****

DPDK 学习

L2 fwd 代码走读报告

导师：***　　学生：***

2016-2-18

目录

一、对于 DPDK 的认识：

主要应用 x86 通用平台 ,转发处理网络数据包 ,定位在不需要专用网络处理器，但通用网络处理器对数据处理性能又不能满足需求的客户。

DPDK，搭载 x86 服务器，成本变化不大，但对数据的处理性能又有非常显著的提高 ,对传统 linux 技术做一定的优化 ,特别之处在于 :hugepage,uio,zero copy, cpu affinity 等。

关于 hugetlbpage（在这块大页面上做自己的内存管理系统），之前讲过，它的主要好处当然是通过利用大内存页提高内存使用效率，。由于 DPDK 是应用层平台，所以与此紧密相连的网卡驱动程序（当然，主要是 intel 自身的千兆 igb 与万兆 ixgbe 驱动程序）都通过 uio(用户层驱动、轮询、0 拷贝)机制运行在用户态下。cpu affinity（多核架构，核线程绑定物理核）机制是多核 cpu 发展的结果，，在越来越多核心的 cpu 机器上，如何提高外设以及程序工作效率的最直观想法就是让各个 cpu 核心各自干专门的事情，比如两个网卡 eth0 和 eth1 都收包，可以让 cpu0 专心处理 eth0，cpu1 专心处理 eth1，没必要 cpu0 一下处理 eth0，一下又处理 eth1，还有一个网卡多队列的情况也是类似，等等，DPDK 利用 cpu affinity 主要是将控制面线程以及各个数据面线程绑定到不同的 cpu，省却了来回反复调度的性能消耗，各个线程一个 while 死循环，专心致志的做事，互不干扰（当然还是有通信的，比如控制面接收用户配置，转而传递给数据面的参数设置等）。

总结如下：

1、 使用大页缓存支持来提高内存访问效率。

2、 利用 UIO 支持，提供应用空间下驱动程序的支持，也就是说网卡驱动是运行在用户空间 的，减下了报文在用户空间和应用空间的多次拷贝。

3、 利用 LINUX 亲和性支持，把控制面线程及各个数据面线程绑定到不同的 CPU 核，节省了 线程在各个 CPU 核来回调度。

4、 提供内存池和无锁环形缓存管理，加快内存访问效率。

在 x86 服务器，1G/10G/40G 网卡包转发，64Byte 小包，基本能做到 70%以上的转发，而传统 linux 系统只能达 5%左右，在网络大数据流时代，DPDK 加码，优势明显。

二、对 **L2fwd** 的认识：

2.1 运行配置

虚拟机软件：VMWare WorkStation 12.0.0 build-2985596

CPU： 2 个 CPU， 每个 CPU2 个核心

内存： 1GB+

网卡：intel 网卡*2， 用于 dpdk 试验；另一块网卡用于和宿主系统进行通信

2.2 运行环境搭建

在 root 权限下：

1）编译 dpdk

进入 dpdk 主目录<dpdk>，输入

make install T=x86_64-native-linuxapp-gcc 进行编译

2）配置大页内存（非 NUMA）

echo 128 >

/sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages

mkdir /mnt/huge

mount -t hugetlbfs nodev /mnt/huge

可以用以下命令查看大页内存状态：

cat /proc/meminfo | grep Huge

3）安装 igb_uio 驱动

modprobe uio

insmod x86_64-native-linuxapp-gcc/kmod/igb_uio.ko

4）绑定网卡

先看一下当前网卡的状态

./tools/dpdk_nic_bind.py --status

```
[root@bogon dpdk-1.7.0]#  ./tools/dpdk_nic_bind.py --status

Network devices using DPDK-compatible driver
============================================
0000:02:05.0 '82545EM Gigabit Ethernet Controller (Copper)' drv=igb_uio unused=e
1000
0000:02:06.0 '82545EM Gigabit Ethernet Controller (Copper)' drv=igb_uio unused=e
1000

Network devices using kernel driver
===================================
0000:02:01.0 '82545EM Gigabit Ethernet Controller (Copper)' if=eth1 drv=e1000 un
used=igb_uio *Active*

Other network devices
=====================
<none>
[root@bogon dpdk-1.7.0]#
```

图 1 网卡已经绑定好

进行绑定：

./tools/dpdk_nic_bind.py -b igb_uio 0000:02:06.0

./tools/dpdk_nic_bind.py -b igb_uio 0000:02:05.0

如果网卡有接口名，如 eth1, eth2，也可以在-b igb_uio 后面使用接口名， 而不使用 pci 地址。

5）设置环境变量：

export RTE_SDK=/home/lv/dpdk/dpdk-1.7.0

export RTE_TARGET=x86_64-native-linuxapp-gcc

之后进入<dpdk>/examples/l2，运行 make，成功会生成 build 目录，其中有编译好的 l2fwd 程序。

6)运行程序

./build/l2fwd -c f -n 2 -- -q 1 -p 0x3

2.3 功能分析：

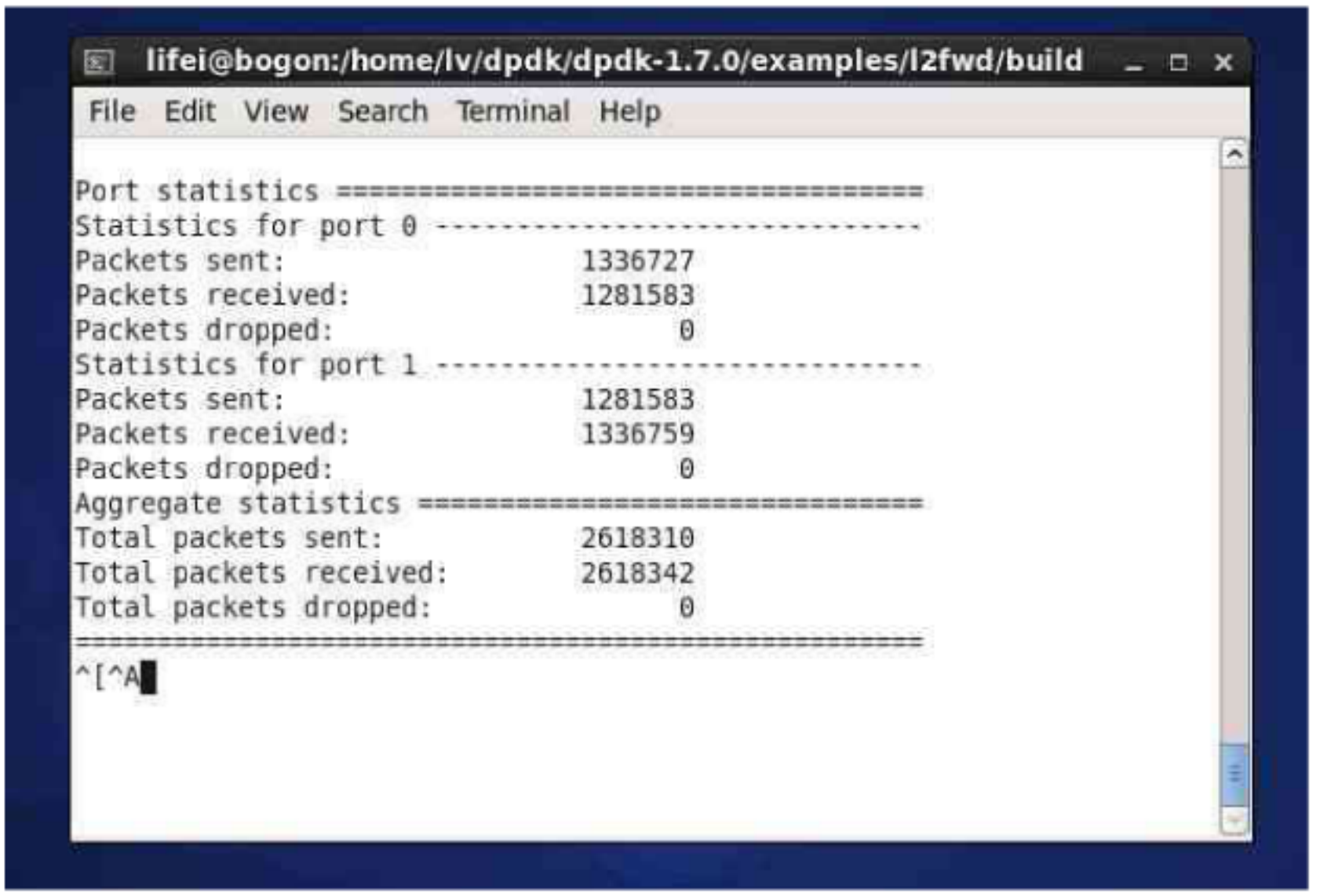
DPDK 搭建环境完成后，网卡绑定到相应 IGB_UIO 驱动接口上，所有的网络数据包都会到 DPDK，网卡接收网络数据包，再从另一个网卡转发出去

2.4 详细流程图（调用关系）如下：

(初学者，欢迎讨论 **QQ:780102849**，望各位指错**)**

DPDK L2fwd 详细流程图（调用关系）

2.5 运行截图



2.6 详细代码注释分析：


```
 1 #include <stdio.h>

 2 #include <stdlib.h>

 3 #include <string.h>

 4 #include <stdint.h>

 5 #include <inttypes.h>

 6 #include <sys/types.h>

 7 #include <sys/queue.h>

 8 #include <netinet/in.h>

 9 #include <setjmp.h>

10 #include <stdarg.h>

11 #include <ctype.h>
```

```
12 #include <errno.h>
13 #include <getopt.h>
14
15 #include <rte_common.h>
16 #include <rte_log.h>
17 #include <rte_memory.h>
18 #include <rte_memcpy.h>
19 #include <rte_memzone.h>
20 #include <rte_eal.h>
21 #include <rte_per_lcore.h>
22 #include <rte_launch.h>
23 #include <rte_atomic.h>
```

```
24 #include <rte_cycles.h>
25 #include <rte_prefetch.h>
26 #include <rte_lcore.h>
27 #include <rte_per_lcore.h>
28 #include <rte_branch_prediction.h>
29 #include <rte_interrupts.h>
30 #include <rte_pci.h>
31 #include <rte_random.h>
32 #include <rte_debug.h>
33 #include <rte_ether.h>
34 #include <rte_ethdev.h>
35 #include <rte_ring.h>
36 #include <rte_mempool.h>
37 #include <rte_mbuf.h>
38
39#define RTE_LOGTYPE_L2FWD RTE_LOGTYPE_USER1
40
```

```
41#define MBUF_SIZE (2048 + sizeof(struct rte_mbuf) + RTE_PKTMBUF_HEADROOM)
42#define NB_MBUF   8192
43
44#define MAX_PKT_BURST 32
45#define BURST_TX_DRAIN_US 100 /* TX drain every ~100us */
46
47/*
48 * Configurable number of RX/TX ring descriptors
49*/
50#define RTE_TEST_RX_DESC_DEFAULT 128
51#define RTE_TEST_TX_DESC_DEFAULT 512
52static uint16_t nb_rxd = RTE_TEST_RX_DESC_DEFAULT;
53static uint16_t nb_txd = RTE_TEST_TX_DESC_DEFAULT;
54
55/*物理端口的 mac 地址的数组       ethernet addresses of ports */
56staticstruct ether_addr l2fwd_ports_eth_addr[RTE_MAX_ETHPORTS];
57
58/*已经启用的物理端口的掩码/位图    mask of enabled ports */
59static uint32_t l2fwd_enabled_port_mask = 0;
60
61/*已经启用的目的物理端口编号的数组    list of enabled ports */
62static uint32_t l2fwd_dst_ports[RTE_MAX_ETHPORTS];
63
64static unsigned int l2fwd_rx_queue_per_lcore = 1; //默认值，每个 lcore 负责
的接收队列数量
65
66struct mbuf_table {  //mbuf 数组，可以存放 32 个数据包
67     unsigned len;
68struct rte_mbuf *m_table[MAX_PKT_BURST];
69 };
```

```
70
71#define MAX_RX_QUEUE_PER_LCORE 16
72#define MAX_TX_QUEUE_PER_PORT 16
73struct lcore_queue_conf {
74     unsigned n_rx_port;  //用于接收数据包的物理端口的实际数量
75     unsigned rx_port_list[MAX_RX_QUEUE_PER_LCORE];
76struct mbuf_table tx_mbufs[RTE_MAX_ETHPORTS]; //保存发送数据包的缓存区
77
78 } __rte_cache_aligned;
79struct lcore_queue_conf lcore_queue_conf[RTE_MAX_LCORE];
80
81staticconststruct rte_eth_conf port_conf = {
82     .rxmode = {
83          .split_hdr_size = 0,
84          .header_split   = 0, /**< Header Split disabled */
85          .hw_ip_checksum = 0, /**< IP checksum offload disabled */
86          .hw_vlan_filter = 0, /**< VLAN filtering disabled */
87          .jumbo_frame    = 0, /**< Jumbo Frame Support disabled */
88          .hw_strip_crc   = 0, /**< CRC stripped by hardware */
89     },
90     .txmode = {
91          .mq_mode = ETH_MQ_TX_NONE,
92     },
93 };
94
95struct rte_mempool * l2fwd_pktmbuf_pool = NULL;
96
97/*每个物理端口的统计结构体   Per-port statistics struct */
98struct l2fwd_port_statistics {
99     uint64_t tx;
```

```
100     uint64_t rx;
101     uint64_t dropped;
102 } __rte_cache_aligned;
103struct l2fwd_port_statistics port_statistics[RTE_MAX_ETHPORTS]; //数据包
的统计信息的全局数组
104
105/* A tsc-based timer responsible for triggering statistics printout */
106#define TIMER_MILLISECOND 2ULL /* around 1ms at 2 Ghz */
107#define MAX_TIMER_PERIOD 86400 /* 1 day max */
108static int64_t timer_period = 10 * TIMER_MILLISECOND * 1; /* default peri
od is 10 seconds */
109
110/* Print out statistics on packets dropped */
staticvoid//打印数据包丢失等统计信息
112 print_stats(void)
113 {
114     uint64_t total_packets_dropped, total_packets_tx, total_packets_rx;
115     unsigned portid;
116
117     total_packets_dropped = 0;
118     total_packets_tx = 0;
119     total_packets_rx = 0;
120
121constchar clr[] = { 27, '[', '2', 'J', '\0' };
122constchar topLeft[] = { 27, '[', '1', ';', '1', 'H','\0' };
123
124/* Clear screen and move to top left */
125     printf("%s%s", clr, topLeft);
126
127     printf("\nPort statistics ====================================");
```

```
128
129for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++) {
130/* skip disabled ports */
131if ((l2fwd_enabled_port_mask & (1<< portid)) == 0)
132continue;
133        printf("\nStatistics for port %u "
134"\nPackets sent: %24"PRIu64
135"\nPackets received: %20"PRIu64
136"\nPackets dropped: %21"PRIu64,
137              portid,
138              port_statistics[portid].tx,
139              port_statistics[portid].rx,
140              port_statistics[portid].dropped);
141
142        total_packets_dropped += port_statistics[portid].dropped;
143        total_packets_tx += port_statistics[portid].tx;
144        total_packets_rx += port_statistics[portid].rx;
145    }
146    printf("\nAggregate statistics ==============================="
147"\nTotal packets sent: %18"PRIu64
148"\nTotal packets received: %14"PRIu64
149"\nTotal packets dropped: %15"PRIu64,
150           total_packets_tx,
151           total_packets_rx,
152           total_packets_dropped);
153     printf("\n===========================================================\n");
154 }
155
156/* Send the burst of packets on an output interface */
157staticint//在一个输出接口上burst发送数据包
```

```
158 l2fwd_send_burst(struct lcore_queue_conf *qconf, unsigned n, uint8_t por
t)
159 {
160struct rte_mbuf **m_table;
161     unsigned ret;
162     unsigned queueid =0;
163
164     m_table = (struct rte_mbuf **)qconf->tx_mbufs[port].m_table;
165//burst 输出数据包
166     ret = rte_eth_tx_burst(port, (uint16_t) queueid, m_table, (uint16_t)
 n);
167     port_statistics[port].tx += ret; //记录发包数量
168if (unlikely(ret < n)) {
169         port_statistics[port].dropped += (n - ret); //记录丢包数量
170do {
171             rte_pktmbuf_free(m_table[ret]);
172         } while (++ret < n);
173     }
174
175return0;
176 }
177
178/* Enqueue packets for TX and prepare them to be sent */
179staticint//把数据包入队到发送缓冲区
180 l2fwd_send_packet(struct rte_mbuf *m, uint8_t port)
181 {
182     unsigned lcore_id, len;
183struct lcore_queue_conf *qconf;
184
185     lcore_id = rte_lcore_id(); //取得正在运行的 lcore 编号
```

```
186
187     qconf = &lcore_queue_conf[lcore_id];//取得 lcore_queue 的配置
188     len = qconf->tx_mbufs[port].len; //得到发包缓存区中数据包的个数
189     qconf->tx_mbufs[port].m_table[len] = m;//指向数据包
190     len++;
191
192/* enough pkts to be sent */
193if (unlikely(len == MAX_PKT_BURST)) { //如果累计到 32 个数据包
194         l2fwd_send_burst(qconf, MAX_PKT_BURST, port); //实际发送数据包
195         len = 0;
196     }
197
198     qconf->tx_mbufs[port].len = len;//更新发包缓存区中的数据包的个数
199return0;
200 }
201
202staticvoid
203 l2fwd_simple_forward(struct rte_mbuf *m, unsigned portid)
204 {
205//想要满足文生提出的需求，主要在这里修改 ip 层和 tcp 层的数据内容。
206
207struct ether_hdr *eth;
208void *tmp;
209     unsigned dst_port;
210
211     dst_port = l2fwd_dst_ports[portid];
212     eth = rte_pktmbuf_mtod(m, struct ether_hdr *);
213
214/* 02:00:00:00:00:xx  修改目的 mac 地址 */
215     tmp = &eth->d_addr.addr_bytes[0];
```

```
216     *((uint64_t *)tmp) = 0x002 + ((uint64_t)dst_port <<40);
217
218/* src addr  修改进入包的目的mac地址为转发包的源mac地址  */
219     ether_addr_copy(&l2fwd_ports_eth_addr[dst_port], &eth->s_addr);
220
221     l2fwd_send_packet(m, (uint8_t) dst_port); //在dst_port上发送数据包
 }
223
224/* main processing loop */
225staticvoid//线程的主处理循环
226 l2fwd_main_loop(void)
227 {
228struct rte_mbuf *pkts_burst[MAX_PKT_BURST];
229struct rte_mbuf *m;
230     unsigned lcore_id;
231     uint64_t prev_tsc, diff_tsc, cur_tsc, timer_tsc;
232     unsigned i, j, portid, nb_rx;
233struct lcore_queue_conf *qconf;
234const uint64_t drain_tsc = (rte_get_tsc_hz() + US_PER_S - 1) / US_PER_S *
 BURST_TX_DRAIN_US;
235
236     prev_tsc = 0;
237     timer_tsc = 0;
238
239     lcore_id = rte_lcore_id(); //获取当期lcore的编号
240     qconf = &lcore_queue_conf[lcore_id]; //读取此lcore上的配置信息
241
242if (qconf->n_rx_port == 0) {  //如果此lcore上的用于接收的物理端口数量为0
243         RTE_LOG(INFO, L2FWD, "lcore %u has nothing to do\n", lcore_id);
244return; //那么结束该线程
```

```
245     }
246
247     RTE_LOG(INFO, L2FWD, "entering main loop on lcore %u\n", lcore_id);
248
249for (i = 0; i < qconf->n_rx_port; i++) {  //遍历所有的用于接收数据包的物理端口
250
251         portid = qconf->rx_port_list[i];//一个lcore可能负责多个接收用的物理端口
252         RTE_LOG(INFO, L2FWD, " -- lcoreid=%u portid=%u\n", lcore_id,
253             portid);
254     }
255
256while (1) { //死循环
257
258         cur_tsc = rte_rdtsc();
259
260/*
261          * TX burst queue drain
262*/
263         diff_tsc = cur_tsc - prev_tsc;
264if (unlikely(diff_tsc > drain_tsc)) {
265
266for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++) {
267if (qconf->tx_mbufs[portid].len == 0)
268continue;
269                 l2fwd_send_burst(&lcore_queue_conf[lcore_id],
270                         qconf->tx_mbufs[portid].len,
271                         (uint8_t) portid);
272                 qconf->tx_mbufs[portid].len = 0;
```

```
273             }
274
275/* if timer is enabled */
276if (timer_period >0) { //如果定时器启动
277
278/* advance the timer */
279                 timer_tsc += diff_tsc;
280
281/* if timer has reached its timeout */
282if (unlikely(timer_tsc >= (uint64_t) timer_period)) {
283
284/* do this only on master core */
285if (lcore_id == rte_get_master_lcore()) {
286                         print_stats();  //十秒钟打印一次收包统计信息
287/* reset the timer */
288                         timer_tsc = 0;
289                     }
290                 }
291         }
292
293             prev_tsc = cur_tsc;
294         }
295
296/*
297         * Read packet from RX queues
298*/
299for (i = 0; i < qconf->n_rx_port; i++) {  //遍历所有的用于接收数据包的物理端口
300
301             portid = qconf->rx_port_list[i]; //第 i 个物理端口
```

```
302             nb_rx = rte_eth_rx_burst((uint8_t) portid, 0, //接收数据包，返
回实际个数
303                          pkts_burst, MAX_PKT_BURST);
304
305             port_statistics[portid].rx += nb_rx;  //记录物理端口上收包数量
306
307for (j = 0; j < nb_rx; j++) { //遍历实际接收到的所有的数据包
308                 m = pkts_burst[j];
309                 rte_prefetch0(rte_pktmbuf_mtod(m, void *)); //预取
310                 l2fwd_simple_forward(m, portid);//简单的二层转发数据包
311             }
312         }
313     }
314 }
315
316staticint
317 l2fwd_launch_one_lcore(__attribute__((unused)) void *dummy)
318 {
319     l2fwd_main_loop();//线程执行函数
320return0;
321 }
322
323/* display usage */
324staticvoid
325 l2fwd_usage(constchar *prgname)
326 {
327     printf("%s [EAL options] -- -p PORTMASK [-q NQ]\n"
328"  -p PORTMASK: hexadecimal bitmask of ports to configure\n"
329"  -q NQ: number of queue (=ports) per lcore (default is 1)\n"
```

```
330"  -T PERIOD: statistics will be refreshed each PERIOD seconds (0 to disa
ble, 10 default, 86400 maximum)\n",
331           prgname);
332 }


334staticint
335 l2fwd_parse_portmask(constchar *portmask)
336 {
337char *end = NULL;
338     unsigned long pm;
339/* 解析十六进制字符串 */
340/* parse hexadecimal string */
341     pm = strtoul(portmask, &end, 16);
342if ((portmask[0] == '\0')  (end == NULL)  (*end != '\0'))
343return -1;
344
345if (pm == 0)
346return -1;
347
348return pm;
349 }
350
351static unsigned int
352 l2fwd_parse_nqueue(constchar *q_arg)
353 {
354char *end = NULL;
355     unsigned long n;
356
357/* parse hexadecimal string */
358     n = strtoul(q_arg, &end, 10); //转换为十进制
```

```
359if ((q_arg[0] == '\0')  (end == NULL)  (*end != '\0'))
360return0;
361if (n == 0)
362return0;
363if (n >= MAX_RX_QUEUE_PER_LCORE)
364return0;
365
366return n;
367 }
368
369staticint
370 l2fwd_parse_timer_period(constchar *q_arg)
371 {
372char *end = NULL;
373int n;
374
375/* parse number string */
376     n = strtol(q_arg, &end, 10); //转换为十进制
377if ((q_arg[0] == '\0')  (end == NULL)  (*end != '\0'))
378return -1;
379if (n >= MAX_TIMER_PERIOD)
380return -1;
381
382return n;
383 }
384
385/*在应用程序的命令行中给出的参数解析 Parse the argument given in the command
line of the application */
386staticint
387 l2fwd_parse_args(int argc, char **argv)
```

```
388 {
389int opt, ret;
390char **argvopt;
391int option_index;
392char *prgname = argv[0];
393staticstruct option lgopts[] = {
394         {NULL, 0, 0, 0}
395     };
396
397     argvopt = argv;
398
399while ((opt = getopt_long(argc, argvopt, "p:q:T:",
400                   lgopts, &option_index)) != EOF) {
401
402switch (opt) {
403/* portmask */
404case'p': //物理端口的掩码
405             l2fwd_enabled_port_mask = l2fwd_parse_portmask(optarg);
406if (l2fwd_enabled_port_mask == 0) {
407                   printf("invalid portmask\n");
408                   l2fwd_usage(prgname);
409return -1;
410             }
411break;
412
413/* nqueue */
414case'q':  //lcore 负责的队列的数量
415             l2fwd_rx_queue_per_lcore = l2fwd_parse_nqueue(optarg);//修改
默认值
416if (l2fwd_rx_queue_per_lcore == 0) {
```

```
417                 printf("invalid queue number\n");
418                 l2fwd_usage(prgname);
419return -1;
420             }
421break;
422
423/* timer period */
424case'T': //定时的长度
425             timer_period = l2fwd_parse_timer_period(optarg) * 1 * TIMER_M
ILLISECOND;
426if (timer_period <0) {
427                 printf("invalid timer period\n");
428                 l2fwd_usage(prgname);
429return -1;
430             }
431break;
432
433/* long options */
434case0:
435             l2fwd_usage(prgname);
436return -1;
437
438default:
439             l2fwd_usage(prgname);
440return -1;
441         }
442     }
443
if (optind >= 0)
445         argv[optind-1] = prgname;
```

```
446
447     ret = optind-1;
448     optind = 0; /* reset getopt lib */
449return ret;
450 }
451
452/* Check the link status of all ports in up to 9s, and print them finally
 */
453staticvoid//检查物理端口的连接状态
454 check_all_ports_link_status(uint8_t port_num, uint32_t port_mask)
455 {
456#define CHECK_INTERVAL 100 /* 100ms */
457#define MAX_CHECK_TIME 90 /* 9s (90 * 100ms) in total */
458     uint8_t portid, count, all_ports_up, print_flag = 0;
459struct rte_eth_link link;
460
461     printf("\nChecking link status");
462     fflush(stdout);
463for (count = 0; count <= MAX_CHECK_TIME; count++) {
464         all_ports_up = 1;
465for (portid = 0; portid < port_num; portid++) {
466if ((port_mask & (1<< portid)) == 0)
467continue;
468             memset(&link, 0, sizeof(link));
469             rte_eth_link_get_nowait(portid, &link);
470/* print link status if flag set */
471if (print_flag == 1) {
472if (link.link_status)
473                     printf("Port %d Link Up - speed %u "
474"Mbps - %s\n", (uint8_t)portid,
```

```
475                         (unsigned)link.link_speed,
476                 (link.link_duplex == ETH_LINK_FULL_DUPLEX) ?
477                     ("full-duplex") : ("half-duplex\n"));
478else
479                     printf("Port %d Link Down\n",
480                         (uint8_t)portid);
481continue;
482             }
483/* clear all_ports_up flag if any link down */
484if (link.link_status == 0) {
485                 all_ports_up = 0;
486break;
487             }
488         }
489/* after finally printing all link status, get out */
490if (print_flag == 1)
491break;
492
493if (all_ports_up == 0) {
494             printf(".");
495             fflush(stdout);
496             rte_delay_ms(CHECK_INTERVAL);
497         }
498
499/* set the print_flag if all ports up or timeout */
500if (all_ports_up == 1  count == (MAX_CHECK_TIME - 1)) {
501             print_flag = 1;
502             printf("done\n");
503         }
504     }
```

```
505 }
506
507int//主函数
508 main(int argc, char **argv)
509 {
510struct lcore_queue_conf *qconf;
511struct rte_eth_dev_info dev_info;
512int ret;
513     uint8_t nb_ports;
514     uint8_t nb_ports_available;
515     uint8_t portid, last_port;
516     unsigned lcore_id, rx_lcore_id;
517     unsigned nb_ports_in_mask = 0;
518
519/* init EAL */
520     ret = rte_eal_init(argc, argv); //初始化环境抽象层，并解析相关参数
521if (ret <0)
522         rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");
523     argc -= ret;
524     argv += ret;
525
526/* parse application arguments (after the EAL ones) */
527     ret = l2fwd_parse_args(argc, argv); //解析 l2fwd 相关的参数: -p -q -P
528if (ret <0)
529         rte_exit(EXIT_FAILURE, "Invalid L2FWD arguments\n");
530
531/* create the mbuf pool */
532     l2fwd_pktmbuf_pool =       //创建 mbuf pool
533         rte_mempool_create("mbuf_pool", NB_MBUF,
534                    MBUF_SIZE, 32,
```

```
535sizeof(struct rte_pktmbuf_pool_private),
536                 rte_pktmbuf_pool_init, NULL,
537                 rte_pktmbuf_init, NULL,
538                 rte_socket_id(), 0);
539if (l2fwd_pktmbuf_pool == NULL)
540         rte_exit(EXIT_FAILURE, "Cannot init mbuf pool\n");
541
542     nb_ports = rte_eth_dev_count();  //得到物理端口的实际数量
543if (nb_ports == 0)
544         rte_exit(EXIT_FAILURE, "No Ethernet ports - bye\n");
545
546if (nb_ports > RTE_MAX_ETHPORTS) //如果物理端口的数量超过限制
547         nb_ports = RTE_MAX_ETHPORTS;
548
549/* 重置目的物理端口的数组  reset l2fwd_dst_ports */
550for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++)
551         l2fwd_dst_ports[portid] = 0;//清零
552     last_port = 0;
553
554/*  每个lcore用在一个专用的发送队列上
    * Each logical core is assigned a dedicated TX queue on each port.
556*/
557for (portid = 0; portid < nb_ports; portid++) {//遍历所有的物理端口
558/* 忽略未启用的物理端口 skip ports that are not enabled  */
559if ((l2fwd_enabled_port_mask & (1<< portid)) == 0)
560continue;
561
562if (nb_ports_in_mask % 2) { //如果是有偶数个物理端口，设为相邻两个物理端口对发
563             l2fwd_dst_ports[portid] = last_port; //奇数号的目的物理端口为偶
数号
```

```
564             l2fwd_dst_ports[last_port] = portid; //偶数号的目的物理端口为奇
数号
565         }
566else//如果是奇数个物理端口
567             last_port = portid;
568
569         nb_ports_in_mask++;  //更新已启用的物理端口的总数
570
571         rte_eth_dev_info_get(portid, &dev_info);
572     }
573if (nb_ports_in_mask % 2) { //如果已启用的物理端口的总数是奇数
574         printf("Notice: odd number of ports in portmask.\n");
575         l2fwd_dst_ports[last_port] = last_port;//last_port 的目的物理端口
还是 last_port
576     }
577
578     rx_lcore_id = 0;
579     qconf = NULL;
580
581/* Initialize the port/queue configuration of each logical core */
582for (portid = 0; portid < nb_ports; portid++) { //遍历所有的物理端口
583/* 忽略未启用的物理端口 skip ports that are not enabled */
584if ((l2fwd_enabled_port_mask & (1<< portid)) == 0)
585continue;
586
587/* 得到此物理端口的 lcore 编号  get the lcore_id for this port */
588while (rte_lcore_is_enabled(rx_lcore_id) == 0//如果此 lcore 未启用
589                lcore_queue_conf[rx_lcore_id].n_rx_port ==  //如果 lcore 上
负责接收的物理端口的实际数量等于
590                l2fwd_rx_queue_per_lcore) {//每个 lcore 负责的接收队列的实际
数量(-q 参数值)
```

```
591             rx_lcore_id++;//接收 lcore 的编号自增
592if (rx_lcore_id >= RTE_MAX_LCORE) //如果接收 lcore 编号超过 lcore 最大数量
593                 rte_exit(EXIT_FAILURE, "Not enough cores\n");
594         }
595
596if (qconf != &lcore_queue_conf[rx_lcore_id])
597/* Assigned a new logical core in the loop above. */
598             qconf = &lcore_queue_conf[rx_lcore_id];
599
600         qconf->rx_port_list[qconf->n_rx_port] = portid;
601         qconf->n_rx_port++;//用于接收数据包的物理端口数量自增
602         printf("Lcore %u: RX port %u\n", rx_lcore_id, (unsigned) porti
d);
603     }
604
605     nb_ports_available = nb_ports;
606
607/*初始化每个物理端口  Initialise each port */
608for (portid = 0; portid < nb_ports; portid++) { //遍历所有的物理端口
609/* 忽略未使能的物理端口   skip ports that are not enabled */
610if ((l2fwd_enabled_port_mask & (1<< portid)) == 0) {
611             printf("Skipping disabled port %u\n", (unsigned) portid);
612             nb_ports_available--;
613continue;
614         }
615/* 初始化某个物理端口 init port */
616         printf("Initializing port %u... ", (unsigned) portid);
617         fflush(stdout);
618         ret = rte_eth_dev_configure(portid, 1, 1, &port_conf); //第一步，
设为 1 个发送队列和 1 个接收队列
```

```
619if (ret <0)
620            rte_exit(EXIT_FAILURE, "Cannot configure device: err=%d, port
=%u\n",
621                  ret, (unsigned) portid);
622
623        rte_eth_macaddr_get(portid,&l2fwd_ports_eth_addr[portid]);//获取
mac 地址
624
625/* 在每个物理端口上建立一个接收队列  init one RX queue */
626        fflush(stdout);
627        ret = rte_eth_rx_queue_setup(portid, 0, nb_rxd, //第二步，0 代表接
收队列的编号
628                        rte_eth_dev_socket_id(portid),
629                        NULL,
630                        l2fwd_pktmbuf_pool);
631if (ret <0)
632            rte_exit(EXIT_FAILURE, "rte_eth_rx_queue_setup:err=%d, port
=%u\n",
633                  ret, (unsigned) portid);
634
635/* 在每个物理端口上建立一个发送队列    init one TX queue on each port */
636        fflush(stdout);
637        ret = rte_eth_tx_queue_setup(portid, 0, nb_txd,//第三步，0代表发送
队列的编号
638                  rte_eth_dev_socket_id(portid),
639                  NULL);
640if (ret <0)
641            rte_exit(EXIT_FAILURE, "rte_eth_tx_queue_setup:err=%d, port
=%u\n",
642                  ret, (unsigned) portid);
643
644/*启动设备   Start device */
```

```
645         ret = rte_eth_dev_start(portid); //第四步，启动物理端口
646if (ret <0)
647             rte_exit(EXIT_FAILURE, "rte_eth_dev_start:err=%d, port=%u\n
",
648                     ret, (unsigned) portid);
649
650         printf("done: \n");
651
652         rte_eth_promiscuous_enable(portid);
653
654         printf("Port %u, MAC address: %02X:%02X:%02X:%02X:%02X:%02X\n\n
",
655                 (unsigned) portid,
656                 l2fwd_ports_eth_addr[portid].addr_bytes[0],
657                 l2fwd_ports_eth_addr[portid].addr_bytes[1],
658                 l2fwd_ports_eth_addr[portid].addr_bytes[2],
659                 l2fwd_ports_eth_addr[portid].addr_bytes[3],
660                 l2fwd_ports_eth_addr[portid].addr_bytes[4],
661                 l2fwd_ports_eth_addr[portid].addr_bytes[5]);
662
663/*清空物理端口的统计信息  initialize port stats */
664         memset(&port_statistics, 0, sizeof(port_statistics));
665     }


667if (!nb_ports_available) {
668         rte_exit(EXIT_FAILURE,
669"All available ports are disabled. Please set portmask.\n");
670     }
671
672     check_all_ports_link_status(nb_ports, l2fwd_enabled_port_mask);
```

```
673
674/* 发送消息给每个 lcore，让 Lcore 启动线程开始工作*/
675     rte_eal_mp_remote_launch(l2fwd_launch_one_lcore, NULL, CALL_MASTER);
676     RTE_LCORE_FOREACH_SLAVE(lcore_id) {
677if (rte_eal_wait_lcore(lcore_id) <0)  //等待线程完成工作
678return -1;
679     }
680
681return0;
682 }
```

代码参考 http://www.lxway.com/2506686.htm