交换机转发实验报告

李昊宸

2017K8009929044

（一）交换机转发实现

一、实验内容

1. 交换机学习实现：

查询操作：每收到一个数据包，根据目的MAC地址查询相应转发条目，如果查询到

对应条目，则根据相应转发端口转发数据包，更新访问时间；否则，广播该数据包

插入操作：每收到一个数据包，如果其源MAC地址在转发表中，更新访问时间；否

则，将该地址与入端口的映射关系写入转发表

老化操作：每秒钟运行一次老化操作，删除超过30秒未访问的转发条目

2. 使用iperf和给定的拓扑进行实验，对比交换机转发与集线器广播的性能从一个端

节点ping另一个端节点

3．思考题：

我们知道，网络中存在广播包，其目的MAC地址设置为全0xFF ，例如ARP请求数

据包。这种广播包对交换机行为逻辑有什么影响？

理论上，足够多个交换机可以连接起全世界所有的终端。请问，使用这种方式连接亿万台主机是否技术可行？并说明理由。

二、实验流程

1. 搭建实验环境

include：相关头文件

scripts：禁用TCP Offloading、IPV6功能，避免抓到无用包

main.c： Switch的代码实现，编译后在交换机结点上运行

mac.c： 实现的mac\_port\_mac相关操作：

iface\_info\_t \*lookup\_port(u8 mac[ETH\_ALEN]);

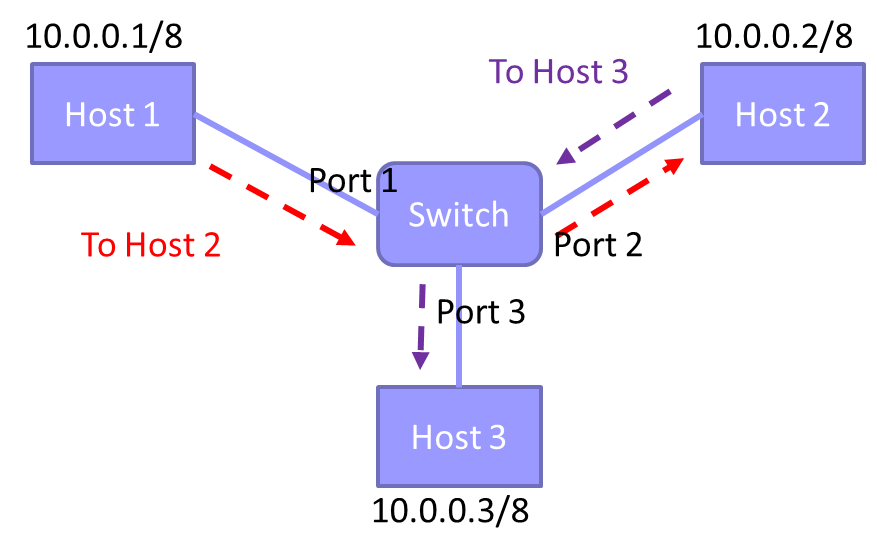
void insert\_mac\_port(u8 mac[ETH\_ALEN], iface\_info\_t \*iface);

int sweep\_aged\_mac\_port\_entry();

void broadcast\_packet(iface\_info\_t \*iface, const char \*packet, int len);

void handle\_packet(iface\_info\_t \*iface, char \*packet, int len);

three\_nodes\_bw.py：实现如下图的节点拓扑



图一 三节点网络拓扑

交换机的实现：

1）转发表的构造

|  |  |  |
| --- | --- | --- |
| **目的地址** | **转发端口** | **老化时间** |
| Host 1 MAC Addr | Port 1 | 30 sec |
| Host 2 MAC Addr | Port 2 | 30 sec |
| Host 3 MAC Addr | Port 3 | 30 sec |

图二 转发表

**转发表**用于存储目的地址和转发端口的映射关系，实际建立的转发表每一个条目还附带老化时间一项，用于表示转发表项的有效期。

交换机转发表的构造基于一个基本假设：如果收到的数据包的源MAC地址为X，端口为Y，那么将数据包从端口Y发送出去就可以到达X。

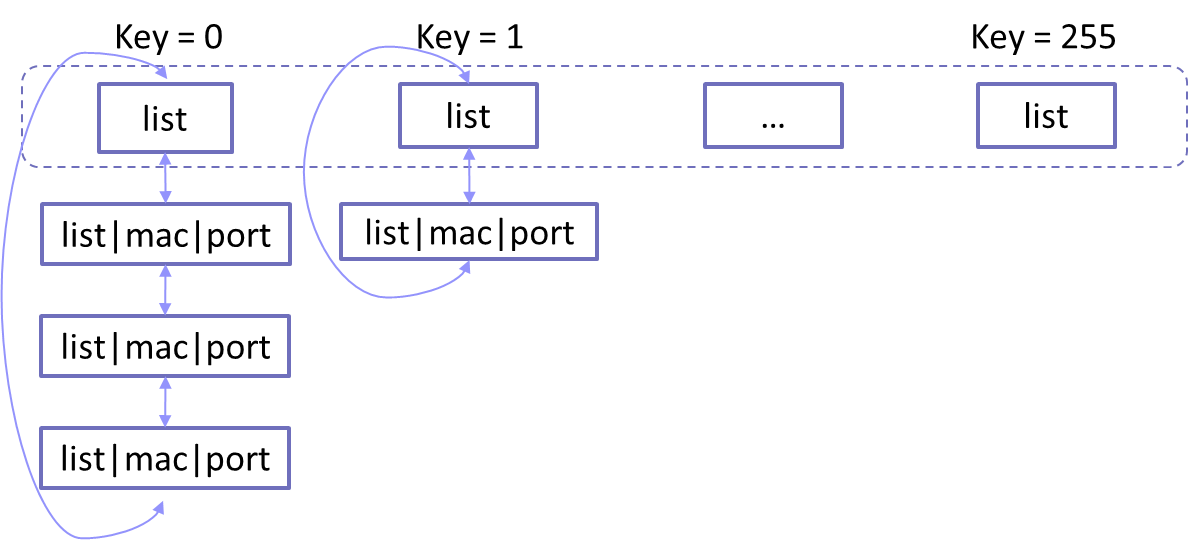
所以，转发表的插入操作为：每收到一个数据包，如果源MAC地址不在表中，就将该MAC地址与收到该数据包的端口的映射关系写入转发表；否则，就更新转发表中该项的访问时间（也即刷新老化时间）。老化操作每秒钟运行一次，删除超过30秒未访问的老化条目。

对于交换机而言，每收到一个数据包，都要根据目的MAC地址查询转发所需要的端口名字。如果该目的MAC地址在转发表中，就根据对应转发表项中的转发端口将该包发送出去；否则就广播该数据包。

2）转发表的查询

构造转发表一个比较自然的想法是使用一个链表。但是，查询链表的时间过长，最差情况下可能要遍历整个链表，造成大量时间的浪费。

所以，我们采用对MAC地址Hash，根据key值进入对应链表表项中查找。



图三 转发表的Hash结构

2.启动脚本

1)交换机转发效率测试

make all

sudo python three\_nodes\_bw.py

mininet> xterm h1 h1 h2 h3 s1

s1# ./switch

h2# iperf -s

h3# iperf -s

h1# iperf -c 10.0.0.2 -t 30

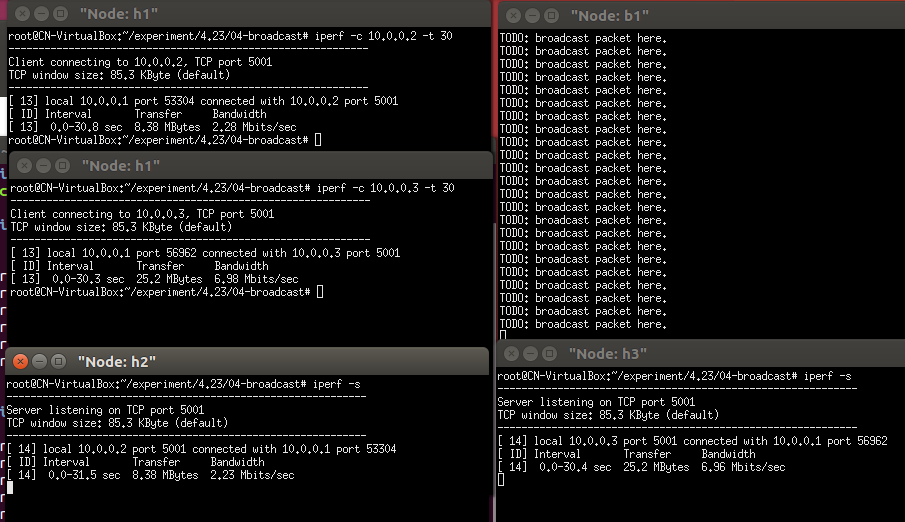
h1# iperf -c 10.0.0.3 -t 30

mininet> quit

上述过程是以h2、h3作为服务器，h1作为客户（通过启动两个终端）同时向二者进行

访问

三、实验结果及分析

1. 实验结果

图四 Hub的转发效率测试

我们先来回顾一下上周实验中广播网络的效率测试。

实验条件：h1：client，同时向h2和h3请求服务

h2：server

h3：server

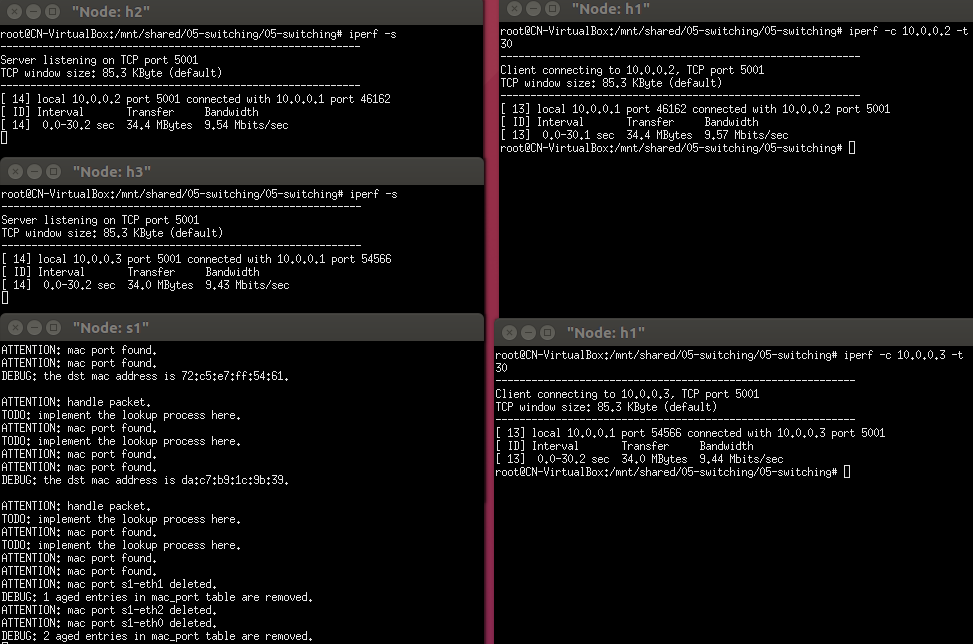
b1：hub

并发请求时，h1向b1以20Mbps的速率发送数据包，其中一半的目的主机是h2，另一半的目的主机是h3。数据包到达b1后，开始向h2和h3转发。每一个数据包，都会被复制后发往h2和h3，所以对于h2，虽然其接受速率理论上最高为10Mbps，但是其中约有一半是h1要发往h3的包，对于h2来言属于无效包，白白占用带宽，真正有用的是h1要发给h2的包。对于h3也是类似的情况。

所以，如果hub对包的转发顺序是绝对随机的话，h1与h2之间的传输速率和h1与h3之间的传输速率都应在5Mbps之下。

实际测试中，h1-h2：2.28Mbps h1-h3：6.98Mbps，二者相加为9.26Mbps，小于10Mbps，满足理论分析。

再来看这次实验实现的交换机的转发效率：



图五 Switch的转发效率测试

下面详细描述交换机进行转发的过程：

实验条件：h1：client，同时向h2和h3请求服务

h2：server

h3：server

s1：hub

h1想要给h2、h3发包，首先需要知道二者的MAC地址。（以h1给h2通信为例）h1（IP:10.0.0.1）首先在自己的ARP缓存中查找h2（IP：10.0.0.2）的MAC地址，查找发现不存在，于是向s1节点发送一个ARP广播包，目的MAC地址为FF:FF:FF:FF:FF:FF，意图为向局域网内所有主机询问h2的MAC地址。

s1节点从PORT 1收到h1发来的数据包。解析该包，获得h1的MAC地址为Host 1 MAC Addr，于是向转发表中加入h1的转发项：

|  |  |  |
| --- | --- | --- |
| **目的地址** | **转发端口** | **老化时间** |
| Host 1 MAC Addr | Port 1 | 30 sec |

图六 实时更新的Switch的转发表

继续解析，s1发现该数据包的目的MAC地址为FF:FF:FF:FF:FF:FF，转发表中没有该项目，于是s1将该包向所有其他端口广播。

该包从PORT 2发送出去，到达h2。h2向自己的ARP缓存中写入h1的MAC地址，随后向s1节点发送一个ARP单播包，源MAC地址为h2，目的MAC地址为h1。

s1节点从PORT 2收到h2发来的数据包。解析该包，获得h2的MAC地址为Host 2 MAC Addr，于是向转发表中加入h2的转发项：

|  |  |  |
| --- | --- | --- |
| **目的地址** | **转发端口** | **老化时间** |
| Host 1 MAC Addr | Port 1 | 30 sec |
| Host 2 MAC Addr | Port 2 | 30 sec |

图七 实时更新的Switch的转发表

继续解析，s1发现该数据包的目的MAC地址为Host 1 MAC Addr，转发表中该项的转发端口为PORT 1，于是s1将该包从PORT 1转发。

h1接收了s1转发来的数据包。解析获得h2的MAC地址，写入ARP缓存。随后h1再向h2发包就从ARP缓存中查找h2的MAC地址，然后填入报文首部。h1与h3间的通信与上类似。

并发请求时，h1向b1以20Mbps的速率发送数据包，其中一半的目的主机是h2，另一半的目的主机是h3。数据包到达s1后，发给h2的包从PORT 2转发，发给h3的包从PORT 3转发。所以，s1-h2和s1-h3间的链路始终满载着10Mbps的包，并且全都是有效包。考虑到中间延迟，实际测量速率：h1-h2：9.54Mbps h1-h3：9.44Mbps，符合理论分析。

|  |  |  |
| --- | --- | --- |
| **目的地址** | **转发端口** | **老化时间** |
| Host 1 MAC Addr | Port 1 | 0 sec |
| Host 2 MAC Addr | Port 2 | 0 sec |
| Host 3 MAC Addr | Port 3 | 0 sec |

图八 老化执行前的Switch的转发表

最后，传输全部结束后，等待30秒，可以在s1的终端中看到，转发表中s1-eth0、s1-eth1和s1-eth2端口对应的项被删除，证明交换机实现了30s老化功能。

|  |  |  |
| --- | --- | --- |
| **目的地址** | **转发端口** | **老化时间** |
|  |  |  |

图九 老化执行后的Switch的转发表

2. 实验分析

本次实验实现的具体代码见第二部分实验代码详解。

实现switch的main函数与hub基本一致，在上次的实验报告中已详细阐述，在此不再赘述，但有所不同的是：

void **init\_ustack**()

{

    instance = **safe\_malloc**(sizeof(ustack\_t));

**bzero**(instance, sizeof(ustack\_t));

**init\_list\_head**(&instance->iface\_list);

**init\_all\_ifaces**();

**init\_mac\_port\_table**();

}

在init\_ustack函数增加了init\_mac\_port\_table函数，代码见下：

void **init\_mac\_port\_table**()

{

**bzero**(&mac\_port\_map, sizeof(mac\_port\_map\_t));

    for (int i = 0; i < HASH\_8BITS; i++) {

**init\_list\_head**(&mac\_port\_map.hash\_table[i]);

    }

**pthread\_mutex\_init**(&mac\_port\_map.lock, NULL);

**pthread\_create**(&mac\_port\_map.thread, NULL, sweeping\_mac\_port\_thread, NULL);

}

主要功能为初始化转发表的哈希结构，初始化转发表的互斥锁，创建老化线程。

void \***sweeping\_mac\_port\_thread**(void \*nil)

{

    while (1) {

**sleep**(1);

        int n = **sweep\_aged\_mac\_port\_entry**();

        if (n > 0)

**log**(DEBUG, "%d aged entries in mac\_port table are removed.", n);

    }

    return NULL;

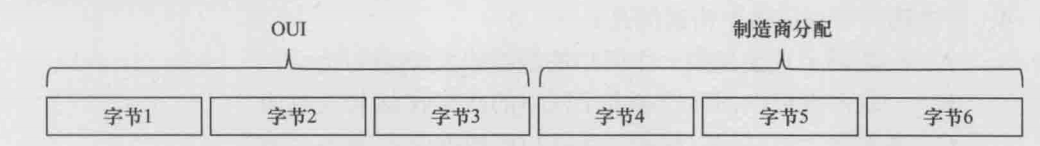
}

老化线程每隔1s执行一次，调用sweep\_aged\_mac\_port\_entry函数清除老化时间到达0的转发表项，n的值为该次清除的转发表表项个数。

3. 思考题

1） 我们知道，网络中存在广播包，其目的MAC地址设置为全0xFF ，例如ARP请求数据包。这种广播包对交换机行为逻辑有什么影响？

首先我们来看一个设备的MAC地址。制造商在生产制造网卡的过程中，会往每一块网卡的 ROM 中烧入一个 48bit 的 BIA（Burned-In Address，固化地址）地址，BIA 地址的前 3 个字节就是该制造商的 OUI，后 3 个字节由该制造商自己确定，但不同的网卡，其 BIA 地址的后 3 个字节不相同。烧入进网卡的 BIA 地址是不能被更改的，只能被读取出来使用。



图十 BIA地址格式

实际上，BIA地址只是MAC地址中的一部分，是一种单播MAC地址。MAC 地址共分为 3 种，分别为单播 MAC 地址、组播 MAC 地址、广播 MAC 地址。这 3 种 MAC 地址的定义分别如下：

1）单播 MAC 地址是指第一个字节的最低位是 0 的 MAC 地址。

2）组播 MAC 地址是指第一个字节的最低位是 1 的 MAC 地址。

3）广播 MAC 地址是指每个比特都是 1 的 MAC 地址。广播 MAC 地址是组播 MAC 地址的一个特例。

也就是说，不存在某个主机的MAC地址为全F。对于交换机而言，一方面，交换机收到的数据包的源MAC地址永远不会是全F，因为不存在这样的主机，所以全F的MAC地址永远不会被学习到交换机的转发表中；另一方面，交换机也永远不会在转发表中查询到全F的MAC地址对应的端口映射。两方面共同导致了交换机在面对目的MAC地址为全F时，采取的策略是广播，这也与ARP协议等设置广播报文的目的MAC地址为全F的初衷一致。

对于MAC地址分类的详细描述记录在我的csdn博客下：

<https://blog.csdn.net/Therock_of_lty/article/details/105864601>

2）理论上，足够多个交换机可以连接起全世界所有的终端。请问，使用这种方式连接亿万台主机是否技术可行？并说明理由。

答案肯定是不行的。

首先，目前我们实现的交换网络拓扑非常简单，树状结构，如果出现了环路，目前的交换机算法将会在数据转发过程中形成环路。所以，目前局域网交换机应用较多的算法是生成树协议。

但是，无论使用哪个算法，交换机要想实现正确的功能，其转发表需要记录网络中的所有主机。10亿台主机所在量级为2^30，仅仅是转发表的项数就有1G，也就是转发表本身占据数十GB的空间，无论是哪个算法都无法快速处理如此庞大的数据，首先从技术上这是不可行的。

其次，任何一个交换机内部都保存网络上所有主机的MAC地址，这意味着网络安全性非常差，因为任何人都可以获取到任何一个主机的真实MAC地址并发起攻击，这对网络环境的安全是不可接受的。

（二）实验代码详解

一、handle\_packet

void **handle\_packet**(iface\_info\_t \*iface, char \*packet, int len)

{

    struct ether\_header \*eh = (struct ether\_header \*)packet;

**log**(DEBUG, "the dst mac address is " ETHER\_STRING ".\n", **ETHER\_FMT**(eh->ether\_dhost));

*// TODO: implement the packet forwarding process here*

**fprintf**(stdout, "ATTENTION: handle packet.\n");

    iface\_info\_t\* dest\_iface = **lookup\_port**(eh->ether\_dhost);

    if(dest\_iface != NULL)

**iface\_send\_packet**(dest\_iface, packet, len);

    else

**broadcast\_packet**(iface, packet, len);

**insert\_mac\_port**(eh->ether\_shost, iface);

}

调用handle\_packet函数的时机在每次收到新的数据包时。首先使用lookup\_port函数在交换机转发表中查找是否有目的MAC地址的表项。如果找到的话，就将该数据包从该表项中记录的端口发送出去；如果没有找到，就向所有其他端口进行广播。

最后，对该数据包的源MAC地址执行insert\_mac\_port函数，修改转发表。

二、lookup\_port

iface\_info\_t \***lookup\_port**(u8 mac[ETH\_ALEN])

{

*// TODO: implement the lookup process here*

**pthread\_mutex\_lock**(&mac\_port\_map.lock);

**fprintf**(stdout, "TODO: implement the lookup process here.\n");

    uint8\_t hash\_val = **hash8**((char\*)mac, ETH\_ALEN);

    mac\_port\_entry\_t \*mac\_entry = NULL;

    int found = 0;

**list\_for\_each\_entry**(mac\_entry, &mac\_port\_map.hash\_table[hash\_val], list)

    {

        found = 1;

        for(int i = 0; i < ETH\_ALEN; i++)

        {

            if(mac\_entry->mac[i] != mac[i])

                found = 0;

        }

        if(found)

        {

**fprintf**(stdout,"ATTENTION: mac port found. \n");

            mac\_entry->visited = **time**(NULL);

**pthread\_mutex\_unlock**(&mac\_port\_map.lock);

            return mac\_entry->iface;

        }

    }

**fprintf**(stdout,"ATTENTION: mac port not found. \n");

**pthread\_mutex\_unlock**(&mac\_port\_map.lock);

    return NULL;

}

调用lookup\_port函数，首先要获取转发表的互斥锁。获得锁后，计算当前MAC地址的哈希值，调用宏list\_for\_each\_entry查找当前哈希值所在的转发链表中是否存在于当前MAC地址匹配的转发表项。如果有，就标准输出找到mac端口映射，将该表项的访问时间修改为当前的时间，释放互斥锁，返回找到的端口。如果没有，就标准输出没有找到相应映射表项，释放互斥锁，返回NULL。

三、broadcast\_packet

void **broadcast\_packet**(iface\_info\_t \*iface, char \*packet, int len)

{

*// TODO: implement the broadcast process here*

    iface\_info\_t \*IFACE = NULL;

**list\_for\_each\_entry**(IFACE, &instance->iface\_list,list)

    {

        if(IFACE->fd != iface->fd)

**iface\_send\_packet**(IFACE, packet, len);

    }

**fprintf**(stdout, "ATTENTION: broadcast completed.\n");

}

当调用到broadcast\_packet函数时，首先新建一个iface\_info\_t变量，为防止野指针赋值NULL。随后调用list\_for\_each\_entry宏，对整个链表进行遍历。遍历过程中，如果当前遍历到的端口不是发送该消息的端口，那么就调用iface\_send\_packet函数将收到的包发送给该主机；如果是发送该消息的主机，就跳过。最后打印广播完成的标识。

四、insert\_mac\_port

void **insert\_mac\_port**(u8 mac[ETH\_ALEN], iface\_info\_t \*iface)

{

    iface\_info\_t \*IFACE = **lookup\_port**(mac);

*//DONT USE lookup\_port between lock and unlock!!!!!!*

    if (IFACE) {

        (**list\_entry**(IFACE, mac\_port\_entry\_t, iface))->visited = **time**(NULL);

**fprintf**(stdout, "ATTENTION: mac port found.\n");

        return;

    }

**pthread\_mutex\_lock**(&mac\_port\_map.lock);

    uint8\_t hash\_val = **hash8**((char\*)mac, ETH\_ALEN);

    mac\_port\_entry\_t \*mac\_entry = **malloc**(sizeof(mac\_port\_entry\_t));

    for (int i = 0; i < ETH\_ALEN; i++)

        mac\_entry->mac[i] = mac[i];

    mac\_entry->iface = iface;

    mac\_entry->visited = **time**(NULL);

**list\_add\_tail**(&mac\_entry->list, &mac\_port\_map.hash\_table[hash\_val]);

**fprintf**(stdout, "ATTENTION: mac port %s inserted.\n", iface->name);

**pthread\_mutex\_unlock**(&mac\_port\_map.lock);

}

对一个到达的数据包解析源MAC地址后，启动insert\_mac\_port函数。首先调用lookup\_port函数查看转发表中是否有该MAC地址，如果有，就用宏list\_entry找到该端口的真实节点，修改其访问时间为当前时间，标准输出找到mac端口映射，返回。

如果没找到，申请转发表的互斥锁（之所以在前面不用加锁，是因为lookup\_port函数本身需要申请互斥锁）。计算MAC地址的哈希值hash\_val，新分配一个转发表项节点，将其mac位修改为MAC地址，端口修改为收到该数据包的端口，访问时间修改为当前时间，调用宏list\_add\_tail将该节点加到hash\_val对应的转发表链表的末尾，最后标准输出插入mac端口映射，释放互斥锁退出。

五、sweep\_aged\_mac\_port\_entry

老化操作有多种实现方式，本次代码选择的是老化时间栏记录最近一次访问的时间。如果当前时间与最近一次访问的时间相差超过30s，就将该表项删除。

int **sweep\_aged\_mac\_port\_entry**()

{

*// TODO: implement the sweeping process here*

**pthread\_mutex\_lock**(&mac\_port\_map.lock);

*//fprintf(stdout, "TODO: implement the sweeping process here.\n");*

    int number = 0;

    mac\_port\_entry\_t \*mac\_entry, \*q;

    for(int i = 0; i < HASH\_8BITS; i++)

    {

**list\_for\_each\_entry\_safe**(mac\_entry, q, &mac\_port\_map.hash\_table[i], list)

        {

            if(mac\_entry->visited + MAC\_PORT\_TIMEOUT < **time**(NULL))

            {

**fprintf**(stdout, "ATTENTION: mac port %s deleted.\n",mac\_entry->iface->name);

**list\_delete\_entry**(&mac\_entry->list);

**free**(mac\_entry);

                number++;

            }

        }

    }

**pthread\_mutex\_unlock**(&mac\_port\_map.lock);

    return number;

}

首先获得转发表的互斥锁，调用宏list\_for\_each\_entry\_safe安全遍历每一个哈希值的转发链表，遍历时如果发现某一表项超时（算法见上），就标准输出删除该表项，调用宏list\_delete\_entry删除链表上的该节点，释放该节点的空间，计数器加1。全部结束后，释放互斥锁，返回删除的表项数。