

## 一 DPDK 环境安装

系统: ubuntu20.04.6

DPDK 版本: 20.11.10

添加桥接网卡, 进入虚拟机文件所在目录, 用记事本打开 vmx 文件, 找到 ethernet1.virtualDev="e1000", 改为 ethernet1.virtualDev="vmxnet3"。并添加 ethernet1.wakeOnPcktRcv="TRUE"。同理 ethernet2。

ifconfig 得到 ens192, MAC: 00:0c:29:2b:01:d4; ens224, MAC: 00:0c:29:2b:01:de

更改 GRUB 启动参数, 配置巨页信息:

```
vim /etc/default/grub
GRUB_CMDLINE_LINUX="transparent_hugepage=never                default_hugepagesz=2M
hugepagesz=2M hugepages=2048"(2048 个, 每个 2M, 共 4G)
sudo update-grub
reboot
grep Huge /proc/meminfo
```

注:

DPDK19.11 及之前版本的编译方式及编译工具为 GCC Make。

DPDK19.11 之后版本编译工具为 meson ninja。

安装编译依赖:

```
mkdir /share
cd /share
sudo apt install build-essential python3-pip python3-pyelftools libnuma-dev libpcap0.8-dev pkg-config
sudo pip3 install meson ninja
```

编译 DPDK:

```
wget http://fast.dpdk.org/rel/dpdk-20.11.10.tar.xz
tar -xvf dpdk-20.11.10.tar.xz
cd dpdk-stable-20.11.10
meson -Dexamples=all build
cd build
ninja
sudo ninja install
sudo ldconfig
```

### 加载网卡驱动:

```
git clone http://dpdk.org/git/dpdk-kmods
cd dpdk-kmods/linux/igb_uio
make
sudo modprobe uio
sudo insmod /share/dpdk-kmods/linux/igb_uio/igb_uio.ko intr_mode=legacy
lsmod | grep igb_uio
```

### DPDK 绑定网口:

```
cd /share/dpdk-stable-20.11.10
usertools/dpdk-devbind.py --status (0000:0b:00.0, 0000:13:00.0)
ifconfig ens192 down
ifconfig ens224 down
usertools/dpdk-devbind.py --bind=igb_uio 0000:0b:00.0
usertools/dpdk-devbind.py --bind=igb_uio 0000:13:00.0
usertools/dpdk-devbind.py --status
```

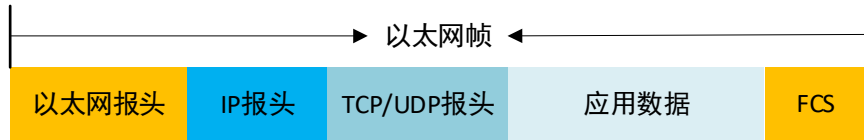
### 测试程序:

```
cd build/examples
./dpdk-helloworld (最后出现 hello from core x 就是成功了)
```

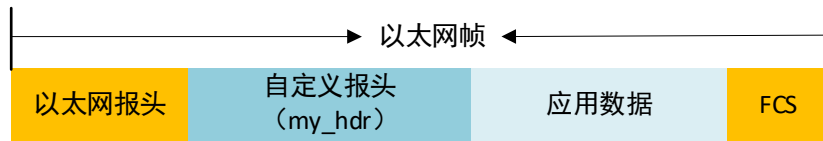
## 二、去 IP 数据包构造与传输

### 1、包结构设计

下图是以太网帧结构，当网卡接收到数据包后，会触发中断请求，内核逐层解析数据包的头部信息，提取出应用数据，并将其上传至用户态空间。



通过使用 DPDK，可以避免频繁的中断请求，并直接将以太网帧上传到应用层的用户态空间进行处理。为了去除 TCP/IP 协议，设计了如下图的以太网帧结构。



在我们的设计中删除了原有的以太网 IP 包头和 TCP/UDP 包头，然后加入了自定义的包头 my\_hdr，因为没有了 IP 包头，网络实体间将直接通过 MAC 地址通信。

在自定义包头中暂时添加了两个字段：id、flag，后续可根据协议栈需求增加其他字段。

```
struct my_hdr {  
    uint16_t id;  
    uint16_t flag;  
};
```

### 2、自定义包结构代码解析

首先给自定义包选择一个以太类型 (EtherType)，以太类型是用于表示上层协议类型，占 2 字节。例如，0x0800 表示 IPv4，0x0806 表示 ARP，0x86DD 表示 IPv6。

```
#define CUSTOM_ETHER_TYPE 0x88B5 // 自定义以太网类型
```

这里选择 0x88B5 为自定义以太网类型。

下面讲解如何构造自定义数据包，首先分配 mbuf，mbuf 是 DPDK 中存储和管理数据包的基本单位。

```
struct rte_mbuf* mbuf = rte_pktmbuf_alloc(mbuf_pool); // 分配内存池中的 mbuf
```

然后计算包的长度，这里的数据包由三部分组成：以太网头、自定义包头、数据。从 mbuf 结构中获取指向实际数据区域的指针 pktdata。

通过 pktdata 可以获得指向以太网包头的指针 eth。

```
// 自定义数据包的总长度 // 以太网头 + 自定义包头 + 自定义数据  
const unsigned total_len = sizeof(struct rte_ether_hdr) + sizeof(struct my_hdr) + sizeof(uint32_t);  
mbuf->pkt_len = total_len;  
mbuf->data_len = total_len;  
  
uint8_t* pktdata = rte_pktmbuf_mtod(mbuf, uint8_t*); // 获取数据指针  
struct rte_ether_hdr* eth = (struct rte_ether_hdr*)pktdata;
```

获得以太网头指针 eth 后就可以填充以太网头了，这里填充了三个字段，源 MAC、目的 MAC 和以太类型。

```
// 设置以太网头
rte_memcpy(eth->s_addr.addr_bytes, src_mac, RTE_ETHER_ADDR_LEN); // 设置源MAC
rte_ether_unformat_addr(DEST_MAC, &eth->d_addr); // 设置目标MAC
eth->ether_type = htons(CUSTOM_ETHER_TYPE); // 设置以太类型为自定义类型
```

后面利用 eth 指针可以获得自定义包头指针和数据指针，分别填充数据，这里填充自定义包头中 ip 为 1，flag 为 1234，填充数据为 0x0xDEADBEEF

```
//自定义包头内容
struct my_hdr* myhdr = (struct my_hdr*)(eth + 1);
myhdr->id = rte_cpu_to_be_16(1);
myhdr->flag = rte_cpu_to_be_16(1234);
//自定义数据内容
uint32_t* custom_data = (uint32_t*)(myhdr + 1);
*custom_data = htonl(0xDEADBEEF); // 设置自定义数据
```

到这里包就构建完成了，构造自定义包完整代码如下：

```
72 // 发送自定义数据包
73 static void send_custom_packet(struct rte_mempool* mbuf_pool) {
74     struct rte_mbuf* mbuf = rte_pktmbuf_alloc(mbuf_pool); // 分配内存池中的mbuf
75     if (mbuf == NULL) {
76         rte_exit(EXIT_FAILURE, "Failed to allocate mbuf for custom packet\n");
77     }
78
79     // 自定义数据包的总长度 // 以太网头 + 自定义包头 + 自定义数据
80     const unsigned total_len = sizeof(struct rte_ether_hdr) + sizeof(struct my_hdr) + sizeof(uint32_t);
81     mbuf->pkt_len = total_len;
82     mbuf->data_len = total_len;
83
84     uint8_t* pktdata = rte_pktmbuf_mtod(mbuf, uint8_t*); // 获取数据指针
85     struct rte_ether_hdr* eth = (struct rte_ether_hdr*)pktdata;
86
87     // 设置以太网头
88     rte_memcpy(eth->s_addr.addr_bytes, src_mac, RTE_ETHER_ADDR_LEN); // 设置源MAC
89     rte_ether_unformat_addr(DEST_MAC, &eth->d_addr); // 设置目标MAC
90     eth->ether_type = htons(CUSTOM_ETHER_TYPE); // 设置以太类型为自定义类型
91
92     //自定义包头内容
93     struct my_hdr* myhdr = (struct my_hdr*)(eth + 1);
94     myhdr->id = rte_cpu_to_be_16(1);
95     myhdr->flag = rte_cpu_to_be_16(1234);
96     //自定义数据内容
97     uint32_t* custom_data = (uint32_t*)(myhdr + 1);
98     *custom_data = htonl(0xDEADBEEF); // 设置自定义数据
99
100     // 发送数据包
101     rte_eth_tx_burst(gDpdkPortId, 0, &mbuf, 1);
102     rte_pktmbuf_free(mbuf); // 释放mbuf
103 }
```

### 3、发包与收包代码解析

发包代码关键部分已经展示在了上一张图片中，总结流程如下：

**分配 mbuf：**从内存池中分配一个 mbuf，用于存储数据包。

**构建以太网头：**

设置源 MAC 地址为本地 MAC。

设置目标 MAC 地址。

指定自定义以太网类型 (CUSTOM\_ETHER\_TYPE)。

**构建自定义包头：**在以太网头后添加自定义头 my\_hdr，填入 id 和 flag 字段。

**添加自定义数据：**在自定义头后追加数据字段，赋值为 0xDEADBEEF。

**发送数据包：**通过 rte\_eth\_tx\_burst 函数将构建的数据包发送到指定端口。

**释放 mbuf：**发送完成后，释放 mbuf 资源。

收包代码关键部分如下:

```
// 主循环, 处理接收到的报文
while (keep_running) {
    struct rte_mbuf* mbufs[BURST_SIZE];
    unsigned num_rcvd = rte_eth_rx_burst(gDpdkPortId, 0, mbufs, BURST_SIZE); // 从接收队列中接收报文
    if (num_rcvd == 0) {
        continue;
    }

    // 处理接收到的每个报文
    for (unsigned i = 0; i < num_rcvd; i++) {
        handle_custom_packet(mbufs[i]); // 处理自定义数据包
        rte_pktmbuf_free(mbufs[i]); // 释放mbuf
    }
}

// 处理接收到的自定义数据包
static void handle_custom_packet(struct rte_mbuf* mbuf) {
    struct rte_ether_hdr* eth = rte_pktmbuf_mtod(mbuf, struct rte_ether_hdr*);

    if (eth->ether_type == htons(CUSTOM_ETHER_TYPE)) {
        // 获取以太网头部之后的数据, 即自定义协议头部分
        struct my_hdr* my_header = (struct my_hdr*)((char*)eth + sizeof(struct rte_ether_hdr));

        // 将id和flag转换为主机字节序并打印出来
        uint16_t id = ntohs(my_header->id);
        uint16_t flag = ntohs(my_header->flag);

        printf("Received custom packet with ID: 0x%X, Flag: 0x%X\n", id, flag);

        // 获取自定义数据部分
        uint32_t* custom_data = (uint32_t*)((char*)my_header + sizeof(struct my_hdr));
        printf("Custom data: 0x%X\n", ntohl(*custom_data));
    }
}
```

总结收包流程:

#### 接收数据包:

使用 `rte_eth_rx_burst` 函数从接收队列中接收一个批次的数据包 (`BURST_SIZE` 大小), 存储到 `mbufs` 数组中。

#### 遍历处理每个数据包:

对于每个接收到的数据包, 调用 `handle_custom_packet` 进行处理。

#### 解析自定义数据包:

从数据包中获取以太网头部, 检查其以太网类型是否匹配自定义类型 (`CUSTOM_ETHER_TYPE`)。

如果匹配, 解析自定义协议头 `my_hdr` 的 `id` 和 `flag` 字段 (转换为主机字节序后打印)。

读取自定义数据部分, 并打印该数据。

#### 释放 mbuf:

处理完数据包后, 调用 `rte_pktmbuf_free` 释放 `mbuf` 资源。

## 4、编译运行

创建两台虚拟机: 主机 A、主机 B

主机 A 作为发送端 Sender

主机 B 作为接收端 Receiver

在两台主机上都创建一张桥接模式的网卡, 使用这两张网卡进行互相通信

获取主机 B 网卡的 MAC 地址

网卡绑定 igb\_uio 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 dpdk-devbind.py --bind=igb_uio 0000:02:05.0
```

将 Sender.c 、 Receiver.c、CMakeLists.txt 这三个文件（见附录）放入同一个文件夹中，终端进入该文件夹目录。

修改 Sender.c 中目标 MAC 地址为主机 B 网卡的 MAC 地址

```
#define DEST_MAC "01:02:03:04:05:06" // 目标 MAC 地址
```

编译：

```
mkdir build
```

```
cd build
```

```
cmake ..
```

```
make
```

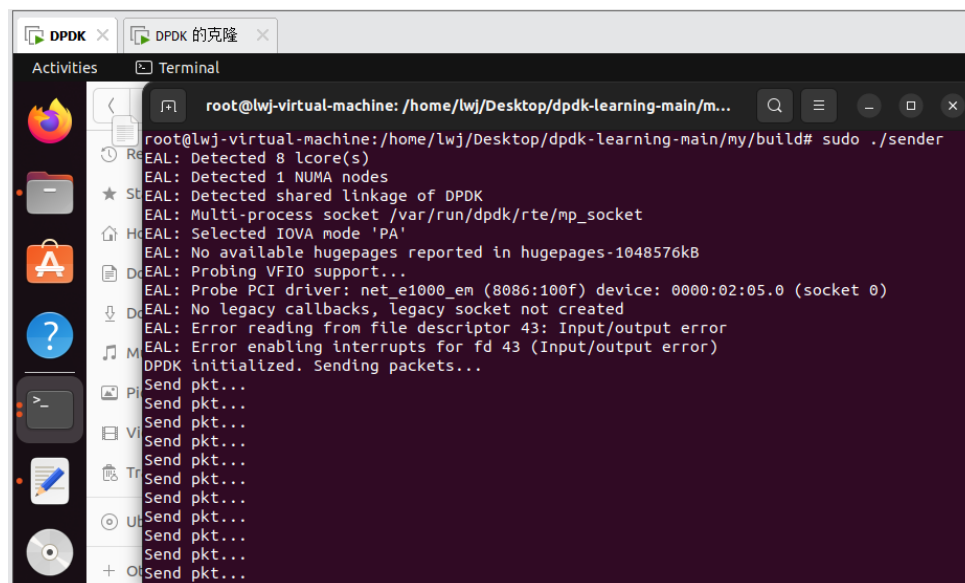
运行：

主机 A ： `sudo ./sender`

主机 B ： `sudo ./receiver`

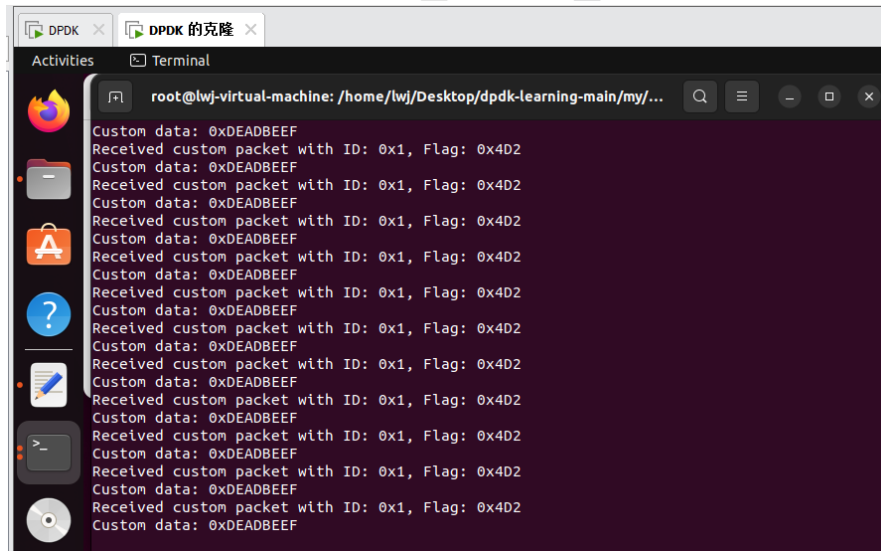
运行结果如下

主机 A 不断发送数据包：



```
root@lwj-virtual-machine: /home/lwj/Desktop/dpdk-learning-main/my/build# sudo ./sender
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...
Send pkt...
Send pkt...
```

主机 B 收到数据包，解析出自定义包头的 id 和 flag 字段以及自定义的数据部分。



## 附录

### 1、Sender.c

```
#include <rte_eal.h>
#include <rte_ethdev.h>
#include <rte_mbuf.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <signal.h>

#define NUM_MBUFS (8192)
#define BURST_SIZE 32
#define CUSTOM_ETHER_TYPE 0x88B5 // 自定义以太网类型
#define DEST_MAC "01:02:03:04:05:06" // 目标MAC地址

static uint8_t src_mac[RTE_ETHER_ADDR_LEN]; // 本地MAC地址
int gDpdkPortId = 0; // 端口ID
volatile int keep_running = 1; // 控制主循环的标志

struct my_hdr {
    uint16_t id;
    uint16_t flag;
};

// 默认端口配置
static const struct rte_eth_conf port_conf_default = {
    .rxmode = {
        .max_rx_pkt_len = RTE_ETHER_MAX_LEN,
```

```

    },
};

// 信号处理函数
void signal_handler(int signum) {
    if (signum == SIGINT) {
        printf("Caught SIGINT, preparing to exit...\n");
        keep_running = 0; // 设置退出标志
    }
}

// 初始化DPDK端口
static void init_port(struct rte_mempool* mbuf_pool) {
    uint16_t nb_sys_ports = rte_eth_dev_count_avail();
    if (nb_sys_ports == 0) {
        rte_exit(EXIT_FAILURE, "No Ethernet ports available\n");
    }

    struct rte_eth_dev_info dev_info;
    rte_eth_dev_info_get(gDpdkPortId, &dev_info);

    const int num_rx_queues = 1;
    const int num_tx_queues = 1;
    struct rte_eth_conf port_conf = port_conf_default;

    rte_eth_dev_configure(gDpdkPortId, num_rx_queues, num_tx_queues, &port_conf);

    if (rte_eth_rx_queue_setup(gDpdkPortId, 0, 128, rte_eth_dev_socket_id(gDpdkPortId),
        NULL, mbuf_pool) < 0) {
        rte_exit(EXIT_FAILURE, "Failed to set up RX queue\n");
    }

    struct rte_eth_txconf txq_conf = dev_info.default_txconf;
    txq_conf.offloads = port_conf.rxmode.offloads;

    if (rte_eth_tx_queue_setup(gDpdkPortId, 0, 1024,
        rte_eth_dev_socket_id(gDpdkPortId), &txq_conf) < 0) {
        rte_exit(EXIT_FAILURE, "Failed to set up TX queue\n");
    }

    if (rte_eth_dev_start(gDpdkPortId) < 0) {
        rte_exit(EXIT_FAILURE, "Failed to start Ethernet device\n");
    }
}

```



```

    rte_eth_promiscuous_enable(gDpdkPortId); // 启用混杂模式，接收所有包
}

// 发送自定义数据包
static void send_custom_packet(struct rte_mempool* mbuf_pool) {
    struct rte_mbuf* mbuf = rte_pktmbuf_alloc(mbuf_pool); // 分配内存池中的mbuf
    if (mbuf == NULL) {
        rte_exit(EXIT_FAILURE, "Failed to allocate mbuf for custom packet\n");
    }

    // 自定义数据包的总长度// 以太网头 + 自定义包头 + 自定义数据
    const unsigned total_len = sizeof(struct rte_ether_hdr) + sizeof(struct my_hdr) +
sizeof(uint32_t);
    mbuf->pkt_len = total_len;
    mbuf->data_len = total_len;

    uint8_t* pktdata = rte_pktmbuf_mtod(mbuf, uint8_t*); // 获取数据指针
    struct rte_ether_hdr* eth = (struct rte_ether_hdr*)pktdata;

    // 设置以太网头
    rte_memcpy(eth->s_addr.addr_bytes, src_mac, RTE_ETHER_ADDR_LEN); // 设置源MAC
    rte_ether_unformat_addr(DEST_MAC, &eth->d_addr); // 设置目标MAC
    eth->ether_type = htons(CUSTOM_ETHER_TYPE); // 设置以太类型为自定义类型

    //自定义包头内容
    struct my_hdr* myhdr = (struct my_hdr*)(eth + 1);
    myhdr->id = rte_cpu_to_be_16(1);
    myhdr->flag = rte_cpu_to_be_16(1234);
    // 自定义数据内容
    uint32_t* custom_data = (uint32_t*)(myhdr + 1);
    *custom_data = htonl(0xDEADBEEF); // 设置自定义数据

    // 发送数据包
    rte_eth_tx_burst(gDpdkPortId, 0, &mbuf, 1);
    printf("send pkt...\n");
    rte_pktmbuf_free(mbuf); // 释放mbuf
}

int main(int argc, char* argv[]) {
    // 注册信号处理器
    signal(SIGINT, signal_handler);

    if (rte_eal_init(argc, argv) < 0) {
        rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
    }
}

```

```

    }

    struct rte_mempool* mbuf_pool = rte_pktmbuf_pool_create("mbuf_pool", NUM_MBUFS, 0,
0, RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());
    if (mbuf_pool == NULL) {
        rte_exit(EXIT_FAILURE, "Could not create mbuf pool\n");
    }

    init_port(mbuf_pool); // 初始化DPDK端口
    rte_eth_macaddr_get(gDpdkPortId, (struct rte_ether_addr*)src_mac); // 获取本地MAC
地址
    printf("DPDK initialized. Sending packets...\n");

    // 发送自定义数据包
    while (keep_running) {
        send_custom_packet(mbuf_pool); // 持续发送数据包
        rte_delay_us(100000); // 发送间隔, 100毫秒
    }

    printf("Exiting...\n");
    return 0;
}

```

## 2、Receiver.c

```

#include <rte_eal.h>
#include <rte_ethdev.h>
#include <rte_mbuf.h>
#include <stdio.h>
#include <stdint.h>
#include <arpa/inet.h>
#include <signal.h>

#define NUM_MBUFS (8192)
#define BURST_SIZE 32
#define CUSTOM_ETHER_TYPE 0x88B5 // 自定义以太网类型

int gDpdkPortId = 0; // 端口ID
volatile int keep_running = 1; // 控制主循环的标志

// 自定义协议头结构体
struct my_hdr {
    uint16_t id; // 16位的id
    uint16_t flag; // 16位的flag
}

```

```
};
```

```
// 默认端口配置
```

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

```
// 信号处理函数
```

```
void signal_handler(int signum) {  
    if (signum == SIGINT) {  
        printf("Caught SIGINT, preparing to exit...\n");  
        keep_running = 0; // 设置退出标志  
    }  
}
```

```
// 初始化DPDK端口
```

```
static void init_port(struct rte_mempool* mbuf_pool) {  
    uint16_t nb_sys_ports = rte_eth_dev_count_avail();  
    if (nb_sys_ports == 0) {  
        rte_exit(EXIT_FAILURE, "No Ethernet ports available\n");  
    }  
  
    struct rte_eth_dev_info dev_info;  
    rte_eth_dev_info_get(gDpdkPortId, &dev_info);  
  
    const int num_rx_queues = 1;  
    const int num_tx_queues = 1;  
    struct rte_eth_conf port_conf = port_conf_default;  
  
    rte_eth_dev_configure(gDpdkPortId, num_rx_queues, num_tx_queues, &port_conf);  
  
    if (rte_eth_rx_queue_setup(gDpdkPortId, 0, 128, rte_eth_dev_socket_id(gDpdkPortId),  
    NULL, mbuf_pool) < 0) {  
        rte_exit(EXIT_FAILURE, "Failed to set up RX queue\n");  
    }  
  
    struct rte_eth_txconf txq_conf = dev_info.default_txconf;  
    txq_conf.offloads = port_conf.rxmode.offloads;  
  
    if (rte_eth_tx_queue_setup(gDpdkPortId, 0, 1024,  
    rte_eth_dev_socket_id(gDpdkPortId), &txq_conf) < 0) {
```

```

    rte_exit(EXIT_FAILURE, "Failed to set up TX queue\n");
}

if (rte_eth_dev_start(gDpdkPortId) < 0) {
    rte_exit(EXIT_FAILURE, "Failed to start Ethernet device\n");
}

rte_eth_promiscuous_enable(gDpdkPortId); // 启用混杂模式，接收所有包
}

// 处理接收到的自定义数据包
static void handle_custom_packet(struct rte_mbuf* mbuf) {
    struct rte_ether_hdr* eth = rte_pktmbuf_mtod(mbuf, struct rte_ether_hdr*);

    if (eth->ether_type == htons(CUSTOM_ETHER_TYPE)) {
        // 获取以太网头部之后的数据，即自定义协议头部分
        struct my_hdr* my_header = (struct my_hdr*)((char*)eth + sizeof(struct
rte_ether_hdr));

        // 将id和flag转换为主机字节序并打印出来
        uint16_t id = ntohs(my_header->id);
        uint16_t flag = ntohs(my_header->flag);

        printf("Received custom packet with ID: 0x%X, Flag: 0x%X\n", id, flag);

        // 获取自定义数据部分
        uint32_t* custom_data = (uint32_t*)((char*)my_header + sizeof(struct my_hdr));
        printf("Custom data: 0x%X\n", ntohl(*custom_data));
    }
}

int main(int argc, char* argv[]) {
    // 注册信号处理器
    signal(SIGINT, signal_handler);

    if (rte_eal_init(argc, argv) < 0) {
        rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
    }

    struct rte_mempool* mbuf_pool = rte_pktmbuf_pool_create("mbuf_pool", NUM_MBUFS, 0,
0, RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());
    if (mbuf_pool == NULL) {
        rte_exit(EXIT_FAILURE, "Could not create mbuf pool\n");
    }
}

```

```

init_port(mbuf_pool); // 初始化DPDK端口
printf("DPDK initialized. Waiting for packets...\n");

// 主循环，处理接收到的报文
while (keep_running) {
    struct rte_mbuf* mbufs[BURST_SIZE];
    unsigned num_recvd = rte_eth_rx_burst(gDpdkPortId, 0, mbufs, BURST_SIZE); //
从接收队列中接收报文
    if (num_recvd == 0) {
        continue;
    }

    // 处理接收到的每个报文
    for (unsigned i = 0; i < num_recvd; i++) {
        handle_custom_packet(mbufs[i]); // 处理自定义数据包
        rte_pktmbuf_free(mbufs[i]); // 释放mbuf
    }
}

printf("Exiting...\n");
return 0;
}

```

### 3、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(sender Sender.c)

add\_executable(receiver Receiver.c)

# 链接 DPDK 库

target\_link\_libraries(sender \${RTE\_LIBRARIES} m)

target\_link\_libraries(receiver \${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")