<https://www.cnblogs.com/danxi/p/6650757.html>

dpdk提供了一个访问控制库，提供了基于一系列分类规则对接收到的报文进行分类的能力。  
ACL库用来在一系列规则上执行N元组查找，可以实现多个分类和对每个分类查找最佳匹配（最高优先级），ACL库的api提供如下基本操作：

* 创建一个新的访问控制（AC）环境实例（context）
* 添加规则到这个环境实例
* 为这个实例里所有的规则，创建必需的运行时结构体来指针报文分类
* 执行接收报文分类
* 删除AC环境实例和对应的运行时结构体，并释放内存

**概述**  
1.规则定义  
当前的实现允许用户对将要执行的报文分类需要的每一个context指定它独有规则（字段集合）。但这在规则字段上有一些限制条件：  
    规则定义的第一个字段必须是一个字节的长度  
    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。  
要定义规则的每一个字段，需要使用如下的结构体：

[复制代码](javascript:void(0);)

1 struct rte\_acl\_field\_def {

2 uint8\_t type; /\*< type - ACL\_FIELD\_TYPE. \*/

3 uint8\_t size; /\*< size of field 1,2,4, or 8. \*/

4 uint8\_t field\_index; /\*< index of field inside the rule. \*/

5 uint8\_t input\_index; /\*< 0-N input index. \*/

6 uint32\_t offset; /\*< offset to start of field. \*/

7 };

[复制代码](javascript:void(0);)

type   
字段的类型，有3种选项：  
    \_MASK    表示有值和掩码的IP地址字段，定义相关的bit位  
    \_RANGE   表示端口字段的低位和高位值  
    \_BITMASK 表示协议标识字段的值和掩码位  
  
size 这个参数定义了字段的字节数大小。允许的值范围有（1,2,4,8）bytes，注意，由于输入字节的分组，1或2字节的字段必须定义为连续的来组成4字节连续。通用，最好的做法是定义8或更多字节数的字段，这样构建进程会消除那些乱的字段。  
  
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。  
  
input\_index  
上面提到过，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。  
  
offset  
这个定义了字段的偏移量，为查找指定了从缓冲区的起始位置的偏移。  
  
举个栗子，定义一个IPv4的五元组的分类：

[复制代码](javascript:void(0);)

1 struct ipv4\_5tuple {

2 uint8\_t proto;

3 uint32\_t ip\_src;

4 uint32\_t ip\_dst;

5 uint16\_t port\_src;

6 uint16\_t port\_dst;

7 };

[复制代码](javascript:void(0);)

需要使用下面的字段定义数组：

[复制代码](javascript:void(0);)

1 struct rte\_acl\_field\_def ipv4\_defs[5] = {

2 /\* first input field - always one byte long. \*/

3 {

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

5 .size = sizeof (uint8\_t),

6 .field\_index = 0,

7 .input\_index = 0,

8 .offset = offsetof (struct ipv4\_5tuple, proto),

9 },

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

11 {

12 .type = RTE\_ACL\_FIELD\_TYPE\_MASK,

13 .size = sizeof (uint32\_t),

14 .field\_index = 1,

15 .input\_index = 1,

16 .offset = offsetof (struct ipv4\_5tuple, ip\_src),

17 },

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

19 {

20 .type = RTE\_ACL\_FIELD\_TYPE\_MASK,

21 .size = sizeof (uint32\_t),

22 .field\_index = 2,

23 .input\_index = 2,

24 .offset = offsetof (struct ipv4\_5tuple, ip\_dst),

25 },

26 /\*

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

28 \* They share the same input index.

29 \*/

30 {

31 .type = RTE\_ACL\_FIELD\_TYPE\_RANGE,

32 .size = sizeof (uint16\_t),

33 .field\_index = 3,

34 .input\_index = 3,

35 .offset = offsetof (struct ipv4\_5tuple, port\_src),

36 },

37 {

38 .type = RTE\_ACL\_FIELD\_TYPE\_RANGE,

39 .size = sizeof (uint16\_t),

40 .field\_index = 4,

41 .input\_index = 3,

42 .offset = offsetof (struct ipv4\_5tuple, port\_dst),

43 },

44 };

[复制代码](javascript:void(0);)

一个典型的IPv4五元组规则如下:

source addr/mask destination addr/mask source ports dest ports protocol/mask

192.168.1.0/24 192.168.2.31/32 0:65535 1234:1234 17/0xff

任何带有协议ID为17（UDP），源地址为192.168.1.[0-255]，目的地址为192.168.2.31，源端口在[0-65535]，目的端口为1234的ipv4报文将会匹配上面的规则。  
  
定义IPv6 2元组: <protocol, IPv6 source address>的报文分类，   
IPv6 头:

[复制代码](javascript:void(0);)

1 struct struct ipv6\_hdr {

2 uint32\_t vtc\_flow; /\* IP version, traffic class & flow label. \*/

3 uint16\_t payload\_len; /\* IP packet length - includes sizeof(ip\_header). \*/

4 uint8\_t proto; /\* Protocol, next header. \*/

5 uint8\_t hop\_limits; /\* Hop limits. \*/

6 uint8\_t src\_addr[16]; /\* IP address of source host. \*/

7 uint8\_t dst\_addr[16]; /\* IP address of destination host(s). \*/

8 } \_\_attribute\_\_((\_\_packed\_\_));

[复制代码](javascript:void(0);)

需要使用下面的字段定义数组:

[复制代码](javascript:void(0);)

1 struct struct rte\_acl\_field\_def ipv6\_2tuple\_defs[5] = {

2 {

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

4 .size = sizeof (uint8\_t),

5 .field\_index = 0,

6 .input\_index = 0,

7 .offset = offsetof (struct ipv6\_hdr, proto),

8 },

9 {

10 .type = RTE\_ACL\_FIELD\_TYPE\_MASK,

11 .size = sizeof (uint32\_t),

12 .field\_index = 1,

13 .input\_index = 1,

14 .offset = offsetof (struct ipv6\_hdr, src\_addr[0]),

15 },

16 {

17 .type = RTE\_ACL\_FIELD\_TYPE\_MASK,

18 .size = sizeof (uint32\_t),

19 .field\_index = 2,

20 .input\_index = 2,

21 .offset = offsetof (struct ipv6\_hdr, src\_addr[4]),

22 },

23 {

24 .type = RTE\_ACL\_FIELD\_TYPE\_MASK,

25 .size = sizeof (uint32\_t),

26 .field\_index = 3,

27 .input\_index = 3,

28 .offset = offsetof (struct ipv6\_hdr, src\_addr[8]),

29 },

30 {

31 .type = RTE\_ACL\_FIELD\_TYPE\_MASK,

32 .size = sizeof (uint32\_t),

33 .field\_index = 4,

34 .input\_index = 4,

35 .offset = offsetof (struct ipv6\_hdr, src\_addr[12]),

36 },

37 };

[复制代码](javascript:void(0);)

一个典型的IPv4二元组规则如下:

source addr/mask protocol/mask

2001:db8:1234:0000:0000:0000:0000:0000/48 6/0xff

任何带有协议ID为6 (TCP),源地址在  
[2001:db8:1234:0000:0000:0000:0000:0000 - 2001:db8:1234:ffff:ffff:ffff:ffff:ffff] 之间的报文会匹配上的规则。  
  
下面的例子，查找Key最后的元素是8bit长，这样就会引起4字节连续的问题：

1 struct acl\_key {

2 uint8\_t ip\_proto;

3 uint32\_t ip\_src;

4 uint32\_t ip\_dst;

5 uint8\_t tos; /\*< This is partially using a 32-bit input element \*/

6 };

如下定义下面的字段定义数组：

[复制代码](javascript:void(0);)

1 struct rte\_acl\_field\_def ipv4\_defs[4] = {

2 /\* first input field - always one byte long. \*/

3 {

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

5 .size = sizeof (uint8\_t),

6 .field\_index = 0,

7 .input\_index = 0,

8 .offset = offsetof (struct acl\_key, ip\_proto),

9 },

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

11 {

12 .type = RTE\_ACL\_FIELD\_TYPE\_MASK,

13 .size = sizeof (uint32\_t),

14 .field\_index = 1,

15 .input\_index = 1,

16 .offset = offsetof (struct acl\_key, ip\_src),

17 },

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

19 {

20 .type = RTE\_ACL\_FIELD\_TYPE\_MASK,

21 .size = sizeof (uint32\_t),

22 .field\_index = 2,

23 .input\_index = 2,

24 .offset = offsetof (struct acl\_key, ip\_dst),

25 },

26 /\*

27 \* Next element of search key (Type of Service) is indeed 1 byte long.

28 \* Anyway we need to allocate all the 4 consecutive bytes for it.

29 \*/

30 {

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

32 .size = sizeof (uint32\_t), /\* All the 4 consecutive bytes are allocated \*/

33 .field\_index = 3,

34 .input\_index = 3,

35 .offset = offsetof (struct acl\_key, tos),

36 },

37 };

[复制代码](javascript:void(0);)

下面是一个典型的IPv4四元组规则：

source addr/mask destination addr/mask tos/mask protocol/mask

192.168.1.0/24 192.168.2.31/32 1/0xff 6/0xff

任何带有协议ID 6 (TCP), 源地址 192.168.1.[0-255], 目的地址 192.168.2.31, ToS 1的IPv4报文就会匹配这个规则.  
  
当创建一个规则集合时，对每一个规则，也必须添加的附加信息：  
priority  
用来度量规则的权重，越高越好。如果输入的元组匹配到多个元组，那么就会使用权重最高的那条规则。如果匹配多个权重相同的规则，那么返回那条规则这个是未定义的。推荐的做法是，每个规则都设置一个唯一的权重。  
  
category\_mask  
每个规则都使用的一个掩码位，用来选择规则的分类。当执行查找时，查找结果里包含了每一个分类。这样在使能了一个查找返回多个结果的情况，有效的支持了并行查找。举个栗子，有4个不同的ACL规则集合，一个用于访问控制，一个用于路由，等等。每一个集合有自己的分类，并组织在一个数据库里，一个查找就可以返回包含4个集合的结果。  
  
userdata  
一个用户定义的字段，该字段可以设置任何值除了0。对每一个分类，一个成功的查找返回匹配最高优先级的规则的用户数据。  
  
注意：添加新规则到ACL环境中是，所有的字段都必须是主机字节序，当对输入的元组执行查找时，元组的所有字段都必须是网络字节序。  
  
2.RT内存大小限制  
rte\_acl\_build()创建了一个给定规则集合的内部数据结构，用来给将来的运行时遍历。当前的实现是一个multi-bit tries（字典树，步长等于8）。根据这个规则集合的规模，可能会消耗大量的内存。如果以固定的空间创建ACL字典树，同时把给定的规则集合分割成数个无关的子集并建立各自的字典树的话，同样取决于规则集合的规模，它会减少RT内存的大小要求但是会增加报文分类的时间。  
在创建AC环境的时候，可以指定内部RT结构的最大内存数量限制。通过rte\_acl\_config的max\_size字段来设置。设置一个大于0的值，在rte\_acl\_build()里：  
    尝试在RT表里最小化字典树的数目，but  
    确保RT表的大小不会超过给定的值。  
      
设置为0的话，rte\_acl\_build()会默认是：尽量以最小大小来创建RT数据，但不做任何限制。  
下面为用户提供了一个权衡性能与空间的方式，举个栗子：

[复制代码](javascript:void(0);)

1 struct rte\_acl\_ctx \* acx;

2 struct rte\_acl\_config cfg;

3 int ret;

4 /\*

5 \* assuming that acx points to already created and

6 \* populated with rules AC context and cfg filled properly.

7 \*/

8 /\* try to build AC context, with RT structures less then 8MB. \*/

9 cfg.max\_size = 0x800000;

10 ret = rte\_acl\_build(acx, &cfg);

11 /\*

12 \* RT structures can't fit into 8MB for given context.

13 \* Try to build without exposing any hard limit.

14 \*/

15 if (ret == -ERANGE) {

16 cfg.max\_size = 0;

17 ret = rte\_acl\_build(acx, &cfg);

18 }

[复制代码](javascript:void(0);)

3.报文分类方法  
rte\_acl\_build()成功之后，将开始执行报文分类，对输入的数据查找最高优先级的规则。下面有几个分类算法的实现：  
RTE\_ACL\_CLASSIFY\_SCALAR 通用实现，不需要任何硬件实现。  
RTE\_ACL\_CLASSIFY\_SSE 向量实现，能并行处理到8条流，要求SS4 4.1支持。  
RTE\_ACL\_CLASSIFY\_AVX2 向量实现，能并行处理到16条流，要求AVX2支持。  
  
这个完全由运行时决定选择哪个算法，在创建的时候并没有什么不同。所有的算法都使用相同的RT数据结构和相同的原理。主要的不同是向量实现能手动利用IA SIMD指令和并行处理数据流。在启动ACL库之后会基于当前的平台来决定使用哪个最有效的分类方法，并把它当成默认设置。但用户可以重写这个默认的分类方法，这要求用户自己实现当前的平台选择的分类方法。  
  
4.API使用举例：  
温馨提示：如果想了解ACL API的更多细节，请参考DPDK API手册。  
下面是一个IPV4五元组报文分类使用多个分类的规则的例子（Classify with Multiple Categories）：

[复制代码](javascript:void(0);)

1 struct rte\_acl\_ctx \* acx;

2 struct rte\_acl\_config cfg;

3 int ret;

4 /\* define a structure for the rule with up to 5 fields. \*/

5 RTE\_ACL\_RULE\_DEF(acl\_ipv4\_rule, RTE\_DIM(ipv4\_defs));

6 /\* AC context creation parameters. \*/

7 struct rte\_acl\_param prm = {

8 .name = "ACL\_example",

9 .socket\_id = SOCKET\_ID\_ANY,

10 .rule\_size = RTE\_ACL\_RULE\_SZ(RTE\_DIM(ipv4\_defs)),

11 /\* number of fields per rule. \*/

12 .max\_rule\_num = 8, /\* maximum number of rules in the AC context. \*/

13 };

14

15 struct acl\_ipv4\_rule acl\_rules[] = {

16 /\* matches all packets traveling to 192.168.0.0/16, applies for categories: 0,1 \*

17 {

18 .data = {.userdata = 1, .category\_mask = 3, .priority = 1},

19 /\* destination IPv4 \*/

20 .field[2] = {.value.u32 = IPv4(192,168,0,0),. mask\_range.u32 = 16,},

21 /\* source port \*/

22 .field[3] = {.value.u16 = 0, .mask\_range.u16 = 0xffff,},

23 /\* destination port \*/

24 .field[4] = {.value.u16 = 0, .mask\_range.u16 = 0xffff,},

25 },

26 /\* matches all packets traveling to 192.168.1.0/24, applies for categories: 0 \*/

27 {

28 .data = {.userdata = 2, .category\_mask = 1, .priority = 2},

29 /\* destination IPv4 \*/

30 .field[2] = {.value.u32 = IPv4(192,168,1,0),. mask\_range.u32 = 24,},

31 /\* source port \*/

32 .field[3] = {.value.u16 = 0, .mask\_range.u16 = 0xffff,},

33 /\* destination port \*/

34 .field[4] = {.value.u16 = 0, .mask\_range.u16 = 0xffff,},

35 },

36 /\* matches all packets traveling from 10.1.1.1, applies for categories: 1 \*/

37 {

38 .data = {.userdata = 3, .category\_mask = 2, .priority = 3},

39 /\* source IPv4 \*/

40 .field[1] = {.value.u32 = IPv4(10,1,1,1),. mask\_range.u32 = 32,},

41 /\* source port \*/

42 .field[3] = {.value.u16 = 0, .mask\_range.u16 = 0xffff,},

43 /\* destination port \*/

44 .field[4] = {.value.u16 = 0, .mask\_range.u16 = 0xffff,},

45 },

46 };

47

48 /\* create an empty AC context \*/

49 if ((acx = rte\_acl\_create(&prm)) == NULL) {

50 /\* handle context create failure. \*/

51 }

52 /\* add rules to the context \*/

53 ret = rte\_acl\_add\_rules(acx, acl\_rules, RTE\_DIM(acl\_rules));

54 if (ret != 0) {

55 /\* handle error at adding ACL rules. \*/

56 }

57 /\* prepare AC build config. \*/

58 cfg.num\_categories = 2;

59 cfg.num\_fields = RTE\_DIM(ipv4\_defs);

60 memcpy(cfg.defs, ipv4\_defs, sizeof (ipv4\_defs));

61 /\* build the runtime structures for added rules, with 2 categories. \*/

62 ret = rte\_acl\_build(acx, &cfg);

63 if (ret != 0) {

64 /\* handle error at build runtime structures for ACL context. \*/

65 }

[复制代码](javascript:void(0);)

对于如下的元组源IP地址：10.1.1.1，目的地址：192.168.1.15，一旦下面语句执行：

1 uint32\_t results[4]; /\* make classify for 4 categories. \*/

2 rte\_acl\_classify(acx, data, results, 1, 4);

那么返回的结果数组里会如下：  
results[4] = {2, 3, 0, 0};

对于分类0，规则1和规则2都匹配了，但是规则2有更高的优先级，因此result[0]包含了规则2的userdata。  
对于分类1，规则1和规则3都匹配了，但是规则3有更高的优先级，因此result[1]包含了规则3的userdata。  
对于分类2,3，都没有匹配，因此result[2],results[3]都是0，表示没有匹配到这些分类。  
  
对于如下的元组源IP地址：192.168.1.1，目的地址：192.168.2.11，一旦下面语句执行：

1 uint32\_t results[4]; /\* make classify for 4 categories. \*/

2 rte\_acl\_classify(acx, data, results, 1, 4);

那么返回的结果数组里会如下：  
results[4] = {1, 1, 0, 0};  
  
对于分类0,1，只有规则1匹配；  
对于分类2,3，都没有匹配。  
  
对于如下的元组源IP地址：10.1.1.1，目的地址：202.212.111.12，一旦下面语句执行：

1 uint32\_t results[4]; /\* make classify for 4 categories. \*/

2 rte\_acl\_classify(acx, data, results, 1, 4);

那么返回的结果数组里会如下：  
results[4] = {0, 3, 0, 0};  
  
对于分类1，只有规则3匹配；  
对于分类0,2,3，都没有匹配。

s