Linux音频编程指南

级别： 初级

肖文鹏 (xiaowp@263.net), 自由软件爱好者

2004 年 2 月 01 日

虽然目前Linux的优势主要体现在网络服务方面，但事实上同样也有着非常丰富的媒体功能，本文就是以多媒体应用中最基本的声音为对象，介绍如何在Linux平台下开发实际的音频应用程序，同时还给出了一些常用的音频编程框架。

一、数字音频

音频信号是一种连续变化的模拟信号，但计算机只能处理和记录二进制的数字信号，由自然音源得到的音频信号必须经过一定的变换，成为数字音频信号之后，才能送到计算机中作进一步的处理。

数字音频系统通过将声波的波型转换成一系列二进制数据，来实现对原始声音的重现，实现这一步骤的设备常被称为模/数转换器（A/D）。A/D转换器以每秒钟上万次的速率对声波进行采样，每个采样点都记录下了原始模拟声波在某一时刻的状态，通常称之为样本（sample），而每一秒钟所采样的数目则称为采样频率，通过将一串连续的样本连接起来，就可以在计算机中描述一段声音了。对于采样过程中的每一个样本来说，数字音频系统会分配一定存储位来记录声波的振幅，一般称之为采样分辩率或者采样精度，采样精度越高，声音还原时就会越细腻。

数字音频涉及到的概念非常多，对于在Linux下进行音频编程的程序员来说，最重要的是理解声音数字化的两个关键步骤：采样和量化。采样就是每隔一定时间就读一次声音信号的幅度，而量化则是将采样得到的声音信号幅度转换为数字值，从本质上讲，采样是时间上的数字化，而量化则是幅度上的数字化。下面介绍几个在进行音频编程时经常需要用到的技术指标：

1. **采样频率**
   采样频率是指将模拟声音波形进行数字化时，每秒钟抽取声波幅度样本的次数。采样频率的选择应该遵循奈奎斯特（Harry Nyquist）采样理论：如果对某一模拟信号进行采样，则采样后可还原的最高信号频率只有采样频率的一半，或者说只要采样频率高于输入信号最高频率的两倍，就能从采样信号系列重构原始信号。正常人听觉的频率范围大约在20Hz~20kHz之间，根据奈奎斯特采样理论，为了保证声音不失真，采样频率应该在40kHz左右。常用的音频采样频率有8kHz、11.025kHz、22.05kHz、16kHz、37.8kHz、44.1kHz、48kHz等，如果采用更高的采样频率，还可以达到DVD的音质。
2. **量化位数**
   量化位数是对模拟音频信号的幅度进行数字化，它决定了模拟信号数字化以后的动态范围，常用的有8位、12位和16位。量化位越高，信号的动态范围越大，数字化后的音频信号就越可能接近原始信号，但所需要的存贮空间也越大。
3. **声道数**
   声道数是反映音频数字化质量的另一个重要因素，它有单声道和双声道之分。双声道又称为立体声，在硬件中有两条线路，音质和音色都要优于单声道，但数字化后占据的存储空间的大小要比单声道多一倍。

二、声卡驱动

出于对安全性方面的考虑，Linux下的应用程序无法直接对声卡这类硬件设备进行操作，而是必须通过内核提供的驱动程序才能完成。在Linux上进行音频编程的本质就是要借助于驱动程序，来完成对声卡的各种操作。

对硬件的控制涉及到寄存器中各个比特位的操作，通常这是与设备直接相关并且对时序的要求非常严格，如果这些工作都交由应用程序员来负责，那么对声卡的编程将变得异常复杂而困难起来，驱动程序的作用正是要屏蔽硬件的这些底层细节，从而简化应用程序的编写。目前Linux下常用的声卡驱动程序主要有两种：OSS和ALSA。

最早出现在Linux上的音频编程接口是OSS（Open Sound System），它由一套完整的内核驱动程序模块组成，可以为绝大多数声卡提供统一的编程接口。OSS出现的历史相对较长，这些内核模块中的一部分（OSS/Free）是与Linux内核源码共同免费发布的，另外一些则以二进制的形式由4Front Technologies公司提供。由于得到了商业公司的鼎力支持，OSS已经成为在Linux下进行音频编程的事实标准，支持OSS的应用程序能够在绝大多数声卡上工作良好。

虽然OSS已经非常成熟，但它毕竟是一个没有完全开放源代码的商业产品，ALSA（Advanced Linux Sound Architecture）恰好弥补了这一空白，它是在Linux下进行音频编程时另一个可供选择的声卡驱动程序。ALSA除了像OSS那样提供了一组内核驱动程序模块之外，还专门为简化应用程序的编写提供了相应的函数库，与OSS提供的基于ioctl的原始编程接口相比，ALSA函数库使用起来要更加方便一些。ALSA的主要特点有：

- 支持多种声卡设备
- 模块化的内核驱动程序
- 支持SMP和多线程
- 提供应用开发函数库

- 兼容OSS应用程序

ALSA和OSS最大的不同之处在于ALSA是由志愿者维护的自由项目，而OSS则是由公司提供的商业产品，因此在对硬件的适应程度上OSS要优于ALSA，它能够支持的声卡种类更多。ALSA虽然不及OSS运用得广泛，但却具有更加友好的编程接口，并且完全兼容于OSS，对应用程序员来讲无疑是一个更佳的选择。

三、编程接口

如何对各种音频设备进行操作是在Linux上进行音频编程的关键，通过内核提供的一组系统调用，应用程序能够访问声卡驱动程序提供的各种音频设备接口，这是在Linux下进行音频编程最简单也是最直接的方法。

3.1 访问音频设备

无论是OSS还是ALSA，都是以内核驱动程序的形式运行在Linux内核空间中的，应用程序要想访问声卡这一硬件设备，必须借助于Linux内核所提供的系统调用（system call）。从程序员的角度来说，对声卡的操作在很大程度上等同于对磁盘文件的操作：首先使用open系统调用建立起与硬件间的联系，此时返回的文件描述符将作为随后操作的标识；接着使用read系统调用从设备接收数据，或者使用write系统调用向设备写入数据，而其它所有不符合读/写这一基本模式的操作都可以由ioctl系统调用来完成；最后，使用close系统调用告诉Linux内核不会再对该设备做进一步的处理。

- **open系统调用**
  系统调用open可以获得对声卡的访问权，同时还能为随后的系统调用做好准备，其函数原型如下所示：
  ```
  int open(const char *pathname, int flags, int mode);
  ```
  参数pathname是将要被打开的设备文件的名称，对于声卡来讲一般是/dev/dsp。参数flags用来指明应该以什么方式打开设备文件，它可以是O_RDONLY、O_WRONLY或者O_RDWR，分别表示以只读、只写或者读写的方式打开设备文件；参数mode通常是可选的，它只有在指定的设备文件不存在时才会用到，指明新创建的文件应该具有怎样的权限。
  如果open系统调用能够成功完成，它将返回一个正整数作为文件标识符，在随后的系统调用中需要用到该标识符。如果open系统调用失败，它将返回-1，同时还会设置全局变量errno，指明是什么原因导致了错误的发生。
- **read系统调用**
  系统调用read用来从声卡读取数据，其函数原型如下所示：
  ```
  int read(int fd, char *buf, size_t count);
  ```
  参数fd是设备文件的标识符，它是通过之前的open系统调用获得的；参数buf是指向缓冲区的字符指针，它用来保存从声卡获得的数据；参数count则用来限定从声卡获得的最大字节数。如果read系统调用成功完成，它将返回从声卡实际读取的字节数，通常情况会比count的值要小一些；如果read系统调用失败，它将返回-1，同时还会设置全局变量errno，来指明是什么原因导致了错误的发生。
- **write系统调用**
  系统调用write用来向声卡写入数据，其函数原型如下所示：
  ```
  size_t write(int fd, const char *buf, size_t count);
  ```
  系统调用write和系统调用read在很大程度是类似的，差别只在于write是向声卡写入数据，而read则是从声卡读入数据。参数fd同样是设备文件的标识符，它也是通过之前的open系统调用获得的；参数buf是指向缓冲区的字符指针，它保存着即将向声卡写入的数据；参数count则用来限定向声卡写入的最大字节数。
  如果write系统调用成功完成，它将返回向声卡实际写入的字节数；如果read系统调用失败，它将返回-1，同时还会设置全局变量errno，来指明是什么原因导致了错误的发生。无论是read还是write，一旦调用之后Linux内核就会阻塞当前应用程序，直到数据成功地从声卡读出或者写入为止。
- **ioctl系统调用**
  系统调用ioctl可以对声卡进行控制，凡是对设备文件的操作不符合读/写基本模式的，都是通过ioctl来完成的，它可以影响设备的行为，或者返回设备的状态，其函数原型如下所示：
  ```
  int ioctl(int fd, int request, ...);
  ```
  参数fd是设备文件的标识符，它是在设备打开时获得的；如果设备比较复杂，那么对它的控制请求相应地也会有很多种，参数request的目的就是用来区分不同的控制请求；通常说来，在对设备进行控制时还需要有其它参数，这要根据不同的控制请求才能确定，并且可能是与硬件设备直接相关的。
- **close系统调用**
  当应用程序使用完声卡之后，需要用close系统调用将其关闭，以便及时释放占用的硬件资源，其函数原型如下所示：
  ```
  int close(int fd);
  ```
  参数fd是设备文件的标识符，它是在设备打开时获得的。一旦应用程序调用了close系统调用，Linux内核就会释放与之相关的各种资源，因此建议在不需要的时候尽量及时关闭已经打开的设备。

3.2 音频设备文件

对于Linux应用程序员来讲，音频编程接口实际上就是一组音频设备文件，通过它们可以从声卡读取数据，或者向声卡写入数据，并且能够对声卡进行控制，设置采样频率和声道数目等等。

- **/dev/sndstat**
  设备文件/dev/sndstat是声卡驱动程序提供的最简单的接口，通常它是一个只读文件，作用也仅仅只限于汇报声卡的当前状态。一般说来，/dev/sndstat是提供给最终用户来检测声卡的，不宜用于程序当中，因为所有的信息都可以通过ioctl系统调用来获得。 Linux提供的cat命令可以很方便地从/dev/sndstat获得声卡的当前状态：
  [xiaowp@linuxgam sound]$ cat /dev/sndstat
- **/dev/dsp**
  声卡驱动程序提供的/dev/dsp是用于数字采样（sampling）和数字录音（recording）的设备文件，它对于Linux下的音频编程来讲非常重要：向该设备写数据即意味着激活声卡上的D/A转换器进行放音，而向该设备读数据则意味着激活声卡上的A/D转换器进行录音。目前许多声卡都提供有多个数字采样设备，它们在Linux下可以通过/dev/dsp1等设备文件进行访问。

  DSP是数字信号处理器（Digital Signal Processor）的简称，它是用来进行数字信号处理的特殊芯片，声卡使用它来实现模拟信号和数字信号的转换。声卡中的DSP设备实际上包含两个组成部分：在以只读方式打开时，能够使用A/D转换器进行声音的输入；而在以只写方式打开时，则能够使用D/A转换器进行声音的输出。严格说来，Linux下的应用程序要么以只读方式打开/dev/dsp输入声音，要么以只写方式打开/dev/dsp输出声音，但事实上某些声卡驱动程序仍允许以读写的方式打开/dev/dsp，以便同时进行声音的输入和输出，这对于某些应用场合（如IP电话）来讲是非常关键的。

  在从DSP设备读取数据时，从声卡输入的模拟信号经过A/D转换器变成数字采样后的样本（sample），保存在声卡驱动程序的内核缓冲区中，当应用程序通过read系统调用从声卡读取数据时，保存在内核缓冲区中的数字采样结果将被复制到应用程序所指定的用户缓冲区中。需要指出的是，声卡采样频率是由内核中的驱动程序所决定的，而不取决于应用程序从声卡读取数据的速度。如果应用程序读取数据的速度过慢，以致低于声卡的采样频率，那么多余的数据将会被丢弃；如果读取数据的速度过快，以致高于声卡的采样频率，那么声卡驱动程序将会阻塞那些请求数据的应用程序，直到新的数据到来为止。

  在向DSP设备写入数据时，数字信号会经过D/A转换器变成模拟信号，然后产生出声音。应用程序写入数据的速度同样应该与声卡的采样频率相匹配，否则过慢的话会产生声音暂停或者停顿的现象，过快的话又会被内核中的声卡驱动程序阻塞，直到硬件有能力处理新的数据为止。与其它设备有所不同，声卡通常不会支持非阻塞（non-blocking）的I/O操作。

  无论是从声卡读取数据，或是向声卡写入数据，事实上都具有特定的格式（format），默认为8位无符号数据、单声道、8KHz采样率，如果默认值无法达到要求，可以通过ioctl系统调用来改变它们。通常说来，在应用程序中打开设备文件/dev/dsp之后，接下去就应该为其设置恰当的格式，然后才能从声卡读取或者写入数据。

- **/dev/audio**
  /dev/audio类似于/dev/dsp，它兼容于Sun工作站上的音频设备，使用的是mu-law编码方式。如果声卡驱动程序提供了对/dev/audio的支持，那么在Linux上就可以通过cat命令，来播放在Sun工作站上用mu-law进行编码的音频文件：

  ```
  [xiaowp@linuxgam sound]$ cat audio.au > /dev/audio
  ```

  由于设备文件/dev/audio主要出于对兼容性的考虑，所以在新开发的应用程序中最好不要尝试用它，而应该以/dev/dsp进行替代。对于应用程序来说，同一时刻只能使用/dev/audio或者/dev/dsp其中之一，因为它们是相同硬件的不同软件接口。
- **/dev/mixer**
  在声卡的硬件电路中，混音器（mixer）是一个很重要的组成部分，它的作用是将多个信号组合或者叠加在一起，对于不同的声卡来说，其混音器的作用可能各不相同。运行在Linux内核中的声卡驱动程序一般都会提供/dev/mixer这一设备文件，它是应用程序对混音器进行操作的软件接口。混音器电路通常由两个部分组成：输入混音器（input mixer）和输出混音器（output mixer）。
  输入混音器负责从多个不同的信号源接收模拟信号，这些信号源有时也被称为混音通道或者混音设备。模拟信号通过增益控制器和由软件控制的音量调节器后，在不同的混音通道中进行级别（level）调制，然后被送到输入混音器中进行声音的合成。混音器上的电子开关可以控制哪些通道中有信号与混音器相连，有些声卡只允许连接一个混音通道作为录音的音源，而有些声卡则允许对混音通道做任意的连接。经过输入混音器处理后的信号仍然为模拟信号，它们将被送到A/D转换器进行数字化处理。
  输出混音器的工作原理与输入混音器类似，同样也有多个信号源与混音器相连，并且事先都经过了增益调节。当输出混音器对所有的模拟信号进行了混合之后，通常还会有一个总控增益调节器来控制输出声音的大小，此外还有一些音调控制器来调节输出声音的音调。经过输出混音器处理后的信号也是模拟信号，它们最终会被送给喇叭或者其它的模拟输出设备。对混音器的编程包括如何设置增益控制器的级别，以及怎样在不同的音源间进行切换，这些操作通常来讲是不连续的，而且不会像录音或者放音那样需要占用大量的计算机资源。由于混音器的操作不符合典型的读/写操作模式，因此除了open和close两个系统调用之外，大部分的操作都是通过ioctl系统调用来完成的。与/dev/dsp不同，/dev/mixer允许多个应用程序同时访问，并且混音器的设置值会一直保持到对应的设备文件被关闭为止。
  为了简化应用程序的设计，Linux上的声卡驱动程序大多都支持将混音器的ioctl操作直接应用到声音设备上，也就是说如果已经打开了/dev/dsp，那么就不用再打开/dev/mixer来对混音器进行操作，而是可以直接用打开/dev/dsp时得到的文件标识符来设置混音器。
- **/dev/sequencer**
  目前大多数声卡驱动程序还会提供/dev/sequencer这一设备文件，用来对声卡内建的波表合成器进行操作，或者对MIDI总线上的乐器进行控制，一般只用于计算机音乐软件中。

四、应用框架

在Linux下进行音频编程时，重点在于如何正确地操作声卡驱动程序所提供的各种设备文件，由于涉及到的概念和因素比较多，所以遵循一个通用的框架无疑将有助于简化应用程序的设计。

4.1 DSP编程

对声卡进行编程时首先要做的是打开与之对应的硬件设备，这是借助于open系统调用来完成的，并且一般情况下使用的是/dev/dsp文件。采用何种模式对声卡进行操作也必须在打开设备时指定，对于不支持全双工的声卡来说，应该使用只读或者只写的方式打开，只有那些支持全双工的声卡，才能以读写的方式打开，并且还要依赖于驱动程序的具体实现。Linux允许应用程序多次打开或者关闭与声卡对应的设备文件，从而能够很方便地在放音状态和录音状态之间进行切换，建议在进行音频编程时只要有可能就尽量使用只读或者只写的方式打开设备文件，因为这样不仅能够充分利用声卡的硬件资源，而且还有利于驱动程序的优化。下面的代码示范了如何以只写方式打开声卡进行放音（playback）操作：

```
int handle = open("/dev/dsp", O_WRONLY);
if (handle == -1) {
          perror("open /dev/dsp");
          return -1;
}
```

运行在Linux内核中的声卡驱动程序专门维护了一个缓冲区，其大小会影响到放音和录音时的效果，使用ioctl系统调用可以对它的尺寸进行恰当的设置。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，一般采用默认的缓冲区大小也就可以了。但需要注意的是，缓冲区大小的设置通常应紧跟在设备文件打开之后，这是因为对声卡的其它操作有可能会导致驱动程序无法再修改其缓冲区的大小。下面的代码示范了怎样设置声卡驱动程序中的内核缓冲区的大小：

```
int setting = 0xnnnnssss;
int result = ioctl(handle, SNDCTL_DSP_SETFRAGMENT, &setting);
if (result == -1) {
          perror("ioctl buffer size");
          return -1;
}
// 检查设置值的正确性
```

在设置缓冲区大小时，参数setting实际上由两部分组成，其低16位标明缓冲区的尺寸，相应的计算公式为buffer_size = 2^ssss，即若参数setting低16位的值为16，那么相应的缓冲区的大小会被设置为65536字节。参数setting的高16位则用来标明分片（fragment）的最大序号，它的取值范围从2一直到0x7FFF，其中0x7FFF表示没有任何限制。

接下来要做的是设置声卡工作时的声道（channel）数目，根据硬件设备和驱动程序的具体情况，可以将其设置为0（单声道，mono）或者1（立体声，stereo）。下面的代码示范了应该怎样设置声道数目：

```
int channels = 0; // 0=mono 1=stereo
int result = ioctl(handle, SNDCTL_DSP_STEREO, &channels);
if ( result == -1 ) {
          perror("ioctl channel number");
          return -1;
}
if (channels != 0) {
          // 只支持立体声
}
```

采样格式和采样频率是在进行音频编程时需要考虑的另一个问题，声卡支持的所有采样格式可以在头文件soundcard.h中找到，而通过ioctl系统调用则可以很方便地更改当前所使用的采样格式。下面的代码示范了如何设置声卡的采样格式：

```
int format = AFMT_U8;
int result = ioctl(handle, SNDCTL_DSP_SETFMT, &format);
if ( result == -1 ) {
          perror("ioctl sample format");
          return -1;
}
// 检查设置值的正确性
```

声卡采样频率的设置也非常容易，只需在调用ioctl时将第二个参数的值设置为SNDCTL_DSP_SPEED，同时在第三个参数中指定采样频率的数值就行了。对于大多数声卡来说，其支持的采样频率范围一般为5kHz到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。下面的代码示范了如何设置声卡的采样频率：

```
int rate = 22050;
int result = ioctl(handle, SNDCTL_DSP_SPEED, &rate);
if ( result == -1 ) {
          perror("ioctl sample format");
          return -1;
}
// 检查设置值的正确性
```

4.2 Mixer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ioctl系统调用来完成的，并且所有控制命令都由SOUND_MIXER或者MIXER开头，表1列出了常用的几个混音器控制命令：

名 称	作 用
SOUND_MIXER_VOLUME	主音量调节
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 转换器
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

表1 混音器命令

对声卡的输入增益和输出增益进行调节是混音器的一个主要作用，目前大部分声卡采用的是8位或者16位的增益控制器，但作为程序员来讲并不需要关心这些，因为声卡驱动程序会负责将它们变换成百分比的形式，也就是说无论是输入增益还是输出增益，其取值范围都是从0到100。在进行混音器编程时，可以使用SOUND_MIXER_READ宏来读取混音通道的增益大小，例如在获取麦克风的输入增益时，可以使用如下的代码：

```
int vol;
ioctl(fd, SOUND_MIXER_READ(SOUND_MIXER_MIC), &vol);
printf("Mic gain is at %d %%\n", vol);
```

对于只有一个混音通道的单声道设备来说，返回的增益大小保存在低位字节中。而对于支持多个混音通道的双声道设备来说，返回的增益大小实际上包括两个部分，分别代表左、右两个声道的值，其中低位字节保存左声道的音量，而高位字节则保存右声道的音量。下面的代码可以从返回值中依次提取左右声道的增益大小：

```
int left, right;
left = vol & 0xff;
right = (vol & 0xff00) >> 8;
printf("Left gain is %d %%, Right gain is %d %%\n", left, right);
```

类似地，如果想设置混音通道的增益大小，则可以通过SOUND_MIXER_WRITE宏来实现，此时遵循的原则与获取增益值时的原则基本相同，例如下面的语句可以用来设置麦克风的输入增益：

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

在编写实用的音频程序时，混音器是在涉及到兼容性时需要重点考虑的一个对象，这是因为不同的声卡所提供的混音器资源是有所区别的。声卡驱动程序提供了多个ioctl系统调用来获得混音器的信息，它们通常返回一个整型的位掩码（bitmask），其中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。例如通过SOUND_MIXER_READ_DEVMASK返回的位掩码，可以查询出能够被声卡支持的每一个混音通道，而通过SOUND_MIXER_READ_RECMAS返回的位掩码，则可以查询出能够被当作录音源的每一个通道。下面的代码可以用来检查CD输入是否是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");
```

如果进一步还想知道其是否是一个有效的录音源，则可以使用如下语句：

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

目前大多数声卡提供多个录音源，通过SOUND_MIXER_READ_RECSRC可以查询出当前正在使用的录音源，同一时刻能够使用几个录音源是由声卡硬件决定的。类似地，使用SOUND_MIXER_WRITE_RECSRC可以设置声卡当前使用的录音源，例如下面的代码可以将CD输入作为声卡的录音源使用：

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

此外，所有的混音通道都有单声道和双声道的区别，如果需要知道哪些混音通道提供了对立体声的支持，可以通过SOUND_MIXER_READ_STEREODEVS来获得。

4.3 音频录放框架

下面给出一个利用声卡上的DSP设备进行声音录制和回放的基本框架，它的功能是先录制几秒种音频数据，将其存放在内存缓冲区中，然后再进行回放，其所有的功能都是通过读写/dev/dsp设备文件来完成的：

```
/*
 * sound.c
 */
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include <stdio.h>
#include <linux/soundcard.h>
#define LENGTH 3    /* 存储秒数 */
#define RATE 8000   /* 采样频率 */
#define SIZE 8      /* 量化位数 */
#define CHANNELS 1  /* 声道数目 */
/* 用于保存数字音频数据的内存缓冲区 */
unsigned char buf[LENGTH*RATE*SIZE*CHANNELS/8];
int main()
{
  int fd; /* 声音设备的文件描述符 */
  int arg;          /* 用于ioctl调用的参数 */
  int status;   /* 系统调用的返回值 */
  /* 打开声音设备 */
  fd = open("/dev/dsp", O_RDWR);
  if (fd < 0) {
    perror("open of /dev/dsp failed");
    exit(1);
  }
  /* 设置采样时的量化位数 */
  arg = SIZE;
  status = ioctl(fd, SOUND_PCM_WRITE_BITS, &arg);
  if (status == -1)
    perror("SOUND_PCM_WRITE_BITS ioctl failed");
  if (arg != SIZE)
    perror("unable to set sample size");
  /* 设置采样时的声道数目 */
```

```
  arg = CHANNELS;
  status = ioctl(fd, SOUND_PCM_WRITE_CHANNELS, &arg);
  if (status == -1)
    perror("SOUND_PCM_WRITE_CHANNELS ioctl failed");
  if (arg != CHANNELS)
    perror("unable to set number of channels");
  /* 设置采样时的采样频率 */
  arg = RATE;
  status = ioctl(fd, SOUND_PCM_WRITE_RATE, &arg);
  if (status == -1)
    perror("SOUND_PCM_WRITE_WRITE ioctl failed");
  /* 循环，直到按下Control-C */
  while (1) {
    printf("Say something:\n");
    status = read(fd, buf, sizeof(buf)); /* 录音 */
    if (status != sizeof(buf))
      perror("read wrong number of bytes");
    printf("You said:\n");
    status = write(fd, buf, sizeof(buf)); /* 回放 */
    if (status != sizeof(buf))
      perror("wrote wrong number of bytes");
    /* 在继续录音前等待回放结束 */
    status = ioctl(fd, SOUND_PCM_SYNC, 0);
    if (status == -1)
      perror("SOUND_PCM_SYNC ioctl failed");
  }
}
```

4.4 混音器框架

下面再给出一个对混音器进行编程的基本框架，利用它可以对各种混音通道的增益进行调节，其所有的功能都是通过读写/dev/mixer设备文件来完成的：

```
/*
 * mixer.c
 */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <linux/soundcard.h>
/* 用来存储所有可用混音设备的名称 */
const char *sound_device_names[] = SOUND_DEVICE_NAMES;
int fd;                  /* 混音设备所对应的文件描述符 */
int devmask, stereodevs; /* 混音器信息对应的位图掩码 */
char *name;
/* 显示命令的使用方法及所有可用的混音设备 */
void usage()
{
  int i;
  fprintf(stderr, "usage: %s <device> <left-gain%%> <right-gain%%>\n"
            "       %s <device> <gain%%>\n\n"
            "Where <device> is one of:\n", name, name);
  for (i = 0 ; i < SOUND_MIXER_NRDEVICES ; i++)
    if ((1 << i) & devmask) /* 只显示有效的混音设备 */
      fprintf(stderr, "%s ", sound_device_names[i]);
  fprintf(stderr, "\n");
  exit(1);
}
int main(int argc, char *argv[])
{
  int left, right, level;  /* 增益设置 */
  int status;              /* 系统调用的返回值 */
  int device;              /* 选用的混音设备 */
  char *dev;               /* 混音设备的名称 */
  int i;
  name = argv[0];
```

```
  /* 以只读方式打开混音设备 */
  fd = open("/dev/mixer", O_RDONLY);
  if (fd == -1) {
    perror("unable to open /dev/mixer");
    exit(1);
  }

  /* 获得所需要的信息 */
  status = ioctl(fd, SOUND_MIXER_READ_DEVMASK, &devmask);
  if (status == -1)
    perror("SOUND_MIXER_READ_DEVMASK ioctl failed");
  status = ioctl(fd, SOUND_MIXER_READ_STEREODEVS, &stereodevs);
  if (status == -1)
    perror("SOUND_MIXER_READ_STEREODEVS ioctl failed");
  /* 检查用户输入 */
  if (argc != 3 && argc != 4)
    usage();
  /* 保存用户输入的混音器名称 */
  dev = argv[1];
  /* 确定即将用到的混音设备 */
  for (i = 0 ; i < SOUND_MIXER_NRDEVICES ; i++)
    if (((1 << i) & devmask) && !strcmp(dev, sound_device_names[i]))
      break;
  if (i == SOUND_MIXER_NRDEVICES) { /* 没有找到匹配项 */
    fprintf(stderr, "%s is not a valid mixer device\n", dev);
    usage();
  }
  /* 查找到有效的混音设备 */
  device = i;
  /* 获取增益值 */
  if (argc == 4) {
    /* 左、右声道均给定 */
    left  = atoi(argv[2]);
    right = atoi(argv[3]);
  } else {
    /* 左、右声道设为相等 */
    left  = atoi(argv[2]);
    right = atoi(argv[2]);
  }

  /* 对非立体声设备给出警告信息 */
  if ((left != right) && !((1 << i) & stereodevs)) {
    fprintf(stderr, "warning: %s is not a stereo device\n", dev);
  }

  /* 将两个声道的值合到同一变量中 */
  level = (right << 8) + left;

  /* 设置增益 */
  status = ioctl(fd, MIXER_WRITE(device), &level);
  if (status == -1) {
    perror("MIXER_WRITE ioctl failed");
    exit(1);
  }
  /* 获得从驱动返回的左右声道的增益 */
  left  = level & 0xff;
  right = (level & 0xff00) >> 8;
  /* 显示实际设置的增益 */
  fprintf(stderr, "%s gain set to %d%% / %d%%\n", dev, left, right);
  /* 关闭混音设备 */
  close(fd);
  return 0;
}
```

编译好上面的程序之后，先不带任何参数执行一遍，此时会列出声卡上所有可用的混音通道：

```
[xiaowp@linuxgam sound]$ ./mixer
usage: ./mixer <device> <left-gain%> <right-gain%>
```

```
       ./mixer <device> <gain%>

Where <device> is one of:
vol pcm speaker line mic cd igain line1 phin video
```

之后就可以很方便地设置各个混音通道的增益大小了，例如下面的命令就能够将CD输入的左、右声道的增益分别设置为80%和90%：

```
[xiaowp@linuxgam sound]$ ./mixer cd 80 90
cd gain set to 80% / 90%
```

五、小结

随着Linux平台下多媒体应用的逐渐深入，需要用到数字音频的场合必将越来越广泛。虽然数字音频牵涉到的概念非常多，但在Linux下进行最基本的音频编程却并不十分复杂，关键是掌握如何与OSS或者ALSA这类声卡驱动程序进行交互，以及如何充分利用它们提供的各种功能，熟悉一些最基本的音频编程框架和模式对初学者来讲大有裨益。

参考资料

- 1. OSS是Linux上最早出现的声卡驱动程序，http://www.opensound.com是它的核心网站，从中可以了解到许多与OSS相关的信息。
- 2. ALSA是目前广泛使用的Linux声卡驱动程序，并且提供了一些库函数来简化音频程序的编写，在其官方网站http://www.alsa-project.org/上可以了解到ALSA的许多信息，并能够下载到最新的驱动程序和工具软件。
- 3. Ken C. Pohlmann著，苏菲译，数字音频原理与应用（第四合版），北京：电子工业出版社，2002
- 4. 钟玉琢等编著，多媒体技术及其应用，北京：机械工业出版社，2003

关于作者

本文作者肖文鹏是一名自由软件爱好者，主要从事操作系统和分布式计算环境的研究，喜爱Linux和Python。你可以通过 xiaowp@263.net与他取得联系。