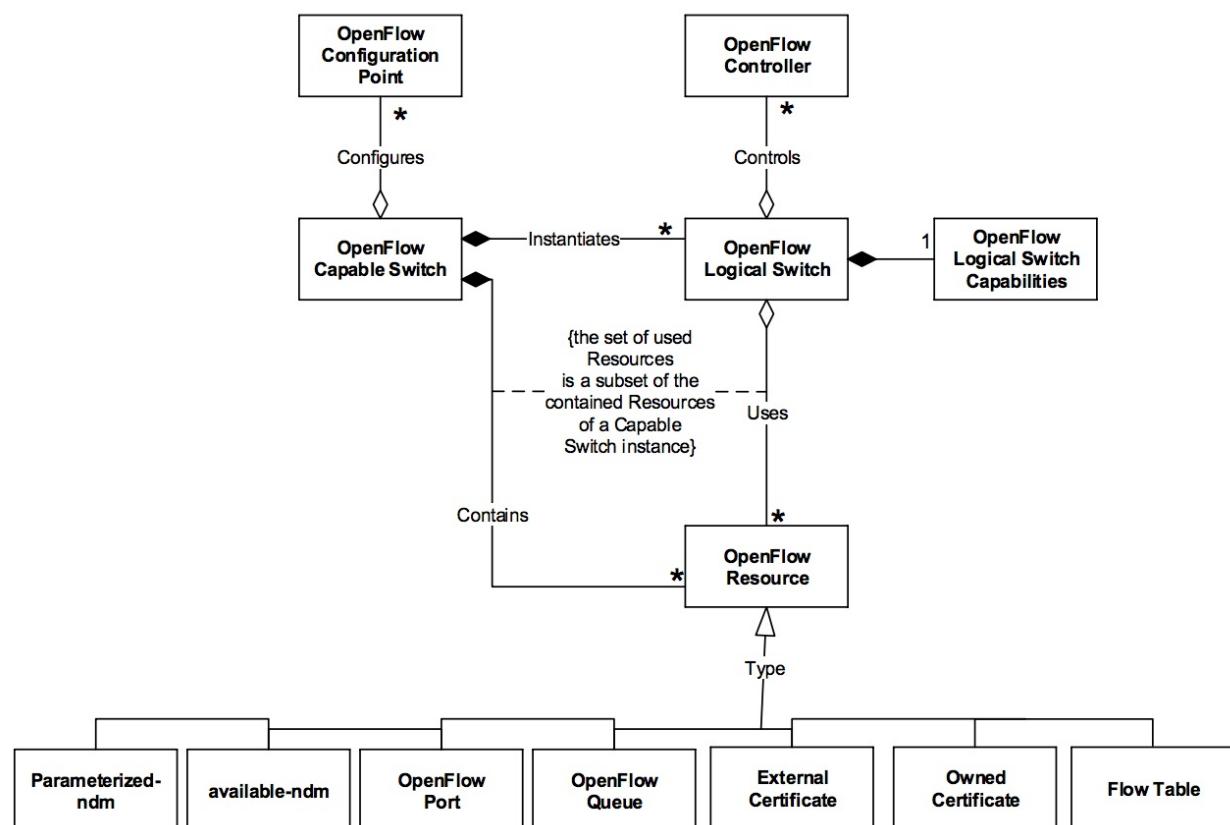
# OF-Config

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。

本质上来说，NETCONF就是利用XML-RPC的通讯机制实现配置客户端和配置服务端之间的通信，实现对网络设备的配置和管理。

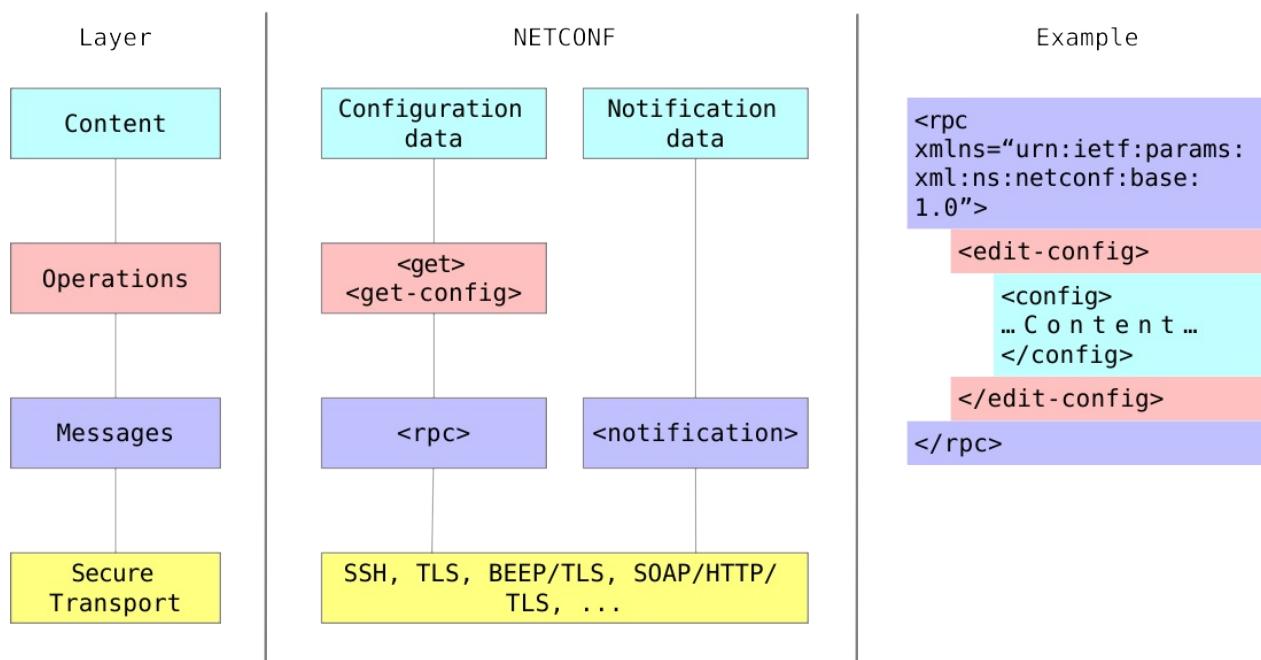
## 协议

NETCONF通过RPC与交换机通信，其协议包含四层

	Layer	Example
(4)	Content	Configuration data
(3)	Operations	<edit-config>
(2)	Messages	<rpc>, <rpc-reply>
(1)	Secure Transport	SSH, TLS, BEEP/TLS, SOAP/HTTP/TLS, ...

- (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

## NETCONF Layering Model And Example

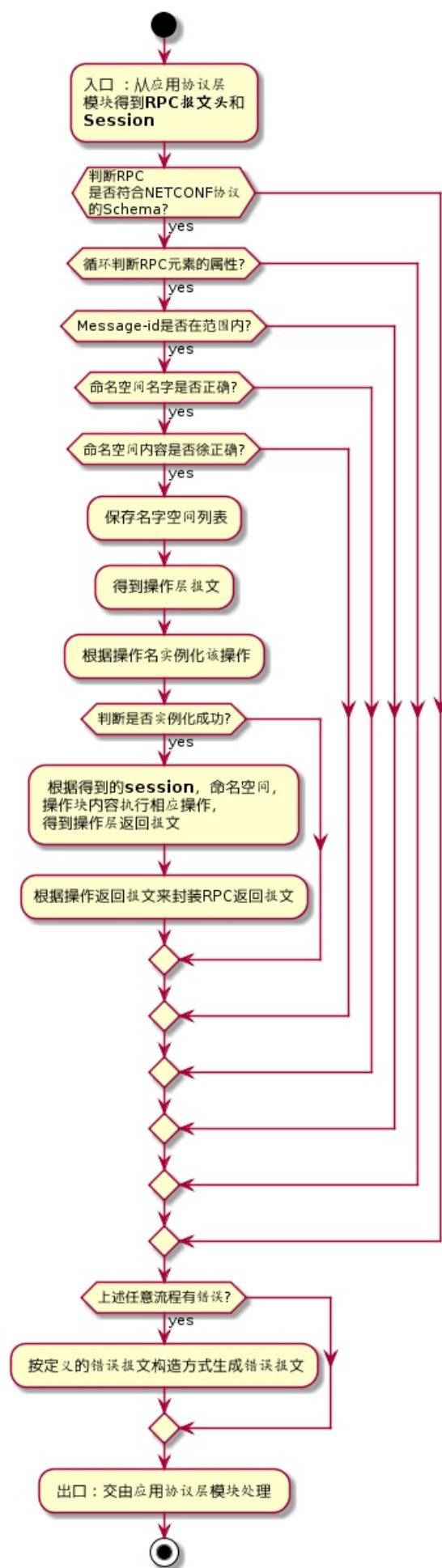
*NETCONF Layering Model*

## NETCONF技术规范

### 安全传输层

可以使用多种传输协议来进行数据传输，官方默认使用 `SSH` 进行加密及数据传输。

### 消息层



消息层流程PlantUML请查看[netconf-messages-layer-flow.puml](#)

上面的流程图看起来比较多，其实总结起来也主要是六步

1. 通过传输协议层模块得到 NETCONF 的请求报文后，就应交由 RPC 模块处理如果 NETCONF 报文经过了压缩或加密的话，先进行解压和解密
2. 然后将 RPC 请求报文，用 RFC4741 的 XML Schema 文件进行验证
3. 如果符合 NETCONF 的报文格式则解析文档中的 RPC 元素部分，进行 RPC 元素中 message-id 和命名空间属性的检查和保存
4. 然后将内部的操作层元素取出传给操作层模块
5. 如果操作层模块处理成功，返回正确的报文
6. RPC模块再将返回的响应报文进行 RPC 层的封装，然后发送给对应的客户端。如果中途在某个环节检查错误或操作层模块错误，则统一封装成错误处理报文，发送给客户端

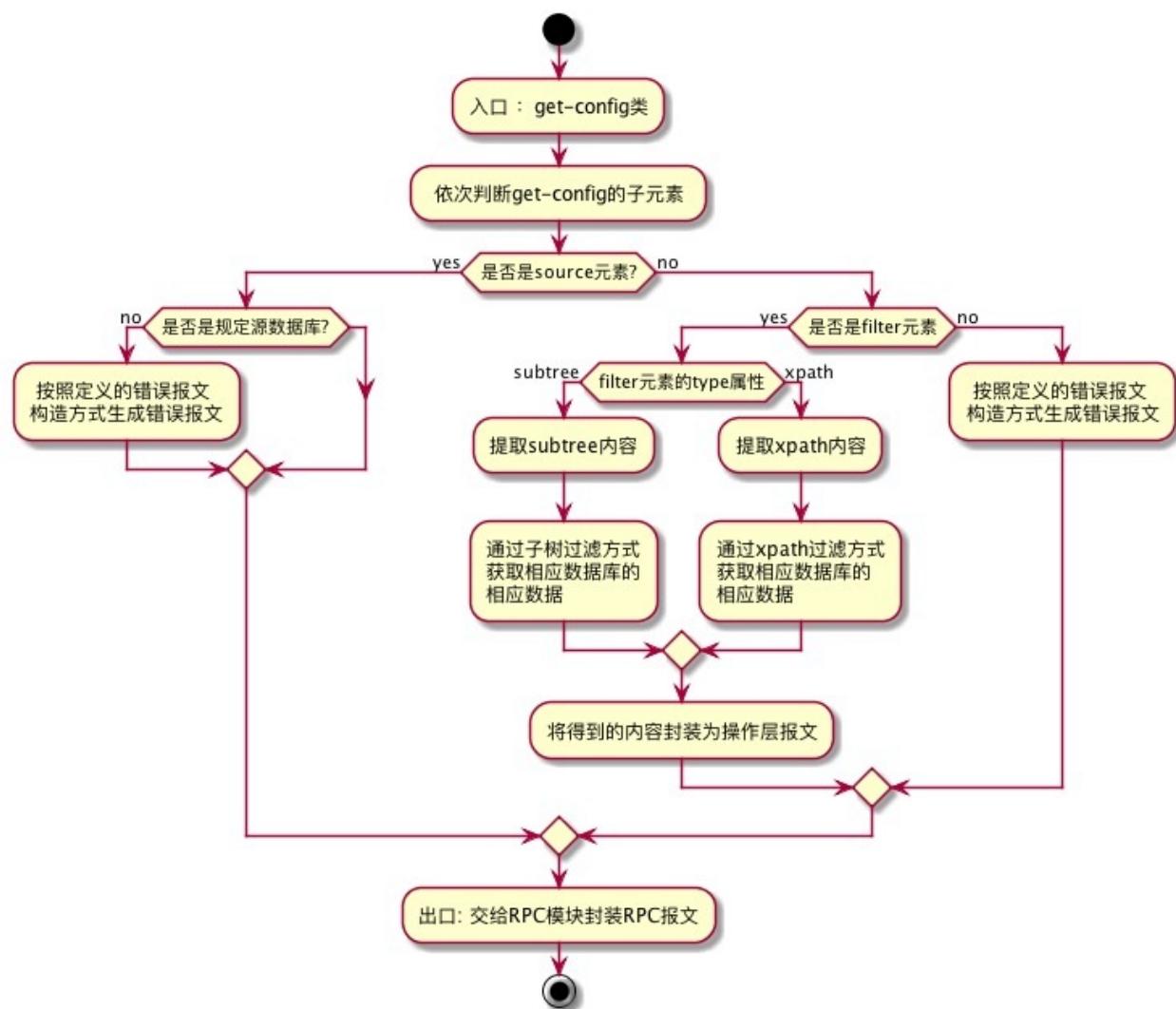
## 操作层

NETCONF提供了[九种基本操作](#)：

1. [get-config](#)
2. [edit-config](#)
3. [copy-config](#)
4. [delete-config](#)
5. [lock](#)
6. [unlock](#)
7. [get](#)
8. [close-session](#)
9. [kill-session](#)

这些操作的参数都各不相同，因此每种操作都有自己的处理流程

以 `get-config` 为例，我们来看看 `get-config` 请求报文的解析流程：

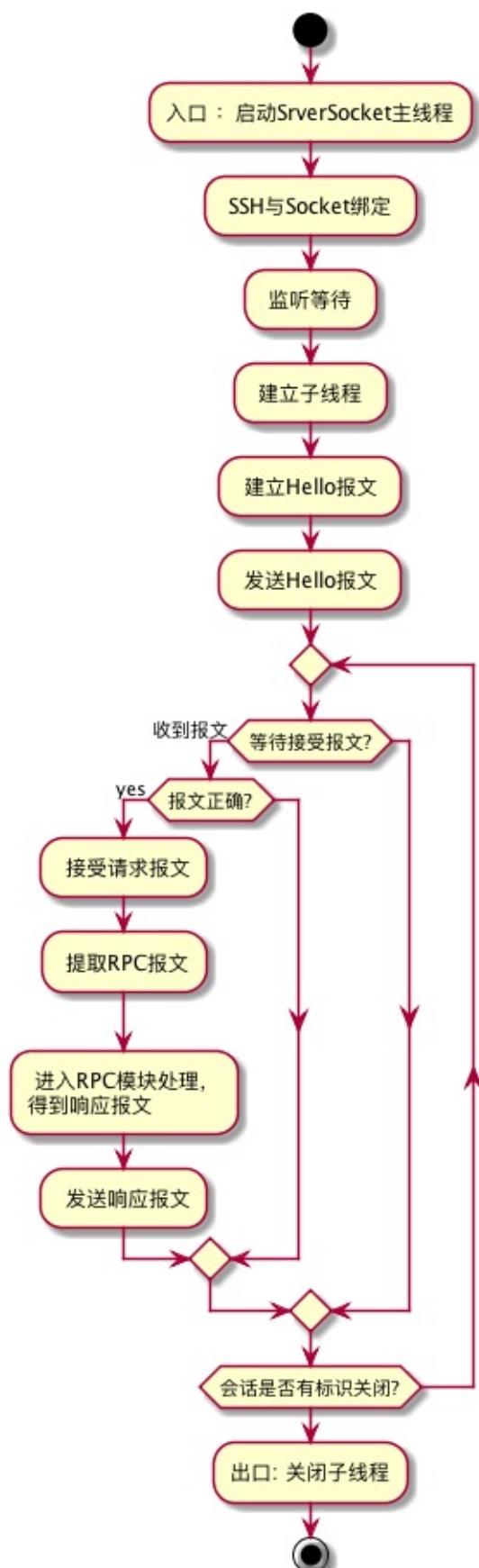


get-config请求报文解析流程PlantUML请查看[netconf-get-config-flow.puml](#)

## 内容层

内容层即配置数据库

## NETCONF 实现流程



NETCONF实现流程PlantUML请查看[netconf-implementation-process.puml](#)

## NETCONF实现的关键技术

关键的环节包括：安全认证、建立加密传输通道、**rpc-xml**消息收发、**rpc-xml**文件解析、**rpc-reply**消息的生成。

### 安全认证实现

使用 `ssh` 密钥认证方法，在 `Client` 端生成一对密钥，将公钥传给 `Server` 相关目录进行备份，借助 `libssh2` 函数库的相关函数完成安全认证。

### 加密通道建立

借助 `SSHv2` 本地端口转发功能分别在 `Client` 端和 `Server` 端建立一个 `SSH隧道`，实现 `Client` 端 `12500` 端口 和 `Server` `830` 端口 的数据经过 `SSH` 传输。

命令：`ssh -2 -N-C -f -L 12500:172.16.15.213:830` （`Client`端执行） 命令：`ssh -2 -N-C -f -L 830:172.16.15.213:12500` （`Server`端执行）

### xml-rpc消息收发

`Client` 端需要将 `rpc-xml` 文件转换为内容为 `rpc` 请求的 `xml` 化的字符串，`send()` 到 `Server`，`Server` 端在 `recv()` 收到字符之后利用 `libxml2` 库函数生成 `rpc-xml` 文件的指针，借助 `xpath` 搜索原理进行相关数据定位，如果检索到相关操作关键字再调用相关函数生成 `rpc-reply` 文件并转换为字符格式 `send()` 到 `client`。

### xml消息的解析

NETCONF 中有9种操作每一种操作的元素各不一样，因此要建立很多个流程来处理每一个操作，对 `rpc-xml` 请求的合法性以及想要获得的数据等进行定位，这个是 `RPC` 层需要实现的核心内容。

### xml消息的生成

原则上需要调用配置数据库的相关数据，执行并返回 `rpc-xml` 请求的结果。

### 参考

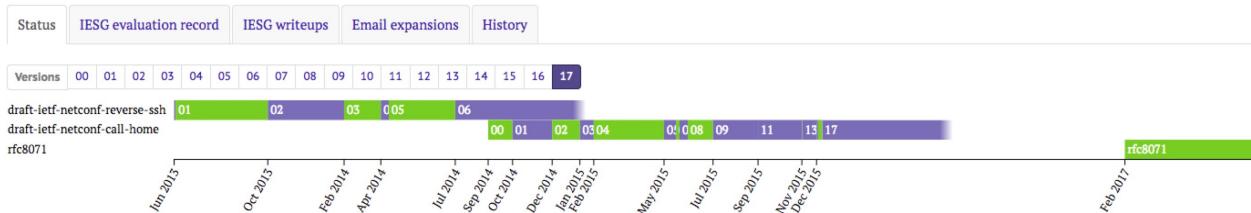
1. [rfc6241-zh](#)

2. RFC6241
3. 基于NETCONF协议的网管系统Agent端设计和实

# NETCONF Call Home

## NETCONF Call Home and RESTCONF Call Home

RFC 8071



本文主要内容都来自于今年二月发布的[RFC8071 - NETCONF Call Home and RESTCONF Call Home](#)，该RFC从2015年4月提出到最终发布一共修改了17个版本，其间修改内容可以[点击查看详细内容](#)。

## 介绍

NETCONF Call Home 支持两种安全传输网络配置协议分别是 Secure Shell(SSH) 和传输层安全 (TLS)。

NETCONF 协议的绑定到 SSH 在[RFC6242](#)中定义。NETCONF 协议的绑定到 TLS 在[RFC7589](#)中定义。SSH 协议在[RFC4253](#)中定义，TLS 协议是在[RFC5246](#)中定义。SSH 和 TLS 协议都是 TCP 协议之上的协议。

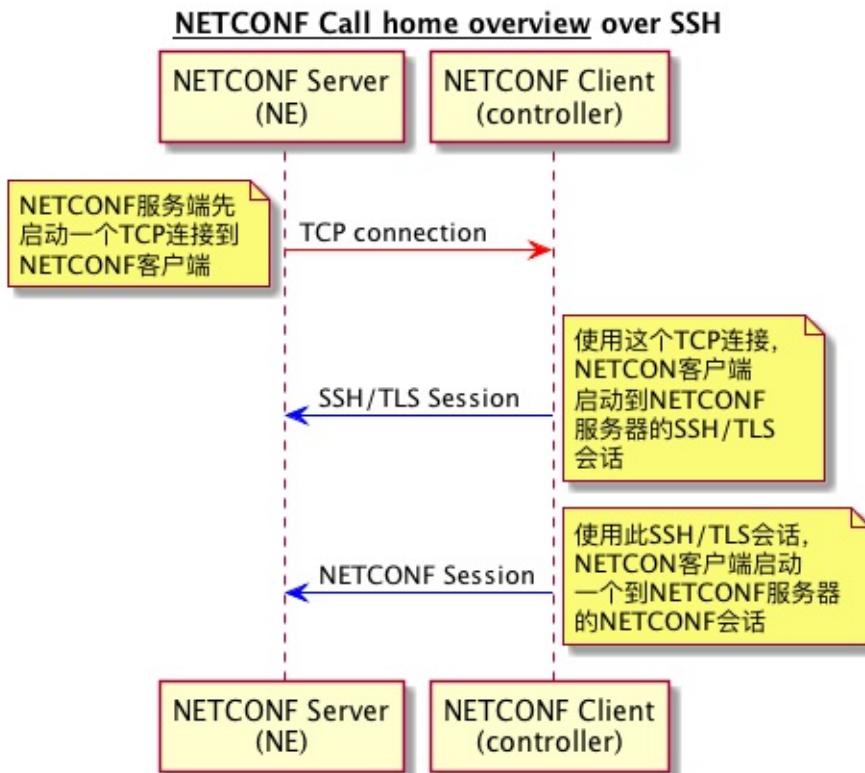
## 动机

call home 对于网络设备的初始化部署和持续管理都是非常有帮助的。那网络设备为什么使用 call home 这种方式？

- 网络设备在第一次启动后可以主动 call home，以便在其管理系统上注册。
- 网络设备可以以一种动态分配 IP 地址的方式访问网络，但是不会将其分配的IP地址注册到映射服务(例如，动态 DNS )。
- 网络设备可以部署在实现所有内部网络IP地址的网络地址转换（NAT）的防火墙后面。
- 网络设备元件可以部署在不允许任何管理访问内部网络的防火墙之后。
- 网络设备可以配置为“隐身模式”，因此没有任何的端口可以提供给管理系统打开连接。
- 运营商可能倾向于让网络设备发起管理连接，认为在数据中心中保护一个开放端口比在网络中的每个网络设备上具有开放端口更容易。

## 解决方案概述

下图说明了协议分层的 call home



消息层流程PlantUML请查看[netconf-messages-layer-flow.puml](#)

这张图有以下几点：

1. NETCONF 服务器首先启动一个 TCP 连接到 NETCONF 客户端。
2. 使用这个 TCP 连接，NETCONF 客户端启动到 NETCONF 服务器的 SSH/TLS 会话。
3. 使用此 SSH/TLS 会话，NETCONF 客户端启动一个到 NETCONF 服务器的 NETCONF 会话。

## NETCONF客户端

术语“客户端”在[RFC6241第1.1节](#)中定义。在网络管理的情况下，NETCONF 客户端可能是一个网络管理系统。

### 客户端协议操作事项

- C1 NETCONF 客户端侦听来自 NETCONF 服务器的 TCP 连接请求。客户端必须支持在[第6节](#)中定义的 IANA 分配的端口上接受 TCP 连接，但可以配置为侦听不同的端口。
- C2 NETCONF 客户端接受传入的 TCP 连接请求，并建立 TCP 连接。
- C3 使用此 TCP 连接，NETCONF 客户端启动 SSH 客户端[RFC4253](#)或 TLS 客户端[RFC5246](#) 协议。例如，假定使用 IANA 分配的端口，则在端口 4334 接受连接时启动 SSH 客户端协议，并且在端口 4335 或端口 4336 上接受连接时启动 TLS 客户端协议。
- C4 当使用 TLS 时，NETCONF 客户端必须告知 "peer\_allowed\_to\_send"，如[RFC6520](#)所定

义。这是必需的，以便 NETCONF 服务器知道在 call home 连接时需要发送心跳包，保持长连接。

- C5 作为建立 SSH 或 TLS 连接的一部分，NETCONF 客户端必须验证服务器提供的主机密钥或证书。该验证可以通过证书路径验证或通过将主机密钥或证书与先前信任的或“固定的”值进行比较来完成。如果证书被提交并且包含撤销检查信息，NETCONF 客户端应该检查证书的撤销状态。如果确定证书已被吊销，客户端必须马上关闭连接。
- C6 如果使用证书路径验证，则 NETCONF 客户端必须确保提供的证书具有对预先配置的颁发者证书的有效信任链，并且所呈现的证书对客户端之前知道的“标识符”[RFC6125](#)进行编码连接尝试。如何在证书中编码标识符可以由与证书颁发者相关的策略来确定。例如，可以知道给定的颁发者只在 X.509 证书的 “CommonName” 字段中签署具有唯一标识符（例如，序列号）的 IDevID 证书[Std-802.1AR-2009](#)。
- C7 服务器的主机密钥或证书经过验证后，客户端将以 SSH 或 TLS 协议进行建立 SSH 或 TLS 连接。在使用 NETCONF 服务器执行客户端认证时，NETCONF 客户端必须仅使用先前为 NETCONF 服务器提供的主机密钥或服务器证书关联的凭证。
- C8 一旦 SSH 或 TLS 连接建立，NETCONF 客户端启动 NETCONF 客户端[RFC6241](#) 或 RESTCONF 客户端[RFC8040](#)协议。假设使用 IANA 分配的端口，当在端口 4334 或端口 4335 上接受连接时启动 NETCONF 客户端协议，并且当在端口 4336 上接受连接时启动 RESTCONF 客户端协议。

## 客户端配置数据模型

如何配置 NETCONF 或 RESTCONF 客户端超出了本文的范围。

例如，可以使用什么样的配置来启用对 call home 的监听，配置可信证书颁发者，或者为预期的连接配置标识符。也就是说，在 [NETCONF-MODELS](#) 和 [RESTCONF-MODELS](#) 中提供了用于配置 NETCONF 和 RESTCONF 客户端的 YANG [RFC7950](#) 数据模块，包括 call home。

## NETCONF 服务器

术语“服务器”在[RFC6241第1.1节](#)中定义。在网络管理的情况下，NETCONF 服务器可能是网络元件或设备。

## 服务器协议操作

- S1 NETCONF 服务器向 NETCONF 客户端发起 TCP 连接请求。源端口可以根据本地策略或由操作系统随机分配。服务器必须支持连接到[第6节](#)中定义的一个 IANA 分配的端口，但可以配置为连接到不同的端口。使用 IANA 分配的端口，服务器通过 SSH 连接到 NETCONF 的端口 4334，通过 TLS 连接到 NETCONF 的端口 4335 和通过 TLS 的 RESTCONF 的端口 4336。
- S2 TCP 连接请求被接受，TCP 连接被建立。

- S3 使用此 TCP 连接，NETCONF 服务器将启动 SSH 服务器 RFC4253 或 TLS 服务 器 RFC5246 协议，具体取决于它如何配置。例如，假定使用 IANA 分配的端口，则在连 接到远程端口 4334 之后使用 SSH 服务器协议，并且在连接到远程端口 4335 或远程端 口 4336 之后使用 TLS 服务器协议。
- S4 作为建立 SSH 或 TLS 连接的一部分，NETCONF 服务器会将其主机密钥或证书发送给 客户端。如果发送了一个证书，服务器还必须发送所有中间证书到一个知名和可信赖的 发行者。如何发送证书列表在 RFC6187 第 2.1 节中为 SSH 定义，在 RFC5246 第 7.4.2 节 中 为 TLS 定义。
- S5 建立 SSH 或 TLS 会话需要在所有情况下对客户端证书进行服务器身份验证， 但 RESTCONF 除外，其中一些客户端身份验证方案在 安全传输连接 (TLS) 建立后发生。如 果需要传输级 (SSH 或 TLS) 客户端身份验证，并且客户端无法在本地策略定义的时 间 内成功向服务器进行身份验证，则服务器必须关闭连接。
- S6 一旦建立了 SSH 或 TLS 连接，NETCONF / RESTCONF 服务器将启动 NETCONF 服 务器 RFC6241 或 RESTCONF 服务器 RFC8040 协议，具体取决于如何配置。假设使 用 IANA 分配的端口，则在连接到远程端口 4334 或远程端口 4335 之后使用 NETCONF 服 务器协议，并且在连接到远程端口 4336 之后使用 RESTCONF 服务器协议。
- S7 如果需要长连接，作为连接发起者的 NETCONF / RESTCONF 服务器应该使用 keep- alive 机制主动测试连接的活跃性。对于基于 TLS 的连接，NETCONF / RESTCONF 服务 器应该发送 RFC6520 定义的 HeartbeatRequest 消息。对于基于 SSH 的连接，根据 RFC4254 的第 4 节，服务器应该发送一个 SSH\_MSG\_GLOBAL\_REQUEST 消息，其中包含一个特 别不存在的 "request name" 值（例如 keepalive@ietf.org）和 "want reply" 值设置 为 "1" 。

## 服务器配置数据模型

如何配置 NETCONF 或 RESTCONF 服务器超出了本文的范围。

这包括可能用于指定主机名，IP 地址，端口，算法或其他相关参数的配置。也就是说，NETCONF-MODELS 和 RESTCONF-MODELS 中提供了用于配置 NETCONF 和 RESTCONF 服务器的 YANG RFC7950 数据模块，包括 call home。

## 安全考虑

RFC6242 和 RFC7589 以及扩展 RFC4253，RFC5246 和 RFC8040 中描述的安全考虑也适用于 此处。

这个 RFC 与 SSH / TLS 服务器启动底层 TCP 连接的方式背离了标准的 SSH 和 TLS 的使用。这种逆转与 RFC4253 中的“客户端启动连接”和 RFC6125 不一致，它们表示“客户端必须构建可接受的引用标识符列表，并且必须独立于服务提供的标识符”。

与这些差异有关的风险主要集中在服务器认证上，客户无法将独立构建的引用标识符与服务器提供的引用标识符进行比较。为了减轻这些风险，要求 NETCONF / RESTCONF 客户端验证服务器的 SSH 主机密钥或证书，通过对预先配置的发行者证书进行证书路径验证，或者通过将主机密钥或证书与先前信任或“固定”值。此外，当使用证书时，要求客户端能够将在提供的证书中编码的标识符与客户端预先配置的标识符（例如，序列号）相匹配。

对于 NETCONF / RESTCONF 服务器提供 X.509 证书的情况，NETCONF / RESTCONF 客户端应确保用于证书路径验证的 “\\" 预配置颁发者证书对于服务器的制造商是唯一的。也就是说，证书不应该属于可能为多个制造商颁发证书的第三方认证机构。当使用将共享秘密（例如，密码）传递给服务器的客户机认证机制时，这是特别重要的。否则可能会导致客户端将共享密钥发送到恰好与客户端配置期望的服务器具有相同身份（例如，序列号）的另一个服务器的情况。

接下来会考虑与服务器身份验证无关的问题。

运行 NETCONF Call Home 或 RESTCONF Call Home 的面向 Internet 的主机将通过诸如“zmap”之类的扫描工具进行指纹识别。SSH 和 TLS 都提供了许多方法可以在主机上进行指纹识别。SSH 和 TLS 服务器相当成熟，能够抵御攻击，但是 SSH 和 TLS 客户端可能不够强大。实施者和部署需要确保提供软件更新机制，以便及时修复漏洞。

攻击者可以在推断出攻击者没有拥有有效密钥之前，对 NETCONF / RESTCONF 客户端进行拒绝服务（DoS）攻击，执行计算量大的操作。例如，在 TLS1.3 中，ClientHello 消息包含成本很高的非对称密钥操作的密钥共享值。推荐使用常见的减轻 DoS 攻击的预防措施，例如在一系列不成功的登录尝试后暂时将源地址列入黑名单。

当使用带有 RESTCONF 协议的 call home 时，在使用一些 HTTP 认证方案时，特别要注意传送共享密钥（例如密码）的 Basic RFC7617 和 Digest RFC7616 方案。对于使用的任何 HTTP 客户机认证方案，实施者和部署都应确保查看 RFC 中的“安全注意事项”部分。

## IANA 考虑事项

IANA已经在“用户端口”范围内为服务名称“netconf-ch-ssh”，“netconf-ch-tls”和“restconf-ch-tls”分配了三个 TCP 端口号。这些端口将是 NETCONF Call Home 和 RESTCONF Call Home 协议的默认端口。以下是遵循 RFC6335 中的规则的注册模板。

```

Service Name:          netconf-ch-ssh
Port Number:           4334
Transport Protocol(s): TCP
Description:           NETCONF Call Home (SSH)
Assignee:              IESG <iesg@ietf.org>
Contact:               IETF Chair <chair@ietf.org>
Reference:             RFC 8071

Service Name:          netconf-ch-tls
Port Number:           4335
Transport Protocol(s): TCP
Description:           NETCONF Call Home (TLS)
Assignee:              IESG <iesg@ietf.org>
Contact:               IETF Chair <chair@ietf.org>
Reference:             RFC 8071

Service Name:          restconf-ch-tls
Port Number:           4336
Transport Protocol(s): TCP
Description:           RESTCONF Call Home (TLS)
Assignee:              IESG <iesg@ietf.org>
Contact:               IETF Chair <chair@ietf.org>
Reference:             RFC 8071

```

## 参考实现

关于 NETCONF Call Home 的实现，可以参考 Juniper 在[github](#)上开源的[netconf-call-home](#)。

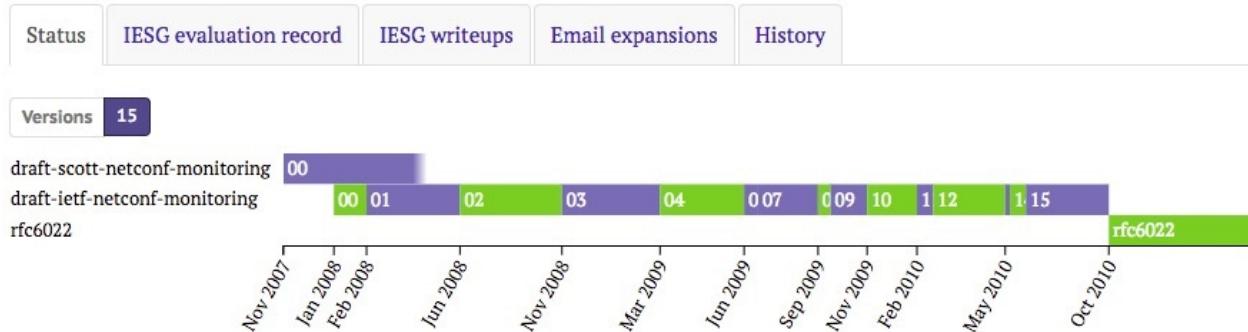
## 参考

- [RFC 8071 - NETCONF Call Home and RESTCONF Call Home](#)
- [RFC6242 - Using the NETCONF Protocol over Secure Shell \(SSH\)](#)
- [RFC7589 - Using the NETCONF Protocol over Transport Layer Security \(TLS\) with Mutual X.509 Authentication](#)
- [RFC4253 - The Secure Shell \(SSH\) Transport Layer Protocol](#)
- [RFC5246 - The Transport Layer Security \(TLS\) Protocol Version 1.2](#)
- [RFC8040 - RESTCONF Protocol](#)
- [Juniper netconf-call-home](#)

# YANG Module for NETCONF Monitoring

## YANG Module for NETCONF Monitoring

RFC 6022



本文主要内容都来自于2010年10月发布的[RFC6022 -YANG Module for NETCONF Monitoring](#)，该RFC从2007年11月提出到最终发布一共修改了15个版本，其间修改内容可以点击查看详细内容。

## 介绍

本文档定义了一个用于监视 NETCONF 协议的 YANG [RFC6020](#)模型。它提供了有关 NETCONF 会话和支持架构的信息，如[RFC4741](#)中所定义。

诸如不同的架构格式，功能可选性和访问控制等考虑因素都会影响 NETCONF 服务器在会话设置期间发送给客户端的详细信息的适用性和级别。

本文档中定义的方法需要进一步的手段来从 NETCONF 服务器查询和检索模式和 NETCONF 状态信息。提供这些是为了补充现有的 NETCONF 功能和操作，绝不会影响现有的行为。

还定义了一个新的 <get-schema> 操作来支持通过 NETCONF 显式的模式检索。

## NETCONF 监控数据模型

本文定义的NETCONF监控数据模型提供了 NETCONF 服务器的操作信息。这包括特定于 NETCONF 协议的细节（例如，特定于协议的计数器，诸如“ in-sessions ”）以及与模式检索（例如， schema list ）有关的数据。

实现本文档定义的数据模型的服务器（“ urn:ietf:params:xml:ns.yang:ietf-netconf-monitoring ”）务必按照[RFC6020](#)中的描述告知 capability URI 。

本节概述了监控数据模型。有关详细说明，请参阅本文档中提供的规范的YANG模块（请参阅第5节）。

## /netconf-state 子树

`netconf-state` 容器是监视数据模型的根。

```
netconf-state
  /capabilities
  /datastores
  /schemas
  /sessions
  /statistics
```

- 功能( `capabilities` )

服务器支持的`NETCONF`功能列表。

- 数据存储( `datastores` )

此设备上支持的`NETCONF`配置数据存储列表（例如，运行，启动，候选）以及相关信息。

- 模式( `schemas` )

服务器上支持的模式列表。 包括识别模式和支持其检索所需的所有信息。

- 会话( `sessions` )

设备上所有活动的NETCONF会话列表。 包括所有`NETCONF`会话的每个会话计数器。

- 统计( `statistics` )

包括`NETCONF`服务器的全局计数器。

## /netconf-state/capabilities 子树

`/netconf-state/capabilities` 子树包含 `NETCONF` 服务器支持的功能。

该列表必须包括在会话建立期间交换的所有能力，在请求时仍然适用。

## /netconf-state/datastores 子树

`/netconf-state/datastores` 子树包含 `NETCONF` 服务器的可用数据存储列表，并包含有关其锁定状态的信息。

```
datastore
  /name
  /locks
```

- 名称( `name` )( `leaf` , `netconf-datastore-type` )

支持枚举的数据存储; `candidate` , `running` , `startup` 。`

- 锁( `locks` )( `grouping` , `lock-info` )

数据存储的锁列表。信息提供全局和部分锁定[RFC5717](<https://tools.ietf.org/html/rfc5717>)。

对于部分锁定，将返回锁定节点列表和最初用于请求锁定的选择表达式。

## /netconf-state/schemas 子树

NETCONF 服务器支持的模式列表。

```
schema
  /identifier  (key)
  /version    (key)
  /format     (key)
  /namespace
  /location
```

network elements 的 `identifier` , `version` 和 `format` 在模式列表中用作关键字。这些用在 `<get-schema>` 操作中。

- `identifier (string)` 模式列表条目的标识符。该标识符在 `<get-schema>` 操作中使用，可用于其他方式，如文件检索。
- `version (string)` 支持的模式的版本。一个 NETCONF 服务器可以同时支持多个版本。每个版本必须在模式列表中单独报告，即具有相同的标识符，可能不同的位置，但是不同的版本。

对于 YANG 数据模型，版本是模块或子模块中最新的 YANG ' `revision` '语句的值，或者如果不存在' `revision` '语句，则为空字符串。

- `format (identifyref, schema-format)` 标识是使用什么格式的数据建模语言模式来描述该 `schema` 。可以使用“ `xsd` ”，“ `yang` ”，“ `yin` ”，“ `rng` ”和“ `rnc` ”这些格式来描述（见第 5 节）。
- `namespace (inet:uri)` 由模式定义的可扩展标记语言（ XML ）名称空间 XML-NAMES 。
- `location (union: enum, inet:uri)` 一个或多个可从中检索特定模式的位置。该列表应该每个模式至少包含一个条目。

## /netconf-state/sessions 子树

包括 NETCONF 管理会话的会话特定数据。会话列表必须包含所有当前活动的 NETCONF 会话。

```

session
  /session-id (key)
  /transport
  /username
  /source-host
  /login-time
  /in-rpcs
  /in-bad-rpcs
  /out-rpc-errors
  /out-notifications

```

- `session-id (uint32, 1..max)` 会话的唯一标识符。该值是[RFC4741](#)中定义的 NETCONF 会话标识符。
- `transport (identityref, transport)` 标识每个会话的传输。本文档定义了“`netconf-ssh`”，“`netconf-soap-over-beep`”，“`netconf-soap-over-https`”，“`netconf-beep`”和“`netconf-tls`”（见[第5节](#)）。
- `username (string)` `username` 是由 NETCONF 传输协议验证的客户端身份。用于导出用户名的算法是 NETCONF 传输协议特定的，另外还特定于 NETCONF 传输协议使用的验证机制。
- `source-host (inet:host)` 主机标识符（IP 地址或名称）的 NETCONF 客户端。
- `login-time (yang:date-and-time)` 在服务器上建立会话的时间。
- `in-rpcs (yang:zero-based-counter32)` 接收到正确的 `<rpc>` 消息的数量。
- `in-bad-rpcs (yang:zero-based-counter32)` 预期发送 `<rpc>` 消息时收到的消息数量，不正确的 `<rpc>` 消息。这包括 rpc 层上的 XML 解析错误和错误。
- `out-rpc-errors (yang:zero-based-counter32)` 在 `<rpc-reply>` 中包含 `<rpc-error>` 元素的消息数量。
- `out-notifications (yang:zero-based-counter32)` 发送的 `<notification>` 消息的数量。

## /netconf-state/statistics 子树

有关 NETCONF 服务器的统计数据。

```

statistics
  /netconf-start-time
  /in-bad-hellos
  /in-sessions
  /dropped-sessions
  /in-rpcs
  /in-bad-rpcs
  /out-rpc-errors
  /out-notifications

```

## statistics:

包含`NETCONF`服务器的与管理会话相关的性能数据。

- `netconf-start-time` (`yang:date-and-time`) 管理子系统启动的日期和时间。
- `in-bad-hellos` (`yang:zero-based-counter32`) 收到无效的消息的数量。
- `in-sessions` (`yang:zero-based-counter32`) 会话开始的数量。
- `dropped-sessions` (`yang:zero-based-counter32`) 异常终止的会话数，例如，由于空闲超时或传输关闭。
- `in-rpcs` (`yang:zero-based-counter32`) 接收到的消息的数量。
- `in-bad-rpcs` (`yang:zero-based-counter32`) 预期发送 `<rpc>` 消息时收到的消息数量，不正确的 `<rpc>` 消息。这包括 `rpc` 层上的 XML 解析错误和错误。
- `out-notifications` (`yang:zero-based-counter32`) 在 `<rpc-reply>` 中包含 `<rpc-error>` 元素的消息数量。

## 模式的具体操作

### 操作

- 描述：

此操作用于从 NETCONF 服务器检索模式。

- 参数：

- `identifier` (`string`) : 模式列表条目的标识符。强制性参数。
- `version` (`string`) : 请求的模式的版本。可选参数。
- `format` (`identityref, schema-format`) : 模式的数据建模语言。未指定时，默认值为“`yang`”。可选参数。

- 正面回应：

NETCONF 服务器返回请求的模式。

- 负面的回应：

如果请求的模式不存在，那么 `<error-tag>` 是“无效值”。如果多个模式匹配请求的参数，那么 `<error-tag>` 是' `operation-failed`'，而 `<error-app-tag>` 是' `data-not-unique`'。

## 例子

### 通过 `<get>` 操作检索模式列表

NETCONF 客户端通过 `<get>` 操作检索 `/netconf-state/schemas` 子树，从 NETCONF 服务器中检索支持的模式列表。

请求会话的可用模式将在包含 `<identifier>`，`<version>`，`<format>` 和 `<location>` 元素的答复中返回。

响应数据可用于确定可用模式及其版本。模式本身（即模式内容）不会在响应中返回。可选的 `<location>` 元素包含一个 URI，可以通过其他协议（如 `ftp RFC0959` 或 `http(s) RFC2616 RFC2818`）或特殊值“NETCONF”来检索架构，这意味着可以通过 `<get-schema>` 操作从设备检索模式。

例子：

```

<rpc message-id="101"
      xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter type="subtree">
      <netconf-state xmlns=
        "urn:ietf:params:xml:yang:ietf-netconf-monitoring">
        <schemas/>
      </netconf-state>
    </filter>
  </get>
</rpc>

```

NETCONF 服务器返回可用于检索的模式列表。

```
<rpc-reply message-id="101"
    xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<netconf-state
    xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
    <schemas>
        <schema>
            <identifier>foo</identifier>
            <version>1.0</version>
            <format>xsd</format>
            <namespace>http://example.com/foo</namespace>
            <location>ftp://ftp.example.com/schemas/foo_1.0.xsd</location>
            <location>http://www.example.com/schema/foo_1.0.xsd</location>
            <location>NETCONF</location>
        </schema>
        <schema>
            <identifier>foo</identifier>
            <version>1.1</version>
            <format>xsd</format>
            <namespace>http://example.com/foo</namespace>
            <location>ftp://ftp.example.com/schemas/foo_1.1.xsd</location>
            <location>http://www.example.com/schema/foo_1.1.xsd</location>
            <location>NETCONF</location>
        </schema>
        <schema>
            <identifier>bar</identifier>
            <version>2008-06-01</version>
            <format>yang</format>
            <namespace>http://example.com/bar</namespace>
            <location>
                http://example.com/schema/bar@2008-06-01.yang
            </location>
            <location>NETCONF</location>
        </schema>
        <schema>
            <identifier>bar-types</identifier>
            <version>2008-06-01</version>
            <format>yang</format>
            <namespace>http://example.com/bar</namespace>
            <location>
                http://example.com/schema/bar-types@2008-06-01.yang
            </location>
            <location>NETCONF</location>
        </schema>
    </schemas>
</netconf-state>
</data>
</rpc-reply>
```

## 检索模式实例

鉴于上一节中的答复，以下示例说明在多个位置，多种格式和多个位置检索“`foo`”，“`bar`”和“`bar-types`”模式。

1. `foo`，`xsd` 格式的 `version 1.0`：

2. 通过FTP使用位置

```
ftp://ftp.example.com/schemas/foo_1.0.xsd
```

3. 通过HTTP使用位置

```
http://www.example.com/schema/foo_1.0.xsd
```

4. 通过 `<get-schema>` 使用标识符，版本和格式参数。

```

<rpc message-id="101"
      xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-schema
      xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
    <identifier>foo</identifier>
    <version>1.0</version>
    <format>xsd</format>
  </get-schema>
</rpc>

<rpc-reply message-id="101"
          xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data
      xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
    <xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">
      <!-- foo 1.0 xsd schema contents here -->
    </xss:schema>
  </data>
</rpc-reply>

```

1. `bar`，默认的 YANG 格式的 `version 2008-06-01`：

2. 通过 HTTP 使用位置

<http://example.com/schema/bar@2008-06-01.yang>

3. 通过 `<get-schema>` 使用标识符和版本参数：

```

<rpc message-id="102"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-schema
    xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
    <identifier>bar</identifier>
    <version>2008-06-01</version>
  </get-schema>
</rpc>

<rpc-reply message-id="102"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data
    xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
    module bar {
      //default format (yang) returned
      //bar version 2008-06-01 yang module
      //contents here ...
    }
  </data>
</rpc-reply>

```

1. `bar-types`，默认的 YANG 格式的 `version 2008-06-01`。

2. 使用 `identifier` 参数：

```

<rpc message-id="103"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-schema
    xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
    <identifier>bar-types</identifier>
  </get-schema>
</rpc>

<rpc-reply message-id="103"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data
    xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
    module bar-types {
      //default format (yang) returned
      //latest revision returned
      //is version 2008-06-01 yang module
      //contents here ...
    }
  </data>
</rpc-reply>

```

## NETCONF 监控数据模型

本备忘录中描述的数据模型在以下的 YANG 模块中定义。

这个YANG模块从[RFC6021](#)和

[RFC4741](#)，[RFC4742](#)，[RFC4743](#)，[RFC4744](#)，[RFC5539](#)，[xmlschema-1](#)，[RFC6020](#)，[ISO / IEC19757-2:2008]，和[RFC5717](#)。

```
<CODE BEGINS> file "ietf-netconf-monitoring@2010-10-04.yang"
```

```
module ietf-netconf-monitoring {

    namespace "urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring";
    prefix "ncm";

    import ietf-yang-types { prefix yang; }
    import ietf-inet-types { prefix inet; }

    organization
        "IETF NETCONF (Network Configuration) Working Group";

    contact
        "WG Web: <http://tools.ietf.org/wg/netconf/>
        WG List: <mailto:netconf@ietf.org>

        WG Chair: Mehmet Ersue
        <mailto:mehmet.ersue@nsn.com>

        WG Chair: Bert Wijnen
        <mailto:bertietf@bwijnen.net>

        Editor: Mark Scott
        <mailto:mark.scott@ericsson.com>

        Editor: Martin Bjorklund
        <mailto:mbj@tail-f.com>";

    description
        "NETCONF Monitoring Module.
        All elements in this module are read-only.

        Copyright (c) 2010 IETF Trust and the persons identified as
        authors of the code. All rights reserved.

        Redistribution and use in source and binary forms, with or
        without modification, is permitted pursuant to, and subject
        to the license terms contained in, the Simplified BSD
        License set forth in Section 4.c of the IETF Trust's
        Legal Provisions Relating to IETF Documents
        (http://trustee.ietf.org/license-info).

        This version of this YANG module is part of RFC 6022; see
        the RFC itself for full legal notices.";
```

```

revision 2010-10-04 {
    description
        "Initial revision.";
    reference
        "RFC 6022: YANG Module for NETCONF Monitoring";
}

typedef netconf-datastore-type {
    type enumeration {
        enum running;
        enum candidate;
        enum startup;
    }
    description
        "Enumeration of possible NETCONF datastore types.";
    reference
        "RFC 4741: NETCONF Configuration Protocol";
}

identity transport {
    description
        "Base identity for NETCONF transport types.";
}

identity netconf-ssh {
    base transport;
    description
        "NETCONF over Secure Shell (SSH).";
    reference
        "RFC 4742: Using the NETCONF Configuration Protocol
            over Secure SHell (SSH)";
}

identity netconf-soap-over-beep {
    base transport;
    description
        "NETCONF over Simple Object Access Protocol (SOAP) over
            Blocks Extensible Exchange Protocol (BEEP).";
    reference
        "RFC 4743: Using NETCONF over the Simple Object
            Access Protocol (SOAP)";
}

identity netconf-soap-over-https {
    base transport;
    description
        "NETCONF over Simple Object Access Protocol (SOAP)
            over Hypertext Transfer Protocol Secure (HTTPS).";
    reference
        "RFC 4743: Using NETCONF over the Simple Object
            Access Protocol (SOAP)";
}

```

```

identity netconf-beep {
    base transport;
    description
        "NETCONF over Blocks Extensible Exchange Protocol (BEEP).";
    reference
        "RFC 4744: Using the NETCONF Protocol over the
            Blocks Extensible Exchange Protocol (BEEP)";
}

identity netconf-tls {
    base transport;
    description
        "NETCONF over Transport Layer Security (TLS).";
    reference
        "RFC 5539: NETCONF over Transport Layer Security (TLS)";
}

identity schema-format {
    description
        "Base identity for data model schema languages.";
}

identity xsd {
    base schema-format;
    description
        "W3C XML Schema Definition.";
    reference
        "W3C REC REC-xmlschema-1-20041028:
            XML Schema Part 1: Structures";
}

identity yang {
    base schema-format;
    description
        "The YANG data modeling language for NETCONF.";
    reference
        "RFC 6020: YANG - A Data Modeling Language for the
            Network Configuration Protocol (NETCONF)";
}

identity yin {
    base schema-format;
    description
        "The YIN syntax for YANG.";
    reference
        "RFC 6020: YANG - A Data Modeling Language for the
            Network Configuration Protocol (NETCONF)";
}

identity rng {
    base schema-format;
    description
        "Regular Language for XML Next Generation (RELAX NG).";
}

```

```

reference
    "ISO/IEC 19757-2:2008: RELAX NG";
}

identity rnc {
    base schema-format;
    description
        "Relax NG Compact Syntax";
    reference
        "ISO/IEC 19757-2:2008: RELAX NG";
}

grouping common-counters {
    description
        "Counters that exist both per session, and also globally,
         accumulated from all sessions./";

leaf in-rpcs {
    type yang:zero-based-counter32;
    description
        "Number of correct <rpc> messages received.";
}
leaf in-bad-rpcs {
    type yang:zero-based-counter32;
    description
        "Number of messages received when an <rpc> message was expected,
         that were not correct <rpc> messages. This includes XML parse
         errors and errors on the rpc layer.";
}
leaf out-rpc-errors {
    type yang:zero-based-counter32;
    description
        "Number of <rpc-reply> messages sent that contained an
         <rpc-error> element.";
}
leaf out-notifications {
    type yang:zero-based-counter32;
    description
        "Number of <notification> messages sent.";
}
}

container netconf-state {
    config false;
    description
        "The netconf-state container is the root of the monitoring
         data model./";

    container capabilities {
        description
            "Contains the list of NETCONF capabilities supported by the
             server.";
    }
}

```

```

        leaf-list capability {
            type inet:uri;
            description
                "List of NETCONF capabilities supported by the server.";
        }
    }

    container datastores {
        description
            "Contains the list of NETCONF configuration datastores.";

        list datastore {
            key name;
            description
                "List of NETCONF configuration datastores supported by
                 the NETCONF server and related information.";

            leaf name {
                type netconf-datastore-type;
                description
                    "Name of the datastore associated with this list entry.";
            }
            container locks {
                presence
                    "This container is present only if the datastore
                     is locked.";
                description
                    "The NETCONF <lock> and <partial-lock> operations allow
                     a client to lock specific resources in a datastore. The
                     NETCONF server will prevent changes to the locked
                     resources by all sessions except the one that acquired
                     the lock(s).

                     Monitoring information is provided for each datastore
                     entry including details such as the session that acquired
                     the lock, the type of lock (global or partial) and the
                     list of locked resources. Multiple locks per datastore
                     are supported.";

                grouping lock-info {
                    description
                        "Lock related parameters, common to both global and
                         partial locks.";

                    leaf locked-by-session {
                        type uint32;
                        mandatory true;
                        description
                            "The session ID of the session that has locked
                             this resource. Both a global lock and a partial
                             lock MUST contain the NETCONF session-id.

                             If the lock is held by a session that is not managed

```

```

        by the NETCONF server (e.g., a CLI session), a session
        id of 0 (zero) is reported.";
    reference
        "RFC 4741: NETCONF Configuration Protocol";
    }
leaf locked-time {
    type yang:date-and-time;
    mandatory true;
    description
        "The date and time of when the resource was
        locked.";
}
}

choice lock-type {
    description
        "Indicates if a global lock or a set of partial locks
        are set.";

    container global-lock {
        description
            "Present if the global lock is set.";
        uses lock-info;
    }

    list partial-lock {
        key lock-id;
        description
            "List of partial locks.";
        reference
            "RFC 5717: Partial Lock Remote Procedure Call (RPC) for
            NETCONF";
    }

    leaf lock-id {
        type uint32;
        description
            "This is the lock id returned in the <partial-lock>
            response.";
    }
    uses lock-info;
    leaf-list select {
        type yang>xpath1.0;
        min-elements 1;
        description
            "The xpath expression that was used to request
            the lock. The select expression indicates the
            original intended scope of the lock.";
    }
    leaf-list locked-node {
        type instance-identifier;
        description
            "The list of instance-identifiers (i.e., the
            locked nodes).

```



```

        in (currently xsd, yang, yin, rng, or rnc).
        For YANG data models, 'yang' format MUST be supported and
        'yin' format MAY also be provided.";
    }
    leaf namespace {
        type inet:uri;
        mandatory true;
        description
            "The XML namespace defined by the data model.

            For YANG data models, this is the module's namespace.
            If the list entry describes a submodule, this field
            contains the namespace of the module to which the
            submodule belongs.";
    }
    leaf-list location {
        type union {
            type enumeration {
                enum "NETCONF";
            }
            type inet:uri;
        }
        description
            "One or more locations from which the schema can be
            retrieved. This list SHOULD contain at least one
            entry per schema.

            A schema entry may be located on a remote file system
            (e.g., reference to file system for ftp retrieval) or
            retrieved directly from a server supporting the
            <get-schema> operation (denoted by the value 'NETCONF').";
    }
}
}

container sessions {
    description
        "The sessions container includes session-specific data for
        NETCONF management sessions. The session list MUST include
        all currently active NETCONF sessions.";

    list session {
        key session-id;
        description
            "All NETCONF sessions managed by the NETCONF server
            MUST be reported in this list.";

        leaf session-id {
            type uint32 {
                range "1..max";
            }
            description
                "Unique identifier for the session. This value is the
                NETCONF session identifier, as defined in RFC 4741.";
        }
    }
}

```

```

reference
    "RFC 4741: NETCONF Configuration Protocol";
}
leaf transport {
    type identityref {
        base transport;
    }
    mandatory true;
    description
        "Identifies the transport for each session, e.g.,
         'netconf-ssh', 'netconf-soap', etc.";
}
leaf username {
    type string;
    mandatory true;
    description
        "The username is the client identity that was authenticated
         by the NETCONF transport protocol. The algorithm used to
         derive the username is NETCONF transport protocol specific
         and in addition specific to the authentication mechanism
         used by the NETCONF transport protocol.";
}
leaf source-host {
    type inet:host;
    description
        "Host identifier of the NETCONF client. The value
         returned is implementation specific (e.g., hostname,
         IPv4 address, IPv6 address)";
}
leaf login-time {
    type yang:date-and-time;
    mandatory true;
    description
        "Time at the server at which the session was established.";
}
uses common-counters {
    description
        "Per-session counters. Zero based with following reset
         behaviour:
            - at start of a session
            - when max value is reached";
}
}

container statistics {
    description
        "Statistical data pertaining to the NETCONF server.";

leaf netconf-start-time {
    type yang:date-and-time;
    description
        "Date and time at which the management subsystem was

```

```

        started.";

}

leaf in-bad-hellos {
    type yang:zero-based-counter32;
    description
        "Number of sessions silently dropped because an
        invalid <hello> message was received. This includes <hello>
        messages with a 'session-id' attribute, bad namespace, and
        bad capability declarations.";
}

leaf in-sessions {
    type yang:zero-based-counter32;
    description
        "Number of sessions started. This counter is incremented
        when a <hello> message with a <session-id> is sent.

        'in-sessions' - 'in-bad-hellos' =
            'number of correctly started netconf sessions'";
}

leaf dropped-sessions {
    type yang:zero-based-counter32;
    description
        "Number of sessions that were abnormally terminated, e.g.,
        due to idle timeout or transport close. This counter is not
        incremented when a session is properly closed by a
        <close-session> operation, or killed by a <kill-session>
        operation.";
}

uses common-counters {
    description
        "Global counters, accumulated from all sessions.
        Zero based with following reset behaviour:
            - re-initialization of NETCONF server
            - when max value is reached";
}

}

}

}

rpc get-schema {
    description
        "This operation is used to retrieve a schema from the
        NETCONF server.

Positive Response:
The NETCONF server returns the requested schema.

Negative Response:
If requested schema does not exist, the <error-tag> is
'invalid-value'.

If more than one schema matches the requested parameters, the
<error-tag> is 'operation-failed', and <error-app-tag> is
'data-not-unique'.";
}

```

```

input {
    leaf identifier {
        type string;
        mandatory true;
        description
            "Identifier for the schema list entry.";
    }
    leaf version {
        type string;
        description
            "Version of the schema requested. If this parameter is not
            present, and more than one version of the schema exists on
            the server, a 'data-not-unique' error is returned, as
            described above.";
    }
    leaf format {
        type identityref {
            base schema-format;
        }
        description
            "The data modeling language of the schema. If this
            parameter is not present, and more than one formats of
            the schema exists on the server, a 'data-not-unique' error
            is returned, as described above.";
    }
}
output {
    anyxml data {
        description
            "Contains the schema content.";
    }
}
}
}

```

&lt;CODE ENDS&gt;

## 安全考虑

本备忘录中定义的 YANG 模块设计为通过 NETCONF 协议[RFC4741](#)访问。最低的 NETCONF 层是安全传输层，实现安全传输的强制是 SSH [RFC4742](#)。

某些网络环境中，这个YANG模块中的一些可读数据节点可能被认为是敏感的或易受攻击的。因此，控制对这些数据节点的读访问（例如，通过get，get-config或通知）是很重要的。

这些容器，列表节点和数据节点具有其特定的敏感性/漏洞：

`/netconf-state/sessions/session/username`：包含可用于尝试向服务器进行身份验证的身份信息。

此用户名仅用于监控，不应用于其他用途，如访问控制，而不详细讨论此用户名的限制。例如，服务器A和服务器B可能会报告相同的用户名，但这可能是针对不同的人。

## IANA考虑事项

本文档在“`IETF XML 注册表`”中注册了一个URI。遵循[RFC3688](#)中的格式，已经注册了以下内容。

```
URI: urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring  
Registrant Contact: The IESG.  
XML: N/A, the requested URI is an XML namespace.
```

本文档在“`YANG 模块名称`”注册表中注册了一个模块。遵循[RFC6020](#)中的格式，已经注册了以下内容。

```
name: ietf-netconf-monitoring  
namespace: urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring  
prefix: ncm  
reference: [RFC 6022](https://tools.ietf.org/html/rfc6022)
```

## 参考

### 规范性参考文献

- [ISO/IEC19757-2:2008](#) ISO/IEC, "Document Schema Definition Language (DSDL) --Part 2: Regular-grammar-based validation -- RELAX NG", December 2008,  
[http://www.iso.org/iso/catalogue\\_detail.htm?csnumber=37605](http://www.iso.org/iso/catalogue_detail.htm?csnumber=37605).
- [RFC2119](#) Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC4741](#) Enns, R., "NETCONF Configuration Protocol", RFC 4741, December 2006.
- [RFC4742](#) Wasserman, M. and T. Goddard, "Using the NETCONF Configuration Protocol over Secure SHell (SSH)", RFC 4742, December 2006.
- [RFC4743](#) Goddard, T., "Using NETCONF over the Simple Object Access Protocol (SOAP)", RFC 4743, December 2006.

- [RFC4744](#) Lear, E. and K. Crozier, "Using the NETCONF Protocol over the Blocks Extensible Exchange Protocol (BEEP)", RFC 4744, December 2006.
- [RFC5539](#) Badra, M., "NETCONF over Transport Layer Security (TLS)", RFC 5539, May 2009.
- [RFC5717](#) Lengyel, B. and M. Bjorklund, "Partial Lock Remote Procedure Call (RPC) for NETCONF", RFC 5717, December 2009.
- [RFC6020](#) Bjorklund, M., Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", October 2010.
- [RFC6021](#) Schoenwaelder, J., Ed., "Common YANG Data Types", October 2010.
- [XML-NAMES](#) Hollander, D., Tobin, R., Thompson, H., Bray, T., and A. Layman, "Namespaces in XML 1.0 (Third Edition)", World Wide Web Consortium Recommendation REC-xml-names-20091208, December 2009, <http://www.w3.org/TR/2009/REC-xml-names-20091208>.
- [xmldata-1](#) Biron, Paul V. and Ashok. Malhotra, "XML Schema Part 1: Structures Second Edition W3C Recommendation 28 October 2004", October 2004, <http://www.w3.org/TR/xmldata-1>.

## 信息参考

- [RFC0959](#) Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, October 1985.
- [RFC2616](#) Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.
- [RFC2818](#) Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
- [RFC3688](#) Mealling, M., "The IETF XML Registry", [BCP 81](#), RFC 3688, January 2004.

# NETCONF请求和响应中的标签

## ]]>]]>

]]>]]> 标签在 NETCONF 服务器和客户端应用程序发送的每个 XML 文档的末尾。

在使用SSH协议作为NETCONF通讯协议时，NETCONF的服务器和客户端在每一次请求中都需要在末尾加上该标签，比如 </ hello> 、 </ rpc> 、 </ rpc-reply> 标记。

例子：

```
<hello>
    <!-- child tag elements included by client application or NETCONF server -->
</hello>
]]>]]>
```

```
<rpc [attributes]>
    <!-- tag elements in a request from a client application -->
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <!-- tag elements in the response from the NETCONF server -->
</rpc-reply>
]]>]]>
```

可以在[RFC 6242 Using the NETCONF Protocol over Secure Shell \(SSH\)](#)查看此约束。

## <data>

用于NETCONF服务器针对 <get> 和 <get-config> 两类 request 来返回的配置数据和设备信息响应。

注意：默认情况下，NETCONF服务器将返回格式为 XML tag elements 的配置数据。如果客户端应用程序在 <get> 请求中请求不同的格式，那么包含在 <data> 元素中的配置数据可能会有所不同。

<configuration> - 包含配置标签元素。它是 XML API 中的顶级标记元素。

例子：

```

<rpc-reply message-id="101"
            xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<top xmlns="http://example.com/schema/1.2/config">
<users>
<user>
<name>root</name>
<company-info>
<dept>1</dept>
<id>1</id>
</company-info>
</user>
<user>
<name>fred</name>
<company-info>
<id>2</id>
</company-info>
</user>
</users>
</top>
</data>
</rpc-reply>
]]>]]>

```

## <hello>

该标签告知了NETCONF服务器支持哪些在NETCONF规范中定义的operations或capabilities。

客户端应用程序必须在NETCONF会话期间在任何其他标记元素之前发出<hello>标记元素，并且不得多次发出它。

- <capabilities> - 包含一个或多个<capabilities>标签，它们共同指定一组支持的NETCONF操作。
- <capability> - 指定NETCONF规范或供应商定义的能力的统一资源标识符（URI）。NETCONF规范中的每个功能由统一资源名称（URN）表示。供应商定义的功能由URN或URL表示。
- <session-id> - （仅由NETCONF服务器生成）指定会话的NETCONF服务器的UNIX进程ID（PID）

例子：

```
<!-- emitted by a client application -->
<hello>
  <capabilities>
    <capability>URI</capability>
  </capabilities>
</hello>
]]>]]>
```

```
<!-- emitted by the NETCONF server -->
<hello>
  <capabilities>
    <capability>URI</capability>
  </capabilities>
  <session-id>session-identifier</session-id>
</hello>
]]>]]>
```

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>
      urn:ietf:params:netconf:base:1.1
    </capability>
    <capability>
      urn:ietf:params:netconf:capability:startup:1.0
    </capability>
    <capability>
      http://example.net/router/2.3/myfeature
    </capability>
  </capabilities>
  <session-id>4</session-id>
</hello>
```

关于 `<hello>` 的详细介绍，请查看 [RFC6241 - Network Configuration Protocol \(NETCONF\) - Capabilities Exchange](#)，[RFC6242 - Using the NETCONF Protocol over Secure Shell \(SSH\) - Starting NETCONF over SSH](#)

## <rpc>

用来封装NETCONF客户端发送给服务端的NETCONF请求，

在 `<rpc>` 中可以有多个attribute，其中 `message-id` 属性是必须存在。它的值由RPC调用者来维护，格式是字符串，通常是一个递增的整数。RPC的接收者不会解码或解释这个字符串，但会将它保存并在接下来的 `<rpc-reply>` 将这个 `message-id` 附加上去。调用方必须确保 `message-id` 值的规范化。

例子：

调用一个名为 `<my-own-method>` 的方法，该方法有两个参数：`<my-first-parameter>`，其值为 `14`，另一个参数的值为 `fred`：

```
<rpc message-id="101"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <my-own-method xmlns="http://example.net/me/my-own/1.0">
    <my-first-parameter>14</my-first-parameter>
    <another-parameter>fred</another-parameter>
  </my-own-method>
</rpc>
]]>]]>
```

调用 `<zip-code>` 参数为 `27606-0100` 的 `<rock-the-house>` 方法：

```
<rpc message-id="101"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <rock-the-house xmlns="http://example.net/rock/1.0">
    <zip-code>27606-0100</zip-code>
  </rock-the-house>
</rpc>
]]>]]>
```

调用不带参数的 NETCONF `<get>` 方法：

```
<rpc message-id="101"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get/>
</rpc>
]]>]]>
```

关于 `<rpc>` 的详细介绍，请查看[RFC6241 - Network Configuration Protocol \(NETCONF\) - \ Element](#)

## `<rpc-reply>`

用来响应 `<rpc>` 消息。

`<rpc-reply>` 必须有一个 `message-id`，它与之前 `<rpc>` 请求中的 `message-id` 应该是一致的。

NETCONF 服务器还必须在 `<rpc-reply>` 中返回 `<rpc>` 请求中未修改的元素以及属性。

`response` 的 `<rpc-reply>` 元素中应该包含为的一个或多个子元素。

例子：

以下 `<rpc>` 元素调用 NETCONF `<get>` 方法，并包含一个名为 `user-id` 的附加属性。返回的 `<rpc-reply>` 元素返回 `user-id` 属性以及请求的内容。

请注意，`user-id` 属性不在 NETCONF 命名空间中。

```

<rpc message-id="101"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
  xmlns:ex="http://example.net/content/1.0" ex:user-id="fred">
  <get/>
</rpc>
]]>]]>

<rpc-reply message-id="101"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
  xmlns:ex="http://example.net/content/1.0"
  ex:user-id="fred">
  <data>
    <!-- contents here... -->
  </data>
</rpc-reply>
]]>]]>

```

关于 `<rpc-reply>` 的详细介绍，请查看[RFC6241 - Network Configuration Protocol \(NETCONF\) - Element](#)

## `<rpc-error>`

如果在处理 `<rpc>` 请求期间发生错误，`<rpc-error>` 元素将在 `<rpc-reply>` 消息中发送。

如果服务器在处理 `<rpc>` 请求期间遇到多个错误，则 `<rpc-reply>` 可能包含多个 `<rpc-error>` 元素。但是，如果请求包含多个错误，则服务器不需要检测或报告多个 `<rpc-error>` 元素。

服务器不需要按照特定的顺序检查特定的错误条件。如果在处理过程中出现任何错误，服务器务必返回一个 `<rpc-error>` 元素。

服务器绝对不能在客户端没有足够访问权限的 `<rpc-error>` 元素中返回应用级或数据模型特定的错误信息。

`<rpc-error>` 元素包含以下信息：

- `error-type`：定义错误发生的概念层，下列类型之一。
  1. `transport` (`layer: Secure Transport`)
  2. `rpc` (`layer: Messages`)
  3. `protocol` (`layer: Operations`)
  4. `application` (`layer: Content`)

- `error-tag` : 包含识别错误条件的字符串。相关的值，请参阅[RFC6261 - Network Configuration Protocol \(NETCONF\) - 附录A](#)。
- `error-severity` : 包含标识错误严重性的字符串，由设备确定。下列之一：
  1. `error`
  2. `warning`

请注意，本文档中没有使用 `warning` 枚举定义的值。这是保留供将来使用。
- `error-app-tag` : 包含标识数据模型特定或特定于实现的错误条件（如果存在）的字符串。如果没有适当的应用程序错误标签可以与特定的错误条件相关联，则这个元素将不存在。如果数据模型特定的和特定于实现的 `error-app-tag` 都存在，那么服务器必须使用数据模型特定的值。
- `error-path` : 包含绝对[XPath](#)表达式，用于标识 `<rpc-error>` 元素中报告的错误关联的节点的元素路径。如果没有适当的有效载荷元素或数据存储节点可以与特定的错误条件相关联，则该元素将不存在。
  - `XPath` 表达式在以下上下文中进行解释。
    - 名称空间声明的集合是 `<rpc-error>` 元素上的范围声明。
    - 变量绑定的集合是空的。
    - 函数库是核心函数库。
  - 上下文节点取决于与所报告的错误相关联的节点：
    - 如果 `payload` 元素可以与错误关联，则上下文节点是 `rpc` 请求的文档节点（即 `<rpc>` 元素）。
    - 否则，上下文节点是所有数据模型的根，即所有数据模型中具有作为子节点的顶级节点的节点。
- `error-message` : 包含一个适合人类阅读的字符串，用于描述错误情况。如果没有为特定错误条件提供适当的消息，则该元素将不存在。这个元素应该包含一个在[\[W3C.REC-xml-20001006\]](#)中定义并在[\[RFC3470\]](#)中讨论的 "`xml:lang`" 属性。
- `error-info` : 包含协议或数据模型特定的错误内容。如果没有为特定的错误条件提供这样的错误内容，则该元素将不存在。[RFC6241 - Network Configuration Protocol \(NETCONF\) - 附录A](#)中的列表定义了每个错误的任何强制错误信息内容。在任何协议规定的内容之后，数据模型定义可能会要求某些应用层错误信息包含在错误信息容器中。一个实现可以包含额外的元素来提供扩展和/或实现特定的调试信息。

例子：

```

<rpc-reply xmlns="URN" xmlns:junos="URL">
  <rpc-error>
    <error-severity>error-severity</error-severity>
    <error-path>error-path</error-path>
    <error-message>error-message</error-message>
    <error-info>...</error-info>
  </rpc-error>
</rpc-reply>
]]>]]>

```

如果接收到元素而没有 message-id 属性，则返回错误。

请注意，只有在这种情况下，NETCONF 对等方可以省略 `<rpc-reply>` 元素中的 message-id 属性。

```

<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
  </get-config>
</rpc>
]]>]]>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <rpc-error>
    <error-type>rpc</error-type>
    <error-tag>missing-attribute</error-tag>
    <error-severity>error</error-severity>
    <error-info>
      <bad-attribute>message-id</bad-attribute>
      <bad-element>rpc</bad-element>
    </error-info>
  </rpc-error>
</rpc-reply>
]]>]]>

```

## RFC6241 - Network Configuration Protocol (NETCONF) - Element

### `<error-info>`

提供导致NETCONF服务器错误或警告的事件会附加在 `<rpc-error>` 标签内告知请求者。

`<bad-element>` - 识别发生错误或警告时正在处理的命令或配置语句。对于相关的配置声明，会在 `<rpc-error>` 标记元素中包含的 `<error-path>` 标记元素指定语句的父层次结构级别。

例子：

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <rpc-error>
    <error-info>
      <bad-element>command-or-statement</bad-element>
    </error-info>
  </rpc-error>
</rpc-reply>
]]>]]>
```

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
  </get-config>
</rpc>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <rpc-error>
    <error-type>rpc</error-type>
    <error-tag>missing-attribute</error-tag>
    <error-severity>error</error-severity>
    <error-info>
      <bad-attribute>message-id</bad-attribute>
      <bad-element>rpc</bad-element>
    </error-info>
  </rpc-error>
</rpc-reply>
```

**</ok>**

如果在处理 `<rpc>` 请求期间没有发生错误或警告，并且操作没有返回任何数据，则 `<rpc-reply>` 消息中会发送 `<ok>` 元素。

例子：

```
<rpc-reply message-id="101"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
]]>]]>
```

**<target>**

指定要在其上执行操作的配置。

如果客户端应用程序在对目标 `<candidate />` 执行 `<copy-config>`，`<delete-config>` 或 `<edit-config>` 操作之前发出 `<open-configuration>` 操作来打开特定的配置数据库，将在打开的配置数据库上执行请求的操作。否则，在候选配置上执行操作。客户端应用程序只能对候选配置执行 `<lock>` 和 `<unlock>` 操作。

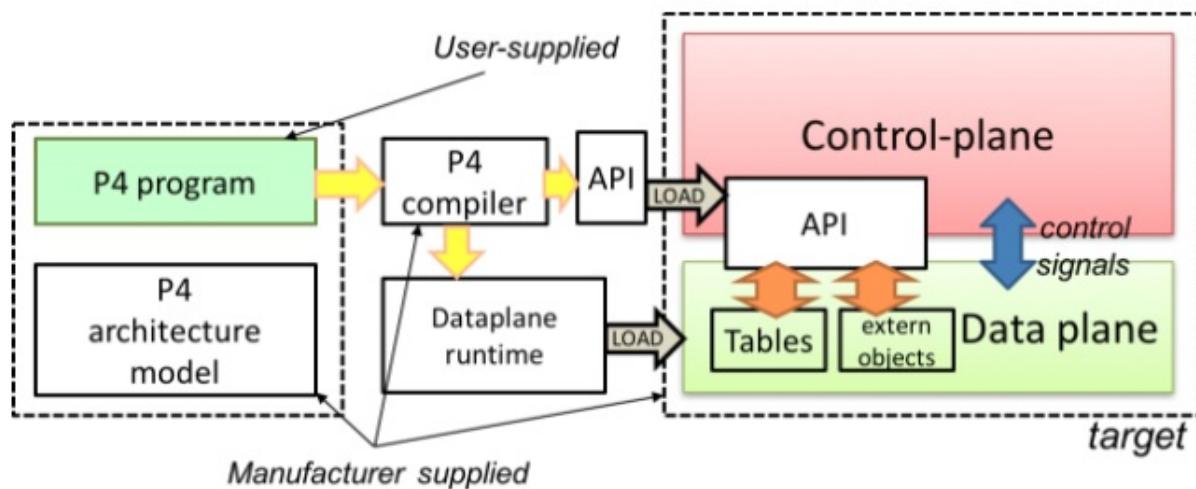
`<candidate />` - 指定要在其上执行操作的配置，打开配置数据库，或者如果没有打开的数据  
库，则为候选配置。

```
<rpc>
  <( copy-config | delete-config | edit-config | lock | unlock )>
    <target>
      <candidate/>
    </target>
  </(>
</rpc>
]]>]]>
```

# 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

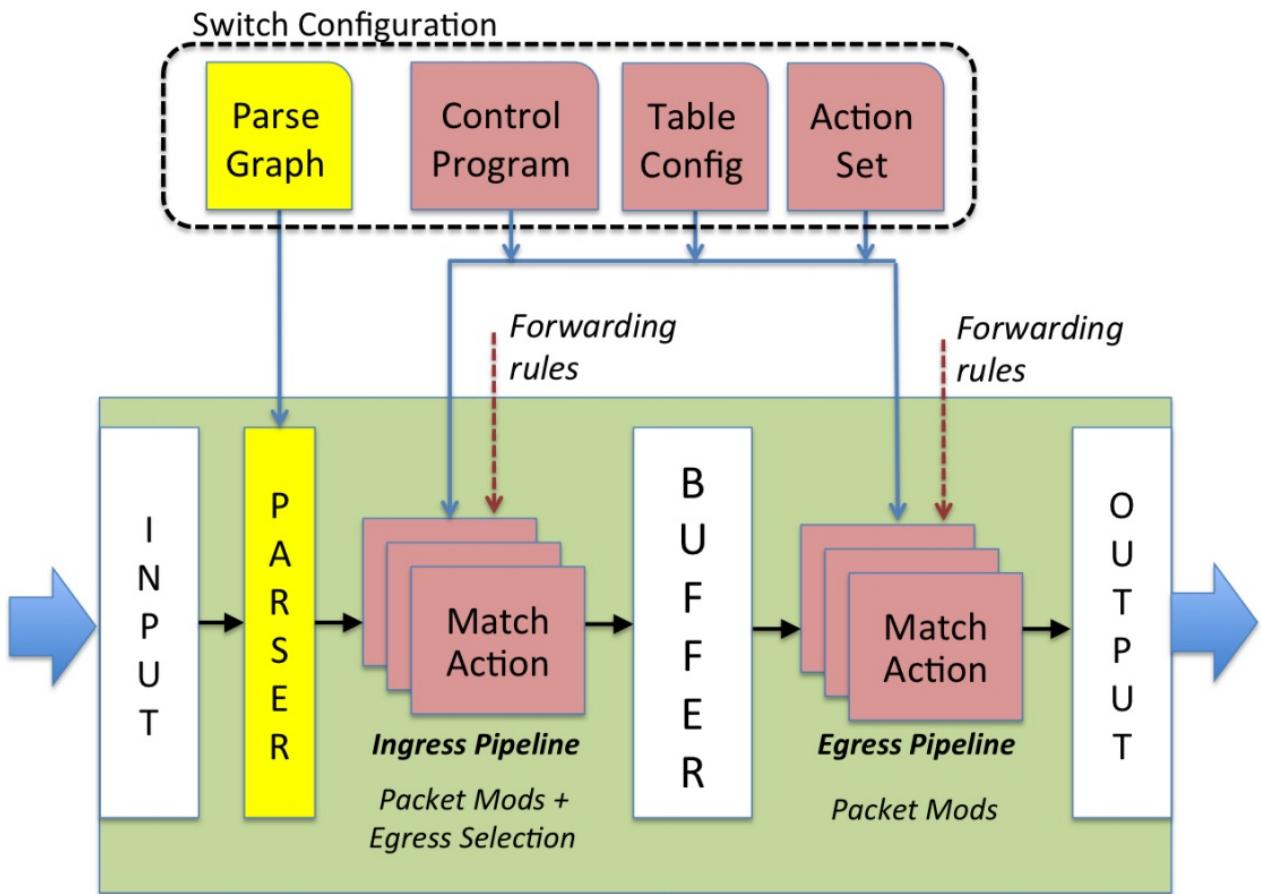
- 灵活定义数据转发流程，支持重新配置匹配域，并支持无中断的重配置
- 协议无关，只需要关注如何处理转发包，而不需要关注底层协议细节
- 设备无关，无需关注底层设备的具体信息



## 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：

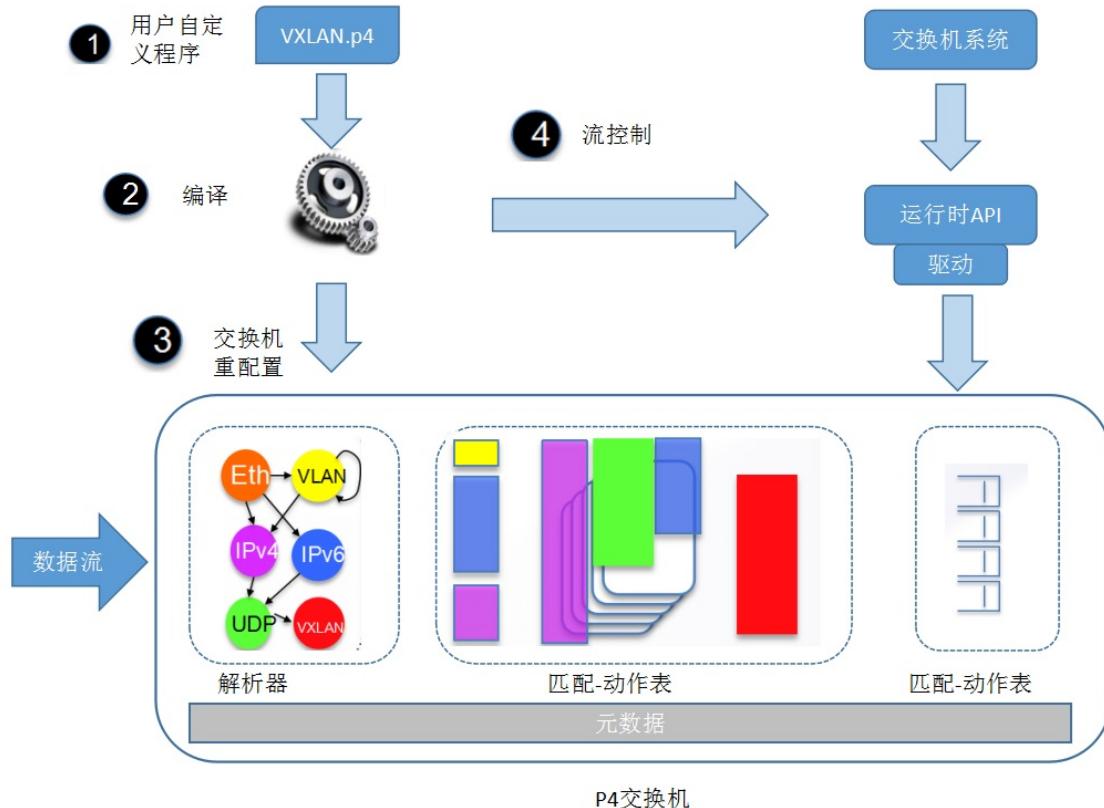
- Headers：定义报文头部格式，支持重定义的头部名称和任意长度字段
- Parses：定义数据包解析流程的有限状态机
- Tables：“匹配-动作”表，定义匹配域以及对应的执行动作
- Actions：动作指令集，包括构造查找键（Construct lookup keys）、根据查找键查表、执行动作等
- Control Flow：控制程序，决定了数据包处理的流程，比如如何在不同表之间跳转等

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。

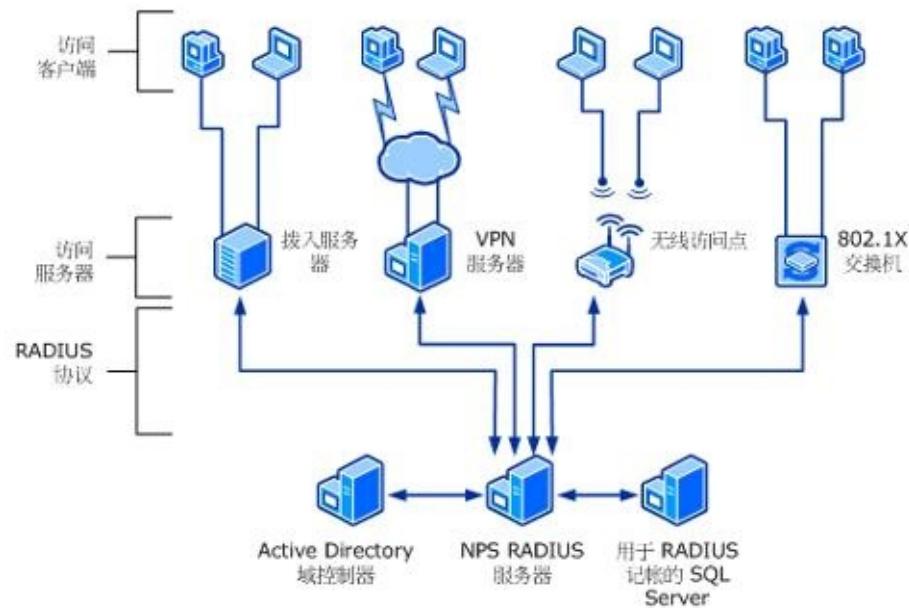
## 参考文档

- P4
- P4:开创数据平面可编程时代

# RADIUS

## 简介

RADIUS 的全称是 Remote Authentication Dial in User Service , 翻译过来是“远程用户拨号认证系统”。



常见Radius结构

## RADIUS和AAA

RADIUS 是由 RFC2865 , RFC2866 来定义的，是目前应用最广泛的 AAA 协议。

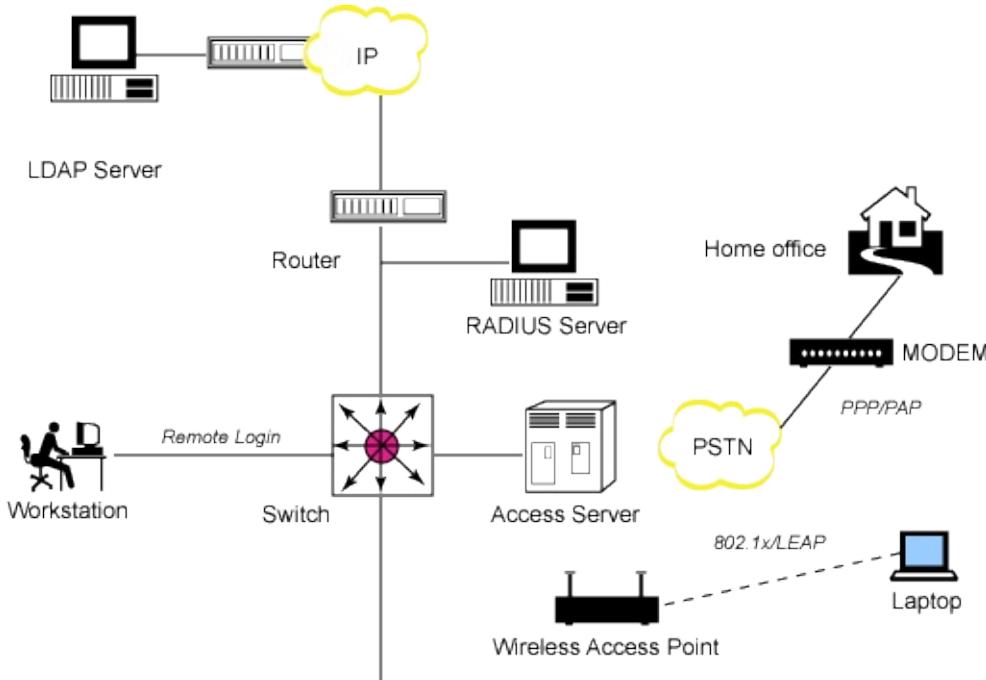
RADIUS 是一种 C/S 结构的协议，他的客户端最初就是 NAS ( Net Access Server ) 服务器，现在任何运行 RADIUS 客户端的软件的计算机都可以成为 RADIUS 的客户端。

如果 NAS 收到用户连接请求，它会将它们传递到指定的 RADIUS 服务器，后者对用户进行验证，并将用户的配置信息返回给 NAS 。然后， NAS 接受或拒绝连接请求。 RADIUS 协议认证机制灵活，功能完整的 RADIUS 服务器可以支持很多不同的用户验证机制，除了 LDAP 以外，还包括：

- PAP ( Password Authentication Protocol , 密码验证协议，与 PPP 一起使用，在此机制下，密码以明文形式被发送到客户机进行比较) ；
- CHAP ( Challenge Handshake Authentication Protocol , 挑战握手验证协议，比 PAP 更安全，它同时使用用户名和密码) ；

- 本地 UNIX/Linux 系统密码数据库（/etc/passwd）；
- 其他本地数据库。

通过Radius和LDAP组合进行认证



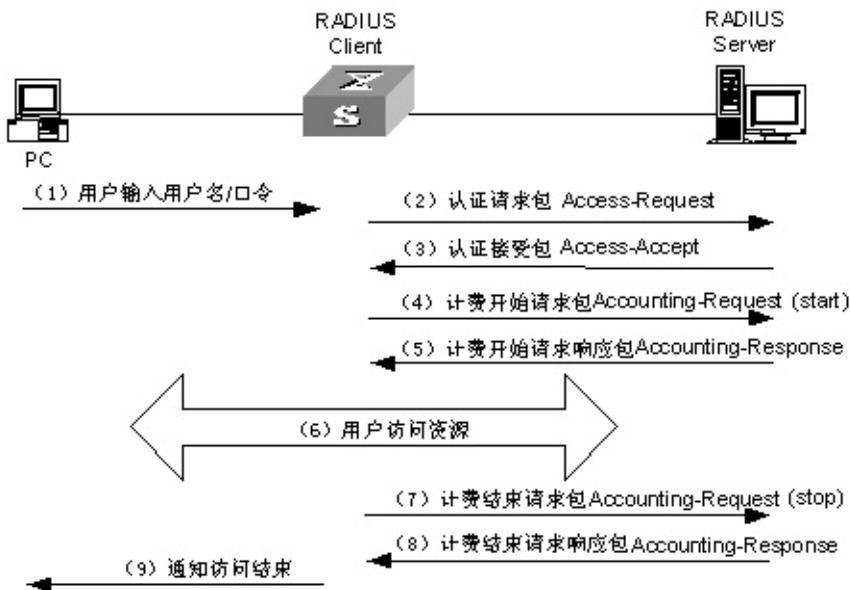
在 **RADIUS** 中，验证和授权是组合在一起的。如果发现了用户名，并且密码正确，那么 **RADIUS** 服务器将返回一个 **Access-Accept** 响应，其中包括一些参数（属性-值对），以保证对该用户的访问。这些参数是在 **RADIUS** 中配置的，包括访问类型、协议类型、用户指定该用户的 **IP** 地址以及一个访问控制列表（**ACL**）或要在 **NAS** 上应用的静态路由，另外还有其他一些值。

**RADIUS** 记帐特性（在 [RFC 2866](#) 中定义）允许在连接会话的开始和结束发送数据，表明在会话期间使用的可能用于安全或开单（**billing**）需要的大量资源——例如时间、包和字节。

**RADIUS** 是一种可扩展的协议，它进行的全部工作都是基于 **Attribute-Length-Value** 的向量进行的。**RADIUS** 也支持厂家专有属性。

**IEEE** 提出了 **802.1x** 标准，这是基于端口的标准，用于对无线网络的接入认证，在认证时，也是采用了 **RADIUS** 协议。

## 基本工作原理



## 认证过程

1. 用户接入 NAS，NAS 向 RADIUS 服务器使用 Access-Request 数据包提交用户信息，包括用户名、密码等相关信息，其中用户密码是经过 MD5 加密的，双方使用共享密钥，这个密钥不经过网络传播
2. RADIUS 服务器对用户名和密码进行校验，必要时可以提出一个 Challenge，要求进一步对用户认知，也可以对 NAS 进行类似认证；如果合法，给 NAS 返回 Access-Accept 数据包，允许用户进行下一步工作，否则返回 Access-Reject 数据包，拒绝用户访问；
3. 如果允许访问，NAS 向 RADIUS 服务器提出计费请求 Account-Request，RADIUS 服务器响应 Account-Accept，对用户的积分开始，同事用户可以进行资金的相关操作。

## 代理和漫游

RADIUS 还支持代理和漫游功能。

简单来说，代理就是一台服务器，可以作为其他Radius服务器的代理，负责转发 RADIUS 认证和计费数据包。

所谓漫游功能，就是代理的一个具体实现，这样可以让用户通过本来与其无关的 RADIUS 服务器进行认证，用户到非归属运营商所在地也可以得到抚慰，也可以实现虚拟运营。

## 通信协议

Radius 服务器和 NAS 服务器通过 UDP 协议进行通信，Radius 服务器的 1812 端口负责认证，1813 端口负责计费工作。

采用 UDP 的基本考虑是因为 NAS 和 RADIUS 服务器大多数在同一个局域网中，使用 UDP 更加快捷方便。

`Raduis` 协议还规定了重传机制。

如果 `NAS` 向某个 `RADIUS` 服务器发起请求没有收到返回信息，那么可以要求备份 `Raduis` 服务器重传。由于有多个备份 `RADIUS` 服务器，因此 `NAS` 进行重传的时候，可以采用轮询的方法。

如果备份 `RADIUS` 服务器的密钥和以前的 `RADIUS` 服务器的密钥不同，则需要重新进行认证。

## 协议结构

```
-----  
1byte | 1byte | 2bytes  
-----  
Code | Identifier | Length  
-----  
Authenticator (16 bytes)  
-----  
Attributes ...  
-----
```

- `Code` - 信息类型如下：
  - 1、请求方法(`Access-Request`);
  - 2、接收访问(`Access-Accept`)
  - 3、拒绝访问(`Access-Reject`)
  - 4、计费请求(`Accounting-Request`)
  - 5、计费响应(`Accounting-Response`)
  - 11、挑战访问(`Access-Challenge`)
  - 12、服务器状况(`Status-Server - Experimental`)
  - 13、客户端状况(`Status-Client - Experimental`)
  - 255、预留(`Reserved`)
- `Identifier` - 匹配请求和响应的标识符。
- `Length` - 信息大小，包括头部
- `Authenticator` - 该字段用来识别 `RADIUS` 服务器和隐藏口令算法中的答复。

## 基本消息交互流程

`RADIUS` 服务器对用户的认证过程通常需要利用 `NAS` 等设备的代理认证工，`RADIUS` 客户端和 `RADIUS` 服务器之间通过共享密钥认证相互的消息，用户密码采用密文方式在网络中传输，增强安全性。

`RADIUS` 协议合并了认证和授权的过程，即响应报文中携带了授权信息。

---

基本交互步骤如下：

1. 用户输入用户名和口令
2. RADIUS 客户端根据获取的用户名和口令，向 RADIUS 服务器发送认证请求包( access-request ).
3. RADIUS 服务器将该用户信息与 users 数据库信息进行对比分析，如果认证成功，则将用户权限信息以认证响应包( access-accept )发送给 RADIUS 客户端，如果认证失败，则返回 access-reject 响应包。
4. RADIUS 客户端根据接受到的认证结果接入/拒绝用户。如查询可以接入用户，则 RADIUS 客户端向 RADIUS 服务器发送计费开始请求包( accounting-request ), status-type 取值为 start
5. RADIUS 服务器返回计费开始响应包( accounting-response )
6. RADIUS 客户端向 RADIUS 服务器发送计费停止请求包( accounting-request ), status-type 取值为 stop
7. RADIUS 服务器返回计费结束响应包( accounting-response )

## 扩展阅读

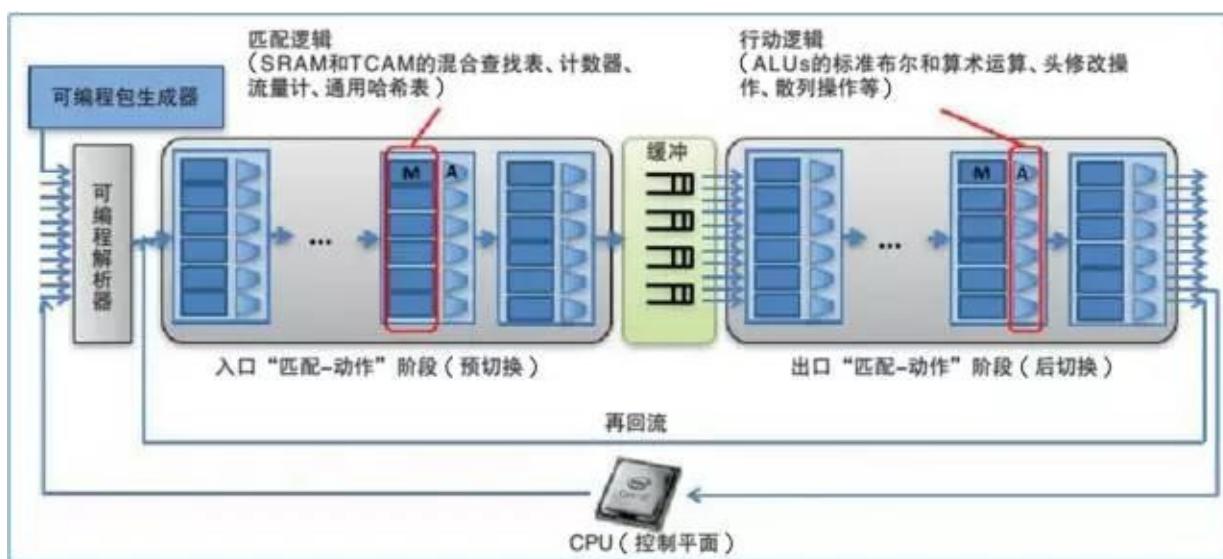
- [RFC2865-Remote Authentication Dial In User Service \(RADIUS\)](#)
- [RFC2866-RADIUS Accounting](#)
- [在 Linux 上构建一个 RADIUS 服务器](#)
- [Radius 认证协议介绍](#)

# 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。

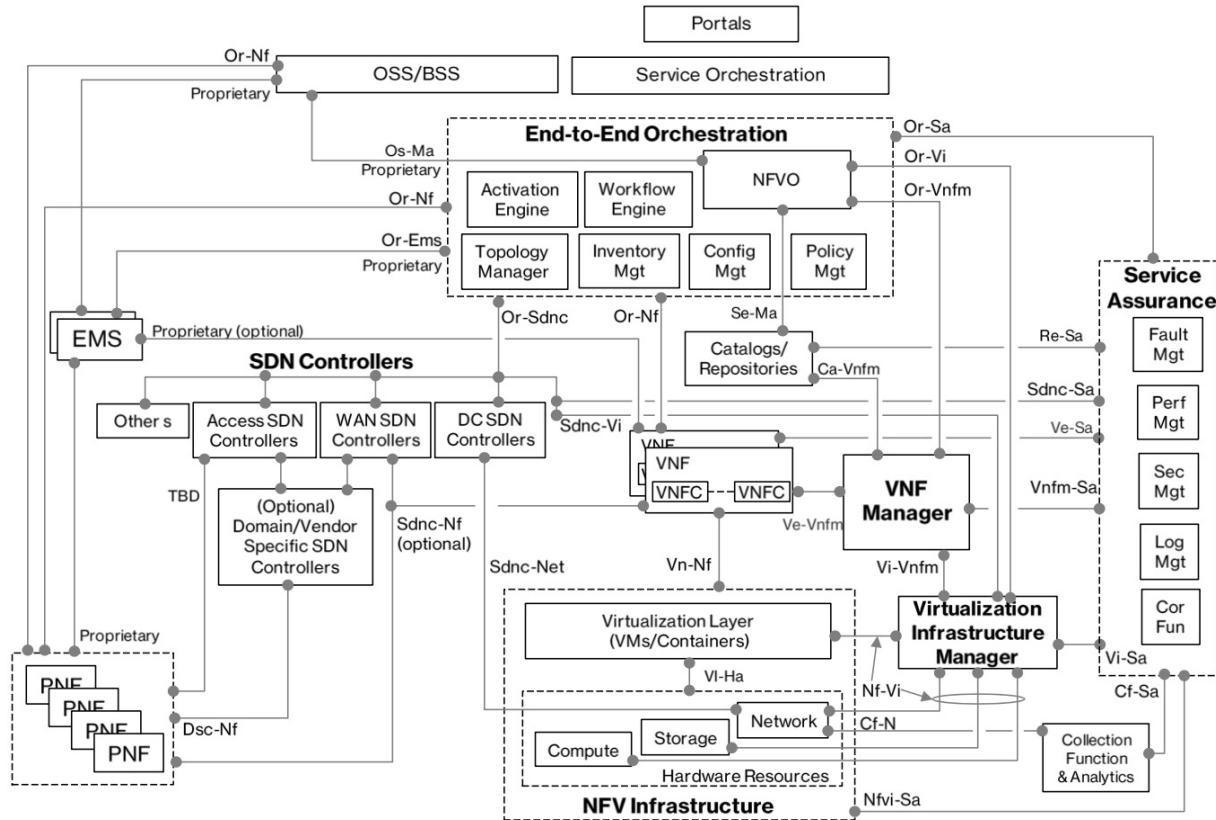
同SDN一样，NFV从根本上讲是从基于硬件的解决方案转向更开放的基于软件的解决方案。例如，取代专用防火墙设备，软件可以通过虚拟防火墙提供相同的功能。再如入侵检测和入侵防御、NAT、负载均衡、WAN加速、缓存、GGSN、会话边界控制器、DNS等等虚拟网络功能。有时，不同的子功能可以组合起来形成一个更高级的多组件VNF，如虚拟路由器。

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

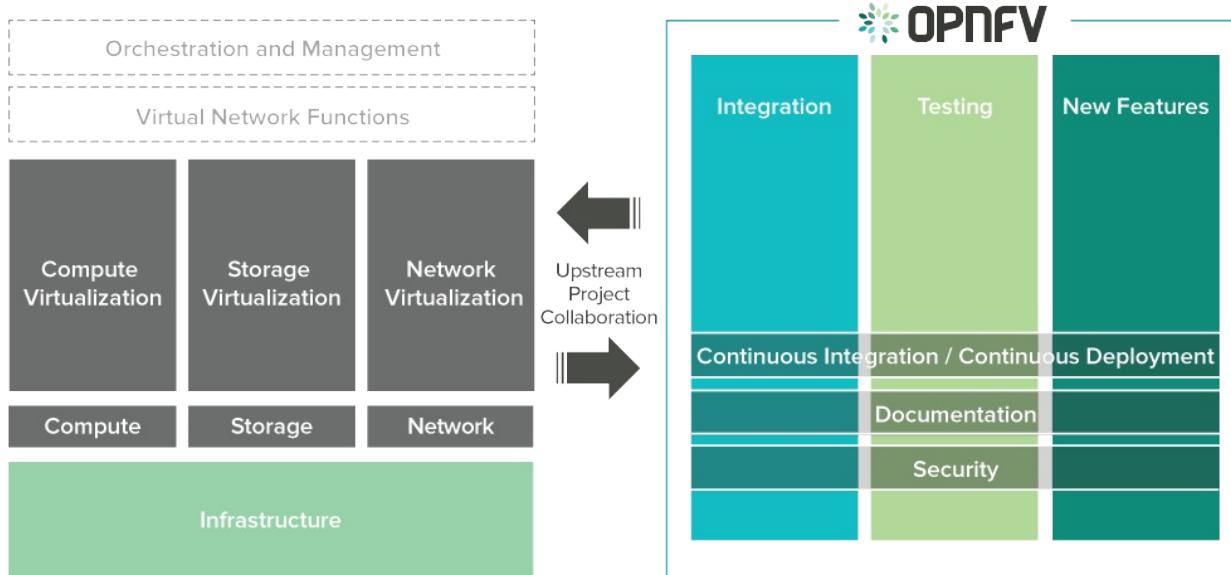
## NFV架构



- 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/>

## 8. NFV

---

- <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。



# 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

# OpenStack Neutron

OpenStack Neutron 实践。

# Google Data Center Networks

## 1. B4

Google B4 是第一个成功的 SD-WAN 应用案例，它也是最出名的 SDN 案例。

B4 网络承载了 Google 数据中心 90% 的内部应用流量。

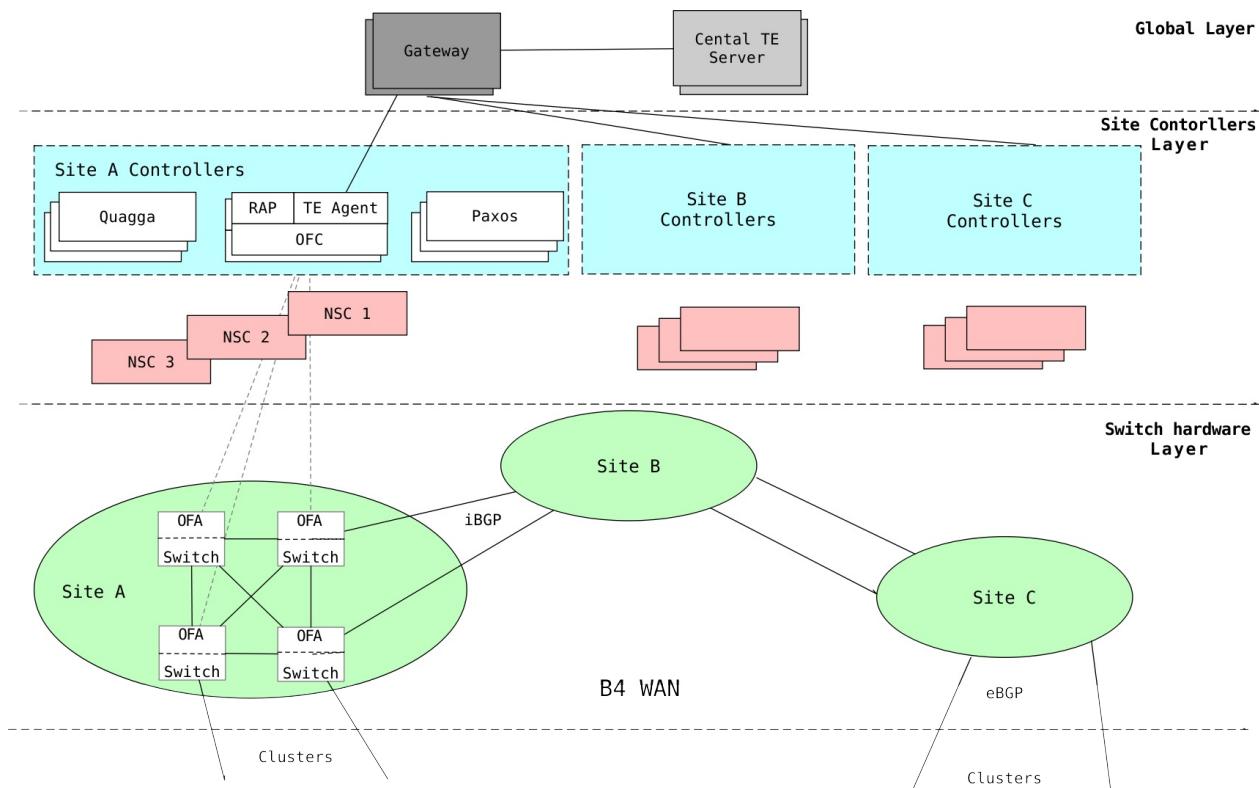
B4 网络具有流量大、突发性强、周期性强等特定，需要网络具备多路径转发与负载均衡，网络带宽动态调整等能力。

B4 面临的问题是网络流量分布的不均匀，高峰期流量达到平时流量的 2-3 倍，而且不同类型的网络流量的数据量、延时要求和优先级都不同。所以，邀请 B4 满足弹性带宽获取( elastic bandwidth demands )、大规模站点( moderate number of sites )、应用端控制( end application control )、低成本( cost sensitivity )等要求。

B4 的 SDN 架构分为交换机硬件层( switch hardware layer )、站点控制层( site controller layer )和全局控制层( global layer )三大部分。

Google B4 采用 SDN 架构之后，其 WAN 的链路层利用率从 30%~40% 提升到 90% 以上，效果非常显著。

B4 是第一个基于 SDN 架构的 WAN 网络部署案例，其设计思路、实现方案和真实部署给 WAN 领域的应用提供了非常重要的参考价值。

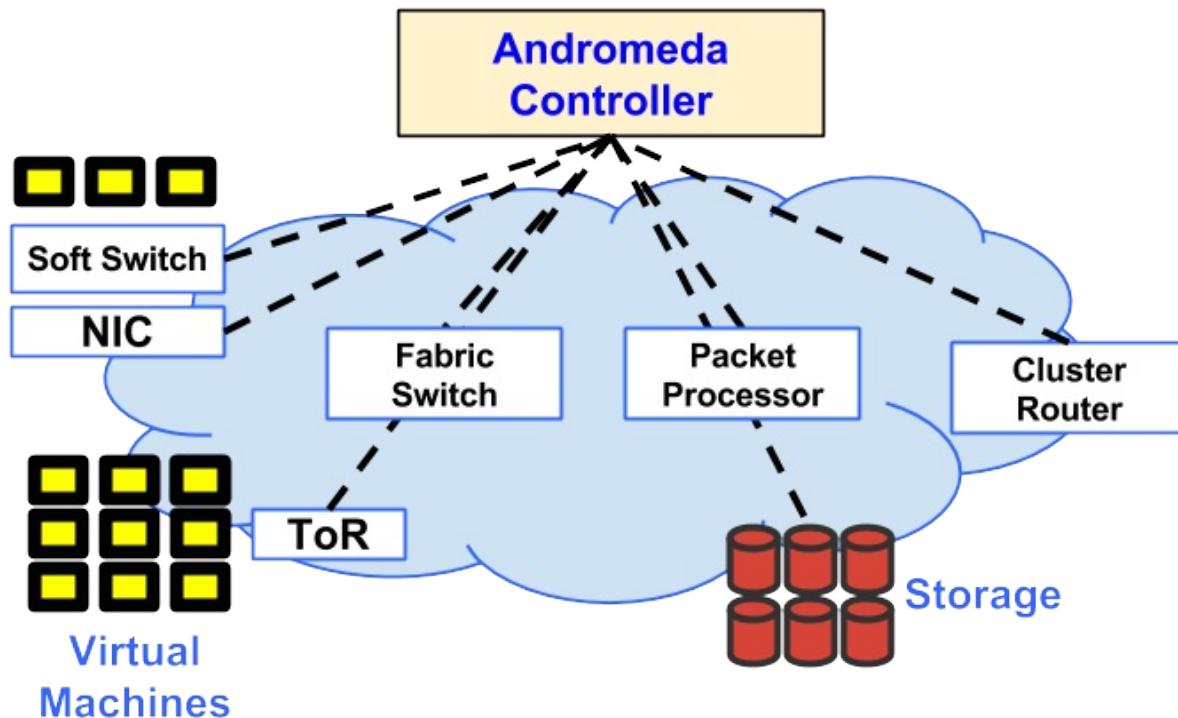


## 2. Jupiter

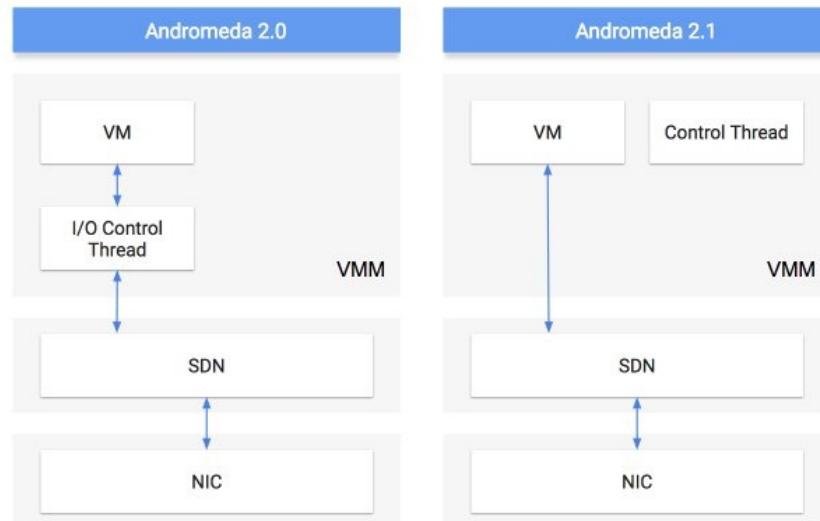
Google 通过 SDN 的途径来构建 Jupiter , Jupiter 是一个能够支持超过10万台服务器规模的数据中心互联架构。它支持超过 1 Pb/s 的总带宽来承载其服务。

## 3. 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。



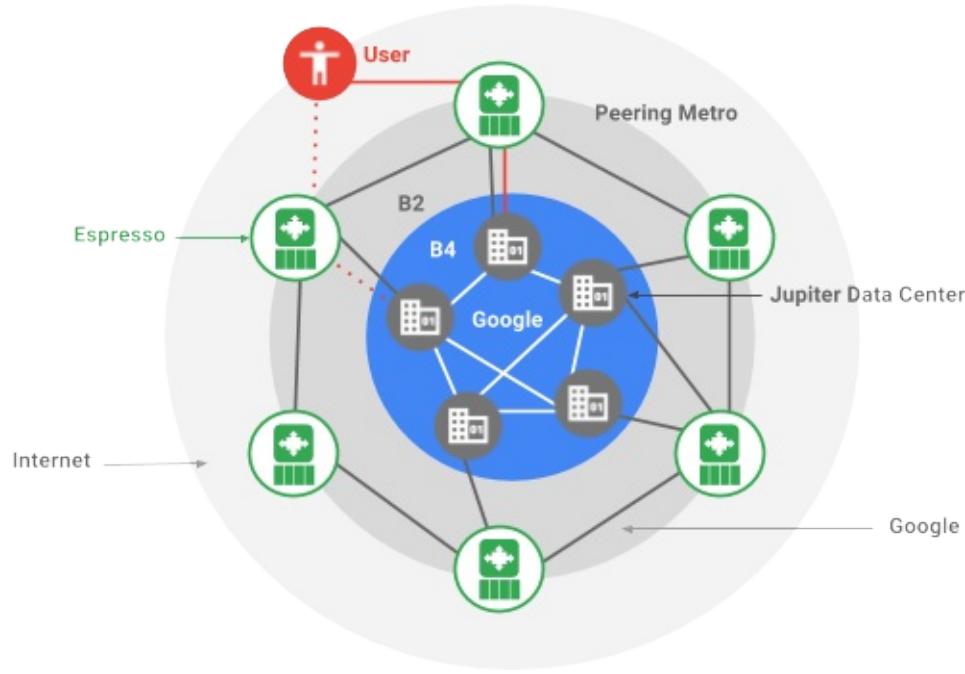
@googlecloud tweet Andromeda 2.1's optimized datapath using hypervisor bypass.



Andromeda 2.1 reduces GCP's intra-zone latency by 40%

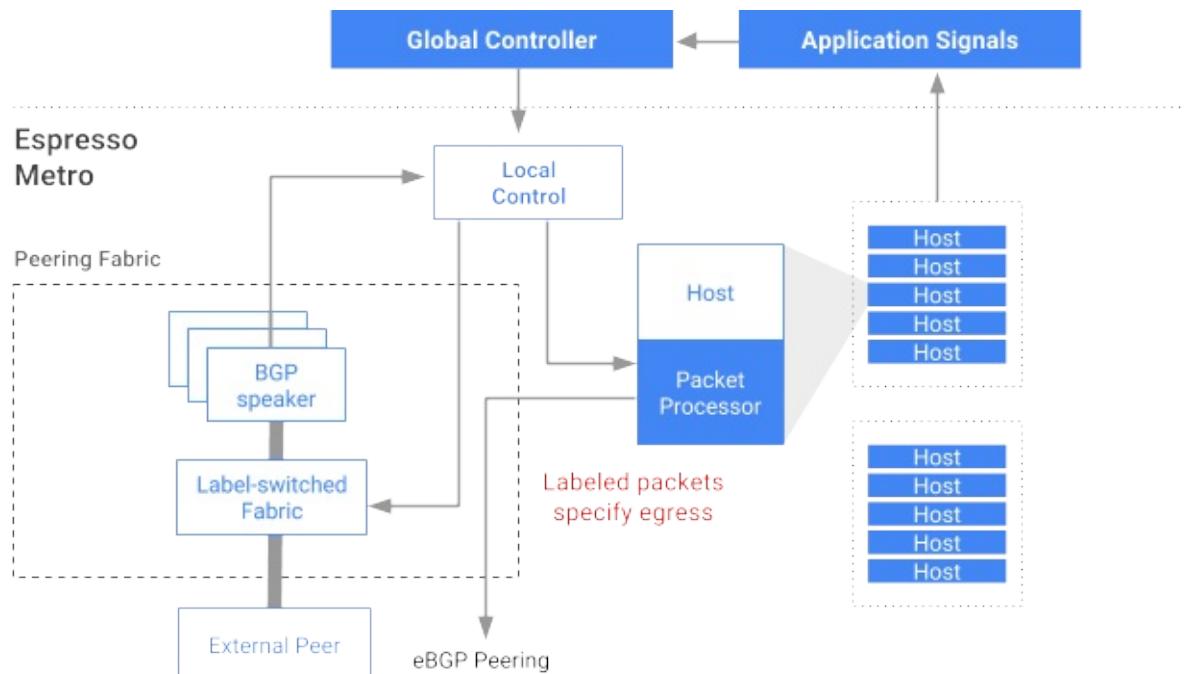
## 4. Espresso

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。”

**Espresso** 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。



## 参考链接

- [A look inside Google's Data Center Networks](#)

- Enter the Andromeda zone - Google Cloud Platform's latest networking stack
- Espresso makes Google cloud faster, more available and cost effective by extending SDN to the public internet
- b4-sigcomm

## 业务示例

待补充

# SDN控制器应用场景

## 场景描述

广义的SDN技术门类非常丰富，可以广泛应用于电信运营商的不同网络层次和业务领域，相关场景包括：

传输网中的：

- 光路自动适配
- 光网络可视化
- IP/传输资源协同等相关业务
- IP骨干/城域网中的流量工程
- 个人/VPN流量差异化承载
- 虚拟化BNG
- ...相关业务

移动网络中的：

- IP RAN资源调度
- 虚拟EPC网元
- Gi-LAN
- ...相关业务

接入网中的：

- WLAN资源调度
- 虚拟化家庭网络
- 虚拟化家庭网关
- ...相关业务

IDC网络中的：

- 虚拟化资源优化
- 网络优化
- DCI
- ...相关业务

由于这些专业网络与业务领域需要的控制功能和控制逻辑差异巨大，因此，上述领域的控制器大多应该独立设计并独立管理各自的网元设备。

从SDN控制器的设计角度来看，依照当前SDN进展和技术分类状况，SDN控制器的应用场景可以分为3类：

- SDN过渡技术控制平面
- NFV控制平面
- OpenFlow控制平面

## SDN过渡技术控制平面

基于传统路由协议的扩展或改进，无需新增任何新型设备，通过分离和公开控制平面、增加网络资源可视化能力的方式，实现SDN承载与控制分离的思路。

具体场景包括：

- 基于各类BGP扩展协议的骨干/城域网集中式流量优化调度
- DCI流量流向优化
- IP RAN与WLAN资源优化等

控制器需要实现 资源可视化、资源调度算法、路由计算、业务**API** 等功能。

## NFV控制平面

为NFV设施提供一个公共的控制器，提供全局的资源优化分配能力。

具体场景包括

- 基于x86服务器的vEPC、vBRAS、vCPE等设备的集中控制器

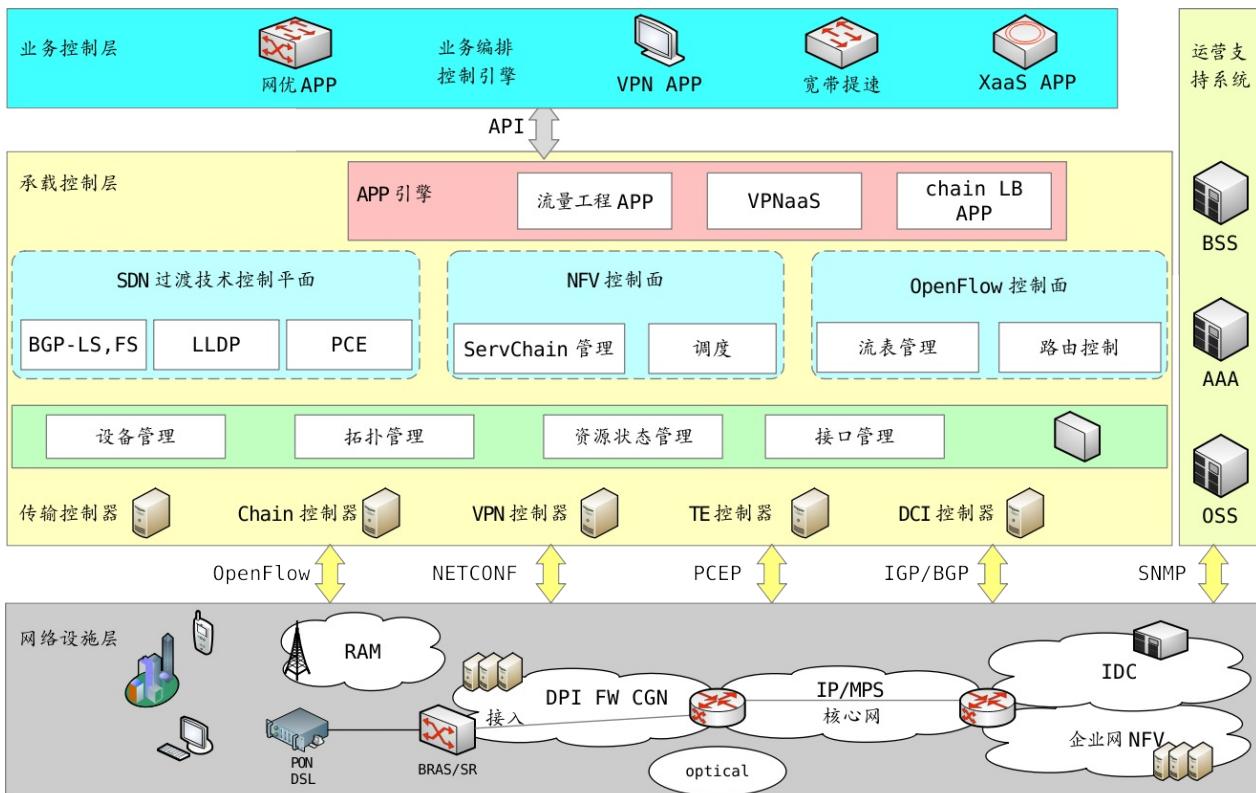
控制器需要完成 流量分类、路由、拓扑发现、资源调度 等功能。

## OpenFlow控制平面

在职场OpenFlow协议的全新网络、OpenFlow与传统网络叠加的混合网络中，通过集中控制器提供更为灵活的调度能力。

控制器需要提供OpenFlow交换机所需要的 流表存储与管理、用户级业务控制、资源拓扑、资源优化调度 以及 **packet-in** 等功能。

## SDN在传统网络中的应用架构



### SDN技术在电信网络中的应用架构

为了实现对网络及其业务的智能控制，实现“智能管道”能力，于是将运营商广域网分为网络设施层、承载控制层、业务控制层3层，并对相对功能进行了重新划分和解耦。

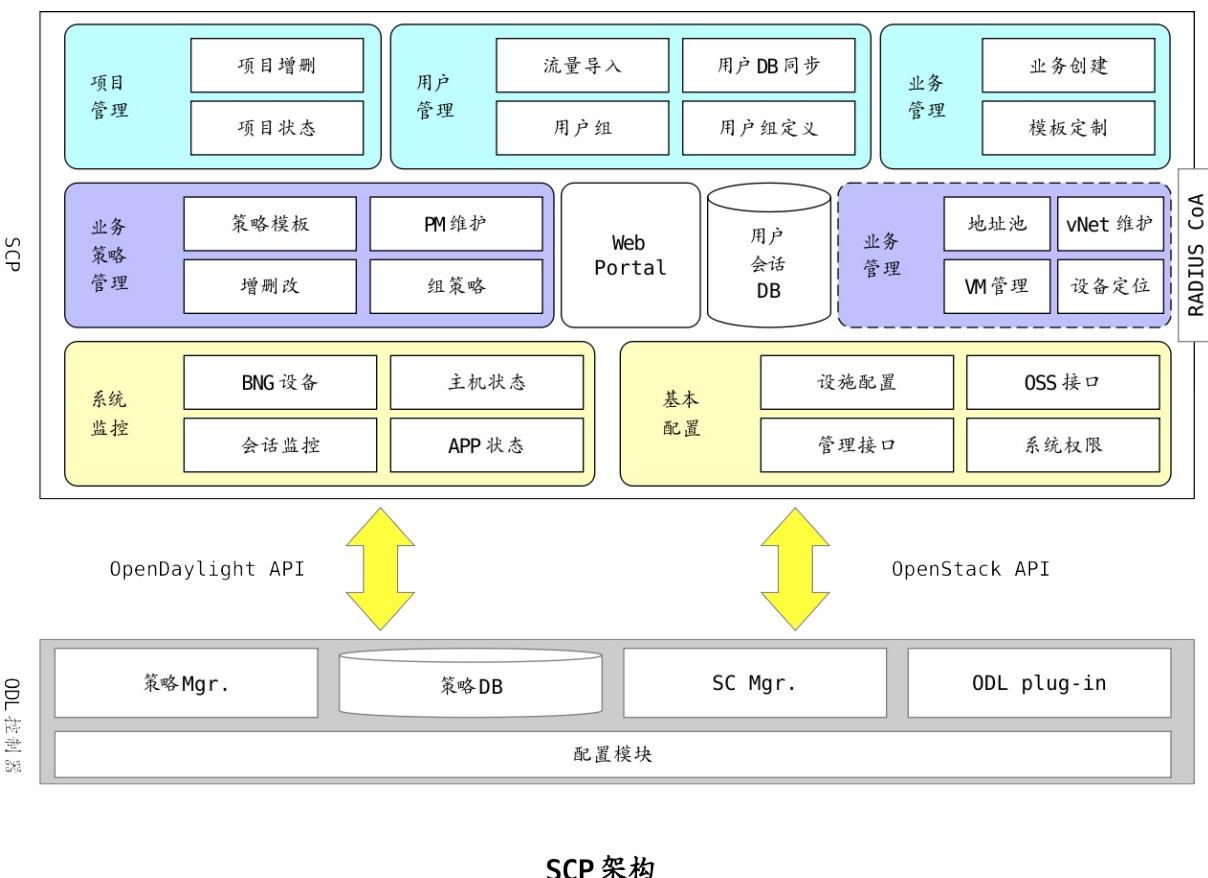
在实际应用过程中，SDN控制器需要连接运营支持系统、业务编排器、南向的网元设定3类系统，并开发大量接口，才能完成对承载网络和业务的全面控制。

# 业务控制平台 (Service control Platform, SCP)

SCP是处于网络边缘业务技术方案的业务控制层，负责管理接入用户的业务定制、业务策略、基础资源监控、运营支撑系统接口等功能。

## SCP参考架构

在本方案中，SCP提供业务定制、用户管理以及资源管理等功能，提供与OSS、AAA等运营支撑系统的接口，提供ODL控制器层、OpenStack资源管理成的对接能力。

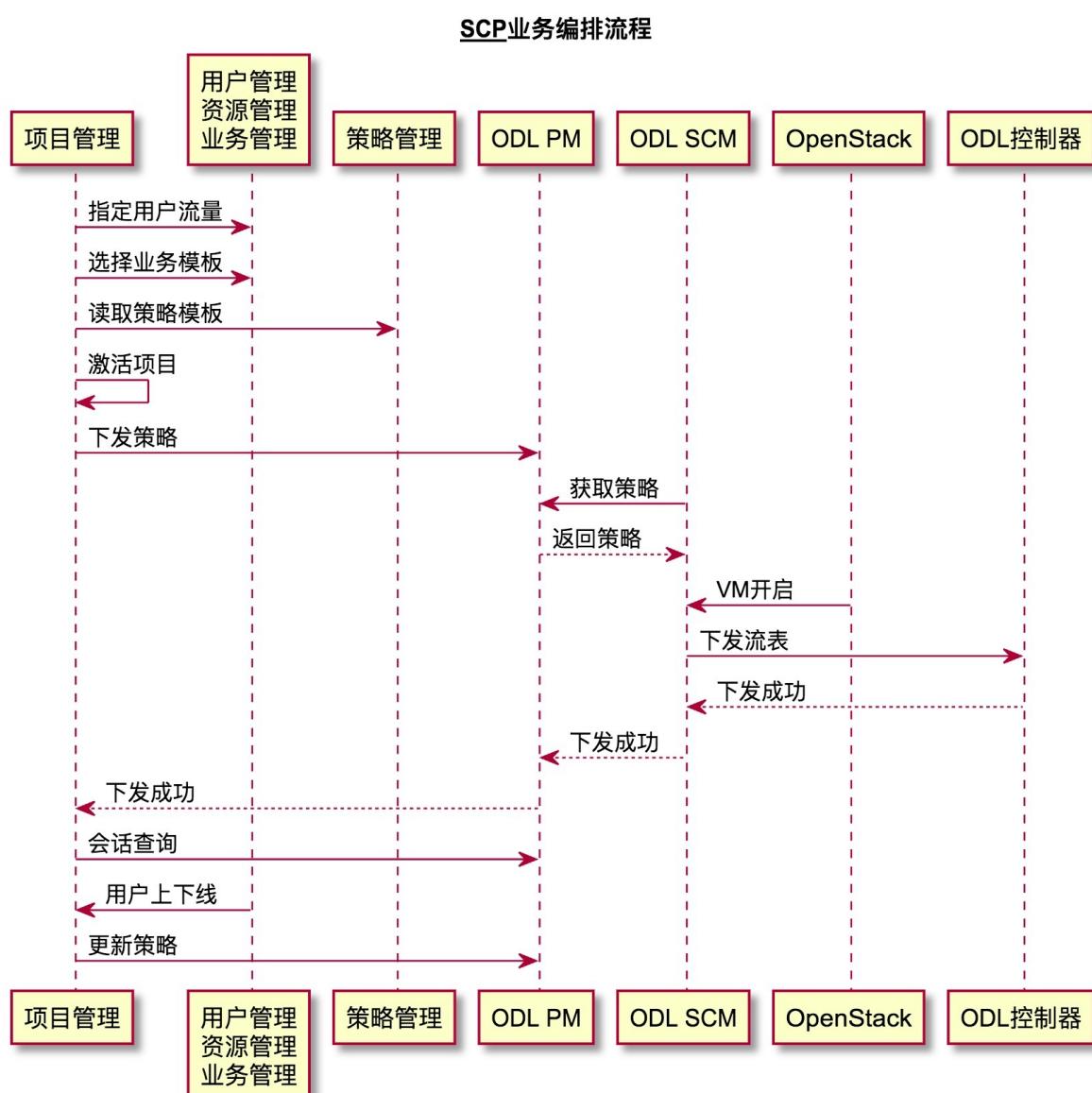


## SCP模块说明

1. **项目管理(Projects):** 管理用户和业务的捆绑关系，实现业务控制能力和业务策略向资源需求的映射，提供项目激活和去激活的操作界面。
2. **用户管理(Subscriber):** 管理用户（资源使用者），提供所有接入用户的SLA配置及用户组定义功能，实现同步用户上下线状态。

3. 业务管理(Service)：管理业务与业务模板，通过封装给中策略，实现各类负责业务配置界面和抽象处理能力管理配置。
4. 策略管理(Policy)：管理策略与组策略，提供基本转发策略、带宽策略、业务策略、PF策略等配置功能。
5. 资源池管理(Resource)：管理资源池，提供全局业务链资源模型、全局资源视图，包括技术资源、VNF状态、物理/虚拟网络资源状态。
6. 系统监控(Monitor)：监控资源池，通过主机上安装 NETCONF 代理实现实时监控，提供资源池的各类资源的实时状态监控和历史记录。
7. 基础配置(System Config)：管理基础配置，提供设施配置、OSS接口配置、系统语言配置以及系统用户权限管理。
8. Web使用平台(Web Portal)：提供用户在线定制各种功能的接口，与AAA服务器以及用户管理模块共同完成。

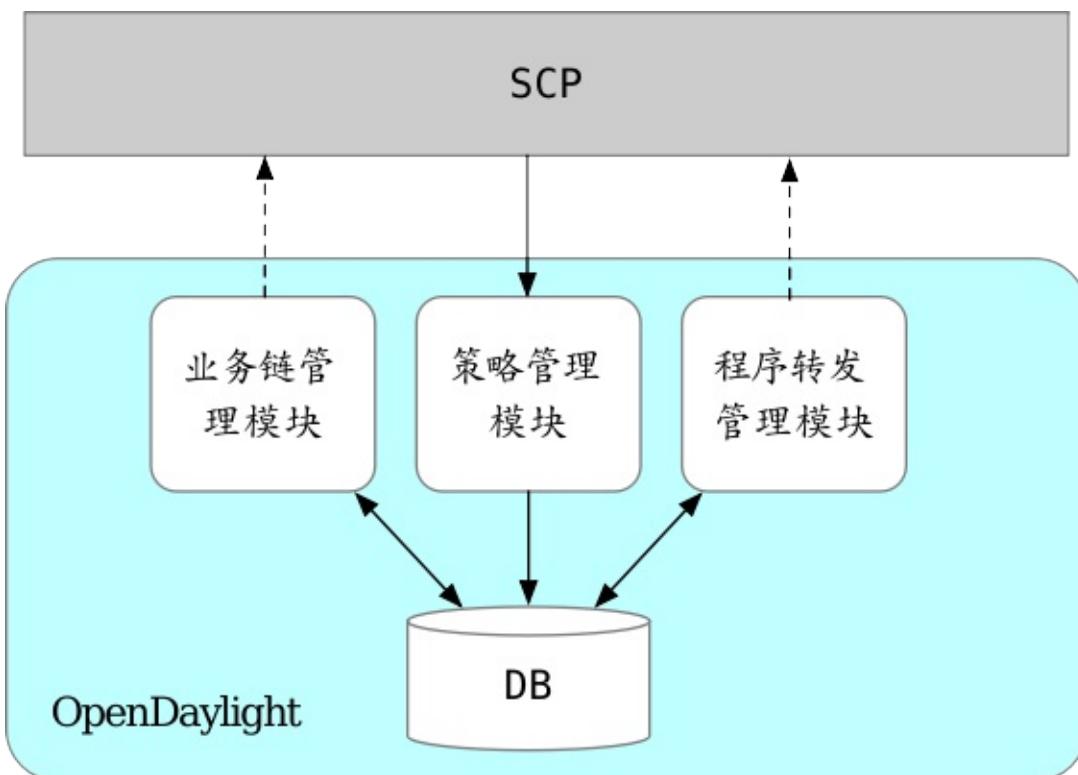
## SCP业务流程编排



SCP的业务编排流程涉及整个方案中的插件交互。

1. 在创建项目之前，必须进行一系列的配置，包括用户组定义或导入(用户管理模块)、策略及其模板(策略管理模块)、业务及其模板(业务管理模块)、资源中的虚拟网络以及VM(资源管理模块)等。
2. 在创建项目时，需要绑定用户组、资源组以及业务套餐。即表示绑定的用户会按照订购的业务套餐，在资源池上分配相应的资源。例如订购在线杀毒服务的用户组的网络流量会引导至提供杀毒服务的VM，其服务链路径由业务套餐决定。
3. 在创建项目之后，需要激活相关的项目，策略才能真正生效。项目成功激活后，SCP将从BRAS监听到用户组内的用户状态，并监控会话的变化：用户上线会产生新的会话，用户下线将会删除相应的会话。

## 系统接口



## SCP 与 OpenDaylight 系统接口

### 添加策略

- URL

```
http://localhost:port/controller/nb/v2/policies
```

- HTTP Method :

```
POST
```

- HTTP Body :

```
{
    "5tuple_v4":{           //用户的五元组信息
        "src_net":"",
        "src_port": "-",
        "protocol": "-",
        "dst_port": "-",
        "dst_net": "-"
    },
    "PolicyList":[]         //策略数据
}
```

## 删除数据

- URL

```
http://localhost:port/controller/nb/v2/policites
```

- HTTP Method :

```
DELETE
```

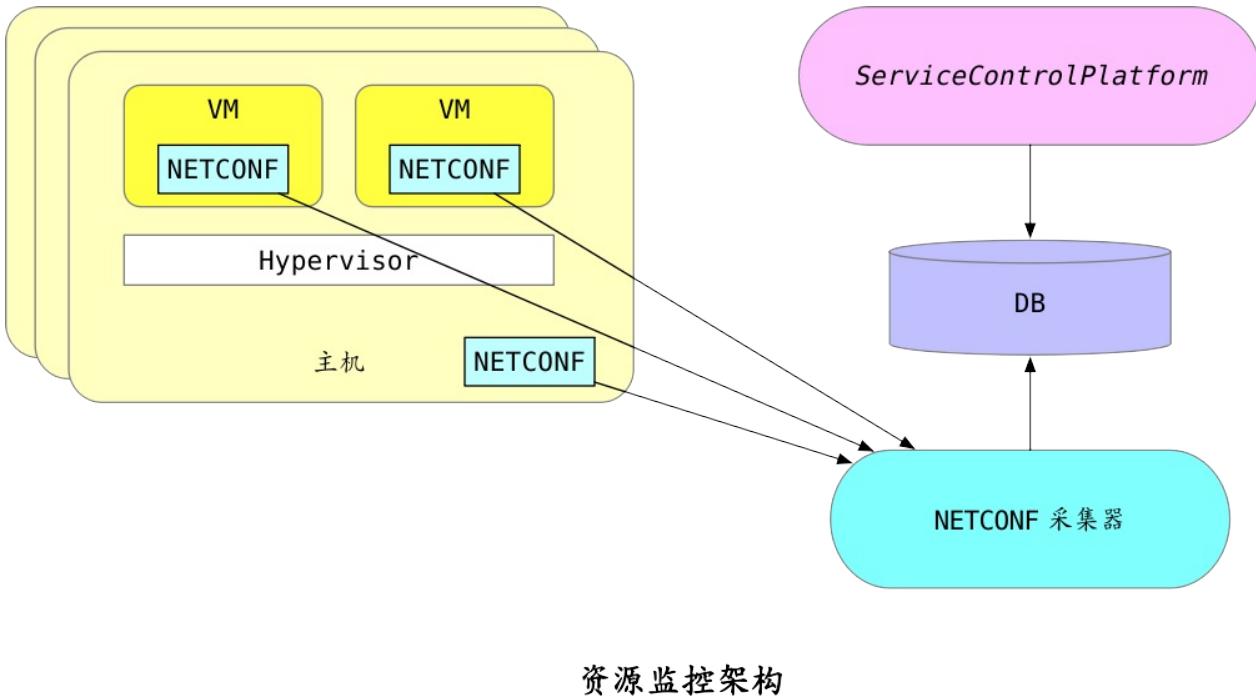
- HTTP Body :

```
{
    "_id": "",      //会话ID
    "5tuple_v4":{   //用户的五元组信息
        "src_net":"",
        "src_port": "-",
        "protocol": "-",
        "dst_port": "-",
        "dst_net": "-"
    }
}
```

## 资源监控

在SCP中，资源监控模块主要管理物理、虚拟服务器或设备的运行状态，为VNF资源的弹性伸缩提供原始数据，包括资源数据统计和资源预警设置。

- 提供7x24h不间断的监控，监控力度为1 min
- 资源数据统计，可以通过图表查询最近60min、最近24h、最近7天、最近30天的监控数据
- 监控项包含内存、磁盘没间隔时间流出的流量

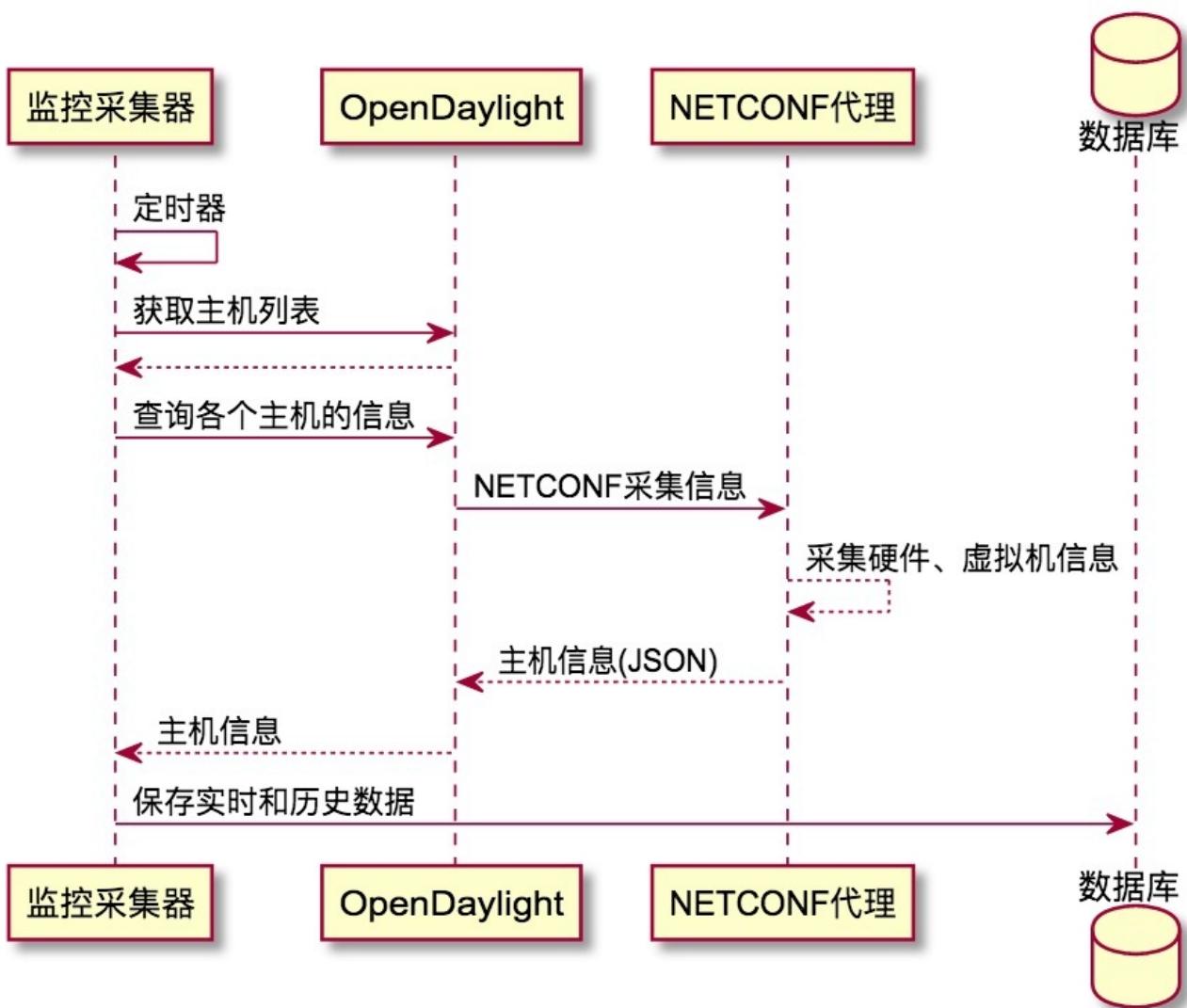


资源监控架构

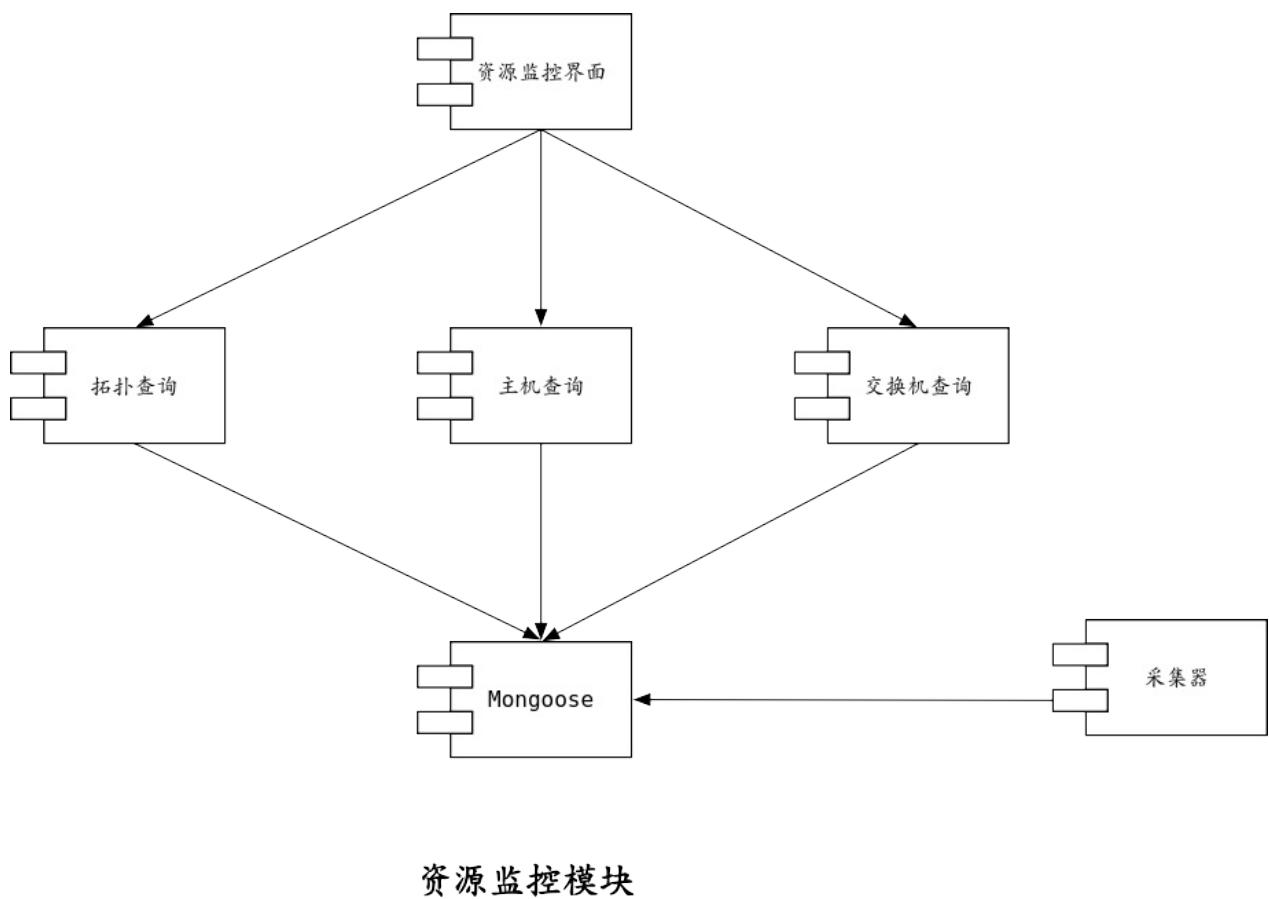
资源监控模块采用 NETCONF 协议采集资源状态，通过给每天主机或虚拟机安装 NETCONF 代理，采集器定时访问代理，从代理读取状态信息，最后保存到数据库中，从而采集资源的使用情况。

## 资源监控采集流程

### SCP资源监控采集流程



资源监控采集模块



# 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>
- [pcnet網路教學](#)

# SDN Handbook Changelog

## Tony Deng (114)

- :bug:fix(1.0): 解决RADIUS的错别字 (2018-06-27 10:51:49) [view commit e625e3a](#)
- :memo: docs(1.0): 添加Radius的RFC链接 (2018-06-13 12:02:33) [view commit b683a8d](#)
- :wrench: chore(1.0): 添加github-buttons plugin (2018-06-13 11:49:23) [view commit 6a056a7](#)
- :bug:fix(1.0): 解决radius图片显示问题 (2018-06-13 11:49:05) [view commit 0a2b530](#)
- :wrench: chore(1.0): 更新token (2018-06-03 14:48:59) [view commit f0ee42d](#)
- :bug:fix(1.0): 更新错误的链接 (2018-06-03 14:26:13) [view commit 6d6e3f7](#)
- :wrench: chore(1.0): 添加travis ci的配置 (2018-06-03 14:23:13) [view commit 6fad88a](#)
- :memo: docs(1.0): 添加Radius认证协议部分 (2018-05-30 15:29:09) [view commit 2dcc549](#)
- :memo: docs(1.0): 添加防火墙演进流程 (2018-04-13 19:17:37) [view commit 9abed68](#)
- :memo: docs(1.0): 完善防火墙的类型 (2018-04-10 14:44:41) [view commit 6d79ec8](#)
- :memo: docs(1.0): 防火墙工作原理之ip address filtering (2018-04-09 17:28:54) [view commit 3741b9f](#)
- :memo: docs(1.0): 添加防火墙的目标 (2018-04-09 17:22:07) [view commit ba1aa85](#)
- :memo: docs(1.0): 添加ConnetOS Firewall配置参考链接 (2018-04-09 01:10:34) [view commit 577e0a7](#)
- :memo: docs(1.0): 添加基本的防火墙分类 (2018-04-09 01:08:48) [view commit da15817](#)
- :memo: docs(1.0): 添加防火墙基本介绍 (2018-04-08 17:48:51) [view commit a77c2ad](#)
- :wrench: chore(1.0): 添加snmp和lldp章节目录 (2018-03-28 12:18:05) [view commit cacd553](#)
- :art: style(1.0): 更新sdn-wan的格式 (2018-03-28 12:16:30) [view commit da91282](#)
- :memo: docs(1.0): 添加基于应用的路由示意图 (2018-02-23 17:24:48) [view commit 0f10955](#)
- :memo: docs(1.0): 添加NETCONF Datasotres交互示意图 (2018-02-23 10:38:49) [view commit bcdd9c8](#)
- :sparkles: feat(1.0): 添加gitbook插件，并更新publish脚本 (2018-01-23 00:31:50) [view commit 08bd04e](#)
- :memo: docs(1.0): 添加spdk架构和nvme direct (2017-12-20 01:03:53) [view commit aaf85ba](#)
- :sparkles: feat(1.0): 添加publish脚本 (2017-12-11 10:30:57) [view commit 3959ba2](#)
- :memo: docs(1.0): 添加rfc6241-zh的参考 (2017-12-09 23:37:15) [view commit cac7840](#)
- :bug:fix(1.0): 修复netconf-tag中的netconf rfc链接描述错误 (2017-12-07 23:45:35) [view commit ed8138a](#)

- :memo: docs(1.0): 添加spkd文档 (2017-12-04 18:59:56) [view commit 83e1ed3](#)
- :memo: docs(1.0): 添加CHANGELOG (2017-12-01 16:23:05) [view commit 4533806](#)
- :art: style(1.0): OVS DPDK文章调整格式以及图片重命名 (2017-12-04 10:25:54) [view commit 339994f](#)
- :memo: docs(1.0): 基本完成ovs dpdk文章，图片还需要调整 (2017-12-04 01:17:31) [view commit ed748f9](#)
- :memo: docs(1.0): 添加DPDK网络虚拟化文章 (2017-12-04 00:52:46) [view commit 42f630e](#)
- :memo: docs(1.0): 添加硬件加速与功能卸载文档 (2017-12-03 23:14:06) [view commit 2872b9e](#)
- :memo: docs(1.0): 添加DPDK网卡多队列 (2017-12-03 22:54:26) [view commit 940f74c](#)
- :memo: docs(1.0): 整理DPDK相关文档 (2017-12-03 00:12:06) [view commit a37adaf](#)
- :memo: docs(1.0): 添加CHANGELOG (2017-12-01 16:23:05) [view commit 6d68365](#)
- :memo: docs(1.0): 添加ovn系列章节 (2017-12-01 16:17:23) [view commit c42f37f](#)
- :memo: docs(1.0): 完善ovs原理，完成ovn介绍 (2017-11-30 23:27:51) [view commit bf628ad](#)
- :memo: docs(1.0): 添加ovs的架构和编译文档 (2017-11-30 19:58:46) [view commit 48a3213](#)
- :memo: docs(1.0): 完善Linux相关的XDP，工具和内核网络参数文章 (2017-11-30 15:56:14) [view commit 296bee2](#)
- :memo: docs(1.0): 完成YANG Module for NETCONF Monitoring RFC文档 (2017-11-28 19:02:21) [view commit 6c92215](#)
- :memo: docs(1.0): 添加安全能力抽象图 (2017-11-28 16:49:31) [view commit 10bbc78](#)
- :memo: docs(1.0): 添加netconf-call-home-over-websocket时序图 (2017-11-28 12:57:28) [view commit 579b41f](#)
- :memo: docs(1.0): 添加juniper的netconf-call-home项目 (2017-11-28 12:56:54) [view commit 84a2d98](#)
- :memo: docs(1.0): 添加netconf call home status图片 (2017-11-28 00:47:33) [view commit 6390d1e](#)
- :memo: docs(1.0): 添加Alibaba NetO arch和SDN中三位一体关系图 (2017-11-27 17:53:57) [view commit a3f8b3a](#)
- :memo: docs(1.0): 添加Full TLS handshake message flow时序图 (2017-11-27 17:48:10) [view commit 3a7088f](#)
- :memo: docs(): (2017-11-27 16:36:21) [view commit 9e0de32](#)
- :memo: docs(1.0): 添加sr-iov、bpf、bbc等文档 (2017-11-25 02:55:29) [view commit 593e3c7](#)
- :memo: docs(1.0): 添加SR-IOV文档 (2017-11-25 02:28:09) [view commit 2f0dfb4](#)
- :memo: docs(1.0): 添加Linux的流量控制文档 (2017-11-25 01:46:50) [view commit d854ec8](#)
- :memo: docs(1.0): 添加负载均衡文档 (2017-11-25 01:39:26) [view commit ce7747f](#)

- :memo: docs(1.0): 添加Linux虚拟网络设备和iptables/netfilter文档 (2017-11-25 01:27:26) [view commit 9f83f12](#)
- :memo: docs(1.0): 添加linux网络配置文档 (2017-11-25 01:15:04) [view commit f445052](#)
- :memo: docs(1.0): 添加vlan文档 (2017-11-25 01:12:23) [view commit 874d993](#)
- :memo: docs(1.0): 完成netconf call home文档 (2017-11-24 23:00:16) [view commit 46c1856](#)
- :memo: docs(1.0): 添加netconf call home文档 (2017-11-24 19:06:48) [view commit 2a72523](#)
- :hammer: refactor(1.0): 调整目录结构以及补充目录需要的README.md (2017-11-24 11:15:13) [view commit aab428f](#)
- :memo: docs(1.0): 添加tcp文档 (2017-11-24 00:15:49) [view commit 3e75d42](#)
- :memo: docs(1.0): 添加dhcp文档 (2017-11-23 23:34:52) [view commit 138e4eb](#)
- :art: style(1.0): 调整显示样式，并补充参考 (2017-11-23 19:07:25) [view commit 2d110a4](#)
- :memo: docs(1.0): 添加交换机文档 (2017-11-23 19:07:06) [view commit 73946cd](#)
- :memo: docs(1.0): 添加UDP文档 (2017-11-23 19:06:43) [view commit 16f90ff](#)
- :memo: docs(1.0): 添加路由文档 (2017-11-23 15:53:16) [view commit e6751bf](#)
- :memo: docs(1.0): 添加icmp文档 (2017-11-23 15:48:44) [view commit 691a338](#)
- :memo: docs(1.0): 添加arp文档 (2017-11-23 15:48:28) [view commit 0ce55d4](#)
- :bug:fix(1.0): 修复ebook的地址错误 (2017-11-23 15:24:41) [view commit f9d6983](#)
- :memo: docs(1.0): 添加netconf各种标签使用场景的介绍及例子 (2017-11-22 20:49:29) [view commit ebbe5f4](#)
- :memo: docs(1.0): 添加netconf各种标签使用场景的介绍及例子 (2017-11-22 20:49:29) [view commit 87ff133](#)
- :memo: docs(1.0): 添加NETCONF Layering Model And Example示例图 (2017-11-22 17:00:44) [view commit da8e5d5](#)
- :memo: docs(1.0): 添加overlay相关文档 (2017-11-22 00:58:01) [view commit 31711e2](#)
- :memo: docs(1.0): 添加tcp/ip模型文档 (2017-11-22 00:40:43) [view commit bfa27b9](#)
- :memo: docs(1.0): 添加netconf实现流程的puml链接 (2017-11-22 00:01:19) [view commit ffd39c6](#)
- :memo: docs(1.0): 添加netconf实现流程图 (2017-11-21 21:53:52) [view commit 24fdc66](#)
- :hammer: refactor(1.0): 调整netconf实现流程uml (2017-11-21 19:03:48) [view commit 191d86b](#)
- :memo: docs(1.0): 添加netconfig的详细描述 (2017-11-21 16:11:47) [view commit 211f7bf](#)
- :hammer: refactor(1.0): 调整sdn中的目录结构 (2017-11-21 11:13:24) [view commit ebbea5c](#)
- :memo: docs(1.0): 调整OpenDaylight相关文章的目录结构，并且更新yang、netconf、odl的参考 (2017-11-21 01:06:23) [view commit 7973c4d](#)
- :memo: docs(1.0): 添加ICG的文档 (2017-11-20 18:56:45) [view commit df19d96](#)
- :memo: docs(1.0): 添加vpn概述中的参考和对sslvpn的描述 (2017-11-20 14:38:01) [view commit 7b1444a](#)

- :memo: docs(1.0): 添加VPN基于协议的分类部分 (2017-11-20 12:04:19) [view commit 0e91dd2](#)
- :memo: docs(1.0): 添加IPSec VPN的说明 (2017-11-17 18:33:26) [view commit 5acaf51](#)
- :memo: docs(1.0): 添加SSL VPN的详细介绍 (2017-11-17 16:25:56) [view commit f3a93b4](#)
- :memo: docs(1.0): 添加安全设备类型的导航以及VPN概述 (2017-11-17 15:53:43) [view commit 7f28c68](#)
- :memo: docs(1.0): 添加SDN控制器应用场景 (2017-11-17 14:51:00) [view commit d9b03b8](#)
- :memo: docs(1.0): 完善SCP的架构和关键业务模型及流程 (2017-11-16 15:16:35) [view commit 66e84aa](#)
- :art: style(1.0): 调整scp架构图的文字样式 (2017-11-16 00:47:31) [view commit 7439b4c](#)
- :memo: docs(1.0): 添加SCP项目类型示例 (2017-11-16 00:39:55) [view commit 3f45519](#)
- :wrench: chore(1.0): graffle调整 (2017-11-15 18:51:38) [view commit 3c95405](#)
- :bug:fix(调整技术成熟度图格式为png): (2017-11-15 10:45:13) [view commit 6cc7abb](#)
- :memo: docs(1.0): 调整所有index文件改成README (2017-11-15 10:41:38) [view commit 113ccfb](#)
- :memo: docs(1.0): 添加Gartner发布的2017年度新兴技术成熟度曲线图 (2017-11-15 10:35:47) [view commit 642581e](#)
- :memo: docs(1.0): 在google示例中添加Andromeda最新2.1的优化文章 (2017-11-14 23:57:22) [view commit 9f1efe9](#)
- :memo: docs(1.0): 添加参考文档和FAQ (2017-11-14 23:41:57) [view commit 632d99d](#)
- :art: style(1.0): 添加onos集群图片说明 (2017-11-14 23:37:10) [view commit 249fe0c](#)
- :memo: docs(1.0): 添加NFV相关介绍 (2017-11-14 23:27:35) [view commit 7146a72](#)
- :memo: docs(1.0): 添加sdwan介绍 (2017-11-14 23:27:13) [view commit f66b8b7](#)
- :memo: docs(1.0): 添加openflow的介绍 (2017-11-14 23:20:08) [view commit 4f2b213](#)
- :memo: docs(1.0): 添加nox/pox介绍 (2017-11-14 23:19:27) [view commit 39e9b38](#)
- :memo: docs(1.0): 添加ryu和floodlight这两个sdn controller介绍 (2017-11-14 23:18:43) [view commit 4c8f05c](#)
- :memo: docs(1.0): 添加ONOS文章 (2017-11-14 23:08:06) [view commit 42acb87](#)
- :memo: docs(1.0): 添加数据平面文章 (2017-11-14 23:03:01) [view commit 248efab](#)
- :memo: docs(1.0): 重新整理google b4案例的说明和架构图 (2017-11-14 16:48:01) [view commit 79fed9c](#)
- :memo: docs(1.0): add google b4 arch image (2017-11-14 00:21:38) [view commit 466153a](#)
- :memo: docs(1.0): 添加sdn南向协议的一些说明，并且添加SDN实践的一些文档 (2017-11-13 23:27:42) [view commit 116ad5e](#)
- :memo: docs(1.0): 添加YANG Language目录 (2017-11-13 18:51:42) [view commit f24d39b](#)
- :hammer: refactor(1.0): 调整.opendaylight子项目的文件名，并添加各个项目的项目地址

和描述补充 (2017-11-13 16:08:44) [view commit db6da1c](#)

- :memo: docs(1.0): 添加OpenDaylight项目相关性示意图 (2017-11-13 01:48:51) [view commit 10cbec5](#)
- :memo: docs(1.0): 添加OpenDaylight各个子项目简单介绍和描述 (2017-11-13 01:08:05) [view commit 2d3103d](#)
- :memo: docs(1.0): 添加网络基础目录 (2017-11-12 23:25:12) [view commit b5eb24a](#)
- :memo: docs(1.0): 添加目录结构 (2017-11-12 23:24:39) [view commit 5c1ed7a](#)
- :memo: docs(1.0): 添加OpenDaylight架构图 (2017-11-12 23:22:54) [view commit 46dabc6](#)
- :sparkles: feat(1.0): 添加画图源文件 (2017-11-12 23:22:03) [view commit 8e07877](#)
- :wrench: chore(1.0): 添加macos中自动生成文件忽略配置 (2017-11-12 23:21:03) [view commit 12bc97a](#)
- :bug:fix(1.0): 修改gitbook地址错误 (2017-11-12 23:20:34) [view commit db9a1bd](#)
- :memo: docs(1.0): 添加前言 (2017-11-09 18:14:20) [view commit 0eb0c71](#)
- Initial commit (2017-11-09 04:07:07) [view commit 216c87c](#)