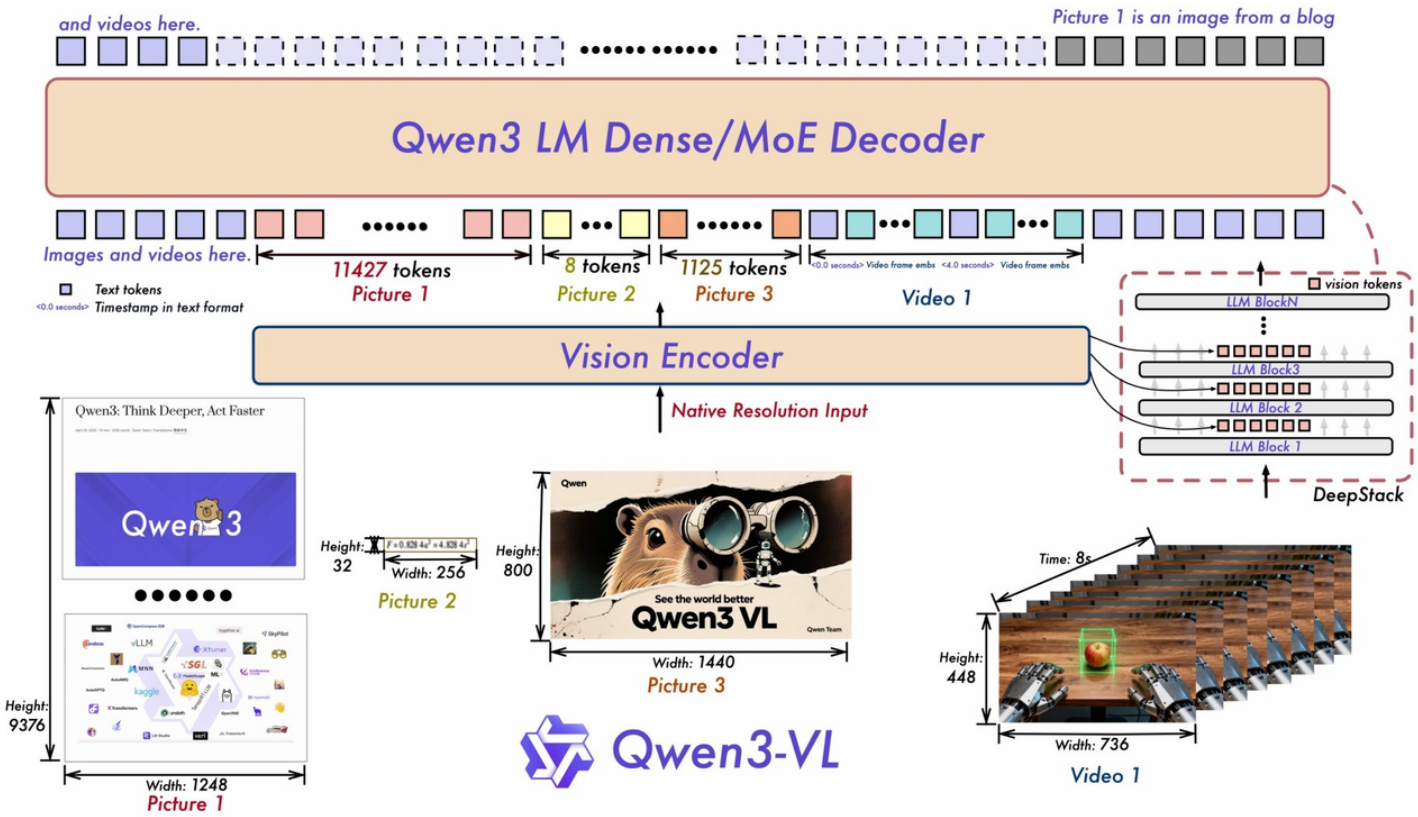
Qwen3VL核心技术解析

Qwen3VL在架构方面，做了以下改进

1. **交错-MRoPE**，原始MRoPE将特征维度按照时间（T）、高度（H)和宽度（W)的顺序分块划分，使得时间信息全部分布在高频维度上。Qwen3-VL将时间、高度、宽度三个维度均匀分布在低频和高频带中，显著提升图像与视频中的时空建模能力；
2. **DeepStack**，ViT不同层的视觉token通过残差连接路由至对应的 LLM 层，能够有效保留从底层（low-level）到高层（high-level）的丰富视觉信息，在不增加额外上下文长度的情况下增强多层级融合，强化视觉-语言对齐；
3. 采用**基于文本的时间对齐机制**，通过显式的文本时间戳对齐替代 Qwen2.5-VL 中通过位置编码实现的绝对时间对齐，采用“时间戳-视频帧”交错的输入形式，实现更精确的时空定位。为平衡纯文本与多模态学习目标，采用平方根重加权策略，在不损害文本能力的前提下显著提升多模态性能。

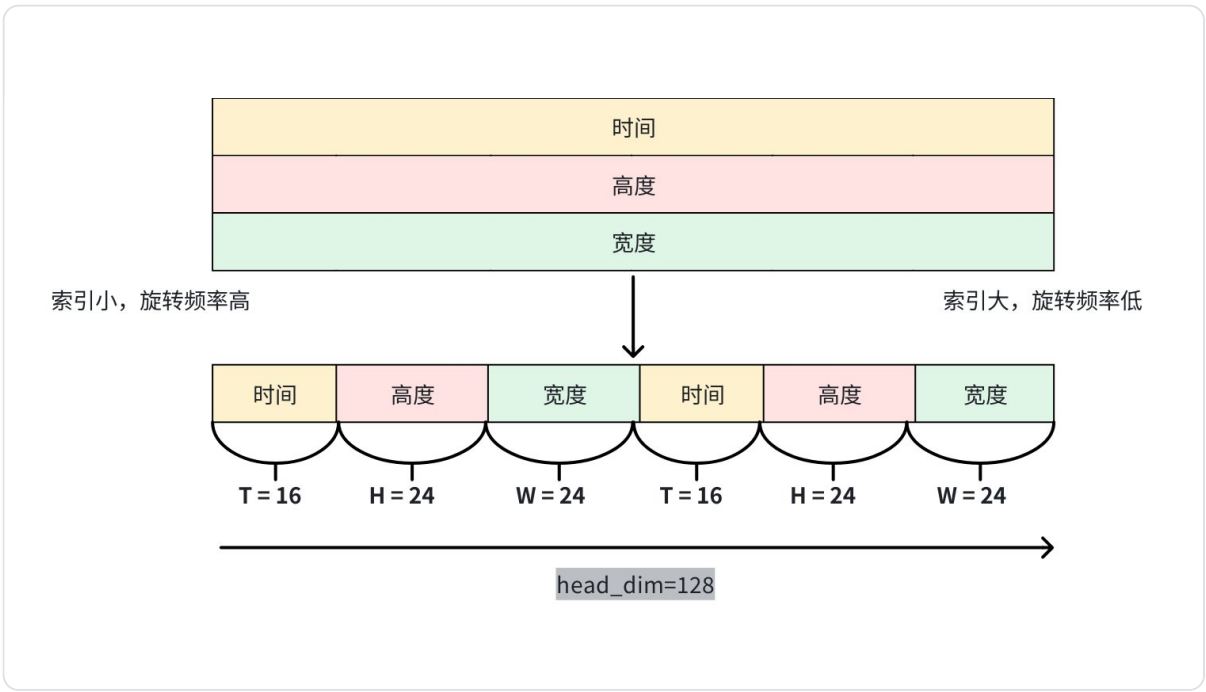
这篇文章，我们就从源码的角度解析这三个技术是怎么实现的



> Qwen3-VL采用**ViT+Merger+LLM**架构，能够处理文本、图像和视频在内的多模态输入。其中ViT部分支持原生分辨率，并且通过Deepstack机制将ViT中多层的视觉特征注入到LLM中，提升对图像的理解。此外Qwen3-VL采用交错MRoPE以实现均衡频谱的多模态输入，并使用基于文本的时间戳标记，更有效地捕捉视频序列的时间结构。

交错-MRoPE

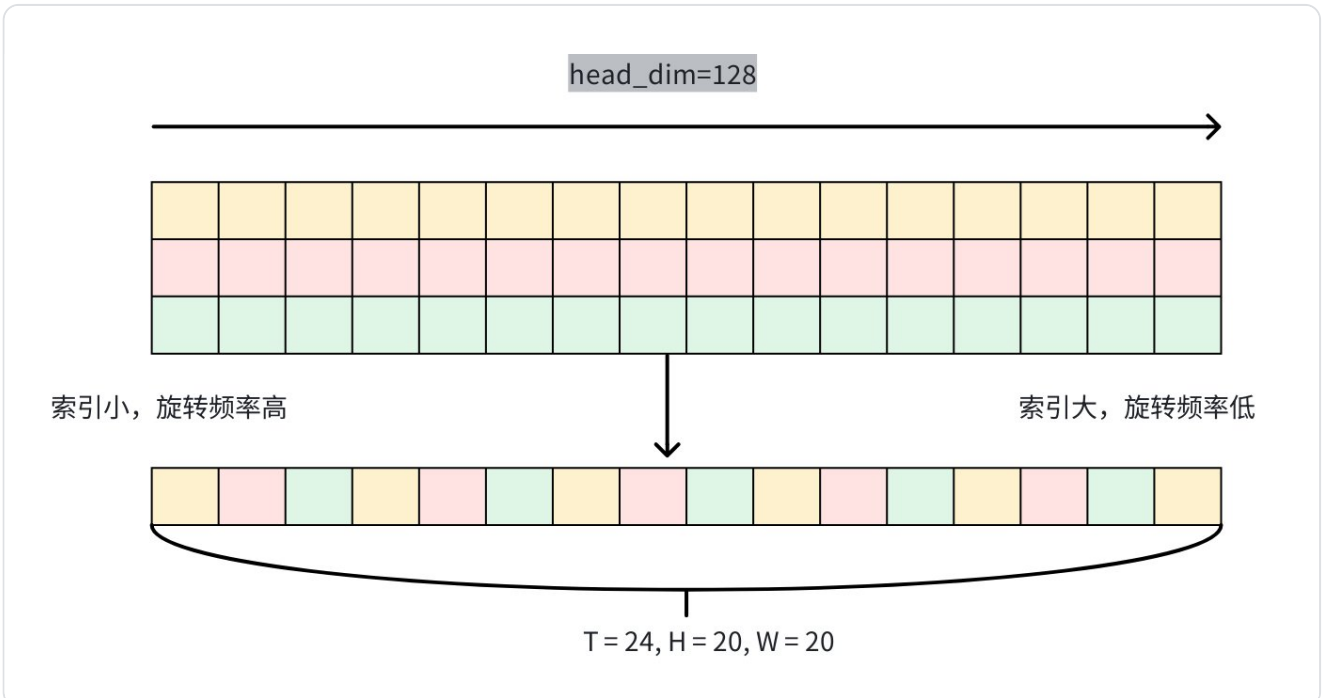
回忆一下Qwen2.5VL 中的MRoPE，使用3D位置信息（**时间，高度，宽度**）。其位置向量的组成方式为：



> 一个token的sin/cos向量

但这种方式存在问题，即RoPE中 $\theta_i = 10000^{\frac{-2i}{d}}$ ，i 表示索引，由于旋转频率随着索引增加而降低，MRoPE会导致时间维度的信息全部在高频维度上，不利于长序列的理解，会导致注意力随着时间快速衰减。

为此，Qwen3-VL在LLM中采用Interleaved MRoPE，以细粒度的轮询方式将特征通道分配到**时间，高度，宽度**轴上，确保每个位置轴都使用从高到低的完整频谱进行编码。



> 上图中黄、粉、绿分别表示T、H、W维度，T=24，H和W=20，1:4缩小，所以最后会有一个单独的时间块。

接下来结合Qwen3-VLTextRotaryEmbedding 源码理解交错MRoPE的实现。类的定义跟Qwen2.5VL类似，定义了频率 $\theta_i = 10000^{\frac{-2i}{d}}$ ，交错MRoPE的分段 $[24, 20, 20]$

代码块

```
class Qwen3-VLTextRotaryEmbedding(nn.Module):
    inv_freq: torch.Tensor

    def __init__(self, config: Qwen3-VLTextConfig, device=None):
        super().__init__()
        self.max_seq_len_cached = config.max_position_embeddings
        self.original_max_seq_len = config.max_position_embeddings
        self.config = config
        self.rope_type = self.config.rope_parameters["rope_type"]

        # 默认使用标准的 RoPE 参数计算函数
        rope_init_fn: Callable = self.compute_default_rope_parameters
        if self.rope_type != "default":
            rope_init_fn = ROPE_INIT_FUNCTIONS[self.rope_type]
        # 计算频率和注意力缩放因子
        inv_freq, self.attention_scaling = rope_init_fn(self.config, device)

        self.register_buffer("inv_freq", inv_freq, persistent=False)
        self.original_inv_freq = inv_freq

        # MROPE 的分段配置，默认为 [24, 20, 20]，分别对应 T（时间/序列）、H（高度）、W（宽度）三个维度的分段数
        self.mrope_section = config.rope_parameters.get("mrope_section", [24, 20, 20])

    @staticmethod
    def compute_default_rope_parameters(
        config: Optional[Qwen3-VLTextConfig] = None,
        device: Optional["torch.device"] = None,
        seq_len: Optional[int] = None,
    ) -> tuple["torch.Tensor", float]:
        base = config.rope_parameters["rope_theta"]
        dim = getattr(config, "head_dim", None) or config.hidden_size // config.num_attention_heads
        attention_factor = 1.0
        # theta_i = 1 / (10000^{2i/d})
        inv_freq = 1.0 / (
            base ** (torch.arange(0, dim, 2, dtype=torch.int64).to(device=device, dtype=torch.float) / dim)
```

```
        )
        return inv_freq, attention_factor
```

接下来关注 `forward` 函数

- 首先将 `inv_freq` 扩展到 `(3, batch_size, head_dim//2, 1)`，将 `position_ids` 拓展为 `(3, batch_size, 1, seq_len)`，两者相乘得到频率矩阵 `freqs`
- `freqs` 布局为 $[TTT...HHH...WWW]$，时间信息全部分布在高频维度上，不利于长序列的理解。这就需要用到交错MRoPE，将其重组为 $[THWTHWTHW...TT]$。
- 最后将位置编码拼接，与 `attention_scaling` 相乘计算出 `cos` 和 `sin` 向量。

代码块

```python
@torch.no_grad()
@dynamic_rope_update
def forward(self, x, position_ids):
    # 如果 position_ids 是 2D 的（batch_size, seq_len），则扩展为 3D（3, batch_size, seq_len）
    if position_ids.ndim == 2:
        position_ids = position_ids[None, ...].expand(3, position_ids.shape[0], -1)
    #从 inv_freq (head_dim//2,) 扩展到 (3, batch_size, head_dim//2, 1)
    inv_freq_expanded = self.inv_freq[None, None, :, None].float().expand(3, position_ids.shape[1], -1, 1)

    # 扩展position_ids以匹配inv_freq_expanded的形状：从 (3, batch_size, seq_len) 扩展到 (3, batch_size, 1, seq_len)
    position_ids_expanded = position_ids[:, :, None, :].float()

    device_type = x.device.type if isinstance(x.device.type, str) and x.device.type != "mps" else "cpu"

    with torch.autocast(device_type=device_type, enabled=False):
        # 计算频率：inv_freq_expanded @ position_ids_expanded 进行矩阵乘法
        # 结果形状为 (3, batch_size, head_dim//2, seq_len)，然后转置为 (3, batch_size, seq_len, head_dim//2)
        freqs = (inv_freq_expanded.float() @ position_ids_expanded.float()).transpose(2, 3)
        # 应用交错 MROPE：将分块的频率布局 [TTT...HHH...WWW] 重组为交错布局 [THWTHWTHW...TT]
        freqs = self.apply_interleaved_mrope(freqs, self.mrope_section)
        # 这是因为旋转位置编码需要成对的维度（实部和虚部），(3, batch_size, token_number, head_dim)
        emb = torch.cat((freqs, freqs), dim=-1)
```

```
        # emb.cos() 计算每个角度的余弦值，* self.attention_scaling 应用缩放因子（通常为1.0）
        cos = emb.cos() * self.attention_scaling
        sin = emb.sin() * self.attention_scaling

    return cos.to(dtype=x.dtype), sin.to(dtype=x.dtype)
```

关注交错位置编码的具体实现，以 T 维度为基础，遍历 H 和 W 维度。

- offset=1 对应 H 维度在交错序列中的位置，idx 为 $[1, 4, 7, 10, ...]$,替换掉T维度中对应位置的值
- offset=2 对应 W 维度在交错序列中的位置，idx 为 $[2, 5, 8, 11, ...]$,替换掉T维度中对应位置的值
- 超过length的低频维度还是采用T维度的值。

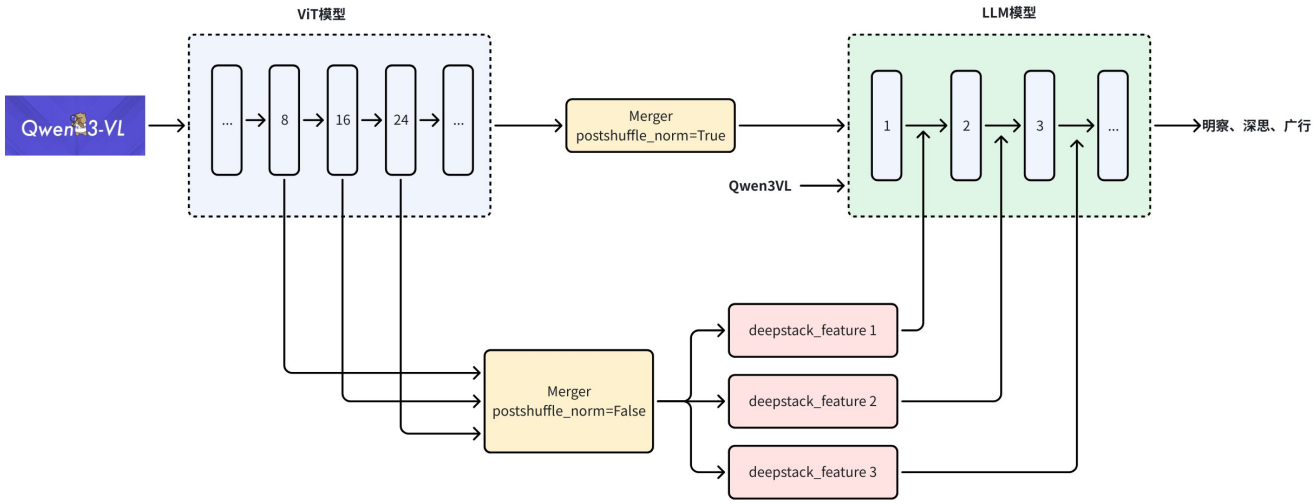
代码块

```
def apply_interleaved_mrope(self, freqs, mrope_section):
    # 以 T 维度为基础，形状为 (batch_size, seq_len, head_dim // 2)
    freqs_t = freqs[0]

    # 遍历 H 和 W 维度（索引从 1 开始，对应 dim=1 和 dim=2）
    for dim, offset in enumerate((1, 2), start=1):
        # 计算当前维度在交错序列中的总长度
        # mrope_section[dim] 是该维度的分段数，乘以 3 是因为交错模式是 THW 三个一组
        length = mrope_section[dim] * 3
        # 例如 offset=1 时，idx 为 [1, 4, 7, 10, ...]，对应 H 维度在交错序列中的位置
        # 例如 offset=2 时，idx 为 [2, 5, 8, 11, ...]，对应 W 维度在交错序列中的位置
        idx = slice(offset, length, 3)
        # 将对应维度的频率值复制到 freqs_t 的相应位置，实现交错排列
        freqs_t[..., idx] = freqs[dim, ..., idx]
    return freqs_t
```

DeepStack

从 ViT的中间层提取视觉标记，注入到LLM的多个层中，保留了从低级到高级表示的丰富视觉信息。从视觉编码器的三个 $[8, 16, 24]$ 不同层级选择特征，使用Merger将这些特征投影为视觉token，然后添加到前三个LLM层的对应hidden states中。



接下来结合Qwen3-VLVisionModel源码理解DeepStack的实现。首先关注类的定义，

- `pos_embed` 表示绝对位置编码，目的是为了适应动态分辨率，支持任意分辨率图像输入，将输入图像的坐标映射到 48 * 48的网格上，得到浮点数坐标，再计算双线性插值的位置编码。
- `merger`：接受ViT输出的特征，将 2×2 视觉特征压缩为1个token
- `deepstack_merger_list`：取出ViT的第 $[8, 16, 24]$ 层输出的hidden_state，经过各自的Merger后作为deepstack 特征，然后与前三个LLM层的对应hidden states相加。

代码块

```
class Qwen3-VLVisionModel(Qwen3-VLPreTrainedModel):
    config: Qwen3-VLVisionConfig
    _no_split_modules = ["Qwen3-VLVisionBlock"]

    def __init__(self, config, *inputs, **kwargs) -> None:
        super().__init__(config, *inputs, **kwargs)
        # spatial_merge_size = 2
        self.spatial_merge_size = config.spatial_merge_size
        # patch_size = 16
        self.patch_size = config.patch_size
        self.spatial_merge_unit = self.spatial_merge_size * self.spatial_merge_size

        # 将图像块转换为嵌入向量Conv3d(3, 1152, kernel_size=(2, 16, 16), stride=(2, 16, 16), bias=True)。
        self.patch_embed = Qwen3-VLVisionPatchEmbed(
            config=config,
        )
        # 可学习的绝对位置编码，(2304, 1152)
        self.pos_embed = nn.Embedding(config.num_position_embeddings, config.hidden_size)
        # 为适应动态分辨率，支持任意分辨率图像输入，根据输入尺寸插值绝对位置嵌入
```

```
        # 将输入图像的坐标映射到 48 * 48的网格上，得到浮点数坐标，再计算双线性插值的位置编码。
        self.num_grid_per_side = int(config.num_position_embeddings**0.5)

        # head_dim = 72
        head_dim = config.hidden_size // config.num_heads
        # 旋转位置嵌入角度
        self.rotary_pos_emb = Qwen3-VLVisionRotaryEmbedding(head_dim // 2)
        self.blocks = nn.ModuleList([Qwen3-VLVisionBlock(config) for _ in range(config.depth)])
        # 注意merger这里use_postshuffle_norm = False
        self.merger = Qwen3-VLVisionPatchMerger(
            config=config,
            use_postshuffle_norm=False,
        )

        # 指定哪些层需要进行deepstack处理 [8, 16, 24]
        self.deepstack_visual_indexes = config.deepstack_visual_indexes
        # 注意merger这里use_postshuffle_norm = False
        self.deepstack_merger_list = nn.ModuleList(
            [
                Qwen3-VLVisionPatchMerger(
                    config=config,
                    use_postshuffle_norm=True,
                )
                for _ in range(len(config.deepstack_visual_indexes))
            ]
        )
        self.gradient_checkpointing = False
```

Qwen3-VLVisionPatchMerger的实现就是一个两层的MLP层，`merger`与`deepstack_merger_list`区别在于是先归一化还是先合并。

- `use_postshuffle_norm = True`：在合并后的特征空间中进行归一化，可以更好地处理合并后的特征分布
- `use_postshuffle_norm = False`：先对每个原始特征进行归一化，然后再合并，保持原始特征的统计特性

代码块

```
class Qwen3-VLVisionPatchMerger(nn.Module):
    def __init__(self, config: Qwen3-VLVisionConfig, use_postshuffle_norm=False) -> None:
        super().__init__()
        # 计算合并后的隐藏层维度：原始维度（1152） × 空间合并尺寸的平方（4）= 4608
        self.hidden_size = config.hidden_size * (config.spatial_merge_size**2)
```

```
        self.use_postshuffle_norm = use_postshuffle_norm
        self.norm = nn.LayerNorm(self.hidden_size if use_postshuffle_norm else config.hidden_size, eps=1e-6)
        self.linear_fc1 = nn.Linear(self.hidden_size, self.hidden_size)
        self.act_fn = nn.GELU()
        self.linear_fc2 = nn.Linear(self.hidden_size, config.out_hidden_size)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # use_postshuffle_norm=True (后置归一化):
        #   1. 先将x reshape到(-1, 4608) - 即先进行空间合并，进行2x2视觉特征堆叠
        #   2. 然后在合并后的特征上进行归一化
        #   3. 最后reshape到(-1, 4608)
        #
        # use_postshuffle_norm=False (前置归一化):
        #   1. 直接在原始x (1152) 上进行归一化
        #   2. 然后reshape到(-1, 4608)
        x = self.norm(x.view(-1, self.hidden_size) if self.use_postshuffle_norm else x).view(-1, self.hidden_size)
        x = self.linear_fc2(self.act_fn(self.linear_fc1(x)))
        return x
```

在Qwen3-VLVisionModel的前向传播过程中，首先在 `hidden_states` 上添加绝对位置编码，然后计算出注意力中的旋转位置编码，对于ViT的第 $[8, 16, 24]$ 层计算deepstack特征。返回merger特征和deepstack特征。

代码块

```
def forward(self, hidden_states: torch.Tensor, grid_thw: torch.Tensor, **kwargs) -> torch.Tensor:
    # 通过图像分割成patch，并投影为为嵌入向量。
    hidden_states = self.patch_embed(hidden_states)
    # 通过双线性插值获取绝对位置编码，支持任意分辨率的输入
    pos_embeds = self.fast_pos_embed_interpolate(grid_thw)
    # 将绝对位置编码添加到hidden_states中
    hidden_states = hidden_states + pos_embeds
    # 计算注意力中的旋转位置编码（RoPE）
    rotary_pos_emb = self.rot_pos_emb(grid_thw)

    seq_len, _ = hidden_states.size()
    hidden_states = hidden_states.reshape(seq_len, -1)
    # 将旋转位置编码重塑为(seq_len, head_dim//2)形状
    rotary_pos_emb = rotary_pos_emb.reshape(seq_len, -1)
    # 旋转位置编码需要成对的维度（实部和虚部），得到完整的旋转嵌入维度
    emb = torch.cat((rotary_pos_emb, rotary_pos_emb), dim=-1)
    position_embeddings = (emb.cos(), emb.sin())

```

```
    cu_seqlens = torch.repeat_interleave(grid_thw[:, 1] * grid_thw[:, 2], grid_thw[:, 0]).cumsum(
        dim=0,
        #  Select dtype based on the following factors:
        #  - FA2 requires that cu_seqlens_q must have dtype int32
        #  - torch.onnx.export requires that cu_seqlens_q must have same dtype as grid_thw
        # See https://github.com/huggingface/transformers/pull/34852 for more information
        dtype=grid_thw.dtype if torch.jit.is_tracing() else torch.int32,
    )
    cu_seqlens = F.pad(cu_seqlens, (1, 0), value=0)

    # deepstack特征列表
    deepstack_feature_lists = []
    for layer_num, blk in enumerate(self.blocks):
        hidden_states = blk(
            hidden_states,
            cu_seqlens=cu_seqlens,
            position_embeddings=position_embeddings,
            **kwargs,
        )
        # [8, 16, 24]
        if layer_num in self.deepstack_visual_indexes:
            deepstack_feature = self.deepstack_merger_list[self.deepstack_visual_indexes.index(layer_num)](
                hidden_states
            )
            deepstack_feature_lists.append(deepstack_feature)

    # 使用merger处理最终的hidden_states
    hidden_states = self.merger(hidden_states)
    # hidden_states: 最终处理后的视觉特征，将输入到LLM
    # deepstack_feature_lists: 特定层的deepstack特征，将在LLM的前3层中使用
    return hidden_states, deepstack_feature_lists
```

然后在Qwen3-VLTextModel的前向传播中，在LLM前3层中添加deepstack视觉特征到 `hidden_states` 中。

代码块

```
for layer_idx, decoder_layer in enumerate(self.layers):
    layer_outputs = decoder_layer(
        hidden_states,
        attention_mask=attention_mask,
        position_ids=text_position_ids,
```

```
            past_key_values=past_key_values,
            cache_position=cache_position,
            position_embeddings=position_embeddings,
            **kwargs,
        )
        hidden_states = layer_outputs

        # 在LLM前3层中添加deepstack视觉特征到hidden_states中，len(deepstack_visual_embeds) = 3
        if deepstack_visual_embeds is not None and layer_idx in range(len(deepstack_visual_embeds)):
            hidden_states = self._deepstack_process(
                hidden_states,
                visual_pos_masks,
                deepstack_visual_embeds[layer_idx],
            )
    hidden_states = self.norm(hidden_states)
    return BaseModelOutputWithPast(
        last_hidden_state=hidden_states,
        past_key_values=past_key_values,
    )
```

LLM的 `hidden_states` 与deepstack特征的融合方式如下，由于Merger特征和deepstack 特征维度一致，直接将hidden_states中视觉token的位置与deepstack的视觉特征相加。

代码块

```
def _deepstack_process(
    self, hidden_states: torch.Tensor, visual_pos_masks: torch.Tensor, visual_embeds: torch.Tensor
):
    visual_pos_masks = visual_pos_masks.to(hidden_states.device)
    visual_embeds = visual_embeds.to(hidden_states.device, hidden_states.dtype)
    hidden_states = hidden_states.clone()
    # visual_pos_masks是一个布尔掩码，只有视觉token位置为True，其他位置为False
    # 将hidden_states中视觉token的位置与deepstack的视觉特征相加
    # 相当于在ViT模型和LLM前三层的hidden_states之间添加了残差连接
    local_this = hidden_states[visual_pos_masks, :] + visual_embeds
    hidden_states[visual_pos_masks, :] = local_this
    return hidden_states
```

基于文本的时间对齐机制

Qwen2.5VL将时间位置 ID 直接关联到绝对时间（即3DRoPE，时间维度的值对应帧数），该方法在处理长视频时会产生过大且稀疏的时间位置 ID，削弱模型对长时序上下文的理解能力。 并且为了有效学

习，需要在不同帧率（fps）下进行广泛且均匀的采样，显著增加了训练数据构建的成本。

Qwen3-VL采用**基于文本的时间对齐机制**，为每个视频时序patch都添加时间戳前缀，在训练过程中添加了“秒”和“时:分:秒”两种格式的时间戳以确保模型能够学习理解多种时间码表示。这种方法会带来适度的上下文长度增加。

代码实现与Qwen2.5VL中的get_rope_index中基本一致，区别只在于每个帧都被视为独立的图像，时间维度都设置为1。

代码块

```
if video_grid_thw is not None:
    # 根据时间维度（第0列）重复每个视频的grid_thw，将多帧视频拆分为单帧
    video_grid_thw = torch.repeat_interleave(video_grid_thw, video_grid_thw[:, 0], dim=0)
    # 将时间维度设置为1（因为每帧单独处理，时间维度为1）
    video_grid_thw[:, 0] = 1
```

在数据预处理时就已经在文本中添加了时间戳，输入是 `聪明的<t1> <vision_start> <video_token> [视觉特征token序列] <vision_end> 小羊` 。其中 `<t1>` 表示时间戳， `[视觉特征token序列]` 包含1个帧，每一帧是2×2 网格 `（llm_grid_h=2， llm_grid_w=2）` 。

Token	类型	位置ID （T， H， W）	解释
聪明	文本	(0, 0, 0)	文本token，THW三维相同
的	文本	(1, 1, 1)	文本token，THW三维相同
<t1>	文本	(2, 2, 2)	时间戳被视为文本token，TH
<vision_start>	文本	(3, 3, 3)	视觉开始标记token，THW三
<video_token>	文本	(4, 4, 4)	视频token标记，THW三维相
(f1_0,0)	视觉	(0, 0, 0) + 5 = (5, 5, 5)	视觉特征token，t=0，h=0, w
(f1_0,1)	视觉	(0, 0, 1) + 5 = (5, 5, 6)	视觉特征token，t=0，h=0, w
(f1_1,0)	视觉	(0, 1, 0) + 5 = (5, 6, 5)	视觉特征token，t=0，h=1, w
(f1_1,1)	视觉	(0, 1, 1) + 5 = (5, 6, 6)	视觉特征token，t=0，h=1, w
<vision_end>	文本	(9, 9, 9)	视觉结束标记token，THW三
小	文本	(10, 10, 10)	文本token，THW三维相同
羊	文本	(11, 11, 11)	文本token，THW三维相同