**mini2440 dm9000 网卡驱动详解**

本文分成以下几个部分：

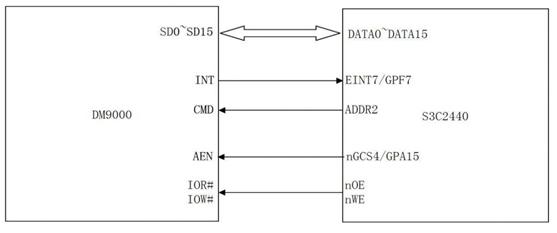
一、Mini2440开发板上DM9000的电气连接和Mach-mini2440.c文件的关系。

二、两个重要的结构体介绍：sk\_buff和net\_device

三、具体[代码](http://www.xuebuyuan.com/" \o "代码" \t "_blank)分析

一、Mini2440开发板上DM9000的电气连接和Mach-mini2440.c文件的关系

Mini2440开发板上DM9000与S3C2440的连接关系如下：



连接状况：

=================================

LnOE(IOR#):读命令;

LnWE(IOW#):写命令;

nLAN\_CS(AEN):地址使能,低电平有效.当S3C2440访问DM9000这颗IC对应的内存地址时,会把此引脚自动拉低;

nRESET(PW\_RST#):DM9000的硬件复位,它和CPU及一些外设的复位引脚接一起,起硬件复位作用.低电平有效;

LADDR2(CMD):数据地址的片选引脚;

IRQ\_LAN(INT):中断引脚;

LDATA[0:15](SD[0:15]):数据地址引脚.

==================================

这个DM9000平台设备作为众多平台设备中的一个在扳子初始化的时候就被添加到了总线上。代码清单如下：

其中片选信号AEN使用了nGCS4，所以网卡的内存区域在BANK4，也就是从地址0x20000000开始-0x27FFFFFF。DM9000的TXD[2:0]作为strap pin在电路图中是空接的，所以IO base是300H。中断使用了EINT7。

由于DM9000连接在bank存储区，因此面向Processor而言,是一个迷你型的一小块SRAM.主控实现DM9000内部寄存器的访问便是对这小块SRAM的形式访问,而不像一些常见IC是通过IIC、IIS、SPI等形式访问.对照MINI2440开发板,会发现其硬件连接方式和NOR FLASH很相近,

基本上每一颗IC都要暴露一些其内部寄存器的操作手段,比如IIC、SPI等.而DM9000则是以RAM的形式被主控操作其内部寄存器的：

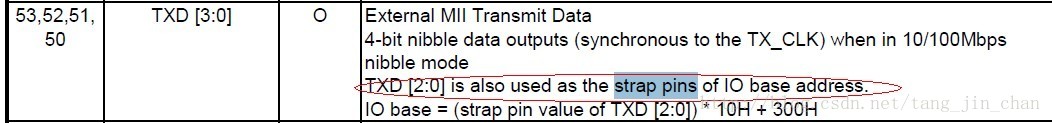
如控制DM9000的软件复位、唤醒、获取VID和PID等动作.

不同于常规IC通过IIC、SPI等常见接口,看看这颗DM9000是如何以RAM的形式被主控访问的.

    2-1.基地址(0x300):

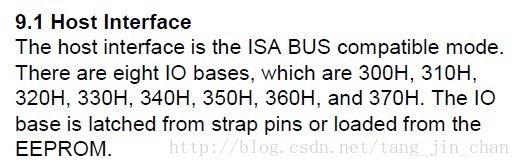
    DM9000内部寄存器并不是直接放在CPU可访问的0地址开始的,而是基于CPU可访问的0地址的0x300的位置处.DM9000一共可以支持8个基地址:300H、310H、320H、330H、340H、350H、360H和370H.有两种方式去决定其基地址:

一种是"strap pins",即TXD[2:0].如下:



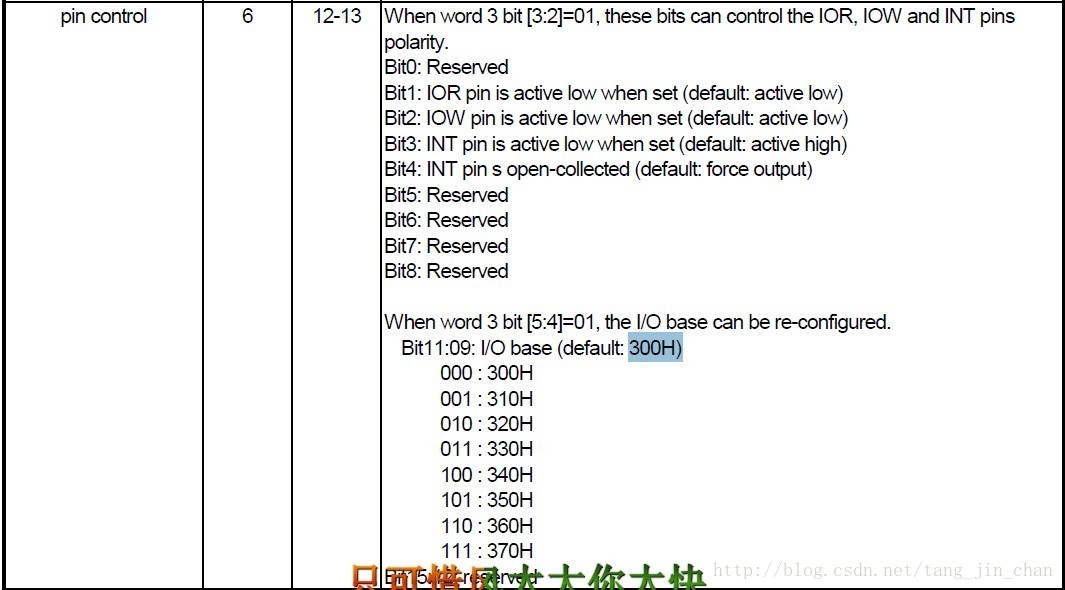
    MINI2440上面的DM9000的TXD[2:0]为空.

    另一种就是从DM9000上面挂载的EEPROM直接LOAD进去的.如下:



     MINI2440上面的DM9000没有挂载EEPROM.

    默认的基地址为0x300,如下:



这些内容在Mach文件中有如下体现：

#define S3C2410\_CS4 (0x20000000)

#define MACH\_MINI2440\_DM9K\_BASE (S3C2410\_CS4 + 0x300) =0x200000300

Mach-s3c2440.c中设备添加流程：

MACHINE\_START(MINI2440, "FriendlyARM Mini2440 development board")

/\* Maintainer: Michel Pollet <buserror@gmail.com> \*/

.phys\_io = S3C2410\_PA\_UART,

.io\_pg\_offst = (((u32)S3C24XX\_VA\_UART) >> 18) & 0xfffc,

.boot\_params = S3C2410\_SDRAM\_PA + 0x100,

.map\_io = mini2440\_map\_io,

.init\_machine = mini2440\_init, /\*初始化函数\*/

.init\_irq = s3c24xx\_init\_irq,

.timer = &s3c24xx\_timer,

MACHINE\_END

static void \_\_init mini2440\_init(void)

platform\_add\_devices(mini2440\_devices, ARRAY\_SIZE(mini2440\_devices));

1:

/\* devices we initialise \*/

static struct platform\_device \*mini2440\_devices[] \_\_initdata = {

……………….

&mini2440\_device\_eth,

&s3c24xx\_uda134x,

………………..

};

2:

static struct platform\_device mini2440\_device\_eth \_\_initdata = {

.name = "dm9000",

.id = -1,

.num\_resources = ARRAY\_SIZE(mini2440\_dm9k\_resource),

.resource = mini2440\_dm9k\_resource,

.dev = {

.platform\_data = &mini2440\_dm9k\_pdata,

},

};

3:

static struct resource mini2440\_dm9k\_resource[] \_\_initdata = {

[0] = {

.start = MACH\_MINI2440\_DM9K\_BASE, 0x200000300

.end = MACH\_MINI2440\_DM9K\_BASE + 3, 0x200000303

.flags = IORESOURCE\_MEM

},

[1] = {

.start = MACH\_MINI2440\_DM9K\_BASE + 4, 0x200000304

.end = MACH\_MINI2440\_DM9K\_BASE + 7, 0x200000307

.flags = IORESOURCE\_MEM

},

[2] = {

.start = IRQ\_EINT7,

.end = IRQ\_EINT7,

.flags = IORESOURCE\_IRQ | IORESOURCE\_IRQ\_HIGHEDGE,

}

};

另外在Mach文件中还定义了DM9000平台设备，设备名称为“dm9000”，设备资源就是上面定义的IO和中断资源。代码清单如下：

static struct dm9000\_plat\_data mini2440\_dm9k\_pdata \_\_initdata = {

.flags = (DM9000\_PLATF\_16BITONLY | DM9000\_PLATF\_NO\_EEPROM),

};

二、两个重要的结构体简单介绍：sk\_buff和net\_device

 \*sk\_buff

    如果把网络传输看成是运送货物的话，那么sk\_buff就是这个“货物”了，所有经手这个货物的人都要干点什么事儿，要么加个包装，要么印个戳儿等等。收货的时候就要拆掉这些包装，得到我们需要的货物（payload data）。没有货物你还运输什么呢？由此可见sk\_buff的重要性了。关于sk\_buff的详细介绍和几个操作它的函数，参考：“linux内核sk\_buff的结构分析” http://www.linuxidc.com/Linux/2011-07/39163.htm，写得非常明白了。赞一个~

 \*net\_device

    又是一个庞大的结构体。好吧，我承认我从来就没有看全过这个结构体。它在内核中就是指代了一个网络设备。驱动程序需要在探测的时候分配并初始化这个结构体，然后使用register\_netdev来注册它，这样就可以把操作硬件的函数与内核挂接在一起。

三、具体代码的分析

   在顺序分析之前先看三个结构体变量和一个自定义的结构体。

   \* dm9000\_driver变量。是platform\_driver结构体变量，其中包含了重要的：驱动的名字（用来match）和几个重要操作函数。

static struct platform\_driver dm9000\_driver = {

.driver = {

.name = "dm9000",

.owner = THIS\_MODULE,

},

.probe = dm9000\_probe,

.remove = \_\_devexit\_p(dm9000\_drv\_remove),

.suspend = dm9000\_drv\_suspend,

.resume = dm9000\_drv\_resume,

};

\* dm9000\_netdev\_ops变量。是net\_device\_ops结构体变量， 其中定义了操作net\_device的重要函数，我们在驱动程序中根据需要的操作要填充这些函数。代码清单如下：

static const struct net\_device\_ops dm9000\_netdev\_ops = {

.ndo\_open = dm9000\_open,

.ndo\_stop = dm9000\_stop,

.ndo\_start\_xmit = dm9000\_start\_xmit,

.ndo\_tx\_timeout = dm9000\_timeout,

.ndo\_set\_multicast\_list = dm9000\_hash\_table,

.ndo\_do\_ioctl = dm9000\_ioctl,

.ndo\_change\_mtu = eth\_change\_mtu,

.ndo\_validate\_addr = eth\_validate\_addr,

.ndo\_set\_mac\_address = eth\_mac\_addr,

#ifdef CONFIG\_NET\_POLL\_CONTROLLER

.ndo\_poll\_controller = dm9000\_poll\_controller,

#endif

};

 \* dm9000\_ethtool\_ops变量。是ethtool\_ops结构体变量，为了支持ethtool，其中的函数主要是用于查询和设置网卡参数（当然也有的驱动程序可能不支持ethtool）。代码清单如下：

static const struct ethtool\_ops dm9000\_ethtool\_ops = {

.get\_drvinfo = dm9000\_get\_drvinfo,

.get\_settings = dm9000\_get\_settings,

.set\_settings = dm9000\_set\_settings,

.get\_msglevel = dm9000\_get\_msglevel,

.set\_msglevel = dm9000\_set\_msglevel,

.nway\_reset = dm9000\_nway\_reset,

.get\_link = dm9000\_get\_link,

.get\_eeprom\_len = dm9000\_get\_eeprom\_len,

.get\_eeprom = dm9000\_get\_eeprom,

.set\_eeprom = dm9000\_set\_eeprom,

};

\*board\_info结构体。用来保存芯片相关的一些私有信息。具体在代码中分析。下面是这个结构体的清单。

下面看一下具体代码。

分析代码还是从init顺序开始。

    1. 注册平台驱动。

    主要完成的任务是：将驱动添加到总线上，完成驱动和设备的match，并执行驱动的probe函数。代码清单如下：

static struct platform\_driver dm9000\_driver = {

.driver = {

.name = "dm9000", /\*用这个名字完成驱动和设备的match\*/

.owner = THIS\_MODULE,

},

.probe = dm9000\_probe,

.remove = \_\_devexit\_p(dm9000\_drv\_remove),

.suspend = dm9000\_drv\_suspend,

.resume = dm9000\_drv\_resume,

};

static int \_\_init

dm9000\_init(void)

{

printk(KERN\_INFO "%s Ethernet Driver, V%s\n", CARDNAME, DRV\_VERSION);

return platform\_driver\_register(&dm9000\_driver);

}

 2.probe函数。

   主要完成的任务是：探测设备获得并保存资源信息，根据这些信息申请内存和中断，最后调用register\_netdev注册这个网络设备。以下是代码清单，可以分成几个部分来看：

   1)首先定义了几个局部变量：

        struct dm9000\_plat\_data \*pdata = pdev->dev.platform\_data;

==========================================================

pdata指向的是:

static struct dm9000\_plat\_data mini2440\_dm9k\_pdata \_\_initdata = {

.flags = (DM9000\_PLATF\_16BITONLY | DM9000\_PLATF\_NO\_EEPROM),

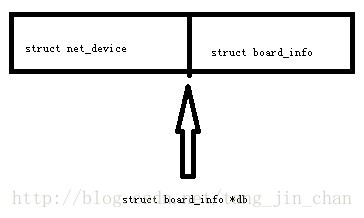
};

=============================================================

struct board\_info \*db; /\* Point a board information structure \*/

平台相关的私有结构,是指向具体平台的信息体.对于内核而言,一个完整的以太网设备的内核表征有两大部分组成:struct net\_device + struct board\_info.如下图所示:

db是专门指向board\_info的起始位置的



        struct net\_device \*ndev;

=====================================

probe中的参数含义：

static int \_\_devinit dm9000\_probe(struct platform\_device \*pdev)

 pdev即为平台设备,即

static struct platform\_device mini2440\_device\_eth \_\_initdata = {

.name = "dm9000",

.id = -1,

.num\_resources = ARRAY\_SIZE(mini2440\_dm9k\_resource),

.resource = mini2440\_dm9k\_resource,

.dev = {

.platform\_data = &mini2440\_dm9k\_pdata,

},

};

===========================================

   2)初始化一个网络设备。关键系统函数：alloc\_etherdev()

   3)获得资源信息并将其保存在board\_info变量db中。

关键系统函数：netdev\_priv(), platform\_get\_resource()

   4)根据资源信息分配内存，申请中断等等, 并将申请后的资源信息也保存到db中，并且填充ndev中的参数。关键系统函数：request\_mem\_region(), ioremap()。自定义函数:dm9000\_set\_io()

   5)完成了第4步以后，回顾一下db和ndev中都有了什么：

      struct board\_info \*db:

                 addr\_res -- 地址资源

                 data\_res -- 数据资源

                 irq\_res    -- 中断资源

                 addr\_req -- 分配的地址内存资源

                 io\_addr   -- 寄存器I/O基地址

                 data\_req -- 分配的数据内存资源

                 io\_data   -- 数据I/O基地址

                 dumpblk  -- IO模式

                 outblk     -- IO模式

                 inblk        -- IO模式

                 lock         -- 自旋锁（已经被初始化）

                 addr\_lock -- 互斥锁（已经被初始化）

       struct net\_device \*ndev：

                 base\_addr  -- 设备IO地址

                irq              -- 设备IRQ号

    6) 设备复位。硬件操作函数dm9000\_reset()

    7) 读一下生产商和制造商的ID，应该是0x9000 0A46。关键函数：ior()

    8) 读一下芯片类型。

    ========以上步骤结束后我们可以认为已经找到了DM9000========

    9) 借助ether\_setup()函数来部分初始化ndev。因为对以太网设备来讲，很多操作与属性是固定的，内核可以帮助完成。

   10) 手动初始化ndev的ops和db的mii部分。

   11) （如果有的话）从EEPROM中读取节点地址。这里可以看到mini2440这个板子上没有为DM9000外挂EEPROM，所以读取出来的全部是0xff。见函数dm9000\_read\_eeprom。关于外挂EEPROM，可以参考datasheet上的7.EEPROM Format一节。

   12)  很显然ndev是我们在probe函数中定义的局部变量，如果我想在其他地方使用它怎么办呢？ 这就需要把它保存起来。内核提供了这个方法，使用函数platform\_set\_drvdata()可以将ndev保存成平台总线设备的私有数据。以后再要使用它时只需调用platform\_get\_drvdata()就可以了。

13) 使用register\_netdev()注册ndev。

**网络设备数据传输：**

在字符设备当中,我们比较习惯的一种思维是"一切皆文件".

一般字符设备硬件：

首先去open一个设备节点.底层的操作集直接或间接对应用户空间的系统调用的API.

但是,在网络子系统里面,我们去操作一个PHY时,并没有直接去操作所谓的节点,

如eth0.比如我们通过TFTP下载一个文件到目标机器的时候,只需要执行下面的命令即可:

1. tftp -r YourFile -g 192.168.1.131

    上述的IP是PC端的IP.但是我们在这里并没有看到eth0任何相关信息.其实,我用要用一个网卡的时候,是需要配置的,如下面命令:

1. ifconfig eth0 192.168.1.22 up

其实,我们每用一个网卡的时候,都是配置一下网卡的.配置网卡的结果是内核网络子系统添加一条路由消息,这条路由消息是和网卡对应的.

网络设备硬件：

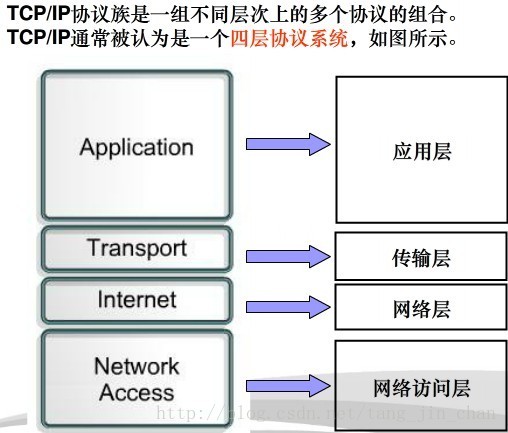
用户空间跟内核网络子系统及网卡驱动交互便是通过TCP/IP协议栈里面的路由算法来找到我们相应的物理设备的,而不是像字符设备那样直接操作和物理设备对应的设备节点.它只是多了一层转换而已.

因此,当我们执行下面的命令时:

1. tftp -r YourFile -g 192.168.1.131

    实际上,首先tftp命令根据参数产生一条路由消息陷入内核根据TCP/IP协议栈找到其网络传递路径最终实现到具体PHY的交互.然后调用具体PHY的设备驱动函数来实现硬件的操作.

    1.TCP/IP四层模型及数据流向:



    各层的意义如下:

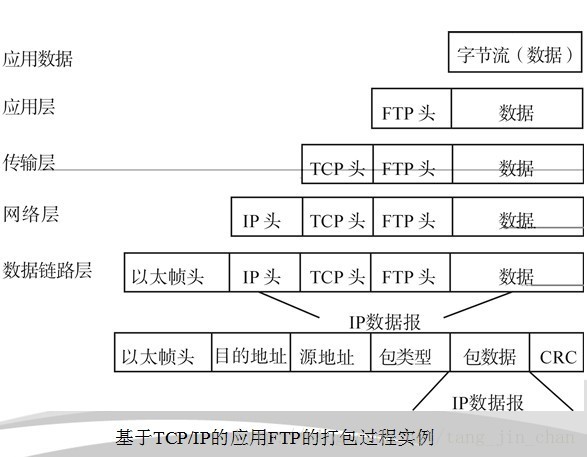
1. 应用层（application layer）：包含各种网络应用协议。如HTTP、FTP、telnet、SMTP、DNS、SNMP等协议。
2. 传输层（transport layer）：负责在源主机和目的主机的应用程序间提供端-端的数据传输服务。主要有TCP和UDP两个传输协议。
3. 网络层（internet layer）：负责将分组独立地从信源传送到信宿，主要解决路由选择、拥塞控制和网络互联等问题。如最重要的协议——IP。
4. 网络接口层（network access layer）：负责将IP分组封装成帧格式并传输；或将从物理网络接收到的帧解封，取出IP分组交给网络互联层。当前几乎所有的物理网络上都可运行TCP/IP协议。

当我们从PC下载一个文件到目标机器的时候,对PC端而言,自上而下"层层封装";到达目标机器的时候,自下而上,"层层解封".这过程从大的方面来讲包括两方面:一方向是不断地添加每个层相应的协议头;另一方面是对实际有效的数据各层的不同处理,例如传输层将应用程序的数据流分成TCP段,并加上TCP包头递交给网络层.

    实例:

下面以使用TCP协议传送文件（如FTP应用程序）为例说明了TCP/IP的工作原理:  
1). 在源主机上,应用层将一串字节流传给传输层;  
2). 传输层将字节流分成TCP段,加上TCP包头交给互联网络（IP）层;  
3). IP层生成一个包,将TCP段放入其数据域,并加上源和目的主机的IP地址,然后交给数据链路层;   
4). 数据链路层在其帧的数据部封装IP包,发往目的主机或IP路由器;

流程示意图如下:



5). 在目的主机上,数据链路层将数据链路层帧头去掉,将IP包交给互联网层;  
6). IP层检查IP包头,如果包头中的校验和与计算出来的不一致,则丢弃该包;  
7). 如果校验和一致,IP层去掉IP头,将TCP段交给TCP层,TCP层检查顺序号来判断是否为正确的TCP段;  
8). TCP层为TCP包头计算TCP头和数据.如果不正确,TCP层丢弃这个包,若正确,则向源主机发送确认;  
9). 在目的主机上,TCP层去掉TCP头,将字节流传给应用程序;  
10). 目的主机收到了自源主机发来的字节流,就像直接从源主机发来的一样.

2.struct sk\_buff:

    如果说net\_device是静态地表征内核一个"网络设备",而不管是具体哪一种网络设备或者说哪个PHY,类比于USB子系统的usb\_device.而struct sk\_buff则是流窜于内核网络子系统的动态"精灵",上述的数据流程便是每层对SKB的的装拆封过程,其意义不亚于USB子系统的URB.内核网络子系统以SKB为核心,根据不同网络层的协议,对这个"核心精灵数据"有不同的操作集,如skb\_put()、skb\_push()等动作.

    2-1.skb的基本定义:

    skb的基本定义位于include/linux/skbbuff.h,包括其流窜过程的各网络层的协议头、实际有效数据域及管理以及各网络层对应的不同的操作集.

    2-1-1.各网络层的协议头:

    sk\_buff结构体定义了3个协议头对应网络协议的不同层次,这三个层次分别是传输层TCP/UDP(及ICMP和IGMP)协议头、网络层协议头和链路层协议头.如下:

1. sk\_buff\_data\_t      transport\_header;
2. sk\_buff\_data\_t      network\_header;
3. sk\_buff\_data\_t      mac\_header;

    2-1-2.实际有效数据域:

    当我们通过网络来传输数据时,比如是一个文件,那么这个文件是我们的有效数据.这个文件路由不同的网络层会被标识成某个网络层的数据包.比如说,对传输层而言,文件及传输层的协议头是其有效数据.skb中用数据缓冲区指针head、data、tail和end.如下:

1. /\* These elements must be at the end, see alloc\_skb() for details.  \*/
2. sk\_buff\_data\_t      tail;
3. sk\_buff\_data\_t      end;
4. unsigned char       \*head,
5. \*data;

    head:指向内存已分配的用于承载网络数据缓冲区的起始地址;

    data:指向当前协议层有效数据的起始地址,这里需要注意每个协议层的有效数据的含义并不一样.如下:

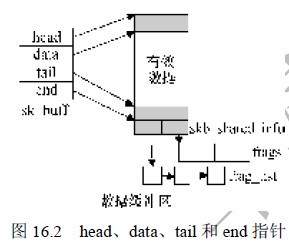
1. 传输层:用户数据和传输层协议头属于有效数据;
2. 网络层:用户数据、传输层协议头和网络层协议头是其有效数据;
3. 数据链路层:用户数据、传输层协议头、网络层协议头和链路层头部是其有效数据.

    因此,data指针将随着各协议层的不同而相应移动.

    tail:指向当前协议层有效数据负载的结尾地址,与data指针相对应;

    end:指向内存分配的数据缓冲区的结尾,与head指针相对应.当skb被分配之后,end指针也就固定不变了.它还包含一个skb\_shared\_info结构体的空间,这个结构体存放分隔存储的数据片段,即可以将数据包的有效数据分成几片存储在不同的内存空间中.

    head、data、tail和end的分布图如下:



    2-1-3.skb操作集:

    由上述2-1-2可知,路由各层协议时,需要不断地装拆封skb,操作集主要包括skb的分配和释放、head,data,tail,end指针在不同协议层的移动等动作.详细可参看宋宝华<<Linux设备驱动开发详解>>网络设备驱动章节.