此文章已于 14:42:40 2016/8/4 发布到 MADHEX

BCM芯片FP原理及相关SDK数据结构介绍

作者: 北京—小武

邮箱：[night\_elf1020@163.com](mailto:night_elf1020@163.com)

新浪微博：北京-小武

BCM芯片有几个大的模块： VLAN、L2、L3和FP等几个，其中FP的使用也最为灵活，能解析匹配数据包文的前128字节比特级的内容，动作包括转发、丢弃、结合qos修改相应字段、分配vid、流镜像、流重定向、指定端口转发（比如CPU口）、指定下一跳转发往、指定隧道转发等，往往在实现功能上有意想不到的功效。简单来说，如果硬件和BSP分别是九阳真经和九阴真经的话，那么port和vlan是少林七十二项绝技的组合，L2转发则是显得有点悠闲的峨眉派功夫，当然L3则是以太极拳为代表的武当派功夫，那么FP可以是以乾坤大挪移、吸星大法等为代表的魔教的邪而又邪的“旁门左道”，当然其他功能是零零散散的其他门派功夫。能够灵活运用好FP是增加很多交换机新功能的一种常用的手段。本文总结下FP这个模块BCM在硬件上的实现原理及SDK的相关数据结构。因为FP在实现功能上的灵活性，在此希望能抛砖引玉，激发大家更多的应用FP实现新功能的火花。

**BCM芯片FP实现原理**

FP的全称是Fields Processors，也称为ContentAwareProcess（CAP），在BCM较早的芯片称为Fast Filter Processors（FFP），和现在的FP相比有一些原理不同，不过现在交换芯片已经不再使用FFP，所以在此也不再介绍。FP本质来说，是一组相互之间有关联的表，一起通过查找、匹配等来决定对报文施加的动作；在BCM芯片交换机中，有三种查找查找方式：hash，index，tcam。FP的查找主要用到了index和tcam，其中CAM的全称是ContentAddressable Memory，中文是内容寻址器，TCAM则是Ternary ContentAddressable Memory，中文称为三态内存寻址器，TCAM的实现是通过对应比特位+掩码产生三种匹配方式：掩码为0表示不关心、掩码为1且对应位为1或掩码为1且对应位为0。 这就是三态的具体含义。

在我们自研交换机所用的芯片中有三个FP：VFP(VLAN FP)、IFP（ingres FP）和EFP（Egress FP），另外在四代芯片kylin卡中曾出现外扩FP，称为E-IFP，表项大小为128K，为L2和L3转发用，有点openflow的意味。其中VFP主要用于对报文tag的处理，比如添删或修改vid的灵活QINQ的实现就基于此FP；IFP的用途比较多，主要是对进入端口后的报文进行处理，主要有入口acl、流重定向、流镜像、设置下一跳、为qos数据报文分类等用途；EFP的用途和IFP类似，但是因为EFP是报文在转出前在出端口进行处理的规则，IFP有的动作类型在EFP不太适用。虽然三种FP用途和数据包流经顺序不太一样，但是硬件原理是一致的。下面介绍下FP的硬件原理。

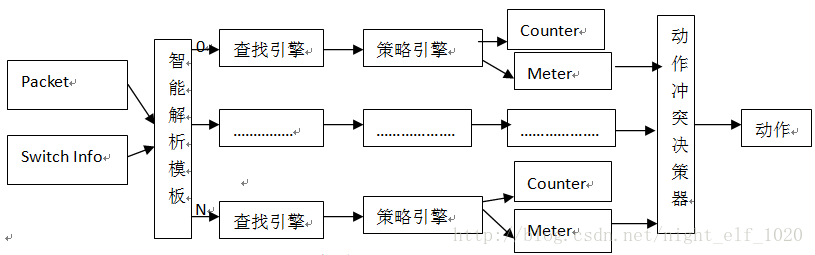


图1 FP原理组成图

图1中，每一个查找引擎和策略引擎及后面的counter资源和meter资源组合成一个规则组，芯片称之为一个slice，从图1可以看出，FP的实现有五部分组成：

智能解析模板：主要将报文信息（最多报文前128字节，可以精确到每一位bit）根据对每个slice的care字段将各对应字段解析出来，再加上前面L2、L3的转发信息，一起送给每个slice的查找引擎去匹配；

查找引擎：将解析出来的字段按照TCAM方式去查找本slice的规则是否有匹配的，即HIT的，只要有一条hit的即刻返回这条规则的index不再继续查找本slice后面规则，后面即使还有匹配的规则；这样做就是为了保证一个slice内部规则的优先级；如果没有匹配说明此slice没有匹配的规则或根本就没有规则，后面的流程也无需再走；

策略引擎：根据查找引擎得到的index直接索引策略引擎的动作，动作类型有转发、丢弃、重定向（包括到CPU口且可指定队列）、流镜像（包括到CPU口且可指定队列）、修改报文特定的字段（COS、DSCP、EXP等）、与后面的meter一起对报文染色并对不同染色报文指定相应动作、指定下一跳、指定ECMP、指定TTL是否修改、指定URPF的模式等相关动作；需要说明的是，一条规可以对报文执行多种动作，当然需要报文动作之间是不冲突的，即slice规则的动作冲突是靠配置下发来检查的，同一条规则有冲突的动作无法下发硬件；

Counter和meter资源： counter资源用于计数，有基于byte和packet的两种方式；meter主要用于测速，然后根据速度对报文进行染色（绿、黄和红）然后对报文应用不同的QOS策略；meter的工作原理可以参见我原先写的有关令牌桶相关文档。

动作冲突决策引擎：前面说过，一条slice的动作冲突是靠配置下发检查来实现的，冲突的动作无法同时下发到硬件；但是FP通常有多个slice，每个slice都有规则被匹配且动作时间有冲突时，需要动作冲突决策引擎来处理到底执行哪一个规则的动作，如果多个动作不冲突都执行；原则是丢弃、重定向等优先级最高，其他时候看slice号（这个slice号有的芯片只支持是物理的，高级芯片支持虚拟slice号），slice号越大优先级越高；

我们一条规则的匹配报文长度信息是有限的，对于IPV4报文同时匹配SMAC、DMAC、SIP、DIP等信息的时候，就不够了，芯片提供了将两条规则合并成一条规则，组成更大长度规则的方法，主要有图2示的两种,：

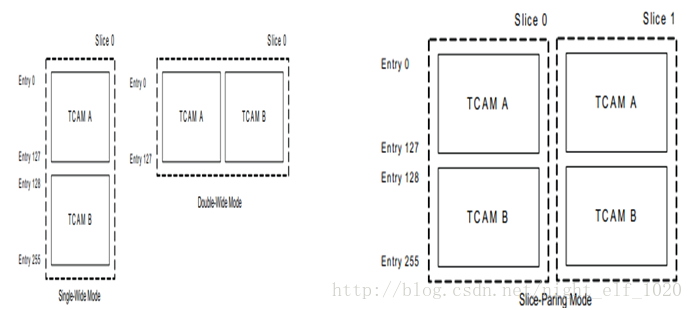


                      图2 两种slice宽模式

第一种是将一条slice的规则分为前后两部分，然后进行如图2左边的方式拼成double模式，这种模式称为double wide模式；第二种是用两条slice，直接如图2右边所示的方式拼成double模式，这种拼接方式称之为slice-paring模式。这两种模式，有的低级芯片都不支持，只能用单倍模式，有的芯片支持其中一种，我们的redstone交换机就只支持左边的这种方式。还有的芯片可以同时支持这两种拼接方式，那么就可以利用这点拼接处具有更大长度信息的四倍模式：

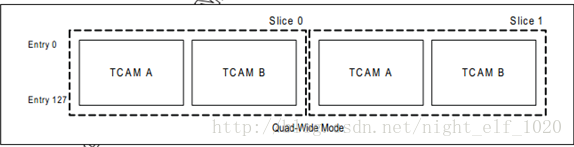


                       图3 四倍key模式

这种模式常用于IPV6报文的匹配中，因为IPV6的SIP和DIP实在太长了，再加上匹配其他信息，只能用四倍模式才能完全覆盖所有字段。但是我们的redstone交换机只支持slice-pairng模式，所以在IPV6报文的匹配中需要做折中。

前面我们提到slice有物理slice和虚拟slice，这个与物理内存和虚拟内存有点类似，FP都有物理slice，在高级的芯片上，为了更好的解决slice之间的动作冲突，对slice进行了虚拟编号，虚拟slice号越大优先级越高，这样就可以实现动作的优先级指定；可能做过物理slice的同学能体会为了保证各种应用slice的优先级在软件处理所做的代码处理工作有多么的艰辛；硬件进步这么一步，支持虚拟slice后，这部分工作就完全交给硬件来处理了，我们只需要指定优先级高低就可以了。而且虚拟slice还支持虚拟slice组的概念，每一个虚拟slice组就像一条slice一样，只会有一个动作产生出，这样就又大大减少了动作冲突的机会，而且还能使得每种应用使用更多的slice资源，无需考虑因为物理slice带来的动作优先级打破应用的优先级，更符合实际。

**BCM对FP操作的接口**

BCM的SDK提供了四套对于FP资源使用和管理的函数接口，需要视具体应用环境和个人喜好来定夺，四种接口如下：

SOC API：直接硬件表项或寄存器操作，BCM各种问题明确不提倡的接口，因为需要配置人员管理和组织大量的逻辑；

Bcmx接口：通常不被使用的接口，因为不太灵活，且SDK被改造成为所有ACL规则为一个大的group，现在暂时IFP只有协议规则和ACL使用，所以还勉强满足需求，以lport作为端口的配置参数；但是每次下发新规则都要先删除原来规则，这个是没有必要的；这套接口和下面BCM接口的区别不是很大。相关函数接口有：

bcmx\_field\_group\_create

bcmx\_field\_group\_create\_id

bcmx\_field\_group\_compress

bcmx\_field\_group\_install

bcmx\_field\_group\_remove

bcmx\_field\_group\_destroy

bcmx\_field\_entry\_create

bcmx\_field\_entry\_destroy

bcmx\_field\_entry\_destroy\_all

bcmx\_field\_data\_qualifier\_destroy

bcmx\_field\_data\_qualifier\_destroy\_all

bcmx\_field\_qualify\_clear

bcmx\_field\_data\_qualifier\_\*\*\_add

bcmx\_field\_data\_qualifier\_\*\*\_ delete等。

Bcm接口：BCM中对FP操作的最灵活的一组接口，非常适合运营商多种应用的场合，这组接口传递的参数也非常详细；相关函数接口有：

bcm\_field\_group\_create

bcm\_field\_group\_create\_id

bcm\_field\_group\_priority\_set

bcm\_field\_group\_compress

bcm\_field\_group\_install

bcm\_field\_group\_remove

bcm\_field\_group\_destroy

bcm\_field\_entry\_create

bcm\_field\_entry\_create\_id

bcm\_field\_entry\_destroy

bcm\_field\_entry\_destroy\_all

bcm\_field\_entry\_reinstall

bcm\_field\_entry\_remove

bcm\_field\_qualify\_clea

bcm\_field\_qualify\_\*\*\*\*

bcm\_field\_action\_add

bcm\_field\_action\_delete等。

Bcma接口：这套接口称为AdvancedContentAware Enhanced Software (ACES) implementation，传递的参数为bcma\_acl\_t\*list，以结构体形式将规则所有参数下发到硬件；

/\* List Management functions \*/

extern int bcma\_acl\_add(bcma\_acl\_t\*list\_id);

extern int bcma\_acl\_remove(bcma\_acl\_list\_id\_tlist\_id);

extern int bcma\_acl\_get(bcma\_acl\_list\_id\_tlist\_id, bcma\_acl\_t \*list);

extern intbcma\_acl\_rule\_add(bcma\_acl\_list\_id\_t list\_id,

                             bcma\_acl\_rule\_t\*rule);

extern int bcma\_acl\_rule\_remove(bcma\_acl\_list\_id\_tlist\_id,

                               bcma\_acl\_rule\_id\_t rule\_id);

extern intbcma\_acl\_rule\_get(bcma\_acl\_rule\_id\_t rule\_id,

                             bcma\_acl\_rule\_t \*\*rule);

/\* Validation and Installation functions \*/

extern int bcma\_acl\_install(void);

extern int bcma\_acl\_uninstall(void); 等。

**SDK对FP资源管理的相关数据结构**

1.      BCM芯片每一个unit都有这么一个结构体来保存芯片所有FP的资源占用情况：

static \_field\_control\_t         \*\_field\_control[BCM\_MAX\_NUM\_UNITS];

field\_control\_t的具体内容为（每个变量都有详细注释，此处不再阐述）：

struct \_field\_control\_s {

sal\_mutex\_t           fc\_lock;       /\* Protectionmutex.             \*/

   bcm\_field\_stage\_t       stage;        /\* Default FP pipeline stage.   \*/

   int                   max\_stage\_id;  /\* Number of fpstages.          \*/

   \_field\_udf\_t           \*udf;          /\* field\_status->group\_total     \*/

   struct \_field\_group\_s \*groups;       /\* List of groupsin unit.       \*/

struct\_field\_stage\_s  \*stages;       /\* Pipeline stage FP info.

}

2.      然后对field\_control\_t中的\_field\_group\_s表示一种应用占用的slice和slice的规则记录：

\_field\_group\_s {

   bcm\_field\_group\_t      gid;            /\* Opaque handle.                \*/

   int                   priority;       /\* Field grouppriority.         \*/

   bcm\_field\_qset\_t       qset;           /\* This group's Qualifier Set.   \*/

   uint8                  flags;          /\* Group configuration flags.    \*/

   \_field\_slice\_t        \*slices;        /\* Pointer intoslice array.     \*/

   bcm\_pbmp\_t             pbmp;           /\* Ports in use this group.      \*/

   \_field\_sel\_t sel\_codes[\_FP\_MAX\_ENTRY\_WIDTH]; /\* Select codes forslice(s). \*/

   \_bcm\_field\_group\_qual\_t qual\_arr[\_FP\_MAX\_ENTRY\_WIDTH];

                                           /\* Qualifiers available in each

                                             individual entry part.        \*/

\_field\_stage\_id\_t      stage\_id;       /\* FP pipeline stage id.         \*/

}

3.      在每一个unit中还有\_field\_stage\_s来对各种FP（VFP/IFP/EFP）的资源记录的数据结构：

typedef struct \_field\_stage\_s {

   \_field\_stage\_id\_t     stage\_id;        /\* Pipeline stageid.           \*/

   uint8                  flags;           /\* Stage flags.                 \*/

   int                   tcam\_sz;         /\* Number ofentries in TCAM.   \*/

   int                   tcam\_slices;     /\* Number ofinternal slices.   \*/

struct\_field\_slice\_s  \*slices;         /\* Array of slices.\*/

}

4.      在在每一个\_field\_stage\_s中用\_field\_slice\_s对每一个slice资源进行记录的结构体：

\_field\_slice\_s {

   uint8                 slice\_number;  /\* Hardware slicenumber.         \*/

   int                   start\_tcam\_idx;/\* Slice first entry tcam index.  \*/

   int                   entry\_count;   /\* Number of entriesin the slice.\*/

   int                   free\_count;    /\* Number of freeentries.        \*/

   int                   counters\_count;/\* Number of counters accessible. \*/

   int                   meters\_count;  /\* Number of metersaccessible.   \*/

   \_field\_counter\_bmp\_t  counter\_bmp;   /\* Bitmap forcounter allocation. \*/

   \_field\_meter\_bmp\_t    meter\_bmp;     /\* Bitmap for meterallocation.   \*/

   \_field\_stage\_id\_t     stage\_id;      /\* Pipeline stageslice belongs.  \*/

   bcm\_pbmp\_t             pbmp;          /\* Ports in use by groups.        \*/

    struct \_field\_entry\_s \*\*entries;     /\* List of entriespointers.      \*/

}

5.      在在每一个\_field\_slice\_s中用\_field\_entry\_s对slice内部的entry进行记录：

struct\_field\_entry\_s {

    bcm\_field\_entry\_t      eid;       /\* BCM unit unique entryidentifier   \*/

    int                    prio;       /\* Entry priority                     \*/

    uint32                 slice\_idx;  /\* Field entry tcam index.            \*/

    uint16                 flags;      /\* \_FP\_ENTRY\_xxx flags                \*/

    \_field\_tcam\_t          tcam;       /\* Fields to be written intoFP\_TCAM  \*/

    \_field\_tcam\_t          extra\_tcam;

#ifdefined(BCM\_RAPTOR\_SUPPORT) || defined(BCM\_TRX\_SUPPORT)

    \_field\_pbmp\_t          pbmp;      /\* Port bitmap \*/

#endif /\*BCM\_RAPTOR\_SUPPORT || BCM\_TRX\_SUPPORT \*/

    \_field\_action\_t        \*actions;   /\* linked list of actions for entry   \*/

    \_field\_slice\_t         \*fs;        /\* Slice where entry lives            \*/

    \_field\_group\_t         \*group;     /\* Group where entry lives            \*/

    \_field\_entry\_stat\_t    statistic; /\* Statistics collection entity.     \*/

                                       /\*Policers attached to the entry.    \*/

    \_field\_entry\_policer\_tpolicer[\_FP\_POLICER\_LEVEL\_COUNT];

#ifdefined(BCM\_KATANA\_SUPPORT)

    \_field\_entry\_policer\_tglobal\_meter\_policer;

#endif

    struct \_field\_entry\_s  \*next;     /\* Entry lookup linked list.         \*/

};

上面actions 是一个\_field\_action\_t的结构体的链表，其信息为：

typedef struct\_field\_action\_s {

    bcm\_field\_action\_t     action;       /\* action type               \*/

    uint32                 param0;       /\* Action specific parameter \*/

    uint32                 param1;       /\* Action specific parameter \*/

    uint8                  inst\_flg;     /\* Installed Flag            \*/

    struct \_field\_action\_s \*next;

}\_field\_action\_t;

6.      在SDK编码中，用UNIT号获取对应的\_field\_control\_t信息的代码可以如下：

\_field\_control\_t    \*fc;

BCM\_IF\_ERROR\_RETURN(\_field\_control\_get(unit,&fc));

7.      进而获取每一个group资源的代码可以如下：

 \_field\_group\_t      \*fg;

 fg = fc->groups;

    while (fg != NULL) {

        if (fg->gid == gid) {

            \*group\_p = fg;

            return (BCM\_E\_NONE);

        }

        fg = fg->next;

    }

8.      获取每一个slice的资源可以如下

\_field\_slice\_t         \*slices;

slice =&fg->slices[0];

while(slice !=NULL){

slice = slice->prev;

}

9.      获取slice中规则的的资源可以如下：

\_field\_entry\_t  \*f\_ent;

 \_field\_action\_t  \*fa\_iter;

\_field\_entry\_get(unit, entry, \_FP\_ENTRY\_PRIMARY,&f\_ent);//entry

fa\_iter = f\_ent->actions;//entry的action

熟悉FP同学可能深知FP资源的稀缺性和重要性，可以用惜slice如黄金来做比喻；虽然FP的规则数很多，但是FP的资源申请和释放是按照slice为单位来进行的，且slice的数目一般都不是很多；所以我们要将尽量多的规则整合到一个slice里，尽量减少slice里有规则被浪费的现象；这个也是再将来的协议改造中必须考虑到的一个因素。

到这里对FP的原理和SDK的相关数据结构介绍到这里，如果描述中有不清晰或者不准确的地方欢迎随时沟通讨论。