# vpp开发文档

## vpp总览

### ****FD.io项目介绍****

FD.io(Fast data - Input/Output)是Linux基金会旗下的又一开源项目，其成立于2016年2月11日。该项目在通用硬件平台上提供了具有灵活性、可扩展、组件化等特点的高性能IO服务框架，用以迎接下一个网络和存储浪潮。该框架支持高吞吐量、低延迟、高资源利用率的user space IO服务，并可适用于多种硬件架构(x86, ARM, and PowerPC)和部署环境(bare metal, VM, container)。

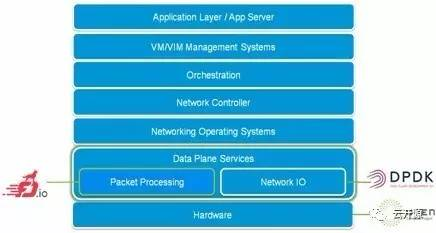
[](https://img2018.cnblogs.com/i-beta/1414775/202002/1414775-20200223163847948-605115254.png)

图 开放网络生态系统概要

FD.io开源项目的关键组件是由Cisco捐赠的已商用的VPP(Vector Packet Processing )库，VPP高度模块化，用户可以根据需求实现定制化的服务节点插件。

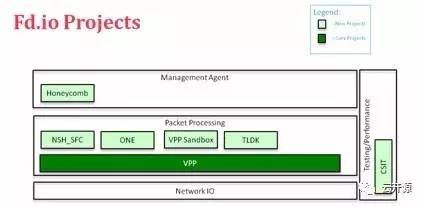
[](https://img2018.cnblogs.com/i-beta/1414775/202002/1414775-20200223163912572-2050879158.png)

图 FD.io项目架构

VPP和FD.io其他子项目如NSH\_ SFC, Honeycomb, and ONE一起加速NFV的data plane。未来的代码贡献将来自开源社区，FD.io的成员期望在各个领域对FD.io进行拓展，例如防火墙、负载均衡、LISP、主机追踪、IDS、硬件加速器集成等。

### VPP简介

VPP全称Vector Packet Processing，据说是Cisco 2002年开发的商用代码。2016年2月11号，Linux基金会创建FD.io项目。Cisco将VPP代码的开源版本加入该项目，目前已成为该项目的核心。VPP商用版看不到我们暂且不说，单看这个开源版本，个人觉得还是一个初级版本，因为自从关注之后代码更新还是很多，包括bug修复和功能开发。VPP运行于用户空间，支持多种收包方式，我个人关注的是DPDK。

VPP有两个关键特性：

* 框架可扩展。(确实易扩展)
* 成熟的交换/路由功能。(不是我说的，官方说的)

### VPP技术原理

（https://www.sohu.com/a/134745174\_468741）

所谓VPP向量报文处理是与传统的标量报文处理相对而言的。传统报文处理方式，同时也是人类常用的逻辑思维方式，即：报文是按照到达先后顺序来处理，第一个报文处理完，处理第二个，依次类推；A callsB calls C….return return return，函数会频繁嵌套调用，并最终返回。

由此可见，传统标量报文处理有如下缺陷：

**1、**I-cache 抖动(cache时间局限性和空间局限性特点)

**2、**I-cache misses

**3、**除了扩大cache外，没有变更方案。

相比较而言，向量报文处理则是一次处理多个报文，也相当于一次处理一个报文数组packet[n]如下图：

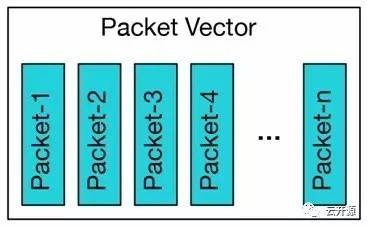
[](https://img2018.cnblogs.com/i-beta/1414775/202002/1414775-20200223164126256-1347873102.png)

图 向量报文

VPP把一批底层硬件队列Rx ring收到的包，组成一个Packet Vector或者是一组包，借助于报文处理图Packet Processing graph来实现处理流程，图节点graph node把整个过程分解为一个个先后连接的服务node。这一组包（packet vector）被第一个graph node节点的任务处理，然后依次被第二个graph node节点的任务处理，依次类推，如下图所示：

[](https://img2018.cnblogs.com/i-beta/1414775/202002/1414775-20200223164704076-1469175591.png)

图 报文处理图

由于向量报文中的第一个报文packet-1为I-cache进行了热身，向量报文中剩下报文的处理性能可以直接达到极限，I-cache 缺失的固定开销平摊到了整个向量处理中，使单个报文的处理开销显著降低。

由此可见，向量报文处理解决了标量处理的主要性能缺陷，并有具有如下优点：

**1、**解决了I-cache抖动问题

**2、**向量报文进行预取缓解了读时延问题，高性能并且更加稳定

### 扩展性

#### 节点操作

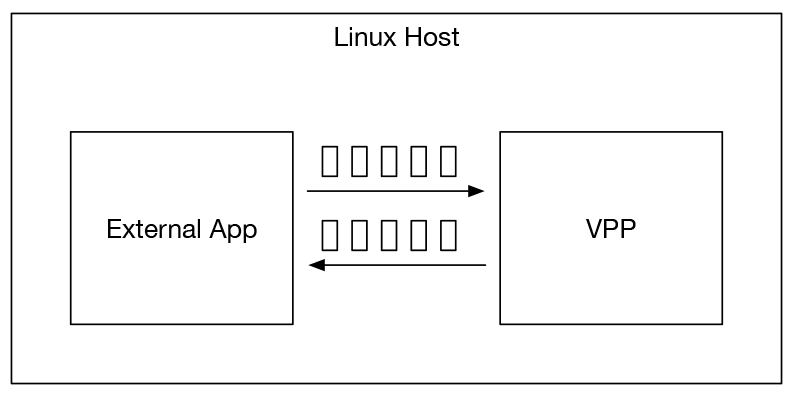
VPP平台是通过graph node串联起来形成一条datapath来处理报文，类似于freebsd的netgraph。通过插件的形式引入新的graph node或者重新排列报文的graph node。将插件添加到插件目录中，运行程序的时候就会自动加载插件。另外插件也可以根据硬件情况通过某个node直接连接硬件进行加速。VPP平台可以用于构建任何类型的报文处理应用。比如负载均衡、防火墙、IDS、主机栈。也可以是一个组合，比如给负载均衡添加一个vSwitch。

通过创建插件，可以任意扩展如下功能：

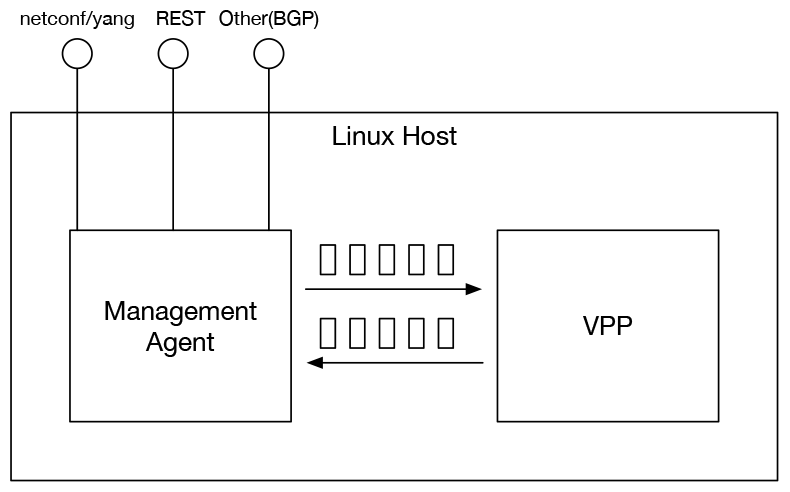
* 自定义新的graph node。
* 重新排列graph node。
* 添加底层API。

### 可编程能力

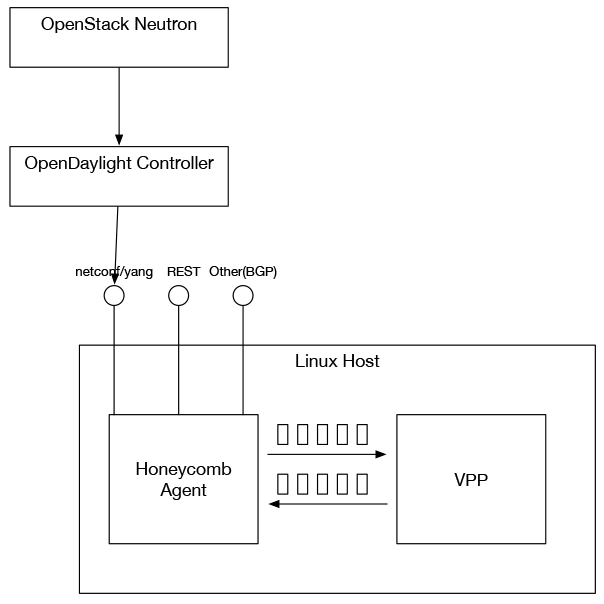
VPP还提供了基于共享内存或者消息队列的高性能内部API。目前VPP平台支持C和JAVA客户端进行内部API绑定。  
如下图所以，我们可以实现一个外部应用来对VPP进行操作：

[](https://img2018.cnblogs.com/i-beta/1414775/202002/1414775-20200223162924306-1631039900.png)

 以上的编程能力是针对内部API的调用，另外还可以支持远程可编程能力。远程可编程能力可以通过Data Plane Management Agent来实现。这里所说的Data Plane Management Agent其实就是类似上面的外部应用，只不过他不参与直接控制，只是一个傀儡，用来传话。通过外部API与Data Plane Management Agent进行通信。Data Plane Management Agent通过内部API与VPP应用(引擎)进行通信。这是一个非常灵活的方法，任何人可以添加，但是需要外部API和Data Plane Management Agent匹配，以实现特定需求的VPP应用。  
下图展示Agent对接VPP和外部程序：

[](https://img2018.cnblogs.com/i-beta/1414775/202002/1414775-20200223163038584-2062684261.png)

案例Honeycomb Agent：  
Honeycomb Agent通过netconf和restconf发布了yang模型的VPP功能。像OpenDaylight这样支持netconf/yang的控制器可以挂载Honeycomb Management Agent来工作。而OpenDaylight支持OpenStack Neutron，所以如下图所示OpenStack Neutron可以集成到vpp来：

[](https://img2018.cnblogs.com/i-beta/1414775/202002/1414775-20200223163055158-1355967417.png)

### 网络功能

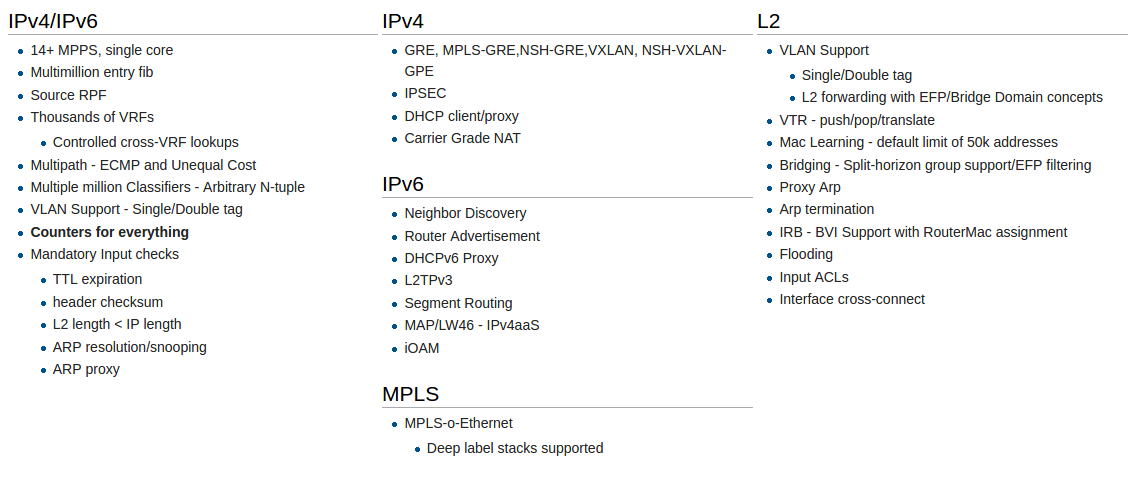
（https://my-vpp-docs.readthedocs.io/en/latest/overview/index.html）

#### 网络特性

VPP拥有的网络特性如下：

* 快速查找路由表、CAM表。
* 任意n元组分类。
* 商用级别的交换/路由功能。

VPP能提供的所有功能如下：

[](https://img2018.cnblogs.com/i-beta/1414775/202002/1414775-20200223163119034-250076138.png)

#### 网络性能

[FD.io](http://fd.io/) VPP的一个优点是在相对低功耗的计算上具有高性能，这种性能基于以下特性：

* 专为商用硬件设计的高性能用户空间网络堆栈：
* L2，L3和L4功能和封装。
* 优化的数据包接口，支持多种用例：
  + 用于高速VM到VM连接的集成vhost用户后端
  + 用于高速Container-to-Container连接的集成memif容器后端
  + 基于vhost的集成vhost接口数据包到Linux内核
* 在主机上运行，​​在VM和Linux容器内运行，都是相同的代码路径
* 利用最佳的开源驱动程序技术：DPDK
* 大规模测试;线性核心扩展，使用数百万个流和mac地址进行测试

这些功能旨在充分利用常见的微处理器优化技术，例如：

* 通过处理向量中的数据包减少缓存和TLS未命中
* 使用向量指令实现IPC增益，例如：SSE，AVX和NEON
* 消除模式切换，上下文切换和阻塞，始终执行有用的工作
* 缓存内置的缓冲区用于缓存和记忆效率

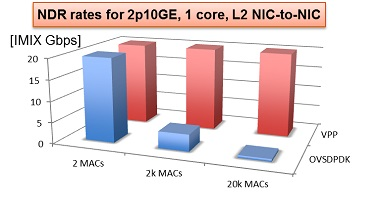
OVS更适用于SDN，因为他的流表更加灵活，实现的功能更加强大，尤其是在IaaS的主机中做网络转发比较常见。

而VPP更适用于NFV，适合做特定功能的网络模块，而且性能占优。

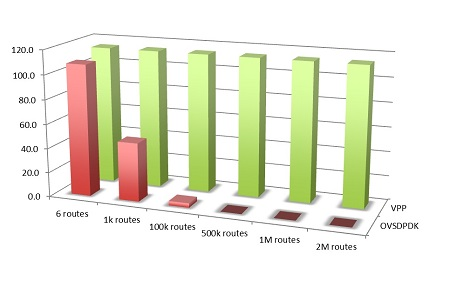
多核基准性能例子 (UCS-C240 M3, 3.5gHz,所有内存通道转发ipv4):

* 1 core: 9 MPPS in+out。
* 2 cores: 13.4 MPPS in+out。
* 4 cores: 20.0 MPPS in+out。

下面几个图展示的是和OVS+DPDK的性能对比：

[](https://img2018.cnblogs.com/i-beta/1414775/202002/1414775-20200223163143904-1142912328.png)

 下图是在Haswell x86 架构的E5-2698v3 2x16C 2.3GHz上测试，图中显示了12口10GE，16核，ipv4转发：

[](https://img2018.cnblogs.com/i-beta/1414775/202002/1414775-20200223163208256-422412609.png)

### ****源码架构****

vpp数据平面分为四个不同的层：

* 基础架构层：包括vppinfra，vlib，svm和二进制api库。源码：/src/{vppinfra, vlib, svm, vlibapi, vlibmemory}
* 通用网络协议栈层：vnet。源码：/src/vnet
* 应用程序shell：vpp。源码：/src/vpp

日益丰富的数据平面插件。源码：/src/plugins

参考：

　　https://my-vpp-docs.readthedocs.io/en/latest/overview/index.html

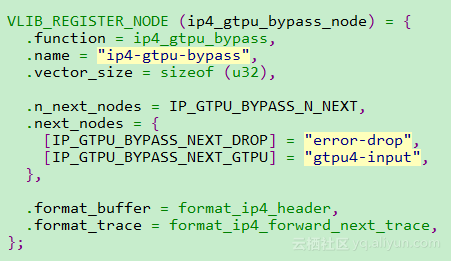
　　https://www.sohu.com/a/134745174\_468741

## vpp节点操作

### vpp添加节点

向VPP中注册新结点的方式比较简单，使用VPP中定义好的VLIB\_REGISTER\_NODE宏来声明我们的新结点，新结点中的处理函数需要自己去实现

VLIB\_REGISTER\_NODE(node\_name\_node, static)

VLIB\_REGISTER\_NODE本质上是新建一个vlib\_node\_registration\_t类型，VPP会在初始化的时候往node表中注册该结点信息，这里该变量名为node\_name\_node的数据结构并定义为static属性  
所以如果想在不同.c文件中引用该变量，需要去掉static关键字，并在.h文件中加上node\_name\_node的声明  


结点中定义的next\_nodes定义了通过该结点后数据应到底何结点，这里可以根据结点的用途自由选取需要的结点名

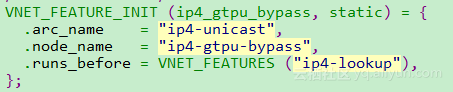
#### 使用feature机制

早期的VPP本身的node框架比较固定，各个node之间逻辑连接已经固化，为此新版本增加了feature机制  
这里feature机制本质上来说还是结点，只不过该结点可以在运行的时候通过命令进行配置是否打开或关闭，从而影响数据流的走向

**选择合适的arc类**

对新加入的结点进行管理，新的feature(即我们新建的结点)必须属于某个arc类，并作用于某个interface实体。  
通过set interface feature <intfc> <feature\_name> arc <arc\_name> [disable]命令来开启或关闭该feature功能。  
通常arc类的名字对应为其起点结点的名字，使用命令开启关闭feature功能能动态的改变数据的流向。  
如果选择按照feature机制来加入结点的话需要注意以下几点：  
VPP提供的arc类比较多，我们需要自己选择合适的arc来插入我们的结点:  
1.nsh-output:  
2.mpls-output:  
3.mpls-input:  
4.ip6-drop:  
5.ip6-punt:  
6.ip6-local:  
7.ip6-output:  
8.ip6-multicast:  
9.ip6-unicast:  
10.ip4-drop:  
11.ip4-punt:  
12.ip4-local:  
13.ip4-output:  
14.ip4-multicast:  
15.ip4-unicast:  
16.ethernet-output:  
17.interface-output:  
18.device-input:

**在arc类上登记feature结点**



arc\_name：为我们选定的feature结点要插入的地方  
node\_name：为我们自己新注册的结点名  
runs\_before：说明新结点必须比某个feature结点先执行，通过新结点后的流可能流入下个feature结点也可能到达其他路径

**关联feature结点及其通过函数**

VLIB\_NODE\_FUNCTION\_MULTIARCH(ip4\_gtpu\_bypass\_node, ip4\_gtpu\_bypass)

#### 不使用feature机制

不使用feature机制的话，结点间的关系相对来说更加静态，只能在编译的时候确定结点间的关系，不能在运行的时候进行改变，可以插入节点的地方只能由系统提供的几个接口  
向这些入口登记函数后，后续的数据流将传到你定义的结点  
可能还有其他的一些插入结点的函数，这里只列出常用到的几个函数：

**L1**

vnet\_hw\_interface\_rx\_redirect\_to\_node (vnet\_main\_t \*vnm, u32 hw\_if\_index, u32 node\_index)  
将某个hw interface的rx数据重定向到某个结点，node\_index为结点的index索引

**L2、L3**

ethernet\_register\_input\_type (vlib\_main\_t \*vm, ethernet\_type\_t type, u32 node\_index)  
将在"ethernet-input"结点后插入特定type的结点，  
这里type包括ethernet\_type(0x806, ARP)、ethernet\_type (0x8100, VLAN)、ethernet\_type (0x800, IP4)等二、三层协议  
具体支持的相关协议见src/vnet/ethernet/types.def文件。

**L4**

ip4\_register\_protocol (u32 protocol, u32 node\_index)  
将在"ip4-local"结点后插入特定protocol的结点，这里protocol包括ip\_protocol (6, TCP)、ip\_protocol (17, UDP)等四层协议。  
具体支持的相关协议见src/vnet/ip/protocols.def文件。

**L5**

udp\_register\_dst\_port (vlib\_main\_t \* vm, udp\_dst\_port\_t dst\_port, u32 node\_index, u8 is\_ip4)  
将在"ip4-udp-lookup"结点后插入特定dst\_port的结点，这里dst\_port包括ip\_port (WWW, 80)等五层应用端口。  
具体支持的相关端口见src/vnet/ip/ports.def文件。