

```

    }
    return 0;
}

int main(void)
{
    int i;
    struct input_event data;
    if(init_device()<0)
        return -1;

    for(;;){
        read(ts_fd,&data,siaeof(data));           //不停地读入数据
        if(data.type != EN_ABS)                 //是否是绝对坐标设备
            printf("wrong type:%d\n",data.type);
        printf("event=%s,value=%d\n",data.code==ABS_X? "ABS_X" : data.code
              ==ABS_Y? "ABS_Y": data.code==ABS_PRESSURE? "ABS_PRESSURE": "unkn-
              own",data.value);                  //打印数据
    }
}

```

## 7.6 小结

本章主要分析了 S3C2440 ADC 控制寄存器的硬件操作和内核 Input 子系统驱动源程序，并在此基础上讲述了 s3c2440\_ts.c 驱动程序的移植。触摸屏驱动程序移植主要是要了解内核中 Input 子系统体系结构及如何操作 LCD 控制器，因此本章的 7.2 节和 7.4 节是后面移植工作的基础，读者要认真掌握。

# 第 8 章 USB 设备驱动移植

USB (Universal Serial Bus, 通用串行总线)，是由 Compaq、DEC、IBM、Intel、NEC、Microsoft 及 Northern Telecom 等公司于 1994 年 11 月共同提出的，主要目的是为了制定统一的 USB 标准。USB 设备使用起来比较方便，其文件传输速率快，而且具有热插拔性能，它的应用越来越广泛，目前与 PC 连接的外围设备基本都具有 USB 接口。常见的 USB 设备包括 USB 鼠标、USB 键盘、USB 摄像头、USB 打印机、U 盘等。本章将主要结合代码详细介绍 USB 协议，并且讲解几种常见 USB 设备的移植步骤。

## 8.1 USB 协议

在 Linux 下进行 USB 设备驱动移植时，首先需要对 USB 协议有初步的了解。本节主要讲解 USB 协议的系统主要组成部分、USB 系统总线的拓扑结构、内部层次关系、数据流模式、USB 的调度等。

### 8.1.1 USB 协议的系统主要组成部分

USB 系统主要可分为 3 个部分：USB 的连接部分、USB 的设备和 USB 的主机。其分层模型如图 8.1 所示。

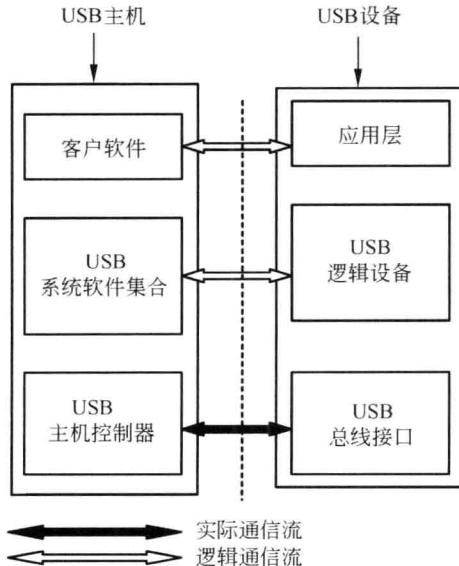


图 8.1 USB 主机、设备分层模型

USB 主机包括 USB 主机控制器、USB 系统软件集合、客户软件。其中，USB 系统软件集合由 USB 驱动程序、主机控制器的驱动程序和主机软件构成。USB 设备包括 USB 总线接口、USB 逻辑设备、应用层。

在 Linux 内核中，USB 设备采用结构体 `usb_device` 表示，该结构体的定义如下：

```
struct usb_device {
    int      devnum;                      //设备号，在 USB 总线上的地址
    char     devpath [16];                 //用于传递消息的 ID 字符串
    enum usb_device_state   state;        //设备状态：连接态、加电态、默认态、编址态、配置态
    enum usb_device_speed  speed;         //设备速度类型，全速、低速、高速其中一种
    struct usb_tt  *tt;                   //事务转换信息
    int      ttport;                     //事务转换 HUB 的设备端口
    unsigned int toggle[2];              //每个端点用一个 bit 表示，[0]表示 IN，[1]表示 OUT
    struct usb_device *parent;           //USB 设备的父结点均为 HUB，根结点没有父结点
    struct usb_bus *bus;                //USB 总线，USB 设备所在的总线
    struct usb_host_endpoint ep0;       //端点 0，默认的控制端点
    struct device dev;                  //通用设备接口
    struct usb_device_descriptor descriptor; //USB 设备描述符
    struct usb_host_config *config;     //所有的设备配置
    struct usb_host_config *actconfig;  //活动的配置
    struct usb_host_endpoint *ep_in[16]; //从设备到主机的端点
    struct usb_host_endpoint *ep_out[16]; //从主机到设备的端点
    char **rawdescriptors;             //每个配置的原始描述
    unsigned short bus_mA;             //当前可用
    u8 portnum;                       //父端口数
    u8 level;                          //USB、HUB 祖先数
    unsigned can_submit:1;              //可以被提交的 URB
    unsigned discon_suspended:1;        //挂起时断开
    unsigned persist_enabled:1;         //该 USB 设备使能 USB_PERSIST
    unsigned have_langid:1;            //tring_langid 是否有效
    unsigned authorized:1;             //权限
    unsigned authenticated:1;          //通过鉴权
    unsigned wusb:1;                   //无线 USB 设备
    int string_langid;                //字符串语言标识
    /*下面是设备的固有属性，包括产品 ID，生产字符串，生产序列号等*/
    char *product;
    char *manufacturer;
    char *serial;
    struct list_head filelist;
#ifndef CONFIG_USB_DEVICE_CLASS
    struct device *usb_classdev;        //USB 类设备
#endif
#ifndef CONFIG_USB_DEVICEFS
    struct dentry *usbfs_dentry;
#endif
    int maxchild;                      //最多能接的子设备个数，也就是 HUB 的端口数
    struct usb_device *children[USB_MAXCHILDREN]; //接在 HUB 上的子设备
    int pm_usage_cnt;                  //使用计数
    u32 quirks;
    atomic_t urbnum;                  //提交的 URB 数目
    unsigned long active_duration;
```

```

#define CONFIG_PM
    struct delayed_work autosuspend;
    struct work_struct autoresume;
    struct mutex pm_mutex;

    unsigned long last_busy;
    int autosuspend_delay;
    unsigned long connect_time;           //设备第一次连接时间

    unsigned auto_pm:1;
    unsigned do_remote_wakeup:1;          //使能远程唤醒
    unsigned reset_resume:1;              //重设而不是重启
    unsigned autosuspend_disabled:1;      //用户禁止自动挂起
    unsigned autoresume_disabled:1;       //用户禁止自动重启
    unsigned skip_sys_resume:1;          //跳过下一个系统重启
#endif
    struct wusb_dev *wusb_dev;           //如果是无线设备，则为设备连接 WUSB 专门数据
};

```

### 8.1.2 总线物理拓扑结构

USB系统中的主机和设备采用的是星形连接方式，其物理连接拓扑图如图8.2所示。

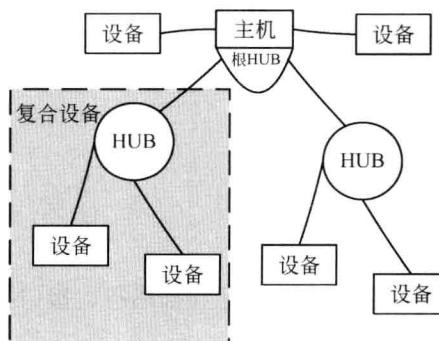


图8.2 USB物理总线的拓扑

图8.2中的HUB是一类特殊的USB设备，它是一组USB的连接点，主机中有一个被嵌入的HUB叫根HUB（root Hub），主机通过根HUB提供多个连接点。为了防止出现环状连接，连接点采用星形连接来体现层次性。

### 8.1.3 USB设备、配置、接口、端点

在USB设备的逻辑组织分层结构中，包含设备、配置、接口和端点4个层次。

每个USB设备都提供了不同级别的配置信息，可以包含一个或多个配置，不同的配置使设备表现出不同的功能组合（在探测/连接期间需从其中选定一个）；每个配置则由多个接口组成；接口由多个端点组成，每个接口代表一个基本的功能，是USB设备驱动程序控制的对象，一个复杂的USB设备可以具有多个接口。

端点是USB通信的最基本形式，对主机来说，每一个USB设备接口就是一组端点的集合。主机只有通过端点才能和设备进行通信，以使用设备的功能。在USB系统中每个端

点都有独一无二的地址，该地址由设备地址和端点号指定。每个端点都有一组属性，其中包括传输方式、总线访问频率、带宽、端点号和数据包的最大容量等。一个 USB 端点只能在一个方向承载数据，或者从主机到设备（称为输出端点），或者从设备到主机（称为输入端点），因此端点传输是单向的。端点 0 通常为控制端点，用于初始化设备参数。只要设备连接到 USB 上并且上电，端点 0 就可以被访问。端点 1、2 等一般用做数据端点，存放主机与设备间通信的数据。

USB 设备非常复杂，由许多不同的逻辑单元组成，如图 8.3 所示。

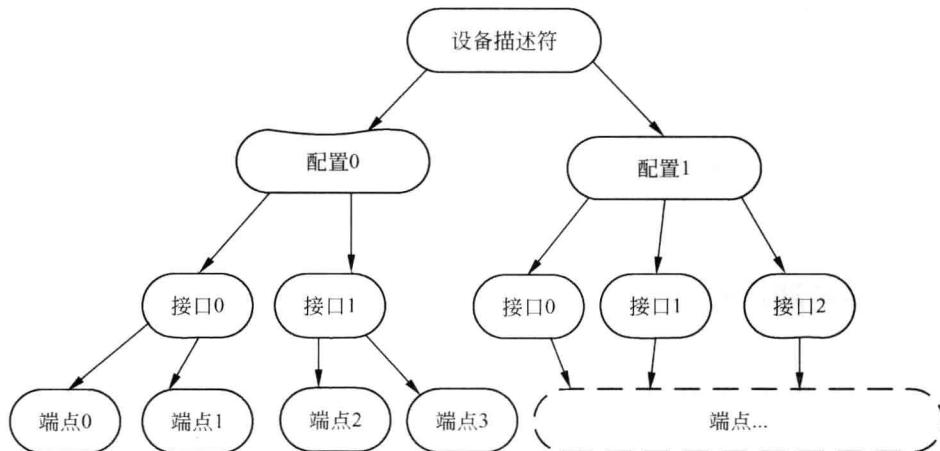


图 8.3 USB 设备、配置、接口和端点

- 设备通常有一个或多个配置；
- 配置通常有一个或多个接口；
- 接口通常有一个或多个设置；
- 接口有 0 个或多个端点。

下面分别结合代码来介绍设备描述符、配置描述符、接口描述符、端点描述符和字符串描述符。

## 1. 设备描述符

设备描述符是关于设备的通用信息，包括供应商 ID、产品 ID 和修订 ID、支持的设备类、子类和适用的协议，以及默认端点的最大包大小等。在 Linux 内核中，USB 设备用 `usb_device` 结构体来描述，USB 设备描述符被定义为 `usb_device_descriptor` 结构体，该结构体的定义代码如下：

```

struct usb_device_descriptor {
    __u8 bLength;           // 描述符长度
    __u8 bDescriptorType;   // 描述符类型编号

    __le16 bcdUSB;          // USB 版本号
    __u8 bDeviceClass;      // USB 分配的设备类 code
    __u8 bDeviceSubClass;   // USB 分配的子类 code
    __u8 bDeviceProtocol;   // USB 分配的协议 code
    __u8 bMaxPacketSize0;   // endpoint0 最大包大小

    __le16 idVendor;        // 厂商编号
  
```

```

    __le16 idProduct;           //产品编号
    __le16 bcdDevice;          //设备出厂编号
    __u8 iManufacturer;        //描述厂商字符串的索引
    __u8 iProduct;             //描述产品字符串的索引
    __u8 iSerialNumber;         //描述设备序列号字符串的索引
    __u8 bNumConfigurations;   //可能的配置数量
} __attribute__ ((packed));

```

## 2. 配置描述符

配置描述符中包括该配置中的接口数、支持的挂起和恢复能力，以及功率要求。USB 配置描述符在内核中被定义为 `usb_host_config` 结构体，结构体 `usb_config_descriptor` 的定义代码如下：

```

struct usb_config_descriptor {
    __u8 bLength;           //描述符长度
    __u8 bDescriptorType;    //描述符类型编号

    __le16 wTotalLength;     //配置所返回的所有数据的大小
    __u8 bNumInterfaces;    //配置所支持的接口数
    __u8 bConfigurationValue; //Set_Configuration 命令需要的参数值
    __u8 iConfiguration;    //描述该配置字符串的索引值
    __u8 bmAttributes;      //供电模式的选择
    __u8 bMaxPower;          //设备从总线提取的最大电流
} __attribute__ ((packed));

```

## 3. 接口描述符

接口描述符中包括接口类、子类和适用的协议，接口备用配置的数目和端点数目。USB 接口描述符在内核中被定义为 `usb_interface` 结构体，结构体 `usb_interface_descriptor` 的定义代码如下：

```

struct usb_interface_descriptor {
    __u8 bLength;           //描述符长度
    __u8 bDescriptorType;    //描述符类型
    __u8 bInterfaceNumber;   //接口的编号
    __u8 bAlternateSetting;  //备用的接口描述符编号
    __u8 bNumEndpoints;      //该接口使用的端点数，不包括端点 0
    __u8 bInterfaceClass;    //接口类型
    __u8 bInterfaceSubClass; //接口子类型
    __u8 bInterfaceProtocol; //接口所遵循的协议
    __u8 iInterface;         //描述该接口的字符串索引值
} __attribute__ ((packed));

```

## 4. 端点描述符

端点描述符中包括端点地址、方向和类型、支持的最大包大小。如果端点为中断类型，则端点描述符中还包括轮询频率。在 Linux 内核中，USB 端点被定义为 `usb_host_endpoint` 结构体，`usb_endpoint_descriptor` 结构体的代码定义如下：

```

struct usb_endpoint_descriptor {
    __u8 bLength;           //描述符长度
    __u8 bDescriptorType;   //描述符类型

    __u8 bEndpointAddress; //端点地址:0~3位是端点号,第7位是方向(0-OUT,1-IN)
    /*端点属性: bit[0:1] 的值为 00 表示控制, 为 01 表示同步, 为 02 表示批量, 为 03 表示
    中断*/
    __u8 bmAttributes;
    __le16 wMaxPacketSize; //本端点接收或发送的最大信息包的大小
    __u8 bInterval;        //轮询数据传送端点的时间间隔
                           //对于批量传送的端点及控制传送的端点, 此域忽略
                           //对于同步传送的端点, 此域必须为 1
                           //对于中断传送的端点, 此域值的范围为 1~255
    /*注意: 下面两个字段仅在音频端点中使用*/
    __u8 bRefresh;
    __u8 bSynchAddress;
} __attribute__((packed));

```

## 5. 字符串描述符

字符串描述符的功能是在其他描述符中为某些字段提供字符串索引, 用来检索描述性字符串, 可以采用多种语言形式提供。字符串描述符是可选的, 有些设备具有该描述符, 而另外一些设备则可能没有该描述符, 字符串描述符被定义为 `usb_string_descriptor` 结构体, `usb_string_descriptor` 结构体的定义代码如下:

```

struct usb_string_descriptor {
    __u8 bLength;           //描述符长度
    __u8 bDescriptorType;   //描述符类型
    __le16 wData[1];        //UTF-16LE 编码
} __attribute__((packed));

```

### 8.1.4 USB 设备状态

USB 设备状态分为 6 种状态, 分别是连接态、加电态、默认态、编址态、配置态和挂起态。各个状态之间的状态转移关系如图 8.4 所示。

- 加电态: USB 设备的电源可来自外部电源, 也可从 USB 接口的集线器而来。电源来自外部电源的 USB 设备被称做自给电源式的 (self-powered) USB 设备。尽管自给电源式的 USB 设备在连接上 USB 接口前可能已经处于带电状态, 但它们连接到 USB 接口后才能被看做是加电状态 (Powered state)。
- 默认态: 设备加电以后, 在从总线接收到复位信号前不应对总线传输发生响应。只有设备在接收到复位信号之后, 才能在默认地址处变为可寻址。
- 编址态: 在加电复位后所有的 USB 设备都使用默认地址与主机通信。每个设备在连接或复位后由主机分配一个唯一的地址。当 USB 设备被挂起时, 它保持这个地址不变。

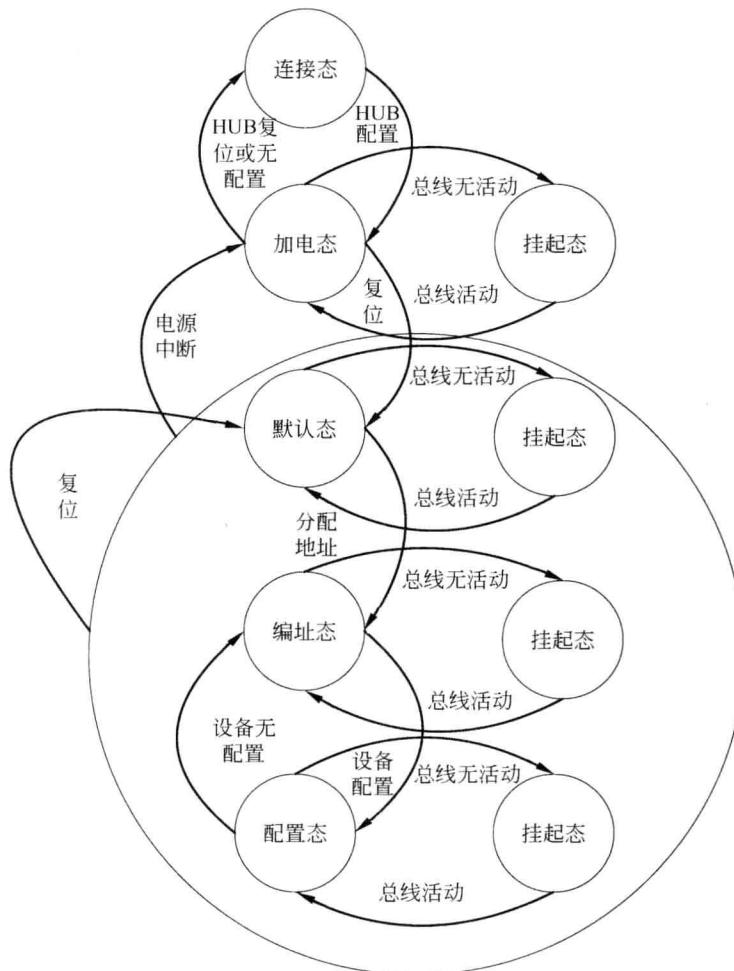


图 8.4 USB 设备状态转化图

- 配置态：在 USB 设备正常工作以前，设备必须被正确配置。从设备的角度来看，配置包括将非 0 值写入设备配置寄存器的操作。配置一个设备或改变一个可变的设备设置，会使得与这个相关接口的终端结点的所有状态与配置值被设成默认值。
- 挂起态：为了省电，USB 设备在探测不到总线传输时自动进入中止状态。当中止时，USB 设备保持本身的内部状态不变，包括它的地址及配置。

### 8.1.5 USB 枚举过程

了解 USB 设备状态后，进一步结合代码和序列图了解 USB 的枚举过程。USB 的枚举过程如图 8.5 所示。主机集线器监视每个端口的信号电压，当有新设备接入时就能被检测到。当 USB 设备插入到 HUB 后，便开始 USB 的枚举过程。USB 的枚举过程主要分为以下几个步骤。

(1) `Get_Port_Status`: 主机发现最新接入的设备，返回消息告诉主机新接入的设备是何时连接到集线器上的。

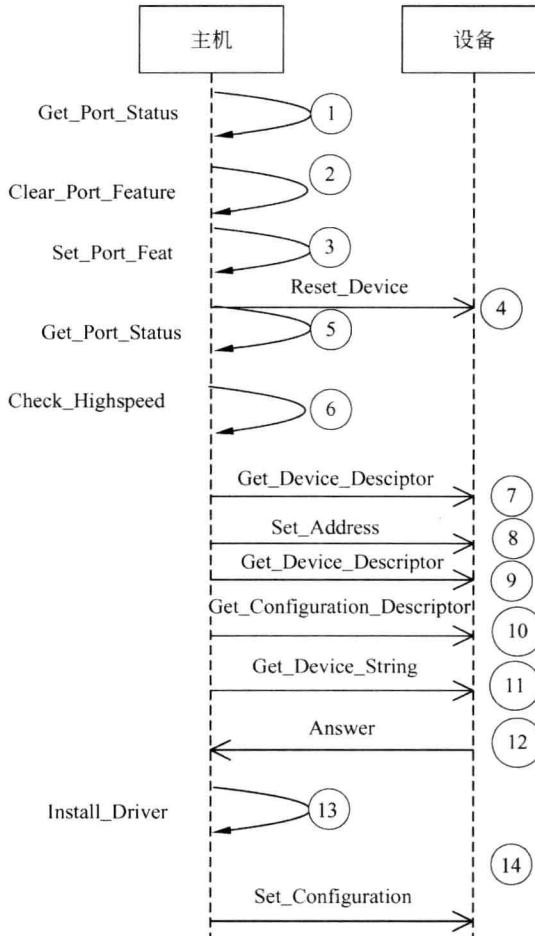


图 8.5 USB 枚举过程序列图

```
static int get_port_status(struct usb_device *hdev, int port1, struct
usb_port_status *data)
{
    int i, status = -ETIMEDOUT;
    for (i = 0; i < USB_STS_RETRIES && status == -ETIMEDOUT; i++) {
        /*从设备 hdev 的端点 0 请求 USB 设备状态，将读取的设备数据保存在 data*/
        status = usb_control_msg(hdev, usb_rcvctrlpipe(hdev, 0),
            USB_REQ_GET_STATUS, USB_DIR_IN | USB_RT_PORT, 0, port1,
            data, sizeof(*data), USB_STS_TIMEOUT);
    }
    return status;
}
```

(2) **Clear\_Port\_Feature:** 用于清除 STATUS\_CHANGE 寄存器中的标志。

```
static int clear_port_feature(struct usb_device *hdev, int port1, int
feature)
{
    /*清除设备 hdev 端口信息*/
    return usb_control_msg(hdev, usb_sndctrlpipe(hdev, 0),
        USB_REQ_CLEAR_FEATURE, USB_RT_PORT, feature, port1,
        NULL, 0, 1000);
}
```

(3) Set\_Port\_Feature: 当主机知道有新的设备时, 主机向集线器发送 Set\_Port\_Feature 请求, 请求集线器重新设置端口。集线器使得设备的 USB 数据线处于重启 (RESET) 状态至少 10ms。

```
static int set_port_feature(struct usb_device *hdev, int port1, int feature)
{
    /*设置设备 hdev 端口信息*/
    return usb_control_msg(hdev, usb_sndctrlpipe(hdev, 0),
                           USB_REQ_SET_FEATURE, USB_RT_PORT, feature, port1,
                           NULL, 0, 1000);
}
```

(4) Reset\_Device: 重启设备。

```
int usb_reset_device(struct usb_device *udev);
```

(5) Get\_Port\_Status: 主机发送一个 Get\_Port\_Status 请求验证设备是否激起重启状态。返回的数据有一位表示设备仍然处于重启状态。集线器释放重启状态后, 设备就进入了默认状态, 即设备已准备好通过 Endpoint 0 的默认流程响应控制传输。即设备目前使用默认地址 0x0 和主机通信。

(6) 集线器检测设备速度: 集线器通过测定哪根信号线 (D+或 D-) 在空闲时有更高的电压来检测设备是低速设备还是全速设备。然后通过 check\_highspeed() 检测是全速设备还是高速设备。

```
static void check_highspeed (struct usb_hub *hub, struct usb_device *udev,
int port1)
{
    struct usb_qualifier_descriptor *qual;
    int status;
    /*给结构体 usb_qualifier_descriptor 分配空间*/
    qual = kmalloc (sizeof *qual, GFP_KERNEL);
    if (qual == NULL)
        return;
    /*从设备 udev 读取设备 USB 描述符, 该描述的类型为 USB_DT_DEVICE_QUALIFIER*/
    status = usb_get_descriptor (udev, USB_DT_DEVICE_QUALIFIER, 0, qual,
sizeof *qual);
    if (status == sizeof *qual) {
        dev_info(&udev->dev, "not running at top speed; "
                 "connect to a high speed hub\n");
        /*hub LEDs are probably harder to miss than syslog*/
        if (hub->has_indicators) {
            hub->indicator[port1-1] = INDICATOR_GREEN_BLINK;
            /*加入工作者线程 hub->leds, 等待时间为 0 后执行*/
            schedule_delayed_work (&hub->leds, 0);
        }
    }
    /*释放所分配结构体 qual 的空间*/
    kfree(qual);
}
```

(7) Get\_Device\_Descriptor: 以取得默认控制管道所支持的最大数据包长度, 并在有限的时间内等待 USB 设备的响应, 该长度包含在设备描述符的 bMaxPacketSize0 字段中, 其地址偏移量为 7, 所以这时主机只需读取该描述符的前 8 个字节。

(8) Set\_Address: PC 主机分配一个设备地址给新接入的 I/O 设备, 所有随后的请求都

将发送到这个新的设备地址。接收新分配的设备地址后，设备进入到已编址状态。

```
static int hub_set_address(struct usb_device *udev, int devnum)
{
    int retval;
    if (devnum <= 1)
        return -EINVAL;
    if (udev->state == USB_STATE_ADDRESS)
        return 0;
    if (udev->state != USB_STATE_DEFAULT)
        return -EINVAL;
    /*向设备 udev 发送请求设置地址*/
    retval = usb_control_msg(udev, usb_sndaddr0pipe(),
        USB_REQ_SET_ADDRESS, 0, devnum, 0,
        NULL, 0, USB_CTRL_SET_TIMEOUT);
    if (retval == 0) {
        /*更新设备的新地址*/
        update_address(udev, devnum);
        /*设置 USB 设备状态为 USB_STATE_ADDRESS*/
        usb_set_device_state(udev, USB_STATE_ADDRESS);
        /*重新初始化设备 udev 的端点 0*/
        usb_ep0_reinit(udev);
    }
    return retval;
}
```

(9) **Get\_Configuration\_Descriptor:** 设备驱动程序开始读取所有关于设备的信息，包括设备的接口和端点。对于一个功能复杂的 I/O 设备，配置可能相当庞大。如果设备有多个配置，驱动程序也要读出所有配置。

(10) **Get\_Device\_String:** 主机发送 **Get\_Device\_String** 命令给设备，获得字符集描述 (unicode)，比如厂商、产品描述、型号等。

(11) 根据 **Device\_Descriptor** 和 **Device\_Configuration** 应答。将设备、配置、端点描述符等信息反馈给主机，方便主机正确加载设备驱动程序。

(12) 安装设备驱动程序：PC 主机需要决定用哪个设备驱动程序去支持新连接的 USB 设备。如果选定的设备驱动程序没有加载到内存中，就要立即加载设备驱动程序到内存中。

(13) **Set\_Configuration:** 现在设备已经被配置好了并且可以运行，此时设备进入已配置状态。

### 8.1.6 USB 请求块 (URB)

USB 请求块 (USB request block, URB) 是 USB 设备驱动中用来描述与 USB 设备通信所用的基本载体和核心数据结构，与网络设备驱动中的 **sk\_buff** 结构体类似，是 USB 主机与设备之间传输数据的封装。下面是对 URB 结构体的定义。

```
struct urb {
    /*私有的：只能由 USB 核心和主机控制器访问的字段*/
    struct kref kref;           //URB 引用计数
    void *hcpriv;              //主机控制器的私有数据
    atomic_t use_count;        //并发传输计数
    atomic_t reject;           //传输将失败
```

```

int unlinked;                                //未连接错误码

/*公共的：可以被驱动使用的字段*/
struct list_head urb_list;                  //当前使用的URB链表头
struct list_head anchor_list;                //the URB may be anchored
struct usb_anchor *anchor;
struct usb_device *dev;                      //关联的USB设备
struct usb_host_endpoint *ep;                //端点指针
unsigned int pipe;                          //管道信息
int status;                                //URB的当前状态
unsigned int transfer_flags;                // (in) URB_SHORT_NOT_OK | ...
void *transfer_buffer;                     //发送数据到设备或从设备接收数据的缓冲区
dma_addr_t transfer_dma;                  //用来以DMA方式向设备传输数据的缓冲区
int transfer_buffer_length;                //transfer_buffer或transfer_dma指向缓冲区的大小
int actual_length;                         //URB结束后，发送或接收数据的实际长度
unsigned char *setup_packet;                //指向控制URB设置数据包的指针
dma_addr_t setup_dma;                      //控制URB设置数据包的DMA缓冲区
int start_frame;                           //等时传输中用于设置或返回初始帧
int number_of_packets;                    //等时传输中包数
int interval;                             //URB被轮询到的时间间隔（对中断和等时urb有效）
int error_count;                          //等时传输错误数量
void *context;                            //completion()函数上下文
usb_complete_t complete;                  //当URB被完全传输或发生错误时，被调用
struct usb_iso_packet_descriptor iso_frame_desc[0];
                                         /*单个URB一次可定义多个等时传输时，描述各个等时传输*/
};

当 transfer_flags 标志中的 URB_NO_TRANSFER_DMA_MAP 被设置时，USB 核心将使用 transfer_dma 指向的缓冲区而不使用 transfer_buffer 指向的缓冲区，这表示即将传输 DMA 缓冲区。
当 transfer_flags 标志中的 URB_NO_SETUP_DMA_MAP 被设置时，如果控制 urb 有 DMA 缓冲区，USB 核心将使用 setup_dma 指向的缓冲区而不使用 setup_packet 指向的缓冲区。
```

URB 是 USB 接口通信的关键数据，下面介绍 URB 处理流程的几个重要函数。

## 1. URB创建函数

创建 URB 的过程主要包括为 URB 分配空间，并初始化该 URB 结构体。

```

struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags)
{
    struct urb *urb;
    /*为URB分配空间*/
    urb = kmalloc(sizeof(struct urb) +
                  iso_packets * sizeof(struct usb_iso_packet_descriptor),
                  mem_flags);
    if (!urb) {
        printk(KERN_ERR "alloc_urb: kmalloc failed\n");
        return NULL;
    }
    /*初始化该URB，后面将会介绍初始化URB的具体实现*/
    usb_init_urb(urb);
    return urb;
}

```

参数 iso\_packets 表示这个 URB 应当包含的等时数据包的个数，若为 0 表示不创建等时数据包。参数 mem\_flags 为分配内存的标志。如果分配成功，该函数返回一个 URB 结构体指针，分配失败则返回 0。在驱动中不要静态创建 URB 结构体，否则可能破坏 USB 核心给 URB 使用的引用计数方法。

## 2. 初始化URB函数

根据 USB 的设备端点类型来分，有 3 种初始化函数。

(1) 对于中断 URB，其初始化函数为：

```
static inline void usb_fill_int_urb(struct urb *urb,
                                    struct usb_device *dev,
                                    unsigned int pipe,
                                    void *transfer_buffer,
                                    int buffer_length,
                                    usb_complete_t complete_fn,
                                    void *context,
                                    int interval)
{
    urb->dev = dev;
    urb->pipe = pipe;
    urb->transfer_buffer = transfer_buffer;
    urb->transfer_buffer_length = buffer_length;
    urb->complete = complete_fn;
    urb->context = context;
    if (dev->speed == USB_SPEED_HIGH)
        urb->interval = 1 << (interval - 1);
    else
        urb->interval = interval;
    urb->start_frame = -1;
}
```

参数 urb 指向要被初始化的 URB 的指针；参数 dev 指向这个 URB 要被发送到的 USB 设备；参数 pipe 是这个 URB 要被发送到的 USB 设备的特定端点；参数 transfer\_buffer 是指向发送数据或接收数据的缓冲区的指针，和 URB 一样，它也不能是静态缓冲区，必须使用 kmalloc() 来分配；参数 buffer\_length 是 transfer\_buffer 指针所指向缓冲区的大小；complete 指针指向当这个 URB 完成时被调用的完成处理函数；参数 context 是完成处理函数的“上下文”；参数 interval 是这个 URB 应当被调度的间隔。

(2) 对于控制 URB，其初始化函数为：

```
static inline void usb_fill_control_urb(struct urb *urb,
                                       struct usb_device *dev,
                                       unsigned int pipe,
                                       unsigned char *setup_packet,
                                       void *transfer_buffer,
                                       int buffer_length,
                                       usb_complete_t complete_fn,
                                       void *context)
{
    urb->dev = dev;
    urb->pipe = pipe;
```

```

    urb->setup_packet = setup_packet;
    urb->transfer_buffer = transfer_buffer;
    urb->transfer_buffer_length = buffer_length;
    urb->complete = complete_fn;
    urb->context = context;
}

```

(3) 对于批量 URB，其初始化函数为：

```

static inline void usb_fill_int_urb(struct urb *urb,
                                    struct usb_device *dev,
                                    unsigned int pipe,
                                    void *transfer_buffer,
                                    int buffer_length,
                                    usb_complete_t complete_fn,
                                    void *context,
                                    int interval)
{
    urb->dev = dev;
    urb->pipe = pipe;
    urb->transfer_buffer = transfer_buffer;
    urb->transfer_buffer_length = buffer_length;
    urb->complete = complete_fn;
    urb->context = context;
    if (dev->speed == USB_SPEED_HIGH)
        urb->interval = 1 << (interval - 1);
    else
        urb->interval = interval;
    urb->start_frame = -1;
}

```

批量 URB 和控制 URB 与中断 URB 的参数基本类似。

### 3. 释放URB

释放由 `usb_alloc_urb()` 分配的 URB 结构体，释放 URB 的函数如下：

```

void usb_free_urb(struct urb *urb)
{
    if (urb)
        kref_put(&urb->kref, urb_destroy);
}

```

### 4. 提交URB

提交 URB 给 USB 核心。在完成分配并设置 URB 后，使用 `usb_submit_urb()` 函数把新的 URB 提交到 USB 核心，提交 URB 函数定义如下：

```
int usb_submit_urb(struct urb *urb, gfp_t mem_flags);
```

参数 URB 指向被提交的 URB 结构体，参数 `mem_flags` 是传递给 USB 核心的内存选项，该参数用于指示 USB 核心如何分配内存缓冲区。如果函数执行成功，URB 的控制权

将被 USB 核心接管，否则函数返回错误。

## 8.2 USB 主机驱动

为了达到 USB 主机功能统一、提高系统的可靠性与可移植性的目的，芯片厂家同时确定 USB 标准和相应的主机规范。现在用得比较广泛的有 3 种：INTEL 推出的用于 USB 2.0 高速设备的 EHCI（Enhanced Host Control Interface，增强主机控制接口）规范和 UHCI（Universal Host Control Interface，通用主机）规范，以及前 Compaq、Microsoft 等推出的可用于全速与低速 USB 系统中的 OHCI（Open Host Control Interface，开放主机控制接口）规范。

### 8.2.1 USB 主机驱动结构和功能

USB 主机控制器有 3 种标准：OHCI、UHCI 和 EHCI。OHCI 对硬件的要求与系统性能、软件复杂的要求相对较低，也能够满足大部分具有 USB 接口嵌入式系统的要求。UHCI 对系统的处理能力与软件的开发要求相对要高（PC 就较多地采用了 UHCI）；EHCI 兼容 OHCI 和 UHCI。在嵌入式的 USB HOST 功能中，采用 OHCI 的规范，后面将主要介绍 OHCI 主机驱动。

Linux 中的 USB 子系统核心模块为 USB Core 模块，它为 USB 驱动（device 和 HC）提供了一个用于访问和控制 USB 硬件的统一接口。应用程序发送的 USB 请求块（urb）经过 USB 设备驱动和 USB Core 后到达 USB 主机控制器（HC），主机控制器解析 URB 后将数据发往指定的 USB 设备，如图 8.6 所示。

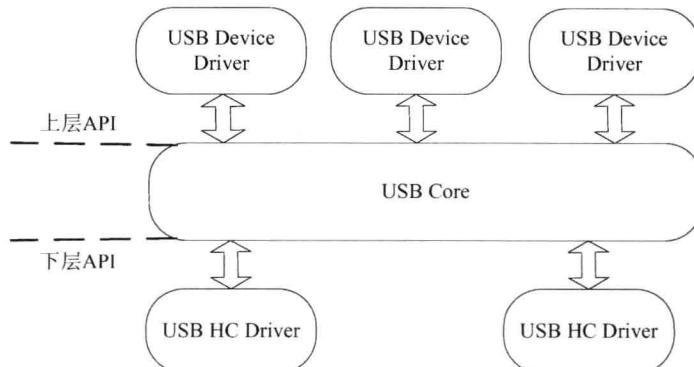


图 8.6 USB 驱动结构

主机控制器驱动程序完成的功能主要包括以下几点：

- 解析和维护 URB，根据不同的端点进行分类缓存 URB。
- 负责不同 USB 传输类型的调度工作。
- 负责 USB 数据的实际传输工作。
- 实现虚拟根 HUB 的功能。

## 8.2.2 主机控制器驱动 (usb\_hcd)

Linux 内核中，USB 主机控制器驱动被定义为 `usb_hcd` 结构体。在 `usb_hcd` 结构体中描述了 USB 主机控制器的硬件信息、状态和操作函数。`usb_hcd` 的定义在内核 `drivers/usb/core/hcd.h` 文件中，其定义代码如下：

```

struct usb_hcd {
    /*控制器的基本信息*/
    struct usb_bus      self;           //hcd is-a bus
    struct kref        kref;            //索引计数器
    const char       *product_desc;    //产品/厂商字符串
    char             irq_descr[24];     //驱动+总线#
    struct timer_list rh_timer;        //根 HUB 轮询时间间隔
    struct urb       *status_urb;       //当前 URB 状态

#ifndef CONFIG_PM
    struct work_struct wakeup_work;   /*针对具备唤醒能力的 USB 设备，USB 设备
也可以远程唤醒的电流信号来请求主机退出中止态或选择中止态。当设备复位时，远程唤醒能力必
须被禁止。*/
#endif

    /*硬件信息和状态*/
    const struct hc_driver *driver;    //控制器驱动使用的回调函数

    /*需要维护的标志*/
    unsigned long      flags;
#define HCD_FLAG_HW_ACCESSIBLE 0x00000001
#define HCD_FLAG_SAW_IRQ    0x00000002

    unsigned          rh_registered:1;   //是否注册根 HUB
    /*The next flag is a stopgap, to be removed when all the HCDs
     * support the new root-hub polling mechanism.*/
    unsigned          uses_new_polling:1;
    unsigned          poll_rh:1;          //是否允许轮询根 HUB 状态
    unsigned          poll_pending:1;    //状态是否改变
    unsigned          wireless:1;        //是否支持无线 USB 主机控制器
    unsigned          authorized_default:1;
    unsigned          has_tt:1;          //Integrated TT in root hub

    int               irq;              //控制器的中断请求号
    void __iomem      *regs;            //控制器使用的内存和 I/O
    u64              rsrc_start;       //控制器使用的内存和 I/O 起始地址
    u64              rsrc_len;          //控制器使用的内存和 I/O 资源长度
    unsigned          power_budget;     //in mA, 0 = 无限制

#define HCD_BUFFER_POOLS 4
    struct dma_pool   *pool [HCD_BUFFER_POOLS];
    int               state;
#define __ACTIVE        0x01
#define __SUSPEND       0x04
#define __TRANSIENT     0x80
#define HC_STATE_HALT   0
#define HC_STATE_RUNNING  (__ACTIVE)
#define HC_STATE QUIESCING  (__SUSPEND|__TRANSIENT|__ACTIVE)
#define HC_STATE_RESUMING  (__SUSPEND|__TRANSIENT)

```

```

#define HC_STATE_SUSPENDED (_SUSPEND)
#define HC_IS_RUNNING(state) ((state) & _ACTIVE)
#define HC_IS_SUSPENDED(state) ((state) & _SUSPEND)
/*主机控制器的私有数据*/
unsigned long hcd_priv[0] __attribute__ ((aligned(sizeof(unsigned
long))));
```

}

usbs\_hcd 结构体中的 hc\_driver 成员也比较重要，该结构包含了具体的用于操作主机控制器的回调函数。该结构体的定义如下：

```

struct hc_driver {
    const char*description;           // "ehci-hcd" 等
    const char*product_desc;          // 产品/厂商字符串
    size_t      hcd_priv_size;        // 私有数据的大小*
    /*中断处理函数*/
    irqreturn_t (*irq) (struct usb_hcd *hcd);
    int flags;
#define HCD_MEMORY 0x0001           // HC 寄存器使用的内存和 I/O
#define HCD_LOCAL_MEM 0x0002         // HC 需要本地内存
#define HCD_USB11 0x0010             // USB 1.1 协议
#define HCD_USB2 0x0020              // USB 2.0 协议

    /*用来初始化 HCD 和 root hub*/
    int (*reset) (struct usb_hcd *hcd);
    int (*start) (struct usb_hcd *hcd);
    /*挂起 HUB 后，进入 D3 etc 前被调用*/
    int (*pci_suspend) (struct usb_hcd *hcd, pm_message_t message);
    /*在进入 D0 (etc) 后，恢复 Hub 前使用*/
    int (*pci_resume) (struct usb_hcd *hcd);
    /*使 HCD 停止写内存和进行 I/O 操作*/
    void(*stop) (struct usb_hcd *hcd);
    /*关闭 HCD*/
    void(*shutdown) (struct usb_hcd *hcd);
    /*返回当前的帧数*/
    int (*get_frame_number) (struct usb_hcd *hcd);
    /*管理 I/O 请求和设备状态*/
    int (*urb_enqueue)(struct usb_hcd *hcd, struct urb *urb, gfp_t
mem_flags);
    int (*urb_dequeue)(struct usb_hcd *hcd, struct urb *urb, int status);
    /*释放端点资源*/
    void (*endpoint_disable)(struct usb_hcd *hcd, struct usb_host_endp-
oint *ep);
    /*支持根 HUB*/
    int (*hub_status_data) (struct usb_hcd *hcd, char *buf);
    int (*hub_control) (struct usb_hcd *hcd, u16 typeReq, u16 wValue, u16
wIndex,
                         char *buf, u16 wLength);
    int (*bus_suspend)(struct usb_hcd *);
    int (*bus_resume)(struct usb_hcd *);
    int (*start_port_reset)(struct usb_hcd *, unsigned port_num);
    /*强制将高速端口转化为全速 companion*/
    void(*relinquish_port)(struct usb_hcd *, int);
    /*是否有端口向 companion 转化*/
    int (*port_handed_over)(struct usb_hcd *, int);
};
```

在Linux内核中，使用函数usb\_create\_hcd()创建HCD：

```
struct usb_hcd *usb_create_hcd (const struct hc_driver *driver, struct device *dev, char *bus_name);
```

在Linux内核中，使用函数usb\_add\_hcd()和usb\_remove\_hcd()增加和移除HCD：

```
int usb_add_hcd(struct usb_hcd *hcd, unsigned int irqnum, unsigned long irqflags);
void usb_remove_hcd(struct usb_hcd *hcd);
```

### 8.2.3 OHCI主机控制器驱动

OHCI HCD 驱动属于主机驱动 HCD 的实例，该结构体中指定了与主机控制器通信的 I/O 内存、主存、主机控制器的队列信息和队列数据管理、驱动状态信息和 ID 等。

```
struct ohci_hcd {
    spinlock_t      lock;
    /*与主机控制器通信的I/O内存(DMA一致) */
    struct ohci_regs __iomem *regs;
    /*与主机控制器通信的主存(DMA一致) */
    struct ohci_hcca   *hcca;
    dma_addr_t       hcca_dma;
    struct ed        *ed_rm_list;           /*指向将被移除的OHCI端点*/
    struct ed        *ed_bulktail;          /*批量队列尾*/
    struct ed        *ed_controltail;      /*控制队列尾*/
    struct ed        *periodic [NUM_INTS];  /*shadow int_table*/
    /*OTG控制器和收发器需要软件交互，其他的外部收发器应该是软件透明的*/
    struct otg_transceiver *transceiver;
    void (*start_hnp)(struct ohci_hcd*ohci);
    /*队列数据的内存管理*/
    struct dma_pool   *td_cache;
    struct dma_pool   *ed_cache;
    struct td        *td_hash [TD_HASH_SIZE];
    struct list_head pending;

    /*驱动状态*/
    int             num_ports;
    int             load [NUM_INTS];
    u32            hc_control;           /*主机控制器控制寄存器的复制*/
    unsigned long   next_statechange;    /*挂起/恢复*/
    u32            fminterval;          /*被保存的寄存器*/
    unsigned        autostop:1;          /*rh auto stopping/stopped*/
    unsigned long   flags;              /*for HC bugs*/

    /*各厂家芯片ID定义*/
#define OHCI_QUIRK_AMD756  0x01          /*erratum #4*/
#define OHCI_QUIRK_SUPERIO 0x02          /*natsemi*/
#define OHCI_QUIRK_INITRESET 0x04         /*SiS, OPTi, ...*/
#define OHCI_QUIRK_BE_DESC  0x08          /*BE descriptors*/
#define OHCI_QUIRK_BE_MMIO  0x10          /*BE registers*/
#define OHCI_QUIRK_ZFMICRO 0x20          /*Compaq ZFMicro chipsets*/
#define OHCI_QUIRK_NECK    0x40          /*lost interrupts*/
```

```

#define OHCI_QUIRK_FRAME_NO      0x80      /*no big endian frame_no shift*/
#define OHCI_QUIRK_HUB_POWER     0x100     /*distrust firmware power/oc setup*/
#define OHCI_QUIRK_AMD_ISO       0x200      /*ISO transfers*/
// there are also chip quirks/bugs in init logic
struct work_struct nec_work;           /*Worker for NEC quirk*/
/*Needed for ZF Micro quirk/
struct timer_list unlink_watchdog;
unsigned          eds_scheduled;
struct ed        *ed_to_check;
unsigned          zf_delay;
#endif DEBUG
struct dentry    *debug_dir;
struct dentry    *debug_async;
struct dentry    *debug_periodic;
struct dentry    *debug_registers;
#endif
};

```

### 8.2.4 S3C24XX OHCI 主机控制器驱动实例

本节将以 S3C24XX OHCI 主机控制器驱动的生命周期过程介绍 OHCI 主机控制器。

(1) S3C24XX OHCI 主机驱动安装系统后，系统自动识别 S3C24XX OHCI 主机驱动。

```

static struct platform_driver ohci_hcd_s3c2410_driver = {
    .probe      = ohci_hcd_s3c2410_drv_probe,
    .remove     = ohci_hcd_s3c2410_drv_remove,
    .shutdown   = usb_hcd_platform_shutdown,
    .driver     = {
        .owner   = THIS_MODULE,
        .name    = "s3c2410-ohci",
    },
};

```

(2) 识别到主机类型为 S3C24XX OHCI 后，自动调用驱动函数 `ohci_hcd_s3c2410_drv_probe()`。在该函数中会调用 `usb_hcd_s3c2410_probe()`。

```

static int ohci_hcd_s3c2410_drv_probe(struct platform_device *pdev)
{
    return usb_hcd_s3c2410_probe(&ohci_s3c2410_hc_driver, pdev);
}

```

(3) 在函数 `usb_hcd_s3c2410_probe()` 中会调用函数 `usb_create_hcd()`，创建主机控制器驱动实例。

```

static int usb_hcd_s3c2410_probe (const struct hc_driver *driver,
                                    struct platform_device *dev)
{
    struct usb_hcd *hcd = NULL;
    int retval;
    /*配置端口 1 电源*/
    s3c2410_usb_set_power(dev->dev.platform_data, 1, 1);
    /*配置端口 2 电源*/
    s3c2410_usb_set_power(dev->dev.platform_data, 2, 1);
    /*创建 USB 主机驱动控制器*/
}

```

```

hcd = usb_create_hcd(driver, &dev->dev, "s3c24xx");
if (hcd == NULL)
    return -ENOMEM;
/*下面为对创建的主机控制器进行初始化*/
hcd->rsrc_start = dev->resource[0].start;
hcd->rsrc_len = dev->resource[0].end - dev->resource[0].start + 1;
if (!request_mem_region(hcd->rsrc_start, hcd->rsrc_len, hcd_name)) {
    dev_err(&dev->dev, "request_mem_region failed\n");
    retval = -EBUSY;
    goto err_put;
}
/*获取时钟信息*/
clk = clk_get(&dev->dev, "usb-host");
if (IS_ERR(clk)) {
    dev_err(&dev->dev, "cannot get usb-host clock\n");
    retval = -ENOENT;
    goto err_mem;
}
usb_clk = clk_get(&dev->dev, "usb-bus-host");
if (IS_ERR(usb_clk)) {
    dev_err(&dev->dev, "cannot get usb-host clock\n");
    retval = -ENOENT;
    goto err_clk;
}
s3c2410_start_hc(dev, hcd);
/*将主机控制器的I/O地址空间映射到内存的虚拟地址空间，便于后面的访问*/
hcd->regs = ioremap(hcd->rsrc_start, hcd->rsrc_len);
if (!hcd->regs) {
    dev_err(&dev->dev, "ioremap failed\n");
    retval = -ENOMEM;
    goto err_ioremap;
}
/*初始化主机控制器*/
ohci_hcd_init(hcd_to_ohci(hcd));
/*USB主机控制器初始化和注册*/
retval = usb_add_hcd(hcd, dev->resource[1].start, IRQF_DISABLED);
return 0;
/*省略了出错处理部分*/
}

```

(4) 函数 `usb_create_hcd()` 创建主机控制器驱动实例。

```

struct usb_hcd *usb_create_hcd (const struct hc_driver *driver, struct
device *dev, const char *bus_name)
{
    struct usb_hcd *hcd;
    /*为主机控制器分配空间*/
    hcd = kzalloc(sizeof(*hcd) + driver->hcd_priv_size, GFP_KERNEL);
    if (!hcd) {
        dev_dbg (dev, "hcd alloc failed\n");
        return NULL;
    }
    /*设置设备 dev 的 driver_data 字段*/
    dev_set_drvdata(dev, hcd);
    /*初始化主机控制器的计数器*/
    kref_init(&hcd->kref);
    /*初始化 usb_bus 结构体*/
    usb_bus_init(&hcd->self);
}

```

```

/*设置结构体 usb_bus 信息*/
hcd->self.controller = dev;
hcd->self.bus_name = bus_name;
hcd->self.uses_dma = (dev->dma_mask != NULL);
/*初始化主机控制器根 HUB 的 polling 时间*/
init_timer(&hcd->rh_timer);
hcd->rh_timer.function = rh_timer_func;
hcd->rh_timer.data = (unsigned long) hcd;
#endif CONFIG_PM
INIT_WORK(&hcd->wakeup_work, hcd_resume_work);
#endif
/*指定驱动，完成了主机控制初始化后，绑定驱动到主机控制器上*/
hcd->driver = driver;
hcd->product_desc = (driver->product_desc) ? driver->product_desc :
    "USB Host Controller";
return hcd;
}

```

(5) 结构体 ohci\_s3c2410\_hc\_driver 定义了 S3C2410 主机控制器驱动。

```

static const struct hc_driver ohci_s3c2410_hc_driver = {
    .description =      hcd_name,
    .product_desc =    "S3C24XX OHCI",
    .hcd_priv_size =   sizeof(struct ohci_hcd),
    /*通用硬件连接*/
    .irq =              ohci_irq,
    .flags =            HCD_USB11 | HCD_MEMORY,
    /*基本生命周期操作*/
    .start =            ohci_s3c2410_start,
    .stop =             ohci_stop,
    .shutdown =         ohci_shutdown,
    /*管理 I/O 请求和相关的资源*/
    .urb_enqueue =      ohci_urb_enqueue,
    .urb_dequeue =      ohci_urb_dequeue,
    .endpoint_disable = ohci_endpoint_disable,
    /*调度支持*/
    .get_frame_number = ohci_get_frame,
    /*根 HUB 支持*/
    .hub_status_data =  ohci_s3c2410_hub_status_data,
    .hub_control =      ohci_s3c2410_hub_control,
#endif CONFIG_PM
    .bus_suspend =       ohci_bus_suspend,
    .bus_resume =        ohci_bus_resume,
#endif
    .start_port_reset = ohci_start_port_reset,
};

```

(6) 当函数 s3c2410\_start\_hc() 被调用时，平台信息被传递。执行 S3C2410 主机驱动的 start 时，会自动执行函数 ohci\_s3c2410\_start()。

```

static int ohci_s3c2410_start (struct usb_hcd *hcd)
{
    struct ohci_hcd *ohci = hcd_to_ohci (hcd);
    int ret;
    if ((ret = ohci_init(ohci)) < 0)           // 初始化 ohci 主机控制器
        return ret;
    if ((ret = ohci_run (ohci)) < 0) {          // 运行 ohci 主机控制器
        err ("can't start %s", hcd->self.bus_name);
    }
}

```

```

        ohci_stop(hcd);
        return ret;
    }
    return 0;
}

```

完成前面 6 步就将 ohci 主机运行起来了。其注销过程与主机驱动注册过程类似。注销时执行驱动移除函数 `ohci_hcd_s3c2410_drv_remove()`, 调用函数 `usb_hcd_s3c2410_remove()`, 在该函数中移除主机控制器实例, 执行 `s3c2410_stop_hc()`, 并释放资源。

```

static void usb_hcd_s3c2410_remove(struct usb_hcd *hcd, struct
platform_device *dev)
{
    usb_remove_hcd(hcd);                                //移除主机控制器
    s3c2410_stop_hc(dev);                             //释放资源
    iounmap(hcd->regs);                            //释放 I/O 映射的内存空间
    release_mem_region(hcd->rsrc_start, hcd->rsrc_len); //释放 HCD 占用的资源
    usb_put_hcd(hcd);                                //注销 HCD 对象
}

```

## 8.3 USB 设备驱动

USB 驱动大致分为：音频设备类、通信设备类、HID（人机接口）设备类、显示设备类、海量设备类、电源设备类、打印设备类和集线器设备类。USB 骨架提供了一个最基础的 USB 驱动程序，本节将通过 USB 骨架来分析 USB 驱动的写法。

### 8.3.1 USB 骨架程序分析

在 Linux kernel 源码目录 `driver/usb/usb-skeleton.c` 中描述 USB 驱动编写的主要框架。下面介绍 USB 驱动主要实现的部分。

#### 1. 设备驱动结构体

结构体 `usb_driver` 定义了驱动的名字和驱动的接口函数，在驱动注册的时候，这些信息被注册到系统中，调用系统的这些函数访问设备时，就会调用对应到驱动的相应的接口函数。

```

static struct usb_driver skel_driver = {
    .name =      "skeleton",           //驱动名字
    .probe =     skel_probe,          //驱动探针函数
    .disconnect = skel_disconnect,   //驱动断开函数
    .suspend =   skel_suspend,       //驱动挂起函数
    .resume =    skel_resume,        //驱动恢复
    .pre_reset =  skel_pre_reset,
    .post_reset = skel_post_reset,
    .id_table =  skel_table,         //驱动支持的产品 ID 和厂家 ID
    .supports_autosuspend = 1,
};

```

## 2. 文件操作结构体与设备初始化

在结构体 skel\_ops 中定义了 usb-skel 设备的各种操作函数。当在 usb-skel 设备上发生相应操作时，USB 文件系统会调用对应的函数进行处理。

```
static const struct file_operations skel_fops = {
    .owner = THIS_MODULE,
    .read = skel_read,           //读操作
    .write = skel_write,         //写操作
    .open = skel_open,           //打开操作
    .release = skel_release,     //释放操作
    .flush = skel_flush,         //清除操作
};
```

从 skel\_driver 结构可以知道 usb-skel 设备的初始化函数是 skel\_probe() 函数。设备初始化过程主要包括探测设备类型、分配 USB 设备使用的 URB 资源、注册 USB 设备操作函数等。skel\_class 结构体记录了 usb-skel 设备信息。

```
static struct usb_class_driver skel_class = {
    .name = "skel%d",
    .fops = &skel_fops,
    .minor_base = USB_SKEL_MINOR_BASE,
};
```

name 变量中采用%d 通配符表示十进制整数。当一个新的 usb-skel 类型的设备被接入 USB 总线后，会按照子设备编号自动为该设备设置设备名称。fops 是设备的文件操作结构体变量。

## 3. USB骨架程序模块的注册和注销

当使用 insmod skeleton.ko 加载模块时，函数 usb\_skel\_init() 被调用，在该函数中调用函数 usb\_register() 注册设备驱动。

```
static int __init usb_skel_init(void)
{
    int result;
    /*注册 usb 驱动*/
    result = usb_register(&skel_driver);
    if (result)
        err("usb_register failed. Error number %d", result);
    return result;
}
```

当使用 rmmod skeleton 卸载模块时，usb\_skel\_exit() 函数被调用，在该函数中调用 usb\_deregister() 函数注销模块驱动。

```
static void __exit usb_skel_exit(void)
{
    /*注销 USB 驱动*/
    usb_deregister(&skel_driver);
}
```

当使用 rmmod skeleton 卸载模块时，usb\_skel\_exit() 函数被调用，调用 usb\_deregister() 函数注销模块驱动。

#### 4. USB骨架驱动所支持的设备

在数组 skel\_table [] 中指定驱动所支持设备的 VENDOR\_ID 和 PRODUCT\_ID，一些类似设备的驱动工作可以通过在该数组后添加该设备的 NEW\_VENDOR\_ID 和 NEW\_PRODUCT\_ID 方式实现。添加方式为：

```
/*定义驱动程序所支持的厂家 ID 和产品 ID*/
#define USB_SKEL_VENDOR_ID 0xffff0
#define USB_SKEL_PRODUCT_ID 0xffff0
/*表中包含驱动所支持的所有厂家 ID 和产品 ID*/
static struct usb_device_id skel_table [] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
    { USB_DEVICE(NEW_VENDOR_ID, NEW_PRODUCT_ID) },
                                //此处添加支持的新设备的 ID 号
    {}                      /*Terminating entry*/
};
```

#### 5. USB骨架驱动探测函数

当一个设备被安装或者有设备插入后，USB 核心认为该驱动程序应该进行处理时，探测函数被调用，探测函数检查传递给它的设备信息，确定驱动程序是否支持该设备。

```
static int skel_probe(struct usb_interface *interface, const struct
usb_device_id *id)
{
    struct usb_skel *dev;
    struct usb_host_interface *iface_desc;
    struct usb_endpoint_descriptor *endpoint;
    size_t buffer_size;
    int i;
    int retval = -ENOMEM;
    /*为设备分配空间并初始化*/
    dev = kzalloc(sizeof(*dev), GFP_KERNEL);
    if (!dev) {
        err("Out of memory");
        goto error;
    }
    kref_init(&dev->kref);
    sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
    mutex_init(&dev->io_mutex);
    spin_lock_init(&dev->err_lock);
    init_usb_anchor(&dev->submitted);
    dev->udev = usb_get_dev(interface_to_usbdev(interface));
    dev->interface = interface;
    /*设置端点信息*/
    /*仅用于第一个块输入和输出端点*/
    iface_desc = interface->cur_altsetting;
    for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
        endpoint = &iface_desc->endpoint[i].desc;

        if (!dev->bulk_in_endpointAddr &&
            usb_endpoint_is_bulk_in(endpoint)) {
            /*we found a bulk in endpoint*/
            buffer_size = le16_to_cpu(endpoint->wMaxPacketSize);
            dev->bulk_in_size = buffer_size;
```

```

dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
if (!dev->bulk_in_buffer) {
    err("Could not allocate bulk_in_buffer");
    goto error;
}
}
if (!dev->bulk_out_endpointAddr &&
    usb_endpoint_is_bulk_out(endpoint)) {
    /*we found a bulk out endpoint*/
    dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
}
if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
    err("Could not find both bulk-in and bulk-out endpoints");
    goto error;
}
/*在接口设备中保存数据指针*/
usb_set_intfdata(interface, dev);
/*注册USB设备*/
retval = usb_register_dev(interface, &skel_class);
if (retval) {
    /*something prevented us from registering this driver*/
    err("Not able to get a minor for this device.");
    usb_set_intfdata(interface, NULL);
    goto error;
}
/*通知用户设备所依附的结点*/
info("USB Skeleton device now attached to USBSkel-%d", interface->
minor);
return 0;
error:
if (dev)
    /*this frees allocated memory*/
    kref_put(&dev->kref, skel_delete);
return retval;
}

```

探针函数在设备插入后被调用，如果没有正确注册设备，则提示设备不是活动的设备；如果正确安装驱动，则告知用户设备所依附的结点。

## 6. USB骨架驱动断开函数

当驱动程序因为某种原因(比如被拔出)不应该控制设备时，断开函数 `skel_disconnect()` 被调用，该函数做一些清理工作。

```

static void skel_disconnect(struct usb_interface *interface)
{
    struct usb_skel *dev;
    int minor = interface->minor;

    dev = usb_get_intfdata(interface);           //获得USB设备接口描述
    usb_set_intfdata(interface, NULL);           //设置USB设备接口描述无效

    /*注销USB设备，释放设备号*/
    usb_deregister_dev(interface, &skel_class);

    /*防止其他I/O访问该设备*/

```

```

mutex_lock(&dev->io_mutex);
dev->interface = NULL;
mutex_unlock(&dev->io_mutex);
usb_kill_anchored_urbs(&dev->submitted);

/*减少引用计数*/
kref_put(&dev->kref, skel_delete);
info("USB Skeleton #%d now disconnected", minor);
}

```

当USB设备与主机断开时,调用驱动程序的断开skel\_disconnect()函数,并释放usb-skel设备用到的资源。函数skel\_disconnect()首先获取USB设备接口描述,然后再将其设置为无效。接着调用usb\_deregister\_dev()函数注销USB设备的操作描述符,注销操作本身需要加锁。注销设备描述符后,更新内核对usb-skel设备的引用计数。

### 8.3.2 USB驱动移植的时钟设置

在移植USB驱动的过程中,USB时钟设置如果不正确,则会发生超时错误。通过查看S3C2440A芯片资料的USB时钟控制,在S3C2440A内需要PLL(upll)产生48MHz的时钟给USB,UCLK直到PLL(UPLL)被配置才可以提供。

输出时钟频率UPLL相对于参考输入时钟频率Fin的计算公式如下:

$$Upll = (m \times Fin) / (p \times 2^s)$$

$$M = MDIV + 8$$

$$p = PDIV + 2$$

$$s = SDIV$$

修改mach-mini2440.c文件,在该文件中添加如下代码:

```

#include <mach/regs-clock.h>
#include <mach/usb-control.h>
#include <linux/device.h>
#include <linux/delay.h>
static struct s3c2410_hcd_info usb_mini2440_info = {
    .port[0]      = {
        .flags  = S3C_HCDFLG_USED
    }
};

/*设置USB时钟直到设置的频率稳定*/
int usb_mini2440_init(void)
{
    unsigned long upllvalue = (0x78<<12) | (0x02<<4) | (0x03);           //设置USB时钟频率为48MHz
    printk("USB Control, (c) 2009 mini2440\n");
    s3c_device_usb.dev.platform_data = &usb_mini2440_info;
    while(upllvalue!=__raw_readl(S3C2410_UPLLCON))
    {
        __raw_writel(upllvalue,S3C2410_UPLLCON);
        mdelay(1);
    }
    return 0;
}

```

USB 时钟的计算过程为：

$$m = 0x78 \text{ ( } 120 \text{ ) } + 8 = 128$$

$$p = 0x02 \text{ ( } 2 \text{ ) } + 2 = 4$$

$$s = 0x03 \text{ ( } 3 \text{ ) } = 3$$

$$UPLL = ( 128 \times 12MHz ) / ( 4 \times 2^3 ) = 48MHz$$

```
static void __init mini2440_map_io(void)
{
    s3c24xx_init_io(mini2440_iodesc, ARRAY_SIZE(mini2440_iodesc));
    s3c24xx_init_clocks(12000000);
    s3c24xx_init_uarts(mini2440_uartcfgs,
    ARRAY_SIZE(mini2440_uartcfgs));
    usb_mini2440_init();           //添加 USB 时钟设置函数
}
```

通过对 USB 时钟进行修改后，将内核移植到 ARM 板上后不会出现超时错误。在后面的 USB 驱动实例中将详细介绍 USB 驱动移植的步骤。

下面将通过具体驱动实例的代码分析、内核配置和移植过程，来深化了解 USB 驱动理论。8.4 节将在 ARM 平台上移植简单的 USB 鼠标驱动和 U 盘驱动。

## 8.4 USB 鼠标键盘驱动

USB 鼠标和键盘驱动程序在 Linux 源代码中已经存在，只需在编译内核时选上即可。USB 鼠标的源代码文件为 `usbmouse.c`，USB 键盘的源代码文件为 `usbkbd.c`。

### 8.4.1 USB 鼠标驱动代码分析

鼠标输入 HID 类型，其数据传输采用中断 URB，鼠标端点类型为 in。USB 鼠标驱动的主要部分为其探针函数 `usb_mouse_probe()` 和中断函数 `usb_mouse_irq()`。

```
static int usb_mouse_probe(struct usb_interface *intf, const struct
usb_device_id *id)
{
    /*接口结构体包含于设备结构体中，interface_to_usbdev 是通过接口结构体获得它的设备结构体.*/
    struct usb_device *dev = interface_to_usbdev(intf);
    /*usb_host_interface 是用于描述接口设置的结构体，内嵌在接口结构体
    usb_interface 中.*/
    struct usb_host_interface *interface;
    /*usb_endpoint_descriptor 是端点描述符结构体，内嵌在端点结构体
    usb_host_endpoint 中，而端点结构体内嵌在接口设置结构体中.*/
    struct usb_endpoint_descriptor *endpoint;
    struct usb_mouse *mouse;
    struct input_dev *input_dev;
    int pipe, maxp;
    int error = -ENOMEM;

    interface = intf->cur_altsetting;
```

```

/*鼠标端点数只能为 1*/
if (interface->desc.bNumEndpoints != 1)
    return -ENODEV;

endpoint = &interface->endpoint[0].desc;
/*鼠标端点类型只能为 in*/
if (!usb_endpoint_is_int_in(endpoint))
    return -ENODEV;
/*返回对应端点能够传输的最大的数据包，鼠标的返回的最大数据包为 4 个字节.*/
pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
/*为 mouse 设备结构体分配内存*/
mouse = kzalloc(sizeof(struct usb_mouse), GFP_KERNEL);
input_dev = input_allocate_device();
if (!mouse || !input_dev)
    goto fail1;
/*usb_buffer_alloc 申请内存空间用于数据传输，data 为指向该空间的地址，data_dma 则是这块内存空间的 dma 映射，即这块内存空间对应的 dma 地址。在使用 dma 传输的情况下，则使用 data_dma 指向的 dma 区域，否则使用 data 指向的普通内存区域进行传输。GFP_ATOMIC 表示不等待，GFP_KERNEL 是普通的优先级，可以睡眠等待，由于鼠标使用中断传输方式，不允许睡眠状态，data 又是周期性获取鼠标事件的存储区，因此使用 GFP_ATOMIC 优先级，如果不能分配到内存则立即返回 0.*/
mouse->data = usb_buffer_alloc(dev, 8, GFP_ATOMIC, &mouse->data_dma);
if (!mouse->data)
    goto fail1;
/*usb_alloc_urb 为 urb 结构体申请内存空间，第一个参数表示等时传输时需要传送包的数量，其他传输方式则为 0。申请的内存将通过 usb_fill_int_urb() 函数进行填充.*/
mouse->irq = usb_alloc_urb(0, GFP_KERNEL);
if (!mouse->irq)
    goto fail2;
/*填充 usb 设备结构体和输入设备结构体*/
mouse->usbdev = dev;
mouse->dev = input_dev;

if (dev->manufacturer)
    strlcpy(mouse->name, dev->manufacturer, sizeof(mouse->name));

if (dev->product) {
    if (dev->manufacturer)
        strlcat(mouse->name, " ", sizeof(mouse->name));
    strlcat(mouse->name, dev->product, sizeof(mouse->name));
}
/*输出鼠标的名称、厂商 ID 和产品 ID*/
if (!strlen(mouse->name))
    sprintf(mouse->name, sizeof(mouse->name),
            "USB HIDBP Mouse %04x:%04x",
            le16_to_cpu(dev->descriptor.idVendor),
            le16_to_cpu(dev->descriptor.idProduct));
/*usb_make_path 用来获取 USB 设备在 Sysfs 中的路径，格式为 usb-usb 总线号-路径名.*/
usb_make_path(dev, mouse->phys, sizeof(mouse->phys));
strlcat(mouse->phys, "/input0", sizeof(mouse->phys));
/*将鼠标设备的名称赋给鼠标设备内嵌的输入子系统结构体*/
input_dev->name = mouse->name;
/*将鼠标设备的设备结点名赋给鼠标设备内嵌的输入子系统结构体*/
input_dev->phys = mouse->phys;

```

```

usb_to_input_id(dev, &input_dev->id);
input_dev->dev.parent = &intf->dev;
/*evbit 用来描述事件，EV_KEY 是按键事件，EV_REL 是相对坐标事件*/
input_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_REL);
/*keybit 表示键值，包括左键、右键和中键*/
input_dev->keybit[BIT_WORD(BTN_MOUSE)] = BIT_MASK(BTN_LEFT) |
    BIT_MASK(BTN_RIGHT) | BIT_MASK(BTN_MIDDLE);
/*relbit 表示相对坐标值*/
input_dev->relbit[0] = BIT_MASK(REL_X) | BIT_MASK(REL_Y);
/*除了左键、右键和中键之外的其他按键*/
input_dev->keybit[BIT_WORD(BTN_MOUSE)] |= BIT_MASK(BTN_SIDE) |
    BIT_MASK(BTN_EXTRA);
/*中键滚轮的滚动值*/
input_dev->relbit[0] |= BIT_MASK(REL_WHEEL);
/*将鼠标结构体对象赋给 input_dev*/
input_set_drvdata(input_dev, mouse);
/*填充输入设备 open() 函数指针和 close() 函数指针*/
input_dev->open = usb_mouse_open;
input_dev->close = usb_mouse_close;
/*usb_fill_int_urb 用于填充 URB，将上面填充好的 mouse 结构体的数据填充进 URB
结构体中，在 open 中递交 URB。当 URB 包含一个即将传输的 DMA 缓冲区时应该设置
URB_NO_TRANSFER_DMA_MAP。USB 核心使用 transfer_dma 变量所指向的缓冲区，而不是
transfer_buffer 变量所指向的缓冲区。URB_NO_SETUP_DMA_MAP 用于 Setup 包，URB_NO_TRANSFER_DMA_MAP 用于所有 Data 包。*/
usb_fill_int_urb(mouse->irq, dev, pipe, mouse->data,
    (maxp > 8 ? 8 : maxp),
    usb_mouse_irq, mouse, endpoint->bInterval);
mouse->irq->transfer_dma = mouse->data_dma;
mouse->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
/*向系统注册输入设备*/
error = input_register_device(mouse->dev);
if (error)
    goto fail3;
/*一般在 probe 函数中，都需要将设备相关信息保存在一个 usb_interface 结构体中，以便以后通过 usb_get_intfdata 获取使用。这里鼠标设备结构体信息将保存在 intf 接口结构体内嵌设备结构体中的 driver_data 数据成员中，即 intf->dev->dirver_data = mouse.*/
usb_set_intfdata(intf, mouse);
return 0;
/*发生错误时释放资源*/
fail3:
    usb_free_urb(mouse->irq);
fail2:
    usb_buffer_free(dev, 8, mouse->data, mouse->data_dma);
fail1:
    input_free_device(input_dev);
    kfree(mouse);
    return error;
}

```

鼠标中断函数 `usb_mouse_irq()` 代码分析如下：

```

static void usb_mouse_irq(struct urb *urb)
{
    /*urb 中的 context 指针用于为 USB 驱动程序保存数据*/
    struct usb_mouse *mouse = urb->context;
    /*mouse->data 指向的内存区域将保存着鼠标的按键和移动坐标信息*/

```

```

signed char *data = mouse->data;
struct input_dev *dev = mouse->dev;
int status;

switch (urb->status) {
/*status 值为 0 表示 urb 成功返回，直接跳出循环把鼠标事件报告给输入子系统。*/
case 0:           /*success*/
    break;
/*ECONNRESET 为出错信息，表示 urb 被 usb_unlink_urb() 函数 unlink 了，ENOENT
为出错信息，表示 urb 被 usb_kill_urb() 函数销毁了。usb_kill_urb 表示彻底结束 urb
的生命周期，而 usb_unlink_urb 则是停止 urb，这个函数不等待 urb 完全终止就会返回给
回调函数。这在运行中断处理程序时或者等待某自旋锁时非常有用，在这两种情况下是不能睡
眠的，而等待一个 urb 完全停止很可能可能出现睡眠的情况。*/
case -ECONNRESET: /*unlink*/
case -ENOENT:
case -ESHUTDOWN:
    return;
/*-EPIPE: should clear the halt*/
default:           /*error*/
    goto resubmit;
}
/*向输入子系统汇报鼠标事件情况，以便作出反应。
data 数组的第 0 个字节：bit 0、1、2、3、4 分别代表左、右、中、SIDE、EXTRA 键的按
下情况：
data 数组的第 1 个字节：表示鼠标的水平位移；
data 数组的第 2 个字节：表示鼠标的垂直位移；
data 数组的第 3 个字节：REL_WHEEL 位移。*/
input_report_key(dev, BTN_LEFT,   data[0] & 0x01);
input_report_key(dev, BTN_RIGHT,  data[0] & 0x02);
input_report_key(dev, BTN_MIDDLE, data[0] & 0x04);
input_report_key(dev, BTN_SIDE,   data[0] & 0x08);
input_report_key(dev, BTN_EXTRA,  data[0] & 0x10);
input_report_rel(dev, REL_X,     data[1]);
input_report_rel(dev, REL_Y,     data[2]);
input_report_rel(dev, REL_WHEEL, data[3]);
/*输入子系统通过这个同步信号在多个完整事件报告中区分每一次完整事件报告。报告的内
容包括完整的鼠标事件，包括按键信息、绝对坐标信息和滚轮信息，即上面的信息。*/
input_sync(dev);
/*系统周期性地获取鼠标的事件信息，因此在 URB 回调函数的末尾再次提交 URB 请求块，这样又
会调用新的回调函数，周而复始。在回调函数中提交 URB 只能是 GFP_ATOMIC 优先级的，因为 URB
回调函数运行于中断上下文中禁止导致睡眠的行为。而在提交 URB 过程中可能会需要申请内存、保
持信号量，这些操作或许会导致 USB core 睡眠。*/
resubmit:
    status = usb_submit_urb (urb, GFP_ATOMIC);
    if (status)
        err ("can't resubmit intr, %s-%s/input0, status %d",
              mouse->usbdev->bus->bus_name,
              mouse->usbdev->devpath, status);
}

```

## 8.4.2 USB 键盘驱动代码分析

USB 键盘的探针函数中不仅通过 `usb_fill_int_urb()` 调用函数 `usb_kbd_irq()` 来提交键盘按下键的信息，也使用了 `usb_fill_control_urb()` 调用函数 `usb_kbd_led()` 控制键盘 Led 的状态。

下面列出其主要代码。

```

static int usb_kbd_probe(struct usb_interface *iface,
                         const struct usb_device_id *id)
{
    input_dev->event = usb_kbd_event;
    usb_fill_int_urb(kbd->irq, dev, pipe,
                     kbd->new, (maxp > 8 ? 8 : maxp),
                     usb_kbd_irq, kbd, endpoint->bInterval);

    usb_fill_control_urb(kbd->led, dev, usb_sndctrlpipe(dev, 0),
                         (void *) kbd->cr, kbd->leds, 1,
                         usb_kbd_led, kbd);
}

/*函数usb_kbd_irq()提交键盘按键状态信息。*/
static void usb_kbd_irq(struct urb *urb)
{
    struct usb_kbd *kbd = urb->context;
    int i;
    switch (urb->status) {
    case 0:           /*success*/
        break;
    case -ECONNRESET: /*unlink*/
    case -ENOENT:
    case -ESHUTDOWN:
        return;
    /*-EPIPE: should clear the halt*/
    default:          /*error*/
        goto resubmit;
    }
    /*获得键盘扫描码并报告键盘事件*/
    for (i = 0; i < 8; i++)
        input_report_key(kbd->dev, usb_kbd_keycode[i + 224], (kbd->new[0]
        >> i) & 1);

    for (i = 2; i < 8; i++) {
        if (kbd->old[i] > 3 && memscan(kbd->new + 2, kbd->old[i], 6) ==
            kbd->new + 8) {
            if (usb_kbd_keycode[kbd->old[i]])
                input_report_key(kbd->dev, usb_kbd_keycode[kbd->old[i]], 0);
            else
                dev_info(&urb->dev->dev,
                         "Unknown key (scancode %#x) released.\n", kbd->
                         old[i]);
        }

        if (kbd->new[i] > 3 && memscan(kbd->old + 2, kbd->new[i], 6) ==
            kbd->old + 8) {
            if (usb_kbd_keycode[kbd->new[i]])
                input_report_key(kbd->dev, usb_kbd_keycode[kbd->new[i]], 1);
            else
                dev_info(&urb->dev->dev,
                         "Unknown key (scancode %#x) released.\n", kbd->
                         new[i]);
        }
    }
    input_sync(kbd->dev);
    memcpy(kbd->old, kbd->new, 8);
}

```

```

resubmit:
    i = usb_submit_urb (urb, GFP_ATOMIC);
    if (i)
        err_hid ("can't resubmit intr, %s-%s/input0, status %d",
                 kbd->usbdev->bus->bus_name,
                 kbd->usbdev->devpath, i);
}

```

函数 `usb_kbd_led()` 控制键盘 LED 的状态。

```

static void usb_kbd_led(struct urb *urb)
{
    struct usb_kbd *kbd = urb->context;

    if (urb->status)
        dev_warn(&urb->dev->dev, "led urb status %d received\n",
                 urb->status);
/*在 usb_kbd_event 中对将更新的 LED 状态赋值给 kbd->newleds, 比较 kbd->leds 和
kbd->newleds 的值, 当发生变化时用 kbd->newleds 更新 kbd->leds 的值。如果两者相同则
不更新。*/
    if (*(kbd->leds) == kbd->newleds)
        return;
    *(kbd->leds) = kbd->newleds;
    kbd->led->dev = kbd->usbdev;
    if (usb_submit_urb(kbd->led, GFP_ATOMIC))
        err_hid("usb_submit_urb(leds) failed");
}

```

当事件的类型为 LED 事件时, 函数 `usb_kbd_event()` 将当前的 LED 值保存在 `kbd->newleds` 中, 以便在 `usb_kbd_led` 中进行比较和处理。具体代码如下:

```

static int usb_kbd_event(struct input_dev *dev, unsigned int type,
                         unsigned int code, int value)
{
    struct usb_kbd *kbd = input_get_drvdata(dev);

    if (type != EV_LED)
        return -1;
/*有 LED 事件发生时对 LED 重新赋值*/
    kbd->newleds = (!!test_bit(LED_KANA, dev->led) << 3) | (!!test_bit
(LED_COMPOSE, dev->led) << 3) | (!!test_bit(LED_SCROLLL, dev->led) << 2)
| (!!test_bit(LED_CAPSL, dev->led) << 1) | (!!test_bit(LED_NUML,
dev->led));

    if (kbd->led->status == -EINPROGRESS)
        return 0;

    if (*(kbd->leds) == kbd->newleds)
        return 0;

    *(kbd->leds) = kbd->newleds;
    kbd->led->dev = kbd->usbdev;
    if (usb_submit_urb(kbd->led, GFP_ATOMIC))

```

```

    err_hid("usb_submit_urb(leds) failed");

    return 0;
}

```

### 8.4.3 内核中添加 USB 鼠标键盘驱动

在内核配置中主要需要添加对 USB 的支持、对 HID 接口的支持、对 OHCI HCD 驱动的支持。

(1) 内核的配置中包括对 HID 接口的支持，如图 8.7 所示。

(2) 在设备驱动对 USB 支持中选择配置对 OHCI HCD 的支持，如图 8.8 所示。

在将内核移植到开发板时，使用 USB 鼠标时还需要开发板套件带 LED 屏，使用超级终端无法看到效果。

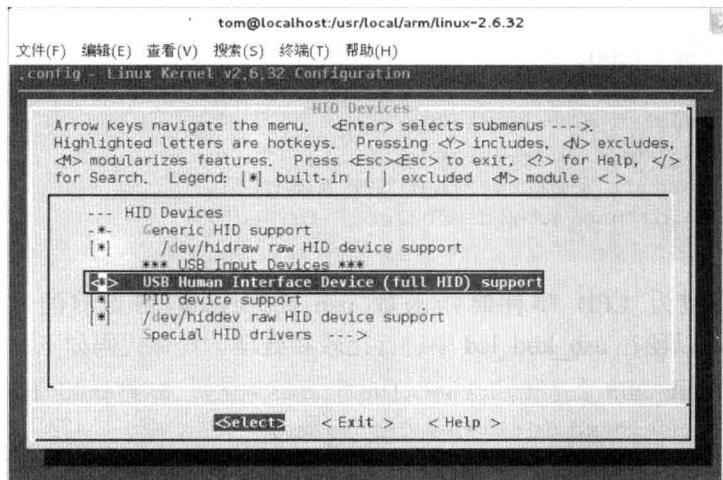


图 8.7 配置对 HID 接口的支持

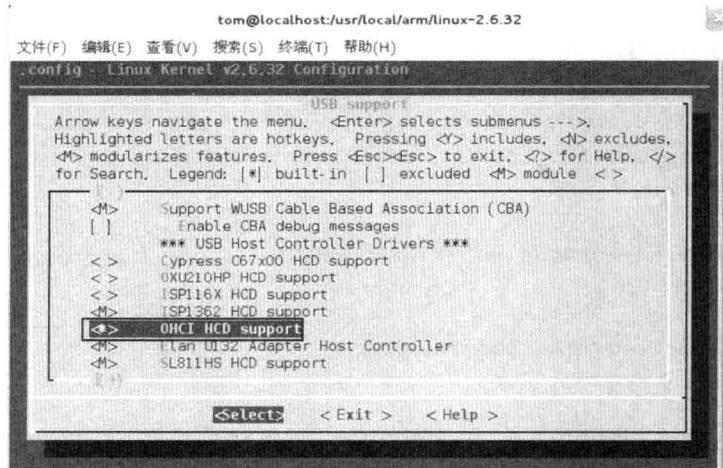


图 8.8 配置对 OHCI HCD 驱动的支持

## 8.5 U 盘驱动

USB 存储设备驱动代码在 drivers/usb/storage 目录中，而对 U 盘的访问只需在内核配置中添加支持 storage 和支持相关的文件系统对 U 盘中内容的识别，添加对字符编码的支持。

### 8.5.1 内核配置

内核中需要添加对 USB 存储设备支持、对 U 盘的存储格式支持、对中文支持的字符编码。

(1) 添加对 USB Mass Storage 驱动的支持，如图 8.9 所示。

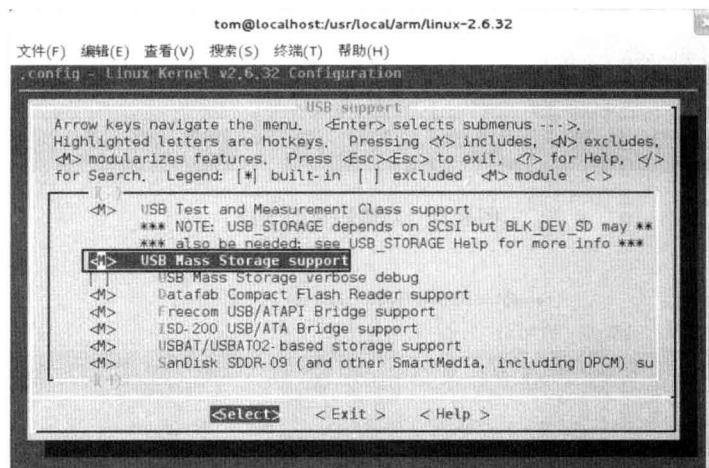


图 8.9 对 USB Mass Storage 驱动的支持

(2) 配置对文件系统的支持，为了能够识别 U 盘中的文件，如图 8.10 所示。

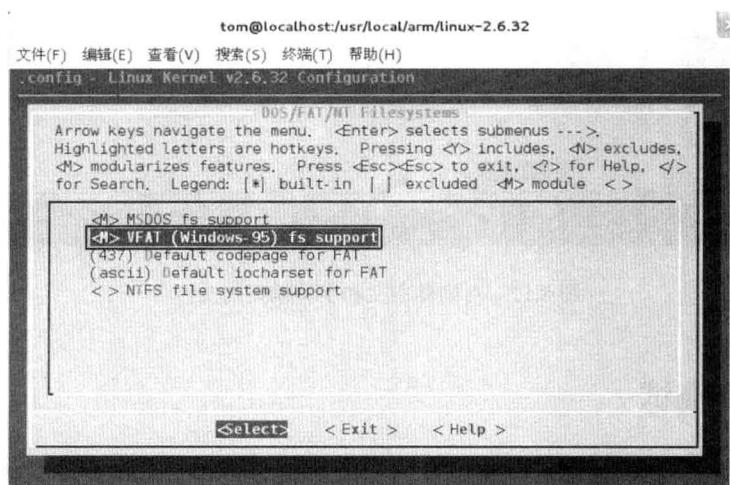


图 8.10 对文件系统的支持

(3) 对字符编码和简体中文的支持, 如图 8.11 和图 8.12 所示。

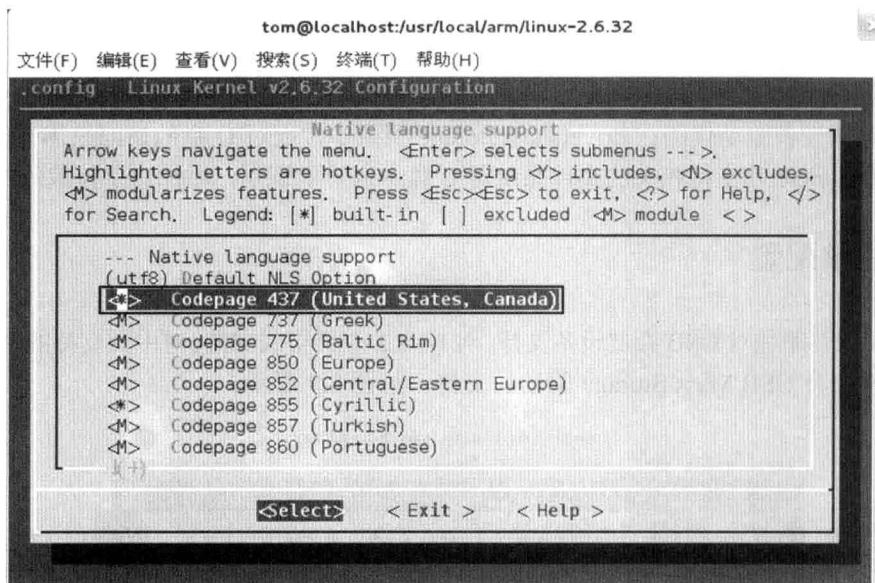


图 8.11 对编码页的支持

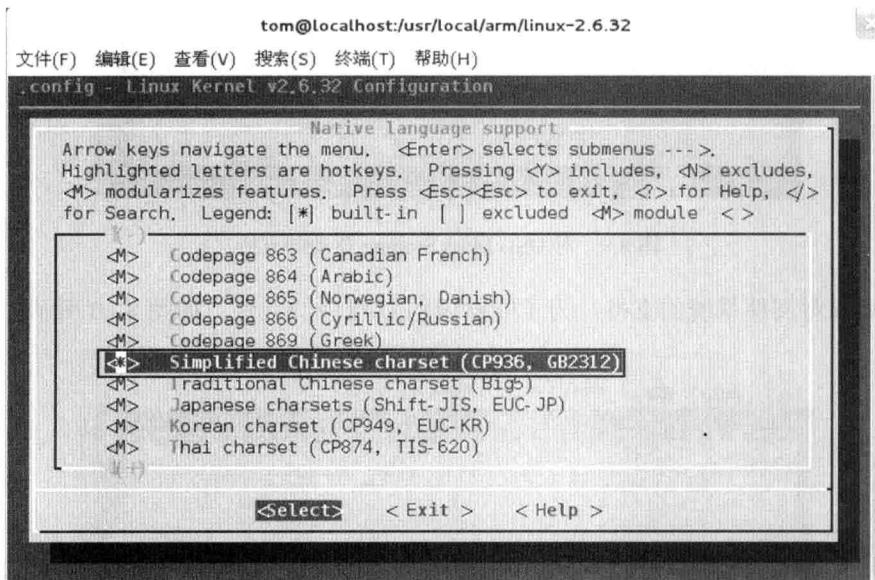


图 8.12 对简体中文和字符编码的支持

## 8.5.2 移植和测试

移植和测试时, 首先需要正确识别设备并找到相应的分区, 然后才能执行挂载操作。

(1) 这里测试时, 采用存储卡大小为 2GB 的 U 盘。当插入到开发板上时打印如下

信息：

```
usb 1-1: new full speed USB device using s3c2410-ohci and address 3
usb 1-1: New USB device found, idVendor=0951, idProduct=1643
usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
usb 1-1: Product: DataTraveler G3
usb 1-1: Manufacturer: Kingston
usb 1-1: SerialNumber: 001CC0EC347CBB50F71C0105
usb 1-1: configuration #1 chosen from 1 choice
```

这些信息表示能够正确识别USB设备，并读取其厂家ID和产品ID，以及存储设备ubal信息。

执行cat /proc/partitions查看磁盘分区信息如下：

```
[root@FriendlyARM/]# cat /proc/partitions
major minor #blocks name

 31      0       256 mtdblock0
 31      1       128 mtdblock1
 31      2      5120 mtdblock2
 31      3     256640 mtdblock3
 31      4     262144 mtdblock4
 8      0    1956864 sda
 8      1    1956864 sda1
```

(2) 挂载U盘。在mnt目录下建立usb目录。挂载前后的信息对比如下所示。

```
[root@FriendlyARM/]# mkdir /mnt/usb
```

挂载前分区信息：

```
[root@FriendlyARM/]# df -h
Filesystem           Size   Used Available Use Mounted on
/dev/root            250.6M  90.4M   160.2M  36% /
tmpfs                29.4M     0    29.4M   0% /dev/shm
```

挂载命令：

```
[root@FriendlyARM/]# mount /dev/ubal /mnt/usb/
```

挂载后的分区信息：

```
[root@FriendlyARM/]# df -h
Filesystem           Size   Used Available Use% Mounted on
/dev/root            61.7M  43.8M   17.9M  71% /
tmpfs                29.9M     0    29.9M   0% /dev/shm
/dev/sda1             1.9G    44K    1.9G   1% /mnt/usb
```

查看U盘信息：

```
[root@FriendlyARM/]# ls /mnt/usb -l
-rwxr-xr-x. 1 tom tom 123 4月 16 2012 mercury_start.bat
-rwxr-xr-x. 1 tom tom 47 4月 16 2012 mercury_stop.bat
-rwxr-xr-x. 1 tom tom 468 4月 16 2012 mysql_start.bat
-rwxr-xr-x. 1 tom tom 205 4月 16 2012 mysql_stop.bat
-rw-r--r--. 1 tom tom 822 9月 29 2012 passwords.txt
-rw-r--r--. 1 tom tom 8241 9月 30 2012 readme_de.txt
-rw-r--r--. 1 tom tom 8096 9月 30 2012 readme_en.txt
```

## 8.6 小 结

本章对 USB 驱动的介绍只起到入门的作用,介绍了简单的 USB 鼠标键盘驱动的移植,在嵌入式平台中访问 U 盘的情况。USB 设备比较复杂,不同类型的 USB 设备包含的接口、端点及 URB 的传输类型都不相同。但不变的是 URB 的生命周期,基本包含创建、初始化、提交、USB core 与 USB 主机传递等过程。URB 的生命周期过程是设计 USB 驱动的关键。

# 第9章 网卡驱动程序移植

网卡是工作在物理层的网络组件，是局域网中连接计算机和传输介质的接口设备。它不仅能实现与局域网传输介质之间的物理连接和电信号匹配，还涉及帧的发送与接收、帧的封装与拆封、介质访问控制、数据的编码与解码及数据缓存的功能等。在嵌入式系统中，网卡是一种常见的外围设备，本章首先讲述以太网的基础知识，之后主要讲解针对 DM9000 网卡的驱动程序移植。

## 9.1 以太网概述

多数人将局域网（Local Area Network，LAN）和以太网（Ethernet）混为一谈，其实这是一种错误的认识。以太网是局域网技术中的一种，它和其他局域网技术比较起来，使用得更普遍、发展得更迅速，以至于人们将“以太网”当做“局域网”的代名词。

以太网（Ethernet）是一种基带局域网规范，它是由 Xerox 公司创建并由 Xerox、Intel 和 DEC 公司联合开发的。它是当今现有局域网采用的最通用的通信协议标准。以太网使用载波监听多路访问及冲突检测技术（CSMA/CD），并以 10M/S 的速率运行在多种类型的电缆上。以太网与 IEEE802.3 系列标准相类似，也是一种技术规范。

### 9.1.1 以太网连接

以太网技术规范中规定了以太网的拓扑结构、传输介质和工作模式，以下分别对其进行描述。

#### 1. 以太网的拓扑结构

以太网拓扑结构有总线型和星型。

- 总线型：总线型网络所采用的传输介质一般也是同轴电缆（包括粗缆和细缆），不过现在也有采用光缆作为总线型传输介质的。早期以太网经常使用总线型的拓扑结构，采用同轴缆作为传输介质，连接比较简单，通常在小规模的网络中不需要用到专用的网络设备。它的特点是所需的电缆较少、价格便宜、管理成本高，不易隔离故障点、采用共享的访问机制，易造成网络拥塞。因为它存在的固有缺陷，已经逐渐被以集线器和交换机为核心的星型网络所代替。
- 星型：网络中的各工作站结点设备通过一个网络集中设备（如集线器或者交换机）连接在一起，各个结点呈星状分布，这便是星型结构。其特点是管理方便、容易扩展、需要专用的网络设备作为网络的核心结点、需要更多的网线、对核心设备

的可靠性要求高。虽然星型网络需要的线缆比总线型多，但布线和连接器比总线型便宜。除此之外，星型拓扑可以通过级联的方式很方便地将网络扩展到很大的规模，因此得到了广泛的应用，被绝大多数的以太网所采用。

## 2. 以太网接口的工作模式

以太网卡可以工作在以下两种模式下：半双工与全双工。

- 半双工：半双工就是指一个时间段内只有一个动作发生。举个例子，一条窄窄的道路上，同一时刻只能有一辆车通过，当同时有两辆车对开，这种情况下就只能一辆车先过，等到这辆车过后另一辆车再开，这个例子很形象地说明了半双工的原理。早期的对讲机及早期集线器等设备都是基于半双工的产品。随着技术的不断进步，半双工会逐渐退出历史的舞台。
- 全双工：全双工（Full Duplex）是指网卡在发送数据的同时也能够接收数据，两者同时进行，这好像我们平时打电话一样，说话的同时也能够听到对方的声音。目前的网卡一般都支持全双工。全双工传输是采用点对点连接的，这种安排没有冲突，这是因为它们使用双绞线中两个独立的线路，这相当于没有安装新的介质就提高了带宽。标准以太网的传输效率可达到 50%~60% 的带宽，双全工在两个方向上都提供了 100% 的效率。

## 3. 传输介质

以太网中采用了多种传输介质，包括同轴缆、双绞线和光纤等。其中双绞线是现在最普通的传输介质，它由两条相互绝缘的铜线组成，典型直径为 1 毫米。两根线绞接在一起是为了防止电磁感应在邻近线对中产生干扰信号，它多用于从主机到集线器或交换机的连接；光纤是软而细的、利用内部全反射原理来传导光束的传输介质，它主要用于交换机间的级联和交换机到路由器间的点到点链路上。同轴缆作为早期的主要连接介质已经逐渐被淘汰了。

### 9.1.2 以太网技术概述

以下简单概述以太网的相关技术标准。

#### 1. 以太网的工作原理

以太网采用带冲突检测的载波帧听多路访问（CSMA/CD）机制。以太网中其他结点都可以看到在网络中发送的所有信息，因此，称以太网是一种广播网络。

以太网的工作过程如下所述。

当以太网中的一台主机要传输数据的时候，它将按如下步骤进行。

- (1) 先检测网络查看是否有其他主机正在传输，即监听信道是否空闲。
- (2) 如果信道忙，则等待，直到信道空闲；如果信道闲，站点就传输数据。

(3) 在发送数据的同时，主机继续监听网络并确信没有其他主机在同时传输数据。因为有可能两个或多个主机都同时检测到网络空闲，然后几乎在同一时刻开始传输数据。如果两个或多个主机同时发送数据，就会产生冲突。

- (4) 当一个传输结点识别出一个冲突，它就发送一个拥塞信号，这个信号使得冲突的

时间足够长，让其他的结点都能发现。

(5) 其他结点收到拥塞信号后，都停止传输，等待一个随机产生的时间间隙（回退时间，Backoff Time）后重发。

## 2. Ethernet地址

以太网中通过给每台主机上的网络适配器（网络接口卡）分配一个唯一的通信地址标识以太网上的每台计算机。这个唯一的通信地址就是人们常说的 Ethernet 地址，通常也称为网卡的物理地址、MAC 地址。

IEEE 给网络适配器制造厂商分配了 Ethernet 地址块，这个叫厂商代号。各厂商又给自己生产的每块网络适配器分配一个唯一的 Ethernet 地址，这个叫设备编号。由于在每块网络适配器出厂时，其 Ethernet 地址都已被烧录到网络适配器中了，有时也将此地址称为烧录地址（Burned-In- Address，BIA）。

Ethernet 地址总长 48 比特，共 6 个字节。其中，前 3 个字节是 IEEE 分配给厂商的厂商代码，后 3 个字节是网络适配器编号，如图 9.1 所示。

## 3. 数据链路层

数据链路层位于 OSI 参考模型中的第二层，介乎于物理层及网络层之间。数据链路层在物理层提供服务的基础上向网络层提供服务，其最基本的服务是将源计算机网络层的数据可靠地传输到相邻结点的目标机网络层。然而在局域网中，多个结点是共享传输介质的，这就必须有某种机制来决定某一个时刻，哪个设备占用传输介质来传送数据。因此，局域网的数据链路层要有介质访问控制的功能。一般数据链路层划分成两个子层，如图 9.2 所示。

- 逻辑链路控制 LLC (Logic Line Control) 子层；
- 介质访问控制 MAC (Media Access Control) 子层。

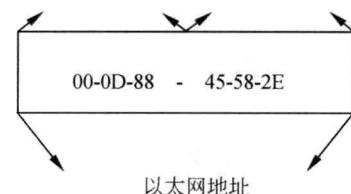


图 9.1 Ethernet 地址格式

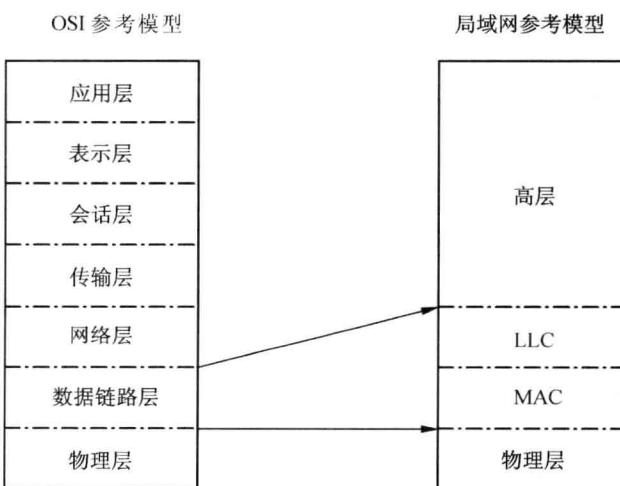


图 9.2 LLC 和 MAC 子层

其中 LLC 子层负责向其上层提供服务；LLC 是在高级数据链路控制(HDLC: High-Level

Data-Link Control) 的基础上发展起来的，并使用了 HDLC 规范子集。LLC 定义了 3 种数据通信操作类型。

- 类型 1：无连接。这种方式对信息的发送通常无法保证接收。
- 类型 2：面向连接。该方式提供了 4 种服务，分别是连接的建立、确认和承认响应、差错恢复（通过请求重发接收到的错误数据实现）及滑动窗口（系数：128）。通过改变滑动窗口可以提高数据传输的速率。
- 类型 3：无连接承认响应服务。

MAC 子层的主要功能包括数据帧的封装或卸装、帧的寻址与识别、帧的接收与发送、链路的管理、帧的差错控制等。MAC 子层屏蔽了不同物理链路种类的差异性；在 MAC 子层诸多功能中，特别重要的一项功能是仲裁介质的使用权，即规定站点何时可以使用共享的通信介质。局域网技术中采用具有冲突检测的载波监听多路访问（Carrier Sense Multiple Access /Collision Detection，CSMA/CD）来控制站点访问共享介质。

### 9.1.3 以太网的帧结构

在 Ethernet 中有几种不同的帧格式，下面就简单介绍一下几种不同的帧格式及它们的差异，先分别列出各种格式的名称。

- Ethernet II 即 DIX 2.0：Xerox 与 DEC、Intel 在 1982 年制定的以太网帧格式标准，是对 Ethernet I 的补充和完善。
- Ethernet 802.3 raw：Novell 在 1983 年公布的专用以太网标准帧格式。
- Ethernet 802.3 SAP：IEEE 在 1985 年公布的 Ethernet 802.3 的 SAP 版本以太网帧格式。
- Ethernet 802.3 SNAP：IEEE 为解决 EthernetII 与 802.3 帧格式的兼容问题推出折衷的 Ethernet SNAP 格式。

在每种格式的以太网帧的开始处都有前导字符，前导字符共 64 比特（8 字节），如图 9.3 所示。这 8 字节的前导符分为两部分：前同步码和帧起始标志符。前 7 个字节为同步码，用 16 进制数 0xAA 填充；最后 1 字节是帧起始标志符，用 0xAB 填充，它标志着以太网帧的开始。前导字符用于使接收结点进行同步并做好接收数据帧的准备。

10101010	10101010	10101010	10101010	10101010	10101010	10101010	10101011
----------	----------	----------	----------	----------	----------	----------	----------

图 9.3 以太网帧前导字符

除前导字符之外，不同格式的以太网帧各字段定义都不相同，彼此并不兼容，下面将分别介绍。

#### 1. Ethernet II 帧格式

Ethernet II 类型以太网帧格式如图 9.4 所示。

Ethernet II 类型以太网帧的前 12 个字节，分别表示发送该数据帧的源机 MAC 地址和接收数据帧的目标机 MAC 地址。在 MAC 地址后面的 2 个字节表示以太网帧所携带的上层数据类型，如十六进制数 0x0800 表示 IP 协议数据，十六进制数 0x809B 表示 AppleTalk

协议数据，十六进制数 0x8138 表示 Novell 类型协议数据等。

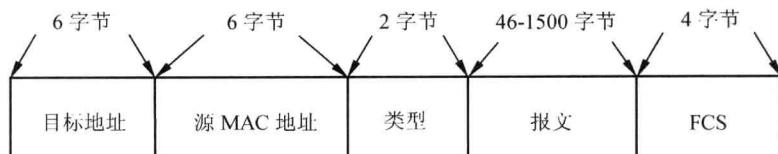


图 9.4 Ethernet II 帧格式

在类型标识后面就是以太帧所携带的真正数据了，它是不确定长度的数据块，最小长度是 46 字节，最大长度是 1500 字节。紧跟其后的是 4 字节的帧校验序列（Frame Check Sequence，FCS），一般采用 32 位 CRC 循环冗余校验，对从“目标 MAC 地址”字段到“数据”字段的数据进行校验。

## 2. Ethernet 802.3 raw 帧格式

Ethernet 802.3 raw 类型以太网帧格式，如图 9.5 所示。

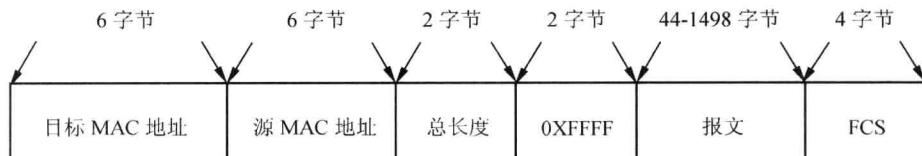


图 9.5 Ethernet 802.3 raw 帧格式

在 Ethernet 802.3 raw 类型以太网帧中，没有专用于表示所携带的上层数据类型的字段。原来 Ethernet II 类型以太网帧中用于表示所携带的上层数据类型的字被“总长度”字段所取代，其取值范围为 46~1500，它指明了其后数据域的长度。

数据字段后面的 2 个字节是固定不变的十六进制数 0xFFFF，它表示该帧为 Novell 以太类型数据帧。

## 3. Ethernet 802.3 SAP 帧格式

Ethernet 802.3 SAP 类型以太网帧格式，如图 9.6 所示。

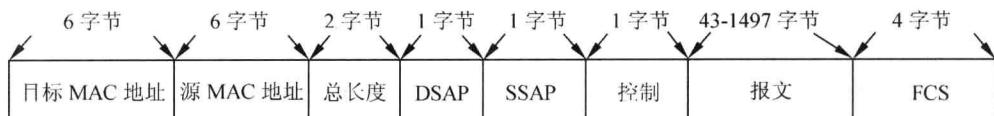


图 9.6 Ethernet 802.3 SAP 帧格式

从图中可以看出，在 Ethernet 802.3 SAP 帧中，在 Ethernet 802.3 raw 类型以太网帧格式的基础上去掉了原来 2 个字节的 0xFFFF 字段，以 1 个字节的 DSAP 和 1 个字节的 SSAP 代之，同时增加了 1 个字节的“控制”字段，这 3 个字段构成了 802.2 逻辑链路控制（LLC）的首部。LLC 提供了无连接（LLC 类型 1）和面向连接（LLC 类型 2）的网络服务。LLC1 应用于以太网环境中，而 LLC2 应用于 IBM SNA 网络环境中。

802.2 LLC 首部包括两个服务访问点：分别是源服务访问点（SSAP）和目标服务访问点（DSAP）。它们用于表示以太网数据帧中所携带的上层数据类型，如 0x06 表示 IP 协议数据，0xE0 表示 Novell 类型协议数据，0xF0 表示 IBM NetBIOS 类型协议数据等。

1 个字节的“控制”字段基本不使用，通常被设置为 0x03，表示采用无连接服务的 802.2 无编号数据格式。

#### 4. Ethernet 802.3 SNAP 帧格式

Ethernet 802.3 SNAP 类型以太网帧格式如图 9.7 所示。

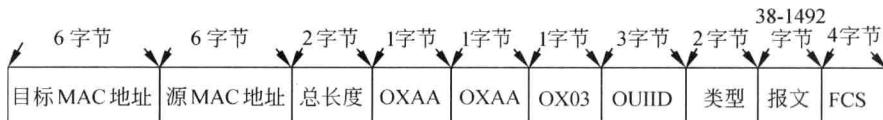


图 9.7 Ethernet 802.3 SNAP 帧格式

Ethernet SNAP 类型数据帧格式与 802.3/802.2 类型数据帧格式的最大区别是增加了一个 5 Bytes 的 SNAP ID，其中前面 3 个字节通常与源 MAC 地址的前 3 个字节相同，为厂商代码，有时也可设为 0。后 2 个字节用来标识以太网帧所携带的上层数据类型。

## 9.2 网络设备驱动程序体系结构

网络设备是 Linux 3 种基本设备之一，它有着其他两种设备不同的特点。网络设备在系统中的作用类似于一个已挂载的块设备。块设备把自己注册到 blk\_dev 数据及其他内核结构中，然后通过自己的 request() 函数在发生请求时传输和接收数据块，同样网络设备也必须在特定的数据结构中注册自己，以便与外界交换数据包时被调用。

网络设备在 Linux 内核里做专门的处理。Linux 的网络系统主要是基于 BSD UNIX 的 Socket 机制。在系统和驱动程序之间定义有专门的数据结构（sk\_buff）进行数据的传递。系统里支持对发送数据和接收数据的缓存，提供了流量控制机制，提供了对多协议的支持。

### 9.2.1 嵌入式 Linux 网络驱动程序介绍

Linux 网络驱动程序是 Linux 网络子系统中的一部分，位于 TCP/IP 网络体系结构的网络接口层，主要实现上层协议栈与网络设备的数据交换。Linux 的网络系统主要是基于 BSD Unix 的套接字（socket）机制，网络设备与字符设备和块设备不同，没有对应的映射到文件系统中的设备结点。

通常，Linux 驱动程序有两种加载方式：一种是静态地编译进内核，内核启动时自动加载；另一种是编写为内核模块，使用 insmod 命令将模块动态加载到正在运行的内核，不需要时可用 rmmod 命令将模块卸载。

Linux 2.6 内核引入了 kbuild 机制，将外部内核模块的编译同内核源码树的编译统一起来，大大简化了特定参数和宏的设置。这样将编写好的驱动模块加入内核源码树，只要修

改相应目录的 Kconfig 文件，把新的驱动加入内核的配置菜单中，然后需要修改相应子目录中与模块编译相关的 Kbuild Makefile，即可使新的驱动在内核源码树中被编译。在嵌入式系统驱动开发时，常常将驱动程序编写为内核模块，方便开发调试。调试完成后，就可以把驱动模块编译进内核，并重新编译出支持特定物理设备的 Linux 内核。

### 9.2.2 Linux 网络设备驱动的体系结构

如图 9.8 所示，Linux 网络驱动程序的体系结构可划分为 4 个层次，即网络协议接口层、网络设备接口层、提供实际功能的设备驱动功能层及设备物理媒介层。

Linux 内核源代码中提供了网络设备接口及以上层的代码。所以移植特定网络硬件驱动程序的主要工作就是编写设备驱动功能层的相应代码，根据底层具体硬件的特性，定义 struct net\_device 这个网络设备接口类型的结构体变量，并实现其中相应操作函数及中断处理程序。

Linux 中所有的网络设备都抽象为一个统一的接口，即网络设备接口，通过 struct net\_device 类型的结构体变量表示网络设备在内核中的运行情况，这里既包括回环（loopback）设备，也包含硬件网络设备接口。内核中是通过以 dev\_base 为头指针的设备链表来管理所有网络设备的。

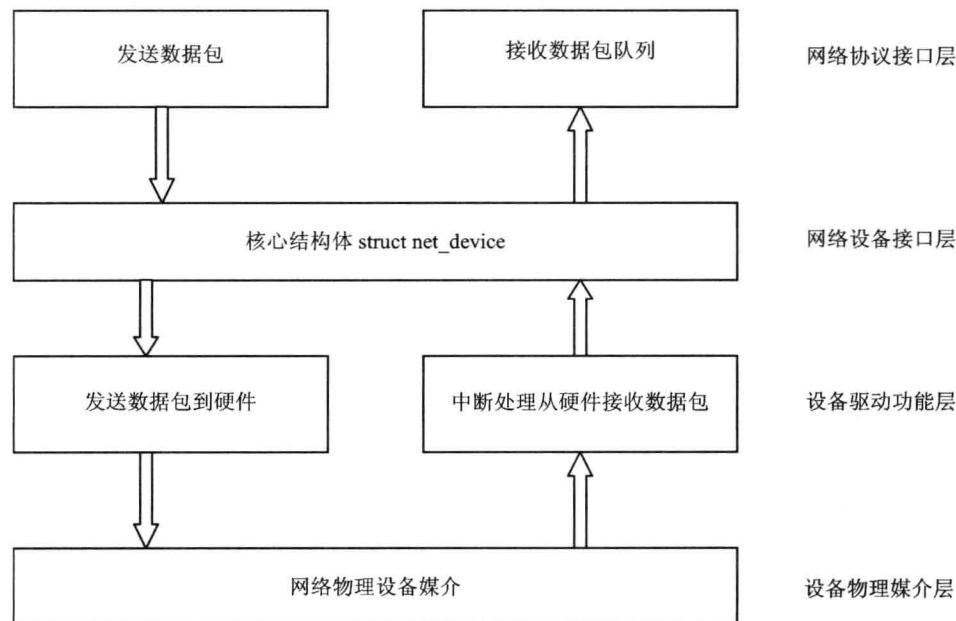


图 9.8 Linux 网络驱动程序体系结构

### 9.2.3 网络设备驱动程序编写方法

网络设备驱动程序编写包括网络设备的初始化，数据包发送和接收函数的编写及其他相关内容，下面将分别讲述。

## 1. 初始化

网络设备的初始化主要是由 device 数据结构中的 init 函数指针所指的初始化函数来完成的，当内核启动或加载网络驱动模块的时候，就会调用初始化过程。在这其中将首先检测网络物理设备是否存在，这是通过检测物理设备的硬件特征来完成的，然后再对设备进行资源配置，这些工作完成之后就要构造设备的 device 数据结构，把检测到的数值对 device 中的变量初始化（这里的设备 device 结构就是前面介绍的 net\_device 结构），这一步非常重要。最后调用 register\_netdevice() 向 Linux 内核中注册该设备并申请内存空间。

## 2. 数据包的发送与接收

数据包的发送和接收是实现 Linux 网络驱动程序中两个最重要的过程，对这两个过程处理的成功与否将直接影响到驱动程序的整体运行质量。首先，在网络设备驱动加载时，通过 device 域中的 init() 函数指针调用网络设备的初始化函数对网络设备进行初始化，如果操作成功了就可以通过 device 域中的 open() 函数指针调用网络设备的打开函数打开设备，再通过 device 域中的建立硬件包头函数指针 hard\_header 建立硬件包头信息。

最后通过协议接口层函数 dev\_queue\_xmit()（详见/linux/net/core/dev.c）调用 device 域中的 hard\_start\_xmit() 函数指针来完成数据包的发送。该函数会把保存在套接字缓冲区中的数据发送到物理设备，该缓冲区是由数据结构 sk\_buff（详见/linux/include/linux/sk\_buff.h）来表示的。

数据包接收是通过系统的中断机制来完成的，当有网络数据到达时，就产生中断信号，网络设备驱动功能程序就调用中断处理程序，即数据包接收函数来处理数据包的接收，然后网络协议接口层调用 netif\_rx() 函数（详见/linux/net/core/dev.c）把接收到的数据包传输到网络协议的上层进行处理。

## 3. 实现模式

实现 Linux 网络设备驱动的功能主要有两种形式：第一种是通过内核进行加载，当内核启动的时候，就开始加载网络设备驱动程序，内核启动完成之后，网络驱动功能也随即实现了。第二种是通过模块加载的形式。比较这两种形式，第二种形式更灵活些，在此重点对模块加载形式进行讨论。

模块设计是 Linux 中特有的技术，它使 Linux 内核功能更容易扩展。采用模块来设计 Linux 网络设备驱动程序会很轻松，并且能够形成固定的模式。任何人只要按照这个模式去设计，都能设计出优良的网络驱动程序。

先简要概述一下基于模块加载的网络驱动程序的设计步骤。首先通过模块加载命令 insmod 把网络设备驱动程序插入到内核中。然后 insmod 将调用 init\_module() 函数首先对网络设备的 init() 函数指针初始化，再通过调用 register\_netdev() 函数在 Linux 系统中注册该网络设备，如果注册成功，再调用 init() 函数指针所指的网络设备初始化函数对设备进行初始化，将设备的 device 数据结构插入到 dev\_base 链表的尾端。最后可以通过执行模块卸载命令 rmmod 调用网络驱动程序中的 cleanup\_module() 函数，对网络驱动程序模块卸载。

通过模块初始化网络接口是在编译内核时做标记，编译为模块，操作系统在启动时并不知道该接口的存在，需要用户在/etc/rc.d/ 目录中定义的初始启动脚本中写入命令或手动

将模块插入内核空间来激活网络接口。这也给我们在何时加载网络设备驱动程序带来了灵活性。

### 9.2.4 网络设备驱动程序应用实例

以 ne2000 兼容网卡为例，来具体介绍基于模块的网络驱动程序的设计过程。可以参考文件 linux/drivers/net/ne.c 和 linux/drivers/net/8390.c。

#### 1. 模块加载和卸载

ne2000 网卡的模块加载功能由 init\_module() 函数完成，具体过程及解释如下：

```
int init_module(void)
{
    int this_dev, found = 0;
    for (this_dev = 0; this_dev < MAX_NE_CARDS; this_dev++)
        //循环检测 ne2000 类型的网络设备接口
    {
        struct net_device *dev = &dev_ne[this_dev];
        //获得网络接口对应的 net-device 结构指针
        dev->irq = irq[this_dev];           //初始化该接口的中断请求号
        dev->mem_end = bad[this_dev];       //初始化接收缓冲区的终点位置
        dev->base_addr = io[this_dev];      //初始化网络接口的 I/O 基地址
        dev->init = ne_probe;
        //初始化 init 为 ne_probe，后面介绍此函数
        /*调用 register_netdevice() 向系统登记网络接口，在这个函数中将分配给网络接口在系统中唯一的名称。并且将该网络接口设备添加到系统管理的链表 dev-base 中进行管理。*/
        if (register_netdev(dev) == 0) {
            found++;
            continue;
        }
        ... //省略
    }
    return 0;
}
```

模块卸载功能由 cleanup\_module() 函数实现，代码如下：

```
void cleanup_module(void)
{
    int this_dev;
    for (this_dev = 0; this_dev < MAX_NE_CARDS; this_dev++) {
        //遍历整个 dev-net 数组
        struct net_device *dev = &dev_ne[this_dev];
        //获得 net-device 结构指针
        if (dev->priv != NULL) {
            void *priv = dev->priv;
            struct pci_dev *idev = (struct pci_dev *)ei_status.priv;
            if (idev) idev->deactivate(idev);
            //调用函数指针 idev->deactivate 将已经激活的网卡关闭使用
            free_irq(dev->irq, dev);
            release_region(dev->base_addr, NE_IO_EXTENT);
            //调用函数 release_region() 释放该网卡占用的 I/O 地址空间
            unregister_netdev(dev);
            //调用 unregister_netdev() 注销这个 net_device() 结构
            kfree(priv);           //释放 priv 空间
        }
}
```

```

    }
}

```

## 2. 网络接口初始化

实现此功能是由 `ne_probe()` 函数完成的，前面已经提到过，在 `init_module()` 函数中用它来初始化 `init()` 函数指针。它主要对网卡进行检测，并且初始化系统中网络设备信息用于后面的网络数据的发送和接收。具体过程及解释如下：

```

int __init ne_probe(struct net_device *dev)
{
    unsigned int base_addr = dev->base_addr;
    /* 初始化 dev-owner 成员，因为使用模块类型驱动，会将 dev-owner 指向对象 modules 结构
     * 指针。*/
    SET_MODULE_OWNER(dev);
    /* 检测 dev->base_addr 是否合法，是则执行 ne-probe1() 函数检测过程，如不是，则需要
     * 自动检测。*/
    if (base_addr > 0x1ff)
        return ne_probe1(dev, base_addr);
    else if (base_addr != 0)
        return -ENXIO;
    /*如果有ISAPnP设备，则调用ne_probe_isapnp()检测这种类型的网卡。*/
    if (isapnp_present() && (ne_probe_isapnp(dev) == 0))
        return 0;
    ...//省略
    return -ENODEV;
}

```

其中函数 `ne_probe_isapnp()` 和 `ne_probe1()` 的区别在于检测中断号上面。PCI 方式只需指定 I/O 基地址就可以自动获得 irq，是由 BIOS 自动分配的。但是 ISA 方式需要获得空闲的中断资源才能分配。

## 9.3 net\_device 数据结构

`struct net_device` 结构体是整个网络驱动结构的核心，在本节中将专门介绍这个结构体。它定义了很多供网络协议接口层调用设备的标准方法，此结构是在 2.6 内核源码树文件中定义的，下面详细列出了其中主要的成员。

### 9.3.1 全局信息

结构体 `net_device` 的第一部分由下面成员组成：

```
char name[IFNAMSIZ]
```

设备名字：设备名字是由驱动设置的，包含一个%d 格式串，`register_netdev` 用一个数字替换它来形成一个唯一的设备名字；分配的编号从 0 开始，如 eth0 后面的 0。

```
unsigned long state
```

设备状态：这个成员包括几个标志。驱动正常情况下不直接操作这些标志；内核提供

了一套实用函数。这些函数在我们进入驱动操作后将进行讨论。

```
struct net_device *next
```

全局列表中指向下一个设备的指针。驱动程序不应该对这个成员进行操作。

```
int (*init)(struct net_device *dev)
```

一个初始化函数。如果设置了这个指针，该函数被 `register_netdev()` 调用完成对 `net_device` 结构的初始化。大部分现代的网络驱动不再使用该函数，初始化在注册接口前进行。

### 9.3.2 硬件信息

下面的成员包含了相对简单的设备低层硬件信息。它们是早期 Linux 网络的延续；大部分现代驱动确实使用它们（可能的例外是 `if_port`）。这里为了完整性，把它们列出。

```
unsigned long rmem_end
unsigned long rmem_start
unsigned long mem_end
unsigned long mem_start
```

设备内存信息。这些成员保存设备使用的共享内存的开始和结束地址。如果设备有不同的接收和发送内存，`mem` 成员由发送内存使用，`rmem` 成员由接收内存使用。`rmem` 成员在驱动之外从不被引用。惯例上，设置 `end` 成员，所以 `end - start` 是可用的板上内存的数量。

```
unsigned long base_addr
```

这个成员表示网络接口的 I/O 基地址，在设备探测时赋值。`ifconfig` 目录可用来显示或修改当前值。`base_addr` 可以在系统启动时，在内核命令行中显式赋值（通过 `netdev=` 参数），或者在模块加载的时候赋值。这个成员内核也不使用它们。

```
unsigned char irq
```

表示中断号。`ifconfig` 可以打印出 `dev->irq` 的值。这个值通常在启动或者加载时设置，并且在后来由 `ifconfig` 打印出来。

```
unsigned char if_port
```

在多端口设备中，这里表示使用的端口。例如这个成员用在同时支持同轴线（`IF_PORT_10BASE2`）和双绞线（`IF_PORT_100BSAET`）以太网连接。完整的已知端口类型设置定义在`<linux/netdevie.h>`中：

```
unsigned char dma
```

为设备分配的 DMA 通道。这个成员只有在一些外设总线时有意义（如 ISA），它不在设备驱动之外使用。

### 9.3.3 接口信息

有关接口的大部分信息是由 `ether_setup` 函数设置的（或者任何其他给定硬件类型适

合的设置函数）。以太网卡可以通过这个通用的函数设置大部分接口信息成员，要指出的是 `flags` 和 `dev_addr` 成员是特定设备的，需在初始化时明确指定。

一些非以太网接口可以使用类似 `ether_setup()` 函数。`deviers/net/net_init.c` 定义了一些类似的函数，包括下列内容。

- ❑ `void ltalk_setup(struct net_device *dev)`: 用于设置一个 LocalTalk 设备的成员。
- ❑ `void fc_setup(struct net_device *dev)`: 用于初始化光通道设备的成员。
- ❑ `void fddi_setup(struct net_device *dev)`: 用于配置一个光纤分布数据接口 (FDDI) 网络的接口。
- ❑ `void hippi_setup(struct net_device *dev)`: 用于预备给一个高性能并行接口 (HIPPI) 的高速互连驱动的成员。
- ❑ `void tr_setup(struct net_device *dev)`: 用于处理令牌环网络接口的设置。

大多数设备会归于这些类别中的某一类。如果你使用的是其他的设备，则需要手动赋值以下的成员。

- ❑ `unsigned short hard_header_len`: 指定硬件头部长度，就是被发送报文前面在 IP 头之前的字节数，或者其他协议信息。对于以太网接口 `hard_header_len` 的值是 14 (`ETH_HLEN`)。
- ❑ `unsigned mtu`: 表示最大传输单元 (MTU)。这个成员在网络层用作驱动报文传输。以太网有一个 1500 字节的 MTU (`ETH_DATA_LEN`)。这个值可用 `ifconfig` 来改变。
- ❑ `unsigned long tx_queue_len`: 在设备发送队列中可以排队的最大帧数。这个值由 `ether_setup` 设置为 1000，但是可以修改它。例如 `plip` 使用 10 来避免浪费系统内存（相比真实以太网接口，`plip` 有一个低些的吞吐量）。
- ❑ `unsigned short type`: 表示接口的硬件类型。这个 `type` 成员由 ARP 用来确定接口支持怎样的硬件地址。对于以太网正确的值是 `ARPHRD_ETHER`，这是由 `ether_setup` 设置的值。可认识的类型定义在 `<linux/if_arp.h>` 中。
- ❑ `unsigned char addr_len`、`unsigned char broadcast[MAX_ADDR_LEN]` 和 `unsigned char dev_addr[MAX_ADDR_LEN]`: 表示硬件 (MAC) 地址长度和设备硬件地址。以太网地址长度是 6 个字节 (我们指的是接口板的硬件 ID)，广播地址由 6 个 `0xff` 字节组成；`ether_setup` 安排成正确的值。设备地址，需要以特定于设备的方式从接口板读出，驱动应当将它复制到 `dev_addr`。硬件地址用来产生正确的以太网头，在报文传递给驱动发送之前。`snull` 设备不使用物理接口，它创造自己的硬件接口。
- ❑ `unsigned short flags` 和 `int features`: 表示接口标志。`flags` 成员是一个位掩码，包括以下的位值：`IFF_` 前缀代表 `interface flags`。有些标志由内核管理，有些由接口在初始化时设置来表明接口的能力和其他特性。有效的标志在 `<linux/if.h>` 中有定义。
- ❑ `IFF_UP`: 对于驱动来说这个标志是只读的。当接口激活并准备传送报文时内核打开它。
- ❑ `IFF_BROADCAST`: 这个标志 (由网络代码维护) 说明接口允许广播。以太网板是这样。
- ❑ `IFF_DEBUG`: 表示调试模式。这个标志用来控制你的 `printk` 调用的复杂性或者用

于其他调试目的。尽管当前没有 in-tree 驱动使用这个标志，它可以通过 ioctl 来设置和重置，驱动中可用它。misc-progs/netifdebug 程序可以用来打开或关闭这个标志。

- ❑ IFF\_LOOPBACK：这个标志只在环回接口中设置。内核检查 IFF\_LOOPBACK，以取代硬连线 lo 名字作为一个特殊接口。
- ❑ IFF\_POINTOPOINT：这个标志说明接口连接到一个点对点链路。它由驱动设置或者由 ifconfig。例如 plip 和 PPP 驱动设置它。
- ❑ IFF\_NOARP：这个表示接口不能进行 ARP。例如点对点接口不需要运行 ARP，它只能增加额外的流量却没有任何有用的信息。
- ❑ IFF\_PROMISC：这个标志（由网络代码）用来激活混杂操作。默认情况下，以太网接口使用硬件过滤器来保证它们只接收广播报文和直接收到接口硬件地址的报文。报文嗅探器（如 tcpdump），在接口上设置混杂模式来存取在接口发送介质上经过的所有报文。
- ❑ IFF\_MULTICAST：驱动设置这个标志来表示接口有组播发送能力。默认的情况下 ether\_setup 设置 IFF\_MULTICAST，如果你的驱动不支持组播，必须在初始化时清除这个标志。
- ❑ IFF\_ALLMULTI：这个标志表示接口接收所有的组播报文。内核在主机进行组播路由时设置它，前提是 IFF\_MULTICAST 置位。IFF\_ALLMULTI 对驱动来说是只读的。
- ❑ IFF\_MASTER 和 IFF\_SLAVE：这些标志给负载均衡代码使用。接口驱动不需要使用它们。
- ❑ IFF\_PORTSEL 和 IFF\_AUTOMEDIA：这些标志表示设备可以在多个介质类型间切换；如无屏蔽双绞线（UTP）和同轴以太网电缆。如果设置了 IFF\_AUTOMEDIA，设备会自动选择正确的介质。
- ❑ IFF\_DYNAMIC：这个标志由驱动设置，表示接口的地址能够变化。目前内核没有使用它。
- ❑ IFF\_RUNNING：这个标志表示接口已启动并在运行。它的存在大部分是因为和 BSD 兼容；内核很少用它。大多数网络驱动不需要关心 IFF\_RUNNING。
- ❑ IFF\_NOTAILERS：在 Linux 中没有使用这个标志，是为了和 BSD 兼容才存在。当一个程序改变 IFF\_UP 时，open() 或者 stop() 设备方法就被调用。进而，当 IFF\_UP 或者其他标志修改了，set\_multicast\_list() 方法被调用。如果驱动需要进行某些动作来响应标志的修改，它必须在 set\_multicast\_list 中采取动作。如当 IFF\_PROMISC 被置位或者复位，set\_multicast\_list 必须通知板上的硬件过滤器。

结构 net\_device 的特性成员由驱动设置来告知，内核关于任何接口拥有的特别硬件能力。完整的集合如下：

- ❑ NETIF\_F\_SG 和 NETIF\_F\_FRAGLIST：这两个标志控制发散/汇聚 I/O 的使用。如果接口可以发送一个报文，其由几个不同的内存段组成，应当设置 NETIF\_F\_SG。

 注意：内核不对设备进行发散/汇聚 I/O 操作，如果它没有同时提供某些校验和形式。理由是如果内核不得不跨过一个分片的报文来计算校验和，它可能也复制数据并同

时接合报文。

- NETIF\_F\_IP\_CSUM、NETIF\_F\_NO\_CSUM 和 NETIF\_F\_HW\_CSUM：这些标志都是告知内核，不需要给一些或所有通过这个接口离开系统的报文进行校验。如果你的接口可以校验 IP 报文，就设置 NETIF\_F\_IP\_CSUM。如果这个接口不曾要求校验和，就设置 NETIF\_F\_NO\_CSUM。环回驱动设置了这个标志，snuff 也设置，因为报文只通过系统内存传送，对它们来说没有机会（1 跳）被破坏，没有必要校验它们。如果你的硬件自己做校验，设置 NETIF\_F\_HW\_CSUM。
- NETIF\_F\_HIGHDMA：如果你的设备能够对高端内存进行 DMA。设置这个标志，没有这个标志，所有提供给你的驱动报文将在低端内存分配。
- NETIF\_F\_HW\_VLAN\_TX、NETIF\_F\_HW\_VLAN\_RX、NETIF\_F\_HW\_VLAN\_FILTER 和 NETIF\_F\_VLAN\_CHALLENGED：这些选项描述你的硬件对 802.1qVLAN 报文的支持。VLAN 支持超出本章的内容。如果 VLAN 报文使你的设备混乱（其实不应该），设置标志 NETIF\_F\_VLAN\_CHALLENGED。
- NETIF\_F\_TSO：如果你的设备能够进行 TCP 分段卸载，设置这个标志。TSO 是一个在这里不涉及的高级特性。

### 9.3.4 设备方法

与字符和块驱动一样，每个网络设备都声明能操作它的函数。本节列出能够对网络接口进行的操作。有些操作可以留做 NULL，其他的通常是不被触动的，因为 ether\_setup 给它们安排了合适的方法。

网络接口的设备方法可分为两组：基本的和可选的。基本的方法包括那些必需的能够使用接口的；可选的方法实现更多高级的不是严格要求的功能。下列是基本方法：

```
int (*open)(struct net_device *dev);
```

打开接口：任何时候 ifconfig 激活它，接口被打开。open()方法应当注册它需要的任何系统资源（I/O 口、IRQ、DMA 等），打开硬件，进行其他设备要求的设置。

```
int (*stop)(struct net_device *dev);
```

停止接口。接口停止当它被关闭。这个函数应当恢复在打开时进行的操作。

```
int (*hard_start_xmit)(struct sk_buff *skb, struct net_device *dev);
```

起始报文的发送方法。完整的报文（协议头和所有）包含在一个 socket 缓存区（sk\_buff）结构。socket 缓存在本章后面介绍。

```
int (*hard_header)(struct sk_buff *skb, struct net_device *dev, unsigned short type, void *daddr, void *saddr, unsigned len);
```

用之前取到的源和目的硬件地址来建立硬件头的函数（在 hard\_start\_xmit 前调用）。它的工作是将作为参数传给它的信息组织成一个合适的特定于设备的硬件头。eth\_header 是以太网类型接口的默认函数：ether\_setup 针对性地对这个成员赋值。

```
int (*rebuild_header)(struct sk_buff *skb);
```

用来在 ARP 解析完成后但是在报文发送前重建硬件头的函数。以太网设备使用默认的函数使用 ARP 支持代码来填充报文缺失的信息。

```
void (*tx_timeout)(struct net_device *dev);
```

一个报文发送没有在一个合理的时间内完成时，由网络代码调用的方法，可能是丢失一个中断或者接口被锁住。它应当处理这个问题并恢复报文发送。

```
struct net_device_stats *(*get_stats)(struct net_device *dev);
```

任何时候当一个应用程序需要获取接口的统计信息时，调用这个方法。当 ifconfig 或者 netstat -i 运行时：

```
int (*set_config)(struct net_device *dev, struct ifmap *map);
```

改变接口配置。这个方法是配置驱动的入口点。设备的 I/O 地址和中断号可以在运行时使用 set\_config 来改变。这种能力可由系统管理员在接口没有探测到时使用。现代硬件正常的驱动一般不需要实现这个方法。

剩下的设备操作是可选的：

```
int weight;
int (*poll)(struct net_device *dev; int *quota);
```

由适应 NAPI 的驱动提供的方法，用来在查询模式下操作的接口。

```
void (*poll_controller)(struct net_device *dev);
```

在中断关闭的情况下，要求驱动检查接口上的事件函数。它用于特殊的内核中的网络任务，例如远程控制台和使用网络的内核调试。

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
```

处理特定于接口的 ioctl 命令。如果接口不需要相应的 net\_device 结构中的成员可留为 NULL，任何特定于接口的命令。

```
void (*set_multicast_list)(struct net_device *dev);
```

当设备的组播列表改变和当标志改变时调用的方法。

```
int (*set_mac_address)(struct net_device *dev, void *addr);
```

如果接口支持改变它的硬件地址的能力，则可以实现这个函数。很多接口根本不支持这个能力。其他的使用默认的 eth\_mac\_addr 实现（在 drivers/net/net\_init.c）。eth\_mac\_addr 只复制新地址到 dev->dev\_addr，只在接口没有运行时做这件事。使用 eth\_mac\_addr 的驱动应当在它们的 open() 方法中从 dev->dev\_addr 里设置硬件 MAC 地址。

```
int (*change_mtu)(struct net_device *dev, int new_mtu);
```

当接口的最大传输单元（MTU）改变时动作的函数。如果用户改变 MTU 时驱动需要做一些特殊的事情，它应当声明它自己的函数，否则，默认的函数会被使用。

```
int (*header_cache)(struct neighbour *neigh, struct hh_cache *hh);
```

header\_cache 被调用来填充 hh\_cache 结构，使用一个 ARP 请求的结果。几乎全部类似

以太网的驱动可以使用默认的 eth\_header\_cache 实现。

```
int (*header_cache_update) (struct hh_cache *hh, struct net_device *dev,
                           unsigned char *haddr);
```

在响应一个变化中，更新 hh\_cache 结构中的目的地址的方法。以太网设备使用 eth\_header\_cache\_update。

```
int (*hard_header_parse) (struct sk_buff *skb, unsigned char *haddr);
```

hard\_header\_parse 方法从包含在 skb 中的报文中抽取源地址，复制到 haddr 的缓存区。函数的返回值是地址的长度。以太网设备通常使用 eth\_header\_parse。

### 9.3.5 公用成员

结构 net\_device 剩下的数据成员由接口使用。有些是 ifconfig 和 netstat 用来提供给用户关于当前配置的信息。因此，接口应该给这些成员赋值：

```
unsigned long trans_start;
unsigned long last_rx;
```

保存一个 jiffy 值的成员。当开始发送和收到一个报文时，驱动负责分别更新这些值。trans\_start 值被网络子系统用来探测发送器加锁。last\_rx 目前没有用到，但是驱动应当尽量维护这个成员以备将来使用。

```
int watchdog_timeo;
```

网络层认为一个传送超时发生前应当过去的最长时间（按 jiffy 计算），调用驱动的 tx\_timeout 函数。

```
void *priv;
```

filp->private\_data 的对等者。在现代的驱动里，这个成员由 alloc\_netdev 设置，不应当直接存取，使用 netdev\_priv 代替。

```
struct dev_mc_list *mc_list;
int mc_count;
```

处理组播发送的成员。mc\_count 是 mc\_list 中的项数目。

```
spinlock_t xmit_lock;
int xmit_lock_owner;
```

xmit\_lock 用来避免对驱动的 hard\_start\_xmit() 函数多个同时调用。xmit\_lock\_owner 已获得 xmit\_lock 的 CPU 号。驱动应当不改变这些成员的值。

结构 net\_device 中有其他的成员，但是网络驱动不需要用到它们。

## 9.4 DM9000 网卡概述

有些嵌入式处理器并没有集成以太网 MAC 层控制器，对于这种处理器，人们选择使

用集成了 MAC 控制器和 PHY 层的以太网芯片，来扩展网络接口。DM9000 就是这样一种常用的自适应以太网芯片。

#### 9.4.1 DM9000 网卡总体介绍

DM9000 是一种快速以太网控制处理器，它合成了 MAC、PHY 和 MMU。该处理器配备了标准 10M/100M 自适应、16K 容量的 FIFO、4 路多功能 GPIO、掉电、全双工工作等功能。支持以太网接口协议。

网络数据有时是以猝发形式收到的，因此，DM9000 还集成了接收缓冲区，以便在接收到数据时能把数据放在这个缓冲区中，然后由数据链路层直接从这个缓冲区里取出数据。链路层通常包括操作系统中的设备驱动程序和计算机中对应的网络接口卡，它们共同处理与电缆的物理接口细节数据，其缓冲区可用来暂时存储要发送或接收的帧。

DM9000 还提供介质无关接口，用来连接所有提供支持介质无关接口功能的家用电话线网络设备或其他收发器。DM9000 支持 8 位、16 位和 32 位接口访问内部存储器，以支持不同的处理器。DM9000 物理协议层接口完全支持使用 10MBps 下 3 类、4 类、5 类非屏蔽双绞线和 100MBps 下 5 类非屏蔽双绞线，这完全符合 IEEE 802.3u 规格。它的自动协调功能可以自动完成配置以最大限度地适合其线路带宽，还支持 IEEE 802.3x 全双工流量控制。这个工作在 DM9000 里面是非常简单的，所以用户可以容易地移植任何系统下的网卡驱动程序。

#### 9.4.2 DM9000 网卡的特点

DM9000 网卡具有如下特点。

- 支持处理器读写内部存储器的数据操作命令以字节/字/双字的长度进行；
- 集成 10/100M 自适应收发器；
- 支持介质无关接口；
- 支持背压模式半双工流量控制模式；
- IEEE 802.3x 流量控制的全双工模式；
- 支持唤醒帧，链路状态改变和远程的唤醒；
- 4K 双字 SRAM；
- 支持自动加载 EEPROM 里面生产商 ID 和产品 ID；
- 支持 4 个通用输入输出口；
- 超低功耗模式；
- 功率降低模式；
- 电源故障模式；
- 可选择 1：1 YL18-2050S, YT37-1107S 或 5：4 变压比例的变压器降低格外功率；
- 兼容 3.3V 和 5.0V 输入输出电压；
- 100 脚 CMOS LQFP 封装工艺。

### 9.4.3 内部寄存器

DM9000 包含一系列可被访问的控制状态寄存器，这些寄存器是字节对齐的，它们在硬件或软件复位时被设置成初始值。以下为 DM9000 的寄存器功能详解。

#### 1. 网络控制寄存器 (NCR)

网络控制寄存器用于对 DM9000 工作状态的控制，可以使 DM9000 复位，功能描述如表 9.1 所述。

表 9.1 网络控制寄存器 (Network Control Register)

功 能	位	描 述
EXT_PHY	[7]	1 选择外部 PHY，0 选择内部 PHY，不受软件复位影响
WAKEEN	[6]	事件唤醒使能，1 使能，0 禁止并清除事件唤醒状态，不受软件复位影响
保留	[5]	
FCOL	[4]	1 强制冲突模式，用于用户测试
FDX	[3]	全双工模式。内部 PHY 模式下只读，外部 PHY 下可读写
LBK	[1-2]	回环模式 (Loopback) 00 通常，01MAC 内部回环，10 内部 PHY 100M 模式数字回环，11 保留
RST	[0]	1 软件复位，10us 后自动清零

#### 2. 网络状态寄存器 (NSR)

网络状态寄存器，通过该寄存器可以获知 DM9000 当前的工作状态，例如是否处于连接状态、发送数据是否完毕、是否处于睡眠状态等，功能描述如表 9.2 所述。

表 9.2 网络状态寄存器 (Network Status Register)

功 能	位	描 述
SPEED	[7]	媒介速度，在内部 PHY 模式下，0 为 100Mbps，1 为 10Mbps。当 LINKST=0 时，此位不用
LINKST	[6]	连接状态，在内部 PHY 模式下，0 为连接失败，1 为已连接
WAKEST	[5]	唤醒事件状态。读取或写 1 将清零该位。不受软件复位影响
保留	[4]	
TX2END	[3]	TX (发送) 数据包 2 完成标志，读取或写 1 将清零该位。数据包指针 2 传输完成
TX2END	[2]	TX (发送) 数据包 1 完成标志，读取或写 1 将清零该位。数据包指针 1 传输完成
RXOV	[1]	RX (接收) FIFO (先进先出缓存) 溢出标志
保留	[0]	

#### 3. 发送控制寄存器 (TCR)

发送控制寄存器，可以控制发送使能，功能描述如表 9.3 所述。

表 9.3 发送控制寄存器 (TX Control Register)

功 能	位	描 述
保留	[7]	
TJDIS	[6]	Jabber 传输使能。1 使能 Jabber 传输定时器（2048 字节），0 禁止
EXCECM	[5]	额外冲突模式控制。0 当额外的冲突计数多于 15 则终止本次数据包，1 始终尝试发送本次数据包
PAD_DIS2	[4]	禁止为数据包指针 2 添加 PAD
CRC_DIS2	[3]	禁止为数据包指针 2 添加 CRC 校验
PAD_DIS2	[2]	禁止为数据包指针 1 添加 PAD
CRC_DIS2	[1]	禁止为数据包指针 1 添加 CRC 校验
TXREQ	[0]	TX（发送）请求。发送完成后自动清零该位

注释：Jabber 是一个有 CRC 错误的长帧（大于 1518byte 而小于 6000byte）或是数据包重组错误。原因：它可能导致网络丢包，多是由于网站有硬件或软件错误。

#### 4. 数据包指针1的发送状态寄存器1 (TSR\_I)

数据包指针 1 的发送状态寄存器 1 功能描述，如表 9.4 所述。

表 9.4 数据包指针 1 的发送状态寄存器 1 (TX Status Register I)

功 能	位	描 述
TJTO	[7]	Jabber 传输超时。该位置位表示由于多于 2048 字节数据被传输而导致数据帧被截掉
LC	[6]	载波信号丢失。该位置位表示在帧传输时发生红载波信号丢失。在内部回环模式下该位无效
NC	[5]	无载波信号。该位置位表示在帧传输时无载波信号。在内部回环模式下该位无效
LC	[4]	冲突延迟。该位置位表示在 64 字节的冲突窗口后又发生冲突
COL	[3]	数据包冲突。该位置位表示传输过程中发生冲突
EC	[2]	额外冲突。该位置位表示由于发生了第 16 次冲突（即额外冲突）后，传送被终止
保留	[1-0]	

#### 5. 数据包指针2的发送状态寄存器2 (TSR\_II)

数据包指针 2 的发送状态寄存器 2 功能描述，如表 9.5 所述。

表 9.5 数据包指针 2 的发送状态寄存器 2 (TX Status Register II)

功 能	位	描 述
TJTO	[7]	Jabber 传输超时。该位置位表示由于多于 2048 字节数据被传输而导致数据帧被截掉
LC	[6]	载波信号丢失。该位置位表示在帧传输时发生红载波信号丢失。在内部回环模式下该位无效

续表

功 能	位	描 述
NC	[5]	无载波信号。该位置位表示在帧传输时无载波信号。在内部回环模式下该位无效
LC	[4]	冲突延迟。该位置位表示在 64 字节的冲突窗口后又发生冲突
COL	[3]	数据包冲突。该位置位表示传输过程中发生冲突
EC	[2]	额外冲突。该位置位表示由于发生了第 16 次冲突（即额外冲突）后，传送被终止
保留	[1-0]	

## 6. 接收控制寄存器 (RCR)

接收控制寄存器，可以控制接收使能，功能描述如表 9.6 所述。

表 9.6 接收控制寄存器 (RX Control Register)

功 能	位	描 述
保留	[7]	
WTDIS	[6]	看门狗定时器禁止。1 禁止，0 使能
DIS_LONG	[5]	丢弃长数据包。1 为丢弃数据包长度超过 1522 字节的数据包
DIS_CRC	[4]	丢弃 CRC 校验错误的数据包
ALL	[3]	忽略所有多点传送
RUNT	[2]	忽略不完整的数据包
PRMSC	[1]	混杂模式 (Promiscuous Mode)
RXEN	[0]	接收使能

## 7. 接收状态寄存器 (RSR)

接收状态寄存器，当有接收中断到来时，可以通过读取该寄存器，进一步了解当前 DM9000 网卡的接收状态，从而确定目前接收的这一帧数据应该如何处理。功能描述如表 9.7 所述。

表 9.7 接收状态寄存器 (RX Status Register)

功 能	位	描 述
RF	[7]	不完整数据帧。该位置位表示接收到小于 64 字节的帧
MF	[6]	多点传送帧。该位置位表示接收到帧包含多点传送地址
LCS	[5]	冲突延迟。该位置位表示在帧接收过程中发生冲突延迟
RWTO	[4]	接收看门狗定时溢出。该位置位表示接收到大于 2048 字节数据帧
PLE	[3]	物理层错误。该位置位表示在帧接收过程中发生物理层错误
AE	[2]	对齐错误 (Alignment)。该位置位表示接收到的帧结尾处不是字节对齐，即不是以字节为边界对齐
CE	[1]	CRC 校验错误。该位置位表示接收到的帧 CRC 校验错误
FOE	[0]	接收 FIFO 缓存溢出。该位置位表示在帧接收时发生 FIFO 溢出

## 8. 接收/发送溢出控制寄存器 (RTFCR)

接收/发送溢出控制寄存器功能描述，如表 9.8 所述。

表 9.8 接收/发送溢出控制寄存器 (RX/TX Flow Control Register)

功 能	位	描 述
TXPO	[7]	1 发送暂停包。发送完成后自动清零，并设置 TX 暂停包时间为 0000H
TXPF	[6]	1 发送暂停包。发送完成后自动清零，并设置 TX 暂停包时间为 FFFFH
TXPEN	[5]	强制发送暂停包使能。按溢出门限最高值使能发送暂停包
BKPA	[4]	背压模式。该模式仅在半双工模式下有效。当接收 SRAM 超过 BPHW 并且接收新数据包时，产生一个拥挤状态
BKPM	[3]	背压模式。该模式仅在半双工模式下有效。当接收 SRAM 超过 BPHW 并且数据包 DA 匹配时，产生一个拥挤状态
RXPS	[2]	接收暂停包状态。只读，清零时允许接收
RXPCS	[1]	接收暂停包当前状态
FLCE	[0]	溢出控制使能控。1 设置使能溢出制模式

## 9. 传送数据长度寄存器

- DM\_TXPLL (0xFC)：传送数据长度低字节寄存器，在发送数据时，该寄存器存放发送的数据长度的低字节。
- DM\_TXPLH (0xFD)：传送数据长度高字节寄存器，在发送数据时，该寄存器存放发送的数据长度的高字节。

## 10. 中断状态寄存器 (ISR)

中断状态寄存器，当一个中断到来时，该寄存器存放着中断类型。DM9000 中断处理函数通过读取该寄存器，得到目前中断信息，从而能够正确调用相应的中断处理子程序。读取该中断状态寄存器之后，还需要将读取结果存放回该寄存器，也就是需要清楚中断状态，否则将无法再次响应中断，功能描述如表 9.9 所述。

表 9.9 中断状态寄存器 (Interrupt Status Register)

功 能	位	描 述
IOMODE	[7-6]	处理器模式。00 为 16 位模式，01 为 32 位模式，10 为 8 位模式，00 保留
LNKCHG	[5]	连接状态改变
UDRUN	[4]	传输 “Underrun”
ROOS	[3]	接收溢出计数器溢出
ROS	[2]	接收溢出
PTS	[1]	数据包传输
PRS	[0]	数据包接收

## 11. 中断掩码寄存器 (IMR)

中断掩码寄存器，该寄存器存放当前 DM9000 使能的中断类型。在该系统中，只让接

收中断使能。利用该寄存器，可以灵活地使 DM9000 屏蔽中断，或者开启中断。例如在发送数据开始时，可以屏蔽中断，在发送结束后，再开启中断，这样可以使 DM9000 工作的稳定性大大提高。功能描述如表 9.10 所述。

表 9.10 中断掩码寄存器 (Interrupt Mask Register)

功 能	位	描 述
PAR	[7]	1 使能指针自动跳回。当 SRAM 的读、写指针超过 SRAM 的大小时，指针自动跳回起始位置。需要驱动程序设置该位，若设置则 REG_F5 (MDRAH) 将自动为 0CH
保留	[6]	
LNKCHGI	[5]	1 使能连接状态改变中断
UDRUNI	[4]	1 使能传输 “Underrun” 中断
ROOI	[3]	1 使能接收溢出计数器溢出中断
ROI	[2]	1 使能接收溢出中断
PTI	[1]	1 使能数据包传输终端
PRI	[0]	1 使能数据包接收中断

以上为 DM9000 (A) 常用寄存器功能的详细介绍，通过对这些寄存器的操作访问，可以实现对 DM9000 的初始化、数据发送、接收等相关操作。而要实现 ARP、IP、TCP 等功能，则需要对相关协议的理解，由编写相关协议或移植协议栈来实现。

#### 9.4.4 功能描述

##### 1. 总线

总线是 ISA 总线兼容模式，8 个 I/O 基址，分别是 300H、310H、320H、330H、340H、350H、360H、370H。I/O 基址与设定引脚或内部 EEPROM 的共同选定。

访问芯片有两个地址端口，分别是地址端口和数据端口。当引脚 CMD 接地时，为地址端口；当引脚 CMD 接高电平时，为数据端口。在访问任何寄存器前，地址端口输入的是数据端口的寄存器地址，寄存器的地址必须保存在地址端口。

##### 2. 存储器直接访问控制

DM9000 提供 DMA (直接存取技术) 来简化对内部存储器的访问。在对内部存储器起始地址完成编程后，然后发出伪读写命令就可以加载当期数据到内部数据缓冲区，可以通过读写命令寄存器来定位内部存储区地址。根据当前总线模式的字长使存储地址自动加 1，下一个地址数据将会自动加载到内部数据缓冲区。要注意的是，在连续突发式的第一次访问时读写命令的内容。

内部存储器空间大少 16K 字节。低 3K 字节单元用作发送包的缓冲区，其他 13K 字节用做接收包的缓冲区。所以在写发送包存储器的时候，当存储器地址越界后，自动跳回 0 地址并置位 IMR 第 7 位。同样，在读接收包存储器的时候，当存储器地址越界后，自动跳回起始地址 0x0c00。

##### 3. 包的发送

有两个指针，顺序命名为指针 1 和指针 2，能同时存储在发送包缓冲区。发送控制寄

存器（02H）控制冗余校验码和填充的插入，其状态分别记录在发送状态寄存器1（03H）和发送状态寄存器2（04H）发送器的起始地址是0x00H，软件或硬件复位后默认是指针1，先通过DMA端口写数据到发送包缓冲区，然后写字节计数长度到字节计数寄存器。

## 9.5 DM9000 网卡驱动程序移植

Linux内核中已经有DM9000网卡驱动，源码文件为drivers/net/dm9000.c。与前面几章移植类似，所要做的工作也是告诉内核DM9000芯片所使用的资源（访问地址、中断号等），使得这些资源可用。本节将主要分析内核源码中的dm9000.c文件。

### 9.5.1 DM9000 网卡连接

由于必须告知内核DM9000芯片所使用的硬件资源，所以移植的首要任务是分析DM9000芯片的硬件连接情况，以获得访问地址、中断号等硬件资源。DM9000芯片在开发板上的连接情况如图9.9所示。

从连接图可以确定：

- (1) 由于用nGCS4作为片选信号，所以访问DM9000的基址是0X20000000，这是物理地址。
- (2) 地址线只有一条，即ADDR2。这是由DM9000的特性决定的，DM9000的地址信号和数据信号是复用的，使用CMD引脚来区分它们，CMD为低电平时，数据总线上传输的是地址信号，CMD为高电平时数据总线上传输的是数据信号。访问DM9000内部寄存器时，需要先将CMD置为低电平，发出地址信号，然后将CMD置为高电平，读写数据。
- (3) 总路线位宽为16位，用nWAIT信号。
- (4) 用EINT7外部中断作为中断引脚。

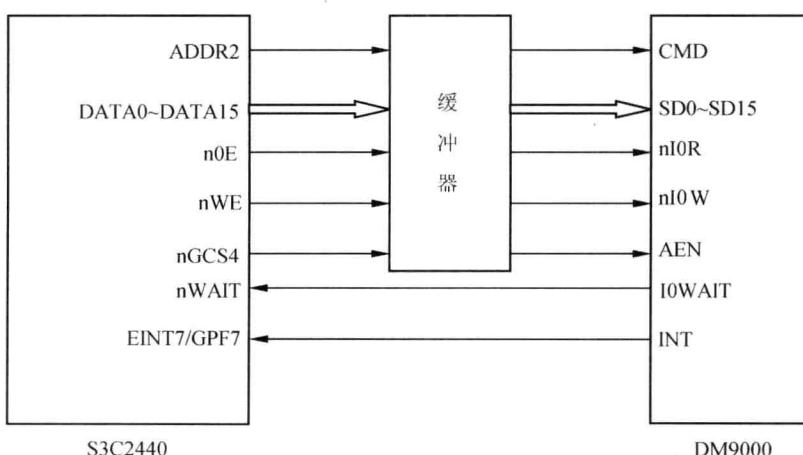


图9.9 DM9000网卡与芯片连接图

对源码的移植必须建立在读懂源码的基础上，所以在介绍驱动移植之前，必须先对内核 DM9000 网卡驱动程序进行分析。

### 9.5.2 驱动分析——硬件的数据结构

在内核源码中用 board\_info 结构体来描述具体的硬件，它保存了一些硬件资源，其定义在 driver/net/dm9000.c 中。

```

typedef struct board_info {
    void __iomem *io_addr; //地址基址，这两个地址是内核虚拟地址，不是物理地址
    void __iomem *io_data; //数据基址
    u16 irq; //中断号

    u16 tx_pkt_cnt; //发包计数
    u16 queue_pkt_len; //队列长度
    u16 queue_start_addr; //队列开始地址
    u16 dbug_cnt;
    u8 io_mode; /*0:word, 2:byte*/
    u8 phy_addr;
    unsigned int flags;
    unsigned int in_suspend :1;
/*下面几个和 include/linux/dm9000.h 中定义的 struct dm9000_plat_data 一样，这个
源码从某种程度上说有点重复定义的嫌疑*/
    int debug_level; //调试级别

    void (*inblk)(void __iomem *port, void *data, int length);
                           //输入方法
    void (*outblk)(void __iomem *port, void *data, int length);
                           //输出方法
    void (*dumpblk)(void __iomem *port, int length);

    struct device *dev; // parent device 指向 platform_device->
device

    struct resource *addr_res; //找到在 devs.c 中定义的资源
    struct resource *data_res;

    struct resource *addr_req; /*根据 addr_res 申请到的资源，上面两个只是知
道了网卡所使用的资源，但是在内核中，物理资源的使用是要经过申请的。*/
    struct resource *data_req;
    struct resource *irq_res;

    struct mutex addr_lock; /*phy and eeprom access lock*/
    spinlock_t lock; //自旋锁

    struct mii_if_info mii; // mii 信息
    u32 msg_enable; //使能标志
} board_info_t;

```

这是一个驱动开发者自定义的结构体，用于保存该设备的相关信息（设备的属性如统计信息、读写操作、占用的 I/O 地址资源、状态）和相关操作，这些属性和操作是该设备的物理抽象。这个结构体最后被挂到了 net\_device 的 priv 成员上，这就是所谓的网络设备

私有成员结构。

### 9.5.3 驱动分析——数据读写函数

由于 DM9000 的地址线和数据线是复用的，所以它有自己特别的读写方法。当要从网卡中某个寄存器读取值时，要先在地址基址中写入要读取的寄存器地址，然后再从数据基址中读出数据，方法如下：

```
static u8
ior(board_info_t * db, int reg)
{
    writeb(reg, db->io_addr);
    return readb(db->io_data);
}
```

当要从网卡中的某个寄存器写入值时，要先在地址基址中写入要写入的寄存器地址，然后再从数据基址中写入要写进去的数据，方法如下：

```
static void
iow(board_info_t * db, int reg, int value)
{
    writeb(reg, db->io_addr);
    writeb(value, db->io_data);
}
```

### 9.5.4 驱动分析——重置网卡

网卡启动前要先对网卡进行重置位，置位方法是向前面所介绍的网络控制寄存器（NCR）的置位位中写入 1，重置函数如下：

```
static void
dm9000_reset(board_info_t * db)
{
    dev_dbg(db->dev, "resetting device\n"); // 调试信息

    /*RESET device*/
    writeb(DM9000_NCR, db->io_addr); // 写入要操作的寄存器地址，这里是 DM9000_NCR
    udelay(200); // 作一下延时
    writeb(NCR_RST, db->io_data); // 写入置位值，这里是 1
    udelay(200);
}
```

### 9.5.5 驱动分析——初始化网卡

DM9000 网卡的初始化工作主要由 driver/net/dm9000.c 中的 dm9000\_probe 来完成，该函数完成的主要工作是获取和申请硬件资源、申请中断号、初始化 net\_device 结构体，最后注册网络设备。这个函数有些长，下面只列出主要代码，读者可自己对照源码进行学习。

```
static int
dm9000_probe(struct platform_device *pdev)
```

```

{

    struct dm9000_plat_data *pdata = pdev->dev.platform_data; /*从传进来的
    平台设备结构中取出设备的 platform_data，在 devs.c 中有定义*/
    struct board_info *db; /*用于指向网卡的私有数据，这也是这个函数要完成的功能，
    下面操作都是为了从传进来的参数 pdev 中探索并获取这个结构的相关信息，pdev->dev.
    platform_data 中包涵了设备的私有信息，一般在 DEV.C 中定义用于描述这个设备，在这
    个驱动中 pdev->dev.platform_data 没什么用处，因为这里的的所有设备信息都是从资源地
    址探索到的*/
    struct net_device *ndev;                                     //网络设备结构体
    const unsigned char *mac_src;
    unsigned long base;
    int ret = 0;
    int iosize;
    int i;
    u32 id_val;

    /*获取*/
    ndev = alloc_etherdev(sizeof (struct board_info));/*分配并初始化一个
    netdevice，传入的参数是给 net_device->priv 用的，net_device->priv 将
    指向这个参数*/
    if (!ndev) {
        dev_err(&pdev->dev, "could not allocate device.\n");
        return -ENOMEM;
    }

    SET_NETDEV_DEV(ndev, &pdev->dev);
    //设置设备的继承关系，ndev->dev.parent = pdev->dev

    dev_dbg(&pdev->dev, "dm9000_probe()");

    /*setup board info structure*/
    db = (struct board_info *) ndev->priv;
    memset(db, 0, sizeof (*db)); //清零网卡私有结构，下面将对里面的成员进行填充

    db->dev = &pdev->dev;           //父设备

    spin_lock_init(&db->lock); //初始化自旋锁
    mutex_init(&db->addr_lock);

    if (pdev->num_resources < 2) {
        /*资源数是否大于等于 2，因为网卡要用到内存资源和中断资源共享至少两种资源。*/
        ret = -ENODEV;
        goto out;
    } else if (pdev->num_resources == 2) {
        base = pdev->resource[0].start;

        if (!request_mem_region(base, 4, ndev->name)) {
            //申请内存资源，资源必须申请后才可用
            ret = -EBUSY;
            goto out;
        }

        ndev->base_addr = base;      //填写基址，这里用的只是物理地址
        ndev->irq = pdev->resource[1].start;      //填写中断资源
        db->io_addr = (void __iomem *)base;         //地址基址
        db->io_data = (void __iomem *) (base + 4); //数据基址
    }
}

```

```

/*ensure at least we have a default set of IO routines*/
dm9000_set_io(db, 2); //设置网卡的默认 I/O 函数，后面将根据实际情况再设置

} else {
    db->addr_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    //获取设备资源
    db->data_res = platform_get_resource(pdev, IORESOURCE_MEM, 1);
    db->irq_res = platform_get_resource(pdev, IORESOURCE_IRQ, 0);

    if (db->addr_res == NULL || db->data_res == NULL ||
        db->irq_res == NULL) {
        dev_err(db->dev, "insufficient resources\n");
        ret = -ENOENT;
        goto out;
    }

    i = res_size(db->addr_res);           //求地址端口资源大小，这里是 4
    db->addr_req = request_mem_region(db->addr_res->start, i,
                                        pdev->name);           //申请 I/O 资源

    if (db->addr_req == NULL) {
        dev_err(db->dev, "cannot claim address reg area\n");
        ret = -EIO;
        goto out;
    }

    db->io_addr = ioremap(db->addr_res->start, i); /*把申请到的 I/O 资源
    (物理地址) 映射到内核并保存映射地址*/
    if (db->io_addr == NULL) {
        dev_err(db->dev, "failed to ioremap address reg\n");
        ret = -EINVAL;
        goto out;
    }

    iosize = res_size(db->data_res);   //这里的返回值作为 DM9000 数据位宽
    db->data_req = request_mem_region(db->data_res->start, iosize,
                                        pdev->name);           //申请数据接口资源

    if (db->data_req == NULL) {
        dev_err(db->dev, "cannot claim data reg area\n");
        ret = -EIO;
        goto out;
    }

    db->io_data = ioremap(db->data_res->start, iosize);
    //映射数据接口为虚拟地址

    if (db->io_data == NULL) {
        dev_err(db->dev, "failed to ioremap data reg\n");
        ret = -EINVAL;
        goto out;
    }

/*fill in parameters for net-dev structure*/

ndev->base_addr = (unsigned long)db->io_addr;
ndev->irq = db->irq_res->start;
/*这里不申请中断线，要在打开时才申请，这样就不会一直占用中断线*/

```

```

/*ensure at least we have a default set of IO routines*/
dm9000_set_io(db, iosize);
    //根据 DM9000 数据位宽设置读写数据帧的函数指针
}

/*check to see if anything is being over-ridden*/
if (pdata != NULL) {
    /*check to see if the driver wants to over-ride the
     *default IO width*/
    /*如果在 devs.c 文件中定义了 platform_device 结构中的 platform_data，则要
     重载这些方法和重设数据位宽，也就是在内核有两个地方可以设置这些数据，一个是 pl-
     atform_data，一个是 platform_device 中的 resource，在移植时可以只设置一
     个地方*/
    if (pdata->flags & DM9000_PLATF_8BITONLY)
        dm9000_set_io(db, 1);           //根据平台数据重设数据位宽

    if (pdata->flags & DM9000_PLATF_16BITONLY)
        dm9000_set_io(db, 2);

    if (pdata->flags & DM9000_PLATF_32BITONLY)
        dm9000_set_io(db, 4);

    /*check to see if there are any IO routine
     * over-rides*/

    if (pdata->inblk != NULL)
        db->inblk = pdata->inblk;//用 devs.c 定义的 platform_device 结构
        中的 platform_data 重载这些方法

    if (pdata->outblk != NULL)
        db->outblk = pdata->outblk;

    if (pdata->dumpblk != NULL)
        db->dumpblk = pdata->dumpblk;

    db->flags = pdata->flags;
}

dm9000_reset(db); //现在可以复位芯片了：dm9000_reset(db); db 中已经包含了
详细的芯片信息*/

/*try two times, DM9000 sometimes gets the first read wrong*/

for (i = 0; i < 8; i++) {           //读取芯片 ID 号
    id_val = ior(db, DM9000_VIDL);
    id_val |= (u32)ior(db, DM9000_VIDH) << 8;
    id_val |= (u32)ior(db, DM9000_PIDL) << 16;
    id_val |= (u32)ior(db, DM9000_PIDH) << 24;

    if (id_val == DM9000_ID)          //判断是否为 0x90000A46
        break;
    dev_err(db->dev, "read wrong id 0x%08x\n", id_val);
}

if (id_val != DM9000_ID) {
    dev_err(db->dev, "wrong id: 0x%08x\n", id_val);
    ret = -ENODEV;
}

```

```

        goto out;
    }

/*from this point we assume that we have found a DM9000*/

/*driver system function*/
ether_setup(ndev);
    //初始化以太网 ndev， 这里设置了以太网的一些通用的（和硬件无关的）参数和方法

/*设置 ndev 的基本操作*/
ndev->open      = &dm9000_open;
ndev->hard_start_xmit = &dm9000_start_xmit;
ndev->tx_timeout      = &dm9000_timeout;
ndev->watchdog_timeo   = msecs_to_jiffies(watchdog);
ndev->stop       = &dm9000_stop;
ndev->set_multicast_list = &dm9000_hash_table;
ndev->ethtool_ops     = &dm9000_ethtool_ops;
ndev->do_ioctl      = &dm9000_ioctl;

#endif CONFIG_NET_POLL_CONTROLLER
ndev->poll_controller = &dm9000_poll_controller;
#endif

db->msg_enable      = NETIF_MSG_LINK;
db->mii.phy_id_mask  = 0x1f;
db->mii.reg_num_mask = 0x1f;
db->mii.force_media   = 0;
db->mii.full_duplex   = 0;
db->mii.dev        = ndev;
db->mii.mdio_read     = dm9000_phy_read;
db->mii.mdio_write    = dm9000_phy_write;

mac_src = "eprom";

/*try reading the node address from the attached EEPROM*/
for (i = 0; i < 6; i += 2)
    dm9000_read_eeprom(db, i / 2, ndev->dev_addr+i);

if (!is_valid_ether_addr(ndev->dev_addr)) {
    /*try reading from mac*/
    mac_src = "chip";
    for (i = 0; i < 6; i++)
        ndev->dev_addr[i] = ior(db, i+DM9000_PAR);
}

if (!is_valid_ether_addr(ndev->dev_addr))
    dev_warn(db->dev, "%s: Invalid ethernet MAC address. Please "
            "set using ifconfig\n", ndev->name);

/*将 ndev 记录于平台设备 platform_dev 中，注册 ndev。这也是 ndev 与 platform_dev
建立联系的地方。可以这么理解，Linux 的设备模型负责的只是设备的管理（检测、启动、移
除），而如何访问这个设备的数据，比如说以字符流模式，块设备方式，网络接口，则定义相
应的 cdev、gendisk、ndev，然后注册到内核。所有的数据访问工作都以这 3 种界面提供。
这里的 ndev 在其他驱动中将换成其他自定义的结构，这里是因为 ndev 包含了 bd*/
platform_set_drvdata(pdev, ndev); //pdev->dev->driver_data=ndev
ret = register_netdev(ndev);

if (ret == 0) {
    DECLARE_MAC_BUF(mac);
}

```

```

        printk("%s: dm9000 at %p,%p IRQ %d MAC: %s (%s)\n",
               ndev->name, db->io_addr, db->io_data, ndev->irq,
               print_mac(mac, ndev->dev_addr), mac_src);
    }
    return 0;

out:
    dev_err(db->dev, "not found (%d).\n", ret);

    dm9000_release_board(pdev, db);
    free_netdev(ndev);

    return ret;
}

```

## 9.5.6 驱动分析——打开和关闭网卡

打开网卡就是激活网络接口，使它能接收来自网络的数据并且传送到网络协议栈的上面，也可以将数据发送到网络上，而网卡的关闭就是使网络接口停止工作。在这里只分析打开函数，关闭函数留给读者自行分析。DM9000 网卡的打开函数分析如下：

```

static int
dm9000_open(struct net_device *dev)
{
    board_info_t *db = (board_info_t *) dev->priv;           //获取网卡数据结构
    unsigned long irqflags = db->irq_res->flags & IRQF_TRIGGER_MASK;

    if (netif_msg_ifup(db))                                     //设置使能标志
        dev_dbg(db->dev, "enabling %s\n", dev->name);
    if (irqflags == IRQF_TRIGGER_NONE) {
        dev_warn(db->dev, "WARNING: no IRQ resource flags set.\n");
        irqflags = DEFAULT_TRIGGER;
    }

    irqflags |= IRQF_SHARED;                                     //设置中断共享

    if (request_irq(dev->irq, &dm9000_interrupt, irqflags, dev->name, dev))
        /*申请中断线，要在打开时才申请，这样就不会一直占用中断线*/
        return -EAGAIN;

    /*Initialize DM9000 board*/
    dm9000_reset(db);             //重置网卡，这个函数在前面讲过了
    dm9000_init_dm9000(dev);      //初始化网卡，这里主要是对 DM9000 网卡的寄存器进行设置

    /*Init driver variable*/
    db->dbuf_cnt = 0;

    mii_check_media(&db->mii, netif_msg_link(db), 1);
    netif_start_queue(dev); /*netif_start_queue 用来告诉上层网络协定这个驱动
                           程序还有空的缓冲区可用，请把下一个封包送进来。*/
}

return 0;
}

```

### 9.5.7 驱动分析——数据包的发送与接收

在驱动程序层次上，数据包的发送和接收都是通过底层对硬件的读写来完成的。当网络上的数据到来时，将触发硬件中断，根据注册的中断向量表确定处理函数，进入中断向量处理函数，将数据送到上层协议进行处理或者转发出去。

当协议层已经封装好上层协议数据的 `skb_buffer` 后，将调用 `dm9000_start_xmit` (`struct sk_buff *skb, struct net_device *dev`) 函数把数据发出。数据的发送函数分析如下：

```

static int
dm9000_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    unsigned long flags;
    board_info_t *db = (board_info_t *) dev->priv;           //获取网卡数据

    dm9000_dbg(db, 3, "%s:\n", __func__);

    if (db->tx_pkt_cnt > 1)                                     //网卡正忙，返回 1
        return 1;

    spin_lock_irqsave(&db->lock, flags);                         //获取自旋锁

    /*Move data to DM9000 TX RAM*/
    writeb(DM9000_MWCMD, db->io_addr);                         //存储器读地址自动增加的读数据命令

    (db->outblk)(db->io_data, skb->data, skb->len);          //把数据发送出去
    dev->stats.tx_bytes += skb->len;                            //统计发送字节数

    db->tx_pkt_cnt++;                                         //正在发送中的数据计数加 1
    /*TX control: First packet immediately send, second packet queue*/
    /*如果只有当前包发送，写指令，写数据帧，发送包。如果多于一个数据包正在发送，当前帧不发送。*/
    if (db->tx_pkt_cnt == 1) {
        /*Set TX length to DM9000*/
        iow(db, DM9000_TXPLL, skb->len);
        iow(db, DM9000_TXPLH, skb->len >> 8);

        /*Issue TX polling command*/
        iow(db, DM9000_TCR, TCR_TXREQ);             /*Cleared after TX complete*/

        dev->trans_start = jiffies; /*save the time stamp*/
    } else {
        /*Second packet*/
        db->queue_pkt_len = skb->len;
        netif_stop_queue(dev);                      //网卡停止接收数据
    }

    spin_unlock_irqrestore(&db->lock, flags); //取消自旋

    /*free this SKB*/
    dev_kfree_skb(skb);                          //执行释放内存空间的动作
}

return 0;
}

```

网络数据包到达，DM9000 自动接收并存放在 DM9000 内部 RAM 中，产生中断。在中断处理中识别中断原因，并调用接收处理函数。接收函数分析如下：

```

static void
dm9000_rx(struct net_device *dev)
{
    board_info_t *db = (board_info_t *) dev->priv;
    struct dm9000_rxhdr rxhdr;
    struct sk_buff *skb;
    u8 rxbyte, *rdptr;
    bool GoodPacket;
    int RxLen;

    /*Check packet ready or not*/
    do {
        /*读取芯片相关寄存器，确认 DM9000 正确的收到一帧数据。*/
        ior(db, DM9000_MRCMDX); /*Dummy read*/

        /*Get most updated data*/
        rxbyte = readb(db->io_data);

        /*Status check: this byte must be 0 or 1*/
        if (rxbyte > DM9000_PKT_RDY) {
            dev_warn(db->dev, "status check fail: %d\n", rxbyte);
            iow(db, DM9000_RCR, 0x00); /*Stop Device*/
            iow(db, DM9000_ISR, IMR_PAR); /*Stop INT request*/
            return;
        }

        if (rxbyte != DM9000_PKT_RDY)
            return;

        /*A packet ready now & Get status/length*/
        GoodPacket = true;
        writeb(DM9000_MRCMD, db->io_addr);

        (db->inblk)(db->io_data, &rxhdr, sizeof(rxhdr));

        RxLen = le16_to_cpu(rxhdr.RxLen);

        if (netif_msg_rx_status(db))
            dev_dbg(db->dev, "RX: status %02x, length %04x\n",
                   rxhdr.RxStatus, RxLen);

        /*Packet Status check*/
        if (RxLen < 0x40) {
            GoodPacket = false;
            if (netif_msg_rx_err(db))
                dev_dbg(db->dev, "RX: Bad Packet (runt)\n");
        }

        if (RxLen > DM9000_PKT_MAX) {
            dev_dbg(db->dev, "RST: RX Len:%x\n", RxLen);
        }

        if (rxhdr.RxStatus & 0xbf) {

```

```

GoodPacket = false;
if (rxhdr.RxStatus & 0x01) {
    if (netif_msg_rx_err(db))
        dev_dbg(db->dev, "fifo error\n");
    dev->stats.rx_fifo_errors++;
}
if (rxhdr.RxStatus & 0x02) {
    if (netif_msg_rx_err(db))
        dev_dbg(db->dev, "crc error\n");
    dev->stats.rx_crc_errors++;
}
if (rxhdr.RxStatus & 0x80) {
    if (netif_msg_rx_err(db))
        dev_dbg(db->dev, "length error\n");
    dev->stats.rx_length_errors++;
}
}

/*Move data from DM9000*/
/*申请 skb_buffer, 将数据从 DM9000 中复制到 skb_buffer 中*/
if (GoodPacket
&& ((skb = dev_alloc_skb(RxLen + 4)) != NULL)) {
    skb_reserve(skb, 2);
    rdptra = (u8 *) skb_put(skb, RxLen - 4);

    /*Read received packet from RX SRAM*/

    (db->inblk)(db->io_data, rdptra, RxLen);
    dev->stats.rx_bytes += RxLen;

    /*Pass to upper layer*/
    skb->protocol = eth_type_trans(skb, dev);      //去除以太网头
    netif_rx(skb);                                //把 skb_buffer 交给上层协议
    dev->stats.rx_packets++;

} else {
    /*need to dump the packet's data*/

    (db->dumpblk)(db->io_data, RxLen);
}

} while (rxbyte == DM9000_PKT_RDY);
}

```

### 9.5.8 DM9000 网卡驱动程序移植

读懂了 DM9000 网卡驱动程序源码后，就可以开始移植这个驱动了，具体方法如下所述。

#### 1. 定义网卡设备

硬件的使用需要知道硬件所用到的资源，如 I/O 端口和中断号等，在 arch/arm/plat-s3c24xx 的 devs.c 中添加 DM9000 用到的地址端口、数据端口和中断号，这些都要在了解了硬

件连接后才知道用到什么资源，读者可以回头去阅读相关的硬件连接图。代码如下：

```
static struct resource s3c_dm9000_resource[] = {
    [0] = {
        .start = 0x20000000, // 对应电路图 nGCS4
        .end   = 0x20000003,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = 0x20000004, // 对应 ADDR2
        .end   = 0x20000007, // 0x3f
        .flags = IORESOURCE_MEM,
    },
    [2] = {
        .start = IRQ_EINT7, // 对应图中的 EINT7
        .end   = IRQ_EINT7,
        .flags = IORESOURCE_IRQ,
    }
};
```

添加 DM9000 平台数据，该数据用于内核传递给驱动程序，这个结构体在 arch/arm/mach-s3c2410/devs.c 定义 platform\_device 时，将被挂载到 struct platform\_device 结构成员 dev.platform\_data 中。DM9000 平台数据结构在 include/linux/dm9000.h 中的定义如下：

```
struct dm9000_plat_data {
    unsigned int flags;

    /* allow replacement IO routines */

    void (*inblk)(void __iomem *reg, void *data, int len);
    void (*outblk)(void __iomem *reg, void *data, int len);
    void (*dumpblk)(void __iomem *reg, int len);
};
```

在 arch/arm/plat-s3c24xx 的 devs.c 中，只添加以下代码就可以了。

```
static struct dm9000_plat_data s3c_device_dm9000_platdata = {
    .flags= DM9000_PLATF_16BITONLY
};

struct platform_device s3c_device_dm9000 = {
    .name= "dm9000", // 设备名字
    .id= -1, // 设备 ID 让内核自动编号
    .num_resources= ARRAY_SIZE(s3c_dm9000_resource), // 用到的资源数
    .resource= s3c_dm9000_resource, // 用到的资源，在前面已经定义了，这里引用它
    .dev= {
        .platform_data = &s3c_device_dm9000_platdata, // 引用平台数据
    }
};

EXPORT_SYMBOL(s3c_device_dm9000); // 让其他文件可以引用到这里定义的变量
/*****************whs add 2009-3-28 12:06-----end *****************/
```

## 2. 添加变量声明

在前面步骤中，定义了 s3c\_device\_dm9000，并用 EXPORT\_SYMBOL(s3c\_device\_dm

9000)使之变为全局变量，所以在这里要添加它的声明，在 arch/arm/plat-S3C24xx/include/plat/devs.hc 添加声明如下：

```
extern struct platform_device s3c_device_dm9000;
```

### 3. 添加平台设备列表

内核中用 smdk2440\_devices 初始化列表保存系统启动时要初始化的设备，所以在这里要把网卡设备加进去，在 arch/arm/mach-s3c2440/mach-smdk2440.c 中将 s3c\_device\_dm9000 添加到平台设备列表中。

```
static struct platform_device *smdk2440_devices[] __initdata = {
    &s3c_device_usb,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c,
    &s3c_device_iis,
    &s3c_device_dm9000,
};
```

### 4. 修改dm9000.c

经过上述步骤，DM9000 网卡设备就已经成功注册进驱动核心。之后还需要做两方面的工作：设置芯片 MAC 地址，使能 DM9000 的中断，下面对 drivers/net/dm9000.c 中初始化函数进行修改。

根据 2440 资料，有以下几个地方需要设置。

(1) 设置 GPGCON 使 GPG1 功能设置为 EINT7，可以用以下函数实现：

```
s3c2410_gpio_cfgpin(S3C2410_GPG1, S3C2410_GPF3_EINT7);
```

(2) 外部中断 EXTINT1 的[6:4]位置 100 上升沿触发中。

因为要用到一些 GPIO 寄存器地址，所以必须在文件的开头把相关的 mach/regs-gpio.h 文件包含进来。在 dm9000.c 的开始添加如下定义：

```
#include <mach/regs-gpio.h>
static void *extint1; //中断寄存器地址
static void *intmsk; //中断屏蔽寄存器地址
#define EINTMASK (0x560000a4) //外部中断屏蔽
#define EXTINT1 (0x5600008c) //外部中断方式
#define INTMSK (0x4A000008) //中断屏蔽
```

现在开始设置芯片的 MAC 地址和使能 DM9000 的中断，在 dm9000.c 中初始化函数 dm9000\_probe() 的 register\_netdev 前添加如下代码：

```
memcpy(ndev->dev_addr, "\0andy1", 6); //MAC 地址
extint1=ioremap_nocache(EXTINT1, 4); //映射为虚拟地址
intmsk=ioremap_nocache(INTMSK, 4);
s3c2410_gpio_cfgpin(S3C2410_GPF7, S3C2410_GPF7_EINT7); //设置外部中断 7
writel(readl(extint1)|0x40, extint1); //上升沿
writel(readl(intmsk)&0xffff1, intmsk); //取消映射
iounmap(intmsk);
iounmap(extint1);
```

## 5. 编译内核

进入源码目录，输入 make menuconfig 进入内核的配置选项菜单，按以下顺序选择选项：Device Drivers-->Network device support-->Ethernet(10 or 100Mbit) -->DM9000 support，如图 9.10 所示。

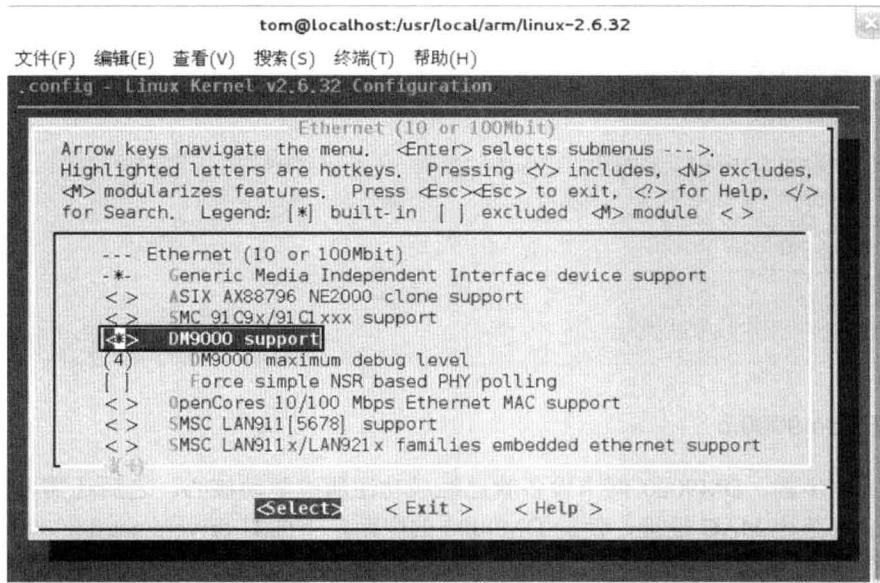


图 9.10 内核配置选项图

## 9.6 小结

本章主要在分析 DM9000 硬件操作和内核中自带的 DM9000 驱动源程序的基础上，讲述了 DM9000 网卡驱动的移植。做网格驱动程序移植主要是要了解内核中网络设备驱动程序的体系结构，所以，本章的 9.2 节是后面移植工作的基础，读者要认真掌握，特别是网络数据结构的主要成员要掌握其代表的具体含义。

# 第 10 章 音频设备驱动程序移植

音频驱动程序广泛地应用在嵌入式产品中，目前 PDA、手机都有音频和视频播放功能。随着终端产品逐渐融入工作和生活，带有音频功能的嵌入式产品将更具吸引力。本章将介绍音频设备接口分类、Linux 音频设备驱动、音频设备应用程序编写、音频设备驱动移植，最后介绍音频播放程序 madplay 的移植。

## 10.1 音频设备接口

Linux 支持的音频设备接口包括 PCM、IIS 和 AC97。这 3 种接口分别应用在不同的场合，IIS 接口多应用在 MP3 随身听、CD 等产品中；PCM 接口多应用在移动电话中；AC97 接口多用在 PDA 中。

### 10.1.1 PCM（脉冲编码调制）接口

PCM 接口针对不同的数字音频子系统，用于数字转换的接口，它是最简单的音频接口，该接口由时钟脉冲（BCLK）、帧同步信号（FS）及接收数据（DR）和发送数据（DX）组成。在 FS 信号的上升沿，数据传输从 MSB（Most Significant Bit）字开始，FS 频率等于采样频率。在 FS 信号之后开始传输数据字，按顺序进行传输单个的数据位，每时钟周期传输 1 个数据字。发送 MSB 时，首先将信号的等级降到最低，以避免在不同终端的接口使用不同的数据方案时造成 MSB 的丢失。

PCM 接口的特点是容易实现，原则上能够支持任何数据方案和任何采样频率，但需要每个音频通道获得一个独立的数据队列。

### 10.1.2 IIS（Inter-IC Sound）接口

IIS 接口在 20 世纪 80 年代首先被飞利浦用于消费音频，并在一个称为 LRCLK（Left/Right CLOCK）的信号机制中经过多路转换，将两路音频信号合成单一的数据队列。当 LRCLK 信号为高时，左声道数据被传输；LRCLK 信号为低时，右声道数据被传输。

IIS 接口的特点：与 PCM 相比，IIS 接口更适用于立体声系统。对于多通道系统，在同样的 BCLK 和 LRCLK 条件下，也可以并行执行几个数据队列。

### 10.1.3 AC97（Audio Codec 1997）接口

AC97（Audio Codec 1997）是以 Intel 为首的 5 个 PC 厂商（Intel、Creative Labs、NS、

Analog Device 与 Yamaha) 共同提出的规格标准。与 PCM 和 IIS 不同, AC97 除了是一种数据格式还具有控制功能, 用于音频编码的内部架构规范。AC97 采用 AC-Link 与外部的编解码器相连, AC-Link 接口包括位时钟 (BITCLK)、同步信号校正 (SYNC) 和从编码到处理器及从处理器中解码 (SDATDIN 与 SDATAOUT) 的数据队列。AC97 数据帧以 SYNC 脉冲开始, 包括 12 个 20 位时间段 (时间段为标准中定义的不同的目的服务) 及 16 位 tag 段, 共计 256 个数据序列。例如, 时间段 1 和 2 用于访问编码的控制寄存器, 而时间段 3 和 4 分别负载左、右两个音频通道。tag 段表示其他段中哪一个包含有效数据。把帧分成时间段, 使传输控制信号和音频数据仅通过 4 根线到达 9 个音频通道或转换成其他数据流成为可能。

AC97 的特点: 与控制接口分离的 IIS 方案相比, AC97 明显减少了整体管脚数。它是一种数据格式, 还具有控制功能。

#### 10.1.4 Linux 音频设备驱动框架

针对音频设备, Linux 内核附有两类音频设备驱动框架: OSS (Open Sound System) 和 ALSA (Advanced Linux Sound Architecture)。前者包含 dev/dsp 和 dev/mixer 字符设备接口, 在用户空间的编程中, 使用文件操作; 后者以 card 和组件 (pcm、mixer 等) 为主线, 在用户空间的编程中不使用文件接口而使用 alsalib。在音频设备驱动中, 几乎都用到 DMA, DMA 的缓冲区会被分割成多段, 每次 DMA 操作进行其中的一个段。OSS 驱动框架的阻塞读写操作具有流控能力, 在用户空间不需要进行流量方面的定时工作, 但是它需要及时地写 (播放) 和读 (录音), 避免缓冲区 underflow 或 overflow。

在内核配置时, 选择 Device Drivers | Sound card support 命令进入 Sound card support 配置窗口。该窗口中包含 OSS 与 ALSA 驱动架构配置选择, 如图 10.1 所示。

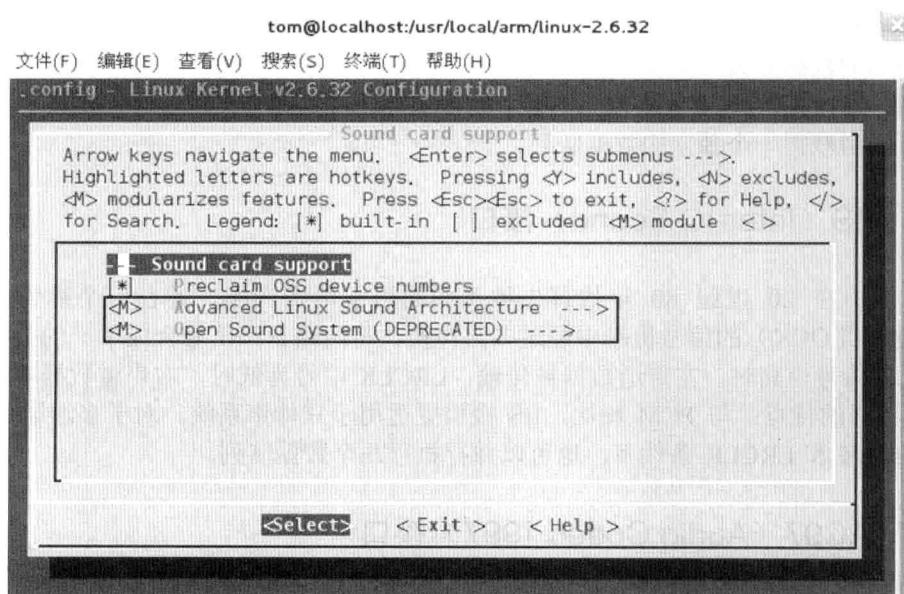


图 10.1 内核配置中 OSS 与 ALSA 驱动架构

**注意：**overflow 是指应用程序读取数据的速率低于声卡的采样速率，多余的数据被丢弃；underflow 是指应用程序读取数据的速率高于声卡的采样速率，那么在新的数据到来之前，声卡将会阻塞应用程序的请求。

前面提到了 Linux 两类音频驱动框架：OSS 和 ALSA，下面将分别介绍这两类驱动架构，并且通过代码进行分析其如何实现录音和播放等功能。

## 10.2 Linux 音频设备驱动——OSS 驱动框架

OSS (Open Sound System) 是 UNIX 平台上一个统一的音频接口。为了在不同的平台之间移植代码，OSS 定义了一套 API，在一个平台上的代码移植到另一个平台上时，不需要重新修改代码，重新编译就可以使用原程序。

### 10.2.1 OSS 驱动架构硬件

数字音频（有时也称 CODEC、PCM、DSP、ADC/DAC 设备）接口，用来实现录音（将模拟信号转变为数字信号）和播放声音（将数字信号转变为模拟信号）的功能。它的主要参数有采样频率（电话为 8K，DVD 为 96K）、channel 数目（单声道，立体声）、采样分辨率（8-bit, 16-bit），对应的设备文件为 /dev/dsp。OSS 驱动支持的硬件接口有以下几种。

- mixer（混频器）接口：用来控制多个输入、输出的音量，也控制输入（microphone, line-in, CD）之间的切换，对应的设备文件为 /dev/mixer。
- synthesizer（合成器）接口：通过一些预先定义好的波形来合成声音，有时用在游戏中声音效果的产生。
- MIDI（Musical Instrument Data Interface）接口：MIDI 接口是为了连接舞台上的 synthesizer、键盘、道具、灯光控制器的一种串行接口。

### 10.2.2 OSS 驱动架构代码

当 vwsnd 驱动模块被加载时函数 init\_vwsnd() 被调用，音频驱动被初始化，驱动初始化过程会查找硬件配置和匹配相应的驱动程序。函数 init\_vwsnd() 的主要内容如下：

```
static int __init init_vwsnd(void)
{
    // 驱动的探针函数
    if (!probe_vwsnd(&the_hw_config))
        return -ENODEV;
    // 驱动附着函数
    err = attach_vwsnd(&the_hw_config);
}
```

在函数 attach\_vwsnd() 中，调用了对数字音频设备的注册函数 register\_sound\_dsp() 和对

混频器接口的注册函数 register\_sound\_mixer()。函数的主要内容如下：

```

static int __init attach_vwsnd(struct address_info *hw_config)
{
    vwsnd_dev_t *devc = NULL;
    int err = -ENOMEM;
    //为 devc 分配空间
    devc = kmalloc(sizeof(vwsnd_dev_t), GFP_KERNEL);
    if (devc == NULL)
        goto fail0;
    //下面是给结构体指针 devc 各个字段赋值
    err = li_create(&devc->lith, hw_config->io_base);
    if (err)
        goto fail1;

    init_waitqueue_head(&devc->open_wait);

    devc->rport.hwbuf_size = HWBUF_SIZE;
    devc->rport.hwbuf_vaddr = __get_free_pages(GFP_KERNEL, HWBUF_ORDER);
    if (!devc->rport.hwbuf_vaddr)
        goto fail2;
    devc->rport.hwbuf = (void *) devc->rport.hwbuf_vaddr;
    devc->rport.hwbuf_paddr = virt_to_phys(devc->rport.hwbuf);

    /*设置输入侧的 DMA 基地址，保持它到驱动被卸载前*/
    li_writel(&devc->lith, LI_COMM1_BASE,
              devc->rport.hwbuf_paddr >> 8 | 1 << (37 - 8));

    devc->wport.hwbuf_size = HWBUF_SIZE;
    devc->wport.hwbuf_vaddr = __get_free_pages(GFP_KERNEL, HWBUF_ORDER);
    if (!devc->wport.hwbuf_vaddr)
        goto fail3;
    devc->wport.hwbuf = (void *) devc->wport.hwbuf_vaddr;
    devc->wport.hwbuf_paddr = virt_to_phys(devc->wport.hwbuf);
    DBGPF("wport hwbuf = 0x%p\n", devc->wport.hwbuf);

    DBGDO(shut_up++);
    err = ad1843_init(&devc->lith);
    DBGDO(shut_up--);
    if (err)
        goto fail4;

    /*安装中断处理程序*/
    err = request_irq(hw_config->irq, vwsnd_audio_intr, 0, "vwsnd", devc);
    if (err)
        goto fail5;

    /*注册 dsp 设备驱动程序*/
    devc->audio_minor = register_sound_dsp(&vwsnd_audio_fops, -1);
    if ((err = devc->audio_minor) < 0) {
        DBGDO(prtink(KERN_WARNING
                     "attach_vwsnd: register_sound_dsp error %d\n",
                     err));
        goto fail6;
    }
    /*注册 mixer 设备驱动程序*/
    devc->mixer_minor = register_sound_mixer(&vwsnd_mixer_fops,
                                              devc->audio_minor >> 4);
}

```

```

if ((err = devc->mixer_minor) < 0) {
    DBGDO	printk(KERN_WARNING
        "attach_vwsnd: register_sound_mixer error %d\n",
        err));
    goto fail7;
}

/*Squirrel away device indices for unload routine.*/
hw_config->slots[0] = devc->audio_minor;
/*对设备执行初始化工作*/
mutex_init(&devc->open_mutex);
mutex_init(&devc->io_mutex);
mutex_init(&devc->mix_mutex);
devc->open_mode = 0;
spin_lock_init(&devc->rport.lock);
init_waitqueue_head(&devc->rport.queue);
devc->rport.swstate = SW_OFF;
devc->rport.hwstate = HW_STOPPED;
devc->rport.flags = 0;
devc->rport.swbuf = NULL;
spin_lock_init(&devc->wport.lock);
init_waitqueue_head(&devc->wport.queue);
devc->wport.swstate = SW_OFF;
devc->wport.hwstate = HW_STOPPED;
devc->wport.flags = 0;
devc->wport.swbuf = NULL;

/*注册成功后，将新设备连接到本地设备。*/
devc->next_dev = vwsnd_dev_list;
vwsnd_dev_list = devc;
return devc->audio_minor;
/*如果注册、分配空间、安装等过程失败，则释放这些过程分配的资源。*/
fail7:
    unregister_sound_dsp(devc->audio_minor);
fail6:
    free_irq(hw_config->irq, devc);
fail5:
    free_pages(devc->wport.hwbuf_vaddr, HWBUF_ORDER);
fail3:
    free_pages(devc->rport.hwbuf_vaddr, HWBUF_ORDER);
fail2:
    li_destroy(&devc->lith);
fail1:
    kfree(devc);
fail0:
    return err;
}

```

OSS 的驱动程序在 sound/oss 目录下，核心代码文件为 soundcard.c。该文件中定义了打开设备文件、读设备文件（录音）、写设备文件（播放）、关闭设备文件等功能。

### 10.2.3 OSS 初始化函数 oss\_init()

在向内核注册该模块的时候调用该函数对 OSS 进行初始化工作，也就是当使用 insmod soundcore.ko 加载驱动模块时调用该函数。

```

static int __init oss_init(void)
{
    int             err;
    int i, j;

#ifdef CONFIG_PCI
    if(dmabug)
        isa_dma_bridge_buggy = dmabug;
#endif
    /*函数create_special_devices()最终调用函数register_sound_special_device()注册声音结点，根据第二个参数来指定声音结点的类型。这里指定注册了两个设备sequencer和sequencer2。*/
    err = create_special_devices();
    if (err) {
        printk(KERN_ERR "sound: driver already loaded/include in kernel
\n");
        return err;
    }

    /*Protecting the innocent*/
    sound_dmap_flag = (dmabuf > 0 ? 1 : 0);
    /*创建设备列表中的设备，并且注册它到系统文件中。*/
    for (i = 0; i < ARRAY_SIZE(dev_list); i++) {
        device_create(sound_class, NULL,
                      MKDEV(SOUND_MAJOR, dev_list[i].minor), NULL,
                      "%s", dev_list[i].name);

        if (!dev_list[i].num)
            continue;
        /*如果设备列表的某项数目多于1个则创建剩下的设备，并且注册它们到系统文件中*/
        for (j = 1; j < *dev_list[i].num; j++)
            device_create(sound_class, NULL,
                          MKDEV(SOUND_MAJOR,
                                dev_list[i].minor + (j*0x10)),
                          NULL, "%s%d", dev_list[i].name, j);
    }

    if (sound_nblocks >= 1024)
        printk(KERN_ERR "Sound warning: Deallocation table was too small.
\n");
}

return 0;
}

```

#### 10.2.4 OSS 释放函数 oss\_cleanup()

从内核中注销模块OSS的时候调用该函数对OSS进行清理工作，也就是当使用rmmod soundcore.ko 卸载驱动模块时调用该函数。

```

static void __exit oss_cleanup(void)
{
    int i, j;
    /*删除函数device_create()创建的设备.*/
    for (i = 0; i < ARRAY_SIZE(dev_list); i++) {
        device_destroy(sound_class, MKDEV(SOUND_MAJOR, dev_list[i].minor));
        if (!dev_list[i].num)

```

```

        continue;
    for (j = 1; j < *dev_list[i].num; j++)
        device_destroy(sound_class, MKDEV(SOUND_MAJOR, dev_list[i]
            .minor + (j*0x10)));
}
/*在初始化函数 oss_init() 中调用函数 create_special_devices ()，在函数
create_special_devices ()中分别注册了设备 sequencer 和 sequencer2，在注销函
数中对应注销这两个设备的驱动程序。*/
unregister_sound_special(1);
unregister_sound_special(8);
/*停止定时器*/
sound_stop_timer();
/*音序器卸载，释放资源*/
sequencer_unload();
/*释放 DMA 资源*/
for (i = 0; i < MAX_DMA_CHANNELS; i++)
    if (dma_alloc_map[i] != DMA_MAP_UNAVAIL) {
        printk(KERN_ERR "Sound: Hmm, DMA%d was left allocated - fixed\n",
            i);
        sound_free_dma(i);
    }
/*释放静态分配的内存资源*/
for (i = 0; i < sound_nblocks; i++)
    vfree(sound_mem_blocks[i]);
}

```

## 10.2.5 打开设备文件函数 sound\_open()

当驱动模块注册到系统中后，调用系统的 open() 函数就会调用函数 sound\_open() 实现打开设备文件。

```

static int sound_open(struct inode *inode, struct file *file)
{
    int dev = iminor(inode);
    int retval;

    DEB	printk("sound_open(dev=%d)\n", dev));
    if ((dev >= SND_NDEVS) || (dev < 0)) {
        printk(KERN_ERR "Invalid minor device %d\n", dev);
        return -ENXIO;
    }
    switch (dev & 0x0f) {
/*音频设备接口为控制设备即混频器*/
        case SND_DEV_CTL:
            dev >>= 4;
            if (dev >= 0 && dev < MAX_MIXER_DEV && mixer_devs[dev] == NULL) {
                request_module("mixer%d", dev);
            }
            if (dev && (dev >= num_mixers || mixer_devs[dev] == NULL))
                return -ENXIO;

            if (!try_module_get(mixer_devs[dev]->owner))
                return -ENXIO;
            break;
/*打开的设备接口为音序器*/
        case SND_DEV_SEQ:

```

```

case SND_DEV_SEQ2:
    if ((retval = sequencer_open(dev, file)) < 0)
        return retval;
    break;
/*打开的设备为 MIDI*/
case SND_DEV_MIDIN:
    if ((retval = MIDIBuf_open(dev, file)) < 0)
        return retval;
    break;
/*打开的设备接口为 AUDIO 或者 DSP*/
case SND_DEV_DSP:
case SND_DEV_DSP16:
case SND_DEV_AUDIO:
    if ((retval = audio_open(dev, file)) < 0)
        return retval;
    break;
default:
    printk(KERN_ERR "Invalid minor device %d\n", dev);
    return -ENXIO;
}

return 0;
}

```

## 10.2.6 录音函数 sound\_read()

在录音时，调用函数 sound\_read()实现读设备文件。

```

static ssize_t sound_read(struct file *file, char __user *buf, size_t count,
loff_t *ppos)
{
    int dev = iminor(file->f_path.dentry->d_inode);
    int ret = -EINVAL;

    /*锁住内核*/
    lock_kernel();

    DEB	printk("sound_read(dev=%d, count=%d)\n", dev, count));
    /*设备接口类型为 AUDIO 或者 DSP 时，调用函数 audio_read () 读取设备文件数据到 buf 中*/
    switch (dev & 0x0f) {
    case SND_DEV_DSP:
    case SND_DEV_DSP16:
    case SND_DEV_AUDIO:
        ret = audio_read(dev, file, buf, count);
        break;
    /*设备接口类型为 音序器时，调用函数 sequencer_read () 读取设备文件数据到 buf 中*/
    case SND_DEV_SEQ:
    case SND_DEV_SEQ2:
        ret = sequencer_read(dev, file, buf, count);
        break;
    /*设备接口类型为 MIDI 时，调用函数 MIDIBuf_read () 读取设备文件数据到 buf*/
    case SND_DEV_MIDIN:
        ret = MIDIBuf_read(dev, file, buf, count);
    }
}

```

```

    }
/*完成读取后解锁内核*/
unlock_kernel();
return ret;
}
}

```

### 10.2.7 播放函数 sound\_write()

在播放音频时，调用函数 sound\_write()向设备文件执行写操作。

```

static ssize_t sound_write(struct file *file, const char __user *buf, size_t
count, loff_t *ppos)
{
    int dev = iminor(file->f_path.dentry->d_inode);
    int ret = -EINVAL;
    /*锁住内核*/
    lock_kernel();
    DEB	printk("sound_write(dev=%d, count=%d)\n", dev, count));
    switch (dev & 0x0f) {
        /*设备接口类型为音序器时，调用函数 sequencer_write ()将 buf 中数据写到设备文
件*/
        case SND_DEV_SEQ:
        case SND_DEV_SEQ2:
            ret = sequencer_write(dev, file, buf, count);
            break;
        /*设备接口类型为 AUDIO 或者 DSP 时，调用函数 audio_write ()将 buf 中数据写到设备
文件*/
        case SND_DEV_DSP:
        case SND_DEV_DSP16:
        case SND_DEV_AUDIO:
            ret = audio_write(dev, file, buf, count);
            break;
        /*设备接口类型为 MIDI 时，调用函数 MIDIBuf_write ()将 buf 中数据写到设备文件*/
        case SND_DEV_MIDIN:
            ret = MIDIBuf_write(dev, file, buf, count);
            break;
    }
    unlock_kernel();
    return ret;
}

```

### 10.2.8 控制函数 sound\_ioctl()

函数 sound\_ioctl()控制功能包括音量控制、低音控制、高音控制、FM 合成器控制、录音音量、播放音量、输入增益、输出增益等控制。

```

static int sound_ioctl(struct inode *inode, struct file *file,
                      unsigned int cmd, unsigned long arg)
{
    int len = 0, dtype;
    int dev = iminor(inode);
    void __user *p = (void __user *)arg;

```

```

if (_SIOC_DIR(cmd) != _SIOC_NONE && _SIOC_DIR(cmd) != 0) {
/* Have to validate the address given by the process.
*/
    len = _SIOC_SIZE(cmd);
    if (len < 1 || len > 65536 || !p)
        return -EFAULT;
    if (_SIOC_DIR(cmd) & _SIOC_WRITE)
        if (!access_ok(VERIFY_READ, p, len))
            return -EFAULT;
    if (_SIOC_DIR(cmd) & _SIOC_READ)
        if (!access_ok(VERIFY_WRITE, p, len))
            return -EFAULT;
}
DEB	printk("sound_ioctl(dev=%d, cmd=0x%lx, arg=0x%lx)\n", dev, cmd,
arg));
if (cmd == OSS_GETVERSION)
    return __put_user(SOUND_VERSION, (int __user *)p);
/*如果命令的类型为 Mixer, 调用 sound_mixer_ioctl ()*/
if (_IOC_TYPE(cmd) == 'M' && num_mixers > 0 && /*Mixer ioctl*/
(dev & 0x0f) != SND_DEV_CTL) {
    dtype = dev & 0x0f;
    switch (dtype) {
    case SND_DEV_DSP:
    case SND_DEV_DSP16:
    case SND_DEV_AUDIO:
        return sound_mixer_ioctl(audio_devs[dev >> 4]->mixer_dev, cmd,
p);
    default:
        return sound_mixer_ioctl(dev >> 4, cmd, p);
    }
}
switch (dev & 0x0f) {
/*设备接口类型为 AUDIO 或者 DSP 时, 调用函数 sound_mixer_ioctl ()*/
case SND_DEV_CTL:
    if (cmd == SOUND_MIXER_GETLEVELS)
        return get_mixer_levels(p);
    if (cmd == SOUND_MIXER_SETLEVELS)
        return set_mixer_levels(p);
    return sound_mixer_ioctl(dev >> 4, cmd, p);
/*设备接口类型为音序器时, 调用函数 sequencer_ioctl ()*/
case SND_DEV_SEQ:
case SND_DEV_SEQ2:
    return sequencer_ioctl(dev, file, cmd, p);
/*设备接口类型为 AUDIO 或者 DSP 时, 调用函数 audio_ioctl ()*/
case SND_DEV_DSP:
case SND_DEV_DSP16:
case SND_DEV_AUDIO:
    return audio_ioctl(dev, file, cmd, p);
    break;
/*设备接口类型为 MIDI 时, 调用函数 MIDIBuf_ioctl ()*/
case SND_DEV_MIDIN:
    return MIDIBuf_ioctl(dev, file, cmd, p);
    break;
}
return -EINVAL;
}

```

## 10.3 Linux 音频设备驱动——ALSA 驱动框架

ALSA 包含了许多的声卡驱动程序，提供 libasound 的 API 库。libasound 提供了方便的高级编程接口，应用程序中使用 libasound 而不是内核中的 ALSA 接口。同时，libasound 提供一个设备逻辑命名功能，因此开发者不必知道类似设备文件这样的底层接口。相反，OSS 驱动是在内核系统调用级上编程，要求应用程序开发者提供设备文件名并且利用 ioctl 来实现相应的功能。为了兼容 OSS，ALSA 框架使用内核模块来模拟 OSS，在 OSS 基础上开发的应用程序不需要任何改动就可以在 ALSA 上运行。另外，libaoss 库也能够模拟 OSS，而它不需要内核模块。ALSA 包含插件功能，使用插件可以扩展新的声卡驱动，包括完全使用软件实现的虚拟声卡。ALSA 提供一系列基于命令行的工具集 alsa-utils，比如 amixer（混音器命令行控制）、aplay（音频文件命令行播放器）、arecord（命令行音频文件录制）等工具。

ALSA 提供给用户的接口主要有：

- 信息接口（Information Interface, /proc/asound）。
- 控制接口（Control Interface, /dev/snd/controlCX）提供管理声卡注册和请求可用设备的通用功能。
- PCM 接口（PCM Interface, /dev/snd/pcmCXDX）管理数字音频回放（playback）和录音（capture）的接口。
- Raw MIDI 接口（Raw MIDI Interface, /dev/snd/midiCXDX）支持 MIDI（Musical Instrument Digital Interface），标准的电子乐器。这些 API 提供对声卡上 MIDI 总线的访问。
- 定时器接口（Timer Interface, /dev/snd/timer）为同步音频事件提供对声卡上时间处理硬件的访问。
- 音序器接口（Sequencer Interface, /dev/snd/seq）。
- 混音器接口（Mixer Interface）。

### 10.3.1 card 和组件

每个声卡必须创建一个 card 实例，下面通过驱动代码介绍 ALSA 声卡驱动是如何管理 card 和组件的。

#### 1. 创建card函数snd\_card\_new()

函数 snd\_card\_new() 用于创建和初始化 snd\_card 结构体。idx 是卡的索引号，xid 是标识字符串，module 是顶层模块，extra\_size 额外分配数据的大小。

```
struct snd_card *snd_card_new(int idx, const char *xid, struct module *module,
int extra_size)
{
    struct snd_card *card;
    int err, idx2;
```

```

if (extra_size < 0)
    extra_size = 0;
/*分配空间同时初始化所分配的空间*/
card = kzalloc(sizeof(*card) + extra_size, GFP_KERNEL);
if (card == NULL)
    return NULL;
if (xid) {
    if (!snd_info_check_reserved_words(xid))
        goto __error;
    strlcpy(card->id, xid, sizeof(card->id));
}
/*锁住互斥量*/
mutex_lock(&snd_card_mutex);
if (idx < 0) {
    for (idx2 = 0; idx2 < SNDRV_CARDS; idx2++)
        /*idx == -1 == 0xffff means: take any free slot*/
        if (~snd_cards_lock & idx & 1<<idx2) {
            if (module_slot_match(module, idx2)) {
                idx = idx2;
                break;
            }
        }
}
if (idx < 0) {
    for (idx2 = 0; idx2 < SNDRV_CARDS; idx2++)
        /*idx == -1 == 0xffff means: take any free slot*/
        if (~snd_cards_lock & idx & 1<<idx2) {
            if (!slots[idx2] || !*slots[idx2]) {
                idx = idx2;
                break;
            }
        }
}
snd_cards_lock |= 1 << idx;      /*lock it*/
if (idx >= snd_ecards_limit)
    snd_ecards_limit = idx + 1; /*increase the limit*/
/*解锁互斥量*/
mutex_unlock(&snd_card_mutex);
/*对 card 各个字段进行初始化和赋值*/
card->number = idx;
card->module = module;
INIT_LIST_HEAD(&card->devices);
init_rwsem(&card->controls_rwsem);
rwlock_init(&card->ctl_files_rwlock);
INIT_LIST_HEAD(&card->controls);
INIT_LIST_HEAD(&card->ctl_files);
spin_lock_init(&card->files_lock);
init_waitqueue_head(&card->shutdown_sleep);
return card;                      /*省略了错误处理部分*/
}

```

## 2. 创建组件函数snd\_device\_new()

在 card 被创建后，创建设备（组件）关联到该 card。参数 card 为 snd\_card\_new() 创建的 card，参数 type 为设备类型，包括 SNDRV\_DEV\_TOPLEVEL、SNDRV\_DEV\_CONTROL、SNDRV\_DEV\_LOWLEVEL\_PRE、SNDRV\_DEV\_LOWLEVEL\_NORMAL、SNDRV\_DEV\_

PCM、SNDRV\_DEV\_RAWMIDI、SNDRV\_DEV\_TIMER、SNDRV\_DEV\_SEQUENCER、SNDRV\_DEV\_CODEC 等（具体的定义在 include/sound/core.h 中），device\_data 为设备数据指针，ops 为函数集的指针。

```
int snd_device_new(struct snd_card *card, snd_device_type_t type,
                   void *device_data, struct snd_device_ops *ops)
{
    struct snd_device *dev;

    if (snd_BUG_ON(!card || !device_data || !ops))
        return -ENXIO;
    /*给设备分配空间并初始化*/
    dev = kzalloc(sizeof(*dev), GFP_KERNEL);
    if (dev == NULL) {
        snd_printk(KERN_ERR "Cannot allocate device\n");
        return -ENOMEM;
    }
    /*给设备的各个字段赋值*/
    dev->card = card;
    dev->type = type;
    dev->state = SNDRV_DEV_BUILD;
    dev->device_data = device_data;
    dev->ops = ops;
    /*将设备列表插入到 card 的设备字段后*/
    list_add(&dev->list, &card->devices); /*add to the head of list*/
    return 0;
}
```

### 3. 释放组件函数 snd\_device\_free ()

函数 snd\_device\_free() 释放与 card 关联的设备，并且对应设备的数据为 device\_data。

```
int snd_device_free(struct snd_card *card, void *device_data)
{
    struct snd_device *dev;

    if (snd_BUG_ON(!card || !device_data))
        return -ENXIO;
    /*根据 device_data 找到要释放的设备*/
    list_for_each_entry(dev, &card->devices, list) {
        if (dev->device_data != device_data)
            continue;
        /*该设备从链表中断开*/
        list_del(&dev->list);
        /*判断断开是否成功*/
        if (dev->state == SNDRV_DEV_REGISTERED && dev->ops->dev_disconnect)
            if (dev->ops->dev_disconnect(dev))
                snd_printk(KERN_ERR "device disconnect failure\n");
        /*判断释放是否成功*/
        if (dev->ops->dev_free) {
            if (dev->ops->dev_free(dev))
                snd_printk(KERN_ERR "device free failure\n");
        }
        /*释放设备占用的内存*/
        kfree(dev);
    }
```

```

        return 0;
    }
    snd_printd("device free %p (from %pF), not found\n", device_data,
               __builtin_return_address(0));
    return -ENXIO;
}

```

#### 4. 释放card函数snd\_card\_free()

该函数释放 snd\_card\_new() 创建和初始化的 snd\_card 结构体。

```

int snd_card_free(struct snd_card *card)
{
    /*释放操作，并通知连接在 card 上的所有设备*/
    int ret = snd_card_disconnect(card);
    if (ret)
        return ret;

    /*所有的设备准备好释放操作*/
    wait_event(card->shutdown_sleep, card->files == NULL);
    /*释放 card*/
    snd_card_do_free(card);
    return 0;
}

```

#### 5. 断开所有的ALSA API函数snd\_card\_disconnect()

在系统中 file\_operations 中对应系统的操作 llseek 与 card 的操作 snd\_disconnect\_llseek、系统的 read 与 snd\_disconnect\_read、系统的 write 与 snd\_disconnect\_write 等。函数 snd\_card\_disconnect() 的作用是将这些对应关系断开，并通知该信息到所有连接在 card 上的设备。

```

int snd_card_disconnect(struct snd_card *card)
{
    /*第一阶段：使 ALSA API 操作失效*/
    mutex_lock(&snd_card_mutex);
    snd_cards[card->number] = NULL;
    snd_cards_lock &= ~(1 << card->number);
    mutex_unlock(&snd_card_mutex);
    /*第二阶段：用专门的哑操作代替 file->f_op*/
    spin_lock(&card->files_lock);
    mfile = card->files;
    while (mfile) {
        file = mfile->file;
        /*it's critical part, use endless loop*/
        /*we have no room to fail*/
        mfile->disconnected_f_op = mfile->file->f_op;
        spin_lock(&shutdown_lock);
        list_add(&mfile->shutdown_list, &shutdown_files);
        spin_unlock(&shutdown_lock);
        mfile->file->f_op = &snd_shutdown_f_ops;
        fops_get(mfile->file->f_op);
        mfile = mfile->next;
    }
}

```

```

spin_unlock(&card->files_lock);
/*第三阶段：通知所有的连接设备断开信息*/
snd_device_disconnect_all(card);
snd_info_card_disconnect(card);
return 0;
}

```

### 10.3.2 PCM 设备

每个声卡设备最多有 4 个 PCM 实例，而每个 PCM 实例对应一个设备文件。每个 PCM 实例由 PCM 播放流和 PCM 录音流组成，每个 PCM 流由一个或多个 PCM 子流组成。

#### 1. 函数 snd\_pcm\_new() 用来创建 PCM 实例

函数 snd\_pcm\_new() 中的各个参数的作用分别如下：参数 card 是与 PCM 对应的声卡；参数 id 为标识字符串；参数 device 是 PCM 设备索引，用来表示第几个 PCM 设备；参数 playback\_count 为播放的子流数；参数 capture\_count 为录音的子流数；参数 rpcm 用来返回构造的 PCM 实例。

```

int snd_pcm_new(struct snd_card *card, char *id, int device,
                int playback_count, int capture_count, struct snd_pcm ** rpcm)
{
    struct snd_pcm *pcm;
    int err;
    static struct snd_device_ops ops = {
        .dev_free = snd_pcm_dev_free,
        .dev_register = snd_pcm_dev_register,
        .dev_disconnect = snd_pcm_dev_disconnect,
    };
    /*为构造的 PCM 实例分配空间并进行初始化*/
    pcm = kzalloc(sizeof(*pcm), GFP_KERNEL);
    if (pcm == NULL) {
        snd_printk(KERN_ERR "Cannot allocate PCM\n");
        return -ENOMEM;
    }
    /*指定 PCM 的声卡与 PCM 设备索引*/
    pcm->card = card;
    pcm->device = device;
    if (id)
        strlcpy(pcm->id, id, sizeof(pcm->id));
    /*构造 PCM 播放流*/
    if ((err = snd_pcm_new_stream(pcm, SNDDRV_PCM_STREAM_PLAYBACK,
                                  playback_count)) < 0) {
        snd_pcm_free(pcm);
        return err;
    }
    /*构造 PCM 录音流*/
    if ((err = snd_pcm_new_stream(pcm, SNDDRV_PCM_STREAM_CAPTURE,
                                  capture_count)) < 0) {
        snd_pcm_free(pcm);
        return err;
    }
    mutex_init(&pcm->open_mutex);
    init_waitqueue_head(&pcm->open_wait);
}

```

```

/*创建设备类型为 SNDDEV_DEV_PCM 的设备，并指定该设备的声卡为 card*/
if ((err = snd_device_new(card, SNDDEV_DEV_PCM, pcm, &ops)) < 0) {
    snd_pcm_free(pcm);
    return err;
}
if (rpcm)
    *rpcm = pcm;
return 0;
}

```

## 2. 函数snd\_pcm\_free()释放PCM实例

函数 snd\_pcm\_free()用来释放 snd\_pcm\_new()创建的 PCM 实例。释放包括释放播放流和录音流。

```

static int snd_pcm_free(struct snd_pcm *pcm)
{
    struct snd_pcm_notify *notify;

    list_for_each_entry(notify, &snd_pcm_notify_list, list) {
        notify->n_unregister(pcm);
    }
    if (pcm->private_free)
        pcm->private_free(pcm);
    /*释放分配的缓冲区*/
    snd_pcm_lib_preallocate_free_for_all(pcm);
    /*释放播放流*/
    snd_pcm_free_stream(&pcm->streams[SNDDEV_PCM_STREAM_PLAYBACK]);
    /*释放录音流*/
    snd_pcm_free_stream(&pcm->streams[SNDDEV_PCM_STREAM_CAPTURE]);
    kfree(pcm);
    return 0;
}

```

## 3. 设置PCM操作

函数 snd\_pcm\_set\_ops()设置 PCM 打开子流、关闭子流等操作。

```

void snd_pcm_set_ops(struct snd_pcm *pcm, int direction, struct snd_pcm_ops
*ops)
{
    struct snd_pcm_str *stream = &pcm->streams[direction];
    struct snd_pcm_substream *substream;
    /*设置子流操作*/
    for (substream = stream->substream; substream != NULL; substream =
substream->next)
        substream->ops = ops;
}

```

snd\_pcm\_ops 结构体定义如下：

```

struct snd_pcm_ops {
    int (*open)(struct snd_pcm_substream *substream);           //打开子流
    int (*close)(struct snd_pcm_substream *substream);          //关闭子流
    int (*ioctl)(struct snd_pcm_substream *substream,
                 unsigned int cmd, void *arg);                      //I/O 控制
}

```

```

int (*hw_params)(struct snd_pcm_substream *substream,
                  struct snd_pcm_hw_params *params);           //硬件参数
int (*hw_free)(struct snd_pcm_substream *substream); //释放资源
int (*prepare)(struct snd_pcm_substream *substream); //准备
int (*trigger)(struct snd_pcm_substream *substream, int cmd);
                                         //在 PCM 开始、停止、暂停时调用
snd_pcm_uframes_t (*pointer)(struct snd_pcm_substream *substream);
                                         //当前缓冲区的硬件位置
int (*copy)(struct snd_pcm_substream *substream, int channel,
            snd_pcm_uframes_t pos,
            void __user *buf, snd_pcm_uframes_t count); //复制缓冲区
int (*silence)(struct snd_pcm_substream *substream, int channel,
                snd_pcm_uframes_t pos, snd_pcm_uframes_t count); //静音
struct page *(*page)(struct snd_pcm_substream *substream,
                     unsigned long offset);                      //cache 操作
int (*mmap)(struct snd_pcm_substream *substream, struct vm_area_struct
            *vma);                                         //内存映射
int (*ack)(struct snd_pcm_substream *substream);      //应答
};

}

```

#### 4. PCM运行时结构体

打开 PCM 子流后，分配 PCM 运行时实例给该子流。运行时实例的结构体定义如下：

```

struct snd_pcm_runtime {
    /*子流的状态*/
    struct snd_pcm_substream *trigger_master;
    struct timespec trigger_tstamp;           //触发时间戳
    int overrange;
    snd_pcm_uframes_t avail_max;
    snd_pcm_uframes_t hw_ptr_base;           //缓冲区复位时位置
    snd_pcm_uframes_t hw_ptr_interrupt;       //缓冲区中断时位置

    /*子流硬件参数*/
    snd_pcm_access_t access;                 //访问模式
    snd_pcm_format_t format;                //SNDDRV_PCM_FORMAT_*
    snd_pcm_subformat_t subformat;          //子格式
    unsigned int rate;                      //速率单位 Hz
    unsigned int channels;                  //通道
    snd_pcm_uframes_t period_size;          //周期大小
    unsigned int periods;                   //周期数
    snd_pcm_uframes_t buffer_size;          //缓冲区大小
    snd_pcm_uframes_t min_align;            //对齐的最小块大小
    size_t byte_align;                     //字节对齐
    unsigned int frame_bits;               //每帧的比特数
    unsigned int sample_bits;              //采样的比特数（编码所使用的比特数）
    unsigned int info;
    unsigned int rate_num;
    unsigned int rate_den;

    /*软件参数*/
    int tstamp_mode;                      //更新内存映射时间戳
    unsigned int period_step;
    snd_pcm_uframes_t start_threshold;
    snd_pcm_uframes_t stop_threshold;
}

```

```

    snd_pcm_uframes_t silence_threshold;      //Silence 填充阈值
    snd_pcm_uframes_t silence_size;           //Silence 填充大小
    snd_pcm_uframes_t boundary;               //封装指针的指针
    snd_pcm_uframes_t silence_start;          //silence 区域开始指针
    snd_pcm_uframes_t silence_filled;         //silence 被填充的大小
    union snd_pcm_sync_id sync;              //硬件同步 ID
    /*内存映射*/
    struct snd_pcm_mmap_status *status;       //内存映射状态
    struct snd_pcm_mmap_control *control;     //内存映射控制
    /*锁和调度*/
    wait_queue_head_t sleep;
    struct fasync_struct *fasync;
    /*私有段*/
    void *private_data;
    void (*private_free)(struct snd_pcm_runtime *runtime);
    /*硬件描述*/
    struct snd_pcm_hardware hw;
    struct snd_pcm_hw_constraints hw_constraints;
    /*中断回调函数*/
    void (*transfer_ack_begin)(struct snd_pcm_substream *substream);
    void (*transfer_ack_end)(struct snd_pcm_substream *substream);
    /*定时器*/
    unsigned int timer_resolution;            //定时器精度
    int tstamp_type;                        //时间戳类型
    /*DMA 缓冲区信息*/
    unsigned char *dma_area;                //DMA 区域*
    dma_addr_t dma_addr;                   //总线物理地址*
    size_t dma_bytes;                      //DMA 区域大小*
    struct snd_dma_buffer *dma_buffer_p;    //分配的缓冲区
#ifndef CONFIG_SND_PCM_OSS || defined(CONFIG_SND_PCM_OSS_MODULE)
    /*OSS 运行时结构体*/
    struct snd_pcm_oss_runtime oss;
#endif
};

```

### 10.3.3 控制接口

控制接口（control）的主要作用是混音器（mixer），所有 mixer 元素都是基于 control API 实现的。下面将简单介绍 control 的几个重要函数。

#### 1. snd\_ctl\_open()

函数 snd\_ctl\_open() 用于打开设备文件操作。

```

static int snd_ctl_open(struct inode *inode, struct file *file)
{
    unsigned long flags;
    struct snd_card *card;
    struct snd_ctl_file *ctl;
    /*从注册的设备获得用户数据*/
    card = snd_lookup_minor_data(iminor(inode), SNDDRV_DEVICE_TYPE_
CONTROL);

```

```

/*增加文件到 card 的文件链表*/
snd_card_file_add(card, file);
/*为 ctl 分配空间并初始化*/
ctl = kzalloc(sizeof(*ctl), GFP_KERNEL);
/*初始化 ctl 事件链表*/
INIT_LIST_HEAD(&ctl->events);
/*对 ctl 的各个字段进行初始化或者赋值*/
init_waitqueue_head(&ctl->change_sleep);
spin_lock_init(&ctl->read_lock);
ctl->card = card;
ctl->prefer_pcm_subdevice = -1;
ctl->prefer_rawmidi_subdevice = -1;
ctl->pid = current->pid;
file->private_data = ctl;
//对链表操作前进行锁保护
write_lock_irqsave(&card->ctl_files_rwlock, flags);
list_add_tail(&ctl->list, &card->ctl_files);
//完成操作进行解锁
write_unlock_irqrestore(&card->ctl_files_rwlock, flags);
return 0;
}

```

## 2. 创建控制实例snd\_ctl\_new()

函数 snd\_ctl\_new() 创建控制实例，参数 control 为控制模板，参数 access 为默认的控制访问。

```

static struct snd_kcontrol *snd_ctl_new(struct snd_kcontrol *control,
                                         unsigned int access)
{
    struct snd_kcontrol *kctl;
    unsigned int idx;

    if (snd_BUG_ON(!control || !control->count))
        return NULL;
    //分配空间并初始化
    kctl = kzalloc(sizeof(*kctl) + sizeof(struct snd_kcontrol_volatile) *
control->count, GFP_KERNEL);
    if (kctl == NULL) {
        snd_printk(KERN_ERR "Cannot allocate control instance\n");
        return NULL;
    }
    //复制控制模板
    *kctl = *control;
    for (idx = 0; idx < kctl->count; idx++)
        kctl->vd[idx].access = access;
    return kctl;
}

```

## 3. 移植控制函数snd\_ctl\_remove()

函数 snd\_ctl\_remove() 从 card 移除 control，并且释放它占用的空间。

```

int snd_ctl_remove(struct snd_card *card, struct snd_kcontrol *kcontrol)
{
    struct snd_ctl_elem_id id;
    unsigned int idx;

```

```

if (snd_BUG_ON(!card || !kcontrol))
    return -EINVAL;
/*移除控制链表*/
list_del(&kcontrol->list);
card->controls_count -= kcontrol->count;
id = kcontrol->id;
for (idx = 0; idx < kcontrol->count; idx++, id.index++, id.numid++)
    snd_ctl_notify(card, SNDDRV_CTL_EVENT_MASK_REMOVE, &id);
/*释放空间*/
snd_ctl_free_one(kcontrol);
return 0;
}

```

### 10.3.4 AC97 API 音频接口

ALSA AC97 已经有各种接口，驱动开发人员通过编写少量的控制函数，就可以开发自己的驱动。

#### 1. 函数snd\_ac97\_bus()创建AC97 bus总线组件

函数 snd\_ac97\_bus()创建 AC97 bus 总线组件及 AC97 的操作。

```

int snd_ac97_bus(struct snd_card *card, int num, struct snd_ac97_bus_ops
*ops,
                  void *private_data, struct snd_ac97_bus **rbus)
{
    int err;
    struct snd_ac97_bus *bus;
    static struct snd_device_ops dev_ops = {
        .dev_free = snd_ac97_bus_dev_free,
    };
    /*分配空间并初始化*/
    bus = kzalloc(sizeof(*bus), GFP_KERNEL);
    /*bus 各字段赋值*/
    bus->card = card;
    bus->num = num;
    bus->ops = ops;
    bus->private_data = private_data;
    bus->clock = 48000;
    spin_lock_init(&bus->bus_lock);
    /*总线初始化*/
    snd_ac97_bus_proc_init(bus);
    /*构造 BUS 类型的设备，并指定声卡为 card*/
    if ((err = snd_device_new(card, SNDDRV_DEV_BUS, bus, &dev_ops)) < 0) {
        snd_ac97_bus_free(bus);
        return err;
    }
    if (rbus)
        *rbus = bus;
    return 0;
}

```

## 2. 创建Codec97 组件函数snd\_ac97\_mixer()

在创建完 AC97 bus 后，调用该函数创建依附于该 bus 的 Codec97 组件。下面列出该函数的主要代码：

```

int snd_ac97_mixer(struct snd_ac97_bus *bus, struct snd_ac97_template
*template, struct snd_ac97 **rac97)
{
    int err;
    struct snd_ac97 *ac97;
    struct snd_card *card;
    char name[64];
    unsigned long end_time;
    unsigned int reg;
    const struct ac97_codec_id *pid;
    static struct snd_device_ops ops = {
        .dev_free = snd_ac97_dev_free,
        .dev_register = snd_ac97_dev_register,
        .dev_disconnect = snd_ac97_dev_disconnect,
    };

    if (rac97)
        *rac97 = NULL;
    /*设置 bus 的 card 字段为该声卡*/
    card = bus->card;
    /*为创建的组件 AC97 分配空间并初始化*/
    ac97 = kzalloc(sizeof(*AC97), GFP_KERNEL);
    /*初始化或者设置所创建的组件 AC97 的各个字段*/
    ac97->private_data = template->private_data;
    ac97->private_free = template->private_free;
    ac97->bus = bus;
    ac97->pci = template->pci;
    ac97->num = template->num;
    ac97->addr = template->addr;
    ac97->scaps = template->scaps;
    ac97->res_table = template->res_table;
    bus->codec[ac97->num] = ac97;
    /*信号量初始化*/
    mutex_init(&ac97->reg_mutex);
    mutex_init(&ac97->page_mutex);
    /*bus 复位*/
    if (bus->ops->reset) {
        bus->ops->reset(ac97);
        goto __access_ok;
    }
    /*读取产品的 ID, AC97_VENDOR_ID1 放置在高两字节, AC97_VENDOR_ID2 放置在低两字节*/
    ac97->id = snd_ac97_read(ac97, AC97_VENDOR_ID1) << 16;
    ac97->id |= snd_ac97_read(ac97, AC97_VENDOR_ID2);
    if (ac97->id && ac97->id != (unsigned int)-1) {
        pid = look_for_codec_id(snd_ac97_codec_ids, ac97->id);
        if (pid && (pid->flags & AC97_DEFAULT_POWER_OFF))
            goto __access_ok;
    }
    /*恢复为默认*/
    if (!(ac97->scaps & AC97_SCAP_SKIP_AUDIO))
        snd_ac97_write(ac97, AC97_RESET, 0);
}

```

```

if (!(ac97->scaps & AC97_SCAP_SKIP_MODEM))
    snd_ac97_write(ac97, AC97_EXTENDED_MID, 0);
if (bus->ops->wait)
    bus->ops->wait(ac97);
else {
    /*延迟 50 微秒*/
    udelay(50);
    /*等待直到寄存器复位至可以访问*/
    if (ac97->scaps & AC97_SCAP_SKIP_AUDIO)
        err = ac97_reset_wait(ac97, msecs_to_jiffies(500), 1);
    else {
        err = ac97_reset_wait(ac97, msecs_to_jiffies(500), 0);
        if (err < 0)
            err = ac97_reset_wait(ac97, msecs_to_jiffies(500), 1);
    }
    if (err < 0) {
        snd_printk(KERN_WARNING "AC'97 %d does not respond - RESET\n",
                   ac97->num);
        /*proceed anyway - it's often non-critical*/
    }
}
__access_ok:
/*读取产品 ID 信息*/
ac97->id = snd_ac97_read(ac97, AC97_VENDOR_ID1) << 16;
ac97->id |= snd_ac97_read(ac97, AC97_VENDOR_ID2);
if (!(ac97->scaps & AC97_SCAP_DETECT_BY_VENDOR) &&
    (ac97->id == 0x00000000 || ac97->id == 0xffffffff)) {
    snd_printk(KERN_ERR "AC'97 %d access is not valid [0x%x], removing
mixer.\n", ac97->num, ac97->id);
    snd_ac97_free(ac97); //若为无效组件，则释放该组件
    return -EIO;
}
pid = look_for_codec_id(snd_ac97_codec_ids, ac97->id);
if (pid)
    ac97->flags |= pid->flags;

/*为接口 AC'97 测试该组件*/
if (!(ac97->scaps & AC97_SCAP_SKIP_AUDIO) && !(ac97->scaps & AC97_SCAP_AUDIO)) {
    /*测试是否可以向 AC97 的 AC97_REC_GAIN 寄存器写值 0x8a06*/
    snd_ac97_write_cache(ac97, AC97_REC_GAIN, 0x8a06);
    /*测试读取 AC97 的 AC97_REC_GAIN 寄存器的值与 0xffff 做与运算的结果是否为
0xa06*/
    if (((err = snd_ac97_read(ac97, AC97_REC_GAIN)) & 0xffff) == 0xa06)
        ac97->scaps |= AC97_SCAP_AUDIO;
}
/*读取 AC97_RESET 和 AC97_EXTENDED_ID 的值分别赋给 caps 和 ext_id 字段*/
if (ac97->scaps & AC97_SCAP_AUDIO) {
    ac97->caps = snd_ac97_read(ac97, AC97_RESET);
    ac97->ext_id = snd_ac97_read(ac97, AC97_EXTENDED_ID);
    if (ac97->ext_id == 0xffff) /*无效组件*/
        ac97->ext_id = 0;
}
/*测试 MC'97，读取 AC97_EXTENDED_MID 寄存器并测试读取的有效性*/
if (!(ac97->scaps & AC97_SCAP_SKIP_MODEM) && !(ac97->scaps & AC97_SCAP_MODEM)) {
    ac97->ext_mid = snd_ac97_read(ac97, AC97_EXTENDED_MID);
    if (ac97->ext_mid == 0xffff) /*无效组件*/
        ac97->ext_mid = 0;
}

```

```

    if (ac97->ext_mid & 1)
        ac97->scaps |= AC97_SCAP_MODEM;
}
/*检查AC97的audio与modem*/
if (!ac97_is_audio(ac97) && !ac97_is_modem(ac97)) {
    if (!(ac97->scaps & (AC97_SCAP_SKIP_AUDIO|AC97_SCAP_SKIP_MODEM)))
        snd_printk(KERN_ERR "AC'97 %d access error (not audio or modem
                           codec)\n", ac97->num);
    snd_ac97_free(ac97); //如果audio与modem不正确，则释放创建的AC97组件
    return -EACCES;
}
/*/*设置mixername名字*/
if (ac97_is_audio(ac97)) {
    /*获得组件ID字符串*/
    char comp[16];
    if (card->mixername[0] == '\0') {
        strcpy(card->mixername, name);
    } else {
        if (strlen(card->mixername) + 1 + strlen(name) + 1 <= sizeof
            (card->mixername)) {
            strcat(card->mixername, ",");
            strcat(card->mixername, name);
        }
    }
    sprintf(comp, "AC97a:%08x", ac97->id);
    /*添加组件ID字符串comp到所给的链表*/
    if ((err = snd_component_add(card, comp)) < 0) {
        snd_ac97_free(ac97); //如添加失败，则释放该组件AC97
        return err;
    }
    /*构造控制器mixer，该构造过程比较复杂，构造内容比较多，包括master controls、
     center controls、LFE controls、surround controls、headphone
     controls、mono controls（单声道控制器）、tone controls（音质控制器）、
     PC Speaker controls（PC扬声器）、Phone controls、MIC controls（麦克风）、
     Line controls、CD controls、PCM controls等。创建失败则释放
     组件AC97。*/
    if (snd_ac97_mixer_build(ac97) < 0) {
        snd_ac97_free(ac97);
        return -ENOMEM;
    }
}
if (ac97_is_modem(ac97)) {
    /*获得组件ID字符串*/
    char comp[16];
    if (card->mixername[0] == '\0') {
        strcpy(card->mixername, name);
    } else {
        if (strlen(card->mixername) + 1 + strlen(name) + 1 <= sizeof
            (card->mixername)) {
            strcat(card->mixername, ",");
            strcat(card->mixername, name);
        }
    }
    sprintf(comp, "AC97m:%08x", ac97->id);
    /*添加组件ID字符串comp到所给的链表*/
    if ((err = snd_component_add(card, comp)) < 0) {
        snd_ac97_free(ac97);
        return err;
    }
}

```

```

    }
    /*构造 modem , 如构造失败则释放 AC97*/
    if (snd_ac97_modem_build(card, ac97) < 0) {
        snd_ac97_free(ac97);
        return -ENOMEM;
    }
}
/*更新 power 寄存器*/
if (ac97_is_audio(ac97))
    update_power_regs(ac97);
snd_ac97_proc_init(ac97);
/*构造设备类型为 SNDDRV_DEV_CODEC 的设备，并设置设备的 card 为该声卡，如果失败则
释放 AC97*/
if ((err = snd_device_new(card, SNDDRV_DEV_CODEC, ac97, &ops)) < 0) {
    snd_ac97_free(ac97);
    return err;
}
*rac97 = ac97;
return 0;
}

```

### 3. 释放bus函数snd\_ac97\_bus\_dev\_free()

函数 snd\_ac97\_bus\_dev\_free() 释放函数 snd\_ac97\_bus()，创建 AC97 bus 总线。

```

static int snd_ac97_bus_dev_free(struct snd_device *device)
{
    struct snd_ac97_bus *bus = device->device_data;
    return snd_ac97_bus_free(bus);
}

```

## 10.4 音频设备应用程序编写

10.3 节中介绍了 OSS 驱动和 ALSA 驱动，本节将主要介绍如何编写这两种驱动程序的应用程序。对于 OSS 驱动，给出了 DSP 接口编程和 MIXER 接口编程实例。

### 10.4.1 DSP 接口编程

OSS 驱动框架提供了音频编程的 3 种设备，分别是 /dev/dsp、/dev/dspW 和 /dev/audio。用户可以直接使用命令播放和录音，命令 cat /dev/dsp > test 可用来录音，录音的结果放在 test 文件中；命令 cat test>/dev/dsp 播放声音文件 test。OSS 应用程序主要包括打开设备、录音、播放、设置参数等部分，下面将分别介绍。

#### 1. 包含相关头文件

在使用 OSS 驱动编程时，应该包含相应的头文件，需要包含的头文件如下：

```
#include <iomanip.h>
#include <unistd.h>
#include <fcntl.h>
```

```
#include <sys/soundcard.h>
```

## 2. 打开设备文件

在对设备进行操作前，打开设备。打开的设备包括/dev/dsp、dev/dspW 和/dev/audio，如下面的设备名 DEVICE\_NAME 可以设为 dev/dsp。打开设备的模式包括 O\_RDONLY、\_WRONLY 和 O\_RDWR，分别表示只读、只写和读写。

```
#define BUF_SIZE 4096
int fd;
unsigned char buf[BUF_SIZE];
if ((fd = open(DEVICE_NAME, O_RDONLY, 0)) == -1) {
    /*如果打开设备失败则退出程序*/
    perror(DEVICE_NAME);
    exit(1);
}
```

## 3. 录音

read()函数中参数 count 为录音数据的字节个数（建议为 2 的指数，如 512），但不能大于 buf 的大小。从读字节的个数可以精确地算时间，例如采样频率为 8kHz，采样精度为 8bit 的立体声的速率为  $8000 \times 1 \times 2 = 16\text{KBytes/second}$ ，这是计算停止录音时间的唯一方法。

```
int len;
if ((len = read(fd, buf, count)) == -1) {
    perror("audio read");
    exit(1);
}
```

## 4. 播放

播放实际上和录音很类似，相应的 buf 中为音频数据，sizeof(buf) 为数据的长度。注意，用户始终要读/写一个完整的采样。例如一个采样精度 16bit 的立体声模式下，每个采样有 4 个字节，所以应用程序每次必须读/写 4 的倍数个字节。

```
if (write (fd, buf, sizeof (buf)) != sizeof (buf))
{
    perror ("Audio write");
    exit (-1);
}
```

 注意：由于 OSS 是一个跨平台的音频接口，所以用户在编程的时候，要考虑到可移植性的问题，其中一个重要的方面是读/写时的字节顺序。

## 5. 设置参数

参数设置包括设置缓冲区大小、设置采用格式、设置通道数目和设置采样速率等。

### (1) 设置缓冲区大小

Linux 内核中的声卡驱动程序专门维护了一个缓冲区，其大小影响到放音和录音时的效果，使用 ioctl 系统调用可以对它的大小进行调整。调节驱动程序中缓冲区大小的操作不是必须的，如果没有特殊的要求，缓冲区大小通常采用默认设置。并且缓冲区大小的设置

通常应紧跟在打开设备文件之后，因为对声卡的其他操作有可能会导致驱动程序无法再修改其缓冲区的大小。设置声卡驱动程序中内核缓冲区的大小方法如下：

```
int size = 0xnnnnnsss;
int result = ioctl(handle, SNDCTL_DSP_SETFRAGMENT, &size);
if (result == -1) {
    perror("ioctl buffer size");
    return -1;
}
```

在设置缓冲区大小时，参数 `size` 由两部分组成，其低 16 位标明缓冲区的尺寸，相应的计算公式为  $size = 2^{ssss}$ ，即若参数 `size` 低 16 位的值为 16，那么相应的缓冲区的大小会被设置为  $2^{16}=65536$  字节。参数 `size` 的高 16 位则用来标明分片（fragment）的最大序号，它的取值范围从 2 到 0x7FFF，其中 0x7FFF 表示没有任何限制。

#### (2) 设置采样格式

在音频编程时需要考虑采样格式和采样频率，在头文件 `soundcard.h` 中定义声卡支持的所有采样格式，而通过系统调用 `ioctl` 则可以很方便地更改当前所使用的采样格式。下面的代码示范对声卡的采样格式进行设置的方法：

```
int format;
format = AFMT_S16_NE; /*Native 16 bits*/
if (ioctl(fd, SNDCTL_DSP_SETFMT, &format) == -1) {
    /*fatal error*/
    perror("SNDCTL_DSP_SETFMT");
    exit(1);
}
if (format != AFMT_S16_NE) {
    /*本设备不支持选择的采样格式*/
}
/*在设置采样格式之前，可以先测试设备能够支持哪些采样格式，方法如下：*/
int mask;
if (ioctl(fd, SNDCTL_DSP_GETFMTS, &mask) == -1) {
    /*Handle fatal error ...*/
}
if (mask & AFMT_MPEG) {
    /*本设备支持 MPEG 采样格式*/
}
```

#### (3) 设置通道数目

```
int channels = 2;           /*1=mono, 2=stereo*/
if (ioctl(fd, SNDCTL_DSP_CHANNELS, &channels) == -1) {
    /*Fatal error*/
    perror("SNDCTL_DSP_CHANNELS");
    exit(1);
}
if (channels != 2) {
    /*本设备不支持立体声模式*/
}
```

#### (4) 设置采样速率

声卡采样频率的设置比较简单，调用 `ioctl` 时设置第 2 个参数为 `SNDCTL_DSP_SPEED`，同时在第 3 个参数中指定采样频率的数值即可。对于大多数声卡来说，其支持的采样频率范围一般为 5kHz~44.1kHz 或者 48kHz，但并不意味着该范围内的所有频率都会被硬件支

持。在 Linux 下进行音频编程时最常用到的几种采样频率是 11025Hz、16000Hz、22050Hz、32000Hz 和 44100Hz。下面的代码示范对声卡的采样频率进行设置：

```
int rate = 11025;
if (ioctl(fd, SNDCTL_DSP_SPEED, & rate)==-1) {
    /*Fatal error*/
    perror("SNDCTL_DSP_SPEED");
    exit(Error code);
}
```

### 10.4.2 MIXER 接口编程

声卡上的混音器由多个混音通道组成，它们可以通过驱动程序提供的设备文件 /dev/mixer 进行编程。对混音器的操作是通过系统调用 ioctl 来完成，所有的混音器控制命令都以 SOUND\_MIXER 或者 MIXER 开头。表 10.1 中列出了常用的混音器控制命令。

表 10.1 常用的混音器控制命令

命 令	作 用
SOUND_MIXER_VOLUME	调节主音量
SOUND_MIXER_BASS	低音控制
SOUND_MIXER_TREBLE	高音控制
SOUND_MIXER_SYNTH	FM 合成器
SOUND_MIXER_PCM	主 D/A 转换器
SOUND_MIXER_SPEAKER	PC 喇叭
SOUND_MIXER_LINE	音频线输入
SOUND_MIXER_MIC	麦克风输入
SOUND_MIXER_CD	CD 输入
SOUND_MIXER_IMIX	回放音量
SOUND_MIXER_ALTPCM	从 D/A
SOUND_MIXER_RECLEV	录音音量
SOUND_MIXER_IGAIN	输入增益
SOUND_MIXER_OGAIN	输出增益
SOUND_MIXER_LINE1	声卡的第 1 输入
SOUND_MIXER_LINE2	声卡的第 2 输入
SOUND_MIXER_LINE3	声卡的第 3 输入

#### 1. 声卡的输入增益和输出增益调节

混音器的一个主要作用是对声卡的输入增益和输出增益进行调节，目前大部分声卡采用的是 8 位或者 16 位的增益控制器。程序员不需要关心这些，因为声卡驱动程序将它们变换成百分比的形式，即无论是输入增益还是输出增益，其取值范围都是从 0~100。在使用混音器编程时，可以通过使用宏 SOUND\_MIXER\_READ 来读取混音通道的增益大小，下面是获取麦克风的输入增益的方法：

```
ioctl(fd, SOUND_MIXER_READ(SOUND_MIXER_MIC), &vol);
```

如果设备只有一个混音通道的单声道，那么返回的增益大小保存在低位字节中。如果

设备支持多个混音通道的双声道，那么返回的增益大小包括左、右声道值两个部分，左声道的音量保存在低位字节，而右声道的音量则保存高位字节。下面是提取左、右声道的增益大小示例代码。

```
int left, right;
left = vol & 0xff;
right = (vol & 0xff00) >> 8;
```

通过 SOUND\_MIXER\_WRITE 宏来设置混音通道的增益大小，与获取增益值时基本相同。下面是设置麦克风的输入增益：

```
vol = (right << 8) + left;
ioctl(fd, SOUND_MIXER_WRITE(SOUND_MIXER_MIC), &vol);
```

## 2. 查询混音器的信息

声卡驱动程序提供了多个系统调用 ioctl 来获得混音器的信息，返回一个整型的位掩码 (bitmask)，掩码的每一位分别代表一个特定的混音通道，如果相应的位为 1，则说明与之对应的混音通道是可用的。下面是通过 SOUND\_MIXER\_READ\_RECMASK 返回的位掩码，检查 CD 输入是否为一个有效的混音通道。

```
ioctl(fd, SOUND_MIXER_READ_DEVMASK, &devmask);
if (devmask & SOUND_MIXER_CD)
    printf("The CD input is supported");
```

检查 CD 输入是否为有效的录音源代码。

```
ioctl(fd, SOUND_MIXER_READ_RECMASK, &recmask);
if (recmask & SOUND_MIXER_CD)
    printf("The CD input can be a recording source");
```

通过使用宏 SOUND\_MIXER\_READ\_RECSRC 可以查询当前声卡正在使用的录音源。

```
if (ioctl(mixer_fd, SOUND_MIXER_READ_RECSRC, &mask) == -1)
{
    perror("/dev/mixer");
}
printf("%x\n", mask);
```

通过使用宏 SOUND\_MIXER\_WRITE\_RECSRC 对当前声卡使用的录音源进行设置，下面是将 CD 输入作为声卡的录音源使用的代码。

```
devmask = SOUND_MIXER_CD;
ioctl(fd, SOUND_MIXER_WRITE_DEVMASK, &devmask);
```

可以通过 SOUND\_MIXER\_READ\_STEREODEVS 查询混音通道是否对立体声提供支持。

### 10.4.3 ALSA 应用程序编程

对于 ALSA 应用程序编程，用户不用使用文件接口，可以使用 alsa-lib。下面给出一个播放 PCM 流的小程序及其编译和测试方法。代码文件名为 talsa.cpp，内容如下：

```

#include <iostream>
#include <alsa/asoundlib.h>
using namespace std;

int main(int argc, char *argv[])
{
    long          loops;
    int           rc;
    int           size;
    snd_pcm_t*    handle;        //PCM 设备句柄
    snd_pcm_hw_params_t* params; //硬件信息和 PCM 流配置
    unsigned int   val= 22050;   //采样频率
    int           dir;
    snd_pcm_uframes_t frames= 16; //采样精度
    char*         buffer;

    /*打开 PCM, 最后一个参数为 0 表示标准配置*/
    if ( (rc = snd_pcm_open(&handle, "default", SND_PCM_STREAM_PLAYBACK,
0)) < 0)
    {
        cerr << "unable to open pcm device: " << snd_strerror(rc) << endl;
        exit(1);
    }

    /*分配 snd_pcm_hw_params_t 结构体*/
    snd_pcm_hw_params_alloca(&params);
    /*初始化 hw_params*/
    snd_pcm_hw_params_any(handle, params);
    /*初始化访问权限*/
    snd_pcm_hw_params_set_access(handle, params, SND_PCM_ACCESS_RW_
INTERLEAVED);
    /*初始化采样格式*/
    snd_pcm_hw_params_set_format(handle, params, SND_PCM_FORMAT_S16_
LE);
    /*设置通道数*/
    snd_pcm_hw_params_set_channels(handle, params, 2);
    /*设置采样率*/
    snd_pcm_hw_params_set_rate_near(handle, params, &val, &dir);
    /*设置周期大小*/
    snd_pcm_hw_params_set_period_size_near(handle, params, &frames,
&dir);
    /*设置 hw_params*/
    if ( (rc = snd_pcm_hw_params(handle, params)) < 0)
    {
        cerr << "unable to set hw paramseters: " << snd_strerror(rc) <<
endl;
        exit(1);
    }

    snd_pcm_hw_params_get_period_size(params, &frames, &dir);
    size = frames * 4;
    buffer = new char[size];
    /*获得周期*/
    snd_pcm_hw_params_get_period_time(params, &val, &dir);
    /*3 秒钟的数据*/
    loops = 3000000 / val;

    while (loops > 0) {
        loops--;
        if ( (rc = read(0, buffer, size)) == 0)

```

```

    {
        cerr << "end of file on input" << std::endl;
        break;
    }
    else if (rc != size)
        cerr << "short read: read " << rc << " bytes" << endl;

    if ( (rc = snd_pcm_writei(handle, buffer, frames)) == -EPIPE)
    {
        cerr << "underrun occurred" << endl;
        snd_pcm_prepare(handle);
    }
    else if (rc < 0)
        cerr << "error from writei: " << snd_strerror(rc) << endl;
    else if (rc != (int)frames)
        cerr << "short write, write " << rc << " frames" << endl;
}
/*把所有挂起没有传输完的声音样本传输完全*/
snd_pcm_drain(handle);
/*关闭该音频流*/
snd_pcm_close(handle);
/*释放之前动态分配的缓冲区*/
free(buffer);

return 0;
}

```

编译上面的 ALSA 应用程序，采用下面的命令：

```
#g++ -o talsa talsa.cpp -lasound
```

运行应用程序测试，下面的命令可以产生随机的白噪声。

```
./talsa </dev/urandom
```

也可以使用 Windows 的录音工具，选择“开始” | “所有程序” | “附件” | “娱乐” | “录音机”命令，录制一段 wav 格式的文件，默认的音频格式是 PCM 22.050 kHz, 16 位，立体声。使用上面编译的 ALSA 应用程序播放录制的音频文件的方法为：

```
./talsa < rev.wav
```

## 10.5 音频设备驱动移植

目前笔者手中的 mini2440 使用的是 UDA1341 芯片，本节将通过移植 UDA1341 讲解移植音频设备驱动的过程。

### 10.5.1 添加 UDA1341 结构体

如果是买了友善之臂 mini2440 的用户，那么在附带的源代码中已经完成了这部分工作。如果读者的内核是从网上下载的，请下载 Linux-2.6.32 版本以上的内核，因为笔者测试的时候采用的是 Linux-2.6.32 版本，交叉编译器采用的是 arm-linux-gcc-4.4.3。修改的文

件为 arch/arm/mach-s3c2440 目录下的 mach-mini2440.c，修改的内容如下：

(1) 在 mach-mini2440.c 中包括头文件。

```
#include <sound/s3c24xx_uda134x.h>
```

(2) 在 mach-mini2440.c 中添加 UDA1341 设备结构。

```
static struct s3c24xx_uda134x_platform_data s3c24xx_uda134x_data = {
    .l3_clk = S3C2410_GPB4,
    .l3_data = S3C2410_GPB3,
    .l3_mode = S3C2410_GPB2,
    .model = UDA134X_UDA1341,
};

static struct platform_device s3c24xx_uda134x = {
    .name = "s3c24xx_uda134x",
    .dev = {
        .platform_data = &s3c24xx_uda134x_data,
    }
};
```

(3) 在下面的结构体中，添加注册 UDA1341 设备平台到内核。

```
static struct platform_device *mini2440_devices[] __initdata = {
    &s3c_device_usb,
    &s3c_device_rtc,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c0,
    &s3c_device_iis,
    &s3c_device_dm9k,
    &net_device_cs8900,
    &s3c24xx_uda134x,
};
```

## 10.5.2 修改录音通道

mini2440 使用录音通道是 VIN2，应该修改 sound/soc/codecs 目录下的 uda134x.c 文件，在函数 uda134x\_startup 中修改录音通道为 VIN2。

```
static int uda134x_startup(struct snd_pcm_substream *substream,
    struct snd_soc_dai *dai)
{
    struct snd_soc_pcm_runtime *rtd = substream->private_data;
    struct snd_soc_device *socdev = rtd->socdev;
    struct snd_soc_codec *codec = socdev->codec;
    struct uda134x_priv *uda134x = codec->private_data;
    struct snd_pcm_runtime *master_runtime;

    if (uda134x->master_substream) {
        master_runtime = uda134x->master_substream->runtime;
        pr_debug("%s constraining to %d bits at %d\n", __func__,
            master_runtime->sample_bits,
```

```

        master_runtime->rate);

    snd_pcm_hw_constraint_minmax(substream->runtime,
                                 SNDRV_PCM_HW_PARAM_RATE,
                                 master_runtime->rate,
                                 master_runtime->rate);

    snd_pcm_hw_constraint_minmax(substream->runtime,
                                 SNDRV_PCM_HW_PARAM_SAMPLE_BITS,
                                 master_runtime->sample_bits,
                                 master_runtime->sample_bits);

    uda134x->slave_substream = substream;
} else
    uda134x->master_substream = substream;

//修改录音通道为 VIN2
uda134x_write(codec, 2, 2|(5U<<2));
return 0;
}
}

```

### 10.5.3 内核中添加 UDA1341 驱动支持

上面对内核代码进行了修改，回到内核代码一级目录下，使用命令 make menuconfig 对内核以窗口的方式进行配置内核。进入内核配置界面后，选择 Device Drivers| Sound card support|Advanced Linux Sound Architecture 命令进入 ALSA 驱动配置界面，选择 ALSA for SoC audio support（位于选项的后部），同时选择对 Mixer API 和 PCM API 的支持，如图 10.2 所示。

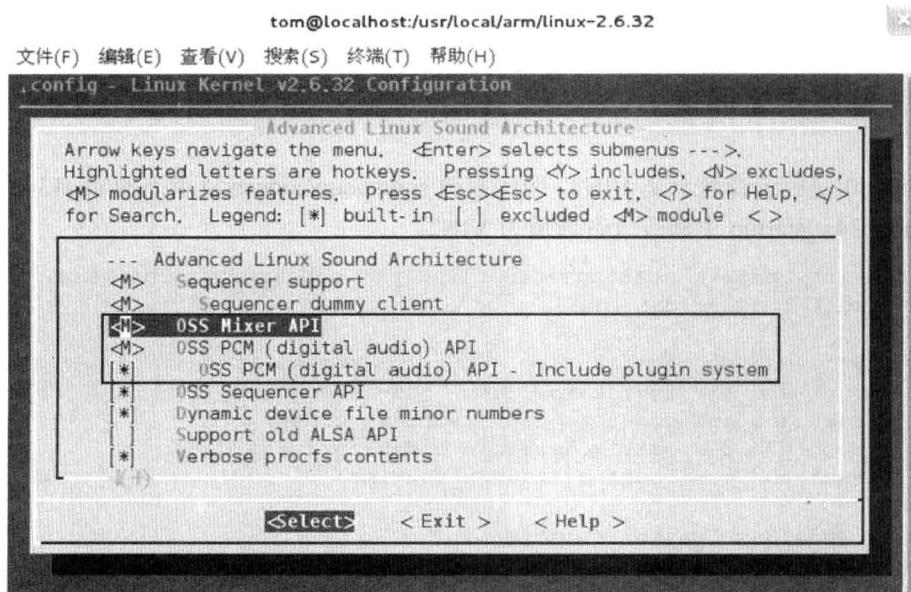


图 10.2 内核配置 ALSA for SoC audio support

**注意：**如果不添加 OSS PCM (digital audio) API 支持，在开发板上播放音乐时会出现无法找到设备的错误“audio: /dev/dsp: No such file or directory”。

进入 ALSA for SoC audio support 的配置界面后，选择 SoC I2S Audio support UDA134X wired to a S3C24XX，如图 10.3 所示。

保存配置后，使用 make clean 清除以前编译的临时文件，再使用 make zImage 编译内核。

```
# make clean
# make zImage
```

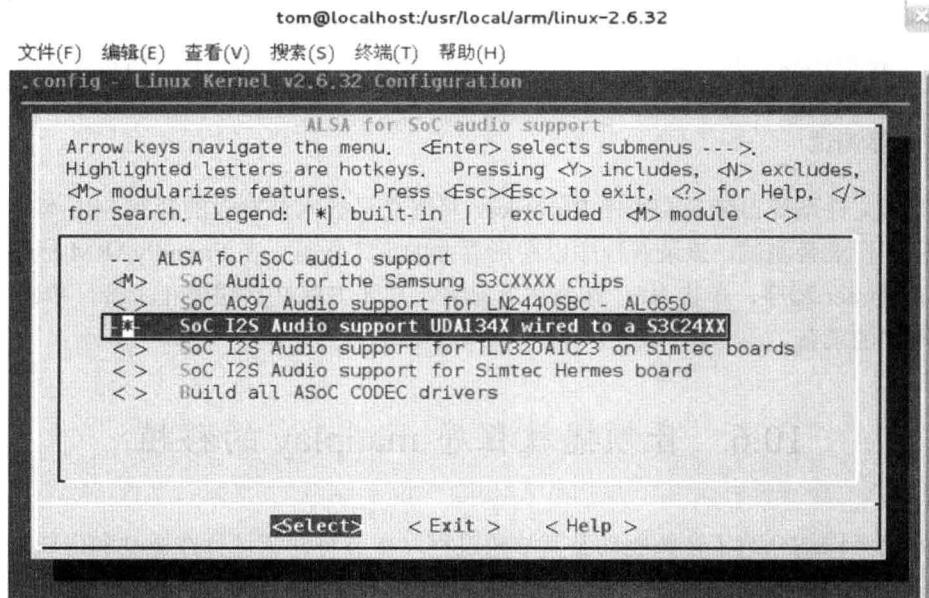


图 10.3 配置 UDA134X 芯片的驱动

#### 10.5.4 移植新内核并进行测试

将新生成的内核 zImage 使用命令 load flash kernel u 下载到开发板上。因为之前已经移植了整个系统，这里只需要移植新的内核映像文件。如果读者是第一次移植，则需要参考前面的章节将新生成的内核连同 Bootloader、文件系统一并下载到开发板上。

##### 1. 播放音频文件测试

使用文件系统中带的 madplay 播放 mp3 文件。播放命令为 madplay 音乐文件，播放过程如图 10.4 所示。在播放的过程中，将音响的输入线或者耳塞插入 mini2440 的音频 OUT 输出口。

```
# madplay The\ Calculation.mp3
```

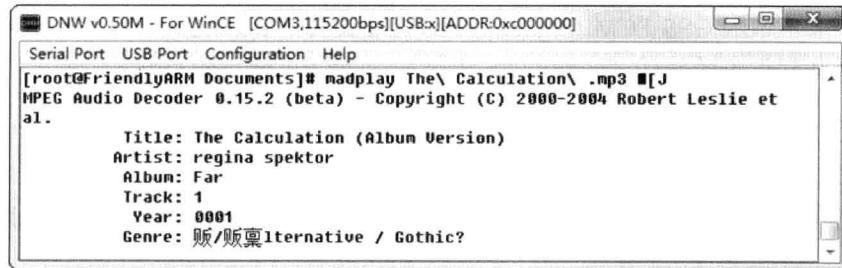


图 10.4 开发板上播放 mp3 文件

注意：如果读者的开发板中的文件系统不带 madplay 播放器，10.6 节将简单介绍 madplay 的移植过程。读者可以先移植 madplay，然后再测试 UDA1341 驱动的移植。

## 2. 录音测试

开发板文件系统自带录音程序 Recorder，将麦克风插入开发板的音频 MIC 输入口，运行该程序进行录音测试，该录音程序以图形界面的形式运行。在 FriendlyARM 标签页中双击运行 Recorder 程序，单击 REC 按钮进行录音，单击 STOP 按钮停止录音，单击 Play 按钮播放刚才的录音。

# 10.6 音频播放程序 madplay 的移植

前面已经详细介绍了音频驱动及其移植过程，本节将介绍运行在驱动和内核之上的播放器 madplay 的移植。

## 10.6.1 准备移植需要的源文件

移植 madplay 需要播放器代码、mp3 库文件和编码、解码库等。下面为移植需要准备相关源代码，读者可以到网上下载最新代码进行移植。

- madplay-0.15.2b.tar.gz：播放程序压缩包；
- libmad-0.15.1b.tar.gz：madplay 库文件；
- libid3tag-0.15.1b.tar.gz：针对 mp3 的解码库；
- zlib-1.2.8.tar.gz：用于文件压缩和解压。

## 10.6.2 交叉编译

编译 madplay 时需要库文件的支持，因此先编译它依赖的库文件，然后再编译播放器。编译的过程包括 configure 生成 Makefile，设置安装路径，设置交叉编译工具，编译和安装等，与多数应用程序交叉编译过程类似。

## 1. 编译安装 zlib-1.2.8.tar.gz

解压文件 `zlib-1.2.8.tar.gz`, 进入解压目录, 使用 `configure` 生成 `Makefile`, 设置交叉编译工具, 进行编译和安装, 具体过程如下:

```
# tar zxvf zlib-1.2.8.tar.gz
# cd zlib-1.2.8
# ./configure --prefix=/usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc
```

修改 `Makefile`, 将 x86 上的编译工具修改为交叉编译工具。

```
CC=arm-linux-gcc
LDSHARED=arm-linux-gcc -shared
-Wl,-soname,libz.so.1,--version-script,zlib.map
CPP=arm-linux-gcc -E
AR=arm-linux-ar
RANLIB=arm-linux-ranlib
TAR=arm-linux-tar
```

修改完 `Makefile` 后, 使用 `make` 和 `make install` 进行编译和安装。

```
# make
# make install
```

## 2. 编译安装 libid3tag-0.15.1b.tar.gz

与编译 `zlib-1.2.8.tar.gz` 的过程基本类似, 首先解压文件 `libid3tag-0.15.1b.tar.gz`, 进入解压目录, 使用 `configure` 生成 `Makefile`, 设置交叉编译工具, 进行编译和安装。不同的是在 `configure` 命令中还指定了依赖的头文件和库文件路径, 具体过程如下:

```
# tar zxvf libid3tag-0.15.1b.tar.gz
# cd libid3tag-0.15.1b
# ./configure --prefix=/usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc
--host=arm-linux \
--enable-static --disable-shared \
CPPFLAGS=-I/usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc/include \
LDFLAGS=-L/usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc/lib
```

## 3. 编译安装 libmad-0.15.1b.tar.gz

交叉编译 `libmad-0.15.1b.tar.gz` 的过程和 `libid3tag-0.15.1b.tar.gz` 基本相同。编译的时候遇到编译选项的错误, 通过下载补丁文件 `libmad.patch`, 打上补丁后, 编译安装通过。

```
# tar zxvf libmad-0.15.1b.tar.gz
# cd libmad-0.15.1b
# patch -p1<libmad.patch
# ./configure --prefix=/usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc
--host=arm-linux \
--enable-static --disable-shared \
CPPFLAGS=-I/usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc/include \
LDFLAGS=-L/usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc/lib
```

## 4. 编译安装 madplay-0.15.2b.tar.gz

交叉编译安装 `madplay-0.15.2b.tar.gz` 的过程如下:

```
# tar zxvf madplay-0.15.2b.tar.gz
# cd madplay-0.15.2b
# ./configure --prefix=/usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc
--host=arm-linux \
--enable-static --disable-shared \
CPPFLAGS=-I/usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc/include \
LDFLAGS=-L/usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc/lib
```

### 10.6.3 移植和测试

可执行文件 madplay 安装在目录 /usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc/bin 下。在交叉编译之前最好能在 X86 平台上编译通过，通过 ldd 查看 madplay 所依赖的库文件，将需要的对应 /usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc/lib 下的库文件一并下载到开发板上。在开发板上执行测试。

```
# madplay madplay The_Calculation.mp3.mp3
```

在开发板上播放音频文件结束后，打印播放歌曲的帧数、所花时间等信息。

```
7317 frames decoded (0:03:11.1), +0.3 dB peak amplitude, 38 clipped samples
```

### 10.6.4 编译中可能遇到的问题

在移植的过程中遇到几个典型的问题，这里列出来供读者参考。

(1) 更换编译器时，可能会出现的问题。

**ELF file OS ABI invalid**

解决方法：库文件格式不对，可能是依赖了错误版本的库文件，清除掉库文件路径：

```
# export LD_LIBRARY_PATH=
```

(2) 写 configure 参数时应该小心格式，格式错误时，可能出现以下问题：

```
checking for suffix of object files... configure: error: cannot compute
suffix of object files: cannot compile
```

解决方法：通过查看 config.log, ac\_cv\_env\_CPPFLAGS\_value 和 ac\_cv\_env\_LDFLAGS\_value 的设置有错。Configure 命令后的 -I 和路径 /usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc/include 之间不能存在空格 -L 和路径 /usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc/lib 之间不能有空格。

## 10.7 小结

本章重点理解音频驱动的代码，熟悉在配置内核时如何添加音频驱动 ALSA 框架和 OSS 框架，同时了解不同的音频接口编程，最后掌握针对 Mini2440 的音频驱动移植和音频播放器的移植过程。本章从底层驱动讲到内核配置，再到音频编程和音频应用程序移植，通过本章的学习，读者可以独立开发嵌入式音频系统。

# 第 11 章 SD 卡驱动移植

SD 卡 (Secure Digital Memory Card)，安全数码卡，是一种基于 Flash 的新一代存储设备，被广泛地用于便携式设备，例如移动电话、数码相机、个人数码助理（PDA）和多媒体播放器等。SD 卡拥有体积小、容量大、数据传输快、移动灵活及安全等优点。因其价格低廉，应用也越来越广泛，本章将重点介绍其驱动分析和移植过程。

## 11.1 SD 卡简介

SD 存储卡是专门为满足安全、大容量和内置于消费者的新型语音和视频电子设备中而设计的。SD 内存卡将包含的机械保护装置，遵循 SDMI 标准，具有安全、快速、大容量等特性。SD 卡的安全系统采用相互认证和“新密码算法”，以防止卡中的内容被非法使用。下面将从以下几个方面简单介绍 SD 卡协议内容。

### 11.1.1 SD 卡系统概念

下面分别描述 SD 卡的读写、容量、速度、电压等特性和分类。

- 读写特性：根据读写特性可以将 SD 卡分为两种。一种为读/写卡，这种卡生产出来就是一张空白卡，专门用于记录用户的视频声音、图像的大容量记忆卡；另一种为只读卡，这种卡在制造时就定制了内容，其典型的应用是在软件、音频或视频等的发行媒体中。
- 支持电压：根据支持的电压可以将 SD 卡分为高电压 SD 卡和双重电压 SD 卡两类。
- 卡容量：根据卡的容量大小将 SD 分为两类型。一类为标准容量的 SD 卡，其支持的容量上限为 2GB，包括 2GB 在内；另一类为高容量 SD 卡，其容量超过 2GB，最大可达 32GB。
- 速度：根据速度类可以将 SD 卡分为 4 种速度类。类 0，这种类型卡兼具所有类型的优点；类 2，其速度大于等于 2MB/S；类 4，其速度大于等于 4MB/S；类 6，其速度大于等于 6MB/S。高容量 SD 卡支持速度类描述，其性能相当于或超过类 2。

### 11.1.2 SD 卡寄存器

每张卡都有一系列寄存器的信息，寄存器的信息如表 11.1 所示。

表 11.1 SD 卡寄存器信息

名字	宽度	描述
CID	128	卡识别号，每张卡都有唯一的识别号
RCA	16	发布卡的地址，卡的局部系统地址，在初始化过程中，由主机和卡动态支持
DSR	16	驱动级寄存器，配置卡的驱动输出
CSD	128	卡的协议数据，关于卡的操作状态数据
SCR	64	卡配置寄存器，关于卡特性容量的信息
OCR	32	操作状态寄存器
SSR	512	SD 状态，有关卡拥有的特性信息
CSR	32	卡状态，有关卡状态的信息

### 11.1.3 SD 功能描述

主机与卡之间的通信都是由主机控制的，主机发送的命令有两种类型，分别为广播命令和地址（点对点）命令。

- 广播命令：该命令是发给所有的卡，有些广播命令需要响应。
- 地址（点对点）命令：这些命令发往具体地址的卡，并且从这些卡生成响应。
- 卡识别模式：主机被复位或者在总线上寻找新卡时，主机处于该状态下。卡在复位以后和收到 SEND\_RCA 命令以前都处于此模式下。
- 数据传输模式：卡在它们的 RCA 第一次发布后进入数据传输模式。主机识别总线上所有的卡后进入数据传输模式。

下面通过表 11.2 说明卡的状态与操作模式之间的依赖关系。SD 的每种状态都关联一种操作模式，其状态图将在随后进行介绍。

表 11.2 卡的状态和操作模式的对应关系

卡 状 态	操 作 模 式
无活动状态	无活动
空闲态	
准备态	卡识别模式
识别态	
等待态	
传输态	
发送数据态	数据传输模式
接收数据态	
编程态	
断开态	

#### 1. 操作状态的验证

通过一系列过程后，主机才能识别卡。下面给出它们的通信过程。

- 在主机和卡通信前，主机不知道卡支持的电压，卡也不知道是否支持主机当前提供的电压。主机将发布一个复位命令（CMD0），带着它能提供给卡的电压信息。

- 为了验证 SD 卡的接口操作状态，主机发送 SEND\_IF\_COND(CMD8)，SD 卡通过分析 SEND\_IF\_COND 命令参数检查操作状态的有效性，主机通过检查 SD 分析后的响应来判断电压的有效性。
- 如果 SD 能够在提供的电压下操作，则发回的响应带上提供的电压，且检验模式被设置在命令参数中。如果 SD 卡不支持主机提供的电压，则不响应且保持在空闲态下。在发送 ACMD41 命令初始化高容量 SD 卡前，强制发送 CMD8 命令。
- 强制低电压主机在发送 CMD8 前发送 ACMD41。万一双重电压 SD 卡没有收到 CMD8 命令且工作在高电压状态，在这种情况下，低电压主机不发送 CMD8 命令给卡，则收到 ACMD41 后进入无活动状态。
- SD\_SEND\_OP\_COND (ACMD41) 命令是为 SD 卡主机识别卡或电压不匹配时拒绝卡的机制而设计的。主机发送命令操作数代表要求的电压窗口大小。如果 SD 卡在所给的范围内不能实现数据传输，将放弃下一步的总线操作而进入无活动状态。操作状态寄存器也将被定义。
- 在主机发出复位命令 (CMD0) 后，主机将先发送 CMD8 再发送 ACMD41 命令重新初始化 SD 卡。

卡的识别模式的状态可以用下面的状态图表示，如图 11-1 所示。

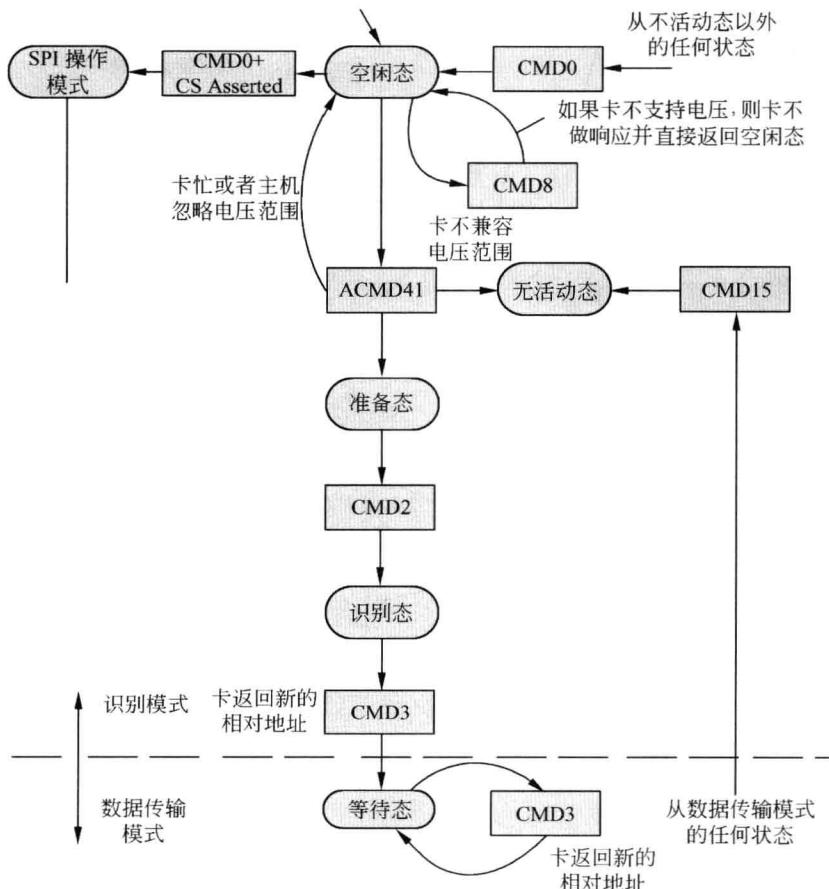


图 11.1 卡在识别模式下的命令流程

## 2. 卡的初始化和识别处理

当总线被激活后，主机就开始卡的初始化和识别处理。初始化处理从设置它的操作状态和设置 OCR 中的 HCS 比特位命令 SD\_SEND\_OP\_COND (ACMD41) 开始。HCS 比特位被设置为 1 表示主机支持高容量 SD 卡。HCS 被设置为 0 表示主机不支持高容量 SD 卡。

卡的初始化和识别更详细的流程，如图 11.2 所示。

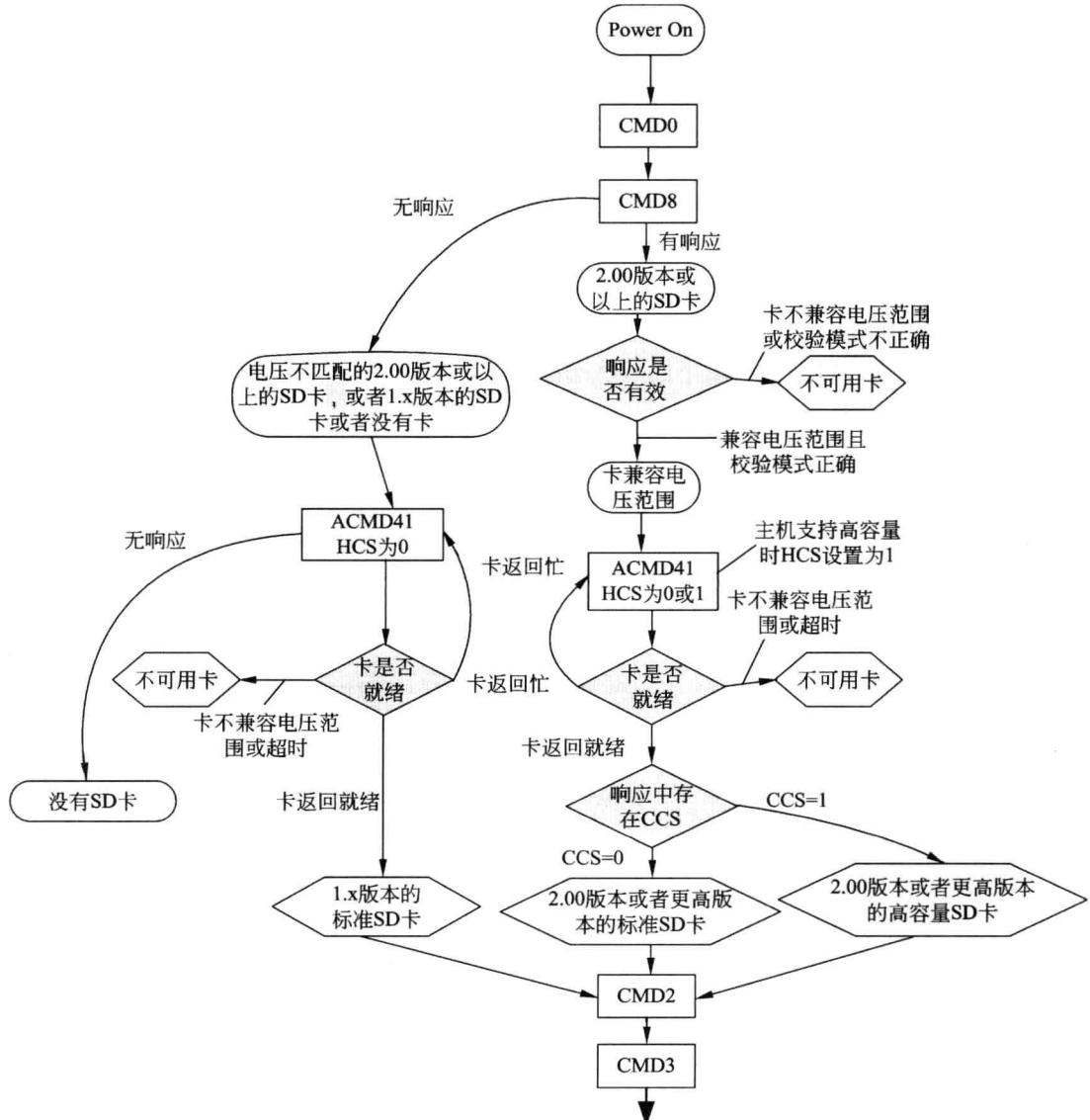


图 11.2 卡的初始化和识别流程

## 3. 数据传输模式

卡的识别模式结束后，主机时钟  $f_{pp}$  (数据传输时钟速率) 将保持为  $f_{OD}$  (卡识别模式下的时钟)，因为有些卡对操作时钟有限制。主机必须发送 SEND\_CSD (CMD9) 来获得

卡规格数据寄存器内容，如块大小、卡容量等。广播命令 SET\_DSR (CMD4) 配置所有识别卡的驱动阶段。它对 DSR 寄存器进行编程以适应应用总线布局、总线上的卡数目和数据传输频率。

SD 卡数据传输模式下的状态图，如图 11.3 所示。

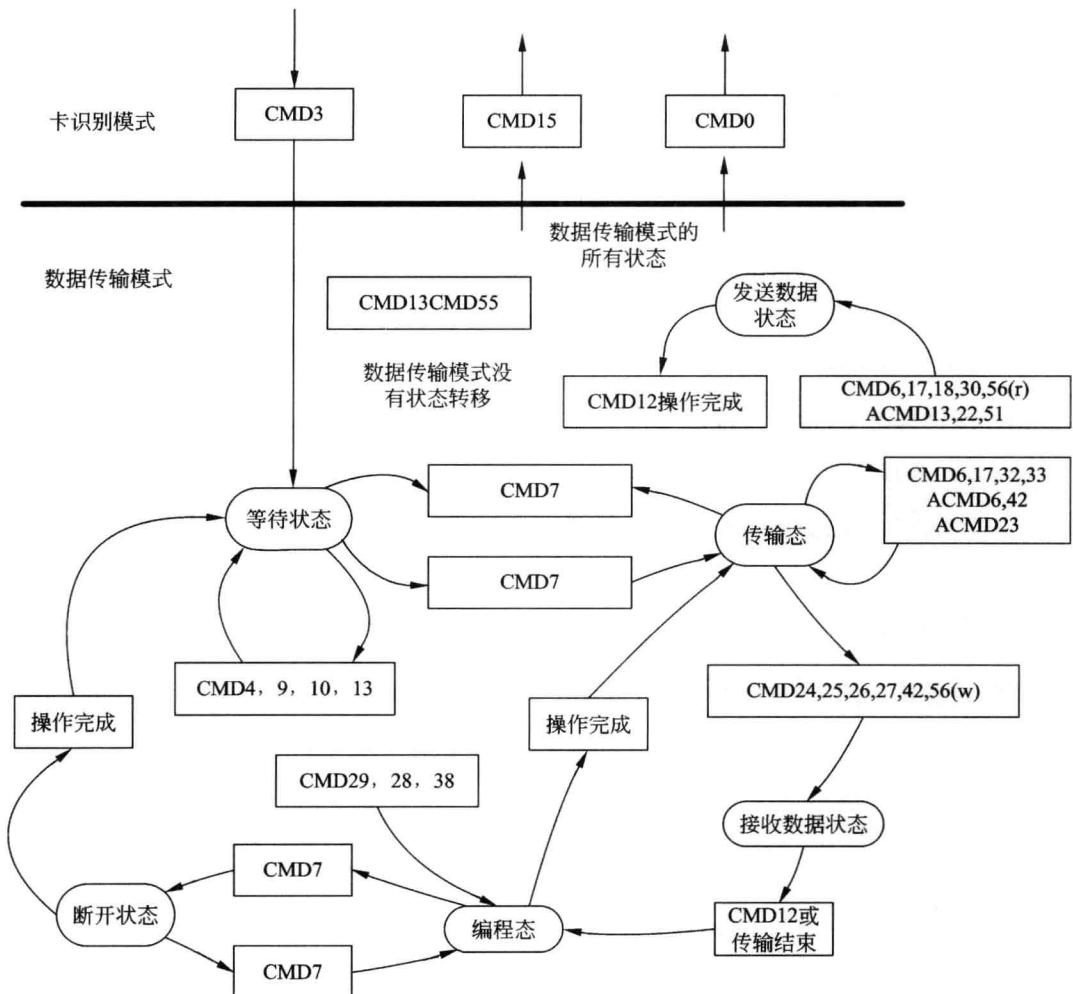


图 11.3 SD 卡数据传输模式的状态图

## 11.2 SD 卡驱动程序分析

SD 卡驱动程序包括驱动的注册和注销、设备接口函数和 I/O 操作。在 linux-2.6.32 内核 MMC 子系统中支持 SD 卡驱动。本节对 MMC 源码进行分析，后面将介绍 SD 卡驱动移植过程。MMC 子系统在 driver/mmc 目录下进行描述，该目录下包括 host、core、card 这 3 个文件夹，下面分别对这 3 个部分进行介绍。

## 11.2.1 host 驱动部分

host 驱动部分是针对不同类型主机的驱动，支持的开发板包括 atmel、S3C 等。这里就以 S3C 系统为例介绍 host 部分的主要内容。

### 1. 驱动的注册函数

驱动的注册函数 s3cmci\_init(), 用于注册平台设备驱动。

```
static int __init s3cmci_init(void)
{
    return platform_driver_register(&s3cmci_driver); //注册平台设备驱动
}
```

### 2. 驱动注销函数

驱动注销函数 s3cmci\_exit(), 用于注销平台设备驱动。

```
static void __exit s3cmci_exit(void)
{
    platform_driver_unregister(&s3cmci_driver); //注销平台设备驱动
}
```

### 3. 接口函数

平台设备接口函数包括 probe、remove、shutdown、suspend、resume。其结构如下：

```
static struct platform_driver s3cmci_driver = {
    .driver = {
        .name      = "s3c-sdi",
        .owner     = THIS_MODULE,
        .pm        = s3cmci_pm_ops,
    },
    .id_table   = s3cmci_driver_ids,
    .probe      = s3cmci_probe,
    .remove     = __devexit_p(s3cmci_remove),
    .shutdown   = s3cmci_shutdown,
};
```

### 4. 探针函数

探针函数 s3cmci\_probe(), 用于分配 s3cmci\_host 结构体，然后对该结构体进行设置。对结构体 mmc\_host 进行设置，将结构体 mmc 添加到主机。

```
static int __devinit s3cmci_probe(struct platform_device *pdev)
{
    struct s3cmci_host *host;
    struct mmc_host *mmc;
    int ret;
    int is2440;
    int i;

    is2440 = platform_get_device_id(pdev) ->driver_data;
```

```

/*为主机设备分配空间*/
mmc = mmc_alloc_host(sizeof(struct s3cmci_host), &pdev->dev);
if (!mmc) {
    ret = -ENOMEM;
    goto probe_out;
}

for (i = S3C2410_GPE(5); i <= S3C2410_GPE(10); i++) {
    ret = gpio_request(i, dev_name(&pdev->dev));
    if (ret) {
        dev_err(&pdev->dev, "failed to get gpio %d\n", i);

        for (i--; i >= S3C2410_GPE(5); i--)
            gpio_free(i);

        goto probe_free_host;
    }
}
/*对 host 结构体各个字段进行设置*/
host = mmc_priv(mmc);
host->mmc = mmc;
host->pdev = pdev;
host->is2440 = is2440;
/*设置平台数据*/
host->pdata = pdev->dev.platform_data;
if (!host->pdata) {
    pdev->dev.platform_data = &s3cmci_def_pdata;
    host->pdata = &s3cmci_def_pdata;
}
/*初始化自旋锁，自旋锁在使用前应该被初始化*/
spin_lock_init(&host->complete_lock);
/*函数 tasklet_init ()用于初始化一个 tasklet，参数 pio_tasklet 是软中断响应函数*/
tasklet_init(&host->pio_tasklet, pio_tasklet, (unsigned long) host);
/*结构体参数设置*/
if (is2440) {
    host->sdiimsk = S3C2440_SDIIMSK;
    host->sdidata = S3C2440_SDIDATA;
    host->clk_div = 1;
} else {
    host->sdiimsk = S3C2410_SDIIMSK;
    host->sdidata = S3C2410_SDIDATA;
    host->clk_div = 2;
}

host->complete_what = COMPLETION_NONE;
host->pio_active = XFER_NONE;

#endif CONFIG_MMC_S3C_PIODMA
host->dodma = host->pdata->dma;
#endif
/*获取平台资源信息*/
host->mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
if (!host->mem) {
    dev_err(&pdev->dev,
            "failed to get io memory region resource.\n");
}

```

```

        ret = -ENOENT;
        goto probe_free_gpio;
    }
/*该函数的任务是检查申请的资源是否可用，如果可用则申请成功，并标志为已经使用，其他
驱动想再申请该资源时就会失败*/
host->mem = request_mem_region(host->mem->start,
                                 resource_size(host->mem), pdev->name);

if (!host->mem) {
    dev_err(&pdev->dev, "failed to request io memory region.\n");
    ret = -ENOENT;
    goto probe_free_gpio;
}
/*系统在运行时，外设的I/O内存资源的物理地址是已知的，由硬件的设计决定。但是CPU
通常并没有为这些已知的外设I/O内存资源的物理地址预定义虚拟地址范围，驱动程序并不能
直接通过物理地址访问I/O内存资源，而必须将它们映射到内核虚地址空间内（采用页表），
然后才能根据映射所得到的内核虚地址范围，通过访内指令访问这些I/O内存资源。*/
host->base = ioremap(host->mem->start, resource_size(host->mem));
if (!host->base) {
    dev_err(&pdev->dev, "failed to ioremap() io memory region.\n");
    ret = -EINVAL;
    goto probe_free_mem_region;
}
/*获取设备的中断号*/
host->irq = platform_get_irq(pdev, 0);
if (host->irq == 0) {
    dev_err(&pdev->dev, "failed to get interrupt resouce.\n");
    ret = -EINVAL;
    goto probe_iounmap;
}
/*向系统申请中断*/
if (request_irq(host->irq, s3cmci_irq, 0, DRIVER_NAME, host)) {
    dev_err(&pdev->dev, "failed to request mci interrupt.\n");
    ret = -ENOENT;
    goto probe_iounmap;
}
/*关闭中断*/
disable_irq(host->irq);
host->irq_state = false;
if (!host->pdata->no_detect) {
    ret = gpio_request(host->pdata->gpio_detect, "s3cmci detect");
    if (ret) {
        dev_err(&pdev->dev, "failed to get detect gpio\n");
        goto probe_free_irq;
    }
}
/*给定端口号转换为中断号*/
host->irq_cd = s3c2410_gpio_getirq(host->pdata->gpio_detect);

if (host->irq_cd >= 0) {
    if (request_irq(host->irq_cd, s3cmci_irq_cd,
                    IRQF_TRIGGER_RISING |
                    IRQF_TRIGGER_FALLING,
                    DRIVER_NAME, host)) {
        dev_err(&pdev->dev,
                "can't get card detect irq.\n");
        ret = -ENOENT;
        goto probe_free_gpio_cd;
    }
}

```

```

    } else {
        dev_warn(&pdev->dev,
                 "host detect has no irq available\n");
        gpio_direction_input(host->pdata->gpio_detect);
    }
} else
    host->irq_cd = -1;

if (!host->pdata->no_wprotect) {
    ret = gpio_request(host->pdata->gpio_wprotect, "s3cmci wp");
    if (ret)
        dev_err(&pdev->dev, "failed to get writeprotect\n");
    goto probe_free_irq_cd;
}

    gpio_direction_input(host->pdata->gpio_wprotect);
}

/*获取dma通道的控制权*/
if (s3cmci_host_usedma(host)) {
    host->dma = s3c2410_dma_request(DMACH_SDI, &s3cmci_dma_client,
                                      host);
    if (host->dma < 0) {
        dev_err(&pdev->dev, "cannot get DMA channel.\n");
        if (!s3cmci_host_canpio())
            ret = -EBUSY;
        goto probe_free_gpio_wp;
    } else {
        dev_warn(&pdev->dev, "falling back to PIO.\n");
        host->dodma = 0;
    }
}
}

/*获取时钟响应给时钟的生产者 producer()*/
host->clk = clk_get(&pdev->dev, "sdi");
if (IS_ERR(host->clk)) {
    dev_err(&pdev->dev, "failed to find clock source.\n");
    ret = PTR_ERR(host->clk);
    host->clk = NULL;
    goto probe_free_dma;
}

/*当时钟源运行的时候通知系统，参数 host->clk 为时钟源*/
ret = clk_enable(host->clk);
if (ret)
    dev_err(&pdev->dev, "failed to enable clock source.\n");
    goto clk_free;
}

/*获得当前时钟频率*/
host->clk_rate = clk_get_rate(host->clk);
/*下面是mmc结构体参数的设置*/
mmc->ops      = &s3cmci_ops;
mmc->ocr_avail = MMC_VDD_32_33 | MMC_VDD_33_34;
#ifndef CONFIG_MMC_S3C_HW_SDIO_IRQ
    mmc->caps   = MMC_CAP_4_BIT_DATA | MMC_CAP_SDIO_IRQ;
#else
    mmc->caps   = MMC_CAP_4_BIT_DATA;
#endif
mmc->f_min    = host->clk_rate / (host->clk_div * 256);
mmc->f_max    = host->clk_rate / host->clk_div;

if (host->pdata->ocr_avail)

```

```

mmc->ocr_avail = host->pdata->ocr_avail;

mmc->max_blk_count = 4095;
mmc->max_blk_size = 4095;
mmc->max_req_size = 4095 * 512;
mmc->max_seg_size = mmc->max_req_size;

mmc->max_phys_segs = 128;
mmc->max_hw_segs = 128;

dbg(host, dbg_debug,
    "probe: mode:%s mapped mci_base:%p irq:%u irq_cd:%u dma:%u.\n",
    (host->is2440?"2440":""),
    host->base, host->irq, host->irq_cd, host->dma);

ret = s3cmci_cpufreq_register(host);
if (ret) {
    dev_err(&pdev->dev, "failed to register cpufreq\n");
    goto free_dmabuf;
}

ret = mmc_add_host(mmc);
if (ret) {
    dev_err(&pdev->dev, "failed to add mmc host.\n");
    goto free_cpufreq;
}

s3cmci_debugfs_attach(host);

platform_set_drvdata(pdev, mmc);
dev_info(&pdev->dev, "%s - using %s, %s SDIO IRQ\n", mmc_hostname(mmc),
        s3cmci_host_usedma(host) ? "dma" : "pio",
        mmc->caps & MMC_CAP_SDIO_IRQ ? "hw" : "sw");

return 0;
/*注册带CPU频率的host驱动*/
free_cpufreq:
    s3cmci_cpufreq_deregister(host);

free_dmabuf:
    clk_disable(host->clk);

clk_free:
    clk_put(host->clk);

probe_free_dma:
    if (s3cmci_host_usedma(host))
        s3c2410_dma_free(host->dma, &s3cmci_dma_client);

probe_free_gpio_wp:
    if (!host->pdata->no_wprotect)
        gpio_free(host->pdata->gpio_wprotect);

probe_free_gpio_cd:
    if (!host->pdata->no_detect)
        gpio_free(host->pdata->gpio_detect);

probe_free_irq_cd:
    if (host->irq_cd >= 0)
        free_irq(host->irq_cd, host);

```

```

probe_free_irq:
    free_irq(host->irq, host);

probe_iounmap:
    iounmap(host->base);

probe_free_mem_region:
    release_mem_region(host->mem->start, resource_size(host->mem));

probe_free_gpio:
    for (i = S3C2410_GPE(5); i <= S3C2410_GPE(10); i++)
        gpio_free(i);

probe_free_host:
    mmc_free_host(mmc);

probe_out:
    return ret;
}

```

## 5. mmc 接口函数

mmc 子系统的接口函数包括 request、set\_ios、get\_ro、get\_cds。其结构如下：

```

static struct mmc_host_ops s3cmci_ops = {
    .request      = s3cmci_request,           // 实现命令和数据的发送
    .set_ios       = s3cmci_set_ios,          // 根据核心层传来的 ios 来设置硬件 IO
    .get_ro        = s3cmci_get_ro,           // 从 GPIO 口读取，判断卡是否写保护
    .get_cd        = s3cmci_card_present,     // 从 GPIO 口读取，判断卡是否存在
    .enable_sdio_irq = s3cmci_enable_sdio_irq, // 允许 STDIO 中断请求
};

```

## 6. 传递结构体为 mmc\_request 类型的请求

函数 s3cmci\_request() 用于 CORE 部分发送 mrq 请求。

```

static void s3cmci_request(struct mmc_host *mmc, struct mmc_request *mrq)
{
    struct s3cmci_host *host = mmc_priv(mmc);

    host->status = "mmc request";
    host->cmd_is_stop = 0;
    host->mrq = mrq;
/* 如果卡准备就绪，则通过 s3cmci_send_request() 发送请求，将 mrq 赋给 host->mrq，如果卡没有准备就绪，则调用 mmc_request_done() 终止请求 */
    if (s3cmci_card_present(mmc) == 0) {
        dbg(host, dbg_err, "%s: no medium present\n", __func__);
        host->mrq->cmd->error = -ENOMEDIUM;
        mmc_request_done(mmc, mrq);
    } else
/* 函数 s3cmci_send_request() 首先判断是否为发送数据命令，如果为发送数据则通过函数 s3cmci_send_request() 建立数据，然后判断是否为 dma 方式，如果为 dma 方式则通过 dma 方式发送数据，否则采用 fifo 方式发送数据。如果为命令则通过函数 s3cmci_send_command() 发送命令 */
        s3cmci_send_request(mmc);
}

```

## 11.2.2 core 驱动部分

core 驱动部分完成不同协议和规范的实现，包括设置在 11.1 节中介绍的有关 SD 卡相关状态或修改状态、修改寄存器等操作。

### 1. 用于卡的探测和初始化函数 mmc\_sd\_init\_card()

在重启时，函数 mmc\_sd\_init\_card() 参数 oldcard 中包含准备初始化的卡，该函数检测卡的有效性，并对该卡初始化。该函数首先让卡的状态回到空闲态，然后设置操作状态寄存器，接着进行 SD 卡主机识别或电压匹配，正确识别和匹配后，读取卡的识别号；比较读取的 CID 与原来的 CID 是否相同，不相同则需要重新为卡分配结构体；最后对卡进行设置和初始化。

```
static int mmc_sd_init_card(struct mmc_host *host, u32 ocr,
    struct mmc_card *oldcard)
{
    struct mmc_card *card;
    int err;
    u32 cid[4];
    unsigned int max_dtr;

    BUG_ON(!host);
    WARN_ON(!host->claimed);
    /* 改变状态寄存器 OCR 的值时，需要卡的状态回到空闲态。等待 1ms 让卡响应 */
    mmc_go_idle(host);
    /* SD_SEND_IF_COND 是用于验证 SD 卡接口操作状态的有效性命令（CMD8）。如果
     * SD_SEND_IF_COND 指示为符合 SD2.0 标准的卡，则设置操作状态寄存器 ocrbit30 指示能够
     * 处理块地址 SDHC 卡 */
    err = mmc_send_if_cond(host, ocr);
    if (!err)
        ocr |= 1 << 30;
    /* SD_SEND_OP_COND（ACMD41）该命令为 SD 卡主机识别或电压不匹配时拒绝机制而设计 */
    err = mmc_send_app_op_cond(host, ocr, NULL);
    if (err)
        goto err;
    /* 从卡中读取 CID，卡的识别号 */
    if (mmc_host_is_spi(host))
        err = mmc_send_cid(host, cid);
    else
        err = mmc_all_send_cid(host, cid);
    if (err)
        goto err;
    /* 比较读取的 CID 与原来的 CID 是否相同 */
    if (oldcard) {
        if (memcmp(cid, oldcard->raw_cid, sizeof(cid)) != 0) {
            err = -ENOENT;
            goto err;
        }
        card = oldcard;
    } else {
        /* 为卡分配结构体 */
    }
}
```

```

card = mmc_alloc_card(host, &sd_type);
if (IS_ERR(card)) {
    err = PTR_ERR(card);
    goto err;
}
/*设置卡的类型*/
card->type = MMC_TYPE_SD;
memcpy(card->raw_cid, cid, sizeof(card->raw_cid));
}
if (!mmc_host_is_spi(host)) {
    /*获得卡的RCA，该寄存器表示发布卡的地址，卡的局部系统地址，在初始化过程中，由主机和卡动态支持*/
    err = mmc_send_relative_addr(host, &card->rca);
    if (err)
        goto free_card;
    /*设置总线模式*/
    mmc_set_bus_mode(host, MMC_BUSMODE_PUSHPULL);
}
if (!oldcard) {
    /*获得卡CSD，该寄存器表示卡的协议数据，关于卡的操作状态数据*/
    err = mmc_send_csd(card, card->raw_csd);
    if (err)
        goto free_card;
    /*卡的CSD结构的解码*/
    err = mmc_decode_csd(card);
    if (err)
        goto free_card;
    /*卡的CID结构解码*/
    mmc_decode_cid(card);
}
if (!mmc_host_is_spi(host)) {
    /*选择卡，后续的命令都依赖该操作*/
    err = mmc_select_card(card);
    if (err)
        goto free_card;
}
if (!oldcard) {
    /*获得卡的SCR，该寄存器表示卡配置寄存器，关于卡特性容量的信息*/
    err = mmc_app_send_scr(card, card->raw_scr);
    if (err)
        goto free_card;
    /*解码SCR结构*/
    err = mmc_decode_scr(card);
    if (err < 0)
        goto free_card;
    /*获得卡的switch信息*/
    err = mmc_read_switch(card);
    if (err)
        goto free_card;
}
if (mmc_host_is_spi(host)) {
    /*如果主机采用SPI总线则采用适当的CRC*/
    err = mmc_spi_set_crc(host, use_spi_crc);
    if (err)
        goto free_card;
}
/*尝试转化为高速*/
err = mmc_switch_hs(card);

```

```

if (err)
    goto free_card;
/*计算总线速率*/
max_dtr = (unsigned int)-1;

if (mmc_card_highspeed(card)) {
    if (max_dtr > card->sw_caps.hs_max_dtr)
        max_dtr = card->sw_caps.hs_max_dtr;
} else if (max_dtr > card->csd.max_dtr) {
    max_dtr = card->csd.max_dtr;
}
/*设置可能的最高主机的时钟*/
mmc_set_clock(host, max_dtr);
/*如果支持，转化为更宽的总线*/
if ((host->caps & MMC_CAP_4_BIT_DATA) &&
    (card->scr.bus_widths & SD_SCR_BUS_WIDTH_4)) {
    err = mmc_app_set_bus_width(card, MMC_BUS_WIDTH_4);
    if (err)
        goto free_card;

    mmc_set_bus_width(host, MMC_BUS_WIDTH_4);
}
/*检查卡只读是否激活*/
if (!oldcard) {
    if (!host->ops->get_ro || host->ops->get_ro(host) < 0) {
        printk(KERN_WARNING "%s: host does not "
               "support reading read-only "
               "switch. assuming write-enable.\n",
               mmc_hostname(host));
    } else {
        if (host->ops->get_ro(host) > 0)
            mmc_card_set_readonly(card);
    }
}

if (!oldcard)
    host->card = card;

return 0;
/*注销mmc卡*/
free_card:
    if (!oldcard)
        mmc_remove_card(card);
err:

    return err;
}

```

## 2. 删除SD卡函数mmc\_sd\_remove()

函数 mmc\_sd\_remove() 用于移除主机 host，释放当前卡。

```

static void mmc_sd_remove(struct mmc_host *host)
{
    BUG_ON(!host);
    BUG_ON(!host->card);

    mmc_remove_card(host->card);
    host->card = NULL;
}

```

}

### 3. 初始化主机结构体函数mmc\_alloc\_host()

函数 mmc\_alloc\_host() 在 host 层探针函数中被调用，该函数为主机结构体分配空间，并初始化主机结构体。

```

struct mmc_host *mmc_alloc_host(int extra, struct device *dev)
{
    int err;
    struct mmc_host *host;

    if (!idr_pre_get(&mmc_host_idr, GFP_KERNEL))
        return NULL;
    /*为主机结构体分配空间*/
    host = kzalloc(sizeof(struct mmc_host) + extra, GFP_KERNEL);
    if (!host)
        return NULL;
    spin_lock(&mmc_host_lock);
    /*分配新的 idr 入口*/
    err = idr_get_new(&mmc_host_idr, host, &host->index);
    spin_unlock(&mmc_host_lock);
    if (err)
        goto free;
    /*设置设备的名字，对 host 设备相关字段进行设置*/
    dev_set_name(&host->class_dev, "mmc%d", host->index);

    host->parent = dev;
    host->class_dev.parent = dev;
    host->class_dev.class = &mmc_host_class;
    /*函数 device_initialize() 用于初始化设备结构体，该函数一般为其他层做准备，这里是为 host 层做准备*/
    device_initialize(&host->class_dev);

    spin_lock_init(&host->lock);
    init_waitqueue_head(&host->wq);
    /*初始化一个工作队列*/
    INIT_DELAYED_WORK(&host->detect, mmc_rescan);
    INIT_DELAYED_WORK_DEFERRABLE(&host->disable,
        mmc_host_deeper_disable);
    /*初始化主机结构体的默认配置*/
    host->max_hw_segs = 1;
    host->max_phys_segs = 1;
    host->max_seg_size = PAGE_CACHE_SIZE;

    host->max_req_size = PAGE_CACHE_SIZE;
    host->max_blk_size = 512;
    host->max_blk_count = PAGE_CACHE_SIZE / 512;

    return host;

free:
    kfree(host);
    return NULL;
}

```

#### 4. 初始化主机硬件函数mmc\_add\_host()

函数 mmc\_add\_host() 用于增加设备类，并启动 host。

```
int mmc_add_host(struct mmc_host *host)
{
    int err;

    WARN_ON((host->caps & MMC_CAP_SDIO_IRQ) &&
            !host->ops->enable_sdio_irq);

    led_trigger_register_simple(dev_name(&host->class_dev), &host->led);

    err = device_add(&host->class_dev);
    if (err)
        return err;

#ifdef CONFIG_DEBUG_FS
    mmc_add_host_debugfs(host);
#endif

    mmc_start_host(host);

    return 0;
}
```

#### 5. 删除host硬件函数mmc\_remove\_host()

函数 mmc\_remove\_host() 与函数 mmc\_add\_host() 相对应，用于停止函数 mmc\_add\_host() 启动的 host，删除设备类，注销 LED trigger。

```
void mmc_remove_host(struct mmc_host *host)
{
    mmc_stop_host(host);

#ifdef CONFIG_DEBUG_FS
    mmc_remove_host_debugfs(host);
#endif

    device_del(&host->class_dev);

    led_trigger_unregister_simple(host->led);
}
```

#### 6. 释放主机结构体函数mmc\_free\_host()

函数 mmc\_free\_host() 与函数 mmc\_alloc\_host() 对应，释放函数 mmc\_alloc\_host() 初始化的 host 结构体。

```
void mmc_free_host(struct mmc_host *host)
{
    spin_lock(&mmc_host_lock);
    idr_remove(&mmc_host_idr, host->index);
    spin_unlock(&mmc_host_lock);
```

```

    put_device(&host->class_dev);
}

```

通过对 core 层函数进行分析，可以看出 core 层除了完成协议描述的部分外，还为 host 层提供了接口函数。

### 11.2.3 card 驱动部分

SD 卡属于块设备，card 驱动部分为了将 SD 卡驱动成为块设备。介绍该部分的内容时先介绍驱动结构体和接口函数结构体，然后介绍几个关键的驱动函数。

#### 1. 驱动的结构体 mmc\_driver

该结构体定义驱动的名字，驱动探针函数、驱动移除函数、驱动阻塞和驱动重启等函数。

```

static struct mmc_driver mmc_driver = {
    .drv      = {
        .name   = "mmcblk",
    },
    .probe     = mmc_blk_probe,
    .remove    = mmc_blk_remove,
    .suspend   = mmc_blk_suspend,
    .resume    = mmc_blk_resume,
};

```

#### 2. 块设备操作结构体 mmc\_bdops

结构体 mmc\_bdops 中定义了块设备操作的接口函数。

```

static const struct block_device_operations mmc_bdops = {
    .open      = mmc_blk_open,
    .release   = mmc_blk_release,
    .getgeo    = mmc_blk_getgeo,
    .owner     = THIS_MODULE,
};

```

#### 3. 块设备探针函数 mmc\_blk\_probe()

该函数主要完成检验卡支持的命令，分配 mmc\_blk\_data 结构体空间，设置块的大小，最后设置 card 的 driver\_data 字段，并注册 mmc 信息到系统。

```

static int mmc_blk_probe(struct mmc_card *card)
{
    struct mmc_blk_data *md;
    int err;

    char cap_str[10];
    /*检查卡支持的命令*/
    if (!(card->csd.cmdclass & CCC_BLOCK_READ))
        return -ENODEV;

```

```

/*为 card 分配 mmc_blk_data 结构体空间*/
md = mmc_blk_alloc(card);
if (IS_ERR(md))
    return PTR_ERR(md);
/*设置块的大小*/
err = mmc_blk_set_blksize(md, card);
if (err)
    goto out;

string_get_size((u64)get_capacity(md->disk) << 9, STRING_UNITS_2,
                cap_str, sizeof(cap_str));
printk(KERN_INFO "%s: %s %s %s %s\n",
       md->disk->disk_name, mmc_card_id(card), mmc_card_name(card),
       cap_str, md->read_only ? "(ro)" : "");
/*将 md 设置为 card 的 driver_data 字段*/
mmc_set_drvdata(card, md);
/*把 mmc 包含的信息向系统进行注册，注册成功后就可以在文件系统对应目录下找到
mmc_card 对应的结点设备*/
add_disk(md->disk);
return 0;

out:
mmc_blk_put(md);

return err;
}

```

#### 4. 驱动的入口函数mmc\_blk\_init()

加载驱动时该函数被调用，该函数向内核申请注册一个块设备，然后进入核心层进行注册。

```

static int __init mmc_blk_init(void)
{
    int res;
    /*向内核申请注册一个块设备*/
    res = register_blkdev(MMC_BLOCK_MAJOR, "mmc");
    if (res)
        goto out;
    /*进入核心层进行注册*/
    res = mmc_register_driver(&mmc_driver);
    if (res)
        goto out2;

    return 0;
out2:
    unregister_blkdev(MMC_BLOCK_MAJOR, "mmc");
out:
    return res;
}

```

#### 5. 为块设备分配空间函数mmc\_blk\_alloc()

函数 mmc\_blk\_alloc() 为块设备分配空间，并初始化一个请求队列，设置设备队列的 sector 大小。

```

static struct mmc_blk_data *mmc_blk_alloc(struct mmc_card *card)
{
    struct mmc_blk_data *md;
    int devidx, ret;
    /*在内存中查找第一个被清理过的 bit*/
    devidx = find_first_zero_bit(dev_use, MMC_NUM_MINORS);
    if (devidx >= MMC_NUM_MINORS)
        return ERR_PTR(-ENOSPC);
    /*从地址 dev_use 开始设置 bit, 设置为 devidx*/
    set_bit(devidx, dev_use);
    /*分配结构体 mmc_blk_data 空间并初始化*/
    md = kzalloc(sizeof(struct mmc_blk_data), GFP_KERNEL);
    if (!md) {
        ret = -ENOMEM;
        goto out;
    }
    /*设置卡的状态为只读*/
    md->read_only = mmc_blk_READONLY(card);
    /*分配设备的次设备号为 8*/
    md->disk = alloc_disk(1 << MMC_SHIFT);
    if (md->disk == NULL) {
        ret = -ENOMEM;
        goto err_kfree;
    }
    spin_lock_init(&md->lock);
    md->usage = 1;
    /*初始化一个请求队列, 并将该队列与卡关联*/
    ret = mmc_init_queue(&md->queue, card, &md->lock);
    if (ret)
        goto err_putdisk;
    /*注册 mmc_blk_issue_rq 到 md->queue, 当 md->queue 上有 request 待处理时,
     mmc_blk_issue_rq 就会被调用*/
    md->queue.issue_fn = mmc_blk_issue_rq;
    md->queue.data = md;
    /*注册相关的 mmc_blk_data 包含的块设备区*/
    md->disk->major = MMC_BLOCK_MAJOR;
    md->disk->first_minor = devidx << MMC_SHIFT;
    md->disk->fops = &mmc_bdops;
    md->disk->private_data = md;
    md->disk->queue = md->queue.queue;
    md->disk->driverfs_dev = &card->dev;
    sprintf(md->disk->disk_name, "mmcblk%d", devidx);
    /*设置传输 sector 大小*/
    blk_queue_hardsect_size(md->queue.queue, 512);
    /*根据卡的类型设置容量*/
    if (!mmc_card_sd(card) && mmc_card_blockaddr(card)) {
        set_capacity(md->disk, card->ext_csd.sectors);
    } else {
        set_capacity(md->disk,
                    card->csd.capacity << (card->csd.read_blkbits - 9));
    }
    return md;
err_putdisk:
    put_disk(md->disk);
}

```

```

err_kfree:
    kfree(md);
out:
    return ERR_PTR(ret);
}

```

在该驱动部分还包括一些对块操作的函数，如 mmc\_blk\_open()、mmc\_blk\_get()、mmc\_blk\_put()、mmc\_blk\_release()和mmc\_blk\_getgeo()等。

## 11.3 SD 卡移植步骤

SD 卡的驱动程序已经包含在内核中，只需要在编译内核时配置对 SD 卡驱动支持。这里使用的开发板为 mini2440，其对应的初始化文件为 mach-mini2440.c。

### 11.3.1 添加延时和中断

文件 mach-mini2440.c 可以参考 mach-smdk2440.c 进行修改，这里不对该文件的修改进行介绍，读者可以下载开发板公司修改好的压缩文件。另外，延时和中断也在附带的代码中添加，添加延时在 s3cmci.c 文件中。

#### 1. 添加延时

修改 drivers/mmc/host/s3cmci.c 文件，在底层函数 pio\_tasklet() 中添加延时。

```

#include <linux/delay.h>
static void pio_tasklet(unsigned long data)
{
    struct s3cmci_host *host = (struct s3cmci_host *) data;
    disable_irq(host->irq);
    udelay(50);
}

```

#### 2. 添加中断设置

在函数 static int \_\_devinit s3cmci\_probe(struct platform\_device \*pdev, int is2440) 中添加中断设置。粗体部分为添加部分。

```

host->irq_cd = s3c2410_gpio_getirq(host->pdata->gpio_detect);
host->irq_cd = IRQ_EINT16;
s3c2410_gpio_cfgpin(S3C2410_GPG8, S3C2410_GPG8_EINT16);

```

### 11.3.2 配置内核

在编译内核前需要对内核进行配置，让内核支持 SD 卡的访问。使用 make menuconfig 命令进入窗口配置界面，进入 Device Drivers 配置界面配置 MMC/SD/SDIO card support，如图 11.4 所示。

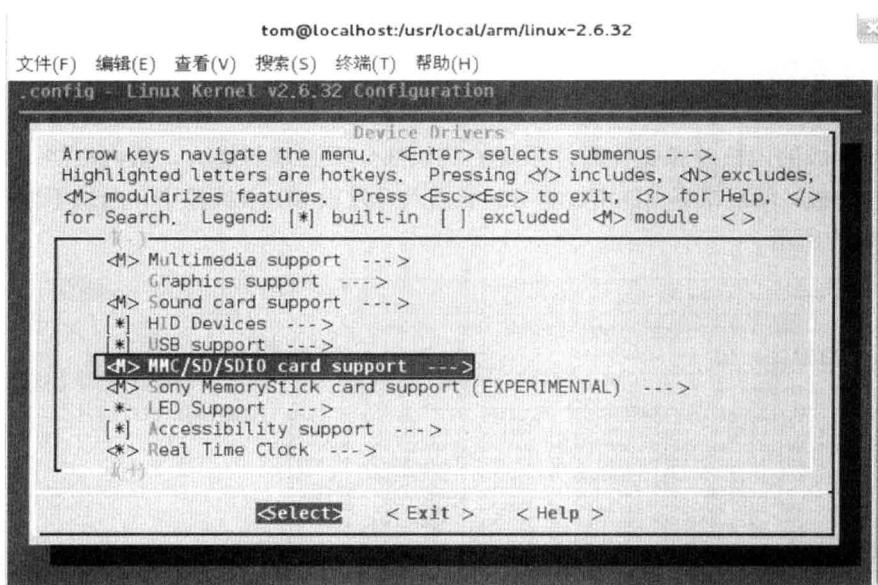


图 11.4 配置 MMC/SD/SDIO card support

选择对平台的支持。进入 MMC/SD/SDIO card support 配置窗口选择配置 Samsung S3C SD/MMC Card Interface support，如图 11.5 所示。选择该项后，内核支持 S3C 系列的 SD 卡驱动。

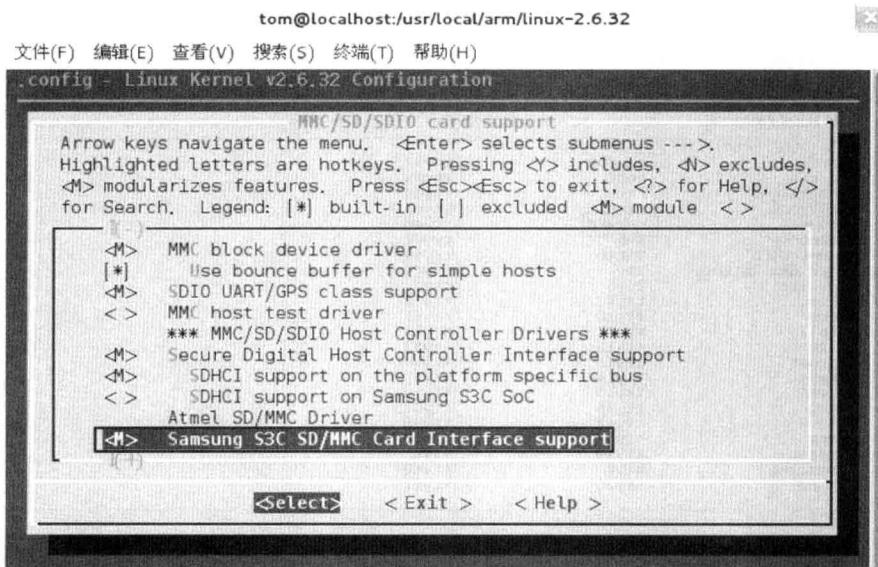


图 11.5 配置 Samsung S3C SD/MMC Card Interface support

保存配置后，使用下面的编译命令编译新内核映像文件，命令如下：

```
#make clean
#make zImage
```

### 11.3.3 烧写新内核

烧写新生成的内核映像文件时，系统的其他部分不必重新烧写。新内核烧写完成后，重新启动开发板，然后准备 SD 卡。将 SD 卡插入开发板 SD 卡插槽中，出现下面的提示信息，如图 11.6 所示。

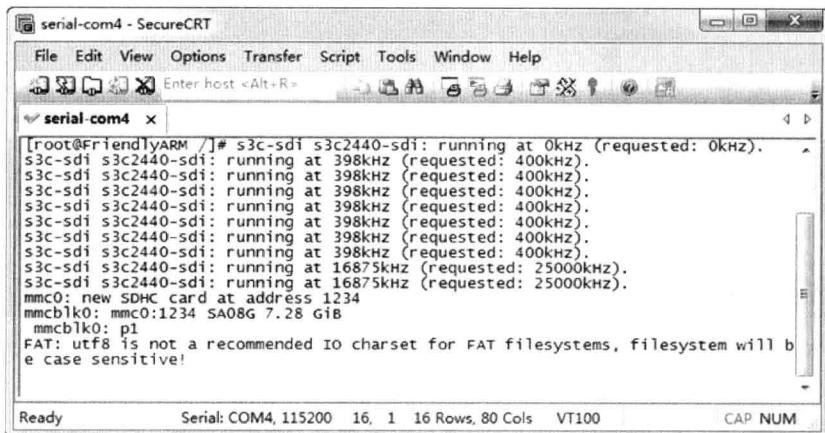


图 11.6 识别到 SD 卡信息

使用 `ls /dev` 命令可以查看在目录下多了设备结点 `sdcard`，如图 11.7 所示。

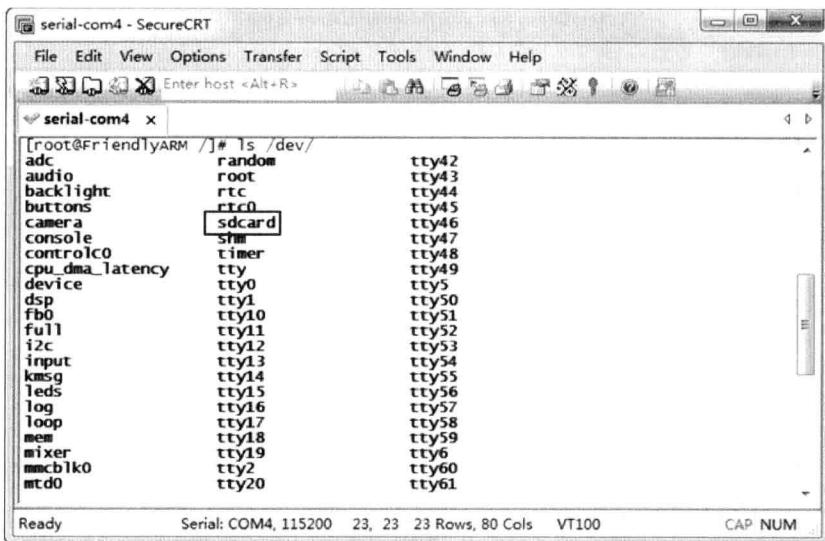


图 11.7 `sdcard` 结点

挂载该设备结点后，可以查看该设备内的信息，挂载前新建目录/`mnt/sdcard`，挂载命令和结果如下：

```
# mkdir /mnt/sdcard
# mount /dev/sdcard /mnt/sdcard
```

```
# ls /mnt/sdcard/
@samsung.ess      My Music           audio_play_list.txt
Audio             Other files        bootex.log
Ebook             Photos            software
Images            Sounds           Videos
```

## 11.4 小 结

本章重点为 SD 卡协议介绍和 SD 卡驱动分析, 本章后面还介绍了 SD 卡驱动移植过程。随着 SD 卡存储容量增加和价格下降, 其应用越来越广泛, SD 卡驱动在嵌入式系统中也将受到关注。Linux 内核已经对 SD 卡驱动进行了支持, 移植到其他平台时只需要做少量的修改即可。

# 第 12 章 NandFlash 驱动移植

在很多章节中都会涉及 NandFlash 的相关知识，比如 U-boot 中涉及对 NandFlash 的支持、在文件系统中涉及 NandFlash 支持的文件系统，因此本章在简单介绍 NandFlash 工作原理后，直接介绍其移植方法。NandFlash 相比 NorFlash 有很多优势，但需要驱动支持，本章将针对 Mini2440 讲解 NandFlash 的驱动移植过程。

## 12.1 NandFlash 介绍

对 NandFlash 存储芯片进行操作，必须通过 NandFlash 控制器才能完成，而不能通过对 NandFlash 进行总线操作。对于 NandFlash 的写操作只能以块方式进行写，对于 NandFlash 的读操作可以按字节进行读。

### 12.1.1 NandFlash 命令介绍

NandFlash 命令的执行过程是通过将命令发送到 NandFlash 控制器的命令寄存器来执行的。其命令的执行是分周期的，每条命令有一个或多个执行周期，每个执行周期有相应代码表示该周期将要执行的动作。NandFlash 命令主要包括 Read1、Read2、Read ID、Reset、Page Program、Block Erase、Read Status 等。

#### 1. Read1

- 命令功能：表示将要读取 NandFlash 存储空间中一页的前半部分，且将内置指针定位到前半部分的第一个字节。
- 命令代码：00h。

#### 2. Read2

- 命令功能：表示将要读取 NandFlash 存储空间中一页的后半部分，且将内置指针定位到后半部分的第一个字节。
- 命令代码：01h。

#### 3. Read ID

- 命令功能：表示读取 NandFlash 芯片的 ID 号。
- 命令代码：90h。

#### 4. Reset

- 命令功能：表示重新启动 NandFlash 芯片。
- 命令代码：FFh。

#### 5. Page Program

- 命令功能：表示对页进行编程，用于对 NandFlash 的写操作。
- 命令代码：首先写入 00h（A 区）/01h（B 区）/05h（C 区），该代码表示目标区；再写入 80h 开始编程模式，即写入模式；接着写入地址和数据；最后写入 10h 表示编程结束。

#### 6. Block Erase

- 命令功能：表示对块擦除操作。
- 命令代码：首先写入 60h 进入擦写模式；再输入块地址，即将要擦除的块；接着写入 D0h 表示擦写结束。

#### 7. Read Status

- 命令功能：表示读取内部状态寄存器值的命令。
- 命令代码：70h。

### 12.1.2 NandFlash 控制器

对于 2440 的 NandFlash 控制器中，寄存器有以下 12 种。与 2410 相比寄存器的设置有些变化，具体寄存器中每个 bit 的设置可以参考 2440 文档。在 S3C2440 芯片手册文档的第 6 章专门介绍 NandFlash 控制器时，对以下每个寄存器的设置做了详细说明。

- 配置寄存器（NFCNF）；
- 控制寄存器（NFCONT）；
- 命令寄存器（NFACMD）；
- 地址寄存器（NFADDR）；
- 数据寄存器（NFDATA）；
- 状态寄存器（NFSTAT）；
- 主数据区域 ECC 寄存器（NFMECCD0/1）；
- 空闲区域 ECC 寄存器（NFSECCD）；
- ECC0/1 状态寄存器（NFESTAT0/1）；
- 主数据区域 ECC 状态寄存器（NFMECC）；
- 空闲区域 ECC 状态寄存器（NFSECC）；
- 块地址寄存器（NFSBLK&NFEBLK）。

了解了 NandFlash 的命令和寄存器后，接下来将介绍 NandFlash 驱动，然后介绍如何修改内核驱动使之适合 2440。在对寄存器操作时，如果有不清楚的地方可以参考 2440 文档，查看对应寄存器各个位的设置情况。

## 12.2 NandFlash 驱动介绍

在 Linux 内核中已经提供了 NandFlash 驱动，驱动的声明在内核的 include/linux/mtd/nand.h 文件中，在 include/linux/mtd/nand\_ecc.h 中还声明了 ECC 算法。驱动的相关实现部分主要在对应的 nand\_base.c、nand\_bbt.c 和 nand\_ecc.c 中。下面介绍 NandFlash 驱动的主要部分。

### 12.2.1 Nand 芯片结构

结构体 nand\_chip 中声明了 Nand 芯片的各种读写接口函数、buffer 操作函数、对芯片状态检查、对坏块检查和标记、芯片的属性等，下面为该结构体的定义。

```
struct nand_chip {
    void __iomem *IO_ADDR_R; /*读地址*/
    void __iomem *IO_ADDR_W; /*写地址*/
    /*对字节的操作函数声明*/
    uint8_t (*read_byte)(struct mtd_info *mtd); /*读一个字节*/
    u16   (*read_word)(struct mtd_info *mtd); /*写一个字节*/
    /*buffer操作*/
    void  (*write_buf)(struct mtd_info *mtd, const uint8_t *buf, int len);
    void  (*read_buf)(struct mtd_info *mtd, uint8_t *buf, int len);
    int   (*verify_buf)(struct mtd_info *mtd, const uint8_t *buf, int len);
    /*选择一个芯片的操作*/
    void  (*select_chip)(struct mtd_info *mtd, int chip);
    /*坏块检查操作*/
    int   (*block_bad)(struct mtd_info *mtd, loff_t ofs, int getchip);
    /*坏块标记操作*/
    int   (*block_markbad)(struct mtd_info *mtd, loff_t ofs);
    /*命令控制操作*/
    void  (*cmd_ctrl)(struct mtd_info *mtd, int dat, unsigned int ctrl);
    /*设备准备操作*/
    int   (*dev_ready)(struct mtd_info *mtd);
    /*发送命令操作*/
    void  (*cmdfunc)(struct mtd_info *mtd, unsigned command, int column,
                     int page_addr);
    /*等待命令完成操作*/
    int   (*waitfunc)(struct mtd_info *mtd, struct nand_chip *this);
    /*擦除操作*/
    void  (*erase_cmd)(struct mtd_info *mtd, int page);
    /*检查坏块表*/
    int   (*scan_bbt)(struct mtd_info *mtd);
    /*进行附加错误状态检查操作*/
    int   (*errstat)(struct mtd_info *mtd, struct nand_chip *this, int
                     state, int status, int page);
    /*按页进行写操作*/
    int   (*write_page)(struct mtd_info *mtd, struct nand_chip *chip,
```

```

        const uint8_t *buf, int page, int cached, int raw);
int      chip_delay;           /*芯片延迟*/
unsigned int options;          /*芯片专有选项*/
int      page_shift;           /*页右移的位数, 即 column 地址位数*/
int      phys_erase_shift;     /*物理擦写块的地址位数*/
int      bbt_erase_shift;      /*坏块表入口的地址位数*/
int      chip_shift;           /*该芯片的地址位数*/
int      numchips;             /*芯片个数*/
uint64_t   chipsize;           /*多个芯片组中, 一个芯片的大小*/
int      pagemask;             /*每个芯片页数的屏蔽字, 通过它取出每个芯片包含多少个页*/
int      pagebuf;              /*在页缓冲区中的页号*/
int      subpagesize;           /*拥有的子页大小*/
uint8_t   cellinfo;             /*MLC 多芯片数据*/
int      badblockpos;           /*坏块标记位置*/
nand_state_t state;            /*芯片状态*/
uint8_t   *oob_poi;             /*缓冲区位置*/
struct nand_hw_control *controller; /*硬件控制器结构体指针*/
struct nand_ecclayout *ecclayout;  /*默认的 ecc 设置方案*/
struct nand_ecc_ctrl ecc;         /*ecc 控制结构体*/
struct nand_buffers *buffers;     /*用于读写的缓冲区结构体*/
struct nand_hw_control hwcontrol; /*专用平台硬件控制结构体*/
struct mtd_oob_ops ops;          /*oob 操作数*/
uint8_t   *bbt;                 /*坏块表*/
struct nand_bbt_descr *bbt_td;    /*坏块表描述*/
struct nand_bbt_descr *bbt_md;    /*坏块表映像描述*/
struct nand_bbt_descr *badblock_pattern; /*坏块检测模板*/
void      *priv;                /*私有数据结构*/
};


```

## 12.2.2 NandFlash 驱动分析

在分析驱动时, 是从普通的 Nand 驱动开始分析, 到识别平台信息, 然后到具体平台接口函数调用, 再到对芯片寄存器读写的过程。首先看 Nand 通用驱动文件, 为目录 drivers/mtd/nand 下的 plat\_nand.c。

### 1. 探针函数 plat\_nand\_probe()

当系统检查到 Nand 设备时就会调用该函数, 在 plat\_nand\_probe() 函数中将参数为 platform\_device 结构体的 pdev 数据 dev.platform\_data, 赋给了结构体 platform\_nand\_data, 然后通过函数 platform\_set\_drvdata() 把信息保存在 driver\_data 字段中。

```

static int __init plat_nand_probe(struct platform_device *pdev)
{
    /*将参数为 platform_device 结构体的 pdev 数据 dev.platform_data, 赋给了结构体
     *platform_nand_data*/
    struct platform_nand_data *pdata = pdev->dev.platform_data;
    struct plat_nand_data *data; .
    int res = 0;
    /*将一个 I/O 地址空间映射到内核的虚拟地址空间上, 便于访问*/

```

```

data = kzalloc(sizeof(struct plat_nand_data), GFP_KERNEL);
if (!data) {
    dev_err(&pdev->dev, "failed to allocate device structure.\n");
    return -ENOMEM;
}

data->io_base = ioremap(pdev->resource[0].start,
                      pdev->resource[0].end - pdev->resource[0].start + 1);
if (data->io_base == NULL) {
    dev_err(&pdev->dev, "ioremap failed\n");
    kfree(data);
    return -EIO;
}
/*对结构体 plat_nand_data 的各种数据、状态、命令、地址进行初始化*/
data->chip.priv = &data;
data->mtd.priv = &data->chip;
data->mtd.owner = THIS_MODULE;
data->mtd.name = dev_name(&pdev->dev);
data->chip.IO_ADDR_R = data->io_base;
data->chip.IO_ADDR_W = data->io_base;
data->chip.cmd_ctrl = pdata->ctrl.cmd_ctrl;
data->chip.dev_ready = pdata->ctrl.dev_ready;
data->chip.select_chip = pdata->ctrl.select_chip;
data->chip.chip_delay = pdata->chip.chip_delay;
data->chip.options |= pdata->chip.options;
data->chip.ecc.hwctl = pdata->ctrl.hwcontrol;
data->chip.ecc.layout = pdata->chip.ecclayout;
data->chip.ecc.mode = NAND_ECC_SOFT;
/*使用函数 platform_set_drvdata() 将信息保存在设备的 driver_data 字段中*/
platform_set_drvdata(pdev, data);
/* 处理特定平台设置 */
if (pdata->ctrl.probe) {
    res = pdata->ctrl.probe(pdev);
    if (res)
        goto out;
}
/*扫描是否存在 mtd 设备*/
if (nand_scan(&data->mtd, 1)) {
    res = -ENXIO;
    goto out;
}
#endif CONFIG_MTD_PARTITIONS
if (pdata->chip.part_probe_types) {
    res = parse_mtd_partitions(&data->mtd,
                               pdata->chip.part_probe_types,
                               &data->parts, 0);
    if (res > 0) {
        add_mtd_partitions(&data->mtd, data->parts, res);
        return 0;
    }
}
if (pdata->chip.partitions) {
    data->parts = pdata->chip.partitions;
    res = add_mtd_partitions(&data->mtd, data->parts,
                            pdata->chip.nr_partitions);
} else
#endif /*添加 mtd 设备*/
res = add_mtd_device(&data->mtd);
if (!res)

```

```

        return res;
    /*释放 nand 设备占用的资源*/
    nand_release(&data->mtd);
out:
    /*资源释放部分：释放设备占用资源、释放 I/O 映射到内存的空间、释放 data 占用空间*/
    platform_set_drvdata(pdev, NULL);
    iounmap(data->io_base);
    kfree(data);
    return res;
}

```

## 2. platform\_nand\_data 结构体

platform\_nand\_data 结构体在 include/linux/mtd/nand.h 中进行的定义。该结构体包括特定平台的数据信息，在该结构体中包括两项内容，即特定芯片结构体和设备控制器结构体。下面为这 3 个结构体的定义。

```

/*platform_nand_data 结构体的定义*/
struct platform_nand_data {
    struct platform_nand_chip    chip;
    struct platform_nand_ctrl    ctrl;
};

/*platform_nand_chip 结构体定义特定芯片的属性*/
struct platform_nand_chip {
    int          nr_chips;
    int          chip_offset;
    int          nr_partitions;
    struct mtd_partition *partitions;
    struct nand_ecclayout *ecclayout;
    int          chip_delay;
    unsigned int   options;
    const char    **part_probe_types;
    void         *priv;
};

/*platform_nand_ctrl 结构体中定义了对芯片控制的操作函数*/
struct platform_nand_ctrl {
    void        (*hwcontrol)(struct mtd_info *mtd, int cmd);
    int         (*dev_ready)(struct mtd_info *mtd);
    void        (*select_chip)(struct mtd_info *mtd, int chip);
    void        (*cmd_ctrl)(struct mtd_info *mtd, int dat,
                           unsigned int ctrl);
    void        *priv;
};

```

## 3. 驱动初始化 s3c24xx\_nand\_init()

加载该驱动后，就会注册 s3c2440\_nand\_driver 驱动。因为 Mini2440 的 CPU 类型为 S3C2440，因此在函数调用 s3c2410\_nand\_init 时，注册支持 Mini2440 的 NandFlash 驱动。

```

module_init(s3c2410_nand_init);
static int __init s3c2410_nand_init(void)
{
    printk("S3C24XX NAND Driver, (c) 2004 Simtec Electronics\n");

    return platform_driver_register(&s3c24xx_nand_driver);
}

```

#### 4. 探针函数s3c24xx\_nand\_probe()

因为调用函数 platform\_driver\_register (&s3c2440\_nand\_driver) 后，调用探针函数时，函数 s3c24xx\_nand\_probe 就会被调用。

```
/*函数 s3c24xx_nand_probe() 的定义*/
static int s3c24xx_nand_probe(struct platform_device *pdev)
{
    struct s3c2410_platform_nand *plat = to_nand_plat(pdev);
    enum s3c_cpu_type cpu_type;
    struct s3c2410_nand_info *info;
    struct s3c2410_nand_mtd *nmtd;
    struct s3c2410_nand_set *sets;
    struct resource *res;
    int err = 0;
    int size;
    int nr_sets;
    int setno;

    cpu_type = platform_get_device_id(pdev)->driver_data;

    pr_debug("s3c2410_nand_probe(%p)\n", pdev);
    /*分配空间并进行初始化*/
    info = kmalloc(sizeof(*info), GFP_KERNEL);
    if (info == NULL) {
        dev_err(&pdev->dev, "no memory for flash info\n");
        err = -ENOMEM;
        goto exit_error;
    }
    memset(info, 0, sizeof(*info));
    /*将 info 保存在 driver_data 字段中*/
    platform_set_drvdata(pdev, info);
    spin_lock_init(&info->controller.lock);
    /*初始化队列*/
    init_waitqueue_head(&info->controller.wq);
    /*获得时钟资源并开启*/
    info->clk = clk_get(&pdev->dev, "nand");
    if (IS_ERR(info->clk)) {
        dev_err(&pdev->dev, "failed to get clock\n");
        err = -ENOENT;
        goto exit_error;
    }
    clk_enable(info->clk);
    /*分配和映射资源*/
    res = pdev->resource;
    size = res->end - res->start + 1;
    /*为 NandFlash 寄存器区申请 I/O 内存地址空间区，并通过 ioremap() 把它映射到虚拟地址空间*/
    info->area = request_mem_region(res->start, size, pdev->name);
    if (info->area == NULL) {
```

```

    dev_err(&pdev->dev, "cannot reserve register region\n");
    err = -ENOENT;
    goto exit_error;
}
info->device      = &pdev->dev;
info->platform     = plat;
info->regs         = ioremap(res->start, size);
info->cpu_type     = cpu_type;
if (info->regs == NULL) {
    dev_err(&pdev->dev, "cannot reserve register region\n");
    err = -EIO;
    goto exit_error;
}

dev_dbg(&pdev->dev, "mapped registers at %p\n", info->regs);
/*初始化NandFlash控制器*/
err = s3c2410_nand_inithw(info);
if (err != 0)
    goto exit_error;

sets = (plat != NULL) ? plat->sets : NULL;
nr_sets = (plat != NULL) ? plat->nr_sets : 1;

info->mtd_count = nr_sets;
/*为mtd设备分配设备信息的存储空间*/
size = nr_sets * sizeof(*info->mtds);
info->mtds = kmalloc(size, GFP_KERNEL);
if (info->mtds == NULL) {
    dev_err(&pdev->dev, "failed to allocate mtd storage\n");
    err = -ENOMEM;
    goto exit_error;
}

memset(info->mtds, 0, size);
/*初始化所有可能的芯片*/
nmtd = info->mtds;

for (setno = 0; setno < nr_sets; setno++, nmtd++) {
    pr_debug("initialising set %d (%p, info %p)\n", setno, nmtd, info);

    s3c2410_nand_init_chip(info, nmtd, sets);

    nmtd->scan_res = nand_scan_ident(&nmtd->mtd,
                                       (sets) ? sets->nr_chips : 1);

    if (nmtd->scan_res == 0) {
        s3c2410_nand_update_chip(info, nmtd);
        nand_scan_tail(&nmtd->mtd);
        s3c2410_nand_add_partition(info, nmtd, sets);
    }

    if (sets != NULL)

```

```

        sets++;
    }
/*CPU 频率驱动*/
err = s3c2410_nand_cpufreq_register(info);
if (err < 0) {
    dev_err(&pdev->dev, "failed to init cpufreq support\n");
    goto exit_error;
}

if (allow_clk_stop(info)) {
    dev_info(&pdev->dev, "clock idle support enabled\n");
    clk_disable(info->clk);
}

pr_debug("initialised ok\n");
return 0;

exit_error:
s3c24xx_nand_remove(pdev);

if (err == 0)
    err = -EINVAL;
return err;
}

```

## 5. 支持的芯片类型s3c\_cpu\_type

在 s3c2410.c 中还定义了支持的芯片类型，枚举类型 s3c\_cpu\_typ 中定义了支持的芯片类型。

```

enum s3c_cpu_type {
    TYPE_S3C2410,
    TYPE_S3C2412,
    TYPE_S3C2440,
};

```

## 6. 控制器初始化s3c2410\_nand\_inithw()

在函数 s3c24xx\_nand\_probe() 中调用函数 s3c2410\_nand\_inithw()，对 NandFlash 控制器进行初始化。在对硬件初始化函数 s3c2410\_nand\_inithw() 中就涉及了对寄存器的访问。

```

static int s3c2410_nand_inithw(struct s3c2410_nand_info *info)
{
    int ret;

    ret = s3c2410_nand_setrate(info);
    if (ret < 0)
        return ret;
/*根据 CPU 型号写不同的配置*/
    switch (info->cpu_type) {
        case TYPE_S3C2410:

```

```

default:
    break;

case TYPE_S3C2440:
case TYPE_S3C2412:
    /*以四字节形式向配置寄存器中写内容。S3C2440_NFCONT_ENABLE 的值为 1，即向
     *置寄存器 NFCONT 中写 1，通过查看芯片资料，对应 bit0 为 1 时表示使能 NandFlash
     *控制，其初始状态为 0*/
    writel(S3C2440_NFCONT_ENABLE, info->regs + S3C2440_NFCONT);
}

return 0;
}

```

对于 Nand 驱动的其他函数和操作也可以用相同的方法进行分析。在对驱动源码进行分析和理解的基础上，下面将介绍如何针对具体芯片进行移植。

## 12.3 NandFlash 驱动移植

移植 NandFlash 驱动时只要对内核代码做少量修改即可，修改的内容主要包括对 NandFlash 类型的支持、NandFlash 分区和 NandFlash 注册。

### 12.3.1 内核的修改

#### 1. NandFlash类型的 support

文件 nand\_ids.c 中的数组 nand\_flash\_ids 中定义了内核支持的各种不同容量的 NandFlash 芯片，包括分页大小、容量大小等信息。

```

struct nand_flash_dev nand_flash_ids[] = {
    /*省略部分代码*/
/*512 Megabit*/
{ "NAND 64MiB 1,8V 8-bit", 0xA2, 0, 64, 0, LP_OPTIONS },
{ "NAND 64MiB 3,3V 8-bit", 0xF2, 0, 64, 0, LP_OPTIONS },
{ "NAND 64MiB 1,8V 16-bit", 0xB2, 0, 64, 0, LP_OPTIONS16 },
{ "NAND 64MiB 3,3V 16-bit", 0xC2, 0, 64, 0, LP_OPTIONS16 },

/*1 Gigabit*/
{ "NAND 128MiB 1,8V 8-bit", 0xA1, 0, 128, 0, LP_OPTIONS },
{ "NAND 128MiB 3,3V 8-bit", 0xF1, 0, 128, 0, LP_OPTIONS },
{ "NAND 128MiB 1,8V 16-bit", 0xB1, 0, 128, 0, LP_OPTIONS16 },
{ "NAND 128MiB 3,3V 16-bit", 0xC1, 0, 128, 0, LP_OPTIONS16 },

/*2 Gigabit*/
{ "NAND 256MiB 1,8V 8-bit", 0xAA, 0, 256, 0, LP_OPTIONS },
{ "NAND 256MiB 3,3V 8-bit", 0xDA, 0, 256, 0, LP_OPTIONS },
{ "NAND 256MiB 1,8V 16-bit", 0xBA, 0, 256, 0, LP_OPTIONS16 },

```

```

    {"NAND 256MiB 3,3V 16-bit", 0xCA, 0, 256, 0, LP_OPTIONS16},

/*4 Gigabit*/
{"NAND 512MiB 1,8V 8-bit", 0xAC, 0, 512, 0, LP_OPTIONS},
 {"NAND 512MiB 3,3V 8-bit", 0xDC, 0, 512, 0, LP_OPTIONS},
 {"NAND 512MiB 1,8V 16-bit", 0xBC, 0, 512, 0, LP_OPTIONS16},
 {"NAND 512MiB 3,3V 16-bit", 0xCC, 0, 512, 0, LP_OPTIONS16},

/*8 Gigabit*/
 {"NAND 1GiB 1,8V 8-bit", 0xA3, 0, 1024, 0, LP_OPTIONS},
 {"NAND 1GiB 3,3V 8-bit", 0xD3, 0, 1024, 0, LP_OPTIONS},
 {"NAND 1GiB 1,8V 16-bit", 0xB3, 0, 1024, 0, LP_OPTIONS16},
 {"NAND 1GiB 3,3V 16-bit", 0xC3, 0, 1024, 0, LP_OPTIONS16},

/*16 Gigabit*/
 {"NAND 2GiB 1,8V 8-bit", 0xA5, 0, 2048, 0, LP_OPTIONS},
 {"NAND 2GiB 3,3V 8-bit", 0xD5, 0, 2048, 0, LP_OPTIONS},
 {"NAND 2GiB 1,8V 16-bit", 0xB5, 0, 2048, 0, LP_OPTIONS16},
 {"NAND 2GiB 3,3V 16-bit", 0xC5, 0, 2048, 0, LP_OPTIONS16},
 | /*省略部分代码*/
};


```

## 2. NandFlash分区

在文件 mach-mini2440.c 中修改 NandFlash 分区表。文件 mach-mini2440.c 可以参考 mach-smdk2440.c 进行修改。

```

static struct mtd_partition mini2440_default_nand_part[] = {
[0] = {
        .name    = "supervivi",           /*bootloader 所在分区*/,
        .size    = 0x00030000,
        .offset  = 0,
},
[1] = {
        .name    = "Kernel",            /*内核所在分区*/
        .offset  = 0x00050000,
        .size    = 0x00200000,
},
[2] = {
        .name    = "root",              /*文件系统所在分区*/
        .offset  = 0x00250000,
        .size    = 0x03dac000,
}
};


```

## 3. NandFlash注册

将 NandFlash 设备注册到系统中，在 \_\_initdata 中添加 NandFlash 设备。

/\*开发板上所有 NandFlash 的设置表，mini2440 上只有一块 NandFlash，如果有多块在后面继续添加\*/

```

static struct s3c2410_nand_set mini2440_nand_sets[] = {
[0] = {
        .name        = "NAND",
        .nr_chips   = 1,
        .nr_partitions = ARRAY_SIZE(mini2440_default_nand_part),
}


```

```

        .partitions = mini2440_default_nand_part,
    },
};

/*NandFlash 信息*/
static struct s3c2410_platform_nand mini2440_nand_info = {
    .tacls      = 20,
    .twrph0     = 60,
    .twrph1     = 20,
    .nr_sets   = ARRAY_SIZE(mini2440_nand_sets),
    .sets      = mini2440_nand_sets,
};

/*将 NandFlash 设备注册到系统中*/
static struct platform_device *mini2440_devices[] __initdata = {
    &s3c_device_usb,
    &s3c_device_rtc,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c0,
    &s3c_device_iis,
    &s3c_device_dm9k,
    &net_device_cs8900,
    &s3c24xx_udai34x,
    &s3c_device_sdi,
    &s3c_device_nand,
};

```

### 12.3.2 内核的配置和编译

在编译内核时，配置对 MTD 的支持选项 Memory Technology Device (MTD) support。进入该配置窗口后，选择 MTD 分区支持 MTD partitioning support，配置如图 12.1 所示。

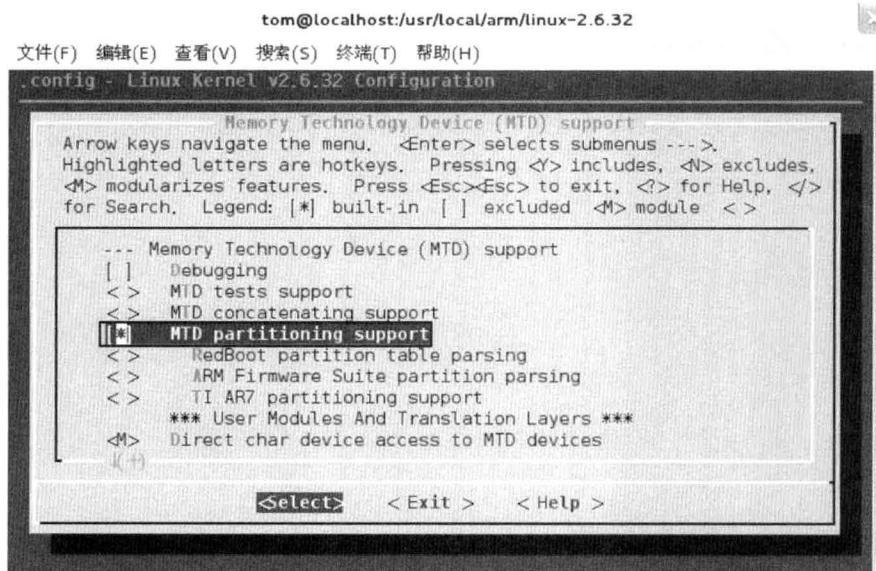


图 12.1 配置 MTD 分区支持

进入 NAND Device Support 配置窗口，选择对具体芯片类型的支持，配置如图 12.2 所示。

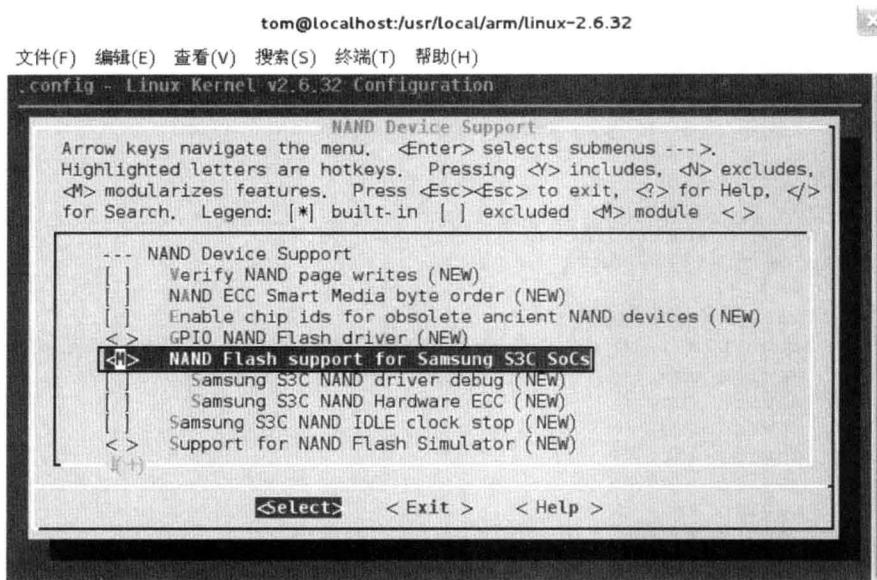


图 12.2 配置对 S3C2440 的支持

## 12.4 小结

NandFlash 驱动移植是比较简单的内容，对于其烧写内核部分没有介绍，读者可以参考前面的章节关于介绍内核编译、移植和烧写的过程。本章主要介绍分析 NandFlash 驱动的实现过程，达到让读者能根据不同型号的 NandFlash 修改对应驱动文件和移植的目的。

# 第4篇 系统移植高级篇

- ▶ 第 13 章 MiniGUI 与移植
- ▶ 第 14 章 Qt 开发与 Qtopia 移植
- ▶ 第 15 章 嵌入式数据库 Berkeley DB 移植
- ▶ 第 16 章 嵌入式数据库 SQLite 移植
- ▶ 第 17 章 嵌入式 Web 服务器 BOA 移植
- ▶ 第 18 章 嵌入式 Web 服务器 Thttpd 移植
- ▶ 第 19 章 JVM 及其移植
- ▶ 第 20 章 VoIP 技术与 Linphone 编译

# 第 13 章 MiniGUI 与移植

MiniGUI 是根据嵌入式系统应用特点量身定做的图形支持系统。MiniGUI 支持多种操作系统，如 uClinux、VxWorks、eCos、uC/OS-II、ThreadX、Nucleus、OSE 等。它也可以运行在 Win32 平台。本章主要介绍 MiniGUI 在上位机中的安装、开发和调试方法，以及移植到 ARM 平台的过程。

## 13.1 MiniGUI 在上位机中的安装

MiniGUI 在移植到开发板之前，必须在上位机中调试通过，然后交叉编译，最后移植到开发板上。下面将以 MiniGUI-3.0.12 在 Fedora release 19 版本上的安装过程为例，介绍 MiniGUI 的安装和编译过程。

### 13.1.1 安装需要的安装文件

在表 13.1 中列出了需要的安装文件。如果需要对各个文件进行更详细的了解，可以查看 MiniGUI 的手册文件。

表 13.1 安装文件和RPM包列表

安 装 文 件	说 明
libminogui-gpl-3.0.12.tar.gz	MiniGUI 的函数库源代码
jpegsrc.v7.tar.gz	MiniGUI 依赖函数库，用来支持 JPEG 图片
libpng-1.2.37.tar.gz	MiniGUI 依赖函数库，用来支持 PNG 图片
minogui-res-be-3.0.12.tar.gz	MiniGUI 所使用的资源文件，包括字体、图标、位图和鼠标光标等
qvfb2-2.0.tar.gz	MiniGUI 的图形引擎
mg-samples-3.0.12.tar.gz	MiniGUI 提供一个游戏范例 tball

- 安装文件的下载地址为：<http://www.minogui.org/en/download/>。

### 13.1.2 MiniGUI 的运行模式

MiniGUI 可以配置成 4 种模式中的一种：多进程的 MiniGUI-Processes、MiniGUI-Lite 运行模式、多线程的 MiniGUI-Threads 运行模式和非多线程的 MiniGUI-Standalone 运行模式。MiniGUI 不同的版本包括不同的 MiniGUI 运行模式。

- MiniGUI-Processes 运行模式：MiniGUI-Processes 上的每个程序都是单独的进程，

每个进程也可以建立多个窗口，并实现了多进程串口系统，来自不同进程的窗口可以在同一桌面上协调存在。

- MiniGUI-Lite 运行模式：MiniGUI-Lite 实现了不同窗口之间的切换，但没能够解决进程间窗口层叠的问题，无法同时管理来自不同进程间的窗口。在 MiniGUI 2.0 后被 MiniGUI-Processes 运行模式取代。
- MiniGUI-Threads 运行模式：MiniGUI-Threads 可以在不同的线程中建立多个窗口，所有的窗口在同一个进程中。
- MiniGUI-Standalone 运行模式：MiniGUI-Standalone 不需要多进程和多线程的支持，适合功能单一的应用场合。

MiniGUI 安装时 MiniGUI 运行模式的配置方法，如表 13.2 所示。

表 13.2 MiniGUI 运行模式的配置方法

Configure 脚本选项	宏	备注	默认
不指定	_MGRM_THREADS	MiniGUI-Threads 运行模式	
procs	_MGRM_PROCESSES _LITE_VERSION	MiniGUI 运行模式，仅用于 Linux/uClinux 操作系统	关闭
standalone	_MGRM_STANDALONE _LITE_VERSION _STAND_ALONE	MiniGUI-Standalone 运行模式，仅用于 Linux/uClinux 操作系统	关闭

目前嵌入式的芯片功能比较强大，各种嵌入式应用场合也越来越复杂，下面的运行模式配置将配置成 MiniGUI-Threads 运行模式，即直接执行 ./configure 即可。

### 13.1.3 编译并安装 MiniGUI

编译 MiniGUI 采用的内核为 Linux-3.10.7 的 Fedora release 19 版本；编译器的版本为 gcc-4.8.1；MiniGUI 的版本为 3.0.12。读者可以使用 uname -a 查看本机的内核版本，使用 cat /etc/issue 查看发行的 Linux 版本，使用 gcc -v 查看本机默认的编译器版本。

```
#uname -a          //查看内核版本
#cat /etc/issue    //查看 Linux 发行版本
#gcc -v            //查看 GCC 版本
```

#### 1. 安装 MiniGUI 的函数库（libminogui-gpl-3.0.12.tar.gz）

(1) 首先建立存放安装源文件目录，将需要安装的文件放在该目录下。

```
#mkdir /usr/local/minigui_setup //建立存放安装文件目录
```

(2) 完成安装文件的复制后，解压安装文件。解压方法如下：

```
#cd /usr/local/minigui_setup
#tar -zxvf libminogui-gpl-3.0.12.tar.gz //解压 libminogui 安装包
```

(3) 进入解压后的目录进行安装。

```
#cd libminogui-gpl-3.0.12
```

```
#./configure          //安装默认的配置，将其配置成 MiniGUI-Threads 运行模式
#make               //执行编译
#make install       //安装，默认安装在 /usr/local 目录下
```

通过 ls 命令可以查看安装的结果。

```
#ls /usr/local/lib           //查看安装的库文件
libminigui_ths-3.0.so.12    libminigui_ths.a   libminigui_ths.so
libminigui_ths-3.0.so.12.0.0 libminigui_ths.la  pkgconfig
#ls /usr/local/include/minigui/ //查看安装的头文件
common.h        mgconfig.h      psos_semaphore.h  win32_pthread.h
control.h       minigui.h      threadx_semaphore.h  win32_sched.h
ctrl            nucleus_pthread.h threadx_semaphore.h  win32_semaphore.h
customial.h     nucleus_semaphore.h ucos2_pthread.h  window.h
dti.c           ose_semaphore.h ucos2_semaphore.h  xvfb.h
endianrw.h     own_malloc.h   vxworks_pthread.h
fixedmath.h    own_stdio.h    vxworks_semaphore.h
gdi.h          psos_pthread.h win32 dirent.h
```

## 2. 安装MiniGUI的资源文件（minogui-res-be-3.0.12.tar.gz）

```
# tar -zxvf minogui-res-be-3.0.12.tar.gz      //解压资源文件包
# cd minogui-res-be-3.0.12                     //进入解压后的资源文件目录
# ./configure
# make
# make install                                //执行安装
```

**注意：**读者在安装的过程中应该以超级用户的身份进行安装，这里的安装过程都是以超级用户的身份执行的。

## 3. 添加共享库的搜索路径

在/etc/ld.so.conf 文件中包含了默认的共享库搜索路径。在该文件的后面添加刚才安装的 MiniGUI 的库文件路径。

```
#vi /etc/ld.so.conf
```

在文件 ld.so.conf 末尾添加 /usr/local/lib 和 /usr/lib。

```
#ldconfig          //刷新共享库缓存
```

## 4. 安装qt3-devel

```
# yum install qt3-devel          //安装支持 Qt 3.0.3 的头文件和库文件
```

**注意：**如果不安装此 rpm 包，在安装配置 QVFB 时会提示找不到相关的库文件和头文件。checking for Qt... configure: error: Qt (>= Qt 3.0.3) (headers and libraries) not found. Please check your installation!

## 5. 安装qvfb2

Qvfb2 是 Qt 提供的一个虚拟 FrameBuffer（帧缓存）工具。这个工具是基于 Qt 开发，

运行在 XWindow 上。如果安装的时候出现错误，请先安装 `jpegsrc.v7.tar.gz` 和 `libpng-1.2.37.tar.gz` 这两个软件。

```
#tar -zxvf qvfb2-2.0.tar.gz
#cd qvfb2-2.0
./configure \
--with-qt-includes=/usr/lib/qt-3.3/include \           //指定头文件目录
--with-qt-libraries=/usr/lib/qt-3.3/lib \             //指定库文件目录
--with-qt-dir=/usr/lib/qt-3.3                         //指定路径
#make
#make install
```

## 6. 修改配置文档

由于 MiniGUI 默认不使用 qvfb，所以需要修改配置文件，使得 qvfb2 生效。修改的配置文件为 `/usr/local/etc/MiniGUI.cfg`。修改的内容如下：

```
[pc_xvfb]
defaultmode=800x600-16bpp
window_caption=QVFB2-for-MiniGUI-3.0-Gtk-2
exec_file=/usr/local/bin/qvfb2
```

### 13.1.4 编译安装 MiniGUI 需要的图片支持库

安装支持 png 格式的文件 `libpng-1.2.37.tar.gz` 和支持 jpeg 格式的文件 `jpegsrc.v7.tar.gz`。为了运行范例程序，还需要安装高级图形库组件 `libmgplus-1.2.4.tar.gz`。

```
#tar zxvf jpegsrc.v7.tar.gz
#cd jpeg-v7
./configure
#make
#make install
# tar zxvf libpng-1.2.37.tar.gz
# cd libpng-1.2.37
./configure
# make -l /usr/lib/                      //指定编译库，否则会出现找不到 libpng.a 错误
#make install
#tar zxvf libmgplus-1.2.4.tar.gz
#cd libmgplus-1.2.4
./configure
#make
#make install
```

### 13.1.5 编译 MiniGUI 应用程序例子

解压例子程序包，通过例子让读者很快熟悉 MiniGUI 的编程格式。下面是这些例子的解压、编译和运行过程。

```
# tar -zxvf mg-samples-3.0.12.tar.gz
# cd mg-samples-3.0.12
# ./configure
# make
# make install
# cd same
# ./same
```

实例运行效果如图 13.1 所示。标题为 QVFB2-for-MiniGUI-3.0-Gtk-2 的窗口就是由 qvfb2 生成的窗口，Same 游戏运行在窗口中。



图 13.1 例子程序运行的结果

## 13.2 MiniGUI 的交叉编译和移植

MiniGUI 作为遵循 GPL 条款发布的自由软件，其目标是为基于 Linux 的实时嵌入式系统提供一个轻量级的图形用户界面支持系统。与 QT/Embedded、MicoroWindows 等其他 GUI 相比，MiniGUI 的最显著特点就是轻型、占用资源少。本节将介绍 MiniGUI 移植到 mini2440 上的过程。

### 13.2.1 交叉编译 MiniGUI

在 13.1 节中已经介绍了 MiniGUI 在 PC 上安装的过程，下载的软件包和相关的地址请

参考 13.1 节。交叉编译 MiniGUI 主要包括交叉编译 libminogui、安装 minigui-res 和交叉编译演示程序 mg-samples。

(1) 进行 MiniGUI 函数库的编译和安装。解压 libminogui-gpl-3.0.12.tar.gz 软件包，进入该目录，运行./configure 脚本。

```
#CC=arm-linux-gcc \
./configure --prefix=/usr/local/arm/4.4.3/arm-none-linux-gnueabi/ \
           //指定编译生成的库存放的路径
--build=i386-linux \
--host=arm-linux \
--target=arm-linux
```

生成定制的 Makefile 文件，然后可以继续执行 make 和 make install 命令编译，并安装 libminogui，安装成功后，MiniGUI 的函数库和头文件及配置文件等资源将被安装到 /usr/local/arm/4.4.3/arm-none-linux-gnueabi 目录中。函数库安装在 lib/子目录中；头文件安装在 include/子目录中；手册被装在 man/子目录中；配置文件被装在 etc/子目录中。

```
#make
#make install
```

(2) MiniGUI 资源的编译安装。解压 minigui-res-be-3.0.12.tar.gz，进入 minigui-res-be-3.0.12 目录。值得注意的是，在执行 make install 操作之前，修改目录中的 configure.linux 文件。打开 configure.linux 文件，prefix 选项部分的默认值为 \$(TOPDIR)/usr/local，需要将这里修改为 prefix=\$(TOPDIR)/usr/local/arm/4.4.3/arm-none-linux-gnueabi/，这里读者可修改为本机的交叉编译器路径，这样执行 make install 操作之后，安装脚本会自动把 MiniGUI 资源文件安装到 /usr/local/arm/4.4.3/arm-none-linux-gnueabi/lib/minogui/res/ 目录下。

(3) 交叉编译 MiniGUI 的演示程序。交叉编译 MiniGUI 的演示程序。解压 mg-samples-str-1.6.10.tar.gz，进入 mg-samples-str-1.6.10 目录执行脚本：

```
#CC=arm-linux-gcc           //指定编译器为交叉编译器 arm-linux-gcc
CFLAGS=-I/usr/local/arm/4.4.3/arm-none-linux-gnueabi/include
                           //指定依赖的头文件路径
LDFLAGS=-L/usr/local/arm/4.4.3/arm-none-linux-gnueabi/lib
          //指定依赖的库文件路径
./configure
--build=i386-linux
--host=arm-linux
--target=arm-linux
```

执行上述命令生成 Makefile 文件，继续执行 make 操作，完成演示程序的编译。

```
#make
```

编译完成后，可以使用 readelf 查看生成的演示程序。进入 src 子目录，查看文件头信息。

```
#readelf -h combobox
```

正确编译后，Machine 字段为 ARM，其文件头信息如下：

```
ELF Header:
 Magic: 7f 45 4c 46 01 01 00 00 00 00 00 00 00 00 00 00
```

```

Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: ARM
Version: 0x1
Entry point address: 0x87dc
Start of program headers: 52 (bytes into file)
Start of section headers: 5672 (bytes into file)
Flags: 0x5000002, has entry point, Version5 EABI
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 8
Size of section headers: 40 (bytes)
Number of section headers: 30
Section header string table index: 27

```

### 13.2.2 移植 MiniGUI 程序

移植 MiniGUI 程序前，文件系统加入 LCD 驱动，才能在 LCD 上查看 MiniGUI 运行结果。另外，内核也需要添加对图形引擎的支持。

(1) 对内核的配置。选择 Device Drivers ---> Graphics support ---> Support for frame buffer devices 配置对图形引擎的支持，如图 13.2 所示。选择 Device Drivers ---> Graphics support ---> Console display driver support ---> Framebuffer Console support 支持控制台帧缓存，如图 13.3 所示。

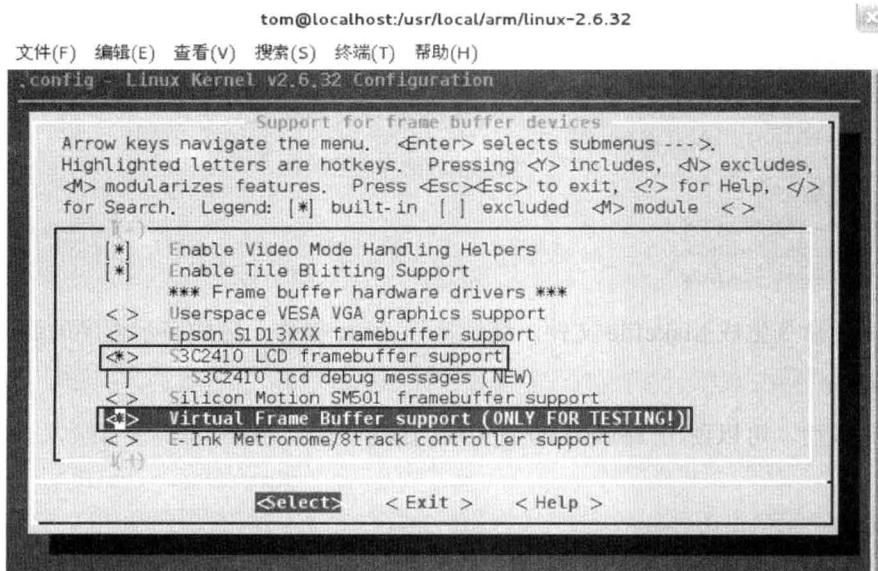


图 13.2 配置内核对图形引擎的支持

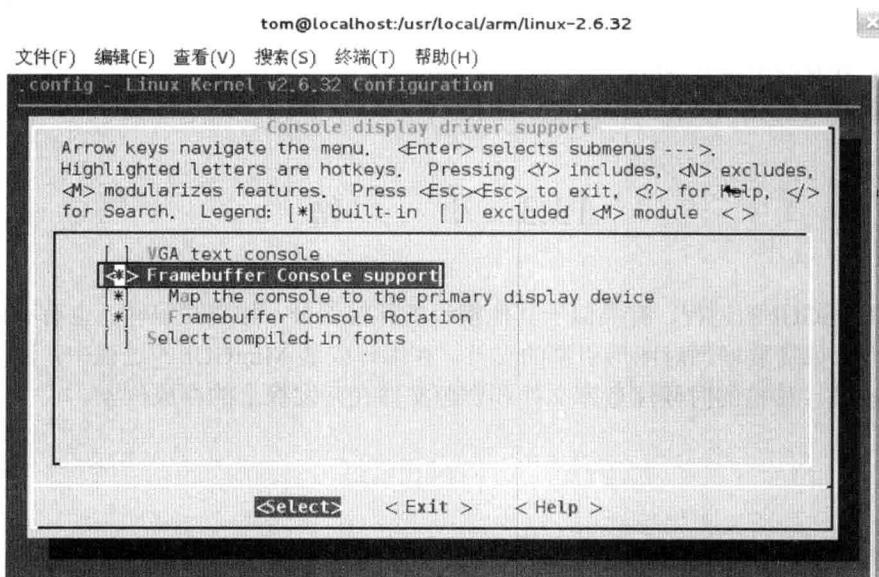


图 13.3 配置支持控制台帧缓存

(2) 加入对 LCD 的驱动，这部分内容在后面将会详细介绍。这里使用开发板供应商提供的文件系统支持 LCD 驱动。

(3) 下载 MiniGUI 配置文件到开发板。将配置文件 MiniGUI.cfg 复制到 mini2440 开发板文件系统的/usr/local/etc/目录下。对配置文件进行如下修改：

```
vi /usr/local/arm/4.3.2/arm-none-linux-gnueabi/etc/MiniGUI.cfg
[system]
# GAL engine and default options
gal_engine=fbcon
defaultmode=480x272-16bpp

# IAL engine
ial_engine=console
mdev=/dev/input/mice
mtype=IMPS2

[fbcon]
defaultmode=480x272-16bpp

[qvfb]
defaultmode=480x272-16bpp
display=0
```

(4) 将 /usr/local/arm/4.4.3/arm-none-linux-gnueabi/lib 目录下的下列库文件下载到 /usr/local/lib 目录下。/usr/local/lib 目录下的库文件包括：

```
libmgext-1.6.so.10 libmgext.so libminigui.la libvcongui-1.6.so.10.0.0
libmgext-1.6.so.10.0.0 libminigui-1.6.so.10 libminigui.so libvcongui.a
libmgext.a libminigui-1.6.so.10.0.0 libsupc++.a libvcongui.la
libmgext.la libminigui.a libvcongui-1.6.so.10 libvcongui.so
```

(5) 将 /usr/local/arm/4.4.3/arm-none-linux-gnueabi/lib/minigui 目录下的整个 res 文件夹下载到 /usr/local/lib/minigui 目录下。开发板上的这个目录是参考 MiniGUI.cfg 配置文件的

目录。

(6) 在开发板上运行 MiniGUI。将 mg-samples-3.0.12 目录中的 src 下生成的可执行文件，下载到开发板/usr/local/bin/minigui 目录下。

### 13.3 小 结

移植 MiniGUI 的过程，有类似于其他模块的移植部分，如交叉编译；也有不同于其他模块的部分，如需要增加对图形引擎的支持。在编译安装 MiniGUI 的过程中，注意对库文件目录的安装。移植的时候注意库文件和资源文件在开发板上的存放位置。

# 第 14 章 Qt 开发与 Qtopia 移植

Qt 是一个用 C++ 编写的成熟的跨平台 GUI 工具包。Qt 提供给应用程序开发者大部分的功能，来完成建立合适、高效的图形界面程序与后台执行的应用程序，它提供的是一种面向对象可扩展的和基于组件的编程模式。本章主要介绍 Qt 在 PC 上的安装、编程及 Qtopia 在上位机上的安装、开发和移植到 ARM 板。本章依然遵循安装、编译、调试、交叉编译和移植这一逐步深入的过程。

## 14.1 Qt 安装与编程

安装软件的方法一般是挂载系统安装盘，通过安装和卸载软件进行安装。如果没有系统安装文件的情况下，可以到网上下载 rpm 包进行安装。Qt 也分为免费版和商业版，免费版缺少支持且不能把 Qt 软件用于商业开发。

### 14.1.1 下载安装 Qt

用户可以使用如下命令在线安装 Qt：

```
#yum install qt3
```

当然，也可以从网站上获得 Qt 的源码包 qt-x11-free-3.3.8b.tar.gz 自己创建 Qt。编译和安装源码的方法如下：

```
#tar zxvf qt-x11-free-3.3.8b.tar.gz      //解压源码包
#cd qt-x11-free-3.3.8b
#./configure                                //执行 configure 后输入 yes 表示接受 GPL
Type 'Q' to view the Q Public License.
Type '2' to view the GNU General Public License version 2.
Type '3' to view the GNU General Public License version 3.
Type 'yes' to accept this license offer.
Type 'no' to decline this license offer.
Do you accept the terms of either license? Yes    //输入 yes 回车后开始生成
                                                    Makefile
#make                                         //编译
```

编译安装完成后，将 Qt 库路径添加到文件/etc/ld.so.conf 中：

```
/usr/lib/qt-3.3/lib                         //读者的 Qt 或许是 /usr/lib/qt-3/lib，可以通过下面的命令进行检验
```

安装好后，可以使用 echo 检验环境变量 QTDIR。

```
#echo $QTDIR
```

```
/usr/lib/qt-3.3
```

通过 `ldconfig` 命令使之在文件 `/etc/ld.so.conf` 中生效。

```
#ldconfig
```

如果上面的过程安装完成，14.1.2 节将带读者进入 Qt 编程，同时检验本节是否安装正确。查看安装的库的名字：

```
#ls /usr/lib/qt-3.3/lib
libdesignercore.a      libqt-mt.prl      libqui.so
libdesignercore.prl    libqt-mt.so       libqui.so.1
libeditor.a            libqt-mt.so.3     libqui.so.1.0
libeditor.prl          libqt-mt.so.3.3   libqui.so.1.0.0
libqassistantclient.a libqt-mt.so.3.3.8
libqassistantclient.prl libqui.prl
```

 注意：通过查看库可以知道这里安装的 Qt 库的名字为 `qt-mt`，读者的 Qt 的库也许叫 `qt`。  
`lib` 代表库，`so` 代表共享库，后面数字代表版本。

## 14.1.2 Qt 编程

Qt 是采用 C++ 编写的，其编程的单元也是类。对于熟悉 C++ 面向对象编程的读者来说，学习 Qt 编程将会非常容易。如果读者有 MFC 或者其他 GUI 开发经验，学习 Qt 编程会非常容易。即使读者没有学过 C++，只要读者有面向对象的基础即可。下面将通过一个简单的注册对话框介绍 Qt 编程。如果读者没有学过任何一门面向对象的开发语言，建议读者在开发 Qt 或者进行嵌入式 Qt 开发之前先学习面向对象的编程思想。

本例实现一个简单的登录对话框界面。程序主要有类的定义和类的实现两部分。`Login.h` 头文件定义了类的声明，声明了类的属性和操作。`Login.cpp` 文件完成方法的实现。下面依次介绍各个部分。

### 1. 类的定义部分

类的定义部分，指定了所继承的父类，定义了构造函数，声明了 3 个 `QLineEdit` 类型属性和 1 个方法 `Clicked()`。其代码如下：

```
#include <qmainwindow.h>
#include <qlineedit.h>
#include <qstring.h>
class Login : public QMainWindow
{
    Q_OBJECT //继承 QMainWindow, 这样可以继承 QMainWindow 的方法
public:
    Login(QWidget *parent = 0, const char *name = 0);
    QLineEdit *username_entry;
    QLineEdit *password_entry;
    QLineEdit *pwcheck_entry;
private slots:
    void Clicked();
};
```

## 2. 类的实现部分

构造函数设置了对话框中控件的显示风格，指定了窗口的大小，定义了信号与槽的连接关系。其代码如下：

```
#include "Login.moc"                                //运行预处理头文件得到的文件
#include <QPushButton.h>
#include < QApplication.h>
#include < QLabel.h>
#include < QLayout.h>
#include < iostream>
Login::Login(QWidget *parent, const char *name) : QMainWindow(parent, name)
{
    QWidget *widget = new QWidget(this);
    setCentralWidget(widget);
    QGridLayout *grid = new QGridLayout(widget, 4, 2, 10, 10, "grid");
                                         //将对话框分为4行2列
    username_entry = new QLineEdit(widget, "username_entry");
    password_entry = new QLineEdit(widget, "password_entry");
    pwcheck_entry = new QLineEdit(widget, "pwcheck_entry");
    password_entry->setEchoMode(QLineEdit::Password);
                                         //回显时为星号
    pwcheck_entry->setEchoMode(QLineEdit::Password);

    grid->addWidget(new QLabel("Username", widget, "userlabel"),
                    0, 0, 0);           // Username为第1行第1列(0,0)
    grid->addWidget(new QLabel("Password", widget, "passwordlabel"),
                    1, 0, 0);           // Password为第2行第1列(1,0)
    grid->addWidget(new QLabel("PasswordCheck", widget, "pwcheck_entry"),
                    2, 0, 0);           // Password为第3行第1列(2,0)

    grid->addWidget(username_entry, 0, 1, 0);
    grid->addWidget(password_entry, 1, 1, 0);
    grid->addWidget(pwcheck_entry, 2, 1, 0);

    QPushButton *button = new QPushButton("Ok", widget, "button");
                                         //定义按钮
    grid->addWidget(button, 3, 1, Qt::AlignRight);
    resize(350, 200);
    connect(button, SIGNAL(clicked()), this, SLOT(Clicked()));
                                         //连接信号与槽，响应函数为Clicked()
}

void Login::Clicked(void)                      //单击Ok按钮时，Clicked响应
{
    std::cout << "password:" << password_entry->text() << "\n";
    std::cout << "pwcheck:" << pwcheck_entry->text() << "\n";
}
```

## 3. 主函数部分

主函数是程序的入口点，创建一个 Login 类实例，显示窗口并启动事件循环机制，当有事件（Clicked()）发生时，响应事件机制（将 password 和 pwcheck 输出到控制台）。其代码如下：

```

int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    Login *window = new Login();
    app.setMainWidget(window);           //设置应用程序的主窗口部件
    window->show();                   //启动事件循环
    return app.exec();
}

```

执行下面命令进行编译和运行，运行结果如图 14.1 所示。

```

# moc Login.h -o Login.moc
# g++ -o login Login.cpp -I$QTDIR/include -L$QTDIR/lib -lqt-mt
#./login

```

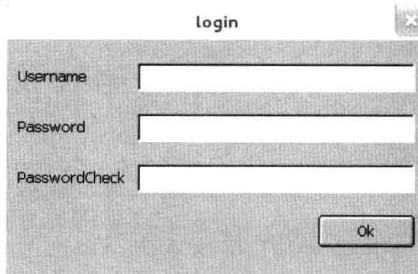


图 14.1 login 运行结果

### 14.1.3 使用 qmake 生成 Makefile

使用 qmake 生成 Makefile 也是 Qt 编程的基础。在 Qt 集成开发工具中，将 qmake 嵌入到开发工具中辅助开发人员生成 Makefile 文件。下面通过一个例子程序说明如何使用 qmake 生成 Makefile。将 qmake 所在的路径加入到环境变量 PATH 中，使用下面命令设置 PATH。

```
#export PATH=/usr/lib/qt-3.3/bin:$PATH
```

通过一个简单的例子说明 qmake 如何生成 Makefile 文件，新建 hello.c 文件。

```

#vi hello.c
#include <stdio.h>
int main()
{
    printf("Hello World!\n");
    return 0;
}

```

编写项目文件 hello.pro，该文件用于 qmake 生成 Makefile 文件。

```

#vi hello.pro
SOURCES = hello.c
CONFIG += qt warn_on release

```

qmake 可以通过下面的命令生成 Makefile，然后直接 make 可以生成可执行程序。

```

#qmake -o Makefile hello.pro
#make
#./hello

```

## 14.2 Qtopia Core 在 X86 平台上的安装和应用

Qtopia Core 是 QT 的嵌入式版本，是 Trolltech（已于 2008 年被 NOKIA 收购）公司从版本 4.1 开始将 Qt/E 并入 Qtopia 产品线产生的结果。Qtopia Core 与 Qt/X11 的最大区别就是不依赖 X Server 或 Xlib，而是直接访问帧缓冲设备（Frambuffer），这样做最显著的特点就是减少了内存的消耗。安装在 X86 平台方便开发和调试，调试验证后移植嵌入式环境。

### 14.2.1 Qtopia Core 安装准备

在 X86 平台上编译 Qtopia Core 时需要准备两个程序：

□ qt-x11-opensource-src-4.3.5.tar.gz，编译其目的主要为了获得帧缓冲（QVFB）。

□ qtopia-core-opensource-src-4.3.5.tar.gz，编译、安装获得环境 QtopiaCore-4.3.5-arm。

将 qt-x11-opensource-src-4.3.5.tar.gz 和 qtopia-core-opensource-src-4.3.5.tar.gz 复制到 /usr/local/arm/mini2440 目录下。

```
#tar zxvf qt-x11-opensource-src-4.3.5.tar.gz
#cd qt-x11-opensource-src-4.3.5
```

如果读者不是第一次安装 Qt，在编译之前可以检查本机是否有旧版本，将 QTDIR 指向新安装的路径下。设置 QTDIR、PATH 和 LD\_LIBRARY\_PATH 这 3 个环境变量。

```
#export QTDIR=$PWD //将 QTDIR 设置为当前目录
#export PATH=$QTDIR/bin:$PATH
#export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
```

完成环境变量设置后，使用 echo 命令检验环境变量是否设置成功。接下来按照下面的命令依次配置、编译、安装 qt-x11-opensource-src-4.3.5 和安装 qvfb。

```
./configure -qvfb
#yes
#gmake
#gmake install
#cd tools/qvfb
#make //进入 qvfb 目录下，编译 qvfb 工具
```

qvfb 安装成功后，在目录 /usr/local/arm/mini2440/qt-x11-opensource-src-4.3.5/bin 下会生成 qvfb、uic、moc、designer 等工具。在该目录下测试 qvfb 如图 14.2 所示，测试 designer 如图 14.3 所示。

```
#cd /usr/local/arm/mini2440/qt-x11-opensource-src-4.3.5/bin
#./qvfb
#./designer //启动 Qt designer 工具
```

到目前为止已经可以进行界面设计，对可执行程序可以在 X86 平台上进行虚拟仿真了。通过 Qt designer 工具进行界面设计保存为.ui 文件，通过 uic 工具将其转为.h 文件，在使用该文件的工程中就可以将该.h 文件包含进来。

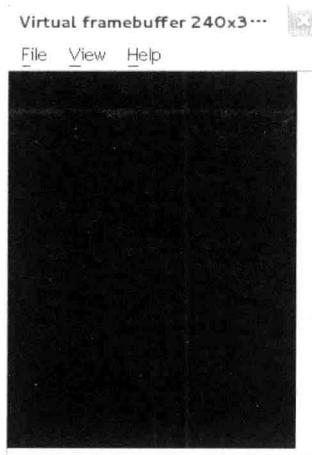


图 14.2 虚拟帧缓冲工具 qvfb

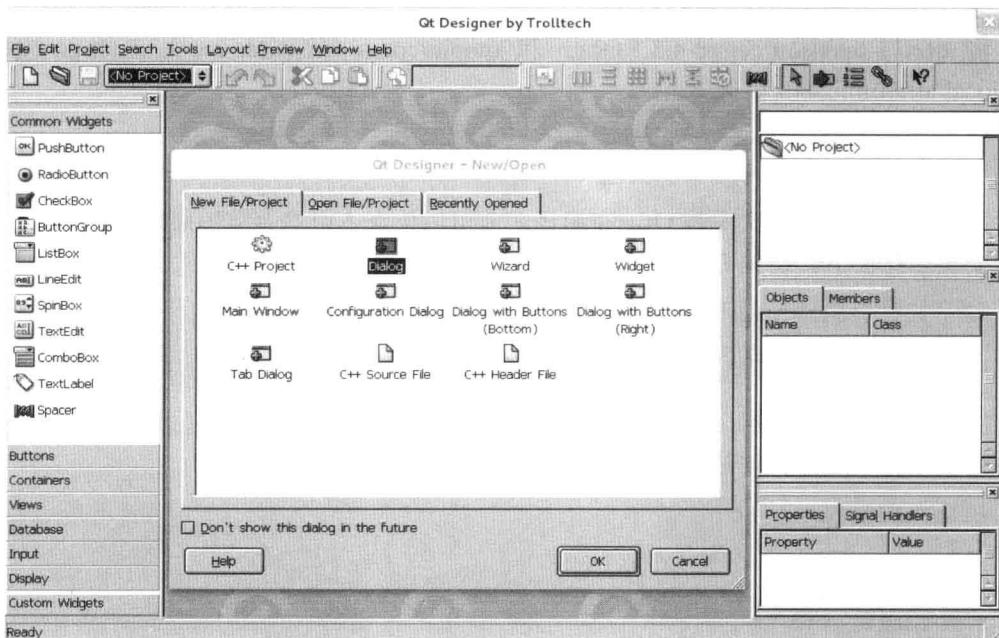


图 14.3 运行 Qt designer

在编译完 `qt-x11-opensource-src-4.3.5` 后，默认安装在`/usr/local/Trolltech/Qt-4.3.5`路径下。将生成的工具复制到`/usr/local/Trolltech/Qt-4.3.5/bin`目录下，同时删除 debug 文件，删除安装目录，修改指定工具的路径。

```
# cd /usr/local/arm/mini2440/qt-x11-opensource-src-4.3.5/bin/
# mv -f* /usr/local/Trolltech/Qt-4.3.5/bin/          //将工具复制到安装目录
# rm -rf qt-x11-opensource-src-4.3.5/                //删除安装目录
# cd /usr/local/Trolltech/Qt-4.3.5/bin
# rm -f *.debug                                     //删除 debug 文件
# export QTDIR=/usr/local/Trolltech/Qt-4.3.5        //修改环境变量
# export PATH=$QTDIR/bin:$PATH
```

```
# export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
```

### 14.2.2 编译 Qtopia Core

首先解压在/usr/local/arm/mini2440 目录下，配置在 X86 平台上编译，编译过程如下：

```
#tar zxvf qtopia-core-opensource-src-4.3.5.tar.gz
#cd qtopia-core-opensource-src-4.3.5
#./configure -embedded x86 -depths 4,8,16,24,32 -qconfig full -qvfb
-qt-libjpeg -qt-libpng -qt-gif
#yes
```

**注意：**embedded 指定 CPU 的体系结构，在刚接触 Qtopia 时，建议先编译 x86 版本，并加入 qvfb 的支持，在主机上模拟帧缓冲运行 Qtopia Core 程序。熟悉了 Qtopia 的开发过程后，将其编译成 ARM 版本，再进行交叉编译和移植。

接下来进行编译和安装。编译和安装使用 gmake。

```
#gmake
#gmake install
```

安装完成后，默认会安装在/usr/local/Trolltech/QtopiaCore-4.3.5 路径下。与 Qt 安装过程类似，安装完成后也同样设置环境变量，以及删除安装文件和 debug 文件。

```
#cd $HOME
#vi .bashrc
```

在文件后面添加下面内容并保存。然后执行 source 命令使之生效。

```
export PATH=$PATH:/usr/local/Trolltech/QtopiaCore-4.3.5/bin
export QTDIR=/usr/local/Trolltech/QtopiaCore-4.3.5
export PATH=$QTDIR/bin:$PATH
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
# source .bashrc
```

### 14.2.3 Qtopia 在 X86 平台上的应用开发

开发 Qtopia 程序的主要部分包括类的定义、类的实现和显示 3 个部分。以 Dialog 为例，在设计 Qtopia 时，开发初期在 dialog.h 中定义好其包含的属性和功能，在详细设计时在 dialog.cpp 中实现类方法，同时编写其测试代码，即显示 main.cpp。dialog.h 和 dialog.cpp 作为一个类的完整描述，可以看成一个小模块。

#### 1. 类的设计

设置 Qt 类时，属性包括界面中包含的对象；方法包括创建或者删除这些对象的操作等。下面为 Dialog 类的设计，它继承 QDialog。继承 QDialog 的公共属性和方法，可省去重新设计的时间。

```
class Dialog : public QDialog
{
```

```

Q_OBJECT

public:
    Dialog();

private:
    void createMenu();                                //创建菜单
    void createHorizontalGroupBox();                  //创建水平组合框
    void createGridGroupBox();                        //创建网格组合框

    enum { NumGridRows = 3, NumButtons = 4 };

    QMenuBar *menuBar;                               //包括菜单
    QGroupBox *horizontalGroupBox;                 //水平组合框
    QGroupBox *gridGroupBox;                        //网格组合框
    QTextEdit *smallEditor;                         //小文本编辑控件
    QTextEdit *bigEditor;                           //大文本编辑控件
    QLabel *labels[NumGridRows];                   //标签
    QLineEdit *lineEdits[NumGridRows];             //输入框空间
    QPushButton *buttons[NumButtons];              //按钮
    QDialogButtonBox *buttonBox;                   //对话框按钮

    QMenu *fileMenu;
    QAction *exitAction;
};


```

## 2. 类的实现

类的实现，即在概要设计的基础上对需求的细化过程。实现在类设计阶段的细节，为类的方法填充实现部分。

```

/*构造函数，创建初始化对话框中的所有控件*/
Dialog::Dialog()
{
    createMenu();                                  //调用创建菜单方法
    createHorizontalGroupBox();                    //调用创建水平组合框方法
    createGridGroupBox();                         //调用创建网格组合框方法

    bigEditor = new QTextEdit;                     //实例化一个文本编辑框
    bigEditor->setPlainText(tr("This widget takes up all the remaining
                                space "
                                "in the top-level layout."));
                                                //设置文本编辑框的初始内容

    buttonBox = new QDialogButtonBox(QDialogButtonBox::Ok
    | QDialogButtonBox::Cancel);
                                                //初始化一个对话框按钮
    connect(buttonBox, SIGNAL(accepted()), this, SLOT(accept()));
                                                //设置按钮的接收信号与槽
    connect(buttonBox, SIGNAL(rejected()), this, SLOT(reject()));
                                                //设置按钮的拒绝信号与槽
}

```

```

QVBoxLayout *mainLayout = new QVBoxLayout;
                                //实例化一个界面显示风格
mainLayout->setMenuBar(menuBar);           //界面菜单设置为刚才实例的对象
mainLayout->addWidget(horizontalGroupBox);   //添加水平组合框
mainLayout->addWidget(gridGroupBox);         //添加网格组合框
mainLayout->addWidget(bigEditor);            //添加文本编辑框
mainLayout->addWidget(buttonBox);             //添加按钮
setLayout(mainLayout);                      //设置显示风格

setWindowTitle(tr("Basic Layouts"));          //设置主题
}

void Dialog::createMenu()                  //创建菜单方法
{
    menuBar = new QMenuBar;                  //实例化一个菜单栏对象

    fileMenu = new QMenu(tr("&File"), this); //新建“文件”菜单
    exitAction = fileMenu->addAction(tr("E&xit"));
                                //在“文件”菜单中增加一个“退出”方法
    menuBar->addMenu(fileMenu);

    connect(exitAction, SIGNAL(triggered()), this, SLOT(accept()));
                                //连接“退出”操作的信号与槽
}

void Dialog::createHorizontalGroupBox() //创建水平组合框
{
    horizontalGroupBox = new QGroupBox(tr("Horizontal layout"));
    QHBoxLayout *layout = new QHBoxLayout;

    for (int i = 0; i < NumButtons; ++i) {
        buttons[i] = new QPushButton(tr("Button %1").arg(i + 1));
        layout->addWidget(buttons[i]);
    }
    horizontalGroupBox->setLayout(layout);
}

void Dialog::createGridGroupBox()          //创建网格组合框
{
    gridGroupBox = new QGroupBox(tr("Grid layout"));
    QGridLayout *layout = new QGridLayout;

    for (int i = 0; i < NumGridRows; ++i) {
        labels[i] = new QLabel(tr("Line %1:").arg(i + 1));
        lineEdits[i] = new QLineEdit;
        layout->addWidget(labels[i], i + 1, 0);
        layout->addWidget(lineEdits[i], i + 1, 1);
    }

    smallEditor = new QTextEdit;
    smallEditor->setPlainText(tr("This widget takes up about two thirds of
the "))
}

```

```

        "grid layout."));  

    layout->addWidget(smallEditor, 0, 2, 4, 1);  

    layout->setColumnStretch(1, 10);  

    layout->setColumnStretch(2, 20);  

    gridGroupBox->setLayout(layout);
}

```

### 3. 实现测试文件

在详细设计完一个功能或者一个类后，就开始对该功能或类进行测试。下面就是对 Dialog 类的测试文件 main.cpp。

```

#include <QApplication>
#include "dialog.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    Dialog dialog;
    return dialog.exec();
}

```

### 4. 编写工程文件

在详细设计完成后，进入集成测试阶段时需要对多个模块进行继承测试。这时需要编写工程文件，在其他项目中可能是编写整个工程的 Makefile 文件。在 Qtopia 程序中，qmake 会辅助生成 Makefile 文件，所以只需要编写工程文件 dialog.pro。

```

HEADERS      = dialog.h
SOURCES      = dialog.cpp \
               main.cpp

target.path = $$[QT_INSTALL_EXAMPLES]/layouts/basiclayouts
sources.files = $$SOURCES $$HEADERS *.pro
sources.path = $$[QT_INSTALL_EXAMPLES]/layouts/basiclayouts
INSTALLS += target sources

```

### 5. 生成Makefile、编译和运行

这部分包括通过工程文件生成 Makefile 和编译工程生成可执行文件，最后运行结果，测试结果是否与概要设计相符合。

```
#qmake -o Makefile dialog.pro
#make
```

打开两个终端，一个运行虚拟帧缓冲 qvfb，另一个运行 Qtopia 应用程序。运行效果如图 14.4 所示。

```
#qvfb
#./dialog -qws
```

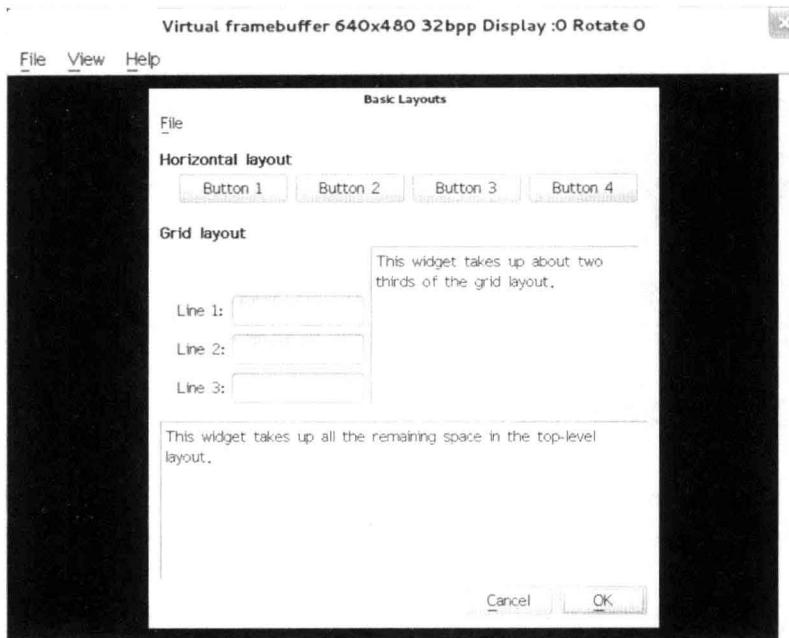


图 14.4 运行结果

注意：运行 qvfb 后在虚拟帧缓冲对话框出现后，选择 File-->configure 命令，在对话框中选择显示的大小为  $640 \times 480$ 。运行应用程序时带参数-qws。

### 14.3 Qtopia Core 在嵌入式 Linux 上的移植

Qtopia Core 的前身是 Qt/Embedded，继承了 Qt 4 的功能与优点，拥有与桌面系统的 Qt 相同的应用程序编程接口（API）和工具包。Qtopia Core 是一个为嵌入式设备上的图形用户接口和应用开发而订做的 C++ 工具开发包。Qtopia Core 采用与桌面版本同样的 API，但在其内部实现上做了很多调整，以适应硬件限制的嵌入式环境。Qtopia Core 包含多个 Qt 工具，可以进行快速优化和开发。

#### 14.3.1 Qtopia Core 移植准备

交叉编译 Qtopia Core 时需要准备两个程序：

- qt-x11-free-3.3.8b.tar.gz，编译其目的主要是为了获得库 libqjpeg。
- qtopia-core-opensource-src-4.3.5.tar.gz，编译、安装获得环境 QtopiaCore-4.3.5-arm。

注意：qt-x11-free-3.3.8b.tar.gz 不是必需的，因为在编译 qtopia-core-opensource-src-4.3.5 时出现缺少库 libqjpeg.so 支持的错误，而刚好编译 qt-x11-free-3.3.8b 后 qt-x11-free-3.3.8b/plugins/imageformats 目录下存在文件 libqjpeg.so。

将 qt-x11-free-3.3.8b.tar.gz 和 qtopia-core-opensource-src-4.3.5.tar.gz 复制到 /usr/local/arm/mini2440 目录下。14.1 节中已经介绍过 qt-x11-free-3.3.8b 编译方法了，这里直接给出其编译 qt-x11-free-3.3.8b 的命令。不对命令进行详细解释，有疑问可以参考 14.1.1 节。

```
# tar zxvf qt-x11-free-3.3.8b.tar.gz
# cd qt-x11-free-3.3.8b
# ./configure
# make
```

如果读者前面已经编译过 qt-x11-free-3.3.8b，而且确定在 qt-x11-free-3.3.8b/plugins/imageformats 目录下存在文件 libqjpeg.so，可以不必重新编译 qt-x11-free-3.3.8b。

### 14.3.2 交叉编译 Qtopia Core

交叉编译 Qtopia Core 的过程类似其他程序的交叉编译方法，总体包括 4 步：设置交叉编译器、配置生成 Makefile 文件、编译和安装。下面详细介绍整个交叉编译过程和注意细节。

#### 1. 设置交叉编译器

设置交叉编译器之前查看本机的交叉编译工具。如果读者现在还没有安装交叉编译工具，请查看前面的章节安装交叉编译工具的介绍。本机使用的交叉编译工具为 arm-linux-gcc-4.4.3。使用命令 arm-linux-gcc -v 查看本机安装好的交叉编译工具。

```
# arm-linux-gcc -v
Using built-in specs.
Target: arm-none-linux-gnueabi //本机的交叉编译器
```

进入 mini2440 目录，解压 qtopia-core-opensource-src-4.3.5.tar.gz。修改 mkspecs/qws/linux-arm-g++ 下的 qmake.conf 文件，把文件里面的编译器指定为 arm-none-linux-gnueabi，用 arm-none-linux-gnueabi-gcc 和 arm-none-linux-gnueabi-g++ 替代以下的 arm-linux-gcc 和 arm-linux-g++。具体的执行过程如下：

```
# tar zxvf qtopia-core-opensource-src-4.3.5.tar.gz
# cd qtopia-core-opensource-src-4.3.5
# cd mkspecs/qws/linux-arm-g++/
# vi qmake.conf
```

在文件 qmake.conf 中将与编译器有关的内容修改如下：

```
QMAKE_CC          = arm-linux-gcc
QMAKE_CXX         = arm-linux-g++
QMAKE_LINK        = arm-linux-g++
QMAKE_LINK_SHLIB = arm-linux-g++
```

改为：

```
QMAKE_CC          = arm-none-linux-gnueabi-gcc
QMAKE_CXX         = arm-none-linux-gnueabi-g++
QMAKE_LINK        = arm-none-linux-gnueabi-g++
QMAKE_LINK_SHLIB = arm-none-linux-gnueabi-g++
```

## 2. 配置编译选项

进入 qtopia-core-opensource-src-4.3.5 目录下，使用 configure 命令创建 qmake 生成 Makefile 文件。

```
# cd /usr/local/arm/mini2440/qtopia-core-opensource-src-4.3.5/
#./configure -no-largefile -no-qt3support -nomake tools -make examples
-silent -xplatform qws/linux-arm-g++ -embedded arm -depths 16,18,24,32
-qt-kbd-tty -qt-kbd-usb -system-libjpeg -qt-gfx-transformed -confirm-
license
```

配置完成后生成 Makefile、plugins 和 qmake 等目录。

## 3. 添加qjpeg库支持

为提供 qjpeg 库支持，将 qt-x11-free-3.3.8b/plugins/imageformats 目录下的文件 libqjpeg.so，复制到目录 /usr/local/arm/mini2440/qtopia-core-opensource-src-4.3.5/plugins/imageformats/ 下。

```
#cp /usr/local/arm/mini2440/qt-x11-free-3.3.8b/plugins/imageformats/
libqjpeg.so \
/usr/local/arm/mini2440/qtopia-core-opensource-src-4.3.5/plugins/imagef-
ormats/
```

## 4. 修改文件qdrawhelper.cpp

如果读者使用的是 arm-linux-gcc-3.3.2 版本的交叉编译器，或许不需要做本步的修改。本机使用 arm-linux-gcc-4.4.3 进行编译的时候，出现错误提示 explicit template specialization cannot have a storage class。解决的方法是做以下修改：

```
#vi src/gui/painting/qdrawhelper.cpp //去掉该文件两处 static
```

第 1 处（根据报错的行数为 5905）：

```
template <>
static inline void qt_memrotate90_template<quint18, quint32>(const quint32
*src,
                           int srcWidth, int srcHeight, int srcStride,
                           quint18 *dest, int dstStride)
```

改为：

```
template <>
inline void qt_memrotate90_template<quint18, quint32>(const quint32 *src,
                           int srcWidth, int srcHeight, int srcStride,
                           quint18 *dest, int dstStride)
```

第 2 处（根据报错的行数为 5929）：

```
template <>
static inline void qt_memrotate90_template<quint24, quint32>(const quint32
*src,
                           int srcWidth, int
                           srcHeight, int srcStride,
                           quint24 *dest, int dstStride)
```

改为：

```
template <>
inline void qt_memrotate90_template<quint24, quint32>(const quint32 *src,
    int srcWidth, int srcHeight, int srcStride,
    quint24 *dest, int dstStride)
```

在没有进行修改前编译会出现下面的错误。解决错误的方法有：根据错误的内容修改代码，或者安装低版本的编译器。

```
painting/qdrawhelper.cpp:5905: error: explicit template specialization
cannot have a storage class
painting/qdrawhelper.cpp:5929: error: explicit template specialization
cannot have a storage class
make[1]: *** [.obj/release-shared-emb-arm/qdrawhelper.o] 错误 1
make[1]: Leaving directory `/usr/local/arm/mini2440/qtopia-core-
opensource-src-4.4.3/src/gui'
make: *** [sub-gui-make_default-ordered] 错误 2
```

## 5. 编译、安装Qtopia Core

执行 make 和 make install，将 QtopiaCore-4.3.5-arm 安装到默认的路径/usr/local/Trolltech 下。

```
#make
#make install
```

安装完成后，检查在/usr/local 目录下多了目录 Trolltech。在 Trolltech 下多了目录 QtopiaCore-4.3.5-arm。/usr/local/Trolltech/QtopiaCore-4.3.5-arm 就是 qtopia-core-opensource-src-4.3.5 交叉编译、安装后的默认路径。

## 6. 修改环境变量PATH

修改环境变量 PATH，添加路径/usr/local/Trolltech/QtopiaCore-4.3.5-arm/bin，使得系统能够直接找到工具 moc、qmake、rcc 和 uic。

```
#cd $HOME
#vi .bashrc
在末尾添加
export PATH=$PATH:/usr/local/Trolltech/QtopiaCore-4.3.5-arm/bin
```

### 14.3.3 编译内核

因为在编译 Qtopia Core 时，编译器为 arm-none-linux-gnueabi。因此内核也应该采用此编译器进行编译，并且内核配置也要对 ARM EABI 进行支持。执行 make menuconfig 对内核进行配置，添加对 EABI 的支持。进入内核配置界面后选择 Kernel Features --->进入 Kernel Features 配置界面，勾选 EABI 项，如图 14.5 所示。

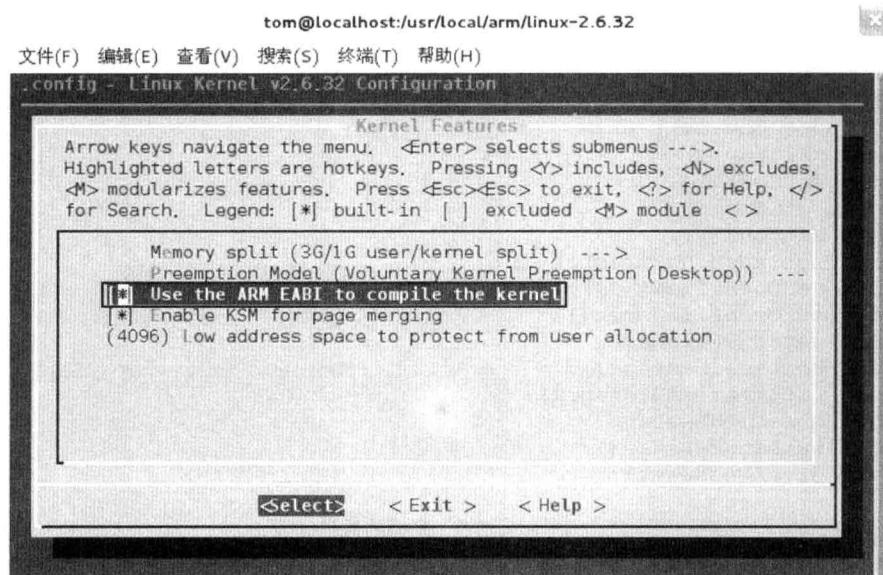


图 14.5 配置内核对 EABI 的支持

在内核源码目录下，修改内核 Makefile，将编译器设置为 arm-none-linux-gnueabi-。

```
#vi Makefile
ARCH      = arm
CROSS_COMPILE = arm-none-linux-gnueabi-
```

执行清理并重新生成内核映像文件。

```
#make clean
#make zImage
```

生成新的映像文件后，重新烧写内核和文件系统。

#### 14.3.4 应用程序开发

应用程序的开发往往是项目的核心内容，是系统功能需求部分的重点。应用程序的开发应该实现项目的需求，或者合同中指定的功能，或者实现子系统或模块的功能。界面程序有针对普通用户和管理员的，在具体项目中应根据实际情况进行功能设计。本例引用例子程序 standarddialogs 介绍 GUI 的移植过程。

##### 1. 编写应用程序

以安装路径下的例子介绍，进入下面的目录可以查看是否存在.cpp 文件和.h 文件 /usr/local/Trolltech/QtopiaCore-4.3.5-arm/examples/dialogs/standarddialogs。编写 dialog.h 文件定义类，包括属性和方法。

```
class Dialog : public QDialog //类的名字和继承关系
{
    Q_OBJECT //使用信号与槽机制时，类的声明中必须加上 Q_OBJECT 语句
public:
```

```

Dialog(QWidget *parent = 0);

private slots:                                //类的方法定义
    void setInteger();
    void setDouble();
    void setItem();
    void setText();
    void setColor();
    void setFont();
    void setExistingDirectory();
    void setOpenFileName();
    void setOpenFileNames();
    void setSaveFileName();
    void criticalMessage();
    void informationMessage();
    void questionMessage();
    void warningMessage();
    void errorMessage();

private:                                         //类的属性定义
    QCheckBox *native;
    QLabel *integerLabel;
    QLabel *doubleLabel;
    QLabel *itemLabel;
    QLabel *textLabel;
    QLabel *colorLabel;
    QLabel *fontLabel;
    QLabel *directoryLabel;
    QLabel *openFileNameLabel;
    QLabel *openFileNamesLabel;
    QLabel *saveFileNameLabel;
    QLabel *criticalLabel;
    QLabel *informationLabel;
    QLabel *questionLabel;
    QLabel *warningLabel;
    QLabel *errorLabel;
    QErrorMessage *errorMessageDialog;

    QString openFilesPath;
};

#endif

```

编写类的实现，在文件 dialog.cpp 中实现类的方法。

```

Dialog::Dialog(QWidget *parent)
    : QDialog(parent)
{
    //构造一个 QErrorMessage 类型的对象赋给 errorMessageDialog
    errorMessageDialog = new QErrorMessage(this);
    int frameStyle = QFrame::Sunken | QFrame::Panel;    //设置帧格式

    integerLabel = new QLabel;           //integerLabel 指向一个 QLabel 对象
    integerLabel->setFrameStyle(frameStyle);    //设置 integerLabel 帧格式
    QPushButton *integerButton = // integerButton 指向 QPushButton 类型实例
        new QPushButton(tr("QInputDialog::getInteger()"));
    //连接信号与槽，当 integerButton 被按下时，执行方法 setInteger()
    connect(integerButton, SIGNAL(clicked()), this, SLOT(setInteger()));
    //下面为设置空间的排列风格，其他的控件可以参照 integerButton 进行设置
}

```

```

QGridLayout *layout = new QGridLayout;
layout->setColumnStretch(1, 1);
layout->setColumnMinimumWidth(1, 250);
layout->addWidget(integerButton, 0, 0);
setLayout(layout);

setWindowTitle(tr("Standard Dialogs"));
}

//当 integerButton 被按下时，该方法被调用
void Dialog::setInteger()
{
    bool ok;
    int i = QInputDialog::getInteger(this, tr("QInputDialog::"
                                               "getInteger()"),
                                      tr("Percentage:"), 25, 0, 100, 1, &ok);
    if (ok)
        integerLabel->setText(tr("%1%").arg(i));
}

```

## 2. 编写工程文件standarddialogs.pro

编写工程文件包括指定依赖的头文件和包含的源文件。指定目标文件路径、源文件和源文件路径。

```

HEADERS      = dialog.h
SOURCES      = dialog.cpp \
               main.cpp
# install
target.path = $$[QT_INSTALL_EXAMPLES]/dialogs/standarddialogs
sources.files = $$SOURCES $$HEADERS *.pro
sources.path = $$[QT_INSTALL_EXAMPLES]/dialogs/standarddialogs
INSTALLS += target sources

```

## 3. 利用qmake和工程文件standarddialogs.pro生成Makefile，并编译生成可执行文件

```
#qmake -o Makefile standarddialogs.pro
#make                                //执行 make 后生成可执行文件
```

生成的 Makefile 如下，是否与预期的编译器、库文件路径、头文件路径、链接库使用的 qmake 等一致。

```

CC          = arm-none-linux-gnueabi-gcc
CXX         = arm-none-linux-gnueabi-g++
DEFINES     = -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_NETWORK_LIB -DQT_CORE_LIB
-DQT_SHARED
CFLAGS      = -pipe -O2 -Wall -W -D_REENTRANT $(DEFINES)
CXXFLAGS    = -pipe -O2 -Wall -W -D_REENTRANT $(DEFINES)
INCPATH     = -I../../../../mkspecs/qws/linux-arm-g++ -I. -I../../../../
include/QtCore -I../../../../include/QtCore -I../../../../include/QtNetwork
-I../../../../include/QtNetwork -I../../../../include/QtGui -I../../../../include/
QtGui -I../../../../include -I. -I.
LINK        = arm-none-linux-gnueabi-g++
LFLAGS      = -Wl,-rpath,/usr/local/Trolltech/QtopiaCore-4.3.5-arm/lib
LIBS        = $(SUBLIBS) -L/usr/local/Trolltech/QtopiaCore-4.3.5-arm/lib
-1QtGui -L/usr/local/Trolltech/QtopiaCore-4.3.5-arm/lib
-1QtNetwork -1QtCore -lm -lrt -ldl -lpthread
AR          = arm-none-linux-gnueabi-ar cqs

```

```
RANLIB      = arm-none-linux-gnueabi-ranlib          //编译器 lib 库
QMAKE      = /usr/local/Trolltech/QtopiaCore-4.3.5-arm/bin/qmake //qmake 路径
```

### 14.3.5 应用程序移植

在移植应用程序前，确定已经正确移植了所需的内核和文件系统，内核需要支持 ARM EABI 编译。接下来主要进行 3 部分工作：移植库、设置环境变量和移植应用程序。

#### 1. 移植库文件到开发板

在开发板 /usr/local 目录下新建目录 QtopiaCore，将上位机 /usr/local/Trolltech /QtopiaCore-4.3.5-arm/lib 目录下的库文件复制到 QtopiaCore/lib 目录下。

```
libQtCore.so.4
libQtNetwork.so.4
libQtGui.so.4
```

#### 2. 设置环境变量

```
export QTDIR=/usr/local/QtopiaCore
export QPEDIR=/usr/local/QtopiaCore
export LD_LIBRARY_PATH=$QTDIR/lib:/usr/local/lib:$LD_LIBRARY_PATH
```

#### 3. 移植应用程序到开发板

将编译好的应用程序复制到 /usr/local/QtopiaCore 目录下，在该目录下执行如下命令，运行程序：

```
./configdialog -qws &
```

## 14.4 小 结

Qtopia 在嵌入式 GUI 应用中占了很大部分，现在其主要运用为电话和 PDA。对于使用 Qtopia 开发，读者需要深入学习 Qt 的 API 接口的使用，了解各个类的功能。Qtopia 安装、编译和移植只是带读者入门，为了完成项目，还需要更深入地学习 Qt 编程。

# 第 15 章 嵌入式数据库 Berkeley DB 移植

Berkeley DB 是一款健壮、高速的工业级嵌入式数据库产品。很多知名公司都采用了这款嵌入式数据库。Berkeley DB 的一个很重要的特点就是高速存储。在高流量、高并发的情况下，Berkeley DB 要比非嵌入式的数据库表现得更加出色。另外，它可以应用在各种操作系统上，也能编译成多种语言的 API 接口。本章将主要介绍在 Linux 上的编译，编译的接口为 C 和 C++。

## 15.1 数据库的基本概念

在移植和使用 Berkeley DB 之前，先了解数据库的一些基本概念。如果读者从来没有数据库设计的经验，或者从来没有使用过 DB2、SQL Server、MySQL、Oracle 等数据库的经验，那么可以把数据库理解为一个存放数据的仓库。下面将介绍以何种方式存放数据到仓库、以何种方式从仓库中取出数据以及提高这些方法的效率。任何数据库的设计都是围绕这些主题来实现的，当然嵌入式数据库除了考虑效率、安全性外，还需考虑占用的空间。

### 15.1.1 利用文档和源代码

首先读者应该从官方网站下载 Berkeley DB 的源码包。建议先在 Windows 目录下解压后，进入 docs 目录下。如果读者习惯使用 API 文档，应知道双击 index.html 就能进入帮助文档主页。下面有关数据库基本方法的介绍都是参照该 API 文档进行分析的。同时，读者还可以借助一些工具（如 Source Insight 或者 UltraEdit）打开源代码，使用工具的同步文件功能。在完成上述两项事情后，会在很大程度上提高读者的学习和开发进度。

### 15.1.2 创建环境句柄

在 Berkeley DB 中，创建一个所有应用程序存放数据的环境。在代码中，将通过一个环境句柄来引用环境，该句柄类型为 DB\_ENV。读者将使用这一句柄操作此环境。利用 API 文档，找到 Programmatic APIs，先查看提供给 C 语言的 API。

在 C 中使用该 API：

```
#include <db.h>
Int db_env_create(DB_ENV **dbenvp, u_int32_t flags);
```

功能描述：函数 db\_env\_create() 为 Berkeley DB 环境创建 DB\_ENV 结构体句柄。为该

结构体分配内存，并返回指向结构体的指针 dbenvp。函数 db\_env\_create()返回 0 表示创建成功，非 0 表示创建失败。

在 C++ 中使用该 API：

```
#include <db_cxx.h>
class DbEnv {
public:
    DbEnv(u_int32 flags);
    ~DbEnv();

    DB_ENV *DbEnv::get_DB_ENV();
    const DB_ENV *DbEnv::get_const_DB_ENV() const;
    static DbEnv *DbEnv::get_DbEnv(DB_ENV *dbenv);
    static const DbEnv *DbEnv::get_const_DbEnv(const DB_ENV *dbenv);
    ...
};
```

功能描述：DbEnv 对象是 Berkeley DB 环境的句柄。该构造函数创建 DbEnv 对象，并为该对象分配空间。在调用 DbEnv::close() 或 DbEnv::remove() 方法时释放该空间。

 注意：在这里结合 Berkeley DB 数据库讲解数据库的基本概念时，如果读者并没有编译、安装 Berkeley DB 就使用上面介绍的 API 或者 API 文档中的 API 接口函数，将会在程序编译时不能通过。在后面将会详细介绍如何安装成 C 或 C++ 的库。如果读者希望能先试验一番，可以先看如何编译和安装部分，然后回头看基本概念部分。

### 15.1.3 创建数据库句柄

Berkeley DB 创建一个数据库句柄来代表创建的表。在 C 中使用该 API：

```
#include <db.h>
int db_create(DB **dbp, DB_ENV *dbenv, u_int32_t flags);
```

功能描述：函数 db\_create() 为 Berkeley DB 数据库创建一个结构为 DB 的句柄。为该结构体分配内存，并返回指向结构体的指针 dbp。

在 C++ 中使用该 API：

```
#include <db_cxx.h>
class Db {
public:
    Db(DbEnv *dbenv, u_int32_t flags);
    ~Db();

    DB *Db::get_DB();
    const DB *Db::get_const_DB() const;
    static Db *Db::get_Db(DB *db);
    static const Db *Db::get_const_Db(const DB *db);
    ...
};
```

功能描述：该构造函数为 Berkeley DB 数据库创建一个 DB 对象的句柄。该构造函数为对象分配空间，在调用 Db::close()、Db::remove() 或 Db::rename() 时释放该空间。

### 15.1.4 打开数据库

打开由文件和数据库参数表示的数据库，为读写数据库做准备。当前支持 Berkeley DB 数据库文件的格式有 Btree、Hash、Queue 和 Recno。Btree 格式表示的是一个有序的平衡树结构；Hash 格式表示的是一个动态、可扩展的散列图；Queue 格式支持快速或通过逻辑记录号访问固定长度的连续记录；Recno 格式支持访问固定或可变长度记录，支持连续访问或者通过逻辑号访问，并且可以通过文本任意备份。

在 C 中使用该 API:

```
#include <db.h>
int
DB->open(DB *db, DB_TXN *txnid, const char *file,
          const char *database, DBTYPE type, u_int32_t flags, int mode)
```

功能描述：函数 DB->open() 打开失败时返回一个非 0 错误值，成功返回 0。如果函数 DB->open() 失败，函数 DB->close() 被调用丢弃 DB 句柄。

在 C++ 中使用该 API:

```
#include <db_cxx.h>
int
Db::open(DbTxn *txnid, const char *file,
          const char *database, DBTYPE type, u_int32_t flags, int mode);
```

功能描述：函数 Db::open() 打开通过文件或数据库表示的数据库。函数 Db::open() 打开失败时返回一个非 0 错误值，成功返回 0。如果函数 Db::open() 失败，函数 Db::close() 被调用丢弃 DB 句柄。

### 15.1.5 DBT 结构

在 Berkeley DB 中，DBT 结构用来定义 Key/Value 对。

在 C 中使用该结构:

```
#include <db.h>
typedef struct {
    void *data;
    u_int32_t size;
    u_int32_t ulen;
    u_int32_t dlen;
    u_int32_t doff;
    u_int32_t flags;
} DBT;
```

功能描述：DBT 结构的所有域在第一次使用前没有明确地初始化为空字节。在声明该结构体时，应该使用库函数 memset() 进行初始化。默认的情况下，flags 被设置为 0。

在 C++ 中使用该结构:

```
#include <db_cxx.h>
class Dbt {
public:
```

```

Dbt(void *data, size_t size);
Dbt();
Dbt(const Dbt &);

Dbt &operator = (const Dbt &);

void *get_data() const;
void set_data(void *);

u_int32_t get_size() const;
void set_size(u_int32_t);

u_int32_t get_ulen() const;
void set_ulen(u_int32_t);

u_int32_t get_dlen() const;
void set_dlen(u_int32_t);

u_int32_t get_doff() const; //返回局部记录的偏移 (bytes)
void set_doff(u_int32_t); //设置在被应用程序读写时，局部记录的偏移 (bytes)

u_int32_t get_flags() const;
void set_flags(u_int32_t);

DBT *Dbt::get_DBT();
const DBT *Dbt::get_const_DBT() const;
static Dbt *Dbt::get_Dbt(DBT *dbt);
static const Dbt *Dbt::get_const_Dbt(const DBT *dbt);
};

}

```

功能描述：Berkeley DB 数据库中，类 Dbt 用于将 key 和 data 项进行编码。key 和 data 项都被表示为 Dbt 对象。

### 15.1.6 存取数据

数据库的存取操作是数据库的主要操作，在 Berkeley DB 数据库中存取操作主要是对 key/data 进行存取。

在 C 中使用该 API:

```
#include <db.h>
int DB->put(DB *db,
    DB_TXN *txnid, DBT *key, DBT *data, u_int32_t flags);
```

功能描述：函数 DB->put()存放 key/data 到数据库 DB 中。

```
int DB->get(DB *db, DB_TXN *txnid, DBT *key, DBT *data, u_int32_t flags);
int DB->pget(DB *db, DB_TXN *txnid, DBT *key, DBT *pkey, DBT *data, u_int32_t flags);
```

功能描述：函数 DB->get()和 DB->pget()从数据库 DB 中取 key/data。

在 C++中使用该 API:

```
#include <db_cxx.h>
int Db::put(DbTxn *txnid, Dbt *key, Dbt *data, u_int32_t flags);
```

功能描述：函数 Db::put()存放 key/data 到数据库 DB 中。

```
int Db::get(DbTxn *txnid, Dbt *key, Dbt *data, u_int32_t flags);
int Db::pget(DbTxn *txnid, Dbt *key, Dbt *pkey, Dbt *data, u_int32_t flags);
```

功能描述：函数 Db::get() 和 Db::pget() 从数据库 DB 中取 key/data。

### 15.1.7 关闭数据库

Berkeley DB 数据库使用完后，应该关闭数据库，释放打开的资源。

在 C 中使用该 API：

```
#include <db.h>
int DB->close(DB *db, u_int32_t flags);
```

功能描述：函数 DB->close() 将所有缓存数据库信息存储到磁盘，关闭任何打开的游标，释放分配的任何资源，关闭任何文件。

在 C++ 中使用该 API：

```
#include <db_cxx.h>
int Db::close(u_int32_t flags);
```

功能描述：函数 Db::close() 将所有缓存数据库信息存储到磁盘，关闭任何打开游标，释放分配的任何资源，关闭任何文件。

本节以 Berkeley DB 数据库为例介绍了数据库有关操作的基本概念。有关数据库的其他概念还有事务处理、互斥锁、内存池和操作异常等。这些概念都可以在 Berkeley DB 的 API 文档上找到，读者可以自己查阅该文档。

## 15.2 Berkeley DB 数据库安装

Berkeley DB 数据库不仅可以针对不同的系统安装，还可以针对不同的语言安装。安装成 C 库，使用为 C 提供的 API 就可以使用该数据库。也可以被安装成 C++ 库或 Java 库，使用为 C++ 或 Java 提供的 API 就可以使用该数据库。下面主要介绍安装成 C 库、C++ 库和交叉编译安装过程。

### 15.2.1 安装成 C 库

在安装 Berkeley DB 数据库前，读者可以去 Berkeley DB 官方网站下载最新版的 Berkeley DB 数据库源代码。在写作本书时，Berkeley DB 的最新版本为 db-6.0.20.tar.gz。

(1) 复制 db-6.0.20.tar.gz 到 /usr/local 目录下，然后解压 db-6.0.20.tar.gz 到 /usr/local。

```
#tar zxvf db-6.0.20.tar.gz
#cd db-6.0.20
```

(2) 新建一个编译目录 build\_linux，进入该目录。使用 configure 命令，按默认方式进行编译安装，就可以安装成 C 库。

```
#mkdir build_linux          //建立编译目录
#cd build_linux
#../dist/configure          //生成 Makefile, 默认生成 C 库
#make                         //编译
#make install                 //安装
```

(3) 正确编译、安装完成后，默认的安装路径为/usr/local/BerkeleyDB.6.0。在该路径下生成目录 bin、docs、include 和 lib。在 lib 路径下生成的库文件有 libdb-6.0.a、libdb-6.0.la、libdb-6.0.so、libdb-6.so、libdb.a 和 libdb.so。

如果要卸载 Berkeley DB，可以在 build\_linux 目录下使用 make uninstall 命令进行卸载。

```
# cd /usr/local/db-6.0.20/build_linux
# make uninstall
```

卸载完成后，/usr/local/BerkeleyDB.6.0 目录下的所有子目录均为空。

## 15.2.2 安装成 C++ 库

安装成 C++ 库与安装成 C 库基本类似。编译时唯一的区别在于 configure 时带上参数 --enable-cxx。下面给出编译成 C++ 库的命令过程：

```
#cd /usr/local/db-6.0.20/build_linux /*如果用户没有建立目录 build_linux, 需
                                         在此步前执行#mkdir /usr/local/db-
                                         db-6.0.20/build_linux, 建立编译目
                                         录*/
# ../dist/configure --enable-cxx      //编译成 C++ 库
# make                           //编译
# make install                   //安装
```

 注意：在此步编译的过程中，可以看到执行编译时采用的编译器为 g++，而不是 GCC 了。

安装完成后，进入/usr/local/BerkeleyDB.6.0/lib 目录查看生成的库文件与编译成 C 库有何区别。

```
libdb-6.0.a    libdb-6.so      libdb_cxx-6.0.la  libdb_cxx.a
libdb-6.0.la   libdb.a        libdb_cxx-6.0.so  libdb_cxx.so
libdb-6.0.so   libdb_cxx-6.0.a libdb_cxx-6.so   libdb.so
```

安装结束后，清除编译/usr/local/db-6.0.20/build\_linux 目录下的所有文件。

## 15.2.3 交叉编译安装 Berkeley DB

交叉编译安装 Berkeley DB 时，以安装成 C++ 库为例。安装成 arm-linux 环境下的库，两点主要的区别是编译器选择 arm-linux-g++，平台选择 ARM。在配置 configure 参数时，如果忘记了该参数，可以使用 -help 进行查看。

```
#mkdir /usr/local/db-6.0.20/build_arm_linux
#cd /usr/local/db-6.0.20/build_arm_linux
#../dist/configure -help
```

查看默认参数配置细节。编译或者交叉编译其他源代码也是对类似的参数进行配置，

然后进行编译和安装。

```

Fine tuning of the installation directories:
--bindir=DIR           user executables [EPREFIX/bin]
--sbindir=DIR           system admin executables [EPREFIX/sbin]
--libexecdir=DIR        program executables [EPREFIX/libexec]
--sysconfdir=DIR        read-only single-machine data [PREFIX/etc]
--sharedstatedir=DIR   modifiable architecture-independent data
                       [PREFIX/com]
--localstatedir=DIR    modifiable single-machine data [PREFIX/var]
--libdir=DIR            object code libraries [EPREFIX/lib]
--includedir=DIR        C header files [PREFIX/include]
--oldincludedir=DIR    C header files for non-gcc [/usr/include]
--datarootdir=DIR      read-only arch.-independent data root
                       [PREFIX/share]
--datadir=DIR          read-only architecture-independent data
                       [DATAROOTDIR]
--infodir=DIR          info documentation [DATAROOTDIR/info]
--localedir=DIR         locale-dependent data [DATAROOTDIR/locale]
--mandir=DIR           man documentation [DATAROOTDIR/man]
--docdir=DIR           documentation root
                       [DATAROOTDIR/doc/db-db-6.0.20]
--htmldir=DIR          html documentation [DOCDIR]
--dvidir=DIR           dvi documentation [DOCDIR]
--pdfdir=DIR           pdf documentation [DOCDIR]
--psdir=DIR            ps documentation [DOCDIR]

Program names:
--program-prefix=PREFIX      prepend PREFIX to installed program names
--program-suffix=SUFFIX       append SUFFIX to installed program names
--program-transform-name=PROGRAM run sed PROGRAM on installed program
                                 names

System types: //交叉编译需要设置的地方
--build=BUILD               configure for building on BUILD [guessed]
--host=HOST                 cross-compile to build programs to run on HOST [BUILD]

Optional Features:
--disable-option-checking ignore unrecognized --enable/--with options
--disable-FEATURE           do not include FEATURE (same as --enable-
                            FEATURE=no)
--enable-FEATURE[=ARG]       include FEATURE [ARG=yes]
--disable-bigfile            Obsolete; use --disable-largefile instead.
--disable-cryptography      Do not build database cryptography support.
--disable-hash               Do not build Hash access method.
--disable-partition          Do not build partitioned database support.
--disable-compression        Do not build compression support.
--disable-mutexsupport       Do not build any mutex support.
--disable-atomicsupport      Do not build any native atomic operation support.
--disable-queue              Do not build Queue access method.
--disable-replication        Do not build database replication support.
--disable-statistics         Do not build statistics support.
--disable-verify             Do not build database verification support.
--enable-compat185           Build DB 1.85 compatibility API.
--enable-cxx                  Build C++ API. (编译成C++库时设置)
--enable-debug                Build a debugging version.
--enable-debug_rop            Build a version that logs read operations.
--enable-debug_wop            Build a version that logs write operations.
--enable-diagnostic          Build a version with run-time diagnostics.
--enable-dump185              Build db_dump185(1) to dump 1.85 databases.
--enable-java                 Build Java API. (编译成Java库时设置)

```

```

--enable-mingw      Build Berkeley DB for MinGW.
--enable-o_direct   Enable the O_DIRECT flag for direct I/O.
--enable-posixmutexes Force use of POSIX standard mutexes.

--enable-smallbuild Build small footprint version of the library.
--enable-stl        Build STL API.
--enable-tcl        Build Tcl API.
--enable-test       Configure to run the test suite.
--enable-uimutexes Force use of Unix International mutexes.
--enable-umrw       Mask harmless uninitialized memory read/writes.
--enable-shared[=PKGS] build shared libraries [default=yes]
--enable-static[=PKGS] build static libraries [default=yes]
--enable-fast-install[=PKGS]
                     optimize for fast installation [default=yes]
--disable-libtool-lock avoid locking (might break parallel builds)
--disable-largefile  omit support for large files

Optional Packages:
--with-PACKAGE[=ARG]  use PACKAGE [ARG=yes]
--without-PACKAGE    do not use PACKAGE (same as --with-PACKAGE=no)
--with-mutex=MUTEX   Select non-default mutex implementation.
--with-mutexalign=ALIGNMENT
                     Obsolete; use DbEnv:::mutex_set_align instead.
--with-tcl=DIR        Directory location of tclConfig.sh.
--with-uniquename=NAME Build a uniquely named library.
--with-pic            try to use only PIC/non-PIC objects [default=use both]
--with-gnu-ld          assume the C compiler uses GNU ld [default=no]

```

查看上面的信息，设置交叉编译的 configure 参数如下：

```

#           ./dist/configure           CC=arm-linux-gcc           --host=arm-linux
--enable-mutexsupport \
--exec-prefix=/usr/local/BerkeleyDB_ARM \
--includedir=/usr/local/BerkeleyDB_ARM/include --enable-cxx
# make
# make install

```

在目录/usr/local/BerkeleyDB\_ARM 下生成 4 个文件夹：

```
bin include lib
```

查看安装的 lib 库文件：

```

libdb-6.0.a  libdb-6.so      libdb_cxx-6.0.1a  libdb_cxx.a
libdb-6.0.1a libdb.a        libdb_cxx-6.0.so   libdb_cxx.so
libdb-6.0.so  libdb_cxx-6.0.a libdb_cxx-6.so   libdb.so

```

## 15.3 使用 Berkeley DB 数据库

前面已经介绍 Berkeley DB 数据库的基本概念和编译安装过程。本节将介绍如何使用安装好的 Berkeley DB 数据库。这里以 C++库为例介绍其使用方法。

### 15.3.1 代码分析

以目录 examples\cxx 下的文件 AccessExample.cpp 为例，说明如何编写程序操作

Berkeley DB 数据库。前面介绍数据库基本概念时，基本上介绍了例子中的数据库操作，读者不明白的地方可以参考 15.1 节或者参考 Berkeley DB 数据库的 C++ API 文档。

AccessExample 主要实现 Key/Data 存储和从数据库取数据的功能。程序的实现主要分为 3 个部分：类定义、主函数和访问数据库。下面分别介绍这 3 个部分。

## 1. 类的定义部分

AccessExample 的定义部分主要定义了访问数据库的方法和构造函数。其代码如下：

```
#include <db_cxx.h>
#define DATABASE      "access.db"
using std::cin;
using std::cout;
using std::cerr;
class AccessExample                                // 定义测试类
{
public:
    AccessExample();
    void run(bool removeExistingDatabase, const char *fileName); // 定义测试数据库方法

private:
    AccessExample(const AccessExample &);           // 复制构造函数
    void operator = (const AccessExample &);          // 赋值构造函数
};
```

## 2. 主函数部分

主函数部分是程序的入口点，指定了运行时选择的数据库。调用 run() 函数进行访问数据库。同时使用 try/catch 机制捕获访问数据库过程中的异常。

```
int main(int argc, char *argv[])
{
    int ch, rflag;
    const char *database;

    rflag = 0;
    while ((ch = getopt(argc, argv, "r")) != EOF)
        switch (ch) {
            case 'r':
                rflag = 1;
                break;
            case '?':
            default:
                return (usage());
        }
    argc -= optind;
    argv += optind;

    /* 如果命令行后面跟上参数为空，则数据库为默认的数据库“access.db”，否则取参数作为数据库 */
    database = *argv == NULL ? DATABASE : argv[0];

    /* 使用 try-catch 捕捉数据库操作过程中的异常 */
    try {
```

```

AccessExample app;
app.run((bool)(rflag == 1 ? true : false), database);
return (EXIT_SUCCESS);
}
catch (DbException &dbe) {
    cerr << "AccessExample: " << dbe.what() << "\n";
    return (EXIT_FAILURE);
}
}

```

### 3. 访问数据库部分

访问数据库部分主要完成构造数据库，打开数据库，向数据库中存储数据，以及遍历数据取数据，最后关闭数据库。

```

void AccessExample::run(bool removeExistingDatabase, const char *fileName)
{
    //移除旧的数据库
    if (removeExistingDatabase)
        (void)remove(fileName);

    //不需要环境句柄进行创建数据库对象
    Db db(0, 0);

    db.set_error_stream(&cerr);
    db.set_errprefix("AccessExample");
    db.set_pagesize(1024);           /* Page size: 1K. */
    db.set_cachesize(0, 32 * 1024, 0);
    //打开数据库 fileName, 如果数据库 fileName 不存在则创建数据库
    db.open(NULL, fileName, NULL, DB_BTREE, DB_CREATE, 0664);

    /*插入键值对到数据库中, 输入的字符串键, 其逆序为值*/
    char buf[1024], rbuf[1024];
    char *p, *t;
    int ret;
    u_int32_t len;

    for (;;) {
        cout << "input> ";
        cout.flush();                      //刷新输出缓冲区到输出流

        cin.getline(buf, sizeof(buf));      //获取输入行
        if (cin.eof())                     //如果输入结束, Windwos 中 Ctrl+Z
                                            //模拟文件结束, Linux 下 Ctrl+D 模拟文件结束
            break;

        if ((len = (u_int32_t)strlen(buf)) <= 0)
            continue;
        for (t = rbuf, p = buf + (len - 1); p >= buf;)           //逆序
            *t++ = *p--;
        *t++ = '\0';

        /*这里创建的两个对象 key 和 data 分别代表要存储的一个键值对的 key 和 data。把字节串的起始地址和长度传给了它们, Berkeley DB 即可得到这两个字节串*/
        Dbt key(buf, len + 1);
        Dbt data(rbuf, len + 1);
    }
}

```

```

/*保存 key 和 data, DB_NOOVERWRITE 表示: 如果存在重复的 key 和 data, 则不覆盖旧的 key 和 data*/
ret = db.put(0, &key, &data, DB_NOOVERWRITE);
if (ret == DB_KEYEXIST) {
    cout << "Key " << buf << " already exists.\n";
}
cout << "\n";

try {
    //建立游标用于遍历数据库
    Dbc *dbc;
    db.cursor(NULL, &dbc, 0);

    //遍历整个表打印 key/data 对
    Dbt key;
    Dbt data;
    while (dbc->get(&key, &data, DB_NEXT) == 0) {
        char *key_string = (char *)key.get_data();
        char *data_string = (char *)data.get_data();
        cout << key_string << " : " << data_string << "\n";
    }
    /*关闭游标。因为游标稳定性导致游标所引用的页面被锁定, 使用同一个数据库的其他
    进程或者线程无法访问这些页面*/
    dbc->close();
}
/*捕获异常*/
catch (DbException &dbe) {
    cerr << "AccessExample: " << dbe.what() << "\n";
}
/*关闭数据库*/
db.close(0);
}

```

### 15.3.2 编译运行程序

AccessExample.cpp 使用了 C++ API。编译 AccessExample.cpp 的命令如下：

```
#g++ -o AccessExample -I /usr/local/BerkeleyDB.6.0/include/ -L
/usr/local/BerkeleyDB.6.0/lib/ AccessExample.cpp -lpthread -ldb_cxx
```

命令说明：

g++	//编译器
-o AccessExample	//输出文件名
-I /usr/local/BerkeleyDB.4.8/include/	//指定头文件路径
-L /usr/local/BerkeleyDB.4.8/lib/	//指定库文件路径
AccessExample.cpp	//源文件
-lpthread -ldb_cxx	//指定库名字为 libpthread 和 libdb_cxx

编译完成后生成可执行文件 AccessExample。如果没有设置 lib 库的路径，运行 AccessExample 时会出现找不到库的错误。添加库的路径到文件/etc/ld.so.conf 后面，并且运行 ldconfig 使之生效。

```
# vi /etc/ld.so.conf
```

添加下面内容:

```
/usr/local/BerkeleyDB.6.0/lib
```

```
# ldconfig
```

运行 AccessExample，命令后面带参数，参数即为数据库名称，不带参数则数据库名称默认为 access.db。保存输入 key/data 对，并且打印整个表信息。运行执行结果如下：

```
# ./AccessExample
input> abc //输入 abc
input> bcd //输入 bcd
input> cde //输入 cde
input> efg //输入 efg
input> ghi //输入 ghi
input> //按下 Ctrl+D 表示输入结束
abc : cba
bcd : dcba
cde : edc
efg : gfe
ghi : ihg
```

## 15.4 移植 Berkeley DB 数据库

本节将以实例讲解 Berkeley DB 设计类似电话本功能的查询、添加功能。在调试的过程中先在上位机中编译、调试后，再进行交叉编译，然后进行移植。

### 15.4.1 数据库设计

使用 Berkeley DB 数据库时，一个数据库中包含一个表。这个简单的表包含两个字段：用户姓名和号码。通过输入用户名和号码保存到数据库，然后遍历打印整个表信息。

数据库名称：user2num.db。

Key: user。

Data: num。

数据库的操作包括：

存数据操作 put()。

取数据操作 get()。

### 15.4.2 编写应用程序

根据上面的设计，编写代码时主要的功能实现为存数据操作和取数据操作。本程序 PhoneBook.cpp 参考例子程序 AccessExample.cpp。下面是程序源代码。

```
//PhoneBook.cpp
#include <sys/types.h>
```

```
//文件名
//添加的所有头文件
```

```

#include <iostream>
#include <iomanip>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <db_cxx.h>

//名字空间
using std::cin;
using std::cout;
using std::cerr;
using std::endl;

int main()
{
    char *fileName;
    Db db(0, 0); //创建没有环境的数据库

    db.set_error_stream(&cerr);
    db.set_errpx("PhoneBook");
    db.set_pagesize(1024); //页大小为 1K
    db.set_cachesize(0, 32 * 1024, 0);

    fileName = "user2num.db";
    /*打开数据库 fileName。如果该数据库不存在则创建名字为 fileName 的数据库*/
    db.open(NULL, fileName, NULL, DB_BTREE, DB_CREATE, 0664);

    char user_buf[1024], num_buf[1024];
    char *p, *t;
    int ret;
    u_int32_t user_len, num_len;

    for (;;) {
        /*提示输入用户名*/
        cout << "input name> ";
        cout.flush();
        cin.getline(user_buf, sizeof(user_buf));
        /*提示输入对应用户的电话号码*/
        cout << " input phone num>";
        cout.flush();
        cin.getline(num_buf, sizeof(num_buf));

        /*判断是否结束输入，结束则跳出循环，否则继续输入*/
        if (cin.eof())
            break;

        /*如果输入为空则不执行后续操作，即不存入数据库*/
        if ((user_len = (u_int32_t)strlen(user_buf)) <= 0) || ((num_len
            = (u_int32_t)strlen(num_buf)) <= 0) )
            continue;

        /*创建的两个对象 key 和 data 分别代表要存储的一个键值对的 key 和 data。把
         字节串的起始地址和长度传给了它们，Berkeley DB 即可得到这两个字节串*/
        Dbt key(user_buf, user_len + 1);
        Dbt data(num_buf, num_len + 1);
    }
}

```

```

/*将输入的 Key 和 Data 对存入数据库，如果已经存在则不覆盖*/
ret = db.put(0, &key, &data, DB_NOOVERWRITE);
if (ret == DB_KEYEXIST) {
    cout << "Key " << user_buf << " already exists.\n";
}
cout << "\n";

try {
    /*定义访问数据库的游标*/
    Dbc *dbc;
    db.cursor(NULL, &dbc, 0);

    /*遍历数据库打印姓名和号码*/
    Dbt key;
    Dbt data;
    while (dbc->get(&key, &data, DB_NEXT) == 0) {
        char *key_string = (char *)key.get_data();
        char *data_string = (char *)data.get_data();
        cout << key_string << " : " << data_string << "\n";
    }
    /*关闭游标*/
    dbc->close();
}
catch (DbException &dbe) {
    cerr << "AccessExample: " << dbe.what() << "\n";
}

db.close(0);
}

```

### 15.4.3 调试和交叉编译应用程序

在上位机中进行编译和调试，使用的是/usr/local/BerkeleyDB.6.0/目录下的头文件和库。编译的方法为：

```

# g++ -o PhoneBook -I /usr/local/BerkeleyDB.6.0/include/ -L /usr/local/
BerkeleyDB.6.0/lib/ PhoneBook.cpp -lpthread -ldb_cxx
# ./PhoneBook                                //运行
input name> Tom                            //输入姓名
input phone num>13012345678                  //对应的电话号码
input name> Jack
input phone num>13787654321
input name> Mary
input phone num>13543216789
/*下面为打印的电话本信息*/
Jack : 13787654321
Mary : 13543216789
Tom : 13012345678

```

在上位机中调试通过后可以进行交叉编译和移植。交叉编译使用的是

/usr/local/BerkeleyDB\_ARM/目录下的头文件和库。编译方法为：

```
#arm-linux-g++ -o PhoneBook -I /usr/local/BerkeleyDB_ARM/include/ -L /usr/local/BerkeleyDB_ARM/lib/ PhoneBook.cpp -lpthread -ldb_cxx
```

交叉编译后生成：PhoneBook。

#### 15.4.4 数据库的移植和测试

移植时需要将交叉编译库/usr/local/BerkeleyDB\_ARM/lib 目录的 libdb\_cxx.so 和 libdb\_cxx-6.0.so 两个文件复制到开发板/lib 目录下。将 PhoneBook 放在开发板/usr/bin 目录下，并且修改其权限。

```
# chmod 777 PhoneBook  
# ./PhoneBook
```

在开发板上运行和上位机中运行的效果相同，在当前目录下也会生成 user2num.db 数据库。运行结果如图 15.1 所示。

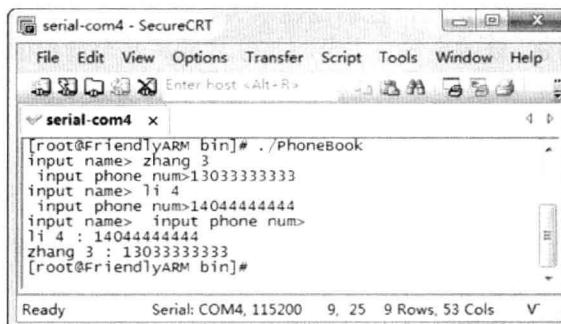


图 15.1 开发板上运行结果

#### 15.5 小结

Berkeley DB 数据库在嵌入式系统中应用越来越广泛，本章的编译和移植过程都比较简单。在实际的项目中使用 Berkeley DB 数据库时，重点是在数据库的设计。掌握数据库的设计，将需求变为数据库的详细设计方案，是读者将 Berkeley DB 数据库应用到嵌入式系统的基础。

# 第 16 章 嵌入式数据库 SQLite 移植

SQLite 是一个轻量级的数据库，非常适合用在嵌入式系统中。与 Berkeley DB 相比，它支持 SQL 语句，而且能在同一个数据库中创建多个表。本章将主要介绍如何使用 SQLite 数据库及其接口，同时介绍其在上位机中的编译和安装，最后介绍如何将它移植到嵌入式系统中。

## 16.1 SQLite 支持的 SQL 语句

SQLite 支持大部分 SQL 语句，包括创建索引、创建表、创建视图、创建虚表、删除表、删除索引、删除视图、修改表等。本节将带读者回顾几个简单的 SQL 语句，为后面安装和使用 SQLite 做铺垫。如果读者熟悉 SQL 语句可以直接阅读 16.2 节。

### 16.1.1 数据定义语句

下面给出 SQL 标准中的数据定义语句种类，并且在括号中指明 SQLite 是否支持该语句，同时通过实例给出所支持语句的使用方法。

- ALTER DATABASE 语法（不支持）。
- ALTER TABLE 语法（支持），用于更改原有表的结构。
- CREATE DATABASE 语法（不支持）。
- CREATE INDEX 语法（支持），用于创建索引。
- CREATE TABLE 语法（支持），用于创建表。
- DROP DATABASE 语法（不支持）。
- DROP INDEX 语法（支持），用于删除索引。
- DROP TABLE 语法（支持），用于删除表。
- RENAME TABLE 语法（不支持）。

下面通过实例说明以上语法：

(1) 创建表 student，该表包含 4 个列 id、name、sex 和 age。

```
create table student(id, name , sex, age );
```

(2) 修改表 student，在表中增加 1 个列 address。

```
alter table student add address;
```

(3) 创建索引 index\_id，索引是根据表 student 的列 id 进行创建。

```
create index index_id on student (id);
```

(4) 删除表中的索引 index\_id，在 SQLite 数据库中索引不能重名，不同的表只能有一个名字为 index\_id 的索引。删除的时候不需要指定是哪个表的索引。

```
drop index index_id;
```

### 16.1.2 数据操作语句

下面给出 SQL 标准中的数据操作语句种类，并且在括号中指明 SQLite 是否支持该语句。同时通过实例给出所支持语句的使用方法。

- DELETE 语法（支持）。
- DO 语法（不支持）。
- HANDLER 语法（不支持）。
- INSERT 语法（支持）。
- LOAD DATA INFILE 语法（不支持）。
- REPLACE 语法（支持）。
- SELECT 语法（支持）。
- Subquery 语法（不支持）。
- TRUNCATE 语法（不支持）。
- UPDATE 语法（支持）。

DELETE 语法，删除表中的一条或多条记录。INSERT 语法，向表中插入一条记录。

```
delete from student where id =1;
```

DELETE 命令用于从表中删除记录。命令包含 DELETE FROM 关键字及需要删除的记录所在的表名。若不使用 WHERE 子句，表中的所有行将全部被删除。否则仅删除符合条件的行。本句删除 id 为 1 的行。

```
insert into student values (11, 'DaSan', 'M', 20, 'beijing');
```

本句向表 student 中插入 id 字段为 11，name 字段为 DaSan，sex 字段为 M，age 字段为 20，address 字段为 beijing 的行。

```
select * from student;
```

select 语句从表 student 中查询所有项。

```
update student set age = 24 where age = 20;
```

update 语句更新所有字段 age 为 20 的值，将其更新为 age 为 24。

如果要对于 SQLite 的各种语法进行更深入的学习，可以参考 SQLite 的文档或者其他资料，本章的重点不是 SQL 语句。下面将介绍如何在上位机中安装 SQLite，上面的语句在安装了 SQLite 后可以一一得到验证。

## 16.2 SQLite 数据库编译、安装和使用

SQLite 的安装过程也比较简单，使用 SQLite 官方提供的混合安装包可以非常简便地

安装到系统中。

### 16.2.1 安装 SQLite

安装 SQLite 的过程比较简单，下面列出安装 `sqlite-autoconf-3080001.tar.gz` 的详细过程及安装过程的命令。

- (1) 复制 `sqlite-autoconf-3080001.tar.gz` 到 `/usr/local` 目录下。
- (2) 解压 `sqlite-autoconf-3080001.tar.gz`。

```
#tar zxvf sqlite-autoconf-3080001.tar.gz
```

- (3) 新建一个安装目录 `/usr/local/sqlite_x86`。

```
#mkdir /usr/local/sqlite_x86
```

- (4) 进入解压目录 `/usr/local/sqlite-autoconf-3080001` 配置 SQLite，执行 `configure` 命令生成 `Makefile` 文件。

```
#cd /usr/local/sqlite-autoconf-3080001
# ./configure --prefix=/usr/local/sqlite_x86
```

- (5) 执行 `make` 安装 SQLite。

```
# make
```

- (6) 执行 `make install` 将 SQLite 安装在 `/usr/local/sqlite_x86` 路径下。

```
# make install
```

安装完成后进入 `/usr/local/sqlite_x86` 目录查看安装文件。

```
# cd /usr/local/sqlite_x86/
# ls
bin include lib share          //安装目录下的文件
#cd bin
#ls
sqlite3                  //访问数据库的工具
```

- (7) 安装完成后，删除安装目录下的临时文件。

```
# rm -rf /usr/local/sqlite-autoconf-3080001
```

### 16.2.2 利用 SQL 语句操作 SQLite 数据库

SQLite 安装完成后对 SQLite 进行测试。根据前面列出的 SQL 语句，对 SQLite 支持的 SQL 语句进行测试。进入安装目录 `/usr/local/sqlite_x86/bin` 测试工具 `sqlite3`。

```
#cd /usr/local/sqlite_x86/bin
# ./sqlite3 testdb.db
```

创建数据库会出现下面的提示信息：

```
SQLite version 3.8.0.1 2013-08-29 17:35:01
```

```
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
```

执行`./sqlite3 testdb.db`后出现 SQLite 版本信息，并且提示用户输入 SQL 语句，下面为创建学生表及插入数据的 SQL 语句。

```
sqlite> create table student(id, name , sex, age );      //创建学生表
sqlite> insert into student values(1, 'Jack', 'M', 20); //向表中插入数据
sqlite> insert into student values(2, 'Tom', 'M', 21);
sqlite> insert into student values(3, 'Mary', 'W', 19); //显示表的所有信息
sqlite> select * from student;
```

显示的结果如下：

```
1|Jack|M|20
2|Tom|M|21
3|Mary|W|19
```

以下为更新表的操作，将学生表中 `age=24` 的项更新为 `age=20`。下面为输入的 SQL 语句。

```
sqlite> update student set age = 24 where age = 20;      //更新表
sqlite> select * from student;
```

上述操作的结果打印如下：

```
1|Jack|M|24
2|Tom|M|21
3|Mary|W|19
```

执行完对数据库的操作后，使用`.quit`命令退出数据库。

```
.quit                                //退出
```

### 16.2.3 利用 C 接口访问 SQLite 数据库

SQLite 为 C 语言提供的支持库位于安装目录下，通过提供的 C 接口可以对 SQLite 数据库进行操作。下面例程演示如何调用接口函数进行访问数据库。其主要过程与 16.2.2 节介绍的利用 SQL 语句访问数据库效果类似。下面给出了测试程序 `test.c` 的代码、编译过程和运行结果。

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

int main( void )
{
    sqlite3 *db=NULL;
    char *zErrMsg = 0;
    int rc;
    int i=0;

    rc = sqlite3_open("test.db", &db); //打开指定的数据库文件，如果不存在将创建
                                    //一个同名的数据库文件
```

```

if( rc )
{
    fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
    sqlite3_close(db);
    exit(1);
}
else
    printf("opened database test.db successfully!\n");

//创建一个表，如果该表存在，则不创建，并给出提示信息，存储在 zErrMsg 中
char *sql = "create table student(id, name , sex, age);";
//执行 SQL 语句
sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );

//插入数据
sql = "insert into student values(1, 'Jack', 'M', 20);";
//执行 SQL 语句
sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );
//插入数据
sql = "insert into student values(2, 'Tom', 'M', 21);";
sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );
//插入数据
sql = "insert into student values(3, 'Mary', 'W', 19);";
sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );
//查询结果
sql = "select * from student;";
sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );

//查询数据
/*
int sqlite3_get_table(sqlite3*, const char *sql,char***result , int
*nrow , int *ncolumn ,char **errmsg );
result 中是以数组的形式存放所查询的数据，首先是表名，再是数据。
nrow、ncolumn 分别为查询语句返回的结果集的行数、列数，没有查到结果时返回 0
*/
int nrow = 0, ncolumn = 0;
char **fristResult;                                //二维数组存放结果

sql = "select * from student;" ;
printf("\n");
sqlite3_get_table( db , sql , &fristResult , &nrow , &ncolumn , &zErrMsg );

//打印查询结果
printf( "row:%d column=%d \n" , nrow , ncolumn );
printf( "\nThe result of querying is : \n" );

for( i=0 ; i<( nrow + 1 ) * ncolumn ; i++ )
printf( "fristResult[%d] = %s\n", i , fristResult[i] );

//释放 fristResult 的内存空间
sqlite3_free_table( fristResult );

sql = "update student set age = 24 where age = 20;" ;
printf("\n");
sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );

nrow = 0;

```

```

ncolumn = 0;
char **secondResult; //二维数组存放结果
//查询数据
sql = "select * from student;" ;
sqlite3_get_table( db , sql , &secondResult , &nrow , &ncolumn ,
&zErrMsg );

//打印查询结果
printf( "row:%d column=%d \n" , nrow , ncolumn );
printf( "\nThe result of querying is : \n" );

for( i=0 ; i<( nrow + 1 ) * ncolumn ; i++ )
printf( "secondResult[%d] = %s\n" , i , secondResult[i] );

//释放 secondResult 的内存空间
sqlite3_free_table( secondResult );

printf("\n");
sqlite3_close(db); //关闭数据库
return 0;
}

```

编译程序命令如下：

```
# gcc -o test -I /usr/local/sqlite_x86/include -L /usr/local/sqlite_x86/lib
test.c -lsqlite3 -static -lpthread -ldl
```

编译命令说明：

gcc	//指定编译器
-o test	//输出文件名
-I /usr/local/sqlite_x86/include	//指定头文件路径
-L /usr/local/sqlite_x86/lib	//指定库文件路径
test.c	//源文件
-lsqlite3 -static -lpthread -ldl	//指定库名字为 libpthread 和 libsqlite3, 且为静态方式编译

执行编译命令后会生成可执行文件 test, 运行可执行文件。打印结果与执行 SQL 语句结果类似。

```

#!/bin/sh
./test
opened database test.db successfully!
row:3 column=4
The result of querying is :
fristResult[0] = id
fristResult[1] = name
fristResult[2] = sex
fristResult[3] = age
fristResult[4] = 1
fristResult[5] = Jack
fristResult[6] = M
fristResult[7] = 20
fristResult[8] = 2
fristResult[9] = Tom
fristResult[10] = M

```

```

fristResult[11] = 21
fristResult[12] = 3
fristResult[13] = Mary
fristResult[14] = W
fristResult[15] = 19

row:3 column=4

The result of querying is :
secondResult[0] = id
secondResult[1] = name
secondResult[2] = sex
secondResult[3] = age
secondResult[4] = 1
secondResult[5] = Jack
secondResult[6] = M
secondResult[7] = 24
secondResult[8] = 2
secondResult[9] = Tom
secondResult[10] = M
secondResult[11] = 21
secondResult[12] = 3
secondResult[13] = Mary
secondResult[14] = W
secondResult[15] = 19

```

 注意：表的第一行被更新了，和 SQL 语句执行的效果相同。通过调用接口后生成了 test.db 数据库和 student 表，此时也可以通过 SQL 语句进行打印。通过 SQL 语句查看调用 API 生成表的结果留给读者自己试验。

## 16.3 移植 SQLite

移植 SQLite 的过程主要有交叉编译数据库工具和库文件，编译应用程序，移植编译好库文件和应用程序到开发板，运行结果。

### 16.3.1 交叉编译 SQLite

为了与 x86 安装相区别，首先建立安装文件的目录 sqlite\_arm，将交叉编译好的库文件和工具安装在此目录下。

#### 1. 建立交叉编译、安装目录

在/usr/local 目录下建立 sqlite\_arm 目录，命令如下：

```
#mkdir /usr/local/sqlite_arm
```

#### 2. 配置交叉编译参数

配置交叉编译、安装参数，包括设置安装目录，目标主机为 arm-linux。配置命令如下：

```
# cd /usr/local/ sqlite-autoconf-3080001
# ./configure CC=arm-linux-gcc --prefix=/usr/local/sqlite_arm
--disable-tcl --host=arm-linux
```

### 3. 交叉编译和安装

执行完 configure 命令后会生成 Makefile 文件。执行 make 和 make install 进行编译和安装 Sqlite。命令如下：

```
#make
#make install
```

安装完成后同样会在 sqlite\_arm 目录下生成 include、lib、bin 和 share 4 个目录。在 bin 目录下有工具 sqlite3；在 lib 和 include 目录下对应生成库文件和头文件。

#### 16.3.2 交叉编译应用程序

交叉编译应用程序是为了在开发板上验证 Sqlite 数据库在开发板上是否能正确运行。交叉编译的应用程序仍然是前面在上位机中使用的 test.c 程序。将该程序复制到 /sqlite\_arm/bin/ 目录下进行交叉编译。实际上不用专门放在该目录下，这里笔者是为了与上位机中的测试程序相区别。执行下面的命令进行交叉编译应用程序：

```
# arm-linux-gcc -o test -I /usr/local/sqlite_arm/include -L /usr/local/
sqlite_arm/lib/ test.c -lsqlite3 -static -lpthread -ldl
```

命令说明：

arm-linux-gcc	//指定交叉编译器
-o test	//输出文件名
-I /usr/local/sqlite_arm/include	//指定头文件路径
-L /usr/local/sqlite_arm/lib	//指定库文件路径
test.c	//源文件
-lsqlite3 -static -lpthread -ldl	//指定库名字为 libpthread 和 libsqlite3，且为静态方式编译

如果应用程序没有错误就会在该目录下生成可执行文件 test，可以通过 file test 或者 readelf test -h 查看生成的文件的格式。

```
# file test
```

下面为生成文件的信息，指定运行平台为 ARM。

```
test: ELF 32-bit LSB executable, ARM, version 1 (SYSV), statically linked,
for GNU/Linux 2.6.32, not stripped
```

## 16.4 移植 SQLite 数据库

在前面的一些章节中，一直使用的是动态库，本例编译和移植将使用静态库方式。在

实际的开发中动态库方式比较常用，使用动态库使得应用程序比较小。在大型项目中，肯定存在很多不同的模块访问相同的库，采用动态库可以减少占用的空间。而在单个模块调试期间采用静态方式编译可减少移植时间。

### 16.4.1 文件移植

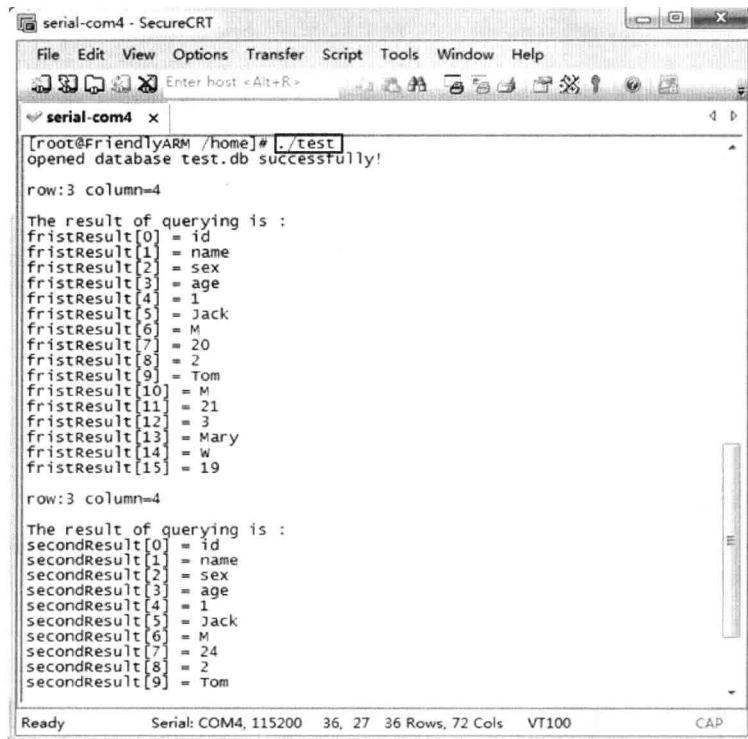
通过 FTP 工具将 16.3 节中 /usr/local/sqlite\_arm 目录下生成的 test 移植到开发板上，并且修改其执行权限。为了测试 SQL 语句是否可以在开发板上正常执行，还要移植库文件 libssqlite3.so.0 和 sqlite3。

```
#chmod +x test
#chmod +x sqlite3
```

 注意：如果只测试其 C/C++ 接口，只移植 test 文件就可以了。库文件 libssqlite3.so.0 放在 /lib 目录下。

### 16.4.2 运行应用程序

运行应用程序测试 C/C++ 接口。使用命令 ./test 执行应用程序和上位机中的运行结果一致，如图 16.1 所示。



```
serial-com4 - SecureCRT
File Edit View Options Transfer Script Tools Window Help
serial-com4 x
[root@FriendlyARM ~]# ./test
opened database test.db successfully!
row:3 column=4
The result of querying is :
fristResult[0] = id
fristResult[1] = name
fristResult[2] = sex
fristResult[3] = age
fristResult[4] = 1
fristResult[5] = Jack
fristResult[6] = M
fristResult[7] = 20
fristResult[8] = 2
fristResult[9] = Tom
fristResult[10] = M
fristResult[11] = 21
fristResult[12] = 3
fristResult[13] = Mary
fristResult[14] = W
fristResult[15] = 19

row:3 column=4
The result of querying is :
secondResult[0] = id
secondResult[1] = name
secondResult[2] = sex
secondResult[3] = age
secondResult[4] = 1
secondResult[5] = Jack
secondResult[6] = M
secondResult[7] = 24
secondResult[8] = 2
secondResult[9] = Tom

Ready Serial: COM4, 115200 36, 27 36 Rows, 72 Cols VT100 CAP
```

图 16.1 测试程序运行结果

### 16.4.3 测试 sqlite3

为了在开发板上运行 SQL 语句，应该测试工具 sqlite3 能否正确运行。使用命令 ./sqlite3 或者 ./sqlite3 test.db 进行测试，测试的具体命令如下，测试结果如图 16.2 所示。

```

serial-com4 - SecureCRT
File Edit View Options Transfer Script Tools Window Help
Enter host <Alt+R>
serial-com4 x
[root@FriendlyARM /home]# ./sqlite3 testdb.db
SQLite version 3.8.0.1 2013-08-29 17:35:01
Enter ".help" for instructions
Enter SQL statements terminated with a ";"

sqlite> create table teacher(id, name , sex, age );
sqlite> insert into teacher values(1, 'Jack', 'M', 40);
sqlite> insert into teacher values(2, 'Tom', 'M', 41);
sqlite> insert into teacher values(3, 'Mary', 'W', 49);
sqlite> select * from teacher;
1|Jack|M|40
2|Tom|M|41
3|Mary|W|49
sqlite> update teacher set age = 44 where age = 40;
sqlite> select * from teacher;
1|Jack|M|44
2|Tom|M|41
3|Mary|W|49
sqlite> .quit
[root@FriendlyARM /home]#

```

Ready      Serial: COM4, 115200  19, 27  19 Rows, 67 Cols    VT100

图 16.2 测试 SQL 语句

```
# ./sqlite3 testdb.db //创建数据库会出现下面的提示信息
```

执行 ./sqlite3 testdb.db 后出现 SQLite 版本信息，并且提示用户输入 SQL 语句：

```

SQLite version 3.6.17
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
```

下面的 SQL 语句为创建教师表以及插入数据。执行语句如下：

```

sqlite> create table teacher(id, name , sex, age ); //创建教师表
sqlite> insert into teacher values(1, 'Jack', 'M', 40); //向表中插入数据
sqlite> insert into teacher values(2, 'Tom', 'M', 41);
sqlite> insert into teacher values(3, 'Mary', 'W', 49);
sqlite> select * from teacher; //显示表的所有信息
```

显示的结果如下：

```
1|Jack|M|40
2|Tom|M|41
3|Mary|W|49
```

下面的 SQL 语句的作用是更新教师表，将 age=44 的项更新为 age=40。执行语句如下：

```

sqlite> update teacher set age = 44 where age = 40; //更新表
sqlite> select * from teacher;
1|Jack|M|44
2|Tom|M|41
3|Mary|W|49
```

完成对数据的操作后，使用.quit 命令退出数据库。

```
.quit //退出
```

⚠ 注意：方框圈住的部分为输入部分。对于其他的 SQL 语句读者可以自行进行测试。

## 16.5 小 结

SQLite 的移植与 Berkeley DB 移植类似，与 Berkeley DB 相比 Sqlite 多了对 SQL 语句的支持，可以通过 SQL 直接操作数据库。如果嵌入式系统中数据库不需要为多种语言（如 Java、C#等）提供接口，可以采用 Sqlite 作为嵌入式数据库。使用 Sqlite 作为数据库可方便使用 SQL 对数据库进行维护。

# 第 17 章 嵌入式 Web 服务器 BOA 移植

早期的嵌入式设备维护的人机接口界面基本采用 C/S 模式。这种方式需要客户端安装特定的客户端程序，当维护界面升级后还要向客户端发布新的安装程序或者补丁。而采用 B/S 方式就不需要制作特定平台的客户端安装程序，也不需要因更新版本而向客户发布新的版本或补丁。BOA 是一款单任务的 Web 服务器，将 BOA 移植到嵌入式设备就能通过网络来维护设备，同时不需要关心操作系统和硬件平台，只需要终端设备安装浏览器。本章将主要介绍 BOA 的特点、编译过程、测试方法及移植过程。

## 17.1 BOA 介绍

BOA 是一款单任务的 HTTP 服务器，与其他 Web 服务器(IIS、APACHE、WEBLOGIC、WEBSPHERE、TOMCAT、JBoss 等)相比，不同之处是当有连接请求到来时，它既不是为每个连接都单独创建进程，也不是采用复制自身进程处理多链接，而是通过建立 HTTP 请求列表来处理多路 HTTP 连接请求。同时，它只为 CGI 程序创建新的进程，在最大程度上节省了系统资源，这对资源受限的嵌入式系统来说至关重要。它还具有自动生成目录、自动解压文件等功能，因此，BOA 具有很高的 HTTP 请求处理速度和效率，应用在嵌入式系统中具有很高的价值。

### 17.1.1 BOA 的功能

嵌入式 Web 服务器 BOA 完成的功能包括接收客户端请求、分析请求、响应请求、向客户端返回请求处理的结果等。BOA 的工作流程如下：

- (1) 修正 BOA 服务器的根目录。
- (2) 读配置文件(boa.conf)。
- (3) 写日志文件。
- (4) 初始化 Web 服务器，包括创建环境变量、创建 TCP 套接字、绑定端口、开始侦听、进入循环结构，以及等待和接收客户的连接请求。
- (5) 当有客户端连接请求到达时，Web 服务器负责接收客户端请求，并保存相关请求信息。
- (6) 收到客户端的连接请求之后，Web 服务器分析客户端请求，解析出请求的方法、URL 目标、可选的查询信息及表单信息，同时根据客户端的请求做出相应的处理。
- (7) Web 服务器处理完客户端的请求后，向客户端发送响应信息，最后关闭与客户机的 TCP 连接。

### 17.1.2 BOA 流程分析

可以通过查看 src/boa.c 文件中的 main() 函数了解 BOA 的整个工作流程。下面将通过源码介绍 BOA 的主要工作流程。

#### 1. 修正BOA服务器的根目录

函数 fixup\_server\_root() 判断 Web 服务器的根目录是否有效。如果 Web 服务器的根目录有效则指定根目录，否则打印错误信息并退出程序。

```
static void fixup_server_root()
{
    char *dirbuf;
    if (!server_root) {                                //如果没有指定根目录
#ifndef SERVER_ROOT                               //该宏在 defines.h 中被定义为"/etc/boa"
        //函数 strdup() 功能为对参数目录字符串复制到新分配的字符指针，并将该指针作为返回值返回
        server_root = strdup(SERVER_ROOT);
        if (!server_root) {
            perror("strdup (SERVER_ROOT)"); //分配空间和复制失败则打印信息并退出
            exit(1);
        }
#else                                         //如果没有在 defines.h 中定义为
    "/etc/boa", 则打印提示信息，并退出程序
    fputs("boa: don't know where server root is. Please #define "
          "SERVER_ROOT in boa.h\n"
          "and recompile, or use the -c command line option to "
          "specify it.\n", stderr);
    exit(1);
#endif
    }
    dirbuf = normalize_path(server_root);           //格式化路径
    free(server_root);                            //释放空间
    server_root = dirbuf;                         //成功指定根目录路径
}
```

#### 2. 读取配置文件

函数 read\_config\_files() 用来读取配置文件信息，有关 Web 服务器的配置信息存放在文件 boa.conf 中。BOA 的配置信息包括 BOA 服务器监听的端口、绑定的 IP 地址、记载错误日志文件、设置存取日志文件等。

```
void read_config_files(void)
{
    char *temp;
    current_uid = getuid();
    yyin = fopen("boa.conf", "r");      //以只读方式打开配置信息文件 boa.conf

    if (!yyin) {                      //读取失败则打印打开文件失败信息
        fputs("Could not open boa.conf for reading.\n", stderr);
        exit(1);
    }
}
```

```

if (!server_name) { //如果没有指定服务器名字则指定服务器名字
    struct hostent *he;
    char temp_name[100];
    if (gethostname(temp_name, 100) == -1) { //获得服务器名字
        perror("gethostname:");
        exit(1);
    }
    he = gethostbyname(temp_name); //获取主机
    if (he == NULL) {
        perror("gethostbyname:");
        exit(1);
    }
    server_name = strdup(he->h_name); //获取主机名
    if (server_name == NULL) {
        perror("strdup:");
        exit(1);
    }
}
tempdir = getenv("TMP");
if (tempdir == NULL)
    tempdir = "/tmp";
//正确获得文档路径
if (document_root) {
    temp = normalize_path(document_root);
    free(document_root);
    document_root = temp;
}
//获得错误日志路径
if (error_log_name) {
    temp = normalize_path(error_log_name);
    free(error_log_name);
    error_log_name = temp;
}
//获得存取日志路径
if (access_log_name) {
    temp = normalize_path(access_log_name);
    free(access_log_name);
    access_log_name = temp;
}
//获得公共网关接口日志路径
if (cgi_log_name) {
    temp = normalize_path(cgi_log_name);
    free(cgi_log_name);
    cgi_log_name = temp;
}
if (dirmaker) {
    temp = normalize_path(dirmaker);
    free(dirmaker);
    dirmaker = temp;
}
}

```

### 3. 写日志文件

函数 `open_logs()` 打开日志文件并向文件中写日志。日志文件包括错误日志文件、存取日志文件、网关日志文件。

```
void open_logs(void)
```

```

{
    int error_log;
    if (error_log_name) {
        /*打开错误日志文件*/
        if (!!(error_log = open_gen_fd(error_log_name))) {
            DIE("unable to open error log");
        }
        /*重定向错误输出到错误日志文件*/
        if (dup2(error_log, STDERR_FILENO) == -1) {
            DIE("unable to dup2 the error log");
        }
        close(error_log);
    }
    /*第2个参数为F_SETFD时，表示设置文件描述符标记。fcntl文件锁有两种类型：建议性锁和强制性锁。系统默认fcntl都是建议性锁，当一个进程对文件加锁后，无论它是否释放所加的锁，只要文件关闭，内核都会自动释放加在文件上的建议性锁*/
    if (fcntl(STDERR_FILENO, F_SETFD, 1) == -1) {
        DIE("unable to fcntl the error log");
    }
    if (access_log_name) {
        /* 打开存取日志文件*/
        if (!(access_log = fopen_gen_fd(access_log_name, "w"))) {
            int errno_save = errno;
            fprintf(stderr, "Cannot open %s for logging: ",
                    access_log_name);
            errno = errno_save;
            perror("logfile open");
            exit(errno);
        }
        /*设置存取日志缓冲区*/
        setvbuf(access_log, (char *) NULL, _IOLBF, 0);
    } else
        access_log = NULL;

    if (cgi_log_name) {
        /*打开网关日志文件*/
        cgi_log_fd = open_gen_fd(cgi_log_name);
        if (cgi_log_fd == -1) {
            WARN("open cgi_log");
            free(cgi_log_name); //打开网关日志文件失败，则释放资源
            cgi_log_name = NULL;
            cgi_log_fd = 0;
        } else {
            //打开成功则加锁
            if (fcntl(cgi_log_fd, F_SETFD, 1) == -1) {
                WARN("unable to set close-on-exec flag for cgi_log");
                //打开失败则关闭该文件，内核将自动释放加在该文件上的建议性锁
                close(cgi_log_fd);
                cgi_log_fd = 0; //标识清零
                free(cgi_log_name); //释放资源
                cgi_log_name = NULL;
            }
        }
    }
}

```

```

    }
}
}
}
```

#### 4. 初始化Web服务器

函数 `create_server_socket()` 是 Web 服务器的核心的函数。该函数的作用是建立服务端 TCP 套接字；然后将其转换为无阻塞套接字；并且给服务套接字加锁；函数 `bind()` 用于建立套接字描述符与指定端口间的关联；并通过函数 `listen()` 在该指定端口进行侦听，等待远程连接请求；当连接请求到达时，BOA 调用函数 `get_request()` 获取请求信息，并通过调用函数 `accept()` 为该请求建立一个连接；在建立连接之后，接收请求信息，同时对请求进行分析；当有 CGI 请求时，为 CGI 程序创建进程，并将结果通过管道发送输出。

```

static int create_server_socket(void)
{
    int server_s;
    server_s = socket(SERVER_AF, SOCK_STREAM, IPPROTO_TCP);
                    //创建 TCP 服务套接字
    if (server_s == -1) {
        DIE("unable to create socket");
    }
    /*将服务套接字转换为无阻塞套接字*/
    if (set_nonblock_fd(server_s) == -1) {
        DIE("fcntl: unable to set server socket to nonblocking");
    }
    /*加锁服务套接字*/
    if (fcntl(server_s, F_SETFD, 1) == -1) {
        DIE("can't set close-on-exec on server socket!");
    }
    /*当设置 TCP 套接口接收缓冲区的大小时，服务端应该在监听前进行设置*/
    if ((setsockopt(server_s, SOL_SOCKET, SO_REUSEADDR, (void *) &sock_opt,
                    sizeof (sock_opt))) == -1) {
        DIE("setsockopt");
    }
    /*绑定套接字*/
    if (bind_server(server_s, server_ip) == -1) {
        DIE("unable to bind");
    }
    /*在指定端口进行监听，等待客户端的连接请求*/
    if (listen(server_s, backlog) == -1) {
        DIE("unable to listen");
    }
    return server_s;
}
```

##### 17.1.3 BOA 配置信息

BOA 的配置信息都保存在文件 `boa.conf` 中，该目录默认是放在`/etc/boa` 目录下，BOA

默认在该路径下读取相关的所有配置信息。下面将介绍 `boa.conf` 文件中相关配置信息的内容。

- Port: BOA 服务器监听的端口默认是 80。如果端口号小于 1024 时，须是 root 用户启动服务器。

```
Port 80
```

- Listen: 指定绑定的 IP 地址。注释掉该参数时，将绑定所有的地址。

```
#Listen 192.68.0.5
```

- User: 连接到服务器的客户端身份，可以是用户名或 UID。可以在文件/etc/passwd 中查看是否存在用户名 `nobody`。

```
User nobody
```

- Group: 连接到服务器的客户端的组，可以是组名或 GID。可以在文件/etc/group 中查看是否存在组名为 `nogroup` 的组。

```
Group nogroup
```

- ErrorLog: 该文件用于指定错误日志文件。如果路径没有以“/”开始，则指定其路径为相对于 ServerRoot 的路径。

```
ErrorLog /var/log/boa/error_log
```

- AccessLog: 用于设置存取日志文件。

```
AccessLog /var/log/boa/access_log
```

- DocumentRoot: 用于指定 HTML 文件的根目录。

```
DocumentRoot /var/www
```

- DirectoryIndex: 指定预生成目录信息的文件，注释掉此变量表示将使用 DirectoryMaker 变量。这个变量也就是设置默认主页的文件名。访问 BOA 服务器主页时就是访问的 `index.html` 页面。

```
DirectoryIndex index.html
```

- KeepAliveMax: 每个连接允许的请求数量。如果将此值设为 0，表示不限制请求的数目。这里表示允许请求的数量为 1000。

```
KeepAliveMax 1000
```

- KeepAliveTimeout: 在关闭持久连接前等待下一个请求的秒数。

```
KeepAliveTimeout 10
```

- CGIPat: CGI 程序的环境变量。

```
CGIPath /bin:/usr/bin:/usr/local/bin
```

- ScriptAlias: 指定服务端脚本路径的虚拟路径。

```
ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
```

在部署 Web 服务器时主要是对该配置文件进行配置，以及设置该配置中指定文件的路径和相关文件。在运行服务器的时候将给出 Web 服务器的具体配置。

## 17.2 BOA 编译和 HTML 页面测试

移植前在上位机中编译和测试 BOA。下面将详细介绍编译 BOA 的过程。本节介绍编译过程时，将会按照遇到错误、解决编译错误的顺序进行介绍。

### 17.2.1 编译 BOA 源代码

BOA 的源代码文件最新稳定版本为 `boa-0.94.13.tar.gz`。BOA 服务器的源代码在解压后的 `src` 目录下。这里在编译的上位机是 Fedora 19，不同版本的上位机可能会出现不同的编译错误，下面是其详细的编译过程，提供给读者在具体编译的时候作为参考。

将 `boa-0.94.13.tar.gz` 源码复制在 `/usr/local` 目录下，进行解压。

```
# tar zxvf boa-0.94.13.tar.gz
```

进入 `src` 目录，使用 `configure` 命令生成 `Makefile` 文件。

```
# cd boa-0.94.13/src/
# ./configure
# make
```

执行编译时，遇到下面的编译错误：

```
util.c:100:39: 错误：毗邻“t”和“->”不能给出一个有效的预处理标识符
make: *** [util.o] 错误 1
```

上面的错误是有关预处理的错误，进入文件 `util.c` 找到第 100 行，可以发现与预处理有关的内容，即宏 `TIMEZONE_OFFSET`。

```
time_offset = TIMEZONE_OFFSET(t);
```

跟踪 `TIMEZONE_OFFSET` 的定义，找到该宏在文件 `compat.h` 中定义。下面是该宏的定义：

```
#ifdef HAVE_TM_GMTOFF
#define TIMEZONE_OFFSET(foo) foo##->tm_gmtoff
#else
#define TIMEZONE_OFFSET(foo) timezone
#endif
```

根据错误提示信息，修改上述宏定义为：

```
#ifdef HAVE_TM_GMTOFF
#define TIMEZONE_OFFSET(foo) foo->tm_gmtoff
#else
#define TIMEZONE_OFFSET(foo) timezone
#endif
```

修改该宏定义后，再执行 `make` 进行编译，在 `src` 目录下生成 `boa` 可执行程序。

## 17.2.2 设置 BOA 配置信息

BOA 配置信息存放在 `boa.conf` 中，BOA 读取配置信息的时候默认路径是`/etc/boa`。因此运行 `boa` 时需要建立目录`/etc/boa`，并将配置信息放在该目录下。也可修改代码中的宏定义 `SERVER_ROOT`（在文件 `defines.h` 中）。

建立正确的配置文件路径，并复制配置文件到该目录下。

```
# mkdir /etc/boa
# cp boa.conf /etc/boa
```

建立日志目录`/var/log/boa`。

```
#mkdir /var/log/boa
```

对配置信息的修改如下。

(1) 文件`/etc/group`中不存在组名为 `nogroup` 的组。

```
Group nogroup
```

修改为：

```
Group 0
```

(2) CGI 程序的环境变量。

```
CGIPath /bin:/usr/bin:/usr/local/bin
```

修改为：

```
CGIPath /bin:/usr/bin:/var/www/cgi-bin
```

(3) `DirectoryIndex`: 不指定预生成目录信息的文件。

```
#DirectoryMaker /usr/lib/boa/boa_indexer //注释掉该句
```

(4) 指定服务端脚本路径的虚拟路径到`/var/www` 目录下。

```
ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
```

修改为：

```
ScriptAlias /cgi-bin/ /var/www/cgi-bin/
```

(5) 取消对 `ServerName` 的注释。

```
ServerName www.your.org.here
```

 注意：默认情况下是注释了 `ServerName`，这样在运行 `boa` 时，会出现“`gethostbyname:: No such file or directory`”或者“`get has tbymame::Success`”等异常，出现无法访问现象。

## 17.2.3 测试 BOA

测试 BOA 主要分为 3 步：编写测试页面，启动 Web 服务器，执行测试。根据配置文

件中的信息可知，测试页面放在/var/www 目录下，文件名为 index.html。

### 1. 编写测试主页index.html

在测试的过程中发现页面在显示中文的时候有乱码，因此，为了解决中文乱码问题为测试页面添加中文字符编码设置，放在 index.html 的开头。

```
<%@ page contentType="text/html; charset=gb2312"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

下面是测试页面的主要代码。

```
<html>
    <title>
        boa test page!
    </title>
    <head>
        <font color="#cc2200"><b></b>欢迎大家测试 BOA 服务器</font><p>
    </head>
    <body>
        这里是 BOA 服务器测试主页 (http+BOA 服务器 ip 地址)<p>
        测试方法在浏览器中输入，BOA 服务器的 IP 然后回车。<p>
        <font style="background-color: #808080">http://192.168.217.128<
        /font><p>
        如果直接在 Linux 上位机中进行测试可以直接在浏览器中输入下面地址并回车。<p>
        <font style="background-color: #808080">http://127.0.0.1<
        /font><p>
    </body>
</html>
```

### 2. 启动Web服务器

执行./boa 运行 Web 服务器。在进行页面访问测试之前，首先可以通过 ps 命令查看进程中是否存在 boa 进程。

```
#!/boa
#ps
```

如果 BOA Web 服务器没有正常起来，可以在/var/log/boa 目录中查看 error\_log 文件。如果在运行 boa 时出现错误。该错误会在错误日志中详细记录错误的原因。

```
boa.c:226 - icky Linux kernel bug!: Success
```

解决上面的错误，可以在 boa.c 文件中注释掉包含该行信息的相关代码，修改后重新编译并运行 BOA Web 服务器。

```
/*
if (setuid(0) != -1) {
    DIE("icky Linux kernel bug!");
}
*/
```

### 3. 启动Web服务器

如果通过 ps 命令可以查看到 boa 已经运行起来了，可以在 Linux 服务器端的浏览器中

通过输入 `http://127.0.0.1` 或者在客户端的浏览器中输入 `http://192.168.1.111`（服务器的 IP 地址）。

在浏览器中能正确显示测试主页，如果在服务器端对页面进行了修改，保存后，客户端只要刷新就能获得更新后的信息。

## 17.3 CGI 脚本测试

HTML 页面测试通过后，CGI 脚本的测试相对就容易很多。CGI 脚本的测试也包括 3 个部分：编写测试代码，编译测试代码，执行测试。

### 17.3.1 编写测试代码

CGI 的文件应该放在目录`/var/www/cgi-bin` 下，在该目录下编写 `hello.c` 文件，该测试文件内容为打印“Hello,World.”。测试文件的代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    printf("Content-type: text/html\n\n");
    printf("<html>\n");
    printf("<head><title>CGI Output</title></head>\n");
    printf("<body>\n");
    printf("<h1>Hello,world.</h1>\n");
    printf("<body>\n");
    printf("</html>\n");
    exit(0);
}
```

### 17.3.2 编译测试程序

将 `hello.c` 文件编译生成 `hello.cgi` 文件。编译命令如下：

```
# gcc -o hello.cgi hello.c
```

编译生成 `hello.cgi` 后，如果该文件不是在目录`/var/www/cgi-bin` 下，则将 `hello.cgi` 文件复制到该目录下。

### 17.3.3 测试 CGI 脚本

打开客户端的浏览器，在浏览器中输入下面的地址进行访问。

```
http://127.0.0.1/cgi-bin/hello.cgi
```

在浏览器中正确显示“Hello,World.”。

## 17.4 BOA 交叉编译与移植

本节将介绍如何在嵌入式产品中应用 BOA。在嵌入式产品中使用 BOA 需要对其进行交叉编译、配置、编写 HTML 页面、编写 CGI、部署上述文件到相应的目录。

### 17.4.1 交叉编译 BOA

进入 /usr/local/boa-0.94.13/src 目录对 BOA 进行交叉编译，这里使用的交叉编译器为 arm-linux-gcc-4.4.3。编译的过程如下：

(1) 在上位机中调试的时候已经通过 configure 命令生成了 Makefile 文件，这里只需要对生成的 Makefile 文件进行修改。在 Makefile 文件中将 gcc 改为 arm-linux-gcc，将 gcc -E 改为 arm-linux-gcc -E。修改后保存，然后进行编译。

```
#cd /usr/local/boa-0.94.13/src
#vi Makefile
gcc 改为 arm-linux-gcc
gcc -E 改为 arm-linux-gcc -E
# make
```

 注意：如果在执行 make 时，出现：make: Nothing to be done for 'all'., 则表示已经存在 BOA，也就是前面生成的 X86 平台的 BOA。可以通过 make clean 命令进行清除，然后执行 make 进行编译。

(2) 编译完成后，通过 file 命令对生成的执行文件进行查看。确认生成的是 ARM 平台格式的文件。

```
#file boa
```

查看该文件的属性如下：

```
boa: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.32, not stripped
```

该信息内容非常丰富，表示生成的 BOA 为可执行文件，运行的平台为 ARM 体系结构，使用的是动态链接库，含有调试信息，而且还包括大小端和 GUN 版本信息。此时生成的 BOA 文件大小为 200K 左右，如果去除调试信息，BOA 文件的大小为 60K 左右。通过下面命令去除调试信息：

```
# arm-linux-strip boa
```

### 17.4.2 准备测试程序

测试程序包括测试的 HTML 页面和 CGI 程序，HTML 和 CGI 程序，使用在上位机中测试的程序。这里只需要对 CGI 程序进行交叉编译即可以使用在 ARM 平台上。

```
# arm-linux-gcc -o hello.cgi hello.c
```

同样，执行完成后对生成 hello.cgi 查看其文件信息。

### 17.4.3 配置 BOA

配置 BOA 时，主要包括创建相关的目录和将文件放在相应的目录中。在向 mini2440 上移植 BOA 的过程中，发现文件系统中已经有了 BOA，并且自动启动。配置文件 boa.conf 的路径与本例不变，放在/etc/boa 目录下。使用 ps 查看进程，找到 BOA 所在的目录/usr/sbin，将该文件删除，使用本例编译的程序。将 BOA 程序移植到开发板上时，注意修改其权限为可执行。

(1) 对原配置文件 boa.conf 基本不作修改，其内容如下：

```
Port 80
User root
Group root
ErrorLog /dev/console
AccessLog /dev/null
ServerName friendly-arm
DocumentRoot /www
DirectoryIndex index.html
KeepAliveMax 1000
KeepAliveTimeout 10
MimeTypes /etc/mime.types
DefaultType text/plain
CGIPath /www
AddType application/x-httdp-cgi cgi
```

(2) 在原有目录/www 下没有 index.html 文件时，则将该目录下存在的文件 leds.html 改名为 index.html。因为默认的主页名字为 index.html。

```
#cd /www
#mv leds.html index.html
```

(3) 将主机中的文件/etc/mime.types 复制到开发板中对应的/etc 目录下。

### 17.4.4 测试

将编译好的程序替代旧的 BOA 程序后，重新启动 mini2440，在控制台打印 BOA 启动消息：

```
boa: server version Boa/0.94.13
boa: server built Aug 30 2013 at 19:15:03.
boa: starting server pid=681, port 80
```

(1) 测试 HTML，在上位机浏览器中输入 http://192.168.1.230（为开发板的 IP 地址，与上位机在同一个 IP 段，上位机的 IP 为 192.168.1.104）。

在上位机中可以正确显示网页。在开发板的终端显示：

```
request from 192.168.1.104 "GET /favicon.ico HTTP/1.1" ("/www/favicon.ico")
document open: No such file or directory
```

上面表明服务器已经正确收到请求 request 的消息，并且能正确解析。

(2) 测试 CGI，在上位机浏览器中输入 <http://192.168.1.230/leds.cgi>。浏览器中正确显示 leds.cgi 的页面。

## 17.5 BOA 与 SQLite 结合

在前面章节中，已经介绍了两种数据库 Berkeley DB 和 SQLite 的移植和使用。其维护过程是通过终端来维护，这种方式在实际产品中主要用于前期的开发，在产品的运行阶段由于条件的限制对嵌入式系统应该采用远程维护的方式。本节将通过实例介绍通过 BOA Web 服务 CGI 接口维护和管理嵌入式产品中的数据库。

### 17.5.1 通过 CGI 程序访问 SQLite

SQLite 提供了 C 语言访问的接口。通过采用 C 语言程序访问数据库，将该访问数据库的操作编译成 CGI 程序，部署在 BOA 的 CGI 路径下，远程维护人员通过调用此 CGI 程序就能实现远程维护数据库的目的。

下面是例子程序，其代码主要分为两部分：一部分是通过调用 C 接口对数据库进行创建、修改等维护工作；另一部分是通过 HTML 页面将结果返回给远程访问者。具体代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

int main( void )
{
    sqlite3 *db=NULL;
    char *zErrMsg = 0;
    int rc;
    int i=0;

    //通过 CGI 将结果返回给远程操作者
    printf("Content-Type:text/html\n\n");
                                //设置编码方案，解决中文乱码
    printf("<html>\n");
    printf("<head><title>CGI Output</title></head>\n");
    printf("<body>\n");
    printf("<h1>Access SQLite Database by CGI of Boa</h1>\n");
    printf("<p>\n");
    printf("<p>\n");

    rc = sqlite3_open("test.db", &db);
                                //打开指定的数据库文件，如果不存在将创建一个同名的数据库文件
    if( rc )
    {
        fprintf(stderr,      "Can't      open      database:      %s<br>",
        sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }
    else
        printf("opened database test.db successfully!<br>");
```

```

//创建一个表，如果该表存在，则不创建，并给出提示信息，存储在 zErrMsg 中
char *sql = "create table student(id, name , sex, age);";
//执行 SQL 语句
sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );

//插入数据
sql = "insert into student values(1, 'Jack', 'M', 20);";
//执行 SQL 语句
sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );
//插入数据
sql = "insert into student values(2, 'Tom', 'M', 21);";
sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );
//插入数据
sql = "insert into student values(3, 'Mary', 'W', 19);";
sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );
//查询结果
sql = "select * from student;" ;
sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );

//查询数据
/*
int sqlite3_get_table(sqlite3*, const char *sql,char***result , int
*nrow , int *ncolumn ,char **errmsg );
result 中是以数组的形式存放所查询的数据，首先是表名，再是数据。
nrow , ncolumn 分别为查询语句返回的结果集的行数、列数，没有查到结果时返回 0
*/
int nrow = 0, ncolumn = 0;
char **fristResult;//二维数组存放结果

sql = "select * from student;" ;
printf("\n");
sqlite3_get_table(db , sql , &fristResult , &nrow , &ncolumn , &zErrMsg );

//打印查询结果
printf( "</br>row:%d column=%d </br>" , nrow , ncolumn );
printf( "</br>The result of querying is : </br></br>" );

for( i=0 ; i<( nrow + 1 ) * ncolumn ; i++ )
printf( "fristResult[%d] = %s\n" , i , fristResult[i] );

//释放掉 fristResult 的内存空间
sqlite3_free_table( fristResult );

sql = "update student set age = 24 where age = 20;" ;
printf("</br>");
sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );

nrow = 0;
ncolumn = 0;
char **secondResult; //二维数组存放结果
//查询数据
sql = "select * from student;" ;
sqlite3_get_table( db , sql , &secondResult , &nrow , &ncolumn ,
&zErrMsg );

//打印查询结果
printf( "</br>row:%d column=%d </br>" , nrow , ncolumn );

```

```

printf( "</br>The result of querying is : </br></br>" );

for( i=0 ; i<( nrow + 1 ) * ncolumn ; i++ )
    printf( "secondResult[%d] = %s\n", i , secondResult[i] );

printf("<body>");
printf("</html>");
//释放掉 secondResult 的内存空间
sqlite3_free_table( secondResult );

printf("</br>");
sqlite3_close(db); //关闭数据库
return 0;
}

```

## 17.5.2 编译和测试

编译 SQLite 程序时，需要 SQLite 接口的头文件和库文件支持。对于 SQLite 的编译和安装过程，本节不再介绍，如果还没有安装 SQLite 的读者，可以参照 SQLite 数据库的移植对 SQLite 进行编译和安装。

### 1. CGI程序的编译和部署

将上述代码起名为 sqlite.c，放在 BOA 服务器的 CGI 路径下，对于本机的路径为 /var/www/cgi-bin。然后使用下面的命令对其进行编译：

```
#gcc -o sqlite.cgi -I /usr/local/sqlite_x86/include -L /usr/local/sqlite_x86/lib sqlite.c -lsqlite3 -static -lpthread -ldl
```

/usr/local/sqlite\_x86 是本机 SQLite 的安装目录。编译完成后在 /var/www/cgi-bin 目录下生成了 sqlite.cgi 文件，同时也完成了部署。

### 2. 测试sqlite.cgi

在 Windows 的浏览器中输入 <http://192.168.1.111/cgi-bin/sqlite.cgi>，来访问虚拟机（虚拟机的 IP 地址为 192.168.1.111）下的 CGI 程序。测试结果如图 17.1 所示。



图 17.1 通过 CGI 访问 SQLite 数据库

注意：在嵌入式产品中结合使用 SQLite 和 BOA 具有很高的效率，实现起来也比较容易。对于其移植过程，请读者结合前面的内容自己动手完成。

## 17.6 小 结

BOA 在嵌入式方面的应用非常简单有效，其编译和移植过程也比较简单。读者熟练掌握 BOA 的源代码和 SQLite 的源代码后，可以在自己的项目中灵活运用。另外，BOA 也可以和 Berkeley DB 数据库结合。本章的重点和难点是 BOA 的流程分析，比较实用并且很多情况下是与嵌入式数据库结合使用，读者可以编译更好的 CGI 程序维护远程的嵌入式数据库。

# 第 18 章 嵌入式 Web 服务器 Thttpd 移植

Thttpd 是一个简单的、小型的、可移植的、快速及安全的 HTTP 服务器。正因为它具有这些特点，将其应用在资源受限的嵌入式产品中非常合适。本章将介绍其编译、调试、使用和移植过程。

## 18.1 Thttpd 介绍

Thttpd 是一款小而快，且安全的 HTTP 服务器。下面将通过分析源代码介绍 Thttpd 工作过程。本章介绍的 thttpd 是基于 thttpd-2.25b 版本进行的。

### 18.1.1 Web 服务器比较

一般有 3 种常用 Web 服务器：Httpd、Thttpd 和 Boa。Httpd 是最简单的一个 Web 服务器，它的功能最弱，不支持认证，不支持 CGI（Common Gateway Interface，通用网关接口）。Thttpd 和 Boa 都支持认证、CGI 等，功能都比较全。Boa 源代码开放、性能可靠、稳定性好，但是仅能作为一个单任务的 Web 服务器。所以，使用简单、小巧、易移植、快速和安全的 Thttpd 嵌入式 Web 服务器是一个明智的选择。

另外，还有几款嵌入式 Web 服务器：Lighttpd、Shttpd、Mathopd、Minihttpd、Appweb、Goahead。读者有兴趣的话可以自己动手编译后进行测试，然后应用在自己的项目中。

### 18.1.2 Thttpd 的特点

Thttpd 的特点是高效、安全，并且支持 URL 流量控制。基于这些特点其在嵌入式方面的应用很有前景。下面分别介绍其特点。

#### 1. 安全性

安全性问题中最大的危险源并不是来自授权协议本身，而是取决于在使用授权协议时所指定的策略和程序。所以 Thttpd 在默认的状况下，仅运行于普通用户模式下，从而能够有效地禁止非授权的系统资源和数据的访问，同时 Thttpd 全面支持 HTTP 基本验证（RFC2617 HTTP Authentication），可有效解决安全性的问题。这一点正像很多人在使用 Windows 时都是使用管理员的身份登录，因此系统经常容易受到病毒和木马程序的袭击，如果将管理员权限改为普通用户访问权限，那么系统将安全得多。

## 2. 高效性

Thttpd 对于并发请求不使用 fork() 来创建子进程处理，而是采用多路复用（Multiplex）技术来实现。而通过 fork() 的方式创建子进程是父进程的一个复制，两者是独立的，使用该方式时，当并发的请求增多时，系统的性能被迅速降低。Thttpd 采用多路复用技术，当并发请求增多时节省了资源，提高了系统效率。

## 3. 流量控制

Thttpd 支持基于 URL 的文件流量限制，便于处理连续的视频流量。与 Apache 比较，随着请求频率增加、请求数量增加时，Thttpd 的优势变得更加明显。

### 18.1.3 Thttpd 核心代码分析

Thttpd 工作流程的主逻辑在 `thttpd.c` 文件的 `main()` 函数中。该函数中描述了 Thttpd 服务建立服务、接收请求、处理请求、日志文件及断开连接的过程。下面分析建立 Web 服务器的核心函数 `httpd_initialize()`。

函数 `httpd_initialize()` 在文件 `libthttpd.c` 中定义，用于初始化 Web 服务器，如成功就返回 `httpd_server` 类型指针指向建立的 Web 服务器。该函数主要为 Web 服务器分配资源，初始化监听套接字，初始化媒体类型表准备接收客户端的请求。该函数的具体定义在 `libthttpd.c` 中，其定义如下：

```

httpd_server*
httpd_initialize(
    char* hostname, httpd_sockaddr* sa4P, httpd_sockaddr* sa6P,
    unsigned short port, char* cgi_pattern, int cgi_limit, char* charset,
    char* p3p, int max_age, char* cwd, int no_log, FILE* logfp,
    int no_symlink_check, int vhost, int global_passwd, char* url_pattern,
    char* local_pattern, int no_empty_referers )
{
    httpd_server* hs;
    static char ghnbuf[256];
    char* cp;

    check_options();

    hs = NEW( httpd_server, 1 );      // 为准备建立的 Web 服务器分配资源，1 表示所建立 Web 服务器的个数
    if ( hs == (httpd_server*) 0 )    // 分配失败退出，并在系统日志中记录
    {
        syslog( LOG_CRIT, "out of memory allocating an httpd_server" );
        return (httpd_server*) 0;
    }

    if ( hostname != (char*) 0 )
    {
        hs->binding_hostname = strdup( hostname );
        // 设置 Web 服务器的 binding_hostname 字段
        if ( hs->binding_hostname == (char*) 0 )
            // 设置失败退出，并在日志中记录
    }
}

```

```

{
    syslog( LOG_CRIT, "out of memory copying hostname" );
    return (httpd_server*) 0;
}
hs->server_hostname = hs->binding_hostname; //设置 Web 服务器的 server_hostname 字段
}
else
{
    hs->binding_hostname = (char*) 0;
    hs->server_hostname = (char*) 0;
    if ( gethostname( ghnbuf, sizeof(ghnbuf) ) < 0 )
        //当主机名为空时, 获取主机名
        ghnbuf[0] = '\0';
#endif SERVER_NAME_LIST
    if ( ghnbuf[0] != '\0' )
        hs->server_hostname = hostname_map( ghnbuf ); //设置 Web 服务器的 server_hostname 字段
#endif /* SERVER_NAME_LIST */
    if ( hs->server_hostname == (char*) 0 )
    {
#endif SERVER_NAME
        //如果没有定义宏 SERVER_NAME_LIST 中, 则采用宏 SERVER_NAME 的定义
        hs->server_hostname = SERVER_NAME;
#else SERVER_NAME
        if ( ghnbuf[0] != '\0' )
            hs->server_hostname = ghnbuf;
#endif /* SERVER_NAME */
    }
}

hs->port = port; //设置 Web 服务器的端口号
if ( cgi_pattern == (char*) 0 ) //设置 Web 服务器的 cgi_pattern 字段
    hs->cgi_pattern = (char*) 0;
else
{
    /* Nuke any leading slashes. */
    if ( cgi_pattern[0] == '/' )
        ++cgi_pattern;
    hs->cgi_pattern = strdup( cgi_pattern );
    if ( hs->cgi_pattern == (char*) 0 )
        //如果该字段设置失败, 则退出程序并记录在系统日志文件中
    {
        syslog( LOG_CRIT, "out of memory copying cgi_pattern" );
        return (httpd_server*) 0;
    }
    /* Nuke any leading slashes in the cgi pattern. */
    while ( ( cp = strstr( hs->cgi_pattern, "/" ) ) != (char*) 0 )
        //返回 “/” 在字段 cgi_pattern 中的位置
        (void) strcpy( cp + 1, cp + 2 );
        //去掉 cgi_pattern 中的 “/”
    }
    hs->cgi_limit = cgi_limit; //设置 Web 服务器中的字段 cgi_limit
    hs->cgi_count = 0; //初始化 Web 服务器中的字段 cgi_count 为 0
    hs->charset = strdup( charset ); //设置 Web 服务器中的字段 charset
    hs->p3p = strdup( p3p );
}

```

```

        //设置 Web 服务器中的字段 p3p
hs->max_age = max_age;           //设置 Web 服务器中的字段 max_age
hs->cwd = strdup( cwd );        //设置 Web 服务器中的字段 cwd
if ( hs->cwd == (char*) 0 )      //字段 cwd 设置失败则退出，并在系统日志中记录
{
    syslog( LOG_CRIT, "out of memory copying cwd" );
    return (httpd_server*) 0;
}
if ( url_pattern == (char*) 0 )   //设置 Web 服务器中的字段 url_pattern
hs->url_pattern = (char*) 0;
else
{
    hs->url_pattern = strdup( url_pattern );
    if ( hs->url_pattern == (char*) 0 )
    {
        syslog( LOG_CRIT, "out of memory copying url_pattern" );
        return (httpd_server*) 0;
    }
}
if ( local_pattern == (char*) 0 )  //设置 Web 服务器中的字段 local_pattern
hs->local_pattern = (char*) 0;
else
{
    hs->local_pattern = strdup( local_pattern );
    if ( hs->local_pattern == (char*) 0 )
    {
        syslog( LOG_CRIT, "out of memory copying local_pattern" );
        return (httpd_server*) 0;
    }
}
/*下面是对 Web 服务器的 no_log、logfp、logfp、vhost、global_passwd、
no_empty_referers 字段的设置*/
hs->no_log = no_log;
hs->logfp = (FILE*) 0;
httpd_set_logfp( hs, logfp );
hs->c = no_symlink_check;
hs->vhost = vhost;
hs->global_passwd = global_passwd;
hs->no_empty_referers = no_empty_referers;

/*下面是对监听套接字初始化，如果优先考虑 IPv6*/
if ( sa6P == (httpd_sockaddr*) 0 )
hs->listen6_fd = -1;
else
hs->listen6_fd = initialize_listen_socket( sa6P );
if ( sa4P == (httpd_sockaddr*) 0 )
hs->listen4_fd = -1;
else
hs->listen4_fd = initialize_listen_socket( sa4P );
/*如果没有得到任何监听套接字，则释放前面分配的空间并退出程序*/
if ( hs->listen4_fd == -1 && hs->listen6_fd == -1 )
{
    free_httpd_server( hs );
}

```

```

    return (httpd_server*) 0;
}
/*初始化 MIME Type，即该资源的媒体类型，浏览器中显示的内容有 HTML、XML、GIF、
Flash、video、vrml 等等，浏览器通过 MIME Type 来区分它们*/
init_mime();

/*Done initializing.*/
if ( hs->binding_hostname == (char*) 0 )
    syslog(
        LOG_NOTICE, "%.80s starting on port %d", SERVER_SOFTWARE,
        (int) hs->port );
else
    syslog(
        LOG_NOTICE, "%.80s starting on %.80s, port %d", SERVER_SOFTWARE,
        /*将 httpd_sockaddr 结构体转化为数字加点组成的 IP 地址字符串*/
        httpd_ntoa( hs->listen4_fd != -1 ? sa4P : sa6P ),
        (int) hs->port );
return hs;
}

```

## 18.2 Thttpd 编译和 HTML 页面测试

第 17 章已经介绍了 BOA 的编译，与 BOA 或其他 Web 服务器类似的是 Thttpd 也有配置文件 thttpd.conf，该文件在 contrib/redhat-rpm 目录下。

### 18.2.1 配置文件介绍

配置文件对于任何 Web 服务器都是非常重要的，Tomcat 是在 Windows 上开发 Web 程序常用的服务器。配置 Tomcat 时有个格式固定的 server.xml 文件，在其中填写对应的内容。Thttpd 也有配置文件 thttpd.conf，其配置也是一样的，和第 17 章 BOA 的配置文件 boa.conf 类似。下面先给出配置文件，在编译的时候如果出现错误，首先对照配置文件进行查找。

配置文件 Thttpd.conf 比较简单，主要指定 HTML 文件路径、日志文件、PID 文件等。配置文件 Thttpd.conf 中注释的部分为默认配置的部分，包括端口号、主机 IP、字符编码格式等信息，具体内容如下：

dir=/home/httpd/html	#html 文件路径
chroot	
user=httpd# default = nobody	#用户名
logfile=/var/log/thttpd.log	#指定日志文件
pidfile=/var/run/thttpd.pid	#指定 PID 文件
# This section _documents_ defaults in effect	
# port=80	#默认的端口号
# nosymlink# default = !chroot	
# novhost	
# nocgipat	#不指定 CGI 程序的路径
# nothrottles	
# host=0.0.0.0	#不设置指本机 IP

```
# charset=iso-8859-1 #默认的编码方式
```

修改该宏定义后，再执行 make 进行编译，在 thttpd-2.25b 目录下生成 thttpd 可执行程序。

## 18.2.2 Thttpd 编译

编译 Thhttpd 的过程与前面介绍的编译方法基本类似，这里按照：编译→遇到问题→修正后重新编译的顺序进行。下面介绍 Thhttpd 编译的详细过程。

(1) 准备 Thhttpd 的源代码。这里使用的源文件为 thttpd-2.25b.tar.gz。

(2) 创建安装目录。在编译安装源代码时，创建自己的安装目录，也可以按照默认的方式安装。笔者认为创建一个安装目录比较合适，安装完成后可以很快知道安装目录下生成哪些工具和哪些库等文件。

```
# mkdir /usr/local/thttpd_x86
```

(3) 解压源代码和编译。解压源码后，进入代码目录使用 configure 命令生成 Makefile 文件。然后进行编译，执行 make install 进行安装。

```
# ./configure --prefix=/usr/local/thttpd_x86  
# make  
# make install
```

在执行 make install 时，会提示子目录安装有问题，不用理会这些错误程序依然可以执行，查看安装目录是否生成所需要的文件。在安装目录/usr/local/thttpd\_x86 下面会生成 3 个文件：sbin 用于放工具文件，man 用于放手册，www 用于放 CGI 和 HTML 文件。下面给出如何解决这些错误的方法。

□ 错误 1：缺少 www 组用户，可以使用 adduser www 增加一个组用户，通过# cat /etc/group 命令显示该文件下多了一个 www 组用户。

```
# adduser www  
# cat /etc/group
```

□ 错误 2：没有使用手册 man1 的路径，这在以前的编译和移植过程中也遇到类似的错误，手动在安装目录下创建一个存放手册文件的路径。

```
# mkdir /usr/local/thttpd_x86/man/man1
```

## 18.2.3 运行和测试 Thhttpd

下面运行生成的 Web 服务器，并通过 HTML 进行测试。本节将介绍 3 部分内容：编写测试主页、运行 Web 服务器，以及通过 HTML 进行测试。

### 1. 编写测试主页index.html

编写 HTML 非常简单，如果读者不想自己动手写代码，可直接打开浏览器，找一个自己喜欢的风格的主页，下载后进行修改。如果会使用 Dreamweaver 更好，如不会使用直接

用 UEdit 或者写字板也可以进行编辑。

**△注意：**修改 HTML 时，直接查找要修改部分的关键字进行修改，查找<title></title>，<head></head>、<body></body>等直接进行内容的修改。

## 2. 启动Web服务器

运行 Thhttpd 服务器时要指定配置文件。配置文件 thhttpd.conf 在 contrib/redhat-rpm 目录下。可以将配置文件复制到 sbin 目录下，也可以在运行 Web 服务器时，指定配置文件的绝对路径。

```
#./thhttpd -C thhttpd.conf
```

或者：

```
#./thhttpd -C /usr/local/thhttpd_x86/thhttpd-2.25b/contrib/redhat-rpm/
thhttpd.conf
```

-C 表示指定配置文件，如果没有带上参数，系统会自动提示读者带上参数，并且给出很多参数的含义。运行的时候会提示没有指定用户 httpd，回头查看配置文件 user=httpd# default = nobody，即程序中默认的 user 为 nobody，目前的 user 指定为 httpd，应该使用 adduser 命令添加一个 user 为 httpd。

```
./thhttpd: unknown user - 'httpd'
```

要解决上面的错误，同样是增加一个 user 名为 httpd。当然也可以修改配置文件，在配置文件中将 user 修改为默认的 user，或者文件/etc/passwd 中存在的 user。

```
# adduser httpd
```

**△注意：**这里暂时不改配置文件，而在实际项目开发中一般是先部署整个网站的框架。即先部署网站必需的文件夹和相关文件，然后再修改配置文件，让配置文件适应实际的部署。

## 3. 测试HTML

在测试主机的浏览器地址栏中输入 <http://192.168.1.123>(服务器的 IP 地址)，通过 HTML 主页访问服务器测试。前面在做 BOA 测试的时候，虚拟机和主机采用 NAT 的方式连接，所以虚拟机和主机不在一个 IP 段也能进行访问。使用 NAT 方式时，不插上网线，也能通过主机对虚拟机进行测试。这次虚拟机和主机采用的是网桥方式，采用这种方式是为了直接将虚拟机和开发板连接起来。测试的时候一定要记得插上网线。

在主机的浏览器地址栏中输入 <http://192.168.1.111>，读者测试时改成自己的虚拟机 IP 地址。如果成功，在主机浏览器中能看到服务器的主页。如果出错，则有两种常见错误。

**□ 错误 1：**回车运行得到无法显示网页。原因是前面修改了 index.html，而没有对它进行部署。

要解决方法：正确部署文件 index.html。首先查看配置文件指定 HTML 路径的设置 dir=/home/httpd/html。可以将 index.html 部署在该目录下，也可以将该配置修改为 dir=

```
/usr/local/thttpd_x86/www。
```

```
# cp index.html /home/httpd/html
```

或者：

```
# cp index.html /usr/local/thttpd_x86/www
```

同时修改配置文件 thttpd.conf，将 dir=/home/httpd/html 改为：dir=/usr/local/thttpd\_x86/www

 注意：服务器修正问题后，保存配置和文件，客户端只要刷新就能获得更新后的信息。

- 错误 2：测试的时候可能会遇到其他问题，导致无法正确显示网页。这时应该查看日志文件/var/log/thttpd.log。从查看配置文件获得日志文件的路径中使用 cat 命令查看日志文件的记录。

```
# cat /var/log/thttpd.log
```

下面是笔者测试出错后在日志中查看到的一条记录，同时在主机的浏览器中得到禁止访问，如图 18.1 所示。

```
192.168.1.104 - - [31/Aug/2013:09:48:14 +0000] "GET / HTTP/1.1" 403 0 ""
"Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko"
```

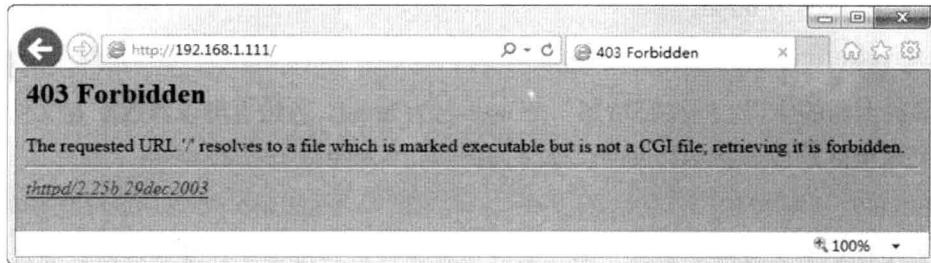


图 18.1 主机浏览器中的显示

错误分析：返回码为“403”，说明服务器已经收到了浏览器的请求，同时出于安全考虑禁止浏览器进行访问。查看 index.html 的执行属性，发现其属性如下所示。

```
-rwxrwxrwx. 1 root root 36 8月 31 17:35 index.html
```

解决方法：修改文件的权限，使其属性为只读，修改命令如下所示。

```
# chmod 444 index.html
```

```
# ls -l
```

```
-r--r--r--. 1 root root 36 8月 31 17:35 index.html
```

修改权限后，能够正确浏览测试页面 index.html，如图 18.2 所示。

- 错误 3：网页中可能会出现乱码。

解决方法：修改配置文件的中字符编码方案，将 charset 设置为 utf-8。笔者做过试验将 charset 设置为 gbk 和 gb2312 仍不能解决中文乱码问题。设置为 utf-8 后显示中文正常，如图 18.3 所示。

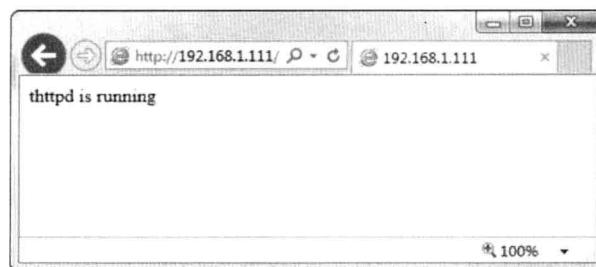


图 18.2 正确的测试主页

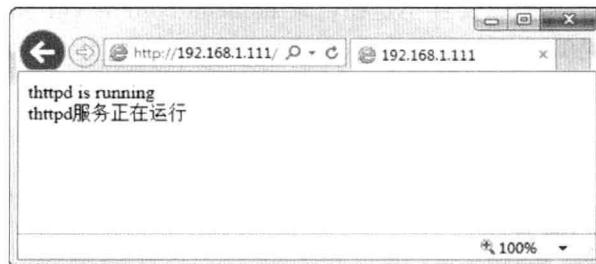


图 18.3 正确显示中文

### 18.3 CGI 脚本测试

下面测试 CGI 脚本程序。对 CGI 程序进行测试时，首先应该修改配置文件，指定 CGI 程序的路径。CGI 脚本的测试也包括 3 个部分：编写测试代码、编译测试代码、执行测试。

#### 18.3.1 编写测试代码

CGI 的文件应该放在目录 /home/httpd/html/cgi-bin 下，同时修改 thhttpd.conf 中的 nocgipat 为 cgipat。

```
cgipat=/cgi-bin
```

在 /home/httpd/html/cgi-bin 目录下编写 hello.c 文件，该测试文件内容为打印“Hello,World.”。测试文件的代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    printf("Content-type: text/html\n\n");
    printf("<html>\n");
    printf("<head><title>CGI Test For Thhttpd</title></head>\n");
    printf("<body>\n");
    printf("<h1>Hello,world.</h1>\n");
    printf("</body>\n");
}
```

```

    printf("</html>\n");
    exit(0);
}

```

### 18.3.2 编译测试程序

将 hello.c 文件编译生成 hello.cgi 文件。编译命令如下：

```
# gcc -o hello hello.c
```

编译生成 hello 后，如果该文件不是在目录/home/httpd/html/cgi-bin 下，则将 hello 文件复制到该目录下。如果编译的文件名为 hello.cgi，在测试的过程中，就会出现下载该文件的情况。编译好的 CGI 文件同样需要将其权限修改为只读。

```
# chmod 444 hello
```

### 18.3.3 测试 CGI 脚本

打开客户端的浏览器，在浏览器中输入下面地址进行访问：

```
http://192.168.1.111/cgi-bin/hello
```

hello 的权限要设置为可读方式，否则无法正确显示。

## 18.4 Thttpd 交叉编译与移植

本节将介绍如何在嵌入式产品中应用 Thttpd，在嵌入式产品中使用 Thttpd 需要对其进行交叉编译、配置、编写 HTML 页面、编写 CGI、部署上述文件到相应的目录。

### 18.4.1 交叉编译 Thttpd

进入 Thttpd 源码的解压目录对 Thttpd 进行交叉编译，这里使用的交叉编译器为 arm-linux-gcc-4.4.3。编译的过程如下所述。

(1) 在上位机中调试的时候已经通过 configure 命令生成了 Makefile 文件，这里只需要对生成的 Makefile 文件进行修改。在 Makefile 文件中将 gcc 改为 arm-linux-gcc，将安装目录指定为/usr/local/thttpd\_arm，同时也建立这样一个目录。

```

# mkdir /usr/local/thttpd_arm
#vi Makefile
CC=arm-linux-gcc
prefix=/usr/local/thttpd_arm
# make clean           //之前进行了x86平台的编译，清除编译生成的文件
# make

```

(2) 编译完成后进行 make install 安装，安装之前需进行一些设置，这与在 X86 平台

上编译安装的过程相同。

```
# mkdir /usr/local/thhttpd_arm/man/man1
# make install
```

### 18.4.2 交叉编译 CGI 程序

测试程序包括测试的 HTML 页面和 CGI 程序，HTML 和 CGI 程序，使用在上位机中测试的程序。这里只需要对 CGI 程序进行交叉编译即可以使用在 ARM 平台上。

```
# arm-linux-gcc -o hello hello.c
```

同样，执行完成后对生成的 hello 文件查看其文件信息，同时修改其权限。

```
# chmod 444 hello
```

### 18.4.3 移植 Thhttpd

在开发板上部署 Web 服务器。部署的文件包括：部署配置信息 thhttpd.conf、部署服务器程序 Thhttpd、部署访问页面和 CGI 程序及相关目录，移植 Thhttpd 需要的库。

(1) 部署配置信息 thhttpd.conf，复制上位机/etc/thhttpd.conf 到开发板的/etc 目录下。配置文件的信息如下：

```
dir=/home/httpd/html
chroot
user=httpd# default = nobody
logfile=/var/log/thhttpd.log
pidfile=/var/run/thhttpd.pid
# This section _documents_ defaults in effect
# port=80
# nosymlink# default = !chroot
# novhost
cgipat=/cgi-bin/*
# nothrottles
# host=0.0.0.0
charset=utf-8
```

(2) 部署服务器程序 Thhttpd，复制上位机/usr/local/thhttpd\_arm/sbin 目录下的 thhttpd 到开发板/usr/sbin 目录下。

(3) 在开发板上也建立目录/home/httpd/html，同时将 index.html 部署在该目录下，在该目录下建立目录 cgi-bin，将交叉编译好的 CGI 程序放置在该目录下。

(4) 复制 Thhttpd 依赖的库文件，可以对 X86 版的 thhttpd 使用 ldd 命令查看其依赖的库文件。从交叉编译路径下复制这些库到开发板的 lib 库下。

```
# ldd thhttpd
    linux-gate.so.1 => (0x00b96000)
    libcrypt.so.1 => /lib/libcrypt.so.1 (0x45231000)
    libc.so.6 => /lib/libc.so.6 (0x44588000)
    /lib/ld-linux.so.2 (0x43bb9000)
```

查看开发板/lib 目录，存在上述文件。依赖库文件就不需要复制。

### 18.4.4 测试

移植到开发板上时，主要是配置内核、添加用户、修改权限等问题。读者在开发板上测试时还可能会发生其他问题。如果遇到其他问题就查看日志/var/log/messages。

(1) 内核支持 IPv6，在内核中添加对 IPv6 的支持，如图 18.4 所示。

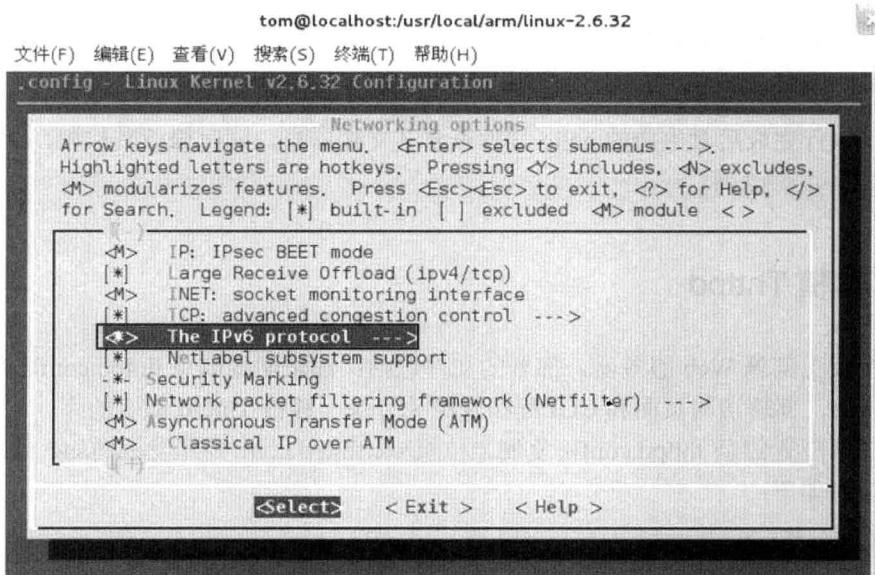


图 18.4 内核支持 IPv6

(2) 修改 thttpd 的执行权限。

```
# chmod 777 thttpd
```

(3) 增加用户 thttpd。

```
# adduser thttpd
```

**注意：**移植到 mini2440 上时，自带的文件系统默认启动 BOA Web 服务，去掉开发板自带的文件系统的/etc/init.d/rcS 中运行的 BOA 代码，然后重新启动。

(4) 启动 Web 服务器，并通过主机进行测试。测试结果和主机上一样。

## 18.5 Thttpd 与嵌入式数据库结合

在前面章节中，已经介绍了嵌入式 Web 服务器 BOA 与 SQLite 在项目中结合使用的情况。本节将以实例介绍通过 Thttpd Web 服务 CGI 接口维护和管理嵌入式数据库。

### 18.5.1 通过 CGI 程序访问 SQLite

SQLite 提供了 C 语言访问的接口。通过采用 C 语言程序访问数据库，将该访问数据库的操作编译成 CGI 程序部署在 Thttpd 的 CGI 路径下，远程维护人员通过调用此 CGI 程序就能实现远程维护数据库的目的。

下面是例子程序，其代码主要分为两部分：一部分是通过调用 C 接口对数据库进行创建、修改等维护工作；另一部分是通过 HTML 页面将结果返回给远程访问者。下面的程序是在第 17 章代码的基础上稍加改动。具体代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

int main( void )
{
    sqlite3 *db=NULL;
    char *zErrMsg = 0;
    int rc;
    int i=0;
    //通过 CGI 将结果返回给远程操作者
    printf("<%@ page contentType=\"text/html; charset=utf-8\"%>"); //设置编码方案，解决中文乱码
    printf("<html>\n");
    printf("<head><title>CGI Output</title></head>\n");
    printf("<body>\n");
    printf("<h1>Access SQLite Database by CGI of Thttpd</h1>\n");
    rc = sqlite3_open("test.db", &db); //打开指定的数据库文件，如果不存在将创建一个同名的数据库文件
    if( rc )
    {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }
    else
        printf("opened database test.db successfully!\n");

    //创建一个表，如果该表存在，则不创建，并给出提示信息，存储在 zErrMsg 中
    char *sql = "create table student(id, name , sex, age);" ;
    //执行 SQL 语句
    sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );

    //插入数据
    sql = "insert into student values(1, 'Jack', 'M', 20);";
    //执行 SQL 语句
    sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );
    //插入数据
    sql = "insert into student values(2, 'Tom', 'M', 21);";
    sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );
    //插入数据
    sql = "insert into student values(3, 'Mary', 'W', 19);";
    sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );
    //在 HTML 中显示操作的内容
```

```

printf("在表 student 中插入数据项(1, 'Jack', 'M', 20);";
printf("<p>\n");
printf("在表 student 中插入数据项(2, 'Tom', 'M', 21);";
printf("<p>\n");
printf("在表 student 中插入数据项(3, 'Mary', 'W', 19);";
printf("<p>\n");

//查询结果
sql = "select * from student;" ;
sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );

//查询数据
/*
int sqlite3_get_table(sqlite3*, const char *sql,char***result , int
*nrow , int *ncolumn ,char **errmsg );
result 中是以数组的形式存放所查询的数据，首先是表名，再是数据。
nrow ,ncolumn 分别为查询语句返回的结果集的行数，列数，没有查到结果时返回 0
*/
int nrow = 0, ncolumn = 0;
char **fristResult;                                //二维数组存放结果

sql = "select * from student;" ;
printf("\n");
sqlite3_get_table( db , sql , &fristResult , &nrow , &ncolumn , &zErrMsg );

//打印查询结果
printf( "row: %d col: %d \n" , nrow , ncolumn );
printf( "\n更新前数据库的数据: \n" );

for( i=0 ; i<( nrow + 1 ) * ncolumn ; i++ ){
if (i%4==0) printf("<p>\n");
printf( "%s\t", i , fristResult[i] );
}

//释放 fristResult 的内存空间
sqlite3_free_table( fristResult );

sql = "update student set age = 24 where age = 20;" ;
printf("\n");
sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );

nrow = 0;
ncolumn = 0;
char **secondResult;                                //二维数组存放结果
//查询数据
sql = "select * from student;" ;
sqlite3_get_table( db , sql , &secondResult , &nrow , &ncolumn ,
&zErrMsg );
printf("<p>\n");
//打印查询结果
printf( "row:%d column=%d \n" , nrow , ncolumn );
printf( "\n更新后的数据库的结果: \n" );

for( i=0 ; i<( nrow + 1 ) * ncolumn ; i++ ) {
if (i%4==0) printf("<p>\n");
printf( "%s\t", i , fristResult[i] );
}

```

```

printf("<body>\n");
printf("</html>\n");
//释放 secondResult 的内存空间
sqlite3_free_table( secondResult );

printf("\n");
sqlite3_close(db); //关闭数据库
return 0;
}

```

### 18.5.2 编译和测试

与 BOA 中大致相同，编译 SQLite 程序时，需要 SQLite 接口的头文件和库文件支持。测试前对 CGI 进行编译和部署。

#### 1. CGI 程序的编译和部署

将上述代码起名为 sqlite.c，放在 Thttpd 服务器的 CGI 路径下，查看 Thttpd 的配置文件/etc/thttpd.conf，对于本机的路径，CGI 文件的路径设置为/home/httpd/html/cgi-bin。编译时需指定依赖的头文件和库文件，具体的编译命令如下：

```
#gcc -o sqlite -I /usr/local/sqlite_x86/include -L /usr/local/sqlite_x86/lib sqlite.c -lsqlite3 -static -lpthread
```

/usr/local/sqlite\_x86 是本机 SQLite 的安装目录。编译完成后在/home/httpd/html/cgi-bin 目录下生成 sqlite 文件，修改其属性为只读，同时也完成了部署。

```
#chmod 444 sqlite
```

 注意：对于 Thttpd 的 CGI 程序，一般命名不带后缀.cgi。命名采用带后缀名的 CGI 程序时，测试的时候，将会得到通过网页下载该文件，而并非是在网页中显示。

#### 2. 测试sqlite

启动 Thttpd Web 服务，在 Windows 的浏览器中输入 <http://192.168.1.111/cgi-bin/sqlite>，来访问虚拟机（192.168.1.111 是虚拟机的 IP 地址）下的 CGI 程序。

```
#./thttpd -C/etc/thttpd.conf //启动 Thttpd Web 服务
```

 注意：前面已经介绍 Thttpd 的移植过程，对于和数据库结合移植的过程留给读者完成。

## 18.6 小结

除了 Thttpd、Boa 外，还有很多嵌入式 Web 服务器，其工作流程及移植过程基本与 Boa、Thttpd 类似。读者可以试着去将前面介绍的技术与之结合起来应用在实际的开发中，如和 GUI 数据库的结合应用到自己的嵌入式项目中。

# 第 19 章 JVM 及其移植

JVM (Java Virtual Machine, Java 虚拟机), 是虚拟出来的计算机, 在实际的计算上通过软件模拟出各种硬件的功能, 如处理器、堆栈、寄存器、指令系统等。JVM 屏蔽了具体平台的信息, 使得 Java 程序能运行在各种安装了 JVM 的平台上。本章内容主要包括 JVM 原理、作用、移植及编程。在介绍移植实例前还会对 Java 程序进行简单分析。

## 19.1 JVM 介绍

JVM 是一种用于计算设备的规范, 可采用不同的方式 (软件或硬件) 实现。编译虚拟机的指令集与编译微处理器的指令集基本类似。Java 虚拟机的组成部分包括一套字节码指令集、一组寄存器、一个栈、一个垃圾回收堆、一个存储方法域和一个执行引擎。Java 虚拟机 (JVM) 是可运行 Java 代码的虚拟计算机。按照 JVM 规格描述将解释器移植到特定的计算机上, 就能保证经过编译的任意 Java 代码都能够正确运行在该系统上。Java 虚拟机是一个虚拟出来的机器, 通过在实际计算机上采用软件模拟实现。Java 虚拟机模拟的硬件包括处理器、堆栈、寄存器、指令系统等。

### 19.1.1 JVM 原理

Java 语言的特点就是平台无关性, 这一特性主要是通过 Java 虚拟机来实现的。其他高级语言如果要在不同的平台上运行, 至少应该重新编译成不同的目标代码。而使用 Java 虚拟机后, Java 语言在不同平台上运行时不需要重新编译。Java 程序经过编译后运行在 Java 虚拟机上, 屏蔽了与具体平台相关的信息, 使得 Java 语言编译程序只需生成在 Java 虚拟机上运行的目标代码 (字节码), 就可以在不同平台 (安装了 JVM) 上不加修改地运行。Java 虚拟机在执行字节码时, 将字节码解释成对应平台上的机器指令进行执行。下面给出 JVM 原理图, 如图 19.1 所示。后面将会对其实现细节进行介绍。

JVM 生命周期开始于运行 Java 程序, 消亡于 Java 程序的关闭退出。Java 虚拟机实例通过调用任意某个初始类的 main() 方法运行一个 Java 程序。只要还有任何非守护线程在运行, 那么这个 Java 程序也在继续运行 (虚拟机仍然存活)。

Sun 公司提供了 3 种运行在小型设备操作系统上的 JVM, 分别为 CVM、KVM 和 Card VM, 这 3 种 JVM 有不同的应用。

- CVM: 应用于瘦客户端, 如数字机顶盒、车载电子系统等;
- KVM: 应用于电池供电的手持移动设备, 如移动电话、PDA 等;
- Card VM: 应用于智能卡 (Smart Card) 系统。

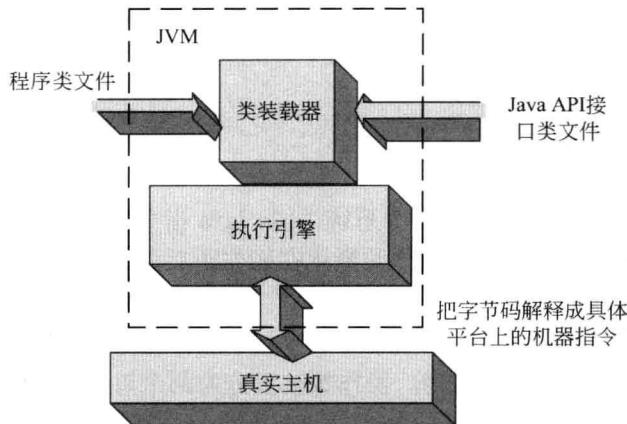


图 19.1 JVM 原理图

CVM、KVM 和 Card VM 适用的硬件资源是由高到低的，根据不同的硬件选择不同的虚拟机。KVM 完成的功能是 CVM 完成功能的子集。CVM 允许设备将 Java 线程映射为本地线程，完成垃圾收集、Java 同步等。在存储系统方面，表现为精确、高效的垃圾收集，虚拟机与存储系统分离；在可移植性方面，CVM 使用 C 语言，实现快速、安全地移植；在本地线程方面，CVM 支持线程抢占。

而 KVM 的最大特点是小而高效，只需要几万字节的存储空间就可以运行。虚拟机和类库的大小约 50~80KB，具有较高的可移植性和可扩展性，垃圾收集独立于系统，支持多线程。在后面的介绍中，多以 KVM 为例讲解。

### 19.1.2 JVM 支持的数据类型

Java 虚拟机不仅支持 Java 语言的基本数据类型，而且支持其他数据类型。Java 支持的基本数据类型类似 C 语言中的 int、long 等，支持的数据结构如表 19.1 所示。

表 19.1 JVM 支持的数据结构

分 类	数 据 类 型	说 明
基本的数据类型	byte	1 字节有符号整数的补码
	short	2 字节有符号整数的补码
	int	4 字节有符号整数的补码
	long	8 字节有符号整数的补码
	float	4 字节 IEEE754 单精度浮点数
	double	8 字节 IEEE754 双精度浮点数
	char	2 字节无符号 Unicode 字符
其他数据类型	object	4 字节，对一个 Javaobject（对象）的引用
	returnAddress	4 字节，用于 jsr/ret/jsr-w/ret-w 指令

应用程序为了能够编译成为 Java 虚拟机的字节码且正确执行，就必须遵守 JVM 支持的类型规定。Java 虚拟机不执行违反了类型规定的字节码程序。从 Java 虚拟机支持的数据

类型看, Java 对数据类型的内部格式进行了严格规定, 这样使得各种 Java 虚拟机的实现对数据的解释是一致的, 从而保证了 Java 的与平台无关性和可移植性。

### 19.1.3 JVM 指令系统

JVM 指令系统与其他计算机的指令系统类似。Java 指令也是由操作码和操作数两部分组成的。操作码是 8 位二进制数, 操作数紧跟在操作码后面, 其长度根据需要而不同。操作码指定一条指令操作的功能, 如 `iload` 表示从存储器中装载一个整型数, `anewarray` 表示给一个新数组分配空间, `iand` 表示两个整数的“与”操作, `ret` 用于流程控制, 表示从某方法的调用中返回。当长度大于 8 位时, 操作数被拆分为两个以上字节进行存放。JVM 采用了“big endian”的编码方式, 即低位 bits 存放在高字节, 高位 bits 存放在低字节。

Java 指令系统是专门为 Java 语言的实现而设计的, 还包括用于调用方法和监视多线程系统的指令。Java 的操作码长度为 8 位, 使得 JVM 最多有 256 种指令, 目前已使用了其中的 160 多种操作码。

### 19.1.4 JVM 寄存器

所有的 CPU 均包含用于保存系统状态和处理器所需信息的寄存器组。如果虚拟机定义较多的寄存器, 便可以从寄存器中得到更多的信息而不必对栈或内存进行访问, 这有利于提高系统的运行速度。然而, 如果虚拟机中的寄存器比实际 CPU 的寄存器还多, 那么在实现虚拟机时就会占用处理器大量的时间来用常规存储器模拟寄存器, 这反而会降低虚拟机的效率。因此, JVM 只设置了以下 4 个最为常用的寄存器。

- `pc`: 程序计数器。
- `optop`: 操作数栈顶指针。
- `frame`: 当前执行环境指针。
- `vars`: 指向当前执行环境中第一个局部变量的指针。

说明: 所有寄存器均为 32 位。`pc` 用于记录程序的执行, `optop`、`frame` 和 `vars` 用于记录指向 Java 栈区的指针。

### 19.1.5 JVM 栈结构

作为基于栈结构的计算机, Java 栈是 JVM 存储信息的主要手段。当 JVM 获得一个 Java 字节码应用程序后, 便为该代码中一个类的每一个方法创建一个栈框架, 以保存该方法的状态信息。每个栈框架包括 3 类信息: 局部变量、执行环境和操作数栈。在线程的执行栈中分配帧, 下面为 KVM 帧结构体 (下面例子都以 JVM 中的 KVM 为例介绍) :

```
struct frameStruct {
    FRAME    previousFp;           // 保存前向帧指针
    BYTE*   previousIp;           // 保存前向的程序计数器
    cell*   previousSp;           // 保存前向的栈指针
```

```

METHOD  thisMethod;          //指向当前执行的方法
STACK    stack;              //stackStruct 类型的指针
OBJECT   syncObject;         //objectStruct 类型指针
};

}

```

在必要的时候，JVM stack 可以动态增加和减小。每个线程都拥有一个私有的 JVM stack，这个堆栈与线程一同创建。JVM 栈和 C 语言的栈结构相似。由于 JVM 的帧可以存放在堆上，所以 JVM stack 可以是不连续的。JVM 的设计者让程序员可以控制初始栈的大小，并控制栈的最大值和最小值。下面为 KVM 栈结构体：

```

struct stackStruct {
    STACK    next;           //指向下一个 stackStruct 指针，即该结构为链表
    short    size;
    short    xxunusedxx;     /*must be multiple of 4 on all platforms*/
    cell     cells[STACKCHUNKSIZE];
};

}

```

对于栈的操作一般有 push、pop 等操作，下面为 KVM 栈操作函数。

### 1. PushFrame()函数操作

函数 pushFrame()为当前执行线程创建一个新的栈帧。在分配帧前，所有的寄存器都必须被正确地初始化。KVM 与其他 JVM 不同，它不为局部函数调用创建栈帧。

```

void pushFrame(METHOD thisMethod)
{
    int thisFrameSize = thisMethod->frameSize;
    int thisArgCount = thisMethod->argCount;
    int thisLocalCount = thisFrameSize - thisArgCount;

    STACK stack = getFP() ? getFP()->stack : CurrentThread->stack;
    int thisMethodHeight = thisLocalCount + thisMethod->u.java.maxStack +
                           SIZEOF_FRAME + RESERVEDFORNATIVE;
    FRAME newFrame;
    int i;
    cell* prev_sp = getSP() - thisArgCount; /* Very volatile! */

    /*检查当前的栈块中是否有足够的空间*/
    if (getSP() - stack->cells + thisMethodHeight >= stack->size) {
        /*没有足够的栈块空间时，需要重新创建一个新的栈块或者重新使用已分配的一个栈块*/
        STACK newstack;                                //重新创建新栈块
        thisMethodHeight += thisArgCount;
        /*检查是否可以重新使用已分配的一个栈块*/
        if (stack->next && thisMethodHeight > stack->next->size) {
            stack->next = NULL;
        }
        /*如果下一栈为空，则需要重新创建栈块*/
        if (stack->next == NULL) {
            int size = thisMethodHeight > STACKCHUNKSIZE ? thisMethodHeight :
                       STACKCHUNKSIZE;
            int stacksize = sizeof(struct stackStruct) / CELL + (size -
                           STACKCHUNKSIZE);
            START_TEMPORARY_ROOTS
            DECLARE_TEMPORARY_ROOT(STACK, stackX, stack);
            newstack = (STACK)mallocHeapObject(stacksize,
                                              GCT_EXECSTACK);
            stack = stackX;
        }
    }
}

```

```

        prev_sp = getSP() - thisArgCount;
END_TEMPORARY_ROOTS
if (newstack == NULL) {
    THROW(StackOverflowObject);
}
} else {
    /*可以使用一个存在的、未使用的栈块*/
    newstack = stack->next;
}
/*将新分配的栈加入到栈链表中*/
for (i = 0; i < thisArgCount; i++) {
    newstack->cells[i] = prev_sp[i + 1];
}
setLP(newstack->cells);
newFrame = (FRAME)(getLP() + thisFrameSize);
newFrame->stack = newstack;
} else {

ASSERTING_NO_ALLOCATION
/*设置开始指针指向执行栈中的局部变量*/
setLP(prev_sp + 1);
/*为局部变量增加空间，并初始化新的帧指针*/
newFrame = (FRAME)(getSP() + thisLocalCount + 1);
newFrame->stack = stack;
END_ASSERTING_NO_ALLOCATION
}
/*填充栈帧中的 remaining 域*/
ASSERTING_NO_ALLOCATION
/*Initialize info needed for popping the stack frame later on*/
newFrame->previousSp = prev_sp;
newFrame->previousIp = getIP();
newFrame->previousFp = getFP();

/*Initialize the frame to execute the given method*/
newFrame->thisMethod = thisMethod;
newFrame->syncObject = NIL; /* Initialized later if necessary*/

/*Change virtual machine registers to execute the new method*/
setFP(newFrame);
setSP((cell*)(newFrame + 1) - 1);
setIP(thisMethod->u.java.code);
setCP(thisMethod->ofClass->constPool);

END_ASSERTING_NO_ALLOCATION
}

```

## 2. popFrame操作

函数 popFrame() 用于删除一个执行帧栈及调用当前执行方法时重启执行方法。

```

void popFrame()
{
    /*复位虚拟机寄存器去继续前向方法的执行*/
    POPFRAMEMACRO
}

```

### 19.1.6 JVM 碎片回收堆

垃圾回收是 Java 语言最重要的特点之一。在 Java 语言中，除了 new 语句外没有其他方法为对象申请和释放内存。Java 程序中对内存进行释放和回收的工作是由运行系统 Java 承担的，即 JVM 来完成。在 Sun 公司开发的 Java 解释器和 Hot Java 环境中，采用后台线程来执行碎片回收。这样不仅提高了运行系统的性能，而且使程序设计人员摆脱了动态管理内存的风险。

JVM 有两种类型的存储区：常量缓冲区和方法区。其中常量缓冲区用于存储类名称、方法和字段名称，以及串常量，方法区则用于存储 Java 方法的字节码。对于这两种存储区域具体实现方式，在 JVM 规格中没有明确规定。因此 Java 应用程序的存储布局只能在运行过程中确定，完全依赖于具体平台的实现方式。

### 19.1.7 JVM 异常抛出和异常捕获

异常抛出会使当前方法异常结束。将类的异常 Handler 放在类文件的一个表中，当异常发生时，JVM 会从存放异常 Handler 的表中找到合适的异常处理执行，如果当前方法没有合适的处理对应当前异常 Handler，则将当前方法的 Frame 弹出，扔掉 Operand stack 和局部变量，返回到当前方法的调用者中，再重复前面的过程，直到到达调用链表的顶端。如果最外层的方法也没有合适的 Handler，就退出当前线程。

当发生异常时，Java 虚拟机采取进行异常捕获的细节如下所述。

- 检查与当前方法关联的 catch 子句表，找到与异常匹配的 catch 子句。每个 catch 子句包含其有效指令范围、能够处理的异常类型，以及处理异常的代码块地址。
- 与异常相匹配的 catch 子句应该符合下面的条件：造成异常的指令在其指令范围之内，发生的异常类型是其能处理的异常类型的子类型。如果找到了与该异常匹配的 catch 子句，那么系统转移到异常处理块处进行处理；如果没有找到异常处理块，重复寻找与该异常匹配的 catch 子句的过程，直到检查当前方法的所有嵌套的 catch 子句。
- 由于虚拟机从首个匹配 catch 子句处继续执行，所以 catch 子句表中的顺序是很重要的。Java 代码为结构化的，因此总可以把某个方法的所有的异常处理都按顺序排列在一个表中，对任何可能的程序计数器的值，都能按线性的顺序找到合适的异常处理块，以处理在该程序计数器值下发生的异常情况。
- 如果找不到匹配的 catch 子句，那么当前方法得到一个“未截获异常”的结果并返回到当前方法的调用者，就像异常刚刚在其调用者中发生一样。如果在调用者中仍然没有找到相应的异常处理块，那么这种错误将被传播下去。如果最终错误被传播到最顶层，那么系统最后将调用默认的异常处理块。

KVM 是 JVM 的一种实现，多用于移动电话、PDA 等。KVM 相对其他两种虚拟机最明显的特点是小而高效，比较适合用于嵌入式环境中。接下来将介绍 KVM 实现 JVM 主要功能的细节。

## 19.2 类 装 载

class 文件成为虚拟机上运行的 Java 程序的一部分，要经过 3 个步骤：装载→连接→初始化。下面将介绍装载类的结构体和相关的主要操作。

### 19.2.1 装载类的结构体

在实现类装载时，KVM 定义了几个数据结构 struct classStruct、struct instanceClassStruct 和 struct arrayClassStruct，这几个数据结构用来保存类的相关信息。

结构体 classStruct 如下：

```
struct classStruct {
    COMMON_OBJECT_INFO(INSTANCE_CLASS)
    UString packageName;           /*最后符号'/'之前的所有内容*/
    UString baseName;             /*最后符号'/'之后的所有内容*/
    CLASS next;                  /*哈希表的下一项*/

    unsigned short accessFlags;   /*访问信息*/
    unsigned short key;          /*类关键字*/
};

typedef struct classStruct* CLASS;
```

结构体 instanceClassStruct 定义如下：

```
struct instanceClassStruct {
    struct classStruct clazz;      /*公共信息*/

    /* instance classes 的专有信息 */
    INSTANCE_CLASS superClass;     /*超类对象*/
    CONSTANTPOOL constPool;        /*常量池指针*/
    FIELDTABLE fieldTable;         /*实例变量表指针*/
    METHODTABLE methodTable;       /*虚拟方法表指针*/
    unsigned short* ifaceTable;    /*接口表指针*/
    POINTERLIST staticFields;      /*保持类的静态域*/
    short instSize;                /*类的实例大小*/
    short status;                  /*类的就绪状态*/
    THREAD initThread;             /*类的初始化线程*/
    NativeFuncPti finalizer;      /*finalizer 指针*/
};

typedef struct instanceClassStruct* INSTANCE_CLASS;
```

结构体 classStruct 和 instanceClassStruct 都是与 class（可能包括类和接口）有关的信息。不同的是，classStruct 所提供的是一些“外围信息”，包括访问信息和可见性等，是一个 class 区分其他 class 的基本信息；instanceClassStruct 所提供的是一些“内容信息”，是一个类本身所定义的内容，比如方法表、字段表等。

结构体 arrayClassStruct 的定义如下：

```
struct arrayClassStruct {
    struct classStruct clazz;           /*公共信息*/
    /*数组类成员专有信息*/
    union {
        CLASS elemClass;      /*数组对象成员类*/
        long primType;       /*原数组的成员类型*/
    } u;
    long itemSize;                   /*单个成员的大小*/
    long gcType;                    /*垃圾收集的类型, GCT_ARRAY 或 GCT_OBJECTARRAY*/
    long flags;
};
typedef struct arrayClassStruct* ARRAY_CLASS;
```

## 19.2.2 装载类的操作

上面介绍了类装载保存的结构体，下面介绍装载类的几个操作函数 LoadClassLocally()、LoadClassFromFile()、LoadClassFromZip()。这 3 个函数分别从不同的方式装载类。

### 1. 函数 LoadClassFromFile()

函数 LoadClassFromFile() 从磁盘装载一个解析器要求的类文件 (.class 格式文件)，通过创建一个类块结构体装载编译好的类。

```
static ClassClass * LoadClassFromFile(char *fn, char *dir, char *class_name)
{
    extern int OpenCode(char *, char *, char *, struct stat*);
    struct stat st;
    ClassClass *cb = 0;
    int codefd = -1;
    unsigned char *external_class;
    char *detail;

    codefd = OpenCode(fn, NULL, dir, &st);           /*打开一个.class 文件*/
    /*打开失败则退出*/
    if (codefd < 0)                                /*打开失败*/
        return 0;
    /*将文件载入内存*/
    external_class = (unsigned char *)sysMalloc(st.st_size);
    if (external_class == 0)
        goto failed;
    /*系统 API, 读取 codefd 中的内容到 external_class , 大小为 st.st_size , 如果读
    取的大小不等于 st.st_size , 则失败, 程序跳转到 failed 处.*/
    if (sysRead(codefd, external_class, st.st_size) != st.st_size)
        goto failed;
    sysClose(codefd);                            /*系统 API 关闭 codefd*/
    codefd = -1;
    cb = allocClassClass();           /*创建内部类*/
    if (cb == NULL ||
        !createInternalClass(external_class, external_class + st.st_size,
            cb, NULL, class_name, &detail)) {
        sysFree(external_class);          /*创建失败则释放 external_class*/
        goto failed;
    }
}
```

```

}
    sysFree(external_class);           /*系统 API 释放 external_class*/
return cb;
/*打开.class 文件出错时，关闭 codefd，创建内部类出错时，释放 cb*/
failed:
if (codefd >= 0)
    sysClose(codefd);
if (cb != 0)
    FreeClass(cb);
return 0;
}

```

## 2. 函数LoadClassFromZip()

函数 LoadClassFromZip() 通过 zip 格式文件装载类。函数 LoadClassFromFile() 与 LoadClassFromZip() 基本类似，只是两者读取的文件格式不同而已，前者为.class 格式，后者为.jar 格式。

```

static ClassClass * LoadClassFromZip(zip_t *zipEntry, char *class_name)
{
    ClassClass *cb = 0;
    JAR_DataStreamPtr jdstream = NULL;
    unsigned char *external_class;
    int data_length;
    char *detail;
    jdstream = loadJARfile (zipEntry, class_name); /*导入.jar格式文件*/
    /*失败则程序跳转到 failed 处*/
    if (jdstream == NULL)
        goto failed;
    /*指定导入的数据和长度*/
    external_class = jdstream->data;
    data_length = jdstream->dataLen;
    cb = allocClassClass();                      /*创建内部类*/
    if (cb == NULL ||
        !createInternalClass(external_class, external_class + data_length,
                             cb, NULL, class_name, &detail)) {
        goto failed;
    }
    if (jdstream != NULL) {
        /*导入.jar格式文件失败，则释放 jdstream*/
        freeBytes(jdstream);
    }
    return cb;
    /*创建内部类失败，则释放 cb；导入.jar格式文件失败，则释放 jdstream*/
failed:
    if (cb != 0)
        FreeClass(cb);
    if (jdstream != NULL) {
        freeBytes(jdstream);
    }
    return 0;
}

```

## 3. 函数LoadClassLocally()

函数 LoadClassLocally()，输入参数为指向类文件的本地路径，其中会调用函数

LoadClassFromFile()。

```

ClassClass *LoadClassLocally(char *name)
{
    ClassClass *cb = 0;
    cpe_t **cpp;
    /*对路径格式进行检验*/
    if (name[0] == DIR_SEPARATOR || name[0] == SIGNATURE_ARRAY)
        return 0;
    /*获取系统环境的 classpath, 通过调用系统函数 getenv()*/
    for (cpp = sysGetClassPath(); cpp && *cpp != 0; cpp++) {
        cpe_t *cpe = *cpp;
        char *path;
        /*classpath 元素类型为一个路径或 zip 文件*/
        if (cpe->type == CPE_DIR) {
            path = (char *)sysMalloc(strlen(cpe->u.dir)
                + sizeof(LOCAL_DIR_SEPARATOR)
                + strlen(name)
                + strlen(JAVAOBJEXT)
                + 2); /*2 is for the . and the \0*/
            if (sprintf(path,
                "%s%c%s." JAVAOBJEXT, cpe->u.dir,
                LOCAL_DIR_SEPARATOR, name) == -1) {
                sysFree(path);
                return 0;
            }
            /*调用函数 LoadClassFromFile ()装载类*/
            if ((cb = LoadClassFromFile(sysNativePath(path),
                cpe->u.dir, name))) {
                sysFree(path);
                return cb;
            }
        } else if (cpe->type == CPE_ZIP) {
            if (JAR_DEBUG && verbose)
                jio_fprintf(stderr, "Loading classes from a ZIP file... \n");
            /*如果为 zip 文件, 则调用函数 LoadClassFromZip ()装载类*/
            if ((cb = LoadClassFromZip(cpe->u.zip, name))) {
                return cb;
            }
        }
    }
    return cb;
}

```

### 19.3 垃圾回收

垃圾回收是 Java 语言的一个特点, 这一特点正是 JVM 的设计者通过内存管理来实现的。内存管理分为内存分配和垃圾回收两部分, 垃圾回收的技术和策略会严重影响系统的效率, 因此垃圾回收是 JVM 设计的重点。下面讲解 KVM 中采用的垃圾回收策略。

### 19.3.1 mark-and-sweep 回收算法

KVM 中的 mark-and-sweep 回收算法分为两个阶段，第一阶段为 mark 阶段，垃圾收集器从 root set 开始搜索，标记每个可达的对象。第二个阶段为 sweep 阶段，垃圾收集器从内存空间的起始地址往后查找，回收那些没有在第一阶段标记的对象所占有的空间，回收的空间加入到内存可用列表中。其实现细节如下所述。

#### 1. 垃圾收集函数garbageCollect()

在函数 garbageCollect() 中实现 mark-and-sweep 回收算法。mark-and-sweep 回收算法核心部分为 garbageCollectForReal()，函数 garbageCollect() 首先保存虚拟机活动线程调用 garbageCollectForReal() 执行垃圾回收，然后恢复环境。

```
void garbageCollect(int moreMemory)
{
    if (gcInProgress != 0) {
        /*不允许循环调用垃圾回收操作*/
        fatalVMEError(KVM_MSG_CIRCULAR_GC_INVOCATION);
    }
    gcInProgress++;
    /*等待所有的异步 I/O 完成*/
    RundownAsynchronousFunctions();
    if (ENABLEPROFILING && INCLUDEDEBUGCODE) {
        checkHeap();
    }
    MonitorCache = NULL;           /*清除所有临时监视器*/
    /*在垃圾收集前，保存当前活动的线程的虚拟机寄存器*/
    if (CurrentThread) {
        storeExecutionEnvironment(CurrentThread);
    }
    /*该函数为 mark-and-sweep 回收算法实现核心部分，后面给出其实现的主要代码*/
    garbageCollectForReal(moreMemory);
    /*完成垃圾收集后，载入执行环境*/
    if (CurrentThread) {
        loadExecutionEnvironment(CurrentThread);
    }
    /*允许异步 I/O 继续*/
    RestartAsynchronousFunctions();
    /*恢复垃圾收集结束标志*/
    gcInProgress = 0;
}
```

#### 2. 垃圾收集函数garbageCollectForReal()

函数 garbageCollectForReal() 是垃圾收集算法 mark-and-sweep 的关键实现部分。函数的标记阶段完成的内容是首先从 root 开始标记对象，然后搜索可达对象，接着标记弱指针列表，将标记为弱引用的对象清除。Sweep 阶段完成释放空间并形成较大的空闲空间。

```
void garbageCollectForReal(int realSize)
{
    CHUNK firstFreeChunk;
```

```

long maximumFreeSize;
/*下面是垃圾收集算法实现部分*/
markRootObjects();           //标志垃圾收集的 root 对象
markNonRootObjects();         //查找堆栈搜索仅从其他堆栈对象可达的对象
markWeakPointerLists();       //标志弱指针列表
markWeakReferences();         //标记弱引用对象，弱引用对象将被清除
/*实现 mark-and-sweep 算法的 sweep 阶段，释放没有活动的方法所占用的堆栈空间*/
firstFreeChunk = sweepTheHeap(&maximumFreeSize);
/*compact 阶段，通过内存的移动构建较大的可用空间*/
#if ENABLE_HEAP_COMPACTION
    if (realSize > maximumFreeSize) {
        /*对堆栈实行紧缩操作，获得可用空间*/
        breakTableStruct currentTable;
        cell* freeStart = compactTheHeap(&currentTable, firstFreeChunk);
        /*得到可用空间后，进行更新操作，更新 root 对象表和堆栈对象表*/
        if (currentTable.length > 0) {
            updateRootObjects(&currentTable);
            updateHeapObjects(&currentTable, freeStart);
        }
        if (freeStart < CurrentHeapEnd - 1) {
            firstFreeChunk = (CHUNK) freeStart;
            firstFreeChunk->size =
                (CurrentHeapEnd - freeStart - HEADERSIZE) << TYPEBITS;
            firstFreeChunk->next = NULL;
        } else {
            /*内存完全满时，没有多余的空间可以通过移动内存来获得*/
            firstFreeChunk = NULL;
        }
    }
#endif
FirstFreeChunk = firstFreeChunk;
}

```

KVM 采用了 mark-and-sweep 回收算法，还存在其他分代回收算法和增量收集算法，下面对其简单介绍。

### 19.3.2 分代回收算法

分代回收算法是根据对象存在时间长短将对象进行分类，每个子堆为一代。随着对象的消亡，垃圾回收器从最年轻的子堆开始回收对象。

分代回收算法在一定程度上降低了垃圾回收给应用带来的负担，使应用的吞吐量达到极限值。但是无法解决垃圾收集所带来的应用暂停。在一些对实时性要求很高的应用场景下，垃圾收集暂停所带来的请求堆积和请求失败是无法接受的。如话音业务，可能要求请求的返回时间在几百甚至几十毫秒内，如果分代垃圾回收算法要达到该指标，只能把最大堆的设置限制在一个相对较小范围内，但是这样又限制了应用本身的处理能力，同样也是不可接受的。

分代垃圾回收算法为考虑实时性要求而提供了并发回收器，支持最大暂停时间的设置，但是受到分代垃圾回收的内存划分模型限制，其效果并不理想。

为了达到实时性的要求（其实 Java 语言最初的设计也是在嵌入式系统上的），一种新垃圾回收方式要求既支持短的暂停时间，又支持大的内存空间分配。这样可以很好地解决

分代方式带来的问题，增量收集可以满足上述要求。

### 19.3.3 增量收集

增量收集的方式在理论上可以解决分代收集方式带来的问题。增量收集把堆空间划分成一系列内存块，使用时，先使用其中一部分，垃圾收集时把之前使用部分中的存活对象再放到后面没有使用的空间中。因此可以实现边使用边收集的效果，避免了分代方式使用完整个内存空间再暂停回收的情况。

分代收集方式中也支持并发收集，分代收集的缺点就是把整个堆作为一个内存块，这样一方面会造成碎片（无法压缩），另一方面它的每次收集都是对整个堆的收集，无法进行选择，在暂停时间的控制上很弱。而增量收集方式，通过内存空间的分块，恰恰可以解决上述问题。

## 19.4 解析器

KVM 的解析器就是将字节码解析成具体平台上的操作指令执行。解析器由循环执行函数 Interpret()，函数 Interpret()中真正执行解析功能的函数为 FastInterpret()，下面对这两个函数进行分析。

### 19.4.1 函数 Interpret()

函数 Interpret() 在函数 KVM\_Start() 中被调用。在该函数的 catch 尾部调用 END\_CATCH\_AND\_GOTO(startTry)，该函数相当于执行 goto 功能跳转到 startTry 处，因此在该函数体内循环执行函数 FastInterpret() 进行解析工作。

```
void Interpret() {
    /*解析器的入口*/
    startTry:
    CurrentNativeMethod = NULL;
    TRY {
        START_TEMPORARY_ROOTS
        IS_TEMPORARY_ROOT(thisObjectGCSafe, NULL);
        FastInterpret();
        END_TEMPORARY_ROOTS
    } CATCH (e) {
        START_TEMPORARY_ROOTS
        IS_TEMPORARY_ROOT(e, e);
        throwException(&e);
        END_TEMPORARY_ROOTS
    }/*END_CATCH_AND_GOTO(startTry) 相当于执行 goto startTry 操作，程序又跳转到 startTry 处，相当于循环函数*/
} END_CATCH_AND_GOTO(startTry)
/*打印解析信息*/
#if INSTRUMENT
```

```

fprintf(stdout,"bytecodes      =\t%ld\n", bytecodes);
fprintf(stdout,"slowcodes   =\t%ld\t(%ld)%%\n", slowcodes,slowcodes/
(bytecodes/100));
fprintf(stdout,"calls     =\t%ld\t(%ld)%%\n", calls, calls/(bytecodes/
100));
fprintf(stdout,"branches taken =\t%ld\t(%ld)%%\n", branches,branches/
(bytecodes/100));
fprintf(stdout,"rescheduled=\t%ld\t(%ld)%%\n", reshed,reshed/
(bytecodes/100));
#endif /* INSTRUMENT */
}

```

## 19.4.2 函数 FastInterpret()

函数 FastInterpret()是执行解析的真正实体函数。下面为函数 FastInterpret()的主要部分，留下函数的主干部分，便于清楚函数的整体实现。函数 FastInterpret()的关键部分为 switch 结构部分，该部分根据寄存器变量 token 的值，执行 callMethod\_general 部分的代码，对 Java 程序中的函数进行解析，包括将函数解析为具体平台的本地函数。

```

void FastInterpret() {
    /*解析器需要定义的局部变量*/
    METHOD thisMethod;
    OBJECT thisObject;
    int invokerSize;
    const char *exception;
#ifndef ENABLE_JAVA_DEBUGGER
    register BYTE token;                                /*寄存器变量*/
#endif
    /* IP 程序计数器*/
    next3: ip++;
    next2: ip++;
    next1: ip++;
    next0:
#ifndef ENABLE_JAVA_DEBUGGER
    token =*/ip;                                     /*用于保存程序计数器 IP 内容*/
    _doSingleStep()
next0a:
#endif
#endif /*RESCHEDULEATBRANCH*/
    /*调用函数 InstructionProfile()*/
    INSTRUCTIONPROFILE
    /*打开指令跟踪*/
    INSTRUCTIONTRACE
    /*增量式字节码计数器*/
    INC_BYTECODES
    /*在每个字节码之前垃圾回收选项*/
    DO VERY EXCESSIVE GARBAGE COLLECTION
    /*指派字节码，通过 switch 分支方式实现解析*/
#ifndef ENABLE_JAVA_DEBUGGER
    switch (token) {
#else
    switch ((unsigned char)*ip) {
#endif
        callMethod_interface:

```

```

    invokerSize = 5;                                /*设置字节码的大小为 5*/
    goto callMethod_general;
callMethod_virtual:
callMethod_static:
callMethod_special:
    invokerSize = 3;                                /*设置字节码的大小为 3*/
/* callMethod_general 部分的代码即对 Java 程序中的函数进行解析，包括将函数
解析为具体平台的本地函数*/
callMethod_general: {
    INC_CALLS
/*检查方法是否为本地方法，实际上 thisMethod 也就是编译 Java 程序中的某个函
数后得到的方法*/
    if (thisMethod->accessFlags & ACC_NATIVE) {
        ip += invokerSize;
        VMSAVE
        invokeNativeFunction(thisMethod);
        VMRESTORE
        TRACE_METHOD_EXIT(thisMethod);
        goto reschedulePoint;
    }
/*检查方法是否为抽象方法*/
    if (thisMethod->accessFlags & ACC_ABSTRACT) {
        VMSAVE
        raiseExceptionWithMessage(AbstractMethodError, methodName
        (thisMethod));
        VMRESTORE
    }
    thisObjectGCSafe = thisObject;
    VMSAVE
    pushFrame(thisMethod);                         //方法入栈
    VMRESTORE
/*执行到下一条指令返回*/
    fp->previousIp += invokerSize;
/*检查该方法是否为同步方法*/
    if (thisMethod->accessFlags & ACC_SYNCHRONIZED) {
        VMSAVE
        monitorEnter(thisObjectGCSafe);
        VMRESTORE
        fp->syncObject = thisObjectGCSafe;
    }
    thisObjectGCSafe = NULL;
    goto reschedulePoint;
}
/*接下来执行各种异常的处理，空指针异常、数组越界异常、算术异常等*/
handleNullPointerException: {
    exception = NullPointerException;
    goto handleException;
}
handleArrayIndexOutOfBoundsException: {
    exception = ArrayIndexOutOfBoundsException;
    goto handleException;
}
handleArithmaticException: {
    exception = ArithmaticException;
    goto handleException;
}
handleArrayStoreException: {
    exception = ArrayStoreException;
    goto handleException;
}

```

```

    }
    handleClassCastException: {
        exception = ClassCastException;
        goto handleException;
    }
    handleException: {
        VMSAVE
        raiseException(exception);
        VMRESTORE
        goto reschedulePoint;
    }
    default: {
        sprintf(str_buffer, KVM_MSG_ILLEGAL_BYTECODE_1LONGPARAM,
            (long)TOKEN);
        fatalError(str_buffer);
        break;
    }
}
fatalError(KVM_MSG_INTERPRETER_STOPPED);
}

```

### 19.4.3 函数 SlowInterpret()

函数 SlowInterpret()在函数 FastInterpret()中被调用，SlowInterpret()作为次级解析器。其完成的功能在 FastInterpret()函数中基本可以完成，SlowInterpret()相当于 FastInterpret()子函数，两者的风格也类似。

```

void SlowInterpret(ByteCode token) {
    METHOD thisMethod;
    OBJECT thisObject;
    int invokerSize;
    switch (token) {
        callMethod_interface: {
            invokerSize = 5; /*设置字节码的大小为 5*/
            goto callMethod_general;
        }
        callMethod_virtual:
        callMethod_static:
        callMethod_special:
            invokerSize = 3; /*设置字节码大小为 3*/
        callMethod_general: {
            /*callMethod_general 部分代码即对 Java 程序中的函数进行解析，包括将函数解析为具体平台的本地函数*/
        }
        /*执行各种异常处理*/
        handleXXXXException() {
        }
    next3: ip++;
    next2: ip++;
    next1: ip++;
    next0:
    reschedulePoint:
        return;
    }
}

```

## 19.5 Java 编程浅析

本节的目标只是让读者了解 Java 编程中的基本规则和概念，包括如何编译源文件、如何命名、类如何定义、主函数如何定义，以及如何编译 Java 程序、如何执行解析等。讲解本节是为分析 19.6 节的 KVM 运行做铺垫，如果读者熟悉 Java 编程可以跳过本节。

### 19.5.1 Java 程序命令

Java 程序源文件采用.class 后缀命名，源文件的名字与类的名字保持一致。在编程风格上，Java 类的首字母采用大写，方法和属性首字母采用小写方式，命令都采用驼峰式方式。下面举例说明其编程方式：

```
public class TestKVM {           //类的定义
    private String theStates;      //属性的定义
    public String getTheStates() {   //方法的定义
        return theStates;
    }
    public void setTheStates(String theStates) {
        this.theStates = theStates;
    }
}
```

在 Java 程序中访问的权限 private、public 都放在每个方法和属性前进行单独指定，与 C++ 有区别。

### 19.5.2 Java 构造函数

下面为构造函数的编写方式，继承类在构造函数中显示调用父类时采用 super()，示例如下：

```
public TestKVM(String theStates) {      //构造函数定义，参数为成员变量
    super();
    this.theStates = theStates;
}
```

构造函数中可以根据需要自定成员变量作为参数，也可以通过参数不同定义多个构造函数。

### 19.5.3 Java 主函数

Java 的编程思想是：一切皆属于类。完全是面向对象的思想，比 C++ 更彻底。所以其主函数 main() 也是放在类中的。下面给出 main() 函数的定义方式：

```
public class TestKVM {
    String theStates;
    public String getTheStates() {
        return theStates;
    }
}
```

```

    }
    public void setTheStates(String theStates) {
        this.theStates = theStates;
    }
    public TestKVM(String theStates) {
        this.theStates = theStates;
    }
    /**
     * 注释的编写方式                                //注释的编写方式
     */
    public static void main(String[] args) {      //主函数的编写方式
        TestKVM t=new TestKVM("kvm");           //对象实例化
        System.out.println(t.getTheStates());     //打印输出
    }
}

```

编写完后命名为 TestKVM.java。

#### 19.5.4 Java 程序编译与运行

编译 Java 程序采用 Java 编译器 javac，运行 Java 程序也就是通过解析器执行解析 Java 程序，将 Java 字节码解析成具体平台的字节码运行，其解析器为 Java。编译和运行的方式如下：

```
# javac TestKVM.java
# java TestKVM
```

执行结果为： kvm

## 19.6 KVM 执行过程

前面从 Java 虚拟机的各个部分分析其主要功能，也了解了 Java 编译和 Java 运行的方法。在前面这些知识的铺垫下，本节将从整体上分析虚拟机的运行过程，通过 KVM 虚拟机展示一般虚拟机执行的整个过程。下面分别介绍 KVM 执行过程中的主要步骤。

#### 19.6.1 KVM 启动过程

在了解 KVM 执行过程前，读者可以先看 KVM 的运行状态图，在接下来的分析过程中主要围绕该状态图进行。该状态图如图 19.2 所示。

KVM 执行过程在函数 KVM\_Start()中体现，函数 KVM\_Start()的代码如下。

```

int KVM_Start(int argc, char* argv[])
{
    ARRAY arguments;
    INSTANCE_CLASS mainClass = NULL;
    volatile int returnValue = 0; /* Needed to make compiler happy */
    /*通过 try...catch 机制捕获异常*/
    TRY {
        VM_START {
            /*创建 ROM 映像，KVM 会先把 class 文件转化为 C 语言源代码，然后编入 KVM 可

```

```

执行程序中，这样当使用系统类库时，KVM 就不再访问外部的 class 文件*/
CreateROMImage();
/*初始化浮点数算术*/
InitializeFloatingPoint();
#if ASYNCHRONOUS_NATIVE_FUNCTIONS
/*初始化异步 I/O 系统*/
InitializeAsynchronousIO();
#endif

/*初始化虚拟机运行时的基本结构*/
InitializeNativeCode();
InitializeVM();
/*初始化全局变量*/
InitializeGlobals();
/*初始化 profiling 变量*/

```

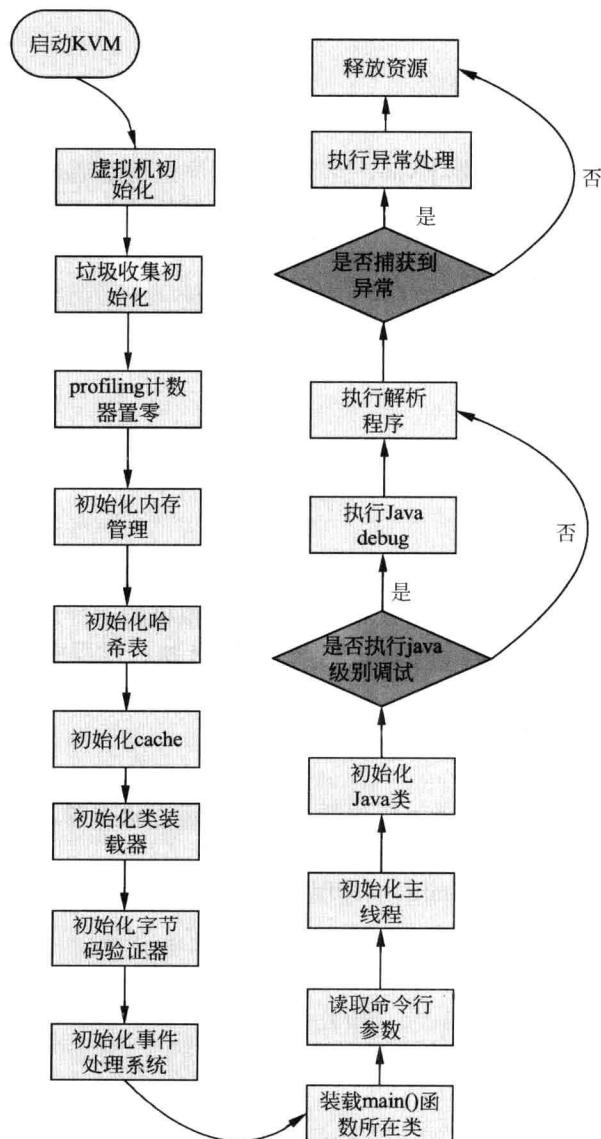


图 19.2 KVM 执行状态图

```

InitializeProfiling();
/*初始化内存管理*/
InitializeMemoryManagement();
/*初始化内部哈希表*/
InitializeHashtables();
/*创建主缓冲区*/
InitializeInlineCaching();
/*创建动态类装载器*/
InitializeClassLoading();
/*导入并初始化虚拟机需要的类*/
InitializeJavaSystemClasses();
/*初始化类文件验证器*/
InitializeVerifier();
/*初始化事件处理系统*/
InitializeEvents();
/*导入主函数所在的类*/
mainClass = loadMainClass(argv[0]);
/*解析命令行参数*/
arguments = readCommandLineArguments(argc - 1, argv + 1);
/*通过命令行参数和主类初始化主线程*/
InitializeThreading(mainClass, arguments);

#if ENABLE_JAVA_DEBUGGER
    /*为 Java 代码级别的调试准备好虚拟机*/
    if(debuggerActive) {
        InitDebugger();
    }
#endif
/*在没有明确执行新指令时创建以下几个 Java 类，并按如下顺序入栈，即栈底部为
类 JavaLangOutOfMemoryError，栈顶为 JavaLangClass*/
initializeClass(JavaLangOutOfMemoryError);
initializeClass(JavaLangSystem);
initializeClass(JavaLangString);
initializeClass(JavaLangThread);
initializeClass(JavaLangClass);

#if ENABLE_JAVA_DEBUGGER
    /*准备 Java 代码级别的调试*/
    if(vmDebugReady) {
        setEvent_VMInit();
        if(CurrentThread == NIL) {
            CurrentThread = removeFirstRunnableThread();
            /*导入执行环境，即通过将线程的执行环境保存在寄存器中*/
            loadExecutionEnvironment(CurrentThread);
        }
        sendAllClassPrepares();
    }
#endif /*ENABLE_JAVA_DEBUGGER*/
    /*开始执行解析*/
    Interpret();
} VM_FINISH(value) {
    returnValue = value;
} VM_END_FINISH
} CATCH(e) {
    /*捕获上面没有捕获的异常*/
    if(mainClass == NULL) {
        /*如果主函数不存在，则打印专门的消息*/
        char buffer[STRINGBUFFERSIZE];
        sprintf(buffer, "%s", getClassName((CLASS)e->ofClass));
    }
}

```

```

        if (e->message != NULL) {
            sprintf(buffer + strlen(buffer), ": %s", getStringContents
                    (e->message));
        }
        sprintf(str_buffer, "%s", buffer);
        AlertUser(str_buffer);
        returnValue = 1;
    } else {
        /*无法捕捉的异常记录在日志中*/
        Log->uncaughtException(e);
        returnValue = UNCAUGHT_EXCEPTION_EXIT_CODE;
    }
} END_CATCH
return returnValue;
}

```

## 19.6.2 KVM 用到的计数器清零

函数 InitializeProfiling()用于将 KVM 用到的计数器清零。该函数中列出了 KVM 使用的计数器，代码如下：

```

void InitializeProfiling()
{
    /*所有计数器清零/
    InstructionCounter      = 0;           /*指令计数器*/
    ThreadSwitchCounter     = 0;           /*线程 Switch 计数器*/
    DynamicObjectCounter    = 0;           /*动态对象计数器*/
    DynamicAllocationCounter = 0;           /*动态内存分配计数器*/
    DynamicDeallocationCounter = 0;         /*动态内存重分配计数器*/
    GarbageCollectionCounter = 0;           /*垃圾收集计数器*/
    TotalGCDeferrals       = 0;           /*延返回收的垃圾收集对象总数*/
    MaximumGCDeferrals     = 0;           /*延返回收的垃圾收集对象最大数目*/
    GarbageCollectionRescans = 0;

#if ENABLEFASTBYTECODES
/*Cache 的使用会大大改善系统性能，其原理为程序的局部性原理（即程序的地址访问流有很强的时序相关性，未来的访问模式与最近已发生的访问模式相似）。根据这一局部性原理，把主存储器中访问概率最高的内容存放在 Cache 中，当 CPU 需要读取数据时就首先在 Cache 中查找是否有所需内容，如果有，则直接从 Cache 中读取；若没有再从主存中读取该数据，然后同时送往 CPU 和 Cache*/
    InlineCacheHitCounter    = 0;           /*访问高速缓冲区的次数*/
    InlineCacheMissCounter   = 0;           /*访问高速缓冲区失败的次数*/
    MaxStackCounter          = 0;           /*需要栈空间的最大数目*/
#endif
#if USESTATIC
    StaticObjectCounter      = 0;           /*静态对象数目*/
    StaticAllocationCounter   = 0;           /*静态空间分配的大小*/
#endif
}

```

### 19.6.3 KVM 初始化内存管理

在 KVM 中执行前期工作主要用于初始化一些相关的资源，初始化过程包括虚拟的硬件和软件的初始化过程。在写 Java 程序时，不需要程序员释放空间，虚拟机会自动帮程序员进行垃圾回收。下面分析虚拟机 KVM 如何初始化内存管理部分。

(1) 函数 InitializeMemoryManagement()用于初始化内存管理。初始化内存时，先初始化堆，然后为第一个对象分配空间。

```
void InitializeMemoryManagement(void)
{
    int index;
    gcInProgress = 0;
    /*初始化堆*/
    InitializeHeap();
    index = 0;
    GlobalRoots[index++].cellpp = (cell **) &AllThreads;
    GlobalRoots[index++].cellpp = (cell **) &CleanupRoots;
    GlobalRootsLength = index;
    TemporaryRootsLength = 0;
    /*为第一个对象分配空间*/
    CleanupRoots =
        (POINTERLIST) callocObject(SIZEOF_POINTERLIST(CLEANUP_ROOT_SIZE),
                                   GCT_POINTERLIST);
}
```

(2) 函数 InitializeHeap()用于初始化堆栈，在该函数中调用函数 allocateHeap()分配堆栈大小 RequestedHeapSize。该堆栈可以根据需要动态分配空间，而不是一个固定的空间。

```
void InitializeHeap(void)
{
    /*堆栈大小*/
    VMHeapSize = RequestedHeapSize;
    /*指向堆栈的底部 cell 指针*/
    AllHeapStart = allocateHeap(&VMHeapSize, &TheHeap);
    /*分配失败*/
    if (TheHeap == NIL) {
        fatalVMEError(KVM_MSG_NOT_ENOUGH_MEMORY);
    }
    /*堆栈通过指向栈底指针、栈顶指针和堆栈大小表示，底部指针偏移堆栈大小的位置就到了堆栈的顶部。该堆栈可以根据需要动态增加大小*/
    CurrentHeap = AllHeapStart;
    CurrentHeapEnd = PTR_OFFSET(AllHeapStart, VMHeapSize);
    /*分配 Java 堆空间时，不采用块*/
    #if !CHUNKY_HEAP
    FirstFreeChunk = (CHUNK) CurrentHeap;
    FirstFreeChunk->size =
        (CurrentHeapEnd - CurrentHeap - HEADERSIZE) << TYPEBITS;
    FirstFreeChunk->next = NULL;
    #endif
    /*如果当前分配的空间为 0，指向当前堆栈顶部指针等于指向所有堆栈的顶部指针*/
    #if ENABLE_HEAP_COMPACTION
    PermanentSpaceFreePtr = CurrentHeapEnd;
```

```
#endif
    AllHeapEnd      = CurrentHeapEnd;
/*将分配堆栈情况写入日志*/
#endif INCLUDEDEBUGCODE
    if (tracememoryallocation) {
        Log->allocateHeap(VMHeapSize, (long)AllHeapStart, (long)
        AllHeapEnd);
    }
    NoAllocation = 0;
#endif
}
```

(3) 再分配堆栈函数 `allocateHeap()`, 参数 `sizeptr` 为分配堆栈空间的大小, 参数 `realresultptr` 返回成功分配的堆栈指针。

```
cell *allocateHeap(long *sizeptr, void **realresultptr) {
    /*分配堆栈大小, 返回分配的空间指针为 space*/
    void *space = malloc(*sizeptr + sizeof(cell) - 1);
    *realresultptr = space;
    return (void *) (((long)space) + (sizeof(cell) - 1)) & ~(sizeof(cell)
    - 1));
}
```

#### 19.6.4 KVM 中的哈希表初始化

KVM 中创建了 3 个哈希表。表 `UTFStringTable` 用于存储所有的 utf C 字符串, 表 `InternStringTable` 存储所有用到 Java 字符串, 表 `ClassTable` 用于存储 Java 类。

```
void InitializeHashTables() {
if (!ROMIZING) {
    /*创建用于存储 utf C 字符串的哈希表*/
    createHashTable(&UTFStringTable, UTF_TABLE_SIZE);
    /*创建用于存储 Java 字符串的哈希表*/
    createHashTable(&InternStringTable, INTERN_TABLE_SIZE);
    /*创建用于存储类的哈希表*/
    createHashTable(&ClassTable, CLASS_TABLE_SIZE);
}
}
```

KVM 对哈希表的定义:

```
typedef struct HashTable {
    long bucketCount;           /*结点个数*/
    long count;                 /*表中所有元素个数*/
    cell *bucket[1];            /*入口指针数组*/
} *HASHTABLE;
/*存储 Java 字符串的哈希表*/
extern HASHTABLE InternStringTable;
/*存储所有的 utf C 字符串的哈希表*/
extern HASHTABLE UTFStringTable;
/*存储 Java 类*/
extern HASHTABLE ClassTable;
```

创建哈希表函数 `createHashTable()`, 参数 `tablePtr` 用于返回创建哈希表的地址, 参数 `bucketCount` 表示哈希表的结点个数, 用于计算哈希表的大小。

```

void createHashTable(HASHTABLE *tablePtr, int bucketCount) {
    /*通过结点个数计算哈希表的大小*/
    int objectSize = SIZEOF_HASHTABLE(bucketCount);
    /*创建哈希表*/
    HASHTABLE table = (HASHTABLE)allocPermanentObject(objectSize);
    /*设置创建的哈希表结点个数*/
    table->bucketCount = bucketCount;
    /*创建的哈希表赋给参数 tablePtr 返回*/
    *tablePtr = table;
}

```

## 19.6.5 KVM 中的事件初始化

函数 InitializeEvents()用于初始化虚拟机的事件系统，方便虚拟机的关闭和重新启动。

```

void InitializeEvents()
{
    waitingThread = 0;           /*等待线程*/
    makeGlobalRoot((cell**)&waitingThread); /*将等待线程存储在全局数组 GlobalRoots 中*/
    eventInP = 0;
    eventCount = 0;
}

```

整个初始化过程完成后，就载入主类，并根据主类和输入命令行参数初始化主线程。接下来创建几个 Java 类并入栈，然后开始执行解析。

## 19.6.6 KVM 中的资源释放

KVM 的资源释放过程与其资源分配过程相对应，执行的顺序与分配的顺序相反。函数 KVM\_Cleanup()用于 KVM 的资源释放。

```

void KVM_Cleanup()
{
    FinalizeVM();           /*结束虚拟机*/
    FinalizeInlineCaching(); /*释放 cache*/
    FinalizeNativeCode();   /*与 InitializeNativeCode() 对应*/
    FinalizeJavaSystemClasses(); /*释放 Java 系统类*/
    FinalizeClassLoader();  /*释放载入类*/
    FinalizeMemoryManagement(); /*结束内存管理*/
    DestroyROMImage();     /*销毁 ROM*/
    FinalizeHashtables();  /*删除哈希表*/
}

```

## 19.7 PC 机安装 JVM

前面对 JVM 的主要功能进行了介绍，同时也分析了其类装载器和解析器的实现。那

么 Java 代码通过在一个平台进行编译就能在安装了 JVM 的其他平台上运行。下面将介绍 JVM 在 Windows 和 Linux 上的安装过程。

### 19.7.1 JVM 在 Windows 上的安装

在 Windows 和 Linux 上安装 JVM 都比较简单，这里给出其安装的步骤。

(1) 下载 JDK 的 Windows 安装文件，安装在指定的目录下。

(2) 设置环境变量。

- 新增的环境变量 JAVA\_HOME，其值为 JDK 的安装路径，如 C:\jdk1.7.0\_25；修改环境变量 path，在 path 值中添加安装 JDK 的 bin 文件路径，如 C:\jdk1.7.0\_25\bin，这样是让系统能够找到 javac 和 java 等工具。
- 新增环境变量 CLASSPATH，其值设置为 D:\java;.;C:\jdk1.7.0\_25\lib\tools.jar，该环境变量指定的路径下的 class 文件能被 JVM 识别并载入。前面分析类装载器时已经提到载入.class 文件，这里就是对类装载器中说的 classpath 环境变量进行设置。

Windows 设置环境变量的方法为，右击“我的电脑”，在弹出的快捷菜单中选择“属性”选项，打开“系统属性”对话框。选择“高级”选项卡，在其中单击“环境变量”按钮，如图 19.3 所示。添加新的环境变量，可以在“环境变量”对话框中单击“新建”按钮进行设置，修改环境变量可以先选好要修改的环境变量，然后再单击“编辑”按钮进行修改。一般进行修改时，新版本的安装路径从前面开始添加，如图 19.4 所示，这样是为了防止受旧版本的影响。

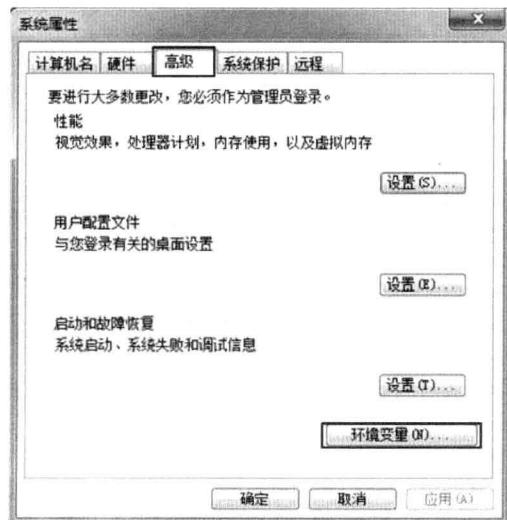


图 19.3 环境变量的设置

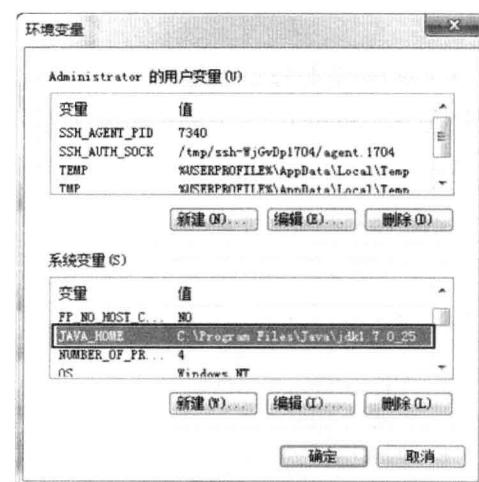


图 19.4 环境变量的增加和修改

(3) 安装完成并正确设置好环境变量后，可以对安装的结果进行简单测试，打开一个控制台，在其中输入 javac 和 java 命令进行测试。安装正确后，测试结果如图 19.5 所示。

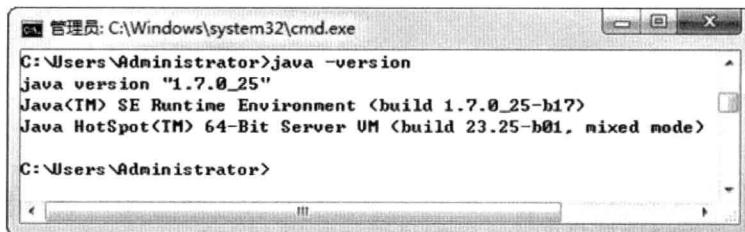


图 19.5 测试 JVM 安装结果

### 19.7.2 JVM 在 Linux 上的安装

JVM 在 Linux 上的安装与在 Windows 上的安装方法类似。其安装过程如下所述。

(1) 下载并安装 Linux 版本的 JDK 文件 jdk-7u25-linux-i586.tar.gz，安装方法如下：

```
#tar zxvf jdk-7u25-linux-i586.tar.gz
#mv jdk1.7.0_25 /usr
```

(2) 设置环境变量，设置的环境变量也包括 JAVA\_HOME、CLASSPATH 和 CLASSPATH，设置的方法如下：

```
# cd $HOME
# vi .bash_profile
//文件.bash_profile 中添加下面的内容
JAVA_HOME=/usr/jdk1.7.0_25
PATH=$JAVA_HOME/bin:$PATH
CLASSPATH=$JAVA_HOME/lib:/usr/JAVA:./
export JAVA_HOME PATH CLASSPATH
```

- /usr/jdk1.7.0\_25 为 JDK 的安装目录。
- PATH：添加指令的搜索路径，再使用 javac 编译一个.java 文件，添加该路径后可以直接在命令行中用 javac 而不要采用绝对路径。
- CLASSPATH：为 Java 代码运行的类路径，/usr/JAVA 为 Java 代码生成的 class 所放置的路径。

(3) 安装和设置完成后，同样可以使用 java -version 和 javac -version 命令进行测试安装的结果，测试结果如图 19.6 所示。



图 19.6 测试 Linux 下 JVM 安装结果

## 19.8 KVM 移植和测试

在对 JVM 进行交叉编译时，为了与 KVM 版本一致，Java SDK 的版本选择为 j2sdk-1\_4\_2。交叉编译器选择低版本的编译器 arm-linux-gcc。

### 19.8.1 SDK 安装和环境变量设置

下载 j2sdk-1\_4\_2\_18-linux-i586.bin 进行安装，j2dk-1\_4\_2 的其他几个版本也试过，都可以成功编译，读者可以选择任意一个进行试验。安装过程如下：

```
# ./ j2sdk-1_4_2_18-linux-i586.bin
# y //同意 license 后进行自动安装
```

安装完成后对环境变量进行修改，修改过程如下：

```
# vi /root/.bash_profile
JAVA_HOME=/usr/j2sdk1.4.2_18
PATH=$JAVA_HOME/bin:$PATH
CLASSPATH=$JAVA_HOME/lib:/usr/JAVA
export JAVA_HOME PATH CLASSPATH
```

修改完成后，需要重新启动计算机或者使用命令 source /root/.bash\_profile，使环境变量生效。

### 19.8.2 修改 Makefile 和代码

对 KVM 需要进行交叉编译，应在对应的 Makefile 中修改 gcc 为 arm-linux-gcc。下载的源码为 j2me\_cldc-1\_1-fcs-src-winunix.zip，在 Linux 下可以直接在 XWindows 模式下通过右击解压，然后复制到/usr 目录中。

#### 1. 修改 kvm 的 Makefile

进入 /usr/j2me\_cldc/kvm/VmUnix/build 目录下修改 Makefile，修改过程如下：

```
# cd /usr/j2me_cldc/kvm/VmUnix/build
# vi Makefile
```

增加平台定义，可以放在所有条件之前，或者放在 Makefile 的开头。

```
export PLATFORM=linux
```

修改 gcc 为 arm-linux-gcc。

```
ifeq ($(GCC), true)
CC = gcc
CFLAGS = -Wall $(CPPFLAGS) $(ROMFLAGS) $(OTHER_FLAGS)
DEBUG_FLAG = -g
OPTIMIZE_FLAG = -O2
```

修改为：

```
ifeq ($(GCC), true)
CC = arm-linux-gcc
CFLAGS = -Wall $(CPPFLAGS) $(ROMFLAGS) $(OTHER_FLAGS)
DEBUG_FLAG = -g
OPTIMIZE_FLAG = -O2
```

## 2. 注释掉浮点数功能

修改目录/usr/j2me\_cldc/kvm/VmUnix/src 下的文件 runtime\_md.c，将浮点数功能注释掉。注释方法如下：

```
/*=====
 * FUNCTION:      InitializeFloatingPoint
 * TYPE:          initialization
 * OVERVIEW:     Called at startup to setup the FPU.
 *
 * INTERFACE:
 *   parameters:  none
 *   returns:     none
 *
 */
void InitializeFloatingPoint() {
#if defined(LINUX) && PROCESSOR_ARCHITECTURE_X86
    /* Set the precision FPU to double precision */
//    fpu_control_t cw = (_FPU_DEFAULT & ~_FPU_EXTENDED) | _FPU_DOUBLE; //可以在此将函数体注释掉
//    _FPU_SETCW(cw);
#endif
}
```

### 19.8.3 KVM 编译

在编译 KVM 前，首先进入 tools/preverifier/build/linux 目录进行编译 preverify。在编译 preverify 时不需要修改该目录下的 Makefile，直接进行编译。

```
#make
```

编译生成 preverify 后回到/usr/j2me\_cldc/build/linux 执行 make 编译 KVM，而并非进入目录/usr/j2me\_cldc/kvm/VmUnix/build 下进行编译。

```
# make
```

编译完成后，在/usr/j2me\_cldc/kvm/VmUnix/build 目录下生成 KVM。

### 19.8.4 测试

在移植前先在 PC 上写个小程序进行测试，确保该程序能在 PC 上顺利通过。对环境变量进行重新修改，主要目的是保证执行命令方便输入，新修改后的环境变量设置如下：

```
JAVA_HOME=/usr/j2sdk1.4.2_18
CLASSPATH=$JAVA_HOME/lib:/usr/JAVA
CLDC_HOME=/usr/j2me_cldc
CLDC_PATH=$CLDC_HOME/bin
PATH=$JAVA_HOME/bin:$CLDC_PATH/linux:$PATH
export JAVA_HOME PATH CLASSPATH CLDC_HOME CLDC_PATH
```

修改完环境变量后，重新启动计算机或者使用命令 `source /root/.bash_profile`，使新设置的环境变量生效。

```
# source /root/.bash_profile
```

在进行测试前，首先写个简单的 Java 测试程序，程序很简单，代码如下：

```
public class TestKVM                                //Java 类定义
{
    public static void main(String[] args)  //主函数
    {
        System.out.println("Hello KVM!"); //打印消息
    }
}
```

该代码文件命名为 `TestKVM`，与类同名，保存在`/usr/JAVA` 目录下。使用 `javac` 命令进行编译，编译生成的.class 文件放在目录 `tmpclasses` 中。

```
# mkdir tmpclasses
# javac -bootclasspath /usr/j2me_cldc/bin/common/api/classes -d tmpclasses
TestKVM.java
```

`javac` 命令的各项参数说明：

- `-bootclasspath` 后面跟 KVM/CLDC java 库的类文件位置。
- `/usr/j2me_cldc/bin/common/api/classes` KVM/CLDC java 库的类文件路径。
- `-d` 后面跟输出文件路径。
- `tmpclasses` 输出编译好的.class 文件路径。
- `TestKVM.java` 待编译的 Java 程序文件。

编译完成后检查生成的.class 文件。

```
# ls tmpclasses
```

在 `tmpclasses` 目录下生成文件 `TestKVM.class`，接下来执行预验证操作。

```
# preverify -classpath /usr/j2me_cldc/bin/common/api/classes:tmpclasses
-d . TestKVM
```

`preverify` 命令的各项参数说明如下：

- `-classpath` 后面跟 KVM/CLDC java 库的类文件位置，与 `-bootclasspath` 相同。
- `/usr/j2me_cldc/bin/common/api/classes` 指 KVM/CLDC java 库的类文件路径。
- `:` 用于分开两个不同的路径。
- `tmpclasses` 存放编译好的.class 文件路径，即待预验证的文件。
- `-d` 后面跟验证后输出文件路径。
- `.` 表示当前路径。
- `TestKVM` 待验证的类名文件，注意不要写成 `TestKVM.class`。

当然上面的写法不唯一，可以作为参考，读者可以根据各项参数说明，用自己习惯的写法。编译完成检查当前目录下生成的文件。

```
# ls
```

在当前目录中生成了验证后的文件 TestKVM.class。执行下面的命令进行测试文件 TestKVM.class。

```
# kvm -classpath . TestKVM
```

执行结果：

```
Hello KVM!
```

执行结果说明文件 TestKVM.class 预验证成功。

### 19.8.5 移植

移植的文件主要有两个，一个是预验证好的文件 TestKVM.class，另一个是交叉编译好的可执行文件 KVM。本次测试采用 NFS 方式进行测试，也回顾一下 NFS。下面提示一下 NFS 的关键步骤，如果有不熟悉的读者，可以重新复习第 2 章的 NFS 挂载部分。

(1) 复制文件到虚拟机的 NFS 相关目录下。

```
# cp /usr/j2me_cldc/kvm/VmUnix/build/kvm /home/nfs/usr/  
                                //复制 KVM 到 USR 目录下  
# cp /usr/JAVA/TestKVM.class /home/nfs/usr/  
                                //复制 TestKVM.class 到 USR 目录下
```

(2) 检查网络是否连接好，连接方式是否为网桥方式。

(3) 检验虚拟机和 ARM 板连接状态。为确保虚拟机和 ARM 板之间为可达，可以使用 ping 命令验证。在设置 IP 时，将两者的 IP 段设置在一个 IP 段，ping 的结果如图 19.7 所示。

```
# ifconfig                                //用于检查两侧的 IP 地址是否在一个网段  
# ping 192.168.1.104                      //在 ARM 板上 ping 主机
```

(4) 检查 NFS 配置。

```
#vi /etc/exports
```

本机的配置如下：

```
#/home/nfs 192.168.1.*(rw,sync,no_root_squash)  
/usr/local/linphone/linphone_arm 192.168.1.*(rw,sync,no_root_squash)
```

将第 1 行注释取消，并注释第 2 行，读者可以针对自己的情况设置，这里主要是提醒读者对于 NFS 的步骤设置。修改配置后应该使用 exportfs -rv 命令使配置重新生效。

```
# exportfs -rv
```

(5) 启动 portmap 服务和 NFS 服务。

```
# service portmap restart  
停止 portmap: [确定]
```

```
启动 portmap: [确定]
# service nfs restart
关闭 NFS mountd: [确定]
关闭 NFS 守护进程: [确定]
关闭 NFS quotas: [确定]
关闭 NFS 服务: [确定]
启动 NFS 服务: [确定]
关掉 NFS 配额: [确定]
启动 NFS 守护进程: [确定]
启动 NFS mountd: [确定]
```

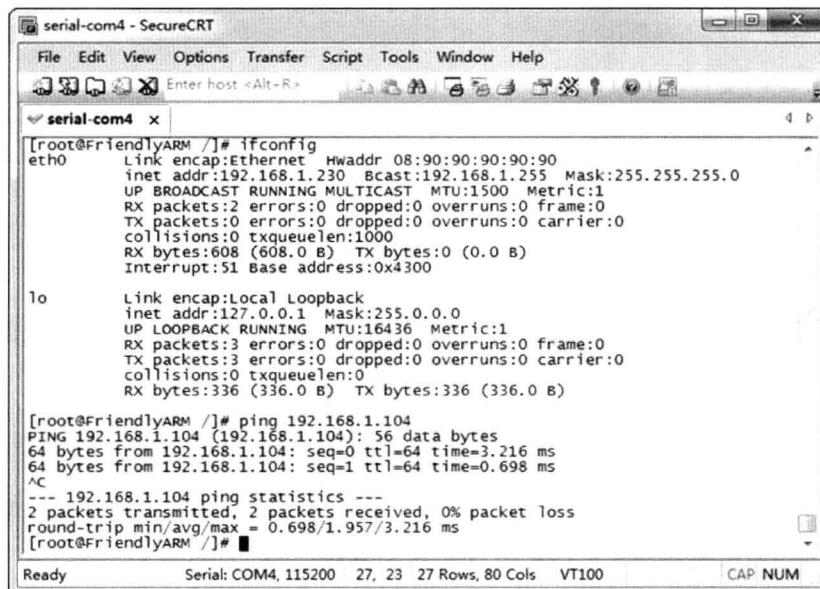


图 19.7 虚拟机和 ARM 可以 ping 通

上面的启动过程还要注意启动顺序，后面是正确启动的信息，如果不能正确启动，请参考第 2 章的说明。

- (6) 关闭防火墙。关闭的防火墙包括 Windows 主机防火墙和 Linux 虚拟机的防火墙。
- (7) 挂载 NFS。

```
# mount -o noblock -t nfs 192.168.1.104:/home/nfs /mnt
```

(8) 移植动态链接库。移植动态链接库前，保证编译内核和文件系统的编译器与编译 KVM 编译器为相同的版本。在目录/usr/j2me\_cldc/bin/linux 下编译的 KVM 是运行在 X86 平台上的，查看其依赖的库文件。将库文件移植到/lib 目录下。

```
# ldd kvm
linux-gate.so.1 => (0x00397000)
libm.so.6 => /lib/libm.so.6 (0x446c7000)
libnsl.so.1 => /lib/libnsl.so.1 (0x43c37000)
libc.so.6 => /lib/libc.so.6 (0x44588000)
/lib/ld-linux.so.2 (0x43bb9000)
```

将交叉编译器下的这些动态链接库移植到开发板的/lib 目录下。

- (9) 运行测试。执行测试的时候最好将 KVM 与 TestKVM 放置在同一个目录下，避免

因路径引起的装载时名字错误而导致异常现象。

```
# kvm TestKVM
```

- 正确的显示结果为 Hello KVM!
- 如果出现找不到 KVM，说明没有正确的库文件；
- 如果出现权限不够，则需要修改 KVM 权限。

移植时为了省去移植动态链接库的麻烦，还可以使用静态编译的方式。静态编译时，需修改目录/usr/j2me\_cldc/kvm/VmUnix/build 下的 Makefile 文件。

```
kvm$(j)$g : obj$j$g/ fp_obj$j$g/ $(CLEANUPXPM) $(OBJFILES) $(FP_OBJFILES)
@echo "Linking ... $@"
@$(CC) $(OBJFILES) $(FP_OBJFILES) -o $@ $(LIBS)
```

添加编译参数-static。

```
kvm$(j)$g : obj$j$g/ fp_obj$j$g/ $(CLEANUPXPM) $(OBJFILES) $(FP_OBJFILES)
@echo "Linking ... $@"
@$(CC) -static $(OBJFILES) $(FP_OBJFILES) -o $@ $(LIBS)
```

修改完成后，回到/usr/j2me\_cldc/build/linux 目录下进行编译，静态编译的可执行文件比较大，编译完后可以通过 file 命令进行查看：

```
# file kvm
kvm: ELF 32-bit LSB executable, ARM, version 1 (SYSV), statically linked,
for GNU/Linux 2.6.32, not stripped
```

## 19.9 小结

Java 语言有很多优点，包括完全面向对象、平台可移植性、不需要程序员进行内存回收等，相对于 C 和 C++来说更安全。而且 Java 有专门针对嵌入式平台的分支 J2ME，目前移动终端、PDA 上越来越流行 Java 程序。移植完成 Java 虚拟机后，就可以将一些在 PC 上的 Java 应用程序移植到嵌入式环境中，移植这些 Java 应用程序比 C 和 C++应用程序简单得多。

# 第 20 章 VoIP 技术与 Linphone 编译

Linphone 是一个运行于 Linux 下的小型万维网电话应用程序 VoIP (Voice over Internet Protocol)。它允许用户通过因特网来进行双方通话和视频。不需要特定的硬件项目，只要安装了声卡的标准工作站、麦克风和扬声器或耳机，即可开始使用 Linphone 进行语音、文本和视频通信。本章重点介绍 VoIP 的基础知识和 Linphone 涉及的协议，同时还介绍 Linphone 编译和移植方法。

## 20.1 VoIP 介绍

VoIP 是利用 IP 网络传送语音、传真、视频和数据等业务，为用户节省通信费用的一种通信设备。它主要适合有分支机构的企业和集团用户，能给企业节省大量的国际、国内长途通信费用。下面将介绍其基本原理、优点、过程和实现方式。

### 20.1.1 VoIP 基本原理

VoIP 就是 IP 分组上承载话音，其基本原理是：通过语音压缩算法对语音数据进行压缩编码处理，然后把这些语音数据按 IP 等相关协议进行打包，经过 IP 网络把数据包传输到接收地，再把这些语音数据包串起来，经过解码解压处理后，恢复成原来的语音信号，从而达到由 IP 网络传送语音的目的。现有的简单 VoIP 系统结构如图 20.1 所示。

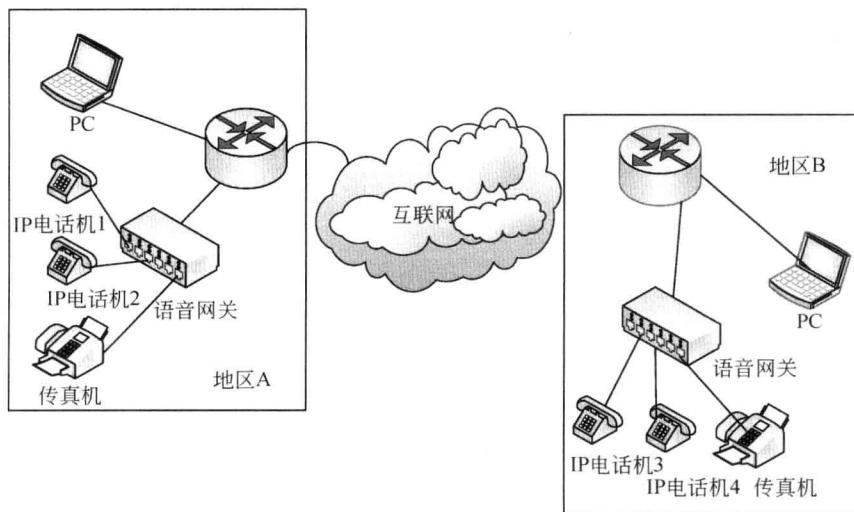


图 20.1 VoIP 的系统结构

### 20.1.2 VoIP 的基本传输过程

VoIP 实现通信双方或多方通信经过了下面一系列的过程。

- (1) 模拟语音转换数据。
- (2) 原数据转换为 IP 包。
- (3) 传送 IP 包。
- (4) IP 包转换为数据。
- (5) 数字语音转换为模拟语音。

VoIP 的语音传输过程, 如图 20.2 所示。

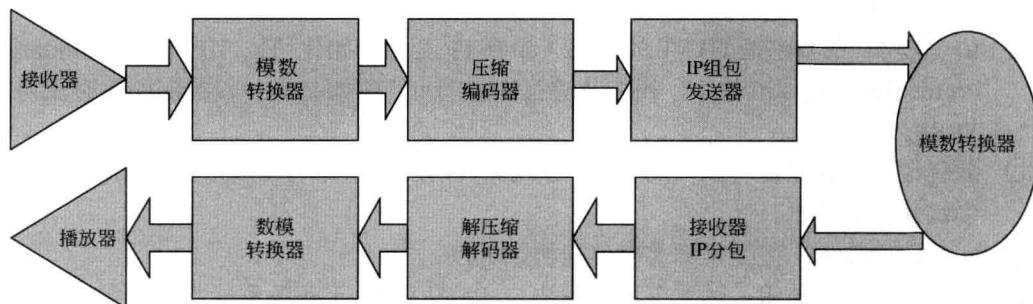


图 20.2 VoIP 语音传输图

### 20.1.3 VoIP 的优势

VoIP 相比传统电话具有明显的优势, 其主要优势体现如下:

- 低廉的通信资费;
- 地理无关和号码漫游;
- 将语音网络和数据网络有机结合;
- 扩展了传统电话的功能, 如视频通话、多方通话、视频会议、统一消息、数据存储转发、传真、流媒体等;
- 更多的应用和服务, 如交互式电子商务、企业传真、多媒体视讯、智能代理等;
- 低廉的网络租赁维护费用。

### 20.1.4 VoIP 的实现方式

VoIP 主要有 4 种实现方式: 电话机到电话机、电话机到 PC、PC 到电话机和 PC 到 PC。最初 VoIP 实现方式主要是 PC 到 PC。它利用 IP 地址进行呼叫, 通过语音压缩、组包传送方式, 实现互联网上 PC 机间的实时话音传送。在 PC 到 PC 的实现方式中, 话音压缩、编码和组包均通过 PC 上的处理器、声卡、网卡等硬件资源完成, 这种方式和公用电话通信之间存在较大的差别, 且限定在因特网内, 所以有较大的局限性。

电话机到电话机, 即普通电话经过电话交换机连到 IP 电话网关, 使用电话号码穿越 IP

网进行呼叫，主叫端网关鉴别主叫用户，翻译电话号码 / 网关 IP 地址，发起 IP 电话呼叫，连接到被叫端网关，并完成话音编码和组包，接收端网关实现分包、解码和连接被叫。

对于电话机到 PC 或是 PC 到电话的情况，是由网关来完成 IP 地址和电话号码的对应和翻译，以及话音编解码和组包。

### 20.1.5 VoIP 的关键技术

IP 网络目的是用来传输数据业务，采用的是尽力而为的、无连接的技术。因此，它没有服务质量保证，存在分组丢失、失序到达和时延抖动等情况。而话音业务属于实时业务，对时序、时延等有严格的要求，必须采取其他服务质量保证业务质量。VoIP 的关键技术包括信令技术、编码技术、实时传输技术、服务质量保证技术及网络传输技术等。

- 信令技术：主要包括 ITU-T 的 H.323 和 IETF 会话初始化协议 SIP (Session Initiation Protocol) 两套标准体系，还涉及进行实时同步连续媒体流传输控制的实时流协议 TRSP。
- 编码技术：包括流行的 G.723.1、G.729、G.729A 话音压缩编码算法和 MPEG-II 多媒体压缩技术。
- 实时传输技术：主要采用实时传输协议 RTP。
- 服务质量保证技术：运用资源预留协议 RSVP 和用于业务质量监控的实时传输控制协议 RTCP 来解决网络拥塞，保证通话质量。
- 网络传输技术：主要是面向连接的 TCP 协议和无连接的 UDP 协议。

后面将结合代码介绍部分关键技术和协议，讲解它们的实现过程和原理。

## 20.2 oSIP 协议概述

SIP 协议会话控制协议，用来建立、修改和终止多媒體会话。oSIP 协议是用标准 C 编写的一个 SIP 协议栈，在编译 Linphone 时采用了支持 oSIP 协议的源码包 libosip2-4.0.0.tar.gz 和 libeXosip2-4.0.0.tar.gz，编译后分别得到 osip 的库和 eXosip 库文件，它们是 oSIP 的协议库和 oSIP 协议扩展库文件。

oSIP 协议栈主要分为 3 大部分：状态机模块、解析器模块和工具模块。3 个模块的功能如下所述。

- 状态机模块功能：记录某个事务（注册过程、呼叫过程）的状态，并在特定的状态下触发某个相应的事件或回调函数。
- 解析器模块功能：该模块主要完成对 SIP 消息的解析、SDP 消息的解析、URL 消息的解析。
- 工具模块功能：提供 SDP 等处理工具。

oSIP 协议栈的状态机又分为 4 种类型，分别如下所述。

- ICT: Invite Client (outgoing) Transaction;
- NICT: Non-Invite Client (outgoing) Transaction;
- IST: Invite Server (incoming) Transaction;

- NIST: Non-Invite Server (incoming) Transaction。

下面通过对 oSIP 几个重要部分的了解来进一步理解 oSIP，并通过对 oSIP 状态机、oSIP 解析器、oSIP 事务层等源码分析理解 oSIP 的实现细节。

## 20.3 oSIP 状态机

前面介绍的 4 种状态机 ICT、NICT、IST 和 NIST 在 osip.h 文件中的 state\_t 结构体中定义，每种状态机都包含 5 种状态：准备呼叫（PRE\_CALLING）、呼叫（CALLING）、处理（PROCEEDING）、完成（COMPLETED）和终止（TERMINATED）。该结构体具体定义如下：

```
typedef enum _state_t
{
    /*ICT 状态机的各个状态*/
    ICT_PRE_CALLING,
    ICT_CALLING,
    ICT_PROCEEDING,
    ICT_COMPLETED,
    ICT_TERMINATED,

    /*IST 状态机的各个状态*/
    IST_PRE_PROCEEDING,
    IST_PROCEEDING,
    IST_COMPLETED,
    IST_CONFIRMED,
    IST_TERMINATED,

    /*NICT 状态机的各个状态*/
    NICT_PRE_TRYING,
    NICT_TRYING,
    NICT_PROCEEDING,
    NICT_COMPLETED,
    NICT_TERMINATED,

    /*NIST 状态机的各个状态*/
    NIST_PRE_TRYING,
    NIST_TRYING,
    NIST_PROCEEDING,
    NIST_COMPLETED,
    NIST_TERMINATED,
}
state_t;
```

状态机一般为一个程序的核心部分，理解了状态机就能清楚地了解整个程序的流程。下面分别对这 4 种状态机进行图示分析与代码分析，读者先获得最新的 libosip2 压缩包，解压后熟悉 src 下面的文件。

### 20.3.1 ICT (Invite Client (outgoing) Transaction) 状态机

在 libosip 源码目录下的 ICT 状态机文件为 ict\_fsm.c，数组 ict\_transition[11] 给定了各个

状态接收事件的各种事务处理情况。

```

transition_t ict_transition[11] =
{
{
    ICT_PRE_CALLING,           //初始状态为 ICT_PRE_CALLING
    SND_REQINVITE,             //当接收到 SND_REQINVITE 事件时
    (void (*)(void *, void *)) &ict_snd_invite,
                                //调用处理事件函数 ict_snd_invite()
    &ict_transition[1], NULL
}

{
    ICT_CALLING,               //初始状态为 ICT_CALLING
    TIMEOUT_A,                 //当接收到 TIMEOUT_A 事件时
    (void (*)(void *, void *)) &osip_ict_timeout_a_event,
                                //调用处理函数 osip_ict_timeout_a_event()
    &ict_transition[2], NULL
}

{
    ICT_CALLING,               //初始状态为 ICT_CALLING
    TIMEOUT_B,                 //当接收到 TIMEOUT_B 事件时
    (void (*)(void *, void *)) &osip_ict_timeout_b_event,
                                //调用处理函数 osip_ict_timeout_b_event()
    &ict_transition[3], NULL
}

{
    ICT_CALLING,               //初始状态为 ICT_CALLING
    RCV_STATUS_1XX,             //当接收到 RCV_STATUS_1XX 事件时
    (void (*)(void *, void *)) &ict_rcv_1xx,
                                //调用处理函数 ict_rcv_1xx ()
    &ict_transition[4], NULL
}

{
    ICT_CALLING,               //初始状态为 ICT_CALLING
    RCV_STATUS_2XX,             //当接收到 RCV_STATUS_2XX 事件时
    (void (*)(void *, void *)) &ict_rcv_2xx,
                                //调用处理函数 ict_rcv_2xx ()
    &ict_transition[5], NULL
}

{
    ICT_CALLING,               //初始状态为 ICT_CALLING
    RCV_STATUS_3456XX,          //当接收到 RCV_STATUS_3456XX 事件时
    (void (*)(void *, void *)) &ict_rcv_3456xx,
                                //调用处理函数 ict_rcv_3456xx ()
    &ict_transition[6], NULL
}

{
    ICT_PROCEEDING,            //初始状态为 ICT_PROCEEDING
    RCV_STATUS_1XX,             //当接收到 RCV_STATUS_1XX 事件时
    (void (*)(void *, void *)) &ict_rcv_1xx,
                                //调用处理函数 ict_rcv_1xx ()
    &ict_transition[7], NULL
}

{
    ICT_PROCEEDING,            //初始状态为 ICT_PROCEEDING
}

```

```
RCV_STATUS_2XX, //当接收到 RCV_STATUS_2XX 事件时
(void *) (void *, void *)) &ict_rcv_2xx,
//调用处理函数 ict_rcv_2xx ()

&ict_transition[8], NULL
}

{
ICT_PROCEEDING, //初始状态为 ICT_PROCEEDING
RCV_STATUS_3456XX, //当接收到 RCV_STATUS_3456XX 事件时
(void *) (void *, void *)) &ict_rcv_3456xx,
//调用处理函数 ict_rcv_3456xx ()

&ict_transition[9], NULL
}

{
ICT_COMPLETED, //初始状态为 ICT_COMPLETED
RCV_STATUS_3456XX, //当接收到 RCV_STATUS_3456XX 事件时
(void *) (void *, void *)) &ict_retransmit_ack,
//调用处理函数 ict_retransmit_ack ()

&ict_transition[10], NULL
}

{
ICT_COMPLETED, //初始状态为 ICT_COMPLETED
TIMEOUT_D, //当接收到 TIMEOUT_D 事件时
(void *) (void *, void *)) &osip_ict_timeout_d_event,
//调用处理函数 osip_ict_timeout_d_event()

NULL, NULL
}
```

### 1. 错误事务处理函数 `ict handle transport error()`

任何状态接收到错误事件时，都会产生一个回调函数`_osip_transport_error_callback()`，再进入`ICT_TERMINATED`状态，然后调用回调函数`_osip_kill_transaction_callback()`退出状态机。

```

static void ict_handle_transport_error (osip_transaction_t * ict, int err)
{
    /*调用回调函数__osip_transport_error_callback()*/
    __osip_transport_error_callback (OSIP_ICT_TRANSPORT_ERROR, ict, err);
    /*进入状态机的终结状态*/
    __osip_transaction_set_state (ict, ICT_TERMINATED);
    /*调用回调函数__osip_kill_transaction_callback()退出状态机*/
    __osip_kill_transaction_callback (OSIP_ICT_KILL_TRANSACTION, ict);
}

```

由上面错误处理函数 `ict_handle_transport_error()` 所表示的状态迁移关系，可以得到状态图 20.3 中粗实线条部分。

## 2. 发送邀请函数ict\_snd\_invite()

发送邀请函数 `ict_snd_invite()`, 是预呼叫状态 (ICT\_PRE\_CALLING) 下的处理函数, 当发送消息成功时, 调用回调函数 `_osip_message_callback()` 并进入呼叫状态 (ICT\_CALLING)。

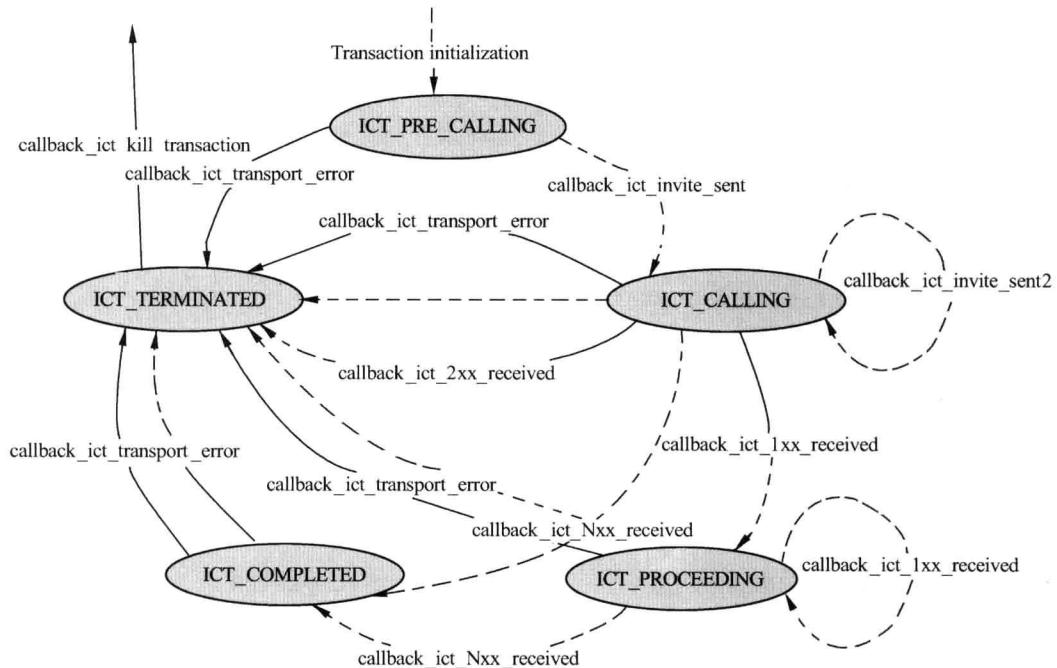


图 20.3 ICT 状态机的错误事务处理

```

void ict_snd_invite (osip_transaction_t * ict, osip_event_t * evt)
{
    int i;
    osip_t *osip = (osip_t *) ict->config;

    ict->orig_request = evt->sip;
    /*发送消息成功则返回 0, 否则返回非 0*/
    i = osip->cb_send_message (ict, evt->sip, ict->ict_context->destination,
                                ict->ict_context->port, ict->out_socket);

    if (i < 0) {
        /*邀请失败则进行错误处理*/
        ict_handle_transport_error (ict, i);
        return;
    }

#ifndef USE_BLOCKINGSOCKET
    if (i == 0) {
        osip_via_t *via;
        char *proto;

        i = osip_message_get_via (ict->orig_request, 0, &via);
        if (i < 0) {
            ict_handle_transport_error (ict, i);
            return;
        }
        proto = via_get_protocol (via);
        if (proto == NULL) {
            ict_handle_transport_error (ict, i);
            return;
        }
        if (osip_strcasecmp (proto, "TCP") != 0 && osip_strcasecmp (proto,
"TLS") != 0 && osip_strcasecmp (proto, "SCTP") != 0) {
    
```

```

    }
    else {
        ict->ict_context->timer_a_length = -1;
        ict->ict_context->timer_a_start.tv_sec = -1;
    }
}
#endif

/*成功则调用回调函数*/
__osip_message_callback (OSIP_ICT_INVITE_SENT, ict, ict->orig_
request);
/*进入呼叫状态*/
__osip_transaction_set_state (ict, ICT_CALLING);
}

```

上面的发送邀请函数 `ict_snd_invite()` 所表示的状态迁移关系，在图 20.4 中用粗实线体现出来。

### 3. 应答函数 `ict_rcv_1xx()`

根据数组 `ict_transition[11]` 的定义，调用函数 `ict_rcv_1xx()` 的初始状态有 `ICT_PROCEEDING` 和 `ICT_CALLING` 两种情况，而根据函数 `ict_rcv_1xx()` 处理的情况看，结束状态均为 `ICT_PROCEEDING`。

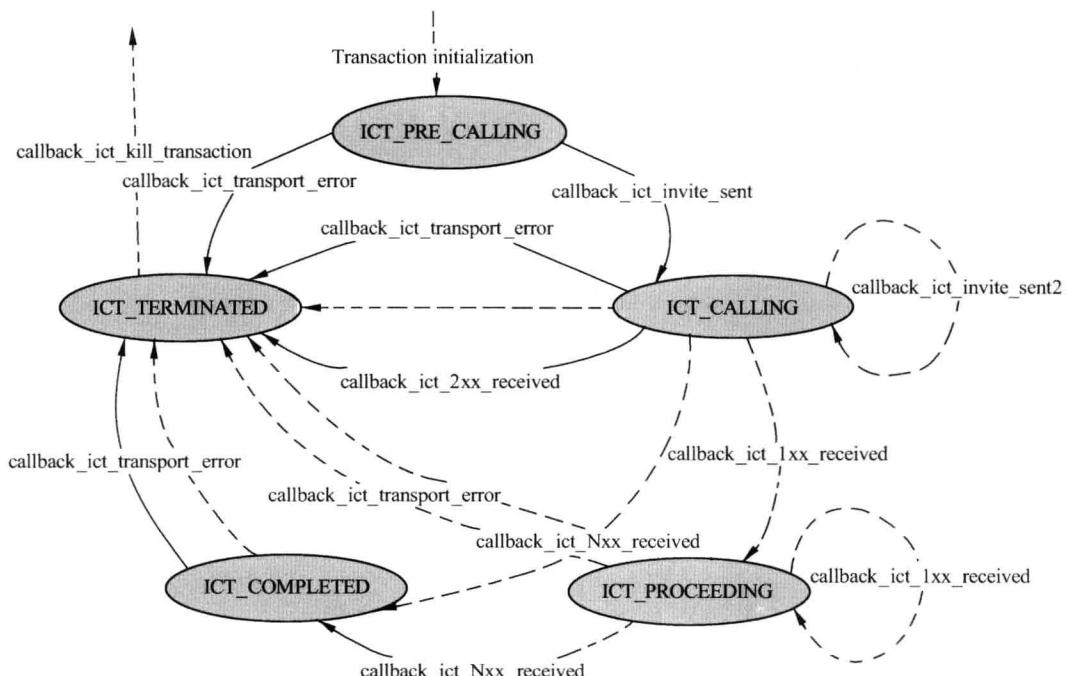


图 20.4 PRE\_CALLING 发送邀请事件

```

void ict_rcv_1xx (osip_transaction_t * ict, osip_event_t * evt)
{
    if (ict->last_response != NULL)
    {

```

```

        osip_message_free (ict->last_response);
    }
    ict->last_response = evt->sip;
/*调用回调函数*/
    __osip_message_callback (OSIP_ICT_STATUS_1XX_RECEIVED, ict, evt->sip);
/*函数的出口状态均为 ICT_PROCEEDING*/
    __osip_transaction_set_state (ict, ICT_PROCEEDING);
}

```

应答函数 `ict_rcv_1xx()` 和应答函数 `ict_rcv_2xx()` 一起在状态转移图中体现的结果如图 20.5 所示。

#### 4. 应答函数 `ict_rcv_2xx()`

根据数组 `ict_transition[11]` 的定义，调用函数 `ict_rcv_2xx()` 的初始状态有 `ICT_PROCEEDING` 和 `ICT_CALLING` 两种状态，而根据函数 `ict_rcv_2xx()` 处理的情况看结束状态均为 `ICT_TERMINATED`，如图 20.5 所示。

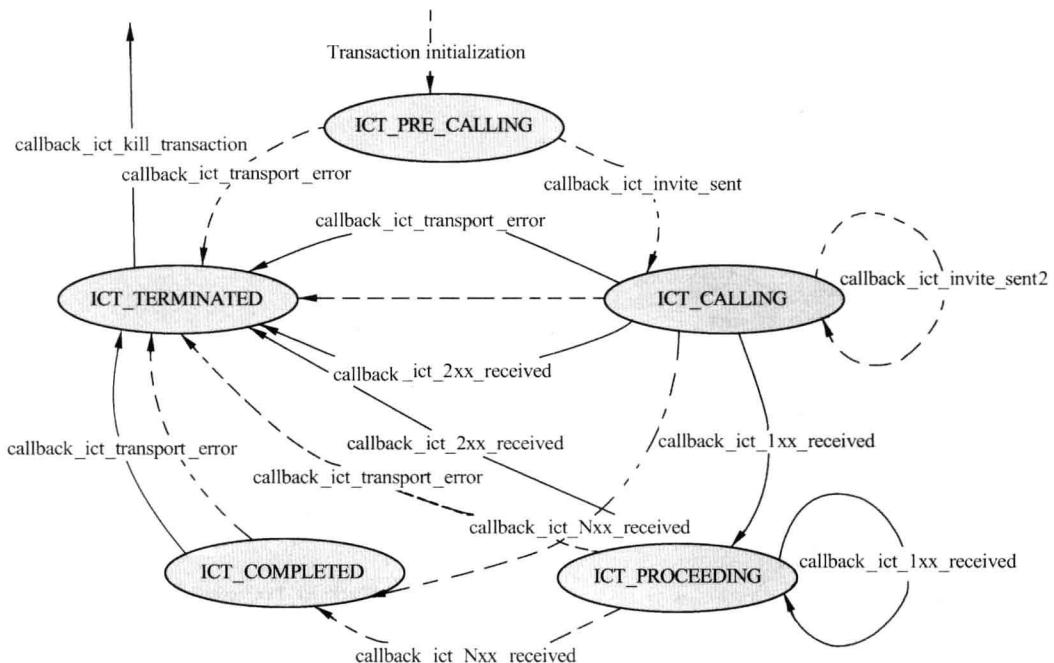


图 20.5 `ict_rcv_1xx()`与 `ict_rcv_2xx()`处理

```

void ict_rcv_2xx (osip_transaction_t * ict, osip_event_t * evt)
{
    if (ict->last_response != NULL)
    {
        osip_message_free (ict->last_response);
    }
    ict->last_response = evt->sip;
/*调用回调函数*/
    __osip_message_callback (OSIP_ICT_STATUS_2XX_RECEIVED, ict, evt->sip);
}

```

```

/*进入 ICT_TERMINATED 状态*/
    __osip_transaction_set_state (ict, ICT_TERMINATED);
/*调用回调函数*/
    __osip_kill_transaction_callback (OSIP_ICT_KILL_TRANSACTION, ict);
}

```

## 5. 应答函数ict\_rcv\_3456xx()

根据数组 ict\_transition[11]的定义，调用应答函数 ict\_rcv\_3456xx() 的初始状态有 ICT\_PROCEEDING、ICT\_COMPLETED 和 ICT\_CALLING 这 3 种状态。函数 ict\_rcv\_3456xx() 处理的结果最终状态为 ICT\_COMPLETED。

```

void ict_rcv_3456xx (osip_transaction_t * ict, osip_event_t * evt)
{
    osip_route_t *route;
    int i;
    osip_t *osip = (osip_t *) ict->config;

    if (ict->last_response != NULL)
        osip_message_free (ict->last_response);
    ict->last_response = evt->sip;
    if (ict->state != ICT_COMPLETED) /* not a retransmission */
    {
        /* automatic handling of ack! */
        osip_message_t *ack = ict_create_ack (ict, evt->sip);
        ict->ack = ack;
        if (ict->ack == NULL)
        {
            /*初始状态不是 ICT_COMPLETED，根据数组 ict_transition[11] 的定义是状态
               ICT_CALLING 和 ICT_PROCEEDING 时，直接跳转到 ICT_TERMINATED 状态*/
            __osip_transaction_set_state (ict, ICT_TERMINATED);
            /*调用回调函数*/
            __osip_kill_transaction_callback (OSIP_ICT_KILL_TRANSACTION,
                                              ict);
            return;
        }
        /*省略一些细节，为了更容易看出状态机跳转的主线*/
        i = osip->cb_send_message (ict, ack, ict->ict_context->destination,
                                    ict->ict_context->port, ict->out_socket);
        if (i != 0)
        {
            /*状态 ICT_CALLING 和 ICT_PROCEEDING 发送消息错误返回时，调用错误处理函数
               ict_handle_transport_error ()*/
            ict_handle_transport_error (ict, i);
            return;
        }
        if (MSG_IS_STATUS_3XX (evt->sip))
            /*调用回调函数*/
            __osip_message_callback (OSIP_ICT_STATUS_3XX_RECEIVED, ict, evt->sip);
        else if (MSG_IS_STATUS_4XX (evt->sip))
            /*调用回调函数*/
            __osip_message_callback (OSIP_ICT_STATUS_4XX_RECEIVED, ict, evt->sip);
    }
}

```

```

else if (MSG_IS_STATUS_5XX (evt->sip))
    /*调用回调函数*/
    __osip_message_callback (OSIP_ICT_STATUS_5XX_RECEIVED, ict, evt->
    sip);
Else
    /*调用回调函数*/
    __osip_message_callback (OSIP_ICT_STATUS_6XX_RECEIVED, ict, evt->
    sip);
    /*调用回调函数*/
    __osip_message_callback (OSIP_ICT_ACK_SENT, ict, evt->sip);
}
/* start timer D (length is set to MAX (64*DEFAULT_T1 or 32000) */
osip_gettimeofday (&ict->ict_context->timer_d_start, NULL);
add_gettimeofday (&ict->ict_context->timer_d_start, ict->ict_context->
timer_d_length);
/*处理完该函数，正常退出时结束状态为 ICT_COMPLETED*/
__osip_transaction_set_state (ict, ICT_COMPLETED);
}

```

根据上面的函数分析，得到下面粗实线所表示的状态转移图，如图 20.6 所示。

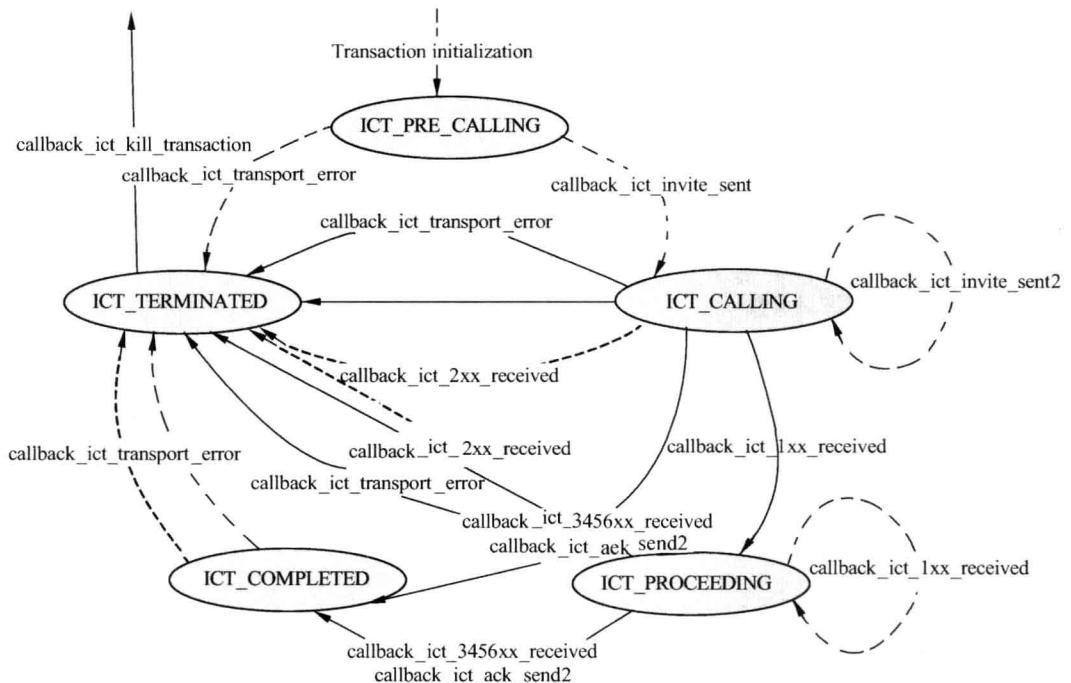


图 20.6 函数 `ict_rcv_3456xx()` 表示的状态转移部分

## 6. 超时事件

超时事件函数 `osip_ict_timeout_b_event()` 和 `osip_ict_timeout_d_event()` 涉及状态之间的跳转，接收事件 `TIMEOUT_B` 时，状态从 `ICT_CALLING` 跳转到 `ICT_TERMINATED`。接收超时事件 `TIMEOUT_D` 时，状态从 `ICT_COMPLETED` 跳转到 `ICT_TERMINATED`。两

个事件处理函数如下，其状态转移图如图 20.7 中粗实线部分所示。

```
void osip_ict_timeout_b_event (osip_transaction_t * ict, osip_event_t * evt)
{
    ict->ict_context->timer_b_length = -1;
    ict->ict_context->timer_b_start.tv_sec = -1;
/*调用超时回调函数*/
    __osip_message_callback (OSIP_ICT_STATUS_TIMEOUT, ict, evt->sip);
/*设置状态为 ICT_TERMINATED*/
    __osip_transaction_set_state (ict, ICT_TERMINATED);
/*调用结束事务处理回调函数*/
    __osip_kill_transaction_callback (OSIP_ICT_KILL_TRANSACTION, ict);
}

void osip_ict_timeout_d_event (osip_transaction_t * ict, osip_event_t * evt)
{
    ict->ict_context->timer_d_length = -1;
    ict->ict_context->timer_d_start.tv_sec = -1;
/*设置状态为 ICT_TERMINATED*/
    __osip_transaction_set_state (ict, ICT_TERMINATED);
/*调用结束事务处理回调函数*/
    __osip_kill_transaction_callback (OSIP_ICT_KILL_TRANSACTION, ict);
}
```

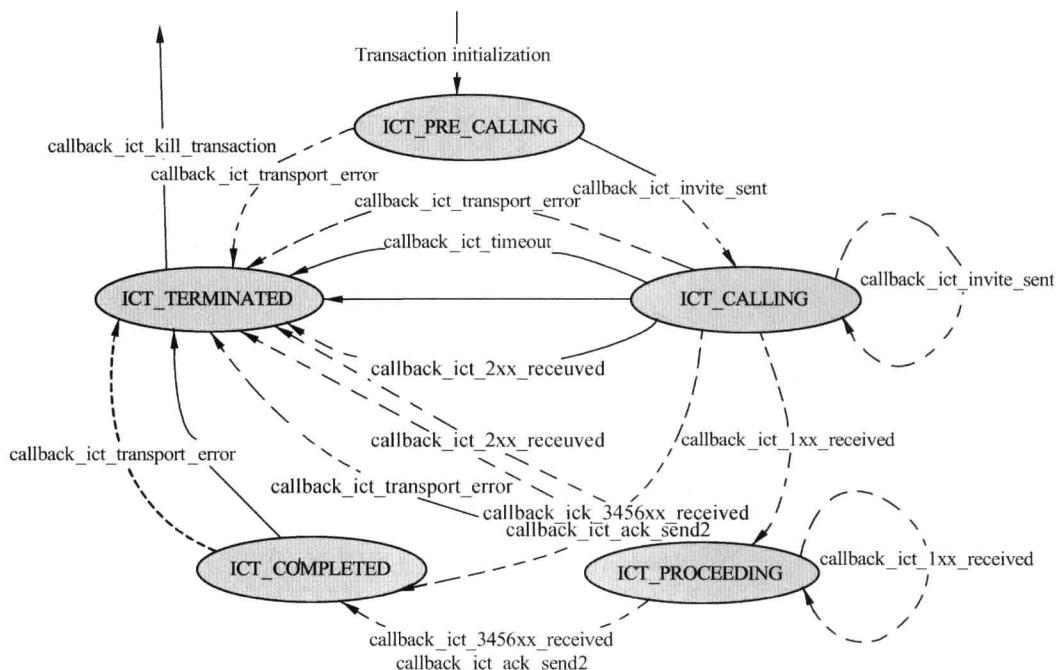


图 20.7 超时事件 TIMEOUT\_B 和 TIMEOUT\_D 所产生的状态转移

## 7. 应答函数 ict\_retransmit\_ack()

根据接收函数 ict\_recv\_3456xx() 中调用的 ACK\_SENT 事件可知，ict\_retransmit\_ack() 的

起始状态为 ICT\_COMPLETED，正常情况下的结束状态为 ICT\_COMPLETED，出错时调用错误处理函数。由该函数得出的状态转移图如图 20.8 中粗实线所示。

```
void ict_retransmit_ack (osip_transaction_t * ict, osip_event_t * evt)
{
    int i;
    osip_t *osip = (osip_t *) ict->config;

    /*调用回调函数*/
    __osip_message_callback (OSIP_ICT_STATUS_3456XX RECEIVED AGAIN, ict,
    evt->sip);
    /*释放消息*/
    osip_message_free (evt->sip);

    i = osip->cb_send_message (ict, ict->ack, ict->ict_context->destination,
                                ict->ict_context->port, ict->out_socket);

    if (i == 0)
    {
        /*调用回调函数再次应答发送*/
        __osip_message_callback (OSIP_ICT_ACK_SENT AGAIN, ict, ict->ack);
        /*设置状态 ICT_COMPLETED*/
        __osip_transaction_set_state (ict, ICT_COMPLETED);
    } else
    {
        /*发送消息错误则进行错误处理*/
        ict_handle_transport_error (ict, i);
    }
}
```

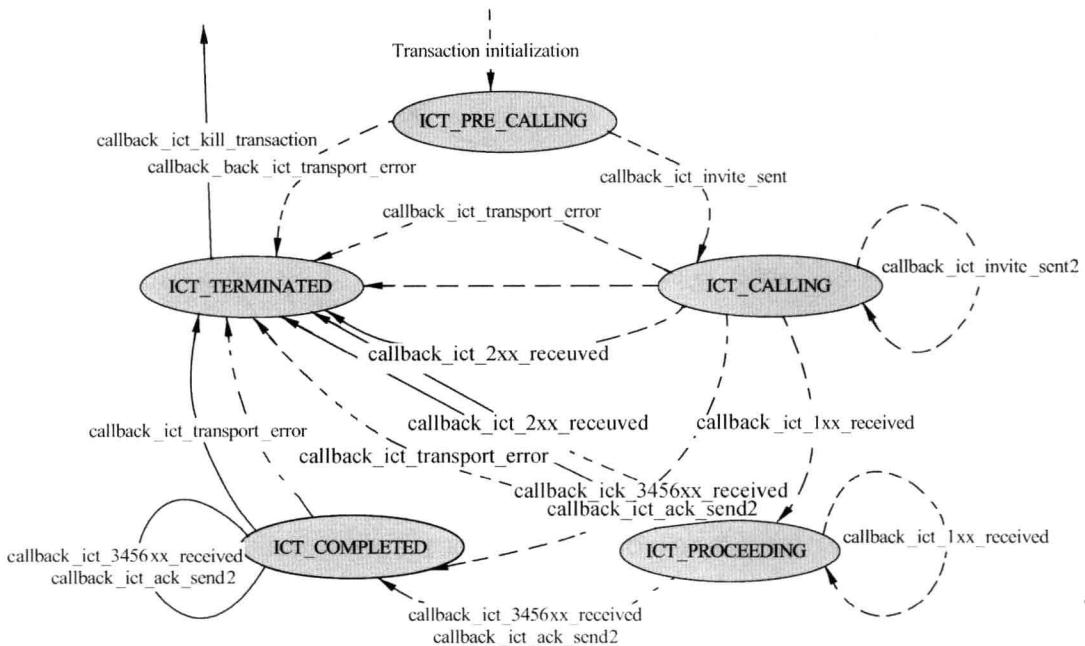


图 20.8 函数 `ict_retransmit_ack()`产生的状态转移

### 20.3.2 NICT (Non-Invite Client (outgoing) Transaction) 状态机

NICT 状态机整个实现过程在 `nict_fsm.c` 文件中定义，在数组 `transition_t nict_transition[12]` 中定义了状态机的初始状态及初始状态接收的事件，各个事件函数表示了状态机处理该事件的跳转情况。下面直接给出 NICT 的状态机跳转图如图 20.9 所示，读者可以根据手中的代码自行推导和验证。NICT 状态机中和 ICT 状态机类似，具有以下 5 种状态：

- NICT\_PRE\_TRYING;
- NICT\_TRYING;
- NICT\_PROCEEDING;
- NICT\_COMPLETED;
- NICT\_TERMINATED.

状态之间的事件函数包括：

- 错误处理 `nict_handle_transport_error()`
- 发送请求 `nict_snd_request()`
- 超时事件 `osip_nict_timeout_k_event()`
- 6 类应答函数 `nict_rcv_123456xx()`

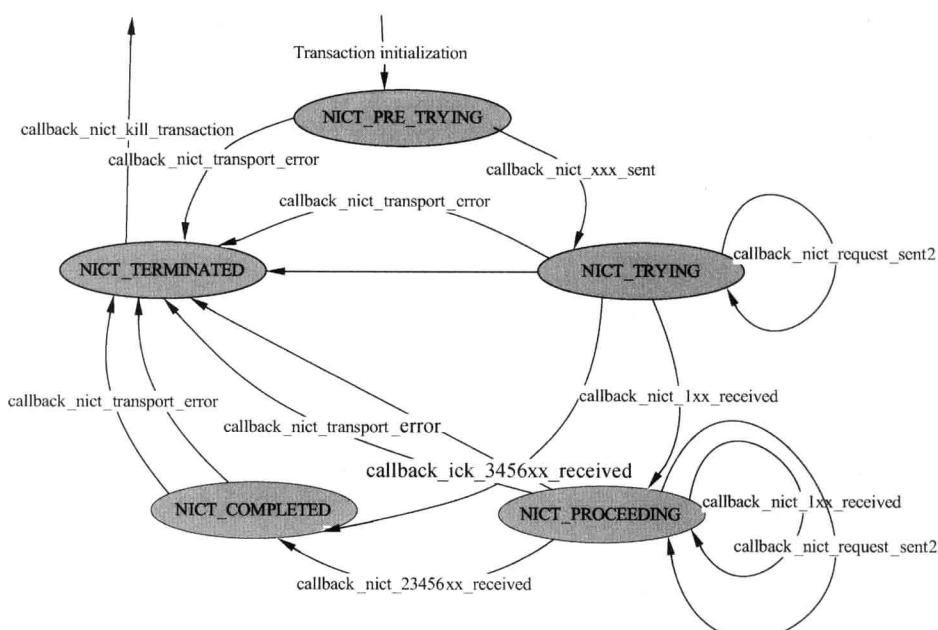


图 20.9 NICT 状态机的状态迁移图

### 20.3.3 IST (Invite Server (incoming) Transaction) 状态机

IST 状态机整个实现过程在 `ist_fsm.c` 文件中定义，在数组 `transition_t ist_transition[11]` 中定义了状态机的初始状态及初始状态接收的事件，各个事件函数表示了状态机处理该事件的跳转情况。下面直接给出 IST 的状态机跳转图，如图 20.10 所示。

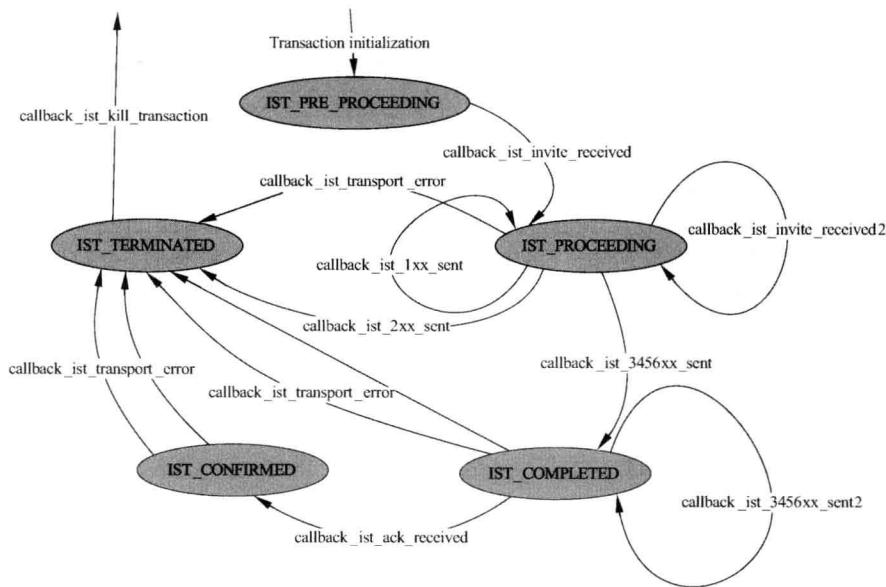


图 20.10 IST 状态机的状态迁移图

### 20.3.4 NIST (Non-Invite Server (incoming) Transaction) 状态机

NIST 状态机整个实现过程在 `nist_fsm.c` 文件中定义，在数组 `transition_t nist_transition[10]` 中定义了状态机的初始状态及初始状态接收的事件，各个事件函数表示了状态机处理该事件的跳转情况。下面直接给出 NIST 的状态机跳转图，如图 20.11 所示。

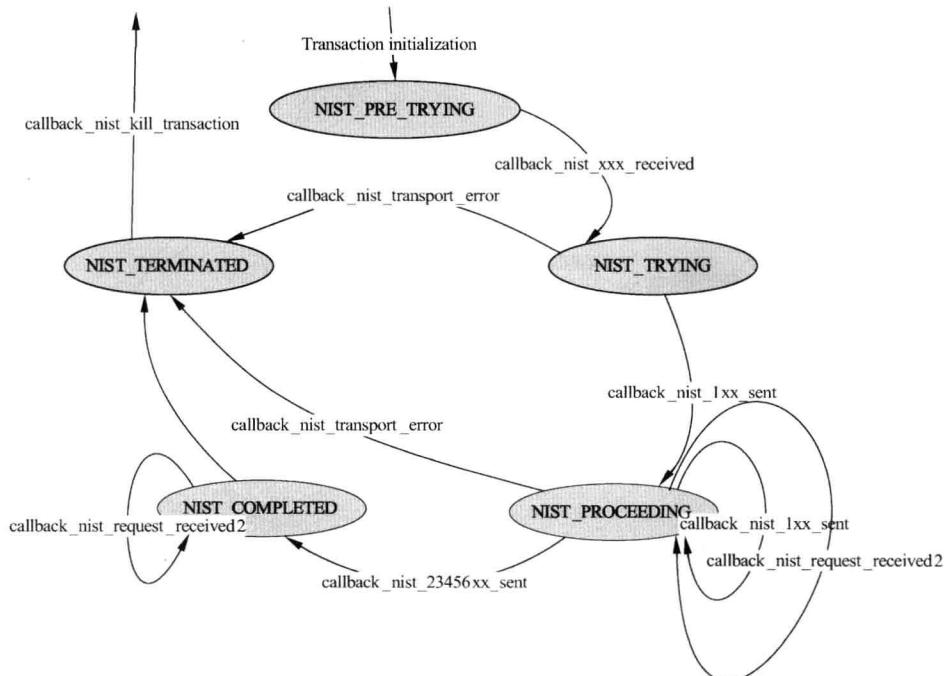


图 20.11 NIST 状态机的状态迁移图

## 20.4 oSIP 解析器

每种类型的 SIP 解析器所提供的 API 基本类似。通过函数 `osip_xxx_init()` 创建解析类型 `xxx`，函数 `osip_xxx_free()` 用于释放解析类型，函数 `osip_xxx_parse()` 用于从字符串到解析类型的解析，函数 `osip_xxx_to_str()` 用于从解析类型到字符串的解析，函数 `osip_xxx_clone()` 用于从旧的解析类型复制到新的解析类型，函数 `osip_xxx_set_header()` 用于设置解析类的头部，函数 `osip_xxx_set_contenttype()` 用于设置解析类型内容。下面通过一个 `body` 类型实例进行说明。

### 20.4.1 初始化解析类型函数 `osip_body_init()`

函数 `osip_body_init()` 为类型 `body` 分配空间，并初始化各个字段，并且将 `body` 的头部也初始化为 0，如果整个初始化过程成功，则返回 `OSIP_SUCCESS`。

```
Int osip_body_init (osip_body_t ** body)
{
    /*为 body 分配空间*/
    *body = (osip_body_t *) osip_malloc (sizeof (osip_body_t));
    /*分配失败则退出*/
    if (*body == NULL)
        return OSIP_NOMEM;
    /*为 body 的各个字段赋值*/
    (*body)->body = NULL;
    (*body)->content_type = NULL;
    (*body)->length = 0;
    /*为 body 的头部分配空间*/
    (*body)->headers = (osip_list_t *) osip_malloc (sizeof (osip_list_t));
    if ((*body)->headers == NULL)
    {
        osip_free (*body);
        *body = NULL;
        return OSIP_NOMEM;
    }
    /*初始化头部*/
    osip_list_init ((*body)->headers);
    return OSIP_SUCCESS;
}
```

### 20.4.2 释放函数 `osip_body_free()`

函数 `osip_body_free()` 用于释放函数 `osip_body_init()` 初始化的结构体 `body`。

```
void osip_body_free (osip_body_t * body)
{
    if (body == NULL)
        return;
    osip_free (body->body);
```

```

if (body->content_type != NULL)
{
    /*释放内容类型*/
    osip_content_type_free (body->content_type);
}
/*移除 body 头部*/
osip_list_special_free(body->headers, (void *(*)(void *)) &osip_header_free;
/*释放头部占用空间*/
osip_free (body->headers);
/*释放 body 结构体占用的空间*/
osip_free (body);
}

```

### 20.4.3 字符串到 body 类型转换函数 osip\_body\_parse()

函数 osip\_body\_parse()用于将字符串类型转换成 body 类型。将输入参数 start\_of\_body 中的字符串以内存复制的方式赋值给 body 的 body 字段。

```

Int osip_body_parse (osip_body_t * body, const char *start_of_body, size_t
length)
{
    /*转换前进行安全检查*/
    if (body == NULL)
        return OSIP_BADPARAMETER;
    if (start_of_body == NULL)
        return OSIP_BADPARAMETER;
    if (body->headers == NULL)
        return OSIP_BADPARAMETER;
    /*为 body 的 body 字段分配空间*/
    body->body = (char *) osip_malloc (length + 1);
    if (body->body == NULL)
        return OSIP_NOMEM;
    /*以内存复制的方式将 start_of_body 中的内容复制长度为 length 到地址为 body-
     >body 处*/
    memcpy (body->body, start_of_body, length);
    /*填充字符串结尾标志*/
    body->body[length] = '\0';
    /*设置 body 长度*/
    body->length = length;
    return OSIP_SUCCESS;
}

```

### 20.4.4 body 类型到字符串类型转换函数 osip\_body\_to\_str()

函数 osip\_body\_to\_str()与函数 osip\_body\_parse()对应，用于将 body 中的 body 字段的内容复制到参数 dest 中。

```

Int osip_body_to_str (const osip_body_t * body, char **dest, size_t *
str_length)
{
    char *tmp_body;

```

```

char *tmp;
char *ptr;
int pos;
int i;
size_t length;

*dest = NULL;
*str_length = 0;
/*用于安全性检查部分*/
if (body == NULL)
    return OSIP_BADPARAMETER;
if (body->body == NULL)
    return OSIP_BADPARAMETER;
if (body->headers == NULL)
    return OSIP_BADPARAMETER;
if (body->length <= 0)
    return OSIP_BADPARAMETER;
/*分配空间用于保存复制中间变量*/
length = 15 + body->length + (osip_list_size (body->headers) * 40);
tmp_body = (char *) osip_malloc (length);
if (tmp_body == NULL)
    return OSIP_NOMEM;
ptr = tmp_body;                                /*保存字符串的初始地址*/

if (body->content_type != NULL)
{
    /*在 tmp_body 后面加上字符串 content-type:*/
    tmp_body = osip_strn_append (tmp_body, "content-type: ", 14);
    /*将 body->content_type 转化为字符串保存在 tmp 中*/
    i = osip_content_type_to_str (body->content_type, &tmp);
    if (i != 0)
    {
        /*转换失败，则释放空间并退出*/
        osip_free (ptr);
        return i;
    }
    /*检查 tmp_body 的空间是否够用，不够则重新分配*/
    if (length < tmp_body - ptr + strlen (tmp) + 4)
    {
        size_t len;

        len = tmp_body - ptr;
        length = len + strlen (tmp) + 4;
        ptr = osip_realloc (ptr, length);
        tmp_body = ptr + len;
    }
    /*将 tmp 添加到 tmp_body 后面*/
    tmp_body = osip_str_append (tmp_body, tmp);
    /*释放为 tmp 分配的空间*/
    osip_free (tmp);
    /*在 tmp_body 后添加字符串 CRLF*/
    tmp_body = osip_strn_append (tmp_body, CRLF, 2);
}

pos = 0;
while (!osip_list_eol (body->headers, pos))
{
    osip_header_t *header;
}

```

```

/*获得 body 头部*/
header = (osip_header_t *) osip_list_get (body->headers, pos);
/*将得到的头部转化为字符串保存在 tmp 中*/
i = osip_header_to_str (header, &tmp);
if (i != 0)
{
    /*转换失败，则释放空间并退出*/
    osip_free (ptr);
    return i;
}
/*检查 tmp_body 的空间是否够用，不够则重新分配*/
if (length < tmp_body - ptr + strlen (tmp) + 4)
{
    size_t len;

    len = tmp_body - ptr;
    length = length + strlen (tmp) + 4;
    ptr = osip_realloc (ptr, length);
    tmp_body = ptr + len;
}
/*将 tmp 添加到 tmp_body 后面*/
tmp_body = osip_str_append (tmp_body, tmp);
/*释放为 tmp 分配的空间*/
osip_free (tmp);
/*在 tmp_body 后添加字符串 CRLF*/
tmp_body = osip_strn_append (tmp_body, CRLF, 2);
pos++;
}

if ((osip_list_size (body->headers) > 0) || (body->content_type != NULL))
{
    tmp_body = osip_strn_append (tmp_body, CRLF, 2);
}
if (length < tmp_body - ptr + body->length + 4)
{
    size_t len;

    len = tmp_body - ptr;
    length = length + body->length + 4;
    ptr = osip_realloc (ptr, length);
    tmp_body = ptr + len;
}
/*将 body 上的 body 字段向地址为 tmp_body 处复制长度为 body->length 个字符*/
memcpy (tmp_body, body->body, body->length);
tmp_body = tmp_body + body->length;

/* end of this body */
if (str_length != NULL)
    *str_length = tmp_body - ptr;
*dest = ptr;
return OSIP_SUCCESS;
}

```

## 20.4.5 克隆函数 osip\_body\_clone()

函数 osip\_body\_clone() 将 body 复制给 dest。

```

Int osip_body_clone (const osip_body_t * body, osip_body_t ** dest)
{
    int i;
    osip_body_t *copy;
    /*安全性检查*/
    if (body == NULL || body->length <= 0)
        return OSIP_BADPARAMETER;
    /*创建一个新的 body 类型*/
    i = osip_body_init (&copy);
    if (i != 0)
        return i;
    /*为新建的类型的 body 字段分配空间*/
    copy->body = (char *) osip_malloc (body->length + 2);
    /*分配失败则退出*/
    if (copy->body==NULL)
        return OSIP_NOMEM;
    /*设置新建结构体 copy 的长度*/
    copy->length = body->length;
    /*复制 body 的字段 body 到 copy 的字段 body*/
    memcpy (copy->body, body->body, body->length);
    /*设置字符结束标志*/
    copy->body[body->length] = '\0';

    if (body->content_type != NULL)
    {
        /*克隆 content_type*/
        i = osip_content_type_clone (body->content_type, &(copy->content_type));
        if (i != 0)
        {
            /*克隆 content_type 失败则退出且释放空间*/
            osip_body_free (copy);
            return i;
        }
    }
    /*克隆结构体的头部*/
    i = osip_list_clone (body->headers, copy->headers, &osip_header_clone);
    if (i != 0)
    {
        /*克隆结构体的头部失败则退出*/
        osip_body_free (copy);
        return i;
    }
    /*将新创建并进行了克隆的结构体赋给 dest 指针*/
    *dest = copy;
    return OSIP_SUCCESS;
}

```

## 20.4.6 oSIP 解析器分类

oSIP 解析器分为 3 类：SIP 解析器、SDP 解析器和 URL 解析器。前面提到的是 SIP 解析器，另外还有两类解析器。3 类解析器的作用分别如下：

- SIP 解析器主要用于解析 SIP 头域及其相应的操作。
- SDP 解析器用于解析 SDP 包及其相关的操作。
- URL 解析器用于处理 SIP URI 的 host、port、username、password 和 scheme 等 get() 和 set() 操作。

SIP URI 是通过 SIP 呼叫对方的 SIP 地址方案，即一个 SIP URI 就是一个用户的 SIP 电话号码。SIP URI 类似电子邮件地址，书写格式如下：

```
SIP URI = sip:a@b:Port
```

其中，a 表示用户名，b 表示服务主机（域名或 IP）。下面举几个例子进行说明 SIP URI 的书写方法。例子如下：

<code>sip:bob@212.123.1.213</code>	//bob 为用户名，212.123.1.213 为服务器 IP 地址
<code>sip:bob@biloxi.com</code>	//bob 为用户名，biloxi.com 为服务器主机名
<code>sip:5201314@biloxi.com</code>	//5201314 为用户名，biloxi.com 为服务器主机名

## 20.5 oSIP 事务层

SIP 是一个基于事务处理的协议：部件之间的交互是通过一系列的消息交换所完成的。特别是一个 SIP 事务由一个单个请求和这个请求的所有应答组成，这些应答包括了零个或者多个临时应答，以及一个或者多个终结应答。

事务分为客户端和服务端两方。客户端的事务是客户端事务，服务器端的事务就是服务端事务。客户端事务发出请求，并且服务端事务送回应答。事务层比较重要的两个概念就是事务和事件，在 oSIP 中用结构体 `osip_transaction_t` 和结构体 `osip_event_t` 表示。

结构体 `osip_transaction_t` 的定义：

```
typedef struct osip_transaction osip_transaction_t;
struct osip_transaction
{
    void *your_instance;           /* 用户定义指针 */
    int transactionid;            /* 内部事务 ID */
    osip_fifo_t *transactionff;   /* 用于存放事件的 fifo 队列 */

    osip_via_t *topvia;           /* CALL-LEG definition (Top Via) */
    osip_from_t *from;             /* CALL-LEG definition (From) */
    osip_to_t *to;                /* CALL-LEG definition (To) */
    osip_call_id_t *callid;       /* CALL-LEG definition (Call-ID) */
    osip_cseq_t *cseq;             /* CALL-LEG definition (CSeq) */
```

```

osip_message_t *orig_request;    /**初始请求*/
osip_message_t *last_response;   /**最后响应*/
osip_message_t *ack;             /**应答请求*/

state_t state;                  /**事务的当前状态*/

time_t birth_time;              /**事务开始时间*/
time_t completed_time;          /**事务结束时间*/

int in_socket;                  /**< Optional socket for incoming message*/
int out_socket;                 /**< Optional place for outgoing message*/

void *config;                   /**@internal transaction is managed by osip_t*/

osip_fsm_type_t ctx_type;       /**事务类型*/
osip_ict_t *ict_context;        /**INVITE CLIENT TRANSACTION 结构体*/
osip_ist_t *ist_context;        /**NON-INVITE CLIENT TRANSACTION 结构体*/
osip_nict_t *nict_context;      /**INVITE SERVER TRANSACTION 结构体*/
osip_nist_t *nist_context;      /**NON-INVITE SERVER TRANSACTION 结构体*/

osip_srv_record_t record;       /**记录 SERVER 入口信息的结构体*/
};


```

结构体 osip\_event\_t 的定义:

```

typedef struct osip_event osip_event_t;
struct osip_event
{
    type_t type;                /**事件类型*/
    int transactionid;          /**与 osip 事务相关的 ID*/
    osip_message_t *sip;         /**< SIP message (optional)*/
};

```

处理事务的函数在 osip\_tansaction.c 中定义, 函数 osip\_transaction\_init() 用于事务的初始化, 函数 osip\_transaction\_free() 用于从 osip 栈中移除事务, 函数 osip\_transaction\_add\_event() 用于向事务的 fifo 队列中添加事件, 函数 osip\_transaction\_execute() 用于执行事务处理, 调用前面分析过的状态机处理事务过程。由于篇幅的限制这里只分析事务初始化函数 osip\_transaction\_init(), 其他函数的定义读者可参考源代码自行分析其实现细节。

事务初始化函数 osip\_transaction\_init() 用于构造一个事务, 并初始化该事务的各个字段, 构造该事务的 fifo 队列用于存放事件, 并初始化该事件队列, 根据输入事务类型设置其事务的类型, 根据事务的类型设置事务的初始状态, 并初始化此状态机。省略函数 osip\_transaction\_init() 的安全检查、trace 机制等, 其核心部分代码如下:

```

int osip_transaction_init (osip_transaction_t ** transaction,
                           osip_fsm_type_t ctx_type, osip_t * osip,
                           osip_message_t * request)
{
    static int transactionid = 1;
    osip_via_t *topvia;
}

```

```

int i;
time_t now;
/*为构造的事务结构体分配空间*/
*transaction = (osip_transaction_t *) osip_malloc (sizeof (osip_
transaction_t));
/*获得当前时间*/
now = time (NULL);
/*初始化事务结构体*/
memset (*transaction, 0, sizeof (osip_transaction_t));
/*设置事务的创建时间和 ID, 事务计数自动加 1*/
(*transaction)->birth_time = now;
(*transaction)->transactionid = transactionid;
transactionid++;
/*从输入参数 request 中获得字段 vias 的值赋给 topvia*/
topvia = osip_list_get (&request->vias, 0);
/*设置事务的 topvia 字段*/
__osip_transaction_set_topvia (*transaction, topvia);
/*下面分别对事务的 from、to、call_id、cseq 字段进行设置*/
i = __osip_transaction_set_from (*transaction, request->from);
i = __osip_transaction_set_to (*transaction, request->to);
i = __osip_transaction_set_call_id (*transaction, request->call_id);
i = __osip_transaction_set_cseq (*transaction, request->cseq);
/*设置事务的 orig_request 和 config*/
(*transaction)->orig_request = NULL;
(*transaction)->config = osip;
/*为事务的事件队列分配空间*/
(*transaction)->transactionff = (osip_fifo_t *) osip_malloc (sizeof
(osip_fifo_t));
/*初始化事件队列*/
osip_fifo_init ((*transaction)->transactionff);
/*设置事务类型*/
(*transaction)->ctx_type = ctx_type;
/*将事务的 ict_context、ist_context、nict_context 和 nist_context 均设置
为空 */
(*transaction)->ict_context = NULL;
(*transaction)->ist_context = NULL;
(*transaction)->nict_context = NULL;
(*transaction)->nist_context = NULL;
/*如果事务的类型为 ICT*/
if (ctx_type == ICT)
{
    /*设置事务的初始状态，并初始化 ict 状态机*/
    (*transaction)->state = ICT_PRE_CALLING;
    __osip_ict_init (&((*transaction)->ict_context), osip, request);
    /*将事务加入 ict 事务处理队列*/
    __osip_add_ict (osip, *transaction);
} else if (ctx_type == IST)
{
    /*设置事务的初始状态，并初始化 ist 状态机*/
}

```

```

(*transaction)->state = IST_PRE_PROCEEDING;
i = __osip_ist_init (&((*transaction)->ist_context), osip, request);
/*将事务加入 ist 事务处理队列*/
__osip_add_ist (osip, *transaction);
} else if (ctx_type == NICT)
{
    /*设置事务的初始状态，并初始化 nict 状态机*/
    (*transaction)->state = NICT_PRE_TRYING;
    i = __osip_nict_init (&((*transaction)->nict_context), osip,
request);
    /*将事务加入 nict 事务处理队列*/
    __osip_add_nict (osip, *transaction);
} else
{
    /*设置事务的初始状态，并初始化 nist 状态机*/
    (*transaction)->state = NIST_PRE_TRYING;
    i = __osip_nist_init (&((*transaction)->nist_context), osip,
request);
    /*将事务加入 nist 事务处理队列*/
    __osip_add_nist (osip, *transaction);
}
return OSIP_SUCCESS;
}

```

## 20.6 SIP 建立会话的过程

下面采用序列图的方式表示 A、B 两个用户间通过 SIP 消息交换建立会话的过程，如图 20.12 所示。A 通过 B 的 SIP 标志“呼叫”B，这个 SIP 标志是统一分配的资源（Uniform Resource Identifier URI）称做 SIP URI。它很像一个 E-mail 地址，典型的 SIP URI 包括一个用户名和一个主机名。在这个范例中，SIP URI 是 sip:bbb@bwww.com，bwww.com 是 B 的 SIP 服务提供商。A 有一个 SIP URI: sip:aaa@awww.com。

下面是 A 通过自己的 Softphone 呼叫 B 的 SIP phone，并建立和 B 的会话的整个过程。A 不知道 B 或者 B 的 SIP 服务器的位置，所以 A 首先将请求发送到 A 的 SIP 服务器 awww.com。SIP 服务器 awww.com 收到 INVITE 请求后，回应 100(Trying) 给 A 的 Softphone，并在 via 头上加入自己的地址转发 INVITE 请求给 B 的 SIP 服务器 bwww.com。B 的 SIP 服务器 bwww.com 收到该 INVITE 请求后回应 100 (Trying) 给代理服务器 awww.com，并在 via 头上加入自己的地址转发 INVITE 请求给 B 的 SIP phone。到此时 B 的 SIP phone 就提示 B，A 在呼叫他。B 的 SIP phone 这时会发送一个 180 (ringing) 响应，该回应会通过原路返回给 A。当 B 接通 SIP phone 时，SIP phone 就会发送 200 (OK) 回应，该回应最后到达 A 的 Softphone，到此 A 和 B 就可以进行语音或视频通话。通话结束后，B 挂机，B 的 SIP phone 就会发送 BYE 给 A 的 Softphone，A 的 Softphone 会回应 200 (OK)。

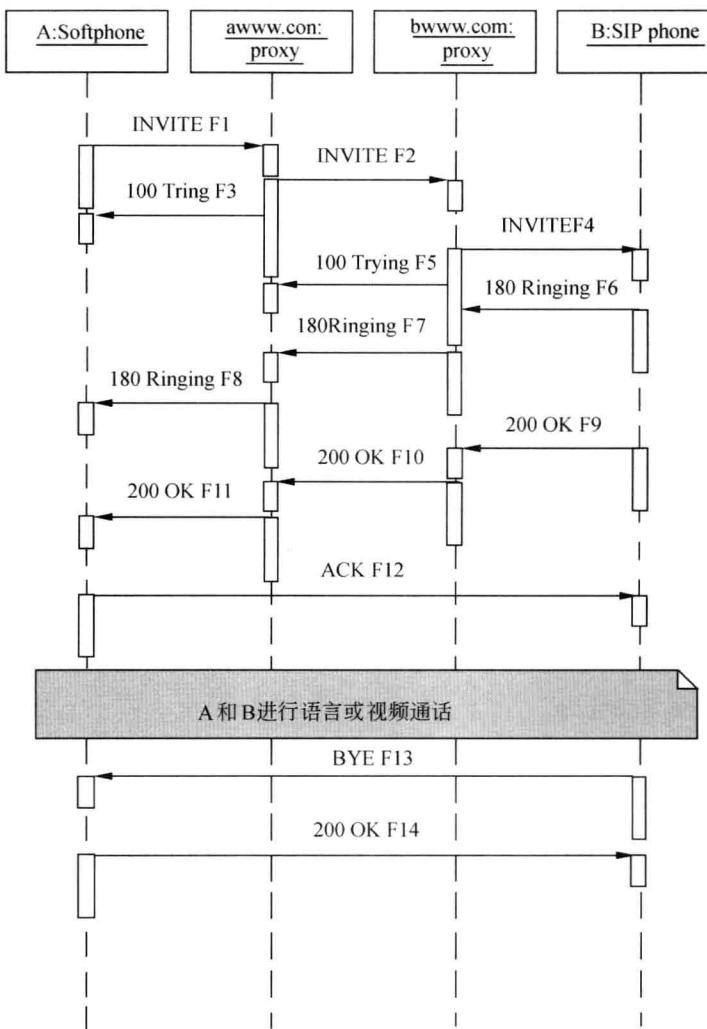


图 20.12 SIP 会话建立过程

## 20.7 RTP 协议

RTP (real-time transport protocol) 实时传输协议，在多点传送（多播）或单点传送（单播）的网络服务上提供端到端的网络传输功能，适合应用程序传输实时数据，如音频、视频或者仿真数据。通过 SIP 协议建立起会话，就可以通过 RTP 传输这些会话的实时数据。

介绍 RTP 协议时，首先了解一下 RTP 协议的一些基本概念，再针对代码查看 RTP 是如何发送和接收 RTP 包，然后细化这个过程。

### 20.7.1 RTP 基本概念

RTP 提供实时的端到端的数据传输服务。传输的数据类型有交互式的音频和视频。服

务的内容包括有效载荷、序列号、时间戳和传输监测控制。应用程序在 UDP 上运行 RTP 来使用它的多路技术与校验和服务。RTP 本身不提供任何机制确保实时传输或服务质量保证，而是由低层的服务来保证。它不保证下层网络可靠，也不保证按顺序传送数据包。RTP 包含的序列号使得接受方可以重构发送方的数据包顺序，但序列号也可以确定一个数据包的正确位置，例如，在视频解码时不用按顺序地对数据包进行解码。

RTP 包括两个密切相关的部分：

- ❑ 实时传输协议 RTP，主要用于实时数据传输。
- ❑ RTP 控制协议 RTCP，用于服务质量监控与反馈、媒体间的同步、传达会议中参与者的信息。不必支持一个应用程序中所有的通信控制条件。

在分析源码前需要了解相关的基本概念，下面列出了一些 RTP 的基本概念，对这些概念深入的了解可以参看 RFC3550。

- ❑ RTP 负载（RTP payload）：RTP 包中的净数据，例如，音频样本或压缩好的视频数据。
- ❑ RTP 包（RTP packet）：一种数据包，其组成部分包括 RTP 包固定的报头、可能为空的作用源（contributing sources）列表和负载数据。下层协议对 RTP 包的进行封包时，一个包中可以包含一个 RTP 包，也可包含多个 RTP 包。
- ❑ RTCP 包（RTCP packet）：一种控制包，其组成部分包括 RTCP 包固定的报头和一个结构化的部分，该部分具体元素还应该根据不同 RTCP 包的类型而定。一般下层协议的包由多个 RTCP 包组成。
- ❑ 端口（Port）：用于在同一主机中区分不同目的地。RTP 需要依靠低层协议提供的多种机制，如“端口”用于多路复用会话中的 RTP 包和 RTCP 包。
- ❑ 传输地址（Transport address）：是网络地址与端口的结合，用来指定一个唯一的传输层次的终端，例如，一个 IP 地址与一个 UDP 端口的组合。包是从源传输地址发往目的传输地址。
- ❑ RTP 媒体类型（RTP media type）：一个 RTP 媒体类型是一个单独 RTP 会话所承载的负载类型的集合。RTP 配置文件中将 RTP 媒体类型分配给 RTP 负载类型。
- ❑ 多媒体会话（Multimedia session）：在一个会话公共组中，并发的 RTP 会话的集合。例如，一个视频会议（为多媒体会话）中，可能包含一个音频 RTP 会话和一个视频 RTP 会话。
- ❑ RTP 会话（RTP session）：一组参与者通过 RTP 协议进行通信时所产生的关联。一个参与者可能同时参与多个 RTP 会话。在一个多媒体会话中，除了使用编码方式将多种媒体多路复用到单一数据流中外，每种媒体都将使用各自的 RTCP 包，通过单独的 RTP 会话进行传送。通过使用不同的目的传输地址对（一个网络地址加上一对分别用于 RTP 和 RTCP 的端口，构成了一个传输地址对）来接收不同的会话，参与者能将多个 RTP 会话分隔开。IP 多播时，单个 RTP 会话中的全部参与者，共享一个公用目的传输地址对；单播时，参与者将使用不同的目的传输地址对，个体单播网络地址加上一个端口对。对于单播而言，参与者可以使用相同的端口对收听其他全部参与者，也可能使用不同的端口对分别收听其他各个参与者。

## 20.7.2 发送 RTP

RTP 的发送过程在 `rtpsend.c` 中定义，该文件的 `main()` 函数清晰地描述构造 RTP、设置 RTP Session 到发送 RTP 包的整个过程，下面是其代码：

```

int main(int argc, char *argv[])
{
    RtpSession *session;
    unsigned char buffer[160];
    int i;
    FILE *infile;
    char *ssrc;
    uint32_t user_ts=0;
    int clockslide=0;
    int jitter=0;
    if (argc<4){
        printf("%s", help);
        return -1;
    }
    for(i=4;i<argc;i++){
        if (strcmp(argv[i],"--with-clockslide")==0) {
            i++;
            if (i>=argc) {
                printf("%s", help);
                return -1;
            }
            /*将输入的参数字符串转化为整数类型*/
            clockslide=atoi(argv[i]);
            ortp_message("Using clockslide of %i milisecond every 50
packets.",clockslide);
        }else if (strcmp(argv[i],"--with-jitter")==0) {
            ortp_message("Jitter will be added to outgoing stream.");
            i++;
            if (i>=argc) {
                printf("%s", help);
                return -1;
            }
            jitter=atoi(argv[i]);
        }
    }
    /*RTP 库的初始化，在调用 RTP 相关的 API 之前首先调用该函数*/
    ortp_init();
    /*初始化 RTP 调度程序*/
    ortp_scheduler_init();
    /*设置日志文件的级别，该函数一般用于开发或者调试中追踪相关的信息*/
    ortp_set_log_level_mask(ORTP_MESSAGE|ORTP_WARNING|ORTP_ERROR);
    /*创建一个 RTP session，用于发送数据*/
    session=rtp_session_new(RTP_SESSION_SENDONLY);
    /*设置 session 的调度模式*/
    rtp_session_set_scheduling_mode(session,1);
    /*设置 session 为阻塞模式，该模式默认在调度程序中执行，因此设置该模式时同时会设置
Session 为调度模式*/
    rtp_session_set_blocking_mode(session,1);
    /*设置连接模式，如果设置了连接模式，当一个 socket 到达目的地时将会调用系统的
connect() 函数*/
}

```

```

rtp_session_set_connected_mode(session,TRUE);
/*设置 RTP 的远程地址, 即 RTP 包准备发往的地址*/
rtp_session_set_remote_addr(session,argv[2],atoi(argv[3]));
/*设置期望的 session 负载类型*/
rtp_session_set_payload_type(session,0);
/*ssrc 的作用: 在随机的时间间隔中, 一个参与者必须检测其他参与者是否已经超时。为此,
对接收者 (we_sent 为 false) , 要计算决定性时间间隔 Td, 如果从时刻 Tc-M*Td (M 为
超时因子, 默认为 5 秒) 开始, 未发送过 RTP 或 RTCP 包, 则超时。其 SSRC 将被从列表中移
除, 成员被更新。在发送者列表中也要进行类似的检测。发送者列表中, 任何从时间 tc-2T
(在最后两个 RTCP 报告时间间隔内) 未发送 RTP 包的发送者, 其 SSRC 从发送者列表中移除,
列表更新*/
ssrc=getenv("SSRC");
if (ssrc!=NULL) {
    printf("using SSRC=%i.\n",atoi(ssrc));
    rtp_session_set_ssrc(session,atoi(ssrc));
}
/*针对不同平台的可移植性的系统函数, 打开文件操作*/
#ifndef _WIN32
infile=fopen(argv[1],"r");
#else
infile=fopen(argv[1],"rb");
#endif
/*对于打开文件操作的安全性检查*/
if (infile==NULL) {
    perror("Cannot open file");
    return -1;
}
/*系统收到信号 SIGINT 就会通过传入的地址调用函数 stophandler () , 该函数就是设置
runcond =0*/
signal(SIGINT,stophandler);
while( ((i=fread(buffer,1,160,infile))>0) && (runcond) )
{
    /*从缓冲区中发送带有时间戳的 RTP 数据包到目的地址*/
    rtp_session_send_with_ts(session,buffer,i,user_ts);
    user_ts+=160;
    if (clockslide!=0 && user_ts%(160*50)==0) {
        ortp_message("Clock sliding of %i miliseconds now",clockslide);
        /*设置时间误差*/
        rtp_session_make_time_distortion(session,clockslide);
    }
    /*下面是模拟的突发延迟包*/
    if (jitter && (user_ts%(8000)==0)) {
        struct timespec pausetime, remtime;
        ortp_message("Simulating late packets now (%i milliseconds)",
                     jitter);
        pausetime.tv_sec=jitter/1000;
        pausetime.tv_nsec=(jitter%1000)*1000000;
        while(nanosleep(&pausetime,&remtime)==-1 && errno==EINTR){
            pausetime=remtime;
        }
    }
}
/*关闭文件*/
fclose(infile);
/*关闭 Session*/
rtp_session_destroy(session);
/*退出 RTP, 包括关闭调度程序*/

```

```

    ortp_exit();
    /*打印 RTP 包的一些情况，包括收到包、丢失包等信息*/
    ortp_global_stats_display();
    return 0;
}

```

### 20.7.3 接收 RTP

RTP 的接收过程在 `rtprecv.c` 中描述，接收过程包括初始化 RTP、初始化调度器、创建接收 Session、设置连接模式、设置对称 RTP、接收数据包存放在缓冲区、将缓冲区数据写入文件、关闭 Session、退出 RTP 等过程。下面是该接收 RTP 过程的代码：

```

int main(int argc, char*argv[])
{
RtpSession *session;
unsigned char buffer[160];
int err;
uint32_t ts=0;
int stream_received=0;
FILE *outfile;
int local_port;
int have_more;
int i;
int format=0;
int soundcard=0;
int sound_fd=0;
int jittcomp=40;
bool_t adapt=TRUE;
/*将第 2 个参数字符串转化为整数类型的本地端口号*/
local_port=atoi(argv[2]);
if (local_port<=0) {
    printf("%s",help);
    return -1;
}
/*根据参数指定采用哪种 PCM 编码算法，u 律还是 A 律*/
for (i=3;i<argc;i++)
{
    if (strcmp(argv[i],"--noadapt")==0) adapt=FALSE;
    if (strcmp(argv[i],"--format")==0) {
        i++;
        if (i<argc){
            if (strcmp(argv[i],"mulaw")==0) {
                format=MULAW;
            }else
                if (strcmp(argv[i],"alaw")==0) {
                    format=ALAW;
                }else{
                    printf("Unsupported format %s\n",argv[i]);
                    return -1;
                }
        }
    }
    else if (strcmp(argv[i],"--soundcard")==0) {
        soundcard=1;
    }
    else if (strcmp(argv[i],"--with-jitter")==0) {
        i++;
    }
}

```

```

    if (i<argc) {
        jittcomp=atoi(argv[i]);
        printf("Using a jitter buffer of %i milliseconds.\n",
        jittcomp);
    }
}
/*打开输出文件，准备向文件中写内容*/
outfile=fopen(argv[1], "wb");
if (outfile==NULL) {
    perror("Cannot open file for writing");
    return -1;
}
/*声卡初始化包括打开声音设备文件、设置采样频率、设置声道数、设置音频数据格式*/
if (soundcard){
    sound_fd=sound_init(format);
}
/*RTP 初始化，在调用 RTP 库前必须要做的工作*/
ortp_init();
/*RTP 调度程序初始化*/
ortp_scheduler_init();
/*设置日志文件的级别，该函数一般用于开发或者调试中追踪相关的信息*/
ortp_set_log_level_mask(ORTP_DEBUG|ORTP_MESSAGE|ORTP_WARNING|
ORTP_ER_ROR);
/*系统收到信号 SIGINT 就会通过传入的地址调用函数 stop_handler ()，该函数就是设
置 cond =0*/
signal(SIGINT,stop_handler);
/*创建一个新的 Session，用于接收 RTP 数据包*/
session=rtp_session_new(RTP_SESSION_RECVONLY);
/*设置为调度模式*/
rtp_session_set_scheduling_mode(session,1);
/*设置 Session 为阻塞模式，该模式默认在调度程序中执行，因此设置该模式时同时会设置
session 为调度模式，在该函数中调用了函数 rtp_session_set_scheduling_mode
(session,1);*/
rtp_session_set_blocking_mode(session,1);
/*这里设置本地地址用于监听 RTP 包，如果本地地址没有设置，则采用默认的 IP 地址和任意
端口*/
rtp_session_set_local_addr(session,"0.0.0.0",atoi(argv[2]));
/*设置连接模式，如果设置了连接模式，当一个 socket 到达目的地时将会调用系统的
connect() 函数*/
rtp_session_set_connected_mode(session,TRUE);
/*设置为对称 RTP*/
rtp_session_set_symmetric_rtp(session,TRUE);
/*设置可以调整时延或者抖动*/
rtp_session_enable_adaptive_jitter_compensation(session,adapt);
/*设置时延或者抖动的补偿参数*/
rtp_session_set_jitter_compensation(session,jittcomp);
/*设置载荷类型*/
rtp_session_set_payload_type(session,0);
/*当用户函数支持的信号量在处理的过程中发生了变化时，用户注册的回调函数将会被通知
*/
rtp_session_signal_connect(session,"ssrc_changed", (RtpCallback)
ssrc_cb,0);
rtp_session_signal_connect(session,"ssrc_changed", (RtpCallback)
rtp_s ession_reset,0);

```

```

while(cond)
{
    have_more=1;
    while (have_more) {
        /*接收到来的 RTP 流存放到缓冲区*/
        err=rtp_session_recv_with_ts(session,buffer,160,ts,&have_more);
        if (err>0) stream_received=1;
        /*在第一个 RTP 数据包返回前防止写静音数据*/
        if ((stream_received) && (err>0)) {
            size_t ret = fwrite(buffer,1,err,outfile);
            if (sound_fd>0)
                /*播放声音*/
                ret = write(sound_fd,buffer,err);
        }
        ts+=160;
        //ortp_message("Receiving packet.");
    }
    /*通信结束后销毁 Session*/
    rtp_session_destroy(session);
    /*退出 RTP*/
    ortp_exit();
    /*打印 RTP 的信息*/
    ortp_global_stats_display();

    return 0;
}

```

## 20.8 Linphone 编译与测试

在对 Linphone 有一定了解的基础上，讲解其编译和测试过程。其编译包括 X86 平台的编译和 ARM 平台的编译。

### 20.8.1 编译 Linphone 需要的软件包

编译 Linphone 依赖一些相关的库，这里列出编译这些库的软件包，同时要注意采用的版本。下面是笔者编译过程中使用的软件包和对应的版本。

linphone-3.6.1.tar.gz 依赖的库如下：

- libogg-1.3.1.tar.gz;
- speex-1.2rc1.tar.gz（依赖于 libogg）；
- libosip2-4.0.0.tar.gz;
- libExosip2-4.0.0.tar.gz;
- SDL2-2.0.0.tar.gz;
- ffmpeg-2.0.1.tar.gz（依赖 SDL2）。

在编译前读者可以查看 linphone 目录下的 README 文件，其中介绍了版本依赖关系。

下面为 linphone-3.6.1 所依赖的相关软件包的信息：

```
*****Building linphone *****
- you need at least:
  - libosip2>=3.5.0
  - libeXosip2>=3.5.0
  - speex>=1.2.0 (including libspeexdsp part)

+ if you want the gtk/glade interface:
  - libgtk >=2.16.0
+ if you want video support:
  - libvpx (VP8 codec)
  - libavcodec (ffmpeg)
  - libswscale (part of ffmpeg too) for better scaling performance
  - libxv (x11 video extension)
  - libgl1-mesa (OpenGL API -- GLX development files)
  - libglew (OpenGL Extension Wrangler library)
  - libv4l (Video for linux)
  - libx11 (x11)
  - theora (optional)
+ gsm codec (gsm source package or libgsm-dev or gsm-devel) (optional)
+ libreadline (optional: for convenient command line in linphonerc)
+ libsoup (optional: for wizard - account creation assistant)
+ libsqlite3 (optional : for a local history of chat messages)
+ if you want uPnP support (optional):
  - libupnp (version 1.6 branch (not patched with
18-url-upnpstrings.patch))
```

## 20.8.2 X86 平台上编译和安装

编译和安装有一定的依赖关系，下面将依次介绍具体每个软件包的编译和安装过程，同时列出编译和安装过程中遇到的问题和相应的解决方法。

### 1. 建立相关目录

在/usr/local 目录下建立本次编译目录 linphone，然后在该目录下建立存放源码目录 src，建立 X86 的安装目录 linphone\_x86。复制 20.8.1 节中下载的源码并放在 src 目录下。

### 2. 编译libogg

libogg 提供了对多媒体格式文件操作的接口，是编译 speex 所依赖的文件。下面给出其具体编译和安装过程：

```
# tar zxvf libogg-1.3.1.tar.gz
# cd libogg-1.3.1
# ./configure --prefix=/usr/local/linphone/linphone_x86 // 指定其安装目录
# make
# make install
```

### 3. 编译speex

speex 不是为移动电话的语音歌曲格式编码设计的，而是专门为网络包和 VOIP 设计

的。下面给出其编译和安装过程：

```
# tar zxvf speex-1.2rc1.tar.gz
# cd speex-1.2rc1
# ./configure --prefix=/usr/local/linphone/linphone_x86 //指定其安装目录
# make
# make install
```

#### 4. 编译libosip2

libosip2 是 SIP 协议的实现库，它的目的是提供给多媒体和电信软件设计者一个方便和强大的接口，让他们在这个接口基础上开发自己的 SIP 应用。下面是其详细的编译和安装过程：

```
# tar zxvf libosip2-4.0.0.tar.gz
# cd libosip2-4.0.0
# ./configure --prefix=/usr/local/linphone/linphone_x86
# make
# make install
```

#### 5. 编译libeXosip

libeXosip2 是 eXosip 的库文件，eXosip 是协议栈 Osip2 的扩展协议集，它部分封装了协议栈 Osip2，使得它更容易被使用。下面是编译安装 eXosip 库文件的过程：

```
# tar zxvf libeXosip2-4.0.0.tar.gz
# cd libeXosip2-4.0.0
# ./configure --prefix=/usr/local/linphone/linphone_x86 \
PKG_CONFIG_PATH=/usr/local/linphone/linphone_x86/lib/pkgconfig
//pkg-config的有效路径
# make
# make install
```

 注意：如果上一步的 libosip2 不是按照默认路径安装，那么在安装 libeXosip 时要在 configure 后面添加 PKG\_CONFIG\_PATH=安装目录\lib\pkgconfig。如果是默认方式安装，则不需要此参数。

#### 6. 编译SDL

SDL (Simple DirectMedia Layer) 是一个跨平台的多媒体库，是用于直接控制底层多媒体硬件的接口。这些多媒体功能包括了音频、键盘和鼠标（事件）、游戏摇杆等。编译 ffmpeg 时需要依赖该库。下面给出其编译安装过程：

```
# tar zxvf SDL2-2.0.0.tar.gz
# cd SDL2-2.0.0
# ./configure --prefix=/usr/local/linphone/linphone_x86
# make
# make install
```

#### 7. 编译ffmpeg

ffmpeg 用于视频文件转换，支持通过实时抓取电视卡并编码成视频文件。在编译

mediastreamer2 时需要该库的支持。下面是其编译和安装过程：

```
# tar zxvf ffmpeg-2.0.1.tar.gz
# cd ffmpeg-2.0.1
#./configure --prefix=/usr/local/linphone/linphone_x86 --enable-gpl
--enable-shared \
--enable-swscale --enable-pthreads --disable-yasm
```

使用 configure 命令生成 Makefile 后，修改 Makefile 手动添加 X11 库，添加方法如下：

```
FFLIBS-$ (CONFIG_AVDEVICE) += avdevice
FFLIBS-$ (CONFIG_AVFILTER) += avfilter
FFLIBS-$ (CONFIG_AVFORMAT) += avformat
FFLIBS-$ (CONFIG_AVRESAMPLE) += avresample
FFLIBS-$ (CONFIG_AVCODEC) += avcodec
FFLIBS-$ (CONFIG_POSTPROC) += postproc
FFLIBS-$ (CONFIG_SWRESAMPLE) += swresample
FFLIBS-$ (CONFIG_SWSCALE) += swscale
```

在其后添加 X11 库支持，修改后为：

```
FFLIBS-$ (CONFIG_AVDEVICE) += avdevice
FFLIBS-$ (CONFIG_AVFILTER) += avfilter
FFLIBS-$ (CONFIG_AVFORMAT) += avformat
FFLIBS-$ (CONFIG_AVRESAMPLE) += avresample
FFLIBS-$ (CONFIG_AVCODEC) += avcodec
FFLIBS-$ (CONFIG_POSTPROC) += postproc
FFLIBS-$ (CONFIG_SWRESAMPLE) += swresample
FFLIBS-$ (CONFIG_SWSCALE) += swscale
FFLIBS += X11
```

修改完成后保存，使用 make 和 make install 进行安装和编译。

```
# make
# make install
```

## 8. 编译oRTP

oRTP 是实时传输协议栈，其代码就包含在 linphone 目录下。下面给出其编译和安装过程：

```
# tar zxvf linphone-3.6.1.tar.gz
# cd linphone-3.6.1
# cd oRTP
# ./configure --prefix=/usr/local/linphone/linphone_x86
# make
# make install
```

## 9. 编译mediastreamer2

mediastreamer2 是一个功能强大、轻量级流媒体引擎，专门为语音/视频电话应用而设计，其源码也在 linphone 目录下。下面给出其编译和安装过程：

```
# cd ../mediastreamer2
# ./configure --prefix=/usr/local/linphone/linphone_x86 \
PKG_CONFIG_PATH=/usr/local/linphone/linphone_x86/lib/pkgconfig
# make
# make install
```

## 10. 编译Linphone

使用 Linphone 可以免费在互联网上与其他人进行通信，通信方式包括语音、视频、直接文本消息。下面给出其编译和安装过程：

```
#cd ../
# ./configure --prefix=/usr/local/linphone/linphone_x86 \
PKG_CONFIG_PATH=/usr/local/linphone/linphone_x86/lib/pkgconfig \
SPEEX_CFLAGS=/usr/local/linphone/linphone_x86/include/speex \
--enable-gtk_ui=no
```

执行 configure 命令，有个错误：error: The intltool scripts were not found. Please install intltool。

需要安装工具 intltool。使用如下命令进行安装：

```
#yum install intltool
```

安装完 Intltool 后，重新执行 configure 命令，然后执行 make 和 make install 进行编译和安装。

```
# make
# make install
```

### 20.8.3 Linphone 测试

测试的方法是通过虚拟机 Linux 和主机 Windows 进行 VoIP 测试。前面在 Linux 下面已经编译安装了 Linphone 的工具，在安装目录 /usr/local/linphone/linphone\_x86/bin 下。该目录下的工具包括：

```
ffmpeg ffserver linphonecsh sipomatic speexdec
ffplay linphonec sdl-config sip_reg speexenc
```

#### 1. Linux中运行linphonec

Linux 下运行 linphonec，使用命令为：

```
# ./linphonec -V //参数-V 表示支持视频
```

Linux 下启动 linphonec 后，运行情况如图 20.13 所示。

#### 2. Windows也安装VoIP工具

下载 linphone-3.6.1-setup.exe，在 Windows 下安装，安装完成后运行该程序，运行的结果如图 20.14 所示。可以通过 Options|Prefernces 对各种参数进行设置，默认情况下不需要设置。使用该工具时，下面“当前地址”处显示的是本地的 SIP 地址，上面“SIP 地址或电话号码”输入对方的 SIP 地址或者电话号码，这里输入的是虚拟机的 SIP 地址：sip:tom@192.168.1.111，然后鼠标单击绿色拨号按钮，就可以进行拨号了。

### 3. 在 Linux 中进行接听

在虚拟机（Linux 系统）中会听到振铃音，同时在 Windows 中听到回铃音，Linux 中接听 VoIP 呼叫使用的命令为 answer。接通过程显示的信息为：

```
linphonec> Receiving new incoming call from <sip:toto@192.168.1.104>, assigned id 2
                                                //听到振铃音
linphonec> answer
Connected.
linphonec> linphonec>
```



图 20.13 运行 linphonec

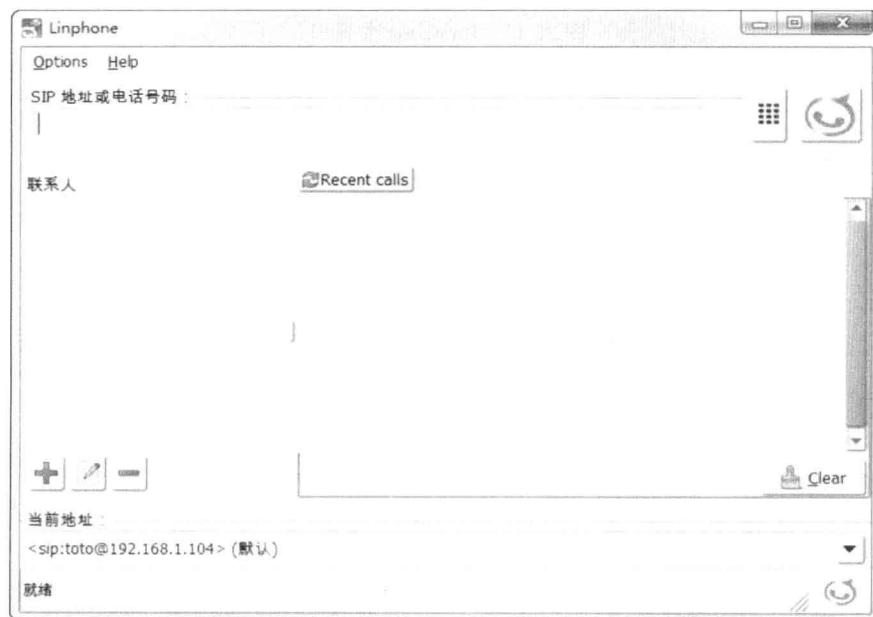


图 20.14 Windows 下运行的 Linphone

#### 4. 双方进入了通话状态

Windows 中的 Linphone 工具标识了对方的 SIP 地址和通话时长，如图 20.15 所示。视频窗口指出视频接收的对端的地址，如图 20.16 所示。

另外，也可以通过 Linux 下的 linphonec 发起呼叫。在 Linux 下呼叫时，因为在同一台计算机中进行测试，如果直接使用 call sip:toto@192.168.1.104 无法正常呼叫时，可以带上端口号，带上 Windows 下 Linphone 工具指定的端口号。如果是默认的端口号 5060，则不需要指定。收到呼叫如图 20.17 所示。

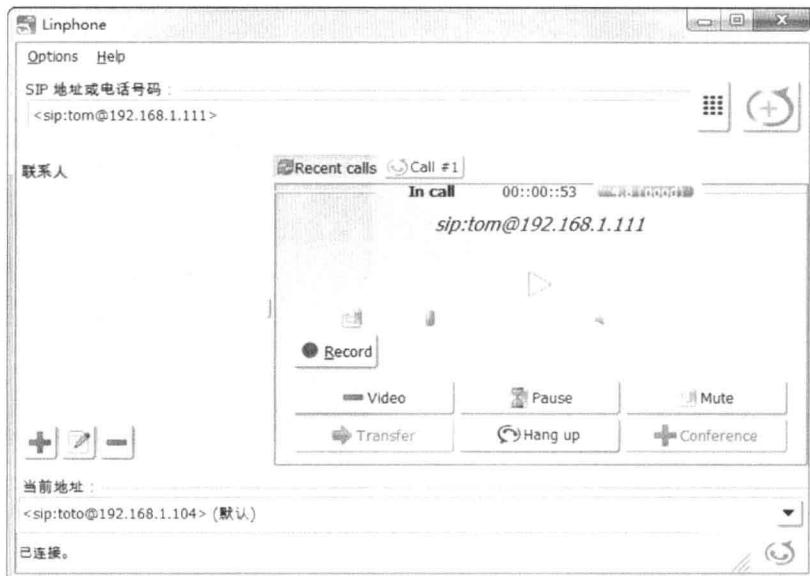


图 20.15 Linphone 主窗口

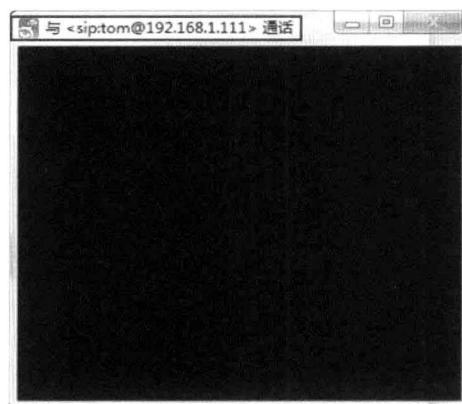


图 20.16 Linphone 视频窗口

```
# call sip:toto@192.168.1.104          //Windows 使用默认的端口号 5060 时
# call sip:toto@192.168.1.104:5066    //Windows 使用非默认的端口号 5066 时
```



图 20.17 Windows 收到 Linux 的 linphonec 呼叫

进入 linphonec 命令状态后，可以使用 help 查看 linphonec 支持的所有命令。退出通话可以使用 quit 命令或者直接使用 Ctrl+C 组合键退出。下面的命令读者可以一一进行测试。

```
linphonec> help
Commands are:
-----
    help      Print commands help.
    call      Call a SIP uri or number
    calls     Show all the current calls with their id and status.
    chat      Chat with a SIP uri
    terminate Terminate a call
    answer    Answer a call
    pause     pause a call
    resume    resume a call
    transfer   Transfer a call to a specified destination.
conference Create and manage an audio conference.
    mute     Mute microphone and suspend voice transmission.
    camera   Send camera output for current call.
    unmute   Unmute microphone and resume voice transmission.
playbackga Adjust playback gain.
    duration  Print duration in seconds of the last call.
autoanswer Show/set auto-answer mode
    proxy    Manage proxies
soundcard  Manage soundcards
    webcam  Manage webcams
    ipv6    Use IPV6
    nat     Set nat address
    stun    Set stun server address
firewall   Set firewall policy
call-logs   Calls history
    friend  Manage friends
    play    play a wav file
    record  record to a wav file
    quit    Exit linphonec
-----
Type 'help <command>' for more details or
  'help advanced' to list additional commands.
```

#### 20.8.4 进一步的测试和开发

由于笔者目前测试的场景有限，读者可以继续完成下面的测试场景：

- 局域网内两台不同的主机进行音频和视频通话；
- 局域网内多台不同的主机进行音频和视频通话；

- 互联网内两台不同主机进行音频和视频通话。
- 笔者编译的时候没有选择 GUI 的支持，读者可以进一步编译出支持 GUI 的 linphonec 工具。

## 20.9 Linphone 交叉编译

在针对 X86 平台编译时，遇到的问题包括代码中存在的语法问题，及相关版本依赖问题，编译时选择的参数问题全部解决。而且在针对 X86 平台编译的时候，也没有采用默认的方式编译，目的是为了方便移植，那么针对 ARM 的交叉编译就非常容易了，只需要修改一条编译命令即可。必须注意的是，因为在针对 ARM 编译前，已经做过针对 X86 的编译，所以在编译每个一部分时应该先使用 make clean 进行清除，否则就会因为在编译针对 ARM 工具时使用了 X86 格式的中间文件，而导致编译出错。

### 20.9.1 Linphone 的交叉编译

不同版本的源代码使用 configure 生成 Makefile 所带的参数有所不同，下面给出的是使用交叉编译器 arm-linux-4.4.3 编译 Linphone-3.6.1 的过程。如果读者使用的版本与本例使用的版本不同，那么就应该使用./configure –help 查看配置 Makefile 所需要带的参数。

查看 linphone 目录下的文件 README.arm:

对编译器设置的介绍

```
* You need the lastest arm toolchain from http://www.handhelds.org.  
Uncompress it in / .
```

```
It contains all the cross-compilation tools. Be sure that the  
arm-linux-gcc binaries  
are in your PATH (export PATH=$PATH:/usr/local/arm/3.4.1/bin/ , for  
example)
```

对源码版本要求的介绍

```
* create within your home directory a arm/ directory, copy into it the fresh  
tarballs of libosip2>=2.2.x, speex>=1.1.6, linphone>=1.2.1  
readline>=5.1 and ncurses>=5.5 (readline needs ncurses)  
Uncompress all these  
tarballs.
```

对安装路径的变量的介绍

```
Very important things common to all packages being cross compiled:  
*****
```

```
* copy the ipaq-config.site in the ipkg/ directory of linphone into some safe  
place,  
for example: ~/ipaq-config.site .  
* You need a directory that we call ARM_INSTALL_TREE that will own files in  
the same way they will be installed on the target computer.  
It is also used to build linphone over the arm binaries of its dependencies  
(speex,osip,ncurses,readline).
```

For example:

```
export CONFIG_SITE=~/ipaq-config.site  
export ARM_INSTALL_TREE=/armbuild
```

下面是安装的详细过程

```
Cross compiling ncurses for ARM:
*****
./configure --prefix=/usr --host=arm-linux --with-gnu-ld --with-shared
make
make install DESTDIR=$ARM_INSTALL_TREE
make install DESTDIR=`pwd`/armbuild

Cross compiling readline for ARM:
*****
./configure --prefix=/usr --host=arm-linux --with-gnu-ld --disable-static
make
make install DESTDIR=$ARM_INSTALL_TREE
make install DESTDIR=`pwd`/armbuild

Cross compiling libosip for ARM:
*****
./configure --prefix=/usr --host=arm-linux --with-gnu-ld --disable-static
make
make install DESTDIR=$ARM_INSTALL_TREE
make install DESTDIR=`pwd`/armbuild

Cross compiling speex for ARM:
*****
First you need to remove ogg headers from your build system to avoid a dirty
conflict between
your build machine binaries and the arm binaries. They are usually in a
libogg-dev package (rpm or deb).
Then:
./configure --prefix=/usr --host=arm-linux --with-gnu-ld --disable-static
--enable-fixed-point --enable-arm-asn
make
make install DESTDIR=$ARM_INSTALL_TREE
make install DESTDIR='pwd'/armbuild

Cross compiling linphone for ARM
*****
First you need to remove all .la files from the $ARM_INSTALL_TREE because
it confuses libtool and makes
the linker use your build machine binaries instead of the arm-crosscompiled
ones.
rm -f $ARM_INSTALL_TREE/usr/lib/*.la
#for some reason pkg-config doesn't like cross-compiling...
export PKG_CONFIG=/usr/bin/pkg-config
./configure --prefix=/usr --host=arm-linux --with-gnu-ld --disable-static
 \
    --disable-glib --with-osip=$ARM_INSTALL_TREE/usr \
    --with-readline=$ARM_INSTALL_TREE/usr \
    SPEEX_CFLAGS="-I$ARM_INSTALL_TREE/usr/include" \
    SPEEX_LIBS="-L$ARM_INSTALL_TREE/usr/lib -lspeex"
make
make install DESTDIR='pwd'/armbuild
```

 注意：上面的编译过程笔者都进行了验证，笔者的安装路径为 /usr/local/linphone/linphone\_arm，在编译 linphone 之前补充编译下面几项。下面列出的编译配置参数都是非常基本和重要的，如果读者要对 linphone 的其他功能进行安装，则需要读者自己进一步研究。

(1) 编译 ffmpeg。

```
# cd ffmpeg-2.0.1
# ./configure --prefix=/usr/local/linphone/linphone_arm --enable-cross-
compile \
--enable-swscale --enable-pthreads --enable-shared --disable-static
# make clean
# make
# make intall
```

(2) 编译 oRTP。

编译过程中会遇到将警告作为错误处理，则需要修改对应的 Makefile 中的编译参数，修改如下：

```
# vi src/Makefile
CFLAGS = -g -O2 -Wall -Werror -DORTP_INET6
修改
CFLAGS = -g -O2 -Wall -DORTP_INET6 # -Werror, 该参数是将警告作为错误来处理
# ./configure --prefix=/usr/local/linphone/linphone_arm --host=arm-linux
--with-gnu-ld --disable-static
# make
# make install
```

(3) 编译 mediastreamer2。

```
# ./configure --prefix=/usr/local/linphone/linphone_arm --host=arm-linux
--with-gnu-ld \
PKG_CONFIG_PATH=/usr/local/linphone/linphone_arm/lib/pkgconfig
# make
# make install
```

(4) 编译 linphone。

```
# ./configure --prefix=/usr/local/linphone/linphone_arm --host=arm-linux
--with-gnu-ld \
--disable-static --disable-glib --with-osip=/usr/local/linphone/
linphone_arm \
--with-readline=/usr/local/linphone/linphone_arm \
SPEEX_CFLAGS="-I/usr/local/linphone/linphone_arm/include" \
SPEEX_LIBS="-L/usr/local/linphone/linphone_arm/lib -lspeex" \
--enable-video=no
PKG_CONFIG_PATH=/usr/local/linphone/linphone_arm/lib/pkgconfig
--enable-gtk_ui=no
# make
# make install
```

## 20.9.2 Linphone 的测试

在目录 /usr/local/linphone/linphone\_arm/bin/ 下会生成下面的工具。

```
# ls /usr/local/linphone/linphone_arm/bin/
captoinfo ffmpeg ffserver infotocap linphonecsh sdl-config sip_reg
speexenc tic tput
clear ffplay infocomp linphonec reset sipomatic speexdec
tack toe tset
```

移植前首先查看其依赖的库文件，可以通过编译在 X86 目录下的 linphone 进行查看，

对应的移植 arm 目录下的 linphone 也需要相应的库文件。

```
# ldd linphonerc
    linux-gate.so.1 => (0x00898000)
    liblinphone.so.3 => /usr/local/linphone/linphone_x86/lib/
    liblinphone.so.3 (0x0034e000)
    libreadline.so.5 => /usr/lib/libreadline.so.5 (0x443c3000)
    libncurses.so.5 => /usr/lib/libncurses.so.5 (0x443f9000)
    libmediastreamer.so.0 => /usr/local/linphone/linphone_x86/lib/
    libmediastreamer.so.0 (0x0053e000)
    libortp.so.8 => /usr/local/linphone/linphone_x86/lib/libortp.so.8
    (0x00218000)
    libspeex.so.1 => /usr/local/linphone/linphone_x86/lib/
    libspeex.so.1 (0x0019c000)
    libm.so.6 => /lib/libm.so.6 (0x446c7000)
    libosipparser2.so.4 => /usr/local/linphone/linphone_x86/lib/
    libosipparser2.so.4 (0x003e9000)
    libosip2.so.4 => /usr/local/linphone/linphone_x86/lib/
    libosip2.so.4 (0x00120000)
    libpthread.so.0 => /lib/libpthread.so.0 (0x446f6000)
    libc.so.6 => /lib/libc.so.6 (0x44588000)
    libeXosip2.so.4 => /usr/local/linphone/linphone_x86/lib/
    libeXosip2.so.4 (0x0091e000)
    libdl.so.2 => /lib/libdl.so.2 (0x446f0000)
    libasound.so.2 => /lib/libasound.so.2 (0x44024000)
    libspeexdsp.so.1 => /usr/local/linphone/linphone_x86/lib/
    libspeexdsp.so.1 (0x00fe0000)
    libavcodec.so.52 => /usr/local/linphone/linphone_x86/lib/
    libavcodec.so.52 (0x00ff1000)
    libswscale.so.0 => /usr/local/linphone/linphone_x86/lib/
    libswscale.so.0 (0x00132000)
    libSDL-1.2.so.0 => /usr/local/linphone/linphone_x86/lib/
    libSDL-1.2.so.0 (0x00232000)
    libX11.so.6 => /usr/lib/libX11.so.6 (0x4481d000)
    librt.so.1 => /lib/librt.so.1 (0x44724000)
    libvorbisenc.so.2 => /usr/lib/libvorbisenc.so.2 (0x43c14000)
    libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x45143000)
    libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x45135000)
    libssl.so.6 => /lib/libssl.so.6 (0x43f02000)
    libcrypto.so.6 => /lib/libcrypto.so.6 (0x43d57000)
    /lib/ld-linux.so.2 (0x43bb9000)
    libnsl.so.1 => /lib/libnsl.so.1 (0x00165000)
    libresolv.so.2 => /lib/libresolv.so.2 (0x0017b000)
    libavutil.so.49 => /usr/local/linphone/linphone_x86/lib/
    libavutil.so.49 (0x00f8a000)
    libz.so.1 => /usr/lib/libz.so.1 (0x4470f000)
    libXau.so.6 => /usr/lib/libXau.so.6 (0x44818000)
    libXdmcp.so.6 => /usr/lib/libXdmcp.so.6 (0x447cf000)
    libvorbis.so.0 => /usr/lib/libvorbis.so.0 (0x43bde000)
    libgssapi_krb5.so.2 => /usr/lib/libgssapi_krb5.so.2 (0x001b2000)
    libkrb5.so.3 => /usr/lib/libkrb5.so.3 (0x002c1000)
    libcom_err.so.2 => /lib/libcom_err.so.2 (0x00110000)
    libk5crypto.so.3 => /usr/lib/libk5crypto.so.3 (0x001dd000)
    libogg.so.0 => /usr/local/linphone/linphone_x86/lib/libogg.so.0
    (0x009d2000)
    libkrb5support.so.0 => /usr/lib/libkrb5support.so.0 (0x00113000)
```

移植这些库文件将是一个漫长的过程，但是已经没有什么难题了。缺少的库通过交叉编译后移植到开发板的/lib 目录下或/usr/lib 下，最后的运行结果将和在虚拟机上运行的结果一样。

## 20.10 小 结

本章主要讲解了 VoIP 技术基本概念，讲解其中两个重要的协议 SIP 和 RTP，并且对 SIP 协议和 RTP 协议进行源码分析；还通过实例介绍了 X86 和 ARM 平台的编译过程，演示了 X86 平台上的使用方法。其在 ARM 上的使用过程留给读者自己去试验。相信本章对读者的项目有很大的帮助，如果还需要扩展一些功能或者需要其他库文件的支持，需要读者进一步研究。