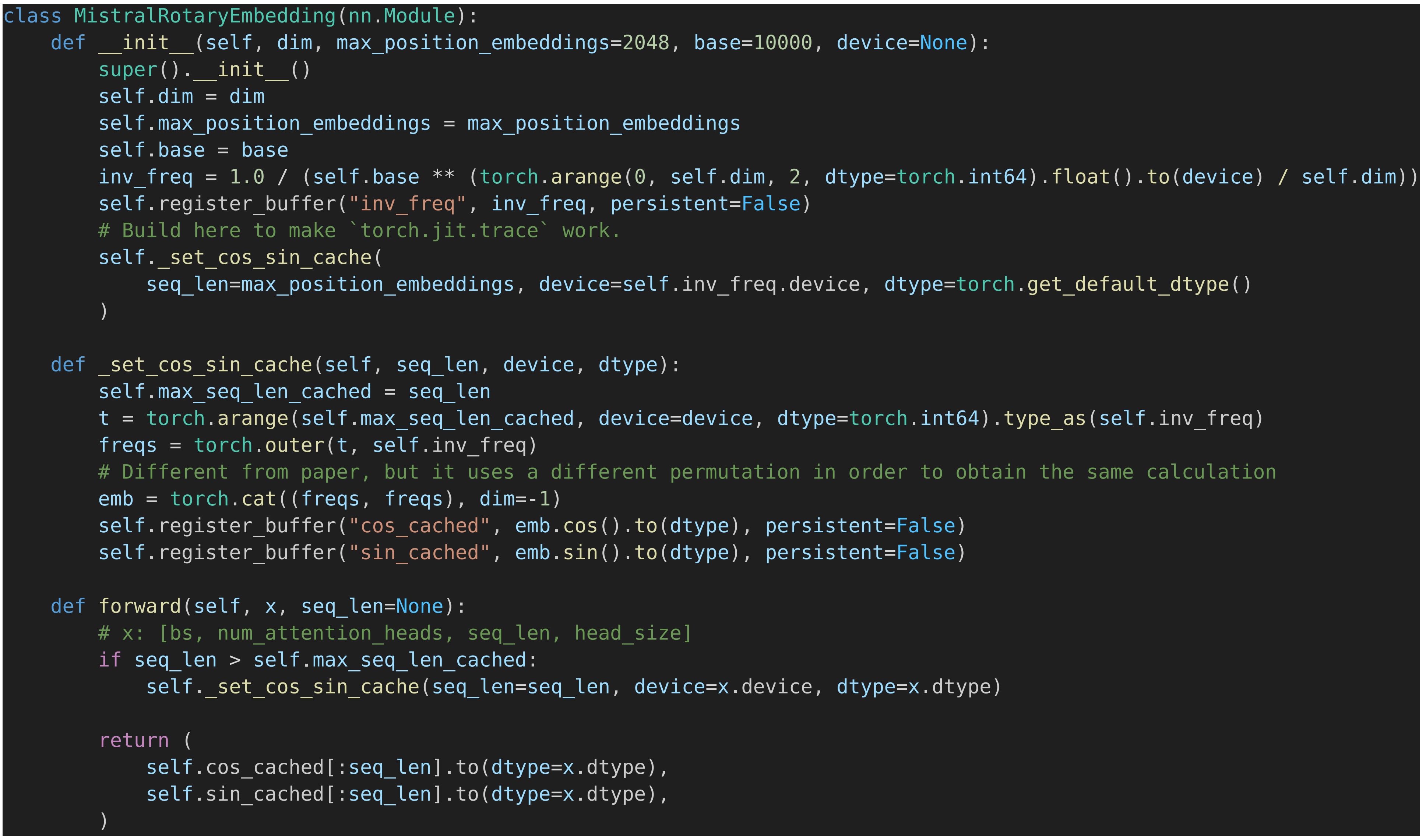
Mistral

Llama

1. RotaryEmbedding



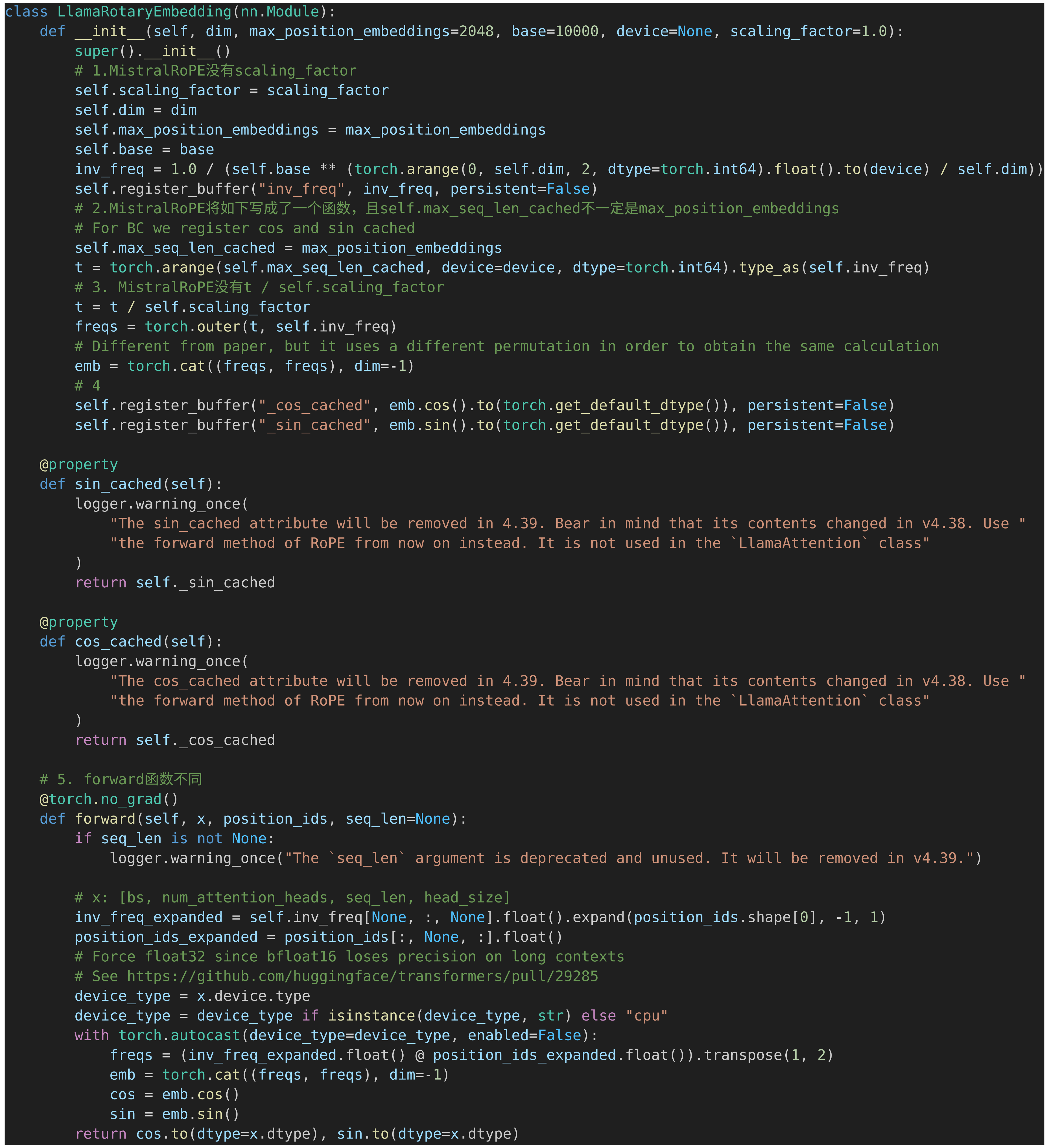
```python
class MistralRotaryEmbedding(nn.Module):
    def __init__(self, dim, max_position_embeddings=2048, base=10000, device=None):
        super().__init__()
        self.dim = dim
        self.max_position_embeddings = max_position_embeddings
        self.base = base
        inv_freq = 1.0 / (self.base ** (torch.arange(0, self.dim, 2, dtype=torch.int64).float().to(device) / self.dim))
        self.register_buffer("inv_freq", inv_freq, persistent=False)
        # Build here to make `torch.jit.trace` work.
        self._set_cos_sin_cache(
            seq_len=max_position_embeddings, device=self.inv_freq.device, dtype=torch.get_default_dtype()
        )

    def _set_cos_sin_cache(self, seq_len, device, dtype):
        self.max_seq_len_cached = seq_len
        t = torch.arange(self.max_seq_len_cached, device=device, dtype=torch.int64).type_as(self.inv_freq)
        freqs = torch.outer(t, self.inv_freq)
        # Different from paper, but it uses a different permutation in order to obtain the same calculation
        emb = torch.cat((freqs, freqs), dim=-1)
        self.register_buffer("cos_cached", emb.cos().to(dtype), persistent=False)
        self.register_buffer("sin_cached", emb.sin().to(dtype), persistent=False)

    def forward(self, x, seq_len=None):
        # x: [bs, num_attention_heads, seq_len, head_size]
        if seq_len > self.max_seq_len_cached:
            self._set_cos_sin_cache(seq_len=seq_len, device=x.device, dtype=x.dtype)

        return (
            self.cos_cached[:seq_len].to(dtype=x.dtype),
            self.sin_cached[:seq_len].to(dtype=x.dtype),
        )
```

```
# copied from transformers.models.llama.modeling_llama.LlamaRotaryEmbedding with Llama->Mistral
# TODO @Arthur no longer copied from LLama after static cache
```



```python
class LlamaRotaryEmbedding(nn.Module):
    def __init__(self, dim, max_position_embeddings=2048, base=10000, device=None, scaling_factor=1.0):
        super().__init__()
        # 1.MistralRoPE没有scaling_factor
        self.scaling_factor = scaling_factor
        self.dim = dim
        self.max_position_embeddings = max_position_embeddings
        self.base = base
        inv_freq = 1.0 / (self.base ** (torch.arange(0, self.dim, 2, dtype=torch.int64).float().to(device) / self.dim))
        self.register_buffer("inv_freq", inv_freq, persistent=False)
        # 2.MistralRoPE将如下写成了一个函数，且self.max_seq_len_cached不一定是max_position_embeddings
        # For BC we register cos and sin cached
        self.max_seq_len_cached = max_position_embeddings
        t = torch.arange(self.max_seq_len_cached, device=device, dtype=torch.int64).type_as(self.inv_freq)
        # 3. MistralRoPE没有t / self.scaling_factor
        t = t / self.scaling_factor
        freqs = torch.outer(t, self.inv_freq)
        # Different from paper, but it uses a different permutation in order to obtain the same calculation
        emb = torch.cat((freqs, freqs), dim=-1)
        # 4
        self.register_buffer("_cos_cached", emb.cos().to(torch.get_default_dtype()), persistent=False)
        self.register_buffer("_sin_cached", emb.sin().to(torch.get_default_dtype()), persistent=False)

    @property
    def sin_cached(self):
        logger.warning_once(
            "The sin_cached attribute will be removed in 4.39. Bear in mind that its contents changed in v4.38. Use "
            "the forward method of RoPE from now on instead. It is not used in the `LlamaAttention` class"
        )
        return self._sin_cached

    @property
    def cos_cached(self):
        logger.warning_once(
            "The cos_cached attribute will be removed in 4.39. Bear in mind that its contents changed in v4.38. Use "
            "the forward method of RoPE from now on instead. It is not used in the `LlamaAttention` class"
        )
        return self._cos_cached

    # 5. forward函数不同
    @torch.no_grad()
    def forward(self, x, position_ids, seq_len=None):
        if seq_len is not None:
            logger.warning_once("The `seq_len` argument is deprecated and unused. It will be removed in v4.39.")

        # x: [bs, num_attention_heads, seq_len, head_size]
        inv_freq_expanded = self.inv_freq[None, :, None].float().expand(position_ids.shape[0], -1, 1)
        position_ids_expanded = position_ids[:, None, :].float()
        # Force float32 since bfloat16 loses precision on long contexts
        # See https://github.com/huggingface/transformers/pull/29285
        device_type = x.device.type
        device_type = device_type if isinstance(device_type, str) else "cpu"
        with torch.autocast(device_type=device_type, enabled=False):
            freqs = (inv_freq_expanded.float() @ position_ids_expanded.float()).transpose(1, 2)
            emb = torch.cat((freqs, freqs), dim=-1)
            cos = emb.cos()
            sin = emb.sin()
        return cos.to(dtype=x.dtype), sin.to(dtype=x.dtype)
```

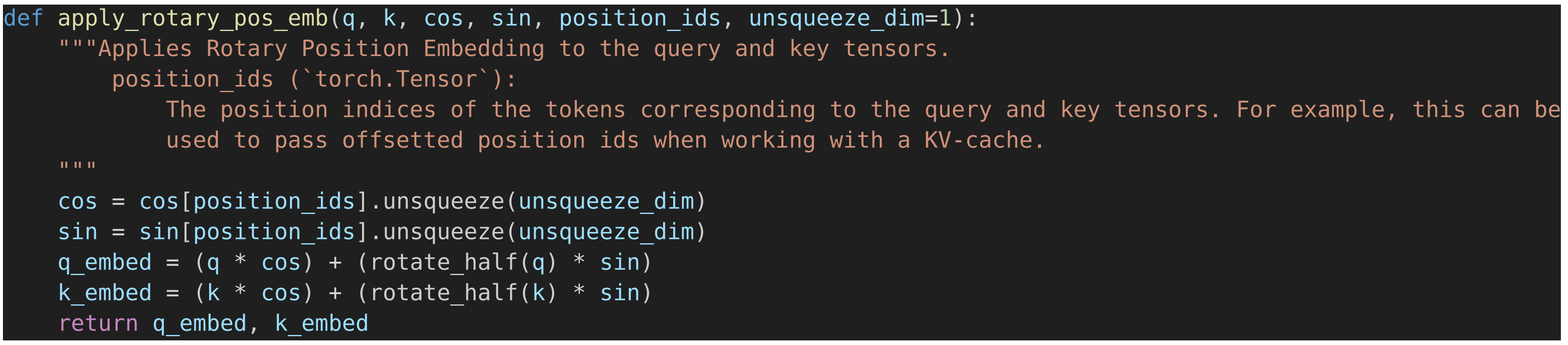
2. rotate_half: 两者完全相同

```
rotate_half: Copied from transformers.models.llama.modeling_llama.rotate_half
```

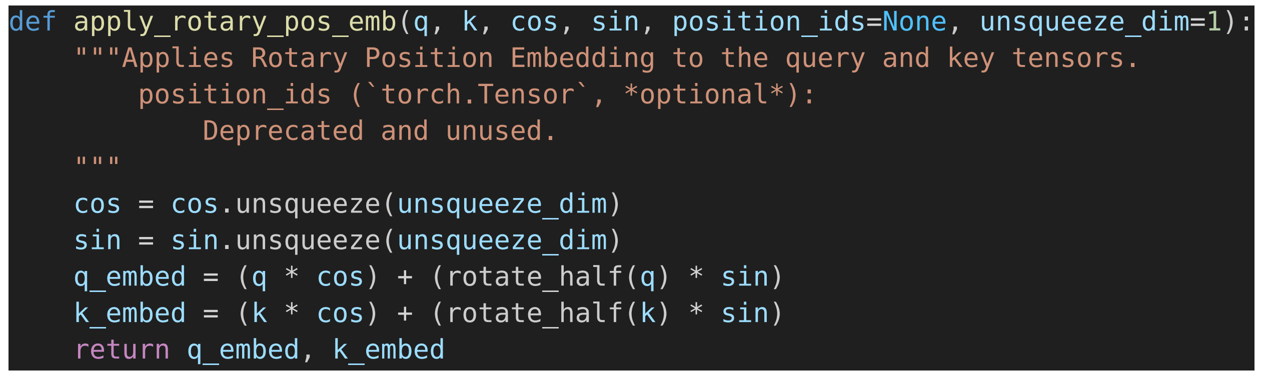
3. repeat_kv: 两者完全相同

```
repeat_kv: Copied from transformers.models.llama.modeling_llama.repeat_kv
```

4. apply_rotary_pos_embed: mistral比Llama多了参数position_ids



```python
def apply_rotary_pos_emb(q, k, cos, sin, position_ids, unsqueeze_dim=1):
    """Applies Rotary Position Embedding to the query and key tensors.
        position_ids (`torch.Tensor`):
            The position indices of the tokens corresponding to the query and key tensors. For example, this can be
            used to pass offsetted position ids when working with a KV-cache.
    """
    cos = cos[position_ids].unsqueeze(unsqueeze_dim)
    sin = sin[position_ids].unsqueeze(unsqueeze_dim)
    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    return q_embed, k_embed
```



```python
def apply_rotary_pos_emb(q, k, cos, sin, position_ids=None, unsqueeze_dim=1):
    """Applies Rotary Position Embedding to the query and key tensors.
        position_ids (`torch.Tensor`, *optional*):
            Deprecated and unused.
    """
    cos = cos.unsqueeze(unsqueeze_dim)
    sin = sin.unsqueeze(unsqueeze_dim)
    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    return q_embed, k_embed
```