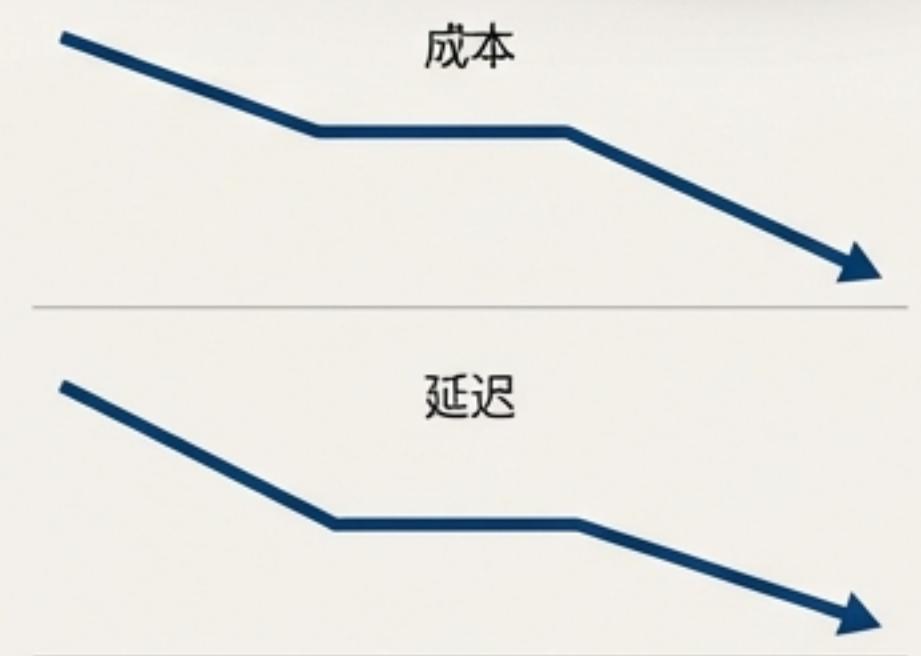


# “缓存”的秘密：LLM API 成本降低10倍，速度提升85%的背后

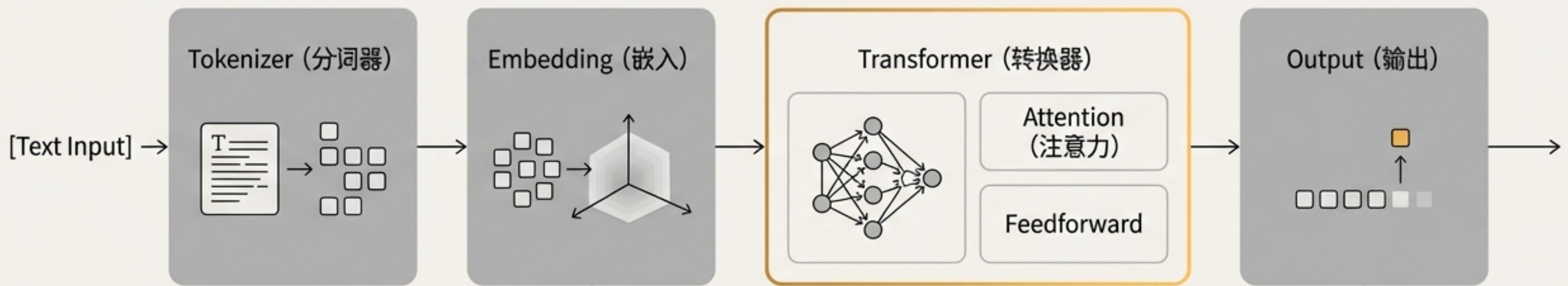
- 主要 API 提供商（如 OpenAI 和 Anthropic）的缓存输入 token 成本比普通 token 便宜 10 倍。
- Anthropic 声称，对于长 prompt，延迟最多可减少 85%。

**核心谜题：**每次 API 调用都会返回不同的答案。那么，他们究竟“缓存”了什么？这并不是简单的结果缓存。

**我们的旅程：**本次我们将深入LLM 内部，揭开这个“黑盒”的秘密，找出真正被缓存的数据，以及它是如何实现这一切的。



# 深入黑盒：一个 LLM 的四个核心处理阶段



从本质上讲，LLM 是一个巨大的数学函数，它接收一串数字作为输入，并生成一个数字作为输出。

这个过程可以分为四个主要部分：

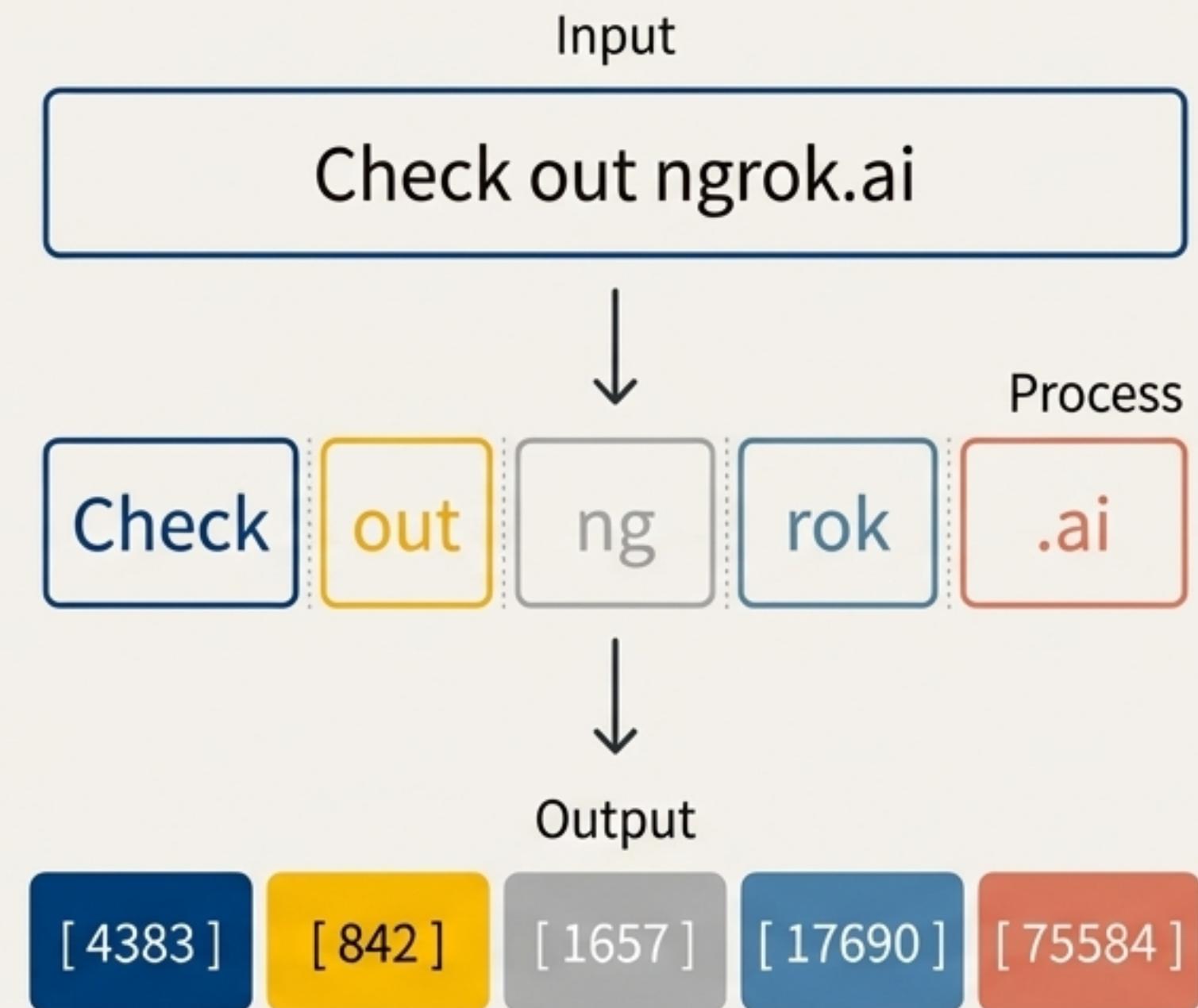
1. **Tokenizer (分词器)**：将文本转换为数字。
2. **Embedding (嵌入)**：将数字转换为有意义的多维向量。
3. **Transformer (转换器)**：通过 **Attention (注意力)** 机制处理关系。
4. **Output (输出)**：生成下一个数字 (token)。

**关键线索：**缓存的秘密就隐藏在“Transformer”的“Attention”机制中。我们将逐一探寻。

# 第一站：分词（Tokenization），将语言变为数字

分词器将输入文本分解成小块，并为每个独特的块分配一个称为“token”的整数 ID。

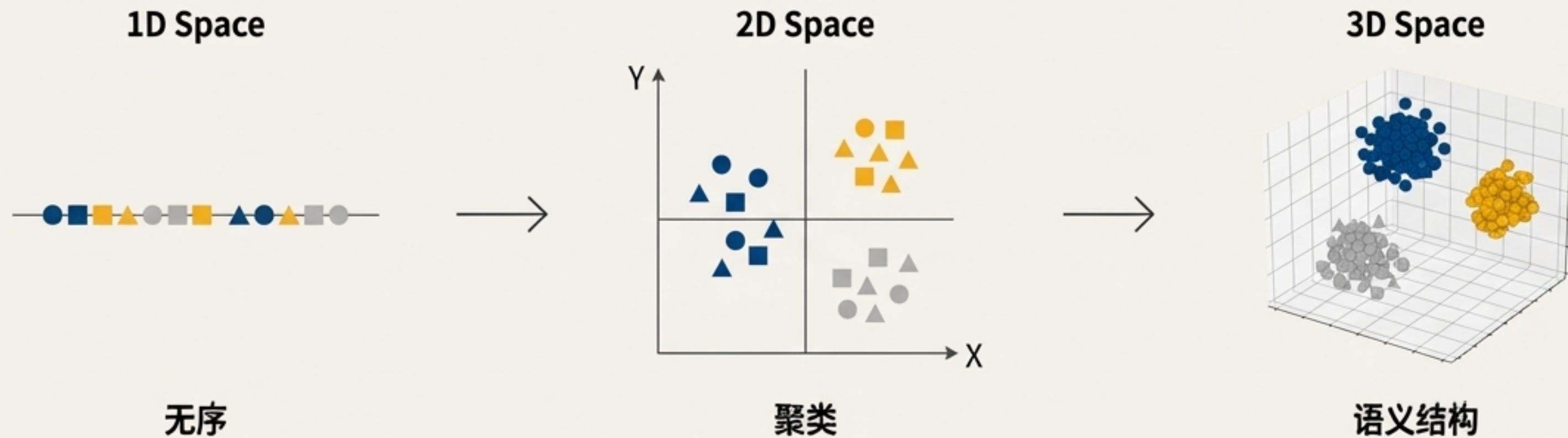
相同的 prompt 总是产生相同的 token。



Token 是 LLM 输入和输出的基本单位。我们在聊天应用中看到的逐字输出，实际上是模型在逐个 token 地生成。

**核心要点：**文本变成了机器可以处理的数字序列。但这还不够，这些数字本身没有“意义”。

# 第二站：嵌入（Embedding），赋予数字“意义”



Token 只是整数，无法表达词语之间的复杂关系（如同义、反义、情感等）。

Embedding 将每个 token 映射到高维空间中的一个点（一个向量）。这个点的位置代表了 token 的“语义”。

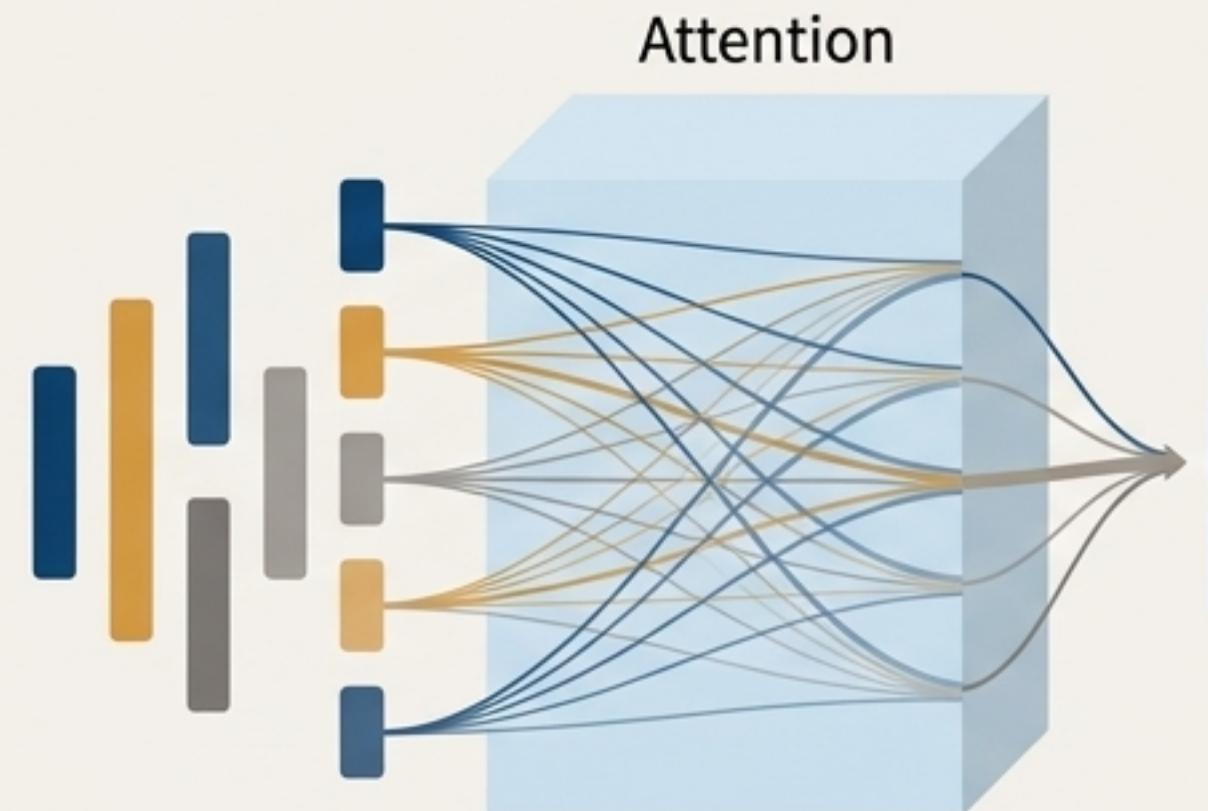
在训练过程中，模型会不断调整这些点的位置，使得意义相近的 token 在空间中也彼此靠近。

## 关键概念：

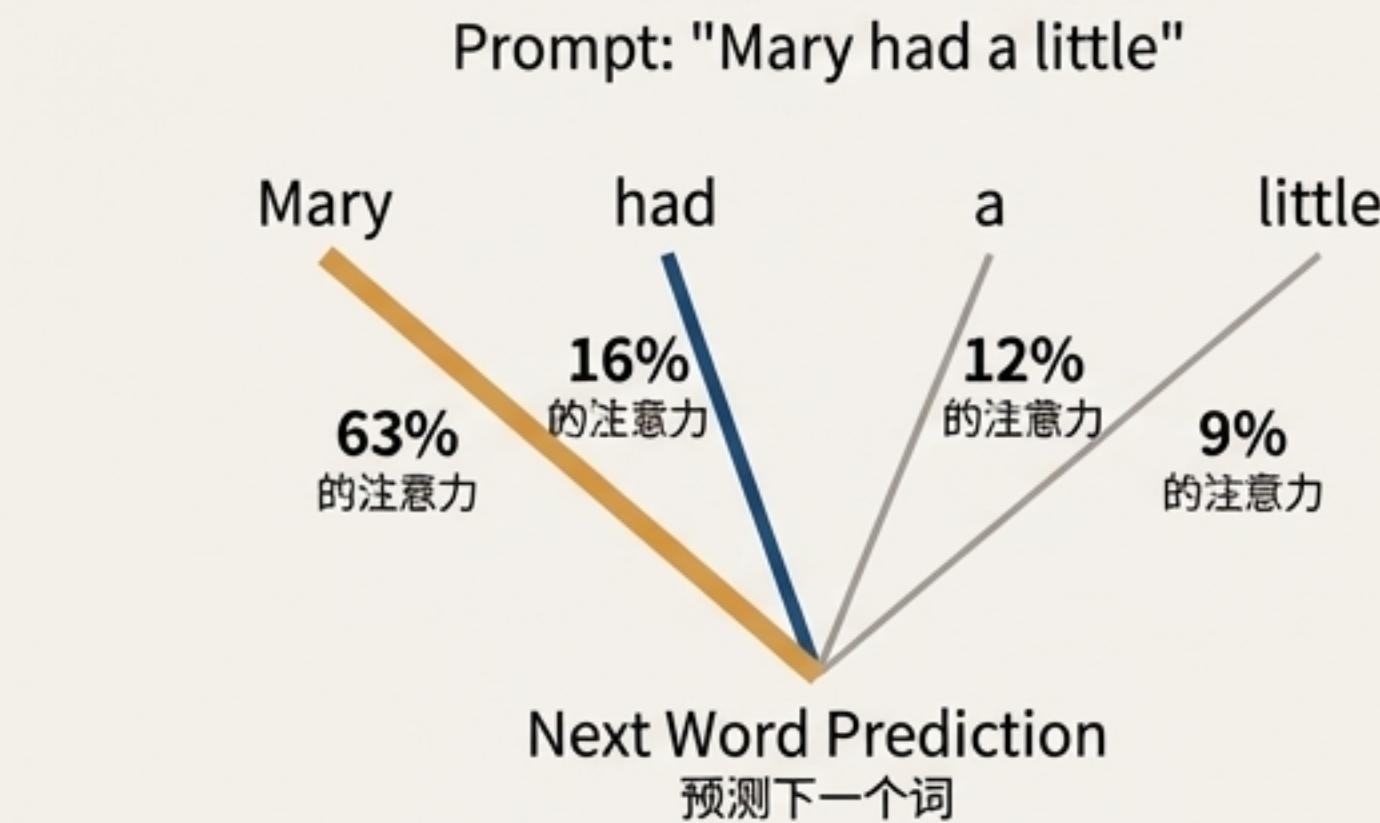
- **维度（Dimensions）**：维度越多，模型就能捕捉越复杂、越细微的语义关系。现代模型拥有数千甚至上万个维度。
- **位置编码（Positional Encoding）**：除了语义，token 在 prompt 中的顺序也很重要。Embedding 阶段也会将位置信息编码进去。

# 核心所在：Transformer 与 Attention 机制

## 核心概念



## 举例说明

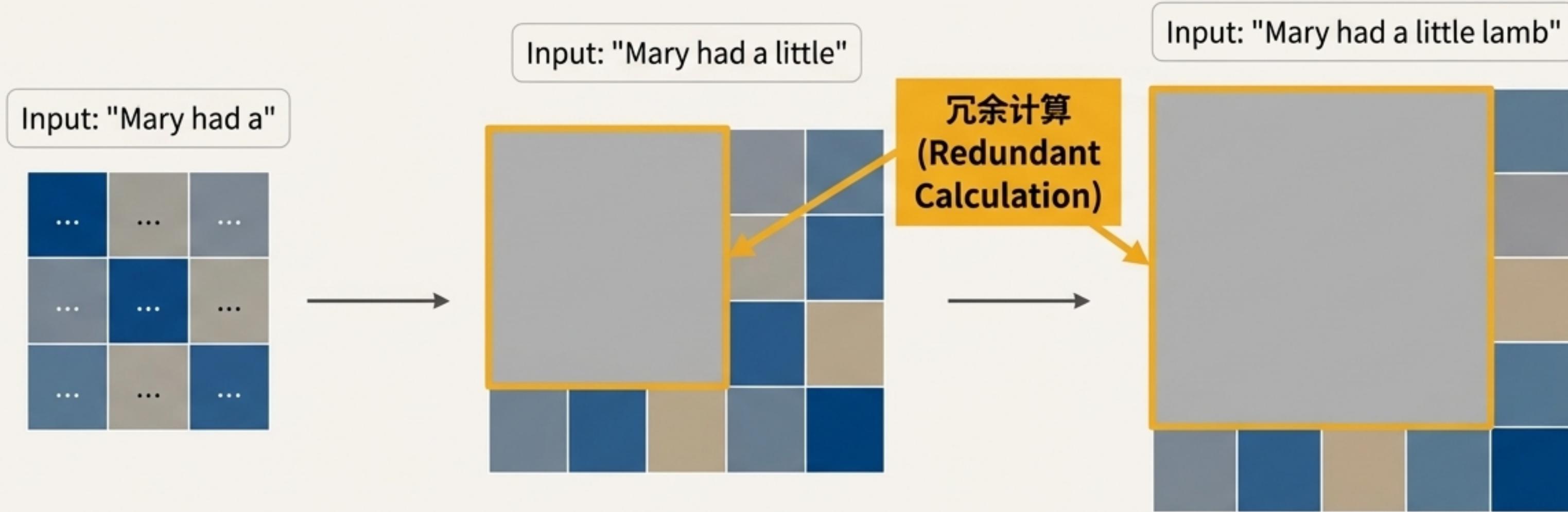


Attention 机制的目的是帮助 LLM 理解 prompt 中各个 token 之间的关系和重要性。

它通过加权组合 prompt 中所有 token 的 embedding 来实现这一点。

通过这种方式，模型“知道”应该更关注哪些词来预测后续内容。

# 推理循环中的巨大浪费：每次生成新 token 都要从头计算

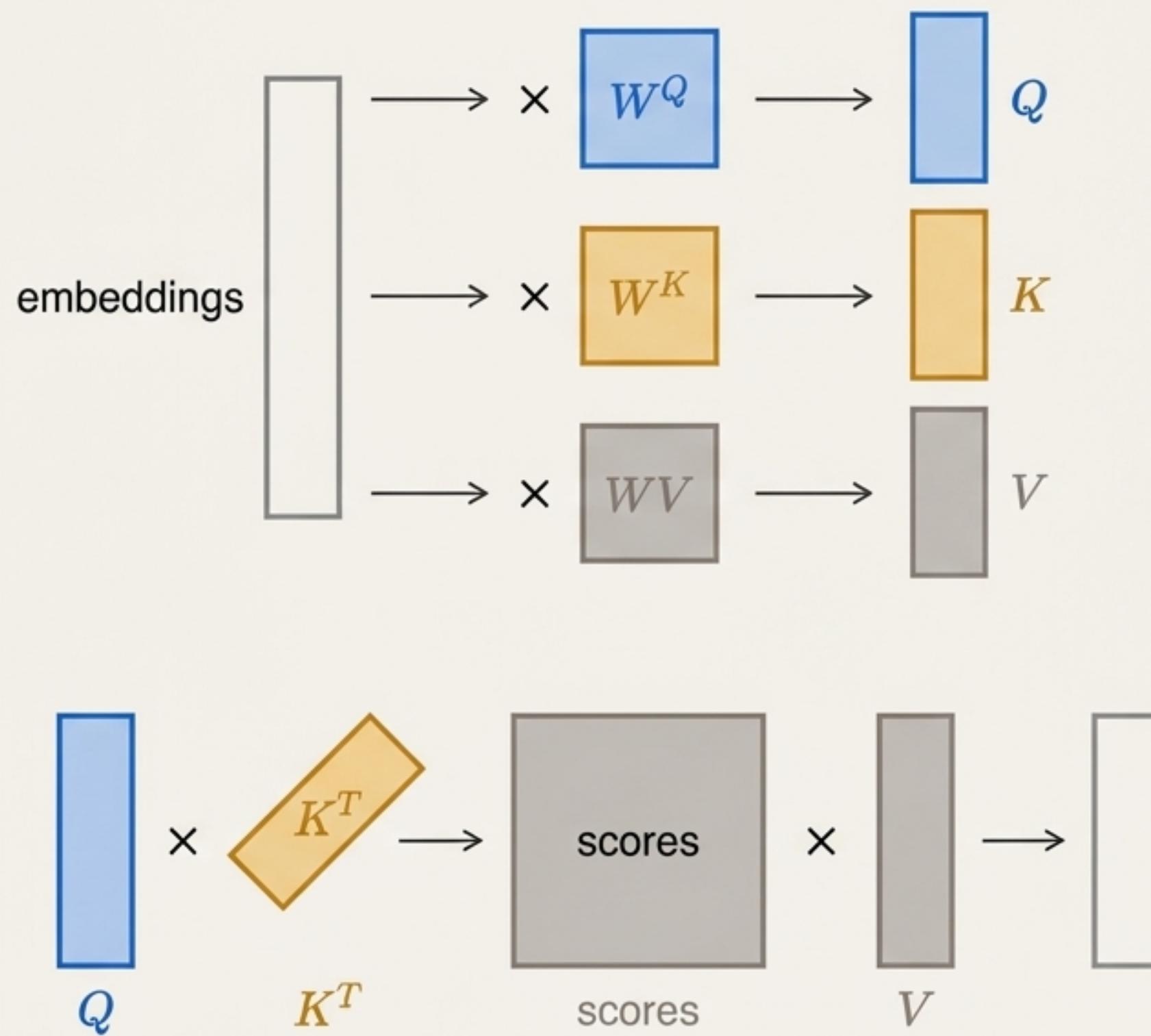


在标准的推理过程中，每生成一个新 token，它都会被附加到输入末尾。

然后，模型会为整个更新后的序列重新计算所有 token 之间的注意力权重。

**关键问题：**之前 token 的注意力权重不会改变，但我们却一遍又一遍地重复计算它们。这浪费了大量的计算资源，也是长 prompt 响应缓慢的根本原因。

# Attention 机制的“配方”：Query, Key, 和 Value



为了计算注意力权重，每个 token 的 embedding 会被转换成三个不同的向量：

- **Query (Q)**：代表当前 token 正在“寻找”什么信息。
- **Key (K)**：代表该 token 能“提供”什么信息。
- **Value (V)**：代表该 token 实际的“内容”或意义。

核心计算：

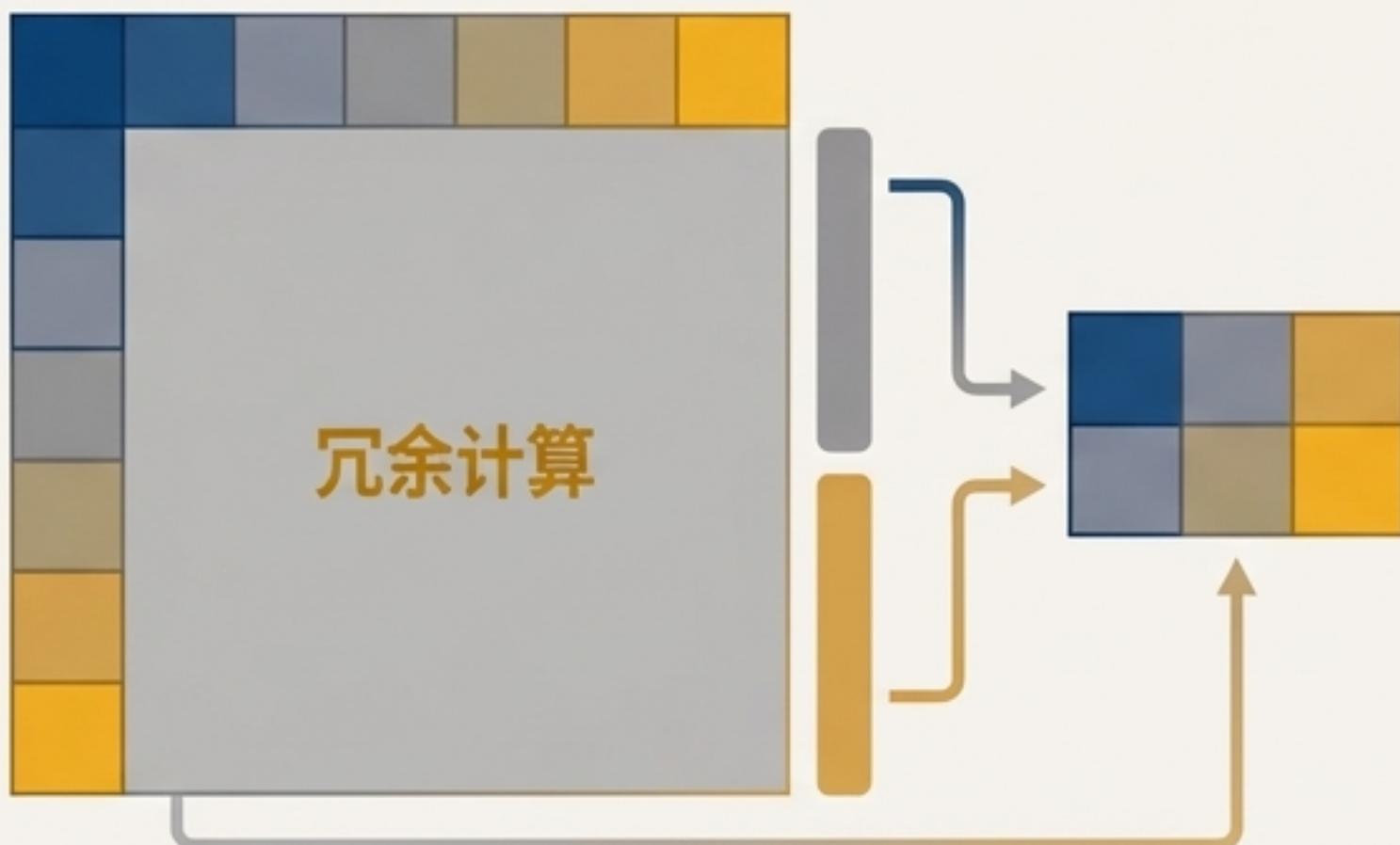
注意力分数是通过将每个 token 的 **Query (Q)** 与所有其他 token 的 **Key (K)** 相乘得到的  
( $\text{scores} = Q * K^T$ )。

这个分数决定了在混合最终结果时，每个 token 的 **Value (V)** 应该占多大的比重。

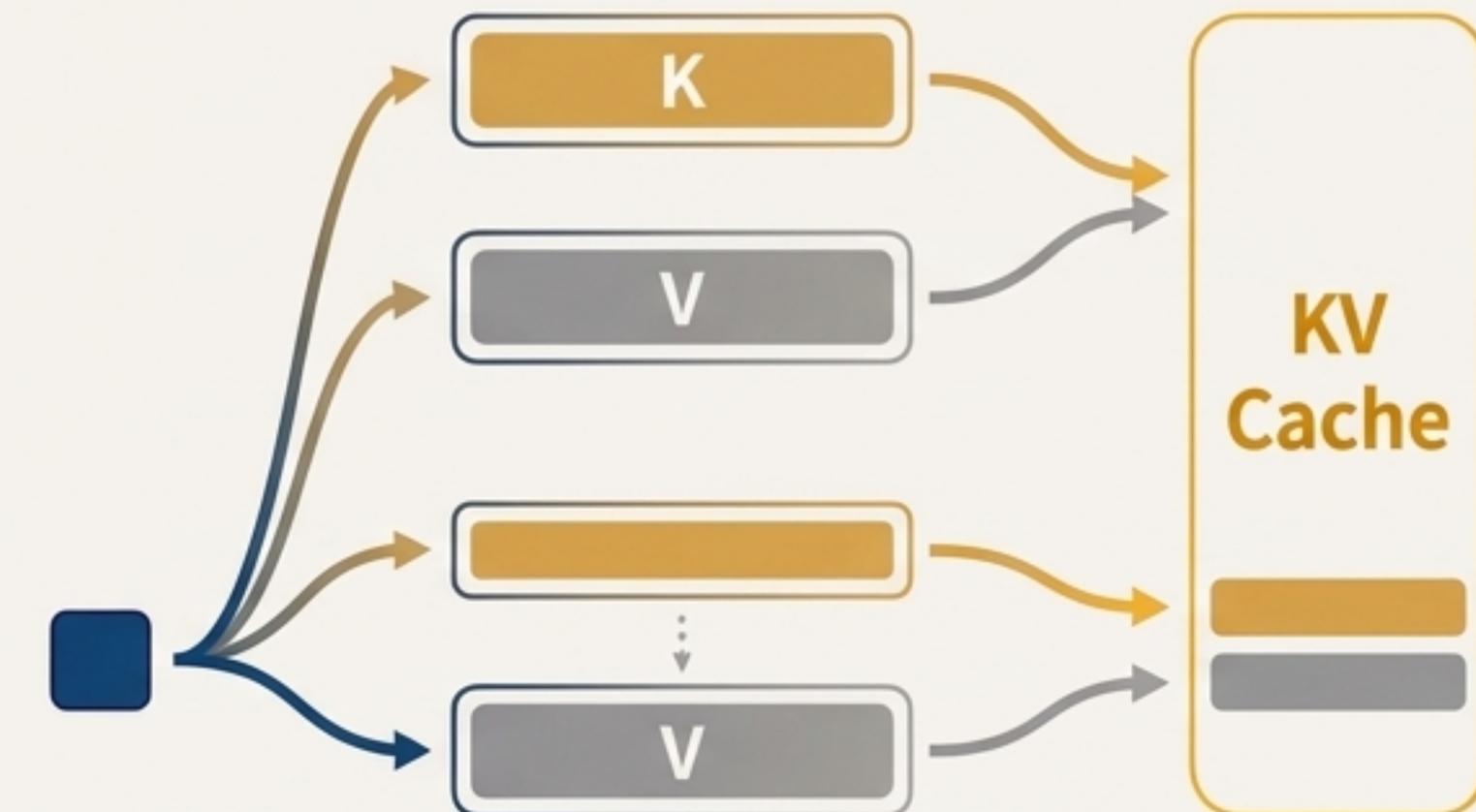
$W^Q$ ,  $W^K$ ,  $W^V$  是模型在训练期间学到的参数矩阵，它们在推理期间保持不变。

# 秘密揭晓：真正被缓存的，是 K 和 V 矩阵

左侧 (Before - 无缓存)



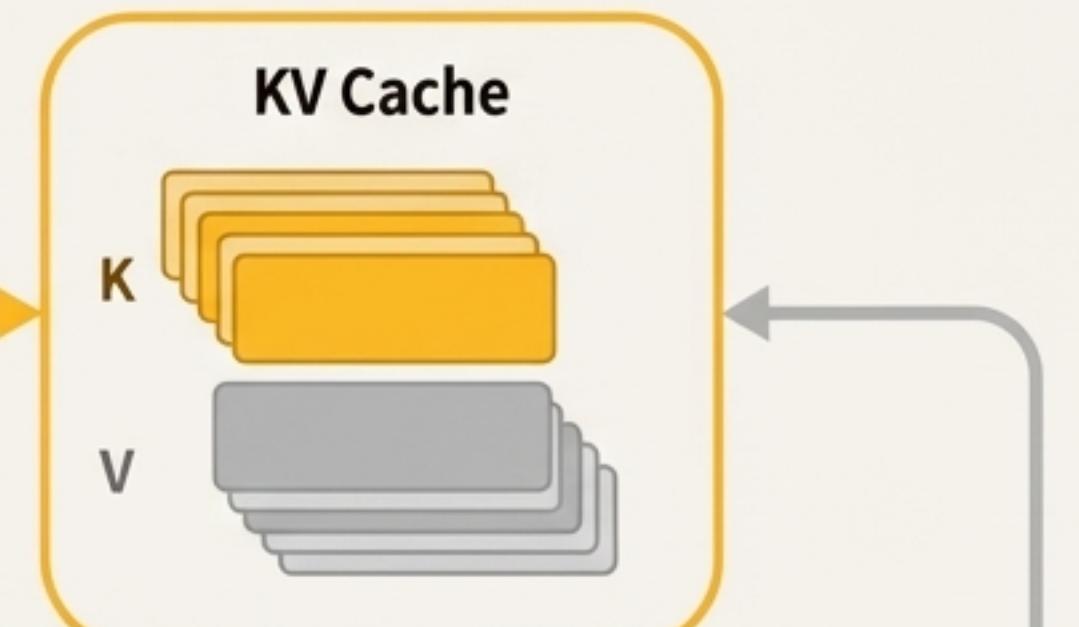
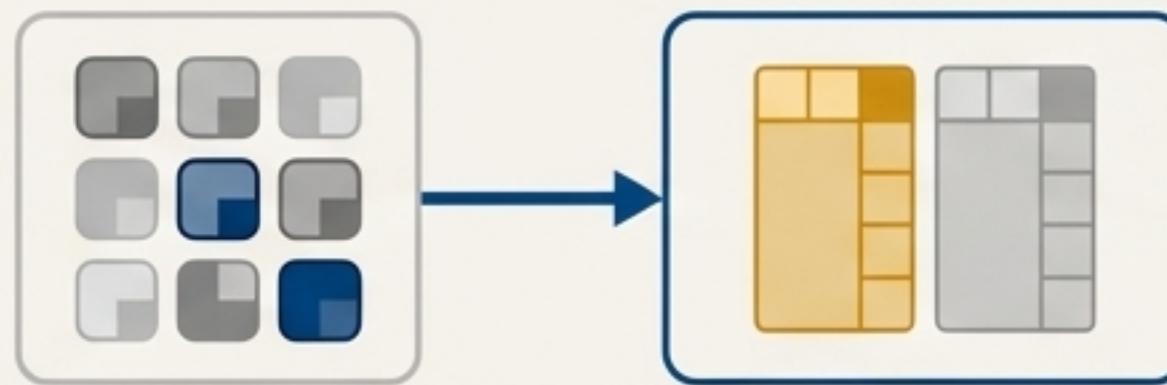
右侧 (After - KV 缓存)



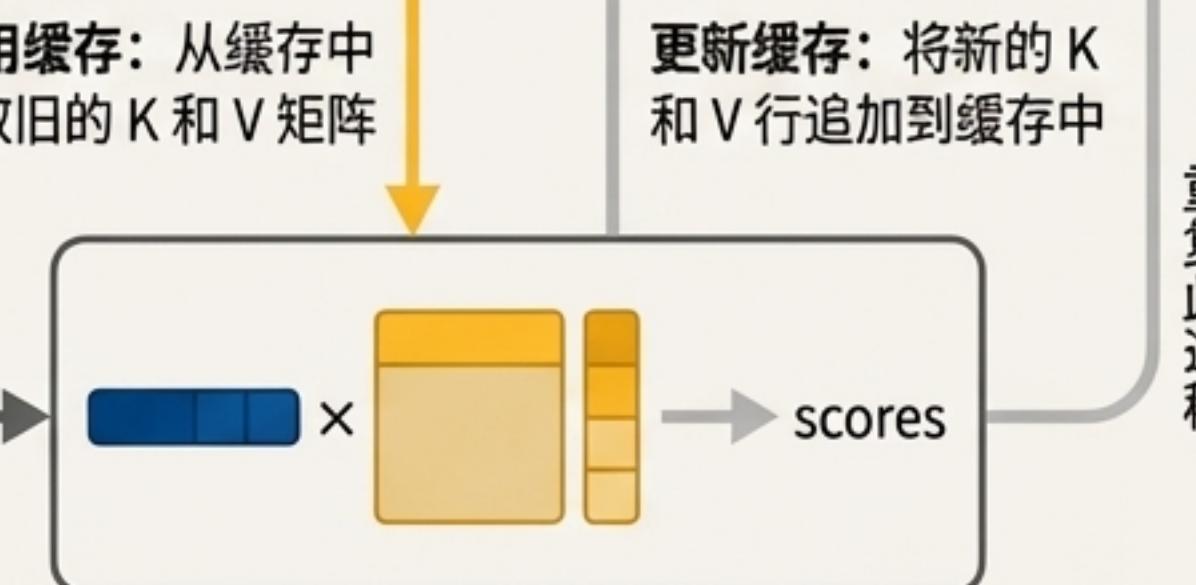
这，就是缓存的秘密。我们缓存的不是最终答案，而是注意力机制的中间产物：  
Key (K) 和 Value (V) 矩阵。这就是它被称为“KV Caching”的原因。

# KV 缓存如何运作：从 $N^2$ 到 $N$ 的飞跃

## Step 1: Initial Prompt



## Step 2: Generate Token #1



## The Benefit

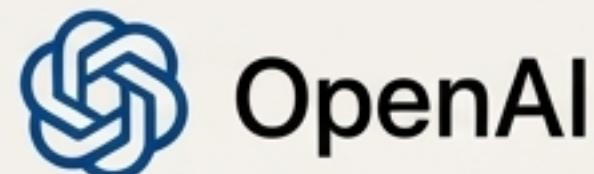
- 我们避免了为整个序列重新计算 K 和 V。
- 计算量从与 prompt 长度的平方 ( $N^2$ ) 相关，减少到只与 prompt 长度 ( $N$ ) 相关。
- 这极大地减少了计算量，从而降低了延迟和成本。

# 从技术原理到商业价值



结论：KV 缓存不是魔法，而是对 Transformer 架构核心机制的巧妙利用。

# 实践中的 KV 缓存：不同提供商的策略



- 策略：自动缓存。OpenAI 会自动尝试将请求路由到已缓存的条目。
- 效果：实验表明，连续发送相同请求的命中率约为 50%。
- 特点：使用简单，但性能可能不稳定。



- 策略：手动控制。用户可以通过 API 参数明确要求缓存 prompt。
- 效果：实验表明，请求缓存时的命中率是 100%。
- 特点：控制力更强，更适合需要可预测延迟的应用。

## 一个常见误区

`temperature`，`top\_p` 等控制随机性的参数会影响缓存吗？

不会。这些参数在注意力计算之后的最后一步（token 选择）才起作用，因此更改它们不会使 KV 缓存失效。

# 理解底层，掌控全局

我们一起揭开了 LLM 的“黑盒”，理解了从文本到 token，再到 embedding，最终通过 Attention 机制生成结果的全过程。

我们发现，KV 缓存是通过保存和重用 K、V 矩阵来避免冗余计算，从而实现降本增效的关键。

---

当您在云端或本地管理 LLM 流量时，深度的理解至关重要。

**\*\*ngrok.ai\*\***: 一个统一的平台，为懂行的构建者而生，帮助您路由、保护和管理通往任何 LLM 的流量。

[Learn more at [ngrok.ai/docs](https://ngrok.ai/docs)]

ngrok.ai