1. 基础

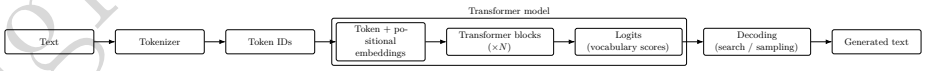
编码、解码与大语言模型的核心构件。

Table of contents

1 概览

大语言模型（LLM, Large Language Model）本质上是一类“下一个词元（token）预测器”：给定前文，预测最可能出现的下一个符号。本章不打算把你训练成“公式背诵机”，而是帮你建立一套够用、清晰、能上手排障的心智模型。

在推理（inference）阶段，典型流程是：文本先经过分词器（tokenizer）变成词元 ID（token ID）；词元 ID 查表得到嵌入向量（embedding）；向量经过多层 Transformer 模块（自注意力 + 前馈网络）；最后输出 logits（词表上的未归一化分数）。解码器再根据这些 logits 选择下一个词元，循环生成文本。



面试和工程里真正有价值的，不是背全公式，而是能回答三件事：1. 系统由哪些关键模块组成。2. 每个模块有哪些核心权衡（质量 vs 成本、延迟 vs 吞吐）。3. 常见故障通常发生在哪一层（分词、训练稳定性、解码策略等）。

1.1 术语约定

为保证全章一致，后文统一采用以下写法：

- 词元（token）
- 词元 ID（token ID）
- 分词器（tokenizer）
- 嵌入（embedding）
- 注意力（attention）
- 前馈网络（FFN, Feed-Forward Network）

读完本章后，你应该能够清楚解释：为什么子词分词成为主流；嵌入与位置编码怎么配合；自注意力与掩码如何工作，以及长上下文为何昂贵；残差、归一化与 FFN 的作用；绝对位置、RoPE、ALiBi 的差异；以及贪心、beam、top-k/top-p、温度等解码方法的适用场景与失效模式。

2 分词（Tokenization）

分词是把文本映射成模型可处理词元序列的过程，而且这个映射通常是确定性的。

它之所以重要，是因为分词会同时影响三件关键事情：

- 质量：领域词汇能否被合理表达。
- 成本：词元数量直接决定计算量。
- 多语言表现：不同语言、噪声文本、特殊符号能否稳定处理。

更换 tokenizer 不是“小改动”，而是改变模型输入符号系统；这会影响训练假设、评估可比性，以及线上行为一致性。

2.1 分词粒度

常见分词粒度有词级、字符级和子词级。

词级（**word-level**）：整词入表，直观，但有三个典型问题：OOV 严重、词表尾部过长、词形变化共享差（如 *swim* 与 *swimming*）。

字符级（**character-level**）：几乎没有 OOV，但序列会明显变长，导致注意力计算成本上升。

子词级（**subword-level**）：在覆盖率和序列长度之间取得折中。它通常能比词级更稳健地覆盖新词，又比字符级更省计算，因此成为主流。

2.2 子词词表如何学习

多数子词算法输入是语料和目标词表大小 V，输出是：

- 一组子词词表。
- 一套把文本切成这些子词的规则。

核心目标很明确：在保持覆盖的同时尽量减少词元数，从而降低注意力成本。

BPE（Byte-Pair Encoding）（包含 byte-level BPE）通过反复合并高频相邻单元来构建词表。[code]

BBPE（byte-level BPE）在字节层面做合并，对多语言与脏数据通常更稳，也能处理任意字节序列。

WordPiece 倾向于选择能提高语言模型似然的合并。直觉上，如果两个片段经常一起出现，就更值得合并成一个词元。

一种常见打分写法是对数增益（也可看作 PMI 风格）：

$$\log P(z) - (\log P(x) + \log P(y)) = \log \frac{P(z)}{P(x)P(y)}$$

其中 $z = xy$。若 $P(z)$ 显著大于 $P(x)P(y)$，说明“连在一起”比“分开出现”更合理，合并就更有收益。

Unigram LM 路径相反：先从大候选集出发，再删掉对整体似然影响最小的词元。可把它理解为“在所有合法切分中找最可能的一种”：

$$p(t_{1:m}) = \prod_{i=1}^{m} p(t_i),$$

$$\hat{t}_{1:m} = \arg \max_{t_{1:m}:\, \mathrm{concat}(t_{1:m})=s} \prod_{i=1}^{m} p(t_i).$$

实践中常用动态规划（Viterbi 风格）求最优切分，并保留单字符回退，避免 OOV。

小例子：把 `unhappiness` 当成一个词看，词级分词容易 OOV；按字符切又太长。子词分词会更像 `un + happi + ness`，既看得懂，又不啰嗦。

一句话总结：BPE/WordPiece 是“从小到大合并”，Unigram 是“从大到小裁剪”。

2.3 面试题

2.3.1 为什么 LLM 更偏好子词分词？

它同时解决了覆盖和效率：比词级更少 OOV，比字符级序列更短。

2.3.2 词表大小如何影响质量与算力？

词表更大：嵌入/softmax 更重；词表更小：序列变长、注意力更贵。最佳点要结合语料分布和部署约束。

2.3.3 什么情况下要扩展 tokenizer？风险是什么？

当领域词（如医学术语、代码标识符）被过度切碎，或多语言覆盖明显不足时可考虑扩展。风险包括：嵌入失配、历史指标不可比、训练与线上预处理不一致。

2.3.4 分词常见坑有哪些？

分词器不一致。训练和服务使用不同 tokenizer/chat template，容易引发格式、工具调用与安全规则问题。

领域词碎片化。关键术语被切成过多词元，成本上升且语义表达变差。

指标不可比。困惑度等词元级指标跨 tokenizer 不能直接比较。

数值脆弱性。数字切分方式会影响计数、排序、算式行为。

3 嵌入（Embedding）

嵌入把离散词元映射到连续向量空间，让模型可以做线性代数计算。可以把它理解成“符号系统与神经网络计算之间的接口层”。

3.1 从 one-hot 到稠密向量

词元 ID 的数字本身没有语义，ID=9 并不比 ID=3“更大”或“更接近”。如果直接把 ID 当数值输入，模型会学到错误结构。

one-hot 表示避免了这个问题：词表大小为 V 时，词元 t 对应 $x_t \in \mathbb{R}^V$，只有第 t 位是 1，其余为 0。它干净但高维，且任何两个词元都是正交关系。

稠密嵌入（dense embedding）把 V 维 one-hot 映射到 d 维向量（$d \ll V$），既降维又让模型学习“相似词元在向量空间更接近”的几何结构。

小例子：可以把 embedding 理解成“仓库坐标系”。功能相近的词元（比如 `deploy`、`release`）往往会被摆在相邻货架，而不是仓库两端。

3.2 嵌入查表（Embedding Lookup）

设词表大小 V、嵌入维度 d，嵌入矩阵为

$$E \in \mathbb{R}^{V \times d}.$$

给定词元 ID t，查表得到 $e_t = E[t]$。在 PyTorch 中对应 `nn.Embedding(vocab_size, embedding_dim)`。

等价地，若 x_t 是 one-hot，则

$$e_t = x_t^\top E.$$

这个式子强调了一点：embedding lookup 本质是稀疏线性映射。

3.3 嵌入怎么学出来

经典路径是 word2vec 系列。

CBOW（Continuous Bag of Words）：用上下文预测中心词。它训练快、语义效果好，但几乎不建模词序。

设上下文词为 $w_1, w_2, \ldots, w_C$，目标词为 w_t。每个上下文词可表示为 one-hot 向量 $x_i \in \mathbb{R}^V$，嵌入矩阵 $W \in \mathbb{R}^{V\times d}$ 把它映射为 $v_i = W^T x_i$。隐藏表示为平均值 $h = \frac{1}{C}\sum_{i=1}^{C} v_i$，再通过第二个矩阵 $W' \in \mathbb{R}^{d\times V}$ 得到分数 $u = {W'}^T h$，于是：

$$p(w_t \mid \text{context}) = \frac{\exp(u_t)}{\sum_{j=1}^{V}\exp(u_j)}.$$

Skip-gram 反过来：用中心词预测上下文。通常对稀有词更友好，但计算量更大。

FastText 把词表示成字符 n-gram 向量之和，能更好处理形态变化和 OOV。例如 `playing` 可拆为 `<pla, lay, ayi, yin, ing>` 等片段：

$$v_{\text{playing}} = \sum_{g\in G} z_g.$$

其中 G 是字符 n-gram 集合，z_g 是对应 n-gram 的向量。训练时可在这个表示之上继续做 CBOW 或 skip-gram。

3.4 训练加速技巧

Hierarchical softmax 用树结构近似全词表 softmax，复杂度由 $|V|$ 降到 $\log V$。

Negative sampling 把多分类近似成“正样本 vs 若干负样本”的二分类训练。示例：

```
(target = "cat", context = "cute")
```

```
("cat", "banana")
("cat", "engine")
("cat", "chair")
```

其目标常写为：

$$L = \log\sigma(v_+^\top v_t) + \sum_{i=1}^{K}\log\sigma(-v_{\text{neg}_i}^\top v_t),$$

其中 v_+ 为真实上下文，v_{neg_i} 为负样本向量。

3.5 权重绑定（Weight Tying）

在 decoder-only LLM 中，输出层常把隐藏态映射到词表 logits：$W_{\text{out}} \in \mathbb{R}^{d\times V}$。权重绑定指输出矩阵与输入 embedding 共享（常见形式 $W_{\text{out}} = E^\top$）。

这样做通常有两个收益：参数更少，样本效率更好。

3.6 词向量 vs 句向量

词元嵌入（token embedding）表示单个词表项；句向量（sentence embedding）表示整段文本。后者通常来自词元隐状态池化（均值池化、[CLS] 等）或专门训练的表示模型。

3.7 面试题

3.7.1 为什么不能直接用词元 ID？

因为 ID 是任意标签。embedding 才能提供可学习、可度量的连续表示空间。

3.7.2 什么是 weight tying，为什么常用？

输入 embedding 和输出投影共享参数，减少模型参数量，常提升训练效率。

3.7.3 词向量和句向量的核心差异？

前者表示“词项”，后者表示“整段语义”。

4 注意力（Attention）

注意力机制让 Transformer 在每个位置动态选择“该看谁、看多少”。工程上它既是表达力来源，也是算力账单来源。

4.1 自注意力（Self-attention）

对序列中每个位置，模型会构造：

- Query：我要找什么信息。
- Key：我能提供什么匹配线索。
- Value：如果我被关注，我贡献什么内容。

小例子：句子“服务器重启后，它恢复了”。这里“它”更可能关注“服务器”而不是“重启”，这就是注意力在做的“指代配对”。

设输入隐藏态为 $X \in \mathbb{R}^{L\times d}$，单头注意力中：

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V,$$

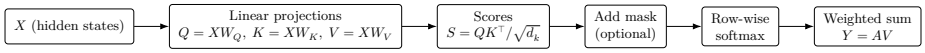
其中 $W_Q, W_K \in \mathbb{R}^{d\times d_k}$，$W_V \in \mathbb{R}^{d\times d_v}$。

然后执行三步：

1. 打分：
$$S = \frac{QK^\top}{\sqrt{d_k}}.$$
除以 $\sqrt{d_k}$ 是为了防止维度增大导致 softmax 饱和。
2. 掩码 + 归一化：
$$A = \mathrm{softmax}(S).$$
3. 加权聚合：
$$Y = AV.$$

合并写法：

$$\text{Attention}(Q,K,V)=\text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V.$$



伪代码 (scaled dot-product attention)

```
# X: [batch, seq, d_model]
# Wq, Wk: [d_model, d_k], Wv: [d_model, d_v]

Q = X @ Wq  # [batch, seq, d_k]
K = X @ Wk  # [batch, seq, d_k]
V = X @ Wv  # [batch, seq, d_v]

# scores: [batch, seq, seq]
scores = Q @ K.transpose(-2, -1)
scores = scores / sqrt(d_k)

# Causal mask (decoder-only)
# set j > i to -inf so softmax gives 0 probability.
# causal_mask: [1, seq, seq] or [batch, seq, seq]
scores = scores + causal_mask

weights = softmax(scores, dim=-1)  # row-wise over keys
Y = weights @ V  # [batch, seq, d_v]
```

多头注意力（MHA）是把上述过程并行做多次（每头维度更小），再拼接回模型维度。这样不同头能关注不同关系模式（局部语法、长程依赖、位置模式等）。

4.2 因果掩码（Causal masking）

decoder-only 训练与推理都必须遵守“不能看未来词元”。因果掩码通过把 $j>i$ 的分数设为 $-\infty$（或极大负数），让 softmax 后该概率为 0，从而保证自回归分解 $p(x_t \mid x_{<t})$。

4.3 交叉注意力（Cross-attention）

交叉注意力中，Query 来自一个序列，Key/Value 来自另一个序列。典型场景是 encoder-decoder：decoder 在生成时读取 encoder 输出；多模态模型也常用该机制（文本关注图像/音频特征）。

4.4 多头注意力（MHA）

多头的价值在于“分工”：每个头学到不同模式，组合后表达力更强。

4.5 为什么长上下文成本高

标准注意力每头都要构造 $L \times L$ 分数矩阵，因此计算和（朴素实现下）内存都是 $O(L^2)$。这也是长上下文 prefill 成本高、TTFT 变慢的根源。

4.6 注意力加速思路

稀疏注意力：限制可见模式（滑窗、块稀疏），减少计算条目。见 [code]。

线性注意力：用核映射近似 softmax，将计算重排到 $O(L)$。

KV cache：推理时缓存历史 K/V，避免每步重算历史词元。

FlashAttention：通过分块与稳定归约提升显存效率，不显式物化完整 $L \times L$。

MQA/GQA：共享部分或全部 K/V，主要优化解码期内存和带宽。

4.7 面试题

4.7.1 什么是因果掩码，为什么必须有？

它阻止未来信息泄露，保证训练目标与生成过程一致。

4.7.2 为什么标准注意力是 $O(L^2)$？

每个位置都要与所有位置交互，形成 $L \times L$ 矩阵。

4.7.3 如何把标准注意力改成“线性注意力”？

用核特征映射近似 softmax：

$$\exp(q^\top k) \approx \phi(q)^\top \phi(k),$$

$$\mathrm{Attn}(Q,K,V) \approx \frac{\phi(Q)(\phi(K)^\top V)}{\phi(Q)(\phi(K)^\top \mathbf{1})}.$$

这样可流式累计而不构造全矩阵，代价是归纳偏置改变、精度可能受影响。

4.7.4 KV cache 的作用是什么？

缓存历史 K/V，生成新词元时只算新增一步，显著降低解码计算量。

4.7.5 MQA/GQA 改了什么，为什么重要？

它们通过共享 K/V 缩小 KV cache，直接缓解解码内存与带宽瓶颈。

5 前馈网络、激活函数、残差连接与归一化

5.1 FFN

FFN（也叫 MLP）是 Transformer 中“每个词元自己做计算”的子层。注意力负责跨词元传递信息，FFN 负责在单词元内构造更复杂特征。

小例子：注意力像“把信息从各个服务拉过来”，FFN 像“在本地做业务逻辑处理”。只拉数据不处理，你拿到的还是半成品。

没有非线性时，多层线性变换可折叠成一层，表达力受限。FFN 提供非线性建模能力，也是 Transformer 的主要计算预算之一。

标准两层形式：

$$\mathrm{FFN}(x) = W_2\,\sigma(W_1 x + b_1) + b_2.$$

实践中常先扩展维度（如 $d_{\mathrm{ff}} \approx 4d_{\mathrm{model}}$），再用门控激活（如 SwiGLU）提高质量/算力比。

5.2 激活函数

激活函数决定 FFN 的非线性形态，也影响训练稳定性和效率。

5.2.1 激活类型

5.2.1.1 经典：ReLU

ReLU 简单高效，但负半轴梯度为 0，可能出现“死亡 ReLU”。

$$\mathrm{FFN}(x) = f(xW_1 + b_1)W_2 + b_2,$$
$$\mathrm{FFN}_{\mathrm{ReLU}}(x) = \mathrm{ReLU}(xW_1 + b_1)W_2 + b_2.$$

5.2.1.2 平滑：GELU / Swish

GELU、Swish 在大模型里往往优化更平滑，是常见默认。

5.2.1.3 门控：GLU 家族（SwiGLU）

门控 MLP 的通式：

$$\mathrm{FFN}_{\mathrm{gated}}(x) = ((xW_{\mathrm{up}}) \odot g(xW_{\mathrm{gate}}))W_{\mathrm{down}} + b,$$

其中 $g(\cdot)$ 是门控函数。GLU 常取 sigmoid，SwiGLU 常取 Swish。

激活函数	公式	说明
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	大 $\lvert x\rvert$ 时易饱和，可能导致梯度消失；含指数运算
Tanh	$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	零中心，但仍会饱和；含指数运算

激活函数	公式	说明
ReLU	$\mathrm{ReLU}(x) = \max(0, x)$	简单高效；$x < 0$ 区域梯度为 0（“死亡 ReLU”）
Leaky ReLU	$\mathrm{LeakyReLU}(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x < 0 \end{cases}$	通过负半轴小斜率缓解死亡 ReLU
ELU	$\mathrm{ELU}(x) = \begin{cases} x, & x > 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$	平滑；负值输出有助于均值回正；指数计算成本较高
Swish	$\mathrm{Swish}(x) = x \cdot \sigma(x)$	平滑、非单调；实证表现常较好
GELU	$\mathrm{GELU}(x) = x \cdot \Phi(x)$	Transformer 常用默认；可理解为概率门控
SwiGLU	$\mathrm{SwiGLU}(x) = \mathrm{Swish}(x_1) \odot x_2$	现代 LLM 中常见且有效的门控 MLP 激活
Softmax	$\mathrm{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$	将 logits 映射为类别概率分布

5.3 残差连接（Residual）

深层网络常见三类问题：梯度消失/爆炸、优化困难、表示尺度漂移。残差连接通过显式恒等路径改善这些问题：

$$x_{l+1} = x_l + f(x_l).$$

这意味着每层只需学习“增量”，训练更稳定，也更容易加深网络。

5.4 归一化（Normalization）

随着层数加深，激活尺度会漂移，导致训练脆弱，尤其在混合精度下更明显。归一化让每层输入尺度更可控。

Transformer 中最常见的是 LayerNorm 和 RMSNorm：

方法	公式	说明
LayerNorm	$x_{\text{norm}} = \frac{x - \mathbb{E}[x]}{\sqrt{\mathrm{Var}(x) + \epsilon}} \cdot \gamma + \beta$	经典选择；早期 Transformer 中非常常见
RMSNorm	$\mathrm{RMS}(x) = \sqrt{\frac{1}{H}\sum_{i=1}^{H} x_i^2 + \epsilon}, \quad x_{\text{norm}} = \frac{x}{\mathrm{RMS}(x)} \cdot \gamma$	比 LayerNorm 更简洁；现代 LLM（如 LLaMA 系）广泛使用

5.4.1 常见 Transformer 变体

关键差异通常是“归一化在前还是在后”：

- **Post-norm** :

$$x_{l+1} = \mathrm{Norm}(x_l + f(x_l)).$$

- **Pre-norm** :

$$x_{l+1} = x_l + f(\mathrm{Norm}(x_l)).$$

现代大多数 decoder-only LLM 更偏好 pre-norm，因为深层下更稳定。

方法	公式	说明
标准残差 (**Standard Residual**)	$x_{l+1} = x_l + f(x_l)$	经典 ResNet 风格跳连
后归一化 **Transformer (Post-Norm)**	$x_{l+1} = \mathrm{Norm}(x_l + f(x_l))$	早期 Transformer（含 BERT）常见；超深时稳定性较弱
前归一化 **Transformer (Pre-Norm)**	$x_{l+1} = x_l + f(\mathrm{Norm}(x_l))$	现代 LLM（GPT、LLaMA）常见；通常更稳定
DeepNorm	$x_{l+1} = x_l + \alpha \cdot f(\mathrm{Norm}(x_l))$ Encoder: $\alpha = \frac{1}{\sqrt{2N}}$ Decoder: $\alpha = \frac{1}{\sqrt{4N}}$	通过残差缩放提升超深模型稳定性；N 为层数
ReZero	$x_{l+1} = x_l + \alpha \cdot f(x_l)$ (α learnable, init $= 0$)	从近零残差起步，可稳定训练早期阶段

5.5 面试题

5.5.1 已有 attention，为什么还需要 FFN？

注意力擅长“信息路由”，FFN 负责“非线性特征构造”，两者职责互补。

5.5.2 为什么现代 LLM 喜欢 SwiGLU 这类门控激活？

门控相当于动态特征开关，通常在相近预算下带来更高表达效率。

5.5.3 表达力与计算的权衡是什么？

更强激活通常带来更多参数与 matmul，成本上升；换来更好训练与精度。

5.5.4 残差连接为什么有效？

它提供稳定梯度通路，让每层学习增量而不是重写全部表示。

5.5.5 pre-norm 和 post-norm 的核心区别？

pre-norm 先归一化再进入子层，通常在深层模型中更稳定；post-norm 在较浅模型中也常见。

6 位置编码（Positional Encoding）

> **Note**
>
> 位置编码告诉模型“先后顺序”。没有位置信号，词袋顺序变化在注意力里难以区分。

6.1 为什么需要位置编码

自注意力本身是“集合运算”倾向的：如果不给位置信号，模型很难可靠地区分词序变化。语言里词序会改变语义，所以必须显式注入位置信息。

它还直接影响长上下文能力：

- 是否能外推到训练之外的长度。
- 是否天然偏好局部依赖或长程依赖。

6.2 位置编码是什么

位置编码可以放在不同层级：

- 嵌入层：给每个位置加位置向量（绝对位置）。
- 注意力分数层：按距离加偏置（相对位置）。
- 几何层：旋转 Q/K（RoPE），把位置信息编码到点积几何关系里。

小例子：没有位置编码时，“先上线再回滚”和“先回滚再上线”在模型眼里容易像同一堆词元。线上事故会告诉你：这两句话绝对不是一回事。

6.3 实现方式

6.4 核心家族

绝对位置编码通常直接加到 token embedding 上：

```
# token_ids: [batch, seq]
x = tok_embed(token_ids)                    # [batch, seq, d_model]
pos = pos_embed(arange(seq_len))            # [seq, d_model] (learned or sinusoidal)
x = x + pos[None, :, :]                     # broadcast over batch
```

经典正弦位置编码：

$$PE(t,2i) = \sin\left(\frac{t}{10000^{2i/d_{\text{model}}}}\right), \quad PE(t,2i+1) = \cos\left(\frac{t}{10000^{2i/d_{\text{model}}}}\right)$$

相对位置编码在注意力分数上加距离偏置：

```
# scores = Q @ K.transpose(-2, -1) / sqrt(d_k)   # [batch, heads, seq, seq]
scores = scores + relative_bias                  # [1 or batch, heads, seq, seq]
weights = softmax(scores, dim=-1)
out = weights @ V
```

RoPE 对 Q/K 做位置相关旋转：

```
# Q, K: [batch, heads, seq, head_dim]
Q = apply_rope(Q, seq_positions)
K = apply_rope(K, seq_positions)
scores = Q @ K.transpose(-2, -1) / sqrt(head_dim)
```

6.5 常见方法与直觉

方法	公式	编码内容	直觉解释
正弦（绝对位置）	$PE(t)_i = \sin(t \cdot \omega_i), \quad \omega_i = 10000^{-i/d}$	绝对位置索引	每个位置都有独特“波形签名”；多频率组合让模型感知顺序。
可学习绝对位置	$PE(t) = E_t$	绝对位置索引	模型直接学习一张“位置词表”。
相对位置（**Shaw** 等）	$\text{score}_{ij} = Q_i K_j^\top + a_{i-j}$	相对距离 $(i-j)$	模型学习“相隔多远”而不是“绝对在哪”，更符合语言局部依赖。
词元内距离（一般形式）	$PE(i,j) = f(\lvert i-j \rvert)$	任意距离函数	可以定义任意“距离 $\to$ 偏置”映射；ALiBi 与 T5 都可视为特例。
ALiBi（线性注意力偏置）	$\text{score}_{ij} \mathrel{+}= w_h \cdot (i-j)$	线性距离惩罚	对远距离词元施加软惩罚，鼓励关注近邻，也利于长长度泛化。
RoPE（旋转位置编码）	$Q_t^{\text{rot}} = R_t Q_t, \quad K_t^{\text{rot}} = R_t K_t$ $R_t = \begin{bmatrix} \cos(\theta t) & -\sin(\theta t) \\ \sin(\theta t) & \cos(\theta t) \end{bmatrix}$	通过几何旋转编码相对位移	位置让 Q/K 在旋转空间“转角”，相对距离体现在角度差里。
T5 / DeBERTa（相对偏置）	$\text{score}_{ij} = Q_i K_j^\top + b_{\lvert i-j \rvert}$	分桶距离	把距离分桶处理；工程稳定、实现成熟。

6.5.1 长度外推（Length Extrapolation）

长度外推指：推理时上下文长度远超训练长度（例如训练 20k，推理 128k）。很多位置方案在这种情况下会退化，核心是“模型是否见过相近索引/相近距离模式”。

方法	数学核心	为什么有帮助	优点	缺点
RPE（相对位置编码）	$\text{score}_{ij} = Q_i K_j^\top + a(i-j)$	学的是距离，不是绝对索引	外推通常较好；符合语言局部性	需要学习偏置/嵌入；常要分桶
RoPE（旋转位置编码）	$Q_t^{\text{rot}} = R_t Q_t,\ K_t^{\text{rot}} = R_t K_t$	相对位移由角度差体现	平滑、主流、效果稳健	超长上下文常需额外缩放
ALiBi（线性偏置）	$\text{score}_{ij} += w_h(i-j)$	通过距离惩罚引入近因偏置	实现简单；长度泛化好	对部分长程模式建模较弱
位置插值（PI）	$t' = \frac{L}{L_{\text{new}}} \cdot t$	把位置压回训练熟悉范围	极易实现；有时很有效	插值可能扭曲结构
Base-N 位置编码	$t = \sum_k d_k B^k,\quad PE_k(d_k)$	多尺度数字表示	理论上长度上限高	不连续，常需平滑/调参
NTK-aware RoPE 缩放	$\theta' = \theta / s$	全局放慢 RoPE 频率	简单且常有效	可能过度或不足修正
分段 NTK 缩放	$\theta'_i = \theta_i / s \cdot \text{part}(i)$	高频缩放大于低频	比全局缩放失真小	参数更多、调参复杂
Dynamic NTK	$\theta'_i = \theta_i \cdot \frac{m}{\beta^{d/2-1}}$	按维度自适应缩放	调好后长程更稳	实现细节复杂
YaRN	$\sqrt{1/t} = 0.1\ln(s) + 1$（配合 RoPE 频率缩放）	在缩放与稳定间做平衡	长上下文扩展鲁棒	实现略复杂

6.6 直觉速查表

方法	核心直觉
RPE	学“距离”
RoPE	用旋转编码位移
NTK	全局放慢旋转频率
分段 NTK	按频段放慢旋转
Dynamic NTK	按维度自适应缩放
YaRN	更平衡、更稳定的 RoPE 缩放
ALiBi	线性近因偏置（软掩码）
PI	把索引压回训练范围
Base-N	把位置看成多位数字
词元内距离	任意距离映射

6.7 面试题

6.7.1 为什么 Transformer 必须有位置编码？

因为注意力本身不自带顺序感，必须额外注入“谁在前谁在后”。

6.7.2 绝对、相对、RoPE 一句话怎么区分？

绝对：给每个位置一个向量。相对：按距离改分数。RoPE：通过旋转把位置写进 Q/K 几何关系。

6.7.3 什么是长度外推，为什么重要？

就是推理长度超出训练长度。长上下文场景里，它决定模型是否“还能正常工作”。

6.7.4 生产里哪些会预计算，哪些在线算？

Sinusoidal 常预计算；相对偏置可按长度缓存；RoPE 多在线作用于 Q/K。

7 解码（Decoding）

> **Note**
>
> 解码决定“从概率到文本”的最后一步。很多看似模型能力问题，实际是解码参数问题。

解码是把“概率”变成“可读文本”的最后一公里。在每个时间步，模型会给出下一个词元的分布 $p(x_t \mid x_{<t})$，而解码策略决定“到底选哪个词元”。

形式化地说，解码就是把 next-token 分布反复转成实际选择，直到命中终止条件（EOS、stop 序列或 max tokens）。

生产中常见管线是：1. logits 变换（温度、重复惩罚）。2. 候选截断（top-k / top-p）。3. 选择策略（argmax 或采样）。4. 终止条件（EOS、stop 序列、max tokens）。

从工程角度，解码往往比你想象得更“有决定性”：很多看似“模型能力不够”的问题，实际是参数与策略没配好，比如重复、过度保守、格式跑偏、随机性失控。

7.1 贪心搜索（Greedy Search）[code]

$$x_t = \arg\max_i p(i \mid x_{<t})$$

最快、最稳定，但容易模板化和重复。

7.2 束搜索（Beam Search）[code]

$$\text{score}(x_{1:t}) = \sum_{\tau=1}^{t} \log p(x_\tau \mid x_{<\tau}), \quad \text{Beam}_t = \text{TopB}(\text{score}(x_{1:t}))$$

适合追求高似然输出，但开放式对话中常显得保守、缺乏变化。

7.3 Top-k 采样（Top-k Sampling）[code]

$$\tilde{p}(i) \propto p(i)\,\mathbf{1}[i \in \mathrm{TopK}(p)], \quad x_t \sim \tilde{p}$$

固定候选大小，简单有效，但在分布平坦时不够稳。

7.4 Top-p 采样（Nucleus Sampling）[code]

$$\sum_{i \in S} p(i) \geq p$$

$$x_t \sim \tilde{p}(\cdot \mid \cdot \in S)$$

候选集合随分布形状自适应，通常比固定 top-k 更稳。

7.5 温度采样（Temperature Sampling）

$$p_i' = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

$T < 1$ 更确定，$T > 1$ 更多样。

7.6 Top-k/Top-p + temperature（常见组合）

温度控制“随机程度”，top-k/top-p 控制“候选边界”。实际部署中常再配重复惩罚和 stop 序列，保证格式稳定（如 JSON、tool call）。

7.7 Best-of-N（自评分）

$$\{y^{(1)}, y^{(2)}, \ldots, y^{(N)}\}$$

采样 N 条，再用模型或奖励函数选最优。质量常更高，但计算几乎线性增加。

7.8 多数投票 / 自一致性（Self-Consistency）

$$y^{(1)}, y^{(2)}, \ldots, y^{(N)}$$

对多次采样结果做投票/聚合，常用于推理任务，能降低单次采样噪声。

7.8.1 总结

方法	是否确定性	多样性	说明
贪心	是	低	速度快；但可能重复
束搜索	部分	低	优化似然；聊天场景常偏“模板化”
Top-k	否	中	截断简单；分布平坦时较敏感
Top-p	否	高	自适应截断；创作类场景常用
温度	否	可调	通过重缩放 logits 控制随机性
Top-k/Top-p + 温度	否	高	生产中最常见参数组合
Best-of-N	生成阶段否	高	若评分器靠谱，质量通常更好

方法	是否确定性	多样性	说明
自一致性	生成阶段否	很高	推理题常用；成本更高

解码是最容易被低估、但最有杠杆的层。你常看到“模型回答变差”，根因不是权重退化，而是解码参数不匹配。

小例子：客服机器人如果温度设得太高，回答会像“深夜加班后语速过快”一样飘；温度太低，又会像“只会读模板”的机器人。

8 架构（Architecture）

架构设计回答的是：同样是 Transformer，为什么有的适合分类，有的适合对话，有的适合翻译，还有的更适合超大规模推理。

常见失败模式：

- 训练成本爆炸（深度/宽度/注意力选择不合理）。
- 推理受限（KV cache 过大，带宽吃紧）。
- 任务不匹配（目标函数和掩码模式错配）。

更具体一点，架构其实在回答四个系统问题：
1. 训练目标是什么（MLM、next-token、seq2seq）。
2. 注意力掩码怎么设计（双向、因果、混合）。
3. 基础模块怎么堆（self-attention + FFN，堆几层、多宽）。
4. 是否引入额外机制（cross-attention、MoE、长上下文技巧）。

高层上，架构要定四件事：1. 训练目标（MLM、next-token、seq2seq）。2. 掩码模式（双向、因果、混合）。3. 块结构（注意力 + FFN 堆叠）。4. 额外机制（cross-attention、MoE、长上下文策略）。

8.1 仅编码器（Encoder-only, 双向）

$$\text{Attention}(x_i) = f(x_{<i}, x_i, x_{>i})$$

双向建模，擅长表示学习任务（分类、检索向量等），典型预训练目标是 MLM。代表：BERT、RoBERTa、DeBERTa。

8.2 仅解码器（Decoder-only, 自回归）

$$p(x_t \mid x_{<t})$$

因果建模，天然适合生成。当前主流聊天模型几乎都在这一范式。代表：GPT、LLaMA、Mistral、OPT、BLOOM。

8.3 编码器-解码器（Encoder–decoder, Seq2Seq）

Encoder:

$$h = \mathrm{Enc}(x_{1:n}), \quad \text{bi-directional}$$

Decoder:

$$p(y_t \mid y_{<t}, h), \quad \text{auto-regressive}$$

天然适合翻译、摘要这类“输入序列 -> 输出序列”任务。代表：T5、BART。

8.4 PrefixLM（部分因果掩码）

$$M_{ij} = \begin{cases} 0, & j \le P \quad \text{(prefix, bi-directional)} \\ 0, & j < i \quad \text{(suffix, causal)} \\ -\infty, & \text{otherwise} \end{cases}$$

前缀双向、后缀因果，是一种折中式设计。

8.5 专家混合（MoE, Mixture of Experts）

$$\mathrm{FFN}(x) = \sum_{k=1}^{N} g_k(x)\, E_k(x)$$

词元只路由到少量专家，达到“参数大、每步计算相对可控”的效果。

小例子：把 MoE 想成“医院分诊台”。每个病人（词元）不需要看所有科室（专家），先分诊到 1~2 个最相关科室，效率和质量都更容易兼顾。

SoftMoE 形式：

$$\mathrm{SoftMoE}(x) = W\Big(\sum_{k} g_k(x) E_k(x)\Big)$$

路由质量决定实际效果：负载均衡差会拖慢吞吐，容量溢出会伤害质量。

8.6 总结

架构	注意力样式	训练目标	优势	代表模型
Encoder-only	双向	MLM	表示提取强	BERT
Decoder-only	因果	NLL	生成与提示学习强	GPT, LLaMA
Encoder–decoder	双向 + 因果	Seq2Seq	翻译、摘要效果好	T5, BART
PrefixLM	混合掩码	Prefix-style LM	推理与对话折中	GLM
MoE	稀疏路由专家	Sparse	高效扩展参数规模	DeepSeek, Mixtral

8.7 面试题

8.7.1 为什么聊天模型大多是 decoder-only？

因为聊天本质是连续生成，decoder-only 的目标函数和推理方式天然一致。

8.7.2 什么时候选 encoder-only，什么时候选 encoder-decoder？

要表示学习选 encoder-only；要高质量 seq2seq 生成选 encoder-decoder。

8.7.3 一句话解释 MoE，为什么路由关键？

MoE 是“多专家 + 稀疏路由”；路由决定负载、容量和最终质量。

8.7.4 推理内存最常由什么主导？

decoder-only 里通常是 KV cache。

8.8 面试题

8.8.1 解码在工程流程里到底做了什么？

logits 变换 -> 候选过滤 -> 词元选择 -> 终止控制，循环执行。

8.8.2 为什么 beam search 在开放生成里常显得“同质化”？

它偏向高似然路径，容易收敛到“安全模板”。

8.8.3 Top-k 和 top-p 区别？哪个更稳？

top-k 固定候选数量，top-p 固定累计概率。top-p 通常跨 prompt 更稳。

8.8.4 温度极低/极高会怎样？

极低：近似贪心，容易无聊重复；极高：随机过头，易失真和跑偏。

8.8.5 重复惩罚/no-repeat n-gram 会带来什么副作用？

抑制循环有效，但可能误伤“合法重复结构”（列表、代码模板、JSON 键等）。

8.8.6 何时用 best-of-N / self-consistency？代价是什么？

追求可靠性时用，代价是近似 N 倍生成成本加额外评分成本。

8.8.7 什么是 stop sequence，为什么重要？

它是“命中即停”的结束标记，能保证结构化输出边界干净可解析。

9 超越 Transformer（Beyond Transformers）

Transformer 仍是今天的主流答案，但它有一个明确的扩展瓶颈：标准注意力在长上下文下成本很高。

生产里最常见的两个压力点是：

- **Prefill** 计算：随序列长度 L 近似按 $O(L^2)$ 增长，常主导首 token 延迟（TTFT）。
- **Decode** 内存/带宽：自回归生成依赖 KV cache，随着上下文和层数增长，内存与带宽压力迅速上升。

“超越 Transformer” 并不只指“换模型架构”，也包括推理期策略变化。核心是改变“序列信息如何被处理”。有些方法改网络本体（如状态空间模型），有些保持 Transformer 主干但改系统编排（如递归、检索、分块）。

一个实用心智图是：
- **Transformer（decoder-only）**：标准栈，质量强、生态成熟，但注意力与 KV 成本高。
- **SSM / 状态空间（如 Mamba）**：用状态更新替代 softmax 注意力，长序列效率更好，但归纳偏置不同。
- **Retention / 混合（如 RetNet）**：尝试在递归特性与并行训练之间折中。
- 循环混合（如 **RWKV**）：训练像 Transformer、推理更像 RNN，弱化对 KV cache 的依赖。
- 扩散语言模型：通过迭代细化生成，常有更强可控性，但步骤更多。
- 递归式 **LM**（推理期）：底模不变，通过检索/分块/汇总/循环等系统手段处理超长任务。

模型家族	生成方式	主要收益	主要代价	常见场景
Transformer (decoder-only)	自回归（逐词元）	质量强、生态成熟	注意力/KV cache 成本高	聊天 LLM 默认方案
SSM / 状态空间（如 Mamba）	自回归	序列处理近线性	归纳偏置不同；生态较小	长序列效率研究与部分生产
Retention / 混合（如 RetNet）	自回归	兼顾递归与并行训练	工程标准化不足	研究与特定部署
循环混合（如 **RWKV**）	自回归	无 KV cache，推理更像 RNN	主流生态相对弱	边缘端与效率导向场景
扩散语言模型	迭代细化	可控性更强、并行潜力更高	去噪步骤多，基础设施不同	新兴研究与少量垂直场景
递归式 **LM**（推理期）	自回归 + 系统编排	处理超上下文窗口任务	编排复杂，不只是模型问题	Agent 与长文档工作流

实践中通常不需要背公式，但必须说清两点：1. 它解决了哪个瓶颈。2. 它引入了哪些新的成本或风险。

10 总结

本章建立了现代 LLM 的工作全景：分词把文本变成离散符号，嵌入把符号变成可计算向量，位置编码补齐顺序信息，Transformer 块通过注意力在词元间路由信息、通过 FFN 在词元内做非线性计算，最后由解码把 logits 变成可读文本。

工程上最关键的不是单点技巧，而是全链路权衡，尤其是“规模化”带来的连锁影响：

- 分词与词表影响覆盖、长度与成本。
- 注意力在长上下文下成本陡增，prefill 常近似 $O(L^2)$，KV cache 常主导推理内存/带宽。
- 残差与归一化决定深层训练稳定性。
- 位置编码决定顺序建模与长度外推表现。
- 解码策略决定输出风格、可靠性与延迟成本。
- 架构（encoder-only / decoder-only / encoder-decoder / MoE）要与任务目标和硬件约束共同设计。

一句话收尾：LLM 不只是“更大的网络”，而是一套从符号到生成、从算法到系统的协同工程。