KV Cache 深度解析：为什么 LLM 第一个 Token 最慢

原创 AI技术立文 AI技术立文 2026年4月5日 08:47 江苏

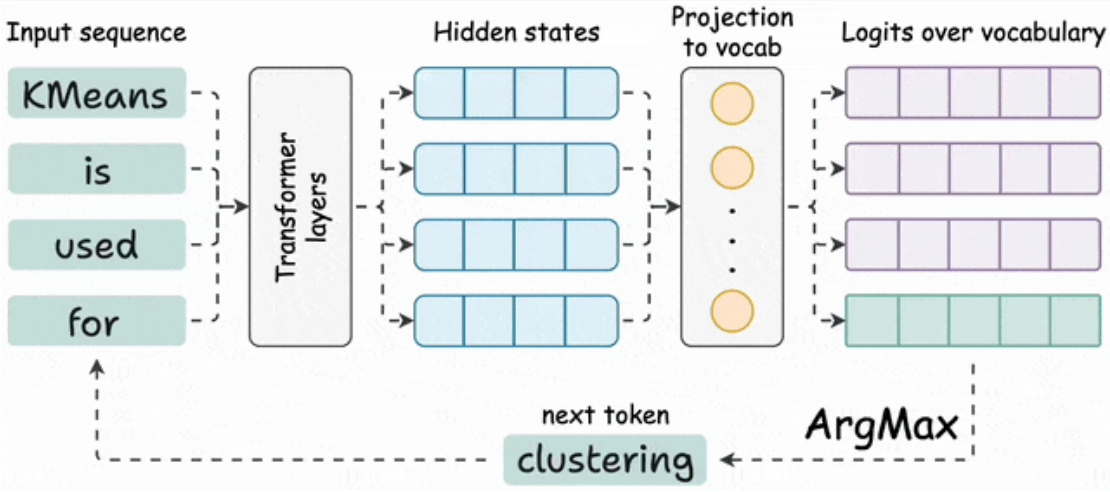
LLM的Token输出过程有一个现象：第一个 Token 往往慢半拍，后面的 Token 却几乎是秒出。其背后有个工程设计：KV 缓存（KV Cache），目标就是让 LLM 推理更快。

之前我也不是很了解这个KV 缓存到底怎么设计的，因为设计到矩阵运算，加上线性代数学艺不精，看到公式就头疼，这块一直没有详细了解。最近看到有篇文章通过大量视频+动图讲相关内容，分享给大家。

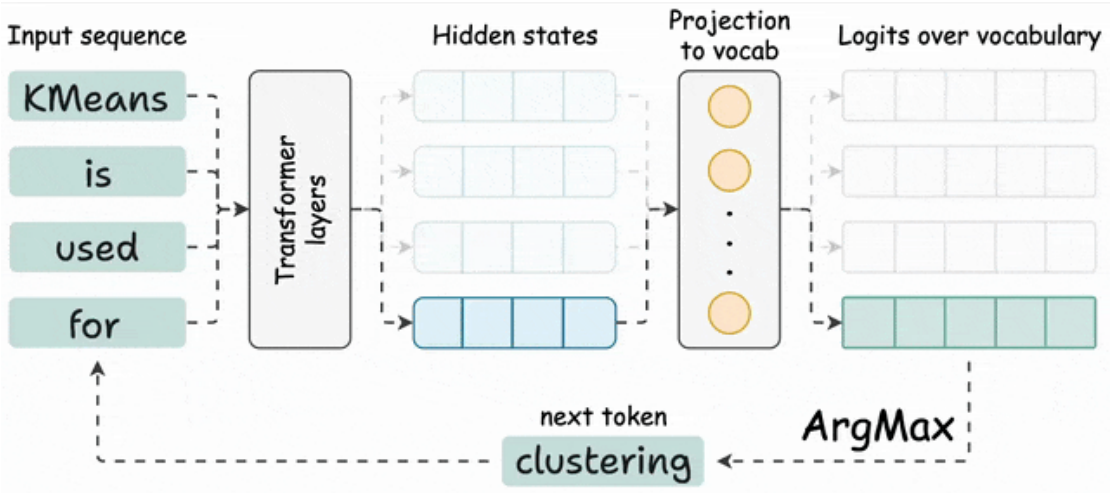
先看一眼有无 KV cache 的并排对比，再聊细节：

1/ LLM 如何生成 Token

Transformer 会先处理整段输入里的所有 Token，并给每个 Token 产出一个 hidden state。接着这些 hidden state 会被投影到词表空间，变成 logits（词表里每个词一个分数）。



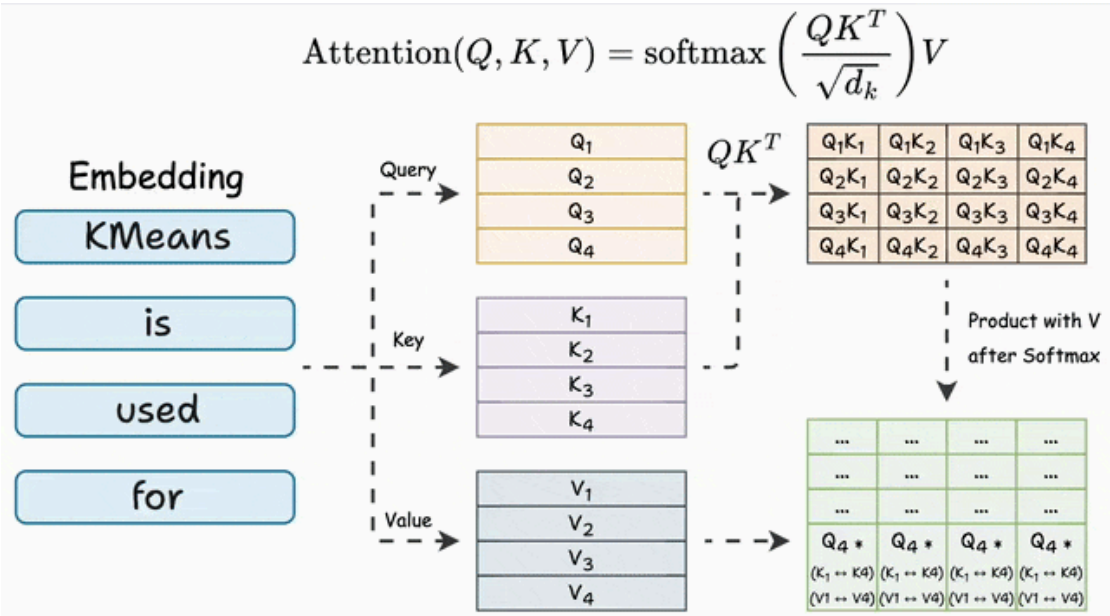
真正用于采样下一个 Token 的，只是最后一个 Token 对应的 logits。采样出新 Token 后把它拼回输入，然后重复这个过程。



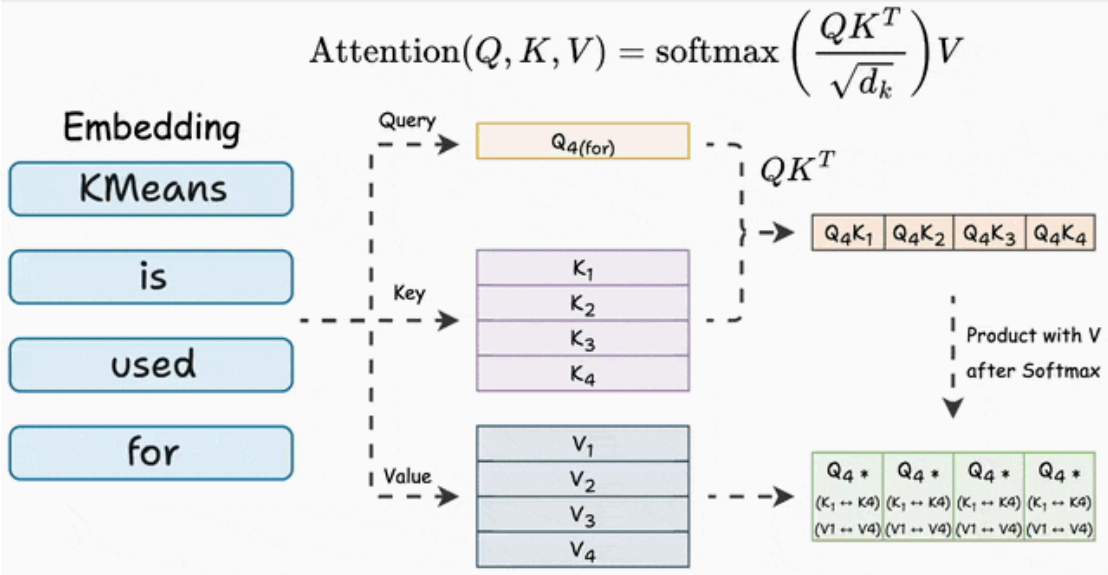
关键点就一句话：生成下一个 Token 时，真正必需的是最新那个 Token 的 hidden state。其他 hidden state 多数只是中间产物。

2/ Attention 到底在算什么

在每一层 Transformer 里，每个 Token 都会得到三个向量：query（Q）、key（K）、value（V）。Attention 先用 Q 和 K 做乘法得到分数，再用分数对 V 加权。



把视角收缩到最后一个 Token：



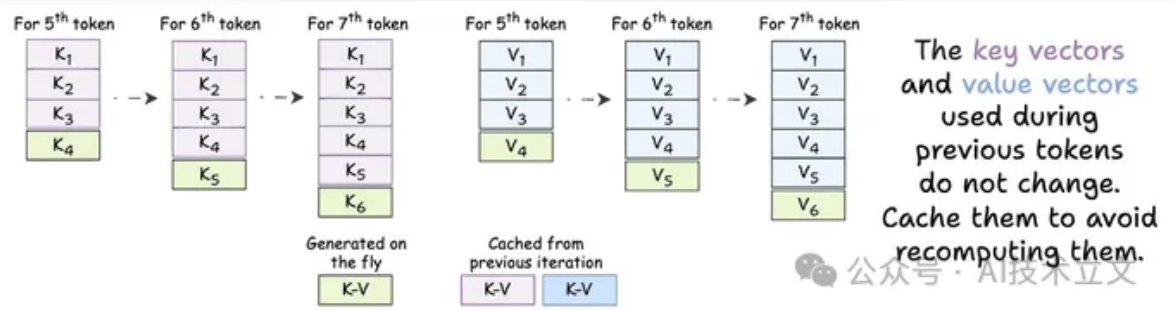
QK^T 的最后一行会用到两类信息：最后一个 Token 的 Q，外加序列里所有 Token 的 K。这个最后一行对应的 attention 输出，同样需要最后一个 Token 的 Q，以及全部的 K 和 V。

这就很明确了：为了算出我们唯一关心的那个 hidden state，每一层 attention 都要拿到最新 Token 的 Q，再配上历史全部 Token 的 K 与 V。

3/ 重复计算到底重复在哪

生成第 50 个 Token，需要 Token 1 到 50 的 K、V。生成第 51 个 Token，需要 Token 1 到 51 的 K、V。

问题在于：Token 1 到 49 的 K、V 早就算过，而且不会变。输入没变，输出当然不变；每一步还从头再算一遍，工程上看真的让人崩溃的。

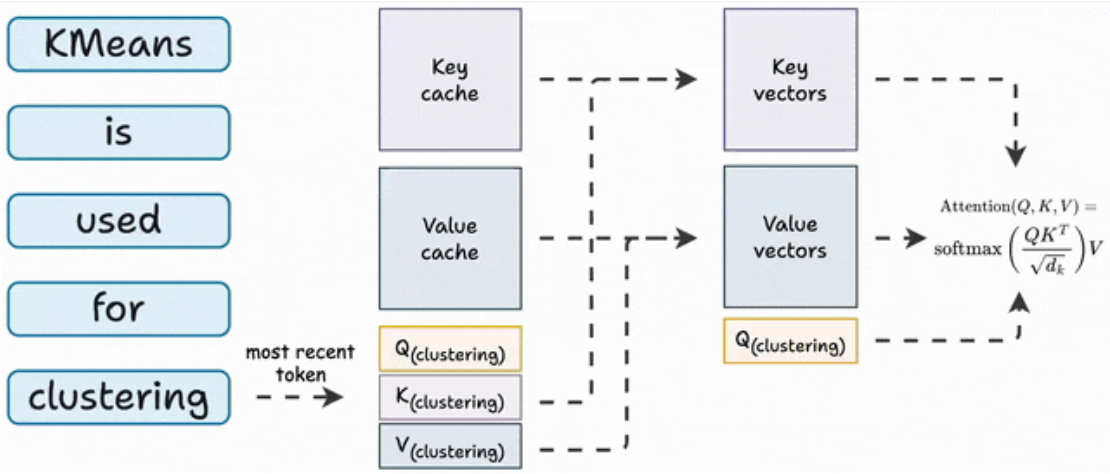


单步是 O(n) 的冗余计算，整段生成累积下来就是 O(n²) 的浪费。

4/ KV cache 怎么修这个问题

办法不复杂：每一步都重算全部 K、V，不如把已经算好的 K、V 存起来。每来一个新 Token，只做下面几件事：

1. 只为最新 Token 计算 Q、K、V。
2. 把新的 K、V 追加进 cache。
3. 从 cache 里取出历史全部 K、V。
4. 用新的 Q 对完整的 K、V cache 跑 attention。



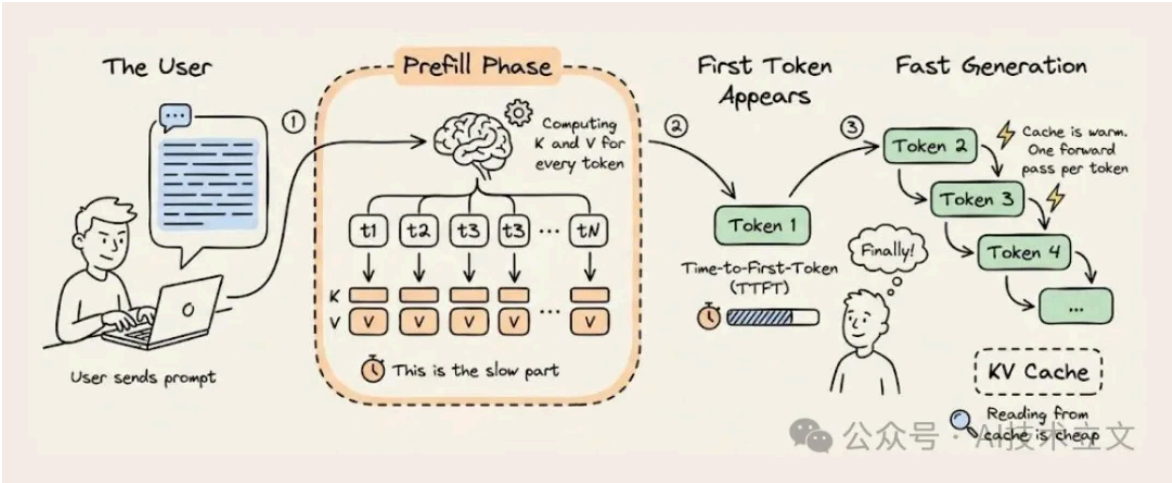
这就是 KV cache 的核心。每一层、每一步只新增一个 K 和一个 V，其余都直接读内存。

话虽这么说，attention 的计算量仍会随序列长度增长，因为你还是在看全部 K、V。真正昂贵的地方是 K、V 投影，这部分从"每一步都算"变成"每个 Token 只算一次"。

5/ 为什么第一个 Token 慢（TTFT）

现在就能解释首 Token 延迟了。你把 prompt 发给模型后，它会先做一次完整前向，把整段输入每个 Token 的 K、V 都算好并写进 cache。

这一段叫 prefill，通常是整次请求里最吃算力的阶段。cache 热起来后，后续每个 Token 基本只要单 Token 前向就行。



这个首 Token 延迟就叫 time-to-first-token（TTFT）。prompt 越长，prefill 越久，等待就越明显；优化 TTFT（chunked prefill、speculative decoding、prompt caching）是另一个大题目。

不管具体优化手段怎么选，底层规律没变：建 cache 贵，读 cache 便宜。

6/ 代价：省算力，吃显存

KV cache 本质是在拿内存换计算。每一层都要为每个 Token 存 K、V；以 Qwen 2.5 72B（80 层、32K context、hidden dim 8192）为例，单个请求就可能吃掉数 GB 的 GPU 显存。

并发一高，KV cache 往往比模型权重本身还占地方。这也是为什么会有 grouped-query attention（GQA）和 multi-query attention（MQA）：让多个 query head 共享 key/value head，显著降内存，质量损失通常可控。

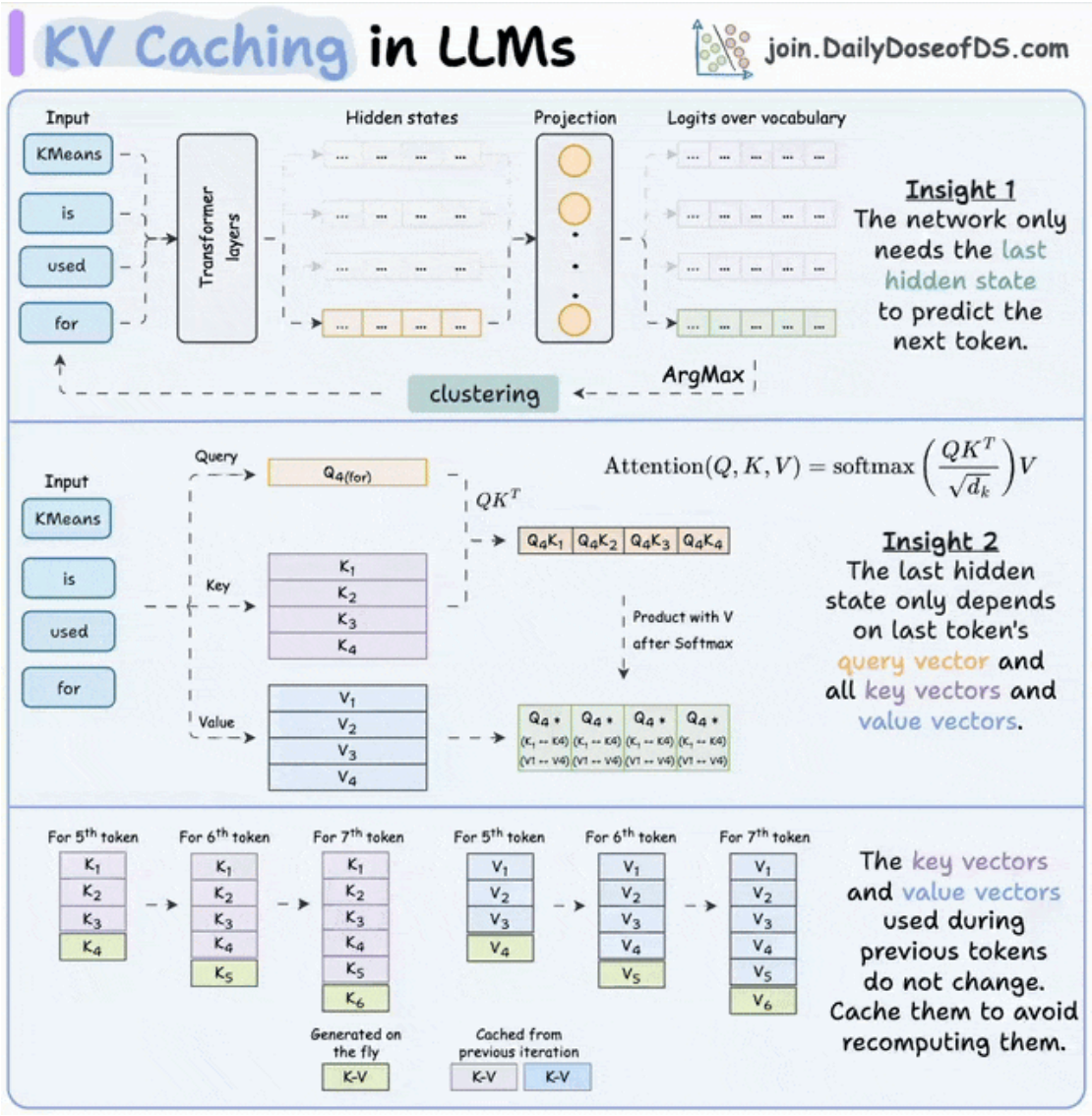
这也解释了为什么 context window 翻倍很难做。窗口翻倍，单请求 KV cache 也翻倍，可承载并发用户数自然会掉下来。

7/ 一页总结

KV cache 把自回归生成里的重复计算砍掉了。历史 Token 的 K、V 固定不变，只算一次并缓存；新 Token 只补自己的 Q、K、V，再拿完整 cache 做 attention。

工程实践里常见到 5x 左右提速。代价是显存占用上升，而且在大规模服务时，这个约束经常比纯算力更先撞线。

vLLM、TGI、TensorRT-LLM 这类主流服务栈，底层都建立在这套思路上。


KV Caching in LLMs
join.DailyDoseofDS.com
Input
KMeans
is
used
for
Transformer layers
Hidden states
Projection
Logits over vocabulary
ArgMax
clustering
Insight 1
The network only needs the last hidden state to predict the next token.
Query
Key
Value
$Q_{4(for)}$
K_1 K_2 K_3 K_4
V_1 V_2 V_3 V_4
QK^T
Q_4K_1 Q_4K_2 Q_4K_3 Q_4K_4
Product with V after Softmax
$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
Insight 2
The last hidden state only depends on last token's query vector and all key vectors and value vectors.
For 5th token
For 6th token
For 7th token
Generated on the fly
K-V
Cached from previous iteration
The key vectors and value vectors used during previous tokens do not change. Cache them to avoid recomputing them.

AI技术立文

喜欢作者