第 3 章

CHAPTER 3

FFmpeg 二次开发采集并预览本地摄像头

FFmpeg 中有一个和多媒体设备交互的类库，即 Libavdevice，使用这个库可以读取计算机（或者其他设备）的多媒体设备中的数据，或者将数据输出到指定的多媒体设备上，也可以使用 Qt 或 SDL 将捕获到的摄像头数据显示到窗口中。

5min

3.1 FFmpeg 的命令行方式处理摄像头

使用 ffmpeg -hide_banner -devices 命令可以查看本机支持的输入/输出设备，以 Windows 和 Linux 为例。显示的信息中 D 表示支持解码，可以作为输入；E 表示支持编码，可以作为输出，具体信息如下：

```
//chapter3/help - others.txt
Devices:
 D = Demuxing supported
 E = Muxing supported
 ---
 D dshow            DirectShow capture
 D lavfi            Libavfilter virtual input device
 E sdl,sdl2         SDL2 output device
 D vfwcap           VfW video capture
Devices:
 D. = Demuxing supported
 E = Muxing supported
 --
 DE alsa            ALSA audio output
 E caca             caca (color ASCII art) output device
 DE fbdev           Linux framebuffer
 D iec61883         libiec61883 (new DV1394) A/V input device
 D jack             JACK Audio Connection Kit
 D kmsgrab          KMS screen capture
 D lavfi            Libavfilter virtual input device
 D libcdio
```

```
 D libdc1394          dc1394 v.2 A/V grab
 D openal             OpenAL audio capture device
 E opengl             OpenGL output
 DE oss               OSS (Open Sound System) playback
 DE pulse             Pulse audio output
 E sdl,sdl2           SDL2 output device
 DE sndio             sndio audio playback
 DE video4linux2,v4l2 Video4Linux2 output device
 E vout_rpi           Rpi (mmal) video output device
 D x11grab            X11 screen capture, using XCB
  E xv                XV (XVideo) output device
```

Windows 平台下使用 vfwcap 的效果比使用 dshow 的效果要差一些，可以通过 ffmpeg -h demuxer=dshow 命令查看支持的操作参数，支持查看设备列表、选项列表，以及可以设置设备输出的视频分辨率、帧率等，具体的输出信息如下：

```
//chapter3/help - others.txt
Demuxer dshow [DirectShow capture]:
dshow indev AVOptions:
  - video_size    < image_size >    . D..... set video size given a string such as 640x480
or hd720.
  - pixel_format < pix_fmt >   . D..... set video pixel format (default none)
  - framerate     < string >   . D..... set video frame rate
  - sample_rate   < int >   . D..... set audio sample rate (from 0 to INT_MAX) (default 0)
  - sample_size   < int >   . D..... set audio sample size (from 0 to 16) (default 0)
  - channels      < int >   . D..... set number of audio channels, such as 1 or 2 (from 0 to INT_
MAX) (default 0)
  - audio_buffer_size < int >   . D..... set audio device buffer latency size in milliseconds
(default is the device's default) (from 0 to INT_MAX) (default 0)
  - list_devices  < boolean >   . D..... list available devices (default false)
  - list_options   < boolean >   . D..... list available options for specified device (default
false)
  - video_device_number   < int >   . D..... set video device number for devices with same name
(starts at 0) (from 0 to INT_MAX) (default 0)
  - audio_device_number < int >   . D..... set audio device number for devices with same name
(starts at 0) (from 0 to INT_MAX) (default 0)
  - crossbar_video_input_pin_number < int >   . D..... set video input pin number for crossbar
device (from - 1 to INT_MAX) (default - 1)
  - crossbar_audio_input_pin_number < int >   . D..... set audio input pin number for crossbar
device (from - 1 to INT_MAX) (default - 1)
  - show_video_device_dialog < boolean >   . D..... display property dialog for video capture
device (default false)
  - show_audio_device_dialog < boolean >   . D..... display property dialog for audio capture
device (default false)
  - show_video_crossbar_connection_dialog < boolean >   . D..... display property dialog for
crossbar connecting pins filter on video device (default false)
  - show_audio_crossbar_connection_dialog < boolean >   . D..... display property dialog for
crossbar connecting pins filter on audio device (default false)
```

```
  -show_analog_tv_tuner_dialog <boolean>   .D..... display property dialog for analog tuner
filter (default false)
  -show_analog_tv_tuner_audio_dialog <boolean>   .D..... display property dialog for analog
tuner audio filter (default false)
  -audio_device_load <string>   .D..... load audio capture filter device (and properties)
from file
  -audio_device_save <string>   .D..... save audio capture filter device (and properties) to
file
  -video_device_load <string>   .D..... load video capture filter device (and properties)
from file
  -video_device_save <string>   .D..... save video capture filter device (and properties) to
file
```

Windows 平台下查询支持的所有设备列表，命令如下：

```
ffmpeg -list_devices true -f dshow -i dummy
```

命令执行后，笔者本地的输出结果如图 3-1 所示（注：有可能出现中文乱码的情况）。列表所显示的设备名称很重要，因为输入时需要使用-f dshow -i video="设备名"的方式。

图 3-1　Windows 平台列举设备列表

获取摄像头数据后，可以保存为本地文件或者将实时流发送到流媒体服务器。例如从摄像头读取数据并编码为 H.264，最后保存成 mycamera.flv 的命令如下：

```
ffmpeg -f dshow -i video="Lenovo EasyCamera" -vcodec libx264 mycamera001.flv
```

使用 ffplay.exe 可以直接播放摄像头的数据，命令如下：

```
ffplay -f dshow -i video="Lenovo EasyCamera"
```

如果设备名称正确，则会直接打开本机的摄像头，效果如图 3-2 所示。

可以查看摄像头的流信息，执行效果如图 3-3 所示，具体命令如下：

```
ffmpeg -hide_banner -f dshow -i "video=Lenovo EasyCamera"
```

查询本机 dshow 设备、查看 USB 2.0 摄像头的信息，只能输出 yuyv422 压缩编码数据（可以理解为 rawvideo 编码器，像素格式为 yuyv422），经过解码后获得 yuyv422 编码的图像数据，所以，在这种情况下可以不解码。

Linux 下大多使用 video4linux2/v4l2 设备，通过 ffmpeg -h demuxer=v4l2 命令可以查看相关的操作参数，输出信息如下：

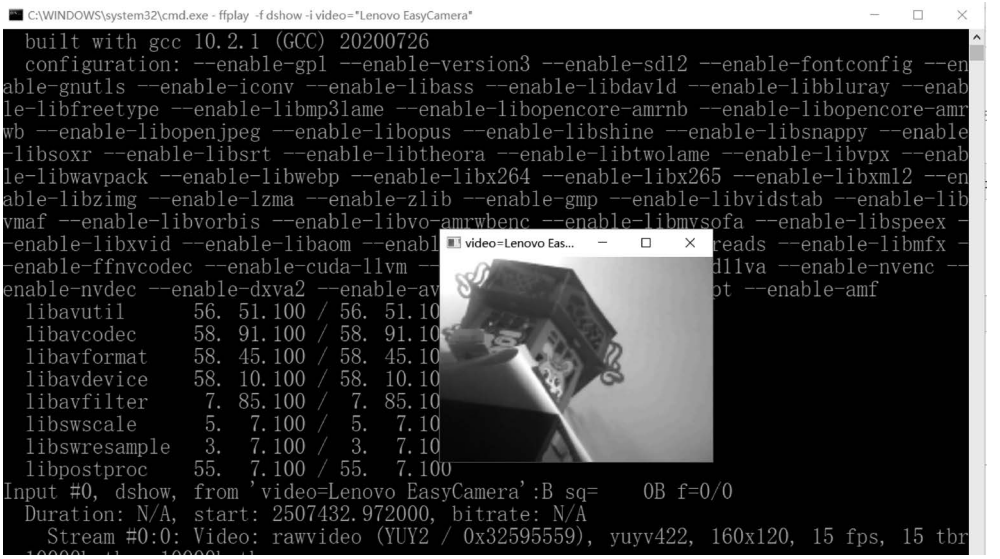

图 3-2 FFplay 播放摄像头

图 3-3 查看摄像头的流信息

```
//chapter3/help - others.txt
Demuxer video4linux2,v4l2 [Video4Linux2 device grab]:
V4L2 indev AVOptions:
   - standard       < string >     .D..... set TV standard, used only by analog frame grabber
   - channel        < int >        .D..... set TV channel, used only by frame grabber (from - 1 to
INT_MAX) (default - 1)
   - video_size     < image_size >.D..... set frame size
   - pixel_format   < string >     .D..... set preferred pixel format
   - input_format   < string >   .D..... set preferred pixel format (for raw video) or codec name
   - framerate      < string >     .D..... set frame rate
   - list_formats   < int >    .D..... list available formats and exit (from 0 to INT_MAX)
(default 0)
     all                          .D..... show all available formats
     raw                          .D..... show only non - compressed formats
     compressed                   .D..... show only compressed formats
   - list_standards < int >         .D..... list supported standards and exit (from 0 to 1)
(default 0)
     all                          .D..... show all supported standards
```

```
  -timestamps     <int>     .D..... set type of timestamps for grabbed frames (from 0 to 2)
(default default)
     default                .D..... use timestamps from the kernel
     abs                    .D..... use absolute timestamps (wall clock)
     mono2abs               .D..... force conversion from monotonic to absolute timestamps
  -ts             <int>     .D..... set type of timestamps for grabbed frames (from 0 to 2)
(default default)
     default       .D..... use timestamps from the kernel
     abs           .D..... use absolute timestamps (wall clock)
     mono2abs      .D..... force conversion from monotonic to absolute timestamps
  -use_libv4l2    <boolean> .D..... use libv4l2 (v4l-utils) conversion functions (default
false)
```

当前机器上挂载了 CSI 接口的相机，可以查看其支持的格式，执行效果如图 3-4 所示，该相机支持多种非压缩编码格式，例如 JFIF JPEG、Motion-JPEG、H.264 等，具体命令如下：

```
ffmpeg -hide_banner -f v4l2 -list_formats all -i /dev/video0
```

```
pi@raspberrypi:~$ ffmpeg -hide_banner -f v4l2 -list_formats all -i /dev/video0
[video4linux2,v4l2 @ 0x9bb1c0] Raw       :     yuv420p :     Planar YUV 4:2:0 : {32-3280, 2}x{32-2464, 2}
[video4linux2,v4l2 @ 0x9bb1c0] Raw       :     yuyv422 :         YUYV 4:2:2 : {32-3280, 2}x{32-2464, 2}
[video4linux2,v4l2 @ 0x9bb1c0] Raw       :       rgb24 :    24-bit RGB 8-8-8 : {32-3280, 2}x{32-2464, 2}
[video4linux2,v4l2 @ 0x9bb1c0] Compressed:        mjpeg :          JFIF JPEG : {32-3280, 2}x{32-2464, 2}
[video4linux2,v4l2 @ 0x9bb1c0] Compressed:         h264 :               H.264 : {32-3280, 2}x{32-2464, 2}
[video4linux2,v4l2 @ 0x9bb1c0] Compressed:        mjpeg :        Motion-JPEG : {32-3280, 2}x{32-2464, 2}
[video4linux2,v4l2 @ 0x9bb1c0] Raw       : Unsupported :         YVYU 4:2:2 : {32-3280, 2}x{32-2464, 2}
[video4linux2,v4l2 @ 0x9bb1c0] Raw       : Unsupported :         VYUY 4:2:2 : {32-3280, 2}x{32-2464, 2}
[video4linux2,v4l2 @ 0x9bb1c0] Raw       :     uyvy422 :         UYVY 4:2:2 : {32-3280, 2}x{32-2464, 2}
[video4linux2,v4l2 @ 0x9bb1c0] Raw       :        nv12 :       Y/CbCr 4:2:0 : {32-3280, 2}x{32-2464, 2}
[video4linux2,v4l2 @ 0x9bb1c0] Raw       :       bgr24 :    24-bit BGR 8-8-8 : {32-3280, 2}x{32-2464, 2}
[video4linux2,v4l2 @ 0x9bb1c0] Raw       :     yuv420p :     Planar YVU 4:2:0 : {32-3280, 2}x{32-2464, 2}
[video4linux2,v4l2 @ 0x9bb1c0] Raw       : Unsupported :       Y/CrCb 4:2:0 : {32-3280, 2}x{32-2464, 2}
[video4linux2,v4l2 @ 0x9bb1c0] Raw       :        bgr0 : 32-bit BGRA/X 8-8-8-8 : {32-3280, 2}x{32-2464, 2}
/dev/video0: Immediate exit requested
```

图 3-4　Linux 查看摄像头支持的格式

Linux 平台下读取摄像头并编码为 H.264 的命令如下：

```
#ffmpeg -f dshow -i video="USB2.0 PC CAMERA" -vcodec libx264 xxxx.h264
ffmpeg -f v4l2 -i video=/dev/video0 -vcodec libx264 xxxx.h264
```

在实时流推送中如果需要提高 libx264 的编码速度，则可以添加-preset:v ultrafast 和-tune:v zerolatency 两个选项。

Windows 平台下使用 gdigrab 设备可以录制桌面屏幕，可以查看 gdigrab 支持的选项，命令如下：

```
ffmpeg -h demuxer=gdigrab
```

该命令的输出信息如下：

```
//chapter3/help - others.txt
Demuxer gdigrab [GDI API Windows frame grabber]:
GDIgrab indev AVOptions:
   - draw_mouse   < int >          .D..... draw the mouse pointer (from 0 to 1) (default 1)
   - show_region < int >   .D..... draw border around capture area (from 0 to 1) (default 0)
   - framerate    < video_rate >  .D..... set video frame rate (default "ntsc")
   - video_size   < image_size >  .D..... set video frame size
   - offset_x     < int >          .D..... capture area x offset (from INT_MIN to INT_MAX)
(default 0)
   - offset_y     < int >          .D..... capture area y offset (from INT_MIN to INT_MAX)
(default 0)
```

录屏并编码为 H.264 的命令如下：

```
ffmpeg -f gdigrab -i desktop -vcodec h264 xxxx.h264
```

录屏并显示鼠标，从屏幕左上角(100,200)的 640×480 区域录屏，帧率为 25 的命令如下：

```
ffmpeg -f gdigrab -draw_mouse -framerate 25 -offset_x 100 -offset_y 200 \
-video_size 640x480 -i desktop out1.mpg
```

Linux 平台下与之类似，使用 x11grab 设备，可以查看支持的选项，命令如下：

```
$ ffmpeg -h demuxer=x11grab
```

该命令的输出信息如下：

```
//chapter3/help - others.txt
Demuxer x11grab [X11 screen capture, using XCB]:
xcbgrab indev AVOptions:
   - x                 < int >   .D..... Initial x coordinate. (from 0 to INT_MAX) (default 0)
   - y                 < int >   .D..... Initial y coordinate. (from 0 to INT_MAX) (default 0)
   - grab_x            < int >   .D..... Initial x coordinate. (from 0 to INT_MAX) (default 0)
   - grab_y            < int >   .D..... Initial y coordinate. (from 0 to INT_MAX) (default 0)
   - video_size        < string >   .D..... A string describing frame size, such as 640x480 or
hd720. (default "vga")
   - framerate         < string >  .D..... (default "ntsc")
   - draw_mouse        < int >   .D..... Draw the mouse pointer. (from 0 to 1) (default 1)
   - follow_mouse      < int >   .D..... Move the grabbing region when the mouse pointer
reaches within specified amount of pixels to the edge of region. (from -1 to INT_MAX) (default 0)
     centered                  .D..... Keep the mouse pointer at the center of grabbing region
when following.
   - show_region       < int >   .D..... Show the grabbing region. (from 0 to 1) (default 0)
   - region_border     < int >   .D..... Set the region border thickness. (from 1 to 128)
(default 3)
```

Linux 平台下录屏并捕获鼠标的命令如下：

```
ffmpeg -f x11grab -draw_mouse -framerate 25 -x 100 -y 200 \
-video_size 640x480 -i :0.0 out3.mpg
```

3.2　FFmpeg 的 SDK 方式读取本地摄像头

使用 FFmpeg 采集并预览本地摄像头的流程如图 3-5 所示。

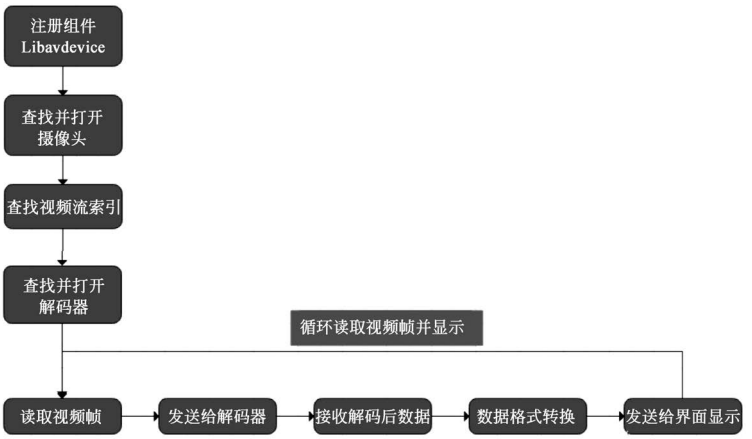

图 3-5　FFmpeg 采集并预览摄像头

使用 Libavdevice 时需要包含头文件，代码如下：

```
#include "libavdevice/avdevice.h"
```

然后在程序中需要注册 Libavdevice，代码如下：

```
avdevice_register_all();
```

接下来就可以使用 Libavdevice 的功能了，使用 Libavdevice 读取数据和直接打开视频文件比较类似。因为系统的设备会被 FFmpeg 当成一种输入的格式（AVInputFormat）。使用 FFmpeg 的 API 可以根据指定参数打开输入流，并返回输入封装的上下文 AVFormatContext，函数的代码如下：

```
//chapter3/code3.2.txt
int avformat_open_input(AVFormatContext **ps,
            const char *url,           //打开的地址,文件或设备等
            AVInputFormat *fmt,        //指定输入格式,若为空,则自动检测
            AVDictionary **options);   //指定解复用器的私有选项
```

通常，打开一个文件或者直播流的代码如下：

```
//chapter3/code3.2.txt
AVFormatContext * input_fmt_ctx = NULL;
const char * file_path = "test.mp4";
//也可以打开网络直播流
//const char * file_path = "rtmp://192.168.1.100:1935/live/test";
avformat_open_input(&input_fmt_ctx, file_path, NULL, NULL);
```

使用 Libavdevice 时，唯一的不同在于首先需要查找用于输入的设备，例如可以使用 av_find_input_format()函数来查找音视频设备。在 Windows 平台上使用 vfw 设备作为输入设备，然后在 URL 中指定打开第 0 个设备(在笔者的计算机上为摄像头设备)，代码如下：

```
//chapter3/code3.2.txt
AVFormatContext * pFormatCtx = avformat_alloc_context();
AVInputFormat * ifmt = av_find_input_format("vfwcap");
avformat_open_input(&pFormatCtx, 0, ifmt,NULL);
```

在 Windows 平台上除了可以使用 vfw 设备作为输入设备之外，还可以使用 DirectShow 作为输入设备，代码如下：

```
//chapter3/code3.2.txt
AVFormatContext * pFormatCtx = avformat_alloc_context();
AVInputFormat * ifmt = av_find_input_format("dshow");
avformat_open_input(&pFormatCtx,"video = Integrated Camera",ifmt,NULL) ;
```

在 Linux 平台上可以使用 v4l2 设备作为输入设备，代码如下：

```
//chapter3/code3.2.txt
//查找设备前要先调用 avdevice_register_all 函数
AVInputFormat * in_fmt = in_fmt = av_find_input_format("video4linux2");
if (in_fmt == NULL) {
   printf("can't find_input_format\n");
   return ;
}

AVFormatContext * fmt_ctx = NULL;
if (avformat_open_input(&fmt_ctx, "/dev/video0", in_fmt, NULL) < 0) {
   printf("can't open_input_file\n");
   return ;
}
```

另外，使用选项 av_dict_set(&options, "f", "v4l2", 0)和指定参数 ifmt 的效果相同。通常在 Linux 平台下可以不设置，默认为支持 v4l2 且自动识别，而在 Windows 平台下的 vfwp、dshow 需要明确指定，代码如下：

```
AVInputFormat * ifmt = av_find_input_format("v4l2");     //加快探测流的速度
avformat_open_input(&fmt_ctx, "/dev/video0", ifmt, NULL);
```

上述代码等效于下面的代码：

```
AVDictionary * options = NULL;
av_dict_set(&options, "f", "v4l2", 0);
avformat_open_input(&fmt_ctx,"/dev/video0",NULL,&options);
```

如果需要指定输入格式，则可以通过 AVOption 设置，并且参数不一样，描述信息如下：

```
-pixel_format <string> .D.... set preferred pixel format
-input_format <string> .D.... set preferred pixel format (for raw video) or codec name
```

当选择像素格式时，一定是非压缩的原始数据，两个参数均可，代码如下：

```
av_dict_set(&options, "pixel_format", "rgb24", 0);
av_dict_set(&options, "input_format", "rgb24", 0);      //同上
```

当选择压缩编码格式，必须只能使用 input_format，代码如下：

```
av_dict_set(&options, "input_format", "h264", 0);
av_dict_set(&options, "input_format", "mjpeg", 0);
```

可以调用 avformat_open_input()函数来打开摄像头设备，通过这个函数的参数指定打开的设备路径“/dev/video0”，使用的驱动“video4linux2”，将相应的格式 pix_fmt 指定为 yuyv422，以及将分辨率指定为 640×480，代码如下：

```
//chapter3/code3.2.txt
AVFormatContext * fmt_ctx = NULL;
AVDictionary * options = NULL;
char * devicename = "/dev/video0";
avdevice_register_all();
AVInputFormat * iformat = av_find_input_format("video4linux2");
av_dict_set(&options,"video_size","640x480",0);
av_dict_set(&options,"pixel_format","yuyv422", 0);
avformat_open_input(&fmt_ctx, devicename, iformat, &options);
avformat_close_input(&fmt_ctx);
```

可以调用 av_read_frame()函数来读取一帧 YUV 数据，代码如下：

```
//chapter3/code3.2.txt
int ret = 0;
AVPacket pkt;
while((ret = av_read_frame(fmt_ctx, &pkt)) == 0) {
   av_log(NULL, AV_LOG_INFO, "packet size is %d(%p)\n",
      pkt.size, pkt.data);
```

```
    av_packet_unref(&pkt);     //释放包
}
```

可以调用 fwrite()函数将读取到的 YUV 数据保存到文件中,代码如下:

```
//chapter3/code3.2.txt
char * out = "out.yuv";
FILE * outfile = fopen(out, "wb+");
fwrite(pkt.data, 1, pkt.size, outfile); //614400
fflush(outfile);
fclose(outfile);
```

打开本地摄像头,循环读取帧数据并存储到文件中的完整代码如下:

```
//chapter3/record-video.c
#include <stdio.h>
#include "libavutil/avutil.h"
#include "libavdevice/avdevice.h"
#include "libavformat/avformat.h"
#include "libavcodec/avcodec.h"


//打开摄像头
static AVFormatContext * open_dev(){
    int ret = 0;
    char errors[1024] = {0, };

    //上下文环境
    AVFormatContext *fmt_ctx = NULL;
    AVDictionary *options = NULL;

    //设备名称
    char *devicename = "/dev/video0";

    //注册 Libavdevice 库
    avdevice_register_all();

    //查找设备
    AVInputFormat *iformat = av_find_input_format("video4linux2");

    av_dict_set(&options,"video_size","640x480",0);       //分辨率
    av_dict_set(&options,"pixel_format","yuyv422", 0);    //YUV 帧格式

    //打开设备
    if((ret = avformat_open_input(&fmt_ctx, devicename, iformat, &options)) < 0 ){
        av_strerror(ret, errors, 1024);
        fprintf(stderr, "Failed to open audio device, [%d]%s\n", ret, errors);
        return NULL;
```

```
    }

    return fmt_ctx;
}

//读取并录制摄像头数据
void rec_video() {
    int ret = 0;
    AVFormatContext *fmt_ctx = NULL;
    int count = 0;

    //packet:数据包
    AVPacket pkt;

    //设置日志级别
    av_log_set_level(AV_LOG_Debug);

    //create file:创建本地文件
    char *out = "out.yuv";
    FILE *outfile = fopen(out, "wb+");

    //打开设备
    fmt_ctx = open_dev();

    //从摄像头中读取 YUV 帧数据
    while((ret = av_read_frame(fmt_ctx, &pkt)) == 0 &&
          count++< 100) {
        av_log(NULL, AV_LOG_INFO,
               "packet size is %d(%p)\n",
               pkt.size, pkt.data);

        fwrite(pkt.data, 1, pkt.size, outfile);        //写入本地文件
        fflush(outfile);
        av_packet_unref(&pkt);                         //release pkt:释放 AVPacket
    }

__ERROR:
    if(outfile){
        //关闭文件
        fclose(outfile);
    }

    //关闭设备并释放上下文环境
    if(fmt_ctx) {
        avformat_close_input(&fmt_ctx);
    }

    av_log(NULL, AV_LOG_Debug, "finish!\n");
    return;
```

```
}


int main(int argc, char * argv[])
{
    rec_video();
    return 0;
}
```

在 Linux 系统中使用编译该文件的命令如下：

```
gcc record_video.c -lavformat -lavutil -lavdevice -lavcodec -o record_video
```

运行该程序会打开摄像头并循环读取帧，然后存储到本地文件 out.yuv 中，命令如下：

```
./record_video
```

使用 FFplay 可以播放生成的 YUV 文件，注意指定分辨率和 YUV 格式，命令如下：

```
ffplay -s 640x480 -pix_fmt yuyv422 out.yuv
```

3.3 FFmpeg+SDL2 读取并显示本地摄像头

使用 FFmpeg 读取摄像头数据之后，可以调用 SDL2 库来渲染。

3.3.1 SDL2 简介

SDL2 使用 GNU 通用公共许可证为授权方式，即动态链接(Dynamic Link)其库并不需要开放本身的源代码。虽然 SDL 时常被比喻为“跨平台的 DirectX”，但事实上 SDL 被定位成以精简的方式来完成基础的功能，大幅度简化了控制图像、声音、输入/输出等工作所需的代码。但更高级的绘图功能或是音效功能则需搭配 OpenGL 和 OpenAL 等 API 来完成。另外，它本身也没有方便创建图形用户界面的函数。SDL2 在结构上是将不同操作系统的库包装成相同的函数，例如 SDL 在 Windows 平台上是 DirectX 的再包装，而在使用 X11 的平台上(包括 Linux)则是调用 Xlib 库来输出图像。虽然 SDL2 本身是使用 C 语言写成的，但是它几乎可以被所有的编程语言所使用，例如 C++、Perl、Python 和 Pascal 等，甚至是 Euphoria、Pliant 这类较不流行的编程语言也都可行。SDL2 库分为 Video、Audio、CD-ROM、Joystick 和 Timer 等若干子系统，除此之外，还有一些单独的官方扩充函数库。这些库由官方网站提供，并包含在官方文档中，它们共同组成了 SDL 的“标准库”。SDL 的整体结构如图 3-6 所示。

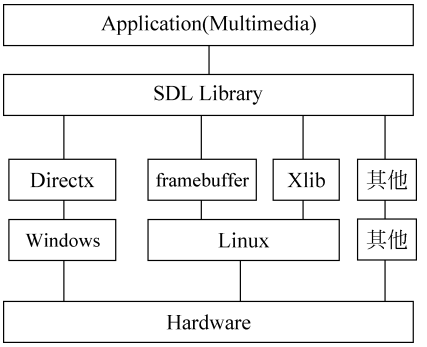

图 3-6　SDL 库的层次结构

3.3.2　VS 2015 搭建 SDL2 开发环境

本节将介绍如何在 VS 2015 下配置 SDL2.0.8 开发库的详细步骤。

1. 下载 SDL2

进入 SDL2 官网，链接网址为 https://github.com/libsdl-org/SDL/releases/。选择 SDL2 的 Development Libraries 中的 SDL2-devel-2.0.12-VC.zip（链接网址为 https://github.com/libsdl-org/SDL/releases/tag/release-2.0.12），如图 3-7 所示。下载并解压以供其他程序调用，在项目配置中可以使用 SDL 库的相对路径。

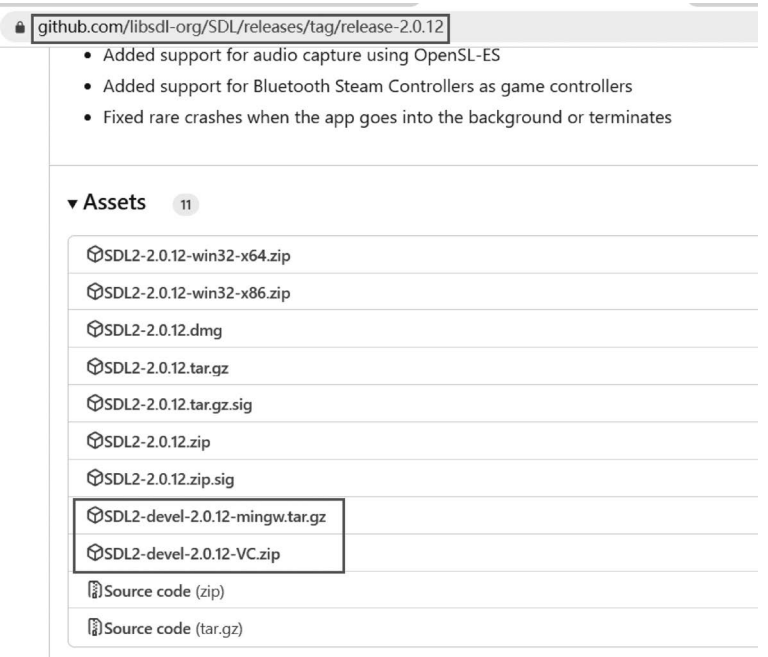

图 3-7　SDL 库的下载网址

2. VS 2015 项目配置

(1) 打开 VS 2015,新建 Win32 控制台项目,将项目命名为 SDLtest1,然后单击右下方的“确定”按钮,如图 3-8 所示。

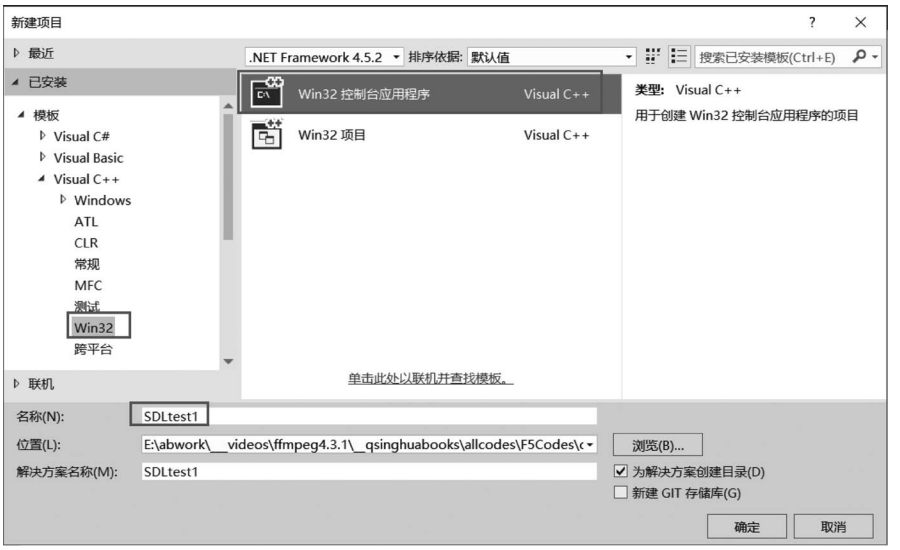

图 3-8　新建 VS 2015 的控制台项目

(2) 右击项目名称(SDLtest1),在弹出的菜单中单击“属性”按钮,然后在弹出的属性页中配置包含目录和库目录,注意笔者这里使用 SDL2 库的相对路径,选择的平台为 Win32,如图 3-9 所示。

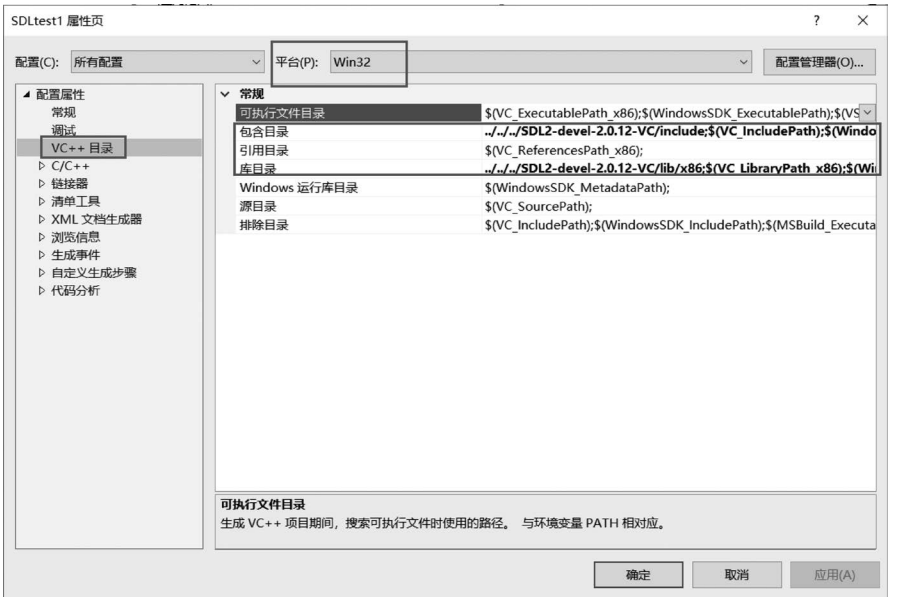

图 3-9　配置 VS 2015 项目的包含目录和库目录

（3）在项目 SDLtest1 属性页中选择“链接器”下的“输入”，编辑右侧的“附加依赖项”，在附加依赖项中添加 SDL2.lib 和 SDL2main.lib（注意中间以英文分号分隔），然后单击右下方的“确定”按钮，如图 3-10 所示。

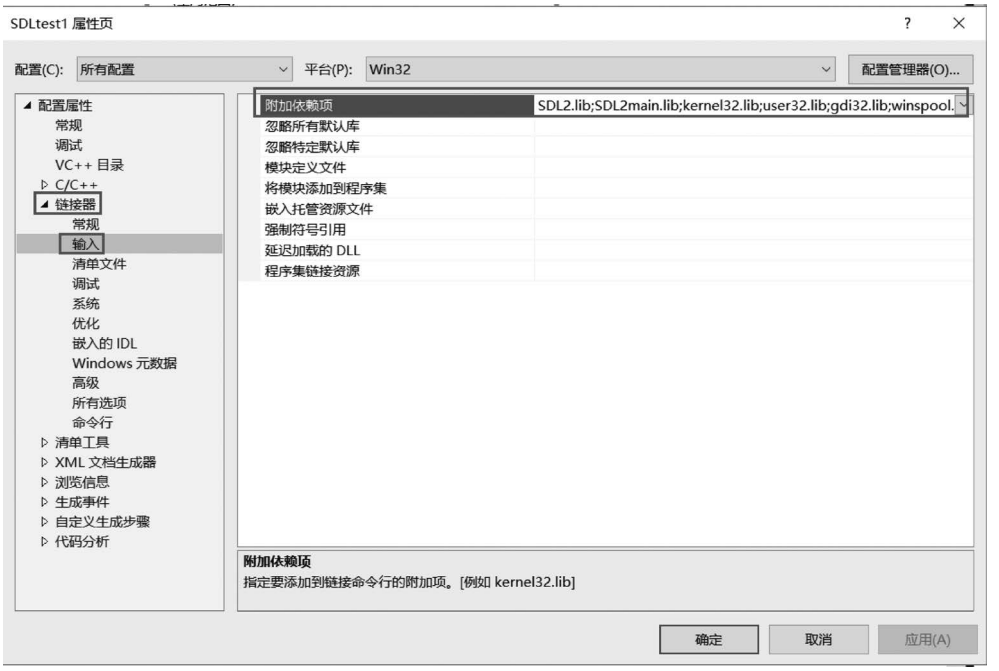

图 3-10　配置 VS 2015 项目的附加依赖项

3. 测试案例

项目配置成功后，可以调用 SDL_Init()函数来测试是否配置成功，代码如下：

```
//chapter3/SDLtest1/SDLtest1.cpp
//SDLtest.cpp : 定义控制台应用程序的入口点
#include "stdafx.h"
#include <iostream>

#define SDL_MAIN_HANDLED      //如果没有此宏,则会报错
#include <SDL.h>

int main(){
   if (SDL_Init(SDL_INIT_VIDEO) != 0)  {
      std::cout << "SDL_Init Error: " << SDL_GetError() << std::endl;
      return 1;
   }
   else{
      std::cout << "SDL_Init OK " << std::endl;
   }
```

```
    SDL_Quit();
    return 0;
}
```

需要注意这个宏语句(#define SDL_MAIN_HANDLED),如果没有定义这个宏,则会报错(并且要放到 SDL.h 之前),错误信息如下:

```
无法解析的外部符号 main,该符号在函数"int cdecl invoke_main(void)" (?invoke_main@@YAHXZ)
中被引用
```

这是因为在 SDL 库的内部重新定义了 main,因此 main()函数需要写成如下形式:

```
int main(int argc,char * argv[])
```

而添加 #define SDL_MAIN_HANDLED 这个宏之后,即使 main()函数的参数列表为空,也不会报错。

编译并运行该程序会提示找不到 SDL2.dll,如图 3-11 所示。将 SDL2-devel-2.0.12-VC\lib\x86 目录下的 SDL2.dll 复制到 SDLtest1.exe 同目录下,如图 3-12 所示。重新编译并运行该程序,若不报错,则表示配置成功,如图 3-13 所示。

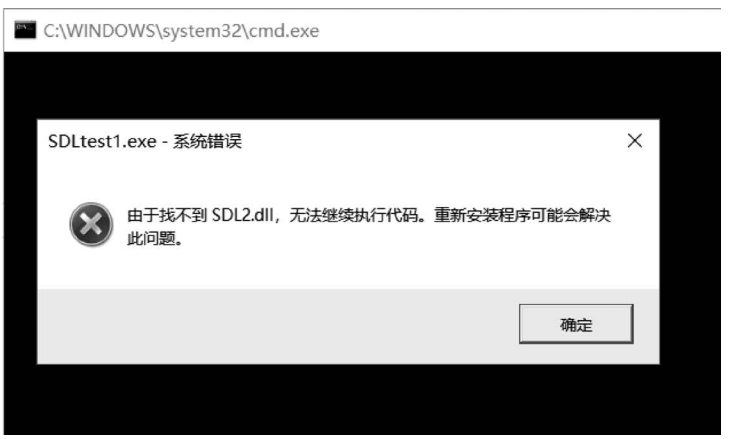

图 3-11　运行时找不到 SDL2.dll 文件

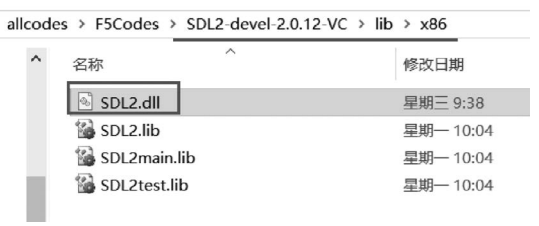

图 3-12　复制 SDL2.dll 文件

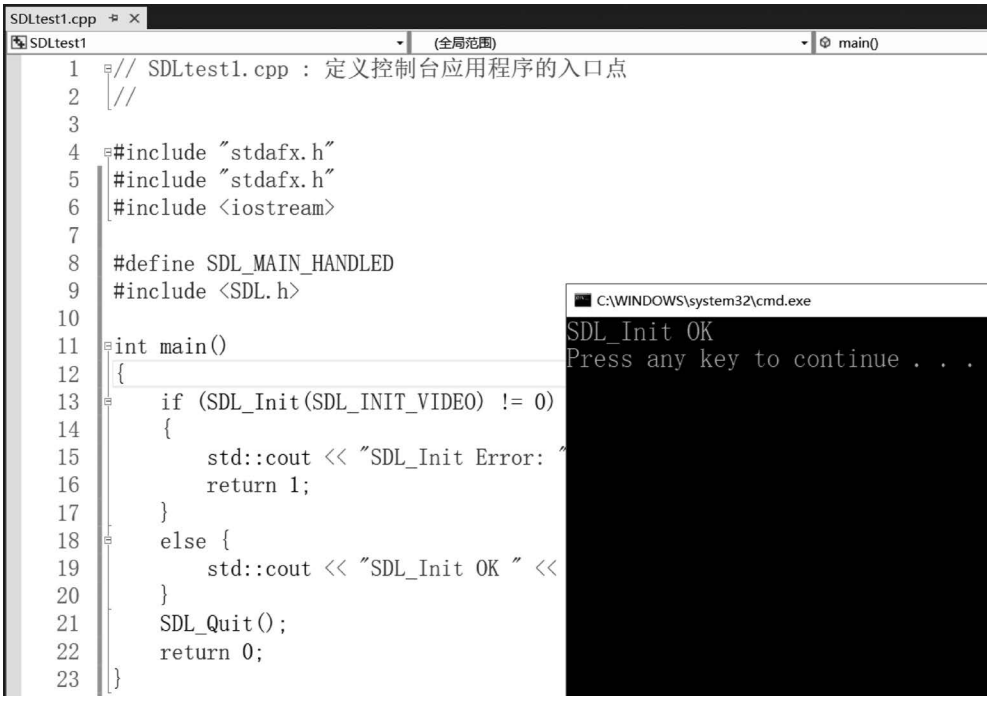

图 3-13　SDL2 库配置成功

3.3.3　Qt 5.9 平台搭建 SDL2 开发环境

笔者本地的 Qt 版本为 5.9.8，配置 SDL2 开发环境的具体步骤如下。

(1) 下载 SDL2 的 mingw 版本，文件名为 SDL2-devel-2.0.12-mingw.tar.gz，链接网址为 https://github.com/libsdl-org/SDL/releases/tag/release-2.0.12。

(2) 打开 Qt Creator，新建 Qt Console Application 类型的项目，单击右下方的 Choose 按钮，如图 3-14 所示。

(3) 在 Project Location 页面输入项目名称(SDLQtDemo1)和路径，如图 3-15 所示。

(4) 在 Kit Selection 页面选中 Desktop Qt 5.9.8 MinGW 32bit，然后单击右下方的“下一步”按钮，如图 3-16 所示。

注意：读者也可以选择其他的编译套件，但不同的编译套件对应着不同的 SDL2 开发包，例如 MinGW 32 位编译套件对应 SDL2-devel-2.0.12-mingw.tar.gz，并且运行时需要对应 32 位的动态链接库。

---

(5) 解压 SDL2-devel-2.0.12-mingw.tar.gz 后有两个重要的子目录，如图 3-17 所示。i686-w64-mingw32 对应的是 32 位的开发库，x86_64-w64-mingw32 对应的是 64 位的开发库。

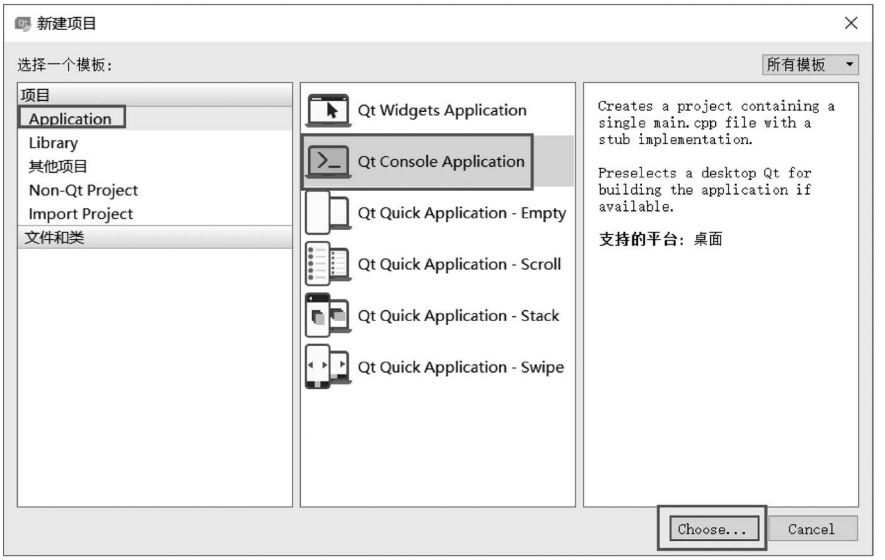

图 3-14　新建 Qt 控制台项目

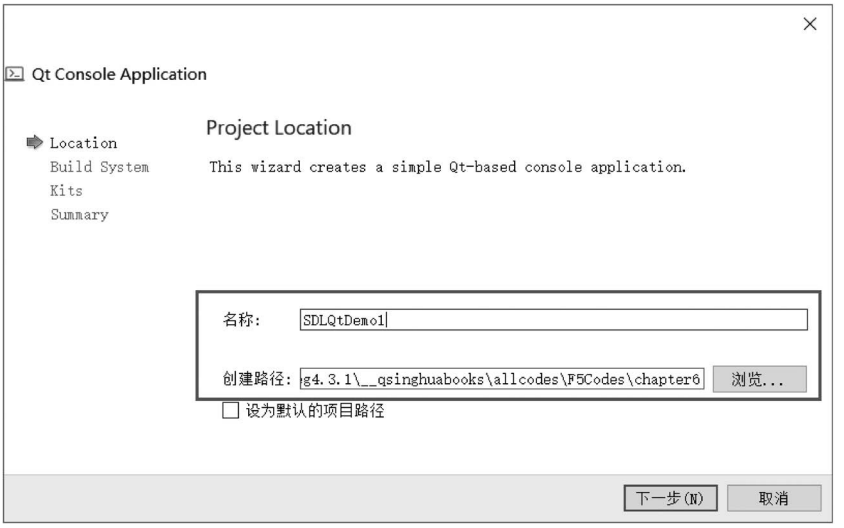

图 3-15　输入 Qt 项目名称和路径

（6）配置 Qt 项目(SDLQtDemo1)，打开 SDLQtDemo1.pro 配置文件，如图 3-18 所示，代码如下：

```
//chapter3/SDLQtDemo1/SDLQtDemo1.pro.txt
INCLUDEPATH += ../../SDL2 - devel - 2.0.12 - mingw/i686 - w64 - mingw32/include/SDL2/
LIBS  +=  - L../../SDL2 - devel - 2.0.12 - mingw/i686 - w64 - mingw32/lib/  - lSDL2  - lSDL2main
```

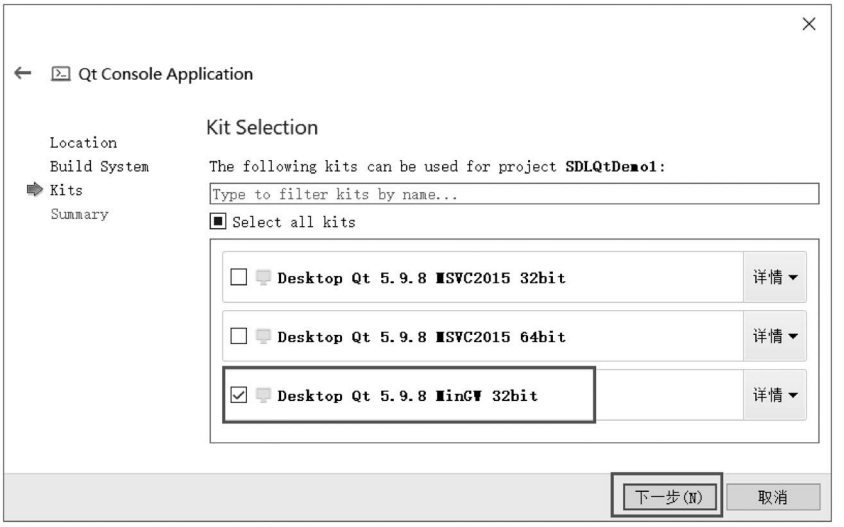

图 3-16　选择 MinGW 32 位编译套件

图 3-17　解压 SDL2-devel-2.0.12-mingw.tar.gz

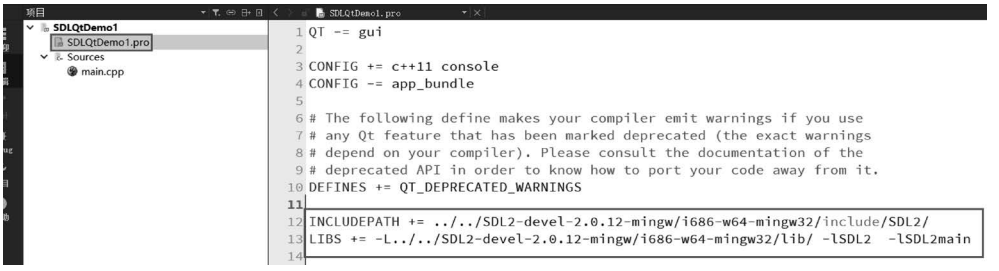

图 3-18　修改 Qt 的项目配置文件

需要注意的是这里使用的是相对路径，如图 3-19 所示。

图 3-19　SDL2 的相对路径

（7）修改 main.cpp，注释掉原来的源码，新增代码如下：

```
//chapter3/SDLQtDemo1/main.cpp
#include <iostream>
#define SDL_MAIN_HANDLED //如果没有此宏,则会报错
#include <SDL.h>

int main(){
    if (SDL_Init(SDL_INIT_VIDEO) != 0){
        std::cout << "SDL_Init Error: " << SDL_GetError() << std::endl;
        return 1;
    }
    else{
        std::cout << "SDL_Init OK " << std::endl;
    }
    SDL_Quit();
    return 0;
}
```

（8）编译并运行该项目，输出的错误信息如下：

```
/.../F5Codes/chapter6/build - SDLQtDemo1 - Desktop_Qt_5_9_8_MinGW_32 位 - Debug/Debug/
SDLQtDemo1.exe exited with code -1073741515
```

这是因为 SDLQtDemo1.exe 程序运行时找不到 SDL2.dll 动态链接库。将 SDL2-devel-2.0.12-mingw\i686-w64-mingw32\bin 目录下的 SDL2.dll 文件复制到 chapter6\build-SDLQtDemo1-Desktop_Qt_5_9_8_MinGW_32bit-Debug\debug 目录下。重新编译并运行该项目会输出 SDL_Init OK，如图 3-20 所示。

图 3-20　成功配置并运行 SDL2 项目

3.3.4　Linux 平台搭建 SDL2 开发环境

笔者本地环境为 Ubuntu 18.04，安装并配置 SDL2 的具体步骤如下。

(1) 安装依赖项，命令如下：

```
//chapter3/help - others.txt
sudo apt - get update && sudo apt - get  - y install \
  autoconf automake build - essential cmake \
  git - core pkg - config texinfo wget yasm zlib1g - dev
```

(2) 安装 SDL2 库(只包含.so 动态链接库)，命令如下：

```
sudo apt - get install libsdl2 - 2.0 libsdl2 - dev libsdl2 - mixer - dev libsdl2 - image - dev
libsdl2 - ttf - dev libsdl2 - gfx - dev
```

(3) 检验是否安装成功，命令如下：

```
sdl2 - config -- exec - prefix -- version - cflag
```

需要注意的是此处安装的 SDL2 库是没有头文件的，只包含系统运行时需要依赖的动态链接库(.so)，而在实际开发过程中没有头文件是不行的，所以需要自己编译 SDL2 并且安装。

(4) 下载并解压 SDL2 库的源码 SDL2-devel-2.0.12.tar.gz，具体的下载网址为 https://github.com/libsdl-org/SDL/releases/tag/release-2.0.12。

(5) 编译并安装 SDL2，命令如下：

```
//chapter3/help - others.txt
#解压下载的文件，然后进入 SDL2 解压目录
```

```
#配置 configure 的可执行命令
sudo chmod +x configure
#配置 configure 的参数命令
//chapter6/other-help.txt
./configure --enable-static --enable-shared
#编译:
    make
#安装
    make install
```

(6) 查看 SDL2 是否安装成功,命令如下:

```
//chapter3/help-others.txt
#在/usr/local/lib 下面查看是否存在 libSDL2.a
ls /usr/local/lib
#在/usr/local/include 下面查看是否存在 SDL2 文件夹
ls /usr/local/include
```

(7) 配置 LD_LIBRARY_PATH 环境变量,命令如下:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

3.3.5 SDL2 播放 YUV 视频文件

SDL2 的核心对象主要包括窗口(SDL_Window)、表面(SDL_Surface)、渲染器(SDL_Renderer)、纹理(SDL_Texture)和事件(SDL_Event)等。使用 SDL2 进行渲染的基本流程如图 3-21 所示,具体步骤如下。

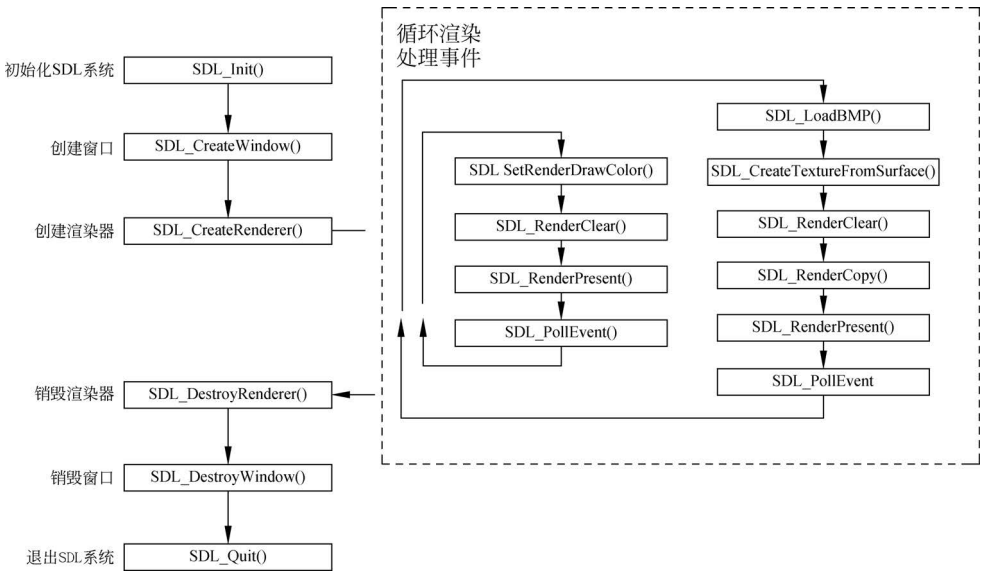

图 3-21 SDL2 渲染流程及 API

(1) 创建窗口。

(2) 创建渲染器。

(3) 清空缓冲区。

(4) 绘制要显示的内容。

(5) 最终将缓冲区内容渲染到 Window 窗口上。

使用 SDL2 播放 YUV 视频文件完全遵循上述 SDL_Renderer 的渲染流程,而 YUV 视频文件不能直接渲染,需要循环读取视频帧,然后将帧数据更新到纹理上进行渲染。

1. SDL2 播放 YUV 视频文件的流程

使用 SDL2 播放 YUV 视频文件的函数调用步骤及相关 API,代码如下:

```
//chapter3/SDLQtDemo1/main.cpp
/* SDL2 播放 YUV 视频文件,函数调用步骤如下
 *
 * [初始化 SDL2 库]
 * SDL_Init(): 初始化 SDL2
 * SDL_CreateWindow(): 创建窗口(Window)
 * SDL_CreateRenderer(): 基于窗口创建渲染器(Render)
 * SDL_CreateTexture(): 创建纹理(Texture)
 *
 * [循环渲染数据]
 * SDL_UpdateTexture(): 设置纹理的数据
 * SDL_RenderCopy(): 纹理复制给渲染器
 * SDL_RenderPresent(): 显示
 * SDL_DestroyTexture(texture);
 *
 * [释放资源]
 * SDL_DestroyTexture(texture): 销毁纹理
 * SDL_DestroyRenderer(render): 销毁渲染器
 * SDL_DestroyWindow(win): 销毁窗口
 * SDL_Quit(): 释放 SDL2 库
 */
```

2. 使用 SDL2 开发 YUV 视频播放器的完整案例

先介绍该案例程序中用到几个的重要变量类型,SDL_Window 就是使用 SDL 时弹出的那个窗口; SDL_Texture 用于显示 YUV 数据,一个 SDL_Texture 对应一帧 YUV 数据(案例中提供的 YUV 视频格式为 YUV420p); SDL_Renderer 用于将 SDL_Texture 渲染至 SDL_Window; SDL_Rect 用于确定 SDL_Texture 显示的位置。为了简单起见,程序中定义了几个全局变量,变量 g_bpp 代表 1 个视频像素占用的位数,例如 1 个 YUV420P 格式的视频像素占用 12 位; 变量 g_pixel_w 和 g_pixel_h 代表视频的宽和高,在本案例中提供的测试视频(ande10_yuv420p_352x288.yuv)的宽和高分别为 352 和 288; 变量 g_screen_w 和 g_screen_h 代表屏幕的宽和高,在本案例中被初始化为 400 和 300,程序运行中可以通过拖曳窗口的右下角来改变窗口的大小; 变量 g_buffer_YUV420p 是一字节数组,用于存储 1

帧 YUV420p 的视频数据，在播放视频的过程中会循环调用 SDL_UpdateTexture()函数以将该数组中存储的视频数据更新到纹理(SDL_Texture)中。refresh_video_SDL2()函数用于定时刷新，在本案例中通过 SDL_CreateThread()函数创建了一条线程，将线程的入口函数指定数为 refresh_video_SDL2()函数，固定的刷新周期为 40ms。本案例的代码如下：

注意：本案例的完整工程及代码可参考 chapter3/SDLQtDemo1 工程，代码位于 main2.cpp 文件中。

```
//chapter3/SDLQtDemo1/main2.cpp
#define SDL_MAIN_HANDLED                    //如果没有此宏,则会报错,并且要放到 SDL.h 之前
#include <iostream>
#include <SDL.h>
#include <vector>
using namespace std;

//刷新事件
#define REFRESH_EVENT (SDL_USEREVENT + 1)

int g_thread_exit = 0;
const int g_bpp = 12;                       //YUV420p,1 像素占用的位数
const int g_pixel_w = 352,g_pixel_h = 288; //在本案例中 YUV420p 视频的宽和高
int g_screen_w = 400, g_screen_h = 300;
//1 帧视频占用的字节数
unsigned char g_buffer_YUV420p[g_pixel_w * g_pixel_w * g_bpp / 8];

//增加画面刷新机制
int refresh_video_SDL2(void *opaque){
    while (g_thread_exit == 0) {
        SDL_Event event;
        event.type = REFRESH_EVENT;
        SDL_PushEvent(&event);
        SDL_Delay(40);
    }
    return 0;
}

int TestYUVPlayer001( ){
    if(SDL_Init(SDL_INIT_VIDEO)) {
        printf( "Could not initialize SDL - %s\n", SDL_GetError());
        return -1;
    }

    SDL_Window *screen;
    //SDL 2.0 对多窗口的支持
    screen = SDL_CreateWindow("SDL2 - YUVPlayer",
        SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
        g_screen_w, g_screen_h,SDL_WINDOW_OPENGL|SDL_WINDOW_RESIZABLE);
```

```
    if(!screen) {
        printf("SDL: could not create window - exiting: %s\n",SDL_GetError());
        return -1;
    }
    //创建渲染器
    SDL_Renderer* sdlRenderer = SDL_CreateRenderer(screen, -1, 0);
    //创建纹理:格式为 YUV420p,宽和高为 352x288
    SDL_Texture* sdlTexture =
        SDL_CreateTexture(sdlRenderer,SDL_PIXELFORMAT_IYUV,
            SDL_TEXTUREACCESS_STREAMING, g_pixel_w, g_pixel_h);

    FILE *fpYUV420p = NULL;          //打开 YUV420p 视频文件
    fpYUV420p = fopen("./ande10_yuv420p_352x288.yuv", "rb+");

    if(fpYUV420p == NULL){
        printf("cannot open this file\n");
        return -1;
    }
    SDL_Rect sdlRect;
    SDL_Thread *refresh_thread =     //创建独立线程,用于定时刷新
        SDL_CreateThread(refresh_video_SDL2,NULL,NULL);
    SDL_Event event;
    while(1){
        SDL_WaitEvent(&event);         //等待事件
        if(event.type == REFRESH_EVENT){
            fread(g_buffer_YUV420p, 1,
                  g_pixel_w * g_pixel_h * g_bpp / 8, fpYUV420p);
            //将 1 帧 YUV420p 的数据更新到纹理中
            SDL_UpdateTexture(sdlTexture,NULL,g_buffer_YUV420p,g_pixel_w);

            //重新定义窗口大小
            sdlRect.x = 0;
            sdlRect.y = 0;
            sdlRect.w = g_screen_w;
            sdlRect.h = g_screen_h;
            //Render:渲染三部曲
            SDL_RenderClear( sdlRenderer );
            SDL_RenderCopy( sdlRenderer, sdlTexture, NULL, &sdlRect);
            SDL_RenderPresent( sdlRenderer );
            //注意这里不再需要延迟,因为有独立的线程来刷新
            //SDL_Delay(40);           //休眠 40ms
            //if(feof(fpYUV420p) != 0 )break;  //如果遇到文件尾,则自动退出循环
        }else if(event.type == SDL_WINDOWEVENT){
            //If Resize:以拖曳方式更改窗口大小
            SDL_GetWindowSize(screen,&g_screen_w,&g_screen_h);
        }else if(event.type == SDL_QUIT){     //退出事件
          break;                       //如果关闭事件,则退出循环
        }
    }
```

```
    g_thread_exit = 1;     //如果跳出循环,则将退出标志量修改为 1
    //释放资源
    if (sdlTexture){
        SDL_DestroyTexture(sdlTexture);
        sdlTexture = nullptr;
    }
    if (sdlRenderer){
        SDL_DestroyRenderer(sdlRenderer);
        sdlRenderer = nullptr;
    }
    if (screen){
        SDL_DestroyWindow(screen);
        screen = nullptr;
    }
    if (fpYUV420p){
        fclose(fpYUV420p);
        fpYUV420p = nullptr;
    }
    SDL_Quit();

    return 0;
}
```

编译并运行该程序,将 ande10_yuv420p_352x288.yuv 这两个音频文件复制到 build-SDLQtDemo1-Desktop_Qt_5_9_8_MinGW_32bit-Debug 目录下,可以通过拖曳改变窗口大小,效果如图 3-22 所示。

图 3-22　SDL2 播放 YUV420p 视频

3. SDL2 画面刷新机制

实现 SDL2 的事件与渲染机制之后,增加画面刷新机制就可以作为一个播放器使用了。在上述案例中,通过 while 循环执行 SDL_RenderPresent(renderer)就可以令视频逐帧播放了,但还是需要一个独立的刷新机制。这是因为在一个循环中,重复执行一个函数的效果通常不是周期性的,因为每次加载和处理的数据所消耗的时间是不固定的,因此单纯地在一个循环中使用 SDL_RenderPresent(renderer)会令视频播放产生帧率跳动的情况,因此需要引入一个定期刷新机制,令视频的播放有一个固定的帧率。通常使用多线程的方式进行画面刷新管理,主线程进入主循环中等待(SDL_WaitEvent)事件,画面刷新线程在一段时间后发送(SDL_PushEvent)画面刷新事件,主线程收到画面刷新事件后进行画面刷新操作。

画面刷新线程定期构造一个 REFRESH_EVENT 事件,然后调用 SDL_PushEvent()函数将事件发送出来,代码如下:

```
//chapter3/help - others.txt
#define REFRESH_EVENT (SDL_USEREVENT + 1)
int g_thread_exit = 0;
int refresh_video_SDL2(void * opaque){
   while (g_thread_exit == 0) {
      SDL_Event event;
      event.type = REFRESH_EVENT;
      SDL_PushEvent(&event);
      SDL_Delay(40);
   }
   return 0;
}
```

该函数只有两部分内容，第一部分是发送画面刷新事件，也就是发信号以通知主线程来干活；另一部分是延时，使用一个定时器，保证自己是定期来通知主线程的。首先定义一个“刷新事件”，代码如下：

```
#define REFRESH_EVENT (SDL_USEREVENT + 1)       //请求画面刷新事件
```

SDL_USEREVENT 是自定义类型的 SDL 事件，不属于系统事件，可以由用户自定义，这里通过宏定义便于后续引用，然后调用 SDL_PushEvent()函数将事件发送出来，代码如下：

```
SDL_PushEvent(&event);                          //发送画面刷新事件
```

SDL_PushEvent()是 SDL2.0 之后引入的函数，该函数能够将事件放入 SDL2 的事件队列中，当它从事件队列中被取出时，被接收事件的函数识别，并采取相应操作。也就是说在刷新操作中，使用刷新线程不断地将“刷新事件”放到 SDL2 的事件队列，在主线程中读取 SDL2 的事件队列里的事件，当发现事件是“刷新事件”时就进行刷新操作。

主线程(main()函数)的主要工作是首先初始化所有的组件和变量，包括 SDL2 的窗口、渲染器和纹理等，然后进入一个大循环，同时读取事件队列里的事件，如果是刷新事件，则进行渲染相关工作，进行画面刷新。随后需要创建一个缓冲区，每次渲染时都是先从视频文件里读一帧，这一帧先存到缓冲区再交给渲染器去渲染。这个缓冲区的大小应该和视频文件的每帧大小是相同的，这也意味着需要提前计算该视频文件类型的每帧大小，所以需要提前计算好 YUV 格式的视频帧的大小，例如在本案例中视频文件的格式为 YUV420p，宽和高为 352×288。主循环其实就是在不断地读取事件队列里的事件，每读取到一个事件，就进行判断，根据该事件的类型采取不同的操作。当收到需要刷新画面的事件后，开始进行读数据帧并渲染的操作，代码如下：

```
//chapter3/help - others.txt
while(1){
```

```
    SDL_WaitEvent(&event);                    //等待事件
    if(event.type == REFRESH_EVENT){
        fread(g_buffer_YUV420p,1,g_pixel_w * g_pixel_h * g_bpp/8, fpYUV420p);
        //将 1 帧 YUV420p 的数据更新到纹理中
        SDL_UpdateTexture( sdlTexture,NULL,g_buffer_YUV420p,g_pixel_w);

        //渲染三部曲
        SDL_RenderClear( sdlRenderer );
        SDL_RenderCopy( sdlRenderer, sdlTexture, NULL, &sdlRect);
        SDL_RenderPresent( sdlRenderer );
        if(feof(fpYUV420p) != 0 )break;    //如果遇到文件尾,则自动退出循环
    }else if(event.type == SDL_WINDOWEVENT){
        //重定义窗口大小
        SDL_GetWindowSize(screen,&g_screen_w,&g_screen_h);
    }else if(event.type == SDL_QUIT){
        break;
    }
}
```

3.3.6 使用 FFmpeg+SDL2 读取本地摄像头并渲染

使用 FFmpeg 可以打开本地摄像头并循环读取视频帧数据，然后可以调用 SDL2 对视频帧进行渲染。打开 Qt Creator，创建一个基于 Widget 的 Qt Widgets Application 项目（项目名称为 FFmpegSDL2QtMonitor），如图 3-23 所示。

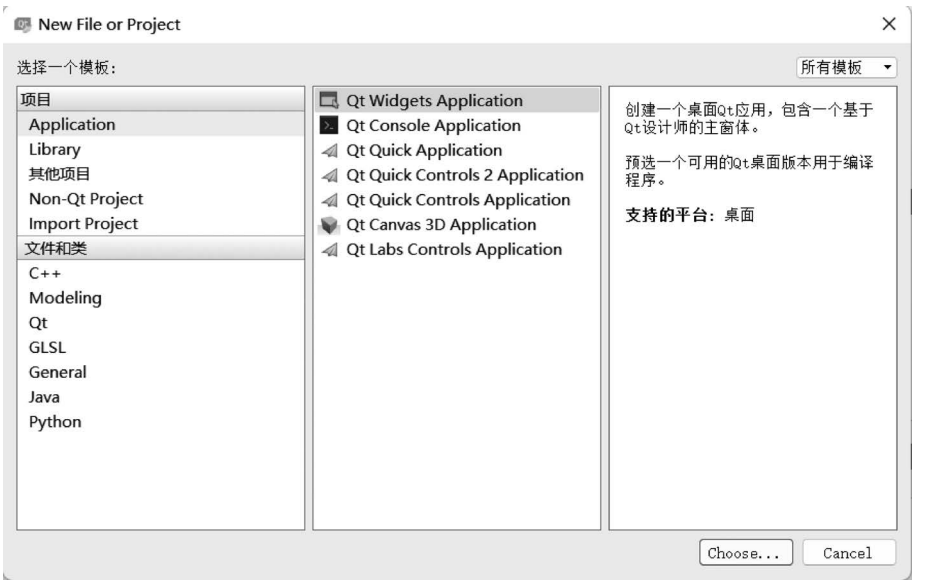

图 3-23 新建 Qt 的 Widgets 项目

双击 widget.ui 界面文件，界面中使用 QVerticalLayout 进行布局，然后往该界面中拖曳一个 QLabel 和两个 QPushButton（它们的文本分别为 Stop Camera 和 Start Camera），如图 3-24 所示。

图 3-24　设计 widget.ui 界面

右击 QPushButton 按钮，在弹出的菜单中选择“转到槽”，分别为这两个 QPushButton 按钮添加 clicked()槽函数，如图 3-25 所示。

图 3-25　为 QPushButton 添加槽函数

1. 封装一个类 QtFFmpegCamera 用于操作 FFmpeg

新增一个类 QtFFmpegCamera 用于操作 FFmpeg，为了响应 Qt 的信号槽机制，将该类的父类设置为 QObject。为了方便使用，需要创建一个 Play()和 SetStopped()等公共成员函数，并且新增 AVPicture、AVFormatContext、AVCodecContext、AVFrame、SwsContext 和 AVPacket 等类型的私有成员函数。该类的头文件为 qtffmpegcamera.h，代码如下：

```
//chapter3/FFmpegSDL2QtMonitor/qtffmpegcamera.h
#ifndef QTFFMPEGCAMERA_H
#define QTFFMPEGCAMERA_H
```

```
//必须加以下内容,否则编译不能通过,为了兼容 C 和 C99 标准
#ifndef INT64_C
#define INT64_C
#define UINT64_C
#endif

//引入 FFmpeg 和 SDL2 的头文件
#include <iostream>
extern "C"
{
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libswscale/swscale.h>
#include <libavdevice/avdevice.h>
#include <libavfilter/avfilter.h>
#include <libavutil/imgutils.h>
#include <SDL.h>
#include <SDL_main.h>
};
#undef main


using namespace std;

#include <QObject>
#include <QMutex>
#include <QImage>

class QtFFmpegCamera : public QObject
{
    Q_OBJECT
public:
    explicit QtFFmpegCamera(QObject * parent = nullptr);

    void Play();

    int GetVideoWidth() const { return this->videoWidth; }
    int GetVideoHeight() const {return this->videoHeight; }
    int GetVideoStreamIndex() const { return this->videoStreamIndex; }
    QString GetVideoURL() const {return this->videoURL; }
    void SetVideoURL(QString url){this->videoURL = url; }
    void SetStopped(int st){this->stopped = st;}

private:
    AVPicture pAVPicture;
    AVFormatContext * pAVFormatCtx;
    AVCodecContext * pAVCodecContext;
    AVFrame * pAVFrame;
```

```
    SwsContext * pSwsContext;          //将 YUYV422 转换为 YUV420p
    SwsContext * pSwsContext2;         //将 YUV420p 转换为 RGB888 (RGB24)
    AVPacket pAVPacket;

    QMutex       mutex;
    int          videoWidth;
    int          videoHeight;
    int          videoStreamIndex;
    QString      videoURL;
    int          stopped;

signals:
    void GetImage(const QImage &image);

public slots:
};

#endif //QTFFMPEGCAMERA_H
```

在该类的构造函数 QtFFmpegCamera::QtFFmpegCamera(QObject * parent)中对成员变量进行初始化,代码如下:

```
//chapter3/FFmpegSDL2QtMonitor/qtffmpegcamera.cpp
QtFFmpegCamera::QtFFmpegCamera(QObject * parent) : QObject(parent)
{
    stopped = 0;
    pAVFormatCtx = NULL;
    pAVCodecContext = NULL;
    pSwsContext = NULL;
    pSwsContext2 = NULL;
    pAVFrame = NULL;
}
```

在该类的成员函数 QtFFmpegCamera::Play()中初始化 FFmpeg 和 SDL2,使用 FFmpeg 打开本地摄像头并循环读取视频帧数据,然后调用 SDL2 进行渲染,代码如下:

```
//chapter3/FFmpegSDL2QtMonitor/qtffmpegcamera.cpp
void QtFFmpegCamera::Play(){
    avformat_network_init();                    //初始化 FFmpeg 的网络库

    pAVFormatCtx = avformat_alloc_context();   //分配格式化上下文环境
    pAVFormatCtx->probesize = 10000 * 1024;
    pAVFormatCtx->duration = 10 * AV_TIME_BASE;

    //1. 打开本地摄像头
    OpenLocalCamera(pAVFormatCtx, true);

    printf("---------- File Information:输出格式信息 ----------\n");
```

```
    av_dump_format(pAVFormatCtx, 0, NULL, 0);

    //2.寻找视频流信息
    if (avformat_find_stream_info(pAVFormatCtx, NULL) < 0)
    {
        printf("Couldn't find stream information.\n");
        return ;
    }

    //打开视频以获取视频流,设置视频默认索引值
    int videoindex = -1;
    for (int i = 0; i < pAVFormatCtx->nb_streams; i++)
    {
        if (pAVFormatCtx->streams[i]->codecpar->codec_type == AVMEDIA_TYPE_VIDEO)
        {
            videoStreamIndex = videoindex = i;
            //break;
        }
    }
    //如果没有找到视频的索引,则说明没有视频流
    if (videoindex == -1) {
        printf("Didn't find a video stream.\n");
        return ;
    }

    //3. 打开解码器
    //AVCodecContext 为解码上下文结构体
    //avcodec_alloc_context3 为解码分配函数
    //avcodec_parameters_to_context 为参数格式转换
    //avcodec_find_decoder(codec_ID) 用于查找解码器
    //avcodec_open2 用于打开解码器
    //分配解码器上下文
    pAVCodecContext = avcodec_alloc_context3(NULL);
    //获取解码器上下文信息
    if (avcodec_parameters_to_context(pAVCodecContext, pAVFormatCtx->streams[videoindex]->
codecpar) < 0)
    {
        cout << "Copy stream failed!" << endl;
        return ;
    }
    //查找解码器//codec_id = 13
    //AV_CODEC_ID_RAWVIDEO: 13
    //AV_CODEC_ID_H264: 27
    printf("codec_id = %d\n", pAVCodecContext->codec_id);
    AVCodec *pCodec = avcodec_find_decoder(pAVCodecContext->codec_id);
    if (pCodec == NULL) {
        printf("Codec not found.\n");
        return ;
    }
```

```
    //打开解码器
    if (avcodec_open2(pAVCodecContext, pCodec, NULL) < 0)
    {
        printf("Could not open codec.\n");
        return ;
    }

    //4. 格式转换
    //(1)sdl: yuyv422 --- > yuv420p:SDL2 渲染需要使用 YUV420p 格式
    //(2)Qt : yuv420p --- > rgb24:Qt 渲染需要使用 RGB24 格式
    //对图形进行裁剪以便于显示得更好
    pSwsContext = sws_getContext(
                pAVCodecContext - > width, pAVCodecContext - > height, pAVCodecContext - > pix_
fmt,pAVCodecContext - > width, pAVCodecContext - > height, AV_PIX_FMT_YUV420P,SWS_BICUBIC,
NULL, NULL, NULL);

    pSwsContext2 = sws_getContext(
                 pAVCodecContext - > width, pAVCodecContext - > height, AV_PIX_FMT_YUV420P,
pAVCodecContext - > width, pAVCodecContext - > height, AV_PIX_FMT_RGB24,SWS_BICUBIC, NULL,
NULL, NULL);


    if (NULL == pSwsContext) {
        cout << "Get swscale context failed!" << endl;
        return ;
    }

    //获取视频流的分辨率大小
    //pAVCodecContext = pAVFormatContext - > streams[videoStreamIndex] - > codec;
    videoWidth = pAVCodecContext - > width;          //视频宽度
    videoHeight = pAVCodecContext - > height;        //视频高度
    avpicture_alloc(&pAVPicture, AV_PIX_FMT_RGB24, videoWidth, videoHeight);
                                                     //以视频格式及分辨率来分配内存


    //5. SDL2.0:初始化 SDL2 的库
    if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER)){
        printf("Could not initialize SDL - % s\n", SDL_GetError());
        return;
    }
    //SDL2 的多窗口支持
    int screen_w = pAVCodecContext - > width;
    int screen_h = pAVCodecContext - > height;
    SDL_Window * screen = SDL_CreateWindow("FFmpegPlayer", SDL_WINDOWPOS_UNDEFINED, SDL_
WINDOWPOS_UNDEFINED, screen_w, screen_h, SDL_WINDOW_OPENGL);
    if (!screen){
        printf("SDL: could not create window - exiting: % s\n", SDL_GetError());
        return ;
    }
```

```
//创建渲染器
SDL_Renderer * sdlRenderer = SDL_CreateRenderer(screen, -1, 0);

//创建纹理
   //IYUV: Y + U + V (3 planes):3 个平面
   //YV12: Y + V + U (3 planes):3 个平面
   SDL_Texture * sdlTexture = SDL_CreateTexture(sdlRenderer, SDL_PIXELFORMAT_IYUV, SDL_
TEXTUREACCESS_STREAMING,pAVCodecContext->width, pAVCodecContext->height);

   SDL_Rect sdlRect;
   sdlRect.x = 0;
   sdlRect.y = 0;
   sdlRect.w = screen_w;
   sdlRect.h = screen_h;

   //创建渲染刷新线程:thread.sdl
   SDL_Thread *video_tid = SDL_CreateThread(sfp_refresh_thread, NULL, NULL);


   //6. 使用 FFmpeg 解封装并解码
   AVPacket *packet = (AVPacket *)av_malloc(sizeof(AVPacket));
   AVFrame *pFrame = av_frame_alloc();
   AVFrame *pFrameYUV420p = av_frame_alloc();
uint8_t *out_buffer = (uint8_t *)av_malloc(av_image_get_buffer_size(
AV_PIX_FMT_YUV420p,pAVCodecContext->width,pAVCodecContext->height, 1));
   av_image_fill_arrays( pFrameYUV420p->data, pFrameYUV420p->linesize, out_buffer,AV_PIX_
FMT_YUV420P,pAVCodecContext->width, AVCodecContext->height, 1);


   //一帧一帧地读取视频:事件循环
   SDL_Event event;
   for (;;) {
      if(this->stopped){          //检测是否需要停止
         thread_exit = 1;
         break;
      }
      //等待事件:Wait
      SDL_WaitEvent(&event);
      if (event.type == SFM_REFRESH_EVENT) {
         //---------- 读取视频包 ----------
            if (av_read_frame(pAVFormatCtx, packet) >= 0) //解封装
            {
                if (packet->stream_index == videoindex) {
                   //解码 : YUYV422 --->YUV420p
                   decode(pAVCodecContext, packet, pFrame, pFrameYUV420p, pSwsContext);


                    //格式转换: YUV420p -->RGB24
                    mutex.lock();
```

```
                    sws_scale(pSwsContext2,
                            (const uint8_t * const * )pFrameYUV420p->data,
                            pFrameYUV420p->linesize,
                            0,videoHeight,
                            pAVPicture.data,
                            pAVPicture.linesize);
                    //Qt 发送获取一帧图像信号:注意下面两行代码使用的是 Qt 的渲染机制
                     //QImage image(pAVPicture.data[0],videoWidth,videoHeight,QImage::Format_
RGB888);
                    //emit GetImage(image);
                    mutex.unlock();

                    //SDL2:渲染视频帧
                        SDL_UpdateTexture(sdlTexture, NULL, pFrameYUV420p->data[0],
pFrameYUV420p->linesize[0]);
                    SDL_RenderClear(sdlRenderer);
                    SDL_RenderCopy(sdlRenderer, sdlTexture, &sdlRect, &sdlRect);
                    SDL_RenderCopy(sdlRenderer, sdlTexture, NULL, NULL);
                    SDL_RenderPresent(sdlRenderer);
                                    }
                av_packet_unref(packet);
            }
            else {
                //退出线程
                thread_exit = 1;
            }
        }
        else if (event.type == SDL_KEYDOWN) {
            qDebug() << "keywdown";
            //暂停
            if (event.key.keysym.sym == SDLK_SPACE)
                thread_pause = !thread_pause;
            else if (event.key.keysym.sym == SDLK_ESCAPE){
                thread_exit = 1;
                qDebug() << SDLK_ESCAPE;
            }
        }
        else if (event.type == SDL_QUIT) {
            thread_exit = 1;
        }
        else if (event.type == SFM_BREAK_EVENT) {
            break;
        }
    }


    sws_freeContext(pSwsContext);
    SDL_Quit();
    av_frame_free(&pFrameYUV420p);
```

```
    av_frame_free(&pFrame);
    avcodec_close(pAVCodecContext);
    avformat_close_input(&pAVFormatCtx);
}
```

在 qtffmpegcamera.cpp 文件中有一个 OpenLocalCamera()静态函数,它的功能是调用 FFmpeg 来打开本地摄像头,代码是通用的,可以兼容 Windows 和 Linux 平台,但传递的参数略有区别。该函数的完整代码如下:

```
//chapter3/FFmpegSDL2QtMonitor/qtffmpegcamera.cpp
/**************************
* 打开本地摄像头
**************************/
static int OpenLocalCamera(AVFormatContext *pFormatCtx, bool isUseDshow = false)
{
    avdevice_register_all();              //注册 Libavdevice 库
#ifdef _WIN32
    if (isUseDshow) {                     //使用 DShow 方式打开摄像头
        AVInputFormat *ifmt = av_find_input_format("dshow");
        //设置视频设备的名称
        //if (avformat_open_input(&pFormatCtx, "video=Lenovo EasyCamera", ifmt, NULL) != 0)
        if (avformat_open_input(&pFormatCtx, "video=HP TrueVision HD Camera", ifmt, NULL) !=
0) {
            printf("Couldn't open input stream.(无法打开输入流)\n");
            return -1;
        }
    }
    else {//使用 VFW 方式打开摄像头
        AVInputFormat *ifmt = av_find_input_format("vfwcap");
        if (avformat_open_input(&pFormatCtx, "0", ifmt, NULL) != 0) {
            printf("Couldn't open input stream.(无法打开输入流)\n");
            return -1;
        }
    }
#endif
    //Linux 平台
#ifdef linux
    AVInputFormat *ifmt = av_find_input_format("video4linux2");
    if (avformat_open_input(&pFormatCtx, "/dev/video0", ifmt, NULL) != 0) {
        printf("Couldn't open input stream.(无法打开输入流)\n");
        return -1;
    }

#endif

    return 0;

}
```

注意：在 Windows 平台下 avformat_open_input(&pFormatCtx, "video=HP TrueVision HD Camera", ifmt, NULL)函数中的参数需要修改为读者本地的摄像头名称，例如笔者这里的摄像头名称为 HP TrueVision HD Camera。

在 qtffmpegcamera.cpp 文件中有一个 sfp_refresh_thread()静态函数，它的功能是手工创建 SDL 的刷新事件，以此来实时刷新视频帧，代码如下：

```
//chapter3/FFmpegSDL2QtMonitor/qtffmpegcamera.cpp
//Refresh Event
#define SFM_REFRESH_EVENT (SDL_USEREVENT + 1)
#define SFM_BREAK_EVENT (SDL_USEREVENT + 2)
static int thread_exit = 0;
static int thread_pause = 0;

static int sfp_refresh_thread(void * opaque)
{
    thread_exit = 0;
    thread_pause = 0;

    while (!thread_exit){
        if (!thread_pause) {           //判断是否暂停
            //手工创建 SDL 刷新事件
            SDL_Event event;
            event.type = SFM_REFRESH_EVENT;
            SDL_PushEvent(&event);
        }
        SDL_Delay(5);

    }
    thread_exit = 0;
    thread_pause = 0;
    //Break:退出事件
    SDL_Event event;
    event.type = SFM_BREAK_EVENT;
    SDL_PushEvent(&event);
    return 0;
}
```

2. 新增 1 个 Qt 的线程类来调用 FFmpeg

因为读取视频帧需要使用 while 循环，从界面上单击 Start Camera 按钮之后，如果直接进入 while 循环，就会导致界面僵死，所以需要开启一个独立的线程来源源不断地读取视频帧并解码，然后将解码出来的 YUV 帧数据送给 SDL2 进行渲染。可以使用 Qt 的 QThread 类来封装一个线程类，头文件代码如下：

```
//chapter3/FFmpegSDL2QtMonitor/qtcamerathread.h
#ifndef QTCAMERATHREAD_H
#define QTCAMERATHREAD_H

#include <QThread>
#include "qtffmpegcamera.h"

class QtCameraThread : public QThread{
    Q_OBJECT
public:
    explicit QtCameraThread(QObject * parent = nullptr);
    void run();
    void setffmpeg(QtFFmpegCamera * f){ffmpeg = f;}

private:
    QtFFmpegCamera * ffmpeg;

signals:

public slots:
};

#endif //QTCAMERATHREAD_H
```

QtCameraThread 类的基类是 QThread，在该类中有一个 QtFFmpegCamera 类型的私有成员变量，用于操作 FFmpeg，然后需要重写 QThread 的 run()虚函数，代码如下：

```
//chapter3/FFmpegSDL2QtMonitor/qtcamerathread.cpp
#include "qtcamerathread.h"
QtCameraThread::QtCameraThread(QObject * parent) :
    QThread(parent) {
}
void QtCameraThread::run() {
    ffmpeg->Play();
}
```

由此可见，在 run()虚函数中主要调用 QtFFmpegCamera 类的 Play()函数，先打开本地摄像头，然后循环读取视频帧并通过 SDL2 进行渲染。

3. 通过 Qt 的界面按钮开启或停止摄像头

在 Start Camera 按钮的 Widget::on_pushButton_OpenCamera_clicked()槽函数中开启线程即可，代码如下：

```
//chapter3/FFmpegSDL2QtMonitor/qtcamerathread.cpp
void Widget::on_pushButton_OpenCamera_clicked(){
    qDebug() << "clicked\n";
    //修改按钮状态
```

```
    ui->pushButton_stop->setEnabled(true);
    ui->pushButton_OpenCamera->setEnabled(false);
    objFmpg.SetStopped(0);
    //objFmpg.Play();//注意:不要直接调用 FFmpeg 封装类的 Play()函数,否则会导致界面僵死
    //通过线程方式来开启 FFmpeg 封装类的 Play()函数
    QtCameraThread *rtsp = new QtCameraThread(this);
    rtsp->setffmpeg(&objFmpg);
    rtsp->start();
}
```

在 Stop Camera 按钮的 Widget::on_pushButton_stop_clicked()槽函数中修改播放状态即可,代码如下:

```
//chapter3/FFmpegSDL2QtMonitor/qtcamerathread.cpp
void Widget::on_pushButton_stop_clicked(){
    objFmpg.SetStopped(1);              //将播放状态设置为 Stopped 即可
    //修改按钮的状态
    ui->pushButton_stop->setEnabled(false );
    ui->pushButton_OpenCamera->setEnabled( true);
}
```

编译并运行该程序,发现使用 SDL2 渲染时会弹出一个单独的窗口,如图 3-26 所示。

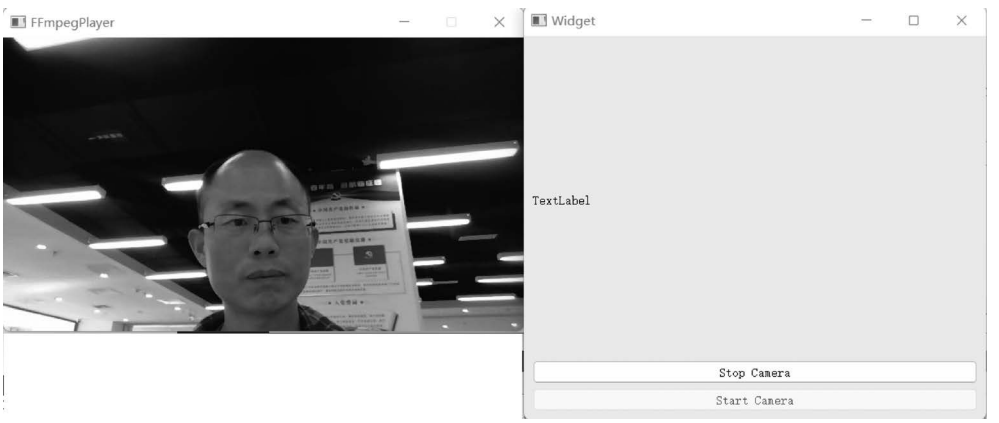

图 3-26　SDL2 弹出的窗口

4. 将 SDL2 弹出的窗口嵌入 Qt 的 QLabel 中

可以将 SDL2 弹出的窗口嵌入 Qt 的 QLabel 中,将 SDL_CreateWindow()函数替换为 SDL_CreateWindowFrom()函数即可,代码如下:

```
//chapter3/FFmpegSDL2QtMonitor/qtcamerathread.cpp
//SDL_Window * screen = SDL_CreateWindow("FFmpegPlayer", SDL_WINDOWPOS_UNDEFINED, SDL_
WINDOWPOS_UNDEFINED, screen_w, screen_h, SDL_WINDOW_OPENGL);
//将 SDL2 窗口嵌入 Qt 子窗口的方法
SDL_Window * screen = SDL_CreateWindowFrom( m_MainWidget );
```

重新编译并运行该程序，SDL2 的窗口已经被嵌入 Qt 的 QLabel 标签中，如图 3-27 所示。

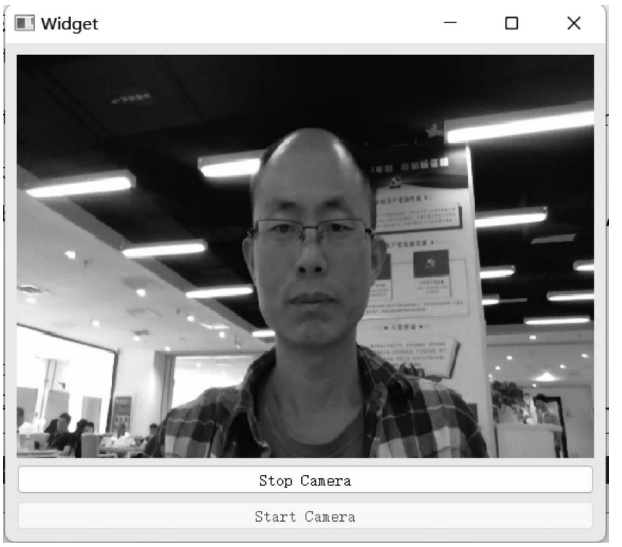

图 3-27　将 SDL2 弹出的窗口嵌入 Qt 窗口中

3.4　FFmpeg＋Qt 读取并显示本地摄像头

信号与槽(Signal & Slot)是 Qt 编程的基础，也是 Qt 的一大创新。因为有了信号与槽的编程机制，在 Qt 中处理界面各个组件的交互操作时会变得更加直观和简单。信号槽是 Qt 框架引以为豪的机制之一。所谓信号槽，实际就是观察者模式。当某个事件发生之后，例如按钮检测到自己被单击了一下，它就会发出一个信号(Signal)。以这种方式发出信号类似广播。如果有对象对这个信号感兴趣，则可以使用连接(Connect)函数将想要处理的信号和自己的一个槽函数(Slot)绑定，以此来处理这个信号。也就是说，当信号发出时，被连接的槽函数会自动被回调。信号与槽机制是 Qt GUI 编程的基础，使用信号与槽机制可以比较容易地将信号与响应代码关联起来。

3.4.1　信号

信号(Signal)就是在特定情况下被发射的事件，例如下压式按钮(PushButton)最常见的信号就是鼠标单击时发射的 clicked()信号，而一个组合下拉列表(ComboBox)最常见的信号是选择的列表项变化时发射的 CurrentIndexChanged()信号。GUI 程序设计的主要内容就是对界面上各组件的信号进行响应，只需知道什么情况下发射哪些信号，合理地去响应和处理这些信号就可以了。信号是一个特殊的成员函数声明，返回值的类型为 void，只能声明而不能通过定义实现。信号必须用 signals 关键字声明，访问属性为 protected，只能通过

emit 关键字调用(发射信号)。当某个信号对其客户或所有者发生的内部状态发生改变时,信号被一个对象发射。只有定义过这个信号的类及其派生类能够发射这个信号。当一个信号被发射时,与其相关联的槽将被立刻执行,就像一个正常的函数调用一样。信号槽机制完全独立于任何 GUI 事件循环。只有当所有的槽返回以后发射函数(emit)才返回。如果存在多个槽与某个信号相关联,则当这个信号被发射时,这些槽将会一个接一个地执行,但执行的顺序将会是随机的,不能人为地指定哪个先执行、哪个后执行。信号的声明是在头文件中进行的,Qt 的 signals 关键字用于指出进入了信号声明区,随后即可声明自己的信号,代码如下:

```
signals:
    void mycustomsignals();
```

signals 是 QT 的关键字,而非 C/C++的关键字。信号可以重载,但信号却没有函数体定义,并且信号的返回类型都是 void,不要指望能从信号返回什么有用信息。信号由 MOC 自动产生,不应该在.cpp 文件中实现。

3.4.2　槽

槽(Slot)就是对信号响应的函数,即槽就是一个函数,与一般的 C++函数是一样的,可以定义在类的任何部分(public、private 或 protected),可以具有任何参数,也可以被直接调用。槽函数与一般函数的不同点在于:槽函数可以与一个信号关联,当信号被发射时,关联的槽函数被自动执行。槽也能够声明为虚函数。槽的声明也是在头文件中进行的,代码如下:

```
public slots:
    void setValue(int value);
```

只有 QObject 的子类才能自定义槽,定义槽的类必须在类声明的最开始处使用 Q_OBJECT,类中声明槽需要使用 slots 关键字,槽与所处理的信号在函数签名上必须一致。

3.4.3　信号与槽的关联

信号与槽关联是用 QObject::connect() 函数实现的,其代码如下:

```
//chapter3/qt-help-apis.txt
//QObject::connect(sender, SIGNAL(signal()), receiver, SLOT(slot()));
bool QObject::connect ( const QObject * sender, const char * signal,
                        const QObject * receiver, const char * method,
                        Qt::ConnectionType type = Qt::AutoConnection );
```

connect()函数是 QObject 类的一个静态函数,而 QObject 是所有 Qt 类的基类,在实际调用时可以忽略前面的限定符,所以可以直接写为如下形式。

```
connect(sender, SIGNAL(signal()), receiver, SLOT(slot()));
```

其中，sender 是发射信号的对象的名称，signal() 是信号名称。信号可以看作特殊的函数，需要带圆括号，有参数时还需要指明参数。receiver 是接收信号的对象名称，slot() 是槽函数的名称，需要带圆括号，有参数时还需要指明参数。SIGNAL 和 SLOT 是 Qt 的宏，用于指明信号和槽，并将它们的参数转换为相应的字符串。一段简单的代码如下：

```
QObject::connect(btnClose, SIGNAL(clicked()), Widget, SLOT(close()));
```

这行代码的作用就是将 btnClose 按钮的 clicked()信号与窗体(Widget)的槽函数 close()相关联，当单击 btnClose 按钮(界面上的 Close 按钮)时，就会执行 Widget 的 close()槽函数。

当信号与槽没有必要继续保持关联时，可以使用 disconnect 函数来断开连接，代码如下：

```
bool QObject::disconnect (const QObject * sender, const char * signal,
                          const QObject * receiver, const char * method);
```

disconnect()函数用于断开发射者中的信号与接收者中的槽函数之间的关联。在 disconnect()函数中 0 可以用作一个通配符，分别表示任何信号、任何接收对象、接收对象中的任何槽函数，但是发射者 sender 不能为 0，其他 3 个参数的值可以等于 0。以下 3 种情况需要使用 disconnect()函数断开信号与槽的关联。

(1) 断开与某个对象相关联的任何对象，代码如下：

```
disconnect(sender, 0, 0, 0);
sender->disconnect();
```

(2) 断开与某个特定信号的任何关联，代码如下：

```
disconnect(sender, SIGNAL(mySignal()), 0, 0);
sender->disconnect(SIGNAL(mySignal()));
```

(3) 断开两个对象之间的关联，代码如下：

```
disconnect(sender, 0, receiver, 0);
sender->disconnect(receiver);
```

3.4.4 信号与槽的注意事项

Qt 利用信号与槽(Signal/Slot)机制取代传统的回调函数机制(callback)进行对象之间的沟通。当操作事件发生时，对象会提交一个信号(Signal)，而槽(Slot)则是一个函数接收特定信号并且运行槽本身设置的动作。信号与槽之间需要通过 QObject 的静态方法

connect()函数连接。信号在任何运行点上皆可发射，甚至可以在槽里再发射另一个信号，信号与槽的链接不限定为一对一的链接，一个信号可以链接到多个槽或者多个信号链接到同一个槽，甚至信号也可连接到信号。以往的 callback 缺乏类型安全，在调用处理函数时，无法确定是传递正确型态的参数，但信号和其接收的槽之间传递的数据型态必须相匹配，否则编译器会发出警告。信号和槽可接收任何数量、任何形态的参数，所以信号与槽机制是完全类型安全。信号与槽机制也确保了低耦合性，发送信号的类并不知道可被哪个槽接收，也就是说一个信号可以调用所有可用的槽。此机制会确保当“连接”信号和槽时，槽会接收信号的参数并且正确运行。关于信号与槽的使用，需要注意以下规则。

(1) 一个信号可以连接多个槽，代码如下：

```
connect(spinNum, SIGNAL(valueChanged(int)), this, SLOT(addFun(int));
connect(spinNum, SIGNAL(valueChanged(int)), this, SLOT(updateStatus(int));
```

当一个对象 spinNum 的数值发生变化时，所在窗体有两个槽函数进行响应，一个 addFun()函数用于计算，另一个 updateStatus()函数用于更新状态。当一个信号与多个槽函数关联时，槽函数按照建立连接时的按顺序依次执行。当信号和槽函数带有参数时，在 connect()函数里要写明参数的类型，但可以不写参数名称。

(2) 多个信号可以连接同一个槽，例如将 3 个选择颜色的 RadioButton 的 clicked()信号关联到相同的一个自定义槽函数 setTextFontColor()，代码如下：

```
//chapter3/qt - help - apis.txt
connect(ui - > rBtnBlue,SIGNAL(clicked()),this,SLOT(setTextFontColor()));
connect(ui - > rBtnRed,SIGNAL(clicked()),this,SLOT(setTextFontColor()));
connect(ui - > rBtnBlack,SIGNAL(clicked()),this,SLOT(setTextFontColor()));
```

当任何一个 RadioButton 被单击时，都会执行 setTextFontColor() 槽函数。

(3) 一个信号可以连接另外一个信号，代码如下：

```
connect(spinNum, SIGNAL(valueChanged(int)), this, SIGNAL (refreshInfo(int));
```

当一个信号发射时，也会发射另外一个信号，实现某些特殊的功能。

(4) 在严格的情况下，信号与槽的参数的个数和类型需要一致，至少信号的参数不能少于槽的参数。如果不匹配，则会出现编译错误或运行错误。

(5) 在使用信号与槽的类中，必须在类的定义中加入宏 Q_OBJECT。

(6) 当一个信号被发射时，与其关联的槽函数通常会被立即执行，就像正常调用一个函数一样。只有当信号关联的所有槽函数执行完毕后，才会执行发射信号处后面的代码。

3.4.5　元对象工具

元对象编译器(Meta Object Compiler，MOC)对 C++文件中的类声明进行分析并生成用于初始化元对象的 C++代码，元对象包含全部信号和槽的名字及指向槽函数的指针。

当 MOC 读 C++源文件时，如果发现有 Q_OBJECT 宏声明的类，就会生成另外一个 C++源文件，新生成的文件中包含该类的元对象代码。假设有一个头文件 mysignal.h，在这个文件中包含信号或槽的声明，那么在编译之前 MOC 工具就会根据该文件自动生成一个名为 mysignal.moc.h 的 C++源文件并将其提交给编译器；对应的 mysignal.cpp 文件 MOC 工具将自动生成一个名为 mysignal.moc.cpp 的文件提交给编译器。

元对象代码是 Signal/Slot 机制所必需的。用 MOC 生成的 C++源文件必须与类实现一起进行编译和连接，或者用 #include 语句将其包含到类的源文件中。MOC 并不扩展 #include 或者 #define 宏定义，只是简单地跳过所遇到的任何预处理指令。

信号和槽函数的声明一般位于头文件中，同时在类声明的开始位置必须加上 Q_OBJECT 语句，Q_OBJECT 语句将告诉编译器在编译之前必须先应用 MOC 工具进行扩展。关键字 signals 是对信号的声明，signals 默认为 protected 等属性。关键字 slots 是对槽函数的声明，slots 有 public、private、protected 等属性。signals、slots 关键字是 Qt 自己定义的，不是 C++中的关键字。信号的声明类似于函数的声明而非变量的声明，左边要有类型，右边要有括号，如果要向槽中传递参数，则可在括号中指定每个形式参数的类型，而形式参数的个数可以多于一个。关键字 slots 指出随后开始槽的声明，这里 slots 用的也是复数形式。槽的声明与普通函数的声明一样，可以携带零或多个形式参数。既然信号的声明类似于普通 C++函数的声明，那么信号也可采用 C++中虚函数的形式进行声明，即同名但参数不同。例如，第 1 次定义的 void mySignal()没有带参数，而第 2 次定义的却带有参数，从这里可以看出 Qt 的信号机制是非常灵活的。信号与槽之间的联系必须事先用 connect()函数进行指定。如果要断开二者之间的联系，则可以使用 disconnect()函数。

3.4.6 案例：标准信号槽

新建一个 Qt Widgets Application 项目(笔者的项目名称为 MySignalSlotsDemo)，基类选择 QWidget，如图 3-28 所示，然后在构造函数中动态地创建一个按钮，实现单击按钮关闭窗口的功能。编译并运行该程序，效果如图 3-29 所示。本项目包含的代码如下：

注意：该案例的完整工程代码可参考本书源码中的 chapter3/MySignalSlotsDemo，建议读者先下载源码将工程运行起来，然后结合本书进行学习。

```
//chapter3/MySignalSlotsDemo/widget.h
//widget.h头文件////
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>

namespace Ui {
```

```
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

public:
    explicit Widget(QWidget * parent = nullptr);
    ~Widget();

private:
    Ui::Widget * ui;
};
#endif //WIDGET_H


/////widget.cpp 文件//////
#include "widget.h"
#include "ui_widget.h"
#include <QPushButton>

Widget::Widget(QWidget * parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);

    //创建一个按钮
    QPushButton * btn = new QPushButton;
    //btn->show();                          //以顶层方式弹出窗口控件
    //让 btn 依赖在 myWidget 窗口中
    btn->setParent(this);                   //this 指当前窗口
    btn->setText("关闭");
btn->move(100,100);

    //关联信号和槽:单击按钮关闭窗口
//参数 1:信号发送者;参数 2:发送的信号(函数地址)
//参数 3:信号接收者;参数 4:处理槽函数地址
    connect(btn, &QPushButton::clicked, this, &QWidget::close);

}

Widget::~Widget()
{
    delete ui;
}
```

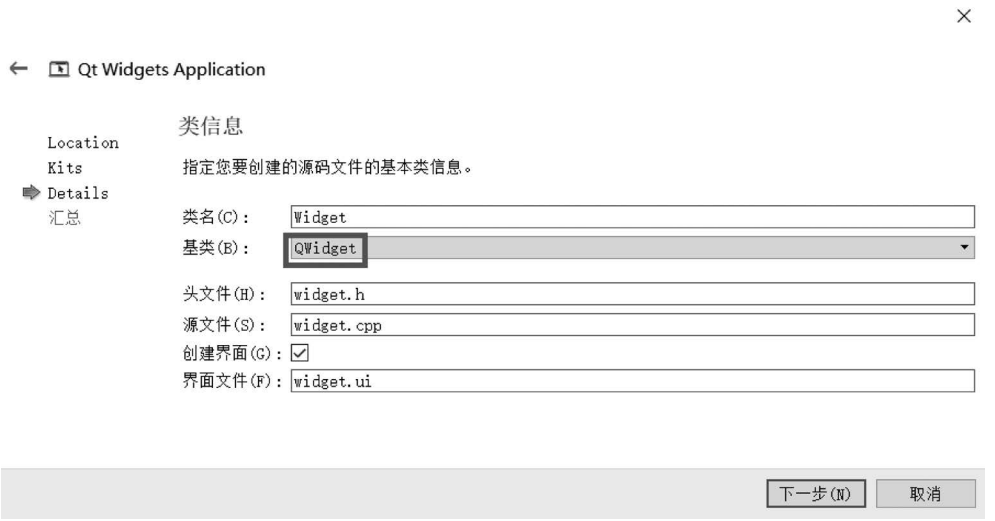

图 3-28　Qt Widgets 项目的基类选择

图 3-29　Qt 信号槽的运行效果

3.4.7　案例：自定义信号槽

当 Qt 提供的标准信号和槽函数无法满足需求时，就需要用到自定义信号槽，可以使用 emit 关键字来发射信号。例如定义老师和学生两个类(都继承自 QObject)，当老师发出“下课”信号时，学生响应“去吃饭”的槽功能。由于“下课”不是 Qt 标准的信号，所以需要用到自定义信号槽机制。这里不再创建新的 Qt 项目，直接使用 3.4.6 节的 MySignalSlotsDemo 项目，先添加两个自定义类 Teacher 和 Student，它们都继承自 QObject。右击项目名称 MySignalSlotsDemo，在弹出的菜单中选择 Add New... 菜单选项，如图 3-30 所示，然后在

弹出的“新建文件”对话框中，单击左侧的 C++模板，在右侧选择 C++Class，如图 3-31 所示，接着在弹出的 C++Class 对话框中，输入 Class name(Teacher)，在 Base class 下拉列表中选择 QObject，如图 3-32 所示，再以同样的步骤创建 Student 类，成功后，项目中多了两个类(Teacher 和 Student)，如图 3-33 所示。

图 3-30　Qt 项目中通过 Add New 添加新项

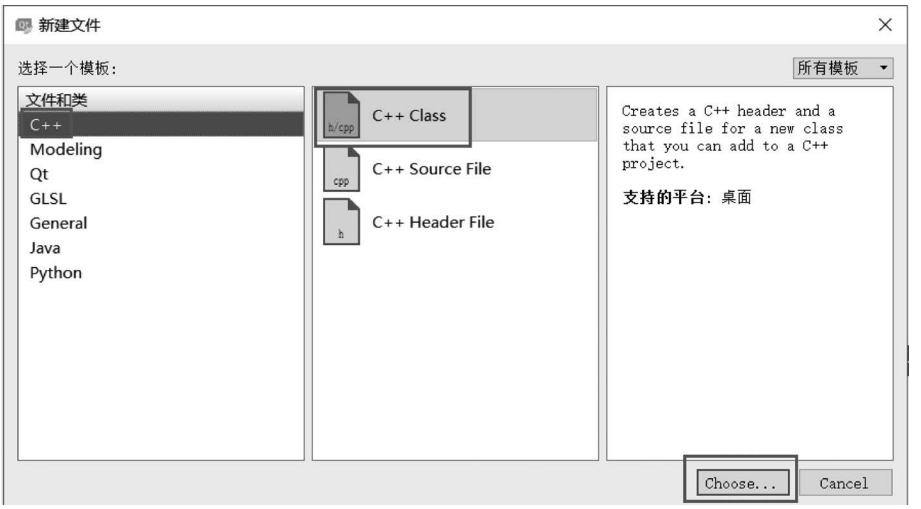

图 3-31　Qt 项目中选择 C++Class

在 Teacher 类中添加一个“下课”信号(finishClass)，代码如下：

```
//chapter3/MySignalSlotsDemo/student.h
signals:
    //自定义信号,写到 signals 下
    //返回值为 void,只用申明,不需要实现
    //可以有参数,也可以重载
    void finishClass();
```

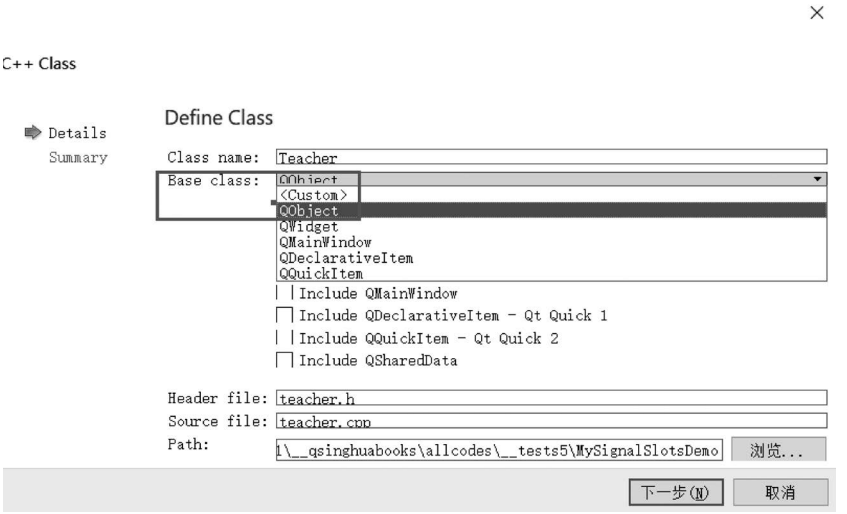

图 3-32　Qt 项目中添加新类并选择 QObject 基类

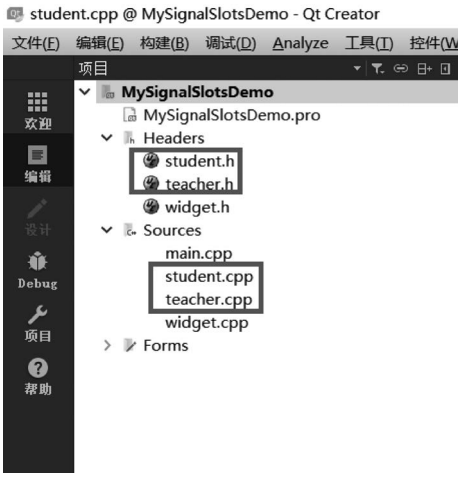

图 3-33　Qt 项目中添加了两个类

在 Student 类中添加一个“去吃饭”槽(gotoEat),代码如下:

```
//chapter3/MySignalSlotsDemo/student.h
public slots:
//早期 Qt 版本需要写到 public slots 下,高级版本可以写到 public 或全局下
//返回值为 void,需要声明,也需要实现
//可以有参数,也可以重载
    void gotoEat();

//sutdent.cpp
void Student::gotoEat()
```

```
{
    qDebug() << "准备去吃饭.....";
}
```

在 Widget 类中声明老师类(Student)和学生类(Student)的成员变量,并在构造函数中通过 new 创建实例,然后通过 connect()函数来关联老师类的“下课”信号和学生类的“去吃饭”槽,代码如下:

```
//chapter3/MySignalSlotsDemo/widget.h
private:
    Teacher * m_teacher;
    Student * m_student;

//widget.cpp
Widget::Widget(QWidget * parent):
    QWidget(parent),
    ui(new Ui::Widget)
{
    ...
    //创建老师对象
    this->m_teacher = new Teacher(this);
    //创建学生对象
    this->m_student = new Student(this);
    //连接老师的"下课"信号和学生的"去吃饭"槽函数
    connect(m_teacher,&Teacher::finishClass,
            m_student,&Student::gotoEat);
}
```

在界面上拖曳一个按钮,将文本内容修改为“下课”,用来模拟老师的下课信号,然后双击这个按钮,在 Qt 自动生成的 Widget::on_pushButton_clicked()函数中添加的代码如下:

```
//chapter3/MySignalSlotsDemo/widget.cpp
void Widget::on_pushButton_clicked()
{
    //通过 emit 发射信号
    emit this->m_teacher->finishClass();
}
```

编译并运行该程序,单击“下课”按钮后会在控制台输出“准备去吃饭……”,证明学生类的槽函数被成功地触发了,如图 3-34 所示。在本案例中老师类和学生类的相关代码如下(其余代码读者可参考源码工程):

```
//chapter3/MySignalSlotsDemo/teacher.h
////teacher.h////
#ifndef TEACHER_H
#define TEACHER_H
```

```
#include <QObject>

class Teacher : public QObject
{
    Q_OBJECT
public:
    explicit Teacher(QObject *parent = nullptr);

signals:
    //自定义信号,写到 signals 下
    //返回值为 void,只用声明,不需要实现
    //可以有参数,也可以重载
    void finishClass();

public slots:

};
#endif //TEACHER_H


////teacher.cpp////
#include "teacher.h"
Teacher::Teacher(QObject *parent) : QObject(parent)
{

}


////student.h////
#ifndef STUDENT_H
#define STUDENT_H
#include <QObject>

class Student : public QObject
{
    Q_OBJECT
public:
    explicit Student(QObject *parent = nullptr);

signals:

public slots:
//早期 Qt 版本需要写到 public slots 下,高级版本可以写到 public 或全局下
//返回值为 void,需要声明,也需要实现
//可以有参数,也可以重载
    void gotoEat();
};
#endif //STUDENT_H
```

```
////student.cpp////
#include "student.h"
#include <QDebug>
Student::Student(QObject *parent) : QObject(parent)
{
}

void Student::gotoEat()
{
    qDebug() << "准备去吃饭.....";
}
```

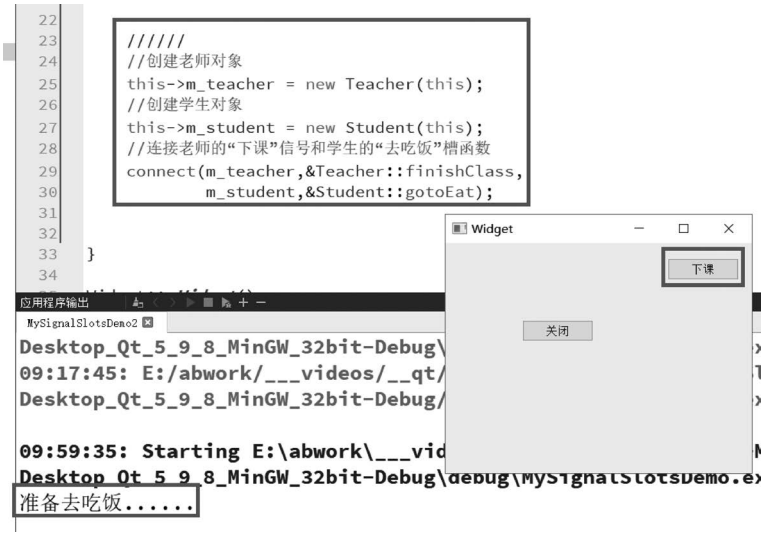

图 3-34　Qt 项目中自定义信号槽的应用

3.4.8　Qt 显示图像

Qt 可显示基本的图像类型，利用 QImage、QPxmap 类可以实现图像的显示，并且利用类中的方法可以实现图像的基本操作（缩放、旋转等）。Qt 可以直接读取并显示的格式有 BMP、GIF、JPG、JPEG、PNG、TIFF、PBM、PGM、PPM、XBM 和 XPM 等。可以使用 QLabel 显示图像，QLabel 类有 setPixmap()函数，可以用来显示图像。也可以直接用 QPainter 画出图像。如果图像过大，则可直接用 QLabel 显示，此时将会出现有部分图像显示不出来，可以用 Scroll Area 部件解决此问题。

首先使用 QFileDialog 类的静态函数 getOpenFileName()打开一张图像，将图像文件加载进 QImage 对象中，再用 QPixmap 对象获得图像，最后用 QLabel 选择一个 QPixmap 图像对象进行显示。该过程的关键代码如下（完整代码可参考 chapter3/QtImageDemo 工程）：

```
//chapter3/QtImageDemo/widget.cpp
//Qt 显示图片
void Widget::on_btnShowImage_clicked()
{
   QString filename;
   filename = QFileDialog::getOpenFileName(this,
tr("选择图像"),"",tr("Images ( * .png * .bmp * .jpg * .tif * .gif )"));
   if(filename.isEmpty()){
        return;
   }
   else{
      m_img = new QImage;
      if(! ( m_img->load(filename) ) )     //加载图像
      {
         QMessageBox::information(this,
                  tr("打开图像失败"),tr("打开图像失败!"));
         delete m_img;
         return;
      }
      ui->lblImage->setPixmap(QPixmap::fromImage( * m_img));
   }
}
```

QImage 对图像的像素级访问进行了优化，QPixmap 使用底层平台的绘制系统进行绘制，无法提供像素级别的操作，而 QImage 则使用了独立于硬件的绘制系统。编译并运行该工程，单击“显示图像”按钮，选择一张本地的图片，如图 3-35 所示。

图 3-35　Qt 使用 QImage 和 QPixmap 显示图像

3.4.9　Qt 缩放图像

Qt 缩放图像可以用 scaled 函数实现，代码如下：

```
//chapter3/qt-help-apis.txt
QImage QImage:: scaled (const QSize & size, Qt:: AspectRatioMode aspectRatioMode = Qt::
IgnoreAspectRatio, Qt::TransformationModetransformMode = Qt::FastTransformation) const;
```

利用上面已经加载成功的图像(m_img),在 scaled 函数中 width 和 height 表示缩放后图像的宽和高,即将原图像缩放到(width×height)大小。例如在本案例中显示的图像原始宽和高为(200×200),缩放后修改为(100×100),编译并运行,如图 3-36 所示,代码如下:

```
//chapter3/QtImageDemo/widget.cpp
void Widget::on_btnScale_clicked(){
    QImage * imgScaled = new QImage;
    * imgScaled = m_img->scaled(100,100, Qt::KeepAspectRatio);
    ui->lblScale->setPixmap(QPixmap::fromImage( * imgScaled));
}
```

图 3-36　Qt 缩放图像

3.4.10　Qt 旋转图像

Qt 旋转图像可以用 QMatrix 类的 rotate 函数实现,代码如下:

```
//chapter3/QtImageDemo/widget.cpp
void Widget::on_btnRotate_clicked(){
    QImage * imgRotate = new QImage;
    QMatrix matrix;
    matrix.rotate(270);
    * imgRotate = m_img->transformed(matrix);
    ui->lblRotate->setPixmap(QPixmap::fromImage( * imgRotate));
}
```

编译并运行该项目，依次单击“显示图像”“缩放”和“旋转”按钮，效果如图 3-37 所示。

图 3-37　Qt 显示、缩放和旋转图像