**https://www.cnblogs.com/ZCplayground/p/9330696.html**

**代码部分**

/\* SPDX-License-Identifier: BSD-3-Clause

\* Copyright(c) 2017 Intel Corporation

\*/

#include <stdint.h>

#include <inttypes.h>

#include <getopt.h>

#include <rte\_eal.h>

#include <rte\_ethdev.h>

#include <rte\_cycles.h>

#include <rte\_lcore.h>

#include <rte\_mbuf.h>

#include <rte\_flow.h>

#include <rte\_flow\_classify.h>

#include <rte\_table\_acl.h>

#define RX\_RING\_SIZE 1024

#define TX\_RING\_SIZE 1024

#define NUM\_MBUFS 8191

#define MBUF\_CACHE\_SIZE 250

#define BURST\_SIZE 32

#define MAX\_NUM\_CLASSIFY 30

#define FLOW\_CLASSIFY\_MAX\_RULE\_NUM 91

#define FLOW\_CLASSIFY\_MAX\_PRIORITY 8

#define FLOW\_CLASSIFIER\_NAME\_SIZE 64

#define COMMENT\_LEAD\_CHAR ('#')

#define OPTION\_RULE\_IPV4 "rule\_ipv4"

#define RTE\_LOGTYPE\_FLOW\_CLASSIFY RTE\_LOGTYPE\_USER3

#define flow\_classify\_log(format, ...) \

RTE\_LOG(ERR, FLOW\_CLASSIFY, format, ##\_\_VA\_ARGS\_\_)

#define uint32\_t\_to\_char(ip, a, b, c, d) do {\

\*a = (unsigned char)(ip >> 24 & 0xff);\

\*b = (unsigned char)(ip >> 16 & 0xff);\

\*c = (unsigned char)(ip >> 8 & 0xff);\

\*d = (unsigned char)(ip & 0xff);\

} while (0)

enum {

CB\_FLD\_SRC\_ADDR, // 0

CB\_FLD\_DST\_ADDR, // 1

CB\_FLD\_SRC\_PORT, // 2

CB\_FLD\_SRC\_PORT\_DLM, // 3

CB\_FLD\_SRC\_PORT\_MASK,// 4

CB\_FLD\_DST\_PORT, // 5

CB\_FLD\_DST\_PORT\_DLM, // 6

CB\_FLD\_DST\_PORT\_MASK,// 7

CB\_FLD\_PROTO, // 8

CB\_FLD\_PRIORITY, // 9

CB\_FLD\_NUM, // 10

};

static struct{

const char \*rule\_ipv4\_name;

} parm\_config; // 用于文件访问的。

const char cb\_port\_delim[] = ":";

static const struct rte\_eth\_conf port\_conf\_default = {

.rxmode = {

.max\_rx\_pkt\_len = ETHER\_MAX\_LEN,

.ignore\_offload\_bitfield = 1,

},

};

struct flow\_classifier {

struct rte\_flow\_classifier \*cls;

};

// flow\_classifer 的结构要看 sample guide

/\*

struct rte\_flow\_classifier {

// classifier的参数，要 create() 时传入结构体。

char name[RTE\_FLOW\_CLASSIFIER\_MAX\_NAME\_SZ];

int socket\_id;

// 其余的内部字段

// n tuple 过滤器，也就是流规则的匹配项目了。

struct rte\_eth\_ntuple\_filter ntuple\_filter;

// tables

struct rte\_cls\_table tables[RTE\_FLOW\_CLASSIFY\_TABLE\_MAX];

uint32\_t table\_mask;

uint32\_t num\_tables;

uint16\_t nb\_pkts;

struct rte\_flow\_classify\_table\_entry

\*entries[RTE\_PORT\_IN\_BURST\_SIZE\_MAX];

} \_\_rte\_cache\_aligned;

\*/

struct flow\_classifier\_acl {

struct flow\_classifier cls;

} \_\_rte\_cache\_aligned;

/\* ACL field definitions for IPv4 5 tuple rule \*/

enum {

PROTO\_FIELD\_IPV4, // 0

SRC\_FIELD\_IPV4, // 1

DST\_FIELD\_IPV4, // 2

SRCP\_FIELD\_IPV4, // 3

DSTP\_FIELD\_IPV4, // 4

NUM\_FIELDS\_IPV4 // 5

};

enum {

PROTO\_INPUT\_IPV4,

SRC\_INPUT\_IPV4,

DST\_INPUT\_IPV4,

SRCP\_DESTP\_INPUT\_IPV4

};

/\* 数据结构 rte\_acl\_field\_def：ACL 访问控制表的字段的定义

ACL规则中的每个字段都有一个关联定义。有五个，分别是：

字段的类型 type，

字段的字节数大小 size，

字段的索引（指示哪一个字段）field\_index 一个0开始的值，用来指定字段在规则内部的位置，0~n-1表示n个字段。

输入索引 input\_index（0-N） 所有输入字段，除了第一个，其他必须以4个连续字节分组，这个input\_index就是来指定字段在那个组

偏移量offset 定义了字段的偏移量，为查找指定从缓冲区的起始位置的偏移。

\*/

/\*

rule “规则” 有一些独有规则：

1. 规则定义的第一个字段必须是一个字节的长度

2. 之后的字段必须以4个连续的字节分组

这主要是为性能考虑，查找函数处理第一个输入字节做为这个流的设置的一部分，然后这查找函数的内部循环被展开来同时处理4字节的输入。

\*/

static struct rte\_acl\_field\_def ipv4\_defs[NUM\_FIELDS\_IPV4] = { // 共 5 个字段，每个字段都要有一个关联的五个定义

/\* first input field - always one byte long. \*/ // 第一个字段 1个字节

{

.type = RTE\_ACL\_FIELD\_TYPE\_BITMASK, // type 字段的类型，有3种选项，见https://www.cnblogs.com/danxi/p/6650757.html

.size = sizeof(uint8\_t), // 1个字节

.field\_index = PROTO\_FIELD\_IPV4, // 两个 index 都是 enum

.input\_index = PROTO\_INPUT\_IPV4,

.offset = sizeof(struct ether\_hdr) + // todo ：数据结构

offsetof(struct ipv4\_hdr, next\_proto\_id),

},

/\* next input field (IPv4 source address) - 4 consecutive bytes. \*/

{ // 第二个字段 源IP地址

/\* rte\_flow uses a bit mask for IPv4 addresses \*/

.type = RTE\_ACL\_FIELD\_TYPE\_BITMASK,

.size = sizeof(uint32\_t),

.field\_index = SRC\_FIELD\_IPV4,

.input\_index = SRC\_INPUT\_IPV4,

.offset = sizeof(struct ether\_hdr) +

offsetof(struct ipv4\_hdr, src\_addr),

},

/\* next input field (IPv4 destination address) - 4 consecutive bytes. \*/

{ // 第三个字段 目的IP地址

/\* rte\_flow uses a bit mask for IPv4 addresses \*/

.type = RTE\_ACL\_FIELD\_TYPE\_BITMASK,

.size = sizeof(uint32\_t),

.field\_index = DST\_FIELD\_IPV4,

.input\_index = DST\_INPUT\_IPV4,

.offset = sizeof(struct ether\_hdr) +

offsetof(struct ipv4\_hdr, dst\_addr),

},

/\*

\* Next 2 fields (src & dst ports) form 4 consecutive bytes.

\* They share the same input index.

\*/

// 接下来的 两个端口号 才组成一个 4 字节，所以共享同样的一个 input index

{

/\* rte\_flow uses a bit mask for protocol ports \*/

.type = RTE\_ACL\_FIELD\_TYPE\_BITMASK,

.size = sizeof(uint16\_t),

.field\_index = SRCP\_FIELD\_IPV4,

.input\_index = SRCP\_DESTP\_INPUT\_IPV4,

.offset = sizeof(struct ether\_hdr) + // (todo)

sizeof(struct ipv4\_hdr) +

offsetof(struct tcp\_hdr, src\_port),

},

{

/\* rte\_flow uses a bit mask for protocol ports \*/

.type = RTE\_ACL\_FIELD\_TYPE\_BITMASK,

.size = sizeof(uint16\_t),

.field\_index = DSTP\_FIELD\_IPV4,

.input\_index = SRCP\_DESTP\_INPUT\_IPV4,

.offset = sizeof(struct ether\_hdr) +

sizeof(struct ipv4\_hdr) +

offsetof(struct tcp\_hdr, dst\_port),

},

};

/\* flow classify data \*/

static int num\_classify\_rules; // rules数组的下标

static struct rte\_flow\_classify\_rule \*rules[MAX\_NUM\_CLASSIFY]; // rules 数组

static struct rte\_flow\_classify\_ipv4\_5tuple\_stats ntuple\_stats; // stats 结构体 （todo）

static struct rte\_flow\_classify\_stats classify\_stats = { // 有计数功能

.stats = (void \*\*)&ntuple\_stats

};

/\* parameters for rte\_flow\_classify\_validate and

\* rte\_flow\_classify\_table\_entry\_add functions

\*/

/\* rte\_flow\_item 四个字段：

1. type，是 enum 定义。见 rte\_flow.h：http://doc.dpdk.org/api/rte\_\_flow\_8h\_source.html

2. spec，指向相关项类型结构的有效指针，在许多情况下，可以设置成 NULL以请求广泛（非特定）匹配。在此情况下，last 和 mask 也要设置成 NULL

3. last，可以指向相同类型的结构，以定义包含范围。

4. Mask，是在解释spec和last的内容之前应用的简单位掩码

\*/

static struct rte\_flow\_item eth\_item = { RTE\_FLOW\_ITEM\_TYPE\_ETH,

0, 0, 0 };

static struct rte\_flow\_item end\_item = { RTE\_FLOW\_ITEM\_TYPE\_END,

0, 0, 0 };

/\* sample actions:

\* "actions count / end"

\*/

struct rte\_flow\_query\_count count = { // 计数器查询的结构体

.reset = 1, // Reset counters after query

.hits\_set = 1, // 启用 hits 字段

.bytes\_set = 1, // 启用 bytes 字段

.hits = 0, // Number of hits for this rule

.bytes = 0, // Number of bytes through this rule

};

static struct rte\_flow\_action count\_action = { RTE\_FLOW\_ACTION\_TYPE\_COUNT, &count};

static struct rte\_flow\_action end\_action = { RTE\_FLOW\_ACTION\_TYPE\_END, 0}; // 本程序就用到了计数和end 两种 action

static struct rte\_flow\_action actions[2];

// rte\_flow\_action 见 programmers’ guides 的第九章 ：http://doc.dpdk.org/guides/prog\_guide/rte\_flow.html

// actions 数组代表当 pkt 被 pattern 匹配时要执行的一系列操作。

// 在这个例子里，数组长度为二，actions[0] 就是计数，actions[1] 就是用来提示结尾。

// rte\_flow\_action的具体定义不清楚

// 估计第一个字段是 enum rte\_flow\_action\_type ，具体的 enum 定义见：http://doc.dpdk.org/api/rte\_\_flow\_8h.html#a78f0386e683cfc491462a771df8b971a

// 第二个字段计数器查询的结构体

/\* sample attributes \*/

static struct rte\_flow\_attr attr;

/\* rte\_flow\_attr 代表一条流规则的属性，文档：http://doc.dpdk.org/api/structrte\_\_flow\_\_attr.html

字段：

uint32\_t group 组号

uint32\_t priority 同组内的优先级

uint32\_t ingress:1 规则适用于入口流量

uint32\_t egress:1 规则适用于出口流量

uint32\_t transfer:1 todo

uint32\_t reserved:29 保留，必须为零。

\*/

/\* flow\_classify.c: \* Based on DPDK skeleton forwarding example. \*/

/\*

\* Initializes a given port using global settings and with the RX buffers

\* coming from the mbuf\_pool passed as a parameter.

\*/

// 端口初始化的代码与 basicfw 一模一样

static inline int

port\_init(uint8\_t port, struct rte\_mempool \*mbuf\_pool)

{

struct rte\_eth\_conf port\_conf = port\_conf\_default;

struct ether\_addr addr;

const uint16\_t rx\_rings = 1, tx\_rings = 1;

int retval;

uint16\_t q;

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

struct rte\_eth\_txconf txconf;

if (!rte\_eth\_dev\_is\_valid\_port(port))

return -1;

rte\_eth\_dev\_info\_get(port, &dev\_info);

if (dev\_info.tx\_offload\_capa & DEV\_TX\_OFFLOAD\_MBUF\_FAST\_FREE)

port\_conf.txmode.offloads |=

DEV\_TX\_OFFLOAD\_MBUF\_FAST\_FREE;

/\* Configure the Ethernet device. \*/

retval = rte\_eth\_dev\_configure(port, rx\_rings, tx\_rings, &port\_conf);

if (retval != 0)

return retval;

/\* Allocate and set up 1 RX queue per Ethernet port. \*/

for (q = 0; q < rx\_rings; q++) {

retval = rte\_eth\_rx\_queue\_setup(port, q, RX\_RING\_SIZE,

rte\_eth\_dev\_socket\_id(port), NULL, mbuf\_pool);

if (retval < 0)

return retval;

}

txconf = dev\_info.default\_txconf;

txconf.txq\_flags = ETH\_TXQ\_FLAGS\_IGNORE;

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

/\* Allocate and set up 1 TX queue per Ethernet port. \*/

for (q = 0; q < tx\_rings; q++) {

retval = rte\_eth\_tx\_queue\_setup(port, q, TX\_RING\_SIZE,

rte\_eth\_dev\_socket\_id(port), &txconf);

if (retval < 0)

return retval;

}

/\* Start the Ethernet port. \*/

retval = rte\_eth\_dev\_start(port);

if (retval < 0)

return retval;

/\* Display the port MAC address. \*/

rte\_eth\_macaddr\_get(port, &addr);

printf("Port %u MAC: %02" PRIx8 " %02" PRIx8 " %02" PRIx8

" %02" PRIx8 " %02" PRIx8 " %02" PRIx8 "\n",

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]);

/\* Enable RX in promiscuous mode for the Ethernet device. \*/

rte\_eth\_promiscuous\_enable(port);

return 0;

}

/\*

\* The lcore main. This is the main thread that does the work, reading from

\* an input port classifying the packets and writing to an output port.

\*/

static \_\_attribute\_\_((noreturn)) void

lcore\_main(struct flow\_classifier \*cls\_app)

{

uint16\_t port;

int ret;

int i = 0;

// 测试：删除一条规则

ret = rte\_flow\_classify\_table\_entry\_delete(cls\_app->cls,

rules[7]);

if (ret)

printf("table\_entry\_delete failed [7] %d\n\n", ret);

else

printf("table\_entry\_delete succeeded [7]\n\n");

/\*

\* Check that the port is on the same NUMA node as the polling thread

\* for best performance.

\*/

RTE\_ETH\_FOREACH\_DEV(port)

if (rte\_eth\_dev\_socket\_id(port) > 0 &&

rte\_eth\_dev\_socket\_id(port) != (int)rte\_socket\_id()) {

printf("\n\n");

printf("WARNING: port %u is on remote NUMA node\n",

port);

printf("to polling thread.\n");

printf("Performance will not be optimal.\n");

}

printf("\nCore %u forwarding packets. ", rte\_lcore\_id());

printf("[Ctrl+C to quit]\n");

/\* Run until the application is quit or killed. \*/

for (;;) {

/\*

\* Receive packets on a port, \*\*classify them\*\* and forward them

\* on the paired port.

\* The mapping is 0 -> 1, 1 -> 0, 2 -> 3, 3 -> 2, etc.

\*/

RTE\_ETH\_FOREACH\_DEV(port) {

/\* Get burst of RX packets, from first port of pair. \*/

struct rte\_mbuf \*bufs[BURST\_SIZE];

const uint16\_t nb\_rx = rte\_eth\_rx\_burst(port, 0,

bufs, BURST\_SIZE); // 收包

if (unlikely(nb\_rx == 0))

continue;

for (i = 0; i < MAX\_NUM\_CLASSIFY; i++) {

if (rules[i]) { // 对classifier里的每条规则（用一个数组来保存插入成功时返回的rule指针）

/\* rte\_flow\_classifier\_query()，查看burst中是否有任何数据包与表中的一条流规则匹配。

参数：流分类器句柄、要处理的数据包的mbuf

一个burst的数据包数量、要查询的规则、查询的stat \*/

ret = rte\_flow\_classifier\_query(

cls\_app->cls,

bufs, nb\_rx, rules[i],

&classify\_stats);

if (ret)

printf(

"rule [%d] query failed ret [%d]\n\n",

i, ret);

else { // 返回 0 代表有match

printf(

"rule[%d] count=%"PRIu64"\n",

i, ntuple\_stats.counter1);

printf("proto = %d\n",

ntuple\_stats.ipv4\_5tuple.proto);

}

}

}

/\* Send burst of TX packets, to second port of pair. \*/

const uint16\_t nb\_tx = rte\_eth\_tx\_burst(port ^ 1, 0,

bufs, nb\_rx);

/\* Free any unsent packets. \*/

if (unlikely(nb\_tx < nb\_rx)) {

uint16\_t buf;

for (buf = nb\_tx; buf < nb\_rx; buf++)

rte\_pktmbuf\_free(bufs[buf]);

}

}

}

}

/\*

\* Parse IPv4 5 tuple rules file, ipv4\_rules\_file.txt.

\* Expected format:

\* <src\_ipv4\_addr>'/'<masklen> <space> \

\* <dst\_ipv4\_addr>'/'<masklen> <space> \

\* <src\_port> <space> ":" <src\_port\_mask> <space> \

\* <dst\_port> <space> ":" <dst\_port\_mask> <space> \

\* <proto>'/'<proto\_mask> <space> \

\* <priority>

\*/

static int

get\_cb\_field(char \*\*in, uint32\_t \*fd, int base, unsigned long lim,

char dlm)

{

unsigned long val;

char \*end;

errno = 0;

val = strtoul(\*in, &end, base);

/\* unsigned long int strtoul(const char \*str, char \*\*endptr, int base)

把参数 str 所指向的字符串根据给定的 base 转换为一个无符号长整数（unsigned long int 型）。

str -- 要转换为无符号长整数的字符串。

endptr -- 对类型为 char\* 的对象的引用，其值会由函数设置为 str 中数值后的下一个字符。

（end 会指向点分十进制中的下一个点）

base -- 基数，必须介于 2 和 36（包含）之间，或者是特殊值 0。

当base = 0,自动判断字符串的类型，并按10进制输出，例如"0xa", 就会把字符串当做16进制处理，输出为 10。

参考：http://www.runoob.com/cprogramming/c-function-strtoul.html

https://blog.csdn.net/chuhongcai/article/details/52032926

\*/

if (errno != 0 || end[0] != dlm || val > lim)

return -EINVAL;

\*fd = (uint32\_t)val;

\*in = end + 1; // 例如 2.2.2.3 会依次转换 2 2 2 3

return 0;

}

static int

parse\_ipv4\_net(char \*in, uint32\_t \*addr, uint32\_t \*mask\_len)

{

// in: 2.2.2.3/24

uint32\_t a, b, c, d, m;

// 这四个if是判断IP地址的每个点分十进制是否小于255（UINT8\_MAX）

if (get\_cb\_field(&in, &a, 0, UINT8\_MAX, '.'))

return -EINVAL;

if (get\_cb\_field(&in, &b, 0, UINT8\_MAX, '.'))

return -EINVAL;

if (get\_cb\_field(&in, &c, 0, UINT8\_MAX, '.'))

return -EINVAL;

if (get\_cb\_field(&in, &d, 0, UINT8\_MAX, '/'))

return -EINVAL;

// 后缀要小于32

if (get\_cb\_field(&in, &m, 0, sizeof(uint32\_t) \* CHAR\_BIT, 0))

return -EINVAL;

addr[0] = IPv4(a, b, c, d);

mask\_len[0] = m;

return 0;

}

static int

parse\_ipv4\_5tuple\_rule(char \*str, struct rte\_eth\_ntuple\_filter \*ntuple\_filter)

// 将 txt 中一行输入，转换成一个 rte\_eth\_ntuple\_filter 结构体。

{

int i, ret;

char \*s, \*sp, \*in[CB\_FLD\_NUM];

static const char \*dlm = " \t\n";

int dim = CB\_FLD\_NUM; // 10

uint32\_t temp;

s = str;

for (i = 0; i != dim; i++, s = NULL) {

in[i] = strtok\_r(s, dlm, &sp);

// linux下的字符串切割函数：strtok\_r

/\* char \*strtok\_r(char \*str, const char \*delim, char \*\*saveptr);

在str中，返回由delim指定的分界符分开str的单词。

参考链接：https://blog.csdn.net/hustfoxy/article/details/23473805

\*/

if (in[i] == NULL)

return -EINVAL;

}

/\* 一条 rule 占一行，格式，以及分词后的在in数组内的下标如下：

#源IP/前缀 目的IP/前缀 源端口号 : 掩码 目的端口号 : 掩码 协议/掩码 优先级

2.2.2.3/24 2.2.2.7/24 32 : 0xffff 33 : 0xffff 17/0xff 0

0 1 2 3 4 5 6 7 8 9 ← in数组下标

\*/

/\* rte\_eth\_ntuple\_filter 的字段：

uint16\_t flags

uint32\_t dst\_ip Destination IP address in big endian.

uint32\_t dst\_ip\_mask

uint32\_t src\_ip in big endian.

uint32\_t src\_ip\_mask

uint16\_t dst\_port Destination port in big endian.

uint16\_t dst\_port\_mask

uint16\_t src\_port in big endian.

uint16\_t src\_port\_mask

uint8\_t proto L4 protocol.

uint8\_t proto\_mask

uint8\_t tcp\_flags only meaningful when the proto is TCP.

uint16\_t priority seven levels (001b-111b), 111b is highest, used when more than one filter matches.

uint16\_t queue Queue assigned to when match

\*/

ret = parse\_ipv4\_net(in[CB\_FLD\_SRC\_ADDR],

&ntuple\_filter->src\_ip,

&ntuple\_filter->src\_ip\_mask); // 解析 src\_ip 得到IP地址和掩码，放到 ntuple\_filter的对应字段里

if (ret != 0) {

flow\_classify\_log("failed to read source address/mask: %s\n",

in[CB\_FLD\_SRC\_ADDR]);

return ret;

}

ret = parse\_ipv4\_net(in[CB\_FLD\_DST\_ADDR], // 解析 dst\_ip

&ntuple\_filter->dst\_ip,

&ntuple\_filter->dst\_ip\_mask);

if (ret != 0) {

flow\_classify\_log("failed to read source address/mask: %s\n",

in[CB\_FLD\_DST\_ADDR]);

return ret;

}

if (get\_cb\_field(&in[CB\_FLD\_SRC\_PORT], &temp, 0, UINT16\_MAX, 0))

return -EINVAL; // 源端口号字符串转 unsigned long ，验证不能大于16位无符号数的最大值。

ntuple\_filter->src\_port = (uint16\_t)temp;

if (strncmp(in[CB\_FLD\_SRC\_PORT\_DLM], cb\_port\_delim,

sizeof(cb\_port\_delim)) != 0) // 检查分隔符是否为: 不然是格式错误。

return -EINVAL;

if (get\_cb\_field(&in[CB\_FLD\_SRC\_PORT\_MASK], &temp, 0, UINT16\_MAX, 0))

return -EINVAL; // 源端口号掩码

ntuple\_filter->src\_port\_mask = (uint16\_t)temp;

if (get\_cb\_field(&in[CB\_FLD\_DST\_PORT], &temp, 0, UINT16\_MAX, 0))

return -EINVAL; // 目的端口号

ntuple\_filter->dst\_port = (uint16\_t)temp;

if (strncmp(in[CB\_FLD\_DST\_PORT\_DLM], cb\_port\_delim,

sizeof(cb\_port\_delim)) != 0)

return -EINVAL;

if (get\_cb\_field(&in[CB\_FLD\_DST\_PORT\_MASK], &temp, 0, UINT16\_MAX, 0))

return -EINVAL; // 目的端口号掩码

ntuple\_filter->dst\_port\_mask = (uint16\_t)temp;

if (get\_cb\_field(&in[CB\_FLD\_PROTO], &temp, 0, UINT8\_MAX, '/'))

return -EINVAL; // 协议号

ntuple\_filter->proto = (uint8\_t)temp;

if (get\_cb\_field(&in[CB\_FLD\_PROTO], &temp, 0, UINT8\_MAX, 0))

return -EINVAL; // 协议号掩码

ntuple\_filter->proto\_mask = (uint8\_t)temp;

if (get\_cb\_field(&in[CB\_FLD\_PRIORITY], &temp, 0, UINT16\_MAX, 0))

return -EINVAL; // 优先级

ntuple\_filter->priority = (uint16\_t)temp;

if (ntuple\_filter->priority > FLOW\_CLASSIFY\_MAX\_PRIORITY)

ret = -EINVAL;

return ret;

}

/\* Bypass comment and empty lines \*/

static inline int

is\_bypass\_line(char \*buff)

{

int i = 0;

/\* comment line \*/

if (buff[0] == COMMENT\_LEAD\_CHAR)

return 1;

/\* empty line \*/

while (buff[i] != '\0') {

if (!isspace(buff[i]))

return 0;

i++;

}

return 1;

}

static uint32\_t

convert\_depth\_to\_bitmask(uint32\_t depth\_val)

{

uint32\_t bitmask = 0;

int i, j;

for (i = depth\_val, j = 0; i > 0; i--, j++)

bitmask |= (1 << (31 - j));

return bitmask;

}

static int

add\_classify\_rule(struct rte\_eth\_ntuple\_filter \*pattern\_ipv4\_5tuple,

struct flow\_classifier \*cls\_app)

// 对 rte\_flow\_classify\_table\_entry\_add() 的一层封装，主要是设定好参数，从rte\_eth\_ntuple\_filter 转换成 flow\_item

{

int ret = -1;

int key\_found;

struct rte\_flow\_error error;

/\* rte\_flow\_item： ACL 规则的详细内容。

会从最低协议层开始堆叠flow\_item来形成一个匹配模式。必须由 end\_item 结尾。

\*/

struct rte\_flow\_item\_ipv4 ipv4\_spec; // (todo) rte\_flow\_item . Matches an IPv4 header.

struct rte\_flow\_item\_ipv4 ipv4\_mask;

struct rte\_flow\_item ipv4\_udp\_item;

struct rte\_flow\_item ipv4\_tcp\_item;

struct rte\_flow\_item ipv4\_sctp\_item;

struct rte\_flow\_item\_udp udp\_spec;

struct rte\_flow\_item\_udp udp\_mask;

struct rte\_flow\_item udp\_item;

struct rte\_flow\_item\_tcp tcp\_spec;

struct rte\_flow\_item\_tcp tcp\_mask;

struct rte\_flow\_item tcp\_item;

struct rte\_flow\_item\_sctp sctp\_spec;

struct rte\_flow\_item\_sctp sctp\_mask;

struct rte\_flow\_item sctp\_item;

struct rte\_flow\_item pattern\_ipv4\_5tuple[4]; // ntuple\_filter 结构体 --> rte\_flow\_item 结构体数组

struct rte\_flow\_classify\_rule \*rule;

uint8\_t ipv4\_proto;

if (num\_classify\_rules >= MAX\_NUM\_CLASSIFY) {

printf(

"\nINFO: classify rule capacity %d reached\n",

num\_classify\_rules);

return ret;

}

/\* set up parameters for validate and add \*/

memset(&ipv4\_spec, 0, sizeof(ipv4\_spec));

ipv4\_spec.hdr.next\_proto\_id = ntuple\_filter->proto; // 协议号

ipv4\_spec.hdr.src\_addr = ntuple\_filter->src\_ip; // 源IP

ipv4\_spec.hdr.dst\_addr = ntuple\_filter->dst\_ip; // 目的IP

ipv4\_proto = ipv4\_spec.hdr.next\_proto\_id;

// 把这三个参数从ntuple\_filter结构体提取到 rte\_flow\_item\_ipv4 的一个专门的结构体：ipv4\_spec

memset(&ipv4\_mask, 0, sizeof(ipv4\_mask));

ipv4\_mask.hdr.next\_proto\_id = ntuple\_filter->proto\_mask; // 协议掩码

ipv4\_mask.hdr.src\_addr = ntuple\_filter->src\_ip\_mask;

ipv4\_mask.hdr.src\_addr =

convert\_depth\_to\_bitmask(ipv4\_mask.hdr.src\_addr);

ipv4\_mask.hdr.dst\_addr = ntuple\_filter->dst\_ip\_mask; // 源IP地址的掩码

ipv4\_mask.hdr.dst\_addr =

convert\_depth\_to\_bitmask(ipv4\_mask.hdr.dst\_addr); // 目的IP地址的掩码

// 把这三个参数从ntuple\_filter结构体提取到 rte\_flow\_item\_ipv4 的一个专门的结构体 ：ipv4\_mask

switch (ipv4\_proto) { // 根据协议设置L3、L4的item

case IPPROTO\_UDP: // UDP

ipv4\_udp\_item.type = RTE\_FLOW\_ITEM\_TYPE\_IPV4;

ipv4\_udp\_item.spec = &ipv4\_spec;

ipv4\_udp\_item.mask = &ipv4\_mask;

ipv4\_udp\_item.last = NULL;

udp\_spec.hdr.src\_port = ntuple\_filter->src\_port;

udp\_spec.hdr.dst\_port = ntuple\_filter->dst\_port;

udp\_spec.hdr.dgram\_len = 0;

udp\_spec.hdr.dgram\_cksum = 0;

udp\_mask.hdr.src\_port = ntuple\_filter->src\_port\_mask;

udp\_mask.hdr.dst\_port = ntuple\_filter->dst\_port\_mask;

udp\_mask.hdr.dgram\_len = 0;

udp\_mask.hdr.dgram\_cksum = 0;

udp\_item.type = RTE\_FLOW\_ITEM\_TYPE\_UDP;

udp\_item.spec = &udp\_spec;

udp\_item.mask = &udp\_mask;

udp\_item.last = NULL;

attr.priority = ntuple\_filter->priority;

pattern\_ipv4\_5tuple[1] = ipv4\_udp\_item; // L3 item 是 ipv4\_upd

pattern\_ipv4\_5tuple[2] = udp\_item; // L4 item 是 udp\_item

break;

case IPPROTO\_TCP: // TCP

ipv4\_tcp\_item.type = RTE\_FLOW\_ITEM\_TYPE\_IPV4;

ipv4\_tcp\_item.spec = &ipv4\_spec;

ipv4\_tcp\_item.mask = &ipv4\_mask;

ipv4\_tcp\_item.last = NULL;

memset(&tcp\_spec, 0, sizeof(tcp\_spec));

tcp\_spec.hdr.src\_port = ntuple\_filter->src\_port;

tcp\_spec.hdr.dst\_port = ntuple\_filter->dst\_port;

memset(&tcp\_mask, 0, sizeof(tcp\_mask));

tcp\_mask.hdr.src\_port = ntuple\_filter->src\_port\_mask;

tcp\_mask.hdr.dst\_port = ntuple\_filter->dst\_port\_mask;

tcp\_item.type = RTE\_FLOW\_ITEM\_TYPE\_TCP;

tcp\_item.spec = &tcp\_spec;

tcp\_item.mask = &tcp\_mask;

tcp\_item.last = NULL;

attr.priority = ntuple\_filter->priority;

pattern\_ipv4\_5tuple[1] = ipv4\_tcp\_item; // L3 item 是 ipv4\_tcp

pattern\_ipv4\_5tuple[2] = tcp\_item; // L4 item 是 tcp\_item

break;

case IPPROTO\_SCTP:

ipv4\_sctp\_item.type = RTE\_FLOW\_ITEM\_TYPE\_IPV4;

ipv4\_sctp\_item.spec = &ipv4\_spec;

ipv4\_sctp\_item.mask = &ipv4\_mask;

ipv4\_sctp\_item.last = NULL;

sctp\_spec.hdr.src\_port = ntuple\_filter->src\_port;

sctp\_spec.hdr.dst\_port = ntuple\_filter->dst\_port;

sctp\_spec.hdr.cksum = 0;

sctp\_spec.hdr.tag = 0;

sctp\_mask.hdr.src\_port = ntuple\_filter->src\_port\_mask;

sctp\_mask.hdr.dst\_port = ntuple\_filter->dst\_port\_mask;

sctp\_mask.hdr.cksum = 0;

sctp\_mask.hdr.tag = 0;

sctp\_item.type = RTE\_FLOW\_ITEM\_TYPE\_SCTP;

sctp\_item.spec = &sctp\_spec;

sctp\_item.mask = &sctp\_mask;

sctp\_item.last = NULL;

attr.priority = ntuple\_filter->priority;

pattern\_ipv4\_5tuple[1] = ipv4\_sctp\_item;

pattern\_ipv4\_5tuple[2] = sctp\_item;

break;

default:

return ret;

}

attr.ingress = 1; // rules 适用于入口流量

pattern\_ipv4\_5tuple[0] = eth\_item;// L2 item，放在pattern\_ipv4\_5tuple[0]，一定是eth\_item

// L3 item 放在数组下标1，L4 item放在数组下标2

pattern\_ipv4\_5tuple[3] = end\_item; // 最后一个 item 一定要用 end\_item 结尾。

actions[0] = count\_action; // 流匹配的动作是 计数

actions[1] = end\_action; // (terminated by the END pattern item)

/\* Validate and add rule \*/

/\* 验证这条规则的有效性

参数：

1. classifer 指针

2. attr 指针，流规则的属性，详细内容见上。

3. rte\_flow\_item 结构体数组(terminated by the END pattern item)，也就是 ACL 规则的详细内容

4. rte\_flow\_action 结构体数组(terminated by the END pattern item)，表示流规则的动作，比如QUEUE, DROP, END等等，

5. struct rte\_flow\_error，出错时存放信息。

\*/

ret = rte\_flow\_classify\_validate(cls\_app->cls, &attr,

pattern\_ipv4\_5tuple, actions, &error);

if (ret) { // 成功时返回 0

printf("table entry validate failed ipv4\_proto = %u\n",

ipv4\_proto);

return ret;

}

// 调用 rte\_flow\_classify\_table\_entry\_add() 将规则添加到 rte\_flow\_classifier 对象中的 table。

/\* 五个参数

1. classifier 的指针。

2. attr 指针。

3. rte\_flow\_item 结构体数组，也就是 ACL 规则的详细内容。

4. rte\_flow\_action 结构体数组，表示流规则的动作。

5. 一个int指针，如果规则已经存在则返回1，否则返回0。

6. 仅出错时存放信息。

\*/

rule = rte\_flow\_classify\_table\_entry\_add(

cls\_app->cls, &attr, pattern\_ipv4\_5tuple,

actions, &key\_found, &error);

if (rule == NULL) { // 添加成功时返回的是rule的有效句柄，否则为NULL

printf("table entry add failed ipv4\_proto = %u\n",

ipv4\_proto);

ret = -1;

return ret;

}

rules[num\_classify\_rules] = rule; // 将rule存放在一个数组里，方便删除等操作

num\_classify\_rules++;

return 0;

}

static int

add\_rules(const char \*rule\_path, struct flow\_classifier \*cls\_app)

// 封装一层，主要是文件操作，把txt中的一行解析成 rte\_eth\_ntuple\_filter 结构体

{

FILE \*fh;

char buff[LINE\_MAX];

unsigned int i = 0;

unsigned int total\_num = 0;

struct rte\_eth\_ntuple\_filter ntuple\_filter; // 用于定义n-tuple过滤器条目的结构体

int ret;

fh = fopen(rule\_path, "rb"); // 打开 ipv4\_rules\_file.txt

if (fh == NULL)

rte\_exit(EXIT\_FAILURE, "%s: fopen %s failed\n", \_\_func\_\_,

rule\_path);

ret = fseek(fh, 0, SEEK\_SET); // 设置文件指针fh的位置指向文件开头

if (ret) // 成功，返回0

rte\_exit(EXIT\_FAILURE, "%s: fseek %d failed\n", \_\_func\_\_,

ret);

i = 0;

while (fgets(buff, LINE\_MAX, fh) != NULL) { // 读取一行内容

i++;

if (is\_bypass\_line(buff)) // 跳过空行 or 以井号开头的注释

continue;

if (total\_num >= FLOW\_CLASSIFY\_MAX\_RULE\_NUM - 1) { // 有最大规则数量（行数）限制

printf("\nINFO: classify rule capacity %d reached\n",

total\_num);

break;

}

if (parse\_ipv4\_5tuple\_rule(buff, &ntuple\_filter) != 0) // 规则的 parser 解析txt的一行输入，存放到ntuple\_filter结构体里

rte\_exit(EXIT\_FAILURE,

"%s Line %u: parse rules error\n",

rule\_path, i);

if (add\_classify\_rule(&ntuple\_filter, cls\_app) != 0) // 添加这条五元组规则到 ACL 中

rte\_exit(EXIT\_FAILURE, "add rule error\n");

total\_num++;

}

fclose(fh);

return 0;

}

/\* display usage \*/

static void

print\_usage(const char \*prgname)

{

printf("%s usage:\n", prgname);

printf("[EAL options] -- --"OPTION\_RULE\_IPV4"=FILE: ");

printf("specify the ipv4 rules file.\n");

printf("Each rule occupies one line in the file.\n");

}

/\* Parse the argument given in the command line of the application \*/

// 解析执行 flow\_classify 的命令行参数

static int

parse\_args(int argc, char \*\*argv)

{

int opt, ret;

char \*\*argvopt;

int option\_index;

char \*prgname = argv[0];

static struct option lgopts[] = {

{OPTION\_RULE\_IPV4, 1, 0, 0},

{NULL, 0, 0, 0}

};

argvopt = argv;

while ((opt = getopt\_long(argc, argvopt, "",

lgopts, &option\_index)) != EOF) {

switch (opt) {

/\* long options \*/

case 0:

if (!strncmp(lgopts[option\_index].name,

OPTION\_RULE\_IPV4,

sizeof(OPTION\_RULE\_IPV4)))

parm\_config.rule\_ipv4\_name = optarg;

break;

default:

print\_usage(prgname);

return -1;

}

}

if (optind >= 0)

argv[optind-1] = prgname;

ret = optind-1;

optind = 1; /\* reset getopt lib \*/

return ret;

}

/\*

\* The main function, which does initialization and calls the lcore\_main

\* function.

\*/

int

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

{

struct rte\_mempool \*mbuf\_pool;

uint8\_t nb\_ports;

uint16\_t portid;

int ret;

int socket\_id;

// 以下可以在 dpdk api data struct 中查看

struct rte\_table\_acl\_params table\_acl\_params; // ACL table 的参数

struct rte\_flow\_classify\_table\_params cls\_table\_params; // Parameters for table creation

struct flow\_classifier \*cls\_app; // 分流器

// 分流器的内部结构要见https://doc.dpdk.org/guides/prog\_guide/flow\_classify\_lib.html#classifier-creation

struct rte\_flow\_classifier\_params cls\_params; // classifier 的参数

uint32\_t size;

/\* Initialize the Environment Abstraction Layer (EAL). \*/

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

if (ret < 0)

rte\_exit(EXIT\_FAILURE, "Error with EAL initialization\n");

argc -= ret;

argv += ret;

/\* parse application arguments (after the EAL ones) \*/

ret = parse\_args(argc, argv); // 解析 flow\_classify 的命令行参数

if (ret < 0)

rte\_exit(EXIT\_FAILURE, "Invalid flow\_classify parameters\n");

/\* Check that there is an even number of ports to send/receive on. \*/

nb\_ports = rte\_eth\_dev\_count(); // 网口数目必须是偶数

if (nb\_ports < 2 || (nb\_ports & 1))

rte\_exit(EXIT\_FAILURE, "Error: number of ports must be even\n");

/\* Creates a new mempool in memory to hold the mbufs. \*/

// 创建mempool

mbuf\_pool = rte\_pktmbuf\_pool\_create("MBUF\_POOL", NUM\_MBUFS \* nb\_ports,

MBUF\_CACHE\_SIZE, 0, RTE\_MBUF\_DEFAULT\_BUF\_SIZE, rte\_socket\_id());

if (mbuf\_pool == NULL)

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

/\* Initialize all ports. \*/

RTE\_ETH\_FOREACH\_DEV(portid) // 端口初始化 与basicfw的一样

if (port\_init(portid, mbuf\_pool) != 0)

rte\_exit(EXIT\_FAILURE, "Cannot init port %"PRIu8 "\n",

portid);

if (rte\_lcore\_count() > 1) // 只需要一个逻辑核心

printf("\nWARNING: Too many lcores enabled. Only 1 used.\n");

socket\_id = rte\_eth\_dev\_socket\_id(0); // 返回 0 号网口所在的NUMA socket id号

/\* Memory allocation \*/

// 为分流器 cls\_app 分配内存

size = RTE\_CACHE\_LINE\_ROUNDUP(sizeof(struct flow\_classifier\_acl));// 返回大于或等于宏定义参数的第一个缓存对齐值

cls\_app = rte\_zmalloc(NULL, size, RTE\_CACHE\_LINE\_SIZE); // DPDK的malloc：从调用该函数的核上的同一个NUMA socket的大页面区域分配堆内存。

// zmalloc 就是清零 与 calloc 相似

/\* rte\_zmalloc 参数三个：

1. 指示这块区域分配给怎样的object类型。用于debug用途。可以写NULL

2. size (in bytes) to be allocated，这里分配一个cache缓存行的字节。

3. align

if 0, 会返回一个适合任何类型变量的指针，就像 malloc

否则，返回一个内存区域是 align 的对齐倍数，显然最小对齐是高速缓存行大小，宏：RTE\_CACHE\_LINE\_SIZE

\*/

if (cls\_app == NULL) // 分配内存失败

rte\_exit(EXIT\_FAILURE, "Cannot allocate classifier memory\n");

// classifier 的参数 有两个： name 和 socket id

// 需要在调用 create() API 之前由应用程序初始化

cls\_params.name = "flow\_classifier";

cls\_params.socket\_id = socket\_id;

// 调用 rte\_flow\_classifier\_create() 函数来创建rte\_flow\_classifier对象。

// 参数是 rte\_flow\_classifier\_params 结构体指针

cls\_app->cls = rte\_flow\_classifier\_create(&cls\_params);

if (cls\_app->cls == NULL) { // 创建失败

rte\_free(cls\_app);

rte\_exit(EXIT\_FAILURE, "Cannot create classifier\n");

}

/\* initialise ACL table params \*/

// 填写 ACL 的初始化参数

// 四个字段：

table\_acl\_params.name = "table\_acl\_ipv4\_5tuple"; // ACL的名字

table\_acl\_params.n\_rules = FLOW\_CLASSIFY\_MAX\_RULE\_NUM; // 表中最大ACL规则数量：91

table\_acl\_params.n\_rule\_fields = RTE\_DIM(ipv4\_defs); // 一条ACL规则中的有多少个字段（fields）

//宏定义如下：#define RTE\_DIM(a) (sizeof (a) / sizeof ((a)[0])) 直观看就是返回数组的长度。

memcpy(table\_acl\_params.field\_format, ipv4\_defs, sizeof(ipv4\_defs));

// ACL rule 的详细内容 specification

// ACL 规则的字段也必须由应用程序初始化。

/\* initialise table create params \*/

// 填写 表 的创建参数

// 三个字段：

cls\_table\_params.ops = &rte\_table\_acl\_ops; //表操作（特定于每个表类型），（todo：这里不清楚具体是怎么操作的

cls\_table\_params.arg\_create = &table\_acl\_params; // 传递给表的用于创建的参数 这里是ACL的初始化参数结构体的指针

cls\_table\_params.type = RTE\_FLOW\_CLASSIFY\_TABLE\_ACL\_IP4\_5TUPLE; // table's type，是一个 enum

// rte\_flow\_classify\_table\_create() 向classifier对象添加一个表。

// 参数两个：1. 流分类器的指针 2. 表创建的参数

ret = rte\_flow\_classify\_table\_create(cls\_app->cls, &cls\_table\_params);

if (ret) { // 返回值：成功时返回 0

rte\_flow\_classifier\_free(cls\_app->cls);

rte\_free(cls\_app);

rte\_exit(EXIT\_FAILURE, "Failed to create classifier table\n");

}

/\* read file of IPv4 5 tuple rules and initialize parameters

\* for rte\_flow\_classify\_validate and rte\_flow\_classify\_table\_entry\_add

\* API's.

\*/

// 然后它读取ipv4\_rules\_file.txt文件，验证流规则是否合法，然后初始化rte\_flow\_classify\_table\_entry\_add() API 的参数，使用此API将规则添加到ACL表。

if (add\_rules(parm\_config.rule\_ipv4\_name, cls\_app)) {

rte\_flow\_classifier\_free(cls\_app->cls);

rte\_free(cls\_app);

rte\_exit(EXIT\_FAILURE, "Failed to add rules\n");

}

/\* Call lcore\_main on the master core only. \*/ // todo

lcore\_main(cls\_app);

return 0;

}

基本看完了，但开头有很多结构体和宏定义，没有办法在 API doc 中找到确切的页面。第一个是因为 DPDK src code 中对那些数据结构有经常的改动，文档上的改动没有跟上。还有就是有用到一些 Intel 各种宏定义，并不是在 DPDK 的 API doc 中有体现。

flow\_classify 这个程序做的事情分为如下几步骤：

1. EAL初始化、端口初始化、分配内存等，与basicfw是一样的。
2. 创建 flow\_classifer对象。这一个过程在代码中体现好几个阶段：为classifier分配内存、填写 ACL 的初始化参数、填写 table 的初始化参数、创建 classifer 对象。
3. 读取 ipv4\_rules\_file.txt 这个文件，文件中一行是一个规则，一行的内容是一个ipv4的五元组。如果符合输入的合法性验证要求，就把里面的内容，提成特定的数据结构，插入到 classifer 里。2、3两步过程中封装了多层，还涉及非常多的数据结构和API。不容易搞懂。（**其实也不需要完全搞懂**，我后面有说，继续往下看）
4. 添加完规则后进入lcore\_main主线程，死循环收包（参照basicfw）。每次收上来的一堆包，就对 classifier 里的每条规则进行都 query，用到DPDK的API。如果其中有符合规则的packet（也就是query rule 匹配），就会在对应 rule 的 counter 加 1 并显示 counter 的数字（匹配成功次数），失败的话就显示“没有匹配到这条规则”的提示语句。然后不论匹配是否成功，都把这批包从另一个端口转发了。

我们可以看看 ipv4\_rules\_file.txt 这个文件的内容：

#src\_ip/masklen dst\_ip/masklen src\_port : mask dst\_port : mask proto/mask priority

#

2.2.2.3/24 2.2.2.7/24 32 : 0xffff 33 : 0xffff 17/0xff 0

9.9.9.3/24 9.9.9.7/24 32 : 0xffff 33 : 0xffff 17/0xff 1

9.9.9.3/24 9.9.9.7/24 32 : 0xffff 33 : 0xffff 6/0xff 2

9.9.8.3/24 9.9.8.7/24 32 : 0xffff 33 : 0xffff 6/0xff 3

6.7.8.9/24 2.3.4.5/24 32 : 0x0000 33 : 0x0000 132/0xff 4

6.7.8.9/32 192.168.0.36/32 10 : 0xffff 11 : 0xffff 6/0xfe 5

6.7.8.9/24 192.168.0.36/24 10 : 0xffff 11 : 0xffff 6/0xfe 6

6.7.8.9/16 192.168.0.36/16 10 : 0xffff 11 : 0xffff 6/0xfe 7

6.7.8.9/8 192.168.0.36/8 10 : 0xffff 11 : 0xffff 6/0xfe 8

可以看到，DPDK 在 classify flow 中对 flow 的定义是根据 IPv4 的**五元组 + 优先级**来的，优先级就是如果有一个包同时满足了多条规则，则匹配的是优先级最高的那一条。

综上所述，这个flow\_classify 的程序的功能就是首先，在文件ipv4\_rules\_file.txt 中预设一些五元组 + 优先级的 rules，然后运行这个程序。在网口收包时，如果收到了满足某条 rule 的流，则会提示并在相对应的 rule 上计数。由于代码太复杂，所以我们**不需要**对代码进行修改或自行编程，**只需修改ipv4\_rules\_file.txt 这个文件的内容后，运行自带的程序即可**。DPDK还有一个 sample 叫做flow\_filtering，我猜想大部分程序内容应该会和flow\_classify是相似的，区别会体现在lcore\_main主线程中，flow\_filtering会把不满足流规则的包丢弃。

**运行情况**

root@ubuntu:/home/chang/dpdk/examples/flow\_classify/build# ./flow\_classify -c 1 -n 4 -- --rule\_ipv4="../ipv4\_rules\_file.txt"

EAL: Detected 8 lcore(s)

EAL: No free hugepages reported in hugepages-1048576kB

EAL: Multi-process socket /var/run/.rte\_unix

EAL: Probing VFIO support...

EAL: PCI device 0000:02:01.0 on NUMA socket -1

EAL: Invalid NUMA socket, default to 0

EAL: probe driver: 8086:100f net\_e1000\_em

EAL: PCI device 0000:02:02.0 on NUMA socket -1

EAL: Invalid NUMA socket, default to 0

EAL: probe driver: 8086:100f net\_e1000\_em

EAL: PCI device 0000:02:03.0 on NUMA socket -1

EAL: Invalid NUMA socket, default to 0

EAL: probe driver: 8086:100f net\_e1000\_em

EAL: PCI device 0000:02:04.0 on NUMA socket -1

EAL: Invalid NUMA socket, default to 0

EAL: probe driver: 8086:100f net\_e1000\_em

Port 0 MAC: 00 0c 29 f7 4d 25

Port 1 MAC: 00 0c 29 f7 4d 2f

table\_entry\_delete succeeded [7]

Core 0 forwarding packets. [Ctrl+C to quit]

rule [0] query failed ret [-22]

rule [1] query failed ret [-22]

rule [2] query failed ret [-22]

rule [3] query failed ret [-22]

rule [4] query failed ret [-22]

rule [5] query failed ret [-22]

rule [6] query failed ret [-22]

rule [7] query failed ret [-22]

rule [8] query failed ret [-22]

rule [0] query failed ret [-22]

rule [1] query failed ret [-22]

rule [2] query failed ret [-22]

rule [3] query failed ret [-22]

rule [4] query failed ret [-22]

rule [5] query failed ret [-22]

rule [6] query failed ret [-22]

rule [7] query failed ret [-22]

rule [8] query failed ret [-22]

rule [0] query failed ret [-22]

rule [1] query failed ret [-22]

rule [2] query failed ret [-22]

rule [3] query failed ret [-22]

rule [4] query failed ret [-22]

rule [5] query failed ret [-22]

rule [6] query failed ret [-22]

rule [7] query failed ret [-22]

rule [8] query failed ret [-22]

我没有改动原来自带的规则文件，因此不会有匹配成功，提示的都是匹配失败。下一步的思路可以是熟悉pktgen等发包工具的使用，发出特定五元组的包，并在规则文件中修改，使其匹配。

**reference**

* 他人的文章：
  + [丹西 - DPDK报文分类与访问控制](https://www.cnblogs.com/danxi/p/6650757.html)
  + [cumirror - DPDK ACL算法介绍](https://www.jianshu.com/p/0f71f814d73e)
  + [Generic Flow API简介](http://www.sysight.com/index.php?qa=471&qa_1=generic-flow-api%E7%AE%80%E4%BB%8B)
* 文档：
  + API doc 中的 flow\_classify、[rte\_flow.h](http://doc.dpdk.org/api/rte__flow_8h_source.html)
* programmer's guides 中的
  + [9. Generic flow API](http://doc.dpdk.org/guides/prog_guide/rte_flow.html) 有 Rules 的详细信息。
  + ※[25. Flow Classification Library](https://doc.dpdk.org/guides/prog_guide/flow_classify_lib.html) 介绍了创建 classifer、添加规则、parsing、query 的过程。
  + [41. Packet Classification and Access Control](https://doc.dpdk.org/guides/prog_guide/packet_classif_access_ctrl.html)，介绍了 ACL 与 rule 的关系。
* sample guieds 中的
  + [9. Flow Classify Sample Application](https://doc.dpdk.org/guides/sample_app_ug/flow_classify.html)

分类: [DPDK](https://www.cnblogs.com/ZCplayground/category/1181813.html)