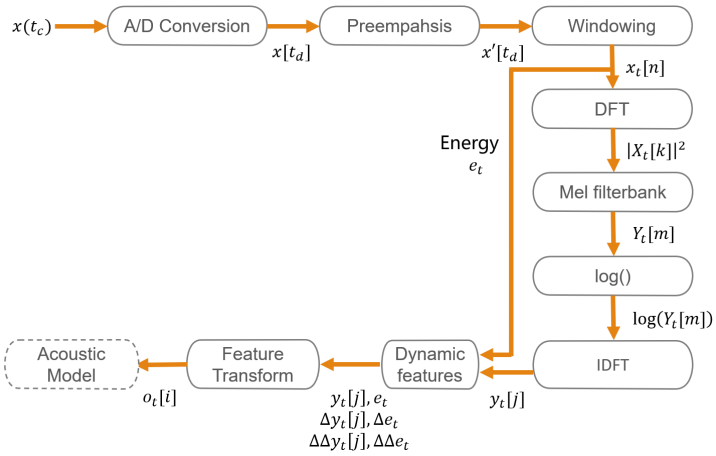
Speech Signal Analysis

- 学号：1952335
- 姓名：代玉琢

对于一段语音，原始语音是不定长的时序信号，不适合直接作为传统机器学习算法的输入，为了后续的分析，一般需要转换成特定的特征向量表示，这个过程称为语音特征提取。本次作业，我们最后需要得到的是一段语音的MFCC特征。

整个过程如下图所示。



语音信号特征提取示意图

一、A/D Conversion

一段语音文件可以被编码为多种格式，如PCM编码、MP3编码等。其中，PCM编码的最大优点是音质好，最大缺点是占用存储空间多。在数模转换这一步，我是将WAV文件读入，并获取其信息。

WAV文件的获取：使用Adobe Audition录制。选择 **采样率** 为44100Hz， **声道** 为单声道， **位深度** 为16位（否则python的wave库不能分析）。

选择单声道的原因：为了进行语音识别，需要将立体声道语音转换为单声道语音。

使用wave库读取wav文件，并提取简单的参数，如 **采样频率** 、**音频总帧数** 、**n帧采样点数据**等。

代码：

```
def wav_read(file_name):
```

```
    """
    读取wav文件并简单地提取参数
    :param file_name: 打开的wav文件名
    :return: data,time
    """
    wav_file = wave.open(file_name, "rb")
    num_channel = wav_file.getnchannels()  # 声道数
    sample_width = wav_file.getsampwidth()  # 采样字节长度
    framerate = wav_file.getframerate()  # 采样频率
    num_frames = wav_file.getnframes()  # 音频总帧数
    # print("声道数：", num_channel)
    # print("采样字节长度：", sample_width)
    # print("采样频率：", framerate)
    # print("音频总帧数：", num_frames)
    data = wav_file.readframes(num_frames)  # readframes(n):读取并返回以 bytes 对象表示的最多 n 帧音频
    data = np.frombuffer(data, dtype=np.int16)  # 将采样的点变为数组
    data = data.T  # 转置
    # print("采样的n帧音频：", data)
    time = np.arange(0, num_frames) * (1 / framerate)  # 时间
    # print("时间：", time)
    return data, time
```

二、Pre-emphasis（预处理）

语音经发音者的口唇辐射发出，受到唇端辐射抑制，高频能量明显降低。可以采用预加重（Pre-emphasis）的方法补偿语音的高频部分的振幅。

预加重的公式如下：

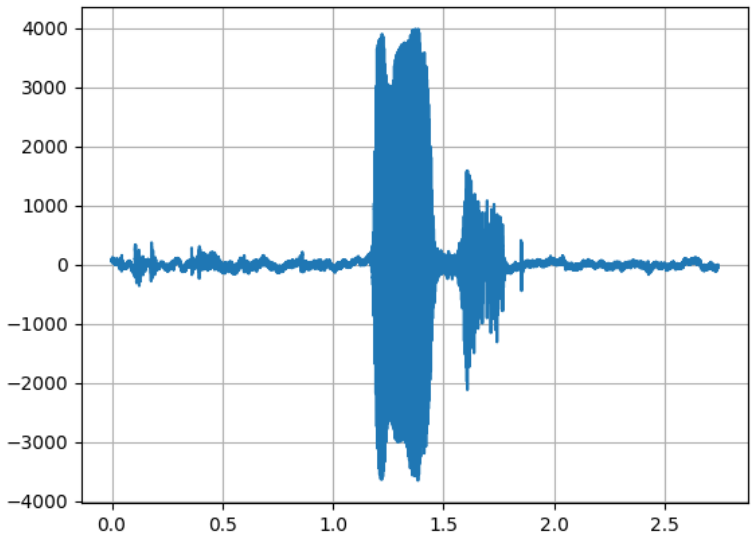
$$x_t'[d] = x_t[d] - \alpha\, x_t[d-1] \qquad 0.95 < \alpha < 0.99$$

在这里，α 取0.97。

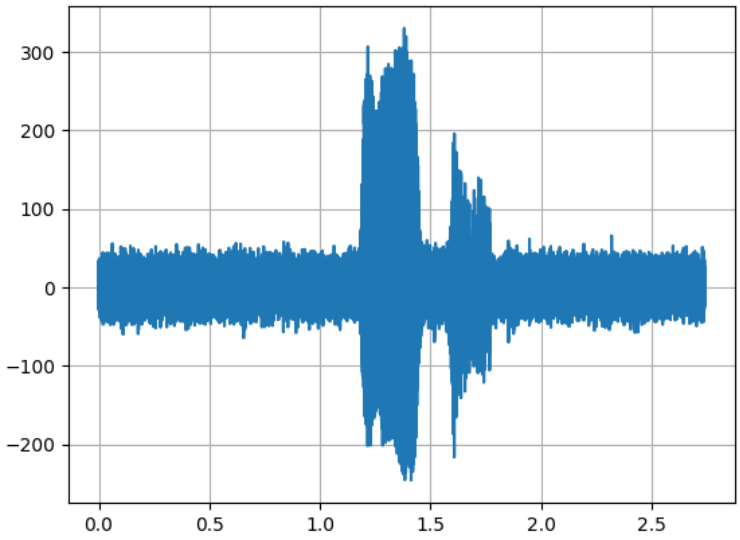
代码如下：

```
def pre_emphasis(data):
    """
    pre-emphasis 预加重
    :param data: 未处理的信号
    :return: signal--预加重后的信号
    """
    signal = []
    for i in range(1, len(data)):
        signal.append(data[i] - 0.97 * data[i - 1])
    signal = np.array(signal)  # 将数组转成NumPy数组
    # print("预加重后的采样点：", signal)
    return signal
```

未处理的信号：



预加重后的信号：



可以看出，整体图像的低频部分增加。

三、Windowing（分帧并加窗）

因为从整体上观察，语音信号是一个非平稳信号，但是考虑到发浊音时声带有规律地振动，即基音频率在短时范围内是相对固定的，所以可以认为语音信号具有短时平稳特性。一般认为10~30ms的语音信号片段是一个准稳态过程。

短时分析主要采用分帧方式，一般认为每帧帧长为20ms或25ms。因为相邻两帧之间的基音可能发生变化，所以为了保证声学特征参数的平滑性，一般采用重叠取帧的方式，即取一个帧移，一般为10ms。

我采用的是取**帧长为25ms**，**帧移为10ms**。

分帧方式相当于对语音信号进行了加矩形窗的处理。矩形窗虽然简单，但是会造成频谱泄露。所以需要为其加窗，一般是汉明窗、汉宁窗或布莱克曼窗。

窗权重函数：

$$w[n] = (1-\alpha) - \alpha\cos\left(\frac{2\pi n}{L-1}\right) \qquad L: window\ width$$

我选择的是汉明窗。

汉明窗代码：

```
def hamming(frame_width):  # 默认帧长和帧移的单位为ms
    """
    汉明窗权重
    :param frame_width: 帧长，默认单位为ms
    :return: w--相应的汉明窗权重
    """
    width = round(frame_width * 44.1)  # 帧长为25ms，采样率为44100Hz。
width=round(frame_width*44.1)
    w = []  # window权重
    for i in range(0, width):
        w.append(0.54 - 0.46 * math.cos(2 * np.pi * i / (width - 1)))  # 计算汉明
窗的权重
    w = np.array(w)
    # print("汉明窗权重：", w)
    return w
```

在分帧结束后，会得到若干帧语音信号。对每帧信号进行加窗处理，得到短时加窗的语音信号$x_l[n]$ 。

计算公式：

$$x_t[n] = w[n]\ x_t'[n] \qquad 0 \le n \le L-1$$

其中，$w[n]$是窗函数，L是窗长，我设置的是帧长=窗长。

代码：

```
def window(signal, w, frame_shift):
    """
    分帧并加窗
    :param signal:采样信号
    :param w:窗函数
    :param frame_shift:帧移
    :return:分帧并加窗后的信号
    """
    n = len(signal)  # 采样点个数
    width = len(w)  # 帧长
    shift = round(frame_shift * 44.1)  # 帧移为10ms，采样率为44100Hz。
shift=round(frame_shift*44.1)
    nf = (n - width + shift) // shift  # //-整除（向下取整）
    # 计算分帧后的帧数：(n-overlap)/shift
    # 重叠部分：overlap=width-shift
    window_signal = np.zeros((nf, width))  # 初始化
    df = np.multiply(shift, np.array([i for i in range(nf)]))  # 设置每帧在x中的位移
量位置
    for i in range(nf):
        window_signal[i, :] = signal[df[i]:df[i] + width]  # 将数据分帧，即nf x
width
    window_signal = np.multiply(window_signal, np.array(w))
```

```
    # print("加窗后的采样点：", window_signal)
    return window_signal
```

四、DFT（离散傅里叶变换）

人类语音的感知过程于听觉系统具有频谱分析功能紧密相关。而声音从频率上可以分为纯音和复合音。大部分声音都是复合音，涉及多个频率段。

由傅里叶级数可以知道，任何周期函数都可以可以用正弦函数和余弦函数构成的无穷级数来表示。利用三角函数的正交性、欧拉公式等，由傅里叶级数推演得到了傅里叶变换。而傅里叶变换的核心就是实现从时域到频域的变换。所以我们可以利用它来得到复合音的多个频率段。

由于我们得到的语音信号实际上是一个个采样点，所以需要使用的是离散傅里叶变换（DFT）。在python中可以直接调用numpy库中的fft（快速傅里叶变换）来实现。

在前面几步中，我们分帧、加窗，将语音信号分割成一帧帧的离散序列，所以这一步可以视作短时傅里叶变换（SFFT）。

DFT的公式：

$$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-j\frac{2\pi}{N}kn}$$

其中，N为一帧的采样点个数。如，我们的采样频率为4000Hz，一帧取25ms，则 $N = 4000Hz \times 25ms$。

代码如下：

```
dft_signal = np.zeros((window_signal.shape[0], 2048), dtype=complex)
for i in range(window_signal.shape[0]):
    temp = np.fft.fft(window_signal[i], 2048)
    dft_signal[i] = temp
```

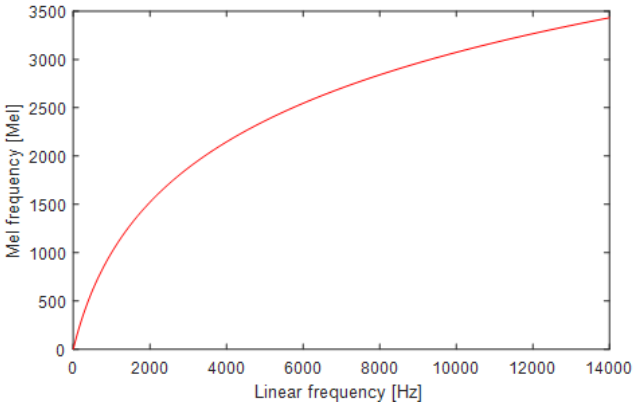
最终得到的是一组一组的**复数**。

五、Mel filterbank（梅尔滤波器）

人类感知声音，受到频率和声强影响。用频率来表示声音的音高，一般来说，频率低的声音，比较低沉；而频率高的声音，则比较尖锐。但是实际上， 音调和频率并不成正比。音高的单位是mel频率，用来模拟人耳对不同频率语音的感知。1mel相当于1kHz音调感知程度的1/1000。

Mel scale：

$$M(f) = 1127 \ln\left(1 + \frac{f}{700}\right)$$



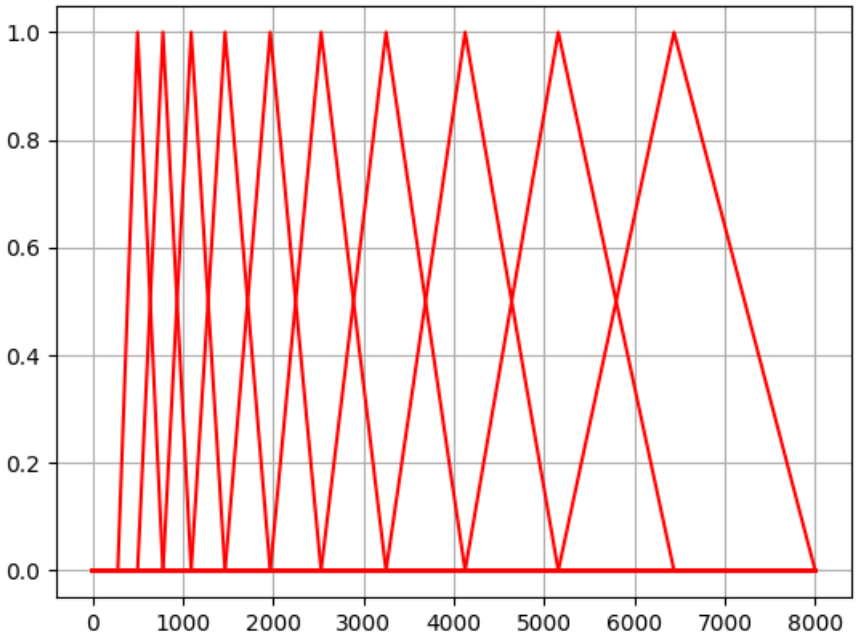
据研究，人耳对低频信号比对高频信号更敏感。所以研究者经过一系列实验得到了类似耳蜗作用的一个滤波器组，用于模拟人耳对不同频段声音的感知能力。每个滤波器带宽不等，线性频率小于1000Hz的部分为线性间隔，而线性频率大于1000Hz的部分为对数间隔。这就是我们需要使用到的Mel滤波器组。

Mel滤波器组的第m个滤波函数$H_m(k)$定义如下：

$$H_m(k)\begin{cases}0, & k<k_{b_{m-1}}\\ \frac{k-k_{b_{m-1}}}{k_{b_m}-k_{b_{m-1}}}, & k_{b_{m-1}}\le k\le k_{b_m}\\ 1, & k=k_{b_m}\\ \frac{k_{b_{m+1}}-k}{k_{b_{m+1}}-k_{b_m}}, & k_{b_m}\le k\le k_{b_{m+1}}\\ 0, & k\ge k_{b_{m+1}}\end{cases}$$

其中，$1\le m\le M$，M是滤波器的个数，k_{b_m}是滤波器的临界频率，k表示K点DFT变换的频谱系数序号。

下图是我绘制的Mel滤波器。其中M=10，采样频率=16kHz，DFT=512。



我们将上述做DFT后得到的**功率谱**通过Mel滤波器组可以得到一系列模拟人耳对声音的感知频率。

其中，**功率谱的公式**如下：

$$P[k]=\frac{X[k]^2}{N}$$

其中，$X[k]$是做DFT得到的第k个点的傅里叶频谱值，N是窗长。

Mel滤波器代码如下：

```
def mel_filters(m, n, fs, fl):
    """
    Mel滤波器
    :param m:滤波器个数
    :param n:一帧FFT后保留的点数
    :param fs:采样频率
    :param fl:最低频率
    :return:
    """
    fh = fs / 2  # 最高频率，为采样频率fs的一半
    ml = 1127 * np.log(1 + fl / 700)
    mh = 1127 * np.log(1 + fh / 700)  # 将Hz转换为Mel
    mel = np.linspace(ml, mh, m + 2)  # 将Mel刻度等间隔
    h = 700 * (np.exp(mel / 1127) - 1)  # 将Mel转换为Hz
    f = np.floor((n + 1) * h / fs)
    w = int(n / 2 + 1)  # 采样频率为fs/2的FFT的点数
    freq = [int(i * fs / n) for i in range(w)]  # 采样频率值，为了画图
    bank = np.zeros((m, w))
    for k in range(1, m + 1):
        f0 = f[k]
        f1 = f[k - 1]
        f2 = f[k + 1]
        for i in range(1, w):
            if i < f1:
                continue
            elif f1 <= i <= f0:
                bank[k - 1, i] = (i - f1) / (f0 - f1)
            elif f0 <= i <= f2:
                bank[k - 1, i] = (f2 - i) / (f2 - f0)
            else:
                break
    return bank
```

我选择的Mel滤波器个数是26个。

通过Mel滤波器的公式如下：

$$Y_t[m] = \sum_{k=1}^{N} w_m[k] \, |X_t[k]|^2$$

其中，m是Mel滤波器的序号，k是DFT的序号。

实际上，这个乘法是两个矩阵相乘，调用numpy中的dot即可完成。

六、Log（取对数）

在通过Mel滤波器组后得到了一组特征后，还需要对其取对数，目的是压缩动态范围。这也是因为人类对信号能量的敏感性是对数级的，也就是说，人类在高频时对频率的微小变化比对低频时的微小变化更不敏感。而对数使特征对声波耦合变化的影响更小。

这样就得到了FBank特征。FBank它根据人耳听觉感知特性进行了压缩，抑制了一部分听觉无法感知的冗余信息。

七、八过程的代码如下：

```
def get_fbank_feature(power, mel):
    """
    计算得到FBANK feature
    :param power: 功率谱
    :param mel: mel滤波器
    :return: fbank feature
    """
    # 功率谱*Mel滤波器
    # yt[m]=Σ(1,N)mel[m,k]power[t,k]
    # power(T,K) mel(M,K)
    temp = power[:, :mel.shape[1]]  # 切片，为了做矩阵乘法
    y = np.dot(temp, mel.T)

    # 得到第一个特征值
    fbank_features = np.log(y)
    # print(fbank_features.shape)
    return fbank_features
```

七、IDFT（倒谱分析）

在前面的过程中，我们将时域转换为频率，并根据语音的特性和人耳的特性对其进行了处理，得到的一个初步的特征。

语音信号的产生模型主要包括发生源（source）和滤波器（filter）。而根据语音信号的产生模型，语音信号可以用激励源和滤波器进行卷积操作来表示。为了更好地处理语音，我们想要得到参与语音信号卷积的各个信号分量，就需要一个解卷积处理。而倒谱分析就可以用于实现解卷积处理。

倒谱分析可以用逆傅里叶变换实现。但是，我们前面得到的特征是通过功率谱计算得到的，即是实数并且是对称的，所以可以通过离散余弦变换（DCT）实现。

离散余弦变换：

$$y[n]=\sum_{m=0}^{M-1}\log\left(Y[m]\right)\cos\left(n(m+0.5)\frac{\pi}{M}\right)$$

其中，n为做DCT后的谱线，m是滤波器的序号，M是滤波器总数。

一般我们保留前12个点。

经过倒谱分析，就初步得到了MFCC特征系数。

代码如下：

```
def get_mfcc(n_dct, mel, fbank_features):
    """
    对fbank feature做倒谱分析，得到MFCC特征系数
    :param n_dct: DCT后的谱线，为了做差分，会增加几位
    :param mel: mel滤波器
    :param fbank_features: fbank特征
    :return: MFCC特征系数
    """
    # 倒谱分析，DCT-离散余弦变换，得到MFCC系数
    # 取前12个系数
    # Σ(0,M-1)FBANK_features*cos(n(m+0.5)π/M)
```

```
    # n--DCT后的谱线 m--第m个滤波器 M--滤波器总数
    n_dct = 16  # 取前十二个，但是为了做差分，所以取16
    M = mel.shape[0]
    m = np.array([i for i in range(M)])  # mel.shape[0]-mel滤波器的个数
    mfcc = np.zeros((fbank_features.shape[0] - 4, n_dct))  # 因为做差分后会有值的减少，所以会从原来的m行变为m-4行
    for i in range(mfcc.shape[0]):
        for j in range(mfcc.shape[1]):
            mfcc[i, j] = np.sum(np.multiply(fbank_features[i, :], np.cos((m + 0.5) * j * np.pi / M)))
    return mfcc
```

八、Dynamic features（动态特征）

因为语音不是固定的帧对帧，所以我们可以添加一些特征来处理倒谱系数随时间的变化，即对前面得到的MFCC系数分别进行一阶差分、二阶差分，并且对于做DFT后的能量谱也进行一阶差分、二阶差分。最终得到了一组组39维的向量。

差分公式为:

$$d(t) = \frac{c(t+1) - c(t-1)}{2}$$

最终得到的是:

12 MFCCs, energy

12 ΔMFCCs, Δenergy

12 Δ^2MFCCs,Δ^2energy

计算差分的代码如下:

```
def get_diff(mfcc, energy):
    """
    计算差分，最后得到每行39维的矩阵
    :param mfcc: MFCC特征系数
    :param energy: 能量谱
    :return: 动态特征
    """
    # 做差分，12+一个能量信息，共39维
    # 差分：d(t)=(c(t+1)-c(t-1))/2
    # 因为做了差分，所以值会有缩减，如能量
    # 先分别做差分
    m, n = mfcc.shape
    # 一阶差分
    d_mfcc = np.zeros((m, n - 2))
    for i in range(m):
        for j in range(1, n - 1):
            d_mfcc[i, j - 1] = (mfcc[i, j + 1] - mfcc[i, j - 1]) / 2
    # print("d_mfcc", d_mfcc[1])
    # print("energy:", energy[:7])
    d_energy = np.zeros(len(energy) - 2)
    for i in range(1, len(d_energy) - 1):
        d_energy[i - 1] = (energy[i + 1] - energy[i - 1]) / 2
    # print("d_energy:", d_energy[:5])
```

```
    # 二阶差分
    dd_mfcc = np.zeros((m, n - 4))
    for i in range(m):
        for j in range(1, n - 3):
            dd_mfcc[i, j - 1] = (d_mfcc[i, j + 1] - d_mfcc[i, j - 1]) / 2
    # print("dd_mfcc", dd_mfcc[1])
    dd_energy = np.zeros(len(d_energy) - 2)
    for i in range(1, len(dd_energy) - 1):
        dd_energy[i - 1] = (d_energy[i + 1] - d_energy[i - 1]) / 2
    # print("dd_energy:", dd_energy[:3])

    m = len(dd_energy)
    n = 3 * (dd_mfcc.shape[1] + 1)  # 应该是3*（12个MFCC系数+1个能量）
    # print(m, n)
    diff = np.zeros((m, n))
    for i in range(m):
        diff[i, :int(n / 3 - 1)] = mfcc[i, :int(n / 3 - 1)]  # 取mfcc的前12个
[0:12]
        diff[i, int(n / 3 - 1)] = energy[i]
        diff[i, int(n / 3):int(2 * n / 3 - 1)] = d_mfcc[i, int(n / 3 - 1)]  # 取
d_mfcc的前12个#[13:25]
        diff[i, int(2 * n / 3 - 1)] = d_energy[i]
        diff[i, int(2 * n / 3):n - 1] = dd_mfcc[i]  # [26:38]
        diff[i, n - 1] = dd_energy[i]
    return diff
```

九、Feature Transform（特征变换）

最后，为了提取的特征能够减少训练和测试的不匹配，所以我们需要对特征做标准化。

标准化公式：

$$\widehat{y_t}[j] = \frac{y_t[j] - \mu(y[j])}{\sigma(y[j])}$$

其中，$\mu(y[j])$是第 j 列的均值，$\sigma(y[j])$是第 j 列的标准差。

代码如下：

```
def normalize(diff):
    """
    标准化
    :param diff:动态特征
    :return: 标准化后的特征
    """
    # 标准化
    # 先计算每一维的均值和标准差
    avr = np.mean(diff, axis=0)
    std = np.std(diff, axis=0)

    # 进行标准化 normalize
    # y'=(y-a(y))/v(y)
    # a(y)--平均值 v(y)--方差
    normalization = np.zeros((diff.shape[0], diff.shape[1]))
    for i in range(normalization.shape[0]):
        normalization[i, :] = np.divide(np.subtract(diff[i, :], avr), std)
```

```
    return normalization
```

最后，将得到的特征值保存到.txt中。

参考资料：

Mel Frequency Cepstral Coefficient(MFCC) tutorial:http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/#eqn1

python语音信号基础： https://blog.csdn.net/sinat_18131557/category_9876029.html