https://www.cnblogs.com/ZCplayground/p/9350354.html

代码部分

main.c

/\*-

\* BSD LICENSE

\*

\* Copyright 2017 Mellanox.

\*

\* Redistribution and use in source and binary forms, with or without

\* modification, are permitted provided that the following conditions

\* are met:

\*

\* \* Redistributions of source code must retain the above copyright

\* notice, this list of conditions and the following disclaimer.

\* \* Redistributions in binary form must reproduce the above copyright

\* notice, this list of conditions and the following disclaimer in

\* the documentation and/or other materials provided with the

\* distribution.

\* \* Neither the name of Mellanox. nor the names of its

\* contributors may be used to endorse or promote products derived

\* from this software without specific prior written permission.

\*

\* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS

\* "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT

\* LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR

\* A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT

\* OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,

\* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT

\* LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,

\* DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY

\* THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT

\* (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE

\* OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdint.h>

#include <inttypes.h>

#include <sys/types.h>

#include <sys/queue.h>

#include <netinet/in.h>

#include <setjmp.h>

#include <stdarg.h>

#include <ctype.h>

#include <errno.h>

#include <getopt.h>

#include <signal.h>

#include <stdbool.h>

#include <rte\_eal.h>

#include <rte\_common.h>

#include <rte\_malloc.h>

#include <rte\_ether.h>

#include <rte\_ethdev.h>

#include <rte\_mempool.h>

#include <rte\_mbuf.h>

#include <rte\_net.h>

#include <rte\_flow.h>

#include <rte\_cycles.h>

static volatile bool force\_quit; // 确保本条指令不会因编译器的优化而省略

// 用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。

static uint16\_t port\_id;

static uint16\_t nr\_queues = 5; // 收发队列各5条

static uint8\_t selected\_queue = 1;

struct rte\_mempool \*mbuf\_pool;

struct rte\_flow \*flow;

#define SRC\_IP ((0<<24) + (0<<16) + (0<<8) + 0) /\* src ip = 0.0.0.0 \*/

#define DEST\_IP ((192<<24) + (168<<16) + (1<<8) + 1) /\* dest ip = 192.168.1.1 \*/

#define FULL\_MASK 0xffffffff /\* full mask \*/

#define EMPTY\_MASK 0x0 /\* empty mask \*/

#include "flow\_blocks.c" // generate\_ipv4\_flow

static inline void

print\_ether\_addr(const char \*what, struct ether\_addr \*eth\_addr)

{

char buf[ETHER\_ADDR\_FMT\_SIZE];

ether\_format\_addr(buf, ETHER\_ADDR\_FMT\_SIZE, eth\_addr); // Format 48bits Ethernet address in pattern xx:xx:xx:xx:xx:xx.

printf("%s%s", what, buf);

}

static void

main\_loop(void)

{

struct rte\_mbuf \*mbufs[32];

struct ether\_hdr \*eth\_hdr;

struct rte\_flow\_error error;

uint16\_t nb\_rx;

uint16\_t i;

uint16\_t j;

while (!force\_quit) {

for (i = 0; i < nr\_queues; i++) {

nb\_rx = rte\_eth\_rx\_burst(port\_id,

i, mbufs, 32); // 收包

if (nb\_rx) { // 该应用程序的主要工作是从所有队列读取数据包并打印目标队列的每个数据包

for (j = 0; j < nb\_rx; j++) {

struct rte\_mbuf \*m = mbufs[j];

eth\_hdr = rte\_pktmbuf\_mtod(m,

struct ether\_hdr \*);

/\*rte\_pktmbuf\_mtod(m,t) 是一个宏，m 是 mbuf 指针

返回一个被强制转换成 t \* 的指针。指向 mbuf 里的数据的开始处。

\*/

print\_ether\_addr("src=",

&eth\_hdr->s\_addr); // 见文件 rte\_ether.h

print\_ether\_addr(" - dst=",

&eth\_hdr->d\_addr);

printf(" - queue=0x%x",

(unsigned int)i);

printf("\n");

rte\_pktmbuf\_free(m); // 收的包就直接free掉

}

}

}

}

/\* closing and releasing resources \*/

rte\_flow\_flush(port\_id, &error); // Destroy all flow rules associated with a port.

rte\_eth\_dev\_stop(port\_id); // Stop an Ethernet device. 是函数 rte\_eth\_dev\_start()的反义词

rte\_eth\_dev\_close(port\_id); // Close a stopped Ethernet device. The device cannot be restarted

}

#define CHECK\_INTERVAL 1000 /\* 100ms \*/

#define MAX\_REPEAT\_TIMES 90 /\* 9s (90 \* 100ms) in total \*/

static void

assert\_link\_status(void)

{

struct rte\_eth\_link link; // 用于检索以太网端口的链路级别信息的结构体

uint8\_t rep\_cnt = MAX\_REPEAT\_TIMES;

memset(&link, 0, sizeof(link));

do {

rte\_eth\_link\_get(port\_id, &link); // 获取链路的状态，可以获得如下信息：

/\*

1. 开启或关闭（ON、OFF）

2. 链路速度（单位 Mbps）

3. 通信模式（半双工 or 全双工）

两个参数，port id 和 rte\_eth\_link 的结构体指针

这个函数可能会阻塞 9 秒钟。

\*/

if (link.link\_status == ETH\_LINK\_UP) // 链路开启

break;

rte\_delay\_ms(CHECK\_INTERVAL); // 等待至少N毫秒，位于rte\_cycles.h

} while (--rep\_cnt);

if (link.link\_status == ETH\_LINK\_DOWN) // 链路关闭

rte\_exit(EXIT\_FAILURE, ":: error: link is still down\n");

}

static void

init\_port(void)

{

int ret;

uint16\_t i;

struct rte\_eth\_conf port\_conf = { // 用于配置以太网口的结构体

.rxmode = { // 端口 rx 配置

.split\_hdr\_size = 0, // hdr buf size （todo）

.ignore\_offload\_bitfield = 1, //（todo）

.offloads = DEV\_RX\_OFFLOAD\_CRC\_STRIP, // 不进行 CRC

},

.txmode = { // 端口 tx 配置

.offloads =

DEV\_TX\_OFFLOAD\_VLAN\_INSERT |

DEV\_TX\_OFFLOAD\_IPV4\_CKSUM |

DEV\_TX\_OFFLOAD\_UDP\_CKSUM |

DEV\_TX\_OFFLOAD\_TCP\_CKSUM |

DEV\_TX\_OFFLOAD\_SCTP\_CKSUM |

DEV\_TX\_OFFLOAD\_TCP\_TSO,

},

};

struct rte\_eth\_txconf txq\_conf;

struct rte\_eth\_rxconf rxq\_conf;

struct rte\_eth\_dev\_info dev\_info;

printf(":: initializing port: %d\n", port\_id);

ret = rte\_eth\_dev\_configure(port\_id, // 配置网口

nr\_queues, nr\_queues, &port\_conf); // 收发队列各5条

if (ret < 0) {

rte\_exit(EXIT\_FAILURE,

":: cannot configure device: err=%d, port=%u\n",

ret, port\_id);

}

rte\_eth\_dev\_info\_get(port\_id, &dev\_info); // 查询以太网设备信息

rxq\_conf = dev\_info.default\_rxconf; // Default RX configuration，类型：struct rte\_eth\_rxconf

rxq\_conf.offloads = port\_conf.rxmode.offloads;

/\* only set Rx queues: something we care only so far \*/

for (i = 0; i < nr\_queues; i++) { // 设置 rx queues

ret = rte\_eth\_rx\_queue\_setup(port\_id, i, 512,

rte\_eth\_dev\_socket\_id(port\_id),

&rxq\_conf, // rx queue的配置数据，类型是 const struct rte\_eth\_rxconf \* 指针

mbuf\_pool);

if (ret < 0) {

rte\_exit(EXIT\_FAILURE,

":: Rx queue setup failed: err=%d, port=%u\n",

ret, port\_id);

}

}

txq\_conf = dev\_info.default\_txconf;

txq\_conf.offloads = port\_conf.txmode.offloads;

for (i = 0; i < nr\_queues; i++) { // 设置 tx queues

ret = rte\_eth\_tx\_queue\_setup(port\_id, i, 512,

rte\_eth\_dev\_socket\_id(port\_id),

&txq\_conf);// tx queue的配置数据，类型是 const struct rte\_eth\_txconf \* 指针

if (ret < 0) {

rte\_exit(EXIT\_FAILURE,

":: Tx queue setup failed: err=%d, port=%u\n",

ret, port\_id);

}

}

rte\_eth\_promiscuous\_enable(port\_id); // 启用混杂模式

ret = rte\_eth\_dev\_start(port\_id); // 启动设备

if (ret < 0) {

rte\_exit(EXIT\_FAILURE,

"rte\_eth\_dev\_start:err=%d, port=%u\n",

ret, port\_id);

}

assert\_link\_status();

printf(":: initializing port: %d done\n", port\_id);

}

static void

signal\_handler(int signum)

{

if (signum == SIGINT || signum == SIGTERM) {

printf("\n\nSignal %d received, preparing to exit...\n",

signum);

force\_quit = true;

}

}

int

main(int argc, char \*\*argv)

{

int ret;

uint8\_t nr\_ports;

struct rte\_flow\_error error;

ret = rte\_eal\_init(argc, argv); // 初始化 EAL

if (ret < 0)

rte\_exit(EXIT\_FAILURE, ":: invalid EAL arguments\n");

/\* signal(sig, handler) 设置某一信号的对应动作

第一个参数signum指明了所要处理的信号类型，它可以取除了SIGKILL和SIGSTOP外的任何一种信号。

第二个参数handler描述了与信号关联的动作，它可以取以下三种值：

1.一个无返回值的函数地址

2. SIG\_IGN ：忽略

3. SIG\_DFL ：恢复默认处理

\*/

force\_quit = false;

signal(SIGINT, signal\_handler); // 中断信号

signal(SIGTERM, signal\_handler); // 程序结束(terminate)信号

nr\_ports = rte\_eth\_dev\_count();

if (nr\_ports == 0)

rte\_exit(EXIT\_FAILURE, ":: no Ethernet ports found\n");

port\_id = 0;

if (nr\_ports != 1) { // 本程序只需要使用 1 个以太网设备

printf(":: warn: %d ports detected, but we use only one: port %u\n",

nr\_ports, port\_id);

}

mbuf\_pool = rte\_pktmbuf\_pool\_create("mbuf\_pool", 4096, 128, 0,

RTE\_MBUF\_DEFAULT\_BUF\_SIZE,

rte\_socket\_id()); // 创建 mempool

if (mbuf\_pool == NULL)

rte\_exit(EXIT\_FAILURE, "Cannot init mbuf pool\n");

init\_port(); // 端口初始化

/\* create flow for send packet with \*/

// 参数：port id、queue index、IP四元组、error

// flow 是 struct rte\_flow

flow = generate\_ipv4\_flow(port\_id, selected\_queue, // 将目的地ip等于192.168.1.1的数据包发送到队列号1

SRC\_IP, EMPTY\_MASK,

DEST\_IP, FULL\_MASK, &error);

// 虽然配置了5条队列，但会把特定IP地址的流量放到一条特定的队列。

// 是对 rte\_flow\_create() 这个函数的一层封装

if (!flow) {

printf("Flow can't be created %d message: %s\n",

error.type,

error.message ? error.message : "(no stated reason)");

rte\_exit(EXIT\_FAILURE, "error in creating flow");

}

main\_loop();

return 0;

}

flow\_blocks.c：

/\*-

\* BSD LICENSE

\*

\* Copyright 2017 Mellanox.

\*

\* Redistribution and use in source and binary forms, with or without

\* modification, are permitted provided that the following conditions

\* are met:

\*

\* \* Redistributions of source code must retain the above copyright

\* notice, this list of conditions and the following disclaimer.

\* \* Redistributions in binary form must reproduce the above copyright

\* notice, this list of conditions and the following disclaimer in

\* the documentation and/or other materials provided with the

\* distribution.

\* \* Neither the name of Mellanox nor the names of its

\* contributors may be used to endorse or promote products derived

\* from this software without specific prior written permission.

\*

\* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS

\* "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT

\* LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR

\* A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT

\* OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,

\* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT

\* LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,

\* DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY

\* THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT

\* (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE

\* OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

\*/

#define MAX\_PATTERN\_NUM 4

struct rte\_flow \*

generate\_ipv4\_flow(uint16\_t port\_id, uint16\_t rx\_q,

uint32\_t src\_ip, uint32\_t src\_mask,

uint32\_t dest\_ip, uint32\_t dest\_mask,

struct rte\_flow\_error \*error);

/\*\*

\* create a flow rule that sends packets with matching src and dest ip

\* to selected queue.

\*

\* @param port\_id

\* The selected port.

\* @param rx\_q

\* The selected target queue.

\* @param src\_ip

\* The src ip value to match the input packet.

\* @param src\_mask

\* The mask to apply to the src ip.

\* @param dest\_ip

\* The dest ip value to match the input packet.

\* @param dest\_mask

\* The mask to apply to the dest ip.

\* @param[out] error

\* Perform verbose error reporting if not NULL.

\*

\* @return

\* A flow if the rule could be created else return NULL.

\*/

struct rte\_flow \*

generate\_ipv4\_flow(uint16\_t port\_id, uint16\_t rx\_q,

uint32\_t src\_ip, uint32\_t src\_mask,

uint32\_t dest\_ip, uint32\_t dest\_mask,

struct rte\_flow\_error \*error)

{

struct rte\_flow\_attr attr; // 流的 attr

struct rte\_flow\_item pattern[MAX\_PATTERN\_NUM]; // 流的 pattern。关于 item，见：http://doc.dpdk.org/api/structrte\_\_flow\_\_item.html

struct rte\_flow\_action action[MAX\_PATTERN\_NUM]; // 流的 action ， 这三个是创建一个流的关键。

struct rte\_flow \*flow = NULL;

struct rte\_flow\_action\_queue queue = { .index = rx\_q };

struct rte\_flow\_item\_eth eth\_spec; // spec 和

struct rte\_flow\_item\_eth eth\_mask; // mask 是 item 的另外两个字段。void \* ，但需要设置成和你选定的特定 type 一样。

struct rte\_flow\_item\_vlan vlan\_spec;

struct rte\_flow\_item\_vlan vlan\_mask;

struct rte\_flow\_item\_ipv4 ip\_spec;

struct rte\_flow\_item\_ipv4 ip\_mask;

int res;

memset(pattern, 0, sizeof(pattern));

memset(action, 0, sizeof(action));

/\*

\* set the rule attribute.

\* in this case only ingress packets will be checked.

\*/

memset(&attr, 0, sizeof(struct rte\_flow\_attr));

attr.ingress = 1; // 意思是只对入口流量生效的属性

/\*

\* create the action sequence.

\* one action only, move packet to queue

\*/

action[0].type = RTE\_FLOW\_ACTION\_TYPE\_QUEUE; // 动作是：Assigns packets to a given queue index.

action[0].conf = &queue;

action[1].type = RTE\_FLOW\_ACTION\_TYPE\_END; // 动作数组必须用 RTE\_FLOW\_ACTION\_TYPE\_END 作为最后一个元素来结尾

/\*

\* set the first level of the pattern (eth).

\* since in this example we just want to get the

\* ipv4 we set this level to allow all.

\*/

// 第一个 item 用的是以太网地址

// rte\_flow\_item\_eth 的字段是 目的MAC地址、源MAC地址、Type

memset(&eth\_spec, 0, sizeof(struct rte\_flow\_item\_eth));

memset(&eth\_mask, 0, sizeof(struct rte\_flow\_item\_eth));

eth\_spec.type = 0;

eth\_mask.type = 0;

pattern[0].type = RTE\_FLOW\_ITEM\_TYPE\_ETH; // item type 是 以太网的 item

pattern[0].spec = &eth\_spec; // 指向 eth\_item 的指针（因为第一个type指定了rte\_flow\_item\_eth）

pattern[0].mask = &eth\_mask; // 掩码设置成全 0，意味着所有以太网的header都是被允许的。

/\* spec、mask、last 在 pattern item 中的用途：

For example, if for an IPv4 address field, spec provides 10.1.2.3, last provides 10.3.4.5

and mask provides 255.255.0.0, the effective range becomes 10.1.0.0 to 10.3.255.255.

\*/

/\*

\* setting the second level of the pattern (vlan).

\* since in this example we just want to get the

\* ipv4 we also set this level to allow all.

\*/

// 第二个 item 用于匹配 vlan 标签

// 也设置成了所有皆可匹配

memset(&vlan\_spec, 0, sizeof(struct rte\_flow\_item\_vlan));

memset(&vlan\_mask, 0, sizeof(struct rte\_flow\_item\_vlan));

pattern[1].type = RTE\_FLOW\_ITEM\_TYPE\_VLAN;

pattern[1].spec = &vlan\_spec;

pattern[1].mask = &vlan\_mask;

/\*

\* setting the third level of the pattern (ip).

\* in this example this is the level we care about

\* so we set it according to the parameters.

\*/

// 第三个 item 匹配特定的IP地址，根据本函数的参数。

memset(&ip\_spec, 0, sizeof(struct rte\_flow\_item\_ipv4));

memset(&ip\_mask, 0, sizeof(struct rte\_flow\_item\_ipv4));

// ipv4 item 的结构体里面有一个字段，是 ipv4\_hdr

ip\_spec.hdr.dst\_addr = htonl(dest\_ip); // 将主机数转换成无符号长整型的网络字节顺序

ip\_mask.hdr.dst\_addr = dest\_mask;

ip\_spec.hdr.src\_addr = htonl(src\_ip);

ip\_mask.hdr.src\_addr = src\_mask;

pattern[2].type = RTE\_FLOW\_ITEM\_TYPE\_IPV4;

pattern[2].spec = &ip\_spec;

pattern[2].mask = &ip\_mask;

/\* the final level must be always type end \*/

// pattern 数组的最后必须用 END 宏结尾。

pattern[3].type = RTE\_FLOW\_ITEM\_TYPE\_END;

// 验证这条流的有效性

res = rte\_flow\_validate(port\_id, &attr, pattern, action, error);

if (!res)

flow = rte\_flow\_create(port\_id, &attr, pattern, action, error);

/\* rte\_flow\_create() 在一个给定的端口上创建一条流规则（flow rule）

参数五个

1. port id

2. attr 数组

3. pattern 数组

4. actions 数组

5. rte\_flow\_error 指针，当有错误发生时，PMD会在此设置内容。

返回值类型是 struct rte\_flow\*

\*/

return flow;

}

classify（分类） 与 filtering（过滤）的区别：

flow\_classify 的思路是创建 classifer 对象，在 classifer 里配置一些 rule，在收到一批包后，主动地调用 query classifer 的 API，查看是否有 packet 符合了 classifer 里面的 rule，并执行对应的 action。

flow\_filtering 的思路则是在 port 上设定 rule。对于满足 rule 的包，会采取特定的 actions。与 classify 的区别在于，filtering 是在端口上面设定 rule，只要经过这个端口的 ingress/egress 流量，都要进行 rule 的匹配。

可以看到，classify 和 filtering 共同的地方就是 rule。差别在于使用rule的方式。DPDK中，classify的库叫做rte\_flow\_classify.h，filtering的库叫做rte\_flow.h，所以说它们之间还是有比较明显的区别。

rule（规则）的代码实现：体现为结构体 rte\_flow\_item\_xxx ，xxx 有 attr、item 和 action

attr 是这条 rule 的属性，主要用到的有 ingress 和 egress，指明这条 rule 是针对入口还是出口流量生效。

item 就是规则的本体，用来匹配包里的特定的 header 的，代码中是通过堆叠 item 的方式来实现特定的匹配模式（pattern）。

action 就是匹配成功后的动作，通过设定宏来定义操作。宏是库文件自带的。常见的例如计数、丢包、转发到特定队列等。

这个 sample flow\_filtering 里实现了这样的行为：设立5条收/发队列，收包时将目的IP地址等于 192.168.1.1 的数据包发送到队列号为1的发送队列。所以本 sample 实现的其实是一种类似 QoS 的功能（特定IP地址进入特定的队列处理）。如果真的想实现 filtering（过滤），就把 actions 定义为 Drop 吧。然后，逻辑核上的主线程运行main\_loop，主要工作收包后，打印所有包的发送队列号。由于 rule 的作用，应该可以看到目的地 ip 等于192.168.1.1 的数据包的发送队列号一定为 1。

执行情况

开始执行后，收到的包都到了0号队列（因为目的IP地址不是192.168.1.1）。然后使用scapy发目的IP地址设定为192.168.1.1的包到0号以太网口，发了十个包，可以看到十个包的转发队列为1号队列。

发包脚本如下：

from scapy.all import \*

p\_join = Ether(dst='a0:36:9f:20:43:92', src='a0:36:9f:20:43:90') / IP(src='172.17.173.64', dst='192.168.1.1', tos=0xc0)

p\_join.show()

i = 0

while i < 10 :

# enp1s0f0 -->

# bind enpis0f1 to dpdk

sendp(p\_join,iface='enp1s0f0')

i = i + 1

注意事项：

本程序只需要绑定一个网口到DPDK即可。

网卡必须支持多队列，且队列数目要5条以上。不然会报错，信息类似于：cannot configure device

reference

API doc 中查看 mbuf、rte flow 和 flow classify

sample guide：flow\_filtering

linux查看网卡型号、驱动版本、队列数

分类: DPDK