第 8 章　大模型实验

本章实验将涉及到大模型实验的人工智能应用在深度学习处理器上的开发和优化。

第8.1节介绍基于 Stable_diffusion 实现图像生成的功能。主要内容为完成各项基础运算配置、实现在潜在空间的反向过程特征采样、补全推理阶段的关键步骤以及一系列基本适配工作，实现图生图、文生图以及图像修复功能。

第8.2节介绍基于 Llama 2 实现聊天机器人应用并在 DLP 平台上进行部署。主要内容为完成模型加载和适配、对话模板的应用、文本生成的基本流程、模型推理的关键步骤，实现 DLP 上的机器人聊天功能。

第8.3节介绍基于 Code Llama 实现代码生成并在 DLP 平台上进行部署。主要内容为补全推理模块代码，具体包括模型加载和设置、文本生成和性能统计、结果处理和打印等步骤，实现 DLP 上的代码生成。

8.1　模型推理——Stable_diffusion

8.1.1　实验目的

熟悉潜在扩散模型的算法原理，能够在 DLP 平台上使用 Python 语言基于 Stable diffusion 实现图生图、文生图以及图像修复功能，具体包括：

1. 介绍基于潜在扩散模型的 Stable diffusion 的基本原理、计算步骤，及其在图像领域生成中的应用；

2. 深入了解潜在扩散模型的关键概念及底层逻辑，包括模型加载、模型推理的关键步骤等；

3. 了解 Stable diffusion 图像生成的基本流程，在 DLP 平台上实现图生图、文生图、图像修复功能，为后续针对不同需求使用不同的图像生成大模型奠定基础。

实验工作量：约 70 行代码，5 小时。

8.1.2　背景介绍

Stable Diffusion 是 StabilityAI 于 2022 年发布的基于潜在扩散模型[28] 的生成式大模型。其通过对海量图像数据进行预训练的方式，让模型能够根据任意输入 (文本/图像) 生成高质量的图像，主要应用包含文本生成图像、图像生成图像以及图像补全等。与传统基于 GAN 的图像生成模型相比，它能根据需求更高效地生成高质量的图像，是 AI 图像生成领域的里程碑。根据预训练版本以及应用任务的不同，目前 Stable Diffusion 的模型包括 Stable Diffusion V1.5、Stable Diffusion V2.1、Stable Diffusion Inpainting、Stable Diffusion x4-Upscaler 等多种模型。接下来对扩散模型和 Stable Diffusion 的相关原理以及基础网络结构进行详细介绍。

8.1.2.1 扩散模型

如图8.1所示，扩散模型包括正向过程（也称为扩散过程）和反向过程（也称为逆扩散过程）。以 DDPM[29] 为例，接下来简单介绍扩散模型的基本原理。具体地，正向扩散过程

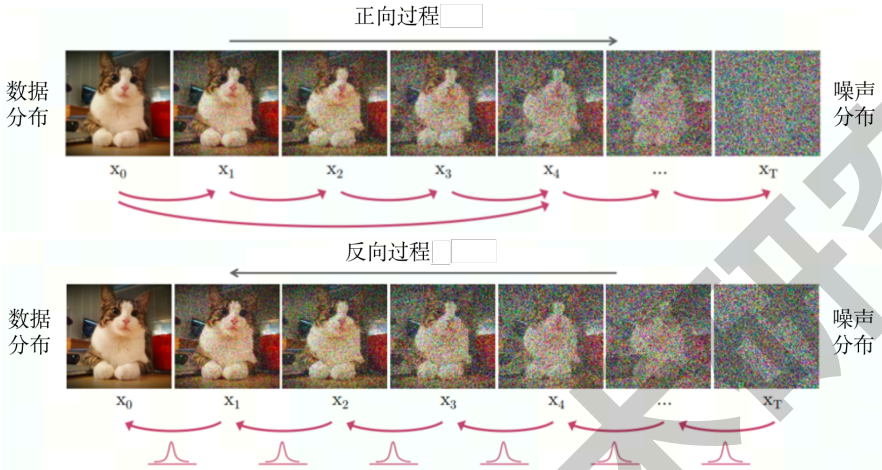

图 8.1 扩散模型的正向过程与反向过程[30]

是一个不断加噪声的过程。给定输入图像 $\boldsymbol{x}_0$，不断在前一时刻的图像上添加高斯噪声，经过 T 步依次得到噪声图像 $\boldsymbol{x}_1, ..., \boldsymbol{x}_t, ..., \boldsymbol{x}_T$，最终将输入数据分布转换为近似标准正态分布（即高斯分布）。其噪声添加路径是一条马尔可夫链，即在第 t 时间步加噪时，$\boldsymbol{x}_t$ 与 $\boldsymbol{x}_{t-1}$ 之间的条件概率 $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ 满足

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) = \mathcal{N}(\boldsymbol{x}_t; \sqrt{1-\beta_t}\boldsymbol{x}_{t-1}, \beta_t \boldsymbol{I}) \tag{8.1}$$

其中，$\boldsymbol{I}$ 为单位矩阵，$\mathcal{N}(\boldsymbol{x}; \mu, \sigma^2)$ 表示产生 $\boldsymbol{x}$ 的均值为 μ、方差为 σ^2 的高斯分布，$\beta_t \in [0, 1]$ 是预先设置的常量，一般设为随着 t 的增加而增大的值。正向过程的后验概率 $q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)$ 为

$$q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0) = \prod_{t=1}^{T} q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) \tag{8.2}$$

令 $\alpha_t = 1 - \beta_t, \bar{\alpha}_t = \prod_{i=1}^{T} \alpha_i$ 来简化参数，则有

$$q(\boldsymbol{x}_t|\boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{x}_t; \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0, (1-\bar{\alpha}_t)\boldsymbol{I}) \tag{8.3}$$

有了 $\boldsymbol{x}_t$ 关于 $\boldsymbol{x}_0$ 的条件概率分布，可以通过采样的方式获得样本：

$$\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\epsilon, \quad \epsilon \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}) \tag{8.4}$$

即，每个图像 $\boldsymbol{x}_t$ 均可用原始输入 $\boldsymbol{x}_0$ 与参数 $\bar{\alpha}_t$ 的表达式表示。换句话说，正向过程不需要对中间每个步骤进行采样，而是可以直接通过输入进行计算。

反向过程则是和正向扩散过程正好相反，是一个不断去噪的过程。将高斯噪声图像 $\boldsymbol{x}_T$ 逐步去噪，最终还原出与输入数据分布接近的数据分布，即重构出图像 $\boldsymbol{x}_0$。反向过程可视为由噪声预测网络拟合参数 θ 的可学习高斯变换。该过程也是一个马尔可夫链，根据当前

时间步和图像数据，对噪声进行采样，进而得到前一时间步的图像数据。具体地，可分为 T 时刻的初始采样：

$$p(\boldsymbol{x}_T) = \mathcal{N}(\boldsymbol{x}_T; \mathbf{0}, \boldsymbol{I}) \tag{8.5}$$

与其他时刻的迭代采样：

$$p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) = \mathcal{N}(\boldsymbol{x}_{t-1}; \mu_\theta(\boldsymbol{x}_t, t), \sigma_t^2 \boldsymbol{I}) \tag{8.6}$$

其中，$\mu_\theta(\boldsymbol{x}_t, t)$ 与 σ_t 分别为 $\boldsymbol{x}_{t-1}$ 的均值与标准差，θ 为噪声预测网络的参数。整个反向过程的联合分布表示为：

$$p_\theta(\boldsymbol{x}_{0:T}) = p(\boldsymbol{x}_T) \prod_{t=1}^{T} p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) \tag{8.7}$$

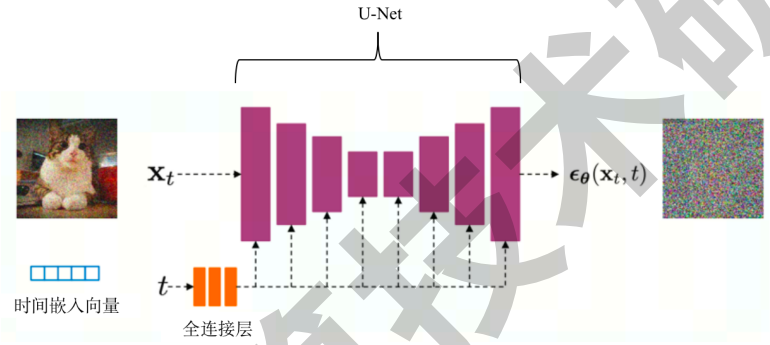

图 8.2　扩散模型噪声预测网络[30]

扩散模型训练的本质在于构建一个噪声预测网络，期望在反向过程的每一步预测合理的噪声参数，从而使采样数据的分布与训练数据分布保持一致。噪声预测网络的结构如图8.2所示，通常使用 U-Net 架构。训练过程如算法8.1所示，其中每次迭代可细分为如下几步：（1）获取数据集中的真实图像 $\boldsymbol{x}_0$，从均匀分布 $\mathcal{U}(1, T)$ 中采样时间步 t，并从标准高斯分布 $\mathcal{N}(0, \boldsymbol{I})$ 中采样需要预测的噪声 ϵ；（2）通过公式(8.4)计算 $\boldsymbol{x}_t$；（3）将 $\boldsymbol{x}_t$ 与 t 输入到模型（如图8.2）去拟合噪声 ϵ，并通过梯度下降最小化损失函数 ∇_θ 更新网络参数。训练过程将不断重复以上一系列操作，直至网络收敛为止。

算法 8.1 训练过程[29]

1: **repeat**
2: $\quad \boldsymbol{x}_0 \sim q(\boldsymbol{x}_0)$
3: $\quad t \sim \mathcal{U}(1, T)$
4: $\quad \epsilon \sim \mathcal{N}(0, \boldsymbol{I})$
5: $\quad$ 进行梯度下降 $\nabla_\theta ||\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)||^2$
6: **until** 收敛

网络训练完成后，在实际推理阶段，仅执行反向过程，将从时间步 T 开始，按算法8.2逐步向前采样生成图片。具体地，结合公式(8.5)，开始从标准正态分布中生成噪声 $\boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$；

算法 8.2 采样过程[29]

1: $\boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$
2: **for** $t = T, ..., 1$ **do**
3: $\quad z \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ 如果 $t > 1$, 否则 $z = \mathbf{0}$
4: $\quad \boldsymbol{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\boldsymbol{x}_t, t)\right) + \sigma_t z$
5: **end for**
6: **return** $\boldsymbol{x}_0$

然后对于每个时间步 t，将获取的图像 $\boldsymbol{x}_t$ 与 t 输入到训练好的噪声预测网络模型中预测噪声 $\epsilon_\theta(\boldsymbol{x}_t, t)$；结合公式(8.6)，计算时间步 t 对应的 $\boldsymbol{x}_{t-1}$：

$$\boldsymbol{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\boldsymbol{x}_t, t)\right) + \sigma_t z \tag{8.8}$$

不断重复上述过程，即可得到生成图像 $\boldsymbol{x}_0$。

值得注意的是，DDPM 不能跨步运算，需要一步一步用网络进行推理。因此，生成图像的效率很低。为了解决这个问题，[31] 对推理过程进行了加速。DDIM 假设采样分布也是一个高斯分布，其均值为关于 $\boldsymbol{x}_0$，$\boldsymbol{x}_t$ 的线性函数，即：

$$\boldsymbol{x}_{prev} = k\boldsymbol{x}_0 + m\boldsymbol{x}_t + \sigma\epsilon, \quad \epsilon \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}) \tag{8.9}$$

将公式8.4带入，可知

$$\boldsymbol{x}_{prev} = (k + m\sqrt{\bar{\alpha}_t})\boldsymbol{x}_0 + \epsilon', \epsilon' \sim \mathcal{N}(\mathbf{0}, m^2(1-\bar{\alpha}_t) + \sigma) \tag{8.10}$$

另外，由公式8.4可知 $\boldsymbol{x}_{prev} = \sqrt{\bar{\alpha}_{prev}}\boldsymbol{x}_0 + \sqrt{(1-\bar{\alpha}_{prev})}\epsilon$, 二者满足对应系数相同，求解后带入公式有：

$$\boldsymbol{x}_{prev} = \sqrt{\bar{\alpha}_{prev}}\left(\frac{\boldsymbol{x}_t - \sqrt{1-\bar{\alpha}_t}\epsilon_t}{\bar{\alpha}_t}\right) + \sqrt{1-\bar{\alpha}_{prev} - \sigma}\epsilon_t + \sigma\epsilon \tag{8.11}$$

其中，$prev$ 和 t 之间可以相隔多步。一般来说，DDPM 需要 1000 步采样，而 DDIM 则只需 50 步就可以完成推理。

8.1.2.2 Stable Diffusion

Stable Diffusion 模型的整体框架如图8.3所示，其主体就是8.1.2.1所述的扩散模型。另外，扩散模型的噪声预测网络输入还包含多模态的条件引导。无论训练还是推理，常规扩散模型的正向过程和反向过程均在像素空间操作，而 Stable Diffusion 则是在将这些过程压缩在低维的潜在隐空间，这样大大降低了计算复杂性以及显存占用，并更容易进行跨模态的迁移。Stable Diffusion 首先需要训练一个自编码模型，包括能将图像压缩到隐空间的图像嵌入的编码器，以及一个能将图像嵌入恢复到像素空间图像的解码器。自编码模型分析与提取图像中的高维特征，并将其压缩到低维空间，随后，在隐空间上进行扩散模型的训练和推理。特别的，Stable Diffusion 在传统扩散模型上进行了两点改动。第一，设计了领域专用编码器来预处理多模态条件，从而方便引入各种模态的条件（如文本、参考图像、分割图等）来控制图片生成。第二，在 U-Net 网络中加入了交叉注意力层，通过注意力映射将引导信

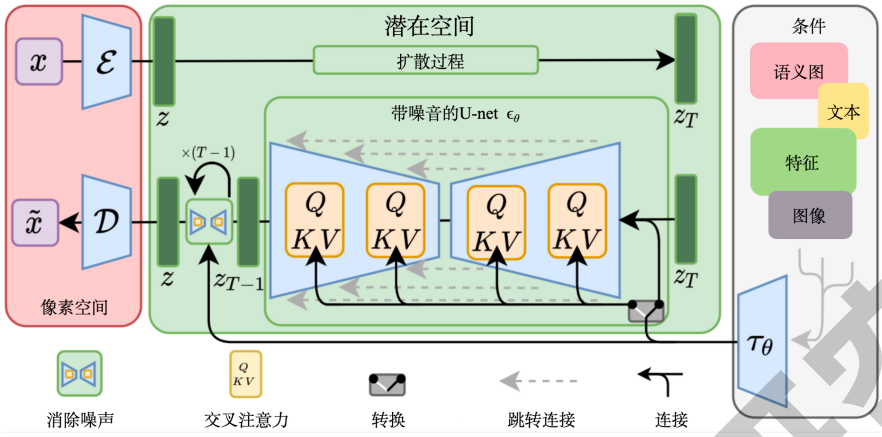

图 8.3　Stable Diffusion 模型[28]

息融入到 U-Net 的中间层，从而使多模态条件参与扩散过程。具体的，交叉注意力层的实现如下：

$$\begin{aligned} &\text{Attention}(Q,K,V) = \text{softmax}(\frac{QK^T}{\sqrt{d}}) \cdot V \\ &Q = W_Q \cdot \phi(z_t), K = W_K \cdot \tau(\boldsymbol{y}), V = W_V \cdot \tau(\boldsymbol{y}) \end{aligned} \tag{8.12}$$

其中 z_t 是 t 时刻的扩散向量，$\phi(z_t)$ 是 U-Net 的中间层特征，$\boldsymbol{y}$ 是输入的条件，$\tau(\boldsymbol{y})$ 是经过领域专用编码器处理后的控制信息。

得益于此，通过简单的提示，Stable Diffusion 已具备在消费级 GPU 上以 10 秒量级的时间生成高质量图片的能力。这大大降低了技术门槛和应用成本，一经推出就对传统的绘画、图像编辑等相关行业带来了巨大的冲击。

8.1.3　实验环境

硬件平台：DLP 云平台环境。

软件环境：编程框架 PyTorch1.6.0、CNNL 高性能 AI 运算库，CNRT 运行时库，以及 python 环境及相关的扩展库。

8.1.4　实验内容

本实验使用 Stable Diffusion 模型实现图生图、图生文以及图像修复功能。在工程实现中，首先依据推理阶段的底层逻辑，实现潜在空间反向过程的特征采样；然后根据潜在扩散模型的基本原理，补全推理阶段的关键步骤；最后完成包含模型加载等一系列的基本适配工作，实现由图像生成图像、由文字生成图像、图像修复等功能。由于本实验只涉及 Stable Diffusion 模型的推理过程，因此本实验并不包括潜在扩散模型噪声预测网络的训练部分。另外，本实验暂不涉及文字生成图像推理过程所采用的采样算法，有兴趣的可进行额外研究。

8.1.5 实验步骤

8.1.5.1 扩散模型 DDIM 采样模块

扩散模型 DDIM 采样模块的代码包含了扩散模型 DDIM 正向过程与逆向过程所有涉及到的采样操作，具体运算公式在8.1.2.1有详细介绍。其中，Stable Diffusion 在推理阶段将会调用 *stochastic_encode*、*decode*、*p_sample_ddim* 三类函数以及 *make_schedule* 中定义的运算操作，其他函数将会在训练阶段调用，本实验将暂不涉及。

1. **make schedule 运算模块**

make_schedule 运算模块对扩散模型采样过程中所需用到的一系列基础运算进行了定义。在训练与推理阶段，扩散模型 DDIM 的大部分中间运算将直接调用此模块的运算定义。值得一提的是，此模块中基础运算的定义与名称直接相关。

代码示例 8.1　make schedule 运算模块

```
# file: stable_diffusion/ldm/models/diffusion/ddim.py
def make_schedule(self, ddim_num_steps, ddim_discretize="uniform", ddim_eta=0.,erbose=True):
    self.ddim_timesteps = make_ddim_timesteps(ddim_discr_method=ddim_discretize, num_ddim_timesteps=
     ddim_num_steps,num_ddpm_timesteps=self.ddpm_num_timesteps,verbose=verbose)
    alphas_cumprod = self.model.alphas_cumprod
    assert alphas_cumprod.shape[0] == self.ddpm_num_timesteps, 'alphas have to be defined for each
     timestep'
    to_torch = lambda x: x.clone().detach().to(torch.float32).to(self.model.device)

    self.register_buffer('betas', to_torch(self.model.betas))
    self.register_buffer('alphas_cumprod', to_torch(alphas_cumprod))
    self.register_buffer('alphas_cumprod_prev', to_torch(self.model.alphas_cumprod_prev))

    # calculations for diffusion q(x_t | x_{t-1}) and others
    #TODO: 定义sqrt_alphas_cumprod运算
    self.register_buffer('sqrt_alphas_cumprod', to_torch(_____________________________))
    #TODO: 定义sqrt_one_minus_alphas_cumprod运算
    self.register_buffer('sqrt_one_minus_alphas_cumprod', to_torch(_____________________))
    self.register_buffer('log_one_minus_alphas_cumprod', to_torch(np.log(1. - alphas_cumprod.cpu())))
    self.register_buffer('sqrt_recip_alphas_cumprod', to_torch(np.sqrt(1. / alphas_cumprod.cpu())))
    self.register_buffer('sqrt_recipm1_alphas_cumprod', to_torch(np.sqrt(1. / alphas_cumprod.cpu() -
     1)))

    # ddim sampling parameters
    ddim_sigmas, ddim_alphas, ddim_alphas_prev = make_ddim_sampling_parameters(alphacums=
     alphas_cumprod.cpu(),ddim_timesteps=self.ddim_timesteps,eta=ddim_eta,verbose=verbose)
    #TODO: 定义ddim_sigmas
    self.register_buffer('ddim_sigmas', ___________)
    #TODO: 定义ddim_alphas
    self.register_buffer('ddim_alphas', ___________)
    #TODO: 定义ddim_alphas_prev
    self.register_buffer('ddim_alphas_prev', _________________)
    #TODO: 定义ddim_sqrt_one_minus_alphas运算
    self.register_buffer('ddim_sqrt_one_minus_alphas', ____________________________)
    sigmas_for_original_sampling_steps = ddim_eta * torch.sqrt((1 - self.alphas_cumprod_prev) / (1 -
     self.alphas_cumprod) * (1 - self.alphas_cumprod / self.alphas_cumprod_prev))
    self.register_buffer('ddim_sigmas_for_original_num_steps', sigmas_for_original_sampling_steps)
```

2. **stochastic_encode 模块**

该模块基于公式8.4获取了推理阶段扩散模型噪声预测网络的输入样本。

代码示例 8.2　stochastic_encode 模块

```python
# file: stable_diffusion/ldm/models/diffusion/ddim.py
def stochastic_encode(self, x0, t, use_original_steps=False, noise=None):
        # fast, but does not allow for exact reconstruction
        # t serves as an index to gather the correct alphas
        if use_original_steps:
            sqrt_alphas_cumprod = self.sqrt_alphas_cumprod
            sqrt_one_minus_alphas_cumprod = self.sqrt_one_minus_alphas_cumprod
        else:
            sqrt_alphas_cumprod = torch.sqrt(self.ddim_alphas)
            sqrt_one_minus_alphas_cumprod = self.ddim_sqrt_one_minus_alphas

        if noise is None:
            noise = torch.randn_like(x0)
        #TODO: 根据公式8.4计算xt并返回
        ______ (extract_into_tensor(____________________, t, x0.shape) * __ +
                extract_into_tensor(_______________________________, t, x0.shape) * ______)
```

3. p_sample_ddim 模块

该模块基于公式8.11实现扩散模型在反向过程的单步采样。

代码示例 8.3　p_sample_ddim 模块

```python
# file: stable_diffusion/ldm/models/diffusion/ddim.py
p_sample_ddim(self, x, c, t, index, repeat_noise=False, use_original_steps=False, quantize_denoised=
    False, temperature=1., noise_dropout=0., score_corrector=None, corrector_kwargs=None,
    unconditional_guidance_scale=1., unconditional_conditioning=None, dynamic_threshold=None):
    b, *_, device = *x.shape, x.device

    if unconditional_conditioning is None or unconditional_guidance_scale == 1.:
        model_output = self.model.apply_model(x, t, c)
    else:
        x_in = torch.cat([x] * 2)
        t_in = torch.cat([t] * 2)
        if isinstance(c, dict):
            assert isinstance(unconditional_conditioning, dict)
            c_in = dict()
            for k in c:
                if isinstance(c[k], list):
                    c_in[k] = [torch.cat([
                        unconditional_conditioning[k][i],
                        c[k][i]]) for i in range(len(c[k]))]
                else:
                    c_in[k] = torch.cat([
                            unconditional_conditioning[k],
                            c[k]])
        elif isinstance(c, list):
            c_in = list()
            assert isinstance(unconditional_conditioning, list)
            for i in range(len(c)):
                c_in.append(torch.cat([unconditional_conditioning[i], c[i]]))
        else:
            c_in = torch.cat([unconditional_conditioning, c])
        model_uncond, model_t = self.model.apply_model(x_in, t_in, c_in).chunk(2)
        model_output = model_uncond + unconditional_guidance_scale * (model_t - model_uncond)

    if self.model.parameterization == "v":
        e_t = self.model.predict_eps_from_z_and_v(x, t, model_output)
    else:
        e_t = model_output

    if score_corrector is not None:
```

```
        assert self.model.parameterization == "eps", 'not implemented'
        e_t = score_corrector.modify_score(self.model, e_t, x, t, c, **corrector_kwargs)

    alphas = self.model.alphas_cumprod if use_original_steps else self.ddim_alphas
    alphas_prev = self.model.alphas_cumprod_prev if use_original_steps else self.ddim_alphas_prev
    sqrt_one_minus_alphas = self.model.sqrt_one_minus_alphas_cumprod if use_original_steps else self.
     ddim_sqrt_one_minus_alphas
    sigmas = self.model.ddim_sigmas_for_original_num_steps if use_original_steps else self.ddim_sigmas
    # select parameters corresponding to the currently considered timestep
    a_t = torch.full((b, 1, 1, 1), alphas[index], device=device)
    a_prev = torch.full((b, 1, 1, 1), alphas_prev[index], device=device)
    sigma_t = torch.full((b, 1, 1, 1), sigmas[index], device=device)
    sqrt_one_minus_at = torch.full((b, 1, 1, 1), sqrt_one_minus_alphas[index],device=device)

    # current prediction for x_0
    if self.model.parameterization != "v":
    #TODO: 根据公式8.11计算pred_x0
        pred_x0 = _____ / _____
    else:
        pred_x0 = self.model.predict_start_from_z_and_v(x, t, model_output)

    if quantize_denoised:
        pred_x0, _, *_ = self.model.first_stage_model.quantize(pred_x0)

    if dynamic_threshold is not None:
        raise NotImplementedError()

    # direction pointing to x_t
    #TODO: 根据公式8.11计算dir_xt
    dir_xt = _____ ######################
    #TODO: 根据公式8.11计算noise
    noise = _____noise_like(x.shape, device, repeat_noise) * temperature
    if noise_dropout > 0.:
        noise = torch.nn.functional.dropout(noise, p=noise_dropout)
    #TODO: 根据公式8.11计算x_prev
    x_prev = _____ + dir_xt + noise      ####################
    return x_prev, pred_x0
```

4. decode 模块

该模块将循环调用采样函数 *p_sample_ddim*, 实现扩散模型的反向过程。

代码示例 8.4　decode 模块

```
# file: stable_diffusion/ldm/models/diffusion/ddim.py
def decode(self, x_latent, cond, t_start, unconditional_guidance_scale=1.0,unconditional_conditioning=
    None,use_original_steps=False, callback=None):
    #TODO: 配置时间步序列,若use_original_steps则选择ddpm的时间步序列, 否则使用ddim的时间步序列
    timesteps = _____
    timesteps = timesteps[:t_start]

    time_range = np.flip(timesteps)
    total_steps = timesteps.shape[0]
    print(f"Running DDIM Sampling with {total_steps} timesteps")

    iterator = tqdm(time_range, desc='Decoding image', total=total_steps)
    x_dec = x_latent
    for i, step in enumerate(iterator):
        index = total_steps - i - 1
        ts = torch.full((x_latent.shape[0],), step, device=x_latent.device, dtype=torch.long)
        #TODO: 调用采样函数p_sample_ddim
        x_dec, _ = _____
        if callback: callback(i)
```

```
    return x_dec
```

8.1.5.2 图生图主函数模块

图生图主函数模块的代码如 8.5所示。这段代码包含了扩散模型在推理阶段的关键步骤以及包含模型参数等一系列基本配置。其中，parser 部分包含了模型的初始参数配置及相关提示。

代码示例 8.5 图生图主函数模块

```
# file: stable_diffusion/scripts/img2img.py
def main():
    parser = argparse.ArgumentParser()

    #TODO:设置prompt参数为 "A fantasy landscape, trending on artstation"
    parser.add_argument("______", type=str, nargs="?", default="
     ___________________________________________", help="the prompt to render")
    parser.add_argument("--init-img", type=str, nargs="?", help="path to the input image")
    parser.add_argument("--outdir", type=str, nargs="?", help="dir to write results to", default="
     outputs/img2img-samples")
    parser.add_argument("--ddim_steps", type=int, default=50, help="number of ddim sampling steps")
    parser.add_argument("--fixed_code", action='store_true', help="if enabled, uses the same starting
     code across all samples ")
    parser.add_argument("--ddim_eta", type=float, default=0.0, help="ddim eta (eta=0.0 corresponds to
     deterministic sampling")
    parser.add_argument("--n_iter", type=int, default=1, help="sample this often")
    parser.add_argument("--C", type=int, default=4, help="latent channels")
    parser.add_argument("--f", type=int, default=8, help="downsampling factor, most often 8 or 16")
    parser.add_argument("--n_samples", type=int, default=2, help="how many samples to produce for each
      given prompt. A.k.a batch size")
    parser.add_argument("--n_rows", type=int, default=0, help="rows in the grid (default: n_samples)")
    parser.add_argument("--scale", type=float, default=9.0, help="unconditional guidance scale: eps =
     eps(x, empty) + scale * (eps(x, cond) - eps(x, empty))")
    parser.add_argument("--strength", type=float, default=0.8, help="strength for noising/unnoising.
     1.0 corresponds to full destruction of information in init image")
    parser.add_argument("--from-file", type=str, help="if specified, load prompts from this file")
    #TODO:设置config参数为configs/stable-diffusion/v2-inference-v.yaml
    parser.add_argument("______", type=str, default="_____________________________________________", help
     ="path to config which constructs model")
    #TODO:设置ckpt参数为./models/v2-1_768-nonema-pruned.ckpt
    parser.add_argument("______", type=str, default="_____________________________________________", help
     ="path to checkpoint of model")
    parser.add_argument("--seed", type=int, default=42, help="the seed (for reproducible sampling)")
    parser.add_argument("--precision", type=str, help="evaluate at this precision", choices=["full", "
     autocast"], default="autocast")

    opt = parser.parse_args()
    seed_everything(opt.seed)

    config = OmegaConf.load(f"{opt.config}")
    model = load_model_from_config(config, f"{opt.ckpt}")

    device = torch.device("mlu") if torch.mlu.is_available() else torch.device("cpu")
    model = model.to(device)

    sampler = DDIMSampler(model)

    os.makedirs(opt.outdir, exist_ok=True)
    outpath = opt.outdir

```

```
    print("Creating invisible watermark encoder (see https://github.com/ShieldMnt/invisible-watermark)
      ...")
    wm = "SDV2"
    wm_encoder = WatermarkEncoder()
    wm_encoder.set_watermark('bytes', wm.encode('utf-8'))

    batch_size = opt.n_samples
    n_rows = opt.n_rows if opt.n_rows > 0 else batch_size
    if not opt.from_file:
        prompt = opt.prompt
        assert prompt is not None
        data = [batch_size * [prompt]]

    else:
        print(f"reading prompts from {opt.from_file}")
        with open(opt.from_file, "r") as f:
            data = f.read().splitlines()
            data = list(chunk(data, batch_size))

    sample_path = os.path.join(outpath, "samples")
    os.makedirs(sample_path, exist_ok=True)
    base_count = len(os.listdir(sample_path))
    grid_count = len(os.listdir(outpath)) - 1

    assert os.path.isfile(opt.init_img)
    #TODO:调用load_img函数对输入opt.init_img进行操作
    init_image = _____.to(device)
    init_image = repeat(init_image, '1 ... -> b ...', b=batch_size)
    #TODO:调用model.encode_first_stage函数对输入init_image进行操作，再将结果作为model.
      get_first_stage_encoding函数的输入得到初始潜在特征
    init_latent = _____  # move to latent space

    sampler.make_schedule(ddim_num_steps=opt.ddim_steps, ddim_eta=opt.ddim_eta, verbose=False)

    assert 0. <= opt.strength <= 1., 'can only work with strength in [0.0, 1.0]'
    t_enc = int(opt.strength * opt.ddim_steps)
    print(f"target t_enc is {t_enc} steps")

    precision_scope = autocast if opt.precision == "autocast" else nullcontext
    with torch.no_grad():
        with precision_scope(True):
            with model.ema_scope():
                all_samples = list()
                for n in trange(opt.n_iter, desc="Sampling"):
                    for prompts in tqdm(data, desc="data"):
                        uc = None
                        if opt.scale != 1.0:
                            uc = model.get_learned_conditioning(batch_size * [""])
                        if isinstance(prompts, tuple):
                            prompts = list(prompts)
                        #TODO:调用model.get_learned_conditioning函数对prompts进行操作得到约束条件
                        c = _____

                        # encode (scaled latent)
                        #TODO:调用sampler.stochastic_encode函数对输入初始潜在特征进行操作得到约束条件
                        z_enc = _____(_____, torch.tensor([t_enc] * batch_size).to(device))
                        # decode it
                        #TODO:调用sampler.decode函数，输入包括t_enc、uc、c、z_enc
                        samples =_____, unconditional_guidance_scale=opt.scale)

                        #TODO:调用model.decode_first_stage函数对输入samples进行操作
                        x_samples = _____
```

```
                        x_samples = torch.clamp((x_samples + 1.0) / 2.0, min=0.0, max=1.0)

                        for x_sample in x_samples:
                            x_sample = 255. * rearrange(x_sample.cpu().numpy(), 'c h w -> h w c')
                            img = Image.fromarray(x_sample.astype(np.uint8))
                            img = put_watermark(img, wm_encoder)
                            img.save(os.path.join(sample_path, f"{base_count:05}.png"))
                            base_count += 1
                        all_samples.append(x_samples)

                # additionally, save as grid
                grid = torch.stack(all_samples, 0)
                grid = rearrange(grid, 'n b c h w -> (n b) c h w')
                grid = make_grid(grid, nrow=n_rows)

                # to image
                grid = 255. * rearrange(grid, 'c h w -> h w c').cpu().numpy()
                grid = Image.fromarray(grid.astype(np.uint8))
                grid = put_watermark(grid, wm_encoder)
                grid.save(os.path.join(outpath, f'grid-{grid_count:04}.png'))
                grid_count += 1

    print(f"Your samples are ready and waiting for you here: \n{outpath} \nEnjoy.")
```

8.1.5.3 文生图主函数模块

文生图主函数模块的代码如 8.6所示。这段代码与图生图的代码基本流程相似。

代码示例 8.6 图生图主函数模块

```
# file: stable_diffusion/scripts/txt2img.py
def main():
    parser = argparse.ArgumentParser()
    #TODO:设置prompt参数为 "a professional photograph of an astronaut riding a triceratops"
    parser.add_argument("_____", type=str, nargs="?", default="
     ____________________________________", help="the prompt to render")
    parser.add_argument("--outdir", type=str, nargs="?", help="dir to write results to", default="
     outputs/txt2img-samples")
    parser.add_argument("--steps", type=int, default=50, help="number of ddim sampling steps")
    #TODO:设置参数使得opt.plms为"true"
    parser.add_argument("_____", ______='_____', help="use plms sampling")
    parser.add_argument("--fixed_code", action='store_true', help="if enabled, uses the same starting
     code across all samples ")
    parser.add_argument("--ddim_eta", type=float, default=0.0, help="ddim eta (eta=0.0 corresponds to
     deterministic sampling")
    parser.add_argument("--n_iter", type=int, default=3, help="sample this often")
    #TODO:设置代表image height的参数H为512
    parser.add_argument("______", type=int, default=______, help="image height, in pixel space")
    #TODO:设置代表image width的参数H为512
    parser.add_argument("______", type=int, default=______, help="image width, in pixel space")
    parser.add_argument("--C", type=int, default=4, help="latent channels")
    parser.add_argument("--f", type=int, default=8, help="downsampling factor, most often 8 or 16")
    parser.add_argument("--n_samples", type=int, default=3, help="how many samples to produce for each
      given prompt. A.k.a batch size")
    parser.add_argument("--n_rows", type=int, default=0, help="rows in the grid (default: n_samples)")
    parser.add_argument("--scale", type=float, default=9.0, help="unconditional guidance scale: eps =
     eps(x, empty) + scale * (eps(x, cond) - eps(x, empty))")
    parser.add_argument("--strength", type=float, default=0.8, help="strength for noising/unnoising.
     1.0 corresponds to full destruction of information in init image")
    parser.add_argument("--from-file", type=str, help="if specified, load prompts from this file")
    #TODO:设置config参数为configs/stable-diffusion/v2-inference-v.yaml
```

```
    parser.add_argument("______", type=str, default="_____________________________________________", help
    ="path to config which constructs model")
    #TODO:设置ckpt参数为 /models/stable_diffusion/models/v2-1_768-nonema-pruned.ckpt
    parser.add_argument("______", type=str, default="_____________________________________________", help
    ="path to checkpoint of model")
    parser.add_argument("--seed", type=int, default=42, help="the seed (for reproducible sampling)")
    parser.add_argument("--precision", type=str, help="evaluate at this precision", choices=["full", "
    autocast"], default="autocast")
    parser.add_argument("--repeat", type=int, default=1, help="repeat each prompt in file this often")
    #TODO:设置程序运行的设备为"mlu"
    parser.add_argument("______", type=str, help="Device on which Stable Diffusion will be run",
    choices=["cpu", "mlu"], ____________ )
    parser.add_argument("--torchscript", action='store_true', help="Use TorchScript")
    parser.add_argument("--ipex", action='store_true', help="Use Intel® Extension for PyTorch*")
    parser.add_argument("--bf16", action='store_true', help="Use bfloat16")
    opt = parser.parse_args()
    seed_everything(opt.seed)

    config = OmegaConf.load(f"{opt.config}")
    device = torch.device("mlu") if opt.device == "mlu" else torch.device("cpu")
    model = load_model_from_config(config, f"{opt.ckpt}", device)
    #TODO:假如opt.plms为真，设置采样算法sampler为PLMSSampler；反之假如opt.dpm为真，设置采样算法为
    DPMSolverSampler；\
    #否则设置采样算法为DDIMSampler,所有算法输入为model，执行设备device为device
    ____________
    ________________________________________
    ____________
    ________________________________________
    ____________
    ________________________________________

    os.makedirs(opt.outdir, exist_ok=True)
    outpath = opt.outdir

    print("Creating invisible watermark encoder (see https://github.com/ShieldMnt/invisible-watermark)
    ...")
    wm = "SDV2"
    wm_encoder = WatermarkEncoder()
    wm_encoder.set_watermark('bytes', wm.encode('utf-8'))

    batch_size = opt.n_samples
    n_rows = opt.n_rows if opt.n_rows > 0 else batch_size
    if not opt.from_file:
        prompt = opt.prompt
        assert prompt is not None
        data = [batch_size * [prompt]]

    else:
        print(f"reading prompts from {opt.from_file}")
        with open(opt.from_file, "r") as f:
            data = f.read().splitlines()
            data = [p for p in data for i in range(opt.repeat)]
            data = list(chunk(data, batch_size))

    sample_path = os.path.join(outpath, "samples")
    os.makedirs(sample_path, exist_ok=True)
    sample_count = 0
    base_count = len(os.listdir(sample_path))
    grid_count = len(os.listdir(outpath)) - 1

    start_code = None
    if opt.fixed_code:
```

```
        start_code = torch.randn([opt.n_samples, opt.C, opt.H // opt.f, opt.W // opt.f], device=device
    )

    if opt.torchscript or opt.ipex:
        transformer = model.cond_stage_model.model
        unet = model.model.diffusion_model
        decoder = model.first_stage_model.decoder
        additional_context = torch.cpu.amp.autocast() if opt.bf16 else nullcontext()
        shape = [opt.C, opt.H // opt.f, opt.W // opt.f]

        if opt.bf16 and not opt.torchscript and not opt.ipex:
            raise ValueError('Bfloat16 is supported only for torchscript+ipex')
        if opt.bf16 and unet.dtype != torch.bfloat16:
            raise ValueError("Use configs/stable-diffusion/intel/ configs with bf16 enabled if " +
                             "you'd like to use bfloat16 with CPU.")
        if unet.dtype == torch.float16 and device == torch.device("cpu"):
            raise ValueError("Use configs/stable-diffusion/intel/ configs for your model if you'd like
     to run it on CPU.")

        if opt.ipex:
            import intel_extension_for_pytorch as ipex
            bf16_dtype = torch.bfloat16 if opt.bf16 else None
            transformer = transformer.to(memory_format=torch.channels_last)
            transformer = ipex.optimize(transformer, level="O1", inplace=True)

            unet = unet.to(memory_format=torch.channels_last)
            unet = ipex.optimize(unet, level="O1", auto_kernel_selection=True, inplace=True, dtype=
    bf16_dtype)

            decoder = decoder.to(memory_format=torch.channels_last)
            decoder = ipex.optimize(decoder, level="O1", auto_kernel_selection=True, inplace=True,
    dtype=bf16_dtype)

        if opt.torchscript:
            with torch.no_grad(), additional_context:
                # get UNET scripted
                if unet.use_checkpoint:
                    raise ValueError("Gradient checkpoint won't work with tracing. " +
                    "Use configs/stable-diffusion/intel/ configs for your model or disable checkpoint
    in your config.")

                img_in = torch.ones(2, 4, 96, 96, dtype=torch.float32)
                t_in = torch.ones(2, dtype=torch.int64)
                context = torch.ones(2, 77, 1024, dtype=torch.float32)
                scripted_unet = torch.jit.trace(unet, (img_in, t_in, context))
                scripted_unet = torch.jit.optimize_for_inference(scripted_unet)
                print(type(scripted_unet))
                model.model.scripted_diffusion_model = scripted_unet

                # get Decoder for first stage model scripted
                samples_ddim = torch.ones(1, 4, 96, 96, dtype=torch.float32)
                scripted_decoder = torch.jit.trace(decoder, (samples_ddim))
                scripted_decoder = torch.jit.optimize_for_inference(scripted_decoder)
                print(type(scripted_decoder))
                model.first_stage_model.decoder = scripted_decoder

        prompts = data[0]
        print("Running a forward pass to initialize optimizations")
        uc = None
        if opt.scale != 1.0:
            uc = model.get_learned_conditioning(batch_size * [""])
        if isinstance(prompts, tuple):
```

```
            prompts = list(prompts)

        with torch.no_grad(), additional_context:
            for _ in range(3):
                #TODO:调用model.get_learned_conditioning函数对prompts进行操作得到约束条件
                c = ____________________________________
            samples_ddim, _ = sampler.sample(S=5,
                                             conditioning=c,
                                             batch_size=batch_size,
                                             shape=shape,
                                             verbose=False,
                                             unconditional_guidance_scale=opt.scale,
                                             unconditional_conditioning=uc,
                                             eta=opt.ddim_eta,
                                             x_T=start_code)
            print("Running a forward pass for decoder")
            for _ in range(3):
                #TODO:调用model.decode_first_stage函数对输入samples_ddim进行操作
                x_samples_ddim = ___________________________________

    precision_scope = autocast if opt.precision=="autocast" or opt.bf16 else nullcontext
    import time
    with torch.no_grad(), \
        precision_scope(True), \
        model.ema_scope():
            all_samples = list()
            for n in trange(opt.n_iter, desc="Sampling"):
                for prompts in tqdm(data, desc="data"):
                    uc = None
                    if opt.scale != 1.0:
                        uc = model.get_learned_conditioning(batch_size * [""])
                    if isinstance(prompts, tuple):
                        prompts = list(prompts)
                    start = time.time()
                    #TODO:调用model.get_learned_conditioning函数对prompts进行操作得到约束条件
                    c = ____________________________________
                    shape = [opt.C, opt.H // opt.f, opt.W // opt.f]
                    samples, _ = sampler.sample(S=opt.steps,
                                                conditioning=c,
                                                batch_size=opt.n_samples,
                                                shape=shape,
                                                verbose=False,
                                                unconditional_guidance_scale=opt.scale,
                                                unconditional_conditioning=uc,
                                                eta=opt.ddim_eta,
                                                x_T=start_code)
                    #TODO:调用model.decode_first_stage函数对输入samples进行操作
                    x_samples = ___________________________________
                    print("first stage bs: {} time: {}", opt.n_samples, time.time() - start)
                    x_samples = torch.clamp((x_samples + 1.0) / 2.0, min=0.0, max=1.0)

                    for x_sample in x_samples:
                        x_sample = 255. * rearrange(x_sample.cpu().numpy(), 'c h w -> h w c')
                        img = Image.fromarray(x_sample.astype(np.uint8))
                        img = put_watermark(img, wm_encoder)
                        img.save(os.path.join(sample_path, f"{base_count:05}.png"))
                        base_count += 1
                        sample_count += 1

                    all_samples.append(x_samples)

            # additionally, save as grid
```

```
            grid = torch.stack(all_samples, 0)
            grid = rearrange(grid, 'n b c h w -> (n b) c h w')
            grid = make_grid(grid, nrow=n_rows)

            # to image
            grid = 255. * rearrange(grid, 'c h w -> h w c').cpu().numpy()
            grid = Image.fromarray(grid.astype(np.uint8))
            grid = put_watermark(grid, wm_encoder)
            grid.save(os.path.join(outpath, f'grid-{grid_count:04}.png'))
            grid_count += 1

    print(f"Your samples are ready and waiting for you here: \n{outpath} \n"
          f" \nEnjoy.")
```

8.1.5.4 图像修复模块

图像修复函数的代码示例如8.7所示。该程序提供一个交互式界面，用户可以上传图像并输入文本提示，然后点击按钮执行图像修复操作。put_watermark 函数用于添加水印到图像中；initialize_model 函数用于初始化稳定扩散模型；make_batch_sd 函数用于批处理的图像、掩码和文本数据。inpaint 函数用于执行图像修复操作，pad_image 函数用于对输入图像进行填充，使其尺寸成为 32 的整数倍。predict 函数用于执行图像修复的预测功能。整个程序基于 Gradio 库构建了一个交互式界面，使用 gr.Button、gr.Slider 和 gr.Image 等组件来实现用户界面的设计和交互操作。

代码示例 8.7　图像修复模块

```
#file: stable_diffusion/scripts/gradio/inpainting.py
import sys
import cv2
import torch
import torch_mlu
import numpy as np
import gradio as gr
from PIL import Image
from omegaconf import OmegaConf
from einops import repeat
from imwatermark import WatermarkEncoder
from pathlib import Path

import os
sys.path.append(os.path.join(os.path.dirname(__file__), "../.."))
from ldm.models.diffusion.ddim import DDIMSampler
from ldm.util import instantiate_from_config


torch.set_grad_enabled(False)


def put_watermark(img, wm_encoder=None):
    if wm_encoder is not None:
        #TODO: 将图片从RGB格式转换为BGR格式（OpenCV默认格式）
        img = _____________________________________________
        img = wm_encoder.encode(img, 'dwtDct')
        #TODO: 将编码后的图片数组转换为PIL格式，并转换为RGB格式
        img = _____________________________________________
    return img

```

```
32
33  def initialize_model(config, ckpt):
34      config = OmegaConf.load(config)
35      model = instantiate_from_config(config.model)
36
37      #TODO: 加载模型参数
38      _____________________________________________
39      #TODO: 根据系统环境来选择运行设备,如果支持 MLU 设备, 则选择 MLU 设备, 否则选择 CPU。
40      device = _____________________________________________
41      #TODO: 将模型加载到指定设备
42      model = _____________________________________________
43      #TODO: 使用DDIMSampler对模型进行采样
44      sampler = _____________________________________________
45
46      return sampler
47
48
49  def make_batch_sd(
50          image,
51          mask,
52          txt,
53          device,
54          num_samples=1):
55      #TODO: 将输入图片转换为RGB格式的NumPy数组
56      image =_____________________________________________
57      #TODO: 添加一个维度, 并将维度顺序从HWC转换为BCHW
58      image = _____________________________________________
59      #TODO: 将NumPy数组转换为PyTorch张量, 进行归一化处理, 使像素值范围在[-1, 1]之间
60      image = _____________________________________________
61
62      mask = np.array(mask.convert("L"))
63      #TODO: 将掩码数组转换为浮点数并归一化处理, 使像素值范围在[0, 1]之间
64      mask = _____________________________________________
65      mask = mask[None, None]
66      #TODO: 将掩码数组中小于0.5的像素值设为0, 表示不需要修复的部分
67      _____________________________________________
68      #TODO: 将掩码数组中大于等于0.5的像素值设为1, 表示需要修复的部分
69      _____________________________________________
70      #TODO: 将NumPy数组转换为PyTorch张量
71      mask = _____________________________________________
72      #TODO: 使用掩码对输入图片进行掩码处理, 将不需要修复的部分设置为0
73      masked_image = _____________________________________________
74
75      batch = {
76          "image": repeat(image.to(device=device), "1 ... -> n ...", n=num_samples),
77          "txt": num_samples * [txt],
78          "mask": repeat(mask.to(device=device), "1 ... -> n ...", n=num_samples),
79          "masked_image": repeat(masked_image.to(device=device), "1 ... -> n ...", n=num_samples),
80          }
81      return batch
82
83
84  def inpaint(sampler, image, mask, prompt, seed, scale, ddim_steps, num_samples=1, w=512, h=512):
85      device = torch.device(
86          "mlu") if torch.mlu.is_available() else torch.device("cpu")
87      model = sampler.model
88
89      print("Creating invisible watermark encoder (see https://github.com/ShieldMnt/invisible-watermark)
        ...")
90      wm = "SDV2"
91      wm_encoder = WatermarkEncoder()
92      wm_encoder.set_watermark('bytes', wm.encode('utf-8'))
```

```
93
94      prng = np.random.RandomState(seed)
95      start_code = prng.randn(num_samples, 4, h // 8, w // 8)
96      #TODO: 将 NumPy 数组 start_code 转换为 PyTorch 张量，并将其发送到指定的设备上，并指定数据类型为
          torch.float32。
97      start_code = ____________________________________________
98
99
100     #TODO:关闭梯度计算，启用MLU自动混合精度
101     with ____________________________________________:
102         #TODO:调用函数构造批次数据
103         batch = ____________________________________________
104
105         c = model.cond_stage_model.encode(batch["txt"])
106
107         c_cat = list()
108         for ck in model.concat_keys:
109             cc = batch[ck].float()
110             if ck != model.masked_image_key:
111                 bchw = [num_samples, 4, h // 8, w // 8]
112                 cc = torch.nn.functional.interpolate(cc, size=bchw[-2:])
113             else:
114                 cc = model.get_first_stage_encoding(
115                     model.encode_first_stage(cc))
116             #TODO: 将处理后的数据添加到列表中
117             ____________________________________________
118         c_cat = torch.cat(c_cat, dim=1)
119
120         # cond
121         cond = {"c_concat": [c_cat], "c_crossattn": [c]}
122
123         # uncond cond
124         uc_cross = model.get_unconditional_conditioning(num_samples, "")
125         uc_full = {"c_concat": [c_cat], "c_crossattn": [uc_cross]}
126
127         shape = [model.channels, h // 8, w // 8]
128         samples_cfg, intermediates = sampler.sample(
129             ddim_steps,
130             num_samples,
131             shape,
132             cond,
133             verbose=False,
134             eta=1.0,
135             unconditional_guidance_scale=scale,
136             unconditional_conditioning=uc_full,
137             x_T=start_code,
138         )
139         x_samples_ddim = model.decode_first_stage(samples_cfg)
140
141         result = torch.clamp((x_samples_ddim + 1.0) / 2.0,
142                              min=0.0, max=1.0)
143         #TODO:将生成的图片结果从 PyTorch 张量转换为 NumPy 数组，并将维度顺序调整为常用的图像格式，最后
          将像素值恢复到原始范围
144         result = ____________________________________________
145     return [put_watermark(Image.fromarray(img.astype(np.uint8)), wm_encoder) for img in result]
146
147 def pad_image(input_image):
148     pad_w, pad_h = np.max(((2, 2), np.ceil(
149         np.array(input_image.size) / 64).astype(int)), axis=0) * 64 - input_image.size
150     im_padded = Image.fromarray(
151         np.pad(np.array(input_image), ((0, pad_h), (0, pad_w), (0, 0)), mode='edge'))
152     return im_padded
```

```
153
154 def predict(input_image, prompt, ddim_steps, num_samples, scale, seed):
155     #TODO: 从输入图像中获取原始图像，并转换为RGB模式
156     init_image = ____________________________________________________
157     #TODO: 从输入图像中获取掩码图像，并转换为RGB模式
158     init_mask = _____________________________________________________
159     #TODO: 调用函数对原始图像和掩码图像进行填充
160     image = _____________________________________________ # resize to integer multiple of 32
161     mask = ______________________________________________ # resize to integer multiple of 32
162     width, height = image.size
163     print("Inpainting...", width, height)
164     #TODO:  调用图像修复函数进行修复
165     result = ____________________________________________
166
167     return result
168
169 #TODO: 调用函数初始化模型
170 sampler = _____________________________(sys.argv[1], sys.argv[2])
171
172 block = gr.Blocks().queue()
173 with block:
174     with gr.Row():
175         gr.Markdown("## Stable Diffusion Inpainting")
176
177     with gr.Row():
178         with gr.Column():
179             input_image = gr.Image(source='upload', tool='sketch', type="pil")
180             prompt = gr.Textbox(label="Prompt")
181             run_button = gr.Button(label="Run")
182             with gr.Accordion("Advanced options", open=False):
183                 num_samples = gr.Slider(
184                     label="Images", minimum=1, maximum=4, value=4, step=1)
185                 ddim_steps = gr.Slider(label="Steps", minimum=1,
186                                        maximum=50, value=45, step=1)
187                 scale = gr.Slider(
188                     label="Guidance Scale", minimum=0.1, maximum=30.0, value=10, step=0.1
189                 )
190                 seed = gr.Slider(
191                     label="Seed",
192                     minimum=0,
193                     maximum=2147483647,
194                     step=1,
195                     randomize=True,
196                 )
197         with gr.Column():
198             gallery = gr.Gallery(label="Generated images", show_label=False).style(
199                 grid=[2], height="auto")
200     #TODO: 定义点击按钮时执行的函数，即调用 predict 函数进行图像修复
201     run_button.click(fn=_____________________________, inputs=_____________________________________, outputs
        =[gallery])
202
203 block.launch(share=True)
```

8.1.5.5 实验运行

根据第8.1.5.1节 ~ 第8.1.5.4节的描述补全 stable_diffusion/ldm/models/diffusion/ddim.py 文件、stable_diffusion/scripts/img2img.py 文件、stable_diffusion/scripts/txt2img.py 文件以及 stable_diffusion/scripts/gradio/inpainting.py 文件后，在指定目录下执行以下命令，完成基于 stable diffusion 图生图、文生图以及图像修复的实现过程。

1) 环境申请与安装

申请实验环境、安装软件包并登录云平台，本实验的代码存放在云平台/opt/code_chap_8_student 目录下。

```
# 登录云平台
ssh root@xxx.xxx.xxx.xxx -p xxxxx
# 进入/opt/tools/accelerate-0.20-release-mlu目录
cd /opt/tools/accelerate-0.20-release-mlu
#安装accelerate软件包
python setup.py install
# 进入/opt/tools/transformers-mlu-dev目录
cd /opt/tools/transformers-mlu-dev
#安装transformers软件包
python setup.py install
#进入/opt/code_chap_8_student/stable_diffusion目录
cd /opt/code_chap_8_student/stable_diffusion
```

2) 代码实现

补全 stable_diffusion/ldm/models/diffusion/ddim.py、stable_diffusion/scripts/img2img.py、stable_diffusion/scripts/txt2img.py、以及 stable_diffusion/scripts/gradio/inpainting.py 文件。

```
# 进入stable_diffusion/ldm/models/diffusion目录
cd stable_diffusion/ldm/models/diffusion
# 补全ddim.py文件
vim ddim.py
# 进入stable_diffusion/scripts目录
cd stable_diffusion/scripts
# 补全img2img.py文件
vim img2img.py
# 补全txt2img.py文件
vim txt2img.py
# 进入stable_diffusion/scripts/gradio目录
cd stable_diffusion/scripts/gradio
# 补全inpainting.py文件
vim inpainting.py
```

3) 运行实验

```
# 进入stable_diffusion目录
cd stable_diffusion
# 运行图生图推理程序
bash run-stable-diffusion-img2img.sh
# 运行文生图推理程序
bash run-stable-diffusion-txt2img.sh
# 运行图像修复推理程序
bash run-stable-diffusion-painting.sh
```

8.1.6 实验评估

- 60 分标准：能够正确实现 make_schedule 运算以及 stochastic_encode 模块，完成扩散模型推理阶段的基础运算。
- 70 分标准：在 60 分标准基础上，能够正确实现 p_sample_ddim 模块、decode 模块以及主函数模块，完成推理阶段反向过程的所有函数。
- 80 分标准：在 70 分标准基础上，能够正确实现图像生成图像模块。

- 90 分标准：在 80 分标准基础上，能够正确实现文字生成图像模块。
- 100 分标准：在 90 分标准基础上，能够正确实现图像修复推理模块。

8.1.7 实验思考

1) 补全的 ddim.py 文件中哪些函数是扩散模型训练时所用到的？其中的编码及采样函数和推理时的有何不同？

2) 图8.3所示的 stable diffusion 模型在哪些方面还能进一步改进？

3) 若实现图像超分辨任务，在推理流程上与一般的图像生成图像任务相同之处在哪？有什么不同？

4) 若需要像 sora 那样实现文字生成视频任务，扩散模型还需要关注哪些方面？

8.2 基于 Llama 2 实现聊天机器人

8.2.1 实验目的

熟悉 Llama 2 大语言模型的算法原理，掌握在 DLP 平台上移植优化聊天机器人的方法和流程，具体包括：

1) 本实验旨在向学生介绍基于大型语言模型 Llama 2 构建聊天机器人的基本原理和操作步骤，深入理解其在对话生成中的应用；

2) 学生将深入了解关键概念，包括模型加载和适配、对话模板的应用、文本生成的基本流程、模型推理的关键步骤，以及使用命令行进行聊天的具体过程；

3) 学生将了解如何在 DLP 平台上部署模型，实现人机快速聊天应用，为将来在实际场景中应用模型提供实用经验。

实验工作量：约 120 行代码，10 小时。

8.2.2 背景介绍

Llama 2[32] 是 Meta AI 于 2023 年 7 月 19 日正式发布最新一代开源可商用大模型，参数规模从 70 亿到 700 亿不等。相较于第一代 LLaMA（Large Language Model Meta AI）[33]，Llama 2 具有强大的参数规模和性能表现，被人称为 “GPT-4 最强平替”。接下来对 LLaMA 和 Llama 2 进行详细介绍。

8.2.2.1 LLaMA

LLaMA 是只使用公开数据集来训练的一系列从 7B 到 65B 参数的大语言模型，在常识推理、闭卷问答、阅读理解、数学推理、代码生成等领域均有着优异的性能表现。与其他大语言模型（LLMs）相比，具有显著的竞争优势。尽管 LLaMA-13B 的大小仅有 GPT-3 的 1/10，但在大多数基准测试中优于 GPT-3；此外，LLaMA-65B 的参数模型也展现出与 Chinchilla 或 PaLM-540B 相媲美的竞争力。

LLaMA 的网络结构基于 Transformer 中的 decoder 架构，属于 decoder-only 结构，它在 transformer 基础上进行了一系列的改进：

1）**预归一化（Pre-normalization）**：为了提高训练稳定性，LLaMa 对每个 Transformer 子层的输入进行归一化，而不是对输出进行归一化。同时，LLaMa 使用了 RMSNorm 归一化函数[34]。

2）**SwiGLU 激活函数**: 使用了 SwiGLU 激活函数[35] 替换 ReLU 激活函数以提高性能。

3）**旋转位置编码（Rotary Embeddings，RoRE）**：LLaMa 在每层网络中没有使用之前的绝对位置编码，而是使用了旋转位置编码[36]。

此外，LLaMA 在训练过程中还使用了 AdamW 优化器。为了提升模型的训练速度，LLaMA 采用了随机多头注意力机制（基于 xformers）[37] 来降低内存占用以及运行时间。为了进一步提高训练效率，通过检查点（checkpointing）减少了反向传播过程中重新计算的激活次数并采用行化技术[38] 提高训练速度。

8.2.2.2 Llama 2

Llama 2 在 LLaMA 的基础上通过引入更多的预训练数据、增大上下文长度等技术进一步提升模型的效果。Llama 2 开源了 7B、13B、70B 三种参数的模型，同时开源了同等参数的 Llama 2-Chat（在 LLaMA2 为基础上进行了对话用例上的针对性微调）模型。

Llama2 模型的训练流程主要包括预训练（Pre-training）、微调（Fine_tuning）两个过程，如图 8.4所示。其中，预训练阶段使用公开的在线资源对 Llama 2 进行预训练；微调阶段使用人类反馈的强化学习（Reinforcement Learning from Human Feedback，RLHF）方法得到 Llama 2-Chat，具体通过拒绝采样（Rejection Sampling）以及近端策略优化（Proximal Policy Optimization，PPO）策略进行微调。在 RLHF 阶段，迭代地训练 Saftety Reward Model 以及 Helpful Reward Model 两个模型对于保持模型分布至关重要。

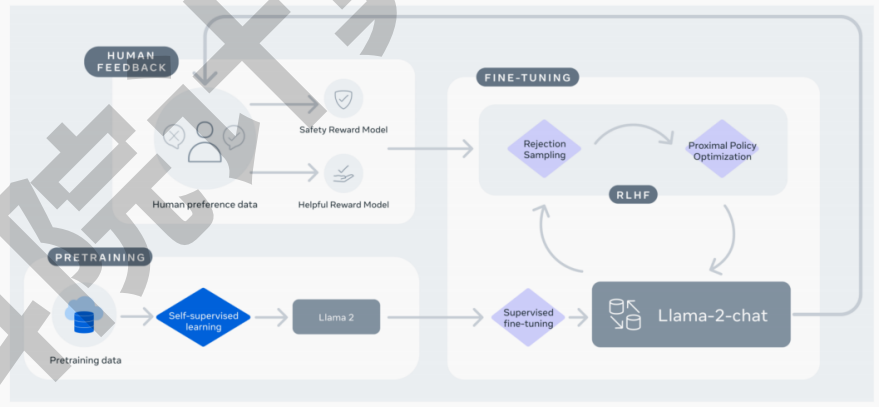

图 8.4　Llama 2 的训练过程[32]

1）**预训练**

Llama 2 采用了 LLaMA1 的大部分预训练设置和模型架构。具体而言，应用了 RMSNorm 进行 pre-normalization，使用了 SwiGLU 激活函数，以及旋转位置嵌入（RoPE）。与 LLaMA 相比，主要的优化点列举如下，具体细节如表 8.1所示。

- 预训练数据增加了 40%；
- 上下文长度由 2048 增大到 4096；

• 在 34B 和 70B 两种参数的模型中引入了分组查询注意力（grouped-query attention，GQA）提升模型的推理速度。

表 8.1　LLaMA 与 Llama 2 模型比较

Name	Training Data	Params	Context Length	GQA	Tokens	LR
LLaMA	See Touvron et al [33]	7B	2k	×	1.0T	3.0×10-4
		13B	2k	×	1.0T	3.0×10-4
		33B	2k	×	1.4T	1.5×10-4
		65B	2k	×	1.4T	1.5×10-4
Llama2	A new mix of publicly available online data	7B	4k	×	2.0T	3.0×10-4
		13B	4k	×	2.0T	3.0×10-4
		34B	4k	√	2.0T	1.5×10-4
		70B	4k	√	2.0T	1.5×10-4

2）**微调**

Llama 2-Chat 中的微调采用了和 ChatGPT 相同的 RLHF 三阶段式的微调，主要分为三个步骤：监督微调（Supervised Fine-Tuning，SFT）、训练奖励模型；迭代微调。

• **监督微调**：SFT 通过构造 prompt-response 文本对微调模型，数据集的构建主要从有用性和安全性两个维度进行标注。具体而言，首先从开源的 SFT 数据集中筛选百万级的数据样例微调 Llama 2-Chat, 再用人工标注的高质量数据进一步对 Llama 2-Chat 进行微调。

• **训练奖励模型**：为了在有用性和安全性之间进行平衡，Llama 2-Chat 中按照有用性和安全性单独训练了两个奖励模型，并通过偏好等级进一步提升奖励模型的效果。

• **迭代微调**：Llama 2-Chat 采用了多轮迭代的方式，结合拒绝采样以及 PPO 算法共同微调。

8.2.3　实验环境

硬件平台：DLP 云平台环境。

软件环境：编程框架 PyTorch1.6.0、CNNL 高性能 AI 运算库，CNRT 运行时库，以及 python 环境及相关的扩展库。

8.2.4　实验内容

1. 本实验主要介绍基于大语言模型 Llama 2 实现聊天机器人的基本功能。使学生了解模型加载和适配、对话模板使用、文本生成流程、模型推理步骤以及命令行聊天等具体过程。

2. 使学生了解 DLP 平台的部署模型的方式，在 DLP 平台上实现人机快速聊天应用。

8.2.5　实验步骤

在该实验中，需要补全模型加载和适配模块、对话模板模块、文本生成模块、FastChat 模型推断模块、命令行聊天应用模块，从而实现基于大语言模型 Llama2 的交互聊天功能。

8.2.5.1 模型适配模块

在模型适配模块中，需要补全设备内存获取模块以及模型加载模块。

1) 设备内存获取模块

设备内存获取模块如代码示例8.8所示。代码定义了一个函数 get_gpu_memory，用于获取每个设备的可用内存。通过使用 torch_mlu 库，代码首先确定要考虑的 MLU 数量，然后遍历每个 MLU，获取其总内存和已分配内存。最终，计算出每个 MLU 的可用内存，并将结果存储在列表 gpu_memory 中，最后返回该列表。函数提供了一种便捷的方式，可以用于监测和优化设备资源的使用情况。

代码示例 8.8　设备内存获取模块

```
# file: fastchat/utils.py
def get_gpu_memory(max_gpus=None):
    """Get available memory for each GPU."""
    #TODO: 存储每个GPU的可用内存信息
    gpu_memory = ____________________________________________
    #TODO: 获取MLU设备的数量，如果未指定最大GPU数（max_gpus为None），则使用所有可用设备；否则，使用
     max_gpus和实际设备数量中的较小值
    num_gpus = (
    ____________________________________________________
    )

    for gpu_id in range(num_gpus):
        #TODO: 将当前MLU设备设置为gpu_id
        ____________________________________
            #TODO: 获取当前MLU设备
            device = ____________________________________
            #TODO: 获取当前MLU设备的属性
            gpu_properties = ____________________________________
            #TODO: 获取总内存，单位转换为GB
            total_memory = ____________________________________
            #TODO: 获取已分配内存，单位转换为GB
            allocated_memory = ____________________________________
            #TODO: 计算可用内存，单位转换为GB
            available_memory = ____________________________________
            #TODO:将可用内存信息添加到列表中
            ____________________________________
    return gpu_memory
```

2) 模型加载模块

模型加载模块的代码如 8.9所示。这段代码是一个用于加载和配置不同深度学习模型的框架，它能支持多种模型适配器，每个适配器负责一个特定类型的深度学习模型。每个模型适配器都继承自 BaseAdapter 类，该类定义了默认的模型加载和配置方法。通过使用装饰器进行缓存，可以加速获取模型适配器的过程。在加载模型时，根据设备类型和配置选项选择合适的模型适配器，并通过适配器加载和配置模型。另外，代码还包括一些用于处理 CPU offloading 和 8 位量化的功能，以及一个用于获取对话模板的函数。整体而言，这个框架提供了一种可扩展的方式来处理不同类型的深度学习模型，并灵活适应不同的硬件和配置需求。

代码示例 8.9　模型适配模块

```
# file: fastchat/model/model_adapter.py
```

```
class BaseAdapter:
    """The base and the default model adapter."""

    def match(self, model_path: str):
        return True

    def load_model(self, model_path: str, from_pretrained_kwargs: dict):
        tokenizer = AutoTokenizer.from_pretrained(model_path, use_fast=False)
        model = AutoModelForCausalLM.from_pretrained(
            model_path, low_cpu_mem_usage=True, **from_pretrained_kwargs
        )
        return model, tokenizer

    def get_default_conv_template(self, model_path: str) -> Conversation:
        return get_conv_template("one_shot")


# A global registry for all model adapters
model_adapters: List[BaseAdapter] = []


def register_model_adapter(cls):
    """Register a model adapter."""
    model_adapters.append(cls())


@lru_cache(maxsize=None)
def get_model_adapter(model_path: str) -> BaseAdapter:
    """Get a model adapter for a model_path."""
    for adapter in model_adapters:
        if adapter.match(model_path):
            return adapter
    raise ValueError(f"No valid model adapter for {model_path}")


def raise_warning_for_incompatible_cpu_offloading_configuration(
    device: str, load_8bit: bool, cpu_offloading: bool
):
    if cpu_offloading:
        if not load_8bit:
            warnings.warn(
                "The cpu-offloading feature can only be used while also using 8-bit-quantization.\n"
                "Use '--load-8bit' to enable 8-bit-quantization\n"
                "Continuing without cpu-offloading enabled\n"
            )
            return False
        if not "linux" in sys.platform:
            warnings.warn(
                "CPU-offloading is only supported on linux-systems due to the limited compatability
    with the bitsandbytes-package\n"
                "Continuing without cpu-offloading enabled\n"
            )
            return False
        if device != "mlu":
            warnings.warn(
                "CPU-offloading is only enabled when using CUDA-devices\n"
                "Continuing without cpu-offloading enabled\n"
            )
            return False
    return cpu_offloading


```

```python
def load_model(
    model_path: str,
    device: str,
    num_gpus: int,
    max_gpu_memory: Optional[str] = None,
    load_8bit: bool = False,
    cpu_offloading: bool = False,
    debug: bool = False,
):
    """Load a model from Hugging Face."""

    #TODO: 处理设备映射，调用函数，检查并更新CPU offloading的配置
    cpu_offloading = ____________________________________________________________
    #TODO: 如果设备是CPU
    if _________________________:
        kwargs = {"torch_dtype": torch.float32}
    #TODO: 如果设备是MLU
    elif _________________________:
        kwargs = {"torch_dtype": torch.float16}
        if num_gpus != 1:
            kwargs["device_map"] = "auto"
            if max_gpu_memory is None:
                kwargs[
                    "device_map"
                ] = "sequential"  # This is important for not the same VRAM sizes
                #TODO:获取每个设备的可用内存
                available_gpu_memory =_____________________
                kwargs["max_memory"] = {
                    i: str(int(available_gpu_memory[i] * 0.85)) + "GiB"
                    for i in range(num_gpus)
                }
            else:
                kwargs["max_memory"] = {i: max_gpu_memory for i in range(num_gpus)}
    #TODO: 如果设备是mps
    elif _____________________:
        kwargs = {"torch_dtype": torch.float16}
        # Avoid bugs in mps backend by not using in-place operations.
        replace_llama_attn_with_non_inplace_operations()
    else:
        raise ValueError(f"Invalid device: {device}")

    #TODO: 如果启用了CPU offloading
    if ___________________:
        # raises an error on incompatible platforms
        from transformers import BitsAndBytesConfig

        if "max_memory" in kwargs:
            kwargs["max_memory"]["cpu"] = (
                str(math.floor(psutil.virtual_memory().available / 2**20)) + "Mib"
            )
        kwargs["quantization_config"] = BitsAndBytesConfig(
            load_in_8bit_fp32_cpu_offload=cpu_offloading
        )
        kwargs["load_in_8bit"] = load_8bit
    #TODO: 如果启用了8位量化但未启用CPU offloading
    ________________________:
        if num_gpus != 1:
            warnings.warn(
                "8-bit quantization is not supported for multi-gpu inference."
            )
        else:
            #TODO:调用压缩模型的函数
```

```
            return _____________________________________________________________

    #TODO: 调用函数，根据给定的模型路径获取适配器，遍历已注册的模型适配器列表，返回第一个匹配的适配器实例。
    adapter = _____________________________________________________________
    #TODO: 使用适配器adapter加载模型和分词器
    model, tokenizer = _____________________________________________________________

    if (device == "mlu" and num_gpus == 1 and not cpu_offloading) or device == "mps":
        model.to(device)

    if debug:
        print(model)

    return model, tokenizer


def get_conversation_template(model_path: str) -> Conversation:
    #TODO: 调用函数,获取适用于给定模型路径的模型适配器
    adapter = _____________________________________________
    #TODO: 使用适配器获取默认的对话模板
    return _____________________________________________
```

8.2.5.2 对话模板模块

对话模板模块的代码如 8.10所示。这段代码定义了两个类 SeparatorStyle 和 Conversation 以及相关的全局变量和函数。SeparatorStyle 是一个枚举类，定义了不同的分隔符样式。Conversation 类用于表示对话历史，包含了对话的各种信息，如模板名称、系统提示、角色、消息内容等。该类提供了一些方法，如生成提示文本、追加新消息、转换为 Gradio Chatbot 格式、转换为 OpenAI API 格式、复制等。最后，通过全局变量 conv_templates 实现了对话模板的注册和获取功能，用于管理多个对话模板。

代码示例 8.10 对话模板模块

```
# file: fastchat/conversation.py
class SeparatorStyle(Enum):
    """Separator styles."""

    ADD_COLON_SINGLE = auto()
    ADD_COLON_TWO = auto()
    NO_COLON_SINGLE = auto()
    BAIZE = auto()
    DOLLY = auto()
    RWKV = auto()
    PHOENIX = auto()
    NEW_LINE = auto()


@dataclasses.dataclass
class Conversation:
    """A class that keeps all conversation history."""

    # The name of this template
    name: str
    # System prompts
    system: str
    # Two roles
    roles: List[str]
```

```
    # All messages
    messages: List[List[str]]
    # Offset of few shot examples
    offset: int
    # Separators
    sep_style: SeparatorStyle
    sep: str
    sep2: str = None
    # Stop criteria (the default one is EOS token)
    stop_str: str = None
    # Stops generation if meeting any token in this list
    stop_token_ids: List[int] = None

    # Used for the state in the gradio servers.
    # TODO(lmzheng): refactor this
    conv_id: Any = None
    skip_next: bool = False
    model_name: str = None

    def get_prompt(self) -> str:
        """Get the prompt for generation."""
        #TODO:# 如果分隔符样式为ADD_COLON_SINGLE
        if _____________________________________________:
            ret = self.system + self.sep
            for role, message in self.messages:
                if message:
                    ret += role + ": " + message + self.sep
                else:
                    ret += role + ":"
            return ret
        #TODO:# 如果分隔符样式为ADD_COLON_TWO
        elif _____________________________________________:
            seps = [self.sep, self.sep2]
            ret = self.system + seps[0]
            for i, (role, message) in enumerate(self.messages):
                if message:
                    ret += role + ": " + message + seps[i % 2]
                else:
                    ret += role + ":"
            return ret
        #TODO: 如果分隔符样式为NO_COLON_SINGLE
        elif _____________________________________________:
            ret = self.system
            for role, message in self.messages:
                if message:
                    ret += role + message + self.sep
                else:
                    ret += role
            return ret
        #TODO: 如果分隔符样式为BAIZE
        elif _____________________________________________:
            ret = self.system + "\n"
            for role, message in self.messages:
                if message:
                    ret += role + message + "\n"
                else:
                    ret += role
            return ret
        #TODO: 如果分隔符样式为DOLLY
        elif _____________________________________________:
            seps = [self.sep, self.sep2]
            ret = self.system
```

```
            for i, (role, message) in enumerate(self.messages):
                if message:
                    ret += role + ":\n" + message + seps[i % 2]
                    if i % 2 == 1:
                        ret += "\n\n"
                else:
                    ret += role + ":\n"
            return ret
        #TODO: 如果分隔符样式为RWKV
        elif ____________________________________________
            ret = self.system
            for i, (role, message) in enumerate(self.messages):
                if message:
                    ret += (
                        role
                        + ": "
                        + message.replace("\r\n", "\n").replace("\n\n", "\n")
                    )
                    ret += "\n\n"
                else:
                    ret += role + ":"
            return ret
        #TODO: 如果分隔符样式为PHOENIX
        elif ____________________________________________
            ret = self.system
            for role, message in self.messages:
                if message:
                    ret += role + ": " + "<s>" + message + "</s>"
                else:
                    ret += role + ": " + "<s>"
            return ret
        #TODO: 如果分隔符样式为NEW_LINE
        elif ____________________________________________
            ret = self.system + self.sep
            for role, message in self.messages:
                if message:
                    ret += role + "\n" + message + self.sep
                else:
                    ret += role + "\n"
            return ret
        else:
            raise ValueError(f"Invalid style: {self.sep_style}")

    def append_message(self, role: str, message: str):
        """Append a new message."""
        #TODO:将角色和消息内容作为列表添加到对话历史中。
        ____________________________________________

    def to_gradio_chatbot(self):
        """Convert the history to gradio chatbot format"""
        ret = []
        for i, (role, msg) in enumerate(self.messages[self.offset :]):
            if i % 2 == 0:
                #TODO:# 对话历史中偶数索引的消息作为用户消息，将其添加为列表的第一个元素，第二个元素置为None
                ____________________________________
            else:
                ret[-1][-1] = msg
        return ret

    def to_openai_api_messages(self):
        """Convert the conversation to OpenAI chat completion format."""
```

```
        ret = [{"role": "system", "content": self.system}]

        for i, (_, msg) in enumerate(self.messages[self.offset :]):
            if i % 2 == 0:
                #TODO:# 对话历史中偶数索引的消息作为用户消息，将其添加为字典到列表
                ____________________________________
            else:
                if msg is not None:
                    ret.append({"role": "assistant", "content": msg})
        return ret

    def copy(self):
        return Conversation(
            name=____________________,
            system=____________________,
            roles=____________________,
            messages=[[x, y] for x, y in self.messages],
            offset=____________________,
            sep_style=____________________,
            sep=____________________,
            sep2=____________________,
            stop_str=self.stop_str,
            stop_token_ids=self.stop_token_ids,
            conv_id=____________________,
            model_name=____________________,
        )

    def dict(self):
        return {
            "name": self.name,
            "system": self.system,
            "roles": self.roles,
            "messages": self.messages,
            "offset": self.offset,
            "conv_id": self.conv_id,
            "model_name": self.model_name,
        }


# A global registry for all conversation templates
conv_templates: Dict[str, Conversation] = {}


def register_conv_template(template: Conversation, override: bool = False):
    """Register a new conversation template."""
    if not override:
        assert template.name not in conv_templates, f"{name} has been registered."
    conv_templates[template.name] = template


def get_conv_template(name: str) -> Conversation:
    """Get a conversation template."""
    return conv_templates[name].copy()
```

8.2.5.3 文本生成模块

1) ChatGLM 文本生成模块

这段代码定义了一个基于 PyTorch 的 ChatGLM 生成器函数 chatglm_generate_stream，用于在推理模式下生成聊天文本。该生成器使用模型的聊天 API，根据给定的参数和历史对

话生成文本。其中，函数 stream_chat_token_num 计算给定对话历史的 token 数。

首先，对话历史通过 stream_chat_token_num 函数计算得到输入的 token 数。然后，通过 model.stream_chat 方法，使用模型生成聊天文本。每一轮生成的文本通过生成器生成出来，包含文本、使用情况和完成原因。最后一轮的生成结果中包含了完成原因，将完成原因设为 “stop”。整体而言，该代码实现了对话的生成，并通过生成器逐步输出生成的聊天文本，其中完成原因为 “stop” 表示生成结束。

代码示例 8.11　ChatGLM 文本生成模块

```
# file: fastchat/model/chatglm_model.py
import torch
import torch_mlu
from typing import List, Tuple

def stream_chat_token_num(tokenizer, query: str, history: List[Tuple[str, str]] = None):
    if history is None:
        history = []
    #TODO: 如果历史记录为空，将当前问题作为提示（prompt）
    if not history:
        prompt = ____________________________________
    else:
        #如果有历史记录，将历史记录和当前问题组合成一个提示
        prompt = ""
        for i, (old_query, response) in enumerate(history):
            prompt += "[Round {}]\n问：{}\n答：{}\n".format(i, old_query, response)
        #TODO: 最后一轮的问题格式为 "[Round len(history)]\n问：query\n答："
        prompt += ____________________________________________________
    #TODO 使用分词器tokenizer对提示进行编码
    inputs = ____________________________
    return sum([len(x) for x in inputs["input_ids"]])


@torch.inference_mode()
def chatglm_generate_stream(
    model, tokenizer, params, device, context_len=2048, stream_interval=2
):
    """Generate text using model's chat api"""
    messages = params["prompt"]
    #TODO 从参数中获取生成文本时的max_new_tokens数，如果参数中未指定，则默认为 256。
    max_new_tokens = ____________________________________________
    #TODO: 从参数中获取生成文本时的temperature（控制文本生成的多样性），如果参数中未指定，则默认为 1.0
    temperature = ____________________________________________
    #TODO:从参数中获取生成文本时的 top_p（控制生成文本的多样性），如果参数中未指定，则默认为 1.0。
    top_p = ____________________________________________
    #TODO: 从参数中获取生成文本时的 repetition penalty（抑制文本中重复的程度），如果参数中未指定，则默认为 1.0。
    repetition_penalty =____________________________________________
    #TODO: 从参数中获取是否在生成的文本中包含输入的历史消息，如果参数中未指定，则默认为 True。
    echo = ____________________________________________

    gen_kwargs = {
        # "max_new_tokens": max_new_tokens,  disabled due to a warning.
        "do_sample": True if temperature > 1e-5 else False,
        "top_p": top_p,
        "repetition_penalty": repetition_penalty,
        "logits_processor": None,
    }
    #TODO: 如果 temperature 大于 1e-5，将温度参数添加到 gen_kwargs 中。
    if temperature > 1e-5:
```

```
        ------------------------------------------------

    hist = []
    for i in range(0, len(messages) - 2, 2):
        #TODO: 遍历消息列表，将奇数索引的消息作为前一方的发言，偶数索引的消息作为后一方的发言，并添加到对话历史hist中。
        ------------------------------------------------
    query = messages[-2][1]

    #TODO:# 计算输入历史的 token 数
    input_echo_len = ------------------------------------------------

    for i, (response, new_hist) in enumerate(
        model.stream_chat(tokenizer, query, hist, **gen_kwargs)
    ):
        if echo:
            output = query + " " + response
        else:
            output = response

        yield {
            "text": output,
            "usage": {
                "prompt_tokens": input_echo_len,
                "completion_tokens": i,
                "total_tokens": input_echo_len + i,
            },
            "finish_reason": None,
        }

    # TODO: 最后一轮的生成结果包含完成原因，将完成原因设为 "stop"
    # Only last stream result contains finish_reason, we set finish_reason as stop

    ret = {
        ------------------------------------------------
        ------------------------------------------------
        ------------------------------------------------
        ------------------------------------------------
        ------------------------------------------------
        ------------------------------------------------
        ------------------------------------------------
    }
    yield ret
```

2）通用文本生成模块

通用文本生成模块的代码如 8.12所示。它接收一个预训练的语言模型、分词器、参数和设备信息作为输入，并生成基于给定提示（prompt）的文本流。生成的文本流具有一定的长度，同时可以根据设置的条件（如温度、重复惩罚等）进行调整。生成过程中，根据设定的停止条件，可以在特定的时刻结束生成。最终，该函数返回生成的文本、使用情况信息以及生成结束的原因。在整个生成过程中，还进行了一些资源的清理工作，包括释放内存和缓存。

代码示例 8.12　通用文本生成模块

```
# file: fastchat/serve/inference.py
def prepare_logits_processor(
    temperature: float, repetition_penalty: float, top_p: float, top_k: int
) -> LogitsProcessorList:
    processor_list = LogitsProcessorList()
```

```
    # TemperatureLogitsWarper doesn't accept 0.0, 1.0 makes it a no-op so we skip two cases.
    #TODO: 如果温度大于等于 1e-5 且不等于 1.0，就添加 TemperatureLogitsWarper 处理器
    if temperature >= 1e-5 and temperature != 1.0:
        _____
    #TODO: 如果重复惩罚大于 1.0，就添加 RepetitionPenaltyLogitsProcessor 处理器
    if repetition_penalty > 1.0:
        _____
    #TODO: 如果 top_p 在 (1e-8, 1.0) 范围内，就添加 TopPLogitsWarper 处理器
    if 1e-8 <= top_p < 1.0:
        _____
    #TODO: 如果 top_k 大于 0，就添加 TopKLogitsWarper 处理器
    if top_k > 0:
        _____
    return processor_list


#@torch.inference_mode()
def generate_stream(
    model, tokenizer, params, device, context_len=2048, stream_interval=2
):
    #TODO:# 关闭梯度计算
    _____
    prompt = params["prompt"]
    #TODO: 获取prompt的长度
    len_prompt = _____
    #TODO: 从参数中获取生成文本时的temperature（控制文本生成的多样性），如果参数中未指定，则默认为 1.0
    。
    temperature = _____
    #TODO: 从参数中获取生成文本时的 repetition_penalty（抑制文本中重复的程度），如果参数中未指定，则默
    认为 1.0。
    repetition_penalty =_____
    #TODO:从参数中获取生成文本时的 top_p（控制生成文本的多样性），如果参数中未指定，则默认为 1.0。
    top_p = _____
    #TODO: 从输入的参数中获取 top_k 的值，如果参数中没有设置，则默认为 -1。
    top_k = _____  # -1 means disable
    #TODO 从参数中获取生成文本时的max_new_tokens数，如果参数中未指定，则默认为 256。
    max_new_tokens = _____
    #TODO: 从参数中获取 stop_str，如果未设置，则默认为 None
    stop_str = _____
    #TODO: 从参数中获取 echo，如果未设置，默认为 True。并将其转换为布尔值。
    echo = _____
    stop_token_ids = params.get("stop_token_ids", None) or []
    #TODO: 将文本生成停止的标记添加到已有的停止标记列表stop_token_ids中。
    _____

    #TODO: 创建一个 logits 处理器列表
    logits_processor = _____

    #TODO: 使用tokenizer将输入文本转换为模型可接受的输入张量
    input_ids = _____
    #TODO: 记录输入文本的长度
    input_echo_len = _____
    #TODO: 创建一个名为 output_ids 的列表，其初始值等于 input_ids 列表的内容
    output_ids = _____

    if model.config.is_encoder_decoder:
        max_src_len = context_len
    else:
        max_src_len = context_len - max_new_tokens - 8

    #TODO: 截取源文本的最后一部分，以适应模型的上下文长度
    input_ids = _____
```

```
66
67      if model.config.is_encoder_decoder:
68          encoder_output = model.encoder(
69              input_ids=torch.as_tensor([input_ids], device=device)
70          )[0]
71          start_ids = torch.as_tensor(
72              [[model.generation_config.decoder_start_token_id]],
73              dtype=torch.int64,
74              device=device,
75          )
76
77      past_key_values = out = None
78      for i in range(max_new_tokens):
79          if i == 0:
80              if model.config.is_encoder_decoder:
81                  #TODO: 使用解码器对起始标记进行处理
82                  out = model.decoder(
83                      input_ids=_____________________________,
84                      encoder_hidden_states=_____________,
85                      use_cache=_____________________________,
86                  )
87                  logits = model.lm_head(out[0])
88              else:
89                  #TODO: 非编码解码器类型，直接使用模型进行处理
90                  out = ___________________________________________________
91                  logits = out.logits
92              #TODO: 记录过去的键值
93              past_key_values = ____________________________________
94          else:
95              if model.config.is_encoder_decoder:
96                  #TODO: 使用解码器对当前标记进行处理
97                  out = model.decoder(
98                      input_ids=_____________________________,
99                      encoder_hidden_states=_____________,
100                     use_cache=_____________________________,
101                     past_key_values=_____________________,
102                 )
103                 #TODO: 获取logits（预测的标记分布）
104                 logits = ____________________________________
105             else:
106                 out = model(
107                 #TODO:非编码解码器类型，直接使用模型对当前标记进行处理
108                     input_ids=_____________________________,
109                     use_cache=_____________________________,
110                     past_key_values=_____________________,
111                 )
112                 logits = out.logits
113             #TODO: 更新过去的键值
114             past_key_values = ____________________________________
115
116         if logits_processor:
117             if repetition_penalty > 1.0:
118                 tmp_output_ids = torch.as_tensor([output_ids], device=logits.device)
119             else:
120                 tmp_output_ids = None
121             #TODO: 使用logits_processor处理最后一个标记的logits
122             last_token_logits = ___________________________________
123         else:
124             #TODO: 没有logits处理器，直接获取最后一个标记的logits
125             last_token_logits = ___________________________________
126
127         if device == "mps":
```

```
            # Switch to CPU by avoiding some bugs in mps backend.
            last_token_logits = last_token_logits.float().to("cpu")

        if temperature < 1e-5 or top_p < 1e-8:  # greedy
            #TODO:# 通过argmax获取概率最高的标记索引，并将其转换为整数类型。
            token = ____________________________________
        else:
            #TODO: 使用softmax将概率分布归一化
            probs = ____________________________________
            #TODO: 使用torch.multinomial函数根据概率分布采样生成标记,并转换为整数类型。
            token = ____________________________________

        # 将生成的token添加到输出序列output_ids中
        ____________________________________

        if token in stop_token_ids:
            stopped = True
        else:
            stopped = False

        #TODO: 判断是否达到生成结果的间隔、已完成生成最大标记数，或者已经停止生成
        if _________________________ or _________________________ or _________________________:
            if echo:
                tmp_output_ids = output_ids
                rfind_start = len_prompt
            else:
                tmp_output_ids = output_ids[input_echo_len:]
                rfind_start = 0

            output = tokenizer.decode(
                tmp_output_ids,
                skip_special_tokens=True,
                spaces_between_special_tokens=False,
            )
            if stop_str:
                if isinstance(stop_str, str):
                    #TODO: 在输出字符串中从右向左搜索停止标记的位置,rfind_start参数指定了搜索的起始位
置
                    pos = ____________________________________
                    if pos != -1:
                        #TODO: 将输出字符串截断，仅保留停止标记位置之前的部分
                        output = ____________________________________
                        stopped = True
                elif isinstance(stop_str, Iterable):
                    for each_stop in stop_str:
                        #TODO: 从指定位置（rfind_start）向前搜索每个停止标记在输出字符串中的最后出现位
置
                        pos = ____________________________________
                        if pos != -1:
                            #TODO: 将输出字符串截断，仅保留停止标记位置之前的部分
                            output = ____________________________________
                            stopped = True
                            break
                else:
                    raise ValueError("Invalid stop field type.")

            yield {
                "text": output,
                "usage": {
                    "prompt_tokens": input_echo_len,
                    "completion_tokens": i,
                    "total_tokens": input_echo_len + i,
```

```
                },
                "finish_reason": None,
            }

        if stopped:
            break

    # finish stream event, which contains finish reason
    if i == max_new_tokens - 1:
        finish_reason = "length"
    elif stopped:
        finish_reason = "stop"
    else:
        finish_reason = None

    #TODO: 返回生成的文本、使用情况和结束原因的字典
    yield {
        ____________________________________
        ____________________________________
        ____________________________________
        ____________________________________
        ____________________________________
        ____________________________________
        ____________________________________
    }

    # clean
    del past_key_values, out
    gc.collect()
    #TODO:释放MLU设备上的缓存空间
    ____________________________________
```

8.2.5.4 FastChat 模型推断模块

FastChat 模型推断模块的代码示例如 8.13所示。这段代码提供了一个灵活的聊天交互框架，允许用户通过实现 ChatIO 的子类来自定义输入、输出和输出流的处理方式。chat_loop 函数通过载入模型、选择对话方式，循环获取用户输入并生成聊天输出，提供了一个简单的聊天应用结构。

代码示例 8.13　命令行聊天应用-主函数模块

```
# file: fastchat/serve/inference.py
class ChatIO(abc.ABC):
    @abc.abstractmethod
    def prompt_for_input(self, role: str) -> str:
        """Prompt for input from a role."""

    @abc.abstractmethod
    def prompt_for_output(self, role: str):
        """Prompt for output from a role."""

    @abc.abstractmethod
    def stream_output(self, output_stream):
        """Stream output."""


def chat_loop(
    model_path: str,
    device: str,
```

```
    num_gpus: int,
    max_gpu_memory: str,
    load_8bit: bool,
    cpu_offloading: bool,
    conv_template: Optional[str],
    temperature: float,
    max_new_tokens: int,
    chatio: ChatIO,
    debug: bool,
):
    #TODO: 调用load_model函数加model和tokenizer
    model, tokenizer =_____________________________________
    is_chatglm = "chatglm" in str(type(model)).lower()

    #TODO: 如果提供了对话模板，使用提供的模板,调用get_conv_template创建会话对象
    if conv_template:
        conv = _____________________________________
    else:
        #TODO:否则使用默认的对话模板,调用get_conversation_template创建会话对象
        conv = _____________________________________

    while True:
        try:
            #TODO: 尝试获取用户输入，传入 conv.roles[0] 作为角色标识
            inp = _____________________________________
        except EOFError:
            inp = ""
        if not inp:
            print("exit...")
            break

        conv.append_message(conv.roles[0], inp)
        conv.append_message(conv.roles[1], None)

        if is_chatglm:
            generate_stream_func = chatglm_generate_stream
            prompt = conv.messages[conv.offset :]
        else:
            generate_stream_func = generate_stream
            prompt = conv.get_prompt()

        gen_params = {
            "model": model_path,
            "prompt": prompt,
            "temperature": temperature,
            "max_new_tokens": max_new_tokens,
            "stop": conv.stop_str,
            "stop_token_ids": conv.stop_token_ids,
            "echo": False,
        }

        #TODO: 获取机器对话输出，传入 conv.roles[1] 作为角色标识
        _____________________________________
        output_stream = generate_stream_func(model, tokenizer, gen_params, device)
        #TODO: # 输出对话的流式输出
        outputs =_____________________________________
        # NOTE: strip is important to align with the training data.
        conv.messages[-1][-1] = outputs.strip()

        if debug:
            print("\n", {"prompt": prompt, "outputs": outputs}, "\n")
```

8.2.5.5 命令行聊天应用模块

命令行聊天应用模块主要负责解析命令行参数、选择聊天界面风格、配置运行参数，最后启动聊天循环与模型进行交互。具体而言，主要包括主函数模块、Simple 聊天交互模块、Rich 聊天交互模块。

1) 主函数模块

命令行应用中主函数模块的代码如 8.30所示。在这段代码中，需要调用 chat_loop 函数，启动聊天循环。该函数接受各种参数，包括模型路径、设备、温度、最大新 token 数量等，以及用于聊天输入和输出的 chatio 对象（SimpleChatIO 或 RichChatIO）。同时，还需要使用 argparse 解析命令行参数。在此过程中，读者需要体会 Simple 模式和 Rich 模式的区别，能够利用这两种模式进行命令行交互。

代码示例 8.14 命令行聊天应用-主函数模块

```python
# file: fastchat/serve/cli.py
def main(args):
    if args.gpus:
        if len(args.gpus.split(",")) < args.num_gpus:
            raise ValueError(
                f"Larger --num-gpus ({args.num_gpus}) than --gpus {args.gpus}!"
            )
        os.environ["CUDA_VISIBLE_DEVICES"] = args.gpus

    if args.style == "simple":
        chatio = SimpleChatIO()
    elif args.style == "rich":
        chatio = RichChatIO()
    else:
        raise ValueError(f"Invalid style for console: {args.style}")
    try:
        #TODO:调用 chat_loop 函数，启动聊天循环
        ----------------------------------------

    except KeyboardInterrupt:
        print("exit...")


if __name__ == "__main__":
    #TODO: 创建一个ArgumentParser对象，用于解析命令行参数
    parser = ----------------------------------------
    #TODO: 向ArgumentParser对象中添加模型相关的参数，这些参数由add_model_args函数定义
    ----------------------------------------------------
    parser.add_argument(
        "--conv-template", type=str, default=None, help="Conversation prompt template."
    )
    parser.add_argument("--temperature", type=float, default=0.7)
    parser.add_argument("--max-new-tokens", type=int, default=512)

    #TODO:试一下simple模式，同时也试一下rich模式。
    parser.add_argument(
        "--style",
        type=str,
        default="----------",
        choices=["simple", "rich"],
        help="Display style.",
    )
    parser.add_argument("--debug", action="store_true", help="Print debug information")
```

```
    args = parser.parse_args()
    main(args)
```

2) **Simple 聊天交互模块**

Simple 聊天交互模块的代码示例如 8.15所示。这段代码定义了一个名为 SimpleChatIO 的类，该类继承自 ChatIO。SimpleChatIO 用于在命令行中实现简单的聊天输入输出界面。其中，prompt_for_input 用于接收用户输入，prompt_for_output 用于在命令行中显示输出的角色，stream_output 对聊天输出进行处理，只在新增内容时进行输出，以提高可读性。通过将 SimpleChatIO 类实例用于聊天交互，可以在终端中进行基本的单轮对话，用户通过输入与角色交互，而输出则经过处理后只显示新增内容。

代码示例 8.15　Simple 聊天交互模块

```
# file: fastchat/serve/cli.py
#TODO: 构建SimpleChatIO类，它继承自ChatIO类
_____:
    def prompt_for_input(self, role) -> str:
        return input(f"{role}: ")

    def prompt_for_output(self, role: str):
        print(f"{role}: ", end="", flush=True)

    def stream_output(self, output_stream):
        pre = 0
        for outputs in output_stream:
            output_text = outputs["text"]
            #TODO: 移除文本两端的空白字符，然后按空格分割
            output_text = _____
            #TODO: 获取处理后的文本中单词的数量
            now = _____
            if now > pre:
                # 输出不同于前次的部分，其中，新增的内容以空格分隔的形式显示，确保每次新增的内容都在同一行，并及时刷新输出缓冲区。
                _____
                pre = now
        print(" ".join(output_text[pre:]), flush=True)
        return " ".join(output_text)
```

3) **Rich 聊天交互模块**

Rich 聊天交互模块的代码如 8.16所示。该代码定义了一个名为 RichChatIO 的类，该类继承自 ChatIO。RichChatIO 用于实现富文本的聊天输入输出界面。在初始化方法中，它创建了一个用于输入的 PromptSession，设定了命令历史、输入补全和控制台对象。prompt_for_input，prompt_for_output 用于显示富文本的角色输出提示。stream_output 通过 rich 库的 Live 模块实现动态刷新，渲染聊天输出为 Markdown 格式，处理换行符等特殊情况，并在控制台上实时更新显示。这使得聊天交互更具可读性和交互性，提供了更丰富的文本显示功能。其中，需要用到 prompt_toolkit 库，它通常用于构建需要用户与命令行进行交互的应用程序，例如命令行工具、交互式控制台等，提供实时的输入提示、自动补全等功能。

代码示例 8.16　Rich 聊天交互模块

```
# file: fastchat/serve/cli.py
#TODO: 构建RichChatIO类，它继承自ChatIO类
_____:
```

```
    def __init__(self):
        #TODO: 创建PromptSession实例，用于获取用户输入，并设置输入历史记录
        self._prompt_session = _____________________________________________
        #TODO: 创建自动补全器，用于用户输入的自动完成
        self._completer =_____________________________(
            words=["!exit", "!reset"], pattern=re.compile("$")
        )
        #TODO:创建Console 实例，在命令行界面中以更丰富的样式显示文本
        self._console = _____________________________

    def prompt_for_input(self, role) -> str:
        self._console.print(f"[bold]{role}:")
        # TODO(suquark): multiline input has some issues. fix it later.
        prompt_input = self._prompt_session.prompt(
            completer=self._completer,
            multiline=False,
            #TODO: 启用自动建议功能
            auto_suggest=_____________________________,
            key_bindings=None,
        )
        self._console.print()
        return prompt_input

    def prompt_for_output(self, role: str):
        self._console.print(f"[bold]{role}:")

    def stream_output(self, output_stream):
        """Stream output from a role."""
        # TODO(suquark): the console flickers when there is a code block
        #  above it. We need to cut off "live" when a code block is done.

        # Create a Live context for updating the console output

        #TODO: 创建Live上下文管理器，用于实现在命令行中实时更新显示。其中，需要指定要在其上执行实时更新的console实例，每秒刷新的次数设为4
        _____________________________________________as live:
            # Read lines from the stream
            for outputs in output_stream:
                if not outputs:
                    continue
                text = outputs["text"]
                # Render the accumulated text as Markdown
                # NOTE: this is a workaround for the rendering "unstandard markdown"
                #  in rich. The chatbots output treat "\n" as a new line for
                #  better compatibility with real-world text. However, rendering
                #  in markdown would break the format. It is because standard markdown
                #  treat a single "\n" in normal text as a space.
                #  Our workaround is adding two spaces at the end of each line.
                #  This is not a perfect solution, as it would
                #  introduce trailing spaces (only) in code block, but it works well
                #  especially for console output, because in general the console does not
                #  care about trailing spaces.
                lines = []
                for line in text.splitlines():
                    lines.append(line)
                    if line.startswith("```"):lines.append("\n")
                    else:lines.append("  \n")
                markdown = Markdown("".join(lines))
                #TODO: 将渲染后的 markdown 文本实时更新到控制台
                _____________________________________
        self._console.print()
        return text
```

8.2.5.6 实验运行

根据第8.2.5.1节～第8.2.5.5节的描述补全 fastchat/utils.py、fastchat/model/model_adapter.py、fastchat/conversation.py、fastchat/model/chatglm_model.py、fastchat/serve/inference.py 以及 fastchat/serve/cli.py 文件后，在指定目录下执行以下命令，完成基于 Llama 2 聊天机器人的实现过程。

1) 环境申请与安装

申请实验环境、安装软件包并登录云平台，本实验的代码存放在云平台/opt/code_chap_8_student 目录下。

```
# 登录云平台
ssh root@xxx.xxx.xxx.xxx -p xxxxx
# 进入/opt/tools/accelerate-0.20-release-mlu目录
cd /opt/tools/accelerate-0.20-release-mlu
#安装accelerate软件包
python setup.py install
# 进入/opt/tools/transformers-mlu-dev目录
cd /opt/tools/transformers-mlu-dev
#安装transformers软件包
python setup.py install
# 进入/opt/code_chap_8_student/llama2_7b目录
cd /opt/code_chap_8_student/llama2_7b
```

2) 代码实现

补全 fastchat/utils.py、fastchat/model/model_adapter.py、fastchat/conversation.py、fastchat/model/chatglm_model.py、fastchat/serve/inference.py 以及 fastchat/serve/cli.py 文件。

```
# 进入fastchat目录
cd fastchat
#补全utils.py文件
vim utils.py
# 补全conversation.py文件
vim conversation.py
# 进入model目录
cd model
# 补全model_adapter.py文件
vim model_adapter.py
# 补全chatglm_model.py文件
vim chatglm_model.py
# 进入serve目录
cd serve
# 补全inference.py文件
vim inference.py
# 补全cli.py文件
vim cli.py
```

3) 运行实验

```
# 执行脚本命令运行实验
bash run_scripts/run_mlu_infer.sh
```

8.2.6 实验评估

- 60 分标准：能够正确实现模型适配模块、对话模板模块，其中无论是否提供对话模板都能正确地实现程序。

• 80 分标准：在 60 分标准基础上，能够正确实现文本生成模块。

• 100 分标准：在 80 分标准基础上，能够采用 Simple 和 Rich 两种聊天模式与机器人进行交互，实现 FastChat 模型推断模块，可以完成聊天应用。

8.2.7 实验思考

1) Llama 2 中采用的 RMSNorm 归一化函数是如何提高训练的稳定性的?

2) Llama 2 中采用的 SwiGLU 函数相对于 ReLU 函数而言做了什么改进，有什么好处?

3) Llama 2 中采用的旋转位置编码相对于绝对位置编码有何好处和优势?

4) Llama 2 中采用的分组查询注意力机制相对于传统注意力机制而言做了哪些调整?

8.3 基于 Code Llama 实现代码生成

8.3.1 实验目的

1) 本实验旨在介绍基于 Code Llama 实现代码生成的基本原理和操作步骤;

2) 学生将深入了解基于 Code Llama 实现代码生成关键步骤，包括基于 transformers 库的 Codel Llama 推理模块、Transformer 模型构建模块、Llama 代码生成模块等;

3) 掌握如何在 DLP 平台上部署基于 Code Llama 的代码生成模型,能够实现 Code Llama 代码生成模块、Code Llama 代码补全模块、Code Llama 指令微调模块。

实验工作量：约 71 行代码，5 小时。

8.3.2 背景介绍

8.3.2.1 Code Llama 介绍

Code Llama[39] 是一系列基于 Llama 2 的大语言模型，用于代码生成、代码补全等，它在几个代码基准测试中达到开放模型的最先进性能。该项目目前发布了多了版本模型，具体如下:

• **Code Llama**：一个用于代码生成任务的基础模型。

• **Code Llama-Python**：一个专门用于 Python 的版本。

• **Code Llama-Instruct**：一个用人类指令和自指导代码合成数据进行微调的版本。

这些模型具有如下特点:

• 分别具有 7B、13B 和 34B 的参数规格。

• 所有模型都是在 16k token 的序列上进行训练，并在最多包含 100k token 的输入上显示出改进。

• 7B 和 13B 的 Code Llama 和 Code Llama-Instruct 变体支持基于周围内容的补全。

Code Llama 的训练流程如图 8.5所示。它主要包括从基础模型进行代码训练（code training from foundation models，图中紫色和蓝色模块）、代码补全（Infilling，紫色模块）、长输入上下文（Long input context，绿色模块）、指令微调（Instruction fine-tuning，黄色模块）这四个步骤:

- **从基础模型进行代码训练**：使用通用语言模型 Llama 2 作为基础模型进行训练。
- **代码补全**：不但提供代码生成的能力，还有代码补全的能力，可以用来在 IDE 中进行代码补全或者进行文档的生成，且该能力所花费的训练代价比较低。
- **长输入上下文**：通过修改 RoPE 的参数，提供了仓库级的代码生成能力，极大地扩展了原始 Llama 2 的 context 的长度，从 4096 提升到了 100k。
- **指令微调**：指令微调可以显著提升有用性、安全性，从而更准确地回应指令性问题。

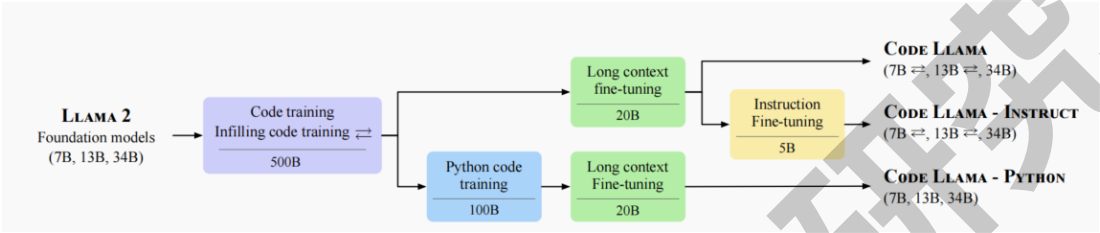

图 8.5　Code Llama 的训练流程[39]。其中，训练过程中的 tokens 的数量在微调的不同阶段都被标注了出来。同时，支持补全的模型用 ⇄ 符号标记。

8.3.2.2　Hugging Face Transformers 简介

Hugging Face Transformers 是一个用于自然语言处理（NLP）的开源库，提供了各种预训练模型。这些模型被广泛应用于各种任务，如文本分类、命名实体识别、问答、文本生成等。Transformers 库易于使用，可方便地集成到现有的深度学习框架，如 PyTorch 和 TensorFlow。具体用法可参见官方文档[①]。

8.3.3　实验环境

硬件平台：DLP 云平台环境。

软件环境：编程框架 PyTorch1.6.0、CNNL 高性能 AI 运算库，CNRT 运行时库，以及 python 环境及相关的扩展库。

8.3.4　实验内容

1. 本实验主要介绍基于 Code Llama 实现代码生成的基本功能。使学生了解基于 transformers 库的 Code Llama 推理模块的具体实现流程。

2. 本实验能够使得学生掌握 Transformer 模型的构建过程、使其能够正确实现 Llama 代码生成模块。

3. 使学生了解 DLP 平台的部署模型的方式，在 DLP 平台上实现 Code Llama 代码生成、Code Llama 代码补全、Code Llama 指令微调的相关应用。

8.3.5　实验步骤

该实验需要补全基于 transformers 库的 Code Llama 推理模块、Transformer 模型构建模块、Llama 代码生成模块、Code Llama 代码生成模块、Code Llama 代码补全模块、Code Llama

[①] https://huggingface.co/docs/transformers/index

指令微调模块，具体介绍如下所示。

8.3.5.1 基于 transformers 库的 Code Llama 推理模块

基于 transformers 库的 Code Llama 主要利用 Hugging Face 的 transformers 库，调用 CodeLlama-13b-Instruct-hf 模型进行代码生成，实现了从给定的代码提示生成代码的功能，并提供了生成时间、标记数量等信息。Code Llama 推理模块可以分为模型加载和设置模块、文本生成和性能统计模块、结果处理和打印模块，接下来进行具体介绍。

1）**模型加载和设置模块**

模型加载和设置模块的代码示例如8.17所示。这段代码使用 Hugging Face 的 transformers 库加载了一个预训练的 Causal Language Model (CLM)，为后续对 CLM 进行文本生成任务提供了必要的基础。首先，通过 AutoTokenizer 加载了指定 ID 的预训练分词器，接着使用 AutoModelForCausalLM 加载了相应的预训练 CLM 模型。在模型加载过程中，通过参数设置，使用了 MLU 作为运行设备，采用了 torch.float16 的数据类型，且关闭了安全张量的使用。最后，将模型设置为评估模式。

代码示例 8.17　模型加载和设置模块

```
# file: Codellama-infer.py
from transformers import AutoTokenizer, AutoModelForCausalLM
import transformers
import torch
import torch_mlu

#TODO: 设置模型路径，对CodeLlama-7b-hf或CodeLlama-13b-Instruct-hf模型进行加载测试
model_id=_____
#TODO: 利用transformers库函数从预训练模型标识符model_id加载分词器tokenizer
tokenizer=_____
#TODO: 利用transformers库函数从预训练模型加载自回归语言模型，配置模型的数据类型为torch.float16、自动选择设备映射，并关闭安全张量选项
model = _____
#TODO: 将模型设置为评估模式
model =_____
```

2）**文本生成和性能统计模块**

文本生成和性能统计模块的代码示例如 8.18所示，主要用于展示模型生成文本的性能及效果，并对其进行了时间和性能方面的统计。具体而言，首先定义了一个包含函数描述的 prompt，其中有一个占位符 <FILL_ME>。接着，使用 time 库记录了代码执行的起始时间点 t1。通过 tokenizer 将 prompt 转化为模型可接受的输入格式，并在 MLU 上进行处理。随后，利用预训练模型 model 生成相应的文本输出，并将输出从 MLU 转回 CPU。最后，计算了代码执行的时间差，输出文本的长度以及每秒生成的 token 数量，并通过 print 语句展示这些信息。

代码示例 8.18　文本生成和性能统计模块

```
# file: Codellama-infer.py
prompt = '''def remove_non_ascii(s: str) -> str:
    """ <FILL_ME>
    return result
'''
import time
```

```
t1 = time.perf_counter()
#TODO:# 使用tokenizer将输入的文本prompt进行分词，并返回PyTorch张量表示的input_ids
input_ids = ____________________________________________________
#TODO: 将input_ids张量移动到MLU（Ascend）设备上
input_ids = ____________________________________________________
#TODO: 使用模型生成文本序列，给定输入张量 input_ids，限制最大生成标记数为 240，启用缓存机制
output = _______________________________________________________
#TODO: 将模型生成的结果从MLU设备移动到CPU设备
output = _______________________________________________________
t2 = time.perf_counter()
#TODO:计算推理延迟
latency =_______________________________________________________
#TODO:计算输出的token数量
output_token_num = _____________________________________________
print('time cost:{} s, output_token_num:{}, total throughout:{} token/s'.\
    format(latency, output_token_num, output_token_num / latency))
```

3) **结果处理和打印模块**

结果处理和打印的代码示例如 8.19所示。这段代码使用 tokenizer 将模型生成的文本 output 还原为自然语言，并将其中的填充文本替换回原始 prompt 中的占位符 <FILL_ME>。最终，通过 print 语句展示了填充完成后的完整文本，实现了生成文本的最终展示效果。

代码示例 8.19　结果处理和打印

```
# file: Codellama-infer.py
#TODO:通过解码生成的张量，获取填充后的文本，跳过特殊标记
filling = ______________________________________________________
#TODO: 打印替换了占位符"<FILL_ME>"的完整代码
print(_________________________________________________________)
```

8.3.5.2　Transformer 模型构建模块

Transformer 模型构建模块实现了一个高效的 Transformer 模型，具备位置编码、多头注意力机制、前馈神经网络等基本组件，可用于处理自然语言处理任务，如文本分类、语言建模等。接下来对模型并行化于初始化模块、RMS 归一化模块、函数工具模块、Transformer 组件模块、Transformer 整体构造模块组成。

1) **模型并行化与初始化模块**

代码示例8.20实现了一个 Transformer 模型，并使用 fairscale 库进行模型并行化。它包括了注意力机制、前馈神经网络等基本组件，支持 MLU 和 MPS 设备。通过数据类 ModelArgs 定义了模型的参数，如维度、层数、头数等。代码中还包括了 RMS 归一化、位置编码等功能。整体而言，这段代码为构建高效的 Transformer 模型提供了基础框架，并支持在不同硬件设备上的部署。

代码示例 8.20　模型并行化与初始化模块

```
# file: llama_mlu/model.py
import math
from dataclasses import dataclass
from typing import Any, Optional, Tuple

import fairscale.nn.model_parallel.initialize as fs_init
import torch
import torch_mlu
```

```python
import torch.nn.functional as F
from fairscale.nn.model_parallel.layers import (
    ColumnParallelLinear,
    ParallelEmbedding,
    RowParallelLinear,
)
from torch import nn
#TODO:检查是否有 MLU设备可用，如果可用，则将设备类型设置为 "mlu"
if _____:
    device = _____
elif torch.backends.mps.is_available():
    device = "mps"
else:
    device = "cpu"

@dataclass
class ModelArgs:
    dim: int = 4096
    n_layers: int = 32
    n_heads: int = 32
    n_kv_heads: Optional[int] = None
    vocab_size: int = -1  # defined later by tokenizer
    multiple_of: int = 256  # make SwiGLU hidden layer size multiple of large power of 2
    ffn_dim_multiplier: Optional[float] = None
    norm_eps: float = 1e-5
    rope_theta: float = 10000

    max_batch_size: int = 32
    max_seq_len: int = 2048
```

2) **RMS 归一化模块**

代码示例8.21定义了一个带可学习参数的 RMS 归一化（RMSNorm）模块。在初始化时，它接受维度和 eps 作为参数，并创建了一个可学习的权重参数。在前向传播过程中，它对输入张量进行了归一化处理，通过计算输入张量沿着指定维度的平方的均值再开方，然后用该值对输入进行除法运算。最后，乘以学习到的权重参数，以加入可学习的缩放因子。这个模块可以被集成到神经网络中，用于增强模型的数值稳定性和收敛速度。RMS 的运算公式如9.13所示。其中，x 表示输入张量，即需要进行归一化处理的数据；mean 表示求取张量元素的平均值的函数；ϵ 表示平滑项（epsilon），用于避免除以零的情况发生。

$$\text{RMSNorm}(x) = x \cdot \frac{1}{\sqrt{\text{mean}(x^2) + \epsilon}} \tag{8.13}$$

代码示例 8.21　RMS 归一化模块

```python
# file: llama_mlu/model.py
class RMSNorm(torch.nn.Module):
    def __init__(self, dim: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        #TODO: 初始化可学习的参数 weight，维度为 dim
        self.weight =_____

    def _norm(self, x):
        #TODO:  计算RMS 归一化
        return _____

    def forward(self, x):
```

```
        #TODO: 执行 RMS 归一化，并将结果的数据类型设为与输入张量 x 一致
        output = ____________________________________________________
        return output * self.weight
```

3) **函数工具模块**

函数工具模块中包含 precompute_freqs_cis 函数、reshape_for_broadcast 函数、apply_rotary_emb 函数、以及 repeat_kv 函数，如代码示例8.22所示。具体而言：

- **precompute_freqs_cis 函数**：用于预先计算旋转注意力机制中的频率。它首先计算频率向量，然后根据给定的维度和结束位置生成频率矩阵。最后，使用极坐标形式计算频率的复数表示，并返回计算后的频率矩阵。
- **reshape_for_broadcast 函数**：用于调整频率矩阵的形状，以便与输入张量进行广播。它会检查输入张量的维度，并确保频率矩阵的形状与输入张量的中间维度和最后维度相匹配，然后返回调整后的频率矩阵。
- **apply_rotary_emb 函数**：实现了旋转操作，将输入的查询和键张量与频率矩阵进行旋转。它首先将输入张量转换为复数形式，然后根据频率矩阵调整形状并执行旋转操作。最后，将旋转后的结果转换为实数形式，并返回旋转后的查询和键张量。
- **repeat_kv 函数**：用于重复键值张量以适应注意力机制中的多头操作。它会检查重复次数，如果重复次数为 1，则直接返回输入张量。否则，将输入张量扩展一个新的维度，并重复该维度以得到多头的键值张量，最后将结果重新调整形状并返回。

代码示例 8.22　函数工具模块

```
# file: llama_mlu/model.py
def precompute_freqs_cis(dim: int, end: int, theta: float = 10000.0):
    freqs = 1.0 / (theta ** (torch.arange(0, dim, 2)[: (dim // 2)].float() / dim))
    #TODO: 创建建了一个张量 t，包含从 0 到 end - 1 的整数值，并根据 freqs 的设备来设置设备属性。
    t = ____________________________________________________________
    #TODO: 将 t 和 freqs 的值计算出外积
    freqs = ____________________________________________________________
    #TODO: 将频率值转换为复数形式的张量，其中每个元素表示一个复数，其幅度为 1，角度由 freqs 张量给出
    freqs_cis = ____________________________________________________________
    return freqs_cis


def reshape_for_broadcast(freqs_cis: torch.Tensor, x: torch.Tensor):
    ndim = x.ndim
    assert 0 <= 1 < ndim
    assert freqs_cis.shape == (x.shape[1], x.shape[-1])
    shape = [d if i == 1 or i == ndim - 1 else 1 for i, d in enumerate(x.shape)]
    return freqs_cis.view(*shape)


def apply_rotary_emb(
    xq: torch.Tensor,
    xk: torch.Tensor,
    freqs_cis: torch.Tensor,
) -> Tuple[torch.Tensor, torch.Tensor]:
    if not torch.mlu.is_available():
        xq = xq.to('cpu')
        xk = xk.to('cpu')
    xq_ = torch.view_as_complex(xq.float().reshape(*xq.shape[:-1], -1, 2))
    xk_ = torch.view_as_complex(xk.float().reshape(*xk.shape[:-1], -1, 2))
    #TODO: 将旋转频率张量调整为与输入张量xq_广播兼容的形状
    freqs_cis = ____________________________________________
```

```
    xq_out = torch.view_as_real(xq_ * freqs_cis).flatten(3)
    xk_out = torch.view_as_real(xk_ * freqs_cis).flatten(3)
    return xq_out.type_as(xq).to(device), xk_out.type_as(xk).to(device)


def repeat_kv(x: torch.Tensor, n_rep: int) -> torch.Tensor:
    """torch.repeat_interleave(x, dim=2, repeats=n_rep)"""
    bs, slen, n_kv_heads, head_dim = x.shape
    if n_rep == 1:
        return x
    return (
        x[:, :, :, None, :]
        .expand(bs, slen, n_kv_heads, n_rep, head_dim)
        .reshape(bs, slen, n_kv_heads * n_rep, head_dim)
    )
```

4) **Transformer 组件模块**

Transformer 组件模块主要由注意力机制模块、前馈神经网络模块、TransformerBlock 模块组成，代码示例如8.23所示。

- **注意力机制模块**：注意力机制模块顾名思义，用于实现自注意力机制。该模块接受一个包含输入张量 x、起始位置 start_pos、频率参数 freqs_cis 和掩码 mask 的输入，并通过自注意力机制计算出输出张量。在初始化过程中，通过设置不同的线性层来计算查询（query）、键（key）和值（value），然后将它们分别转换为本地头（local heads）形式。通过调用 apply_rotary_emb 函数对查询和键进行旋转，然后通过缓存机制将它们存储在 cache_k 和 cache_v 张量中。在前向传播过程中，计算查询和键之间的分数，然后应用 softmax 函数得到注意力权重。最后，利用注意力权重加权求和值，然后通过输出线性层得到最终的输出张量。注意力机制的表达式可以用8.14表示。

$$\text{Attention}(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{8.14}$$

- **前馈神经网络模块**：FeedForward 的神经网络模块，用于实现 Transformer 中的前馈神经网络（Feed Forward Network）。在初始化过程中，根据给定的参数，计算隐藏层的维度，并创建了三个线性层，分别为 w1、w2 和 w3。在前向传播过程中，输入通过第一个线性层 w1，并经过激活函数 SiLU（Sigmoid-weighted Linear Unit），然后与第三个线性层 w3 的输出相乘，最终通过第二个线性层 w2 得到输出。
- **TransformerBlock 模块**：TransformerBlock 为 Transformer 模型的基本组成单元。该模块包含了一个自注意力层（Attention）和一个前馈神经网络层（FeedForward）。在前向传播过程中，输入先经过自注意力层和残差连接，然后再经过前馈神经网络层和残差连接，最终输出。TransformerBlock 作为 Transformer 模型的核心组件之一，在序列建模任务中扮演着重要的角色，能够有效地捕获序列中的长程依赖关系和语义信息。

代码示例 8.23　Transformer 组件模块

```
# file: llama_mlu/model.py

class Attention(nn.Module):
    def __init__(self, args: ModelArgs):
        super().__init__()
        self.n_kv_heads = args.n_heads if args.n_kv_heads is None else args.n_kv_heads
```

```
        model_parallel_size = fs_init.get_model_parallel_world_size()
        self.n_local_heads = args.n_heads // model_parallel_size
        self.n_local_kv_heads = self.n_kv_heads // model_parallel_size
        self.n_rep = self.n_local_heads // self.n_local_kv_heads
        self.head_dim = args.dim // args.n_heads

        self.wq = ColumnParallelLinear(
            args.dim,
            args.n_heads * self.head_dim,
            bias=False,
            gather_output=False,
            init_method=lambda x: x,
        )
        self.wk = ColumnParallelLinear(
            args.dim,
            self.n_kv_heads * self.head_dim,
            bias=False,
            gather_output=False,
            init_method=lambda x: x,
        )
        self.wv = ColumnParallelLinear(
            args.dim,
            self.n_kv_heads * self.head_dim,
            bias=False,
            gather_output=False,
            init_method=lambda x: x,
        )
        self.wo = RowParallelLinear(
            args.n_heads * self.head_dim,
            args.dim,
            bias=False,
            input_is_parallel=True,
            init_method=lambda x: x,
        )

        self.cache_k = torch.zeros(
            (
                args.max_batch_size,
                args.max_seq_len,
                self.n_local_kv_heads,
                self.head_dim,
            )
        ).to(device)
        self.cache_v = torch.zeros(
            (
                args.max_batch_size,
                args.max_seq_len,
                self.n_local_kv_heads,
                self.head_dim,
            )
        ).to(device)

    def forward(
        self,
        x: torch.Tensor,
        start_pos: int,
        freqs_cis: torch.Tensor,
        mask: Optional[torch.Tensor],
    ):
        bsz, seqlen, _ = x.shape
        #TODO: 获取Q、K、V
        xq, xk, xv = _____________________________________________________
```

```
69
70          xq = xq.view(bsz, seqlen, self.n_local_heads, self.head_dim)
71          xk = xk.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)
72          xv = xv.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)
73          #TODO: 对Q和K应用旋转嵌入
74          xq, xk = ____________________________________________________
75
76          self.cache_k = self.cache_k.to(xq)
77          self.cache_v = self.cache_v.to(xq)
78
79          self.cache_k[:bsz, start_pos : start_pos + seqlen] = xk
80          self.cache_v[:bsz, start_pos : start_pos + seqlen] = xv
81
82          keys = self.cache_k[:bsz, : start_pos + seqlen]
83          values = self.cache_v[:bsz, : start_pos + seqlen]
84
85          # repeat k/v heads if n_kv_heads < n_heads
86          keys = repeat_kv(keys, self.n_rep)  # (bs, seqlen, n_local_heads, head_dim)
87          values = repeat_kv(values, self.n_rep)  # (bs, seqlen, n_local_heads, head_dim)
88
89          xq = xq.transpose(1, 2)  # (bs, n_local_heads, seqlen, head_dim)
90          keys = keys.transpose(1, 2)
91          values = values.transpose(1, 2)
92          #TODO: 根据注意力机制的公式计算查询张量 xq 与键张量 keys 的点积注意力得分,并进行缩放
93          scores = ____________________________________________________________
94          if mask is not None:
95              scores = scores + mask  # (bs, n_local_heads, seqlen, cache_len + seqlen)
96          #TODO:使用 softmax 函数将注意力分数 scores 沿着最后一个维度进行归一化，并将结果转换为与输入张
        量 xq 相同的数据类型以获得注意力权重
97          scores = ____________________________________________________
98          #TODO: 将注意力权重与值相乘，得到最终输出
99          output = ____________________________________________ # (bs, n_local_heads, seqlen,
        head_dim)
100         output = output.transpose(1, 2).contiguous().view(bsz, seqlen, -1)
101         return self.wo(output)
102
103 class FeedForward(nn.Module):
104     def __init__(
105         self,
106         dim: int,
107         hidden_dim: int,
108         multiple_of: int,
109         ffn_dim_multiplier: Optional[float],
110     ):
111         super().__init__()
112         hidden_dim = int(2 * hidden_dim / 3)
113         # custom dim factor multiplier
114         if ffn_dim_multiplier is not None:
115             hidden_dim = int(ffn_dim_multiplier * hidden_dim)
116         hidden_dim = multiple_of * ((hidden_dim + multiple_of - 1) // multiple_of)
117
118         self.w1 = ColumnParallelLinear(
119             dim, hidden_dim, bias=False, gather_output=False, init_method=lambda x: x
120         )
121         self.w2 = RowParallelLinear(
122             hidden_dim, dim, bias=False, input_is_parallel=True, init_method=lambda x: x
123         )
124         self.w3 = ColumnParallelLinear(
125             dim, hidden_dim, bias=False, gather_output=False, init_method=lambda x: x
126         )
127
128     def forward(self, x):
```

```
        #TODO: 补全前馈神经网络的前向传播过程
        return _____________________________________________________


class TransformerBlock(nn.Module):
    def __init__(self, layer_id: int, args: ModelArgs):
        super().__init__()
        self.n_heads = args.n_heads
        self.dim = args.dim
        self.head_dim = args.dim // args.n_heads
        #TODO: 添加注意力层
        self.attention = _____________________________________________
        #TODO: 创建前馈网络层
        self.feed_forward = _____________________(
            dim=args.dim,
            hidden_dim=4 * args.dim,
            multiple_of=args.multiple_of,
            ffn_dim_multiplier=args.ffn_dim_multiplier,
        )
        self.layer_id = layer_id
        #TODO:添加RMS归一化
        self.attention_norm = _____________________________________________
        self.ffn_norm = ____________________________________________

    def forward(
        self,
        x: torch.Tensor,
        start_pos: int,
        freqs_cis: torch.Tensor,
        mask: Optional[torch.Tensor],
    ):
        h = x + self.attention.forward(
            self.attention_norm(x), start_pos, freqs_cis, mask
        )
        out = h + self.feed_forward.forward(self.ffn_norm(h))
        return out
```

5) **Transformer 整体构造模块**

Transformer 整体构造模块定义了一个完整的 Transformer 模型，其中包含多个 TransformerBlock 组成的堆叠层，代码示例如8.24所示。模型的输入是一个 token 序列，通过 token embeddings 转换为向量表示后，经过一系列 TransformerBlock 层的处理，最终输出一个经过线性变换的概率分布，表示每个 token 的生成概率。Transformer 模型通过自注意力机制和前馈神经网络层来有效地捕获输入序列中的信息，并生成符合语言模型目标的输出序列。

代码示例 8.24　Transformer 整体构造模块

```
# file: llama_mlu/model.py
class Transformer(nn.Module):
    def __init__(self, params: ModelArgs):
        super().__init__()
        self.params = params
        self.vocab_size = params.vocab_size
        self.n_layers = params.n_layers

        self.tok_embeddings = ParallelEmbedding(
            params.vocab_size, params.dim, init_method=lambda x: x,
        )
        #TODO: 创建PyTorch 中用于存储子模块的容器
        self.layers = ____________________________________________________
```

```
        for layer_id in range(params.n_layers):
            #TODO: 将 TransformerBlock 的层添加到模型的层列表中。
            _____________________________________________
        #TODO: 初始化RMS归一化层
        self.norm = _____________________________________________
        self.output = ColumnParallelLinear(
            params.dim, params.vocab_size, bias=False, init_method=lambda x: x
        )
        #调用函数计算用于多头自注意力机制中的频率值。
        self.freqs_cis = _____________________(
            self.params.dim // self.params.n_heads,
            self.params.max_seq_len * 2,
            params.rope_theta,
        )

    @torch.inference_mode()
    def forward(self, tokens: torch.Tensor, start_pos: int):
        _bsz, seqlen = tokens.shape
        #TODO: 获取 tokens的embeddings
        h = _____________________________________________________
        #TODO: 将 self.freqs_cis 张量移动到 MLU 或 CPU 设备上，以便在该设备上进行后续计算
        self.freqs_cis = _____________________________________________________
        #TODO: 从 self.freqs_cis 中提取一个长度为 seqlen 的子张量，其初始位置为start_pos
        freqs_cis =_____________________________________________________

        mask = None
        if seqlen > 1:
            mask = torch.full(
                (1, 1, seqlen, seqlen), float("-inf"), device=torch.device('cpu')
            )
            mask = mask.to(torch.float32).triu(diagonal=start_pos+1).type_as(h)

        for layer in self.layers:
            h = layer(h, start_pos, freqs_cis, (mask.to(device) if mask is not None else mask))
        h = self.norm(h)
        output = self.output(h).float()
        return output
```

8.3.5.3 Llama 代码生成模块

Llama 代码生成模块用于实现文本生成、文本补全和指令微调。它依赖于 PyTorch 库和其他自定义模块。主要功能包括加载预训练的 Transformer 模型和 Tokenizer，然后使用这些模型和 Tokenizer 执行代码生成任务。它通过调用 generate 方法来实现这些功能，该方法接受文本提示，并生成对应的文本输出。此外，它还支持设置不同的温度和 top-p 参数以调整生成文本的多样性和准确性。接下来主要分为硬件设备及类型定义模块、Llama 函数模块、辅助函数模块进行具体介绍。

1) 硬件设备及类型定义模块

代码示例8.25定义了一个名为 Message 的 TypedDict 类，用于描述对话中的消息，包括角色、内容和目标。此外，还定义了几个 TypedDict 类来表示填充、完成和对话生成的预测结果。代码还包括了一些常量和错误消息，以及一些用于特殊标记和安全检查的辅助功能。其中，需要根据系统环境选择合适的设备来运行后续的代码，以实现更好的性能和兼容性。

代码示例 8.25　硬件设备及类型定义模块

```python
# file: llama_mlu/generation.py
import json
import os
import sys
import time
from pathlib import Path
#from typing import List, Literal, Optional, Tuple, TypedDict
from typing_extensions import List, Literal, Optional, Tuple, TypedDict

import torch
import torch_mlu
import torch.nn.functional as F
from fairscale.nn.model_parallel.initialize import (
    get_model_parallel_rank,
    initialize_model_parallel,
    model_parallel_is_initialized,
)

from llama_mlu.model import ModelArgs, Transformer
from llama_mlu.tokenizer import Tokenizer


#TODO: 如果 MLU 设备可用
_____________________________________
#TODO: 将设备设置为 MLU
    device = _____________________________________
else:
#TODO: 将设备设置为 CPU
    device = _____________________________________

Role = Literal["system", "user", "assistant"]


class Message(TypedDict):
    role: Role
    content: str
    destination: str  # required for model responses


class InfillingPrediction(TypedDict, total=False):
    generation: str
    full_text: str
    tokens: List[str]  # not required
    logprobs: List[float]  # not required


class CompletionPrediction(TypedDict, total=False):
    generation: str
    tokens: List[str]  # not required
    logprobs: List[float]  # not required


class ChatPrediction(TypedDict, total=False):
    generation: Message
    tokens: List[str]  # not required
    logprobs: List[float]  # not required


Dialog = List[Message]

B_INST, E_INST = "[INST]", "[/INST]"
B_SYS, E_SYS = "<<SYS>>\n", "\n<</SYS>>\n\n"
```

```
SPECIAL_TAGS = [B_INST, E_INST, "<<SYS>>", "<</SYS>>", "<step>"]
UNSAFE_ERROR = "Error: special tags are not allowed as part of the prompt."
```

2) **Llama 函数模块**

主要分为模型初始化与构建模块、生成模块、文本生成模块、文本填充模块、文本对话模块、内部辅助功能模块进行介绍。

- **模型初始化与构建模块**

模型初始化与构建模块的代码示例如8.26所示，主要包括两个重要的函数：

a. build 函数用于构建一个 Llama 对象，该对象包含一个预训练的 Transformer 模型和相应的分词器。该方法首先初始化分布式进程组，然后初始化模型并行，加载模型检查点，并设置模型的张量类型。最后，返回一个包含已构建模型和分词器的 Llama 对象。

b. __init__ 函数用于初始化 Llama 类的实例，接受一个 Transformer 模型和一个分词器作为参数，并将它们存储在实例中供后续使用。

以上函数结合起来提供了一个方便的接口，用于加载预训练的 Transformer 模型、构建相应的分词器，并将它们封装到一个 Llama 对象中，以便进行文本生成和处理任务。

代码示例 8.26　模型初始化与构建模块

```
# file: llama_mlu/generation.py
class Llama:
    @staticmethod
    def build(
        ckpt_dir: str,
        tokenizer_path: str,
        max_seq_len: int,
        max_batch_size: int,
        model_parallel_size: Optional[int] = None,
    ) -> "Llama":
        if not torch.distributed.is_initialized():
            if device == "mlu":
                #TODO: 使用 MLU 设备初始化分布式进程组
                ------------------------------------------------
            else:
                torch.distributed.init_process_group("gloo")
        if not model_parallel_is_initialized():
            if model_parallel_size is None:
                model_parallel_size = int(os.environ.get("WORLD_SIZE", 1))
            initialize_model_parallel(model_parallel_size)

        local_rank = int(os.environ.get("LOCAL_RANK", 0))
        if device == "mlu":
            #TODO:  如果设备为 MLU，则设置当前进程的 MLU 设备
            ------------------------------------------------

        # seed must be the same in all processes
        torch.manual_seed(1)

        if local_rank > 0:
            sys.stdout = open(os.devnull, "w")

        start_time = time.time()
        checkpoints = sorted(Path(ckpt_dir).glob("*.pth"))
        assert len(checkpoints) > 0, f"no checkpoint files found in {ckpt_dir}"
        assert model_parallel_size == len(
            checkpoints
```

```
        ), f"Loading a checkpoint for MP={len(checkpoints)} but world size is {model_parallel_size}"
        ckpt_path = checkpoints[get_model_parallel_rank()]
        #TODO: 加载模型的检查点文件，并将模型加载到 CPU 上。
        checkpoint = ____________________________________________________
        with open(Path(ckpt_dir) / "params.json", "r") as f:
            params = json.loads(f.read())

        model_args: ModelArgs = ModelArgs(
            max_seq_len=max_seq_len,
            max_batch_size=max_batch_size,
            **params,
        )
        #TODO: 调用Tokenizer函数
        tokenizer = _____________________________________________
        model_args.vocab_size = tokenizer.n_words
        # support for mac
        print(device)
        if device == "mlu":
            torch.set_default_tensor_type(torch.HalfTensor)
            #if torch.mlu.is_bf16_supported():
            #    torch.set_default_tensor_type(torch.mlu.BFloat16Tensor)
            #else:
            #    torch.set_default_tensor_type(torch.mlu.HalfTensor)
        else:
            torch.set_default_tensor_type(torch.HalfTensor)
        #TODO: 调用Transformer 模型
        model =  ____________________________________________
        #TODO: 加载模型的参数字典
        ____________________________________________________
        #add start
        print(device)
        if device == "cpu":
            model = model.float()

        #add end
        model.to(device)
        print(f"Loaded in {time.time() - start_time:.2f} seconds")

        return Llama(model, tokenizer)

    def __init__(self, model: Transformer, tokenizer: Tokenizer):
        self.model = model
        self.tokenizer = tokenizer
```

- **生成模块**

生成模块代码示例如8.27所示。它接受一个或多个提示（tokenized prompts），并基于这些提示生成文本序列。该方法会根据给定的最大生成长度（max_gen_len）以及温度（temperature）和 top-p 采样（top_p）参数生成文本。如果指定了停止标记（stop_token），则生成过程将在遇到停止标记后结束。该方法还可以选择是否返回生成文本的对数概率（logprobs）。生成过程中，方法会在每次迭代中预测下一个 token，并将其添加到生成的文本序列中。生成的文本序列和对应的对数概率（如果选择了 logprobs）将作为输出返回。

代码示例 8.27　生成模块

```
# file: llama_mlu/generation.py
    @torch.inference_mode()
    def generate(
        self,
        prompt_tokens: List[List[int]],
```

```
        max_gen_len: int,
        temperature: float = 0.6,
        top_p: float = 0.9,
        logprobs: bool = False,
        echo: bool = False,
        stop_token: Optional[int] = None,
    ) -> Tuple[List[List[int]], Optional[List[List[float]]]]:
        if stop_token is None:
            stop_token = self.tokenizer.eos_id
        params = self.model.params
        bsz = len(prompt_tokens)
        assert bsz <= params.max_batch_size, (bsz, params.max_batch_size)
        #TODO:  获取提示文本序列中最短的长度
        min_prompt_len = _____________________________________________________
        #TODO: 获取提示文本序列中最长的长度
        max_prompt_len = _____________________________________________________
        assert max_prompt_len <= params.max_seq_len
        #TODO: 计算生成的总长度，需考虑提示文本和最大生成长度
        total_len = _____________________________________________________

        pad_id = self.tokenizer.pad_id
        tokens = torch.full((bsz, total_len), pad_id, dtype=torch.long, device=device)
        for k, t in enumerate(prompt_tokens):
            #TODO: 将提示文本编码添加到张量中
            tokens[k, : len(t)] = _____________________________________________________
        if logprobs:
            #TODO: 创建一个与tokens张量具有相同形状的全零张量
            token_logprobs = _____________________________________________________
        prev_pos = 0
        stop_reached = torch.tensor([False] * bsz, device=device)
        input_text_mask = tokens != pad_id
        for cur_pos in range(min_prompt_len, total_len):
            logits = self.model.forward(tokens[:, prev_pos:cur_pos], prev_pos)
            if logprobs:
                token_logprobs[:, prev_pos + 1 : cur_pos + 1] = -F.cross_entropy(
                    input=logits.transpose(1, 2),
                    target=tokens[:, prev_pos + 1 : cur_pos + 1],
                    reduction="none",
                    ignore_index=pad_id,
                )
            if temperature > 0:
                #TODO: 对模型的输出进行 softmax 归一化，以得到每个可能的下一个token的概率分布，其中 temperature 用于控制模型输出的多样性
                probs = _____________________________________________________
                #TODO: 根据概率分布采样出下一个 token
                next_token = _____________________________________________________
            else:
                #TODO: 直接选择logits最大的位置作为下一个token，不进行随机采样
                next_token = _____________________________________________________

            next_token = next_token.reshape(-1)
            # only replace token if prompt has already been generated
            next_token = torch.where(
                input_text_mask[:, cur_pos], tokens[:, cur_pos], next_token
            )
            tokens[:, cur_pos] = next_token
            stop_reached |= (~input_text_mask[:, cur_pos]) & (next_token == stop_token)
            prev_pos = cur_pos
            if all(stop_reached):
                break

        if logprobs:
```

```
            #TODO: 将张量转换为列表格式
            token_logprobs = ____________________________________________________
        out_tokens, out_logprobs = [], []
        for i, toks in enumerate(tokens.tolist()):
            # cut to max gen len
            start = 0 if echo else len(prompt_tokens[i])
            #TODO: 截取生成的标记序列，直到达到最大生成长度
            toks = ______________________________________________________
            probs = None
            if logprobs:
                probs = token_logprobs[i][start : len(prompt_tokens[i]) + max_gen_len]
            # cut to stop token if present
            if stop_token in toks:
                stop_idx = toks.index(stop_token)
                toks = toks[:stop_idx]
                probs = probs[:stop_idx] if logprobs else None
            #TODO: 将截取后的标记序列添加到输出列表中
            ____________________________________________________
            #TODO: 将截取后的log概率列表添加到输出列表中
            ____________________________________________________
        return (out_tokens, out_logprobs if logprobs else None)
```

- **文本生成模块**

文本生成模块代码示例如9.36所示，用于完成给定的文本提示。该方法接受一组文本提示作为输入，并使用预训练的 Transformer 模型生成相应的文本完成结果。可以通过调整温度（temperature）和 top-p 参数来控制生成文本的多样性和可控性。如果设置了 logprobs 参数为 True，则返回的结果中将包含每个生成文本的对数概率值。最终，该方法返回一个列表，其中每个元素包含生成的文本完成结果以及相应的 tokenized 形式和对数概率值（如果指定了 logprobs 参数）。

代码示例 8.28　文本生成模块

```
# file: llama_mlu/generation.py
    def text_completion(
        self,
        prompts: List[str],
        temperature: float = 0.6,
        top_p: float = 0.9,
        max_gen_len: Optional[int] = None,
        logprobs: bool = False,
        echo: bool = False,
    ) -> List[CompletionPrediction]:
        if max_gen_len is None:
            max_gen_len = self.model.params.max_seq_len - 1
        prompt_tokens = [self.tokenizer.encode(x, bos=True, eos=False) for x in prompts]
        #TODO: 调用 generate 方法生成文本
        generation_tokens, generation_logprobs = ____________________________________
        if logprobs:
            assert generation_logprobs is not None
            return [
                {
                    "generation": self.tokenizer.decode(t),
                    "tokens": [self.tokenizer.token_piece(x) for x in t],
                    "logprobs": logprobs_i,
                }
                for t, logprobs_i in zip(generation_tokens, generation_logprobs)
            ]
        return [{"generation": self.tokenizer.decode(t)} for t in generation_tokens]
```

- **文本填充模块**

文本填充模块的代码示例如 8.29，其作用是对给定的前缀和后缀进行文本填充（infilling），生成完整的文本。该方法首先确保分词器具有结束标记，然后根据给定的前缀和后缀生成对应的填充文本，并调用生成模块生成填充后的文本序列。接着，将生成的文本序列解码为完整的文本，并根据需要返回带有日志概率和填充文本的信息或仅返回填充后的完整文本。该方法提供了一种简便的方式来对文本进行填充操作，可用于语言模型的生成任务中。

代码示例 8.29　文本填充模块

```python
# file: llama_mlu/generation.py
    def text_infilling(
        self,
        prefixes: List[str],
        suffixes: List[str],
        temperature: float = 0.6,
        top_p: float = 0.9,
        max_gen_len: Optional[int] = None,
        logprobs: bool = False,
        suffix_first: bool = False,
    ) -> List[InfillingPrediction]:
        assert self.tokenizer.eot_id is not None
        if max_gen_len is None:
            max_gen_len = self.model.params.max_seq_len - 1
        prompt_tokens = [
            #TODO: 调用函数对每个前缀和后缀进行处理，生成填充问题的编码
            ____________________________________________________________
            for prefix, suffix in zip(prefixes, suffixes)
        ]
        #TODO: 调用 generate 方法生成文本
        generation_tokens, generation_logprobs = _____________________________________________

        generations = [self.tokenizer.decode_infilling(t) for t in generation_tokens]

        if logprobs:
            assert generation_logprobs is not None
            return [
                {
                    "generation": generation,
                    "logprobs": logprobs_i,
                    "tokens": [self.tokenizer.token_piece(x) for x in t],
                    "full_text": prefix + generation + suffix,
                }
                for prefix, suffix, generation, t, logprobs_i in zip(
                    prefixes,
                    suffixes,
                    generations,
                    generation_tokens,
                    generation_logprobs,
                )
            ]
        else:
            return [
                {
                    "generation": generation,
                    "full_text": prefix + generation + suffix,
                }
                for prefix, suffix, generation in zip(prefixes, suffixes, generations)
            ]
```

- **文本对话模块**

文本对话模块如代码示例8.30所示，它用于处理对话文本的自动完成，主要由两个函数组成。其中，chat_completion 函数用于处理用户与助手之间的对话，生成助手的响应。如果对话中存在步骤标记，则调用 _chat_completion_turns 函数，否则将对话分为系统消息和用户消息，然后生成助手的响应。_chat_completion_turns 函数处理具有步骤标记的对话，确保适当地插入系统消息，并生成助手的响应。这两个函数的输出是包含对话中生成文本的字典列表，其中包括生成文本、生成文本的标记和对应的对数概率（如果指定了）。

代码示例 8.30　文本对话模块

```
# file: llama_mlu/generation.py
    def chat_completion(
        self,
        dialogs: List[Dialog],
        temperature: float = 0.6,
        top_p: float = 0.9,
        max_gen_len: Optional[int] = None,
        logprobs: bool = False,
    ) -> List[ChatPrediction]:
        if self.tokenizer.step_id is not None:
            ## 如果模型支持 step_id，则使用另一种chat_completion的方法
            return _____________________________________________________
        if max_gen_len is None:
            max_gen_len = self.model.params.max_seq_len - 1
        prompt_tokens = []
        unsafe_requests = []
        for dialog in dialogs:
            unsafe_requests.append(
                any([tag in msg["content"] for tag in SPECIAL_TAGS for msg in dialog])
            )
            if dialog[0]["role"] == "system":
                dialog = [  # type: ignore
                    {
                        "role": dialog[1]["role"],
                        "content": B_SYS
                        + dialog[0]["content"]
                        + E_SYS
                        + dialog[1]["content"],
                    }
                ] + dialog[2:]
            assert all([msg["role"] == "user" for msg in dialog[::2]]) and all(
                [msg["role"] == "assistant" for msg in dialog[1::2]]
            ), (
                "model only supports 'system', 'user' and 'assistant' roles, "
                "starting with 'system', then 'user' and alternating (u/a/u/a/u...)"
            )
            dialog_tokens: List[int] = sum(
                [
                    self.tokenizer.encode(
                        f"{B_INST} {prompt['content'].strip()} {E_INST} {answer['content'].strip()} ",
                        bos=True,
                        eos=True,
                    )
                    for prompt, answer in zip(
                        dialog[::2],
                        dialog[1::2],
                    )
                ],
                [],
```

```
            )
            assert (
                dialog[-1]["role"] == "user"
            ), f"Last message must be from user, got {dialog[-1]['role']}"
            dialog_tokens += self.tokenizer.encode(
                f"{B_INST} {dialog[-1]['content'].strip()} {E_INST}",
                bos=True,
                eos=False,
            )
            prompt_tokens.append(dialog_tokens)
        #TODO: 调用 generate 方法生成文本
        generation_tokens, generation_logprobs = ________________________________________________
        if logprobs:
            assert generation_logprobs is not None
            return [
                {
                    "generation": {  # type: ignore
                        "role": "assistant",
                        "content": self.tokenizer.decode(t)
                        if not unsafe
                        else UNSAFE_ERROR,
                    },
                    "tokens": [self.tokenizer.token_piece(x) for x in t],
                    "logprobs": logprobs_i,
                }
                for t, logprobs_i, unsafe in zip(
                    generation_tokens, generation_logprobs, unsafe_requests
                )
            ]
        return [
            {
                "generation": {  # type: ignore
                    "role": "assistant",
                    "content": self.tokenizer.decode(t) if not unsafe else UNSAFE_ERROR,
                }
            }
            for t, unsafe in zip(generation_tokens, unsafe_requests)
        ]

    def _chat_completion_turns(
        self,
        dialogs: List[Dialog],
        temperature: float = 0.6,
        top_p: float = 0.9,
        max_gen_len: Optional[int] = None,
        logprobs: bool = False,
    ) -> List[ChatPrediction]:
        if self.tokenizer.step_id is None:
            raise RuntimeError("Model not suitable for chat_completion_step()")
        if max_gen_len is None:
            max_gen_len = self.model.params.max_seq_len - 1

        prompt_tokens = []
        unsafe_requests = []
        for dialog in dialogs:
            unsafe_requests.append(
                any([tag in msg["content"] for tag in SPECIAL_TAGS for msg in dialog])
            )

            # Insert system message if not provided
            if dialog[0]["role"] != "system":
                dialog = [{"role": "system", "content": ""}] + dialog  # type: ignore
```

```
            #TODO:调用函数将对话格式化为模型可处理的对话提示编码
            dialog_tokens = ____________________________________________________________
            prompt_tokens.append(dialog_tokens)
        #TODO: 调用 generate 方法生成文本
        generation_tokens, generation_logprobs = ____________________________________________________
        if logprobs:
            assert generation_logprobs is not None
            return [
                {
                    "generation": {
                        "role": "assistant",
                        "destination": "user",
                        "content": self.tokenizer.decode(t)
                        if not unsafe
                        else UNSAFE_ERROR,
                    },
                    "tokens": [self.tokenizer.token_piece(x) for x in t],
                    "logprobs": logprobs_i,
                }
                for t, logprobs_i, unsafe in zip(
                    generation_tokens, generation_logprobs, unsafe_requests
                )
            ]
        return [
            {
                "generation": {
                    "role": "assistant",
                    "destination": "user",
                    "content": self.tokenizer.decode(t) if not unsafe else UNSAFE_ERROR,
                }
            }
            for t, unsafe in zip(generation_tokens, unsafe_requests)
        ]
```

- **辅助函数模块**

辅助函数模块如代码示例8.31所示，它定义了几个辅助函数，用于在对话生成模型中处理文本和生成提示。sample_top_p 函数用于按照 top-p 采样策略从概率分布中采样下一个 token。infilling_prompt_tokens 函数格式化并编码填充式问题的提示，根据需要可以选择前缀-中缀-后缀或后缀-前缀-中缀的格式。dialog_prompt_tokens 函数用于格式化多轮对话的提示，确保将用户和助手的消息交替输入，并在最后添加助手的回复起始标记。这些函数提供了对输入文本进行格式化和编码的工具，以便于在对话生成模型中使用。

代码示例 8.31　辅助函数模块

```
# file: llama_mlu/generation.py
def sample_top_p(probs, p):
    probs_sort, probs_idx = torch.sort(probs, dim=-1, descending=True)
    probs_sum = torch.cumsum(probs_sort, dim=-1)
    mask = probs_sum - probs_sort > p
    probs_sort[mask] = 0.0
    probs_sort.div_(probs_sort.sum(dim=-1, keepdim=True))
    next_token = torch.multinomial(probs_sort, num_samples=1)
    next_token = torch.gather(probs_idx, -1, next_token)
    return next_token


def infilling_prompt_tokens(
    tokenizer: Tokenizer,
    pre: str,
```

```
    suf: str,
    suffix_first: bool = False,
) -> List[int]:
    """
    Format and encode an infilling problem.
    If suffix_first is set, format in suffix-prefix-middle format.
    """
    assert tokenizer.prefix_id is not None
    assert tokenizer.middle_id is not None
    assert tokenizer.suffix_id is not None
    if suffix_first:
        # format as "<PRE> <SUF>{suf} <MID> {pre}"
        return (
            [tokenizer.bos_id, tokenizer.prefix_id, tokenizer.suffix_id]
            + tokenizer.encode_infilling(suf)
            + [tokenizer.middle_id]
            + tokenizer.encode(pre, bos=False, eos=False)
        )
    else:
        # format as "<PRE> {pre} <SUF>{suf} <MID>"
        return (
            [tokenizer.bos_id, tokenizer.prefix_id]
            + tokenizer.encode(pre, bos=False, eos=False)
            + [tokenizer.suffix_id]
            + tokenizer.encode_infilling(suf)
            + [tokenizer.middle_id]
        )


def dialog_prompt_tokens(tokenizer: Tokenizer, dialog: Dialog) -> List[int]:
    """
    Prompt formatting for multi-turn dialogs.
    The dialog is expected to start with a system message and then alternate
    between user and assistant messages.
    """
    assert tokenizer.step_id is not None
    assert all([msg["role"] == "user" for msg in dialog[1::2]]) and all(
        [msg["role"] == "assistant" for msg in dialog[2::2]]
    ), (
        "model only supports 'system', 'user' and 'assistant' roles, "
        "starting with 'system', then 'user' and alternating (u/a/u/a/u...)"
    )
    assert (
        dialog[-1]["role"] == "user"
    ), f"Last message must be from user, got {dialog[-1]['role']}"

    # Format context
    dialog_tokens: List[int] = [tokenizer.bos_id]
    headers: List[str] = []
    for message in dialog:
        headers.clear()
        headers.append(f"Source: {message['role'].strip()}")
        if message.get("destination") is not None:
            headers.append(f"Destination: {message['destination'].strip()}")
        header = " " + "\n".join(headers)
        dialog_tokens += tokenizer.encode(header, bos=False, eos=False)

        if message["content"]:
            body = "\n\n " + message["content"].strip()
            dialog_tokens += tokenizer.encode(body, bos=False, eos=False)

        dialog_tokens += [tokenizer.step_id]
```

```
78
79      # Start of reply
80      headers.clear()
81      headers.append("Source: assistant")
82      headers.append("Destination: user")
83      header = " " + "\n".join(headers)
84      dialog_tokens += tokenizer.encode(header, bos=False, eos=False)
85      dialog_tokens += tokenizer.encode("\n\n ", bos=False, eos=False)
86
87      return dialog_tokens
```

8.3.5.4 Code Llama 代码生成模块

Code Llama 代码生成的代码示例如实8.32所示，它实现了一个使用 LLAMA 模型进行文本生成的命令行工具。用户可以通过命令行参数指定 LLAMA 模型的路径、tokenizer 的路径以及生成文本的相关参数（如温度、top-p 等）。然后，用户提供一系列的提示文本，程序将使用 LLAMA 模型生成这些提示文本的自然延续部分。最后，程序将生成的文本结果打印出来，并计算生成过程的时间。通过 fire 库，用户可以方便地运行这个命令行工具，并获取生成的文本结果。

代码示例 8.32　Code Llama 代码生成

```
1  # file: example_completion_mlu.py
2  from typing import Optional
3
4  import fire
5  import time
6  from llama_mlu import Llama
7
8
9  def main(
10     ckpt_dir: str,
11     tokenizer_path: str,
12     temperature: float = 0.2,
13     top_p: float = 0.9,
14     max_seq_len: int = 256,
15     max_batch_size: int = 4,
16     max_gen_len: Optional[int] = None,
17 ):
18     #TODO: 使用Llama模型构建生成器
19     generator = ____________________________________
20
21     prompts = [
22         # For these prompts, the expected answer is the natural continuation of the prompt
23         """\
24 def fizzbuzz(n: int):""",
25         """\
26 import argparse
27
28 def main(string: str):
29     print(string)
30     print(string[::-1])
31
32 if __name__ == "__main__":"""
33     ]
34     start_time = time.time()
35
36     #TODO: 使用Llama模型生成文本
```

```
    results = ______________________________________
    for prompt, result in zip(prompts, results):
        print(prompt)
        print(f"> {result['generation']}")
        print("\n==================================\n")

    print(f"Infer {time.time() - start_time:.6f} seconds")

if __name__ == "__main__":
    fire.Fire(main)
```

8.3.5.5 Code Llama 代码补全模块

Code Llama 代码补全的代码示例如8.33所示，它使用了 LLAMA 模型进行文本填充任务。用户提供一系列包含填充标记的文本片段，程序会调用 LLAMA 模型来填充这些文本片段中的空白部分。用户可以通过命令行参数指定 LLAMA 模型的路径、tokenizer 的路径以及生成文本的参数（如温度、top-p 等）。然后，程序会对每个文本片段进行文本填充，并将填充后的结果打印出来。整个过程将在命令行中进行，用户只需运行脚本并提供必要的参数即可获取填充后的文本结果。

代码示例 8.33　Code Llama 代码补全

```
# file: example_infilling_mlu.py
import fire
import time
from llama_mlu import Llama


def main(
    ckpt_dir: str,
    tokenizer_path: str,
    temperature: float = 0.0,
    top_p: float = 0.9,
    max_seq_len: int = 192,
    max_gen_len: int = 128,
    max_batch_size: int = 4,
):
    #TODO:使用Llama模型构建生成器
    generator = ____________________________________________

    prompts = [
        '''def remove_non_ascii(s: str) -> str:
    """ <FILL>
    return result
''',
        """# Installation instructions:
    bash<FILL>
This downloads the LLaMA inference code and installs the repository as a local pip package.
""",
        """class InterfaceManagerFactory(AbstractManagerFactory):
    def __init__(<FILL>
def main():
    factory = InterfaceManagerFactory(start=datetime.now())
    managers = []
    for i in range(10):
        managers.append(factory.build(id=i))
""",
        """/-- A quasi-prefunctoid is 1-connected iff all its etalisations are 1-connected. -/
```

```
theorem connected_iff_etalisation [C D : precategoroid] (P : quasi_prefunctoid C D) :
  π₁ P = 0 ↔ <FILL> = 0 :=
begin
  split,
  { intros h f,
    rw pi_1_etalisation at h,
    simp [h],
    refl
  },
  { intro h,
    have := @quasi_adjoint C D P,
    simp [←pi_1_etalisation, this, h],
    refl
  }
end
""",
    ]
    start_time = time.time()
    prefixes = [p.split("<FILL>")[0] for p in prompts]
    suffixes = [p.split("<FILL>")[1] for p in prompts]
    # 使用Llama模型填充文本
    results = _____________________________________________________
    for prompt, result in zip(prompts, results):
        print("\n================ Prompt text ================\n")
        print(prompt)
        print("\n================ Filled text ================\n")
        print(result["full_text"])

    print(f"Infer {time.time() - start_time:.6f} seconds")


if __name__ == "__main__":
    fire.Fire(main)
```

8.3.5.6 Code Llama 指令微调模块

Code Llama 指令微调的代码示例如8.34所示，它实现了一个命令行工具，使用 Llama 来生成对话文本。用户可以通过命令行参数指定模型的路径、tokenizer 的路径以及一些生成对话的参数（如温度、top-p 等）。然后，用户可以定义一系列的对话场景，每个场景包含用户的问题和系统的回答。最后，调用 LLAMA 模型生成对话文本，并将结果打印出来，以用户和系统的角色区分。整个过程将在命令行中进行，用户只需运行脚本并提供必要的参数即可获取生成的对话文本。

代码示例 8.34　Code Llama 指令微调

```
# file: example_instructions_mlu.py
from typing import Optional

import fire

from llama_mlu import Llama


def main(
    ckpt_dir: str,
    tokenizer_path: str,
    temperature: float = 0.2,
    top_p: float = 0.95,
```

```
    max_seq_len: int = 512,
    max_batch_size: int = 8,
    max_gen_len: Optional[int] = None,
):
    #TODO: 使用LLAMA模型构建生成器
    generator = ____________________________________________________

    instructions = [
        [
            {
                "role": "user",
                "content": "In Bash, how do I list all text files in the current directory (excluding subdirectories) that have been modified in the last month?",
            }
        ],
        [
            {
                "role": "user",
                "content": "What is the difference between inorder and preorder traversal? Give an example in Python.",
            }
        ],
        [
            {
                "role": "system",
                "content": "Provide answers in JavaScript",
            },
            {
                "role": "user",
                "content": "Write a function that computes the set of sums of all contiguous sublists of a given list.",
            }
        ],
    ]
    #TODO: 使用LLAMA模型生成对话文本
    results = ____________________________________________

    for instruction, result in zip(instructions, results):
        for msg in instruction:
            print(f"{msg['role'].capitalize()}: {msg['content']}\n")
        print(
            f"> {result['generation']['role'].capitalize()}: {result['generation']['content']}"
        )
        print("\n==================================\n")


if __name__ == "__main__":
    fire.Fire(main)
```

8.3.5.7 实验运行

根据第8.3.5.1~8.3.5.6节的描述补全 Codellama-infer.py、llama_mlu/model.py、llama_mlu/generation.py、example_completion_mlu.py、example_infilling_mlu.py、example_instructions_mlu.py 文件后，在指定目录下执行以下命令，完成基于 Code Llama 的代码生成过程。

1) 环境申请与安装

申请实验环境、安装软件包并登录云平台，本实验的代码存放在云平台/opt/code_chap_8_student 目录下。

```
# 登录云平台
ssh root@xxx.xxx.xxx.xxx -p xxxxx
# 进入/opt/tools/accelerate-0.20-release-mlu目录
cd /opt/tools/accelerate-0.20-release-mlu
#安装accelerate软件包
python setup.py install
# 进入/opt/tools/transformers-mlu-dev目录
cd /opt/tools/transformers-mlu-dev
#安装transformers软件包
python setup.py install
# 进入/opt/code_chap_8_student/codellama目录
cd /opt/code_chap_8_student/codellama
```

2) **代码实现**

补全 Codellama-infer.py、llama_mlu/model.py、llama_mlu/generation.py、example_completion_mlu.py、example_infilling_mlu.py、example_instructions_mlu.py 文件。

```
# 补全Codellama-infer.py文件
vim Codellama-infer.py
# 补全llama_mlu/model.py文件
vim llama_mlu/model.py
# 补全llama_mlu/generation.py文件
vim llama_mlu/generation.py
# 补全example_completion_mlu.py文件
vim example_completion_mlu.py
# 补全example_infilling_mlu.py文件
vim example_infilling_mlu.py
# 补全example_instructions_mlu.py文件
vim example_instructions_mlu.py
```

3) **运行实验**

```
# 执行run-cll.sh文件
bash run-cll.sh
# 执行run_completion.sh文件
bash run_completion.sh
# 执行run_infilling.sh文件
bash run_infilling.sh
# 执行run_instruction.sh文件
bash run_instruction.sh
```

8.3.6 实验评估

该实验的评分标准如下：

- 60 分标准：能够正确实现基于 transformers 库的 Code Llama 推理模块。
- 70 分标准：在 60 分标准基础上，能够正确实现 Transformer 模型构建模块、Llama 代码生成模块。
- 80 分标准：在 70 分标准基础上，能够正确实现 Code Llama 代码生成模块。
- 90 分标准：在 80 分标准基础上，能够正确实现 Code Llama 代码补全模块。
- 100 分标准：在 90 分标准基础上，能够正确实现 Code Llama 指令微调模块。

8.3.7 实验思考

1) 思考为何 Code Llama 34B 为何不支持代码补全功能？

2) Code Llama 后续可以用来做什么？畅想一下，如何结合 Code Llama 做数据分析、自动编程等应用？

3) 目前有很多代码生成的大模型，请将 Code Llama 与其他代码生成的大模型（开源、闭源）进行比较，分析优劣性。