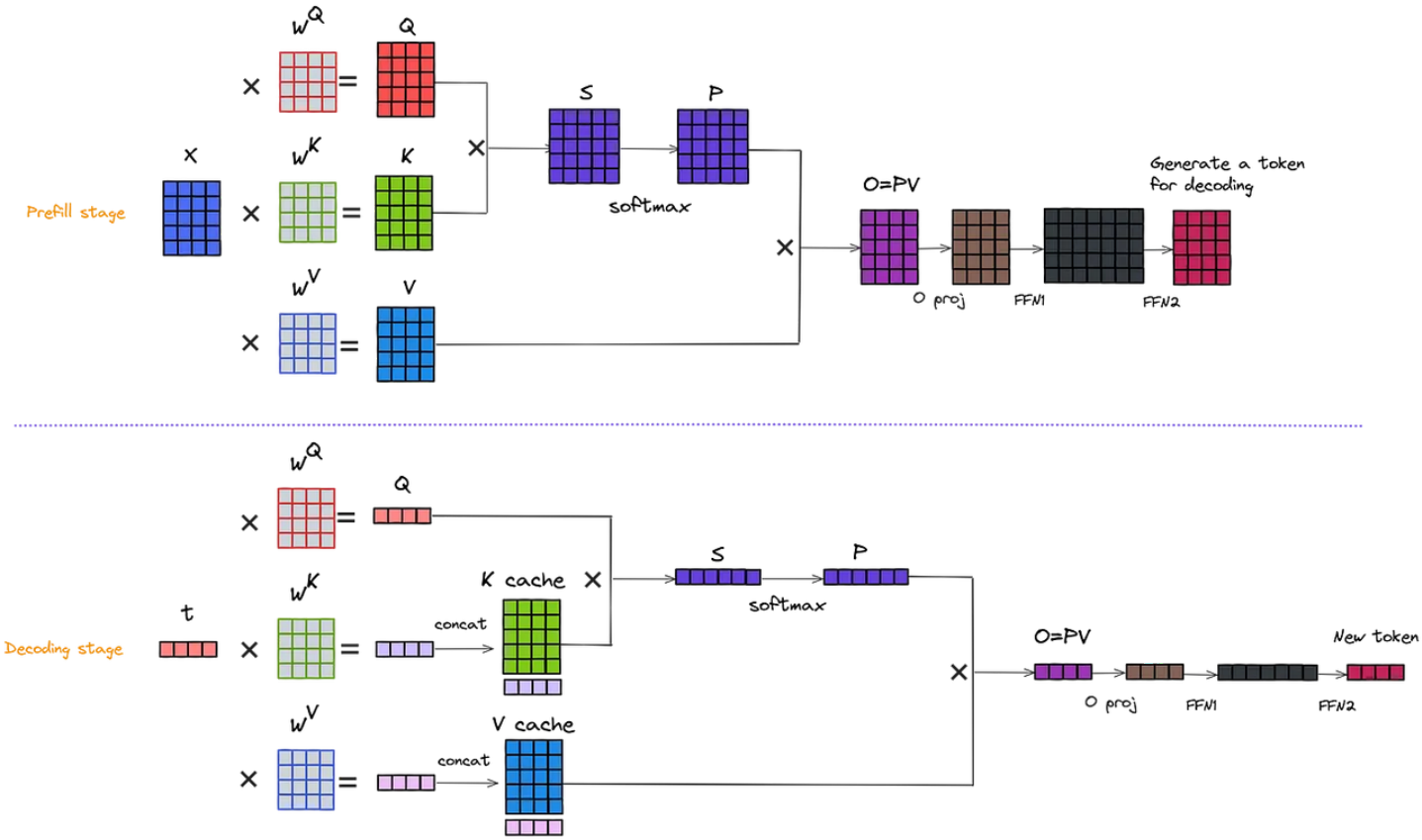
Prefill与Decode

在基于 Transformer 的大模型推理中，整个生成过程可分为两大阶段：

1. **预填充（Prefill）阶段**：对完整的输入prompt进行一次性并行计算，为每一个token生成并缓存键（Key）和值（Value）向量（即KV cache）。这一步只需运行一次，缓存内容将用于后续的生成步骤；
2. **解码（Decode）阶段**：模型以自回归方式逐步生成新 token。每一轮仅需计算当前要生成的token，并结合此前缓存的 KV （包括用户查询和已经生成的token）进行注意力计算，从而避免对整个历史序列重复计算，显著降低了计算量。



图引用自https://aiexpjourney.substack.com/p/main-stages-of-auto-regressive-decoding

Prefill阶段

- **Token 化**：输入为批大小为 `batch_size` 的若干prompt，每个 prompt 被处理为长度为 `seq_len` 的 Token 序列。
- **嵌入层**：这些Token序列会被映射为隐藏向量 `X` ，其形状为 `(batch_size, seq_len, hidden_dim)` 。

- **线性映射**：通过权重矩阵 W_Q, W_K, W_V （形状为 `(hidden_dim, num_heads * head_dim)` ），分别计算得到 Q, K, V ，并重塑为 `(batch_size, num_heads, seq_len, head_dim)` 。
- **KV cache缓存**：计算得到的 `K 和 V` 矩阵会被缓存下来，供后续解码阶段使用。
- **注意力计算：**

$$A = softmax(\frac{Q \cdot K^T}{\sqrt{d}})$$

其中 A 的形状为 `(batch_size, num_heads, seq_len, seq_len)` ，计算复杂度为 $O(batch\_size * num\_heads * seq\_len^2 * head\_dim) = O(batch\_size * seq\_len^2 * hidden\_dim)$

。

$$O = A \cdot V$$

其中 O 形状为 `(batch_size, num_heads, seq_len, head_dim)` ，随后会被重塑为 `(batch_size, seq_len, hidden_dim)` 并传递到下一层，计算复杂度同上。

> 这里只列出了复杂度最高的两步，总体计算复杂度为 $O(batch\_size * seq\_len^2 * hidden\_dim)$ 。

- **生成第一个token**：在工程实践中，为减少一次显存读写和前向传播开销，通常会直接利用prompt 最后一个位置的 hidden state来生成第一个token。

为什么要做prefill?

- **避免重复计算**：每一步解码都依赖于之前所有生成的中间结果，如果不在prefill阶段缓存KV，每生成一个新 token，都需要对整个历史序列（prompt + 已生成的 tokens）重新跑一次完整的自注意力计算，计算复杂度为 O(seq_len^2)，效率很低。
- **提高GPU 利用率**：Prefill阶段是典型的**计算密集型任务**，需要进行大规模的矩阵乘法运算。由于输入prompt是完整的，可以通过高度并行来最大化发挥 GPU 的算力优势。
- **便于并行优化：将prefill和decode阶段解耦后，可以分别在不同的GPU上执行。**prefill阶段适合**张量并行**来加速首token的生成，提升TTFT（Time To First Token），而decode 阶段则适合用**数据并行或流水线并行**来提升逐token的生成吞吐量，也就是TPOT（Tokens Per Output Time，生成两个连续生成的词元之间的平均时延）。这种解耦方式可以针对不同阶段采用最合适的并行策略，既能缩短首token的响应时间，也能提升整体生成速度。（相关结论可参考论文DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving）

Decode阶段

- **输入准备**：使用prefill阶段已经计算并存储的**KV cache，**以及**当前轮刚生成的 token（第一次decode时，是prefill阶段生成的第一个token）。**

- **嵌入层**：在第t步，将新生成的 token 转换为嵌入向量，计算得到 Q_t, K_t, V_t ，并将 K_t, V_t 更新到KV cache中。
- **计算注意力**：基于 Q_t 和现有的KV cache进行注意力计算，计算复杂度为 $O(seq\_len)$ ，相比prefill阶段大幅降低。
- **生成下一个token**：将最后一层的注意力输出经过线性层和softmax，得到下一个token的概率分布，并根据解码策略（如greedy search、beam search）选出下一个token，作为下一次decode的输入。

Decode阶段是典型的**通信密集型**任务，主要受限于显存带宽。因为生成一条回复通常需要多次decode，每次都需要访问和更新KV cache。随着生成内容长度的增加，KV cache也会越来越大，对GPU内存带宽的压力也随之增大。

代码块

```
# 基于KV cache实现的prefill和decode阶段
import torch
import torch.nn as nn
import torch.nn.functional as F

class MultiHeadAttentionWithKVCache(nn.Module):
    def __init__(self, hidden_dim, num_heads):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.num_heads = num_heads
        self.head_dim = hidden_dim // num_heads
        assert hidden_dim % num_heads == 0, "hidden_dim must be divisible by num_heads"

        self.q_proj = nn.Linear(hidden_dim, hidden_dim)
        self.k_proj = nn.Linear(hidden_dim, hidden_dim)
        self.v_proj = nn.Linear(hidden_dim, hidden_dim)
        self.out_proj = nn.Linear(hidden_dim, hidden_dim)

        self.k_cache = None
        self.v_cache = None

    def _split_heads(self, x):
        batch, seq_len, _ = x.size()
        x = x.view(batch, seq_len, self.num_heads, self.head_dim)
        return x.permute(0, 2, 1, 3)

    def _combine_heads(self, x):
        batch, num_heads, seq_len, head_dim = x.size()
        x = x.permute(0, 2, 1, 3).contiguous()
```

```
        return x.view(batch, seq_len, num_heads * head_dim)
    # Prefill
    def prefill(self, x):
        Q = self._split_heads(self.q_proj(x))
        K = self._split_heads(self.k_proj(x))
        V = self._split_heads(self.v_proj(x))
        self.k_cache, self.v_cache = K, V

        scores = torch.matmul(Q, K.transpose(-2, -1)) / (self.head_dim ** 0.5)
        attn = F.softmax(scores, dim=-1)
        context = torch.matmul(attn, V)
        context = self._combine_heads(context)
        return self.out_proj(context)
    # Decode
    def decode(self, new_x):
        Q_t = self._split_heads(self.q_proj(new_x))
        K_t = self._split_heads(self.k_proj(new_x))
        V_t = self._split_heads(self.v_proj(new_x))

        K_cat = torch.cat([self.k_cache, K_t], dim=2)   # 1..t
        V_cat = torch.cat([self.v_cache, V_t], dim=2)

        scores = torch.matmul(Q_t, K_cat.transpose(-2, -1)) / (self.head_dim ** 0.5)
        attn = F.softmax(scores, dim=-1)
        context = torch.matmul(attn, V_cat)
        out = self.out_proj(self._combine_heads(context))

        # 最后更新 cache
        self.k_cache, self.v_cache = K_cat, V_cat
        return out


# 测试
torch.manual_seed(0)
batch_size, seq_len, hidden_dim, num_heads = 2, 100, 16, 4
mha = MultiHeadAttentionWithKVCache(hidden_dim, num_heads)
prompt_embeddings = torch.randn(batch_size, seq_len, hidden_dim)

hidden_states = mha.prefill(prompt_embeddings)
print("Prefill 完成后，cache shapes:", mha.k_cache.shape, mha.v_cache.shape)
# Prefill 完成后，cache shapes: torch.Size([2, 4, 100, 4]) torch.Size([2, 4, 100, 4])

new_emb = torch.randn(batch_size, 1, hidden_dim)
mha.decode(new_emb)
print("Decode 1步后，cache shapes:", mha.k_cache.shape, mha.v_cache.shape)
```

```
# Decode 1步后，cache shapes: torch.Size([2, 4, 101, 4]) torch.Size([2, 4, 101, 4])

mha.decode(torch.randn(batch_size, 1, hidden_dim))
print("Decode 2步后，cache shapes:", mha.k_cache.shape, mha.v_cache.shape)
# Decode 2步后，cache shapes: torch.Size([2, 4, 102, 4]) torch.Size([2, 4, 102, 4])
```

Prefill和Decode阶段的一些优化方法：

1. **DistServe：**将预填充（prefill）和解码（decode）阶段解耦，分配到不同的GPU上，避免两阶段在同一设备上的互相争抢资源，从而提升整体性能。
2. **SARATHI**：针对不同长度的Prompt 导致的 Padding 冗余，采用Chunked Prefill技术，将较长的输入拆分为等长的chunk，每个chunk都能充分利用算力。同时，将Decode阶段的请求则“捎带”到空闲时段并行处理。
3. **缓解 KV cache带来的显存压力：**通过**分组查询注意力GQA**、将KV cache分布在多张GPU上存储与计算、或者建立CPU-GPU之间的KV cache调度机制，减轻单张显卡的显存压力。
4. **KV cache复用**：在客服、检索等高并发场景下，许多请求共享相同的prompt 前缀，可以跨请求复用已计算好的KV cache，节省重复计算资源。
5. **自注意力机制优化**：比如**FlashAttention**、**SparseAttention**等高效算法。

对比项	Prefill 阶段	Decode 阶段
主要功能	一次性并行处理完整输入序列，计算并缓存 Key/Value，用于后续生成首个及所有后续 token	每次只生成一个新 token，利用缓存的 Key/Value，生成新输出并持续更新缓存
计算类型	全序列自注意力计算，高度并行、计算密集型	增量式注意力计算，串行生成、通信密集型
GPU 利用率	充分利用算力资源，能高度并行，效率高	并行度低，通常是小批量或单 token 操作，GPU 利用率较低
计算复杂度	$O(seq\_len^2)$	$O(seq\_len)$
KV Cache 管理	不依赖缓存，直接基于原始输入并行计算，构建并缓存所有 prompt 的 KV cache	每步都查 KV cache，利用其计算注意力，并将新 token 的 K_t, V_t 持续追加到缓存中