```
class LlamaDecoderLayer(nn.Module):
    def __init__(self, config: LlamaConfig):
        super().__init__()
        self.hidden_size = config.hidden_size
        self.self_attn = LlamaAttention(config=config)
        self.mlp = LlamaMLP(
            hidden_size=self.hidden_size,
            intermediate_size=config.intermediate_size,
            hidden_act=config.hidden_act,
        )
        self.input_layernorm = LlamaRMSNorm(config.hidden_size, eps=config.rms_norm_eps)
        self.post_attention_layernorm = LlamaRMSNorm(config.hidden_size, eps=config.rms_norm_eps)

    def forward(
        self,
        hidden_states: torch.Tensor,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        past_key_value: Optional[Tuple[torch.Tensor]] = None,
        output_attentions: Optional[bool] = False,
        use_cache: Optional[bool] = False,
    ) -> Tuple[torch.FloatTensor, Optional[Tuple[torch.FloatTensor, torch.FloatTensor]]]:

        residual = hidden_states
        hidden_states = self.input_layernorm(hidden_states)

        # 自注意力模块
        hidden_states, self_attn_weights, present_key_value = self.self_attn(
            hidden_states=hidden_states,
            attention_mask=attention_mask,
            position_ids=position_ids,
            past_key_value=past_key_value,
            output_attentions=output_attentions,
            use_cache=use_cache,
        )
        hidden_states = residual + hidden_states

        # 全连接层
        residual = hidden_states
        hidden_states = self.post_attention_layernorm(hidden_states)
        hidden_states = self.mlp(hidden_states)
        hidden_states = residual + hidden_states

        outputs = (hidden_states,)

        if output_attentions:
            outputs += (self_attn_weights,)

        if use_cache:
            outputs += (present_key_value,)
```

2.3.2 注意力机制优化

在 Transformer 结构中，自注意力机制的时间和存储复杂度与序列的长度呈平方的关系，因此占用了大量的计算设备内存并消耗了大量的计算资源。如何优化自注意力机制的时空复杂度、增强计算效率是大语言模型面临的重要问题。一些研究从近似注意力出发，旨在减少注意力计算和内存需求，提出了稀疏近似、低秩近似等方法。此外，有一些研究从计算加速设备本身的特性出发，研究如何更好地利用硬件特性对 Transformer 中的注意力层进行高效计算。本节将分别介绍上述方法。

1. 稀疏注意力机制

对一些训练好的 Transformer 结构中的注意力矩阵进行分析时发现，其中很多是稀疏的，因此可以通过限制 Query-Key 对的数量来降低计算复杂度。这类方法称为**稀疏注意力**（Sparse Attention）机制。可以将稀疏化方法进一步分成基于位置的和基于内容的两类。

基于位置的稀疏注意力机制的基本类型如图2.6 所示，主要包含如下五种类型。

（1）全局注意力（Global Attention）：为了增强模型建模长距离依赖关系的能力，可以加入一些全局节点。

（2）带状注意力（Band Attention）：大部分数据都带有局部性，限制 Query 只与相邻的几个节点进行交互。

（3）膨胀注意力（Dilated Attention）：与 CNN 中的 Dilated Conv 类似，通过增加空隙获取更大的感受野。

（4）随机注意力（Random Attention）：通过随机采样，提升非局部的交互能力。

（5）局部块注意力（Block Local Attention）：使用多个不重叠的块（Block）来限制信息交互。

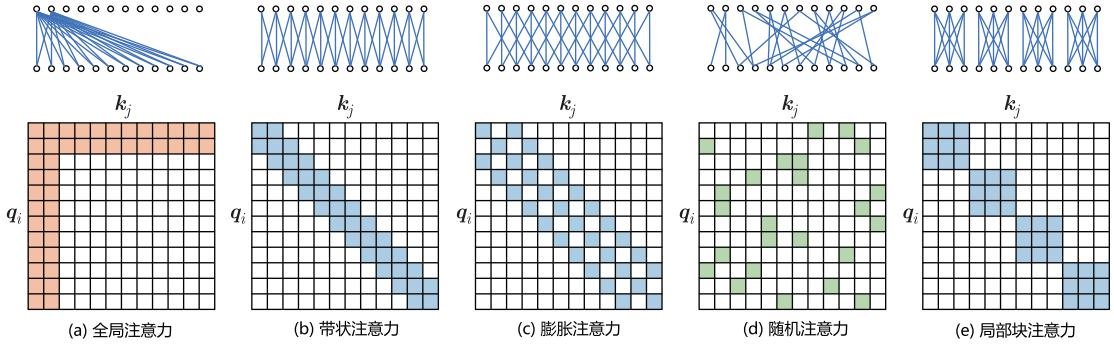

图 2.6 五种基于位置的稀疏注意力机制[49]

现有的稀疏注意力机制，通常是基于上述五种基于位置的稀疏注意力机制的复合模式，图2.7 给出了一些典型的稀疏注意力模型。Star-Transformer[50] 使用带状注意力和全局注意力。具体来说，

Star-Transformer 只包括一个全局注意力节点和宽度为 3 的带状注意力，其中任意两个非相邻节点通过一个共享的全局注意力连接，相邻节点则直接相连。Longformer[51] 使用带状注意力和内部全局节点注意力（Internal Global-node Attention）。此外，Longformer 将上层中的一些带状注意力头部替换为具有膨胀窗口的注意力，在增加感受野的同时并不增加计算量。ETC（Extended Transformer Construction）[52] 使用带状注意力和外部全局节点注意力（External Global-node Attention）。ETC 稀疏注意力还包括一种掩码机制来处理结构化输入，并采用对比预测编码（Contrastive Predictive Coding，CPC）[53] 进行预训练。BigBird[54] 使用带状注意力和全局注意力，并使用额外的随机注意力来近似全连接注意力。此外，BigBird 揭示了稀疏编码器和稀疏解码器的使用可以模拟任何图灵机，这也在一定程度上解释了为什么稀疏注意力模型可以取得较好的结果。

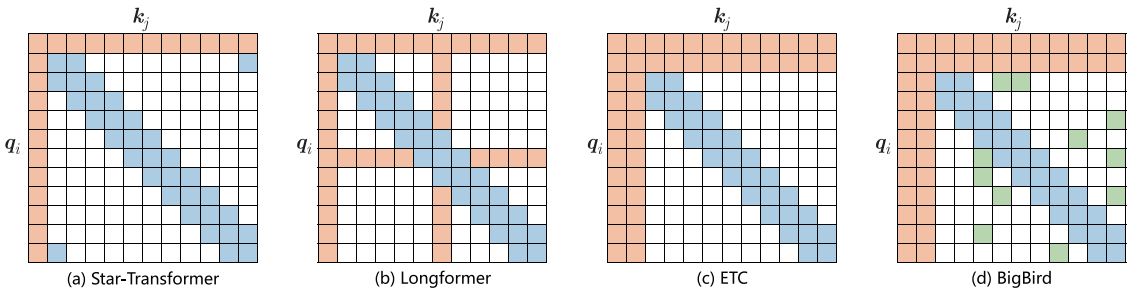

图 2.7　典型的稀疏注意力模型[49]

基于内容的稀疏注意力机制根据输入数据创建稀疏注意力，其中一种很简单的方法是选择和给定查询（Query）有很高相似度的键（Key）。Routing Transformer[55] 采用 K-means 聚类方法，针对 Query$\{\boldsymbol{q}_i\}_{i=1}^T$ 和 Key$\{\boldsymbol{k}_i\}_{i=1}^T$ 进行聚类，类中心向量集合为 $\{\boldsymbol{\mu}_i\}_{i=1}^k$，其中 k 是类中心的个数。每个 Query 只与其处在相同簇（Cluster）下的 Key 进行交互。中心向量采用滑动平均的方法进行更新：

$$\widetilde{\boldsymbol{\mu}} \leftarrow \lambda\widetilde{\boldsymbol{\mu}} + (1-\lambda)\left(\sum_{i:\mu(\boldsymbol{q}_i)=\boldsymbol{\mu}} \boldsymbol{q}_i + \sum_{j:\mu(\boldsymbol{k}_j)=\boldsymbol{\mu}} \boldsymbol{k}_j\right) \tag{2.28}$$

$$c_\mu \leftarrow \lambda c_\mu + (1-\lambda)|\boldsymbol{\mu}| \tag{2.29}$$

$$\boldsymbol{\mu} \leftarrow \frac{\widetilde{\boldsymbol{\mu}}}{c_\mu} \tag{2.30}$$

其中 $|\boldsymbol{\mu}|$ 表示在簇 $\boldsymbol{\mu}$ 中向量的数量。

Reformer[56] 则采用局部敏感哈希（Local-Sensitive Hashing，LSH）的方法为每个 Query 选择 Key-Value 对。其主要思想是使用 LSH 函数对 Query 和 Key 进行哈希计算，将它们划分到多个桶内，以提升在同一个桶内的 Query 和 Key 参与交互的概率。假设 b 是桶的个数，给定一个大小为

$[D_k, b/2]$ 的随机矩阵 $\boldsymbol{R}$，LSH 函数的定义为

$$h(\boldsymbol{x}) = \arg\max([\boldsymbol{x}R; -\boldsymbol{x}R]) \tag{2.31}$$

当 $h\boldsymbol{q}_i = h\boldsymbol{k}_j$ 时，$\boldsymbol{q}_i$ 才可以与相应的 Key-Value 对进行交互。

2. FlashAttention

NVIDIA GPU 中的不同类型的内存（显存）有不同的速度、大小及访问限制。这主要取决于它们物理上是在 GPU 芯片内部还是在板卡 RAM 存储芯片上。GPU 显存分为全局内存（Global Memory）、本地内存（Local Memory）、共享存储（Shared Memory，SRAM）、寄存器（Register）、常量内存（Constant Memory）、纹理内存（Texture Memory）六大类。图 2.8 为 NVIDIA GPU 的整体内存结构示意图。全局内存、本地内存、共享存储和寄存器具有读写能力。全局内存和本地内存使用的高带宽显存（High Bandwidth Memory，HBM）位于板卡 RAM 存储芯片上，该部分内存容量很大。所有线程都可以访问全局内存，而本地内存只能由当前线程访问。NVIDIA H100 中全局内存有 80GB 空间，其访问速度虽然可以达到 3.35TB/s，但当全部线程同时访问全局内存时，其平均带宽仍然很低。共享存储和寄存器位于 GPU 芯片上，因此容量很小，并且只有在同一个 GPU 线程块（Thread Block）内的线程才可以并行访问共享存储，而寄存器仅限于同一个线程内部访问。虽然 NVIDIA H100 中每个 GPU 线程块在流式多处理器（Stream Multi-processor，SM）上可以使用的共享存储容量仅有 228KB，但是其速度比全局内存的访问速度快很多。

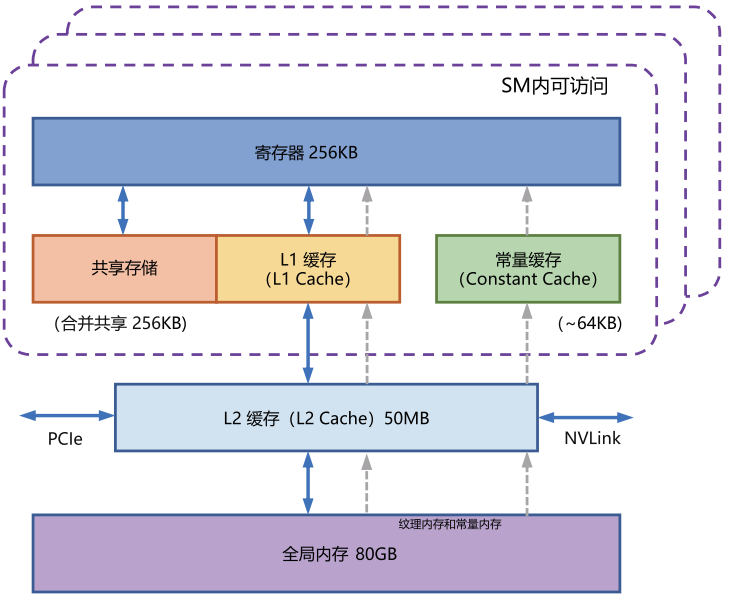

图 2.8 NVIDIA GPU 的整体内存结构示意图

前文介绍了自注意力机制的原理，在 GPU 中进行计算时，传统的方法还需要引入两个中间矩阵 $\boldsymbol{S}$ 和 $\boldsymbol{P}$ 并存储到全局内存中。具体计算过程如下：

$$\boldsymbol{S} = \boldsymbol{QK}, \quad \boldsymbol{P} = \text{Softmax}(\boldsymbol{S}), \quad \boldsymbol{O} = \boldsymbol{PV} \tag{2.32}$$

按照上述计算过程，需要先从全局内存中读取矩阵 $\boldsymbol{Q}$ 和 $\boldsymbol{K}$，并将计算好的矩阵 $\boldsymbol{S}$ 写入全局内存，然后从全局内存中获取矩阵 $\boldsymbol{S}$，计算 Softmax 得到矩阵 $\boldsymbol{P}$，再将其写入全局内存，最后读取矩阵 $\boldsymbol{P}$ 和矩阵 $\boldsymbol{V}$，计算得到矩阵 $\boldsymbol{O}$。这样的过程会极大地占用显存的带宽。在自注意力机制中，GPU 的计算速度比内存速度快得多，因此计算效率越来越受全局内存访问的制约。

FlashAttention[57] 利用 GPU 硬件中的特殊设计，针对全局内存和共享存储的 I/O 速度的不同，尽可能地避免从 HBM 中读取或写入注意力矩阵。FlashAttention 的目标是尽可能高效地使用 SRAM 来加快计算速度，避免从全局内存中读取和写入注意力矩阵。达成该目标需要做到在不访问整个输入的情况下计算 Softmax 函数，并且后向传播中不能存储中间注意力矩阵。在标准 Attention 算法中，Softmax 计算按行进行，即在与 $\boldsymbol{V}$ 做矩阵乘法之前，需要完成 $\boldsymbol{Q}$、$\boldsymbol{K}$ 每个分块中的一整行的计算。在得到 Softmax 的结果后，再与矩阵 $\boldsymbol{V}$ 分块做矩阵乘。而在 FlashAttention 中，将输入分割成块，并在输入块上进行多次传递，以增量的方式执行 Softmax 计算。

自注意力算法的标准实现将计算过程中的矩阵 $\boldsymbol{S}$、$\boldsymbol{P}$ 写入全局内存，而这些中间矩阵的大小与输入的序列长度有关且为二次型。因此，FlashAttention 就提出了不使用中间注意力矩阵，通过存储归一化因子来减少全局内存消耗的方法。FlashAttention 算法并没有将 $\boldsymbol{S}$、$\boldsymbol{P}$ 整体写入全局内存，而是通过分块写入，存储前向传播的 Softmax 归一化因子，在后向传播中快速重新计算片上注意力，这比从全局内存中读取中间注意力矩阵的标准方法更快。虽然大幅减少了全局内存的访问量，重新计算也导致 FLOPS 增加，但总体来看运行的速度更快且使用的显存更少。具体算法如代码2.1 所示，其中内层循环和外层循环所对应的计算可以参考图2.9。

代码 2.1: FlashAttention 算法

输入: $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{N \times d}$ 位于 HBM 中，GPU 芯片中的 SRAM 大小为 M

输出: $\boldsymbol{O}$

$B_{\mathrm{c}} = \lceil \frac{M}{4d} \rceil$，$B_{\mathrm{r}} = \min(\lceil \frac{M}{4d} \rceil, d)$ // 设置块大小（`block size`）

在 HBM 中初始化 $\boldsymbol{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\boldsymbol{l} = (0)_N \in \mathbb{R}^N$, $\boldsymbol{m} = (-\infty)_N \in \mathbb{R}^N$

将矩阵 $\boldsymbol{Q}$ 切分成 $T_{\mathrm{r}} = \lceil \frac{M}{B_{\mathrm{r}}} \rceil$ 块 $\boldsymbol{Q}_1, \boldsymbol{Q}_2, \cdots, \boldsymbol{Q}_{T_{\mathrm{r}}}, \boldsymbol{Q}_i \in \mathbb{R}^{B_{\mathrm{r}} \times d}$

将矩阵 $\boldsymbol{K}$ 切分成 $T_{\mathrm{c}} = \lceil \frac{M}{B_{\mathrm{c}}} \rceil$ 块 $\boldsymbol{K}_1, \boldsymbol{K}_2, \cdots, \boldsymbol{K}_{T_{\mathrm{c}}}, \boldsymbol{K}_i \in \mathbb{R}^{B_{\mathrm{c}} \times d}$

将矩阵 $\boldsymbol{V}$ 切分成 T_{c} 块 $\boldsymbol{V}_1, \boldsymbol{V}_2, \cdots, \boldsymbol{V}_{T_{\mathrm{c}}}, \boldsymbol{V}_i \in \mathbb{R}^{B_{\mathrm{c}} \times d}$

将矩阵 $\boldsymbol{O}$ 切分成 T_{r} 块 $\boldsymbol{O}_1, \boldsymbol{O}_2, \cdots, \boldsymbol{O}_{T_{\mathrm{r}}}, \boldsymbol{O}_i \in \mathbb{R}^{B_{\mathrm{r}} \times d}$

将 $\boldsymbol{l}$ 切分成 T_{r} 块 $\boldsymbol{l}_1, \boldsymbol{l}_2, \cdots, \boldsymbol{l}_{T_{\mathrm{r}}}, \boldsymbol{l}_i \in \mathbb{R}^{B_{\mathrm{r}}}$

将 $\boldsymbol{m}$ 切分成 T_{r} 块 $\boldsymbol{m}_1, \boldsymbol{m}_2, \cdots, \boldsymbol{m}_{T_{\mathrm{r}}}, \boldsymbol{m}_i \in \mathbb{R}^{B_{\mathrm{r}}}$

for $j = 1$ **to** T_{c} **do**

  将 $\boldsymbol{K}_j$ 和 $\boldsymbol{V}_j$ 从芯片外部的 HBM 中读入芯片内部存储 SRAM

  for $i = 1$ **to** T_{r} **do**

    计算 $\boldsymbol{S}_{ij} = \boldsymbol{Q}_i \boldsymbol{K}_j^T \in \mathbb{R}^{B_{\mathrm{r}} \times B_{\mathrm{c}}}$

    计算 $\tilde{\boldsymbol{m}}_{ij}$=rowmax($\boldsymbol{S}_{ij}$) $\in \mathbb{R}^{B_{\mathrm{r}}}$, $\tilde{\boldsymbol{P}}_{ij} = \exp(\boldsymbol{S}_{ij} - \tilde{\boldsymbol{m}}_{ij}) \in \mathbb{R}^{B_{\mathrm{r}} \times B_{\mathrm{c}}}$

    计算 $\tilde{\boldsymbol{l}}_{ij}$=rowsum($\tilde{\boldsymbol{P}}_{ij}$) $\in \mathbb{R}^{B_{\mathrm{r}}}$

    计算 $\boldsymbol{m}_i^{\mathrm{new}} = \max(\boldsymbol{m}_i, \tilde{\boldsymbol{m}}_{ij}) \in \mathbb{R}^{B_{\mathrm{r}}}$, $\boldsymbol{l}_i^{\mathrm{new}} = \mathrm{e}^{\boldsymbol{m}_i - \boldsymbol{m}_i^{\mathrm{new}}} \boldsymbol{l}_i + \mathrm{e}^{\tilde{\boldsymbol{m}}_{ij} - \boldsymbol{m}_i^{\mathrm{new}}} \tilde{\boldsymbol{l}}_{ij} \in \mathbb{R}^{B_{\mathrm{r}}}$

    将 $\boldsymbol{O} \leftarrow \mathrm{diag}(\boldsymbol{l}_i^{\mathrm{new}})^{-1}(\mathrm{diag}(\boldsymbol{l}_i)\mathrm{e}^{\boldsymbol{m}_i - \boldsymbol{m}_i^{\mathrm{new}}} \boldsymbol{O}_i + \mathrm{e}^{\tilde{\boldsymbol{m}}_{ij} - \boldsymbol{m}_i^{\mathrm{new}}} \tilde{\boldsymbol{P}}_{ij} \boldsymbol{V}_j)$ 写回 HBM 中

    将 $\boldsymbol{l}_i \leftarrow \boldsymbol{l}_i^{\mathrm{new}}$ 和 $\boldsymbol{m}_i \leftarrow \boldsymbol{m}_i^{\mathrm{new}}$ 写回 HBM 中

  end

end

return $\boldsymbol{O}$

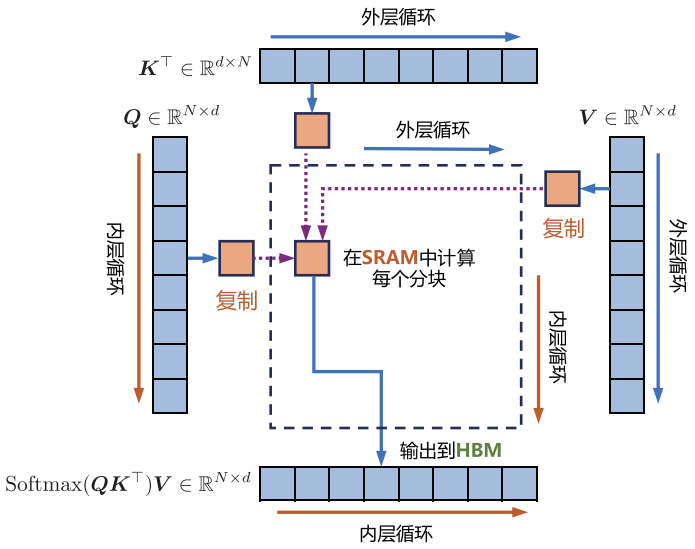

图 2.9 FlashAttention 计算流程图[57]

PyTorch 2.0 已经支持 FlashAttention，使用 torch.backends.cuda.enable_flash_sdp() 函数可以启用或者关闭 FlashAttention。

3. 多查询注意力

多查询注意力（Multi Query Attention）[58] 是多头注意力的一种变体。它的特点是，在多查询注意力中不同的注意力头共享一个键和值的集合，每个头只单独保留了一份查询参数，因此键和值的矩阵仅有一份，这大幅减少了显存占用，使其更高效。由于多查询注意力改变了注意力机制的结构，因此模型通常需要从训练开始就支持多查询注意力。文献 [59] 的研究结果表明，可以通过对已经训练好的模型进行微调来添加多查询注意力支持，仅需要约 5% 的原始训练数据量就可以达到不错的效果。包括 Falcon[60]、SantaCoder[61]、StarCoder[62] 在内的很多模型都采用了多查询注意力。

以 LLM Foundry 为例，多查询注意力的实现代码如下：

```python
class MultiQueryAttention(nn.Module):
    """
    多查询注意力
    使用torch或triton实现的注意力允许用户使用加性偏置
    """

    def __init__(
        self,
        d_model: int,
        n_heads: int,
        device: Optional[str] = None,
    ):
        super().__init__()

        self.d_model = d_model
        self.n_heads = n_heads
        self.head_dim = d_model // n_heads

        self.Wqkv = nn.Linear(                # 创建Multi Query Attention
            d_model,
            d_model + 2 * self.head_dim,      # 只创建查询的头向量，所以只有1个d_model
            device=device,                    # 键和值不再使用单独的头向量
        )

        self.attn_fn = scaled_multihead_dot_product_attention
        self.out_proj = nn.Linear(
            self.d_model,
            self.d_model,
            device=device
        )
        self.out_proj._is_residual = True

    def forward(
        self,
        x,
    ):
        qkv = self.Wqkv(x)                                          # (1, 512, 960)

        query, key, value = qkv.split(                              # query -> (1, 512, 768)
            [self.d_model, self.head_dim, self.head_dim],           # key   -> (1, 512, 96)
            dim=2                                                   # value -> (1, 512, 96)
        )

        context, attn_weights, past_key_value = self.attn_fn(
            query,
            key,
            value,
```

与 LLM Foundry 中实现的多头注意力代码相比，其区别仅在建立 Wqkv 层上：

```
# Multi Head Attention
self.Wqkv = nn.Linear(                          # Multi Head Attention的创建方法
    self.d_model,
    3 * self.d_model,                           # 查询、键和值3个矩阵，所以是3 * d_model
    device=device
)

query, key, value = qkv.chunk(                  # 每个tensor都是(1, 512, 768)
    3,
    dim=2
)


# Multi Query Attention
self.Wqkv = nn.Linear(                                  # Multi Query Attention的创建方法
    d_model,
    d_model + 2 * self.head_dim,                        # 只创建查询的头向量，所以是1* d_model
    device=device,                                      # 键和值不再使用单独的头向量
)

query, key, value = qkv.split(                          # query -> (1, 512, 768)
    [self.d_model, self.head_dim, self.head_dim],       # key   -> (1, 512, 96)
    dim=2                                               # value -> (1, 512, 96)
)
```

4. 多头潜在注意力

多头潜在注意力（Multi-Head Latent Attention，MLA）[63] 是在 DeepSeek-V2 中引入的注意力优化模型。多头潜在注意力通过在键值层利用低秩矩阵，实现对压缩潜在键值状态的缓存（更详细的 KV 缓存可以参考本书第 10 章内容），从而大幅减少了 KV 缓存大小，有效缓解了通信瓶颈。

具体来说，MLA 方法的核心是是将传统多头注意力中的键（Key）和值（Vale）进行低秩联合压缩，得到一个低秩表示形式，以减少键值（KV）缓存。设 d 为嵌入维度，n_h 为注意力头的数量，d_h 为每个头的维度，$\boldsymbol{h}_t \in \mathbb{R}^d$ 是注意力层中第 t 个词元的输入。标准的多头注意力机制（MHA）首先通过三个矩阵 $\boldsymbol{W}^Q$、$\boldsymbol{W}^K$、$\boldsymbol{W}^V \in \mathbb{R}^{d_h n_h \times d}$ 生成 $\boldsymbol{q}_t$、$\boldsymbol{q}_t$、$\boldsymbol{q}_t \in \mathbb{R}^{d_h n_h}$。MLA 方法则通过如下公式对 KV 缓存进行压缩：

$$\boldsymbol{c}_t^{KV} = \boldsymbol{W}^{DKV}\boldsymbol{h}^t \tag{2.33}$$

$$\boldsymbol{k}_t^C = \boldsymbol{W}^{UK}\boldsymbol{c}_t^{KV} \tag{2.34}$$

$$\boldsymbol{v}_t^C = \boldsymbol{W}^{UV}\boldsymbol{c}_t^{KV} \tag{2.35}$$

其中，$\boldsymbol{c}_t^{KV} \in \mathbb{R}^{d_c}$ 是键和值的压缩潜在向量（Comressed Latent Vector）；$d_c(\ll d_h n_h)$ 表示键值压缩维度；$\boldsymbol{W}^{DKV} \in \mathbb{R}^{d_c \times d}$ 是下投影矩阵；而 $\boldsymbol{W}^{UK}, \boldsymbol{W}^{UV} \in \mathbb{R}^{d_h n_h \times d_c}$ 分别是键和值的上投影矩阵。在推理过程中，MLA 方法只需要缓存 $\boldsymbol{c}_t^{KV}$，因此其键值缓存仅有 $d_c l$ 个元素，其中 l 表示层数。

此外，在推理过程中，由于 $\boldsymbol{W}^{UK}$ 可以合并到 $\boldsymbol{W}^{Q}$ 中，$\boldsymbol{W}^{UV}$ 可以合并到 $\boldsymbol{W}^{O}$ 中，甚至无需在注意力计算中真正获得键和值。为了在训练过程中减少激活内存，还可以进一步对查询（Query）进行低秩压缩：

$$\boldsymbol{c}_t^{Q} = \boldsymbol{W}_{DQ}\boldsymbol{h}_t \tag{2.36}$$

$$\boldsymbol{q}_t^{C} = \boldsymbol{W}^{UQ}\boldsymbol{c}_t^{Q} \tag{2.37}$$

其中，$\boldsymbol{c}_t^{Q} \in \mathbb{R}^{d_c'}$ 是查询的压缩潜在向量；$d_c'(\ll d_h n_h)$ 表示查询压缩维度，$\boldsymbol{W}^{DQ} \in \mathbb{R}^{d_c' \times d}$ 和 $\boldsymbol{W}^{UQ} \in \mathbb{R}^{d_h n_h \times d_c'}$ 分别是查询的下投影矩阵和上投影矩阵。

文献 [64] 还进一步在理论上证明了 MLA 方法在表现力上优于组查询注意力（Group Query Attention，GQA）。当 MLA 和 GQA 使用相同大小的 KV 缓存时，MLA 表现出更强的能力。这是因为在某些情况下，MLA 能够在通道输出上展现更大的多样性，而 GQA 由于组内头部是复制的，导致组内所有头部的输出相同，无法捕捉到 MLA 所能处理的某些情况。文献 [64] 还提出了 TransMLA 后训练方法，该方法能够将广泛使用的基于 GQA 的预训练模型（例如 LLaMA、Qwen、Mixtral）转换为基于 MLA 的模型。转换后，通过进一步训练，在不增加 KV 缓存大小的前提下有效提升模型的表现力。

2.4 混合专家模型

随着 GPT-4[65]、Mixtral-8x7B[66]、DeepSeek-V3[40] 等模型的相继推出，**混合专家模型** (Mixed Expert Models，MoEs) 日益受到关注。依据大模型缩放法则，模型规模是提升性能的关键，然而规模扩大必然使计算资源大幅增加。因此，在有限计算资源预算下，如何用更少训练步数训练更大模型成为关键问题。为解决该问题，混合专家模型基于一个简洁的思想：模型不同部分（即“专家”）专注不同任务或数据层面。混合专家架构的引入使得训练具有数千亿甚至万亿参数的模型成为可能，如开源的 1.6 万亿参数的 Switch Transformers[67] 等。

在采用混合专家架构的大语言模型中，MoE 层通常由门控网络（Gating Network）$\mathcal{G}$ 和 N 个专家网络（Experts Network）$\{f_1, f_2, ..., f_N\}$ 组成。门控网络充当着选择器的角色，也称为路由，它负责决定将哪些输入数据发送给哪些专家。专家网络则分别处理特定的不同子任务。在这一过程中，并非所有专家都同时运作，而是由门控网络依据数据特性，精准地将数据路由到与之最为相关的专家那里，最终再根据一个或者多个专家输出的结果综合得到整体的预测结果。在模型架构的设计中，MoE 层通常安置于每个 Transformer 模块中前馈层（FFN）。当模型不断扩大时，FFN 层在计算方面的需求也越来越高。例如，在参数数量达 5400 亿的 PaLM[14] 模型中，90% 的参数

都位于前馈网络层内。

混合专家架构中，每个专家网络 f_i 通常由一个前馈层组成，其参数使用 $\mathbf{W}_i$ 表示。对于给入的输入 X，其输出使用 $f_i(X;\mathbf{W}_i)$ 表示。门控网络 $\mathcal{G}$ 通常使用线性 Softmax（Linear-Softmax）网络构成，使用 $\mathbf{\Theta}$ 表示其参数，其输出使用 $\mathcal{G}_i(\mathbf{x};\mathbf{\Theta})$ 表示。混合专家模型按照门控网络（Gate）类型，可以从广义上讲可以分为三个大类：**稀疏混合专家模型**（Sparse MoE）、**稠密混合专家模型**（Dense MoE）、**软混合专家模型**（Soft MoE），如图2.10所示。

本节将按照门控网络类型类型的分类，分别介绍稀疏混合专家模型、稠密混合专家模型和软混合专家模型的定义、特点和代表性工作。

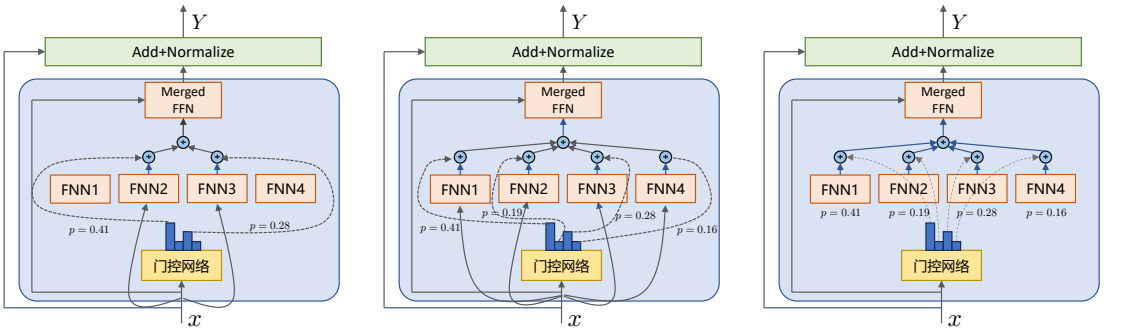

图 2.10 混合专家模型三种主要类型[68]

2.4.1 稀疏混合专家模型

稀疏混合专家模型，如图2.10(a) 所示，对于每个输入词元，在前向计算中仅激活专家集合中的一个子集。门控网络对专家子集进行选择，通过计算排名前 K 位专家的输出加权和来实现稀疏性。这个过程可以形式化的表示为：

$$\mathcal{F}^{MoE}_{Sparse}(\mathbf{x};\mathbf{\Theta};\{\mathbf{W}_i\}_{i=1}^N)=\sum_{i=1}^{N}\mathcal{G}(\mathbf{x};\mathbf{\Theta})_i f_i(\mathbf{x};\mathbf{W}_i) \tag{2.38}$$

$$\mathcal{G}(\mathbf{x};\mathbf{\Theta})_i=\text{softmax}(\text{TopK}(g(\mathbf{x};\mathbf{\Theta})+\mathcal{R}_{noise},K))_i \tag{2.39}$$

$$\text{Top-K}(g(\mathbf{x};\mathbf{\Theta}),K)_i=\begin{cases} g(\mathbf{x};\mathbf{\Theta})_i, g(\mathbf{x};\mathbf{\Theta})_i\text{的值属于前 K 项} \\ -\infty, \text{其他} \end{cases} \tag{2.40}$$

其中，$g(\mathbf{x};\mathbf{\Theta})$ 表示在进行 softmax 操作之前的门控值，$\mathcal{G}(\mathbf{x};\mathbf{\Theta})_i$ 表示门控网络针对第 i 个专家的输出，$\text{TopK}(\cdot,K)$ 函数的目标是保持向量的前 K 项不变，其它维度设置为 $-\infty$。鉴于 softmax 函数自身所具有的独特性质，当把其中某些项设置为 $-\infty$ 时，这些项所对应的值会近似等同于 0。超参数 K 是根据具体应用来选取的，常见的取值选择为 $K=1$[67, 69] 或者 $K=2$[66, 70–72]。添加噪

声项 $\mathcal{R}_{noise}$ 是训练稀疏混合专家层的一种常用策略，一方面，它能够为模型创造更多的探索空间，促使不同专家模块之间展开多样化的尝试与协作，挖掘出潜在的优化路径；另一方面，通过打破可能出现的局部最优情况，提高了整个混合专家训练过程的稳定性[67]。

由 Mixtral AI 公司推出的 Mixtral-8x7B 模型[66] 就采用了稀疏混合专家方式，与早期的 Mistral 7B 模型[73] 共享基础架构。但是，Mixtral-8x7B 模型使用了稀疏混合专家层代替每个 Transformer 块中的前馈层，每个稀疏混合专家层包含 8 个专家网络，门控网络每次激活 2 个专家。但是在 Mixtral-8x7B 模型中没有引入噪声项 $\mathcal{R}_{noise}$，每个专家网络则使用了 SwiGLU 结构[46]。由于采用了稀疏混合专家方式，虽然 Mixtral-8x7B 模型的总参数量大约 560 亿，但是每次仅使用 130 亿个活跃参数。并且，Mixtral-8x7B 模型在很多基准测试中，展现出了优于或等同于包含了 700 亿参数的 Llama-2-70B[37] 的性能。此外，众多大语言模型也都采用了稀疏混合专家架构，包括 Switch Transformer[67]、DeepSeekMoE[74]、AdaMoE[75]、Yuan 2.0-M32[76]、OpenMoE[77]、Qwen1.5-MoE-A2.7B[78] 等。更多相关模型可以参考文献 [68]。

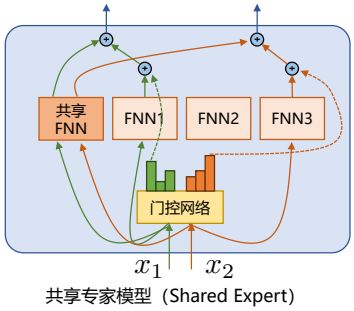

图 2.11　共享专家模型[68]

稀疏混合专家模型中采用常规的门控策略时，分配给不同专家的词元可能需要一些共有知识或信息才能处理。因此，多个专家可能会在各自的参数中获取同样的知识，进而导致专家参数出现冗余。如果构建专门用于捕捉并整合不同情境下共有知识的共享专家，那么其他专家之间的参数冗余情况将可能得到缓解。这种冗余情况的缓解，有助于构建一个参数利用更高效且专家专业性更强的模型。因此，DeepSeekMoE[74] 提出了分离 K_s 个专家作为共享专家的思路。无论门控网络所给出的结果如何，每个词元都将被确定性地分配给这些共享专家，如图 2.11 所示，深色块 Shared FFN 为共享专家，所有输入都会分配给共享专家。为保持计算成本恒定，其他经门控网络分配的专家中被激活专家的数量将减少 K_s 个。

稀疏混合专家模型中的 MoE 层对于并行计算也十分友好，能更便捷地在单个 GPU 上实现高效计算。常规稠密模型中，全部参数都会参与对所有输入数据的处理流程。与之不同，稀疏混合专家模型具备的稀疏特性，使得计算仅在系统的特定局部展开。也就是说，并非所有参数在处理各个输入时都会被触发或启用，而是依据输入的具体特性与需求，仅有特定的部分参数集被唤起

并运行。因此，在并行计算中可以有效利用上述特性。例如，Megablocks[79] 将 MoE 层的前馈网络运算转换为大型稀疏矩阵乘法，极大地提高了执行速度，并且能够很好地处理不同专家分配到的数量不等的词元情况。此外，MoE 层可以通过标准的模型并行技术分布到多个 GPU 上，还可以借助专家并行（Expert Parallelism，EP）[80] 实现特殊的分区策略。

2.4.2 稠密混合专家模型

稠密混合专家模型，如图2.10(b) 所示，对于每个输入词元，在前向计算中激活所有专家网络 $\{f_1, ..., f_N\}$。门控网络根据输入赋予专家不同的权重。这个过程可以形式化的表示为：

$$\mathcal{F}_{Dense}^{MoE}(\mathbf{x}; \mathbf{\Theta}; \{\mathbf{W}_i\}_{i=1}^N) = \sum_{i=1}^{N} \mathcal{G}(\mathbf{x}; \mathbf{\Theta})_i f_i(\mathbf{x}; \mathbf{W}_i) \tag{2.41}$$

$$\mathcal{G}(\mathbf{x}; \mathbf{\Theta})_i = \text{softmax}(g(\mathbf{x}; \mathbf{\Theta}))_i = \frac{\exp(g(\mathbf{x}; \mathbf{\Theta})_i)}{\sum_j^N \exp(g(\mathbf{x}; \mathbf{\Theta})_j)} \tag{2.42}$$

由于稠密混合专家模型在前向计算过程中会激活所有参数，不能降低模型计算量。因此，大语言模型采用稠密混合专家结构的并不多，主要包括 EvoMoE[81]、MoLE[82]、LoRAMoE[83] 以及 DS-MoE[84] 等。

虽然稠密混合专家模型需要使用全部参数进行计算，并不能减少模型计算时间，但是研究人员却发现，如果能够将 LoRA 方法和 MoE 相结合，可以在占用很少 GPU 显存的同时，减少微调数据的大规模扩增与模型世界知识维持之间存在的冲突。有监督微调是大语言模型应用的一个关键步骤，当模型需要与更广泛的下游任务保持一致，或者希望显著提高在特定任务上的表现时，大规模增加微调数据通常成为解决方案。然而当指令数据的大规模扩增可能会破坏大语言模型中之前储存的世界知识，即世界知识遗忘。LoRAMoE[83] 采用融合混合专家和 LoRA 插件的思想，插件形式确保了在训练阶段冻结主模型，保证了主模型世界知识的完整性。

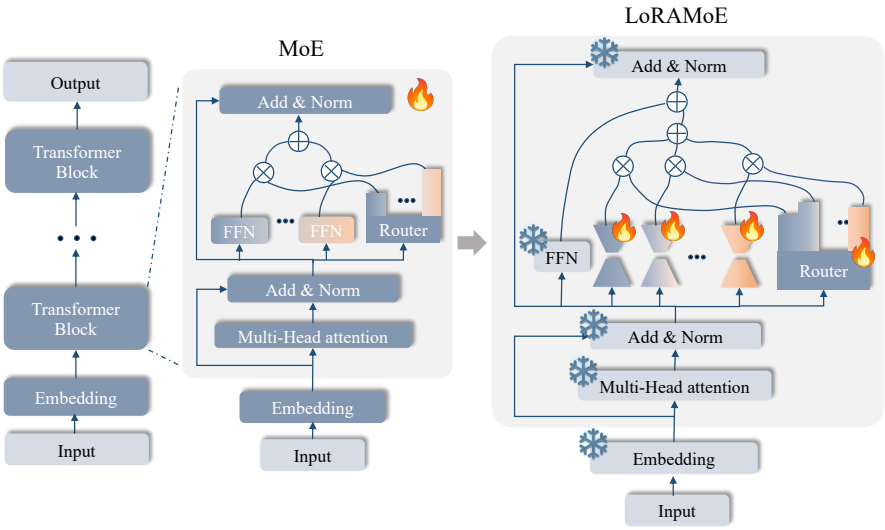

图 2.12　LoRAMoE 模型架构图[83]

LoRAMoE 模型架构如图2.12所示。基于插件的微调能够将参数的改动集中在额外引入的插件中，从而保证了模型知识的完整性，有机会引入其他插件来通过与主模型的交互来缓解知识遗忘。LoRAMoE 引入了多个与前反馈神经网络并列的专家，并通过路由相连，如图2.12中标注了“火焰”符号的部分，这些部分也是需要在后续学习中进行参数学习的结构。LoRAMoE 在训练阶段使用局部平衡约束损失（Localized Balancing Constraint），这种约束能够让专家自动划分为两个组：使一部分专家在专注于做下游任务的同时，另一部分专家专注于将指令与主模型的世界知识对齐，以缓解世界知识遗忘。同时局部平衡约束还能防止单个专家组内的专家退化现象，使路由平衡地关注于单个专家组的所有专家，防止个别专家长期占据优势，而其他专家未被充分训练或使用。这有助于专家之间相互配合以提高下游任务能力。微调后的 LoRAMoE 中的路由能够根据数据类型灵活地关注相应的专家，并使专家们相互配合，在保证下游任务表现的同时，也几乎不丧失世界知识。

2.4.3 软混合专家模型

软混合专家模型,如图2.10(c)所示,门控网络依然根据输入为各个专家分配不同的权重,但与稠密混合专家模型在前向计算中激活所有专家网络不同,软混合专家模型引入了融合前馈层(Merged FFN)。该方法通过门控网络分配的权重对不同专家的参数进行融合，仅对融合后的前馈层参数进行计算。这种设计既能在几乎不增加计算成本的情况下完成计算，又保留了稠密混合专家模型中

可使用基于梯度的训练方法的优势。这个过程可以形式化的表示为：

$$\mathcal{F}_{Soft}^{MoE}(\mathbf{x};\mathbf{\Theta};\{\mathbf{W}_i\}_{i=1}^{N}) = f_{merged}(\mathbf{x};\sum_{i=1}^{N}\mathcal{G}(\mathbf{x};\mathbf{\Theta})_i\mathbf{W}_i) \tag{2.43}$$

$$\mathcal{G}(\mathbf{x};\mathbf{\Theta})_i = \text{softmax}(g(\mathbf{x};\mathbf{\Theta}))_i = \frac{\exp(g(\mathbf{x};\mathbf{\Theta})_i)}{\sum_j^N \exp(g(\mathbf{x};\mathbf{\Theta})_j)} \tag{2.44}$$

其中，f_{merged} 表示融合前向层，其结构与其余专家网络 f_i 的结构相同。SMEAR 算法[85] 就采用了这种软混合专家结构。

软混合专家模型始终只计算单个专家的输出，其计算成本可能与单专家稀疏混合模型相当，明显低于稠密混合专家模型。但是，软混合专家模型的平均操作仍然会产生不可忽视的计算成本。为了量化这一成本，文献 [85] 分析了 SMEAR 算法的计算复杂度。假设专家网络架构是一个从 d 维激活值投射到 m 维向量的稠密计算，随后经过非线性变换，再附加一个从 m 维投射回 d 维的稠密计算。为简便起见，这里忽略成本相对较小的非线性变换成本。假定输入是一个长度为 L 的激活值序列，其大小为 $L \times d$。在这种情况下，计算合并专家的输出会产生大约 $L \times 4 \times d \times m$ 次浮点运算（FLOPs）的计算成本，而采用 N 个专家的稠密混合专家模型则需要 $N \times L \times 4 \times d \times m$ 次浮点运算。此外，软混合专家模型还必须对 N 个专家的参数进行平均，这又会额外产生 $N \times 2 \times d \times m$ 次浮点运算的成本。整体上 SMEAR 算法的计算复杂度是 $(L \times 4 + N \times 2) \times d \times m$。综合整体计算成本，软混合专家模型计算复杂度仍然远低于稠密混合专家模型。

3. 大语言模型预训练数据

在预训练阶段，大语言模型从海量“高质量”文本数据中学习广泛的知识，随后这些知识存储在其模型参数当中。通过预训练使得大语言模型具备了一定程度的语言理解和生成能力。因此，如何构造海量“高质量”数据对于大语言模型预训练具有至关重要的作用。研究表明，预训练数据需要涵盖各种类型的文本，也需要覆盖尽可能多的领域、语言、文化和视角，从而提高大语言模型的泛化能力和适应性。当前大模型预训练使用的语料库涵盖网页内容、学术资料、百科、社交媒体和书籍等文本内容，同时也包含来自不同领域的文本内容，比如法律文件、年度财务报告、医学教科书等其他特定领域的数据。

本章将介绍常见的大语言模型预训练数据的来源、处理方法、预训练数据对大语言模型影响的分析及开源数据集等。

3.1 数据来源

文献 [13] 介绍了 OpenAI 训练 GPT-3 使用的主要数据来源，包含经过过滤的 CommonCrawl 数据集[19]、WebText 2、Books 1、Books 2 及英文 Wikipedia 等数据集。其中 CommonCrawl 的原始数据有 45TB，过滤后仅保留了 570GB 的数据。通过词元方式对上述数据进行切分，大约包含 5000 亿个词元。为了保证模型使用更多高质量数据进行训练，在 GPT-3 训练时，根据数据来源的不同，设置不同的采样权重。在完成 3000 亿个词元的训练时，英文 Wikipedia 的数据平均训练轮数为 3.4 次，而 CommonCrawl 和 Books 2 仅有 0.44 次和 0.43 次。由于 CommonCrawl 数据集的过滤过程烦琐复杂，Meta 公司的研究人员在训练 OPT[29] 模型时采用了混合 RoBERTa[86]、Pile[87] 和 PushShift.io Reddit[88] 数据的方法。由于这些数据集中包含的绝大部分数据都是英文数据，因此 OPT 也从 CommonCrawl 数据集中抽取了部分非英文数据加入训练数据。

大语言模型预训练所需的数据来源大体上分为通用数据和专业数据两大类。**通用数据**（General Data）包括网页、图书、新闻、对话文本等[14, 29, 39]。通用数据具有规模大、多样性和易获取等特点，因此支持大语言模型的语言建模和泛化能力。**专业数据**（Specialized Data）包括多语言数据、科学文本数据、代码及领域特有资料等。通过在预训练阶段引入领域数据可以有效提升大语言模

型的任务解决能力。图3.1 给出了一些典型的大语言模型所使用数据类型的分布情况。可以看到，不同的大语言模型在训练数据类型分布上的差距很大，截至 2025 年 2 月，业界关于预训练数据的配比还没达成广泛的共识。

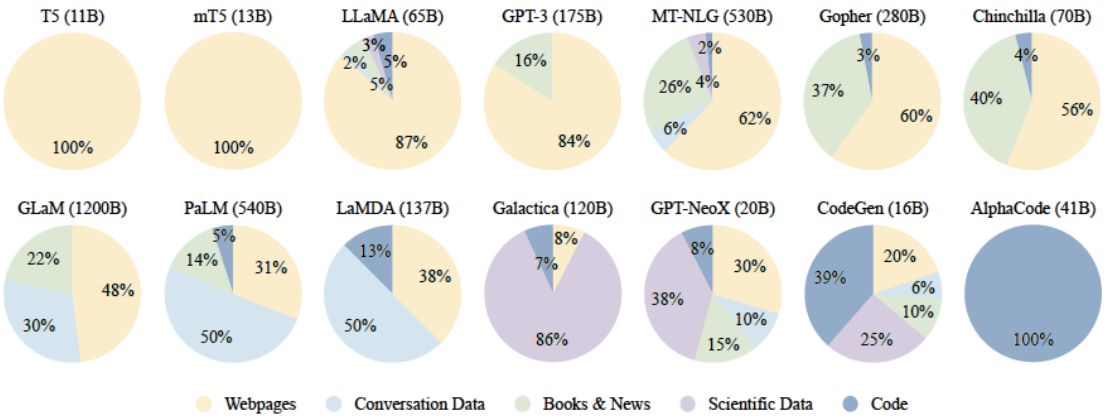

图 3.1　典型的大语言模型所使用数据类型的分布情况[18]

3.1.1 通用数据

通用数据在大语言模型训练数据中占比非常高，主要包括网页、对话文本、书籍、代码、百科等不同类型的数据，为大语言模型提供了大规模且多样的训练数据。

网页（Webpage）是通用数据中数量最多的一类。随着互联网的大规模普及，人们通过网站、论坛、博客、App 创造了海量的数据。根据 2016 年 Google 公开的数据，其搜索引擎索引处理了超过 130 万亿个网页数据。网页数据所包含的海量内容，使语言模型能够获得多样化的语言知识并增强其泛化能力[11, 19]。爬取和处理海量网页内容并不是一件容易的事情，因此一些研究人员构建了 ClueWeb09[89]、ClueWeb12[90]、SogouT-16[91]、CommonCrawl[19] 等开源网页数据集。虽然这些爬取的网络数据包含大量高质量的文本，但也包含非常多低质量的文本（如垃圾邮件等）。因此，过滤并处理网页数据以提高数据质量对大语言模型训练非常重要。

对话文本（Conversation Text）是指有两个或更多参与者交流的文本内容。对话文本包含书面形式的对话、聊天记录、论坛帖子、社交媒体评论等。当前的一些研究表明，对话文本可以有效增强大语言模型的对话能力[29]，并潜在地提高大语言模型在多种问答任务上的表现[14]。对话文本可以通过收集、清洗、归并等过程从社会媒体、论坛、邮件组等处构建。相较于网页数据，对话文本数据的收集和处理更加困难，数据量也少很多。常见的对话文本数据集包括 PushShift.io Reddit[88, 92]、Ubuntu Dialogue Corpus[93]、Douban Conversation Corpus、Chromium Conversations Corpus 等。此外，文献 [94] 也提出了使用大语言模型自动生成对话文本数据的 UltraChat 方法。

书籍（Book）是人类知识的主要积累方式之一，从古代经典著作到现代学术著述，承载了丰富多样的人类思想。书籍通常包含广泛的词汇，包括专业术语、文学表达及各种主题词汇。利用书籍数据进行训练，大语言模型可以接触多样化的词汇，从而提高其对不同领域和主题的理解能力。相较于其他数据库，书籍也是最重要的，甚至是唯一的长文本书面语的数据来源。书籍提供了完整的句子和段落，使大语言模型可以学习到上下文之间的联系。这对于模型理解句子中的复杂结构、逻辑关系和语义连贯性非常重要。书籍涵盖了各种文体和风格，包括小说、科学著作、历史记录，等等。用书籍数据训练大语言模型，可以使模型学习到不同的写作风格和表达方式，提高大语言模型在各种文本类型上的能力。受限于版权因素，开源书籍数据集很少，现有的开源大语言模型研究通常采用 Pile 数据集[87] 中提供的 Books 3 和 BookCorpus 2 数据集。

多语言数据（Multilingual Text）对于增强大语言模型的多语言理解和生成多语言能力具有至关重要的作用。当前的大语言模型训练除了需要目标语言中的文本，通常还要整合多语言数据库。例如，BLOOM[31] 的预训练数据中包含 46 种语言的数据，PaLM[14] 的预训练数据中甚至包含高达 122 种语言的数据。此前的研究发现，通过多语言数据混合训练，预训练模型可以在一定程度上自动构建多语言之间的语义关联[95]。因此，多语言数据混合训练，可以有效提升翻译、多语言摘要和多语言问答等任务能力。此外，由于不同语言中不同类型的知识获取难度不同，多语言数据还可以有效地增加数据的多样性和知识的丰富性。

科学文本（Scientific Text）数据包括教材、论文、百科及其他相关资源。这些数据对于提升大语言模型在理解科学知识方面的能力具有重要作用[96]。科学文本数据的来源主要包括 arXiv 论文[97]、PubMed 论文[98]、教材、课件和教学网页等。由于科学领域涉及众多专业领域且数据形式复杂，通常还需要对公式、化学式、蛋白质序列等采用特定的符号标记并进行预处理。例如，公式可以用 LaTeX 语法表示，化学结构可以用 SMILES（Simplified Molecular Input Line Entry System）表示，蛋白质序列可以用单字母代码或三字母代码表示。这样可以将不同格式的数据转换为统一的形式，使大语言模型更好地处理和分析科学文本数据。

百科（Encyclopedia）数据包含百科全书、在线百科网站及其他知识数据库，这些数据中蕴含着极为丰富的知识。百科知识内容通常是经由专家严谨编撰、志愿者无私奉献以及社区贡献者协同努力，得以创作与完善，具备一定的权威性与可靠性。由于此类知识资源易于获取，在大语言模型的预训练语料构建进程中发挥着至关重要的作用。最常见的百科语料库是维基百科（Wikipedia）。它具有免费、开源、多语言以及文本价值高的特点。几乎所有的大语言模型预训练都会将维基百科作为其预训练语料库的一部分。就中文百科语料库而言，除了中文版维基百科外，还有百度百科、搜狗百科等来源。它们几乎涵盖了所有知识领域，TigerBot-wiki[99] 就是从百度百科的数据中筛选出来的。

代码（Code）是进行程序生成任务所必需的训练数据。近期的研究和 ChatGPT 的结果表明，通过在大量代码上进行预训练，大语言模型可以有效提升代码生成的效果[100, 101]。代码不仅包含程序代码本身，还包含大量的注释信息。与自然语言文本相比，代码具有显著的不同。代码是一种格

式化语言，它对应着长程依赖和准确的执行逻辑[102]。代码的语法结构、关键字和特定的编程范式都对其含义和功能起着重要的作用。代码的主要来源是编程问答社区（如 Stack Exchange[103, 104]）和公共软件仓库（如 GitHub[27, 100, 105]）。编程问答社区中的数据包含了开发者提出的问题、其他开发者的回答及相关代码示例。这些数据提供了丰富的语境和真实世界中的代码使用场景。公共软件仓库中的数据包含了大量的开源代码，涵盖多种编程语言和不同领域。这些代码库中的很多代码经过了严格的代码评审和实际的使用测试，因此具有一定的可靠性。

3.1.2 领域数据

特定领域预训练语料库是为特定领域或主题量身定制的。这类语料库通常用于大语言模型的增量预训练阶段。在用通用预训练语料库训练出一个基础模型之后，如果该模型需要应用于某一特定领域的下游任务，就可以进一步利用特定领域预训练语料库对模型进行增量预训练。这一过程在基于初始通用预训练所获得的通用能力基础上，增强了模型在特定领域的能力。虽然领域数据相比通用数据所占比例通常较低，但是其对改进大语言模型在特定领域任务上的能力有着非常重要的作用。专业数据有非常多的种类，文献 [106] 总结了当前开源或部分开源领域数据情况。

金融领域的预训练语料库有助于大言模型学习金融市场、经济学、投资及金融相关主题知识。金融领域文本数据通常来源于金融新闻、财务报表、公司年报、金融研究报告、金融文献、市场数据等。BBT-FinCorpus[107] 是一个大规模的中文金融领域语料库，由公司公告、研究报告、金融新闻和社交媒体这四个部分组成。该语料集用于 BBT-FinT5 基础模型的预训练[107]。FinCorpus[108] 是一个中文金融领域语料库，包含公司公告、金融信息与新闻、金融考试题目等。FinGLM 则致力于构建一个开放的、公益的、持久的金融大模型项目，数据涵盖 10000 份 2019 年至 2021 年上市公司的年报。FinGPT[109] 收集了金融新闻、社交媒体、金融监管机构文件、金融趋势分析文章以及金融学术数据集等数据。为了充分利用这些不同来源的丰富信息，FinGPT 还构建了能够抓取结构化和非结构化数据的数据采集工具。TigerBot-research[99] 和 TigerBot-earning[99] 则分别侧重于研究报告和财务报告。

医疗领域的预训练语料库通常包含大量的医学文本语料库（包括结构化和非结构化文本），包括电子健康记录、临床记录以及医学文献等。PubMed[98] 是一个由美国国家医学图书馆（NLM）维护的在线数据库，用于检索医学和生物医学领域的文献，包括期刊文章、会议论文、技术报告、书籍、政府出版物和学位论文等大量资源。PubMed Central（PMC）则是免费全文数据库。MIMIC-III[110] 是一个大型的、可免费获取的用于医疗研究的数据库，收集了从 2001 年到 2012 年期间在 Beth Israel Deaconess Medical Center 的重症监护病房（ICU）中的患者数据，包含了患者的生命体征、实验室测试结果、药物使用、诊断和治疗过程等详细的临床信息。Medical-GPT[111] 以及 Baichuan-M1 都使用了可开放获取的医学百科全书和医学教科书数据。Huatuo-26M[112] 是目前规模最大的中文医疗问答数据集之一，该数据集包含逾 2600 万条高质量的医疗问答对，涵盖疾病、症状、治疗方法以及药物信息等诸多方面。MedDialog[113] 是一个多语言的医疗对话数据集，包含中文和英文的医

疗对话数据。中文数据集包含 340 万条医生-患者对话，覆盖 172 个疾病领域，而英文数据集包含 26 万条对话，覆盖 96 个疾病领域。

法律领域也包含许多可用于模型训练的数据资源，主要包括法律法规、裁判文书等法律数据。这些数据通常可以从相关官方网站下载获得，且数据规模较大，能够为大模型提供大量的法律专业知识。此外，还还通过收集司法考试题目、法律咨询、法律问答等相关数据，这类数据涉及了真实用户的法律需求与基于法律专业知识的解答。CUAD[114] 是一个包含 510 个商业法律合同、超过 1.3 万个标注的合同审查数据集，由数十名法律专业人士和机器学习研究人员共同创建，通过法律专业人士对这些合同数据进行扩充和详细标注。TigerBot-law[99] 则汇集了 11 类中国法律法规，以及一些多类别语料库，还纳入了从法律相关网站抓取的数据。

3.2 数据处理

大语言模型的相关研究表明，数据质量对于模型的影响非常大。因此，在收集了各种类型的数据之后，需要对数据进行处理，去除低质量数据、重复数据、有害信息、个人隐私等内容[14, 115]。典型的数据处理流程如图3.2 所示，主要包括质量过滤、冗余去除、隐私消除、词元切分这几个步骤。本节将依次介绍上述内容。

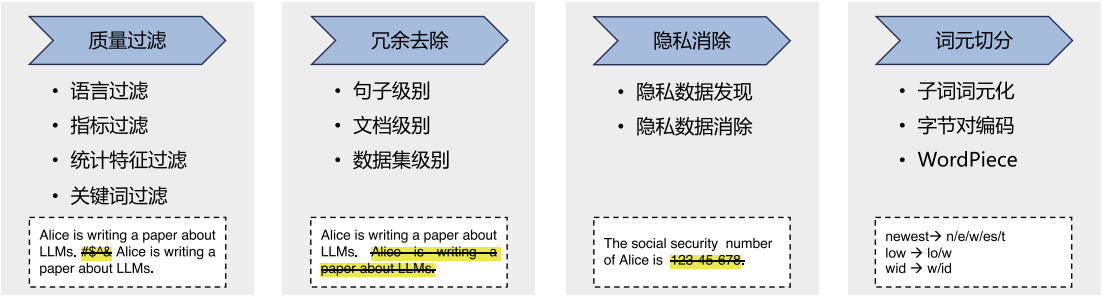

图 3.2 典型的数据处理流程图[18]

3.2.1 质量过滤

互联网上的数据质量参差不齐，无论是 OpenAI 联合创始人 Andrej Karpathy 在微软 Build 2023 的报告，还是当前的一些研究都表明，训练数据的质量对于大语言模型效果具有重大影响。因此，从收集到的数据中删除低质量数据成为大语言模型训练中的重要步骤。大语言模型训练中所使用的低质量数据过滤方法可以大致分为两类：**基于分类器的方法**和**基于启发式的方法**。

基于分类器的方法的目标是训练文本质量判断模型，利用该模型识别并过滤低质量数据。GPT-3[39]、PaLM[14] 和 GLaM[116] 模型在训练数据构造时都使用了基于分类器的方法。文献 [116] 采用了基于特征哈希的线性分类器（Feature Hash Based Linear Classifier），可以非常高效地完成文

本质量判断。该分类器使用一组精选文本（维基百科、书籍和一些选定的网站）进行训练，目标是给于训练数据类似的网页较高分数。利用这个分类器可以评估网页的内容质量。在实际应用中，还可以通过使用 Pareto 分布对网页进行采样，根据其得分选择合适的阈值，从而选定合适的数据集。然而，一些研究发现，基于分类器的方法可能会删除包含方言或者口语的高质量文本，从而损失一定的多样性[115, 116]。

基于启发式的方法则通过一组精心设计的规则来消除低质量文本，BLOOM[31] 和 Gopher[115] 采用了基于启发式的方法。一些启发式规则如下。

- 语言过滤：如果一个大语言模型仅关注一种或者几种语言，则可以大幅过滤数据中其他语言的文本。
- 指标过滤：利用评测指标也可以过滤低质量文本。例如，可以使用语言模型对给定文本的困惑度进行计算，利用该值可以过滤非自然的句子。
- 统计特征过滤：针对文本内容可以计算包括标点符号分布、符号字比（Symbol-to-Word Ratio）、句子长度在内的统计特征，利用这些特征过滤低质量数据。
- 关键词过滤：根据特定的关键词集，可以识别并删除文本中的噪声或无用元素。例如，HTML 标签、超链接及冒犯性词语等。

在大语言模型出现之前，在自然语言处理领域已经开展了很多**文章质量判断**（Text Quality Evaluation）相关的研究，主要应用于搜索引擎、社会媒体、推荐系统、广告排序及作文评分等任务中。在搜索和推荐系统中，结果的内容质量是影响用户体验的重要因素之一，因此，此前很多工作都是针对用户生成内容（User-Generated Content，UGC）的质量进行判断的。自动作文评分也是文章质量判断领域的一个重要子任务，自 1998 年文献 [117] 提出使用贝叶斯分类器进行作文评分预测以来，基于 SVM[118]、CNN-RNN[119]、BERT[120, 121] 等方法的作文评分算法相继被提出，并取得了较大的进展。这些方法都可以应用于大语言模型预训练数据过滤。由于预训练数据量非常大，并且对质量判断的准确率要求并不非常高，因此一些基于深度学习和预训练的方法还没有应用于低质过滤中。

3.2.2 冗余去除

文献 [122] 指出，大语言模型训练数据库中的重复数据，会降低大语言模型的多样性，并可能导致训练过程不稳定，从而影响模型性能。因此，需要对预训练语料库中的重复数据进行处理，去除其中的冗余部分。**文本冗余发现**（Text Duplicate Detection）也被称为文本重复检测，是自然语言处理和信息检索中的基础任务之一，其目标是发现不同粒度上的文本重复，包括句子、段落、文档等不同级别。冗余去除就是在不同的粒度上去除重复内容，包括句子、文档和数据集等粒度。

在句子级别上，文献 [123] 指出，包含重复单词或短语的句子很可能造成语言建模中引入重复的模式。这对语言模型来说会产生非常严重的影响，使模型在预测时容易陷入**重复循环**（Repetition Loops）。例如，使用 GPT-2 模型，对于给定的上下文“In a shocking finding, scientist discovered

a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.”使用束搜索（Beam Search），当设置 $b = 32$ 时，模型就会产生如下输出，进入重复循环模式。“The study, published in the Proceedings of the National Academy of Sciences of the United States of America (PNAS), was conducted by researchers from the Universidad Nacional Autónoma de México (UNAM) and the Universidad Nacional Autónoma de México (UNAM/Universidad Nacional Autónoma de México/Universidad Nacional Autónoma de México/Universidad Nacional Autónoma de México/Universidad Nacional Autónoma de ···”。由于重复循环对语言模型生成的文本质量有非常大的影响，因此在预训练数据中需要删除这些包含大量重复单词或者短语的句子。

在 RefinedWeb[60] 的构造过程中使用了文献 [124] 提出的过滤方法，进行了句子级别的过滤。该方法提取并过滤文档间超过一定长度的相同字符串。给定两个文档 x_i 和 x_j，其中存在长度为 k 的公共子串 $x_i^{a\cdots a+k} = x_j^{b\cdots b+k}$。当 $k \geqslant 50$ 时，就将其中一个子串过滤。公共子串匹配的关键是如何高效地完成字符串匹配，文献 [60] 将整个文档 $\mathcal{D}$ 转换为一个超长的字符串序列 $\mathcal{S}$，之后构造序列 $\mathcal{S}$ 的后缀数组（Suffix Array）A。该数组包含该序列中所有后缀按字典顺序排列的列表。具体而言，后缀数组 A 是一个整数数组，其中每个元素表示 $\mathcal{S}$ 中的一个后缀的起始位置。A 中的元素按照后缀的字典顺序排列。例如，序列“banana”的后缀包括“banana”“anana”“nana”“ana”“na”“a”，对应的后缀数组 A 为 $[6, 4, 2, 1, 5, 3]$。根据数组 A，可以很容易地找出相同的子串。如果 $\mathcal{S}_{i\cdots i+|s|} = \mathcal{S}_{j\cdots j+|s|}$，那么 i 和 j 在数组 A 中一定在紧邻的位置上。文献 [124] 中设计了并行的后缀数组构造方法，针对 Wiki-40B 训练数据（约包含 4GB 文本内容），使用拥有 96 核 CPU 以及 768GB 内存的服务器，可以在 140 秒内完成计算。对于包含 350GB 文本的 C4 数据集，仅需要 12 小时就可以完成后缀数组构造。

在文档级别上，大部分大语言模型依靠文档之间的表面特征相似度（例如 n-gram 重叠比例）进行检测并删除重复文档[31, 34, 60, 124]。LLaMA[34] 采用 CCNet[125] 的处理模式，先将文档拆分为段落，并把所有字母转换为小写字母、将数字替换为占位符，删除所有 Unicode 标点符号和重音符号对每个段落进行规范化处理。然后，使用 SHA-1 方法为每个段落计算一个哈希码（Hash Code），并使用前 64 位数字作为键。最后，利用每个段落的键进行重复判断。RefinedWeb[60] 先去除页面中的菜单、标题、页脚、广告等内容，仅抽取页面中的主要内容。在此基础上，在文档级别进行过滤，采用与文献 [115] 类似的方法，使用 n-gram 重复程度来衡量句子、段落及文档的相似度。如果重复程度超过预先设定的阈值，则会过滤重复段落或文档。

此外，数据集级别上也可能存在一定数量的重复情况，比如很多大语言模型预训练数据集都会包含 GitHub、Wikipedia、C4 等。需要特别注意的是，预训练数据中混入测试数据，造成数据集污染的情况。在实际产生预训练数据时，需要从句子、文档、数据集三个级别去除重复，这对于改善语言模型的训练效果具有重要的作用[14, 126]。

3.2.3 隐私消除

由于绝大多数预训练数据源于互联网,因此不可避免地会包含涉及敏感或个人信息(Personally Identifiable Information，PII）的用户生成内容，这可能会增加隐私泄露的风险[127]。如图3.3 所示，输入前缀词“East Stroudsburg Stroudsburg”，语言模型在此基础上补全了姓名、电子邮件地址、电话号码、传真号码及实际地址。这些信息都是模型从预训练数据中学习得到的。因此，非常有必要从预训练语料库中删除包含个人身份信息的内容。

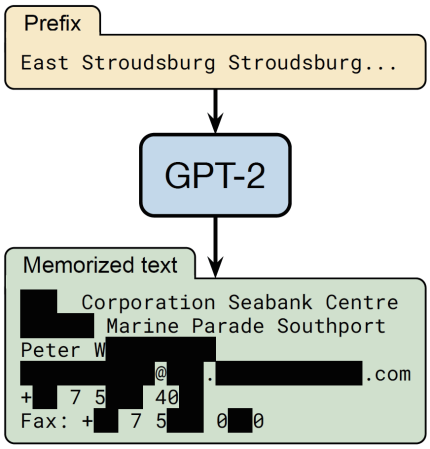

图 3.3　从大语言模型中获得隐私数据的例子[127]

删除隐私数据最直接的方法是采用基于规则的算法，BigScience ROOTS Corpus[128] 在构建过程中就采用了基于命名实体识别的方法，利用命名实体识别算法检测姓名、地址、电话号码等个人信息内容并进行删除或者替换。该方法使用了基于 Transformer 的模型，并结合机器翻译技术，可以处理超过 100 种语言的文本，消除其中的隐私信息。该方法被集成在 muliwai 类库中。

3.2.4 词元切分

传统的自然语言处理通常以单词为基本处理单元，模型都依赖预先确定的词表 $\mathbb{V}$，在对输入词序列编码时，这些词表示模型只能处理词表中存在的词。因此，使用时，如果遇到不在词表中的未登录词，模型无法为其生成对应的表示，只能给予这些**未登录词**（Out-of-Vocabulary，OOV）一个默认的通用表示。在深度学习模型中，词表示模型会预先在词表中加入一个默认的“[UNK]”（unknown）标识，表示未知词，并在训练的过程中将 [UNK] 的向量作为词表示矩阵的一部分一起训练，通过引入某些相应机制来更新 [UNK] 向量的参数。使用时，对全部未登录词使用 [UNK] 向量作为表示向量。此外，基于固定词表的词表示模型对词表大小的选择比较敏感。当词表过小时，未登录词的比例较高，影响模型性能；当词表过大时，大量低频词出现在词表中，这些词的词向量很难