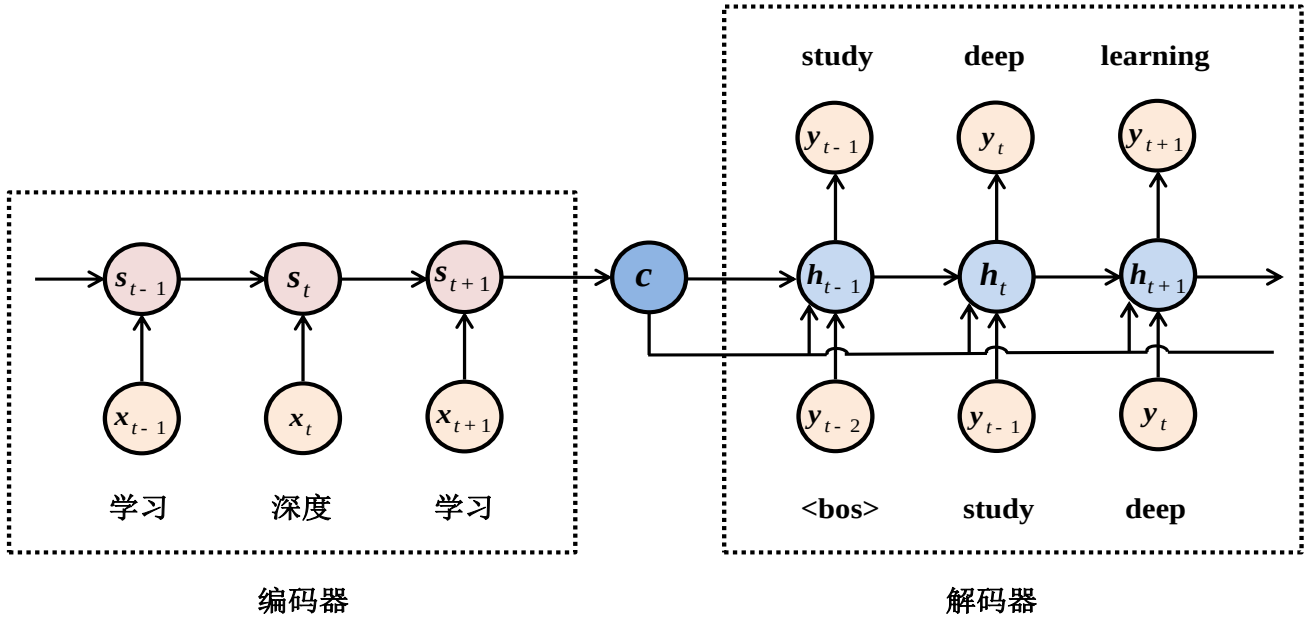
第八讲 注意力与Transformer

《深度学习原理》

南开大学 人工智能学院

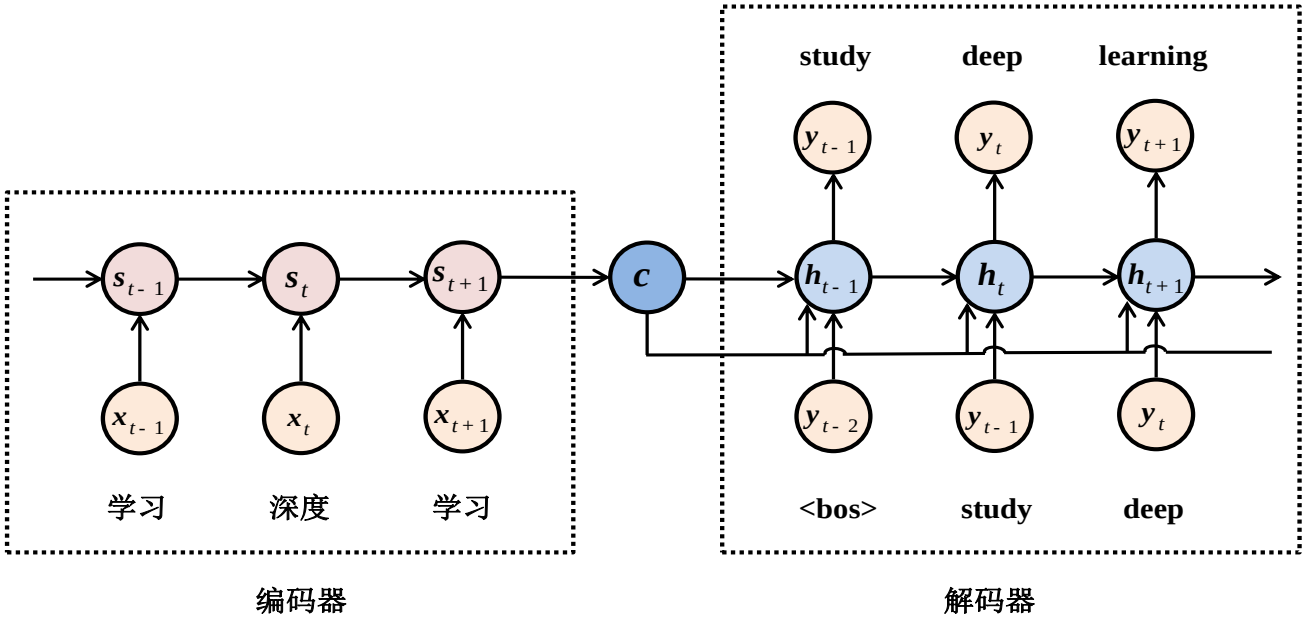
回顾：编码器—解码器框架

- 编码器—解码器框架的不足
  - 编码器RNN输出的 C 的维度太小而难以适当地概括一个长序列
    - 常将最后一层隐状态输出作为C
    - 源句子中越在前面的单词，对形成 C 的影响越小；如果源句子过长，那么源句子前面单词的语义信息几乎没有被编码到 C 中，因而解码器就很难使用前面单词的语义



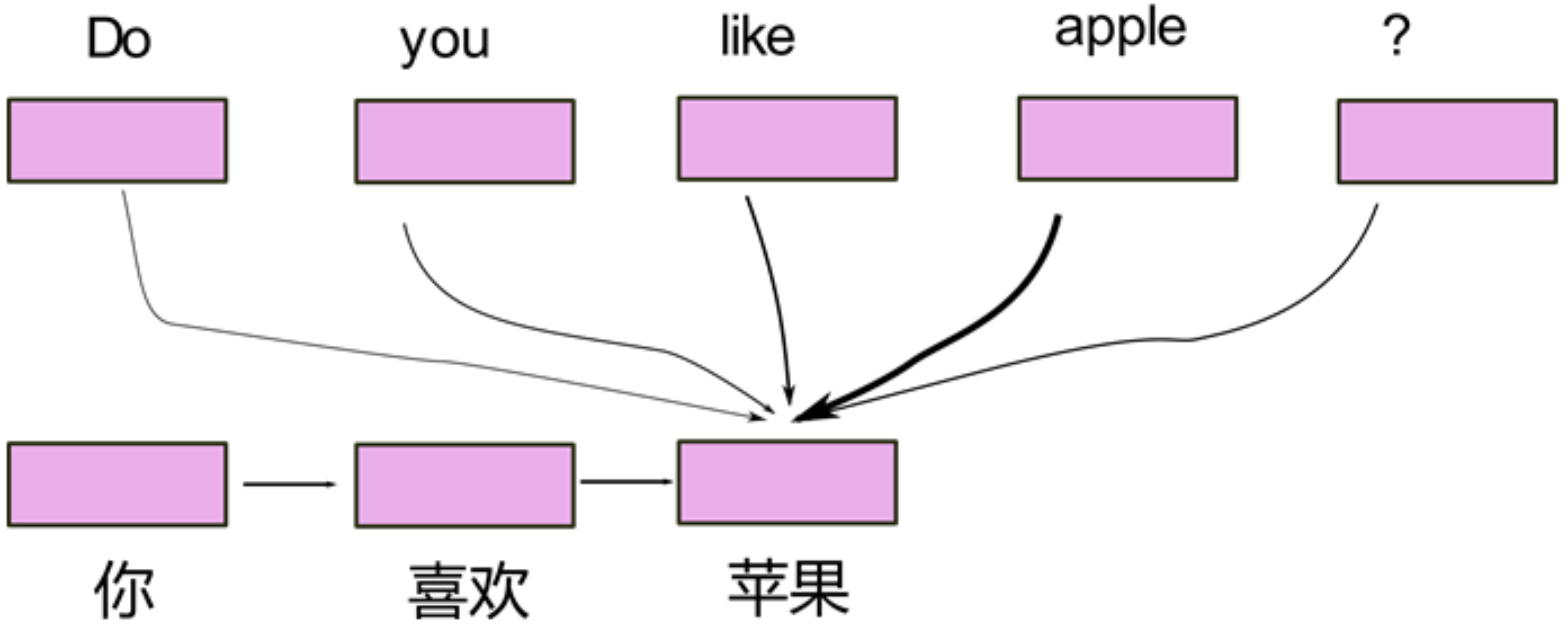
回顾：编码器—解码器框架

- 编码器—解码器框架的不足
  - 并非所有输入词元都对解码某个词元有用，但在每个解码步骤中仍使用相同的上下文变量 C
    - 当解码器生成目标句子中的每一个词时，都使用同一个C；这意味着不管生成哪个词，都按照同一信息量使用源句子中的每个词，这显然不合理



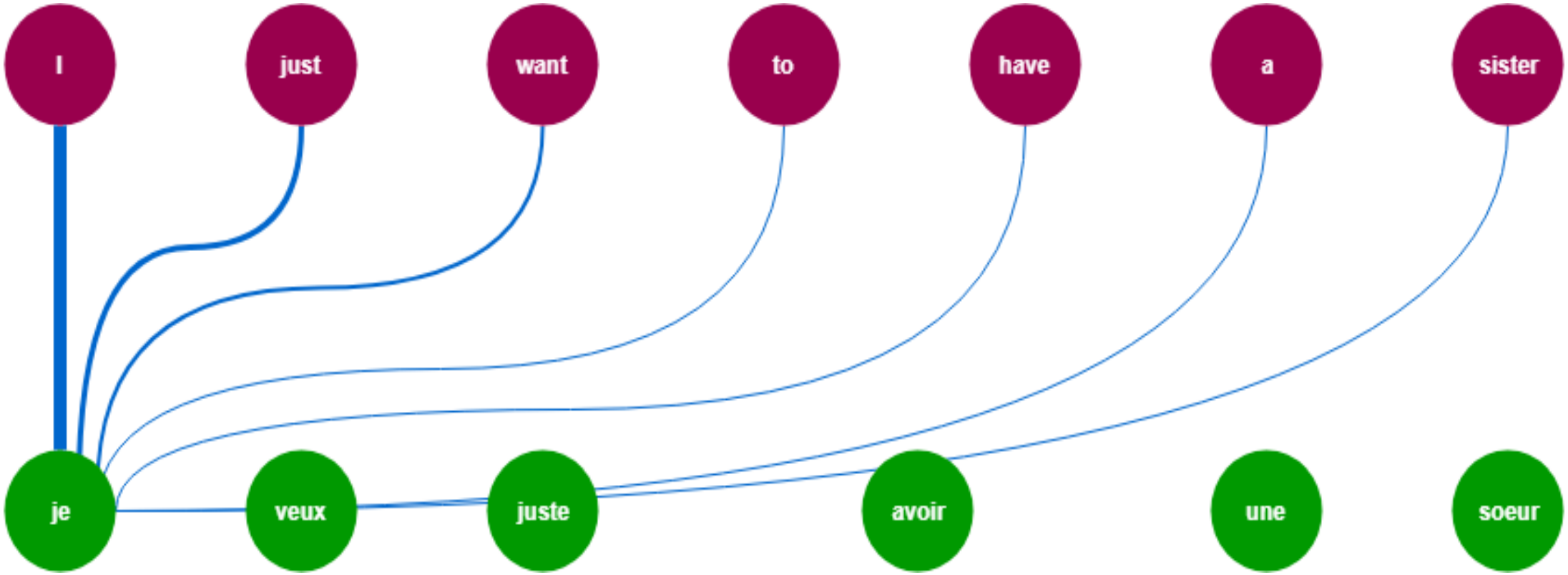
回顾：编码器—解码器框架

- 解决方案：
  - 注意力机制：解码不同词元时，把注意力集中在更相关的输入词元上



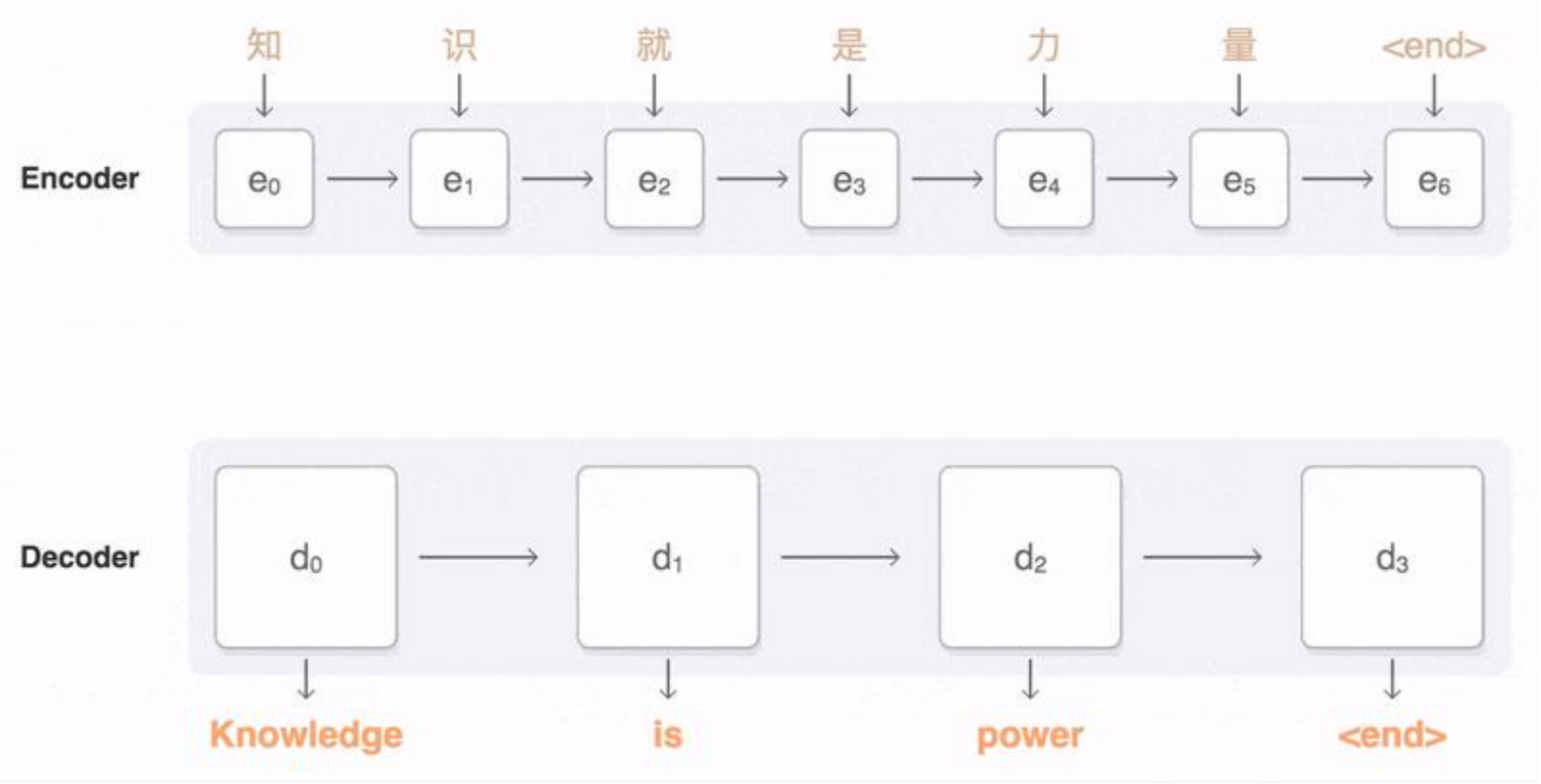
回顾：编码器—解码器框架

- 解决方案：
  - 注意力机制：解码不同词元时，把注意力集中在更相关的输入词元上



回顾：编码器—解码器框架

- 解决方案：
  - 注意力机制：解码不同词元时，把注意力集中在更相关的输入词元上



8.1 示例：Nadaraya-Watson 核回归

- 问题描述：给定成对的“输入－输出”数据集 $\{(x_1, y_1), \cdots, (x_n, y_n)\}$，如何学习 f 来预测任意一个新输入 x 的输出 $y = f(x)$？
- 思想：将注意力关注在与 x 相近的 x_i 上，通过对应的 y_i 预测 $y = f(x)$
- 模型：根据输入对输出加权

$$f(x) = \sum_{i=1}^{n} \underbrace{\frac{K(x - x_i)}{\sum_{j=1}^{n} K(x - x_j)}}_{\text{权重}} y_i$$

  - 其中 $x - x_i$ 越接近0，$K(x - x_i) > 0$ 越大，权重越大
  - 如果一个 x_i 越是接近给定的查询 x，那么分配给 y_i 的“**注意力权重**”就会越大，也就“**获得了更多的注意力**”

8.2 注意力机制组件

- 注意力：是指序列中两个元素之间的**相关性**程度
  - 可将注意力理解为权重，谁越相关，越关注谁，权重越大
  - 允许神经网络在处理输入数据时集中注意力于关键信息，忽略不相关信息

8.2 注意力机制组件

概念

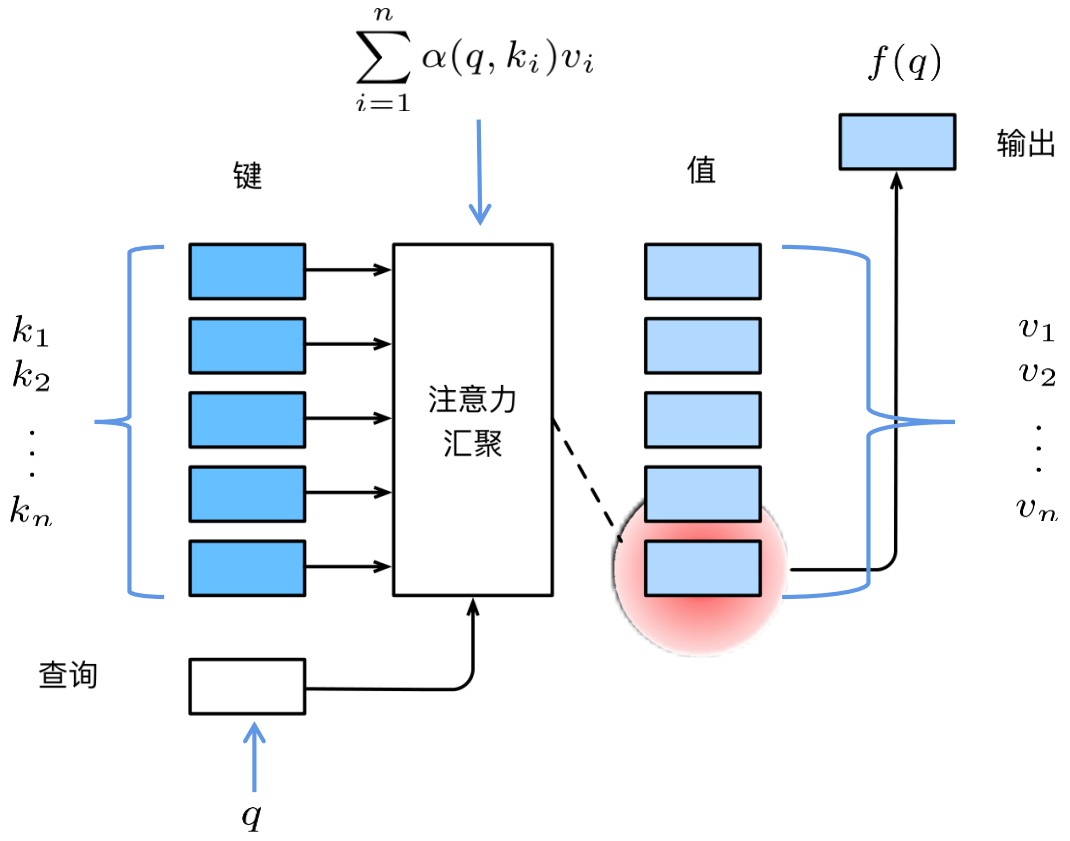
- 注意力机制组件
  - （键key，值value）对
    - 键可以理解为特征向量
    - 值可理解为信息的载体
  - 查询（query）
  - 注意力汇聚
- 注意力汇聚：对值 v_i加权平均，其中权重 $\alpha(q, k_i)$ 是在给定的查询q和不同的键 k_i之间计算得出,度量二者相似度，越相关，权重越大
  - 不同的v_i赋予不同的注意力权重，重要的v_i赋予更高的权重
  - 注意力权重满足概率分布要求：非负，和为1

类比：图书馆检索系统

Query：查询请求

Key： 每本书的索引标签或书名关键词

Value： 书本的实际内容



8.2.1 注意力汇聚

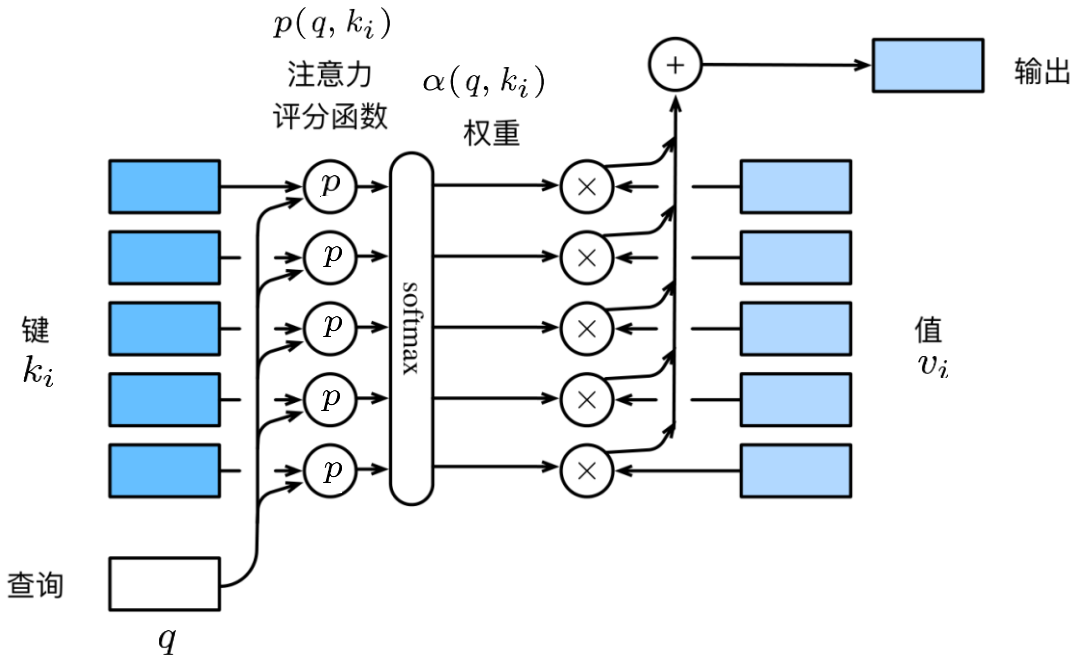
计算

- 注意力汇聚
  - 根据查询 q和键 $k_i, i = 1, \cdots, n$ ，计算一个**评分函数** $p(q, k_i)$
  - 把评分函数的结果输入到softmax函数，得到对应值 v_i的**注意力权重**（或者称为**注意力分数**）

$$\alpha(q, k_i) = \text{softmax}(p(q, k_i))$$

    - softmax的计算结果和为1，满足概率分布要求
  - 最后，注意力汇聚的输出就是基于这些注意力权重对值（value）做加权求和

$$f(q, (k_i, v_i), i = 1, \cdots, m) = \sum_{i=1}^{m} \alpha(q, k_i) v_i$$



8.2.2 缩放点积注意力

- **评分函数：缩放点积注意力**
  - 当查询 q 和键 k 具有相同的长度 d 时，可以使用**缩放点积**作为评分函数

点积 → $q^T k$，缩放 → $\sqrt{d}$

$$p(q,k) = \frac{q^T k}{\sqrt{d}} = \frac{\sum_{i=1}^{d} q_i k_i}{\sqrt{d}}$$

  - 缩放点积注意力用于Transformer中
  - 此时的注意力汇聚输出为

$$f(q,(k_i,v_i)) = \sum_{i=1}^{n} \mathrm{softmax}\left(\frac{q^T k_i}{\sqrt{d}}\right) v_i$$

8.2.2 缩放点积注意力

- 缩放点积中为什么除以$\sqrt{d}$？
  - 当q和k的维度较高时，计算出的点积较大，输入Softmax函数时，指数函数的放大效应会导致某些位置的权重趋近于1，其他位置的权重趋近于0

$$\hat{y} = \text{Softmax}(x)$$

    - 现在假设第一个元素x_1很大，由于$\hat{y}_1 = \frac{\exp(x_1)}{\sum_{j=1}^{d}\exp(x_j)}$，指数函数会将$x_1$进一步放大，Softmax会将绝大部分概率分给$\hat{y}_1$，即$\hat{y}_1 \approx 1, \hat{y}_j \approx 0, j \neq 1$，使得注意力权重过于集中在某个输入上

不考，但面试会面

  - 此时会使得softmax的梯度过小，出现**梯度消失**现象

$$\frac{\partial \hat{y}}{\partial x} = \text{diag}(\hat{y}) - \hat{y}\hat{y}^T$$

$$= \begin{bmatrix} \hat{y}_1 & 0 & \cdots & 0 \\ 0 & \hat{y}_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \hat{y}_d \end{bmatrix} - \begin{bmatrix} \hat{y}_1^2 & \hat{y}_1\hat{y}_2 & \cdots & \hat{y}_1\hat{y}_d \\ \hat{y}_2\hat{y}_1 & \hat{y}_2^2 & \cdots & \hat{y}_2\hat{y}_d \\ \vdots & \vdots & \ddots & \vdots \\ \hat{y}_d\hat{y}_1 & \hat{y}_d\hat{y}_2 & \cdots & \hat{y}_d^2 \end{bmatrix} \approx \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} = 0$$

8.2.2 缩放点积注意力

- 缩放点积中为什么除以$\sqrt{d}$？
  - 缩放之后，每个x_j都在1附近，与维度无关，缓解了指数函数的放大效应，每个$\hat{y}_j$都能分到一些概率
    - 避免注意力权重过于集中，让每个输入都能得到一些关注
    - 缓解softmax的梯度趋于0的问题，使训练更加稳定
  - 示例：

$$
\begin{aligned}
&x = [0.1, 0.1, 3] \\
&\exp(x) = [1.1052, 1.1052, 20.0855] \\
&\text{softmax}(y) = [0.0496, 0.0496, 0.9009]
\end{aligned}
$$

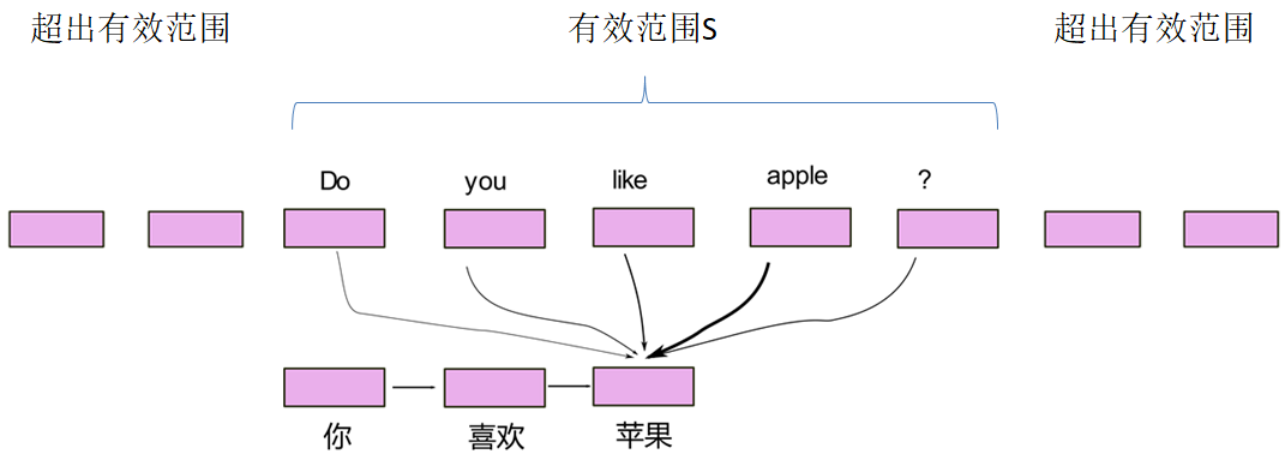
无缩放

$$
\begin{aligned}
&x = [0.1, 0.1, 3] \\
&y = x/\text{sqrt}(3) = [0.0577, 0.0577, 1.7321] \\
&\exp(y) = [1.0594, 1.0594, 5.6522] \\
&\text{softmax}(y) = [0.1363, 0.1363, 0.7273]
\end{aligned}
$$

有缩放

8.2.3 掩蔽softmax

- 有时我们并不想考虑所有键值对，而只希望考虑一定范围内的键值对
- 指定一个有效序列范围，计算softmax时掩蔽掉超出指定范围的位置



- 超出指定范围的键值对的注意力权重为0，即被掩蔽

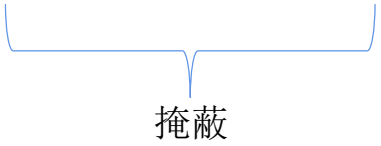
$$\text{softmax}(p_i) = \begin{cases} \frac{\exp(p_i)}{\sum_{j \in \mathcal{S}} \exp(p_j)} & i \in \mathcal{S}, \text{that is, i is not masked} \\ 0 & i \notin \mathcal{S}, \text{that is, i is masked} \end{cases}$$

实现时令$\mathcal{S}$之外的$p_j = -\inf$

那么$\exp(p_j) = 0$，从而避免if判断，加快计算

8.2.3 掩蔽softmax

- 在Transformer解码器中的掩蔽注意力中使用
  - 避免模型偷看后面的正确答案，见8.7.3节
- 处理变长序列时，需要将输入填充到固定长度，计算注意力时对填充部分掩码，确保模型只关注有效的输入数据
  - 我爱南开 -> 我爱南开<pad><pad><pad>



8.3 自注意力

- 自注意力：查询、键和值来自同一组输入，多用于文本数据
- 形式化描述
  - 给定一个由词元组成的输入序列 $x_1, x_2, \cdots, x_n$
  - 计算 $q_i = W^q x_i, k_i = W^k x_i, v_i = W^v x_i$
  - 该序列的自注意力输出是一个长度相同的序列 $y_1, y_2, \cdots, y_n$

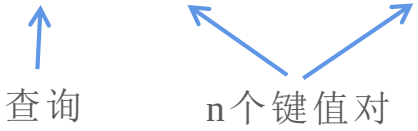
$$y_i = f\left(q_i, (k_1, v_1), \cdots, (k_n, v_n)\right) = \sum_{j=1}^{n} \mathrm{softmax}\left(\frac{q_i^T k_j}{\sqrt{d}}\right) v_j$$

查询　　n个键值对

直观解释：每个词 i 可以同时关注句子中的其他所有词元，计算注意力权重，表示序列中每个词元对当前词元 i 的相关性大小，加权求和，得到该词的上下文向量表示。注意力权重度量了序列中不同词元对当前词元的关联程度

8.3 自注意力

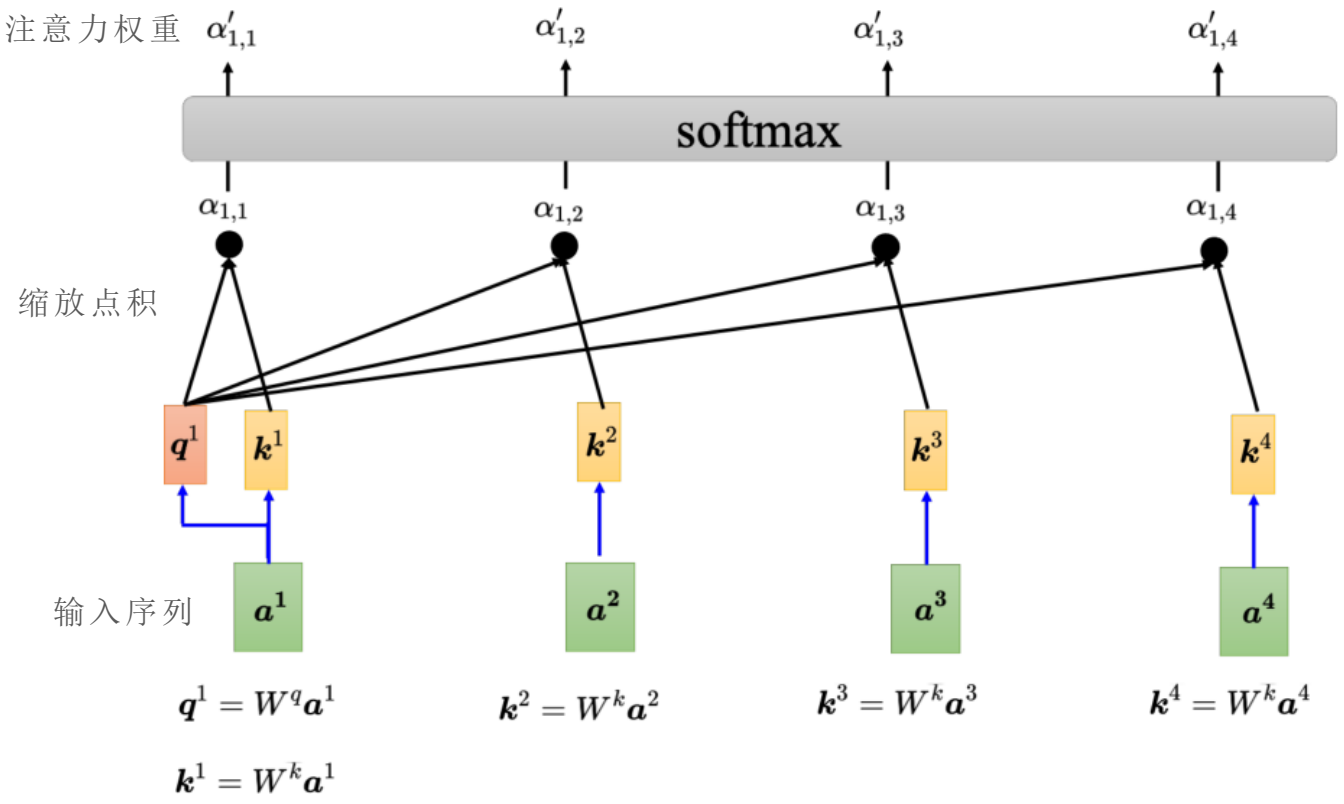
- 自注意力：查询、键和值来自同一组输入
- 形式化描述 $y_i = f\left(q_i, (k_1, v_1), \cdots, (k_n, v_n)\right)$



- 劣势：复杂度高，$O(n^2)$
  - 每一个词元的输出都是其他所有词元的值的加权和，计算量为 $O(n)$； n个词元的输出的计算量为 $O(n^2)$
  - 当序列较长时，开销较大。通常在硬件上加速进行缓解，如FlashAttention
- 优势：可以**并行计算**
  - 注意力汇聚，$\sum_{j=1}^{n} \alpha(q_i, k_j) v_j$ 对不同的查询 q_i 可以并行计算

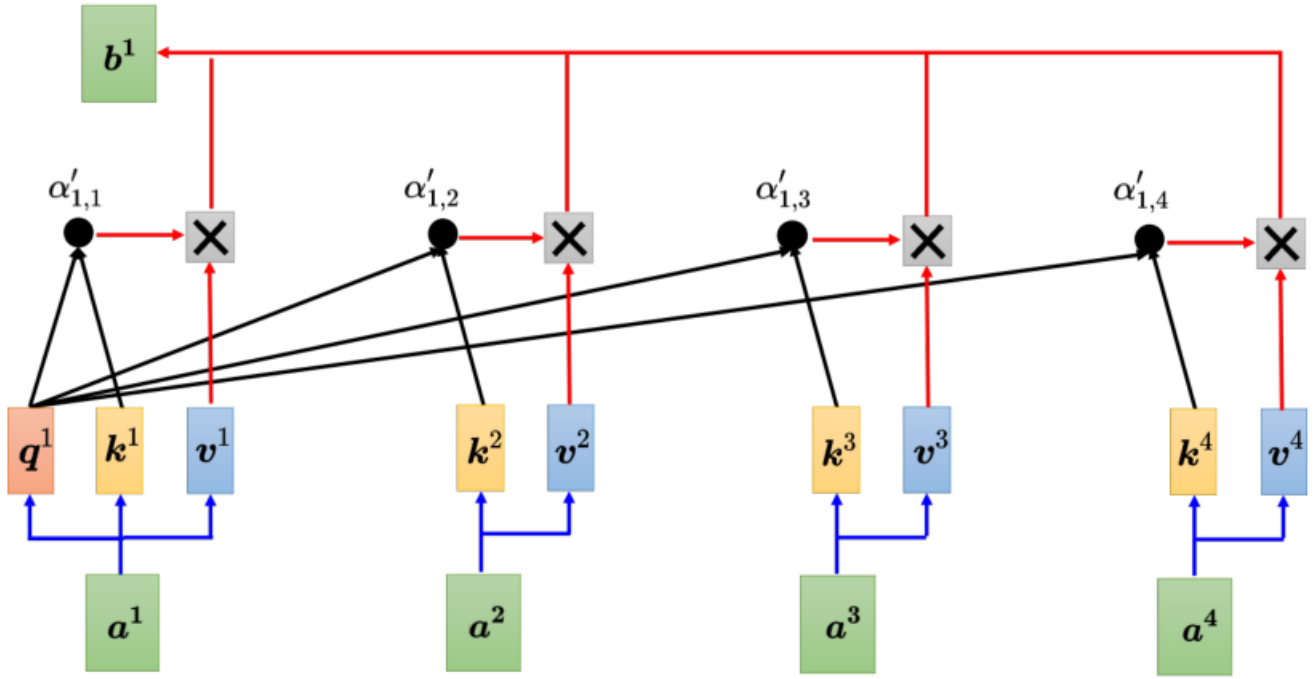
8.3.1 自注意力计算过程

- 计算查询和键之间的注意力权重



8.3.1 自注意力计算过程

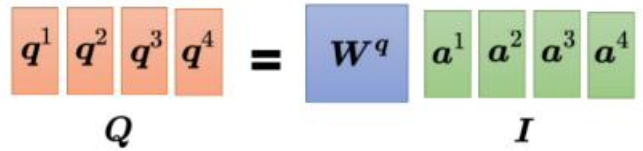
- 对值加权求和



8.3.1 自注意力计算过程

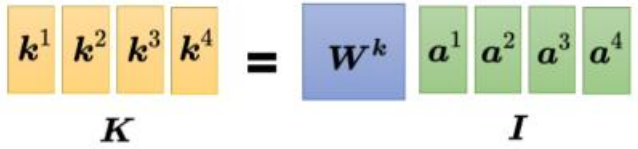
- 矩阵乘法形式
  - 通过线性变化得到查询、键、值

$$q^i = W^q a^i$$



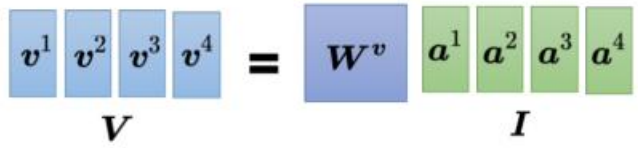
$$Q = W^q I$$

$$k^i = W^k a^i$$

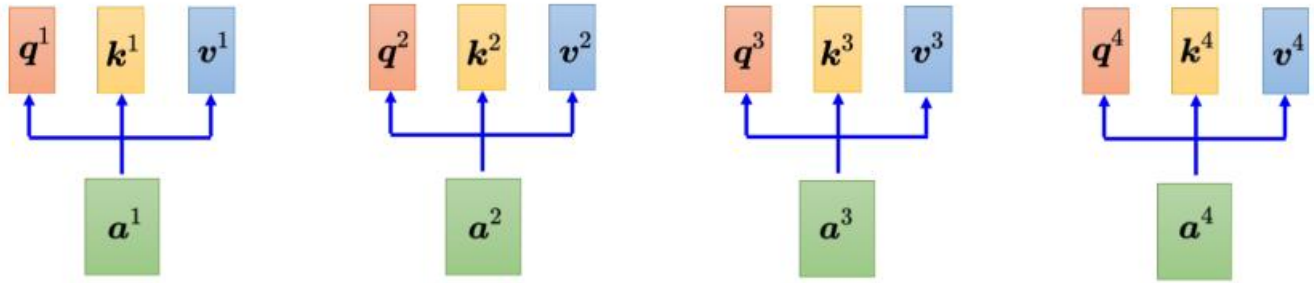


$$K = W^k I$$

$$v^i = W^v a^i$$

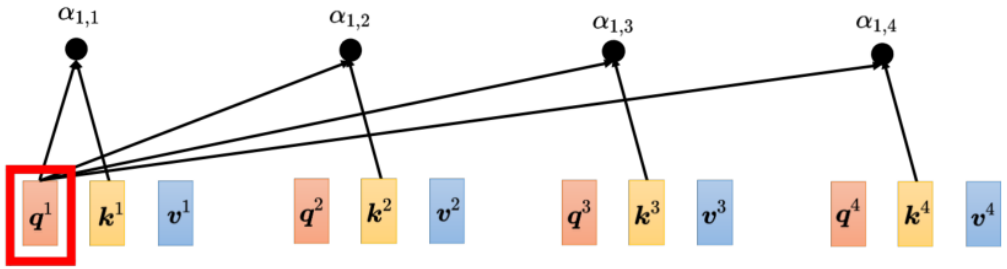


$$V = W^v I$$



8.3.1 自注意力计算过程

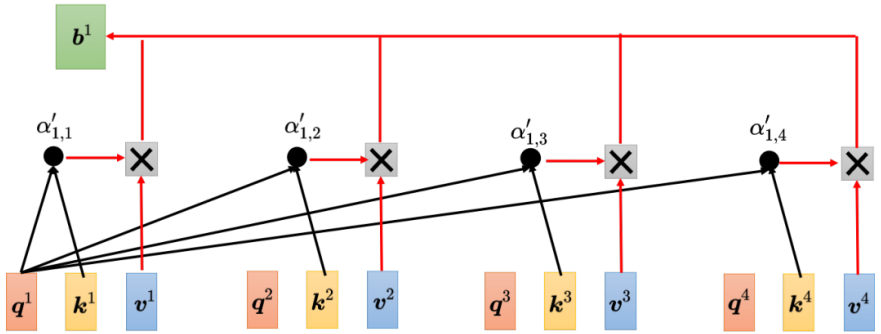
- 矩阵乘法形式
  - 计算注意力权重，加权求和



$$\alpha_{1,1} = (k^1)^{\mathrm{T}} q^1 \quad \alpha_{1,2} = (k^2)^{\mathrm{T}} q^1 \quad \alpha_{1,3} = (k^3)^{\mathrm{T}} q^1 \quad \alpha_{1,4} = (k^4)^{\mathrm{T}} q^1$$

$$\begin{bmatrix} \alpha_{1,1} \\ \alpha_{1,2} \\ \alpha_{1,3} \\ \alpha_{1,4} \end{bmatrix} = \begin{bmatrix} (k^1)^{\mathrm{T}} \\ (k^2)^{\mathrm{T}} \\ (k^3)^{\mathrm{T}} \\ (k^4)^{\mathrm{T}} \end{bmatrix} q^1$$

K^{T}



$$\begin{bmatrix} b^1 & b^2 & b^3 & b^4 \end{bmatrix} = \begin{bmatrix} v^1 & v^2 & v^3 & v^4 \end{bmatrix} \begin{bmatrix} \alpha'_{1,1} & \alpha'_{2,1} & \alpha'_{3,1} & \alpha'_{4,1} \\ \alpha'_{1,2} & \alpha'_{2,2} & \alpha'_{3,2} & \alpha'_{4,2} \\ \alpha'_{1,3} & \alpha'_{2,3} & \alpha'_{3,3} & \alpha'_{4,3} \\ \alpha'_{1,4} & \alpha'_{2,4} & \alpha'_{3,4} & \alpha'_{4,4} \end{bmatrix}$$

O V A'

$$\begin{bmatrix} \alpha'_{1,1} & \alpha'_{2,1} & \alpha'_{3,1} & \alpha'_{4,1} \\ \alpha'_{1,2} & \alpha'_{2,2} & \alpha'_{3,2} & \alpha'_{4,2} \\ \alpha'_{1,3} & \alpha'_{2,3} & \alpha'_{3,3} & \alpha'_{4,3} \\ \alpha'_{1,4} & \alpha'_{2,4} & \alpha'_{3,4} & \alpha'_{4,4} \end{bmatrix} \xleftarrow{\text{softmax}} \begin{bmatrix} \alpha_{1,1} & \alpha_{2,1} & \alpha_{3,1} & \alpha_{4,1} \\ \alpha_{1,2} & \alpha_{2,2} & \alpha_{3,2} & \alpha_{4,2} \\ \alpha_{1,3} & \alpha_{2,3} & \alpha_{3,3} & \alpha_{4,3} \\ \alpha_{1,4} & \alpha_{2,4} & \alpha_{3,4} & \alpha_{4,4} \end{bmatrix} = \begin{bmatrix} (k^1)^{\mathrm{T}} \\ (k^2)^{\mathrm{T}} \\ (k^3)^{\mathrm{T}} \\ (k^4)^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} q^1 & q^2 & q^3 & q^4 \end{bmatrix}$$

A' softmax A K^{T} Q

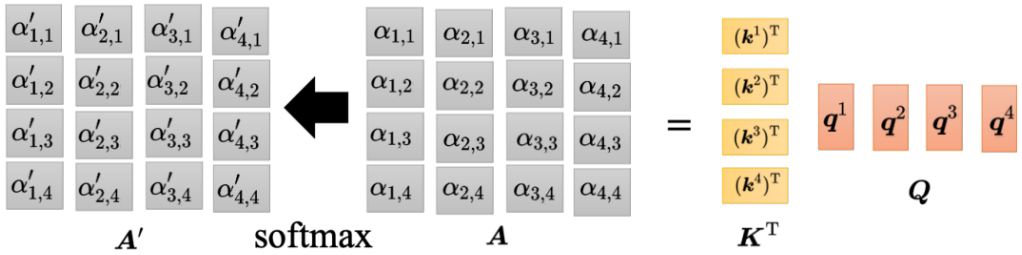
对每一列做softmax

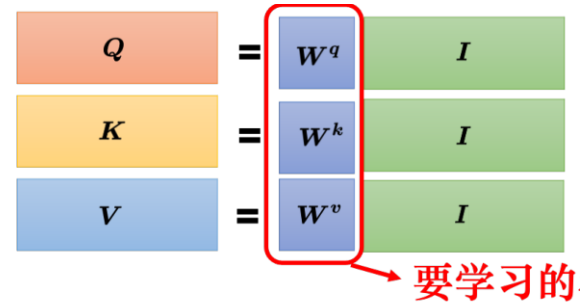
8.3.1 自注意力计算过程

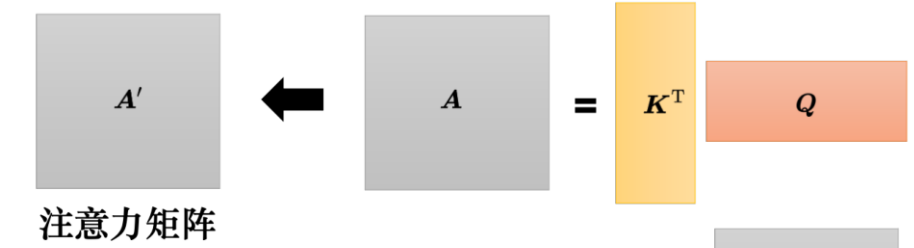
- 矩阵乘法形式

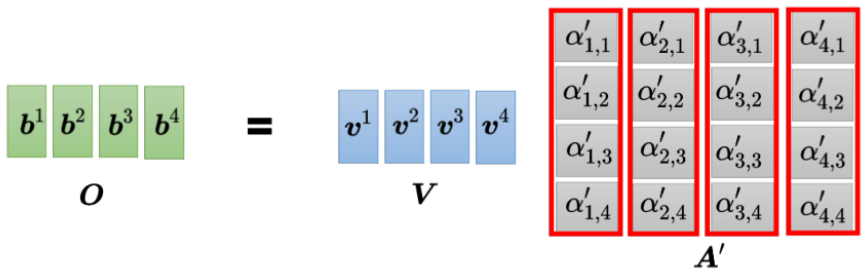
$$q^i = W^q a^i \qquad k^i = W^k a^i \qquad v^i = W^v a^i$$

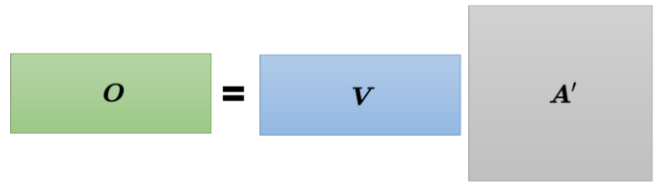
$$Q = W^q I$$
$$K = W^k I$$
$$V = W^v I$$
$$O = V \text{softmax}\left(\frac{K^T Q}{\sqrt{d}}\right)$$







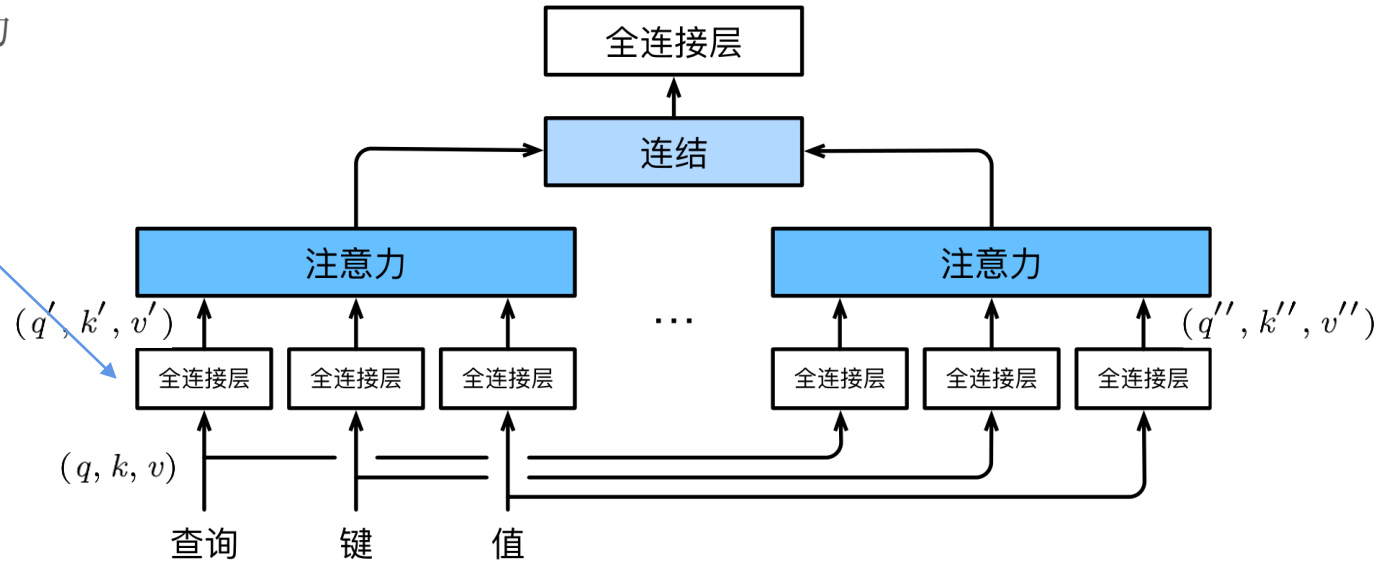




8.4 多头注意力

- 动机：当给定查询、键和值时，我们希望模型可以从不同角度学习到不同的行为和知识，然后将不同的行为和知识组合起来
- 多头注意力
  - **不同的头从不同角度挖掘序列中的信息，学习不同的知识**
    - 例如处理自然语言时，一个头可能专注于语法结构，另一个头可能关注语义关系
    - **多样性**使得模型能够从多个角度理解输入，

      增强模型捕捉多种复杂依赖关系的能力
  - 将查询、键和值通过全连接层做线性变化，并行地送入多个注意力头，将这多个注意力头的输出拼接在一起，通过一个全连接层产生最终输出

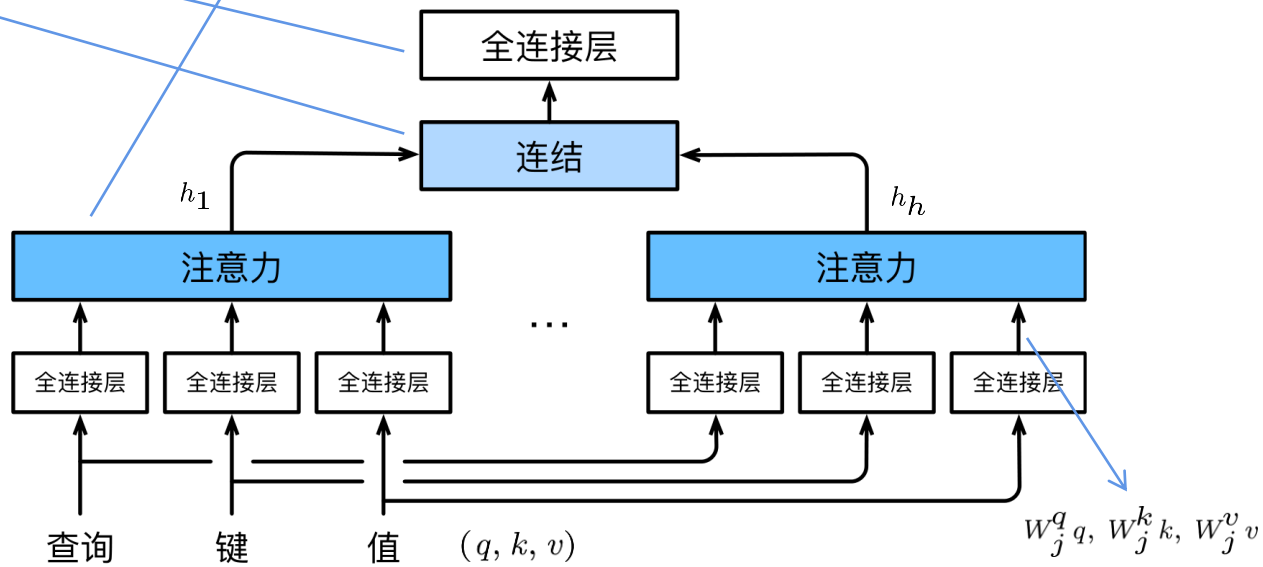


8.4 多头注意力

- 形式化描述
  - 每个**注意力头**：$h_j = f\left(W_j^q q, W_j^k k, W_j^v v\right)$，注意力汇聚函数为 $f(q,k,v) = \sum_{i=1}^{n} \text{softmax}\left(\frac{q^T k_i}{\sqrt{d}}\right) v_i$
  - 不同的头使用相同的注意力机制，故需要对查询、键和值做线性变换，使得不同的头输入不一样
  - 结果连接及全连接层输出 $W_o \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_h \end{bmatrix}$
  - 线性变换矩阵 W_j^q, W_j^k, W_j^v, W_o 通过学习得到

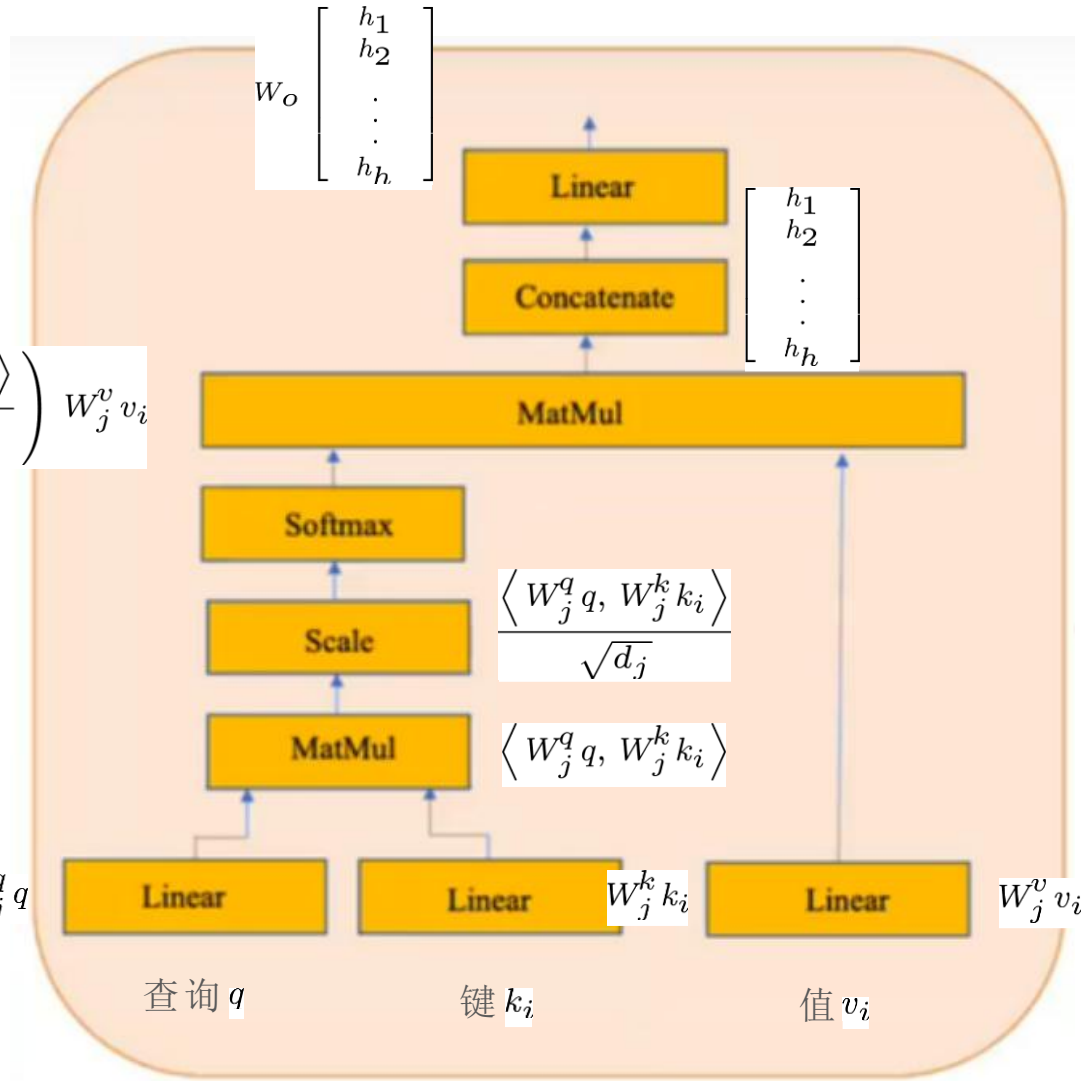
    注意力机制中需要学习的参数
  - 如果是**多头自注意力**，将输入词元的向量表征与不用的 W_j^q, W_j^k, W_j^v 相乘，得到不同的 q_j, k_j, v_j 并行地送入多个注意力头，$h_j = f\left(W_j^q x, W_j^k x, W_j^v x\right)$ 将这多个注意力头的输出拼接在一起，通过一个全连接层产生最终输出

通过为不同的头学习不同的权重矩阵，使得不同的头从不同的角度学习上下文信息



8.4 多头注意力

$$h_j = \sum_{i=1}^{d_j} \text{softmax}\left(\frac{\left\langle W_j^q q, W_j^k k_i \right\rangle}{\sqrt{d_j}}\right) W_j^v v_i$$



第 j 个头

为了控制计算量，一般在线性变换时会降低qkv的维度，例如从d维降到d/h维，其中h为头数，拼接时再恢复到d维

8.5 位置编码

为什么使用
怎么用的

- 自注意力计算中没有考虑序列中词元的位置信息，即不同词元的前后顺序关系
  - 回顾：

$$y_i = f\left(q_i, (k_1, v_1), \cdots, (k_n, v_n)\right) = \sum_{j=1}^{n} \mathrm{softmax}\left(\frac{q_i^T k_j}{\sqrt{d}}\right) v_j$$

  - 处理词元序列时，循环神经网络按顺序逐个处理词元
  - **注意力因为并行计算放弃了顺序操作，即丢掉了不同词元的前后位置信息，将每个词元视为彼此独立的**
- 为了使用序列的顺序信息，可以在输入表示中添加位置编码来注入绝对的或相对的位置信息，即词元在序列中的位置
  - 位置编码：对位置信息进行编码
  - 好处：**既能使用序列中的位置信息，又能保留并行计算的高效性**

8.5 位置编码

- 示例：基于正弦函数和余弦函数的固定位置编码
  - 假设输入 $X^{n\times d}$ 包含一个序列中n个词元的d维向量表示
  - 位置编码矩阵 $P^{n\times d}$ 的第 i 行元素：

$$P_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right)$$
$$P_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right)$$

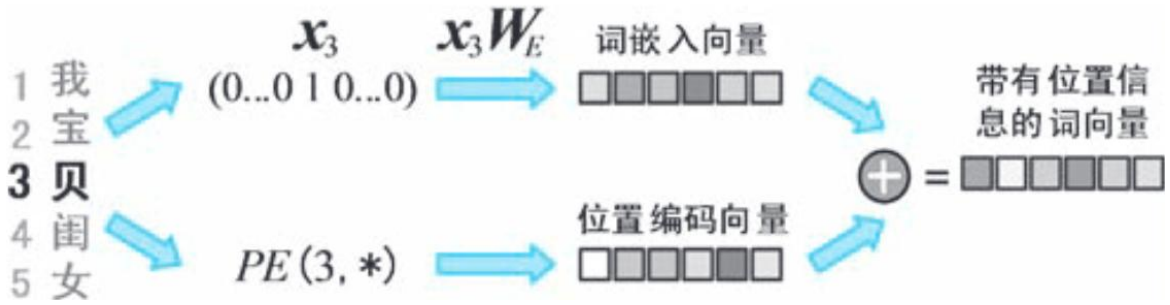
P的i行，即第i个词元，**隐含序列中的位置信息**

为了让相邻位置产生差异，偶数位置用正弦，奇数位置用余弦

第 i个词元的位置编码：

$$\left[\sin(i\omega_0), \cos(i\omega_0), \sin(i\omega_1), \cos(i\omega_1), \cdots, \sin(i\omega_{d/2}), \cos(i\omega_{d/2})\right] \qquad \omega_j = \frac{1}{10000^{2j/d}}$$

  - 将 $X+P$ 作为模型输入



8.5 位置编码

- 示例：旋转位置编码（rotary position embedding, RoPE），在注意力机制中融入相对位置信息
  - 位置索引 t 对应的旋转矩阵如下，其中 $\theta_r = \frac{1}{10000^{2(r-1)/d}}$

$$\boldsymbol{R}_{\theta,t} = \begin{bmatrix} \cos t\theta_1 & -\sin t\theta_1 & 0 & 0 & \dots & 0 & 0 \\ \sin t\theta_1 & \cos t\theta_1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \cos t\theta_2 & -\sin t\theta_2 & \dots & 0 & 0 \\ 0 & 0 & \sin t\theta_2 & \cos t\theta_2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \cos t\theta_{H/2} & -\sin t\theta_{H/2} \\ 0 & 0 & 0 & 0 & \dots & \sin t\theta_{H/2} & \cos t\theta_{H/2} \end{bmatrix}$$

$$\begin{aligned} &\begin{bmatrix} \cos i\theta_r & -\sin i\theta_r \\ \sin i\theta_r & \cos i\theta_r \end{bmatrix}^T \begin{bmatrix} \cos j\theta_r & -\sin j\theta_r \\ \sin j\theta_r & \cos j\theta_r \end{bmatrix} \\ =&\begin{bmatrix} \cos i\theta_r & \sin i\theta_r \\ -\sin i\theta_r & \cos i\theta_r \end{bmatrix} \begin{bmatrix} \cos j\theta_r & -\sin j\theta_r \\ \sin j\theta_r & \cos j\theta_r \end{bmatrix} \\ =&\begin{bmatrix} \cos i\theta_r \cos j\theta_r + \sin i\theta_r \sin j\theta_r & -\cos i\theta_r \sin j\theta_r + \sin i\theta_r \cos j\theta_r \\ -\sin i\theta_r \cos j\theta_r + \cos i\theta_r \sin j\theta_r & \sin i\theta_r \sin j\theta_r + \cos i\theta_r \cos j\theta_r \end{bmatrix} \\ =&\begin{bmatrix} \cos(j-i)\theta_r & -\sin(j-i)\theta_r \\ \sin(j-i)\theta_r & \cos(j-i)\theta_r \end{bmatrix} \end{aligned}$$

将查询和键乘以旋转矩阵

$$\widetilde{q}_i = R_{\theta,i} q_i$$

$$\widetilde{k}_j = R_{\theta,j} k_j$$

注意力权重只与查询、键及其相对位置有关

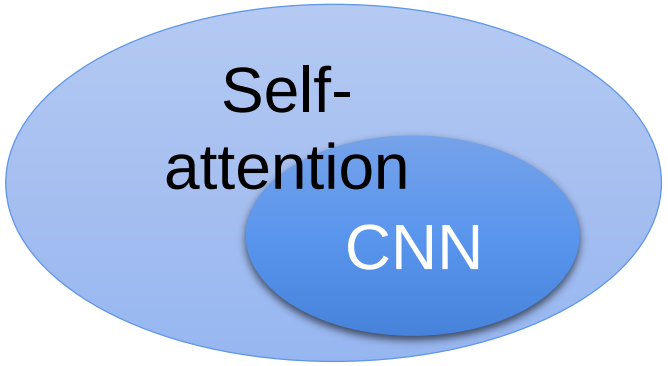
$$\begin{aligned} \text{softmax}\left(\frac{\widetilde{q}_i^T \widetilde{k}_j}{\sqrt{d}}\right) &= \text{softmax}\left(\frac{q_i^T R_{\theta,i}^T R_{\theta,j} k_j}{\sqrt{d}}\right) \\ &= \text{softmax}\left(\frac{q_i^T R_{\theta,j-i} k_j}{\sqrt{d}}\right) \end{aligned}$$

$$R_{\theta,i}^T R_{\theta,j} = R_{\theta,j-i}$$

8.6 注意力 V.S. CNN

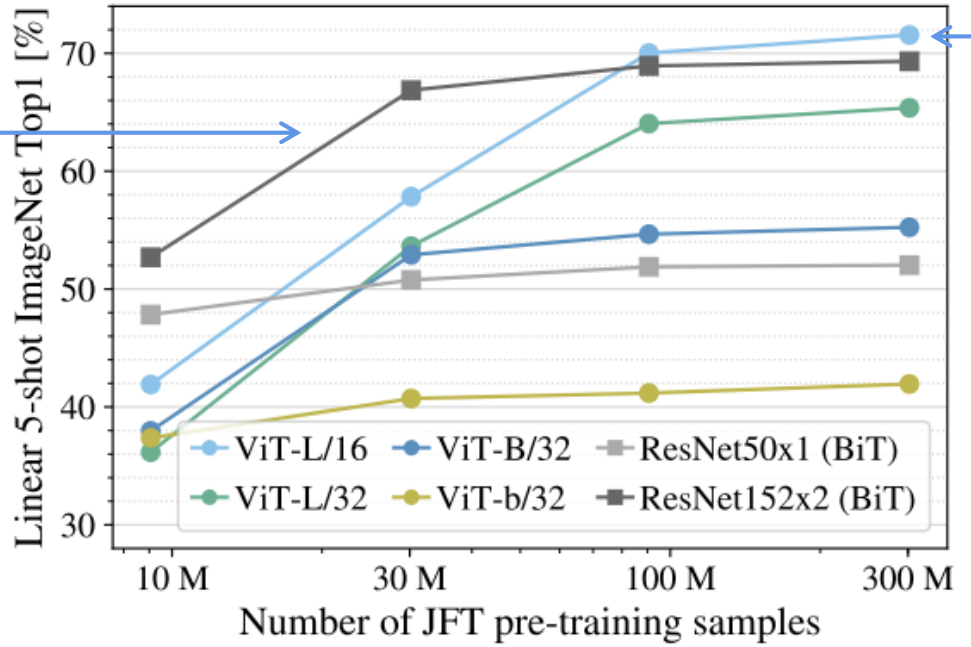
3解

- CNN只考虑了局部信息，默认局部信息是需要重点关注的
- 注意力考虑了全局信息，通过学习得到哪些信息需要重点关注
- CNN是简化版本的注意力
  - 注意力是更灵活的CNN，而CNN是受限制的注意力
  - 更灵活的模型需要更多的数据，如果数据不够，有可能过拟合
  - 有限制的模型适合在数据少的时候使用，可能比较不会过拟合，如果限制设的好，也会有不错的结果。



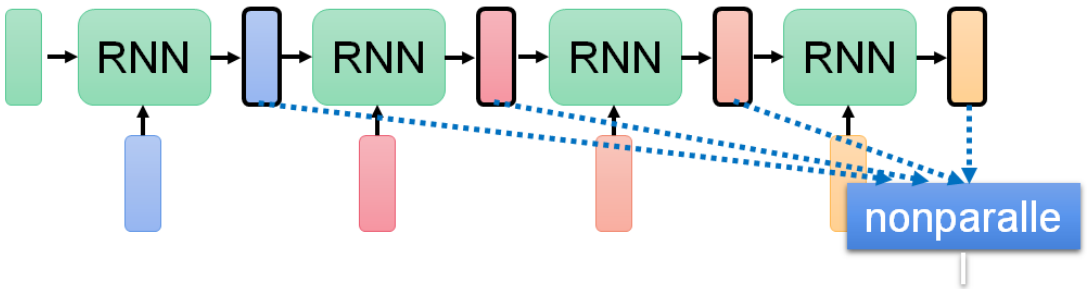
CNN适合数据量较少时

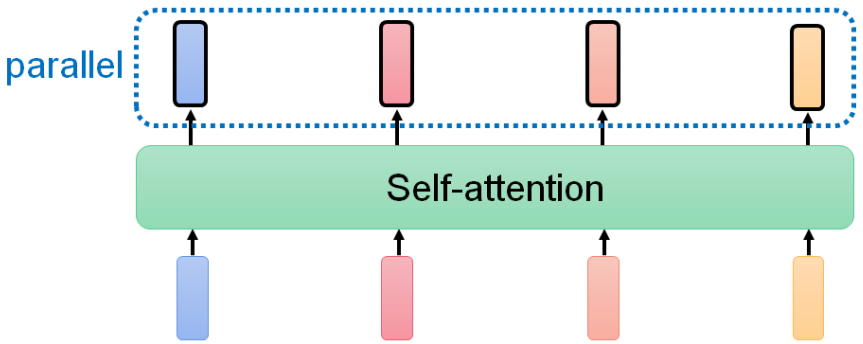
注意力适合数据量较大时



8.6 注意力 V.S. RNN

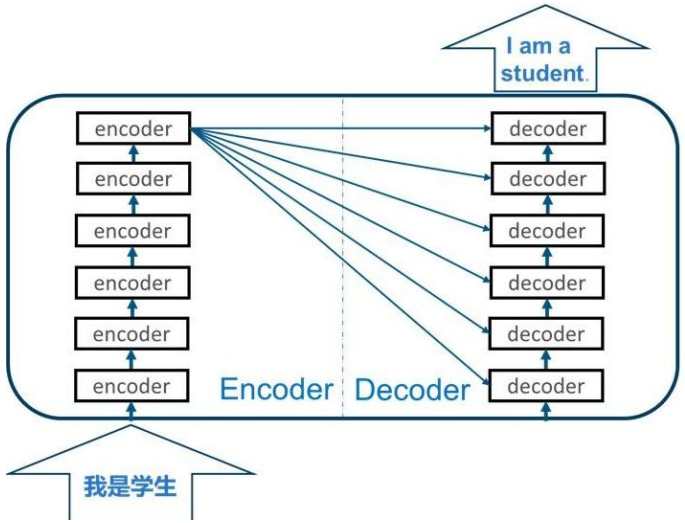
- 计算效率
  - RNN当前时刻的计算依赖前一时刻，需要一个接一个地处理信息，无法并行，计算复杂度高
  - 注意力机制丢掉了序列的顺序信息，可以并行
- 长距离依赖
  - 序列较长时，RNN存在梯度消失和梯度爆炸问题，不能很好解决长距离依赖关系
  - 注意力使得序列中任意两个词之间的依赖关系可以直接被建模而不基于传统的循环结构，不因序列太长导致梯度消失和爆炸，更好地解决长距离依赖关系

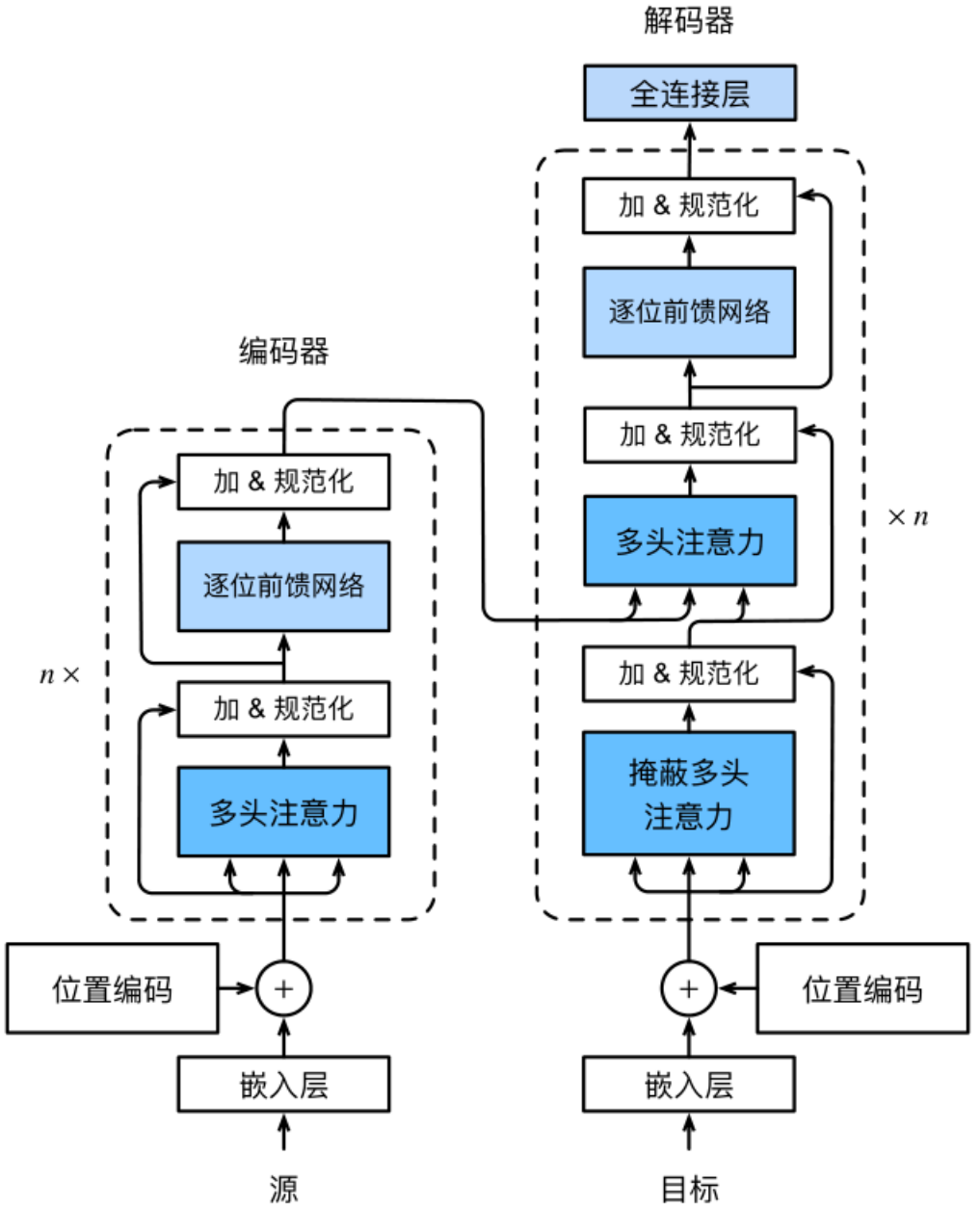




8.7 Transformer

- Transformer：由谷歌公司于 2017 年提出，继CNN、RNN之后的另一个潮流
- 模型
  - Transformer由编码器和解码器组成
  - Transformer的编码器和解码器由基于**多头自注意力**的模块叠加而成





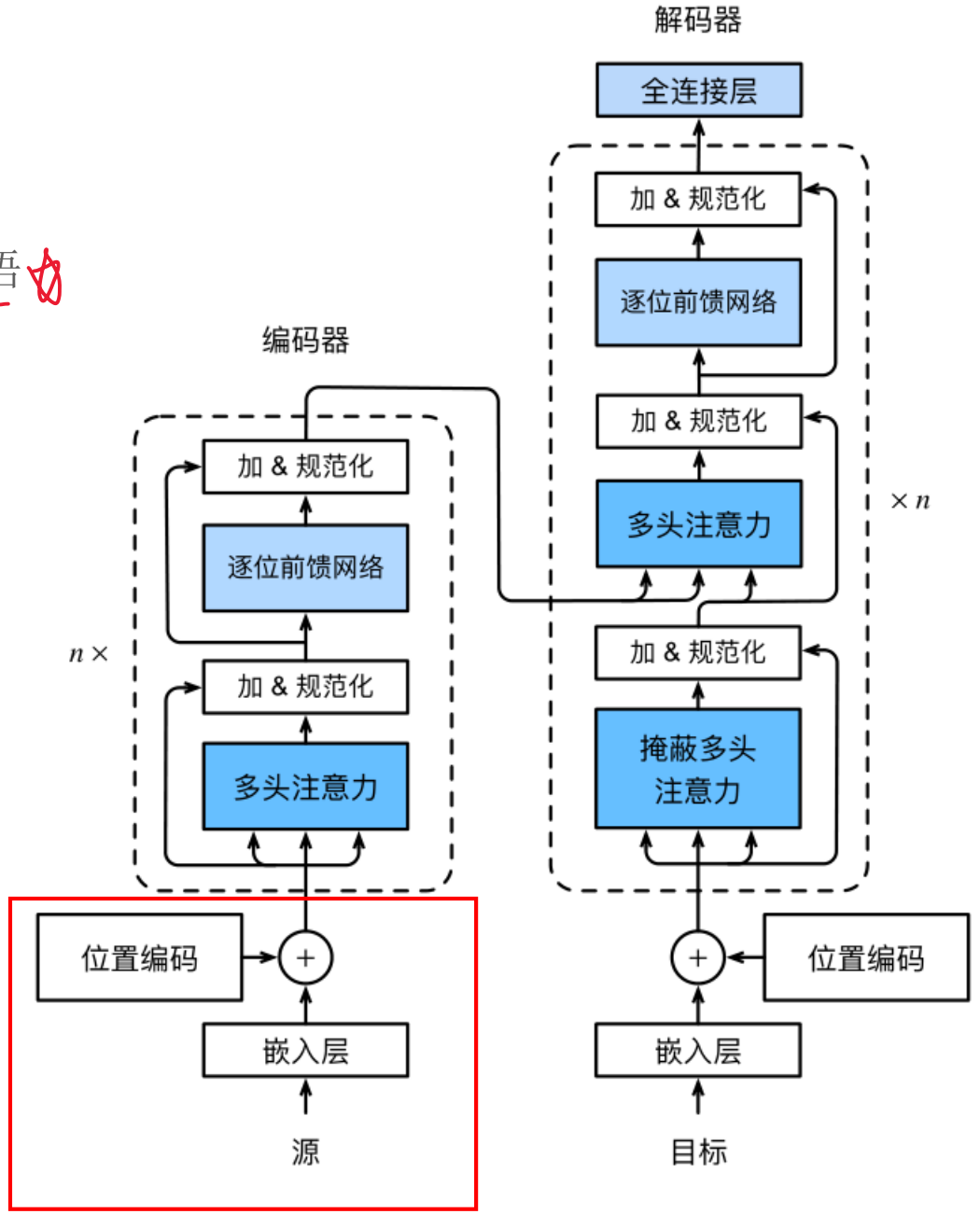
8.7.1 输入和位置编码

- Token：词元，文本中的基本单元，通常为词或短语
- Tokenization（词元化）：字符序列切分token
  - 词粒度、字符粒度、子词粒度
  - 词粒度：Let’s do totenization!

    Let’s/do/totenization!
  - 字符粒度：Let’s do totenization!

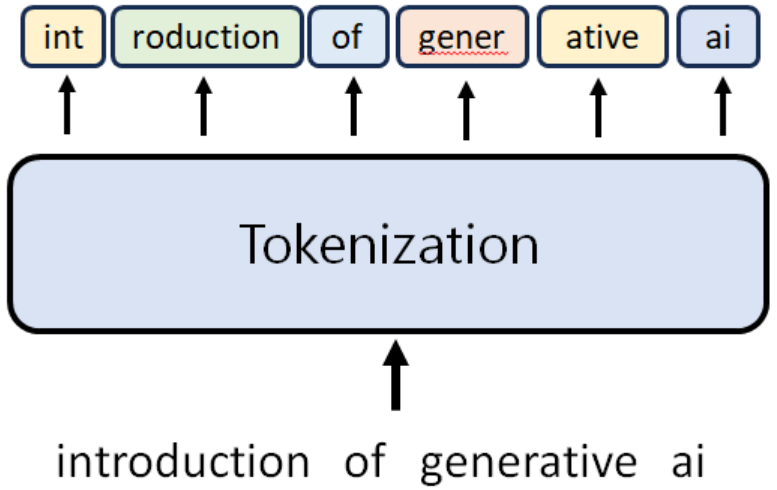
    L/e/t/’/s/d/o/t/o/t/e/n/i/z/a/t/i/o/n/!
  - 子词粒度：分解到词根词缀等粒度
    - 例句：he is likely to be unfriendly to me

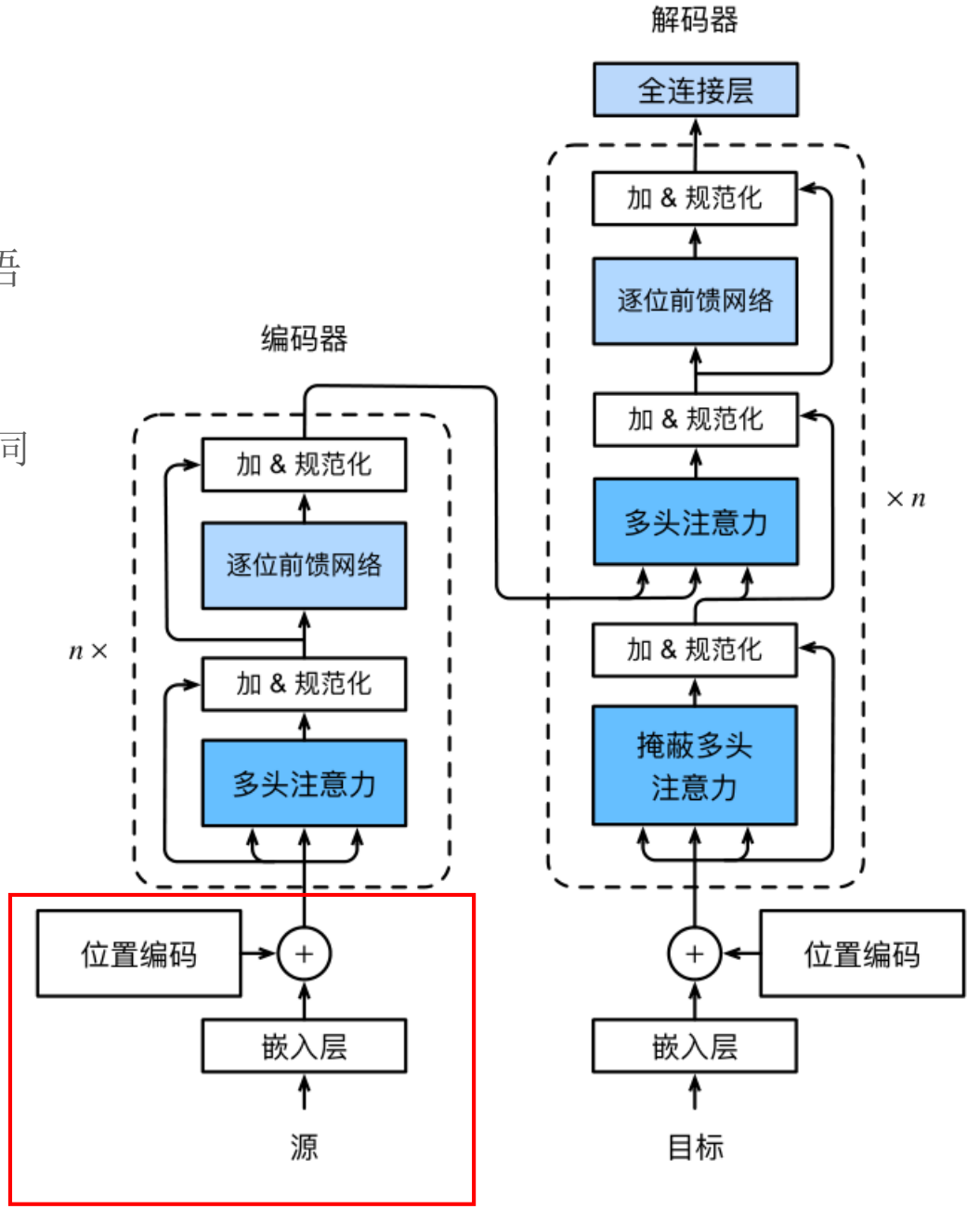
      he/is/like/ly/to/be/un/friend/ly/to/me



8.7.1 输入和位置编码

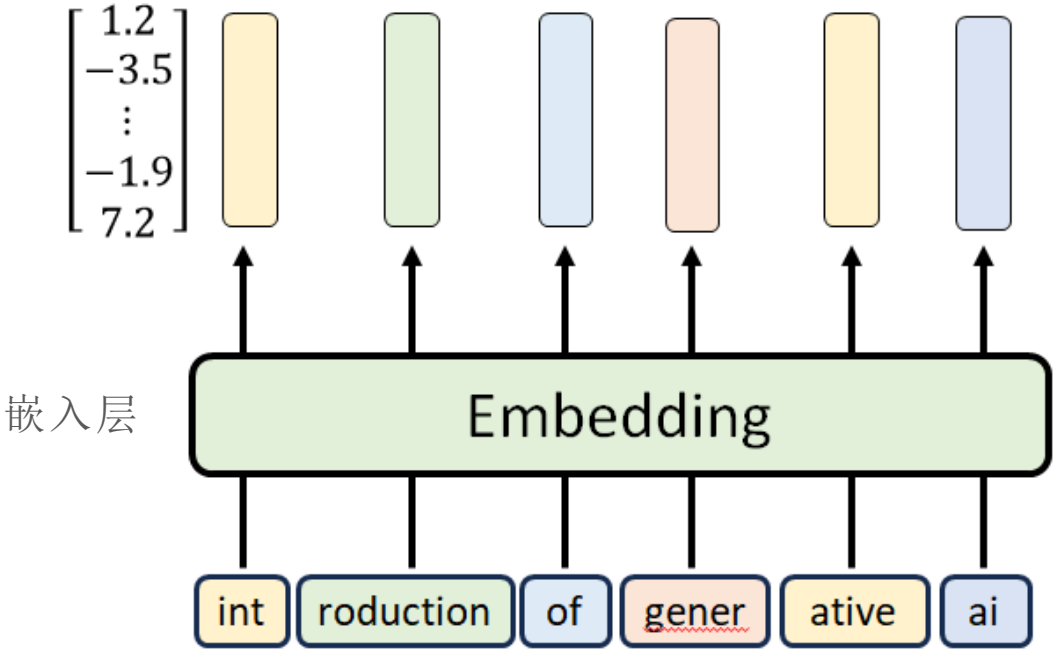
- Token：词元，文本中的基本单元，通常为词或短语
- Tokenization（词元化）：字符序列切分token
  - 词粒度缺点：dog、dogs、do、doing、done视为不同词元，词表庞大，会出现不在词表中的词
  - 字符粒度缺点：蕴含的语义信息较少
  - 子词粒度优点：词表尽可能小、具有语义信息、更好地处理同前缀同后缀的词、较好处理未见过词



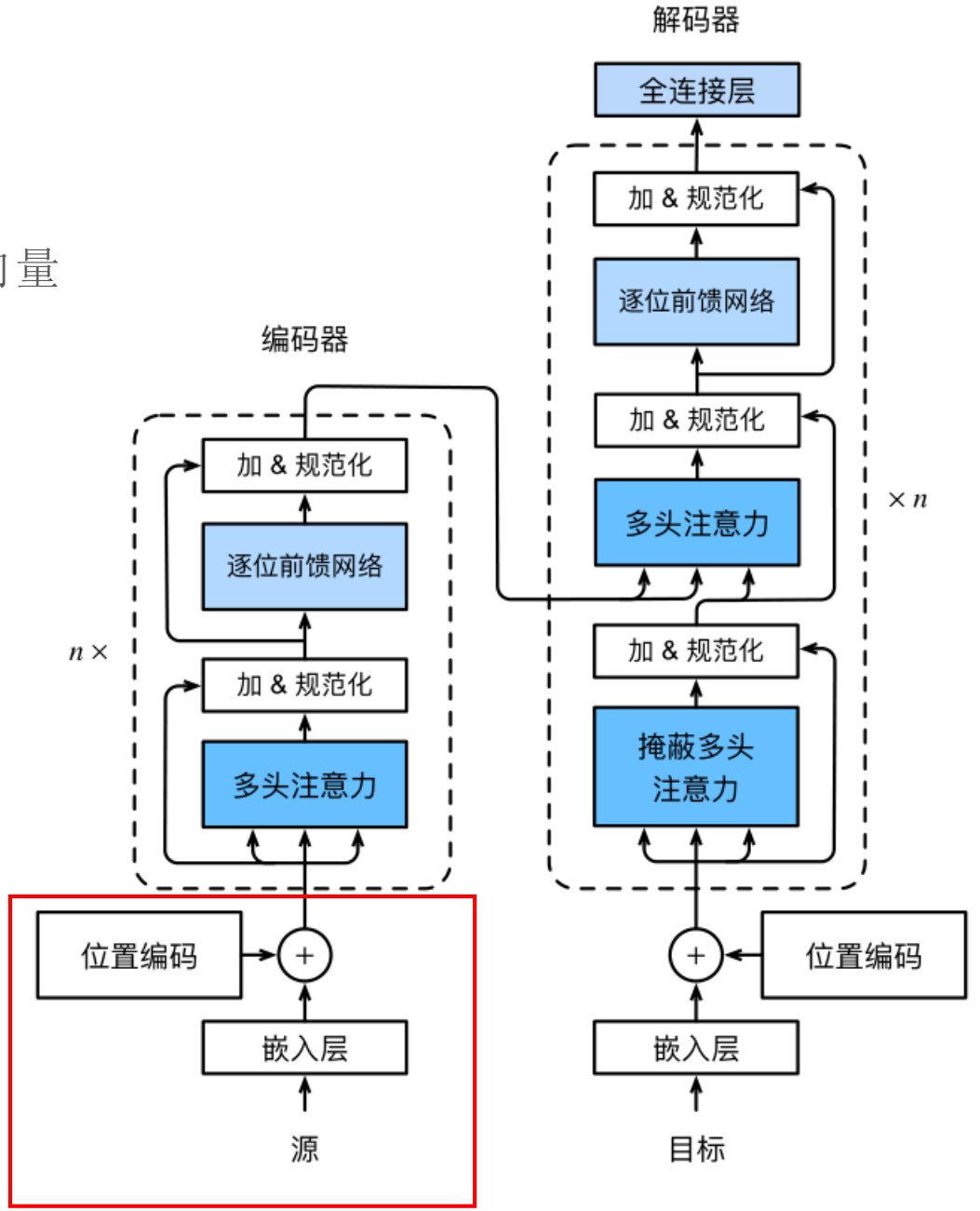


8.7.1 输入和位置编码

- 嵌入层：将输入序列（源）的token转换为词嵌入向量



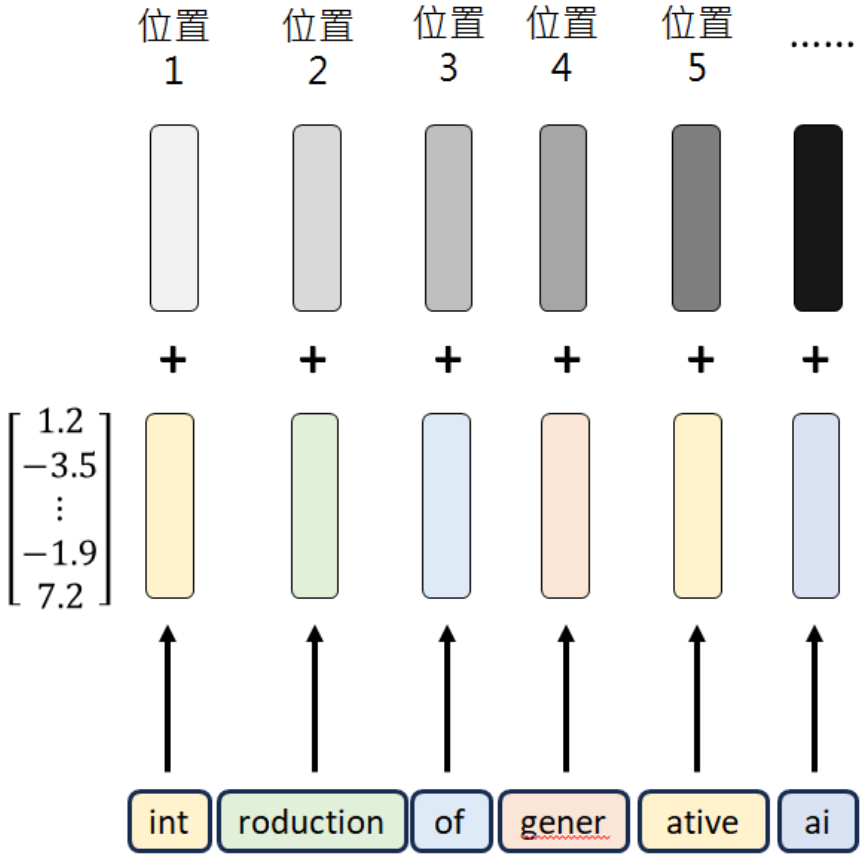
语义相近的token拥有相近的嵌入向量

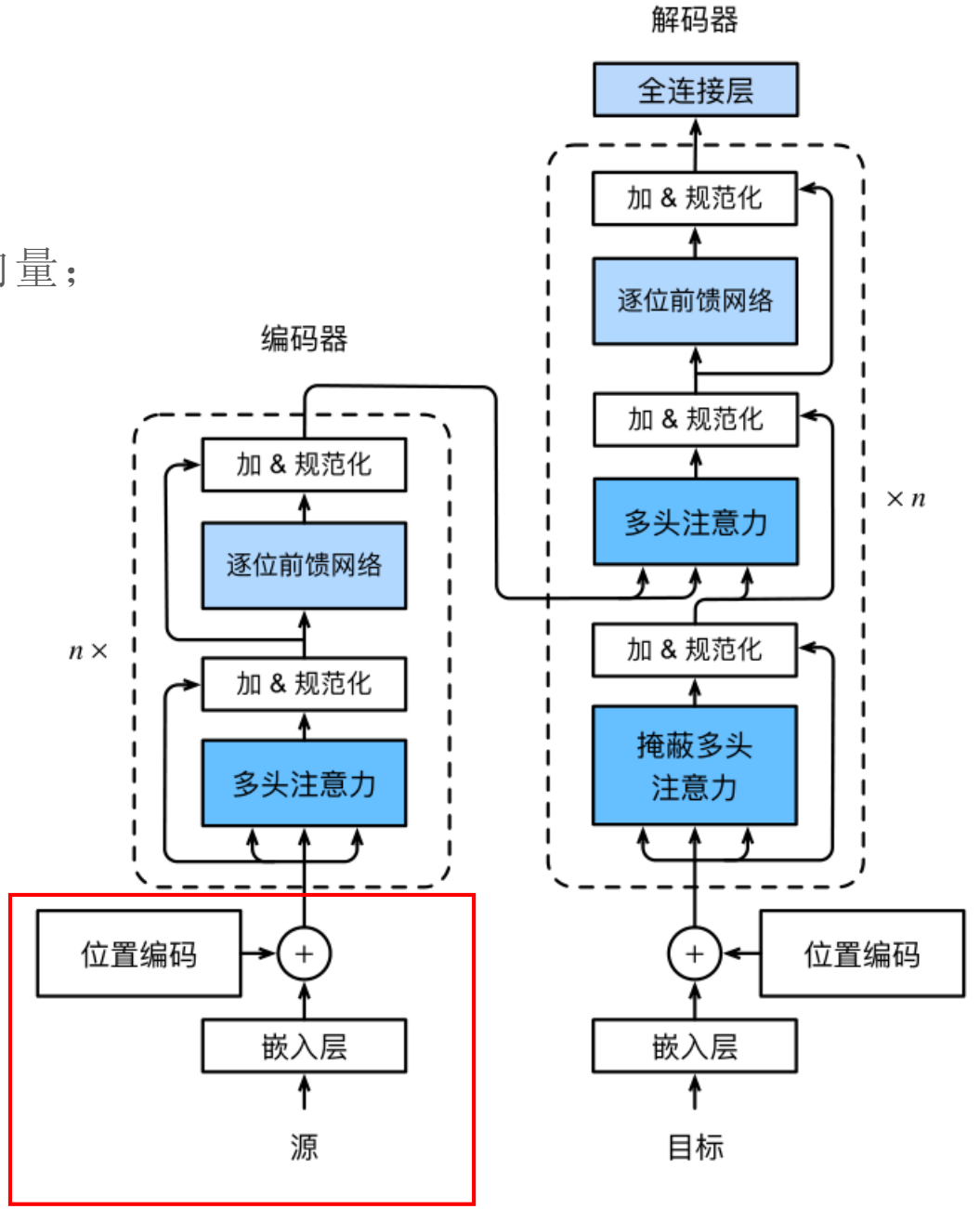


8.7.1 输入和位置编码

- 嵌入层：将输入序列（源）的token转换为词嵌入向量；

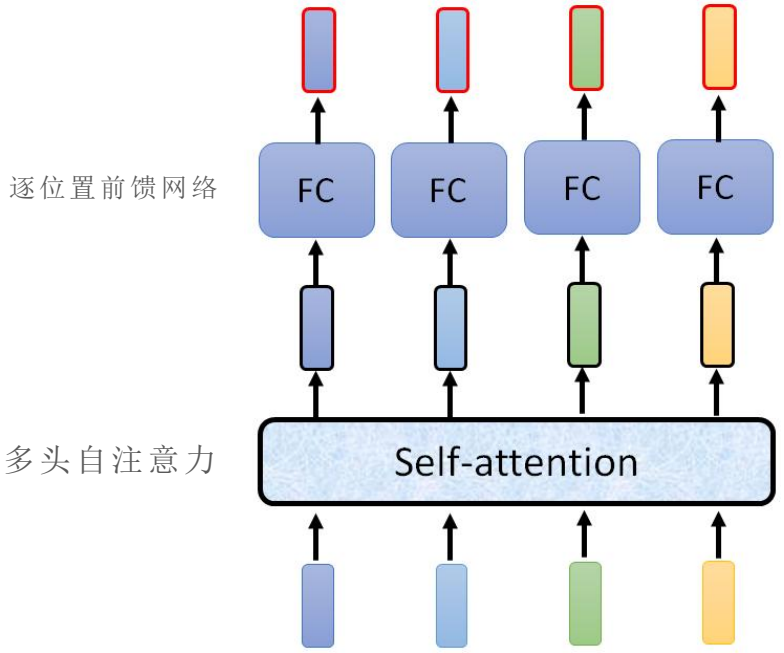
  词嵌入向量加上位置编码向量，作为编码器的输入





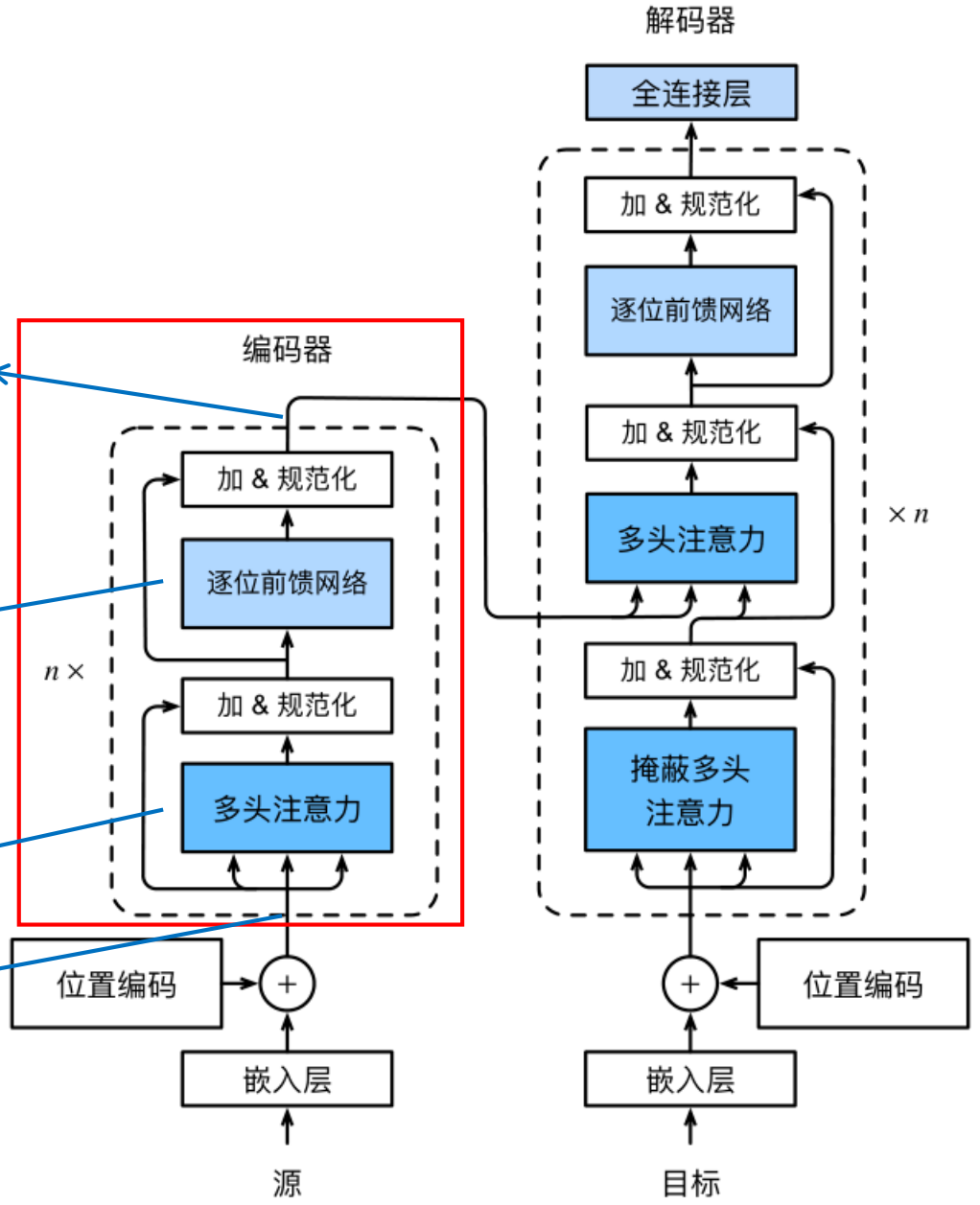
8.7.2 编码器

- 编码器用于编码源语言序列中的信息
- 编码器由多个相同的层叠加而成的，每个层都有两个子层 怎么计算



输出：**相同长度**的序列，序列每个位置也是一个**d维向量**（保证残差连接可以顺利展开）

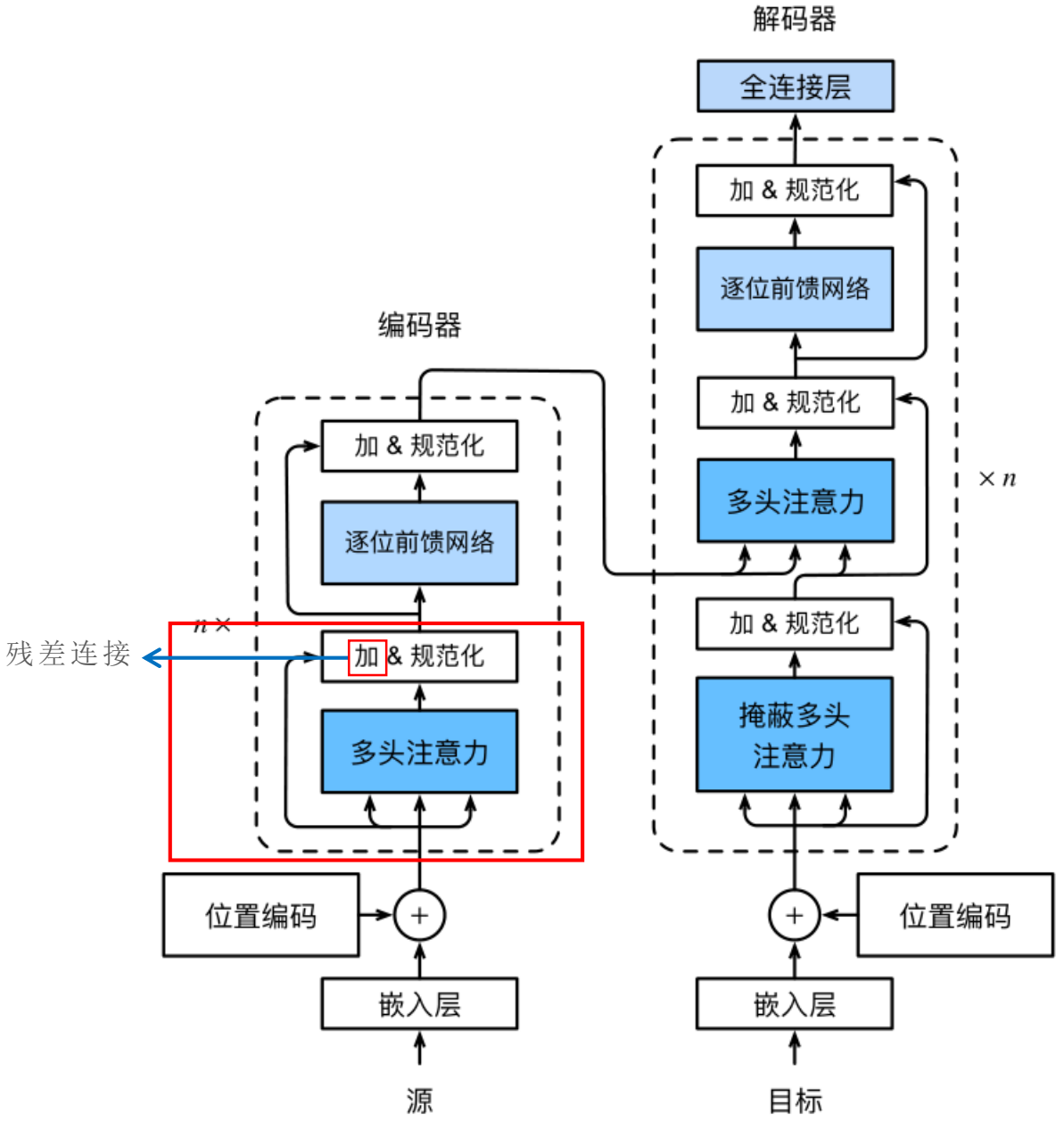
输入：一个序列，序列每个位置为一个d维向量



8.7.2 编码器

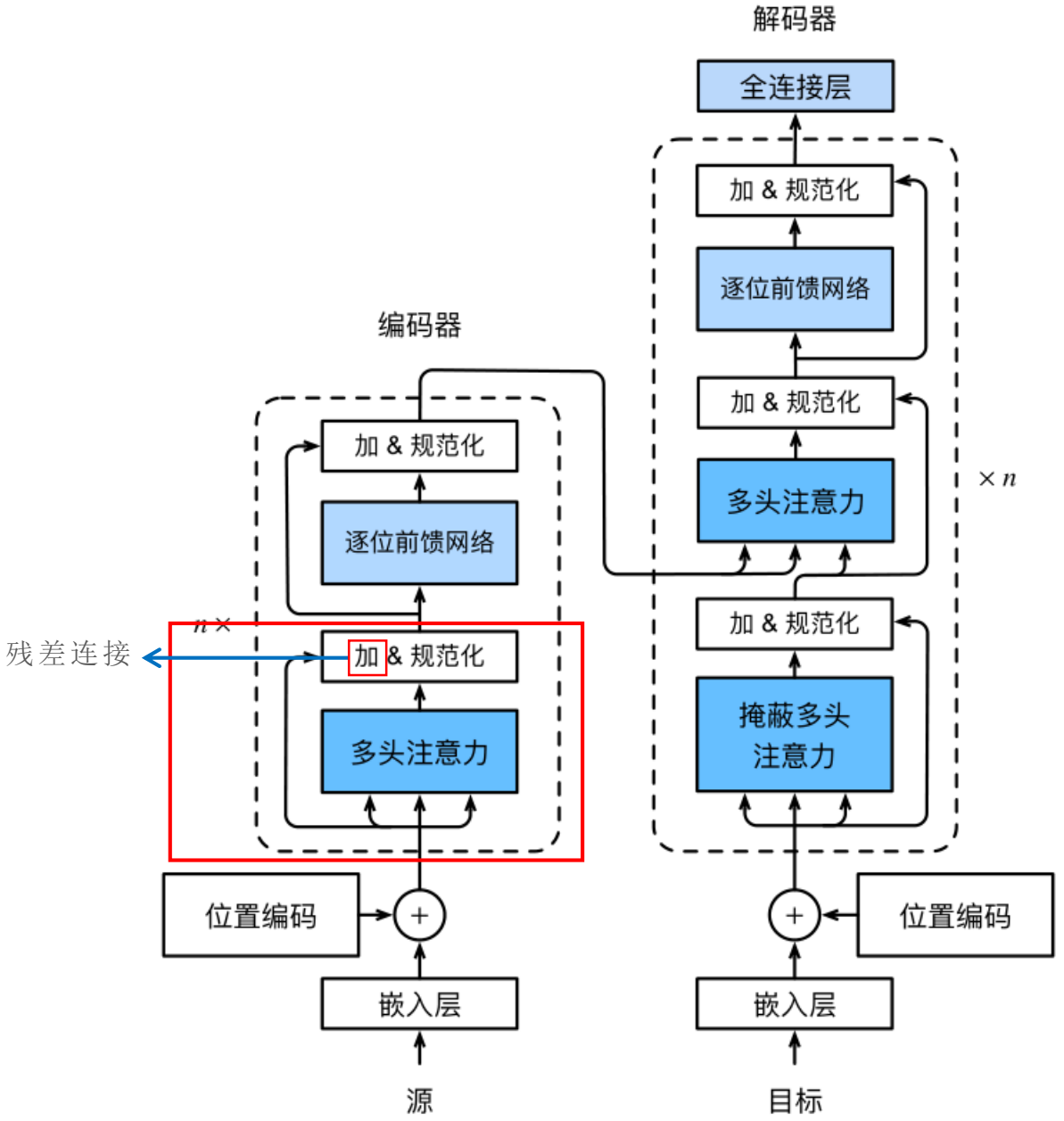
- 第一个子层是**多头自注意力**
  - 查询、键和值都来自前一个编码器层的输出，常用**缩放点积**作为评分函数
  - 受残差网络的启发，每个子层都采用了残差连接
  - 层规范化（LayerNorm）：受BatchNorm启发，但层规范化是基于每一层的每个token的特征向量进行规范化
- 形式化描述

$$X_A = \text{LayerNorm}(\text{MultiheadSelfAttention}(X) + X)$$



8.7.2 编码器

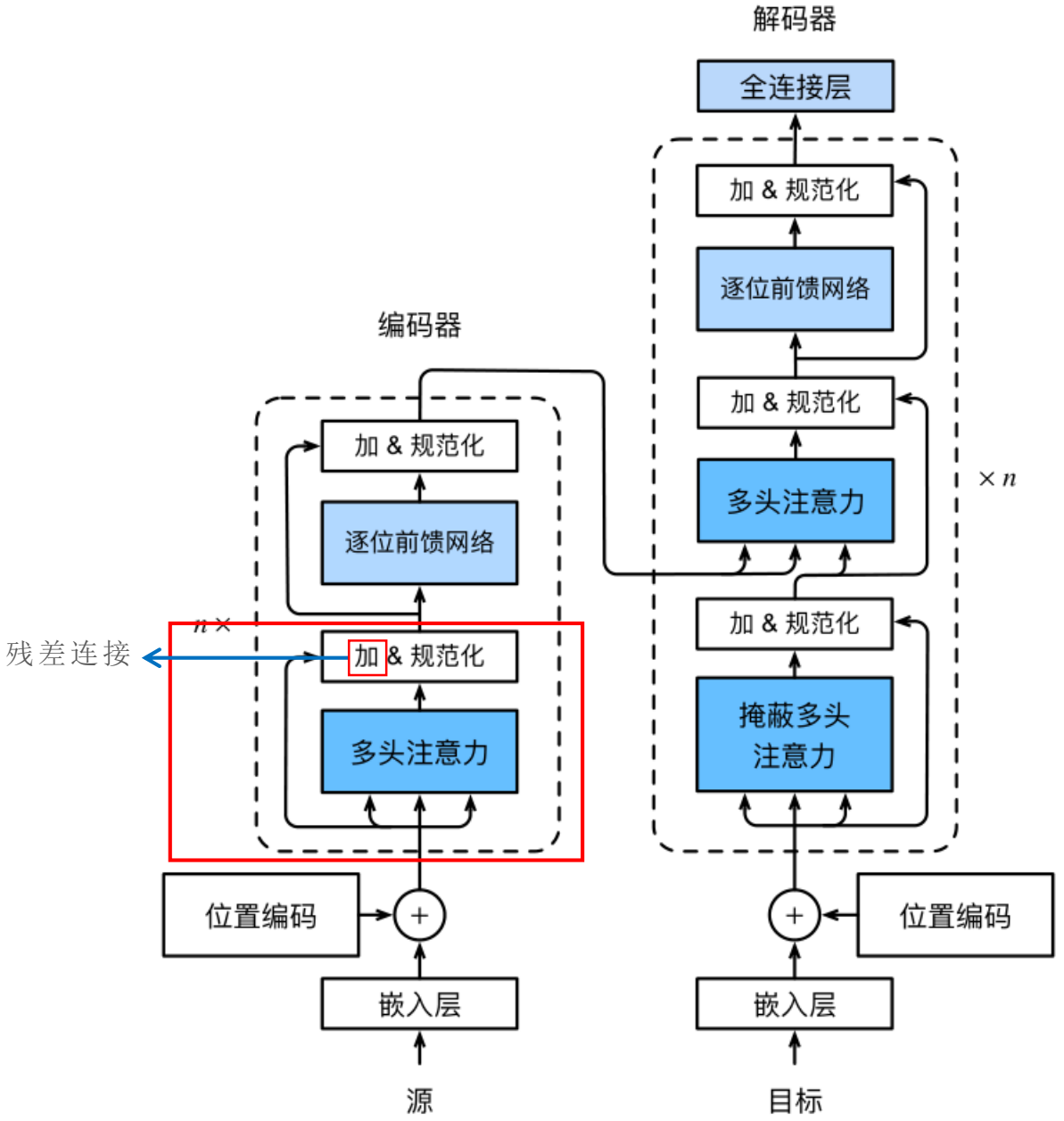
- 第一个子层是**多头自注意力**
  - 注意力机制作用：捕捉输入序列中元素与元素之间的关系，学习输入序列不同元素的相关性，捕捉元素之间的长距离依赖关系，更好地理解输入序列
  - 实现方式：编码器一次性看到完整的序列，使用标准的多头自注意力，以并行的方式对每个元素进行编码



8.7.2 编码器

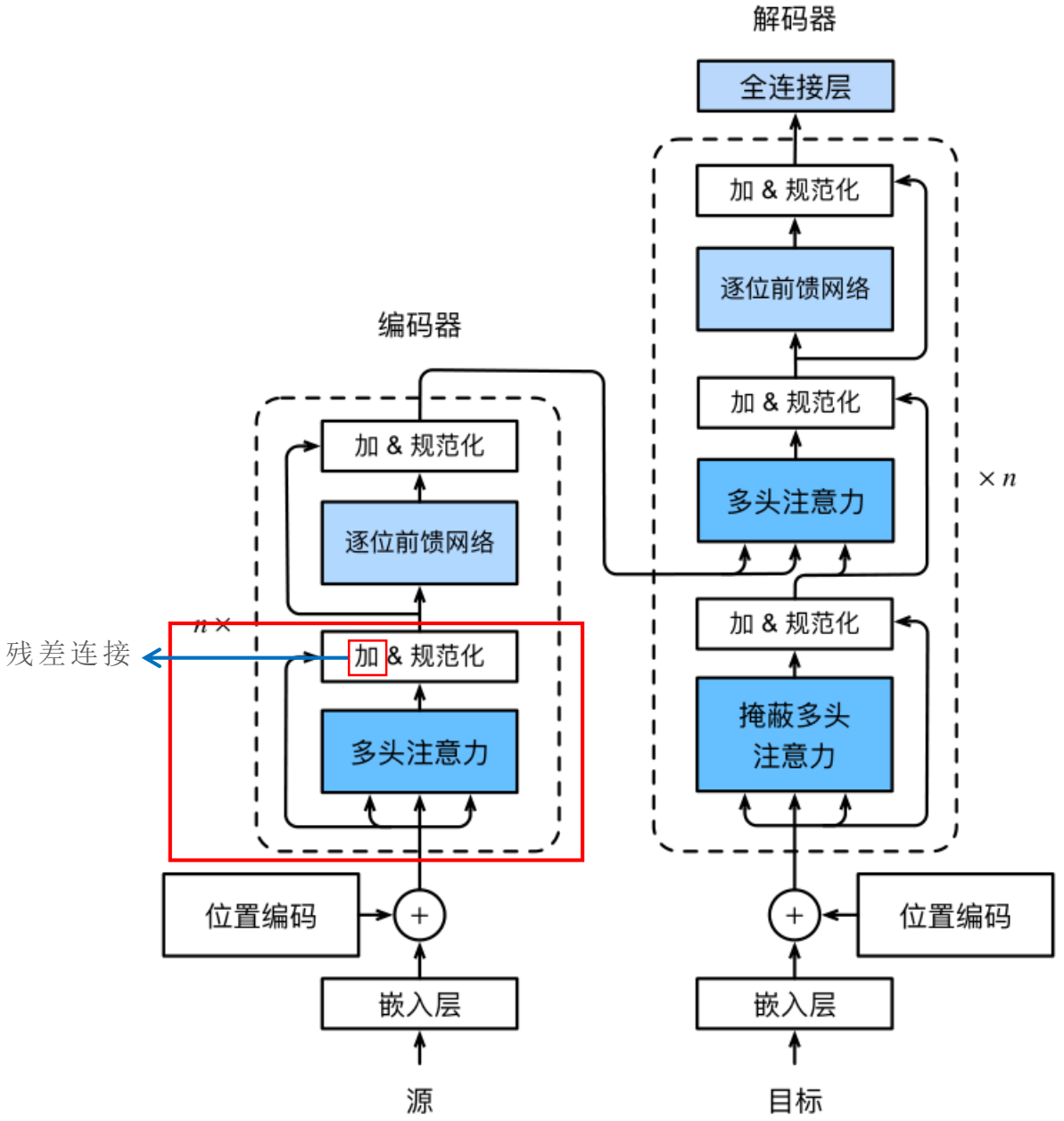
- 第一个子层是多头自注意力
  - LayerNorm：对**某个token的某一层的特征向量的所有元素**，按层计算均值和均方差，进行归一化
  - LayerNorm的目标是让每个token的特征向量变得“稳定”，即拥有均值为0、方差为1的分布

不考，面试



8.7.2 编码器

- 第一个子层是多头自注意力
  - LayerNorm 优点
    - 对变长序列： LayerNorm对每个token独立归一化，不受序列长度影响；
    - 对小批次和稀疏数据：当批次较小时， BatchNorm的批次统计量可能不准确； LayerNorm不依赖批次统计量，适合小批次或稀疏数据



8.7.2 编码器

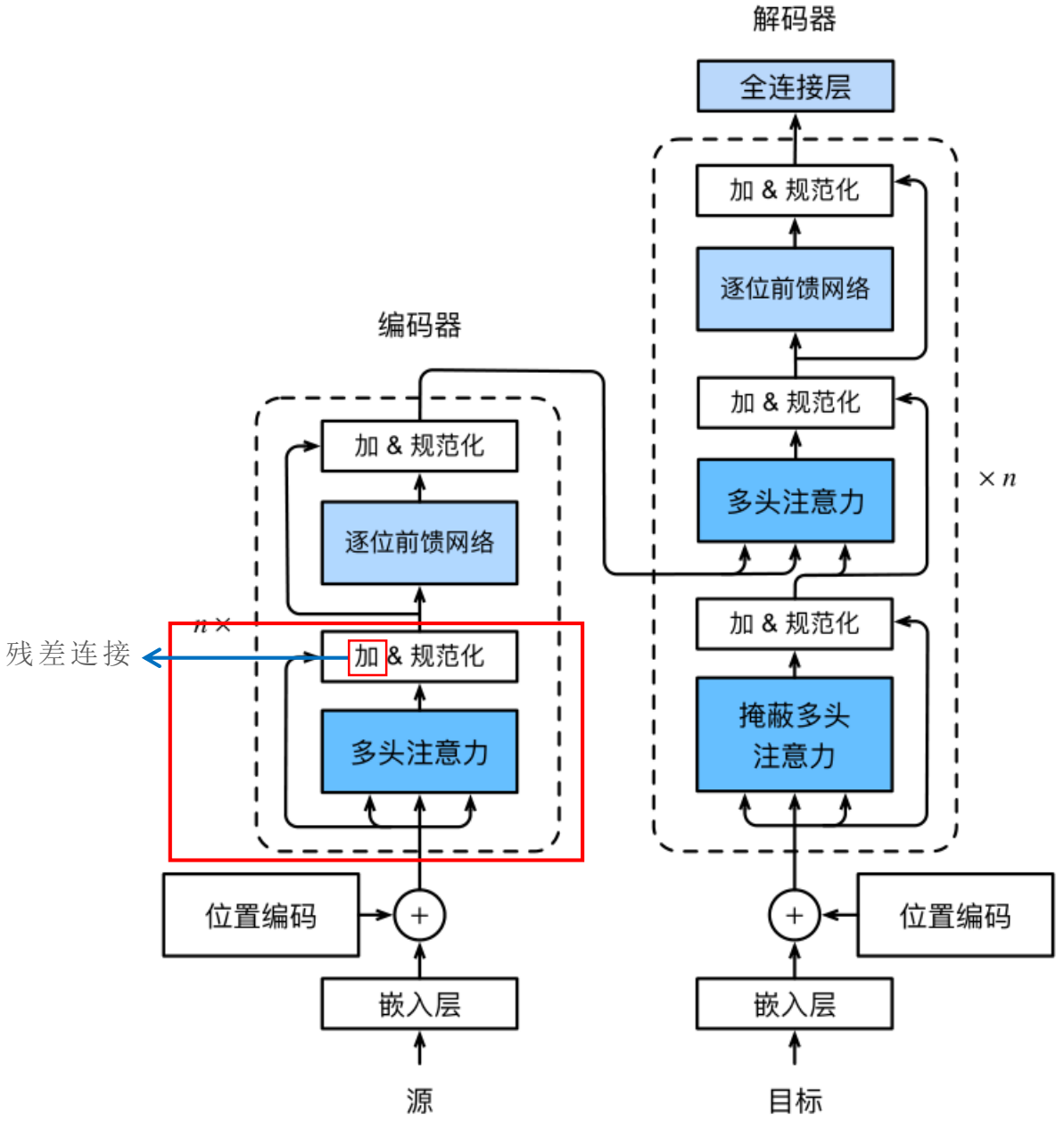
- 第一个子层是多头自注意力
  - LayerNorm之后出现了很多改进的规范化方法
  - 均方根层规范化（RMS-Norm）
    - RMS-Norm 的作者认为，居中（即减去均值）可能不是关键所在，归一化的主要好处来自于其重新缩放（即除以标准差） 的特性
    - RMS-Norm只使用特征向量的均方根进行缩放，计算效率更高

$$\text{RMS}^l = \sqrt{\frac{1}{n_l}\sum_{i=1}^{n_l}(h_i^l)^2}, \forall l$$

$$\hat{h}_i^l = \gamma_i \frac{h_i^l}{\text{RMS}^l}, \forall l$$

$$LN(h) = \hat{h}$$

某个token在第l层的第i个特征为h_i^l，第l层的特征总数为n_l



8.7.2 编码器

- 第二个子层是**逐位置前馈网络**
  - 使用前馈网络对序列中的**每个位置**的特征向量表示进行变换

$$W_2\,(\mathrm{GeLU}(W_1 x + b_1)) + b_2$$

将前一层的输出映射回与输入相同的维度（或与模型其他部分兼容的维度）。这一层通常没有非线性激活函数

GeLU函数帮助模型捕获非线性关系，提高模型的表达能力

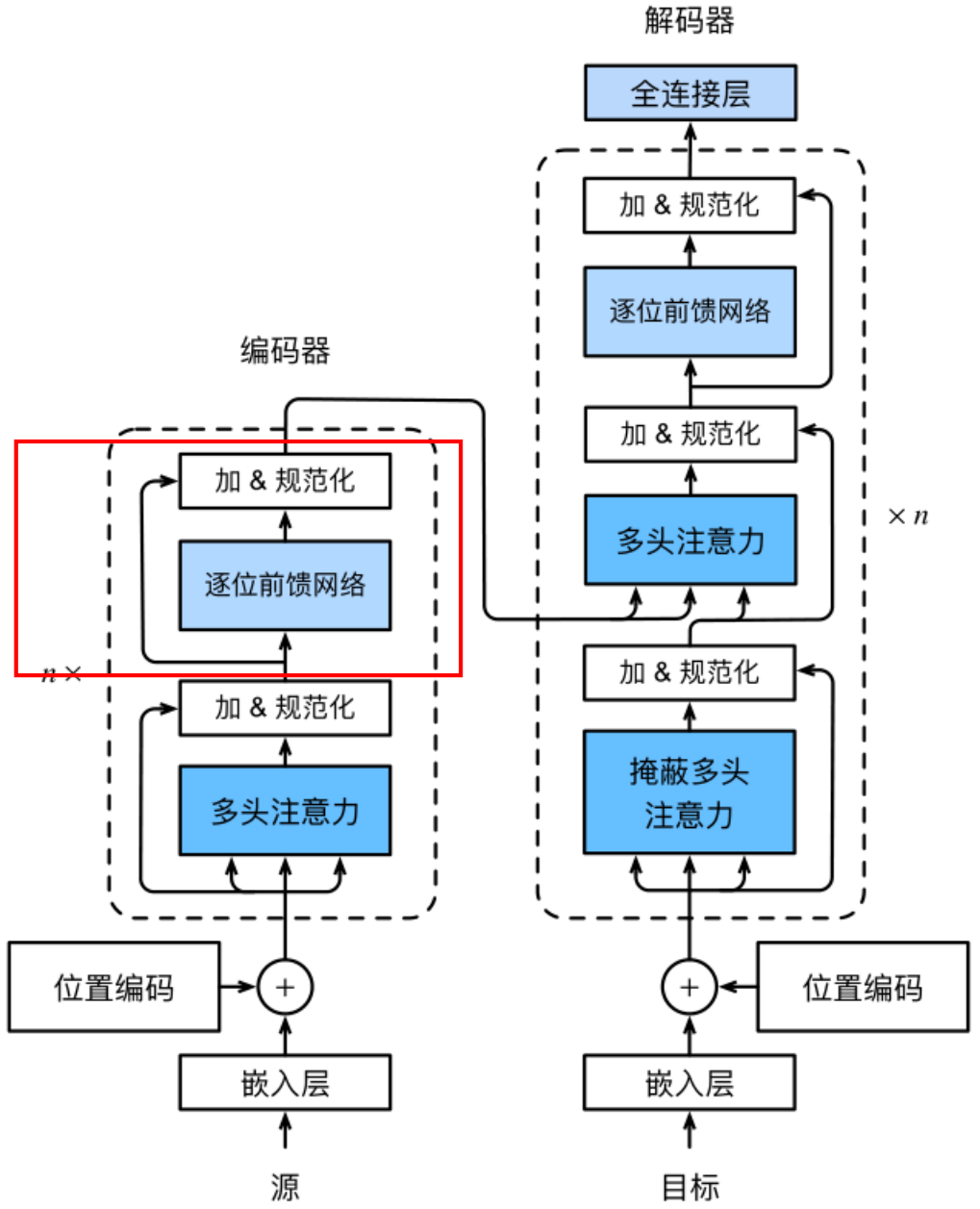
接收自注意力层的输出作为输入，并将其映射到一个更高维度的空间（一般为输入的4倍）。有助于模型学习更复杂的特征表示

经典设置中，

$W_1 \in \mathbb{R}^{4d \times d}$

$W_2 \in \mathbb{R}^{d \times 4d}$

FFN层的整体作用：**整合理解注意力层的学习结果**



8.7.2 编码器

- 第二个子层是**逐位置前馈网络**
  - 使用前馈网络对序列中的**每个位置**的特征向量表示进行变换

$$W_2\,(\mathrm{GeLU}(W_1 x + b_1)) + b_2$$

  - 前馈网络的参数 W_1, W_2, b_1, b_2 针对句子不同位置的特征向量是相同的（参数共享），但对不同编码器层是不同的
    - 不同层需要学习不同的参数，但同一层只需要学习 W_1, W_2, b_1, b_2 即可
- 针对整个输入序列的矩阵形式描述：

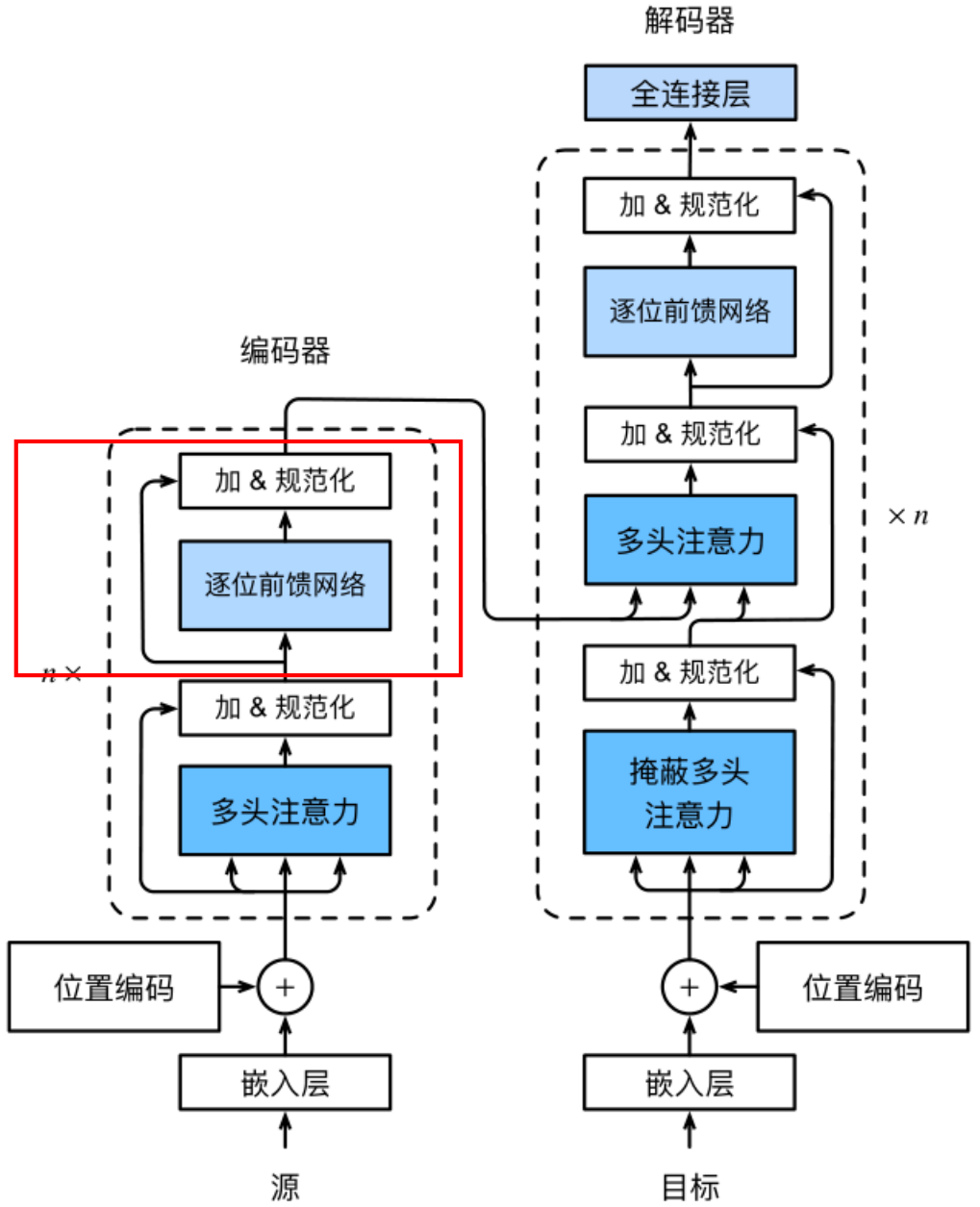
$$\mathrm{PositionFFN}(X_A) = W_2\mathrm{GeLU}(W_1 X_A + B_1) + B_2$$

X_A 的每一列对应一个token的特征向量

B_1（B_2）的每一列均为 b_1（b_2）

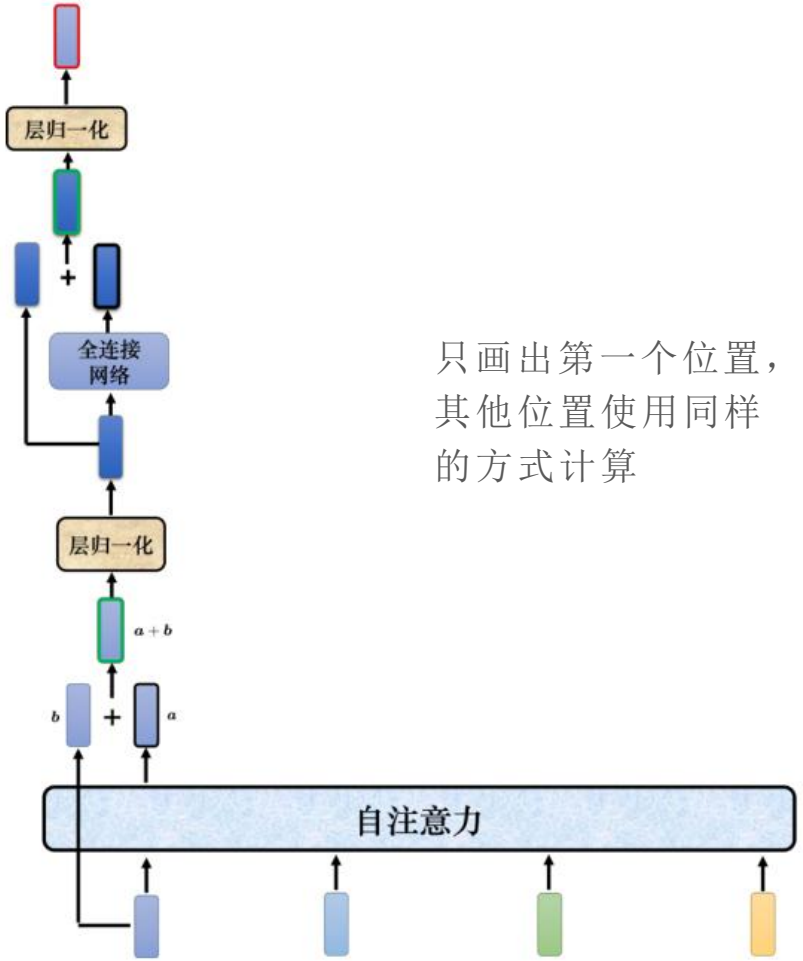
- FFN、残差连接、层规范化

$$X_B = \mathrm{LayerNorm}(\mathrm{PositionFFN}(X_A) + X_A)$$

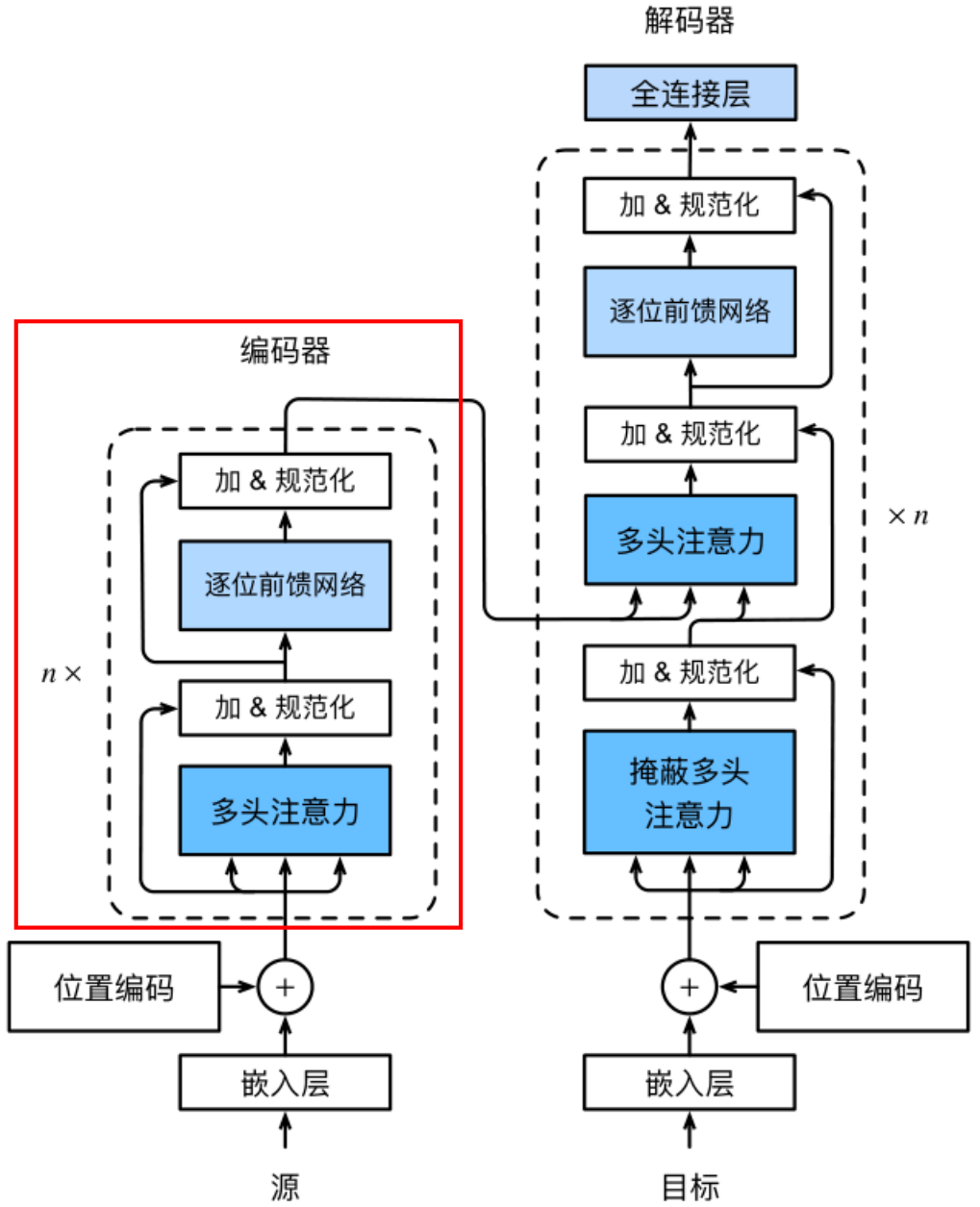


8.7.2 编码器

- 编码器小结

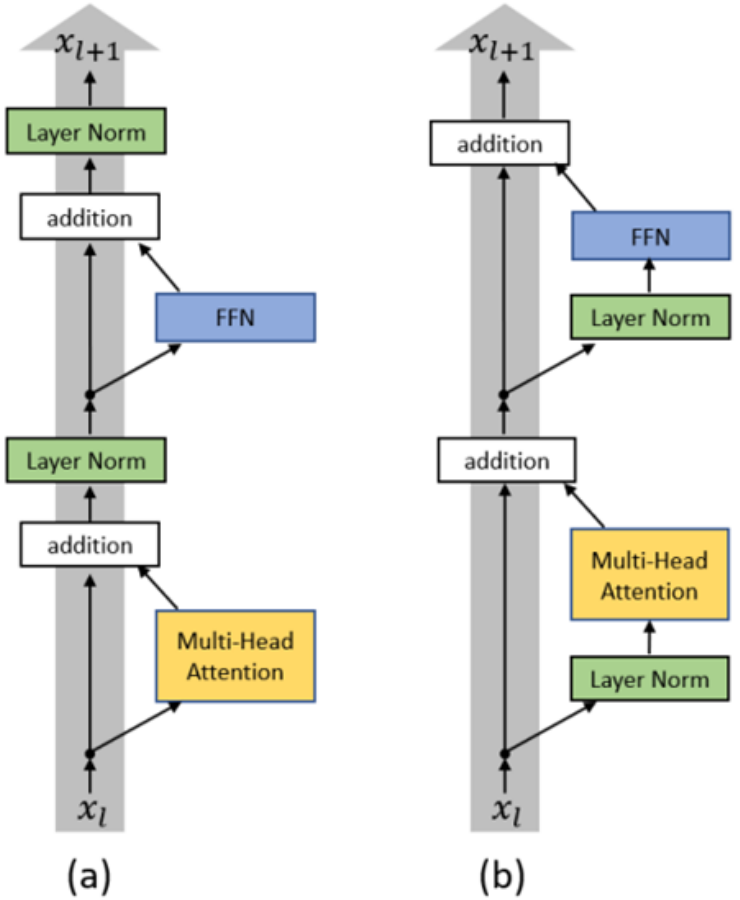


只画出第一个位置，其他位置使用同样的方式计算



8.7.2 编码器

- 编码器层内不同模块不同的组合顺序



后做规范化

Post Norm

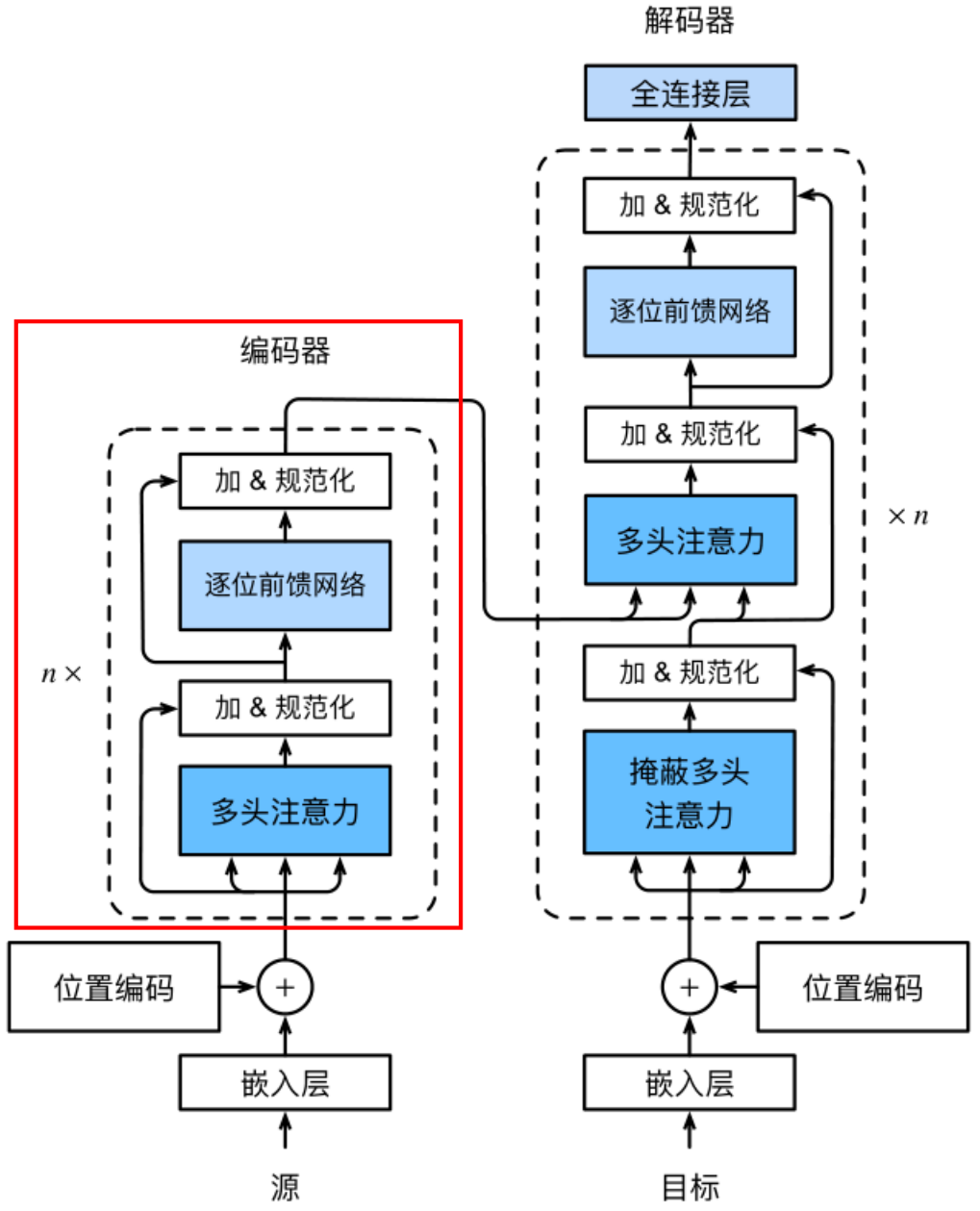
$x_{l+1} = \text{Norm}(x_l + F(x_l))$

先做规范化

Pre Norm

$x_{l+1} = x_l + F(\text{Norm}(x_l))$

F泛指MHA和FFN



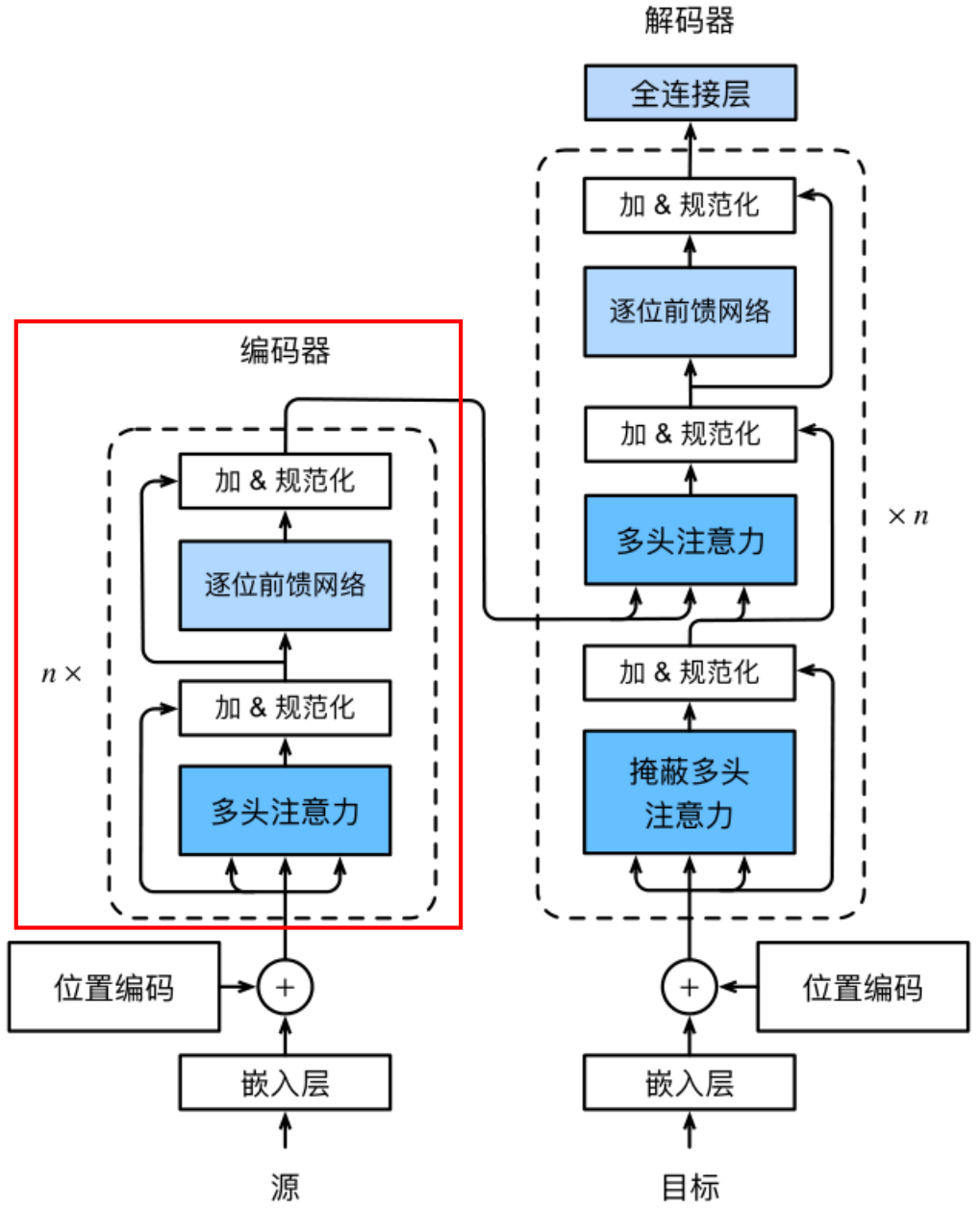
8.7.2 编码器

- 编码器层内不同模块不同的组合顺序
  - Post Norm在残差之后做Norm，残差的缓解梯度消失/爆炸的作用被削弱，导致模型的梯度不稳定

$$x_{l+1} = \text{Norm}(x_l + F(x_l))$$

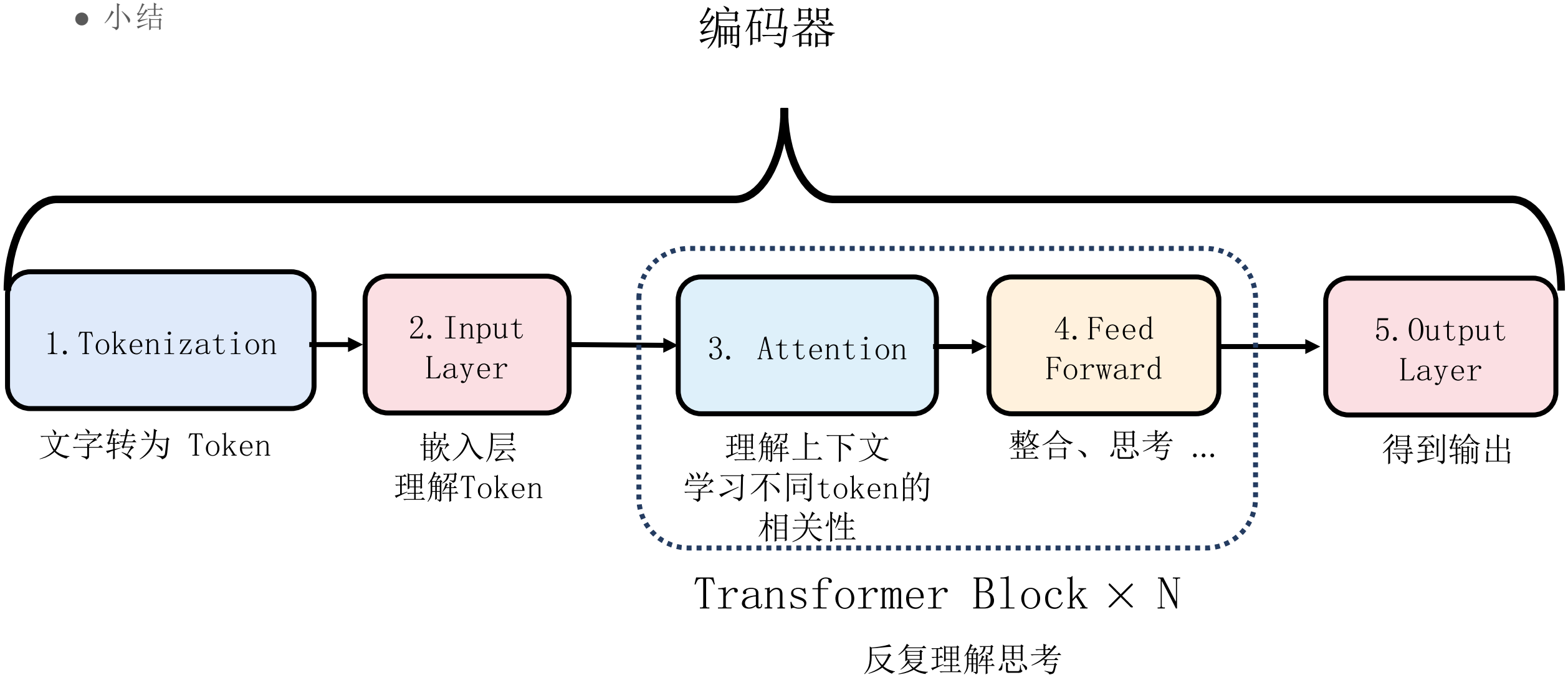
  - Pre Norm不会削弱残差的作用，训练稳定性高，被用于Llama、GPT等主流模型

$$x_{l+1} = x_l + F(\text{Norm}(x_l))$$



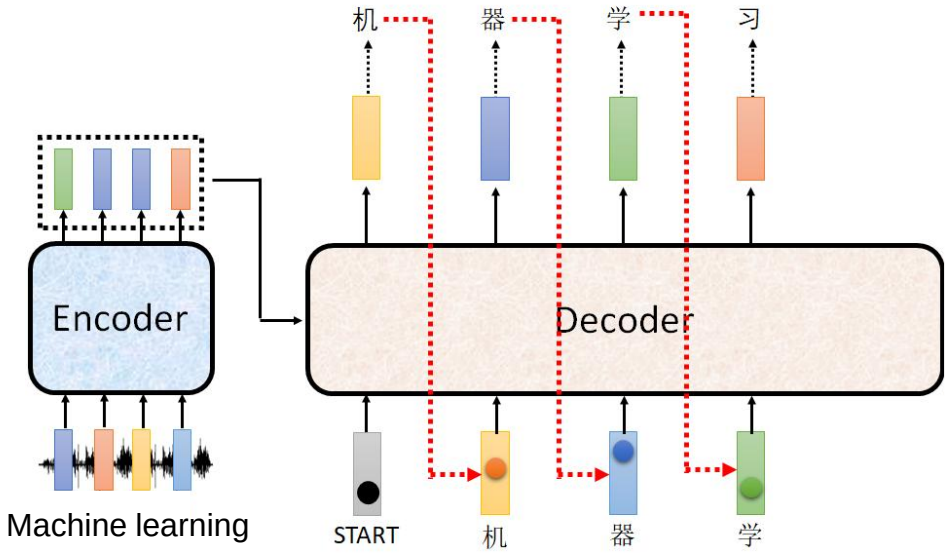
8.7.2 编码器

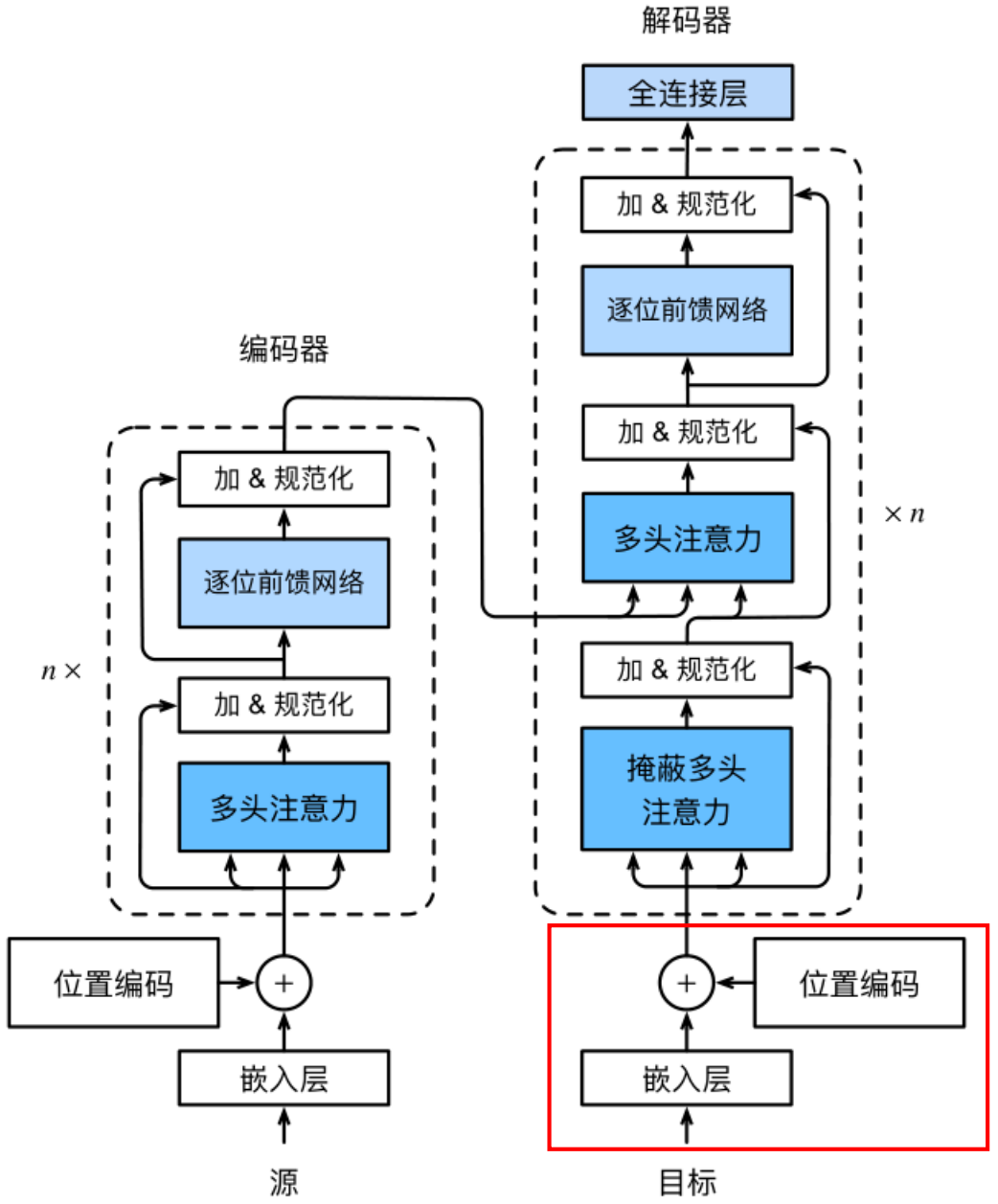
- 小结


编码器
1. Tokenization
2. Input Layer
3. Attention
4. Feed Forward
5. Output Layer
文字转为 Token
嵌入层
理解Token
理解上下文
学习不同token的
相关性
整合、思考 ...
得到输出
Transformer Block × N
反复理解思考

8.7.3 解码器

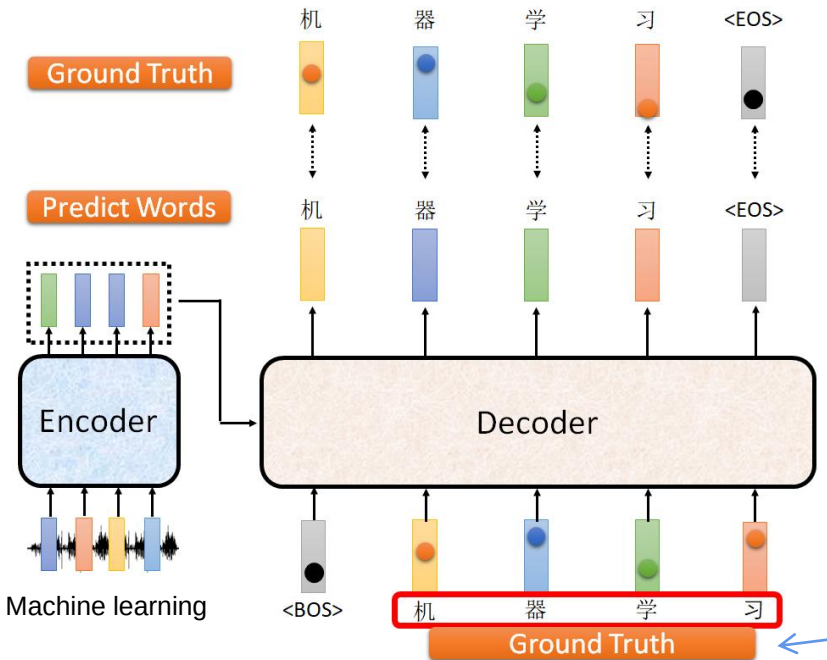
- 输出序列（目标）的词嵌入向量加上位置编码，输入到解码器中
  - 推理时，按顺序生成每一个词元：将上一时刻的预测输出作为下一时刻的输入，预测下一个输出词
  - **自回归**：生成当前词元时，会用到已经生成的全部词元

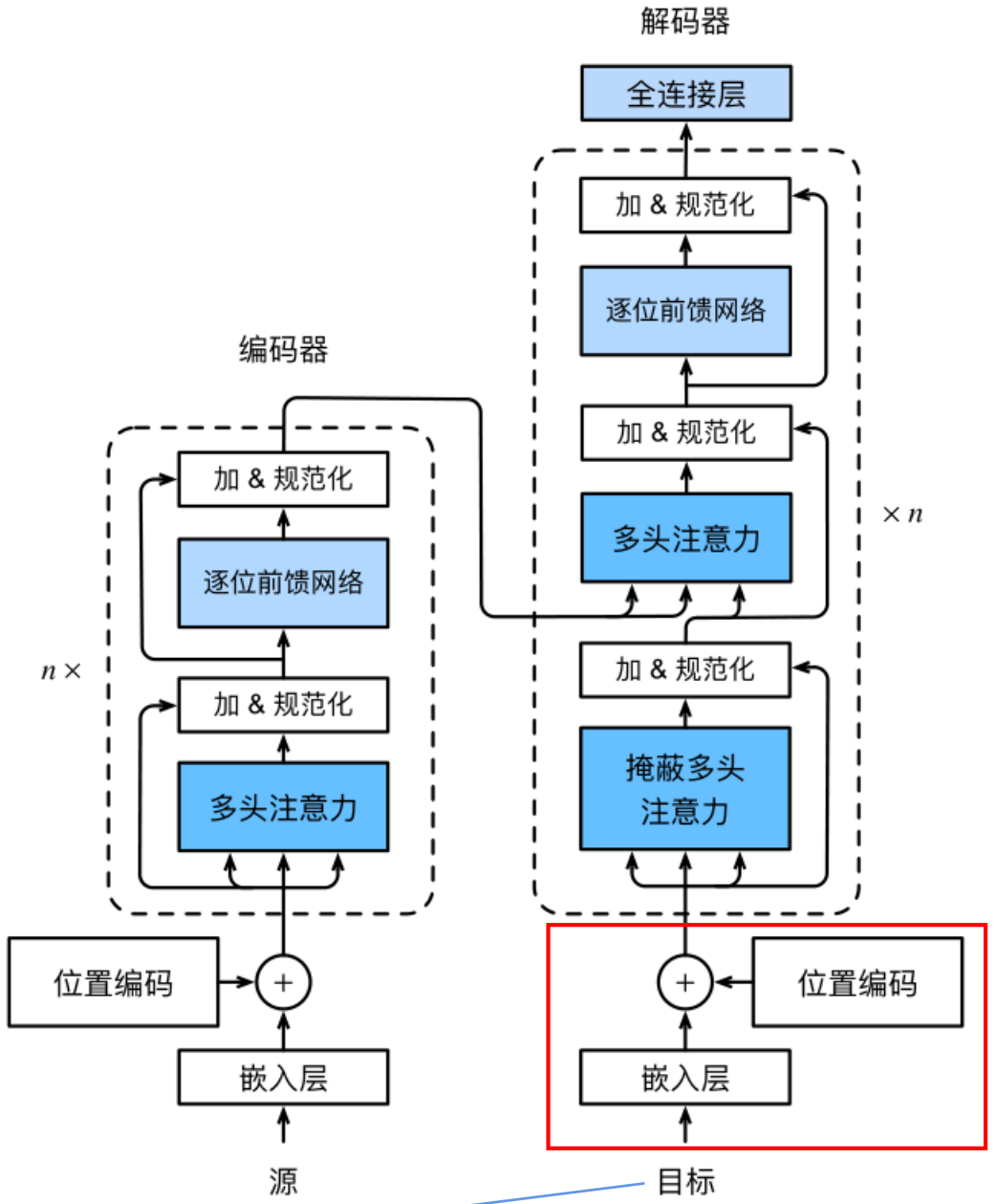




8.7.3 解码器

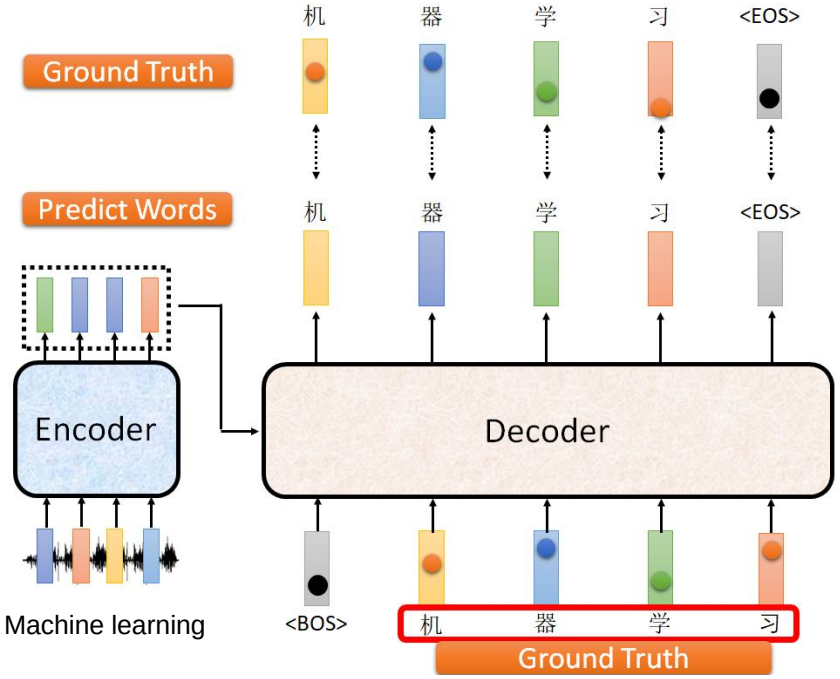
- 输出序列（目标）的词嵌入表示加上位置编码，输入到解码器中
  - 训练时，**使用真实输出作为当前时刻的输入**（Teacher Forcing模式）
    - 前一时刻的预测输出可能出错，一步错步步错
    - 使用真值作为输入，即看着标准答案学习





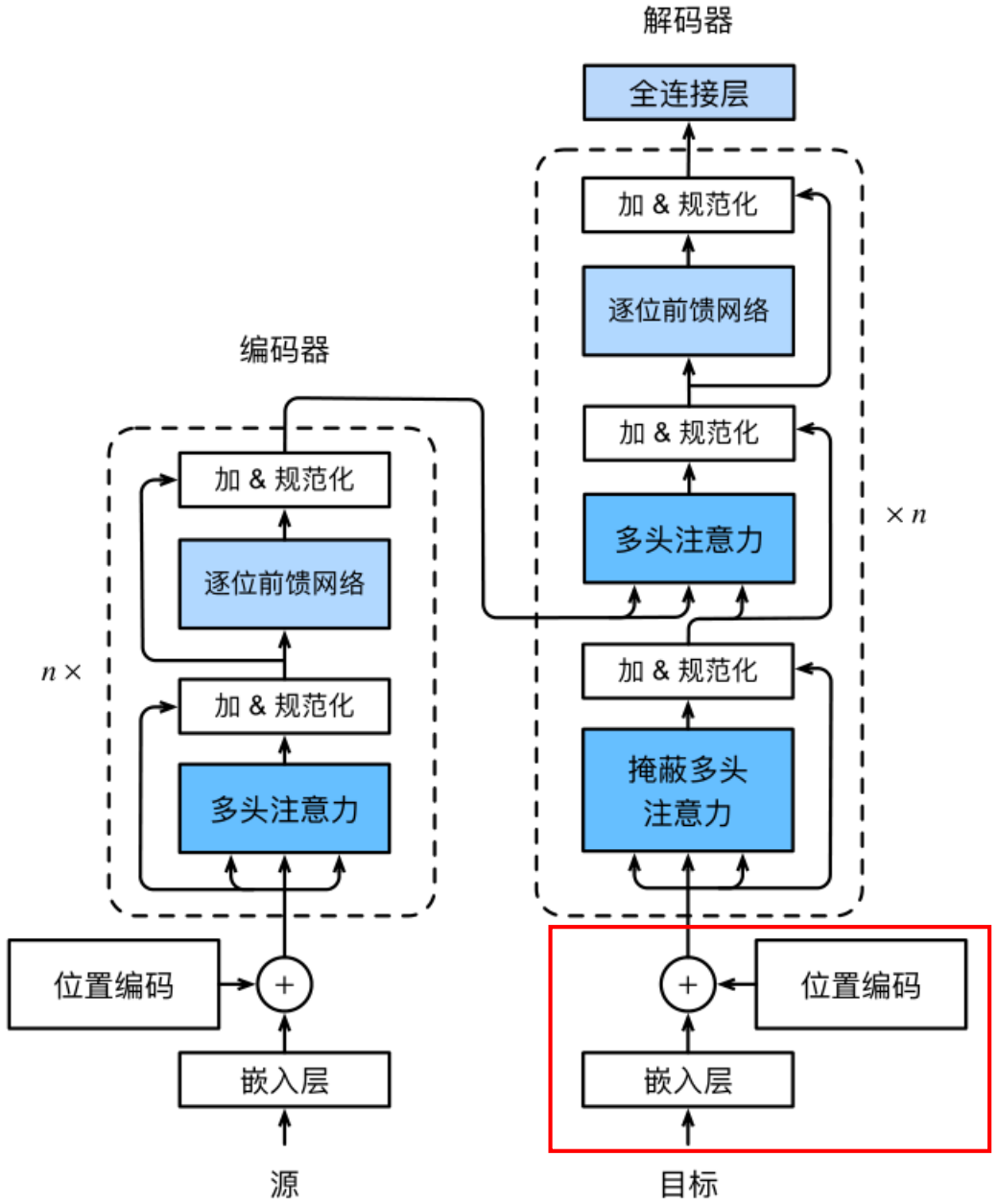
8.7.3 解码器

- 输出序列（目标）的词嵌入表示加上位置编码，输入到解码器中 去年考了
  - 训练时，**使用真实输出作为当前时刻的输入**
    - 使用模型预测作为输入，只能串行进行
    - 使用真值作为输入，方便并行训练



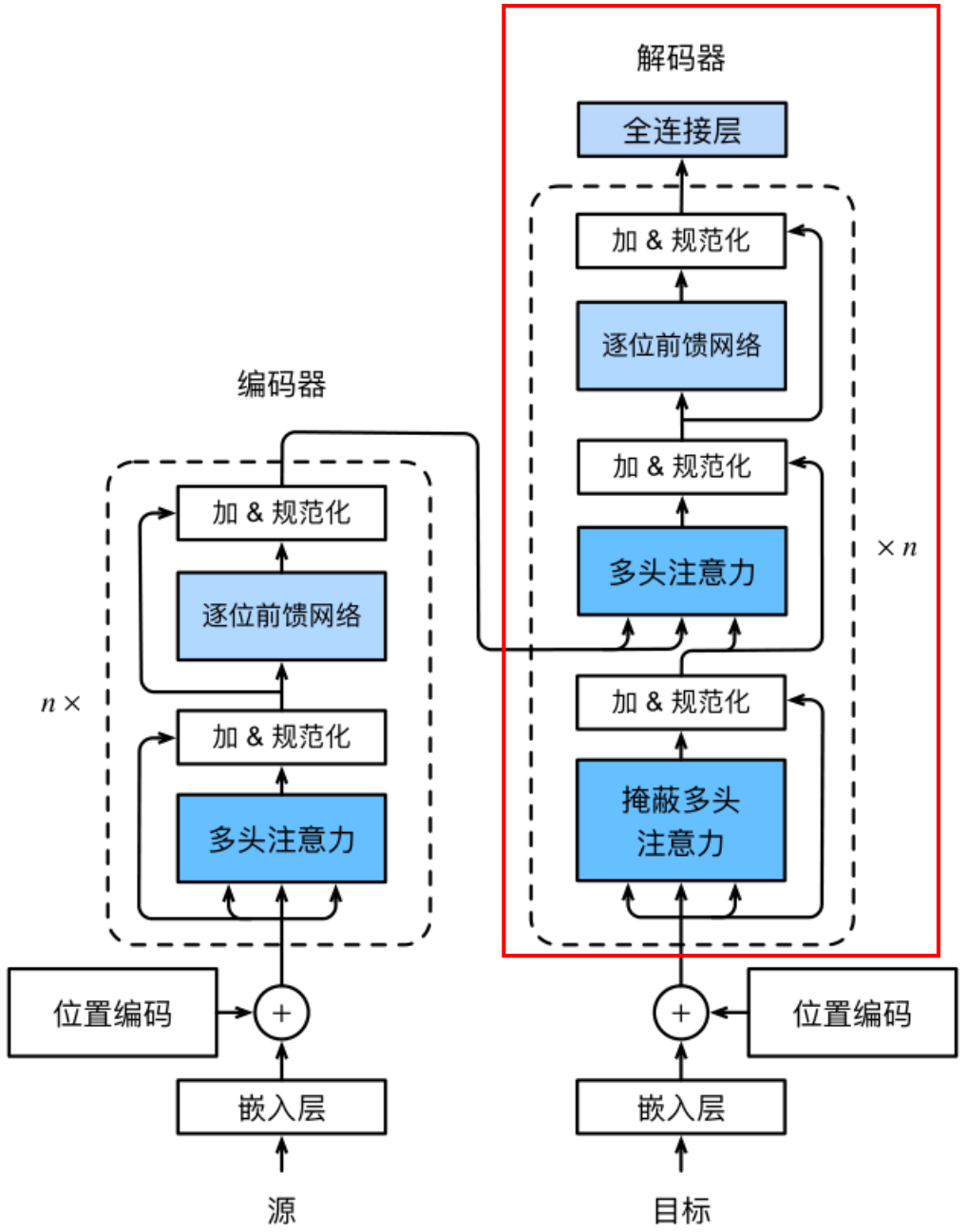
并行训练解释：

将真实输出序列一次性全部输入解码器（回顾8.3.3节矩阵形式的注意力计算），而非逐个输入，生成不同词元时互不影响，因此生成后面词元时不必等待前面词元生成完毕，故可以并行生成每个词元



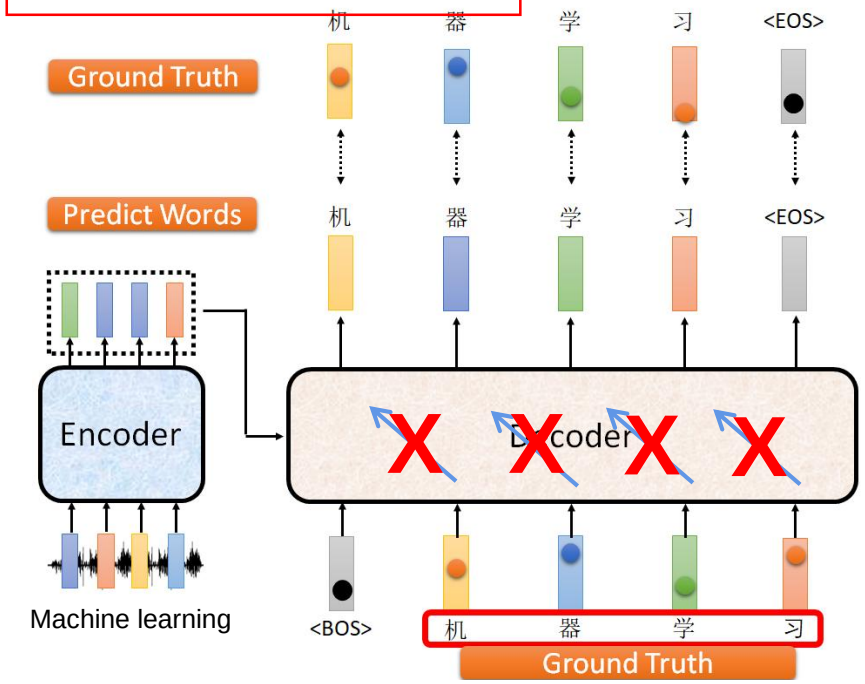
8.7.3 解码器

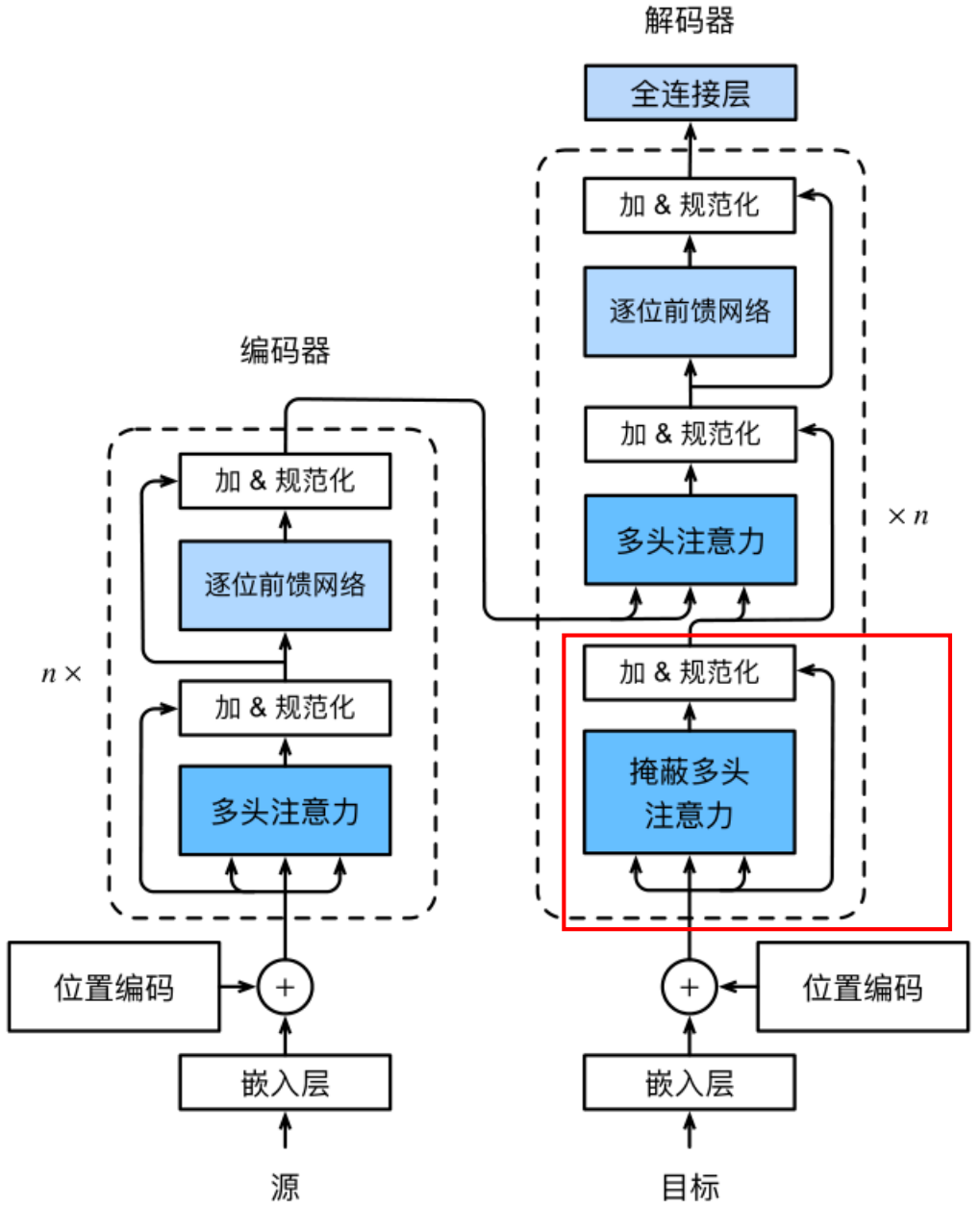
- 解码器负责生成目标语言序列
- 解码器由多个相同的层叠加而成的，每个层都有三个子层



8.7.3 解码器

- 第一个子层是**掩蔽多头自注意力**
  - **掩蔽**：训练时，所有的正确答案输入给解码器，解码每个词时只能使用之前生成的词，不能偷看后面的正确答案，掩蔽注意力将后面的正确答案掩蔽，确保每个输出仅依赖于前面已生成的输出词元
  - 作用：**并行训练、掩码后续输入**





8.7.3 解码器

- 第一个子层是**掩蔽多头自注意力**
  - 具体实现

$$O = V\text{softmax}\left(\frac{(K^TQ)\circ M}{\sqrt{d}}\right)$$

其中M为上三角0-1掩码矩阵，$\circ$表示按元素相乘

$$(K^TQ)\circ M$$

$$= \begin{bmatrix} k_1^Tq_1 & k_1^Tq_2 & \cdots & k_1^Tq_s \\ k_2^Tq_1 & k_2^Tq_2 & \cdots & k_2^Tq_s \\ \vdots & \vdots & \ddots & \vdots \\ k_s^Tq_1 & k_s^Tq_2 & \cdots & k_s^Tq_s \end{bmatrix} \circ \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 0 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

$$= \begin{bmatrix} k_1^Tq_1 & k_1^Tq_2 & \cdots & k_1^Tq_s \\ 0 & k_2^Tq_2 & \cdots & k_2^Tq_s \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & k_s^Tq_s \end{bmatrix}$$

softmax仅在未抹去的元素上按列进行归一化

$$\text{softmax}\left(\frac{(K^TQ)\circ M}{\sqrt{d}}\right) = \begin{bmatrix} \alpha_{1,1} & \alpha_{1,2} & \cdots & \alpha_{1,s} \\ 0 & \alpha_{2,2} & \cdots & \alpha_{2,s} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \alpha_{s,s} \end{bmatrix}$$

列和为1

$$V\text{softmax}\left(\frac{(K^TQ)\circ M}{\sqrt{d}}\right) = \begin{bmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,s} \\ v_{2,1} & v_{2,2} & \cdots & v_{2,s} \\ \vdots & \vdots & \ddots & \vdots \\ v_{s,1} & v_{s,2} & \cdots & v_{s,s} \end{bmatrix} \begin{bmatrix} \alpha_{1,1} & \alpha_{1,2} & \cdots & \alpha_{1,s} \\ 0 & \alpha_{2,2} & \cdots & \alpha_{2,s} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \alpha_{s,s} \end{bmatrix}$$

$$= \begin{bmatrix} v_{1,1} & \sum_{j=1}^{2}\alpha_{j,2}v_{1,j} & \cdots & \sum_{j=1}^{s}\alpha_{j,s}v_{1,j} \\ v_{2,1} & \sum_{j=1}^{2}\alpha_{j,2}v_{2,j} & \cdots & \sum_{j=1}^{s}\alpha_{j,s}v_{2,j} \\ \vdots & \vdots & \ddots & \vdots \\ v_{s,1} & \sum_{j=1}^{2}\alpha_{j,2}v_{s,j} & \cdots & \sum_{j=1}^{s}\alpha_{j,s}v_{s,j} \end{bmatrix}$$

每一列理解为一个词的向量表示

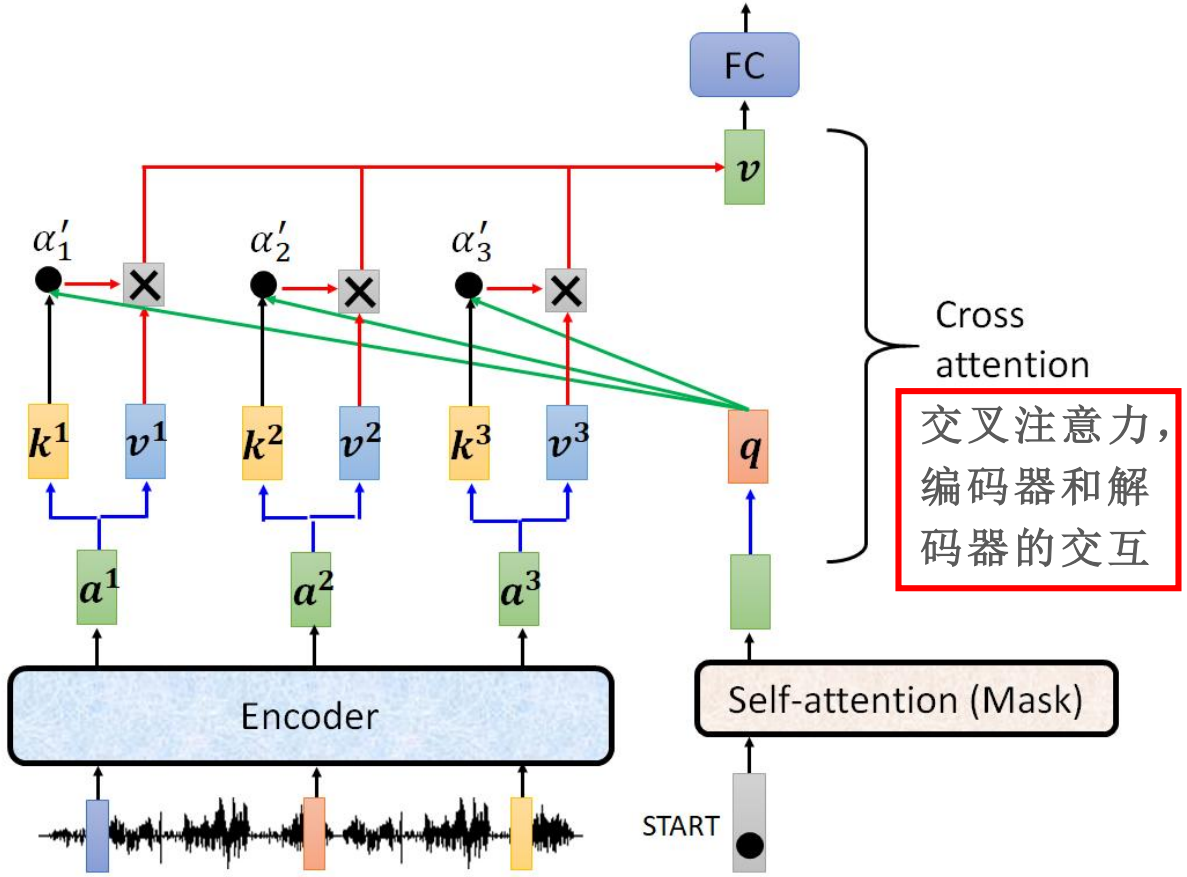
输出O的第一列时只看到V的第一列

输出O的第二列时只看到V的前两列

输出O的最后一列时看到V的所有列

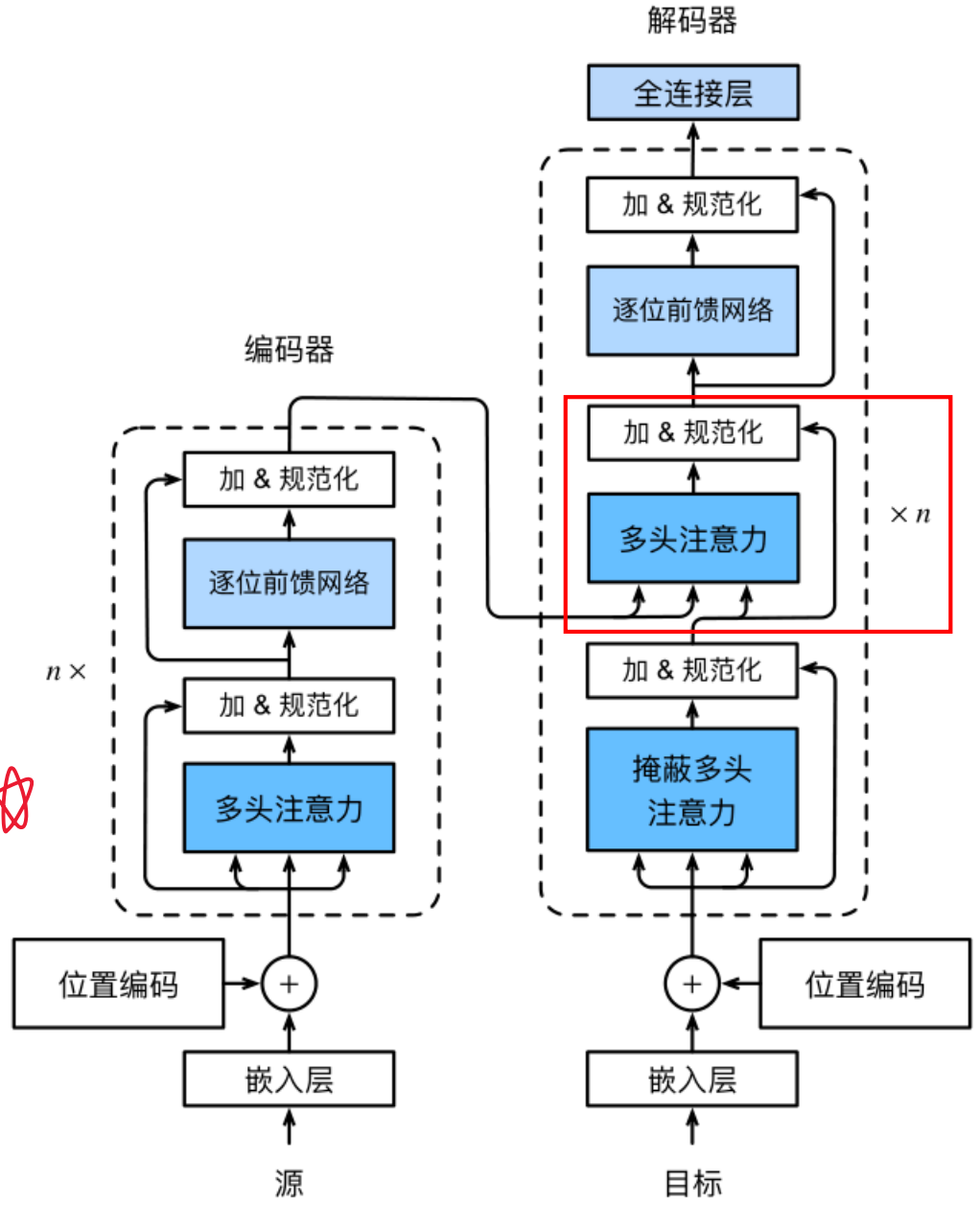
8.7.3 解码器

- 第二个子层是**编码器－解码器注意力层**
  - 查询来自前一个解码器子层的输出
  - 键和值来自编码器最后一层的输出



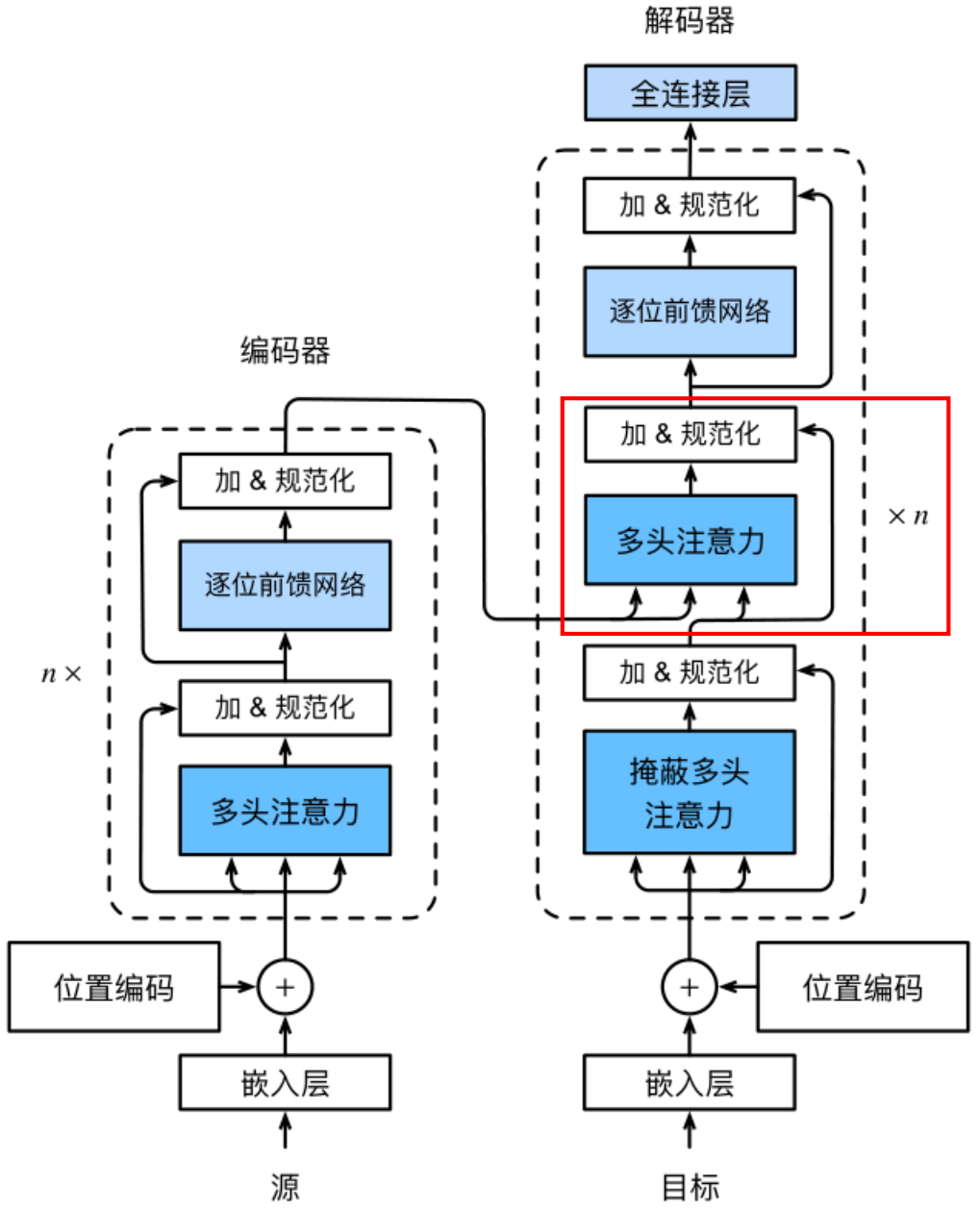
Cross attention

交叉注意力，编码器和解码器的交互



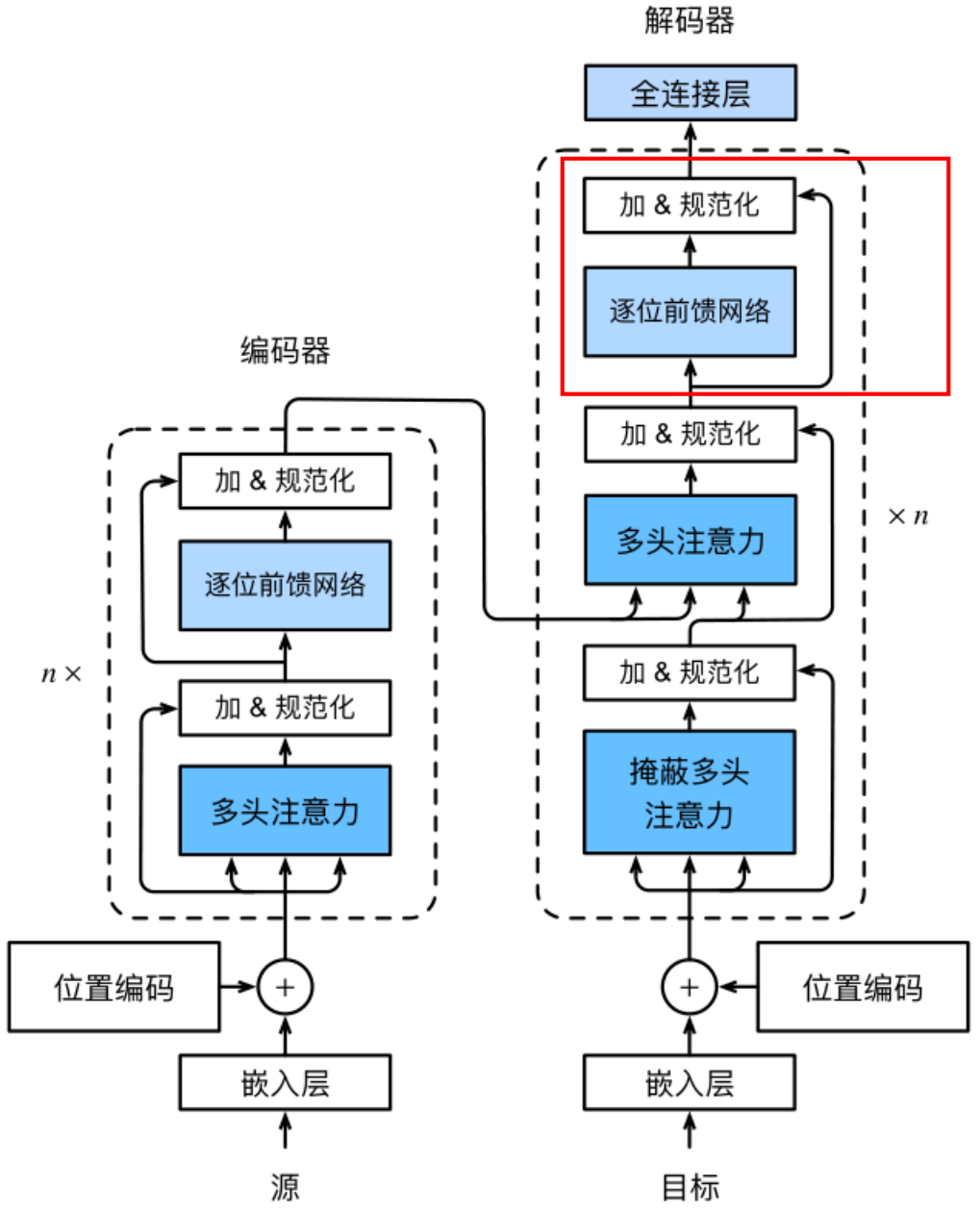
8.7.3 解码器

- 第二个子层是**编码器－解码器注意力层**
  - 作用：解码器产生输入序列的内部编码（即理解输入序列），编码器基于该内部编码产生输出（即产生输出序列），**交叉注意力构建起了编码器到解码器之间的信息桥梁**



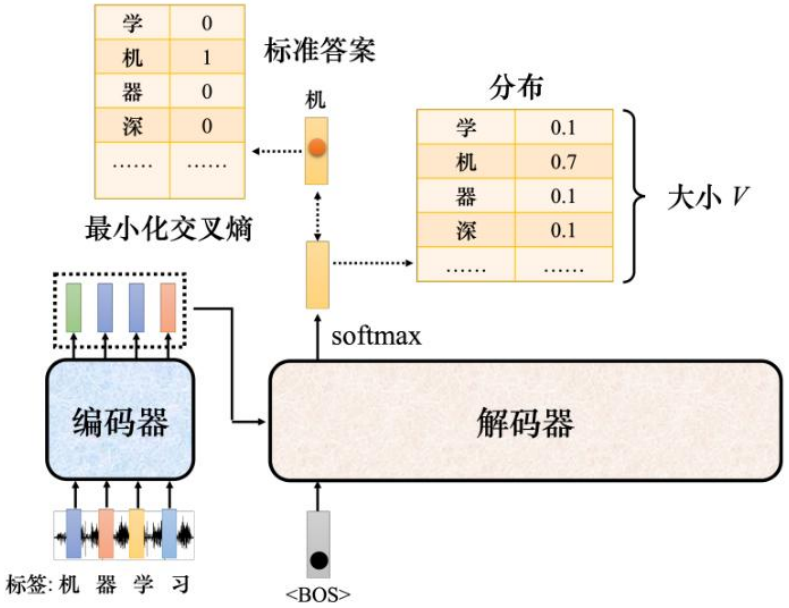
8.7.3 解码器

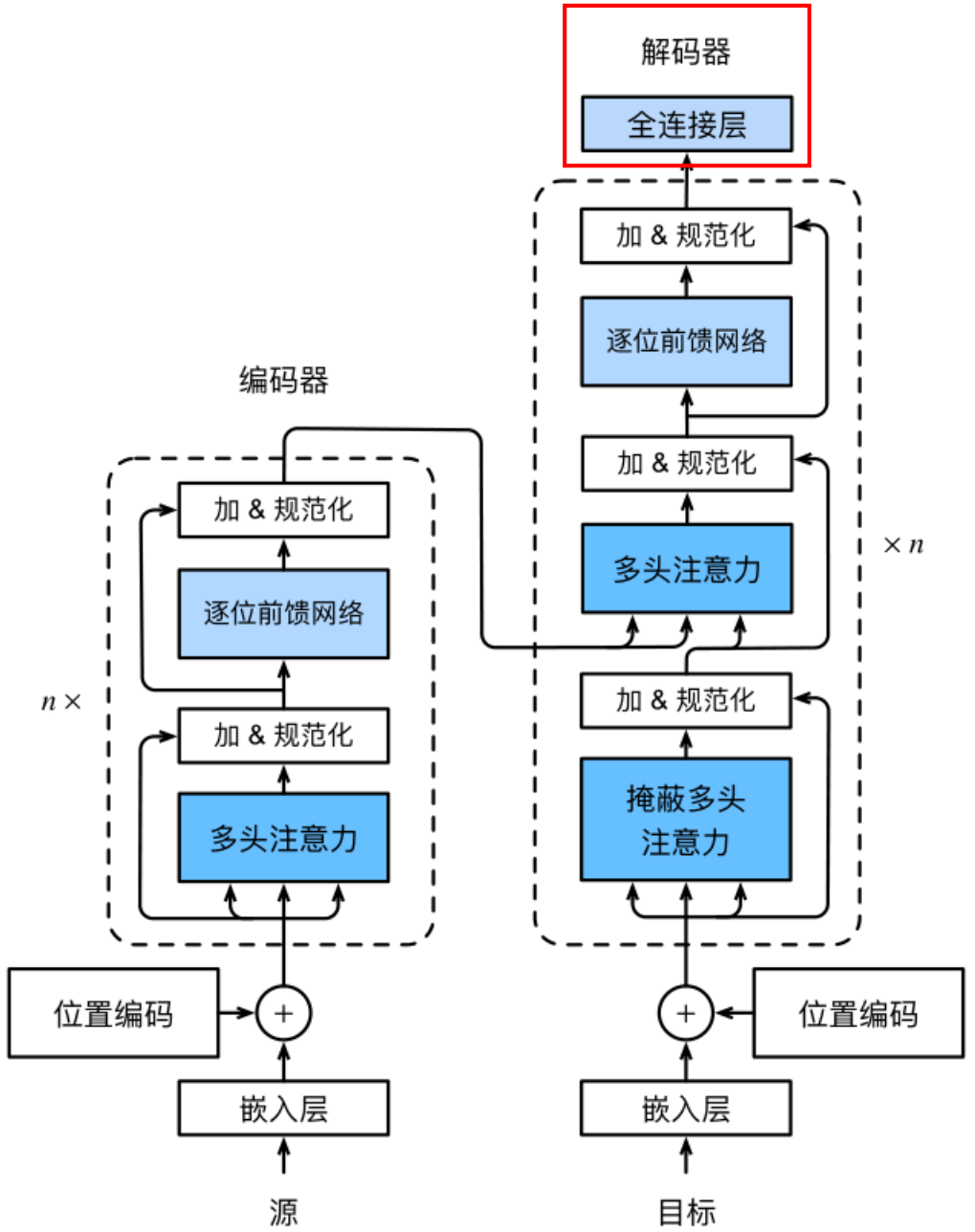
- 第三个子层是逐位置前馈网络
  - 同编码器第二个子层
- 每个子层都使用了残差连接和规范化



8.7.4 输出

- 全连接层将解码器的输出维度映射到词表大小，并做softmax变换，判断每个位置输出哪一个词的概率最大





8.7.5 Transformer优缺点

Transformer	
性能强大的原因	**缺点**
自注意力机制（Self-Attention Mechanism）	数据要求高
并行计算	
层次化表示	解释性差
位置编码（Positional Encoding）	
大规模训练数据	处理长序列时计算开销高
优化技巧	

8.7.6 Transformer模型大小

Model	Layers	Width	Heads	Params	Data	Training
Transformer-Base	12	512	8	65M		8x P100 (12 hours)
Transformer-Large	12	1024	16	213M		8x P100 (3.5 days)
BERT-Base	12	768	12	110M	13 GB	
BERT-Large	24	1024	16	340M	13 GB	
XLNet-Large	24	1024	16	~340M	126 GB	512x TPU-v3 (2.5 days)
RoBERTa	24	1024	16	355M	160 GB	1024x V100 GPU (1 day)
GPT-2	48	1600	?	1.5B	40 GB	
Megatron-LM	72	3072	32	8.3B	174 GB	512x V100 GPU (9 days)
Turing-NLG	78	4256	28	17B	?	256x V100 GPU
GPT-3	96	12288	96	175B	694GB	?

M：Million，百万

B：Billion，十亿

8.7.6 Transformer模型大小

模型	类别	大小	归一化	位置编码	激活函数	L	N	H
GPT-3	因果	175B	Pre Layer	Learned	GELU	96	96	12288
PanGU-α	因果	207B	Pre Layer	Learned	GELU	64	128	16384
OPT	因果	175B	Pre Layer	Learned	ReLU	96	96	12288
PaLM	因果	540B	Pre Layer	RoPE	SwiGLU	118	48	18432
BLOOM	因果	176B	Pre Layer	ALiBi	GELU	70	112	14336
MT-NLG	因果	530B	-	-	-	105	128	20480
Gopher	因果	280B	Pre RMS	Relative	-	80	128	16384
Chinchilla	因果	70B	Pre RMS	Relative	-	80	64	8192
Galactica	因果	120B	Pre Layer	Learned	GELU	96	80	10240
LaMDA	因果	137B	-	Relative	GeGLU	64	128	8192
Jurassic-1	因果	178B	Pre Layer	Learned	GELU	76	96	13824
LLaMA-2	因果	70B	Pre RMS	RoPE	SwiGLU	80	64	8192
Pythia	因果	12B	Pre Layer	RoPE	GELU	36	40	5120
Baichuan-2	因果	13B	Pre RMS	ALiBi	SwiGLU	40	40	5120
Qwen-1.5	因果	72B	Pre RMS	RoPE	SwiGLU	80	64	8192
InternLM-2	因果	20B	Pre RMS	RoPE	SwiGLU	48	48	6144
Falcon	因果	180B	Pre Layer	RoPE	GELU	80	232	14848
MPT	因果	30B	Pre Layer	ALiBi	GELU	48	64	7168
Mistral	因果	7B	Pre RMS	RoPE	SwiGLU	32	32	4096
Gemma	因果	7B	Pre RMS	RoPE	GELU	28	16	3072
DeepSeek	因果	67B	Pre RMS	RoPE	SwiGLU	95	64	8192
Yi	因果	34B	Pre RMS	RoPE	SwiGLU	60	56	7168
YuLan	因果	12B	Pre RMS	RoPE	SwiGLU	40	38	4864
GLM-130B	前缀	130B	Post Deep	RoPE	GeGLU	70	96	12288
T5	编-解	11B	Pre RMS	Relative	ReLU	24	128	1024

L：层数

N：注意力头数

H：隐含状态维度

因果：因果解码器，使用了掩码注意力，只能关注当前输出位置之前的输出位置；无编码器，无交叉注意力

Pre：Pre Norm

Layer：LayerNorm

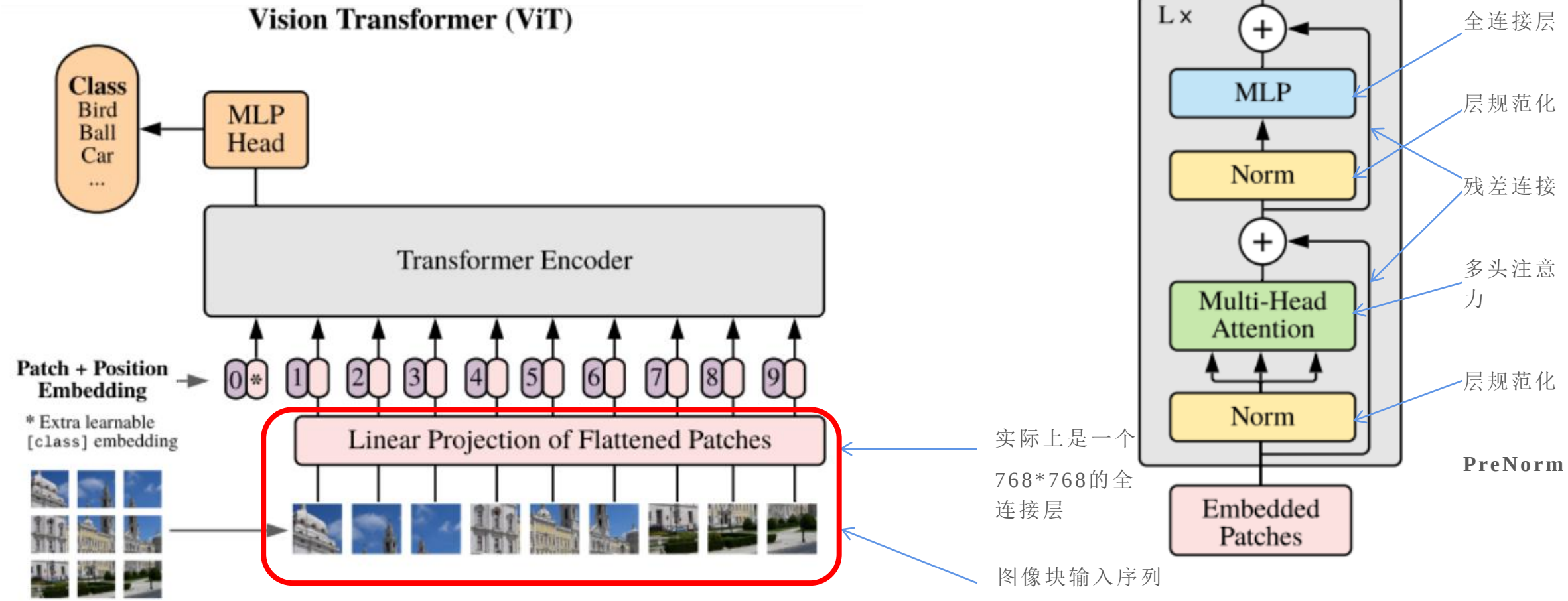
RMS：RMS-Norm

RoPE：旋转位置编码

8.8 Vision Transformer

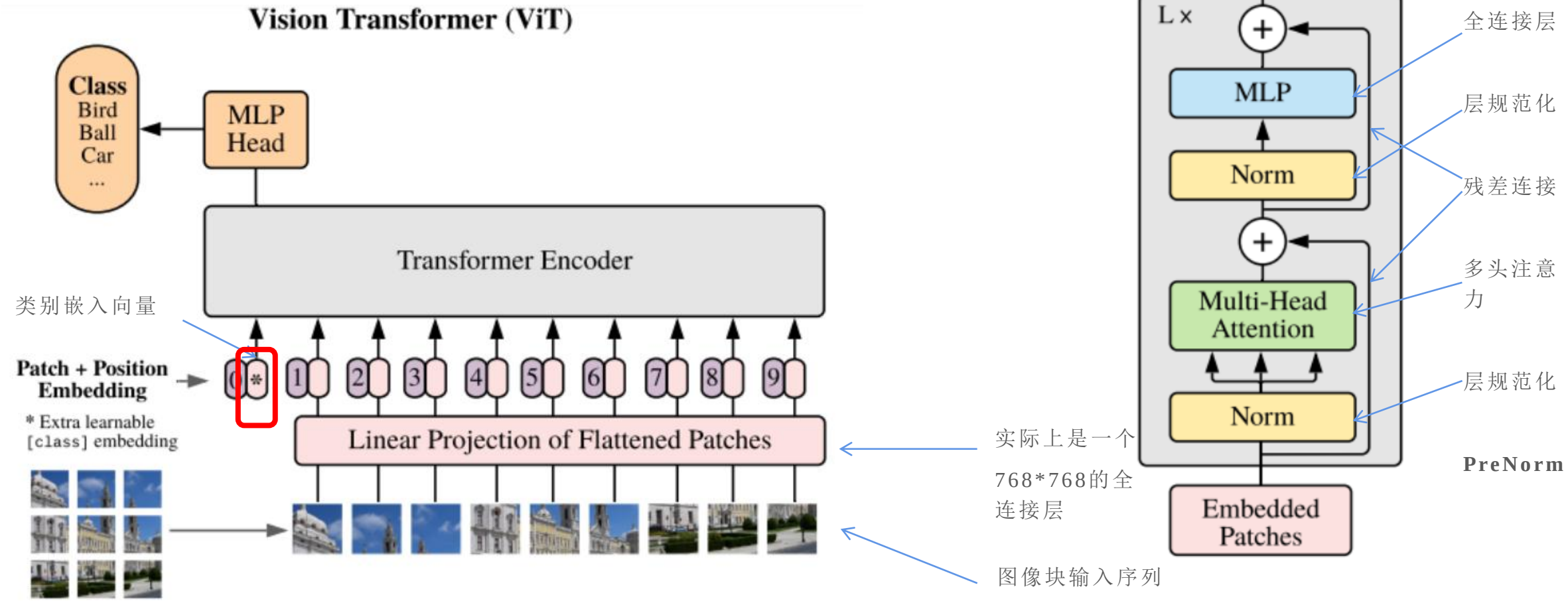
- Vision Transformer（ ViT ），是面向图像处理的 Transformer，由谷歌技术团队于 2020 年提出
- ViT主体部分基于Transformer编码器，主要改变在输入端，将图像进行分块和降维，从而将图像变成一种类似于词编码的表达方式，方便后续处理
- ViT由嵌入层、Transformer编码器层、MLP Head组成

8.8 Vision Transformer



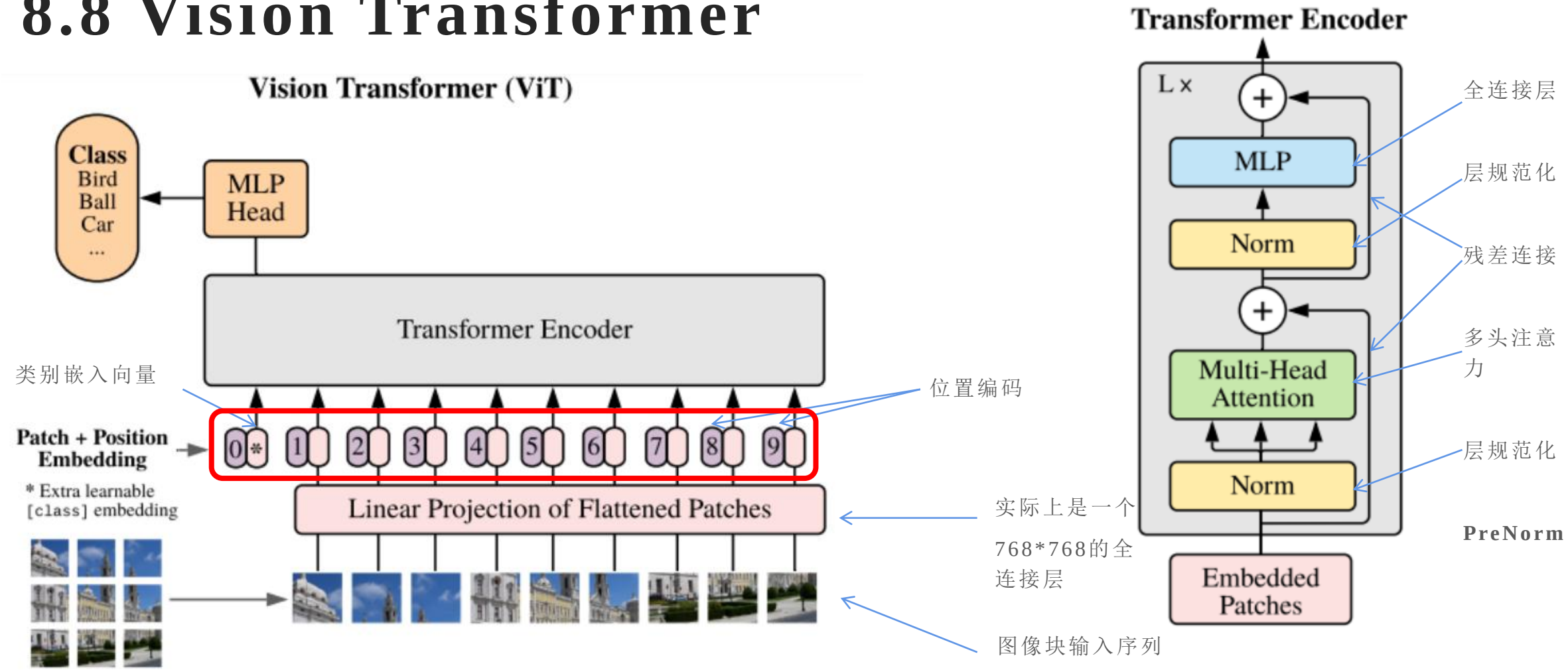
Step 1：对于尺寸为 224*224*3 的输入图像，将输入图像划分成 196 个 16*16*3 的图像块，将每一个图像块扁平化为向量（长度为 16*16*3=768），线性投影得到编码向量，将所有 196 个编码向量放在一起形成输入序列

8.8 Vision Transformer



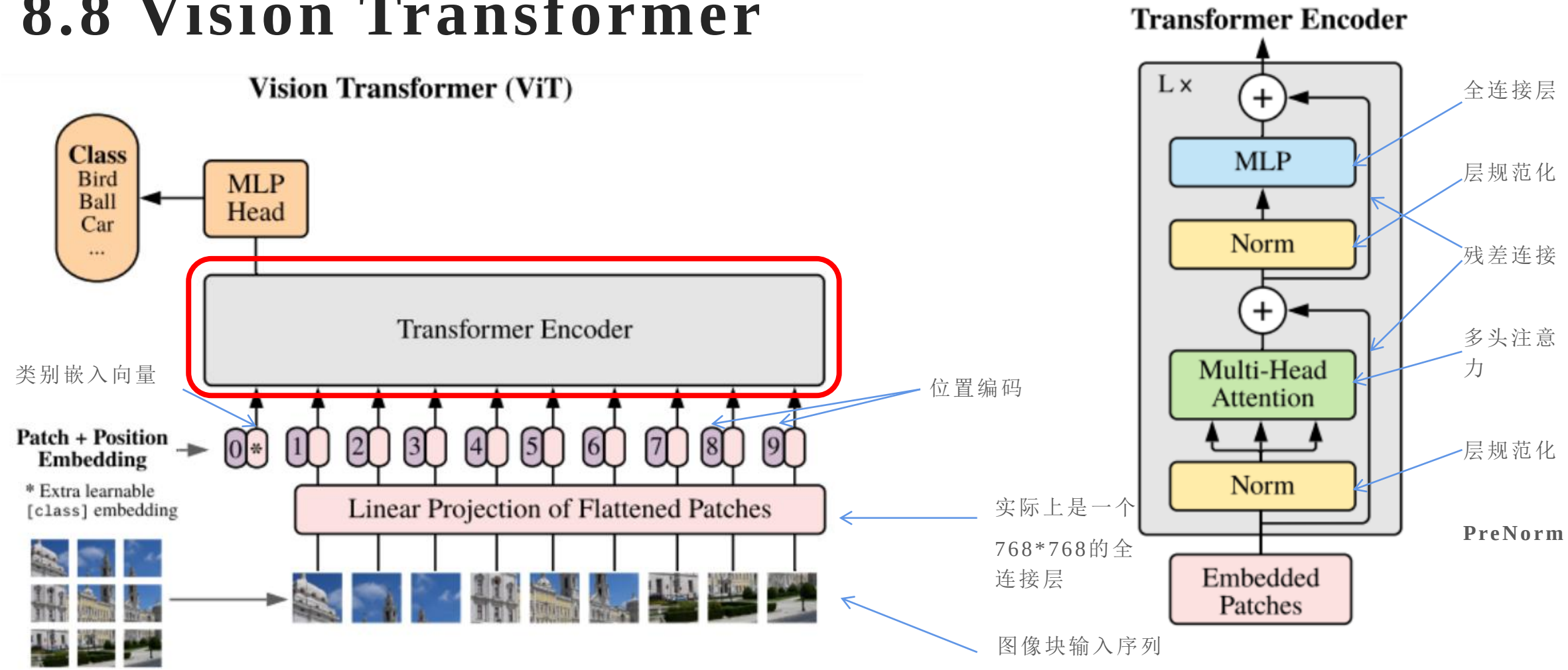
Step 2：定义一个**类别嵌入向量**，长度也为 768，该向量跟前面的编码向量放在一起，而且放在序列的最左边（索引值为 0），其作用相当于ViT 模型的输出向量，即模型收敛后该向量的值刻画了整个序列的特征

8.8 Vision Transformer



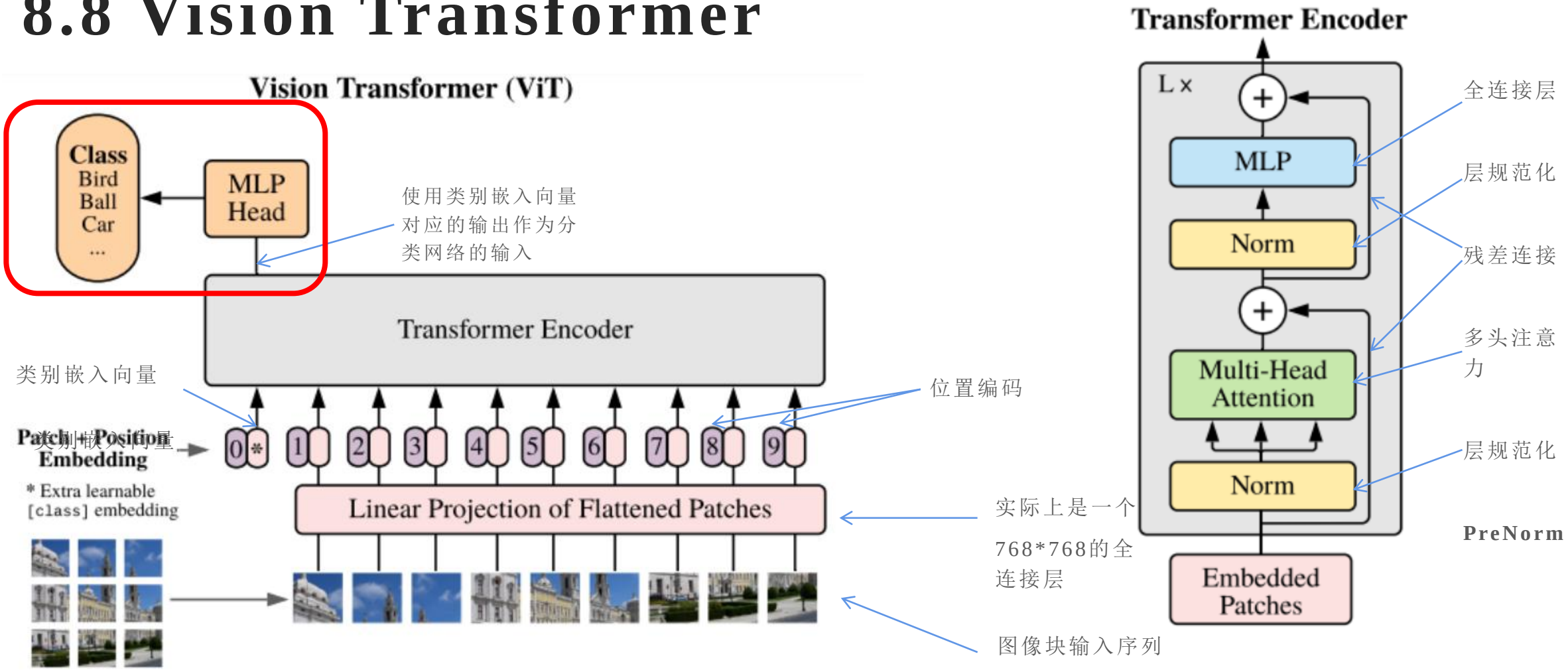
Step 3：定义 197 个位置嵌入向量，与前面的 197 个编码向量分别相加，送入Transformer

8.8 Vision Transformer



Step 4：使用Transformer模型处理输入，这里使用了PreNorm模式

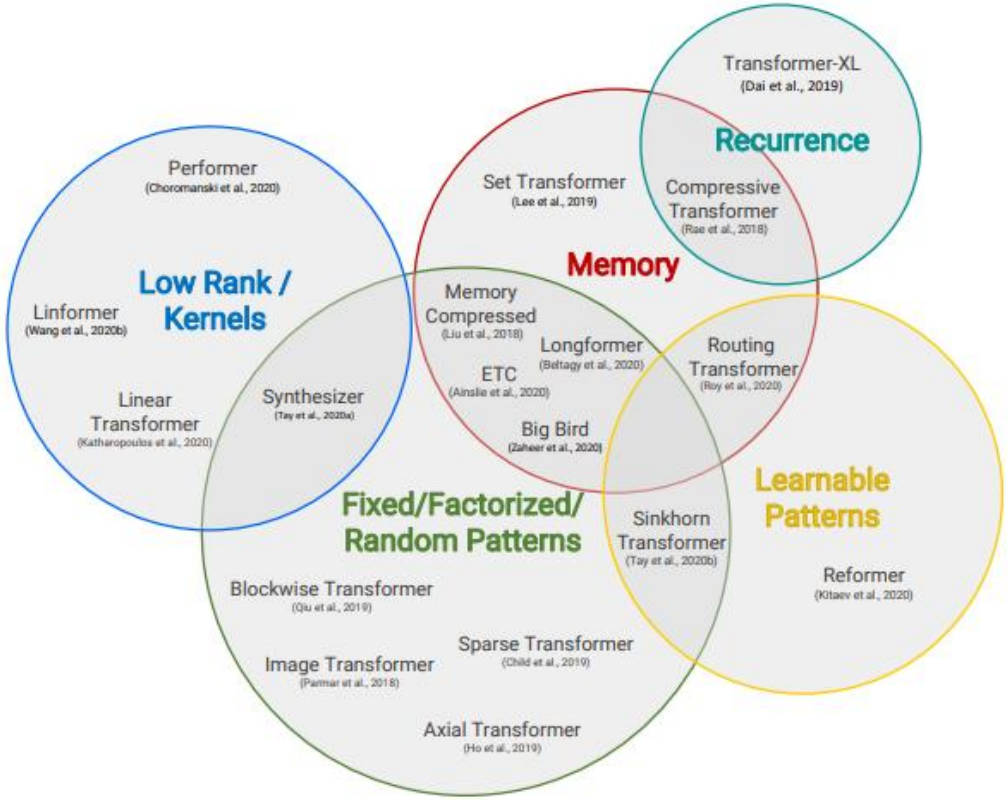
8.8 Vision Transformer



Step 5：在 Transformer 模型的输出端构建一个分类网络（即全连接网络），将类别嵌入向量对应的输出作为分类网络的输入

8.9 Efficient Transformer

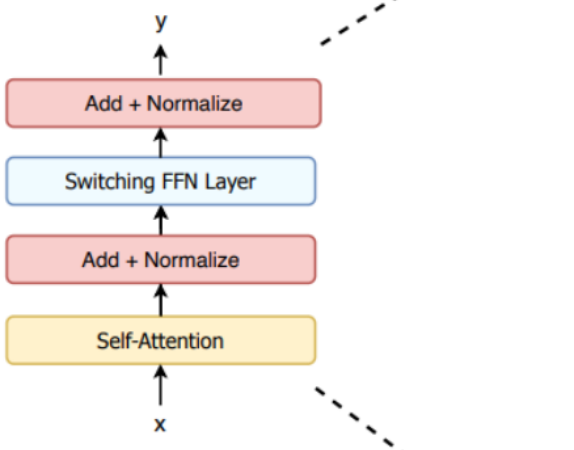
- 自注意力复杂度：$O(n^2)$
  - 当序列长度 n很大时，Transformer的主要计算开销来自于自注意力
    - 在图像处理中，将 224*224*3 的图像划分成 196 个 16*16*3 的图像块，序列长度为196，序列每个位置为 16*16*3 的向量
    - GPT-3.5的上下文达到2048个Token
    - 目前最强大的GPT-4模型的上下文已经达到12.8万个Token
- 各种高效Transformer设计，但性能都略有下降
- FlashAttention：在硬件上实现加速，计算上与传统注意力等价，性能不会下降



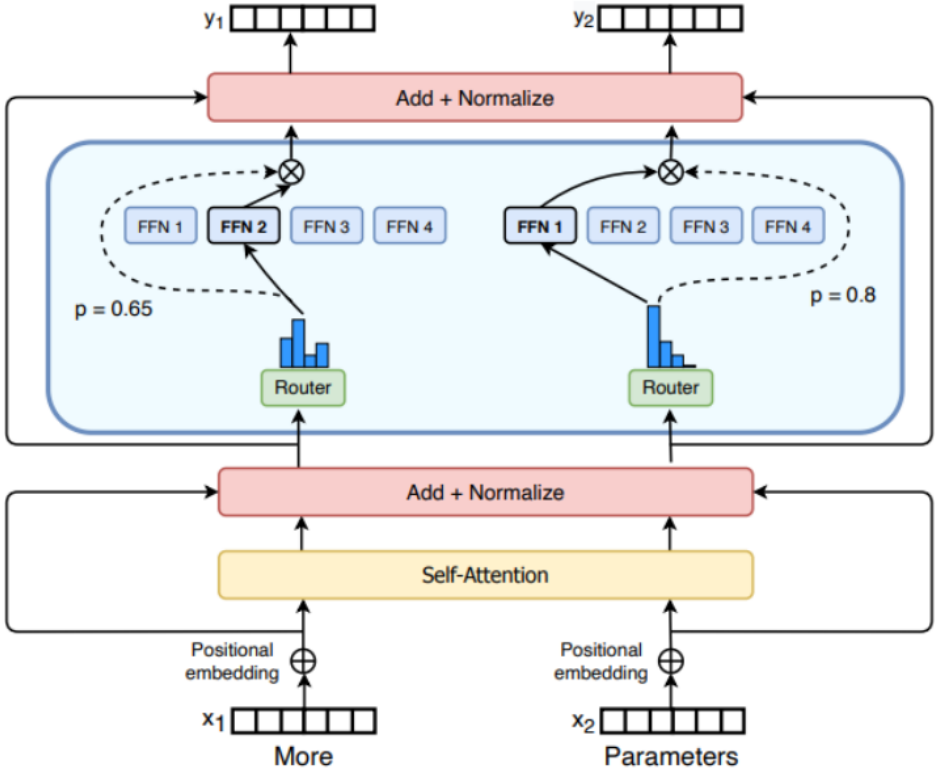
8.10 混合专家模型

什么是

- 混合专家模型 (Mixed Expert Models，MoE)旨在通过多个专家的协同工作来提升模型的预测效果
  - 思想：每个专家都专注于特定的领域知识，让不同的输入选择最合适的专家
- MoE包含两个部分：稀疏MoE层和路由
  - 稀疏 MoE 层
    - 代替了传统 Transformer 中的前馈网络 (FFN) 层
    - MoE 层包含若干“专家”，每个专家本身是一个独立的神经网络
  - 路由（Router）
    - 用于决定哪些Token 被发送到哪个专家
    - 负载均衡：让专家负载保持均衡

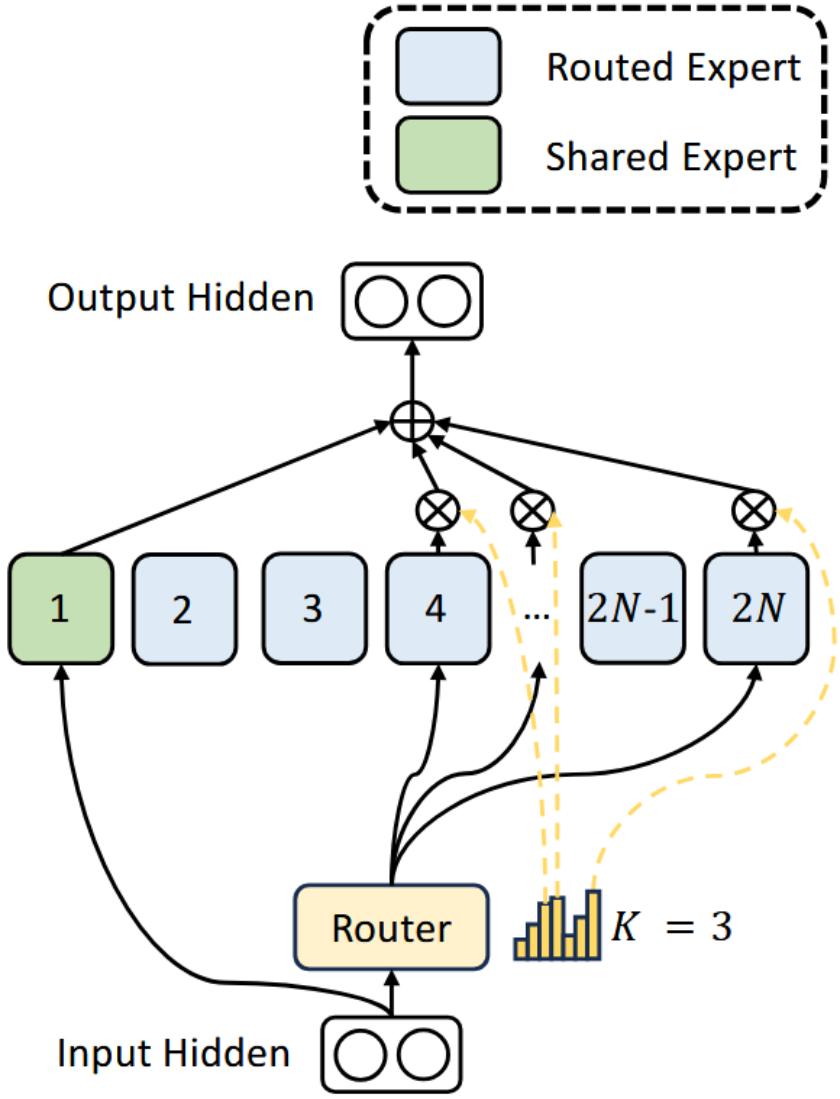


Switch Transformers



8.10 混合专家模型

- Deepseek MOE
  - 引入共享专家，所有 tokens 都会经过的共享专家
  - 每个 token 会用计算的 Router 权重，来选择 top K个专家，然后和共享专家的输出一起加权求和
  - 减少不同专家间的知识冗余，提升计算效率
  - 采集大数量小尺寸的MOE设计，将每个专家的粒度变细
    - DeepSeek-V1：64个路由专家+2个共享专家
    - DeepSeek-V2：160个路由专家+2个共享专家
    - DeepSeek-V3：256个路由专家+1个共享专家



总结

- 介绍了注意力机制
  - 自注意力、多头注意力、位置掩码
- 介绍了Transformer架构
  - 编码器、解码器
  - Transformer中的三种注意力：编码器中的标准多头自注意力、解码器中的掩蔽多头自注意力、编码器-解码器交叉注意力
- 参考：李宏毅老师B站授课视频
  - https://www.bilibili.com/video/BV1ou411N7X3/?p=49