In [1]:
```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import jieba
import numpy as np
import time
```

In [2]:
```python
# 定义翻译器网络结构
class my_translator(nn.Module):
    # 初始化模型参数
    def __init__(self, eng_vocab_size, chs_vocab_size, embedding_size, hidden_size, num_layers=1):
        super().__init__()
        # 构建模型的神经网络结构
        # 网络结构为Encoder-Decoder，其中Encoder输入英文句子，Decoder输入中文，故需对两类的网络结构区分
        self.encoder_embedding = nn.Embedding(eng_vocab_size, embedding_size)
        # Encoder的LSTM层用于提取输入英文的序列特征
        self.encoder_lstm = nn.LSTM(embedding_size, hidden_size, num_layers, batch_first=True)
        # Decoder输入中文句子，故其处理的维度在这里应该是chs_vocab_size
        self.decoder_embedding = nn.Embedding(chs_vocab_size, embedding_size)
        # Decoder的LSTM层用于提取输入中文的序列特征
        self.decoder_lstm = nn.LSTM(embedding_size, hidden_size, num_layers, batch_first=True)
        # 全连接层，将输出映射到字典大小，用于对字的预测
        self.fc = nn.Linear(hidden_size, chs_vocab_size)

        # 首先实例化输入变量
        self.chs_vocab_size = chs_vocab_size
        self.eng_vocab_size = eng_vocab_size
        self.embedding_size = embedding_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers

    # 之后构建网络的前向传播函数
    def forward(self, eng, chs, hidden_vec=None):
        """
        输入大小
        eng_txt, chs_txt: [batch_size, seq_len]
        隐藏向量h，c的大小应为[num_layers, batch_size, hidden_size]
        """
        # LSTM层需要初始的隐藏层，如果没有的话，需要手动初始化一下，并将其移至cuda上
        if hidden_vec == None:
            # 简单初始为零张量
            # 用eng_txt.shape[0]/chs_txt.shape[0]来代表batch_size
            h0 = torch.zeros(self.num_layers, eng.shape[0], self.hidden_size).cuda()
            c0 = torch.zeros(self.num_layers, eng.shape[0], self.hidden_size).cuda()
        # 初始化完毕后，将英文文本输入嵌入，之后传入Encoder_LSTM层提取序列特征
        eng_embedded = self.encoder_embedding(eng)
        # eng_embedded: [batch_size, seq_len, embedding_size]
        _, eng_feature = self.encoder_lstm(eng_embedded, (h0, c0))
        # eng_feature: [num_layers, batch_size, hidden_size]
        # Encoder的工作做完后，将中文传入Decoder
        chs_embedded = self.decoder_embedding(chs)
        # 将Encoder_LSTM输出的英文序列特征作为Decoder的初始状态，对中文输入进行解密，预测下一位置的字
        output, _ = self.decoder_lstm(chs_embedded, eng_feature)
        # output size:[batch_size, seq_len, hidden_size]，需要经过全连接
        output = self.fc(output)
        return output
```

In [4]:
```python
# 读取语料
end_symb = ["。", "？", "！"]
def read_corpus(path):
    # 创建英文与中文的空列表，用于存放每一个句子
    English = []
    Chinese = []
    # 用open函数读取文档
    f = open(path, 'r', encoding = 'utf-8')
    # 遍历文档的每一行
    for line in f.readlines():
        # 用line.strip()剔除掉每行的开头序号与结尾换行符
        # split函数按中间的“tab”将语段切分，分为英文与中文句子
        eng, chs = line.strip().split('\t')
        # 剔除掉英文句子的最后一个标点符号
        eng = eng[:-1]
        # 按空格切分英文语段，将每个词单独出来
        eng = eng.split(' ')
        # 将这一句加入English列表
        English.append(eng)
        # 同样去除中文的结束标点符号
        if chs[-1] in end_symb:
```

```
            chs = chs[:-1]
        # 用jieba对中文分词
        chs = jieba.lcut(chs)
        # 在每个中文语句前加入开始符，结尾加入结束符
        chs = ['B'] + chs + ['S']
        # 将这一句加入Chinese列表
        Chinese.append(chs)
    return English, Chinese

English, Chinese = read_corpus('../Pytorch_Book_ZhouRUC/dataset/cmn.txt')
print(English[10000])
print(Chinese[10000])
```

```
Building prefix dict from the default dictionary ...
Loading model from cache C:\Users\dgq\AppData\Local\Temp\jieba.cache
Loading model cost 0.559 seconds.
Prefix dict has been built successfully.
['Tom', 'is', 'standing', 'in', 'the', 'garden']
['B', '汤姆', '在', '花园里', '站', '着', 'S']
```

In [5]:

```
# 中英文字典编码
def lang_encode(language):
    # 初始化语言对应的字典
    lang2idx = {}
    i = 1
    # 循环遍历每一句话中的每一个词
    for chs in language:
        for c in chs:
            # 如果这个词没有被包含在字典中，就将他纳入字典，索引记为i
            if lang2idx.get(c) == None:
                lang2idx[c] = i
                i += 1
    return lang2idx
# 生成中文字典
chs2idx = lang_encode(Chinese)
# 生成英文字典
eng2idx = lang_encode(English)
# 中文字典大小
chs_vocab_size = len(chs2idx.keys()) + 1
# 英文字典大小
eng_vocab_size = len(eng2idx.keys()) + 1

print('中文字典大小', chs_vocab_size)
print('英文字典大小', eng_vocab_size)
```

```
中文字典大小 13684
英文字典大小 7814
```

In [6]:

```
# 对文本编码
def text_encode(lang2idx, language):
    # 首先生成一个存放编码后句子的列表，功能类似前面的English/Chinese
    text_digit = []
    # 循环每一句
    for txt in language:
        # 将每一个词替换成它的编码，放入t_digit中
        t_digit = []
        for t in txt:
            t_digit.append(lang2idx[t])
        text_digit.append(t_digit)
    return text_digit

# 生成编码后的中文语句列表
chs_digit = text_encode(chs2idx, Chinese)
# 生成编码后的英文语句列表
eng_digit = text_encode(eng2idx, English)
print("原始中文：")
print(Chinese[0])
print("中文编码后的结果：")
print(chs_digit[0])
print("原始英文：")
print(English[0])
print("英文编码后的结果：")
print(eng_digit[0])
```

```
原始中文:
['B', '嗨', 'S']
中文编码后的结果:
[1, 2, 3]
原始英文:
['Hi']
英文编码后的结果:
[1]
```

In [7]:
```
# 生成训练输入输出序列函数
def generate_XY(chs_digit, eng_digit, max_len):
    # X用来存放输入样本
    X = []
    # Y用来存放输出的标签
    Y = []
    # 循环编码后的每个中文与英文句子
    for i in range(len(chs_digit)):
        # 中文的输入部分是每一句的结束符前面的部分，空缺的部分用0补齐
        x1 = chs_digit[i][:-1] + [0]*(max_len - len(chs_digit[i])+1)
        # 英文的输入部分就是一整个编码后的句子成分，空缺的部分用0补齐
        x2 = eng_digit[i] + [0]*(max_len - len(eng_digit[i]))
        # 将x1与x2合并在一起作为模型的输入X
        X.append(x1 + x2)
        # 模型的输出应为中文每一句起始符后面的部分
        y = chs_digit[i][1:] + [0]*(max_len - len(chs_digit[i])+1)
        # 将y添加进最后的整个输出列表Y
        Y.append(y)
    return X, Y

# 生成本例中的输入与输出样本
X, Y = generate_XY(chs_digit, eng_digit, max_len=32)
print("原始中文: ")
print(Chinese[500])
print("变量X_chs: ")
print(X[500][0:32])
print("变量X_eng: ")
print(X[500][32:])
print("变量Y: ")
print(Y[500])
```

```
原始中文:
['B', '汤姆', '告诉', '了', '他', 'S']
变量X_chs:
[1, 36, 545, 14, 19, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
变量X_eng:
[29, 454, 76, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
变量Y:
[36, 545, 14, 19, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

In [8]:
```
# 划分训练集和验证集
# 将所有数据的顺序打乱重排
idx = np.random.permutation(range(len(X)))
# 按照打乱的顺序重新编排输入与输出样本X、Y
X = [X[i] for i in idx]
Y = [Y[i] for i in idx]

# 切分出1/5的数据作为验证集
validX = X[: len(X) // 5]
trainX = X[len(X) // 5 :]
validY = Y[: len(Y) // 5]
trainY = Y[len(Y) // 5 :]
```

In [9]:
```
# 构建训练集、测试集的dataset与dataloader
batch_size = 50
# 生成训练集的dataset，将训练部分的X和Y先转化为np矩阵再转化为tensor，然后用TensorDataset构建
train_set = torch.utils.data.TensorDataset(torch.IntTensor(np.array(trainX, dtype=int)),
                                           torch.IntTensor(np.array(trainY, dtype=int)))
# 用相同的办法构建验证集的dataset
val_set = torch.utils.data.TensorDataset(torch.IntTensor(np.array(validX, dtype=int)),
                                         torch.IntTensor(np.array(validY, dtype=int)))
# 构建训练集的dataloader，打乱
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True)
# 构建验证集的dataloader，不打乱
val_loader = torch.utils.data.DataLoader(val_set, batch_size=batch_size, shuffle=False)
```

In [10]:
```
#给定超参数
lr = 1e-3
epochs = 50
# 创建机器翻译模型实例
translator = my_translator(eng_vocab_size=eng_vocab_size, chs_vocab_size=chs_vocab_size, embedding_size=64, h
```

```python
# 将translator的参数转移到cuda上训练
translator = translator.cuda()
# 损失函数选定为交叉熵
criterion = torch.nn.CrossEntropyLoss()
# 优化器选定为Adam
optimizer = torch.optim.Adam(translator.parameters(), lr=lr)
#查看模型具体信息
print(translator)
```

```
my_translator(
  (encoder_embedding): Embedding(7814, 64)
  (encoder_lstm): LSTM(64, 128, batch_first=True)
  (decoder_embedding): Embedding(13684, 64)
  (decoder_lstm): LSTM(64, 128, batch_first=True)
  (fc): Linear(in_features=128, out_features=13684, bias=True)
)
```

In [11]:

```python
# 定义预测准确率函数
# outputs shape: [batch_size, seq_len, chs_vocab_size]
# labels shape: [batch_size, seq_len]
def accuracy(outputs, labels):
    # 首先通过softmax把outputs第三维转化为对应字的预测概率
    pre = F.softmax(outputs, dim=2)
    # 然后再提取出预测最大值的下标
    pre = torch.max(pre, dim=2)[1]
    # pre shape: [batch_size, seq_len]
    # 预测正确的个数占总个数的比为准确率
    acc = torch.sum(pre == labels).item() / (labels.shape[0] * labels.shape[1])
    return acc
```

In [12]:

```python
# 定义一个tensor分割函数
# 提取出输入encoder的英文部分和输入decoder的中文部分
def split_chs_eng(x, max_len):
    # 先将输入转化为列表
    x = x.tolist()
    # 前一部分为中文部分
    x1 = [x[i][0:max_len] for i in range(len(x))]
    # 后一部分为英文部分
    x2 = [x[i][max_len:] for i in range(len(x))]
    # 再将这两部分转化回tensor
    x1 = torch.IntTensor(np.array(x1, dtype=int))
    x2 = torch.IntTensor(np.array(x2, dtype=int))
    return x1.cuda(), x2.cuda()
```

In [13]:

```python
# 定义训练过程打印函数
def print_log(epoch, train_time, train_loss, train_acc, val_loss, val_acc, epochs=10):
    print(f"Epoch [{epoch}/{epochs}], time: {train_time:.2f}s, loss: {train_loss:.4f}, acc: {train_acc:.4f}, v
```

In [14]:

```python
# 定义模型验证过程
def validate(model, val_loader, max_len = 32):
    # 在验证集上运行一遍并计算损失和准确率
    # 只进行一次epoch
    val_loss = 0
    val_acc = 0
    # 标识模型进行预测，不更新梯度
    model.eval()
    for batch, data in enumerate(val_loader):
        # 提取x作为输入，y作为标签
        x, y = data[0], data[1]
        # 将输出移至GPU
        y = y.cuda()
        # 将输入切割成两个张量
        # 一个为输入encoder的英文部分，另一个为输入decoder的中文部分
        chs, eng = split_chs_eng(x, max_len)
        # 将chs、eng传入模型，得到输出
        y_pred = model(eng, chs)
        # 将标签转化为long型，以便进行交叉熵运算
        y = y.long()
        # 计算当前损失
        # 将预测的形状转化为[batch_size, num_classes, seq_len]，这样才符合交叉熵计算的标准
        y_pred = y_pred.permute(0,2,1)
        # 计算交叉熵
        loss = criterion(y_pred, y)
        # 将预测变回原来的形状
        y_pred = y_pred.permute(0,2,1)
        # 将这一mini-batch的损失值加入val_loss
        val_loss += loss.item()
        # 将这一mini-batch的精度值加入train_acc
        val_acc += accuracy(y_pred, y)
```

```python
    # 计算平均损失
    val_loss /= len(val_loader)
    # 计算平均准确率
    val_acc /= len(val_loader)
    return val_loss, val_acc
```

In [15]:

```python
# 定义模型训练函数
def train(model, optimizer, train_loader, val_loader, max_len = 32, epochs=50):
    # 首先定义一组存储训练/验证过程中的损失和准确率的列表
    train_losses = []
    train_accs = []
    val_losses = []
    val_accs = []
    # 进行一个训练的epoch
    for epoch in range(epochs):
        # 先初始化训练损失和训练精度
        train_loss = 0
        train_acc = 0
        # 记录当前epoch开始时间
        start = time.time()
        # batch为数字，表示已经进行了几个batch
        # data为一个二元组，存储了一个样本的输入和标签
        for batch, data in enumerate(train_loader):
            # 标识模型进行训练
            model.train()
            # 提取x作为输入，y作为标签
            x, y = data[0], data[1]
            # 将y移至GPU上存储、计算
            y = y.cuda()
            # 将输入切割成两个张量
            # 一个为输入encoder的英文部分，另一个为输入decoder的中文部分
            chs, eng = split_chs_eng(x, max_len)
            # 将优化器的梯度清空，准备训练
            optimizer.zero_grad()
            # 将chs、eng传入模型，得到输出
            y_pred = model(eng, chs)
            # 将标签转化为long型，以便进行交叉熵运算
            y = y.long()
            # 计算当前损失
            # 将预测的形状转化为[batch_size, num_classes, seq_len]，这样才符合交叉熵计算的标准
            y_pred = y_pred.permute(0,2,1)
            # 计算交叉熵
            loss = criterion(y_pred, y)
            # 将预测变回原来的形状
            y_pred = y_pred.permute(0,2,1)
            # 将这一mini-batch的损失值加入train_loss
            train_loss += loss.item()
            # 将这一mini-batch的精度值加入train_acc
            train_acc += accuracy(y_pred, y)
            # 进行反向传播，梯度下降
            loss.backward()
            optimizer.step()

        # 记录当前epoch结束时间
        end = time.time()
        # 计算当前epoch的训练耗时
        train_time = end - start
        # 计算平均损失
        train_loss /= len(train_loader)
        # 计算平均准确率
        train_acc /= len(train_loader)
        # 计算验证集上的损失函数和准确率
        val_loss, val_acc = validate(model, val_loader, max_len)
        train_losses.append(train_loss)
        train_accs.append(train_acc)
        val_losses.append(val_loss)
        val_accs.append(val_acc)
        print_log(epoch + 1, train_time, train_loss, train_acc, val_loss, val_acc, epochs=epochs)

    return train_losses, train_accs, val_losses, val_accs
```

In [16]:

```python
# 模型训练
history = train(translator, optimizer, train_loader, val_loader, epochs=50)
```

```
Epoch [1/50], time: 7.93s, loss: 1.7076, acc: 0.8130, val_loss: 1.2913, val_acc: 0.8257
Epoch [2/50], time: 7.31s, loss: 1.2390, acc: 0.8298, val_loss: 1.2640, val_acc: 0.8285
Epoch [3/50], time: 7.05s, loss: 1.1886, acc: 0.8324, val_loss: 1.2272, val_acc: 0.8311
Epoch [4/50], time: 7.02s, loss: 1.1333, acc: 0.8358, val_loss: 1.1904, val_acc: 0.8355
Epoch [5/50], time: 6.98s, loss: 1.0772, acc: 0.8411, val_loss: 1.1630, val_acc: 0.8402
Epoch [6/50], time: 7.07s, loss: 1.0258, acc: 0.8458, val_loss: 1.1389, val_acc: 0.8451
Epoch [7/50], time: 7.23s, loss: 0.9746, acc: 0.8513, val_loss: 1.1185, val_acc: 0.8476
Epoch [8/50], time: 7.10s, loss: 0.9299, acc: 0.8542, val_loss: 1.1094, val_acc: 0.8493
Epoch [9/50], time: 7.22s, loss: 0.8887, acc: 0.8572, val_loss: 1.1000, val_acc: 0.8507
Epoch [10/50], time: 7.21s, loss: 0.8501, acc: 0.8600, val_loss: 1.0930, val_acc: 0.8520
Epoch [11/50], time: 7.04s, loss: 0.8136, acc: 0.8622, val_loss: 1.0880, val_acc: 0.8534
Epoch [12/50], time: 7.06s, loss: 0.7779, acc: 0.8652, val_loss: 1.0862, val_acc: 0.8543
Epoch [13/50], time: 6.90s, loss: 0.7446, acc: 0.8679, val_loss: 1.0860, val_acc: 0.8552
Epoch [14/50], time: 7.10s, loss: 0.7121, acc: 0.8713, val_loss: 1.0854, val_acc: 0.8555
Epoch [15/50], time: 7.20s, loss: 0.6812, acc: 0.8752, val_loss: 1.0901, val_acc: 0.8563
Epoch [16/50], time: 7.40s, loss: 0.6519, acc: 0.8792, val_loss: 1.0901, val_acc: 0.8563
Epoch [17/50], time: 7.32s, loss: 0.6244, acc: 0.8833, val_loss: 1.0951, val_acc: 0.8564
Epoch [18/50], time: 7.06s, loss: 0.5987, acc: 0.8870, val_loss: 1.0997, val_acc: 0.8570
Epoch [19/50], time: 6.99s, loss: 0.5746, acc: 0.8906, val_loss: 1.1069, val_acc: 0.8570
Epoch [20/50], time: 7.01s, loss: 0.5521, acc: 0.8939, val_loss: 1.1116, val_acc: 0.8578
Epoch [21/50], time: 6.94s, loss: 0.5314, acc: 0.8970, val_loss: 1.1215, val_acc: 0.8579
Epoch [22/50], time: 7.22s, loss: 0.5119, acc: 0.9001, val_loss: 1.1273, val_acc: 0.8579
Epoch [23/50], time: 7.09s, loss: 0.4932, acc: 0.9032, val_loss: 1.1376, val_acc: 0.8578
Epoch [24/50], time: 7.25s, loss: 0.4759, acc: 0.9061, val_loss: 1.1443, val_acc: 0.8579
Epoch [25/50], time: 7.39s, loss: 0.4590, acc: 0.9087, val_loss: 1.1533, val_acc: 0.8580
Epoch [26/50], time: 6.96s, loss: 0.4432, acc: 0.9114, val_loss: 1.1628, val_acc: 0.8581
Epoch [27/50], time: 6.72s, loss: 0.4289, acc: 0.9138, val_loss: 1.1731, val_acc: 0.8581
Epoch [28/50], time: 6.99s, loss: 0.4139, acc: 0.9164, val_loss: 1.1832, val_acc: 0.8584
Epoch [29/50], time: 7.10s, loss: 0.4001, acc: 0.9189, val_loss: 1.1915, val_acc: 0.8579
Epoch [30/50], time: 7.42s, loss: 0.3871, acc: 0.9212, val_loss: 1.2013, val_acc: 0.8581
Epoch [31/50], time: 7.40s, loss: 0.3745, acc: 0.9233, val_loss: 1.2131, val_acc: 0.8581
Epoch [32/50], time: 7.25s, loss: 0.3623, acc: 0.9256, val_loss: 1.2224, val_acc: 0.8576
Epoch [33/50], time: 6.97s, loss: 0.3511, acc: 0.9276, val_loss: 1.2348, val_acc: 0.8575
Epoch [34/50], time: 6.85s, loss: 0.3398, acc: 0.9296, val_loss: 1.2454, val_acc: 0.8576
Epoch [35/50], time: 7.01s, loss: 0.3293, acc: 0.9314, val_loss: 1.2583, val_acc: 0.8581
Epoch [36/50], time: 6.87s, loss: 0.3190, acc: 0.9332, val_loss: 1.2675, val_acc: 0.8577
Epoch [37/50], time: 7.09s, loss: 0.3089, acc: 0.9355, val_loss: 1.2794, val_acc: 0.8577
Epoch [38/50], time: 7.28s, loss: 0.2995, acc: 0.9370, val_loss: 1.2922, val_acc: 0.8579
Epoch [39/50], time: 7.29s, loss: 0.2910, acc: 0.9386, val_loss: 1.3023, val_acc: 0.8574
Epoch [40/50], time: 7.17s, loss: 0.2814, acc: 0.9405, val_loss: 1.3126, val_acc: 0.8577
Epoch [41/50], time: 7.09s, loss: 0.2732, acc: 0.9422, val_loss: 1.3243, val_acc: 0.8573
Epoch [42/50], time: 7.17s, loss: 0.2647, acc: 0.9441, val_loss: 1.3362, val_acc: 0.8571
Epoch [43/50], time: 7.06s, loss: 0.2563, acc: 0.9457, val_loss: 1.3484, val_acc: 0.8577
Epoch [44/50], time: 7.13s, loss: 0.2488, acc: 0.9472, val_loss: 1.3590, val_acc: 0.8569
Epoch [45/50], time: 7.43s, loss: 0.2414, acc: 0.9487, val_loss: 1.3670, val_acc: 0.8568
Epoch [46/50], time: 7.46s, loss: 0.2338, acc: 0.9506, val_loss: 1.3802, val_acc: 0.8567
Epoch [47/50], time: 7.37s, loss: 0.2267, acc: 0.9518, val_loss: 1.3896, val_acc: 0.8567
Epoch [48/50], time: 7.24s, loss: 0.2204, acc: 0.9532, val_loss: 1.4012, val_acc: 0.8562
Epoch [49/50], time: 7.01s, loss: 0.2134, acc: 0.9547, val_loss: 1.4160, val_acc: 0.8560
Epoch [50/50], time: 7.06s, loss: 0.2067, acc: 0.9561, val_loss: 1.4229, val_acc: 0.8561
```

In [31]:

```python
# 模型翻译测试
max_len = 32

# 准备英文文本
test = 'I love you'
# test = ["I", "love", "you"]
test = test.split(' ')
# 用eng存储每个字母的编码
eng = []
# 将每个词的编码添加到eng中
for t in test:
    eng.append(eng2idx[t])
# eng shape: [1, 32]
eng = eng + [0]*(max_len - len(eng))
eng = torch.IntTensor(np.array([eng], dtype=int))
# 初始化中文文本
# 只保留起始符，有encoder的信息一步一步翻译
# chs shape: [1, 32]
chs = [chs2idx['B']] + [0]*(max_len - 1)
chs = torch.IntTensor(np.array([chs], dtype=int))
# 将eng，chs移至GPU
eng, chs = eng.cuda(), chs.cuda()
chs_txt = ""
# 进行一步步的翻译步骤
for i in range(max_len - 1):
    # 得到输出
    # output shape: [1, 32, chs_vocab_size]
    output = translator(eng, chs)
    # 提取第i个位置所预测的词
```

```python
    pred = torch.argmax(output[0, i])
    # 将这个词的编码加入chs中，用于下一轮的下一个位置的预测
    chs[0, i+1] = pred
    # 找到这个预测编码所对应的词
    char = [k for k, v in chs2idx.items() if v == pred][0]
    # 如果这个词是结束符，则翻译结束
    if char == "S":
        break
    # 不然的话，将其纳入翻译的文本中
    chs_txt += char

print(chs_txt)
```

我喜歡你