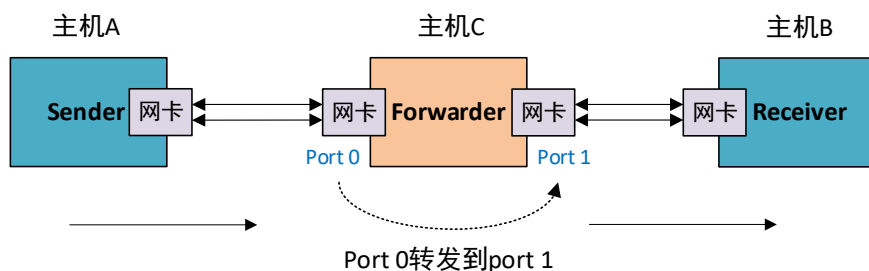


# DKDK 去 IP 分组转发

## 1、转发流程

如下图是一个转发示例，其中有三个实体，数据发送者 A、转发器 C、数据接收者 B（分别由主机 A、C、B 扮演），主机 A 发送数据包到转发器（主机 C），在 DPDK 中每一个 Port 绑定一个物理端口，转发器从 Port 0 收到数据包后解析以太网包头中的目的 MAC 地址，根据目的 MAC 地址选择下一跳端口（Port 1），然后将数据包放入 Port 1 的发送队列中发送给主机 B。



## 2、代码解析

代码可以分为这几块：

- 初始化 DPDK 环境。

- 检查可用端口数量（至少需要两个）。

- 创建内存池以存储数据包缓冲区。

- 初始化所有可用端口。

- 启动主处理循环。

与前面去 IP 化分组收发实验代码相比区别主要是多了检查端口数量的步骤以及处理主循环不同。

首先主机 C 作为转发器必须有两个及以上物理端口，所以要给主机 C 创建至少两张网卡。

```
// 检查可用端口数量
```

```
nb_ports = rte_eth_dev_count_avail();
```

```
printf("可用端口数量: %d\n", nb_ports);
```

```
if (nb_ports < 2)
```

```
    rte_exit(EXIT_FAILURE, "Error: number of ports must be >= 2\n");
```

处理主循环代码解释：

程序在这里使用一个无限循环来不断地检查和处理网络数据包。

**遍历所有端口：**通过 `rte_eth_dev_count_avail()` 获取可用的网络端口数量，依次遍历每个端口。

**接收数据包：**使用 `rte_eth_rx_burst` 函数从指定端口的接收队列中读取数据包，存储在 `bufs` 数组中。返回值 `nb_rx` 表示成功接收到的数据包数量。如果没有接收到数据包，程序继续下一次循环。

**遍历接收到的数据包：**对每个接收到的数据包进行处理。

**获取数据包和以太网头部：**从 `bufs` 数组中获取每个数据包的指针 `m`，并使用 `rte_pktmbuf_mtod` 函数获取以太网头部指针 `eth_hdr`。

**检查以太网类型：**通过 `ntohs` 函数将网络字节序转换为主机字节序，检查以太网类型是否为自定义类型（`CUSTOM_ETHER_TYPE`）（因为转发器内部会有其他程序一直通过这两张网卡发数据，这里我们加了一个过滤，仅让转发器转发我们新设计的数据包，也就是以太网类型为 `CUSTOM_ETHER_TYPE` 的数据包）

同时转发器还对传入数据包的自定义包头 `my_hdr` 的 `id` 字段进行了修改，将其设为了 2，后面在接收端主机 B 检查改字段内容就可以判断数据包是否经过了转发器。

**决定转发端口：**根据接收数据包的目标 MAC 地址的最后一个字节，取模可用端口的数量，确定数据包的转发端口。

这里转发端口的选择非常简易，后续可在此设计相关转发协议。

**发送数据包：**使用 `rte_eth_tx_burst` 函数将数据包发送到确定的目标端口，并打印出转发的目标端口。

这部分代码如下：

```
while (1) {
    // 遍历所有端口
    for (port = 0; port < rte_eth_dev_count_avail(); port++) {
        // 从接收队列中读取数据包
        const uint16_t nb_rx = rte_eth_rx_burst(port, 0, bufs, MAX_PKT_BURST);
        if (nb_rx == 0)
            continue;

        // 遍历接收到的数据包
        for (uint16_t i = 0; i < nb_rx; i++) {
            struct rte_mbuf* m = bufs[i];
            struct rte_ether_hdr* eth_hdr = rte_pktmbuf_mtod(m, struct rte_ether_hdr*);

            // 检查以太网类型是否为自定义类型
            if (ntohs(eth_hdr->ether_type) != CUSTOM_ETHER_TYPE) {
                rte_pktmbuf_free(m); // 释放不需要转发的数据包
                continue; // 跳过不转发的数据包
            }

            printf("从port %d 收到数据包\n", port);
            struct my_hdr* myhdr = (struct my_hdr*)(eth_hdr + 1);
            myhdr->id = rte_cpu_to_be_16(2);

            // 根据目的 MAC 地址决定转发端口
            uint16_t dst_port = (eth_hdr->d_addr.addr_bytes[5] % rte_eth_dev_count_avail());

            // 发送数据包
            printf("转发到port %d \n", dst_port);
            const uint16_t nb_tx = rte_eth_tx_burst(dst_port, 0, &m, 1);

            // 释放未能发送的数据包
            if (nb_tx < 1) {
                rte_pktmbuf_free(m);
            }
        }
    }
}
```

### 3、编译运行

主机 C 创建两张网卡，绑定驱动：

```
sudo modprobe uio
```

```
sudo insmod /home/lwj/Desktop/dpdk-kmods/linux/igb_uio/igb_uio.ko intr_mode=legacy
```

```
sudo ifconfig ens37 down
```

```
sudo ifconfig ens38 down
```

```
sudo dpdk-devbind.py --bind=igb_uio 0000:02:05.0
```

```
sudo dpdk-devbind.py --bind=igb_uio 0000:02:06.0
```

```
dpdk-devbind.py --status
```

将转发器代码 `l2fwd.c` 和 `CMakeLists.txt`（见附录）放入同一个文件中：

编译

```
mdkir build
```

```
cd build
```

```
cmake ..
```

```
make
```

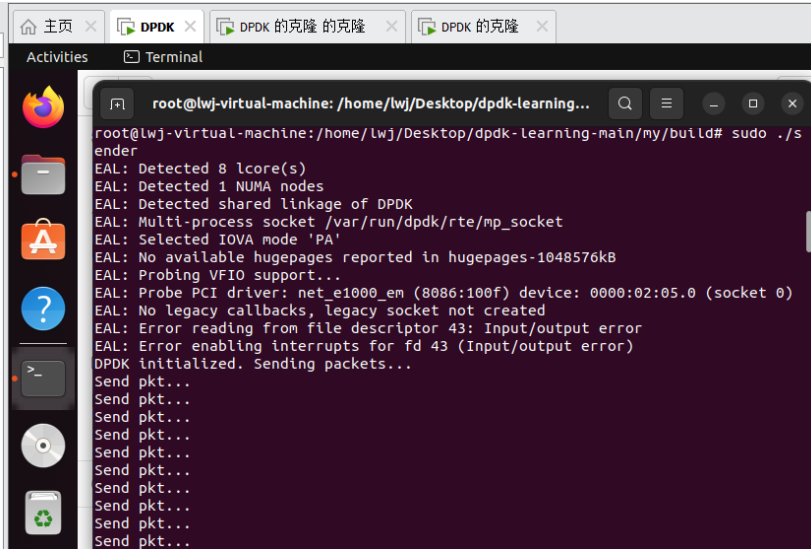
启动转发器：sudo ./l2fwd

后面按照 DPDK 发送去 IP 分组文档中的步骤，先启动数据发送者 Sender 再启动 Receiver

（注：启动顺序不能变，因为实验是在虚拟机里面进行的，每个主机的网卡是用同一个物理网卡虚拟出来的，所以主机网卡的连接方式和第一张图并不一样，而是所有的虚拟网卡连在了一起，如果最后启动转发器可能会导致数据包直接通过主机 A 网卡发送到主机 B 网卡，而不会经过转发器）

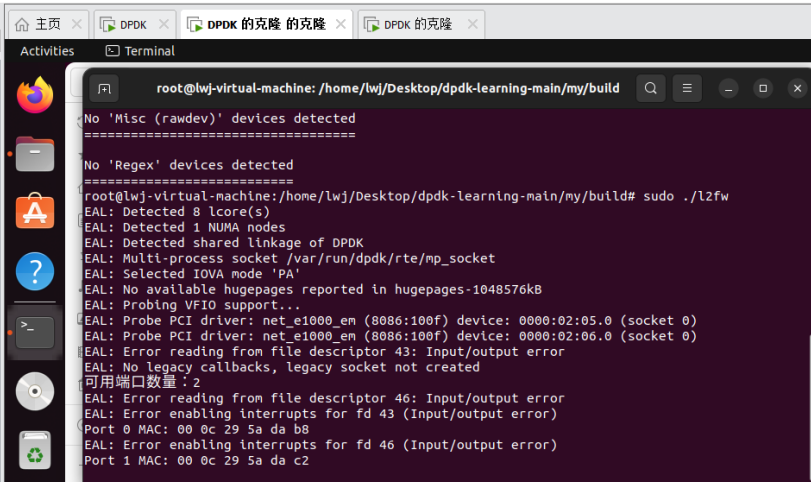
实验结果如下：

主机 A 不断发送数据包

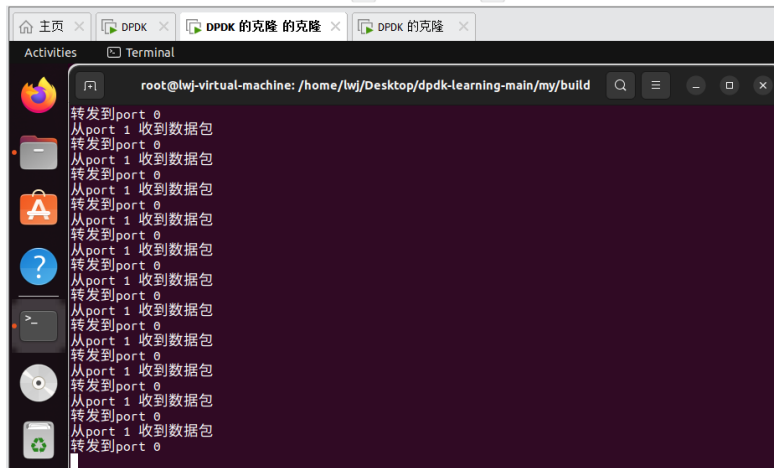


```
root@lwj-virtual-machine: /home/lwj/Desktop/dpdk-learning-main/my/build# sudo ./s
ender
EAL: Detected 8 lcore(s)
EAL: Detected 1 NUMA nodes
EAL: Detected shared linkage of DPDK
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
EAL: Selected IOVA mode 'PA'
EAL: No available hugepages reported in hugepages-1048576kB
EAL: Probing VFIO support...
EAL: Probe PCI driver: net_e1000_em (8086:100f) device: 0000:02:05.0 (socket 0)
EAL: No legacy callbacks, legacy socket not created
EAL: Error reading from file descriptor 43: Input/output error
EAL: Error enabling interrupts for fd 43 (Input/output error)
DPDK initialized. Sending packets...
Send pkt...
Send pkt...
Send pkt...
Send pkt...
Send pkt...
Send pkt...
Send pkt...
Send pkt...
Send pkt...
Send pkt...
```

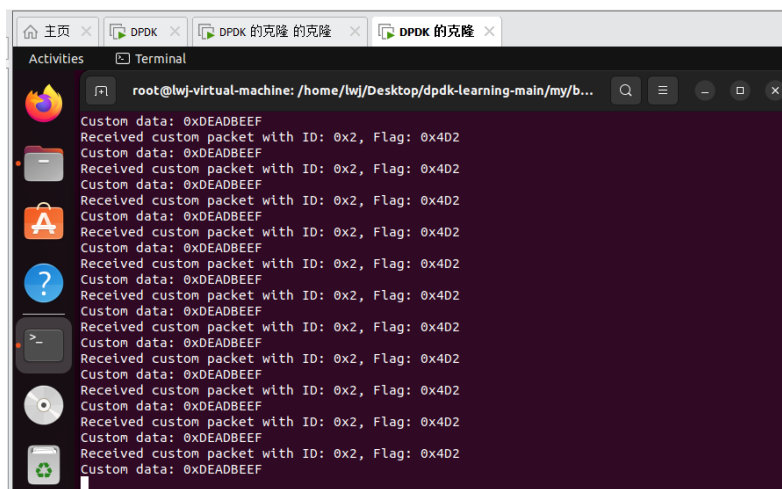
主机 C 启动时先打印出可用端口数量，以及每个端口的 MAC 地址，当执行收包转发时，打印数据包的入端口和出端口。



```
root@lwj-virtual-machine: /home/lwj/Desktop/dpdk-learning-main/my/build# sudo ./l2fw
No 'Misc (rawdev)' devices detected
=====
No 'Regex' devices detected
=====
root@lwj-virtual-machine: /home/lwj/Desktop/dpdk-learning-main/my/build# sudo ./l2fw
EAL: Detected 8 lcore(s)
EAL: Detected 1 NUMA nodes
EAL: Detected shared linkage of DPDK
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
EAL: Selected IOVA mode 'PA'
EAL: No available hugepages reported in hugepages-1048576kB
EAL: Probing VFIO support...
EAL: Probe PCI driver: net_e1000_em (8086:100f) device: 0000:02:05.0 (socket 0)
EAL: Probe PCI driver: net_e1000_em (8086:100f) device: 0000:02:06.0 (socket 0)
EAL: Error reading from file descriptor 43: Input/output error
EAL: No legacy callbacks, legacy socket not created
可用端口数量: 2
EAL: Error reading from file descriptor 46: Input/output error
EAL: Error enabling interrupts for fd 43 (Input/output error)
Port 0 MAC: 00 0c 29 5a da b8
EAL: Error enabling interrupts for fd 46 (Input/output error)
Port 1 MAC: 00 0c 29 5a da c2
```



主机 B 收到数据包后发现 id 字段被修改为 2, 说明该数据包是经由转发器转发给主机 B 的。



## 附录

### l2fwd.c

```
#include <rte_eal.h>
#include <rte_ethdev.h>
#include <rte_mbuf.h>
#include <rte_lcore.h>
#include <rte_ether.h>
#include <stdio.h>

#define MAX_PKT_BURST 32
#define MEMPOOL_CACHE_SIZE 256
#define CUSTOM_ETHER_TYPE 0x88B5 // 自定义以太网类型

static const struct rte_eth_conf port_conf_default = {
    .rxmode = {
        .max_rx_pkt_len = RTE_ETHER_MAX_LEN,
```

```

    },
};

struct my_hdr {
    uint16_t id;
    uint16_t flag;
};

// 初始化端口函数
static int port_init(uint16_t port, struct rte_mempool* mbuf_pool) {
    struct rte_eth_conf port_conf = port_conf_default;
    const uint16_t rx_rings = 1, tx_rings = 1;
    uint16_t nb_rxd = 1024;
    uint16_t nb_txd = 1024;
    int retval;
    uint16_t q;
    struct rte_eth_addr addr; // 定义一个 ether_addr 类型的变量用于存储MAC地址

    // 检查端口是否有效
    if (port >= rte_eth_dev_count_avail())
        return -1;

    // 配置端口
    retval = rte_eth_dev_configure(port, rx_rings, tx_rings, &port_conf);
    if (retval != 0)
        return retval;

    // 分配和设置 RX 队列
    for (q = 0; q < rx_rings; q++) {
        retval = rte_eth_rx_queue_setup(port, q, nb_rxd,
            rte_eth_dev_socket_id(port), NULL, mbuf_pool);
        if (retval < 0)
            return retval;
    }

    // 分配和设置 TX 队列
    for (q = 0; q < tx_rings; q++) {
        retval = rte_eth_tx_queue_setup(port, q, nb_txd,
            rte_eth_dev_socket_id(port), NULL);
        if (retval < 0)
            return retval;
    }

    // 启动端口

```

```

    retval = rte_eth_dev_start(port);
    if (retval < 0)
        return retval;

    //打印MAC
    rte_eth_macaddr_get(port, &addr);
    printf("Port %u MAC: %02" PRIx8 " %02" PRIx8 " %02" PRIx8
           " %02" PRIx8 " %02" PRIx8 " %02" PRIx8 "\n",
           (unsigned)port,
           addr.addr_bytes[0], addr.addr_bytes[1],
           addr.addr_bytes[2], addr.addr_bytes[3],
           addr.addr_bytes[4], addr.addr_bytes[5]);

    // 启用混杂模式（可选）
    rte_eth_promiscuous_enable(port);

    return 0;
}

// 主循环函数
static void l2fwd_main_loop(void) {
    uint16_t port;
    struct rte_mbuf* bufs[MAX_PKT_BURST];
    unsigned lcore_id = rte_lcore_id();

    // 检查每个端口的 NUMA 节点
    for (port = 0; port < rte_eth_dev_count_avail(); port++) {
        if (rte_eth_dev_socket_id(port) != (int)rte_socket_id()) {
            printf("Warning: port %u is on remote NUMA node to polling thread.\n"
                   "Performance will not be optimal.\n", port);
        }
    }
}

// 无限循环处理数据包
while (1) {
    // 遍历所有端口
    for (port = 0; port < rte_eth_dev_count_avail(); port++) {
        // 从接收队列中读取数据包
        const uint16_t nb_rx = rte_eth_rx_burst(port, 0, bufs, MAX_PKT_BURST);
        if (nb_rx == 0)
            continue;

        // 遍历接收到的数据包
        for (uint16_t i = 0; i < nb_rx; i++) {

```

```

        struct rte_mbuf* m = bufs[i];
        struct rte_ether_hdr* eth_hdr = rte_pktmbuf_mtod(m, struct
rte_ether_hdr*);

        // 检查以太网类型是否为自定义类型
        if (ntohs(eth_hdr->ether_type) != CUSTOM_ETHER_TYPE) {
            rte_pktmbuf_free(m); // 释放不需要转发的数据包
            continue; // 跳过不转发的数据包
        }
        printf("从port %d 收到数据包\n", port);

        struct my_hdr* myhdr = (struct my_hdr*)(eth_hdr + 1);
        myhdr->id = rte_cpu_to_be_16(2);

        // 根据目的 MAC 地址决定转发端口
        uint16_t dst_port = (eth_hdr->d_addr.addr_bytes[5] %
rte_eth_dev_count_avail());

        // 发送数据包
        printf("转发到port %d \n", dst_port);
        const uint16_t nb_tx = rte_eth_tx_burst(dst_port, 0, &m, 1);

        // 释放未能发送的数据包
        if (nb_tx < 1) {
            rte_pktmbuf_free(m);
        }
    }
}

}

}

int main(int argc, char** argv) {
    struct rte_mempool* mbuf_pool;
    unsigned nb_ports;
    uint16_t portid;

    // 初始化 DPDK 环境
    int ret = rte_eal_init(argc, argv);
    if (ret < 0)
        rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
    argc -= ret;
    argv += ret;

    // 检查可用端口数量

```

```

nb_ports = rte_eth_dev_count_avail();
printf("可用端口数量: %d\n", nb_ports);
if (nb_ports < 2)
    rte_exit(EXIT_FAILURE, "Error: number of ports must be >= 2\n");

// 创建内存池
mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", 8192 * nb_ports,
    MEMPOOL_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());
if (mbuf_pool == NULL)
    rte_exit(EXIT_FAILURE, "Cannot create mbuf pool\n");

// 初始化所有端口
RTE_ETH_FOREACH_DEV(portid) {
    if (port_init(portid, mbuf_pool) != 0)
        rte_exit(EXIT_FAILURE, "Cannot init port %" PRIu16 "\n", portid);
}

// 启动主循环
l2fwd_main_loop();

return 0;
}

```

## 2、CMakeLists.txt

```

cmake_minimum_required(VERSION 3.10)
project(dpdk_ping)

# 设置 C 标准
set(CMAKE_C_STANDARD 99)
set(CMAKE_C_STANDARD_REQUIRED ON)

# 设定编译选项
add_compile_options(-O3 -march=native)
add_definitions(-DALLOW_EXPERIMENTAL_API)

# 查找 DPDK 包
find_package(PkgConfig REQUIRED)
pkg_check_modules(RTE REQUIRED libdpdk)

# 包含 DPDK 头文件
include_directories(${RTE_INCLUDE_DIRS})

# 添加可执行文件

```



```
add_executable(l2fw l2fw.c)
```

```
# 链接 DPDK 库
```

```
target_link_libraries(l2fw ${RTE_LIBRARIES} m)
```

```
# 设置库搜索路径
```

```
link_directories(/usr/local/lib/x86_64-linux-gnu)
```

```
# 链接选项
```

```
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Wl,--as-needed")
```