

# FAST UA 编程入门手册

## 目录

前言 .....	2
1、FAST API 概述 .....	4
1.1 fast 源码结构 .....	4
1.2 fast API 详解 .....	5
2、l2switch ——fast 架构实现二层交换案例详解 .....	13
2.1 二层交换机工作原理 .....	13
2.2 l2switch 实现的思路 .....	14
2.3 FAST API 回顾 .....	20
2.4 FAST 核心数据结构 .....	21
2.5 小结 .....	26
2.6 扩展——流表规则 .....	30
3、tools 工具——寄存器读写操作案例详解 .....	32
3.1 寄存器地址 .....	32
3.2 端口信息获取详解 .....	33
3.3 tools 工具的使用 .....	36
4、代码编译操作介绍 .....	39
附录 .....	42
一、OpenBox 操作指南 .....	42
1.1 串口的使用方式 .....	42
1.2 IP 地址的配置 .....	45
1.3 固定 eth 的 IP 地址 .....	47
1.4 ssh 远程登录 .....	48
1.5 固件烧录 rbf 的文件 .....	49
二、常见问题解答 .....	51

# 前言

假如从来没有听说过 fast，或者仅听说过名字但 fast 原理完全陌生，建议先补充一下原理知识。想了解 fast 架构的原理，可以去 FAST 开源社区下载《fast 开源平台手册》等资料。

本文档可以作为 FAST UA 编程新手入门的一个参考，适用于想在 CPU 上通过 FAST UA 实现自定义软件功能的用户读者。

为了便于更好的理解 FAST UA 编程和快速上手， 本文选取了两个示例进行详解，分别为 l2switch 二层交换机的实现和获取端口信息。事实上，本文选取的两个示例，其实是 FAST UA 编程中经常会碰到的两类相对典型的基础功能。其一，使用 FAST UA 编程去接收和发送报文。对于网络相关的研究者来说，经常可能会有分析报文的需求，会对网络中一些报文的头部信息如 mac 地址、IP 地址、tcp/udp 端口号、消息类型等感兴趣。为实现某一功能，也许需要把这些报文头部信息做一些统计分析，甚至修改。也就是说，我们需要使用 fast 接口接收报文，然后对 收到的报文进行一些特别的处理，再使用 fast 接口把报文发送出去。使用 fast 接口接收报文，如果 UA 是处理报文的第一个 UA，接收的报文就是来自网络，而如果是处理报文的中间某个 UA，报文就可能是来自其他 UA。同样的，使用 fast 接口发送报文，如果 UA 是处理报文的最后一个 UA，报文就需要发送到网络，而如果是处理报文的中间某个 UA，报文则需要发送给其他 UA。那么报文来自哪里又要发送到哪里，这个怎么判别怎么去实现呢？如何去使用 fast 编程去接收和发送报文呢？阅读完 l2switch 二层交换机案例详解章节后，这些疑问就都解开了。其二，使用 FAST UA 编程去进行读写寄存器操作。在实现端口扫描、统计收发包字节数等功能时，这些都会涉及到读写寄存器操作。Fast UA 编程中如何去读写寄存器呢？tools 工具寄存器读写操作案例有详细说明。

最后，描述一下本文档的组织结构。

第一节，FAST API 概述。主要是对每一个 FAST API 的功能和原理进行详细说明，帮助用户读者从整体上理解 FAST UA 编程。为什么用户只要简单的调用一个 API，写上一两行代码，就能实现这么强大的功能呢？看完本章节之后就明白了。

第二节，l2switch 二层交换机案例详解。以实现 UA 程序 l2switch 为示例，描述了使用 FAST UA 编程去接收和发送报文的实现思路、开发步骤。同时，对使用到的 FAST API 和核心数据结构也进行了详细说明。

第三节，tools 工具寄存器读写操作案例详解。以读取寄存器获取端口信息为示例，描述了如何使用 FAST UA 编程去进行读写寄存器操作。

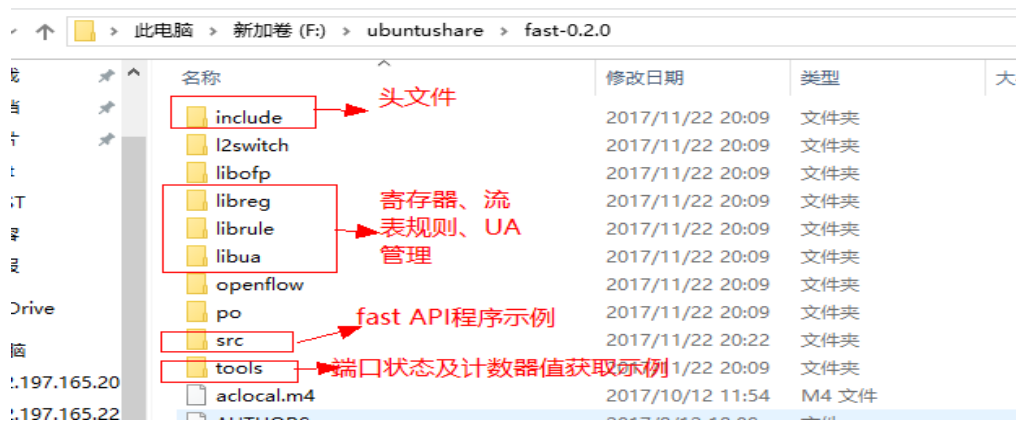
第四节，代码编译操作介绍。对于 UA 软件开发者来说，代码编译调试是必须要有的操作。通过前面三个章节的学习，差不多可以写一个简单的 UA 小程序试试手了。那么编译环境怎么搭建呢？写出来的 UA 代码怎么去调试呢？看完这个章节就知道了。

# 1、FAST API 概述

Fast 提供了若干 API 函数供用户直接调用。可以归结为三类：UA 管理 API、规则管理 API 和硬件管理 API。为了更好的理解各 API 函数的功能和使用，阅读 fast 源码会是一个不错的方法。

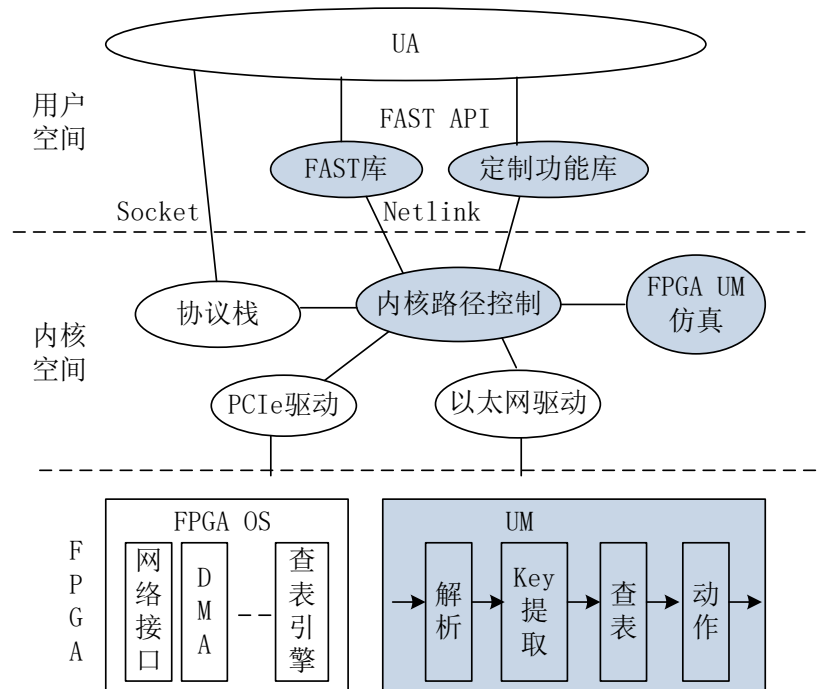
## 1.1 fast 源码结构

以 fast-0.2.0 为例，把 fast 源码解压以后，如下所示：



如果没有涉及到 openflow 相关的，可以先重点关注上图标红部分的代码。Include 目录是 fast 相关的头文件。libreg、librule、libua 三个目录，是 fast 提供所有 API 函数的源码实现，编译之后生成的 fast 静态库 libreg.a、librule.a、libua.a 也是分别放在这三个目录中。Src 目录中给出了 UA 注册和流表规则下发相关 API 调用的一个源码示例。Tools 目录下含有 port\_counts 和 port\_status 两个目录，分别是实现端口收发数据报文和端口状态获取功能的示例源码。

## 1.2 fast API 详解



FAST 库提供 FAST API 函数, 用户 UA 进程可直接调用 FAST API。UA 进程向平台相关软件注册模块信息、注销模块信息、收发报文, 都是先通过 netlink 协议与内核通信, 再到硬件设备。而 UA 进程对于硬件资源的读写, 则是通过操作虚拟地址实现。

### ● UA 管理 API

#### (1) Fast UA 初始化

```
int fast_ua_init(int mid, fast_ua_recv_callback callback);
```

设置向系统注册 UA 进程的 MID 和接收报文处理的回调函数。

#### (2) Fast UA 注册

```
int fast_ua_register(int mid);
```

首先创建一个 NetLink 类型 SOCKET 并绑定, 且设置好通信源、目的节点信息。也就是说配置好 UA 进程与内核之间的 socket 通道。此 socket 句柄为全局变量, 后续的收发报文及 fast UA 注销, 都是通过此 socket 句柄与内核通信。

UA 进程向内核发起注册请求, 并等待注册回应消息。如果回应的消息为 UA\_OK, 则注册成功。

### (3) Fast UA 注销

```
void fast_ua_destroy(void);
```

构建 UA 注销报文，并通过 socket 给内核发送消息。

UA 程序正常或非法退出时执行的操作，向内核告知自己将要结束，让内核清除此 MID 号上的报文转发功能。

### (4) UA 发送报文功能函数

```
int fast_ua_send(struct fast_packet *pkt,int pkt_len);
```

构建好 Netlink 消息报文，消息内容为 fast 报文，然后通过 socket 发送给内核。

### (5) UA 接收报文功能函数

```
void fast_ua_recv();
```

接收内核分派的报文，并调用 UA 初始化时设置好的回调函数对报文进行处理。

## ● 硬件管理 API

### (1) 初始化硬件

```
int fast_init_hw(u64 addr,u64 len);
```

针对 openbox 系统和 NetMagic08 设备有所区别。Irouter 设备是 openbox 系统且内含 CPU，初始化硬件主要是读 PCI 的配置信息。NetMagic08 设备需要外置 CPU，所以初始化硬件主要是把外置 CPU 与 NetMagic08 设备建立 socket 通道。请注意，这里建立了两个 socket，一个用于发消息，一个用于收消息。

### (2) 释放硬件资源

```
void fast_distroy_hw(void);
```

同样的，针对 openbox 系统和 NetMagic08 设备有所区别。openbox 系统主要是解除内存映射。而 NetMagic08 设备则是通过 socket 向驱动发送释放资源的消息。

### (3) 读取指定寄存器

```
u64 fast_reg_rd(u64 regaddr);
```

openbox 系统是直接操作虚拟地址。而 NetMagic08 设备则是通过 socket 向驱动发送读的消息。

(4) 往硬件写入一条指定的规则内容

```
void fast_reg_wr(u64 regaddr,u64 regvalue);
```

openbox 系统是直接操作虚拟地址。而 NetMagic08 设备则是通过 socket 向驱动发送写的消息。

## ● 规则管理 API

### 规则管理关键结构体详解

若干条 fast 规则，便组成了一张表。

```
struct rule_table
{
    u64 cnt; //计数器，统计当前流表下发了多少条规则
    u64 pad; //保留字段
    struct fast_rule rules[FAST_RULE_CNT];
};

struct fast_rule
{
    struct flow key; //**< @brief 规则内容存储结构 @see ::flow*/
    struct flow mask; //**< @brief 规则掩码设置结构,与 key 一一对应 @see ::flow*/
    u32 priority; //**< @brief 规则的优先级设置*/
    u32 valid; //**< @brief 规则的有效标志设置,要删除一条规则只需要将此标记置 0,并更新到硬件*/
    u32 action; //**< @brief 规则所对应的执行动作，动作由两部分组成，高 4 位为 ACTION 枚举类型，低 28 位不此类型对应的值 @see ::ACTION*/

    /*此前数据必须要写入到硬件*/
    u32 flow_stats_len; //**< @brief 流表信息长度*/
    u64 *flow_stats_info; //**< @brief 流表信息存储指针位置*/
    u64 cookie; //**< @brief 用来存储流的 cookie 信息，软件使用*/
    u64 cookie_mask; //**< @brief 用来存储流 cookie 的掩码信息，软件使用*/
    u32 md5[4]; //**< @brief MD5 值 @note 用来区分规则的唯一性，软件生成规则时即会判断是否规则重复*/
    u32 pad[18]; //**< @brief 总长 256B，此为确保护数据结结构大小做的填充*/
};
```

请注意，*struct fast\_rule* 中的两个成员 *key* 和 *mask*。*key* 和 *mask* 的类型都是 *struct flow*。规则内容是保存在 *key* 中。*Mask* 中的内容是为了标识 *key* 中对应域的规则是否需要匹配。为 1 表示使用，为 0 表示忽略。

例如：

```
rule[i].key.port = 0x0;
rule[i].mask.port = 0xF;
```

这就标识 *port* 域的规则是要被使用的。

还请注意一下，*struct fast\_rule* 中的成员 *key*、*mask*、*priority*、*valid* 及 *action* 这些数据必须写入到硬件中。

```
struct flow /*2017/06/01 开始启用，支持 IPv6*/
{
    //eth;
    u8  dmac[ETH_ALEN]; /**< @brief Ethernet source address. */
    u8  smac[ETH_ALEN]; /**< @brief Ethernet destination address. */
    u16 tci; /**< @brief 0 if no VLAN, VLAN_TAG_PRESENT set otherwise. */
    u16 type; /**< @brief Ethernet frame type. */

    //ip;
    u8  proto; /**< @brief IP protocol or lower 8 bits of ARP opcode. */
    u8  tos; /**< @brief IP ToS. */
    u8  ttl; /**< @brief IP TTL/hop limit. */
    u8  frag:4, /**< @brief One of OVS_FRAG_TYPE_*. */
    port:4; /**< @brief Input Port*/

    union
    {
        /**< @brief IPv4 协议相关字段*/
        struct
        {
            //addr;
            u32 src; /**< @brief IP source address. */
            u32 dst; /**< @brief IP destination address. */

            union{
```



```

/**< @brief IPv4 传输层信息与 ARP 信息共用体，二者互斥存在*/
/**< @brief IPv4 的传输层端口与标志信息*/
struct
{
    u16 sport;  /**< @brief TCP/UDP/SCTP source port. */
    u16 dport;  /**< @brief TCP/UDP/SCTP destination port. */
    u16 flags;   /**< @brief TCP flags. */
} tp;

/**< @brief ARP 的 MAC 地址信息*/
struct
{
    u8 sha[ETH_ALEN];  /**< @brief ARP source hardware
                        address. */
    u8 tha[ETH_ALEN];  /**< @brief ARP target hardware
                        address. */
} arp;
};
} ipv4;

/**< @brief IPv6 协议相关字段*/
Struct
{
    //addr;
    struct in6_addr src;  /**< @brief IPv6 source address. */
    struct in6_addr dst;  /**< @brief IPv6 destination address. */

    u32 label;           /**< @brief IPv6 flow label. */

    union
    {
        /**< @brief IPv6 的传输层端口与标志信息*/
        /**512 位宽表项不够，暂不使用 ND 协议中的 targetIP!!!
        为方便以后扩展，将其移至末尾*/
        Struct
        {
            u16 sport;  /**< @brief TCP/UDP/SCTP source port. */
            u16 dport;  /**< @brief TCP/UDP/SCTP destination port. */
            u16 flags;   /**< @brief TCP flags. */
        } tp;
    };
} ipv6;
};
};

```

## 规则管理 API 详解

### (1) 添加一条规则

```
int fast_add_rule(struct fast_rule *rule);
```

先判断这条规则是否存在，若存在不用添加，若不存在则在第一个可用的位置添加此规则，并把计数器值加 1。

### (2) 修改指定位置的规则

```
int fast_modify_rule(struct fast_rule *rule,int idx);
```

将指定的规则重新写入到硬件设备中

### (3) 删除一条规则

```
int fast_del_rule(int idx);
```

把零规则写入到指定的规则索引号中，且有效标致置为 0，并同步到硬件设备中。

### (4) 初始化规则

```
void init_rule(u32 default_action);
```

主要是将硬件流表规则清空，并写入默认的动作操作指令

@param *default\_action* : 匹配不了所有规则后，执行的动作指令

FAST API 函数 *init\_rule(u32 default\_action)* 的功能就是把硬件流表规则清空，并且下发默认规则。如果调用了 *init\_rule* 函数之后而没有再设置其他的流表项，那么 *default\_action* 就是唯一的流表规则，所有报文都按照 *default\_action* 执行。

```
/*初始化规则库,将硬件规则存储空间清零*/  
void init_rule(u32 default_action)  
{  
    /*最后的加2 指优先级和有效位*/  
    u32 i = 0,j = 0,cnt = sizeof(struct flow)*2/sizeof(struct reg_value) + 2;  
  
    memset(&table,0,sizeof(struct rule_table));  
}
```

```

memset(&zero_rule,0,sizeof(struct fast_rule));

#ifndef LOOKUP_BV

for(i<FAST_RULE_CNT;i++)
{
    for(j=0;j<cnt;j++)
    {
        //将每条规则数据清零
        openbox_rule_reg_wr(RULE_LEN*i + j*4,0);
    }
    openbox_action_reg_wr(i*8,0); //将规则清零
}
#else
/*BV 算法不需要将表空间写一次零，可通过表复位操作实现清零，暂未支持*/
#endif

//给硬件配置默认规则
openbox_action_reg_wr(FAST_DEFAULT_RULE_ADDR,default_action);
}

```

$cnt = \text{sizeof}(\text{struct flow}) * 2 / \text{sizeof}(\text{struct reg\_value}) + 2$  指优先级和有效位\*/

即  $cnt$  为 `struct fast_rule` 中的成员 `key`、`mask`、`priority`、`valid` 以 4 字节为单位的计数值。

`openbox_rule_reg_wr(RULE_LEN*i + j*4,0);` //将每条规则数据清零  
 每条 fast 规则大小为 256 字节，所以第  $i$  条规则的地址偏移量为  $RULE\_LEN*i$ 。 $j*4$  表示以 4 字节为单位读取。

`openbox_action_reg_wr(i*8,0);` //将规则清零  
`action` 和 `key`、`mask`、`priority`、`valid` 的存储地址是分开的，另外有一块地址专门存放 `action` 的值。 $i*8$  是因为寄存器地址为 64 位即 8 字节。

```

/* 读规则寄存器*/
u64 openbox_rule_reg_rd(u64 regaddr)
{
    fast_reg_wr(FAST_RULE_REG_RADDR,(u64)regaddr<<32);
    return fast_reg_rd(FAST_RULE_REG_VADDR);
}

/* 写规则寄存器*/
void openbox_rule_reg_wr(u32 addr,u32 value)
{
    //原来使用的规则写法, 正常
    fast_reg_wr(FAST_RULE_REG_WADDR,(u64)addr<<32/value);
}

/* 读动作寄存器*/
u32 openbox_action_reg_rd(u32 addr)
{
    return 0xFFFFFFFF & fast_reg_rd(FAST_ACTION_REG_ADDR + addr);
}

/* 写动作寄存器*/
void openbox_action_reg_wr(u32 addr,u32 value)
{
    fast_reg_wr(FAST_ACTION_REG_ADDR + addr,value);
}

```

读写规则寄存器、读写动作寄存器，都是对虚拟地址的操作。

## 2、l2switch ——fast 架构实现二层交换案例详解

l2switch 为运行在 fast 架构（irouter 设备）上面的一个 UA 程序，实现二层交换机的功能。也就是说 irouter 设备运行 UA 程序 l2switch 之后，就能够充当一台二层交换机。

之前只知道交换机的功能，从来没想过交换机功能怎么实现的，事实上是也看不到内部的实现。现在一个 UA 程序就能实现二层交换机的功能，有点好奇了。那么，l2switch 是怎么做到的呢？

### 2.1 二层交换机工作原理

UA 程序 l2switch 要做的事情就是实现二层交换机的功能，所以在实现 l2switch 之前应该明白二层交换机工作原理。

二层交换机通过解析和学习以太网帧的源 MAC 来维护 MAC 地址与端口的对应关系（保存 MAC 与端口对应关系的表称为 MAC 表），通过其目的 MAC 来查找 MAC 表决定向哪个端口转发，基本流程如下：

（1）二层交换机收到以太网帧，将其源 MAC 与输入端口的对应关系写入 MAC 表，作为以后的二层转发依据。如果 MAC 表中已有相同表项，那么就刷新该表项的老化时间。MAC 表表项采取一定的老化更新机制，老化时间内未得到刷新的表项将被删除掉；

（2）根据以太网帧的目的 MAC 去查找 MAC 表，如果没有找到匹配表项，那么向所有端口转发（输入端口除外）；如果目的 MAC 是广播地址，那么向所有端口转发（接收端口除外）；如果能够找到匹配表项，则向表项所示的对应端口转发，但是如果表项所示端口与收到以太网帧的端口相同，则丢弃该帧。

归纳一下，实现二层交换机的要点在于：

(1) 学习以太网帧的源 MAC，维护 MAC 表；

(2) 查找 MAC 表决定向哪个端口转发报文。

所以我们要用 fast 架构去实现二层交换机的功能，关键是实现上面两点。那么我们怎么使用 fast 编程去实现呢？这个问题的方案，其实就是我们要实现 l2switch 的思路。

## 2.2 l2switch 实现的思路

### (1) 获取所有经过 irouter 设备的报文

首先如果要去学习以太网帧的源 MAC，那么 l2switch 就要获取所有经过 irouter 设备的报文，才能提取源 MAC、目的 MAC 以及输入端口号。对于网络中的报文最先到达 irouter 设备硬件底层，也就是 fast 架构中的硬件流水线中。UA 程序 l2switch 是运行在 irouter 设备上面，所以我们应该想办法把经过 irouter 设备硬件底层的所有报文送至 UA 程序 l2switch。思路有了，代码怎么写呢？

#### FAST API

```
/*  
FAST 的 UA 程序初始化调用函数，需要 UA 程序提供两个参数。参数  
一表示接收目的 MID 为此值的报文，参数二表示接收到报文后通过此  
回调函数输出。  
*/  
int fast_ua_init(int mid, fast_ua_recv_callback callback);  
  
/*  
启动 UA 接收线程（用户代码继续执行），  
*/  
void fast_ua_recv();  
  
/*  
主要是将硬件流表规则清空，并写入默认的动作操作指令  
@param default_action : 匹配不了所有规则后，执行的动作指令  
*/  
void init_rule(u32 default_action);
```

FAST API 函数 *fast\_ua\_init* 的功能有两个，一是向系统注册自己的 mid，二是设置收到报文后的回调函数。

函数 *init\_rule(u32 default\_action)* 的功能就是把硬件流表规则清空，并且下发默认规则。如果调用了 *init\_rule* 函数之后而没有再设置其他的流表项，那么 *default\_action* 就是唯一的流表规则，所有报文都按照 *default\_action* 执行。

函数 *fast\_ua\_recv()* 是真正接收数据报文的函数，从内核协议栈中通过 netlink socket 把报文数据传给我们的 UA 程序。

回到 l2switch 中来，假设 UA 程序 l2switch 的 mid 为 129。那么上面想要的功能，如下三行代码就能实现。

1 *fast\_ua\_init(129, callback);*

向系统注册 l2switch 的 mid 为 129，收到报文后的回调函数为 *callback*。

2 *init\_rule(ACTION\_SET\_MID << 28/129);*

把硬件流表规则清空，并且将硬件所有报文送到模块 ID 为 129 的进程处理。这一行代码只是下发规则，告诉硬件把所有报文送至 UA 程序 l2switch。

3 *fast\_ua\_recv();*

从内核协议栈中通过 netlink socket 把报文数据传给我们的 UA 程序 l2switch。

## (2) 学习以太网帧的源 MAC，维护 MAC 表，发送报文

到目前为止，所有经过 irouter 设备的报文已经到达 UA 程序 l2switch。所以我们下一步就是要对收到的报文提取源 MAC、目的 MAC 以及输入端口号，维护 MAC 表，查找 MAC 表决定向哪个端口转发报文。

函数 *fast\_ua\_init* 有两个功能，上一步我们用到了功能一向系统注册 l2switch 的 mid 为 129。功能二是设置收到报文后的回调函数。也就是说，收到报文之后需要做的事情都是在回调函数中实现。所以在 l2switch 中，对收到的报文提取源 MAC、目的 MAC 以及输入端口号，

维护 MAC 表，查找 MAC 表决定向哪个端口转发报文等这些功能，就可以全部放在回调函数 *callback* 中去实现。

如下所示为 *callback* 的伪代码示例。

```
int callback(struct fast_packet *pkt,int pkt_len)
{
    int outport = -1;

    /*MAC 地址学习过程*/
    /*用源MAC 位置开始学习*/
    learn_smac(pkt->um.inport,&pkt->data[MAC_LEN]);

    /*查表过程*/
    /*用目的MAC 地址开始查表*/
    outport = find_dmac(pkt->um.inport,pkt->data);

    /*发送报文*/
    if(outport == -1)/*报文需要泛洪操作*/
    {
        pkt_send_flood(pkt,pkt_len);/*泛洪发送，保留输入端口不变调用*/
    }
    else/*正常转发*/
    {
        pkt->um.outport = outport;/*修改报文输出端口号*/
        pkt_send_normal(pkt,pkt_len);/*正常发送报文*/
    }
    return 0;
}
```

请注意，使用 fast 编程去发送和接收报文，必然会涉及到对报文的处理。所以熟悉 *fast\_packet* 定义和以太网数据报文结构是必须的。如若以太网数据报文结构不太清楚，请参考计算机网络相关书籍，下文补充一下 *fast\_packet* 相关定义。

```
struct fast_packet{
    struct um_metadata um; /*UM 模块数据格式定义*/
    u16 flag; /*2 字节对齐标志*/
    u8 data[1514]; /*完整以太网报文数据*/
}__attribute__((packed));
```



*fast\_metadata* 是 fast 数据报文的控制信息，包含报文时间戳，序号，源目的 MID，报文长度，端口号等信息。UM 控制信息数据格式详细定义如下：

```
/*UM 模块数据格式定义*/
struct um_metadata
{
    u64 ts:44,    /**< @brief 报文接收的时间戳 @note 如果用户需要使用表示更大的时间，建议存储在第二拍数据中（user[2] 字段）*/
                flowID:14, /**< @brief 流 ID 号*/
                priority:3, /**< @brief 报文优先级*/
                discard:1,  /**< @brief 指示报文是否丢弃@note 默认为0，表示不丢弃，置1时表示丢弃*/
                pktdst:1,   /**< @brief 报文的输出目的方向 @note
                                0 表示输出到网络端口，1 表示输出到 CPU*/
                pktsrc:1;   /**< @brief 报文的输入源方向 @note
                                0 表示网络端口输入，1 表示从 CPU 输入*/
    u64 outport:16, /**< @brief 报文输出端口号 @note 以 bitmap 形式表示，
                                1 表示从 0 号端口输出；8 表示从 3 号端口输出*/
    seq:12,    /**< @brief 报文接收时的序列号 @note 每个端口
                                独立维护一个编号*/
    dstmid:8,   /**< @brief 报文下次处理的目的模块编号*/
    srcmid:8,   /**< @brief 报文上次处理时的模块编号*/
    len:12,     /**< @brief 报文长度 @note 最大可表示 4095 字节，
                                但 FAST 平台报文缓存区最大为 2048，
                                完整以太网报文的 MTU 不要超过 1500*/
    inport:4,   /**< @brief 输入端口号 @note 取值：0——15，
                                最多表示 16 个输入端口*/
    ttl:4;      /**< @brief 报文通过模块的 TTL 值，
                                每过一个处理模块减 1*/
    u64 user[2]; /**< @brief 用户自定义 metadata 数据格式与内容
                                @remarks 此字段由可用户改写，但需要保证数据大小
                                严格限定在 16 字节*/
};
```

*fast\_packet* 结构体成员 **data** 存放的是完整的以太网报文数据。若熟悉以太网报文格式，我们知道通过偏移量是很容易得到报文的源 MAC 及目的 MAC 的，如 *data[0]-data[5]* 存放的为源 mac，*data[6]-data[11]* 存放的为目的 mac。再观察 *fast\_packet* 结构体成员 *um*，而 *um* 的成员 *inport* 为输入端口号，有了这些信息，自然我们很容易实现

mac 表的维护和查找 MAC 表决定向哪个端口转发报文。

到此为止，我们完全可以使用 fast 架构去实现 l2switch 了。

L2switch 实现的 UA 主函数 main 的伪代码示例：

```
int main(int argc, char* argv[])
{
    int ret = 0;

    /* 申请地址表存储空间*/
    nm08_table = (struct nm08_neigh_table *)malloc(sizeof(struct
nm08_neigh_table));

    /* 空间清零*/
    memset(nm08_table, 0, sizeof(struct nm08_neigh_table));

    /* 初始化平台硬件*/
    fast_init_hw(0, 0);

    /* UA 模块初始化 */
    if((ret = fast_ua_init(129, callback)))
    {
        perror("fast_ua_init!\n");
        exit (ret); // 如果初始化失败,则需要打印失败信息,并将程序结束退出!
    }

    /* 配置硬件规则, 将硬件所有报文送到模块 ID 为 129 的进程处理*/
    init_rule(ACTION_SET_MID << 28 | 129);

    /* 启动地址学习表老化线程*/
    nm08_start_aging();
}
```

```

/*启动线程接收分派给 UA 进程的报文*/
} fast_ua_recv();

/*主进程进入循环休眠中,数据处理主要在回调函数*/
while(1){sleep(9999);}

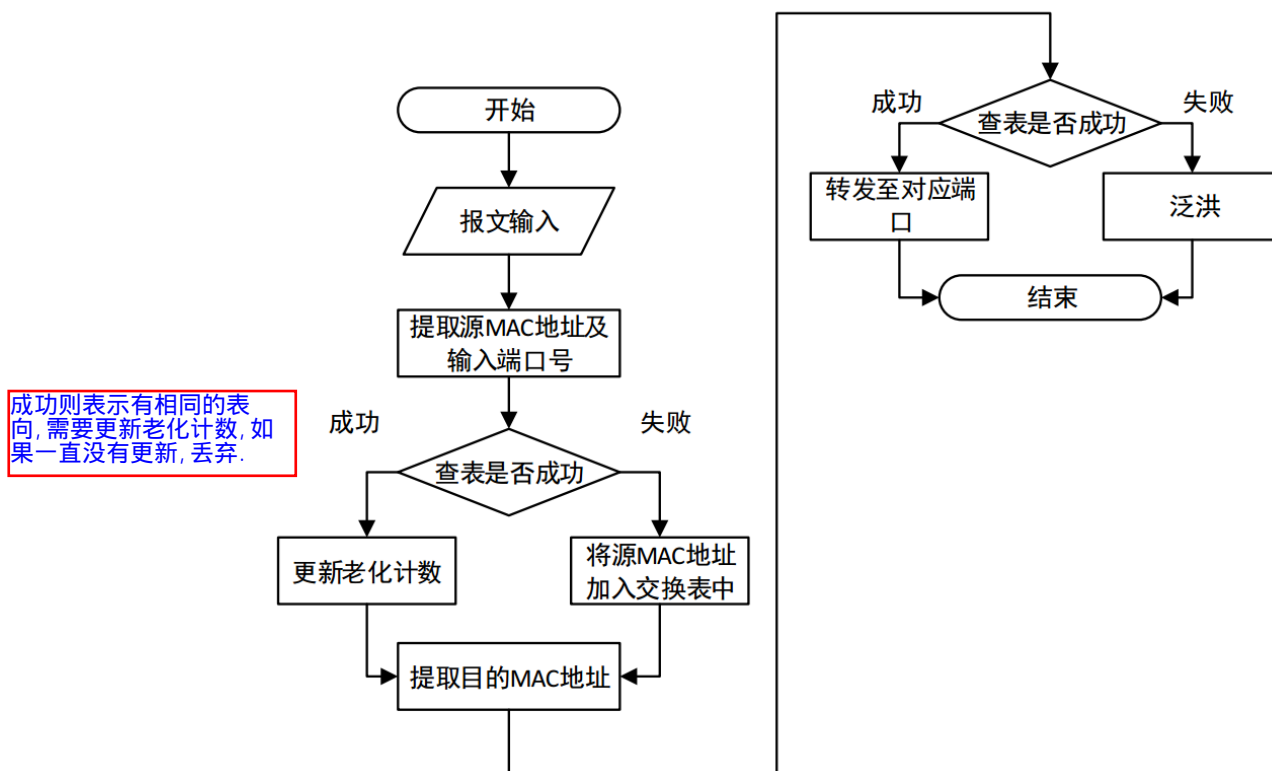
/*UA 注销操作*/
fast_ua_destroy(void);

/*释放硬件资源*/
fast_distroy_hw (void);

return 0;
}

```

## L2switch 流程图



## 2.3 FAST API 回顾

虽然之前已经有对 fast API 每一个函数做详细的说明，为了加深记忆，还是很有必要把在 l2switch 用到的 API 再罗列出来进行一下回顾。

(1) *int fast\_ua\_init(int mid, fast\_ua\_recv\_callback callback);*

FAST 的 UA 程序初始化调用函数，需要 UA 程序提供两个参数。参数一表示接收目的 MID 为此值的报文，参数二表示接收到报文后通过此回调函数输出。

函数 *fast\_ua\_init* 的功能有两个，一是向系统注册自己的 mid，二是设置收到报文后的回调函数。

(2) *void fast\_ua\_recv();*

启动 UA 接收线程，真正的接收数据报文函数。从内核协议栈中通过 netlink socket 把报文数据传给我们的 UA 程序。

(3) *int fast\_ua\_send(struct fast\_packet \*pkt, int pkt\_len);*

UA 发送报文操作。

@param pkt : 发送的报文指针

@param pkt\_len : 发送的报文长度

函数 *fast\_ua\_send* 主要是用于 UA 发送 fast 报文。通过 netlink socket 把报文数据从我们的 UA 程序传给内核协议栈。我们可以通过设置参数 *pkt->um.dstmid* 来告诉内核协议栈，我们期望把报文发送到硬件流水线，还是发送给其他 UA。

(4) *void fast\_ua\_destroy(void);*

UA 注销操作。UA 程序正常或非法退出时执行的操作，向内核告

知自己将要结束，让内核清除此 MID 号上的报文转发功能。

(5) *fast\_init\_hw(0,0);*

初始化硬件

(6) *void fast\_distroy\_hw(void);*

释放硬件资源

(7) *void init\_rule(u32 default\_action);*

主要是将硬件流表规则清空，并写入默认的动作操作指令

@*param default\_action* : 匹配不了所有规则后，执行的动作指令

FAST API 函数 *init\_rule(u32 default\_action)* 的功能就是把硬件流表规则清空，并且下发默认规则。如果调用了 *init\_rule* 函数之后而没有再设置其他的流表项，那么 *default\_action* 就是唯一的流表规则，所有报文都按照 *default\_action* 执行。

请注意，规则是要写入硬件中的，所以调用规则相关的 API 或者配置规则，那么就必然要调用初始化硬件函数 *fast\_init\_hw* 和释放硬件资源函数 *fast\_distroy\_hw*。

## 2.4 FAST 核心数据结构

使用 fast 编程去接收和发送报文，必然要用到下面这些核心数据结构。

### (1) *fast\_packet*

*fast\_packet* 结构体成员 *um* 为 fast 数据报文的控制信息，稍后有详细介绍。成员 *data* 也是值得关注的。*data* 存放的是原始以太网报文，所以 mac 地址、IP 地址、tcp/udp 端口号等信息，可以通过计算偏移量从 *data* 中获取，这也要求我们需要熟悉以太网报文格式。

```

struct fast_packet{
    struct um_metadata um; /*UM 模块数据格式定义*/
    u16 flag; /*2 字节对齐标志*/
    u8 data[1514]; /*完整以太网报文数据，暂时不含 CRC 数据*/
}__attribute__((packed));

```

## (2) *um\_metadata*

*um\_metadata* 是 fast 数据报文的控制信息，包含报文时间戳，序号，源目的 MID，报文长度，端口号等信息。对于 *um\_metadata* 结构体，需要重点关注，尤其是成员 *discard*、*pktdst*、*pktsrc*、*outport*、*dstmid*、*srcmid*、*inport*。

*Discard* 表示报文是否丢弃，默认为 0，表示不丢弃，置 1 时表示丢弃。如执行 ddos 防御策略时，异常报文流要丢弃，此时 *Discard* 必须设置为 1。

*Pktsrc*、*pktdst* 标识报文的输入输出方向。0 表示输出到网络端口，1 表示输出到 CPU。如 UA 处理后的报文需要发送到网络中，则应该设置 *Pktsrc* 为 1，*pktdst* 为 0。

*Inport*、*outport* 标识报文从设备的物理端口输入输出。一般是结合流表规则一起使用。

*dstmid*、*srcmid* 标识报文下次处理的目的模块编号和上次处理时的模块编号。如 UA 处理后的报文需要发送到网络中，则 *dstmid* 设置为 5。

为了更好的描述，我们以上文提到的回调函数 *callback* 中报文转发 *pkt\_send\_normal* 函数为例说明一下。

在 *callback* 函数中，有如下一段代码：

```

else/*正常转发*/
{
    pkt->um.outport = outport;/*修改报文输出端口号*/
    pkt_send_normal(pkt,pkt_len);/*正常发送报文*/
}

```

可以看到，在调用 `pkt_send_normal` 函数之前，我们先设置了从哪个物理端口把报文转发出去。

紧接着，在 `pkt_send_normal` 函数中我们又设置了报文下次处理的目的模块编号、报文的输入输出方向。下面这段代码的含义就是，报文来自 CPU，希望发送到网络中，下一个处理模块为 5 号 UA。注意一下 mid 为 5 的 UA，它是硬件流水线五元组最后一个 UA，即用户定义输出引擎模块。一般来说，如果我们希望报文送到硬件流水线送至网络中，则 `pkt->um.dstmid = 5`。

```
void pkt_send_normal(struct fast_packet *pkt,int pkt_len)
{
    pkt->um.dstmid = 5;
    pkt->um.pktdst = 0; 目的, 到网络中
    pkt->um.pktsrc = 1; 源, 来自CPU
    fast_ua_send(pkt,pkt_len);
}
```

对于其他成员的详细使用请参考 UM 模块数据格式定义。

```
/*UM 模块数据格式定义*/
struct um_metadata
{
    u64 ts:44,    /**< @brief 报文接收的时间戳 @note 如果用户需要使用表示更大的时间，建议存储在第二拍数据中（user[2] 字段）*/
    flowID:14,   /**< @brief 流 ID 号*/
    priority:3,  /**< @brief 报文优先级*/
    discard:1,   /**< @brief 指示报文是否丢弃@note 默认为0，表示不丢弃，置1时表示丢弃*/
    pktdst:1,    /**< @brief 报文的输出目的方向 @note
                  0 表示输出到网络端口，1 表示输出到CPU*/
    pktsrc:1;    /**< @brief 报文的输入源方向 @note
                  0 表示网络端口输入，1 表示从CPU输入*/
    u64 outport:16, /**< @brief 报文输出端口号 @note 以 bitmap 形式表示，
                  1 表示从0号端口输出；8 表示从3号端口输出*/
    seq:12,      /**< @brief 报文接收时的序列号 @note 每个端口
```

```

        独立维护一个编号*/
dstmid:8, /**< @brief 报文下次处理的目的模块编号*/
srcmid:8, /**< @brief 报文上次处理时的模块编号*/
len:12, /**< @brief 报文长度 @note 最大可表示 4095 字节,
        但 FAST 平台报文缓存区最大为 2048,
        完整以太网报文的 MTU 不要超过 1500*/
inport:4, /**< @brief 输入端口号 @note 取值: 0——15,
        最多表示 16 个输入端口*/
ttl:4; /**< @brief 报文通过模块的 TTL 值,
        每过一个处理模块减 1*/
u64 user[2]; /**< @brief 用户自定义 metadata 数据格式与内容
        @remarks 此字段由可用户改写, 但需要保证数据大小
        严格限定在 16 字节*/
};

```

### (3) fast\_rule

fast\_rule 是 fast 流表配置数据结构，fast 流表通过此接口配置到硬件 UM 模块，实现硬件对于报文的转发功能。流表这个概念，如果是第一次接触可能有点难以接受，来源于 SDN 网络的术语。

```

struct fast_rule
{
    struct flow key; /**< @brief 规则内容存储结构 @see ::flow*/
    struct flow mask; /**< @brief 规则掩码设置结构, 与 key 一一对应
        @see ::flow*/
    u32 priority; /**< @brief 规则的优先级设置*/
    u32 valid; /**< @brief 规则的有效标志设置, 要删除一条规则
        只需要将此标记置 0, 并更新到硬件*/
    u32 action; /**< @brief 规则所对应的执行动作, 动作由两部分组
        成, 高 4 位为 ACTION 枚举类型, 低 28 位不此类型对应
        的值 @see ::ACTION*/

    /* 此前数据必须要写入到硬件*/
    u32 flow_stats_len; /**< @brief 流表信息长度*/
    u64 *flow_stats_info; /**< @brief 流表信息存储指针位置*/
    u64 cookie; /**< @brief 用来存储流的 cookie 信息,
        软件使用*/
    u64 cookie_mask; /**< @brief 用来存储流 cookie 的掩码信息,
        软件使用*/
    u32 md5[4]; /**< @brief MD5 值 @note 用来区分规则的
        唯一性, 软件生成规则时即会判断是否规则重复*/
    u32 pad[18]; /**< @brief 总长 256B, 此为确保护数据结大小
        做的填充*/
};

```



#### (4) flow

flow 是 FAST 中对于流表项字段的定义，详细内容如下表：

```
struct flow /*2017/06/01 开始启用，支持 IPv6*/
{
    //eth;
    u8  dmac[ETH_ALEN]; /**< @brief Ethernet source address. */
    u8  smac[ETH_ALEN]; /**< @brief Ethernet destination address. */
    u16 tci; /**< @brief 0 if no VLAN, VLAN_TAG_PRESENT set otherwise. */
    u16 type; /**< @brief Ethernet frame type. */

    //ip;
    u8  proto; /**< @brief IP protocol or lower 8 bits of ARP opcode. */
    u8  tos; /**< @brief IP ToS. */
    u8  ttl; /**< @brief IP TTL/hop limit. */
    u8  frag:4; /**< @brief One of OVS_FRAG_TYPE_*. */
    port:4; /**< @brief Input Port*/

    union
    {
        /**< @brief IPv4 协议相关字段*/
        struct
        {
            //addr;
            u32 src; /**< @brief IP source address. */
            u32 dst; /**< @brief IP destination address. */

            union{
                /**< @brief IPv4 传输层信息与 ARP 信息共用体，二者互斥存在*/
                /**< @brief IPv4 的传输层端口与标志信息*/
                struct
                {
                    u16 sport; /**< @brief TCP/UDP/SCTP source port. */
                    u16 dport; /**< @brief TCP/UDP/SCTP destination port. */
                    u16 flags; /**< @brief TCP flags. */
                } tp;

                /**< @brief ARP 的 MAC 地址信息*/
                struct
                {
                    u8 sha[ETH_ALEN]; /**< @brief ARP source hardware
                                         address. */
                    u8 tha[ETH_ALEN]; /**< @brief ARP target hardware
                                         address. */
                };
            };
        };
    };
};
```

```

        } arp;
    };
} ipv4;

/**< @brief IPv6 协议相关字段*/
Struct
{
    //addr;
    struct in6_addr src; /**< @brief IPv6 source address. */
    struct in6_addr dst; /**< @brief IPv6 destination address. */

    u32 label;          /**< @brief IPv6 flow label. */

    union
    {
        /**< @brief IPv6 的传输层端口与标志信息*/
        /**512 位宽表项不够，暂不使用 ND 协议中的 targetIP!!!
        为方便以后扩展，将其移至末尾*/
        Struct
        {
            u16 sport; /**< @brief TCP/UDP/SCTP source port. */
            u16 dport; /**< @brief TCP/UDP/SCTP destination port. */
            u16 flags; /**< @brief TCP flags. */
        } tp;
    };
} ipv6;
};
};

```

## 2.5 小结

### Fast\_UA 开发步骤主要包含

#### (1) 初始化

初始化包括调用函数 `fast_init_hw` 硬件初始化和调用函数 `fast_ua_init` UA 模块初始化。

硬件初始化为申请硬件资源，UA 模块初始化则是向系统平台注册

自己的 MID，并设置处理接收报文的回调函数。

## (2) 添加流表规则

调用函数 *init\_rule* 来设置默认规则是一种简单的添加流表规则方式。只有设置了规则，我们所关心的报文才会被送至 UA 程序。

## (3) UA 接收报文

真正接收报文数据的函数 *fast\_ua\_recv*，从内核协议栈中通过 netlink socket 把报文数据传给我们的 UA 程序。

## (4) 回调函数处理报文

收到报文之后需要做的事情都是在回调函数 *Callback* 中实现。

## (5) 发送报文

函数 *fast\_ua\_send* 主要是用于 UA 发送 fast 报文。可以把报文发送到硬件流水线，也可以发送给其他 UA。具体是发送到哪由参数 *pkt->um.dstmid* 决定。

## (6) 注销。

调用函数 *fast\_ua\_destroy* 实现 UA 注销操作和调用函数 *fast\_distroy\_hw* 释放硬件资源。

## UA 主函数 main 的伪代码示例

```
int main(int argc, char* argv[])
{
    int ret = 0;
```

```

/* 申请地址表存储空间*/
nm08_table = (struct nm08_neigh_table *)malloc(sizeof(struct
nm08_neigh_table));
    静态开辟存储空间

/* 空间清零*/
memset(nm08_table,0,sizeof(struct nm08_neigh_table));

/* 初始化平台硬件*/
fast_init_hw(0,0);

/*UA 模块初始化 */
if((ret=fast_ua_init(129,callback)))
{
    perror("fast_ua_init!\n");
    exit (ret);// 如果初始化失败,则需要打印失败信息,并将程序结束退出!
}

/*配置硬件规则, 将硬件所有报文送到模块ID 为129 的进程处理*/
init_rule(ACTION_SET_MID<<28|129);

/*启动地址学习表老化线程*/
nm08_start_aging();

/*启动线程接收分派给 UA 进程的报文*/
fast_ua_rcv();

/*主进程进入循环休眠中,数据处理主要在回调函数*/
while(1){sleep(9999);}

/*UA 注销操作*/
    fast_ua_destroy(void);

```

```
/*释放硬件资源*/  
  
fast_distroy_hw (void);  
  
return 0;  
  
}
```

### 说明:

① 函数 *fast\_ua\_init*、*fast\_ua\_recv*、*fast\_ua\_send*、*fast\_ua\_destroy* 为 UA 管理 API，源码实现在 *libua* 目录下面的 *main\_libua.c* 中。

一般来说，如果想使用 *fast* 编程去发送和接收报文，这几个函数都会被调用。首先调用函数 *fast\_ua\_init* 向系统注册自己的 *mid* 和设置处理接收报文的回调函数 *callback*。然后调用 *fast\_ua\_recv* 去接收报文数据，且对收到的报文需要做的处理全部在回调函数 *callback* 中实现。完成报文的处理之后，我们需要把报文再发送出去，调用函数 *fast\_ua\_send* 即可。当我们的 UA 程序退出时（包括正常和非法退出），调用函数 *fast\_ua\_destroy* 向内核告知自己将要结束，让内核清除此 MID 号上的报文转发功能。

② 函数 *fast\_init\_hw*、*fast\_distroy\_hw* 为硬件管理 API，源代码实现在 *libreg* 目录下面的 *mian\_libreg.c* 中。

如果涉及到下发规则操作或者读写寄存器，那么就必须调用硬件管理 API，且成对出现。另外，如果有 UA 初始化和注销动作，我们建议先硬件初始化再 UA 初始化。注销的时候则是先 UA 注销再硬件释放资源。

③ 函数 *init\_rule* 为规则管理 API，源代码实现在 *librule* 目录下面的 *main\_librule.c* 中。

## 2.6 扩展——流表规则

配置流表规则在整个 FAST UA 编程中很常用也很关键。下面一起回忆一下 l2switch 流表规则配置过程。

### ① 初始化规则

首先一开始 UA 程序 l2switch 运行的时候，我们调用初始化规则 API，执行代码

*init\_rule(ACTION\_SET\_MID<<28/129)*，把硬件流表规则清空，并且下发默认规则。此时 *default\_action* 是唯一的流表规则，所有报文都按照 *default\_action* 执行，即所有报文会被送至 mid 为 128 的 UA 程序 l2switch。

### ② 下发流表规则

当报文送至 UA 程序 l2switch 后，我们会对报文进行处理，学习 mac 地址、查找 mac 表、报文转发。对于未知的 mac 地址，报文送至 UA 去学习 mac 地址再把数据转发。而对于已经学习到 mac 地址的报文，虽然 UA 也可以实现查找 mac 表再转发报文，但是软件的处理毕竟如硬件速度快，所以建议通过下发流表规则的方式让硬件去实现该功能。

UA 怎么去下发流表规则呢？举个例子，假如在 2 号端口学到 mac 地址为 0x0023CD76631A。也就是说如果目的 mac 地址为 0x0023CD76631A，我们应该让报文从 2 号端口转发出去。

伪代码示例：

```
void rule_config ()
{
    Int inport = 2;

    //定义 rule 变量
    struct fast_rule rule = {{0}};

    //配置规则
    rule.key.dmac = htole64(0x0023CD76631A);
```

```
rule.mask.dmac = 0xFFFFFFFFFFFFL;

//匹配规则之后需要执行的动作
rule.action = ACTION_PORT<<28/inport;

//md5 值必须与其他流表项的md5 值不相同
rule.md5[0] = inport++;

//把流表规则添加到硬件中
fast_add_rule (&rule);
}
```

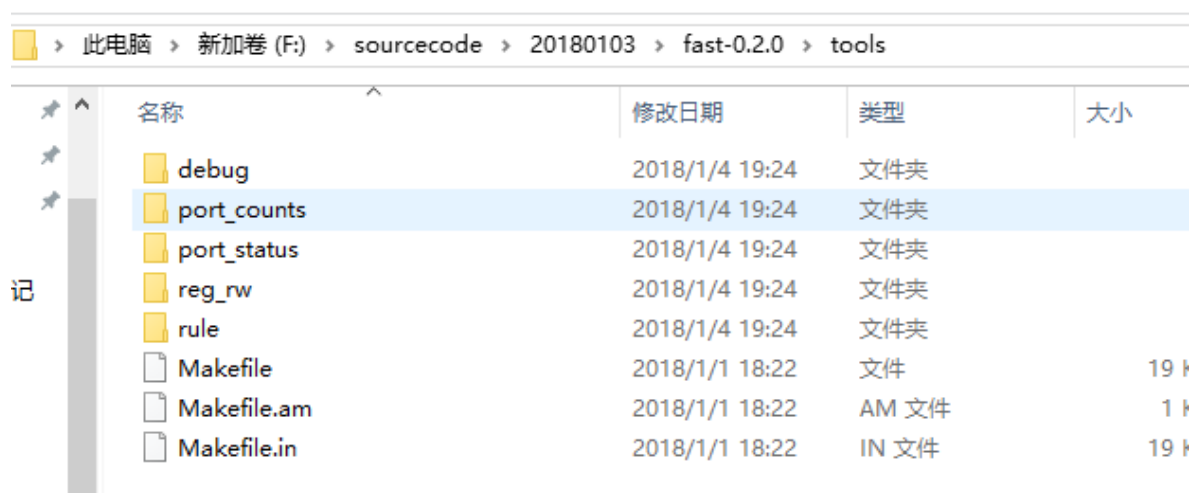
请注意。第一，key 与 mask 是对应的。key 为匹配关键字，mask 置为 1 表示对应的 key 项必须精确匹配。比如我们现在需要匹配目的 mac，所以需要配置 *rule.key.dmac*，而且还要把 *rule.mask.dmac* 置为 1。第二，流表规则是要写到硬件中，所以要注意大端小端问题。第三，md5 值要保证在流表中唯一。如果 md5 值与已存在的流表项 md5 值相同，硬件会认为该流表项已经存在了而写入硬件失败。

整理一下 l2switch 的流表规则。一开始 UA 程序运行调用初始化规则 API，把硬件所有规则清空并配置默认规则。所有报文都按照默认规则执行送至 mid 为 128 的 UA 程序 l2switch。然后 UA 程序学习到 mac 地址之后，下发 mac 表项规则。也就是说硬件中除了默认规则外，还有 mac 表项规则。新进来的报文首先匹配 mac 表项规则，若匹配成功则直接由硬件转发，若匹配失败则执行默认规则报文送至 UA 学习 mac 地址，致此，二层交换机功能就全部实现了。

### 3、tools 工具——寄存器读写操作案例详解

fast 源码目录下的 tools 目录，我们既可以把它当做 FAST UA 编程中寄存器读写操作的代码示例学习，也可以在代码测试的时候当做调试工具检验我们寄存器读写值是否正确。

如下所示为 tools 目录结构：



名称	修改日期	类型	大小
debug	2018/1/4 19:24	文件夹	
port_counts	2018/1/4 19:24	文件夹	
port_status	2018/1/4 19:24	文件夹	
reg_rw	2018/1/4 19:24	文件夹	
rule	2018/1/4 19:24	文件夹	
Makefile	2018/1/1 18:22	文件	19 B
Makefile.am	2018/1/1 18:22	AM 文件	1 B
Makefile.in	2018/1/1 18:22	IN 文件	19 B

Port\_counts 目录为硬件端口计数器访问示例，port\_status 目录为硬件端口状态访问示例，reg\_rw 目录为硬件寄存器读写访问示例，rule 目录为硬件流表规则打印示例。

#### 3.1 寄存器地址

在 OpenBox 系统中地址和数据位宽都是 64 位，但在实际工程中只用到了地址的低 19 位。

表 1： CDP 读写寄存器地址及其含义

寄存器地址	名称	类型	寄存器含义	默认值
0x40000	Address0	读/写	端口 0 基地址	16'b0
0x42000	Address1	读/写	端口 1 基地址	16'b0
0x44000	Address2	读/写	端口 2 基地址	16'b0
0x46000	Address3	读/写	端口 3 基地址	16'b0

表 2： 各个端口的偏移寄存器

寄存器地址	名称	类型	寄存器含义	初始值
0x1A	Frame_transmit	只读	发出的报文数目	8'b0



0x1B	Frame_receive	只读	接收的报文数目	8'b0
0x22	Errors	只读	接收的错误报文数目	8'b0
0x81 (PHY)	Link_status	只读	Link_Status[2]: 0-down, 1-up	1'b0

注意，软件调用时，需将表中地址左移 3 位，再加上对应端口的基地址。其中对于 0x81 偏移寄存器，需要先写地址寄存器 0x0f（同样需要左移 3 位再加上对应的端口基地址，写的值：该端口号值。），然后再读：将 0x81 左移 3 位，加上对应端口的基地址，即可读出该端口的 UP/Down 状态。

### 3.2 端口信息获取详解

端口信息获取是通过读取端口寄存器的取值实现的。对于寄存器的操作，必须先初始化硬件。也就是说寄存器的读写操作，必然会调用硬件管理 API 初始化资源函数 `fast_init_hw` 和释放资源函数 `fast_distroy_hw`。当然，如果没有涉及到 UA 管理相关操作，也就不需要调用 UA 初始化和 UA 注销 API。

对于软件 UA 开发者来说，看着寄存器地址就已经很头疼了，还要如上文所说的将地址左移 3 位再加上对应端口的基地址，这实在是太复杂了。所幸的是，fast 源码中 `fast_vaddr.h` 文件对常用的地址都进行了宏定义，用户需要读取某个端口的寄存器信息，不再需要自己写寄存器地址，可以直接使用宏。

```
#define FAST_PORT_BASE 0x40000 /**< 此为端口寄存器起始地址*/
#define FAST_PORT_OFT 0x2000 /**< 此为端口寄存器端口的偏移量*/

/*计数寄存器*/
#define FAST_COUNTS_SEND_OK 0x1A /*发送成功的计数*/
#define FAST_COUNTS_RECV_OK 0x1B /*接收成功的计数*/
#define FAST_COUNTS_CRC_ERR 0x1C /*CRC 错误的计算*/
#define FAST_COUNTS_ALIGNER 0x1D /*帧对齐错误*/
#define FAST_COUNTS_SEND_D0_OK 0x1E /*成功发送字节数低位计数*/
```

```

#define FAST_COUNTS_SEND_D1_OK 0x0F /*成功发送字节数高位计数*/
#define FAST_COUNTS_RECV_D0_OK 0x1F /*成功发送字节数低位计数*/
#define FAST_COUNTS_RECV_D1_OK 0x3D /*成功发送字节数高位计数*/
#define FAST_COUNTS_RCVFERR 0x22 /*接收的错误帧数*/
#define FAST_COUNTS_SDNFERR 0x23 /*发送的错误帧数*/
#define FAST_COUNTS_RCVSPKT 0x24 /*接收到的单播报文数*/
#define FAST_COUNTS_RCVMPKT 0x25 /*接收到的多播报文数*/
#define FAST_COUNTS_RCVBPKT 0x26 /*接收到的广播报文数*/
#define FAST_COUNTS_SNDSPKT 0x28 /*发送的单播报文数*/
#define FAST_COUNTS_SNDMPKT 0x29 /*发送的多播报文数*/
#define FAST_COUNTS_SNDBPKT 0x2A /*发送的广播报文数*/

/* 配置寄存器*/
#define FAST_PORT_MAC_CORE_CONFIG 0x2 /*MAC 核配置地址*/
#define FAST_PORT_MAC_0 0x3 /*MAC0 地址寄存器*/
#define FAST_PORT_MAC_1 0x4 /*MAC1 地址寄存器*/
#define FAST_PORT_FRAME_MAX_LEN 0x5 /*支持最大帧长度*/
#define FAST_PORT_BUF_LEVEL 0xE /*BUF LEVEL*/
#define FAST_PORT_FRAME_SPACE 0x17 /*帧间隔*/

/* 状态寄存器*/
#define FAST_PORT_PCS_MODE 0x94 /*PCS 模式*/
#define FAST_PORT_PCS_STATUS 0x81 /*PCS 状态*/
#define FAST_PORT_PCS_STATUS_LINK_OK (0x1<<2) /* LINK OK 状态*/
#define FAST_PORT_PCS_STATUS_AUTONEG_EN (0x1<<3) /*自协商使能状态*/
#define FAST_PORT_PCS_STATUS_AUTONEG_OK (0x1<<5) /*自协商成功状态*/
#define FAST_PORT_NEG_STATUS (0x85) /*自协商状态*/
#define FAST_PORT_NEG_STATUS_10M (0x0<<10) /*链路协商为10M*/
#define FAST_PORT_NEG_STATUS_100M (0x1<<10) /*链路协商为100M*/
#define FAST_PORT_NEG_STATUS_1G (0x2<<10) /*链路协商为1000M*/
#define FAST_PORT_NEG_STATUS_HALF (0x0<<12) /*链路为半双工状态*/
#define FAST_PORT_NEG_STATUS_FULL (0x1<<12) /*链路为全双工状态*/
#define FAST_PORT_NEG_STATUS_UP (0x1<<15) /*端口UP 状态*/
#define FAST_PORT_NEG_STATUS_DOWN (0x0<<15) /*端口DOWN 状态*/

```

端口计数寄存器的访问由 3 部分组成：

基地址|端口偏移地址\*端口序号|计数寄存器地址

REG=FAST\_PORT\_BASE|FAST\_PORT\_OFT\*port\_idx|XX\_REG;

XX\_REG 可以是计数寄存器、状态寄存器和端口的其他配置寄存

器的地址

所以上面宏定义的所有端口计数信息，我们都可以使用下面函数去获取。

```
/*参数port 为我们想要获取的是哪个端口号, 参数regaddr 为上面定义的任意一个宏*/  
u64 PORT_REG(int port,u64 regaddr)  
{  
    return fast_reg_rd(FAST_PORT_BASE/(FAST_PORT_OFT*port)/(regaddr<<3));  
}
```

例如，我们想获取 0 号端口的发送报文数和接收报文数，代码示例如下：

```
Int port = 0;  
PORT_REG(port,FAST_COUNTS_SEND_OK);  
PORT_REG(port,FAST_COUNTS_RECV_OK);
```

获取 0 号端口 UA 中主函数 main 的伪代码示例：

```
int main(int argc,char *argv[])  
{  
    int port = 0;  
  
    //初始化硬件资源  
    fast_init_hw(0,0);  
  
    //获取发送的单播报文数  
    PORT_REG(port,FAST_COUNTS_SNDSPKT),  
  
    //获取接收的单播报文数  
    PORT_REG(port,FAST_COUNTS_RCVSPKT),  
  
    //销毁硬件资源  
    fast_distroy_hw();  
    return 0;  
}
```

### 3.3 tools 工具的使用

有时候在代码测试中，需要调试工具检验我们寄存器读写值是否正确。Tools 目录下面有 4 个小工具，下面一一介绍一下其功能和使用方法。

注意，下面的小工具运行，都需要升至 root 权限，且要进入对应的目录。

#### (1) port\_counts

功能：打印硬件端口计数器信息

使用方法：如果是只想获取单个端口计数器信息需后面加端口，否则就是获取所有端口计数器信息。

如获取端口 0 的计数器信息，执行命令 `./port_counts 0`

```
root@kylin:/home/kylin/fast-0.2.0/tools/port_counts#  
root@kylin:/home/kylin/fast-0.2.0/tools/port_counts# ./port_counts 0  
Addr:0xF7280000, len:0x80000, 524288  
fastU->REG Version:1.0.0, OpenBox HW Version:20171215  
-----SHOW_PORT_COUNTS-----  
PORT  <Send OK> <SNDSPKT> <SNDMPKT> <SNDBPKT> | <Recv OK> <RCVSPKT> <RCVMPKT> <RCVBPKT> | <CRC ERR> <ALIGNER> <SNDERR> <RCVERR>  
0      0          0          0          0      | 0          0          0          0      | 0          0          0          0  
root@kylin:/home/kylin/fast-0.2.0/tools/port_counts#
```

如需获取所有端口计数器信息，执行命令 `./port_counts`

```
root@kylin:/home/kylin/fast-0.2.0/tools/port_counts#  
root@kylin:/home/kylin/fast-0.2.0/tools/port_counts# ./port_counts  
Addr:0xF7280000, len:0x80000, 524288  
fastU->REG Version:1.0.0, OpenBox HW Version:20171215  
-----SHOW_PORT_COUNTS-----  
PORT  <Send OK> <SNDSPKT> <SNDMPKT> <SNDBPKT> | <Recv OK> <RCVSPKT> <RCVMPKT> <RCVBPKT> | <CRC ERR> <ALIGNER> <SNDERR> <RCVERR>  
0      0          0          0          0      | 0          0          0          0      | 0          0          0          0  
1      0          0          0          0      | 0          0          0          0      | 0          0          0          0  
2      0          0          0          0      | 0          0          0          0      | 0          0          0          0  
3      0          0          0          0      | 0          0          0          0      | 0          0          0          0  
4      0          0          0          0      | 0          0          0          0      | 0          0          0          0  
5      0          0          0          0      | 0          0          0          0      | 0          0          0          0  
6      0          0          0          0      | 0          0          0          0      | 0          0          0          0  
7      0          0          0          0      | 0          0          0          0      | 0          0          0          0  
8      0          0          0          0      | 0          0          0          0      | 0          0          0          0  
9      0          0          0          0      | 0          0          0          0      | 0          0          0          0  
root@kylin:/home/kylin/fast-0.2.0/tools/port_counts#
```

#### (2) port\_status

功能：打印硬件端口状态信息

使用方法：如果是只想获取单个端口状态信息需后面加端口，否则就是获取所有端口状态信息。

如获取端口 0 的状态信息，执行命令 `./port_status 0`

```

root@kylin:/home/kylin/fast-0.2.0/tools/port_status# ./port_status 0
Addr:0xF7280000, len:0x80000, 524288
fastU->REG Version:1.0.0, OpenBox HW Version:20171215
-----SHOW_PORT_STATUS-----
PORT      <MAC ADDR> <FRAME LEN> <FRAME SPACE> <BUF LEVEL> <PCS MODE> <PCS STATUS> <NEG STATUS> <LINK> <AUTONEG EN> <AUTONEG> <SPEED> <DUPLEX>
0 0x008002986000 1800 0 3 0x7 0xAD 0x00005801 DOWN YES YES 1000 full
root@kylin:/home/kylin/fast-0.2.0/tools/port_status#

```

如需获取所有端口状态信息，执行命令./port\_status

```

root@kylin:/home/kylin/fast-0.2.0/tools/port_status# ./port_status
Addr:0xF7280000, len:0x80000, 524288
fastU->REG Version:1.0.0, OpenBox HW Version:20171215
-----SHOW_PORT_STATUS-----
PORT      <MAC ADDR> <FRAME LEN> <FRAME SPACE> <BUF LEVEL> <PCS MODE> <PCS STATUS> <NEG STATUS> <LINK> <AUTONEG EN> <AUTONEG> <SPEED> <DUPLEX>
0 0x008002986000 1800 0 3 0x7 0xAD 0x00005801 DOWN YES YES 1000 full
1 0x008002986001 1800 0 3 0x7 0xAD 0x00005801 DOWN YES YES 1000 full
2 0x008002986002 1800 0 3 0x7 0xAD 0x00005801 DOWN YES YES 1000 full
3 0x008002986003 1800 0 3 0x7 0xA9 0x00005801 DOWN YES YES 1000 full
4 0x008002986004 1800 0 3 0x7 0xAD 0x00005801 DOWN YES YES 1000 full
5 0x008002986005 1800 0 3 0x7 0xAD 0x00005801 DOWN YES YES 1000 full
6 0x008002986006 1800 0 3 0x7 0xAD 0x00005801 DOWN YES YES 1000 full
7 0x008002986007 1800 0 3 0x7 0xAD 0x00005801 DOWN YES YES 1000 full
8 0x00000000000000 0 0 0 0x0 0x00 0x00000000 DOWN NO NO 10 half
9 0x00000000000000 0 0 0 0x0 0x00 0x00000000 DOWN NO NO 10 half
root@kylin:/home/kylin/fast-0.2.0/tools/port_status#

```

### (3) reg\_rw

功能：硬件寄存器读写

使用方法：./reg\_rw {rd|wr} regaddr [regvalue]

Regaddr: 寄存器地址

Regvalue: 寄存器值

读寄存器：./reg\_rw rd regaddr

写寄存器：./reg\_rw wr regaddr regvalue

例如，读取寄存器 0x1A 的值，则执行命令./reg\_rw rd 0x1A

```

root@kylin:/home/kylin/fast-0.2.0/tools/reg_rw#
root@kylin:/home/kylin/fast-0.2.0/tools/reg_rw# ./reg_rw rd 0x1A
Addr:0xF7280000, len:0x80000, 524288
fastU->REG Version:1.0.0, OpenBox HW Version:20171215
REG READ ->Addr:0x1A = 0xFFFFFFFFFFFFFFFF
root@kylin:/home/kylin/fast-0.2.0/tools/reg_rw#
root@kylin:/home/kylin/fast-0.2.0/tools/reg_rw#

```

### (4) rule

功能：打印硬件规则

使用方法：./rule hw

打印硬件规则

```

root@kylin:/home/kylin/fast-0.2.0/tools/rule#
root@kylin:/home/kylin/fast-0.2.0/tools/rule# ./rule hw
Addr:0xF7280000,len:0x80000,524288
fastU->REG Version:1.0.0,OpenBox HW Version:20171215
-----SHOW_HW_RULE-----
-----Default Action:0x40000080-----
0x0000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
-----00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
-----00000000 00000000 Action:0x0
0x0001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
-----00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
-----00000000 00000000 Action:0x0
0x0002 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
-----00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
-----00000000 00000000 Action:0x0
0x0003 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

```

## 4、代码编译操作介绍

对于 UA 软件开发者来说，代码编译是少不了的操作。如果你已经学习了 fast UA 的编程文档，并且简单的写了一个 UA 小程序，想简单试试手；又或者你只是想试试 fast 源码中的简单示例执行效果，那该怎么做呢？请看下文。

以下所有的操作环境都是以 openbox 系统，irouter 设备为例。也是基于你已经会开机启动 irouter 设备，能 ssh 登录到 irouter 设备中为前提。如果还从来没有接触过 irouter 设备，请先阅读附录《OpenBox 操作指南》，了解怎么登录到 irouter 设备中后，再继续阅读下文。

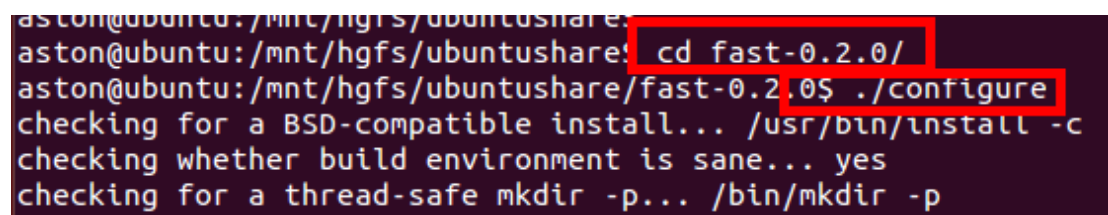
另外，就目前来说，fast UA 的代码编译可以在 Linux 虚拟机上面进行，但程序的运行目前还不支持。同时，fast 的编译环境要求是 64 位操作系统。所以，为了操作简单方便，建议编译环境直接是 irouter 设备。

### （1）把 fast 源码上传到 irouter 设备上

Fast 的源码获取途径可以直接在 fast 社区下载。把源码上传到 irouter 设备的方法也有多种。如果是 Linux 操作系统，可以使用 scp 命令。如果是 Windows 操作系统，可以使用 ssh 等工具。另外，也可以把 U 盘插到 irouter 设备的方式。不过目前 irouter 设备还不支持 U 盘自动挂载。最简单的操作方式，就是在 Windows 操作系统下面利用 ssh 等工具传输文件。

### （2）编译 fast 源码

① 执行命令：./configure，生成 Makefile 文件



```
aston@ubuntu:/mnt/hgfs/ubuntu-share: cd fast-0.2.0/  
aston@ubuntu:/mnt/hgfs/ubuntu-share/fast-0.2.0$ ./configure  
checking for a BSD-compatible install... /usr/bin/install -c  
checking whether build environment is sane... yes  
checking for a thread-safe mkdir -p... /bin/mkdir -p
```

请注意，目前 irouter 设备内置的 intltool 工具版本较低。如果 irouter

设备没有联网，就没办法自动更新 intltool 工具而导致编译失败的情况。可以下载较新的 intltool 安装包手动安装，但是由于此工具会有一些别的文件依赖关系，会比较麻烦，不建议此方法。

另一种简单可行性高一些的方法就是，可以找台能联网的电脑，把虚拟机中的 intltool 工具自动更新了，然后再把 fast 源码放到虚拟机中，执行 ./configure，生成 Makefile 文件。在把这整个含 Makefile 的 fast 源码上传到 irouter 设备中。如果已经在虚拟机中执行 configure 命令生成 Makefile 文件了，那么此步骤省略。

② 在 fast 源码的根目录下面执行 make 命令，生成 fast 的静态库

```
aston@ubuntu:/mnt/hgfs/ubuntu/share/fast-0.2.0$  
aston@ubuntu:/mnt/hgfs/ubuntu/share/fast-0.2.0$ make
```

到此，fast 的静态库已经生成了，并且一些示例应用程序也已经生成了。如果想试一下执行示例应用程序，可以进入对应的目录执行。如 tools 目录中有 port\_counts 和 port\_status，分别对应端口收发报文和端口状态的程序示例。

### (3) 自己编写的代码调试

如果自己写了一段 ua 代码实现了某个功能想要调试。有两种方式。

① 直接把自己的代码放到 src 目录下面编译。

如果只是很简单的一个 c 文件，可以直接把这个 c 文件放到 src 目录下面。然后把之前的原 main.c 文件删除或者重命名，再把你自己的文件名称改成 main.c，便可直接 make，进行编译。

如果自己的代码比较复杂，有多个文件和依赖关系，那就除了把自己的代码放到 src 目录下面外，还要修改 makefile 才能编译。请注意，fast 源码中所有的 makefile 是自动生成的，所以此时你也需要用工具自动生成你想要的 makefile。

② 自己写 makefile 编译

首先单独建一个目录，然后放入 fast 静态库、头文件，以及自己写的代码和 Makefile，再进入含有 makefile 的目录中执行 make 编



译。

```
aston@ubuntu:/mnt/hgfs/ubuntushare$ cd huangr
aston@ubuntu:/mnt/hgfs/ubuntushare/huangr$ ls
data_s  include  libreg  librule  libua
aston@ubuntu:/mnt/hgfs/ubuntushare/huangr$
```

```
aston@ubuntu:/mnt/hgfs/ubuntushare/huangr$ cd data_s/
aston@ubuntu:/mnt/hgfs/ubuntushare/huangr/data_s$ ls
data_s.h  main.c  Makefile  server.c
aston@ubuntu:/mnt/hgfs/ubuntushare/huangr/data_s$
```

# 附录

## 一、OpenBox 操作指南

### 1.1 串口的使用方式

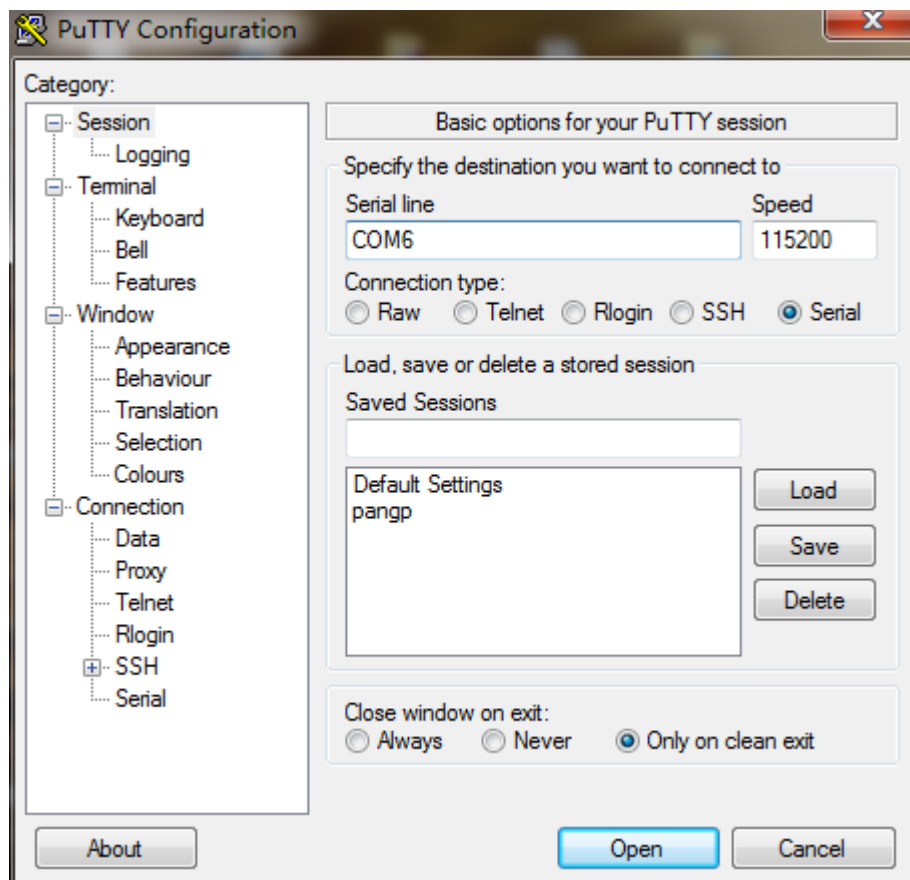
将串口线插入 USB 接口，另一端插入 openbox 的“console”口，然后右键“我的电脑”，选择“属性”：



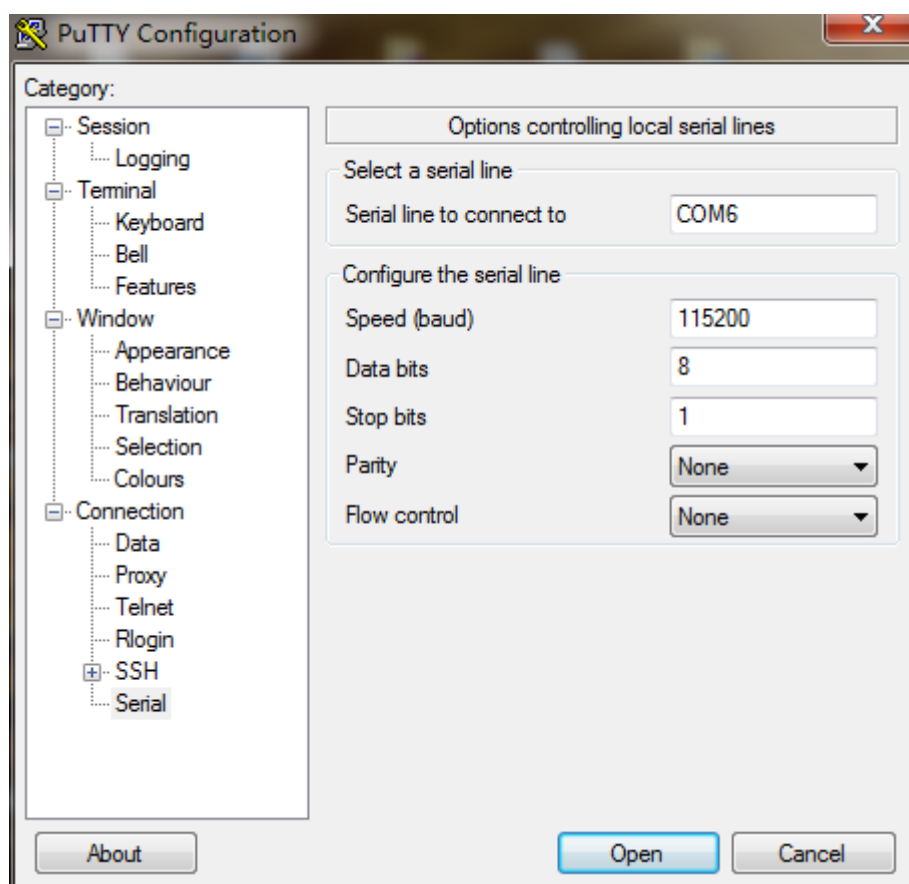
点击“设备管理器”，查看“端口”选项，查看当前使用的串口端口号（如使用我司提供的串口线，则设备名称应与下图一致）：



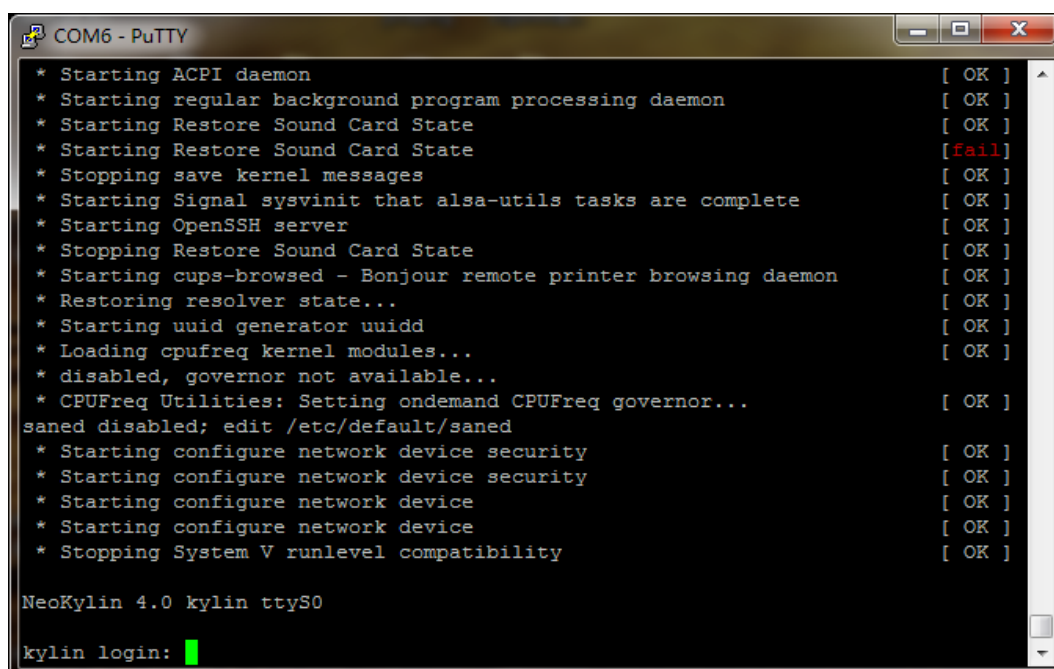
如上图所示为 COM6，然后打开 putty 软件，按照下图设置：



切换到“Serial”选项，如下设置：

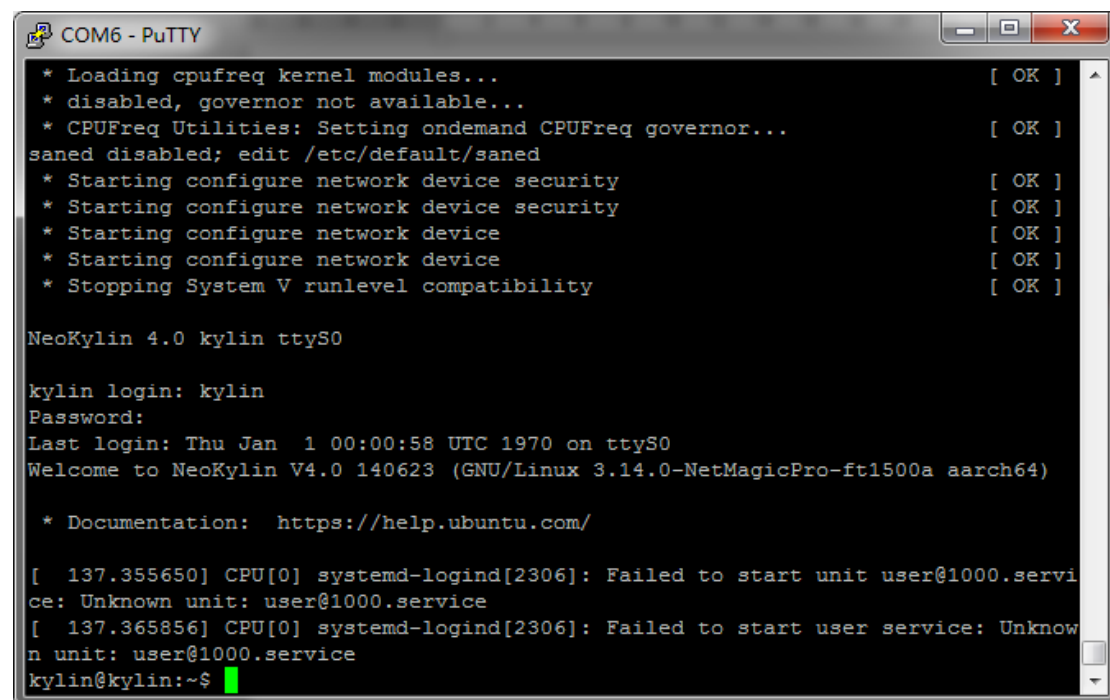


最后点击“Open”按钮，即可打开串口控制界面，等待片刻，即可进入 linux 登录窗口：



## 1.2 IP 地址的配置

首先,在串口窗口中输入用户名“kylin”(OpenBox 设备为 openbox),  
按下回车,然后输入密码“123123”(不含引号),按下回车,登录 linux  
系统:



```
COM6 - PuTTY
* Loading cpufreq kernel modules... [ OK ]
* disabled, governor not available...
* CPUFreq Utilities: Setting ondemand CPUFreq governor... [ OK ]
saned disabled; edit /etc/default/saned
* Starting configure network device security [ OK ]
* Starting configure network device security [ OK ]
* Starting configure network device [ OK ]
* Starting configure network device [ OK ]
* Stopping System V runlevel compatibility [ OK ]

NeoKylin 4.0 kylin ttyS0

kylin login: kylin
Password:
Last login: Thu Jan  1 00:00:58 UTC 1970 on ttyS0
Welcome to NeoKylin V4.0 140623 (GNU/Linux 3.14.0-NetMagicPro-ft1500a aarch64)

* Documentation:  https://help.ubuntu.com/

[ 137.355650] CPU[0] systemd-logind[2306]: Failed to start unit user@1000.servi
ce: Unknown unit: user@1000.service
[ 137.365856] CPU[0] systemd-logind[2306]: Failed to start user service: Unknow
n unit: user@1000.service
kylin@kylin:~$
```

输入命令“su”, 按下回车, 然后输入密码“123123”, 登入 Root 账  
户:

```
COM6 - PuTTY
* CPUFreq Utilities: Setting ondemand CPUFreq governor... [ OK ]
saned disabled; edit /etc/default/saned
* Starting configure network device security [ OK ]
* Starting configure network device security [ OK ]
* Starting configure network device [ OK ]
* Starting configure network device [ OK ]
* Stopping System V runlevel compatibility [ OK ]

NeoKylin 4.0 kylin ttyS0

kylin login: kylin
Password:
Last login: Thu Jan 1 00:00:58 UTC 1970 on ttyS0
Welcome to NeoKylin V4.0 140623 (GNU/Linux 3.14.0-NetMagicPro-ft1500a aarch64)

* Documentation: https://help.ubuntu.com/

[ 137.355650] CPU[0] systemd-logind[2306]: Failed to start unit user@1000.servi
ce: Unknown unit: user@1000.service
[ 137.365856] CPU[0] systemd-logind[2306]: Failed to start user service: Unknow
n unit: user@1000.service
kylin@kylin:~$ su
密码:
root@kylin:/home/kylin#
```

输入命令“ifconfig -a”，查看当前可用的网络接口（不同设备可能存在差异）：

```
COM6 - PuTTY
root@kylin:/home/kylin# ifconfig -a
eth0      Link encap:以太网  硬件地址 02:08:25:5b:f0:15
          BROADCAST MULTICAST  MTU:1500  跃点数:1
          接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:0 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:0 (0.0 B)  发送字节:0 (0.0 B)
          中断:12

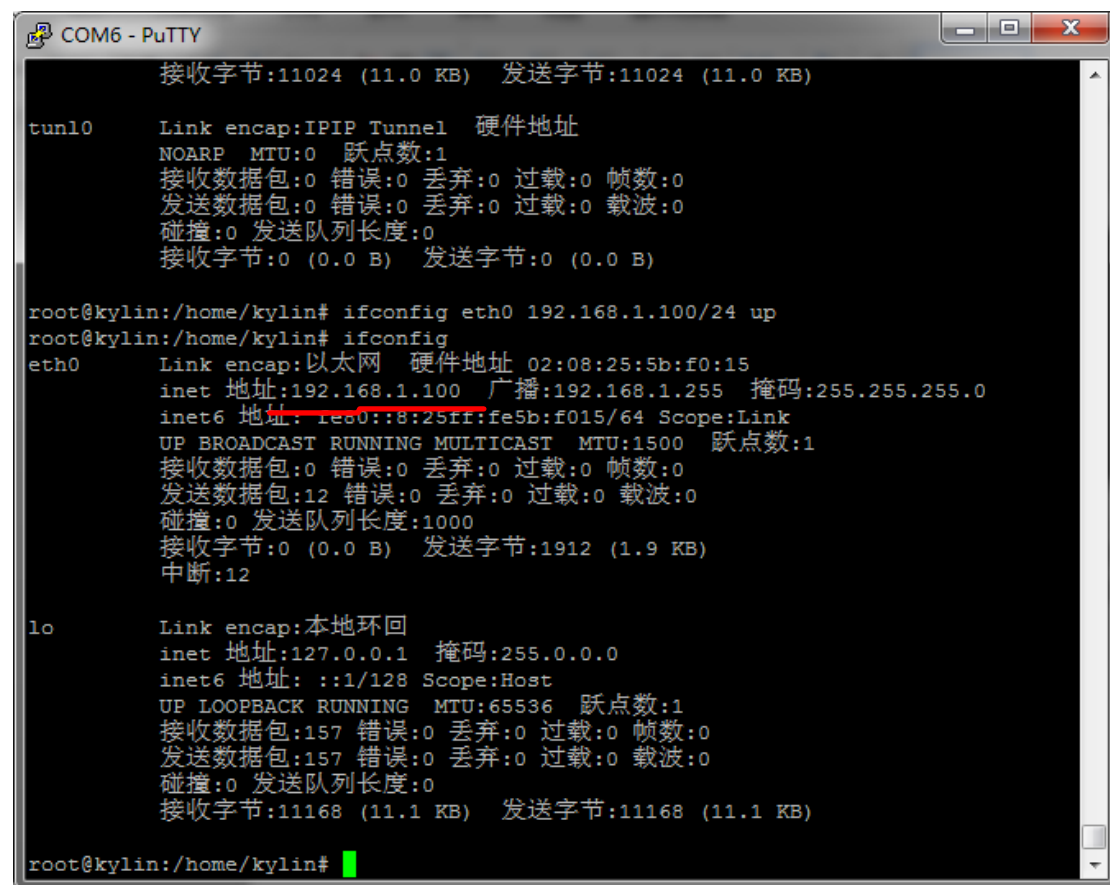
eth1      Link encap:以太网  硬件地址 4e:bd:d3:a2:b3:77
          BROADCAST MULTICAST  MTU:1500  跃点数:1
          接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:0 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:0 (0.0 B)  发送字节:0 (0.0 B)
          中断:13 基本地址:0x4000

lo        Link encap:本地环回
          inet 地址:127.0.0.1  掩码:255.0.0.0
          inet6 地址: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536 跃点数:1
          接收数据包:155 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:155 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:0
          接收字节:11024 (11.0 KB)  发送字节:11024 (11.0 KB)

tunl0     Link encap:IPIP Tunnel  硬件地址
          NOARP  MTU:0  跃点数:1
          接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:0 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:0
```

接下来设置接口“eth0”（即 OpenBox 设备的 MGMT 口）的 IP 地

址，输入命令 “ifconfig eth0 192.168.1.100/24 up”（其中 IP 地址可以更换为用户自己预先分配的其他地址），按下回车执行命令，然后执行命令“ifconfig”，即可查看 eth0 接口是否被分配上了 IP 地址（如果发现网络中断的情况，可以查看 IP 地址是否失效，如失效，使用上述步骤，重新设置 IP 地址即可）：



```
COM6 - PuTTY
接收字节:11024 (11.0 KB) 发送字节:11024 (11.0 KB)

tunl0    Link encap:IP/IP Tunnel  硬件地址
         NOARP  MTU:0  跃点数:1
         接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
         发送数据包:0 错误:0 丢弃:0 过载:0 载波:0
         碰撞:0 发送队列长度:0
         接收字节:0 (0.0 B) 发送字节:0 (0.0 B)

root@kylin:/home/kylin# ifconfig eth0 192.168.1.100/24 up
root@kylin:/home/kylin# ifconfig
eth0     Link encap:以太网  硬件地址 02:08:25:5b:f0:15
         inet 地址:192.168.1.100 广播:192.168.1.255 掩码:255.255.255.0
         inet6 地址: fe80::8:25ff:fe5b:f015/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500 跃点数:1
         接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
         发送数据包:12 错误:0 丢弃:0 过载:0 载波:0
         碰撞:0 发送队列长度:1000
         接收字节:0 (0.0 B) 发送字节:1912 (1.9 KB)
         中断:12

lo       Link encap:本地环回
         inet 地址:127.0.0.1 掩码:255.0.0.0
         inet6 地址: ::1/128 Scope:Host
         UP LOOPBACK RUNNING  MTU:65536 跃点数:1
         接收数据包:157 错误:0 丢弃:0 过载:0 帧数:0
         发送数据包:157 错误:0 丢弃:0 过载:0 载波:0
         碰撞:0 发送队列长度:0
         接收字节:11168 (11.1 KB) 发送字节:11168 (11.1 KB)

root@kylin:/home/kylin#
```

### 1.3 固定 eth 的 IP 地址

按照上一节的方式，配置好 ip 地址之后，设备重启后会失效。如果想固定 IP 地址，可以修改配置文件。例如要设置 eth 的 ip 地址为 192.168.1.8，在 `/etc/network/interfaces` 文件中修改如下三行对应的参数：

*address 192.168.1.8*

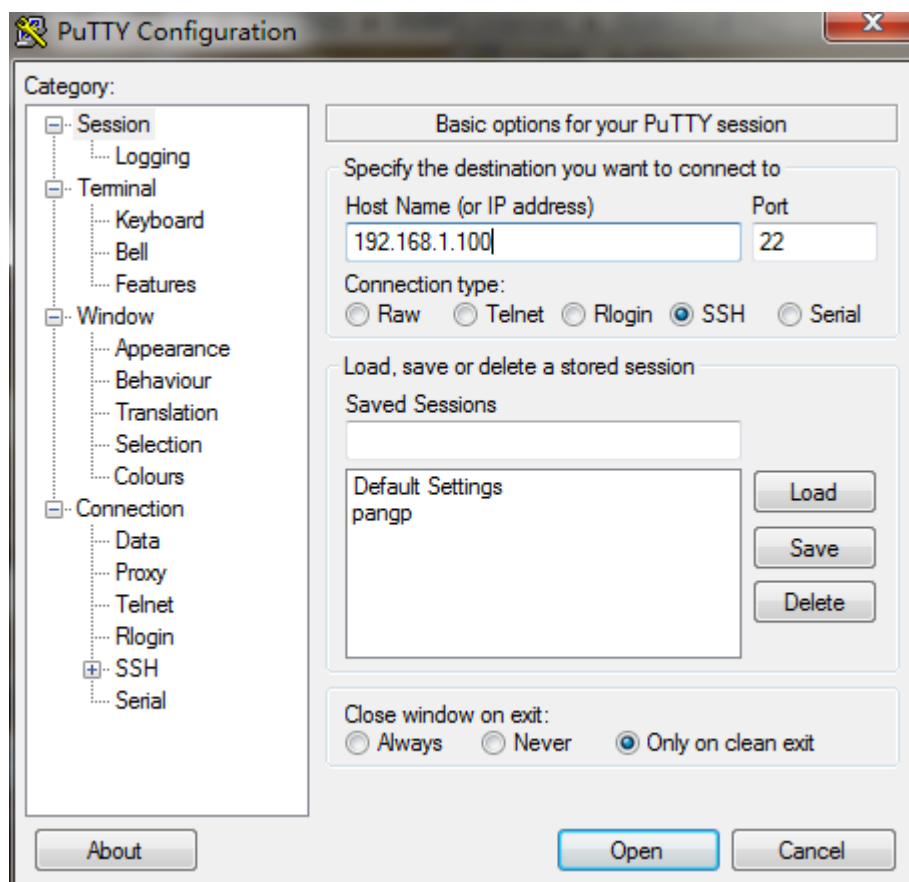
*netmask 255.255.255.0*

*gateway 192.168.1.1*

请注意，有些设备中的*/etc/rc.local* 脚本也有 *ipv4* 的配置，请把有关 *ipv4* 的配置在 */etc/rc.local* 脚本中删除，统一由 */etc/network/interfaces* 文件控制 *eth* 的 *ip* 地址。

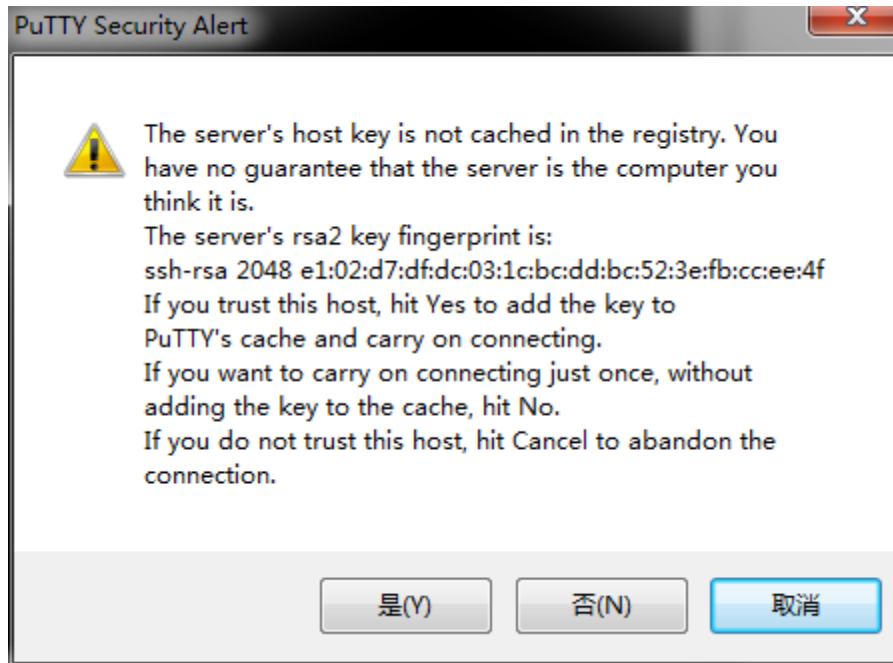
## 1.4 ssh 远程登录

设置好 IP 地址之后，即可使用 *ssh* 的方式远程登录设备进行操作，首先准备一台 PC 机，将其 IP 地址设置为与 *eth0* 相同的网段，确保可以 *ping* 通设备的 *eth0* 口，如发现无法 *ping* 通，请检查 PC 机的网络设置以及 *eth0* 的 IP 地址是否失效，然后在 PC 机上打开 *putty* 软件：按下图设置：

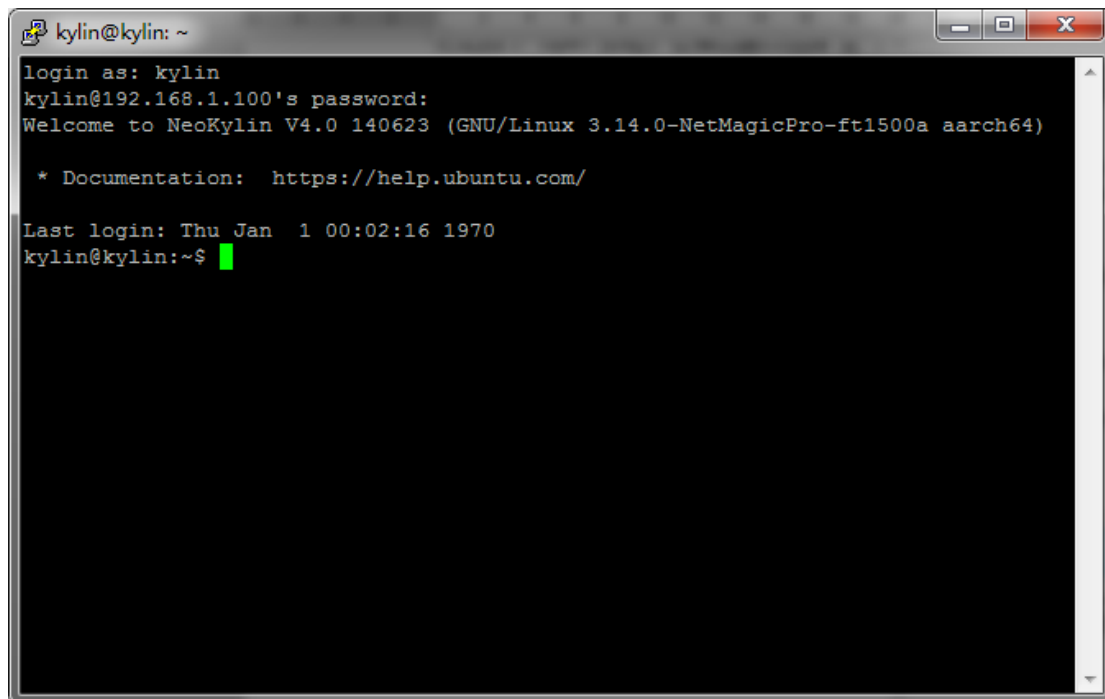


然后按下“Open”按钮，在弹出窗口选择“是”：





然后输入用户名和密码（OpenBox 设备用户名为 openbox），即可连接 openbox 进行远程操作：



## 1.5 固件烧录 rbf 的文件

需要的文件：烧录工具 *xdebug*、*openbox.ko*，硬件版本 *xx.rbf* 文件。  
示例中硬件版本文件为 *OpenBox\_1215.rbf*。

烧录步骤:

1. 把所需要的文件全部上传到 *irouter* 设备中。例如本示例把 *xdebug* 和 *openbox.ko* 文件放在 */home/irouter/xzp* 目录下, *OpenBox\_1215.rbf* 文件放在 */home/irouter/xzp/1215* 目录下。
2. 切换 root 权限, 命令: *su* , 码 123123;
3. 进入到 */home/irouter/* 目录下, 将两个烧录工具文件改为可执行权限, 命令 (root 权限下): *chmod a+x xdebug openbox.ko*。
4. 查看是否原来有 *openbox* 的驱动加载, 命令: *lsmod* , 如果有 *openbox* 驱动, 使用命令: *rmmmod openbox* 卸载驱动, 在 */home/irouter/xzp* 目录下执行 *insmod openbox.ko* 加载 */home/irouter/xzp* 目录下的 *openbox.ko* 驱动。
5. 进入 1215 目录内, 记录该目录内 *OpenBox\_1215.rbf* 文件的绝对路径 (以后要输入该文件绝对路径, 例如: */home/irouter/xzp/1215/OpenBox\_1215.rbf*)
6. 进入 */home/irouter/xzp* 目录下, 执行 *./xdebug*, 会弹出如下界面

```
irouter@iRouter: /xzp$ ./xdebug
help      :input value
0         :写寄存器。输入:寄存器地址, 要写入的值。
1         :读寄存器。输入:寄存器地址。返回:寄存器的值。
2         :RAM写测试。
3         :RAM读测试。
4         :读取指定虚拟地址位置开始的一段内存信息。输入:虚拟地址(hex), 读取长度。
5         :读取指定物理地址位置开始的一段内存信息。输入:物理地址(hex), 读取长度。
6         :读取指定DMA地址位置开始的一段内存信息。输入:DMA地址(hex:21位值), 读取长度。-
7         :参数测试
8         :写虚拟地址开始的内存位置。输入:虚拟地址, 偏移位置, 写入的值。
9         :写物理地址开始的内存位置。输入:物理地址, 偏移位置, 写入的值。
10        :写DMA地址开始的内存位置。输入:DMA地址, 偏移位置, 写入的值。
11        :-
12        :修改软件轮循线程状态。
13        :固化FPGA代码
14        :退出程序。

input:█
```

7. 在 input 下输入 13
8. 在读写模式下输入 0

```
scanf:0, input:13,      固化FPGA代码
请选择读写模式 (0: 写, 1: 读):█
```

9. 在输入文件名下输入步骤 4 记录的路径

```
请输入文件名:/home/irouter/xzp/1215/OpenBox_1215.rbf
```

- 10.回车。如果发现如下错误，请先执行命令 `mknod /dev/npe_debug c 319 0` 加载设备节点，再重新执行 6-9 步骤。

```
请输入文件名:/home/irouter/xzp/1215/OpenBox_1215.rbf
filename=/home/irouter/xzp/1215/OpenBox_1215.rbf
Open /dev/npe_debug Error!
: Permission denied
irouter@iRouter: ~/xzp$
```

11. 如果没有出现步骤 10 错误，回车后就会固化驱动，第一次一般会固化失败，重新执行 7-9 步骤即可。
12. 完成烧录后 `ctrl+c` 推出执行 `dmesg -c` 查看烧录结果，最后出现打印 ok 即可
13. 在 root 权限下执行 `poweroff`，然后断电重启即可烧录完成。

## 二、常见问题解答

**Q1: 预定义的 MID 号的含义。**

A1:当前 MID 的划分分为硬件 MID 以及软件 MID，具体关系如下：

硬件 MID: 0-127 为硬件 MID，其中 1-5 对应当前通用查表 UM 中的 5 个模块，1 对应 UDP，2 对应 UKE，3 对应 GME，4 对应 UDA，5 对应 GOE。其中 GOE 为通用转发模块，当 `dstmid=5` 时，报文会被直接送往 GOE 模块，在提取 metadata 中的 `outport` 字段之后，从相应的物理端口发送出来。

软件 MID: 128-255 为软件 MID，其中 128 对应系统的协议栈，其他 MID 为用户可自由使用的 MID，如果用户的 UA 需要将报文送往协议栈或者其他的 UA，只需将报文的 `um_metadata` 中 `dstmid` 置为 128 或者其他 UA 注册时的 MID，然后调用 `fast_ua_send()` 函数发送报文，系统底层驱动就会自动将报文送往对应的对象。

**Q2: 协议栈和 UA 是并行处理的还是互斥关系，为什么会导致不同 UA 中的 `socket` 通信无法完成。**

**A2:** 据我们观察，当用户不需要用流表做加速而仅仅需要使用 UA 作分析功能时，往往会使用 `init_rule()` 函数将硬件流表中的默认 ACTION 修改为送往相应 MID 的 UA 作处理，但是当 UA 处理完报文之后，无法将报文送给协议栈，导致基于协议栈通信的 `socket` 无法正常工作。过去的驱动对于 `dstmid=128` 的报文没有作特殊处理，而系统中的 UA 往往不被允许使用 128 作为 `dstmid`，所以当报文的 `dstmid` 被置为 128 时，往往会因为找不到对应的 UA、而被丢弃。现在的驱动由于新增了对于 `dstmid` 为 128 的报文的特殊处理，会将其送往系统的协议栈，所以 `socket` 也能够正常的工作了。

**Q3: metadata 和 flow 里的一些字段，默认值是什么？设为不同的值代表什么意思？**

**A3:** `flow` 中的字段在使用 `init_rule()` 函数之后会全部清零，具体每个字段的含义建议查看头文件；而 `metadata` 的值在硬件收到报文转送给 UA 时，有些部分会被硬件自动填写好，其中包括时间戳 `ts`、输入端口号 `inport`、序列号 `seq` 字段，具体含义也建议查看头文件。

**Q4: metadata 数据结构对于每一条报文的初始值，是什么时候被填写好的，是用户自己填写还是底层根据解析的报文自己填写。**

**A4:** 硬件填写的部分可以参照上一个问题，用户可以在 UA 中修改其余 `metadata` 的内容。