Android平台音视频开发基础

相关参考资料：

```
http://soundfile.sapp.org/doc/WaveFormat/
```

音视频API回顾

开发Android上的音频应用，最常见的是使用MediaRecorder和MediaPlayer来实现音频的录制和播放，更基础点的会使用AudioRecord和AudioTrack来实现。用这两种方式已经能应对绝大部分的音频开发需求了。更底层的API，如NDK层的OpenSL ES则很少有人知道。

MediaRecorder和MediaPlayer一般用于音视频的硬解，兼容性差，但是省电，主要依赖于机器的性能。

https://developer.android.com/reference/android/media/MediaRecorder

基础API：一定要注意流程图（详见官网）

MediaRecorder:

```
 MediaRecorder recorder = new MediaRecorder();
 recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
 recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
 recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
 recorder.setOutputFile(PATH_NAME);
 recorder.prepare();
 recorder.start();   // Recording is now started
 ...
 recorder.stop();
 recorder.reset();   // You can reuse the object by going back to
setAudioSource() step
 recorder.release(); // Now the object cannot be reused
```

Initial：初始状态，当使用new()方法创建一个MediaRecorder对象或者调用了reset()方法时，该MediaRecorder对象处于Initial状态。在设定视频源或者音频源之后将转换为Initialized状态。另外，在除Released状态外的其它状态通过调用reset()方法都可以使MediaRecorder进入该状态。

Initialized：已初始化状态，可以通过在Initial状态调用setAudioSource()或setVideoSource()方法进入该状态。在这个状态可以通过setOutputFormat()方法设置输出格式，此时MediaRecorder转换为DataSourceConfigured状态。另外，通过reset()方法进入Initial状态。

DataSourceConfigured：数据源配置状态，这期间可以设定编码方式、输出文件、屏幕旋转、预览显示等等。可以在Initialized状态通过setOutputFormat()方法进入该状态。另外，可以通过reset()方法回到Initial状态，或者通过prepare()方法到达Prepared状态。

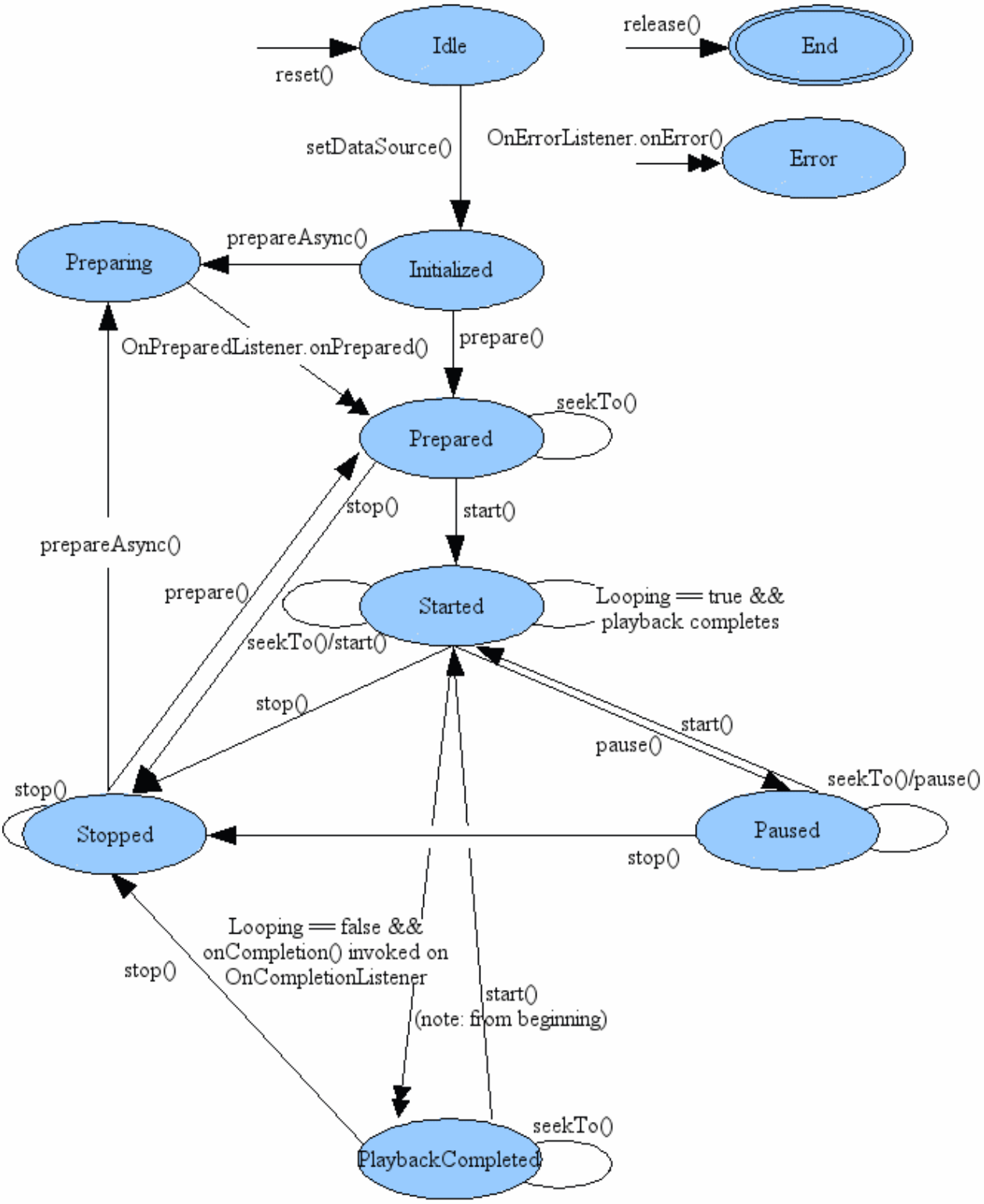
Prepared：就绪状态，在DataSourceConfigured状态通过prepare()方法进入该状态。在这个状态可以通过start()进入录制状态。另外，可以通过reset()方法回到Initialized状态。

Recording：录制状态，可以在Prepared状态通过调用start()方法进入该状态。另外，它可以通过stop()方法或reset()方法回到Initial状态。

Released：释放状态（官方文档给出的词叫做Idle state 空闲状态），可以通过在Initial状态调用release()方法来进入这个状态，这时将会释放所有和MediaRecorder对象绑定的资源。

Error：错误状态，当错误发生的时候进入这个状态，它可以通过reset()方法进入Initial状态。

MediaPlayer:



部分核心API:

```
private static MediaPlayer mMediaPlayer;
private static boolean isPause = false;

public static void playSound(String filePath) {
    playSound(filePath, null);
}

public static void playSound(String filePath,
MediaPlayer.OnCompletionListener onCompletionListener) {
```

```
    if (mMediaPlayer == null) {
        mMediaPlayer = new MediaPlayer();
        mMediaPlayer.setOnErrorListener(new MediaPlayer.OnErrorListener() {
            @Override
            public boolean onError(MediaPlayer mp, int what, int extra) {
                mMediaPlayer.reset();
                return false;
            }
        });
    } else {
        mMediaPlayer.reset();
    }
    try {
        mMediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
        if (onCompletionListener != null) {
            mMediaPlayer.setOnCompletionListener(onCompletionListener);
        }


        mMediaPlayer.setDataSource(filePath);
        mMediaPlayer.prepare();
        mMediaPlayer.start();
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (SecurityException e) {
        e.printStackTrace();
    } catch (IllegalStateException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }


}

/**
 * 暂停播放
 */
public static void pause() {
    if (mMediaPlayer != null && mMediaPlayer.isPlaying()) {
        mMediaPlayer.pause();
        isPause = true;
    }
}


/**
 * 继续播放
 */
public static void resume() {
    if (mMediaPlayer != null && isPause) {
        mMediaPlayer.start();
```

```
            isPause = false;
        }
    }

    /**
     * 释放资源
     */
    public static void realese() {
        if (mMediaPlayer != null) {
            mMediaPlayer.release();
            mMediaPlayer = null;
            isPause = true;
        }
    }
```

作业：完成上述api的实战回顾。

音频中的一些重要概念：

麦克风降噪

由于顶部和底部在通话时和音源(发出声音的源头)的距离不同，所以两个麦克风拾取的音量大小也是有不同的，利用这个差别，我们就可以过滤掉噪声保留人声了。在打电话时，两个麦克风所拾取的背景噪声音量是基本相同的，而记录的人声会有6dB左右的音量差。顶端麦收集噪声后，通过解码生成补偿信号后就可以用来消除噪音了。

回声

当声投射到距离声源有一段距离的大面积上时，声能的一部分被吸收，而另一部分声能要反射回来，如果听者听到由声源直接发来的声和由反射回来的声的时间间隔超过十分之一秒（在15°C空气中，距声源至少17米处反射），它就能分辨出两个声音这种反射回来的声叫“回声”。如果声速已知，当测得声音从发出到反射回来的时间间隔，就能计算出反射面到声源之间的距离。例如在室温（20°C）时空气中的声速是343米每秒，所以站在声源处的人要听到回声需要障碍物到声源的距离至少17米。

回声消除

很多时候在收集语音的同时也会播放着声音，这就要在语音收集的时候就需要对采集的声音进行回声消除。当在语音收集的时候，如果还同时播放着声音的话，就不能保证采集的声音不包括正在播放的声音，在这种情况下采集的语音即包括原声又包括回声，在这样的恶性循环下，就会使得回声越来越多，最后出现嗡鸣声。回声消除就是在麦克风录制外音的时候去除掉手机自身播放出来的声音，这样就将播放的声音从采集的声音中过滤出去，从而就避免了回声的产生。

关于声音的采集

麦克风的原理是将将采集的声音转换为模拟电信号，之后将模拟电信号数字化，也就是用高低电平表示的信号(计算机能识别的信号),在Android中有一个AudioRecord类就能录制语音，并将语音转换为PCM数据，声音在经过麦克风转换为模拟电信号并最终又转换为PCM数据，在转换为PCM数据时就要依赖于三个参数，分别是：声道数、采样位数和采样频率。

声道数：有单声道和立体声之分，单声道的声音只能使用一个喇叭发声（有的也处理成两个喇叭输出同一个声道的声音），立体声的PCM可以使两个喇叭都发声（一般左右声道有分工），更能感受到空间效果。

采样位数：即采样值或取样值（就是将采样样本幅度量化）。它是用来衡量声音波动变化的一个参数，也可以说是声卡的分辨率。它的数值越大，分辨率也就越高，所发出声音的能力越强。在计算机中采样位数一般有8位和16位之分，但有一点请大家注意，8位不是说把纵坐标分成8份，而是分成2的8次方即256份； 同理16位是把纵坐标分成2的16次方65536份。

采样频率：即取样频率，指每秒钟取得声音样本的次数。采样频率越高，声音的质量也就越好，声音的还原也就越真实，但同时它占的资源比较多。由于人耳的分辨率很有限，太高的频率并不能分辨出来。在16位声卡中有22KHz、44KHz等几级，其中，22KHz相当于普通FM广播的音质，44KHz已相当于CD音质了，目前的常用采样频率都不超过48KHz。

总结：一帧音频的数据量=pcm文件存储量= (采样频率 · 采样位数 · 声道 · 时间) / 8 (单位：字节数)。

音频重采样：

音频重采样是指这样的一个过程——把一个采样率的数据转换为另一个采样率的数据。Android 原生系统上，音频硬件设备一般都工作在一个固定的采样率上（如 48 KHz），因此所有音轨数据都需要重采样到这个固定的采样率上，然后再输出。为什么这么做？系统中可能存在多个音轨同时播放，而每个音轨的采样率可能是不一致的；比如在播放音乐的过程中，来了一个提示音，这时需要把音乐和提示音混音并输出到硬件设备，而音乐的采样率和提示音的采样率不一致，问题来了，如果硬件设备工作的采样率设置为音乐的采样率的话，那么提示音就会失真；因此最简单见效的解决方法是：硬件设备工作的采样率固定一个值，所有音轨在 AudioFlinger 都重采样到这个采样率上，混音后输出到硬件设备，保证所有音轨听起来都不失真。

再来回看AudioRecord这个类的使用技巧：

```
public AudioRecord (
int audioSource, // 音频源
int sampleRateInHz, // 采样率
int channelConfig,  // 声道
int audioFormat, // 编码制式
int bufferSizeInBytes) // 采集数据缓冲区大小

int bufferSizeInBytes = AudioRecord.getMinBufferSize(frequency,
channelConfiguration, EncodingBitRate); // 采样率 // 声道  // 编码制式

//注意：如果buffer容量过小，将导致对象构造的失败。

audioSource 去源码里面找

/**默认声音**/
public static final int DEFAULT = 0;
/**麦克风声音*/
public static final int MIC = 1
/**通话上行声音*/
public static final int VOICE_UPLINK = 2;
```

```
/**通话下行声音*/
public static final int VOICE_DOWNLINK = 3;
/**通话上下行声音*/
public static final int VOICE_CALL = 4;
/**根据摄像头转向选择麦克风*/
public static final int CAMCORDER = 5;
/**对麦克风声音进行声音识别，然后进行录制*/
public static final int VOICE_RECOGNITION = 6;
/**对麦克风中类似ip通话的交流声音进行识别，默认会开启回声消除和自动增益*/
public static final int VOICE_COMMUNICATION = 7;
/**录制系统内置声音*/
public static final int REMOTE_SUBMIX = 8;

//android支持双声道立体声和单声道。MONO单声道，STEREO立体声  AudioFormat这里定义了声道和编码
public static final int CHANNEL_IN_MONO = CHANNEL_IN_FRONT;
public static final int CHANNEL_IN_STEREO = (CHANNEL_IN_LEFT |
CHANNEL_IN_RIGHT);

public static final int ENCODING_PCM_16BIT = 2;
public static final int ENCODING_PCM_8BIT = 3;
```

AudioTrack:

播放声音可以使用 MediaPlayer 和 AudioTrack，两者都提供 Java API 给应用开发者使用。两者的差别在于：MediaPlayer 可以播放多种格式的音源，如 mp3、flac、wma、ogg、wav 等，而 AudioTrack 只能播放解码后的 PCM 数据流。MediaPlayer 在 Native 层会创建对应的音频解码器和一个 AudioTrack，解码后的数据交由 AudioTrack 输出。在性能为王的今天尤其是对声音时延要求非常苛刻的应用场景才需要用到 AudioTrack。

看Google给我们的资料啊：frameworks/base/media/tests/audiotests/shared_mem_test.cpp：

```
int AudioTrackTest::Test01() {

    sp<MemoryDealer> heap;
    sp<IMemory> iMem;
    uint8_t* p;

    short smpBuf[BUF_SZ];
    long rate = 44100;
    unsigned long phi;
    unsigned long dPhi;
    long amplitude;
    long freq = 1237;
    float f0;

    f0 = pow(2., 32.) * freq / (float)rate;
    dPhi = (unsigned long)f0;
    amplitude = 1000;
    phi = 0;
    Generate(smpBuf, BUF_SZ, amplitude, phi, dPhi);  // fill buffer
```

```
    for (int i = 0; i < 1024; i++) {
        // 分配一块匿名共享内存
        heap = new MemoryDealer(1024*1024, "AudioTrack Heap Base");

        iMem = heap->allocate(BUF_SZ*sizeof(short));

        // 把音频数据拷贝到这块匿名共享内存上
        p = static_cast<uint8_t*>(iMem->pointer());
        memcpy(p, smpBuf, BUF_SZ*sizeof(short));

        // 构造一个 AudioTrack 实例，该 AudioTrack 的数据方式是 MODE_STATIC
        // 音频数据已经一次性拷贝到共享内存上了，不用再调用 track->write() 填充数据了
        sp<AudioTrack> track = new AudioTrack(AUDIO_STREAM_MUSIC,// stream
type
                rate,
                AUDIO_FORMAT_PCM_16_BIT,// word length, PCM
                AUDIO_CHANNEL_OUT_MONO,
                iMem);

        // 检查 AudioTrack 实例是否构造成功饿了
        status_t status = track->initCheck();
        if(status != NO_ERROR) {
            track.clear();
            ALOGD("Failed for initCheck()");
            return -1;
        }

        // start play
        ALOGD("start");
        track->start(); // 开始播放

        usleep(20000);

        ALOGD("stop");
        track->stop(); // 停止播放
        iMem.clear();
        heap.clear();
        usleep(20000);
    }

    return 0;
}
```

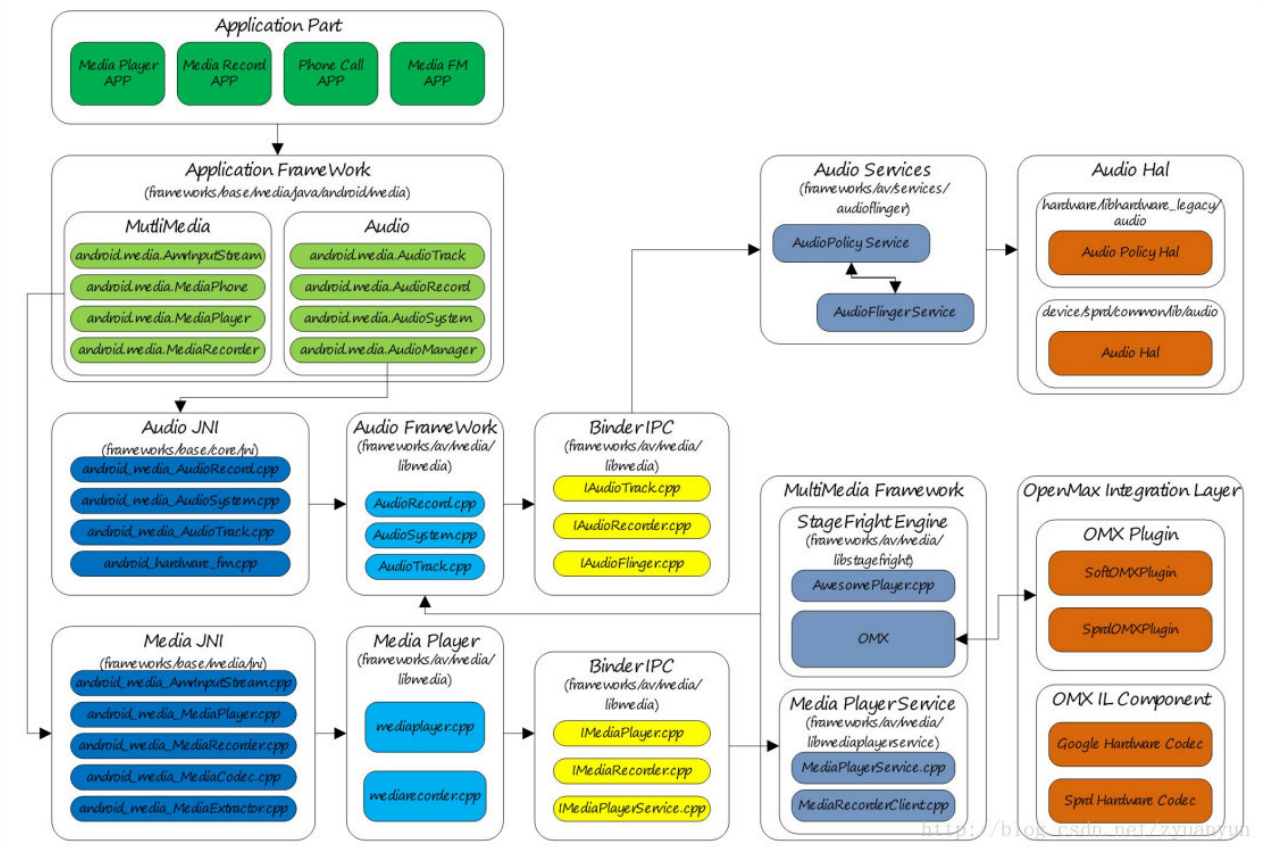
```
//Test case 1: setStereoVolume() with max volume returns SUCCESS
    @LargeTest
    public void testSetStereoVolumeMax() throws Exception {
        // constants for test
        final String TEST_NAME = "testSetStereoVolumeMax";
        final int TEST_SR = 22050;
        final int TEST_CONF = AudioFormat.CHANNEL_OUT_STEREO;
```

```
        final int TEST_FORMAT = AudioFormat.ENCODING_PCM_16BIT;
        final int TEST_MODE = AudioTrack.MODE_STREAM;
        final int TEST_STREAM_TYPE = AudioManager.STREAM_MUSIC;

        //-------- initialization --------------
        //  getMinBufferSize
        int minBuffSize = AudioTrack.getMinBufferSize(TEST_SR, TEST_CONF,
TEST_FORMAT);
        // 创建一个 AudioTrack 实例
        AudioTrack track = new AudioTrack(TEST_STREAM_TYPE, TEST_SR,
TEST_CONF, TEST_FORMAT,
                minBuffSize, TEST_MODE);
        byte data[] = new byte[minBuffSize/2];
        //--------    test        --------------
        // 调用 write() 写入回放数据
        track.write(data, 0, data.length);
        track.write(data, 0, data.length);
        // 调用 play() 开始播放
        track.play();
        float maxVol = AudioTrack.getMaxVolume();
        assertTrue(TEST_NAME, track.setStereoVolume(maxVol, maxVol) ==
AudioTrack.SUCCESS);
        // 播放完成后，调用 release() 释放 AudioTrack 实例
        track.release();
    }
```

AudioTrack Native API 音频流类型：

```
====Stream Type Description=======
AUDIO_STREAM_VOICE_CALL 电话语音
AUDIO_STREAM_SYSTEM 系统声音
AUDIO_STREAM_RING   铃声声音，如来电铃声、闹钟铃声等
AUDIO_STREAM_MUSIC  音乐声音
AUDIO_STREAM_ALARM  警告音
AUDIO_STREAM_NOTIFICATION   通知音
AUDIO_STREAM_DTMF   DTMF 音（拨号盘按键音）
```

抖音70k 前辈赠送的：Android音频架构图



OPENSL ES

官方资料：

```
https://www.khronos.org/registry/OpenSL-
ES/specs/OpenSL_ES_Specification_1.1.pdf 编程规范
https://www.khronos.org/opensles/
https://developer.android.com/ndk/guides/audio/opensl-for-android?hl=zh-cn
https://developer.android.com/ndk/guides/audio/opensl-prog-notes?hl=zh-cn
https://github.com/googlesamples/android-ndk/tree/android-mk  代码
https://www.khronos.org/registry/OpenSL-ES/api/1.0.1/OpenSLES.h 头文件
```

OpenSL ES™ is a royalty-free, cross-platform, hardware-accelerated audio API tuned for embedded systems. It provides a standardized, high-performance, low-latency method to access audio functionality for developers of native applications on embedded mobile multimedia devices, enabling straightforward cross-platform deployment of hardware and software audio capabilities, reducing implementation effort, and promoting the market for advanced audio.

一定要学习方法：头文件—Android的实现函数----定义—实现编码的规范—看ndk的官方demo---自己项目----总结

实战1:搞懂opensl es播放/录制/调频音频的套路

参考资料：http://www.10tiao.com/html/223/201612/2651232298/1.html

OpenSL ES的开发流程

1、创建接口对象(createEngine)

2、设置混音器（效果）（CreateOutputMix）

3、创建播放器（录音器）（CreateAudioPlayer）

4、设置缓冲队列和回调函数（RegisterCallback）

5、设置播放状态（SetPlayState）

6、启动回调函数（）

其中4和6是播放PCM等数据格式的音频是需要用到的。

注意OpenSL ES中最重要的接口类SLObjectItf，通过SLObjectItf接口类我们可以创建所需要的各种类型的类接口，比如：

- 创建引擎接口对象：SLObjectItf engineObject
- 创建混音器接口对象：SLObjectItf outputMixObject
- 创建播放器接口对象：SLObjectItf playerObject

OpenSL ES 有两个必须理解的概念，就是 Object 和 Interface，Object 可以想象成 Java 的 Object 类，Interface 可以想象成 Java 的 Interface，但它们并不完全相同，下面进一步解释他们的关系：

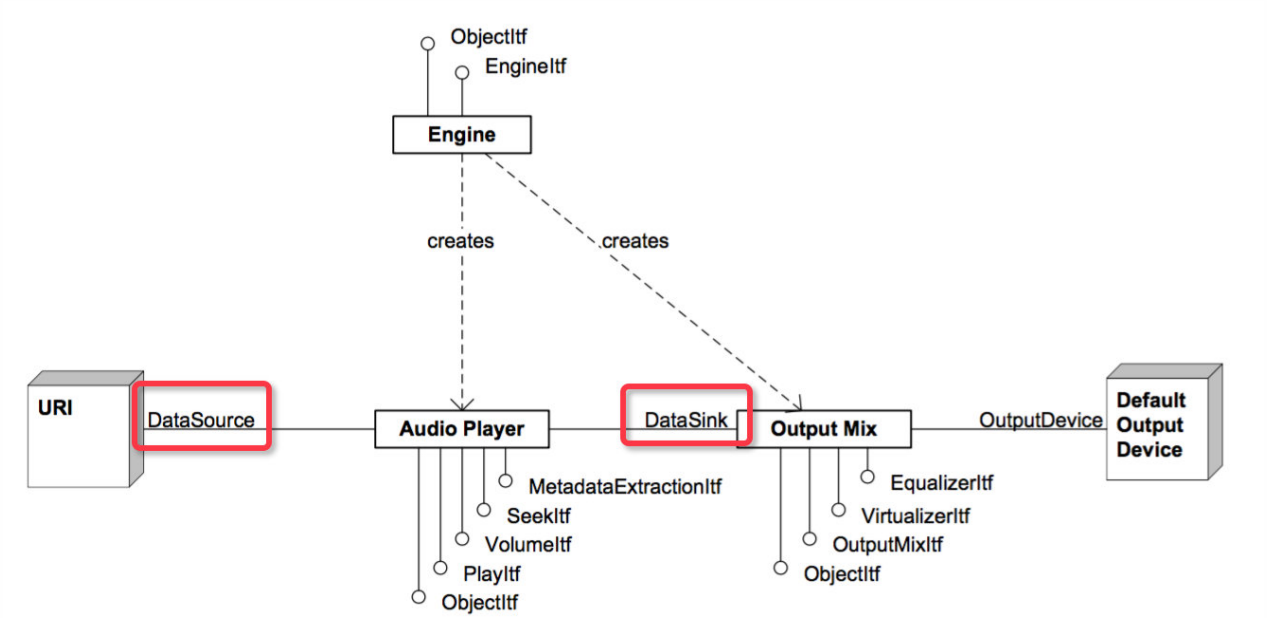
- 每个 Object 可能会存在一个或者多个 Interface，官方为每一种 Object 都定义了一系列的 Interface。
- 每个 Object 对象都提供了一些最基础的操作，比如：Realize，Resume，GetState，Destroy 等等，如果希望使用该对象支持的功能函数，则必须通过其 GetInterface 函数拿到 Interface 接口，然后通过 Interface 来访问功能函数。
- 并不是每个系统上都实现了 OpenSL ES 为 Object 定义的所有 Interface，所以在获取 Interface 的时候需要做一些选择和判断。

通过查看 “OpenSLES.h” 文件，我们可以看到 OpenSL ES 定义的所有 Object 对象的 ID，我们可以通过 Object ID 来创建对应的对象实例。

```
/* Objects ID's */

#define SL_OBJECTID_ENGINE          ((SLuint32) 0x00001001)
#define SL_OBJECTID_LEDDEVICE       ((SLuint32) 0x00001002)
#define SL_OBJECTID_VIBRADEVICE     ((SLuint32) 0x00001003)
#define SL_OBJECTID_AUDIOPLAYER     ((SLuint32) 0x00001004)
#define SL_OBJECTID_AUDIORECORDER   ((SLuint32) 0x00001005)
#define SL_OBJECTID_MIDIPLAYER      ((SLuint32) 0x00001006)
#define SL_OBJECTID_LISTENER        ((SLuint32) 0x00001007)
#define SL_OBJECTID_3DGROUP         ((SLuint32) 0x00001008)
#define SL_OBJECTID_OUTPUTMIX       ((SLuint32) 0x00001009)
#define SL_OBJECTID_METADATAEXTRACTOR   ((SLuint32) 0x0000100A)
```

同样，“OpenSLES.h” 文件中还定义了所有的 Interface ID，通过 Interface ID 我们可以从对象中获取到对应的功能接口。

```
extern SL_API const SLInterfaceID SL_IID_NULL;
extern SL_API const SLInterfaceID SL_IID_OBJECT;
extern SL_API const SLInterfaceID SL_IID_AUDIOIODEVICECAPABILITIES;
extern SL_API const SLInterfaceID SL_IID_LED;
extern SL_API const SLInterfaceID SL_IID_VIBRA;
extern SL_API const SLInterfaceID SL_IID_METADATAEXTRACTION;
extern SL_API const SLInterfaceID SL_IID_METADATATRAVERSAL;
extern SL_API const SLInterfaceID SL_IID_DYNAMICSOURCE;
extern SL_API const SLInterfaceID SL_IID_OUTPUTMIX;
extern SL_API const SLInterfaceID SL_IID_PLAY;
extern SL_API const SLInterfaceID SL_IID_PREFETCHSTATUS;
extern SL_API const SLInterfaceID SL_IID_PLAYBACKRATE;
extern SL_API const SLInterfaceID SL_IID_SEEK;
extern SL_API const SLInterfaceID SL_IID_RECORD;
extern SL_API const SLInterfaceID SL_IID_EQUALIZER;
extern SL_API const SLInterfaceID SL_IID_VOLUME;
extern SL_API const SLInterfaceID SL_IID_DEVICEVOLUME;
extern SL_API const SLInterfaceID SL_IID_BUFFERQUEUE;
extern SL_API const SLInterfaceID SL_IID_PRESETREVERB;
extern SL_API const SLInterfaceID SL_IID_ENVIRONMENTALREVERB;
extern SL_API const SLInterfaceID SL_IID_EFFECTSEND;
extern SL_API const SLInterfaceID SL_IID_3DGROUPING;
extern SL_API const SLInterfaceID SL_IID_3DCOMMIT;
extern SL_API const SLInterfaceID SL_IID_3DLOCATION;
extern SL_API const SLInterfaceID SL_IID_3DDOPPLER;
extern SL_API const SLInterfaceID SL_IID_3DSOURCE;
extern SL_API const SLInterfaceID SL_IID_3DMACROSCOPIC;
extern SL_API const SLInterfaceID SL_IID_MUTESOLO;
extern SL_API const SLInterfaceID SL_IID_DYNAMICINTERFACEMANAGEMENT;
extern SL_API const SLInterfaceID SL_IID_MIDIMESSAGE;
extern SL_API const SLInterfaceID SL_IID_MIDIMUTESOLO;
extern SL_API const SLInterfaceID SL_IID_MIDITEMPO;
extern SL_API const SLInterfaceID SL_IID_MIDITIME;
extern SL_API const SLInterfaceID SL_IID_AUDIODECODERCAPABILITIES;
extern SL_API const SLInterfaceID SL_IID_AUDIOENCODERCAPABILITIES;
extern SL_API const SLInterfaceID SL_IID_AUDIOENCODER;
extern SL_API const SLInterfaceID SL_IID_BASSBOOST;
extern SL_API const SLInterfaceID SL_IID_PITCH;
extern SL_API const SLInterfaceID SL_IID_RATEPITCH;
extern SL_API const SLInterfaceID SL_IID_VIRTUALIZER;
extern SL_API const SLInterfaceID SL_IID_VISUALIZATION;
extern SL_API const SLInterfaceID SL_IID_ENGINE;
extern SL_API const SLInterfaceID SL_IID_ENGINECAPABILITIES;
extern SL_API const SLInterfaceID SL_IID_THREADSYNC;
```



上面的这个架构图一定要记住，针对播放的。

OpenSL ES 里面，这两个结构体均是作为创建 Media Object 对象时的参数而存在的，data source 代表着输入源的信息，即数据从哪儿来、输入的数据参数是怎样的；而 data sink 则代表着输出的信息，即数据输出到哪儿、以什么样的参数来输出。

Data Source 的定义如下：

```
typedef struct SLDataSource_ {
      void *pLocator;
      void *pFormat;
} SLDataSource;
```

Data Sink 的定义如下：

```
typedef struct SLDataSink_ {
    void *pLocator;
    void *pFormat;
} SLDataSink;
```

其中，pLocator 主要有如下几种：

```
SLDataLocator_Address
SLDataLocator_BufferQueue
SLDataLocator_IODevice
SLDataLocator_MIDIBufferQueue
SLDataLocator_URI
```

也就是说，Media Object 对象的输入源/输出源，既可以是 URL，也可以 Device，或者来自于缓冲区队列等等，完全是由 Media Object 对象的具体类型和应用场景来配置。

实战：使用opensl es播放pcm文件。

总结：OpenSL ES 是基于 c 语言实现的，但其提供的接口是采用面向对象的方式实现，OpenSL ES 的大多数 API 是通过对象来调用的。

```
Lresult result;
// realize the engine
result = (*engineObject)->Realize(engineObject, SL_BOOLEAN_FALSE);
assert(SL_RESULT_SUCCESS == result);
(void)result;
result = (*engineObject)->GetInterface(engineObject, SL_IID_ENGINE,
&engineEngine);
assert(SL_RESULT_SUCCESS == result);
(void)result;
```

关于对象和接口的概念：

Object 和 Interface 两大基本的概念，是很重要的。一定要记住：通过对象拿接口

对象的生命周期：

SL_OBJECT_STATE_UNREALIZED （不可用），

SL_OBJECT_STATE_REALIZED（可用），

SL_OBJECT_STATE_SUSPENDED（挂起）

经典的Object和Interface：

```
Audio 引擎对象和接口：

SLresult result;
// 创建引擎对象
result = slCreateEngine(&engineObject, 0, NULL, 0, NULL, NULL);
assert(SL_RESULT_SUCCESS == result);
(void)result;
// 实例化
result = (*engineObject)->Realize(engineObject, SL_BOOLEAN_FALSE);
assert(SL_RESULT_SUCCESS == result);
(void)result;
// 获取引擎对象接口
result = (*engineObject)->GetInterface(engineObject, SL_IID_ENGINE,
&engineEngine);
assert(SL_RESULT_SUCCESS == result);
(void)result;
// 释放引擎对象的资源
result = (*engineObject)->Destroy(engineObject, SL_BOOLEAN_FALSE);
assert(SL_RESULT_SUCCESS == result);
(void)result;

SLRecordItf 和 SLPlayItf:多媒体功能 recorder 和 player

// 创建 audio recorder 对象
```

```
result = (*engineEngine)->CreateAudioRecorder(engineEngine, &recorderObject ,
&recSource, &dataSink,

 NUM_RECORDER_EXPLICIT_INTERFACES, iids, required);
// 创建 audio player 对象
SLresult result = (*engineEngine)->CreateAudioPlayer(
        engineEngine,
        &audioPlayerObject,
        &dataSource,
        &dataSink,
        1,
        interfaceIDs,
        requiredInterfaces
);

SLDataSource 和 SLDataSink:构建 audio player 和 recorder 对象,输入和输出
// 数据源简单缓冲队列定位器
SLDataLocator_AndroidSimpleBufferQueue dataSou
        SL_DATALOCATOR_ANDROIDSIMPLEBUFFERQUEU
        1
};
// PCM 数据源格式
SLDataFormat_PCM dataSourceFormat = {
        SL_DATAFORMAT_PCM, // 格式类型
        wav_get_channels(wav), // 通道数
        wav_get_rate(wav) * 1000, //采样率
        wav_get_bits(wav), // 位宽
        wav_get_bits(wav),
        SL_SPEAKER_FRONT_CENTER, // 通道屏蔽
        SL_BYTEORDER_LITTLEENDIAN // 字节顺序
};
// 数据源
SLDataSource dataSource = {
        &dataSourceLocator,
        &dataSourceFormat
};
// 针对数据接收器的输出混合定位器(混音器)
SLDataLocator_OutputMix dataSinkLocator = {
        SL_DATALOCATOR_OUTPUTMIX, // 定位器类型
        outputMixObject // 输出混合
};
// 输出
SLDataSink dataSink = {
        &dataSinkLocator, // 定位器
        0,
};
```

camera2

参考资料：

```
https://www.youtube.com/watch?v=Xtp3tH27OFs
https://www.youtube.com/watch?v=KhqGphh6KPE
https://github.com/florent37/CameraFragment （一般用开做sdk的开发 需要将UI和业务分离）
https://github.com/sucese/phoenix （国人）
https://github.com/googlesamples/android-Camera2Basic （googlesamples）
https://developer.android.com/reference/android/hardware/camera2/package-summary.html （document）
```

Android 5.0(21) 之后 android.hardware.Camera 就被废弃了，取而代之的是全新的android.hardware.Camera2。API不仅大幅提高了Android系统拍照的功能，还能支持RAW照片输出，甚至允许程序调整相机的对焦模式、曝光模式、快门等。

Camera2API总结：

CameraManager：摄像头管理器。这是一个全新的系统管理器，专门用于检测系统摄像头、打开系统摄像头。除此之外，调用CameraManager的getCameraCharacteristics(String)方法即可获取指定摄像头的相关特性。

CameraCharacteristics：摄像头特性。该对象通过CameraManager来获取，用于描述特定摄像头所支持的各种特性。

```
//否需要闪光灯支持
characteristics.get(CameraCharacteristics.FLASH_INFO_AVAILABLE);
//是否为前置摄像头
characteristics.get(CameraCharacteristics.LENS_FACING);
//获取图片输出的尺寸和预览画面输出的尺寸
characteristics.get(CameraCharacteristics.SCALER_STREAM_CONFIGURATION_MAP);
获取到这个map后可以获取到图片输出的尺寸和预览画面输出的尺寸

configurationMap.getOutputSizes(ImageFormat.JPEG);
configurationMap.getOutputSizes(SurfaceTexture.class)
```

CameraDevice：代表系统摄像头。该类的功能类似于早期的Camera类。

CameraCaptureSession：这是一个非常重要的API，当程序需要预览、拍照时，都需要先通过该类的实例创建Session。而且不管预览还是拍照，也都是由该对象的方法进行控制的，其中控制预览的方法为setRepeatingRequest()；控制拍照的方法为capture()。

```
mCameraDevice.createCaptureSession(Arrays.asList(surface,
mImageReader.getSurface()),
                    new CameraCaptureSession.StateCallback() {
                        @Override
                        public void onConfigured(@NonNull
CameraCaptureSession session) {
```

```
                            mCaptureSession = session;
                            //发送预览请求
                            sendRepeatPreviewRequest();

                        }

                        @Override
                        public void onConfigureFailed(@NonNull
CameraCaptureSession session) {

                        }
                    }, mCameraHandler);
//发送预览请求
private boolean sendRepeatPreviewRequest() {
        try {
            CaptureRequest.Builder builder =
mCameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW);
            builder.addTarget(mPreviewSurface);
            builder.set(CaptureRequest.CONTROL_MODE,
CameraMetadata.CONTROL_MODE_AUTO);
            builder.setTag(RequestTag.Preview);
            addBaselineCaptureKeysToRequest(builder);

            mCaptureSession.setRepeatingRequest(builder.build(),
                    mFocusStateListener,
                    mCameraHandler);
            return true;
        } catch (CameraAccessException e) {
            e.printStackTrace();
            return false;
        }
    }
```

CameraRequest和CameraRequest.Builder：当程序调用setRepeatingRequest()方法进行预览时，或调用capture()方法进行拍照时，都需要传入CameraRequest参数。CameraRequest代表了一次捕获请求，用于描述捕获图片的各种参数设置，比如对焦模式、曝光模式等。程序需要对照片所做的各种控制，都通过CameraRequest参数进行设置。CameraRequest.Builder则负责生成CameraRequest对象。

关于权限:

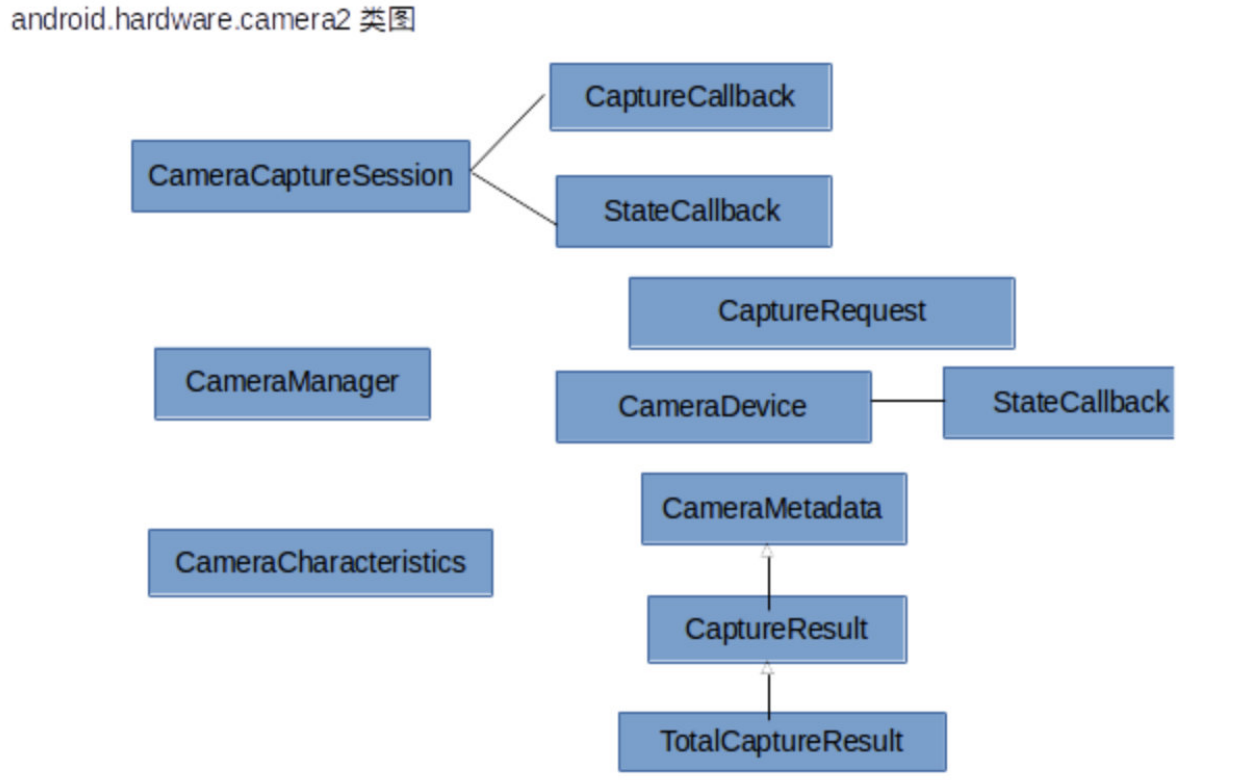
```
<uses-permission android:name="android.permission.CAMERA" />
<uses-feature android:name="android.hardware.camera" />
<uses-feature android:name="android.hardware.camera.autofocus" />
```

拍照流程总结:

https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/hardware/camera2/package.html

```
<BODY>
<p>The android.hardware.camera2 package provides an interface to
individual camera devices connected to an Android device. It replaces
the deprecated {@link android.hardware.Camera} class.</p>

<p>This package models a camera device as a pipeline, which takes in
input requests for capturing a single frame, captures the single image
per the request, and then outputs one capture result metadata packet,
plus a set of output image buffers for the request. The requests are
processed in-order, and multiple requests can be in flight at
once. Since the camera device is a pipeline with multiple stages,
having multiple requests in flight is required to maintain full
framerate on most Android devices.</p>

<p>To enumerate, query, and open available camera devices, obtain a
{@link android.hardware.camera2.CameraManager} instance.</p>

<p>Individual {@link android.hardware.camera2.CameraDevice
CameraDevices} provide a set of static property information that
describes the hardware device and the available settings and output
parameters for the device. This information is provided through the
{@link android.hardware.camera2.CameraCharacteristics} object, and is
available through {@link
android.hardware.camera2.CameraManager#getCameraCharacteristics}</p>

<p>To capture or stream images from a camera device, the application
must first create a {@link
android.hardware.camera2.CameraCaptureSession camera capture session}
with a set of output Surfaces for use with the camera device, with
{@link
android.hardware.camera2.CameraDevice#createCaptureSession}. Each
Surface has to be pre-configured with an {@link
android.hardware.camera2.params.StreamConfigurationMap appropriate
size and format} (if applicable) to match the sizes and formats
available from the camera device. A target Surface can be obtained
```

android.hardware.camera2 类图



实战：分析cameraview这个lib的源码

参考：https://github.com/google/cameraview （已经修改了多次，目前已经运行在线上）

自定义camera核心知识点总结

如何使用camera

```
1. 获取一个 Camera 实例，通过 open 方法，Camera.open(0),0 是后置摄像头，1 表示前置摄像头。
2. 设置 Camera 的参数，比如聚焦，是否开闪光灯，预览高宽，修改 Camera 的默认参数：mCamera.getParameters() 通过初始化 SurfaceHolder 去 setPreviewDisplay(SurfaceHolder)，没有 surface,Camera 不能开始预览。
3. 调用 startPreview 方法开始更新预览到 surface，在拍照之前，startPreview 必须调用，预览必须开启。
4. 当你想开始拍照时，使用 takePicture(Camera.ShutterCallback, Camera.PictureCallback, Camera.PictureCallback, Camera.PictureCallback)，等待回调提供真实的图像数据。当拍完一张照片时，预览（preview）将会停止，当你想要拍更多的照片时，须要再一次调用 startPreview 方法 。
5. 当调用 stopPreview 方法时，将停止更新预览的 surface。
6. 当调用 release 方法时，将马上释放 camera。
```

SurfaceView，TextureView，GlSurfaceView

SurfaceView

继承自View,拥有View的大部分属性，但是由于holder的存在，不能设置透明度。 优点：可以在一个独立的线程中进行绘制，不会影响主线程，使用双缓冲机制，播放视频时画面更流畅 缺点：surface的显示不受View属性的控制，不能将其放在ViewGroup中，SurfaceView不能嵌套使用。

GlSurfaceView

GlSurfaceView继承自SurfaceView类，专门用来显示OpenGL渲染的，简单理解可以显示视频，图像及3D场景这些的。

SurfaceTexture

和SurfaceView功能类似，区别是，SurfaceTexure可以不显示在界面中。使用OpenGl对图片流进行美化，添加水印，滤镜这些操作的时候我们都是通过SurfaceTexre去处理，处理完之后再通过GlSurfaceView显示。缺点，可能会导致个别帧的延迟。本身管理着BufferQueue,所以内存消耗会多一点。

TextureView

同样继承自View，必须在开启硬件加速的设备中使用（保守估计目前百分之九十的Android设备都开启了），TextureView通过setSurfaceTextureListener的回调在子线程中进行更新UI. 优点：支持动画效果。 缺点：在5.0之前在主线程渲染，在5.0之后在单独线程渲染。

	TextureView	SurfaceView
绘制	稍微延时	及时
内存	高	低
动画	支持	不支持
耗电	高	低
适用场景（推荐）	视频播放，相机应用	大量画布更新(游戏绘制)

实战思考：我们需要预览相机怎么做？surfaceview？OpenGl ES？我来GLSurfaceView做一次，你们就复习下其他几种。思考ui绘制性能等问题。

相继开发的总结

相机开发流程

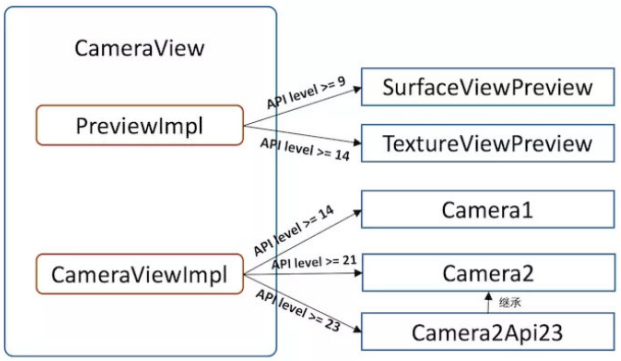
1. 检测并访问相机资源 检查手机是否存在相机资源，如果存在则请求访问相机资源。
2. 创建预览界面，创建继承自SurfaceView并实现SurfaceHolder接口的拍摄预览类。有了拍摄预览类，即可创建一个布局文件，将预览画面与设计好的用户界面控件融合在一起，实时显示相机的预览图像。
3. 设置拍照监听器，给用户界面控件绑定监听器，使其能响应用户操作，开始拍照过程。
4. 拍照并保存文件，将拍摄获得的图像转换成位图文件，最终输出保存成各种常用格式的图片。
5. 释放相机资源，相机是一个共享资源，当相机使用完毕后，必须正确地将其释放，以免其它程序访问使用时发生冲突。

相机开发注意问题

6. 版本兼容性问题，Android 5.0以下的Camera和Android 5.0以上使用Camera2，Android 4.0以下的SurfaceView和Android 4.0以上的TextureView，Android 6.0以上要做相机等运行时权限兼容。
7. 设备兼容性问题，Camera/Camera2里的各种特性在有些手机厂商的设备实现方式和支持程度是不一样的，这个需要做兼容性测试，一点点踩坑。
8. 各种场景下的生命周期变化问题，最常见的是后台场景和锁屏场景，这两种场景下的相机资源的申请与释放，Surface的创建与销毁会带来一些问题。

版本选择?

参考Google仓库给出的建议



表：Android系统版本、API Level及其对应的实现方案

System version	API level	Camera API	Preview View
2.3 - 3.2	9 - 13	Camera 1	SurfaceView
4.0 - 4.4	14 - 20	Camera 1	TextureView
5.0 - 5.1	21 - 22	Camera 2	TextureView
6.0 -	23 -	Camera 2	TextureView

opengl es

```
https://learnopengl.com/Getting-started/OpenGL
https://developer.android.com/guide/topics/graphics/opengl
```

这块的的确确很复杂，我在这里主要结合实战案例来指导你们，我不会把opengl的全部知识都说一遍，你们要抓到放小，知道怎么学习？

> OpenGL(Open Graphics Library开发图形接口)是一个跨平台的图形API，用于指定3D图形处理硬件中的标准软件接口。OpenGl一般用于在图形工作站，PC端使用，由于性能各方面原因，在移动端使用OpenGl基本带不动。为此，Khronos公司就为OpenGl提供了一个子集，OpenGl ES(OpenGl for Embedded System)。OpenGl ES是免费的跨平台的功能完善的2D/3D图形库接口的API,是OpenGL的一个子集。

Google专门为OpenGl提供了android.opengl包，并且提供了GlSurfaceView,GLU,GlUtils等工具类。

```
OpenGL ES 1.0 和 1.1 ：Android 1.0和更高的版本支持这个API规范。
OpenGL ES 2.0 ：Android 2.2(API 8)和更高的版本支持这个API规范。
OpenGL ES 3.0 ：Android 4.3(API 18)和更高的版本支持这个API规范。
OpenGL ES 3.1 : Android 5.0(API 21)和更高的版本支持这个API规范。
<!--提供给应用市场来解析的-->
<uses-feature android:glEsVersion="0x00020000" android:required="true"/>
```

GlSurfaceView.Renderer

继承至SurfaceView，它内嵌的surface专门负责OpenGL渲染。这个类主要有几个作用

- 管理Surface与EGL。
- 允许自定义渲染器(render)。
- 让渲染器在独立的线程里（GLThread）运作，和UI线程分离。
- 支持按需渲染(on-demand)和连续渲染(continuous)。
- onSurfaceCreated()：系统调用这个方法一次创建时GLSurfaceView。使用此方法来执行只需要发生一次的操作，比如设置OpenGL的环境参数或初始化的OpenGL图形对象。
- onDrawFrame()：系统调用上的每个重绘此方法GLSurfaceView。使用此方法作为主要执行点用于绘

制（和重新绘制）的图形对象。
- 系统调用onSurfaceChanged()时表示GLSurfaceView几何形状的变化，包括尺寸变化GLSurfaceView或设备屏幕的取向。例如，当设备从纵向变为横向的系统调用这个方法。使用此方法可以在变化做出反应GLSurfaceView容器。

EGL：OpenGL是一个跨平台的操作GPU的API，但OpenGL需要本地视窗系统进行交互，这就需要一个中间控制层， EGL就是连接OpenGL ES和本地窗口系统的接口，引入EGL就是为了屏蔽不同平台上的区别。

实战：opengl_hello project 看看怎么实现一个opengl图像。

opengl es关键知识

GLSL着色器语言

学不死资料：https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.1.20.pdf

实战：GlSurfaceView 实现照相机渲染。 Douyin project

搞OpenGL 需要下看懂搞大牛的资料在自己写

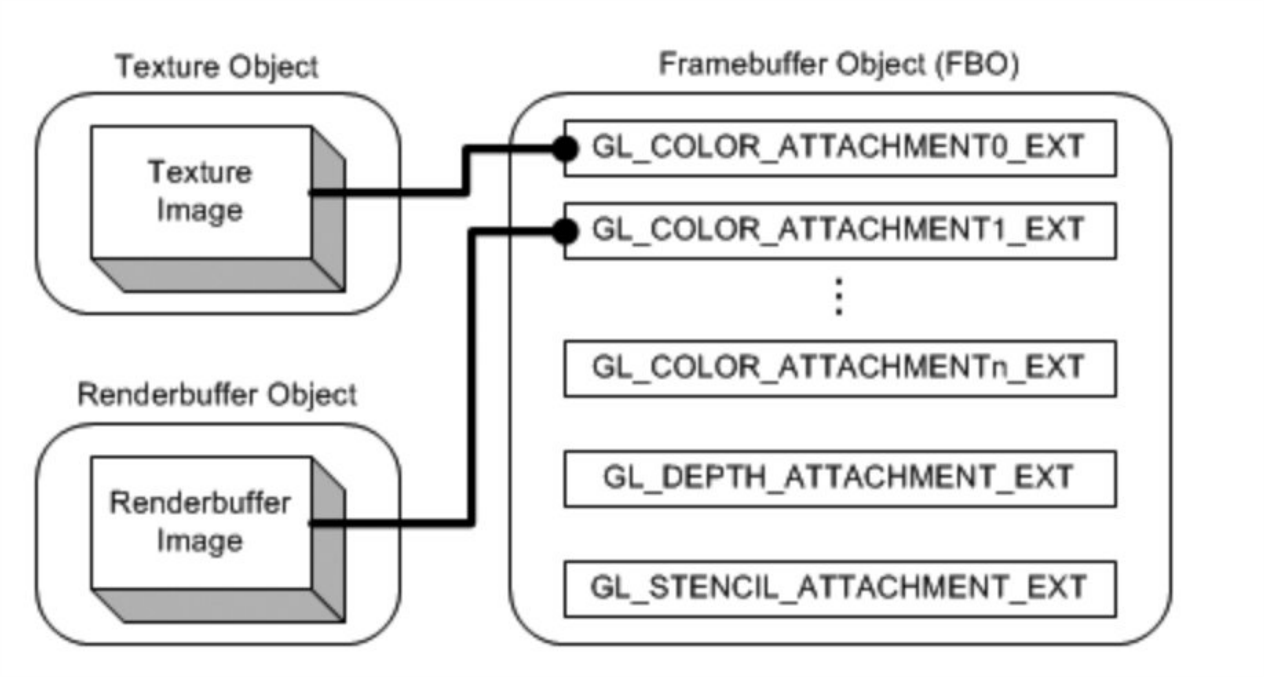
https://github.com/google/grafika/tree/master/app/src/main/java/com/android/grafika

> 在OpenGL中，任何事物都在3D空间中，而屏幕和窗口却是2D像素数组，这导致OpenGL的大部分工作都是关于把3D坐标转变为适应你屏幕的2D像素。3D坐标转为2D坐标的处理过程是由OpenGL的图形渲染管线（Graphics Pipeline，大多译为管线，实际上指的是一堆原始图形数据途经一个输送管道，期间经过各种变化处理最终出现在屏幕的过程）管理的。
>
> 图形渲染管线可以被划分为几个阶段，每个阶段将会把前一个阶段的输出作为输入。所有这些阶段都是高度专门化的（它们都有一个特定的函数），并且很容易并行执行。正是由于它们具有并行执行的特性，当今大多数显卡都有成千上万的小处理核心，它们在GPU上为每一个（渲染管线）阶段运行各自的小程序，从而在图形渲染管线中快速处理你的数据。这些小程序叫做着色器(Shader)。

FBO(帧缓冲)

FBO一个最常见的应用就是：渲染到纹理(render to texture)，通过这项技术可以实现发光效果，环境映射，阴影映射等很炫的效果。

OpenGL中的Frame Buffer Object(FBO)扩展，被推荐用于把数据渲染到纹理对像。相对于其它同类技术，如数据拷贝或交换缓冲区等，使用FBO技术会更高效并且更容易实现。在OpenGL渲染管线中，几何数据和纹理最终都是以2d像素绘制到屏幕上。最后一步的渲染目标在OpenGL渲染管线中被称为帧缓存(frame buffer)。帧缓存是颜色缓存、深度缓存、模板缓存、累积缓存的集合。默认情况下， OpenGL使用的帧缓存是由窗体系统创建和管理的。



应用场景：一般情况下，图像会被渲染到窗口上，但是有些情况需要多次渲染，最后才将最终的渲染图像输出到窗口，这时候就需要用到帧缓冲区对象。用帧缓冲区对象，可以先把图像渲染到纹理，然后操作这个纹理。