# 概述

DPDK（Data Plane Development Kit）是一个开源的软件开发工具集，旨在加速数据平面应用的开发和执行。它提供了一套高性能的数据平面库和驱动，使开发人员能够构建高性能的网络功能，如包处理、数据包过滤、流量管理等。

DPDK 的设计目标是通过绕过操作系统内核的数据路径，直接与硬件交互，从而提高网络应用的性能和效率。它主要关注数据包的快速处理，通过使用零拷贝技术、轮询模式、用户空间驱动等方法来降低处理延迟和提高吞吐量。

DPDK 提供了一系列的库和工具，包括：

* EAL（Environment Abstraction Layer）：提供了与硬件交互的抽象接口，包括内存管理、线程管理、设备管理等。
* 网络库：包括网络设备的驱动和管理，支持各种以太网卡、虚拟设备和加速卡。
* 数据包库：提供了高效的数据包处理和转发功能，包括数据包接收、解析、修改、封装和转发等。
* 流量管理库：用于管理和控制网络流量，包括流量分类、负载均衡、队列调度等。
* 统计和诊断工具：用于监测和分析数据平面应用的性能和行为。

使用 DPDK 进行开发需要了解其基本概念和使用方法。首先，需要安装 DPDK，并配置环境变量以支持应用程序的编译和执行。然后，可以使用 DPDK 提供的库和工具来构建自定义的数据平面应用。

在编写应用程序时，需要注意以下几点：

* 内存管理：DPDK 提供了自定义的内存分配和管理机制，可以通过预分配大页内存来提高性能。
* 线程模型：DPDK 使用多线程模型来充分利用多核处理器的性能，需要合理地进行线程管理和任务分配。
* 设备配置：DPDK 支持各种网络设备，需要配置和初始化设备，设置合适的参数和属性。
* 数据包处理：使用 DPDK 的数据包库进行数据包的接收、处理和转发，可以利用多核处理器的并行性来加速处理过程。
* 错误处理和调试：DPDK 提供了丰富的错误处理和调试工具，可以帮助定位和解决问题。
* 通过合理使用 DPDK，可以构建高性能的数据平面应用程序，实现快速的数据包处理和转发。

优势

* 高性能：DPDK 通过绕过操作系统内核，直接与硬件交互，避免了操作系统调度和处理的开销，提供了更低的延迟和更高的吞吐量。
* 零拷贝技术：DPDK 使用零拷贝技术，在数据包的接收和发送过程中避免了数据的复制，减少了内存访问和数据拷贝的开销。
* 用户空间驱动：DPDK 提供了用户空间驱动程序，使得网络设备的驱动程序在用户空间运行，减少了用户态和内核态之间的上下文切换。
* 多核优化：DPDK 充分利用多核处理器的并行性，通过多线程和任务分配，实现了更好的负载均衡和处理性能。

注意事项：

* 硬件兼容性：DPDK 支持多种硬件设备，但需要确保所选硬件设备与 DPDK 兼容，并安装相应的驱动程序。
* 环境配置：在使用 DPDK 之前，需要正确配置环境变量，包括设置正确的 PATH、LD\_LIBRARY\_PATH 和 RTE\_SDK 等变量。
* 内存管理：DPDK 使用大页内存来提高性能，需要在系统中配置和启用大页内存支持。
* 线程管理：合理管理和分配线程，避免线程竞争和资源冲突，提高多核处理器的利用率。
* 错误处理和调试：DPDK 提供了丰富的错误处理和调试工具，如日志记录、统计信息和调试命令等，可用于定位和解决问题。

总结：

DPDK 是一个强大的开发工具集，可以加速数据平面应用的开发和执行。它提供了高性能的数据平面库和驱动，使开发人员能够构建高性能的网络功能。使用 DPDK 需要了解其基本概念和使用方法，并注意硬件兼容性、环境配置、内存管理、线程管理以及错误处理和调试等方面的注意事项。

通过合理使用 DPDK，可以充分发挥硬件的性能潜力，构建高效的数据平面应用，满足网络处理的需求。

官网手册

http://doc.dpdk.org/guides/linux\_gsg/sys\_reqs.html#bios-setting-prerequisite-on-x86

传统的收发数据包方式，首先网卡通过中断方式通知内核协议栈对数据包进行处理，内核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为本机的 Socket，满足条件则会将数据包拷贝一份向上递交到用户态 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。

内核态（Kernel Mode）：运行操作系统程序，操作硬件

用户态（User Mode）：运行用户程序

运行在用户态下的程序不能直接访问操作系统内核数据结构和程序。当我们在系统中执行一个程序时，大部分时间是运行在用户态下的，在其需要操作系统帮助完成某些它没有权力和能力完成的工作时就会切换到内核态（比如系统调用、异常、硬件中断）

为了使得网卡驱动（e.g. PMD Driver）运行在用户态，实现内核旁路。Linux 提供了 UIO（User Space I/O）机制。使用 UIO 可以通过 read() 感知中断，通过 mmap() 实现和网卡设备的通讯。

https://blog.csdn.net/weixin\_39094034/article/details/123829415

DPDK是INTEL公司开发的一款高性能的网络驱动组件，旨在为数据面应用程序提供一个简单方便的，完整的，快速的数据包处理解决方案，主要技术有用户态、轮询取代中断、零拷贝、网卡RSS、访存DirectIO等。

高性能网关：可以使数据通过dpdk而不经过内核的协议栈直达应用层，用以提升网关性能。dns可以这么做。

虚拟交换机：利用dpdk优秀的性能，跑一个虚拟的交换机或是路由器，以提升性能。

防火墙：对数据的出入口做一些限制。

依赖环境

GCC，版本

配置内存大页和系统启动设置

Dpdk版本查看

/usr/local/lib/x86\_64-linux-gnu/pkgconfig

pkg-config --modversion libdpdk-libs

<https://blog.csdn.net/wangquan1992/article/details/120822519>

## 安装部署

## RedHawk7.3当前环境状态

* Linux内核版本：4.4.60
* GNU libc：2.17
* GCC版本：4.8.5
* pkg-config版本：0.27.1
* python版本：2.7.5

## 适配网卡

* Network Adapter X710-DA4
* 其他待确认

## DPDK版本选择

DPDK 19.11.14 (LTS)，19年LTS版，为长期支持版本，经过这几年使用与迭代，应该较为稳定；且网上唯一DPDK公开课基于19版，可间接获取支持。且20.11及以后版本只能使用meson编译，不支持make编译，需部署python3.5+，引入额外部署成本。

## 环境需求

### BIOS设置

对大多数平台，使用基本的DPDK功能无需对BIOS进行特殊设置。

（1）但对于HPET（High Precision Event Timer，高精度定时器，到时了会产生中断）定时器和电源管理功能，以及为了获得小包处理的高性能，则可能需要更改 BIOS 设置。

查看是否开启了HPET：

grep hpet /proc/timer\_list

（2）如果使能 UEFI 安全启动方式，则 Linux 内核会禁止 uio 模块加载，进而无法挂载 igb\_uio.ko 模块。在这种情况下可以使用 vfio 模块代替 igb\_uio/uio\_pci\_generic 模块。

### 依赖环境

（1）编译环境：

* C编译器：GCC 4.9及以上，或者clang3.4及以上；
* Python 3.5及以上，使用meson and ninja编译DPDK所需，帮助脚本所需；
* Meson (version 0.47.1+) and ninja，20.11及以后版本需要使用
* NUMA库numactl-devel；

（2）运行环境

* 内核版本：3.16及以上；
* GNU libc：2.7及以上；

Export指令，临时修改环境变量

* yum groupinstall "Development Tools" -y //For RHEL/Fedora系统
* Python 3.5+ //用来安装 meson and ninja （20.11及以后版本需要使用）
* Python 2.7+/3.2+ //用来执行 DPDK 源码中包含脚本
* Meson (version 0.47.1+) and ninja //大部分Linux系统中自带安装，python安装命令：pip3 install meson ninja
* yum install kernel-devel-$(uname -r) //安装内核开发包

DPDK的主要优势是提高吞吐量、降低延时

Gcc不一定向下兼容，新版gcc为适用新的语言标准可能将不识别老的程序

## DPDK工具安装

### 环境编译

进入DPDK主目录执行：

export RTE\_SDK=$PWD

export RTE\_TARGET=x86\_64-native-linuxapp-gcc

make config T=$RTE\_TARGET O=$RTE\_TARGET

cd $RTE\_TARGET

make -j 4

make install

注：使用./usertools/dpdk-setup.py编译会产生T定义问题



/opt/f-stack-1.21/dpdk/x86\_64-native-linuxapp-gcc

export FF\_PATH=`pwd`

export FF\_DPDK=$RTE\_SDK/x86\_64-native-linuxapp-gcc

端口绑定驱动（内核态，用户态），已绑定端口数与核心数对应？

### 挂载UIO模块

modprobe uio

insmod opt/dpdk-stable-19.11.14/x86\_64-native-linuxapp-gcc/kmod/igb\_uio.ko

### 绑定网卡设备到 IGB\_UIO

ifconfig enp1s0f0 down //关闭即将绑定的以太网口

./usertools/dpdk-devbind.py -s //查看网络设备状态，获取pcieID

./usertools/dpdk-devbind.py -b igb\_uio 0000:01:00.0 0000:01:00.1

Demo搭建入门

https://blog.csdn.net/qq\_51011672/article/details/130087497?ops\_request\_misc=&request\_id=&biz\_id=102&utm\_term=dpdk%E4%BD%BF%E7%94%A8&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~sobaiduweb~default-4-130087497.142^v88^control\_2,239^v2^insert\_chatgpt&spm=1018.2226.3001.4187

UIO安装

大页内存

meson -v查看Meson 版本

clang -v

Ip down

Lspci获取网卡pci地址

网口绑定，通过pci端口

运行时环境rte

配置rx。Tx队列

Buffer深度。Mbuffer

加组，用内核协议

组播加组协议

内核协议栈,用户协议栈

直接ringbuffer取数

环形队列深度？

测试工具，testpmd

Skbuff（socket buffer）与mbuff的区别

Skbuff，网络一帧数据的组织方式，内核相关，多个buf之间双向链表，红黑树组织

Mbuff，内存池中数据的基本单位

队列放进内存池

运行时参数，内存池参数

1908版本

绑定提供了两种模式，

IGB Uio 接管网卡数据

Vfio

Kni：过滤后给内核

一般绑定igb uio

先down后绑定，ifconfig看不到了，无mac，ip

网卡本身，多中断支持的，

Cat /proc/interrupts | grep eth0

E1000模拟网卡

Vmxnet3支持多队列网卡，e1000不支持，类型选vmxnet3

接收数据两种方式，轮询与事件驱动

Poll 大块数据，实时性要求不高

Event 小块数据，实时性要求高

Dpdk为网口与进程中间，是否截获？

Dpdk自己封装了线程

Dpdk与netmap

### 设置大页内存

2M大页内存设置

echo 2048 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr\_hugepages

mkdir -p /mnt/huge

mount -t hugetlbfs nodev /mnt/huge

查看大页内存设置是否生效

grep HugePages\_ /proc/meminfo

HugePages\_Total: 1024

HugePages\_Free: 1024

HugePages\_Rsvd: 0

HugePages\_Surp: 0

### 开机测试流程

1.设置环境变量

DPDK主目录下：

export RTE\_SDK=$PWD

export RTE\_TARGET=x86\_64-native-linuxapp-gcc

DMA控制器：支持设置数据缓冲区起止地址，控制IO写入数据至内存。

通过testpmd获取

Configuring Port 0 (socket 0)

Port 0: 9C:C2:C4:30:70:39

Configuring Port 1 (socket 0)

Port 1: 9C:C2:C4:30:70:3A

 应用层通过while死循环的方式，调用rte\_eth\_rx\_burst接口轮询接收来自网卡的报文，相当于绕过了内核协议栈，将内核旁路了

这里解释下dd标记，当dma控制器将接收到的报文保存到描述符指向的mbuf空间时，由dma控制器打上dd标记，表示dma控制器已经把报文放到mbuf中了。应用层在获取完报文后，需要清除dd标记。

pmd通过轮询的方式直接从dma控制器获取报文，而无需经过内核

e1000网卡驱动eth\_igb\_dev\_init初始化函数中，会将网卡硬件结构的hw\_addr指针直接指向网卡的配置空间。应用层后面就可以对这个配置空间进行读写操作。应用层之所以可以对这个配置空间中的寄存器进行操作，得益于igb\_uio驱动，igb\_uio驱动将网卡的配置空间暴露到应用层来，应用层才能知道网卡的寄存器地址在哪里，进而对这些寄存器进行读写操作。

消息队列，链式结构，节点动态增加，不需要初始化节点数量

与ringbuffer，环形缓冲，有长度，超长取模，从头开始

Kni，内核网络接口，dpdk提供的应用层写数到内核的方式，非系统调用的方式

Net——device，eth0在内核里会生成net-device

IO Conrtrol /dev/kin操作方式

Kni—ioctl

以非系统调用的形式访问内核协议栈，降低系统调用开销

不将数据包拷贝进内核协议栈，降低数据拷贝开销

# 常用API

rte\_eth\_promiscuous\_enable()

端口对应的是以太网设备，对一个以太网设备进行收发包前的初始化时，需要经过以下几步：

● rte\_eth\_dev\_configure（）：基于应用所需的收发数据包队列数及一些特定的配置信息配置设备。

● rte\_eth\_tx\_queue\_setup（）：创建发送队列，对于硬件设备，驱动需要为其分配DMA ring用于发送

数据包。

● rte\_eth\_rx\_queue\_setup（）：创建接收队列，对于硬件设备，驱动需要为其分配DMA ring用于接收数据包。

● rte\_eth\_dev\_start（）：启动该设备，在启动后，该设备就可以用于收发数据包了

## 网卡模式设置

rte\_eth\_promiscuous\_disable(uint16\_t port\_id)

返回值：

* 0：成功

rte\_eth\_promiscuous\_enable(uint16\_t port\_id)

返回值：

* 0：成功

rte\_eth\_promiscuous\_get(uint16\_t port\_id)

返回值：

* 1：混杂模式激活
* 0：混杂模式未激活
* -1：错误

"(0,lcore\_rx,lcore\_tx[,lcore\_kthread,...])[,(port,lcore\_rx,lcore\_tx[,lcore\_kthread,...])]"

## rte\_eth\_dev\_configure

保存应用层传进来的网卡配置信息

int rte\_eth\_dev\_configure (uint16\_t port\_id,uint16\_t nb\_rx\_queue,uint16\_t nb\_tx\_queue,const struct rte\_eth\_conf \* eth\_conf )

* port\_id：端口标识符
* nb\_tx\_queue：端口发送队列数量
* nb\_rx\_queue：端口接收队列数量
* eth\_conf：网卡配置信息，详见api

宏定义（依赖文件已定义）

#define RTE\_ETHER\_MAX\_LEN 1518

#define RTE\_MBUF\_DEFAULT\_DATAROOM 2048

#define RTE\_ETHER\_TYPE\_IPV4 0x0800

RTE\_MBUF\_DEFAULT\_BUF\_SIZE

IPPROTO\_UDP <netinet/in.h中

结构体

struct rte\_mempool

struct rte\_eth\_conf

struct rte\_ether\_addr

struct rte\_ether\_hdr

struct rte\_ether\_hdr \*

struct rte\_ipv4\_hdr

struct rte\_udp\_hdr

struct in\_addr

函数

init\_port

## rte\_eth\_dev\_count\_avail

uint16\_t rte\_eth\_dev\_count\_avail(void) 获取程序可用端口数量

## rte\_eth\_rx\_queue\_setup

为网卡开辟接收队列空间

int rte\_eth\_rx\_queue\_setup(uint16\_t port\_id,uint16\_t rx\_queue\_id,uint16\_t nb\_rx\_desc,unsigned int socket\_id,const struct rte\_eth\_rxconf \* rx\_conf,struct rte\_mempool \* mb\_pool)

* port\_id：端口标识符
* rx\_queue\_id：接收队列标识符，0起始
* nb\_rx\_desc：分配给环形缓存的接收描述符数量
* socket\_id：NUMA系统下socket\_id套接字标识符。若非NUMA系统，则该值可以为SOCKET\_ID\_ANY
* rx\_conf：NULL，使用默认的RX配置，
* mb\_pool：内存池指针，其中分配mbuf

返回值

* 0:成功，接收队列正确设置。
* -EIO:如果设备被移除。
* -EINVAL:内存池指针为空，或者可以从该内存池分配的网络缓冲区的大小不适合设备控制器允许的各种缓冲区大小。
* -ENOMEM:初始化接收描述符时，无法分配接收环描述符或从内存池中分配网络内存缓冲区。

## rte\_eth\_dev\_socket\_id

返回端口连接的套接字ID

int rte\_eth\_dev\_socket\_id(uint16\_t port\_id)

* port\_id：端口ID

返回值：

* 0：端口ID超出范围，或该端口未连接
* 1：端口已连接

## rte\_eth\_dev\_start

端口配置完成后启动，有1标志，调用PMD端口启动回调函数后不允许改MAC

int rte\_eth\_dev\_start (uint16\_t port\_id )

* port\_id：端口ID

返回值：

* 0：成功启动
* <0:ErrorCode

## rte\_pktmbuf\_pool\_create

创建并初始化内存池，封装了rte\_mempool结构体

struct rte\_mempool\* rte\_pktmbuf\_pool\_create ( const char \*  name,

unsigned n,

unsigned cache\_size,

uint16\_t priv\_size,

uint16\_t data\_room\_size,

int socket\_id

)

* Name：内存池名称，”name”
* n,mbuf数量，n = (2^q - 1)；
* cache\_size,单核缓存
* priv\_size,数据buf与mbuf之间的用户私有，与RTE\_MBUF\_PRIV\_ALIGN（8，mbuf私有区域的对齐约束）一致
* data\_room\_size, mbuf大小，RTE\_PKTMBUF\_HEADROOM（2048），2M
* socket\_id，内存应分配的socket，如果保留区没NUMA限制，则为SOCKET\_ID\_ANY，demo中为rte\_socket\_id()，运行逻辑核的物理套接字ID

## rte\_eth\_rx\_burst

端口接收队列中检索数据包，存入mbuf，

static uint16\_t rte\_eth\_rx\_burst ( uint16\_t port\_id,

uint16\_t queue\_id,

struct rte\_mbuf \*\* rx\_pkts,

const uint16\_t nb\_pkts

)

* port\_id,端口ID
* queue\_id,队列ID
* rx\_pkts,指向mbuf的指针数组地址
* nb\_pkts，队列最大数据包数

返回值：实际检索到的数据包梳理，也是rx\_pkts内有效指针数。

## rte\_pktmbuf\_mtod

返回指向mbuf第一个数据的指针

#define rte\_pktmbuf\_mtod(m,t )  rte\_pktmbuf\_mtod\_offset(m, t, 0)

* m,当前mbuf地址
* t,返回值类型，一般为结构体

## rte\_cpu\_to\_be\_16

rte\_pktmbuf\_mtod\_offset

ntohs

rte\_eth\_dev\_configure接口使得应用层可以对网卡进行配置下发操作，将应用层提供的网卡信息保存到网卡数据空间struct rte\_eth\_dev\_data中，这样pmd用户态驱动就可以根据应用层提供的配置对网卡进行设置操作。

另外这个接口还会为网卡的发送队列，接收队列开辟二级指针空间；

需要注意的是，每个网卡都有一个数据空间struct rte\_eth\_dev\_data结构，这个网卡数据空间在pmd驱动初始化过程中开辟， 这在上一篇dpdk pmd驱动初始化已经分析过了。

1）EAL: No available hugepages reported in hugepages-1048576kB

1048576kB 就是1G，这行 log 应该只是一个警告，因为我们根本没设置过 1G 的 hugepage，找不到是预料之中的。

/opt/dpdk-stable-19.11.14/x86\_64-native-linuxapp-gcc/build/app/test-pmd

Sh只能用quit退出

# l3fwd

dpdk-stable/examples/l3fwd 直接make编译

* -p，PortMask 参数指定使用的网口掩码，例如，假设当前有 4 个网口由 dpdk 进行管理，则 0x3 表示使用网口 0 和 网口 1，0xf 表示使用网口 0、网口 1、网口 2 和网口 3。
* -P，参数表示将所有网口设置为混杂模式，以便收到所有数据包
* --config (port,queue,lcore)[,(port,queue,lcore)] 参数用以配置网口、队列、核之间的对应关系，例如，--config (0,0,1) 表示网口 0 的队列 0 由核 1 进行处理。

./l3fwd -l 1-2 -n 4 -- -P -p 0x3 --config="(0,0,1),(1,0,2)

这个实例占用的逻辑核为1,2