llama3源码解析---generate生成推理部分

定义Llama类，其中核心的函数为build(),generate(),text_completion(),chat_completion()

build()：主要作用是保证模型运行之前参数的正确设置，初始化运行环境（硬件环境），创建模型对象，创建分词器对象，将检查点文件（预训练好的参数）导入模型。

```
@staticmethod
def build(
        ckpt_dir: str,#检查点文件路径
        tokenizer_path: str,#分词器模型的路径
        max_seq_len: int,#最长句子长度
        max_batch_size: int,#最大批量大小
        model_parallel_size: Optional[int] = None,#模型并行数
        seed: int = 1,#种子
    ) -> "Llama":

        ## 确保序列最大长度在1到8192之间
        assert 1 <= max_seq_len <= 8192, f"max_seq_len must be between 1 and
8192, got {max_seq_len}."
        # 确保检查点目录存在
        assert os.path.isdir(ckpt_dir), f"Checkpoint directory '{ckpt_dir}' does
not exist."
        #确保分词器文件存在
        assert os.path.isfile(tokenizer_path), f"Tokenizer file
'{tokenizer_path}' does not exist."
        #如果没有初始化分布式环境，则使用nccl后端初始化
        if not torch.distributed.is_initialized():
            torch.distributed.init_process_group("nccl")
        # 如果模型并行环境未初始化，则进行初始化
        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)
         # 获取当前设备的局部排名
        local_rank = int(os.environ.get("LOCAL_RANK", 0))
        # 设置CUDA设备
        torch.cuda.set_device(local_rank)

        # 在所有进程中设置相同的随机种子
        torch.manual_seed(seed)
        # 如果是分布式训练中的非主进程，则重定向输出到空设备
        if local_rank > 0:
            sys.stdout = open(os.devnull, "w")

        start_time = time.time()
        # 获取检查点文件列表
        checkpoints = sorted(Path(ckpt_dir).glob("*.pth"))
        # 确保检查点文件列表不为空
        assert len(checkpoints) > 0, f"no checkpoint files found in {ckpt_dir}"
        # 确保模型并行大小与检查点文件数量匹配
        assert model_parallel_size == len(
```

```
            checkpoints
        ), f"Loading a checkpoint for MP={len(checkpoints)} but world size is
{model_parallel_size}"
        # 获取当前模型并行排名对应的检查点路径
        ckpt_path = checkpoints[get_model_parallel_rank()]
        # 加载检查点
        checkpoint = torch.load(ckpt_path, map_location="cpu")
        # 加载模型参数
        with open(Path(ckpt_dir) / "params.json", "r") as f:
            params = json.loads(f.read())
        # 创建模型参数对象
        model_args: ModelArgs = ModelArgs(
            max_seq_len=max_seq_len,
            max_batch_size=max_batch_size,
            **params,
        )
        # 创建分词器对象
        tokenizer = Tokenizer(model_path=tokenizer_path)
        # 确保模型参数中的词汇表大小与分词器中的词汇表大小匹配
        assert model_args.vocab_size == tokenizer.n_words
        # 如果CUDA支持BF16，则设置默认张量类型为BFloat16，否则使用Half
        if torch.cuda.is_bf16_supported():
            torch.set_default_tensor_type(torch.cuda.BFloat16Tensor)
        else:
            torch.set_default_tensor_type(torch.cuda.HalfTensor)
        # 创建Transformer模型
        model = Transformer(model_args)
        # 加载模型状态字典，如果 strict=False，则允许状态字典中有额外的键，这些键不会被加载到
模型中
        model.load_state_dict(checkpoint, strict=False)
        # 打印加载时间
        print(f"Loaded in {time.time() - start_time:.2f} seconds")
        # 返回Llama实例
        return Llama(model, tokenizer)
```

generate(): 真正生成token的函数

```
@torch.inference_mode()#通配符：1.让当前函数在inference状态工作不用算梯度，加快速度2.关
droupout
    def generate(
        self,
        prompt_tokens: List[List[int]],#外层列表是batch，内层列表元素是prompt
        max_gen_len: int,#生成文本序列的最大长度
        temperature: float = 0.6,
        top_p: float = 0.9,#用于核采样（nucleus sampling）的概率阈值。默认为0.9
        logprobs: bool = False,#：一个布尔值，指示是否计算token的对数概率（算loss用）。默
认为False
        echo: bool = False,
    ) -> Tuple[List[List[int]], Optional[List[List[float]]]]:
        params = self.model.params
        bsz = len(prompt_tokens)
        assert bsz <= params.max_batch_size, (bsz, params.max_batch_size)
        #计算提示的最小和最大长度
```

```
        min_prompt_len = min(len(t) for t in prompt_tokens)
        max_prompt_len = max(len(t) for t in prompt_tokens)
        #确保最大长度不超过模型允许的最大序列长度
        assert max_prompt_len <= params.max_seq_len
        #计算总长度，要把max_gen_len和max_prompt_len放到一起
        total_len = min(params.max_seq_len, max_gen_len + max_prompt_len)

        #获取填充ID，一般从分词器中获得
        pad_id = self.tokenizer.pad_id
        #实例化全元素都为pad_id的张量，来开辟内存来存储max_gen_len和max_prompt_len
        tokens = torch.full((bsz, total_len), pad_id, dtype=torch.long,
device="cuda")
        #把已知的prompt_tokens填入之前开辟的张量中
        for k, t in enumerate(prompt_tokens):
            tokens[k, : len(t)] = torch.tensor(t, dtype=torch.long,
device="cuda")
        #如果需要计算对数概率，初始化一个新张量
        if logprobs:
            token_logprobs = torch.zeros_like(tokens, dtype=torch.float)

        #初始位置设为0
        prev_pos = 0
        #序列解码结束符，共batch_size个（对应每一个prompt）
        eos_reached = torch.tensor([False] * bsz, device="cuda")
        #定义mask。如果某位置元素是prompt_token,则为ture，如果是pad_id，则为false
        input_text_mask = tokens != pad_id
        #输入的提示（prompt）长度已经达到了模型允许的最大长度，那么就不需要进一步生成新的
token，而是直接计算每个token的对数概率
        if min_prompt_len == total_len:
            logits = self.model.forward(tokens, prev_pos)
            token_logprobs = -F.cross_entropy(
                input=logits.transpose(1, 2),
                target=tokens,
                reduction="none",#不对所有token的损失进行求和或平均
                ignore_index=pad_id,
            )


        #stop_tokens是包含从分词器中获取的所有停止符的张量，用于告诉模型何时应该停止生成文本。
        #它们通常是一些特定的词汇或标记，比如句号（.）、问号（?）或感叹号（!），或者是特定的词
汇，如“结束”、“完毕”等
        stop_tokens = torch.tensor(list(self.tokenizer.stop_tokens)
        """
        生成过程：1.QK计算得到得分矩阵
                  2.得分矩阵经过softmax转换为概率
                  3.用概率和V相乘得到注意力输出
                  4.注意力输出传入FFN
                  5.FFN的输出传入线性层，将其映射到词汇表大小的维度得到logits（所有单词的概率
分布）
                  6.进行采样
        """
        #生成文本的主循环，cur_pos用于跟踪在生成文本时每个批次（batch）中每个序列的当前处理位
置
        for cur_pos in range(min_prompt_len, total_len):
            #算logits，传入从prev_pos到cur_pos这一段tokens（所有batch的），forward中会
生成推理过程中mask
```

```
            logits = self.model.forward(tokens[:, prev_pos:cur_pos], prev_pos)
            #如果温度值大于0，则下一个token的选择使用核采样
            if temperature > 0:
                #根据温度进行概率调节
                probs = torch.softmax(logits[:, -1] / temperature, dim=-1)
                #使用sample_top_p函数进行采样得到next_token
                next_token = sample_top_p(probs, top_p)
            #如果温度小于等于0，就用贪婪采样直接取概率最大的
            else:
                next_token = torch.argmax(logits[:, -1], dim=-1)
            #把next_token变成一维的
            next_token = next_token.reshape(-1)
            #如果cur_pos位置的token是pad_id,那么next_token就用刚生成的新token，否则
next_token保留tokens序列中原来的那个token
            next_token = torch.where(
                input_text_mask[:, cur_pos], tokens[:, cur_pos], next_token
            )
            #然后next_token赋值给cur_pos位置
            tokens[:, cur_pos] = next_token

            #是否计算每个token的对数概率
            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,
                )

            #~input_text_mask[:, cur_pos]：取反后若ture则表示当前位置是pad_id，是要解码
的位置
            #torch.isin(next_token, stop_tokens)：解码出的next_token是停止符
            #然后进行|=或操作。
            #即next_token是停止符且当前位置是待生成token，整体为TURE，跟原本FALSE的
eos_reached或操作之后
            #序列解码结束表示eos_reached变为TURE，且会永远为TURE（ture跟谁或都是ture）。
本序列解码结束。
            #一个batch中所有prompt都结束，整个解码就结束了
            eos_reached |= (~input_text_mask[:, cur_pos]) & (
                torch.isin(next_token, stop_tokens)
            )
            #更新prev_pos到cur_pos
            prev_pos = cur_pos
            if all(eos_reached):
                break
        #将token_logprobs张量转换为Python列表，以便可以返回给用户
        if logprobs:
            token_logprobs = token_logprobs.tolist()
        #初始化输出token列表和输出logprobs列表
        out_tokens, out_logprobs = [], []
        #把tokens变成一个列表然后遍历（索引i，内容toks）
        for i, toks in enumerate(tokens.tolist()):
            # cut to max gen len
            #开始位置从0还是prompt之后，是否为echo模式决定了输出带不带prompt
            start = 0 if echo else len(prompt_tokens[i])
            toks = toks[start : len(prompt_tokens[i]) + max_gen_len]
```

```
            probs = None
            if logprobs:
                probs = token_logprobs[i][start : len(prompt_tokens[i]) +
max_gen_len]

            #遍历所有预定义的结束标记（stop tokens）
            for stop_token in self.tokenizer.stop_tokens:
                try:
                    #尝试在当前生成的token序列（toks）中找到结束标记（stop_token）的索引
                    eos_idx = toks.index(stop_token)
                    #如果找到了结束标记，将生成的token序列截断到结束标记的位置
                    toks = toks[:eos_idx]
                    #如果需要计算对数概率（logprobs为True），也将对数概率列表截断到相同的位
置
                    probs = probs[:eos_idx] if logprobs else None
                except ValueError:#果在当前生成的token序列中没有找到结束标记，则忽略该异
常，继续处理下一个结束标记
                    pass
            #将对应token加入列表
            out_tokens.append(toks)
            out_logprobs.append(probs)
        return (out_tokens, out_logprobs if logprobs else None)
```

text_completion(): 下游任务之文本续写

```
def text_completion(
        self,
        prompts: List[str],#文本提示列表
        temperature: float = 0.6,
        top_p: float = 0.9,  # 核采样的概率阈值
        max_gen_len: Optional[int] = None,  # 生成序列的最大长度
        logprobs: bool = False,  # 是否计算token的log概率
        echo: bool = False,  # 是否在生成的输出中包含提示token
    ) -> List[CompletionPrediction]:
        """
        Perform text completion for a list of prompts using the language
generation model.
    使用语言生成模型为一系列提示完成文本。
    Returns:
        List[CompletionPrediction]: List of completion predictions, each
containing the generated text completion.
        返回包含生成文本完成的完成预测列表。
        """
        # 如果未提供max_gen_len，则设置为模型最大序列长度减1
        if max_gen_len is None:
            max_gen_len = self.model.params.max_seq_len - 1
        # 将提示文本编码为token,它将字符串 x 编码成token序列,bos=True:指定编码的序列以开始
标记开始,eos=False:指定编码的序列不以结束标记结束
        prompt_tokens = [self.tokenizer.encode(x, bos=True, eos=False) for x in
prompts]
        # 生成文本,调用前面的generate函数，generation_tokens生成的token内容，
generation_logprobs，生成token的log概率
        generation_tokens, generation_logprobs = self.generate(
```

```
            prompt_tokens=prompt_tokens,
            max_gen_len=max_gen_len,
            temperature=temperature,
            top_p=top_p,
            logprobs=logprobs,
            echo=echo,
        )
        # 如果需要计算log概率，则返回包含生成文本、token和log概率的结果
        if logprobs:
            return [
            {   #tokenizer负责将文本分割成模型可以理解的小块（即token），而解码器则执行相反的操作，将这些token重新组合成可读的文本
                "generation": self.tokenizer.decode(t),  # 解码生成的文本，t是一个列表
                "tokens": [self.tokenizer.decode([x]) for x in t],  # 解码每个token
                "logprobs": logprobs_i,  # log概率
            }
            #zip将两个列表的对应的一对元素打包成一个元组，返回一个新列表，元素是元组，此处就是token和其log概率一一对应
            for t, logprobs_i in zip(generation_tokens, generation_logprobs)
        ]
            # 如果不需要计算log概率，则只返回包含生成文本的结果
        return [{"generation": self.tokenizer.decode(t)} for t in generation_tokens]
```

chat_completion():下游任务之对话功能

```
def chat_completion(
        self,
        dialogs: List[Dialog],  # 对话列表
        temperature: float = 0.6,  # 控制采样随机性的temperature值，默认为0.6
        top_p: float = 0.9,  # 核采样的概率阈值，默认为0.9
        max_gen_len: Optional[int] = None,  # 生成序列的最大长度，默认为模型最大序列长度减1
        logprobs: bool = False,  # 是否计算token的log概率，默认为False
    ) -> List[ChatPrediction]:
        """
        Generate assistant responses for a list of conversational dialogs using the language generation model.
    使用语言生成模型为一系列对话生成助手的响应。
    Returns:
        List[ChatPrediction]: List of chat predictions, each containing the assistant's generated response.
        返回包含助手生成响应的聊天预测列表
        """
         # 如果未提供max_gen_len，则设置为模型最大序列长度减1
        if max_gen_len is None:
            max_gen_len = self.model.params.max_seq_len - 1

        # 将对话编码为token
        prompt_tokens = [
            self.formatter.encode_dialog_prompt(dialog) for dialog in dialogs
```

```
        ]
        # 生成文本
        generation_tokens, generation_logprobs = self.generate(
            prompt_tokens=prompt_tokens,
            max_gen_len=max_gen_len,
            temperature=temperature,
            top_p=top_p,
            logprobs=logprobs,
        )
        # 如果需要计算log概率，则返回包含生成文本、token和log概率的结果
        if logprobs:
            return [
                {
                    "generation": {
                    "role": "assistant",  # 角色为助手，此处硬编码为assistant，表示是模型的回复，若是用户文本应为“user”
                    "content": self.tokenizer.decode(t),  # 解码生成的文本
                },
                    "tokens": [self.tokenizer.decode([x]) for x in t],  # 解码每个token
                    "logprobs": logprobs_i,  # log概率
                }
            for t, logprobs_i in zip(generation_tokens, generation_logprobs)
            ]
        # 如果不需要计算log概率，则只返回包含生成文本的结果
        return [
            {
            "generation": {
                "role": "assistant",  # 角色为助手
                "content": self.tokenizer.decode(t),  # 解码生成的文本
            },
        }
        for t in generation_tokens
    ]
```