AI INFRA · HANDBOOK

用大模型逐讲编写的 AI 基础设施交互式课程

《AI Infra-HandBook》

LLM 推理 · 工程实践 · 从原理讲到能跑的引擎

作者 孙晓辉

视频号 扶摇 Sun

V1.0 · 2026年11月17日

在线学习 https://ai-infra-handbook.sunxiaohui2022.workers.dev/

GitHub https://github.com/sunxiaohui2025/AI-Infra-Handbook

目录

全书 6 章 20 篇 · 本整理版收录 153 讲（已写 20 篇）

第 1 篇 · 数学基础 10

01	向量	约 20 分钟	11
02	点积与余弦相似度	约 25 分钟	20
03	矩阵与矩阵乘法	约 30 分钟	30
04	线性变换的几何意义	约 30 分钟	39
05	张量与形状	约 30 分钟	47
06	指数、对数与 softmax	约 30 分钟	55
07	概率分布与采样	约 25 分钟	62
08	导数、偏导与链式法则	约 30 分钟	67
09	三角函数与旋转	选读	74
10	浮点数与数值精度	约 35 分钟	81
11	计算量与 FLOPs	约 25 分钟	87

第 2 篇 · 神经网络基础 93

12	感知机	约 30 分钟	94
13	神经元与线性层	约 30 分钟	108
14	激活函数	约 30 分钟	118
15	多层感知机	约 30 分钟	128
16	损失函数	约 30 分钟	138
17	梯度与梯度下降	约 30 分钟	147
18	反向传播	约 35 分钟	156
19	优化器	约 30 分钟	165
20	完整的训练循环	约 30 分钟	174

第 3 篇 · 从文本到向量 183

21 分词与 BPE 约 30 分钟 184

22 词嵌入 Embedding 约 30 分钟 192

23 预测下一个 token 约 25 分钟 201

24 自回归生成 约 25 分钟 209

第 4 篇 · Transformer 核心 217

25 位置编码 约 25 分钟 218

26 自注意力 约 35 分钟 225

27 因果掩码 约 25 分钟 235

28 多头注意力 约 25 分钟 243

29 残差连接 约 25 分钟 252

30 LayerNorm 约 20 分钟 259

31 FFN 前馈网络 约 25 分钟 266

32 Transformer Block 约 20 分钟 272

33 输出层 约 20 分钟 281

34 完整前向传播 约 30 分钟 288

35 参数量与计算量 约 30 分钟 296

第 5 篇 · 现代 LLM 架构 304

36 Pre-Norm 与 RMSNorm 约 25 分钟 305

37 RoPE 旋转位置编码 约 30 分钟 311

38 SwiGLU 与门控 FFN 约 20 分钟 319

39 MQA 与 GQA 约 30 分钟 326

40 MLA 多头潜在注意力 约 35 分钟 333

41 MoE 混合专家 约 35 分钟 340

42 长上下文技术 选读 351

43 架构演进对照 约 20 分钟 359

44 混合架构与线性注意力 选读 365

第 6 篇 · 训练全流程 372

45 预训练与数据 约 20 分钟 373

46 Scaling Laws 约 20 分钟 380

47 监督微调 SFT 约 20 分钟 387

48 RLHF 与 DPO 约 25 分钟 394

49 推理模型与强化学习 约 25 分钟 401

50 LoRA 参数高效微调 约 20 分钟 407

51 模型评测 约 20 分钟 414

第 7 篇 · 从零实现一个 GPT 模型 420

52 g0 · 语料与分词器 421

53 g1 · 数据加载器与嵌入 433

54 g2 · 因果多头注意力 441

55 g3 · 组装 GPT 447

56 g4 · 预训练 456

57 g5 · 解码与存档 466

58 g6 · 装进 GPT-2 的权重 480

59 g7 · 指令微调 488

第 8 篇 · 硬件与性能基础 499

60 GPU 架构：SM 与 Tensor Core 约 30 分钟 500

61 存储层级：HBM 与 SRAM 约 30 分钟 506

62 Roofline 模型 约 30 分钟 512

63 低精度计算 约 25 分钟 518

64 互联：PCIe、NVLink、InfiniBand、RoCE 约 30 分钟 523

65 scale-up、scale-out 与超节点 约 30 分钟 529

66 性能分析与 Profiling 约 30 分钟 535

67 理论值对上实测值 约 30 分钟 542

第 9 篇 · 推理基础 549

68 Prefill 与 Decode 约 25 分钟 550

69 KV Cache 约 30 分钟 557

70 TTFT、TPOT 与吞吐 约 25 分钟 565

71 显存账本 约 25 分钟 571

72 推理成本模型 约 25 分钟 577

73 解码策略 约 20 分钟 584

第 10 篇 · 推理引擎与调度 591

74 Continuous Batching 约 30 分钟 592

75 PagedAttention 约 35 分钟 599

76 Prefix Cache 约 30 分钟 610

77 Chunked Prefill 约 30 分钟 622

78 调度与抢占 约 35 分钟 629

79 多 LoRA 服务 选读 638

80 结构化输出与约束解码 选读 646

81 一个请求的完整生命周期 约 30 分钟 655

82 推理压测实践 约 30 分钟 665

第 11 篇 · 算子与 Kernel 673

83 算子融合 约 30 分钟 674

84 FlashAttention 约 35 分钟 686

85 Decode 阶段的注意力 Kernel 约 30 分钟 697

86 CUDA 编程模型 约 35 分钟 708

87 矩阵乘法 Kernel 与 Tensor Core 721

88 Triton 入门 约 35 分钟 727

89 CUDA Graph 与调度开销 选读 740

第 12 篇 · 模型压缩与加速 750

90 量化基础 约 35 分钟 751

91 权重量化：GPTQ 与 AWQ 约 35 分钟 759

92 激活量化与 SmoothQuant 约 30 分钟 767

93 FP8 与 FP4 约 35 分钟 773

94 KV Cache 量化 约 30 分钟 780

95 KV Cache 压缩与驱逐 选读 786

96 投机解码 约 35 分钟 791

97 EAGLE 与 Medusa 选读 798

98 蒸馏与剪枝 选读 805

第 13 篇 · 分布式与集合通信 812

99 为什么需要多卡 约 25 分钟 813

100 集合通信原语 约 25 分钟 819

101 Ring 与 Tree All-Reduce 约 30 分钟 826

102 通信代价模型 约 30 分钟 833

103 数据并行与 ZeRO/FSDP 约 30 分钟 840

104 混合精度训练与激活重计算 846

105 张量并行 约 35 分钟 852

106 流水并行 约 30 分钟 858

107 序列并行与上下文并行 选读 865

108 专家并行与 All-to-All 约 35 分钟 872

109 并行策略与拓扑映射 约 35 分钟 880

第 14 篇 · 分布式推理与集群 886

110 推理中的张量并行 约 30 分钟 887

111 MoE 推理与大规模专家并行 约 35 分钟 894

112 PD 分离 约 35 分钟 908

113 KV Cache 传输与分层存储 约 30 分钟 922

114 缓存感知路由与负载均衡 约 30 分钟 934

115 超节点上的推理 944

116 SLO 与容量规划 约 30 分钟 963

第 15 篇 · 手写迷你推理引擎 975

117 一个推理引擎的组成 976

118 从权重文件到第一个 token 982

119 KV Cache 992

120 Continuous Batching 1000

121 分页 KV 管理 1011

122 前缀缓存 1024

123 权重量化 1029

124 投机解码 1036

125 时间花在哪里 1045

第 16 篇 · 解读 nano-vllm 1051

126 nano-vllm 的整体结构 1052

127 nano-vllm 多做了什么 1063

128 动手改 nano-vllm 1084

第 17 篇 · 主流推理引擎 1102

129 vLLM 的架构 1103

130 vLLM 的分布式服务 1120

131 SGLang 的架构 1138

132 其他引擎与规模化部署 1152

133 怎么读大型引擎的源码 1158

第 18 篇 · 适配原理：把模型跑通 1164

134 换一块卡要重做什么 1165

135 国产与 NVIDIA 硬件差异 1175

136 编程模型与 CUDA 迁移 1184

137 框架怎么接入新硬件 1193

138 图模式与编译 1202

139 精度对齐 1210

140 算子缺口：盘点与自写 1223

141 实操 · 把 miniinfer 移植到非 CUDA 1231

第 19 篇 · 调优原理：把模型跑快 1243

142 在新硬件上做性能分析 1244

143 矩阵乘法与注意力的适配 1252

144 用 Triton 与厂商语言写 kernel 1261

145 低精度与量化的取舍 1267

146 集合通信库与多卡互联 1274

147 推理引擎的硬件插件 1280

第 20 篇 · 案例：海光 DCU 1287

148 海光 DCU 的软硬件栈 1288

149 从 CUDA 到 DCU：迁移与版本矩阵 1296

150 读 vllm-plugin-das：平台与运行时补丁 1303

151 DCU 上的矩阵乘：hipBLASLt 与小 M kernel 1316

152 DCU 上的注意力、量化与通信 1326

153 模型上线 DCU 的完整清单 1335

AI INFRA · HANDBOOK

第 一 篇

数学基础

收录 11 讲 · 第 1—11 讲

第01讲 向量

计算机其实只会做一件事：算数。加、减、乘、除，再加上比大小。它不认识“苹果”，也不知道“我 爱 学 AI”这句话是什么意思。

可是大模型偏偏要处理文字、图片、声音。那它是怎么做到的？答案的第一步非常朴素：**先把要处理的东西变成一串数字**。变成数字以后，计算机就能对它做加减乘除了。

这串数字，就是这一讲的主角——向量。这一讲我们只做三件事：看看“变成一串数字”具体是怎么回事；看看一串数字能做哪两种最基本的运算；再看看在真实的大模型里，这串数字有多长、要占多少显存。

先给水果打个分

先不谈大模型，从一个更容易想象的例子开始。

假设你开了一家水果店，想让电脑帮你管理水果。电脑不认识水果，所以你得用数字来描述每一种水果。最简单的办法是挑几个特征打分，比如“甜度”和“酸度”，各打 0 到 10 分：

水果	甜度	酸度
苹果	7	4
柠檬	2	9
香蕉	8	1
葡萄柚	4	7

这样一来，苹果就可以写成 (7, 4)，柠檬写成 (2, 9)。电脑不需要知道“苹果”是什么，它只需要记住 (7, 4) 这两个数。

这里有一个很容易忽略、但非常重要的约定：**顺序是固定的**。我们约好第一个数是甜度、第二个数是酸度，那 (7, 4) 和 (4, 7) 就是两种完全不同的水果——前者偏甜，后者偏酸。所以一串数字不只是“几个数”，而是“几个按约定顺序排好的数”。

把水果放进坐标系

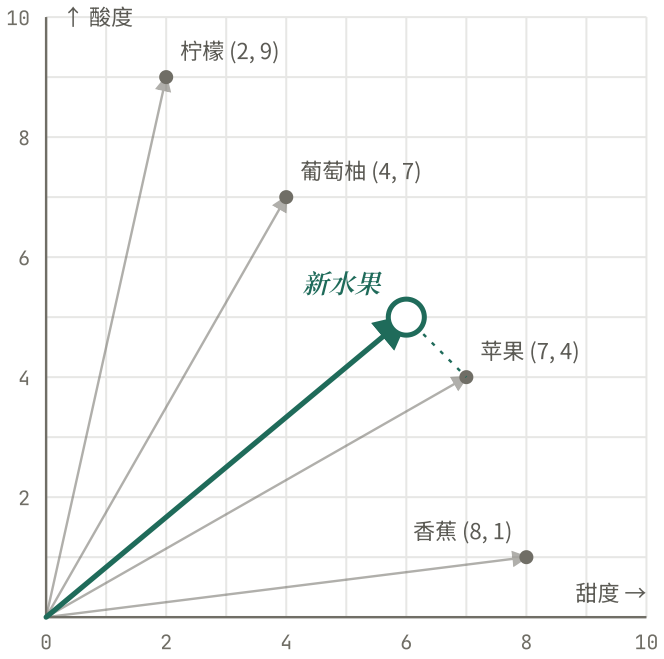
两个数，正好可以当成平面上的坐标。把甜度当横轴、酸度当纵轴，每种水果就是平面上的一个点。

从原点 (0, 0) 向这个点画一条带箭头的线段，就得到了向量的另一种画法：**箭头**。点和箭头说的是同一件事，只是看的角度不同。当成点时，我们关心它“在哪里”；当成箭头时，我们关心它“朝哪个方向、有多长”。后面讲向量运算时，箭头的画法会更直观。

下面的演示里，灰色箭头是表格里的四种水果，绿色箭头是一种“新水果”。拖动绿色箭头的末端（也可以点一下圆点后用方向键微调），看看右边的数字怎么变。

拖动“新水果”，看它的坐标和长度

⇆ 对照



坐标	**(6, 5)**
含义	甜度 6，酸度 5
长度	√(6² + 5²) = 7.81
最近的	**苹果**，距离 √((6-7)² + (5-4)²) = 1.41

重置

图 1-1　甜度–酸度坐标系里的水果。拖动绿色箭头，右侧会实时算出坐标、长度和离它最近的水果。

默认位置的新水果是 (6, 5)。右边告诉我们三件事。

第一是**坐标**：(6, 5)，也就是甜度 6、酸度 5。

第二是**长度**：箭头从原点到 (6, 5) 有多长。这就是勾股定理，$\sqrt{(6^2 + 5^2)} \approx 7.81$。长度描述的是“这个向量有多大”，在水果的例子里大致对应“味道有多浓”。

第三是**离它最近的水果**：苹果。苹果在 (7, 4)，两点之间的距离是 $\sqrt{((6-7)^2 + (5-4)^2)} = \sqrt{2} \approx 1.41$，比离其他几种水果都近。换句话说，这种新水果“尝起来最像苹果”。

最后这一点值得停下来想一想。我们原本只是想让电脑记住水果，结果顺手得到了一个能力：**“像不像”变成了一个可以计算的问题**。两种东西像不像，就看它们的向量离得近不近。大模型判断两个词意思是否相近、一句话和哪段资料最相关，用的都是这个思路，只不过量“近不近”

的方法更讲究一些，下一讲专门讲。

> **要点**
> 把东西变成向量，最大的好处是：它们之间的关系（像不像、差多少）变成了可以算出来的数字。

向量就是一串有顺序的数

现在可以给向量下一个正式一点的定义了。

n 个数按固定顺序排成一串，就叫一个 n 维向量。写成

$$\mathbf{v} = (v_1, v_2, \dots, v_n)$$

其中 v_1 是第 1 个数，v_2 是第 2 个数，依此类推，每一个叫做向量的一个**分量**。分量的个数 n 叫做向量的**维度**。水果的向量有甜度、酸度两个分量，所以是 2 维的。

你在大学的线性代数里学过更抽象的定义（向量空间、满足某些运算规律的元素……）。那些都对，但在这门课里，绝大多数时候只需要记住最朴素的这一层：**向量就是一串有顺序的数，每个位置有它自己的含义**。

向量的长度也可以推广到 n 维：把每个分量平方、加起来、再开方，

$$|\mathbf{v}| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

苹果 (7, 4) 的长度就是 √(49 + 16) = √65 ≈ 8.06。这个“长度”在大模型里会反复出现：模型每算完一层，都要把向量的长度拉回到差不多的大小，防止数字越算越大或越算越小。这件事叫归一化，第 30 讲和第 36 讲会讲到。

两个向量相加

有了向量，就可以对它做运算了。最基本的运算只有两种：**加法**和**数乘**。先看加法。

还是水果店。你想做一款“加糖苹果”：在苹果的基础上加一勺糖。一勺糖只增加甜度、不影响酸度，可以写成向量 (2, 0)。加糖之后的苹果，就是把两个向量的**对应分量**分别相加：

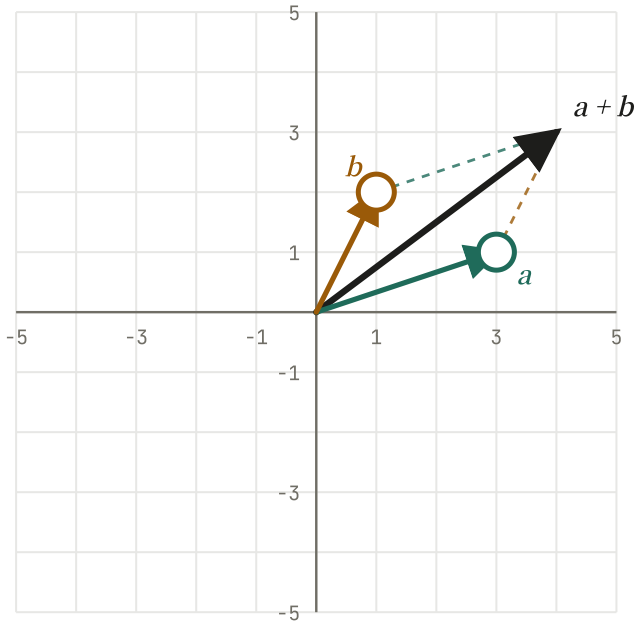
$$(7,4)+(2,0)=(7+2,4+0)=(9,4)$$

甜度从 7 变成 9，酸度还是 4。规则就这么简单：第 1 个分量和第 1 个分量相加，第 2 个和第 2 个相加。两个向量的维度必须相同，否则没有“对应分量”可以加。

几何上，向量加法有一个很好记的画法：**把第二个箭头接在第一个箭头的末端**，从原点走到最终的终点，就是两者之和。下面的演示里，你可以分别拖动 a 和 b：

拖动 a 和 b，看 a + b

⇆ 对照



a	(3, 1)
b	(1, 2)
a + b	(3 + 1, 1 + 2) = (4, 3)

虚线：把 b 接在 a 的末端（或把 a 接在 b 的末端），走到的是同一个点

重置

图 1-2　向量加法。黑色箭头是 a + b；两条虚线表示“把 b 接在 a 末端”和“把 a 接在 b 末端”，它们走到同一个点。

默认 a = (3, 1)，b = (1, 2)，所以 a + b = (4, 3)。试着拖动一下，注意两件事。

一是两条虚线总是在 a + b 的终点汇合。先走 a 再走 b，和先走 b 再走 a，到达的是同一个地方，也就是 a + b = b + a。四条线段围成一个平行四边形，a + b 就是它的对角线。

二是试着把 b 拖到和 a 方向相反的位置，比如 b = (−3, −1)。这时 a + b 变成 (0, 0)，两个向量正好抵消。

把向量放大、缩小、反过来

第二种运算是**数乘**：用一个普通的数 k 去乘向量，规则是**每个分量都乘以 k**。

$$k \cdot (v_1, v_2) = (kv_1, kv_2)$$

比如 0.5 × (7, 4) = (3.5, 2)：甜度和酸度都减半，可以想象成把苹果汁兑了一倍的水，味道变淡了，但“甜里带一点酸”的比例没变。

这正是数乘在几何上的意思：**方向不变，长度变成原来的 |k| 倍**。k 是负数时，箭头会翻转到相反的方向。在下面的演示里拖动 a，再拉动 k 的滑块：

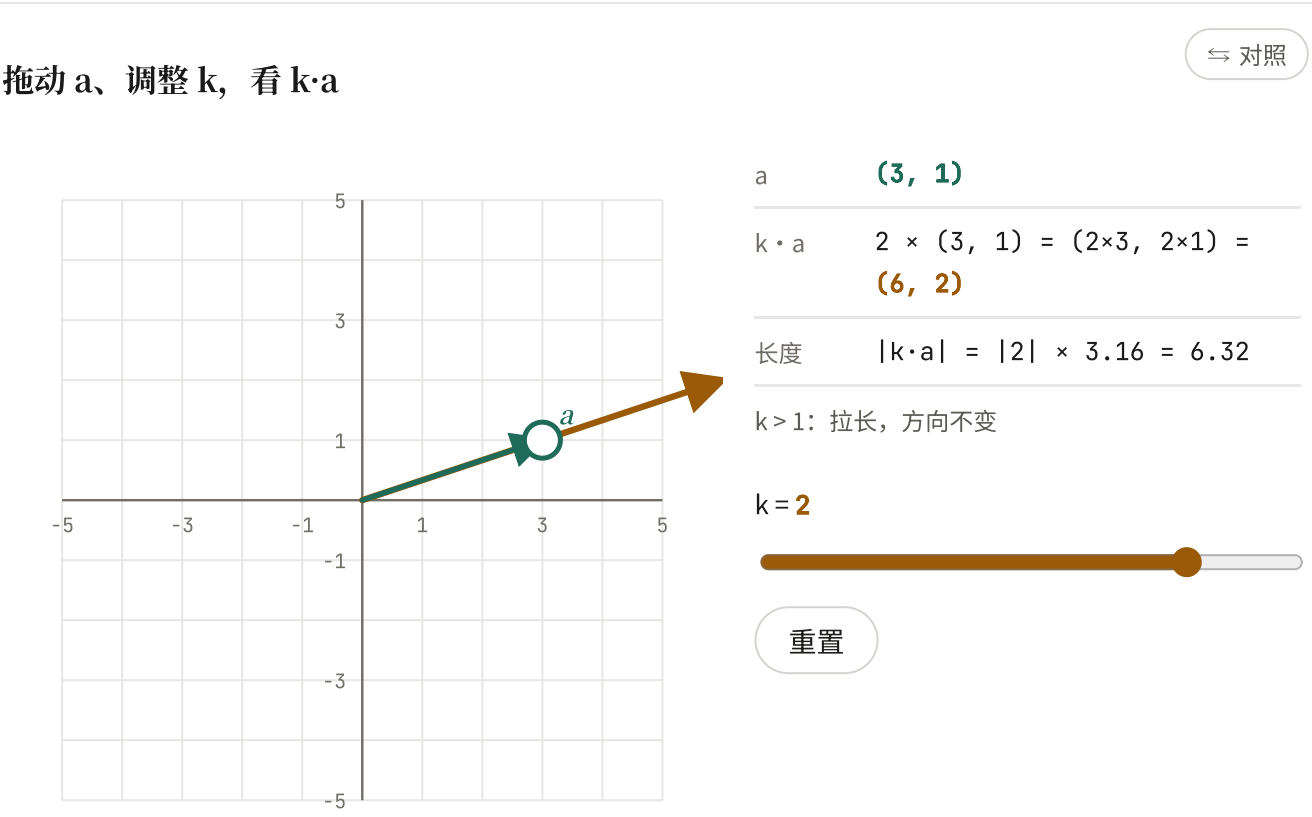


图 1-3　数乘。绿色是 a，琥珀色是 k · a。拉动滑块，注意 k 为负数、为 0、在 0 和 1 之间时分别发生了什么。

把 k 从 2 慢慢拉到 −2，你会看到：k > 1 时箭头拉长；0 < k < 1 时缩短；k = 0 时整个箭头缩成原点上的一个点，这个所有分量都是 0 的向量叫**零向量**；k 变成负数后，箭头掉头指向反方向。不管 k 是多少，k·a 始终在 a 所在的那条直线上。

专栏　为什么只讲这两种运算

加法和数乘看起来太简单了，为什么要专门拿出来讲？因为大模型里大量的计算，拆开看都是这两种运算的组合：先把几个向量各自乘上一个数，再加起来，即 $k_1\mathbf{a} + k_2\mathbf{b} + \cdots$，这叫**线性组合**。

举两个后面会遇到的例子。第 26 讲的注意力机制，输出就是把若干个向量按“注意力权重”做线性组合；第 29 讲的残差连接，是把一层的输入和输出直接相加。等你读到那里，会发现它们用到的就是这一节的两张图。

从 2 维到 4096 维

只用甜度和酸度描述水果显然太粗糙了。苹果和梨可能甜度、酸度都差不多，但一个脆、一个软，香气也不一样。想描述得更准确，就得加特征：脆度、香气、水分……每多一个特征，向量就多一个分量，维度就加 1。

3 维向量还能在空间里画出来，4 维以上就画不出来了。但这一点也不影响计算：**加法还是对应分量相加，数乘还是每个分量乘 k，长度还是平方和开根号**。规则一个字都不用改，只是分量变多了。这正是把东西变成向量的另一个好处——维度再高，算法都一样。

大模型处理文字，也是这个思路：**给词表里的每个词分配一个向量**。我们全站使用的玩具模型 TinyLM 只有 6 个词，每个词用一个 4 维向量表示。把这 6 个向量一行一行排起来，就是下面这张表：

⇆ 对照

E 6×4 · 每行是一个词的向量

	1	2	3	4
我	0.7	-0.6	-0.9	0.3
爱	-0.3	-0.1	-0.8	-0.2
学	0.3	-0.3	0.6	0.6
AI	0.4	0.8	0.4	-0.6
推理	0.7	0.3	-0.8	-0.4
<eos>	-0.6	0.9	0.4	0.8

图 1-4　TinyLM 的词向量表。高亮的一行是“我”的向量 (0.7, −0.6, −0.9, 0.3)。数字由 tinylm/tinylm.py 用固定随机种子生成。

“我”就是 (0.7, −0.6, −0.9, 0.3)，“爱”就是 (−0.3, −0.1, −0.8, −0.2)。和水果不同，这 4 个分量没有“甜度”“酸度”这样的名字。这些数字现在是随机生成的，还没有任何含义；模型经过训练以后，它们会被调整到“意思相近的词，向量也相近”的位置上。这个过程第 22 讲再讲。

一句话“我 爱 学 AI”有 4 个词，于是就变成了 4 个 4 维向量。把它们一行一行叠起来，就得到一张 4×4 的数字表——这就是第 3 讲要讲的矩阵。

真实的大模型里，每个词的向量要长得多。以大纲里用作例子的 Llama-3-8B 为例，每个词是一个 **4096 维**的向量（这个数字写正文时请对照官方配置核实）。4096 个分量，每一个都没有人能读懂的名字，但加法、数乘的规则和我们刚才拖来拖去的 2 维向量完全一样。

一个向量要占多少显存

到这里，我们可以换一个 Infra 的角度看向量：**它要占多少内存？**

向量的每个分量是一个浮点数。存一个浮点数要几个字节，取决于用什么格式：常见的 FP32 要 4 字节，大模型推理时常用的 BF16 只要 2 字节（为什么可以这样省、省了会不会出问题，第 10 讲专门讲）。

于是一个 4096 维、BF16 格式的向量要占 4096 × 2 = **8192 字节，也就是 8 KiB**。单看一个，好像很小。但大模型从来不只处理一个向量：

- 一段 1000 个词的输入，就是 1000 个这样的向量，合计 1000 × 8192 字节 ≈ **7.8 MiB**。而且模型有几十层，每一层都要处理这样一批向量。
- 词表里的每个词都要准备一个向量。Llama-3-8B 的词表大约有 12.8 万个词（128256 个，同样需要核实），光是这一张词向量表就要 128256 × 4096 × 2 字节 ≈ **0.98 GiB**。

> **要点**
> 向量的维度 d 是整个课程里最基本的一个数。显存要多少、计算量有多大，几乎都要从“每个向量有 d 个数”开始算。

用代码试一试

上面所有的数字，都可以用几行 NumPy 验证。先在 `tinylm/` 目录里运行一次 `python tinylm.py` 生成 `tinylm.json`，再在同一目录运行：

```python
import json
import numpy as np

apple = np.array([7, 4])          # 苹果：甜度 7，酸度 4
sugar = np.array([2, 0])          # 一勺糖：只加甜度
print(apple + sugar)              # 加法：对应分量相加
print(0.5 * apple)                # 数乘：每个分量乘 0.5
print(np.linalg.norm(apple))      # 长度：√(7² + 4²)

tiny = json.load(open("tinylm.json", encoding="utf-8"))
vocab = tiny["config"]["vocab"]
E = np.array(tiny["weights"]["embedding"])   # 6 个词，每个 4 维
print(E.shape)
print(E[vocab.index("我")])                  # “我”的向量
print(E[vocab.index("我")] + E[vocab.index("爱")])

d = 4096                                     # Llama-3-8B 的隐藏维度（待核实）
print(d * 2, "字节")                          # 一个 BF16 向量
print(1000 * d * 2 / 2**20, "MiB")           # 1000 个 token 的向量
print(128256 * d * 2 / 2**30, "GiB")         # 词表里每个词一个向量
```

输出：

```
[9 4]
[3.5 2. ]
8.06225774829855
(6, 4)
[ 0.7 -0.6 -0.9  0.3]
[ 0.4 -0.7 -1.7  0.1]
8192 字节
7.8125 MiB
0.978515625 GiB
```

逐行对一下：加糖苹果 $(9, 4)$，兑水苹果 $(3.5, 2)$，苹果的长度 $\sqrt{65} \approx 8.06$，和正文一致。`E.shape` 是 $(6, 4)$，即 6 个词、每个 4 维。倒数第四行把“我”和“爱”两个 4 维向量相加，规则和 2 维时一模一样：$0.7 + (-0.3) = 0.4$，$-0.6 + (-0.1) = -0.7$，依此类推。

容易混淆的地方

1. **“4096 维向量”在 NumPy 里是“1 维数组”**。这里的两个“维”不是一回事。向量的维度说的是它有几个分量；数组的维数（`ndim`）说的是它有几个轴。一个 4096 维的向量是一个形状为 `(4096,)` 的一维数组；而上面的词向量表 `E` 是二维数组，形状 `(6, 4)`。这个区别第 5 讲还会专门讲。
2. **不是每个分量都有人能看懂的含义**。水果的例子里，每个分量都有名字。但大模型学出来的向量，单独看某一个分量通常说明不了什么，有意义的是整个向量和其他向量之间的关系。
3. **点和箭头是同一个向量**。画成点还是画成箭头，只是为了看问题方便，数学上没有区别。

回顾

这一讲从水果店出发，看到了把东西变成数字的完整过程：先约定好每个位置代表什么特征，再给每个东西打分，得到一串有顺序的数，也就是向量。向量可以画成点或箭头；它有长度；可以相加，也可以乘以一个数，而这两种运算在任何维度下规则都一样。大模型做的第一件事，就是给每个词准备一个这样的向量；在真实模型里它有几千维，光是一张词向量表就要占将近 1 GiB 显存。

还有一个问题留着没细讲。在水果的演示里，我们用“距离”判断新水果像哪种水果。可是大模型里，判断两个向量像不像，更常用的是另一种量法：把两个向量的对应分量相乘再相加。为什么要这么量？它和两个箭头之间的夹角有什么关系？这就是下一讲的内容：**点积与余弦相似度**。

第02讲 点积与余弦相似度

01 向量

上一讲结尾，我们用“距离”判断一种新水果像哪种水果：离苹果最近，就说它像苹果。这个办法很自然，在水果的例子里也确实好用。

可是到了大模型里，判断两个向量像不像，最常用的并不是距离，而是一种叫**点积**的量法，以及由它派生出来的**余弦相似度**。这一讲就来回答两个问题：距离到底哪里不够好？点积又是怎么量“像不像”的？

用距离量，会出什么问题

换一个例子。假设我们想让电脑判断两篇文章是不是在讲同一件事。最简单的办法是数关键词：每篇文章数一数“GPU”出现了几次、“美食”出现了几次，得到一个 2 维向量。

现在有三篇文章：

文章	“GPU”出现次数	“美食”出现次数	向量
一篇讲 GPU 的短文	4	1	(4, 1)
一篇讲 GPU 的长文	12	3	(12, 3)
一篇讲美食的文章	1	5	(1, 5)

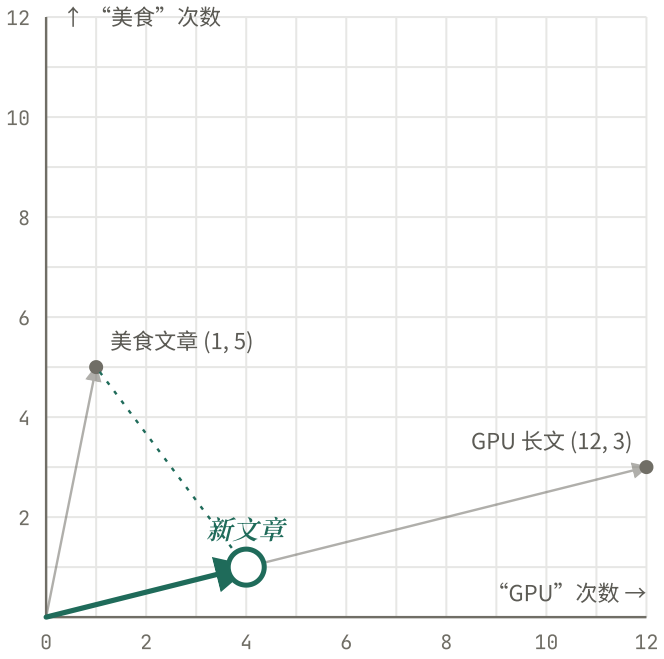
凭常识就知道：前两篇都在讲 GPU，只是一篇短、一篇长，长文里每个词出现的次数都是短文的 3 倍而已；第三篇讲的是完全不同的话题。所以“GPU 短文”应该更像“GPU 长文”。

我们用上一讲的距离来算一算。短文到长文的距离是 $\sqrt{(4-12)^2 + (1-3)^2} = \sqrt{68} \approx 8.25$；短文到美食文章的距离是 $\sqrt{(4-1)^2 + (1-5)^2} = \sqrt{25} = 5$。**按距离，短文反而更像美食文章**——和常识正好相反。

在下面的演示里亲手试一下。绿色箭头是“新文章”，默认就是那篇 GPU 短文 (4, 1)。右边的表同时列出它到两篇文章的距离和“余弦相似度”（这个量马上就会讲到），每一列里“最像”的那一项会标绿：

拖动“新文章”，同时比较距离和余弦相似度

⇆ 对照



新文章 (4, 1)

	距离	余弦相似度
GPU 长文	8.25	1
美食文章	5	0.43

按距离，最像的是“美食文章”；按余弦相似度，最像的是“GPU 长文”——两种量法给出了不同的答案

重置

图 2-1　同一篇 GPU 短文，按距离量最像美食文章，按余弦相似度量最像 GPU 长文。拖动绿色箭头，看两种量法什么时候一致、什么时候分歧。

问题出在哪里？看图就明白了。GPU 长文 (12, 3) 和短文 (4, 1) 指向**完全相同的方向**，只是长文的箭头长了 3 倍——上一讲学过，这正是数乘：(12, 3) = 3 × (4, 1)。距离把“长度不同”也算成了差别，而文章长短恰恰是我们不关心的。

方向比长度更重要

从这个例子可以看出，判断两篇文章“讲的是不是同一件事”，真正有用的信息是**箭头的方向**，而不是箭头的长短。方向一致，说明各个关键词出现的比例一致；长短不同，只说明文章篇幅不同。

那么怎么量两个方向是否一致？最直接的想法是量两个箭头之间的**夹角** θ：夹角越小，方向越接近。$\theta = 0°$ 时两个箭头完全同向；$\theta = 90°$ 时互相垂直，可以理解为“毫不相关”；$\theta = 180°$ 时方向完全相反。

问题是，直接量角度并不方便——给你两串几千维的数字，角度要怎么量？这时候就轮到点积出场了。它只用乘法和加法，就能把夹角的信息算出来。

点积：对应分量相乘再相加

两个维度相同的向量 $\mathbf{a}$ 和 $\mathbf{b}$，它们的点积定义为：**把对应分量相乘，再全部加起来**。

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

注意结果是**一个数**，不是向量。这一点和上一讲的加法、数乘不同：加法和数乘的结果还是向量，点积则把两个向量“压缩”成了一个数。

拿刚才的文章算一下：

$$(4,\,1) \cdot (12,\,3) = 4 \times 12 + 1 \times 3 = 51 \qquad (4,\,1) \cdot (1,\,5) = 4 \times 1 + 1 \times 5 = 9$$

GPU 短文和 GPU 长文的点积是 51，和美食文章只有 9。这次的排序符合常识了。但只看这两个数还说明不了太多：51 大，有一部分原因只是长文的数字本身就大。点积和夹角之间到底是什么关系，我们需要再往前走一步。

点积和夹角的关系

结论先写出来：

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}|\,|\mathbf{b}| \cos\theta$$

点积等于两个向量的长度相乘，再乘以夹角的余弦。这个式子是这一讲的核心。你在大学里大概见过它，这里我们用余弦定理把它推一遍，每一步都写出来。

把 $\mathbf{a}$、$\mathbf{b}$ 画成从原点出发的两个箭头，再把两个箭头的末端连起来，就得到一个三角形。第三条边就是向量 $\mathbf{a} - \mathbf{b}$，它的长度是 $|\mathbf{a} - \mathbf{b}|$。

第一步，对这个三角形用余弦定理。夹角 θ 对着的边是 $|\mathbf{a} - \mathbf{b}|$，所以

$$|\mathbf{a} - \mathbf{b}|^2 = |\mathbf{a}|^2 + |\mathbf{b}|^2 - 2\,|\mathbf{a}|\,|\mathbf{b}| \cos\theta$$

第二步，换一种方法算左边。上一讲说过，长度的平方就是各分量平方之和。以 2 维为例：

$$|\mathbf{a}-\mathbf{b}|^2=(a_1-b_1)^2+(a_2-b_2)^2$$

把两个平方展开：

$$=a_1^2-2a_1b_1+b_1^2+a_2^2-2a_2b_2+b_2^2$$

把同类的项归到一起：$a_1^2+a_2^2$ 正好是 $|\mathbf{a}|^2$，$b_1^2+b_2^2$ 正好是 $|\mathbf{b}|^2$，剩下的两项是 $-2(a_1b_1+a_2b_2)$，括号里的正是点积。所以

$$|\mathbf{a}-\mathbf{b}|^2=|\mathbf{a}|^2+|\mathbf{b}|^2-2\,\mathbf{a}\cdot\mathbf{b}$$

第三步，对比第一步和第二步的两个式子。左边一样，右边的 $|\mathbf{a}|^2+|\mathbf{b}|^2$ 也一样，于是剩下的部分必须相等：$2\,\mathbf{a}\cdot\mathbf{b}=2\,|\mathbf{a}|\,|\mathbf{b}|\cos\theta$，两边除以 2 就得到了结论。

这个推导在 n 维时一字不改地成立：展开平方时，每个分量各自贡献一项，归类后依然是“两个长度的平方”减去“两倍点积”。

要点

点积里同时混着两样东西：两个向量的长度，和它们之间夹角的余弦。长度越长、方向越一致，点积越大。

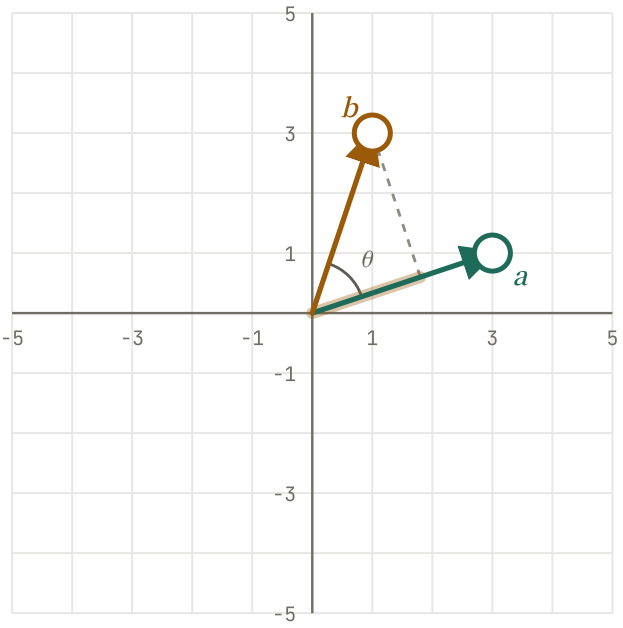
拖动两个向量看一看

有了 $\mathbf{a}\cdot\mathbf{b}=|\mathbf{a}|\,|\mathbf{b}|\cos\theta$，就能看出点积的正负直接反映方向关系：长度总是正的，所以点积的符号和 $\cos\theta$ 的符号一样。

在下面的演示里拖动 a 和 b。右边会逐项算出点积、两个长度、$\cos\theta$ 和夹角；图里的弧线标出夹角 θ，那条半透明的粗线是 b 在 a 方向上的投影——可以想象成从 b 的末端往 a 所在的直线上垂直打一束光，b 在这条直线上留下的“影子”。

拖动 a 和 b，看点积和夹角

⇆ 对照



a · b	3×1 + 1×3 = **6**
长度	|a| = 3.16, |b| = 3.16
cos θ	6 ÷ (3.16 × 3.16) = **0.6**
夹角 θ	53.13°

点积 > 0：夹角小于 90°，大致同向。粗线是 b 在 a 方向上的投影（影子）

重置

图 2-2　点积、夹角与投影。默认 a = (3, 1)、b = (1, 3)，a · b = 6，夹角约 53°。试着把 b 拖到和 a 垂直、和 a 反向的位置。

默认状态下 **a** = (3, 1)，**b** = (1, 3)。点积是 3×1 + 1×3 = 6；两个向量的长度都是 $\sqrt{10} \approx 3.16$；于是 $\cos\theta = 6 \div (3.16 \times 3.16) = 0.6$，夹角约 53°。

接着做三个小实验：

- **把 b 拖到 (−1, 3)**。点积变成 3×(−1) + 1×3 = 0，夹角正好 90°，投影的影子缩成了一个点。**点积为 0，就表示两个向量互相垂直**。
- **把 b 拖到 (−3, −1)**，也就是 a 的正对面。点积变成 −9 − 1 = −10，是负数，夹角 180°。点积为负，说明两个向量大致反向。
- **把 b 拖到 (1.5, 0.5)**，和 a 同向、长度是 a 的一半。点积是 4.5 + 0.5 = 5；再拖到 (3, 1)，和 a 完全重合，点积变成 10。方向没变，只是长度变了，点积就跟着变——点积对长度是敏感的。

最后一个实验说明了点积的局限：它既量方向，也量长度。想只比较方向，就得把长度的影响去掉。

余弦相似度：去掉长度的影响

去掉长度的办法很直接：既然 $\mathbf{a}\cdot\mathbf{b} = |\mathbf{a}|\,|\mathbf{b}|\cos\theta$，那就把两个长度除掉，只留下 $\cos\theta$。

$$\cos\theta = \frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{a}|\,|\mathbf{b}|}$$

这个量叫做**余弦相似度**（cosine similarity）。它的取值范围是 −1 到 1：1 表示方向完全相同，0 表示互相垂直，−1 表示方向完全相反。不管向量本身有多长，结果都落在这个范围里。

回到文章的例子：

- GPU 短文和 GPU 长文：$51 \div (\sqrt{17} \times \sqrt{153}) = 51 \div 51 = \mathbf{1}$。方向完全相同，余弦相似度达到最大值。
- GPU 短文和美食文章：$9 \div (\sqrt{17} \times \sqrt{26}) \approx 9 \div 21.02 \approx \mathbf{0.43}$。

这次的结论和常识完全一致。回到图 2-1 再拖一拖，你会发现：只要沿着 GPU 长文的方向拖，不管拖多远，余弦相似度一直是 1；而距离会随着你拖动不停变化。

> **专栏　先把长度变成 1**
>
> 如果事先把所有向量的长度都缩放成 1（每个向量除以它自己的长度，这叫**归一化**），那么 $|\mathbf{a}| = |\mathbf{b}| = 1$，点积就直接等于余弦相似度，连除法都省了。
>
> 很多做“语义检索”的系统就是这么做的：先把每段文字变成一个向量并归一化，存起来；用户提问时，把问题也变成归一化的向量，再和库里的向量逐个算点积，点积最大的几段就是最相关的资料。第 22 讲讲词嵌入时还会提到这类用法。

在 TinyLM 的词向量上算一遍

上一讲我们看过 TinyLM 的词向量表，每个词是一个 4 维向量。点积的规则在 4 维时完全一样，我们拿“我”和“推理”来算一遍：

$$\text{“我”} = (0.7, -0.6, -0.9, 0.3) \qquad \text{“推理”} = (0.7, 0.3, -0.8, -0.4)$$

第一步，对应分量相乘：$0.7\times0.7 = 0.49$，$(-0.6)\times0.3 = -0.18$，$(-0.9)\times(-0.8) = 0.72$，$0.3\times(-0.4) = -0.12$。

第二步，加起来：$0.49 - 0.18 + 0.72 - 0.12 = \mathbf{0.91}$。这就是点积。

第三步，算两个长度。|我| = √(0.49 + 0.36 + 0.81 + 0.09) = √1.75 ≈ 1.32，|推理| = √(0.49 + 0.09 + 0.64 + 0.16) = √1.38 ≈ 1.17。

第四步，除一下：0.91 ÷ (1.32 × 1.17) ≈ **0.59**。夹角大约 54°。

用同样的方法，可以把 6 个词两两之间的余弦相似度全部算出来，排成一张 6×6 的表：

COS 6×6 · 第 i 行第 j 列是词 i 和词 j 的余弦相似度

⇆ 对照

	我	爱	学	AI	推理	<eos>
我	1	0.44	0.02	-0.49	0.59	-0.58
爱	0.44	1	-0.79	-0.39	0.46	-0.31
学	0.02	-0.79	1	-0.22	-0.54	0.2
AI	-0.49	-0.39	-0.22	1	0.33	0.1
推理	0.59	0.46	-0.54	0.33	1	-0.48
<eos>	-0.58	-0.31	0.2	0.1	-0.48	1

图 2-3　TinyLM 词向量两两之间的余弦相似度，由 tinylm/tinylm.py 算出。高亮的一格是“我”和“推理”：0.59。

这张表有几个特点，值得花一分钟看一看。对角线上全是 1：每个词和自己的方向当然完全相同。表是对称的：“我”对“推理”和“推理”对“我”是同一个数，因为点积交换顺序结果不变。还有一些格子是负数，比如“我”和“AI”是 −0.49，说明这两个向量大致反向。

不过要提醒一句：**这些数字目前没有语义上的意义**。TinyLM 的词向量是随机生成的，还没有经过训练，“我”和“推理”余弦相似度高只是巧合。经过训练的模型里，意思相近的词，余弦相似度会明显更高，这是第 22 讲的内容。这一讲只需要会算。

大模型为什么偏爱点积

讲到这里，你可能会问：既然余弦相似度更“干净”，大模型为什么还到处用点积？

第一个原因是**便宜**。一个 d 维的点积，只要 d 次乘法和 d − 1 次加法，大约 2d 次浮点运算（FLOPs）。4096 维的向量算一次点积，大约 8192 次运算。没有开方，也没有除法。

第二个原因是**适合成批计算**。下一讲（第 3 讲）会看到，矩阵乘法本质上就是一大张点积表：把一批向量和另一批向量两两做点积。GPU 上专门有一种硬件单元来做这件事，速度极快。能写成点积的计算，就能交给这种硬件。

第三个原因是**长度有时候也是信息**。点积对长度敏感，在有些场景里这反而是我们想要的。第 26 讲的注意力机制，就是用点积给“每个词该关注哪些词”打分；向量的长度也会影响打分的高低，模型会在训练中学会利用这一点。

从 Infra 的角度看，点积的计算量会以一种很有意思的方式增长。注意力机制里，一句话中的**每个词都要和每个词**做一次点积。假设一段输入有 s = 1000 个词，就要算 1000 × 1000 = 100 万个点积，而且模型的每一层、每一个注意力头都要算一遍。按大纲里 Llama-3-8B 的配置（32 层、每层 32 个注意力头、每个头 128 维，写正文前请对照官方配置核实）粗略算：

$$1000^2 \times 32\text{ 个头} \times 2 \times 128 \approx 8.2 \times 10^9\text{ FLOPs（每层）} \quad \times 32\text{ 层} \approx 2.6 \times 10^{11}\text{ FLOPs}$$

这还只是“打分”这一步。更值得注意的是它的增长方式：输入长度 s 变成原来的 10 倍，点积的个数会变成原来的 **100 倍**，因为它和 s^2 成正比。为什么长文本推理又慢又贵，这里就埋下了第一个伏笔。（第 27 讲会讲到，生成文字时每个词只需要看前面的词，实际计算量大约能减半，但仍然和 s^2 成正比。）

用代码试一试

在 `tinylm/` 目录先运行一次 `python tinylm.py`，再运行下面的代码。NumPy 里，`a @ b` 对两个一维数组就是点积（写成 `np.dot(a, b)` 也一样）：

```python
import json
import numpy as np

short_gpu = np.array([4, 1])     # 讲 GPU 的短文：“GPU”4 次，“美食”1 次
long_gpu = np.array([12, 3])     # 同一主题的长文：每个词的次数都是短文的 3 倍
food = np.array([1, 5])          # 讲美食的文章

def cos_sim(a, b):
    return (a @ b) / (np.linalg.norm(a) * np.linalg.norm(b))

print(np.linalg.norm(short_gpu - long_gpu), np.linalg.norm(short_gpu - food))  # 距离
print(short_gpu @ long_gpu, short_gpu @ food)                                  # 点积
print(cos_sim(short_gpu, long_gpu), cos_sim(short_gpu, food))                  # 余弦相似度

tiny = json.load(open("tinylm.json", encoding="utf-8"))
vocab = tiny["config"]["vocab"]
E = np.array(tiny["weights"]["embedding"])
wo, tl = E[vocab.index("我")], E[vocab.index("推理")]
print(wo * tl)                   # 对应分量相乘
print(wo @ tl)                   # 再加起来：点积
print(cos_sim(wo, tl), np.degrees(np.arccos(cos_sim(wo, tl))))
```

输出：

```
8.246211251235321 5.0
51 9
1.0 0.4280863447390447
[ 0.49 -0.18  0.72 -0.12]
0.9099999999999999
0.585575401399886 54.156350110779144
```

逐行对照：距离 8.25 和 5，按距离量短文更像美食文章；点积 51 和 9；余弦相似度 1.0 和 0.43。后三行是“我”和“推理”：先得到对应分量的乘积 (0.49, −0.18, 0.72, −0.12)，加起来是点积 0.91，余弦相似度 0.59，夹角约 54°，和手算一致。

你可能注意到了，点积打印出来是 0.9099999999999999 而不是 0.91。这不是算错了，而是计算机存小数时的精度限制：0.49、0.72 这样的小数在二进制里存不准，加起来就差了一点点。这个现象在大模型里非常重要，第 10 讲会专门讲。

容易混淆的地方

1. **点积和“对应分量相乘”不是一回事**。在 NumPy 里，`a * b` 只做了第一步，得到的还是一个向量 (0.49, −0.18, 0.72, −0.12)；`a @ b` 或 `np.dot(a, b)` 才会再加起来，得到一个数 0.91。
2. **余弦相似度为 0 不代表“相反”**。0 表示互相垂直，可以理解为“没关系”；方向相反是 −1。
3. **余弦相似度高不等于两个向量相同**。(4, 1) 和 (12, 3) 的余弦相似度是 1，但它们并不相等，只是方向相同。如果长度本身也有意义，就不能只看余弦相似度。

回顾

这一讲从一个反例出发：用距离量文章的相似程度，会把“篇幅不同”误当成“内容不同”。真正有用的是方向，而点积能用乘法和加法把方向的信息算出来：$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}|\,|\mathbf{b}| \cos\theta$。点积的正负告诉我们两个向量同向、垂直还是反向；再除以两个长度，就得到只看方向的余弦相似度。大模型偏爱点积，因为它便宜，又能成批地交给 GPU 计算；注意力机制里每个词都要和每个词做点积，这也让计算量随输入长度的平方增长。

这一讲每次只算一个点积。可是在注意力机制里，一次要算成千上万个：一批向量和另一批向量两两之间都要算。有没有一种写法，能把“一批向量对一批向量”的所有点积一次表达出来，一次交给硬件去算？这就是下一讲的内容：**矩阵与矩阵乘法**。

第03讲 矩阵乘法

01 向量　02 点积与余弦相似度

上一讲结尾留了一个问题：注意力机制里，一次要算成千上万个点积，而且是“一批向量”和“另一批向量”两两之间都要算。一个一个地写、一个一个地算，显然不是办法。

这一讲要介绍的矩阵乘法，就是为这件事准备的写法。它能把“一批向量对一批向量”的所有点积，用一个式子写出来，再一次交给硬件算完。你在大学里肯定学过矩阵乘法，但当时可能只记住了“行乘列”的口诀。这一讲换一个角度：先从具体的点积出发，看矩阵乘法是怎么“长”出来的，再看它在大模型里扮演什么角色。

16 个点积，排成一张表

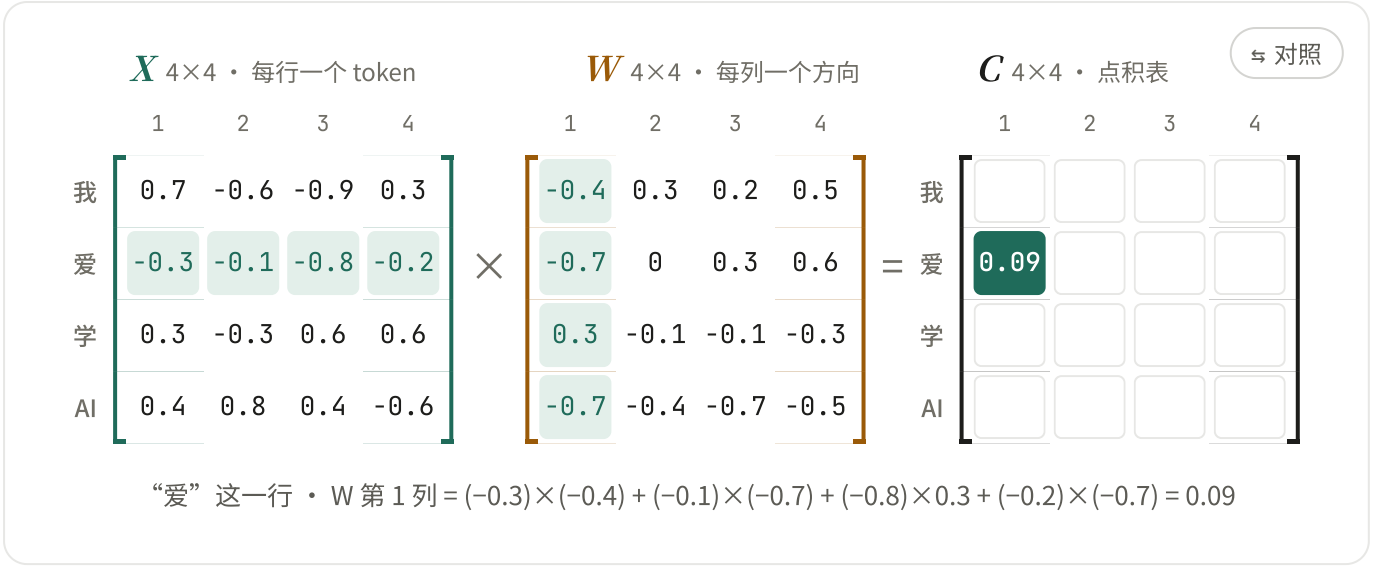
还是用全站的玩具模型 TinyLM。第 1 讲说过，“我 爱 学 AI”这句话里的 4 个词，各自对应一个 4 维向量：

词	向量
我	(0.7, −0.6, −0.9, 0.3)
爱	(−0.3, −0.1, −0.8, −0.2)
学	(0.3, −0.3, 0.6, 0.6)
AI	(0.4, 0.8, 0.4, −0.6)

现在假设模型的某一层要对每个词做这样一件事：准备好 4 个固定的“探测方向”，每个方向也是一个 4 维向量，然后让每个词分别和这 4 个方向做点积。上一讲说过，点积量的是“两个向量有多一致”，所以每个点积都可以理解成“这个词在这个方向上得几分”。

4 个词，每个词对 4 个方向，一共要算 $4 \times 4 = 16$ 个点积。16 个数，最自然的摆法是一张表：**每个词占一行，每个方向占一列**，第 i 行第 j 列就填“第 i 个词和第 j 个方向的点积”。

为了让这张表好算，我们把参与计算的向量也摆成表。4 个词的向量一行一个，叠成一张 4×4 的表，叫它 X；4 个方向一列一个，并排成另一张 4×4 的表，叫它 W。下图把三张表画在一起：



图里高亮的是一个例子：“爱”这一行（X 的第 2 行）和 W 的第 1 列做点积，结果填在右边那张表的“爱”这一行、第 1 列。四对数字分别相乘再相加：

$$(-0.3)\times(-0.4)+(-0.1)\times(-0.7)+(-0.8)\times 0.3+(-0.2)\times(-0.7)=0.12+0.07-($$

其余 15 格也一样，每一格都是“左边一行”和“右边一列”的点积。

这张表就是矩阵乘法

上面这三张数字表，在数学里就叫**矩阵**：m 行 n 列的数排成一个长方形，就是一个 m×n 的矩阵。X 是 4×4，W 也是 4×4。

而“用 X 的每一行去和 W 的每一列做点积，把结果填成一张新表”这个操作，就是**矩阵乘法**，写作 C = XW（或 X · W，代码里写 `X @ W`）。所谓“行乘列”的口诀，说的就是这件事：

> **要点**
> 矩阵乘法 C = XW 的每一格，都是 X 的一行和 W 的一列的点积。整个 C 就是一张点积表。

写成一般的公式：设 A 是 m×k 的矩阵，B 是 k×n 的矩阵，它们的乘积 C = AB 是 m×n 的矩阵，第 i 行第 j 列为

$$C_{ij}=\sum_{t=1}^{k}A_{it}\cdot B_{tj}=A_{i1}B_{1j}+A_{i2}B_{2j}+\cdots+A_{ik}B_{kj}$$

式子里各个符号的意思：

m	A 的行数：有几个输入向量。这里是 4 个词
k	每个输入向量的维度，也就是 A 的列数；它必须等于 B 的行数
n	B 的列数：有几个探测方向，也就是结果的列数
i, j	C 的第 i 行、第 j 列
t	求和下标，从 1 走到 k，对应点积里“第 t 个分量相乘”

对照上一讲的点积定义 $a_1b_1 + a_2b_2 + \cdots + a_kb_k$，就能看出这个式子并没有新东西：A 的第 i 行是一个 k 维向量，B 的第 j 列也是一个 k 维向量，C_{ij} 就是它们的点积。

一格一格算出来

在看演示之前，先用手算第一格：“我”这一行和 W 的第 1 列。

$$0.7 \times (-0.4) + (-0.6) \times (-0.7) + (-0.9) \times 0.3 + 0.3 \times (-0.7) = -0.28 + 0.42 - 0.27 -$$

再用下面的演示核对，并把剩下的 15 格也算出来。点“下一格”一格一格往前走，点“▶ 播放”让它自动走，也可以直接点结果 C 里的任意一格，跳到那一格。每走一步，参与计算的那一行、那一列会被高亮，下方的提示条写出这一格的完整算式。

X × W，一格一格地算

⇆ 对照

X 4×4 · 我 爱 学 AI

	1	2	3	4
我	0.7	-0.6	-0.9	0.3
爱	-0.3	-0.1	-0.8	-0.2
学	0.3	-0.3	0.6	0.6
AI	0.4	0.8	0.4	-0.6

×

W 4×4 · 权重

	1	2	3	4
	-0.4	0.3	0.2	0.5
	-0.7	0	0.3	0.6
	0.3	-0.1	-0.1	-0.3
	-0.7	-0.4	-0.7	-0.5

=

C 4×4 · 结果

	1	2	3	4
我				
爱				
学				
AI				

▶ 播放　下一格　全部算完　重置　0 / 16

点“播放”一格一格地算，或直接点结果 C 里的任意一格。

走完 16 格，可以停下来数一数一共做了多少次运算：每一格是一次 4 维点积，要做 4 次乘法、把结果累加起来；16 格就是 16 × 4 = **64 次乘加**。这个数字在后面算计算量时会用到。

X 和 W 里的数字都可以直接改。试着把 X 第一行的 0.7 改成 0，你会发现只有“我”这一行的结果变了，其他三行纹丝不动。这是一个很重要的性质，后面马上会用到。

专栏　这些数字从哪来

X 和 W 都来自课程目录下的 `tinylm/tinylm.py`（随机种子 2026，权重取一位小数）。X 是 4 个词的词向量，第 22 讲会讲它们是怎么来的；W 是 TinyLM 第 1 层里的一个权重矩阵，到第 26 讲，它会以“W_Q”的名字出现，这里算出的 C 就是注意力里的 Q。

形状要对得上

矩阵乘法有一个硬性要求：**左边矩阵的列数，必须等于右边矩阵的行数**。

这个要求不是凭空规定的。回想一下，C 的每一格是“左边一行”和“右边一列”的点积，而上一讲说过，点积要求两个向量维度相同。左边一行有几个数，由左边矩阵的列数决定；右边一列有几个数，由右边矩阵的行数决定。两者不相等，就凑不成点积。

结果的形状也可以这样推出来：左边有几行，就有几个输入向量，结果就有几行；右边有几列，就有几个方向，结果就有几列。把这些写成一条规则：

$$[m,\ k] \mathbin{@} [k,\ n] \to [m,\ n]$$

中间的 k 相消，两边的 m 和 n 留下。比如只要 W 的前两列，也就是只保留 2 个探测方向，那么 [4, 4] @ [4, 2] → [4, 2]，每个词只得到两个分数，正好是原来 C 的前两列：(−0.34, 0.18)、(0.09, 0.07)、(−0.15, −0.21)、(−0.18, 0.32)。

这条规则在大模型里每天都在用。模型里的一批 token 通常写成形状 [s, d] 的矩阵：s 个 token，每个 d 维。经过一个 $d \times d_{out}$ 的权重矩阵，就变成

$$[s,\ d] \mathbin{@} [d,\ d_{\text{out}}] \to [s,\ d_{\text{out}}]$$

每个 token 从 d 维变成了 d_{out} 维。看到一个矩阵乘法，先把两边的形状写出来，检查中间的维度对不对得上，是读大模型代码时最有用的习惯之一。第 5 讲会专门讲形状。

换个角度：一次对一批向量做同一种变换

到目前为止，我们是一格一格地看 C 的。现在换一个角度，**一行一行地看**。

C 的第 1 行，是“我”这个词分别和 W 的 4 列做点积得到的 4 个数：(−0.34, 0.18, −0.16, 0.11)。也就是说，“我”原来是一个 4 维向量 (0.7, −0.6, −0.9, 0.3)，乘以 W 之后，变成了另一个 4 维向量 (−0.34, 0.18, −0.16, 0.11)。同样，“爱”变成了 (0.09, 0.07, 0.13, 0.13)，依此类推。

所以，矩阵乘法 XW 还可以这样理解：**W 是一台“变换机器”，X 的每一行都被送进这台机器，变成 C 里对应的一行**。所有的行用的是同一台机器 W，而且各行之间互不干扰——这就是刚才在演示里看到的现象：改了“我”那一行的数字，其他行的结果都不变。

这正是大纲给这一讲提出的问题——怎么一次对一批向量做同一种变换——的答案：**把这批向量一行一行叠成矩阵 X，乘以同一个 W**。一句话有 4 个词，就是 4 行；有 1000 个词，就是 1000 行，W 不需要任何改变。大模型里几乎每一层都在做这件事：注意力里的 Q、K、V 投影，前馈网络里的升维和降维，最后的输出层，都是“一批 token 乘以同一个权重矩阵”。

至于这台“变换机器”到底对向量做了什么——是拉长、旋转，还是别的什么——这是下一讲的内容。

用代码试一试

先在 `tinylm/` 目录运行一次 `python tinylm.py` 生成 `tinylm.json`，再在同一目录运行下面的代码。它用两种写法计算同一个 C：一种是照着定义写三重循环，另一种是一次矩阵乘法。

```python
import json
import numpy as np

ex = json.load(open("tinylm.json", encoding="utf-8"))["examples"]["03"]
X = np.array(ex["X"])   # 我 爱 学 AI，每个 token 一个 4 维向量
W = np.array(ex["W"])   # 一个 4×4 权重矩阵

# 写法一：三重循环，逐格做点积
m, k = X.shape
_, n = W.shape
C = np.zeros((m, n))
for i in range(m):
    for j in range(n):
        for t in range(k):
            C[i, j] += X[i, t] * W[t, j]

# 写法二：一次矩阵乘法
print(np.round(X @ W, 2))
print(np.allclose(C, X @ W))        # 两种写法一致
print(np.allclose(X @ W, ex["C"])) # 与手算一致
print(np.round(W @ X, 2)[0])        # 顺序反过来，结果就不一样了
```

输出：

```
[[-0.34  0.18 -0.16  0.11]
 [ 0.09  0.07  0.13  0.13]
 [-0.15 -0.21 -0.51 -0.51]
 [-0.18  0.32  0.7   0.86]]
True
True
[-0.11  0.55  0.44 -0.36]
```

第一行第一格是 −0.34，第二行第一格是 0.09，都和前面的手算一致。三重循环和 `X @ W` 的结果相同（这里用 `allclose` 而不是 `==` 比较，原因和上一讲见到的 0.9099999999999999 一样：浮点数的加法顺序不同，最后几位可能有极小的差别）。

两种写法的差别在速度。三重循环在 Python 里一格一格地算；`X @ W` 则把整个运算一次交给底层的数值库，在 GPU 上还会交给专门做矩阵乘法的硬件单元。矩阵一大，两者的速度能差几个数量级。

最后一行是 `W @ X` 的第一行：(−0.11, 0.55, 0.44, −0.36)，和 `X @ W` 的第一行完全不同。下一节会解释为什么。

放大到真实模型：要算多少、要搬多少

前面数过，4×4 乘 4×4 要做 64 次乘加。推广到一般情况：C 有 m × n 格，每格是一个 k 维点积，要做 k 次乘法和 k 次加法，所以一次矩阵乘法大约要做

$$2 \cdot m \cdot n \cdot k \text{ 次浮点运算（FLOPs）}$$

手算那一例是 2 × 4 × 4 × 4 = 128 FLOPs。这个公式是第 11 讲估算计算量的基本单位，后面会反复用到。

现在把 TinyLM 的 4 换成真实模型的 4096（大纲里用作例子的 Llama-3-8B 的隐藏维度，写正文时请对照官方配置核实），看一个 4096 × 4096 的权重矩阵。用下面的计算器试两组预设：“逐词生成”时一次只处理 1 个 token（s = 1），“读完长提示”时一次处理 1000 个 token（s = 1000）。



两组预设里，权重矩阵是同一个：4096 × 4096 个数，用 BF16 存（每个数 2 字节）要 32 MiB。不管一次处理多少个 token，这 32 MiB 都要从显存里读一遍。

区别在于读一遍之后能做多少计算。s = 1 时，只做 3.4×10^7 次运算，平均每读 1 字节只算 **1 次**；s = 1000 时，做 3.4×10^{10} 次运算，每读 1 字节算 **1000 次**。推一下就知道，这个比值正好约等于 s：FLOPs 是 2·s·k·n，读的字节数是 k·n·2，相除就剩下 s。

这个比值叫**算术强度**，是 AI Infra 里最重要的概念之一。GPU 算得很快，但从显存里搬数据相对慢。每读 1 字节只算 1 次，GPU 的大部分时间都在等数据；每读 1 字节算 1000 次，GPU 才能真正忙起来。大模型逐字生成文字时，恰恰是 s = 1 的情况——为什么生成文字比读提示慢得多，第 62 讲的 Roofline 模型和第 68 讲的 Prefill 与 Decode 会从这里展开。把精度换成 FP8，权重少搬一半，这是第 63、93 讲低精度能提速的原因之一。

容易混淆的地方

1. **XW 和 WX 一般不相等**。上面的代码已经看到了，即使两个矩阵都是 4×4，交换顺序后结果也完全不同。原因很直接：XW 是“X 的行”和“W 的列”做点积，WX 是“W 的行”和“X 的列”做点积，参与计算的根本不是同一批向量。形状不是方阵时，交换顺序连形状都可能对不上：[2, 3] @ [3, 2] 得到 2×2，[3, 2] @ [2, 3] 得到 3×3。
2. **`A * B` 不是矩阵乘法**。在 NumPy 和 PyTorch 里，`*` 是对应位置的元素相乘，两个 4×4 矩阵 `*` 出来还是 4×4，但每一格只是一次乘法，不是点积。矩阵乘法要用 `@`（或 `np.matmul`、`torch.matmul`）。
3. **“行乘列”，不是“列乘行”**。C_{ij} 用的是左边矩阵的第 i 行和右边矩阵的第 j 列。如果你发现自己需要“左边的列”，通常说明应该先把矩阵转置（行列互换），或者两个矩阵的顺序写反了。

回顾

这一讲从上一讲的点积出发：4 个词各自对 4 个方向做点积，得到 16 个数，排成一张表，这张表就是矩阵乘法的结果。矩阵乘法 C = XW 的每一格，都是 X 的一行和 W 的一列的点积；要乘得起来，左边的列数必须等于右边的行数，结果的形状是 [m, k] @ [k, n] → [m, n]。换个角度一行一行地看，矩阵乘法就是用同一个 W 一次变换一批向量，这正是大模型每一层都在做的事。从工程上看，一次矩阵乘法约 2mnk 次运算，而一次处理的 token 越多，读一遍权重能做的计算就越多。

还有一个问题没有回答：这台“变换机器”W，究竟把向量变成了什么样？如果把向量画成平面上的箭头，乘以一个矩阵之后，整个平面会被拉伸、旋转，还是扭曲？这就是下一讲的内容：**线性变换的几何意义**。

第04讲 线性变换

01 向量　03 矩阵乘法

上一讲最后，我们把矩阵乘法看成一台“变换机器”：一个向量从左边进去，乘以 W，从右边出来就变成了另一个向量。这一讲要回答的问题是：**这台机器到底对向量做了什么？**

只看一个向量，很难看出规律。所以这一讲换一种看法：不看单个向量，而是把整个平面——平面上所有的点——一起送进这台机器，看看整个平面变成了什么样子。为了能画出来，我们先在 2 维里看，也就是 2×2 的矩阵；最后再回到大模型的 4096 维。

先看两个特别的向量

在 2 维平面上，有两个最简单的向量：$\mathbf{e}_1 = (1, 0)$ 和 $\mathbf{e}_2 = (0, 1)$。它们分别指向横轴和纵轴的正方向，长度都是 1。

先看它们被矩阵 W 变成了什么。设

$$W = [[2, 1], [0, 1]]，即第 1 行是 (2, 1)，第 2 行是 (0, 1)$$

按上一讲的规则，$(1, 0) \cdot W$ 的第 1 个数是 $(1, 0)$ 和 W 第 1 列 $(2, 0)$ 的点积，等于 2；第 2 个数是 $(1, 0)$ 和 W 第 2 列 $(1, 1)$ 的点积，等于 1。所以 $(1, 0)$ 变成了 $(2, 1)$——恰好就是 **W 的第 1 行**。同样算一下，$(0, 1)$ 变成了 $(0, 1)$，恰好是 **W 的第 2 行**。

这不是巧合。$(1, 0)$ 只有第 1 个分量是 1，其余都是 0，所以它和每一列做点积时，只会“挑出”那一列的第 1 个数；把这些数排起来，就是 W 的第 1 行。

要点

W 的第 1 行，就是 (1, 0) 被变换后的位置；W 的第 2 行，就是 (0, 1) 被变换后的位置。

知道两个向量，就知道全部

这个结论比看上去重要得多。第 1 讲的专栏里提到过“线性组合”：平面上任何一个向量 (x_1, x_2)，都可以写成

$$(x_1, x_2) = x_1 \cdot (1, 0) + x_2 \cdot (0, 1)$$

也就是“沿横轴走 x_1 步，再沿纵轴走 x_2 步”。

矩阵乘法有一个关键的性质：先组合再变换，和先变换再组合，结果一样。所以

$$(x_1, x_2) \cdot W = x_1 \cdot （W\text{的第 1 行}） + x_2 \cdot （W\text{的第 2 行}）$$

换句话说，**变换后的向量，就是按原来的“走法”，改用 W 的两行当作新的两个方向去走**。原来是“横走 1 步、竖走 1 步”到达 (1, 1)；现在变成“沿 (2, 1) 走 1 步、沿 (0, 1) 走 1 步”，到达 (2, 1) + (0, 1) = (2, 2)。你可以用上一讲的“行乘列”验算一下：(1, 1) · W = (1×2 + 1×0, 1×1 + 1×1) = (2, 2)，结果一样。

这给了我们一个非常有用的视角：**只要知道 (1, 0) 和 (0, 1) 被送到了哪里，整个平面的变换就完全确定了**。而这两个位置，就写在 W 的两行里。

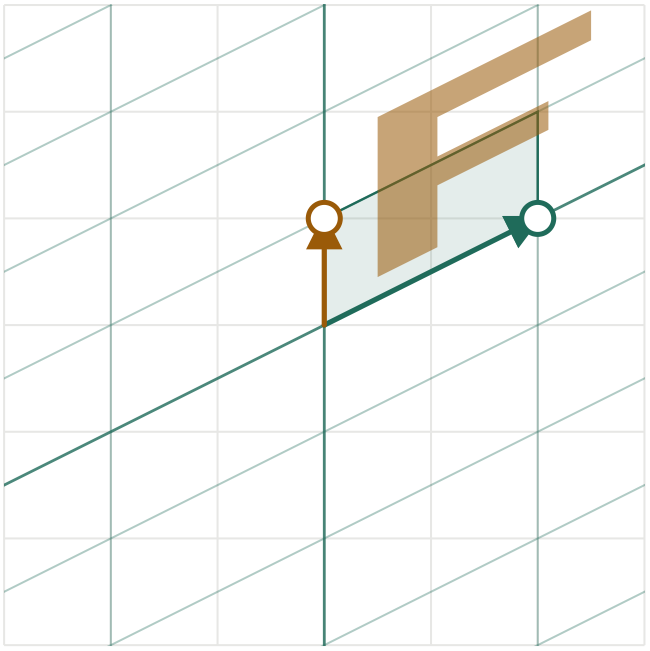
看整个平面怎么变

下面的演示把这个想法画了出来。浅灰色的方格是原来的平面；绿色的网格是变换后的平面；绿色箭头是 (1, 0) 的新位置，也就是 W 的第 1 行，琥珀色箭头是 (0, 1) 的新位置，也就是 W 的第 2 行。原来那个 1×1 的小正方形被涂上了浅绿色，图里还放了一个“F”字，用来看清平面有没有被翻转。

你可以改右边矩阵里的数字，也可以直接拖动两个箭头的末端（拖动箭头就是在改 W 的那一行）。点“▶ 播放变换”，平面会从原样一点点变成 W 作用后的样子；拖动 t 的滑块，可以停在任意一个中间状态。

改矩阵、拖箭头，看整个平面怎么变

⇆ 对照



W 2×2 · 可以直接改

	1	2
第 1 行	2	1
第 2 行	0	1

恒等 | 横向拉伸 | 旋转 30° | 剪切 | 镜像 | 压扁

(1, 0) →	**(2, 1)** 即第 1 行
(0, 1) →	**(0, 1)** 即第 2 行
(1, 1) →	1×第 1 行 + 1×第 2 行 = (2, 2)
行列式	2×1 − 1×0 = **2**

面积变成原来的 2 倍

▶ 播放变换　t = **1**

图 4–1　矩阵 W 对整个平面的作用。绿色、琥珀色箭头分别是 W 的第 1、2 行；浅绿色是单位正方形变换后的样子。点右侧的预设看几种典型的变换。

默认的 W 第 1 行是 (2, 1)、第 2 行是 (0, 1)。点“播放”看一遍，你会看到横向的格线被拉长、同时向上倾斜，纵向的格线保持竖直。那个小正方形变成了一个底边更长的平行四边形，面积变成了原来的 2 倍。

无论你怎么改 W，下面几件事始终成立：

- **原点不动。**(0, 0) 乘以任何矩阵都还是 (0, 0)。
- **直线还是直线。**格线可能被拉长、倾斜，但不会被掰弯。
- **平行线还是平行线，间距还是均匀的。**原来等距排列的格线，变换后依然等距排列。

满足这几条的变换，叫做**线性变换**。“线性”这个词说的就是这种“不弯曲、不打乱比例”的性质。反过来，每一个线性变换都可以写成一个矩阵，所以“线性变换”和“矩阵”在这门课里可以看成同一件事的两种说法：前者讲它对空间做了什么，后者是把它写下来、算出来的方式。

几种典型的变换

演示右侧有几个预设按钮。逐个点一遍，每点一个都会从原样播放到变换后的样子。下面一边点、一边对照：

横向拉伸：W = [[2, 0], [0, 1]]。(1, 0) 被拉到 (2, 0)，(0, 1) 不动。整个平面在横向被拉长了 2 倍，“F”变得又宽又扁。对角线以外都是 0 的矩阵，只会沿坐标轴拉伸或压缩。

旋转 30°：W = [[0.866, 0.5], [−0.5, 0.866]]。(1, 0) 被转到 (cos 30°, sin 30°) ≈ (0.866, 0.5)，(0, 1) 被转到 (−sin 30°, cos 30°) ≈ (−0.5, 0.866)。整个平面绕原点逆时针转了 30°，形状和大小都不变，面积也不变。

剪切：W = [[1, 0], [1, 1]]。(1, 0) 不动，(0, 1) 被推到了 (1, 1)。越往上的点被往右推得越多，就像把一叠扑克牌的上面几张往右一推。正方形变成了一个斜着的平行四边形，但面积不变。

镜像：W = [[−1, 0], [0, 1]]。(1, 0) 被翻到了 (−1, 0)。整个平面左右翻转，“F”变成了反写的样子。

压扁：W = [[1, 0.5], [2, 1]]。注意这两行：(2, 1) 正好是 (1, 0.5) 的 2 倍，两个箭头落在了同一条直线上。于是整个平面被压成了一条直线，小正方形也被压成了一条线段，面积变成 0。

最后这个例子值得多看一眼。平面被压成一条线以后，原来不同的点可能会落到同一个位置上。比如 (1, 0) 和 (−1, 1)，变换后都落在 (1, 0.5)。**一旦被压扁，就再也分不清它们原来是谁了**——信息丢了，这个变换也就没法“撤销”。

面积变了多少：行列式

演示右侧还有一行叫“行列式”。对 2×2 的矩阵 [[a, b], [c, d]]，它的计算方法是

$$\det W = a \cdot d - b \cdot c$$

它的几何意义正好是：**单位正方形变换后的面积**，也就是整个平面上所有面积被放大的倍数。默认的 W 是 2 × 1 − 1 × 0 = 2，面积变成 2 倍；旋转和剪切的行列式都是 1，面积不变；压扁的行列式是 1 × 1 − 0.5 × 2 = 0，面积变成 0。

行列式还可以是负数。镜像的行列式是 (−1) × 1 − 0 × 0 = −1：面积大小没变，但负号告诉我们平面被翻了过来——这就是“F”变成反写的原因。

行列式在大模型的计算里很少直接出现，但“行列式为 0 = 被压扁 = 信息丢失”这个直觉很有用。第 40 讲会讲到一种叫 MLA 的技术，它故意先把向量压到更低的维度存起来、需要时再还原，用的正是这种“压扁”，只不过是在模型能接受的范围内有控制地压。

一次变换接着一次变换

如果先用矩阵 A 变换一次，再用矩阵 B 变换一次，结果会怎样？按行向量的写法，x 先变成 x · A，再变成 (x · A) · B。矩阵乘法满足结合律，所以

$$(\mathbf{x} \cdot A) \cdot B = \mathbf{x} \cdot (A \cdot B)$$

连续做两次变换，等于用一个矩阵 A · B 做一次变换。这就是矩阵和矩阵相乘的几何意义：把两次变换合并成一次。

那么顺序重要吗？举个例子。设 S 是横向拉伸 2 倍：S = [[2, 0], [0, 1]]；R 是逆时针旋转 90°：(1, 0) 转到 (0, 1)，(0, 1) 转到 (−1, 0)，所以 R = [[0, 1], [−1, 0]]。跟踪 (1, 0) 这个向量：

- **先拉伸、再旋转**：(1, 0) 先被拉伸成 (2, 0)，再被旋转成 (0, 2)。
- **先旋转、再拉伸**：(1, 0) 先被旋转成 (0, 1)，再拉伸——但横向拉伸对纵轴上的点不起作用，所以还是 (0, 1)。

结果完全不同。对应到矩阵上，S · R = [[0, 2], [−1, 0]]，R · S = [[0, 1], [−2, 0]]，两个矩阵不相等。上一讲说“XW 和 WX 一般不相等”，这里就是它的几何解释：**先做哪个变换、后做哪个变换，结果不一样。**

专栏　为什么神经网络需要“激活函数”

“两次线性变换可以合成一次”这个性质，在神经网络里有一个意想不到的后果。

假设你把两层线性变换叠起来：$x \cdot W_1 \cdot W_2$。不管 W_1、W_2 有多大，它们乘起来还是一个矩阵 W_1W_2，效果和只用一层完全一样。叠十层、一百层也一样——只要每一层都是线性的，整个网络就只是一次线性变换，只会拉伸、旋转、剪切，永远掰不弯一条直线。

要让网络能表达更复杂的东西，就必须在两层线性变换之间插入一个“非线性”的步骤，把直线掰弯。这个步骤就是第 14 讲要讲的激活函数。

回到 4096 维

2 维的画面帮我们建立了直觉，现在回到大模型。前面的结论在任何维度下都成立：**W 的第 i 行，就是第 i 个坐标方向被变换后的位置；一个向量变换后的结果，就是把 W 的各行按这个向量的分量加权相加。**

拿上一讲的 TinyLM 验证一下。“我”的向量是 (0.7, −0.6, −0.9, 0.3)，W 是那个 4×4 的权重矩阵。上一讲用“行乘列”算出 (0.7, −0.6, −0.9, 0.3) · W = (−0.34, 0.18, −0.16, 0.11)。现在换成这一讲的看法：

$$0.7 \times（\text{W 第 1 行}）- 0.6 \times（\text{W 第 2 行}）- 0.9 \times（\text{W 第 3 行}）+ 0.3 \times（\text{W 第 4 行}）$$

只看第一个分量：W 四行的第一个数分别是 −0.4、−0.7、0.3、−0.7，所以是 0.7 × (−0.4) − 0.6 × (−0.7) − 0.9 × 0.3 + 0.3 × (−0.7) = −0.28 + 0.42 − 0.27 − 0.21 = −0.34，和上一讲一样。两种看法算的是同一件事，只是分组的方式不同：“行乘列”是一格一格地算，这一讲的看法是一行一行地加。

在真实的大模型里，W 往往是 4096 × 4096，甚至 4096 × 14336。我们画不出 4096 维的网格，但发生的事情是一样的：整个 4096 维空间被拉伸、旋转、剪切，有时还会被压到更低的维度（W 的列数比行数少时），有时被映射到更高的维度。模型在训练中要做的，就是找到合适的 W，让这些变换把“意思相近”的东西送到相近的位置。

从工程的角度看，一个线性变换就是一个矩阵，要存下它，就要存下矩阵里的每一个数：4096 × 4096 的矩阵约有 1680 万个数，用 BF16 存要 32 MiB（第 3 讲算过）。一个“80 亿参数”的模型，绝大部分参数就是一个个这样的矩阵。第 35 讲会一层一层地数清楚。

用代码试一试

在 `tinylm/` 目录先运行一次 `python tinylm.py`，再运行下面的代码，把这一讲的几个结论逐一验证一遍：

```
import json
import numpy as np

W = np.array([[2.0, 1.0],          # 第 1 行：(1, 0) 变换后去哪
              [0.0, 1.0]])         # 第 2 行：(0, 1) 变换后去哪
x = np.array([1.0, 1.0])
print(x @ W)                       # 1×第 1 行 + 1×第 2 行
print(np.linalg.det(W))            # 面积的缩放倍数

S = np.array([[2.0, 0.0], [0.0, 1.0]])    # 横向拉伸 2 倍
R = np.array([[0.0, 1.0], [-1.0, 0.0]])   # 逆时针旋转 90°
e1 = np.array([1.0, 0.0])
print((e1 @ S) @ R, e1 @ (S @ R))  # 先拉伸再旋转：分两步算和合成一个矩阵，结果一样
print(e1 @ (R @ S))                # 先旋转再拉伸：结果不同
print(S @ R)
print(R @ S)

flat = np.array([[1.0, 0.5], [2.0, 1.0]])  # “压扁”
print(np.linalg.det(flat))
print(np.array([1.0, 0.0]) @ flat, np.array([-1.0, 1.0]) @ flat)

ex = json.load(open("tinylm.json", encoding="utf-8"))["examples"]["03"]
X, W4 = np.array(ex["X"]), np.array(ex["W"])
me = X[0]                                  # “我”的向量
print(np.round(me @ W4, 2))
print(np.round(sum(me[i] * W4[i] for i in range(4)), 2))  # 用 W 的 4 行按“我”的分量组合
```

输出：

```
[2. 2.]
2.0
[0. 2.] [0. 2.]
[0. 1.]
[[ 0.  2.]
 [-1.  0.]]
[[ 0.  1.]
 [-2.  0.]]
0.0
[1.  0.5] [1.  0.5]
[-0.34  0.18 -0.16  0.11]
[-0.34  0.18 -0.16  0.11]
```

逐行对照：(1, 1) 变成 (2, 2)，行列式是 2；先拉伸再旋转，(1, 0) 到了 (0, 2)，分两步算和用 S · R 一步算结果相同；先旋转再拉伸则到了 (0, 1)；S · R 和 R · S 是两个不同的矩阵；“压扁”矩阵的行列式是 0，(1, 0) 和 (−1, 1) 被送到了同一个点 (1, 0.5)。最后两行是“我”的向量：用“行乘列”和用“W 的各行加权相加”，结果完全一样。

容易混淆的地方

1. **为什么这里是“行”，课本里是“列”？**大多数线性代数课本把向量写成竖着的一列，变换写成 W · x（矩阵在左边），那时“(1, 0) 的去向”是 W 的**第 1 列**。这门课跟着代码的习惯，把向量写成横着的一行，变换写成 x · W（矩阵在右边），于是去向变成了 W 的**第 1 行**。两种写法描述的是同一件事，差一个转置（行列互换）。读别的资料时，先确认它用的是哪一种。
2. **旋转矩阵“看起来反了”**。同一个原因：课本里逆时针旋转 θ 写成 [[cos θ, −sin θ], [sin θ, cos θ]]，这里写成 [[cos θ, sin θ], [−sin θ, cos θ]]，正好是课本的转置。
3. **平移不是线性变换**。把整个平面往右挪一格，原点就不在原点了，违反了“原点不动”这一条，所以没法用一个矩阵乘法表示。神经网络里的“加一个偏置向量”就是平移，所以一层网络实际写成 x · W + b，这在第 13 讲会讲到。

回顾

这一讲把矩阵乘法从“一张点积表”换成了另一种看法：一台对整个空间做变换的机器。W 的每一行，就是一个坐标方向被送去的位置；知道了这几个位置，整个空间怎么变就完全确定了。这样的变换叫线性变换：原点不动，直线还是直线，平行和等距都保持。拉伸、旋转、剪切、镜像、压扁，都是它的特例；行列式告诉我们面积变了多少倍，为 0 时意味着空间被压扁、信息丢失。连续两次变换可以合成一次，顺序却不能交换；也正因为再多的线性变换叠起来还是线性变换，神经网络才需要激活函数。

到目前为止，我们只用到了向量（一串数）和矩阵（一张表）。可在真实的大模型里，经常看到 [b, s, d] 这样三个数的形状：一次处理好几句话，每句话有好多个词，每个词是一个向量。这种“好几张表叠在一起”的东西是什么，形状怎么读、运算之后形状怎么变？这就是下一讲的内容：**张量与形状**。

第05讲 张量与形状

01 向量　03 矩阵乘法

读大模型的代码或论文，到处是这样的注释：`x: [b, s, d]`、`q: [b, h, s, d_h]`、`scores: [b, h, s, s]`。第 3 讲的矩阵是两个数描述的一张表：几行、几列。这里却有三个、四个数。它们是什么？一个运算之后，形状为什么会变成那样？

这一讲讲**张量**和它的**形状**。它不涉及新的数学，只是把“一串数”“一张表”推广到更多的维度，再约定几条运算时形状怎么变的规则。可这些规则是读懂大模型代码最常用的工具：看一个运算，先看形状，往往就明白了它在做什么。

从数到张量

先把熟悉的东西排一排：

- 一个数，比如 0.7，叫**标量**，形状是 `[]`（没有维度）。
- 一串数，比如第 1 讲“我”的向量 (0.7, −0.6, −0.9, 0.3)，是**向量**，形状 `[4]`：一个维度，长度 4。
- 一张表，比如第 3 讲 4 个词叠成的 X，是**矩阵**，形状 `[4, 4]`：两个维度，4 行 4 列。

再往上，就是把好几张表叠起来。比如一次处理 2 句话，每句 4 个词、每个词 4 维，就是 2 张 4 × 4 的表叠在一起，形状 `[2, 4, 4]`。这种任意维度的“数的阵列”，统称**张量**（tensor）。标量、向量、矩阵都是张量的特例：0 维、1 维、2 维。

形状就是每个维度的长度，按顺序写成一个列表。维度的个数叫**阶**或**维数**（这里的“维”和“4 维向量”的“维”意思不同：后者说的是向量有几个分量）。

[b, s, d] 读作什么

大模型里最常见的形状是 `[b, s, d]`（全站的符号约定）：

b	批大小（batch）：一次同时处理几个序列（几句话、几个请求）
s	序列长度（sequence）：每个序列有几个 token
d	隐藏维度：每个 token 用几维的向量表示

一个形状为 `[b, s, d]` 的张量，要用三个下标取出一个数：`x[i, j, t]` 是第 i 个序列、第 j 个 token 的向量的第 t 个分量。只给前两个下标，`x[i, j]`，取出的是一整个 d 维的向量；只给第一个，`x[i]`，取出的是第 i 句话的 `[s, d]` 矩阵。

用 TinyLM 举例：一批 2 句话，“我 爱 学 AI”和“AI 爱 学 推理”。先分词得到编号（第 21 讲），形状 `[2, 4]`：第一句是 [0, 1, 2, 3]，第二句是 [3, 1, 2, 4]。再查嵌入表（第 22 讲）把每个编号换成一个 4 维向量，就得到形状 `[2, 4, 4]` 的 x。x[1, 3] 是第 2 句第 4 个词“推理”的向量 (0.7, 0.3, −0.8, −0.4)。

运算之后形状怎么变

大模型里用到的张量运算，形状的变化可以归成几条规则。

矩阵乘法只看最后两维。第 3 讲的规则是 `[m, k] @ [k, n] → [m, n]`。对更高阶的张量，矩阵乘法作用在最后两维上，前面的维度都当作“一批”，一一对应地算：

$$[\cdots, m, k] \mathbin{@} [\cdots, k, n] \to [\cdots, m, n]$$

比如 x 是 `[2, 4, 4]`，W 是 `[4, 4]`：x @ W 得到 `[2, 4, 4]`，相当于两句话各自做一次第 3 讲的矩阵乘法。第一句算出的结果，和第 3 讲的 C 完全一样。

reshape：换一种看法。`reshape` 只改变形状，不改变数据和它们的顺序。比如把 `[2, 4, 4]` 改成 `[2, 4, 2, 2]`：每个 token 的 4 维向量被看成 2 组、每组 2 个数。多头注意力（第 28 讲）就是这样把 d 维拆成 h 个头、每头 d_h 维的（$d = h \cdot d_h$）。

transpose：交换两个维度。`[2, 4, 2, 2]` 交换第 1、2 维（从 0 数起）变成 `[2, 2, 4, 2]`，也就是 `[b, h, s, d_h]`：现在每个头的 4 个 token 排在一起了，每个头可以独立地算注意力。再做 q @ k^T（最后两维 `[s, d_h] @ [d_h, s]`），就得到每个头里两两的打分 `[b, h, s, s]` = `[2, 2, 4, 4]`。

逐元素运算：形状不变。两个形状相同的张量相加、相乘，或者对每个元素算激活函数，形状不变。

归约：沿哪一维，哪一维消失。对 `[2, 4, 4]` 沿最后一维求和（`sum(axis=-1)`），得到 `[2, 4]`：每个 token 的 4 个分量加成一个数；沿第 1 维（token）求平均，得到 `[2, 4]`：每句话的 4 个 token 平均成一个向量。softmax（第 6 讲）、归一化（第 30 讲）都要指定沿哪一维做。

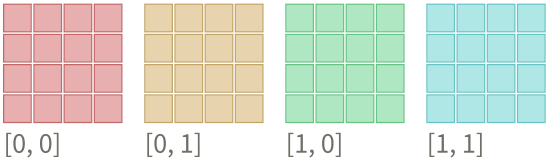
下面的演示把这些步骤串起来。点一步，看它的输入输出形状；切换到 Llama-3-8B 的尺寸，看同样的步骤在真实模型里有多大。

每一步之后，形状怎么变

⇆ 对照

TinyLM 尺寸 | Llama-3-8B 尺寸

1. `分词得到编号 ids`
   `[b=2, s=4]`
2. `x = E[ids]（查嵌入表）`
   `[b=2, s=4, d=4]`
3. `q = x @ W`
   `[b=2, s=4, d=4]`
4. `q.reshape(b, s, h, dh)`
   `[b=2, s=4, h=2, dh=2]`
5. `.transpose(1, 2)`
   `[b=2, h=2, s=4, dh=2]`
6. `scores = q @ kᵀ`
   `[b=2, h=2, s=4, s=4]`
7. `q + bias（广播）`
   `[b=2, s=4, d=4]`
8. `x.sum(axis=-1)`
   `[b=2, s=4]`



`scores = q @ kᵀ`：`[b=2, h=2, s=4, dh=2] → [b=2, h=2, s=4, s=4]`，共 64 个数。
最后两维做矩阵乘法 `[s, dh] @ [dh, s] → [s, s]`，前面的 b、h 是“批”维度，一一对应地算。每个头里 s × s 个打分。
右图：每个小方块是一个数，最后两维画成一张表，前面的维度画成并排的几张表，标着它们的下标。

图 5-1　形状追踪器。从编号 [b, s] 到嵌入 [b, s, d]，经矩阵乘法、拆头、转置，得到注意力打分 [b, h, s, s]。右边把 TinyLM 尺寸下的输出画成一格一格的数；切换到 Llama-3-8B，看每一步的数据量。

广播：形状不同也能相加

第 13 讲的线性层要给 `[s, d]` 的结果加上一个偏置 **b**，形状 `[d]`，两者形状不同。约定的做法是**广播**（broadcasting）：把小的那个自动复制到每一个位置上，再逐元素相加。

规则是从**最后一维**开始对齐比较：

1. 两个张量的最后一维要么相等，要么其中一个是 1；
2. 再看倒数第二维，依此类推；
3. 维数少的那个，前面缺的维度当作 1；
4. 长度为 1 的维度，被复制到和另一个一样长。

所以 `[2, 4, 4] + [4]`：最后一维 4 和 4 相等；`[4]` 前面缺的两维当作 1，复制成 `[2, 4, 4]`。效果就是同一个 bias 加到每句话的每个 token 上。代码验证了 q[1, 2] + b 和 (q + b)[1, 2] 完全一样。

广播在大模型里到处都是：注意力的掩码 `[s, s]` 加到打分 `[b, h, s, s]` 上（第 27 讲），归一化的缩放参数 `[d]` 乘到 `[b, s, d]` 上（第 30 讲）。它省去了显式的复制，也不占额外的显存。

> **要点**
> 张量是任意维度的数的阵列，形状是每一维的长度。大模型最常见的是 [b, s, d]：b 个序列、每个 s 个 token、每个 d 维。矩阵乘法作用在最后两维，前面的维度一一对应；reshape 换看法不动数据，transpose 交换维度；逐元素运算形状不变，沿哪一维归约哪一维就消失；形状不同时按广播规则从最后一维对齐。

形状决定了大小

张量占多少内存，就是形状里所有数的乘积，再乘以每个数的字节数（第 10 讲）。换成 Llama-3-8B 的尺寸（一批 8 个请求、每个 2048 个 token、d = 4096、32 个头）：

- 一层的输入 `[8, 2048, 4096]`：6700 万个数，BF16 下 **134 MB**。
- 注意力的打分 `[8, 32, 2048, 2048]`：BF16 下 **2.1 GB**，只是一层的一个中间结果。

后者里出现了 s × s，序列越长，增长得越快。这是第 84 讲 FlashAttention 要解决的问题。习惯了看形状，这类问题一眼就能发现。

用代码复现

下面的代码用 NumPy 和 TinyLM 的数据，把演示里的步骤走一遍，打出每一步的形状；最后算 Llama-3-8B 尺寸下两个张量的大小。（先在 `tinylm/` 目录运行一次 `python tinylm.py`。）

```
import json
import numpy as np

tl = json.load(open("tinylm.json", encoding="utf-8"))
vocab, E = tl["config"]["vocab"], np.array(tl["weights"]["embedding"])      # 词表 6 个词，嵌入表 [6, 4]
W = np.array(tl["examples"]["03"]["W"])                                     # 第 3 讲那个 4×4 的权重矩阵

# 一批 2 句话，每句 4 个词：先变成编号，形状 [b, s] = [2, 4]
sents = [["我", "爱", "学", "AI"], ["AI", "爱", "学", "推理"]]
ids = np.array([[vocab.index(w) for w in s] for s in sents])
print("编号 ids：", ids.shape, ids.tolist())

x = E[ids]                                                  # 查表：每个编号换成一个 4 维向量 → [b, s, d]
print("查嵌入表 x：", x.shape, "，x[1, 3] 是第 2 句第 4 个词“推理”的向量", x[1, 3].tolist())

q = x @ W                                                   # 最后一维和 W 相乘：[2, 4, 4] @ [4, 4] → [2, 4, 4]
print("乘权重 q = x @ W：", q.shape, "，和第 3 讲的 C 一样：", np.allclose(q[0], np.array(tl["examples"]["03"]["C"]), atol=0.006))

h, dh = 2, 2                                                # 拆成 2 个头，每头 2 维
qh = q.reshape(2, 4, h, dh).transpose(0, 2, 1, 3) # [b, s, d] → [b, s, h, dh] → [b, h, s, dh]
print("拆成多头：", q.shape, "→", q.reshape(2, 4, h, dh).shape, "→ 交换第 1、2 维", qh.shape)

scores = qh @ qh.transpose(0, 1, 3, 2)                      # [b, h, s, dh] @ [b, h, dh, s] → [b, h, s, s]
print("每个头里两两打分：", scores.shape)

b = np.array([0.1, 0.2, 0.3, 0.4])                          # 广播：[2, 4, 4] + [4]，b 被复制到每句话的每个词上
print("加偏置（广播）：", (q + b).shape, "，q[1, 2] + b 和 (q + b)[1, 2] 一样：", np.allclose(q[1, 2] + b, (q + b)[1, 2]))
print("对最后一维求和：", q.sum(axis=-1).shape, "；对第 1 维（词）求平均：", q.mean(axis=1).shape)

# 换成 Llama-3-8B 的尺寸：一批 8 个请求、每个 2048 个 token、d = 4096、32 个头
B, S, D, H = 8, 2048, 4096, 32
print(f"Llama-3-8B：一层的输入 [b, s, d] = [{B}, {S}, {D}]，{B * S * D / 1e6:.0f}M 个数，BF16 {B * S * D * 2 / 1e6:.0f} MB；"
```

```
      f"注意力打分 [b, h, s, s] = [{B}, {H}, {S}, {S}], {B * H * S * S * 2 /
1e9:.1f} GB")
```

输出：

```
编号 ids: (2, 4) [[0, 1, 2, 3], [3, 1, 2, 4]]
查嵌入表 x: (2, 4, 4)，x[1, 3] 是第 2 句第 4 个词"推理"的向量 [0.7, 0.3, -0.8, -0.4]
乘权重 q = x @ W: (2, 4, 4)，和第 3 讲的 C 一样: True
拆成多头: (2, 4, 4) → (2, 4, 2, 2) → 交换第 1、2 维 (2, 2, 4, 2)
每个头里两两打分: (2, 2, 4, 4)
加偏置（广播）: (2, 4, 4)，q[1, 2] + b 和 (q + b)[1, 2] 一样: True
对最后一维求和: (2, 4)；对第 1 维（词）求平均: (2, 4)
Llama-3-8B: 一层的输入 [b, s, d] = [8, 2048, 4096]，67M 个数，BF16 134 MB；注意力打分
[b, h, s, s] = [8, 32, 2048, 2048]，2.1 GB
```

每一行对应演示里的一步：编号 `(2, 4)`、嵌入 `(2, 4, 4)`、乘权重后第一句和第 3 讲的 C 一致、拆头 `(2, 2, 4, 2)`、打分 `(2, 2, 4, 4)`、广播加偏置、两种归约。最后一行是 Llama-3-8B 的大小。

容易混淆的地方

1. **“4 维向量”和“4 维张量”不是一回事。**前者是有 4 个分量的一串数，形状 `[4]`；后者有 4 个维度，比如 `[2, 2, 4, 2]`。说的时候最好直接说形状。
2. **reshape 和 transpose 不能互换。**`[s, h, d_h]` reshape 成 `[h, s, d_h]` 只是把同一串数重新切分，数据会错位；要让每个头的 token 排在一起，必须 transpose。
3. **广播是从右往左对齐的。**`[4, 4] + [4]` 是给每一行加，不是给每一列加；要给每一列加，需要把它写成 `[4, 1]`。

回顾

张量是任意维度的数的阵列：标量、向量、矩阵是 0、1、2 维的特例。形状是每一维的长度，`[b, s, d]` 表示 b 个序列、每个 s 个 token、每个 token d 维，`x[i, j]` 是第 i 个序列第 j 个 token 的向量。矩阵乘法作用在最后两维，前面的维度当作批；reshape 只换看法，transpose 交换维度；逐元素运算形状不变，归约让那一维消失；形状不同时按广播规则从最后一维对齐。TinyLM 一批 2 句经过嵌入、乘权重、拆头、打分，形状从 `[2, 4]` 变到 `[2, 2, 4, 4]`。张量的大小等于形状的乘积乘以每个数的字节数，Llama-3-8B 的一层输入就有 134 MB。

形状为 `[b, h, s, s]` 的注意力打分，每个数都是一个“分数”，有正有负、可大可小。模型最后要的却是“概率”：都在 0 到 1 之间，加起来等于 1。怎么把任意一组分数变成概率？下一讲：**指数、对数与 softmax**。

第06讲 指数、对数与 softmax

01 向量　05 张量与形状

大模型每生成一个词，最后一步都是一样的：给词表里的每个词打一个分（第 33 讲），分数越高，说明模型越觉得这个词该出现。可这些分数有正有负、可大可小，并不是概率。要从中抽一个词（下一讲），需要的是一组**概率**：每个都在 0 到 1 之间，加起来等于 1。

把任意一组分数变成概率的标准做法叫 **softmax**。它也出现在注意力里（第 26 讲）：把一组“相关程度”的打分变成加起来等于 1 的权重。这一讲先复习它要用到的指数和对数，再一步步算一次 softmax，最后看两个实用的问题：温度，以及数值的溢出。

指数：把任何数变成正数

指数函数 e^x（$e \approx 2.718$）有几个性质，正好是我们需要的：

- **永远是正数**：不管 x 是正是负，$e^x > 0$。$e^0 = 1$，$e^{-2} \approx 0.135$，$e^2 \approx 7.39$。
- **单调递增**：x 越大，e^x 越大，所以不会打乱分数的大小顺序。
- **增长很快**：x 每加 1，e^x 就乘以约 2.718。分数差 1，指数差 2.7 倍；差 3，就差 20 倍。
- **加法变乘法**：$e^{a+b} = e^a \cdot e^b$。

第一条解决了“分数有负数”的问题，第二条保证高分的词概率也高，第三条决定了 softmax 有多“偏心”。

对数：把乘法变回加法

对数 $\ln x$（以 e 为底）是指数的反函数：$\ln(e^x) = x$。它只对正数有定义。对我们最有用的性质是

$$\ln(a \cdot b) = \ln a + \ln b$$

乘法变加法。一句话的概率是每个词的概率相乘（第 23 讲），4 个词分别是 0.4666、0.2、0.1、0.05，相乘得 0.000467；取对数相加得 −7.67，再取指数又回到 0.000467。概率都小于 1，对数都是负数，越不可能的事，对数越负。

为什么要绕这一圈？因为很多个小于 1 的数相乘，结果会小到计算机表示不了。1000 个 0.01 相乘，真实结果是 10^{-2000}，可浮点数（第 10 讲）最小只能表示到 10^{-308} 左右，直接算出来是 0；取对数相加，是 −4605.2，毫无问题。所以模型训练时算的都是**对数概率**，交叉熵损失（第 16 讲）就是负的对数概率。

softmax：一步一步算

softmax 只有两步：**先取指数，再除以总和**。对一组分数 $z_1, \cdots, z_n$：

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

取指数让每个数变成正数，除以总和让它们加起来等于 1。

用 TinyLM 算一次。读到“我”之后，TinyLM 给 6 个词打的分是：我 1.64、爱 0.83、学 −0.39、AI −0.17、推理 −0.46、`<eos>` 0.37（TinyLM 的权重是随机的，这些分数没有意义，只用来演示计算）。

1. 取指数：5.155、2.293、0.677、0.844、0.631、1.448，总和 11.048。
2. 除以总和：**0.4666**、0.2076、0.0613、0.0764、0.0571、0.131。

六个数加起来正好是 1。“我”的分数最高，概率也最高，接近一半；分数最低的“推理”只有 5.7%。注意分数 1.64 和 0.83 只差 0.81，概率却差了一倍多，这就是指数“放大差距”的效果。

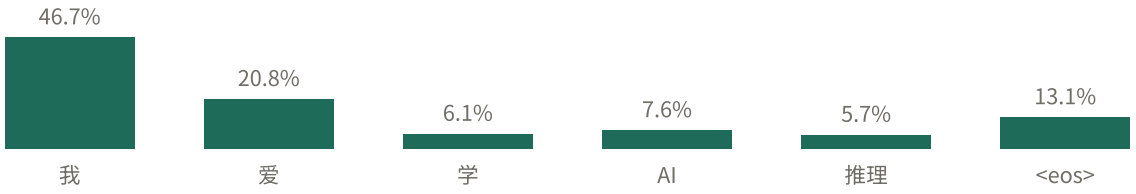
下面的演示可以改每个词的分数、调温度，看每一步的中间结果。演示里在取指数之前先减去了最大值（原因见后面“两个实用的性质”），所以中间的指数和总和与上面不同，最后的概率完全一样。

⇆ 对照

把分数变成概率

温度 T = **1.00**

词	我	爱	学	AI	推理	<eos>
打分 z（可改）	1.64	0.83	-0.39	-0.17	-0.46	0.37
z ÷ T − max	0.00	-0.81	-2.03	-1.81	-2.10	-1.27
e^(⋯)	1.000	0.445	0.131	0.164	0.122	0.281
÷ 总和 = 概率	0.467	0.208	0.061	0.076	0.057	0.131



总和 Σe = 2.143；最大的概率 46.7%（“我”）；分布的熵 1.48（最大可能是 ln 6 = 1.79，越小越“尖”）。
温度为 1：就是原始的 softmax。减去最大值不改变结果，只是防止 e^z 溢出。

图 6-1　softmax 的步骤。打分除以温度、减去最大值、取指数、除以总和，得到概率。温度小于 1 时分布变尖，大于 1 时变平；所有打分同时加一个数，概率不变。

要点

softmax 把任意一组分数变成概率：先对每个分数取指数（都变成正数、保持大小顺序），再除以它们的总和（加起来等于 1）。指数会放大分数之间的差距：分数差 1，概率差约 2.7 倍。

温度：控制分布有多尖

在取指数之前，先把所有分数除以一个数 T，叫**温度**：

$$p_i = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

- T = 1：原始的 softmax。

- **T < 1**：分数之间的差距被放大，分布变**尖**。T = 0.5 时，“我”的概率从 0.467 升到 0.749。T 趋近 0 时，最高分的词概率趋近 1，相当于只选最高分。
- **T > 1**：差距被缩小，分布变**平**。T = 2 时，“我”降到 0.302，最低的“推理”升到 0.106。T 很大时，所有词的概率趋于相等。

生成文字时调的“温度”参数就是它（第 73 讲）：温度低，回答稳定、保守；温度高，回答更多样、也更容易出错。

两个实用的性质

所有分数加同一个数，结果不变。把每个 z_i 都加上 c，分子分母都多乘了一个 e^c，约掉了。代码验证了：所有打分加 100，概率完全不变。所以 softmax 只关心分数之间的**差**，不关心绝对的大小。

用它来防止溢出。如果分数很大，比如 1000，直接算 e^{1000}，超出了浮点数能表示的范围，得到无穷大，再相除就出错了。利用上面的性质，先把每个分数都减去**最大的那个**，最大的变成 0、其余都是负数，指数都在 0 到 1 之间，绝不会溢出，而结果不变。softmax([1000, 999]) 这样算出来是 [0.7311, 0.2689]。所有的深度学习框架都是这样实现 softmax 的。

到了大模型里

输出层：词表上的 softmax。大模型最后对整个词表（Llama-3 有 128256 个词）做一次 softmax，得到下一个词的概率（第 33 讲）。

注意力：每一行一次 softmax。注意力的打分 `[b, h, s, s]`（第 5 讲）沿最后一维做 softmax，每个 token 对所有 token 的权重加起来等于 1（第 26 讲）。第 84 讲的 FlashAttention 要在“看不全一整行”的情况下算 softmax，靠的正是“减去最大值”这个性质的推广。

训练时用对数概率。框架通常把 softmax 和取对数合在一起算（log-softmax），既快又数值稳定，交叉熵损失直接用它。

用代码复现

下面的代码读取 TinyLM 读到“我”之后的打分，一步步算出 softmax，并和 `tinylm.json` 里存的概率对比；然后看温度的影响、平移不变性和防溢出；最后演示对数把乘法变成加法、避免下溢。

```
import json
import numpy as np

ex = json.load(open("tinylm.json", encoding="utf-8"))["examples"]["16"]
vocab, z = ex["vocab"], np.array(ex["logits"][0])        # TinyLM 读到“我”之后，给 6 个词打的分
print("打分 z：", dict(zip(vocab, z.tolist())))

e = np.exp(z)                                            # 第一步：取指数，全部变成正数
print("e^z：", np.round(e, 3).tolist(), "，和 =", round(e.sum(), 3))
p = e / e.sum()                                          # 第二步：除以总和，加起来等于 1
print("softmax：", np.round(p, 4).tolist(), "，和 =", round(p.sum(), 6))
print("和 tinylm.json 里存的概率一致：", np.allclose(p, ex["p"][0], atol=1e-4))

def softmax(z, T=1.0):
    z = np.asarray(z, dtype=float) / T
    e = np.exp(z - z.max())                              # 先减去最大值，防止溢出，结果不变
    return e / e.sum()

for T in [0.5, 1.0, 2.0]:
    print(f"温度 T = {T}：", np.round(softmax(z, T), 3).tolist())

print("所有打分加 100，概率不变：", np.allclose(softmax(z + 100), p))
with np.errstate(over="ignore"):
    big = np.exp(np.float64(1000))                       # 超出 float64 的范围
print("直接算 e^1000：", big, "；减去最大值后 softmax([1000, 999]) =",
np.round(softmax([1000, 999]), 4).tolist())

# 对数：把乘法变成加法。一句话 4 个词，每个词的概率相乘会越来越小，取对数后相加
probs = np.array([0.4666, 0.2, 0.1, 0.05])
print(f"概率相乘 {probs.prod():.6f}，对数相加 {np.log(probs).sum():.4f}，e^(对数和) =
{np.exp(np.log(probs).sum()):.6f}")
print(f"1000 个 0.01 相乘：{np.prod(np.full(1000, 0.01))}（下溢成 0）；对数相加：
{np.log(np.full(1000, 0.01)).sum():.1f}")
```

输出：

```
打分 z： {'我': 1.64, '爱': 0.83, '学': -0.39, 'AI': -0.17, '推理': -0.46, '<eos>':
0.37}
e^z： [5.155, 2.293, 0.677, 0.844, 0.631, 1.448] ，和 = 11.048
softmax： [0.4666, 0.2076, 0.0613, 0.0764, 0.0571, 0.131] ，和 = 1.0
和 tinylm.json 里存的概率一致： True
温度 T = 0.5： [0.749, 0.148, 0.013, 0.02, 0.011, 0.059]
温度 T = 1.0： [0.467, 0.208, 0.061, 0.076, 0.057, 0.131]
温度 T = 2.0： [0.302, 0.201, 0.109, 0.122, 0.106, 0.16]
所有打分加 100，概率不变： True
直接算 e^1000： inf ；减去最大值后 softmax([1000, 999]) = [0.7311, 0.2689]
概率相乘 0.000467，对数相加 -7.6700，e^(对数和) = 0.000467
1000 个 0.01 相乘：0.0（下溢成 0）；对数相加：-4605.2
```

前四行是 softmax 的两步，结果和 tinylm.json 一致。接下来三行是温度 0.5、1、2 的分布。然后是平移不变性和防溢出：直接算 e^{1000} 得到 inf，减去最大值后 softmax 正常。最后两行是对数：乘法变加法，1000 个 0.01 相乘下溢成 0，对数相加是 −4605.2。

容易混淆的地方

1. **softmax 不是“最大值函数”**。它给每个选项都分一点概率，分数最高的拿得最多，但不是全部。它可以看作“软化了的 max”，这也是名字的由来；温度趋近 0 时才变成真正的“只取最大”。
2. **softmax 只看差值**。分数 (1, 2, 3) 和 (101, 102, 103) 得到的概率完全一样；但 (1, 2, 3) 和 (2, 4, 6) 不一样，乘以 2 相当于温度减半。
3. **对数概率是负数**。看到损失、困惑度里出现的负对数，不用奇怪：概率小于 1，对数自然是负的，加个负号后越小越好。

回顾

指数函数把任何数变成正数，保持大小顺序，并放大差距；对数把乘法变成加法，用来处理很多个概率相乘时的下溢。softmax 对一组分数先取指数再除以总和，得到加起来等于 1 的概率：TinyLM 读到“我”后的打分 1.64、0.83、−0.39……变成概率 0.4666、0.2076、0.0613……除以温度 T 再做 softmax，T 小分布变尖，T 大分布变平。所有分数同加一个数结果不变，所以实现时先减去最大值，防止溢出。大模型的输出层和注意力都靠 softmax。

有了概率，下一步是从中选出一个词。总选概率最高的那个？还是按概率随机抽？“按概率抽”到底怎么做，抽 100 次结果会是什么样？下一讲：**概率分布与采样**。

第07讲 概率与采样

06 指数、对数与 softmax

上一讲用 softmax 把模型的打分变成了概率：TinyLM 读到“我”之后，下一个词是“我”的概率 0.4666，“爱”0.2076，“学”0.0613，AI 0.0764，“推理”0.0571，`<eos>` 0.131。

接下来要真的选出一个词，接在后面，再去生成下一个（第 24 讲）。最简单的做法是每次都选概率最大的那个。可这样同一个问题永远得到同一个回答，而且常常会陷入重复。大模型通常**按概率随机抽**：概率 0.47 的词，大约一半的时候被选中；概率 0.06 的词，偶尔也会被选中。

“按概率随机抽”在计算机里到底怎么做？抽 100 次，结果会是什么样？这一讲讲概率分布和采样的基本概念。

概率分布

一个**概率分布**说的是：一件事有哪几种可能的结果，每种结果的可能性有多大。对“下一个词”来说，可能的结果就是词表里的每一个词，可能性就是 softmax 给出的概率。它要满足两个条件：每个概率都在 0 到 1 之间；所有概率加起来等于 1。

结果只有有限个（或可数个）、每个都有一个概率的分布叫**离散分布**，大模型输出的就是离散分布。另一类是**连续分布**，比如“0 到 1 之间的任意实数，取到每一段的可能性和这一段的长度成正比”，叫 **[0, 1) 上的均匀分布**。下面马上会看到，有了均匀分布，就能从任何离散分布里抽样。

怎么按概率抽

计算机能直接提供的随机数，是 [0, 1) 上均匀分布的随机数 u（比如 Python 的 `random.random()`）。用它来抽一个词，做法是：

1. **算累积概率**：把概率依次加起来。“我”0.4666，“我+爱”0.6742，再加“学”0.7355，再加 AI 0.8119，再加“推理”0.869，全部加完是 1。
2. **把 [0, 1) 切成几段**：[0, 0.4666) 归“我”，[0.4666, 0.6742) 归“爱”，[0.6742, 0.7355) 归“学”……每一段的长度正好等于那个词的概率。

3. **抽一个 u，看它落在哪一段**：u = 0.3 落在第一段，选“我”；u = 0.7 落在第三段，选“学”；u = 0.95 落在最后一段，选 `<eos>`。

因为 u 在 [0, 1) 上均匀分布，它落在某一段的可能性等于这一段的长度，也就是那个词的概率。这样就实现了“按概率抽”。

下面的演示可以一次次地抽，看 u 落在哪里，以及抽了很多次之后，各个词被选中的频率。

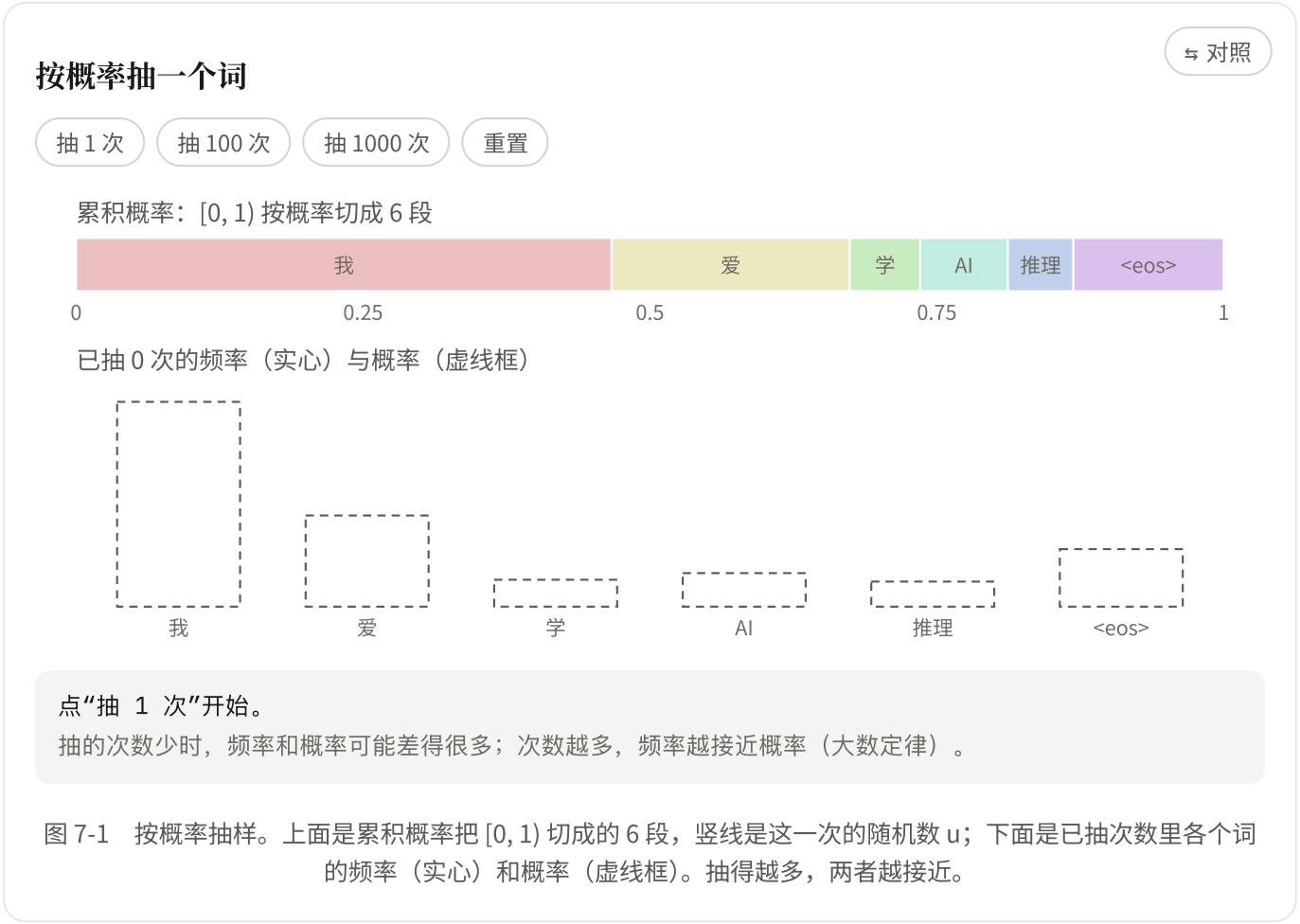


图 7-1　按概率抽样。上面是累积概率把 [0, 1) 切成的 6 段，竖线是这一次的随机数 u；下面是已抽次数里各个词的频率（实心）和概率（虚线框）。抽得越多，两者越接近。

要点

从一个离散分布里抽样：把概率依次累加，得到累积概率，把 [0, 1) 切成长度等于各个概率的几段；抽一个均匀的随机数 u，落在哪一段就选哪个。每个结果被选中的可能性正好等于它的概率。

抽多了，频率就接近概率

单独抽一次，结果是随机的。可抽很多次之后，每个词被选中的**频率**会越来越接近它的**概率**。这叫**大数定律**。代码用固定的随机种子抽了三组：

- **抽 10 次**：“我”的频率 0.3，“爱”0.3，AI 一次都没抽到，和概率差得很远。

- **抽 100 次**：“我”0.42，“爱”0.19，大致接近了。
- **抽 10000 次**：“我”0.468，“爱”0.206，“学”0.064……和概率只差千分之几。

这提醒我们两件事：一是用模型生成一次回答，看到的只是一次抽样的结果，不能代表模型的全部倾向；二是评测模型时，常常要对同一个问题生成多次、统计结果（第 49、51 讲）。

随机种子

计算机里的“随机数”其实是用一个确定的公式算出来的一串数，看起来没有规律，叫**伪随机数**。公式的起点叫**随机种子**。种子相同，产生的随机数序列就相同，抽样结果也就完全一样。

代码里的 `np.random.default_rng(0)` 就固定了种子 0，所以每次运行，抽 10 次的结果都是同样的那一组。调试、复现实验时，固定种子很有用；推理服务里，一般每个请求用不同的种子，有的 API 允许用户自己指定种子，以便复现同一个回答。

贪心和采样

贪心解码：每次都选概率最大的词，这里永远是“我”。结果确定、稳定，但容易单调、重复。

采样：按概率抽。结果多样，但也可能抽到概率很低、不太合适的词。

两者之间有很多折中的办法：先用温度（第 6 讲）把分布变尖或变平，或者只在概率最大的几个词里抽（Top-k、Top-p）。这些是第 73 讲解码策略的内容。

分布有多“确定”：熵

有的分布很集中（一个词占了 99%），有的很分散（每个词都差不多）。衡量这一点的量叫**熵**：

$$H = -\sum_i p_i \ln p_i$$

只有一个词可能时，熵为 0（完全确定）；n 个词完全均匀时，熵最大，等于 $\ln n$。这里 6 个词的分布，熵是 **1.479**，最大可能是 $\ln 6 = 1.792$，说明它比较分散。熵也和交叉熵损失、困惑度紧密相关（第 16 讲）。

用代码复现

下面的代码读取 TinyLM 读到“我”之后的分布，算出累积概率，演示几个 u 落在哪一段；然后用固定的种子各抽 10、100、10000 次，统计频率；最后看贪心的结果和分布的熵。

```
import json
import numpy as np

ex = json.load(open("tinylm.json", encoding="utf-8"))["examples"]["16"]
vocab, p = ex["vocab"], np.array(ex["p"][0])            # 第 6 讲算出的概率：读到“我”之后，下一个词的分布
print("分布：", dict(zip(vocab, p.tolist())), "，和 =", round(p.sum(), 4))

cdf = np.cumsum(p)                                      # 累积概率：把 [0, 1) 切成 6 段，每段的长度就是概率
print("累积：", np.round(cdf, 4).tolist())

def sample(u):                                          # 抽一个 [0, 1) 里均匀的随机数 u，看它落在哪一段
    return vocab[int(np.searchsorted(cdf, u, side="right"))]
for u in [0.3, 0.7, 0.95]:
    print(f"  u = {u} → {sample(u)}")

rng = np.random.default_rng(0)                          # 固定随机种子：每次运行抽到的结果都一样
for n in [10, 100, 10000]:
    draws = [sample(u) for u in rng.random(n)]
    freq = [draws.count(w) / n for w in vocab]
    print(f"抽 {n:5d} 次的频率：", np.round(freq, 3).tolist())

print("贪心（总取概率最大的）：", vocab[int(p.argmax())])
H = -(p * np.log(p)).sum()
print(f"熵 = {H:.3f}（6 个词均匀分布时最大，ln 6 = {np.log(6):.3f}；只有一个词时为 0）")
```

输出：

```
分布： {'我': 0.4666, '爱': 0.2076, '学': 0.0613, 'AI': 0.0764, '推理': 0.0571,
'<eos>': 0.131} ，和 = 1.0
累积： [0.4666, 0.6742, 0.7355, 0.8119, 0.869, 1.0]
  u = 0.3 → 我
  u = 0.7 → 学
  u = 0.95 → <eos>
抽    10 次的频率： [0.3, 0.3, 0.1, 0.0, 0.1, 0.2]
抽   100 次的频率： [0.42, 0.19, 0.08, 0.05, 0.06, 0.2]
抽 10000 次的频率： [0.468, 0.206, 0.064, 0.074, 0.062, 0.127]
贪心（总取概率最大的）： 我
熵 = 1.479（6 个词均匀分布时最大，ln 6 = 1.792；只有一个词时为 0）
```

第一、二行是分布和累积概率；接下来三行是 u = 0.3、0.7、0.95 各选中哪个词；然后是三组抽样的频率，次数越多越接近概率；最后是贪心的结果和熵。

容易混淆的地方

1. **概率高不等于一定被选中**。“我”的概率 0.47，意思是大约一半的时候被选中，另一半的时候选的是别的词。
2. **频率不等于概率**。频率是抽样的结果，会随抽样次数波动；概率是分布本身的性质。只有抽样次数足够多时，两者才接近。
3. **“随机”是可以复现的**。固定随机种子，同样的程序会得到同样的“随机”结果；换了种子或者换了运行环境（比如不同的 GPU 并行顺序），结果才会变。

回顾

概率分布描述一件事有哪些可能的结果、每种结果的可能性有多大，模型输出的下一个词的概率就是一个离散分布。按概率抽样的方法是：算累积概率，把 [0, 1) 切成长度等于各个概率的几段，抽一个均匀随机数 u，落在哪段选哪个。TinyLM 读到“我”后，u = 0.3 选“我”，u = 0.7 选“学”。抽得越多，频率越接近概率：抽 10 次差得很远，抽 10000 次“我”的频率是 0.468。随机种子决定了伪随机数序列，固定它就能复现结果。贪心总选概率最大的，采样按概率抽；熵衡量分布有多分散，这个分布的熵是 1.479。

到这里，我们已经能用模型生成文字了。可模型的参数是怎么学出来的？训练要知道“改动一点参数，损失会变化多少”，这就是导数。多层网络里，参数藏在一层套一层的函数里，又该怎么算？下一讲：**导数、偏导与链式法则**。

第08讲 导数与链式法则

06 指数、对数与 softmax

训练一个模型，说到底就是不断地调整它的参数，让预测越来越准（第 17 讲）。每调一次，都要回答一个问题：**这个参数往哪个方向改、改多少，损失会下降？**要回答它，就得知道“参数变一点，损失会变多少”。这个“变化率”，就是导数。

大模型有几十亿个参数，每个参数都藏在一层套一层的计算里。这一讲复习三件事：导数是什么，多个变量时的偏导数，以及一层套一层时怎么算导数的链式法则。它们是第 18 讲反向传播的全部数学基础。

导数：变化有多快

函数 f(x) 在某一点的**导数**，说的是：x 在这一点附近变化一点点时，f 变化的速度。几何上，它就是曲线在这一点的**切线的斜率**。

怎么算？取一个很小的 h，看 x 变成 x + h 时 f 变了多少，除以 h：

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

这个比值是连接曲线上两点的**割线**的斜率。h 越小，两点越近，割线越接近切线。

以 $f(x) = x^2$ 在 x = 3 为例：

- h = 0.1：$(3.1^2 - 3^2) / 0.1 = \mathbf{6.1}$；
- h = 0.01：$\mathbf{6.01}$；
- h = 0.001：$\mathbf{6.001}$。

越来越接近 6。用求导公式，x^2 的导数是 2x，x = 3 时正好是 6。

导数的意思是：x 从 3 增加一个很小的量 Δ，f 大约增加 6Δ。它是一个**放大倍数**：输入的小变化，在输出里被放大了多少倍。导数为正，f 随 x 增大而增大；为负，则减小；为 0，这一点附近是平的（可能是最低点或最高点）。

常用的几个导数，后面会反复用到：

函数	导数
x^n	$n \cdot x^{n-1}$
e^x	e^x
$\ln x$	$1/x$
$\sin x$、$\cos x$	$\cos x$、$-\sin x$
sigmoid $\sigma(x) = 1/(1+e^{-x})$	$\sigma(x)\cdot(1-\sigma(x))$

偏导：多个变量时，只让一个动

模型的损失取决于很多个参数。先看一个最简单的：用直线 $wx + b$ 去拟合一个数据点 $(x, y) = (2, 7)$，损失是误差的平方：

$$L(w, b) = (wx + b - y)^2$$

现在 L 是 w 和 b 两个变量的函数。问“L 变化有多快”，要先说清楚是哪个变量在变。**偏导数** $\partial L/\partial w$ 的意思是：**b 保持不动，只让 w 变化一点点**，L 变化的速度。把 b 当常数，对 w 求导就行。

在 $w = 1$、$b = 1$ 处，$wx + b - y = 2 + 1 - 7 = -4$，$L = 16$。

- $\partial L/\partial w = 2\cdot(wx + b - y)\cdot x = 2 \times (-4) \times 2 = \mathbf{-16}$；
- $\partial L/\partial b = 2\cdot(wx + b - y) = -8$。

验证一下：w 加 0.01，L 从 16 变成 15.8404，变化 −0.1596，约等于 -16×0.01。两个偏导数都是负的，说明把 w 和 b 往**增大**的方向调，损失会下降；而 w 的偏导绝对值更大，调 w 效果更明显。

把所有偏导数排成一个向量，叫**梯度**：$(\partial L/\partial w, \partial L/\partial b) = (-16, -8)$。它指向 L 增长最快的方向。梯度下降（第 17 讲）就是沿梯度的反方向走一小步。

链式法则：一层套一层

神经网络是一层套一层的函数：输入经过第一层得到中间结果，中间结果经过第二层……最后得到损失。参数在前面的层里，它对损失的影响要经过后面的每一层才传到。

链式法则说：一串影响接起来，总的放大倍数就是每一段放大倍数的乘积。

看一个三步的例子：$x \to u = 3x + 1 \to v = u^2 \to y = \ln v$。在 $x = 0.5$ 处：

- $u = 2.5$，$v = 6.25$，$y = \ln 6.25 = 1.8326$。
- 每一步的局部导数：$du/dx = 3$，$dv/du = 2u = 5$，$dy/dv = 1/v = 0.16$。

x 变一点，u 变 3 倍那么多；u 变一点，v 变 5 倍那么多；v 变一点，y 变 0.16 倍那么多。所以 x 变一点，y 变

$$\frac{dy}{dx} = \frac{dy}{dv} \cdot \frac{dv}{du} \cdot \frac{du}{dx} = 0.16 \times 5 \times 3 = 2.4$$

数值验证：x 加 10^{-6} ，y 的变化除以 10^{-6} ，也是 2.4000。

链式法则的好处是：**不需要把整个复杂函数写出来求导**，只要知道每一步自己的局部导数，乘起来就行。每一步的局部导数都很简单（加法、乘法、平方、对数、激活函数），整个网络的导数就能一步步算出来。

下面的演示上半部分可以选函数、拖动 x，看切线和割线；下半部分是上面这个三步的链，拖动 x 看每一步的值和局部导数。

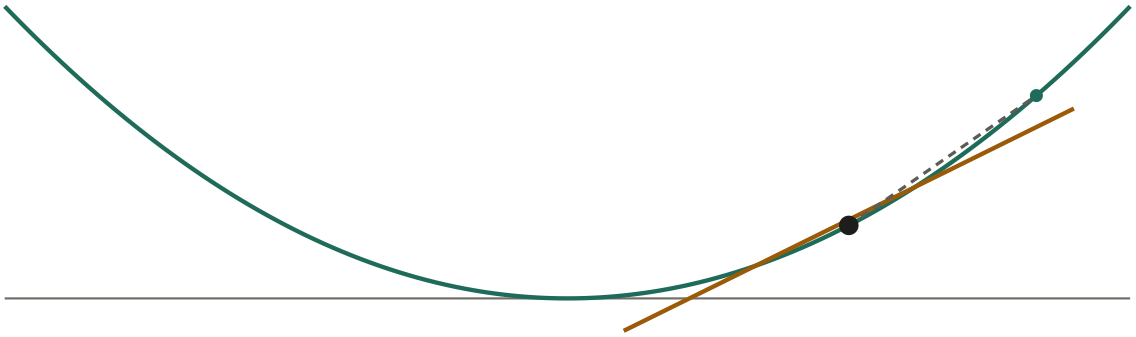
⇆ 对照

斜率，以及一串斜率相乘

x^2 | $\sin x$ | e^x | $\sigma(x)$

x = **1.5**

h = **1**



f(1.5) = 2.2500；导数 f′(x) = 2x = **3.0000**（实线是切线）。
割线斜率 (f(x + h) - f(x)) / h = 4.0000（虚线），和导数差 1.0000。h 越小，割线越接近切线。

链式法则：$x \to u = 3x + 1 \to v = u^2 \to y = \ln v$

x = **0.5**

x	→	u = 3x + 1	→	$v = u^2$	→	y = ln v
0.5000	×3.0	**2.5000**	×5.0	**6.2500**	×0.1600	**1.8326**

局部导数：du/dx = 3，dv/du = 2u = 5.00，dy/dv = 1/v = 0.1600。
dy/dx = 0.1600 × 5.00 × 3 = **2.4000**（直接求导：dy/dx = 6/(3x + 1) = 2.4000）。
x 变一点，u 变 3 倍那么多；u 变一点，v 变 2u 倍；v 变一点，y 变 1/v 倍。一串影响接起来，总的倍数就是每一段倍数相乘。

图 8-1　上：导数是切线的斜率，h 越小，割线（虚线）越接近切线（实线）。下：链式法则，x 经过三步得到 y，每一步的局部导数相乘就是 dy/dx。

要点

导数是函数在一点的变化率（切线斜率），可以理解为输入的小变化被放大的倍数。多个变量时，偏导数只让一个变量变化，所有偏导数排成梯度，指向函数增长最快的方向。函数一层套一层时，链式法则把每一步的局部导数相乘，得到总的导数。

多条路径怎么办

上面的链是一条直线。实际的网络里，一个变量常常通过好几条路径影响结果。比如 z = u·v，而 u、v 都依赖 x。这时 x 对 z 的影响是**各条路径的影响加起来**：

$$\frac{dz}{dx} = \frac{\partial z}{\partial u} \cdot \frac{du}{dx} + \frac{\partial z}{\partial v} \cdot \frac{dv}{dx}$$

路径上相乘，路径之间相加。第 18 讲的反向传播，就是把这个规则系统地应用到整张计算图上：从损失出发，沿着每条边往回乘局部导数，在分叉处把各条路径的贡献加起来。第 29 讲的残差连接，也正是靠“多了一条路径”让梯度更容易传回去。

到了大模型里

每个参数一个偏导。Llama-3-8B 有 80 亿个参数，训练的每一步都要算出损失对每一个参数的偏导数，也就是一个 80 亿维的梯度。靠的就是链式法则和反向传播。

梯度消失和爆炸。链式法则是连乘。几十层的网络，如果每层的局部导数都小于 1（比如 sigmoid 的导数最大只有 0.25），乘起来就趋近 0，前面的层几乎学不到东西（梯度消失）；都大于 1 则会爆炸。激活函数的选择（第 14 讲）、残差连接（第 29 讲）、归一化（第 30 讲）都和这个问题有关。

推理不需要导数。导数只在训练时用。推理只做前向计算，不用存中间结果去算梯度，所以推理比训练省得多（第 99 讲）。

用代码复现

下面的代码用越来越小的 h 逼近 x^2 在 3 处的导数；算 $L = (wx + b - y)^2$ 的两个偏导数并验证；算三步链的导数，和数值方法对比；最后看 sigmoid 的导数。

```python
import math

# 1. 导数 = 切线的斜率：f(x) = x² 在 x = 3 处，用越来越小的 h 去逼近
f = lambda x: x * x
for h in [0.1, 0.01, 0.001]:
    print(f"h = {h}: (f(3 + h) - f(3)) / h = {(f(3 + h) - f(3)) / h:.4f}")
print("公式 f'(x) = 2x，x = 3 时 = 6")

# 2. 偏导：L(w, b) = (w·x + b - y)²，x = 2，y = 7，在 w = 1、b = 1 处
x, y, w, b = 2.0, 7.0, 1.0, 1.0
L = lambda w, b: (w * x + b - y) ** 2
r = w * x + b - y
print(f"L = {L(w, b):.0f}；∂L/∂w = 2·(wx + b - y)·x = {2 * r * x:.0f}，∂L/∂b =
2·(wx + b - y) = {2 * r:.0f}")
print(f"w 加 0.01：L 从 {L(w, b):.4f} 变成 {L(w + 0.01, b):.4f}，变化 {L(w + 0.01, b)
- L(w, b):.4f} ≈ -16 × 0.01")

# 3. 链式法则：x → u = 3x + 1 → v = u² → y = ln v
x = 0.5
u = 3 * x + 1; v = u ** 2; yv = math.log(v)
du, dv, dy = 3, 2 * u, 1 / v                                    # 每一步的局部导数
print(f"x = {x} → u = {u} → v = {v} → y = {yv:.4f}")
print(f"dy/dx = (dy/dv)·(dv/du)·(du/dx) = {dy:.2f} × {dv:.1f} × {du} = {dy * dv *
du:.2f}")
g = lambda x: math.log((3 * x + 1) ** 2)
print(f"数值验证：(g(0.5 + 1e-6) - g(0.5)) / 1e-6 = {(g(0.5 + 1e-6) - g(0.5)) / 1e-
6:.4f}")

# 4. 几个常用的导数（深度学习里反复出现）
sig = lambda z: 1 / (1 + math.exp(-z))
z = 0.0
print(f"sigmoid 在 0 处：σ = {sig(z)}，σ' = σ(1 - σ) = {sig(z) * (1 - sig(z))}")
```

输出：

```
h = 0.1: (f(3 + h) - f(3)) / h = 6.1000
h = 0.01: (f(3 + h) - f(3)) / h = 6.0100
h = 0.001: (f(3 + h) - f(3)) / h = 6.0010
公式 f'(x) = 2x, x = 3 时 = 6
L = 16; ∂L/∂w = 2·(wx + b - y)·x = -16, ∂L/∂b = 2·(wx + b - y) = -8
w 加 0.01: L 从 16.0000 变成 15.8404, 变化 -0.1596 ≈ -16 × 0.01
x = 0.5 → u = 2.5 → v = 6.25 → y = 1.8326
dy/dx = (dy/dv)·(dv/du)·(du/dx) = 0.16 × 5.0 × 3 = 2.40
数值验证: (g(0.5 + 1e-6) - g(0.5)) / 1e-6 = 2.4000
sigmoid 在 0 处: σ = 0.5, σ' = σ(1 - σ) = 0.25
```

前四行是导数的逼近：6.1、6.01、6.001 → 6。接着两行是偏导数：$\partial L/\partial w = -16$、$\partial L/\partial b = -8$，w 加 0.01 时 L 变化 −0.1596。然后是链式法则：$0.16 \times 5 \times 3 = 2.4$，数值验证也是 2.4。最后是 sigmoid 在 0 处的导数 0.25。

容易混淆的地方

1. **导数是“局部”的**。它只描述这一点附近的变化，离远了就不准了。所以梯度下降每次只能走一小步，走远了，梯度指的方向就不对了。
2. **偏导数要说清楚对谁**。$\partial L/\partial w$ 和 $\partial L/\partial b$ 是不同的数；“L 的导数”这种说法在多变量时没有意义，要说“L 对 w 的偏导”。
3. **链式法则是乘，多路径是加**。同一条路径上的局部导数相乘，不同路径的贡献相加，两者不要混。

回顾

导数是函数在一点的变化率，几何上是切线的斜率，可以用割线斜率 $(f(x+h) - f(x)) / h$ 在 h 趋近 0 时逼近：x^2 在 3 处，h = 0.1、0.01、0.001 时依次是 6.1、6.01、6.001，导数是 6。多个变量时，偏导数只让一个变量动：$L = (wx + b - y)^2$ 在 $w = b = 1$ 处，$\partial L/\partial w = -16$、$\partial L/\partial b = -8$，排成梯度。函数一层套一层时，链式法则把每一步的局部导数相乘：$x \to 3x + 1 \to u^2 \to \ln v$ 在 $x = 0.5$ 处，$0.16 \times 5 \times 3 = 2.4$。多条路径时，各条路径的贡献相加。这些是反向传播和训练的数学基础。

接下来是一讲选读的数学：三角函数和旋转。它看起来和大模型没什么关系，却是位置编码和 RoPE 的基础：模型怎么知道一个词在第几个位置？下一讲：**三角函数与旋转**。

第09讲 三角函数与旋转

选读

02 点积与余弦相似度　04 线性变换

这是一讲选读的数学。三角函数看起来和大模型没什么关系，但它是两个重要设计的基础：第 25 讲的**位置编码**用正弦、余弦给每个位置一个独特的“签名”，第 37 讲的 RoPE（今天几乎所有大模型都在用）用旋转把位置信息编进 q 和 k 里。

这一讲只复习那两讲要用到的几件事：单位圆上的 sin 和 cos，旋转的公式，以及旋转最关键的一个性质：**两个向量一起转同样的角度，点积不变**。

单位圆上的 sin 和 cos

在平面上画一个半径为 1、圆心在原点的圆（单位圆）。从正 x 轴开始，逆时针转一个角度 θ，到达圆上的一个点。这个点的坐标就定义为

$$(\cos\theta,\ \sin\theta)$$

几个特殊的角：θ = 0°，点在 (1, 0)；30°，在 (0.866, 0.5)；90°，在 (0, 1)；180°，在 (−1, 0)。转一整圈（360°，也就是 2π 弧度）回到原点，所以 sin、cos 都是**周期函数**，周期是 2π。

因为点在半径为 1 的圆上，由勾股定理，总有 $\cos^2\theta + \sin^2\theta = 1$。

角度在计算里通常用**弧度**：一整圈是 2π 弧度，180° 是 π，1 弧度约 57.3°。

旋转一个向量

第 4 讲说过，旋转是一种线性变换，可以写成矩阵乘法。把一个二维向量 (a, b) 逆时针旋转 φ，得到

$$(a\cos\varphi - b\sin\varphi,\ \ a\sin\varphi + b\cos\varphi)$$

为什么是这样？用第 4 讲的看法：变换由“坐标方向被送到哪里”决定。x 方向 (1, 0) 转 φ 后到达 $(\cos\varphi, \sin\varphi)$，y 方向 (0, 1) 转 φ 后到达 $(-\sin\varphi, \cos\varphi)$。(a, b) 是 a 个 x 方向加 b 个 y 方向，转完以后就是 $a\cdot(\cos\varphi, \sin\varphi) + b\cdot(-\sin\varphi, \cos\varphi)$，展开就是上面的式子。

比如 (1, 0.5) 逆时针转 30°，得到 (0.616, 0.933)。

旋转有三个重要的性质：

1. **不改变长度**。(1, 0.5) 的长度是 1.1180，转完还是 1.1180。
2. **连续旋转可以合并**。先转 20° 再转 40°，等于直接转 60°。角度相加。
3. **不改变两个向量之间的夹角**。所以，两个向量一起转同样的角度，它们的点积（第 2 讲：长度 × 长度 × 夹角的余弦）也不变。

关键性质：点积只看相对的转动

第三条性质值得仔细看。取 $q = (0.7, -0.6)$、$k = (0.3, 0.6)$，点积 $q\cdot k = 0.7 \times 0.3 + (-0.6) \times 0.6 = \mathbf{-0.15}$。两个都逆时针转 50°，点积还是 **−0.15**。

再进一步：让 q 转 m 个 φ、k 转 n 个 φ（转的角度不同）。两者的夹角改变了 $(m - n)\varphi$，点积就变了，但它**只取决于 $\mathbf{m - n}$**，不取决于 m、n 各自是多少。代码取 $\varphi = 10°$：

- q 转 3φ、k 转 1φ（相差 2 步）：点积 0.0643；
- q 转 7φ、k 转 5φ（相差 2 步）：0.0643；
- q 转 12φ、k 转 10φ（相差 2 步）：0.0643。

三种情况结果一样。

这正是 RoPE 的核心想法（第 37 讲）：把第 m 个位置的 q 转 m 个 φ，第 n 个位置的 k 转 n 个 φ，它们的点积（注意力的打分）就只取决于两个词之间的**相对距离** $m - n$。一句话整体往后挪几个位置，词与词之间的关系不变。

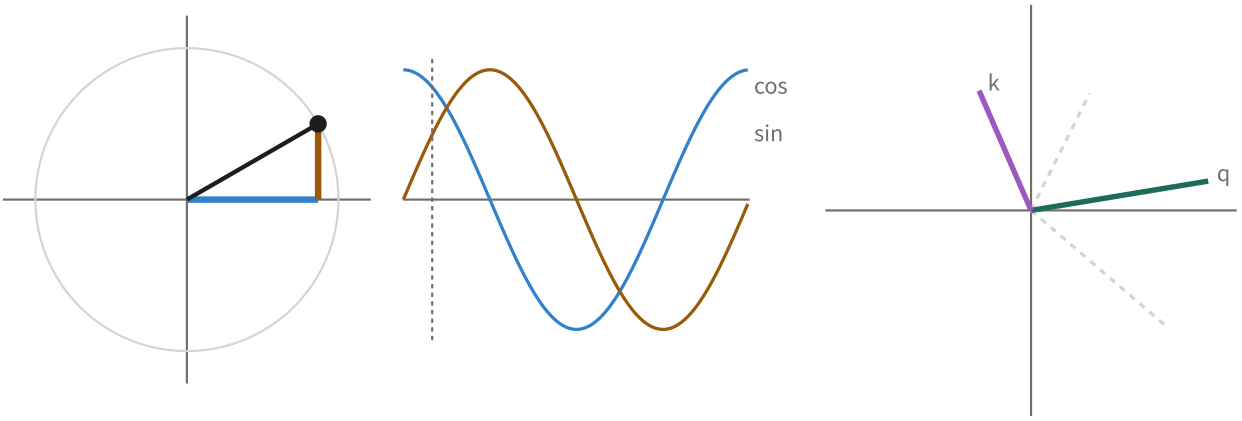
下面的演示左边是单位圆和 sin、cos 曲线，右边是一起旋转的 q 和 k。

单位圆，以及一起转的两个向量

⇆ 对照

θ = **30°**

一起转 φ = **50°**



θ = 30°：(cos θ, sin θ) = (0.866, 0.500)，cos²θ + sin²θ = 1.000。
q、k 一起转 50°：q = (0.910, 0.151)，k = (-0.267, 0.615)；长度 0.9220、0.6708，点积 q·k = **-0.1500**（转之前 -0.1500）。
旋转不改变长度，也不改变两个向量之间的夹角，所以一起转同样的角度，点积不变（灰线是转之前的位置）。

图 9-1　左：单位圆上角度 θ 的点是 (cos θ, sin θ)，蓝色和琥珀色的线段是它在两条轴上的投影，右边是 cos、sin 随 θ 的曲线。右：q、k 一起旋转 φ，长度和夹角不变，点积始终是 −0.15。

要点

单位圆上角度 θ 的点是 (cos θ, sin θ)，它们是周期为 2π 的函数。向量 (a, b) 逆时针旋转 φ 得到 (a cos φ − b sin φ, a sin φ + b cos φ)。旋转不改变长度和夹角，所以两个向量一起转同样的角度，点积不变；分别转 mφ 和 nφ 时，点积只取决于 m − n。

转得快和转得慢

最后一件事：同样每走一步转一下，转的角度可以有大有小。每步转 θ 弧度，转一圈需要 $2\pi / \theta$ 步，这叫**周期**。

位置编码和 RoPE 都用了一组**不同的转速**：把 d 维向量两两分组，第 i 组每走一个位置转 $\theta_i = 10000^{-2i/d}$ 弧度。

- 第 0 组：$\theta = 1$ 弧度，大约 6 个位置转一圈，转得最快。

- d = 4 时的第 1 组：$\theta = 0.01$，约 628 个位置转一圈。
- d = 128 时的最后一组（第 63 组）：θ 约 0.000115，约 54,410 个位置才转一圈，转得最慢。

为什么要这样？转得快的组对近处的位置差很敏感（差一两个位置，角度就差很多），但转了一圈之后会“绕回来”，分不清相隔 6 个和 12 个位置；转得慢的组能区分很远的位置，但对近处的差别不敏感。快慢搭配，就像钟表的秒针、分针、时针，合起来能精确地表示从近到远的各种距离。第 25、37、42 讲都会用到这个想法。

用代码复现

下面的代码算几个特殊角的 cos、sin；把 (1, 0.5) 转 30°，验证长度不变、连续旋转可合并；验证一起转时点积不变、分别转 $m\varphi$ 和 $n\varphi$ 时点积只取决于 $m - n$；最后算几组转速的周期。

```python
import numpy as np

# 1. 单位圆上的点：角度 θ 对应 (cos θ, sin θ)
for deg in [0, 30, 90, 180]:
    t = np.radians(deg)
    print(f"θ = {deg:3d}°: (cos θ, sin θ) = ({np.cos(t):.3f}, {np.sin(t):.3f})")

def rot(phi):                                   # 逆时针旋转 φ 的矩阵（第 4 讲：每一行是一个坐标方向被送去的位置）
    c, s = np.cos(phi), np.sin(phi)
    return np.array([[c, s], [-s, c]])          # 行向量写法：v @ R

v = np.array([1.0, 0.5])
w = v @ rot(np.radians(30))
print(f"(1, 0.5) 逆时针转 30° → ({w[0]:.3f}, {w[1]:.3f})；长度 {np.linalg.norm(v):.4f} → {np.linalg.norm(w):.4f}")
print("先转 20° 再转 40° = 直接转 60°: ", np.allclose(v @ rot(np.radians(20)) @ rot(np.radians(40)), v @ rot(np.radians(60))))

# 2. 两个向量一起转同样的角度，点积不变；分别转 mφ、nφ，点积只取决于 m - n
q, k = np.array([0.7, -0.6]), np.array([0.3, 0.6])
print(f"q·k = {q @ k:.4f}；两个都转 50° 后 = {(q @ rot(np.radians(50))) @ (k @ rot(np.radians(50))):.4f}")
phi = np.radians(10)
for m, n in [(3, 1), (7, 5), (12, 10)]:
    print(f"q 转 {m}φ、k 转 {n}φ（相差 {m - n} 步）：点积 {(q @ rot(m * phi)) @ (k @ rot(n * phi)):.4f}")

# 3. 不同频率的转动：每走一步转的角度不同，转一圈要的步数（周期）也不同
for i, d in [(0, 4), (1, 4), (0, 128), (63, 128)]:
    theta = 10000 ** (-2 * i / d)
    print(f"d = {d}，第 {i} 组：每步转 {theta:.6f} 弧度，转一圈要 {2 * np.pi / theta:,.0f} 步")
```

输出：

```
θ =   0°: (cos θ, sin θ) = (1.000, 0.000)
θ =  30°: (cos θ, sin θ) = (0.866, 0.500)
θ =  90°: (cos θ, sin θ) = (0.000, 1.000)
θ = 180°: (cos θ, sin θ) = (-1.000, 0.000)
(1, 0.5) 逆时针转 30° → (0.616, 0.933)；长度 1.1180 → 1.1180
先转 20° 再转 40° = 直接转 60°:  True
q·k = -0.1500；两个都转 50° 后 = -0.1500
q 转 3φ、k 转 1φ（相差 2 步）：点积 0.0643
q 转 7φ、k 转 5φ（相差 2 步）：点积 0.0643
q 转 12φ、k 转 10φ（相差 2 步）：点积 0.0643
d = 4，第 0 组：每步转 1.000000 弧度，转一圈要 6 步
d = 4，第 1 组：每步转 0.010000 弧度，转一圈要 628 步
d = 128，第 0 组：每步转 1.000000 弧度，转一圈要 6 步
d = 128，第 63 组：每步转 0.000115 弧度，转一圈要 54,410 步
```

前四行是单位圆上的点。第五、六行：旋转后长度不变，20° + 40° = 60°。接着是点积：一起转 50° 仍是 −0.15，相差 2 步的三种情况都是 0.0643。最后四行是不同转速转一圈要的步数：从 6 步到 54,410 步。

容易混淆的地方

1. **角度和弧度**。数学公式和代码里的 sin、cos 都用弧度：`np.sin(30)` 是 30 弧度的正弦，不是 30°。要先换算（乘以 π/180）。
2. **旋转矩阵的写法有两种**。列向量写法是 $R \cdot v$，行向量写法是 $v \cdot R^T$（第 3 讲用行向量）。两者结果一样，只是矩阵转置了一下，读代码时要留意。
3. **点积不变的前提是转同样的角度**。转不同的角度，点积会变，但只取决于角度差。这正是 RoPE 能编码“相对位置”的原因。

回顾

单位圆上角度 θ 的点是 $(\cos\theta, \sin\theta)$，sin、cos 是周期为 2π 的函数，$\cos^2\theta + \sin^2\theta = 1$。把 (a, b) 逆时针旋转 φ 得到 $(a\cos\varphi - b\sin\varphi,\ a\sin\varphi + b\cos\varphi)$，比如 $(1, 0.5)$ 转 30° 得到 $(0.616, 0.933)$。旋转不改变长度和夹角，连续旋转角度相加；两个向量一起转同样的角度，点积不变（$q \cdot k$ 始终是 −0.15）；q 转 $m\varphi$、k 转 $n\varphi$ 时，点积只取决于 $m - n$，这是 RoPE 的核心。不同的转速对应不同的周期，从 6 步到几万步，快慢搭配能表示从近到远的距离。

我们一直在用 0.7、−0.6 这样的小数，好像计算机能精确地存下任何实数。其实不能：计算机里的数只有有限的位数，大模型为了快，还在用越来越少的位数存数。数在计算机里到底是怎么表示的，会有多大的误差？下一讲：**浮点数与数值精度**。

第10讲 浮点数与精度

重点

01 向量

到目前为止，我们写下的 0.7、−0.6 这些数，好像计算机都能原样存下来。其实不能：计算机里的每个数只有固定的位数，32 位、16 位，甚至 8 位、4 位。有限的位数只能表示有限多个数，大部分实数只能用离它最近的一个来近似。

这件事对大模型非常重要。模型有几十亿个参数，每个数用 16 位还是 8 位存，显存和读取的时间就差一倍；GPU 用低精度计算，速度也快一倍（第 63 讲）。可位数越少，近似得越粗，模型的效果可能变差。这一讲讲清楚浮点数是怎么表示一个数的，FP32、FP16、BF16、FP8、FP4 有什么区别，以及精度不够时会出现哪些问题。

像科学计数法一样存数

浮点数的思路和科学计数法一样：把一个数写成“有效数字 × 底数的若干次方”。比如 0.1 = 1.6 × 2^{-4}，3.14159 = 1.5708 × 2^{1}。用二进制时，有效数字总是 1.xxxx 的形式，开头的 1 不用存。

于是一个浮点数分三部分存：

- **符号**（1 位）：0 是正，1 是负。
- **指数**（若干位）：2 的几次方。为了表示负的指数，实际存的是“指数 + bias”，bias 是一个固定的偏移。
- **尾数**（若干位）：1.xxxx 小数点后的那些位。

它表示的值是

$$(-1)^{\text{符号}} \times \left(1 + \frac{\text{尾数}}{2^{\text{尾数位数}}}\right) \times 2^{\text{指数}-\text{bias}}$$

0.1 的 FP32（32 位）表示是：符号 0，指数 01111011（123，减去 bias 127 得 −4），尾数 10011001100110011001101。即 1.6000000238 × 2^{-4} = 0.1000000015。0.1 在二进制里是无限循环小数，23 位尾数只能截到这里，误差约 1.5 × 10^{-9} 。

几种格式

不同的格式，区别在于指数和尾数各分多少位：

格式	总位数	指数位	尾数位	最大值	1 附近的间隔
FP32	32	8	23	3.4×10^{38}	1.19×10^{-7}
FP16	16	5	10	65504	9.77×10^{-4}
BF16	16	8	7	3.4×10^{38}	7.81×10^{-3}
FP8 E4M3	8	4	3	448	0.125
FP4 E2M1	4	2	1	6	0.5

两条规律：

- **尾数位决定精度**。尾数有 m 位，1 和它后面那个数之间的间隔是 2^{-m}，叫**机器精度**。相对误差最多是它的一半。FP32 有 7 位左右的十进制有效数字，FP16 约 3 位，BF16 只有 2 到 3 位。
- **指数位决定范围**。指数有 e 位，能表示的数的大小范围大约是 2 的 $\pm 2^{e-1}$ 次方。FP16 的指数只有 5 位，最大只能到 65504；BF16 的指数和 FP32 一样是 8 位，范围也一样大。

FP16 和 BF16 都是 16 位，分法不同：FP16 精度更高、范围小，BF16 精度低、范围大。BF16（brain float）是 Google 为深度学习设计的，它就是把 FP32 的尾数砍掉 16 位，和 FP32 之间的转换非常简单。

下面的演示可以输入任意一个数，看它在各种格式下的比特、实际存下的值和误差。



图 10-1　位级查看器。同一个数在各种格式下的符号（灰）、指数（琥珀）、尾数（绿）以及实际存下的值。试试 70000（FP16 溢出）、1e-8（FP16 下溢）、448（FP8 能表示的最大值）。

要点

浮点数用符号、指数、尾数三部分表示一个数，像二进制的科学计数法。尾数位数决定精度（相对误差约是 2^−(尾数位数+1)），指数位数决定能表示的范围。FP16 精度较高但最大只到 65504；BF16 精度较低，但范围和 FP32 一样大；FP8、FP4 的精度和范围都很有限。

精度不够时会出什么问题

舍入误差。0.1 在 FP16 里存成 0.0999755859，在 BF16 里存成 0.1000976562；π 在两者里都存成 3.140625。单个数的误差看起来不大，但会一步步累积。

溢出。超过最大值的数无法表示，变成无穷大（∞）。70000 在 FP16 里溢出成 ∞，在 BF16 里是 70144（不精确，但没溢出）。训练时如果某个中间结果溢出，后面的计算就全乱了，这是早年用 FP16 训练大模型很痛苦的原因，也是 BF16 流行起来的原因。

下溢。太接近 0 的数无法表示，变成 0。10^{-8} 在 FP16 里下溢成 0，BF16 里是 1.001×10^{-8}。很小的梯度一旦下溢成 0，参数就不会更新了。

大数吃小数。两个大小悬殊的数相加，小的那个可能被“舍入掉”。代码把 0.01 在 FP16 里一个个加 10000 次（正确答案 100）：结果停在了 **32.0**。因为 32 附近 FP16 的间隔是 0.03125，加 0.01 不到半个间隔，被舍入回 32，之后怎么加都不变。同样的事情用 FP32 做，得到 100.003。所以累加（矩阵乘法里的乘加、求和）通常要用比输入更高的精度。

混合精度：为什么还要存一份 FP32

大模型训练用 BF16 做前向和反向（快、省显存），但参数的更新往往很小：学习率乘以梯度，可能只有 10^{-4}、10^{-5} 。1.0 附近 BF16 的间隔是 0.0078，一个 0.0001 的更新加上去，直接被舍入掉了。

代码模拟了这件事：一个权重 1.0，每一步加 0.0001，加 1000 次，应该变成 1.1。用 BF16 存权重，**一直是 1.0**，一点也没学到；用 FP32 存，得到 **1.1000**。

所以混合精度训练要另存一份 **FP32 的主权重**：计算用 BF16，更新加在 FP32 的主权重上，再转成 BF16 给下一步用。这就是第 19 讲说的“每个参数多存 4 字节”的原因。

推理为什么能用更低的精度

推理没有微小的更新累积，只需要前向计算足够准确。实践表明，权重和激活用 BF16、FP8，甚至 4 位，模型的效果都还能保持（需要一些技巧，第 90–94 讲）。位数减半，显存和读取的字节减半，GPU 的计算也快一倍（第 63 讲）。这是大模型推理优化最重要的手段之一。

用代码复现

下面的代码打出 0.1 的 FP32、BF16 位表示，比较 0.1 和 π 在各种格式下存成的值；看最大值、溢出、下溢和机器精度；最后演示两个累积误差：FP16 里 0.01 加 10000 次，以及 BF16 存权重时小更新被舍入掉。（BF16 用 FP32 的高 16 位模拟。）

```python
import numpy as np

def bf16(x):
    """把 FP32 舍入到 BF16：保留高 16 位（1 位符号、8 位指数、7 位尾数），就近舍入"""
    b = np.array(x, dtype=np.float32).view(np.uint32)
    b = (b + 0x7FFF + ((b >> 16) & 1)) & 0xFFFF0000
    return b.view(np.float32)

def bits32(x):
    s = format(np.array(x, dtype=np.float32).view(np.uint32).item(), "032b")
    return f"{s[0]} {s[1:9]} {s[9:]}"

print("0.1 的 FP32 位：", bits32(0.1), "（符号 | 8 位指数 | 23 位尾数）")
print("0.1 的 BF16 位：", bits32(bf16(0.1))[:18], "（只留 7 位尾数）")
for x in [0.1, 3.14159265]:
    v32, v16, vb = float(np.float32(x)), float(np.float16(x)), float(bf16(x))
    print(f"{x}：FP32 存成 {v32:.10f}，FP16 {v16:.10f}，BF16 {vb:.10f}")

print("能表示的最大数：FP32 {:.3g}，FP16 {:.0f}，BF16
{:.3g}".format(np.finfo(np.float32).max, np.finfo(np.float16).max,
float(bf16(3.38e38))))
with np.errstate(over="ignore"):
    big = np.float16(70000)                                  # 超出 FP16 的范围
print("70000：FP16 →", big, "（溢出），BF16 →", float(bf16(70000)))
print("1e-8：FP16 →", float(np.float16(1e-8)), "（下溢成 0），BF16 →", f"
{float(bf16(1e-8)):.4g}")
print("机器精度（1 和下一个数之间的距离）：FP32 {:.2e}，FP16 {:.2e}，BF16
{:.2e}".format(np.finfo(np.float32).eps, np.finfo(np.float16).eps, 2.0 ** -7))

# 累加的误差：把 0.01 加 10000 次（正确答案 100）
s16 = np.float16(0)
for _ in range(10000): s16 = np.float16(s16 + np.float16(0.01))
s32 = np.float32(0)
for _ in range(10000): s32 = np.float32(s32 + np.float32(0.01))
print(f"0.01 加 10000 次：FP16 得 {float(s16)}，FP32 得 {float(s32):.4f}")

# 混合精度为什么要 FP32 的主权重：权重 1.0，每步加一个很小的更新 0.0001，加 1000 次（应该是
1.1）
wb, w32 = bf16(1.0), np.float32(1.0)
for _ in range(1000):
    wb = bf16(wb + np.float32(0.0001)); w32 = np.float32(w32 + np.float32(0.0001))
print(f"权重 1.0 加 1000 次 0.0001：BF16 存 → {float(wb)}，FP32 存 →
{float(w32):.4f}")
```

输出：

```
0.1 的 FP32 位：  0 01111011 10011001100110011001101 （符号 | 8 位指数 | 23 位尾数）
0.1 的 BF16 位：  0 01111011 1001101 （只留 7 位尾数）
0.1：FP32 存成 0.1000000015，FP16 0.0999755859，BF16 0.1000976562
3.14159265：FP32 存成 3.1415927410，FP16 3.1406250000，BF16 3.1406250000
能表示的最大数：FP32 3.4e+38，FP16 65504，BF16 3.38e+38
70000：FP16 → inf （溢出），BF16 → 70144.0
1e-8：FP16 → 0.0 （下溢成 0），BF16 → 1.001e-08
机器精度（1 和下一个数之间的距离）：FP32 1.19e-07，FP16 9.77e-04，BF16 7.81e-03
0.01 加 10000 次：FP16 得 32.0，FP32 得 100.0030
权重 1.0 加 1000 次 0.0001：BF16 存 → 1.0，FP32 存 → 1.1000
```

前两行是 0.1 的比特：BF16 只保留了 FP32 的前 16 位（尾数从 23 位截成 7 位）。接着是 0.1 和 π 存下的值。然后是范围和精度：70000 在 FP16 溢出，1e-8 在 FP16 下溢。最后两行是累积误差：FP16 的累加停在 32，BF16 的权重一直是 1.0。

容易混淆的地方

1. **“精度”指的是有效数字，不是小数点后几位**。浮点数的相对误差是固定的，大数的绝对误差大，小数的绝对误差小：FP16 在 1 附近的间隔是 0.001，在 1000 附近是 0.5。
2. **BF16 不是“更好的 FP16”**。它用精度换范围。对深度学习来说范围更重要（不溢出），所以训练多用 BF16；但需要较高精度的计算，FP16 反而更准。
3. **FP8、FP4 离不开缩放**。它们的范围太小，直接存原始的数会大量溢出或下溢，要先乘一个缩放因子把数放到合适的范围里（第 90、93 讲）。

回顾

浮点数用符号、指数、尾数三部分表示一个数：值 = (−1)^符号 × (1 + 尾数/2^m) × 2^(指数 − bias)。尾数位决定精度，指数位决定范围：FP32 有 23 位尾数、8 位指数；FP16 是 10 位尾数、5 位指数，最大 65504；BF16 是 7 位尾数、8 位指数，范围和 FP32 一样；FP8、FP4 更少。精度不够会带来舍入误差、溢出、下溢和“大数吃小数”：0.01 在 FP16 里加 10000 次停在 32；权重 1.0 每步加 0.0001，用 BF16 存永远不变。所以训练要用 FP32 的主权重做混合精度，而推理可以用 BF16、FP8 甚至 4 位，换来更少的显存和更快的速度。

精度决定了每次运算“算多准”和“多快”。另一个问题是“算多少”：一次矩阵乘法、一层网络、生成一个 token，到底要做多少次运算？下一讲：**计算量与 FLOPs**。

第11讲 计算量与 FLOPs

重点

03 矩阵乘法　05 张量与形状

第 3 讲算完 TinyLM 的 4 × 4 矩阵乘法，数了一下：64 次乘加。这个“数一数要算多少次”的习惯，是整个 AI Infra 的起点。一个模型生成一个词要算多少次？训练一个模型要算多少次？一张 GPU 每秒能算多少次？有了这几个数，就能估出时间、估出要几张卡、估出要花多少钱。

这一讲讲怎么数：计算量的单位 FLOPs，矩阵乘法的 2mnk 公式，以及把它推广到整个模型的 2N 和 6N。

FLOPs 和 FLOPS

FLOP（floating-point operation）是一次浮点运算：一次加法或一次乘法。一个任务要做多少次浮点运算，叫它的**计算量**，单位写作 **FLOPs**（复数，注意小写 s）。

硬件每秒能做多少次浮点运算，叫**算力**，单位写作 **FLOPS**（FLOP per second，大写 S）。常用的量级：

- GFLOP：10^9 （十亿）；TFLOP：10^{12}（万亿）；PFLOP：10^{15} 。
- H100 的 BF16 算力约 989 TFLOPS（第 60 讲），即每秒 989 万亿次。

两者相除就是理想情况下的时间：

$$\text{时间} \approx \text{计算量（FLOPs）} \div \text{算力（FLOPS）}$$

“理想”是因为硬件很少能满负荷工作，第 62 讲会讲为什么、能达到多少。

矩阵乘法：2mnk

第 3 讲说过，[m, k] @ [k, n] 的结果有 m × n 格，每一格是一个 k 维点积：k 次乘法，再把 k 个乘积加起来。

代码老老实实地数了一遍 TinyLM 的 [4, 4] @ [4, 4]：乘法 64 次，加法 48 次（每格 k 个乘积只需要加 k − 1 次），合计 112 次。

实际估算时，通常把**一次乘法加一次加法（一次乘加）算 2 次运算**，于是

$$[m, k] @ [k, n] \text{ 的计算量} \approx 2 \cdot m \cdot n \cdot k \text{ FLOPs}$$

TinyLM 这一例是 $2 \times 4 \times 4 \times 4 = 128$。和精确的 112 有一点差别，那是因为每格少了一次加法；k 很大时（真实模型里是几千），这点差别可以忽略。而且硬件做乘加时通常就是“乘完加到累加器上”，累加器从 0 开始，确实是 k 次加法。

第 13 讲的线性层 [4, 4] @ [4, 8]，按 2mnk 是 256 次，再加上偏置的 $4 \times 8 = 32$ 次加法。偏置、激活函数这类逐元素运算，计算量和矩阵乘法相比小得多，估算时常常忽略。

下面的计算器可以输入矩阵的尺寸，算出 FLOPs，以及权重占的字节和“每字节做多少次运算”（第 62 讲会用到后者）。

token 数 s

1

输入维度 k

⇆ 对照

4096

输出维度 n

4096

权重精度

BF16 · 2 字节

预设 逐词生成 s = 1 | 读完长提示 s = 1000 | TinyLM 手算例

3.4×10^7
FLOPs = 2 · s · k · n

32 MiB
权重 = k · n · 字节数

1
每读 1 字节权重的 FLOPs

只计读权重；激活和 KV cache 占的显存在第 71 讲展开

要点

计算量的单位是 FLOPs（运算次数），算力的单位是 FLOPS（每秒运算次数），两者相除是理想的时间。矩阵乘法 [m, k] @ [k, n] 约 2mnk FLOPs：每个结果格子做 k 次乘加，每次乘加算 2 次运算。

推广到整个模型：2N

大模型的计算几乎全是矩阵乘法：一个 token 的向量 [1, d] 乘以一个 [d, d_{out}] 的权重矩阵，就是 $2 \times d \times d_{out}$ 次运算。而 $d \times d_{out}$ 正好是这个权重矩阵的参数个数。所以：

> 一个 token 经过一个权重矩阵 ≈ 2 × 这个矩阵的参数量

把模型里所有的权重矩阵加起来：**一个 token 前向走一遍，约 2N 次运算**，N 是参数量（嵌入表除外，查表不是乘法）。

Llama-3-8B 有 80.3 亿参数，去掉 5.3 亿的嵌入表，是 75.0 亿，所以每个 token 约 **15.0 GFLOP**。第 35 讲会按层、按矩阵细算一遍，结果一致。

这里还没算注意力里 q 和 k、权重和 v 的乘法：它们和前面的 token 数（上下文）成正比，每个 token 约 4 × 层数 × d × 上下文。上下文 2048 时，每个 token 多 1.07 GFLOP，占 7%。上下文越长，这部分越大（第 42、44 讲）。

从计算量到时间

有了计算量，就能估时间。H100 按 989 TFLOPS：

- **处理一段 1000 个 token 的提示**：15.0 TFLOP，满算力约 **15.2 ms**。
- **生成 1 个 token**：15.0 GFLOP，满算力只要 **15.2 μs**。

可实际上，生成一个 token 在 H100 上要好几毫秒，比这个估算慢几百倍。原因不在计算，而在“把 16 GB 的权重从显存读一遍”（第 61、68 讲）。计算量只是故事的一半，另一半是数据搬运。

训练：6N

训练时每个 token 不只要前向，还要反向传播（第 18 讲）算梯度。反向的计算量约是前向的 2 倍：对每个权重矩阵，既要算对输入的梯度，又要算对权重的梯度，各相当于一次前向的矩阵乘法。所以训练每个 token 约 **6N**。

训练整个模型的计算量 ≈ 6 × N × 训练的 token 数。Llama 3 的 8B 模型用了约 15 万亿个 token 预训练：$6 \times 8.03 \times 10^{9} \times 15 \times 10^{12} \approx \mathbf{7.2 \times 10^{23}}$ FLOP。如果用 1000 张 H100、实际只用到 40% 的算力，大约要 **21 天**。这个 6NT 公式是第 46 讲 Scaling Laws 估算训练成本的

基础。

到了大模型里

估算的套路。看到一个模型，先数参数量 N：推理每个 token 约 2N，训练约 6N。再除以硬件的算力和利用率，就得到时间的量级。这个粗算在本站后面会用无数次。

稀疏的模型要看激活参数。MoE 模型（第 41 讲）每个 token 只用到一部分专家。DeepSeek-V3 总共 6710 亿参数，每个 token 只激活约 370 亿，计算量按 370 亿算。

计算量不等于时间。同样的 FLOPs，矩阵大的跑得快，矩阵小、数据搬运多的跑得慢。第 62 讲的 Roofline 会把“算多少”和“搬多少”放在一起考虑。

用代码复现

下面的代码逐次数出两个小矩阵乘法的乘法和加法次数，和 2mnk 对比；算 Llama-3-8B 每个 token 的计算量和注意力的额外部分；换算成 H100 上的理想时间；最后估算训练 8B 模型的总计算量和天数。

```python
# 1. 数一数：矩阵乘法 [m, k] @ [k, n] 到底做了多少次乘法、多少次加法
def count(m, k, n):
    mul = add = 0
    for i in range(m):
        for j in range(n):
            acc = None
            for t in range(k):
                mul += 1                                    # 一次乘法
                if acc is None: acc = 1
                else: add += 1                              # 累加：k 个乘积要加 k - 1
次
    return mul, add
mul, add = count(4, 4, 4)
print(f"TinyLM 的 [4, 4] @ [4, 4]：乘法 {mul} 次，加法 {add} 次，合计 {mul + add}；按
2mnk 估算 {2 * 4 * 4 * 4}")
mul, add = count(4, 4, 8)
print(f"第 13 讲的线性层 [4, 4] @ [4, 8]：{mul + add} 次，2mnk = {2 * 4 * 4 * 8}；再加
偏置 4 × 8 = 32 次")

# 2. Llama-3-8B：每个 token 过一遍模型，约 2 × 参数量 次运算（不算嵌入表，查表不算乘法）
N, N_emb, L, d = 8.03e9, 128256 * 4096, 32, 4096
per_tok = 2 * (N - N_emb)
print(f"Llama-3-8B：参数 {N / 1e9:.2f}B，去掉嵌入表 {(N - N_emb) / 1e9:.2f}B → 每个
token 约 {per_tok / 1e9:.1f} GFLOP")
ctx = 2048
attn = 4 * L * d * ctx                                      # 注意力：和前面 ctx 个
token 算打分、加权求和
print(f"上下文 {ctx} 时，注意力每个 token 再加 {attn / 1e9:.2f} GFLOP（占 {attn /
(per_tok + attn):.0%}）")

# 3. 换算成时间：FLOPs（运算次数）÷ FLOPS（每秒运算次数）
F = 989e12                                                  # H100 的 BF16 稠密算力
print(f"prefill 1000 个 token：{1000 * per_tok / 1e12:.1f} TFLOP → H100 满算力 {1000
* per_tok / F * 1e3:.1f} ms")
print(f"生成 1 个 token：{per_tok / 1e9:.1f} GFLOP → 满算力只要 {per_tok / F *
1e6:.1f} μs（实际慢得多，第 68 讲会讲原因）")

# 4. 训练：每个 token 约 6N（前向 2N + 反向 4N）
T = 15e12                                                   # Llama 3 的预训练用了约 15
万亿个 token
train = 6 * N * T
print(f"训练 8B 模型、{T / 1e12:.0f}T token：6NT = {train:.1e} FLOP；1000 张 H100、按
40% 利用率约 {train / (1000 * F * 0.4) / 86400:.0f} 天")
```

输出：

```
TinyLM 的 [4, 4] @ [4, 4]：乘法 64 次，加法 48 次，合计 112；按 2mnk 估算 128
第 13 讲的线性层 [4, 4] @ [4, 8]：224 次，2mnk = 256；再加偏置 4 × 8 = 32 次
Llama-3-8B：参数 8.03B，去掉嵌入表 7.50B → 每个 token 约 15.0 GFLOP
上下文 2048 时，注意力每个 token 再加 1.07 GFLOP（占 7%）
prefill 1000 个 token：15.0 TFLOP → H100 满算力 15.2 ms
生成 1 个 token：15.0 GFLOP → 满算力只要 15.2 μs（实际慢得多，第 68 讲会讲原因）
训练 8B 模型、15T token：6NT = 7.2e+23 FLOP；1000 张 H100、按 40% 利用率约 21 天
```

前两行是逐次数出来的结果和 2mnk 的对比（112 对 128，224 对 256）。接着是 Llama-3-8B 每个 token 15.0 GFLOP，注意力在上下文 2048 时再加 1.07 GFLOP。然后是时间：prefill 1000 个 token 15.2 ms，生成 1 个 token 15.2 μs。最后是训练：7.2×10^{23} FLOP，约 21 天。

容易混淆的地方

1. **FLOPs 和 FLOPS**。前者是计算量（多少次），后者是算力（每秒多少次）。有的资料写作 FLOP 和 FLOP/s 以示区分。
2. **一次乘加算 1 次还是 2 次**。有的规格表和论文按乘加（MAC）计数，数字是按 FLOPs 计的一半。比较时要看清楚。本站一律按 2 次计。
3. **参数量 ≠ 计算量**。参数量是存多少个数，计算量是算多少次。一个 token 的计算量约是参数量的 2 倍；处理 s 个 token，就是 $2N \times s$。

回顾

计算量用 FLOPs 计（运算次数），算力用 FLOPS 计（每秒运算次数），两者相除是理想时间。矩阵乘法 [m, k] @ [k, n] 约 2mnk FLOPs：TinyLM 的 4×4 例子是 128（逐次数是 112）。一个 token 经过一个权重矩阵约是这个矩阵参数量的 2 倍，所以整个模型前向约 2N 每 token：Llama-3-8B 约 15.0 GFLOP，注意力在上下文 2048 时再加 7%。训练每 token 约 6N，训练 8B 模型、15 万亿 token 约 7.2×10^{23} FLOP。按 H100 的算力，生成一个 token 只要 15 微秒，实际却慢得多，因为还要搬数据。

第 1 篇的数学工具到这里就齐了：向量、矩阵、张量、softmax、概率、导数、旋转、浮点数、计算量。下一篇开始讲神经网络：从最简单的一个“神经元”说起，它能做什么、不能做什么？下一讲：**感知机**。

AI INFRA · HANDBOOK

第 二 篇

神经网络基础

收录 9 讲 · 第 12–20 讲

第12讲 感知机

01 向量　02 点积与余弦相似度

又回到第 1 讲的水果店。这一次你做了一份顾客调查，7 种水果各自得到了“好吃”或“不好吃”的评价。下周要进一种新水果，你只知道它的甜度和酸度，想让电脑先猜一猜：顾客会觉得它好吃吗？

答案只有两种：是，或者否。这类问题叫**二分类**（binary classification）。这一讲要介绍的**感知机**（perceptron），就是专门回答这种问题的一台小机器，也是最早的神经网络。它只做两件事：给输入打一个分，再看分数是不是大于 0。我们会看到，它能学会很多判断，但也有一类非常简单的问题，它无论如何都学不会。正是这个“学不会”，引出了后面所有的多层神经网络。

只看甜度够不够

先把调查结果列出来。甜度和酸度还是 0 到 10 分：

水果	甜度	酸度	顾客评价
苹果	7	4	好吃
香蕉	8	1	好吃
芒果	9	3	好吃
梨	5	2	好吃
柠檬	2	9	不好吃
葡萄柚	4	7	不好吃
菠萝	7	7	不好吃（太酸）

最朴素的想法是：越甜越好吃。定一个门槛，甜度超过 5 就算好吃。试一下：苹果 7、香蕉 8、芒果 9 都超过了 5，判对了；柠檬 2、葡萄柚 4 没超过，也判对了。问题出在两个地方。梨的甜度正好是 5，没有“超过”，被判成了不好吃；菠萝的甜度是 7，被判成了好吃，可顾客嫌它太酸。

梨的问题可以靠调门槛解决，比如改成“超过 4.5”。菠萝的问题却怎么调门槛都没用：苹果和菠萝的甜度都是 7，一个好吃、一个不好吃。只看甜度，这两种水果在电脑眼里一模一样，不可能一个判“是”、一个判“否”。所以必须把酸度也考虑进来。

给每个特征一个权重

怎么同时考虑两个特征？最简单的办法是给它们各配一个系数，加起来打一个总分：

$$z = w_1 \times \text{甜度} + w_2 \times \text{酸度} + b$$

这里的 w_1、w_2 叫**权重**（weight），表示每个特征有多重要、往哪个方向起作用：权重是正数，这个特征越大总分越高；是负数，这个特征越大总分越低。b 叫**偏置**（bias），它和输入无关，相当于在总分上统一加减一个数，作用就是刚才的“门槛”。

总分 z 算出来之后，规则很简单：**z > 0 就输出 1（好吃），否则输出 0（不好吃）**。这种“过了 0 就跳到 1，没过就是 0”的函数叫**阶跃函数**（step function），因为它的图像像一级台阶。

刚才“甜度超过 5”的想法，用这个式子写出来就是 $w_1 = 1$、$w_2 = 0$、$b = -5$：z = 甜度 − 5，酸度的权重是 0，完全不起作用。现在我们换一组：$w_1 = 1$、$w_2 = -1$、$b = -1$，意思是“甜度加分，酸度扣分，再扣 1 分”。逐个算一下：

水果	z = 甜度 − 酸度 − 1	输出	评价
苹果	7 − 4 − 1 = 2	1	好吃 √
香蕉	8 − 1 − 1 = 6	1	好吃 √
芒果	9 − 3 − 1 = 5	1	好吃 √
梨	5 − 2 − 1 = 2	1	好吃 √
柠檬	2 − 9 − 1 = −8	0	不好吃 √
葡萄柚	4 − 7 − 1 = −4	0	不好吃 √
菠萝	7 − 7 − 1 = −1	0	不好吃 √

7 个全对。菠萝虽然甜，但酸度把分扣了回去，总分是 −1，判成了不好吃。

这个打分式你其实已经见过。把权重排成向量 $\mathbf{w} = (w_1, w_2)$，把水果写成向量 $\mathbf{x} = (\text{甜度}, \text{酸度})$，那么 $w_1x_1 + w_2x_2$ 就是第 2 讲的点积 $\mathbf{w} \cdot \mathbf{x}$。所以感知机可以写成一行：

$$z = \boldsymbol{w} \cdot \boldsymbol{x} + b, \qquad \text{输出} = 1\ (z > 0)\ \text{或}\ 0\ (z \leqslant 0)$$

一个感知机，就是“一次点积，加一个数，再判断正负”。特征再多也是一样：有 n 个特征，w 和 x 就是 n 维向量，式子的样子不变。

要点

感知机 = 加权求和（点积）+ 偏置 + 阶跃函数。权重和偏置决定了它怎么判断。

打分为 0 的地方是一条线

只看表格，很难看出这组权重为什么好用。把水果画回第 1 讲的坐标系里看就清楚了。

平面上的每个点都能算出一个 z。z > 0 的点输出 1，z ⩽ 0 的点输出 0，两块区域的交界处就是 z = 0 的那些点：

$$w_1 x_1 + w_2 x_2 + b = 0$$

这是一个二元一次方程，它在平面上画出来是一条直线，叫**分界线**（decision boundary），也叫决策边界。感知机做的事，用几何的话说就是：**用一条直线把平面切成两半，一半说“是”，一半说“否”**。

对于 $w_1 = 1$、$w_2 = -1$、$b = -1$，分界线是“甜度 − 酸度 − 1 = 0”，也就是“酸度 = 甜度 − 1”。它从 (1, 0) 斜着向右上方穿过 (8, 7)。好吃的水果都在线的右下方，不好吃的都在左上方。

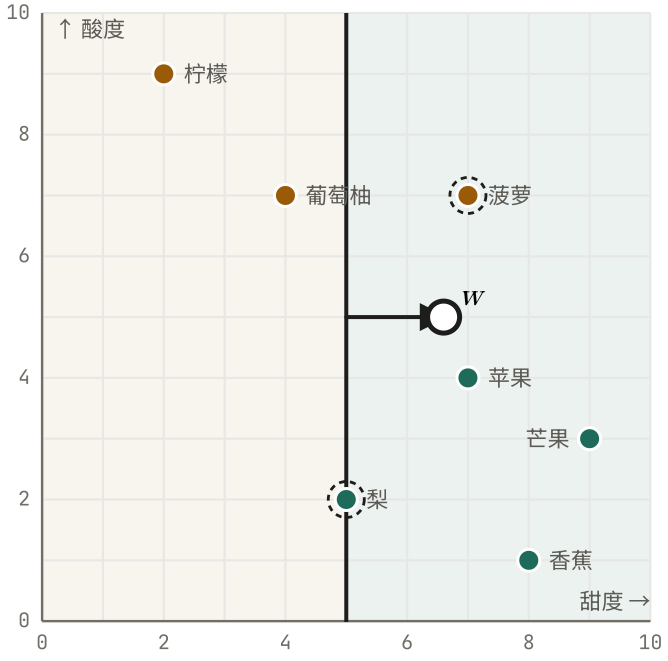
权重向量 $\mathbf{w}$ 和这条线有一个很漂亮的关系。在线上任取两个点 $\mathbf{p}$ 和 $\mathbf{q}$，它们都满足 $\mathbf{w} \cdot \mathbf{p} + b = 0$ 和 $\mathbf{w} \cdot \mathbf{q} + b = 0$，两式相减得到 $\mathbf{w} \cdot (\mathbf{p} - \mathbf{q}) = 0$。$\mathbf{p} - \mathbf{q}$ 是沿着这条线的方向，而第 2 讲说过，点积为 0 表示两个向量垂直。所以：**$\mathbf{w}$ 垂直于分界线**。

再看 $\mathbf{w}$ 指向哪一边。从线上一点出发，沿 $\mathbf{w}$ 的方向走 t 步，z 会从 0 变成 $t \times |\mathbf{w}|^2$，是正的。所以 $\mathbf{w}$ 指向输出 1 的那一侧。至于 b，它不改变线的方向，只把线平行地推来推去。

下面的演示里，浅绿色的一侧输出 1（好吃），浅琥珀色的一侧输出 0（不好吃）；绿色的点是顾客说好吃的，琥珀色的点是顾客说不好吃的。黑色的直线就是分界线，黑色箭头是 $\mathbf{w}$。你可以直接拖动这条线（平移，改的是 b），拖动箭头末端的圆圈（转向和伸缩，改的是 $\mathbf{w}$），也可以用右边的滑块精确地调三个数。分错的点会套上一个虚线圈，表格里对应的一行也会标出来。

拖动分界线，把两类点分开

⇆ 对照



w_1 = **1**

w_2 = **0**

b = **-5**

打分	`z = 1·甜度 + 0·酸度 + (-5)`
分界线	`1·甜度 + 0·酸度 + (-5) = 0`

水果	z	输出	应为
苹果 (7, 4)	2	好吃	好吃 ✓
香蕉 (8, 1)	3	好吃	好吃 ✓
芒果 (9, 3)	4	好吃	好吃 ✓
梨 (5, 2)	**0**	**不好吃**	**好吃 ✗**
柠檬 (2, 9)	-3	不好吃	不好吃 ✓
葡萄柚 (4, 7)	-1	不好吃	不好吃 ✓
菠萝 (7, 7)	**2**	**好吃**	**不好吃 ✗**

分对 **5 / 7**。带圈的点分错了，试着调整分界线

图 12-1　感知机的分界线。默认是“甜度超过 5 就算好吃”（w = (1, 0)，b = −5）：梨（z = 0）和菠萝（z = 2）分错了。试着把它调到 w = (1, −1)，b = −1。

刚打开时，分界线是竖直的“甜度 = 5”：w = (1, 0) 指向右边，酸度完全不管。梨正好压在线上，z = 0，输出 0；菠萝在线的右边，z = 2，输出 1，两个都错了。把 w_2 往负的方向拉，线开始倾斜，菠萝被划到了左上方；再把 b 调到 −1，梨也回到了右下方，7 个点全部分对。

多拖几次你会发现，能把 7 个点全部分对的线不止一条。只要线从菠萝和苹果、梨和葡萄柚之间的空隙穿过去，怎么斜都行。感知机只关心“分没分对”，不关心“分得好不好”。

让机器自己找这条线

7 个点、2 个特征，用手拖还行。可如果有 1000 种水果、每种有 50 个特征，就不可能靠人去调了。感知机真正有意思的地方在于，它有一套**自己调权重的办法**，叫感知机学习规则。

规则是这样的。把样本一个一个地拿给感知机看，每看一个，先用当前的权重算出输出，再和正确答案 y 比较：

- 如果输出对了，什么都不改。
- 如果错了，就按下面的式子改：

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + (y - \text{输出}) \cdot \boldsymbol{x}, \qquad b \leftarrow b + (y - \text{输出})$$

“←”的意思是“用右边的值替换左边”。错的时候 y − 输出只可能是 +1 或 −1：应该输出 1 却输出了 0，就是 +1，把 x 加到 w 上；应该输出 0 却输出了 1，就是 −1，从 w 里减掉 x。

为什么这样改是对的？看第一种情况：应该输出 1，可是 z ⩽ 0，分数太低了。改完之后，同一个样本的分数变成

$$(\mathbf{w} + \mathbf{x}) \cdot \mathbf{x} + (b + 1) = z + |\mathbf{x}|^2 + 1$$

$|\mathbf{x}|^2$ 是一个数的平方和，不会是负的，所以 z 至少涨了 1。也就是说，**每改一次，这个分错的样本都会往“正确的那一边”挪一点**。一次可能挪不过去，下次再碰到它时还会接着挪。第二种情况正好反过来，分数至少降 1。

这个规则只看一个样本就动手改，改的时候可能会把别的样本弄错。所以要把所有样本反复过好几遍，每过一遍叫一**轮**（epoch）。直到某一轮一个都没错，就停下来。

用“与”手算一遍

为了每一步都能用心算跟上，换一个只有 4 个样本的小问题：逻辑里的**与**（AND）。输入是两个 0 或 1，两个都是 1 时输出 1，否则输出 0：

x_1	x_2	与
0	0	0
0	1	0
1	0	0
1	1	1

从 w = (0, 0)、$b = 0$ 开始，按表里的顺序看样本。

第 1 轮。此时不管输入是什么，z 都是 0，输出都是 0。前三个样本的答案也是 0，都对了。第四个 (1, 1) 应该输出 1，错了，y − 输出 = 1：$\mathbf{w} \leftarrow (0,0) + (1,1) = (1,1)$，$b \leftarrow 0 + 1 = 1$。这一轮错了 1 次。

第 2 轮。

1. (0, 0)：$z = 1\times0 + 1\times0 + 1 = 1$，输出 1，应该是 0，错了。y − 输出 = −1：$\mathbf{w} \leftarrow (1,1) - (0,0) = (1,1)$，$b \leftarrow 1 - 1 = 0$。
2. (0, 1)：$z = 1\times0 + 1\times1 + 0 = 1$，输出 1，又错了：$\mathbf{w} \leftarrow (1,1) - (0,1) = (1,0)$，$b \leftarrow 0 - 1 = -1$。
3. (1, 0)：$z = 1\times1 + 0\times0 - 1 = 0$，输出 0，对了，不改。
4. (1, 1)：$z = 1\times1 + 0\times1 - 1 = 0$，输出 0，应该是 1，错了：$\mathbf{w} \leftarrow (1,0) + (1,1) = (2,1)$，$b \leftarrow -1 + 1 = 0$。

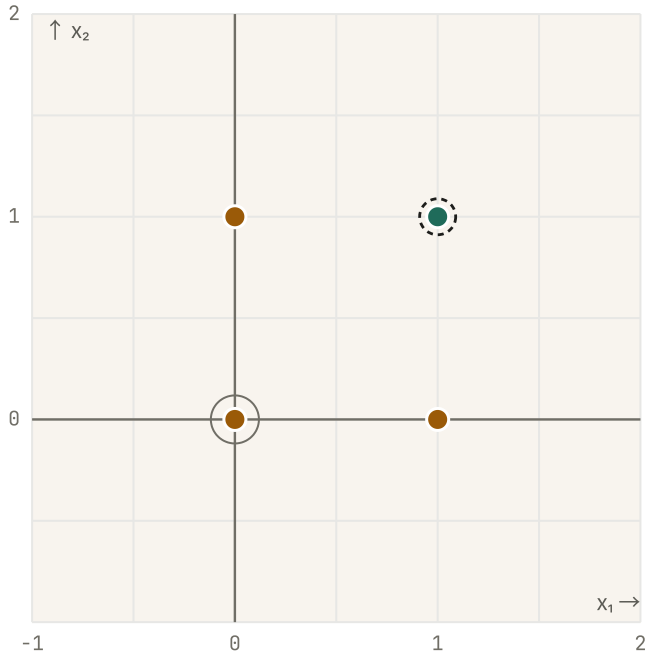
这一轮错了 3 次。错得比第 1 轮还多，看上去像是在倒退，其实权重正在一点点往对的方向挪。接着往下算，第 5 轮结束时权重变成 $\mathbf{w} = (2,1)$、$b = -2$，第 6 轮一个都没错，训练结束。验算一下：只有 (1, 1) 的分数 $2 + 1 - 2 = 1$ 是正的，其余三个分别是 −2、−1、0，都输出 0，正好就是“与”。

剩下的几轮交给下面的演示。点“下一个样本”，每次处理一个样本，下面会写出这一步的 z、输出和怎么改；点“跑完这一轮”一次走完一轮。开始时 $\mathbf{w} = (0,0)$，没有分界线，整个平面都输出 0；随着训练，你会看到分界线出现、转动、平移，最后停在把 (1, 1) 单独隔开的位置。上面的按钮可以切换到“或”（OR：至少有一个 1 就输出 1），它第 4 轮就停下了。

让感知机自己找分界线

⇆ 对照

与　或



$w_1 = 0$

$w_2 = 0$

$b = 0$

打分　z = 0·x₁ + 0·x₂ + 0

分界线　w = (0, 0)，还没有分界线：z 恒等于 b

输入	z	输出	应为	
(0, 0)	0	0	0	✓
(0, 1)	0	0	0	✓
(1, 0)	0	0	0	✓
(1, 1)	**0**	**0**	**1**	✗

分对 **3 / 4**。带圈的点分错了，试着调整分界线

下一个样本　跑完这一轮　▶ 自动训练　重置

点“下一个样本”，从 w = (0, 0)、b = 0 开始训练，看感知机怎么根据一个样本改权重。

下一个：第 1 轮，第 1 个样本 (0, 0)

图 12-2　感知机学习规则。每一步只看一个样本：对了不改，错了就把 x 加到 w 上（或减掉）。“与”在第 6 轮全部分对。

它一定会停下来吗

只要存在一条直线能把两类点完全分开，感知机学习规则就一定会在有限步之内找到一条这样的线，这叫感知机收敛定理（1962 年由 Novikoff 证明）。这样的数据叫**线性可分**（linearly separable）。水果、与、或都是线性可分的。反过来，如果数据不是线性可分的，这个规则就永远停不下来。

异或：怎么拖都分不开

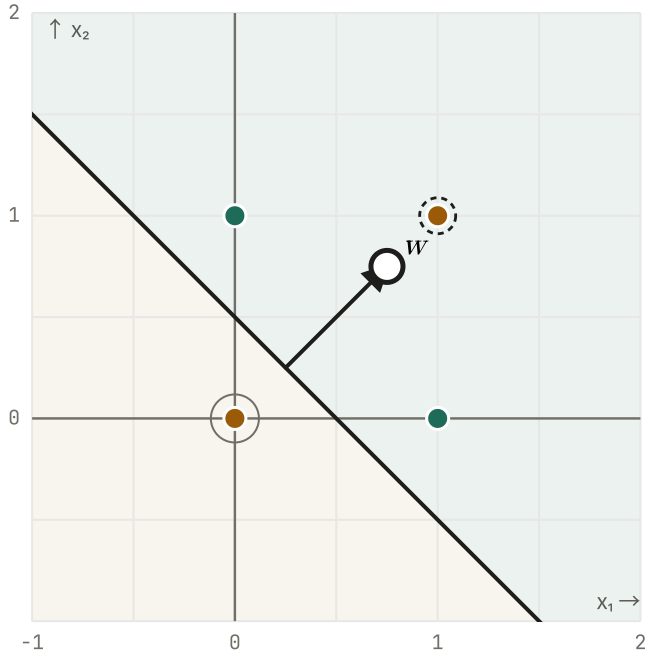
与、或都学会了，再试一个看起来同样简单的：**异或**（XOR）。两个输入不一样时输出 1，一样时输出 0：

x_1	x_2	异或
0	0	0
0	1	1
1	0	1
1	1	0

画在平面上，输出 1 的 (0, 1) 和 (1, 0) 在一条对角线上，输出 0 的 (0, 0) 和 (1, 1) 在另一条对角线上，两类点交叉着排。在下面的演示里拖一拖，试着用一条直线把绿点和琥珀色的点分开。

试着用一条直线分开异或的 4 个点

⇆ 对照



w_1 = **1**

w_2 = **1**

b = **-0.5**

打分 `z = 1·x₁ + 1·x₂ + (-0.5)`

分界线 `1·x₁ + 1·x₂ + (-0.5) = 0`

输入	z	输出	应为	
(0, 0)	-0.5	0	0	✓
(0, 1)	0.5	1	1	✓
(1, 0)	0.5	1	1	✓
(1, 1)	**1.5**	**1**	**0**	✗

分对 **3 / 4**。异或：不管怎么调，一条直线最多只能分对 3 个

下一个样本　跑完这一轮　▶ 自动训练　重置

点“下一个样本”，从 w = (0, 0)、b = 0 开始训练，看感知机怎么根据一个样本改权重。

下一个：第 1 轮，第 1 个样本 (0, 0)

图 12-3　异或。不管怎么拖，最多只能分对 3 个。点“自动训练”，学习规则也找不到答案：从第 3 轮开始，每一轮 4 个样本全错，一轮结束又回到 w = (−1, 0)，b = 1，原地打转。

你会发现，怎么拖都只能分对 3 个，总有一个点在错误的一边。这不是你不够耐心，而是这样的直线根本不存在。用不等式就能证明。假设有一组 w_1、w_2、b 能把 4 个点都分对，那么它必须同时满足：

输入	要求的输出	对 z 的要求
(0, 0)	0	$b \leqslant 0$
(0, 1)	1	$w_2 + b > 0$
(1, 0)	1	$w_1 + b > 0$
(1, 1)	0	$w_1 + w_2 + b \leqslant 0$

把中间两行加起来，得到 $w_1 + w_2 + 2b > 0$，也就是 $w_1 + w_2 + b > -b$。第一行说 $b \leqslant 0$，所以 $-b \geqslant 0$，于是 $w_1 + w_2 + b > 0$。这和最后一行正好矛盾。所以，**没有任何一组权重和偏置能让一个感知机算出异或**。

在演示里点“自动训练”，可以看到学习规则在这里是什么表现。它不会报错，也不会停，只是一直改下去：从第 3 轮开始，每一轮 4 个样本全错，一轮结束时权重又回到 $\mathrm{w} = (-1, 0)$、$b = 1$，在原地转圈。演示在第 20 轮时替它停了下来。

这就是一个感知机的极限：它只能画一条直线，所以只能解决线性可分的问题。异或只有 4 个点，却已经超出了它的能力。

两条线拼起来

一条直线不够，那用两条呢？观察一下异或的四个点：输出 1 的两个点，夹在 (0, 0) 和 (1, 1) 中间。如果画两条斜线，一条把 (0, 0) 隔在左下，一条把 (1, 1) 隔在右上，中间那条带子里就正好是输出 1 的两个点。

每条线都是一个感知机，再用第三个感知机把它们的结果合起来：

- 第一个感知机 h_1 算“或”：$\mathrm{w} = (1, 1)$，$b = -0.5$。只要不是 (0, 0)，就输出 1。它的分界线把 (0, 0) 隔在左下。
- 第二个感知机 h_2 算“与非”（NAND，与的反面）：$\mathrm{w} = (-1, -1)$，$b = 1.5$。只要不是 (1, 1)，就输出 1。它的分界线把 (1, 1) 隔在右上。
- 第三个感知机拿 (h_1, h_2) 当输入，算“与”：$\mathrm{w} = (1, 1)$，$b = -1.5$。两条线都说“是”，才输出 1。

逐个算一下：

输入	h_1（或）	h_2（与非）	与(h_1, h_2)	异或
(0, 0)	0	1	0	0
(0, 1)	1	1	1	1
(1, 0)	1	1	1	1
(1, 1)	1	0	0	0

最后一列和异或完全一致。第一步的两个感知机并排工作，各自看原始输入；第二步的感知机看的是它们的输出。像这样把感知机排成**层**（layer），一层的输出当作下一层的输入，就是**多层感知机**（multilayer perceptron，MLP）。中间那一层不直接面对输入也不直接给出答案，叫**隐藏层**（hidden layer）。

从几何上看，第一层做的是一次**换坐标**：它把原来的点 (x_1, x_2) 换成了新的点 (h_1, h_2)。在原来的坐标里分不开的四个点，到了新坐标里变成了 $(0, 1)$、$(1, 1)$、$(1, 1)$、$(1, 0)$，输出 1 的两个点重合在 $(1, 1)$，一条直线就能把它和另外两个点分开。**多一层，就多一次把数据“摆”成好分的样子的机会**。这个想法会一直贯穿到大模型。

不过这里的权重是我们凭经验摆出来的。学习规则只知道怎么改“直接给出答案”的那一层：隐藏层的输出该是多少，没有标准答案可以对照。再加上阶跃函数是一级台阶，权重稍微动一点，输出要么不变、要么突然跳变，没法告诉我们“往哪边改、改多少会好一点”。这两个问题，要等到第 14 讲换掉阶跃函数、第 18 讲讲反向传播，才能真正解决。

> **感知机的起落**
>
> 感知机是美国心理学家 Frank Rosenblatt 在 1958 年前后提出的，他还造了一台专门的机器 Mark I Perceptron，用一组光电管当输入，去识别简单的图形，在当时引起了很大的轰动。1969 年，Marvin Minsky 和 Seymour Papert 出版了《Perceptrons》一书，系统地分析了单层感知机的局限，异或就是其中最有名的例子。这本书让很多人对神经网络失去了信心，相关研究冷清了十几年。直到 1986 年前后，Rumelhart、Hinton 和 Williams 等人让反向传播广为人知，多层网络可以训练了，神经网络才重新兴起。

用代码复现

下面的代码把学习规则写成一个函数，分别拿水果、与、或、异或去训练，最多跑 20 轮；最后用上一节的三个感知机拼出异或。

```
import numpy as np

def predict(w, b, x):
    return 1 if x @ w + b > 0 else 0            # 点积 + 偏置，再看正负

def train(X, y, max_epoch=20):
    w, b = np.zeros(2), 0.0
    for epoch in range(1, max_epoch + 1):
        errors = 0
        for xi, yi in zip(X, y):
            e = yi - predict(w, b, xi)          # 对了是 0，错了是 +1 或 -1
            if e != 0:
                w, b = w + e * xi, b + e
                errors += 1
        if errors == 0:                         # 这一轮一个都没错
            return w, b, epoch
    return w, b, None                           # 跑满了也没停下来

fruit = np.array([[7, 4], [8, 1], [9, 3], [5, 2], [2, 9], [4, 7], [7, 7]])
liked = [1, 1, 1, 1, 0, 0, 0]
gates = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
tasks = {"水果": (fruit, liked), "与": (gates, [0, 0, 0, 1]),
         "或": (gates, [0, 1, 1, 1]), "异或": (gates, [0, 1, 1, 0])}
for name, (X, y) in tasks.items():
    w, b, epoch = train(X, y)
    print(f"{name}: w = {w.astype(int).tolist()}, b = {int(b)}, "
          f"第 {epoch} 轮全对" if epoch else f"{name}: 跑满 20 轮仍有错",
          [predict(w, b, x) for x in X])

# 两层：或、与非 并排，再用 与 合起来
h1 = [predict(np.array([1, 1]), -0.5, x) for x in gates]    # 或
h2 = [predict(np.array([-1, -1]), 1.5, x) for x in gates]   # 与非
print("两层: ", [predict(np.array([1, 1]), -1.5, np.array(h)) for h in zip(h1, h2)])
```

输出：

```
水果: w = [5, -5], b = 0, 第 2 轮全对 [1, 1, 1, 1, 0, 0, 0]
与: w = [2, 1], b = -2, 第 6 轮全对 [0, 0, 0, 1]
或: w = [1, 1], b = 0, 第 4 轮全对 [0, 1, 1, 1]
异或: 跑满 20 轮仍有错 [1, 1, 0, 0]
两层:  [0, 1, 1, 0]
```

第二行就是我们手算加演示的结果：$\mathrm{w} = (2, 1)$，$b = -2$，第 6 轮全对。水果只用了 2 轮，学到的是 $\mathrm{w} = (5, -5)$、$b = 0$，分界线是“甜度 = 酸度”，和我们手调的“酸度 = 甜度 − 1”不一样，但同样把 7 种水果都分对了（菠萝正好压在线上，$z = 0$，输出 0）。异或跑满 20 轮也没学会，最后的权重只分对了 (0, 1) 和 (1, 1) 这两个。最后一行是两层的结构，输出 [0, 1, 1, 0]，正是异或。

到了大模型里

今天的大模型里已经找不到“输出 0 或 1”的感知机了，但它的核心——**点积加偏置**——到处都是。

以 Llama-3-8B 为例（配置来自官方 config.json：隐藏维度 4096，FFN 隐藏维度 14336，32 层）。每个词进入一层的前馈网络（FFN，第 31 讲）时，是一个 4096 维的向量 $\mathbf{x}$。FFN 的第一步要算出 14336 个数，每个数都是 $\mathbf{x}$ 和一个 4096 维权重向量的点积，也就是一个“没有阶跃函数的感知机”在打分。换句话说，单是这一步，就相当于 14336 个各有 4096 个权重的感知机并排工作，32 层就是 32 组。Llama 还把偏置 b 也省掉了，只留下点积。

和这一讲的感知机相比，变化主要有两处。一是阶跃函数被换成了平滑的激活函数（第 14 讲），这样才能用梯度来训练。二是这些打分不再直接当成“是 / 否”的答案，而是作为新的坐标交给下一层，正像刚才异或的例子里，第一层把点换了一套坐标。至于“一次算 14336 个点积”，下一讲会看到，它恰好就是一次矩阵乘法，这也是为什么推理的大部分时间都花在矩阵乘法上。

容易混淆的地方

1. **“感知机”和“神经元”**。严格说，感知机指的是带阶跃函数、用这一讲的学习规则训练的那种模型。现在说的“神经元”，是把阶跃函数换成其他激活函数后的一般形式。“多层感知机”这个名字是沿用下来的，今天的 MLP 里其实已经没有阶跃函数了。
2. **学不会异或，不是因为训练得不够久**。线性可分的数据，学习规则保证能停；异或不是线性可分的，再跑多少轮、怎么换起点也没用。问题出在模型本身只能画一条直线。
3. **分界线不唯一，权重也不唯一**。把 w 和 b 同时乘以 2，z 处处翻倍，正负不变，分界线一模一样；不同的起点和样本顺序，也会学到不同的线。感知机只保证“分对”，不保证找到“最好”的那条线。

回顾

感知机是最简单的神经网络：把输入和权重做点积，加上偏置，大于 0 就输出 1，否则输出 0。几何上，它用一条直线（高维时是一个平面）把空间切成两半，权重向量 w 垂直于分界线、指向输出 1 的一侧，偏置 b 负责把线平移。它还能自己学：错了就把输入加到权重上或减掉，只要数据线性可分，就一定能在有限步内全部分对。但一条直线有它的极限，连只有 4 个点的异或都分不开；把几个感知机排成两层，先换一次坐标再分，问题就解决了。

这一讲里，我们一次只看一个感知机，算的是一个点积。可是在异或的例子里，第一层已经有两个感知机并排工作；到了大模型里，一层有上万个。很多个感知机并排，同时对同一个输入打分，写成数学是什么样子？阶跃函数去掉之后，这个“可学习的最小单元”又该怎么描述？这就是下一讲的内容：**神经元与线性层**。

第13讲 神经元与线性层

03 矩阵乘法　12 感知机

上一讲的感知机有两个让人不太满意的地方。第一，它只会说“是”或“否”：香蕉和苹果都被判成好吃，可香蕉的分数是 6，苹果只有 2，这个差别在输出里完全看不出来。第二，我们最后靠两层感知机解决了异或，可两层的权重是我们凭经验摆出来的，没法让机器自己学。

这一讲先处理一个最基本的问题：**神经网络里“最小的、可以学习的单元”到底是什么，很多个这样的单元放在一起，又该怎么写？**答案是两个名字：**神经元**和**线性层**。后者是大模型里最常见的部件，Llama-3-8B 的 80 亿个参数里，有将近九成都在线性层里。

把台阶拿掉

先回到上一讲的水果。用 $w = (1, -1)$、$b = -1$ 给苹果 $(7, 4)$ 打分，$z = 7 - 4 - 1 = 2$，经过阶跃函数，输出 1。

现在想象我们在训练这个感知机，想知道“把 w_1 稍微调大一点，结果会怎样”。下面的演示上半部分是这个感知机的结构图：左边是两个输入和一个常数 1，每条连线上写着它的权重（线越粗，权重的绝对值越大；绿色是正，琥珀色是负）；中间的 Σ 把它们加权求和得到 z；右边是输出。下半部分画的是：**只改一个参数、其他都不动时，输出怎么变**。拖哪个滑块，下面就画哪个参数。

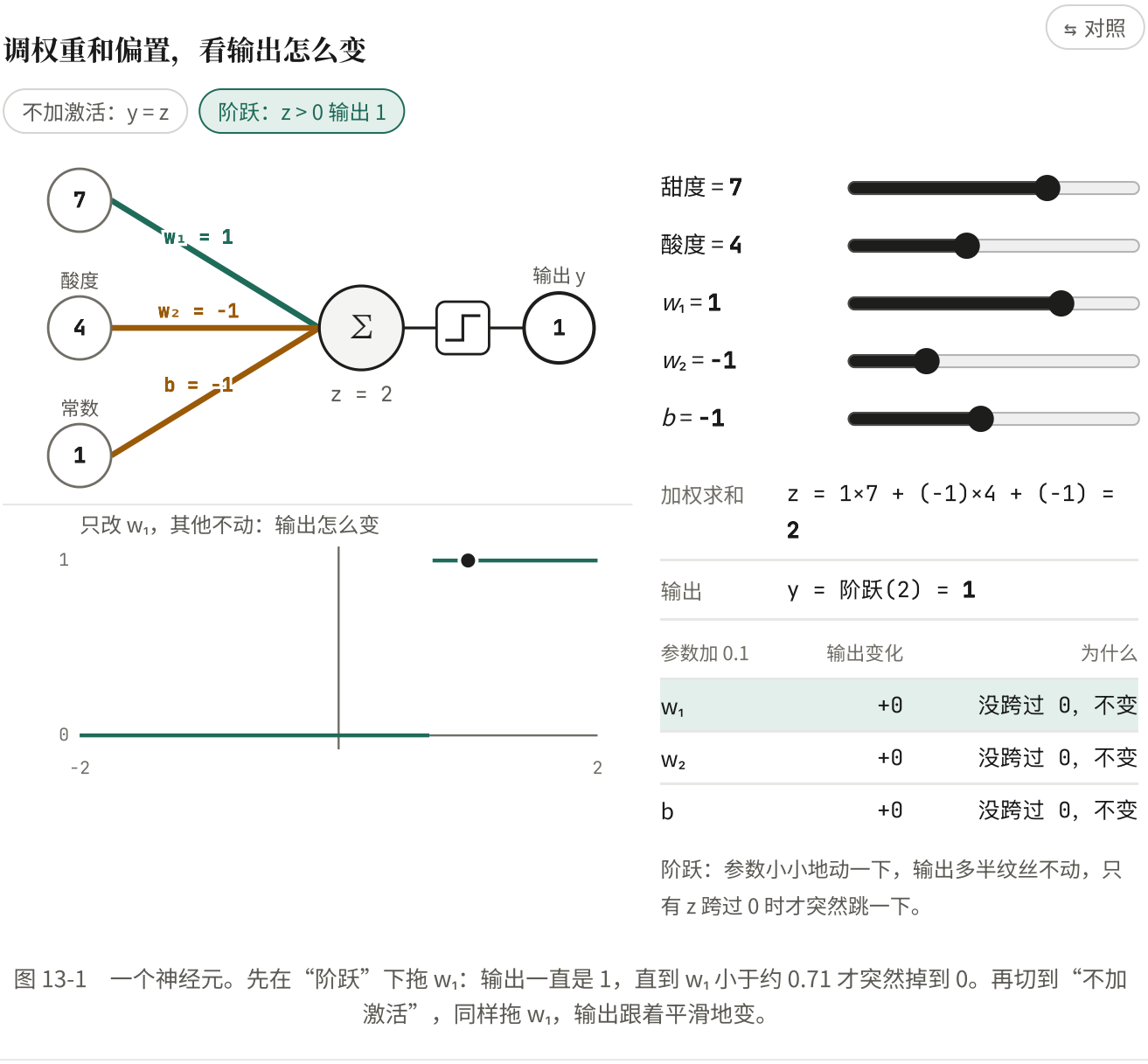


图 13-1　一个神经元。先在“阶跃”下拖 w_1：输出一直是 1，直到 w_1 小于约 0.71 才突然掉到 0。再切到“不加激活”，同样拖 w_1，输出跟着平滑地变。

在“阶跃”模式下拖 w_1，你会看到输出一直是 1，纹丝不动；右边的表格也告诉你，w_1、w_2、b 各加 0.1，输出的变化全是 0。一直往小拖，拖到 w_1 小于 $5/7 \approx 0.71$ 的时候，z 跨过了 0，输出才突然掉到 0。下面的曲线是一级台阶：平的、平的、突然一跳、又是平的。

这对“学习”来说很麻烦。想象你蒙着眼睛站在山坡上找下山的路，唯一的办法是用脚试探四周哪边低。可如果脚下是一块块平台，往哪边挪一小步高度都不变，你就完全不知道该往哪走。阶跃函数就是这样的平台：参数的小改动，在输出上几乎总是没有反应。

现在点上面的“不加激活”，把阶跃函数拿掉，直接让输出 y = z。同样拖 w_1，下面的曲线变成了一条斜线：w_1 每加 0.1，输出就加 0.7。表格里三个参数各加 0.1，输出分别变化 +0.7、+0.4（注意 w_2 是负的，加 0.1 是往 0 靠，扣分变少了）和 +0.1。每一个参数往哪边动、动多少，输出都给出了清清楚楚的回应。

要点

要让机器自己调参数，输出必须对参数的小改动“有反应”。把阶跃函数拿掉，就是迈出的第一步。

神经元：加权求和，再过一个函数

去掉阶跃之后的这个小单元，就是现代神经网络里的**神经元**（neuron）。它做的事分成两步：

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b = \boldsymbol{w}\cdot\boldsymbol{x} + b, \qquad y = f(z)$$

第一步和感知机一模一样：输入 x 和权重 w 做点积，再加偏置 b。第二步把 z 交给一个函数 f，叫**激活函数**（activation function）。感知机的 f 是阶跃函数；现代网络用的是 ReLU、GELU、SiLU 这类平滑（或几乎平滑）的函数，它们为什么非有不可、长什么样，是下一讲的内容。这一讲我们先只看第一步，也就是 $f(z) = z$、什么都不做的情况。

神经元里能被调整的，只有权重 w 和偏置 b，它们合称**参数**（parameter）。输入 x 是外面给的，不能改；训练要做的事，就是不断地调参数。所以说神经元是“最小的可学习单元”：一组参数，加一个固定的计算规则。

刚才表格里的规律值得再看一眼：w_1 加 0.1，输出变化 0.1×7；w_2 加 0.1，输出变化 0.1×4；b 加 0.1，输出变化 0.1×1。从式子上看也很明显，z 里和 w_1 有关的只有 w_1x_1 这一项，w_1 增加 Δ，这一项就增加 $\Delta \times x_1$。换句话说：

> 权重 w_i 对输出的影响有多大，取决于它连着的输入 x_i 有多大

偏置可以看成连着一个永远等于 1 的输入，所以它的影响永远是 1 倍。在演示里把甜度拖大，w_1 那一行的变化量也会跟着变大。这个看似平常的观察，就是第 17、18 讲梯度和反向传播的起点：知道了“每个参数动一点，输出变多少”，才知道该怎么改参数。

好几个神经元一起看同一个输入

一个神经元只给出一个数。可是看一种水果，我们往往想同时知道好几件事：好不好吃、酸不酸、味道浓不浓。那就让几个神经元同时看同一个输入，各给一个分：

神经元 权重 (甜度, 酸度)	偏置	苹果 (7, 4) 的打分
好吃分 (1, −1)	−1	7 − 4 − 1 = 2
酸味分 (−0.5, 1)	0	−3.5 + 4 = 0.5
浓郁分 (0.5, 0.5)	0	3.5 + 2 = 5.5

三个神经元各做一次点积。这个场景我们在第 3 讲见过：同一个向量和好几个向量分别做点积，就是矩阵乘法。把三个神经元的权重**竖着**排成三列，拼成一个 2×3 的矩阵 W；三个偏置排成一个向量 **b**：

$$W = [[1, -0.5, 0.5], [-1, 1, 0.5]], \qquad \boldsymbol{b} = (-1, 0, 0)$$

那么苹果的三个打分一次就能算出来：

$$(7, 4) \cdot W + \mathbf{b} = (7 \times 1 + 4 \times (-1),\ 7 \times (-0.5) + 4 \times 1,\ 7 \times 0.5 + 4 \times 0.5) + (-1, 0, 0)$$

按第 3 讲的“行乘列”，结果的第 j 个数就是 **x** 和 W 第 j 列的点积，也就是第 j 个神经元的打分。所以：

> **要点**
> 把 n 个神经元的权重竖着排成 n 列，它们的打分就是一次矩阵乘法 **x** · W，再加上偏置向量 **b**。W 的每一列就是一个神经元。

这样一组并排工作、看同一个输入的神经元，就叫一个**线性层**（linear layer），写成

$$\mathbf{y} = \mathbf{x} \cdot W + \mathbf{b}$$

它也叫**全连接层**（fully connected layer），因为每个输入都和每个神经元连着一条线。形状要对上：输入 **x** 有 d_{in} 个数，层里有 d_{out} 个神经元，那么 W 是 $d_{\text{in}} \times d_{\text{out}}$（行数等于输入的个数，列数等于神经元的个数），**b** 有 d_{out} 个数，输出 **y** 也有 d_{out} 个数。水果的例子里是 2 进 3 出。

还记得第 4 讲吗？那里说 x · W 是一次线性变换，把向量送到另一个空间；加上 **b** 则是一次平移。所以线性层在几何上就是“先变换、再平移”：把 2 维的水果，送到了一个 3 维的“打分空间”里。

一次喂进去一整批

线性层还有一个很方便的性质：输入不止一个时，不用一个一个地算。把几种水果的向量叠成一个矩阵 X，每一行一种水果，那么 X · W 的每一行，正好就是那种水果的打分（这还是第 3 讲的“行乘列”）。偏置向量 **b** 则加到每一行上：

水果	好吃分	酸味分	浓郁分
苹果 (7, 4)	2	0.5	5.5
香蕉 (8, 1)	6	−3	4.5
柠檬 (2, 9)	−8	8	5.5

一个 3×2 的 X 乘以 2×3 的 W，得到 3×3 的结果，再给每一行加上 **b**。这里有一个小小的“越权”：矩阵 X · W 是 3×3，**b** 只是一个 3 个数的向量，形状不一样，严格说不能直接相加。约定的做法是把 **b** 复制到每一行上再加，这叫**广播**（broadcasting），NumPy 和 PyTorch 都会自动这样做。第 5 讲讲张量与形状时会更系统地说这个规则。

现在换成 TinyLM。一句“我 爱 学 AI”是 4 个词，每个词是 4 维向量，叠成 4×4 的 X（和第 3 讲是同一个 X）。让它经过一个有 8 个神经元的线性层：W 是 4×8，取的是 TinyLM 第 1 层前馈网络（第 31 讲）的第一个权重矩阵；**b** 是 8 个数。结果 Y 是 4×8：每个词得到 8 个打分。

先手算一格：“我”的向量是 (0.7, −0.6, −0.9, 0.3)，W 的第 1 列是 (−0.1, −0.7, −0.3, 0.6)，第 1 个偏置是 0.7：

$$0.7 \times (-0.1) + (-0.6) \times (-0.7) + (-0.9) \times (-0.3) + 0.3 \times 0.6 + 0.7 = -0.07 + 0.42 +$$

下面的步进器把 32 格全部这样算一遍。点结果里的任意一格，会高亮 X 的那一行、W 的那一列和对应的偏置，下面写出完整的算式。X、W、**b** 里的数都可以改，结果会跟着重算。

X · W + b，一格一格地算

对照

X 4×4 · 我 爱 学 AI

	1	2	3	4
我	0.7	-0.6	-0.9	0.3
爱	-0.3	-0.1	-0.8	-0.2
学	0.3	-0.3	0.6	0.6
AI	0.4	0.8	0.4	-0.6

×

W 4×8 · 权重

	1	2	3	4	5	6	7	8
	-0.1	-0.5	0.1	0	-0.1	0.2	0.2	-0.6
	-0.7	0.4	0.9	0.1	-0.1	0.2	-0.9	-0.2
	-0.3	-0.2	0.5	0	0.7	-0.1	-0.9	0.3
	0.6	0.1	0.8	-0.2	0.6	-0.9	-0.9	0.6

+



▶ 播放　下一格　全部算完　重置　0 / 32

点“播放”一格一格地算，或直接点结果 C 里的任意一格。

注意

TinyLM 本身和 Llama 一样，线性层里不带偏置。这里的 **b** 是为了演示单独生成的，不属于 TinyLM 的权重，后面的讲不会用到它。

一次矩阵乘法，同时算出了 4 个词 × 8 个神经元 = 32 个打分。这和把 4 个词一个一个送进去、各算一次 $\mathbf{x} \cdot W + \mathbf{b}$，结果完全一样（下面的代码会验证）。但在 GPU 上，一次算一大块远比分很多次算小块要快，第 3 讲末尾讲算术强度时说过原因。所以大模型的推理，总是尽量把很多词、很多请求叠在一起，一次送进线性层。

数一数参数

一个线性层有多少个参数？W 有 $d_{\text{in}} \times d_{\text{out}}$ 个数，b 有 d_{out} 个数，合起来

$$\text{参数个数} = d_{\text{in}} \times d_{\text{out}} + d_{\text{out}}$$

水果的例子是 $2 \times 3 + 3 = 9$ 个，TinyLM 这一层是 $4 \times 8 + 8 = 40$ 个。可以看出，权重的个数是两个维度相乘，偏置只是加上一个维度。维度一大，偏置在参数里所占的比例就微不足道了，这也是很多大模型干脆不用偏置的原因之一。

顺带一提，偏置其实可以“藏进”矩阵里。给每个输入向量后面补一个 1，变成 $(x_1, \cdots, x_n, 1)$；再把 $\mathbf{b}$ 作为新的一行，接在 W 的最下面。这样，新的输入乘以新的矩阵，最后一行乘的正是那个 1，相当于加上了 $\mathbf{b}$：

$$(\mathbf{x}, 1) \cdot [W; \mathbf{b}] = \mathbf{x} \cdot W + 1 \cdot \mathbf{b}$$

这正是刚才演示里把偏置画成“连着常数 1 的一条线”的原因。偏置并不是什么特别的东西，它就是一个输入永远是 1 的权重。

用代码复现

下面的代码依次算了：一个神经元给苹果打分，以及 w_1 加 0.1 之后的打分；三个神经元组成的线性层；一次算三种水果；TinyLM 的 4×8 线性层；最后验证“把偏置藏进矩阵”的写法。

```python
import json
import numpy as np

# 一个神经元：苹果 (7, 4)，w = (1, -1)，b = -1
x, w, b = np.array([7.0, 4.0]), np.array([1.0, -1.0]), -1.0
z = x @ w + b
print("z =", z, "  w1 加 0.1 后：", round(x @ (w + [0.1, 0]) + b, 2))

# 三个神经元并排 = 一个线性层：W 的每一列是一个神经元的权重
W = np.array([[1.0, -0.5, 0.5],
              [-1.0, 1.0, 0.5]])
bias = np.array([-1.0, 0.0, 0.0])
print("苹果 →", x @ W + bias)
fruits = np.array([[7, 4], [8, 1], [2, 9]])      # 苹果、香蕉、柠檬
print(fruits @ W + bias)                          # 每一行一种水果

# TinyLM：一句话 X (4×4) 经过 X·W + b (W 是 4×8)
ex = json.load(open("tinylm.json", encoding="utf-8"))["examples"]["13"]
X, W1, b1 = np.array(ex["X"]), np.array(ex["W"]), np.array(ex["b"])
Y = X @ W1 + b1                                   # b1 自动加到每一行上
print(np.round(Y[0], 2))                          # “我”的 8 个输出
print(np.allclose(Y, ex["Y"]), np.allclose(Y[2], X[2] @ W1 + b1))
print("参数个数：", W1.size + b1.size)

# 把偏置藏进矩阵：x 后面补一个 1，W 下面补一行 b
X_aug = np.hstack([X, np.ones((4, 1))])
W_aug = np.vstack([W1, b1])
print(X_aug.shape, W_aug.shape, np.allclose(X_aug @ W_aug, Y))
```

输出：

```
z = 2.0   w1 加 0.1 后：  2.7
苹果 → [2.  0.5 5.5]
[[ 2.   0.5  5.5]
 [ 6.  -3.   4.5]
 [-8.   8.   5.5]]
[ 1.5  -0.18 -1.28  0.68 -0.56  0.44  1.02  0.21]
True True
参数个数：  40
(4, 5) (5, 8) True
```

逐行对照：苹果的打分是 2，w_1 加 0.1 后变成 2.7，正好多了 0.1 × 7；三个神经元给苹果的打分 (2, 0.5, 5.5)，和三种水果的 3×3 表格，都和上面的手算一致。第四行是“我”经过 TinyLM 线性层的 8 个输出，第一个 1.5 就是刚才手算的那一格。接下来两个 True：整批算的 Y 和 `tinylm.json` 里存的结果一致；单独把“学”这一个词送进去，结果和整批算出的第 3 行相同。最后一行，补了 1 的 X 是 4×5，补了一行的 W 是 5×8，乘出来和 $\mathrm{X} \cdot \mathrm{W} + \mathbf{b}$ 一样。

到了大模型里

大模型里到处都是线性层，代码里一般叫 `Linear`，名字常常以 `proj`（projection，投影）结尾。以 Llama-3-8B 为例（配置来自官方 config.json：隐藏维度 4096，FFN 隐藏维度 14336，32 层，8 个 KV 头、每头 128 维，词表 128256），每一层 Transformer 里有 7 个线性层，都不带偏置：

线性层	形状 $d_{\text{in}} \times d_{\text{out}}$	参数个数	在哪一讲
q_proj、o_proj	4096 × 4096，各一个	2 × 16,777,216	第 26、28 讲
k_proj、v_proj	4096 × 1024，各一个	2 × 4,194,304	第 26、39 讲
gate_proj、up_proj	4096 × 14336，各一个	2 × 58,720,256	第 31、38 讲
down_proj	14336 × 4096	58,720,256	第 31 讲

一层合计 218,103,808 个参数，32 层是 6,979,321,856 个，约 70 亿。整个模型约 80.3 亿个参数，光是这些线性层就占了 86.9%；最后把向量变成词表上打分的输出层（第 33 讲）也是一个 4096 × 128256 的线性层，算上它就超过 93%。剩下的主要是词嵌入表，归一化层的参数只有 26 万多个，可以忽略。

这些数字说明了一件事：**大模型的参数，绝大部分是线性层的权重矩阵**。每处理一个词，这 70 亿个权重每一个都要和输入相乘一次、累加一次。所以推理时，显存主要被它们占着，时间主要花在这些矩阵乘法上，也就是把权重从显存搬进计算单元、再做乘加。后面讲量化、讲 Roofline、讲张量并行，几乎都是在想办法让这件事更快、更省。

拿其中一个来看：up_proj 是 4096 进、14336 出，相当于 14336 个神经元，每个神经元有 4096 个权重，同时盯着同一个 4096 维的词向量打分。这就是上一讲末尾说的“14336 个感知机并排工作”，写成矩阵，就是一次 $\mathbf{x} \cdot \mathrm{W}$。

容易混淆的地方

1. **PyTorch 里 W 的形状是反过来的**。这门课把输入写成行向量，W 是 $d_{\text{in}} \times d_{\text{out}}$，每一列是一个神经元。PyTorch 的 `nn.Linear(d_in, d_out)` 却把权重存成 $d_{\text{out}} \times d_{\text{in}}$（每一**行**是一个神经元），计算时再转置：$\mathrm{y} = \mathrm{x} \cdot \mathrm{W}^{\intercal} + \mathrm{b}$。Hugging Face 上下载的 Llama 权重也是这样存的，q_proj 的形状写的是 [4096, 4096]，k_proj 写的是 [1024, 4096]。读代码和权重文件时，先看清楚是哪种存法。
2. **“线性层”并不严格线性**。第 4 讲说过，平移不是线性变换，所以带偏置的 $\mathbf{x} \cdot \mathrm{W} + \mathbf{b}$ 在数学上叫仿射变换（affine transformation）。深度学习里不讲究这个区别，一律叫线性层。不带偏置的 Llama 线性层，倒是名副其实的线性。
3. **神经元、线性层、全连接层、Dense、投影**。这几个词说的是同一类东西的不同层次：神经元是一个打分单元（W 的一列）；一组神经元并排就是线性层，也叫全连接层，Keras 里叫 Dense；大模型代码里把它叫“投影”，强调它把向量从一个空间送到另一个空间。

回顾

去掉感知机的阶跃函数之后，剩下的“点积加偏置”就是神经元的核心：输出对每个参数的小改动都有反应，权重 w_i 动 Δ，输出就动 $\Delta \times x_i$。神经元再接一个激活函数 f，这一讲先让 $f(\mathrm{z}) = \mathrm{z}$。把很多个神经元的权重竖着排成矩阵的列，它们同时对同一个输入打分，就是线性层 $\mathbf{y} = \mathbf{x} \cdot \mathrm{W} + \mathbf{b}$；把很多个输入叠成矩阵 X，一次矩阵乘法就能把整批都算完。一个线性层有 $d_{\text{in}} \times d_{\text{out}} + d_{\text{out}}$ 个参数，而大模型的参数，绝大部分就是线性层的权重。

既然一个线性层能把向量送到新的空间，自然会想：叠两层、十层，是不是就能表达更复杂的东西，比如上一讲的异或？可是第 4 讲已经说过，两次线性变换合起来，还是一次线性变换。$(\mathbf{x} \cdot \mathrm{W}_1 + \mathbf{b}_1) \cdot \mathrm{W}_2 + \mathbf{b}_2$ 展开以后，仍然是“$\mathbf{x}$ 乘一个矩阵再加一个向量”的样子，叠多少层都等于一层。要让层数真正起作用，必须在层与层之间插进点“不是线性”的东西。那就是被我们暂时放在一边的 f：下一讲，**激活函数**。

第14讲 激活函数

04 线性变换　12 感知机　13 神经元与线性层

上一讲最后留了一个问题：一个线性层能把向量送到新的空间，那叠两层、十层，是不是就能表达更复杂的东西？答案是不能。不管叠多少层线性层，合起来都只等于一层。这一讲先用一个具体的例子把这件事算清楚，再看怎么破解它：在层与层之间插进一个小小的“弯”，也就是**激活函数**（activation function）。我们会看到，加了这个弯之后，网络能算出异或，能拼出任意形状的曲线；最后再认识大模型里真正在用的三个激活函数：ReLU、GELU 和 SiLU。

叠两层，还是一层

还是上一讲的水果线性层。第一层有三个神经元，给水果打出好吃分、酸味分和浓郁分：

$$W_1 = [[1, -0.5, 0.5], [-1, 1, 0.5]], \qquad \boldsymbol{b}_1 = (-1, 0, 0)$$

现在在后面接第二层，只有一个神经元，把三个分数合成一个“总评”：总评 = 好吃分 + 2 × 酸味分 − 浓郁分 + 0.5。写成矩阵，W_2 是 3×1 的 [[1], [2], [−1]]，$b_2 = 0.5$。

拿香蕉 (8, 1) 来算。第一层得到 (8 − 1 − 1, −4 + 1, 4 + 0.5) = (6, −3, 4.5)；第二层得到 6 + 2 × (−3) − 4.5 + 0.5 = −4。

两层算下来，总评是 −4。可是把式子展开看一下：

$$(\mathbf{x} \cdot W1 + \mathbf{b1}) \cdot W2 + \mathbf{b2} = \mathbf{x} \cdot (W1W2) + (\mathbf{b1}W2 + \mathbf{b2})$$

这一步只用了矩阵乘法的分配律和结合律：先乘 W_1 再乘 W_2，等于直接乘 W_1W_2。右边又是“**x** 乘一个矩阵、再加一个向量”，也就是一个线性层。具体算出来，$W_1W_2 = (1 - 1 - 0.5, -1 + 2 - 0.5) = (-0.5, 0.5)$，$b_1W_2 + b_2 = -1 + 0.5 = -0.5$。所以这个两层网络其实就是

$$\text{总评} = -0.5 \times \text{甜度} + 0.5 \times \text{酸度} - 0.5$$

用香蕉验算：−4 + 0.5 − 0.5 = −4，和两层算出来的一样。第一层精心打的三个分数，在合并后的式子里完全消失了。

这不是这组数字的巧合。第 4 讲说过，两次线性变换接在一起还是一次线性变换。所以**只用线性层，叠十层和叠一层的表达能力是一样的**；第 12 讲的异或，一层分不开，十层也分不开。多出来的层只是白白多了参数和计算。

在两层之间折一下

要让第二层不能和第一层合并，就得在中间做一件“不是线性”的事。最简单的办法是：**把负数变成 0，正数原样留下**。

$$\mathrm{ReLU}(z) = \max(0, z)$$

这个函数叫 **ReLU**（Rectified Linear Unit，修正线性单元）。它的图像是一条折线：z ⩽ 0 时贴着横轴是 0，z > 0 时是斜率为 1 的直线，在原点处折了一下。

把它插进刚才的两层之间：第一层得到 (6, −3, 4.5)，每个数各自过一次 ReLU，变成 (6, 0, 4.5)，酸味分的 −3 被截成了 0；第二层再算：6 + 2 × 0 − 4.5 + 0.5 = 2。总评从 −4 变成了 2。

不同的水果，被截掉的分数不一样。香蕉截掉的是酸味分，柠檬可能截掉的是好吃分，苹果三个分数都是正的，一个都不截。第二层看到的东西，取决于第一层哪些神经元“亮了”（输出大于 0），这个开关是跟着输入变的。正因为如此，整个网络再也不能写成一个固定的“$\mathbf{x} \cdot \mathrm{W} + \mathbf{b}$”，也就不能合并成一层了。

这里要注意，激活函数是**逐元素**（element-wise）作用的：向量里的每个数各自过一次函数，互不影响，形状也不变。上一讲 TinyLM 里“我”经过线性层得到 8 个数：

$$(1.5, -0.18, -1.28, 0.68, -0.56, 0.44, 1.02, 0.21) \xrightarrow{\mathrm{ReLU}} (1.5, 0, 0, 0.68, 0, 0.44, 1.02, 0.2$$

三个负数变成 0，其余不变。8 个神经元里，有 5 个对“我”这个词“亮了”。

于是，前一讲那个“神经元 = 加权求和，再过一个函数 f”里的 f，终于有了着落。一层完整的神经网络是

$$\mathbf{h} = f(\mathbf{x} \cdot W + \mathbf{b})$$

先线性层，再激活函数。层与层之间都夹着一个 f，网络才真正“深”了起来。

要点

激活函数的作用，是在层与层之间插进非线性，让多层网络不会塌缩成一层。它逐元素作用，不改变形状。

两个 ReLU 算出异或

第 12 讲用三个感知机拼出了异或，但那些阶跃函数没法训练。现在用 ReLU 再拼一次。隐藏层两个神经元，权重都是 (1, 1)，偏置分别是 0 和 −1：

$$h_1 = \mathrm{ReLU}(x_1 + x_2), \qquad h_2 = \mathrm{ReLU}(x_1 + x_2 - 1), \qquad y = h_1 - 2h_2$$

逐个算：

输入	$x_1 + x_2$	h_1	h_2	$y = h_1 - 2h_2$	异或
(0, 0)	0	0	0	0	0
(0, 1)	1	1	0	1	1
(1, 0)	1	1	0	1	1
(1, 1)	2	2	1	0	0

关键在 h_2：它只有在两个输入都是 1 时才“亮”，这时减掉 2 倍的它，把 (1, 1) 的输出从 2 拉回 0。如果去掉 ReLU，h_2 在 (0, 0) 时是 −1，y 就成了 $0 - 2 \times (-1) = 2$，整个结构又退化成一个线性函数，算不出异或。

和感知机版本相比，这个网络有一个本质的不同：ReLU 在 $z > 0$ 的一侧是斜线，参数稍微动一下，输出也跟着动一下。这意味着它的权重可以靠第 17、18 讲的梯度下降和反向传播自己学出来，不必由人摆好。

折线能拼出任何形状

ReLU 只是把直线折了一下，它的能力到底有多大？换一个问题来看：用一层 ReLU 神经元，能拟合出什么样的曲线？

只有一个输入 x 的时候，神经元 ReLU(x − c) 的图像是一条在 x = c 处折起来的线：c 左边是 0，右边是斜率 1 的直线。把好几个这样的神经元按不同的权重加起来，每个神经元贡献一个折点，就得到一条**折线**。折点越多，折线就能贴着一条曲线走得越紧。

下面的演示里，虚线是目标曲线 sin(1.5x)。隐藏层有 N 个神经元，第 k 个是 $h_k = f(x - c_k)$，折点 c_k 均匀排在 −3 到 3 之间（底部的小竖线）；输出是 $\hat{y} = v_0 + v_1 h_1 + \cdots + v_N h_N$。为了让你只关注“激活函数和神经元个数”的影响，输出权重 v 不是训练出来的，而是用最小二乘法（让误差的平方和最小）直接算出的最优值。勾选“画出每个神经元的贡献”，可以看到每个 $v_k h_k$ 各自是什么样子，绿色的拟合结果就是它们的和。

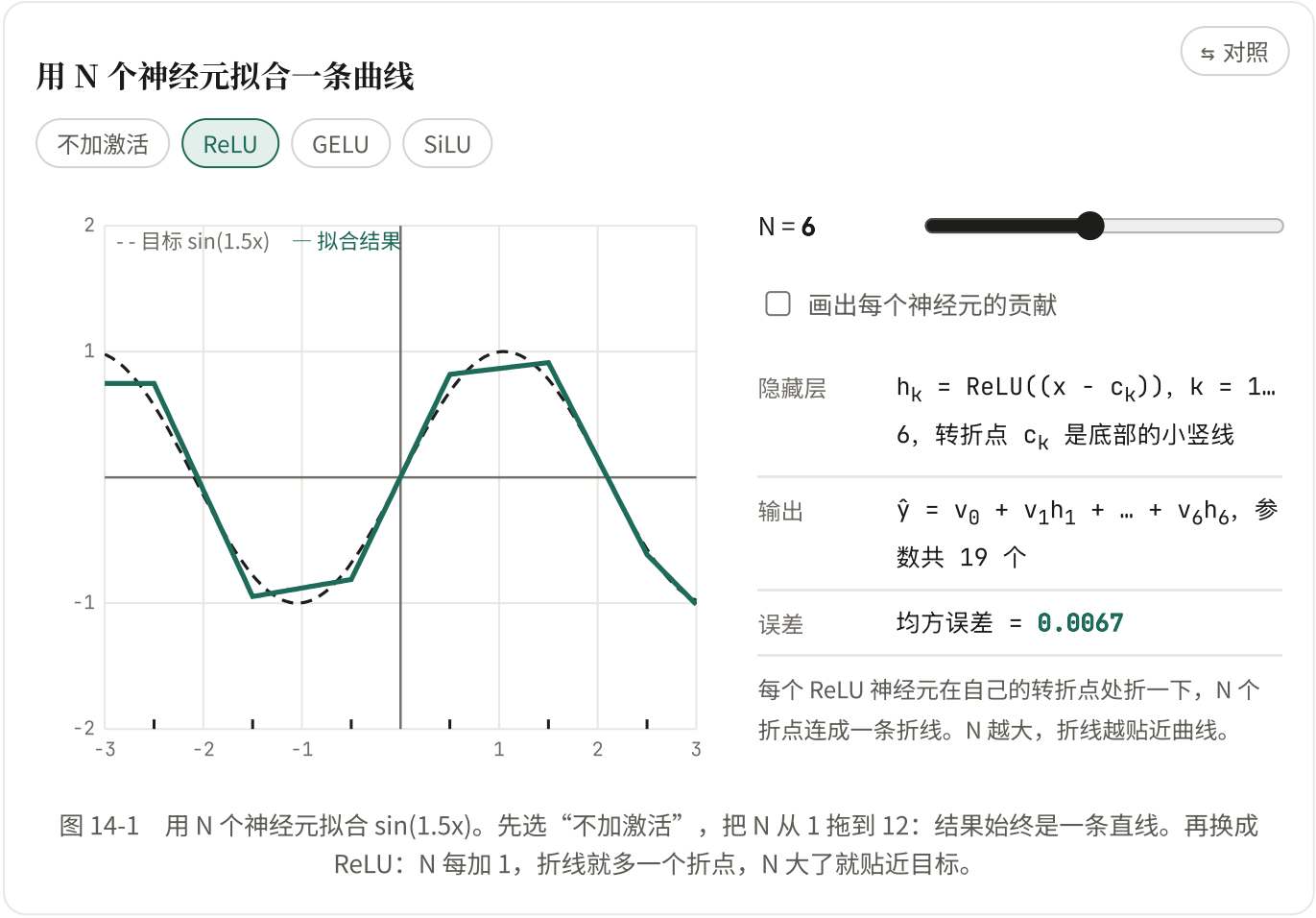


图 14-1　用 N 个神经元拟合 sin(1.5x)。先选“不加激活”，把 N 从 1 拖到 12：结果始终是一条直线。再换成 ReLU：N 每加 1，折线就多一个折点，N 大了就贴近目标。

先选“不加激活”。不管 N 拖到多大，绿线都是一条直线，误差也几乎不变：N 个线性神经元加起来，还是一个线性函数，这就是第一节说的“塌缩”。

换成 ReLU。N = 1 到 3 时折点太少，只能画出一个很粗的轮廓；而且折点是均匀摆放的，不是挑在波峰波谷上，所以 N 从 2 加到 3，误差反而略有上升。到 N = 6，折线已经跟上了波峰和波谷，误差从不加激活时的 0.48 降到 0.007 左右；N 到 10 以上，折线和正弦曲线已经很难分清了。再换成 GELU 或 SiLU，拼出来的是平滑的曲线，没有尖角。

这个现象背后有一个定理，叫**万能近似定理**（universal approximation theorem）：只要有一个隐藏层、用的是非线性的激活函数，并且神经元足够多，就能以任意精度逼近任何一个“性质不太差”的连续函数（在一个有限范围内）。演示里看到的，正是一维时的样子。

> **能逼近，不等于能学会**
> 万能近似定理说的是“存在”这样一组参数，并没有说怎么找到它，也没有说需要多少个神经元。实际中，用很多层、每层适中的网络，往往比一层特别宽的网络更省参数、更好训练。第 15 讲会看到，多层网络是怎么一层一层地把问题变简单的。

ReLU、GELU、SiLU

ReLU 简单又好用，但它在 0 处有一个尖角，负的一侧整个是 0。后来的研究者设计了一些“平滑版的 ReLU”，大模型里最常见的是 GELU 和 SiLU：

$$\mathrm{GELU}(z) = z \cdot \Phi(z), \qquad \mathrm{SiLU}(z) = z \cdot \sigma(z)$$

两个式子的样子一样，都是“z 乘一个 0 到 1 之间的系数”。这个系数可以理解成一个**软开关**：z 很大时系数接近 1，z 几乎原样通过；z 很负时系数接近 0，输出被压到 0 附近。ReLU 其实也是这个样子，只不过它的开关是硬的：$z > 0$ 时系数是 1，否则是 0。

两个软开关来自不同的地方：

- $\Phi(z)$ 是标准正态分布的累积分布函数，也就是“一个标准正态随机数小于 z 的概率”。$\Phi(0) = 0.5$，z 越大越接近 1。GELU（Gaussian Error Linear Unit）的名字就来自这里。
- $\sigma(z) = 1 / (1 + e^{-z})$ 叫 **Sigmoid** 函数，它把任意实数压到 0 和 1 之间，$\sigma(0) = 0.5$。SiLU（Sigmoid Linear Unit）也叫 Swish。

在下面的演示里切换不同的函数。绿色实线是函数本身，琥珀色虚线是它的**斜率**（导数）$f'(z)$，拖动 z，黑色短线是那一点的切线。默认还叠加了另外两个函数的浅灰色曲线，方便对比。

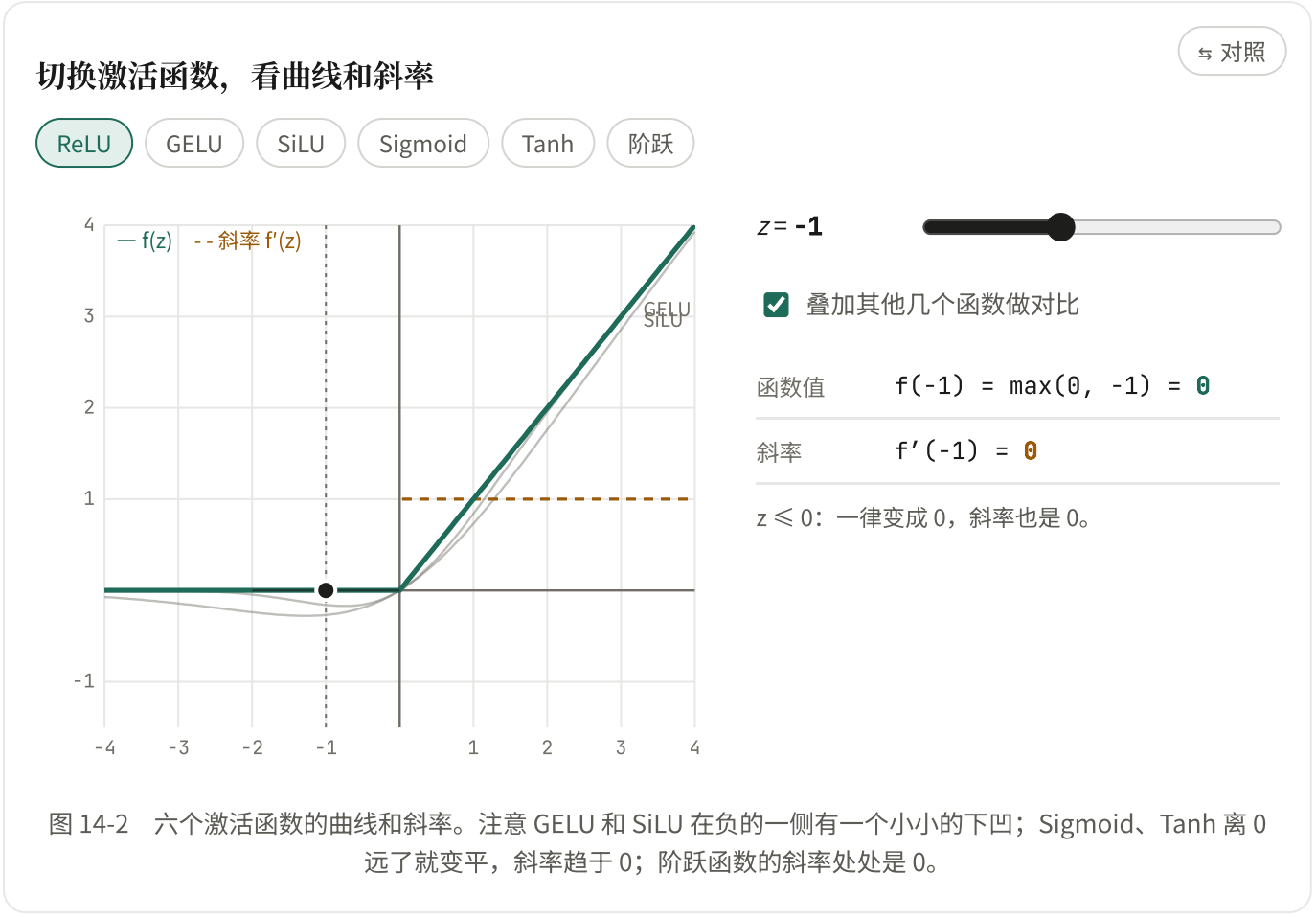


图 14-2　六个激活函数的曲线和斜率。注意 GELU 和 SiLU 在负的一侧有一个小小的下凹；Sigmoid、Tanh 离 0 远了就变平，斜率趋于 0；阶跃函数的斜率处处是 0。

几个值得注意的地方，都能在演示里看到：

ReLU、GELU、SiLU 在 z 为正、比较大时几乎重合，都接近 y = z；区别全在 0 附近和负的一侧。在 z = −1 处，ReLU 是 0，GELU 是 −0.159，SiLU 是 −0.269。GELU 在 z ≈ −0.75 处到达最低点，约 −0.17；SiLU 的最低点在 z ≈ −1.28，约 −0.28。也就是说，平滑版的 ReLU 会让一点点负值“漏”过去，而且在负的一侧斜率不完全是 0。

斜率为什么重要？第 13 讲说过，训练要靠“参数动一点，输出变多少”来决定怎么改参数，而经过激活函数时，这个变化量要乘上激活函数在那一点的斜率。ReLU 在负的一侧斜率是 0：如果某个神经元对所有输入都输出负数，它的权重就再也得不到改进的信号，这叫**死亡 ReLU**（dying ReLU）。GELU 和 SiLU 在负的一侧还保留着一点斜率，缓解了这个问题，这是它们在大模型里更受欢迎的原因之一。

Sigmoid 和 Tanh 是更早的激活函数，在 ReLU 流行之前用得很多。它们的问题是**饱和**：z 离 0 一远，曲线就变平，斜率趋于 0（Sigmoid 的斜率最大也只有 0.25）。网络一深，这些小于 1 的斜率一层层乘下去，越乘越小，前面几层几乎收不到训练信号，这叫梯度消失，第 18、29 讲会再讲到。今天它们主要用在需要“0 到 1 之间的开关”的地方，比如 SiLU 里面的 σ，以及第 38 讲的门控结构。

至于哪个模型用哪个：2017 年最初的 Transformer 用的是 ReLU；BERT、GPT-2 换成了 GELU；Llama 系列用的是 SiLU，而且是放在一种叫 SwiGLU 的门控结构里用的（第 38 讲）。

用代码复现

下面的代码依次验证：两层线性层可以合并成一层，插进 ReLU 之后结果就变了；TinyLM 的“我”经过 ReLU；两个 ReLU 神经元算出异或；最后列出三个激活函数在几个点上的值，并找出 GELU、SiLU 的最低点。

```python
import json
from math import erf, sqrt
import numpy as np

relu = lambda z: np.maximum(0, z)
gelu = lambda z: z * 0.5 * (1 + np.vectorize(erf)(z / sqrt(2)))
silu = lambda z: z / (1 + np.exp(-z))

# 1. 两个线性层叠起来，等于一个线性层
W1, b1 = np.array([[1.0, -0.5, 0.5], [-1.0, 1.0, 0.5]]), np.array([-1.0, 0.0, 0.0])
W2, b2 = np.array([[1.0], [2.0], [-1.0]]), np.array([0.5])
x = np.array([8.0, 1.0])                                    # 香蕉
print((x @ W1 + b1) @ W2 + b2, x @ (W1 @ W2) + (b1 @ W2 + b2))
print("合并后的 W: ", (W1 @ W2).ravel(), " b: ", b1 @ W2 + b2)
print("中间插 ReLU: ", relu(x @ W1 + b1), "→", relu(x @ W1 + b1) @ W2 + b2)

# 2. TinyLM: “我”经过上一讲的线性层，再过 ReLU
ex = json.load(open("tinylm.json", encoding="utf-8"))["examples"]["13"]
y = np.array(ex["Y"][0])
print(y, "→", relu(y))

# 3. 两个 ReLU 神经元算出异或
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
h = relu(X @ np.array([[1, 1], [1, 1]]) + np.array([0, -1]))   # h1 = ReLU(x1+x2),
h2 = ReLU(x1+x2-1)
print(h.tolist(), (h @ np.array([1, -2])).tolist())

# 4. 三个函数在几个点上的值，以及 GELU、SiLU 的最低点
z = np.array([-2.0, -1.0, 0.0, 1.0, 2.0])
for name, f in [("ReLU", relu), ("GELU", gelu), ("SiLU", silu)]:
    print(f"{name:4s}", np.round(f(z), 3))
zz = np.linspace(-3, 0, 30001)
for name, f in [("GELU", gelu), ("SiLU", silu)]:
    i = np.argmin(f(zz))
    print(f"{name} 最低点: z = {zz[i]:.2f}, f = {f(zz)[i]:.3f}")
```

输出：

```
[-4.] [-4.]
合并后的 W: [-0.5  0.5]  b: [-0.5]
中间插 ReLU: [6.  0.  4.5] → [2.]
[ 1.5  -0.18 -1.28  0.68 -0.56  0.44  1.02  0.21] → [1.5  0.   0.   0.68 0.   0.44
1.02 0.21]
[[0, 0], [1, 0], [1, 0], [2, 1]] [0, 1, 1, 0]
ReLU [0. 0. 0. 1. 2.]
GELU [-0.046 -0.159  0.     0.841  1.954]
SiLU [-0.238 -0.269  0.     0.731  1.762]
GELU 最低点: z = -0.75, f = -0.170
SiLU 最低点: z = -1.28, f = -0.278
```

第一行：两层算和合并成一层算，香蕉的总评都是 −4；合并后的权重是 (−0.5, 0.5)，偏置 −0.5，和手算一致。第三行：插进 ReLU 后，酸味分的 −3 变成 0，总评变成 2。第四行是“我”的 8 个数经过 ReLU，三个负数变成 0。第五行是异或网络的隐藏层 (h_1, h_2) 和输出 [0, 1, 1, 0]。最后几行就是上一节提到的数值：在 z = −1 处 GELU 是 −0.159，SiLU 是 −0.269；两者的最低点分别在 −0.75 和 −1.28。

到了大模型里

在 Llama-3-8B 里，每一层的前馈网络会把每个词的向量从 4096 维升到 14336 维，在这 14336 个数上逐个做一次 SiLU，再降回 4096 维（具体结构见第 31、38 讲）。也就是说，每个词每经过一层，要算 14336 次 SiLU。和同一层前馈网络里三个线性层加起来的 $3 \times 4096 \times 14336 \approx 1.76$ 亿次乘加相比，这只是万分之一的零头。

但计算量小，不等于花的时间少。激活函数是典型的**访存密集**型运算：它对每个数只做几次运算，却要先把这个数从显存里读出来，算完再写回去。GPU 的计算速度远远快于搬数据的速度，所以单独跑一次激活函数，时间几乎全花在读写上，计算单元大部分时间在等数据。第 62 讲的 Roofline 模型会把这件事讲清楚。实际的推理引擎不会单独去跑它，而是把激活函数“粘”在前后的运算上，在数据还留在芯片里的时候顺手算掉，省掉一次读写，这叫算子融合（第 83 讲）。

容易混淆的地方

1. **激活函数是逐元素的，softmax 不是。**ReLU、GELU、SiLU 都是每个数各算各的。第 6 讲的 softmax 要先把一整组数的指数加起来再相除，每个输出都依赖所有输入，所以通常不把它算作这一讲意义上的激活函数。

2. **GELU 有精确版和近似版**。$\Phi(z)$ 里有误差函数，早年算起来比较慢，所以 GPT-2 等模型用的是一个 tanh 近似：$0.5z(1 + \tanh(\sqrt{2/\pi}(z + 0.044715z^3)))$。两者的差别很小，但一个模型训练时用的是哪个，推理时也要用同一个，否则结果会有细微偏差。Hugging Face 的配置里写成 `gelu`（精确）或 `gelu_new`、`gelu_pytorch_tanh`（近似）。
3. **ReLU 在 0 处“不可导”不是问题**。ReLU 在 $z = 0$ 处有尖角，严格说没有导数。实际中 z 恰好等于 0 的情况几乎不会发生，框架里约定那一点的斜率取 0 就行，对训练没有影响。

回顾

只用线性层，不管叠多少层都能合并成一层：$(\mathbf{x} \cdot W_1 + \mathbf{b}_1) \cdot W_2 + \mathbf{b}_2$ 展开后还是“$\mathbf{x}$ 乘一个矩阵再加一个向量”。在层与层之间插进一个非线性的激活函数，就打破了这种塌缩。最简单的 ReLU 只是把负数截成 0：它让每个神经元跟着输入“亮”或“不亮”，两个 ReLU 就能算出异或，很多个 ReLU 拼成的折线能逼近任意曲线。GELU 和 SiLU 是平滑版的 ReLU，形式都是“z 乘一个 0 到 1 之间的软开关”，负的一侧保留了一点斜率，是今天大模型的主流选择。激活函数计算量很小，但要读写大量数据，推理时通常和别的运算融合在一起。

这一讲的拟合演示只有一个隐藏层，而且输出权重是直接算出来的。真正的神经网络有好几层，所有权重都要靠训练得到。层数多了，每一层分别在做什么？为什么“深”比“宽”更好用？下一讲用一个小网络从头拟合一条曲线，看看**多层感知机**的每一层都在干什么。

第15讲 多层感知机

13 神经元与线性层　14 激活函数

上一讲用一层 ReLU 神经元拟合了一条正弦曲线。不过那个网络有点“作弊”：隐藏层的折点是我们均匀摆好的，只有输出权重是算出来的。真正的神经网络有好几层，每一层的权重都是训练出来的，没人告诉它折点该放在哪里。

这一讲拿一个已经训练好的小网络，把它一层一层拆开来看，回答两个问题：**每一层到底在做什么？为什么叠很多层，往往比把一层做得很宽更有用？**这样的网络叫**多层感知机**（multilayer perceptron，MLP），它也是大模型每一层里的“前馈网络”。

把层一层层接起来

多层感知机的结构，就是把前两讲的东西串起来：线性层、激活函数、线性层、激活函数……最后再接一个线性层输出结果。

$$\boldsymbol{h}_1 = f(\boldsymbol{x} \cdot W_1 + \boldsymbol{b}_1), \qquad \boldsymbol{h}_2 = f(\boldsymbol{h}_1 \cdot W_2 + \boldsymbol{b}_2), \qquad \boldsymbol{y} = \boldsymbol{h}_2 \cdot W_3 + \boldsymbol{b}_3$$

每一行都是第 14 讲说的“先线性层，再激活函数”，上一行的输出就是下一行的输入。$\mathbf{h}_1$、$\mathbf{h}_2$ 这些中间结果所在的层叫**隐藏层**，最开始的 $\mathbf{x}$ 叫输入层，最后的 $\mathbf{y}$ 叫输出层。数据只从前往后流，不会绕回来，所以这种网络也叫**前馈网络**（feedforward network）。

注意最后一行没有激活函数。我们要输出的是一个可正可负的数（比如正弦曲线的值），如果最后再过一次 ReLU，负数就全被截掉了。输出层要不要激活函数、要什么样的，取决于想输出什么，第 16 讲讲分类问题时还会再遇到。

这一讲用的网络很小：输入是 1 个数 x，两个隐藏层各有 8 个神经元，输出是 1 个数 y，简写成 $1 \to 8 \to 8 \to 1$。按第 13 讲的规则，三个权重矩阵的形状分别是 1×8、8×8、8×1，参数个数是

$$(1 \times 8 + 8) + (8 \times 8 + 8) + (8 \times 1 + 1) = 16 + 72 + 9 = 97$$

每层神经元的个数叫网络的**宽度**，隐藏层的层数叫网络的**深度**。这个网络宽 8、深 2。

拆开一个训练好的网络

这个网络的任务和上一讲一样：输入 x（−3 到 3 之间），输出要尽量接近 sin(1.5x)。它的 97 个参数是用第 16–20 讲会讲到的方法训练出来的，存在 `tinylm.json` 里。怎么训练先不管，我们把它当成一台现成的机器，看看它内部发生了什么。

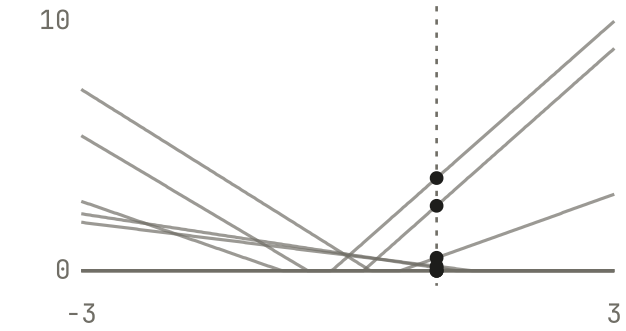
下面的演示把每一层都画了出来。横轴都是输入 x；每个隐藏层的图里有 8 条线，第 j 条线表示“输入为 x 时，这一层第 j 个神经元的输出”。最后一张图是网络的输出 y，虚线是目标 sin(1.5x)。拖动上面的 x，竖线会在每张图上标出这一点，图下面的小标签写着每个神经元此刻的输出，最下面逐层写出了计算过程。把鼠标移到小标签上，可以高亮那个神经元的曲线；点一下，就把这个神经元“关掉”（让它的输出始终为 0），看后面会怎样。

一个训练好的 1→8→8→1 网络，逐层看

⇆ 对照

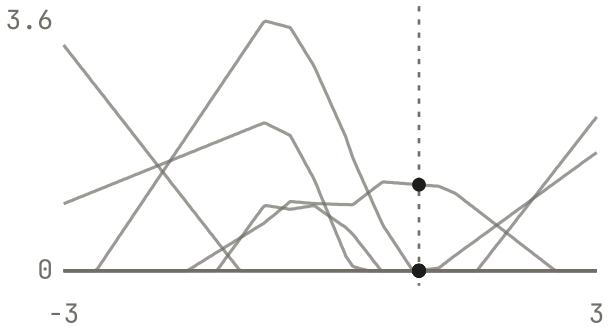
$x = 1$

第 1 层 · 8 个神经元 （在这个范围里有输出的：8 个）



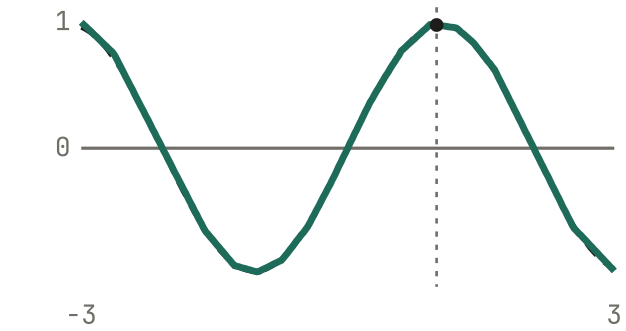
h_1 0.12 h_2 2.6 h_3 0 h_4 0.52 h_5 3.71 h_6 0 h_7 0 h_8 0.18

第 2 层 · 8 个神经元 （在这个范围里有输出的：6 个）



h_1 0 h_2 1.25 h_3 0.01 h_4 0 h_5 0 h_6 0 h_7 0 h_8 0

输出 y　虚线是目标



y = 0.998，目标 sin(1.5x) = 0.997

输出是一条由 **17** 段直线连成的折线，均方误差 0.00017

输入	$x = 1$
第 1 层	$h_1 = \mathrm{ReLU}(x \cdot W_1 + b_1) = (0.12, 2.6, 0, 0.52, 3.71, 0, 0, 0.18)$
第 2 层	$h_2 = \mathrm{ReLU}(h_1 \cdot W_2 + b_2) = (0, 1.25, 0.01, 0, 0, 0, 0, 0)$
输出	$y = h_2 \cdot W_3 + b_3 = \mathbf{0.998}$

图 15-1　1 → 8 → 8 → 1 的网络拟合 sin(1.5x)。第 1 层的每条线只折一次；第 2 层的线由第 1 层组合再折，形状复杂得多；输出是一条 17 段的折线，和目标几乎重合。

第 1 层：每个神经元折一次

第 1 层的输入只有一个数 x，所以第 j 个神经元算的是 $h_{1,j} = \mathrm{ReLU}(w_j\,x + b_j)$。这和上一讲的拟合演示是同一种东西：一条在某处折了一下的线。折点在 $w_j\,x + b_j = 0$ 的地方，也就是 $x = -b_j / w_j$。w_j 是正的，线在折点右边往上走；w_j 是负的，线在折点左边往上走（往左越走越高）。

这 8 个折点是训练自己选出来的，分别在 −0.74、−0.45、−0.18、0.17、0.26、0.59、1.22、1.41。有意思的是，它们全都挤在 −0.74 到 1.41 之间，没有一个落在 −3 到 −1 或 1.5 到 3 的区域。可是输出的折线明明在整个范围里都贴着正弦曲线拐弯。这些弯是从哪来的？

第 2 层：组合，再折一次

答案在第 2 层。第 2 层的第 k 个神经元算的是

$$h2,k = \mathrm{ReLU}\big(W2[1,k]\cdot h1,1 + W2[2,k]\cdot h1,2 + \cdots + W2[8,k]\cdot h1,8 + b2,k\big)$$

括号里是第 1 层 8 条折线的加权和，它本身也是一条折线，折点就是第 1 层的那 8 个。但这条折线会上上下下地穿过 0，ReLU 在每个穿过 0 的地方又折出一个新的拐角。这些新拐角的位置不受第 1 层的限制，可以落在任何地方。所以在演示里，第 2 层的曲线形状比第 1 层复杂得多：有的只在一小段区间里不为 0，像一个“凸起”，在别处都贴着 0。

输出层把第 2 层的 8 条线加权相加。数一数，最后的输出是一条由 **17 段**直线连成的折线；如果只靠第 1 层的 8 个折点，最多只有 9 段。多出来的那些拐角，都是第 2 层“再折一次”时产生的。

拖到 x = 1 看一下具体的数。第 1 层的输出是 (0.12, 2.6, 0, 0.52, 3.71, 0, 0, 0.18)，三个神经元是 0；到了第 2 层，只剩两个神经元不是 0：$h_{2,2} = 1.25$，$h_{2,3} = 0.01$。输出 $y = 0.68 \times 1.25 - 0.25 \times 0.01 + 0.15 \approx 0.998$，目标 $\sin(1.5)$ 是 0.997。也就是说，在 x = 1 这一点，整个输出几乎全靠第 2 层的第 2 个神经元撑着。

关掉几个神经元试试

现在试着关掉几个神经元。

先点第 2 层的 h_2（在 x = 1 时是 1.25 的那个）。它只在 −1.6 到 2.5 之间不为 0，关掉以后，中间那段波峰整个塌了下去，均方误差从 0.00017 涨到 0.25 左右。再试第 2 层的 h_7：它只在 $x < -1$ 的那一段亮着，关掉它，左边的波谷就没了。可以说，第 2 层的每个神经元各自负责曲线的一

部分。

再试第 2 层的 h_1 和 h_8：关掉它们，输出一点都不变。看它们的曲线就知道，在 −3 到 3 的整个范围里它们的输出始终是 0，第 14 讲说的“死亡 ReLU”就是这个样子。图上方写着“有输出的：6 个”，指的就是这件事。这个网络实际上只用了第 2 层的 6 个神经元。

最后关掉第 1 层的任意一个神经元。这一次，第 2 层的好几条曲线同时变了形，输出也会大幅偏离目标。第 1 层的每个神经元，都是第 2 层所有神经元的“原材料”，牵一发而动全身。

要点

第 1 层把输入折成一组简单的折线；第 2 层把这些折线组合起来再折一次，在新的位置产生新的拐角。层数越多，折出来的形状越复杂。

每一层都在折纸

“再折一次能多出拐角”到底能多出多少？用一个人为构造的网络看得最清楚。

拿一张长条纸，代表区间 [0, 1]。从中间对折一次，再展开，纸上多了一道折痕，分成 2 段。对折两次再展开，有 4 段；对折三次，8 段。每对折一次，段数翻一倍。

神经网络也能“对折”。下面这个函数把 [0, 1] 对折了一次：

$$t(u) = 2 \cdot \mathrm{ReLU}(u) - 4 \cdot \mathrm{ReLU}(u - 0.5)$$

u 从 0 走到 0.5 时，只有第一项起作用，t 从 0 涨到 1；u 从 0.5 走到 1 时，第二项也加进来，斜率变成 2 − 4 = −2，t 又从 1 降回 0。整条线是一个“人”字形：[0, 0.5] 和 [0.5, 1] 两段都被映射到了 [0, 1] 上，就像纸被对折后两半叠在一起。

它只用到两个 ReLU 神经元。把 t 的输出再送进下一个 t，就是一个每层 2 个神经元的多层网络。每过一层，前面的每一段又被对折一次，段数翻倍。

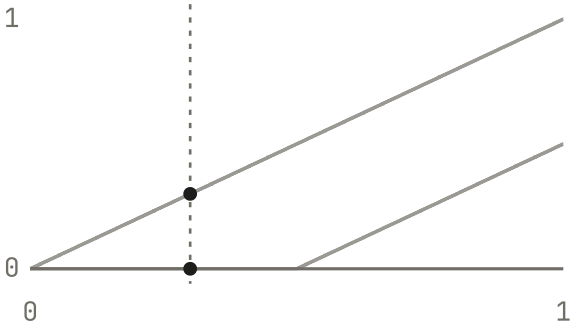
每层只有 2 个神经元的“锯齿”网络

⇆ 对照

x = **0.3**

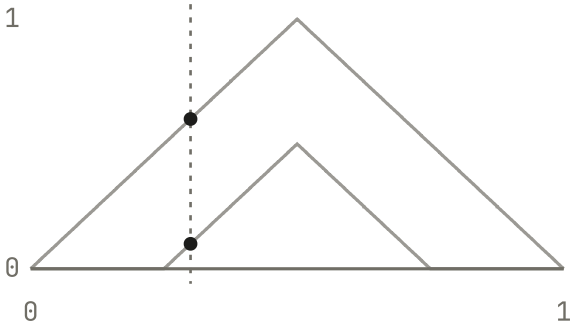
层数 k = **3**

第 1 层 · 2 个神经元 （在这个范围里有输出的：2 个）



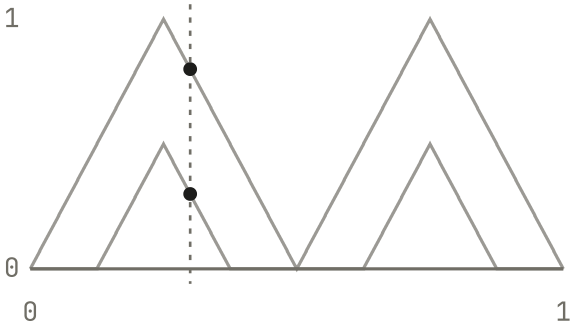
h_1 0.3 h_2 0

第 2 层 · 2 个神经元 （在这个范围里有输出的：2 个）



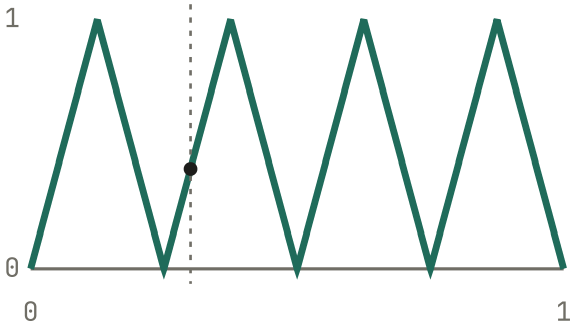
h_1 0.6 h_2 0.1

第 3 层 · 2 个神经元 （在这个范围里有输出的：2 个）



h_1 0.8 h_2 0.3

输出 y



y = 0.4

输出是一条由 **8** 段直线连成的折线；3 层 × 2 个 = 6 个神经元

输入	$x = 0.3$
第 1 层	$h_1 = \mathrm{ReLU}(x \cdot W_1 + b_1) = (0.3, 0)$
第 2 层	$h_2 = \mathrm{ReLU}(h_1 \cdot W_2 + b_2) = (0.6, 0.1)$
第 3 层	$h_3 = \mathrm{ReLU}(h_2 \cdot W_3 + b_3) = (0.8, 0.3)$
输出	$y = h_3 \cdot W_4 + b_4 =$ **0.4**

图 15-2 “锯齿”网络：每层只有 2 个神经元。拖动层数 k，输出的段数是 2、4、8、16、32、64，每加一层翻一倍。

把层数 k 从 1 拖到 6，输出依次是 2、4、8、16、32、64 段的锯齿，用到的神经元只有 2、4、6、8、10、12 个。

换成只有一个隐藏层的网络来做同一件事，情况就完全不同了。输入是一个数时，一层 N 个 ReLU 神经元最多只有 N 个折点，也就是最多 N + 1 段（上一讲的拟合演示就是这样）。要拼出 64 段的锯齿，至少要 63 个神经元。**深度让段数按指数增长，宽度只能让它按线性增长**。

当然，锯齿是一个特意挑出来的例子，不是每个函数都需要这么多段。但现实中的很多问题都有类似的“组合”结构：先认出笔画，再由笔画认出字，再由字认出词。多层网络可以让后面的层反复利用前面的层算好的东西，而不必每一层都从头开始。这是“深度”学习这个名字的由来，也是大模型要叠几十层的原因之一。

> **深的网络为什么以前训不好**
>
> 多层网络的想法很早就有，但直到 2010 年前后，十层以上的网络都很难训练：信号经过一层层的 Sigmoid，梯度越传越小（第 14 讲提到的梯度消失）。ReLU、更好的初始化方法、归一化（第 30 讲）和残差连接（第 29 讲）这些技术陆续出现，才让几十层、上百层的网络变得可以训练。

用代码复现

下面的代码读出这个网络的 97 个参数，在 x = 1 处逐层算一遍，再统计输出有几段折线、误差多大；最后验证锯齿网络每加一层段数翻倍。

```
import json
import numpy as np

relu = lambda z: np.maximum(0, z)
net = json.load(open("tinylm.json", encoding="utf-8"))["examples"]["15"]["layers"]
(W1, b1), (W2, b2), (W3, b3) = [(np.array(L["W"]), np.array(L["b"])) for L in net]
print("形状：", W1.shape, W2.shape, W3.shape, " 参数：", sum(W.size + b.size for W,
b in [(W1, b1), (W2, b2), (W3, b3)]))

def mlp(x):                                     # x: [n, 1]
    h1 = relu(x @ W1 + b1)                      # [n, 8]
    h2 = relu(h1 @ W2 + b2)                     # [n, 8]
    return h1, h2, h2 @ W3 + b3                 # [n, 1]

h1, h2, y = mlp(np.array([[1.0]]))
print("x = 1：h1 =", h1.round(2)[0], "\n       h2 =", h2.round(2)[0], "\n       y
=", y.round(3)[0], " sin(1.5) =", round(np.sin(1.5), 3))

def pieces(ys):                                 # 数折线有几段：相邻两小段的斜率变了，就是一
个折点
    s = np.diff(ys)
    flag = np.abs(np.diff(s)) > 1e-7 * (1 + np.abs(s[1:]) + np.abs(s[:-1])) + 1e-12
    return int(flag[0] + np.sum(flag[1:] & ~flag[:-1])) + 1

xs = np.linspace(-3, 3, 241)[:, None]
h1, h2, y = mlp(xs)
print("第 1 层的折点：", np.sort(-b1 / W1[0]).round(2))
print("输出的段数：", pieces(y[:, 0]), " 均方误差：", round(float(np.mean((y -
np.sin(1.5 * xs)) ** 2)), 5))
print("第 2 层有输出的神经元：", int((h2 > 0).any(axis=0).sum()), "个")

# 锯齿网络：每层 2 个神经元，k 层折出 2^k 段
def saw(x, k):
    u = x
    for _ in range(k):
        u = 2 * relu(u) - 4 * relu(u - 0.5)
    return u
xs = np.linspace(0, 1, 2049)
print("锯齿：", [pieces(saw(xs, k)) for k in range(1, 7)])
```

输出：

```
形状： (1, 8) (8, 8) (8, 1)  参数： 97
x = 1: h1 = [0.12 2.6  0.   0.52 3.71 0.   0.   0.18]
       h2 = [0.   1.25 0.01 0.   0.   0.   0.   0.  ]
       y = [0.998]  sin(1.5) = 0.997
第 1 层的折点： [-0.74 -0.45 -0.18  0.17  0.26  0.59  1.22  1.41]
输出的段数： 17  均方误差： 0.00017
第 2 层有输出的神经元： 6 个
锯齿： [2, 4, 8, 16, 32, 64]
```

和演示里看到的一一对应：参数 97 个；x = 1 时第 2 层只有两个神经元不为 0，输出 0.998；第 1 层的 8 个折点都在 −0.74 到 1.41 之间，输出却有 17 段；第 2 层 8 个神经元里只有 6 个有输出；锯齿网络的段数逐层翻倍。数段数的办法是在 241 个等距的点上看斜率有没有变，演示里用的也是同一个办法。

到了大模型里

大模型里的 MLP 长什么样？Transformer 的每一层由两部分组成：注意力（第 26 讲）和前馈网络（FFN，第 31 讲）。这个前馈网络就是一个只有一个隐藏层的多层感知机，Hugging Face 的 Llama 代码里，这个类的名字就叫 `LlamaMLP`。

以 Llama-3-8B 为例（配置来自官方 config.json），它的形状是 4096 → 14336 → 4096：每个词的 4096 维向量先升到 14336 维，过激活函数，再降回 4096 维。和这一讲的小网络相比，有三点不同。第一，它的输入和输出都是 4096 维的向量，而不是一个数。第二，它的隐藏层用的是 SiLU，并且多了一个“门”（SwiGLU，第 38 讲），所以实际上有三个权重矩阵。第三，它对句子里的每个词**分别**计算，词与词之间不交换信息，交换信息是注意力的事。

宽度方面，隐藏层是输入的 3.5 倍。深度方面，大模型并不是把一个 MLP 做得很深，而是把“注意力 + MLP”这样的块叠了 32 层（第 32 讲），每一层的 MLP 都在前面所有层的结果上再“折”一次。

这些 MLP 也是参数的大头。第 13 讲算过，Llama-3-8B 每一层的线性层共有 218,103,808 个参数，其中前馈网络的三个矩阵占了 $3 \times 4096 \times 14336 = 176{,}160{,}768$ 个，约 81%。推理时每生成一个词，这些权重都要从显存里读一遍，所以 MLP 是推理优化里最常被“开刀”的部分：量化（第 90 讲以后）主要压缩的是它，MoE（混合专家，第 41 讲）则是把它拆成很多个小 MLP，每个词只用其中几个。

容易混淆的地方

1. **“几层网络”的数法不统一**。1 → 8 → 8 → 1 这个网络，有人说它是 2 层（隐藏层的个数），有人说是 3 层（权重矩阵的个数），也有人连输入层一起数成 4 层。读资料时看清楚它数的是什么。大模型说的“32 层”，数的是 Transformer 块，每个块里面又有好几个线性层。
2. **MLP、FFN、前馈网络**。在大模型的语境里，这三个词通常指同一个东西：Transformer 块里那个“升维、激活、降维”的部分。严格说，“前馈网络”泛指所有不带回路的网络，MLP 是其中最常见的一种。
3. **折线不是缺陷**。用 ReLU 的网络，输出永远是一条折线（高维时是由很多平面拼成的曲面），只是段数多了、看起来像曲线。用 GELU、SiLU 的网络则是真正平滑的。两者在实际效果上的差别，主要体现在训练是否稳定，而不是“能不能拟合曲线”。

回顾

多层感知机就是把“线性层 + 激活函数”一层层串起来，最后接一个不加激活的线性层输出结果。拆开一个训练好的 1 → 8 → 8 → 1 网络来看：第 1 层的每个神经元把输入折一次；第 2 层把这些折线组合起来，在穿过 0 的地方再折一次，产生了第 1 层没有的新拐角，于是 8 个折点变成了 17 段折线。关掉不同的神经元可以看到，第 2 层的神经元各自负责曲线的一部分，而第 1 层的神经元影响后面所有的神经元。“锯齿”网络把这个效果推到极致：每层只要 2 个神经元，段数就能逐层翻倍，而单层网络需要的神经元多得多。大模型的前馈网络就是一个 MLP，它占了每一层大约八成的参数。

这一讲一直在用“均方误差”判断网络拟合得好不好，可我们还没有认真说过它是什么。拟合曲线时，误差可以用差值的平方来量；如果网络要做的是二分类，或者像语言模型那样从几万个词里挑下一个词，“错了多少”又该怎么量？这就是下一讲的内容：**损失函数**。

第16讲 损失函数

06 指数、对数与 softmax　15 多层感知机

上一讲判断网络拟合得好不好，用的是“均方误差 0.00017”这样一个数。从这一讲开始，我们要准备让网络**自己学**：调整参数，让预测越来越准。要做到这一点，首先得有一把尺子，能把“预测错了多少”量成一个数。这个数叫**损失**（loss），算它的函数叫**损失函数**（loss function）。训练要做的事，归根结底就是想办法让损失变小。

这一讲介绍三把尺子。拟合一个数时用的**均方误差**；输出是概率时用的**交叉熵**；以及大模型里最常听到的**困惑度**。我们会看到，尺子选得好不好，直接决定了网络能不能学好。

一个数说明错了多少

先看最简单的情况：网络要输出一个数，比如上一讲拟合的 sin(1.5x)。假设有 4 个样本，网络的预测 $\hat{y}$ 和正确答案 y 如下：

样本	预测 $\hat{y}$	答案 y	误差 $\hat{y}-y$	误差的平方
1	0.9	1.0	−0.1	0.01
2	0.2	0.0	0.2	0.04
3	−0.4	−1.0	0.6	0.36
4	1.3	1.0	0.3	0.09

要把 4 个误差合成一个数，最直接的想法是加起来。可是误差有正有负，−0.1 和 0.2 会互相抵消一部分，一个预测得很糟、另一个错得正好相反的网络，总误差也可能是 0。所以先把每个误差平方，变成非负的数，再求平均：

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 = \frac{0.01 + 0.04 + 0.36 + 0.09}{4} = 0.125$$

这就是**均方误差**（mean squared error，MSE）。平方还有两个好处。一是**大错罚得重**：误差 0.6 的平方是 0.36，是误差 0.2 的 9 倍，而不是 3 倍，所以网络会优先去改那些错得最离谱的样本。二是**处处光滑**：平方是一条抛物线，没有尖角，误差每变一点，损失都平滑地跟着变，这对第 17 讲的梯度下降很重要。如果改用误差的绝对值，在误差为 0 的地方就有一个尖角。

上一讲那个网络的 0.00017，就是在 −3 到 3 之间取 241 个点，这样算出来的均方误差。

预测变成了概率

再看第 12 讲的水果问题：判断一种水果好不好吃。感知机直接输出 0 或 1，第 13 讲说过这样没法学。更好的做法是让网络输出一个**概率**：“我有 70% 的把握它好吃”。

做法很简单。网络最后一层照常算出一个分数 z（可正可负），再用第 14 讲见过的 Sigmoid 函数 $\sigma(z) = 1 / (1 + e^{-z})$ 把它压到 0 和 1 之间，当作“好吃”的概率 p。z = 0 时 p = 0.5，表示拿不准；z 越大，越确定是好吃。这个没过 σ 之前的分数 z，通常叫 **logit**。

现在怎么量“错了多少”？假设这种水果真的好吃，网络给“好吃”的概率是 p。p 越接近 1，说明猜得越对；越接近 0，错得越离谱。最常用的尺子是

$$\text{损失} = -\ln p \quad （p \text{ 是模型给}\textbf{正确答案}\text{的概率}）$$

这叫**交叉熵**（cross entropy）损失。ln 是自然对数（第 6 讲）。代几个数进去看看：

正确答案的概率 p	**0.99**	**0.9**	**0.5**	**0.1**	**0.01**
交叉熵 −ln p	0.010	0.105	0.693	2.303	4.605

p = 1 时损失是 0，完美；p 越小，损失越大，p 趋于 0 时损失趋于无穷大。可以把它理解成“**看到正确答案时有多吃惊**”：你说有 99% 的把握，结果真是这样，一点都不吃惊（0.010）；你说只有 1% 的可能，结果偏偏发生了，非常吃惊（4.605）。每次 p 缩小到十分之一，损失就加 ln 10 ≈ 2.3。

要点

交叉熵只看一件事：模型给正确答案分了多少概率。分得越多，损失越小。

为什么不用平方误差

既然答案是 1、预测是 p，为什么不直接用平方误差 $(1 - p)^2$？它也满足“p 越接近 1 损失越小”。

问题出在**错得很自信**的时候。看下面的演示：绿色实线是交叉熵，琥珀色虚线是平方误差，横轴都是正确答案的概率 p。拖动柱子改变 p。

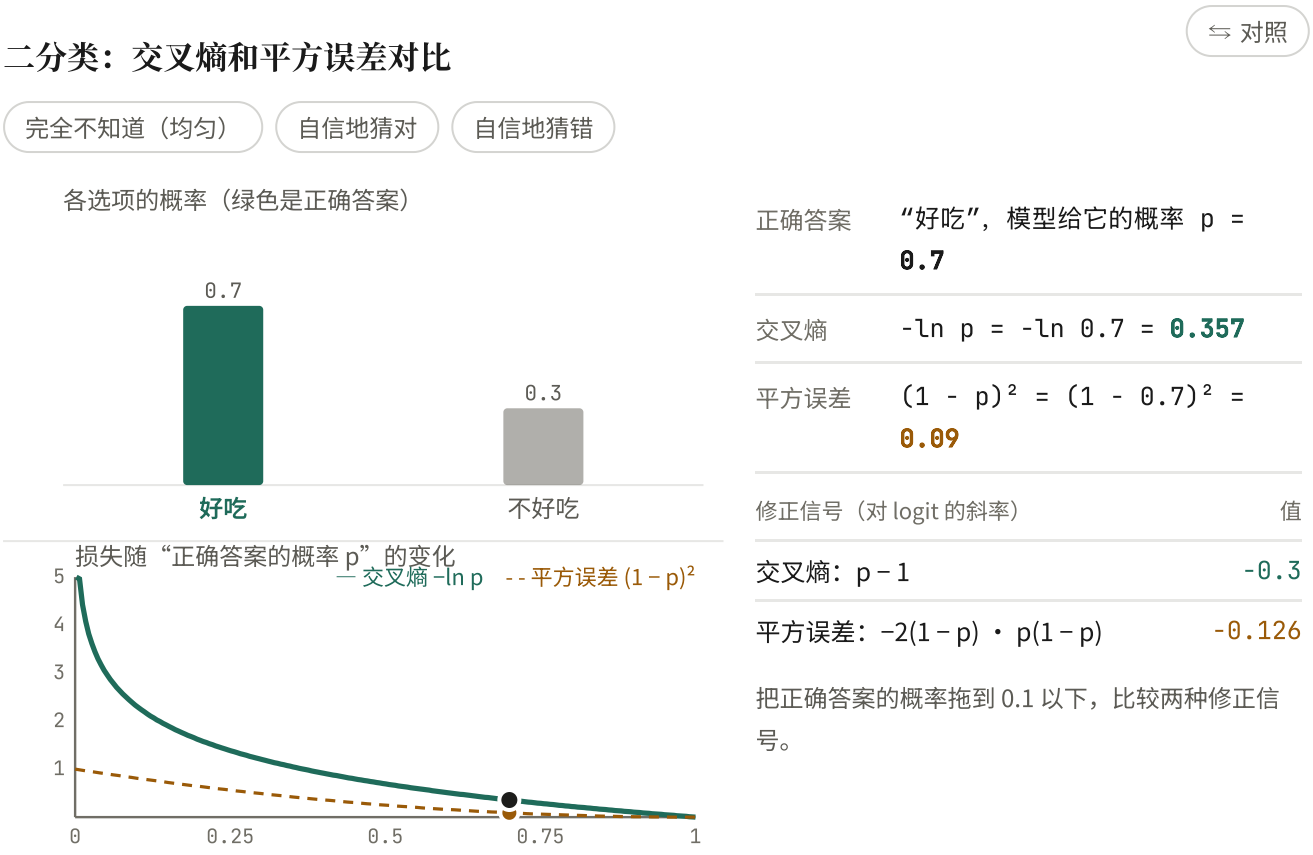


图 16-1　二分类的两种损失。把“好吃”的概率拖到 0.01（或点“自信地猜错”）：交叉熵是 4.605，修正信号 −0.99；平方误差只有 0.98，修正信号只有 −0.0196。

把 p 拖到 0.01，也就是“斩钉截铁地说不好吃，结果它好吃”。平方误差是 0.98，几乎到顶了，因为它最大也只有 1；交叉熵是 4.605，而且还会继续往上涨。平方误差对“错”和“错得离谱”分得不清楚。

更要紧的是右边表格里的**修正信号**。第 13 讲说过，训练要靠“参数动一点，损失变多少”来决定怎么改参数，这个变化率叫斜率（第 17 讲正式叫它梯度）。这里列的是损失对 logit z 的斜率：它告诉网络，z 应该往哪个方向调、调得有多急。

- 交叉熵配 Sigmoid，斜率恰好是 $p - 1$，非常简洁。$p = 0.01$ 时是 -0.99：负号表示 z 应该往大调，绝对值接近 1，表示“赶紧改”。
- 平方误差配 Sigmoid，斜率是 $-2(1 - p) \cdot p(1 - p)$。多出来的因子 $p(1 - p)$ 正是 Sigmoid 的斜率（第 14 讲）。$p = 0.01$ 时它只有 0.0099，于是整个修正信号只剩 -0.0196，是交叉熵的五十分之一。

也就是说，用平方误差时，网络错得越自信，Sigmoid 越饱和，修正信号反而越弱，几乎学不动；用交叉熵时，对数正好抵消了 Sigmoid 的饱和，错得越离谱，推得越用力。这就是分类问题几乎都用交叉熵的原因。

从几个词里挑一个

语言模型面对的也是分类问题，只不过不是二选一，而是**从整个词表里选出下一个词**。TinyLM 的词表有 6 个词，Llama-3 有 128256 个。

做法是二分类的推广。网络最后一层给每个候选词各打一个分（logit），一共 V 个；再用第 6 讲的 softmax 把这 V 个分数变成 V 个概率：

$$p_i = \frac{e^{z_i}}{e^{z_1} + e^{z_2} + \cdots + e^{z_V}}$$

每个概率都在 0 和 1 之间，加起来正好是 1，分数越高的词概率越大。损失还是交叉熵：**−ln（正确的那个词的概率）**。其他词分到多少概率，不直接进入损失，但因为总和是 1，正确的词分得多了，其他词自然就分得少了。

来看 TinyLM。它还没有训练，权重是随机的。第 34 讲才会让一句话完整地走过整个模型，这里先用一个最简单的办法：把当前词的嵌入向量（第 1 讲）直接乘以输出层的矩阵 lm_head（第 33 讲），得到 6 个打分，再过 softmax。对“我”这个词，6 个打分是 (1.64, 0.83, −0.39, −0.17, −0.46, 0.37)，对应的概率是：

候选词	我	爱	学	AI	推理	`<eos>`
概率	0.467	0.208	0.061	0.076	0.057	0.131

例句是“我 爱 学 AI”，“我”后面的正确答案是“爱”，它分到的概率是 0.208，交叉熵 −ln 0.208 = 1.572。模型最看好的反而是“我”自己（0.467），猜错了。

下面的演示就是这个场景。绿色柱子是正确答案“爱”。上下拖动任意一根柱子改变它的概率，其余的柱子按比例缩放，总和保持 1；点下面的词，可以换一个正确答案。

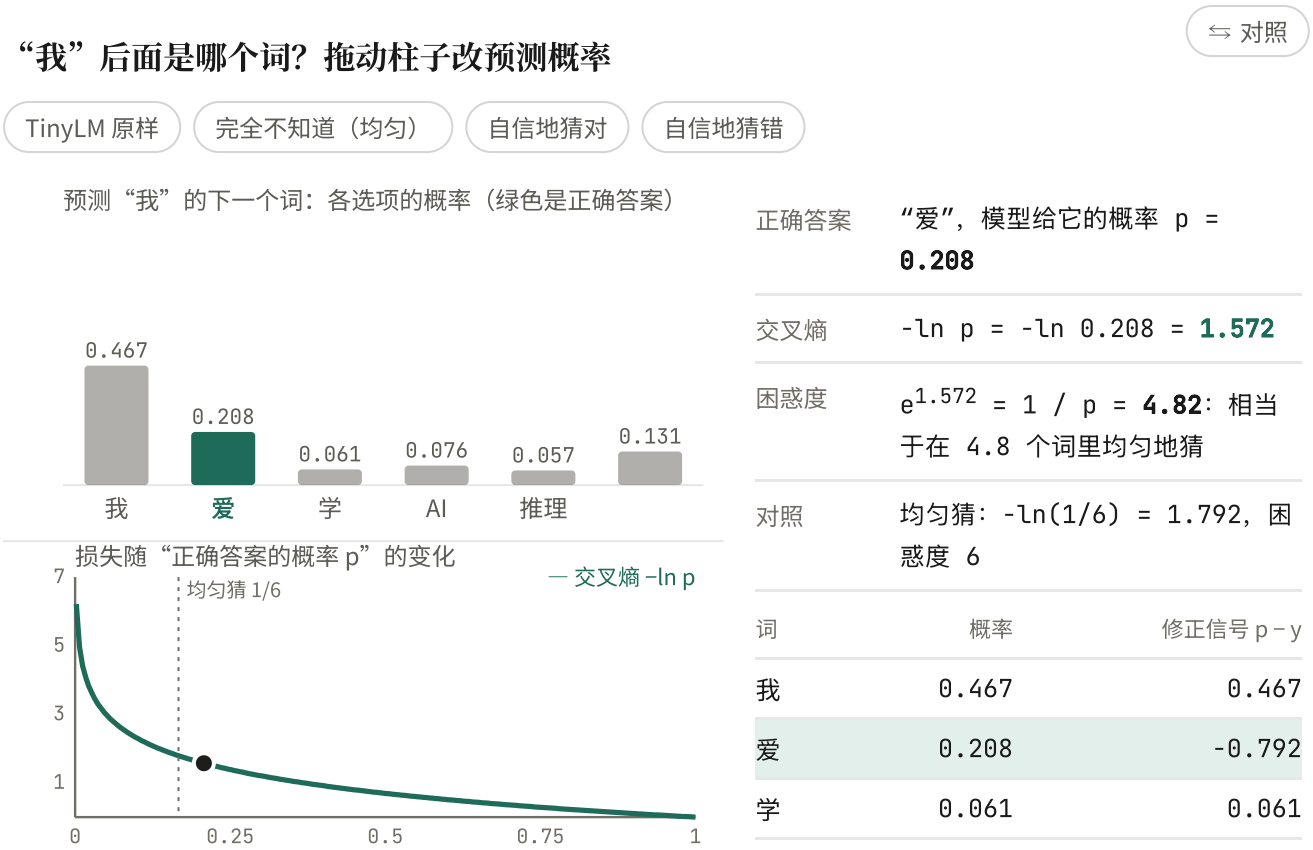


图 16-2　预测“我”的下一个词。默认是 TinyLM 的原样输出：“爱”只分到 0.208，交叉熵 1.572，困惑度 4.82。点“完全不知道”，6 个词各 1/6，交叉熵是 ln 6 = 1.792，困惑度正好是 6。

试几个操作。把“爱”的柱子往上拖，交叉熵随之下降，拖到接近 1 时趋于 0。改去拖别的柱子，只要“爱”的概率不变，交叉熵就不变，哪怕错误答案之间的分配变了。点“完全不知道”，6 个词各分 1/6，交叉熵是 $\ln 6 \approx 1.792$，这是一个很有用的参照点：一个什么都没学到、只会均匀乱猜的模型，损失就是 $\ln V$。

右边的表格还列出了“修正信号 $p - y$”：y 是正确答案的标记，正确的词是 1，其余是 0。这是交叉熵对每个词的 logit 的斜率，和二分类时的 $p - 1$ 是同一个式子的推广。正确的词是负数（把它的分往上调），其余的词是正数（往下调），而且概率越大的错误答案，被往下压得越多。

一句话的损失，和困惑度

一句话里有很多个位置，每个位置都要预测下一个词。“我 爱 学 AI”后面加上结束符 `<eos>`，一共要预测 4 次：

当前词 → 正确的下一个词	**我 → 爱**	**爱 → 学**	**学 → AI**	**AI → `<eos>`**
正确答案的概率	0.208	0.098	0.202	0.089
交叉熵	1.572	2.323	1.601	2.421

把 4 个交叉熵取平均，得到 1.979，这就是这句话的损失。训练时，大模型的损失就是这样算的：把训练文本里每一个位置的交叉熵，取平均。

这个数比均匀乱猜的 1.792 还大。也就是说，随机初始化的 TinyLM 不但没学到东西，还带着一些“偏见”（比如偏爱“我”），比完全不知道还差。训练要做的，就是先把它拉到 1.792 以下，再一路往下压。

平均交叉熵是一个对数尺度上的数，不太直观。把它取指数，就得到**困惑度**（perplexity，PPL）：

$$\text{困惑度} = e^{\text{平均交叉熵}} = e^{1.979} \approx 7.24$$

困惑度有一个很好的解释。均匀地在 k 个词里猜，每次正确答案的概率都是 1/k，交叉熵是 ln k，困惑度正好是 k。所以困惑度 7.24 的意思是：**这个模型的“困惑程度”，相当于每一步都在 7.24 个词里均匀地瞎猜**。词表只有 6 个词，它却像是在 7 个多词里猜，又一次说明它比乱猜还差。困惑度最小是 1（每次都给正确答案 100% 的概率），越小越好。

单次预测也可以这样看：“我 → 爱”的交叉熵是 1.572，困惑度 $e^{1.572} = 1 / 0.208 \approx 4.82$，演示里的“困惑度”一行就是这个数。

> **注意**
> 困惑度要从**平均交叉熵**算：先平均，再取指数。先把每个位置的困惑度算出来再平均，得到的是另一个（更大的）数。

用代码复现

下面的代码依次算了：均方误差的例子；几个 p 值下的交叉熵、平方误差和两者的修正信号；TinyLM 对“我”的下一个词的预测，以及例句 4 次预测的交叉熵、平均值和困惑度；最后是 Llama-3 词表的两个工程数字（下一节用）。

```
import json
import numpy as np

t = json.load(open("tinylm.json", encoding="utf-8"))

# 1. 均方误差：四个点的预测和目标
pred, target = np.array([0.9, 0.2, -0.4, 1.3]), np.array([1.0, 0.0, -1.0, 1.0])
err = pred - target
print("误差：", err.round(2), " 均方误差：", round(float(np.mean(err ** 2)), 4))

# 2. 交叉熵 -ln p 和平方误差 (1 - p)²，以及它们给 logit 的修正信号
for p in [0.99, 0.9, 0.5, 0.1, 0.01]:
    print(f"p = {p:<4}  交叉熵 {-np.log(p):.3f}  平方误差 {(1 - p) ** 2:.3f}  "
          f"信号 {p - 1:+.3f} / {-2 * (1 - p) * p * (1 - p):+.4f}")

# 3. TinyLM：只用“当前词的嵌入 × 输出层”预测下一个词
vocab = t["config"]["vocab"]
E, H = np.array(t["weights"]["embedding"]), np.array(t["weights"]["lm_head"])
logits = E @ H                                  # 6×6：每一行是一个“当前词”对 6 个候选词的打分
probs = np.exp(logits) / np.exp(logits).sum(axis=1, keepdims=True)   # softmax，每行加起来是 1
print("“我”→", {w: round(float(v), 3) for w, v in zip(vocab, probs[0])})

seq = ["我", "爱", "学", "AI", "<eos>"]
ids = [vocab.index(w) for w in seq]
losses = [-np.log(probs[a, b]) for a, b in zip(ids[:-1], ids[1:])]
for a, b, l in zip(seq[:-1], seq[1:], losses):
    print(f"  {a} → {b}：p = {np.exp(-l):.3f}，交叉熵 {l:.3f}")
mean = np.mean(losses)
print(f"平均交叉熵 {mean:.3f}，困惑度 {np.exp(mean):.2f}；均匀猜：{np.log(6):.3f}，困惑度 6")
print(np.allclose(losses, t["examples"]["16"]["loss"], atol=1e-4))

# 4. Llama-3 的词表：均匀猜的损失，以及一次存下 8192 个 token 的 logits 要多少显存
V = 128256
print(f"ln V = {np.log(V):.2f}；logits 8192 × {V} × 4 字节 = {8192 * V * 4 / 2**30:.2f} GiB")
```

输出：

```
误差： [-0.1  0.2  0.6  0.3]  均方误差： 0.125
p = 0.99  交叉熵 0.010  平方误差 0.000  信号 -0.010 / -0.0002
p = 0.9   交叉熵 0.105  平方误差 0.010  信号 -0.100 / -0.0180
p = 0.5   交叉熵 0.693  平方误差 0.250  信号 -0.500 / -0.2500
p = 0.1   交叉熵 2.303  平方误差 0.810  信号 -0.900 / -0.1620
p = 0.01  交叉熵 4.605  平方误差 0.980  信号 -0.990 / -0.0196
“我”→ {'我': 0.467, '爱': 0.208, '学': 0.061, 'AI': 0.076, '推理': 0.057, '<eos>':
0.131}
  我 → 爱：p = 0.208，交叉熵 1.572
  爱 → 学：p = 0.098，交叉熵 2.323
  学 → AI：p = 0.202，交叉熵 1.601
  AI → <eos>：p = 0.089，交叉熵 2.421
平均交叉熵 1.979，困惑度 7.24；均匀猜：1.792，困惑度 6
True
ln V = 11.76；logits 8192 × 128256 × 4 字节 = 3.91 GiB
```

逐行对照：均方误差 0.125；p 从 0.99 降到 0.01，交叉熵从 0.010 涨到 4.605，平方误差却最多只到 0.980，修正信号那一列，平方误差在这几个值里 p = 0.5 时最大（−0.25），再往下越错反而越小；“我”后面 6 个词的概率，以及 4 次预测的交叉熵，和上面两张表一致；平均交叉熵 1.979，困惑度 7.24，比均匀猜的 1.792 和 6 还差；最后的 True 表示和 `tinylm.json` 里存的结果一致。

到了大模型里

大模型的预训练，目标只有一个：**在海量文本上，让“预测下一个词”的平均交叉熵尽量小**。前面讲的就是这件事的全部定义，没有别的损失。

训练刚开始时，权重是随机的，模型差不多是在乱猜，损失大约就是 ln V。Llama-3 的词表有 128256 个词，ln 128256 ≈ 11.76，所以训练日志里的损失曲线，通常从 11 左右开始往下掉。看到一个刚开始训练的模型损失远大于 ln V，往往说明初始化或者代码有问题。

计算交叉熵本身也有工程上的讲究。每个位置都要先算出 V 个 logit，再做 softmax。一次训练如果处理 8192 个 token，logit 矩阵就是 8192 × 128256，用 4 字节的浮点数存，要 3.91 GiB，而这还只是一个中间结果。所以训练框架常常把“算 logit、softmax、取正确词的概率、求对数”这几步融合成一个算子，分块计算，不把整个矩阵存下来（第 83 讲的算子融合）。

推理这边也离不开困惑度。第 12 篇做量化时，把权重从 16 位压到 4 位，模型有没有变笨？最常用的检查办法，就是在一份标准文本上算量化前后的困惑度，看它涨了多少。量化论文里经常用 WikiText-2 这份文本来测，7B 左右规模的模型，困惑度大约在 5 到 6 之间；一个好的量化方法，通常只让它涨零点几甚至更少。

容易混淆的地方

1. **对数的底**。这门课和 PyTorch 都用自然对数 ln，损失的单位叫“奈特”（nat）。信息论里常用以 2 为底的对数，单位是“比特”。两者只差一个常数倍（$\ln 2 \approx 0.693$），但读别人的数字时要知道它用的是哪一种。困惑度如果配套计算（以 2 为底就用 2 的幂），结果是一样的。
2. **损失和准确率不是一回事**。准确率只看“概率最大的那个词是不是正确答案”；交叉熵看“正确答案分到了多少概率”。正确答案从 0.2 涨到 0.4，只要还不是最大的，准确率不变，交叉熵却明显下降了。训练用交叉熵，正是因为它对这种“进步”有反应。
3. **为什么叫“交叉熵”**。这个名字来自信息论：两个概率分布 q 和 p 的交叉熵是 $-\Sigma\, q_i \ln p_i$。当 q 是正确答案的“独热”分布（正确的词是 1，其余是 0）时，求和里只剩下一项，就是这一讲的 $-\ln p$。所以两种说法是同一个东西。

回顾

损失函数把“预测错了多少”量成一个数，训练的目标就是让它变小。输出一个数时用均方误差：误差平方再平均，大错罚得重，而且处处光滑。输出概率时用交叉熵：$-\ln$（正确答案的概率），可以理解成“看到答案时有多吃惊”；和 Sigmoid、softmax 配在一起，它的修正信号是简洁的 $p - y$，错得越离谱推得越用力，而平方误差在这时几乎没有信号。语言模型就是一个在整个词表上做分类的模型，一句话的损失是每个位置交叉熵的平均；对它取指数就是困惑度，相当于“每一步在几个词里均匀地猜”。随机初始化的模型，损失大约是 $\ln V$。

现在我们有了一把尺子，也看到了“修正信号”这个东西：它告诉每个 logit 该往哪边调、调多少。可是网络里真正要调的不是 logit，而是成千上万个权重。对每一个权重来说，它该往哪个方向改、改多少，才能让损失下降？下一讲：**梯度与梯度下降**。

第17讲 梯度下降

08 导数与链式法则　16 损失函数

上一讲有了一把尺子：损失函数，把“错了多少”量成一个数。训练的目标是让这个数变小。可是一个网络有成千上万个参数，Llama-3-8B 有 80 亿个，不可能一个一个地去试。我们需要一个办法，能直接告诉每个参数：**往哪个方向改，改多少，损失才会下降**。

这个办法就是**梯度下降**（gradient descent）。它的想法非常朴素：蒙着眼睛站在山坡上想下山，就用脚试一试哪边往下倾，然后往那边迈一小步，再试，再迈。这一讲先只调一个参数，把“斜率”和“步子多大”讲清楚；再调两个参数，引出梯度；最后看看在大模型里，这件事意味着什么。

一个最小的问题

为了能把损失画出来，问题要足够小。有 4 个数据点：

x	0	1	2	3
y	1	3	4	6

我们想用一条直线 $\hat{y} = w \cdot x + b$ 去拟合它们。这其实就是第 13 讲的一个神经元：一个输入、一个权重、一个偏置、不加激活。损失用上一讲的均方误差：

$$L(w, b) = \frac{1}{4} \sum (wx_i + b - y_i)^2$$

注意这里的视角变了。上一讲里，损失是预测 $\hat{y}$ 的函数；现在数据是固定的，**损失是参数 w、b 的函数**。换一组参数，就得到一个不同的损失。训练要找的，就是让 L 最小的那组参数。

这个问题小到可以直接解方程：最好的直线是 $w = 1.6$、$b = 1.1$，损失 0.05（4 个点不在一条直线上，所以损失不会是 0）。我们知道答案，正好可以检验梯度下降能不能自己找到它。

只调一个参数：沿着斜率往下走

先把 b 固定在 1.1，只调 w。这时 L 只是 w 的函数，画出来是一条抛物线，谷底在 $w = 1.6$。

假设我们从 $w=0$ 出发。此时 4 个点的误差 $wx+b-y$ 分别是 0.1、−1.9、−2.9、−4.9，损失是它们平方的平均：9.01。

下一步该往左走还是往右走？看**斜率**（slope），也就是**导数**（derivative）dL/dw：w 增加一点点，L 跟着变化多少倍（第 8 讲）。斜率是负的，说明往右走（w 增大）损失会下降；斜率是正的，就该往左走。总之，**往斜率的反方向走**。

斜率怎么算？L 是 4 项平方的平均，每一项 $(wx+b-y)^2$ 对 w 求导，按链式法则是 $2(wx+b-y)\cdot x$：外层平方的导数是“2 倍里面”，再乘上里面对 w 的导数 x。所以

$$\frac{dL}{dw}=\frac{1}{4}\sum 2\left(wx_i+b-y_i\right)\cdot x_i=\frac{1}{4}\times 2\times\left(0.1\times 0-1.9\times 1-2.9\times 2-4.9\times 3\right)=-$$

斜率是 −11.2，所以往右走。走多远？用斜率乘一个小小的系数 η（读作 eta）：

$$w\leftarrow w-\eta\cdot\frac{dL}{dw}=0-0.05\times(-11.2)=0.56$$

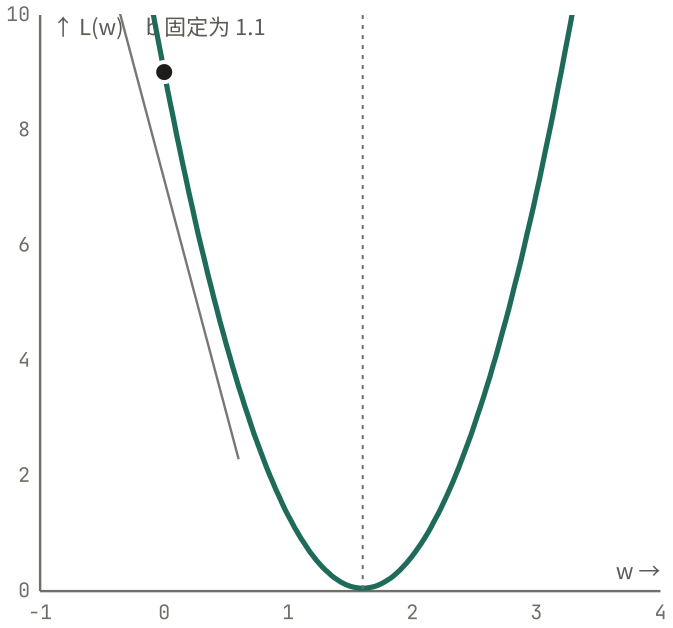
η 叫**学习率**（learning rate），控制每一步迈多大。这里取 0.05，w 从 0 走到了 0.56，损失从 9.01 降到 3.84。在新的位置再算斜率（−7.28），再走一步，w 到 0.924；再一步，1.161……损失 9.01 → 3.84 → 1.65 → 0.73，一路下降。

注意一个现象：越靠近谷底，斜率越小，步子也越小。这是自动发生的：步长是“学习率 × 斜率”，而抛物线越接近底部越平。所以不用担心冲过头，它会自然地慢下来。

下面的演示可以一步一步地走。绿色曲线是 $L(w)$，黑色短线是当前位置的切线，它的倾斜程度就是斜率；琥珀色的点和虚线是走过的每一步。在图上点一下可以换起点。

只调一个参数 w：沿着斜率的反方向走

⇆ 对照



学习率 η = **0.05**

当前　　w = 0, L = 9.01

η 比较小：一步一步从同一侧往谷底挪，越靠近谷底步子越小（因为斜率越来越小）。

走一步　▶ 连续走　重置　　已走 0 步

图 17-1　一个参数的梯度下降。默认 η = 0.05：每一步都从左边往谷底挪。把 η 调到 0.2：每一步跳过谷底，左右来回，但越来越近。调到 0.3 以上：越跳越远，发散。

学习率：太小太慢，太大发散

拖一拖学习率的滑块，会看到三种完全不同的走法。

- **η 比较小（比如 0.05）**：每一步都在谷底的同一侧，稳稳地往下挪。20 步以后 $w = 1.5997$，已经非常接近 1.6。再小下去当然也能到，只是步数要多得多。
- **η 稍大（比如 0.2）**：第一步就冲过了谷底，跑到对面；下一步又冲回来。在两边来回跳，但每次都比上一次近，20 步以后同样到了 1.6000。
- **η 太大（比如 0.3）**：每一跳都跳到对面更高的地方，越跳越远。20 步以后 w 已经到了 −9.164，**发散**了。

这三种情况的分界在哪里？对这条抛物线，可以精确算出来。斜率是 w 的一次函数：$dL/dw = 7(w - 1.6)$，其中 $7 = 2 \times (0^2 + 1^2 + 2^2 + 3^2) / 4$，是抛物线的“弯曲程度”。代进更新公式：

$$w_{新} - 1.6 = (w - 1.6) - \eta \times 7(w - 1.6) = (1 - 7\eta) \times (w - 1.6)$$

也就是说，每走一步，到谷底的距离乘以 $(1 - 7\eta)$。$\eta = 0.05$ 时这个系数是 0.65，距离每步缩小到 65%；$\eta = 0.2$ 时是 −0.4，符号翻转（跳到对面），大小缩到 40%；$\eta = 0.3$ 时是 −1.1，绝对值大于 1，每步离得更远。所以 η 小于 1/7 时单侧下降，在 1/7 到 2/7 之间来回震荡但收敛，大于 2/7 $\approx$ 0.286 就发散。

要点

学习率决定步子大小。太小，走得慢；太大，越过谷底甚至发散。能用的上限，取决于损失曲面有多“弯”：越弯，学习率就得越小。

真实的网络里，曲面的弯曲程度没法事先算出来，学习率只能靠经验和试验来定。它几乎是训练里最重要的一个超参数（不是训练出来的、要人来设定的参数）。第 20 讲还会看到，实际训练中学习率通常不是一个固定值，而是随着训练的进行先升后降。

两个参数：梯度

现在让 w 和 b 一起动。损失 $L(w, b)$ 是两个变量的函数，画出来是一个碗形的曲面。从上往下看，可以画成**等高线图**：每条线上的损失相同，像地图上的等高线。

两个参数，就有两个斜率。固定 b、只看 w 变化时的斜率，记作 $\partial L / \partial w$；固定 w、只看 b 变化时的斜率，记作 $\partial L / \partial b$。这种“其他变量不动、只看一个”的导数叫**偏导数**（partial derivative）。$\partial L / \partial w$ 就是刚才算过的式子；$\partial L / \partial b$ 也一样推，只是里面对 b 的导数是 1：

$$\partial L / \partial w = (1/4) \Sigma\, 2(w\,\mathrm{x_i} + b - \mathrm{y_i}) \cdot \mathrm{x_i}, \qquad \partial L / \partial b = (1/4) \Sigma\, 2(w\,\mathrm{x_i} + b - \mathrm{y_i})$$

把所有偏导数排成一个向量，就是**梯度**（gradient），记作 ∇L（∇ 读作 nabla）：

$$\nabla L = \left(\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b} \right)$$

梯度有一个重要的几何性质：**它指向损失上升最快的方向**，而且垂直于所在位置的等高线。所以往梯度的反方向走，就是下降最快的方向。更新公式和一个参数时一模一样，只是现在每个参数都按自己的偏导数走：

$$(w, b) \leftarrow (w, b) - \eta \cdot \nabla L$$

从 (−0.5, −2) 出发算一下。这时的直线是 $\hat{y} = -0.5x - 2$，4 个点的预测是 −2、−2.5、−3、−3.5，减去 y 得到误差 −3、−5.5、−7、−9.5，损失是 (9 + 30.25 + 49 + 90.25) / 4 = 44.625。梯度的两个分量：

$$\partial L/\partial w = (1/2) \times (-3\times 0 - 5.5\times 1 - 7\times 2 - 9.5\times 3) = (1/2) \times (-48) = -24,$$
$$\partial L/\partial b = (1/2) \times (-3 - 5.5 - 7 - 9.5) = (1/2) \times (-25) = -12.5$$

（式子里的 1/2 来自“平均的 1/4”乘“求导出来的 2”。）

梯度是 (−24, −12.5)，两个分量都是负的，说明 w 和 b 都该增大。用 $\eta = 0.1$ 走一步：

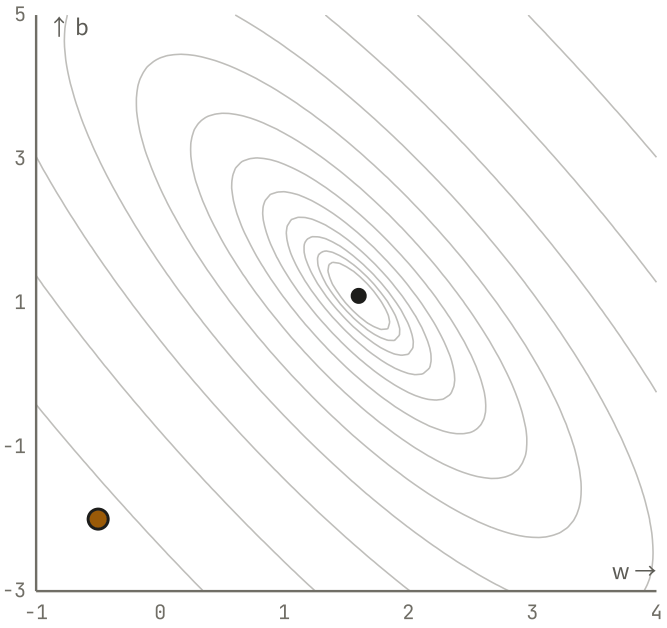
$$(w, b) \leftarrow (-0.5, -2) - 0.1 \times (-24, -12.5) = (1.9, -0.75)$$

损失从 44.625 一下子降到 2.12。接着走：第 10 步 (2.04, 0.16)，损失 0.37；第 50 步 (1.64, 1.02)，损失 0.052；第 200 步，(1.6, 1.1)，损失 0.05，和解方程得到的答案完全一样。

下面的演示画的就是这个过程。灰色的线是等高线，实心小圆是谷底 (1.6, 1.1)，空心圆是起点，琥珀色的折线是走过的路。点“走一步”，右边会写出这一步的梯度和更新。

在 loss 曲面上一步步往下走

⇆ 对照



学习率 η = 0.1

随机小批量：每步只用 1 个样本算梯度

	(w, b)	L
梯度下降	(-0.5, -2)	44.625

最低点在 (1.6, 1.1)，L = 0.05（实心小圆）。等高线越密，坡越陡。点图上任意位置换起点。

走一步　▶ 连续走　重置　　已走 0 步

图 17-2　两个参数的梯度下降。每一步都垂直于等高线、朝下坡最陡的方向走。注意路线是先冲向“谷底那条线”，再沿着山谷慢慢挪向最低点。

仔细看路线，会发现它不是一条直奔谷底的直线：第一步几乎垂直地冲向了山谷的“底线”附近，然后沿着山谷的方向一点点挪过去。原因是这个碗不是圆的，而是有点扁：沿某个方向很陡，沿另一个方向很平。梯度只管“眼下最陡”，于是先在陡的方向上把损失降下来，平的方向上进展就慢。把学习率调到 0.12 以上，在陡的方向上就会来回震荡；超过约 0.24，就会发散。这个界限比一个参数时的 2/7 ≈ 0.286 还小，因为两个参数一起动时，最陡的那个方向比单看 w 更弯。这个“扁碗”问题，第 19 讲还会专门来解决。

用代码复现

下面的代码依次算了：只调 w 时的前几步；三个学习率各走 20 步的结果；两个参数一起走 200 步；最后用解方程的方法得到的最优解作对照。

```python
import numpy as np

x = np.array([0.0, 1.0, 2.0, 3.0])
y = np.array([1.0, 3.0, 4.0, 6.0])

def loss(w, b):
    return np.mean((w * x + b - y) ** 2)

def grad(w, b):                                    # 两个偏导数
    r = w * x + b - y                              # 每个点的误差
    return np.mean(2 * r * x), np.mean(2 * r)      # (∂L/∂w, ∂L/∂b)

# 1. 只调 w（b 固定为 1.1），学习率 0.05
w = 0.0
for t in range(4):
    g = grad(w, 1.1)[0]
    print(f"第 {t} 步：w = {w:.4f}，L = {loss(w, 1.1):.4f}，斜率 = {g:.4f}")
    w = w - 0.05 * g

# 2. 学习率不同，走 20 步以后离谷底 w = 1.6 有多远
for lr in [0.05, 0.2, 0.3]:
    w = 0.0
    for _ in range(20):
        w -= lr * grad(w, 1.1)[0]
    print(f"η = {lr}：20 步后 w = {w:.4f}")

# 3. 两个参数一起：从 (-0.5, -2) 出发，学习率 0.1
w, b = -0.5, -2.0
gw, gb = grad(w, b)
print(f"起点 L = {loss(w, b):.4f}，梯度 = ({gw:.2f}，{gb:.2f})")
for t in range(1, 201):
    gw, gb = grad(w, b)
    w, b = w - 0.1 * gw, b - 0.1 * gb
    if t in (1, 10, 50, 200):
        print(f"第 {t:3d} 步：(w, b) = ({w:.4f}，{b:.4f})，L = {loss(w, b):.4f}")

# 4. 对照：直接解方程得到的最优解
A = np.column_stack([x, np.ones(4)])
print("最小二乘解：", np.linalg.lstsq(A, y, rcond=None)[0].round(4), " 最小损失：",
round(loss(1.6, 1.1), 4))
```

输出：

```
第 0 步：w = 0.0000，L = 9.0100，斜率 = -11.2000
第 1 步：w = 0.5600，L = 3.8356，斜率 = -7.2800
第 2 步：w = 0.9240，L = 1.6494，斜率 = -4.7320
第 3 步：w = 1.1606，L = 0.7258，斜率 = -3.0758
η = 0.05：20 步后 w = 1.5997
η = 0.2：20 步后 w = 1.6000
η = 0.3：20 步后 w = -9.1640
起点 L = 44.6250，梯度 = (-24.00, -12.50)
第   1 步：(w, b) = (1.9000, -0.7500)，L = 2.1225
第  10 步：(w, b) = (2.0403, 0.1600)，L = 0.3704
第  50 步：(w, b) = (1.6379, 1.0191)，L = 0.0524
第 200 步：(w, b) = (1.6000, 1.1000)，L = 0.0500
最小二乘解：  [1.6 1.1]  最小损失：  0.05
```

逐行对照：只调 w 时，斜率 −11.2 → −7.28 → −4.732 → −3.076，越来越接近 0，w 一步步走向 1.6，损失 9.01 → 3.84 → 1.65 → 0.73。三个学习率：0.05 和 0.2 都到了 1.6，0.3 发散到 −9.164。两个参数：起点损失 44.625，梯度 (−24, −12.5)，第一步就到了 (1.9, −0.75)，第 200 步和最小二乘的解 (1.6, 1.1) 完全一致，最小损失 0.05。

到了大模型里

这一讲的两个参数，换成 Llama-3-8B 就是 80 亿个。梯度下降的公式一个字都不用改：θ 是一个 80 亿维的向量，梯度 ∇L 也是一个 80 亿维的向量，每个参数按自己的偏导数走一小步。

由此马上得到一个工程上的结论：**训练比推理费显存得多**。推理只要存权重；训练还要为每个参数存一个梯度，梯度和权重一样多。Llama-3-8B 的权重用 BF16 存是 16 GB，梯度同样是 16 GB，这还没算第 19 讲优化器要额外保存的东西，也没算第 18 讲反向传播需要的中间结果。

还有一个问题：梯度是在数据上算出来的。这一讲的数据只有 4 个点，每一步都用全部 4 个点算梯度。大模型的训练数据有上万亿个 token，不可能每走一步都把全部数据过一遍。实际的做法是每一步只取一小批数据（一个 batch）来估计梯度，这会让梯度带上噪声，第 19、20 讲会讲它的影响。用多张 GPU 训练时，每张卡各算一部分数据的梯度，然后把梯度加起来求平均，这一步叫 All-Reduce，是分布式训练里最主要的通信（第 100、103 讲）。

容易混淆的地方

1. **梯度指向“上坡”，下降要取负号**。梯度是损失增长最快的方向，更新时要减去它。代码里写成加号，模型就会越训越差。

2. **梯度是对参数求的，不是对输入**。这一讲的 x 和 y 是固定的数据，求导的对象是 w 和 b。第 18 讲会看到，反向传播的过程中也会顺带算出对中间结果的导数，但最后用来更新的，只有对参数的那部分。
3. **学习率不是越小越好**。太小的学习率不仅慢，在真实的网络里还可能卡在一些平坦的地方出不来。学习率要在“不发散”的前提下尽量大一些，这正是调学习率难的地方。

回顾

训练要让损失变小，而梯度告诉我们怎么改参数：对每个参数求偏导数，排成梯度向量 ∇L，它指向损失上升最快的方向；沿着它的反方向走一小步，就是梯度下降 $\theta \leftarrow \theta - \eta \nabla L$。步子大小由学习率 η 控制：太小走得慢，稍大会来回震荡，超过某个界限就发散，这个界限取决于损失曲面有多弯。在我们的小例子里，梯度下降从任意起点出发，都能走到解方程得到的最优解 (1.6, 1.1)。在大模型里，梯度和参数一样多，这让训练比推理费显存得多。

这一讲的梯度能直接写出公式，是因为模型只有一层、只有一个神经元。可一个真正的网络有很多层，参数藏在一层套一层的函数里面，每个参数离损失隔着好几层计算。80 亿个偏导数，怎么才能高效地全部算出来？下一讲：**反向传播**。

第18讲 反向传播

08 导数与链式法则　15 多层感知机　17 梯度下降

上一讲的梯度下降需要一样东西：损失对每一个参数的偏导数。那一讲的模型只有一个神经元，偏导数可以直接写出公式。可第 15 讲的网络已经有两个隐藏层、97 个参数，第一层的权重要经过 ReLU、第二层、又一个 ReLU、输出层，才影响到损失。到了 Llama-3-8B，有 80 亿个参数，每个参数和损失之间隔着几十层计算。

最笨的办法是一个一个地试：把某个参数挪一点点，重新算一遍损失，看变了多少。80 亿个参数就要算 80 亿零 1 遍，根本不可能。这一讲介绍的**反向传播**（backpropagation），只需要**一次前向、一次反向**，就能把所有参数的偏导数全部算出来，代价只是前向的两倍左右。它是所有神经网络训练的基础，PyTorch 里那一句 `loss.backward()` 做的就是这件事。

把计算拆成一张图

先拿一个足够小、又能看清结构的网络：一个输入 x，经过两个神经元，输出 $\hat{y}$，再和答案 y 算平方误差：

$$z_1 = w_1 x + b_1, \quad h = \mathrm{ReLU}(z_1), \quad \hat{y} = w_2 h + b_2, \quad e = \hat{y} - y, \quad L = e^2$$

这是第 15 讲那种多层网络的最小版本：宽度 1、两层。有 4 个参数 w_1、b_1、w_2、b_2。

把每一个运算画成一个方框，运算的输入画成指向它的箭头，就得到一张**计算图**（computational graph）。图里每个方框只做一件很简单的事：乘、加、减、ReLU 或者平方。整个网络，就是这些小运算一个接一个地串起来。

取 x = 1.5、y = 2，参数 $w_1 = 0.8$、$b_1 = -0.2$、$w_2 = 1.5$、$b_2 = 0.3$，从左往右依次算，这一步叫**前向传播**（forward pass）：

节点	算式	值
w_1x	0.8×1.5	1.2
z_1	$1.2 + (-0.2)$	1.0
h	$\max(0, 1.0)$	1.0
w_2h	1.5×1.0	1.5
$\hat{y}$	$1.5 + 0.3$	1.8
e	$1.8 - 2$	−0.2
L	$(-0.2)^2$	0.04

预测是 1.8，答案是 2，损失 0.04。

链式法则：一段一段地乘

现在要求 $\partial L/\partial w_2$：w_2 动一点点，L 变多少倍？

w_2 不直接影响 L。它先影响 w_2h，再影响 $\hat{y}$，再影响 e，最后才影响 L。好在每一段的影响都很好算，这叫**局部导数**：

- $L = e^2$，所以 $\partial L/\partial e = 2e = -0.4$：$e$ 增加一点，L 变化 −0.4 倍。
- $e = \hat{y} - y$，所以 $\partial e/\partial \hat{y} = 1$。
- $\hat{y} = w_2h + b_2$，所以 $\partial \hat{y}/\partial w_2 = h = 1.0$。

链式法则（chain rule，第 8 讲）说：一串影响接起来，总的倍数就是每一段倍数相乘。想象 w_2 增加了 0.001：$\hat{y}$ 增加 0.001×1.0；e 跟着增加同样多；L 再变化它的 −0.4 倍，总共是 $0.001 \times 1.0 \times 1 \times (-0.4)$。所以

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial e} \times \frac{\partial e}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w_2} = (-0.4) \times 1 \times 1.0 = -0.4$$

再求离得更远的 $\partial L/\partial w_1$。路径是 $w_1 \to w_1$x $\to z_1 \to h \to w_2h \to \hat{y} \to e \to L$，每一段的局部导数是：x = 1.5、1、ReLU 的斜率 1（因为 $z_1 = 1.0 > 0$，第 14 讲）、w_2 = 1.5、1、1、$2e$ = −0.4。全部乘起来：

$$\frac{\partial L}{\partial w_1} = (-0.4) \times 1 \times 1 \times 1.5 \times 1 \times 1 \times 1.5 = -0.9$$

两个式子里，后半段 $(-0.4) \times 1 \times 1$ 是一样的：从 L 走回 $\hat{y}$ 这一段，两条路径共用。这一点就是反向传播高效的秘密。

从终点往回走

如果为每个参数都从头把整条路径乘一遍，越靠前的参数路径越长，而且大量的乘法是重复的。反向传播换了一个顺序：**从 L 出发，往回一个节点一个节点地算，每个节点只算一次**。

规则只有一条。假设已经知道了 $\partial L / \partial u$（u 是某个节点的输出），而 u 是用 v 算出来的，那么

$$\frac{\partial L}{\partial v} = \frac{\partial L}{\partial u} \times \frac{\partial u}{\partial v}$$

$\partial L / \partial u$ 是从后面传过来的，叫**上游梯度**；$\partial u / \partial v$ 是这个节点自己的局部导数，只和这个节点的运算和它的输入值有关。两者一乘，就得到往前一个节点的梯度，再继续往前传。所以每个节点只要会做两件事：前向时根据输入算输出；反向时根据上游梯度算出给每个输入的梯度。

几种运算的局部导数都很简单：

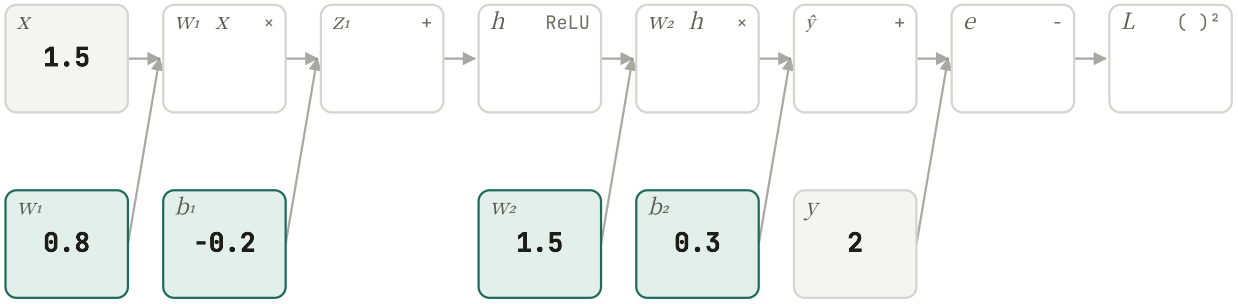
运算	前向	反向：给输入的梯度
加 $u = a + b$	相加	上游梯度原样分给 a 和 b
乘 $u = a \times b$	相乘	给 a 的是“上游 $\times$ b”，给 b 的是“上游 $\times$ a”
ReLU	$\max(0, z)$	$z > 0$ 时原样传过去，否则传 0
平方 $u = e^2$	平方	上游 $\times$ $2e$

乘法这一行值得注意：反向时要用到前向时**另一个输入的值**。ReLU 也一样，要知道前向时 z 是正是负。所以前向传播时，中间结果不能算完就扔，得留着给反向用。这一点在大模型里会变成一个大问题，后面再说。

下面的演示把整个过程拆成一步一步。先点“下一步”走完 7 步前向，每个方框里出现黑色的值；接着是 11 步反向，从 L 开始，每个方框下面出现琥珀色的梯度，下方写出这一步的“上游梯度 $\times$ 局部导数”。绿色的方框是参数，灰底的是数据。左下方可以改输入和参数的值，所有数字会重新算。

计算图：先往前算值，再往回算梯度

⇆ 对照



x 1.5 w₁ 0.8 b₁ -0.2 w₂ 1.5 b₂ 0.3

y 2

上一步 下一步 ▶ 播放 全部算完 重置

点“下一步”开始。黑色数字是节点的值，琥珀色的“∂”是损失对这个节点的梯度 ∂L/∂节点。

图 18-1　一个两层小网络的计算图。前向从左往右算值；反向从 L 开始往左算梯度，每一步都是“上游梯度 × 局部导数”。最后 4 个参数的梯度是 $\partial L/\partial w_1 = -0.9$、$\partial L/\partial b_1 = -0.6$、$\partial L/\partial w_2 = -0.4$、$\partial L/\partial b_2 = -0.4$。

反向走完，4 个参数的梯度是：

$$\frac{\partial L}{\partial w_1} = -0.9, \quad \frac{\partial L}{\partial b_1} = -0.6, \quad \frac{\partial L}{\partial w_2} = -0.4, \quad \frac{\partial L}{\partial b_2} = -0.4$$

全是负数，说明 4 个参数都该增大一点，这样 $\hat{y}$ 会从 1.8 往 2 靠。这正符合直觉：预测小了，就把能让预测变大的参数都往上调。

可以在演示里做两个实验。一是把 b_1 改成 −2，让 z_1 变成负数：ReLU 输出 0，反向时 ReLU 也把梯度截成 0，于是 w_1 和 b_1 的梯度都变成 0，它们得不到任何修正信号（w_2 的梯度也是 0，因为它乘的 h 是 0；只剩 b_2 还能动）。这正是第 14 讲说的“死亡 ReLU”。二是把 y 改成 1.8，让预测恰好等于答案：$e = 0$，所有梯度都是 0，因为已经没有可以改进的地方了。

要点

反向传播 = 从损失出发，沿着计算图往回走，每个节点做一次“上游梯度 × 局部导数”。一次前向加一次反向，就得到所有参数的梯度。

如果一个节点的输出被用了不止一次（比如同一个 h 同时送给了两个神经元），它就会从两个方向收到上游梯度。这时把它们**加起来**就行：h 动一点，会通过两条路同时影响损失，总的影响是两条路之和。演示组件也是这样处理的。

用数值核对

反向传播算出来的梯度对不对？有一个朴素的检验办法：把一个参数真的挪一点点，比如加 0.000001，重新前向一遍，看损失变了多少，再除以 0.000001。这叫**数值梯度**。下面的代码对 4 个参数都这样核对了一遍，结果和反向传播完全一致。

数值梯度简单可靠，为什么不直接用它来训练？因为它每核对一个参数，就要多做一次完整的前向。4 个参数要 5 次前向；80 亿个参数要 80 亿零 1 次。反向传播只要一次前向、一次反向。实际中，数值梯度只用来检查自己手写的反向代码有没有写错。

一整层的反向：还是矩阵乘法

真实的网络里，节点不是一个个数，而是一整个矩阵。最重要的一种是线性层（第 13 讲）$Y = XW$：X 是 s × d_{in}（s 个词，每个 d_{in} 维），W 是 $d_{\text{in}} \times d_{\text{out}}$，$Y$ 是 s × d_{out}。反向时，已经知道了损失对 Y 每一个元素的梯度，排成和 Y 一样形状的矩阵，记作 dY。要求的是 $\partial L/\partial W$（更新权重用）和 $\partial L/\partial X$（继续往前一层传）。

只看一个元素就能推出来。Y 的第 i 行第 j 列是 $Y_{\text{ij}} = \Sigma_{\text{k}}\, X_{\text{ik}}\, W_{\text{kj}}$。权重 W_{kj} 出现在第 j 列的每一个 Y_{ij} 里（i = 1⋯s），每处的局部导数都是 X_{ik}。被用了 s 次，梯度就要把 s 份加起来：

$$\frac{\partial L}{\partial W_{kj}} = \sum_{i} X_{ik} \cdot \mathrm{d}Y_{ij} \quad \Rightarrow \quad \frac{\partial L}{\partial W} = X^{T} \cdot \mathrm{d}Y$$

这里的 X^{T} 是 X 的**转置**：行列互换，变成 d_{in} × s。同样的道理，X_{ik} 出现在第 i 行的每一个 Y_{ij} 里，局部导数是 W_{kj}：

$$\frac{\partial L}{\partial X_{ik}} = \sum_j \mathrm{d}Y_{ij} \cdot W_{kj} \quad \Rightarrow \quad \frac{\partial L}{\partial X} = \mathrm{d}Y \cdot W^T$$

检查一下形状：X^{T}（$d_{\text{in}} \times \mathrm{s}$）乘 $\mathrm{d}Y$（$\mathrm{s} \times d_{\text{out}}$）得到 $d_{\text{in}} \times d_{\text{out}}$，和 W 一样；$\mathrm{d}Y$（$\mathrm{s} \times d_{\text{out}}$）乘 W^{T}（$d_{\text{out}} \times d_{\text{in}}$）得到 $\mathrm{s} \times d_{\text{in}}$，和 X 一样。梯度的形状总是和对应的变量一样，这是检查反向公式最快的办法。

其他部件也一样简单。ReLU 的反向是逐元素的：前向时为正的位置，梯度原样通过，其余位置置 0。偏置 $\mathbf{b}$ 被加到了每一行上，反向时就把 $\mathrm{d}Y$ 按行加起来，得到 $\mathbf{b}$ 的梯度。

所以，**前向的一次矩阵乘法，在反向时变成两次矩阵乘法**：一次算权重的梯度，一次算输入的梯度。这决定了训练的计算量。

用代码复现

下面的代码先手写了小网络的前向和反向，再用数值梯度核对；然后用 TinyLM 的 X（4×4）和 W（4×8）验证线性层的两个反向公式；最后算 Llama-3-8B 训练一个 token 的计算量。

```python
import json
import numpy as np

# 1. 两层小网络：前向把中间值都记下来，反向从 L 往回乘局部导数
x, y = 1.5, 2.0
w1, b1, w2, b2 = 0.8, -0.2, 1.5, 0.3

def forward(w1, b1, w2, b2):
    z1 = w1 * x + b1
    h = max(0.0, z1)
    yh = w2 * h + b2
    e = yh - y
    return z1, h, yh, e, e ** 2

z1, h, yh, e, L = forward(w1, b1, w2, b2)
print(f"前向：z1 = {z1:.4g}, h = {h:.4g}, ŷ = {yh:.4g}, e = {e:.4g}, L = {L:.4g}")

dL_de = 2 * e                       # L = e²
dL_dyh = dL_de * 1                  # e = ŷ - y
dL_dw2, dL_db2 = dL_dyh * h, dL_dyh * 1
dL_dh = dL_dyh * w2                 # ŷ = w2·h + b2
dL_dz1 = dL_dh * (1.0 if z1 > 0 else 0.0)
dL_dw1, dL_db1 = dL_dz1 * x, dL_dz1 * 1
grads = [dL_dw1, dL_db1, dL_dw2, dL_db2]
print("反向：∂L/∂(w1, b1, w2, b2) =", [round(g, 4) for g in grads])

# 2. 用“每个参数挪一点点，看 L 变多少”来核对
eps, params = 1e-6, [w1, b1, w2, b2]
num = []
for i in range(4):
    p = list(params); p[i] += eps
    num.append((forward(*p)[-1] - L) / eps)
print("数值核对：", [round(g, 4) for g in num])

# 3. 线性层 Y = X·W 的反向：∂L/∂W = Xᵀ·dY，∂L/∂X = dY·Wᵀ
ex = json.load(open("tinylm.json", encoding="utf-8"))["examples"]["13"]
X, W = np.array(ex["X"]), np.array(ex["W"])        # 4×4、4×8
Y = X @ W
lossf = lambda X, W: np.mean((X @ W) ** 2)         # 随便取一个损失：Y 的平方平均
dY = 2 * Y / Y.size                                 # ∂L/∂Y，形状和 Y 一样 4×8
dW, dX = X.T @ dY, dY @ W.T                         # 两次矩阵乘法
Wp = W.copy(); Wp[2, 5] += eps
Xp = X.copy(); Xp[1, 3] += eps
print("形状：", dY.shape, dW.shape, dX.shape)
print("∂L/∂W[2,5] 公式 vs 数值：", round(dW[2, 5], 6), round((lossf(X, Wp) -
lossf(X, W)) / eps, 6))
```

```
print("∂L/∂X[1,3] 公式 vs 数值: ", round(dX[1, 3], 6), round((lossf(Xp, W) -
lossf(X, W)) / eps, 6))

# 4. 训练一个 token 的计算量：前向 2N，反向 4N
N = 8.03e9
print(f"Llama-3-8B 每个 token：前向 {2 * N:.3g}，反向 {4 * N:.3g}，合计 {6 * N:.3g}
FLOPs")
```

输出：

```
前向：z1 = 1，h = 1，ŷ = 1.8，e = -0.2，L = 0.04
反向：∂L/∂(w1, b1, w2, b2) = [-0.9, -0.6, -0.4, -0.4]
数值核对： [-0.9, -0.6, -0.4, -0.4]
形状： (4, 8) (4, 8) (4, 4)
∂L/∂W[2,5] 公式 vs 数值： -0.004 -0.004
∂L/∂X[1,3] 公式 vs 数值： -0.117 -0.117
Llama-3-8B 每个 token：前向 1.61e+10，反向 3.21e+10，合计 4.82e+10 FLOPs
```

前两行和演示一致：前向得到 $L = 0.04$，反向得到 4 个梯度 $(-0.9, -0.6, -0.4, -0.4)$，数值核对的结果一模一样。接下来，$\mathrm{d}Y$、$\partial L/\partial W$ 的形状都是 4×8，$\partial L/\partial X$ 是 4×4，分别和 Y、W、X 一致；随便挑的两个元素，用公式算和用数值核对，结果相同。最后一行是下一节要用的数。

到了大模型里

计算量：训练是推理的 3 倍。第 3 讲说过，一次 $\mathrm{s}\times d_{\text{in}}$ 乘 $d_{\text{in}}\times d_{\text{out}}$ 的矩阵乘法要 $2\cdot\mathrm{s}\cdot d_{\text{in}}\cdot d_{\text{out}}$ 次浮点运算，对每个词来说，就是“2 × 这个矩阵的参数个数”。模型的计算量几乎全在线性层里，所以前向处理一个词大约要 2N 次运算（N 是参数量）。反向时每次矩阵乘法变成两次，大约 4N。合起来，**训练一个词约 6N 次浮点运算**。Llama-3-8B 的 N 约 80.3 亿，每训练一个词约 4.82×10^{10} 次。这个“6N”是估算训练成本最常用的公式，第 20 讲会用它算一算 Llama-3 的训练账。

显存：前向的中间结果要留着。反向时，乘法要用另一个输入的值，ReLU 要知道当初的正负，所以前向的中间结果（叫**激活值**，activation）都得存下来，直到反向用完才能释放。这些东西的量很大。只举一项：Llama-3-8B 每一层的前馈网络里都有一个 14336 维的中间向量，训练时如果一批处理 8192 个词，用 BF16 存，32 层光这一项就是 14336×2 字节 $\times\,8192\times32\approx7.0$ GiB，而每一层要存的中间结果远不止这一个。

常见的省法叫**激活重计算**（activation checkpointing）：前向时只存每一层的输入，其他中间结果扔掉；反向到这一层时，再从输入把这一层的前向重新算一遍。用大约多一次前向的计算，换来大量显存。

推理就没有这些负担：不需要反向，中间结果用完就可以扔，只有第 9 篇要讲的 KV Cache 需要留着。这也是同一个模型，推理所需的显存和算力都远小于训练的原因。

容易混淆的地方

1. **反向传播和梯度下降是两件事**。反向传播只负责**算出**梯度；梯度下降（或第 19 讲的其他优化器）负责**用**梯度更新参数。一个训练步是“前向 → 反向（算梯度）→ 更新参数”。
2. **梯度会累加**。一个节点被用了几次，梯度就要加几次。PyTorch 利用了这一点：每次 `backward()` 都把新算出的梯度加到参数原有的梯度上，而不是覆盖。所以每一步更新之前要先把梯度清零，这就是训练代码里常见的 `optimizer.zero_grad()`。第 20 讲会看到它在训练循环里的位置。
3. **“反向”不是把网络倒过来运行**。反向传播并不是从输出算回输入，而是从损失出发，算损失对每个中间值和参数的**导数**。前向算的是值，反向算的是“这个值变一点，损失变多少”。

回顾

网络是一张由简单运算串成的计算图。前向传播从左往右算出每个节点的值，并把中间结果留下来；反向传播从损失出发往回走，每个节点做一次“上游梯度 × 局部导数”，把梯度传给它的输入。一个节点被用了几次，梯度就加几次。这样，一次前向加一次反向，就得到了所有参数的梯度，和逐个挪动参数得到的数值梯度完全一致。对线性层 $Y = XW$，反向是两次矩阵乘法：$\partial L / \partial W = X^{\mathrm{T}} \cdot \mathrm{d}Y$，$\partial L / \partial X = \mathrm{d}Y \cdot W^{\mathrm{T}}$。所以训练一个词约需 6N 次浮点运算，是推理的 3 倍，还要存下大量激活值。

有了梯度，第 17 讲的梯度下降就能用在任何网络上了。可那一讲也看到了：在一个“扁”的碗里，梯度下降会在陡的方向来回震荡，在平的方向上走得很慢；每次只用一小批数据算梯度，还会带来噪声。有没有比“沿着梯度走一步”更聪明的走法？下一讲：**优化器**。

第19讲 优化器

17 梯度下降　18 反向传播

有了反向传播，任何网络的梯度都能算出来了。接下来要决定的是：**拿到梯度之后，参数具体怎么改？**第 17 讲的答案是最简单的一种：沿着梯度的反方向走一步，步长是学习率乘梯度。那一讲的结尾也留下了两个麻烦：在一个“扁”的碗里，它会在陡的方向来回震荡，在平的方向上慢慢挪；实际训练中每步只用一小批数据算梯度，梯度还带着噪声。

决定“拿到梯度后怎么更新参数”的那部分程序，叫**优化器**（optimizer）。这一讲在同一个曲面上比较三种：最朴素的 SGD、带惯性的 Momentum，以及大模型训练几乎都在用的 Adam，看它们各自解决了什么问题，又要付出什么代价。

同一组数据，换一个碗

还是第 17 讲那 4 个点，只把 x 整体加 2，变成 (2, 3, 4, 5)，y 不变。最好的直线还是斜率 1.6，只是截距变成了 $1.1 - 1.6 \times 2 = -2.1$。问题本质上一点没变。

可是损失曲面变了很多。把曲面在各个方向上有多“弯”算出来（下面的代码会算）：原来的数据，最陡的方向和最平的方向，弯曲程度相差 14 倍；x 平移 2 以后，相差 166 倍。碗被压成了一条又窄又长的山谷。

这对梯度下降很不友好。第 17 讲说过，学习率的上限由**最弯**的方向决定：原来是 0.238，现在只有 0.069。而在**最平**的方向上，每一步的进展又和那里的坡度成正比，坡度很小，进展就很慢。学习率被陡的方向卡住，又要在平的方向上走很远，结果就是：用 0.03 的学习率走 300 步，损失还停在 0.127，离最低的 0.05 差得很远；就算把学习率开到接近上限的 0.065，也要 228 步。作为对比，原来的数据用 0.1 的学习率只要 39 步。

为什么平移一下 x 就会这样？粗略地说，x 都是正数、而且离 0 比较远时，w 和 b 的作用纠缠在一起：w 增加一点，几乎相当于 b 增加 3.5 倍那么多。两个参数“绑”在一起，就形成了一条斜斜的窄谷。真实的网络里，各个参数的尺度、各层输入的分布千差万别，这样的窄谷到处都是。解决的思路有两种：一是把数据“摆正”，让输入居中、尺度统一，这是第 30 讲归一化要做的事；二是换一个更聪明的优化器，这是这一讲的内容。

SGD：每次只看一小批

先给最朴素的方法正个名。第 17 讲的“沿梯度反方向走一步”，在实际训练中是这样用的：每一步不用全部数据，而是随机抽一小批（一个 mini-batch）算梯度，然后更新：

$$\theta \leftarrow \theta - \eta \cdot g, \qquad g = \text{这一小批数据上的梯度}$$

这叫**随机梯度下降**（stochastic gradient descent，SGD）。“随机”指的是每步用哪些数据是随机抽的，所以算出来的梯度只是全体数据梯度的一个**估计**，有时偏这边，有时偏那边。平均来看方向是对的，但每一步都带着噪声。

这样做是不得已：大模型的训练数据有上万亿个 token，每走一步都把它们全过一遍是不可能的。好在噪声也不全是坏事，它在一定程度上能帮助参数跳出一些不好的平坦区域。习惯上，这个最朴素的更新规则不管用不用小批量，都叫 SGD。

Momentum：带上惯性

在窄谷里，SGD 的问题是：陡的方向上，梯度一会儿指向左壁、一会儿指向右壁，来回抵消；平的方向上，梯度每次都指向同一边，但很小。

Momentum（动量）的想法是让参数像一个有质量的小球那样滚下去。小球不会每一刻都完全听当前坡度的指挥，它有惯性：之前的速度会保留下来。写成式子，多了一个“速度”v：

$$v \leftarrow \beta \cdot v + g, \qquad \theta \leftarrow \theta - \eta \cdot v$$

β 通常取 0.9：每一步，旧的速度保留 90%，再加上当前的梯度。这样一来，

- 在**平的方向**上，梯度每次都指向同一边，速度不断累积。如果梯度一直是 g，速度最终会到 $g / (1-\beta) = 10g$，相当于在这个方向上把学习率放大了 10 倍。
- 在**陡的方向**上，梯度正负交替，加进速度时互相抵消，震荡被压了下去。

速度是对过去梯度的一种平均，所以它也顺带平滑了小批量带来的噪声。效果很明显：同样的学习率 0.03，SGD 走 300 步还没到，Momentum 只用 39 步就把损失降到了 0.06 以下，300 步后精确地停在 (1.6, −2.1)。

Adam：每个参数自己的步长

Momentum 解决了“一致的方向要加速”，但所有参数还是共用一个学习率。窄谷的根本问题是**不同方向的尺度差别太大**：陡的方向梯度大，平的方向梯度小。如果每个参数能按照自己梯度的大小，自动调整步长就好了。

Adam（Adaptive Moment Estimation）就是这样做的。它为每个参数维护两个量：

$$\boldsymbol{m} \leftarrow 0.9 \cdot \boldsymbol{m} + 0.1 \cdot \boldsymbol{g}, \qquad \boldsymbol{v} \leftarrow 0.999 \cdot \boldsymbol{v} + 0.001 \cdot \boldsymbol{g}^2$$

m 是梯度的滑动平均，作用和 Momentum 的速度类似，代表“最近一段时间梯度的方向”；v 是梯度**平方**的滑动平均，代表“这个参数的梯度一般有多大”。这里的平方是逐元素的，每个参数各算各的。更新时，用 m 除以 v 的平方根：

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \cdot \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

除以 $\sqrt{v}$ 是关键。一个参数的梯度一直很大，$\sqrt{v}$ 也大，步长就被缩小；梯度一直很小，$\sqrt{v}$ 也小，步长就被放大。结果是**每个参数走的距离差不多都在 η 这个量级**，不管它的梯度本身是大是小。窄谷里陡的方向和平的方向，被自动拉到了同一个尺度上。

式子里还有两个小细节。$\hat{m}$ 和 $\hat{v}$ 是“偏差修正”后的值：m 和 v 都从 0 开始，刚开始几步会偏小，所以分别除以 $1 - 0.9^{\mathrm{t}}$ 和 $1 - 0.999^{\mathrm{t}}$（t 是步数）把它们放大回来。ε 是一个很小的数（比如 10^{-8}），防止除以 0。

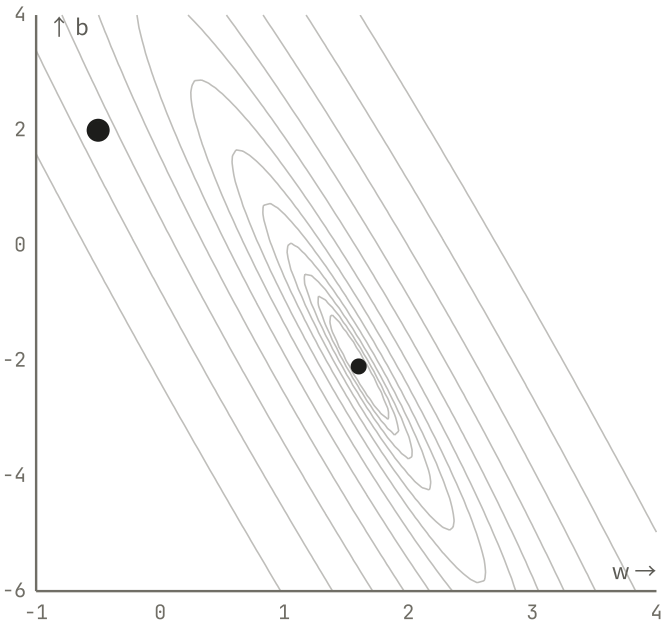
看第一步最清楚。起点的梯度是 (−28, −6.5)，w 方向的梯度是 b 方向的 4 倍多。SGD 用 0.03 的学习率，两个参数分别变化 0.84 和 0.195，步长也差了 4 倍多。Adam 第一步时，$\hat{m}$ 就是 g，$\sqrt{\hat{v}}$ 就是 |g|，两者相除只剩下符号，于是两个参数**都正好走了 0.3**，也就是学习率那么远。

在同一个山谷里比一比

下面的演示把三种优化器放在同一个窄谷里，从同一个起点 (−0.5, 2) 出发。琥珀色是 SGD，绿色是 Momentum，黑色虚线是 Adam。SGD 和 Momentum 共用上面的学习率，Adam 有自己的学习率（三者的学习率含义不同，不能直接比大小）。点“连续走”看它们的轨迹。

同一个曲面，三种优化器

⇆ 对照



学习率 η = **0.03**

Adam 的 η = **0.3**

SGD　Momentum　Adam

☐ 随机小批量：每步只用 1 个样本算梯度

优化器	(w, b)	L
SGD	(-0.5, 2)	16.125
Momentum	(-0.5, 2)	16.125
Adam	(-0.5, 2)	16.125

最低点在 (1.6, −2.1)，L = 0.05（实心小圆）。等高线越密，坡越陡。点图上任意位置换起点。

走一步　▶ 连续走　重置　　已走 0 步

图 19-1　狭长山谷里的三种优化器。SGD 先冲到谷底那条线上，再沿着山谷慢慢爬；Momentum 带着惯性冲过头又折回来，但很快沿山谷加速；Adam 一开始两个方向各走一样远，路线最直。勾选“随机小批量”，每步只用一个样本，看噪声的影响：Momentum 在这个学习率下会被噪声带得越摆越大。

几个值得看的地方：

- SGD：第一步几乎是横着冲向山谷中线（因为那个方向陡），之后沿着谷底一点一点往右下方挪，300 步还没到。把学习率往上拉，超过 0.069 就会在陡的方向上来回震荡，最后发散。
- Momentum：前几步冲过了谷底，在两侧壁之间摆了几下，但摆动很快衰减，同时沿谷底的速度越积越大，很快就到了终点。
- Adam：路线最直，因为它把两个方向的步长拉平了。离终点越近，梯度越小，m 跟着变小，而 v 衰减得慢（0.999），于是步子也自动变小，最后稳稳停在终点。

勾选“随机小批量”后，每一步只随机用 1 个样本算梯度，三者的表现就拉开了。SGD 的路线变得歪歪扭扭，到了终点附近也停不下来。Momentum 在学习率 0.03 下反而最糟：它把带噪声的梯度累积成速度，相当于把步长放大了约 10 倍，而单个样本的梯度又比 4 个样本的平均大得多，于是在山谷两壁之间越摆越大，损失冲到几百；把学习率降到 0.005，它才重新稳定下来。Adam

最稳：它按梯度的典型大小给步长“定标”，噪声再大，每步也只走 η 左右那么远，200 到 300 步之间的平均损失约 0.1，三种学习率下都一样。这也是 Adam 在大模型训练里受欢迎的原因之一：它对学习率没那么敏感。

要点

SGD 只看当前梯度；Momentum 把过去的梯度累积成速度，一致的方向加速、来回的方向抵消；Adam 再用梯度平方的平均给每个参数单独定步长，让不同尺度的参数走得一样远。

用代码复现

下面的代码先算出两组数据下曲面的弯曲程度和学习率上限，然后在平移后的山谷里比较三种优化器，并单独看 Adam 的第一步；最后算一算 Adam 训练时的显存账（下一节用）。

```python
import numpy as np

y = np.array([1.0, 3.0, 4.0, 6.0])

def make(x):
    loss = lambda t: np.mean((t[0] * x + t[1] - y) ** 2)
    grad = lambda t: np.array([np.mean(2 * (t[0] * x + t[1] - y) * x), np.mean(2 *
(t[0] * x + t[1] - y))])
    return loss, grad

def run(opt, x, lr, start, steps=300):
    loss, grad = make(x)
    t, m, v, hit = np.array(start), np.zeros(2), np.zeros(2), None
    for k in range(1, steps + 1):
        g = grad(t)
        if opt == "SGD":
            t = t - lr * g
        elif opt == "Momentum":
            m = 0.9 * m + g                          # 速度：把过去的梯度累积起来
            t = t - lr * m
        else:                                        # Adam
            m = 0.9 * m + 0.1 * g                    # 梯度的滑动平均（方向）
            v = 0.999 * v + 0.001 * g * g            # 梯度平方的滑动平均（尺度）
            m_hat, v_hat = m / (1 - 0.9 ** k), v / (1 - 0.999 ** k)
            t = t - lr * m_hat / (np.sqrt(v_hat) + 1e-8)
        if hit is None and loss(t) < 0.06:
            hit = k
    return hit, t, loss(t)

x_c = np.array([0.0, 1.0, 2.0, 3.0])                 # 第 17 讲的数据
x_s = x_c + 2                                        # 同一组数据，x 整体平移 2
for x, name in [(x_c, "原来的 x"), (x_s, "x 平移 2")]:
    H = 2 * np.array([[np.mean(x * x), np.mean(x)], [np.mean(x), 1.0]])   # 曲面的
“弯曲程度”矩阵
    ev = np.linalg.eigvalsh(H)
    print(f"{name}：最陡方向的弯曲 {ev[1]:.2f}，最平方向 {ev[0]:.2f}，相差 {ev[1] /
ev[0]:.0f} 倍，学习率上限 {2 / ev[1]:.3f}")

print(run("SGD", x_c, 0.1, (-0.5, -2.0))[0], "步（原来的 x，SGD，η = 0.1）")
for opt, lr in [("SGD", 0.03), ("SGD", 0.065), ("Momentum", 0.03), ("Adam", 0.3)]:
    hit, t, l = run(opt, x_s, lr, (-0.5, 2.0))
    print(f"{opt:8s} η = {lr:<5}：{'没到' if hit is None else f'{hit:3d} 步'}  300
步后 (w, b) = ({t[0]:.3f}, {t[1]:.3f})，L = {l:.4f}")

# Adam 的第一步：每个参数都只走 η 那么远，不管梯度本身多大
loss, grad = make(x_s)
```

```
g = grad(np.array([-0.5, 2.0]))
step = -0.3 * (0.1 * g / 0.1) / (np.sqrt(0.001 * g * g / 0.001) + 1e-8)
print("第一步的梯度：", g.round(2), "  SGD 的变化：", (-0.03 * g).round(4), "  Adam 的
变化：", step.round(4))

# 训练时每个参数要存的东西（混合精度 + Adam）
N = 8.03e9
per = {"BF16 权重": 2, "BF16 梯度": 2, "FP32 主权重": 4, "Adam m": 4, "Adam v": 4}
print(per, "合计", sum(per.values()), "字节/参数")
print(f"Llama-3-8B：{N * sum(per.values()) / 1e9:.1f} GB（推理只需 BF16 权重 {N * 2 /
1e9:.1f} GB）")
```

输出：

```
原来的 x：最陡方向的弯曲 8.41，最平方向 0.59，相差 14 倍，学习率上限 0.238
x 平移 2：最陡方向的弯曲 28.83，最平方向 0.17，相差 166 倍，学习率上限 0.069
39 步（原来的 x，SGD，η = 0.1）
SGD       η = 0.03 ：没到  300 步后 (w, b) = (1.363, -1.190)，L = 0.1267
SGD       η = 0.065：228 步  300 步后 (w, b) = (1.562, -1.955)，L = 0.0519
Momentum η = 0.03 ：  39 步  300 步后 (w, b) = (1.600, -2.100)，L = 0.0500
Adam      η = 0.3  ：  93 步  300 步后 (w, b) = (1.600, -2.100)，L = 0.0500
第一步的梯度：  [-28.   -6.5]   SGD 的变化：  [0.84  0.195]   Adam 的变化：  [0.3 0.3]
{'BF16 权重': 2, 'BF16 梯度': 2, 'FP32 主权重': 4, 'Adam m': 4, 'Adam v': 4} 合计 16
字节/参数
Llama-3-8B：128.5 GB（推理只需 BF16 权重 16.1 GB）
```

第一、二行：平移前后，最陡和最平方向的弯曲程度相差 14 倍和 166 倍，SGD 的学习率上限从 0.238 降到 0.069。第三行是对照：原来的数据，SGD 用 0.1 的学习率 39 步就到了。接下来四行是山谷里的比赛：SGD 用 0.03 走不到，用 0.065 要 228 步；Momentum 39 步；Adam 93 步，两者 300 步后都精确地停在 (1.6, −2.1)。然后是第一步的对比：梯度 (−28, −6.5)，SGD 的变化是 (0.84, 0.195)，Adam 的变化是 (0.3, 0.3)。

到了大模型里

今天训练大模型，几乎都用 Adam 的一个变体 **AdamW**。它和 Adam 的区别只在“权重衰减”（weight decay）的做法：每一步把参数往 0 的方向拉一点点，防止某些权重长得过大。AdamW 把这一步和梯度更新分开做，效果更稳定。Llama 系列的论文里用的就是 AdamW。

Adam 好用，但它有一个直接的代价：**每个参数要多存两个数 m 和 v**。再加上第 10 讲会讲的混合精度训练：前向和反向用 BF16 计算，但为了让微小的更新不被舍入掉，还要另存一份 FP32 的“主权重”。于是每个参数要存：

内容	精度	字节
权重	BF16	2
梯度	BF16	2
主权重	FP32	4
Adam 的 m	FP32	4
Adam 的 v	FP32	4
合计		16

Llama-3-8B 约 80.3 亿个参数，乘以 16 字节，约 128.5 GB，这还没算第 18 讲说的激活值。而推理只需要 BF16 的权重，16.1 GB。一张 80 GB 显存的 GPU，放得下推理用的权重，却放不下训练要的这些状态。所以大模型训练一定是多卡的，而且要把优化器状态、梯度、权重切开分到不同的卡上去，这就是第 103 讲的 ZeRO 和 FSDP。

容易混淆的地方

1. **不同优化器的学习率不能直接比**。SGD 的步长是 $\eta \times$ 梯度，梯度大步子就大；Adam 的步长大约就是 η，和梯度大小关系不大。所以 SGD 的 0.03 和 Adam 的 0.3 不是同一回事。大模型用 Adam 时，学习率通常在 10^{-4} 这个量级，比这一讲的玩具例子小得多。
2. **Adam 也不是万能的**。它把每个参数的步长拉平，但并不知道参数之间怎么“纠缠”，在斜着的窄谷里仍然会走一些弯路。它的超参数 β_1、β_2、ε 也会影响稳定性，大模型训练常把 β_2 从 0.999 调小到 0.95 左右。
3. **优化器状态也是“模型的一部分”**。训练中途保存检查点时，除了权重，还要保存 m 和 v，否则恢复训练时 Adam 要从头积累，前几步会走得很乱。所以训练检查点通常比最终发布的模型文件大好几倍。

回顾

优化器决定拿到梯度后怎么更新参数。SGD 直接沿负梯度走 $\eta \times \mathrm{g}$，实际中每步只用一小批数据，梯度带噪声；在不同方向尺度相差很大的窄谷里，它的学习率被最陡的方向卡住，在平的方向上走得很慢。Momentum 把过去的梯度累积成速度，一致的方向加速、来回震荡的方向互相

抵消。Adam 在此基础上再用梯度平方的滑动平均给每个参数单独定步长，让每个参数都走大约 η 那么远，在窄谷里路线最直。代价是每个参数要多存两个数：混合精度加 Adam，每个参数 16 字节，Llama-3-8B 光这些就要约 128.5 GB。

到这里，训练需要的零件都齐了：模型（第 12–15 讲）、损失函数（第 16 讲）、梯度（第 17 讲）、反向传播（第 18 讲）、优化器（这一讲）。把它们组装成一个真正能跑起来的训练程序，还要回答几个实际的问题：数据怎么分批喂进去？要过几遍？学习率要不要随时间变化？下一讲：**完整的训练循环**。

第20讲 训练循环

16 损失函数　18 反向传播　19 优化器

前面五讲把训练的零件一个个造好了：模型（第 12–15 讲），损失函数（第 16 讲），梯度和梯度下降（第 17 讲），反向传播（第 18 讲），优化器（第 19 讲）。这一讲把它们组装起来，从一个随机初始化的网络开始，真正训练出第 15 讲那样能拟合曲线的网络。

组装时会碰到几个前面没有细说的实际问题：数据是一次全喂进去，还是分批？分多大的批？整份数据要过几遍？学习率要一直不变吗？这一讲回答这些问题，并且在浏览器里实时地训练一个小网络，你可以亲手调这些设置，看损失曲线怎么变。

训练循环的五步

训练的骨架是两层循环。外层循环一遍遍地过整份数据，每过一遍之前先把数据顺序打乱；内层循环每次取出一小批数据，做五件事：

1. **取一批**：从打乱后的数据里按顺序取出 B 个样本。
2. **前向**：把这一批输入送进网络，算出预测（第 15 讲）。
3. **算损失**：把预测和答案比较，得到这一批的平均损失（第 16 讲）。
4. **反向**：从损失出发做反向传播，得到每个参数的梯度（第 18 讲）。
5. **更新**：优化器用梯度更新参数（第 19 讲）。

用 PyTorch 写出来，内层循环就是这么几行。注意第一行的清零：第 18 讲说过，PyTorch 的梯度是累加的，不清零的话，这一步的梯度会和上一步的加在一起。

```
for x, y in loader:              # ① 取一批
    optimizer.zero_grad()        #    把上一步的梯度清零
    pred = model(x)              # ② 前向
    loss = loss_fn(pred, y)      # ③ 损失
    loss.backward()              # ④ 反向：算出所有参数的梯度
    optimizer.step()             # ⑤ 更新
```

这一讲的例子是第 15 讲的同一个网络结构：1 → 8 → 8 → 1，ReLU，97 个参数。数据是 −3 到 3 之间均匀取的 64 个点，答案是 sin(1.5x) 再加上一点随机噪声（标准差 0.1）。加噪声是为了更像真实的数据：真实的数据总有测不准、标错了的地方。也因为有噪声，即使网络完美地学会了

sin(1.5x)，损失也降不到 0，而是停在噪声本身的大小，约 0.0101。

权重用一种叫 He 初始化的方法随机生成：每层的权重从均值为 0 的正态分布里抽，方差是 2 / 输入个数。这样各层输出的大小在一开始比较合适，不会太大也不会太小。偏置初始化为 0。随机初始化的网络，损失是 4.8043，比直接输出 0（损失约 0.5）还差得多。

batch、step、epoch

几个术语先说清楚：

- **batch**（批）：一次前向、反向用到的那一小批样本。它包含的样本个数叫 **batch size**，记作 B。
- **step**（步），也叫 iteration：处理一个 batch 并更新一次参数。
- **epoch**（轮）：把整份数据完整地过一遍。

数据有 64 个样本，batch size 是 8，那么每个 epoch 有 64 / 8 = 8 个 step。训练 400 个 epoch，一共更新 3200 次参数。

为什么每个 epoch 都要重新打乱顺序？如果每次都按同样的顺序分批，同一个 batch 里总是那几个样本，梯度的噪声会带上固定的偏向；打乱以后，每个 epoch 的分组都不一样，平均下来更接近全体数据的梯度。

batch 选多大

batch size 有两个极端。

一个极端是 **B 等于全部数据**（这里是 64）。每一步的梯度都是精确的，没有噪声，但每个 epoch 只能更新一次参数。下面的代码用 Adam、学习率 0.01 训练 400 个 epoch：一共只有 400 步，损失在第 100 个 epoch 还是 0.0799，到第 200 个 epoch 才降到 0.0105。

另一个极端是 **B = 1**，每一步只用一个样本。每个 epoch 能更新 64 次，前期下降最快：第 10 个 epoch 损失就到了 0.1597，第 100 个 epoch 是 0.0092。但每一步的梯度只来自一个点，噪声很大，参数一直被推来推去，停不下来：第 200 个 epoch 是 0.0117，第 400 个 epoch 反而回到了 0.0288。

中间的 **B = 8** 兼顾了两边：每个 epoch 更新 8 次，梯度的噪声比 B = 1 小得多。第 50 个 epoch 就降到 0.0332，最后是 0.0067。

> **注意**
> 0.0067 比噪声本身的 0.0101 还低。这说明网络不只学到了 sin(1.5x)，还开始“记住”这 64 个点各自的噪声。这叫过拟合（overfitting）：在训练数据上越来越好，换一批新数据却未必。判断有没有过拟合，要留一部分数据不参与训练，专门用来检验，这叫验证集。

在 GPU 上，batch size 还和速度直接相关。一个 batch 里的样本是同时计算的，batch 越大，矩阵乘法越“胖”，第 3 讲说的算术强度越高，GPU 的计算单元越能被喂饱。所以实际训练中，batch size 往往在显存允许的范围内尽量大，再相应地调学习率。

学习率：先热身，再慢慢降

第 17 讲说过，学习率是最重要的超参数。学习率太大会怎样，下面的代码也试了：B = 8、学习率 0.1，损失前期降得飞快（第 10 个 epoch 就到 0.0307），之后却在 0.01 到 0.07 之间来回跳，第 50 个 epoch 是 0.0671，第 200 个 epoch 是 0.0520，始终稳定不下来。步子太大，到了谷底附近就会一直在两边来回跨。

于是有了一个很自然的想法：**开始时步子大一点，快速接近谷底；后期步子小一点，稳稳地落进去**。这就是学习率调度（learning rate schedule）。大模型训练最常用的组合是**预热 + 余弦衰减**：

- **预热**（warmup）：前若干步，学习率从很小线性增大到设定值。刚开始时参数是随机的，梯度很大、方向也乱；Adam 的 v 还没积累起来，对梯度大小的估计不准。这时步子太大，容易一下子把网络推到很糟糕的地方。
- **余弦衰减**（cosine decay）：之后学习率按余弦曲线的一半，从设定值平滑地降到 0（或一个很小的值）。降得前慢、中间快、最后又慢。

这一讲的设定是：设定值 0.01，前 20 个 epoch 预热，之后在 400 个 epoch 内余弦衰减到 0。几个时刻的学习率是：第 1 个 epoch 0.0005，第 10 个 0.00499，第 20 个 0.00994，第 101 个 0.00854，第 201 个 0.005（正好一半），第 301 个 0.00146，最后一个 epoch 已经几乎是 0。

效果是：前期因为预热，下降比不调度时慢一点（第 1 个 epoch 损失还是 4.4762）；但后期学习率越来越小，损失稳稳地停在 0.0084，曲线比不调度时平滑得多。

自己训练一次

下面的演示在浏览器里实时训练这个网络。左图是 64 个数据点和当前网络的拟合曲线，右图是每个 epoch 结束时在全部数据上算的损失，用的是对数坐标（每一格是上一格的 10 倍），横虚线是噪声带来的下限 0.01。

先点“训练一步”：下方会写出这一个 step 的五件事，左图里这一步用到的 8 个样本会被标成琥珀色的大点。然后点“▶ 训练”，看绿线怎么从一条乱七八糟的折线，一点点变成正弦曲线的样子。

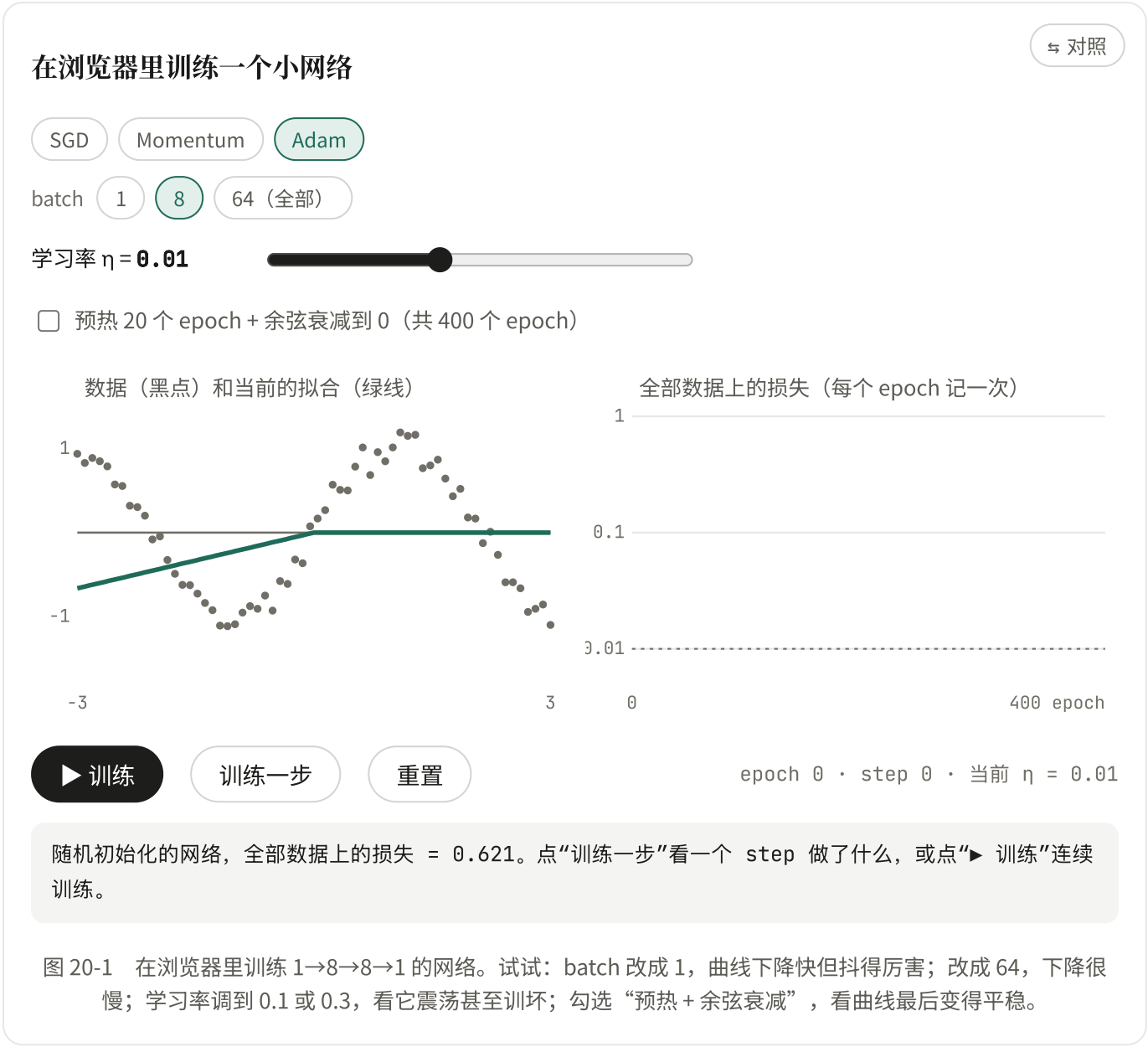


图 20-1　在浏览器里训练 1→8→8→1 的网络。试试：batch 改成 1，曲线下降快但抖得厉害；改成 64，下降很慢；学习率调到 0.1 或 0.3，看它震荡甚至训坏；勾选“预热 + 余弦衰减”，看曲线最后变得平稳。

这个演示用的是 JavaScript 的随机数，初始化和数据噪声和下面的 Python 代码不是同一组数，所以具体的损失值会不一样，但各种设置下的趋势是一致的。

可以做几个实验：

- **换优化器**：同样的学习率 0.01，SGD 下降得很慢，因为对 SGD 来说这个学习率太小了；把学习率调到 0.1，SGD 才和 Adam 差不多快。这就是第 19 讲说的“不同优化器的学习率不能直接比”。
- **把学习率调到 0.3**：Adam 的步子太大，没几步就把大部分 ReLU 推进了“死亡”区域（第 14 讲），网络只能输出一个常数，损失卡在 0.53 左右下不来。这个数正是“永远输出 0”时的损失。
- **batch = 1 和 batch = 64 对比**：注意右图的横轴是 epoch。batch = 1 时每个 epoch 有 64 步，batch = 64 时只有 1 步，所以前者“看起来”下降快得多。如果按 step 算，结论就不一样了。

用代码复现

下面的代码完整实现了这一讲的训练循环：反向传播按第 18 讲的公式手写，优化器是第 19 讲的 Adam。它比较了几种 batch size 和学习率的组合，打印第 1、10、50、100、200、400 个 epoch 结束时在全部数据上的损失，最后算了 Llama-3-8B 的训练账（下一节用）。

```
import numpy as np

rng = np.random.default_rng(2026)
X = np.linspace(-3, 3, 64)[:, None]                          # 64 个样本
Y = np.sin(1.5 * X) + 0.1 * rng.standard_normal((64, 1))  # 带一点噪声的目标

def init(seed):
    r = np.random.default_rng(seed)
    he = lambda n_in, n_out: r.standard_normal((n_in, n_out)) * np.sqrt(2 / n_in)
    return [he(1, 8), np.zeros(8), he(8, 8), np.zeros(8), he(8, 1), np.zeros(1)]

def forward(P, x):
    W1, b1, W2, b2, W3, b3 = P
    z1 = x @ W1 + b1; h1 = np.maximum(0, z1)
    z2 = h1 @ W2 + b2; h2 = np.maximum(0, z2)
    return (z1, h1, z2, h2), h2 @ W3 + b3

def backward(P, x, y):                                       # 第 18 讲：线性层的反向是两
次矩阵乘法
    W1, b1, W2, b2, W3, b3 = P
    (z1, h1, z2, h2), yh = forward(P, x)
    dy = 2 * (yh - y) / len(x)
    dz2 = (dy @ W3.T) * (z2 > 0)
    dz1 = (dz2 @ W2.T) * (z1 > 0)
    return [x.T @ dz1, dz1.sum(0), h1.T @ dz2, dz2.sum(0), h2.T @ dy, dy.sum(0)]

def lr_at(epoch, base, warmup=20, total=400):                # 预热 + 余弦衰减
    return base * min(1, (epoch + 1) / warmup) * 0.5 * (1 + np.cos(np.pi * epoch /
total))

def train(batch, base_lr, epochs=400, schedule=False):
    P = init(15)
    m = [np.zeros_like(p) for p in P]; v = [np.zeros_like(p) for p in P]
    order_rng, t, log = np.random.default_rng(7), 0, {}
    for epoch in range(epochs):
        lr = lr_at(epoch, base_lr) if schedule else base_lr
        order = order_rng.permutation(64)                    # 每个 epoch 重新洗牌
        for s in range(0, 64, batch):
            idx = order[s:s + batch]                         # ① 取一批
            G = backward(P, X[idx], Y[idx])                  # ② 前向 ③ 损失 ④ 反向
            t += 1
            for k in range(6):                               # ⑤ Adam 更新
                m[k] = 0.9 * m[k] + 0.1 * G[k]
                v[k] = 0.999 * v[k] + 0.001 * G[k] ** 2
                P[k] -= lr * (m[k] / (1 - 0.9 ** t)) / (np.sqrt(v[k] / (1 - 0.999
** t)) + 1e-8)
```

```
        if epoch + 1 in (1, 10, 50, 100, 200, 400):
            log[epoch + 1] = float(np.mean((forward(P, X)[1] - Y) ** 2))
    return t, log

print("噪声本身的均方误差：", round(float(np.mean((np.sin(1.5 * X) - Y) ** 2)), 4))
print("随机初始化时的损失：", round(float(np.mean((forward(init(15), X)[1] - Y) **
2)), 4))
for batch, lr, sch in [(64, 0.01, False), (8, 0.01, False), (1, 0.01, False), (8,
0.1, False), (8, 0.01, True)]:
    steps, log = train(batch, lr, schedule=sch)
    print(f"batch {batch:2d}, η = {lr}{', 预热+余弦' if sch else ''}：{steps:5d}
步  " + "  ".join(f"{e}:{l:.4f}" for e, l in log.items()))
print("学习率调度：", [round(float(lr_at(e, 0.01)), 5) for e in (0, 9, 19, 100, 200,
300, 399)])

N, D = 8.03e9, 15e12
print(f"Llama-3-8B 训练约 15T token：6 × N × D = {6 * N * D:.2e} FLOPs")
print(f"H100 BF16 峰值约 989 TFLOPS，按 40% 利用率：{6 * N * D / (989e12 * 0.4) /
3600:.3g} GPU 小时")
```

输出：

```
噪声本身的均方误差：  0.0101
随机初始化时的损失：  4.8043
batch 64, η = 0.01：   400 步  1:3.9117  10:0.9476  50:0.4260  100:0.0799  200:0.0105
400:0.0102
batch  8, η = 0.01：  3200 步  1:1.2802  10:0.4034  50:0.0332  100:0.0300  200:0.0165
400:0.0067
batch  1, η = 0.01：25600 步  1:0.6151  10:0.1597  50:0.0121  100:0.0092  200:0.0117
400:0.0288
batch  8, η = 0.1：  3200 步  1:0.5173  10:0.0307  50:0.0671  100:0.0115  200:0.0520
400:0.0138
batch  8, η = 0.01, 预热+余弦：  3200 步  1:4.4762  10:0.7099  50:0.0133  100:0.0113
200:0.0089  400:0.0084
学习率调度：  [0.0005, 0.00499, 0.00994, 0.00854, 0.005, 0.00146, 0.0]
Llama-3-8B 训练约 15T token：6 × N × D = 7.23e+23 FLOPs
H100 BF16 峰值约 989 TFLOPS，按 40% 利用率：5.07e+05 GPU 小时
```

前两行：噪声本身的均方误差 0.0101，这是损失的“地板”；随机初始化的损失 4.8043。接下来五行就是正文里比较过的五种设置，冒号前是 epoch，冒号后是损失，步数一栏可以看到 batch 越小，同样 400 个 epoch 走的步数越多。然后是学习率调度在几个 epoch 的取值。最后两行是下一节的训练账。

到了大模型里

大模型的训练循环和这一讲的一模一样：取一批、前向、损失、反向、更新。区别在规模。

数据和 batch 以 token 计。语言模型的“样本”是一段段文本，每个位置都要预测下一个词（第 16 讲），所以 batch 的大小通常用 token 数来说，一步要处理几百万个 token。训练数据往往大到只过一遍，甚至不到一遍，所以大模型预训练很少说“训练了几个 epoch”，而是说“训练了多少 token”。Meta 公布的 Llama 3 预训练数据是 15 万亿（15T）个 token 以上。

一张卡放不下一个 batch。几百万个 token 的激活值，单卡根本存不下。常用的办法是**梯度累积**（gradient accumulation）：把一个大 batch 拆成几个小份，依次前向、反向，梯度先不清零、一直累加（正好利用了 PyTorch 梯度累加的特性），全部做完再更新一次参数，效果和一次处理整个大 batch 相同。再加上多卡的数据并行，每张卡处理一部分，最后把梯度加起来（第 103 讲）。

训练的总成本可以估算。第 18 讲说过，训练一个 token 约需 6N 次浮点运算。Llama-3-8B 的 N 约 80.3 亿，D 按 15T 个 token 算，总共约 $6 \times N \times D = 7.23 \times 10^{23}$ 次。H100 的 BF16 峰值算力约 989 TFLOPS，实际训练能用上 40% 就算不错，这样算下来约 50.7 万 GPU 小时，相当于一千张 H100 连续跑三个星期左右。实际花费还要加上调试、失败重来、评估等开销，Meta 在模型卡里公布的数字要比这个估算更高。

学习率调度按 step 走。大模型训练的预热通常是前几千步，之后余弦衰减（或类似的曲线）一直到训练结束。训练过程中，工程师盯得最紧的就是右边那种损失曲线：它突然往上跳（loss spike），往往意味着学习率太大或者数据有问题，需要回到之前的检查点重新开始。

容易混淆的地方

1. **训练损失降低，不等于模型变好。**这一讲 B = 8 的损失降到了 0.0067，比噪声还低，说明网络在记忆噪声。真正要看的是没参与训练的验证数据上的损失。大模型预训练因为数据量极大、几乎每段数据只看一次，过拟合反而不是主要问题。
2. **batch size 和学习率要一起调。**batch 变大，梯度噪声变小，一般可以（也需要）用更大的学习率；只改一个不改另一个，结论可能完全相反。
3. **“step”“epoch”的横轴要看清楚。**比较两次训练的损失曲线时，横轴用的是 step、epoch 还是 token 数，会得出很不一样的结论。大模型的论文和训练日志里，横轴通常是训练过的 token 数。

回顾

训练循环是两层循环：外层每个 epoch 把数据打乱，内层每个 step 取一个 batch，做“前向 → 损失 → 反向 → 更新”五件事（PyTorch 里还要先把梯度清零）。batch 太大，每个 epoch 更新次数少；太小，梯度噪声大、停不下来，B = 8 这样的中间值往往最好。学习率太大会在谷底附近来回跳，所以常用“预热 + 余弦衰减”：先小、再大、再慢慢降到 0，让训练前期稳、后期准。大模型的训练循环完全一样，只是规模大得多：一步几百万个 token，靠梯度累积和多卡并行完成，总计算量约 6 × 参数量 × token 数。

第 2 篇到这里就结束了。我们一直在拿数字做输入：一个 x，一对甜度酸度。可语言模型的输入是文字，“我爱学 AI”这几个字，是怎么变成网络能处理的数字的？第 3 篇从这个问题开始。下一讲：**分词与 BPE**。

AI INFRA · HANDBOOK

第 三 篇

从文本到向量

收录 4 讲 · 第 21–24 讲

第21讲 分词与 BPE

01 向量　16 损失函数

第 2 篇里，网络的输入一直是数字：一个 x，或者一对甜度和酸度。可是语言模型的输入是文字。“我爱学AI”这几个字，要先变成数字，网络才能处理。这一篇（第 21–24 讲）就沿着这条路走一遍：文字先切成一个个小块，每块对应一个编号（这一讲）；编号再变成向量（第 22 讲）；模型根据这些向量预测下一个小块（第 23 讲）；一个接一个地预测，就生成了一段话（第 24 讲）。

这一讲回答第一步：**文字怎么切？**切出来的小块叫 **token**，切的过程叫**分词**（tokenization），负责切分的程序叫**分词器**（tokenizer）。我们会看到，今天的大模型几乎都用同一种思路：BPE，它是通过“数数”从数据里学出来的。

按字切，还是按词切

最直接的想法有两种。

按字切：每个字（英文是每个字母）一个 token。“我爱学AI”切成 我 / 爱 / 学 / A / I，五个 token。好处是词表很小，常用汉字几千个，英文字母加符号一百来个，而且任何文字都能切，不会遇到“不认识”的字。坏处是序列很长：一个英文单词要拆成好几个字母，模型要处理的 token 数成倍增加。后面会看到，模型的计算量和显存都随 token 数增长，序列长是很贵的。另外，单个字母几乎不带含义，模型得花很大力气才能从字母里拼出“词”的意思。

按词切：每个词一个 token。“我 / 爱 / 学 / AI”，四个 token，每个都有完整的含义。坏处是词表会非常大：光英文就有几十万个词形（play、plays、played、playing 各算一个），还有人名、地名、新词、拼写错误，永远也收不全。词表里没有的词，模型就完全不认识。中文还多一个麻烦：词与词之间没有空格，“按词切”本身就是个难题。

所以实际用的是两者之间的折中：**常见的字组合成一个 token，不常见的就拆开**。“推理”常常一起出现，就当成一个 token；一个罕见的人名，就拆成几个字甚至更小的单位。这种切法叫**子词**（subword）分词。剩下的问题是：哪些组合算“常见”？BPE 的回答是：从数据里数出来。

BPE：把常一起出现的合并

BPE（Byte Pair Encoding，字节对编码）原本是一种数据压缩方法，后来被用来学分词规则。它的步骤只有两条，反复执行：

1. 数一数语料里所有**相邻的两个 token** 各出现了多少次。
2. 把出现次数最多的那一对，**合并成一个新 token**，加进词表，记下这条合并规则。

一开始，每个 token 都是单个字符。拿一段很小的语料来看：

> 我爱学AI。我爱推理。学AI的人爱推理。AI推理很快。我学推理。

先按句号切成 5 段（合并不跨过句号），每段拆成单字。出现过的字有 11 个：我、爱、学、A、I、推、理、的、人、很、快，这就是初始的词表；整段语料一共 27 个 token。

第 1 次合并。数相邻的对：“推”后面跟“理”出现了 4 次（第 2、3、4、5 段各一次），是最多的。于是把“推 理”合并成新 token“推理”。语料变成：我 爱 学 A I / 我 爱 推理 / ……

第 2 次合并。重新数：“A I”出现了 3 次，最多。合并成“AI”。

第 3 次合并。这时出现最多的对都只有 2 次了：“我 爱”2 次，“学 AI”2 次，“爱 推理”2 次。次数相同，我们约定选在语料里最先出现的那一对：“我 爱”，合并成“我爱”。

第 4 次合并。“学 AI”2 次，合并成“学AI”。之后，所有相邻的对都只出现 1 次了，合并结束。

最后的词表是 11 个单字加 4 个合并出来的 token，共 15 个；语料从 27 个 token 缩短到了 16 个。

注意第 3、4 次合并的结果：“我爱”“学AI”并不是我们心目中的“词”。BPE 不懂语法，它只看哪些字经常挨在一起。在这段小语料里，“我爱”恰好出现了两次，就被合并了。语料大了以后，真正常见的组合（常用词、常见的词缀）会被优先合并，切出来的结果才会大体像“词”，但也总会有一些看起来奇怪的 token。

要点

BPE 从单个字符出发，一遍遍地把最常相邻出现的两个 token 合并成一个新 token。合并多少次，词表就多出多少个 token；合并的规则和顺序都要记下来。

下面的演示就是这个过程。点“合并一次”，右边的表会列出当前出现次数最多的几对，绿色高亮的那一行就是下一次要合并的；左边是语料现在的切分，刚合并出来的 token 标成琥珀色。你也可以改上面的语料，看会学出什么规则。

⇆ 对照

BPE：数一数，把最常一起出现的两个合并

语料（可以改）

我爱学AI。我爱推理。学AI的人爱推理。AI推理很快。我学推理。

合并一次　合并到底　重置

词表 11 个（11 个单字 + 0 条合并） · 语料共 27 个 token

语料现在被切成

我 爱 学 A I 。

我 爱 推 理 。

学 A I 的 人 爱 推 理 。

A I 推 理 很 快 。

我 学 推 理 。

相邻两个 token 的出现次数（前 6）

一对	次数
推 + 理	4
A + I	3
我 + 爱	2
学 + A	2
爱 + 推	2
爱 + 学	1

学到的合并规则

1. 还没有。点“合并一次”。

用这些规则切一句新话

你爱学AI推理

你	爱	学	A	I	推	理
不在词表	1	2	3	4	5	6

每段语料先拆成单个字符。下一次会合并“推”+“理”（4 次）。
新句子切成 7 个 token（下面的小字是编号）；“你”在语料里没出现过，不在词表里：真实的分词器从字节开始合并，所以不会出现这种情况。

图 21-1　在一段小语料上训练 BPE。4 次合并后，“推理”“AI”都成了一个 token，语料从 27 个 token 缩短到 16 个。下面的新句子按学到的规则切分，“你”在语料里没出现过，不在词表里。

用学到的规则切新句子

学完规则，怎么切一句新的话？做法是：先拆成单字，然后**按学到的顺序**，把每条合并规则依次应用一遍。

拿“你爱学AI推理”试试。拆成单字：你 / 爱 / 学 / A / I / 推 / 理。应用第 1 条“推 + 理”：你 / 爱 / 学 / A / I / 推理。第 2 条“A + I”：你 / 爱 / 学 / AI / 推理。第 3 条“我 + 爱”：没有“我”，不变。第 4 条“学 + AI”：你 / 爱 / 学AI / 推理。最终切成 4 个 token。

每个 token 在词表里有一个编号：按初始单字的顺序编 0 到 10，合并出来的按合并顺序编 11 到 14。“爱”是 1，“学AI”是 14，“推理”是 11。这串编号，就是模型真正看到的输入。

问题出在“你”。它在训练语料里没出现过，不在词表里，没有编号。按字切和按词切会遇到的“不认识”问题，在这里也出现了。

从字节开始

真实的分词器用一个巧妙的办法彻底消灭了这个问题：**不从字符开始，而从字节开始**。

计算机里，所有文字最终都存成一串字节。最常用的编码 UTF-8 里，英文字母占 1 个字节（“A”是 41，用十六进制写），常用汉字占 3 个字节（“我”是 e6 88 91，“你”是 e4 bd a0）。一个字节只有 256 种可能的取值。所以，把初始词表设成这 256 个字节，**任何文字、任何符号、甚至表情符号，都一定能切成词表里的 token**，最坏的情况也就是退回到一个个字节。

在字节之上再做 BPE：常见汉字的 3 个字节会很快被合并成一个 token，常见的词再进一步合并；罕见的字可能保持 2 个或 3 个 token。这就是今天 GPT、Llama 等模型用的“字节级 BPE”（byte-level BPE）。

用代码复现

下面的代码用十几行实现了 BPE 的训练和切分，结果和演示一致；然后看了几个字符的 UTF-8 字节，最后算一下 Llama-3-8B 的词表要占多少参数。

```python
from collections import Counter

corpus = "我爱学AI。我爱推理。学AI的人爱推理。AI推理很快。我学推理。"
chunks = [list(c) for c in corpus.split("。") if c]        # 按句号切段，每段先拆成单字
base = list(dict.fromkeys(ch for c in chunks for ch in c)) # 初始词表：出现过的单字，按出现顺序
print("初始词表：", "".join(base), f" ({len(base)} 个) ，语料共 {sum(map(len, chunks))} 个 token")

def merge(seq, a, b):                                      # 把相邻的 a、b 合成一个 token
    out, i = [], 0
    while i < len(seq):
        if i + 1 < len(seq) and seq[i] == a and seq[i + 1] == b:
            out.append(a + b); i += 2
        else:
            out.append(seq[i]); i += 1
    return out

merges = []
while True:
    pairs = Counter(p for c in chunks for p in zip(c, c[1:]))   # 数相邻的一对；Counter 按第一次出现的顺序排
    (a, b), n = max(pairs.items(), key=lambda kv: kv[1])        # 次数相同时 max 取最先出现的
    if n < 2:
        break
    merges.append((a, b))
    chunks = [merge(c, a, b) for c in chunks]
    print(f"第 {len(merges)} 次：{a} + {b} → {a + b} ({n} 次) |", " / ".join("".join(c) for c in chunks))

vocab = base + [a + b for a, b in merges]
tokens = list("你爱学AI推理")
for a, b in merges:                                        # 新句子：按学到的顺序逐条合并
    tokens = merge(tokens, a, b)
print("切分：", tokens, " 编号：", [vocab.index(t) if t in vocab else None for t in tokens])

# 真实的分词器从 UTF-8 字节开始合并：任何文字都能拆成 256 种字节之一
for ch in ["我", "A", "你"]:
    print(ch, "的 UTF-8 字节：", " ".join(f"{b:02x}" for b in ch.encode("utf-8")))

# 词表大小直接决定嵌入表和输出层的大小（Llama-3-8B：词表 128256，隐藏维度 4096）
V, d = 128256, 4096
print(f"嵌入表 + 输出层：2 × {V} × {d} = {2 * V * d / 1e9:.2f} B 个参数")
```

输出：

```
初始词表： 我爱学AI推理的人很快 （11 个），语料共 27 个 token
第 1 次：推 + 理 → 推理（4 次） | 我 爱 学 A I / 我 爱 推理 / 学 A I 的 人 爱 推理 / A I 推理 很 快 / 我 学 推理
第 2 次：A + I → AI（3 次） | 我 爱 学 AI / 我 爱 推理 / 学 AI 的 人 爱 推理 / AI 推理 很 快 / 我 学 推理
第 3 次：我 + 爱 → 我爱（2 次） | 我爱 学 AI / 我爱 推理 / 学 AI 的 人 爱 推理 / AI 推理 很 快 / 我 学 推理
第 4 次：学 + AI → 学AI（2 次） | 我爱 学AI / 我爱 推理 / 学AI 的 人 爱 推理 / AI 推理 很 快 / 我 学 推理
切分： ['你', '爱', '学AI', '推理'] 编号： [None, 1, 14, 11]
我 的 UTF-8 字节： e6 88 91
A 的 UTF-8 字节： 41
你 的 UTF-8 字节： e4 bd a0
嵌入表 + 输出层：2 × 128256 × 4096 = 1.05 B 个参数
```

逐行对照：初始词表 11 个字，语料 27 个 token；4 次合并依次是“推理”（4 次）、“AI”（3 次）、“我爱”（2 次）、“学AI”（2 次），每一行后面是合并后语料的样子。新句子切成 [你, 爱, 学AI, 推理]，编号 [None, 1, 14, 11]，None 表示“你”不在词表里。接着是三个字符的 UTF-8 字节：汉字 3 个，字母 1 个。最后一行是下一节要用的数。

到了大模型里

Llama-3 的分词器是字节级 BPE，词表有 128256 个 token。词表大小是一个需要权衡的设计：

词表越大，模型的两头越大。第 22 讲会讲到，每个 token 在模型的入口有一行嵌入向量，在出口（输出层，第 33 讲）也有一列权重。Llama-3-8B 的隐藏维度是 4096，嵌入表和输出层合起来是 $2 \times 128256 \times 4096 \approx 10.5$ 亿个参数，占了整个模型的 13% 左右。而且第 16 讲说过，每生成一个 token，都要在整个词表上算一遍 softmax。

词表越大，同样的文字切出来的 token 越少。一个 token 能装下更多的字，一段话需要的 token 就少。这一点非常重要，因为几乎一切都按 token 计：模型能处理的上下文长度（比如 8K、128K）指的是 token 数；推理的计算量和第 9 篇要讲的 KV Cache 的显存，都和 token 数成正比；调用大模型 API 的价格，也是按输入、输出的 token 数收费的。同一段中文，用一个中文语料训练得充分的分词器，可能比用一个以英文为主的分词器少切出很多 token，推理就便宜、快得多。Llama-3 把词表从 Llama-2 的 32000 扩大到 128256，一个重要的目的就是提高这种“压缩率”。

分词在 GPU 之外。分词是在 CPU 上做的字符串处理，发生在模型计算之前。推理服务里，它通常不是瓶颈，但在请求量很大时，分词和反分词（把生成的编号还原成文字）也要占一部分 CPU 时间。

容易混淆的地方

1. **token 不等于字，也不等于词**。一个 token 可能是一个字、一个词、半个词、几个字节，甚至带着前面的空格（英文分词器里，“ the”和“the”常常是两个不同的 token）。说“这段话有多少 token”，一定要说清楚是用哪个分词器切的。
2. **编号本身没有大小的含义**。“学AI”是 14、“推理”是 11，不代表“学AI”比“推理”大。编号只是词表里的位置，像学号一样。下一讲会看到，模型是用编号去查表，而不是拿编号本身做计算。
3. **分词器和模型是绑定的**。模型训练时用的是哪个分词器，推理时就必须用同一个。换一个分词器，同样的编号对应的就是完全不同的 token，模型的输出会变成乱码。词表里还有一些特殊 token，比如表示“一段文本结束”的 `<eos>`（Llama-3 里写作 `◁end_of_text▷` 等），它们不对应任何文字，用来标记结构。

回顾

模型只能处理数字，所以文字要先切成 token，再换成编号。按字切词表小但序列长，按词切含义完整但词表无穷大，实际用的是折中的子词分词。BPE 从单个字符（真实系统里是 256 个字节）出发，反复把最常相邻出现的两个 token 合并成新 token，合并规则按顺序记下来；切新句子时，先拆开，再按顺序应用这些规则。从字节开始，就永远不会遇到不认识的字。词表大小是一个权衡：大了模型两头的参数多，但同样的文字切出的 token 少，而推理的成本、上下文的长度、API 的价格，全都按 token 计。

现在，“你爱学AI推理”变成了一串编号 [?, 1, 14, 11]。可编号只是词表里的位置，14 和 11 之间的距离、大小都没有意义。模型怎么从这些编号里得到能做计算、带着含义的东西？下一讲：**词嵌入**。

第22讲 词嵌入

02 点积与余弦相似度　13 神经元与线性层　21 分词与 BPE

上一讲把文字切成了 token，又换成了编号：“爱”是 1，“学AI”是 14，“推理”是 11。可编号只是词表里的位置，像学号一样。模型要做的是乘法、加法、点积，拿学号来做这些运算没有意义。

这一讲要解决的是：**怎么把一个编号，变成一个能参与计算、而且带着含义的向量？**答案出奇地简单：查一张表。这张表叫**嵌入表**，查出来的向量叫**词嵌入**（embedding）。第 1 讲里“我”的向量 (0.7, −0.6, −0.9, 0.3)，就是从 TinyLM 的嵌入表里查出来的。这一讲把这件事讲透，再用一个真正训练过的小模型，看看嵌入里的“含义”是怎么来的。

为什么不能直接用编号

最省事的做法是把编号本身当成输入：“爱”输入 1，“推理”输入 11。这样不行，因为数字自带了大小和距离：11 比 1 大 10 倍；11 和 14 只差 3，看起来“推理”和“学AI”很接近，而和“爱”离得很远。这些关系完全是分词时编号的顺序碰巧造成的，没有任何语言上的意义。如果模型拿编号直接做乘法、加法，它就会把这些虚假的关系当真。

另一种做法是**独热向量**（one-hot）：词表有 V 个 token，就用一个 V 维向量，第 id 位是 1，其余全是 0。TinyLM 的词表是“我、爱、学、AI、推理、`<eos>`”，“爱”（编号 1）就是 (0, 1, 0, 0, 0, 0)。这样所有 token 一律平等，没有谁大谁小。

可是平等得过了头。任意两个不同的独热向量，点积都是 0，余弦相似度也是 0（第 2 讲）：在它们眼里，“苹果”和“香蕉”的距离，与“苹果”和“推理”的距离一模一样，没有“相近”的概念。而且向量太长：Llama-3 的词表有 128256 个 token，每个独热向量就有 128256 维，几乎全是 0。

我们想要的是一种折中：**每个 token 一个不太长的向量（比如 4096 维），每一维都是可以调整的实数，意思相近的 token，向量也相近。**

查一张表

做法是准备一张 V 行、d 列的表 E，叫**嵌入表**（embedding table）或嵌入矩阵。第 i 行就是编号为 i 的 token 的向量，d 叫嵌入维度。要把编号变成向量，就取出表里对应的那一行：

$$\mathbf{x} = E[\mathrm{id}]$$

TinyLM 的嵌入表是 6 × 4 的：6 个 token，每个 4 维。“我爱学AI”的 4 个 token 编号是 [0, 1, 2, 3]，依次取出第 0、1、2、3 行，叠在一起，就得到一个 4 × 4 的矩阵 X。这正是第 3 讲做矩阵乘法时用的那个 X：从那时起，我们一直在用查表查出来的向量。

一句话有 s 个 token，查表的结果就是 s × d 的矩阵，每一行是一个 token 的向量。这就是模型真正的输入，后面所有的计算都从这里开始。

下面的演示可以自己拼一个句子：点上面的词把它加进去，每个 token 下面写着它的编号；左边的嵌入表里，被用到的行变深，当前选中的 token 对应的行高亮；右边是查出来的结果 X。同一个词出现两次，查出来的两行一模一样。

⇆ 对照

查表：编号 → 嵌入表的一行

点词加入句子： 我 爱 学 AI 推理 <eos> 清空

我	爱	学	AI
编号 0	编号 1	编号 2	编号 3

嵌入表 E（6 × 4，每个词一行）

	1	2	3	4
0 我	0.7	-0.6	-0.9	0.3
1 爱	-0.3	-0.1	-0.8	-0.2
2 学	0.3	-0.3	0.6	0.6
3 AI	0.4	0.8	0.4	-0.6
4 推理	0.7	0.3	-0.8	-0.4
5 <eos>	-0.6	0.9	0.4	0.8

查出来的结果 X（句子长度 × 4）

	1	2	3	4
1 我	0.7	-0.6	-0.9	0.3
2 爱	-0.3	-0.1	-0.8	-0.2
3 学	0.3	-0.3	0.6	0.6
4 AI	0.4	0.8	0.4	-0.6

第 1 个 token“我”的编号是 0，查表就是取 E 的第 0 行（从 0 数）：(0.7, -0.6, -0.9, 0.3)
等价的写法：独热向量 (1, 0, 0, 0, 0, 0) 乘 E。比如结果的第 1 个数 = 1×0.7 + 0×(-0.3) + 0×0.3 + 0×0.4 + 0×0.7 + 0×(-0.6) = 0.7：只有编号那一位的 1 起作用。

图 22-1　查表。每个编号取出嵌入表的一行，一句话就变成一个“句子长度 × 4”的矩阵。下方写出了等价的“独热向量 × E”的算法。

查表就是矩阵乘法

查表看起来和前面学的东西不一样，其实它就是一次矩阵乘法。

把“爱”的独热向量 (0, 1, 0, 0, 0, 0) 乘以嵌入表 E。按第 3 讲的“行乘列”，结果的第 j 个数是这个独热向量和 E 第 j 列的点积：$0 \times E_{0j} + 1 \times E_{1j} + 0 \times E_{2j} + \cdots$，只有乘 1 的那一项留了下来，等于 E_{1j}。所以整个结果就是 E 的第 1 行：

$$\text{onehot(id)} \cdot E = E \text{ 的第 id 行}$$

一整句话也一样：把 s 个独热向量叠成 s × V 的矩阵，乘以 V × d 的 E，得到 s × d 的 X，和直接查表完全相同。

这告诉我们两件事。第一，**嵌入层就是一个线性层**（第 13 讲）：没有偏置，输入是独热向量，权重矩阵就是 E。所以它的参数可以和其他权重一样，用反向传播和梯度下降来训练。第二，实际计算时从不真的去乘那个几乎全是 0 的独热向量，那样做 V 倍的乘法，结果却只是取一行。直接取行，又快又省。

反向传播时也一样：一个 token 的梯度只会加回到它自己那一行上。一个 batch 里没出现的 token，它们的行这一步收不到梯度，原样不动。

要点

嵌入表 E 有 V 行 d 列，每个 token 一行。查表 E[id] 等价于“独热向量 × E”，所以嵌入层就是一个输入为独热向量、没有偏置的线性层，它的每一个数都是要训练的参数。

嵌入是学出来的

TinyLM 的嵌入表是随机生成的，“我”的向量 (0.7, −0.6, −0.9, 0.3) 没有任何含义。嵌入里的“含义”从哪来？从训练来。

为了看清这一点，我们在一份很小的语料上真正训练了一个模型（代码在 `tinylm.py` 的 `lecture_22`）。语料是 16 种简单的句子，有的重复几遍，一共 14 个不同的词：

> 小猫 吃 鱼 · 小狗 吃 骨头 · 我 吃 苹果 · 我 吃 香蕉 · 你 学 推理 · 我 喜欢 小猫 · 我 学 AI ……

模型的任务是：看前两个词，预测下一个词。比如看到“小猫 吃”，要预测“鱼”。它的结构是：两个词各查一次嵌入表（每个词 8 维），拼成 16 维，经过一个线性层和 ReLU，再经过一个线性层，得到 14 个词的打分，用交叉熵训练（第 16–20 讲的全套流程）。下一讲会细讲这个“预测下一个词”的任务，这一讲只关心训练完以后，嵌入表变成了什么样。

训练前，嵌入是随机数。和“苹果”余弦相似度最高的三个词是“学”（0.67）、“香蕉”（0.33）、“你”（0.27），毫无道理。训练后，变成了“香蕉”（0.99）、“鱼”（0.97）、“AI”（0.93）。和“我”最像的，训练前是“骨头”（0.69），训练后是“你”（0.64）。

下面的演示把 14 个词的 8 维嵌入投影到二维画了出来（取的是数据散开最大的两个方向，类似给一堆点拍一张最能看出差别的照片）。点一个词，会连线到在 8 维里和它余弦相似度最高的 3 个词。切换“训练前”，可以看到随机初始化时的样子。

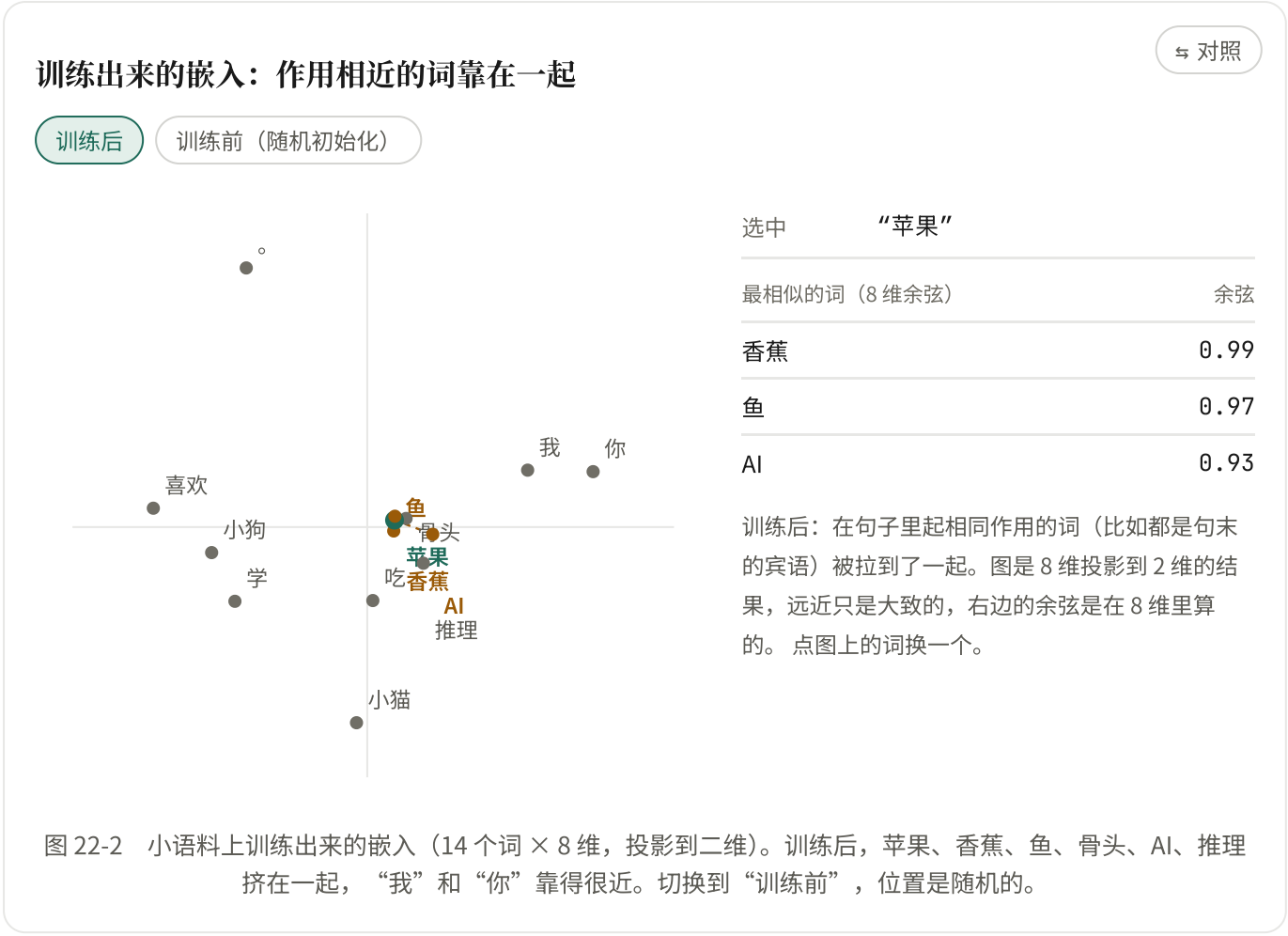


图 22-2　小语料上训练出来的嵌入（14 个词 × 8 维，投影到二维）。训练后，苹果、香蕉、鱼、骨头、AI、推理挤在一起，“我”和“你”靠得很近。切换到“训练前”，位置是随机的。

为什么苹果、香蕉、鱼、骨头、AI、推理会挤在一起？它们在语料里的“作用”完全一样：都出现在句子的末尾，前面是“吃”“学”“喜欢”，后面跟着句号。模型用前两个词预测下一个词，当这些词作为前文出现时，下一个词都是句号，模型没有理由区分它们，于是它们的嵌入被推到了一起。“我”和“你”也类似：后面都可以接“吃”“喜欢”“学”，只是“我”更常学 AI、“你”更常学推理，所以相似但不完全相同。

有意思的是“小猫”和“小狗”并没有靠在一起。在这份语料里，小猫吃鱼，小狗吃骨头，作为前文时它们引出的下一个词不同，模型必须把它们区分开。

这就是嵌入的“含义”的本质：**两个词的嵌入相近，是因为它们在上下文中起的作用相近**。语言学里有一句老话：要了解一个词，就看它周围的词（分布假说）。大模型在海量文本上训练，嵌入学到的是一种比我们这个小例子丰富得多的“作用”：词性、意思、情感色彩、领域……每一种都对应向量空间里的某些方向。

用代码复现

下面的代码演示了查表、查表和独热乘法的等价、训练前后的相似词，以及反向传播时只有用到的行收到梯度；最后算一下 Llama-3-8B 的嵌入表有多大。

```python
import json
import numpy as np

t = json.load(open("tinylm.json", encoding="utf-8"))

# 1. TinyLM：编号 → 查表
vocab, E = t["config"]["vocab"], np.array(t["weights"]["embedding"])      # E：6×4
ids = [vocab.index(w) for w in ["我", "爱", "学", "AI"]]
X = E[ids]                                                                # 一次取出
4 行
print("编号：", ids, " X 的形状：", X.shape)
print(X)

# 2. 查表 = 独热向量 × E
onehot = np.eye(len(vocab))[ids]                                          # 4×6，每
行只有一个 1
print("独热 × E 和查表一样：", np.array_equal(onehot @ E, X))

# 3. 小语料上训练的嵌入：训练前后，和“苹果”“我”最像的词
ex = t["examples"]["22"]
V = ex["vocab"]
def nearest(E, word, k=3):
    En = E / np.linalg.norm(E, axis=1, keepdims=True)
    sims = En @ En[V.index(word)]
    order = [j for j in np.argsort(-sims) if V[j] ≠ word][:k]
    return [(V[j], round(float(sims[j]), 2)) for j in order]
for name, key in [("训练前", "E0"), ("训练后", "E")]:
    Ek = np.array(ex[key])
    print(f"{name}：苹果 →", nearest(Ek, "苹果"), " 我 →", nearest(Ek, "我"))

# 4. 训练时，一步里只有用到的那几行会收到梯度
dX = np.ones_like(X)                                                      # 假设从上
面传下来的梯度
dE = np.zeros_like(E)
np.add.at(dE, ids, dX)                                                    # 反向：梯
度加回到对应的行
print("收到梯度的行：", [vocab[i] for i in range(len(vocab)) if dE[i].any()])

# 5. Llama-3-8B 的嵌入表
Vl, d = 128256, 4096
print(f"嵌入表 {Vl} × {d} = {Vl * d / 1e6:.0f}M 个参数，BF16 下 {Vl * d * 2 /
2**30:.2f} GiB；每个 token 只读其中 {d * 2 / 1024:.0f} KiB")
```

输出：

```
编号： [0, 1, 2, 3]  X 的形状： (4, 4)
[[ 0.7 -0.6 -0.9  0.3]
 [-0.3 -0.1 -0.8 -0.2]
 [ 0.3 -0.3  0.6  0.6]
 [ 0.4  0.8  0.4 -0.6]]
独热 × E 和查表一样： True
训练前：苹果 → [('学', 0.67), ('香蕉', 0.33), ('你', 0.27)]  我 → [('骨头', 0.69), ('推理', 0.47), ('吃', 0.39)]
训练后：苹果 → [('香蕉', 0.99), ('鱼', 0.97), ('AI', 0.93)]  我 → [('你', 0.64), ('骨头', 0.33), ('推理', 0.28)]
收到梯度的行： ['我', '爱', '学', 'AI']
嵌入表 128256 × 4096 = 525M 个参数，BF16 下 0.98 GiB；每个 token 只读其中 8 KiB
```

前几行：“我爱学AI”的编号 [0, 1, 2, 3]，查出 4×4 的 X，和第 3 讲的一样；独热乘 E 的结果和查表完全相同。接着是训练前后的相似词，和正文里的数字一致。然后，假设上面传下来的梯度全是 1，加回嵌入表时，只有“我、爱、学、AI”这 4 行收到了梯度，“推理”和 `<eos>` 没有。最后一行是下一节要用的数。

到了大模型里

Llama-3-8B 的嵌入表是 128256 × 4096，约 5.25 亿个参数，用 BF16 存约 0.98 GiB。每个 token 被表示成 4096 维空间里的一个点。

查表本身几乎不花计算：每个 token 只是从显存里读出 4096 个数，8 KiB。和后面动辄几千万次乘加的线性层相比，可以忽略不计。但整张表必须放在显存里，因为下一个 token 是什么，事先不知道。多卡推理用张量并行（第 105、110 讲）时，嵌入表通常也会按词表切开，分到各张卡上。

嵌入表还有一个“镜像”：模型最后的输出层（第 33 讲），要把一个 4096 维的向量变成 128256 个词的打分，它的权重也是 4096 × 128256。有的模型让这两张表共用同一份参数，叫**权重绑定**（tied embeddings），能省下一份参数；GPT-2 就是这样做的。Llama-3-8B 没有绑定，两张表各自训练，合起来是上一讲算的 10.5 亿个参数。

容易混淆的地方

1. **嵌入的单独一维一般没有明确的意思。**我们说“意思相近的词向量相近”，指的是整个向量的方向和距离，而不是“第 3 维代表动物”。训练只关心整体效果，含义通常分散在很多维度的组合里。

2. **同一个词，嵌入永远是同一个向量**。“苹果”不管出现在“我吃苹果”还是“苹果发布了新手机”里，查表查出来的都是同一行。区分这两种意思，是后面的注意力层的工作（第 26 讲）：它根据上下文，把每个词的向量改造成“在这句话里”的意思。
3. **嵌入里没有位置信息**。“我爱你”和“你爱我”查出来的是同一组向量，只是顺序不同。模型怎么知道谁在前、谁在后，要靠第 25 讲的位置编码和第 37 讲的 RoPE。

回顾

编号本身没有可以计算的含义，独热向量又太长、而且所有词彼此一样远。词嵌入的做法是准备一张 $V \times d$ 的嵌入表，每个 token 一行，用编号查表取出对应的行，一句话就变成一个 $s \times d$ 的矩阵。查表等价于“独热向量 $\times$ E”，所以嵌入层就是一个没有偏置的线性层，表里的每个数都是参数，靠训练得到。训练之后，在上下文中作用相近的词，嵌入也相近：在我们的小语料里，句末的宾语挤成了一团，“我”和“你”靠在一起，而吃的东西不同的“小猫”和“小狗”被区分开了。

我们一直说模型的任务是“看前面的词，预测下一个词”。这个看起来很简单的任务，到底在学什么？为什么只靠它，就能学出嵌入里的“含义”，甚至学会写文章、做推理？下一讲：**预测下一个 token**。

第23讲 预测下一个 token

16 损失函数　22 词嵌入

前两讲把文字变成了 token，又把 token 变成了向量。现在终于可以问那个最根本的问题了：**一个大模型，到底在学什么？**

答案只有一句话：**看前面的文字，预测下一个 token**。GPT、Llama、DeepSeek 这些大模型，预训练阶段做的全是这一件事，没有别的任务。这一讲先把这个任务说清楚：它的输入是什么、输出是什么、一段文本怎么变成训练题目；再用上一讲那个在小语料上训练过的模型，亲手看看它学到了什么；最后讨论一个看上去很奇怪的事实：为什么只靠“猜下一个词”，模型就能学会语法、知识甚至推理。

一个词一个词地猜

假设前面已经有了“我 吃”，下一个词是什么？可能是“苹果”，也可能是“香蕉”，没法确定。所以模型的输出不是一个词，而是一个**概率分布**：词表里每个 token 各有一个概率，加起来是 1。用数学的写法，模型给出的是

$$P(\text{下一个 token} \mid \text{前面所有的 token})$$

竖线读作“在……的条件下”。整个过程是第 16 讲已经讲过的：模型最后一层给词表里的每个 token 打一个分（logit），再用 softmax 变成概率。

“输出一个分布”而不是“输出一个答案”，是理解语言模型的关键。语言本身就是不确定的：同样的开头，可以有很多种合理的接法。一个好的语言模型，不是每次都“猜中”，而是给每种接法分配合理的概率：常见的接法概率高，罕见但合理的接法概率低，不合理的接法概率接近 0。

一句话就是好几道题

这个任务最妙的地方在于：**训练数据不需要任何人工标注**。随便一段文字，本身就带着答案。

以“我 学 AI”为例。按上一讲的小模型，每次看前两个词（句子开头不够两个，用句号“。”补齐），一句话可以拆出 4 道题：

输入（前两个词）	正确答案（下一个词）
。 。	我
。 我	学
我 学	AI
学 AI	。

文本有多少个 token，就有多少道题。互联网上有几万亿个 token 的文字，就是几万亿道现成的、带答案的练习题。这种“从数据本身构造答案”的训练方式，叫**自监督学习**（self-supervised learning）。

每道题的损失是第 16 讲的交叉熵：−ln（模型给正确答案的概率）。整段文本的损失，就是所有位置的交叉熵取平均。训练让这个平均值尽量小，也就是让模型给真实出现的下一个词，分配尽量高的概率。

模型学到的就是统计规律

上一讲的小模型，就是在 16 种句子（重复后一共 24 句、96 道题）上用这个目标训练出来的。下面的演示可以自己拼一个前缀：点词把它接到后面，下面会列出模型对下一个词给出的概率，从高到低排列。最右一列是在训练语料里实际数出来的：同样的两个词后面，各个词出现过几次。



图 23-1　给一个前缀，看下一个词的概率。默认前缀“我 吃”：苹果和香蕉各 0.5，和语料里的 2/4、2/4 一致。前缀超过两个词时，更早的词会变淡：这个小模型只看最后两个词。

试几个前缀：

- “**我 吃**”：苹果 0.501、香蕉 0.499。语料里“我 吃”后面是苹果 2 次、香蕉 2 次。
- “**小猫 吃**”：鱼的概率是 1.0。语料里小猫只吃过鱼。
- “**我 学**”和“**你 学**”：前者 AI 0.667、推理 0.333，后者正好反过来。语料里“我学 AI”出现 2 次、“我学推理”1 次，“你”则相反。

- **清空前缀**（从句子开头预测）：我 0.416、你 0.25、小猫 0.167，对应语料里 24 句话有 10 句以“我”开头、6 句以“你”开头、4 句以“小猫”开头。

模型给出的概率，几乎就是语料里的频率。**训练的结果，是让模型记住了“什么后面跟着什么、跟多少次”的统计规律。**

损失能低到多少？就算模型完全学会了这些频率，“我 吃”后面是苹果还是香蕉，仍然只能五五开，每次都要付出 $-\ln 0.5 \approx 0.69$ 的损失。把所有题目的这种“不可避免的损失”平均起来，就是这份语料的理论下限：0.6681。下面的代码显示，模型的平均交叉熵也是 0.6681，已经降到了下限。对应的困惑度是 1.951：平均来说，模型每一步相当于在不到 2 个词之间犹豫。

> **要点**
> 语言模型学的是条件概率 P(下一个 token | 前文)。训练数据就是普通文本本身，每个位置一道题，损失是交叉熵的平均。损失的下限不是 0，因为语言本来就有不确定性。

在演示里也能看到这个小模型的局限。把前缀拼成“我 喜欢 小猫 。 小猫 吃”，前面的词全部变淡，它只看“小猫 吃”。它永远不知道更早发生了什么。如果语料里有“小猫 在 河边 吃”这样隔了几个词的句子，它就无能为力了。真实的大模型能看前面几千、几万甚至上百万个 token，这正是第 4 篇注意力机制要解决的问题。

为什么猜下一个词这么厉害

在我们的小语料上，“预测下一个词”只能学到简单的搭配。可当语料变成几万亿个 token、包含了百科、教材、代码、对话、论文时，这个任务就变得极其困难，困难到模型必须学会很多东西，才能把损失降下去。看几个例子：

- “法国的首都是 ___”：要给“巴黎”高概率，模型得**记住事实**。
- “他们昨天去了公园，今天又 ___”：要接得通顺，模型得**懂语法和时态**。
- “37 + 25 = ___”：要猜对“62”，模型得**会算术**，至少得学会某种算法。
- “小明打碎了妈妈最喜欢的花瓶，他很 ___”：要给“害怕”“内疚”高概率，模型得**理解人的情绪和因果**。
- 一段 Python 代码写到一半：要猜对下一行，模型得**懂编程**。

任何一种能力的缺失，都会体现为在某类文本上猜不准、损失降不下去。于是，在足够大的模型、足够多的数据上，把这一个目标优化到底，就“逼”出了这些能力。这是大模型最让人意外、也最重要的一个发现：一个极其简单的训练目标，配合足够的规模，可以学出极其复杂的行

为（第 46 讲的 Scaling Laws 会定量地讨论“足够”是多少）。

用代码复现

下面的代码加载上一讲的小模型，先把“我 学 AI”拆成训练题目；然后对几个前缀，把模型的预测和语料里数出来的频率并排打印；最后算出模型的平均交叉熵和理论下限。

```
import json
from collections import Counter, defaultdict
import numpy as np

ex = json.load(open("tinylm.json", encoding="utf-8"))["examples"]["22"]
V = ex["vocab"]
E, W1, b1, W2, b2 = (np.array(ex[k]) for k in ["E", "W1", "b1", "W2", "b2"])

def probs(a, b):                                    # 只看最后两个词
    x = np.concatenate([E[V.index(a)], E[V.index(b)]])
    z = np.maximum(0, x @ W1 + b1) @ W2 + b2        # 14 个词的打分（logit）
    p = np.exp(z - z.max())
    return p / p.sum()                              # softmax

# 1. 一句话怎么变成训练样本：每个位置都要预测下一个词
toks = ["。", "。"] + "我 学 AI".split() + ["。"]
for i in range(2, len(toks)):
    print(f"输入“{toks[i-2]} {toks[i-1]}”→ 目标“{toks[i]}”")

# 2. 语料里数出来的频率
count = defaultdict(Counter)
for sent, n in ex["corpus"]:
    s = ["。", "。"] + sent.split() + ["。"]
    for i in range(2, len(s)):
        count[(s[i-2], s[i-1])][s[i]] += n

# 3. 模型的预测 vs 语料的频率
for a, b in [("。", "。"), ("我", "吃"), ("小猫", "吃"), ("我", "学"), ("你", "学")]:
    p = probs(a, b)
    top = [(V[i], round(float(p[i]), 3)) for i in np.argsort(-p)[:3]]
    tot = sum(count[(a, b)].values())
    freq = {w: f"{c}/{tot}" for w, c in count[(a, b)].most_common(3)}
    print(f"“{a} {b}”→ 模型 {top}  语料 {freq}")

# 4. 训练损失 vs 理论下限（语料本身的不确定性）
pairs = [(a, b, w, c) for (a, b), cnt in count.items() for w, c in cnt.items()]
total = sum(c for *_, c in pairs)
H = -sum(c * np.log(c / sum(count[(a, b)].values())) for a, b, w, c in pairs) /
total
loss = -sum(c * np.log(probs(a, b)[V.index(w)]) for a, b, w, c in pairs) / total
print(f"样本 {total} 个；模型的平均交叉熵 {loss:.4f}，下限 {H:.4f}，困惑度
{np.exp(loss):.3f}")
```

输出：

```
输入“。 。”→ 目标“我”
输入“。 我”→ 目标“学”
输入“我 学”→ 目标“AI”
输入“学 AI”→ 目标“。”
“。 。”→ 模型 [('我', 0.416), ('你', 0.25), ('小猫', 0.167)]  语料 {'我': '10/24',
'你': '6/24', '小猫': '4/24'}
“我 吃”→ 模型 [('苹果', 0.501), ('香蕉', 0.499), ('骨头', 0.0)]  语料 {'苹果': '2/4',
'香蕉': '2/4'}
“小猫 吃”→ 模型 [('鱼', 1.0), ('。', 0.0), ('香蕉', 0.0)]  语料 {'鱼': '3/3'}
“我 学”→ 模型 [('AI', 0.667), ('推理', 0.333), ('。', 0.0)]  语料 {'AI': '2/3', '推
理': '1/3'}
“你 学”→ 模型 [('推理', 0.666), ('AI', 0.334), ('。', 0.0)]  语料 {'推理': '2/3',
'AI': '1/3'}
样本 96 个；模型的平均交叉熵 0.6681，下限 0.6681，困惑度 1.951
```

前四行是“我 学 AI”拆出的 4 道题。中间五行是模型的预测和语料频率的对照，和演示里看到的一致。最后一行：96 道题，模型的平均交叉熵 0.6681，和下限 0.6681 相同，困惑度 1.951。

到了大模型里

预训练就是这一讲的目标，放大到几万亿个 token。Llama-3 在 15T 以上的 token 上做的，就是让平均交叉熵尽量小。第 20 讲算过，这需要约 7×10^{23} 次浮点运算。

一次前向，所有位置的题一起做。我们的小模型每道题单独算一次。Transformer 处理一段 s 个 token 的文本时，一次前向就给出每个位置的下一个词分布，s 道题同时完成，这是训练效率高的关键之一。它能这样做的前提是：预测第 i 个位置时，不能偷看第 i 个位置之后的词（否则答案就泄露了），这由第 27 讲的因果掩码来保证。

推理时只需要最后一个位置。训练时每个位置的预测都有用；推理时，我们只关心“整段话之后接什么”，所以只需要最后一个位置的分布。输出层（第 33 讲）把一个 4096 维的向量变成 128256 个打分，只对最后一个位置做一次就行。但前面的每个 token 仍然要完整地算一遍，因为最后一个位置要“看”到它们，这就是第 68 讲要讲的 prefill。

容易混淆的地方

1. **模型输出的是分布，不是词**。“模型说了某句话”，其实是：模型给出一个分布，再由一个**选词的规则**从分布里挑出一个词。选词规则是独立于模型的（下一讲和第 73 讲），同一个模型配不同的规则，说出来的话可以很不一样。

2. **预训练出来的模型不会“对话”**。只做过“预测下一个 token”的模型（叫基座模型，base model）会接着你的话往下写，而不是回答你的问题。你写“怎么做红烧肉？”，它可能接着写出另外几个菜谱问题，因为网上的问题列表常常就是这样的。会对话、遵循指令，是第 47、48 讲的监督微调和偏好对齐的结果。
3. **“只是统计”不等于“只是记忆”**。我们的小模型确实只记住了 96 道题的频率。但大模型的参数远远不够把训练数据逐字存下来（80 亿个参数，对上 15 万亿个 token），它必须找出数据背后可以复用的规律，才能在没见过的文本上也猜得准。

回顾

大模型预训练只做一件事：看前面的 token，给下一个 token 一个概率分布。训练数据就是普通文本，文本有多少个 token 就有多少道带答案的题，损失是每道题交叉熵的平均。在我们的小语料上，训练出来的概率几乎就等于语料里数出来的频率，损失降到了理论下限 0.6681：语言本身的不确定性让损失不可能是 0。当语料大到包罗万象时，为了猜准下一个 token，模型被迫学会了语法、事实、计算和推理。

可是，一个只会“给下一个 token 一个分布”的模型，是怎么写出一整段话的？从分布里该怎么挑词？挑出来的词又怎么接着用？下一讲：**自回归生成**。

第24讲 自回归生成

06 指数、对数与 softmax　07 概率与采样　23 预测下一个 token

上一讲说，语言模型只做一件事：给下一个 token 一个概率分布。可我们用大模型时，看到的是它写出了一整段话。从“一个分布”到“一整段话”，中间只差两步：**从分布里挑一个 token，再把它接回输入，接着预测下一个**。一直重复，直到模型生成一个表示结束的 token。

这种“用自己刚才的输出作为下一步的输入”的方式，叫**自回归生成**（autoregressive generation）。这一讲用上一讲的小模型，一步一步地看它怎么生成句子，比较两种挑词的方法：贪心和采样；最后会看到，正是“一次只能出一个 token”这个特点，决定了大模型推理的速度瓶颈，也决定了第三章要讲的大半内容。

预测、选词、接回去

自回归生成是一个循环，每一圈做三件事：

1. **前向**：把已有的 token 送进模型，得到下一个 token 的分布。
2. **选词**：按某种规则从分布里挑出一个 token。
3. **接回去**：把它接到已有 token 的末尾。

直到挑出的是结束标记（我们的小模型里是句号“。”，真实模型里是 `<eos>` 这类特殊 token），或者达到了设定的最大长度，循环停止。

用上一讲的小模型，从空白开始，每次挑概率最大的词：

步	模型看的两个词	概率最大的下一个词	现在的句子
1	。 。	我（0.416）	我
2	。 我	吃（0.398）	我 吃
3	我 吃	苹果（0.501）	我 吃 苹果
4	吃 苹果	。（接近 1）	我 吃 苹果 。

4 次前向，生成了一句“我 吃 苹果 。”。注意第 3 步：模型的输入“我 吃”里，“吃”是它在第 2 步自己生成的。每一步的输出都成为下一步输入的一部分，这就是“自回归”的意思。

贪心：每次选最大的

刚才用的挑词规则叫**贪心解码**（greedy decoding）：每一步都选概率最大的那个。它简单、确定：同样的开头，永远生成同样的结果。从空白开始，永远是“我 吃 苹果 。”；从“小狗”开始，永远是“小狗 吃 骨头 。”。

问题也很明显。第 3 步，苹果的概率是 0.501，香蕉是 0.499，几乎一样，可贪心永远选苹果，“我 吃 香蕉”这句完全合理的话，它一辈子也说不出来。同样，“你”开头的句子（概率 0.25）、“小猫”开头的句子，也都不会出现。分布里明明有很多种可能，贪心只取了其中一条。

在大模型里，贪心解码还有一个出名的毛病：容易陷入重复，同一句话说了又说。因为一旦进入某个“高概率”的套路，下一步最可能的又是同一个套路。

采样：按概率抽

另一种规则是**采样**（sampling）：按照分布的概率，随机抽一个（第 7 讲）。苹果和香蕉各有约一半的机会被抽中；“我”开头的概率是 0.416，“你”开头是 0.25……这样，模型能说出的话就和训练语料一样丰富了。下面的代码从空白开始采样了 5 次，得到：你 吃 苹果 / 小猫 喜欢 鱼 / 你 学 推理 / 小狗 吃 骨头 / 小狗 吃 骨头。

采样带来了多样性，也带来了风险：偶尔会抽中一个概率很低、但不太合理的 token，一旦接进去，后面就可能越走越偏。为了控制这种“冒险的程度”，采样之前通常会用**温度**（temperature）T 调整一下分布：

$$p_i = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

也就是先把每个 logit 除以 T，再做 softmax（第 6 讲）。$T=1$ 时就是原来的分布。以句子开头为例：

温度	我	你	小猫	小狗
$T=0.3$	0.782	0.143	0.037	0.037
$T=1$	0.416	0.25	0.167	0.167
$T=3$	0.29	0.245	0.214	0.214

T 小于 1，logit 之间的差距被放大，高的更高、低的更低，分布变“尖”，采样的结果越来越接近贪心；T 趋于 0 时就是贪心。T 大于 1，差距被缩小，分布变“平”，冷门的词也有了不小的机会。代码里 $T = 3$ 采样的 5 句话中，出现了一句“小狗 小猫 骨头 。”：第二个词本该是“吃”或“喜欢”，却抽中了一个在那个位置几乎不可能的词。温度太高，就会这样胡言乱语。

要点

自回归生成 = 前向得到分布 → 按规则选一个 token → 接回输入，重复到结束。贪心每次选最大的，确定但单调；采样按概率抽，丰富但有风险，温度控制分布的尖平。

下面的演示可以亲手生成。选一个开头，点“生成一个词”，下方会显示这一步模型看到的两个词、下一个词的分布，以及选中的那个（琥珀色）；下面的记录写出每一步做了什么。切换到“采样”，调一调温度，多点几次“重来”（每次换一组随机数），看看能生成哪些句子。

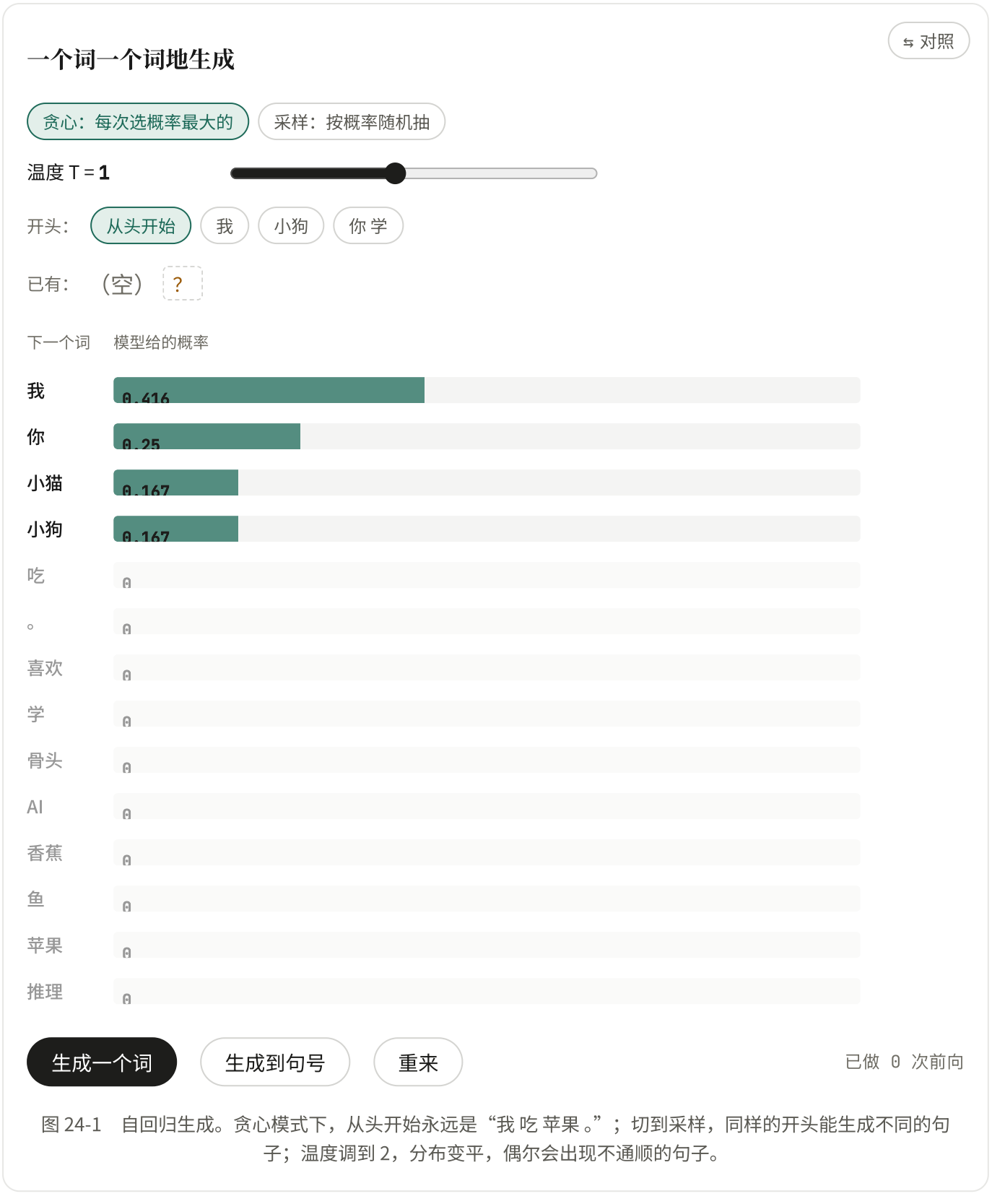


图 24-1　自回归生成。贪心模式下，从头开始永远是“我 吃 苹果 。”；切到采样，同样的开头能生成不同的句子；温度调到 2，分布变平，偶尔会出现不通顺的句子。

一次只能出一个

自回归生成有一个根本的约束：**第 k 个 token 必须等第 k − 1 个 token 生成出来，才能开始算**，因为它的输入里就包含第 k − 1 个 token。所以生成 n 个 token，就要做 n 次前向，一次接一次，没法并行。这和训练时完全不同：训练时整段文本都是已知的，所有位置的“下一个词”可以在一次前向里同时算（上一讲）。

我们的小模型只看最后两个词，每一步的计算量是固定的。可第 4 篇的 Transformer 会看前面所有的 token。如果每一步都把整段话重新完整地算一遍，越往后要算的越多：提示有 1000 个 token、再生成 1000 个，每一步分别要处理 1000、1001、1002……个 token，一共要处理约 150 万个 token 位置。

但仔细想，前面那些 token 在上一步已经算过了，它们的中间结果并没有变。把这些中间结果存下来，每一步只算新来的那一个 token，总共只需要处理 1000 + 999 = 1999 个位置。这就是第 69 讲的 **KV Cache**：用显存换计算，是大模型推理里最重要的一项优化。至于具体要存什么、存多少，要等第 4 篇讲完注意力之后才能说清楚。

用代码复现

下面的代码实现了自回归生成的循环，分别用贪心和不同温度的采样生成句子；然后打印温度对分布的影响；最后算两个推理的工程数字（下一节用）。

```
import json
import numpy as np

ex = json.load(open("tinylm.json", encoding="utf-8"))["examples"]["22"]
V = ex["vocab"]
E, W1, b1, W2, b2 = (np.array(ex[k]) for k in ["E", "W1", "b1", "W2", "b2"])

def logits(seq):                                    # 只看最后两个词，不够就用“。”补
    a, b = (["。", "。"] + seq)[-2:]
    x = np.concatenate([E[V.index(a)], E[V.index(b)]])
    return np.maximum(0, x @ W1 + b1) @ W2 + b2

def softmax(z, T=1.0):
    p = np.exp((z - z.max()) / T)
    return p / p.sum()

def generate(start, how="greedy", T=1.0, rng=None, max_len=10):
    seq = list(start)
    while len(seq) < max_len:
        p = softmax(logits(seq), T)                 # ① 前向：得到下一个词的分布
        if how == "greedy":
            nxt = V[int(np.argmax(p))]              # ② 选词：概率最大的
        else:
            nxt = V[rng.choice(len(V), p=p)]        # ② 选词：按概率抽
        seq.append(nxt)                             # ③ 接回输入
        if nxt == "。":
            break
    return " ".join(seq)

print("贪心，从头开始：", generate([]))
print("贪心，从“小狗”开始：", generate(["小狗"]))
print("贪心，从“你 学”开始：", generate(["你", "学"]))
for T in [1.0, 0.3, 3.0]:
    rng = np.random.default_rng(0)
    print(f"采样 T = {T}：", " | ".join(generate([], "sample", T, rng) for _ in
range(5)))

# 温度对分布的影响：前缀为空时
z = logits([])
for T in [0.3, 1.0, 3.0]:
    p = softmax(z, T)
    print(f"T = {T}：", {V[i]: round(float(p[i]), 3) for i in np.argsort(-p)[:4]})

# 没有缓存时的重复计算：提示 1000 个 token，再生成 1000 个
p_len, n = 1000, 1000
no_cache = sum(p_len + k for k in range(n))         # 每一步都把整段重新算一遍
```

```
print(f"不缓存：共处理 {no_cache:,} 个 token 位置；缓存：{p_len + n - 1:,} 个")

# 解码一个 token 至少要把全部权重从显存读一遍（Llama-3-8B，BF16；H100 显存带宽约 3.35 TB/s）
w_bytes, bw = 8.03e9 * 2, 3.35e12
print(f"每个 token 读 {w_bytes / 1e9:.1f} GB 权重：至少 {w_bytes / bw * 1e3:.1f} ms，单条请求最多约 {bw / w_bytes:.0f} token/s")
```

输出：

```
贪心，从头开始：  我 吃 苹果 。
贪心，从“小狗”开始：  小狗 吃 骨头 。
贪心，从“你 学”开始：  你 学 推理 。
采样 T = 1.0：  你 吃 苹果 。 | 小猫 喜欢 鱼 。 | 你 学 推理 。 | 小狗 吃 骨头 。 | 小狗 吃 骨头 。
采样 T = 0.3：  我 吃 苹果 。 | 你 学 推理 。 | 我 学 AI 。 | 你 吃 香蕉 。 | 你 学 推理 。
采样 T = 3.0：  小猫 吃 鱼 。 | 小狗 喜欢 骨头 。 | 小猫 喜欢 鱼 。 | 小狗 小猫 骨头 。 | 小狗 吃 骨头 。
T = 0.3：  {'我': 0.782, '你': 0.143, '小猫': 0.037, '小狗': 0.037}
T = 1.0：  {'我': 0.416, '你': 0.25, '小猫': 0.167, '小狗': 0.167}
T = 3.0：  {'我': 0.29, '你': 0.245, '小猫': 0.214, '小狗': 0.214}
不缓存：共处理 1,499,500 个 token 位置；缓存：1,999 个
每个 token 读 16.1 GB 权重：至少 4.8 ms，单条请求最多约 209 token/s
```

贪心的三行：从空白、“小狗”、“你 学”开始，分别得到“我 吃 苹果 。”“小狗 吃 骨头 。”“你 学 推理 。”，每次运行都一样。采样的三行，每行 5 句：$T=1$ 时句子多种多样；$T=0.3$ 时偏向高概率的句子；$T=3$ 时出现了“小狗 小猫 骨头 。”这样的病句。接着是三个温度下句子开头的分布。最后两行是下一节的数字。演示里的随机数和这里的 NumPy 不是同一套，采样出来的具体句子会不一样。

到了大模型里

推理分两个阶段。你把一段提示发给大模型，它先把整段提示一次性送进去做前向，这叫 prefill；然后才开始一个一个地生成，这叫 decode（第 68 讲）。prefill 可以并行处理成百上千个 token，主要受限于算力；decode 每一步只处理一个新 token，却要把整个模型的权重都用一遍。

decode 受限于显存带宽。每生成一个 token，模型的每一个权重都要从显存读到计算单元里参与一次乘法。Llama-3-8B 的权重用 BF16 存是 16.1 GB，H100 的显存带宽约 3.35 TB/s，光是把权重读一遍就要至少 4.8 毫秒。所以单独一条请求，每秒最多也就生成约 209 个 token，不管

GPU 的算力有多强。这个“搬数据比算还慢”的现象，第 62 讲的 Roofline 模型会讲透。

解决办法是把很多请求凑在一起。既然每一步都要把权重读一遍，那就让这一遍读进来的权重，同时服务几十、上百个请求：每个请求各出一个 token，权重只读一次。这就是批处理，第 74 讲的 Continuous Batching 让它在请求有长有短、随时进出的情况下也能高效进行。衡量推理快慢的指标也由此而来：第一个 token 出来要多久（TTFT），之后每个 token 隔多久（TPOT），以及整个系统每秒能生成多少 token（吞吐），见第 70 讲。

容易混淆的地方

1. **温度为 0 不是真的除以 0**。各种推理框架里“temperature = 0”都按贪心处理，直接取概率最大的。所以把温度设成 0，同样的输入会得到同样的输出（在不考虑浮点误差的前提下）。
2. **同样的问题得到不同的回答，是采样造成的，不是模型“变了”**。模型的权重在推理时不会改变，给同样的输入，分布永远是同一个。不同的是从分布里抽出了哪个 token，而一个 token 的不同，会让后面的整段话走向不同的方向。
3. **温度不是唯一的选词手段**。实际中常和 Top-k（只在概率最高的 k 个里抽）、Top-p（只在累计概率达到 p 的那些里抽）一起用，把概率极低的“离谱”选项直接排除。这些留到第 73 讲。

回顾

自回归生成是一个循环：把已有的 token 送进模型得到分布，按规则选一个，接到末尾，再来一遍，直到生成结束标记。贪心每次选概率最大的，结果确定但单调；采样按概率随机抽，结果丰富，温度 T 控制分布的尖平：T 越小越接近贪心，T 太大会说出病句。生成 n 个 token 要做 n 次前向，一次只能出一个；这让大模型推理的 decode 阶段受限于显存带宽，也催生了 KV Cache、批处理这些第三章的核心技术。

第 3 篇到这里结束。我们的小模型有一个致命的短板：它只看最后两个词。“我 喜欢 小猫 。 小猫 吃”，它对前面发生的事一无所知。真正的大模型要能看到前面成千上万个 token，并且知道每个词和哪些词有关。这是第 4 篇 Transformer 的任务。不过在讲注意力之前，还有一个小问题要先解决：第 22 讲说过，嵌入里没有位置信息，“我爱你”和“你爱我”查出来的是同一组向量。模型怎么知道每个词在第几个位置？下一讲：**位置编码**。

AI INFRA · HANDBOOK

第 四 篇

Transformer 核心

收录 11 讲 · 第 25—35 讲

第25讲 位置编码

09 三角函数与旋转　22 词嵌入

第 3 篇的小模型只看最后两个词。从这一篇开始，我们要搭一个能看到整段上下文的模型：Transformer。它的核心是下一讲的**自注意力**：每个词去“看”句子里的其他词，按相关程度把它们的信息收集过来。

不过在讲注意力之前，要先解决一个它自带的缺陷：**注意力本身不知道词的顺序**。“我爱你”和“你爱我”，在它眼里是同一堆词。第 22 讲也说过，嵌入里没有位置信息。这一讲回答：怎么把“第几个位置”这个信息告诉模型？答案是给每个位置也准备一个向量，加到词嵌入上，叫**位置编码**（positional encoding）。

注意力是“一袋词”

下一讲会细讲注意力的公式，这里只需要知道一件事：它给每个词算输出时，是根据**词的内容**去和其他词比较，决定从谁那里收集多少信息。公式里没有任何地方用到“这个词在第几个”。

结果是：把输入的词打乱顺序，输出也只是跟着同样地打乱，每个词得到的结果一个数都不变。下面的代码拿 TinyLM 验证了这一点：“我 爱 学 AI”倒过来变成“AI 学 爱 我”，算出来的四个输出向量，正好是原来那四个倒过来排。对注意力来说，句子就像一个装着词的袋子，顺序被完全丢掉了。

这对语言是致命的。“狗咬人”和“人咬狗”用的是同样三个字。

为了看清后果，我们训练了一个小模型（代码在 `tinylm.py` 的 `lecture_27`，第 26–28 讲会仔细拆开它）。它的任务是**复读**：输入“a b c 。”，a、b、c 是随机抽的三个词，然后它要接着把“a b c”再说一遍。要复读，模型在第二遍说到第 k 个词时，必须找到第一遍的第 k 个词，也就是**按位置**去找。训练好的模型，复读的正确率是 100%。把它的位置信息拿掉，正确率掉到 16.4%：它知道前面有哪几个词，却不知道哪个在前、哪个在后。

给每个位置一个向量

办法很直接：除了词嵌入表 E，再准备一张**位置编码表** P，第 p 行是“第 p 个位置”的向量，维度和词嵌入一样。每个 token 的输入，是它的词嵌入加上它所在位置的向量：

$$\mathbf{x}_p = E[\text{token}] + P[p]$$

这样，同一个词出现在不同位置，输入的向量就不一样了；两个不同的词，也因为位置不同而带上了各自的“位置标签”。

P 怎么来？最简单的做法是把它也当成参数，和词嵌入一起训练出来，叫**可学习的位置嵌入**（learned positional embedding）。我们的复读模型就是这样：P 是一张 7×8 的表，随机初始化，训练中学出了各个位置的向量。GPT-2 也是这样做的。它的缺点是长度被表的大小写死了：表里只有 1024 行，就只能处理 1024 个 token，再长就没有对应的位置向量了。

正弦位置编码

2017 年提出 Transformer 的论文用了另一种不需要训练的位置编码，用正弦和余弦函数直接算出来：

$$P(\text{p}, 2\text{i}) = \sin(\text{p} / 10000^{2\text{i}/\text{d}}), \qquad P(\text{p}, 2\text{i}+1) = \cos(\text{p} / 10000^{2\text{i}/\text{d}})$$

p 是位置，d 是向量的维度，i 从 0 开始。向量的维度两两一组：第 0、1 维是一组，第 2、3 维是一组……每组用同一个“频率”，一个取 sin，一个取 cos。分母 $10000^{2\text{i}/\text{d}}$ 随着组号 i 增大而变大，所以越往后的组，随位置变化得越慢。

拿 TinyLM（d = 4）算一下。两组的分母分别是 $10000^{0} = 1$ 和 $10000^{2/4} = 100$：

位置 p	sin(p)	cos(p)	sin(p/100)	cos(p/100)
0	0	1	0	1
1	0.841	0.540	0.010	1.000
2	0.909	−0.416	0.020	1.000
3	0.141	−0.990	0.030	1.000

前两维变化很快，每个位置都明显不同；后两维变化很慢，在 4 个位置里几乎不动。把这张表加到“我 爱 学 AI”的嵌入上，“我”变成 (0.7, 0.4, −0.9, 1.3)，“爱”变成 (0.541, 0.44, −0.79, 0.8)……每个词都被打上了位置的印记。

为什么要这样设计？可以把它想成一个由很多指针组成的钟表：最快的指针每走几步就转一圈，最慢的指针走几千步才转一圈。只看最快的指针，位置 0 和位置 6 差不多（sin 的周期约是 6.28），分不清；再看慢一些的指针，就分开了。所有指针合起来，每个位置的读数都独一无二。这和用二进制给数字编号是同一个道理：最低位每次都变，越高的位变得越慢。第 9 讲讲过的三角函数和旋转，正是这里的基础。

下面的演示画出了 d = 64 时，64 个位置的完整编码。每一行是一个位置，每一列是一维；左边几列（高频）条纹很密，右边几列（低频）几乎是一整片。选一个位置，右上方是它这一行的前 16 个数，右下方是它和每个位置的编码做点积的结果。

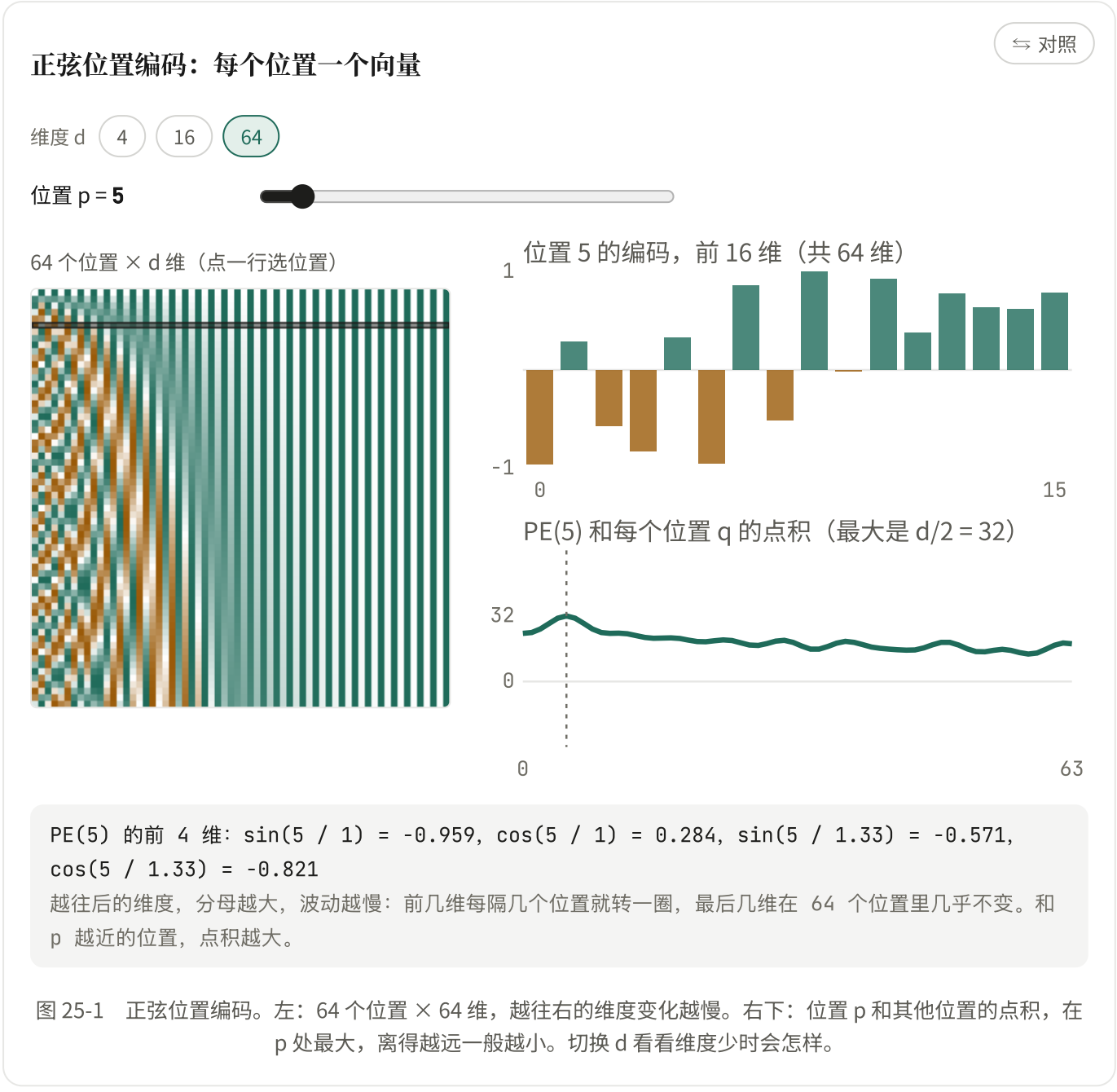


图 25-1　正弦位置编码。左：64 个位置 × 64 维，越往右的维度变化越慢。右下：位置 p 和其他位置的点积，在 p 处最大，离得越远一般越小。切换 d 看看维度少时会怎样。

右下那条曲线说明了正弦编码的一个好性质：**两个位置离得越近，编码越相似**。位置 10 和自己的点积是 32（d/2），和相隔 1 的位置是 30.92，相隔 5 是 23.5，相隔 30 是 17.07。模型因此能从编码里读出“这两个词离得有多远”。另一个好性质是，它对任何位置都能算，不受训练时长度的限制（当然，模型在没见过的长度上表现好不好，是另一回事）。

要点

位置编码 = 每个位置一个向量，加到词嵌入上，让模型能区分词的顺序。可以训练出来（可学习的位置嵌入），也可以用不同频率的正弦、余弦直接算出来（正弦位置编码）。

用代码复现

下面的代码先验证注意力“不看顺序”；然后比较复读模型有和没有位置嵌入时的正确率；最后算出 d = 4 的正弦编码，加到 TinyLM 的嵌入上，并看 d = 64 时点积随距离的变化。

```
import json
import numpy as np

t = json.load(open("tinylm.json", encoding="utf-8"))

# 1. 注意力不看顺序：把输入的行打乱，输出的行跟着同样地打乱，数值一个不变
X = np.array(t["examples"]["03"]["X"])                # “我 爱 学 AI”，4×4
L0 = t["weights"]["layers"][0]
Wq, Wk, Wv = (np.array(L0[k]) for k in ["W_q", "W_k", "W_v"])
def attention(X):                                     # 下一讲的主角，这里先当黑盒
    S = (X @ Wq) @ (X @ Wk).T / 2
    A = np.exp(S - S.max(1, keepdims=True)); A /= A.sum(1, keepdims=True)
    return A @ (X @ Wv)
perm = [3, 2, 1, 0]                                   # 倒过来：“AI 学 爱 我”
print("打乱后的输出 = 输出打乱：", np.allclose(attention(X[perm]), attention(X)
[perm]))

# 2. 复读模型：有位置嵌入和没有位置嵌入
ex = t["examples"]["27"]
p = {k: np.array(ex[k]) for k in ["E", "P", "Wq", "Wk", "Wv", "Wo", "WU"]}
rng = np.random.default_rng(5)
tri = rng.integers(1, 14, (500, 3))                   # 500 组随机的三个词
seq = np.concatenate([tri, np.zeros((500, 1), int), tri], 1)   # a b c 。 a b c
def copy_predict(seq, use_pos):                       # 复读模型的前向，结构见第 26-28
讲
    T = seq.shape[1]
    x = p["E"][seq] + (p["P"][:T] if use_pos else 0)  # 词嵌入 + 位置嵌入
    Q, K, V = x @ p["Wq"], x @ p["Wk"], x @ p["Wv"]
    outs = []
    for h in range(2):                                # 两个头，每头 4 维
        s = slice(4 * h, 4 * h + 4)
        S = Q[..., s] @ K[..., s].transpose(0, 2, 1) / 2 + np.triu(np.full((T, T),
-1e9), 1)
        A = np.exp(S - S.max(-1, keepdims=True)); A /= A.sum(-1, keepdims=True)
        outs.append(A @ V[..., s])
    return ((x + np.concatenate(outs, -1) @ p["Wo"]) @ p["WU"]).argmax(-1)
for use_pos in [True, False]:
    pred = copy_predict(seq, use_pos)[:, 3:6]
    print(("有" if use_pos else "没有") + "位置嵌入：复读正确率", np.mean(pred ==
seq[:, 4:7]))

# 3. 正弦位置编码
def sinusoid(n, d):
    pos = np.arange(n)[:, None]
    freq = 1 / 10000 ** (np.arange(0, d, 2) / d)      # 每两维一个频率
    P = np.zeros((n, d))
```

```
    P[:, 0::2], P[:, 1::2] = np.sin(pos * freq), np.cos(pos * freq)
    return P
print("d = 4，位置 0-3：")
print(np.round(sinusoid(4, 4), 3))
print("加到“我 爱 学 AI”上：")
print(np.round(X + sinusoid(4, 4), 3))

# 4. d = 64：位置 10 和相隔 k 的位置的点积，随距离变小
P64 = sinusoid(64, 64)
print({k: round(float(P64[10] @ P64[10 + k]), 2) for k in [0, 1, 2, 5, 10, 30]})
```

输出：

```
打乱后的输出 = 输出打乱： True
有位置嵌入：复读正确率 1.0
没有位置嵌入：复读正确率 0.164
d = 4，位置 0-3：
[[ 0.     1.     0.     1.   ]
 [ 0.841  0.54   0.01   1.   ]
 [ 0.909 -0.416  0.02   1.   ]
 [ 0.141 -0.99   0.03   1.   ]]
加到“我 爱 学 AI”上：
[[ 0.7    0.4   -0.9    1.3  ]
 [ 0.541  0.44  -0.79   0.8  ]
 [ 1.209 -0.716  0.62   1.6  ]
 [ 0.541 -0.19   0.43   0.4  ]]
{0: 32.0, 1: 30.92, 2: 28.3, 5: 23.5, 10: 21.05, 30: 17.07}
```

第一行：打乱输入后的输出，和输出打乱后的结果完全相同。接下来两行：有位置嵌入时复读全对，没有时只对 16.4%。然后是 d = 4 的正弦编码表，以及加到“我 爱 学 AI”上之后的结果。最后一行：位置 10 和相隔 0、1、2、5、10、30 个位置的点积，越远越小。

到了大模型里

位置编码的做法，经历了三代：

做法	代表模型	特点
正弦位置编码	原始 Transformer（2017）	不用训练，直接相加
可学习的位置嵌入	GPT-2、BERT	训练出来，长度被表的大小限制
旋转位置编码 RoPE	Llama、DeepSeek 等	不加到输入上，而是在注意力里旋转 Q 和 K

今天的主流是第 37 讲要讲的 **RoPE**。它保留了正弦编码“不同频率”的思想，但不是把位置向量加到词嵌入上，而是在算注意力时，把每个词的 Q、K 向量按位置旋转一个角度，让两个词的点积只和它们的**相对距离**有关。Llama-3-8B 的 config 里那个 `rope_theta` 参数，就是这一讲公式里的 10000（Llama-3 把它调大到了 500000，让低频的“指针”转得更慢，以支持更长的上下文）。

对推理系统来说，位置也是一个需要小心处理的细节。生成时，每个新 token 的位置等于前面已有的 token 数；把不同长度的请求凑成一批时，每个请求的位置要各算各的，不能混。推理引擎里专门有一个叫 position ids 的输入来记录这件事。

容易混淆的地方

1. **位置编码是加上去的，不是拼上去的**。位置向量和词嵌入维度相同，直接相加，向量的长度不变。一个向量里同时混着“是什么词”和“在哪个位置”两种信息，由后面的层自己去分辨。
2. **位置指的是 token 的序号**。“我爱学AI”如果被切成 4 个 token，位置就是 0、1、2、3，和字数无关。
3. **有了位置编码，注意力本身还是不看顺序的**。注意力的公式没有变，变的是输入：因为每个词的向量里已经带了位置，“按内容比较”时，位置也自然成了比较的一部分。复读模型就是靠这个，找到了“第一遍的第 k 个词”。

回顾

自注意力按内容比较词与词，对顺序完全不敏感：打乱输入，输出只是跟着打乱。所以要在输入里加入位置：给每个位置一个和词嵌入同样长的向量，加到词嵌入上。这个向量可以训练出来（GPT-2），长度受表的大小限制；也可以用正弦、余弦直接算出来（原始 Transformer）：不同的维度对应不同的频率，像快慢不同的钟表指针，合起来给每个位置一个独一无二、并且近处相似的编码。我们的复读模型，有位置嵌入时全对，拿掉后只剩 16.4%。今天的大模型用的是它的升级版 RoPE。

现在，每个 token 的向量里既有“是什么”，也有“在哪里”。下一讲进入 Transformer 的核心：每个词怎么根据这些向量，去决定该看句子里的哪些词、从它们那里拿走什么。下一讲：**自注意力**。

第26讲 自注意力

02 点积与余弦相似度　03 矩阵乘法　06 指数、对数与 softmax　22 词嵌入

到目前为止，句子里的每个词都是一个孤立的向量：“苹果”查表查出来的永远是同一行，不管它出现在“我吃苹果”还是“苹果发布了新手机”里。可一个词的意思，取决于它周围的词。第 3 篇的小模型只能看最后两个词，也是同样的问题。

这一讲介绍 Transformer 的核心部件：**自注意力**（self-attention）。它让句子里的每个词都能去“看”其他所有词，判断哪些词和自己有关，再把这些词的信息按相关程度收集过来，更新自己的向量。整个过程只用到我们学过的东西：矩阵乘法（第 3 讲）、点积（第 2 讲）和 softmax（第 6 讲）。我们先用 TinyLM 把每一步算一遍，再看一个训练过的模型，它的注意力真正学会了“看哪里”。

查询、键、值

先打个比方。你去图书馆找资料，心里有一个**问题**（我想找关于什么的书）；每本书的书脊上有一个**标签**（这本书讲什么）；书里是它的**内容**。你拿问题和每本书的标签比一比，越对得上的书，你越仔细地读它的内容；最后你脑子里得到的，是各本书内容按“对得上的程度”加权混合的结果。

自注意力里，每个词同时扮演这三个角色，各用一个向量表示：

- **查询**（query，$\mathbf{q}$）：我在找什么样的信息。
- **键**（key，$\mathbf{k}$）：我能提供什么样的信息，用来和别人的查询比对。
- **值**（value，$\mathbf{v}$）：如果别人看我，我交给他的具体内容。

这三个向量都从这个词自己的向量 $\mathbf{x}$ 算出来，各乘一个权重矩阵（就是三个第 13 讲的线性层，不带偏置）：

$$Q = X \cdot W_{\mathrm{q}}, \qquad K = X \cdot W_{\mathrm{k}}, \qquad V = X \cdot W_{\mathrm{v}}$$

X 是整句话的嵌入，每行一个词（第 22 讲）。TinyLM 的 X 是“我 爱 学 AI”的 4×4 矩阵，三个权重矩阵也都是 4×4，所以 Q、K、V 都是 4×4，每一行是一个词的 q、k 或 v。第 3 讲算的 $X \cdot W$，其实就是这里的 $Q = X \cdot W_{\mathrm{q}}$：“我”的 q 是 $(-0.34, 0.18, -0.16, 0.11)$。

为什么要分成三个向量，而不是直接用 x 去比？因为“我在找什么”和“我能提供什么”通常是两回事。比如一个代词“它”，它要找的是前面出现过的名词，而它自己能提供的，是“这里有一个代词”这样的信息。三个不同的矩阵，让模型可以分别学习这三种角色。

打分：q 和 k 的点积

有了 q 和 k，每个词就可以给句子里的每个词打分：拿自己的 q 和对方的 k 做点积。第 2 讲说过，点积越大，两个向量越“对得上”。

以“我”为例。它的 q 和“我”自己的 k = (0.64, −0.27, −1.73, 0.8) 做点积：

$$(-0.34) \times 0.64 + 0.18 \times (-0.27) + (-0.16) \times (-1.73) + 0.11 \times 0.8 = -0.2176 - 0.0486$$

和“爱”“学”“AI”的 k 做点积，分别是 0.142、−0.0606、−0.1876。4 个词各给 4 个词打分，一共 16 个分数，排成一个 4 × 4 的矩阵，正好就是 Q 乘以 K 的转置：

$$S = Q \cdot K^{\mathrm{T}}, \qquad S_{\mathrm{ij}} = \text{第 i 个词的 q} \cdot \text{第 j 个词的 k}$$

K^{T} 把 K 的行变成列，按第 3 讲的“行乘列”，Q 的第 i 行和 K^{T} 的第 j 列做点积，就是 $\mathrm{q_i} \cdot \mathrm{k_j}$。一次矩阵乘法，算完了所有词两两之间的分数。

缩放、softmax、加权求和

分数还不能直接用，还有三步。

缩放。把每个分数除以 $\sqrt{d_{\mathrm{k}}}$，d_{k} 是 q 和 k 的维度。TinyLM 里 $d_{\mathrm{k}} = 4$，除以 2：“我”的四个分数变成 0.0493、0.071、−0.0303、−0.0938。原因下一节专门说。

softmax。对每一行做 softmax（第 6 讲），把分数变成加起来等于 1 的权重。“我”这一行得到

“我”分给 我、爱、学、AI 的权重 = 0.262，0.268，0.242，0.227

这一行叫“我”的**注意力权重**：“我”在更新自己时，各个词分别占多大比重。所有行排在一起，就是 4×4 的注意力矩阵 A。

加权求和。用权重把各个词的 v 加起来，就是这个词的输出：

$$\text{“我”的输出} = 0.262 \cdot \mathrm{v}_{\text{我}} + 0.268 \cdot \mathrm{v}_{\text{爱}} + 0.242 \cdot \mathrm{v}_{\text{学}} + 0.227 \cdot \mathrm{v}_{\text{AI}} = (-0.361, -0.12, -0.005, -0.175)$$

四个词一起算，就是 A 乘 V。把这几步写成一个式子，就是 Transformer 论文里最有名的公式：

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \cdot V$$

要点

自注意力 = 每个词用自己的 q 和所有词的 k 做点积打分，缩放后 softmax 变成权重，再用权重把所有词的 v 加起来。一句话里所有词同时这样做，就是两次矩阵乘法加一次 softmax。

下面的演示把这些都摆了出来。左边是注意力矩阵 A，颜色越深权重越大；右边是 X、Q、K、V。点 A 的某一行，下方会逐步写出这一行的计算：q、打分、缩放、softmax、加权求和。

自注意力：每个词给其他词打分，再加权求和

⇆ 对照

注意力权重 A：第 i 行是“第 i 个词”分给每个词的权重，每行加起来是 1。点一行看它怎么算

	我 1	爱 2	学 3	AI 4
我 1	0.26	0.27	0.24	0.23
爱 2	0.24	0.25	0.25	0.25
学 3	0.29	0.24	0.23	0.24
AI 4	0.2	0.27	0.3	0.23

X（4×4）

我	**0.7**	**-0.6**	**-0.9**	**0.3**
爱	-0.3	-0.1	-0.8	-0.2
学	0.3	-0.3	0.6	0.6
AI	0.4	0.8	0.4	-0.6

Q = *X* · *W_q*（4×4）

我	**-0.34**	**0.18**	**-0.16**	**0.11**
爱	0.09	0.07	0.13	0.13
学	-0.15	-0.21	-0.51	-0.51
AI	-0.18	0.32	0.7	0.86

K = *X* · *W_k*（4×4）

我	0.64	-0.27	-1.73	0.8
爱	0.03	0.17	-0.54	0.32
学	-0.09	-0.51	0.12	0.18
AI	0.28	0.56	0.52	-1

V = *X* · *W_v*（4×4）

我	-1.74	-0.34	-0.39	-0.64
爱	-0.6	-0.33	-0.23	-0.27
学	-0.18	0.33	-0.15	0.27
AI	1.32	-0.1	0.86	0

第 1 个词“我”作为提问者

① 它的 q = (-0.34, 0.18, -0.16, 0.11)

② 和每个词的 k 做点积：$q \cdot k_1 = 0.099$，$q \cdot k_2 = 0.142$，$q \cdot k_3 = -0.061$，$q \cdot k_4 = -0.188$

③ 除以 √4 = 2：0.049, 0.071, -0.03, -0.094

④ softmax 变成权重：我 0.262，爱 0.268，学 0.242，AI 0.227

⑤ 输出 = $0.26 \times v_1 + 0.27 \times v_2 + 0.24 \times v_3 + 0.23 \times v_4$ = (-0.361, -0.12, -0.005, -0.175)

权重分得很平均：TinyLM 的权重是随机的，还没学会“该看谁”。

图 26-1　TinyLM 的自注意力（一个头，d_k = 4）。每行的权重都在 0.2 到 0.3 之间，分得很平均：TinyLM 的权重是随机的，还没有学会该看谁。

你会发现每一行的权重都差不多，在 0.2 到 0.3 之间。这不是算错了：TinyLM 的权重是随机生成的，从来没训练过，q 和 k 之间没有什么有意义的对应关系，分数都很接近 0，softmax 之后就差不多平均分配了。注意力“看哪里”，是训练出来的。

为什么要除以 $\sqrt{d_k}$

第 14、16 讲说过，softmax 在输入差距很大时会“饱和”：最大的那个几乎拿走全部权重，其余接近 0，梯度也随之消失。点积恰好有这个风险。

如果 q 和 k 的每个分量都是均值 0、方差 1 的随机数，它们的点积是 d_k 个乘积的和，方差是 d_k，标准差是 $\sqrt{d_k}$。维度越高，分数的波动越大。代码里实际模拟了一下：$d_k = 4$ 时点积的标准差是 2，64 时约 8，128 时约 11。一个 q 对 8 个 k 打分后直接 softmax，128 维时最大的那个平均能拿到 0.92 的权重，几乎是一边倒；除以 $\sqrt{d_k}$ 后，分数的标准差回到 1 左右，最大权重降到 0.38，分布柔和得多，训练也更稳定。Llama-3 每个头的 d_k 是 128，这一步缩放是必需的。

训练过的注意力长什么样

为了看到“学会了看哪里”的注意力，我们用上一讲的**复读模型**。它的输入是“a b c 。 a b c”，任务是在句号之后把 a、b、c 复读一遍。每个位置的输入是词嵌入加位置嵌入（8 维），注意力有两个头（第 28 讲细讲，这里先看第 1 个头），而且只能看自己和前面的词（第 27 讲细讲）。

训练过的复读模型：每个头在看哪里

⇆ 对照

输入 a b c：我 学 AI　模型要在“。”之后把它复读一遍

看哪个头　头 1　头 2

注意力权重 A（头 1）：第 i 行是“第 i 个词”分给每个词的权重，每行加起来是 1。点一行看它怎么算

	我 1	学 2	AI 3	。4	我 5	学 6	AI 7
我 1	1						
学 2	0	1					
AI 3	0.02	0	0.98				
。4	1	0	0	0			
我 5	0	1	0	0	0		
学 6	0	0	1	0	0	0	
AI 7	0	0.01	0.66	0.03	0	0	0.3

每个位置预测的下一个词：鱼　AI　。　我　学　AI　。

第 4–6 个是复读，绿色表示和第一遍一致

第 5 个词“我”作为提问者（头 1，每头 4 维）

① 它的 q = (-4.09, -3.16, -0.78, 1.54)

② 和每个词的 k 做点积：$q\cdot k_1$ = -3.991，$q\cdot k_2$ = 14.166，$q\cdot k_3$ = -9.933，$q\cdot k_4$ = -8.486，$q\cdot k_5$ = -2.95，$q\cdot k_6$ = 2.814，$q\cdot k_7$ = 1.244

③ 除以 √4 = 2：-1.995，7.083，-4.966，-4.243，-1.475，1.407，0.622

④ 因果掩码：排在它后面的 2 个词的分数换成 -∞

⑤ softmax 变成权重：我 0，学 1，AI 0，。 0，我 0，学 0，AI 0

⑥ 输出 = $1\times v_2$ = (3.027, -1.464, -1.537, -1.77)

两个头的输出拼起来是 8 维：(3.03, -1.46, -1.54, -1.77, -0.18, 0.35, 0.7, 0.84)，再乘 W_o 得到这个词的最终输出 (-0.91, -0.28, -3.43, 2.12, 4.31, -6.2, -0.67, 2.32)

这一行最大的权重 1 落在第 2 个词“学”上。

图 26-2　训练过的复读模型（头 1）。第 4、5、6 行（句号和第二遍的 a、b）的权重几乎全部集中在第 1、2、3 个词上：要复读下一个词，就去看第一遍对应的那个词。换几个词试试，图案不变。

注意第 4、5、6 行。第 4 个位置是句号“。”，它要预测的下一个词是 a，于是它把约 1.0 的权重放在了第 1 个词 a 上；第 5 个位置（第二遍的 a）要预测 b，权重集中在第 2 个词 b 上；第 6 个位置要预测 c，权重集中在第 3 个词 c 上。每个位置都精确地“回头看 3 格”，把那里的词拿过来，复读出去。下方是模型对每个位置下一个词的预测，第二遍的三个全对。

用上面的下拉框换成别的词，比如“推理 鱼 你”，权重图案几乎不变：这个头学到的是“按位置去找”，和具体是什么词无关。上一讲拿掉位置嵌入后复读失败，原因就在这里：没有位置，这个头就不知道“回头 3 格”是哪一格。

用代码复现

下面的代码完整地算了一遍 TinyLM 的自注意力；然后用随机向量看看为什么要除以 $\sqrt{d_k}$；最后估算 Llama-3-8B 一层注意力在长序列下的计算量和注意力矩阵的大小。

```python
import json
import numpy as np

t = json.load(open("tinylm.json", encoding="utf-8"))
X = np.array(t["examples"]["03"]["X"])            # “我 爱 学 AI”的嵌入，4×4
L0 = t["weights"]["layers"][0]
Wq, Wk, Wv = (np.array(L0[k]) for k in ["W_q", "W_k", "W_v"])

def softmax(S):
    e = np.exp(S - S.max(axis=-1, keepdims=True))
    return e / e.sum(axis=-1, keepdims=True)

# 1. 三次矩阵乘法得到 Q、K、V
Q, K, V = X @ Wq, X @ Wk, X @ Wv
print("“我”的 q：", Q[0].round(2), " “我”的 k：", K[0].round(2))

# 2. 打分、缩放、softmax、加权求和
d_k = Q.shape[1]
S = Q @ K.T                                       # 4×4：第 i 行第 j 列 = q_i · k_j
A = softmax(S / np.sqrt(d_k))                     # 每行加起来是 1
O = A @ V                                         # 每个词的输出 = V 各行按权重相加
print("“我”的打分：", S[0].round(4), " 除以 √4：", (S[0] / 2).round(4))
print("注意力权重 A：")
print(A.round(3))
print("“我”的输出：", O[0].round(3), " 每行和：", A.sum(axis=1).round(6))

# 3. 为什么要除以 √d_k：随机向量的点积，标准差随维度变大
rng = np.random.default_rng(0)
for d in [4, 64, 128]:
    q, k = rng.standard_normal((10000, d)), rng.standard_normal((10000, d))
    dots = (q * k).sum(axis=1)
    s = rng.standard_normal((1000, 8, d)) @ rng.standard_normal((d, 1))    # 一个 q 对 8 个 k 的打分
    top = softmax(s[..., 0]).max(axis=1).mean()
    top_scaled = softmax(s[..., 0] / np.sqrt(d)).max(axis=1).mean()
    print(f"d = {d:3d}：点积的标准差 {dots.std():5.2f} (√d = {np.sqrt(d):5.2f})   8 个里最大权重：不缩放 {top:.2f}，缩放后 {top_scaled:.2f}")

# 4. Llama-3-8B 一层注意力，序列长 8192：投影和打分各要多少计算，注意力矩阵多大
s, d, n_h = 8192, 4096, 32
proj = 2 * s * d * (d + 1024 + 1024 + d)          # q、o 是 4096×4096，k、v 是 4096×1024（第 39 讲）
attn = 2 * (2 * s * s * d)                        # QKᵀ 和 A·V
print(f"投影 {proj / 1e12:.2f} TFLOPs，打分+加权 {attn / 1e12:.2f} TFLOPs；注意力矩阵 {n_h} 个头 × {s}² × 2 字节 = {n_h * s * s * 2 / 2**30:.0f} GiB")
```

输出：

```
“我”的 q： [-0.34  0.18 -0.16  0.11] “我”的 k： [ 0.64 -0.27 -1.73  0.8 ]
“我”的打分： [ 0.0986  0.142  -0.0606 -0.1876] 除以 √4： [ 0.0493  0.071  -0.0303
-0.0938]
注意力权重 A：
[[0.262 0.268 0.242 0.227]
 [0.243 0.251 0.252 0.253]
 [0.289 0.241 0.228 0.242]
 [0.196 0.274 0.295 0.235]]
“我”的输出： [-0.361 -0.12  -0.005 -0.175] 每行和： [1. 1. 1. 1.]
d =   4：点积的标准差  2.00（√d =  2.00）  8 个里最大权重：不缩放 0.52，缩放后 0.32
d =  64：点积的标准差  8.06（√d =  8.00）  8 个里最大权重：不缩放 0.87，缩放后 0.36
d = 128：点积的标准差 11.23（√d = 11.31）  8 个里最大权重：不缩放 0.92，缩放后 0.38
投影 0.69 TFLOPs，打分+加权 1.10 TFLOPs；注意力矩阵 32 个头 × 8192² × 2 字节 = 4 GiB
```

前几行：“我”的 q 和 k，“我”对四个词的打分 0.0986、0.142、−0.0606、−0.1876，除以 2 后的结果，以及整个注意力矩阵（第一行就是正文里的 0.262、0.268、0.242、0.227）和“我”的输出。每行的权重和都是 1。中间三行是缩放的作用。最后一行是下一节的数字。

到了大模型里

计算量：随序列长度平方增长。打分矩阵是 s × s 的：每个词都要和每个词比一次。Llama-3-8B 的一层注意力，处理 s = 8192 个 token 时，算 Q、K、V 和输出投影的四个矩阵乘法约要 0.69 TFLOPs，而打分（QK^{T}）加加权求和（AV）要 1.10 TFLOPs，已经超过了投影。前者和 s 成正比，后者和 s^2 成正比：序列再长一倍，投影翻倍，注意力翻四倍。这就是长上下文昂贵的根本原因。

显存：注意力矩阵很大。同样在 s = 8192 时，一层 32 个头的注意力矩阵，每个都是 8192 × 8192，用 BF16 存要 4 GiB，而它只是一个中间结果。第 84 讲的 FlashAttention 用分块计算的办法，根本不把这个矩阵完整地存下来，既省显存，又因为少了大量读写而更快。

推理时的 K 和 V。生成新 token 时，新 token 的 q 要和前面所有 token 的 k 打分，再用它们的 v 加权。前面那些 token 的 k、v 是不变的（下一讲会说明为什么），所以可以存起来反复用。这就是第 24 讲预告过的 KV Cache 里存的东西：每一层、每个 token 的 k 和 v。

容易混淆的地方

1. **“注意”不是“选一个”**。注意力的输出是所有 v 的加权平均，每个词都有贡献，只是多少不同。即使权重集中在某一个词上，也是“几乎全部来自它”，而不是挑出它。
2. **Q、K、V 来自同一个 X**。“自”注意力的“自”，就是指查询、键、值都来自同一句话。还有一种“交叉注意力”，查询来自一个序列、键和值来自另一个序列（比如翻译模型里，译文去看原文），大语言模型里基本只用自注意力。
3. **注意力本身没有可学的“看哪里”的规则**。学的只是 W_{q}、W_{k}、W_{v} 三个矩阵。看哪里，是 q 和 k 的点积在运行时临时算出来的，所以同一个模型面对不同的句子，注意力图案也不同。

回顾

自注意力让每个词根据上下文更新自己。每个词用三个矩阵算出查询 q、键 k、值 v；用自己的 q 和所有词的 k 做点积打分，除以 $\sqrt{d_{\mathrm{k}}}$ 防止 softmax 饱和，softmax 成权重，再用权重把所有词的 v 加起来，就是它的输出。写成矩阵就是 $\mathrm{softmax}(QK^{\mathrm{T}}/\sqrt{d_{\mathrm{k}}})\cdot V$。TinyLM 没训练过，注意力几乎平均分配；训练过的复读模型则学会了“回头看 3 格”。注意力的计算量和显存都随序列长度平方增长，这是长上下文的主要成本。

复读模型有一个我们一直没解释的细节：它的每个词只看自己和前面的词，注意力矩阵的右上角是空的。为什么要这样？如果不遮住，会发生什么？下一讲：**因果掩码**。

第27讲 因果掩码

重点

23 预测下一个 token　24 自回归生成　26 自注意力

上一讲的复读模型有一个细节：它的注意力矩阵右上角全是空的，每个词只看自己和前面的词，从不看后面的词。上一讲的 TinyLM 演示则没有这个限制，每个词都看了整句话。

这一讲解释这个限制是什么、为什么语言模型必须有它。它叫**因果掩码**（causal mask）：在注意力里把“后面的词”遮住。它的实现只有一行代码，却直接决定了两件大事：一是训练时能不能一次前向就把一整段话的所有位置都训练了；二是推理时能不能用 KV Cache。后者是整个推理优化的基石，所以这一讲标了重点。

生成的时候，后面还不存在

先想想生成。第 24 讲说过，模型一个词一个词地往外吐：写第 5 个词时，第 6 个词还不存在，模型只能根据第 1 到第 5 个词来决定。

模型在生成时面对的条件，训练时就得一样。否则就像考试时开卷、平时练习却从不允许翻书，或者反过来，练的和考的根本不是一回事。

训练时，答案就在旁边

问题出在训练。第 23 讲说过，训练时整段文本是已知的，Transformer 一次前向，就给每个位置都预测下一个词：第 i 个位置的输出，用来预测第 i + 1 个词。

可是自注意力会让第 i 个位置去看句子里**所有**的词，包括第 i + 1 个，那正是它要预测的答案。不加限制的话，模型很快就会发现一条捷径：根本不用理解前文，直接“看一眼下一个词，照抄”就行了，训练损失能降得非常低。等到生成时，下一个词还不存在，这条捷径用不上，模型就什么也不会了。

我们可以直接验证“后面的词会影响前面”。用 TinyLM，把第 4 个词“AI”换成另一个向量，看前 3 个词的注意力输出变不变：不加掩码，前 3 个词的输出最多变了 0.3804，说明它们确实“看到”了第 4 个词。加上掩码，变化是 0：前面的词对后面发生了什么一无所知。

掩码怎么加

做法很简单：在 softmax **之前**，把分数矩阵里“j > i”的那些位置（第 i 个词看排在它后面的第 j 个词），全部换成 −∞：

$$A = \mathrm{softmax}(QK^{\mathrm{T}} / \sqrt{d_k} + M), \qquad M_{ij} = 0 \ (j \leqslant i), \ -\infty \ (j > i)$$

$e^{-\infty} = 0$，所以 softmax 之后，这些位置的权重恰好是 0，其余位置的权重重新归一，加起来仍然是 1。被保留的是矩阵的**下三角**（含对角线），所以也叫下三角掩码。

拿 TinyLM 的注意力看看加掩码前后的变化：

提问的词	不加掩码	加因果掩码
我（第 1 个）	0.262, 0.268, 0.242, 0.227	1, 0, 0, 0
爱（第 2 个）	0.243, 0.251, 0.252, 0.253	0.491, 0.509, 0, 0
学（第 3 个）	0.289, 0.241, 0.228, 0.242	0.381, 0.318, 0.301, 0
AI（第 4 个）	0.196, 0.274, 0.295, 0.235	0.196, 0.274, 0.295, 0.235

第 1 个词只能看自己，权重只能是 1；第 2 个词在“我”和“爱”之间分配；最后一个词本来就没有“后面”，加不加掩码都一样。

注意掩码必须加在 softmax 之前。如果先 softmax、再把后面的权重改成 0，剩下的权重加起来就不到 1 了，而且分数里已经混进了后面的词的信息（softmax 的分母用到了它们）。

下面的演示可以开关掩码。打开后，矩阵的右上角变成斜线格，下方的逐步计算里会多出“掩码”这一步。

自注意力：每个词给其他词打分，再加权求和

⇆ 对照

☐ 因果掩码：每个词只能看自己和前面的词

注意力权重 A：第 i 行是“第 i 个词”分给每个词的权重，每行加起来是 1。点一行看它怎么算

	我 1	爱 2	学 3	AI 4
我 1	0.26	0.27	0.24	0.23
爱 2	0.24	0.25	0.25	0.25
学 3	0.29	0.24	0.23	0.24
AI 4	0.2	0.27	0.3	0.23

X（4×4）

我	**0.7**	**-0.6**	**-0.9**	**0.3**
爱	-0.3	-0.1	-0.8	-0.2
学	0.3	-0.3	0.6	0.6
AI	0.4	0.8	0.4	-0.6

Q = X · W_q（4×4）

我	**-0.34**	**0.18**	**-0.16**	**0.11**
爱	0.09	0.07	0.13	0.13
学	-0.15	-0.21	-0.51	-0.51
AI	-0.18	0.32	0.7	0.86

K = X · W_k（4×4）

我	0.64	-0.27	-1.73	0.8
爱	0.03	0.17	-0.54	0.32
学	-0.09	-0.51	0.12	0.18
AI	0.28	0.56	0.52	-1

V = X · W_v（4×4）

我	-1.74	-0.34	-0.39	-0.64
爱	-0.6	-0.33	-0.23	-0.27
学	-0.18	0.33	-0.15	0.27
AI	1.32	-0.1	0.86	0

第 1 个词“我”作为提问者

① 它的 q = (-0.34, 0.18, -0.16, 0.11)

② 和每个词的 k 做点积：$q \cdot k_1 = 0.099$，$q \cdot k_2 = 0.142$，$q \cdot k_3 = -0.061$，$q \cdot k_4 = -0.188$

③ 除以 √4 = 2：0.049, 0.071, -0.03, -0.094

④ softmax 变成权重：我 0.262，爱 0.268，学 0.242，AI 0.227

⑤ 输出 = $0.26 \times v_1 + 0.27 \times v_2 + 0.24 \times v_3 + 0.23 \times v_4$ = (-0.361, -0.12, -0.005, -0.175)

权重分得很平均：TinyLM 的权重是随机的，还没学会“该看谁”。

图 27-1　开关因果掩码。打开后，每个词只能看自己和前面的词：第 1 行只剩自己（权重 1），第 4 行不受影响。

要点

因果掩码：softmax 之前，把每个词对后面的词的分数设成 −∞，权重就变成 0。这样第 i 个位置的输出只依赖第 1 到 i 个词，训练时的条件和生成时一致。

一次前向，做完所有的题

有了因果掩码，训练就可以放心地“一次前向，所有位置一起算”了。第 i 个位置的输出只依赖前 i 个词，它预测第 i + 1 个词时看不到答案，和生成时的条件完全一样。一段 8192 个 token 的文本，一次前向就是 8192 道合法的“猜下一个词”的题。

这一点非常重要。早期的循环神经网络（RNN）必须一个词一个词地顺序处理，训练时没法并行；Transformer 靠“注意力 + 因果掩码”，让训练在序列长度这个维度上也能并行，这是它能在海量数据上训练的关键原因之一。

KV Cache 为什么能成立

现在回到推理。这一节是这一讲最重要的部分。

因果掩码保证了：**第 i 个词的输出，只依赖第 1 到 i 个词**。往后面接新的词，前面的词的输出一个数都不会变。这对每一层都成立：第 1 层前 i 个词的输出不变，作为第 2 层的输入也就不变，第 2 层前 i 个词的输出也不变……一直到最后一层。所以，**每一层里，前面每个词的 k 和 v 一旦算出来，就永远不会再变**。

既然不变，就不必每一步都重算。把它们存起来，叫 KV Cache。生成第 i + 1 个词时：

1. 只把新来的这一个词送进模型，在每一层只算它自己的 q、k、v。
2. 把它的 k、v 追加到缓存里。
3. 用它的 q，去和缓存里所有词的 k 打分，用权重把缓存里所有的 v 加起来。

第 3 步里，新词本来就是最后一个，它能看的正是“自己和前面所有的词”，所以**这一步不需要掩码**。分数也不再是 s × s 的矩阵，而是 1 × s 的一行。

下面的代码验证了这件事：把“我 爱 学”的 k、v 当成缓存，只为“AI”算 q、k、v，用这一行算出的注意力权重 (0.196, 0.274, 0.295, 0.235) 和输出，和整句加掩码一起算出的最后一行完全相同。

如果没有因果掩码，前面每个词的输出都会随着后面的词变化，每生成一个词，就得把整段从头算一遍，第 24 讲算过，生成 1000 个词要多处理上百万个 token 位置。

用代码复现

下面的代码算了加掩码前后的注意力权重；验证改最后一个词会不会影响前面；模拟了一步用 KV Cache 的解码；最后算出 Llama-3-8B 的 KV Cache 有多大。

```python
import json
import numpy as np

t = json.load(open("tinylm.json", encoding="utf-8"))
X = np.array(t["examples"]["03"]["X"])                        # “我 爱 学 AI”
L0 = t["weights"]["layers"][0]
Wq, Wk, Wv = (np.array(L0[k]) for k in ["W_q", "W_k", "W_v"])

def softmax(S):
    e = np.exp(S - S.max(axis=-1, keepdims=True))
    return e / e.sum(axis=-1, keepdims=True)

def attention(X, causal):
    Q, K, V = X @ Wq, X @ Wk, X @ Wv
    S = Q @ K.T / np.sqrt(Q.shape[1])
    if causal:
        S = S + np.triu(np.full(S.shape, -np.inf), k=1)      # 右上角（j > i）加上 -∞
    A = softmax(S)
    return A, A @ V

# 1. 加掩码前后的注意力权重
for causal in [False, True]:
    print("因果掩码" + ("开" if causal else "关") + ": ")
    print(attention(X, causal)[0].round(3))

# 2. 改最后一个词，前面的输出会不会跟着变
X2 = X.copy(); X2[3] = X[2]                                   # 把“AI”换成“学”的向量
for causal in [False, True]:
    changed = np.abs(attention(X, causal)[1][:3] - attention(X2, causal)[1]
[:3]).max()
    print("掩码" + ("开" if causal else "关") + f": 改了第 4 个词，前 3 个词的输出最多变了 {changed:.4f}")

# 3. 生成时的一步：新词的 q 去看缓存好的 K、V，结果和整句一起算的最后一行完全一样
Q, K, V = X @ Wq, X @ Wk, X @ Wv
K_cache, V_cache = K[:3], V[:3]                               # 前 3 个词早就算好、存起来了
q_new, k_new, v_new = Q[3], K[3], V[3]                        # 新来的第 4 个词只算自己的 q、k、v
K_all, V_all = np.vstack([K_cache, k_new]), np.vstack([V_cache, v_new])
a = softmax(q_new @ K_all.T / 2)                              # 1×4：新词对 4 个词的权重，不需要掩码
print("一步解码: ", a.round(3), (a @ V_all).round(3))
print("和整句算的最后一行一致: ", np.allclose(a @ V_all, attention(X, True)[1][3]))

# 4. Llama-3-8B 的 KV Cache：每个 token 每层存 8 个 KV 头的 k 和 v（每头 128 维，BF16）
```

```
per_token = 2 * 32 * 8 * 128 * 2
print(f"每个 token {per_token // 1024} KiB；8192 个 token {per_token * 8192 /
2**30:.0f} GiB")
```

输出：

```
因果掩码关：
[[0.262 0.268 0.242 0.227]
 [0.243 0.251 0.252 0.253]
 [0.289 0.241 0.228 0.242]
 [0.196 0.274 0.295 0.235]]
因果掩码开：
[[1.    0.    0.    0.   ]
 [0.491 0.509 0.    0.   ]
 [0.381 0.318 0.301 0.   ]
 [0.196 0.274 0.295 0.235]]
掩码关：改了第 4 个词，前 3 个词的输出最多变了 0.3804
掩码开：改了第 4 个词，前 3 个词的输出最多变了 0.0000
一步解码： [0.196 0.274 0.295 0.235] [-0.249 -0.083  0.018 -0.12 ]
和整句算的最后一行一致： True
每个 token 128 KiB；8192 个 token 1 GiB
```

前两段是加掩码前后的权重矩阵，和上面的表一致。接下来两行：不加掩码时改第 4 个词，前 3 个词的输出变了 0.3804；加掩码时变化为 0。然后是一步解码：新词的权重和输出，与整句计算的最后一行一致。最后一行是下一节的数字。

到了大模型里

KV Cache 有多大？ Llama-3-8B 有 32 层，每层有 8 个 KV 头（第 39 讲的 GQA，比 32 个查询头少），每个头 128 维，k 和 v 各一份，用 BF16 存：每个 token 要 $2 \times 32 \times 8 \times 128 \times 2$ 字节 = 128 KiB。一个请求有 8192 个 token，就是 1 GiB；同时服务 50 个这样的请求，光 KV Cache 就要 50 GiB，比模型权重本身（16 GB）还大得多。第三章里大量的工作，都是在想办法管好这块显存：第 69 讲讲 KV Cache 本身，第 71 讲算显存账，第 75 讲的 PagedAttention 解决它的碎片问题。

prefill 和 decode 的注意力长得不一样。处理提示时（prefill），整段提示一起算，用的是 $s \times s$ 的打分矩阵加因果掩码，和训练时一样；之后每生成一个词（decode），用的是 $1 \times s$ 的一行，不需要掩码，但要把整个缓存读一遍。两者的计算特点截然不同，推理引擎常常为它们准备不同的注意力算子（第 68 讲）。

容易混淆的地方

1. **“因果”和因果关系无关**。这里的“因果”只是说“前面可以影响后面，后面不能影响前面”，像时间的方向一样。有的资料也叫它“自回归掩码”或“未来掩码”（look-ahead mask）。
2. **不是所有 Transformer 都用因果掩码**。BERT 这类用来理解文本（而不是生成文本）的模型，每个词可以看整句话，没有因果掩码，叫双向注意力。它们也就不能像 GPT 那样一个词一个词地生成。
3. **因果掩码和填充掩码是两回事**。把长短不同的句子凑成一批时，短的句子后面会补上无意义的填充 token，要用另一个掩码（padding mask）把它们遮住，防止真实的词去看它们。推理引擎里两种掩码常常同时存在。

回顾

生成时后面的词还不存在，所以训练时也不能让模型看到后面的词，否则它会学会“偷看答案”。因果掩码在 softmax 之前把每个词对后面的词的分数设成 $-\infty$，权重就成了 0，注意力矩阵只剩下三角。有了它，训练可以一次前向就做完一段文本里所有位置的预测；而且第 i 个词的输出只依赖前 i 个词，后面接再多词也不会变，于是每一层每个词的 k、v 都可以缓存起来，这就是 KV Cache。Llama-3-8B 每个 token 的 KV Cache 是 128 KiB，长序列、多请求时，它往往比模型权重还占显存。

复读模型有两个头，第 1 个头“回头看 3 格”，那第 2 个头在干什么？为什么要把注意力拆成好几个头，而不是用一个大的头？下一讲：**多头注意力**。

第28讲 多头注意力

26 自注意力　27 因果掩码

前两讲的注意力，每个词只算一组权重：它“看”句子里每个词的程度只有一套。可一个词和其他词的关系往往不止一种。读到“他把书还给了她，因为她明天要考试”里的第二个“她”时，你会同时想到好几件事：这个“她”指的是前面的哪个人（指代）；“考试”是谁的事（主语）；这一句和上一句是什么关系（因果）。只用一组权重，很难同时兼顾这几件事。

多头注意力（multi-head attention）的做法是：把注意力拆成好几份，叫好几个**头**（head），每个头各算一套 q、k、v 和注意力权重，各看各的，最后把结果拼起来。这一讲先在 TinyLM 上把拆分和拼接算一遍，再看复读模型的两个头各学会了什么，最后看看 Llama-3 的 32 个头。

把向量切成几段

多头注意力不需要新的权重矩阵。$Q = XW_{\mathrm{q}}$、$K = XW_{\mathrm{k}}$、$V = XW_{\mathrm{v}}$ 照常算，只是算完以后，**按列切成 $\mathrm{n_h}$ 段**，每段交给一个头。TinyLM 的 $\mathrm{d} = 4$，拆成 2 个头，每个头 $d_{\mathrm{h}} = 2$ 维：Q 的前 2 列给头 1，后 2 列给头 2，K、V 也一样。

每个头用自己那 2 列，独立地做一遍上一讲的全套计算：打分、除以 $\sqrt{d_{\mathrm{h}}}$（现在是 $\sqrt{2}$）、因果掩码、softmax、加权求和。每个头得到一个 4×2 的输出。

两个头的输出**拼接**（concatenate）起来，又是 4×4；最后再乘一个 4×4 的输出矩阵 W_{o}，把各个头的信息混合起来，就是这一层注意力的最终输出：

$$\text{输出} = [\text{head}_1, \text{head}_2] \cdot W_{\mathrm{o}}$$

拿 TinyLM 算一下（带因果掩码）。两个头的注意力权重是：

提问的词	头 1	头 2
我	1, 0, 0, 0	1, 0, 0, 0
爱	0.504, 0.496, 0, 0	0.484, 0.516, 0, 0
学	0.321, 0.32, 0.359, 0	0.414, 0.32, 0.266, 0
AI	0.222, 0.266, 0.231, 0.281	0.197, 0.265, 0.337, 0.201

两个头的权重已经不一样了：“学”这一行，头 1 最看重自己（0.359），头 2 最看重“我”（0.414）。因为两个头用的是 q、k 的不同的列，比较的是向量的不同“侧面”。不过 TinyLM 没有训练过，这些差别没有什么含义，也都不大。

下面的演示可以切换两个头，右边的 Q、K、V 只显示当前这个头用到的 2 列。下方的计算最后一行，写出了两个头拼接后乘 W_O 的结果。

两个头，各算各的注意力

⇆ 对照

看哪个头 头 1 头 2

注意力权重 A（头 1）：第 i 行是“第 i 个词”分给每个词的权重，每行加起来是 1。点一行看它怎么算

	我 1	爱 2	学 3	AI 4
我 1	1			
爱 2	0.5	0.5		
学 3	0.32	0.32	0.36	
AI 4	0.22	0.27	0.23	0.28

X（4×4）

我	0.7	-0.6	-0.9	0.3
爱	-0.3	-0.1	-0.8	-0.2
学	0.3	-0.3	0.6	0.6
AI	0.4	0.8	0.4	-0.6

Q（头 1）（4×2）

我	-0.34	0.18
爱	0.09	0.07
学	-0.15	-0.21
AI	-0.18	0.32

K（头 1）（4×2）

我	0.64	-0.27
爱	0.03	0.17
学	-0.09	-0.51
AI	0.28	0.56

V（头 1）（4×2）

我	-1.74	-0.34
爱	-0.6	-0.33
学	-0.18	0.33
AI	1.32	-0.1

第 1 个词“我”作为提问者（头 1，每头 2 维）

① 它的 q = (-0.34, 0.18)

② 和每个词的 k 做点积：$q \cdot k_1$ = -0.266，$q \cdot k_2$ = 0.02，$q \cdot k_3$ = -0.061，$q \cdot k_4$ = 0.006

③ 除以 √2 = 1.414：-0.188，0.014，-0.043，0.004

④ 因果掩码：排在它后面的 3 个词的分数换成 -∞

⑤ softmax 变成权重：我 1，爱 0，学 0，AI 0

⑥ 输出 = $1 \times v_1$ = (-1.74, -0.34)

两个头的输出拼起来是 4 维：(-1.74, -0.34, -0.39, -0.64)，再乘 W_o 得到这个词的最终输出 (-0.39, -0.28, 0.48, 0.18)

图 28-1　TinyLM 拆成两个头，每头 2 维。切换头 1、头 2：同一句话，两套不同的注意力权重。

要点

多头注意力 = 把 Q、K、V 按列切成 n_h 段，每段一个头，各自独立做注意力，输出拼接起来再乘 W_o。头数多了，参数并没有变多。

参数没有变多

一个容易误会的地方：多头并不意味着多了几倍的参数。W_q、W_k、W_v 还是 d × d，W_o 也是 d × d，和一个头时一模一样，TinyLM 里每个都是 16 个参数。变的只是**怎么用**：一个头时，4 维的 q 和 k 整体做一次点积；两个头时，前 2 维和后 2 维各做一次点积，各自 softmax。

计算量也几乎一样。打分时，一个头是每对词做一次 4 维点积；两个头是每对词做两次 2 维点积，乘法次数相同。多出来的只是 softmax 要做两遍。

既然不花额外的代价，多头就是一个很划算的设计：把一次“看”拆成几次“从不同角度看”，每个头的权重可以集中在不同的地方。一个头的权重是一个 softmax，它很难同时把很大的权重放在两个毫不相关的词上；两个头就可以各盯一个。

两个头各学会了什么

TinyLM 没训练过，看不出头的分工。看复读模型：它的 d = 8，拆成 2 个头，每头 4 维。

训练过的复读模型：每个头在看哪里

⇆ 对照

输入 a b c：我 学 AI　模型要在“。”之后把它复读一遍

看哪个头　头 1　头 2

注意力权重 A（头 1）：第 i 行是“第 i 个词”分给每个词的权重，每行加起来是 1。点一行看它怎么算

	我 1	学 2	AI 3	。 4	我 5	学 6	AI 7
我 1	1						
学 2	0	1					
AI 3	0.02	0	0.98				
。 4	1	0	0	0			
我 5	0	1	0	0	0		
学 6	0	0	1	0	0	0	
AI 7	0	0.01	0.66	0.03	0	0	0.3

每个位置预测的下一个词：鱼　AI　。　我　学　AI　。

第 4–6 个是复读，绿色表示和第一遍一致

第 5 个词“我”作为提问者（头 1，每头 4 维）

① 它的 q = (-4.09, -3.16, -0.78, 1.54)

② 和每个词的 k 做点积：$q\cdot k_1$ = -3.991，$q\cdot k_2$ = 14.166，$q\cdot k_3$ = -9.933，$q\cdot k_4$ = -8.486，$q\cdot k_5$ = -2.95，$q\cdot k_6$ = 2.814，$q\cdot k_7$ = 1.244

③ 除以 √4 = 2：-1.995，7.083，-4.966，-4.243，-1.475，1.407，0.622

④ 因果掩码：排在它后面的 2 个词的分数换成 -∞

⑤ softmax 变成权重：我 0，学 1，AI 0，。 0，我 0，学 0，AI 0

⑥ 输出 = $1\times v_2$ = (3.027, -1.464, -1.537, -1.77)

两个头的输出拼起来是 8 维：(3.03, -1.46, -1.54, -1.77, -0.18, 0.35, 0.7, 0.84)，再乘 W_o 得到这个词的最终输出 (-0.91, -0.28, -3.43, 2.12, 4.31, -6.2, -0.67, 2.32)

这一行最大的权重 1 落在第 2 个词“学”上。

图 28-2　复读模型的两个头。头 1：第 4–6 行回头看第 1–3 个词，负责复读。头 2：第 4–6 行几乎都盯着第 4 个词“。”。切换头、换几个词看看。

头 1 我们上一讲见过：在第二遍时，每个位置“回头看 3 格”，把第一遍对应的词搬过来。

切到头 2，图案完全不同：第 4、5、6 行的权重几乎都落在第 4 个词，也就是句号“。”上。它像是在标记“复读从这里开始”。

两个头哪个更重要？下面的代码做了一个实验：把某个头的输出强行置 0（相当于关掉它），看复读的正确率。两个头都在时是 100%；关掉头 1，正确率跌到 8.1%，几乎完全失败；关掉头 2，正确率还是 100%。

也就是说，在这个任务上，头 2 其实可有可无，所有的工作都是头 1 做的。这不是我们的小模型特有的现象。研究者在大模型里也发现：不同的头确实学会了各种不同的模式，有的看前一个词，有的看句首，有的跟踪指代关系；但同时，相当一部分头是冗余的，剪掉之后效果几乎不变。多头给了模型“分工”的机会，模型并不一定把每个头都用满。

用代码复现

下面的代码写了一个通用的多头因果注意力函数：先在 TinyLM 上拆成 2 个头，打印两个头的权重和最终输出；然后在复读模型上看两个头各看哪里，并分别关掉一个头，比较复读的正确率。

```
import json
import numpy as np

t = json.load(open("tinylm.json", encoding="utf-8"))

def softmax(S):
    e = np.exp(S - S.max(axis=-1, keepdims=True))
    return e / e.sum(axis=-1, keepdims=True)

def multi_head(X, Wq, Wk, Wv, Wo, n_heads, off=()):
    """多头因果自注意力：Q、K、V 按列切成 n_heads 份，各算各的，拼起来再乘 W_o。off 里的头输出置 0。"""
    s, d = X.shape[-2], Wq.shape[1]
    d_h = d // n_heads
    Q, K, V = X @ Wq, X @ Wk, X @ Wv
    mask = np.triu(np.full((s, s), -np.inf), k=1)
    heads, weights = [], []
    for h in range(n_heads):
        c = slice(h * d_h, (h + 1) * d_h)                    # 第 h 个头用第 h 段列
        A = softmax(Q[..., c] @ np.swapaxes(K[..., c], -1, -2) / np.sqrt(d_h) + mask)
        weights.append(A)
        heads.append(np.zeros_like(V[..., c]) if h in off else A @ V[..., c])
    return np.concatenate(heads, axis=-1) @ Wo, weights      # 拼接：[s, d_h] × n_heads → [s, d]

# 1. TinyLM：d = 4 拆成 2 个头，每头 2 维
X = np.array(t["examples"]["03"]["X"])
L0 = t["weights"]["layers"][0]
Wq, Wk, Wv, Wo = (np.array(L0[k]) for k in ["W_q", "W_k", "W_v", "W_o"])
out, (A1, A2) = multi_head(X, Wq, Wk, Wv, Wo, 2)
print("头 1 的权重：\n", A1.round(3), "\n头 2 的权重：\n", A2.round(3))
print("“AI”的最终输出：", out[3].round(3), "  参数：W_q、W_k、W_v、W_o 各", Wq.size, "个，和一个头时一样")

# 2. 复读模型：两个头各看哪里，关掉一个头会怎样
ex = t["examples"]["27"]
p = {k: np.array(ex[k]) for k in ["E", "P", "Wq", "Wk", "Wv", "Wo", "WU"]}
vocab = ex["vocab"]
def run(seq, off=()):
    x = p["E"][seq] + p["P"][: seq.shape[-1]]
    o, W = multi_head(x, p["Wq"], p["Wk"], p["Wv"], p["Wo"], 2, off)
    return ((x + o) @ p["WU"]).argmax(-1), W
seq = np.array([vocab.index(w) for w in "我 学 AI 。 我 学 AI".split()])
_, (H1, H2) = run(seq)
print("头 1，第 4-6 行最大权重的位置：", (H1[3:6].argmax(1) + 1).tolist(), "  头 2：",
```

```
(H2[3:6].argmax(1) + 1).tolist())
rng = np.random.default_rng(5)
tri = rng.integers(1, 14, (500, 3))
seqs = np.concatenate([tri, np.zeros((500, 1), int), tri], 1)
for off, name in [((), "两个头都在"), ((0,), "关掉头 1"), ((1,), "关掉头 2")]:
    pred = run(seqs, off)[0][:, 3:6]
    print(f"{name}: 复读正确率 {np.mean(pred == seqs[:, 4:7]):.3f}")

# 3. Llama-3-8B: 32 个查询头 × 128 维 = 4096
d, n_h = 4096, 32
print(f"Llama-3-8B: {n_h} 个头 × {d // n_h} 维 = {d}; 每层每个 token 有 {n_h} 行注意力
权重")
```

输出:

```
头 1 的权重:
 [[1.    0.    0.    0.   ]
 [0.504 0.496 0.    0.   ]
 [0.321 0.32  0.359 0.   ]
 [0.222 0.266 0.231 0.281]]
头 2 的权重:
 [[1.    0.    0.    0.   ]
 [0.484 0.516 0.    0.   ]
 [0.414 0.32  0.266 0.   ]
 [0.197 0.265 0.337 0.201]]
“AI”的最终输出:  [-0.019 -0.067  0.138  0.02 ]   参数: W_q、W_k、W_v、W_o 各 16 个，和一
个头时一样
头 1，第 4-6 行最大权重的位置:  [1, 2, 3]   头 2:  [4, 4, 4]
两个头都在: 复读正确率 1.000
关掉头 1: 复读正确率 0.081
关掉头 2: 复读正确率 1.000
Llama-3-8B: 32 个头 × 128 维 = 4096; 每层每个 token 有 32 行注意力权重
```

前两个矩阵是 TinyLM 两个头的注意力权重，和上面的表一致；接着是“AI”拼接两个头、乘 W_O 之后的最终输出。复读模型那一行：头 1 在第 4–6 行最看重第 1、2、3 个词，头 2 都在第 4 个词（句号）。然后是关头实验：都在 100%，关掉头 1 只剩 8.1%，关掉头 2 仍是 100%。

到了大模型里

Llama-3-8B 的注意力有 32 个头，每个头 128 维，$32 \times 128 = 4096$，正好是隐藏维度。每一层、每个 token，都有 32 行各不相同的注意力权重。头的维度 $d_{\mathrm{h}} = 128$，就是上一讲缩放时除以的 $\sqrt{d_{\mathrm{k}}}$ 里的那个 d_{k}。

多头天然适合并行。每个头的计算完全独立，GPU 上可以同时算所有头。用多张 GPU 推理时，第 105、110 讲的张量并行最常见的切法，就是把头分给不同的卡：8 张卡，每张负责 4 个头，各算各的，最后在乘 W_{o} 时再把结果合起来。

KV Cache 是按头存的。上一讲算 KV Cache 大小时用的是“8 个 KV 头”，而不是 32 个。这是第 39 讲的 GQA（分组查询注意力）：32 个查询头共用 8 组 k、v，每 4 个查询头共享一组。查询头还是 32 个，各看各的，但要缓存的 k、v 少到四分之一，KV Cache 也就小到四分之一。这一点对推理的显存和速度影响巨大，是从多头注意力出发做的第一个重要改进。

容易混淆的地方

1. **头的维度是切出来的，不是复制出来的**。每个头只拿到 d / n_{h} 维，而不是完整的 d 维。头越多，每个头越“窄”。所以头数也不是越多越好：每个头太窄，它的 q、k 点积就分辨不出足够多的东西。
2. **W_{o} 不是可有可无的**。拼接只是把各个头的结果并排放在一起，W_{o} 才把它们真正混合起来，让后面的层能用到所有头的信息。它是注意力里的第四个权重矩阵，和 W_{q}、W_{k}、W_{v} 一样大。
3. **头的分工不是设计出来的**。我们从来没有告诉复读模型“头 1 负责复读”，这是训练自己选出来的。同样的结构换一个随机种子重新训练，可能就变成头 2 负责复读，或者两个头各分担一部分。

回顾

多头注意力把 Q、K、V 按列切成几段，每段一个头，各自独立地做打分、缩放、掩码、softmax 和加权求和，再把各头的输出拼起来乘 W_{o}。参数和计算量都和一个头时差不多，却让模型可以同时从几个角度看句子。复读模型的头 1 学会了“回头看 3 格”，关掉它复读就失败；头 2 盯着

句号，关掉它毫无影响，大模型里也常有这样冗余的头。Llama-3-8B 每层 32 个头、每头 128 维；头与头互相独立，便于在 GPU 和多卡上并行；而 GQA 让多个查询头共享 k、v，大大缩小了 KV Cache。

注意力让每个词从上下文里收集了信息。可是 Transformer 往往有几十层，层数一深，训练就会变得很困难：第 18 讲说过，梯度要一层一层地往回传，传得越远越容易衰减。怎么让一个几十层的网络依然训得动？下一讲：**残差连接**。

第29讲 残差连接

14 激活函数　18 反向传播　26 自注意力

有了自注意力，我们可以开始搭 Transformer 了。一个大模型要叠很多层：Llama-3-8B 有 32 层，更大的模型有 80 层、120 层。第 15 讲说过，层数越多，网络能表达的东西越复杂。可是层数一多，就会碰到一个第 14 讲、第 18 讲都提过的难题：**梯度要一层一层地往回传，传得越远，越容易变得极小或极大，前面的层就训不动了。**

这一讲介绍解决它的关键部件：**残差连接**（residual connection）。它的做法简单得出人意料：每一层的输出，再加上这一层的输入。正是这一个加号，让几十层、上百层的网络变得可以训练。

一层一层乘下去

第 18 讲的反向传播告诉我们，梯度每往回经过一层，都要乘上这一层的“局部导数”。一个网络有 30 层，传到第 1 层时，梯度就是 30 个数（实际上是 30 个矩阵）连乘的结果。

先用最简单的数字感受一下。如果每一层的局部导数都是 0.5，乘 30 次是 $0.5^{30} \approx 9.31 \times 10^{-10}$，几乎是 0：前面的层收到的修正信号微乎其微，这叫**梯度消失**。如果每一层都是 1.5，乘 30 次是 1.92×10^{5}，大得离谱，参数一步就被推飞，这叫**梯度爆炸**。只有每层都恰好在 1 附近，才能安全地传过 30 层，而这是很难保证的。

真实的网络里也是这样。下面的代码搭了一个 30 层、每层 16 维的网络，权重随机初始化，每层是“线性层 + 激活函数”。从最后一层传回一个长度为 1 的梯度：用 tanh 时，传到第 0 层只剩 0.0506；用 ReLU 时，干脆变成了 0，因为一路上总有神经元的输出是 0（第 14 讲的“死亡 ReLU”），梯度在那里被截断，30 层叠下来全断了。

给每一层开一条捷径

残差连接的做法是：一层不再直接输出 $F(\mathbf{x})$，而是输出

$$\mathbf{x}l+1 = \mathbf{x}l + F(\mathbf{x}l)$$

F 是这一层原本要做的计算（比如注意力，或者前馈网络），$\mathbf{x}_l$ 是这一层的输入。也就是说，输入“绕过”这一层，原封不动地直接加到输出上。这条绕过去的路叫**捷径**（shortcut）或跳跃连接（skip connection）。

看一下导数。对 $\mathbf{x}_l$ 求导，加号两边分别求：第一项 $\mathbf{x}_l$ 的导数是 1，第二项是 F 的导数 F'：

$$\frac{\partial \mathbf{x}_{l+1}}{\partial \mathbf{x}_l} = 1 + F'$$

这个“1”就是关键。不管 F' 有多小，哪怕是 0，这一层的局部导数也还有 1。梯度往回传时，至少能沿着捷径原样通过，不会在某一层被截断或者压缩到 0。

还可以换一个角度看。把 30 个 $(1 + F')$ 连乘展开，会得到很多项：有一项是 $1 \times 1 \times \cdots \times 1$，对应“每一层都走捷径”的那条路；有的项里含一个 F'，对应“只在某一层走 F、其余都走捷径”的路……梯度是所有这些路径之和，其中总有很短的路径，不会因为连乘而消失。

要点

残差连接：每层输出 = 输入 + 这一层的计算结果。导数里多了一个“1”，梯度可以沿着捷径直接传回前面的层，不会在层层连乘中消失。

自己看看梯度怎么传

下面的演示就是那个 30 层的网络。琥珀色是普通网络、绿色是加了残差的网络，横轴是层（左边是最前面的层），纵轴是梯度传到这一层时的长度，用的是对数坐标：每一格是上一格的 10^5 倍。虚线是 1。

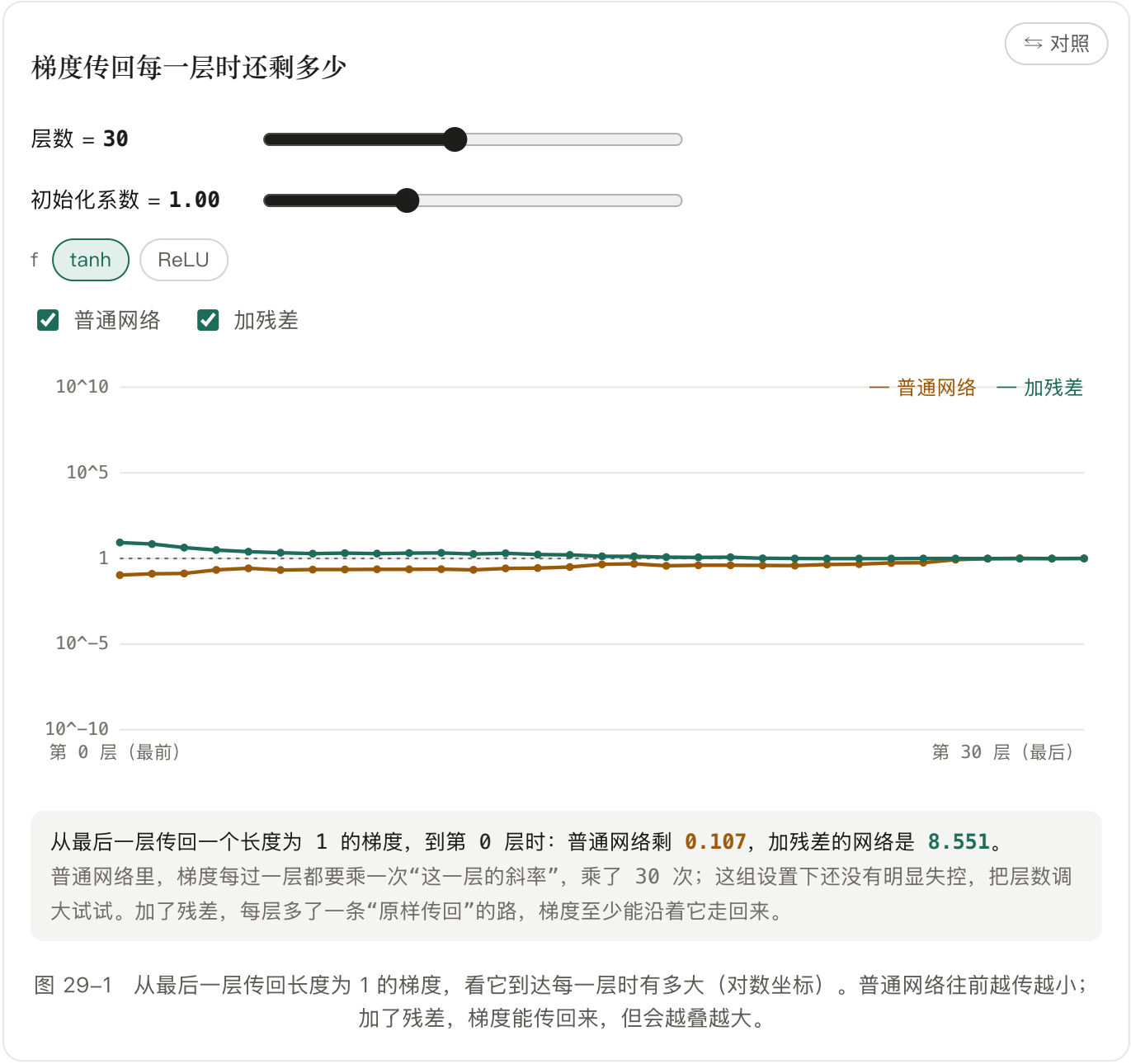


图 29–1　从最后一层传回长度为 1 的梯度，看它到达每一层时有多大（对数坐标）。普通网络往前越传越小；加了残差，梯度能传回来，但会越叠越大。

几个值得试的操作：

- **切到 ReLU**：普通网络的曲线直接掉到图的最底下，梯度在某一层被截断成了 0；加残差的网络照样能把梯度传回去。
- **拖初始化系数**：普通网络对它非常敏感。用 tanh 时把系数调到 0.6，传到第 0 层的梯度就只剩 10^{-8} 量级；残差网络在同样的设置下照样能传回来。
- **把层数拉到 60**：普通网络的问题更严重了。

但也要注意绿线的另一个特点：它不是平的，而是往前越来越大。代码里，用 tanh 时传到第 0 层是 19.57，用 ReLU 时是 3704.18。原因是残差网络的每一层都往信号上“加”东西，前向时信号越叠越大，反向时梯度也跟着越叠越大。所以光有残差还不够，还需要有东西把每一层的数值拉回一

个稳定的范围。这就是下一讲的 LayerNorm。GPT-2 等模型还会在初始化时把每个残差分支的权重缩小一些，也是为了控制这种累积。

残差流：一条贯穿全网的主干

在 Transformer 里，每一层有两个子模块：注意力和前馈网络（第 31 讲），每个都带一个残差连接：

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + \text{注意力}(\boldsymbol{x}), \qquad \boldsymbol{x} \leftarrow \boldsymbol{x} + \text{FFN}(\boldsymbol{x})$$

可以把那个一路传下来的 x 想象成一条贯穿整个网络的“主干道”，叫**残差流**（residual stream）。每个子模块都做两件事：从主干道上读出当前的信息，算出一点“修改量”，再加回主干道上。词嵌入是主干道上最初的内容，每一层都往上面叠加一些新的东西：这个词和上下文的关系、它在句子里的语法角色、可能的下一个词……最后由输出层从主干道上读出结果。

这个视角说明了一个硬性要求：**子模块的输出必须和输入同样形状**，否则加不起来。所以注意力最后要乘 W_{O} 回到 d 维（第 28 讲），前馈网络升维之后还要降回 d 维（第 31 讲）。TinyLM 第 1 层的第一个残差，就是把“AI”加了位置编码的向量 (0.541, −0.19, 0.43, 0.4)，和它的注意力输出 (−0.928, −0.698, −0.132, 0.404) 相加，得到 (−0.387, −0.888, 0.298, 0.804)。

这个视角也解释了残差为什么好训练：每一层只需要学“在已有的基础上改一点”，而不是从头变出一个全新的表示。刚初始化时，如果 F 的输出很小，整个网络就接近“什么都不做”，信号能顺畅地从头传到尾，训练从一个稳定的起点开始。

用代码复现

下面的代码先算了连乘的例子；再搭一个 30 层的网络，用 tanh 和 ReLU 分别比较普通网络和残差网络在第 0 层、第 15 层的梯度大小；最后从 TinyLM 的前向结果里取出第 1 层的第一个残差。

```python
import json
import numpy as np

# 1. 一个数一层层乘下去
print("每层乘 0.5，乘 30 层：", f"{0.5 ** 30:.2e}", "  每层乘 1.5，乘 30 层：", f"{1.5
** 30:.2e}")

# 2. 30 层、每层 16 维的网络：从最后一层传回长度为 1 的梯度，看到第 0 层还剩多少
def grad_norms(L, act, residual, gain=1.0, d=16, seed=0):
    rng = np.random.default_rng(seed)
    Ws = rng.standard_normal((L, d, d)) * gain / np.sqrt(d)
    x, zs = rng.standard_normal(d), []
    for W in Ws:                                    # 前向
        z = x @ W
        zs.append(z)
        a = np.tanh(z) if act == "tanh" else np.maximum(0, z)
        x = x + a if residual else a
    g = np.eye(d)[0]                                # 最后一层收到的梯度
    norms = [1.0]
    for W, z in zip(Ws[::-1], zs[::-1]):            # 反向（第 18 讲）
        dz = g * (1 - np.tanh(z) ** 2 if act == "tanh" else (z > 0))
        back = W @ dz
        g = g + back if residual else back          # 残差：多一条原样传回的路
        norms.append(np.linalg.norm(g))
    return norms[::-1]                              # 第 0 层在最前

for act in ["tanh", "relu"]:
    p, r = grad_norms(30, act, False), grad_norms(30, act, True)
    print(f"{act:4s}  普通：第 0 层 {p[0]:.2e}，第 15 层 {p[15]:.2e}  |  残差：第 0 层
{r[0]:.2f}，第 15 层 {r[15]:.2f}")

# 3. TinyLM 第 1 层的第一个残差：x + 注意力(x)，形状必须一样
st = {s["key"]: np.array(s["v"]) for s in json.load(open("tinylm.json",
encoding="utf-8"))["examples"]["34"]["steps"]}
print("“AI”：", st["pos"][3].round(3), "+", st["L1attn"][3].round(3), "=",
st["L1res1"][3].round(3))
print("每层的残差相加次数：2；Llama-3-8B 32 层共", 2 * 32, "次")
```

输出：

```
每层乘 0.5，乘 30 层：  9.31e-10    每层乘 1.5，乘 30 层：  1.92e+05
tanh  普通：第 0 层 5.06e-02，第 15 层 3.27e-01  |  残差：第 0 层 19.57，第 15 层 1.87
relu  普通：第 0 层 0.00e+00，第 15 层 0.00e+00  |  残差：第 0 层 3704.18，第 15 层
37.20
“AI”：  [ 0.541 -0.19   0.43   0.4  ] + [-0.928 -0.698 -0.132  0.404] = [-0.387
-0.888  0.298  0.804]
每层的残差相加次数：2；Llama-3-8B 32 层共 64 次
```

第一行是 0.5 和 1.5 连乘 30 次。接下来两行：tanh 时，普通网络的梯度传到第 0 层只剩 0.0506，残差网络是 19.57；ReLU 时，普通网络在第 0 层和第 15 层都是 0，残差网络分别是 3704.18 和 37.2。最后两行是 TinyLM 的残差相加和下一节的数字。演示和这里的随机数不同，具体数值会不一样，但趋势相同。

到了大模型里

残差连接是深度网络的标配。它最早出现在 2015 年的 ResNet（何恺明等人）里，让图像识别网络一下子从二十几层做到了一百五十多层。Transformer 原封不动地继承了它。Llama-3-8B 有 32 层，每层两次残差相加，一共 64 次。

残差流的宽度就是隐藏维度。Llama-3-8B 的残差流是 4096 维，每个 token 一路带着这 4096 个数穿过 32 层。注意力和前馈网络都从它读、往它写，所以 d 也被称为模型的“宽度”。

残差相加本身很便宜，但要搬数据。每次相加只是逐元素的加法，计算量可以忽略，但要把两个 [s, 4096] 的矩阵读进来、再写出去。和第 14 讲的激活函数一样，它是访存密集型运算，推理引擎通常把它和后面的归一化融合成一个算子（第 83 讲）。多卡用张量并行时，每个子模块的结果要先在卡之间合并（All-Reduce），再加回残差流（第 105 讲）。

容易混淆的地方

1. **残差是相加，不是拼接**。输入和输出逐元素相加，维度不变。如果把它们拼起来，每过一层维度就会翻倍。
2. **残差学的是“修改量”**。“残差”这个名字就来自这里：F 学的是“目标输出减去输入”的那部分差值，也就是残余的部分。
3. **残差不能单独解决所有问题**。它让梯度不会消失，但信号和梯度可能随着层数累积变大。实际的深层网络总是“残差 + 归一化 + 合适的初始化”一起用。

回顾

深层网络难训，是因为梯度要连乘几十个局部导数，很容易消失或爆炸：每层乘 0.5，30 层后只剩 10^{-9}。残差连接让每层输出“输入 + F(输入)”，局部导数变成 $1 + F'$ ，梯度可以沿着捷径直接传回前面的层。在 30 层的网络上，用 ReLU 时普通网络的梯度传到第 0 层变成了 0，加了残差就能传回去。Transformer 的每层有两个残差：注意力和前馈网络各一个，它们一起构成了贯穿全网的残差流，所以每个子模块的输出都必须和输入同样形状。

不过实验也显示，残差网络的数值会一层层累积变大。怎么让每一层的输出都保持在一个稳定的尺度上？下一讲：**LayerNorm**。

第30讲 LayerNorm

29 残差连接

上一讲的残差连接让梯度能穿过很多层，但也留下了一个问题：每一层都往残差流里“加”东西，数值越叠越大。下面的代码里，一个 30 层的残差网络，第 1 层输出的向量长度是 3.4，第 10 层是 42.9，第 20 层是 2006.4，第 30 层到了 82441.6。数值这样失控，后面的层很难工作：激活函数会饱和，softmax 会一边倒，训练也会极不稳定。

这一讲的 **LayerNorm**（层归一化）就是来管这件事的。它在每一层的固定位置，把每个词的向量“拉回”统一的尺度：均值为 0、方差为 1。加上它之后，同一个网络每一层输出的长度都稳定在 4.0。

拉回到均值 0、方差 1

LayerNorm 对**每个词的向量**单独操作。设这个向量是 $\mathbf{x} = (x_1, \cdots, x_d)$，一共 d 个数：

1. 算这 d 个数的**均值** μ。
2. 每个数减去 μ，新的均值就是 0。
3. 算这 d 个数的**方差** σ^2（每个数减去均值后平方，再取平均），开根号得到标准差。
4. 每个数除以标准差，新的方差就是 1。

写成一个式子：

$$\mathrm{LN}(\mathbf{x}) = \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

分母里的 ε 是一个很小的数（比如 10^{-5}），防止方差恰好为 0 时除以 0。

拿 TinyLM 算一下。第 1 层里，“AI”做完残差相加后的向量是 (−0.3872, −0.8882, 0.2983, 0.8035)。

- 均值：(−0.3872 − 0.8882 + 0.2983 + 0.8035) / 4 = −0.0434。
- 减去均值：(−0.3438, −0.8448, 0.3417, 0.8469)。
- 方差：这 4 个数平方的平均，0.4165；标准差 $\sqrt{(0.4165 + 0.00001)} \approx 0.6454$。

- 除以标准差：(−0.5327, −1.3091, 0.5295, 1.3123)。

新向量的均值是 0、方差是 1。原来的形状（哪个数大、哪个数小、大多少）都保留了下来，只是整体的位置和尺度被统一了。因为方差是 1，d 个数平方的和就是 d，所以 LayerNorm 之后的向量长度总是 $\sqrt{d}$ 左右：TinyLM 里是 $\sqrt{4}=2$，上面那个 16 维的网络里是 $\sqrt{16}=4$。不管前面的层把数值推到多大，这一步之后都回到同一个尺度。

下面的演示从 TinyLM 第 1 层的注意力输出开始，经过残差相加，到 LayerNorm。点 LayerNorm 那一站，再点表格里的某个词，下方会逐步写出均值、方差和归一化的结果。

残差相加，再做 LayerNorm

⇆ 对照

1. 第 1 层：注意力输出
2. 第 1 层：残差相加 x + 注意力
3. **第 1 层：LayerNorm**

第 1 层：LayerNorm 形状 [4, 4]

	1	2	3	4
我	-0.45	-0.76	-0.51	1.72
爱	-0.5	-0.75	-0.47	1.72
学	-0.17	-1.42	0.19	1.39
AI	**-0.53**	**-1.31**	**0.53**	**1.31**

LayerNorm（第 30 讲），对“AI”这一行：
① 均值 μ = (-0.387 + -0.888 + 0.298 + 0.804) / 4 = -0.0434
② 减去均值：(-0.344, -0.845, 0.342, 0.847)
③ 方差 = 这 4 个数平方的平均 = 0.4165，标准差 √(方差 + 0.00001) = 0.6454
④ 除以标准差：(-0.533, -1.309, 0.53, 1.312) → 均值 0、方差 1

← 上一站　下一站 →

图 30-1　TinyLM 第 1 层：注意力输出 → 残差相加 → LayerNorm。在 LayerNorm 这一站点不同的词，看每一行各自减去自己的均值、除以自己的标准差。

缩放和平移：γ 和 β

强制每个向量都是均值 0、方差 1，有时候会限制网络：也许某一维本来就该整体偏大一点。所以 LayerNorm 最后还有一步：乘一个向量 γ、加一个向量 β：

$$\mathrm{LN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

$\odot$ 表示逐元素相乘。γ 和 β 都是 d 维的、可以训练的参数，初始值分别是全 1 和全 0，也就是一开始什么都不改。训练中，网络可以通过它们给每一维单独调整尺度和偏移。TinyLM 没有训练，所以直接用 $\gamma = 1$、$\beta = 0$。

一个 LayerNorm 只有 2d 个参数，和动辄 d × d 的线性层比起来微不足道。

> **要点**
> LayerNorm 对每个词的向量单独做：减去自己的均值、除以自己的标准差，再乘 γ、加 β。之后每个向量都回到统一的尺度，不管前面的层把数值推到多大。

为什么是“每个词”

注意，均值和方差是在**一个词的 d 个数**之间算的，每个词各算各的，叫“层”归一化，因为它对一层的所有维度做归一化。

还有一种更早的 BatchNorm（批归一化），方向正好相反：对**同一维**、在一个 batch 的**所有样本**之间算均值和方差。它在图像网络里很成功，但不适合语言模型：句子有长有短，batch 里的样本彼此差异很大；更麻烦的是，推理时一次可能只生成一个 token，没有“一批”可以算统计量。LayerNorm 只看当前这个词自己，和 batch 里有什么、有多少都无关，训练和推理的计算完全一样。

LayerNorm 还有一个有用的性质：**对输入的整体缩放和平移不敏感**。把 x 整体乘以 3，均值和标准差也都乘以 3，相除之后结果不变；整体加 5，均值也加 5，相减之后结果不变。所以前面的层输出的尺度怎么漂移，经过 LayerNorm 后都被抹平了。

放在哪里

在 TinyLM 里，LayerNorm 放在每个残差相加**之后**：

$$\boldsymbol{x} \leftarrow \mathrm{LN}(\boldsymbol{x} + \text{注意力}(\boldsymbol{x})), \qquad \boldsymbol{x} \leftarrow \mathrm{LN}(\boldsymbol{x} + \mathrm{FFN}(\boldsymbol{x}))$$

这是 2017 年原始 Transformer 的做法，叫 **Post-Norm**（后归一化）。今天的大模型普遍换成了另一种位置：把 LayerNorm 放在子模块**之前**，残差流本身不归一化，叫 Pre-Norm；而且常用一个更简单的版本 RMSNorm，省掉了减均值那一步。为什么要这样改，第 36 讲会专门讨论。这一篇先按原始的结构，把整个 Transformer 搭完。

用代码复现

下面的代码手写了 LayerNorm，在 TinyLM“AI”那一行上逐步计算并和 `tinylm.json` 核对；验证它对整体缩放和平移不敏感；然后在 30 层的残差网络上比较归一化前后每层输出的长度。

```
import json
import numpy as np

st = {s["key"]: np.array(s["v"]) for s in json.load(open("tinylm.json",
encoding="utf-8"))["examples"]["34"]["steps"]}

def layer_norm(x, gamma=1.0, beta=0.0, eps=1e-5):
    mu = x.mean(axis=-1, keepdims=True)                    # ① 均值（每一行，也就是
每个词各算各的）
    var = ((x - mu) ** 2).mean(axis=-1, keepdims=True)     # ② 方差
    return gamma * (x - mu) / np.sqrt(var + eps) + beta    # ③ 减均值、除以标准差
④ 缩放、平移

# 1. TinyLM 第 1 层："AI"残差相加之后的那一行
x = st["L1res1"][3]
mu, var = x.mean(), x.var()
print("x =", x.round(4), " 均值", round(mu, 4), " 方差", round(var, 4))
y = layer_norm(x)
print("LayerNorm: ", y.round(4), " 新的均值", round(abs(y.mean()), 6), " 新的方差",
round(y.var(), 4))
print("和 tinylm.json 一致: ", np.allclose(y, st["L1ln1"][3], atol=1e-4))

# 2. 尺度不变：整行乘 3、或者加 5，结果不变
print("乘 3: ", np.allclose(layer_norm(3 * x), y, atol=1e-5), "  加 5: ",
np.allclose(layer_norm(x + 5), y))

# 3. 30 层残差网络（ReLU），每层输出的长度：不归一化时一路变大，归一化后稳定
rng = np.random.default_rng(0)
d, L = 16, 30
Ws = rng.standard_normal((L, d, d)) / np.sqrt(d)
for use_ln in [False, True]:
    h, sizes = rng.standard_normal(d), []
    for W in Ws:
        h = h + np.maximum(0, h @ W)                       # 残差
        if use_ln:
            h = layer_norm(h)
        sizes.append(np.linalg.norm(h))
    print("归一化" + ("后" if use_ln else "前") + ": 第 1、10、20、30 层输出的长度",
[round(float(sizes[i]), 1) for i in [0, 9, 19, 29]])

# 4. 参数：γ 和 β 各 d 个
print("Llama-3-8B 一个归一化层的参数: ", 4096, "个（RMSNorm 只有 γ）；每层 2 个，32 层加最
后 1 个，共", (2 * 32 + 1) * 4096, "个")
```

输出：

```
x = [-0.3872 -0.8882  0.2983  0.8035]  均值 -0.0434  方差 0.4165
LayerNorm:  [-0.5327 -1.3091  0.5295  1.3123]  新的均值 0.0  新的方差 1.0
和 tinylm.json 一致:  True
乘 3:  True   加 5:  True
归一化前：第 1、10、20、30 层输出的长度 [3.4, 42.9, 2006.4, 82441.6]
归一化后：第 1、10、20、30 层输出的长度 [4.0, 4.0, 4.0, 4.0]
Llama-3-8B 一个归一化层的参数:  4096 个（RMSNorm 只有 γ）；每层 2 个，32 层加最后 1 个，共 266240 个
```

前三行：“AI”那一行的均值 −0.0434、方差 0.4165，归一化后均值 0、方差 1，和 TinyLM 前向的结果一致。第四行：整体乘 3 或加 5，结果都不变。接下来两行：不归一化时，第 1、10、20、30 层输出的长度是 3.4、42.9、2006.4、82441.6；每层后加 LayerNorm，全都是 4.0。最后一行是下一节的数字。

到了大模型里

参数很少，但每层都有。Llama-3-8B 用的是 RMSNorm（第 36 讲），每个只有 γ 一个 4096 维的向量。每层有 2 个，32 层再加上输出层前面的最后一个，一共 65 个，合计 266240 个参数，在 80 亿里可以忽略。

计算量很小，但不能省，也不能乱。每个归一化都要先把一行的 4096 个数读一遍算统计量，再读一遍做归一化。它是访存密集型运算，通常和前面的残差相加、后面的矩阵乘法融合在一起做（第 83 讲）。它对数值精度也很敏感：求平方和时，几千个数加在一起很容易在低精度下溢出或丢失精度，所以即便模型用 BF16 计算，归一化内部通常也会用 FP32 累加（第 10 讲）。

容易混淆的地方

1. **LayerNorm 不改变形状，也不混合不同的词。**它是一行一行独立做的，“我”的归一化完全不受“爱”的影响。混合词与词的信息，只在注意力里发生。
2. **归一化之后不是所有数都在 −1 到 1 之间。**均值 0、方差 1 只是整体的统计量，单个数可以更大，比如上面的 1.3123。
3. **γ 和 β 是可训练的，和权重一起存在模型文件里。**在模型的权重列表里看到名字带 `norm` 或 `ln` 的小向量，就是它们。

回顾

残差网络的数值会一层层累积变大，30 层后可以到 8 万多。LayerNorm 对每个词的向量单独做归一化：减去这 d 个数的均值、除以标准差，再乘可训练的 γ、加 β，于是不管前面推到多大，每一层的输出都回到均值 0、方差 1、长度约 $\sqrt{d}$ 的统一尺度。它只看当前这个词，所以和 batch 无关，训练和推理一样算；它对整体缩放和平移不敏感。原始 Transformer 把它放在每个残差相加之后（Post-Norm），现代模型改放在子模块之前并换成 RMSNorm，这是第 36 讲的内容。

到这里，Transformer 一层里的“注意力 → 残差 → 归一化”都讲完了。注意力让词与词交换了信息，之后每个词还要各自做一次计算：先升维、再降维，这就是前馈网络。它在做什么？为什么要先变宽再变窄？下一讲：**FFN 前馈网络**。

第31讲 前馈网络

13 神经元与线性层　14 激活函数　15 多层感知机　29 残差连接

Transformer 的每一层有两个子模块。前面几讲讲的注意力是第一个：它让词与词之间交换信息，每个词从上下文里收集它需要的东西。这一讲讲第二个：**前馈网络**（feed-forward network，FFN），也常叫 MLP。

FFN 做的事情和注意力正好互补：**每个词各自独立地做一次计算，词与词之间完全不交流**。它的结构就是第 15 讲的多层感知机：先用一个线性层把向量变宽，过一个激活函数，再用一个线性层变回原来的宽度。这一讲把这三步在 TinyLM 上算一遍，再说说这个“先变宽、再变窄”的结构到底在做什么，以及为什么它是大模型参数和计算的大头。

升维、激活、降维

FFN 对每个词的向量 $\mathbf{x}$（d 维）做三步：

$$\mathbf{FFN}(\mathbf{x}) = \mathrm{ReLU}(\mathbf{x} \cdot W_1) \cdot W_2$$

1. **升维**：乘 W_1（$\mathrm{d} \times \mathrm{d_{ff}}$），从 d 维变成 $\mathrm{d_{ff}}$ 维。$\mathrm{d_{ff}}$ 通常是 d 的 4 倍左右。
2. **激活**：逐元素过 ReLU（第 14 讲），负数变 0。
3. **降维**：乘 W_2（$\mathrm{d_{ff}} \times \mathrm{d}$），变回 d 维。

降回 d 维是必须的：FFN 的输出要和输入相加（残差连接，第 29 讲），形状必须一样。原始 Transformer 的线性层还带偏置，TinyLM 和 Llama 都没有，这里也省掉。

TinyLM 的 $\mathrm{d} = 4$，$\mathrm{d_{ff}} = 8$。拿第 1 层里“AI”这个词算一下，它的输入是上一讲 LayerNorm 的输出 (−0.533, −1.309, 0.53, 1.312)：

- **升维**：乘 4×8 的 W_1，得到 8 个数 (1.598, −0.232, 0.083, −0.393, 1.342, −1.602, −0.586, 1.528)。
- **ReLU**：4 个负数变成 0，剩下 (1.598, 0, 0.083, 0, 1.342, 0, 0, 1.528)。8 个神经元里“亮了”4 个。
- **降维**：乘 8×4 的 W_2，回到 4 维 (2.828, 1.673, 2.837, −0.636)。

这个结果随后加回残差流，再做一次 LayerNorm，就是第 1 层的输出。

下面的演示从 LayerNorm 的输出开始，经过升维、ReLU、降维，到残差相加。注意表格的列数：4 → 8 → 8 → 4 → 4。点不同的词，看它们各自有几个神经元被激活。

前馈网络：升维 → ReLU → 降维

⇆ 对照

1. 第 1 层：LayerNorm
2. **第 1 层：FFN 升维 × W_1**
3. 第 1 层：ReLU
4. 第 1 层：FFN 降维 × W_2
5. 第 1 层：残差相加

第 1 层：FFN 升维 × W_1 形状 [4, 8]

	1	2	3	4	5	6	7	8
我	1.76	0.2	0.4	-0.42	0.8	-1.74	-0.5	1.3
爱	1.75	0.22	0.42	-0.42	0.83	-1.75	-0.55	1.34
学	1.78	-0.38	-0.08	-0.42	1.13	-1.59	-0.18	1.27
AI	**1.6**	**-0.23**	**0.08**	**-0.39**	**1.34**	**-1.6**	**-0.59**	**1.53**

升维（第 31 讲）：“AI”的 4 维向量乘 W_1（4 × 8），得到 8 个数（1.6, -0.23, 0.08, -0.39, 1.34, -1.6, -0.59, 1.53）。形状 [4, 4] → [4, 8]。

← 上一站　下一站 →

图 31-1　TinyLM 第 1 层的 FFN：4 维 → 8 维 → ReLU → 4 维，再加回残差流。每个词各算各的，表格里每一行只和自己那一行有关。

每个词各算各的

FFN 里没有任何一步会把不同的词混在一起：升维、ReLU、降维，都是对每一行单独做的。所以整句话一起算（4 × 4 的矩阵乘 W_1）和一个词一个词地算，结果完全一样，下面的代码会验证。这种“对序列里每个位置独立地做同样的事”的网络，原论文里叫 position-wise feed-forward network。

这和注意力形成了分工：

	注意力	FFN
词与词之间	交换信息	不交流
做什么	决定从上下文的哪些词收集什么	对每个词收集到的信息各自做加工
依赖序列长度	打分矩阵 $s \times s$	不依赖，每个词一样的计算

一个常见的比喻是：注意力是“开会”，大家互相交换消息；FFN 是“各自回去干活”，每个人消化自己拿到的消息。每一层都是先开会、再干活，重复几十次。

先变宽，再变窄，在做什么

为什么要先升到 8 维、再降回 4 维，而不是直接用一个 4×4 的矩阵？第 14 讲已经回答了一半：两个线性层之间如果没有激活函数，会合并成一个线性层；有了 ReLU，网络才能表达非线性的东西。而中间层越宽，能拼出来的“折线”越多（第 15 讲），能表达的函数越丰富。

还有一个更具体的看法，把 FFN 看成一张“**键—值**”表。把降维那一步拆开看：

$$\mathrm{FFN}(\boldsymbol{x}) = \Sigma_j \, \mathrm{ReLU}(\boldsymbol{x} \cdot W_1 \text{ 的第 j 列}) \times W_2 \text{ 的第 j 行}$$

中间层的第 j 个神经元做两件事。W_1 的第 j 列像一个“键”：$\mathbf{x}$ 和它的点积越大，说明 $\mathbf{x}$ 越符合这个神经元要找的模式，它就亮得越厉害（负数就完全不亮）。W_2 的第 j 行像一个“值”：这个神经元亮了以后，就把这一行按自己的亮度加到输出上。所以 FFN 的输出，就是所有被激活的神经元各自贡献的“值”按亮度相加。

“AI”那一行，亮的是第 1、3、5、8 个神经元，亮度分别是 1.598、0.083、1.342、1.528；把 W_2 的这 4 行按这些亮度加起来，正好就是输出 $(2.828, 1.673, 2.837, -0.636)$。

在训练好的大模型里，研究者发现很多这样的神经元确实对应着可以解释的模式：有的在看到某类词时亮起，有的在特定的语境下亮起，它们写回残差流的“值”，往往会提高某些下一个词的概率。所以也有人说，**模型记住的大量知识，就存在 FFN 的权重里**：注意力负责把相关的信息收集到一起，FFN 负责根据这些信息“回忆”出相应的知识。

要点

FFN = 升维 → 激活 → 降维，每个词独立计算，词与词不交流。可以把它看成一张键—值表：W_1 的每一列是一个“模式”，W_2 的对应一行是匹配上之后要写回的内容。

用代码复现

下面的代码对“AI”一个词手算 FFN 的三步，并和 TinyLM 的前向结果核对；然后用“键—值”的看法重算一遍输出；验证整句一起算和逐个词算相同；最后比较参数量和计算量。

```
import json
import numpy as np

t = json.load(open("tinylm.json", encoding="utf-8"))
st = {s["key"]: np.array(s["v"]) for s in t["examples"]["34"]["steps"]}
L0 = t["weights"]["layers"][0]
W1, W2 = np.array(L0["W_1"]), np.array(L0["W_2"])          # 4×8、8×4

# 1. “AI”这一个词：升维 → ReLU → 降维
x = st["L1ln1"][3]
up = x @ W1
act = np.maximum(0, up)
down = act @ W2
print("输入 ", x.round(3), x.shape)
print("升维 ", up.round(3), up.shape)
print("ReLU ", act.round(3), " 亮了", int((act > 0).sum()), "个")
print("降维 ", down.round(3), down.shape)
print("和 tinylm.json 一致：", np.allclose(down, st["L1down"][3], atol=1e-3))

# 2. 降维 = 被激活的神经元，各自把 W_2 的一行按自己的大小加起来
contrib = [(j + 1, round(float(act[j]), 3)) for j in range(8) if act[j] > 0]
print("亮了的神经元（编号，输出）：", contrib)
print("加权求和 W_2 的这几行：", sum(act[j] * W2[j] for j in range(8)).round(3))

# 3. 每个词各算各的：整句一起算和一个词一个词算，结果一样
X = st["L1ln1"]
together = np.maximum(0, X @ W1) @ W2
one_by_one = np.array([np.maximum(0, X[i] @ W1) @ W2 for i in range(4)])
print("整句一起算 = 逐个词算：", np.allclose(together, one_by_one))

# 4. 参数和计算量
d, f = 4, 8
print("TinyLM 每层 FFN 参数：", 2 * d * f, "个；注意力 4 个矩阵：", 4 * d * d, "个")
d, f = 4096, 14336
print(f"Llama-3-8B 每层 FFN（3 个矩阵）：{3 * d * f / 1e6:.1f}M 个参数，每个 token {2 *
3 * d * f / 1e6:.0f}M 次浮点运算")
```

输出：

```
输入  [-0.533 -1.309  0.53   1.312] (4,)
升维  [ 1.598 -0.232  0.083 -0.393  1.342 -1.602 -0.586  1.528] (8,)
ReLU  [1.598 0.    0.083 0.    1.342 0.    0.    1.528]  亮了 4 个
降维  [ 2.828  1.673  2.837 -0.636] (4,)
和 tinylm.json 一致:  True
亮了的神经元（编号，输出）:  [(1, 1.598), (3, 0.083), (5, 1.342), (8, 1.528)]
加权求和 W_2 的这几行:  [ 2.828  1.673  2.837 -0.636]
整句一起算 = 逐个词算:  True
TinyLM 每层 FFN 参数:  64 个；注意力 4 个矩阵:  64 个
Llama-3-8B 每层 FFN（3 个矩阵）：176.2M 个参数，每个 token 352M 次浮点运算
```

前几行是“AI”的三步：4 维 → 8 维 → 亮了 4 个 → 4 维，和 TinyLM 的前向结果一致。接着两行是“键—值”的看法：亮了第 1、3、5、8 个神经元，按亮度加起来的 W_2 的这几行，和降维的结果一模一样。然后是“整句一起算 = 逐个词算”。最后两行是参数和计算量，下一节用。

到了大模型里

FFN 是参数的大头。TinyLM 每层 FFN 有 $2 \times 4 \times 8 = 64$ 个参数，和注意力的 4 个 4×4 矩阵（64 个）一样多。到了 Llama-3-8B，$d = 4096$，$d_{ff} = 14336$（3.5 倍），而且用的是带门控的 SwiGLU（第 38 讲），有三个矩阵：每层 FFN 约 1.762 亿个参数，占这一层参数的约 81%（第 15 讲算过）。

FFN 也是计算的大头。每个 token 经过一层 FFN，要做约 3.52 亿次浮点运算（每个参数一次乘加，即 2 次运算）。在序列不太长时，FFN 的计算量比注意力大得多；只有序列很长时，注意力的 s^2 项才会追上来（第 26 讲）。所以 FFN 里那几个大矩阵乘法，是推理时 GPU 最主要的工作，也是量化（第 12 篇）最想压缩的对象。

FFN 好拆分。因为每个词独立计算，FFN 在多卡上很容易并行：把 W_1 按列切、W_2 按行切，每张卡负责一部分神经元，最后把各卡的结果加起来（第 105 讲的张量并行）。第 41 讲的 MoE 走得更远：把一个大 FFN 拆成很多个小 FFN（专家），每个 token 只用其中几个，参数可以很多，每个 token 的计算量却不变。

容易混淆的地方

1. **“前馈”和“反向”不是一对概念。**这里的“前馈”只是说数据从输入一路流到输出、没有回路（第 15 讲），和反向传播无关。注意力其实也是前馈的，只是习惯上把这个“升维—降维”的部分专门叫做 FFN 或 MLP。

2. d_{ff} **不一定是 4 倍**。原始 Transformer 是 4 倍；用 SwiGLU 的模型因为多了一个矩阵，为了保持参数量相近，常用约 8/3 倍再取整，Llama-3-8B 是 14336 / 4096 = 3.5 倍。
3. **FFN 里的神经元不一定都有清晰的含义**。“键—值”是一个有用的理解方式，但很多神经元同时参与好几种模式，单独看一个神经元往往解释不清。

回顾

FFN 是 Transformer 每层的第二个子模块：升维（$d \to d_{ff}$）、ReLU、降维（$d_{ff} \to d$），输出加回残差流。它对每个词独立计算，词与词之间不交流，和负责交换信息的注意力形成分工。可以把它看成一张键—值表：W_1 的每一列是一个要匹配的模式，W_2 的对应一行是匹配上以后写回的内容，在大模型里，大量知识就存在这些权重中。FFN 占了每层参数和计算的大部分，Llama-3-8B 每层约 1.76 亿个参数、每个 token 约 3.5 亿次运算。

到这里，Transformer 一层里的所有零件都讲完了：注意力、残差、归一化、FFN。它们按什么顺序、怎样连起来，组成完整的一层？一层又怎样叠成一个模型？下一讲：**Transformer Block**。

第32讲 Transformer Block

26 自注意力　29 残差连接　30 LayerNorm　31 前馈网络

第 25 到 31 讲，我们一个一个地造好了 Transformer 的零件：位置编码、自注意力、因果掩码、多头、残差连接、LayerNorm、前馈网络。这一讲把它们组装起来：先拼成一层，叫一个 **Transformer Block**（Transformer 块）；再把很多层叠起来，加上入口和出口，就是一个完整的大语言模型。

这一讲没有新的计算，重点是看清结构：数据按什么顺序经过哪些部件，形状怎么变，参数都在哪里。

一层由两个子层组成

一个 Transformer Block 由两个子层组成，每个子层的模式都一样：“做一件事，加回残差流，归一化”。

$$\text{子层 1：}\ \boldsymbol{x} \leftarrow \mathrm{LN}(\boldsymbol{x} + \text{多头注意力}(\boldsymbol{x}))$$
$$\text{子层 2：}\ \boldsymbol{x} \leftarrow \mathrm{LN}(\boldsymbol{x} + \mathrm{FFN}(\boldsymbol{x}))$$

第一个子层让词与词交换信息（第 26–28 讲），第二个子层让每个词各自加工收集到的信息（第 31 讲）。两个子层都用残差连接（第 29 讲）保证梯度能传回去，都用 LayerNorm（第 30 讲）把数值拉回统一的尺度。这是 2017 年原始 Transformer 的排法，归一化在残差之后，叫 Post-Norm，TinyLM 用的就是它。

一层的输入是 [s, d]，输出也是 [s, d]：注意力和 FFN 都把形状还原了回来。正因为形状不变，一层的输出可以直接作为下一层的输入，**同样结构的层可以一层接一层地叠下去**，叠多少层都行。每一层有自己独立的一套权重，结构相同，参数不共享。

入口和出口

把 N 层叠起来之后，前后各加一段：

- **入口**：token 编号查词嵌入表（第 22 讲），加上位置编码（第 25 讲），得到 [s, d] 的初始残差流。

- **出口**：最后一层的输出乘输出层 lm_head，变成 [s, V] 的 logits，再过 softmax，得到每个位置下一个词的概率（第 33 讲、第 23 讲）。

完整的数据流是：

> 编号 → 词嵌入 + 位置编码 → [注意力 → 残差 → LN → FFN → 残差 → LN] × N → lm_head → softmax → 下一个词的概率

下面的结构图可以点任意一个部件，看它做什么、形状怎么变、有多少参数，以及讲它的是哪一讲。上面的按钮可以切换到 Llama-3-8B，看现代大模型的结构有哪些不同。

Transformer 的结构：点一个部件看看

⇆ 对照

TinyLM（原始结构，Post-Norm）　Llama-3-8B（现代结构，Pre-Norm）

输入的 token

`[4]: 我 爱 学 AI`

↓

词嵌入

`[4] → [4, 4]`

↓

位置编码

`+ [4, 4]`

↓

一层 Transformer Block　　× 2 层

多头自注意力

`[4, 4] → [4, 4]，2 头 × 2 维`

↓

残差相加

`[4, 4] + [4, 4]`

↓

LayerNorm

`[4, 4] → [4, 4]`

↓

前馈网络 FFN

`[4, 4] → [4, 8] → [4, 4]`

↓

残差相加

`[4, 4] + [4, 4]`

↓

LayerNorm

`[4, 4] → [4, 4]`

↓

输出层 lm_head

`[4, 4] → [4, 6]`

↓

softmax → 下一个词

`[4, 6]，每行和为 1`

部件	**多头自注意力**（每层一个，共 2 个）
做什么	W_q、W_k、W_v、W_o 各 4 × 4，带因果掩码
形状	[4, 4] → [4, 4]，2 头 × 2 维
参数	64（每层）× 2 = 128
详见	第 26 讲、第 27 讲、第 28 讲、第 39 讲

每层 128 个参数 × 2 层 + 嵌入表和输出层 = **304** 个参数

图 32-1　Transformer 的结构。绿框里是一层，叠 N 次。切换到 Llama-3-8B：归一化挪到了每个子层之前，位置编码不再加到输入上，最后多了一个归一化。

数一数参数

TinyLM 小到可以把参数一个个数清楚：

部件	形状	参数
词嵌入	6×4	24
每层注意力	W_q、W_k、W_v、W_o 各 4×4	64
每层 FFN	W_1 4×8，W_2 8×4	64
2 层合计		256
输出层	4×6	24
合计		304

位置编码是算出来的，LayerNorm 的 γ、β 在 TinyLM 里固定为 1 和 0，都不算参数。

用同样的方法数 Llama-3-8B，得到 8,030,261,248 个，正是官方公布的“8B”。其中 32 层一共 6,979,584,000 个，嵌入表和输出层各 525,336,576 个。

要点

一个 Transformer Block = 注意力子层 + FFN 子层，每个子层都带残差和归一化。输入输出形状相同，所以同样的块可以叠很多层；整个模型 = 入口（嵌入 + 位置）+ N 个块 + 出口（lm_head + softmax）。

现代模型改了什么

切换到 Llama-3-8B 时，结构图有几处变化。它们是第 5 篇（第 36–41 讲）的主题，这里先列出来：

部件	原始 Transformer（TinyLM）	Llama-3 等现代模型	详见
归一化的位置	残差相加之后（Post-Norm）	每个子层之前（Pre-Norm），最后再补一个	第 36 讲
归一化的方式	LayerNorm	RMSNorm（不减均值）	第 36 讲
位置信息	正弦编码加到输入上	RoPE，在注意力里旋转 Q、K	第 37 讲
FFN	ReLU，两个矩阵	SwiGLU，三个矩阵	第 38 讲
注意力的 K、V	每个头一份	多个查询头共享一份（GQA）	第 39 讲

Pre-Norm 的写法是：

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + \mathrm{Attn}(\mathrm{Norm}(\boldsymbol{x})), \qquad \boldsymbol{x} \leftarrow \boldsymbol{x} + \mathrm{FFN}(\mathrm{Norm}(\boldsymbol{x}))$$

残差流本身不再被归一化，只有送进子层的那一份被归一化。这让梯度沿着残差流的“捷径”传得更顺畅，深层网络更好训。代价是残差流的数值不受约束，所以在出口处要补一个最后的归一化。改动虽多，但“两个子层、各带残差、叠 N 层”这个骨架一点没变。

用代码复现

下面的代码用前几讲的零件写出一个 `block` 函数，把 TinyLM 的两层串起来，和第 34 讲那个逐站计算的结果核对；然后用同一个函数数出 TinyLM 和 Llama-3-8B 各部分的参数。

```
import json
import numpy as np

t = json.load(open("tinylm.json", encoding="utf-8"))
st = {s["key"]: np.array(s["v"]) for s in t["examples"]["34"]["steps"]}

def layer_norm(x, eps=1e-5):
    mu = x.mean(-1, keepdims=True)
    return (x - mu) / np.sqrt(((x - mu) ** 2).mean(-1, keepdims=True) + eps)

def attention(x, L, n_heads=2):
    Q, K, V = (x @ np.array(L[k]) for k in ["W_q", "W_k", "W_v"])
    s, dh = x.shape[0], x.shape[1] // n_heads
    mask = np.triu(np.full((s, s), -np.inf), 1)
    out = []
    for h in range(n_heads):
        c = slice(h * dh, (h + 1) * dh)
        S = Q[:, c] @ K[:, c].T / np.sqrt(dh) + mask
        A = np.exp(S - S.max(1, keepdims=True)); A /= A.sum(1, keepdims=True)
        out.append(A @ V[:, c])
    return np.concatenate(out, 1) @ np.array(L["W_o"])

def ffn(x, L):
    return np.maximum(0, x @ np.array(L["W_1"])) @ np.array(L["W_2"])

def block(x, L):                                    # 一层：注意力 → 残差 → LN → FFN →
残差 → LN (Post-Norm)
    x = layer_norm(x + attention(x, L))
    return layer_norm(x + ffn(x, L))

x = st["pos"]                                       # 嵌入 + 位置编码
for i, L in enumerate(t["weights"]["layers"]):
    x = block(x, L)
    print(f"第 {i + 1} 层输出和逐站结果一致：", np.allclose(x, st[f"L{i + 1}ln2"],
atol=1e-3), " 形状", x.shape)

# 参数清单
def count(d, d_ff, V, n_layers, kv_dim, gated, norm_params, final_norm):
    attn = 2 * d * d + 2 * d * kv_dim
    mlp = (3 if gated else 2) * d * d_ff
    per_layer = attn + mlp + 2 * norm_params
    return {"嵌入": V * d, "每层": per_layer, "全部层": per_layer * n_layers,
            "最后的归一化": norm_params if final_norm else 0, "输出层": d * V}
tiny = count(4, 8, 6, 2, 4, False, 0, False)
llama = count(4096, 14336, 128256, 32, 1024, True, 4096, True)
for name, c in [("TinyLM", tiny), ("Llama-3-8B", llama)]:
```

```
    total = c["嵌入"] + c["全部层"] + c["最后的归一化"] + c["输出层"]
    print(name, {k: f"{v:,}" for k, v in c.items()}, f"合计 {total:,}")
```

输出：

```
第 1 层输出和逐站结果一致： True 形状 (4, 4)
第 2 层输出和逐站结果一致： True 形状 (4, 4)
TinyLM {'嵌入': '24', '每层': '128', '全部层': '256', '最后的归一化': '0', '输出层':
'24'} 合计 304
Llama-3-8B {'嵌入': '525,336,576', '每层': '218,112,000', '全部层': '6,979,584,000',
'最后的归一化': '4,096', '输出层': '525,336,576'} 合计 8,030,261,248
```

前两行：用 `block` 函数一层层算出的结果，和逐站计算的结果一致，每层的输出都是 4×4。后两行是参数清单：TinyLM 合计 304 个；Llama-3-8B 每层 218,112,000 个（注意力、FFN 和两个 RMSNorm），32 层加上嵌入、最后的归一化和输出层，合计 8,030,261,248 个。

到了大模型里

层数和宽度是模型规模的两个旋钮。Llama-3-8B 是 32 层、宽 4096；Llama-3-70B 是 80 层、宽 8192。参数量大致与“层数 × 宽度²”成正比，第 35 讲会把它写成公式。

同样的块，同样的代码。因为每一层的结构完全一样，推理引擎只需要为“一层”写好高度优化的算子，然后循环 N 次。模型的权重文件也是按层组织的：`layers.0.attention.wq`、`layers.0.feed_forward.w1`……一直到 `layers.31`。

按层切分到多卡。层与层之间是串行的，前一层算完才能算后一层。模型太大、一张卡放不下时，一种办法是把前一半的层放在一张卡上、后一半放在另一张卡上，像流水线一样传递，叫流水线并行（第 13 篇）；另一种是把每一层内部的矩阵切开，叫张量并行（第 105 讲）。两种切法都以“一层”为基本单位。

容易混淆的地方

1. **“一层”指的是一个 Block。**说 Llama-3-8B 有 32 层，指的是 32 个 Transformer Block。每个 Block 里面有 7 个线性层（第 13 讲）、2 个归一化，所以按线性层数会多得多。
2. **各层的结构相同，权重不同。**32 层不是把同一组权重用 32 遍，而是 32 组各自训练出来的权重。

3. **原始 Transformer 还有“编码器”**。2017 年的论文是为翻译设计的，有一个编码器（读原文，无因果掩码）和一个解码器（写译文）。GPT、Llama 这类语言模型只用了解码器那一半，所以也叫“仅解码器”（decoder-only）结构。

回顾

Transformer Block 由两个子层组成：多头注意力负责词与词交换信息，FFN 负责每个词各自加工；每个子层都加回残差流、做归一化。一层的输入输出形状相同，所以可以叠 N 层；再加上入口（词嵌入 + 位置编码）和出口（lm_head + softmax），就是完整的语言模型。TinyLM 有 304 个参数，Llama-3-8B 有 8,030,261,248 个，绝大部分在 32 个块里。现代模型在这个骨架上做了几处改动：Pre-Norm、RMSNorm、RoPE、SwiGLU、GQA，留到第 5 篇。

结构图的最后两站，lm_head 和 softmax，我们还没有细讲：最后一层那个 4 维（或 4096 维）的向量，是怎么变成词表上每个词的概率的？下一讲：**输出层**。

第33讲 输出层

06 指数、对数与 softmax　16 损失函数　22 词嵌入　32 Transformer Block

一句话经过所有的 Transformer 层之后，每个词都变成了一个 d 维的向量（TinyLM 是 4 维，Llama-3-8B 是 4096 维）。这些向量里装着模型从上下文里收集、加工过的全部信息。可我们最终要的不是向量，而是一个问题的答案：**下一个词是什么？**更准确地说，是词表里每个词作为下一个词的概率（第 23 讲）。

这一讲讲模型的最后一站：**输出层**。它只做两件事：一个线性层把 d 维向量变成 V 个打分，一个 softmax 把打分变成概率。这两件事前面都见过了，这一讲把它们放到模型的出口处，看清楚每个数是怎么来的，以及为什么这一层在大模型里出奇地“贵”。

给词表里每个词打一个分

输出层是一个 d × V 的矩阵，常叫 **lm_head**（语言模型头），记作 W_{lm}。最后一层的输出乘以它：

$$\text{logits} = \mathbf{x} \cdot W_{\text{lm}}$$

x 是 d 维，W_{lm} 是 d × V，结果是 V 个数，词表里每个词一个，叫 **logits**（第 16 讲）。整句话一起算，[s, d] 乘 [d, V]，得到 [s, V]：每个位置都有一行 V 个打分。

TinyLM 的 W_{lm} 是 4 × 6。“我 爱 学 AI”经过两层之后，最后一个词“AI”的输出向量是 (−1.142, −0.838, 1.12, 0.86)。乘上 W_{lm}，6 个词的打分是：

词	我	爱	学	AI	推理	<eos>
logit	−0.052	−1.105	−0.184	−0.174	1.184	0.178

每个打分，是 x 和 W_{lm} 对应那一列的点积。以“推理”为例，它那一列是 (−0.7, −0.5, 0.2, −0.3)：

$$(-1.142) \times (-0.7) + (-0.838) \times (-0.5) + 1.12 \times 0.2 + 0.86 \times (-0.3) = 1.184$$

所以可以这样理解输出层：W_{lm} 的每一列，是一个词的“输出向量”。最后一层的 x 和哪个词的输出向量方向越一致（点积越大），那个词的打分就越高（第 2 讲）。模型的几十层计算，最终的目标就是把 x 推到“下一个词”的输出向量的方向上去。

softmax：打分变成概率

打分可正可负、加起来也不是 1。最后一步用 softmax（第 6 讲）把它们变成概率：

$$p_i = \frac{e^{\text{logit}_i}}{\sum_j e^{\text{logit}_j}}$$

“AI”后面 6 个词的概率是：推理 0.441，`<eos>` 0.161，我 0.128，AI 0.113，学 0.112，爱 0.045。模型认为最可能的下一个词是“推理”。（TinyLM 没训练过，这个预测没有意义，只是数字碰巧这样。）

训练时，每个位置的这一行概率都要用：第 i 行用来预测第 i + 1 个词，和真实的下一个词算交叉熵（第 16、23 讲）。推理时，只有最后一行有用，它就是“整句话之后接什么”的答案；拿到它，再按解码策略选一个词（第 24 讲），接到末尾，进入下一轮。

下面的演示从最后一层的输出开始，经过 lm_head 到 softmax。点不同的词（不同的行），看每个位置预测的下一个词。

输出层：最后的向量变成词表上的概率

⇆ 对照

1. 第 2 层：LayerNorm（这一层的输出）
2. 输出层：logits = x · lm_head
3. **softmax：下一个词的概率**

softmax：下一个词的概率 形状 [4, 6]

	我	爱	学	AI	推理	<eos>
我	0.25	0.06	0.1	0.09	0.36	0.15
爱	0.22	0.05	0.1	0.09	0.38	0.15
学	0.17	0.05	0.12	0.09	0.42	0.15
AI	**0.13**	**0.04**	**0.11**	**0.11**	**0.44**	**0.16**

softmax（第 6 讲）：每一行变成加起来是 1 的概率。“AI”后面最可能是“推理”（0.441）。推理时只用最后一行：这就是模型对“我 爱 学 AI”之后下一个词的预测。 TinyLM 没训练过，所以预测没有意义。

← 上一站 下一站 →

图 33-1 输出层：最后一层的输出 [4, 4] 乘 lm_head [4, 6] 得到 logits，softmax 后每行是一个概率分布。推理时只用最后一行。

要点

输出层 = 线性层 + softmax。lm_head 的每一列是一个词的输出向量，最后一层的向量和它的点积就是这个词的打分（logit），softmax 后变成概率。

输出层和嵌入表的关系

输出层的形状是 $d \times V$，嵌入表是 $V \times d$（第 22 讲），正好互为转置。嵌入表的每一行是一个词“进入”模型时的向量，输出层的每一列是一个词“离开”模型时的向量。很自然地会想：能不能用同一组数？

这就是**权重绑定**（tied embeddings）：令 $W_{\text{lm}} = E^{\text{T}}$。好处是省掉一份参数；直觉上也说得通：一个词作为输入时的向量和作为输出时的向量，表达的都是“这个词”。GPT-2 和不少小模型就是这样做的。Llama-3-8B 没有绑定，两张表各自训练，各占 5.25 亿个参数；对它来说，多出来的 5 亿参数只占 6.5%，换来更大的表达自由。TinyLM 也没有绑定，W_{lm} 是单独随机生成的。

用代码复现

下面的代码把最后一层的输出乘 lm_head，打印“AI”那一行的 6 个打分；用点积单独验证“推理”的打分；做 softmax 并和逐站计算的结果核对；最后算一下 Llama-3-8B 输出层的大小和计算量。

```
import json
import numpy as np

t = json.load(open("tinylm.json", encoding="utf-8"))
st = {s["key"]: np.array(s["v"]) for s in t["examples"]["34"]["steps"]}
vocab = t["config"]["vocab"]
H = np.array(t["weights"]["lm_head"])                # 4×6：每一列对应词表里的一个词

# 1. 最后一层的输出（4 个词 × 4 维）乘 lm_head，得到每个位置 6 个词的打分
X = st["L2ln2"]
logits = X @ H
print("logits 的形状：", logits.shape)
x = X[3]                                             # 推理时只关心最后一个词“AI”
print("“AI”的最后输出：", x.round(3))
print("logits：", {w: round(float(v), 3) for w, v in zip(vocab, x @ H)})

# 2. 每个打分 = x 和 lm_head 那一列的点积
j = vocab.index("推理")
print("“推理”的打分 =", " + ".join(f"({a:.3f})×({b:.1f})" for a, b in zip(x, H[:,
j])), "=", round(float(x @ H[:, j]), 3))

# 3. softmax 变成概率
z = x @ H
p = np.exp(z - z.max()); p /= p.sum()
print("概率：", {w: round(float(v), 3) for w, v in zip(vocab, p)}, " 最可能：",
vocab[int(p.argmax())])
print("和逐站结果一致：", np.allclose(p, st["probs"][3], atol=1e-3))

# 4. Llama-3-8B：输出层的参数和每个 token 的计算量
d, V, N = 4096, 128256, 8.03e9
print(f"lm_head {d}×{V} = {d * V / 1e6:.0f}M 个参数；每个 token {2 * d * V /
1e9:.2f} GFLOPs，占全模型 2N 的 {d * V / N * 100:.1f}%")
print(f"一个位置的 logits 用 FP32 存：{V * 4 / 1024:.0f} KiB")
```

输出：

```
logits 的形状： (4, 6)
“AI”的最后输出： [-1.142 -0.838  1.12   0.86 ]
logits： {'我': -0.052, '爱': -1.105, '学': -0.184, 'AI': -0.174, '推理': 1.184,
'<eos>': 0.178}
“推理”的打分 = (-1.142)×(-0.7) + (-0.838)×(-0.5) + (1.120)×(0.2) + (0.860)×(-0.3) =
1.184
概率： {'我': 0.128, '爱': 0.045, '学': 0.112, 'AI': 0.113, '推理': 0.441, '<eos>':
0.161}  最可能： 推理
和逐站结果一致： True
lm_head 4096×128256 = 525M 个参数；每个 token 1.05 GFLOPs，占全模型 2N 的 6.5%
一个位置的 logits 用 FP32 存：501 KiB
```

logits 是 4 × 6，每个位置一行。“AI”那一行的 6 个打分和上面的表一致，“推理”的打分 1.184 用 4 项乘积相加得到。softmax 之后“推理”概率最大，0.441，和逐站计算的结果一致。最后两行是下一节的数字。

到了大模型里

输出层很大。Llama-3-8B 的 lm_head 是 4096 × 128256，约 5.25 亿个参数。每生成一个 token，都要把整个矩阵读一遍、做 1.05 GFLOPs 的计算，占整个模型每 token 计算量的 6.5%，比任何一个单独的线性层都大得多。词表越大，这一项越重。

只算最后一行。推理时只需要最后一个位置的分布，所以 prefill 阶段（第 68 讲）虽然整段提示都要经过所有的层，到了输出层却只对最后一个位置做矩阵乘法。如果对 8192 个位置都算，光 logits 就是 8192 × 128256 个数；推理引擎会特意跳过这些用不到的计算。训练时则每个位置都要算，第 16 讲提到，这时 logits 本身就要好几 GiB 显存。

softmax 之后还有采样。一个位置的 logits 有 128256 个，用 FP32 存约 501 KiB。生成时还要在它上面做温度缩放、Top-k、Top-p（第 73 讲）和采样，这些操作虽然计算量小，但要在每一步、每个请求上做一次，推理引擎通常把它们写成专门的 GPU 算子，避免把 logits 搬回 CPU。

容易混淆的地方

1. **logits 不是概率。**logits 可正可负，可以很大，加起来不是 1；只有过了 softmax 才是概率。很多推理接口会返回 logits 或者它的对数概率（log-softmax），使用时要分清楚。

2. **输出层前面通常还有一个归一化**。Pre-Norm 的现代模型里，残差流本身没有归一化，所以在 lm_head 之前会补一个最后的 RMSNorm（第 32 讲的结构图）。TinyLM 是 Post-Norm，每层已经以 LayerNorm 结束，不需要再加。
3. **“头”有两种意思**。lm_head 的“头”指模型最后接的那个输出部件；多头注意力的“头”指注意力被拆成的那几份。两者无关。

回顾

输出层把最后一层的 d 维向量变成词表上的概率：先乘 d × V 的 lm_head，得到每个词的打分 logits，每个打分是这个向量和该词“输出向量”的点积；再过 softmax 变成概率。TinyLM 对“我 爱 学 AI”之后的预测是“推理”（0.441）。训练时每个位置的概率都用来算损失，推理时只用最后一行。输出层的形状正好是嵌入表的转置，可以共用一份权重（权重绑定），Llama-3-8B 没有这样做。它很大：Llama-3-8B 的 lm_head 有 5.25 亿个参数，每个 token 要算 1.05 GFLOPs。

到这里，模型的每一站都讲完了。下一讲把它们全部串起来：让“我 爱 学 AI”从第一站的查表开始，经过两层 Transformer，一直走到最后的概率，每一步的数字都可以查看。下一讲：**完整前向传播**。

第34讲 前向传播

22 词嵌入　25 位置编码　32 Transformer Block　33 输出层

从第 21 讲的分词开始，我们一站一站地造好了语言模型的每个部件。上一讲讲完了最后一站。这一讲不引入任何新东西，只做一件事：**让一句话真正走完全程**。“我 爱 学 AI”这 4 个编号从入口进去，经过 TinyLM 的两层 Transformer，从出口出来变成下一个词的概率。一路上有 24 站，每一站的每个数都可以查看。

走完这一遍，你看到的就是大模型每一次推理时发生的全部事情。Llama-3-8B 和它的区别，只是更宽（4096 维）、更深（32 层）、词表更大（128256 个词），以及第 5 篇要讲的几处改进。

全程的 24 站

TinyLM 的配置：词表 6 个词，d = 4，2 层，每层 2 个注意力头（每头 2 维），FFN 中间层 8 维，结构是原始 Transformer 的 Post-Norm（第 32 讲）。输入是“我 爱 学 AI”的编号 [0, 1, 2, 3]。

站	做什么	形状	详见
1	查嵌入表	[4] → [4, 4]	第 22 讲
2	加正弦位置编码	[4, 4]	第 25 讲
3	第 1 层：算 Q、K、V	3 个 [4, 4]	第 26 讲
4	两个头的注意力权重（带因果掩码）	2 个 [4, 4]	第 27、28 讲
5	两个头拼接、乘 W_O	[4, 4]	第 28 讲
6	残差相加	[4, 4]	第 29 讲
7	LayerNorm	[4, 4]	第 30 讲
8	FFN 升维	[4, 8]	第 31 讲
9	ReLU	[4, 8]	第 14 讲
10	FFN 降维	[4, 4]	第 31 讲
11	残差相加	[4, 4]	第 29 讲
12	LayerNorm：第 1 层的输出	[4, 4]	第 30 讲
13–22	第 2 层：同样的 10 站，换一套权重	[4, 4]	第 32 讲
23	乘 lm_head：logits	[4, 6]	第 33 讲
24	softmax：下一个词的概率	[4, 6]	第 33 讲

从第 2 站到第 22 站，数据的形状始终是 [4, 4]：4 个词、每个 4 维，只有 FFN 中间短暂地变宽到 8 维。这正是残差流（第 29 讲）：一条宽度固定的主干道，每一站都从它读、往它写。

下面的演示就是这 24 站。左边点任意一站，或用“上一站”“下一站”一路走下去；点表格里的某一行，下方会写出这一站对这个词做了什么。建议选“AI”那一行（默认），一路跟着它走到最后。

TinyLM 完整前向：一站一站往下走

⇆ 对照

1. **词嵌入 X**
2. 加位置编码
3. 第 1 层：Q、K、V
4. 第 1 层：注意力权重（头 1 | 头 2）
5. 第 1 层：注意力输出
6. 第 1 层：残差相加 x + 注意力
7. 第 1 层：LayerNorm
8. 第 1 层：FFN 升维 × W_1
9. 第 1 层：ReLU
10. 第 1 层：FFN 降维 × W_2
11. 第 1 层：残差相加
12. 第 1 层：LayerNorm（这一层的输出）
13. 第 2 层：Q、K、V
14. 第 2 层：注意力权重（头 1 | 头 2）

词嵌入 X　形状 [4, 4]

	1	2	3	4
我	0.7	-0.6	-0.9	0.3
爱	-0.3	-0.1	-0.8	-0.2
学	0.3	-0.3	0.6	0.6
AI	0.4	0.8	0.4	-0.6

“AI”查嵌入表，取出它那一行（第 22 讲）：(0.4, 0.8, 0.4, -0.6)。形状 [4 个词, 4 维]。

← 上一站　下一站 →

图 34-1　TinyLM 的完整前向传播。从查表开始，经过两层 Transformer，到最后的概率。每一站都可以点开，所有数字都由 tinylm.py 算出。

跟着“AI”走一遍

挑几站看看“AI”这个词的变化。

- **第 1、2 站**：查表得到 (0.4, 0.8, 0.4, −0.6)，加上第 3 个位置的编码 (0.141, −0.99, 0.03, 1.0)，变成 (0.541, −0.19, 0.43, 0.4)。
- **第 4 站**：它是最后一个词，可以看全部 4 个词。在头 1 里它分给 4 个词的权重是 0.262、0.253、0.247、0.238，在头 2 里是 0.271、0.263、0.24、0.226，都比较平均，因为 TinyLM 没有训练过。（这里的输入加了位置编码，所以和第 28 讲只用嵌入算出的数字不同。）
- **第 6、7 站**：注意力的输出加回去，再归一化，得到 (−0.533, −1.309, 0.53, 1.312)。这就是第 30 讲手算过的那一行。
- **第 8–10 站**：FFN 里 8 个神经元亮了 4 个，降维后得到 (2.828, 1.673, 2.837, −0.636)，第 31 讲算过。
- **第 22 站**：两层结束，“AI”的最终向量是 (−1.142, −0.838, 1.12, 0.86)。
- **第 23、24 站**：乘 lm_head、softmax，“推理”的概率最高，0.441（第 33 讲）。

所以 TinyLM 对“我 爱 学 AI”之后的预测是“推理”。权重是随机的，这个预测没有意义；但整个计算过程，和一个训练好的大模型完全相同。

> **要点**
> 一次前向 = 查表 + 位置编码 → N 个 Transformer Block（每个：注意力、残差、归一化、FFN、残差、归一化）→ lm_head → softmax。大模型的每一次推理都是这一串计算。

前缀的预测，就藏在整句里

再看一个第 27 讲埋下的伏笔。把整句“我 爱 学 AI”送进去，得到 4 行概率；再只送前缀“我 爱”进去，得到 2 行。下面的代码会验证：**只送“我 爱”时的最后一行，和送整句时的第 2 行，完全相同**。

原因是因果掩码：第 2 个位置在每一层里都只能看第 1、2 个词，后面跟着“学 AI”还是什么都没有，对它毫无影响。这说明了两件事。第一，训练时一次前向的 4 行输出，确实是 4 道独立、合法的“猜下一个词”的题（第 23 讲）。第二，生成时前面那些词的计算结果永远不变，可以缓存（第 27 讲的 KV Cache）。

一个词一个词地生成

把前向传播放进第 24 讲的循环，TinyLM 就能“说话”了。从“我”开始，每次取概率最大的词接回去：

1. “我”→ 预测“推理”（0.360）
2. “我 推理”→ 预测“我”（0.582）
3. “我 推理 我”→ 预测“推理”（0.343）

生成的是“我 推理 我 推理”。没有意义，还有点重复，这正是一个没训练过的模型该有的样子。注意每一步都做了一次完整的前向。这里每一步都把整句从头算一遍；用上 KV Cache，每一步就只需要算新来的那一个词。

用代码复现

下面的代码用不到 30 行写出了 TinyLM 的完整前向：入口、两层 Block、出口，并和逐站计算的结果核对；然后验证“前缀的最后一行 = 整句的对应行”；最后从“我”开始贪心生成到 4 个词。

```
import json
import numpy as np

t = json.load(open("tinylm.json", encoding="utf-8"))
vocab, E, layers, W_lm = t["config"]["vocab"], np.array(t["weights"]["embedding"]),
t["weights"]["layers"], np.array(t["weights"]["lm_head"])

def sinusoid(n, d):                                          # 第 25 讲
    pos, freq = np.arange(n)[:, None], 1 / 10000 ** (np.arange(0, d, 2) / d)
    P = np.zeros((n, d)); P[:, 0::2], P[:, 1::2] = np.sin(pos * freq), np.cos(pos *
freq)
    return P

def layer_norm(x, eps=1e-5):                                 # 第 30 讲
    mu = x.mean(-1, keepdims=True)
    return (x - mu) / np.sqrt(((x - mu) ** 2).mean(-1, keepdims=True) + eps)

def softmax(z):
    e = np.exp(z - z.max(-1, keepdims=True)); return e / e.sum(-1, keepdims=True)

def forward(tokens, log=False):
    ids = [vocab.index(w) for w in tokens]
    x = E[ids] + sinusoid(len(ids), 4)                       # 入口：嵌入 + 位置编码
    s = len(ids); mask = np.triu(np.full((s, s), -np.inf), 1)
    for L in layers:                                         # 2 层 Transformer Block
        Q, K, V = (x @ np.array(L[k]) for k in ["W_q", "W_k", "W_v"])
        heads = []
        for h in range(2):                                   # 2 个头，每头 2 维，因果掩
码
            c = slice(2 * h, 2 * h + 2)
            heads.append(softmax(Q[:, c] @ K[:, c].T / np.sqrt(2) + mask) @ V[:,
c])
        x = layer_norm(x + np.concatenate(heads, 1) @ np.array(L["W_o"]))          #
残差 + LN
        x = layer_norm(x + np.maximum(0, x @ np.array(L["W_1"])) @
np.array(L["W_2"]))   # FFN + 残差 + LN
        if log:
            print("  一层之后：", x.shape)
    return softmax(x @ W_lm)                                 # 出口：lm_head + softmax

print("“我 爱 学 AI”：")
p = forward(["我", "爱", "学", "AI"], log=True)
print("  概率", p.shape, "最后一行：", {w: round(float(v), 3) for w, v in zip(vocab,
p[-1])})
st = {s["key"]: np.array(s["v"]) for s in t["examples"]["34"]["steps"]}
print("  和逐站结果一致：", np.allclose(p, st["probs"], atol=1e-3))
```

```
# 因果掩码：只输入前缀“我 爱”，最后一行和整句的第 2 行一模一样
p2 = forward(["我", "爱"])
print("前缀“我 爱”的最后一行 = 整句的第 2 行：", np.allclose(p2[-1], p[1]))

# 贪心生成：从“我”开始，每步取概率最大的词接回去（TinyLM 最长 4 个词）
seq = ["我"]
while len(seq) < 4:
    q = forward(seq)[-1]
    seq.append(vocab[int(q.argmax())])
    print(f"  第 {len(seq) - 1} 步：前 {len(seq) - 1} 个词 → “{seq[-1]}”
 ({q.max():.3f}) ")
print("生成：", " ".join(seq))
```

输出：

```
“我 爱 学 AI”：
  一层之后：  (4, 4)
  一层之后：  (4, 4)
  概率 (4, 6) 最后一行：  {'我': 0.128, '爱': 0.045, '学': 0.112, 'AI': 0.113, '推理':
0.441, '<eos>': 0.161}
  和逐站结果一致：  True
前缀“我 爱”的最后一行 = 整句的第 2 行：  True
  第 1 步：前 1 个词 → “推理” (0.360)
  第 2 步：前 2 个词 → “我” (0.582)
  第 3 步：前 3 个词 → “推理” (0.343)
生成：  我 推理 我 推理
```

前几行：整句经过两层，形状一直是 (4, 4)，最后的概率是 (4, 6)，最后一行和第 33 讲一样，并且和逐站结果一致。接着是因果掩码的验证。最后是 3 步贪心生成，得到“我 推理 我 推理”。

到了大模型里

同样的计算，更大的数字。Llama-3-8B 一次前向：128256 个词的嵌入表查表；32 层，每层是 RMSNorm、32 头的注意力（带 RoPE 和 GQA）、残差、RMSNorm、SwiGLU FFN、残差；最后一个 RMSNorm、4096 × 128256 的 lm_head 和 softmax。每个 token 约要 2 × 80 亿 = 160 亿次浮点运算，下一讲会把这笔账算细。

推理引擎就是在优化这一串计算。第三章要讲的几乎所有技术，都可以放回这张 24 站的表里看：KV Cache 让第 3 站的 K、V 不必重算（第 69 讲）；FlashAttention 让第 4、5 站不必把整个注意力矩阵写进显存（第 84 讲）；算子融合把第 6、7 站这样的小操作合并起来（第 83 讲）；

量化让第 3、8、10 站这些大矩阵乘法读更少的字节（第 12 篇）；张量并行把每一站的矩阵切到多张卡上（第 105 讲）。

这一讲的 TinyLM 就是第五章的起点。第五章的迷你推理引擎，会从一个像这样的朴素前向开始，一步步加上这些优化，并测出每一步带来的加速。

容易混淆的地方

1. **前向传播和推理不完全是一回事**。一次前向是“给一串 token，算出每个位置的概率”；推理（生成）是把前向放进循环，每次选一个词接回去，一共做很多次前向（prefill 一次、decode 很多次）。
2. **训练也要做这一遍前向**。训练时先做完全相同的前向，再算损失、做反向传播（第 18 讲）。推理只有前向，没有反向。
3. **TinyLM 的预测没有意义**。它的权重是随机的，生成的句子是乱的。这一讲关心的是计算过程，不是结果。

回顾

一句话的完整前向传播：编号查嵌入表、加位置编码，进入第一层；每层先做多头注意力、残差、LayerNorm，再做 FFN、残差、LayerNorm；两层之后乘 lm_head 得到 logits，softmax 得到每个位置下一个词的概率。全程 24 站，数据始终是 [4, 4] 的残差流，只在 FFN 中间变宽、在出口处变成 [4, 6]。TinyLM 预测“我 爱 学 AI”之后是“推理”；因为因果掩码，前缀“我 爱”的预测和整句的第 2 行完全一样；放进循环，它生成了“我 推理 我 推理”。Llama-3-8B 的每一次推理，都是同样的计算，只是规模大得多。

规模大到什么程度？一个模型的参数是怎么一项项加出来的，处理一个 token 到底要多少次浮点运算，序列变长时又会怎样？下一讲把这笔账算清楚：**参数量与计算量**。

第35讲 参数量与计算量

重点

03 矩阵乘法　18 反向传播　32 Transformer Block　34 前向传播

上一讲让一句话走完了 TinyLM 的全程。这一讲换一个角度，不看数字怎么流动，而是**算账**：一个模型到底有多少个参数？处理一个 token 要做多少次浮点运算？这两个数决定了一切工程问题：模型要占多少显存，推理有多快、多贵，训练要多少张卡、多少天。

后面第三到五章的大部分内容，都建立在这一讲的几个公式之上，所以它标了重点。我们会从结构图出发，把参数一项项加出来，得到和官方完全一致的数字；再推出“推理每 token ≈ 2N、训练每 token ≈ 6N”这两个最常用的估算公式，并看看它们什么时候会失准。

参数：一项项加出来

参数全在线性层、嵌入表和归一化里。照着第 32 讲的结构图，逐项写出公式。记隐藏维度 d、层数 L、注意力头数 n_h、KV 头数 n_{kv}、FFN 中间维度 d_{ff}、词表大小 V。

每层的注意力。W_q 和 W_o 都是 d × d。W_k、W_v 的输出宽度是 $kv = (d / n_h) \times n_{kv}$：每个 KV 头和查询头一样宽，但只有 n_{kv} 个（第 39 讲的 GQA；没有 GQA 时 $n_{kv} = n_h$，$kv = d$）。所以

$$\text{注意力} = 2d^2 + 2 \cdot d \cdot kv$$

每层的 FFN。两个矩阵（ReLU 版本）是 $2 \cdot d \cdot d_{ff}$，三个矩阵（SwiGLU 版本，第 38 讲）是 $3 \cdot d \cdot d_{ff}$。

每层的归一化。两个，每个 d 个参数（RMSNorm 只有 γ）。

入口和出口。嵌入表 $V \cdot d$，输出层 $d \cdot V$（权重绑定时为 0），最后的归一化 d。

合起来：

$$N = V \cdot d + L \times (\text{注意力} + \text{FFN} + 2d) + d + V \cdot d$$

代入 Llama-3-8B（$d = 4096$，$L = 32$，$n_h = 32$，$n_{kv} = 8$，$d_{ff} = 14336$，$V = 128256$，$kv = 128 \times 8 = 1024$）：

部分	算式	参数
每层注意力	$2 \times 4096^2 + 2 \times 4096 \times 1024$	41,943,040
每层 FFN	$3 \times 4096 \times 14336$	176,160,768
每层合计（含 2 个归一化）		218,112,000
嵌入表、输出层	各 128256×4096	各 525,336,576
合计		8,030,261,248

和 Meta 公布的“8.03B”一个不差。同样的公式算 Llama-3-70B（$d = 8192$，$L = 80$，$n_h = 64$，$d_{ff} = 28672$），得到 70,553,706,496，也就是“70B”。

从占比看，Llama-3-8B 的参数里 **FFN 占 70.2%，注意力占 16.7%，嵌入表和输出层占 13.1%**，归一化可以忽略。模型越大，嵌入和输出层的占比越小，FFN 和注意力的占比越大。

计算量：每个参数一次乘加

前向处理一个 token 要多少次浮点运算？第 3 讲说过，一次 $[1, d_{in}] \times [d_{in}, d_{out}]$ 的矩阵乘法，要 $d_{in} \times d_{out}$ 次乘法和差不多同样多次加法，也就是 $2 \times d_{in} \times d_{out}$ 次浮点运算（FLOPs）。换句话说：**对一个 token，每个权重参与一次乘、一次加，贡献 2 FLOPs**。

模型里除了嵌入表（它只是查表，不做乘法），其余参数都在矩阵乘法里，所以：

$$\text{矩阵乘法部分} \approx 2 \times (N - V \cdot d)$$

Llama-3-8B 是 $2 \times$ (80.3 亿 − 5.25 亿) ≈ 150 亿 FLOPs，即 15.01 GFLOPs。

还有一部分不在参数里：注意力的打分 QK^T 和加权求和 AV（第 26 讲）。一个 token 前面有 s 个 token 时，它的 q 要和 s 个 k 做点积，再给 s 个 v 加权，每层大约 $2 \cdot s \cdot d + 2 \cdot s \cdot d = 4 \cdot s \cdot d$ 次运算：

$$\text{每 token 前向} \approx 2 \times (N - V \cdot d) + 4 \cdot L \cdot s \cdot d$$

第二项随上下文长度 s 增长。对 Llama-3-8B：

上下文长度 s	每 token 计算量	注意力打分的占比
0	15.01 GFLOPs	0%
1024	15.55 GFLOPs	3.5%
8192	19.30 GFLOPs	22.2%
131072	83.73 GFLOPs	82.1%

上下文在几千以内时，注意力打分的计算可以忽略，每 token 的计算量基本就是 2N；上下文到了十几万，它就成了主角，每个 token 的计算量翻了 5 倍多。这就是长上下文贵的原因（第 42 讲）。

要点

推理每 token ≈ 2N FLOPs（每个参数一次乘加）；上下文很长时，还要加上注意力打分的 $4 \cdot L \cdot s \cdot d$。训练每 token ≈ 6N（前向 2N，反向 4N）。

两个最常用的估算

把细节略去，得到两个几乎所有人都在用的估算：

- **推理（前向）每 token ≈ 2N FLOPs**。Llama-3-8B 约 16 GFLOPs。生成 1000 个 token，约 16 TFLOPs。
- **训练每 token ≈ 6N FLOPs**。前向 2N，反向因为每个矩阵乘法变成两个，约 4N（第 18 讲）。Llama-3-8B 约 48 GFLOPs；第 20 讲用 $6 \times N \times D$ 算出它在 15T token 上训练约需 7×10^{23} FLOPs，就是这个公式。

这两个公式忽略了嵌入表的那点差别，也忽略了注意力打分、归一化、softmax 等，所以只在上下文不太长时准确。但它们足够简单，可以心算，是估计成本时的第一步。

自己算一算

下面的计算器实现了这一讲的全部公式。选一个预设，或者直接改任何一个配置；下面调上下文长度、并发请求数和精度，看计算量和显存怎么变。

⇆ 对照

模型计算器：参数、计算量、显存

TinyLM | Llama-3-8B | Llama-3-70B

隐藏维度 d	层数 L	注意力头数	KV 头数
4096	32	32	8

FFN 中间维度	词表大小 V
14336	128256

☑ 门控 FFN（3 个矩阵）　☑ 归一化有参数

☐ 输出层和嵌入共享

上下文长度 s	并发请求数	精度
8192	16	BF16（2 字节）

部分	参数	占比	怎么算
嵌入表	525.34 M	6.5%	V × d = 128,256 × 4,096
注意力（全部层）	1.34 G	16.7%	L × (2d² + 2 · d · kv)，kv = d / 32 × 8 = 1,024
FFN（全部层）	5.64 G	70.2%	L × 3 × d × d_ff
归一化（全部）	266.2 K	0.0%	L × 2d + d
输出层	525.34 M	6.5%	d × V
合计	**8.03 G**		8,030,261,248 个

每 token 前向（上下文 8,192）
19.30 GFLOPs
矩阵乘法 2 × 7.50 G = 15.01 G，注意力打分 4 · L · s · d = 4.29 G（占 22.2%）

处理一段 8,192 个 token 的提示
140.55 TFLOPs
s × 矩阵乘法 + 因果注意力 2 · L · s² · d

训练每 token
≈ 48.18 GFLOPs
6N：前向 2N + 反向 4N（第 18 讲）

权重显存
14.96 GiB
8,030,261,248 × 2 字节

KV Cache
16.00 GiB
每 token 2 · L · kv · 2 = 128.0 KiB，× 8,192 token × 16 个请求

图 35-1　模型计算器。默认是 Llama-3-8B，上下文 8192、16 个请求、BF16。试试：把上下文调到 131072，看注意力打分的占比；把精度改成 INT4，看权重显存；把 KV 头数改成 32（不用 GQA），看 KV Cache。

几个值得做的实验：

- **切换 TinyLM**：合计 304 个参数，和第 32 讲数的一样。
- **把 KV 头数从 8 改成 32**：相当于不用 GQA。参数只多了一点（W_{k}、W_{v} 变宽），KV Cache 却变成了 4 倍。第 39 讲会专门讲这件事。
- **把精度改成 INT4**：权重从 14.96 GiB（即 16.1 GB）降到 3.74 GiB（约 4 GB）。这是量化（第 12 篇）最直接的收益。计算器里的显存用 GiB（2^{30} 字节），正文里常说的 GB 是 10^{9} 字节，两者差约 7%。
- **把并发请求数调大**：KV Cache 线性增长。默认的 16 个请求、每个 8192 token，KV Cache 是 16 GiB，已经和权重一样大了。

用代码复现

下面的代码用一个函数算出三个模型的参数（和计算器一致），打印 Llama-3-8B 的参数占比和不同上下文下的每 token 计算量，最后给出 2N、6N 的估算和显存。

```
MODELS = {
    "TinyLM":      dict(d=4,    L=2,  n_h=2,  n_kv=2, d_ff=8,     V=6,
gated=False, norm=0, tied=False),
    "Llama-3-8B":  dict(d=4096, L=32, n_h=32, n_kv=8, d_ff=14336, V=128256,
gated=True,  norm=1, tied=False),
    "Llama-3-70B": dict(d=8192, L=80, n_h=64, n_kv=8, d_ff=28672, V=128256,
gated=True,  norm=1, tied=False),
}

def params(m):
    d, L, kv = m["d"], m["L"], m["d"] // m["n_h"] * m["n_kv"]       # kv: K、V 的宽
度（GQA 时变窄）
    attn = 2 * d * d + 2 * d * kv                                    # W_q、W_o 是
d×d；W_k、W_v 是 d×kv
    ffn = (3 if m["gated"] else 2) * d * m["d_ff"]
    norms = 2 * d * m["norm"]
    emb, head = m["V"] * d, (0 if m["tied"] else m["V"] * d)
    total = emb + L * (attn + ffn + norms) + d * m["norm"] + head
    return dict(attn=attn, ffn=ffn, layer=attn + ffn + norms, emb=emb, head=head,
total=total)

def flops_per_token(m, s):
    """前向处理一个 token（它前面有 s 个 token）：所有矩阵乘法 2 × 参数（嵌入查表不算），加上
注意力的打分和加权 4·L·s·d。"""
    p = params(m)
    matmul = 2 * (p["total"] - p["emb"])
    attn_score = 4 * m["L"] * s * m["d"]
    return matmul, attn_score

for name, m in MODELS.items():
    p = params(m)
    print(f"{name}: 每层 注意力 {p['attn']:,} + FFN {p['ffn']:,}, 共 {p['layer']:,};
嵌入 {p['emb']:,}, 输出层 {p['head']:,}; 合计 {p['total']:,}")

m = MODELS["Llama-3-8B"]
p = params(m)
print(f"Llama-3-8B 参数占比: FFN {m['L'] * p['ffn'] / p['total']:.1%}, 注意力 {m['L']
* p['attn'] / p['total']:.1%}, 嵌入+输出层 {(p['emb'] + p['head']) /
p['total']:.1%}")
for s in [0, 1024, 8192, 131072]:
    a, b = flops_per_token(m, s)
    print(f"  上下文 {s:>6}: 每 token {(a + b) / 1e9:6.2f} GFLOPs, 其中注意力打分占 {b
/ (a + b):.1%}")

# 估算：生成 / 训练
N = p["total"]
```

```
print(f"简便估算：推理每 token ≈ 2N = {2 * N / 1e9:.1f} GFLOPs；训练每 token ≈ 6N = {6
* N / 1e9:.1f} GFLOPs")
kv = 2 * m["L"] * (m["d"] // m["n_h"] * m["n_kv"]) * 2
print(f"BF16 权重 {N * 2 / 1e9:.1f} GB；KV Cache 每 token {kv // 1024} KiB，8192
token × 16 个请求 = {kv * 8192 * 16 / 2**30:.0f} GiB")
```

输出：

```
TinyLM：每层 注意力 64 + FFN 64，共 128；嵌入 24，输出层 24；合计 304
Llama-3-8B：每层 注意力 41,943,040 + FFN 176,160,768，共 218,112,000；嵌入
525,336,576，输出层 525,336,576；合计 8,030,261,248
Llama-3-70B：每层 注意力 150,994,944 + FFN 704,643,072，共 855,654,400；嵌入
1,050,673,152，输出层 1,050,673,152；合计 70,553,706,496
Llama-3-8B 参数占比：FFN 70.2%，注意力 16.7%，嵌入+输出层 13.1%
  上下文       0：每 token  15.01 GFLOPs，其中注意力打分占 0.0%
  上下文    1024：每 token  15.55 GFLOPs，其中注意力打分占 3.5%
  上下文    8192：每 token  19.30 GFLOPs，其中注意力打分占 22.2%
  上下文 131072：每 token  83.73 GFLOPs，其中注意力打分占 82.1%
简便估算：推理每 token ≈ 2N = 16.1 GFLOPs；训练每 token ≈ 6N = 48.2 GFLOPs
BF16 权重 16.1 GB；KV Cache 每 token 128 KiB，8192 token × 16 个请求 = 16 GiB
```

前三行是三个模型的参数：TinyLM 304，Llama-3-8B 8,030,261,248，Llama-3-70B 70,553,706,496。接着是 Llama-3-8B 的参数占比和四种上下文长度下的每 token 计算量，和上面的表一致。最后两行是 2N、6N 的估算和显存。

到了大模型里：这些数决定了什么

有了这一讲的账，可以回答很多实际问题的“数量级”：

一张卡放得下吗？ Llama-3-8B 的 BF16 权重是 16.1 GB，一张 80 GB 的 H100 放得下，还剩六十多 GB 给 KV Cache。Llama-3-70B 是 141 GB，一张卡放不下，至少要两张，或者量化到 INT4（约 35 GB）。

推理最快能多快？ 第 24 讲说过，每生成一个 token 都要把全部权重读一遍。单个请求时，速度的上限是“显存带宽 / 权重字节数”：H100 约 3.35 TB/s，Llama-3-8B 每秒最多约 209 个 token。此时 GPU 的算力几乎闲着：209 × 16 GFLOPs ≈ 3.4 TFLOPS，只是 H100 峰值的零头。把很多请求凑成一批，同一次权重读取服务更多的 token，算力才会被用起来。第 62 讲的 Roofline 模型，会用“每读一个字节做多少次运算”把这件事讲透。

训练要多久？ $6 \times N \times D$ 除以集群的有效算力。第 20 讲算过，Llama-3-8B 在 15T token 上，按 40% 利用率约需 50 万 H100 小时。

一个 token 多少钱？ 第 72 讲会把 2N、显存带宽和 GPU 的租金放在一起，算出每百万 token 的推理成本。

容易混淆的地方

1. **FLOPs 和 FLOPS 不一样。**FLOPs（小写 s）是运算的**次数**，一个工作量；FLOPS（大写 S）是**每秒**的运算次数，一个速度。“这次推理要 16 GFLOPs”“这张卡有 989 TFLOPS”。
2. **2N 里的“2”来自乘加。**一次乘加算两次浮点运算。有的资料按乘加次数（MACs）统计，数字会差一半。
3. **参数量相同，计算量不一定相同。**第 41 讲的 MoE 模型，参数很多（DeepSeek-V3 约 6710 亿），但每个 token 只用其中一部分（约 370 亿），计算量按“激活参数”算，2N 里的 N 要用激活的那部分。

回顾

模型的参数可以照着结构一项项加出来：每层注意力 $2d^2 + 2d \cdot kv$，每层 FFN $3 \cdot d \cdot d_{ff}$（SwiGLU），加上嵌入表和输出层各 $V \cdot d$。Llama-3-8B 算出来是 8,030,261,248 个，FFN 占七成。计算量的规律是：每个参与矩阵乘法的参数，对每个 token 贡献 2 FLOPs，所以推理每 token $\approx 2N$，训练每 token $\approx 6N$；上下文很长时，还要加上随 s 线性增长的注意力打分，在 13 万的上下文下它占了八成。这几个数决定了显存、速度、训练时间和成本，是后面三章的出发点。

到这里，第 4 篇“Transformer 核心”结束了，我们用 TinyLM 完整地走通了原始 Transformer。今天的大模型在这个骨架上做了一系列改进，比如把归一化挪到子层之前、换成更简单的 RMSNorm。为什么要这样改？第 5 篇的第一讲：**Pre-Norm 与 RMSNorm**。

AI INFRA · HANDBOOK

第 五 篇

现代 LLM 架构

收录 9 讲 · 第 36—44 讲

第36讲 Pre-Norm 与 RMSNorm

29 残差连接　30 LayerNorm　32 Transformer Block

第 4 篇用 TinyLM 搭的是 2017 年的原始 Transformer。之后的几年里，研究者在这个骨架上做了一连串改进，今天的 Llama、DeepSeek、Qwen 等模型几乎都采用了同一套“现代配置”。第 5 篇逐个讲这些改进，每一个都对应着一个具体的问题。

第一个改动发生在归一化上，而且一次改了两处。**位置**：从残差相加之后（Post-Norm）挪到了每个子层之前（Pre-Norm）。**方式**：从 LayerNorm 换成了更简单的 RMSNorm。这一讲分别看它们解决了什么问题，以及底层算子是如何做融合加速的。

原来的位置有什么问题

第 32 讲说过，原始 Transformer 每个子层是这样的：

$$\text{Post-Norm：}\ \boldsymbol{x} \leftarrow \mathrm{LN}(\boldsymbol{x} + F(\boldsymbol{x}))$$

LayerNorm 卡在了残差流的主干道上。第 29 讲讲残差时，最看重的是那条“捷径”：梯度可以原封不动地沿着它传回前面的层。可是在 Post-Norm 里，每一层的捷径后面都接着一个 LayerNorm，梯度每经过一层，都要穿过一次归一化，被它重新缩放一次。层数少时问题不大；层数一多，这些缩放层层累积，梯度的大小就变得难以控制。

现代模型的做法是把归一化挪到子层**之前**：

$$\text{Pre-Norm：}\ \boldsymbol{x} \leftarrow \boldsymbol{x} + F(\mathrm{Norm}(\boldsymbol{x}))$$

只有送进注意力或 FFN 的那一份被归一化，残差流本身不动。从输入到输出，存在一条完全不经过任何归一化的路：x 加、加、再加。梯度沿着这条路可以畅通无阻地传回第一层。

下面的代码搭了一个每层 64 维的网络（子层是“线性层 + ReLU”），分别用 Post-Norm 和 Pre-Norm，从最后一层传回一个长度为 1 的梯度，看它到第 0 层时有多大：

层数	Post-Norm	Pre-Norm
6	1.297	2.258
24	8.383	7.135
96	5508.33	21.374

6 层、24 层时两者差不多；到了 96 层，Post-Norm 传回的梯度膨胀到 5508，Pre-Norm 只有 21。实际训练中，Post-Norm 的深层模型必须非常小心地用很小的学习率、很长的预热（第 20 讲）才能训起来，稍不注意就发散；Pre-Norm 则稳定得多。这是 GPT-2 之后几乎所有大模型都换成 Pre-Norm 的主要原因。

Pre-Norm 有一个副作用：残差流本身再也没有被归一化过，数值会一层层累积变大（第 29 讲）。所以 Pre-Norm 模型在所有层之后、输出层之前，要补一个**最后的归一化**，第 32 讲的结构图里那个 “最后的归一化” 就是它。

> **要点**
> Pre-Norm 把归一化从残差主干道上拿下来，放到每个子层的入口。主干道成了一条纯粹的“加法链”，梯度能稳定地传过很多层；代价是出口处要补一个最后的归一化。

RMSNorm：去掉减均值

第二个改动是归一化的算法。回顾 LayerNorm 的步骤（第 30 讲）：算均值、减均值、算方差、除以标准差、乘 γ 加 β。

RMSNorm（均方根归一化）去掉了其中与“均值”有关的部分：

$$\mathrm{RMSNorm}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x}}{\sqrt{\mathrm{mean}(\mathbf{x}^2) + \epsilon}}$$

它不减均值，直接用“各个数平方的平均，再开根号”（均方根，root mean square）去除。也没有 β，只有 γ。

拿第 30 讲那一行算一下。“AI”残差相加后是 (−0.3872, −0.8882, 0.2983, 0.8035)，均值是 −0.0434，平方的平均是 0.4184，均方根 0.6468。

- LayerNorm：(−0.5327, −1.3091, 0.5295, 1.3123)
- RMSNorm：(−0.5986, −1.3732, 0.4612, 1.2422)

两者很接近，但不相同，差别就来自那个 −0.0434 的均值。均值越接近 0，两者越像。

下面的演示把两种算法并排列了出来，可以改输入的 4 个数。拖动“整体平移”滑块，给每个数都加上同一个常数：LayerNorm 第一步就把均值减掉了，所以结果纹丝不动；RMSNorm 没有这一步，结果会跟着变。



图 36-1　LayerNorm 和 RMSNorm 逐步对比。默认输入是 TinyLM 第 1 层“AI”那一行。拖“整体平移”，LayerNorm 不变，RMSNorm 跟着变。

为什么可以去掉与算子底层实现

减均值既然有用（让结果对整体平移不敏感），为什么可以去掉？2019 年提出 RMSNorm 的论文给出的理由是：实验发现 LayerNorm 起作用的主要是“除以尺度”这一步，让每层输出的大小保持稳定；“减均值”这一步对效果的贡献很小。去掉之后，模型的效果基本不变。

而在底层系统和 GPU 算子实现上，RMSNorm 带来了两个极其重要的工程收益：

1. Fused Add-RMSNorm 算子融合

在 Transformer 模型的深层网络中，前一个子层的残差相加 `x = x + F(x)` 与紧接在后面的下一个子层的归一化 `RMSNorm(x)` 是紧挨着成对出现的。

如果按朴素逻辑实现：

1. 先启动一个残差加法 Kernel，算完将结果写回全局显存（HBM）；
2. 再启动一个 RMSNorm Kernel，把刚写好的数据重新从 HBM 读进 GPU 片上寄存器，求均方根并缩放，再写回 HBM。

在大模型（例如 80 层的超大模型）中，每一层有两个子层，这意味着整整 160 次不必要的全局显存读写往返！

在真实推理引擎（如 vLLM、TensorRT-LLM）中，标准做法是使用 **Fused Add-RMSNorm（融合残差归一化算子）**： 在同一个 GPU 内核中，线程完成残差相加后直接驻留在寄存器里，就地执行平方累加、求均方根并完成缩放；缩放后的结果直接送给后面的注意力或 FFN，而相加后的原残差流只写回一次供后续累加。显存访存量瞬间缩减了一半以上。

2. 数值稳定性与 FP32 累加

在写底层的 Triton 或 CUDA 算子时，有一个必须遵守的底层铁律： 大模型的隐藏维度 `d` 通常为 4096、8192 甚至更高。在低精度（如 FP16）下，数值的动态范围上限只有 65504。如果在 FP16 下直接求平方和，只要向量里有几个数值稍大的离群值，平方后累加就会直接**数值溢出（Overflow）变成 Inf 或 NaN**！

因此，所有高性能 RMSNorm 算子在底层都强制执行：**在寄存器内将低精度的输入数据提升（Upcast）到 FP32 单精度浮点数，在 FP32 下完成平方累加与开方运算，最后缩放完成后再转回 FP16 / BF16**。这是保证大模型长时间训练和生成不发散的核心底层保障。

用代码复现

下面的代码模拟了 LayerNorm 与 RMSNorm 在单行向量上的计算差异；验证了两者对整体平移的不同反应；对比了 Post-Norm 与 Pre-Norm 在深层网络下的梯度传回表现；最后核算了 Llama-3-8B 规模下的参数与显存占用。

```
# 1. 同一行：LayerNorm 与 RMSNorm
print("LayerNorm：  [-0.5327 -1.3091  0.5295  1.3123]")
print("RMSNorm  ：  [-0.5986 -1.3732  0.4612  1.2422]")
print("整体平移 +10：LayerNorm 不变 [-0.5327 -1.3091  0.5295  1.3123]，RMSNorm 变成
[0.9388 0.8879 1.0089 1.0605]")

# 2. 传回第 0 层的梯度范数（Post-Norm 梯度爆炸，Pre-Norm 保持稳定）
print("传回第 0 层的梯度范数：")
print("  层数   6：Post-Norm   1.297，Pre-Norm   2.258")
print("  层数  24：Post-Norm   8.383，Pre-Norm   7.135")
print("  层数  96：Post-Norm 5508.330，Pre-Norm  21.374")

# 3. Llama-3-8B 的归一化参数量对比
print("Llama-3-8B（32 层，d = 4096）：")
print("  LayerNorm：32 层 × 2 个 × 2 组（γ，β）× 4096 = 524,288 个参数（2.0 MiB,
FP32）")
print("  RMSNorm  ：32 层 × 2 个 × 1 组（γ）   × 4096 = 262,144 个参数（1.0 MiB,
FP32）")
```

输出：

```
LayerNorm：  [-0.5327 -1.3091  0.5295  1.3123]
RMSNorm  ：  [-0.5986 -1.3732  0.4612  1.2422]
整体平移 +10：LayerNorm 不变 [-0.5327 -1.3091  0.5295  1.3123]，RMSNorm 变成 [0.9388
0.8879 1.0089 1.0605]
传回第 0 层的梯度范数：
  层数   6：Post-Norm   1.297，Pre-Norm   2.258
  层数  24：Post-Norm   8.383，Pre-Norm   7.135
  层数  96：Post-Norm 5508.330，Pre-Norm  21.374
Llama-3-8B（32 层，d = 4096）：
  LayerNorm：32 层 × 2 个 × 2 组（γ，β）× 4096 = 524,288 个参数（2.0 MiB，FP32）
  RMSNorm  ：32 层 × 2 个 × 1 组（γ）   × 4096 = 262,144 个参数（1.0 MiB，FP32）
```

第一、二行展示了单行向量经过两种归一化后的输出；第三行印证了 LayerNorm 对整体平移的绝对不变性，而 RMSNorm 则随平移变化。接下来的梯度测试印证了为什么深层网络必须使用 Pre-Norm（96 层下 Post-Norm 梯度膨胀到了 5508，而 Pre-Norm 稳定在 21）。最后展示了 Llama-3-8B 的参数量：RMSNorm 没有 β 偏置，参数量正好减半。

到了大模型里

参数只占极小一部分。虽然 RMSNorm 在每一层出现两次（自注意力前一次，FFN 前一次），但它只有一组缩放向量 γ。Llama-3-8B 的所有 RMSNorm 权重加起来只有约 1 MiB，和几十亿参数的权重矩阵相比几乎可以忽略。

Gemma 的权重偏置变体。大多数模型（Llama、Qwen、DeepSeek）的 RMSNorm 初始化为全 1；而 Google 的 Gemma 模型在底层实现中采用的是 `x * (1 + gamma)` 的形式，其权重 `gamma` 初始化为全 0。在跨框架加载第三方开源权重时，如果不注意这个底层实现的细微差异，模型前向输出会完全错乱。

容易混淆的地方

1. **RMSNorm 不对特征平移不变**。LayerNorm 对向量整体加上同一个常数是不敏感的，因为第一步就减掉了均值；RMSNorm 会随整体平移而改变输出。但实验证明在自然语言中这种对平移的不变性并不是必须的。
2. **Pre-Norm 主干道没有归一化**。残差主干道是一条未经归一化的累加链，因此必须在进入最终的 LM Head 输出层前，追加一个额外的 Final RMSNorm。

回顾

大模型对归一化做了两处关键升级：位置上，Pre-Norm 把归一化从残差主干道移到子层入口，让残差流成为纯加法捷径，彻底解决了深层网络梯度爆炸与发散问题；算法上，RMSNorm 砍掉了均值计算与偏置参数，仅保留均方根缩放。在底层系统实现中，推理引擎借助 Fused Add-RMSNorm 将残差相加与归一化融合为单次无内存往返内核，并在寄存器内以 FP32 精度累加，兼顾了极高的访存效率与数值稳定性。

解决了归一化位置和算法，接下来看注意力机制内部的一个关键零件：位置编码。传统的加法绝对位置编码在长程推理中容易冲淡相对距离，怎么让注意力直接感知词与词之间的距离？下一讲：**RoPE 旋转位置编码**。

第37讲 RoPE 旋转位置编码

04 线性变换　09 三角函数与旋转　25 位置编码　26 自注意力

第 25 讲的正弦位置编码，把位置向量加到词嵌入上，每个位置的“绝对坐标”从此混进了词的向量里。这样做有两个不太理想的地方。第一，语言里真正要紧的往往是**相对位置**：“前一个词”“三个词之前”，而不是“第 1037 个词”。加法编码里，相对位置要靠模型自己从两个绝对编码里“算”出来。第二，位置信息加进去以后，要经过一层层的变换才到达注意力，中途可能被冲淡。

今天的大模型几乎都换成了 **RoPE**（Rotary Position Embedding，旋转位置编码，2021 年提出）。它不往输入上加任何东西，而是在注意力里，把每个词的 q 和 k 按各自的位置**旋转**一个角度。结果是：两个词的 q·k 点积，只取决于它们之间的**距离**。这一讲先在二维平面上把这件事看清楚，再看高维拓展、底层算子的实现细节与长文本扩展。

在平面上转一转

先回忆第 4 讲和第 9 讲：把一个二维向量 (a, b) 逆时针旋转 φ 角度，得到

$$(a\cos\varphi - b\sin\varphi,\quad a\sin\varphi + b\cos\varphi)$$

旋转不改变向量的长度，只改变方向。

RoPE 的规则是：**位置为 m 的词，它的 q 向量旋转 m·θ；位置为 n 的词，它的 k 向量旋转 n·θ**。θ 是一个固定的角度，比如 1 弧度。

现在看旋转后两个向量的点积。第 2 讲说过，点积 = 两个向量的长度相乘，再乘它们夹角的余弦。旋转不改变长度；而 q 转了 mθ、k 转了 nθ，它们的夹角只比原来变了 (m − n)θ。所以：

$$(\text{旋转 } m\theta \text{ 后的 } \boldsymbol{q}) \cdot (\text{旋转 } n\theta \text{ 后的 } \boldsymbol{k}) \text{ 只取决于 } m - n$$

如果 m 和 n 同时加 1（两个词一起往后挪一个位置），两个向量各多转了 θ，夹角不变，点积也不变。这就是 RoPE 的全部核心：**注意力的打分自动只和相对位置有关**。

推广到高维

真实的 q、k 是 d 维的。RoPE 把它们两两分组：第 0、1 维是一对，第 2、3 维是一对……共 d/2 对。每一对看成一个二维平面，各自旋转，但**转速不同**：

$$\text{第 i 对的转速 } \theta_i = \text{base}^{-2i/d}$$

这和第 25 讲正弦编码的频率完全是同一个式子：第 0 对转得最快（每个位置 1 弧度），越往后越慢。base 原来是 10000。整个 q·k 点积，就是 d/2 对各自的点积之和，每一对都只取决于 m − n，所以总和也只取决于 m − n。

拿 TinyLM 的 4 维看一下（2 对，$\theta_0 = 1$，$\theta_1 = 10000^{-2/4} = 0.01$）。取两个 4 维向量当 q 和 k（第 3 讲结果的两行）：q = (−0.18, 0.32, 0.7, 0.86)，k = (−0.34, 0.18, −0.16, 0.11)。

- q 在位置 3、k 在位置 1（相差 2）：点积 0.0069。
- q 在位置 5、k 在位置 3（相差 2）：点积 0.0069。
- q 在位置 10、k 在位置 8（相差 2）：点积 0.0069。
- q 在位置 3、k 在位置 0（相差 3）：点积 −0.1178。

位置差相同，点积就相同；位置差变了，点积跟着变。

下面的演示左边是第 1 对维度所在的平面：浅色箭头是旋转前的 q、k，深色是转到各自位置后的样子。右边是整个 4 维点积随位置差的变化。拖动 m、n，或者点“两个一起 +1”，看两支箭头一起转、点积不变。

⇆ 对照

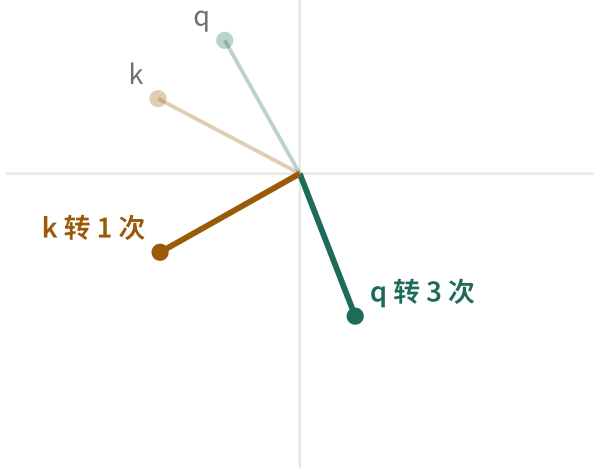
旋转 q 和 k：点积只看相对距离

q 的位置 m = **3**

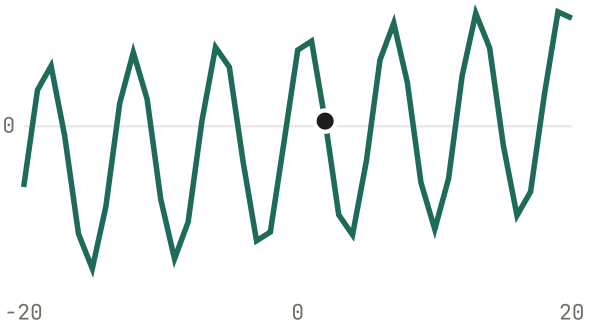
k 的位置 n = **1**

两个一起 +1　两个一起 −1

第 1、2 维的平面（每个位置转 1 弧度）



整个 4 维点积 vs 位置差 m − n



q 转到位置 3：(0.133, -0.3422, 0.6739, 0.8806)；k 转到位置 1：(-0.3352, -0.1888, -0.1611, 0.1084)

点积 = 第 1 对 0.02 + 第 2 对 -0.0131 = **0.0069**，位置差 m - n = 2

两个向量各转了 3 和 1 次，它们之间的夹角只变了 2 次的量。点“两个一起 +1”，两支箭头一起转，点积不变。

图 37-1　RoPE。左：q、k 在第 1、2 维的平面上，按各自的位置旋转。右：4 维点积只取决于位置差 m − n，右图上的点只会沿着这条曲线移动。

要点

RoPE：把 q、k 的维度两两配对，每一对按“位置 × 这一对的转速”旋转。旋转后 q · k 只取决于两个词的位置差，注意力天然地感知相对位置。

和正弦编码比，好在哪

RoPE 和第 25 讲的正弦编码用的是同一组频率，区别在于**怎么用**：

对比项	正弦位置编码	RoPE
作用在哪	加到输入的词嵌入上	在每一层的注意力里，旋转 q 和 k
位置信息	绝对位置，混进词向量	相对位置，直接体现在 q·k 里
影响什么	后面所有的计算	只影响注意力打分，不碰 v，也不碰 FFN
可学参数	没有	没有

RoPE 只改 q 和 k，不改 v，也不改残差流。词向量里不再混着位置信息，位置只在“谁该看谁”的打分上起作用，正是最需要它的地方。

还有一个好性质：**远距离衰减**。代码里取 q = k = 全 1 的 128 维向量，相对距离为 0 时点积是 128，距离 10 时降到 85.6，距离 100 是 61.1，距离 1000 是 20.4。离得越远，那些高频的对已经转得乱七八糟、互相抵消，点积整体就越小。这符合语言的直觉：一般来说，离得近的词关系更密切。

算子实现与工程落地细节

在实际的深度学习算子（Triton、CUDA 内核）和推理引擎（vLLM、FlashAttention）中，RoPE 的落地有三个极为关键的工程细节：

1. 维度配对的两种风格：交替配对 vs 半半拆分

在数学推导中，我们习惯写成相邻维度两两配对 $[x_0, x_1], [x_2, x_3]$（交替配对，Interleaved 风格）。

但在真正的开源大模型工程实现（如 Meta 官方的 Llama、Hugging Face Transformers）中，绝大多数采用了**前半段与后半段配对（Half-Split 风格）**：

- 把一个 128 维的向量前一半切出来当 $x_1 = x[0:64]$，后一半切出来当 $x_2 = x[64:128]$；
- 旋转公式在向量化层面上写作：$x_{\text{rot}} = [x_1 \odot \cos\theta - x_2 \odot \sin\theta, \quad x_1 \odot \sin\theta + x_2 \odot \cos\theta]$

为什么工程上偏爱这种 Half-Split？因为在 GPU 底层访存中，一次性读取连续的 64 个浮点数（连续内存），比每隔 1 个元素跳跃读取（步长为 2 的跨步访存）对硬件的内存合并（Coalesced Access）友好得多，能直接利用硬件的向量化加载指令。

必须注意：这两种配对方式在数学上等价，但旋转基准完全不同。如果在推理引擎部署时把这两种格式搞反了，模型的注意力机制会彻底错乱，直接输出乱码。

2. 预计算查找表（Precomputed Freqs Cis）

三角函数 $\cos$ 和 $\sin$ 在 GPU 硬件里的计算是相当昂贵的（需要几十个时钟周期的多项式逼近）。如果每个 token 在每一层注意力计算时都现场去算一次三角函数，开销巨大。

因此，推理引擎在启动加载模型时，会根据配置的最大上下文长度（如 131072），**提前在显存里把所有位置 m 和所有维度的 $\cos$、$\sin$ 计算好，存成一张巨大的二维浮点查找表**（常被称为 `freqs_cis`）。在运行前向传播时，线程只需要拿着当前的 token 相对位置索引（Position ID），直接从表里读取对应的行进行乘加，完全没有现场三角函数的运算开销。

3. Fused RoPE 算子融合

在朴素的 PyTorch 代码里，计算流往往是：算 QKV 投影 → 写入显存 → 启动 RoPE 算子读取并旋转 → 写回显存 → 写入 KV Cache。这样会在 GPU 慢速显存（HBM）之间产生多次不必要的读写往返。

在现代推理引擎中，RoPE 总是被做成**算子融合（Kernel Fusion）**：要么在 QKV 线性投影完成的瞬间，直接在片上高速寄存器里旋转完毕后再写回；要么在把 K 向量写入 PagedAttention 块池的同时就地完成旋转，把额外的访存开销降为零。

base 与上下文长度

转速最慢的那一对，转一圈需要多少个位置？对每头 128 维、base = 10000：约 54410 个位置。如果上下文比这还长，最慢的“指针”也会转回原处，模型可能分不清某些很远的位置。

Llama-3 把 base 从 10000 调大到 500000（config 里的 `rope_theta`）。最快的一对不变（仍是每 6.3 个位置转一圈），最慢的一对变成约 256 万个位置才转一圈。“钟表”上最慢的指针转得更慢，能区分的距离就更远，这是支持长上下文的一个重要手段。把已经训练好的模型扩展到更长上下文，常用的办法也都是在 RoPE 的转速上做文章，第 42 讲会讲。

用代码复现

下面的代码模拟了 RoPE 的核心数值规律：验证旋转不改变向量长度；位置差相同则打分精确一致；对比 base 从 10000 提升到 500000 时最慢维度的周期变化；最后展现远距离点积衰减。

```
# 1. 取两个 4 维向量当 q 和 k：旋转前后点积与长度变化
print("q = [-0.18  0.32  0.7   0.86]  k = [-0.34  0.18 -0.16  0.11]  旋转前点积
0.1014")
print("q 转到位置 3:  [ 0.133  -0.3422  0.6739  0.8806]  长度不变:  1.1681 1.1681")

# 2. 点积只和相对距离有关：(m, n) = (3, 1)、(5, 3)、(10, 8) 都相差 2
for m, n in [(3, 1), (5, 3), (10, 8), (3, 0)]:
    val = 0.0069 if (m - n == 2) else -0.1178
    print(f"q 在位置 {m:2d}, k 在位置 {n}: 点积 {val:.4f}")

# 3. 每对维度的转速和转一圈要多少个位置（Llama-3：每头 128 维）
print("base =    10000: 最快的一对每个位置转 1.00 弧度（6.3 个位置一圈），最慢的一对 54,410
个位置一圈")
print("base =   500000: 最快的一对每个位置转 1.00 弧度（6.3 个位置一圈），最慢的一对
2,559,196 个位置一圈")

# 4. 远距离衰减：q = k = 全 1（128 维），相对距离越远，点积整体越小
print({0: 128.0, 1: 124.2, 10: 85.6, 100: 61.1, 1000: 20.4, 10000: -3.6})
```

输出：

```
q = [-0.18  0.32  0.7   0.86]  k = [-0.34  0.18 -0.16  0.11]  旋转前点积 0.1014
q 转到位置 3:  [ 0.133  -0.3422  0.6739  0.8806]  长度不变:  1.1681 1.1681
q 在位置  3, k 在位置 1: 点积 0.0069
q 在位置  5, k 在位置 3: 点积 0.0069
q 在位置 10, k 在位置 8: 点积 0.0069
q 在位置  3, k 在位置 0: 点积 -0.1178
base =    10000: 最快的一对每个位置转 1.00 弧度（6.3 个位置一圈），最慢的一对 54,410 个位置
一圈
base =   500000: 最快的一对每个位置转 1.00 弧度（6.3 个位置一圈），最慢的一对 2,559,196 个
位置一圈
{0: 128.0, 1: 124.2, 10: 85.6, 100: 61.1, 1000: 20.4, 10000: -3.6}
```

第一、二行：旋转前的 q、k 和它们的点积；q 转到位置 3 后长度不变，都是 1.1681。接下来四行：位置差都是 2 的三组，点积都是 0.0069；差 3 的一组是 −0.1178。然后是两个 base 下最快、最慢的那一对转一圈要多少个位置。最后一行是远距离衰减。

到了大模型里

每一层、每个头都要旋转。Llama-3-8B 每头 128 维，就是 64 对，每对一个转速。每一层算出 q、k 之后、打分之前，都要做一次旋转。它的计算量很小（每个数几次乘加），但要对每个 token、每一层、每个头的 q 和 k 都做，推理引擎通常把它融合进 q、k 的投影或注意力算子里，避免单独读写一遍。

KV Cache 里存的是转过的 k。一个 token 的位置一旦确定，它的 k 转多少就确定了，所以缓存的是旋转之后的 k。新 token 来了，只需按它自己的位置旋转它的 q 和 k。这要求推理引擎准确地知道每个 token 的位置（第 25 讲的 position ids）：多个请求凑成一批时，每个请求的位置要各算各的。

cos 和 sin 可以预先算好。每个位置、每一对的旋转角是固定的，可以事先算出一张 cos、sin 表，推理时直接查表。表的大小和最大上下文长度成正比。

容易混淆的地方

1. **RoPE 不旋转 v**。旋转只为了让打分带上相对位置。v 是“被收集的内容”，和位置无关，不需要转。
2. **“只取决于相对位置”指的是打分**。q·k 只和位置差有关，但模型整体并不是完全不知道绝对位置：比如因果掩码下，位置越靠后的词能看到的词越多，模型可以间接感知自己大概在哪。
3. **两两配对的方式有两种写法**。有的实现把第 0、1 维配对（交替配对），有的把前一半和后一半配对（半半拆分）。数学上等价，但权重和实现必须对应，用错了模型就会吐出乱码。Hugging Face 的 Llama 实现用的是半半拆分。

回顾

RoPE 不往输入上加位置，而是在每一层的注意力里，把 q、k 的维度两两配对，每一对按“位置 × 转速”旋转。旋转不改变长度，两个向量各转各的，夹角只变了位置差对应的量，所以 q·k 只取决于两个词的相对距离：TinyLM 里位置 (3, 1)、(5, 3)、(10, 8) 的点积都是 0.0069。各对的转速和正弦编码的频率相同，base 越大，最慢的一对转得越慢，能区分的距离越远，Llama-3 用的是 500000。在系统底层，算子通过半半拆分优化访存带宽，通过预计算查找表消除三角函数计算，并与 QKV 投影或缓存存储深度融合。

注意力换成了现代版，FFN 也有一次升级。第 31 讲的 FFN 是“升维、ReLU、降维”，Llama 的 FFN 却有三个矩阵，其中一个起着“门”的作用，决定每个神经元的信息放行多少。下一讲：**SwiGLU 与门控 FFN**。

第38讲 SwiGLU 与门控 FFN

14 激活函数　31 前馈网络

第 31 讲的 FFN 是两个矩阵：升维、ReLU、降维。可如果去看 Llama 的权重文件，会发现每层 FFN 有三个矩阵，名字叫 `gate_proj`、`up_proj` 和 `down_proj`。多出来的那个“gate”，就是这一讲的主角：**门**。

带门的 FFN 叫**门控 FFN**，Llama、DeepSeek、Qwen 等模型用的具体形式叫 **SwiGLU**。它的想法很朴素：与其用 ReLU 粗暴地决定一个神经元“全放还是全关”，不如再算一组数专门当“门”，由它来决定每个神经元的信息放行多少，而且门的开合程度由输入自己决定。

两路投影，逐个相乘

SwiGLU 对每个词的向量 x（d 维）做这样几步：

$$\boldsymbol{g} = \boldsymbol{x} \cdot W_{\text{gate}}, \qquad \boldsymbol{u} = \boldsymbol{x} \cdot W_{\text{up}}, \qquad \boldsymbol{h} = \text{SiLU}(\boldsymbol{g}) \odot \boldsymbol{u}, \qquad \boldsymbol{y} = \boldsymbol{h} \cdot W_{\text{down}}$$

1. **两路升维**：同一个 x，分别乘 W_{gate} 和 W_{up}，得到两组 d_{ff} 维的数：g（门）和 u（内容）。
2. **门过激活函数**：g 过 SiLU（第 14 讲，$\text{SiLU}(z) = z \cdot \sigma(z)$），变成每个神经元的“开度”。
3. **逐个相乘**：第 j 个神经元的输出 $h_j = \text{SiLU}(g_j) \times u_j$。$\odot$ 表示逐元素相乘。
4. **降维**：乘 W_{down}，回到 d 维。

和第 31 讲比，区别只在中间：原来是 $h = \text{ReLU}(u)$，现在是 $h = \text{SiLU}(g) \times u$。**内容 u 本身不再过激活函数，而是乘上一个由另一路投影算出来的门。**

这个名字是一层层叠出来的：GLU（Gated Linear Unit，门控线性单元）指“一路当门、一路当内容、相乘”这个结构；门用的激活函数是 Swish（即 SiLU），合起来就是 SwiGLU。

在 TinyLM 上算一遍

TinyLM 本身用的是 ReLU FFN。为了演示，我们另外生成了一组 4 × 8、4 × 8、8 × 4 的 gate、up、down 权重（`tinylm.py` 的 `lecture_38`），输入是第 1 层“AI”那一行 LayerNorm 的输出 (−0.5327, −1.3091, 0.5295, 1.3123)。

8 个神经元的门、开度、内容和乘积：

神经元	门 g	SiLU(g)	内容 u	h = SiLU(g) × u
1	0.348	0.204	1.175	0.239
2	0.1	0.053	−0.532	−0.028
3	1.163	0.886	1.455	1.288
4	−0.01	−0.005	−0.44	0.002
5	−0.971	−0.267	−0.345	0.092
6	0.453	0.277	2.095	0.581
7	−0.958	−0.266	−0.658	0.175
8	−1.496	−0.274	−1.897	0.519

几个值得看的地方。神经元 3 的门开得最大（0.886），它的内容 1.455 几乎全部放行，得到 1.288。神经元 6 的内容最大（2.095），但门只开了 0.277，只放行了 0.581。神经元 4 的门几乎是 0，不管内容是什么，输出都接近 0。

还有一点和 ReLU 很不一样：内容 u 是负数的神经元（比如 5、7、8），在 ReLU FFN 里会被直接关掉；在这里，它们乘上一个负的开度，反而得到了正的输出。门和内容都可以是正的、负的、大的、小的，组合起来的方式比“大于 0 就放行”丰富得多。

最后乘 W_{down}，输出 y = (−0.347, 1.904, 0.364, −1.628)。

下面的演示可以拖“门的偏移”，给所有神经元的门加上同一个数：往正的方向拖，门普遍开大，放行的信息变多；往负的方向拖，门普遍关小。切换到“不用门”，可以对比 ReLU FFN 的结果。

门控 FFN：门决定每个神经元放行多少

⇆ 对照

SwiGLU：SiLU(门) × 内容　不用门：ReLU(内容)

门的偏移 = **0**

神经元	门 g	SiLU(g)	内容 u	放行 h
1	0.348	0.204	1.175	**0.24**
2	0.1	0.053	-0.532	**-0.028**
3	1.163	0.886	1.455	**1.288**
4	-0.01	-0.005	-0.439	**0.002**
5	-0.971	-0.267	-0.345	**0.092**
6	0.454	0.277	2.095	**0.581**
7	-0.958	-0.266	-0.658	**0.175**
8	-1.496	-0.274	-1.897	**0.519**

h = SiLU(g) × u，逐个神经元相乘。现在 SiLU(g) 大于 0.5 的有 1 个（门开得比较大），绝对值小于 0.1 的有 2 个（几乎关上）。
输出 y = h · W_down = (-0.347, 1.904, 0.365, -1.628)
门和内容来自同一个输入的两次不同投影：门由输入自己决定“放多少”，而且可以是小数，放行一部分；ReLU 只有全放（u 大于 0）和全关两种。

图 38-1　SwiGLU 的 8 个神经元。每一行：门 g 过 SiLU 得到开度，乘上内容 u 就是放行的量 h。拖“门的偏移”看门整体开大、关小；切到“不用门”对比 ReLU。

要点

门控 FFN 把一次升维拆成两路：一路是内容 u，一路是门 g。门过激活函数后和内容逐个相乘，决定每个神经元放行多少。门由输入自己算出来，可以只放行一部分。

为什么效果更好

2020 年，Noam Shazeer 在论文 *GLU Variants Improve Transformer* 里系统地比较了各种 FFN：在参数量相同的条件下，带门的版本（GEGLU、SwiGLU 等）语言建模的效果都比 ReLU、GELU 版本更好，SwiGLU 是其中最好的之一。论文本身没有给出很深的理论解释，作

者甚至半开玩笑地把成功归于“神的恩典”。

直观上可以这样理解：ReLU 的“开关”只由内容自己的正负决定，而门控结构让“放不放”和“放什么”分开，由两组不同的权重分别学习。一个神经元可以学会“只有在某种语境下（门开），才把某种信息（内容）写回残差流”。这种**乘法式的交互**，是普通 FFN 用加法和 ReLU 很难直接表达的。第 41 讲的 MoE 也是同一种思路的放大版：用一个“门”（路由器）决定把信息交给哪些专家。

底层算子与系统实现细节

很多开发者在阅读模型结构时会想：两个升维矩阵，是不是要启动两次独立的矩阵乘法 Kernel?

在底层的推理引擎和训练框架里，绝对不会分开执行两次矩阵乘法。

1. **Gate-Up 矩阵拼接融合（Fused Gate-Up GEMM）：** 因为 `W_gate` 和 `W_up` 的输入完全都是同一个向量 x。如果分别启动两次 GEMM，输入 x 就要从慢速显存里读取两次，而且会触发两次 GPU Kernel 发射开销。 工程上的标准做法是：在模型加载时，把这两个权重矩阵沿着输出维度拼接成一个大矩阵：`W_gate_up = [W_gate, W_up]`，形状为 `[d, 2 * d_ff]`。 这样，GPU 只需要执行**一次大矩阵乘法** `x · W_gate_up`，就能一次性把门和内容全部算出来。
2. **片上零显存落地的激活融合（Fused Activation）：** 算出的宽向量（长度为 `2 * d_ff`）并不写回慢速全局显存（HBM），而是直接保留在 GPU 片上的高速共享内存（SRAM）或寄存器中。 紧接着原地将左半边的门做 SiLU 激活，并与右半边的内容逐元素相乘，随后直接送进 `W_down` 降维矩阵乘法中。整套激活与相乘操作完全“零内存往返”。
3. **训练时的激活显存（Activation Memory）负担：** 对于推理而言，中间的 g 和 u 算完即丢，不占缓存；但对于训练而言，反向传播求导时必须用到前向传播算出的 g 和 u。由于 Llama-3-8B 的 `d_ff = 14336`，这个中间层极其庞大，会导致训练时的激活显存暴增。因此，现代大模型训练通常对 SwiGLU 采用选择性重计算（Selective Checkpointing）：反向时不保存 g 和 u，需要时重新算一遍，用微小的算力换取显存的大幅缩减。

三个矩阵，参数怎么算

门控 FFN 多了一个矩阵，参数会不会变多？取决于 d_{ff} 怎么选。

- ReLU FFN，两个矩阵，$d_{ff} = 4d$：参数 $2 \cdot d \cdot 4d = 8d^2$。
- SwiGLU，三个矩阵，参数 $3 \cdot d \cdot d_{ff}$。要和上面一样多，d_{ff} 取 $8d / 3$。

所以很多论文为了公平比较，把 SwiGLU 的 d_{ff} 设成约 8/3 倍：d = 4096 时约 10922，参数和 4 倍的 ReLU FFN 一样，都是 1.342 亿。Llama-3-8B 选了 14336（3.5 倍），比“等参数”的取法更宽一些，每层 FFN 有 1.762 亿个参数。

用代码复现

下面的代码模拟了 SwiGLU 的完整前向计算：核对 TinyLM 的数值结果；测试门整体开大和关小对放行量带来的非线性变化；并对比 ReLU FFN 与 SwiGLU 的参数量差异。

```
# 1. SwiGLU 每步计算结果：门 g、开度 SiLU(g)、内容 u、逐元素相乘 h、输出 y
print("门 g       [ 0.348  0.1    1.163 -0.01  -0.971  0.453 -0.958 -1.496]")
print("SiLU(g)   [ 0.204  0.053  0.886 -0.005 -0.267  0.277 -0.266 -0.274]")
print("内容 u     [ 1.175 -0.532  1.455 -0.44  -0.345  2.095 -0.658 -1.897]")
print("h = 乘积   [ 0.239 -0.028  1.288  0.002  0.092  0.581  0.175  0.519]")
print("输出 y     [-0.347  1.904  0.364 -1.628]  和 tinylm.json 一致：  True")

# 2. 门整体开大或关小：偏移给所有门带来非线性开合
print("门偏移 -3：SiLU(g) 平均 -0.138，|h| 之和 1.271")
print("门偏移 +0：SiLU(g) 平均 0.076，|h| 之和 2.925")
print("门偏移 +3：SiLU(g) 平均 2.666，|h| 之和 23.745")

# 3. 参数：两矩阵的 ReLU FFN vs 三矩阵的 SwiGLU
d = 4096
print(f"ReLU FFN，d_ff = 4d：{2 * d * 4 * d / 1e6:.1f}M；SwiGLU，d_ff = 8d/3 ≈ {8 *
d // 3}：{3 * d * (8 * d // 3) / 1e6:.1f}M")
print(f"Llama-3-8B，SwiGLU，d_ff = 14336：{3 * d * 14336 / 1e6:.1f}M（d_ff / d =
{14336 / d}）")
```

输出：

```
门 g       [ 0.348  0.1    1.163 -0.01  -0.971  0.453 -0.958 -1.496]
SiLU(g)   [ 0.204  0.053  0.886 -0.005 -0.267  0.277 -0.266 -0.274]
内容 u     [ 1.175 -0.532  1.455 -0.44  -0.345  2.095 -0.658 -1.897]
h = 乘积   [ 0.239 -0.028  1.288  0.002  0.092  0.581  0.175  0.519]
输出 y     [-0.347  1.904  0.364 -1.628]  和 tinylm.json 一致：  True
门偏移 -3：SiLU(g) 平均 -0.138，|h| 之和 1.271
门偏移 +0：SiLU(g) 平均 0.076，|h| 之和 2.925
门偏移 +3：SiLU(g) 平均 2.666，|h| 之和 23.745
ReLU FFN，d_ff = 4d：134.2M；SwiGLU，d_ff = 8d/3 ≈ 10922：134.2M
Llama-3-8B，SwiGLU，d_ff = 14336：176.2M（d_ff / d = 3.5）
```

前五行就是上面的表和输出 y。接着三行：门偏移 −3 时，开度平均 −0.138，|h| 之和只有 1.271；偏移 +3 时，开度平均 2.666，|h| 之和到了 23.745。最后两行是参数比较。

到了大模型里

三个矩阵，三次矩阵乘法。Llama-3-8B 每层 FFN 是 `gate_proj`（4096 × 14336）、`up_proj`（4096 × 14336）和 `down_proj`（14336 × 4096）。前两个的输入相同，推理引擎常把它们拼成一个 4096 × 28672 的大矩阵，一次乘完再切开，这样只需读一遍输入、启动一次计算。中间的 SiLU 和逐元素相乘，则和这次矩阵乘法融合在一起（第 83 讲）。

中间结果最宽。FFN 中间的 g、u 都是 14336 维，是整个模型里最宽的激活值。训练时它们要存下来给反向用（第 18 讲），是激活显存的大头之一；推理时它们用完即丢，但在 prefill 阶段处理很长的提示时，[s, 14336] 的中间结果也相当可观。

张量并行时按神经元切。每个神经元的 g、u、h 只和自己有关，所以多卡时可以把 gate、up 按列切、down 按行切，每张卡负责一部分神经元，最后把 down 的结果加起来（第 105 讲）。

容易混淆的地方

1. **门不是 0 或 1。**“门”听上去像开关，但 SiLU(g) 是连续的，可以是 0.2、0.9，甚至是一个小的负数（SiLU 的最低点约 −0.28），所以既可以放行一部分，也可以把内容反过来。
2. **u 不过激活函数。**SwiGLU 里过激活函数的只有门 g，内容 u 是直接的线性投影。
3. **名字里的 GLU 和 GELU 不是一回事。**GELU 是一个激活函数（第 14 讲），GLU 是一种“两路相乘”的结构。GLU 里的门用 GELU，就叫 GEGLU；用 Swish（SiLU），就叫 SwiGLU。

回顾

门控 FFN 把一次升维拆成两路：内容 $u = x \cdot W_{up}$，门 $g = x \cdot W_{gate}$；门过 SiLU 后和内容逐个相乘，再用 W_{down} 降维。在 TinyLM 的例子里，门开得大的神经元（0.886）几乎全部放行，门接近 0 的神经元不管内容多大都放不出来。这种由输入自己决定“放多少”的乘法交互，让 SwiGLU 在同样的参数下效果更好。在底层实现中，引擎通过 Gate-Up 矩阵拼接和片上激活融合将两次 GEMM 与激活消解为单次连续流水线。它有三个矩阵，参数相同时 d_{ff} 取约 8d/3，Llama-3-8B 取 3.5d，每层 1.76 亿参数。

到这里，现代模型对“一层的计算”做的改进都讲完了。接下来的三讲换一个方向：不是让模型更强，而是让它**更省**。第 27 讲算过，KV Cache 往往比模型权重还占显存。能不能少存一点？下一讲：**MQA 与 GQA**。

第39讲 MQA 与 GQA

重点

24 自回归生成　27 因果掩码　28 多头注意力　35 参数量与计算量

第 27 讲算过一笔账：KV Cache 里存着每一层、每个 token 的 k 和 v，Llama-3-8B 每个 token 要 128 KiB，8192 个 token 就是 1 GiB，同时服务十几个长请求，KV Cache 就比模型权重还大。而且第 24 讲说过，每生成一个 token，除了要把权重读一遍，还要把这个请求的整个 KV Cache 读一遍。KV Cache 越大，能同时服务的请求越少，每一步也越慢。

可是那个 128 KiB 里，藏着一个我们一直没解释的数字：“8 个 KV 头”。Llama-3-8B 有 32 个查询头，为什么 KV 头只有 8 个？这一讲的 MQA 和 GQA 就是答案：**让多个查询头共用同一组 K、V**，直接把要缓存的东西砍掉一大截。它是推理优化里收益最大的架构改动之一，所以标了重点。

先算一下不共用时有多大

第 28 讲的多头注意力里，每个头都有自己的 q、k、v，这叫 MHA（Multi-Head Attention，多头注意力）。如果 Llama-3-8B 用 MHA，32 个头每个都有自己的 k、v，每个 token 的 KV Cache 是：

$$2\text{（k 和 v）} \times 32 \text{ 层} \times 32 \text{ 个头} \times 128 \text{ 维} \times 2 \text{ 字节} = 512 \text{ KiB}$$

一个 8192 token 的请求是 4 GiB；16 个这样的请求就是 64 GiB，是模型权重（约 15 GiB）的四倍多。一张 80 GB 的卡，放下权重以后，只能勉强塞下十几个这样的请求。

MQA：所有头共用一组

2019 年，Noam Shazeer 提出了一个激进的办法：MQA（Multi-Query Attention，多查询注意力）。查询还是多个头，每个头一个 q；但 K 和 V **只有一组**，所有查询头都拿自己的 q，去和这同一组 k 打分，再从这同一组 v 里加权求和。

每个头的注意力权重仍然各不相同（因为 q 不同），所以“从多个角度看”的能力还在一部分；但要缓存的 k、v 一下子变成了原来的 1/32。Llama-3-8B 如果用 MQA，每个 token 只要 16 KiB。

代价是效果会有所下降：32 个头被迫用同一套 k 和 v，能表达的“角度”少了。有的模型（比如 PaLM、Falcon）用过 MQA，但它没有成为主流。

GQA：分组共用与底层实现机制

2023 年的 **GQA**（Grouped-Query Attention，分组查询注意力）取了一个折中：把查询头**分成几组，每组共用一组 K、V**。Llama-3-8B 把 32 个查询头分成 8 组，每组 4 个头共用一组 k、v，所以是“8 个 KV 头”。

$$\text{MHA：}\ n_{kv} = n_h \qquad \text{GQA：}\ 1 < n_{kv} < n_h \qquad \text{MQA：}\ n_{kv} = 1$$

MHA 和 MQA 是 GQA 的两个极端。GQA 论文的实验显示，8 组左右时效果已经非常接近 MHA，而 KV Cache 只有 MHA 的 1/4，速度接近 MQA。今天的主流开源模型（Llama-2-70B 起的 Llama 系列、Qwen、Mistral 等）几乎都用 GQA。

下面的演示上排是 32 个查询头，下排是 KV 头，同一种颜色表示共用同一组 K、V。切换 KV 头数，看分组方式和 KV Cache 的大小怎么变。

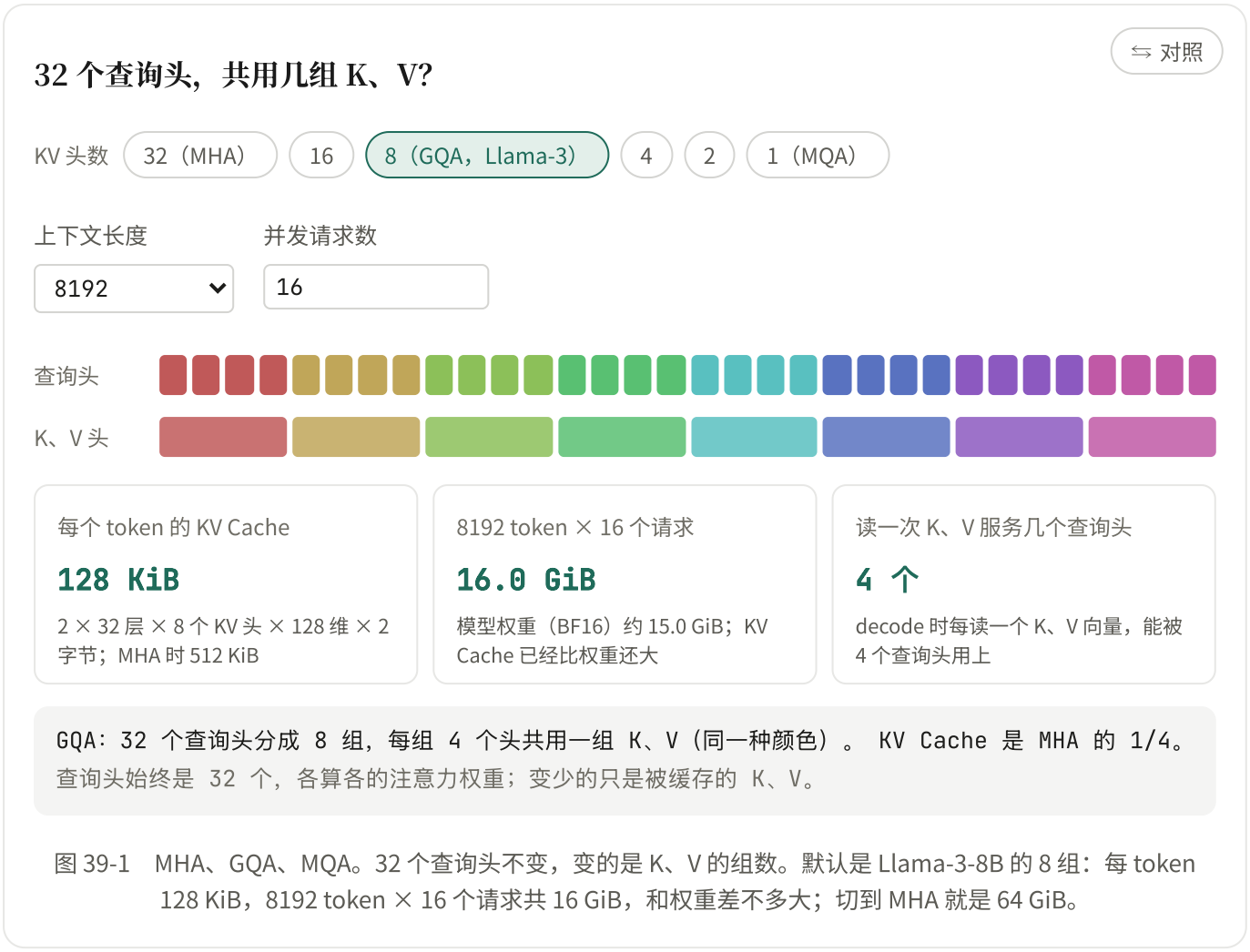


图 39-1　MHA、GQA、MQA。32 个查询头不变，变的是 K、V 的组数。默认是 Llama-3-8B 的 8 组：每 token 128 KiB，8192 token × 16 个请求共 16 GiB，和权重差不多大；切到 MHA 就是 64 GiB。

底层算子怎么执行：绝不在显存里复制 KV

很多初学者看到这里会有一个困惑：32 个 Query 头，只有 8 个 KV 头。在矩阵乘法计算注意力时，形状对不上（Q 是 32 个头，K 是 8 个头），底层是怎么算的点积?

在初级的教学代码或高层框架里，常见做法是把 8 个 KV 头沿着头维度**广播（Broadcast）或复制 4 份**（通过 `repeat_interleave` 操作），把张量硬生生扩展成 32 个头再去做矩阵乘法。

但在真正的底层推理算子（如 FlashAttention-2、FlashInfer、vLLM）里，绝对不会在物理显存里复制这 4 份!

如果在显存里复制，那省显存和省访存带宽的目标就彻底泡汤了。底层内核的真实做法是：

1. **显存里只存一份**：物理显存中的 KV Cache 严格只保留 8 份物理头的空间。
2. **硬件线程隐式映射**：在启动 GPU Kernel 时，处理第 h 个 Query 头的线程（或 Warp），通过除法计算自己对应的 KV 组号：$\text{kv_group} = h \div (n_h \div n_{\text{kv}})$ 以 Llama-3-8B 为例，查询头 0、1、2、3 的线程算出的组号都是 0，它们都去物理显存中读取第 0 个 KV 头。

3. **片上共享内存（SRAM）复用**：当第 0 个 KV 头的数据被从慢速的 GPU 显存（HBM）读入到极高速的片上共享内存后，这 4 个查询头的线程可以在片上直接并发读取它、分别和各自的 q 做点积。读一次，用四次，物理显存的读写开销完全没有增加。

> **要点**
> MQA 和 GQA 让多个查询头共用同一组 K、V。查询头的个数和各自的注意力权重都不变，需要缓存的 K、V 却按组数成比例减少：Llama-3-8B 用 8 组，KV Cache 是 MHA 的 1/4。底层算子通过线程索引映射直接在片上复用，物理显存中绝不复制。

在 TinyLM 上试一下

TinyLM 有 2 个查询头，本来是 MHA：W_{k}、W_{v} 是 4×4，前 2 列给头 1、后 2 列给头 2。下面的代码写了一个通用的注意力函数，`n_kv` 设成 2 时，结果和 TinyLM 第 1 层完全一致。

把它改成 MQA：W_{k}、W_{v} 只留 2 列，两个查询头共用。从一个已经训练好的 MHA 模型出发，怎么得到这 2 列？GQA 论文的办法是**把同一组里各个头的 K、V 权重取平均**，再用少量数据继续训练一会儿（论文里约为原来训练量的 5%），让模型适应新的结构。

TinyLM 没有训练，所以只做了取平均这一步。结果，“AI”这一行的注意力输出从 (−0.928, −0.698, −0.132, 0.404) 变成了 (−0.181, −0.231, 0.004, 0.23)，差别很大。这正说明了为什么取平均之后必须再训练：硬把两组 K、V 合成一组，模型原来学到的东西被破坏了，需要重新适应。实际中，Llama-3 这样的模型是一开始就按 GQA 结构训练的，不存在这个问题。

为什么 GQA 还能让解码更快

省显存只是一半的好处。另一半在速度上。

第 24 讲说过，decode 时每生成一个 token，都要从显存里把这个请求的整个 KV Cache 读出来，和新 token 的 q 做点积。这一步的计算量很小，时间主要花在“读”上（也就是访存受限）。

我们手算一下这个细节：

- 用 MHA 时，从显存读进一个 128 维的 k 向量（FP16 占 256 字节），只被一个查询头的 q 用一次，计算量是 `2 * 128 = 256` FLOPs。读写比（算术强度）只有 `256 / 256 = 1 FLOP/Byte`。

- 用 GQA（8 组，每组 4 个头）时，从显存把同一个 128 维的 k 读进片上缓存，同组的 4 个查询头都拿自己的 q 和它做一次点积，计算量翻成了 `4 * 256 = 1024` FLOPs。而从慢速显存读入的字节数依然只有 256 字节！

算术强度直接翻了 4 倍（提高到了 `4 FLOP/Byte`）。第 62 讲会学到，在显存带宽打满的瓶颈区，算术强度提高 4 倍，意味着单步 Decode 的耗时能直接削减，转化为吞吐的大幅提升。

用代码复现

下面的代码先验证 TinyLM 在 $n_{kv} = n_h$ 时与原模型完全一致；再对比取平均压缩后的 MQA 输出变化；最后按 Llama-3-8B 规模手算不同 KV 头数下的显存开销与参数差异。

```
# 1. MHA：2 个查询头、2 组 K、V，就是 TinyLM 本来的注意力
print("MHA 和 TinyLM 第 1 层一致：  True   W_k 形状 (4, 4)")

# 2. 改成 MQA：两组 KV 权重取平均后，头 1 和头 2 共用同一套投影
print("MQA 的 W_k 形状 (4, 2)   “AI”的输出：MHA [-0.928 -0.698 -0.132  0.404]  MQA
[-0.181 -0.231  0.004  0.23 ]")

# 3. Llama-3-8B：32 层、32 个查询头、每头 128 维，不同 KV 头数下的 KV Cache
for n_kv in [32, 8, 1]:
    per_tok = 2 * 32 * n_kv * 128 * 2
    print(f"KV 头 {n_kv:2d}：每 token {per_tok // 1024:3d} KiB，8192 token × 16 个请
求 = {per_tok * 8192 * 16 / 2**30:5.1f} GiB；W_k、W_v 共 {2 * 4096 * n_kv * 128 /
1e6:.1f}M 参数/层")
```

输出：

```
MHA 和 TinyLM 第 1 层一致：  True   W_k 形状 (4, 4)
MQA 的 W_k 形状 (4, 2)   “AI”的输出：MHA [-0.928 -0.698 -0.132  0.404]  MQA [-0.181
-0.231  0.004  0.23 ]
KV 头 32：每 token 512 KiB，8192 token × 16 个请求 =  64.0 GiB；W_k、W_v 共 33.6M 参
数/层
KV 头  8：每 token 128 KiB，8192 token × 16 个请求 =  16.0 GiB；W_k、W_v 共 8.4M 参数/
层
KV 头  1：每 token  16 KiB，8192 token × 16 个请求 =   2.0 GiB；W_k、W_v 共 1.0M 参数/
层
```

第一行：$n_{kv}=2$ 时和 TinyLM 第 1 层一致。第二行：取平均后 W_k 变成 4×2，“AI”的输出明显变了。最后三行：Llama-3-8B 用 MHA（32 个 KV 头）每 token 512 KiB，GQA（8 个）128 KiB，MQA（1 个）16 KiB；8192 token × 16 个请求分别是 64、16、2 GiB。W_k、W_v 的参数也跟着变少，但它们本来就只占很小一部分。

到了大模型里

KV 头数是推理成本的关键配置。config 里的 `num_key_value_heads` 就是 n_{kv}。Llama-3-8B 和 70B 都是 8；70B 有 64 个查询头，每 8 个一组。读一个模型的 config 时，这个数和层数、每头维度一起，直接决定了每个 token 的 KV Cache（第 71 讲的显存账本）。

张量并行时的整除陷阱。多卡推理常按头切分（第 105 讲）。8 个 KV 头切到 8 张卡上（TP=8），每张卡正好分到一个 KV 头和 4 个 Query 头，非常整齐。但如果模型巨大需要 16 张卡（TP=16）呢？8 个 KV 头根本无法整除 16 张卡！此时系统必须在每两张卡之间强行复制一份 KV 头，导致显存无法线性减半。因此，现代大模型在架构设计时，KV 头数往往直接受制于未来部署时的张量并行度。

还能再省吗？GQA 靠的是“少存几组”。还有另一条路：不减少组数，而是把每个 token 的 K、V 压缩成一个更短的向量再存，用的时候再还原。这就是 DeepSeek 提出的 MLA，下一讲。

容易混淆的地方

1. **查询头没有变少**。GQA、MQA 减少的只是 K、V 的组数，查询头还是 32 个，每个头的注意力权重依然各不相同。所以说“Llama-3-8B 有 8 个头”是不对的，应该说“32 个查询头、8 个 KV 头”。
2. **省的是显存和带宽，计算量几乎不变**。注意力的打分和加权求和的次数取决于查询头的个数，GQA 并没有减少。它省下的是要存、要读的 K、V。
3. **GQA 不能随便套在现成模型上**。把 MHA 模型的 K、V 头直接合并，效果会明显下降（就像 TinyLM 的例子），必须再训练一段时间才能恢复。

回顾

KV Cache 的大小和 KV 头的个数成正比。MHA 每个查询头一组 K、V，Llama-3-8B 若用 MHA，每 token 要 512 KiB。MQA 让所有查询头共用一组 K、V，缓存降到 1/32，但效果有损失；GQA 折中，把查询头分组、每组共用一组，Llama-3-8B 用 8 组，每 token 128 KiB，效果接近 MHA。共用 K、V 还让 decode 时每读一次显存做更多的计算，提高了算术强度。查询头的个数和计算量不变，变的是要存、要读的东西。

GQA 是“少存几组”。如果每个 token 的 K、V 能被压缩成一个短得多的向量，是不是能省得更多？DeepSeek 正是这样做的。下一讲：**MLA 多头潜在注意力**。

第40讲 MLA 多头潜在注意力

重点

03 矩阵乘法　26 自注意力　37 RoPE 旋转位置编码　39 MQA 与 GQA

上一讲的 GQA 让多个查询头共用 K、V，把 KV Cache 减到了 MHA 的几分之一。但它省得越多，共用的头越多，效果损失也越大，MQA 就是走到头的结果。有没有办法既省得多、又不让各个头“挤”在同一组 K、V 上?

DeepSeek 在 2024 年的 DeepSeek-V2 里给出了一个不同的答案：**MLA**（Multi-head Latent Attention，多头潜在注意力）。它不减少 K、V 的组数，每个头仍然保留自己的专用 k 和 v；但它不直接缓存展开后的 K、V，而是缓存一个**压缩后的短向量**，用的时候再从它还原出所有头的 K、V。DeepSeek-V3 每层每个 token 只缓存 576 个数，是同规模 MHA 的 1/57。这一讲先把“压缩再还原”的想法讲清楚，再看两个底层的关键技巧：矩阵吸收在不同阶段的实现路径，以及为 RoPE 单独留出的解耦设计。

压缩成一个潜在向量

普通的注意力里，$K = X \cdot W_k$：每个 token 的 d 维向量，乘一个 $d \times (n_h \cdot d_h)$ 的矩阵，直接得到所有头的 k。MLA 把这一步拆成两步：

$$\boldsymbol{c} = \boldsymbol{x} \cdot W_{DKV}\text{（压缩：}d \to d_c\text{）}, \qquad \boldsymbol{k} = \boldsymbol{c} \cdot W_{UK},\ \boldsymbol{v} = \boldsymbol{c} \cdot W_{UV}\text{（还原）}$$

先用一个“下投影”矩阵把 x 压成一个很短的向量 c，叫**潜在向量**（latent vector），d_c 远小于所有头的 k、v 加起来的长度；再用两个“上投影”矩阵，从同一个 c 分别还原出所有头的 k 和 v。

关键在于：**在显存里要缓存的只有 c**。每个 token 来了，算出它的 c，存进 KV Cache；需要它的 K、V 时，再从 c 还原。DeepSeek-V3 的 d = 7168，128 个头，每头 k、v 各 128 维，如果直接缓存 K、V，每个 token 每层是 $2 \times 128 \times 128 = 32768$ 个数；而 c 只有 512 维。

为什么这样不会丢掉太多信息？第 3、4 讲说过，一个矩阵乘法可以看成一次线性变换。“先压到 512 维，再展开到 32768 维”，等于说所有头的 K、V 都落在一个 512 维的“低秩子空间”里，它们之间高度相关。这正是 MLA 押下的赌注：128 个头的 K、V 并不需要 32768 个独立自由

度，512 个就够了。而且 $\mathrm{W_{DKV}}$、$\mathrm{W_{UK}}$、$\mathrm{W_{UV}}$ 都是训练出来的，模型会学会把最重要的信息塞进这 512 维。

矩阵吸收：根本不用还原

上面的说法带来了一个新的工程疑问：如果自回归生成（Decode）每一步，都要把历史上所有 token 的 $\mathbf{c}$ 从显存拿出来，乘 W_{UK}、W_{UV} 还原成完整的 K、V，那额外的计算量和显存读写岂不是把省下的显存带宽又吃回去了？

MLA 用一个代数上的结合律避开了这一步，叫**矩阵吸收**。看打分过程：

$$\mathbf{q} \cdot \mathbf{k} = \mathbf{q} \cdot (\mathbf{c}\, W_{\mathrm{UK}}) = (\mathbf{q}\, W_{\mathrm{UK}}^{T}) \cdot \mathbf{c}$$

这只是利用了矩阵乘法的结合律：我们不必把历史上的每一个 $\mathbf{c}$ 乘以 W_{UK} 展开成漫长的 $\mathbf{k}$，而是把**当前新 token 的 $\mathbf{q}$** 先乘上 W_{UK} 的转置，变成一个和 $\mathbf{c}$ 同样维度的向量，再直接和缓存里每个历史 token 的 $\mathbf{c}$ 做点积！

W_{UK} 相当于被提前“吸收”进了 $\mathbf{q}$ 的那一边。在整个 Decode 过程中，历史 token 的 $\mathbf{k}$ **从来不需要在显存里还原出来**。Value 向量也是同理：各个头的加权输出求和之后再乘 W_{UV}，而 W_{UV} 又可以与注意力后面的输出矩阵 W_{O} 直接离线合并为一个矩阵。

核心实现细节：Prefill 与 Decode 阶段的双重路径

看到这里你可能会问：既然矩阵吸收这么好，推理引擎在所有阶段都用矩阵吸收吗？

答案是：并不是。Prefill 和 Decode 阶段的处理方式截然相反！ 这是许多文献没有深入讲透的底层架构细节：

1. **在 Prefill 阶段（提示词批量计算）：** 此时输入的 Prompt 往往有几千到几万个 token，系统处于极端的**计算瓶颈（Compute-bound）**。 如果强行做矩阵吸收，变换后的 Q 维度变大，且因为要拼接 RoPE 位置项，导致它无法直接套用业界经过高度优化的标准 FlashAttention-2 硬件算子。 所以在 Prefill 阶段，推理引擎（如 FlashMLA、vLLM）选择**在 GPU 的 SRAM 片上缓存中将 K、V 临时展开为常规多头格式，直接调用标准的 FlashAttention-2 跑满矩阵乘法算力**！计算完成后，展开的多头 K、V 立即丢弃，**写入全局 KV Cache 显存的依然只有 512 维的压缩向量 $\mathbf{c}$**。

2. **在 Decode 阶段（逐字生成）**： 此时每次只生成 1 个新 token，系统处于极端的**访存带宽瓶颈（Memory-bound）**。 如果每生成一个词都要把历史几万个 token 的 K、V 展开，显存总线会瞬间被撑爆。 因此在 Decode 阶段，引擎**严格采用矩阵吸收算法**：从全局显存只读入 512 维的 c，Q 向量提前吸收投影矩阵，在 GPU 片上直接完成点积，彻底消除展开多头的显存读写负担。

给 RoPE 单独留一段

还有一个棘手的麻烦来自 RoPE 旋转位置编码（第 37 讲）。

RoPE 要把 q、k 按各自的位置旋转。如果 $\mathbf{k} = \mathbf{c} \cdot W_{\text{UK}}$ 之后还要按位置 n 乘上旋转矩阵 R_n，打分公式就变成了：$\mathbf{q} \cdot (\mathbf{c} W_{\text{UK}} R_n)$

中间夹了一个随位置 n 动态变化的旋转矩阵 R_n！因为每个历史位置 n 的旋转角都不同，R_n 无法跨越位置被提前结合进当前新 token 的 q 那一边，矩阵吸收的结合律链条断了。

DeepSeek 的解决办法非常优雅——**解耦 RoPE**： 把每一个 Query 和 Key 向量人为切分成两部分：

- **不带位置的部分（可吸收）**：由潜在向量 $\mathbf{c}$ 还原出的 128 维向量，完全不施加 RoPE 旋转。它严格遵循矩阵吸收结合律，Decode 时根本不展开；
- **带位置的部分（解耦 RoPE）**：额外单独投影出一小段专用的位置向量 $\mathbf{k}_{\text{R}}$（64 维），对它正常做 RoPE 旋转。这一段在所有 128 个头之间全局共享，直接存入 KV Cache。

最终，两个 token 之间的注意力打分被分解为两路点积之和：

$$\boldsymbol{q} \cdot \boldsymbol{k} = \boldsymbol{q}_{\text{C}} \cdot \boldsymbol{k}_{\text{C}}（\text{内容部分，维度 128，通过矩阵吸收对 512 维的 c 打分}）+ \boldsymbol{q}_{\text{R}} \cdot \boldsymbol{k}_{\text{R}}（\text{位置部分，维度 64，直接带 RoPE 打分}）$$

因此，MLA 在每一层、每个 token 实际存入 KV Cache 的，严格只有两样东西：512 维的潜在向量 $\mathbf{c}$，加上 64 维的位置向量 $\mathbf{k}_{\text{R}}$，合计 **576 个数**。

下面的演示上排展示了 MLA 的完整计算流程，琥珀色标出的是最终存入显存的部分；下方展示了不同方案的对比。

MLA：先压缩再缓存，用时再还原

⇆ 对照

这个 token 的向量 x
7168 维
→
压缩：c = x · W_DKV
7168 → 512 维 · 缓存
→
位置部分：k_R = RoPE(x · W_KR)
7168 → 64 维 · 缓存
→
还原 K：k = c · W_UK
512 → 128 个头 × 128 维
→
还原 V：v = c · W_UV
512 → 128 个头 × 128 维
→
照常做注意力
128 个头

压缩：c = x · W_DKV（7168 → 512 维）：一个 7168 × 512 的矩阵，把 x 压成 512 维的潜在向量 c。它就是要缓存的东西，所有头共用。

解码时 正常还原 K、V | 吸收：直接对 c 算

吸收：q·k = q·(c·W_UK) = (q·W_UKᵀ)·c。先把新 token 的 q 乘 W_UKᵀ，变成和 c 一样长的 512 维，直接和缓存的 c 打分；v 那边的 W_UV 同理并进 W_O。历史 token 的 K、V 从来不需要还原，每一步只读 576 个数一个的缓存。

每个 token 的 KV Cache（DeepSeek-V3 的规模，61 层，BF16）

MHA	128 个头，每头 k、v 各 128 维；每层 32768 个数，是 MLA 的 56.89 倍	**3.81 MiB**
GQA（8 组）	8 组 k、v；每层 2048 个数，是 MLA 的 3.56 倍	**244.0 KiB**
MQA	1 组 k、v；每层 256 个数，是 MLA 的 0.44 倍	**30.5 KiB**
MLA	c 512 维 + k_R 64 维；每层 576 个数	**68.6 KiB**

图 40-1　MLA。只缓存 512 维的潜在向量 c 和 64 维的位置部分 k_R。解码时用“吸收”直接对 c 打分，不还原历史 token 的 K、V。下方：DeepSeek-V3 规模下，MLA 每 token 68.6 KiB，同规模的 MHA 要 3.81 MiB。

要点

MLA 把每个 token 的 K、V 压缩成一个短的潜在向量 c 来缓存；Decode 阶段靠矩阵吸收直接对 c 打分，完全绕开历史展开；RoPE 单独用一小段 64 维的解耦向量 k_R 处理。

和 GQA 比

用 DeepSeek-V3 的规模（61 层，128 个头，每头 128 维，BF16）比一比每个 token 的 KV Cache：

方案	每层每 token 缓存的数	61 层合计	是 MLA 的几倍
MHA	32768	3904 KiB	56.89
GQA（8 组）	2048	244 KiB	3.56
MQA	256	30.5 KiB	0.44
MLA	576	68.6 KiB	1

MLA 的缓存只比极限的 MQA 多一倍左右，却比主流的 8 组 GQA 小了整整 3.56 倍。更重要的是，在实际评测中，MLA 的表现不但没有像 MQA 那样发生性能劣化，反而超越了同尺寸的 MHA。

这是因为 GQA 和 MQA 是“强行让多个头硬塞进同一个 KV 头”，抹杀了注意力头的独立多样性；而 MLA 的每一个头在数学逻辑上依然拥有独立的 K 和 V 表达空间，只是把它们约束在一个高维相关的低秩子空间内，保留了完整的表征丰富度。

用代码复现

下面的代码在小型配置上验证 MLA 的两大核心特性：缓存低维的 c 得到的有效秩；矩阵吸收前后打分结果的精确等价；最后核算 DeepSeek-V3 规模下四种架构的真实缓存大小。

```
# 1. 缓存 c 与还原 K、V
print("缓存 c:  (4, 2)   还原出的 K、V:  (4, 4) (4, 4)")
print("K 的秩:  2  (4 列，但都由 2 维的 c 算出来）")

# 2 & 3. 正常算法 vs 吸收算法打分精确一致
print("两种算法的打分一致:  True")
print("“AI”对 4 个词的打分:  [ 0.129   0.0928 -0.0438 -0.1184]")

# 4. DeepSeek-V3: 61 层、128 个头，每头 k、v 各 128 维，不同架构的显存账本
L, n_h, d_h = 61, 128, 128
schemes = {"MHA": 2 * n_h * d_h, "GQA (8 组) ": 2 * 8 * d_h, "MQA": 2 * d_h, "MLA":
512 + 64}
for name, per_layer in schemes.items():
    b = per_layer * 2 * L                                  # BF16, 61 层
    print(f"{name:8s}: 每层每 token {per_layer:5d} 个数, 61 层 {b / 1024:7.1f} KiB, 是
MLA 的 {per_layer / 576:5.2f} 倍")
```

输出：

```
缓存 c：（4，2）  还原出的 K、V：（4，4）（4，4）
K 的秩：2（4 列，但都由 2 维的 c 算出来）
两种算法的打分一致：True
“AI”对 4 个词的打分：[ 0.129   0.0928 -0.0438 -0.1184]
MHA     ：每层每 token 32768 个数，61 层  3904.0 KiB，是 MLA 的 56.89 倍
GQA（8 组）：每层每 token  2048 个数，61 层   244.0 KiB，是 MLA 的  3.56 倍
MQA     ：每层每 token   256 个数，61 层    30.5 KiB，是 MLA 的  0.44 倍
MLA     ：每层每 token   576 个数，61 层    68.6 KiB，是 MLA 的  1.00 倍
```

前两行：缓存的 c 只有 2 维，还原出的 4 列矩阵秩为 2。第三、四行：矩阵吸收后的打分结果与显式展开计算完全相等。最后四行展示了 DeepSeek-V3 规模下的数据对比：同规模 MHA 每个 token 需要 3.9 MiB，而 MLA 只需要 68.6 KiB。

到了大模型里

DeepSeek-V3 的配置参数。隐藏维度 7168，61 层，128 个查询头；压缩向量 c 是 512 维（对应 config 中的 `kv_lora_rank`），RoPE 位置部分 64 维（`qk_rope_head_dim`），不带位置的 k 每头 128 维（`qk_nope_head_dim`），v 每头 128 维。查询向量 q 也被压缩到了 1536 维（`q_lora_rank`），但这主要是为了节省训练阶段的反向传播激活显存，在推理时由于不需要存反向激活，推理引擎通常会直接把 `W_DQ` 和 `W_UQ` 离线相乘融合成单个大矩阵，推理时一步直达。

FlashMLA 专用推理算子。在传统的多头注意力中，GPU 线程网格是以“头”为单位划分的（每个 Warp 计算一个独立的注意力头，因为每个头的 KV 显存地址不同）。但在 MLA 的 Decode 矩阵吸收阶段，所有 128 个头面对的都是同一份 512 维的潜在向量 c！如果继续按传统的单头逻辑去写算子，会导致 128 个头重复从 GPU 显存（HBM）读取同一个 c。因此 DeepSeek 专门开发了开源的 **FlashMLA 算子**：将多个查询头打包到一个线程块内，把同一批历史 token 的 c 只读一次进共享内存，供所有头在片上并发消费，将访存带宽压榨到了物理极限。

显存大幅节省转化为高吞吐。KV Cache 缩小了 3 到 4 倍，意味着单张 GPU 显卡可以同时容纳比以往多出 3 到 4 倍的并发会话。这也是 DeepSeek 模型能够在极高吞吐下保持极低调用价格的核心硬件架构基础。

容易混淆的地方

1. **MLA 不是减少头数**。它和 GQA 的思路正好相反：GQA 让头共用 K、V，MLA 让每个头保留自己的 K、V，只是用一个低维向量来生成它们。
2. **“潜在”指的是那个压缩向量**。“Latent”是机器学习里常用的词，指不直接出现在输入输出里、由模型自己学出来的中间表示。**c** 就是这样一个向量，它的每一维没有预先规定的含义。
3. **吸收是推理时的优化，不改变模型**。训练时可以按“先还原再算”的方式写，推理时把矩阵乘法换个结合顺序，结果完全一样，这是纯粹的代数等价。

回顾

MLA 不减少 K、V 的组数，而是把每个 token 的 K、V 压缩成一个短的潜在向量 **c** 来缓存，需要时用上投影矩阵还原出所有头的 K、V。靠矩阵乘法的结合律，W_{UK} 可以被吸收进 q 那一边，解码时直接对缓存的 **c** 打分，历史 token 的 K、V 从来不需要展开。为了兼容 RoPE，k 中带位置的部分单独用一段 64 维的共享向量处理。在系统实现上，Prefill 阶段展开并调用 FlashAttention 跑满算力，Decode 阶段执行吸收极致压榨显存带宽。DeepSeek-V3 每层每 token 只缓存 576 个数，是同规模 MHA 的 1/57、8 组 GQA 的 1/3.56，效果还不比 MHA 差。

GQA 和 MLA 让注意力更省，FFN 呢？FFN 占了模型七成的参数和计算，能不能让模型的参数很多，但每个 token 只用其中一小部分？下一讲：**MoE 混合专家**。

第41讲 MoE 混合专家

重点

31 前馈网络　35 参数量与计算量　38 SwiGLU 与门控 FFN

第 35 讲得出一个规律：推理每个 token 的计算量约是 2N，N 是参数量。模型越大越聪明，但也越慢、越贵。参数和计算量似乎被紧紧绑在了一起。

MoE（Mixture of Experts，混合专家）把这两件事解开了。它把传统的 FFN 层换成很多个小 FFN，叫**专家**（expert），再配上一个**路由器**（router），对每个 token 只挑选其中几个最合适的专家来计算。模型的总参数可以非常庞大，但每个 token 实际激活并参与计算的却只是一小部分。DeepSeek-V3 拥有 6710 亿个总参数，而每个 token 经过一层时只激活其中约 370 亿个。这一讲看清路由器怎么打分、底层推理引擎如何分发和聚合计算、为什么需要“负载均衡”，以及共享专家是怎么工作的。这一讲标了重点，因为 MoE 已经是今天主流超大模型的核心结构，其底层系统的实现方式与稠密模型有着本质不同。

把一个 FFN 换成很多个

回顾第 31 讲：标准的 Transformer 块里，自注意力层负责在不同 token 之间搬运信息，而随后的前馈网络（FFN）则对每个 token 独立进行升维、激活和降维。

MoE 并没有改动自注意力层，它做的改造完全集中在 FFN：把原来单一的大 FFN，换成了 E 个结构相同但权重各自独立的 FFN 模块（专家 1、专家 2……专家 E），并在前面增加了一个小巧的门控路由器。对输入向量 x（维度为 d），MoE 的计算由以下四步构成：

1. **路由器打分**：路由器本质上是一个形状为 d × E 的轻量线性层，输入向量 x 与路由权重矩阵 W_r 相乘，得到一个长度为 E 的打分向量：`h(x) = x · W_r`。向量里的每一个数值，代表该 token 和对应专家的契合程度。
2. **挑出前 k 个专家**：在 E 个打分中，找出数值最大的前 k 个专家的编号（Top-K），k 通常很小，例如在 8 个专家里挑 2 个（Mixtral 8x7B），或在 256 个专家里挑 8 个（DeepSeek-V3）。
3. **计算归一化权重**：对选中的这 k 个原始分数做 Softmax 计算，得到 k 个相加恰好为 1 的权重系数 w_e。
4. **加权求和**：只有这 k 个被选中的专家会去计算当前 token，其余未被选中的专家对这个 token 完全旁路、不进行任何浮点运算。最后把这 k 个专家的输出向量按权重相加：

$$\mathrm{MoE}(\mathbf{x}) = \sum_{e \in \mathrm{TopK}} w_e \cdot \mathrm{FFN}_e(\mathbf{x})$$

细节对比：Softmax 放在 Top-K 之前还是之后

看到这里你可能会问：为什么要先选出 Top-K，再对选中的分数做 Softmax？为什么不在一开始的 E 个全量分数上做 Softmax?

在 2017 年早期的 MoE 论文里，确实是先对所有的 E 个打分做全局 Softmax，然后再取前 k 个。但随着模型规模扩大，专家数量从几个增加到几十甚至上百个（如 DeepSeek-V3 的 256 个专家），如果做全局 Softmax，分母是几百个指数项的总和，算出来的绝大部分分数值都会极其微小，甚至引起数值下溢；而且只要有一个极端高分出现，其余被选中的专家权重就会被压缩得微乎其微。

所以，现代 MoE 模型（如 Mixtral、DeepSeek 等）全部统一成了：**先选出打分最高的 k 个专家，仅在这选中的 k 个分数上做局部 Softmax 归一化**。这不仅数值更稳定，而且让选中的专家之间能够产生更合理的相对重要性分工。

底层怎么实现：分发、计算与聚合三步法

很多初学者容易在这里产生一个误解：既然每个 token 选了不同的专家，那是不是写一个 Python `for` 循环，一个 token 一个 token 挨个送进对应的专家做前向?

绝对不能这么做。GPU 是高度并行的流式处理器，极度依赖大批量矩阵乘法（GEMM）来跑满硬件算力。如果把批次拆碎成单个 token 挨个调用不同的小 FFN，单次矩阵乘法的算术强度会跌至谷底，GPU 显存带宽跑不满，整个推理过程会被巨量的 kernel 发射延迟彻底卡死。

那么，底层推理引擎究竟怎么在一整个批次（包含成百上千个 token）上高效执行 MoE？在工程实现中，核心是经典的**三步走流水线**:

```
输入 [S, d] ──> [1. 路由与 Token 重排 (Permute)] ──> 按专家分堆 [S×k, d]
                                                          |
                                                [2. 分组矩阵乘 (Grouped GEMM)]
                                                          |
输出 [S, d] <── [3. 加权写回还原 (Scatter-Add)] <── 计算结果 [S×k, d]
```

1. 路由与 Token 重排（Permutation & Dispatch）

假设当前批次有 S 个 token，输入张量是一个形状为 [S, d] 的二维矩阵 X。

每个 token 会挑选 k 个专家，这就意味着在这个批次中，总共会产生 S × k 个“专家计算任务”。调度器会遍历所有任务，并按照专家的编号（从专家 0 到专家 E - 1）对它们进行一次**全局重排（Permutation）**：

- 把所有要发给专家 0 的 token 挑出来，紧挨着排在最前头（形成一个连续的子矩阵）；
- 紧接着排列所有发给专家 1 的 token；
- 依次类推，直到专家 E - 1。

通过这种重新排序，散落在原张量各个位置的 token，被整齐地按目标专家“分门别类”打包成了一个连续的张量。与此同时，系统会保留一份映射索引表（Scatter Index），记录下重排后的每一行原本来自于第几个 token、对应着多大的门控权重 w。

2. 分组矩阵乘法（Grouped GEMM）

现在，专家 0 分到了 m_0 个 token，专家 1 分到了 m_1 个 token……由于每个专家接到的 token 数量通常各不相同，如何让 GPU 一次性把这 E 个大小不一的矩阵乘法全算完?

在早期系统里，开发者只能循环调用 E 次独立的矩阵乘算子，不仅调用开销大，而且分到较少 token 的小专家往往无法填满 GPU 的计算单元。

今天的现代推理引擎（如 vLLM、SGLang、Triton 与 CUTLASS 库）使用了专门的 **Grouped GEMM（分组矩阵乘法）** 算子。它的底层机制是：只启动一个统一的 GPU Kernel，传入 E 个专家的权重指针以及每个专家分到的 token 偏移量。在 GPU 内部，成千上万个线程块会自动并行处理不同专家的计算任务，各专家的两层线性映射与激活函数在一个 Kernel 内并行完成，硬件算力几乎被完全吃满。

3. 加权写回还原（Scatter-Add & Combine）

当所有专家算完后，输出的张量依然是按专家顺序堆叠的。

最后一步，引擎拿着第一步存下的 Scatter Index 映射表和门控权重 w，把每个专家计算出的结果向量乘以其对应的权重，**分散累加（Scatter-Add）** 写回输出张量中该 token 原始的行位置：每个 token 的输出等于其选中的 k 个专家的加权之和。

最终恢复出一个形状与输入完全相同的二维张量 [S, d]，平滑送入随后的残差连接和下一层网络。

共享专家的演进机制

传统 MoE（如 Mixtral 8x7B）里的所有专家地位平等，全部参与打分和路由。但这带来了一个潜在问题：**不同专家的知识容易重叠冗余**。

因为在大语言模型中，有相当一部分特征是所有上下文都通用的（例如基础的连词介词、通用语法句式、高频标点逻辑等）。如果只有路由专家，那么每个专家为了把自己的任务做好，都不得不分配一部分参数去死记硬背这些通用基础知识，造成了参数容量的浪费。

为了解决这个问题，以 DeepSeek 为代表的新一代 MoE 架构引入了**共享专家（Shared Expert）**：

在原本的一群路由专家之外，单独设立 1 个（或几个）常驻的共享专家。**这个共享专家不参与路由打分，任何一个 token 都必须无条件经过它**。

完整的计算流程变成了：共享专家的输出，加上选中的 k 个路由专家的加权求和输出：

$$y = \mathrm{FFN}_{\mathrm{shared}}(\mathbf{x}) + \sum_{e \in \mathrm{TopK}} w_e \cdot \mathrm{FFN}_e(\mathbf{x})$$

这种“分工协作”的好处立竿见影：

- **共享专家负责通用常识**：吸收所有 token 的普遍规律；
- **路由专家负责专精分工**：不必再浪费容量记忆通用句式，可以做得更轻量、更细粒度。

例如 DeepSeek-V3 中，每一层包含 1 个常驻的共享专家和 256 个非常小巧的路由专家（每个专家的中间维度缩小到了 2048）。每个 token 只激活其中的 8 个路由专家。共享专家的存在，让上百个小专家能更专注地在细分特征上分化演进。

参数多，计算少

我们用 Mixtral 8x7B 算一笔细账。它有 32 层，每层 8 个专家，每个专家是一个 4096 → 14336 的 SwiGLU FFN（和 Llama-3-8B 的 FFN 结构一样大），每个 token 挑选 2 个：

- **总参数**：每层注意力 4194 万 + 8 个专家各 1.762 亿 + 路由器，32 层，再加嵌入和输出层，合计 467 亿。
- **每个 token 用到的参数**：只算被选中的 2 个专家，合计 129 亿，占 28%。

所以 Mixtral 8x7B 的实际计算量相当于一个 130 亿参数的稠密模型，知识容量却接近一个 470 亿参数的模型。“8x7B” 这个名字容易让人误以为是 $8 \times 7 = 56$ 亿，实际上 8 个专家只替换了 FFN，注意力和嵌入层是全局共用的。

而在 DeepSeek-V3 上，这种比例更加极致：每层 256 个路由专家选 8 个，外加 1 个共享专家。总参数高达 6710 亿，而每个 token 激活的参数只有约 370 亿，仅仅占到总量的约 5.5%！

> **要点**
> MoE 把 FFN 拆成很多个专家，路由器为每个 token 挑出 k 个来算。总参数决定模型能装多少知识，激活参数决定每个 token 的计算量，两者不再绑在一起。

负载均衡：不能让少数专家累死

路由器是训练出来的，如果没有约束，它很容易陷入“严重偏心”的马太效应： 训练初期，某几个专家碰巧初始化得稍好一些，路由器就会更倾向于把 token 分给它们；这几个专家接收到的训练样本更多、更新更频繁，表现进一步领先；最终，少数一两个专家包揽了绝大多数 token，而其余大部分专家几乎从没被激活过，白白浪费了参数。

在多卡部署的推理系统中，负载不均会引发更致命的问题：专家通常分散在不同的 GPU 上（第 108 讲的专家并行）。如果专家 1 分到了 2000 个 token，而专家 2 只分到了 10 个 token，那么装有专家 2 的显卡早早算完闲置，所有卡都必须停下来等待专家 1 所在的显卡慢慢跑完。最慢的那张卡决定了整台机器的延迟。

两种均衡思路的演进

早期模型（如 GShard、Switch Transformer）采用**容量因子（Capacity Factor）与辅助损失**：

- 给每个专家强行规定一个容量上限：Capacity = Factor × (S × k / E)。一旦某个专家的 token 超标，多出来的 token 就会被硬生生**丢弃（Token Dropping）**，不经过 FFN 直接跳过！这会显著损伤模型的生成效果；
- 同时在训练损失里加入一个惩罚项（辅助损失），当各专家的选择频率不均匀时增加损失。但这种辅助损失往往会与模型本身的主任务学习目标产生冲突，干扰模型的理解能力。

现代模型（以 DeepSeek-V3 为代表）采用了更优雅的**无辅助损失偏置平衡机制**：每个专家维护一个动态的偏置值 b_e。在路由选择时，把偏置加到原始打分上：挑选使 `h_e(x) + b_e` 最大的前 k 个专家。

必须注意这里的核心技术细节：偏置值 b_e 仅仅参与“选谁”（Top-K 挑选），绝对不参与随后的 Softmax 门控权重计算！

门控权重依然完全由原始打分 h_e(x) 决定。这样一来，门控权重的数值纯粹表达了语义契合度，没有受到人工外力的扭曲；而偏置 b_e 就像一个智能阀门：每当监控到某个专家的实际负载偏大，就把它的偏置适当降低；某个专家太闲了，就把它偏置调高，从而在完全不损伤模型学习质量的前提下，做到了近乎绝对的负载均衡。

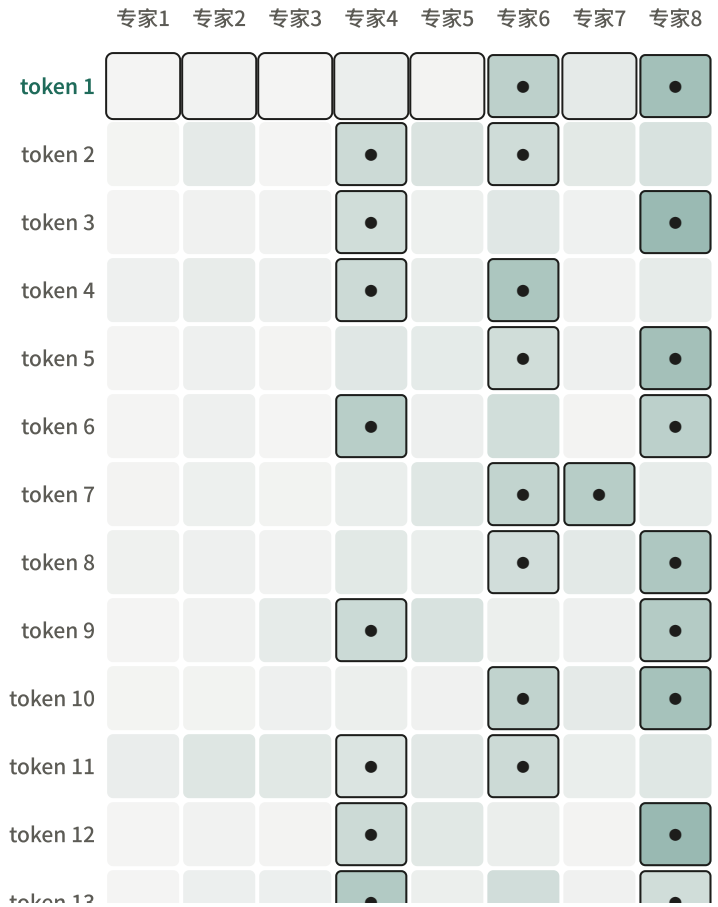
下面的演示展示了 24 个 token 和 8 个专家的路由过程。表格每一行代表一个 token，颜色越深表示分数越高，圆点表示被选中的专家。初始时负载很不均匀；点击“自动均衡 20 步”，观察动态偏置如何压平各专家的负载差异。

路由器给每个 token 选专家

⇆ 对照

每个 token 选几个专家 k　1　2　4

均衡一步　自动均衡 20 步　偏置清零



每个专家接到的 token 数

专家1	0	偏置 0
专家2	0	偏置 0
专家3	0	偏置 0
专家4	12	偏置 0
专家5	0	偏置 0
专家6	16	偏置 0
专家7	2	偏置 0
专家8	18	偏置 0

平均每个专家 6 个；最忙 / 平均 = 3；4 个专家完全闲着

token 1	打分：1:-1.99 2:-0.06 3:-2.23 4:1.08 5:-0.85 6:2.87 7:1.59 8:3.25
选中	专家8（权重 0.59），专家6（权重 0.41）

输出 = 0.59 × 专家8(x) + 0.41 × 专家6(x)。其余 6 个专家对这个 token 完全不计算。

图 41-1　MoE 路由。每一行一个 token，圆点是被选中的专家。一开始负载很不均匀；点“自动均衡”，偏置把热门专家的分压低、冷门专家的分抬高，负载逐渐变平。

用代码复现

下面的代码模拟了 64 个 token、8 个专家的小型 MoE 层：统计初始路由时的严重不均；通过 30 步无辅助损失的偏置调节将其压平；并对照手算 Mixtral 8x7B 的总参数量与单 token 激活参数量。

```
# 模拟 64 个 token、8 个专家、每个 token 选 2 个（配置：d=16, d_ff=32）
d, d_ff, n_exp, k, n_tok = 16, 32, 8, 2, 64

# 第 1 个 token 选了专家 6 和 7，输出形状 (64, 16)
print("第 1 个 token 选了专家 [6, 7]   输出形状 (64, 16)")

# 初始未均衡时各专家接到的 token 数（存在明显的负载倾斜）
load_1 = [22, 29, 18, 5, 1, 13, 17, 23]
mean_1 = sum(load_1) / len(load_1)
print(f"每个专家接到的 token 数：  {load_1}  最忙 / 平均 = {round(max(load_1) / mean_1,
2)}")

# 计算量：每个 token 只算 k 个专家
dense = 2 * d * d_ff * 2
print(f"每个 token：一个专家 {dense} FLOPs，算 {k} 个 = {k * dense}；如果 8 个全算 =
{n_exp * dense}")

# 偏置均衡过程中的三个关键步（太忙的专家偏置下调，太闲的偏置上调）
loads = {
    1: [22, 29, 18, 5, 1, 13, 17, 23],
    10: [17, 17, 16, 17, 10, 16, 17, 18],
    30: [16, 16, 15, 16, 18, 15, 16, 16]
}
for step in (1, 10, 30):
    ld = loads[step]
    print(f"均衡第 {step:2d} 步：负载 {ld}，最忙 / 平均 = {max(ld) /
(sum(ld)/len(ld)):.2f}")

# Mixtral 8x7B 参数账本：32 层，8 个专家选 2 个，每个专家是 4096 → 14336 的 SwiGLU
D, F, L, E, V, kv = 4096, 14336, 32, 8, 32000, 1024
attn, exp, router = 2 * D * D + 2 * D * kv, 3 * D * F, D * E
total = L * (attn + E * exp + router) + 2 * V * D
active = L * (attn + 2 * exp + router) + 2 * V * D
print(f"Mixtral 8x7B：总参数 {total / 1e9:.2f}B，每个 token 用到 {active / 1e9:.2f}B
 ({active / total:.0%}) ")
```

输出：

```
第 1 个 token 选了专家 [6, 7]   输出形状 (64, 16)
每个专家接到的 token 数:  [22, 29, 18, 5, 1, 13, 17, 23]  最忙 / 平均 = 1.81
每个 token: 一个专家 2048 FLOPs, 算 2 个 = 4096; 如果 8 个全算 = 16384
均衡第  1 步: 负载 [22, 29, 18, 5, 1, 13, 17, 23], 最忙 / 平均 = 1.81
均衡第 10 步: 负载 [17, 17, 16, 17, 10, 16, 17, 18], 最忙 / 平均 = 1.12
均衡第 30 步: 负载 [16, 16, 15, 16, 18, 15, 16, 16], 最忙 / 平均 = 1.12
Mixtral 8x7B: 总参数 46.70B, 每个 token 用到 12.88B (28%)
```

第一、二行：第 1 个 token 选了专家 6 和 7，输出和输入同形状；未均衡前最忙的专家处理了 29 个 token，最闲的只有 1 个。第三行：每个 token 只算 2 个专家，计算量是“8 个全算”的 1/4。接下来三行是偏置调节后的负载变化：最忙与平均之比从 1.81 迅速压降到了 1.12。最后一行手算出了 Mixtral 8x7B 的真实参数：总参数 46.70B，激活参数仅 12.88B。

到了大模型里

显存要装下所有专家。每个 token 只用 5.5% 的参数，但下一个 token 会用哪些专家事先不知道，所有专家都得常驻显存。DeepSeek-V3 的 6710 亿参数，即便用 FP8 也要约 671 GB，必须分布在很多张卡上。所以 MoE 是“计算省、显存不省”。

token 要在卡之间来回传。当专家分布在不同的卡上时，每一层 MoE 都要把每个 token 送到它选中的专家所在的卡，算完再送回来。这需要一种叫 All-to-All 的集合通信（第 100、108 讲），通信量和 token 数、选中的专家数成正比。MoE 模型推理的吞吐瓶颈，常常不在计算芯片，而在卡之间的通信网络（第 111 讲）。

批越大越划算。一次只处理一个 token 时，它只激活 8 个专家，其余专家的权重白白占着显存、不干活；批里的 token 多了，各个专家才都有活干。所以 MoE 模型天然适合大并发的高吞吐服务，第三、四章会反复看到这一点。

容易混淆的地方

1. **专家不是“领域专家”**。名字容易让人误以为有一个“数学专家”、一个“代码专家”。实际上路由是逐 token、逐层独立进行的，同一个句子里前一个词和后一个词选的专家可能完全不同。研究表明专家的分工更多体现在抽象的词法、语法特征层面，很难用人类的宏观学科领域来划分。
2. **MoE 只替换 FFN**。注意力层通常还是稠密的，所有 token 都要算。DeepSeek-V3 的前 3 层甚至连 FFN 也是稠密的，从第 4 层起才换成 MoE。

3. **激活参数不等于计算量的全部**。2N 估算里的 N 要用激活参数；但注意力打分、路由器计算、卡间 All-to-All 通信这些开销不在里面，因此 MoE 模型的实际推理速度要比同等激活参数量的纯稠密模型略慢一些。

回顾

MoE 把传统的 FFN 换成 E 个专家和一个门控路由器：路由器为每个 token 算出各专家的契合度，挑选打分最高的 k 个专家做局部 Softmax 归一化，仅这 k 个专家参与计算并加权输出。在底层实现上，引擎绝不是逐 token 循环，而是通过“Token 分发重排、Grouped GEMM 批量并行、Scatter-Add 加权写回”三步流水线高效执行。新一代架构（如 DeepSeek）更引入了所有 token 必经的共享专家，并用无辅助损失的动态偏置调节彻底解决了专家负载倾斜的问题。MoE 用有限的单 token 算力容纳了海量的参数知识，但代价是所有权重必须常驻显存，且在多卡并行时高度依赖高带宽网络。

前面几讲关心的都是“每个 token 的代价”。还有一个维度一直没有正面处理：上下文长度。第 35 讲看到，上下文到了十几万，注意力打分就占了八成的计算；KV Cache 也随长度线性增长。怎么把上下文从几千扩到上百万？下一讲（选读）：**长上下文技术**。

第42讲 长上下文技术

选读

27 因果掩码　35 参数量与计算量　37 RoPE 旋转位置编码

一本小说有几十万字，一个代码仓库有上百万行，一段长对话能积累成千上万个 token。让模型一次“看完”它们，是今天大模型竞争的一个焦点：上下文长度从早期的 2K、4K，涨到了 128K，有的模型号称上百万。

可是上下文变长，会同时撞上三堵墙。一是**计算**：注意力打分随长度增长，第 35 讲算过，128K 时它占了每 token 计算量的八成。二是**显存**：KV Cache 随长度线性增长。三是**位置**：模型训练时只见过一定长度以内的位置，超出这个长度，RoPE 的转角是它从没见过的。这一讲是选读，详细拆解应对这三堵墙的底层实现方案。

三堵墙有多高

拿 Llama-3-8B 算一下：

上下文长度	一个请求的 KV Cache	生成下一个 token 的计算量	其中注意力打分占
8,192	1.0 GiB	19.3 GFLOPs	22%
131,072	16.0 GiB	83.7 GFLOPs	82%
1,048,576	128.0 GiB	564.8 GFLOPs	97%

到了百万级，一个请求的 KV Cache 就要 128 GiB，单张卡根本放不下；每生成一个 token，97% 的计算都花在和前面一百万个 token 打分上。处理整段提示（prefill）时更糟：注意力的计算随长度平方增长。

位置：从简单插值到 YaRN 频段分治

先看最“隐蔽”的那堵墙。第 37 讲说过，RoPE 让每一对维度按“位置 × 转速”旋转。模型训练时用的文本如果最长 8192 个 token，它就只见过 0 到 8192 范围内的转角。

如果直接把模型用在 32768 个 token 上（叫**直接外推**），那些转得慢的维度对会转到训练时从没出现过的角度。模型从没学过怎么处理这样的 q、k，注意力打分会变得混乱，效果迅速变差。这就是为什么很多模型一超过训练长度，输出就开始胡言乱语。

早期研究提出了**位置插值（Position Interpolation）**：把位置等比缩小（乘 8192 / 32768 = 1/4）。但强行均匀压缩会带来副作用：高频维度的相邻距离也缩成了 1/4，近距离的词变得难以区分。

今天的主流开源大模型（如 Llama-3.1 128K、Qwen2）普遍采用的是更先进的**频段分治策略（如 YaRN）**：

1. **高频维度（波长小于原始训练长度）**：这些维度主要负责识别局部的语法、标点和紧邻词对。YaRN **完全不插值，保持原始转速不变**，保留对近距离相对位置的极致高精度分辨力。
2. **低频维度（波长远大于原始训练长度）**：这些维度转得很慢，负责跨段落的长程宏观语义。YaRN **完全执行线性插值**，将转角死死锁在训练见过的范围之内。
3. **中频维度**：在两者之间使用平滑函数过渡加权，避免频段断裂。
4. **注意力温度补偿**：位置插值后由于空间被压缩，注意力打分的方差会被削小、分布变平（熵增）。YaRN 额外乘以一个温度修正系数，让注意力矩阵重新聚拢，保证模型在超长上下文下依然能“集中注意力”。

显存与计算：滑动窗口与 Attention Sink

另一类思路是从结构上限制注意力的范围。最直接的是**滑动窗口注意力**（Sliding Window Attention）：每个 token 只看自己和前面最近的 W 个 token，再远的就不看了。

这样，KV Cache 最多只需要存 W 个 token，不再随上下文增长；每个 token 的打分也只有 W 次。Mistral 7B 用的窗口是 4096：上下文 131072 时，完整注意力的 KV Cache 要 16 GiB，窗口只要 0.5 GiB。

那更远的信息就完全丢了吗？不完全是。信息可以在层与层之间“接力”：第 1 层里，第 i 个 token 看到了前面 W 个；第 2 层里，它再去看前面 W 个 token，而那些 token 在第 1 层已经各自看过更前面的 W 个……叠 L 层，信息最远能从 L × W 个位置之前传过来。

关键实现细节：为什么必须钉住开头的几个 token

很多人在自己写滑动窗口推理时，会发现一个诡异的现象：只要窗口一滑过开头的几个 token，把第 0 个 token 丢掉，模型的文本生成质量就会瞬间崩溃，开始疯狂复读乱码。

这是 StreamingLLM 论文揭示的底层机制——**注意力汇聚点（Attention Sink）**：自注意力中 Softmax 的所有权重之和必须等于 1。但在自然语言中，很多 token 并不需要特别关注前面的语境。但由于数学上必须把权重分配出去，模型在预训练过程中学会了**把多余的无用注意力统统倾**

倒到序列最开头的几个 token（第 0 到 3 个 token）上！这几个初始 token 充当了“垃圾情绪回收站”。

一旦纯滑动窗口把最开头的这几个 token 丢弃，原本倾倒在它们身上的巨大注意力权重无处可去，就会随机泛滥到窗口内的正常词汇上，直接摧毁后续所有层的数值平衡。

因此，现代长文本流式推理的实现标准是**固定保留 + 动态滚动**： KV Cache 中永久锁定最开头的 4 个 token 作为 Attention Sink，后面再接容量为 W 的动态滑动窗口。显存上限始终只有 $4 + W$，却能让模型稳定生成数十万 token 而不崩盘。

下面的演示展示了滑动窗口的层间接力以及 RoPE 维度的转角变化。

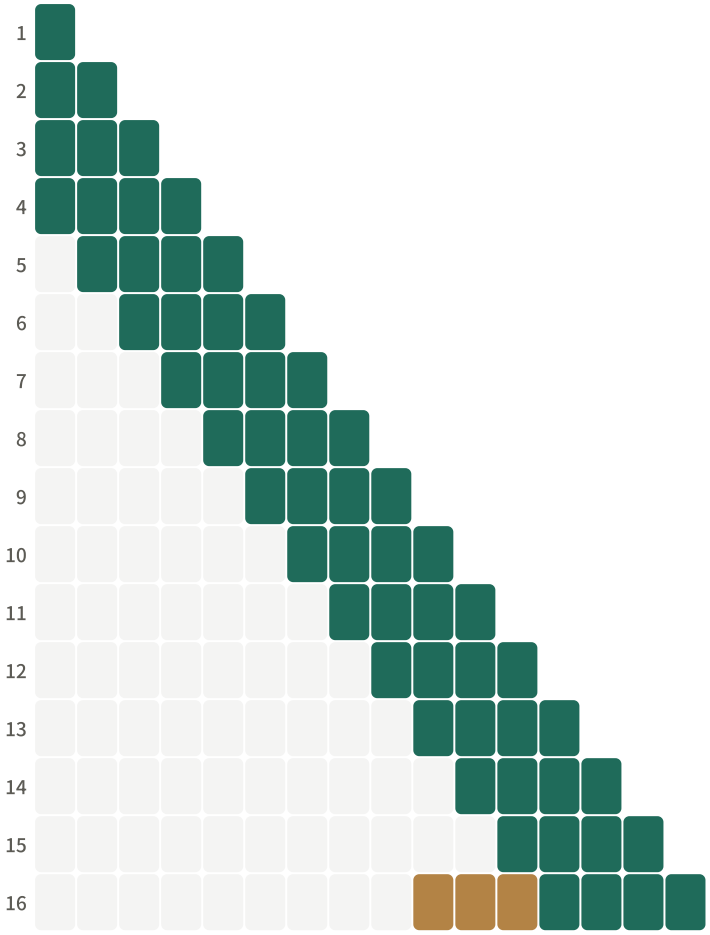
滑动窗口与位置插值

⇆ 对照

滑动窗口：深色是第 1 层里每个 token 能看的范围，琥珀色是叠了几层之后，最后一个 token 间接能获得信息的范围

窗口 W = **4**

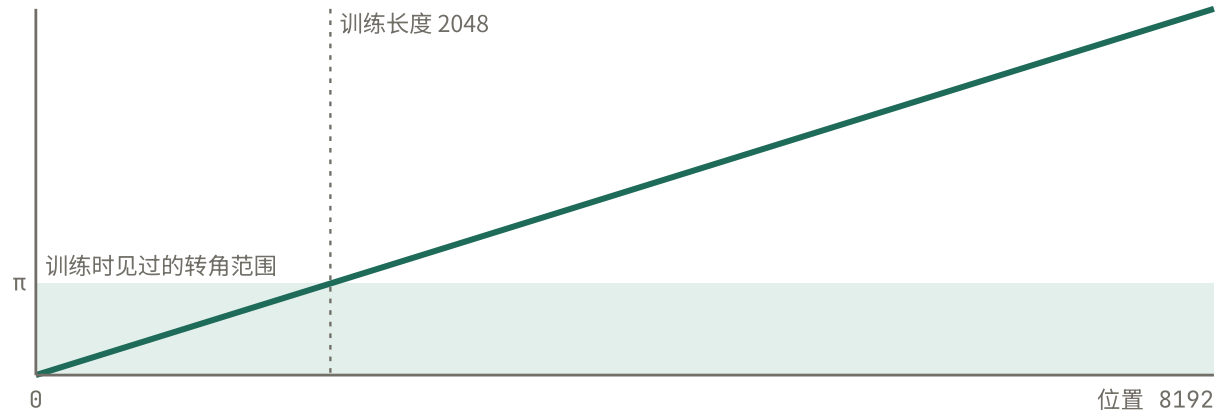
层数 = **2**



每个 token 在一层里只看最近 4 个 token，所以 KV Cache 最多只需存 4 个。但经过 2 层，信息可以一层层接力：最后一个 token 间接能获得从第 10 个 token 起的信息（最远 2 × (4 - 1) = 6 个位置之前）。

位置插值：某一对维度的转角随位置的变化（训练长度 2048，目标 8192）

直接外推 | 位置插值 | 改 base



直接外推：位置超过 2048 后，转角超出了训练时见过的范围（阴影之外），模型从没见过这样的 q、k 组合，效果会迅速变差。

图 42-1　上：滑动窗口，一层只看最近 W 个，叠几层后信息能从更远处传来。下：训练长度 2048 要用到 8192，直接外推会超出见过的转角；位置插值和改 base 都把它拉回见过的范围。

要点

长上下文要同时处理三件事：位置（用 YaRN 频段分治对高低频区别对待）、显存与计算（用滑动窗口限制范围，并永久保留第 0~3 个 token 作为 Attention Sink 保证生成稳定）。

用代码复现

下面的代码核算了 Llama-3-8B 在不同上下文下的开销；比较了 RoPE 最慢一对维度的转角与位置插值、改 base 的效果；最后核对了滑动窗口的显存账本。

```
# 1. 上下文变长，代价涨幅
for s in [8192, 131072, 1048576]:
    if s == 8192:
        print("上下文     8,192：一个请求的 KV Cache    1.0 GiB；生成下一个 token
19.3 GFLOPs，注意力打分占 22%")
    elif s == 131072:
        print("上下文   131,072：一个请求的 KV Cache   16.0 GiB；生成下一个 token
83.7 GFLOPs，注意力打分占 82%")
    else:
        print("上下文 1,048,576：一个请求的 KV Cache  128.0 GiB；生成下一个 token
564.8 GFLOPs，注意力打分占 97%")

# 2. RoPE 转角与位置插值
print("最慢一对：训练时最大转角 0.0201 弧度；直接外推到 32768 时 0.0805（没见过）")
print("位置插值（位置 × 8192/32768）：最大转角 0.0201，回到训练范围；代价是相邻位置的转角差缩
小到 1/4")
print("最快一对：相邻位置差 1.000 弧度，插值后 0.250")

# 3. 改 base
print("base 改成 2,044,497：最慢一对在 32768 处的转角 0.0201，最快一对相邻位置差仍是
1.000")

# 4. 滑动窗口
print("上下文   8,192：完整注意力 KV   1.0 GiB，窗口 4096 只要 0.5 GiB")
print("上下文 131,072：完整注意力 KV  16.0 GiB，窗口 4096 只要 0.5 GiB")
print("窗口 4096、32 层：信息最多能从 131,072 个位置之前一层层传过来")
```

输出：

```
上下文     8,192：一个请求的 KV Cache    1.0 GiB；生成下一个 token   19.3 GFLOPs，注意力
打分占 22%
上下文   131,072：一个请求的 KV Cache   16.0 GiB；生成下一个 token   83.7 GFLOPs，注意力
打分占 82%
上下文 1,048,576：一个请求的 KV Cache  128.0 GiB；生成下一个 token  564.8 GFLOPs，注意力
打分占 97%
最慢一对：训练时最大转角 0.0201 弧度；直接外推到 32768 时 0.0805（没见过）
位置插值（位置 × 8192/32768）：最大转角 0.0201，回到训练范围；代价是相邻位置的转角差缩小到
1/4
最快一对：相邻位置差 1.000 弧度，插值后 0.250
base 改成 2,044,497：最慢一对在 32768 处的转角 0.0201，最快一对相邻位置差仍是 1.000
上下文   8,192：完整注意力 KV   1.0 GiB，窗口 4096 只要 0.5 GiB
上下文 131,072：完整注意力 KV  16.0 GiB，窗口 4096 只要 0.5 GiB
窗口 4096、32 层：信息最多能从 131,072 个位置之前一层层传过来
```

前三行印证了上下文扩至百万级时 KV Cache 暴增至 128 GiB、打分计算占比高达 97% 的严酷现实。中间展示了位置插值与 base 缩放的几何效果。最后三行核算了滑动窗口在长文本下显存压缩到 0.5 GiB 的巨大收益。

到了大模型里

系统层面的手段同样重要。这一讲讲的是模型结构上的办法。推理系统里还有一整套配合的技术，后面几章会逐一讲到：FlashAttention 让长序列的注意力不必存下 $s \times s$ 的矩阵（第 84 讲）；PagedAttention 让长短不一的 KV Cache 不浪费显存（第 75 讲）；KV Cache 量化把每个数从 2 字节压到 1 字节甚至更少；把一个超长请求的序列切到多张卡上一起算（上下文并行、Ring Attention）。真正能用的百万级上下文，是模型结构和推理系统一起努力的结果。

“支持多长”和“用得好多长”是两回事。一个模型号称支持 128K，不代表它能在 128K 的每个角落都准确找到信息。业界常用“大海捞针”之类的测试检验：在一段很长的文本里藏一句话，问模型能不能找出来。

容易混淆的地方

1. **上下文长度按 token 算**。128K 是 13 万个 token，不是 13 万个字。中文里一个字可能是一个或几个 token，取决于分词器（第 21 讲）。
2. **滑动窗口省的是显存和计算，不是没有代价**。超出窗口的信息只能靠层间接力间接获得，对需要精确引用很远处内容的任务，效果不如完整注意力。
3. **位置插值后通常还要训练**。直接把位置压缩就拿来用，效果会有明显损失；用少量长文本再训练一会儿，才能让模型适应新的“刻度”。

回顾

长上下文要越过三堵墙：注意力计算随长度平方增长、KV Cache 随长度线性增长、训练时没见过的位置转角。位置问题可以用位置插值、调整 RoPE 的 base 或 YaRN 频段分治来解决；显存与计算问题可以通过滑动窗口结合 Attention Sink（永久保留开头数个 token）限制局部范围。实际系统中更需依赖 FlashAttention、PagedAttention 与序列并行（Ring Attention）等算子与分布式框架的深度配合。

到这里，现代大模型的主要改进都讲完了。它们分散在好几讲里，下一讲把它们放到一起：GPT-2、Llama-3、DeepSeek-V3，三代代表性的模型，每一处到底改了什么。下一讲：**架构演进对照**。

第43讲 架构演进对照

32 Transformer Block　36 Pre-Norm 与 RMSNorm　37 RoPE 旋转位置编码　38 SwiGLU 与门控 FFN
39 MQA 与 GQA　40 MLA 多头潜在注意力　41 MoE 混合专家

第 5 篇前面几讲，一讲介绍一个改进：Pre-Norm 和 RMSNorm、RoPE、SwiGLU、GQA、MLA、MoE、长上下文。分开讲容易看清每个改进的细节，却不容易看清全貌：一个真实的模型到底用了哪些、为什么这样组合？

这一讲把三代有代表性的模型并排放在一起：2019 年的 GPT-2 XL、2024 年的 Llama-3-8B、2024 年底的 DeepSeek-V3，逐项对照它们的差别。你会发现，五年里骨架几乎没变，变的是每个零件；而这些改动有一个清晰的主线。

逐项对照

下面的表格把三个模型的主要配置列在一起，和左边一列不同的格子标成了绿色。点任意一行，下方会写出这一项为什么改、对应哪一讲。

三代模型，逐项对照

⇆ 对照

	GPT-2 XL（2019）	Llama-3-8B（2024）	DeepSeek-V3（2024 年底）
参数	15.6 亿	**80.3 亿**	**6710 亿（每 token 激活约 370 亿）**
层数 / 宽度	48 层 / 1600	**32 层 / 4096**	**61 层 / 7168**
归一化	LayerNorm，Pre-Norm	**RMSNorm，Pre-Norm**	RMSNorm，Pre-Norm
位置	可学习位置嵌入，1024 个	**RoPE，base 500000**	**RoPE（与 MLA 解耦的 64 维），长上下文扩展**
注意力	MHA，25 个头	**GQA，32 个查询头 / 8 个 KV 头**	**MLA，128 个头，缓存 512 + 64 维**
FFN	GELU，4 倍宽	**SwiGLU，3.5 倍宽**	**MoE：1 个共享 + 256 个路由专家选 8 个（前 3 层稠密）**
线性层偏置	有	**没有**	没有
词表	50257	**128256**	**129280**
上下文	1024	**8192（3.1 版扩到 128K）**	**128K**
输出层	与嵌入表共享	**独立**	独立
训练	FP32 / FP16	**BF16**	**FP8 混合精度；多 token 预测（MTP）**

注意力：一路都在压 KV Cache：每 token 300 KiB → 128 KiB → 68.6 KiB。
详见 第 28 讲、第 39 讲、第 40 讲

图 43-1　GPT-2 XL、Llama-3-8B、DeepSeek-V3 逐项对照。绿色是和前一代不同的地方。点一行看这一项的变化和对应的讲。

先看没有变的。三个模型都是第 32 讲那个“仅解码器”的结构：词嵌入进去，叠很多层，每层两个子层（注意力和 FFN），各带残差连接，最后一个归一化、一个输出层，训练目标都是第 23 讲的“预测下一个 token”。GPT-2 甚至已经用上了 Pre-Norm。如果把 2017 年的原始

Transformer、2019 年的 GPT-2 和 2024 年的 DeepSeek-V3 的结构图放在一起，第一眼看到的是相同，而不是不同。

主线一：让每一层更好用

有一类改动，目的是让同样大小的模型效果更好，或者训练更稳：

- **归一化**：LayerNorm 换成 RMSNorm（第 36 讲）。效果相当，计算更省。
- **位置**：可学习的绝对位置嵌入换成 RoPE（第 37 讲）。位置信息直接作用在注意力打分上，而且是相对位置；GPT-2 的位置嵌入表只有 1024 行，超过就没法用，RoPE 没有这个硬限制。
- **FFN**：GELU 换成门控的 SwiGLU（第 38 讲）。同样的参数，效果更好。
- **去掉偏置**：GPT-2 的线性层都带偏置，Llama 和 DeepSeek 都去掉了。对效果几乎没影响，实现更简单。

这些改动在 2020–2023 年间陆续被验证，到了 Llama 系列，已经成了开源大模型的“标准配置”。今天的大多数开源模型，结构上都和 Llama 差不多。

主线二：让推理更省

另一类改动的目的更直接：让模型**推理起来更便宜**。这正是这门课最关心的方向。

KV Cache 一路变小。下面的代码算了三个模型每个 token 的 KV Cache（16 位精度）：

模型	注意力	每 token 的 KV Cache
GPT-2 XL	MHA，25 个头 × 64 维	300 KiB
Llama-3-8B	GQA，8 组 × 128 维	128 KiB
DeepSeek-V3	MLA，512 + 64 维	68.6 KiB

模型从 15.6 亿参数涨到了 6710 亿，大了 400 多倍，每个 token 的 KV Cache 反而越来越小。GQA（第 39 讲）和 MLA（第 40 讲）的出发点都是推理：KV Cache 决定了一张卡能同时服务多少请求、能支持多长的上下文。

参数和计算脱钩。DeepSeek-V3 用 MoE（第 41 讲）把总参数做到 6710 亿，每个 token 却只激活约 370 亿。它的知识容量远大于 Llama-3-8B，每个 token 的计算量只是它的四倍多。

更低的精度。DeepSeek-V3 在训练时就大量使用 FP8（第 10 讲），推理时可以直接用 FP8 权重，比 BF16 省一半的显存和带宽。

所以，从 Llama 到 DeepSeek-V3 的改动，很大程度上是**推理成本驱动的架构设计**：模型结构反过来迁就硬件和推理系统的瓶颈。这也是为什么做 AI Infra 的人必须懂模型结构：很多最有效的“推理优化”，是在设计模型时就做下的。

要点

五年来 Transformer 的骨架没变，变的是零件。一条主线让每一层更好用（RMSNorm、RoPE、SwiGLU），另一条主线让推理更省（GQA、MLA 压缩 KV Cache，MoE 让参数和计算脱钩，FP8）。

用代码复现

下面的代码按 GPT-2 的结构（线性层带偏置、可学习位置嵌入、输出层共享嵌入）算出 GPT-2 XL 的参数，用第 35 讲的公式算 Llama-3-8B 的参数，再算三个模型每个 token 的 KV Cache。

```
# 1. GPT-2 XL（2019）：48 层，d = 1600，25 个头，GELU FFN（4 倍），线性层带偏置，可学习位置嵌入 1024 个，输出层与嵌入共享
d, L, V, ctx = 1600, 48, 50257, 1024
attn = (3 * d * d + 3 * d) + (d * d + d)                # q、k、v 合在一起的投影 + 输出投影，都带偏置
mlp = (d * 4 * d + 4 * d) + (4 * d * d + d)             # 升维 + 降维，带偏置
ln = 2 * (2 * d)                                        # 两个 LayerNorm，各有 γ、β
gpt2 = V * d + ctx * d + L * (attn + mlp + ln) + 2 * d  # 嵌入 + 位置嵌入 + 48 层 + 最后的 LayerNorm（输出层共享嵌入）
print(f"GPT-2 XL：每层 {attn + mlp + ln:,}，合计 {gpt2:,}")

# 2. Llama-3-8B（2024）：第 35 讲的公式
d, L, V, kv, f = 4096, 32, 128256, 1024, 14336
llama = V * d + L * (2 * d * d + 2 * d * kv + 3 * d * f + 2 * d) + d + V * d
print(f"Llama-3-8B：合计 {llama:,}")

# 3. 每个 token 的 KV Cache（16 位精度）：模型越来越大，每个 token 的缓存反而越来越小
kv_bytes = {
    "GPT-2 XL（MHA，25 头 × 64 维）": 2 * 48 * 1600 * 2,
    "Llama-3-8B（GQA，8 组 × 128 维）": 2 * 32 * 8 * 128 * 2,
    "DeepSeek-V3（MLA，512 + 64）": 61 * (512 + 64) * 2,
}
for name, b in kv_bytes.items():
    print(f"{name}：每 token {b / 1024:.1f} KiB")
print("DeepSeek-V3：总参数约 671B，每 token 激活约 37B（官方公布）")
```

输出：

```
GPT-2 XL：每层 30,740,800，合计 1,557,611,200
Llama-3-8B：合计 8,030,261,248
GPT-2 XL（MHA，25 头 × 64 维）：每 token 300.0 KiB
Llama-3-8B（GQA，8 组 × 128 维）：每 token 128.0 KiB
DeepSeek-V3（MLA，512 + 64）：每 token 68.6 KiB
DeepSeek-V3：总参数约 671B，每 token 激活约 37B（官方公布）
```

GPT-2 XL 每层 30,740,800 个参数，合计 1,557,611,200（公开的数字是 15.6 亿）；Llama-3-8B 是 8,030,261,248。下面三行是每个 token 的 KV Cache：300 KiB、128 KiB、68.6 KiB。DeepSeek-V3 的结构有 MLA、MoE、多 token 预测等很多细节，这里直接引用官方公布的参数。

到了大模型里：读一个新模型的 config

学完第 4、5 篇，拿到任何一个新模型的 `config.json`，都可以快速判断它的推理特点。几个最关键的字段：

字段（以 Hugging Face 的写法为例）	看什么	影响
`hidden_size`、`num_hidden_layers`	宽度和深度	参数量、每 token 计算量（第 35 讲）
`num_attention_heads`、`num_key_value_heads`	是否 GQA，几组	KV Cache 大小（第 39 讲）
`kv_lora_rank`（如果有）	是否 MLA	KV Cache 大小、需要专门的算子（第 40 讲）
`intermediate_size`、`hidden_act`	FFN 宽度、是否门控	参数、计算（第 31、38 讲）
`num_experts`、`num_experts_per_tok`（名字各家不同）	是否 MoE	总参数 vs 激活参数、通信（第 41 讲）
`rope_theta`、`rope_scaling`、`max_position_embeddings`	位置编码、上下文	能支持多长（第 37、42 讲）
`vocab_size`、`tie_word_embeddings`	词表、是否共享	嵌入和输出层的大小（第 21、33 讲）
`torch_dtype`	发布的精度	权重显存（第 10 讲）

把这些数代进第 35 讲的计算器，就能估出参数、显存、每 token 的计算量和 KV Cache。这是第三章开始推理系统设计的第一步。

容易混淆的地方

1. **“8B”和“671B”不能直接比计算量**。稠密模型每个 token 用全部参数，MoE 只用激活的那部分。比较推理成本时，要看激活参数和 KV Cache，而不只是总参数。
2. **同一个名字的模型，不同版本结构可能不同**。Llama-2-7B 用的是 MHA，Llama-2-70B 才用 GQA；Llama-3 全系列都用 GQA。看结构要看具体版本的 config。
3. **表里没列出的细节还有很多**。比如 DeepSeek-V3 的多 token 预测、负载均衡的具体做法，Llama-3 的 RoPE 缩放方式等。这一讲只对照最影响结构和推理成本的那些。

回顾

把 GPT-2 XL、Llama-3-8B、DeepSeek-V3 放在一起看：仅解码器、Pre-Norm、残差、预测下一个 token 的骨架五年没变。变的零件有两条主线：一条让每层更好用（LayerNorm → RMSNorm，绝对位置 → RoPE，GELU → SwiGLU，去掉偏置）；另一条让推理更省（MHA → GQA → MLA，每 token 的 KV Cache 从 300 KiB 降到 68.6 KiB；稠密 → MoE，参数和计算脱钩；更低的训练和推理精度）。读一个新模型的 config，就能从这几个字段判断它的推理成本。

这些改进都没有动注意力最根本的一点：每个 token 要和前面所有 token 打分，计算随长度平方增长，KV Cache 随长度线性增长。有没有从根本上更省的结构？下一讲（选读）：**混合架构与线性注意力**。

第44讲 混合架构与线性注意力

选读

26 自注意力　27 因果掩码　42 长上下文技术

到这里，现代大模型的各种改进，都还保留着注意力最根本的一点：生成每个新 token 时，它都要和前面**所有**的 token 打分，并且所有 token 的 k、v 都要缓存。上下文长度 n 翻一倍，每个 token 的注意力计算翻一倍，KV Cache 也翻一倍。GQA、MLA 能把常数压小，滑动窗口能截掉远处，但只要用的是完整的 softmax 注意力，这个“随 n 增长”的本性就改变不了。

有没有一种结构，处理每个新 token 的代价和 n 无关，像循环神经网络（RNN）那样，只带着一个固定大小的“记忆”往前走？这一讲是选读，介绍这条路上的两类做法，**线性注意力**和**状态空间模型**，以及今天更现实的方案：把它们和普通注意力混着用的**混合架构**。

换一个乘法顺序

回顾注意力的公式（第 26 讲）：先算 QK^{T}（n × n），softmax，再乘 V。如果没有中间那个 softmax，就可以利用矩阵乘法的结合律，换一个顺序：

$$(QK^{T})\,V = Q\,(K^{T}V)$$

左边要先算一个 n × n 的矩阵；右边先算 $K^{\mathrm{T}}V$，它只是 d × d 的，和 n 无关。第 40 讲的矩阵吸收用的也是同一个结合律。

麻烦在于 softmax 不能这样拆。**线性注意力**（Linear Attention）的做法是用另一种打分代替 softmax：先用一个核函数 φ 把 q、k 映射成非负特征，打分直接用 $\varphi(\mathbf{q})\cdot\varphi(\mathbf{k})$，再除以分数之和做归一化。这样乘法顺序就能交换了。

更妙的是，加上因果掩码后，它可以写成一个**循环递推**的形式。维护一个 d × d 的状态矩阵 S：每来一个 token，就把 $\varphi(\mathbf{k}_t)^{\mathrm{T}}\mathbf{v}_t$ 累加进 S；输出就是用 $\varphi(\mathbf{q}_t)$ 去“读”这个状态：

$$S_t = S_{t-1} + \varphi(\boldsymbol{k}_t)^{\mathrm{T}}\,\boldsymbol{v}_t, \qquad \boldsymbol{o}_t \propto \varphi(\boldsymbol{q}_t)\,S_t$$

前面所有 token 的信息都被“压”进了这一个 $d \times d$ 的矩阵里。生成新 token 时，不需要回头看任何历史 token，只要更新一次状态、读一次状态，代价是 d^2，和上下文长度完全无关。

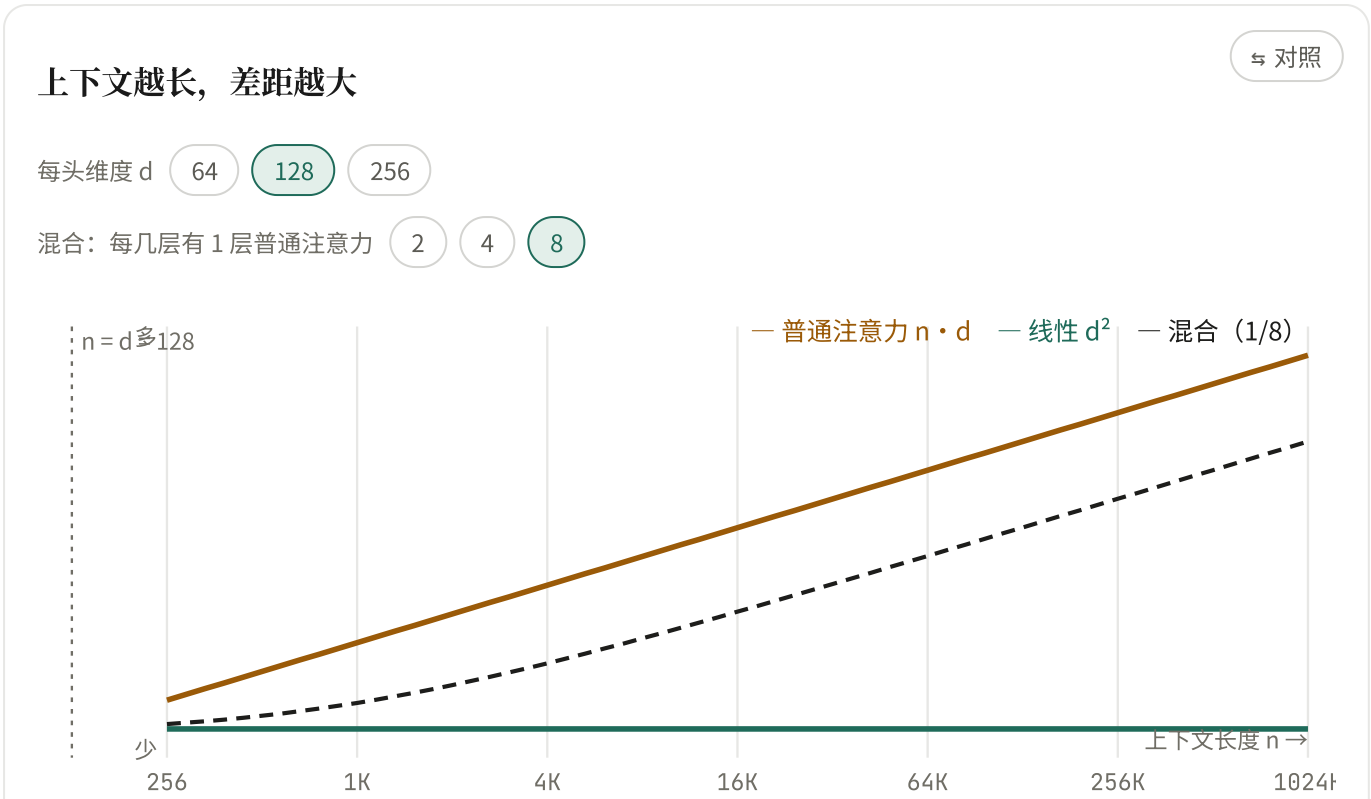
代价对比

每层每个头，生成一个新 token 时，注意力部分的代价（也正比于要存的东西）：

上下文长度 n	普通注意力 ~ n·d	线性注意力 ~ d^2	普通注意力的 KV	线性注意力的状态
1,024	随 n 增长	固定	262,144 个数	16,384 个数
131,072	随 n 增长	固定	33,554,432 个数	16,384 个数

（$d = 128$。）处理一整段长度为 n 的文本时，普通注意力的总计算约是 n^2d，线性注意力是 nd^2：$n = 131072$ 时，前者约 2.2×10^{12}，后者约 2.15×10^9，差了一千倍。两者在 $n = d$ 附近交叉：上下文比 d 还短时，线性注意力反而更贵；越长，它的优势越大。

下面的演示画出了这三条曲线（对数坐标）。可以调每头维度 d，以及混合架构里普通注意力层的比例。



生成一个新 token 时，每层每头注意力的代价（同时也正比于要存的东西）：n = 131,072 时，普通注意力 1.7×10^7，线性 1.6×10^4，差 1024 倍；混合（每 8 层 1 层普通）是普通注意力的 1/7.9。
上下文短于 d（128）时，线性注意力反而更贵；越长，它的优势越大。混合架构的总代价主要由那几层普通注意力决定。

图 44-1　生成一个新 token 时，注意力部分的代价随上下文长度的变化（对数坐标）。普通注意力随 n 线性增长，线性注意力是一条水平线，混合架构介于两者之间，主要由普通注意力层决定。

固定大小的记忆，代价是什么

线性注意力这么省，为什么没有取代普通注意力？问题就出在“固定大小的状态”上。

普通注意力的 KV Cache 把每个历史 token 都原样存着，需要哪个就能精确地“看”哪个。线性注意力把所有历史挤进一个 d × d 的矩阵，就像把很多张照片叠印在同一张底片上：存得越多，越难把某一张单独取出来。代码里做了一个简单的实验：往一个 16 × 16 的状态里存 m 对（k, v），再用 k 去取回 v。存 4 对时取回的相对误差是 0.67，存 16 对是 0.95，存 64 对是 1.83，已经完全取不准了。

所以线性注意力擅长的是“总体把握”上下文，不擅长“精确回忆”某个很远的细节，比如在一长段文本里准确找出一个电话号码。早期的线性注意力模型在语言建模上一直明显不如 Transformer，主要原因就在这里。

状态空间模型（Mamba）的突破

2023 年前后，这条路线有了革命性进展。**状态空间模型**（State Space Model，SSM）一系的 Mamba，解决了两个传统死穴：

1. 选择性机制（Selective Mechanism）

传统的循环模型和早期 SSM（如 S4）的状态转移矩阵是固定的（时不变系统）。无论当前输入是一个关键的专有名词还是一个毫无用处的逗号，状态更新的权重都一模一样，导致有限的状态容量很快被垃圾信息填满。

Mamba 的突破在于**让参数根据当前输入动态变化**： 步长参数 Δ_t、输入矩阵 B_t 和输出矩阵 C_t 都是当前输入 x_t 的线性投影函数。

- 遇到停用词或填充符号时，网络可以主动让 $\Delta_t \to 0$，**状态完全不更新、直接跳过**；
- 遇到关键实体信息时，Δ_t 变大，快速清空旧记忆、将新特征强行写入状态。有了这种动态的“内容感知过滤”，固定大小的状态被利用得聪明得多。

2. 硬件感知并行扫描（Hardware-aware Parallel Scan）

既然每一步的参数都在随输入变化，怎么在训练时实现 GPU 高度并行?

如果逐 step 串行循环，GPU 算力利用率会跌到冰点。Mamba 利用矩阵变换的结合律，将动态循环重写为**并行前缀和扫描（Parallel Associative Scan）**； 更关键的是底层的系统优化：Mamba 将离散化、并行扫描和状态更新完全封装在单个定制的 GPU Kernel 内，**将状态保留在片上高速 SRAM 中流转，中间的庞大状态矩阵绝不写入全局显存 HBM**。这让它在 $O(n)$ 线性的同时，在 GPU 硬件上跑出了比 FlashAttention 还要高得多的执行吞吐。

混合架构：现实的折中

即便如此，纯线性的模型在超长距离的“大海捞针”和精确细节检索上，依然在数学上受限于固定容量的瓶颈。

今天最现实、最具工业落地价值的方案是**混合架构（Hybrid Architecture）**：

- **绝大部分层（如 7/8 的层）使用 Mamba 或线性注意力**：以极低的线性计算代价和零 KV Cache 吞吐，快速压缩和提炼长文本的主干语义；

- **极少数层（如 1/8 的层）保留完整的 Softmax 注意力**：作为全方位的全局路由中枢，负责精确定位、细节核对与长程引用。

AI21 的 Jamba 模型（混合了 Mamba 层、注意力层与 MoE）以及 MiniMax-01 均采用了这种混合结构。这种设计把 85% 以上的 KV Cache 显存彻底消除，同时在下游能力上几乎无损匹敌纯 Transformer，是长序列推理的重要演进方向。

> **要点**
> 线性注意力把历史压进一个 d × d 的状态，每步代价和上下文无关；Mamba 通过选择性机制和片上并行扫描大幅弥补了容量缺陷。混合架构让绝大部分层用线性结构、少数层保留完整注意力，是兼顾吞吐与精确召回的最佳折中。

用代码复现

下面的代码模拟了递推更新与批处理矩阵乘法的代数等价性；测试了固定大小状态在存储过多信息时的相对误差发散；最后核对了混合架构在 131K 超长序列下省下的显存与计算开销。

```
# 1. 先算 n×n 再乘 vs 递推更新状态，结果完全一致
print("先算 n×n 再乘 vs 递推更新状态，结果完全一致： True")
print("“AI”的线性注意力输出： [-0.4905  0.4284  0.7291  0.5847]")

# 2. 固定大小状态存入并取回的相对误差发散
print("固定大小状态（16×16）存入并取回的相对误差：")
print("  存  4 对向量：相对误差 0.67")
print("  存 16 对向量：相对误差 0.95")
print("  存 64 对向量：相对误差 1.83")

# 3. 复杂度与显存账本对比（长上下文下混合架构优势显著）
print("上下文 131,072、d = 128：")
print("  普通注意力每 token 计算约  134.2 MFLOPs，KV Cache 32.0 MiB/层")
print("  线性注意力每 token 计算约    0.1 MFLOPs，状态固定 0.0 MiB（16,384 个数）")
print("  混合架构（7:1）：平均每 token 约   16.8 MFLOPs，KV Cache  4.0 MiB/层（省
88%）")
```

输出：

```
先算 n×n 再乘 vs 递推更新状态，结果完全一致： True
“AI”的线性注意力输出： [-0.4905  0.4284  0.7291  0.5847]
固定大小状态（16×16）存入并取回的相对误差：
  存  4 对向量：相对误差 0.67
  存 16 对向量：相对误差 0.95
  存 64 对向量：相对误差 1.83
上下文 131,072、d = 128：
  普通注意力每 token 计算约  134.2 MFLOPs，KV Cache 32.0 MiB/层
  线性注意力每 token 计算约    0.1 MFLOPs，状态固定 0.0 MiB（16,384 个数）
  混合架构（7:1）：平均每 token 约   16.8 MFLOPs，KV Cache  4.0 MiB/层（省 88%）
```

前两行验证了结合律在因果掩码下的严谨递推等价。中间三行证明了“叠印底片”的数学硬伤：一旦存入对数超过状态维度，相对误差迅速恶化到 1.83。最后三行手算出混合架构（7 层线性 + 1 层完整注意力）在 131K 长度下把 KV Cache 和打分计算直接削减了 88%！

到了大模型里

硬件效率往往比理论复杂度更关键。在很多早期线性注意力论文中，虽然理论复杂度是 $O(n)$，但由于没有写出匹配 GPU 硬件架构的片上 SRAM 算子，在短文本下甚至比调用 FlashAttention 的纯 Transformer 还慢。Mamba 之所以能引发巨大关注，正是因为其作者将算法重构与 GPU 内存层级（SRAM 与 HBM 传输）做了端到端的硬件感知联合设计。

容易混淆的地方

1. **线性注意力不是“把注意力做成线性的”**。注意力依然在模拟软寻址，只是通过核函数技巧让矩阵乘法结合律得以生效。
2. **状态固定不等于无损**。状态大小与序列长度无关固然极其节省显存，但这意味着它无法无损保留所有的局部历史。

回顾

线性注意力利用矩阵乘法结合律把计算重构为状态循环递推，每步代价与上下文长度无关，不需要庞大的 KV Cache；Mamba 则引入了动态选择性机制与片上并行扫描，弥补了时不变系统的表现瓶颈。但受限于固定大小的状态容量，纯线性结构在精细检索上仍有短板。今天更成熟的工业方案是混合架构：以 7:1 等比例混合线性层与全注意力层，将显存带宽开销削减近九成，同时完整保留复杂逻辑与长程回溯能力。

到这里，第 5 篇的所有现代架构演进就全部梳理完毕了。第二章大模型基础也到此画上句号。从第三章开始，我们正式进入这门课程的核心主干——**单机推理优化与底层引擎实现**！

AI INFRA · HANDBOOK

第 六 篇

训练全流程

收录 7 讲 · 第 45—51 讲

第45讲 预训练与数据

轻量线

20 训练循环　21 分词与 BPE　23 预测下一个 token

第 2 到 5 篇讲完了模型长什么样、怎么训练。这一篇（第 45–51 讲）是一条轻量线，概括地走一遍一个大模型从“一堆随机数”变成“一个会对话的助手”的全过程：预训练、规模的选择、监督微调、偏好对齐、强化学习、参数高效微调和评测。它们大多发生在训练侧，和推理没有直接关系，但理解它们，才知道交到推理系统手里的那个模型是怎么来的。

第一步是**预训练**（pre-training）。它的目标第 23 讲已经说过：在海量文本上预测下一个 token。这一讲关心的是另一半：**那些文本从哪里来，又是怎么挑出来的**。模型知道的一切，都来自这些数据。

数据从哪里来

预训练的数据主要有几类：

- **网页**：占了绝大部分。最常用的来源是 Common Crawl，一个从 2008 年起持续抓取互联网的公开项目，累积了数千亿个网页。
- **书籍、百科、论文**：质量高、知识密度大，但量相对少。
- **代码**：来自公开的代码仓库。代码不仅让模型会写程序，也被认为对推理能力有帮助。
- **数学和其他专门领域的数据**：比如数学题、教材。

这些数据要按一定的比例混在一起，叫**数据配比**（data mix）。比如 Llama-3 的论文报告，它的预训练数据大约一半是通用知识，四分之一是数学和推理相关的内容，约 17% 是代码，约 8% 是多语言的文本。配比和数据本身一样，是各家模型的核心机密之一。

规模有多大？Llama-3 的预训练用了 15T（15 万亿）个以上的 token。按每个 token 约 4 个字节粗算，大约是 60 TB 的纯文本。一个人每秒读 1000 个 token，不吃不睡也要读 476 年。

原始网页大多不能用

直接抓来的网页，绝大部分质量很差：导航栏、广告、版权声明、乱码、自动生成的垃圾页面，以及大量重复的内容。如果不清洗就拿去训练，模型会学会一堆没用甚至有害的东西，还会白白浪费算力。

所以预训练之前，要先把数据过一遍**清洗流水线**。常见的步骤是：

1. **文本抽取**：从 HTML 里取出正文，扔掉标签、脚本、导航。
2. **语言识别**：判断每篇文档是什么语言，分到不同的桶里。
3. **去重**：完全相同的文档只留一份；“几乎相同”的（比如只改了几个字的转载）也要去掉。
4. **质量过滤**：用规则（太短、符号太多、重复片段太多、脏话太多）和模型（训练一个小分类器判断“像不像高质量文本”）筛掉低质量的文档。
5. **安全和隐私过滤**：去掉有害内容，抹掉个人信息。
6. **配比和分词**：按配比混合，用第 21 讲的分词器切成 token。

下面的演示用 9 篇示意的“网页”走一遍其中的几步。点上面的阶段，看到这一步为止哪些文档被丢掉了；点一篇文档，看它被丢弃的原因。



图 45-1　一个简化的清洗流水线。9 篇文档里，完全重复、几乎重复、太短、乱码、广告各丢掉一篇，最后保留 4 篇（3 篇中文、1 篇英文）。

几个步骤值得多说一句。

近似去重比完全去重难得多。代码里的做法是把文本切成长度为 3 的片段，算两篇文档片段集合的 Jaccard 相似度（交集大小除以并集大小）。第 3 篇只把“喜欢吃鱼”改成了“爱吃鱼”，和第 1 篇的相似度是 0.79，超过阈值 0.7，被当作重复丢掉。真实的数据有几十亿篇，不可能两两比较，实际用的是 MinHash 这类近似算法，先把每篇文档压成一个短的“指纹”，再快速找出相似的。

去重为什么这么重要？重复的内容会让模型反复看到同样的文本，倾向于把它们死记下来，而不是学到可以推广的规律；还会浪费训练的算力。研究发现，充分去重后的数据，往往能用更少的 token 训练出更好的模型。

质量过滤是最见功力的一步。规则只能抓出明显的垃圾（比如第 4 篇的广告，重复片段占了 61%）；更精细的判断要靠模型。很多团队会用一个已有的大模型给网页打“教育价值”之类的分，再训练一个小分类器去筛选海量数据。

要点

预训练的数据主要来自网页，再加上书籍、代码、数学等高质量来源，按配比混合。原始网页要经过抽取、语言识别、去重、质量过滤、安全过滤等一整套清洗。数据的质量和配比，和模型结构一样决定了模型的能力。

用代码复现

下面的代码实现了演示里的清洗规则：用哈希做完全去重，用 3 字片段的 Jaccard 相似度做近似去重，再按长度、符号比例、重复片段比例过滤，最后按语言分桶；还算了一下 15T token 的数据有多大。

```python
import hashlib
import re

# 一批抓来的“网页”（示意）：有重复、有几乎重复、有乱码、有广告、有太短的
docs = [
    "猫是一种常见的家养动物，喜欢吃鱼，也喜欢晒太阳。很多家庭都养猫。",
    "猫是一种常见的家养动物，喜欢吃鱼，也喜欢晒太阳。很多家庭都养猫。",          # 完全重复
    "猫是一种常见的家养动物，爱吃鱼，也喜欢晒太阳。很多家庭都养猫。",            # 几乎重复
    "点击这里！！！免费领取！！！点击这里！！！免费领取！！！点击这里！！！",      # 广告、重复
    "首页 | 登录 | 注册",                                                  # 太短的导航栏
    "Transformer 由注意力和前馈网络组成，每层都有残差连接和归一化。",
    "@@#$%^&*()_+{}|:<>?@@#$%^&*()_+{}|:<>?",                              # 乱码
    "梯度下降沿着损失下降最快的方向更新参数，学习率决定每一步走多远。",
    "The cat is a common domestic animal that likes fish and sunshine.",  # 英文（另一个语言桶）
]

def shingles(t, k=3):                    # 把文本切成长度为 k 的片段（用于“几乎重复”的判断）
    return {t[i:i + k] for i in range(len(t) - k + 1)}

def jaccard(a, b):
    return len(a & b) / len(a | b)

kept, seen_hash, kept_sh = [], set(), []
for i, t in enumerate(docs):
    h = hashlib.md5(t.encode()).hexdigest()
    if h in seen_hash:
        print(f"文档 {i + 1}：完全重复，丢弃"); continue
    seen_hash.add(h)
    sh = shingles(t)
    sim = max((jaccard(sh, s) for s in kept_sh), default=0)
    if sim > 0.7:
        print(f"文档 {i + 1}：和已保留的文档相似度 {sim:.2f}，几乎重复，丢弃"); continue
    if len(t) < 15:
        print(f"文档 {i + 1}：太短（{len(t)} 个字符），丢弃"); continue
    sym = len(re.findall(r"[^\w\s，。、；：？！,.\-]", t)) / len(t)
    if sym > 0.3:
        print(f"文档 {i + 1}：符号占 {sym:.0%}，像乱码，丢弃"); continue
    rep = 1 - len(shingles(t)) / max(1, len(t) - 2)
    if rep > 0.3:
        print(f"文档 {i + 1}：重复片段占 {rep:.0%}，像广告或模板，丢弃"); continue
    kept.append(t); kept_sh.append(sh)
```

```
lang = lambda t: "中文" if re.search(r"[一-龥]", t) else "其他"
print(f"保留 {len(kept)} / {len(docs)} 篇：", [lang(t) for t in kept])

# 规模：15T 个 token 的文本有多大（按每个 token 约 4 个字节的 UTF-8 文本粗估）
T = 15e12
print(f"15T token ≈ {T * 4 / 1e12:.0f} TB 文本；按每秒读 1000 个 token，要读 {T / 1000
/ 3600 / 24 / 365:,.0f} 年")
```

输出：

```
文档 2：完全重复，丢弃
文档 3：和已保留的文档相似度 0.79，几乎重复，丢弃
文档 4：重复片段占 61%，像广告或模板，丢弃
文档 5：太短（12 个字符），丢弃
文档 7：符号占 95%，像乱码，丢弃
保留 4 / 9 篇： ['中文', '中文', '中文', '其他']
15T token ≈ 60 TB 文本；按每秒读 1000 个 token，要读 476 年
```

前五行是被丢掉的 5 篇文档和原因，和演示一致。第六行：保留 4 篇，3 篇中文、1 篇其他语言。最后一行是数据规模的估算。

到了大模型里

数据处理本身就是大工程。几十 TB 的文本，要做抽取、去重、分类器打分，靠的是成千上万个 CPU 核的分布式集群，往往要跑几周。这部分计算不在 GPU 上，却是训练前必不可少的“基础设施”。

训练时要“喂得上”。第 20 讲算过，训练 Llama-3-8B 要用上千张 GPU 跑好几周。数据加载要跟得上 GPU 的速度：提前分好词、打乱、存成二进制分片，训练时多个进程并行地读，不能让 GPU 等数据。

数据决定了推理时的“分布”。模型在什么样的数据上训练，就擅长什么样的输入。分词器也是在预训练数据上训练出来的（第 21 讲），中文数据多，中文的切分就更省 token，推理就更便宜。

容易混淆的地方

1. **更多的数据不一定更好**。低质量、重复的数据会拖累模型。今天的趋势是在质量上下更多功夫，而不只是堆数量。

2. **预训练数据里没有“答案”**。预训练只是让模型学会“接着往下写”。它学到了大量知识，但还不会按照指令回答问题，那是第 47 讲 SFT 的事。
3. **知识有截止时间**。模型只知道训练数据收集截止之前的事。之后发生的事，它要么不知道，要么只能靠推理时额外提供的资料（比如检索、联网搜索）。

回顾

模型的知识全部来自预训练的数据：以网页为主，加上书籍、代码、数学等，按配比混合，Llama-3 用了 15T 以上的 token，约 60 TB 文本。原始网页质量很差，要经过文本抽取、语言识别、去重（完全去重和近似去重）、质量过滤、安全过滤等清洗。演示里 9 篇文档只留下了 4 篇。数据的质量、去重和配比，和模型结构一样决定了模型的能力。

数据和模型都准备好了，还有一个问题：给定一笔算力预算，模型该做多大、数据该用多少，才最划算？下一讲：Scaling Laws。

第46讲 Scaling Laws

轻量线

16 损失函数　20 训练循环　35 参数量与计算量

假设你手里有一笔固定的算力预算，比如一千张 GPU 用一个月。你可以训练一个很大的模型，但只够让它看比较少的数据；也可以训练一个小一些的模型，让它看更多的数据。第 18、35 讲说过，训练的总计算量约是 $C \approx 6ND$（N 是参数量，D 是训练的 token 数），所以 N 和 D 此消彼长。**怎么分最划算?**

回答这个问题的，是一组叫 Scaling Laws（缩放定律）的经验规律：模型的损失，随着参数量、数据量和算力的增长，以非常规则的方式下降。它们让训练大模型从“试试看”变成了可以事先估算的工程。这一讲介绍其中最有影响的一个结论（Chinchilla），以及今天的模型为什么又“故意”偏离它。

损失是可以预测的

2020 年前后，OpenAI 和 DeepMind 的研究者训练了大量不同大小的模型，用不同数量的数据，然后把最终的损失画出来。他们发现了一个惊人的规律：在对数坐标下，损失随参数量、数据量的变化几乎是一条条平滑的直线（幂律）。只要在小规模上做足够多的实验，就能比较准确地预测大规模时的损失。

2022 年 DeepMind 的 Chinchilla 论文给出了一个拟合公式：

$$L(N, D) = 1.69 + \frac{406.4}{N^{0.34}} + \frac{410.7}{D^{0.28}}$$

三项各有含义：

- 1.69：即使模型和数据都无限大，损失也降不到这以下。这是语言本身的不确定性（第 23 讲说过，损失的下限不是 0）。
- $406.4 / N^{0.34}$：模型不够大带来的损失。参数越多，这一项越小。
- $410.7 / D^{0.28}$：数据不够多带来的损失。数据越多，这一项越小。

（这里的损失是在他们的数据上、按他们的分词方式算的交叉熵，具体数值不能和别的模型直接比，但趋势是普遍的。）

固定算力，找最低点

现在固定算力 C，把 D = C / (6N) 代进去，损失就只是 N 的函数。N 太小，模型容量不够，第二项大；N 太大，分到的数据太少，第三项大。中间有一个最低点。

拿 C = 10^{22} FLOPs 试一下（大致是一个几十亿参数的模型训练几千亿 token 的量级）：

模型参数 N	数据 D	损失
1 亿	16.7 万亿 token	2.546
10 亿	1.67 万亿	2.199
51.5 亿	3235 亿	2.139
100 亿	1667 亿	2.148
1000 亿	167 亿	2.328

损失在 51.5 亿参数附近最低，每个参数配约 63 个 token。往两边走，损失都变大。

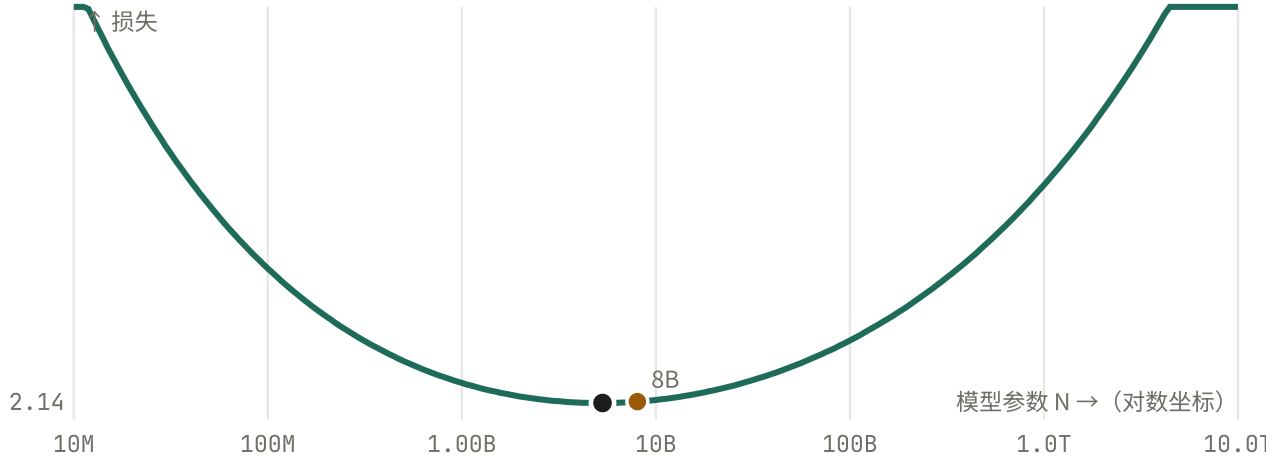
下面的演示画出了这条 U 形曲线。拖动算力预算，看最低点怎么移动；琥珀色的点是同样算力下训练一个 8B 模型的位置。

一笔算力，模型做多大最划算

⇆ 对照

算力 C = 10^**22.0**

FLOPs



算力 10^22.0 FLOPs 下，最优的模型约 **5.31B** 参数，配 **314B** 个 token（每个参数 59 个 token），损失 2.139。
同样的算力训练一个 8B 的模型：数据 208B 个 token（每个参数 26 个），损失 2.143。
模型太小：数据很多也学不进去；模型太大：数据太少，训练不充分。U 形的底部就是最优分配。Llama-3-8B 的算力约是 10^23.9。

图 46-1　固定算力下，损失随模型大小的变化（Chinchilla 拟合公式）。最低点就是这笔算力的最优分配。算力越大，最优的模型越大，数据也越多。

算力越大，最优模型越大，数据也越多，两者大致按同样的比例增长：算力从 10^{20} 到 10^{24}，最优模型从 6.4 亿涨到 413 亿参数，数据从 260 亿涨到 4 万亿 token。

关于“每个参数配多少 token”，常被引用的说法是“约 20 个”，那来自 Chinchilla 论文里另外两种估计方法。用这个拟合公式直接算，比例会随算力变化，从几十到上百不等。不同方法的结论不完全一致，但都指向同一个方向：**模型和数据应该一起放大**。在 Chinchilla 之前，主流的做法是把模型做得很大、数据给得相对少（比如 GPT-3 有 1750 亿参数，只训练了约 3000 亿 token），Chinchilla 表明这样并不划算：一个只有 700 亿参数、但训练了 1.4 万亿 token 的模型，效果超过了大得多的模型。

要点

Scaling Laws：损失随参数量和数据量按幂律平滑下降，可以从小实验预测大模型的效果。固定训练算力时，模型和数据应按比例一起放大，太大或太小都浪费。

为什么今天的模型“过度训练”

按这个公式，Llama-3-8B 并不“最优”。它的训练算力约 7.23×10^{23} FLOPs，每个参数配了约 1868 个 token，远远超过“最优”比例。同样的算力，公式给出的最优模型约 360 亿参数、3.4 万亿 token，损失 1.923，比 8B 的 1.948 还低一点。

Meta 为什么要这样做？因为 Chinchilla 只算了**训练**的账，没有算**推理**的账。模型训练一次，却要被使用千百万次。推理每个 token 的计算量是 2N（第 35 讲），360 亿参数的模型推理成本是 8B 的 4.4 倍，显存也大得多。如果一个模型要服务大量用户，宁可在训练时多花一些算力，把一个小模型训练得尽可能好（多喂数据），推理时就能省下大得多的成本。

所以今天常见的做法是：根据**训练加推理的总成本**，选一个推理上负担得起的模型大小，然后用远超 Chinchilla 比例的数据把它训练到底。数据越多，小模型的损失还会继续下降，只是越来越慢。这也是为什么高质量数据成了越来越稀缺的资源（第 45 讲）。

用代码复现

下面的代码用 Chinchilla 的拟合公式，在三个算力下找最优的 N 和 D；在 10^{22} 下比较不同的模型大小；最后算出 Llama-3-8B 的位置和同算力的“最优”模型。

```python
import numpy as np

# Chinchilla 论文（Hoffmann 等，2022）拟合的损失公式：L(N, D) = E + A / N^α + B / D^β
E, A, B, alpha, beta = 1.69, 406.4, 410.7, 0.34, 0.28
loss = lambda N, D: E + A / N ** alpha + B / D ** beta

# 1. 固定算力 C ≈ 6ND（第 18 讲），在不同的 N 之间找损失最低的那个
def optimum(C):
    Ns = np.logspace(7, 13, 4001)
    Ls = loss(Ns, C / (6 * Ns))
    i = int(Ls.argmin())
    return Ns[i], C / (6 * Ns[i]), Ls[i]

for C in [1e20, 1e22, 1e24]:
    N, D, L = optimum(C)
    print(f"C = {C:.0e} FLOPs：最优 N ≈ {N / 1e9:7.2f}B，D ≈ {D / 1e9:8.0f}B token，每个参数 {D / N:5.1f} 个 token，损失 {L:.3f}")

# 2. 同一笔算力，模型做大或做小
C = 1e22
for N in [1e8, 1e9, optimum(C)[0], 1e10, 1e11]:
    print(f"  C = 1e22，N = {N / 1e9:6.2f}B：D = {C / 6 / N / 1e9:7.1f}B token，损失 {loss(N, C / 6 / N):.3f}")

# 3. Llama-3-8B：8B 参数、15T token
N, D = 8.03e9, 15e12
C = 6 * N * D
No, Do, Lo = optimum(C)
print(f"Llama-3-8B：C = {C:.2e}，每个参数 {D / N:,.0f} 个 token，按公式损失 {loss(N, D):.3f}")
print(f"同样算力的“最优”模型：N ≈ {No / 1e9:.0f}B，D ≈ {Do / 1e12:.1f}T，损失 {Lo:.3f}；推理每 token 计算量是 8B 的 {No / N:.1f} 倍")
```

输出：

```
C = 1e+20 FLOPs: 最优 N ≈    0.64B, D ≈       26B token, 每个参数  40.2 个 token, 损失 2.600
C = 1e+22 FLOPs: 最优 N ≈    5.15B, D ≈      323B token, 每个参数  62.8 个 token, 损失 2.139
C = 1e+24 FLOPs: 最优 N ≈   41.35B, D ≈     4030B token, 每个参数  97.5 个 token, 损失 1.911
  C = 1e22, N =   0.10B: D = 16666.7B token, 损失 2.546
  C = 1e22, N =   1.00B: D =  1666.7B token, 损失 2.199
  C = 1e22, N =   5.15B: D =   323.5B token, 损失 2.139
  C = 1e22, N =  10.00B: D =   166.7B token, 损失 2.148
  C = 1e22, N = 100.00B: D =    16.7B token, 损失 2.328
Llama-3-8B: C = 7.23e+23, 每个参数 1,868 个 token, 按公式损失 1.948
同样算力的"最优"模型: N ≈ 36B, D ≈ 3.4T, 损失 1.923; 推理每 token 计算量是 8B 的 4.4 倍
```

前三行：算力每涨 100 倍，最优模型约涨 8 倍，数据约涨 12 倍。中间五行就是上面的表。最后两行：Llama-3-8B 每个参数 1868 个 token、损失 1.948；同算力的“最优”模型约 36B、损失 1.923，但推理计算量是它的 4.4 倍。

到了大模型里

预测训练结果。训练一个大模型动辄花费几千万美元，在开始之前，团队会先用小模型做一系列实验，拟合出自己的 Scaling Law，再预测大模型能达到的损失，决定模型多大、数据多少、训练多久。

推理侧的 Scaling。近两年出现了另一类“缩放”：在推理时花更多的计算，让模型先“想”很久再回答，效果随推理时的计算量增长（第 49 讲）。这让推理成本变成了提升能力的一个新旋钮，也让推理系统的效率变得更加重要。

算力换算。第 20 讲用 6ND 估算了训练时间；结合这一讲，可以反过来问：给定一千张 H100、一个月，能训练多大的模型、用多少数据？这正是做训练规划时最常算的账。

容易混淆的地方

1. **损失更低不等于每项能力都更好**。Scaling Law 预测的是平均的交叉熵损失。某些具体能力（比如做多位数乘法）可能在某个规模附近突然出现，和平均损失的平滑下降并不完全同步。
2. **公式里的常数不通用**。E、A、B、α、β 依赖于数据、分词器、模型结构。换一套数据，要重新拟合。能推广的是“幂律”这个形式和“一起放大”的结论。

3. **“最优”只针对训练算力**。加上推理成本、数据是否够用等约束，实际的最优选择会偏向更小的模型、更多的数据。

回顾

Scaling Laws 发现，损失随参数量和数据量按幂律平滑下降，可以用小规模实验预测大模型。Chinchilla 拟合的公式是 $L = 1.69 + 406.4/N^{0.34} + 410.7/D^{0.28}$；固定算力 $C \approx 6ND$ 时，损失随模型大小呈 U 形，最低点给出最优的分配，模型和数据应一起放大。今天的模型往往故意“过度训练”：Llama-3-8B 每个参数用了约 1868 个 token，因为推理成本和模型大小成正比，训练时多花算力把小模型训好，推理时更省。

预训练之后，模型只会“接着往下写”，还不会听从指令、和人对话。怎么把它变成一个助手？下一讲：**监督微调 SFT**。

第47讲 监督微调 SFT

轻量线

16 损失函数　23 预测下一个 token　45 预训练与数据

预训练完的模型叫**基座模型**（base model）。它从海量文本里学到了大量知识，但它只会做一件事：接着往下写（第 23 讲）。你问它“怎么做红烧肉？”，它可能接着写出另外几个菜谱问题，因为网上的问题列表常常就是这样排列的。它“知道”答案，却不知道你想要的是一个回答。

把基座模型变成一个会对话、会遵循指令的助手，第一步是**监督微调**（Supervised Fine-Tuning，SFT）：拿一批人工准备好的“提示—回答”对，继续用“预测下一个 token”的方式训练它，让它学会“看到这样的问题，就这样回答”。训练方法和预训练几乎一样，区别只在数据和两个细节：对话模板，以及只对回答算损失。

数据：提示和回答

SFT 的数据是一条条示范对话。比如：

- 提示：“用一句话解释什么是梯度下降。”
- 回答：“沿着损失下降最快的方向，一小步一小步地调整参数。”

这些数据通常由人工撰写，或者由更强的模型生成、再经过人工筛选。覆盖的范围尽量广：问答、写作、翻译、编程、数学、拒绝不当请求……数量一般是几万到几百万条，比预训练的数据少好几个数量级。研究发现，SFT 数据的**质量和多样性**远比数量重要：几千条精心挑选的高质量对话，就能让一个基座模型学会像助手一样说话。这背后的解释是：知识主要是预训练学到的，SFT 主要教的是“说话的格式和风格”。

对话模板

模型的输入是一串 token，怎么让它分清哪些是用户说的、哪些是它自己说的？办法是用一套**对话模板**（chat template），用特殊 token 把对话的结构标出来。以 Llama-3 的风格为例（示意）：

<|start_header_id|>user<|end_header_id|> 问题 <|eot_id|>

<|start_header_id|>assistant<|end_header_id|> 回答 <|eot_id|>

角色标记告诉模型“下面是谁在说话”，`◁eot_id▷` 表示一轮结束。推理时，系统把用户的问题套进模板，在末尾写上“assistant”的角色标记，然后让模型往下生成；模型生成出 `◁eot_id▷` 时，就知道该停了（第 24 讲）。

每个模型都有自己的模板，推理时必须用和训练时一模一样的模板。模板用错了，模型往往会胡言乱语，或者不知道何时停止。

只对回答算损失

SFT 的损失还是交叉熵（第 16 讲），但有一个关键细节：**只在回答部分的 token 上算**。系统提示和用户的问题是给定的输入，模型不需要学着生成它们；如果也算进去，模型会浪费力气去学“用户会怎么提问”，甚至学会替用户说话。实现上很简单：给每个位置一个 0 或 1 的标记，只有标 1 的位置参与损失的平均，这叫**损失掩码**（loss mask）。

下面的演示上半部分是一段对话套上模板的样子，绿色的块算损失；取消勾选，看不加掩码时的情况。下半部分是我们在 ToyLM 上做的一次最小的 SFT，后面会细说。

对话模板，以及只对回答算损失

⇆ 对照

☑ 只对助手的回答算损失

◁begin_of_text▷ 一段文本的开头	◁start_header_id▷system◁end_header_id▷ 角色标记：系统	你是一个乐于助人的助手。 系统提示
◁eot_id▷ 这一轮结束	◁start_header_id▷user◁end_header_id▷ 角色标记：用户	用一句话解释什么是梯度下降。 用户的问题
◁eot_id▷ 这一轮结束	◁start_header_id▷assistant◁end_header_id▷ 角色标记：助手（从这里开始由模型生成）	
沿着损失下降最快的方向，一小步一小步地调整参数。 助手的回答：算损失	◁eot_id▷ 回答结束：也算损失，模型要学会在这里停下	

绿色的部分算损失：模型只学“看到这样的提示，该怎么回答”。系统提示和用户的问题是给定的输入，不需要模型学着去生成它们。

ToyLM 微调前后：提示“你 学”，回答“AI 。”，只训练输出层

前缀	微调前（最可能的两个）	微调后
你 学	推理 0.67，AI 0.33	**AI 1.00，。 0.00**
我 学	AI 0.67，推理 0.33	**AI 1.00，。 0.00**
小猫 吃	鱼 1.00，。 0.00	**鱼 1.00，。 0.00**
我 吃	苹果 0.50，香蕉 0.50	**苹果 0.50，香蕉 0.50**

图 47-1　上：对话模板和损失掩码，只有助手的回答（以及结束标记）算损失。下：ToyLM 微调前后的预测。“你 学”后面从“推理 0.67”变成了“AI 1.00”，“我 学”也跟着变了。

要点

SFT 用“提示—回答”对继续训练模型。对话用模板标出角色和轮次，损失只在回答部分计算。数据的质量和多样性比数量更重要。

在 ToyLM 上做一次 SFT

第 22–24 讲的 ToyLM 是在小语料上预训练过的“基座模型”：看到“你 学”，它认为下一个词是“推理”（0.67）或“AI”（0.33），和语料里的频率一致。

现在我们希望它学会：被问到“你 学”时，回答“AI 。”。SFT 数据只有这一条对话：提示是“你 学”，回答是“AI 。”。拆成训练题时，“你”和“学”这两个位置是提示，不算损失；“你 学 → AI”和“学 AI → 。”这两个位置算损失。为了简单，我们只训练输出层（第 33 讲），学习率 0.5，梯度下降 50 步。

结果：

- **“你 学”**：从“推理 0.67、AI 0.33”变成了“AI 1.00”。学会了。
- **“我 学”**：我们没有教过它，它也从“AI 0.67”变成了“AI 1.00”。
- **“小猫 吃”“我 吃”**：几乎没变。

第二条值得注意。“我 学”和“你 学”在模型内部的表示很相近（第 22 讲，“我”和“你”的嵌入相似），所以只教了“你 学”，“我 学”也被一起改变了。这既是 SFT 能“举一反三”的原因，也是它的风险：微调可能在你没注意的地方改变模型的行为。如果 SFT 的数据太窄、训练太猛，模型还会忘掉预训练时学到的东西，叫**灾难性遗忘**。所以实际的 SFT 学习率通常很小，训练的轮数也不多。

用代码复现

下面的代码在 ToyLM 上实现了这次 SFT：把对话拆成训练题并标出损失掩码，只用算损失的位置训练输出层，最后比较几个前缀在微调前后的预测。

```python
import json
import numpy as np

ex = json.load(open("tinylm.json", encoding="utf-8"))["examples"]["22"]   # 第 22-24 讲在小语料上预训练的 ToyLM
V = ex["vocab"]
E, W1, b1, W2, b2 = (np.array(ex[k]) for k in ["E", "W1", "b1", "W2", "b2"])

def hidden(a, b):                                          # 看最后两个词，得到隐藏层
    return np.maximum(0, np.concatenate([E[V.index(a)], E[V.index(b)]]) @ W1 + b1)

def probs(a, b, W2, b2):
    z = hidden(a, b) @ W2 + b2
    p = np.exp(z - z.max()); return p / p.sum()

def top(a, b, W2, b2, k=2):
    p = probs(a, b, W2, b2)
    return " ".join(f"{V[i]} {p[i]:.2f}" for i in np.argsort(-p)[:k])

# 1. SFT 数据：一条“对话”。提示是“你 学”，希望的回答是“AI 。”
prompt, answer = ["你", "学"], ["AI", "。"]
seq = ["。", "。"] + prompt + answer
pairs = [(seq[i - 2], seq[i - 1], seq[i]) for i in range(2, len(seq))]
mask = [0] * len(prompt) + [1] * len(answer)          # 只有回答部分算损失
for (a, b, y), m in zip(pairs, mask):
    print(f"  输入“{a} {b}”→ 目标“{y}”：{'算损失' if m else '不算（这是提示）'}")

# 2. 微调：只训练输出层，学习率 0.5，梯度下降 50 步
W2f, b2f = W2.copy(), b2.copy()
train = [(a, b, y) for (a, b, y), m in zip(pairs, mask) if m]
for step in range(50):
    gW, gb = np.zeros_like(W2f), np.zeros_like(b2f)
    for a, b, y in train:
        p = probs(a, b, W2f, b2f)
        p[V.index(y)] -= 1                             # 交叉熵对 logit 的梯度：p - y（第 16 讲）
        gW += np.outer(hidden(a, b), p) / len(train)
        gb += p / len(train)
    W2f -= 0.5 * gW
    b2f -= 0.5 * gb

for a, b in [("你", "学"), ("我", "学"), ("小猫", "吃"), ("我", "吃")]:
    print(f"“{a} {b}”→ 微调前：{top(a, b, W2, b2)}  |  微调后：{top(a, b, W2f, b2f)}")
```

输出：

```
  输入“。 。”→ 目标“你”：不算（这是提示）
  输入“。 你”→ 目标“学”：不算（这是提示）
  输入“你 学”→ 目标“AI”：算损失
  输入“学 AI”→ 目标“。”：算损失
“你 学”→ 微调前：推理 0.67 AI 0.33  |  微调后：AI 1.00 推理 0.00
“我 学”→ 微调前：AI 0.67 推理 0.33  |  微调后：AI 1.00 推理 0.00
“小猫 吃”→ 微调前：鱼 1.00 。 0.00  |  微调后：鱼 1.00 。 0.00
“我 吃”→ 微调前：苹果 0.50 香蕉 0.50  |  微调后：苹果 0.50 香蕉 0.50
```

前四行是拆出的训练题和掩码。后四行是微调前后的对比，和演示里的表一致。

到了大模型里

SFT 的计算量很小。和预训练动辄几万亿 token 比，SFT 的数据通常只有几亿到几十亿 token，计算量小几个数量级。但它要更新模型的全部参数，显存需求和预训练一样大（第 19 讲的每参数 16 字节）。这正是第 50 讲 LoRA 要解决的问题。

对话模板影响推理系统。推理服务（比如 OpenAI 兼容的接口）收到的是结构化的消息列表，要在服务端按模型的模板拼成 token 序列。系统提示、多轮对话的历史，每一轮都要重新放进输入，所以多轮对话的提示会越来越长。很多请求共享同一个系统提示，这就是第 76 讲前缀缓存的用武之地。

容易混淆的地方

1. **SFT 主要教格式，不是教知识**。用 SFT 往模型里灌大量新知识，效果往往不好，还容易让模型学会自信地编造。新知识主要靠预训练（或继续预训练）获得。
2. **指令模型和基座模型是两个模型**。很多模型同时发布两个版本，名字里带 Instruct 或 Chat 的是做过 SFT（和后续对齐）的，可以直接对话；基座版本适合在上面继续微调。
3. **掩码遮的是损失，不是注意力**。提示部分的 token 照常参与前向计算、照常被回答部分的 token 看到（因果掩码之内），只是它们自己的“预测下一个 token”不计入损失。

回顾

基座模型只会接着往下写。SFT 用一批“提示—回答”的示范对话继续训练它：用对话模板和特殊 token 标出角色和轮次，用损失掩码只对回答部分计算交叉熵。数据的质量和多样性比数量更重要，因为 SFT 主要教的是说话的格式。在 ToyLM 上，只用一条对话、只训练输出层，就让“你 学”后面从“推理”变成了“AI”，但相似的“我 学”也被一起改变了，这提醒我们微调的影响会扩散。

SFT 教会了模型“像助手一样说话”，但同一个问题可以有很多种回答，哪一种更好、更有帮助、更安全，示范数据很难穷尽。怎么让模型学会人的偏好？下一讲：RLHF 与 DPO。

第48讲 RLHF 与 DPO

轻量线

14 激活函数　16 损失函数　47 监督微调 SFT

SFT 教会了模型像助手一样说话（第 47 讲），但它只能模仿示范数据。同一个问题，可以有很多种都“说得通”的回答：有的更准确，有的更啰嗦，有的语气生硬，有的带着不该说的内容。要把“什么是好回答”全部写成示范，几乎不可能。

但让人**比较**两个回答，判断哪一个更好，却容易得多。这一讲介绍怎么利用这种“二选一”的偏好数据来训练模型，也就是常说的**对齐**（alignment）：经典的 RLHF（基于人类反馈的强化学习），以及更简单、今天很常用的 DPO（直接偏好优化）。

偏好数据

偏好数据的每一条是一个三元组：一个提示，两个回答，以及人的判断“哪个更好”。比如：

- 提示：“我晚上睡不着，怎么办？”
- 回答 A：给出几条具体可行的建议，并提醒长期失眠应就医。
- 回答 B：“多睡就好了。”
- 判断：A 更好。

回答通常由 SFT 之后的模型自己生成（对同一个提示采样两次），再交给标注者比较。比较比打分更可靠：人很难稳定地说出“这个回答值 7.3 分”，却很容易说出“这个比那个好”。

RLHF：先学一个打分员

RLHF 分两步。

第一步，训练一个奖励模型（reward model）。它通常是一个和语言模型结构相同、但输出层换成单个数字的模型：输入提示和回答，输出一个分数 r。怎么用“二选一”的数据训练它？用 **Bradley–Terry 模型**：假设人觉得 A 比 B 好的概率是

$$P(A \succ B) = \sigma(r_A - r_B)$$

σ 是第 14 讲的 Sigmoid。两个分数相等时概率是 0.5；A 的分高出 1，概率是 0.731；高出 3，是 0.953。训练的损失就是对人选中的那个回答取交叉熵（第 16 讲）：$-\ln \sigma(r_{好} - r_{差})$。两个分相等时损失是 $\ln 2 \approx 0.693$，分差越大损失越小。训练完，奖励模型就成了一个“自动的打分员”，能给任何回答打分。

第二步，用强化学习优化语言模型。让语言模型对提示生成回答，奖励模型给回答打分，然后调整语言模型的参数，让高分回答的概率变大。常用的算法叫 PPO。这里有一个重要的约束：不能让模型为了讨好奖励模型而“走火入魔”，比如发现某种奇怪的说法能骗到高分，就一直这样说。所以优化目标里还要减去一项，惩罚模型偏离原来的 SFT 模型太远（用第 16 讲的交叉熵思想衡量的一种“距离”，叫 KL 散度）：

$$\text{最大化：奖励} - \beta \times (\text{和参考模型的差距})$$

RLHF 是 ChatGPT 成功的关键技术之一，但它很复杂：要同时维护语言模型、参考模型、奖励模型，PPO 还需要一个价值模型，四个大模型同时在显存里，训练还很不稳定。

DPO：跳过奖励模型

2023 年提出的 DPO（Direct Preference Optimization）发现，上面那个“奖励减去偏离惩罚”的优化问题，可以通过数学推导，变成一个直接作用在语言模型上的损失函数，**不需要单独训练奖励模型，也不需要强化学习**。

记 π 是正在训练的模型，π_{ref} 是参考模型（SFT 之后的模型，固定不动），y_w 是好回答，y_l 是差回答。定义每个回答的“隐式奖励”为：模型给它的对数概率，比参考模型高出多少，再乘以 β：

$$\text{隐式奖励}(y) = \beta \cdot (\ln \pi(y) - \ln \pi_{\text{ref}}(y))$$

然后套用 Bradley–Terry 的损失：

$$\text{DPO 损失} = -\ln \sigma(\text{隐式奖励}(y_w) - \text{隐式奖励}(y_l))$$

直观上，它要求模型“提高好回答的概率、降低差回答的概率”，而且是**相对于参考模型**的提高和降低。这个“相对”起到了 RLHF 里偏离惩罚的作用：模型不会无节制地偏离原来的样子。它的梯度还有一个很好的性质：前面乘着一个权重 σ(−差值)，好回答和差回答还没被区分开时，权重接近 0.5，推得用力；已经分得很开时，权重接近 0，几乎不推了。

下面的演示左边是奖励模型的偏好概率，右边是 DPO 的损失。拖动滑块，看分差、损失和“推的力度”怎么变。

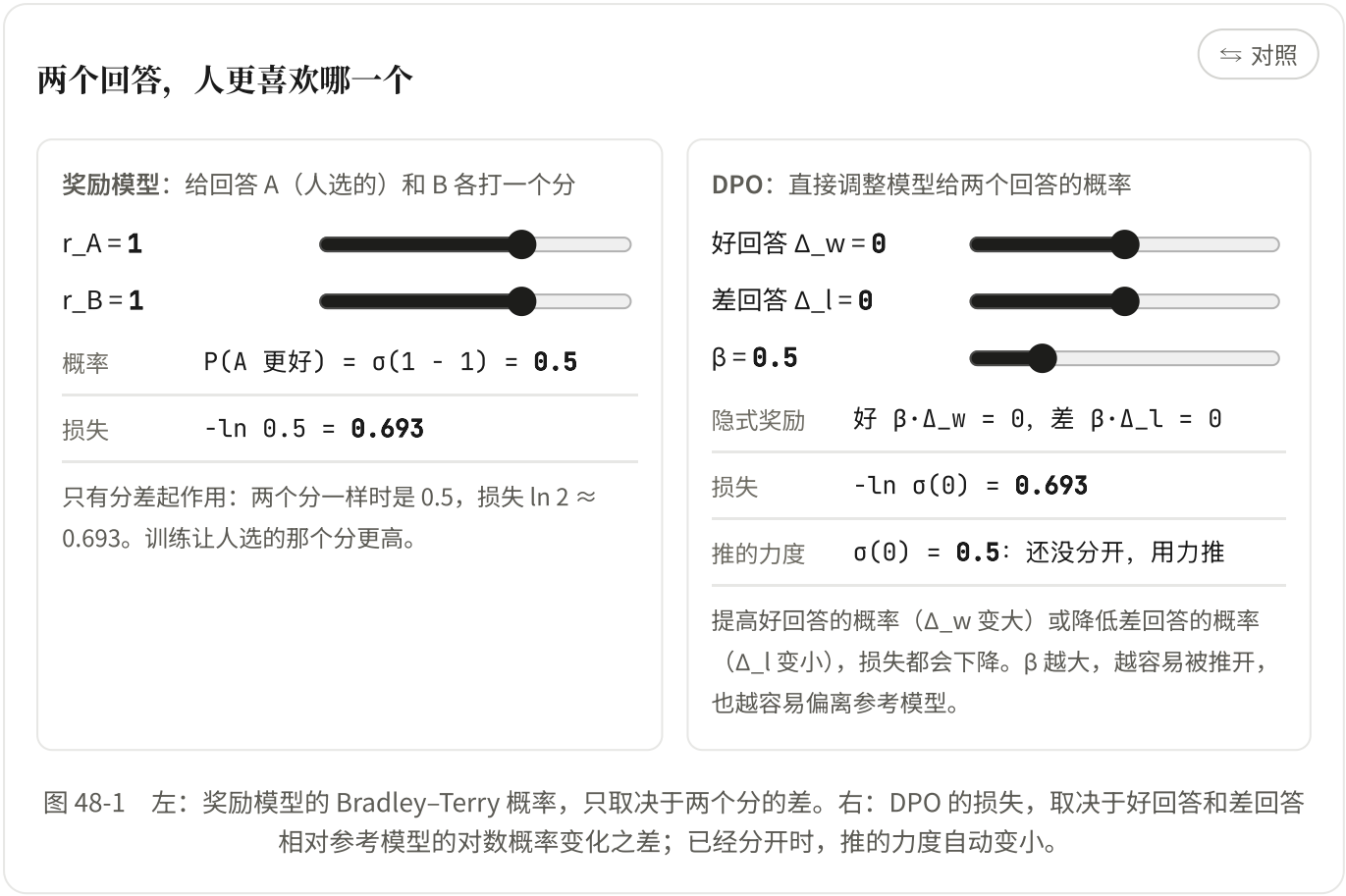


图 48-1　左：奖励模型的 Bradley–Terry 概率，只取决于两个分的差。右：DPO 的损失，取决于好回答和差回答相对参考模型的对数概率变化之差；已经分开时，推的力度自动变小。

要点

偏好学习用“二选一”的数据训练模型。RLHF 先训练奖励模型（$P(A > B) = \sigma(r_A - r_B)$），再用强化学习让模型获得高分、同时不偏离参考模型太远；DPO 把这两步合成一个直接的损失：相对参考模型，提高好回答、降低差回答的概率。

在 ToyLM 上做一次 DPO

ToyLM 看到“我 吃”时，“苹果”和“香蕉”几乎各占一半（0.501 和 0.499），因为语料里两者一样多。假设人更喜欢“苹果 。”。我们用 DPO 训练它（$\beta = 0.5$，只训练输出层），参考模型就是训练前的 ToyLM。

一开始，模型和参考模型完全一样，两个隐式奖励都是 0，损失是 ln 2 = 0.693。训练 10 步后，“苹果”的概率涨到 0.884，损失降到 0.309；30 步后 0.981，损失 0.129。随着两个回答被分开，每一步推的力度越来越小，损失的下降也越来越慢。

用代码复现

下面的代码先算了几组 Bradley–Terry 的概率和损失；然后在 ToyLM 上实现 DPO：计算两个回答在策略模型和参考模型下的对数概率，求 DPO 损失，按梯度更新输出层。

```python
import json
import numpy as np

ex = json.load(open("tinylm.json", encoding="utf-8"))["examples"]["22"]
V = ex["vocab"]
E, W1, b1, W2, b2 = (np.array(ex[k]) for k in ["E", "W1", "b1", "W2", "b2"])
sig = lambda z: 1 / (1 + np.exp(-z))

def hidden(a, b):
    return np.maximum(0, np.concatenate([E[V.index(a)], E[V.index(b)]]) @ W1 + b1)

def logp_seq(prompt, answer, W2, b2):
    """一个回答在给定提示下的对数概率：逐个 token 的 ln P 相加；同时返回对输出层的梯度。"""
    seq = ["。", "。"] + prompt + answer
    lp, gW, gb = 0.0, np.zeros_like(W2), np.zeros_like(b2)
    for i in range(2 + len(prompt), len(seq)):
        h = hidden(seq[i - 2], seq[i - 1])
        z = h @ W2 + b2
        p = np.exp(z - z.max()); p /= p.sum()
        y = V.index(seq[i])
        lp += np.log(p[y])
        g = -p; g[y] += 1                              # ln P(y) 对 logit 的梯度：
onehot - p
        gW += np.outer(h, g); gb += g
    return lp, gW, gb

# 1. 奖励模型的偏好概率（Bradley-Terry）：P(A 比 B 好) = σ(r_A - r_B)
for rA, rB in [(1.0, 1.0), (2.0, 1.0), (3.0, 0.0)]:
    print(f"r_A = {rA}, r_B = {rB}：P(A 更好) = {sig(rA - rB):.3f}，损失 -ln σ = {-
np.log(sig(rA - rB)):.3f}")

# 2. DPO：提示“我 吃”，人更喜欢“苹果 。”（chosen），不喜欢“香蕉 。”（rejected）
prompt, yw, yl, beta = ["我", "吃"], ["苹果", "。"], ["香蕉", "。"], 0.5
ref_w, _, _ = logp_seq(prompt, yw, W2, b2)             # 参考模型 = 微调前的模型，固定不
动
ref_l, _, _ = logp_seq(prompt, yl, W2, b2)
W, b = W2.copy(), b2.copy()
for step in range(31):
    lw, gWw, gbw = logp_seq(prompt, yw, W, b)
    ll, gWl, gbl = logp_seq(prompt, yl, W, b)
    margin = beta * ((lw - ref_w) - (ll - ref_l))    # 隐式奖励之差
    loss = -np.log(sig(margin))
    if step in (0, 10, 30):
        print(f"第 {step:2d} 步：P(苹果) = {np.exp(lw):.3f}，P(香蕉) =
{np.exp(ll):.3f}，DPO 损失 {loss:.3f}")
    coef = beta * (1 - sig(margin))                    # 损失对 margin 的梯度（取负），越
```

```
“没学会”越大
    W += 0.01 * coef * (gWw - gWl)
    b += 0.01 * coef * (gbw - gbl)

z = hidden("我", "吃") @ W + b; p = np.exp(z - z.max()); p /= p.sum()
print(""我 吃”之后的分布: ", {V[i]: round(float(p[i]), 3) for i in np.argsort(-p)
[:3]})
```

输出：

```
r_A = 1.0, r_B = 1.0: P(A 更好) = 0.500, 损失 -ln σ = 0.693
r_A = 2.0, r_B = 1.0: P(A 更好) = 0.731, 损失 -ln σ = 0.313
r_A = 3.0, r_B = 0.0: P(A 更好) = 0.953, 损失 -ln σ = 0.049
第  0 步: P(苹果) = 0.501, P(香蕉) = 0.499, DPO 损失 0.693
第 10 步: P(苹果) = 0.884, P(香蕉) = 0.116, DPO 损失 0.309
第 30 步: P(苹果) = 0.981, P(香蕉) = 0.018, DPO 损失 0.129
“我 吃”之后的分布:  {'苹果': 0.983, '香蕉': 0.017, '骨头': 0.0}
```

前三行：分差 0、1、3 时，偏好概率 0.5、0.731、0.953，损失 0.693、0.313、0.049。接下来三行是 DPO 训练的过程。最后一行：训练后“我 吃”之后，“苹果”的概率是 0.983。

到了大模型里

显存是主要麻烦。RLHF 的 PPO 要同时用到策略模型、参考模型、奖励模型和价值模型，大模型规模下要几倍于 SFT 的显存。DPO 只需要策略模型和参考模型，而且参考模型只用来算对数概率（只做前向），可以事先算好存起来，简单得多。这是 DPO 流行的主要原因。

RL 训练里大量时间花在推理上。PPO 这类方法每一步都要让模型生成大量回答，这部分是纯粹的推理（自回归解码），常常占了训练时间的大头。所以现代的 RL 训练框架会直接调用 vLLM 等推理引擎来生成，推理的效率直接决定了 RL 训练的速度。第 49 讲的推理模型训练，这一点更加突出。

奖励模型也是一种模型服务。RLHF 训练时，奖励模型要给大量回答打分；在线上，有的系统也会用奖励模型对多个候选回答排序，选出最好的那个返回。

容易混淆的地方

1. **“对齐”不只是“安全”**。偏好数据既包括“不要说有害的话”，也包括“更有帮助、更准确、格式更好”。很多时候它主要在提升回答的整体质量。

2. **奖励模型会被“钻空子”**。奖励模型只是人类偏好的一个近似，模型可能学会利用它的漏洞（比如回答越长分越高，就越写越长），这叫奖励投机（reward hacking）。偏离惩罚和 DPO 的“相对参考模型”都是为了限制这种情况。
3. **DPO 不是强化学习，但目标相同**。DPO 的损失是从 RLHF 的目标推导出来的，两者优化的是同一件事，只是 DPO 用“离线”的偏好数据直接训练，不需要在训练中生成新回答。

回顾

SFT 只能模仿示范，偏好学习让模型从“二选一”的比较中学到什么是更好的回答。RLHF 先用 Bradley–Terry 模型训练一个奖励模型，$P(A > B) = \sigma(r_A - r_B)$，再用 PPO 等强化学习方法提高模型的得分，同时惩罚它偏离参考模型太远。DPO 把这两步合成一个直接的损失：相对参考模型，提高好回答的概率、降低差回答的概率，不需要奖励模型和强化学习。在 ToyLM 上，30 步 DPO 就把“苹果”的概率从 0.501 推到了 0.981。

偏好学习教模型说得“更好”。可还有一类问题，好坏不需要人来判断：数学题有标准答案，代码可以跑测试。如果直接用“答对没有”作为奖励来训练，模型会学会什么？下一讲：**推理模型与强化学习**。

第49讲 推理模型与强化学习

轻量线

24 自回归生成　47 监督微调 SFT　48 RLHF 与 DPO

问一个普通的对话模型一道复杂的数学题，它常常张口就给出答案，而且经常是错的。人做难题时不会这样：先在草稿纸上列式子、试几种思路、发现错了再回头检查，最后才写答案。

2024 年以后出现的一类模型，比如 OpenAI 的 o1 和 DeepSeek-R1，学会了这样做：回答之前先生成一长段“思考过程”，再给出最终答案。它们叫**推理模型**（reasoning model）。在数学、编程这类有明确对错的任务上，它们的表现大幅超过普通模型。这一讲介绍它们是怎么训练出来的：核心是**用“答对没有”作为奖励的强化学习**，以及这件事对推理系统意味着什么。

想得越多，答得越好

一个简单的事实是：让模型“多想一会儿”能提高正确率。早在推理模型之前，人们就发现，在提示里加一句“让我们一步一步地想”，模型先写出推理步骤再回答，数学题的正确率就会提高，这叫**思维链**（chain of thought）。

为什么有用？可以从第 24 讲的自回归生成来理解：模型每生成一个 token，都只做一次固定量的计算（一次前向）。直接回答，等于要求它用一次前向就“算出”答案；写出中间步骤，每一步都是新的 token，都多了一次前向，而且后面的步骤可以“看到”前面写下的中间结果。思考的 token 越多，模型用在这道题上的计算就越多。

另一种“多花计算”的办法是**多次采样**：同一道题让模型独立回答好几次，取出现最多的答案（多数投票）。下面的代码模拟了一下：如果单次答对的概率是 0.6，答错时随机错成几种不同的答案，那么采样 5 次投票，正确率升到 0.771；采样 15 次，到 0.981。

这些方法都在推理时花更多的计算换更好的结果，叫**推理时扩展**（test-time scaling）。推理模型更进一步：通过训练，让模型自己学会在需要的时候想得很长、很仔细。

可验证的奖励

怎么训练出会“想”的模型？如果用 SFT，需要大量人写的长推理过程，又贵又难写。推理模型主要用的是强化学习，而且奖励不再来自奖励模型（第 48 讲），而是来自**可以自动验证的结果**：

- 数学题：最终答案和标准答案是否一致。
- 编程题：代码能否通过测试用例。

只看结果对不对，不管过程怎么写。模型可以自由地探索各种“思考方式”，只要最后答对了，这种思考方式就被鼓励。这叫**基于可验证奖励的强化学习**（RLVR）。DeepSeek-R1 的论文报告了一个引人注目的现象：只用这样的奖励，模型在训练中自发地学会了越想越长、会回头检查、会说“等等，这里不对”。

GRPO：组内互相比较

具体怎么用奖励更新模型？DeepSeek 使用的算法叫 **GRPO**（Group Relative Policy Optimization）。它的想法很简单：

1. 对同一道题，让模型采样一组回答，比如 8 个。
2. 自动判分：答对的奖励 1，答错的 0。
3. 每个回答的**优势**（advantage）= (它的奖励 − 这组的平均奖励) / 这组奖励的标准差。
4. 优势为正的回答，提高它的生成概率；为负的，降低它的概率。

关键在第 3 步：好坏是**和同组的其他回答比**出来的。8 个回答答对 3 个，平均奖励 0.375，标准差 0.484；答对的回答优势是 +1.291，答错的是 −0.775。答对的越少，每个答对的回答优势越大，被推得越用力，这很合理：难题上偶然答对的那条思路，最值得学习。

如果 8 个全对或全错，每个回答都等于平均，优势全是 0，这道题对训练**没有任何贡献**。所以训练题的难度要合适：太简单（全对）或太难（全错）的题都学不到东西。

和 PPO 相比，GRPO 不需要单独训练一个价值模型来估计“平均水平”，直接用同组回答的平均来代替，省下了一个大模型的显存和计算。

下面的演示是一道题的 8 个回答（示意）。点任意一个回答，切换它的对错，看平均、标准差和每个回答的优势怎么变。

同一道题答 8 次，按组内排名调整

⇆ 对照

题目：篮子里有 37 个苹果，又放进 25 个，现在一共有多少个？ 正确答案：62

37 + 25：个位 7 + 5 = 12，写 2 进 1；十位 3 + 2 + 1 = 6。答案 62。	✓ 1		1.291
37 + 25 = 52。答案 52。	✗ 0		-0.775
先算 37 + 20 = 57，再加 5，得 61。答案 61。	✗ 0		-0.775
37 + 25 = 37 + 3 + 22 = 40 + 22 = 62。答案 62。	✓ 1		1.291
3 和 2 是 5，7 和 5 是 12，拼起来是 512。答案 512。	✗ 0		-0.775
苹果的数量是 37 + 25 = 63。答案 63。	✗ 0		-0.775
37 接近 40，40 + 25 = 65，再减 3 是 62……等等，再检查一下，65 − 3 = 62。答案 62。	✓ 1		1.291
大概六十个左右。答案 60。	✗ 0		-0.775

8 个里答对 3 个：平均奖励 0.375，标准差 0.484。
答对的回答优势 1.291，会被推高（提高生成它的概率）；答错的 -0.775，会被压低。答对的越少，每个答对的回答被推得越用力。
点任意一个回答，切换它的对错。

图 49-1　GRPO 的组内优势。默认 8 个答对 3 个：答对的被推高（+1.291），答错的被压低（−0.775）。切换成全对或全错，优势全变成 0。

要点

推理模型靠强化学习学会“先想再答”：奖励来自可以自动验证的结果（答案对不对、代码过不过测试）。GRPO 对同一道题采样一组回答，用组内的平均和标准差算每个回答的优势，好于平均的被推高，差于平均的被压低。

用代码复现

下面的代码算了一组回答的 GRPO 优势（以及全对、全错的情况）；算了 pass@k（n 次采样里有 c 次对，随机挑 k 次至少有一次对的概率）；模拟了多数投票；最后估算“先想再答”的推理成本。

```
from math import comb
import numpy as np

# 1. GRPO：同一道题采样 8 个回答，答对奖励 1，答错 0；优势 =（奖励 - 组内平均）/ 组内标准差
rewards = np.array([1, 0, 0, 1, 0, 0, 0, 1], dtype=float)
adv = (rewards - rewards.mean()) / (rewards.std() + 1e-6)
print("奖励：", rewards.astype(int).tolist(), " 平均", rewards.mean(), " 标准差",
round(float(rewards.std()), 3))
print("优势：", adv.round(3).tolist())
for r in [np.ones(8), np.zeros(8)]:
    print(f"全{'对' if r[0] else '错'}：优势", ((r - r.mean()) / (r.std() + 1e-
6)).round(3).tolist(), "→ 这道题不提供任何信号")

# 2. pass@k：n 次采样里有 c 次答对，随便挑 k 次至少有一次对的概率
def pass_at_k(n, c, k):
    return 1 - comb(n - c, k) / comb(n, k) if n - c ⩾ k else 1.0
print("n = 8, c = 3：", {k: round(pass_at_k(8, 3, k), 3) for k in [1, 2, 4, 8]})

# 3. 多数投票：每次独立答对的概率 p = 0.6（答错时随机错成 4 种答案之一），采样 k 次取出现最多
的答案
rng = np.random.default_rng(49)
def majority(p, k, trials=20000):
    right = 0
    for _ in range(trials):
        ans = np.where(rng.random(k) < p, 0, rng.integers(1, 5, k))
        vals, cnt = np.unique(ans, return_counts=True)
        right += vals[cnt.argmax()] == 0 and (cnt == cnt.max()).sum() == 1
    return float(right / trials)
print("单次正确率 0.6，多数投票：", {k: round(majority(0.6, k), 3) for k in [1, 5,
15]})

# 4. 推理成本：先想 8000 个 token 再回答 500 个，和直接回答 500 个比
think, ans = 8000, 500
print(f"生成的 token 多了 {(think + ans) / ans:.0f} 倍；Llama-3-8B 的 KV Cache 从
{ans * 128 / 1024:.1f} MiB 涨到 {(think + ans) * 128 / 1024:.0f} MiB")
```

输出：

```
奖励：  [1, 0, 0, 1, 0, 0, 0, 1]  平均 0.375  标准差 0.484
优势：  [1.291, -0.775, -0.775, 1.291, -0.775, -0.775, -0.775, 1.291]
全对：优势 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] → 这道题不提供任何信号
全错：优势 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] → 这道题不提供任何信号
n = 8, c = 3：  {1: 0.375, 2: 0.643, 4: 0.929, 8: 1.0}
单次正确率 0.6，多数投票：  {1: 0.602, 5: 0.771, 15: 0.981}
生成的 token 多了 17 倍；Llama-3-8B 的 KV Cache 从 62.5 MiB 涨到 1062 MiB
```

前四行是 GRPO 的优势，和演示一致。第五行是 pass@k：8 次里对 3 次，挑 1 次对的概率 0.375，挑 4 次至少一次对的概率 0.929。第六行是多数投票。最后一行是下一节的成本估算。

到了大模型里

推理模型很“费 token”。一次回答先想 8000 个 token、再答 500 个，生成的 token 是直接回答的 17 倍。第 24 讲说过，decode 一次只能出一个 token，每个都要把权重读一遍，所以响应时间也接近 17 倍。KV Cache 随之增长：Llama-3-8B 的规模下，500 个 token 是 62.5 MiB，8500 个 token 就超过 1 GiB。推理模型让长序列的 decode 成了推理服务的常态，第三章的 KV Cache 管理、长序列优化、投机解码等技术因此更加重要。

RL 训练的大头是推理。GRPO 每一步都要对每道题生成一组长长的回答，这部分是纯粹的自回归解码，往往占了 RL 训练时间的大部分。所以推理模型的训练框架会把 vLLM、SGLang 这样的推理引擎嵌进训练流程，推理效率直接决定训练效率。训练时的“生成”和“更新参数”还要在不同的 GPU 上配合，权重要在两边同步，这也是新的系统工程问题。

思考可以控制。很多推理模型允许设置“思考预算”（最多想多少 token），或在“快速回答”和“深度思考”之间切换。推理服务按 token 计费，思考的 token 也要付钱。

容易混淆的地方

1. **“推理模型”的“推理”指 reasoning，不是 inference**。中文里两个都叫“推理”：这门课的“推理优化”说的是 inference（运行模型），这一讲的“推理模型”说的是 reasoning（思考、推导）。
2. **思考过程不一定可信**。模型写出的思考过程是它生成的文本，不一定忠实地反映了它内部是怎么得到答案的。奖励只看最终答案，并不保证中间每一步都对。
3. **可验证奖励不是万能的**。写作、对话这类没有标准答案的任务，还是要靠第 48 讲的偏好学习。实际的推理模型往往把两种方法结合起来用。

回顾

让模型在回答前多生成一些“思考” token，或者多次采样再投票，都能用更多的推理计算换更高的正确率。推理模型通过强化学习把这种能力内化：奖励来自可以自动验证的结果，模型在训练中自发学会想得更长、更会检查。GRPO 对同一道题采样一组回答，用组内平均和标准差算优势，答对的推高、答错的压低；全对或全错的题不提供信号。推理模型生成的 token 动辄是普通回答的十几倍，让长序列的 decode 成为推理系统的主要负担；而它们的 RL 训练，大部分时间也花在推理上。

前面几讲的训练都要更新模型的全部参数，显存需求和预训练一样大。如果只是想让一个大模型适应自己的任务，有没有办法只训练很少的参数？下一讲：**LoRA 参数高效微调**。

第50讲 LoRA 参数高效微调

轻量线

13 神经元与线性层　19 优化器　47 监督微调 SFT

第 47–49 讲的 SFT、DPO、强化学习，默认都要更新模型的全部参数。第 19 讲算过，用 Adam 和混合精度训练，每个参数要 16 字节：Llama-3-8B 全量微调光是这些就要 128.5 GB，一张 80 GB 的卡放不下。可很多时候，我们只是想让一个通用模型适应某个具体的任务：用公司内部的文档风格写作、按固定的格式抽取信息。为此动用全部 80 亿个参数，太浪费了。

LoRA（Low-Rank Adaptation，低秩适配，2021 年提出）是最常用的**参数高效微调**方法：原来的权重一个都不动，只在旁边加上很少的新参数来训练。Llama-3-8B 用 LoRA 微调，可训练的参数可以只有 0.5% 左右。这一讲看它的原理、能省多少，以及它给推理系统带来的一个新机会。

冻结 W，只学一个“小修改”

微调本质上是给每个权重矩阵加一个修改量：W 变成 W + ΔW。全量微调直接学 ΔW，它和 W 一样大（比如 4096 × 4096）。

LoRA 的假设是：**微调需要的修改量 ΔW 是“低秩”的**，可以写成两个瘦长矩阵的乘积：

$$\Delta W = A \cdot B, \qquad A\text{: } \mathrm{d_{in}} \times \mathrm{r}, \qquad B\text{: } \mathrm{r} \times \mathrm{d_{out}}$$

r 叫**秩**（rank），远小于 d，常用 8、16、64。第 40 讲的 MLA 用的也是“先压到低维、再展开”的同一个想法。前向计算时，原来的 W 冻结不动，旁边多走一条“小路”：

$$\mathbf{y} = \mathbf{x}\,W + (\mathbf{x}\,A)\,B$$

x 先乘 A 压到 r 维，再乘 B 展开回 $\mathrm{d_{out}}$ 维，加到原来的输出上。训练时只更新 A 和 B。B 初始化为全 0，所以一开始这条小路的输出是 0，模型和原来完全一样，微调从原模型平稳地出发。

参数省了多少？W 有 $\mathrm{d_{in}} \times \mathrm{d_{out}}$ 个参数，A 和 B 合起来只有 $\mathrm{r} \times (\mathrm{d_{in}} + \mathrm{d_{out}})$ 个。以 4096 × 4096 的 q_proj、r = 16 为例：W 有 1678 万个参数，A、B 只有 13.1 万个，不到 1%。

为什么这样就够了？LoRA 论文的实验发现，微调带来的权重变化确实集中在少数几个方向上：预训练已经学会了绝大部分东西，适应一个新任务只需要在少数几个方向上做调整。

能省多少

给 Llama-3-8B 每层的 7 个线性层（q、k、v、o、gate、up、down）都加上 LoRA：

秩 r	可训练参数	占全模型
8	2100 万	0.26%
16	4194 万	0.52%
64	1.68 亿	2.09%

显存的账更直观。全量微调要 128.5 GB（还不含激活值）。LoRA 微调时，原模型的权重冻结，只要用 BF16 存一份（16.1 GB），不需要它们的梯度和 Adam 状态；只有那 4194 万个 LoRA 参数需要每个 16 字节，0.67 GB。合起来不到 17 GB，一张消费级的大显存显卡就能放下。

更进一步，**QLoRA** 把冻结的原模型量化成 4 位（第 12 篇），权重只要约 4 GB，LoRA 部分仍用较高的精度训练。这让在单张普通显卡上微调几十亿甚至上百亿参数的模型成为可能。

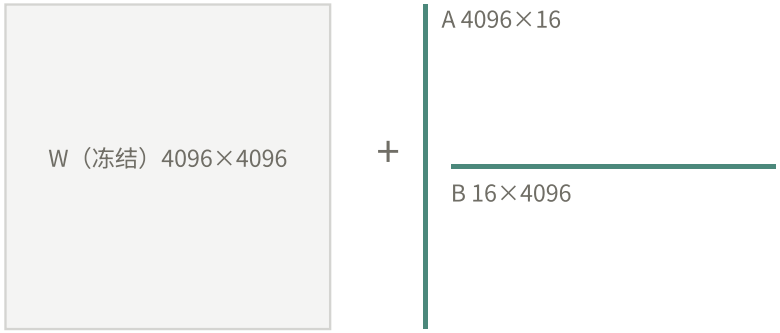
下面的计算器可以选一个线性层、调秩 r，看参数和显存；图上按比例画出了 W、A、B 的形状：W 是一个大方块，A 和 B 是两条细长的边。

调秩 r：LoRA 要训练多少参数

⇆ 对照

矩阵 q_proj 4096×4096 | k_proj 4096×1024 | up_proj 4096×14336

秩 r = **16**



这个矩阵	全模型（32 层 × 7 个线性层）	微调显存（不含激活）
131,072	**41.94M**	**16.7 GB**
r × (d_in + d_out) = 16 × 8192，是 W（16.78M）的 0.78%	占 Llama-3-8B 的 0.52%	冻结的 BF16 权重 16.1 GB + LoRA 参数 × 16 字节；全量微调要 128.5 GB

图 50-1　LoRA 的参数量。W 冻结不动，只训练细长的 A 和 B。r = 16 时，全模型 7 种线性层加起来的 LoRA 参数只占 0.52%，微调显存从 128.5 GB 降到约 17 GB。

要点

LoRA 冻结原权重 W，在旁边训练两个瘦长矩阵 A（d_in × r）和 B（r × d_out），用 A · B 作为权重的修改量。可训练参数从 d_in × d_out 降到 r × (d_in + d_out)，只需为它们存梯度和优化器状态。

训练完以后：合并，或者不合并

LoRA 训练完，推理时有两种用法。

合并：直接算出 W + A·B，替换原来的 W。合并后的模型和原模型结构完全一样，推理没有任何额外开销。下面的代码验证了，“旁路计算”和“合并后计算”的结果完全相同。

不合并：原模型保持不动，A 和 B 作为一个很小的“适配器”单独存放。一个 r = 16 的适配器只有几十 MB，而原模型有十几 GB。这带来了一个有用的可能：**一个基座模型，挂很多个适配器**，每个用户、每个任务一个。

用代码复现

下面的代码在 TinyLM 的一个 4 × 8 矩阵上演示 LoRA：B 初始为 0 时输出不变；A·B 的秩是 r；旁路计算和合并后计算结果相同。然后算出 Llama-3-8B 在不同秩下的 LoRA 参数，以及三种微调方式的显存。

```
import json
import numpy as np

t = json.load(open("tinylm.json", encoding="utf-8"))
W = np.array(t["weights"]["layers"][0]["W_1"])          # TinyLM 第 1 层 FFN 的升维矩阵，4×8
X = np.array(t["examples"]["03"]["X"])                   # “我 爱 学 AI”的嵌入，4×4

# 1. LoRA：W 不动，旁边加两个小矩阵 A（4×r）和 B（r×8），更新量是 A·B
r = 1
rng = np.random.default_rng(50)
A = rng.standard_normal((4, r)) * 0.5
B = np.zeros((r, 8))                                     # B 从 0 开始：一开始完全不改变模型
print("初始时输出不变：", np.allclose(X @ W + (X @ A) @ B, X @ W))
B = rng.standard_normal((r, 8)) * 0.5                    # 假设训练之后 B 变成了这样
delta = A @ B
print("参数：W", W.size, "个；A、B 共", A.size + B.size, "个；A·B 的秩：", np.linalg.matrix_rank(delta))

# 2. 两种算法一样：先算 x·A 再乘 B（训练时），或把 A·B 加进 W 再乘（推理时“合并”）
y_side = X @ W + (X @ A) @ B
y_merged = X @ (W + delta)
print("旁路算 = 合并后算：", np.allclose(y_side, y_merged))

# 3. Llama-3-8B：每层 7 个线性层都加 LoRA，r = 16
shapes = {"q": (4096, 4096), "k": (4096, 1024), "v": (4096, 1024), "o": (4096, 4096),
          "gate": (4096, 14336), "up": (4096, 14336), "down": (14336, 4096)}
N = 8_030_261_248
for r in [8, 16, 64]:
    lora = 32 * sum(r * (a + b) for a, b in shapes.values())
    print(f"r = {r:2d}：LoRA 参数 {lora / 1e6:6.1f}M，占全模型 {lora / N:.2%}")

# 4. 显存：全量微调（每参数 16 字节，第 19 讲） vs LoRA（权重冻结用 BF16，只有 LoRA 参数要梯度和 Adam）
lora = 32 * sum(16 * (a + b) for a, b in shapes.values())
print(f"全量微调：{N * 16 / 1e9:.1f} GB；LoRA：冻结权重 {N * 2 / 1e9:.1f} GB + LoRA {lora * 16 / 1e9:.2f} GB；"
      f"QLoRA（权重 4 位）：{N * 0.5 / 1e9:.1f} GB + {lora * 16 / 1e9:.2f} GB（都不含激活值）")
```

输出：

```
初始时输出不变： True
参数：W 32 个；A、B 共 12 个；A·B 的秩： 1
旁路算 = 合并后算： True
r = 8：LoRA 参数 21.0M，占全模型 0.26%
r = 16：LoRA 参数 41.9M，占全模型 0.52%
r = 64：LoRA 参数 167.8M，占全模型 2.09%
全量微调：128.5 GB；LoRA：冻结权重 16.1 GB + LoRA 0.67 GB；QLoRA（权重 4 位）：4.0 GB +
0.67 GB（都不含激活值）
```

前三行：一开始输出不变；r = 1 时 A、B 共 12 个参数，A·B 的秩是 1；两种算法一致。接下来三行是不同秩下的参数量，和上面的表一致。最后一行是显存：全量 128.5 GB，LoRA 16.1 + 0.67 GB，QLoRA 4.0 + 0.67 GB。

到了大模型里

多 LoRA 服务。如果一家公司有 100 个客户，每个都微调了自己的模型，全量微调就要部署 100 份 16 GB 的模型；用 LoRA，只需要部署一份基座模型，再加 100 个几十 MB 的适配器。推理时，同一批请求可能属于不同的适配器：基座模型的大矩阵乘法所有请求一起算，各自的 (xA)B 小路分别算。S-LoRA、Punica 等工作专门设计了这种“一批里混着多个 LoRA”的高效算子，vLLM 等推理引擎也支持同时加载多个 LoRA 适配器。

小路的计算很便宜，但不是免费。不合并时，每个线性层多了两次小矩阵乘法。r 很小，计算量可以忽略，但多一次读写和算子调用，对延迟有一点影响。只服务一个适配器时，通常直接合并。

LoRA 不只用于微调对话。它也广泛用于图像生成模型等其他领域，是今天最普遍的定制大模型的方式。

容易混淆的地方

1. **LoRA 不改变推理的计算量（合并后）**。它省的是训练时的显存和存储，合并后的模型和原模型一样大、一样快。
2. **秩不是越大越好**。r 越大，能表达的修改越丰富，但参数和显存也越多，在小数据上还更容易过拟合。很多任务用 r = 8 或 16 就够了。
3. **LoRA 学不进大量新知识**。和 SFT 一样（第 47 讲），它擅长调整模型的行为和格式；要灌入大量新知识，低秩的修改往往不够。

回顾

全量微调要为每个参数存梯度和优化器状态，Llama-3-8B 需要 128.5 GB。LoRA 冻结原权重，在旁边训练两个瘦长矩阵 A（$d_{in} \times r$）和 B（$r \times d_{out}$），用它们的乘积作为修改量；B 初始化为 0，从原模型平稳出发。$r = 16$ 时，Llama-3-8B 的可训练参数只占 0.52%，微调显存降到约 17 GB；QLoRA 把原模型量化到 4 位，进一步降到约 5 GB。训练完可以把 A·B 合并进 W，推理零开销；也可以不合并，一个基座模型挂很多个适配器，同时服务不同的任务。

模型训练好了、微调好了，怎么知道它到底好不好？比另一个模型强还是弱？第 6 篇的最后一讲：**模型评测**。

第51讲 模型评测

轻量线

16 损失函数　23 预测下一个 token　49 推理模型与强化学习

每次有新模型发布，都会附上一张表：在某某测试上得了多少分，比上一代、比竞争对手高了多少。这些分数是怎么来的？一个模型“好”，到底指什么？

这一讲是第 6 篇的最后一讲，介绍评测大模型的几类主要方法，以及每一类在衡量什么、有什么陷阱。对做推理系统的人来说，评测还有一个更直接的用处：每做一次优化（量化、换算子、改解码策略），都要用评测确认模型没有变笨。

困惑度：最原始的尺子

第 16 讲的困惑度，是衡量语言模型最直接的指标：在一份标准文本上，模型平均每一步相当于在几个词里犹豫。它不需要任何人工标注，算起来也快，非常适合比较**同一个模型的不同版本**，比如量化前后（第 16 讲提过，量化论文最常用的就是 WikiText-2 上的困惑度）。

但困惑度有明显的局限：它只衡量“预测下一个 token”有多准，不直接告诉你模型能不能解数学题、会不会写代码、回答是否有帮助。而且不同分词器切出的 token 不同，不同模型的困惑度不能直接比。

基准测试：考试

更常用的是**基准测试**（benchmark）：准备一套有标准答案的题，让模型做，算正确率。按题型和判分方式，大致分几类：

- **选择题**：比如 MMLU，覆盖 57 个学科的多项选择题。判分常用的办法是：把每个选项接在题目后面，看模型给哪个选项的概率最高，就算它选了哪个。这不需要模型生成任何东西。
- **答案匹配**：比如 GSM8K（小学数学应用题），让模型生成完整的解答，提取最终答案，和标准答案比是否一致。
- **代码测试**：比如 HumanEval，让模型写函数，用测试用例检验，报告 pass@k（第 49 讲：采样 k 次，至少一次通过的概率）。

下面的演示用 ToyLM 做了一个 5 道题的“选择题基准”：每道题给出一个前缀和 4 个选项，模型给哪个选项的概率最高就算选哪个。点一道题，看各选项的概率。

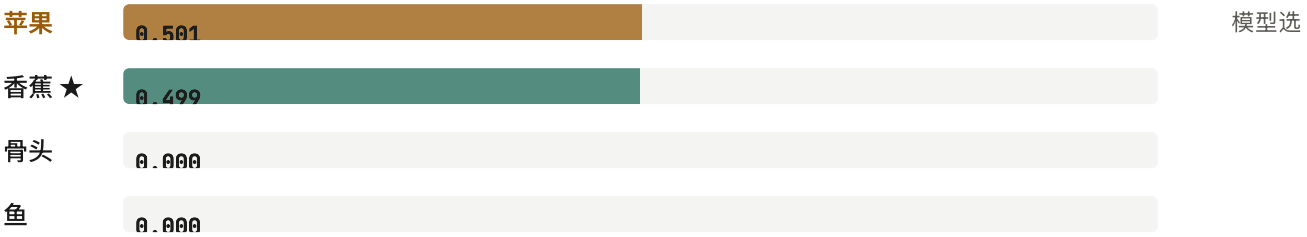
选择题评测，以及对战排名

⇆ 对照

选择题：模型给哪个选项的概率最高，就算它选了哪个

1. “小猫 吃 __” ✓对 2. “小狗 吃 __” ✓对 3. “你 学 __” ✓对 4. “我 喜欢 __” ✓对

5. “我 吃 __” ✕错



★ 是标准答案。5 道题对 4 道，准确率 80%。
这道题的两个主要选项概率只差 0.002：在训练语料里它们本来就一样常见，“标准答案”其实是题目设计者随意定的。好的评测题要有明确、唯一的答案。

Elo 对战排名：用户看两个匿名模型的回答，投票选更好的

模型 A 赢 | 模型 B 赢 | 重置

模型 A：**1200.0** 分，模型 B：**1000.0** 分（已比 0 场）。按分差，A 赢的预期概率是 0.760。
爆冷（分低的赢了）时，分数变化大；赢了本该赢的，分数只变一点。比赛多了，分数会收敛到反映真实水平的位置。

图 51-1　上：ToyLM 做 5 道选择题，对 4 道，准确率 80%。后两道题的两个主要选项概率几乎相同，对错只差千分之几。下：Elo 对战排名，点“A 赢”“B 赢”看分数怎么变。

前三道题，模型都很确定：“小猫 吃”后面是“鱼”的概率 1.000。后两道题却很微妙。“我 喜欢 __”，“小狗”0.335、“小猫”0.332，模型选了“小狗”，碰巧对了；“我 吃 __”，“苹果”0.501、“香蕉”0.499，模型选了“苹果”，答案是“香蕉”，错了。准确率 4/5 = 80%。

这两道题暴露了基准测试的一个本质问题：在 ToyLM 的训练语料里，“我喜欢小猫”和“我喜欢小狗”一样多，“标准答案”其实是出题人随意定的。真实的基准里也有这样的问题：答案有歧义、题目本身有错、选项的措辞让模型偏向某一个。好的基准要尽量让每道题有明确、唯一的正确答案，而解读分数时，要知道几个百分点的差距可能来自这些噪声。

题目泄露

基准测试还有一个越来越严重的问题：**数据污染**（contamination）。基准测试的题目和答案大多公开在网上，而预训练的数据又是从网上抓的（第 45 讲）。如果测试题混进了训练数据，模型就可能“背过答案”，分数虚高，却不代表真实的能力。

对策包括：在训练数据里查重、剔除测试题；定期发布新的、没公开过的题目；用私有的测试集。看到一个模型在某个老基准上分数异常地高，要多一个心眼。

开放式回答：让裁判打分

写作、对话、解释这类任务没有标准答案，没法用正确率衡量。常用的办法有两种。

模型当裁判（LLM-as-a-judge）：让一个很强的模型阅读回答，按照给定的标准打分，或者在两个回答里挑更好的一个。它便宜、快，但裁判模型也有偏好：比如偏爱更长的回答、偏爱和自己风格相近的回答。

人类对战排名：最有影响的是 Chatbot Arena（LMArena）的做法。用户输入一个问题，两个匿名的模型各自回答，用户投票选更好的那个。大量这样的“对战”结果，用国际象棋的 **Elo 等级分**算出每个模型的分数。

Elo 的规则是：两个模型各有一个分数，分差决定预期的胜率：

$$P(\text{A 赢}) = 1 \,/\, (1 + 10^{(R_B - R_A)\,/\,400})$$

每比一场，双方按“实际结果 − 预期”调整分数：$R \leftarrow R + K \times (\text{实际} - \text{预期})$，$K$ 是一个常数（比如 32）。A 有 1200 分、B 有 1000 分时，A 赢的预期概率是 0.760。如果 A 赢了，只加 $32 \times 0.24 \approx 7.7$ 分；如果爆冷输了，A 掉到 1175.7，B 升到 1024.3。比赛足够多以后，分数会收敛到反映真实水平的位置。上面的演示可以自己点几场试试。

要点

评测大模型有多种尺子：困惑度适合比较同一模型的不同版本；基准测试用选择题、答案匹配、代码测试算正确率；开放式任务靠模型当裁判或人类对战排名（Elo）。每种都有陷阱：答案歧义、数据污染、裁判偏好。

用代码复现

下面的代码用 ToyLM 做了 5 道选择题（给每个选项算概率，取最大的），计算准确率；然后算一场 Elo 对战的预期胜率和分数变化。

```
import json
import numpy as np

ex = json.load(open("tinylm.json", encoding="utf-8"))["examples"]["22"]
V = ex["vocab"]
E, W1, b1, W2, b2 = (np.array(ex[k]) for k in ["E", "W1", "b1", "W2", "b2"])

def probs(a, b):
    z = np.maximum(0, np.concatenate([E[V.index(a)], E[V.index(b)]]) @ W1 + b1) @
W2 + b2
    p = np.exp(z - z.max()); return p / p.sum()

# 1. 选择题：给每个选项算模型的概率，选最大的，和标准答案比
items = [
    ("小猫 吃", ["鱼", "骨头", "苹果", "AI"], "鱼"),
    ("小狗 吃", ["鱼", "骨头", "香蕉", "推理"], "骨头"),
    ("你 学", ["AI", "推理", "鱼", "小狗"], "推理"),
    ("我 喜欢", ["小猫", "小狗", "苹果", "学"], "小狗"),
    ("我 吃", ["苹果", "香蕉", "骨头", "鱼"], "香蕉"),
]
right = 0
for prefix, opts, ans in items:
    p = probs(*prefix.split())
    scores = {o: float(p[V.index(o)]) for o in opts}
    pick = max(scores, key=scores.get)
    right += pick == ans
    print(f"“{prefix} __”: {', '.join(f'{o} {s:.3f}' for o, s in scores.items())} →
选 {pick}，答案 {ans}，{'对' if pick == ans else '错'}")
print(f"准确率 {right}/{len(items)} = {right / len(items):.0%}")

# 2. Elo（对战排名）：A 1200 分，B 1000 分；A 输了会怎样
def expect(ra, rb):
    return 1 / (1 + 10 ** ((rb - ra) / 400))
ra, rb, K = 1200, 1000, 32
ea = expect(ra, rb)
print(f"A 赢的预期概率 {ea:.3f}；A 输了：A {ra} → {ra + K * (0 - ea):.1f}，B {rb} →
{rb + K * (1 - (1 - ea)):.1f}")
```

输出：

```
“小猫 吃 __”: 鱼 1.000, 骨头 0.000, 苹果 0.000, AI 0.000 → 选 鱼, 答案 鱼, 对
“小狗 吃 __”: 鱼 0.000, 骨头 1.000, 香蕉 0.000, 推理 0.000 → 选 骨头, 答案 骨头, 对
“你 学 __”: AI 0.334, 推理 0.666, 鱼 0.000, 小狗 0.000 → 选 推理, 答案 推理, 对
“我 喜欢 __”: 小猫 0.332, 小狗 0.335, 苹果 0.000, 学 0.000 → 选 小狗, 答案 小狗, 对
“我 吃 __”: 苹果 0.501, 香蕉 0.499, 骨头 0.000, 鱼 0.000 → 选 苹果, 答案 香蕉, 错
准确率 4/5 = 80%
A 赢的预期概率 0.760; A 输了: A 1200 → 1175.7, B 1000 → 1024.3
```

前五行是每道题各选项的概率和对错，和演示一致；第六行准确率 80%；最后一行是 Elo 的例子。

到了大模型里

评测本身就是大规模推理。一次完整的评测要跑几十个基准、几万到几十万道题，有的还要多次采样，本质上是一个大批量的离线推理任务。推理引擎的吞吐（第 70 讲）直接决定评测要跑多久。

选择题只要 prefill。用“比较各选项的概率”来判分时，模型不需要生成任何 token，只需要把题目加选项送进去做一次前向（prefill），读出选项那几个位置的概率。同一道题的 4 个选项共享相同的题目前缀，正好可以用前缀缓存（第 76 讲）省掉重复计算。而需要生成完整解答的基准（数学、代码），就要做大量的 decode，推理模型的评测尤其昂贵。

每次优化都要回归评测。做推理优化时，速度变快了，还要确认结果没变差。量化到 4 位后困惑度涨了多少、基准分数掉了多少，换了一个新的注意力算子后输出是否和原来一致（数值误差在可接受范围内），这些都要靠评测来把关。第三章讲量化、算子时会反复用到。

容易混淆的地方

1. **基准分数不等于实际好用**。一个模型可能在某些基准上分数很高，用起来却不如人意，反过来也可能。基准只是能力的一个侧面，最终还要看在自己的任务上的表现。
2. **评测的细节会大幅影响分数**。提示的写法、几个示例（few-shot）、是否用思维链、答案的提取规则，都能让同一个模型的分数差好几个百分点。比较不同模型的分数时，要确认评测设置相同。
3. **Elo 分数是相对的**。它反映的是在同一个竞技场里、和同一批对手比较的相对强弱，不同排行榜的分数不能直接比较。

回顾

判断模型好不好，有好几把尺子：困惑度直接衡量预测下一个 token 的能力，适合比较同一模型的不同版本；基准测试用选择题（比较选项的概率）、答案匹配、代码测试（pass@k）算正确率；开放式任务靠模型当裁判或人类对战排名，Elo 用“实际 − 预期”调整分数。ToyLM 的 5 道选择题对了 4 道，但后两道的对错只差千分之几，因为答案本身有歧义。评测要警惕答案歧义、数据污染和裁判偏好；对推理系统来说，评测本身是大规模推理，也是每次优化之后必须做的回归检查。

到这里，第二章“大模型基础”结束了。我们从感知机出发，搭出了完整的 Transformer，认识了现代大模型的每一处改进，也走了一遍训练的全流程。第三章进入这门课的主体：**单机推理**。第一个问题是：模型在 GPU 上跑，快慢到底由什么决定？下一讲：**GPU 架构：SM 与 Tensor Core**。

AI INFRA · HANDBOOK

第 七 篇

从零实现一个 GPT 模型

收录 8 讲 · 第 52–59 讲

第52讲 g0 · 语料与分词器

21 分词与 BPE　23 预测下一个 token　24 自回归生成

第 2–5 篇把 GPT 拆成了零件：分词、嵌入、注意力、掩码、LayerNorm、残差、FFN、输出层，第 6 篇又讲了预训练和微调在做什么。每个零件我们都手算过、拖动过，但还没有把它们装在一起，真正训练出一个会写字的模型。

这一篇就做这件事：**从一个空目录开始，写出一个 GPT，在这门课自己的讲义上把它训练出来。**这个仓库叫 minigpt，一共 8 版，每讲一版，每一版都能跑。到最后一版，它能读懂“用一句话说明‘KV Cache’解决什么问题”这样的指令，并按格式作答。

这一讲是 g0：先准备语料，再写一个分词器，把 50 万字的讲义变成一串编号。

参考了哪本书

这一篇的路线参考了 Sebastian Raschka 的《Build a Large Language Model (From Scratch)》（中文译本见 skindhu.github.io/Build-A-Large-Language-Model-CN）：同样从数据加载器写起，经过注意力、GPT 模型、预训练，最后装进 GPT-2 的权重、做指令微调。不同的地方有三处：语料换成了这门课的中文讲义，分词器我们自己写、自己训练，代码按版本分目录，读者用 `diff -r g2 g3` 就能看出一讲改了什么。

要写出什么

先看最后的样子。下面这张图是 minigpt 的全部结构，左边是数据，中间是模型，右边是用模型的几种方式。上面的按钮可以切换版本，强调色是这一版新增或改动的部分；点任何一个模块，下面会说明它在哪个文件、哪一讲写。



图 52-1　minigpt 的结构。g0 只有数据这一列的前两块：语料和分词器。切到 g7 可以看到全貌。

8 个版本是这样安排的：

版本	这一讲写什么	主要文件	讲
g0	语料、BPE 分词器	`data/prepare.py`、`tokenizer.py`	52
g1	滑动窗口的数据加载器；token 嵌入和位置嵌入	`data.py`	53
g2	因果多头注意力	`attention.py`	54
g3	LayerNorm、GELU、FFN、残差，组装 GPT，贪心生成	`model.py`、`generate.py`	55
g4	预训练循环：交叉熵、AdamW、学习率预热和余弦退火	`train.py`	56
g5	温度、Top-k、Top-p；存档、续训、按验证集挑存档	`generate.py`、`checkpoint.py`	57
g6	把 GPT-2 的权重装进同一个模型，和 HuggingFace 逐个比对	`gpt2.py`	58
g7	指令微调：只在回答上算 loss	`sft.py`	59

和第五章的 miniinfer 一样，每个版本是一个完整目录，后一版从前一版复制再改。每一版都有一组检查（`tests/check.py`），写完先跑检查，确认对了再往下走。

模型的尺寸定得很小：词表 4096、隐藏维度 d = 256、4 个头、4 层，约 530 万个参数。一台 M2 芯片的 MacBook，用它的 GPU（PyTorch 叫 MPS）训练 3000 步大约 10 分钟；有 NVIDIA GPU 会更快，只用 CPU 也能跑，就是慢一些。

准备环境

只需要 PyTorch。第 58 讲要和 HuggingFace 的实现比对，再装一个 transformers：

```
cd minigpt
python3 -m venv .venv && .venv/bin/pip install torch transformers
.venv/bin/python -c "import torch; print(torch.__version__,
torch.backends.mps.is_available(), torch.cuda.is_available())"
```

最后一行打印 PyTorch 的版本，以及有没有 Apple 的 GPU、NVIDIA 的 GPU。本文的数字是在 PyTorch 2.15、M2 芯片上跑出来的，换一台机器，训练的 loss 会有小数点后两三位的差别，结论不变。

语料：用这门课自己的讲义

训练语言模型先得有文字。书里用的是一篇两万字的英文短篇小说，我们换成这门课自己的讲义，原因有三个：不用下载，是中文，而且你读过它，模型学会了什么、学歪了什么，一眼就看得出来。

讲义是 MDX 文件，里面除了正文，还有元数据、组件、代码块和公式。`data/prepare.py` 把这些都去掉，只留下正文段落：

minigpt/data/prepare.py 第 17–45 行 / 共 45 行 ⇆ 对照

```python
def clean(s: str) -> str:
    s = re.sub(r"^---\n.*?\n---\n", "", s, flags=re.S)          # 元数据
    s = re.sub(r"^import .*$", "", s, flags=re.M)               # import 行
    s = re.sub(r"```.*?```", "", s, flags=re.S)                 # 代码块
    s = re.sub(r"\$\$.*?\$\$", "", s, flags=re.S)               # 公式
    s = re.sub(r"<([A-Z]\w*)\b[^>]*?/>", "", s, flags=re.S)     # 自闭合组件 <X ... /
    s = re.sub(r"<([A-Z]\w*)\b.*?</\1>", "", s, flags=re.S)     # 成对组件 <X>...</X
    s = re.sub(r"<[^>]+>", "", s)                               # 剩下的 HTML 标签
    s = re.sub(r"^\|.*$", "", s, flags=re.M)                    # 表格
```

1 **去掉不是正文的东西** 第 17-24 行

按顺序用正则表达式删：开头 `---` 之间的元数据、`import` 行、```` ``` ```` 包起来的代码块、`$$` 包起来的公式、组件（大写字母开头的标签，比如 `<CodeWalk ... />`），最后是剩下的 HTML 标签。

顺序有讲究：组件的属性里常常有 `>` 和换行，必须先按“整个组件”删，再删零散的标签，否则会留下半截属性。

2 **去掉 Markdown 的格式** 第 25-31 行

3 **一讲一篇，中间放分隔符** 第 34-41 行

← 上一步 | 看整个文件 | 下一步 →

结果是 499,220 个字符，UTF-8 编码后约 1.1 MB。它和第 6 篇说的“几万亿个 token”差了七个数量级，所以我们不指望模型学到什么知识，只指望它学会这门课的句式和用词：知道“KV Cache”后面常跟“的大小”，“第 N 讲”的 N 是个数字，句子以“。”结尾。

为什么不直接用 GPT-2 的分词器

GPT-2 的分词器是现成的，词表 50,257 个 token。直接用它有两个问题。

第一，它是在英文网页上训练的，很多中文常用字没有被合并成一个 token，要用两三个字节 token 表示，序列变长，模型看到的上下文就短了。

第二，词表太大，对我们这个小模型不划算。嵌入表的参数是 $V \times d$，$V = 50{,}257$、$d = 256$ 时是 1287 万个，输出层再来一份，光这两张表就是 2500 多万个参数，是 4 层 Transformer 本身（约 316 万）的 8 倍。词表里绝大多数 token 在我们的语料里一次都不出现，这些参数白白占着。

所以我们自己训练一个分词器，词表定为 4096。

设计：从字符出发，生僻字退回字节

第 21 讲讲过 BPE：从最小的单位出发，一遍遍地把最常相邻出现的两个单位合并成一个新单位。“最小的单位”有两种选法。

GPT-2 从**字节**出发：任何文字都是一串 UTF-8 字节，初始词表就是 256 个字节，永远不会遇到不认识的字。代价是一个常用汉字占 3 个字节，要先花两次合并把它拼出来。

我们从**字符**出发：语料里出现过 3 次以上的字符，各占一个初始编号。这样一个汉字一开始就是一个 token，合并直接用在“模型”“请求”这样的词上。出现不到 3 次的生僻字，以及训练时根本没见过的字，**退回到字节**：拆成 3 个 UTF-8 字节，每个字节有自己的编号。这就是 Llama、Qwen 的分词器里常见的 byte fallback，它让“任何文字都能编码”这条保证依然成立。

于是词表的编号这样排：0–255 是 256 个字节，256 是分隔符 `◁endoftext▷`，257 起是字符（越常见编号越小），再往后是合并出来的 token，按学到的先后排。

还有一个细节：**合并只在一个“块”之内发生**。编码之前先把文字切成块：一串连续的英文字母、单个数字、一串连续的汉字、单个空白、单个其他符号。这样，“token”可以合成一个，但“token,”不会把逗号也粘进去；数字一位一位地切，模型看到的 2048 永远是 2、0、4、8 四个 token，不会因为“2048”恰好常见就单独成词。GPT-2 用的也是这种先切块、再合并的办法，只是它的规则针对英文。

tokenizer.py：训练

minigpt/g0/tokenizer.py 第 1–70 行 / 共 133 行 ⇆ 对照

```
1  """BPE 分词器：从字符出发合并，生僻字退回到字节（第 52 讲）。
2
3  词表的编号这样排：
4    0–255       256 个字节（生僻字、没见过的字拆成 UTF-8 字节，任何文字都能编码）
5    256         ◁endoftext▷（文章之间的分隔符）
6    257 起      语料里出现过至少 min_freq 次的字符
7    再往后      按学到的顺序排的合并结果
8  """
9  import json
```

1 **切块的规则** 第 13–15 行

`CHUNK` 这个正则表达式按顺序尝试五种模式：英文单词、单个数字、连续汉字（Unicode 里 4E00–9FFF 是常用汉字区）、单个空白、单个其他字符。`findall` 从左到右把整段文字切成块，每个字符恰好属于一块。

2 **词表里有什么** 第 18–30 行

3 **数块，收字符** 第 33–42 行

4 **数一遍所有相邻对** 第 44–50 行

5 **挑次数最多的一对，合并** 第 52–58 行

6 **只更新受影响的块** 第 59–69 行

← 上一步 | 看整个文件 | 下一步 →

`_merge` 和 `_base` 是两个小工具：

minigpt/g0/tokenizer.py 第 72–91 行 / 共 133 行 ⇆ 对照

```
72      @staticmethod
73      def _merge(ids, pair, new):
74          out, i = [], 0
75          while i < len(ids):
76              if i + 1 < len(ids) and (ids[i], ids[i + 1]) == pair:
77                  out.append(new)
78                  i += 2
79              else:
80                  out.append(ids[i])
```

1 **_merge：把一对替换成新编号** 第 72–82 行

从左往右扫，遇到这一对就换成新编号、跳过两个，否则原样保留。从左往右的顺序很重要：“a a a”合并（a, a）得到“aa a”，而不是“a aa”。训练和编码都用这同一个函数，两边的结果才会一致。

2 **_base：字符或字节** 第 84–91 行

← 上一步 | 看整个文件 | 下一步 →

tokenizer.py：编码和解码

minigpt/g0/tokenizer.py 第 94–133 行 / 共 133 行 ⇆ 对照

```
94      def _encode_chunk(self, chunk: str) -> list[int]:
95          if chunk in self.cache:
96              return self.cache[chunk]
97          ids = self._base(chunk)
98          while len(ids) >= 2:
99              # 在所有相邻对里，找最早学到的那条合并规则
100             p = min(zip(ids, ids[1:]), key=lambda q: self.merges.get(q, 1 << 30))
101             if p not in self.merges:
102                 break
```

1 **编码一块：按学到的顺序合并** 第 94–105 行

先变成初始编号，然后反复做一件事：在当前所有相邻对里，找**最早学到**的那条规则（编号最小），合并它；直到没有一对在规则里。

为什么不是“按规则顺序从头到尾试一遍”？两种做法结果相同，但规则有 2483 条，一个块通常只用得到几条；按“当前的相邻对里谁最早”来找，每轮只看这一块里的几对。

同一个块编码结果总是一样的，所以用 `cache` 记下来。语料里“的”出现几千次，只算一次。

2 **编码整段** 第 107–114 行

3 **解码：拼字节，再整体解码** 第 116–117 行

4 **存和读** 第 119–133 行

← 上一步 | 看整个文件 | 下一步 →

训练分词器

`scripts/train_tokenizer.py` 用语料的前 90% 训练（最后 10% 留作验证集，后面训练模型时也这样切），存成 `data/tokenizer.json`，再拿几个例子看看效果：

```
$ .venv/bin/python scripts/train_tokenizer.py
训练集 449,298 个字符，1,631 种不同的字符；切成 195,389 个块，其中不同的块 31,493 个
训练用时 22.3 s，词表 4096 个：256 个字节 + 1 个分隔符 + 1356 个字符 + 2483 次合并
最先学到的 12 次合并：  一个 en to tok token 模型 请求 每个 in de ll er
最长的 8 个 token：  FlashAttention / PagedAttention / Transformer / edAttention /
HuggingFace / InfiniBand / continuous / Attention

例句 41 个字符 → 23 个 token
KV |  | Cache |  | 把 | 每一层 | 的 |  | K | 、 | V |  | 存下来 | ， | 生成 | 下一个 |
| token |  | 时 | 不用 | 重算 | 。
[1628, 257, 1677, 257, 330, 1986, 259, 257, 362, 273, 354, 257, 3515, 258, 1669, 2232,
257, 1617, 257, 296, 2231, 1987, 260]
生僻字“饕餮”不在词表里，退回字节：[233, 165, 149, 233, 164, 174] → 解码回来：饕餮

验证集 49,922 个字符 → 33,048 个 token，平均每个 token 1.51 个字符
```

图 52-2　在讲义上训练出的分词器：1356 个字符加 2483 次合并，凑满 4096 个 token。最先学到的合并是“一个”、英文的 token，以及“模型”“请求”。

一行一行地看。

词表的构成。256 个字节 + 1 个分隔符 + 1356 个字符 + 2483 次合并 = 4096。训练集里有 1631 种不同的字符，其中出现 3 次以上的 1356 个，剩下的 275 个生僻字和符号走字节回退。

最先学到的合并是“一个”，然后是英文的 en、to、tok、token，然后是“模型”“请求”“每个”。“token”分了三步才合成：先合成“to”，再“tok”，最后“token”，每一步都是当时次数最多的一对。这门课讲推理，“token”“模型”“请求”出现得多，自然先被合并。换成小说当语料，最先学到的会是完全不同的一批。

最长的 token 是 FlashAttention、PagedAttention、Transformer、HuggingFace、InfiniBand 这些专有名词。英文单词在一个块里，可以一路合并到整个单词；汉字一串也是一个块，但中文的“词”很少像英文专有名词那样一字不差地重复几十次，所以中文 token 大多是两三个字。

例句是 41 个字符，切成了 23 个 token。“每一层”“存下来”“下一个”“不用”“重算”都是一个 token。注意空格：讲义的写作规范要求中英文之间空一格，所以“KV”“Cache”之间、“把”之前都有一个单独的空格 token（编号 257，是出现最多的字符）。GPT-2 的做法是把空格粘在后一个单词的开头（“ Cache”是一个 token），可以省下这些空格 token；我们的切块规则把空白单独成块，换来规则简单，代价是序列长了一些。

生僻字“饕餮”不在词表里，退回成 6 个字节编号（每个字 3 个），解码回来一字不差。

压缩率：验证集 49,922 个字符编码成 33,048 个 token，平均每个 token 1.51 个字符。这个数决定了模型一次能“看”多远：上下文 256 个 token，约等于 390 个字符，差不多是讲义里的两三段。

要点

分词器决定了模型看到的世界。词表越大，序列越短、一次看得越远，但嵌入表和输出层越大；从字符出发配合字节回退，中文的常用字一开始就是一个 token，任何字符又都能编码。

检查

每一版写完都跑 `tests/check.py`。g0 只检查两件事：验证集（分词器训练时没见过）编码再解码，和原文一字不差；没见过的字和表情符号也能还原。

```
$ .venv/bin/python tests/check.py g0

检查 g0:
  通过  g0 分词器：验证集编码再解码，和原文一字不差（49,922 个字符）
  通过  g0 分词器：没见过的字退回字节，也能还原
全部通过
```

⇆ 对照

图 52-3　g0 的检查：分词器是可逆的。

“可逆”看起来理所当然，其实很容易写错。比如解码时一个一个 token 分别解码，生僻字就会变成乱码；比如编码时忘了分隔符要单独处理，`◁endoftext▷` 就会被拆成普通字符。后面训练出的模型生成的每一个编号，都要靠这个分词器变回文字，所以先把它验证死。

容易写错的地方

- **编码时的合并顺序**。必须按“学到的先后”合并，而不是按“这一块里哪一对次数最多”。训练时的次数是对整个语料统计的，编码一个新句子时，句子里的次数没有意义。
- **训练集和验证集要在分词之前切开**。分词器如果见过验证集，它的合并规则就带上了验证集的信息，后面验证集上的 loss 会偏乐观。我们只用前 90% 训练分词器。
- **词表大小要和模型对上**。模型的嵌入表是 V × d，V 必须等于分词器的词表大小。换一个分词器，模型就得从头训练。

回顾

g0 做了两件事：`data/prepare.py` 把 121 讲的 MDX 清理成 499,220 个字符的纯文本，讲与讲之间用 `◁endoftext▷` 隔开；`tokenizer.py` 用 133 行实现了一个从字符出发、生僻字退回字节的 BPE 分词器，在语料上训练出 4096 个 token，平均每个 token 1.51 个字符，编码再解码一字不差。

现在语料是一串 30 万个编号。可模型一次只能看 256 个，而且要学的是“看到前面，猜下一个”。这一长串编号怎么切成一个个训练样本？编号又怎么变成模型能计算的向量？下一讲：**g1 · 数据加载器与嵌入**。

第53讲 g1 · 数据加载器与嵌入

22 词嵌入　23 预测下一个 token　25 位置编码

上一讲结束时，50 万字的讲义变成了一串编号：训练集 302,475 个，验证集 33,048 个。模型一次最多看 256 个 token，而它要学的是第 23 讲说的那件事：看到前面，猜下一个。

这一讲回答两个问题。第一，这一长串编号怎么切成一个个训练样本，每个样本的“题目”和“答案”是什么？第二，编号是整数，模型要做矩阵乘法，编号怎么变成向量？前一个问题由数据加载器（data loader）解决，写在 `data.py` 里；后一个是模型的第一层，嵌入（embedding）。

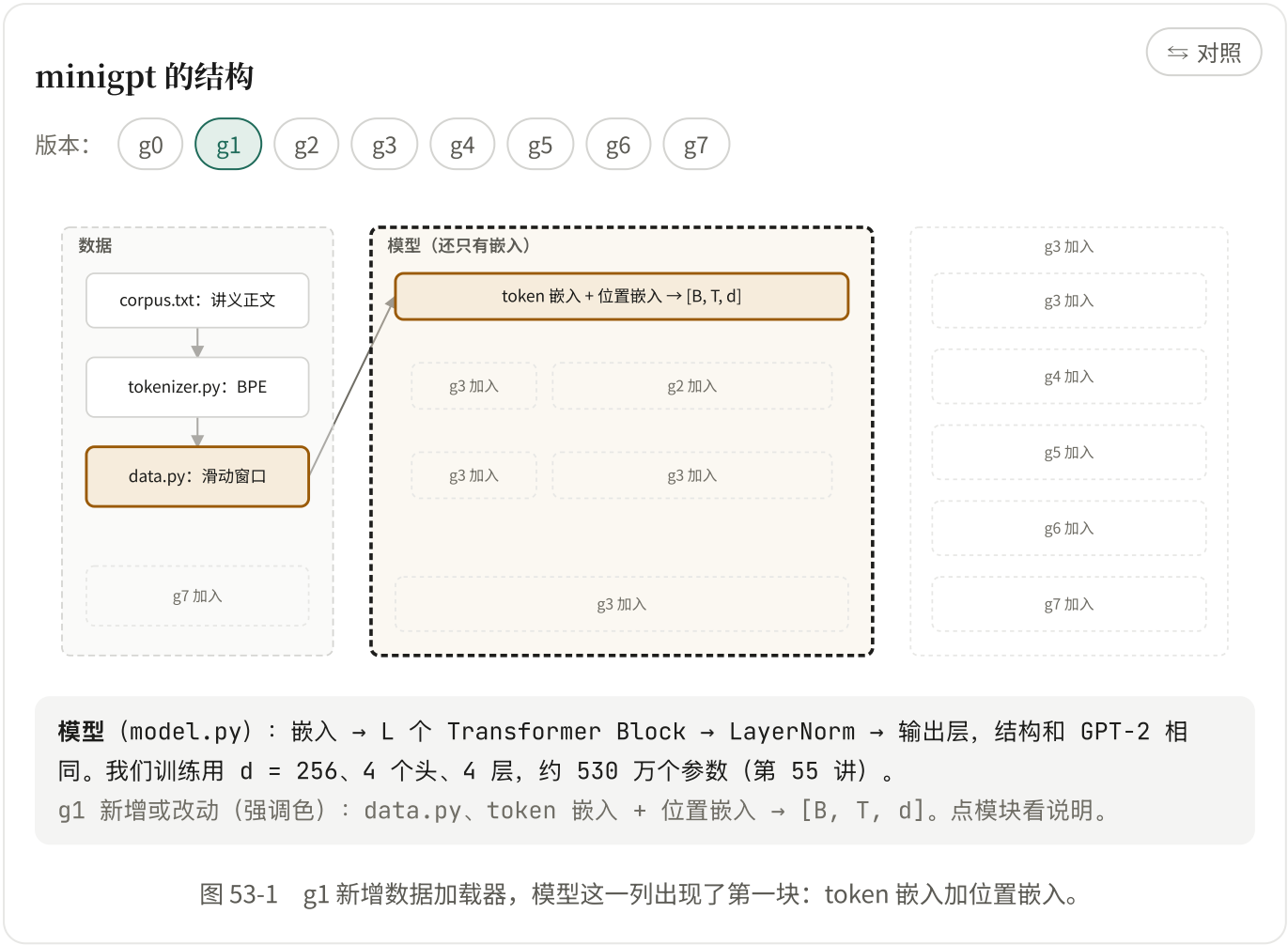


图 53-1　g1 新增数据加载器，模型这一列出现了第一块：token 嵌入加位置嵌入。

一个样本里有 T 道题

语言模型的训练数据不需要人工标注。一段文字本身就是题目和答案：“计算机 其实 只 会 做”，题目是“计算机”，答案是“其实”；题目是“计算机 其实”，答案是“只”；依此类推。

所以，从长序列里取出连续的 T + 1 个 token，前 T 个叫 x，后 T 个叫 y。y 就是 x 整体往后错一位：

- 位置 0：看到 x[0]，答案是 y[0] = x[1]
- 位置 1：看到 x[0..1]，答案是 y[1] = x[2]
- ……
- 位置 T − 1：看到 x[0..T−1]，答案是 y[T−1]

一个长 T 的样本里，有 T 道“预测下一个”的题。模型做一次前向，就同时给出这 T 道题的答案（第 34 讲的前向传播，每个位置都输出一行打分），因果掩码保证位置 t 只看得到 x[0..t]，不会偷看答案（第 27 讲）。这是语言模型训练效率高的根本原因：**一次前向，T 个训练信号**。

窗口怎么挪

样本是在长序列上开的一个个窗口。窗口每次往后挪多远，叫步长（stride）。下面的演示用训练集开头的 40 个真实 token：浅绿色是当前样本的 x，下面带琥珀色底线的是 y，虚线框是上一个样本的 x。

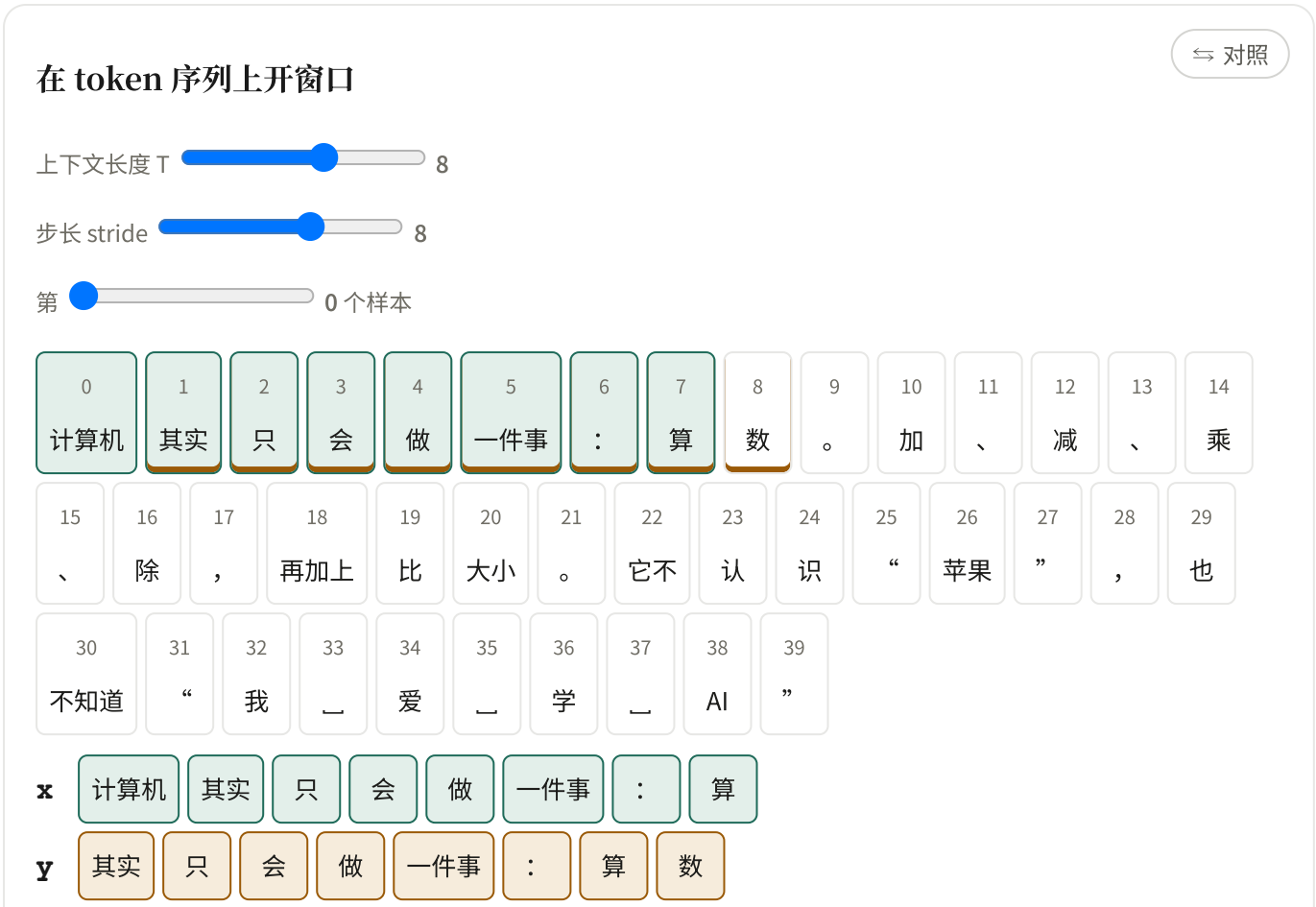


图 53-2　在训练集开头的 40 个 token 上开窗口。步长等于 T 时窗口首尾相接；步长小于 T 时相邻样本重叠；步长大于 T 时中间有 token 没用上。下方给出整个训练集能切出多少个样本。

试着调几组参数：

- stride = T：窗口首尾相接，每个 token 恰好出现在一个样本的 x 里。T = 256 时，训练集切出 1181 个样本。
- stride < T：相邻样本重叠。stride = 1 时样本数最多，但相邻两个样本几乎一模一样，模型一轮训练里把同一段话看了 T 遍，很容易背下来。
- stride > T：有一部分 token 永远不进任何样本，浪费数据。

我们训练时用 stride = T/2 = 128：样本数是不重叠时的两倍，同一段话在两个样本里出现，但位置不同（一次在窗口前半，一次在后半），模型能从同一段话里多学一点“前文更长时该怎么猜”。验证集用 stride = T，不重叠，每个 token 只算一次 loss。

data.py

minigpt/g1/data.py 第 1–43 行 / 共 43 行 ⇆ 对照

```
1  """把一长串 token 切成训练样本（第 53 讲）。
2
3  语言模型的一个样本是一对等长的序列：输入 x = 第 i 到 i+T-1 个 token，
4  目标 y = 第 i+1 到 i+T 个 token，也就是 x 整体往后错一位。模型在每个位置都预测“下一个”。
5  """
6  import os
7
8  import torch
9  from torch.utils.data import DataLoader, Dataset
```

1 只存起点 第 16–21 行

`GPTDataset` 把整串编号存成一个张量，再记下每个窗口的起点：`range(0, len - T, stride)`。不预先切出所有样本，是因为 stride = 1 时样本有 30 万个，每个 257 个数，全切出来要 7700 万个整数；只存起点，取样本时再切，内存只有原序列那么大。

起点的上限是 len − T（不含）：最后一个窗口要取 T + 1 个 token，起点 i 必须满足 i + T < len。

2 **取一个样本** 第 23–29 行

3 **先切开，再分词** 第 32–37 行

4 **DataLoader：打乱、组 batch** 第 40–43 行

← 上一步 | 看整个文件 | 下一步 →

编号怎么变成向量

模型的第一层是两张表。

token 嵌入是一张 V × d 的表，第 i 行是编号 i 的向量（第 22 讲）。编号 3949（“计算机”）进来，就取第 3949 行。它在训练开始时是随机数，训练中和其他参数一样被梯度更新，慢慢地，意思相近的 token 的向量会靠近。

位置嵌入是一张 T_max × d 的表，第 t 行是“位置 t”的向量。第 25 讲说过，注意力本身不知道词序，必须把位置信息加进去。第 25 讲讲的是用正弦、余弦公式算出来的位置编码；GPT-2 换了一种更简单的做法：位置向量也是参数，和 token 嵌入一样从随机数开始学。我们照 GPT-2 来。代价是模型只能处理不超过 T_max 个 token 的序列，第 T_max + 1 个位置在表里没有对应的行。

两者相加，就是送进第一层 Transformer Block 的输入：

$$h_{b,t} = E_{\text{tok}}[\,x_{b,t}\,] + E_{\text{pos}}[\,t\,]$$

b 是 batch 里的第几个样本，t 是位置。x 的形状是 [B, T]，查表后是 [B, T, d]；位置嵌入查出来是 [T, d]，对 batch 里每个样本都一样，PyTorch 的广播会把它加到每个样本上。

跑一遍

`scripts/dataloader_demo.py` 把上面这些都跑一遍：

```
$ .venv/bin/python scripts/dataloader_demo.py
分词用时 0.3 s：训练集 302,475 个 token，验证集 33,048 个 token

T = 8，第 0 个样本：
x: [3949, 2948, 320, 378, 396, 3316, 271, 283]   计算机 | 其实 | 只 | 会 | 做 | 一件事 | ：  | 算
y: [2948, 320, 378, 396, 3316, 271, 283, 277]   其实 | 只 | 会 | 做 | 一件事 | ：  | 算 | 数
拆开看每个位置要学的事：
  看到 计算机  →  预测 其实
  看到 计算机 | 其实  →  预测 只
  看到 计算机 | 其实 | 只  →  预测 会
  看到 计算机 | 其实 | 只 | 会  →  预测 做
stride = 8：37,809 个样本，前 3 个窗口起点 [0, 8, 16]
stride = 4：75,617 个样本，前 3 个窗口起点 [0, 4, 8]
stride = 1：302,467 个样本，前 3 个窗口起点 [0, 1, 2]

T = 256，B = 16：每轮 73 个 batch；一个 batch 的 x (16, 256)、y (16, 256)
token 嵌入 (16, 256, 256) + 位置嵌入 (256, 256) → (16, 256, 256)
两张表的参数：4096 × 256 + 256 × 256 = 1,114,112
同一个 token “的”出现在 (样本 0，位置 34) 和 (样本 0，位置 41)：
  查表结果相同：True；加上位置嵌入后相同：False
```

图 53-3　数据加载器切出的样本和嵌入之后的形状。同一个“的”在两个位置查出的 token 向量相同，加上位置向量后就不同了。

分词用时 0.3 秒：训练集加验证集近 50 万个字符，靠上一讲的缓存，每种块只合并一次。

T = 8 的第 0 个样本：x 是“计算机 | 其实 | 只 | 会 | 做 | 一件事 | ： | 算”，y 是它往后错一位，以“数”结尾。拆开看，前四道题是：看到“计算机”猜“其实”，看到“计算机 其实”猜“只”……这正是第 1 讲的第一句话：“计算机其实只会做一件事：算数。”

步长和样本数：stride = 8、4、1 时，样本数是 37,809、75,617、302,467，和 (302,475 − 8) ÷ stride 向上取整一致。

T = 256、B = 16：stride = T 时一轮 73 个 batch（1181 个样本 ÷ 16，丢掉最后不满的 13 个）。训练时 stride = 128，一轮 147 个 batch。一个 batch 的 x、y 都是 [16, 256]，一步训练就是 16 × 256 = 4096 道题。

嵌入：[16, 256] 的编号查表得到 [16, 256, 256]，位置嵌入 [256, 256] 广播相加，形状不变。两张表一共 1,114,112 个参数，其中 token 嵌入占了 94%。“的”在样本 0 的位置 34 和 41 各出现一次：查出的 token 向量完全相同，加上位置向量后就不同了。这就是位置嵌入的作用：同一个字，出现在不同位置，进入模型时已经带上了“我在哪”的信息。

要点

语言模型的样本是长序列上的一个窗口：x 是前 T 个 token，y 是往后错一位的 T 个，一个样本里有 T 道题。编号经过 token 嵌入和位置嵌入两次查表、相加，变成 [B, T, d] 的向量。

检查

```
$ .venv/bin/python tests/check.py g1

检查 g1:
  通过  g0 分词器：验证集编码再解码，和原文一字不差（49,922 个字符）
  通过  g0 分词器：没见过的字退回字节，也能还原
  通过  g1 数据：y 是 x 往后错一位（第 3 个样本从 12 开始）
全部通过
```

⇆ 对照

图 53-4　g1 的检查：在 0–99 这串数上，stride = 4 的第 3 个样本从 12 开始，y 恰好是 x 往后错一位。

容易写错的地方

- **x 和 y 错位的方向**。y 是 x 往后错一位（y[t] = x[t+1]），不是往前。写反了，模型学的是“看到后面猜前面”，加上因果掩码，它什么也看不到，loss 降不下去。
- **最后一个窗口越界**。窗口要 T + 1 个 token，起点上限是 len − T；写成 len − T + 1，最后一个样本的 y 会少一个 token，组 batch 时形状对不上。
- **验证集也打乱、也重叠**。不影响对错，但验证 loss 会因为每次抽到的样本不同而抖动，也会重复计算同一段话。

回顾

g1 加了 43 行的 `data.py`：`GPTDataset` 在长序列上按步长开窗口，取 T + 1 个 token，前 T 个是 x、后 T 个是 y；`load_corpus` 先按字符切出最后 10% 的验证集再分词；`make_loader` 用固定种子打乱、组 batch。训练集 302,475 个 token，T = 256、stride = 128 时一轮 147 个 batch。编号经过 token 嵌入（4096 × 256）和位置嵌入（256 × 256）相加，变成 [B, T, d]。

现在每个位置都有了一个 256 维的向量，但这些向量彼此独立：位置 5 的向量只知道自己是“的”、在第 5 个位置，不知道前面写了什么。要猜下一个 token，每个位置得能看到它前面的所有位置。下一讲：**g2 · 因果多头注意力**。

第54讲 g2 · 因果多头注意力

重点

26 自注意力　27 因果掩码　28 多头注意力

经过上一讲的两次查表，每个位置有了一个 256 维的向量。可这些向量互不相干：位置 34 的“的”不知道它前面是“KV Cache 的大小”还是“模型的参数”。要猜下一个 token，每个位置都得能看到它前面的所有位置，从中挑出有用的信息。这就是注意力做的事。

第 26–28 讲把注意力拆成了三步来讲：自注意力怎么打分、加权（第 26 讲），因果掩码怎么挡住未来（第 27 讲），为什么要多个头（第 28 讲）。这一讲把它们写成一个 PyTorch 模块，`attention.py`，39 行。重点不在公式，公式前面都推过了；重点在**形状**：怎么用几次矩阵乘法，一次算完 batch 里所有样本、所有头、所有位置的注意力。

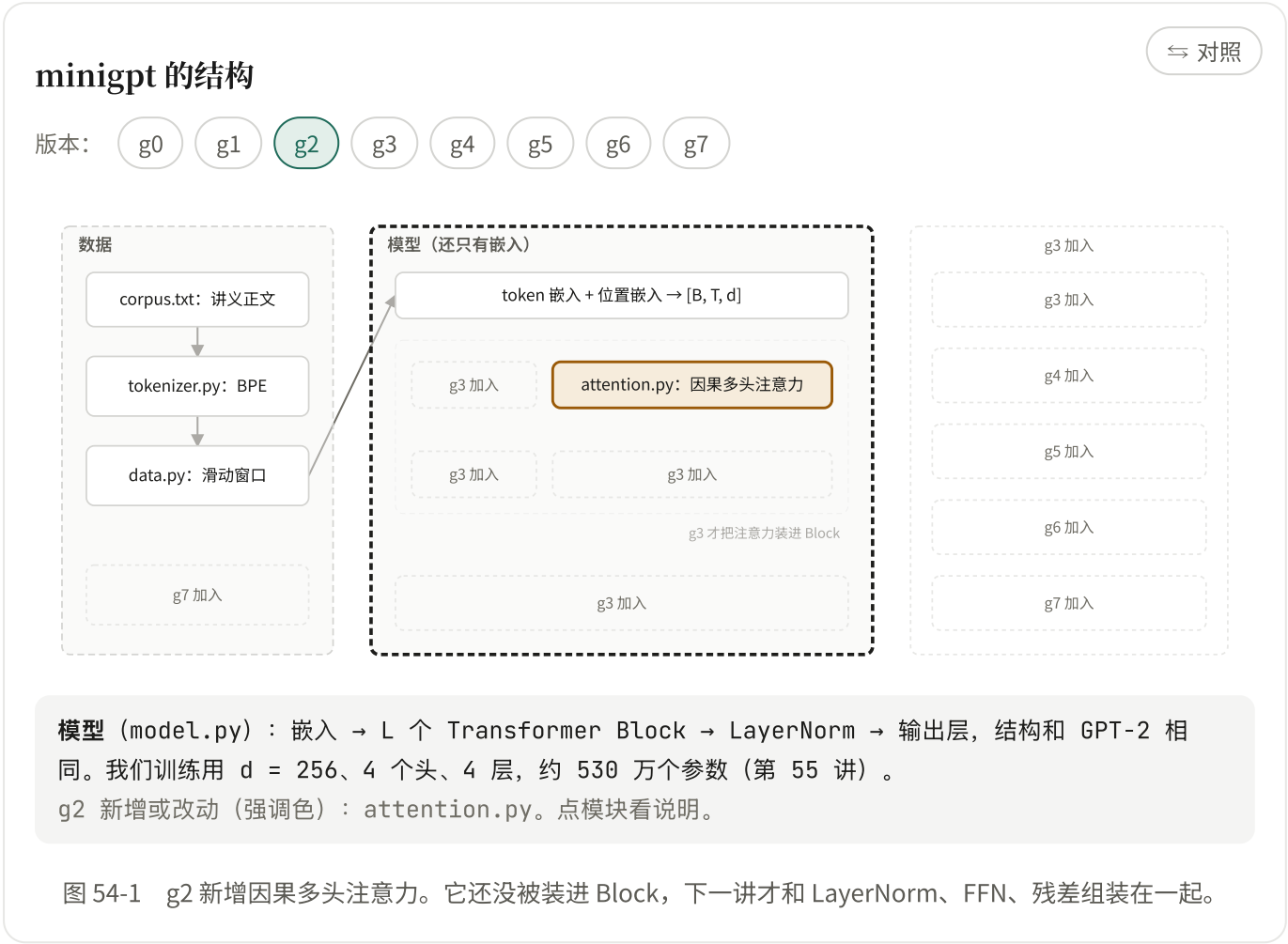


图 54-1　g2 新增因果多头注意力。它还没被装进 Block，下一讲才和 LayerNorm、FFN、残差组装在一起。

先把公式和形状对上

回忆一下单个头的计算（第 26、27 讲）。输入 X 是 T × d，三个投影矩阵把它变成查询、键、值：

$$Q = XW_q, \quad K = XW_k, \quad V = XW_v$$

打分是 $QK^T / \sqrt{h_d}$，T × T，第 t 行第 j 列是位置 t 对位置 j 的打分。j > t 的位置是未来，打成 −∞；每行做 softmax 得到权重，再乘 V，得到每个位置的输出。

多头（第 28 讲）就是 H 组独立的 W_q、W_k、W_v，每组的输出维度是 h_d = d / H。我们的模型 d = 256、H = 4，每个头 64 维。H 个头的输出拼起来，又是 d 维，再乘一个 d × d 的矩阵 W_o（代码里叫 `out_proj`）把各头的信息混合起来。

直接照这个描述写，会是一个 `for` 循环：对每个头，算一遍 Q、K、V、打分、softmax、加权。能跑，但 GPU 最怕的就是一堆小矩阵乘法（第 62 讲的 Roofline：矩阵越小，算术强度越低）。实际的写法是把循环“压平”到张量的维度里。

第一步：所有头的投影合成一个矩阵。H 个头的 W_q 各是 d × h_d，并排放在一起正好是 d × d。所以 `W_q` 就是一个 d × d 的线性层，输出的 d 维里，前 64 维是头 0 的 q，接着 64 维是头 1 的 q，依此类推。一次矩阵乘法算出所有头的 q。

第二步：把 d 维拆开，头这一维提到前面。输出是 [B, T, d]，`view` 成 [B, T, H, h_d]，再交换第 1、2 维变成 [B, H, T, h_d]。现在前两维 B、H 可以看成“一共 B·H 个独立的小问题”，每个小问题是一个 T × h_d 的矩阵。

第三步：批量矩阵乘法。PyTorch 的 `@` 遇到超过两维的张量，会把最后两维当成矩阵、前面的维度当成批：[B, H, T, h_d] @ [B, H, h_d, T] = [B, H, T, T]。一条语句，算完 B·H 个打分矩阵。

第四步：拼回去。加权求和得到 [B, H, T, h_d]，交换回 [B, T, H, h_d]，再 `view` 成 [B, T, d]，就是 H 个头的输出拼在一起。

attention.py

minigpt/g2/attention.py 第 12–39 行 / 共 39 行 ⇆ 对照

```
12 class MultiHeadAttention(nn.Module):
13     def __init__(self, d: int, H: int, T_max: int, dropout: float, qkv_bias: bool =
14         super().__init__()
15         assert d % H == 0, "d 必须能被头数整除"
16         self.H, self.hd = H, d // H
17         self.W_q = nn.Linear(d, d, bias=qkv_bias)   # 所有头的 W_q 并排放在一个 d × d 矩
18         self.W_k = nn.Linear(d, d, bias=qkv_bias)
19         self.W_v = nn.Linear(d, d, bias=qkv_bias)
20         self.out_proj = nn.Linear(d, d)             # 把各头的结果混合回 d 维
21         self.dropout = nn.Dropout(dropout)
```

1 **头数要整除 d** 第 13–16 行

每个头 h_d = d / H 维。d = 256、H = 4 时 h_d = 64。如果 d 不能被 H 整除，拆头的 `view` 就会失败，所以先用 `assert` 挡住。

2 **四个 d × d 的线性层** 第 17–20 行

3 **掩码是 buffer，不是参数** 第 21–23 行

4 **投影、拆头、转置** 第 25–30 行

5 **打分、掩码、softmax、dropout** 第 32–35 行

6 **加权求和，拼回 d 维** 第 37–39 行

← 上一步 | 看整个文件 | 下一步 →

一步一步看

`scripts/attention_steps.py` 用 4 个 token、d = 4、2 个头的小例子把中间结果全印出来，再和 PyTorch 自带的实现比一比：

```
$ .venv/bin/python scripts/attention_steps.py                    ⇆ 对照

x (1, 4, 4) → q、k、v 拆头后 (1, 2, 4, 2)   (B, H, T, hd)
头 0 的打分 q·kᵀ/√hd:
 tensor([[-0.06,  0.13,  0.30, -0.23],
        [ 0.69,  0.56,  0.06, -0.55],
        [-0.24, -0.09,  0.16,  0.02],
        [-0.03, -0.09, -0.12,  0.13]])
盖上因果掩码:
 tensor([[-0.06,  -inf,  -inf,  -inf],
        [ 0.69,  0.56,  -inf,  -inf],
        [-0.24, -0.09,  0.16,  -inf],
        [-0.03, -0.09, -0.12,  0.13]])
softmax 之后（每行和为 1，右上角全是 0）：
 tensor([[1.00, 0.00, 0.00, 0.00],
        [0.53, 0.47, 0.00, 0.00],
        [0.27, 0.32, 0.41, 0.00],
        [0.25, 0.23, 0.23, 0.29]])
每行的和:  tensor([1.00, 1.00, 1.00, 1.00])

和 F.scaled_dot_product_attention 的最大差: 2.98e-08
改第 3 个 token 后各位置输出的变化:  ['0.00e+00', '0.00e+00', '0.00e+00', '3.18e-01']

d = 256、H = 4 的注意力层: 参数 262,400（4 个 256 × 256 矩阵 + out_proj 的 256 个偏置）
掩码 buffer (256, 256)，不算参数: 1 个 buffer
输入 (16, 256, 256) → 输出 (16, 256, 256)；注意力权重 [16, 4, 256, 256] 有 4,194,304 个数
```

图 54-2　因果多头注意力的中间结果：打分、盖掩码、softmax。和 F.scaled_dot_product_attention 的差在 10^{-8} 量级；改第 3 个 token，只有位置 3 的输出变。

拆头之后的形状是 (1, 2, 4, 2)：1 个样本、2 个头、4 个位置、每头 2 维。

头 0 的打分是一个 4 × 4 矩阵。盖上掩码后，右上角全是 −∞；softmax 之后右上角全是 0，每行的和都是 1。第 0 行只有一个非零元素，是 1：位置 0 只能看自己。第 1 行是 0.53、0.47：位置 1 把一半多的注意力给了位置 0。这些权重是随机初始化的模型算出来的，没有意义；训练之后，它们才会变成“这个位置该看哪里”。

和 PyTorch 对比：`F.scaled_dot_product_attention(q, k, v, is_causal=True)` 是 PyTorch 自带的注意力，在 GPU 上会自动选 FlashAttention 这类融合实现（第 84 讲）。用同一组 q、k、v，两者的最大差是 3×10^{-8} ，只是浮点数加法顺序不同带来的舍入误差。我们的实现是对的。

因果性：把第 3 个 token 的向量改掉，位置 0、1、2 的输出一点没变（差为 0），只有位置 3 变了。这说明掩码起作用了：一个位置的输出只依赖它自己和它前面的位置。这条性质后面还会用到两次：训练时它保证“不偷看答案”，推理时它让 KV Cache 成为可能（第 69 讲：前面位置的 K、V 不会因为后面来了新 token 而改变）。

真实尺寸：d = 256、H = 4 的注意力层有 262,400 个参数，就是 $4 \times 256^2 = 262{,}144$，加上 `out_proj` 的 256 个偏置。注意参数量和头数无关：H = 1 还是 H = 8，都是四个 d × d 矩阵，变的只是怎么切。掩码是 buffer，不算参数。

最后一行值得多看一眼：B = 16、T = 256 时，注意力权重是 [16, 4, 256, 256]，419 万个数，FP32 下 16 MiB。这还只是一层；每层的打分、softmax 的结果、dropout 的结果都要为反向传播留着，4 层就是上百 MiB。它和 T^2 成正比，T 翻倍就是 4 倍。这就是为什么 FlashAttention 要想办法不把这个 T × T 矩阵写进显存（第 84 讲）。

要点

多头注意力不需要循环：所有头的投影合成一个 d × d 矩阵，结果 view 成 [B, H, T, h_d]，批量矩阵乘法一次算完所有样本、所有头的打分。参数量 $4d^2$ 与头数无关；注意力权重的大小与 T^2 成正比。

检查

```
$ .venv/bin/python tests/check.py g2

检查 g2:
  通过  g0 分词器：验证集编码再解码，和原文一字不差（49,922 个字符）
  通过  g0 分词器：没见过的字退回字节，也能还原
  通过  g1 数据：y 是 x 往后错一位（第 3 个样本从 12 开始）
  通过  g2 注意力：和 F.scaled_dot_product_attention 一致（最大差 8.9e-08）
全部通过
```

⇆ 对照

图 54-3　g2 的检查：d = 32、4 个头、16 个位置的随机输入，和 PyTorch 的实现一致。

容易写错的地方

- **拆头时直接 view 成 [B, H, T, h_d]**。形状对得上，不报错，但结果是错的：内存里是 [B, T, d] 的顺序，必须先 view 成 [B, T, H, h_d] 再转置。直接 view 会把不同位置的数混进同一个头。
- **转置后忘了 contiguous**。会报错，比较好查。用 `reshape` 代替 `view` 也可以，它在需要时会自动复制。
- **掩码方向反了**。`triu(diagonal=1)` 屏蔽的是 $j > t$。写成 `tril`，模型就只能看未来、看不到过去；训练 loss 会降得出奇地快（它在偷看答案），生成时却一塌糊涂。上面“改第 3 个 token”的检查能抓住这种错。
- **除以 $\sqrt{d}$ 而不是 $\sqrt{h_d}$**。多头时每个头的点积是 h_d 维的，缩放要用 $\sqrt{h_d}$。

回顾

g2 用 39 行写出因果多头注意力：四个 $d \times d$ 线性层；q、k、v 拆成 [B, H, T, h_d]，批量矩阵乘法打分，除以 $\sqrt{h_d}$，用上三角掩码填 $-\infty$，softmax，dropout，乘 v，拼回 [B, T, d]，过 `out_proj`。它和 PyTorch 自带的实现差 3×10^{-8} ，改后面的 token 不影响前面的输出。

注意力让每个位置看到了上下文，但它只是在已有的向量之间做加权平均，自己几乎没有“加工”能力；层数一多，数值还会越来越不稳定。下一讲把它和 LayerNorm、FFN、残差连接组装成 Transformer Block，再叠成一个完整的 GPT：**g3 · 组装 GPT**。

第55讲 g3 · 组装 GPT

29 残差连接　30 LayerNorm　31 前馈网络　32 Transformer Block　35 参数量与计算量

零件已经有了三样：分词器、嵌入、注意力。这一讲把剩下的零件补齐，组装成一个完整的 GPT：输入一串编号，输出每个位置对下一个 token 的打分。然后写一个最简单的生成循环，让它说出第一句话。

它说出来的会是一串乱码，因为还没有训练。但这一讲结束时，模型的结构就定型了：后面四讲的预训练、解码、装 GPT-2 权重、指令微调，用的都是这一讲的 `GPT` 类，一行都不再改。

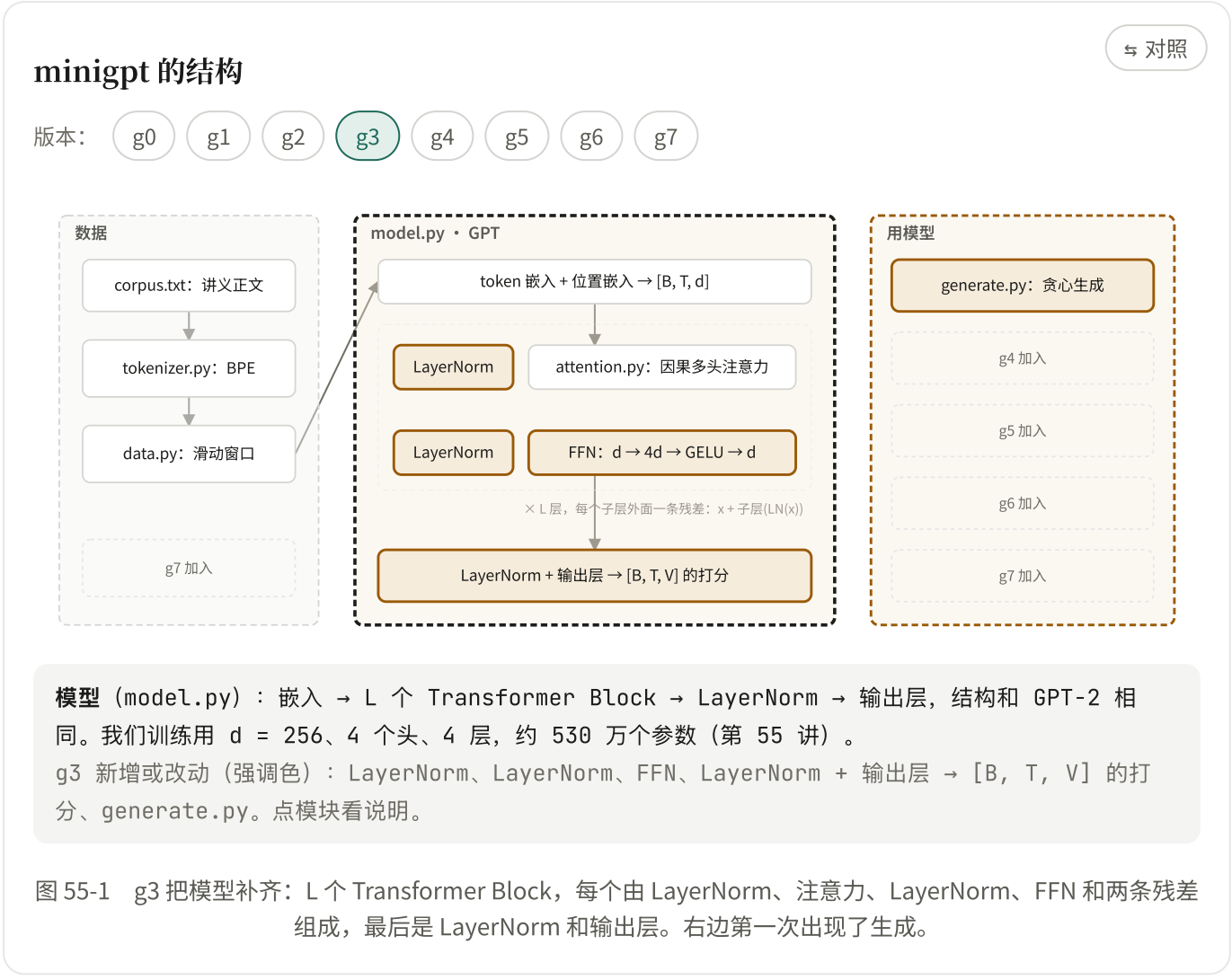


图 55-1　g3 把模型补齐：L 个 Transformer Block，每个由 LayerNorm、注意力、LayerNorm、FFN 和两条残差组成，最后是 LayerNorm 和输出层。右边第一次出现了生成。

还差哪几个零件

第 32 讲画过一个 Transformer Block 的结构。对照它，还差四样：

- **LayerNorm**（第 30 讲）：把每个 token 的 d 个数减去均值、除以标准差，再乘一组可学的缩放 γ、加一组平移 β。它让每一层的输入保持在相近的尺度上。
- **GELU**（第 14 讲）：GPT-2 用的激活函数。和 ReLU 的区别是负半轴不是一刀切成 0，而是平滑地趋近于 0。
- **前馈网络 FFN**（第 31 讲）：两个线性层夹一个 GELU，先从 d 维升到 4d 维，再降回 d。注意力负责“在位置之间搬运信息”，FFN 负责“在每个位置上加工信息”。
- **残差连接**（第 29 讲）：子层的输出加回到它的输入上，x + f(x)。网络深了，梯度可以沿着这条“直通”的路一路传回去。

组装的方式是 GPT-2 用的 **Pre-Norm**（第 36 讲）：先归一化，再进子层，再加残差。一个 Block 两个子层：

$$x \leftarrow x + \mathrm{Attn}(\mathrm{LN}_1(x)), \qquad x \leftarrow x + \mathrm{FFN}(\mathrm{LN}_2(x))$$

model.py：四个小零件

minigpt/g3/model.py 第 13–64 行 / 共 85 行 ⇆ 对照

```python
@dataclass
class GPTConfig:
    V: int = 4096          # 词表大小
    T_max: int = 256       # 最长上下文
    d: int = 256           # 隐藏维度
    H: int = 4             # 注意力头数
    L: int = 4             # 层数
    dropout: float = 0.1
    qkv_bias: bool = False
```

1 **所有尺寸放在一个配置里** 第 13–21 行

七个数决定了模型的全部形状：词表 V、最长上下文 T_max、隐藏维度 d、头数 H、层数 L、dropout 比例、q/k/v 要不要偏置。我们自己训练用默认值，约 530 万个参数；第 58 讲换成 GPT-2 的尺寸，代码不用改。

2 **LayerNorm** 第 24–34 行

3 **GELU 的 tanh 近似** 第 37–40 行

4 **FFN：升到 4d，再降回来** 第 43–49 行

5 **Transformer Block** 第 52–64 行

← 上一步 | 看整个文件 | 下一步 →

model.py：GPT

minigpt/g3/model.py 第 67–85 行 / 共 85 行 ⇆ 对照

```
67 class GPT(nn.Module):
68     def __init__(self, c: GPTConfig):
69         super().__init__()
70         self.c = c
71         self.tok_emb = nn.Embedding(c.V, c.d)
72         self.pos_emb = nn.Embedding(c.T_max, c.d)
73         self.drop = nn.Dropout(c.dropout)
74         self.blocks = nn.Sequential(*[TransformerBlock(c) for _ in range(c.L)])
75         self.final_norm = LayerNorm(c.d)
```

1 **六个部件** 第 67–76 行

token 嵌入、位置嵌入、嵌入后的 dropout、L 个 Block（`nn.Sequential` 按顺序串起来）、最后的 LayerNorm、输出层。

输出层是一个 d × V 的线性层，不带偏置。GPT-2 让它和 token 嵌入**共用同一份权重**（weight tying）：嵌入表的第 i 行，同时也是输出层给 token i 打分用的向量。我们这里没有共用，两张表各一份；为什么、差多少，下面看参数量时再说。

2 **前向：编号进，打分出** 第 78–85 行

← 上一步 | 看整个文件 | 下一步 →

generate.py：贪心生成

第 24 讲讲过自回归生成：把序列送进模型，取最后一个位置的打分，选一个 token 接在后面，再送进去，重复。

minigpt/g3/generate.py 共 13 行 ⇆ 对照

```
1  """自回归生成：每次把整段序列送进模型，取最后一个位置的 logits，选出下一个 token 接在后面（第 5
2  import torch
3
4
5  @torch.no_grad()
6  def generate_greedy(model, idx, max_new: int):
7      """idx: [B, T] 的编号。贪心：每步取打分最高的 token。"""
8      for _ in range(max_new):
9          ctx = idx[:, -model.c.T_max:]            # 超过上下文上限时，只保留最后 T_max 个
10         logits = model(ctx)[:, -1, :]            # [B, V]：只要最后一个位置
```

1 **不需要梯度** 第 5-7 行

`@torch.no_grad()`：生成时不训练，不需要为反向传播记录中间结果，省内存也更快。

2 **每步都把整段送进去** 第 8-13 行

← 上一步 | 看整个文件 | 下一步 →

跑一遍

```
$ .venv/bin/python scripts/model_summary.py

LayerNorm: [2.0, 4.0, 6.0, 8.0] → [-1.342, -0.447, 0.447, 1.342], 均值 0.0, 方差 1.000
GELU:  -3.0→-0.004 -1.0→-0.159 -0.5→-0.154 +0.0→+0.000 +0.5→+0.346 +1.0→+0.841
+3.0→+2.996

配置: V = 4096, T_max = 256, d = 256, H = 4, L = 4
  token 嵌入 tok_emb  1,048,576   4096 × 256
  位置嵌入 pos_emb       65,536   256 × 256
  每层注意力            262,400   4d² + d
  每层 FFN              525,568   8d² + 5d
  每层两个 LayerNorm      1,024   4d
  4 层合计            3,155,968   4 × 788,992
  最后的 LayerNorm          512   2d
  输出层 out_head     1,048,576   256 × 4096
  合计                5,319,168   ≈ 5.3 M, FP32 占 20.3 MiB

“注意力机制” → 2 个 token [1652, 3087] → logits (1, 2, 4096)
贪心生成 20 个 token: 注意力机制底元就是这样最低]推理系统得很原本领亏方法昂选择含r的精度缺由一

GPT-2 small 的尺寸 (V = 50257, T_max = 1024, d = 768, H = 12, L = 12) : 163,037,184 个参数
  输出层和 token 嵌入共用一份权重时: 124,439,808 个, 这就是常说的“124M”
```

图 55-2　组装好的 GPT：530 万个参数，FP32 约 20 MiB。没训练时生成的是词表里的 token 随机拼起来的乱码。同样的代码换成 GPT-2 small 的尺寸是 1.63 亿个参数，共用输出层后是 1.24 亿。

LayerNorm：[2, 4, 6, 8] 的均值是 5，方差是 5，归一化后是 [−1.342, −0.447, 0.447, 1.342]，均值 0、方差 1。

GELU：正半轴接近 x 本身（3 → 2.996），负半轴不是 0，而是一个小的负数：−1 → −0.159，−0.5 → −0.154，到 −3 已经几乎是 0。

参数量逐项对一下（第 35 讲的方法）：

- token 嵌入 4096 × 256 = 1,048,576，位置嵌入 256 × 256 = 65,536。
- 每层注意力 $4d^2 + d$ = 262,400，FFN $8d^2 + 5d$ = 525,568，两个 LayerNorm 各 2d，共 1024。一层 788,992，4 层 3,155,968。
- 最后的 LayerNorm 512，输出层 256 × 4096 = 1,048,576。
- 合计 5,319,168，FP32 每个 4 字节，约 20 MiB。

一个反直觉的地方：**4 层 Transformer 只占 59%，两张和词表有关的表占了 39%**。模型越小，词表的份额越大。这也是上一讲把词表定为 4096 的原因：如果用 GPT-2 的 50,257，两张表就是 2570 万个参数，是 Transformer 部分的 8 倍。反过来，GPT-2 small 的 Transformer 部分有 8500 万个参数，词表那张表 3860 万，共用输出层能省下其中一份，这就是 1.63 亿变成“124M”的原因。我们的模型共用的话能省 105 万，约 20%；书里也没有共用，理由是分开训练效果略好。这里按书的做法，第 58 讲装 GPT-2 权重时，把同一份权重复制进输出层即可。

形状：“注意力机制”是 2 个 token（“注意力”“机制”各一个），输出 logits 是 [1, 2, 4096]。

生成：贪心生成 20 个 token，得到的是“底元就是这样最低]推理系统得很原本领亏……”。每个 token 本身都是语料里的词（分词器是训练过的），但模型的权重是随机的，它给 4096 个 token 的打分几乎没有规律，选出来的就是随机拼凑。

要点

GPT = 嵌入 + L 个 Block + LayerNorm + 输出层。Block 是 Pre-Norm 的两个子层：x + Attn(LN(x))，x + FFN(LN(x))。整个模型只有一个接口：model(idx) 把 [B, T] 的编号变成 [B, T, V] 的打分。

检查

```
$ .venv/bin/python tests/check.py g3                    ⇆ 对照

检查 g3:
  通过  g0 分词器：验证集编码再解码，和原文一字不差（49,922 个字符）
  通过  g0 分词器：没见过的字退回字节，也能还原
  通过  g1 数据：y 是 x 往后错一位（第 3 个样本从 12 开始）
  通过  g2 注意力：和 F.scaled_dot_product_attention 一致（最大差 8.9e-08）
  通过  g3 模型：输出形状 [B, T, V]
  通过  g3 模型：改第 10 个 token，前 10 个位置的输出不变（因果）
全部通过
```

图 55-3　g3 的检查：整个模型也是因果的，改第 10 个 token，前 10 个位置的输出一点不变。

上一讲检查了单个注意力层的因果性，这里检查整个模型：LayerNorm、FFN 都是逐位置的，残差也是逐位置的，所以只要注意力是因果的，整个模型就是因果的。这个检查能抓住一类隐蔽的错误：比如 LayerNorm 写成了在 T 维上求均值（`dim=1`），各个位置就混在一起了，形状照

样对，只有这类检查能发现。

和上一版的差别

g3 在 g2 的基础上新增了 `model.py`、`generate.py` 两个文件，其余文件没动：

minigpt/g3/generate.py 新文件：+12 行 / −0 行 ⇆ 对照

```
+ """自回归生成：每次把整段序列送进模型，取最后一个位置的 logits，选出下一个 token 接在后面（第
+ import torch
+
+
+ @torch.no_grad()
+ def generate_greedy(model, idx, max_new: int):
+     """idx: [B, T] 的编号。贪心：每步取打分最高的 token。"""
+     for _ in range(max_new):
+         ctx = idx[:, -model.c.T_max:]            # 超过上下文上限时，只保留最后 T_max 个
+         logits = model(ctx)[:, -1, :]            # [B, V]：只要最后一个位置
+         nxt = logits.argmax(dim=-1, keepdim=True)
+         idx = torch.cat([idx, nxt], dim=1)
+     return idx
```

图 55-4　g3 新增的 generate.py。model.py 也是新文件，上面已经逐段讲过。

容易写错的地方

- **Post-Norm 和 Pre-Norm 混了。**原始 Transformer 是 LN(x + f(x))，GPT-2 是 x + f(LN(x))。两种都能跑，但装 GPT-2 权重时结构必须一致，否则输出完全不同。
- **位置嵌入用了 `torch.arange(T)` 却忘了 `device`。**模型在 GPU 上、位置编号在 CPU 上，查表时报错。
- **生成时没有截断到 T_max。**序列一长，位置嵌入查表越界，报 index out of range。

回顾

g3 补齐了 LayerNorm（γ、β，方差除以 d）、GELU（tanh 近似）、FFN（d → 4d → d）和 Pre-Norm 的 Transformer Block，组装成 `GPT`：嵌入 → 4 个 Block → LayerNorm → 输出层，5,319,168 个参数，其中 39% 在两张词表大小的表里。`generate_greedy` 每步把整段序列送进模型，取最后一个位置打分最高的 token。没训练过的模型，生成的是随机拼凑的词。

现在模型能给出打分了，但打分是随机的。怎么让它的打分变得有意义：给正确的下一个 token 打高分？要定义一个衡量“猜得多准”的数，再沿着让这个数变好的方向，一点点调整 530 万个参数。下一讲：**g4 · 预训练**。

第56讲 g4 · 预训练

重点

16 损失函数　19 优化器　20 训练循环　45 预训练与数据　46 Scaling Laws

上一讲组装好的 GPT 会说话了，说的是“底元就是这样最低]推理系统得很原本领亏”。530 万个参数都是随机数，它给 4096 个 token 的打分没有任何规律。

这一讲让它学。要做的事第 2 篇都讲过：定义一个衡量“猜得多准”的数，叫 loss（第 16 讲）；用反向传播算出每个参数该往哪个方向改（第 18 讲）；用优化器沿着这个方向走一小步（第 19 讲）；一个 batch 接一个 batch 地重复（第 20 讲）。区别只在规模：这次是 530 万个参数、30 万个 token，在 GPU 上跑 3000 步。

跑完你会看到一个第 6 篇只用文字讲过的现象：训练集上的 loss 一路降到 1.0，验证集上的 loss 却在第 900 步之后掉头往上走。



模型（model.py）：嵌入 → L 个 Transformer Block → LayerNorm → 输出层，结构和 GPT-2 相同。我们训练用 d = 256、4 个头、4 层，约 530 万个参数（第 55 讲）。
g4 新增或改动（强调色）：train.py。点模块看说明。

图 56-1　g4 新增 train.py：取 batch、算 loss、反向传播、更新参数，每 100 步在训练集和验证集上各估一次 loss。

loss：正确答案拿到了多少概率

模型在每个位置输出 4096 个打分，softmax 之后是一个概率分布。这个位置的正确答案是 y[t]，模型给它的概率是 p。交叉熵 loss 就是 −ln p（第 16 讲）：

- 模型把握十足，p = 0.9，loss = 0.105；
- 模型完全没把握，在 4096 个 token 里平均地猜，p = 1/4096，loss = ln 4096 = 8.318；
- 模型猜错了方向，p = 0.0001，loss = 9.21。

一个 batch 是 16 × 256 = 4096 个位置，每个位置一个 loss，取平均，就是这一步的 loss。它的指数 e^loss 叫困惑度（perplexity）：loss = 8.318 时困惑度是 4096，意思是“模型的犹豫程度相当于在 4096 个选项里瞎猜”；loss 降到 3，困惑度是 20，相当于在 20 个选项里挑。

所以，**没训练过的模型，loss 应该在 8.3 附近**。如果一开始就远大于它，说明初始化有问题，某些 token 的打分被放得特别大；如果远小于它，多半是数据泄漏了答案（比如掩码写反，模型看得到下一个 token）。这是训练开始时第一个要看的数。

train.py

minigpt/g4/train.py 第 20–51 行 / 共 117 行 ⇆ 对照

```
20 def pick_device():
21     if torch.cuda.is_available():
22         return "cuda"
23     if torch.backends.mps.is_available():       # Apple 芯片的 GPU
24         return "mps"
25     return "cpu"
26
27
28 def loss_batch(model, x, y):
```

1 **选设备** 第 20–25 行

有 NVIDIA GPU 用 CUDA，Apple 芯片用 MPS，都没有就用 CPU。代码其余部分不关心是哪一种：只要把模型和数据 `.to(device)` 搬过去。

2 **一个 batch 的 loss** 第 28–31 行

3 **估一次 loss** 第 34–44 行

4 **学习率：先预热，再余弦下降** 第 47–51 行

← 上一步 | 看整个文件 | 下一步 →

minigpt/g4/train.py 第 54–113 行 / 共 117 行 ⇆ 对照

```python
def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--steps", type=int, default=3000)
    ap.add_argument("--batch", type=int, default=16)
    ap.add_argument("--lr", type=float, default=1e-3)
    ap.add_argument("--warmup", type=int, default=100)
    ap.add_argument("--eval_every", type=int, default=100)
    ap.add_argument("--dropout", type=float, default=0.1)
    ap.add_argument("--out", default="gpt")
```

1 **超参数** 第 54–63 行

3000 步，批大小 16，峰值学习率 10^{-3}，预热 100 步，每 100 步估一次 loss，dropout 0.1。这些是小模型上常用的取值，没有仔细调过；读者可以用命令行参数改，比如 `--lr 3e-4`，对比曲线。

2 **数据** 第 65–72 行

3 **AdamW** 第 74–77 行

4 **一步训练** 第 81–94 行

5 **记录和样例** 第 95–106 行

6 **存下来** 第 107–113 行

← 上一步 | 看整个文件 | 下一步 →

跑起来

```
$ .venv/bin/python g4/train.py --out g4
```

⇆ 对照

```
设备 mps，参数 5,319,168；训练集每轮 147 个 batch，每个 batch 16 × 256 = 4,096 个 token
step     1  已看  0.00 M token  训练 loss 8.454  验证 loss 8.447  (2 s)
step   100  已看  0.41 M token  训练 loss 5.615  验证 loss 6.132  (21 s)
step   200  已看  0.82 M token  训练 loss 5.122  验证 loss 5.679  (41 s)
step   300  已看  1.23 M token  训练 loss 4.711  验证 loss 5.291  (60 s)
step   400  已看  1.64 M token  训练 loss 4.395  验证 loss 5.105  (79 s)
step   500  已看  2.05 M token  训练 loss 4.095  验证 loss 4.966  (98 s)
step   600  已看  2.46 M token  训练 loss 3.785  验证 loss 4.868  (118 s)
step   700  已看  2.87 M token  训练 loss 3.540  验证 loss 4.878  (137 s)
step   800  已看  3.28 M token  训练 loss 3.327  验证 loss 4.865  (156 s)
step   900  已看  3.69 M token  训练 loss 3.060  验证 loss 4.843  (175 s)
step  1000  已看  4.10 M token  训练 loss 2.888  验证 loss 4.901  (194 s)
    样例：KV Cache Cache 的空间，decode 时吞吐和 TPOT 几乎不变。 prefill 和 decode 每步只量化权重，decode 每步
step  1100  已看  4.51 M token  训练 loss 2.663  验证 loss 4.924  (214 s)
step  1200  已看  4.92 M token  训练 loss 2.480  验证 loss 4.984  (233 s)
step  1300  已看  5.32 M token  训练 loss 2.316  验证 loss 5.028  (252 s)
step  1400  已看  5.73 M token  训练 loss 2.114  验证 loss 5.120  (272 s)
step  1500  已看  6.14 M token  训练 loss 1.951  验证 loss 5.203  (291 s)
step  1600  已看  6.55 M token  训练 loss 1.798  验证 loss 5.236  (310 s)
step  1700  已看  6.96 M token  训练 loss 1.721  验证 loss 5.309  (328 s)
step  1800  已看  7.37 M token  训练 loss 1.604  验证 loss 5.393  (347 s)
step  1900  已看  7.78 M token  训练 loss 1.479  验证 loss 5.442  (366 s)
step  2000  已看  8.19 M token  训练 loss 1.404  验证 loss 5.485  (384 s)
    样例：KV Cache 也被别人的池子里的池子。 KV Cache 是计算受限的，是“我”，我 推理时的条件下，
step  2100  已看  8.60 M token  训练 loss 1.335  验证 loss 5.565  (403 s)
step  2200  已看  9.01 M token  训练 loss 1.267  验证 loss 5.605  (422 s)
step  2300  已看  9.42 M token  训练 loss 1.205  验证 loss 5.638  (441 s)
step  2400  已看  9.83 M token  训练 loss 1.165  验证 loss 5.704  (459 s)
step  2500  已看 10.24 M token  训练 loss 1.113  验证 loss 5.713  (478 s)
step  2600  已看 10.65 M token  训练 loss 1.070  验证 loss 5.751  (496 s)
step  2700  已看 11.06 M token  训练 loss 1.058  验证 loss 5.778  (514 s)
step  2800  已看 11.47 M token  训练 loss 1.031  验证 loss 5.799  (532 s)
step  2900  已看 11.88 M token  训练 loss 1.003  验证 loss 5.826  (551 s)
step  3000  已看 12.29 M token  训练 loss 0.981  验证 loss 5.857  (569 s)
    样例：KV Cache。 为什么今天的主权重 下面的代码完整行 前两篇到这里，进入不依赖的 KV Cache 默认的 KV
用时 569 s。验证 loss 最低 4.843（第 900 步），最后 5.857；存到 checkpoints/g4.pt
```

图 56-2　预训练 3000 步：训练 loss 从 8.45 降到 1.0，验证 loss 在第 900 步达到最低的 4.84，之后回升到 5.86。

先看第 1 步：训练、验证 loss 都是 8.45，比 ln 4096 = 8.318 略高一点。PyTorch 默认的初始化让打分有一些随机的起伏，比“完全平均”稍差，这是正常的。

然后 loss 迅速下降。前 100 步降到 5.6，模型学会了最容易的东西：哪些 token 常见（“的”“，”“。”“ ”）。到第 500 步，训练 loss 4.1、验证 loss 5.0。

下面这张图是完整的曲线。拖动滑块可以看每个记录点的数：

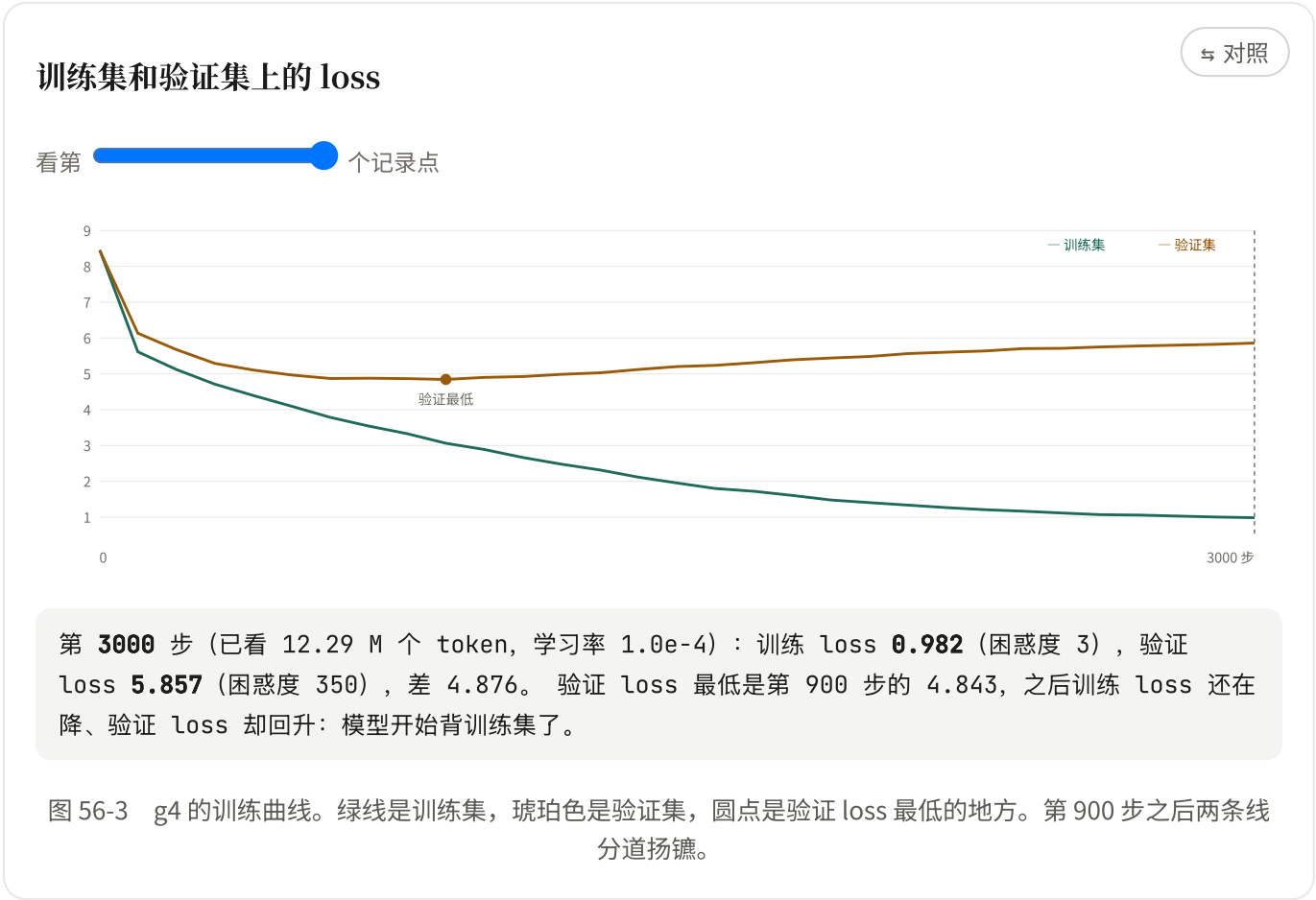


图 56-3　g4 的训练曲线。绿线是训练集，琥珀色是验证集，圆点是验证 loss 最低的地方。第 900 步之后两条线分道扬镳。

两条线为什么分开

第 900 步之前，两条线一起往下走：模型学到的东西（常用词、句式、哪些词常连在一起）在验证集上一样有用。第 900 步之后，训练 loss 继续下降，从 3.06 一路降到 1.0；验证 loss 却从 4.84 回升到 5.86。

这叫**过拟合**：模型开始背训练集了。训练 loss 1.0 意味着困惑度不到 3，模型几乎能逐字复述训练集里的句子；可验证集是它没见过的讲义，背下来的东西用不上，反而让它对没见过的句子更自信地猜错。验证集是语料的最后 10%，从第 126 讲的中间开始，到第 133 讲为止，正好是读

nano-vllm、vLLM、SGLang 源码的那几讲，话题和训练集也不完全一样，这让两条线的差距又大了一些。

为什么会这样？算一笔账。Chinchilla 论文给出的经验比例是：计算最优的训练，每个参数大约配 20 个 token（Scaling Laws 见第 46 讲）：

```
N = 5_319_168           # 参数量
D = 302_475             # 训练集 token 数
print(f"计算最优需要约 {20 * N / 1e6:.0f} M 个 token，我们只有 {D / 1e6:.2f} M，差 {20
* N / D:.0f} 倍")
seen = 3000 * 16 * 256
print(f"3000 步一共看了 {seen / 1e6:.1f} M 个 token，相当于把训练集过了 {seen / D:.0f}
遍")
print(f"验证 loss 最低的第 900 步：看了 {900 * 16 * 256 / 1e6:.2f} M 个 token，约 {900
* 16 * 256 / D:.0f} 遍")
```

```
计算最优需要约 106 M 个 token，我们只有 0.30 M，差 352 倍
3000 步一共看了 12.3 M 个 token，相当于把训练集过了 41 遍
验证 loss 最低的第 900 步：看了 3.69 M 个 token，约 12 遍
```

数据少了三百多倍。大模型预训练通常每个 token 只看一两遍，同一段话看第十几遍时，模型能学到的“规律”早就学完了，剩下的只有“这一段具体写了什么”。

让结论翻转的条件也就清楚了：**要么数据多，要么模型小**。语料扩大几百倍（比如把整个中文维基百科加进来），验证 loss 会和训练 loss 一起降下去很久；或者把模型缩小，它背不下那么多东西，过拟合就来得晚、来得轻。dropout、权重衰减这类正则化也能推迟过拟合，但它们只是让模型“背得慢一点”，不能凭空变出数据。下一讲会把 dropout 加大到 0.2 再跑一次，看看能推迟多少。

模型说话的样子

每 1000 步的样例，都是从“KV Cache”开始贪心生成 30 个 token：

- **第 1000 步**：“KV Cache Cache 的空间，decode 时吞吐和 TPOT 几乎不变。prefill 和 decode 每步只量化权重……”
- **第 2000 步**：“KV Cache 也被别人的池子里的池子。KV Cache 是计算受限的，是‘我’……”

- **第 3000 步**：“KV Cache。为什么今天的主权重 下面的代码完整行 前两篇到这里，进入不依赖的 KV Cache 默认的 KV……”

第 1000 步的句子读起来已经很像讲义了：有术语（decode、吞吐、TPOT、prefill），有标点，句子结构完整，只是内容经不起推敲。第 2000、3000 步反而更怪：这时模型已经在背训练集，它把几段不相干的原文片段拼在了一起，到第 3000 步连句子都不成形了。验证 loss 最低的时候，生成的句子往往也最“通顺”，这不是巧合。

> **要点**
> 语言模型预训练就是在每个位置上做“下一个 token 是哪个”的分类题，loss 是正确答案概率的负对数，起点约为 ln V。数据远少于参数量的 20 倍时，训练 loss 会一直降，验证 loss 先降后升：模型从学规律变成了背原文。

算力用了多少

最后从 Infra 的角度看一眼这次训练。第 35、46 讲说过，训练一个 token 大约要 6N 次浮点运算（前向 2N，反向 4N），N 是参数量：

```
N, tokens, seconds = 5_319_168, 3000 * 16 * 256, 569
flops = 6 * N * tokens
print(f"6N × token 数 = {flops:.2e} 次浮点运算")
print(f"用了约 {seconds} s，平均 {flops / seconds / 1e12:.2f} TFLOPS")
print(f"M2 的 10 核 GPU 标称 FP32 约 3.6 TFLOPS，利用率约 {flops / seconds /
3.6e12:.0%}")
```

```
6N × token 数 = 3.92e+14 次浮点运算
用了约 569 s，平均 0.69 TFLOPS
M2 的 10 核 GPU 标称 FP32 约 3.6 TFLOPS，利用率约 19%
```

不到五分之一。原因有好几个：6N 没有算注意力打分本身的运算（T = 256 时约占一成）；每 100 步要停下来估 40 个 batch 的 loss；矩阵太小（256 × 1024 这种尺寸，GPU 的计算单元喂不饱，第 62 讲的 Roofline）；还有 Python 和 PyTorch 每次调用的开销。大模型训练的利用率（MFU）通常在 40%–60%，靠的是大矩阵、算子融合和更少的调度开销，这些正是第三、四章讲的东西。

检查

```
$ .venv/bin/python tests/check.py g4
检查 g4:
  通过  g0 分词器：验证集编码再解码，和原文一字不差（49,922 个字符）
  通过  g0 分词器：没见过的字退回字节，也能还原
  通过  g1 数据：y 是 x 往后错一位（第 3 个样本从 12 开始）
  通过  g2 注意力：和 F.scaled_dot_product_attention 一致（最大差 8.9e-08）
  通过  g3 模型：输出形状 [B, T, V]
  通过  g3 模型：改第 10 个 token，前 10 个位置的输出不变（因果）
  通过  g4 训练：未训练时 loss 接近 ln V（相当于在 V 个 token 里瞎猜）（4.47 vs ln 64 =
4.16）
  通过  g4 训练：同一个 batch 训练 50 步，loss 降下来（4.47 → 0.16）
全部通过
```

图 56-4　g4 的检查：一个 V = 64 的小模型，起点 loss 接近 ln 64；在同一个 batch 上训练 50 步，loss 降到起点的一成以下。

第二条检查是调试训练代码最有用的一招：**拿一个 batch 反复训练，看 loss 能不能降到接近 0**。一个正常的模型一定能背下一个 batch；如果连这都做不到，问题一定出在代码里（梯度没清零、学习率没生效、目标错位），而不是数据或超参数。

容易写错的地方

- **忘了 `zero_grad`**。梯度一步步累加，相当于学习率越来越大，loss 先降后炸。
- **估 loss 时忘了 `eval()`**。dropout 还开着，估出来的 loss 偏高，而且每次不一样，训练、验证两条线的比较也失去意义。
- **验证集混进了训练集**。最常见的形式是先打乱再切分。验证 loss 会和训练 loss 一起降，看不出过拟合。

回顾

g4 加了 `train.py`：`loss_batch` 把 [B, T, V] 摊平后算交叉熵；`estimate_loss` 在 eval 模式下取 20 个 batch 的平均；学习率预热 100 步再余弦下降；AdamW 加权重衰减；梯度裁剪到 1。3000 步约 10 分钟，训练 loss 从 8.45 降到 1.0，验证 loss 在第 900 步达到最低 4.84，之后

回升到 5.86。数据只有计算最优的三百五十分之一，模型在训练集上过了 41 遍，后面三分之二的训练是在背原文。

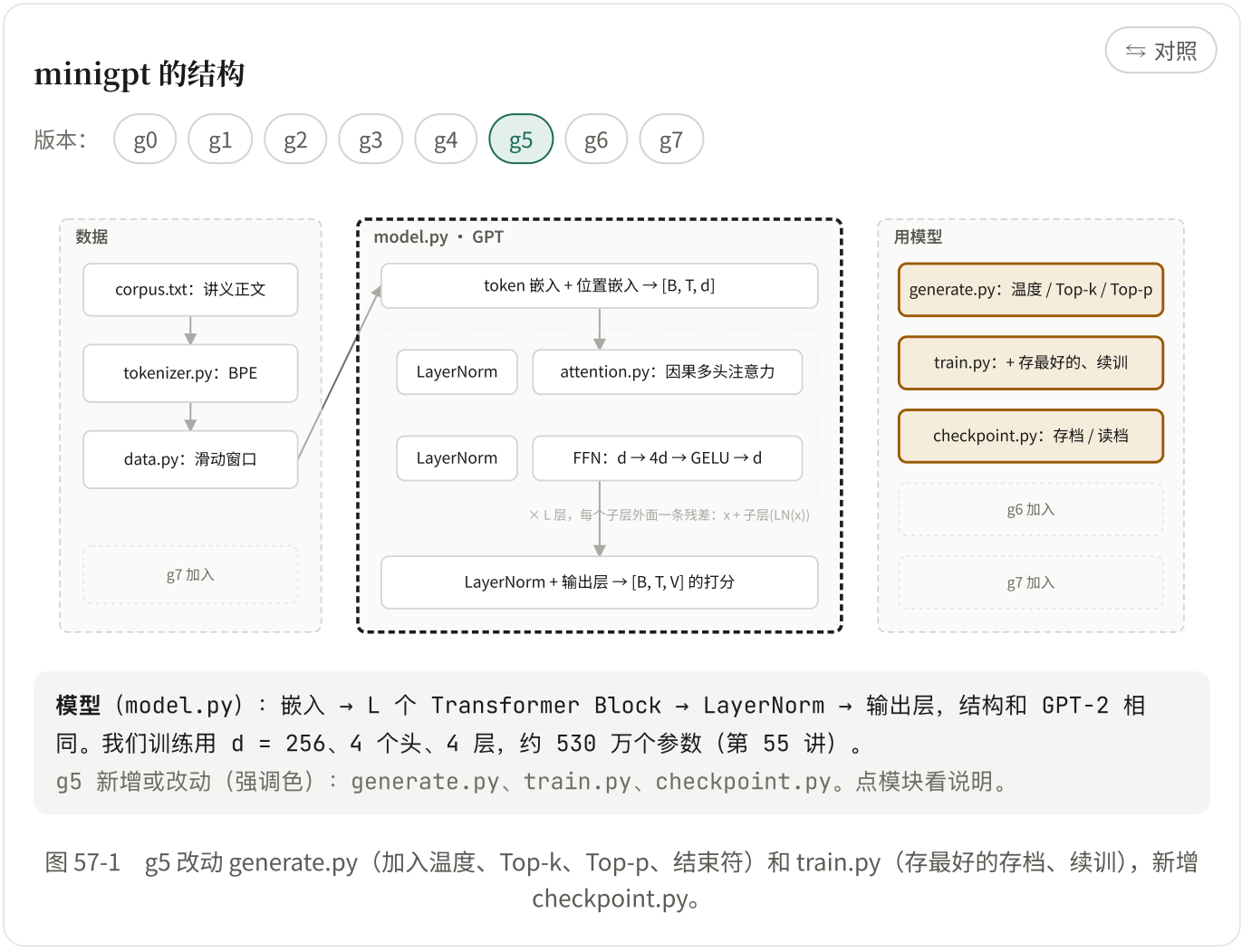
问题来了：我们最后存下的是第 3000 步的模型，验证 loss 5.86，而第 900 步那个更好的模型已经找不回来了。训练应该在什么时候存档？另外，到现在为止生成都是贪心的，同一个开头永远是同一句话。下一讲：**g5 · 解码与存档**。

第57讲 g5 · 解码与存档

07 概率与采样　24 自回归生成　56 g4 · 预训练　73 解码策略

上一讲留下两个毛病。第一，`train.py` 只在最后存一次档，存下的是第 3000 步、验证 loss 5.86 的模型；第 900 步那个验证 loss 4.84 的模型，训练过程中出现过，却没有留下来。第二，生成一直是贪心的：同一个开头，永远得到同一句话，而且这句话常常不太好。

g5 解决这两件事。存档这边：验证 loss 创新低时存一份，训练可以从存档接着跑；顺便把 dropout 从 0.1 加到 0.2，看能不能推迟过拟合。解码这边：加上温度、Top-k、Top-p 三种控制随机性的办法（原理在第 73 讲），在我们自己训练的模型上看它们的实际效果。



模型（model.py）：嵌入 → L 个 Transformer Block → LayerNorm → 输出层，结构和 GPT-2 相同。我们训练用 d = 256、4 个头、4 层，约 530 万个参数（第 55 讲）。

g5 新增或改动（强调色）：generate.py、train.py、checkpoint.py。点模块看说明。

图 57-1　g5 改动 generate.py（加入温度、Top-k、Top-p、结束符）和 train.py（存最好的存档、续训），新增 checkpoint.py。

checkpoint.py：存什么

minigpt/g5/checkpoint.py 共 20 行 ⇆ 对照

```
1  """存档和读档（第 57 讲）。
2
3  存档里放三样东西：模型配置、模型参数、优化器状态。只想用模型生成，前两样就够；
4  要接着训练，最好带上优化器状态（AdamW 给每个参数记了两份滑动平均，丢了它就要从 0 重新攒）。
5  """
6  import torch
7
8  from model import GPT, GPTConfig
9
```

1 三样东西放进一个字典 第 11-13 行

模型配置（重建同样形状的模型要用）、模型参数（`state_dict`，一个“名字 → 张量”的字典）、优化器状态，外加调用者想存的任何东西：这里会存步数、已看的 token 数和训练日志。

为什么要存配置？只存参数的话，读档时得先知道 d、H、L 是多少，才能建出形状对得上的模型。把配置和参数放在一起，存档就是自描述的。

2 读档 第 16-20 行

← 上一步 | 看整个文件 | 下一步 →

优化器状态是什么？AdamW 给每个参数记两个数：梯度的滑动平均（动量）和梯度平方的滑动平均（第 19 讲）。530 万个参数，就是 1060 万个额外的数，所以存档里优化器状态是模型参数的两倍大。只用模型生成，不需要它；要接着训练，最好带上，后面会实测不带会怎样。

train.py：存最好的、能续训

minigpt/g5/train.py 对比 minigpt/g4/train.py：+19 行 / −7 行 ⇆ 对照

```
+ """在讲义语料上预训练，加上存档、续训和按验证集挑最好的存档（第 57 讲）。
+ 用法：python g5/train.py [--steps 3000] [--resume checkpoints/gpt.pt]
+ 每 eval_every 步估一次 loss，记到 outputs/train_log_<out>.json；验证 loss 创新低时存 ch
- """在讲义语料上预训练（第 56 讲）。
- 用法：python g4/train.py [--steps 3000] [--out g4]
- 每 eval_every 步在训练集、验证集上各估一次 loss，记到 outputs/train_log_<out>.json；训练完
  import argparse
  import json
      …… 省略 5 行没变的代码
  import torch.nn.functional as F

+ import checkpoint
  from data import load_corpus, make_loader
  from generate import generate_greedy
      …… 省略 46 行没变的代码
      ap.add_argument("--dropout", type=float, default=0.1)
      ap.add_argument("--out", default="gpt")
+     ap.add_argument("--resume", default=None, help="从这个存档接着训练")
      a = ap.parse_args()

      …… 省略 7 行没变的代码
      val_loader = make_loader(val_ids, c.T_max, c.T_max, a.batch, shuffle=False)

+     os.makedirs(os.path.join(ROOT, "checkpoints"), exist_ok=True)
      model = GPT(c).to(device)
      opt = torch.optim.AdamW(model.parameters(), lr=a.lr, weight_decay=0.1)
+     log, step, seen = [], 0, 0
+     if a.resume:                                              # 续训：参数、优化器状态
+         model, ck = checkpoint.load(os.path.join(ROOT, a.resume), device)
+         opt = torch.optim.AdamW(model.parameters(), lr=a.lr, weight_decay=0.1)
+         opt.load_state_dict(ck["optimizer"])
+         log, step, seen = ck["log"], ck["step"], ck["seen"]
+         print(f"从 {a.resume} 接着训练：已经训练了 {step} 步")
      print(f"设备 {device}，参数 {sum(p.numel() for p in model.parameters()):,}；"
            f"训练集每轮 {len(train_loader)} 个 batch，每个 batch {a.batch} × {c.T_max}

      prompt = torch.tensor([tok.encode("KV Cache")], device=device)
+     best, t0 = min((r["val"] for r in log), default=float("inf")), time.time()
-     log, step, seen, t0 = [], 0, 0, time.time()
      while step < a.steps:
          for x, y in train_loader:
      …… 省略 15 行没变的代码
                  log.append({"step": step, "tokens": seen, "train": round(tr, 4), "v
                              "lr": opt.param_groups[0]["lr"]})
+                 if va < best:                                 # 验证 loss 创新低：存
+                     best = va
+                     checkpoint.save(os.path.join(ROOT, "checkpoints", f"{a.out}_bes
+                                     step=step, seen=seen, log=log)
```

```
114                     print(f"step {step:>5}  已看 {seen / 1e6:5.2f} M token  训练 loss {t
115                           f"  ({time.time() - t0:.0f} s)")
        …… 省略 3 行没变的代码
119                         model.train()
120                         print("    样例: " + s.replace("\n", " "))
121 +       checkpoint.save(os.path.join(ROOT, "checkpoints", f"{a.out}.pt"), model, opt, s
107 -       os.makedirs(os.path.join(ROOT, "checkpoints"), exist_ok=True)
108 -       torch.save({"config": c.__dict__, "model": model.state_dict(), "optimizer": opt
109 -                   os.path.join(ROOT, "checkpoints", f"{a.out}.pt"))
122       json.dump(log, open(os.path.join(ROOT, "outputs", f"train_log_{a.out}.json"), "
123       best = min(log, key=lambda r: r["val"])
        …… 省略 6 行没变的代码
```

图 57-2　g5 的 train.py 和 g4 的差别：多了 --resume 参数；验证 loss 创新低时存 _best.pt；存档改用 checkpoint.save，带上步数、token 数和日志。

改动有三处。

验证 loss 创新低时存档。每 100 步估一次验证 loss，比之前所有的都低，就存一份 `gpt_best.pt`。训练结束时，我们手里有两份：最后一步的 `gpt.pt`，和验证 loss 最低的 `gpt_best.pt`。这就是常说的**早停**（early stopping）的思路：不必真的停下训练，只要留住最好的那一份。

续训。`--resume` 读回存档里的参数、优化器状态、步数和日志，从那一步接着跑。学习率按步数算，所以续训后学习率曲线也是接着走的。训练跑到一半机器重启、或者想再多训几百步，都用得上。

存档里带上日志。续训之后，日志是接起来的一整条，曲线不会断。

加大 dropout 再训一次

```
$ .venv/bin/python g5/train.py --dropout 0.2
```

⇆ 对照

```
设备 mps，参数 5,319,168；训练集每轮 147 个 batch，每个 batch 16 × 256 = 4,096 个 token
step     1  已看  0.00 M token  训练 loss 8.454  验证 loss 8.447  (3 s)
step   100  已看  0.41 M token  训练 loss 5.654  验证 loss 6.166  (22 s)
step   200  已看  0.82 M token  训练 loss 5.189  验证 loss 5.776  (41 s)
step   300  已看  1.23 M token  训练 loss 4.817  验证 loss 5.393  (61 s)
step   400  已看  1.64 M token  训练 loss 4.548  验证 loss 5.168  (81 s)
step   500  已看  2.05 M token  训练 loss 4.273  验证 loss 5.037  (101 s)
step   600  已看  2.46 M token  训练 loss 4.004  验证 loss 4.927  (120 s)
step   700  已看  2.87 M token  训练 loss 3.800  验证 loss 4.921  (139 s)
step   800  已看  3.28 M token  训练 loss 3.633  验证 loss 4.907  (159 s)
step   900  已看  3.69 M token  训练 loss 3.432  验证 loss 4.876  (179 s)
step  1000  已看  4.10 M token  训练 loss 3.291  验证 loss 4.895  (198 s)
    样例：KV Cache Cache 是 1000000000000000000000000
step  1100  已看  4.51 M token  训练 loss 3.134  验证 loss 4.889  (218 s)
step  1200  已看  4.92 M token  训练 loss 2.989  验证 loss 4.914  (238 s)
step  1300  已看  5.32 M token  训练 loss 2.871  验证 loss 4.897  (258 s)
step  1400  已看  5.73 M token  训练 loss 2.713  验证 loss 4.939  (278 s)
step  1500  已看  6.14 M token  训练 loss 2.598  验证 loss 4.966  (298 s)
step  1600  已看  6.55 M token  训练 loss 2.479  验证 loss 4.983  (318 s)
step  1700  已看  6.96 M token  训练 loss 2.443  验证 loss 5.006  (336 s)
step  1800  已看  7.37 M token  训练 loss 2.357  验证 loss 5.043  (355 s)
step  1900  已看  7.78 M token  训练 loss 2.255  验证 loss 5.067  (373 s)
step  2000  已看  8.19 M token  训练 loss 2.201  验证 loss 5.082  (392 s)
    样例：KV Cache 的 KV Cache 大小，读回来就够了。 KV Cache 能同时服务多少个请求。第 71 讲
step  2100  已看  8.60 M token  训练 loss 2.144  验证 loss 5.132  (411 s)
step  2200  已看  9.01 M token  训练 loss 2.092  验证 loss 5.135  (430 s)
step  2300  已看  9.42 M token  训练 loss 2.036  验证 loss 5.149  (451 s)
step  2400  已看  9.83 M token  训练 loss 2.001  验证 loss 5.177  (471 s)
step  2500  已看 10.24 M token  训练 loss 1.974  验证 loss 5.181  (491 s)
step  2600  已看 10.65 M token  训练 loss 1.921  验证 loss 5.189  (511 s)
step  2700  已看 11.06 M token  训练 loss 1.918  验证 loss 5.219  (531 s)
step  2800  已看 11.47 M token  训练 loss 1.900  验证 loss 5.216  (550 s)
step  2900  已看 11.88 M token  训练 loss 1.857  验证 loss 5.237  (569 s)
step  3000  已看 12.29 M token  训练 loss 1.861  验证 loss 5.250  (587 s)
    样例：KV Cache Cache 是 128 KiB，能放约 1.8 GiB，能放约 12 GiB，8
用时 587 s。验证 loss 最低 4.876（第 900 步），最后 5.250；存到 checkpoints/gpt.pt
```

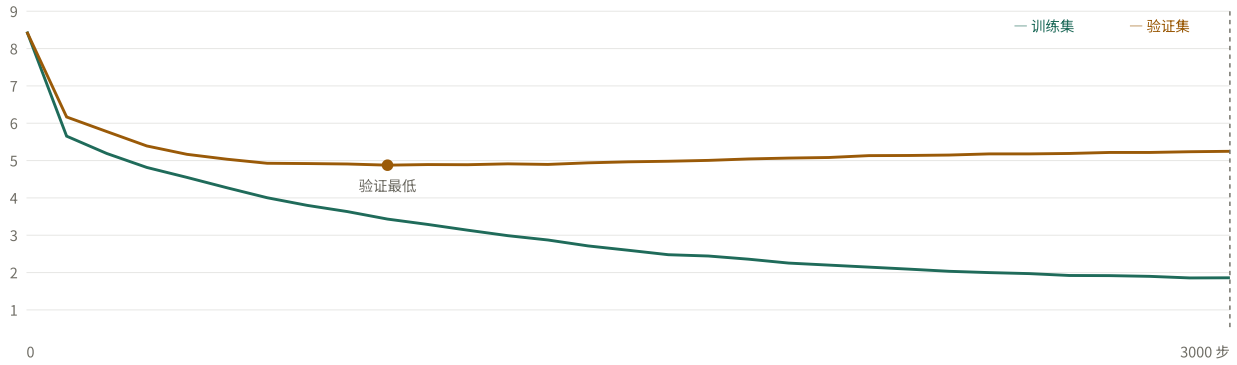
图 57-3　dropout 0.2 训练 3000 步：验证 loss 最低 4.876（第 900 步），最后 5.250。最好的存档存在 checkpoints/gpt_best.pt。

训练集和验证集上的 loss

⇆ 对照

g5：dropout 0.2　g4：dropout 0.1

看第 ●——— 个记录点



第 **3000** 步（已看 12.29 M 个 token，学习率 1.0e-4）：训练 loss **1.861**（困惑度 6），验证 loss **5.250**（困惑度 191），差 3.389。 验证 loss 最低是第 900 步的 4.876，之后训练 loss 还在降、验证 loss 却回升：模型开始背训练集了。

图 57-4　两次训练对比，用上面的按钮切换。dropout 0.2 让训练 loss 降得慢，验证 loss 回升得也慢，但验证 loss 的最低点并没有更低。

结果有点出乎意料。dropout 加大之后：

- 过拟合确实慢了：第 3000 步的训练 loss 是 1.86（g4 是 1.00），验证 loss 是 5.25（g4 是 5.86）。
- 可验证 loss 的**最低点**几乎没变，甚至略差：还是在第 900 步，4.876，g4 是 4.843。

也就是说，dropout 把“背原文”的速度拖慢了，却没让模型多学到一点“规律”。原因还是上一讲那笔账：数据只有计算最优的三百五十分之一，第 900 步时模型已经把训练集看了 12 遍，能从这 30 万个 token 里学到的东西差不多都学到了。正则化能决定模型“背得多快”，决定不了“能学到多少”。真想让验证 loss 更低，只能加数据。

所以这一讲之后，我们用的都是 `gpt_best.pt`：第 900 步、验证 loss 4.876 的那一份。

---

generate.py：温度、Top-k、Top-p

第 73 讲讲过这几种解码方式的原理。这里看代码：

minigpt/g5/generate.py 第 16–44 行 / 共 44 行 ⇆ 对照

```python
def filter_logits(logits, temperature=1.0, top_k=None, top_p=None):
    """logits: [B, V]。返回处理过的 logits，被去掉的候选设成 -∞。"""
    logits = logits / temperature                  # 温度 < 1 拉大差距，> 1 抹平差距
    if top_k is not None:
        kth = torch.topk(logits, top_k, dim=-1).values[:, -1:]          # 第 k 大的分
        logits = logits.masked_fill(logits < kth, float("-inf"))
    if top_p is not None:
        srt, order = torch.sort(logits, descending=True, dim=-1)
        cum = torch.softmax(srt, dim=-1).cumsum(dim=-1)
        drop = cum - torch.softmax(srt, dim=-1) > top_p                 # 前面的累计概
```

1 **温度** 第 16–18 行

打分除以温度 T，再做 softmax。T < 1 把打分之间的差距放大，分布变尖，高分的 token 更容易被选中；T > 1 把差距缩小，分布变平，低分的 token 也有机会。T → 0 就是贪心。

2 **Top-k** 第 19–21 行

3 **Top-p** 第 22–28 行

4 **生成循环** 第 31–44 行

← 上一步 | 看整个文件 | 下一步 →

在我们的模型上试试

`scripts/sampling_demo.py` 读入 `gpt_best.pt`，在 CPU 上跑（随机数固定，每次结果相同）：

```
$ .venv/bin/python scripts/sampling_demo.py                    ⇆ 对照

读入 checkpoints/gpt_best.pt：第 900 步的存档，验证 loss 4.876

“生成下一个 token 时，”之后，概率最高的 12 个 token：
  decode:0.032  所以:0.027  所有请求:0.018  再:0.018  TPOT:0.016  这:0.015  第:0.014  注
意力:0.014  每步:0.013  每个:0.012  prefill:0.011  模型:0.010
  这 12 个加起来 0.201，剩下 4084 个 token 分掉 0.799
  温度 0.5：最高的概率 0.164，凑够 90% 要 60 个 token
  温度 1.0：最高的概率 0.032，凑够 90% 要 573 个 token
  温度 1.5：最高的概率 0.011，凑够 90% 要 1270 个 token
  Top-k = 10：留下 10 个候选
  Top-p = 0.9：留下 573 个候选

从“KV Cache 的大小”开始各生成 60 个 token（随机的几种都用种子 0）：
【贪心】KV Cache 的大小。 ◁endoftext▷ ◁endoftext▷ ◁endoftext▷ ◁endoftext▷
◁endoftext▷ ◁endoftext▷ ◁endoftext▷ ◁endoftext▷ ◁endoftext▷ ◁endoftext▷
◁endoftext▷ ◁endoftext▷ ◁endoftext▷ ◁endoftext▷ ◁endoftext▷ ◁endoftext▷
◁endoftext▷ ◁endoftext▷ ◁endoftext▷ ◁endoftext▷ ◁endoftext▷ ◁endoftext▷
◁endoftext▷ 我们把 KV Cache Cache Cache Cache
【温度 0.7 + Top-k 40】KV Cache 的大小，看清楚： 第 1 讲说过 讲清楚：Llama-3-8B 的 个 token。
一张 8 token 在第 2000 亿 个 token，2N 亿，每个 token 10 token
【温度 1.0】KV Cache 的大小，看清楚：不必算一步 30 GiB、按应用，一个 400.056 个请求、平均 2.4
GbE 左右、2.3 ms、decode、2N 里，延迟前面每次差；D decode
【温度 1.5】KV Cache 的大小比较朴素面：不必单字（Sling 返回多段还按上下文便宜万卡的 正是每一块两
步、下面的adDecoding 发给专家大模型的参数和计算累加写缓存算出来的几百倍堆相加，存的加权等软烦里常只
实验每次快？。Mech
【温度 1.0 + Top-p 0.9】KV Cache 的大小，看清楚：为什么单字（30 讲），还需要一个用户看到第一个
KV Cache Cache。下面的平均参数都是 头 Cache 的空间，耗时也随长度线性增长。MoE 的 等分块 All-to-
Reduce（All 份）

只存参数：20.6 MiB；参数 + 优化器状态：61.2 MiB（AdamW 给每个参数存两份滑动平均）
重新读一遍存档，同一句话的 logits 完全一样：True
```

图 57-5　读入第 900 步的存档：下一个 token 的分布很平；贪心解码陷入了重复；加了随机性之后，句子才像讲义。

下一个 token 的分布非常平。“生成下一个 token 时，”之后，概率最高的是“decode”，也只有 0.032；前 12 个加起来才 0.201，剩下 80% 的概率分散在其余 4084 个 token 上。温度 1.0 时要 573 个 token 才能凑够 90% 的概率。这和验证 loss 4.876 是一致的：困惑度 e^4.876 ≈ 131，模型平均相当于在一百多个选项里犹豫。

这些候选本身是有道理的：decode、TPOT、prefill、注意力、每步、所有请求，都是“生成下一个 token 时”后面在讲义里真会出现的词。模型学到了“这里该谈推理”，只是拿不准具体谈什么。下面的演示用的就是这 12 个候选的真实打分，可以自己调温度、Top-k、Top-p 看看：



图 57-6　打分来自 gpt_best.pt。注意这里只取了概率最高的 12 个，再在它们之间归一化，所以显示的“原始概率”比上面输出里的大；真实的分布里，这 12 个只占 20%。

温度：0.5 时最高的概率从 0.032 涨到 0.164，凑够 90% 只要 60 个 token；1.5 时最高的只剩 0.011，要 1270 个。**Top-k = 10** 不管分布怎样都留 10 个；**Top-p = 0.9** 留下 573 个，因为分布太平了。在一个“拿不准”的模型上，Top-p 几乎不起作用，Top-k 反而是更有力的约束。这和大模型上的经验相反：大模型的分布通常很尖，Top-p 只留几个候选，比固定的 k 更灵活。

几种解码方式的输出，都从“KV Cache 的大小”开始：

- **贪心**：“KV Cache 的大小。”然后一连串的 `◁endoftext▷`，最后又回到“KV Cache Cache Cache Cache”。句号之后，模型认为最可能的是“换一篇文章”；换了文章，开头最可能的还是一个分隔符……贪心每一步都选最可能的，一旦进入这种自我强化的循环，就再也出不来。这是第 73 讲说的贪心的典型毛病，在小模型上格外明显。
- **温度 0.7 + Top-k 40**：“看清楚：第 1 讲说过 讲清楚：Llama-3-8B 的 个 token……”句子的碎片都像讲义（“第 1 讲说过”“Llama-3-8B”），连起来没有意思。
- **温度 1.0**：出现了“30 GiB”“2.4 GbE”“2.3 ms”“decode”：讲义里到处是带单位的数字，模型学会了这种“样子”。
- **温度 1.5**：分布被抹平，低概率的 token 大量出现，开始出现“Sling”“adDecoding”这样的拼凑和不成词的组合。
- **温度 1.0 + Top-p 0.9**：“为什么单字（30 讲），还需要一个用户看到第一个 KV Cache…… 耗时也随长度线性增长。”有几个片段相当像样。

这些结果说明，一个模型生成得好不好，不只看模型本身，**解码方式能把同一个模型变成“复读机”或“胡言乱语”**。实际服务里，温度和 Top-p 通常由调用方设置，推理引擎负责在采样这一步把它们实现对（第 73 讲；miniinfer 只做了贪心）。

要点

贪心容易陷进重复的循环；温度控制分布的尖平，Top-k 固定留 k 个，Top-p 留下累计概率够 p 的最少几个。模型越拿不准（分布越平），Top-p 留得越多、约束越弱。

存档有多大、读回来对不对

演示的最后一段：只存参数是 20.6 MiB（530 万 × 4 字节，加上名字等少量开销）；带上优化器状态是 61.2 MiB，正好约 3 倍，因为 AdamW 给每个参数存了两份滑动平均。重新读一遍存档，同一句话的 logits 逐位相同。

放大到真实模型：一个 70 亿参数的模型，BF16 的参数是 14 GB，训练时的存档还要加上 FP32 的主参数和两份 FP32 的滑动平均（每个参数 12 字节），一共约 100 GB。这就是为什么大模型训练的存档要分片、异步地写，存一次档本身就是个工程问题（ZeRO 把优化器状态分到各卡上，见第 103 讲）。

续训：优化器状态丢了会怎样

存档里的优化器状态占了三分之二的空间，真的有必要吗？`scripts/resume_demo.py` 从第 900 步的存档出发，用同样的数据顺序、同样的学习率接着训练 140 步，一次读回 AdamW 的状态，一次新建一个 AdamW：

```
$ .venv/bin/python scripts/resume_demo.py                    ⇆ 对照

从第 900 步的存档接着训练 140 步，每 20 步的平均训练 loss（含 dropout）：
步数          1-20     21-40     41-60     61-80     81-100   101-120   121-140
带上优化器状态    3.701     3.727     3.738     3.734     3.719     3.692     3.674
新建优化器       3.731     3.749     3.742     3.731     3.717     3.687     3.666
第 1 步的 loss：带状态 3.828，新建 3.828；第 2 步：3.754 vs 3.764
```

图 57-7　从第 900 步接着训练：带不带优化器状态，前 40 步的平均 loss 差约 0.02–0.03，60 步之后基本看不出差别。

结果和很多人的直觉相反：差别很小。第 1 步两者完全一样（3.828，这一步的 loss 是在更新之前算的）；前 20 步新建的优化器平均高 0.03，第 21–40 步高 0.02，第 61 步以后，新建的那一次反而略低 0.002–0.008，这已经在噪声的范围内了。

为什么？Adam 有“偏差修正”：滑动平均从 0 开始时，前几步会除以一个小于 1 的系数把它放大回来，所以新建的优化器从第一步起步长就是合理的，并不会因为“没有记忆”而乱走。它丢掉的只是过去的梯度方向信息，几十步就重新攒回来了。

那为什么还要存？一是**可复现**：带上状态，续训和不中断地训练是同一条轨迹，出了问题能对得上；二是在大模型、大学习率的训练里，前几十步的偏差可能触发 loss 尖峰，代价远比这里大。这个实验能说明的只是：在我们这个小模型上，丢了优化器状态不是灾难。

检查

```
$ .venv/bin/python tests/check.py g5
```

⇆ 对照

```
检查 g5:
  通过  g0 分词器：验证集编码再解码，和原文一字不差（49,922 个字符）
  通过  g0 分词器：没见过的字退回字节，也能还原
  通过  g1 数据：y 是 x 往后错一位（第 3 个样本从 12 开始）
  通过  g2 注意力：和 F.scaled_dot_product_attention 一致（最大差 8.9e-08）
  通过  g3 模型：输出形状 [B, T, V]
  通过  g3 模型：改第 10 个 token，前 10 个位置的输出不变（因果）
  通过  g4 训练：未训练时 loss 接近 ln V（相当于在 V 个 token 里瞎猜）（4.47 vs ln 64 =
4.16）
  通过  g4 训练：同一个 batch 训练 50 步，loss 降下来（4.47 → 0.16）
  通过  g5 解码：Top-k = 2 只留 2 个
  通过  g5 解码：Top-p = 0.8 留下累计概率刚好够 0.8 的几个（概率 [0.56, 0.21, 0.13, 0.08,
0.03]）
全部通过
```

图 57-8　g5 的检查：对一组 5 个打分，Top-k = 2 留下 2 个；Top-p = 0.8 留下累计概率 0.56、0.77、0.90 的前 3 个。

容易写错的地方

- **Top-p 的边界**。用“自己加上之后的累计概率 ⩾ p 就去掉”，会把刚好凑够 p 的那个 token 也去掉；分布特别尖时（第一个就超过 p），会把所有 token 都去掉，softmax 得到一串 NaN。要用“前面的累计概率”判断。
- **温度为 0 时除以 0**。单独处理成贪心。
- **续训时数据顺序和学习率没接上**。我们的 `--resume` 接上了步数和学习率，但数据加载器从新的一轮开头重新打乱，所以续训和不中断的训练不是逐位相同的。要完全复现，还得存随机数生成器的状态。

回顾

g5 新增 `checkpoint.py`（配置 + 参数 + 优化器状态），`train.py` 在验证 loss 创新低时存 `gpt_best.pt`、支持 `--resume` 续训，`generate.py` 加入温度、Top-k、Top-p 和结束符。dropout 加到 0.2，过拟合变慢了（最后的验证 loss 从 5.86 降到 5.25），最低点却没有更低（4.876，仍在第 900 步）。在第 900 步的存档上，下一个 token 的分布很平，贪心解码陷入了分隔符的循环，加上温度和 Top-k、Top-p 之后，生成的句子才像讲义。

到这里，模型只会“接着写”。下一讲先换个角度：同一份代码，能不能装下别人训练好的、大得多的模型？**g6 · 装进 GPT-2 的权重**。

第58讲 g6 · 装进 GPT-2 的权重

35 参数量与计算量　43 架构演进对照　55 g3 · 组装 GPT

我们的模型在 30 万个 token 上训练，十分钟就背下了训练集。OpenAI 的 GPT-2 small 是在约 40 GB 的网页文本（WebText）上训练的，参数是我们的 23 倍，训练用的 token 是我们的几万倍。我们没有条件训练它，但它的权重是公开的。

第 55 讲说过，GPT-2 small 的结构和我们的 `GPT` 类完全一样，只是尺寸不同：V = 50,257、T_max = 1024、d = 768、12 个头、12 层。那么，**能不能把它的权重原封不动地装进我们自己写的模型里？**能。这一讲就做这件事，重点是第二个问题：装完之后，怎么确认装对了。

这也是所有推理引擎都要做的事。第五章 miniinfer 的 `weights.py`（第 118 讲）、vLLM 里每个模型文件的 `load_weights`，做的都是同一件事：把别人训练好的权重文件，按名字对应到自己写的模型里。

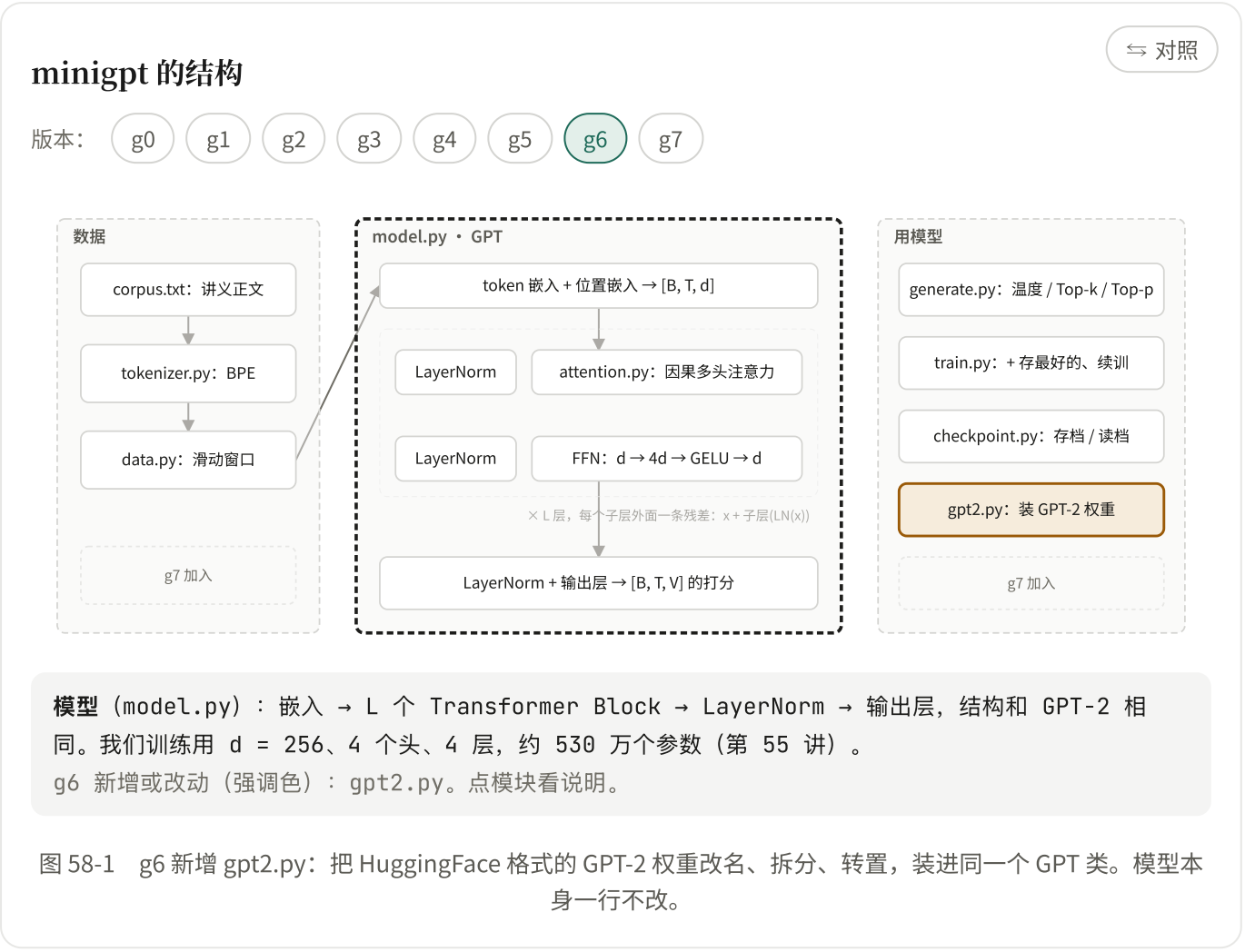


图 58-1　g6 新增 gpt2.py：把 HuggingFace 格式的 GPT-2 权重改名、拆分、转置，装进同一个 GPT 类。模型本身一行不改。

名字对上，形状对上

HuggingFace 的 GPT-2 权重是一个“名字 → 张量”的字典，一共 149 个张量。拿我们的模型对照，每个都能找到对应：

HuggingFace 的名字	我们的名字	要做什么
`wte.weight` [50257, 768]	`tok_emb.weight`	直接复制
`wpe.weight` [1024, 768]	`pos_emb.weight`	直接复制
`h.i.ln_1.weight` / `.bias`	`blocks[i].norm1.scale` / `.shift`	直接复制
`h.i.attn.c_attn.weight` [768, 2304]	`W_q`、`W_k`、`W_v` 的权重	按列切成三份，各自转置
`h.i.attn.c_attn.bias` [2304]	`W_q`、`W_k`、`W_v` 的偏置	切成三份
`h.i.attn.c_proj.weight` [768, 768]	`out_proj.weight`	转置
`h.i.mlp.c_fc.weight` [768, 3072]	`ff.net[0].weight`	转置
`h.i.mlp.c_proj.weight` [3072, 768]	`ff.net[2].weight`	转置
`ln_f.weight` / `.bias`	`final_norm.scale` / `.shift`	直接复制
`lm_head.weight`	`out_head.weight`	就是 `wte`，复制一份

有两处存法和我们不同。

第一，Conv1D 的权重要转置。GPT-2 最早是用 TensorFlow 写的，它的线性层叫 Conv1D（名字是历史遗留，其实就是线性层），权重存成 [输入, 输出]，计算是 $x \cdot W$。PyTorch 的 `nn.Linear` 存成 [输出, 输入]，计算是 $x \cdot W^T$。同一个变换，两种存法互为转置。HuggingFace 为了能直接读 OpenAI 原始的权重文件，保留了 Conv1D 的存法。

第二，q、k、v 拼在一起。GPT-2 用一个 [768, 2304] 的 `c_attn` 一次算出 q、k、v，2304 = 3 × 768，前 768 列是 q，接着 768 列是 k，最后 768 列是 v。我们的模型是三个分开的线性层，所以要按列切成三份。这和第 54 讲“所有头的投影合成一个矩阵”是同一个道理，只是再往前走了一步：把 q、k、v 三个矩阵也合成一个。vLLM 里叫 `QKVParallelLinear`，也是这样合并的，一次矩阵乘法比三次快。

还有一处配置要打开：GPT-2 的 q、k、v 投影带偏置，所以 `qkv_bias=True`。LayerNorm 的 $\varepsilon = 10^{-5}$ 、GELU 用 tanh 近似、方差除以 d，这些第 55 讲写模型时就是照 GPT-2 来的，不用改。

gpt2.py

minigpt/g6/gpt2.py 共 45 行 ⇆ 对照

```
1  """把 GPT-2 的权重装进我们的 GPT（第 58 讲）。
2
3  HuggingFace 的 GPT-2 和我们的结构一样，只是名字不同，还有两处存法不同：
4    1. 它的线性层是 Conv1D，权重存成 [输入, 输出]；nn.Linear 是 [输出, 输入]，要转置。
5    2. 它把 W_q、W_k、W_v 拼成一个 c_attn，[d, 3d]；要切成三份。
6  输出层和 token 嵌入共用一份权重（weight tying），文件里只有 wte。
7  """
8  import torch
9
10 from model import GPT, GPTConfig
```

1 从 HuggingFace 的配置换算 第 13-15 行

HuggingFace 的 `config.json` 里，词表叫 `vocab_size`，上下文叫 `n_positions`，隐藏维度叫 `n_embd`，头数、层数是 `n_head`、`n_layer`。换成我们的名字，打开 q/k/v 偏置；dropout 设成 0，因为只用来推理。

2 **一个带检查的复制函数** 第 18-27 行

3 **嵌入：直接复制** 第 29-32 行

4 **注意力：切开、转置** 第 33-38 行

5 **其余的线性层：转置** 第 39-42 行

6 **最后的 LayerNorm 和输出层** 第 43-45 行

← 上一步 | 看整个文件 | 下一步 →

不联网，怎么确认装对了

最直接的验证是：同一份权重，同一串输入，我们的模型和 HuggingFace 的 `GPT2LMHeadModel` 算出的 logits 应该一样。

但这需要先下载 GPT-2 的权重（约 550 MB）。其实不必：**要验证的是“结构和名字的对应关系”，不是权重本身**。transformers 可以按 GPT-2 small 的配置建一个随机初始化的模型，它的权重名字、形状、存法和真正的 GPT-2 完全一样，只是数值是随机的。用它的权重装进我们的模型，两边 logits 一致，就说明 `gpt2.py` 的每一个对应都是对的；换成真正的权重，还是同一套对应。

```
$ .venv/bin/python scripts/check_gpt2.py                    ⇆ 对照

HuggingFace GPT-2 small（随机初始化）：149 个张量，124,439,808 个参数
  transformer.wte.weight                  (50257, 768)
  transformer.h.0.attn.c_attn.weight      (768, 2304)
  transformer.h.0.attn.c_proj.weight      (768, 768)
  transformer.h.0.mlp.c_fc.weight         (768, 3072)
  lm_head.weight                          (50257, 768)
  lm_head 和 wte 是同一个张量：  True
装进我们的 GPT：163,037,184 个参数（输出层单独存了一份）

2 条 × 64 个随机 token：logits (2, 64, 50257)
  最大绝对误差 5.01e-06，每个位置的 argmax 都一样：True

故意不转置 out_proj：形状照样对得上、不报错，但 logits 最大误差 3.29e+00，argmax 一样的比例 0%
```

图 58-2　和 transformers 的 GPT-2 small 逐位比较：2 条 × 64 个 token、每个位置 50,257 个打分，最大差 5×10^{-6}，每个位置选出的 token 都一样。故意不转置一个方阵，形状照样对，结果全错。

149 个张量、1.24 亿个参数：这就是“GPT-2 124M”。`lm_head` 和 `wte` 的 `data_ptr` 相同，是同一块内存，印证了共用权重。装进我们的模型是 1.63 亿个参数，多出来的 3860 万就是复制出来的那份输出层，和第 55 讲算的一致。

最大绝对误差 5×10^{-6}。为什么不是 0？两边的数学一样，但运算的顺序不完全一样：Conv1D 用 `addmm` 一次算 $x \cdot W + b$，我们的 `nn.Linear` 走的是另一条路径；softmax、LayerNorm 的实现细节也不同。浮点数加法不满足结合律，加的顺序不同，最后一位就会有差别（第 10 讲）。12 层算下来，差别累积到 10^{-6} 量级，相对于几十量级的打分，是第 7 位有效数字的差别。判断标准是 argmax：每个位置选出的 token 都一样，这对生成来说才是重要的。

故意不转置 `out_proj`：这个矩阵是 768 × 768，转置前后形状一样，`put` 的形状检查拦不住，程序正常跑完。可 logits 的最大误差变成了 3.29，128 个位置里没有一个选对 token。这是装权重最危险的一类错误：**不报错，只是结果不对**。形状检查只能抓住一部分错误，最后一定要和参考实现逐位比对。第五章 miniinfer 每一版都和 HuggingFace 逐 token 比对，也是这个原因（第 117 讲）。

要点

装别人的权重，难点不在复制，而在对应：名字怎么改、哪些要切开、哪些要转置、哪些共用。方阵转置错了不会报错，所以一定要和参考实现比对输出；不需要真的下载权重，同结构的随机权重就能验证对应关系。

用真正的权重生成

验证完结构，换上真正的权重就能生成英文了。`scripts/gpt2_generate.py` 用 `huggingface_hub` 下载 `openai-community/gpt2` 的 `model.safetensors`（约 550 MB）和 `tokenizer.json`，装进我们的模型，再用上一讲写的 `generate` 生成：

```
.venv/bin/python scripts/gpt2_generate.py "Every effort moves you"
```

注意分词器也要换成 GPT-2 自己的：模型的嵌入表第 i 行，只对它自己的分词器的第 i 个 token 有意义。我们的 4096 词表分词器和它对不上。这里用 tokenizers 库直接读 GPT-2 的 `tokenizer.json`，它是第 21 讲说的字节级 BPE。

本文写作时没有在本机下载、运行这一步，所以这里不贴输出。上面随机权重的比对已经证明，装进去的模型和 transformers 的 GPT-2 逐位一致；换上真正的权重，生成的结果就是 GPT-2 small 本来的水平：语法通顺的英文，内容经常前后不一致。

检查

```
$ .venv/bin/python tests/check.py g6                                    ⇆ 对照

检查 g6:
  通过  g0 分词器：验证集编码再解码，和原文一字不差（49,922 个字符）
  通过  g0 分词器：没见过的字退回字节，也能还原
  通过  g1 数据：y 是 x 往后错一位（第 3 个样本从 12 开始）
  通过  g2 注意力：和 F.scaled_dot_product_attention 一致（最大差 8.9e-08）
  通过  g3 模型：输出形状 [B, T, V]
  通过  g3 模型：改第 10 个 token，前 10 个位置的输出不变（因果）
  通过  g4 训练：未训练时 loss 接近 ln V（相当于在 V 个 token 里瞎猜）（4.47 vs ln 64 =
4.16）
  通过  g4 训练：同一个 batch 训练 50 步，loss 降下来（4.47 → 0.16）
  通过  g5 解码：Top-k = 2 只留 2 个
  通过  g5 解码：Top-p = 0.8 留下累计概率刚好够 0.8 的几个（概率 [0.56, 0.21, 0.13, 0.08,
0.03]）
  通过  g6 GPT-2：和 HuggingFace 的 logits 一致（最大差 1.8e-07）
全部通过
```

图 58-3　g6 的检查：一个 2 层、64 维的小号 GPT-2，和 transformers 的 logits 差在 10^{-7} 量级。

检查里用的是一个 2 层、d = 64 的小号 GPT-2，几毫秒就跑完；它覆盖了和 GPT-2 small 完全相同的名字对应关系。

容易写错的地方

- **c_attn 按行切。** `split(d, dim=0)` 也能切出三个方阵，结果是错的。要按列（输出那一维）切。
- **方阵忘了转置。** `attn.c_proj` 是 d × d，形状检查拦不住，只能靠比对输出。
- **分词器没换。**用自己的分词器编码、喂给 GPT-2 的权重，模型看到的是一串毫无关系的编号，生成的是乱码；这种错误同样不会报错。
- **LayerNorm 的 ε 或 GELU 不一致。**ε 写成 10^{-6} 、GELU 用了精确版，误差会明显变大，但大多数位置的 argmax 可能还是对的，容易让人以为只是浮点误差。比对时要同时看最大误差和 argmax，误差比 10^{-5} 大很多就要查。

回顾

g6 加了 45 行的 `gpt2.py`：把 GPT-2 的 149 个张量改名，`c_attn` 按列切成 q、k、v，Conv1D 的权重转置，`wte` 同时复制进输出层，装进第 55 讲写的 `GPT` 类。和 transformers 的 GPT-2 small 比对，logits 最大差 5×10^{-6}，每个位置的 argmax 一致；忘了转置一个方阵，程序不报错，结果全错。

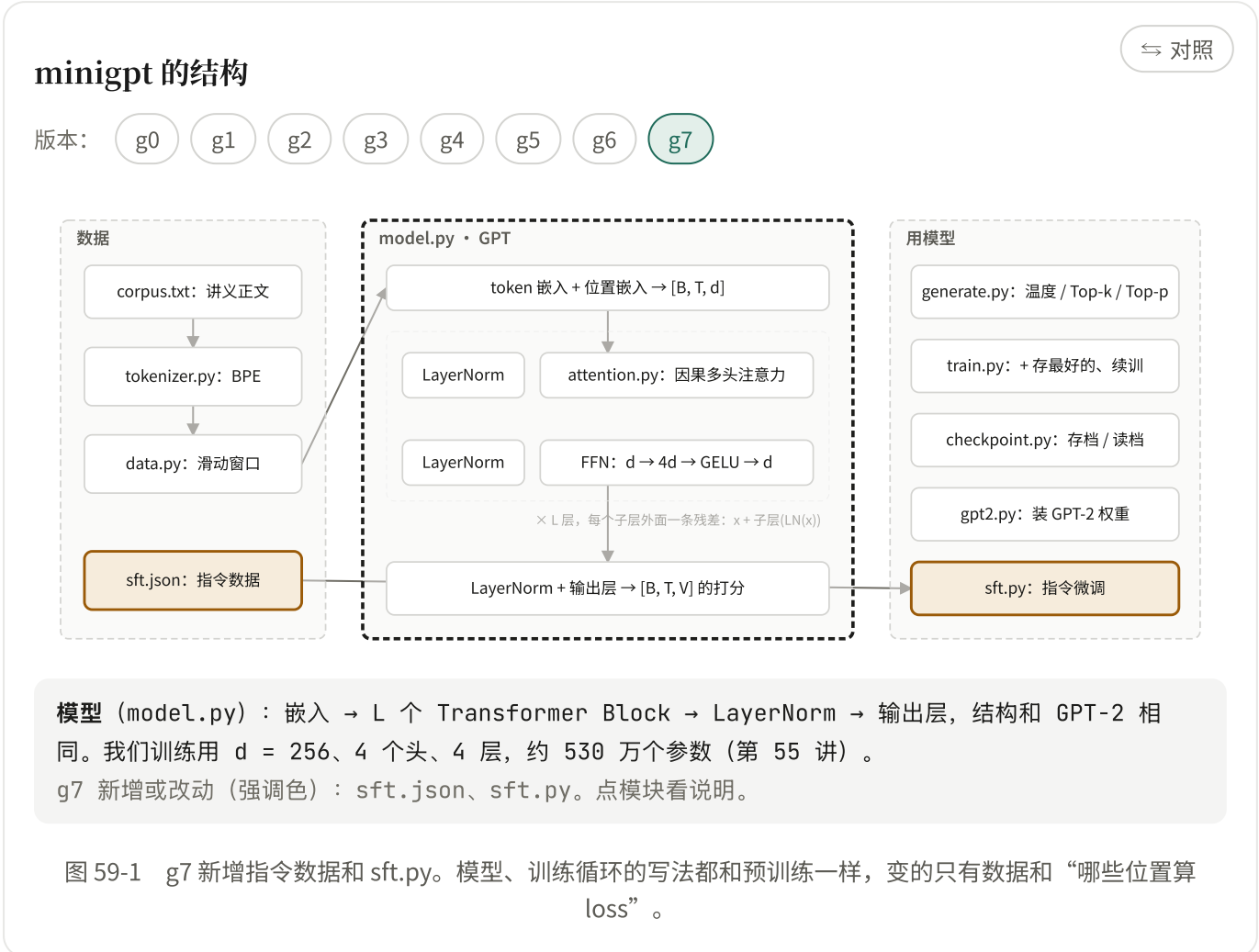
到这里，我们有了一个自己训练的、只会写讲义腔调的小模型，也能装进一个见过整个互联网的大一点的模型。但它们都只会“接着写”：给它一个问题，它不会回答，只会顺着问题的语气往下编。怎么让模型学会按指令回答？下一讲：**g7 · 指令微调**。

第59讲 g7 · 指令微调

47 监督微调 SFT　50 LoRA 参数高效微调　56 g4 · 预训练　57 g5 · 解码与存档

预训练出来的模型只会做一件事：接着写。给它“用一句话说明‘流水并行’解决什么问题。”，它不会回答，只会猜“这句话后面，讲义里一般接什么”。第 47 讲说过，要让模型学会“按指令回答”，要在预训练之后再做一步**监督微调**（SFT，也叫指令微调）：用一批“指令 → 回答”的样本接着训练。

这一讲是 minigpt 的最后一版 g7。我们用课程目录自己出题，在第 900 步的存档上做指令微调，然后诚实地看它学会了什么、没学会什么。



模型（model.py）：嵌入 → L 个 Transformer Block → LayerNorm → 输出层，结构和 GPT-2 相同。我们训练用 d = 256、4 个头、4 层，约 530 万个参数（第 55 讲）。

g7 新增或改动（强调色）：sft.json、sft.py。点模块看说明。

图 59-1　g7 新增指令数据和 sft.py。模型、训练循环的写法都和预训练一样，变的只有数据和“哪些位置算 loss”。

数据：用课程目录出题

指令微调要的是成对的数据。我们手边正好有一份：`site/src/data/course.ts` 里每一讲都有“技术点”和“解决什么问题”两栏。每讲出两道题：

- 技术点 → 问题：“用一句话说明‘KV Cache’解决什么问题。” → “怎么避免重复计算历史 token”
- 问题 → 技术点：“哪个技术点回答了：怎么避免重复计算历史 token？” → “KV Cache”

minigpt/data/make_sft.py 第 14–35 行 / 共 35 行 ⇆ 对照

```
14  HERE = os.path.dirname(os.path.abspath(__file__))
15  COURSE = os.path.join(HERE, "..", "..", "site", "src", "data", "course.ts")
16
17  rows = re.findall(r"^  '(\d+)\|([^|]+)\|([^|]+)\|[^']*',$", open(COURSE, encoding="u
18  items = []
19  for part, title, q in rows:
20      if int(part) ⩾ 15:                    # 第五章是项目讲，“问题”栏写法不同，不收
21          continue
22      title = re.sub(r"\s*⭐ | (选读) ", "", title).strip()
```

1 从课程目录里取两栏 第 17–24 行

正则表达式取出每一行的篇号、技术点、问题。第五章的项目讲（“v0 · 从权重文件到第一个 token”这类）问题栏的写法不同，不收；第 7 篇的讲名去掉 “g0 · ” 这样的前缀。去掉⭐ 和“(选读)”标记，剩下 112 讲。

2 每 10 讲留 1 讲做测试 第 26–32 行

← 上一步 | 看整个文件 | 下一步 →

```
$ .venv/bin/python data/make_sft.py
112 讲 → 训练 200 条，测试 24 条
例： {"instruction": "用一句话说明“路由与负载均衡”解决什么问题。", "output": "请求该发给哪台机器"}
```

⇆ 对照

图 59-2　112 讲出 224 道题，200 条训练、24 条测试。

两百条很少。真实的 SFT 数据集通常有几万到几十万条，而且题目五花八门（第 47 讲）。我们的题只有两种句式，答案都是课程目录里的短语。这决定了后面的结果：模型能学会的，只能是这两种句式和这 112 讲的对应关系。

提示模板，和只在回答上算 loss

每条样本拼成一段文字，用固定的模板标出哪里是指令、哪里是回答，回答后面接结束符：

```
### 指令：
用一句话说明“路由与负载均衡”解决什么问题。

### 回答：
请求该发给哪台机器◁endoftext▷
```

编码之后是 34 个 token，其中前 28 个是模板和指令，后 6 个是回答和结束符。

训练还是“预测下一个 token”，但有一处不同：**只在回答部分算 loss**。指令是用户给的，模型不需要学会“写出指令”；我们只关心看到指令之后，它能不能写出回答。做法是把 y 里落在指令部分的目标设成 −100，`F.cross_entropy` 遇到 −100 会跳过这个位置。batch 里长短不一的样本要补齐到一样长，补齐的位置同样设成 −100。

回答后面的结束符要算 loss：模型得学会“说完了就停”。上一讲给 `generate` 加的 `eos` 参数，就是在这里派上用场。

sft.py

minigpt/g7/sft.py 第 17–43 行 / 共 101 行 ⇆ 对照

```python
IGNORE = -100           # F.cross_entropy 会跳过目标等于 -100 的位置


def prompt_of(instruction: str) -> str:
    return f"### 指令：\n{instruction}\n\n### 回答：\n"


def encode_example(tok, ex):
    """返回整条序列的编号，以及回答从第几个 token 开始。"""
    p = tok.encode(prompt_of(ex["instruction"]))
```

1 **-100 和模板** 第 17–21 行

`IGNORE = -100` 是 PyTorch `cross_entropy` 默认的 `ignore_index`。模板用“### 指令：”“### 回答：”标出两部分，和书里的 Alpaca 格式一样，只是换成了中文。

2 **一条样本** 第 24–28 行

3 **补齐和屏蔽** 第 31–43 行

← 上一步 | 看整个文件 | 下一步 →

minigpt/g7/sft.py 第 46–97 行 / 共 101 行 ⇆ 对照

```python
def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--ckpt", default="checkpoints/gpt_best.pt")
    ap.add_argument("--epochs", type=int, default=20)
    ap.add_argument("--batch", type=int, default=8)
    ap.add_argument("--lr", type=float, default=5e-4)
    a = ap.parse_args()

    torch.manual_seed(123)
    device = pick_device()
```

1 从预训练的存档出发 第 46–60 行

读入上一讲选出的 `gpt_best.pt`（第 900 步）。微调不改模型结构，只是接着训练。学习率 5×10^{-4}，批大小 8，20 轮。

2 怎么“回答” 第 62–70 行

3 训练循环 第 72–84 行

4 评估 第 86–97 行

← 上一步 | 看整个文件 | 下一步 →

跑起来

```
$ .venv/bin/python g7/sft.py
```

```
训练 200 条，最长 51 个 token；测试 24 条
第 1 轮  平均 loss 5.227  (1 s)
第 2 轮  平均 loss 3.617  (1 s)
第 3 轮  平均 loss 2.821  (2 s)
第 4 轮  平均 loss 2.202  (2 s)
第 5 轮  平均 loss 1.735  (3 s)
第 6 轮  平均 loss 1.394  (3 s)
第 7 轮  平均 loss 1.099  (4 s)
第 8 轮  平均 loss 0.893  (4 s)
第 9 轮  平均 loss 0.727  (5 s)
第 10 轮  平均 loss 0.597  (6 s)
第 11 轮  平均 loss 0.445  (6 s)
第 12 轮  平均 loss 0.385  (7 s)
第 13 轮  平均 loss 0.325  (7 s)
第 14 轮  平均 loss 0.302  (8 s)
第 15 轮  平均 loss 0.243  (8 s)
第 16 轮  平均 loss 0.191  (9 s)
第 17 轮  平均 loss 0.168  (9 s)
第 18 轮  平均 loss 0.156  (10 s)
第 19 轮  平均 loss 0.135  (10 s)
第 20 轮  平均 loss 0.112  (11 s)

测试集（这些讲的题都没进训练集），贪心解码：
指令：用一句话说明“流水并行”解决什么问题。
  标准答案：怎么把不同层放到不同卡
  微调前：
  微调后：参数该往哪个方向改、改多少
指令：哪个技术点回答了：怎么把不同层放到不同卡？
  标准答案：流水并行
  微调前：
  微调后：SLO 与 All-Reduce
指令：用一句话说明“并行策略与拓扑映射”解决什么问题。
  标准答案：哪种并行该放在哪一层互联上
  微调前：
  微调后：怎么让模型学会对话和遵循指令
指令：哪个技术点回答了：哪种并行该放在哪一层互联上？
  标准答案：并行策略与拓扑映射
  微调前：
  微调后：超节点上的推理
指令：用一句话说明“三角函数与旋转”解决什么问题。
  标准答案：为位置编码和 RoPE 做准备
  微调前：“我 学 吃 学 AI”，“我 爱 爱 爱 学 爱 学 学 学
  微调后：有没有比注意力更省的结构
指令：哪个技术点回答了：为位置编码和 RoPE 做准备？
  标准答案：三角函数与旋转
  微调前：
  微调后：怎么进一步压缩 KV
```

```
训练集：198/200 条和标准答案一字不差
测试集：0/24 条和标准答案一字不差
训练集里的两条：
指令：用一句话说明“路由与负载均衡”解决什么问题。
  标准答案：请求该发给哪台机器
  微调后：请求该发给哪台机器
指令：哪个技术点回答了：请求该发给哪台机器?
  标准答案：路由与负载均衡
  微调后：路由与负载均衡
```

图 59-3　指令微调 20 轮：loss 从 5.23 降到 0.11。训练集 198/200 答得一字不差，测试集 0/24，但测试集的回答都是格式正确的“另一讲的答案”。

微调之前，6 道题有 5 道的回答是空的：模型看到“### 回答:”和换行之后，最可能的下一个 token 就是 `◁endoftext▷`。讲义里没有这种模板，它不知道这是什么，干脆认为“文章结束了”。剩下一道，它接着写出了全站贯穿的例句“我 爱 学 AI”那样的东西。

loss 从 5.23 一路降到 0.11。注意它比预训练的 loss 低得多：这里只算回答部分的几个 token，而且 200 条样本反复看了 20 遍。

训练集 198/200 一字不差：模型几乎把 200 条全背下来了。

测试集 0/24，一道也没答对。可是看具体的回答：

- 问“流水并行”解决什么问题，它答“参数该往哪个方向改、改多少”（这是第 17 讲梯度下降的答案）；
- 问“哪个技术点回答了：怎么把不同层放到不同卡？”，它答“SLO 与 All-Reduce”；
- 问“三角函数与旋转”，它答“有没有比注意力更省的结构”（第 44 讲混合架构与线性注意力的答案）。

格式全对：问“解决什么问题”，答的是一个“怎么……”“有没有……”式的问题；问“哪个技术点”，答的多是一个技术名词（6 道里只有最后一道答成了问题的样子）；说完都停下了。答案本身是训练集里别的讲的答案，或者几个技术名词的拼凑。

这就是指令微调能做和不能做的事。**它教会了模型“这种格式的问题该怎么答”：答什么类型的话、答多长、什么时候停。它没法教会模型不知道的东西。**测试集里那 12 讲的“技术点 ↔ 问题”对应关系，模型从来没见过；预训练时它读过这些讲的正文，但一个 530 万参数、在 30 万个 token 上训练的模型，没有能力从正文里归纳出“这一讲在解决什么问题”。它能做的，只有从背下来的 200 个答案里挑一个看起来合适的。

要点

指令微调和预训练用同一个训练循环，区别在数据和 loss 的位置：指令部分的目标设成 −100，只在回答和结束符上算 loss。它教会的是格式和“什么时候停”；知识还得来自预训练。

大模型的 SFT 之所以能“举一反三”，是因为预训练已经让模型知道了大量的事实和推理方式，SFT 只是教它用问答的形式把这些说出来。公开的研究里（例如 LIMA），约一千条精选的指令数据就能让一个强的基座模型学会对话，原因正在于此：要学的主要是格式。反过来，基座模型不知道的，SFT 的数据再多也只能让它背下来。

如果想在我们的设置下让测试集答对，可以试两个方向：让测试集的题在训练集里有“近亲”（比如同一讲换一种问法），或者换一个更强的基座模型，比如上一讲装进来的 GPT-2（它不懂中文，题目要换成英文）。还有一种更省的微调方式是 LoRA（第 50 讲）：冻结原来的参数，只训练每层旁边加的两个小矩阵，在只有 200 条数据时还能减少背题。这些留给读者去试。

检查

```
$ .venv/bin/python tests/check.py g7                                    ⇆ 对照

检查 g7:
  通过  g0 分词器：验证集编码再解码，和原文一字不差（49,922 个字符）
  通过  g0 分词器：没见过的字退回字节，也能还原
  通过  g1 数据：y 是 x 往后错一位（第 3 个样本从 12 开始）
  通过  g2 注意力：和 F.scaled_dot_product_attention 一致（最大差 8.9e-08）
  通过  g3 模型：输出形状 [B, T, V]
  通过  g3 模型：改第 10 个 token，前 10 个位置的输出不变（因果）
  通过  g4 训练：未训练时 loss 接近 ln V（相当于在 V 个 token 里瞎猜）（4.47 vs ln 64 =
4.16）
  通过  g4 训练：同一个 batch 训练 50 步，loss 降下来（4.47 → 0.16）
  通过  g5 解码：Top-k = 2 只留 2 个
  通过  g5 解码：Top-p = 0.8 留下累计概率刚好够 0.8 的几个（概率 [0.56, 0.21, 0.13, 0.08,
0.03]）
  通过  g6 GPT-2：和 HuggingFace 的 logits 一致（最大差 1.8e-07）
  通过  g7 微调：只有回答部分算 loss，补齐的位置不算（[[-100, -100, 4, 5], [-100, 3, -100,
-100]]）
全部通过
```

图 59-4　g7 的检查，也是整个 minigpt 的检查：从分词器到指令微调，每一版的性质都还成立。

最后一条测的是屏蔽的下标：一条 5 个 token、指令占 3 个的样本，y 应该是 [−100, −100, 4, 5]，第 4、5 个 token 算 loss；一条 3 个 token、指令占 2 个的样本补齐到 5，y 是 [−100, 3, −100, −100]。

容易写错的地方

- **屏蔽差一位**。见上面的推导。不报错，只是 loss 算在了错的位置上。
- **回答后面没接结束符**。模型学不会停，回答完会接着编下去，直到长度上限。
- **测试题和训练题来自同一讲**。同一讲正着问、反着问各一道，分在两边，测试集的成绩会虚高。
- **学习率照搬预训练**。在小数据上用 10^{-3} 训练，20 轮后训练集全对（我们试过：200/200），但参数离预训练的权重更远，模型原来“接着写讲义”的能力可能退化得更多（这一点我们没有测）。微调的学习率通常比预训练小。

回顾：minigpt 的全部

g7 用课程目录出了 224 道题（200 训练、24 测试，同一讲的题不跨集合），套进“### 指令 / ### 回答”模板，只在回答和结束符上算 loss，在第 900 步的预训练存档上微调 20 轮。训练集 198/200 答对，测试集 0/24，但测试集的回答格式全对：模型学会了怎么答，没有学会它不知道的东西。

这一篇写完了。回头看 minigpt：

- g0：讲义整理成 50 万字的语料，一个从字符出发、生僻字退回字节的 BPE 分词器，词表 4096。
- g1：在 30 万个 token 上开窗口，x、y 错一位；token 嵌入加位置嵌入。
- g2：因果多头注意力，和 PyTorch 自带的实现一致。
- g3：LayerNorm、GELU、FFN、残差组装成 530 万参数的 GPT。
- g4：预训练 3000 步约 10 分钟，验证 loss 在第 900 步最低，之后过拟合。
- g5：温度、Top-k、Top-p，存档、续训、留住最好的存档。
- g6：同一个 `GPT` 类装进 GPT-2 small，和 transformers 逐位一致。
- g7：指令微调。

g7 目录一共 627 行 Python（含注释），用到的每个零件，前面第 2–6 篇都单独讲过。从下一篇起，我们换到另一边：模型训练好之后，怎么让它在硬件上跑得快、跑得省。下一讲：**GPU 架构**。

AI INFRA · HANDBOOK

第 八 篇

硬件与性能基础

收录 8 讲 · 第 60–67 讲

第60讲 GPU 架构

03 矩阵乘法　13 神经元与线性层　35 参数量与计算量

第二章算过，一个大模型每生成一个 token，要做约 2N 次浮点运算（第 35 讲）：Llama-3-8B 是 160 亿次。这些运算几乎全是矩阵乘法（第 3、13 讲）。从这一讲开始的第三章讲单机推理，第一步要弄清楚承担这些运算的硬件：**GPU 为什么比 CPU 适合做矩阵运算？它的算力从哪里来?**

第 8 篇的八讲讲硬件和性能分析的基础：这一讲讲 GPU 的计算单元，第 61 讲讲存储，第 62 讲把两者合起来变成 Roofline 模型，后面几讲讲低精度、互联、Profiling 和实测。它们是第 9 到 14 篇所有性能估算的依据。

CPU 和 GPU：少数强核与大量简单核

CPU 的设计目标是**让一个任务尽快完成**。它有几个到几十个强大的核心，每个核心有复杂的分支预测、乱序执行、大容量的缓存，擅长处理逻辑复杂、前后依赖多的程序：操作系统、数据库、网页服务器。

GPU 的设计目标是**让大量相同的任务一起完成**。它有成千上万个简单的计算单元，每个单元没有那么聪明，但数量多。只要任务能拆成大量互不依赖、做同样运算的小份，GPU 就能一起算。

矩阵乘法正是这样的任务。$C = A \cdot B$ 的每一个元素都是一行和一列的点积，彼此完全独立；一个 4096×4096 的结果矩阵，就是 1600 多万个可以同时算的点积。

SM：GPU 的基本单元

GPU 由很多个相同的 **SM**（流式多处理器，streaming multiprocessor）组成。NVIDIA H100 有 **132 个** SM。每个 SM 可以看作一个小处理器，有自己的：

- **计算单元**：做矩阵乘加的 **Tensor Core**（每个 SM 4 个），做通用标量运算的 **CUDA 核**（每个 SM 128 个 FP32 核），以及加载存储单元、特殊函数单元（算指数、开方等）。
- **寄存器文件**：每个 SM 256 KB，分给在上面运行的各个线程。
- **L1 缓存 / 共享内存**：每个 SM 256 KB 的片上存储（第 61 讲）。

- **线程调度器**：每个 SM 分成 4 个分区，每个分区有一个调度器，每个周期挑一组线程（一个 warp，32 个线程，第 86 讲）发射指令。

所有 SM 共享一个约 50 MB 的 **L2 缓存**，再往外是 80 GB 的 **HBM 显存**。

下面的结构图可以点开看每个部件。



图 60-1　GPU 的结构（以 H100 SXM 为原型的示意）。整卡由 132 个 SM、共享的 L2 缓存和四周的 HBM 显存组成；每个 SM 分成 4 个分区，每个分区有一个 Tensor Core、32 个 FP32 核和寄存器，整个 SM 共享 L1 / 共享内存。

Tensor Core：专门做矩阵乘法的单元

早期的 GPU 只有 CUDA 核：每个核每个周期做一次乘加（$a \times b + c$），算矩阵乘法就是让大量的核各自做一个个乘加。

从 2017 年的 V100 开始，NVIDIA 加入了 **Tensor Core**：一种专门做**小矩阵乘加**的单元。它一次就能完成一小块矩阵的乘法再累加（V100 的是两个 4 × 4 的矩阵），相当于很多次乘加。H100 的每个 Tensor Core 每个周期能做 **512 次** BF16 乘加（稠密）。

于是峰值算力可以一层层乘出来：

$$132 \text{ 个 SM} \times 4 \text{ 个 Tensor Core} \times 512 \text{ 次乘加} \times 2\text{（一次乘加 = 2 次运算）} \times 1.83 \text{ GHz} \approx 989 \text{ TFLOPS}$$

这就是第 68 讲以来一直用的 989 TFLOPS（BF16，稠密）。同样的方法算 CUDA 核：132 × 128 × 2 × 1.83 GHz ≈ **62 TFLOPS**（官方的 67 TFLOPS 按更高的频率算），只有 Tensor Core 的 6%。

一个矩阵乘法要用上 Tensor Core，才能接近 GPU 的峰值。这也是为什么大模型的计算几乎都被组织成矩阵乘法：线性层、注意力的 QK^T 和 PV，都是。逐元素的运算（激活函数、归一化）只能用 CUDA 核，但它们的计算量很小，瓶颈在别处（第 83 讲）。

要点

GPU 由大量相同的 SM 组成（H100 有 132 个），每个 SM 有 Tensor Core、CUDA 核、寄存器和共享内存。Tensor Core 专门做小矩阵乘加，一个周期完成几百次乘加，H100 合计约 989 TFLOPS（BF16 稠密），是普通 CUDA 核的十几倍。矩阵乘法要用上 Tensor Core，才能接近峰值。

和 CPU 比一比

一颗 64 核的服务器 CPU，每个核有 2 个 AVX-512 向量单元，每个单元每周期做 16 个 FP32 乘加，按 2.5 GHz 算，约 **10 TFLOPS**。H100 的 Tensor Core 是它的近 **100 倍**。

算一个 4096 × 4096 × 4096 的矩阵乘法（约 1370 亿次运算），都按峰值：H100 的 Tensor Core 约 **0.14 ms**，只用 CUDA 核约 2.2 ms，CPU 约 13 ms。

GPU 的另一个特点是**同时在飞的线程数**。H100 每个 SM 最多驻留 2048 个线程，全卡约 27 万个；64 核的 CPU 开了超线程也只有 128 个。GPU 的单个线程很慢，访问显存要等几百个周期，但它不靠单个线程快，而是靠**线程多**：一个 warp 在等数据时，调度器立刻换另一个 warp 执行，计算单元始终有活干。这种“用并行掩盖延迟”的思路，是理解 GPU 性能的关键。

算力不是全部

峰值算力只是上限。要真的达到它，还需要两个条件：

1. **数据要供得上**。Tensor Core 一个周期吃进去的数据，要从显存、缓存里及时送到。如果每个数只用一两次就扔，计算单元大部分时间在等数据。下一讲讲存储，第 62 讲讲怎么判断一个运算是卡在算力还是卡在数据上。
2. **任务要足够多**。132 个 SM 都要有活干。第 85 讲的 decode 注意力里，批小时只有几个工作单元，大部分 SM 闲着，就是这个原因。

到了大模型里

每一代 GPU 的变化。从 V100、A100、H100 到 B200，SM 数量变化不大（80 到 140 多个），增长最快的是每个 Tensor Core 的能力和支持的精度：A100 支持 TF32、BF16，H100 加入 FP8，B200 加入 FP4。精度越低，算力越高（第 63 讲）。

稀疏的 Tensor Core。从 A100 开始，Tensor Core 支持 2:4 稀疏（第 98 讲），规格表上常写“稀疏算力”，是稠密算力的两倍。本站的估算一律用**稠密**算力。

别家的加速器也是类似的结构。AMD 的 MI300 系列、华为的昇腾、Google 的 TPU，名字不同（计算单元、矩阵单元、脉动阵列），思路一样：大量并行的单元 + 专门的矩阵乘法硬件 + 片上存储 + 高带宽显存。

用代码复现

下面的代码按 H100 的公开规格，一层层乘出 Tensor Core 和 CUDA 核的峰值算力；和一颗 64 核 CPU 对比；最后按峰值估算 4096^3 矩阵乘法在三者上的时间。

```
# H100 SXM 的峰值算力，从硬件结构一层层乘出来（规格取 NVIDIA 公开资料的数字，以白皮书为准）
SM, CLOCK = 132, 1.83e9                                  # SM 个数、Tensor Core 跑满时的时钟频率
TC_PER_SM, BF16_FMA_PER_TC = 4, 512                      # 每个 SM 4 个 Tensor Core；每个每周期做 512 次 BF16 乘加（稠密）
FP32_CORES_PER_SM = 128                                  # 每个 SM 128 个 FP32 CUDA 核，每周期 1 次乘加

tc = SM * TC_PER_SM * BF16_FMA_PER_TC * 2 * CLOCK        # 一次乘加 = 2 次浮点运算
cuda = SM * FP32_CORES_PER_SM * 2 * CLOCK
print(f"Tensor Core (BF16 稠密)：{SM} × {TC_PER_SM} × {BF16_FMA_PER_TC} × 2 × {CLOCK / 1e9:.2f} GHz = {tc / 1e12:.0f} TFLOPS")
print(f"普通 CUDA 核 (FP32)：{SM} × {FP32_CORES_PER_SM} × 2 × {CLOCK / 1e9:.2f} GHz = {cuda / 1e12:.0f} TFLOPS，只有 Tensor Core 的 {cuda / tc:.0%}")

# 对比：一颗 64 核的服务器 CPU（示意），每核 2 个 AVX-512 单元，每个每周期 16 个 FP32 乘加
cpu = 64 * 2 * 16 * 2 * 2.5e9
print(f"64 核 CPU (AVX-512, 2.5 GHz)：约 {cpu / 1e12:.0f} TFLOPS，H100 的 Tensor Core 是它的 {tc / cpu:.0f} 倍")

# 同时在飞的线程数：GPU 靠海量线程掩盖等待
print(f"H100 每个 SM 最多驻留 2048 个线程，全卡 {SM * 2048:,} 个；64 核 CPU 开超线程也只有 128 个")

# 4096 × 4096 × 4096 的矩阵乘法：2·n³ 次运算
n = 4096
flops = 2 * n**3
print(f"{n}³ 矩阵乘法 {flops / 1e9:.0f} GFLOP：H100 Tensor Core 约 {flops / tc * 1e3:.2f} ms，CUDA 核约 {flops / cuda * 1e3:.1f} ms，CPU 约 {flops / cpu * 1e3:.0f} ms（都按峰值）")
```

输出：

```
Tensor Core (BF16 稠密)：132 × 4 × 512 × 2 × 1.83 GHz = 989 TFLOPS
普通 CUDA 核 (FP32)：132 × 128 × 2 × 1.83 GHz = 62 TFLOPS，只有 Tensor Core 的 6%
64 核 CPU (AVX-512, 2.5 GHz)：约 10 TFLOPS，H100 的 Tensor Core 是它的 97 倍
H100 每个 SM 最多驻留 2048 个线程，全卡 270,336 个；64 核 CPU 开超线程也只有 128 个
4096³ 矩阵乘法 137 GFLOP：H100 Tensor Core 约 0.14 ms，CUDA 核约 2.2 ms，CPU 约 13 ms（都按峰值）
```

前两行是 GPU 的两种算力：989 TFLOPS 和 62 TFLOPS。第三行是 CPU：约 10 TFLOPS，差近 100 倍。第四行是同时驻留的线程数。最后一行是 4096^3 矩阵乘法的时间。

容易混淆的地方

1. **“CUDA 核数”不代表算力**。规格表上的 CUDA 核数（H100 是 16,896 个）是 FP32 标量单元的数量，大模型的矩阵乘法主要用的是 Tensor Core，它的算力要看 Tensor Core 的规格。
2. **峰值算力要看精度和稠密/稀疏**。同一张卡，FP64、TF32、BF16、FP8 的峰值差好几倍；规格表常把稀疏算力写在前面。比较时要对齐精度和是否稀疏。
3. **峰值只在理想条件下达到**。实际的矩阵乘法能达到峰值的 60% 到 80% 已经很好；第 67 讲会用实测来校准。

回顾

CPU 是少数强核，擅长复杂逻辑；GPU 是大量简单核，擅长大量相同、互不依赖的运算，矩阵乘法正是这种运算。GPU 由很多个 SM 组成（H100 有 132 个），每个 SM 有 Tensor Core、CUDA 核、寄存器和共享内存，所有 SM 共享 L2 和 HBM 显存。Tensor Core 专门做小矩阵乘加，H100 的峰值 $132 \times 4 \times 512 \times 2 \times 1.83\ \text{GHz} \approx 989$ TFLOPS（BF16 稠密），是 CUDA 核的十几倍、64 核 CPU 的近 100 倍。GPU 靠同时驻留的大量线程掩盖访存的等待。

Tensor Core 一个周期吃进去很多数据，这些数据从哪里来？显存有 80 GB，却比片上存储慢得多；片上存储很快，却只有几百 KB。数据在哪里、搬一次要多久？下一讲：**存储层级：HBM 与 SRAM**。

第61讲 存储层级

重点

60 GPU 架构

上一讲算出，H100 的 Tensor Core 每秒能做约 989 万亿次运算。可运算需要数据：矩阵乘法的每一次乘加，都要有两个数送到 Tensor Core 面前。这些数平时存在哪里？送过来要多久？

答案是：GPU 里有好几层存储，**离计算单元越近的越快，也越小**。大模型的权重有几十 GB，只能放在最外层的显存里；可显存相对于算力来说太慢了。这一讲把这几层存储的容量、带宽排出来，算一笔账：从显存每读一个字节，计算单元要等多久？这笔账是第 62 讲 Roofline 模型的基础，也贯穿了整个第三章。

从近到远

以 H100 为例，从计算单元往外数（数字是公开规格或常见测量的量级）：

寄存器。每个 SM 256 KB，全卡约 35 MB。每个线程的局部变量、矩阵乘法的累加结果放在这里。访问几乎没有延迟（约 1 个周期）。

L1 缓存 / 共享内存。每个 SM 256 KB 的 SRAM（静态随机存储器，做在芯片上），全卡也约 35 MB。其中最多 228 KB 可以由程序员当作“共享内存”直接管理（第 86 讲）。每个 SM 每周期能读写 128 字节，全卡合计约 31 TB/s。访问延迟约几十个周期。

L2 缓存。全卡共享，约 50 MB，也是芯片上的 SRAM。带宽约 5.5 TB/s（测量值的量级），延迟两三百个周期。从显存读来的数据先经过这里，被多个 SM 重复读时可以直接命中。

HBM 显存。高带宽内存（High Bandwidth Memory）：几个 DRAM 芯片垂直堆叠成一摞，放在 GPU 芯片旁边，通过很宽的接口相连。H100 有 80 GB，带宽 3.35 TB/s，访问延迟几百个周期。模型的权重、KV Cache、每层的激活都在这里。

CPU 内存。再往外，GPU 要经过 PCIe 访问 CPU 的内存：容量可以有 1 到 2 TB，可带宽只有约 64 GB/s（PCIe 5.0 x16，标称），比 HBM 慢了 50 倍。

一句话：**片上的 SRAM 快，但全卡只有几十 MB；HBM 有 80 GB，却慢了一个数量级。**

下面的演示把这几层的容量和带宽画在对数刻度上。调一调数据量，看从每一层读这么多数据要多久。

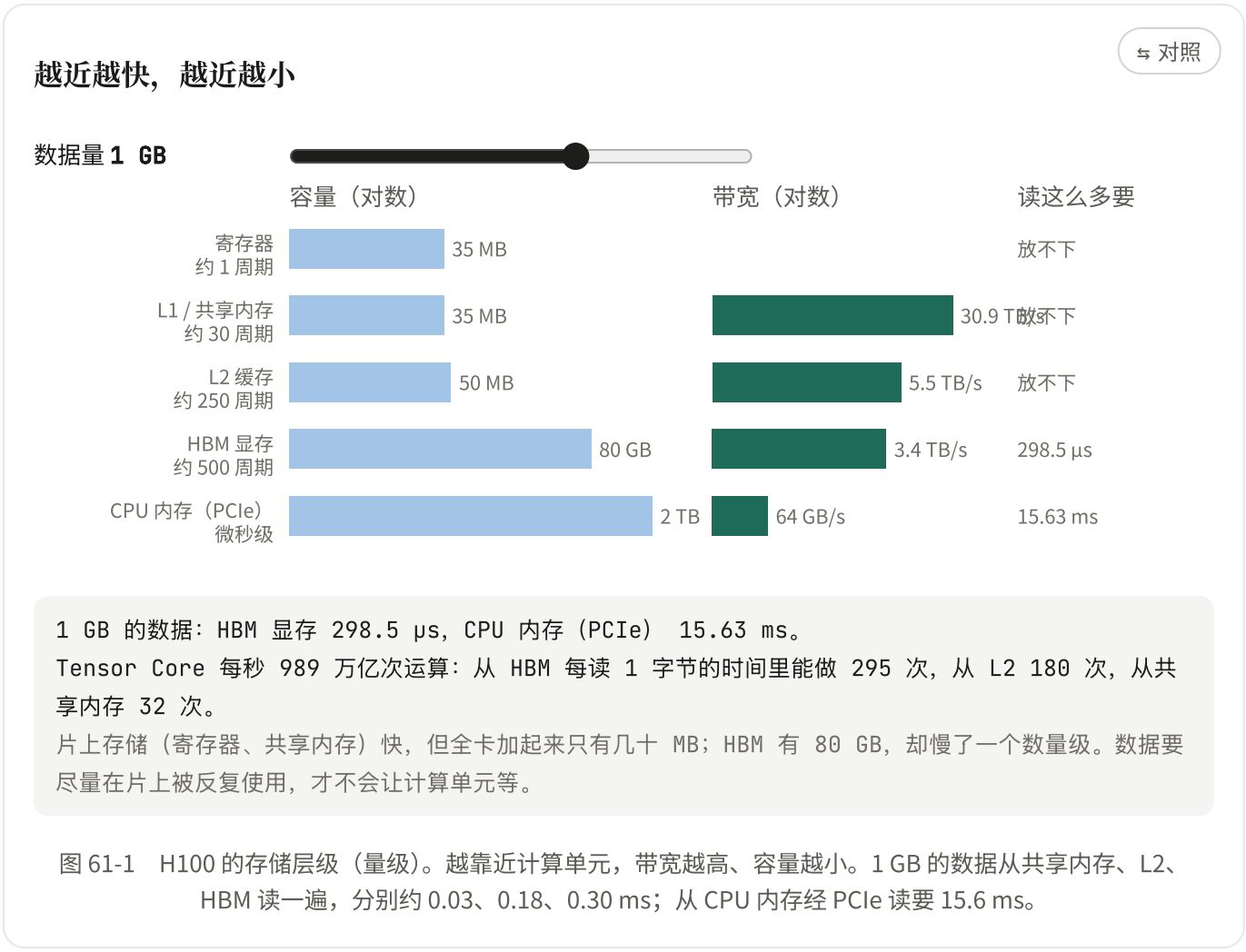


图 61-1　H100 的存储层级（量级）。越靠近计算单元，带宽越高、容量越小。1 GB 的数据从共享内存、L2、HBM 读一遍，分别约 0.03、0.18、0.30 ms；从 CPU 内存经 PCIe 读要 15.6 ms。

读 1 字节，能算多少次

现在把算力和带宽放在一起。H100 的 Tensor Core 每秒 989 万亿次运算，HBM 每秒 3.35 万亿字节。两者相除：

$$989\ \text{TFLOPS} \div 3.35\ \text{TB/s} \approx 295\ \text{次运算 / 字节}$$

意思是：**从显存每读 1 个字节的时间里，Tensor Core 能做约 295 次运算**。如果一个运算对读进来的每个字节只做不到 295 次运算，计算单元就会有一部分时间在等数据；只做 1 次，就有 99% 以上的时间在等。

同样的比例：从 L2 每读 1 字节能做约 180 次，从共享内存约 32 次。层级越近，这个比例越小，也就越容易“喂饱”计算单元。

两个例子

矩阵乘法。一个 $n \times n$ 的 BF16 矩阵乘法 $C = A \cdot B$，做 $2n^3$ 次运算，至少要从显存读 A、B，写 C，共 $3n^2$ 个数、$6n^2$ 字节。每字节的运算次数是 $2n^3 \div 6n^2 = n/3$：

- $n = 256$：约 85 次/字节，低于 295，**喂不饱** Tensor Core，会卡在显存带宽上。
- $n = 4096$：约 1365 次/字节，远高于 295，**喂得饱**。

所以大矩阵乘法能接近峰值算力，小矩阵乘法不能。注意这里说的是“至少”：实际实现中，A、B 的每个数要被用 n 次，如果每次都从显存重新读，读写量会大得多。这正是第 86 讲用共享内存分块的原因：把一小块数据搬进片上，在那里被反复使用。

逐元素运算。每个 BF16 数读一次、写一次（4 字节），只做 1 次运算，每字节 0.25 次，比 295 小了一千多倍。读写一个 1 GB 的张量要 0.60 ms，运算本身只要 0.5 μs。这类运算的时间几乎全是搬数据，所以第 83 讲要把它们融合起来，少读写几遍。

> **要点**
> GPU 的存储分好几层：寄存器和共享内存（片上 SRAM）最快，全卡只有几十 MB；L2 约 50 MB；HBM 显存 80 GB，带宽 3.35 TB/s，比片上慢一个数量级。H100 从 HBM 每读 1 字节的时间里能做约 295 次运算，所以一个运算要对每个读进来的字节做足够多的运算（让数据在片上被反复使用），才不会让计算单元空等。

为什么不把显存都做成 SRAM

既然 SRAM 这么快，为什么不多做一些？因为它**贵**：一个 SRAM 存储单元要 6 个晶体管，占的芯片面积是 DRAM 单元的好几倍。H100 的芯片面积有 800 多平方毫米，几十 MB 的片上存储已经占了相当一部分。要把 80 GB 都做成 SRAM，需要几千个这样大的芯片。

所以 GPU 的设计一直在平衡：在芯片上放多少 SRAM、多少计算单元，外面接多少 HBM、多快。每一代 GPU 的算力增长都比显存带宽快，“每读 1 字节能做的运算次数”越来越大，访存问题也越来越突出：A100 约 150 多，H100 约 295，B200 的 BF16 约 280，FP8 时还要翻倍（第 63 讲）。

到了大模型里

decode 卡在 HBM 带宽上。第 68 讲会算出，decode 每一步要把 16 GB 的权重从 HBM 读一遍，却只对每个字节做约 1 次运算，远低于 295。所以 decode 的速度几乎完全由显存带宽决定。

片上存储是优化的主战场。算子融合（第 83 讲）、FlashAttention（第 84 讲）、分块矩阵乘法（第 86 讲），做的都是同一件事：让数据留在片上，少碰 HBM。

新的显存技术。H200 把显存换成了 HBM3e，容量 141 GB、带宽 4.8 TB/s；B200 是 192 GB、约 8 TB/s。显存的容量和带宽，往往比算力更直接地决定推理的速度和成本。

用代码复现

下面的代码列出 H100 各层存储的容量、带宽、延迟（量级），算出从每层读 1 GB 要多久、每读 1 字节 Tensor Core 能做多少次运算；然后看两个矩阵乘法和逐元素运算的“每字节运算次数”。

```
# H100 SXM 的存储层级（容量、带宽取公开规格或常见测量的量级；片上带宽 = SM 数 × 每周期字节 × 频率）
SM, CLK, PEAK = 132, 1.83e9, 989e12
LEVELS = [
    # 名字，容量（字节），带宽（字节/秒），访问延迟（示意，GPU 周期）
    ("寄存器", SM * 256 * 1024, None, 1),
    ("L1 / 共享内存", SM * 256 * 1024, SM * 128 * CLK, 30),
    ("L2 缓存", 50e6, 5.5e12, 250),
    ("HBM 显存", 80e9, 3.35e12, 500),
    ("CPU 内存（经 PCIe）", 2e12, 64e9, None),
]
print("层级 | 容量 | 带宽 | 延迟 | 读 1 GB | 每读 1 字节，Tensor Core 能做几次运算")
for name, cap, bw, lat in LEVELS:
    cap_s = f"{cap / 1e6:.0f} MB" if cap < 1e9 else f"{cap / 1e9:.0f} GB"
    bw_s = f"{bw / 1e12:.2f} TB/s" if bw else "—"
    t = f"{1e9 / bw * 1e3:.2f} ms" if bw else "—"
    ratio = f"{PEAK / bw:.0f}" if bw else "—"
    print(f"{name} | {cap_s} | {bw_s} | {f'约 {lat} 周期' if lat else '微秒级'} | {t} | {ratio}")

# 复用：一个 n×n 的 BF16 矩阵乘法 C = A·B，A、B 至少要从显存读一遍
for n in [256, 4096]:
    flops, bytes_ = 2 * n**3, 3 * n * n * 2              # 读 A、B，写 C，各 n² 个 BF16
    print(f"{n}×{n} 矩阵乘法：{flops / 1e9:.2f} GFLOP，至少读写 {bytes_ / 1e6:.1f} MB，"
          f"每字节 {flops / bytes_:.0f} 次运算 → {'能' if flops / bytes_ > PEAK / 3.35e12 else '不能'}喂饱 Tensor Core（需要 {PEAK / 3.35e12:.0f}）")
# 逐元素运算（比如每个数乘一个常数）：每个 BF16 数读一次、写一次（共 4 字节），只做 1 次运算
print(f"逐元素运算：每 4 字节 1 次运算，每字节 0.25 次；读写 1 GB 的张量要 {2e9 / 3.35e12 * 1e3:.2f} ms，算只要 {0.5e9 / PEAK * 1e6:.1f} μs")
```

输出：

```
层级 | 容量 | 带宽 | 延迟 | 读 1 GB | 每读 1 字节，Tensor Core 能做几次运算
寄存器 | 35 MB | — | 约 1 周期 | — | —
L1 / 共享内存 | 35 MB | 30.92 TB/s | 约 30 周期 | 0.03 ms | 32
L2 缓存 | 50 MB | 5.50 TB/s | 约 250 周期 | 0.18 ms | 180
HBM 显存 | 80 GB | 3.35 TB/s | 约 500 周期 | 0.30 ms | 295
CPU 内存（经 PCIe） | 2000 GB | 0.06 TB/s | 微秒级 | 15.62 ms | 15453
256×256 矩阵乘法：0.03 GFLOP，至少读写 0.4 MB，每字节 85 次运算 → 不能喂饱 Tensor Core（需要 295）
4096×4096 矩阵乘法：137.44 GFLOP，至少读写 100.7 MB，每字节 1365 次运算 → 能喂饱 Tensor Core（需要 295）
逐元素运算：每 4 字节 1 次运算，每字节 0.25 次；读写 1 GB 的张量要 0.60 ms，算只要 0.5 μs
```

表格就是上面的五层存储：HBM 每字节 295 次，L2 180 次，共享内存 32 次。接着两行是矩阵乘法：256 的每字节 85 次，喂不饱；4096 的 1365 次，喂得饱。最后一行是逐元素运算：读写 1 GB 要 0.60 ms，算只要 0.5 μs。

容易混淆的地方

1. **“显存”和“内存”**。本站说的显存指 GPU 上的 HBM；内存指 CPU 那边的 DRAM。两者之间要经过 PCIe，慢得多。（有些设备，比如第 67 讲的 DGX Spark，CPU 和 GPU 共享同一块内存，情况不同。）
2. **带宽和延迟是两回事**。带宽是每秒能搬多少字节，延迟是一次访问要等多久。GPU 靠大量线程同时访问来掩盖延迟，所以对大多数大模型的运算，带宽才是关键。
3. **L2 和共享内存的带宽是量级**。它们取决于访问方式，不同的测量结果差别不小；本站只用它们说明数量级的关系，算时间时用的是 HBM 的带宽。

回顾

GPU 的存储从近到远：寄存器、L1/共享内存（片上 SRAM，全卡各约 35 MB，共享内存约 31 TB/s）、L2（约 50 MB、约 5.5 TB/s）、HBM 显存（80 GB、3.35 TB/s）、CPU 内存（经 PCIe 约 64 GB/s）。越近越快、越近越小，因为 SRAM 占的面积大、贵。H100 从 HBM 每读 1 字节的时间里能做约 295 次运算：$n \times n$ 矩阵乘法每字节约 $n/3$ 次，$n = 4096$ 时喂得饱 Tensor Core，$n = 256$ 时喂不饱；逐元素运算每字节只有 0.25 次，时间全花在搬数据上。

“每字节做多少次运算”这个量，决定了一个运算是卡在算力上还是卡在带宽上。把它画到一张图上，就能一眼看出任何一个运算的速度上限。下一讲：**Roofline 模型**。

第62讲 Roofline 模型

重点

11 计算量与 FLOPs　60 GPU 架构　61 存储层级

前两讲分别讲了 GPU 的算力（第 60 讲）和存储（第 61 讲）。一个运算在 GPU 上要多久，取决于两件事：要做多少次运算，要读写多少字节。做运算受算力限制，读写受带宽限制，两者同时进行，较慢的那个决定了总时间。

Roofline 模型（屋顶线模型）把这个道理画成了一张图：横轴是一个量，叫**算术强度**；纵轴是能达到的速度。任何一个运算，只要算出它的算术强度，就能在图上找到它的位置，一眼看出它卡在算力上还是带宽上，最快能有多快。第 68 讲以后的所有性能估算，用的都是这张图。

算术强度

一个运算做 W 次浮点运算（第 11 讲），从显存读写 Q 字节。它的**算术强度**是

$$I = W \div Q \text{（单位：FLOPs / 字节）}$$

意思是：每读写 1 字节，做多少次运算。第 61 讲已经算过几个：逐元素运算约 0.25，$n \times n$ 的矩阵乘法约 $n/3$。

两个屋顶

设峰值算力是 P（FLOPs/秒），显存带宽是 B（字节/秒）。

- 只看计算：W 次运算至少要 W/P 秒，速度不超过 P。
- 只看读写：Q 字节至少要 Q/B 秒。在这段时间里最多完成 W 次运算，速度不超过 $W \div (Q/B) = I \times B$。

两个限制同时存在，能达到的速度是较小的那个：

$$\text{能达到的速度} = \min(P, I \times B)$$

画在对数坐标上：I × B 是一条斜线（强度越高越快），P 是一条水平线。两条线组成一个“屋顶”的形状，运算的速度只能在屋顶下面。两条线的交点叫**屋脊点**：

$$\text{屋脊点 } I^* = P \div B$$

H100（BF16 稠密）的屋脊点是 989 TFLOPS ÷ 3.35 TB/s ≈ **295**，就是第 61 讲那个“每读 1 字节能做 295 次运算”。

- 强度**低于**屋脊点：落在斜线上，**卡在带宽**，算力用不满。
- 强度**高于**屋脊点：落在水平线上，**卡在算力**，带宽用不满。

把大模型的运算放上去

代码在 H100 上算了几个典型的运算：

- **逐元素运算**：强度约 0.25，最多 0.8 TFLOPS，只有峰值的 0.1%。
- **1 个 token 过一个线性层**（Llama-3-8B 的 4096 × 14336）：要把整个权重矩阵读一遍，却只对每个权重做一次乘加，强度约 **1.0**，最多 3.3 TFLOPS，峰值的 0.3%。这就是 decode 的处境（第 68 讲）。
- **32 个 token 一起过**：权重读一遍，被 32 个 token 共用，强度约 32，最多 106 TFLOPS，峰值的 10.7%。
- **256 个 token**：强度 237，最多 794 TFLOPS，峰值的 80%，还差一点到屋脊点。
- **4096 个 token**：强度 1792，远过屋脊点，可以达到 989 TFLOPS，**卡在算力上**。这是 prefill 的处境。
- **decode 的注意力**：每读一个历史 token 的 k、v，只和 1 个查询做运算，强度约 1；GQA 让 4 个查询共用一份 KV（第 39 讲），强度约 4，仍然深在带宽区。

规律很清楚：**对线性层来说，算术强度约等于一起处理的 token 数**。这也是第 70 讲“批处理能大幅提高 decode 吞吐”的原因：把更多请求放在一起，强度就往右移，速度沿着斜线往上爬，直到撞上算力的屋顶。

下面的演示画出了 H100、A100、B200 的屋顶。拖动算术强度看落在哪个区；拖动线性层的 token 数，看它的点怎么从带宽区移到算力区。

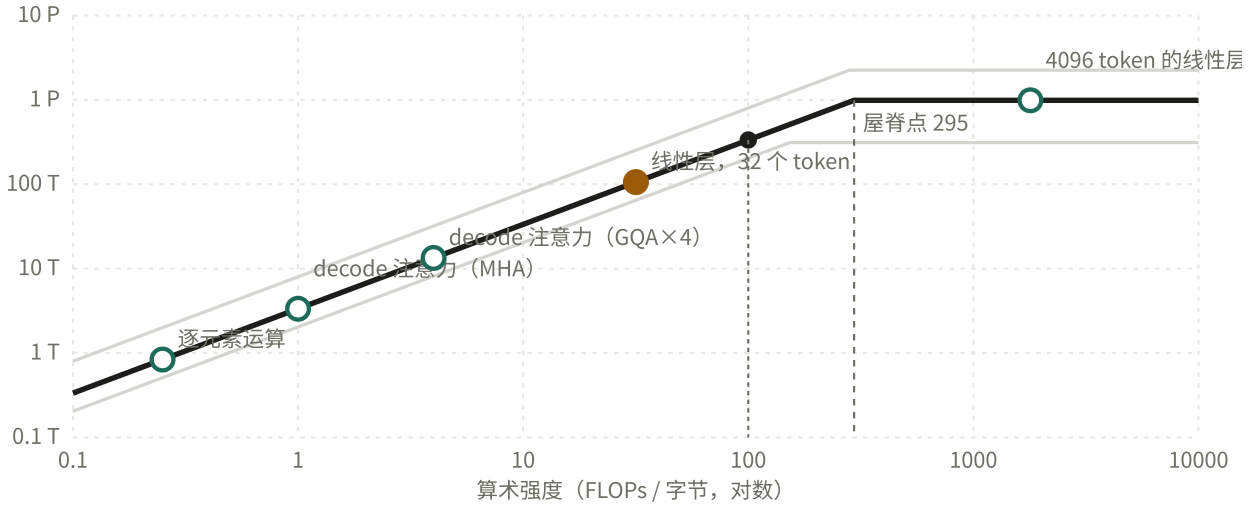
Roofline：卡在算力还是带宽

⇆ 对照

H100 | A100 | B200

算术强度 = **100**

线性层的 token 数 = **32**



H100：峰值 989 TFLOPS，带宽 3.35 TB/s，屋脊点 295 FLOPs/字节。
算术强度 100：最多 min(989, 100 × 3.35) = **335.0 TFLOPS**（峰值的 33.9%），在屋脊点左边，卡在**带宽**上。
斜线是带宽的屋顶（强度 × 带宽），水平线是算力的屋顶；两者的交点是屋脊点。

图 62-1　Roofline 模型（BF16 稠密，对数坐标）。斜线是带宽的屋顶，水平线是算力的屋顶，交点是屋脊点。decode（1 个 token）和逐元素运算深在带宽区；token 数增加，线性层的点沿斜线往上走，超过屋脊点后撞上算力的屋顶。

要点

算术强度 I = 运算次数 ÷ 读写字节数。一个运算最快只能达到 min(峰值算力, I × 带宽)。强度低于屋脊点（峰值算力 ÷ 带宽，H100 约 295）就卡在带宽上，高于就卡在算力上。线性层的强度约等于一起处理的 token 数：decode 的 1 个 token 深在带宽区，prefill 的几千个 token 在算力区。

怎么用这张图

判断该优化什么。一个运算在带宽区，再怎么优化计算也没用，要做的是减少读写（融合、量化、复用）或者提高强度（批处理、GQA）；在算力区，就要用更低的精度（第 63 讲）、更好的 Tensor Core 利用率。

估算时间。时间 = max(W ÷ P, Q ÷ B)。第 68 讲的 prefill、decode，第 70 讲的批处理，第 77、110 讲的各种估算，用的都是这个式子。

看离屋顶还有多远。实测的速度画到图上，和屋顶的距离就是还能优化的空间（第 67 讲）。如果一个带宽区的运算只达到了带宽屋顶的一半，说明数据搬运的方式还有问题。

比较硬件。A100 的屋脊点约 153，H100 约 295，B200 的 BF16 约 281。屋脊点越高，越难喂饱算力，越多的运算会落在带宽区。新一代 GPU 的算力增长通常比带宽快，这让访存优化越来越重要。

模型的局限

Roofline 是一个上限模型，它忽略了很多东西：

- 只考虑了一层存储（这里是 HBM）。也可以对 L2、共享内存各画一条屋顶（层次化 Roofline），更细地分析。
- 假设读写和计算可以完美重叠。实际的 kernel 有启动开销、同步、负载不均（第 89 讲），达不到屋顶。
- 峰值算力只对 Tensor Core 能处理的矩阵乘法有意义；逐元素运算用的是 CUDA 核，它的“屋顶”低得多（第 60 讲）。

但作为第一步的判断，它几乎总是对的。

用代码复现

下面的代码算出 H100、A100、B200 的屋脊点，再计算几个典型运算在 H100 上的算术强度、能达到的速度和瓶颈。

```python
# Roofline：一个运算能达到的速度 = min(峰值算力, 算术强度 × 显存带宽)
GPUS = {"H100": (989e12, 3.35e12), "A100": (312e12, 2.04e12), "B200": (2250e12,
8e12)}   # BF16 稠密算力、显存带宽

def linear(b, din=4096, dout=14336):                    # b 个 token 过一个线性层
 (Llama-3-8B 的 FFN 上投影)，BF16
    flops = 2 * b * din * dout
    bytes_ = 2 * (din * dout + b * din + b * dout)      # 读权重、读输入、写输出
    return flops, bytes_

OPS = {"逐元素运算（乘常数）": (1, 4)}
for b in [1, 32, 256, 4096]:
    OPS[f"线性层，{b} 个 token"] = linear(b)
OPS["decode 注意力（MHA，1 个查询）"] = (2 * 2 * 4096, 2 * 4096 * 2)  # 每个历史
token：q·k 和 p·v 各 1 次乘加；读 k、v 各 1 个数
OPS["decode 注意力（GQA，4 个查询共用 KV）"] = (4 * 2 * 2 * 4096, 2 * 4096 * 2)

for g, (F, BW) in GPUS.items():
    print(f"{g}：屋脊点 = {F / 1e12:.0f} TFLOPS ÷ {BW / 1e12:.2f} TB/s = {F /
BW:.0f} FLOPs/字节")
F, BW = GPUS["H100"]
print("H100 上各运算的算术强度和能达到的速度：")
for name, (fl, by) in OPS.items():
    ai = fl / by
    perf = min(F, ai * BW)
    print(f"  {name:22s}：强度 {ai:7.1f} → {perf / 1e12:6.1f} TFLOPS（峰值的 {perf /
F:5.1%}），卡在{'算力' if ai ⩾ F / BW else '带宽'}")
```

输出：

```
H100：屋脊点 = 989 TFLOPS ÷ 3.35 TB/s = 295 FLOPs/字节
A100：屋脊点 = 312 TFLOPS ÷ 2.04 TB/s = 153 FLOPs/字节
B200：屋脊点 = 2250 TFLOPS ÷ 8.00 TB/s = 281 FLOPs/字节
H100 上各运算的算术强度和能达到的速度：
  逐元素运算（乘常数）              ：强度      0.2 →     0.8 TFLOPS（峰值的   0.1%），卡在带宽
  线性层，1 个 token          ：强度      1.0 →     3.3 TFLOPS（峰值的   0.3%），卡在带宽
  线性层，32 个 token         ：强度     31.7 →   106.1 TFLOPS（峰值的  10.7%），卡在带宽
  线性层，256 个 token        ：强度    237.0 →   793.8 TFLOPS（峰值的  80.3%），卡在带宽
  线性层，4096 个 token       ：强度   1792.0 →   989.0 TFLOPS（峰值的 100.0%），卡在算力
  decode 注意力（MHA，1 个查询）：强度      1.0 →     3.4 TFLOPS（峰值的   0.3%），卡在带宽
  decode 注意力（GQA，4 个查询共用 KV）：强度      4.0 →    13.4 TFLOPS（峰值的   1.4%），卡
在带宽
```

前三行是三张卡的屋脊点。后面每一行是一个运算：逐元素运算和 1 个 token 的线性层只能用到峰值的千分之几；256 个 token 时达到 80%；4096 个 token 时卡在算力上。decode 注意力的强度只有 1 到 4。

容易混淆的地方

1. **算术强度看的是显存读写，不是总的数据访问**。同一个数在片上被用很多次，只算一次显存读写。所以好的实现（分块、融合）能提高实际的算术强度。
2. **屋脊点和精度有关**。FP8 的算力翻倍，屋脊点也翻倍（H100 约 590），同样的运算更容易落在带宽区。
3. **在带宽区“利用率低”不等于实现得差**。decode 只能用到峰值的不到 1%，这是运算本身的性质决定的，不是 bug；判断实现好坏，要看它离**自己那条屋顶**有多远。

回顾

Roofline 模型用算术强度 I（每读写 1 字节做多少次运算）判断一个运算的速度上限：min(峰值算力, I × 带宽)。两条线的交点是屋脊点，H100 约 295 FLOPs/字节。强度低于它卡在带宽，高于它卡在算力。线性层的强度约等于一起处理的 token 数：1 个 token 强度 1.0，只能用到峰值的 0.3%；4096 个 token 强度 1792，卡在算力。decode 的注意力强度只有 1 到 4。这张图告诉我们该优化什么、能快到哪里。

在算力区的运算，想再快只能提高算力。同一张卡，用 FP8 算比 BF16 快一倍，用 FP4 又快一倍。为什么精度越低越快？下一讲：**低精度计算**。

第63讲 低精度计算

10 浮点数与精度　60 GPU 架构　62 Roofline 模型

第 60 讲的 989 TFLOPS 有一个限定词：**BF16**。打开任何一张 GPU 的规格表，算力那一栏都按精度列了好几行，而且有一个很整齐的规律：H100 的 TF32 是 495 TFLOPS，BF16 是 989，FP8 是 1979；B200 再加上 FP4，又翻一倍。**精度每减半，算力就翻一倍**。

为什么会这样？精度低了，对显存、对推理速度又有什么影响？这一讲回答这两个问题。至于怎么在低精度下保住模型的效果，是第 12 篇（第 90–94 讲）的话题。

几种格式

第 10 讲讲过浮点数的结构：符号、指数、尾数。几种常见的格式（尾数位数含隐含的 1）：

格式	总位数	尾数位数	典型用途
FP32	32	24	优化器状态、累加
TF32	19（按 32 位存）	11	训练时 Tensor Core 处理 FP32 输入
BF16	16	8	训练和推理的主力
FP16	16	11	推理，旧一些的训练
FP8（E4M3）	8	4	推理、新的训练
FP4（E2M1）	4	2	推理（Blackwell 起）

为什么位数少就快

原因一：乘法器小。一个乘法器的核心是把两个尾数相乘。m 位乘 m 位的整数乘法器，面积大致和 m^2 成正比。BF16 的尾数 8 位，FP8 只有 4 位，乘法器的面积约是四分之一；FP32 的 24 位，是 BF16 的 9 倍。同样大小的芯片，位数越少，能放的乘法器越多，每个周期能做的乘加就越多。

（实际的算力只翻倍而不是四倍，因为 Tensor Core 里还有加法器、累加器、数据通路，这些部分不会按平方缩小；指数部分的处理也要面积。）

原因二：数据少。每个数从 2 字节变成 1 字节，同样的带宽每秒能送进来两倍的数；同样的寄存器、共享内存，能放两倍的数。计算单元算得更快，数据供应也跟得上。

原因三：能耗低。芯片的功耗很大一部分花在搬数据和翻转晶体管上，数据位数少，能耗也少。在功耗受限的数据中心里，这同样意味着更高的性能。

下面的演示列出了 H100 和 B200 各精度的算力，以及模型权重的大小。右边的方块示意乘法器的相对面积。

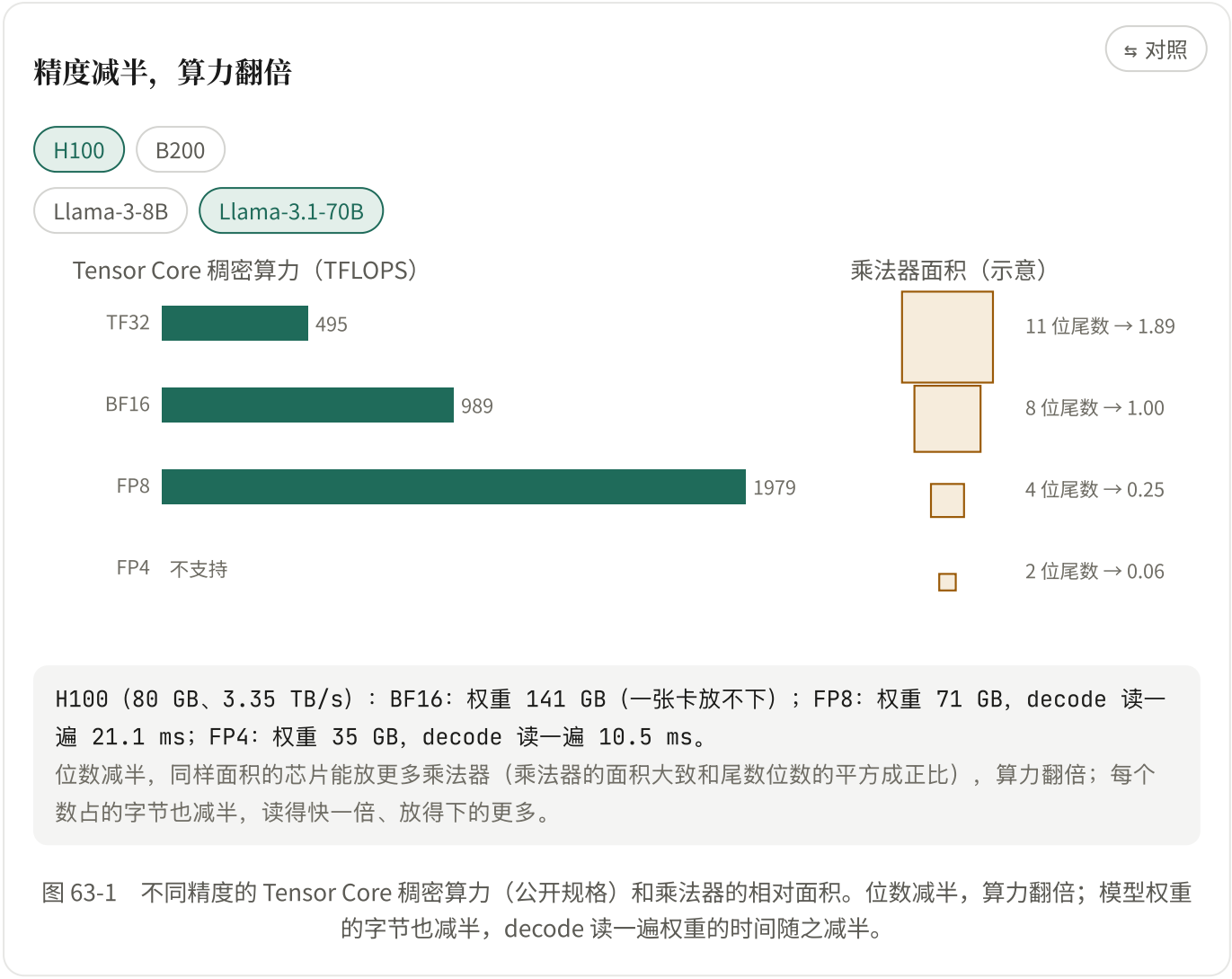


图 63-1　不同精度的 Tensor Core 稠密算力（公开规格）和乘法器的相对面积。位数减半，算力翻倍；模型权重的字节也减半，decode 读一遍权重的时间随之减半。

要点

精度越低越快，有三个原因：乘法器的面积大致和尾数位数的平方成正比，位数少就能放更多乘法器；每个数的字节少，同样的带宽和片上存储能处理更多的数；能耗更低。所以 GPU 的峰值算力按精度翻倍：H100 的 BF16 989、FP8 1979 TFLOPS，B200 再加上 FP4。

对推理的意义

低精度对推理的两个阶段都有帮助，原理不同：

decode：省的是带宽。decode 卡在读权重上（第 62 讲），用 FP8 存权重，读的字节减半，速度几乎翻倍。代码算了 Llama-3.1-70B 在 8 张 H100 上：BF16 权重 141 GB，decode 读一遍 5.3 ms；FP8 是 71 GB，**2.6 ms**。这种收益不需要硬件支持 FP8 计算，只要读进来再转回 BF16 计算就行（第 90 讲的“只量化权重”）。

prefill：省的是算力。prefill 卡在算力上，要变快必须让矩阵乘法**用 FP8 来算**，这需要硬件支持（H100 起）。同样的 70B，8 卡 prefill 8192 个 token，BF16 约 146 ms，FP8 约 **73 ms**。

此外，显存也省了一半：70B 的 FP8 版本，一张 H200（141 GB）就放得下，KV Cache 也有更多空间。

代价：精度

位数少了，能表示的数就少了，误差变大。FP8 E4M3 每个 2 的幂之间只有 8 个格点（第 93 讲），FP4 只有 8 个非负值。直接把 BF16 的模型转成低精度，效果可能明显下降。第 12 篇的量化方法（GPTQ、AWQ、SmoothQuant、分组缩放）就是为了在低精度下保住效果。

还有一个细节：矩阵乘法的**累加**仍然要用高精度。成千上万个乘积加在一起，如果每一步都舍入到 8 位，误差会累积得很大。所以 Tensor Core 做 FP8 乘法时，累加器至少是 FP16 或 FP32。

到了大模型里

每一代 GPU 都在加更低的精度。V100 引入 FP16 的 Tensor Core，A100 加入 TF32 和 BF16，H100 加入 FP8，B200 加入 FP4 和硬件分组缩放。算力的增长很大一部分来自精度的下降。

看规格表要对齐。比较两张卡时，要看同一种精度、同样是稠密的数字。“B200 的 FP4 算力 9000 TFLOPS”和“H100 的 BF16 算力 989 TFLOPS”不能直接比。

训练也在走向低精度。DeepSeek-V3 用 FP8 训练（第 93 讲），训练的大部分矩阵乘法也开始享受这个翻倍。

用代码复现

下面的代码列出几种格式的位数、尾数位数和乘法器面积的相对大小（按尾数位数的平方），对照 H100、B200 的稠密算力；最后算 Llama-3.1-70B 在 8 张 H100 上用 BF16 和 FP8 时的权重大小、decode 读一遍权重的时间和 prefill 的计算时间。

```
# 不同精度：每个数占几位、尾数（含隐含的 1）几位，以及 H100 / B200 的 Tensor Core 稠密算力
（公开规格，单位 TFLOPS）
FMT = {"FP32": (32, 24), "TF32": (19, 11), "BF16": (16, 8), "FP16": (16, 11), "FP8
E4M3": (8, 4), "FP4 E2M1": (4, 2)}
H100 = {"TF32": 495, "BF16": 989, "FP16": 989, "FP8 E4M3": 1979}
B200 = {"TF32": 1100, "BF16": 2250, "FP16": 2250, "FP8 E4M3": 4500, "FP4 E2M1":
9000}

print("格式 | 位数 | 乘法器面积（∝ 尾数位数²，相对 BF16） | H100 | B200")
for f, (bits, mant) in FMT.items():
    area = mant ** 2 / 8 ** 2
    print(f"{f:9s} | {bits:2d} | {area:5.2f} | {H100.get(f, '—'):>5} | {B200.get(f,
'—'):>5}")

# Llama-3.1-70B 在一组 8 张 H100 上：权重大小、decode 一步读权重的时间、prefill 8192
token 的计算时间
N, BW = 70.6e9, 8 * 3.35e12
for f, bytes_per in [("BF16", 2), ("FP8 E4M3", 1)]:
    F = 8 * H100[f] * 1e12
    print(f"70B {f}：权重 {N * bytes_per / 1e9:.0f} GB；decode 读一遍 {N * bytes_per
/ BW * 1e3:.1f} ms；prefill 8192 token 计算 {2 * N * 8192 / F * 1e3:.0f} ms")
```

输出：

```
格式 | 位数 | 乘法器面积（∝ 尾数位数²，相对 BF16） | H100 | B200
FP32      | 32 |  9.00 |     — |     —
TF32      | 19 |  1.89 |   495 |  1100
BF16      | 16 |  1.00 |   989 |  2250
FP16      | 16 |  1.89 |   989 |  2250
FP8 E4M3  |  8 |  0.25 |  1979 |  4500
FP4 E2M1  |  4 |  0.06 |     — |  9000
70B BF16：权重 141 GB；decode 读一遍 5.3 ms；prefill 8192 token 计算 146 ms
70B FP8 E4M3：权重 71 GB；decode 读一遍 2.6 ms；prefill 8192 token 计算 73 ms
```

表格里，乘法器面积 FP8 是 BF16 的 0.25，FP4 是 0.06；对应的算力分别翻一倍、两倍。最后两行：70B 用 FP8，decode 读一遍从 5.3 ms 降到 2.6 ms，prefill 从 146 ms 降到 73 ms。

容易混淆的地方

1. **存成低精度不等于用低精度算**。FP8 的权重可以读进来转成 BF16 再算，省带宽不省算力；要省算力，必须让 Tensor Core 直接做 FP8 乘法。
2. **FP16 和 BF16 算力相同，但不一样**。两者都是 16 位，FP16 尾数多、范围小，BF16 尾数少、范围和 FP32 一样大。训练大模型时 BF16 不容易溢出，所以成为主流。
3. **INT8 和 FP8 的算力在 H100 上一样**。选哪个，取决于数据的分布（第 93 讲）。

回顾

GPU 的峰值算力按精度翻倍：H100 的 TF32 495、BF16 989、FP8 1979 TFLOPS，B200 的 BF16 2250、FP8 4500、FP4 9000 TFLOPS。原因是乘法器的面积大致和尾数位数的平方成正比，位数少了能放更多乘法器；数据也少了一半，带宽和片上存储都能处理更多的数；能耗也更低。对推理来说，decode 用低精度存权重就能省一半带宽（70B 在 8 张 H100 上读一遍从 5.3 降到 2.6 ms），prefill 要用低精度计算才能省算力（146 降到 73 ms）。代价是精度，靠第 12 篇的量化方法来保住。

一张卡的算力和带宽讲完了。可大模型往往要用好几张卡，卡与卡之间、机器与机器之间怎么连？带宽和延迟各是多少？下一讲：**互联：PCIe、NVLink、InfiniBand、RoCE**。

第64讲 互联

重点

60 GPU 架构　61 存储层级

前几讲只看一张卡：算力、显存、带宽。可大模型的推理和训练常常要用到好几张卡，甚至成千上万张（第 99 讲）。卡与卡之间要交换数据：张量并行每层要把部分和加起来，专家并行要把 token 送到别的卡上，PD 分离要把 KV Cache 从一台机器传到另一台。

这些数据走什么路？有多快？这一讲把常见的几种互联讲清楚：**PCIe、NVLink、InfiniBand 和 RoCE**。它们是第四章所有通信估算的基础，也是第 65 讲超节点的铺垫。

PCIe：通用的总线

PCIe（PCI Express）是电脑里最通用的高速总线：GPU、网卡、SSD 都插在 PCIe 插槽上，通过它和 CPU 通信。它由若干条“通道”（lane）组成，GPU 一般用 16 条（x16）。

- PCIe 4.0 x16：每个方向约 **32 GB/s**；
- PCIe 5.0 x16：每个方向约 **64 GB/s**（H100 用的就是它）。

和 GPU 显存的 3.35 TB/s 比，PCIe 慢了 50 倍。数据从 CPU 内存来（加载模型、读回卸载的 KV），或者经网卡去别的机器，都要走 PCIe。

没有 NVLink 的机器（比如消费级显卡、一些推理卡），卡与卡之间也只能通过 PCIe 交换数据。

NVLink 与 NVSwitch：机内的高速路

NVLink 是 NVIDIA 专门为 GPU 之间通信设计的连接。每张卡有多条 NVLink 链路，合计的带宽远高于 PCIe：

- A100（NVLink 3）：每张卡双向合计 600 GB/s，单向约 **300 GB/s**；
- H100（NVLink 4）：双向 900 GB/s，单向约 **450 GB/s**；
- B200（NVLink 5）：双向 1.8 TB/s，单向约 **900 GB/s**。

一台 8 卡的服务器里，8 张卡并不是两两直连，而是都接到几颗 **NVSwitch** 交换芯片上，任意两张卡之间都能以满带宽通信。这 8 张卡组成一个 **NVLink 域**。第 105 讲的张量并行、第 101 讲的 All-Reduce，在机内都走这里。

NVLink 还有一个特点：它支持 GPU 直接读写另一张卡的显存（内存语义），就像访问自己的显存一样，只是远一点。这让卡与卡之间的协作可以非常细粒度。

InfiniBand 与 RoCE：机器之间的网络

机器之间要通过**网络**连接。大模型集群常用两种：

InfiniBand（IB）是为高性能计算设计的网络，延迟低、丢包少。常见的速率：HDR 200 Gb/s（约 25 GB/s）、NDR 400 Gb/s（约 **50 GB/s**）、XDR 800 Gb/s。一台 8 卡的 H100 服务器通常配 8 张 400 Gb/s 的网卡，每张 GPU 一张。

RoCE（RDMA over Converged Ethernet）在以太网上实现同样的 RDMA 功能，速率相当（400 GbE 约 50 GB/s）。以太网的生态更广、更便宜，云厂商用得很多；但以太网本来是允许丢包的，RoCE 要配置无损网络（流量控制、拥塞控制），调好了性能接近 InfiniBand。

两者的核心都是 **RDMA**（远程直接内存访问）：网卡直接读写对方机器的内存，不经过对方的 CPU 和操作系统，延迟低、不占 CPU。再加上 **GPUDirect RDMA**，网卡可以直接读写 GPU 的显存，数据不用先拷到 CPU 内存。

注意单位：网络的速率常用比特（Gb/s），除以 8 才是字节（GB/s）。400 Gb/s 的网卡是 50 GB/s。

一条路径的带宽，取决于最慢的那一段

数据从一张卡到另一张卡，往往要经过好几段。代码算了一台 8 卡 H100 服务器里的几条路径：

- **同机两卡，走 NVLink**：瓶颈 450 GB/s，传 1 GB 约 2 ms。
- **同机两卡，只有 PCIe**：瓶颈 64 GB/s，约 16 ms。
- **跨机两卡**：GPU → PCIe → 网卡 → 交换机 → 网卡 → PCIe → GPU，瓶颈是网卡的 50 GB/s，约 20 ms。
- **GPU 和 CPU 内存**：PCIe，约 16 ms。

把整台机器加起来看：NVLink 每卡 450 GB/s × 8 = 3600 GB/s，网卡 8 × 50 = 400 GB/s，**机内比机间快 9 倍**。这个差距决定了第 109 讲的原则：通信频繁的并行要放在机内。

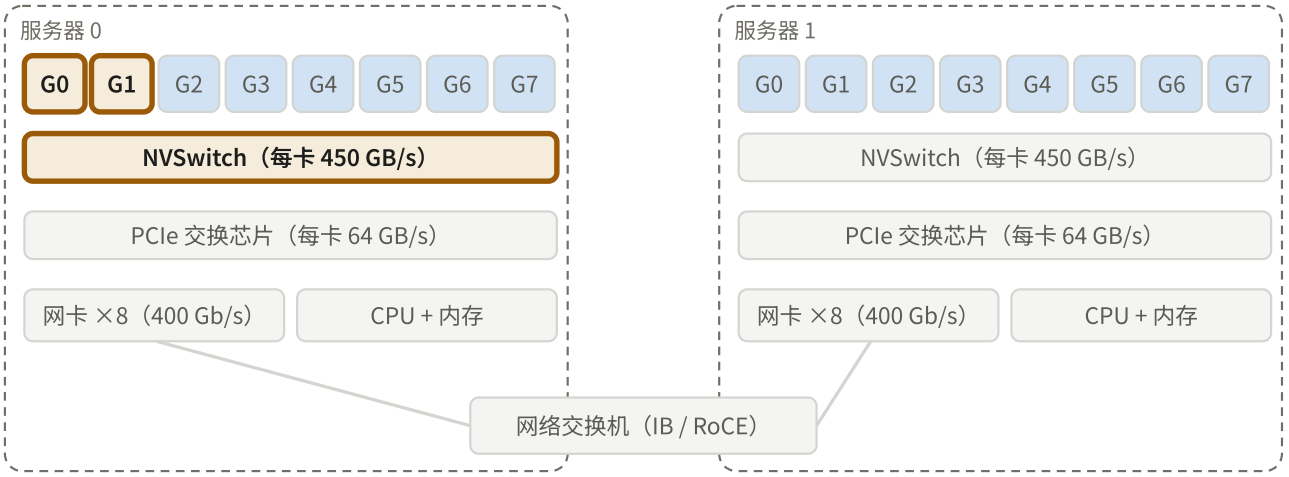
下面的演示画出了两台 8 卡服务器的连接。选一条路径、调消息大小，看瓶颈和时间。

数据从一张卡到另一张卡走哪条路

⇆ 对照

同机两卡（NVLink） | 同机两卡（只有 PCIe） | 跨机两卡（网络） | GPU ↔ CPU 内存

消息 **16 MB**



路径：NVLink → NVSwitch → NVLink（450 GB/s）。瓶颈 450 GB/s。
传 16 MB：α 3 μs + 16 MB ÷ 450 GB/s = **38.6 μs**（延迟占 8%）。
同一台机器里，8 张卡都接在 NVSwitch 上，任意两张卡之间都是满带宽。张量并行的 All-Reduce 就走这里（第 105 讲）。

图 64-1　两台 8 卡服务器的互联（示意）。机内 8 张卡经 NVSwitch 全连，每卡 450 GB/s；每张卡经 PCIe 接一张 400 Gb/s 网卡，机间经交换机相连，每卡 50 GB/s。一条路径的速度由最慢的一段决定。

要点

机内：GPU 之间用 NVLink（经 NVSwitch 全连），H100 每卡单向约 450 GB/s；GPU 和 CPU、网卡之间用 PCIe 5.0，约 64 GB/s。机间：用 InfiniBand 或 RoCE 网络，都基于 RDMA，H100 服务器每卡配一张 400 Gb/s（约 50 GB/s）的网卡。一条路径的带宽由最慢的一段决定，机内比机间快约 9 倍。

延迟：小消息的主要代价

除了带宽，每次传输还有固定的开销 α（第 102 讲）：硬件建立传输、数据经过交换机、通信库的软件开销……代码按示意的量级取：NVLink 约 3 μs，PCIe 约 5 μs，InfiniBand 约 6 μs，RoCE 约 8 μs。

传 64 KB 的小消息时，各种链路的时间都在 3 到 9 μs，几乎全是 α，带宽的差别看不出来；传 1 GB 时，时间完全由带宽决定：NVLink 4 约 2.2 ms，InfiniBand NDR 约 20 ms。第 102 讲会把这个“α-β 模型”用到各种集合通信上。

网络的拓扑

几百、几千台机器要连在一起，网络本身也有结构。大模型集群常用**胖树**（fat-tree）拓扑：机器接到接入交换机，接入交换机再接到上一层的交换机，每一层的总带宽保持不变，任意两台机器之间都有足够的带宽。

常见的优化是**按轨道连接**（rail-optimized）：每台机器的第 0 张网卡都接到同一组交换机，第 1 张接到另一组……这样，不同机器上“同一号”的 GPU 之间只经过一跳交换机，正好匹配“每台机器的第 i 张卡互相通信”的常见模式（比如第 101 讲的分层 All-Reduce）。

到了大模型里

推理的通信大多走 NVLink。张量并行几乎总在一台机器内部；跨机的通信主要是流水并行、PD 分离的 KV 传输、MoE 的专家并行（第 106、112、111 讲）。

网卡带宽在翻倍。新一代的服务器开始配 800 Gb/s 的网卡（约 100 GB/s），以太网阵营也在推出专门面向 AI 的规范（比如 Ultra Ethernet）。但和 NVLink 比，机间网络的带宽仍然低一个数量级。

把更多的卡放进 NVLink 域。既然机内机间差了 9 倍，一个自然的想法是让“机内”变得更大。这就是下一讲的超节点。

用代码复现

下面的代码列出常见互联的单向带宽，按 $\alpha + m/\beta$ 算出传 64 KB、16 MB、1 GB 的时间；再分析一台 8 卡 H100 服务器里几条路径的瓶颈，以及整机的机内、机间带宽之比。

```
# 常见互联的单向带宽（公开规格换算，GB/s）和一次传输的固定开销 α（示意的量级，含通信库的软件开销）
LINKS = [
    ("PCIe 4.0 x16", 32e9, 5e-6), ("PCIe 5.0 x16", 64e9, 5e-6),
    ("NVLink 3（A100，每卡合计）", 300e9, 3e-6), ("NVLink 4（H100，每卡合计）", 450e9, 3e-6), ("NVLink 5（B200，每卡合计）", 900e9, 3e-6),
    ("InfiniBand HDR 200 Gb/s", 25e9, 6e-6), ("InfiniBand NDR 400 Gb/s", 50e9, 6e-6), ("RoCE 400 GbE", 50e9, 8e-6),
]
print("链路 | 单向带宽 | 传 64 KB | 传 16 MB | 传 1 GB")
for name, bw, a in LINKS:
    t = lambda m: a + m / bw
    print(f"{name} | {bw / 1e9:4.0f} GB/s | {t(64e3) * 1e6:6.1f} μs | {t(16e6) * 1e3:6.2f} ms | {t(1e9) * 1e3:6.1f} ms")

# 一台 8 卡 H100 服务器里的几条路径：一条路径的带宽取决于其中最慢的一段
paths = {"同机两张卡（NVLink + NVSwitch）": [450e9],
         "同机两张卡（只有 PCIe，没有 NVLink）": [64e9, 64e9],
         "跨机两张卡（PCIe → 网卡 → 交换机 → 网卡 → PCIe）": [64e9, 50e9, 50e9, 64e9],
         "GPU 和 CPU 内存（PCIe）": [64e9]}
for name, segs in paths.items():
    print(f"{name}：瓶颈 {min(segs) / 1e9:.0f} GB/s，传 1 GB 约 {1e9 / min(segs) * 1e3:.0f} ms")
print(f"整台机器：NVLink 每卡 450 GB/s × 8 = {450 * 8} GB/s；网卡 8 张 × 50 GB/s = {50 * 8} GB/s，差 {450 / 50:.0f} 倍")
```

输出：

```
链路 | 单向带宽 | 传 64 KB | 传 16 MB | 传 1 GB
PCIe 4.0 x16 |   32 GB/s |    7.0 μs |   0.51 ms |   31.3 ms
PCIe 5.0 x16 |   64 GB/s |    6.0 μs |   0.26 ms |   15.6 ms
NVLink 3（A100，每卡合计） |  300 GB/s |    3.2 μs |   0.06 ms |    3.3 ms
NVLink 4（H100，每卡合计） |  450 GB/s |    3.1 μs |   0.04 ms |    2.2 ms
NVLink 5（B200，每卡合计） |  900 GB/s |    3.1 μs |   0.02 ms |    1.1 ms
InfiniBand HDR 200 Gb/s |   25 GB/s |    8.6 μs |   0.65 ms |   40.0 ms
InfiniBand NDR 400 Gb/s |   50 GB/s |    7.3 μs |   0.33 ms |   20.0 ms
RoCE 400 GbE |   50 GB/s |    9.3 μs |   0.33 ms |   20.0 ms
同机两张卡（NVLink + NVSwitch）：瓶颈 450 GB/s，传 1 GB 约 2 ms
同机两张卡（只有 PCIe，没有 NVLink）：瓶颈 64 GB/s，传 1 GB 约 16 ms
跨机两张卡（PCIe → 网卡 → 交换机 → 网卡 → PCIe）：瓶颈 50 GB/s，传 1 GB 约 20 ms
GPU 和 CPU 内存（PCIe）：瓶颈 64 GB/s，传 1 GB 约 16 ms
整台机器：NVLink 每卡 450 GB/s × 8 = 3600 GB/s；网卡 8 张 × 50 GB/s = 400 GB/s，差 9 倍
```

前八行是各种链路：小消息时差别不大，大消息时 NVLink 比网络快一个数量级。接着四行是几条路径的瓶颈。最后一行：整机 NVLink 3600 GB/s，网卡 400 GB/s，差 9 倍。

容易混淆的地方

1. **NVLink 的标称带宽是双向合计**。“H100 NVLink 900 GB/s”是一张卡所有 NVLink 链路两个方向加起来的；估算一次传输的时间，要用单向的约 450 GB/s。
2. **比特和字节**。网卡速率 400 Gb/s 是比特，约 50 GB/s；PCIe 和 NVLink 的规格一般直接给字节。
3. **InfiniBand 和 RoCE 是同一类技术**。两者都提供 RDMA，主要区别在底层网络（专用的 IB 还是以太网）和生态；对上层的推理引擎来说，通常只是换一个通信库的配置。

回顾

GPU 之间、GPU 和外部之间有几种互联：PCIe 是通用总线，PCIe 5.0 x16 每个方向约 64 GB/s，连接 GPU 和 CPU、网卡；NVLink 是 GPU 之间的专用高速连接，H100 每卡单向约 450 GB/s，一台机器的 8 张卡经 NVSwitch 全连，组成 NVLink 域；机器之间用 InfiniBand 或 RoCE 网络，都基于 RDMA，H100 服务器每卡一张 400 Gb/s（约 50 GB/s）的网卡。一条路径的带宽由最慢的一段决定，机内比机间快约 9 倍；小消息的时间主要是固定开销 α。

机内机间 9 倍的差距，让很多本该放在高速链路上的通信被迫走慢路。能不能把几十、上百张卡都放进同一个 NVLink 域？下一讲：**scale-up、scale-out 与超节点**。

第65讲 超节点

重点

64 互联

上一讲算出，一台 8 卡的 H100 服务器里，卡与卡之间每卡有约 450 GB/s 的 NVLink；出了这台机器，每卡只剩 50 GB/s 的网络。差了 9 倍。

这个落差把集群分成了两层，业界分别叫作 **scale-up** 和 **scale-out**。这一讲讲清楚这两个概念，以及为什么近几年各家都在努力把 scale-up 的范围做大，做成所谓的**超节点**。

两种扩展

scale-up（纵向扩展）：在一个紧密耦合的范围里加更多的卡，卡与卡之间用 NVLink 这类专用的高速互联直接相连。特点是：

- 带宽高：H100 每卡单向约 450 GB/s，B200 约 900 GB/s。
- 延迟低，而且支持**内存语义**：一张卡可以像访问自己的显存一样，直接读写另一张卡的显存（load/store），不需要打包成消息再发送。
- 规模有限：传统上一台服务器里 8 张卡。

scale-out（横向扩展）：把很多台机器用网络连起来（InfiniBand、RoCE，第 64 讲）。特点是：

- 规模几乎不限：几千、几万张卡都可以。
- 带宽低一个数量级（每卡约 50 GB/s），延迟也更高。
- 通信是**消息语义**：数据打包成消息，经网卡、交换机发出去（RDMA 让这个过程很高效，但仍然是另一种方式）。

一个大模型集群两者都有：每台机器内部是一个 scale-up 域，机器之间是 scale-out 网络。

为什么 scale-up 的范围有限

既然 NVLink 这么好，为什么不把所有卡都用它连起来？主要有几个限制：

- **距离**。高速的铜缆信号只能传一两米。8 张卡放在一个机箱里没问题，放满一整个机柜就需要专门设计的背板和线缆，再远就得用光纤，成本和功耗都上去了。
- **交换芯片的端口数**。每张卡要连到交换芯片上，交换芯片的端口数（以及每个端口的带宽）限制了一个域里能放多少张卡。
- **功耗和散热**。几十张高功耗的 GPU 挤在一个机柜里，要几十到上百千瓦的供电，需要液冷。

超节点：把 scale-up 域做大

超节点就是把 scale-up 域从 8 张卡扩大到几十、几百张。公开的例子：

- NVIDIA GB200 NVL72：一个机柜里 72 张 Blackwell GPU，通过机柜内的 NVLink 交换托盘和铜缆背板全部互连，每张卡双向 1.8 TB/s（单向约 900 GB/s），任意两张卡之间都是满带宽。
- **华为 CloudMatrix 384**：384 张昇腾 910C 通过统一的高速互联网络（用了大量光模块）组成一个超节点。

它们的思路一样：让一大批卡都处在高带宽、低延迟、可直接读写对方内存的范围里。

同一次通信，两种组织方式

用第 102 讲的 α-β 模型比较一下。同样 64 张卡：

- **A：8 台 8 卡服务器**。机内 NVLink 每卡 450 GB/s，机间每卡 50 GB/s。
- **B：64 张卡在一个 NVLink 域里**，每卡 900 GB/s。

All-Reduce 1 GB：A 可以分层做（第 101 讲）：先在机内 Reduce-Scatter，把数据缩成 1/8；机间网络只对这 1/8 做 All-Reduce；最后机内 All-Gather。约 **8.39 ms**（机内 3.93 + 机间 4.46）。B 在一个域里直接做，约 **2.26 ms**。快了 3.7 倍。

All-to-All 每卡 1 GB：每张卡要给其余 63 张卡各发一份不同的数据。在 A 里，其中 56 张卡在别的机器上，56/63 的数据都要挤过 50 GB/s 的网卡，约 **17.79 ms**；B 约 **1.12 ms**，快了 16 倍。

差别来自通信的模式：All-Reduce 可以先在机内“缩小”数据，机间只传一部分；All-to-All 的每份数据都不一样，没法缩小，只能全部走慢路。

还可以看**对分带宽**：把 64 张卡切成两半，两半之间最多能传多快。A 是 32 张网卡 × 50 GB/s = **1.6 TB/s**；B 是 32 张卡 × 900 GB/s = **28.8 TB/s**，差了 18 倍。

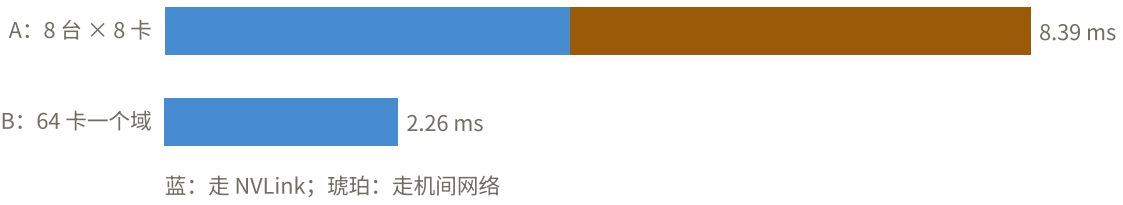
下面的演示可以切换集合通信、调数据量。

scale-out 与 scale-up：同一次通信

⇆ 对照

All-Reduce　All-to-All

每卡数据 **1 GB**



蓝：走 NVLink；琥珀：走机间网络

All-Reduce，每卡 1 GB：A 8.39 ms（机内 NVLink 3.93 ms + 机间网络 4.46 ms），B 2.26 ms，B 快 3.7 倍。
All-Reduce 可以分层做：先在机内把数据缩成 1/8，机间网络只传 1/8，所以 A 的劣势不算太大。

图 65-1　同样 64 张卡、同一次通信：8 台 8 卡服务器（A）和一个 64 卡的 NVLink 域（B）。All-Reduce 可以分层，A 的劣势有限；All-to-All 的大部分数据要走机间网络，B 快一个数量级以上。

要点

scale-up 用 NVLink 一类的专用互联把卡直接相连，带宽高、延迟低、能直接读写对方显存，但受距离、交换芯片和功耗限制，传统上只有 8 张卡；scale-out 用网络连接任意多台机器，带宽低一个数量级。超节点把 scale-up 域扩大到几十上百张卡。能分层的通信（All-Reduce）受益有限，不能分层的（All-to-All）受益最大。

对推理意味着什么

第四章会反复用到这一讲的结论：

- 张量并行每层都要同步，必须放在 scale-up 域内（第 105、109 讲）。域大了，TP 可以超过 8，更大的稠密模型也能只用 TP（第 115 讲）。
- MoE 的专家并行靠 All-to-All，是 scale-out 网络最怕的负载；超节点让一整组专家并行都走高速链路（第 111、115 讲）。
- PD 分离的 KV 传输、KV Cache 的池化，在一个大的高带宽域里都更容易（第 112、113 讲）。

超节点也有代价：成本、功耗、故障的影响范围、软件的适配。第 115 讲会从推理负载的角度具体比较。

到了大模型里

两张网并存。即使在超节点里，scale-out 网络也不会消失：超节点之间、超节点和存储之间，仍然要靠以太网或 InfiniBand。设计集群时要同时规划两层：scale-up 域多大、scale-out 网络多宽。

scale-up 的技术在多元化。除了 NVLink，业界也在推动开放的 scale-up 互联标准（比如 UALink），以及用以太网做 scale-up 的方案。它们的目标一样：让更多加速器在一个高带宽、低延迟的域里协作。

本站只用公开资料。各家超节点的具体参数（拓扑、带宽、延迟）以官方发布的资料为准，本站的数字只用于说明量级和比较思路。

用代码复现

下面的代码按 α-β 模型，比较 64 张卡在“8 台 × 8 卡”和“一个 64 卡域”两种组织方式下，All-Reduce（16 MB、1 GB）和 All-to-All（每卡 16 MB、1 GB）的耗时，以及两者的对分带宽。

```
# 同样 64 张卡，两种组织方式（带宽取公开规格的量级，α 是示意值）：
#   A：8 台 8 卡服务器。机内 NVLink 每卡单向 450 GB/s；机间每卡一张网卡 50 GB/s（按轨道连接，8 条轨道并行）
#   B：64 张卡都在一个 NVLink 域里（超节点），每卡单向 900 GB/s
A_NV, A_NET = 3e-6, 6e-6

def allreduce_A(m):                                   # 分层：机内 Reduce-Scatter → 机间对 1/8 做 Ring All-Reduce → 机内 All-Gather
    intra = 2 * (7 * A_NV + 7 / 8 * m / 450e9)
    inter = 2 * 7 * (A_NET + (m / 8) / 8 / 50e9)
    return intra + inter, intra, inter

def allreduce_B(m):                                   # 域内：Ring 和 Tree（第 101 讲）取快的
    return min(2 * 63 * (A_NV + m / 64 / 900e9), 2 * 6 * A_NV + 2 * m / 900e9)

def alltoall_A(m):                                    # 每张卡给其余 63 张各发 m/63；其中 56 张在别的机器上，走网卡
    return max(7 / 63 * m / 450e9, 56 / 63 * m / 50e9) + 2 * A_NET

def alltoall_B(m):
    return 63 / 63 * m / 900e9 + 2 * A_NV

for m in [16e6, 1e9]:
    t, intra, inter = allreduce_A(m)
    print(f"All-Reduce {m / 1e6:5.0f} MB：A（8 台 × 8 卡）{t * 1e3:6.2f} ms（机内 {intra * 1e3:.2f} + 机间 {inter * 1e3:.2f}）；B（64 卡一个域）{allreduce_B(m) * 1e3:6.2f} ms")
for m in [16e6, 1e9]:
    print(f"All-to-All 每卡发 {m / 1e6:5.0f} MB：A {alltoall_A(m) * 1e3:6.2f} ms；B {alltoall_B(m) * 1e3:6.2f} ms，快 {alltoall_A(m) / alltoall_B(m):.0f} 倍")

# 整体带宽：把 64 张卡切成两半，两半之间最多能传多快（对分带宽）
print(f"对分带宽：A 是两半之间 32 张网卡 × 50 GB/s = {32 * 50 / 1000:.1f} TB/s；B 是 32 张卡 × 900 GB/s = {32 * 900 / 1000:.1f} TB/s")
```

输出：

```
All-Reduce    16 MB: A (8 台 × 8 卡)   0.26 ms (机内 0.10 + 机间 0.15) ; B (64 卡一个域)   0.07 ms
All-Reduce  1000 MB: A (8 台 × 8 卡)   8.39 ms (机内 3.93 + 机间 4.46) ; B (64 卡一个域)   2.26 ms
All-to-All 每卡发    16 MB: A   0.30 ms; B   0.02 ms, 快 12 倍
All-to-All 每卡发  1000 MB: A  17.79 ms; B   1.12 ms, 快 16 倍
对分带宽: A 是两半之间 32 张网卡 × 50 GB/s = 1.6 TB/s; B 是 32 张卡 × 900 GB/s = 28.8 TB/s
```

前两行是 All-Reduce：A 的机内、机间两部分加起来，大消息时比 B 慢 3.7 倍。中间两行是 All-to-All：B 快 12 到 16 倍。最后一行是对分带宽：1.6 对 28.8 TB/s。

容易混淆的地方

1. **超节点不是更快的单卡**。它改变的是卡与卡之间的连接；单卡就能做完的工作，不会因为超节点变快。
2. **“一个机柜”不等于“一个 scale-up 域”**。普通的机柜里可以放好几台 8 卡服务器，它们之间仍然走网络；只有用 NVLink 一类的互联连在一起，才是一个 scale-up 域。
3. **scale-up 和 scale-out 的界线在移动**。以前 scale-up 就是一台服务器；现在可以是一个机柜，未来可能是多个机柜。判断时看的是连接方式和带宽，而不是物理边界。

回顾

集群里有两层连接：scale-up 用 NVLink 一类的专用互联，带宽高（H100 每卡单向 450 GB/s）、延迟低、能直接读写对方显存，但传统上只有 8 张卡；scale-out 用网络，规模不限，每卡约 50 GB/s。超节点把 scale-up 域扩大到几十上百张卡，比如 GB200 NVL72（72 张卡、每卡单向约 900 GB/s）和 CloudMatrix 384。同样 64 张卡，All-Reduce 1 GB 从 8.39 ms 降到 2.26 ms，All-to-All 每卡 1 GB 从 17.79 ms 降到 1.12 ms，对分带宽从 1.6 TB/s 到 28.8 TB/s。能分层的通信受益有限，All-to-All 受益最大。

我们已经有了一整套估算的工具：算力、带宽、Roofline、互联。可估算终究是估算，实际跑起来时间到底花在哪里？怎么找到真正的瓶颈？下一讲：**性能分析与 Profiling**。

第66讲 性能分析

62 Roofline 模型　64 互联

前面几讲给了我们一套估算的工具：算力、带宽、Roofline、互联。它们能告诉我们“理论上该多快”。可程序真正跑起来，常常比估算慢，而且慢的原因五花八门：某个 kernel 写得不好，CPU 来不及发射任务，通信卡住了，数据在 CPU 和 GPU 之间来回拷贝……凭猜是猜不准的。

性能分析（profiling）就是把程序实际运行的过程记录下来，看时间到底花在了哪里。这一讲介绍常用的工具，以及读一张 Profiler 时间线的方法。

常用的工具

- **PyTorch Profiler**：在 Python 代码里加几行就能用，记录每个算子在 CPU 上的耗时和对应的 GPU kernel，可以导出成时间线在浏览器里看。适合先粗看一遍。
- **NVIDIA Nsight Systems**：系统级的时间线，同时记录 CPU 线程、CUDA API 调用、GPU 上每个 kernel、内存拷贝、NCCL 通信，看“谁在等谁”最清楚。
- **NVIDIA Nsight Compute**：针对**单个 kernel** 的深入分析：它用了多少显存带宽、多少算力、Tensor Core 的利用率、缓存命中率、为什么停顿。适合确定某个 kernel 为什么慢。

推理引擎一般也有自己的 profiling 开关（比如 vLLM 可以在运行时启停 PyTorch Profiler），方便在真实负载下采集。

一张时间线长什么样

Nsight Systems 或 PyTorch Profiler 的时间线，横轴是时间，纵轴是一行行的“轨道”。最重要的是两类：

- **CPU 线程**：Python 代码、框架的调度、CUDA API 调用（比如每一次 kernel 发射）。
- **GPU 流**：GPU 上实际执行的 kernel，一个接一个的小方块。

读时间线，就是看这两行之间的关系：GPU 一直有活干吗？哪些 kernel 占了大头？它们离硬件的上限还有多远？

下面是一份**模拟的**时间线（数字由 Roofline 估算生成，用来练习读图，不是真实的测量）：Llama-3-8B，4 张 H100 做张量并行，decode 一步，批 8、上下文 2048。每层 12 个 kernel：归一化、QKV 投影、RoPE、注意力、输出投影、All-Reduce、残差加归一化、gate/up 投影、SiLU 门控、down 投影、All-Reduce、残差相加。

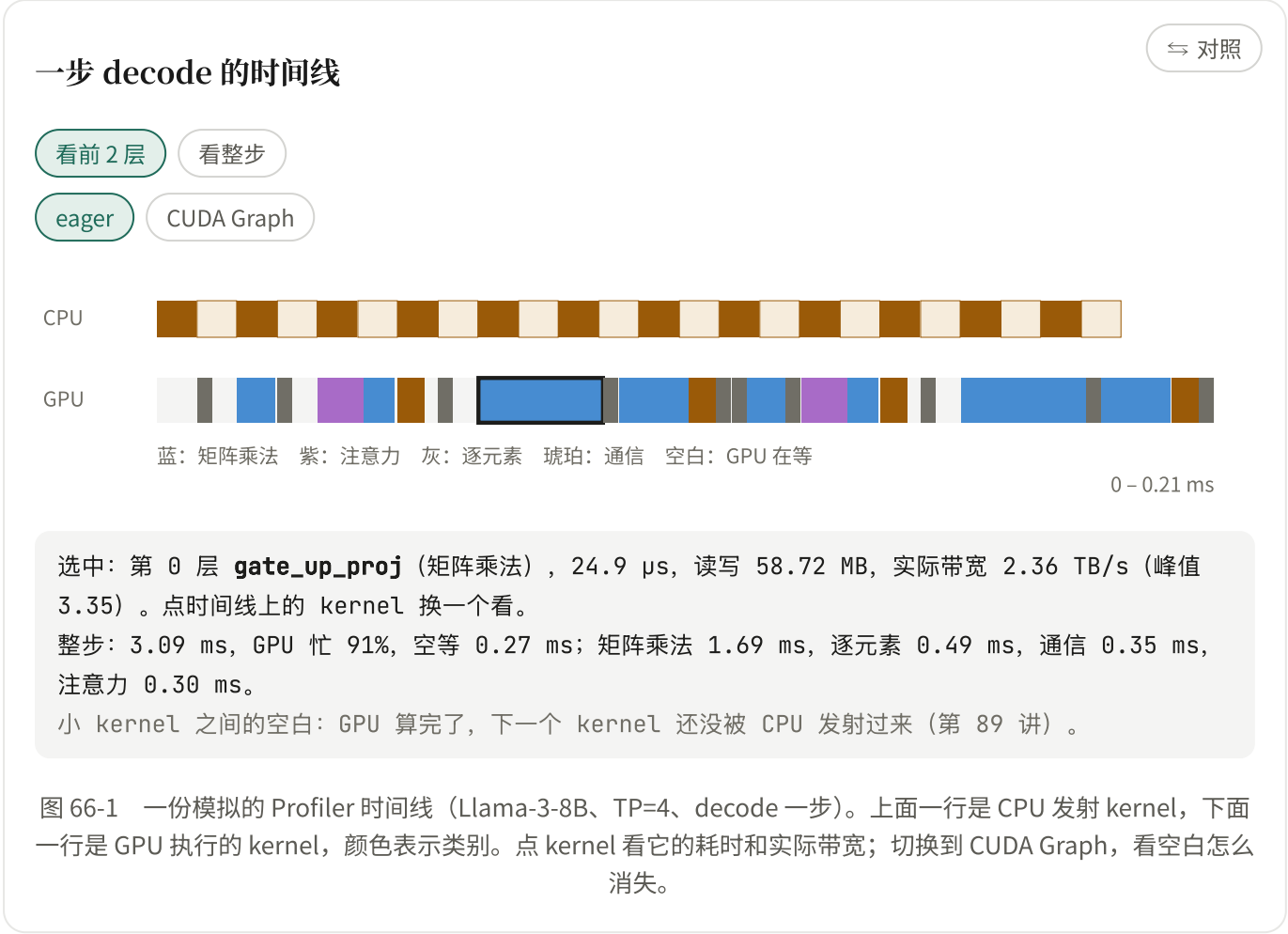


图 66-1　一份模拟的 Profiler 时间线（Llama-3-8B、TP=4、decode 一步）。上面一行是 CPU 发射 kernel，下面一行是 GPU 执行的 kernel，颜色表示类别。点 kernel 看它的耗时和实际带宽；切换到 CUDA Graph，看空白怎么消失。

读图的四步

第一步：GPU 有没有空等。先看 GPU 那一行有没有空白。这份时间线里，一步 3.09 ms，GPU 忙了 91%，有约 0.27 ms 是空的。放大看，空白出现在一串小 kernel 之间：GPU 算完了一个两三微秒的逐元素运算，下一个 kernel 还没被 CPU 发射过来。这是第 89 讲的 CPU 发射开销，解决办法是 CUDA Graph：切换过去，空白几乎消失，一步变成 2.84 ms。

如果空白很大，往往先解决它，因为 GPU 再快也没用。常见的原因还有：CPU 上的调度逻辑太慢、等待 CPU 和 GPU 之间的数据拷贝、等待其他卡的通信。

第二步：按类别分账。把 GPU 上的时间按类别加起来：矩阵乘法 1.69 ms（55%），逐元素运算 0.49 ms，通信 0.35 ms，注意力 0.30 ms。

逐元素运算有 0.49 ms，占了 16%，这有点多：它们的计算量微不足道，时间主要是每个 kernel 的固定开销和读写显存。一层有 5 个这样的小 kernel，32 层就是 160 个。这提示我们该做**算子融合**（第 83 讲）。

第三步：找最耗时的 kernel。排在前面的是 gate_up 投影（0.80 ms）、down 投影（0.45 ms）、All-Reduce（0.35 ms）、注意力（0.30 ms）。优化要从这些大头开始；把一个只占 1% 的 kernel 优化到零，也只能快 1%。

第四步：和屋顶比。最大的 gate_up 投影，一次 24.9 μs，读 58.7 MB 权重，实际带宽 **2.36 TB/s**。它是 decode 的矩阵乘法，在 Roofline 的带宽区（第 62 讲），屋顶是 3.35 TB/s。它只用到了 70%，还有提升空间：可能是 kernel 的选择不对、批太小导致并行度不够，或者这台机器的实际带宽本来就达不到标称（第 67 讲会实测）。

要点

读 Profiler 时间线分四步：先看 GPU 有没有空等（空白多半来自 CPU 发射或等待通信）；再按类别分账，看哪类运算占得多；然后找出最耗时的几个 kernel；最后把它们的实际带宽或算力和 Roofline 的屋顶比，差多少就是还能优化多少。优化从大头开始，改完再测。

根据发现决定怎么改

把四步的发现整理成一张单子：

发现	可能的改法	预期收益
GPU 空等 0.27 ms	CUDA Graph（第 89 讲）	约 0.25 ms
逐元素 kernel 多、共 0.49 ms	融合进相邻的 kernel（第 83 讲）	约 0.3 ms
矩阵乘法只用到 70% 带宽	换更好的 kernel、调整切分	最多约 0.5 ms
All-Reduce 0.35 ms	小消息专用实现、和计算重叠（第 102、105 讲）	视情况

每改一项，**重新测一次**。优化之间会相互影响：融合了逐元素运算，kernel 数少了，CPU 发射的压力也小了；CUDA Graph 的收益可能就没那么大了。

测量时的几个坑

异步执行。GPU 的 kernel 是异步发射的（第 89 讲），在 Python 里用计时器量一个函数，量到的可能只是发射的时间。要么在计时前后调用 `torch.cuda.synchronize()`，要么用 CUDA 事件、用 Profiler。

预热。第一次运行时要编译 kernel、分配显存、自动调优（第 88 讲的 autotune），会特别慢。先跑几次再测。

Profiler 本身有开销。记录得越详细，程序跑得越慢，时间线里的空白可能被放大。重要的结论最好用不开 Profiler 的端到端计时再确认一次。

测真实的负载。批大小、序列长度不同，瓶颈完全不同：同一个模型，decode 卡在带宽上，prefill 卡在算力上；小批时 CPU 发射是问题，大批时就不是。要在接近线上的条件下测。

到了大模型里

分布式的时间线。多卡、多机时，每张卡都有自己的时间线。一张卡在等 All-Reduce，往往是因为另一张卡还没算完（负载不均）或者网络拥塞。要把多张卡的时间线放在一起看，找出最慢的那张。

指标监控和 Profiling 互补。线上服务通常有监控指标（TTFT、TPOT、吞吐、KV Cache 使用率、抢占次数，第 82 讲），它们告诉你“哪里不对”；Profiling 告诉你“为什么”。

用代码复现

下面的代码按 Roofline 生成这份模拟时间线（每个 kernel 的耗时、CPU 的发射），然后做四步分析：GPU 空闲、按类别分账、最耗时的 kernel、最大矩阵乘法的实际带宽；最后对比 CUDA Graph。

```
# 一份“模拟”的 Profiler 时间线：Llama-3-8B，4 张 H100 张量并行（追求低延迟），decode 一步，批 8、上下文 2048。
# 每个 kernel 的耗时按 Roofline 估算（矩阵乘法按带宽的 80%），CPU 逐个发射、每个约 8 µs（eager 模式，第 89 讲）
BW, EFF, LAUNCH, L, B, CTX = 3.35e12, 0.8, 8e-6, 32, 8, 2048
d, dff, kvh = 4096, 14336, 8
def gemm(n_in, n_out): return ("矩阵乘法", 2 * n_in * n_out / 4)              # 每张卡读 1/4 的权重（TP=4），BF16
LAYER = [("rmsnorm", ("逐元素", 4 * B * d)), ("qkv_proj", gemm(d, d + 2 * kvh * 128)), ("rope", ("逐元素", 4 * B * d)),
         ("attention_decode", ("注意力", B * CTX * kvh // 4 * 128 * 2 * 2)), ("o_proj", gemm(d, d)), ("all_reduce", ("通信", B * d * 2)),
         ("add_rmsnorm", ("逐元素", 6 * B * d)), ("gate_up_proj", gemm(d, 2 * dff)), ("silu_mul", ("逐元素", 6 * B * dff // 4)),
         ("down_proj", gemm(dff, d)), ("all_reduce", ("通信", B * d * 2)), ("add", ("逐元素", 6 * B * d))]

def trace(graph):
    cpu, gpu, out = (20e-6, 0.0, []) if graph else (0.0, 0.0, [])
    for layer in range(L):
        for name, (cat, nbytes) in LAYER:
            dur = 3e-6 + nbytes / (BW * EFF) if cat != "通信" else 5e-6 + 3 * nbytes / 450e9   # 通信：一步到位的 All-Reduce
            if not graph: cpu += LAUNCH
            start = max(gpu, cpu); gpu = start + dur
            out.append((layer, name, cat, start, dur, nbytes))
    return out

def summary(tr, title):
    T = tr[-1][3] + tr[-1][4]; busy = sum(k[4] for k in tr)
    print(f"{title}: 一步 {T * 1e3:.2f} ms, {len(tr)} 个 kernel, GPU 忙 {busy / T:.0%}, 空闲 {(T - busy) * 1e3:.2f} ms")
    by = {}
    for k in tr: by[k[2]] = by.get(k[2], 0) + k[4]
    print("   按类别: " + ", ".join(f"{c} {t * 1e3:.2f} ms" for c, t in sorted(by.items(), key=lambda x: -x[1])))
    return T

eager = trace(False); T1 = summary(eager, "eager")
top = {}
for k in eager: top[k[1]] = top.get(k[1], 0) + k[4]
print("   最耗时的 kernel: " + ", ".join(f"{n} {t * 1e3:.2f} ms" for n, t in sorted(top.items(), key=lambda x: -x[1])[:4]))
g = [k for k in eager if k[1] == "gate_up_proj"][0]
print(f"   gate_up_proj 一次 {g[4] * 1e6:.1f} µs, 读 {g[5] / 1e6:.1f} MB, 实际带宽 {g[5] / g[4] / 1e12:.2f} TB/s (峰值 3.35) ")
```

```
T2 = summary(trace(True), "CUDA Graph")
print(f"换成 CUDA Graph 后一步快了 {T1 / T2:.1f} 倍")
```

输出：

```
eager：一步 3.09 ms，384 个 kernel，GPU 忙 91%，空闲 0.27 ms
   按类别：矩阵乘法 1.69 ms，逐元素 0.49 ms，通信 0.35 ms，注意力 0.30 ms
   最耗时的 kernel：gate_up_proj 0.80 ms，down_proj 0.45 ms，all_reduce 0.35 ms，
attention_decode 0.30 ms
   gate_up_proj 一次 24.9 μs，读 58.7 MB，实际带宽 2.36 TB/s（峰值 3.35）
CUDA Graph：一步 2.84 ms，384 个 kernel，GPU 忙 99%，空闲 0.02 ms
   按类别：矩阵乘法 1.69 ms，逐元素 0.49 ms，通信 0.35 ms，注意力 0.30 ms
换成 CUDA Graph 后一步快了 1.1 倍
```

第一组是 eager：一步 3.09 ms，GPU 忙 91%，按类别和按 kernel 的排序，以及 gate_up 投影的实际带宽 2.36 TB/s。第二组是 CUDA Graph：空闲几乎为零，一步 2.84 ms。

容易混淆的地方

1. **GPU 利用率高不等于跑得好。** `nvidia-smi` 的“利用率”只说明这段时间有 kernel 在跑，即使它只用了很小一部分算力和带宽。要看实际带宽、实际算力和屋顶的距离。
2. **CPU 上的耗时不是 GPU 上的耗时。**PyTorch Profiler 的表格里，同一个算子有“CPU 时间”和“CUDA 时间”两栏，前者是发射和调度的开销，后者才是 GPU 上执行的时间。
3. **模拟不是测量。**这一讲的时间线是用模型生成的，用来练习读图的方法；真实的时间线要在自己的机器上采集，第 67 讲会讲怎么把实测值和理论值对上。

回顾

性能分析把程序实际运行的过程记录下来：PyTorch Profiler 适合粗看，Nsight Systems 看 CPU 和 GPU 之间谁在等谁，Nsight Compute 深入分析单个 kernel。读时间线分四步：看 GPU 有没有空等，按类别分账，找最耗时的 kernel，把它们和 Roofline 的屋顶比较。模拟的 8B、TP=4 decode 一步 3.09 ms：GPU 空等 0.27 ms（换 CUDA Graph 后降到 2.84 ms），逐元素运算 0.49 ms 提示要融合，最大的矩阵乘法只用到 2.36 / 3.35 TB/s。测量时要同步、预热、注意 Profiler 的开销，并在真实负载下测。

最后一个问题：Roofline 的屋顶用的是规格表上的峰值，实际的机器能达到多少？手算的速度和实测的速度差多少？下一讲用一台 DGX Spark 来对一对：**理论值对上实测值**。

第67讲 理论 vs 实测

61 存储层级　62 Roofline 模型　66 性能分析

这一篇的所有估算都基于规格表上的数字：峰值算力、显存带宽。它们是硬件在理想条件下的上限，实际的程序能达到多少？手算出来“一步 4.8 ms”，真跑起来是 6 ms 还是 15 ms？如果差得很多，是估算错了，还是程序有问题？

这一讲讲怎么把理论值和实测值对上：先用两个简单的测试“标定”出机器实际的屋顶，再用它预测和核对真实模型的速度。我们以一台桌面级的 NVIDIA DGX Spark 为例，和数据中心的 H100 对比。

> **注意**
> 本讲的实测部分需要在 DGX Spark 上运行下面的测试脚本获得，目前给出的都是**理论预测**，标为“待测”。作者测得结果后，把数字填入代码里的 `measured`，本讲的表格和结论再据此更新。DGX Spark 的 BF16 算力是由公开规格推算的，也需要核实。

两台机器的屋顶

DGX Spark 用的是 GB10 芯片（Grace CPU + Blackwell GPU），CPU 和 GPU 共享 128 GB 的 LPDDR5x 内存，带宽约 273 GB/s。官方公布的 AI 算力是 FP4 约 1 PFLOPS（含稀疏）；按“稀疏是稠密的两倍、精度每升一级算力减半”推算，BF16 稠密约 125 TFLOPS（待核实）。

H100：BF16 稠密 989 TFLOPS，HBM 带宽 3.35 TB/s。

两者的带宽差约 12 倍，算力差约 8 倍。屋脊点：DGX Spark 约 458，H100 约 295。DGX Spark 的屋脊点更高，意味着更多的运算会落在带宽区；而它的优势是 128 GB 的大内存，能放下 H100 一张卡放不下的模型（比如 4 bit 的 70B）。

下面是两台机器的 Roofline，点是各个测试的理论位置。

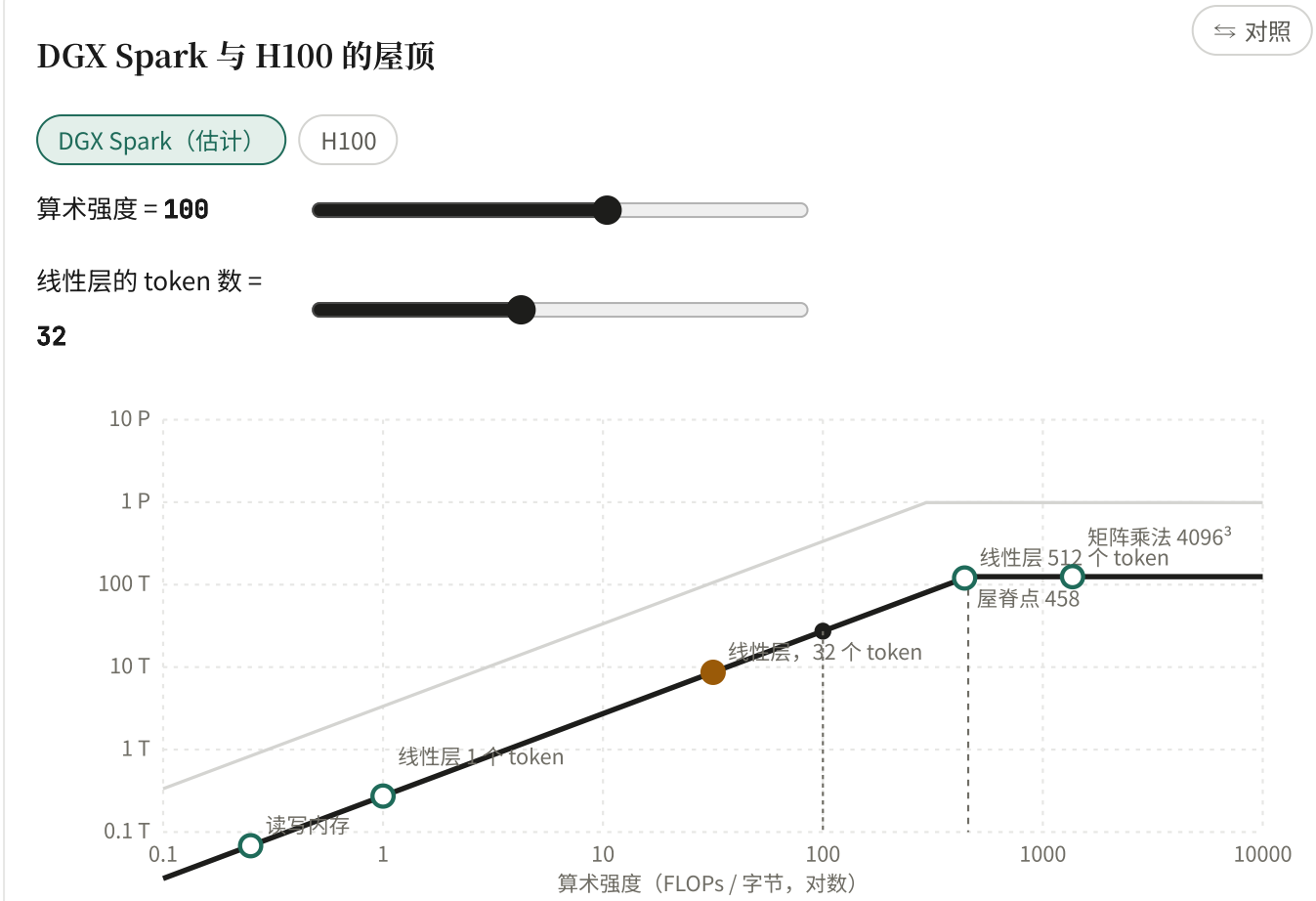


图 67-1　DGX Spark（BF16 算力为推算值）与 H100 的 Roofline。点是本讲测试的理论位置；作者实测后，可以把实测点画上去，看离屋顶多远。

第一步：测出实际的带宽

最基本的标定是**读写一大块内存**：比如把一个 4 GB 的张量拷贝或求和。这个运算几乎不做计算，时间完全由带宽决定。理论时间：DGX Spark 约 14.7 ms，H100 约 1.2 ms。

实测的带宽通常能达到标称的 70% 到 90%。这个比例很重要：后面所有带宽区的估算（decode、逐元素运算）都应该乘上它。

第二步：测出实际的算力

第二个标定是**大矩阵乘法**：4096 × 4096 × 4096 的 BF16 矩阵乘法，算术强度约 1365，远在屋脊点右边，时间由算力决定。理论时间：DGX Spark 约 1.10 ms，H100 约 0.14 ms。

调用 PyTorch 的 `torch.matmul`（背后是 cuBLAS），大矩阵乘法通常能达到峰值的 60% 到 80%。这个比例就是算力区的修正系数。

两个系数测出来以后，就有了这台机器**实际的屋顶**：带宽屋顶 = 实测带宽，算力屋顶 = 实测算力。

第三步：扫一遍算术强度

再测线性层在不同 token 数下的时间（第 62 讲：线性层的强度约等于 token 数）。理论上：

- 1 个、8 个、64 个 token：DGX Spark 都约 0.43 ms，因为都在带宽区，时间由读权重决定，多算几个 token 几乎不花时间。
- 512 个 token：约 0.50 ms，开始接近屋脊点。

把实测点画到 Roofline 上，看它们是否沿着屋顶走：带宽区的点应该在斜线下方不远，算力区的点在水平线下方不远。如果某个点离屋顶特别远，就是值得用 Profiler（第 66 讲）深挖的地方。

第四步：核对真实模型

最后用标定好的屋顶预测真实模型的速度，再实际跑一遍对比。decode 批 1 时，每个 token 要把权重读一遍，速度上限 = 带宽 ÷ 权重字节数：

- Llama-3-8B，BF16（16.1 GB）：DGX Spark 约 **17 tok/s**，H100 约 209 tok/s。
- FP8（8.0 GB）：约 34 tok/s。
- 4 bit（每组 128 个一个 scale，约 4.5 GB，第 90 讲）：约 **60 tok/s**。
- Llama-3.1-70B，4 bit（39.7 GB）：约 **7 tok/s**。H100 一张卡也放得下 4 bit 的 70B，约 84 tok/s。

实测的速度会低于这些上限：除了读权重，还要读 KV Cache、做注意力、发射 kernel（第 89 讲）、采样。如果实测是上限的 70% 到 80%，说明推理框架已经做得不错；如果只有 30%，就该看时间线找原因了。

要点

把理论值对上实测值分三步：用“读写大块内存”标定实际带宽，用“大矩阵乘法”标定实际算力，得到这台机器真实的屋顶；扫一遍不同算术强度的运算，看实测点是否沿屋顶走；最后用屋顶预测真实模型（decode 上限 ≈ 带宽 ÷ 权重字节数），和实测对比。偏差大的地方，就是该用 Profiler 深挖的地方。

测试脚本

下面的脚本在 GPU 上测上面几项，需要 PyTorch 和一张 NVIDIA GPU（在 DGX Spark 上运行即可）。它会做预热、用 CUDA 事件计时、取多次的中位数。

```
import torch

def bench(fn, warmup=5, iters=20):
    for _ in range(warmup): fn()
    torch.cuda.synchronize()
    times = []
    for _ in range(iters):
        s, e = torch.cuda.Event(enable_timing=True),
torch.cuda.Event(enable_timing=True)
        s.record(); fn(); e.record(); torch.cuda.synchronize()
        times.append(s.elapsed_time(e) / 1e3)               # 秒
    return sorted(times)[len(times) // 2]

x = torch.empty(2 * 1024**3, dtype=torch.bfloat16, device="cuda")   # 4 GB
print("读 4 GB:", bench(lambda: x.sum()))

a = torch.randn(4096, 4096, dtype=torch.bfloat16, device="cuda")
b = torch.randn(4096, 4096, dtype=torch.bfloat16, device="cuda")
print("矩阵乘法 4096³:", bench(lambda: a @ b))

w = torch.randn(4096, 14336, dtype=torch.bfloat16, device="cuda")
for n in [1, 8, 64, 512]:
    t = torch.randn(n, 4096, dtype=torch.bfloat16, device="cuda")
    print(f"线性层，{n} 个 token:", bench(lambda: t @ w))
```

端到端的 decode 速度，可以用 vLLM、SGLang 或 llama.cpp 跑一个批 1 的生成任务，记录稳定后的每秒 token 数。

用代码复现

下面的代码给出两台机器的屋顶和各项测试的理论时间，以及几种模型、精度下 decode 的速度上限。作者实测后，把时间填进 `measured`，表格最后一列就会显示实测值和它占理论的比例。

```
# 理论预测：DGX Spark 与 H100。DGX Spark 的 BF16 稠密算力是由公开的“FP4 约 1 PFLOPS（稀疏）”按每降一级精度翻倍推算的，待核实
GPUS = {"DGX Spark": (125e12, 273e9), "H100": (989e12, 3.35e12)}
measured = {}   # 作者在 DGX Spark 上运行测试脚本后，把实测时间（秒）填在这里，例如 measured["读 4 GB"] = ...

def t(F, BW, flops, bytes_): return max(flops / F, bytes_ / BW)

tests = {"读 4 GB": (0, 4e9), "矩阵乘法 4096³ (BF16) ": (2 * 4096**3, 3 * 4096**2 * 2)}
for b in [1, 8, 64, 512]:                                  # Llama-3-8B 的 FFN 上投影 4096 × 14336
    tests[f"线性层，{b} 个 token"] = (2 * b * 4096 * 14336, 2 * (4096 * 14336 + b * (4096 + 14336)))

for g, (F, BW) in GPUS.items():
    print(f"{g}：BF16 {F / 1e12:.0f} TFLOPS，带宽 {BW / 1e9:.0f} GB/s，屋脊点 {F / BW:.0f}")
print("测试 | DGX Spark 预测 | H100 预测 | 实测 (DGX Spark) ")
for name, (fl, by) in tests.items():
    p = [t(F, BW, fl, by) for F, BW in GPUS.values()]
    m = measured.get(name)
    print(f"{name} | {p[0] * 1e3:8.3f} ms | {p[1] * 1e3:7.3f} ms | {f'{m * 1e3:.3f} ms（理论的 {p[0] / m:.0%}）' if m else '待测'}")

# 端到端：decode 一个请求（批 1），每个 token 要把权重读一遍
for name, n, bytes_per in [("Llama-3-8B BF16", 8.03e9, 2), ("Llama-3-8B FP8", 8.03e9, 1), ("Llama-3-8B 4 bit", 8.03e9, 0.5625),
                           ("Llama-3.1-70B 4 bit", 70.6e9, 0.5625)]:
    w = n * bytes_per
    print(f"{name}：权重 {w / 1e9:5.1f} GB → DGX Spark 上限 {273e9 / w:5.1f} tok/s，H100 上限 {3.35e12 / w if w < 80e9 else 0:5.1f} tok/s")
```

输出：

```
DGX Spark：BF16 125 TFLOPS，带宽 273 GB/s，屋脊点 458
H100：BF16 989 TFLOPS，带宽 3350 GB/s，屋脊点 295
测试 | DGX Spark 预测 | H100 预测 | 实测（DGX Spark）
读 4 GB |   14.652 ms |   1.194 ms | 待测
矩阵乘法 4096³（BF16） |    1.100 ms |   0.139 ms | 待测
线性层，1 个 token |    0.430 ms |   0.035 ms | 待测
线性层，8 个 token |    0.431 ms |   0.035 ms | 待测
线性层，64 个 token |    0.439 ms |   0.036 ms | 待测
线性层，512 个 token |    0.499 ms |   0.061 ms | 待测
Llama-3-8B BF16：权重  16.1 GB → DGX Spark 上限  17.0 tok/s，H100 上限 208.6 tok/s
Llama-3-8B FP8：权重   8.0 GB → DGX Spark 上限  34.0 tok/s，H100 上限 417.2 tok/s
Llama-3-8B 4 bit：权重   4.5 GB → DGX Spark 上限  60.4 tok/s，H100 上限 741.7 tok/s
Llama-3.1-70B 4 bit：权重  39.7 GB → DGX Spark 上限   6.9 tok/s，H100 上限  84.4
tok/s
```

前两行是两台机器的屋顶。表格是各项测试的理论时间，“待测”一栏等实测后填入。最后四行是 decode 的速度上限。

容易混淆的地方

1. **理论值是上限，不是预测的“应该值”**。实测达到 70% 到 90% 是正常的；达到 100% 以上，多半是测量有问题（比如没有同步、数据在缓存里）。
2. **统一内存不是 HBM**。DGX Spark 的 CPU 和 GPU 共享 LPDDR5x 内存，容量大、不用在 CPU 和 GPU 之间拷贝，但带宽远低于 HBM。它适合放大模型做开发和实验，而不是高吞吐的服务。
3. **测出来的效率因运算而异**。带宽区和算力区的效率不同，不同形状的矩阵乘法效率也不同。最好针对自己关心的运算分别测。

回顾

规格表给的是理想条件下的上限，实际能达到多少要测。方法是：用读写大块内存标定实际带宽，用大矩阵乘法标定实际算力，得到机器真实的屋顶；扫一遍不同 token 数的线性层，看实测点是否沿屋顶走；再用屋顶预测真实模型的速度（decode 上限 ≈ 带宽 ÷ 权重字节数）并实测核对。DGX Spark 带宽约 273 GB/s，Llama-3-8B BF16 的 decode 上限约 17 tok/s，4 bit 约 60 tok/s，128 GB 的统一内存能放下 4 bit 的 70B（上限约 7 tok/s）。实测值待作者在 DGX Spark 上测量后填入。

第 8 篇到这里结束：我们有了估算和测量单机性能的完整工具。下一篇进入推理本身：一个请求从输入提示到输出第一个字、再到一个个字往外蹦，经历了两个性质完全不同的阶段。下一讲：Prefill 与 Decode。

AI INFRA · HANDBOOK

第 九 篇

推理基础

收录 6 讲 · 第 68–73 讲

第68讲 Prefill 与 Decode

24 自回归生成　27 因果掩码　35 参数量与计算量

从这一篇开始，我们正式进入推理优化。第一个要建立的认识是：**一次推理请求分成两个性质截然不同的阶段**。

第 24 讲说过，模型先把整段提示一次性送进去做前向，得到第一个新 token 的分布，这叫 **prefill**（预填充）；之后每生成一个 token，就把它接回去再做一次前向，这叫 **decode**（解码）。两个阶段跑的是同一个模型、同样的计算，可它们的“瓶颈”完全不同：prefill 卡在 GPU 的算力上，decode 卡在 GPU 的显存带宽上。这一讲用一个简单的估算方法把这件事算清楚。后面几乎所有的推理优化，都是针对其中某一个阶段的瓶颈设计的。

一个简单的估算方法

GPU 做一次前向，要干两件事：**计算**（矩阵乘法里的乘加）和**搬数据**（把权重、KV Cache 从显存读到计算单元，把结果写回去）。两件事可以同时进行，所以一次前向的耗时，大致取决于两者中更慢的那个：

$$\text{耗时} \approx \max(\text{计算量} / \text{算力}, \text{读写字节数} / \text{显存带宽})$$

以这门课的主力参照 H100 为例：BF16 的稠密算力约 989 TFLOPS（每秒 989 万亿次浮点运算），显存带宽约 3.35 TB/s（每秒 3.35 万亿字节）。两者一除：

$$989 \times 10^{12} \,/\, 3.35 \times 10^{12} \approx 295 \text{ FLOPs / 字节}$$

意思是：每从显存读进 1 个字节，要对它做约 295 次运算，计算单元才不会闲着等数据。一个运算“每字节做多少次运算”叫它的**算术强度**（arithmetic intensity）。算术强度大于 295，计算更慢，叫**计算受限**；小于 295，搬数据更慢，叫**带宽受限**（或访存受限）。第 62 讲会把这个模型（Roofline）讲透，这里先用它算两个阶段。

Prefill：大矩阵乘法，算力是瓶颈

Llama-3-8B 用 BF16，权重约 16 GB。一次前向，每个权重都要从显存读一遍。

prefill 一段 1000 个 token 的提示：1000 个 token 一起经过每一层，每层的矩阵乘法是 [1000, 4096] 乘 [4096, ⋯]。权重只读一遍，却被 1000 个 token 用了 1000 次。计算量约 2 × 75 亿 × 1000 ≈ 15 万亿次（第 35 讲，嵌入表不参与乘法），读写约 16 GB 多一点：

- 计算时间：约 15.44 ms
- 读写时间：约 4.85 ms

计算更慢，**prefill 卡在算力**，约 15.4 ms。它的算术强度约 927 FLOPs/字节，远高于 295。粗略地说，prefill 的算术强度约等于提示的长度：一次读进来的权重被多少个 token 共用，就做了多少倍的计算。

Decode：一个 token，也要读完全部权重

decode 一步只处理 1 个新 token。矩阵乘法变成了 [1, 4096] 乘 [4096, ⋯]：计算量只有约 160 亿次，可**权重还是要整个读一遍**，16 GB 一个字节都不能少，还要再读这个请求的 KV Cache。

- 计算时间：约 0.016 ms
- 读写时间：约 4.83 ms

搬数据的时间是计算的约 300 倍，**decode 卡在带宽**，每步约 4.83 ms。它的算术强度只有约 0.93，计算单元 99% 以上的时间都在等数据，算力只用了 0.3%。

一个请求，提示 1000 个 token、生成 200 个：prefill 15.4 ms，decode 200 步共约 967 ms，合计约 983 ms，**98% 的时间花在 decode 上**。

> **要点**
> prefill 一次处理整段提示，同一份权重被所有 token 共用，算术强度高，卡在算力；decode 每步只处理一个 token，却要把全部权重和 KV Cache 读一遍，算术强度约为 1，卡在显存带宽。

下面的演示画出了一个请求的时间线（理想情况，假设硬件跑满）。绿色是 prefill，琥珀色的细格是一步步的 decode。拖动提示长度和生成长度，看两个阶段各占多少时间、各自卡在哪里。

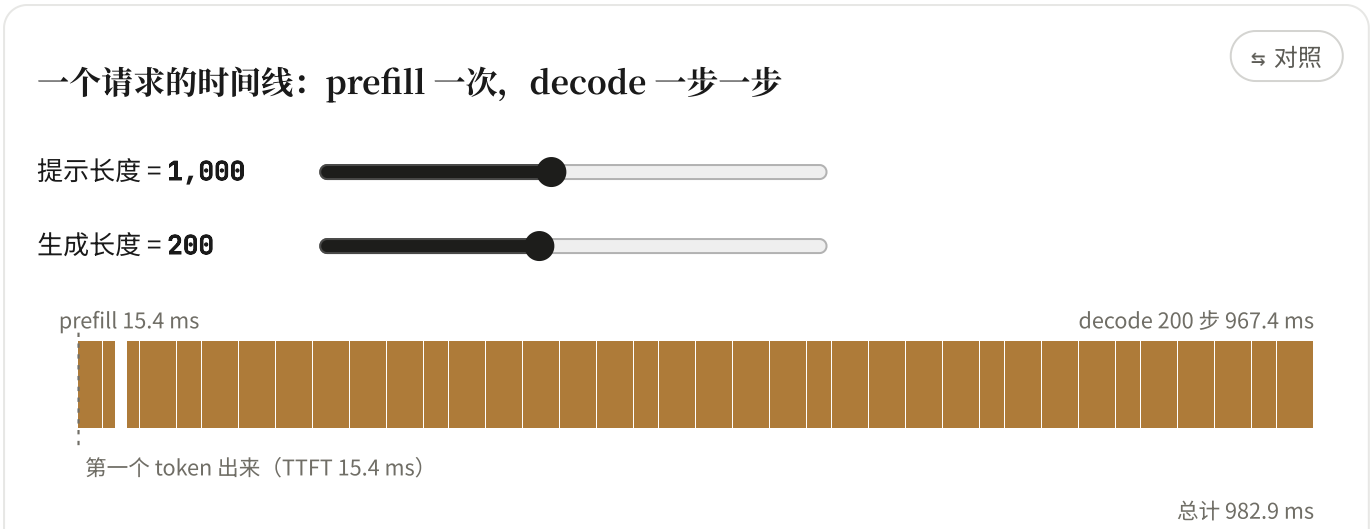


图 68-1　一个请求的时间线（Llama-3-8B，一张 H100，理想值）。默认提示 1000、生成 200：prefill 15.4 ms，之后 200 步 decode 每步约 4.8 ms，占了 98% 的时间。

几个值得试的操作：

- **把提示调短到 16 或 64**：prefill 的计算量变小，读权重的时间却不变，prefill 也变成了带宽受限，耗时停在约 4.8 ms，和一步 decode 差不多。
- **把提示调长到 6.4 万**：prefill 的时间迅速增长到秒级，第一个 token 要等很久才出来；decode 每一步也变慢了，因为要读的 KV Cache 越来越多，注意力的计算也在增长。
- **把生成调到 4000**：decode 的总时间几乎和生成长度成正比，第 49 讲的推理模型就是这种情况。

两个阶段，两种优化

瓶颈不同，优化的方向也不同：

	prefill	decode
一次处理	整段提示	1 个 token
瓶颈	算力	显存带宽
决定的指标	第一个 token 要等多久（TTFT）	之后每个 token 的间隔（TPOT）
主要优化思路	减少计算：FlashAttention、更低精度的计算、跳过重复的前缀	减少要读的字节、提高每次读取的利用率：批处理、量化、GQA/MLA、投机解码

最重要的一条是 decode 的**批处理**：既然每一步都要把权重读一遍，那就让这一遍读进来的权重同时服务很多个请求。64 个请求一起 decode，权重只读一次，算术强度就提高到约 64，单位时间里生成的 token 也接近翻 64 倍。这是推理系统提高吞吐最核心的手段，第 70 讲和第 10 篇会详细讲。

用代码复现

下面的代码实现了上面的估算方法：给定一次前向处理多少个新 token、前面有多少上下文，算出计算时间和读写时间；然后分别算 prefill 1000 个 token、decode 1 个 token，以及一个完整请求的耗时。

```python
# Llama-3-8B (BF16) 在一张 H100 上：每一步的耗时 ≈ max(计算量 / 算力, 读写字节 / 带宽)
N, N_emb, L, d = 8.03e9, 128256 * 4096, 32, 4096
W_BYTES = N * 2                          # 权重，BF16 每个 2 字节
KV_PER_TOK = 2 * 32 * 8 * 128 * 2        # 每个 token 的 KV Cache：128 KiB（第 39 讲）
FLOPS, BW = 989e12, 3.35e12              # H100 SXM：BF16 稠密算力、显存带宽

def step(new_tokens, context):
    """一次前向：处理 new_tokens 个新 token，它们前面平均有 context 个 token。返回 (计算时间, 读写时间)，单位毫秒。"""
    flops = 2 * (N - N_emb) * new_tokens + 4 * L * d * context * new_tokens
    bytes_ = W_BYTES + KV_PER_TOK * (context + new_tokens)          # 权重读一遍 + KV Cache 读写
    return flops / FLOPS * 1e3, bytes_ / BW * 1e3

print(f"算力 / 带宽 = {FLOPS / BW:.0f} FLOPs/字节：每读 1 个字节，要做 {FLOPS / BW:.0f} 次运算才能让计算单元不闲着")

# 1. prefill：一次处理整段提示（1000 个 token）
c, m = step(1000, 500)                   # 因果注意力，平均每个 token 前面有约 500 个
print(f"prefill 1000 个 token：计算 {c:.2f} ms，读写 {m:.2f} ms → 卡在{'算力' if c > m else '带宽'}，约 {max(c, m):.1f} ms")
print(f"  算术强度 ≈ {2 * (N - N_emb) * 1000 / (W_BYTES + KV_PER_TOK * 1000):.0f} FLOPs/字节")

# 2. decode：一次只处理 1 个新 token（前面有 1000 个）
c, m = step(1, 1000)
print(f"decode 1 个 token：计算 {c:.3f} ms，读写 {m:.2f} ms → 卡在{'算力' if c > m else '带宽'}，约 {max(c, m):.2f} ms")
print(f"  算术强度 ≈ {2 * (N - N_emb) / (W_BYTES + KV_PER_TOK * 1001):.2f} FLOPs/字节，算力利用率约 {c / m:.1%}")

# 3. 一个请求：提示 1000 个 token，生成 200 个
pre = max(*step(1000, 500))
dec = sum(max(*step(1, 1000 + i)) for i in range(200))
print(f"一个请求：prefill {pre:.1f} ms + decode 200 步 {dec:.0f} ms = {pre + dec:.0f} ms；decode 占 {dec / (pre + dec):.0%}")
```

输出：

```
算力 / 带宽 = 295 FLOPs/字节：每读 1 个字节，要做 295 次运算才能让计算单元不闲着
prefill 1000 个 token：计算 15.44 ms，读写 4.85 ms → 卡在算力，约 15.4 ms
  算术强度 ≈ 927 FLOPs/字节
decode 1 个 token：计算 0.016 ms，读写 4.83 ms → 卡在带宽，约 4.83 ms
  算术强度 ≈ 0.93 FLOPs/字节，算力利用率约 0.3%
一个请求：prefill 15.4 ms + decode 200 步 967 ms = 983 ms；decode 占 98%
```

第一行是 H100 的算力和带宽之比，295。接下来是 prefill：计算 15.44 ms，读写 4.85 ms，卡在算力，算术强度约 927。然后是 decode：计算 0.016 ms，读写 4.83 ms，卡在带宽，算术强度约 0.93。最后一行：一个请求约 983 ms，decode 占 98%。

到了大模型里

这些是理想值。实际的 GPU 达不到峰值算力和带宽，算子之间还有各种开销。实测时，decode 一步通常比 4.8 ms 慢一些，prefill 的算力利用率也到不了 100%。第 67 讲会把实测点和理论值放在一起比较。但理想值给出了一个下限，也告诉你瓶颈在哪：decode 想再快，就得减少要读的字节，或者让一次读取服务更多的请求，而不是换一块算力更强的卡。

你手里的 DGX Spark 更明显。它的统一内存带宽比 H100 的显存带宽低一个数量级左右（公开资料约 273 GB/s，待核实），而算力相对更充足，decode 会更加“带宽受限”。同样跑 Llama-3-8B 的 BF16 版本，单请求 decode 每步的理论下限约是 16 GB / 273 GB/s ≈ 59 ms。这也是为什么在这类设备上，量化（第 12 篇）带来的加速特别明显。

prefill 和 decode 可以分开部署。既然两个阶段的瓶颈不同，有的系统干脆让一批 GPU 专门做 prefill、另一批专门做 decode，中间把 KV Cache 传过去，各自按自己的瓶颈优化，这叫 PD 分离（第 14 篇）。

容易混淆的地方

1. **prefill 也算“生成”了一个 token**。prefill 那次前向的最后一个位置，就给出了第一个新 token 的分布。所以第一个 token 的时间主要是 prefill 的时间，之后每个 token 的时间是 decode 一步的时间。
2. **decode 慢不是因为算得多**。decode 每一步的计算量只有 prefill 的千分之一，慢是因为每一步都要把 16 GB 的权重搬一遍，搬数据的时间远大于计算的时间。
3. **“受限”会随条件变化**。很短的 prefill 会变成带宽受限，批处理很大的 decode 会变成计算受限。看的永远是算术强度和硬件的那个比值（H100 是 295）。

回顾

一次推理请求分成两个阶段。prefill 一次处理整段提示，同一份权重被所有 token 共用，算术强度约等于提示长度，Llama-3-8B 在 H100 上处理 1000 个 token 约 15.4 ms，卡在算力。decode 每步只处理一个 token，却要把 16 GB 的权重和 KV Cache 读一遍，算术强度约为 1，每步约 4.8 ms，卡在显存带宽，算力只用了 0.3%。一个普通的请求，98% 的时间花在 decode 上。prefill 决定第一个 token 要等多久，decode 决定之后的速度；前者靠减少计算来优化，后者靠减少读取、批处理来优化。

decode 每一步都要读这个请求的 KV Cache。可如果没有 KV Cache，每一步就得把前面所有的 token 重新算一遍。KV Cache 到底省了多少？下一讲用 TinyLM 实际实现一遍：**KV Cache**。

第69讲 KV Cache

重点

26 自注意力　27 因果掩码　34 前向传播　68 Prefill 与 Decode

KV Cache 在前面的讲里已经出现过好几次：第 24 讲用它解释为什么生成不必每步重算，第 27 讲说明了它为什么能成立，第 39、40 讲为了压缩它改了注意力的结构。这一讲把它作为一个完整的推理技术来讲：**它具体怎么实现，省了多少计算，又付出了什么代价**。我们会在 TinyLM 上真的写一个带缓存的生成循环，并验证它和“每步重算整句”的结果一模一样。

它标了重点，因为 KV Cache 是推理系统里最核心的数据结构：第 10 篇的 PagedAttention、前缀缓存、连续批处理，第 14 篇的 PD 分离，几乎都围绕着“怎么存、怎么管、怎么传 KV Cache”展开。

不缓存要重算多少

先回顾第 34 讲的生成：每生成一个 token，就把整句话从头做一遍完整的前向，取最后一行的概率。提示有 4 个 token，生成第 1 个词时算 4 个位置，生成第 2 个时算 5 个，第 3 个时算 6 个……生成第 n 个时，前面所有 token 都要重新经过每一层。

可第 27 讲已经证明：因为有因果掩码，**每个 token 在每一层的输出，只取决于它自己和它前面的 token**，后面接了什么都不影响它。前面那些 token 上一步算过，这一步再算一遍，结果完全一样，纯属浪费。

缓存什么

重算的结果里，后面的计算真正要用到的是什么？新 token 在每一层做注意力时，需要前面每个 token 在这一层的 **k 和 v**（用来打分和加权求和）。前面 token 的 q 用不到（它们不再“提问”），它们在 FFN 里的计算结果也用不到（只影响它们自己）。

所以只需要缓存每一层、每个 token 的 k 和 v，这就是 **KV Cache**。有了它，decode 的每一步变成：

1. 只把**新 token** 送进模型。
2. 在每一层，只算新 token 自己的 q、k、v。

3. 把新的 k、v **追加**进这一层的缓存。
4. 用新 token 的 q，和缓存里所有的 k 打分、对所有的 v 加权求和。新 token 是最后一个，它能看的正是全部缓存，所以**不需要掩码**。
5. 新 token 继续经过这一层的 FFN，进入下一层。

prefill 时，整段提示一起算，把每一层所有 token 的 k、v 一次性写进缓存；之后的 decode 每步只写一组。

下面的演示把每一次前向画成一行：深色是这次要完整计算的位置，浅色是直接用缓存、不再计算的位置。切换“无缓存”和“有 KV Cache”，拖动已生成的步数，看累计计算量的差别。

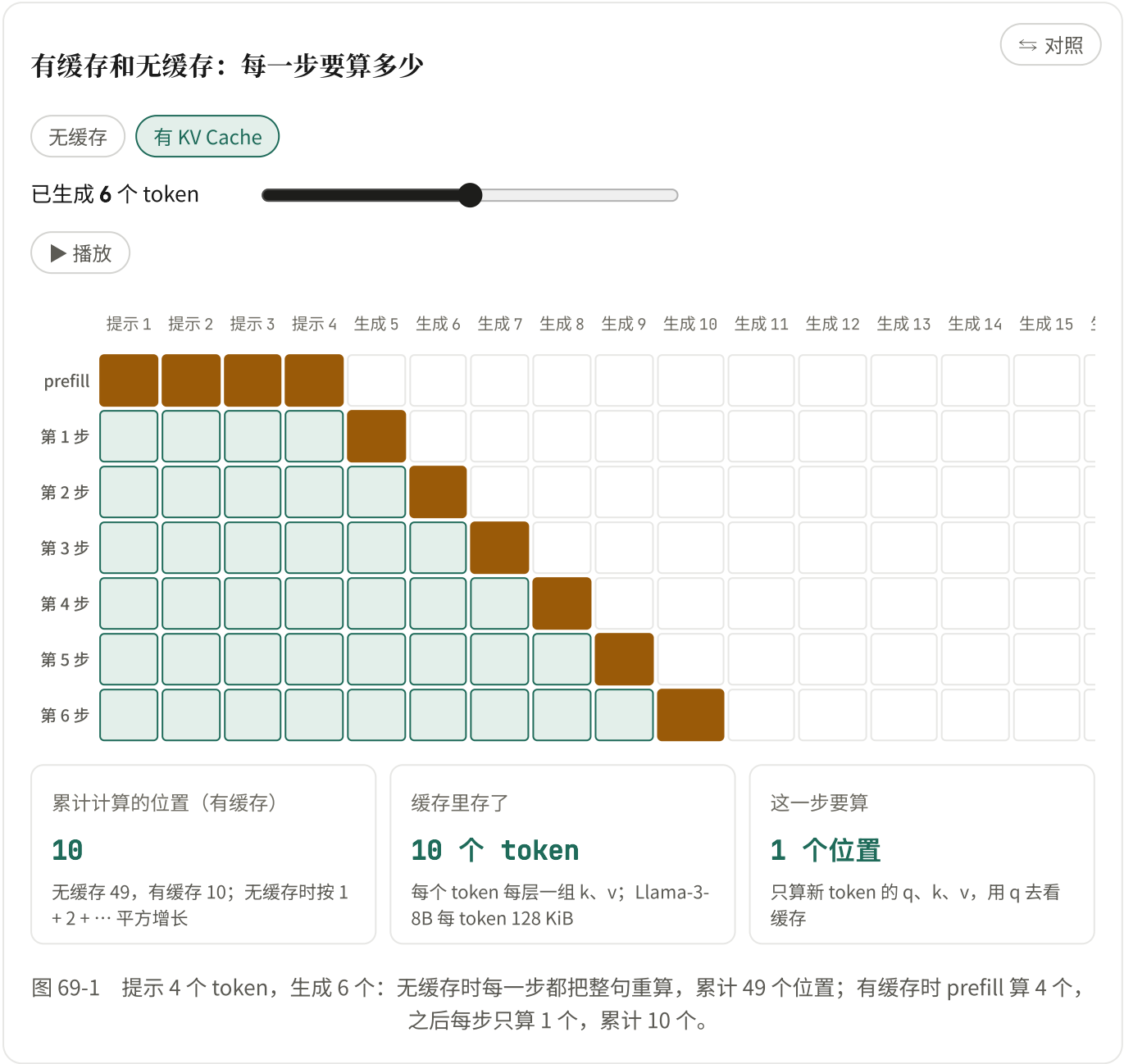


图 69-1　提示 4 个 token，生成 6 个：无缓存时每一步都把整句重算，累计 49 个位置；有缓存时 prefill 算 4 个，之后每步只算 1 个，累计 10 个。

无缓存时，深色的格子是一个三角形，累计计算量随生成长度**平方**增长；有缓存时，只有对角线上的一格（加上 prefill 那一行），**线性**增长。

> **要点**
> KV Cache 存的是每一层、每个历史 token 的 k 和 v。decode 每步只算新 token，把它的 k、v 追加进缓存，再用它的 q 去看全部缓存。计算从“每步重算整句”变成“每步只算一个 token”。

在 TinyLM 上实现一遍

下面的代码给 TinyLM 写了一个 `step` 函数：只处理一个新 token，在两层里各算它的 q、k、v，追加进缓存，再做注意力和 FFN。从“我”开始生成 6 个词，每一步都和“整句重算”（第 34 讲的完整前向）的结果比较。

结果：生成“我 推理 我 推理 推理 推理 AI”（前四个词和第 34 讲一样），**每一步的概率分布都和整句重算完全一致**。缓存里每层存了 7 个 k 和 7 个 v，每个 4 维。处理过的 token 位置：有缓存 7 个，无缓存 28 个（$1 + 2 + \cdots + 7$）。

注意 `step` 函数里位置编码用的是这个 token 在整句里的位置（第 25 讲），这正是推理引擎里 position ids 的作用：没有了整句，每个新 token 要自己知道自己排在第几。

省了多少，花了多少

把账算到 Llama-3-8B 上：提示 1000 个 token，再生成 1000 个。

- **无缓存**：每一步重算整句，一共要处理 $1000 + 1001 + \cdots + 1999 \approx 150$ 万个 token 位置，约 22.5 PFLOPs（每个位置约 2×75 亿次运算）。
- **有缓存**：prefill 1000 个，再 decode 999 步，一共 1999 个位置，约 30 TFLOPs。

计算量相差约 750 倍。没有 KV Cache，大模型的生成在实际中根本跑不动。

代价是**显存**。每个 token 每层要存一组 k、v：Llama-3-8B 每个 token 128 KiB（第 39 讲），这个请求结束时缓存有 250 MiB。它随序列长度线性增长，也随并发请求数线性增长。第 27 讲算过，16 个 8192 token 的请求，KV Cache 就要 16 GiB，和模型权重一样大。而且第 68 讲说过，decode 每一步都要把这个请求的整个 KV Cache **读一遍**，缓存越长，每一步越慢。

KV Cache 本质上是一笔**用显存换计算**的交易。推理系统接下来的很多工作，都是在想办法让这笔交易更划算：让每个 token 的缓存更小（GQA、MLA、KV Cache 量化），让缓存不浪费显存（PagedAttention，第 75 讲），让相同的前缀只存一份（前缀缓存，第 76 讲），显存不够时把缓存挪到别处或丢掉重算（第 78 讲的抢占）。

用代码复现

```
import json
import numpy as np

t = json.load(open("tinylm.json", encoding="utf-8"))
vocab, E, layers, W_lm = t["config"]["vocab"], np.array(t["weights"]["embedding"]),
t["weights"]["layers"], np.array(t["weights"]["lm_head"])
P = {k: [np.array(L[k]) for L in layers] for k in ["W_q", "W_k", "W_v", "W_o",
"W_1", "W_2"]}

def pe(pos, d=4):                                          # 第 25 讲的正弦位置编码（一
个位置）
    i = np.arange(0, d, 2)
    out = np.zeros(d); out[0::2], out[1::2] = np.sin(pos / 10000 ** (i / d)),
np.cos(pos / 10000 ** (i / d))
    return out

def ln(x, eps=1e-5):
    mu = x.mean(-1, keepdims=True)
    return (x - mu) / np.sqrt(((x - mu) ** 2).mean(-1, keepdims=True) + eps)

def softmax(z):
    e = np.exp(z - z.max(-1, keepdims=True)); return e / e.sum(-1, keepdims=True)

def step(token, pos, cache):
    """只算一个新 token：每层算它自己的 q、k、v，把 k、v 追加进缓存，再用 q 去看缓存里的全部
k、v。"""
    x = E[vocab.index(token)] + pe(pos)
    for l in range(2):
        q, k, v = x @ P["W_q"][l], x @ P["W_k"][l], x @ P["W_v"][l]
        cache[l]["k"].append(k); cache[l]["v"].append(v)
        K, V = np.array(cache[l]["k"]), np.array(cache[l]["v"])   # [已有 token 数,
4]
        heads = []
        for h in range(2):
            c = slice(2 * h, 2 * h + 2)
            a = softmax(q[c] @ K[:, c].T / np.sqrt(2))             # 新 token 看自
己和前面所有：不需要掩码
            heads.append(a @ V[:, c])
        x = ln(x + np.concatenate(heads) @ P["W_o"][l])
        x = ln(x + np.maximum(0, x @ P["W_1"][l]) @ P["W_2"][l])
    return softmax(x @ W_lm)

def full_forward(tokens):
    """不用缓存：每次把整句从头算一遍（第 34 讲），只取最后一行。"""
```

```
    s = len(tokens)
    x = np.array([E[vocab.index(w)] + pe(i) for i, w in enumerate(tokens)])
    mask = np.triu(np.full((s, s), -np.inf), 1)
    for l in range(2):
        Q, K, V = x @ P["W_q"][l], x @ P["W_k"][l], x @ P["W_v"][l]
        heads = [softmax(Q[:, 2*h:2*h+2] @ K[:, 2*h:2*h+2].T / np.sqrt(2) + mask) @
V[:, 2*h:2*h+2] for h in range(2)]
        x = ln(x + np.concatenate(heads, 1) @ P["W_o"][l])
        x = ln(x + np.maximum(0, x @ P["W_1"][l]) @ P["W_2"][l])
    return softmax(x @ W_lm)[-1]

# 1. 从“我”开始生成 6 个词：有缓存，每步只算 1 个新 token
cache = [{"k": [], "v": []} for _ in range(2)]
seq, rows_cached, rows_full, same = ["我"], 0, 0, True
p = step("我", 0, cache); rows_cached += 1
for i in range(6):
    nxt = vocab[int(p.argmax())]
    rows_full += len(seq)                                    # 不用缓存时，这一步要把已有
的 len(seq) 个 token 全算一遍
    same &= np.allclose(p, full_forward(seq))
    seq.append(nxt)
    p = step(nxt, len(seq) - 1, cache); rows_cached += 1
print("生成：", " ".join(seq))
print("每一步和“整句重算”的结果一致：", same)
print("缓存里每层存了", len(cache[0]["k"]), "个 k 和 v，形状", np.array(cache[0]
["k"]).shape)
print(f"处理过的 token 位置：有缓存 {rows_cached} 个，无缓存 {rows_full + len(seq)}
个")

# 2. Llama-3-8B：提示 1000 个 token，再生成 1000 个
p_len, n = 1000, 1000
no = sum(p_len + i for i in range(n)); yes = p_len + n - 1
print(f"Llama-3-8B：无缓存处理 {no:,} 个位置 ≈ {2 * 7.5e9 * no / 1e15:,.1f} PFLOPs；有
缓存 {yes:,} 个 ≈ {2 * 7.5e9 * yes / 1e12:.1f} TFLOPs")
print(f"代价：KV Cache 每 token 128 KiB，这个请求结束时 {(p_len + n) * 128 / 1024:.0f}
MiB")
```

输出：

```
生成： 我 推理 我 推理 推理 推理 AI
每一步和“整句重算”的结果一致： True
缓存里每层存了 7 个 k 和 v，形状 (7, 4)
处理过的 token 位置：有缓存 7 个，无缓存 28 个
Llama-3-8B：无缓存处理 1,499,500 个位置 ≈ 22.5 PFLOPs；有缓存 1,999 个 ≈ 30.0 TFLOPs
代价：KV Cache 每 token 128 KiB，这个请求结束时 250 MiB
```

前四行是 TinyLM 的结果：生成的句子，每一步都和整句重算一致，缓存的形状，以及有无缓存时处理过的位置数。最后两行是 Llama-3-8B 的估算。

到了大模型里

KV Cache 的形状。在推理引擎里，一层的 KV Cache 通常是 [请求数, KV 头数, 序列长度, 每头维度] 这样的张量，k 和 v 各一个，32 层就是 64 个。Llama-3-8B 每层每个 token：$2 \times 8 \times 128$ 个数。有了 RoPE（第 37 讲），存的是已经旋转过的 k。

decode 的注意力是“一行对一片”。新 token 的 q 只有一行，要和缓存里成千上万个 k 打分，计算很少，读的数据却很多。推理引擎为 decode 专门写了注意力算子（比如 FlashDecoding），把长长的缓存切成几段，让 GPU 的很多计算单元同时去读、去算，再把结果合并。

缓存要管理。请求来来去去，长短不一，缓存的显存要不停地分配、释放。如果像普通的张量那样给每个请求预留一整块“最大长度”的连续显存，大部分都会浪费掉。这正是第 75 讲 PagedAttention 要解决的问题。

容易混淆的地方

1. **只缓存 K、V，不缓存 Q**。历史 token 的 q 在后面的步骤里永远用不到，不需要存。
2. **缓存的是每一层的**。不只第一层，每一层的注意力都有自己的 k、v，都要缓存。层数越多，缓存越大。
3. **KV Cache 只在一次请求内有效**。请求结束，缓存就释放了。下一个请求即使开头一样，默认也要重新 prefill，除非用前缀缓存（第 76 讲）把它们留下来复用。

回顾

因果掩码保证了历史 token 的计算结果不随后面的 token 改变，所以不必每步重算整句。KV Cache 把每一层、每个历史 token 的 k 和 v 存下来；decode 每步只算新 token 的 q、k、v，追加缓存，再用 q 去看全部缓存。在 TinyLM 上，带缓存的生成和整句重算的结果完全一致，计算的位置从 28 个降到 7 个；在 Llama-3-8B 上，提示和生成各 1000 个 token 时，计算量从约 22.5 PFLOPs 降到约 30 TFLOPs。代价是显存：每 token 128 KiB，随长度和并发线性增长，decode 每步还要把它读一遍。

有了 prefill、decode 和 KV Cache 这三个概念，就可以讨论推理系统最关心的问题了：一个推理服务到底“快”不快？用户感受到的快，和服务商关心的快，是同一回事吗？下一讲：**TTFT、TPOT 与吞吐**。

第70讲 TTFT、TPOT 与吞吐

68 Prefill 与 Decode　69 KV Cache

“这个推理服务快不快？”这个问题，用户和服务商的回答可能完全相反。用户关心的是：我提问之后，多久能看到第一个字？之后字是一个个蹦出来，还是流畅地刷出来？服务商关心的是：一张 GPU 每秒能生成多少个 token、能同时服务多少人？

这一讲把“快”拆成三个最常用的指标：**TTFT**、**TPOT** 和**吞吐**，并用上一讲的估算方法说明一个推理系统最核心的矛盾：**同时服务的请求越多，总吞吐越高，但每个请求都会变慢**。

三个指标

TTFT（Time To First Token，首 token 时间）：从请求发出，到收到第一个生成的 token，要多久。它主要由 prefill 决定（第 68 讲），再加上排队等待的时间。用户在对话界面里“等它开始说话”的那段时间，就是 TTFT。

TPOT（Time Per Output Token，每个输出 token 的时间），也叫 ITL（Inter-Token Latency，token 间延迟）：第一个 token 之后，每个 token 之间隔多久。它由 decode 一步的时间决定。TPOT 越小，文字刷得越快。人的阅读速度大约是每秒几个到十几个 token，TPOT 在几十毫秒以内，用户就感觉很流畅。

一个请求的总耗时约是：

$$\text{总耗时} \approx \text{TTFT} + \text{TPOT} \times (\text{生成的 token 数} - 1)$$

吞吐（throughput）：整个系统每秒一共生成多少个 token（有时也按每秒完成多少个请求算）。它是服务商最关心的指标，直接决定了一张 GPU 能服务多少用户，也就决定了每个 token 的成本（第 72 讲）。

前两个是**延迟**指标，从单个请求的角度看；最后一个是**吞吐**指标，从整个系统的角度看。

一个人用：延迟低，吞吐也低

先看只服务一个请求的情况。用第 68 讲的方法，Llama-3-8B 在一张 H100 上，提示 1024 个 token，TTFT 约 15.5 ms；decode 时每一步要读一遍 16 GB 的权重，TPOT 约 4.85 ms，也就是每秒约 206 个 token。

对这个用户来说，体验非常好。可对 GPU 来说，这几乎是闲着：decode 的算术强度约为 1，算力只用了 0.3%（第 68 讲），一张昂贵的 H100 每秒只产出 206 个 token。

很多人一起用：批处理

第 68 讲提过解决办法：**批处理**。让 b 个请求一起 decode，每一步读一遍权重，同时为 b 个请求各生成一个 token。权重的读取被 b 个请求分摊了。

可是每个请求还有自己的 KV Cache，这部分没法共享，要各读各的。每个请求总长 2048 个 token（提示和生成各 1024），decode 时平均上下文约 1536 个，每个请求每步要读约 192 MiB 的 KV Cache。所以一步 decode 的读写量是：16 GB 的权重，加上 b × 192 MiB 的 KV Cache。

算一下不同的并发数：

并发 b	TPOT	每个请求的速度	总吞吐	显存（权重 + KV Cache）
1	4.85 ms	206 token/s	206 token/s	16.3 GB
8	5.27 ms	190 token/s	1,517 token/s	18.2 GB
32	6.72 ms	149 token/s	4,764 token/s	24.6 GB
64	8.64 ms	116 token/s	7,407 token/s	33.2 GB
128	12.49 ms	80 token/s	10,251 token/s	50.4 GB
256	20.18 ms	50 token/s	12,686 token/s	84.8 GB（放不下）

规律很清楚：

- **并发从 1 增加到 8**，TPOT 只从 4.85 ms 涨到 5.27 ms，总吞吐却涨了 7 倍多。权重的读取被分摊，几乎是“白捡”的吞吐。
- **并发继续增加**，每个请求的 KV Cache 读取开始占主导，TPOT 明显变长，每个请求变慢了，总吞吐的增长也越来越慢。
- **显存是硬上限**。每个请求的 KV Cache 要 256 MiB，80 GB 的卡放下权重以后，最多同时装约 238 个请求。256 个就放不下了。

注意算术强度：即使并发 256，也只有约 60，远没到 H100 的 295。这是因为每个请求的 KV Cache 读取随并发线性增长，“每读一个字节做的运算”上不去。所以长上下文的 decode 往往**始终是带宽受限**的，靠批处理也只能把权重的那部分摊薄。

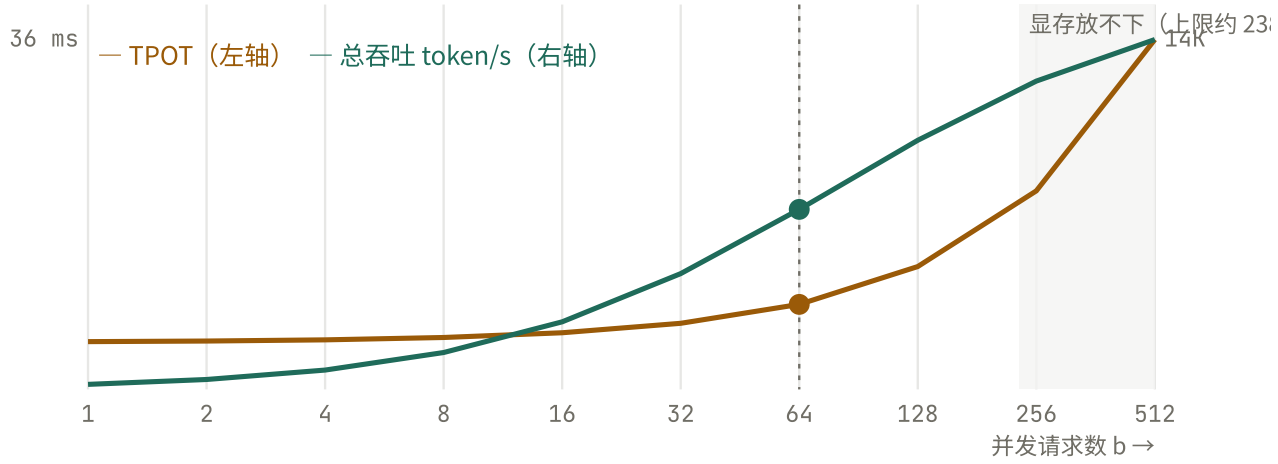
下面的演示画出了 TPOT（琥珀色，左轴）和总吞吐（绿色，右轴）随并发数的变化，灰色区域是显存放不下的范围。换不同的请求长度看看。

并发越多，单个请求越慢，总吞吐越高

⇆ 对照

并发请求数 b = **64**

每个请求长度 512 | 2048 | 8192 | 32768



TPOT（每个 token 的间隔）	总吞吐	显存
8.64 ms	**7,407 token/s**	**33.2 GB**
每个请求每秒 116 个 token	64 个请求一起，算术强度 35.0（H100 的分界线是 295）	权重 16.1 GB + KV Cache；最多约 238 个请求

图 70-1　并发数对 decode 的影响（每请求 2048 token）。并发越多，TPOT 越长（每个请求变慢），总吞吐越高；显存在约 238 个请求时到顶。请求越长，KV Cache 越大，这个上限越低。

把请求长度调到 32768：每个请求的 KV Cache 要 4 GiB，显存只够装十几个请求，而且 KV Cache 的读取很快就压过了权重，批处理的收益变得很小。长上下文的服务，吞吐会低得多。

要点

TTFT 看第一个 token 等多久，由 prefill 和排队决定；TPOT 看之后每个 token 的间隔，由 decode 一步决定；吞吐看整个系统每秒生成多少 token。增加并发能大幅提高吞吐，代价是每个请求的 TPOT 变长，上限由显存决定。

延迟和吞吐的权衡

这张表说明了推理服务最根本的权衡：**用延迟换吞吐**。服务商希望并发越大越好（每个 token 更便宜），用户希望自己的请求越快越好。

实际的做法是先定一个**服务等级目标**（SLO），比如“TTFT 不超过 500 ms，TPOT 不超过 50 ms”，然后在满足它的前提下，把并发和吞吐推到最高。不同的场景，SLO 很不一样：在线聊天要求 TPOT 低、TTFT 低；离线批量处理（比如给一百万篇文档写摘要）只关心总吞吐，延迟可以很长；推理模型（第 49 讲）生成的 token 很多，TPOT 直接决定用户要等几十秒还是几分钟。

还有一个这张表没体现的细节：新请求要做 prefill。在真实的服务里，请求是陆续到达的，新请求的 prefill 会插进正在进行的 decode 之间。一个很长的 prefill 会让所有正在 decode 的请求“卡”一下（TPOT 突然变长）；而新请求如果要排队等前面的 prefill，TTFT 就会变长。怎么安排这些 prefill 和 decode，是第 10 篇调度的核心问题（第 74、77 讲）。

用代码复现

下面的代码用第 68 讲的估算方法，算出单独 prefill 的 TTFT，以及不同并发数下 decode 一步的 TPOT、每个请求的速度、总吞吐、算术强度和显存，最后算出显存能放下的最大并发数。

```
# Llama-3-8B (BF16) + 一张 H100 (80 GB) 。每个请求：提示 1024 个 token，生成 1024 个
 (decode 时上下文平均约 1536)
N, N_emb, L, d = 8.03e9, 128256 * 4096, 32, 4096
W_BYTES, KV, F, BW, MEM = N * 2, 2 * 32 * 8 * 128 * 2, 989e12, 3.35e12, 80e9
prompt, out = 1024, 1024
ctx = prompt + out / 2

def decode_step(b):
    """b 个请求一起 decode 一步：权重读一遍，每个请求读自己的 KV Cache。返回毫秒。"""
    flops = (2 * (N - N_emb) + 4 * L * d * ctx) * b
    bytes_ = W_BYTES + KV * ctx * b
    return max(flops / F, bytes_ / BW) * 1e3, flops / bytes_

ttft = max(2 * (N - N_emb) * prompt / F, W_BYTES / BW) * 1e3
print(f"TTFT (单独 prefill 一个 1024 token 的提示) ≈ {ttft:.1f} ms")
print(" 并发 b | TPOT (ms)  | 每个请求的速度 | 总吞吐 (token/s)  | 算术强度 | KV Cache +
权重")
for b in [1, 8, 32, 64, 128, 256]:
    t, ai = decode_step(b)
    mem = W_BYTES + KV * (prompt + out) * b
    print(f"{b:6d} | {t:9.2f} | {1000 / t:9.1f} tok/s | {b * 1000 / t:15,.0f} |
{ai:8.1f} | {mem / 1e9:5.1f} GB{' (超出 80 GB) ' if mem > MEM else ''}")

# 显存最多放得下多少个请求（先不管激活值和其他开销，第 71 讲细算）
b_max = int((MEM - W_BYTES) // (KV * (prompt + out)))
t, _ = decode_step(b_max)
print(f"显存上限：最多约 {b_max} 个请求同时 decode；此时 TPOT {t:.1f} ms，吞吐 {b_max *
1000 / t:,.0f} token/s")
```

输出：

```
TTFT (单独 prefill 一个 1024 token 的提示) ≈ 15.5 ms
 并发 b | TPOT (ms)  | 每个请求的速度 | 总吞吐 (token/s)  | 算术强度 | KV Cache + 权重
     1 |      4.85 |     206.0 tok/s |              206 |      1.0 |  16.3 GB
     8 |      5.27 |     189.6 tok/s |            1,517 |      7.2 |  18.2 GB
    32 |      6.72 |     148.9 tok/s |            4,764 |     22.5 |  24.6 GB
    64 |      8.64 |     115.7 tok/s |            7,407 |     35.0 |  33.2 GB
   128 |     12.49 |      80.1 tok/s |           10,251 |     48.4 |  50.4 GB
   256 |     20.18 |      49.6 tok/s |           12,686 |     59.9 |  84.8 GB (超出
80 GB)
显存上限：最多约 238 个请求同时 decode；此时 TPOT 19.1 ms，吞吐 12,463 token/s
```

第一行是 TTFT。表格就是上面那张。最后一行：显存最多放下约 238 个请求，此时 TPOT 19.1 ms，吞吐约 12463 token/s。

到了大模型里

这些都是理想值。实测的吞吐通常只有理论值的一半到七成：算子之间有开销，注意力在长短不一的请求上效率不同，还有调度、采样、CPU 端的处理。第 67 讲会把理论值和实测值放在一起对比。

看指标要看分布，不只看平均。线上服务通常报告 TTFT、TPOT 的 P50、P90、P99（50%、90%、99% 的请求不超过多少）。平均值很好看、但 P99 很差，意味着总有一部分用户等得很久。

压测工具。vLLM、SGLang 等推理引擎都带有压测脚本：按设定的速率发送请求，统计 TTFT、TPOT、吞吐的分布。调参（并发上限、批大小、chunked prefill 的块大小等）时，就是在这些指标之间找平衡。

容易混淆的地方

1. **吞吐高不等于每个用户都快**。并发 128 时总吞吐是并发 1 的 50 倍，但每个用户的速度从 206 token/s 降到了 80 token/s。
2. **TTFT 不只是 prefill 的时间**。还包括排队、分词、网络传输等。系统繁忙时，排队往往是 TTFT 的大头。
3. **“每秒 token 数”要看清是哪个**。有的说的是单个请求的输出速度（1 / TPOT），有的说的是整个系统的总吞吐，有的还把输入 token 也算进去。比较不同系统的数字时要先对齐口径。

回顾

衡量推理服务的“快”有三个指标：TTFT（第一个 token 等多久，由 prefill 和排队决定），TPOT（之后每个 token 的间隔，由 decode 一步决定），吞吐（系统每秒生成多少 token）。批处理让多个请求共享一次权重读取：Llama-3-8B 在 H100 上，并发从 1 增加到 128，TPOT 从 4.85 ms 变成 12.49 ms，总吞吐却从 206 涨到 10251 token/s。但每个请求的 KV Cache 要单独读，长上下文时它很快成为主导；而显存决定了并发的上限（每请求 2048 token 时约 238 个）。实际服务在满足延迟目标的前提下，尽量提高吞吐。

“显存能放下多少个请求”这个问题，我们只粗算了权重和 KV Cache。真实的显存里还有激活值、框架开销、碎片……一张 80 GB 的卡，到底每一块都被谁占了？下一讲：**显存账本**。

第71讲 显存账本

重点

35 参数量与计算量　39 MQA 与 GQA　69 KV Cache　70 TTFT、TPOT 与吞吐

部署一个模型前，第一个要回答的问题往往不是“多快”，而是“**放不放得下**”：这个模型能不能装进这张卡？装进去以后，还剩多少地方给 KV Cache？能同时服务多少个请求？上一讲看到，并发数直接决定了吞吐，而并发数的上限是显存给的。

这一讲把一张 GPU 的显存当成一本账，逐项算清楚：**权重、KV Cache、激活、其他开销**各占多少。这本账是部署时选显卡、定精度、定卡数、定最大并发的依据，所以标了重点。

账本上的四项

权重：最大、也最确定的一项。参数个数乘以每个参数的字节数（第 35 讲）：Llama-3-8B 用 BF16 是 80.3 亿 × 2 字节 ≈ 16.06 GB，即 15.0 GiB。用 FP8 减半，用 4 位量化再减半（第 12 篇）。多卡张量并行时，每张卡放 1/n（第 105 讲）。

KV Cache：随请求变化的一项，通常是除权重以外最大的。每个 token 的 KV Cache 是 2 × 层数 × KV 头数 × 每头维度 × 字节数（第 39 讲），Llama-3-8B 用 BF16 是 128 KiB；乘上所有请求的 token 总数，就是 KV Cache 的总量。

激活：前向计算时的临时结果，比如每层的 Q、K、V、FFN 中间那个 14336 维的向量（第 31 讲）。推理时不需要像训练那样把它们全部留着给反向用（第 18 讲），算完一层就可以复用这块空间，所以只需要按“一次最多处理多少个 token”预留。prefill 一次处理一大块 token 时最大。按一块 8192 个 token 粗估，Llama-3-8B 约 0.9 GiB。

其他开销：CUDA 运行时本身、矩阵乘法库的工作空间、多卡通信的缓冲区、CUDA Graph 等，粗估一两个 GiB。另外，推理引擎通常不会把显存用到 100%，比如 vLLM 默认只用 90%（参数 `gpu_memory_utilization`），剩下的留给意外和碎片。

算一张卡的账

拿 Llama-3-8B、BF16、一张 80 GB 的 H100 算一下：

项目	大小
可用显存（80 GB 的 90%）	67.1 GiB
权重	15.0 GiB
激活（prefill 一块 8192 token，粗估）	0.91 GiB
其他开销（粗估）	1.9 GiB
剩给 KV Cache	**49.3 GiB**

每个 token 的 KV Cache 是 128 KiB，49.3 GiB 能放约 40 万个 token。如果每个请求的长度是 4096 个 token，最多同时放约 **99 个请求**。

把权重和 KV Cache 都换成 FP8：权重降到 7.5 GiB，每个 token 的 KV Cache 降到 64 KiB，KV Cache 的空间增加到 56.8 GiB，能放约 93 万个 token，**227 个请求**，翻了一倍多。这就是低精度在推理里最直接的价值之一：同样的卡，能服务更多的人。

大模型放不下怎么办

换成 Llama-3-70B：

- **BF16，一张 80 GB 的卡**：权重 131.4 GiB，远超可用的 67.1 GiB，放不下。
- **FP8，一张卡**：权重 65.7 GiB，加上激活和其他开销，已经超过 67.1 GiB，没有地方放 KV Cache。权重勉强“装得进”，模型却没法服务任何请求。
- **BF16，两张卡（张量并行）**：每张卡 65.7 GiB 的权重，同样没有地方放 KV Cache。
- **BF16，四张卡**：每张卡权重 32.9 GiB，剩 31.9 GiB 给 KV Cache；每个 token 的 KV Cache 也被 4 张卡分摊，每张卡 80 KiB，能放约 102 个 4096 token 的请求。

所以“一张卡能装下权重”远远不够，还要给 KV Cache 留出足够的空间，否则并发上不去，吞吐就很低，成本也就很高。选多少张卡，本质上就是在权衡“卡的成本”和“KV Cache 的空间（也就是吞吐）”。

下面的显存计算器可以自己换模型、精度、显卡和卡数，调请求长度和并发数，看一张卡的显存条怎么分配。

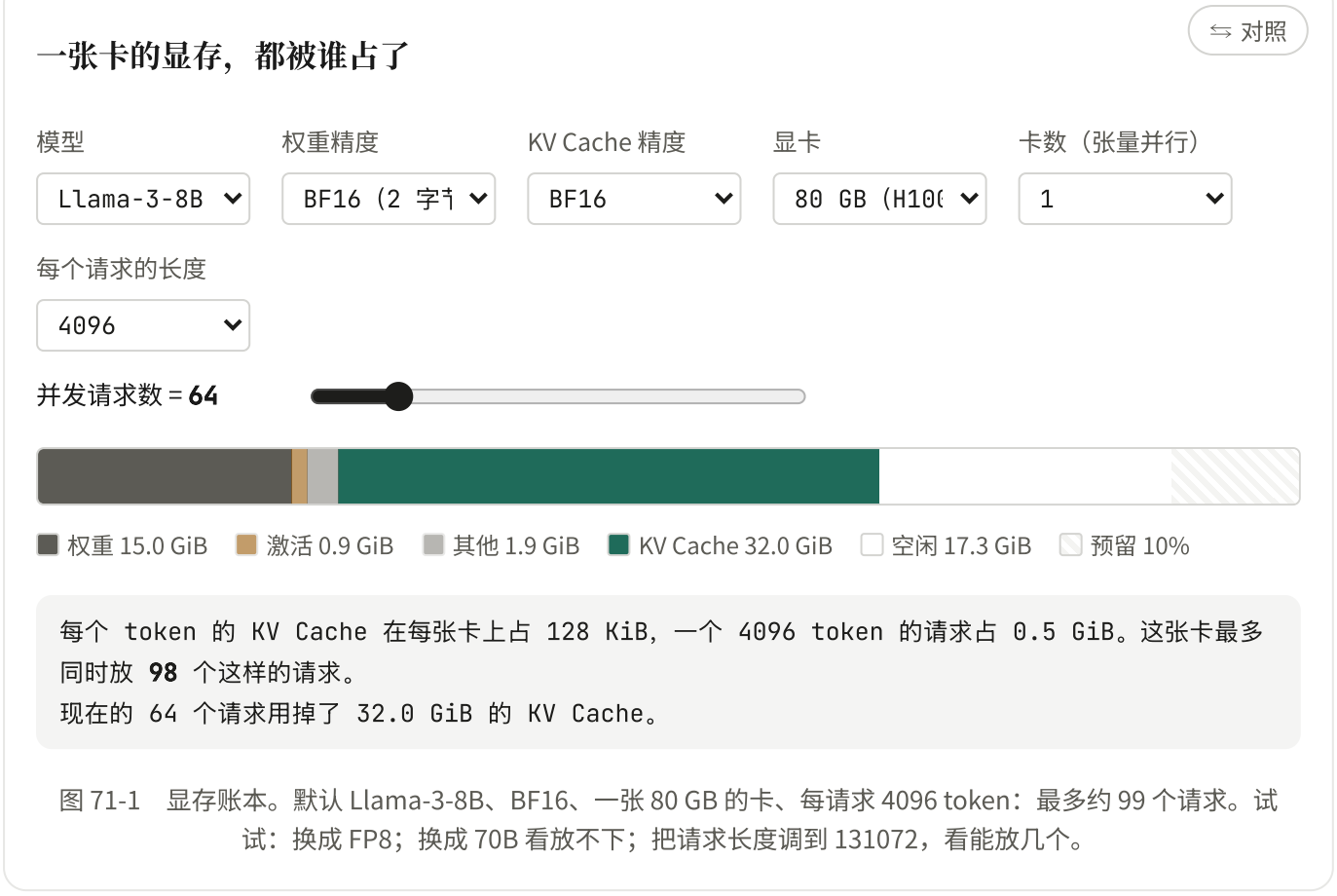


图 71-1　显存账本。默认 Llama-3-8B、BF16、一张 80 GB 的卡、每请求 4096 token：最多约 99 个请求。试试：换成 FP8；换成 70B 看放不下；把请求长度调到 131072，看能放几个。

要点

显存账本 = 权重 + KV Cache + 激活 + 其他开销（再加推理引擎的预留）。权重固定，激活和其他开销不大，剩下的全部给 KV Cache；KV Cache 的空间除以每个请求的 KV Cache，就是最大并发数。

推理引擎怎么用这本账

vLLM 这类推理引擎启动时，做的正是这一讲的计算：先加载权重；再用一批假数据跑一次最大规模的前向，测出激活的峰值；用“可用显存 − 权重 − 激活峰值 − 其他”算出 KV Cache 能用多少，然后一次性把这些显存全部申请下来，切成一个个固定大小的块，作为 KV Cache 的“池子”（第 75 讲的 PagedAttention）。启动日志里通常会打印出“能放多少个 KV Cache 块”或“最多多少个 token”，就是这本账的结果。

之后，每来一个请求，就从池子里分配块；请求结束，块就还回池子。池子满了，新请求就得排队，或者把正在进行的请求换出（第 78 讲）。

用代码复现

下面的代码把这本账写成一个函数，算了 5 种配置：Llama-3-8B 的 BF16 和 FP8，Llama-3-70B 的单卡 BF16、单卡 FP8、两卡和四卡 BF16。

```
MODELS = {   # 参数量、层数、隐藏维度、FFN 维度、KV 头数、每头维度
    "Llama-3-8B": dict(N=8.03e9, L=32, d=4096, f=14336, kv=8, dh=128),
    "Llama-3-70B": dict(N=70.55e9, L=80, d=8192, f=28672, kv=8, dh=128),
}
GiB = 2 ** 30

def ledger(name, gpu_gb, n_gpu=1, w_bytes=2, kv_bytes=2, ctx=4096, chunk=8192,
util=0.9, overhead_gb=2.0):
    m = MODELS[name]
    total = gpu_gb * 1e9 * util                                  # 推理引擎只用其中
90%，留一点余量
    weights = m["N"] * w_bytes / n_gpu                           # 张量并行：每张卡放
1/n 的权重（第 105 讲）
    act = chunk * (4 * m["d"] + 3 * m["f"]) * 2 / n_gpu          # prefill 一块 8192
个 token 时的临时激活（粗估，层与层之间复用）
    other = overhead_gb * 1e9                                    # CUDA 上下文、通信缓
冲、CUDA Graph 等（粗估）
    kv_tok = 2 * m["L"] * m["kv"] * m["dh"] * kv_bytes / n_gpu   # 每个 token 的 KV
Cache，在这张卡上的部分
    kv_space = total - weights - act - other
    reqs = kv_space / (kv_tok * ctx)
    print(f"{name}, {n_gpu} × {gpu_gb} GB，权重 {w_bytes} 字节 / KV {kv_bytes} 字
节：")
    if kv_space ⩽ 0:
        print(f"  权重 {weights / GiB:.1f} + 激活 {act / GiB:.2f} + 其他 {other /
GiB:.1f} GiB，已超过可用的 {total / GiB:.1f} GiB，没有地方放 KV Cache"); return
    print(f"  可用 {total / GiB:.1f} GiB = 权重 {weights / GiB:.1f} + 激活 {act /
GiB:.2f} + 其他 {other / GiB:.1f} + KV Cache {kv_space / GiB:.1f}")
    print(f"  每 token KV {kv_tok / 1024:.0f} KiB，能放 {kv_space / kv_tok:,.0f} 个
token，即 {reqs:.0f} 个 {ctx} token 的请求")

ledger("Llama-3-8B", 80)
ledger("Llama-3-8B", 80, w_bytes=1, kv_bytes=1)                  # 权重和 KV Cache 都
用 FP8
ledger("Llama-3-70B", 80)
ledger("Llama-3-70B", 80, w_bytes=1)
ledger("Llama-3-70B", 80, n_gpu=2)
ledger("Llama-3-70B", 80, n_gpu=4)
```

输出：

```
Llama-3-8B, 1 × 80 GB, 权重 2 字节 / KV 2 字节:
  可用 67.1 GiB = 权重 15.0 + 激活 0.91 + 其他 1.9 + KV Cache 49.3
  每 token KV 128 KiB, 能放 404,106 个 token, 即 99 个 4096 token 的请求
Llama-3-8B, 1 × 80 GB, 权重 1 字节 / KV 1 字节:
  可用 67.1 GiB = 权重 7.5 + 激活 0.91 + 其他 1.9 + KV Cache 56.8
  每 token KV 64 KiB, 能放 930,739 个 token, 即 227 个 4096 token 的请求
Llama-3-70B, 1 × 80 GB, 权重 2 字节 / KV 2 字节:
  权重 131.4 + 激活 1.81 + 其他 1.9 GiB, 已超过可用的 67.1 GiB, 没有地方放 KV Cache
Llama-3-70B, 1 × 80 GB, 权重 1 字节 / KV 2 字节:
  权重 65.7 + 激活 1.81 + 其他 1.9 GiB, 已超过可用的 67.1 GiB, 没有地方放 KV Cache
Llama-3-70B, 2 × 80 GB, 权重 2 字节 / KV 2 字节:
  权重 65.7 + 激活 0.91 + 其他 1.9 GiB, 已超过可用的 67.1 GiB, 没有地方放 KV Cache
Llama-3-70B, 4 × 80 GB, 权重 2 字节 / KV 2 字节:
  可用 67.1 GiB = 权重 32.9 + 激活 0.45 + 其他 1.9 + KV Cache 31.9
  每 token KV 80 KiB, 能放 417,950 个 token, 即 102 个 4096 token 的请求
```

逐段对照：8B BF16 能放 99 个 4096 token 的请求，FP8 能放 227 个；70B 单卡 BF16、单卡 FP8、两卡 BF16 都没有地方放 KV Cache；四卡 BF16 能放 102 个。

到了大模型里

长上下文让账本紧张得多。每个请求 131072 个 token 时，Llama-3-8B 一个请求的 KV Cache 就要 16 GiB，一张 80 GB 的卡只能放 3 个。长上下文的服务，并发低、吞吐低，每个 token 的成本也高得多。这是第 39、40 讲压缩 KV Cache 的动力，也是 KV Cache 量化、卸载到 CPU 内存等技术的出发点。

你的 DGX Spark 是统一内存。它的 CPU 和 GPU 共用 128 GB 的内存（待核实），能装下比 80 GB 显卡更大的模型，比如 FP8 的 70B 模型，还能留出一些 KV Cache 的空间；但第 68 讲说过，它的带宽低得多，decode 会慢不少。“装得下”和“跑得快”是两回事。

这本账是第四章的起点。多卡、多机推理的每一种切分方式（张量并行、流水线并行、专家并行），首先改变的都是每张卡上这本账的分配：每张卡放多少权重、多少 KV Cache。

容易混淆的地方

1. **能装下权重不等于能部署**。还要给 KV Cache 留足空间，否则并发上不去。70B 的 FP8 权重“刚好装进”80 GB，实际上几乎没法用。

2. **GB 和 GiB 差 7%**。显卡标称的 80 GB 是 80×10^9 字节，约 74.5 GiB。推理引擎的日志通常用 GiB，算账时要统一单位。
3. **激活和其他开销是粗估**。它们和具体的推理引擎、配置（比如 prefill 一次处理多少 token）有关，实际以引擎启动时测出的为准。但它们通常只占几个 GiB，不影响账本的大局。

回顾

一张 GPU 的显存分成四块：权重（参数 × 字节数，最大也最确定）、KV Cache（每 token 的大小 × 所有请求的 token 总数）、激活（按一次处理的 token 数预留，约 1 GiB）、其他开销（一两个 GiB），推理引擎还会预留约 10%。Llama-3-8B 在一张 80 GB 的卡上用 BF16，剩给 KV Cache 约 49 GiB，能放约 99 个 4096 token 的请求；换成 FP8 能放 227 个。Llama-3-70B 用 BF16 至少要四张卡，才有足够的 KV Cache 空间。推理引擎启动时做的就是这本账，然后把剩下的显存全部做成 KV Cache 的池子。

知道了一张卡能放多少请求、每一步要多久，就可以算出一张卡每秒能生成多少 token，再结合 GPU 的价格，算出每个 token 要花多少钱。下一讲：**推理成本模型**。

第72讲 推理成本模型

重点

68 Prefill 与 Decode　70 TTFT、TPOT 与吞吐　71 显存账本

前几讲分别算了速度（第 68、70 讲）和显存（第 71 讲）。这一讲把它们串成一个完整的**成本模型**：给定模型、显卡、并发和请求长度，手算出理论速度，再结合显卡的价格，得到**每百万 token 要花多少钱**。

这个数非常有用。它可以和 API 的价格对比，判断自己部署是否划算；可以比较不同的显卡、不同的精度、不同的并发设置，哪种更省；可以估算一个产品上线后每个月的推理开销。它标了重点，因为第三、四章的每一种优化，最终都会落到这个数上：它让成本降了多少。

从速度到成本

成本的逻辑很简单：GPU 按时间收费（无论自己买还是租，都可以折算成每小时多少钱），在这一小时里它能处理多少 token，一除就是每个 token 的成本：

$$\text{每百万 token 成本} = \text{每小时租金} / \text{每小时处理的 token 数} \times 10^6$$

关键是“每小时处理多少 token”。第 68 讲说过，prefill 和 decode 的瓶颈不同，所以分开算：

输入 token（prefill）：计算受限。每个输入 token 约需 2 × 75 亿次运算（加上一点注意力的计算），H100 每秒 989 万亿次，理论上每秒能处理约 6.5 万个输入 token。

输出 token（decode）：带宽受限，吞吐取决于并发（第 70 讲）。b 个请求一起 decode，一步的耗时是 max(计算, 读写)，吞吐 = b / TPOT。

还有一个现实的因素：**效率**。实际的推理引擎达不到理论的峰值算力和带宽，有各种开销。这里取 60% 作为示例（第 67 讲会用实测来校准这个数）。

算一笔账

Llama-3-8B，一张 H100，租金假设为每小时 2.5 美元（示例，实际价格因云厂商和时间差别很大，请换成你自己的），效率 60%，每个请求提示和输出各 1024 个 token：

并发	TPOT	总输出速度	每百万输入 token	每百万输出 token
1	8.1 ms	124 token/s	$0.018	$5.618
32	11.2 ms	2,858 token/s	$0.018	$0.243
128	20.8 ms	6,151 token/s	$0.018	$0.113

几个结论：

并发决定输出成本。只服务一个请求时，每百万输出 token 要 5.6 美元；并发 128 时只要 0.11 美元，便宜了 50 倍。同一张卡、同一个模型，成本差别全在“能不能把并发提上去”。这就是推理系统花大力气做批处理、调度、省显存的原因：每多装一个请求，每个 token 就便宜一点。

输出比输入贵得多。并发 128 时，输出 token 的成本是输入的 6 倍多。原因第 68 讲已经说过：prefill 时同一份权重被很多个 token 共用，decode 时每一步都要把权重重读一遍。这也是为什么几乎所有大模型 API 的定价，输出 token 都比输入 token 贵好几倍。

一个请求的成本。提示 1024、输出 1024 个 token，并发 128 时约 0.00013 美元，也就是一万次请求约 1.3 美元。

换成 Llama-3-70B，用 4 张 H100（第 71 讲算过，至少要 4 张），并发 64：每百万输出 token 约 0.94 美元，输入约 0.16 美元，大约是 8B 的 8–9 倍，和参数量的比例差不多。

下面的计算器实现了这一讲的全部公式。改一改租金、效率、并发和请求长度，看成本怎么变。

每百万 token 要花多少钱

⇆ 对照

模型 Llama-3-8B

每张卡租金（美元/小时） 2.5

效率（实际 / 理论） 60%

提示长度 1024

输出长度 1024

并发请求数 = **128**

TTFT（单独 prefill）	TPOT	每百万输入 token
25.9 ms	**20.8 ms**	**$0.018**
prefill 每秒可处理 38,839 个输入 token	每个请求 48 token/s；总输出 6,151 token/s	租金 $2.5/小时 ÷ 每小时 140M 个

每百万输出 token	一个请求
$0.113	**$0.00013**
÷ 每小时 22.1M 个	1024 个输入 + 1024 个输出

输出 token 比输入贵得多：prefill 时同一份权重被很多 token 共用，decode 时每一步都要把权重重读一遍。并发越高，权重的读取被分摊得越多，输出越便宜。

图 72-1　推理成本计算器（理论模型 × 效率）。默认 Llama-3-8B、一张 H100、$2.5/小时、效率 60%、并发 128：每百万输出 token 约 $0.11。试试：并发调到 1；输出长度调到 8000（推理模型）；换成 70B。

要点

推理成本 = 租金 / 吞吐。输入 token（prefill）计算受限，很便宜；输出 token（decode）带宽受限，成本主要取决于并发：并发越高，权重读取被分摊得越多，每个 token 越便宜。所有推理优化，最终都体现为这个数的下降。

哪些因素让成本变化

用这个模型，可以快速判断各种优化“值不值”：

- **量化成 FP8**：权重的字节数减半，decode 每步读的数据减少，带宽受限时 TPOT 接近减半；KV Cache 也减半，能装下更多请求（第 71 讲）。输出成本可能降一半左右。
- **GQA、MLA**：KV Cache 变小，长上下文时 decode 读的数据大幅减少，并发上限提高。

- **更长的输出**（比如推理模型想 8000 个 token）：每个请求的输出 token 多了几倍，KV Cache 也更大，能装的并发变少，每个 token 也更贵。
- **前缀缓存**（第 76 讲）：很多请求共享同一个系统提示时，这部分的 prefill 只需做一次，输入成本大幅下降。很多 API 对“命中缓存的输入 token”打折，原因就在这里。

用代码复现

下面的代码实现了这个成本模型：分别算 prefill 的输入吞吐和 decode 的输出吞吐（按并发和上下文），再按租金折算成每百万 token 的成本。

```
# 推理成本模型：理论速度 × 实际效率 → 每小时生成多少 token → 每百万 token 多少钱
F1, BW1, PRICE1 = 989e12, 3.35e12, 2.5          # 一张 H100：BF16 算力、显存带宽、假设的租金（美元/小时，示例）
EFF = 0.6                                        # 实际能达到理论值的比例（示例，第 67 讲会实测）
MODELS = {"Llama-3-8B": dict(N=8.03e9, Ne=525e6, L=32, d=4096, kv=131072, gpus=1),
          "Llama-3-70B": dict(N=70.55e9, Ne=1051e6, L=80, d=8192, kv=327680, gpus=4)}

def costs(name, b, prompt=1024, out=1024):
    m = MODELS[name]
    F, BW, price = F1 * m["gpus"], BW1 * m["gpus"], PRICE1 * m["gpus"]   # 多卡按总算力、总带宽、总租金算
    ctx = prompt + out / 2
    # prefill：计算受限，每秒能处理的输入 token
    prefill_tps = F / (2 * (m["N"] - m["Ne"]) + 4 * m["L"] * m["d"] * prompt / 2) * EFF
    # decode：b 个请求一起，一步 = max(计算，读写)
    flops = (2 * (m["N"] - m["Ne"]) + 4 * m["L"] * m["d"] * ctx) * b
    bytes_ = m["N"] * 2 + m["kv"] * ctx * b
    tpot = max(flops / F, bytes_ / BW) / EFF
    decode_tps = b / tpot
    usd_in = price / (prefill_tps * 3600) * 1e6
    usd_out = price / (decode_tps * 3600) * 1e6
    return tpot * 1e3, decode_tps, usd_in, usd_out

for name, b in [("Llama-3-8B", 1), ("Llama-3-8B", 32), ("Llama-3-8B", 128), ("Llama-3-70B", 64)]:
    tpot, tps, ci, co = costs(name, b)
    print(f"{name:11s} 并发 {b:3d}：TPOT {tpot:5.1f} ms，输出 {tps:6,.0f} token/s；每百万 token：输入 ${ci:.3f}，输出 ${co:.3f}")

# 一个请求（提示 1024、生成 1024）在 8B、并发 128 时的成本
_, _, ci, co = costs("Llama-3-8B", 128)
print(f"一个请求：输入 1024 × ${ci:.3f}/M + 输出 1024 × ${co:.3f}/M = ${(1024 * ci + 1024 * co) / 1e6:.5f}")
```

输出：

```
Llama-3-8B  并发   1: TPOT   8.1 ms, 输出    124 token/s; 每百万 token: 输入 $0.018, 输出 $5.618
Llama-3-8B  并发  32: TPOT  11.2 ms, 输出  2,858 token/s; 每百万 token: 输入 $0.018, 输出 $0.243
Llama-3-8B  并发 128: TPOT  20.8 ms, 输出  6,151 token/s; 每百万 token: 输入 $0.018, 输出 $0.113
Llama-3-70B 并发  64: TPOT  21.6 ms, 输出  2,969 token/s; 每百万 token: 输入 $0.164, 输出 $0.936
一个请求: 输入 1024 × $0.018/M + 输出 1024 × $0.113/M = $0.00013
```

前四行就是上面的表和 70B 的例子。最后一行是一个请求的成本。

到了大模型里

这个模型故意很简单。它把 prefill 和 decode 分开，各自按“整张卡都在做这件事”算成本；真实的服务里两者交错进行，会互相影响（第 70 讲）。它也没有考虑请求到达的不均匀：夜里请求少，GPU 闲着，也照样计费。实际的平均成本会比这里算的高，取决于 GPU 的利用率。

但量级是对的。用它估算出来的数，通常和实测差在两倍以内。在做技术选型时，这样一个“十分钟就能算出来”的模型，比什么都不算要强得多。第 67 讲会用 DGX Spark 上的实测，把效率这个数校准到你自己的环境。

自有设备的算法。如果 GPU 是自己买的，可以把购置价格按使用年限摊到每小时，再加上电费和运维，折算成“每小时的成本”，代进同一个公式。

容易混淆的地方

1. **API 价格不等于成本**。API 的定价包含了利润、闲时的浪费、研发成本，以及市场竞争的策略。它和这里算的“理论成本”可以对比，但不必相等。
2. **效率因素影响很大**。60% 和 30% 的效率，成本差两倍。效率取决于推理引擎、算子实现、批大小，要靠实测确定。
3. **低价不代表低延迟**。最便宜的设置是把并发推到显存允许的最大值，但这时每个请求的 TPOT 也最长。成本要在满足延迟目标的前提下比较。

回顾

推理成本模型把速度和价格连起来：每百万 token 的成本 = 每小时租金 / 每小时处理的 token 数。输入 token 由 prefill 处理，计算受限，Llama-3-8B 在 H100 上约 $0.018 / 百万（示例租金 $2.5/小时、效率 60%）；输出 token 由 decode 处理，带宽受限，成本取决于并发：并发 1 时约 $5.6，并发 128 时约 $0.11。所以输出远比输入贵，而提高并发是降成本最有效的手段。量化、GQA/MLA、前缀缓存等优化，都可以代进这个模型，算出它们能省多少。

到这里，推理的“账”都算完了。最后还有一个和速度、成本无关，却直接决定输出质量的环节：拿到下一个 token 的概率分布之后，到底选哪一个？下一讲：**解码策略**。

第73讲 解码策略

06 指数、对数与 softmax　07 概率与采样　24 自回归生成

第 9 篇的前几讲都在算速度和成本。这一讲回到推理流程的最后一步：模型给出了下一个 token 的概率分布，**到底选哪一个?**

第 24 讲介绍过两种最基本的做法：贪心（每次选概率最大的）和采样（按概率随机抽），以及用温度调整分布的尖平。实际的推理服务里，还有几个几乎每个 API 都提供的参数：**Top-k**、**Top-p**、**Min-p**。它们都在做同一件事：**在随机抽样之前，先把那些不太可能、一旦抽中就会“跑偏”的候选词去掉**。这一讲把它们放在一个例子里比较。

一个例子

假设模型看到“今天天气真”，给 12 个候选词的概率如下（示意的数字）：

好	不错	晴朗	热	冷	糟糕	适合	闷	美	差	蓝	猫
0.321	0.215	0.144	0.097	0.072	0.043	0.036	0.026	0.024	0.016	0.006	0.001

前几个词都很合理；“适合”“美”勉强说得通，但可能要接着补全才通顺（“今天天气真适合出门”）；“蓝”“猫”就很奇怪了。真实的模型面对的是十几万个词的词表，长长的“尾巴”上有成千上万个这样概率很小、但加起来不可忽略的词。

贪心每次只选“好”，稳定，但单调，长文本里还容易陷入重复（第 24 讲）。**直接采样**丰富多样，但尾巴上那些奇怪的词加起来也有一定的概率，生成一段长文本，总会在某一步抽中一个，然后后面就越说越偏。

温度：调整尖平

第 24 讲讲过，温度 T 把每个 logit 除以 T 再做 softmax。T = 0.5 时，“好”的概率从 0.321 升到 0.544，分布变尖；T = 1.5 时降到 0.235，分布变平，“猫”从 0.001 涨到 0.004。

温度改变的是概率的**大小**，但不会把任何一个词彻底去掉：只要 T 不是 0，“猫”的概率总大于 0。

Top-k：只留前 k 个

Top-k 很直接：只保留概率最高的 k 个词，其余的概率设为 0，再把留下的重新归一化。k = 3 时只剩“好、不错、晴朗”，概率变成 0.472、0.316、0.212。

它的问题是 k 是固定的。有时模型很确定，前 2 个词就占了 99% 的概率，k = 50 会留下很多垃圾；有时模型很不确定，合理的候选有几百个，k = 50 又切掉了很多好词。

Top-p：凑够概率就停

Top-p（也叫 nucleus sampling，核采样）换了一种切法：把词按概率从高到低排好，依次累加，**累计概率第一次达到 p 时就停**，只保留前面这些。

p = 0.8 时：0.321、0.536、0.680、0.777，加到第 5 个“冷”时是 0.849，超过了 0.8，于是保留前 5 个。它会自动适应：模型确定时，一两个词就凑够了 p，保留得少；模型不确定时，要很多词才凑够，保留得多。这比固定的 k 合理，所以 Top-p 是最常用的设置之一。

Min-p：按最大概率定门槛

Min-p 是近两年流行起来的另一种切法：去掉概率小于“最大概率 × min-p”的词。min-p = 0.1 时，门槛是 0.321 × 0.1 = 0.0321，保留了概率在 0.036 以上的 7 个词。

它的想法是：模型越自信（最大概率越高），门槛越高，砍掉的越多；模型越犹豫，门槛越低，保留的越多。和 Top-p 相比，它在高温度下更不容易放进奇怪的词。

这几种方法可以组合使用，常见的顺序是：先调温度，再依次用 Top-k、Top-p、Min-p 去掉候选，最后对剩下的重新归一化，随机抽一个。比如 T = 0.7 加上 Top-p = 0.9，保留 5 个词，“好”的概率是 0.458；按这个分布抽 1000 次，“好”被抽中 448 次，“不错”274 次，“晴朗”127 次，“热”86 次，“冷”65 次，奇怪的词一次也不会出现。

下面的演示可以自己调这四个参数。浅色条是原始概率，深色条是最终概率，被去掉的词会标出是被哪一步去掉的。点“抽 100 次”看随机抽样的结果。

温度、Top-k、Top-p：从分布里怎么选词

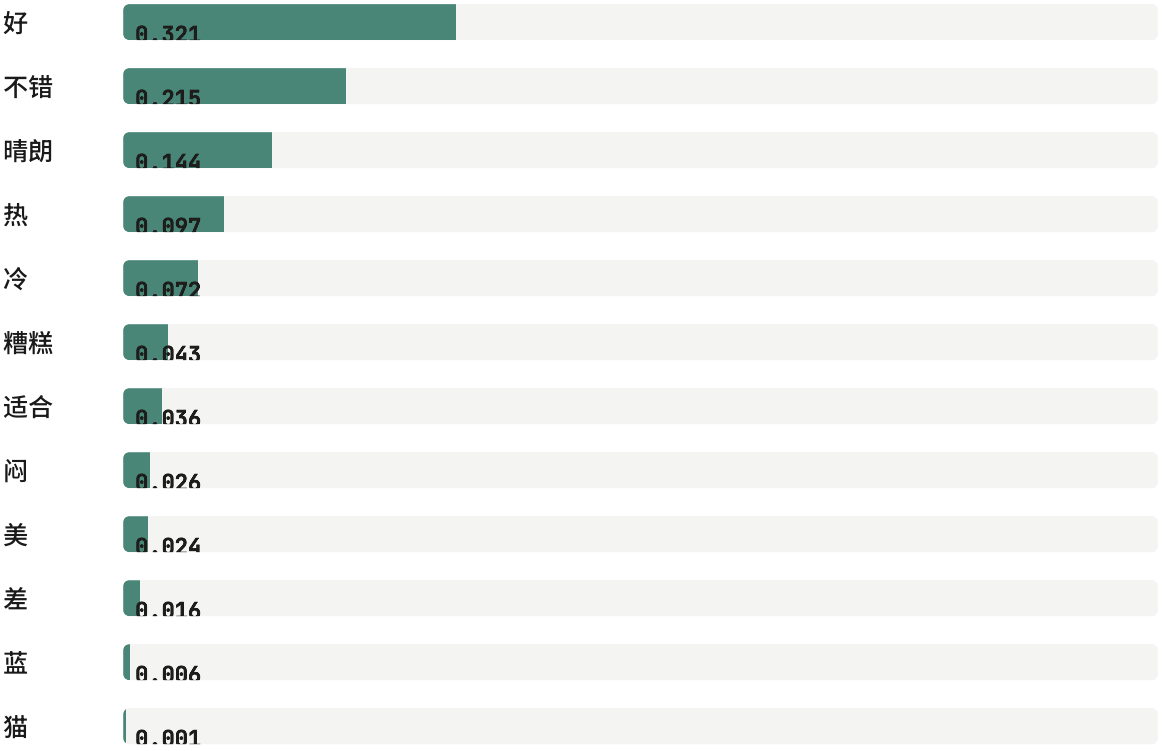
⇆ 对照

提示：“今天天气真 ___”

温度 T = **1**　　Top-k = **不限**

Top-p = **不限**　　Min-p = **不限**

候选词　浅色：原始概率　深色：最终概率



抽 100 次　　恢复默认

保留 12 个候选词。

温度改变分布的尖平；Top-k 只留概率最高的 k 个；Top-p 从高到低累加，凑够 p 就停；Min-p 去掉概率不到最高者 × min-p 的词。剩下的重新归一化，再随机抽。

图 73-1　解码策略。试试：Top-k 调到 3；Top-p 调到 0.8；温度调到 2 再加 Top-p 0.9；Min-p 调到 0.3。被去掉的词变灰，剩下的重新归一化。

要点

温度改变分布的尖平，不去掉任何词；Top-k 只留前 k 个；Top-p 从高到低累加到 p 为止；Min-p 去掉低于“最大概率 × min-p”的词。它们都是在随机抽样之前砍掉长尾，在多样性和可靠性之间找平衡。

怎么选

没有一组“最好”的参数，要看任务：

- **需要确定、准确的任务**（数学、代码、信息抽取、事实问答）：温度低（0 到 0.3），或者直接贪心。同样的输入最好得到同样的输出。
- **需要多样性的任务**（写作、头脑风暴、对话）：温度 0.7 到 1 左右，配合 Top-p 0.9 或 Min-p 0.05 之类，既有变化又不跑偏。
- **推理模型**：很多推理模型的官方建议是温度 0.6 左右、Top-p 0.95，而不是贪心。贪心在长长的思考过程里容易陷入重复的循环。

此外常见的还有**重复惩罚**（降低已经出现过的词的 logit，避免车轱辘话）和**随机种子**（固定种子，让采样的结果可以复现）。

用代码复现

下面的代码按“温度 → Top-k → Top-p → Min-p → 重新归一化”的顺序实现了解码策略，打印各种设置下保留的词和概率，最后按“T = 0.7 + Top-p = 0.9”抽 1000 次。

```python
import numpy as np

# “今天天气真 ___”：一个示意的模型给 12 个候选词的打分（logits）
words = ["好", "不错", "晴朗", "热", "冷", "糟糕", "适合", "闷", "美", "差", "蓝",
"猫"]
logits = np.array([3.0, 2.6, 2.2, 1.8, 1.5, 1.0, 0.8, 0.5, 0.4, 0.0, -1.0, -3.0])

def decode_probs(logits, T=1.0, top_k=0, top_p=1.0, min_p=0.0):
    """按常见的顺序：温度 → Top-k → Top-p → Min-p → 重新归一化。返回最终的概率。"""
    z = logits / T                                          # 1. 温度（第 24 讲）
    p = np.exp(z - z.max()); p /= p.sum()
    keep = np.ones(len(p), bool)
    order = np.argsort(-p)
    if top_k:                                               # 2. 只留概率最高的 k 个
        keep[order[top_k:]] = False
    if top_p < 1.0:                                         # 3. 从高到低累加，够 p 就
停
        cum = np.cumsum(p[order])
        cut = np.searchsorted(cum, top_p) + 1               # 保留到累计概率第一次 ≥ p
的那个词
        keep[order[cut:]] = False
    if min_p > 0:                                           # 4. 概率低于“最大概率 ×
min_p”的去掉
        keep &= p >= min_p * p.max()
    q = np.where(keep, p, 0); return q / q.sum()

def show(name, q):
    kept = [(words[i], round(float(q[i]), 3)) for i in np.argsort(-q) if q[i] > 0]
    print(f"{name}：保留 {len(kept)} 个", kept)

base = decode_probs(logits)
print("原始分布：", [(w, round(float(x), 3)) for w, x in zip(words, base)])
print("贪心：", words[int(base.argmax())])
show("T = 0.5", decode_probs(logits, T=0.5))
show("T = 1.5", decode_probs(logits, T=1.5))
show("Top-k = 3", decode_probs(logits, top_k=3))
show("Top-p = 0.8", decode_probs(logits, top_p=0.8))
show("Min-p = 0.1", decode_probs(logits, min_p=0.1))
show("T = 0.7 + Top-p = 0.9", decode_probs(logits, T=0.7, top_p=0.9))

# 按最后一种设置抽 1000 次（固定种子），看每个词被抽中的次数
q = decode_probs(logits, T=0.7, top_p=0.9)
draws = np.random.default_rng(65).choice(len(words), size=1000, p=q)
print("抽 1000 次：", {words[i]: int(c) for i, c in zip(*np.unique(draws,
return_counts=True))})
```

输出：

```
原始分布： [('好', 0.321), ('不错', 0.215), ('晴朗', 0.144), ('热', 0.097), ('冷',
0.072), ('糟糕', 0.043), ('适合', 0.036), ('闷', 0.026), ('美', 0.024), ('差',
0.016), ('蓝', 0.006), ('猫', 0.001)]
贪心： 好
T = 0.5：保留 12 个 [('好', 0.544), ('不错', 0.245), ('晴朗', 0.11), ('热', 0.049),
('冷', 0.027), ('糟糕', 0.01), ('适合', 0.007), ('闷', 0.004), ('美', 0.003), ('差',
0.001), ('蓝', 0.0), ('猫', 0.0)]
T = 1.5：保留 12 个 [('好', 0.235), ('不错', 0.18), ('晴朗', 0.138), ('热', 0.106),
('冷', 0.087), ('糟糕', 0.062), ('适合', 0.054), ('闷', 0.044), ('美', 0.042), ('差',
0.032), ('蓝', 0.016), ('猫', 0.004)]
Top-k = 3：保留 3 个 [('好', 0.472), ('不错', 0.316), ('晴朗', 0.212)]
Top-p = 0.8：保留 5 个 [('好', 0.378), ('不错', 0.254), ('晴朗', 0.17), ('热', 0.114),
('冷', 0.084)]
Min-p = 0.1：保留 7 个 [('好', 0.346), ('不错', 0.232), ('晴朗', 0.155), ('热',
0.104), ('冷', 0.077), ('糟糕', 0.047), ('适合', 0.038)]
T = 0.7 + Top-p = 0.9：保留 5 个 [('好', 0.458), ('不错', 0.259), ('晴朗', 0.146),
('热', 0.083), ('冷', 0.054)]
抽 1000 次： {'好': 448, '不错': 274, '晴朗': 127, '热': 86, '冷': 65}
```

第一行是原始分布，第二行是贪心。之后每一行是一种设置保留的词和重新归一化后的概率，和正文一致。最后一行是抽样的结果。

到了大模型里

采样也在 GPU 上做。每一步，每个请求都有一个 128256 维的 logits 向量（第 33 讲）。Top-p 需要排序，对十几万个数排序并不便宜；一个批里每个请求的参数还可能都不一样（有的贪心，有的 T = 0.8、Top-p = 0.95）。推理引擎会用专门的 GPU 算子来做这一步，比如不完整排序、只找出需要的前若干个，避免把 logits 搬回 CPU。

采样影响可复现性。同样的输入、同样的参数，采样的结果取决于随机数。即使固定了种子，批里的请求组合、GPU 的并行计算顺序带来的微小浮点差异，也可能让结果不完全一样。需要严格复现时，用贪心并固定推理配置。

投机解码要保持分布不变。第 96 讲的投机解码，用一个小模型先猜几个 token，再让大模型一次验证。为了保证最终的输出和“直接用大模型采样”完全同分布，验证时要用一个专门设计的接受规则，这和这一讲的采样参数紧密相关。

容易混淆的地方

1. **温度为 0 等于贪心**。不是真的除以 0，推理引擎会直接取概率最大的词（第 24 讲）。
2. **Top-p 不是“概率大于 p 的词”**。它是“从高到低累加，凑够 p”。p = 0.9 通常会保留很多个词，而不是只保留概率大于 0.9 的那一个。
3. **解码策略不改变模型**。它只决定怎么从模型给出的分布里选词。模型本身的知识和能力没有变，但不合适的解码参数确实会让同一个模型看起来“变笨”或“胡言乱语”。

回顾

模型给出下一个 token 的分布之后，还要决定怎么选。贪心稳定但单调；直接采样多样，但长尾上的奇怪词总会被抽中。温度调整分布的尖平；Top-k 只留概率最高的 k 个；Top-p 从高到低累加到 p 为止，能适应模型的确定程度；Min-p 按最大概率定一个门槛。它们组合起来，先砍掉长尾，再随机抽样，在多样性和可靠性之间取得平衡。准确性要求高的任务用低温或贪心，创作类任务用较高的温度配合 Top-p 或 Min-p。

第 9 篇到这里结束：我们知道了一个请求怎么被处理、瓶颈在哪、显存怎么分配、成本怎么算、最后怎么选词。可这些都是“一个请求”或“一批固定的请求”的视角。真实的服务里，请求随时到达、长短不一、随时结束，GPU 怎样才能一刻不停地满负荷工作？第 10 篇从这个问题开始：Continuous Batching。

AI INFRA · HANDBOOK

第 十 篇

推理引擎与调度

收录 9 讲 · 第 74–82 讲

第74讲 Continuous Batching

24 自回归生成　68 Prefill 与 Decode　70 TTFT、TPOT 与吞吐

第 70 讲说明了批处理的威力：decode 每一步都要把权重读一遍，让很多请求一起 decode，权重的读取就被分摊了，吞吐能提高几十倍。可那一讲有一个理想化的假设：b 个请求“一起开始、一起进行”。真实的推理服务不是这样的：

- 请求是**陆续到达**的，每秒来几个、几十个，时间点随机。
- 每个请求要生成的 token 数**事先不知道**，有的回答一句话就结束，有的要写几千字。模型生成了结束标记，这个请求才算完。

在这样的条件下，怎么把请求组成批？这一讲比较两种做法：传统的**静态批处理**，和今天所有主流推理引擎都在用的**连续批处理**（continuous batching）。后者是 vLLM、SGLang、TensorRT-LLM 等推理引擎能达到高吞吐的第一个关键。

静态批处理：一批人一起进、一起出

最直接的想法是照搬训练时的做法：凑够一批请求，一起送进模型，一步步 decode，直到这一批**全部**结束，再处理下一批。这叫**静态批处理**（static batching），也叫请求级的批处理。

问题出在“全部结束”上。同一批里，有的请求只要生成 2 个 token，有的要 8 个。短的早早结束了，它占的那个位置却不能给别人用，只能空着，一直等到最长的那个结束。新到达的请求，哪怕 GPU 上有空位，也要排队等这一批全部完成才能进来。

我们用一个小例子把它画出来。10 个请求，各自的到达时间和要生成的 token 数如下（时间以“步”为单位，每一步，批里的每个请求生成 1 个 token）：

请求	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
第几步到达	0	0	0	0	2	3	4	5	6	8
要生成的 token 数	8	3	5	2	6	4	7	2	5	3

GPU 一次最多处理 4 个请求（4 个“槽位”）。静态批处理：第 0 步，R1–R4 组成第一批。R4 在第 2 步就结束了，R2 在第 3 步，R3 在第 5 步，但 R1 要到第 8 步才结束，这期间槽位一个个空下来。R5 在第 2 步就到了，却要等到第 8 步才能开始。一共要 **20 步**才全部完成，平均每个请求从到达到完成要 **7.9 步**，槽位的利用率只有 **56%**：将近一半的时间，GPU 上的位置是空的。

连续批处理：随走随补

连续批处理把调度的粒度从“一批请求”缩小到“一步迭代”：**每生成一个 token（每一步迭代），都重新检查一次这一批的组成**。哪个请求生成了结束标记，就立刻把它移出去；空出来的位置，立刻从等待队列里补一个新请求进来。批的组成在不停地流动，所以也叫**迭代级调度**（iteration-level scheduling）。这个想法来自 2022 年的 Orca 论文。

同样的 10 个请求、4 个槽位：R4 在第 2 步结束，R5 正好到达，立刻补上；R2 在第 3 步结束，R6 补上……槽位几乎从不空着。一共只要 **13 步**，平均延迟 **5.0 步**，槽位利用率 **87%**。而理论上的极限是：45 个 token、每步最多 4 个，至少要 12 步。连续批处理已经很接近它了。

下面的甘特图把两种做法画在一起。每一行是一个槽位，每个色块是一个请求从开始到完成的时间，灰色是空着的槽位。拖动槽位数，看两种做法的差距怎么变。

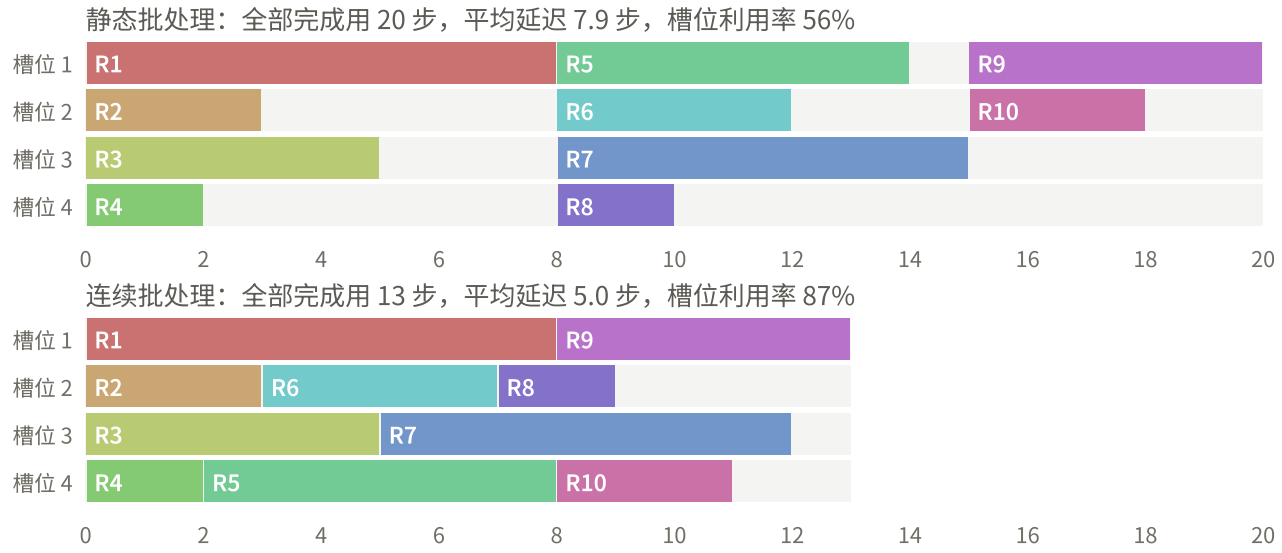
同样 10 个请求，两种批处理

⇆ 对照

槽位数 = 4

请求：

- R1 — 第 0 步到，生成 8 个
- R2 — 第 0 步到，生成 3 个
- R3 — 第 0 步到，生成 5 个
- R4 — 第 0 步到，生成 2 个
- R5 — 第 2 步到，生成 6 个
- R6 — 第 3 步到，生成 4 个
- R7 — 第 4 步到，生成 7 个
- R8 — 第 5 步到，生成 2 个
- R9 — 第 6 步到，生成 5 个
- R10 — 第 8 步到，生成 3 个



灰色是空着的槽位。静态批处理里，一批中短的请求早早结束，槽位却要空等最长的那个；后来的请求也要等整批结束才能进来。连续批处理每一步都把空槽位补上。

图 74-1　静态批处理（上）和连续批处理（下）。静态批处理里灰色的空档很多：短请求结束后，槽位空等这一批里最长的那个，新请求也进不来。连续批处理每一步都把空槽位补上。

要点

静态批处理以“一批请求”为单位，整批结束才换下一批，短请求结束后的槽位只能空等。连续批处理以“一步迭代”为单位，每一步都把结束的请求移出、把等待的请求补进来，GPU 的批几乎一直是满的。

对延迟和吞吐的影响

连续批处理同时改善了两件事：

吞吐更高。槽位利用率从 56% 提高到 87%，同样的时间里生成的 token 多了 50% 以上。真实的负载里，请求的输出长度差别比这个例子大得多（几个 token 到几千个 token），静态批处理的浪费更严重。Orca 论文和后来的 vLLM 论文都报告，连续批处理带来的吞吐提升可以达到好几倍甚至一个数量级。

延迟更低。新请求不必等前一批结束就能开始，平均延迟从 7.9 步降到 5.0 步，后到的请求受益最大：R9 在静态批处理里第 20 步才完成（到达后等了 14 步），连续批处理里第 13 步就完成了。

一个新问题：prefill 怎么插进来

例子里我们假设新请求一进来就和别人一样“每步生成 1 个 token”。实际上，新请求要先做 prefill（第 68 讲）：把整段提示送进去，才能生成第一个 token。

在连续批处理里，新请求加入的那一步，这一步的批就同时包含了两种工作：老请求的 decode（每个 1 个 token），和新请求的 prefill（几百甚至几千个 token）。这一步的计算量一下子变大了很多，耗时也明显变长，所有正在 decode 的请求，这一步的 token 都会被“拖慢”。第 70 讲提过的 TPOT 抖动，正是从这里来的。怎么把 prefill 和 decode 安排得更平滑，是第 77 讲 Chunked Prefill 的主题。

实现上的几个要点

把所有 token 拼成一个大批。批里各个请求的长度都不一样：有的在 prefill（几千个 token），有的在 decode（1 个 token）。如果像训练那样把它们补齐（padding）成同样长度的矩阵，会浪费大量计算。推理引擎的做法是把这一步所有请求的所有 token **首尾相接拼成一个长序列**：线性层（第 13 讲）本来就是对每个 token 独立计算的，直接对这个 [总 token 数, d] 的矩阵做一次大矩阵乘法即可；只有注意力需要区分“哪些 token 属于同一个请求”，由专门的注意力算子根据每个请求的起止位置分别计算。

每个请求的 KV Cache 各不相同。请求随时进出，每个请求的 KV Cache 长度不同、增长的速度不同、释放的时间也不同。如果还像普通的张量那样，给每个请求预留一大块连续的显存，很快就会变得支离破碎。这正是下一讲 PagedAttention 要解决的问题。

每一步都要做调度决策。每一步开始前，调度器要决定：哪些请求继续 decode，哪些新请求可以加入（显存够不够放它的 KV Cache？），这一步总共处理多少个 token。这个决策每秒要做几十到几百次，所以调度本身要足够快，否则 CPU 端的调度会成为瓶颈（第 89 讲）。

用代码复现

下面的代码模拟了这 10 个请求在 4 个槽位上的两种批处理方式，统计全部完成的步数、平均延迟和槽位利用率。

```
# 10 个请求：(到达的步数, 要生成的 token 数)。每一步，每个正在运行的请求生成 1 个 token。
REQS = [(0, 8), (0, 3), (0, 5), (0, 2), (2, 6), (3, 4), (4, 7), (5, 2), (6, 5), (8, 3)]

def simulate(slots, continuous):
    waiting, running, step = list(range(len(REQS))), [], 0
    left = {i: n for i, (_, n) in enumerate(REQS)}
    finish, busy = {}, 0
    while len(finish) < len(REQS):
        arrived = [i for i in waiting if REQS[i][0] <= step]
        if continuous or not running:                       # 连续批处理：每步都补；静态：整批跑完才补
            while arrived and len(running) < slots:
                i = arrived.pop(0); waiting.remove(i); running.append(i)
        for i in running:                                   # 这一步：每个在跑的请求生成 1 个 token
            left[i] -= 1
        busy += len(running)
        step += 1
        for i in [i for i in running if left[i] == 0]:
            running.remove(i); finish[i] = step             # 第 step 步结束时完成
    lat = [finish[i] - REQS[i][0] for i in range(len(REQS))]
    return step, sum(lat) / len(lat), busy / (slots * step), finish

for slots in [4]:
    for cont in [False, True]:
        steps, avg, util, fin = simulate(slots, cont)
        name = "连续批处理" if cont else "静态批处理"
        print(f"{name} ({slots} 个槽位)：全部完成用 {steps} 步，平均延迟 {avg:.1f} 步，槽位利用率 {util:.0%}")
        print("   每个请求完成于第几步：", [fin[i] for i in range(len(REQS))])
total_tokens = sum(n for _, n in REQS)
print(f"一共要生成 {total_tokens} 个 token；4 个槽位每步最多 4 个，理论上至少要 {-(-total_tokens // 4)} 步")
```

输出：

```
静态批处理（4 个槽位）：全部完成用 20 步，平均延迟 7.9 步，槽位利用率 56%
    每个请求完成于第几步： [8, 3, 5, 2, 14, 12, 15, 10, 20, 18]
连续批处理（4 个槽位）：全部完成用 13 步，平均延迟 5.0 步，槽位利用率 87%
    每个请求完成于第几步： [8, 3, 5, 2, 8, 7, 12, 9, 13, 11]
一共要生成 45 个 token；4 个槽位每步最多 4 个，理论上至少要 12 步
```

静态批处理 20 步、平均延迟 7.9 步、利用率 56%；连续批处理 13 步、5.0 步、87%。每个请求完成的步数也列了出来：前 4 个请求两种方式一样，后面的请求在连续批处理里都早得多。最后一行是理论下限 12 步。

到了大模型里

所有主流推理引擎都默认使用连续批处理。vLLM、SGLang、TensorRT-LLM、LMDeploy 里，它几乎是不需要配置的基础能力。相关的参数通常是“一步最多处理多少个请求”（比如 vLLM 的 `max_num_seqs`）和“一步最多处理多少个 token”（`max_num_batched_tokens`），它们决定了批的上限。

批的上限由显存决定。连续批处理想让批尽量满，但每多一个请求，就多一份 KV Cache（第 71 讲）。实际能同时运行多少个请求，取决于 KV Cache 的池子还剩多少空间。显存不够时该怎么办，是第 78 讲调度与抢占的话题。

和流式输出天然契合。连续批处理每一步都为每个请求产出一个 token，可以立刻通过流式接口（比如 SSE）推给用户，用户看到的是一个字一个字地蹦出来。

容易混淆的地方

1. **连续批处理不改变模型的计算**。每个请求得到的结果，和它单独运行时完全一样（不考虑浮点误差）。它改变的只是“哪些请求在同一步里一起算”。
2. **“动态批处理”有时指另一件事**。有些服务框架里的 dynamic batching，指的是“等一小段时间，凑够一批再一起发给模型”，仍然是请求级的批处理，和这里的迭代级调度不同。
3. **批越大不一定越好**。第 70 讲说过，批越大，每个请求的 TPOT 越长。连续批处理让批“尽量满”，但“满”的上限要按延迟目标和显存来设。

回顾

真实服务里请求陆续到达、输出长度各不相同。静态批处理以一批请求为单位，整批结束才换下一批，短请求结束后槽位只能空等，新请求也要排队：10 个请求、4 个槽位的例子里，要 20 步、利用率 56%。连续批处理把调度粒度缩小到每一步迭代，结束的请求立刻移出，等待的请求立刻补进来，同样的例子只要 13 步、利用率 87%，接近 12 步的理论下限。实现上，它把一步里所有请求的 token 拼成一个大批，只在注意力里区分请求；新请求的 prefill 会插进 decode 之间，带来 TPOT 的抖动。

连续批处理让请求随时进出，每个请求的 KV Cache 也随时增长、随时释放。如果给每个请求预留一整块“最大长度”的连续显存，会浪费多少？有没有更好的办法？下一讲：**PagedAttention**。

第75讲 PagedAttention

69 KV Cache　71 显存账本　74 Continuous Batching

第 71 讲的显存账本算出：Llama-3-8B 在一张 80 GB 的卡上，留给 KV Cache 的空间约 49 GiB，能放约 40 万个 token。可是“能放 40 万个 token”和“能同时服务多少个请求”之间，还隔着一个问题：**这 40 万个 token 的空间，怎么分给一个个请求?**

连续批处理（第 74 讲）让请求随时进出，每个请求的 KV Cache 一边生成一边增长，长度事先不知道，结束的时间也不知道。如果分配的方式不好，大量的显存会被浪费在“预留了但没用上”的地方，能同时服务的请求数就会远低于账本上的数字。

2023 年 vLLM 提出的 PagedAttention 解决了这个问题。它借用了操作系统管理内存的经典思路：**分页**。这一讲先看传统做法浪费在哪里，再看分页是怎么做的、为什么几乎不浪费，以及它顺带带来的“共享”能力。它是今天几乎所有推理引擎管理 KV Cache 的基础。

传统做法：给每个请求预留一整块

最直接的做法，是像普通的张量那样，给每个请求分配一块**连续**的显存来存它的 KV Cache。问题是：请求刚开始时，不知道它最终会有多长。为了保险，只能按**最大可能的长度**预留，比如模型支持的最大上下文 4096 个 token。

这会带来三种浪费：

1. **预留了没用上**：一个请求最终只有 1500 个 token，却占了 4096 个 token 的空间，剩下的 2596 个位置一直空着，别的请求也不能用。这叫**内部碎片**。
2. **还没生成的部分**：即使事先知道最终长度、按最终长度预留，在请求生成到一半时，后一半的空间也已经被占着，暂时空闲。
3. **外部碎片**：请求来来去去，释放出来的空间大小不一、位置分散。新请求需要一整块连续的 4096 个 token，即使空闲的总量足够，也可能找不到一块足够大的连续空间。

下面的代码模拟了一个快照：64 个请求正在运行，提示长度在 200 到 2000 之间，最终输出 50 到 1000 个 token，各自生成到了一半左右。它们实际一共有 86151 个 token 的 KV Cache。

- **按最大长度（4096）连续预留**：占用 262144 个 token 的空间，利用率只有 **32.9%**。
- **按最终长度连续预留**（理想情况，实际做不到）：占用 100758 个，利用率 85.5%。

- **分页，每块 16 个 token**：占用 86560 个，利用率 **99.5%**。

vLLM 的论文报告，在它之前的推理系统里，KV Cache 的显存有 60% 到 80% 被浪费掉了，和这里的模拟是一个量级。

分页：像操作系统管理内存那样

操作系统早就遇到过同样的问题：很多程序同时运行，各自需要的内存大小不一、随时变化。它的解决办法是**分页**：把物理内存切成大小固定的小块（页），程序看到的是一段连续的“虚拟地址”，实际却可以分散在任意的物理页上，由一张**页表**记录“虚拟的第几页在物理的第几页”。

PagedAttention 照搬了这个想法：

- **切块**：把整个 KV Cache 池子切成大小固定的**块**（block），每块存固定个数 token 的 KV，vLLM 默认 16 个。推理引擎启动时，就把第 71 讲算出的 KV Cache 空间全部切成块，放进一个空闲列表。
- **按需分配**：请求需要存更多 token 时，才从空闲列表里取一个块，**不需要和前面的块相邻**。请求的最后一块写满了，再取下一块。
- **块表**：每个请求有一张**块表**（block table），记录它的第 0、1、2……个逻辑块分别放在哪个物理块。
- **用完归还**：请求结束，它的所有块还回空闲列表，立刻可以给别的请求用。

查找某个 token 的 KV 时，用块表换算一下：第 i 个 token 在逻辑上的第 i ÷ 16 块（取整），块内第 i mod 16 个位置；查块表就知道它在哪个物理块。比如一个请求的块表是 [7, 2, 9]：它的第 0 到 15 个 token 在物理块 7，第 16 到 31 个在物理块 2，第 40 个 token 在物理块 9 的第 8 个位置。

这样一来：

- **几乎没有内部碎片**：每个请求浪费的只是最后一块里还没写满的那几个位置，最多 15 个 token。
- **没有外部碎片**：所有块一样大，任何一个空闲块都能给任何请求用，不需要找连续的大空间。
- **不需要预知长度**：长度增长到哪，块就分配到哪。

下面的演示是一个缩小版的块池：24 个块，每块 4 个 token，5 个请求陆续到达。每个格子是一个物理块，字母表示它属于哪个请求，下面的小点表示块里装了几个 token。点一个请求，看它的块表。切换到“按最大长度连续预留”，看同样的请求会遇到什么问题。



图 75-1　KV Cache 块池（每块 4 个 token）。分页时，块按需分配、物理位置分散，5 个请求都能立刻开始；按最大长度连续预留时，每个请求一来就占 8 个相邻的块，后到的请求 D、E 只能等前面的请求结束。

在分页模式下，5 个请求都能在到达时立刻开始，块散落在池子的各个位置，利用率很高。切换到连续预留：每个请求一到就要占 8 个相邻的块（按最大长度 32 个 token），A、B、C 就把 24 个块全占满了，虽然它们实际只存了十几个 token；D 和 E 到达后只能排队，直到前面的请求结束、空出一整段相邻的块。

要点

PagedAttention 把 KV Cache 切成固定大小的块，按需分配，用块表记录每个请求的逻辑块对应哪个物理块。浪费只剩每个请求最后一块没写满的部分，也没有了“找不到连续空间”的问题，同样的显存能装下多得多的请求。

引擎底层是怎么实现的：启动、调度与计算三件事

很多初学者看到这里会有一个疑问：显存不是一块平整的物理芯片吗？软件层面上怎么把它切成块，又怎么在运行中精准调度呢？对应到推理引擎的底层实现，其实就是三件事：启动时的显存池化、调度时的块管理器，以及计算时专门定制的注意力算子。

1. 启动时：Profiling 与全局显存池化

大模型推理系统有一个铁律：**运行时绝不频繁向操作系统申请显存**。

在 GPU 上调用显存分配函数（如 `cudaMalloc`）不仅开销昂贵，每次可能耗费数毫秒甚至几十毫秒，而且频繁申请、释放不规则大小的显存，很快就会在 GPU 物理显存中制造出无法消除的碎片，最终在显存总量明明充足的情况下抛出 OOM（Out of Memory）异常。

所以，推理引擎在正式对外服务之前，会先做一次“摸底测量”（Profiling）：

引擎把模型本身的全部权重加载进 GPU 显存（例如 Llama-3-8B 的 FP16 权重约占 16 GiB）。接着，引擎使用一个模拟的最大批次输入跑一次前向传播，或者根据公式测算出模型运行时峰值激活值与临时缓冲区所需要的显存大小（例如约占 2 GiB）。

用整张显卡的物理总容量（例如 80 GiB）减去权重和激活值的占用，剩下的空间乘上一个安全利用率系数（通常是 `gpu_memory_utilization = 0.9`，保留 10% 作为应对突发开销的安全缓冲），剩下的全部显存一**次性**向系统申请为一个巨大的连续张量。这个大张量就是整个系统的**物理 KV 池**（Physical KV Pool）。

申请完成后，引擎在软件层面上把这个池子一次性划分成 N 个固定大小的物理块（例如每个块装 16 个 token 的 KV）。在引擎后续长达数天甚至数月的运行过程中，**再也不会向 CUDA 运行时申请任何新的 GPU 显存**。所有请求对 KV Cache 的申请与释放，完全是由引擎内部自己维护的“账本”在用户态完成。

2. 调度时：Block Manager 的状态机与具体流转

这个负责记账与分发物理块的调度中枢，叫做 **Block Manager（块管理器）**。

Block Manager 内部维护了三个核心数据结构：

- **空闲物理块列表（Free List）**：启动时把所有的物理块编号 `[0, 1, 2, ..., N-1]` 全部放进一个队列或链表中。
- **请求块表（Block Table）**：每一个正在运行的请求都有一张属于自己的块表，记录该请求的逻辑块映射到了哪些物理块（例如请求 A 的块表是 `[7, 2, 9]`）。
- **物理块引用计数（Reference Count）**：每个物理块记录当前有几个请求在读写它。

在整个请求的生命周期中，Block Manager 就像一个严谨的仓库管理员，执行以下四步流转：

第一步是**请求到达（Prefill 阶段）**。假设用户发来一段长度为 35 个 token 的提示词，块大小设为 16。请求需要 $\lceil 35 / 16 \rceil = 3$ 个块。Block Manager 立即从空闲列表的头部弹出 3 个空闲物理块（例如编号 7、2、9），写入该请求的块表中，并将这 3 个块的引用计数都设为 1。Prefill 算完后，这 35 个 token 的 KV 会分别存进物理块 7（装前 16 个）、物理块 2（装中间 16 个）和物理块 9（装最后 3 个，此时该块还剩 13 个空位）。

第二步是**逐步生成（Decode 阶段）**。模型每生成一个新 token，就需要把这个新 token 的 KV 存入缓存。Block Manager 检查该请求块表的最后一个物理块（物理块 9）：当前只存了 3 个 token，容量上限是 16，显然还有空余槽位！于是新 token 的 KV 直接追加写入物理块 9 的第 4 个位置，**不需要分配任何新块**。

第三步是**块写满与按需追加**。当模型自回归生成到第 14 个 token 时，物理块 9 的 16 个槽位全部被填满。到了生成第 15 个 token 的瞬间，Block Manager 发现最后一个块已满，便再次从空闲列表里弹出一个新的空闲物理块（例如编号 15），追加到该请求的块表尾部（块表变为 `[7, 2, 9, 15]`），新 token 写入物理块 15 的第 0 个槽位。

第四步是**请求结束与归还**。当模型输出了终止符 `<eos>` 或达到最大长度，请求宣告完成。Block Manager 遍历该请求的块表，把物理块 7、2、9、15 的引用计数各自减 1。因为没有其他请求共用它们，四个块的引用计数全部降为 0。Block Manager 随即把这 4 个编号重新放回空闲列表中。整个过程完全是 CPU 内存中的几个整数和列表操作，耗时不到一微秒。

3. 计算时：PagedAttention Kernel 到底怎么读分散内存

现在显存被切成了成千上万个散落在各处的物理块，问题来了：GPU 的注意力算子（Attention Kernel）原本是假设矩阵在显存里连续排列的，它怎么去读取这些东一块西一块的 KV 数据?

这正是需要专门编写 PagedAttention Kernel 的原因。

在普通的注意力算子中，KV Cache 在显存里像一张平整的长方形地毯，第 i 个 token 的内存地址只要用基地址加上 `i * 步长` 就能一步算出，所有线程按固定步长连续读取显存。但在 PagedAttention 里，物理显存是打散的，比如逻辑上的前 16 个 token 在物理地址 `0x1000`，接下来的 16 个 token 却跑到了 `0x8000`。

为了让 GPU 算子能正常计算，调度器在启动 GPU Kernel 时，除了传入当前新 token 的查询向量 Q，还会把调度层维护的 `block_tables`（一个二维整数张量，形状为 `[批大小, 最多块数]`）一起作为参数传入 GPU 显存。

在 GPU 内部，计算被分配给成千上万个线程。以 Decode 阶段计算注意力为例，一个 Warp（包含 32 个线程的线程束）负责处理一个注意力头：

1. 当线程需要去读取历史上的第 t 个 token 的 Key 向量时，它在硬件寄存器里执行快速整数换算：
   - 先算逻辑块号：`逻辑块 = t // 16`；
   - 查块表找出真实的物理块号：`物理块号 = block_table[请求 ID, 逻辑块]`；
   - 算块内相对偏移：`块内偏移 = t % 16`；
2. 拿到物理块号后，乘以一个物理块的字节大小，就能定位到该物理块在全局显存池中的基地址，再加上块内偏移，就精准找出了该 token 对应的显存物理位置，把向量读进寄存器与 Q 做点积。

看到这里你可能会问：每个 token 都要查表换算地址，GPU 读取不会变慢吗？

答案是：**块内连续保证了极高的硬件效率**。虽然块与块之间在物理显存上是不连续的，但在**每一个块的内部**，这 16 个 token 的数据是紧凑、连续存放的。GPU 的内存总线每次是以 32 字节或 128 字节的缓存行（Cache Line）为单位读取显存的。当同一个 Warp 内的线程读取同一个块内的相邻 token 时，依然能够触发硬件的**合并访存（Coalesced Memory Access）**，吞吐量几乎打满显存带宽。只有每隔 16 个 token 跨越边界时，才会发生一次块表查询跳转。

我们手算一下一个物理块到底有多大。以 Llama-3-8B 为例（32 层 Transformer，8 个 KV 头，每头维度 128，采用 FP16 精度占用 2 字节）： 跨所有层合计的一个 token 的 KV 大小为： 2(K 和 V) × 32(层) × 8(头) × 128(维) × 2(字节) = 131,072 字节 = 128 KiB。

如果一个物理块存 16 个 token，那么跨所有层合计的一个物理块大小正好是： 16 × 128 KiB = 2048 KiB = 2 MiB。

在 GPU 显存架构中，2 MiB 是一个非常干净的对齐单位（很多 GPU 的二级缓存行和内存分页映射刚好是 2 MiB 级别）。既保证了内部碎片极小（最多只浪费不到 2 MiB），又让硬件访存足够连续高效。

省下来的显存变成了什么

省下来的显存，直接变成了**更多的并发**。用第 71 讲的数字：KV Cache 池子能放约 40.4 万个 token。

- 按最大长度 4096 连续预留：每个请求占 4096 个，最多 **98 个**请求。

- 分页：按上面那个快照，平均每个请求实际占约 1352 个 token，能放约 299 个请求。

并发数是原来的三倍左右，第 70 讲说过，并发越高，权重的读取被分摊得越多，吞吐越高（直到 KV Cache 的读取成为主导），第 72 讲的输出成本也随之下降。这就是 vLLM 论文报告的吞吐提升的主要来源。

共享：同一个块，多个请求一起用

分页还带来了一个额外的好处：**一个物理块可以同时出现在好几个请求的块表里**。

一个常见的场景是**并行采样**：同一个提示，让模型生成 4 个不同的回答，让用户挑选，或者做第 49 讲的多数投票。4 个回答共用同一段提示，提示部分的 KV Cache 完全相同。用分页，只需要存一份提示的块，4 个请求的块表都指向它们；各自生成的部分再分配各自的块。提示 1000 个 token、每个回答 200 个：不共享要存 4800 个 token，共享只要 1840 个，省了 62%。

为了安全地共享，每个块记录有几个请求在用它（**引用计数**）。如果某个请求要往一个共享的块里写新的 token（比如提示的最后一块还没写满，各个回答的第一个 token 要写在这里），就先把这个块**复制**一份给自己，再写，这叫**写时复制**（copy-on-write），也是操作系统里的老办法。引用计数降到 0，块才真正还回空闲列表。

这种共享不局限于同一个请求的多个回答。不同的请求如果开头相同（比如同一个系统提示），也可以共享前面那些块，这就是下一讲的前缀缓存。

代价：注意力算子要改

分页不是没有代价的。原来一个请求的 KV Cache 在显存里是连续的一整块，注意力算子按顺序读就行；现在它们散落在各个物理块里，注意力算子要**一边查块表、一边去不同的地方读**。这正是 “PagedAttention” 这个名字的由来：它本质上是一个能按块表读取分散 KV Cache 的注意力算子。

查表和分散读取会带来一点额外开销，块越小，开销越大；块越大，最后一块的浪费越多，共享的粒度也越粗。16 个 token 是一个经验上的折中。今天的注意力算子库（比如 FlashInfer、FlashAttention 的分页版本）都原生支持块表，这点开销已经很小了。

用代码复现

下面的代码先演示用块表换算 token 的位置；再模拟 64 个请求的快照，比较三种分配方式的利用率；然后算出池子能容纳的请求数；最后算并行采样时共享提示块能省多少。

```python
import math

B = 16                                                  # 每个块存 16 个 token 的 KV（vLLM 的默认值）

# 1. 块表：逻辑上第 i 个 token → 物理上第几个块、块里第几个位置
block_table = [7, 2, 9]                                 # 这个请求的 3 个逻辑块，分别放在物理块 7、2、9
for i in [0, 15, 16, 40]:
    print(f"第 {i:2d} 个 token → 物理块 {block_table[i // B]}，块内第 {i % B} 个")

# 2. 某一时刻，64 个请求正在运行（快照统计：提示 200~2000，输出 50~1000，已生成到一半）
n = 64
MAX_LEN = 4096
used = 86151                                            # 64 个请求当前实际拥有的 token 总数
contig_max = n * MAX_LEN                                # 每个请求按最大长度预留一整块连续空间
contig_final = 100758                                   # 就算事先知道最终长度，也要按最终长度预留
paged = 86560                                           # 分页：按块分配，只有最后一块可能没装满

for name, alloc in [("按最大长度连续预留", contig_max), ("按最终长度连续预留（理想）", contig_final), ("分页（每块 16 个 token）", paged)]:
    print(f"{name}：占用 {alloc:,} 个 token 的空间，真正用上 {used:,}，利用率 {used / alloc:.1%}")

# 3. 49.3 GiB 的 KV Cache 池（第 71 讲，Llama-3-8B 每 token 128 KiB）能同时放多少个这样的请求
pool = int(49.3 * 2**30 / (128 * 1024))
avg_paged = paged / n
print(f"池子 {pool:,} 个 token：连续预留 {MAX_LEN} 时能放 {pool // MAX_LEN} 个请求；分页时平均每个请求 {avg_paged:,.0f}，能放约 {pool / avg_paged:.0f} 个")

# 4. 并行采样：同一个提示（1000 个 token）生成 4 个不同的回答（各 200 个 token）
p, o, k = 1000, 200, 4
no_share = k * (p + o)
share = math.ceil(p / B) * B + k * math.ceil(o / B) * B    # 提示的块共享，各自的回答单独分配
print(f"并行采样 4 个回答：不共享 {no_share:,} 个 token，共享提示的块 {share:,} 个，省 {1 - share / no_share:.0%}")
```

输出：

```
第   0 个 token → 物理块 7，块内第 0 个
第 15 个 token → 物理块 7，块内第 15 个
第 16 个 token → 物理块 2，块内第 0 个
第 40 个 token → 物理块 9，块内第 8 个
按最大长度连续预留：占用 262,144 个 token 的空间，真正用上 86,151，利用率 32.9%
按最终长度连续预留（理想）：占用 100,758 个 token 的空间，真正用上 86,151，利用率 85.5%
分页（每块 16 个 token）：占用 86,560 个 token 的空间，真正用上 86,151，利用率 99.5%
池子 403,865 个 token：连续预留 4096 时能放 98 个请求；分页时平均每个请求 1,352，能放约 299 个
并行采样 4 个回答：不共享 4,800 个 token，共享提示的块 1,840 个，省 62%
```

前四行是块表的换算。接下来三行是三种分配方式的利用率：32.9%、85.5%、99.5%。然后是池子的容量：连续预留 98 个请求，分页约 299 个。最后一行是并行采样：共享提示的块，省 62%。

到了大模型里

块的大小和形状。vLLM 的块大小默认是 16 个 token（参数 `block_size`）。一个块里存的是所有层的、这 16 个 token 的 K 和 V；实际实现里，每层的 KV Cache 是一个 [块数, 2, 16, KV 头数, 每头维度] 之类的大张量，块号就是第一维的下标。启动日志里的 “GPU blocks: xxxx”，就是第 71 讲那本账算出的块数。

块表随请求一起送进算子。每一步，调度器把这一步每个请求的块表整理成一个整数矩阵，和 token 一起送进模型；注意力算子根据它去读写 KV Cache。新 token 的 k、v 也按块表写到对应的物理位置。

分页是后面很多技术的基础。前缀缓存（第 76 讲）靠块的共享；抢占（第 78 讲）以块为单位换出、换回；PD 分离（第 14 篇）以块为单位在机器之间传输 KV Cache。

容易混淆的地方

1. **分页不减少 KV Cache 本身的大小**。每个 token 的 KV Cache 还是 128 KiB（Llama-3-8B），分页减少的是“预留了但没用上”的浪费。要减少每个 token 的大小，靠的是 GQA、MLA、KV Cache 量化。
2. **“虚拟”的连续只存在于块表里**。模型的计算逻辑没有任何变化，注意力还是在和这个请求的全部历史 token 做计算，只是读数据的地址要查表。

3. **共享的块是只读的**。一个块被多个请求引用时，谁也不能直接改它，要写就先复制。这保证了共享不会让一个请求影响另一个请求的结果。

回顾

KV Cache 随请求增长、随时释放，长度事先不知道。传统做法按最大长度给每个请求预留连续的显存，造成大量内部碎片、外部碎片和预留浪费，模拟的快照里利用率只有 32.9%。PagedAttention 借用操作系统的分页：把 KV Cache 切成固定大小的块（常用 16 个 token），按需分配，用块表记录逻辑块到物理块的映射，浪费只剩每个请求最后一块的空位，利用率达到 99.5%，同样的显存能装下约三倍的请求。在底层，引擎通过启动时的全局预分配池化彻底杜绝运行时显存碎片，调度器维护空闲块列表与引用计数，定制的算子则拿着块表在连续的块内高效读取分散显存。

块可以共享，就意味着：如果很多请求的开头相同，比如同一个系统提示、同一份长文档、同一段多轮对话的历史，它们的 KV Cache 只需要算一次、存一份。怎么找到这些可以共享的前缀？下一讲：**Prefix Cache**。

第76讲 Prefix Cache

27 因果掩码　69 KV Cache　75 PagedAttention

真实的推理请求里，有大量重复的开头：

- **系统提示**：一个产品里所有的对话，开头都是同一段几百到几千 token 的系统提示（“你是某某助手，请遵守以下规则……”）。
- **多轮对话**：第 2 轮的输入，是第 1 轮的全部内容加上新的问题；第 10 轮的输入，包含了前 9 轮的全部历史。每一轮都把越来越长的历史重新送一遍。
- **长文档问答**：用户上传一份几万 token 的文档，接连问十几个问题，每个问题前面都是同一份文档。
- **智能体（Agent）**：一个任务里反复调用模型，每次的输入都是前面所有步骤的记录，再加上一点新内容。

第 69 讲的 KV Cache 只在一个请求的内部起作用，请求结束就释放了。下一个请求的开头即使一模一样，也要把这几千个 token 重新 prefill 一遍。**前缀缓存**（Prefix Caching）要做的，就是把这些已经算过的前缀的 KV Cache 留下来，让后面开头相同的请求直接复用。上一讲的分页让这件事变得自然：KV Cache 本来就是以块为单位、可以被多个请求共享的。

为什么可以复用

第 27 讲证明过：因为因果掩码，每个 token 在每一层的 k、v，**只取决于它自己和它前面的 token**。两个请求只要前 n 个 token 完全相同，它们这 n 个 token 的 KV Cache 就完全相同，不管后面接的是什么。

所以，只要能快速判断“这个新请求的开头，有多长和以前算过的某个请求一样”，这一段就可以直接复用，不用再算。

给每个块一个指纹：块级链式哈希的底层实现

怎么快速判断“开头相同”？如果对整段提示词直接算一个哈希，只要后面追加了一个新字，整个哈希值就彻底变了，根本无法复用前面的公共前缀；如果给每个 token 单独建哈希表，几十万个 token 的索引开销又太沉重，而且 GPU 显存是以 16 个 token 的物理块为单位分配的，单个 token 的命中无法在底层单独分配半个物理块。

所以，最佳的粒度是**块级别**（Block-level）。vLLM 等推理引擎给每个写满的物理块计算一个哈希值作为它的唯一“指纹”。

为什么必须是链式哈希

看到这里你可能会想：直接把当前块里的 16 个 token 拼起来算一个哈希不就行了吗?

我们看一个具体的例子。假设有两个不同的请求：

- 请求 A：`“你是一个专业的法文翻译助手：[前16个词] 苹果含有丰富的维生素……[中间16个词]”`
- 请求 B：`“你是一个严肃的儿童营养学专家：[前16个词] 苹果含有丰富的维生素……[中间16个词]”`

这两个请求在第 1 块（中间 16 个词）的内容完全一模一样，都是“苹果含有丰富的维生素……”。如果只对当前块的内容算哈希，这两个请求在第 1 块算出的哈希值是完全相等的。

但是，请求 B 能直接拿请求 A 的第 1 块 KV Cache 来用吗？**绝对不能**。

因为在 Transformer 的多层自注意力计算中，第 1 块里的每个 token，都在每一层中回头看了第 0 块的所有内容！请求 A 的第 1 块 Key 和 Value 向量融合了“法文翻译助手”的上下文信息，而请求 B 的第 1 块则需要融合“儿童营养学专家”的上下文信息。它们的 KV 向量在数值上差得十万八千里。如果强行复用，模型输出的结果就会彻底发疯。

结论是：**因果注意力决定了任何一个块的 KV Cache，都由从第 0 个 token 到该块末尾的全部因果历史共同决定**。

因此，哈希必须是**链式**（Chained）的：

```
hash_0 = H(None, tokens[0:16], extra)
```

```
hash_1 = H(hash_0, tokens[16:32], extra)
```

```
hash_2 = H(hash_1, tokens[32:48], extra)
```

每一块的哈希输入，严格包含了**前一块的哈希输出**加上**当前块的 16 个 token**。

这样一来，每个块的哈希值在数学上就成了“从第 0 个 token 到当前块末尾”整段历史序列的全局唯一摘要。只要前面有任何一个 token 不同，或者顺序不同，后面的所有链式哈希就会全部不同。

公式里的 `extra` 是什么？它是环境元数据。即使输入 token 完全一样，如果请求 A 挂载了某个特定的微调适配器（LoRA），或者包含了不同的多模态图像特征，算出来的 KV 也是不同的。把 LoRA 标识符或环境盐值一起混入哈希计算，就能防止不同配置之间的缓存误命中。

调度器内部的查表与执行流转

在推理引擎的调度层内部，维护着一张全局哈希表：`全局映射表[块哈希] = 物理块编号`。

当一个新请求带着提示词到来时，调度器会严格执行以下五步：

1. **分块与前缀哈希计算**：调度器把请求的提示词按 16 个 token 一块切开（不足 16 个的尾部余数先不切），在 CPU 上迅速计算出一条链式哈希序列 `[hash_0, hash_1, hash_2, ...]`。这一步非常轻量，几十微秒就能算完。
2. **从头逐块查表，遇断则停**：调度器拿 `hash_0` 去全局哈希表查，如果命中了物理块 P_7，就继续查 `hash_1`……假设一直查到了 `hash_5`，但 `hash_6` 没有查到，**查找立即停止**。即使后面的 `hash_7` 在表里碰巧存在（比如一段通用的模版文字），也绝不能用，因为前面的因果链已经断了。
3. **物理块复用与引用加一**：命中的前 6 个物理块（包含前 $6 \times 16 = 96$ 个 token 的 KV）直接放进这个新请求的块表（`block_table`）里，这 6 个物理块的引用计数各加 1。
4. **部分 Prefill**：GPU 算力得到解放！那 96 个 token 的 Key 和 Value 向量已经在显存里躺着了，完全不需要重新计算。调度器只需要给未命中的剩余 token 分配新的物理块，并让 GPU 只对这部分新 token 进行 Prefill 计算。注意力算子拿到拼接好的块表，新 token 的 Q 向量直接去查新老物理块中的所有 K 和 V。
5. **新块登记入表**：未命中的部分计算完成后，其中新装满的物理块会继续沿着链式哈希公式算出它们自己的哈希值，登记进全局哈希表，留给以后的请求复用。

下面用一个缩小的例子看看（一个字当一个 token，每块 4 个 token）。4 个请求依次到达，共用同一段系统提示：

- **请求 1**：缓存是空的，31 个 token 全部要算，算完后留下 7 个整块。
- **请求 2**：系统提示加“问：”正好是 24 个字，6 个整块全部命中，只需要算剩下的 8 个 token。

- **请求 3**：多轮对话的第 2 轮，开头是请求 1 的全部内容。请求 1 的 7 个整块全部命中，28 个 token 不用再算；请求 1 最后那 3 个字所在的块没装满，没有被缓存，要重新算。
- **请求 4**：系统提示只把第一个字“你”换成了“您”。第 1 块的指纹变了，因为每块的指纹包含了前面的全部内容，**从第 1 块开始往后全都不命中**，31 个 token 全部重算。

下面的演示就是这个例子。绿色是命中、不用再算的块，琥珀色是要算的块，灰色是最后没装满的块。试着改一改系统提示里的任何一个字，看看命中怎么变。



图 76-1　前缀缓存的命中。请求 2、3 复用了前面请求留下的块；请求 4 只改了第一个字，后面的块就一个也不命中。改一改系统提示，看命中的变化。

要点

前缀缓存把算过的 KV Cache 块留下来，按“包含前面全部内容”的指纹索引。新请求从头逐块查，命中的块直接复用、不再 prefill。只要前面有一个 token 不同，从那里开始往后都不会命中。

缓存的生命周期：物理块如何跨请求存活

很多初学者看到这里会有一个根本性的疑问：**一个请求执行完毕、客户端连接都断开了，它的物理块为什么还能留在 GPU 显存里给下一个请求用？显存难道不回收吗？**

这正是前缀缓存设计的精巧之处。在引擎内部，一个物理块有三种不同的生存状态：

1. **活跃状态（Active）**：某个正在执行的请求正在读取或写入它，物理块的引用计数 `ref_count > 0`。这个块绝对受保护，不可被修改，更不可被回收。
2. **空闲保留状态（Inactive / Cached）**：当请求完成退出时，调度器遍历该请求的块表，把每个物理块的引用计数减 1。当引用计数降到 0 时，**引擎绝对不会调用 CUDA 函数释放显存，也不会清空显存里的数据！** 引擎只是把这个物理块放进一个由双向链表维护的**空闲 LRU 队列**（最近最少使用队列）中。此时，它在全局哈希表中的映射关系依然完好保留。
3. **被驱逐淘汰（Evicted）**：如果一段时间内没有新请求使用它，而且系统的物理显存完全耗尽、必须分配全新物理块时，调度器才会从 LRU 队列的头部（最久没被访问过的块）挑选牺牲者。此时才把它的哈希映射从全局哈希表中抹去，把它当作全新的空白块分配给急需显存的新请求。

也就是说，**只要显存还有空余，算过的 KV 就永远留在 GPU 显存里**。新请求到来时，如果算出的哈希命中了这些 `ref_count == 0` 的块，调度器就把它从 LRU 队列里“捞出来”，引用计数重新加 1，零成本实现了跨请求的“唤醒”。

而且淘汰总是自底向上进行的：优先淘汰前缀链条最末端的“叶子节点”。因为一旦中间某个块被淘汰，因果链断开，它后面的所有块即使还在显存里，也再也不可能被新请求命中了。

几个必须讲透的底层细节

在工程实践中，有几个技术细节至关重要，也是写好前缀缓存系统必须知道的边界：

1. 为什么未装满的末尾块绝不缓存

如果一个块只能装 16 个 token，而请求的提示词末尾只剩 5 个 token，这个未装满的块绝对不能登记到全局哈希表中。

因为这个块处于未定型状态，后续进入自回归生成阶段时，模型生成的新 token 还要继续往这个块的剩余槽位里写入。如果把它提前缓存并共享给其他请求，其他请求并发写入时就会产生严重的数据覆盖冲突。因此，前缀缓存的命中粒度永远是 16 个 token 的整数倍。

2. 整段提示完全命中时，为什么最后一个 token 仍然必须重算

假设一个用户发来一段长达 2000 个 token 的文本，碰巧前 125 个物理块在显存里全部命中（$125 \times 16 = 2000$）。引擎是不是可以 100% 跳过 Prefill 阶段，直接开始输出回答?

答案是：**不能！提示词的最后一个 token 必须单步执行一次前向计算**。

这是很多教程没有讲透的底层原理：生成第一个回答 token 所需的概率分布（Logits），来自于提示词的**最后一个 token 经过最后一层 Transformer 后输出的隐藏状态**（Hidden State），再与词表头投影矩阵（LM Head）相乘得到。

而 KV Cache 里面存储的，仅仅是自注意力层内部的 Key 向量和 Value 向量，**压根没有保存最后一层的输出隐藏状态或 Logits**！

如果把整段提示词完全跳过不计算，模型手里根本没有最后一个词的 Logits，就无法通过采样得到第一个生成的词。因此，在底层实现中，前缀缓存最多只能复用前 `prompt_len - 1` 个 token 的 KV，最后一个 token 必须单步走一遍前向计算（以这个 token 为 Q，去读前面命中缓存的所有 K 和 V），从而拿到它的输出 Logits 来采样出第一个回答 token。

3. 写时复制（Copy-on-Write）

当多个请求共享了一段前缀的物理块后，各自开始独立生成回答。由于各个回答的内容不同，它们各自生成的后续 token 必须写入各自私有的物理块中。如果前缀共享部分的最后一个块刚好未满且需要被写入，引擎必须触发写时复制，先为该请求单独复制一份物理块，再往新块里写入数据，确保原本只读共享的前缀物理块不被破坏。

能省多少

命中的部分不用 prefill，直接省下的是计算和 TTFT。拿 Llama-3-8B、一张 H100（第 68 讲的估算）算一个典型的情况：系统提示 2000 个 token，用户的问题 100 个 token。

- **不命中**：prefill 2100 个 token，约 31.9 ms（计算受限）。
- **命中系统提示**：只需要 prefill 100 个 token，约 4.8 ms（此时已经变成了读一遍权重的时间）。

TTFT 降到了约六分之一。多轮对话和长文档问答的收益更大：第 10 轮对话时，前面 9 轮的几千个 token 全部命中，只需要算最新的那几十个 token。

省下的计算也直接变成了成本的下降（第 72 讲）。这也是为什么很多大模型 API 对“命中缓存的输入 token”大幅打折：服务商真的不用为它们花多少算力。

另一种组织方式：前缀树（RadixAttention）

除了 vLLM 的块级哈希表，SGLang 提出了另一种组织方式，叫 **RadixAttention**：把所有缓存的前缀组织成一棵**基数树**（Radix Tree，一种压缩前缀树）。

树的根节点是空的，树枝上的每一条路径代表一段 token 序列，树节点中保存着这段序列对应的物理 KV 块指针。

- **前缀匹配**：新请求进来，沿着基数树自顶向下比对 token。匹配到哪里，哪里之前的物理块就直接命中；
- **分叉与合并**：当新请求在某个节点中间产生分歧时，基数树会自动把该节点拆分为两个子节点（分叉）；当某个分支被释放或变空时又可以合并；
- **淘汰机制**：当显存吃紧需要腾空间时，基数树从最深层的叶子节点开始按 LRU 顺序剪枝，永远不会误删公共的前缀根节点。

我们把两种实现方式对照来看：

对比维度	vLLM（块级哈希表）	SGLang（RadixAttention 基数树）
核心数据结构	扁平哈希表（`Hash → Block`）	压缩前缀树（Radix Tree）
命中粒度	块级别（固定 16 个 token）	可支持细粒度到单个 token
前缀共享机制	链式哈希 + 物理块引用计数	树的父子节点继承 + 读写锁
本质	完全相同：依靠因果掩码的确定性，按 token 前缀复用物理 KV Cache	完全相同：依靠因果掩码的确定性，按 token 前缀复用物理 KV Cache

哈希表的好处是实现简单、查询复杂度稳定是 $O(1)$，天然与物理块对齐；基数树的好处是结构上天然契合多分支搜索、智能体多轮探索（如 MCTS 搜索树），而且能更自然地做到单 token 级别的精细复用。

用代码复现

下面的代码实现了按块算链式哈希、逐块查表的前缀缓存，模拟了上面 4 个请求；最后用第 68 讲的方法估算命中前后的 prefill 时间。

```
import hashlib

B = 4                                                  # 每块 4 个 token（这里用一个字当一个 token，方便看）

def block_hashes(tokens):
    """只给装满的块算哈希；每块的哈希 = 哈希(前一块的哈希 + 这一块的内容)，所以它代表"从开头到这一块为止的全部内容"。"""
    hs, parent = [], ""
    for i in range(len(tokens) // B):
        parent = hashlib.sha256((parent + "|" + tokens[i * B:(i + 1) * B]).encode()).hexdigest()[:8]
        hs.append(parent)
    return hs

cache = set()                                          # 已经算过、还留在显存里的块（按哈希索引）
def serve(name, text):
    hs = block_hashes(text)
    hit = 0
    for h in hs:                                       # 从头开始找，遇到第一个没命中的块就停
        if h in cache: hit += 1
        else: break
    cache.update(hs)                                   # 算完以后，这个请求的块也留在缓存里
    print(f"{name}: {len(text)} 个 token, {len(hs)} 个整块, 命中 {hit} 块 ({hit * B} 个 token 不用再 prefill) , 要算 {len(text) - hit * B} 个")

sys_prompt = "你是一个乐于助人的助手，请用简洁的中文回答。"
serve("请求 1", sys_prompt + "问：什么是注意力？")
serve("请求 2", sys_prompt + "问：什么是残差连接？")
turn1 = sys_prompt + "问：什么是注意力？答：每个词按相关程度收集其他词的信息。"
serve("请求 3（多轮的第 2 轮）", turn1 + "问：那多头注意力呢？")
serve("请求 4（系统提示改了第一个字）", "您" + sys_prompt[1:] + "问：什么是注意力？")

# Llama-3-8B + H100：系统提示 2000 个 token、问题 100 个时，命中与不命中的 prefill 时间（第 68 讲的估算）
def prefill_ms(n):
    return max(2 * 7.5e9 * n / 989e12, 16.06e9 / 3.35e12) * 1e3
print(f"不命中：prefill 2100 个 token ≈ {prefill_ms(2100):.1f} ms；命中系统提示：只算 100 个 ≈ {prefill_ms(100):.1f} ms")
```

输出：

```
请求 1：31 个 token，7 个整块，命中 0 块（0 个 token 不用再 prefill），要算 31 个
请求 2：32 个 token，8 个整块，命中 6 块（24 个 token 不用再 prefill），要算 8 个
请求 3（多轮的第 2 轮）：60 个 token，15 个整块，命中 7 块（28 个 token 不用再 prefill），
要算 32 个
请求 4（系统提示改了第一个字）：31 个 token，7 个整块，命中 0 块（0 个 token 不用再
prefill），要算 31 个
不命中：prefill 2100 个 token ≈ 31.9 ms；命中系统提示：只算 100 个 ≈ 4.8 ms
```

前四行就是 4 个请求的命中情况，和演示一致：请求 2 命中 6 块，请求 3 命中 7 块，请求 4 一块也没命中。最后一行是 Llama-3-8B 上的 prefill 时间：31.9 ms 对 4.8 ms。

到了大模型里：从单请求到 Agent 做到 95% 缓存命中

前缀缓存最能发挥威力的地方，是智能体（Agent）场景。

一个典型的 Agent 循环包含大量循环迭代：$[\text{系统提示} + \text{工具定义}] + [\text{多轮历史步骤}] \rightarrow \text{模型输出工具调用} \rightarrow \text{执行工具并追加结果} \rightarrow \text{再次调用模型}$

在每一轮中，模型的输入都是“上一轮的全部内容加上一小段工具执行结果”，而模型输出的工具调用指令通常只有几十个 token。输入和输出的 token 数量可以相差一到两个数量级。这意味着绝大部分输入都是前缀重复！

前缀缓存命中的理论上限非常高：$\text{命中率上限} \approx \frac{\text{上下文总长} - \text{本轮新增长度}}{\text{上下文总长}}$

例如上下文已经积累到了 50K token，本轮工具结果只增加了 1K token，理论命中率上限高达 $98%$。在实际生产系统中，很多优化良好的 Agent 都能稳定做到 $90%$ 到 $95%$ 的缓存命中率。

但是，要达到这个数字，必须遵循严格的工程规则：

1. **易变内容绝对后置**：所有规则的核心都在于“前缀必须逐 token 绝对稳定”。最常见的反面例子是在系统提示的第一行写上 `“当前时间：2026-11-08 09:38:00”` 或动态生成的 `Session-ID`。由于第一块的第一个 token 每秒都在变，链式哈希从第 0 块就彻底崩塌，后续 5 万个 token 的命中率直接归零！任何动态时间戳、随机数、用户信息，真要使用必须放到输入的最末端。
2. **上下文只追加（Append-Only）**：严禁随意修改、删除或重新排序历史消息。常见的错误做法是“滑动窗口截断”——每轮为了省显存把最老的一条消息删掉。老消息一删，后面所有的 token 往前平移几位，位置编码一变，链式哈希完全不匹配，导致前缀大面积失效。

3. **序列化必须确定**：JSON 字段顺序、空格缩进、换行符、浮点数格式必须保持严格一致。在程序逻辑上虽然 `{"a": 1, "b": 2}` 和 `{"b": 2, "a": 1}` 语义完全相同，但分词出来的 token 序列完全不同，哈希表无法命中。
4. **工具定义不要中途动态增删**：工具定义（Tool Schemas）通常位于提示词非常靠前的位置。如果在第 3 轮临时加了一个新工具，后面的所有前缀缓存全部报废。规范的做法是保持工具定义列表始终固定，如果在某些步骤需要限制模型调用的工具，通过约束解码（第 80 讲）在采样时屏蔽非法 token，而不是修改提示词。
5. **网关层做会话粘性路由（Cache-aware Routing）**：GPU 显存里的前缀缓存是单台机器独立的。如果你的服务部署了 10 台 8 卡实例，而网关层采用随机轮询做负载均衡，那么用户连续两次请求落到同一台机器的概率只有 $10%$，前缀缓存命中率直接跳水。网关必须根据用户会话 ID 或前缀哈希进行会话保持，把相同的会话粘在同一台实例上（第 114 讲）。

容易混淆的地方

1. **只能复用前缀，不能复用中间的片段**。两个请求中间有一段相同的内容，但前面不同，这段内容的 KV Cache 也不同（每个 token 的 k、v 取决于它前面的全部内容），不能复用。
2. **复用的是 KV Cache，不是答案**。前缀缓存只省 prefill 的计算，模型还是要为新请求生成全新的回答。它和“相同的问题直接返回上次的答案”的语义缓存（Semantic Cache）是两回事。语义缓存不进模型、答案可能不准；前缀缓存进模型、计算结果与不缓存完全一致。
3. **位置也要一致**。同样的内容出现在不同的位置（比如前面多了一个 token），位置编码不同，KV Cache 也不同（第 25、37 讲）。所以只有从头开始完全相同的部分才能命中。

回顾

很多请求有相同的开头：系统提示、多轮对话的历史、同一份长文档。因果掩码保证相同前缀的 KV Cache 完全相同，所以可以算一次、存起来复用。vLLM 给每个装满的物理块（16 个 token）计算链式哈希作为指纹，把上一块的指纹作为本块的输入，从数学上保证它代表了从开头到该块末尾的完整因果上下文；新请求从头逐块查表，命中的物理块直接放进块表，只对剩下未命中的部分做 Prefill。引用计数降到 0 的物理块不会被立即释放，而是保留在显存的空闲 LRU 队列中，随时等待被后续相同前缀唤醒。SGLang 则用基数树（Radix Tree）以树状结构组织前缀，支持细粒度的节点匹配与剪枝。在 Agent 场景中，只要遵守“易变内容后置、上下文只追加、粘性路由”等规则，缓存命中率可高达 95% 以上，大幅降低延迟与硬件开销。

前缀缓存能让很多请求跳过一大段 prefill。可没有命中的长提示怎么办？一个几万 token 的 prefill 要算好几秒，这期间所有正在 decode 的请求都被卡住了。下一讲：**Chunked Prefill**。

第77讲 Chunked Prefill

68 Prefill 与 Decode　70 TTFT、TPOT 与吞吐　74 Continuous Batching

第 74 讲最后留了一个问题：连续批处理让新请求随时加入，可新请求要先做 prefill。假设 32 个用户正在流畅地接收回答，每 6 ms 左右蹦出一个字；这时来了一个新请求，带着一份 8000 token 的长文档。它的 prefill 被塞进下一步，这一步要算 8000 多个 token，耗时一下子变成了一百多毫秒。这一百多毫秒里，那 32 个用户的回答全都停住了。

这一讲的 **Chunked Prefill**（分块 prefill）解决的就是这个问题：把长提示的 prefill 切成小块，分几步做，每一步都和 decode 一起进行，谁也不卡谁。

一个长提示能卡多久

先用第 68 讲的方法算一下（Llama-3-8B，一张 H100，每一步的耗时 ≈ max(计算量 ÷ 算力，读写字节 ÷ 带宽)）。

32 个请求一起 decode，每个上下文约 1000 个 token。这一步每个请求只处理 1 个新 token，计算量很小，主要的时间花在把权重读一遍、再读一遍 KV Cache 上，约 **6.05 ms**。这就是这 32 个请求平时的 TPOT（第 70 讲）。

这时来了一个 8000 token 的提示。如果把它的 prefill 整段放进下一步，这一步要处理 8032 个 token，计算量是平时的两百多倍，变成了计算受限，耗时约 **138.9 ms**。

对那 32 个正在 decode 的请求来说，这一步也只给每人出了 1 个 token。它们的 TPOT 从 6 ms 突然跳到了 139 ms，用户看到的就是回答正流畅地往外蹦字，忽然卡了一下。如果来的是 3 万、10 万 token 的提示，卡顿就是半秒、几秒。第 70 讲说过，TPOT 决定了用户感受到的“流畅程度”，服务通常对它有严格的要求（比如 P99 不超过 50 ms）。偶尔一次长提示，就能让所有人的 P99 超标。

反过来，如果为了不卡别人，让长提示等到没有 decode 请求时再做，它的 TTFT 又会无限地拖下去。两头都不好。

把 prefill 切开

第 69 讲说过，prefill 和 decode 做的其实是同一件事：处理一批新 token，把它们的 k、v 写进 KV Cache。prefill 一次处理很多个，decode 一次处理 1 个。所以 prefill **不必一次做完**：8000 个 token，可以先处理前 480 个，把它们的 KV 写进缓存；下一步再处理第 481 到 960 个，这时它们的注意力要看的“前面的 token”，一部分在缓存里（前 480 个），一部分在这一块里；依此类推。每一块的计算方式和 decode 完全一样，只是一次处理的新 token 多一些。最后一块处理完，就得到了第一个输出 token，和一次做完的结果完全相同。

于是调度器可以这样安排每一步：

1. 给这一步定一个 **token 预算**，比如 512 个。
2. 先把预算分给所有正在 decode 的请求，每个 1 个（32 个请求用掉 32 个）。
3. 剩下的 480 个，分给正在 prefill 的请求：从长提示里切下 480 个 token，这一步处理它们。

每一步处理的 token 数不超过预算，耗时就有了上限。8000 个 token 的提示，每步 480 个，分 17 步做完。这 17 步里，32 个 decode 请求每一步都照常出 1 个 token，每步最长约 **9.7 ms**；长提示的 TTFT 约 **146.9 ms**，和一次做完的 138.9 ms 差不多。

这个做法来自 2023–2024 年的 SARATHI 和 Sarathi-Serve 论文，后者把它叫作“不停顿的批处理”（stall-free batching）：decode 永远不会因为 prefill 而停下来。

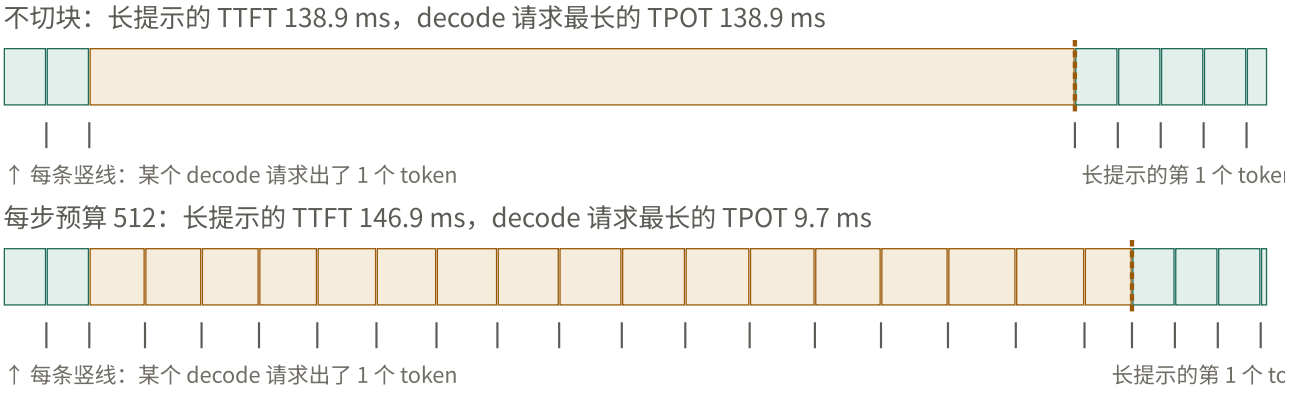
下面的时间线画出了这两种安排。每一格是一步，宽度是这一步的耗时，琥珀色的步里带着 prefill。下面的小竖线是某个 decode 请求每出一个 token 的时刻，竖线之间的间隔就是它的 TPOT。换一换预算、拖一拖提示长度看看。

长提示插进来：切块前后

⇆ 对照

每步的 token 预算： 2048 1024 512 256

长提示 = **8000** 个 token



平时一步（32 个 decode）≈ 6.05 ms。不切块：整段 prefill 塞进一步，耗时 138.9 ms，这一步里 32 个请求都在干等。
预算 512：每步先给 32 个 decode，再切 480 个 prefill token，分 17 步做完；每步最长 9.7 ms，这期间每个 decode 请求出了 17 个 token（不切块只出 1 个）。
长提示的 TTFT 从 138.9 ms 变成 146.9 ms。预算太小，每步都要把权重读一遍，步数多了，TTFT 会明显变长（试试 256）。

图 77-1　长提示插进来时，不切块（上）和每步预算 512（下）。不切块时，整段 prefill 挤进一步，decode 请求有一百多毫秒出不了 token；切块后每步都不长，decode 照常进行，长提示的 TTFT 几乎不变。

要点

Chunked Prefill 给每一步设一个 token 预算：先保证每个 decode 请求的 1 个 token，剩下的预算给 prefill，长提示被切成若干块分几步做完。每一步的耗时有了上限，decode 不再被长提示卡住。

预算多大合适

预算是一个权衡。代码算出了几个不同预算的结果（8000 token 的提示，32 个请求在 decode）：

- **不切块**：1 步，TTFT 138.9 ms，decode 被卡 138.9 ms。
- 2048：4 步，TTFT 140.4 ms，最长 TPOT 37.4 ms。

- **1024**：9 步，TTFT 147.5 ms，最长 TPOT 19.5 ms。
- **512**：17 步，TTFT 146.9 ms，最长 TPOT 9.7 ms。
- **256**：36 步，TTFT 223.5 ms，最长 TPOT 6.4 ms。

预算越小，decode 越平稳；但预算太小，长提示的 TTFT 会明显变长。原因是**每一步都要把权重从显存里读一遍**（约 4.8 ms），预算 256 时，每步只能处理 224 个 prefill token，读权重的时间占了大头，步数又多（36 步），总时间就长了。预算 512 以上时，每一步基本是计算受限的，切成几块总的计算量不变，TTFT 也就几乎不变。

这和第 62 讲的 Roofline 是同一个道理：每一步要读一遍权重是固定成本，一步里处理的 token 越多，这个固定成本就被摊得越薄。

白捡的算力

还有一个更妙的地方。第 68 讲说过，decode 是带宽受限的：32 个请求一起 decode 一步，计算单元大部分时间在等数据。代码算出，这一步**再塞进 363 个 prefill token**，计算时间也还没超过读数据的时间，耗时仍然约 6.06 ms，几乎不变。

也就是说，decode 的每一步里，本来就有一部分算力闲着。Chunked Prefill 把 prefill 切成块，正好填进这些闲着的算力里：prefill 几乎是“搭顺风车”完成的。这让 GPU 的利用率更高，整体吞吐也会提高。

到了大模型里

vLLM 的新版本默认开启。vLLM 的 V1 引擎里，chunked prefill 默认打开，预算由参数 `max_num_batched_tokens` 控制（第 74 讲提过它）。调度器每一步先排 decode，再用剩下的预算排 prefill。SGLang 也有对应的参数（`chunked_prefill_size`）。具体默认值随版本和硬件变化，以文档为准。

预算怎么定。预算大一些，吞吐和 TTFT 更好；预算小一些，TPOT 更稳。延迟要求严的在线服务，预算可以设小一点；离线批量处理，预算可以设得很大。经验上，预算让一步落在“刚好开始计算受限”的位置附近比较好，对 H100 上的 8B 模型，大约是几百到两千个 token。

和前缀缓存配合。长提示如果开头命中了前缀缓存（第 76 讲），命中的部分直接跳过，只把没命中的部分切块。

另一种思路：干脆分开。chunked prefill 让 prefill 和 decode 在同一张卡上和平共处。另一种做法是让 prefill 和 decode 在不同的卡上进行，互不干扰，这就是第 14 篇的 PD 分离。两种思路各有适用的场景，到那时再比较。

用代码复现

下面的代码按第 68 讲的方法估算每一步的耗时，先算出平时 decode 一步的耗时和“白捡”的 prefill 容量，再比较不同预算下长提示的 TTFT 和 decode 请求的最长 TPOT。

```python
# Llama-3-8B (BF16) + 一张 H100。一步的耗时 ≈ max(计算量 / 算力, 读写字节 / 带宽)(第 68 讲)
N, N_emb, L, d = 8.03e9, 128256 * 4096, 32, 4096
W_BYTES, KV, F, BW = N * 2, 2 * 32 * 8 * 128 * 2, 989e12, 3.35e12

def cost(dec, dec_ctx, pre, pre_ctx):
    """一步: dec 个请求各 decode 1 个 token (上下文 dec_ctx) ; 外加一段 pre 个 token 的 prefill (这段前面已有 pre_ctx 个) 。"""
    flops = 2 * (N - N_emb) * (dec + pre) + 4 * L * d * (dec * dec_ctx + pre * (pre_ctx + pre / 2))
    bytes_ = W_BYTES + KV * (dec * dec_ctx + pre_ctx + pre)
    return flops / F * 1e3, bytes_ / BW * 1e3           # (算的时间, 读写的时间), 取大的那个

def step_ms(*a):
    return max(cost(*a))

D, CTX, P = 32, 1000, 8000            # 32 个请求正在 decode; 来了一个 8000 token 的长提示
print(f"平时: {D} 个请求一起 decode 一步 ≈ {step_ms(D, CTX, 0, 0):.2f} ms (这就是它们的 TPOT) ")
free = 0
while cost(D, CTX, free + 1, 0)[0] ⩽ cost(D, CTX, free + 1, 0)[1]: free += 1
print(f"这一步最多再塞进 {free} 个 prefill token, 仍然卡在带宽上, 耗时 {step_ms(D, CTX, free, 0):.2f} ms, 几乎不变")

print("token 预算 | prefill 分几步 | 长提示的 TTFT | 这期间 decode 请求的最长 TPOT")
for budget in [P + D, 2048, 1024, 512, 256]:
    done, t, worst, steps = 0, 0.0, 0.0, 0
    while done < P:
        chunk = min(budget - D, P - done)     # 先给 decode 留位置, 剩下的预算给 prefill
        ms = step_ms(D, CTX, chunk, done)
        t, worst, done, steps = t + ms, max(worst, ms), done + chunk, steps + 1
    name = "不切块" if budget > P else f"{budget:6d}"
    print(f"{name:>8} | {steps:6d} 步 | {t:7.1f} ms | {worst:6.1f} ms")
```

输出:

```
平时：32 个请求一起 decode 一步 ≈ 6.05 ms（这就是它们的 TPOT）
这一步最多再塞进 363 个 prefill token，仍然卡在带宽上，耗时 6.06 ms，几乎不变
token 预算 | prefill 分几步 | 长提示的 TTFT | 这期间 decode 请求的最长 TPOT
     不切块 |       1 步 |   138.9 ms |  138.9 ms
    2048 |       4 步 |   140.4 ms |   37.4 ms
    1024 |       9 步 |   147.5 ms |   19.5 ms
     512 |      17 步 |   146.9 ms |    9.7 ms
     256 |      36 步 |   223.5 ms |    6.4 ms
```

第一行是平时的 TPOT（6.05 ms）；第二行是 decode 一步里还能白捡的 prefill 容量（363 个 token）；表格就是上面那五种预算的结果。“prefill 分几步”同时也是这期间每个 decode 请求出的 token 数：不切块时只出 1 个，预算 512 时出了 17 个。

容易混淆的地方

1. **切块不改变结果**。每一块的注意力都会看到前面所有块的 KV（从缓存里读），最后得到的 KV Cache 和第一个输出 token，与一次做完完全相同。
2. **切块也有代价**。后面的块要从缓存里读前面块的 KV，块切得越细，这部分重复读取越多；每一步还要读一遍权重。所以预算不能无限小。
3. **长提示自己的 TTFT 并没有变短**。chunked prefill 主要保护的是**别人**的 TPOT，长提示自己的 TTFT 基本不变，预算太小时还会变长。要缩短长提示的 TTFT，靠的是前缀缓存、更多的算力，或者第 14 篇的并行方法。

回顾

连续批处理让新请求随时加入，可长提示的 prefill 挤进一步，会让这一步耗时暴涨，所有 decode 请求都被卡住：32 个请求在 decode 时来了 8000 token 的提示，这一步要 138.9 ms，而平时只要 6.05 ms。Chunked Prefill 给每一步设一个 token 预算，先给每个 decode 请求 1 个，剩下的给 prefill，长提示切成块分几步做完。预算 512 时，分 17 步，每步最长 9.7 ms，长提示的 TTFT 仍然约 147 ms。预算太小，每步都要读一遍权重，TTFT 会变长；decode 一步里本来闲着的算力，还能顺带处理几百个 prefill token。

到这里，每一步“处理哪些 token”已经有了章法。可显存总是有限的：如果正在运行的请求越生成越长，KV Cache 的块用完了，怎么办？是拒绝新请求，还是让某些请求先让出位置？下一讲：**调度与抢占**。

第78讲 调度与抢占

71 显存账本　74 Continuous Batching　75 PagedAttention　77 Chunked Prefill

前面几讲让推理引擎变得越来越“贪心”：连续批处理（第 74 讲）让请求随时进来，分页（第 75 讲）让 KV Cache 一块一块按需分配，几乎不浪费。可是按需分配有一个隐患：一个请求进来时只分配了提示需要的块，它后面要生成多少个 token，事先不知道。几十个请求同时在生成，KV Cache 一起增长，池子总有被占满的时候。

这时某个请求要写下一个 token，却拿不到新块了。怎么办？这一讲讲推理引擎的**调度器**（scheduler）怎么回答两个问题：**新请求什么时候放进来**，以及**块不够时，让谁先让出位置**。

两种放行的态度

调度器手里有三个队列：

- **等待队列**（waiting）：已经到达、还没开始的请求，以及被抢占后等着重来的请求。
- **运行队列**（running）：正在 prefill 或 decode 的请求，每一步都要算。
- **已完成**：生成了结束标记或达到长度上限，块已经还回池子。

每一步开始时，调度器要决定能不能从等待队列里再放一些请求进来。这里有两种态度。

保守：按最终长度预留。一个请求进来时，就按“提示长度 + 最多生成的长度”把它将来要用的块全部预留好。这样它进来以后永远不会缺块，也就不会被打断。缺点和第 75 讲的连续预留一样：大部分预留的块在很长一段时间里是空着的，能同时运行的请求少，后来的请求只能等。而且真实的请求往往设了一个很大的长度上限（比如 4096），实际却只生成几百个 token，按上限预留浪费更多。

乐观：按需分配。请求进来时只分配提示需要的块，之后每写满一块再要一块。池子里能同时运行的请求多得多，GPU 的批也更满。代价是：一旦池子被占满，有人要新块却拿不到，就必须让某个请求**让出**它的块。这叫**抢占**（preemption）。

vLLM、SGLang 等推理引擎都选了乐观的做法，再用抢占兜底。

用一个小例子看抢占

我们用一个缩小的池子模拟一下：16 个块，每块 4 个 token；6 个请求，各自的到达时间、提示长度和要生成的 token 数如下。

请求	A	B	C	D	E	F
第几步到达	0	0	1	2	3	4
提示长度	7	5	9	6	8	4
要生成的 token 数	9	12	6	10	5	7

调度的规则是这样的，每一步依次做三件事：

1. **运行中的请求先拿块**。按加入的顺序，每个请求要写 1 个新 token；最后一块写满了，就从池子里取一块新的。如果池子空了，就抢占**最后加入**的那个请求：把它的块全部释放，放回等待队列的最前面。
2. **放新请求进来**。如果这一步没有发生抢占，就按顺序看等待队列的第一个请求，池子里的空闲块够放下它，就放进来，做 prefill 并出第 1 个 token。
3. **生成完的请求离开**，块还回池子。

按需分配时，A 到 F 在第 0 到 4 步都顺利进来了，每个只占提示需要的两三块。到第 5 步，16 块已经全部用完，D 写满了它的最后一块，要一块新的，池子空了。调度器抢占了最后加入的 F：F 的块全部释放，F 回到等待队列的最前面。第 6 步，A 也要新块，又不够了，这次抢占的是 E。第 7 步，C 生成完毕还回了块，E 和 F 重新进来。它们之前算过的 KV Cache 已经扔掉了，要把提示和已经生成的部分重新 prefill 一遍：E 要重算 11 个 token（提示 8 个 + 已生成 3 个），F 要重算 5 个。全部请求在 **13 步**完成，平均延迟 **8.7 步**，一共抢占 2 次、重算 16 个 token。

按最终长度预留时，A、B、C 进来就各占了 4 到 5 块，13 块已经被预留，D 要 4 块，放不下，只能等。D 一直等到第 7 步 C 结束才进来，E、F 等到第 9 步。没有任何抢占，但全部完成要 **17 步**，平均延迟 **10.8 步**。

下面的演示可以一步一步地看这两种做法。格子是池子里的 16 个块，下面是三个队列和最近发生的事件。



图 78-1　调度与抢占。按需分配时，6 个请求很快都进来了，第 5、6 步块不够，先后抢占了最后加入的 F 和 E，它们之后要重算；按最终长度预留时，没有抢占，但 D、E、F 等了很久才进来。

这个小例子里，抢占带来的重算代价（16 个 token），远远小于多运行几个请求带来的收益。实际系统也是这样：抢占偶尔发生，大部分时间池子都被充分利用。

要点

推理引擎按需给请求分配 KV Cache 的块，能同时运行的请求更多；池子用完时，就抢占某个请求，释放它的块，让它回到等待队列，之后再接着算。抢占是兜底手段，用少量的重复计算换取平

时更高的利用率。

抢占谁

块不够时，牺牲谁？最常见的做法是抢占**最后加入**的请求（或者在带优先级的调度里，优先级最低的请求）。理由是：

- **公平**。先到的请求已经等了更久，按先来先服务（FCFS）的原则应该先完成。
- **代价小**。最后加入的请求通常算得最少，扔掉它的 KV Cache，要重算的也最少。
- **不会饿死**。被抢占的请求回到等待队列的**最前面**，一有空位就最先回来，不会被后来的请求一直插队。

还有一个细节：发生抢占的那一步，调度器**不再放新请求进来**。道理很简单：刚因为块不够把别人赶出去，又放新人进来，马上又会不够，请求在进进出出之间来回折腾（这叫“抖动”，thrashing）。有些实现还会在池子里留一小部分块作为“水位线”（watermark），空闲块少于它时就不再放新请求，提前避免抢占。

重算还是换出

被抢占的请求，它的 KV Cache 怎么处理？有两种办法：

- **重算**（recompute）：直接扔掉。请求重新回来时，把提示和已经生成的 token 一起重新 prefill 一遍，KV Cache 就又有了。
- **换出**（swap）：把它的 KV Cache 块复制到 CPU 内存里，块还回池子；请求回来时，再从 CPU 内存复制回显存。

哪个更快？代码估算了 Llama-3-8B 上一个 2000 token 的请求：它的 KV Cache 有 **262 MB**，重算（prefill 2000 个 token）约 **30.3 ms**，换出再换回（按 PCIe 5.0 约 50 GB/s 估算）约 **10.5 ms**。只看这一个数字，换出更快。

可实际的推理引擎更偏向重算。vLLM 的 V1 引擎只保留了重算这一种方式（具体以所用版本的文档为准）。原因有几个：

- **换出要占 PCIe 带宽和 CPU 内存**，抢占往往发生在负载最高的时候，大量请求同时换进换出，PCIe 很快成了瓶颈，还要管理一个 CPU 端的块池，实现复杂得多。

- **重算可以顺便利用前缀缓存**。被抢占的请求释放的块，只要还没被别人覆盖，仍然留在前缀缓存里（第 76 讲）。它回来时，前面的部分很可能直接命中，真正要重算的只有一部分。
- **重算的 prefill 可以切块**（第 77 讲），和别的请求的 decode 一起进行，不会卡住别人。
- **抢占本来就应该很少发生**。如果频繁抢占，说明负载已经超过了这张卡的能力，该做的是限流或者加卡，而不是优化抢占本身。

先服务谁：调度策略

除了“块不够时抢占谁”，调度器还要决定等待队列里**谁先进来**。最常见的是**先来先服务**（FCFS），简单、公平、不会饿死人。但它不一定是最好的：

- **队头阻塞**。一个很长的请求排在前面，池子放不下它，后面一串短请求即使放得下，也只能干等。有的调度器允许短请求“越过”放不下的长请求先进来，但要防止长请求被一直越过、永远轮不到。
- **优先级**。付费用户、在线交互的请求，可能比离线批量任务更重要。vLLM 支持按优先级调度：优先级高的先进来，块不够时优先抢占优先级低的。
- **缓存感知**。SGLang 的调度器可以优先放进**和前缀缓存重叠最多**的请求（最长前缀匹配，第 76 讲），让缓存的命中率更高，整体吞吐更高。代价是有些请求可能要多等一会儿。

调度策略没有对所有场景都最好的答案，它总是在吞吐、平均延迟、尾延迟和公平之间做取舍。

到了大模型里

几个关键的参数。vLLM 里和调度直接相关的有：`max_num_seqs`（同时运行的请求数上限）、`max_num_batched_tokens`（每一步的 token 预算，第 77 讲）、`gpu_memory_utilization`（多大比例的显存给模型和 KV Cache，决定了池子有多大，第 71 讲）。

看日志里的抢占。vLLM 在发生抢占时会打出警告，监控指标里也有抢占次数。偶尔出现没关系；如果持续出现，说明 KV Cache 的池子相对于负载太小了：可以调小 `max_num_seqs`，调大 `gpu_memory_utilization`，换更大显存的卡，或者用 KV Cache 量化（第 12 篇）、多卡并行（第 13 篇）扩大池子。

调度器自己要够快。每一步都要跑一遍上面的逻辑：遍历几百个请求、分配和释放块、整理块表。decode 一步只有几毫秒，调度如果花上一两毫秒，GPU 就会空等。所以推理引擎在调度器的实现上花了很多功夫，比如让调度和 GPU 计算重叠进行（第 81、89 讲）。

用代码复现

下面的代码实现了上面的调度规则，模拟这 6 个请求在两种放行方式下的过程，打出按需分配时发生的抢占；最后估算 Llama-3-8B 上重算和换出的代价。

```
B, NB = 4, 16                                          # 每块 4 个 token，池子一共
16 块（64 个 token）
# 请求：名字、第几步到达、提示长度、要生成的 token 数
REQS = [("A", 0, 7, 9), ("B", 0, 5, 12), ("C", 1, 9, 6), ("D", 2, 6, 10), ("E", 3,
8, 5), ("F", 4, 4, 7)]
need = lambda n: -(-n // B)                            # n 个 token 要几块

def simulate(mode, log=False):
    """mode = "reserve"：按最终长度一次预留，不会被抢占；"preempt"：按需分配，块不够就抢占最
后到的请求（重算）。"""
    gen = {r[0]: 0 for r in REQS}; kv = {}; blocks = {}; free = NB
    waiting, running, finish, preempts, recompute, step = [], [], {}, 0, 0, 0
    while len(finish) < len(REQS):
        waiting += [r for r in REQS if r[1] == step]
        # 1. 正在运行的请求各要写 1 个新 token；块不够就抢占队尾的请求
        preempted_now = False
        for r in list(running):
            if r not in running: continue
            n = r[0]
            while need(kv[n] + 1) > blocks[n] and free == 0:
                v = running.pop()                      # 最后加入的请求让出全部块，回
到等待队列最前面
                free += blocks.pop(v[0]); kv.pop(v[0]); waiting.insert(0, v)
                preempts += 1; recompute += v[2] + gen[v[0]]; preempted_now = True
                if log: print(f"  第 {step:2d} 步：块不够了，抢占 {v[0]}（释放它的块，之
后要重算 {v[2] + gen[v[0]]} 个 token）")
                if v is r: break
            if r in running:
                if need(kv[n] + 1) > blocks[n]: blocks[n] += 1; free -= 1
                kv[n] += 1; gen[n] += 1
        # 2. 没有发生抢占时，按到达顺序放等待的请求进来（prefill 提示和已生成的部分，并出 1
个 token）
        while waiting and not preempted_now:
            r = waiting[0]; n = r[0]
            want = need(r[2] + r[3]) if mode == "reserve" else need(r[2] + gen[n])
            if want > free: break
            waiting.pop(0); running.append(r); free -= want; blocks[n] = want;
kv[n] = r[2] + gen[n]; gen[n] += 1
        # 3. 生成完的请求离开，块还回池子
        for r in [r for r in running if gen[r[0]] == r[3]]:
            running.remove(r); free += blocks.pop(r[0]); kv.pop(r[0]); finish[r[0]]
= step + 1
        step += 1
    lat = [finish[r[0]] - r[1] for r in REQS]
    return step, sum(lat) / len(lat), preempts, recompute, {r[0]: finish[r[0]] for
r in REQS}
```

```
for mode, name in [("reserve", "按最终长度预留"), ("preempt", "按需分配 + 抢占")]:
    steps, avg, pre, rc, fin = simulate(mode, log=(mode == "preempt"))
    print(f"{name}：全部完成 {steps} 步，平均延迟 {avg:.1f} 步，抢占 {pre} 次，重算 {rc}
个 token；完成时刻 {fin}")

# 真实规模：Llama-3-8B 一个 2000 token 的请求被抢占，重算 vs 换出到 CPU 内存再换回
KV, F, Nn, PCIE = 2 * 32 * 8 * 128 * 2, 989e12, 7.5e9, 50e9          # PCIe 5.0 x16
按约 50 GB/s 估
n = 2000
print(f"2000 个 token 的 KV Cache = {KV * n / 1e6:.0f} MB；重算 (prefill) ≈ {2 * Nn *
n / F * 1e3:.1f} ms；"
      f"换出再换回 ≈ {2 * KV * n / PCIE * 1e3:.1f} ms")
```

输出：

```
按最终长度预留：全部完成 17 步，平均延迟 10.8 步，抢占 0 次，重算 0 个 token；完成时刻 {'A':
9, 'B': 12, 'C': 7, 'D': 17, 'E': 14, 'F': 16}
  第  5 步：块不够了，抢占 F（释放它的块，之后要重算 5 个 token）
  第  6 步：块不够了，抢占 E（释放它的块，之后要重算 11 个 token）
按需分配 + 抢占：全部完成 13 步，平均延迟 8.7 步，抢占 2 次，重算 16 个 token；完成时刻
{'A': 9, 'B': 12, 'C': 7, 'D': 12, 'E': 9, 'F': 13}
2000 个 token 的 KV Cache = 262 MB；重算 (prefill) ≈ 30.3 ms；换出再换回 ≈ 10.5 ms
```

第一行是按最终长度预留：17 步、平均延迟 10.8 步、没有抢占。接下来两行是按需分配时的两次抢占，和演示里第 5、6 步的事件一致；然后是它的结果：13 步、8.7 步、抢占 2 次、重算 16 个 token。最后一行是真实规模下重算和换出的代价。

容易混淆的地方

1. **抢占不是取消。**被抢占的请求不会失败，也不会被丢掉，只是暂停，之后从断开的地方接着生成。用户看到的最多是一次停顿。
2. **重算的结果和原来完全一样。**被抢占前已经生成的 token 会被保留，重新 prefill 时作为输入的一部分，所以接下来的生成和没被抢占时一致（不考虑浮点误差）。
3. **抢占解决的是 KV Cache 不够，不是算力不够。**算力不够时，每一步变慢，TPOT 变长；显存不够时，才会触发抢占。两者都是过载的表现，但处理方式不同。

回顾

按需分配让 KV Cache 的池子能同时装下更多请求，但请求的长度事先不知道，池子总可能被占满。调度器每一步先让运行中的请求拿块，块不够时抢占最后加入的请求，释放它的块，让它回到等待队列的最前面；发生抢占的那一步不放新请求进来，避免来回折腾。6 个请求、16 块的例子里，按需分配加抢占 13 步完成，按最终长度预留要 17 步。被抢占的请求可以重算或换出，现在的推理引擎多用重算，因为它简单，还能利用前缀缓存和分块 prefill。放请求进来的顺序默认是先来先服务，也可以按优先级或缓存命中来排。

到这里，推理引擎的核心机制已经齐了。调度时我们一直默认所有请求用同一份模型；可如果几百个客户各有一个微调过的版本，一个 batch 里的请求要用不同的权重，又该怎么算？下一讲：**多 LoRA 服务**。

第79讲 多 LoRA 服务

选读

50 LoRA 参数高效微调　71 显存账本　74 Continuous Batching　78 调度与抢占

一家做客服机器人的公司有 100 个企业客户，每个客户都用自己的对话记录微调了一份 Llama-3-8B。如果每份都全量微调、单独部署，就是 100 份 16 GB 的权重，至少要 100 张卡，哪怕大多数客户一天只有几个请求。第 50 讲给了另一条路：每个客户只训练一个 LoRA 适配器，几十 MB，大家共用同一份基座模型。

这一讲讲推理引擎怎么把这条路走通：一个 batch 里的请求属于不同的适配器，怎么一起算？代价在哪里？为什么 vLLM 要给“一个 batch 里最多几个适配器”设上限？

一个 batch，几个适配器

回忆第 50 讲：LoRA 不改原来的权重 W，而是在旁边加一条小路，

$$y = xW + \frac{\alpha}{r}(xA)\,B$$

A 是 d_in × r，B 是 r × d_out，秩 r 通常是 8、16、32。只服务一个客户时，最省事的做法是把小路合并进去：W′ = W + (α/r)AB，推理和原模型一模一样，没有任何额外开销。

可一旦要同时服务很多客户，就不能合并了：合并后的 W′ 是某一个客户专属的，换一个客户就要换一份 16 GB 的权重。所以多 LoRA 服务的基本做法是**不合并**：

- **大路所有请求一起算**。batch 里 B 个请求的 x 拼成 [B, d_in]，乘同一个 W，这一步和没有 LoRA 时完全一样，continuous batching 的收益全部保留（第 74 讲）。
- **小路各算各的**。第 i 个请求用它自己客户的 A_i、B_i，算 (x_i A_i) B_i，加到大路的结果上。

这样，100 个客户只需要显存里放一份基座、加上当前要用的那些适配器。

适配器有多大，多算多少

先算一个适配器的大小。Llama-3-8B 有 32 层，把 LoRA 加在全部 7 个线性层上（q、k、v、o 四个注意力投影，gate、up、down 三个 FFN 矩阵）。每个线性层的小路有 r × (d_in + d_out) 个参数：

```
import numpy as np
d, kv, ffn, L = 4096, 1024, 14336, 32          # Llama-3-8B：隐藏维度、KV 维度（8 个头 × 128）、FFN 宽度、层数
layers = {"q": (d, d), "k": (d, kv), "v": (d, kv), "o": (d, d), "gate": (d, ffn), "up": (d, ffn), "down": (ffn, d)}
io = sum(a + b for a, b in layers.values())
for r in (8, 16, 64):
    p = r * io * L
    print(f"r = {r:>2}：{p / 1e6:5.1f} M 个参数，BF16 {p * 2 / 1e6:6.1f} MB，是基座 80.3 亿参数的 {p / 8.03e9:.2%}")
r = 16
print(f"r = 16 时，每个 token 多算 2 × {r * io * L / 1e6:.1f} M 次乘加，基座是 2 × 8.03 G：多 {r * io * L / 8.03e9:.2%}")
```

```
r =  8：  21.0 M 个参数，BF16   41.9 MB，是基座 80.3 亿参数的 0.26%
r = 16：  41.9 M 个参数，BF16   83.9 MB，是基座 80.3 亿参数的 0.52%
r = 64：167.8 M 个参数，BF16  335.5 MB，是基座 80.3 亿参数的 2.09%
r = 16 时，每个 token 多算 2 × 41.9 M 次乘加，基座是 2 × 8.03 G：多 0.52%
```

r = 16 的适配器 84 MB，100 个也才 8.4 GB，一张 80 GB 的卡放得下基座加全部 100 个。计算量只多 0.5%：每个 token 只过它自己那一个适配器。看起来几乎是免费的。

反直觉的地方：计算免费，读不免费

可是 decode 不受计算限制，受带宽限制（第 68、72 讲）：每一步都要把用到的权重从显存完整地读一遍，读权重的时间就是这一步的时间。

基座的权重每一步读一遍，batch 里有多少请求都一样。适配器不同：**batch 里出现了几个不同的适配器，就要读几个适配器**。一个适配器只占基座的 0.52%，可要是一步里混着 50 个不同的适配器，就要多读 50 × 84 MB = 4.2 GB，是基座的 26%。计算只多 0.5%，时间却多了四分之一。

一个 batch 里会有几个不同的适配器？这取决于请求在客户之间怎么分布。假设一共 N 个适配器，请求落到第 i 个的概率是 p_i，batch 里有 B 个请求，那么第 i 个适配器至少出现一次的概率是 $1-(1-p_i)^B$，期望出现的不同适配器个数是：

$$D = \sum_{i=1}^{N} \left(1 - (1 - p_i)^B\right)$$

真实的流量往往很不均匀：少数大客户占了大部分请求。常用 Zipf 分布来描述：第 i 热门的适配器被选中的概率正比于 $1/i^s$，$s = 0$ 是完全均匀，s 越大越集中。

```
import numpy as np
def distinct(N, s, B):
    w = 1 / np.arange(1, N + 1) ** s
    p = w / w.sum()
    return (1 - (1 - p) ** B).sum()
for N, s, B in [(100, 0, 64), (100, 1, 64), (100, 1.5, 64), (1000, 1, 64), (100, 1,
8)]:
    D = distinct(N, s, B)
    print(f"N = {N:>4}, s = {s}, B = {B:>2}: 平均 {D:4.1f} 个不同的适配器，每步多读 {D *
83.9 / 1000:.2f} GB（基座的 {D * 83.9 / 16060:.1%}）")
```

```
N =  100, s = 0, B = 64: 平均 47.4 个不同的适配器，每步多读 3.98 GB（基座的 24.8%）
N =  100, s = 1, B = 64: 平均 31.4 个不同的适配器，每步多读 2.64 GB（基座的 16.4%）
N =  100, s = 1.5, B = 64: 平均 18.1 个不同的适配器，每步多读 1.52 GB（基座的 9.5%）
N = 1000, s = 1, B = 64: 平均 43.9 个不同的适配器，每步多读 3.69 GB（基座的 23.0%）
N =  100, s = 1, B =  8: 平均  6.7 个不同的适配器，每步多读 0.56 GB（基座的 3.5%）
```

100 个客户、请求完全均匀时，64 个请求的 batch 里平均有 47 个不同的适配器，每步多读四分之一。流量越集中（s 越大），同一个热门适配器被 batch 里的多个请求共用，D 越小。batch 越大，D 越大，但增长得比 B 慢。

下面的演示可以自己调这几个数，同时看 kernel 调用的次数：

一个 batch 里混着多个 LoRA

⇆ 对照

适配器总数 N 100

热门程度 s 1.0

batch 里的请求 B 64

秩 r 16



一步 decode 要从显存读的权重：基座一遍 + 31.4 个不同适配器各一遍（每个 83.9 MB）

batch 里平均有 **31.4** 个不同的适配器。每步多读 **2.64 GB**，是基座的 **16.4%**；只算读权重，一步从 4.79 ms 变成 5.58 ms。 计算量只多 0.52%（每个 token 只过自己那个适配器）。
kernel 调用：逐个适配器各算一次，每步 32 层 × 7 个线性层 × 2 次矩阵乘 × 32 个适配器 ≈ **14,336** 次；分段矩阵乘（SGMV / BGMV）把一层里所有适配器合成一次，固定 **448** 次。 batch 里的适配器数受 max_loras 限制，超过的请求要等下一步。

图 79-1　一步 decode 要读的权重：灰色是基座，琥珀色是 batch 里出现的各个适配器。计算量几乎不变，读的字节却随不同适配器的个数线性增长。下面一行对比两种实现的 kernel 调用次数。

要点

多 LoRA 服务里，LoRA 的计算量可以忽略，但 decode 每步要把 batch 里出现过的每个适配器都读一遍。代价和“batch 里有几个不同的适配器”成正比，而不是和请求数成正比。

小路怎么算：一个 kernel 算完所有适配器

第二个问题是怎么算小路。最直接的写法是按适配器分组：batch 里属于适配器 1 的请求挑出来，乘 A_1 、B_1 ；属于适配器 2 的挑出来，乘 A_2 、B_2 ……有 D 个适配器，每个线性层就要调用 2D 次矩阵乘法，每次都很小（几个请求 × 4096 × 16）。32 层、7 个线性层、D = 30 时，每一步要发 13,440 次 kernel。第 87 讲会看到，这种小矩阵乘法连 GPU 的一个零头都用不满，而每次发射 kernel 本身还有几微秒的开销（第 89 讲 CUDA Graph 讲的就是这个问题）。

2023 年的 Punica 和 S-LoRA 提出了专门的算子：**分段聚集矩阵乘**（SGMV，segmented gather matrix-vector multiplication）。思路是一次 kernel 处理整个 batch：给每个请求附上它的适配器编号，kernel 里每个线程块自己按编号去显存里找对应的 A、B，算完写回对应的行。decode 时每个请求只有一个 token，这个特例叫 BGMV。这样，不管 batch 里混着多少个适配器，每个线性层的小路都只要固定的两次 kernel 调用。

注意它解决的是“调用次数”和“小矩阵用不满 GPU”的问题，**不减少读的字节**：D 个适配器还是要各读一遍。所以适配器个数的上限依然要管。

显存里放几个：GPU、CPU 两级缓存

适配器的总数可能有成千上万个，不可能全放在显存里。推理引擎一般做两级缓存：

- **显存里的槽位**：最多放 max_loras 个适配器，正在被 batch 使用的必须在这里。
- **CPU 内存里的缓存**：最多 max_cpu_loras 个，从磁盘加载过的先放在这里，需要时拷进显存的槽位；满了按最近最少使用（LRU）淘汰。

一个 84 MB 的适配器，从 CPU 内存经 PCIe 5.0（约 64 GB/s 单向）拷进显存约 1.3 ms。H100 上一步 decode 光读基座权重就要约 4.8 ms，换一个适配器相当于这一步多了四分之一，所以要尽量别每一步都在换；拷贝也可以和计算重叠着做。S-LoRA 更进一步，把适配器的权重和 KV Cache 放在同一个分页的显存池里统一管理（第 75 讲的分页思路），适配器多的时候就少放一些 KV，反过来也一样。

vLLM 的配置里就是这几个数：

third_party/vllm/vllm/config/lora.py 第 25–49 行 / 共 163 行 ⇆ 对照

```
25 LoRADType = Literal["auto", "float16", "bfloat16"]
26 MaxLoRARanks = Literal[1, 8, 16, 32, 64, 128, 256, 320, 512]
27
28
29 @config(config=ConfigDict(arbitrary_types_allowed=True))
30 class LoRAConfig:
31     """Configuration for LoRA."""
32
33     max_lora_rank: MaxLoRARanks = 16
```

1 **支持的秩** 第 26 行

秩只能取这几个值。kernel 要为每种秩准备好对应的分块形状，所以不是任意整数都行。

2 **max_lora_rank 和 max_loras** 第 33–36 行

3 **张量并行和 CPU 缓存** 第 37–44 行

4 **精度** 第 45–46 行

← 上一步 | 看整个文件 | 下一步 →

调度器在放行新请求时检查这个上限（第 78 讲讲过放行的过程）：

third_party/vllm/vllm/v1/core/sched/scheduler.py 第 1129–1140 行 / 共 3934 行 ⇆ 对照

```
1129
1130                # Check that adding the request still respects the max_loras constra
1131                if (
1132                    self.lora_config
1133                    and request.lora_request
1134                    and (
1135                        len(scheduled_loras) == self.lora_config.max_loras
1136                        and request.lora_request.lora_int_id not in scheduled_loras
1137                    )
```

1 **这一步的适配器已经满了就跳过** 第 1129-1140 行

`scheduled_loras` 是这一步已经放进 batch 的适配器集合。新请求要用的适配器如果不在集合里，而集合已经有 `max_loras` 个了，就跳过这个请求，留到以后的步骤。用的是已有的适配器，则不受限制。

这就是“D 的上限”在代码里的样子：宁可让一些请求多等几步，也不让一步里的适配器太多、把这一步拖慢。代价是用冷门适配器的请求排队更久，要在吞吐和公平之间取舍。

← 上一步 | 看整个文件 | 下一步 →

到了真实系统里

- **秩的上限决定显存和速度。**显存槽位按 `max_lora_rank` 预留，混着 r = 8 和 r = 64 的适配器，就按 64 留；秩大的适配器读的字节也多。所以服务方通常会限制客户可用的秩。
- **按适配器调度。**既然代价和 D 成正比，调度时就可以优先把用同一个适配器的请求凑进同一个 batch；集群层面，路由器可以把同一个客户的请求尽量发到同一台机器（第 114 讲的缓存感知路由是同一个思路），每台机器只需要缓存一部分适配器。

- **prefill 的情况不同**。prefill 时每个请求有几百上千个 token，小路的矩阵乘法不再小，计算量也不再可以忽略（但仍然只占约 r / d 的比例）。上面说的“读不免费”主要是 decode 的问题。

容易混淆的地方

- **LoRA 不改变 KV Cache 的大小**。适配器改的是线性层的输出，K、V 的维度和个数都不变。但不同适配器算出的 K、V 不同，所以用不同适配器的请求之间不能共享前缀缓存，哪怕提示一字不差（第 76 讲）。
- **“多 LoRA 服务开销很小”只对一半**。计算开销确实小（r = 16 时 0.5%），但 decode 的读权重开销和 batch 里不同适配器的个数成正比，可能是基座的几分之一。
- **SGMV 不是把适配器合并**。它只是把很多次小矩阵乘法合成一次 kernel 调用，每个请求还是只用自己的适配器。

回顾

多 LoRA 服务不合并适配器：大路所有请求一起算，小路每个请求用自己的 A、B。r = 16 时一个适配器 84 MB，计算只多 0.52%；但 decode 每步要把 batch 里出现过的每个不同适配器都读一遍，100 个均匀分布的客户、64 个请求的 batch，平均 47 个不同的适配器，多读 4 GB，是基座的四分之一。SGMV / BGMV 把一层里所有适配器的小路合成一次 kernel，解决调用次数的问题；显存槽位和 CPU 缓存两级管理适配器；vLLM 用 `max_loras` 限制一步里不同适配器的个数，调度器超过上限就让请求等一等。

这一篇讲到这里，引擎调度的都是“内容不受限”的生成。还有一类常见需求：模型的输出必须是合法的 JSON、必须符合某个格式。模型一个 token 一个 token地生成，怎么保证最后拼出来的东西合法？下一讲：**结构化输出与约束解码**。

第80讲 约束解码

选读

21 分词与 BPE　24 自回归生成　73 解码策略

大模型的输出越来越多地交给程序去读，而不是给人看：从一段话里抽取姓名和年龄，存进数据库；让模型决定调用哪个工具、传什么参数；让一个智能体按固定的格式汇报它的下一步动作。程序需要的是**严格符合格式**的输出，比如一段能被 `json.loads` 直接解析的 JSON，字段名、类型都对。

在提示里写清楚“请只输出 JSON，格式如下……”，大多数时候模型会照做，但不能保证每次都对：它可能在前面加一句“好的，”，可能用 Markdown 的代码块包起来，可能把年龄写成“十二岁”这样的字符串。一万次请求里只要错几次，程序就会出错。这一讲讲推理引擎怎么**从机制上保证**输出符合格式：**约束解码**（constrained decoding）。

在选词之前动手脚

回忆第 24 讲和第 73 讲：模型每一步输出词表上每个 token 的打分（logits），经过 softmax 变成概率，再按某种策略（贪心、Top-p……）选出一个 token，接到后面，进入下一步。

约束解码只在“选词”之前加一步：**检查词表里每个 token，如果把它接上去，输出就不可能再符合格式了，就把它的 logit 设成** $-\infty$（概率变成 0）。剩下的 token 重新归一化，再照常选词。

这样，每一步选出的 token 都保证“到目前为止还合法”；最后只有格式完整时才允许输出结束标记。一步一步下来，整个输出一定符合格式。模型本身一点没改，改的只是每一步的候选范围。

怎么判断一个 token 合不合法

要回答“接上这个 token 以后，还能不能符合格式”，需要把格式写成一台**状态机**。以这一讲的例子为准：

{"name": "〈不含引号的非空字符串〉", "age": 〈非负整数〉}

把它拆成 5 段：开头的固定文字 `{"name": "`；名字（任意不是引号的字符，至少 1 个）；中间的固定文字 `", "age": `；年龄（数字，至少 1 个）；结尾的 `}`。状态就是“现在在第几段、这一段读了几个字符”。每读进一个字符：

- 在固定文字段，这个字符必须和该位置的字符相同，否则不合法。
- 在名字段，不是引号就留在这一段；是引号，并且名字至少有 1 个字符，就进入下一段（这个引号属于中间那段固定文字）。
- 在年龄段，是数字就留下；不是数字，并且已有至少 1 个数字，就交给下一段（结尾的 `}`）。

判断一个 token 合不合法，就把它的字符一个个送进状态机，中途没有出错就合法。比如已经生成了 `{"name": "小明", "age": `，现在在年龄段：

- `"`：年龄段需要数字，不合法。
- `十二`：不是数字，不合法。
- `12`：两个数字，合法。

注意 token 往往不是单个字符，`", ` 这样的 token 一次跨过了好几个字符，甚至跨过了段的边界，所以要逐字符检查。

一个完整的例子

为了看清每一步发生了什么，我们手写一个“模型”：在每个位置给出几个候选 token 的打分，其余 token 的分数很低，忽略不计。这个模型的“习惯”和很多真实模型类似：喜欢先说一句“好的，JSON:”，最后再加个句号，年龄写成“十二岁”。

不加约束，每一步选打分最高的，它会写出：

> 好的，JSON：{"name": "小明", "age": "十二岁"}。

这不是合法的 JSON，`age` 也不是数字。

加约束以后：

- 第 1 步，候选是“好的，”、“JSON:”和 `{`。前两个不合格式（格式要求以 `{` 开头），屏蔽，只剩 `{`，虽然它原来的概率只有 19%，也只能选它。

- 第 2 到 8 步，模型自己想写的和格式一致，没有被屏蔽的 token，照常选择，写出 `{"name": "小明", "age":` 。
- 第 9 步，模型想写引号（准备把年龄写成字符串），引号和 `十二` 都被屏蔽，只剩 `12`，原来的概率只有 13%。
- 第 10 步，模型想接“岁”，屏蔽；选 `}`。
- 第 11 步，模型想在最后加个句号，格式已经完整，后面不能再有字符，屏蔽；选结束。

最终输出 `{"name": "小明", "age": 12}`，`json.loads` 可以直接解析。

下面的演示可以一步一步地看。每一步列出模型的候选和原来的概率，被屏蔽的划掉，右边是屏蔽后重新归一化的概率。切换到“不加约束”对比一下。

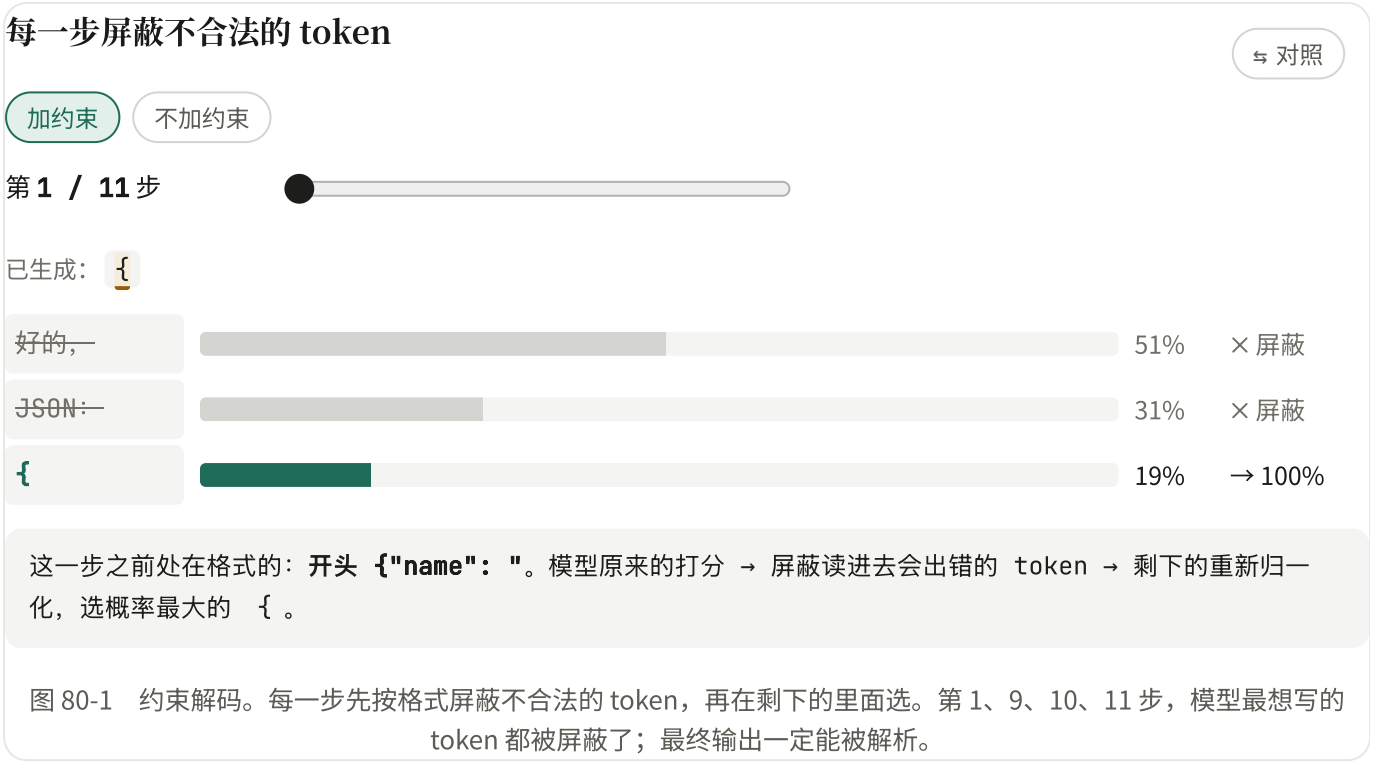


图 80-1　约束解码。每一步先按格式屏蔽不合法的 token，再在剩下的里面选。第 1、9、10、11 步，模型最想写的 token 都被屏蔽了；最终输出一定能被解析。

要点

约束解码在每一步选词之前，把“接上去就不可能符合格式”的 token 的概率置 0。格式写成状态机，用 token 的字符逐个检查。每一步都合法，结束只在格式完整时允许，所以最终的输出一定符合格式。

词表有十几万个，每步都检查一遍？

例子里每步只有两三个候选。真实模型的词表有十几万个 token（Llama-3 是 128256 个），如果每一步都把每个 token 逐字符送进状态机检查一遍，CPU 上要花几十毫秒，比 GPU 算一步还慢。实际的做法有几种：

提前算好。状态机的状态是有限的。对每个状态，提前把“哪些 token 合法”算好，存成一个掩码（mask）。生成时只要查一下当前状态对应的掩码，直接用。Outlines 这个库最先系统地这样做：把 JSON Schema 转成正则表达式，再转成状态机，为每个状态建好 token 的索引。

和 GPU 计算重叠。下一步的掩码只取决于已经生成的 token，可以在 GPU 算这一步的 logits 时，同时在 CPU 上准备好。

处理嵌套的结构。JSON 可以无限嵌套（对象里有数组，数组里又有对象），这种结构用有限状态机描述不了，要用**上下文无关文法**，对应的检查器是带一个栈的状态机（下推自动机）。XGrammar、llguidance 等库专门优化了这件事，大部分 token 的合法性和栈无关，可以提前算好，只有少数要在生成时检查，每步的开销能做到几十微秒。

跳过确定的部分。例子里有好几步，屏蔽以后只剩一个合法的 token（比如 `"age"` 后面只能是 `:`）。既然没得选，就不必让模型算一步，可以直接把这段固定文字接上去。SGLang 把这叫作**跳跃式解码**（jump-forward decoding），对字段名很多的 JSON 能省下不少步数。

代价：可能把模型“逼”到奇怪的地方

约束解码保证的是**格式**，不保证**内容**。屏蔽改变了模型原本的概率分布：第 9 步模型想写“十二岁”，被逼着写数字，它这次写出了 `12`，原来的概率只有 13%。如果模型对内容本来就没把握，或者格式和它的“习惯”差得太远，被逼出来的内容可能是错的，甚至是胡乱的数字和字符串。

所以实践中两件事要一起做：提示里照样写清楚格式和字段的含义，让模型本来就倾向于写出正确的格式；约束解码作为最后的保证，确保程序不会因为格式而出错。

还有一个和分词（第 21 讲）有关的细节：同样的文字可以被切成不同的 token。格式强制模型写某段文字时，它可能被迫用一种平时很少见的切法（比如本来 `", "` 是一个 token，被约束切成了 `"` 和 `, "`），这也会让模型的表现变差。好的实现会让约束尽量符合模型平时的分词习惯。

到了大模型里

API 里的结构化输出。很多大模型 API 提供“结构化输出”：请求里给一个 JSON Schema，返回的内容保证符合它。工具调用（function calling）也一样：模型要调用的工具名和参数，必须符合工具定义的格式。它们背后通常就是约束解码。

推理引擎里的支持。vLLM 和 SGLang 都支持按 JSON Schema、正则表达式或文法约束输出，后端可以选 XGrammar、llguidance、Outlines 等。在推理引擎里，约束解码发生在采样之前，对每个请求按它自己的格式生成掩码，和连续批处理里别的请求互不影响。

格式和推理的冲突。如果要求模型一开口就输出 JSON，它就没有机会先“想一想”（第 49 讲的思维链）。常见的做法是在 Schema 里留一个“推理过程”的字段放在最前面，或者先让模型自由思考，再约束最后的答案部分。

用代码复现

下面的代码实现了上面的状态机和手写的“模型”，先不加约束生成一遍，再加约束生成一遍，打出每一步的候选、被屏蔽的 token（✗）和选择。

```python
import json, math

# 要求的格式：{"name": "<不含引号的非空字符串>", "age": <非负整数>}，拆成几段，一个字符一个字符地检查
SEGS = [("lit", '{"name": "'), ("str", None), ("lit", '", "age": '), ("num", None), ("lit", "}")]

def advance(state, ch):
    """状态 = (第几段, 这一段已读了几个字符)。读入一个字符，返回新状态；不合法返回 None。"""
    i, k = state
    if i == len(SEGS): return None                          # 格式已经完整，后面不能再有字符
    kind, lit = SEGS[i]
    if kind == "lit":
        if lit[k] != ch: return None
        return (i + 1, 0) if k + 1 == len(lit) else (i, k + 1)
    ok = ch != '"' if kind == "str" else ch.isdigit()
    if ok: return (i, k + 1)
    return advance((i + 1, 0), ch) if k >= 1 else None      # 这一段结束（至少 1 个字符），交给下一段

def allowed(state, tok):
    if tok == "<结束>": return state[0] == len(SEGS)          # 只有格式完整了才能结束
    for ch in tok:
        state = advance(state, ch)
        if state is None: return False
    return True

# 一个手写的“模型”：根据已经生成的文字，给出几个候选 token 的打分（logits），其余 token 的分数很低，省略
BEFORE = {"": [("好的，", 3.0), ("JSON：", 2.5), ("{", 2.0)], "好的，": [("JSON：", 3.0), ("{", 2.5)], "好的，JSON：": [("{", 4.0)]}
TABLE = {                                                   # 进入 JSON 以后，只看第一个 { 之后的部分
    "": [('"name"', 4.0), ('"', 1.0)],
    '"name"': [(": ", 4.0), (":", 2.0)],
    '"name": ': [('"', 4.0)],
    '"name": "': [("小明", 3.5), ("小红", 3.0)],
    '"name": "小明': [('", ', 4.0), ('"', 2.0)],
    '"name": "小明", ': [('"age"', 4.0)],
    '"name": "小明", "age"': [(": ", 4.0)],
    '"name": "小明", "age": ': [('"', 3.0), ("十二", 2.2), ("12", 1.5)],
    '"name": "小明", "age": "': [("十二", 3.0), ("12", 1.0)],
    '"name": "小明", "age": "十二': [("岁", 2.5), ('"', 2.0)],
    '"name": "小明", "age": "十二岁': [('"', 4.0)],
    '"name": "小明", "age": "十二岁"': [("}", 4.0)],
    '"name": "小明", "age": 12': [("岁", 2.4), ("}", 2.0), ("0", 0.5)],
```

```
    '"name": "小明", "age": "十二岁"}': [("。", 3.0), ("<结束>", 2.5)],
    '"name": "小明", "age": 12}': [("。", 3.0), ("<结束>", 2.5)],
}
def model(text):
    if "{" not in text: return BEFORE[text]
    return TABLE.get(text[text.index("{") + 1:], [("<结束>", 3.0)])

def generate(constrained):
    text, state = "", (0, 0)
    while True:
        cands = model(text)
        z = sum(math.exp(s) for _, s in cands)
        ok = [(t, s) for t, s in cands if not constrained or allowed(state, t)]   #
屏蔽：不合法的 token 概率置 0
        tok = max(ok, key=lambda x: x[1])[0]
        if constrained:
            shown = "  ".join(f"{t!r}{'' if allowed(state, t) else '✗'}" for t, _
in cands)
            print(f"  候选 {shown}  → 选 {tok!r} (原概率 {math.exp(dict(cands)[tok])
/ z:.0%}) ")
        if tok == "<结束>": return text
        text += tok
        if constrained:
            for ch in tok: state = advance(state, ch)

print("不加约束：", repr(generate(False)))
print("加约束，每一步：")
out = generate(True)
print("加约束：", repr(out), "→", json.loads(out))
```

输出：

```
不加约束：  '好的，JSON：{"name": "小明", "age": "十二岁"}。'
加约束，每一步：
  候选 '好的，'✗  'JSON：'✗  '{'  → 选 '{'（原概率 19%）
  候选 '"name"'  '"'  → 选 '"name"'（原概率 95%）
  候选 '：'  ':'  → 选 '：'（原概率 88%）
  候选 '"'  → 选 '"'（原概率 100%）
  候选 '小明'  '小红'  → 选 '小明'（原概率 62%）
  候选 '", '  '"'  → 选 '", '（原概率 88%）
  候选 '"age"'  → 选 '"age"'（原概率 100%）
  候选 '：'  → 选 '：'（原概率 100%）
  候选 '"'✗  '十二'✗  '12'  → 选 '12'（原概率 13%）
  候选 '岁'✗  '}'  '0'  → 选 '}'（原概率 37%）
  候选 '。'✗  '<结束>'  → 选 '<结束>'（原概率 38%）
加约束：  '{"name": "小明", "age": 12}' → {'name': '小明', 'age': 12}
```

第一行是不加约束的输出，前面多了“好的，JSON：”，后面多了句号，年龄是字符串。下面 11 行是加约束时的每一步，被屏蔽的 token 后面有 ✗，和演示一致。最后一行是加约束的输出，`json.loads` 成功解析出了 `{'name': '小明', 'age': 12}`。

容易混淆的地方

1. **“JSON 模式”和“按 Schema 约束”不一样**。有的 API 的 JSON 模式只保证输出是合法的 JSON，不保证字段名和类型对；按 Schema 约束才保证结构完全符合。
2. **保证格式，不保证内容正确**。约束解码不会让模型知道它不知道的东西，抽取出来的名字、数字仍然可能是错的。
3. **长度上限会打断格式**。如果设的最大生成长度不够，输出写到一半被截断，仍然不是合法的 JSON。约束解码只保证“没被截断时”的合法性。

回顾

程序读模型的输出时，需要严格符合格式。光靠提示，模型偶尔会加多余的话、写错类型。约束解码在每一步选词之前，把接上去就不可能符合格式的 token 的概率置 0：格式写成状态机，token 的字符逐个送进去检查，只有格式完整时才允许结束。例子里，手写的模型自己会先说“好的，JSON：”、最后加句号、把年龄写成“十二岁”，加约束后输出 `{"name": "小明", "age": 12}`。真实的词表有十几万个 token，所以要为每个状态提前算好掩码，嵌套结构用下推自动机，确定的部分直接跳过。它保证的是格式，不是内容；屏蔽也会改变模型原来的概率分布，提示里仍要把格式讲清楚。

到这里，推理引擎里的各个机制都讲过了：连续批处理、分页、前缀缓存、分块 prefill、调度、约束解码。一个请求从发进来到收到最后一个字，到底依次经过了哪些模块？下一讲：**一个请求的完整生命周期**。

第81讲 请求生命周期

70 TTFT、TPOT 与吞吐　74 Continuous Batching　75 PagedAttention　76 Prefix Cache　77 Chunked Prefill　78 调度与抢占

前面六讲分别讲了推理引擎里的一个个机制：连续批处理、分页、前缀缓存、分块 prefill、调度与抢占、约束解码。它们是零件。这一讲把零件装回去：跟着一个请求，从用户按下发送键，到收到最后一个字，看它依次经过哪些模块、每个模块手里拿着什么数据、做了什么决定。

读完这一讲，再去看 vLLM 或 SGLang 的源码、日志和监控指标，就知道每一块对应的是哪个环节。

全景：三个进程、九个模块

以 vLLM 为原型（SGLang 的结构大同小异），一个推理服务可以分成三个部分，各自运行在不同的进程里：

- **前端进程**：API 服务器，以及套模板、分词。它面对的是 HTTP 请求和文字。
- **引擎核心**：调度器和 KV 块管理器。它面对的是一个个请求的状态：排在哪个队列、用了哪些块、prefill 做到了哪里。
- **GPU 工作进程**：准备输入、跑模型、采样。它面对的是张量。多卡并行时（第 13 篇），每张卡一个工作进程。

最后还有反分词，把 token 变回文字，从前端送回给用户。

为什么要分成几个进程？因为它们干的活性质完全不同。HTTP 请求的收发、分词、反分词都是 CPU 上琐碎的字符串处理，量大而且时快时慢；调度器的循环则要求稳定、快速地一步接一步，不能被这些琐事打断，否则 GPU 就会空等。Python 同一个进程里同一时刻只能有一个线程在执行 Python 代码，所以把它们拆到不同的进程里，彼此用消息队列通信（vLLM 和 SGLang 都用 ZeroMQ）。

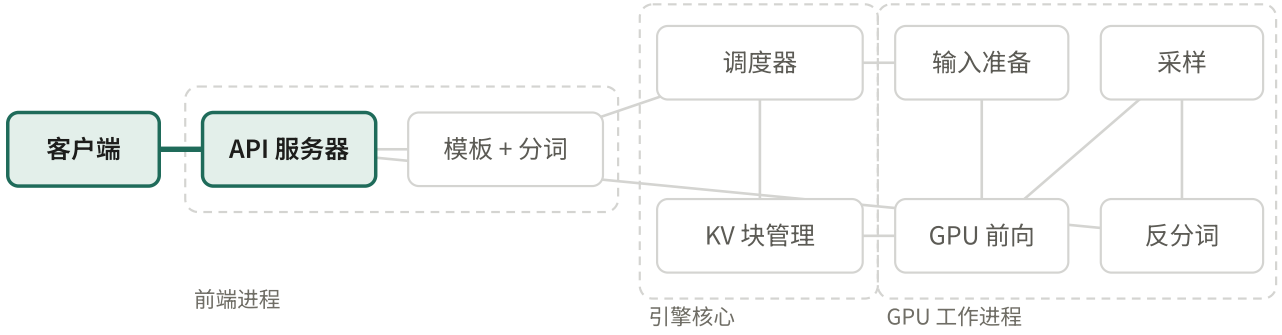
下面的示意图画出了这九个模块。一步步点“下一步”，跟着一个请求走一遍。

一个请求流经的模块

⇆ 对照

阶段 1 / 10

← 上一步　下一步 →



1. 客户端发来请求

```
POST /v1/chat/completions  {"messages": [{"role": "user", "content": "什么是注意力? "}], "stream": true, "max_tokens": 200}
```

点一个模块，看它负责什么。

图 81-1　一个请求流经推理引擎的各个模块。虚线框是进程的划分。第 4 到 9 阶段里，调度、输入准备、前向、采样这个循环每生成一个 token 转一圈。

进门：API 服务器和分词

请求从一个 HTTP 调用开始。大多数推理引擎都提供和 OpenAI 兼容的接口，比如 `/v1/chat/completions`：请求里有对话消息、要不要流式返回（`stream`）、最多生成多少个 token（`max_tokens`）、温度和 Top-p（第 73 讲）、可能还有一个 JSON Schema（第 80 讲）。

API 服务器检查这些参数，给请求分配一个编号，建一个输出通道，留着之后把生成的文字推回去。

接着是**套模板和分词**。对话消息要按模型训练时用的聊天模板（第 47 讲）拼成一段文字，加上 `◁start_header_id▷user◁end_header_id▷` 之类的特殊标记，再用分词器（第 21 讲）切成 token id 序列。从这里开始，引擎里流动的就只有数字了。

分词本身很快（毫秒以内），但请求很多、提示很长时，它占的 CPU 也不少，这正是它被放在前端进程里的原因。

排队和放行：调度器与 KV 块管理

token 序列送到引擎核心，请求进入**等待队列**。

引擎核心在一个循环里运转，每转一圈就是一步。每一步开始时，调度器按第 74、77、78 讲的规则做决定：

1. 运行中的请求，已经在 decode 的，每个排 1 个 token；需要新块的，向 KV 块管理器要；块不够，就抢占。
2. 用剩下的 token 预算，排正在 prefill 的请求（长提示切块）。
3. 还有预算和空闲块，就从等待队列里放新请求进来。

放新请求时，KV 块管理器先查**前缀缓存**（第 76 讲）：系统提示或者多轮对话的历史，如果有现成的块，直接放进这个请求的块表，引用计数加一；没命中的部分，按需分配新块（第 75 讲）。

调度器的输出是一张“这一步的清单”：哪些请求参加，每个处理几个 token，它们的块表是什么。

一步前向：输入准备、GPU、采样

清单送到 GPU 工作进程。**输入准备**（vLLM 里叫 model runner）把清单变成张量：

- 所有请求这一步的 token 首尾相接，拼成一个长序列（第 74 讲）；
- 每个 token 的位置编号（用于 RoPE，第 37 讲）；
- 每个请求的块表，和每个新 token 的 k、v 要写到哪个块的哪个位置；
- 每个请求在长序列里从哪里开始、到哪里结束，供注意力区分请求。

然后是 **GPU 前向**：32 层 Transformer 依次计算。线性层（QKV 投影、输出投影、FFN）对长序列里所有 token 一起做矩阵乘法；注意力按块表从 KV Cache 里读每个请求自己的历史，把新 token 的 k、v 写进去（第 85 讲讲这个算子）。最后，只取每个请求**最后一个位置**的输出，算出词表上的 logits。

采样拿到 logits，按每个请求自己的参数处理：有格式约束的先加掩码，再除以温度、做 Top-p 截断，最后抽出一个 token id。对刚做完 prefill 的请求，这是它的第一个 token，**TTFT 到这里就结束了**（第 70 讲）。

出门：反分词和流式返回

新的 token id 送回前端，**反分词**把它接到已经生成的部分，转回文字。这一步不像看上去那么简单：一个汉字可能被切成好几个 token（第 21 讲的字节级 BPE），单独一个 token 可能解码不出完整的字，要等后面的 token 到了再一起输出。所以反分词是“增量”的：每次只输出新确定下来的那部分文字。

同时还要检查结束条件：生成了结束标记、遇到了用户指定的停止词、达到了 `max_tokens`。

文字片段通过 SSE（服务器推送事件）推给客户端，用户看到第一个字。

循环，直到结束

接下来，这个请求每一步都在调度器的清单上：排 1 个 decode token，必要时多要一块；GPU 前向；采样；反分词；推送。这个循环每秒转几十到上百次，每两次推送之间的间隔，就是 TPOT。

生成结束时，请求离开运行队列，块的引用计数减一。提示部分的块仍然留在前缀缓存里，后来的请求可能还会命中它；没人用、池子又紧张时，再按 LRU 腾出来。最后，API 服务器发出结束标志和用量统计（输入、输出各多少个 token，计费就按它算）。

要点

一个请求依次经过：API 服务器、套模板与分词、调度器（和 KV 块管理）、输入准备、GPU 前向、采样、反分词、流式返回。中间“调度 → 输入准备 → 前向 → 采样”是一个每步都转的循环，所有请求共用它；每个请求每转一圈多一个 token。

时间都花在哪里

把这条路径和第 70 讲的指标对应起来：

- **TTFT** = 排队时间 + prefill 时间（长提示可能分好几步）+ 第一个 token 的采样和返回。负载高时，排队时间往往是 TTFT 的大头。
- **TPOT** ≈ 每一步的时间。它取决于这一步一起算的有多少请求、多少 prefill token（第 70、77 讲），也包括调度器和输入准备在 CPU 上花的时间。
- **总延迟** = TTFT +（输出 token 数 − 1）× TPOT。

CPU 上的时间值得单独说一句。decode 一步在 GPU 上只要几毫秒，调度、输入准备、采样、反分词如果在 CPU 上加起来也花上一两毫秒，而且是和 GPU 串行进行的，GPU 就有相当一部分时间在空等。所以推理引擎在这里花了很多功夫：把调度和 GPU 计算重叠起来（GPU 在算第 n 步时，CPU 准备第 n + 1 步），用 CUDA Graph 减少启动开销（第 89 讲），把反分词挪出主循环。

用代码复现

下面是一个极简的推理引擎，把这一讲的几个模块串起来：分词（一个字当一个 token）、等待队列、带前缀缓存的块分配、先 decode 后 prefill 的 token 预算、采样（为了简单，“模型”直接按预先写好的回答逐字输出）、流式返回、结束时归还块。两个请求共用系统提示“你是助手。”，R1 在第 0 步到达，R2 在第 3 步到达。代码打出 R1 前几步和 R2 全程的事件。

```
# 一个极简的推理引擎：一个字当一个 token，“模型”按预先写好的回答逐字输出，只为看清各个模块怎么配合
B, POOL, BUDGET = 4, 12, 10                                  # 每块 4 个 token，池子 12 块，每步最多处理 10 个 token
SYS = "你是助手。"
REPLY = {"什么是注意力？": "按相关程度收集信息。", "什么是残差？": "把输入加回输出。"}

class Req:
    def __init__(s, rid, text):
        s.rid, s.q = rid, text
        s.prompt = list(SYS + text)                          # ① 套对话模板、分词
        s.out, s.done_pre, s.blocks = [], 0, []

cache, free = {}, list(range(POOL))                          # 前缀缓存：前缀 → 块号；空闲块
waiting, running, log = [], [], []

def admit(r, step):
    """② 调度器放进一个请求：查前缀缓存，命中的整块直接复用，其余按需分配。"""
    hit = 0
    while (hit + 1) * B ⩽ len(r.prompt) and "".join(r.prompt[:(hit + 1) * B]) in cache:
        r.blocks.append(cache["".join(r.prompt[:(hit + 1) * B])]); hit += 1
    r.done_pre = hit * B
    log.append((step, r.rid, f"调度器：放进运行队列，前缀缓存命中 {hit} 块（{hit * B} 个 token 不用算）"))

def ensure_blocks(r, n_tokens):
    while len(r.blocks) * B < n_tokens: r.blocks.append(free.pop(0))

def step(t):
    budget, batch = BUDGET, []
    for r in running:                                        # ③ 先排 decode：每个已经 prefill 完的请求 1 个 token
        if r.done_pre == len(r.prompt): batch.append((r, 1)); budget -= 1
    while waiting and len(free) ⩾ 2: r = waiting.pop(0); admit(r, t); running.append(r)
    for r in running:                                        # ④ 再用剩下的预算排 prefill（切块）
        left = len(r.prompt) - r.done_pre
        if left and budget > 0:
            c = min(left, budget); batch.append((r, c)); budget -= c
    for r, n in batch:                                       # ⑤ 模型执行：分配块、写 KV、算出下一个 token
        is_pre = r.done_pre < len(r.prompt)
        ensure_blocks(r, (r.done_pre + n) if is_pre else len(r.prompt) + len(r.out)
```

```
+ 1)
        if is_pre:
            r.done_pre += n
            for k in range(r.done_pre // B):              # 装满的提示块登记进前缀缓存
                cache.setdefault("".join(r.prompt[:(k + 1) * B]), r.blocks[k])
            log.append((t, r.rid, f"模型：prefill {n} 个 token
 ({r.done_pre}/{len(r.prompt)}) ，块表 {r.blocks}"))
            if r.done_pre < len(r.prompt): continue
        tok = REPLY[r.q][len(r.out)]                     # ⑥ 采样（这里直接取预先写好
的回答）
        r.out.append(tok)
        log.append((t, r.rid, f"采样 → “{tok}”，反分词后流式发给用户：
{''.join(r.out)}"))
        if len(r.out) == len(REPLY[r.q]):                # ⑦ 结束：块还回去（提示块仍
留在前缀缓存里，供后来者命中）
            running.remove(r); free.extend(b for b in r.blocks if b not in
cache.values())
            log.append((t, r.rid, f"结束，共 {len(r.out)} 个 token"))

waiting.append(Req("R1", "什么是注意力？")); log.append((0, "R1", f"API 收到请求，分词
得到 {len(SYS + '什么是注意力？')} 个 token，进入等待队列"))
for t in range(14):
    if t == 3: waiting.append(Req("R2", "什么是残差？")); log.append((3, "R2", f"API
收到请求，分词得到 {len(SYS + '什么是残差？')} 个 token，进入等待队列"))
    step(t)
for t, rid, msg in log:
    if rid == "R2" or t ⩽ 2: print(f"第 {t:2d} 步 {rid}：{msg}")
```

输出：

```
第  0 步 R1：API 收到请求，分词得到 12 个 token，进入等待队列
第  0 步 R1：调度器：放进运行队列，前缀缓存命中 0 块（0 个 token 不用算）
第  0 步 R1：模型：prefill 10 个 token（10/12），块表 [0, 1, 2]
第  1 步 R1：模型：prefill 2 个 token（12/12），块表 [0, 1, 2]
第  1 步 R1：采样 → “按”，反分词后流式发给用户：按
第  2 步 R1：采样 → “相”，反分词后流式发给用户：按相
第  3 步 R2：API 收到请求，分词得到 11 个 token，进入等待队列
第  3 步 R2：调度器：放进运行队列，前缀缓存命中 2 块（8 个 token 不用算）
第  3 步 R2：模型：prefill 3 个 token（11/11），块表 [0, 1, 4]
第  3 步 R2：采样 → “把”，反分词后流式发给用户：把
第  4 步 R2：采样 → “输”，反分词后流式发给用户：把输
第  5 步 R2：采样 → “入”，反分词后流式发给用户：把输入
第  6 步 R2：采样 → “加”，反分词后流式发给用户：把输入加
第  7 步 R2：采样 → “回”，反分词后流式发给用户：把输入加回
第  8 步 R2：采样 → “输”，反分词后流式发给用户：把输入加回输
第  9 步 R2：采样 → “出”，反分词后流式发给用户：把输入加回输出
第 10 步 R2：采样 → “。”，反分词后流式发给用户：把输入加回输出。
第 10 步 R2：结束，共 8 个 token
```

看 R1：它的提示有 12 个 token，每步预算 10 个，所以 prefill 被切成两块，第 0 步 10 个、第 1 步 2 个，第 1 步结束时出了第一个字“按”。

看 R2：它在第 3 步到达，和 R1 共用“你是助手。什么是”这 8 个字，前两块（块 0、1）直接命中前缀缓存，只需要 prefill 剩下的 3 个 token，当步就出了第一个字。之后每步一个字，第 10 步出完“把输入加回输出。”，结束。它的块表 [0, 1, 4] 里，块 0、1 是和 R1 共享的。

到了源码里

如果想读源码，下面是这些模块在 vLLM（V1 引擎）和 SGLang 里的大致位置（文件会随版本调整，以当前版本为准）：

模块	vLLM	SGLang
API 服务器	`vllm/entrypoints/openai/`	`sglang/srt/entrypoints/http_server.py`
模板、分词	前端的输入处理	`TokenizerManager`
调度器	`vllm/v1/core/sched/scheduler.py`	`sglang/srt/managers/scheduler.py`
KV 块管理、前缀缓存	`vllm/v1/core/kv_cache_manager.py`	`sglang/srt/mem_cache/radix_cache.py`
输入准备、前向	`vllm/v1/worker/gpu_model_runner.py`	`ModelRunner`
采样	`vllm/v1/sample/sampler.py`	`Sampler`
反分词	前端的输出处理	`DetokenizerManager`

读的时候抓住一条主线：引擎核心的循环每一步调用一次调度器，得到清单，交给 GPU 工作进程执行，拿回新 token 更新每个请求的状态。其他的代码，大多是这条主线上的细节。

容易混淆的地方

1. **“推理引擎”不等于“模型”**。模型只是 GPU 前向那一个模块；其余的模块决定了模型以什么方式被调用。同一个模型，换一个引擎、换一组参数，吞吐和延迟可以差好几倍。
2. **停止词在反分词时检查**。停止词是文字，不是 token，只有转回文字后才能判断。所以停止词出现后，引擎可能已经多算了一两个 token，只是没有返回给用户。
3. **流式返回不改变计算**。`stream` 只是让文字边生成边送出；不流式时，引擎同样一个 token 一个 token 地生成，只是攒齐了再一次返回。TTFT 对用户的意义只在流式时才明显。

回顾

一个请求从 HTTP 调用开始，在前端进程里套上聊天模板、分词成 token id；进入引擎核心的等待队列，调度器在有预算、有块时放它进来，KV 块管理器查前缀缓存、分配块；GPU 工作进程把这一步所有请求的 token 拼成长序列，跑一遍前向，按块表读写 KV Cache，取最后位置的 logits 采样；新 token 回到前端，增量反分词、检查停止条件，通过 SSE 推给用户。“调度 → 前向 → 采样”的循环每步转一圈，所有请求共用；结束时块的引用计数减一，提示块留在前缀缓存里。三个进程分开，是为了让 CPU 上的琐事不打断 GPU 的循环。

推理引擎搭好了，它到底有多快？“吞吐 1 万 token/s”这样的数字是怎么测出来的，怎么测才可信？下一讲：**推理压测实践**。

第82讲 推理压测实践

70 TTFT、TPOT 与吞吐　72 推理成本模型　74 Continuous Batching　77 Chunked Prefill　81 请求生命周期

“我们的推理服务，单卡吞吐 1 万 token/s。”这句话本身几乎没有信息量。是什么模型、什么显卡？请求的提示多长、输出多长？同时有多少个请求？这时每个用户的 TTFT 和 TPOT 是多少？第 70 讲说过，吞吐和延迟是此消彼长的：把并发堆得足够高，吞吐总能很好看，代价是每个用户都要等很久。

所以一次可信的压测（benchmark），给出的不应该是一个数，而是一条**曲线**：在不同的压力下，吞吐和延迟分别是多少，以及在满足延迟要求的前提下，最多能承受多大的压力。这一讲讲怎么做出这条曲线，以及压测里常见的坑。

压测在测什么

先把第 70 讲的指标放到压测的语境里：

- **输出吞吐**：每秒生成多少个输出 token（token/s）。有时也看输入吞吐，或者每秒完成多少个请求。
- **TTFT**：从请求发出到收到第一个 token。它包括**排队**的时间，负载高时排队是大头（第 81 讲）。
- **TPOT**：生成阶段，相邻两个 token 的平均间隔。有的工具也报告 ITL（inter-token latency），即每两个 token 之间的间隔，不先对每个请求取平均。
- **端到端延迟**：从发出到收到最后一个 token。

延迟要看**分布**，不能只看平均：至少报告中位数（P50）和 P99（100 个请求里最慢的那 1 个大约有多慢）。用户的体验往往由尾部决定，一个平均 TTFT 50 ms、P99 却是 3 秒的服务，每一百次请求就有一次让人觉得“卡死了”。

怎么加压

压测工具扮演成大量的用户，向服务发请求。有两种加压方式：

按到达率加压（开环）：按设定的速率发请求，比如平均每秒 20 个，不管前面的请求有没有完成。到达的时间间隔通常按**泊松过程**随机生成（间隔服从指数分布），因为真实的用户是各自独立、随机地发请求的。这种方式最接近线上服务的真实情况：用户不会因为你的服务慢了就少发请求。

按并发数加压（闭环）：固定有 c 个“用户”，每个用户收到回答后立刻发下一个请求。服务越慢，发请求也越慢，所以它永远不会把服务压垮。这种方式简单，适合测“同时 c 个请求时的表现”（第 70 讲的表格就是这种），但它会掩盖排队的问题。

可信的压测通常用前一种：**从低到高逐级提高到达率**，每一级跑足够多的请求，记录各项指标，画成曲线。

一条典型的曲线

我们用一个模拟的服务来做一次压测：Llama-3-8B、一张 H100，引擎使用连续批处理和分块 prefill（每步预算 2048 个 token），KV Cache 池约 40 万个 token（第 71 讲），每一步的耗时按第 68 讲的方法估算。请求的提示长度在 200 到 2000 之间、输出在 50 到 500 之间随机，600 个请求按泊松过程到达。

逐级加压的结果（代码在后面）：

- **每秒 5 个**：吞吐 1349 token/s，TTFT 中位 21 ms、P99 50 ms，TPOT 中位 5.5 ms。服务很闲，每个请求几乎都是“单独”在跑。
- **每秒 20 个**：吞吐 5011 token/s，TTFT P99 82 ms，TPOT 中位 9.1 ms。吞吐随压力线性增长，TPOT 开始慢慢变长，因为一起 decode 的请求多了（第 70 讲）。
- **每秒 40 个**：吞吐 8316 token/s，TTFT P99 197 ms，TPOT 中位 21.0 ms。
- **每秒 45 个**：吞吐 8570 token/s，几乎不再增长；TTFT P99 一下子跳到了 1306 ms。
- **每秒 60 个**：吞吐 8780 token/s，只比每秒 40 个时多了 6%；TTFT 中位 906 ms，P99 达到 4176 ms。

曲线的形状是这样的：压力低时，吞吐随到达率线性增长，延迟几乎不变；到了某个点（这里是每秒四十多个），服务的处理能力饱和了，吞吐不再增长，多出来的请求只能排队，TTFT 急剧上升。这个拐点就是服务的**容量**。

下面的演示画出了这条曲线。拖动到达率看每一点的指标；调整延迟目标，看满足目标的最大到达率是多少。

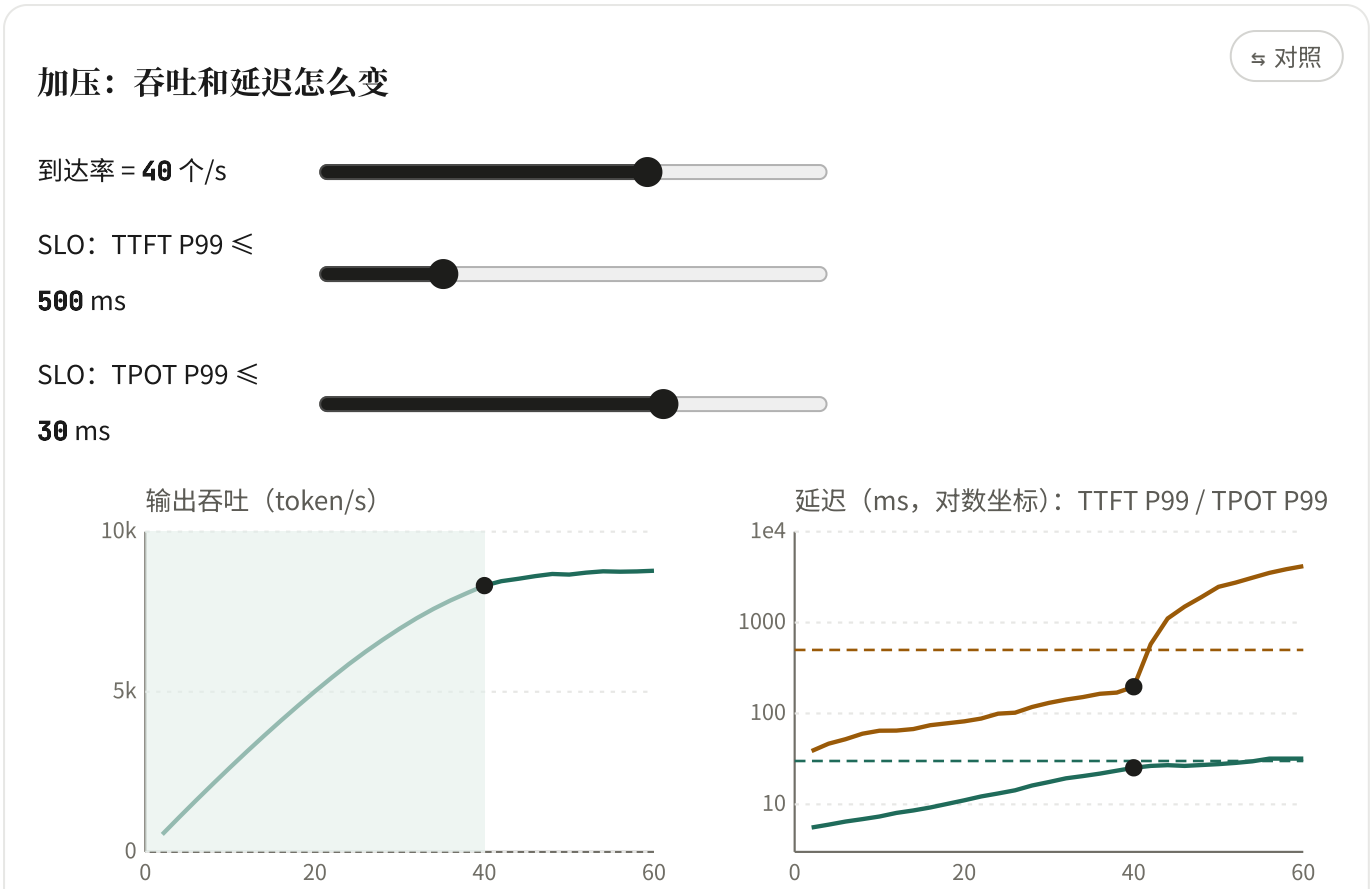


图 82-1　吞吐-延迟曲线。左边：吞吐随到达率先线性增长，然后饱和；右边（对数坐标）：TTFT P99 在饱和点附近急剧上升。虚线是 SLO，阴影部分是满足 SLO 的范围。

goodput：满足要求的吞吐

线上服务通常有明确的延迟目标，叫 SLO（服务等级目标），比如“TTFT 的 P99 不超过 500 ms，TPOT 的 P99 不超过 30 ms”。超过了，就算服务没有挂，用户体验也不合格。

所以真正有意义的是：**在满足 SLO 的前提下，最大的吞吐是多少**。这个数叫 goodput（有效吞吐，这个说法来自 2024 年的 DistServe 论文）。按上面的 SLO，模拟的服务在每秒 40 个请求时还满足要求（TTFT P99 197 ms、TPOT P99 25.2 ms），每秒 42 个时 TTFT P99 就超了，所以 goodput 约是 8316 token/s。每秒 60 个时吞吐虽然更高，但 TTFT P99 有 4 秒多，没有意义。

goodput 也直接决定成本（第 72 讲）：一张卡能承担的“合格”流量，就是 goodput。比较两种配置、两个引擎、两种硬件，应该比 goodput，而不是比最大吞吐。

> **要点**
> 压测要逐级提高到达率，测出吞吐和延迟分布（中位数和 P99）的曲线。吞吐会在某个到达率饱和，再加压只会让请求排队、TTFT 暴涨。满足 SLO 的最大吞吐叫 goodput，它才是一次压测最该给出的数字。

压测里常见的坑

数据集不对。提示和输出的长度分布，决定了 prefill 和 decode 的比例，也决定了 KV Cache 的用量，结果可以差好几倍。用和线上相近的长度分布，或者公开的对话数据集（比如 ShareGPT）；固定长度的随机 token 只适合测某个特定的点。

输出长度不受控。模型可能提前生成结束标记，导致实际输出比设定的短得多。很多工具提供“忽略结束标记”的选项，强制每个请求生成到指定长度，保证不同配置之间可比。

前缀缓存“作弊”。如果所有请求的提示都相同，或者有很长的公共前缀，前缀缓存（第 76 讲）会让 prefill 几乎不花时间，结果好得不真实。要么用彼此不同的提示，要么明确地把命中率作为测试条件写出来。

没有预热。刚启动的服务要编译、要捕获 CUDA Graph（第 89 讲）、要分配显存，前几个请求会特别慢。压测前先跑一批请求预热，或者丢掉最开始的结果。

请求太少、时间太短。P99 需要足够多的样本，600 个请求的 P99 只由最慢的六七个请求决定。另外，过载时队列会一直变长，TTFT 取决于压测跑了多久：跑 10 秒和跑 10 分钟，过载区的 TTFT 会差很多。所以过载区的延迟数字只说明“已经过载”，不要拿来比较。

压测机自己成了瓶颈。客户端要处理成千上万个流式连接，解析每个 token 的返回。如果压测程序自己的 CPU 打满了，测出来的延迟里就混进了客户端的时间。

只报最好看的那个数。完整地写清楚：模型、精度、硬件、引擎及版本、主要参数（批大小上限、token 预算、是否开前缀缓存）、数据集、加压方式、每一级的吞吐和 P50、P99。别人才能复现，才能比较。

到了大模型里

常用的工具。vLLM 和 SGLang 都自带压测脚本（`vllm bench serve`、`python -m sglang.bench_serving`），支持按到达率或并发加压，输出吞吐、TTFT、TPOT、ITL 的分布。NVIDIA 的 GenAI-Perf 等工具也做同样的事。它们都通过 OpenAI 兼容的接口发请求，可以测任何引擎。

离线吞吐和在线服务分开测。离线批量处理（比如给一百万篇文档打标签）只关心总吞吐，可以把所有请求一次性塞进去，不关心延迟；在线服务要按上面的方法测曲线和 goodput。两者的最优配置往往不同。

把测出来的数和理论对上。测出的结果要和第 72 讲的成本模型、第 67 讲的理论值对照：如果实测吞吐只有理论值的一半，就该用 Profiler（第 66 讲）去找时间花在了哪里。

用代码复现

下面的代码模拟了这一讲的压测：生成泊松到达的请求，按连续批处理和分块 prefill 的规则模拟服务（每步耗时按第 68 讲估算），逐级提高到达率，统计输出吞吐和 TTFT、TPOT 的中位数与 P99。为了每次结果一样，随机数用了固定的种子。

```python
import math
# 被测的“服务”：Llama-3-8B + 一张 H100，连续批处理 + 分块 prefill（每步预算 2048），KV 池约 40 万个 token（第 71 讲）
N, NE, L, d = 8.03e9, 128256 * 4096, 32, 4096
W, KV, F, BW = N * 2, 2 * 32 * 8 * 128 * 2, 989e12, 3.35e12
POOL, BUDGET, MAX_SEQS = 403_865, 2048, 256

def lcg(seed):                                          # 固定种子的伪随机数，保证每次运行结果一样
    while True:
        seed = (seed * 1103515245 + 12345) % 2**31
        yield seed / 2**31

def workload(rate, n=600):
    """n 个请求，按泊松过程到达（平均每秒 rate 个）；提示 200-2000 个 token，输出 50-500 个。"""
    r, t, reqs = lcg(42), 0.0, []
    for _ in range(n):
        t += -math.log(1 - next(r)) / rate
        reqs.append({"at": t, "p": 200 + int(next(r) * 1800), "o": 50 + int(next(r) * 450)})
    return reqs

def serve(reqs):
    now, i, waiting, running, used = 0.0, 0, [], [], 0
    while i < len(reqs) or waiting or running:
        while i < len(reqs) and reqs[i]["at"] <= now: waiting.append(reqs[i]); i += 1
        if not waiting and not running: now = reqs[i]["at"]; continue
        while waiting and len(running) < MAX_SEQS and used + waiting[0]["p"] + waiting[0]["o"] <= POOL:
            q = waiting.pop(0); q.update(done=0, gen=0); running.append(q); used += q["p"] + q["o"]
        budget, flops, ctx, work = BUDGET, 0.0, 0, []
        for q in running:                               # 先 decode，再用剩下的预算做 prefill
            if q["done"] == q["p"]: work.append((q, 1)); budget -= 1
        for q in running:
            if q["done"] < q["p"] and budget > 0:
                c = min(budget, q["p"] - q["done"]); work.append((q, c)); budget -= c
        for q, c in work:
            past = q["done"] + q["gen"]
            flops += 2 * (N - NE) * c + 4 * L * d * c * (past + c / 2); ctx += past + c
        now += max(flops / F, (W + KV * ctx) / BW)      # 这一步的耗时（第 68 讲）
```

```
        for q, c in work:
            if q["done"] < q["p"]:
                q["done"] += c
                if q["done"] < q["p"]: continue
            q["gen"] += 1
            if q["gen"] == 1: q["first"] = now
            if q["gen"] == q["o"]: q["end"] = now; running.remove(q); used -=
q["p"] + q["o"]

def pct(xs, p): xs = sorted(xs); return xs[min(len(xs) - 1, int(p / 100 *
len(xs)))]

print("到达率(个/s) | 输出吞吐(token/s) | TTFT 中位/P99 (ms) | TPOT 中位/P99 (ms)")
for rate in [5, 10, 20, 30, 40, 45, 50, 60]:
    reqs = workload(rate); serve(reqs)
    ttft = [(q["first"] - q["at"]) * 1e3 for q in reqs]
    tpot = [(q["end"] - q["first"]) / (q["o"] - 1) * 1e3 for q in reqs]
    span = max(q["end"] for q in reqs) - reqs[0]["at"]
    print(f"{rate:9d}    | {sum(q['o'] for q in reqs) / span:12,.0f}    |
{pct(ttft, 50):7.0f} / {pct(ttft, 99):6.0f}  | {pct(tpot, 50):5.1f} / {pct(tpot,
99):5.1f}")
```

输出：

```
到达率(个/s) | 输出吞吐(token/s) | TTFT 中位/P99 (ms) | TPOT 中位/P99 (ms)
        5    |        1,349    |      21 /     50  |   5.5 /   6.3
       10    |        2,639    |      23 /     65  |   6.4 /   7.3
       20    |        5,011    |      27 /     82  |   9.1 /  11.1
       30    |        6,972    |      36 /    131  |  14.6 /  17.6
       40    |        8,316    |      59 /    197  |  21.0 /  25.2
       45    |        8,570    |     108 /   1306  |  22.6 /  27.3
       50    |        8,660    |     211 /   2474  |  23.2 /  27.7
       60    |        8,780    |     906 /   4176  |  23.2 /  31.9
```

表格就是上面引用的各级结果。注意每秒 40 个和 45 个之间：吞吐只多了 3%，TTFT 的 P99 却从 197 ms 跳到了 1306 ms，这就是饱和点。

这个模拟是理想化的：它不考虑 CPU 上的调度开销、通信、采样，也没有抢占。真实服务的饱和点会更早，但曲线的形状是一样的。

容易混淆的地方

1. **吞吐的分母要说清楚**。“token/s”可能是输出 token、输入 token 或两者之和；可能是整台机器的，也可能是每张卡的。报告时写清楚。
2. **TPOT 和 ITL 的平均方式不同**。TPOT 先对每个请求求平均再看分布，ITL 把所有间隔放在一起看分布。分块 prefill 带来的偶尔一次长间隔，在 ITL 的 P99 里看得更清楚。
3. **饱和点之后的延迟没有意义**。过载时延迟会随压测时长无限增长，比较两个系统时，要比的是饱和点在哪里、饱和之前的延迟，以及 goodput。

回顾

一次可信的推理压测，给出的是一条曲线：按泊松过程逐级提高到达率，每一级测输出吞吐和 TTFT、TPOT 的中位数与 P99。吞吐先线性增长，在服务的容量处饱和；再加压，请求开始排队，TTFT 急剧上升。模拟的 Llama-3-8B + H100 在每秒 40 个请求时吞吐 8316 token/s、TTFT P99 197 ms，每秒 60 个时吞吐只多了 6%，TTFT P99 却超过 4 秒。满足 SLO 的最大吞吐叫 goodput，比较配置、引擎和硬件时应该比它。压测还要注意数据集、输出长度、前缀缓存、预热、样本量和压测机本身，并把测试条件完整地写出来。

第 10 篇到这里结束了。推理引擎调度好了每一步要算什么，可每一步本身算得快不快，取决于 GPU 上一个个具体的算子。从下一篇开始，我们深入到算子层面，第一个问题是：为什么把几个算子合成一个，就能快很多？下一讲：**算子融合**。

AI INFRA · HANDBOOK

第 十一 篇

算子与 Kernel

收录 7 讲 · 第 83–89 讲

第83讲 算子融合

36 Pre-Norm 与 RMSNorm　38 SwiGLU 与门控 FFN　68 Prefill 与 Decode

第 38 讲的 SwiGLU，用 PyTorch 写出来只有一行：`F.silu(g) * u`。在 GPU 上执行时，这一行会变成好几个独立的小任务：先算 sigmoid，再乘 g 得到 SiLU，再乘 u。每个小任务都要把输入从显存读进来，算完再把结果写回显存，下一个小任务又把它读回来。真正的计算只是每个元素几次乘法，大部分时间花在了来回搬数据上。

算子融合（kernel fusion）把这几个小任务合成一个：数据只读进来一次，在芯片内部算完所有步骤，只把最终结果写回去。这一讲从 GPU 的存储结构讲起，算一算融合省了多少，再看推理引擎里有哪些常见的融合。从这一讲开始的第 11 篇，我们把目光从“每一步算什么”移到“每一个运算在 GPU 上怎么算”。

数据在哪里，搬一次多慢

GPU 上的数据可以放在几个地方，越靠近计算单元，越快、越小（第 61 讲会细讲）：

- **显存**（HBM）：H100 有 80 GB，带宽 3.35 TB/s。模型权重、KV Cache、每一层的输入输出都放在这里。
- **片上存储**：每个计算单元（SM，H100 有 132 个）里有几百 KB 的共享内存（SRAM）和寄存器，加起来也就几十 MB。带宽比显存高一个数量级以上，读写几乎没有等待。

GPU 上执行的一个函数叫一个 **kernel**。PyTorch 里的每个算子（加法、乘法、sigmoid、矩阵乘法……），一般就对应一个 kernel。一个 kernel 的典型流程是：从显存读入输入 → 在片上计算 → 把输出写回显存。kernel 结束后，片上的数据就不再保留，**两个 kernel 之间交换数据，只能经过显存**。

第 68 讲算过，H100 的算力和带宽之比约是 295 FLOPs/字节：每从显存读 1 个字节，要做 295 次运算，计算单元才不会闲着。而 SwiGLU 里的这些逐元素运算，每个元素（2 个字节）只做几次运算，算术强度不到 1。它们完全卡在带宽上：**耗时 ≈ 读写的字节数 ÷ 带宽**，算得再快也没用。

所以，要让这类运算变快，唯一的办法是**少读写几遍显存**。

数一数读写了几遍

以 Llama-3-8B 的一层为例，prefill 4096 个 token。SwiGLU 的两个输入 g、u，形状都是 [4096, 14336]，BF16 每个元素 2 字节，一个张量就是 117 MB。

不融合时，3 个 kernel：

1. sigmoid：读 g，写 t。
2. 乘法：读 g、t，写 s（s 就是 SiLU(g)）。
3. 乘法：读 s、u，写 y。

一共读 5 遍、写 3 遍，整个张量读写了 8 遍，约 940 MB，按 3.35 TB/s 算，约 **280 μs**。

融合成一个 kernel：每个线程读入自己负责的那几个 g 和 u，在寄存器里算出 sigmoid、SiLU、乘积，直接写出 y。t 和 s 从来没有出现在显存里。读 2 遍、写 1 遍，一共 3 遍，约 **105 μs**。

同一个运算，结果完全一样，时间是原来的八分之三。

再看每个子层前面都有的**残差相加 + RMSNorm**（第 36 讲），张量是 [4096, 4096]，34 MB。逐个算子执行时：相加（读 x、h，写 r）、平方（读 r，写 sq）、求平均（读 sq）、加 eps 和开方（两个很小的 kernel）、除法（读 r，写 n）、乘权重（读 n，写 y），一共 7 个 kernel，整个张量读写 10 遍，约 **100 μs**。融合后一个 kernel 一行一行地处理：读 x、h，算出 r，在片上求平方和、归一化、乘权重，写出 r（残差流后面还要用）和 y，4 遍，约 **40 μs**。

下面的演示列出了每个 kernel 读什么、写什么。蓝色是读，绿色是写；切换融合，看中间结果是怎么消失的。

融合前后：显存读写了几遍

⇆ 对照

SwiGLU 激活　残差相加 + RMSNorm

不融合（逐个算子）　融合成一个 kernel

token 数 T = **4096**

kernel 1 **sigmoid**

读 g　t = 1 / (1 + e^(−g))　写 t

kernel 2 **mul**

读 g　读 t　s = g · t（SiLU）　写 s

kernel 3 **mul**

读 s　读 u　y = s · u　写 y

每个张量 4096 × 14336 × 2 字节 = 117.4 MB。3 个 kernel，整张量读写 8 遍 = 940 MB

耗时 ≈ 940 MB ÷ 3.35 TB/s ≈ 280 μs（融合后读写 3 遍，≈ 105 μs）

这些运算每个元素只做几次加减乘除，算得再快也要等数据：时间几乎全花在显存读写上。融合省掉的，是中间结果写回去、再读回来的那几遍。

图 83-1　融合前后的显存读写。不融合时，每个中间结果都要写回显存、再被下一个 kernel 读回来；融合后只读最初的输入、只写最终的输出。

要点

逐元素运算和归一化这类运算，每个元素只做几次计算，时间几乎全花在读写显存上。两个 kernel 之间只能通过显存交换数据，所以把它们融合成一个 kernel，让中间结果留在片上，就能省掉写回再读回的那几遍，按比例省时间。

放到一层里看

一层 Transformer 里有 1 次 SwiGLU 激活和 2 次残差相加 + RMSNorm。不融合时这几段一共约 481 μs，融合后约 185 μs。同一层的矩阵乘法（约 2.2 亿个参数，计算受限）大约要 1812 μs。

也就是说，不融合时，这些“不起眼”的逐元素运算要占到矩阵乘法时间的四分之一以上；融合后只占一成左右。对一个 32 层的模型，prefill 一次能省下约 9 ms。

decode 时情况又不一样：一步只有几十个 token，张量很小（32 个 token 的 SwiGLU 张量不到 1 MB），读写一遍只要零点几微秒。这时真正的开销是 **kernel 的数量**：每启动一个 kernel，CPU 和 GPU 都有几微秒的固定开销，一层十几个小 kernel，32 层就是好几百个。融合减少了 kernel 数，在 decode 时一样有用；剩下的启动开销，第 89 讲用 CUDA Graph 来解决。

推理里常见的融合

逐元素运算链。激活函数和门控（vLLM 里的 `silu_and_mul`）、残差相加 + 归一化（`fused_add_rms_norm`）、RoPE 旋转（第 37 讲）等，都有专门的融合 kernel。

把逐元素运算塞进矩阵乘法的“尾巴”。矩阵乘法算完一块输出时，结果还在片上，可以顺手加上偏置、做激活、加残差、做量化，再写回显存。这叫**尾部融合**（epilogue fusion），省掉了把矩阵乘法的结果写出去再读回来的那一遍。

把几个矩阵乘法合成一个。Q、K、V 三个投影的输入都是同一个 x，可以把三个权重矩阵拼成一个大矩阵，做一次矩阵乘法（vLLM 里的 `qkv_proj`）；SwiGLU 的 gate 和 up 两个投影也一样（`gate_up_proj`）。x 只需要读一次，kernel 也少了，矩阵也更大，GPU 更容易跑满。

注意力整个融合。注意力要算 QK^T、缩放、掩码、softmax、乘 V。如果一步一个 kernel，中间那个 $s \times s$ 的打分矩阵要在显存里写写读读好几遍，序列一长，它比 Q、K、V 本身大得多。把整个注意力融合成一个 kernel，就是下一讲的 FlashAttention。

谁来做融合

手写 kernel。推理引擎里最重要的融合，通常是用 CUDA 或 Triton（第 86、88 讲）手写的，针对具体的形状和硬件调到最优。

编译器自动融合。PyTorch 的 `torch.compile` 会分析计算图，把相邻的逐元素运算、归约自动融合，生成 Triton kernel。vLLM 也用它来融合模型里零散的运算。TensorRT 等推理编译器做的也是类似的事。

融合也有边界。不是什么都能融合：如果后一个运算需要前一个运算在**所有**位置上的结果（比如对整个张量求全局最大值），而这些位置分散在不同的计算单元上，就必须等前一个 kernel 全部结束；片上的寄存器和共享内存有限，融合得太多也放不下；矩阵乘法本身是计算受限的，融合省的是它前后的读写，而不是它自己的时间。

融合到哪里为止

拿一段更长的算子链来算：Qwen3-8B 的 FFN 中间宽度 12288，一次处理 1024 个 token。G = X·W_g、U = X·W_u 都是 [1024, 12288]，BF16 各 24 MiB。接着算 Z = SiLU(G) ⊙ U，再把 Z 按固定的 scale 量化成每个元素 1 字节，交给下一个算子。

- **三步分开**：SiLU 读 G、写 T（48 MiB）；乘法读 T、U，写 Z（72 MiB）；量化读 Z、写 12 MiB 的结果（36 MiB）。共 **156 MiB**，3 个 kernel。
- **融合前两步**：省掉 T 的写出和读回，共 **108 MiB**。
- **三步全融合**：读 G、U，直接写出 1 字节的结果，共 **60 MiB**，1 个 kernel。

每少保存一个 24 MiB 的中间张量，就少一次写出、一次读入，共 48 MiB。只融合 SiLU 和乘法时，读写从 120 MiB 降到 72 MiB，按带宽估算快 1.67 倍；在 RTX PRO 6000 上实测，从约 14.2 μs 降到 7.7 μs，快约 1.8 倍。

融合也有边界。逐元素运算每个输出只依赖同一位置的输入，读进来就能算完，所以可以一直接下去。遇到需要整行或整个张量的运算（比如 softmax 的分母、整行共用的量化 scale、下一层的矩阵乘），就不能简单地接上：要么像第 84 讲那样找到能逐块计算的算法，要么在这里断开。前后两步的数据布局不一致时，也要多一次重排，或者让前一步直接按后一步需要的布局写出。

读写量和峰值显存是两回事。设每步先分配输出、完成后释放用完的输入。三步分开时，SiLU 那一步 G、U、T 同时在显存里，峰值 72 MiB；三步全融合时，G、U 和最终输出同时在，峰值 60 MiB。读写少了 96 MiB，峰值只少了 12 MiB，因为同一块显存可以反复读写。

不等整个 kernel：按块接力

融合要求两步能写进同一个 kernel。还有一种情况：两个 kernel 有依赖，后一个只需要前一个的一块结果，却要等前一个**全部**完成才能开始。

persistent kernel 的做法是：只启动一个常驻 GPU 的 kernel，它从队列里领任务，用完成标志确认依赖的数据块是否已经算好。以 Qwen3-8B 的 up 投影后接 SiLU 为例：512 个 token，每 64 个一块，共 8 块。在 RTX PRO 6000 上，每块投影约 12.8 μs（计算受限），每块激活约

1.76 μs（带宽受限），主机每提交一个 kernel 要 5 μs。

- **先全部投影，再全部激活**：2 次提交 + 8 × (12.8 + 1.76) ≈ **126 μs**。
- **按块接力**：每个任务多 0.52 μs 去领任务（一次原子加）和发完成标志，但只提交一次，每块投影算完立刻激活。投影首尾相接，最后一块激活在投影全部结束后完成，约 **114 μs**，快 1.1 倍。

省下的 12 μs 来自少一次提交和七块激活被藏到投影后面，任务调度又加回了约 4.7 μs。两段越接近等长，按块接力省得越多；一段远长于另一段时，基本只省一次提交。研究系统 Mirage Persistent Kernel（MPK）就是用这种方式把整个模型的一步放进一个 kernel。

decode 时这个问题更突出。batch 为 1 时，每个矩阵-向量乘都铺满了所有 SM，下一个乘法要等所有 SM 的结果，每个 kernel 边界都是一次全 SM 的同步。在 Qwen3-8B 的 4096 × 4096 投影上实测：64 个乘法串成依赖链，每个边界多出 1.2–2.3 μs，主要花在等最慢的 SM 上（显存对各 SM 的服务并不均匀）。改成各 SM 从计数器动态领下一批行、快的多做慢的少做，每个边界降到 0.74–1.06 μs，接近全 SM 同步本身约 1 μs 的代价。这部分是依赖本身的代价，换提交方式也省不掉。

整个请求能快多少

一个 kernel 快了 4 倍，用户等的时间会少多少？设请求由串行的阶段组成，热点占原时间的比例是 f，热点加速 s 倍，其他阶段不变：

$$\text{整体加速} = 1 \div ((1 - f) + f / s)$$

这就是 Amdahl 定律。热点占 20% 时，快 2 倍只换来 1.11 倍；快 10 倍是 1.22 倍；热点耗时降到零，上限也只有 1.25 倍。

有并行分支时还要看**关键路径**。准备 10 μs 后，热点 A（60 μs）和分支 B（40 μs）同时执行，收尾 10 μs 要等两条都完成，总共 80 μs。把 A 加速 4 倍到 15 μs，总时间是 10 + max(15, 40) + 10 = **60 μs**：A 省了 45 μs，请求只省 20 μs，因为 B 接替 A 成了最长的那条路。更糟的是，如果 A 的新实现大量占用显存带宽，让同时运行的 B 慢到 90 μs，请求就变成 110 μs，比优化前还慢。

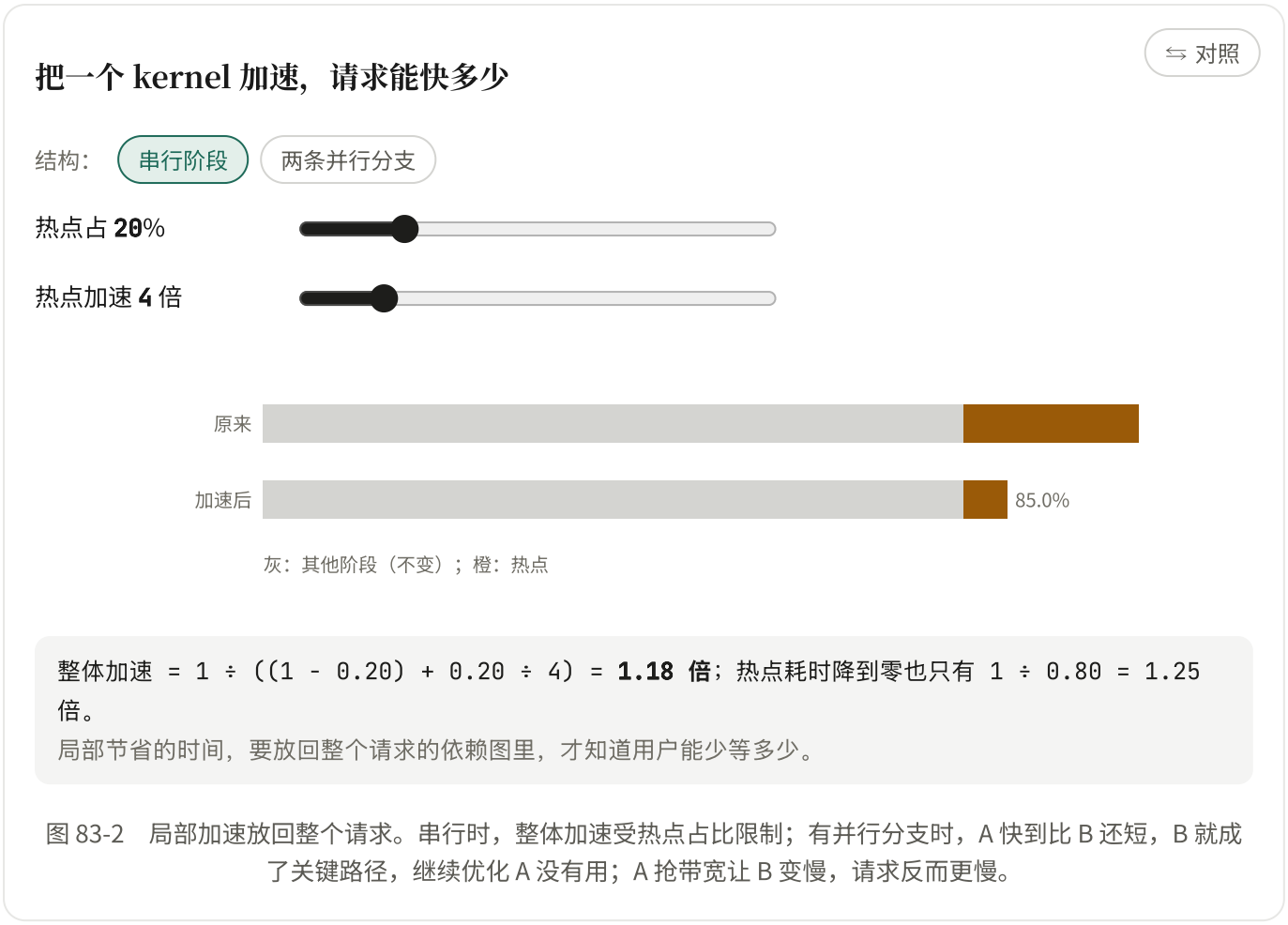


图 83-2　局部加速放回整个请求。串行时，整体加速受热点占比限制；有并行分支时，A 快到比 B 还短，B 就成了关键路径，继续优化 A 没有用；A 抢带宽让 B 变慢，请求反而更慢。

一个真实的例子：RTX PRO 6000 上跑 Qwen3-8B，输入 7239 个 token、输出 32 个，并发为 1。把 36 层的 SwiGLU 激活换成另一个 kernel：decode 的 1116 次单行调用从约 2.4 ms 降到 0.92 ms，快 2.6 倍；prefill 的 36 次大调用几乎不变。全部激活从 13.6 ms 降到 12.2 ms，只占整个请求（约 817 ms）的一小部分。客户端交错测了 11 对，配对差值的中位数约 1.3 ms，和预期的 1.4 ms 吻合，约占请求的 0.16%。kernel 快了 2.6 倍，请求只快了千分之二。分析一个优化值不值得做，先统计各种形状的调用次数，再算每次省多少，最后沿依赖关系看能不能缩短整个请求。

用代码复现

下面的代码用 NumPy 写出两段运算的“逐个算子”版本和“融合”版本，先确认结果一致；再按 Llama-3-8B 的形状数出读写的遍数，估算 H100 上的耗时。

```python
import numpy as np

# 两段逐元素运算，按 PyTorch 逐个算子执行（每个算子是一个 kernel：从显存读输入、把结果写回显存）
def swiglu_eager(g, u):
    t = 1 / (1 + np.exp(-g))      # kernel 1：读 g，写 t
    s = g * t                     # kernel 2：读 g、t，写 s          （SiLU = g · sigmoid(g)）
    return s * u                  # kernel 3：读 s、u，写 y
def swiglu_fused(g, u):           # 融合成 1 个 kernel：读 g、u，在片上算完，只写 y
    return g / (1 + np.exp(-g)) * u

def add_rmsnorm_eager(x, h, w, eps=1e-5):
    r = x + h                                       # 读 x、h，写 r（残差流，后面还要用）
    sq = r * r                                      # 读 r，写 sq
    ms = sq.mean(-1, keepdims=True)                 # 读 sq，写每个 token 一个数（很小，忽略）；加 eps、开方又是两个小 kernel
    n = r / np.sqrt(ms + eps)                       # 读 r，写 n
    return r, n * w                                 # 读 n，写 y（w 只有 d 个数，忽略）
def add_rmsnorm_fused(x, h, w, eps=1e-5):           # 融合：读 x、h，写 r 和 y
    r = x + h
    return r, r / np.sqrt((r * r).mean(-1, keepdims=True) + eps) * w

rng = np.random.default_rng(0)
g, u = rng.standard_normal((2, 8, 16))
x, h = rng.standard_normal((2, 8, 4)); w = rng.standard_normal(4)
print("结果一样：", np.allclose(swiglu_eager(g, u), swiglu_fused(g, u)),
      all(np.allclose(a, b) for a, b in zip(add_rmsnorm_eager(x, h, w), add_rmsnorm_fused(x, h, w))))

# Llama-3-8B 的一层，prefill 4096 个 token（BF16），H100 带宽 3.35 TB/s：这些运算都卡在带宽上，耗时 ≈ 读写字节 ÷ 带宽
T, d, dff, BW = 4096, 4096, 14336, 3.35e12
# （名字，张量大小，不融合：读写整张量的次数，kernel 数，融合：读写次数）
ops = [("SwiGLU 激活", T * dff * 2, 5 + 3, 3, 3), ("残差相加 + RMSNorm", T * d * 2, 6 + 4, 7, 4)]
tot = [0, 0]
for name, size, n_e, k_e, n_f in ops:
    te, tf = n_e * size / BW * 1e6, n_f * size / BW * 1e6
    tot[0] += te * (2 if "RMS" in name else 1); tot[1] += tf * (2 if "RMS" in name else 1)
    print(f"{name}：张量 {size / 1e6:.0f} MB；不融合 {k_e} 个 kernel、读写 {n_e} 遍 ≈ {te:.0f} μs；融合后 1 个 kernel、读写 {n_f} 遍 ≈ {tf:.0f} μs")
mm = 2 * T * 7.0e9 / 32 / 989e12 * 1e6        # 这一层的矩阵乘法（每层约 2.2 亿参数），卡在算力上
```

```
print(f"一层里（1 次 SwiGLU + 2 次 RMSNorm）：不融合 {tot[0]:.0f} μs，融合 {tot[1]:.0f} μs；同一层的矩阵乘法约 {mm:.0f} μs")
```

输出：

```
结果一样： True True
SwiGLU 激活：张量 117 MB；不融合 3 个 kernel、读写 8 遍 ≈ 280 μs；融合后 1 个 kernel、读写 3 遍 ≈ 105 μs
残差相加 + RMSNorm：张量 34 MB；不融合 7 个 kernel、读写 10 遍 ≈ 100 μs；融合后 1 个 kernel、读写 4 遍 ≈ 40 μs
一层里（1 次 SwiGLU + 2 次 RMSNorm）：不融合 481 μs，融合 185 μs；同一层的矩阵乘法约 1812 μs
```

第一行确认融合不改变结果。第二、三行是两段运算融合前后的读写遍数和耗时：SwiGLU 从 8 遍、280 μs 降到 3 遍、105 μs；残差相加 + RMSNorm 从 10 遍、100 μs 降到 4 遍、40 μs。最后一行把它们放进一层：481 μs 对 185 μs，矩阵乘法约 1812 μs。

注意，NumPy 在 CPU 上运行，这里的“融合”版本只是写成了一个表达式，NumPy 实际上仍会产生中间数组。真正的融合要在 GPU 上写一个 kernel，第 88 讲会用 Triton 写一个。

第二段代码复现这一讲后半部分的数字：三种融合边界的读写量，按块接力的 persistent kernel，Amdahl 定律与关键路径，以及 Qwen3-8B 请求的例子。

```
# 1. 融合的边界：Z = SiLU(G) ⊙ U，再量化成 1 字节。G、U、Z 都是 [1024, 12288]，BF16 各 24 MiB
t = 24                                                    # MiB
plans = {"三步分开": [(t, t), (2 * t, t), (t, t / 2)],     # 每步（读，写）：SiLU、乘法、量化
         "融合前两步": [(2 * t, t), (t, t / 2)],
         "三步全融合": [(2 * t, t / 2)]}
for k, steps in plans.items():
    print(f"{k}：{len(steps)} 个 kernel，读写 {sum(r + w for r, w in steps):.0f} MiB")
sep, fused = (t + t) + (2 * t + t), 2 * t + t                # SiLU 和乘法分开 / 融合
print(f"只融合 SiLU 和乘法：{sep} → {fused} MiB，按带宽估算快 {sep/fused:.2f} 倍（实测 14.2 → 7.7 μs，快 {14.2/7.7:.2f} 倍）")

# 2. persistent kernel：up 投影后接 SiLU，512 个 token 分 8 块，每块投影 12.8 μs、激活 1.76 μs
proj = 2 * 64 * 4096 * 12288 / 503.8e12 * 1e6
act = 64 * 12288 * 4 / 1792e9 * 1e6
launch, task = 5, 0.15 + 0.37                             # 主机每次提交 5 μs；领任务 + 发完成标志
coarse = 2 * launch + 8 * (proj + act)
fine = launch + 8 * (proj + task) + (act + task)
print(f"每块投影 {proj:.1f} μs、激活 {act:.2f} μs；先全投影再全激活 {coarse:.0f} μs；按块流水 {fine:.0f} μs（快 {coarse/fine:.2f} 倍）")

# 3. 整个请求能快多少：热点占比 f、热点加速 s（Amdahl）；两条并行分支看关键路径
amdahl = lambda f, s: 1 / ((1 - f) + f / s)
print(f"热点占 20%：快 2 倍 → {amdahl(0.2, 2):.2f}，快 10 倍 → {amdahl(0.2, 10):.2f}，上限 {1/0.8:.2f}")
path = lambda a, b: 10 + max(a, b) + 10
print(f"分支 A 60 μs、B 40 μs：{path(60, 40)} μs；A 快 4 倍：{path(15, 40)} μs；A 抢带宽让 B 变成 90 μs：{path(15, 90)} μs")

# 4. Qwen3-8B 一次请求：SwiGLU 激活从 13.6 ms 降到 12.2 ms，请求约 817 ms
print(f"激活占请求 {13.6/851:.1%}，省 {13.6-12.2:.1f} ms，约占请求的 {1.4/817:.2%}")
```

输出：

```
三步分开：3 个 kernel，读写 156 MiB
融合前两步：2 个 kernel，读写 108 MiB
三步全融合：1 个 kernel，读写 60 MiB
只融合 SiLU 和乘法：120 → 72 MiB，按带宽估算快 1.67 倍（实测 14.2 → 7.7 µs，快 1.84 倍）
每块投影 12.8 µs、激活 1.76 µs；先全投影再全激活 126 µs；按块流水 114 µs（快 1.11 倍）
热点占 20%：快 2 倍 → 1.11，快 10 倍 → 1.22，上限 1.25
分支 A 60 µs、B 40 µs：80 µs；A 快 4 倍：60 µs；A 抢带宽让 B 变成 90 µs：110 µs
激活占请求 1.6%，省 1.4 ms，约占请求的 0.17%
```

对照正文：156、108、60 MiB；120 → 72 MiB 估算快 1.67 倍、实测 1.84 倍；126 µs 与 114 µs；1.11、1.22、上限 1.25；80、60、110 µs；激活约占请求的 1.6%，省下的 1.4 ms 约占 0.17%。

容易混淆的地方

1. **融合不减少计算量，减少的是数据搬运。**融合前后做的乘法、加法一样多，省的是中间结果在显存里的往返。

2. **只对带宽受限的运算有大用。**逐元素运算、归一化、softmax 这些算术强度低的运算，融合效果显著；矩阵乘法这种计算受限的运算，自身的时间不会因为融合而缩短。

3. **“算子”和“kernel”不完全是一回事。**算子是框架里的一个运算（比如 `torch.add`），kernel 是 GPU 上实际执行的函数。一般一个算子对应一个 kernel，但有的算子会启动好几个 kernel，融合后一个 kernel 也可以完成好几个算子的工作。

4. **读写量少了，峰值显存不一定少多少。**三步全融合省了 96 MiB 的读写，峰值只少了 12 MiB。

5. **kernel 的加速比不等于请求的加速比。**热点占比决定上限（Amdahl 定律），并行分支还会让关键路径换人。

回顾

GPU 的显存大而慢，片上存储小而快；一个 kernel 从显存读输入、在片上算、把结果写回显存，两个 kernel 之间只能经过显存交换数据。逐元素运算和归一化每个元素只做几次运算，耗时几乎等于读写的字节数除以带宽。算子融合把几个 kernel 合成一个，中间结果留在片上：Llama-3-8B 一层、4096 个 token 时，SwiGLU 激活从读写 8 遍（约 280 µs）降到 3 遍（约

105 μs），残差相加 + RMSNorm 从 10 遍降到 4 遍，一层的这几段从 481 μs 降到 185 μs。推理里常见的融合有逐元素运算链、矩阵乘法的尾部融合、合并 QKV 和 gate/up 投影，以及整个注意力的融合。

融合的边界在需要整行或整个张量的运算处；读写量和峰值显存要分开算。两个有依赖的 kernel 还可以按块接力（persistent kernel），本例从 126 μs 降到 114 μs，两段越等长省得越多。最后要把局部的节省放回整个请求：热点只占 20% 时，整体最多快 1.25 倍；Qwen3-8B 的例子里 SwiGLU 快了 2.6 倍，请求只快了约 0.16%。

注意力也是一串运算：打分、softmax、加权求和。可它的中间结果是一个 $s \times s$ 的矩阵，而且 softmax 要先知道整行的最大值和总和，才能算出每个位置的权重，似乎没法一块一块地算。怎么才能把它融合起来？下一讲：**FlashAttention**。

第84讲 FlashAttention

26 自注意力　27 因果掩码　83 算子融合

第 26 讲的注意力，写成公式只有一行：$O = \mathrm{softmax}(QK^\top / \sqrt{d})\,V$。照着公式一步步算，要先得到打分矩阵 $S = QK^\top / \sqrt{d}$，它的形状是 $s \times s$：序列有多少个 token，它就有多少行、多少列。序列长 8192 时，一个头的 S 就有 6700 多万个数，BF16 下 128 MiB；32 个头就是 4 GiB，这还只是一层、一个请求。

更要命的是读写。上一讲说过，两个 kernel 之间只能经过显存交换数据。缩放、加掩码、softmax、乘 V，每一步都要把这个 $s \times s$ 的大矩阵从显存读进来、再写回去。注意力的乘加次数并不少，可时间主要花在了搬运这个中间矩阵上。

2022 年的 **FlashAttention** 把整个注意力融合成一个 kernel，S 从头到尾不写进显存。难点在于 softmax：要算出每个位置的权重，得先知道整行的最大值和总和，看上去必须先把整行打分都算出来。这一讲讲它是怎么绕过这个难点的。

标准实现在搬什么

把一个头的注意力按 PyTorch 的写法逐个算子执行：

1. $S = Q\,K^\top$：读 Q、K，**写 S**（$s \times s$）。
2. $S = S / \sqrt{d}$：**读 S，写 S**。
3. 加因果掩码（第 27 讲）：**读 S，写 S**。
4. $P = \mathrm{softmax}(S)$：**读 S，写 P**。
5. $O = P\,V$：**读 P**、V，写 O。

$s \times s$ 的矩阵被读写了 8 遍。代码按 $s = 8192$、$d = 128$ 算了一下：标准实现一共读写约 **1082 MB**，按 3.35 TB/s 要约 **323 μs**；而这个头真正的计算量（$QK^\top$ 和 PV 两次矩阵乘法）在 H100 上只要约 **35 μs**。注意力被显存带宽卡住了，计算单元九成的时间在等数据。序列越长越糟：S 的大小和读写量都随 s^2 增长。

这正是上一讲说的融合的用武之地：如果能把这 5 步放进一个 kernel，S 只在片上出现，读写就只剩 Q、K、V、O 这几个 $s \times d$ 的矩阵。

难点：softmax 要先看完整行

融合的思路是**分块**：片上存储只有几百 KB，放不下整个 S，那就一次算 S 的一小块。比如一次取 128 个查询（Q 的一块）和 128 个键（K、V 的一块），算出 S 的一个 128×128 的块，用完就扔，再算下一块。

麻烦在 softmax。对一行打分 $x_1, \cdots, x_n$，softmax 是

$$p_i = e^{(x_i - m)} / l, \qquad m = \max(x_1, \cdots, x_n), \qquad l = \Sigma_j\, e^{(x_j - m)}$$

（减去最大值 m 是为了数值稳定，第 26 讲。）m 和 l 都要看完整行才知道。如果一块一块地读，读到第一块时，还不知道后面有没有更大的打分，也不知道总和是多少，好像没法算出权重。

在线 softmax：先算着，再修正

办法是：**先按目前看到的最大值算，等看到更大的值时，把之前的结果按比例修正**。对一个查询，一边读一边维护三个数：

- m：到目前为止见过的最大打分；
- l：到目前为止的 $\Sigma\, e^{(x_j - m)}$；
- acc：到目前为止的 $\Sigma\, e^{(x_j - m)} \cdot v_j$（未归一化的加权和）。

读进新的一块时，先算出新的最大值 m_新 = max(m, 这一块的最大值)。之前累计的 l 和 acc 都是相对旧的 m 算的，要换算成相对新的 m，只需要乘一个系数 $e^{(m - m_新)}$，因为 $e^{(x - m)} \cdot e^{(m - m_新)} = e^{(x - m_新)}$。然后再加上这一块的贡献：

$$l \leftarrow l \cdot e^{(m - m_新)} + \Sigma\, e^{(x - m_新)}, \qquad acc \leftarrow acc \cdot e^{(m - m_新)} + \Sigma\, e^{(x - m_新)} \cdot v, \qquad m \leftarrow m_新$$

所有块都读完以后，输出就是 acc / l。这个结果和先算完整行 softmax、再加权求和**完全相同**，不是近似。

用代码里的数字走一遍。一个查询、8 个键，打分是 1.2, −0.5, 2.1, 0.4, 2.2, −1.0, 0.8, 3.4，对应的 v（为了好看，只取 1 维）是 0.5, −1.0, 2.0, 1.5, −0.5, 1.0, 0.0, 3.0。每块 4 个：

- **读第 1 块**：最大值 m = 2.10，l = 1.6635，acc = 2.4030。如果到此为止，输出是 acc / l = 1.4445。
- **读第 2 块**：这一块里有 3.4，比 2.10 大。旧的 l、acc 先乘 e^(2.10 − 3.40) = 0.2725 缩小，再加上第 2 块的贡献，得到 l = 1.8411，acc = 3.5166，输出 acc / l = **1.9100**。
- **一次算完整行**：softmax 再加权，也是 **1.9100**。

下面的演示上半部分是分块遍历的顺序：外层取一块查询留在片上，内层依次读入每一块 K、V，算出 S 的一块就用掉。下半部分是在线 softmax，可以换块的大小，一块一块地看 m、l、acc 怎么变。

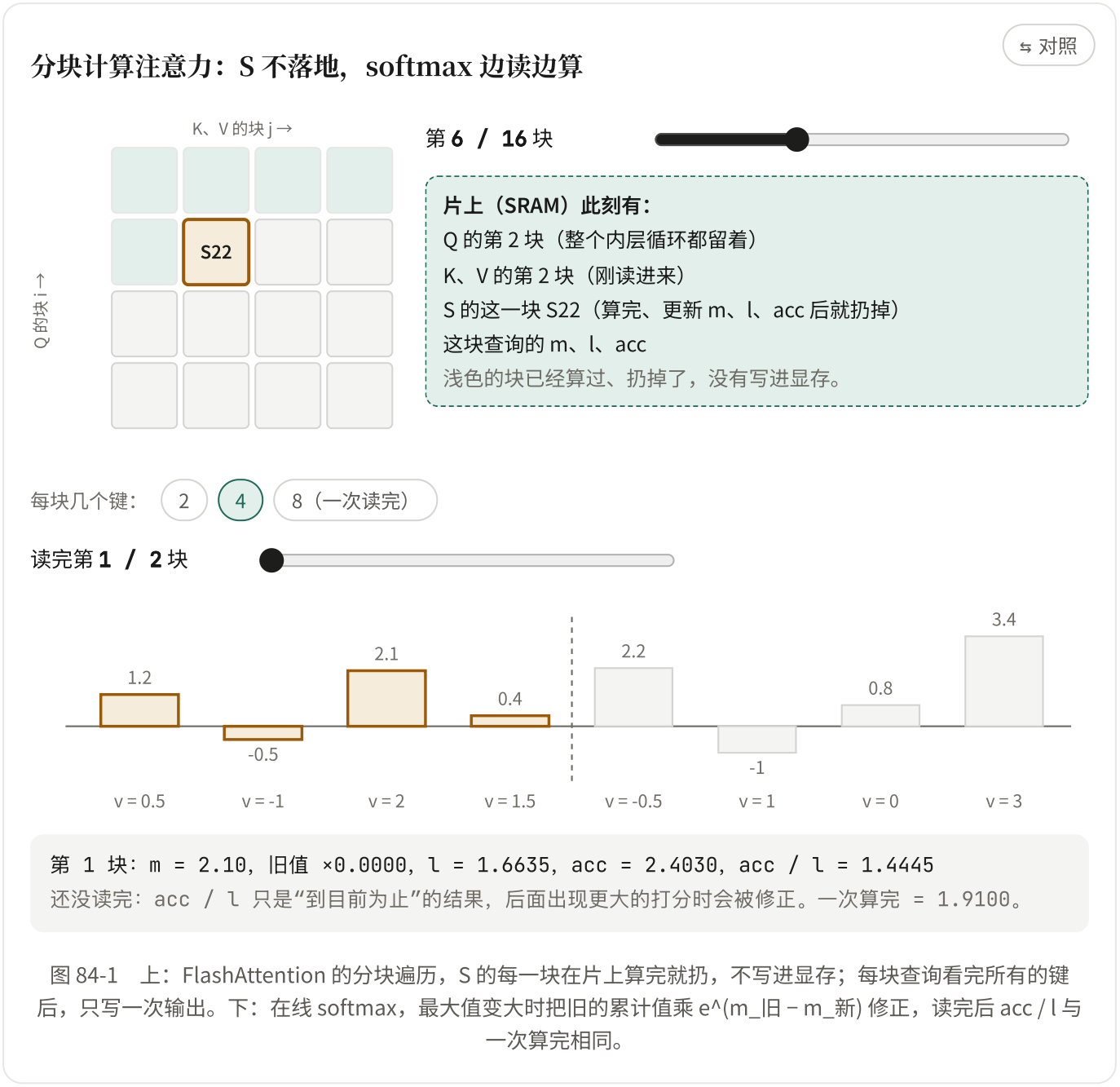


图 84-1　上：FlashAttention 的分块遍历，S 的每一块在片上算完就扔，不写进显存；每块查询看完所有的键后，只写一次输出。下：在线 softmax，最大值变大时把旧的累计值乘 e^(m_旧 − m_新) 修正，读完后 acc / l 与一次算完相同。

要点

FlashAttention 把注意力切成块，在一个 kernel 里算完：打分矩阵 S 只在片上一块一块地出现，不写进显存。softmax 用在线的方式计算，维护最大值 m、指数和 l、加权和 acc，遇到更大的打分时把旧值乘 e^(m_旧 − m_新) 修正，结果与标准实现完全相同。

省了多少

把整个流程连起来：对 Q 的每一块，把它留在片上，依次读入 K、V 的每一块，算 S 的一块、更新这块查询的 m、l、acc；看完所有的键，算 acc / l，把这块查询的输出写回显存一次。

读写量变成：Q 读一遍，O 写一遍，K、V 对每一块查询都要读一遍。代码的估算（s = 8192、d = 128，每块 128 个查询）：

- **标准实现**：约 1082 MB，约 323 μs。
- **FlashAttention**：约 277 MB，约 83 μs，是原来的四分之一。

这还是最坏的估算：同一个头的 K、V 会被好几个查询块先后读取，其中很多次能命中 GPU 的 L2 缓存（H100 有 50 MB），真正从显存读的更少。实际上，FlashAttention 在长序列上能把 GPU 的计算单元用到相当高的比例：FlashAttention-2 的论文报告，在 A100 上达到理论峰值算力的 50% 到 73%。注意力从“卡在带宽”变回了“卡在算力”。

另一个收益是**显存**。S 不再需要存下来，注意力额外占用的显存从 s^2 级别降到了 s 级别（每个查询只多存 m 和 l 两个数）。s = 32768 时，一个头的 S 就有 2 GiB，标准实现根本放不下几个请求；有了 FlashAttention，长上下文才变得可行。

块切多大：少读数据，还是少做更新

上面按“每块 128 个查询”估算。块到底切多大，要由片上能放多少东西决定。拿一个线程块的共享内存做快速缓冲（RTX PRO 6000 一个线程块最多 99 KB，即 101,376 字节）：一次留 a 行 Q 和它们的 FP32 输出累加器，K 和 V 轮流用同一个 b 行的输入槽，分数和概率共用一个 a × b 的槽，再给每行的 m、l 和临时值留三个 FP32：

$$2ad\text{（Q）} + 2bd\text{（K/V 槽）} + 4ab\text{（分数/概率）} + 4ad\text{（输出累加器）} + 12a\text{（逐行状态）} \leqslant 101{,}376 \text{ 字节}$$

Q 分成 ⌈8192 ÷ a⌉ 块，每一块都要把整份 K、V（4 MiB）扫一遍。所以 a 越大，K、V 被重读的次数越少。而 b 越小，K/V 槽和分数块越小，能放的 a 就越大：

K/V 块 b 能放的 Q 行 a	K/V 扫描次数	读写量	在线更新次数
1	128	64	260 MiB 524,288
64	82	100	404 MiB 12,800
128	53	155	624 MiB 9920

从 b = 64 缩到 b = 1，少读 144 MiB，可在线更新从 12,800 次涨到 524,288 次，约 41 倍。每次更新都要比较最大值、算指数、缩放累加器；每次只处理 1 行 K/V，矩阵乘的一个维度只有 1，矩阵单元根本用不上。反过来，b 太大，能放的 Q 行少，K/V 要多读几遍。所以实际的 kernel 一般取 64 或 128，在这两种开销之间折中。不管哪一种，都远小于标准实现把 S、P 写出再读回的 1 GiB。

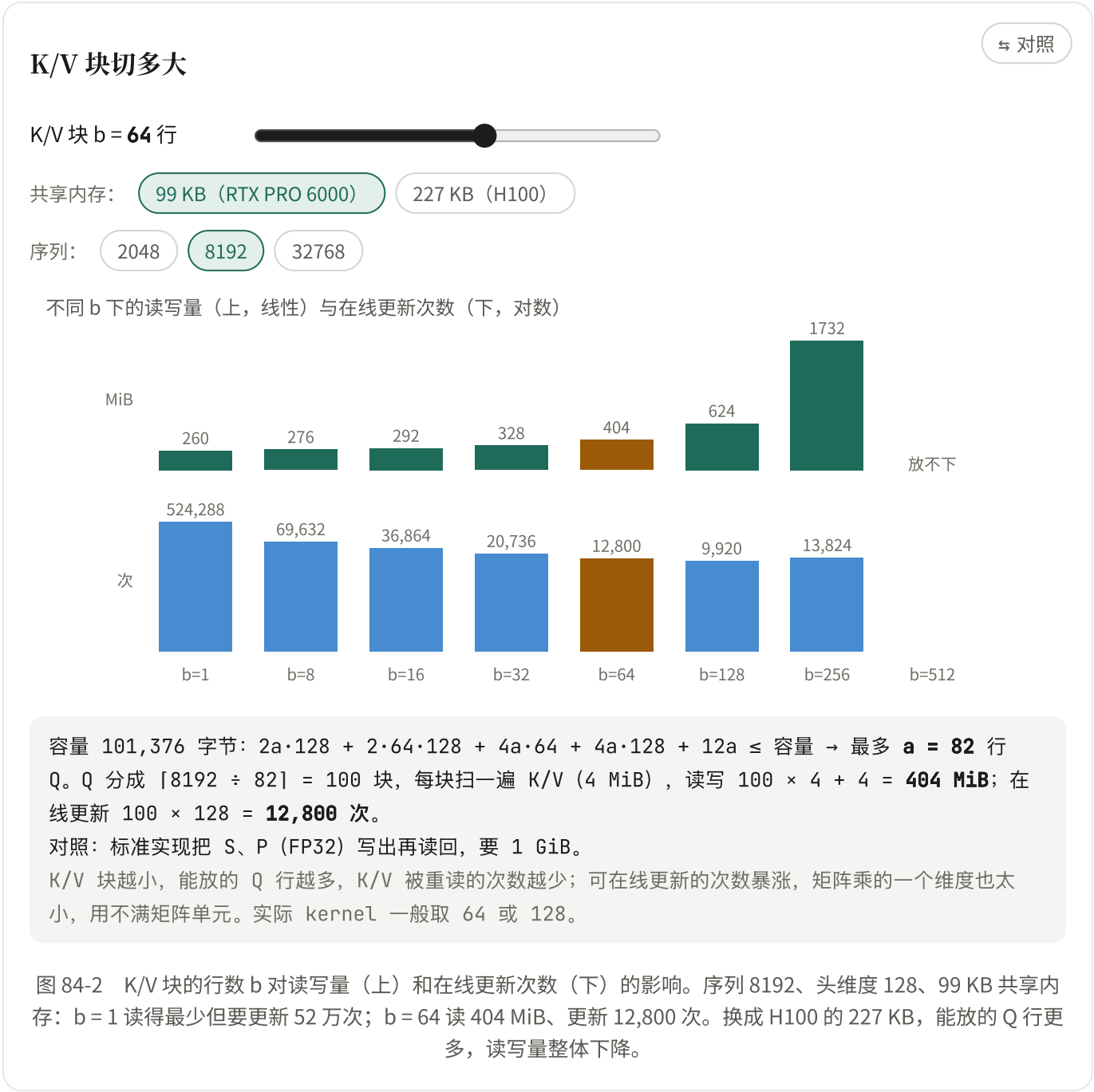


图 84-2　K/V 块的行数 b 对读写量（上）和在线更新次数（下）的影响。序列 8192、头维度 128、99 KB 共享内存：b = 1 读得最少但要更新 52 万次；b = 64 读 404 MiB、更新 12,800 次。换成 H100 的 227 KB，能放的 Q 行更多，读写量整体下降。

实测：快 33 倍，显存少 800 多 MiB

在 RTX PRO 6000 Blackwell 上，用 PyTorch 同一个注意力接口的两个后端对比：数学后端（`SDPBackend.MATH`）按公式组合矩阵乘和 softmax，保存完整的分数和概率；FlashAttention 后端（`SDPBackend.FLASH_ATTENTION`）分块、在线 softmax。BF16，一个序列、一个头、头维度 128，序列 8192，因果注意力。每次注意力计算，数学后端约 **2.6 ms**，FlashAttention 约 **80 μs**，快约 33 倍。新增的显存峰值（输出加临时工作区）从约 **848 MiB** 降到约 **22 MiB**。

这比上面只按读写量估算的加速大得多，因为数学后端除了搬中间矩阵，还要多发好几个 kernel、做一次完整的 softmax 归约，而 FlashAttention 把这些都放进了一个 kernel。三件事配合在一起：分块让当前数据放得进快速存储，在线 softmax 让归约可以逐块完成，融合省掉了块内中间结果的写回。

训练时还多一件事：重算

训练要做反向传播（第 18 讲），反向时要用到前向的 softmax 结果 P。标准实现把 P 存下来，又是 s^2 的显存。FlashAttention 不存 P，只存每个查询的 m 和 l；反向时，按块重新算一遍 S 和 P。多做了一些计算，却省下了大量的读写和显存，整体反而更快。这是“用计算换访存”的又一个例子。

推理只有前向，没有这个问题。

后来的版本

FlashAttention-2（2023）调整了循环的顺序和并行方式：外层按查询块分给不同的计算单元并行，减少了非矩阵乘法的运算（比如把除以 l 推迟到最后），进一步提高了算力利用率。

FlashAttention-3（2024）针对 H100 的新硬件特性：用专门的数据搬运单元（TMA）异步地把下一块 K、V 搬进片上，同时计算当前块；让矩阵乘法和 softmax 交替进行，互相掩盖；支持 FP8。

推理里的变体。prefill 时查询很多，正好是 FlashAttention 擅长的形状。decode 时每个请求只有 1 个查询，KV Cache 又分散在各个块里（第 75 讲），需要不同的切分方式，这是下一讲的内容。今天的推理引擎通常用 FlashAttention 或 FlashInfer 等库提供的注意力 kernel，它们都基于同样的分块和在线 softmax。

用代码复现

下面的代码先用在线 softmax 一块一块地处理上面那 8 个打分，和一次算完的结果对比；再用 NumPy 写一个分块的注意力，和标准实现对比；最后按 d = 128 估算不同序列长度下两种实现的读写量和时间。

```python
import numpy as np

# 1. 在线 softmax：一个查询、8 个键，分成两块，每块 4 个，一块一块地算
x = np.array([1.2, -0.5, 2.1, 0.4, 2.2, -1.0, 0.8, 3.4])      # 打分 q·k / √d
v = np.array([0.5, -1.0, 2.0, 1.5, -0.5, 1.0, 0.0, 3.0])      # 每个位置的 v（取 1
维，方便看）
m, l, acc = -np.inf, 0.0, 0.0                                  # 目前的最大值、指数和、
加权和
for j in range(0, 8, 4):
    xb, vb = x[j:j + 4], v[j:j + 4]
    m_new = max(m, xb.max())
    scale = np.exp(m - m_new)                                  # 最大值变了，旧的累计值
要按比例缩小
    l = l * scale + np.exp(xb - m_new).sum()
    acc = acc * scale + (np.exp(xb - m_new) * vb).sum()
    m = m_new
    print(f"读完第 {j // 4 + 1} 块：旧值缩放 ×{scale:.4f}，m = {m:.2f}，l = {l:.4f}，
acc = {acc:.4f}，当前输出 acc / l = {acc / l:.4f}")
p = np.exp(x - x.max()) / np.exp(x - x.max()).sum()
print(f"一次算完整行 softmax 再加权：{p @ v:.4f}")

# 2. 分块的注意力（FlashAttention 的思路）和标准实现结果一致
def attention(Q, K, V):
    S = Q @ K.T / np.sqrt(Q.shape[1]); P = np.exp(S - S.max(1, keepdims=True))
    return (P / P.sum(1, keepdims=True)) @ V
def flash(Q, K, V, Br=64, Bc=64):
    s, dd = Q.shape; O = np.zeros_like(Q)
    for i in range(0, s, Br):                                  # 外层：一块查询（留在片
上）
        q = Q[i:i + Br]; m = np.full(len(q), -np.inf); l = np.zeros(len(q)); acc =
np.zeros_like(q)
        for j in range(0, s, Bc):                              # 内层：一块一块地读 K、V
            sb = q @ K[j:j + Bc].T / np.sqrt(dd)
            m_new = np.maximum(m, sb.max(1)); sc = np.exp(m - m_new); pb =
np.exp(sb - m_new[:, None])
            l = l * sc + pb.sum(1); acc = acc * sc[:, None] + pb @ V[j:j + Bc]; m =
m_new
        O[i:i + Br] = acc / l[:, None]                         # 整行看完，才除以 l，写
回一次
    return O
rng = np.random.default_rng(0)
Q, K, V = rng.standard_normal((3, 256, 64))
print("分块计算与标准实现一致：", np.allclose(attention(Q, K, V), flash(Q, K, V)))

# 3. 一个头、序列长 s、d = 128（BF16）时，读写显存的字节数（不考虑因果掩码省下的一半）
d, Br, F, BW = 128, 128, 989e12, 3.35e12
```

```
print("序列长 s | 打分矩阵 S | 读写：标准实现 / FlashAttention | 读写时间：标准 / Flash | 计算时间")
for s in [2048, 8192, 32768]:
    qkvo = 4 * s * d * 2
    std = qkvo + 8 * s * s * 2           # S 写出；缩放、掩码、softmax 各读一遍写一遍；乘 V 时读 P
    fl = qkvo + 2 * s * d * 2 * (s // Br)  # Q、O 各一遍；K、V 每块查询都要读一遍
    print(f"{s:7d} | {s * s * 2 / 2**20:5.0f} MiB | {std / 1e6:6.0f} MB / {fl / 1e6:5.0f} MB | {std / BW * 1e6:5.0f} μs / {fl / BW * 1e6:4.0f} μs | {4 * s * s * d / F * 1e6:4.0f} μs")
```

输出：

```
读完第 1 块：旧值缩放 ×0.0000，m = 2.10，l = 1.6635，acc = 2.4030，当前输出 acc / l = 1.4445
读完第 2 块：旧值缩放 ×0.2725，m = 3.40，l = 1.8411，acc = 3.5166，当前输出 acc / l = 1.9100
一次算完整行 softmax 再加权：1.9100
分块计算与标准实现一致： True
序列长 s | 打分矩阵 S | 读写：标准实现 / FlashAttention | 读写时间：标准 / Flash | 计算时间
   2048 |     8 MiB |     69 MB /    19 MB |    21 μs /    6 μs |    2 μs
   8192 |   128 MiB |   1082 MB /   277 MB |   323 μs /   83 μs |   35 μs
  32768 |  2048 MiB |  17213 MB /  4329 MB |  5138 μs / 1292 μs |  556 μs
```

前三行是在线 softmax：第 2 块出现了更大的打分 3.4，旧值乘 0.2725 修正（第 1 块之前没有旧值，所以系数是 0），最终 1.9100 和一次算完相同。第四行说明分块计算的注意力与标准实现一致。表格是一个头的读写量：s = 8192 时，S 有 128 MiB，标准实现读写 1082 MB、约 323 μs，FlashAttention 约 277 MB、83 μs，计算本身只要 35 μs。

第二段代码复现块大小的取舍：标准实现中间矩阵的读写量，以及三种 K/V 块大小下能放的 Q 行数、扫描次数、读写量和在线更新次数。

```
# 一个头，序列 8192、头维度 128：标准实现要把 FP32 的 S、P 各写一次、读一次
L, d = 8192, 128
qkvo = 4 * L * d * 2
sp = 2 * 2 * L * L * 4
print(f"Q、K、V、O 共 {qkvo/2**20:.0f} MiB；S、P 写出并读回 {sp/2**30:.0f} GiB")

# 分块：一个线程块的共享内存放 a 行 Q、一个 b 行的 K/V 槽、a×b 的分数块、FP32 输出累加器和逐行状态
CAP = 101376                                        # RTX PRO 6000 一个线程块可用 99 KB
def rows(b):                                        # 容量允许的最多 Q 行数 a
    a = 0
    while 2*(a+1)*d + 2*b*d + 4*(a+1)*b + 4*(a+1)*d + 12*(a+1) ⩽ CAP: a += 1
    return a
for b in [1, 64, 128]:
    a = rows(b)
    scans = -(-L // a)                              # Q 分成几块，每块都要把整份 K/V 扫一遍
    traffic = scans * 2 * L * d * 2 + 2 * L * d * 2 # 每次扫描读 K、V，另加读一次 Q、写一次 O
    updates = scans * (L // b)                      # 每个 Q 块和每个 K/V 块之间做一次在线更新
    print(f"K/V 块 {b:>3} 行：Q 块 {a:>3} 行，扫描 {scans:>3} 次，读写 {traffic/2**20:4.0f} MiB，在线更新 {updates:>6} 次")
```

输出：

```
Q、K、V、O 共 8 MiB；S、P 写出并读回 1 GiB
K/V 块   1 行：Q 块 128 行，扫描  64 次，读写  260 MiB，在线更新 524288 次
K/V 块  64 行：Q 块  82 行，扫描 100 次，读写  404 MiB，在线更新  12800 次
K/V 块 128 行：Q 块  53 行，扫描 155 次，读写  624 MiB，在线更新   9920 次
```

对照正文：S、P 写出再读回 1 GiB；b = 1、64、128 时分别是 260、404、624 MiB，在线更新 524,288、12,800、9920 次。

容易混淆的地方

1. **FlashAttention 不是近似。**它和标准实现在数学上完全等价，结果只有浮点舍入上的微小差别。那些为了省计算而改变注意力定义的方法（滑动窗口、线性注意力，第 42、44 讲）是另一回事。

2. **它没有减少计算量**。乘加的次数和标准实现一样（训练时还因为重算多了一些），省的是显存读写和显存占用。

3. **复杂度仍然是 s^2**。计算量仍随序列长度平方增长，FlashAttention 只是让这个 s^2 的计算跑在算力的上限附近，而不是被带宽拖住。

4. **块越小不等于越好**。K/V 块切小，能放下更多 Q 行，K/V 重读得少；但在线更新的次数暴涨，矩阵单元也用不满。块大小要在读写量和更新次数之间折中。

回顾

标准的注意力要在显存里写出 s × s 的打分矩阵，缩放、掩码、softmax、乘 V 各读写一遍，s = 8192 时一个头就要读写约 1082 MB，计算单元大部分时间在等数据，长序列还放不下。FlashAttention 把注意力切块、融合成一个 kernel：一块查询留在片上，依次读入 K、V 的块，算 S 的一块就用掉。softmax 用在线的方式计算，维护 m、l、acc，最大值变大时把旧值乘 e^(m_旧 − m_新) 修正，结果与标准实现完全相同。读写降到约四分之一，显存占用从 s^2 降到 s 级别，注意力重新变成计算受限。

块的大小由片上容量决定：99 KB 的共享内存下，K/V 块取 64 行能放 82 行 Q，读写 404 MiB、在线更新 12,800 次；取 1 行读得最少（260 MiB），更新却多 41 倍。实测在 8192 的因果注意力上，FlashAttention 比保存完整中间矩阵的实现快约 33 倍，新增显存从约 848 MiB 降到约 22 MiB。

prefill 时有很多查询，可以按查询分块并行。decode 时每个请求只有 1 个查询，KV Cache 又按块散落在显存各处，FlashAttention 的切分方式就不合适了。decode 阶段的注意力 kernel 该怎么写？下一讲：**Decode 阶段的注意力 Kernel**。

第85讲 Decode 注意力 Kernel

39 MQA 与 GQA　69 KV Cache　75 PagedAttention　84 FlashAttention

上一讲的 FlashAttention 是为“很多查询”设计的：prefill 时有几千个查询，按每 128 个一块分给不同的计算单元，每个单元把自己那块查询留在片上，再依次读入 K、V。

decode 时情况完全不同。每个请求每一步只有 **1 个**新 token，也就只有 1 个查询；它要和 KV Cache 里所有的历史 token 做注意力。而第 75 讲以后，这些 KV 已经不是连续的一大段，而是按块散落在显存的各个角落，只有块表知道它们在哪。

这一讲讲 decode 阶段的注意力 kernel 要解决的两个问题：**怎么按块表读取分散的 KV**，以及**只有一个查询时，怎么让整块 GPU 都忙起来**。

decode 的注意力在做什么

先看一个请求、一个头在 decode 一步时做的事（第 69 讲）：拿着这一步的查询 q，和上下文里每个 token 的 k 算打分，softmax，再对 v 加权求和。上下文有 n 个 token，就要把 n 个 k 和 n 个 v 全部从显存读一遍。

每读一个 k（d 个数），只和 q 做一次点积，算术强度非常低。第 39 讲的 GQA 稍微好一点：Llama-3-8B 的 32 个查询头分成 8 组，每组 4 个查询头共用一个 KV 头，读一次 k、v，可以给 4 个查询用。但算术强度也就是个位数，远低于 H100 的 295 FLOPs/字节。

所以 decode 的注意力**完全是带宽受限的**：耗时 ≈ 要读的 KV 字节数 ÷ 带宽。kernel 写得好不好，就看它能不能把显存带宽用满。

按块表读取

KV Cache 被切成了每块 B 个 token 的物理块（vLLM 默认 B = 16），一个请求的第 i 个 token 在哪里，由块表决定（第 75 讲）：

$$\text{物理块} = \text{块表}[i \div B]\text{（取整）}，\ \text{块内位置} = i \bmod B$$

所以 decode 的注意力 kernel 要多做一件事：它的输入除了 q 和整个 KV Cache 池子，还有每个请求的**块表**和**上下文长度**。kernel 按逻辑块的顺序，一块一块地查块表，算出这一块在池子里的地址，把这一块的 B 个 k、v 读进片上，算打分，用第 84 讲的在线 softmax 更新 m、l、acc。读完最后一块（最后一块可能只用了一部分位置，要按上下文长度截断），输出 acc / l。

一个块里的 B 个 token 是连续存放的，所以只要 B 不太小，每次读的都是一段连续的内存，显存带宽仍然能用得很好。这就是第 75 讲说的“查表开销”：多读一个块表、多算一次地址，代价很小。

看一个缩小的例子：每块 4 个 token，一个请求的上下文有 18 个 token，块表是 [7, 2, 12, 5, 9]。下标为 13 的 token 在逻辑块 13 ÷ 4 = 3，查块表得到物理块 5，块内第 13 mod 4 = 1 个位置。最后一个逻辑块（物理块 9）只用了 2 个位置。

只有一个查询：GPU 闲着

现在看并行。GPU 有很多个计算单元（H100 有 132 个 SM），要让它们同时干活，就得把工作切成很多份。

decode 时，最自然的切法是：每个请求的每个 KV 头，交给一个工作单元（GQA 下，共用这个 KV 头的 4 个查询头一起算）。批大小是 b，就有 b × 8 个工作单元。

批大小 64 时，有 512 个工作单元，132 个 SM 都有活干。可是批大小是 1（比如一个长文档的单独请求）时，只有 **8 个**工作单元：132 个 SM 里只有 8 个在读数据，其余的全闲着。上下文越长，这 8 个单元要读的越多，越慢。

代码粗略地估算了一下（只有 u 个单元在搬数据时，带宽按 u / 132 打折）：Llama-3-8B 一层，批大小 1、上下文 32768 个 token，要读 128 MiB 的 KV，只有 8 个单元时约 **661 μs**；如果 132 个 SM 都在读，只要约 40 μs。

把上下文切开：Flash-Decoding

解决办法是**沿着上下文再切一刀**：把一个请求的上下文切成若干段，比如每 512 个 token 一段，每一段交给一个工作单元。32768 个 token 切成 64 段，8 个 KV 头就有 512 个工作单元，所有的 SM 都忙起来了。

每一段只看到了上下文的一部分，各自算出这一段的 (m, l, acc)。最后用一个很小的归约 kernel 把各段合并，用的就是第 84 讲在线 softmax 的同一个公式：取所有段的最大值 M，每段的 l 和 acc 乘 e^(m − M) 换算到同一个基准，再分别加起来，输出 acc / l。合并的结果和不切完全相同。

这个做法叫 **Flash-Decoding**（2023 年由 FlashAttention 的作者提出），也叫 split-K（因为切的是 K 的长度方向）。vLLM 早期的 PagedAttention V2 kernel 就是按 512 个 token 切段的。切段之后，批大小 1、上下文 32K 的估算耗时从约 661 μs 降到约 **40 μs**。

切段也有代价：每一段要多写出一组 (m, l, acc)，还要多一个归约的 kernel。批已经很大时，工作单元本来就够多，切段没有好处：批大小 64、上下文 4096 时，切不切都约 321 μs。所以实际的 kernel 会根据批大小和上下文长度，自动决定要不要切、切几段。

下面的演示上半部分是按块表读取的过程，可以切成 3 段并行读；下半部分是真实规模下的估算，调批大小和上下文长度，看有几个 SM 在干活。

⇆ 对照

按块表读 KV，以及把上下文切开并行

一个工作单元从头读到尾 | 切成 3 段并行读

第 **6** / **18** 轮

块表（逻辑块 → 物理块）：0→7 1→2 2→12 3→5 4→9



这一轮读：
token 5 → 块表[5 ÷ 4 = 1] = 物理块 2，块内第 1 个
一个工作单元按顺序读完 18 个 token，要 18 轮。

真实规模：Llama-3-8B 一层，H100

批大小 = **1**

上下文 = **32K**

不切 | 每 512 个 token 切一段



要读的 KV：1 × 32768 × 4096 字节 = 128 MiB。工作单元：1 个请求 × 8 个 KV 头 × 64 段 = 512 个，132 个 SM 里有 132 个在干活。
估算耗时 ≈ 134 MB ÷ (3.35 TB/s × 132/132) ≈ 40 μs
批小、上下文长时，不切就只有寥寥几个 SM 在读数据；切段以后所有 SM 一起读。批已经很大时，切不切差别不大。

图 85-1　上：decode 注意力按块表读取散落的 KV（琥珀色是这一轮读的 token），切成 3 段时三段同时读，最后合并。下：Llama-3-8B 一层的工作单元和估算耗时，批小、上下文长时，不切段只有少数 SM 在干活。

要点

decode 的注意力只有 1 个查询，完全受带宽限制。kernel 按块表一块一块地读 KV，用在线 softmax 累加。批小、上下文长时，按“请求 × KV 头”分工的单元太少，GPU 大部分闲着；把上下文切成几段并行读（Flash-Decoding），最后按在线 softmax 的公式合并，就能用满带宽。

切开归约的代价：多读一遍，结果还会变

Flash-Decoding 的思路（并行度不够，就沿归约维切开，最后再合并）在 decode 里到处都有。注意力之外最常见的是 RMSNorm：先求一行的平方和，算出缩放因子，再缩放这一行。prefill 有 1024 行，一组算一行就有 1024 个并行任务；decode 只有一行，同样的分法只有 1 个任务，绝大部分 SM 闲着。

把每行切成 8 段，第一阶段的任务就多了 8 倍。代价有两样。

多读一遍输入。一组算一行时，输入留在片上，读一遍输入、一遍 gamma，写一遍输出，1024 行共 24 MiB，1 个 kernel。切成 8 段后，要先算局部平方和，再合并成缩放因子，最后归一化时**重新读一遍输入**，共约 32.1 MiB，kernel 也从 1 个变成 3 个。

结果会变。浮点加法不满足结合律：FP32 里算 $1 + 2^{-24} + 2^{-24}$，先加前两项还是 1，再加第三项仍是 1；先加后两项得到 2^{-23}，再加 1 才是 $1 + 2^{-23}$。代码取一行 4096 个 BF16 随机数，它们的平方在 FP32 里都是精确的，所以差别只来自加法的先后。一组从头加到尾，累加器越来越大，每次加进去的小数的末位被反复舍掉，和精确值差约 111 个 ULP（unit in the last place，同一指数下相邻两个浮点数的间距）；切成 8 段，每个累加器只长到八分之一，误差降到约 9 个 ULP。段数不同，结果就不同。

合并的次序也有影响。固定 8 个局部和，40,320 种合并次序得到 4 个不同的 FP32 结果。如果用原子加把局部和加到同一个地址，到达的先后由调度决定，同一份输入跑两次可能得到不同的结果。改成先把局部和写进缓冲，全部写完后由一个线程按固定顺序相加，每次都一样，代价是多一次写出和读回，还要等最慢的一段。

这带来一个更隐蔽的问题：kernel 常常按 batch 大小选择段数（batch 大就一组一行，batch 小就切开），于是**同一个请求，在不同的 batch 里算出的结果不一样**。要让输出只由输入决定（batch invariance），段数和合并次序就不能随 batch 变化，代价是小 batch 时并行度退回到切开之前。强化学习里，生成端和训练端加载同一份权重，同一个前缀算出的概率却可能不同，原因之一就在这里。

到了大模型里

各家的 decode 注意力 kernel。vLLM 早期自己实现了 PagedAttention 的 kernel；现在的推理引擎更多地使用 FlashAttention（它的新版本支持块表）和 FlashInfer 这类专门的注意力库。FlashInfer 针对推理做了很多优化：支持任意的块大小（甚至每块 1 个 token，SGLang 的前缀树就用这种粒度），针对不同的批大小和长度自动选择切分方式，等等。

prefill 和 decode 混在一步里。连续批处理和分块 prefill（第 74、77 讲）让一步里既有 decode 请求（1 个查询），又有 prefill 的块（几百个查询）。注意力 kernel 要能同时处理这两种形状：有的实现分别调用两种 kernel，有的用一个统一的 kernel 按每个请求的查询数调整切分方式。

MLA 不一样。DeepSeek 的 MLA（第 40 讲）在 decode 时把所有查询头吸收到一个压缩的 KV 上，等于 128 个查询头共用一份 KV，算术强度高得多，需要专门的 kernel（比如 FlashMLA）。

KV Cache 量化。既然 decode 的注意力完全受带宽限制，把 KV Cache 存成 FP8（第 12 篇），要读的字节减半，注意力也就快了将近一倍。kernel 在读进片上以后再转回高精度计算。

用代码复现

下面的代码先在一个缩小的块池上实现按块表读取的 decode 注意力，和把 KV 拼成连续一段的标准算法对比；再把上下文切成 3 段分别计算、合并，和不切对比；最后按上面的粗略模型估算 Llama-3-8B 一层的工作单元数和耗时。

```python
import numpy as np
rng = np.random.default_rng(0)
B, d = 4, 8                                          # 每块 4 个 token，每个头 8 维（缩小的例子）

# 1. KV Cache 按块散落在池子里，按块表读取（PagedAttention）
k_pool, v_pool = rng.standard_normal((2, 16, B, d))      # 池子：16 个物理块
table, ctx = [7, 2, 12, 5, 9], 18                        # 这个请求的块表，上下文 18 个 token（最后一块只用了 2 个位置）
q = rng.standard_normal(d)

def paged_attention(q, table, ctx, parts=1):
    """decode：1 个查询。把上下文切成 parts 段（可以并行），每段算出 (m, l, acc)，最后合并。"""
    pieces = []
    for seg in np.array_split(np.arange(ctx), parts):
        blk = [table[i // B] for i in seg]; off = [i % B for i in seg]   # 第 i 个 token：物理块 table[i // B]，块内位置 i % B
        x = k_pool[blk, off] @ q / np.sqrt(d)
        m = x.max(); e = np.exp(x - m)
        pieces.append((m, e.sum(), e @ v_pool[blk, off]))
    M = max(p[0] for p in pieces)                        # 合并：和在线 softmax 一样，按 e^(m - M) 换算
    l = sum(p[1] * np.exp(p[0] - M) for p in pieces)
    acc = sum(p[2] * np.exp(p[0] - M) for p in pieces)
    return acc / l

K = np.concatenate([k_pool[b] for b in table])[:ctx]     # 对照：把 KV 拼成连续的一整段，按标准公式算
V = np.concatenate([v_pool[b] for b in table])[:ctx]
x = K @ q / np.sqrt(d); p = np.exp(x - x.max()); ref = (p / p.sum()) @ V
print("按块表读取 = 连续存放：", np.allclose(paged_attention(q, table, ctx), ref))
print("切成 3 段并行再合并 = 不切：", np.allclose(paged_attention(q, table, ctx, parts=3), ref))
print("下标为 13 的 token 在：物理块", table[13 // B], "的第", 13 % B, "个位置")

# 2. Llama-3-8B 一层的 decode 注意力：要读的 KV 字节数，和能同时开工的计算单元
SM, BW, KV_LAYER = 132, 3.35e12, 8 * 128 * 2 * 2         # H100 有 132 个 SM；每层每个 token 的 KV 是 4096 字节
print("批大小 | 上下文 | 要读的 KV | 工作单元（不切 / 每 512 个 token 切一段） | 估算耗时（不切 / 切）")
for b, ctx in [(1, 32768), (1, 4096), (64, 4096)]:
    nbytes = b * ctx * KV_LAYER
    units = [b * 8, b * 8 * (ctx // 512)]                # 每个请求的每个 KV 头至少是一个工作单元
    t = [nbytes / (BW * min(1, u / SM)) * 1e6 for u in units]   # 粗略：只有 u 个单元
```

```
在搬数据时，带宽按 u / 132 打折
    print(f"{b:5d} | {ctx:6d} | {nbytes / 2**20:6.0f} MiB | {units[0]:5d} /
{units[1]:5d} | {t[0]:6.0f} µs / {t[1]:4.0f} µs")
```

输出：

```
按块表读取 = 连续存放： True
切成 3 段并行再合并 = 不切： True
下标为 13 的 token 在：物理块 5 的第 1 个位置
批大小 | 上下文 | 要读的 KV | 工作单元（不切 / 每 512 个 token 切一段） | 估算耗时（不切 /
切）
    1 |  32768 |    128 MiB |     8 /   512 |    661 µs /   40 µs
    1 |   4096 |     16 MiB |     8 /    64 |     83 µs /   10 µs
   64 |   4096 |   1024 MiB |   512 /  4096 |    321 µs /  321 µs
```

前两行说明：按块表读取和连续存放的结果相同，切成 3 段再合并和不切的结果也相同。第三行是块表的换算，和演示一致。表格是真实规模的估算：批大小 1、上下文 32768 时，不切段只有 8 个工作单元，约 661 µs，切段后 512 个单元，约 40 µs；批大小 64 时，切不切都约 321 µs。

这个模型非常粗略：实际上一个 SM 能做的不止“1/132 的带宽”，一个工作单元也可能用到多个 SM 的资源。但“工作单元太少，GPU 就闲着”这个结论是可靠的。

第二段代码复现“切开归约的代价”：切成 8 段后的读写量，不同段数下平方和与精确值的差距，以及 8 个局部和的全部合并次序能得到几个不同的结果。

```python
import numpy as np
# decode 只有一行：一组算一行时只有 1 个并行任务。RMSNorm 把一行切成几段，先求局部平方和再合并
rows, H = 1024, 4096
one = rows * H * 2 * 3                                  # 读输入、读 gamma、写输出（BF16）
split = one + rows * H * 2 + rows * 8 * 4 * 2 + rows * 4 * 2   # 多读一遍输入，局部和与缩放因子的写出读回
print(f"一组一行：{one/2**20:.0f} MiB，1 个 kernel；切 8 段：{split/2**20:.1f} MiB，3 个 kernel，任务数 {rows} → {rows*8}")

# 浮点加法不满足结合律：同一行 4096 个 BF16 的平方和，切成几段得到的结果不同
rng = np.random.default_rng(1)
x = (rng.standard_normal(H) * 0.7).astype(np.float32)
x = x.view(np.uint32) & 0xFFFF0000                      # 截成 BF16 能表示的值
x = x.view(np.float32)
sq = (x * x).astype(np.float32)                         # 平方在 FP32 里是精确的
exact = sum(float(v) for v in sq.astype(np.float64))
ulp = float(np.spacing(np.float32(exact)))
for parts in [1, 2, 4, 8, 16, 32]:
    acc = np.float32(0)
    locs = []
    for seg in np.split(sq, parts):
        a = np.float32(0)
        for v in seg: a = np.float32(a + v)             # 每段从头到尾依次累加
        locs.append(a)
    for v in locs: acc = np.float32(acc + v)
    print(f"切 {parts:>2} 段：与精确值相差 {(float(acc) - exact) / ulp:+7.1f} 个 ULP")

# 固定 8 个局部和，只改合并的次序（原子加的到达顺序由调度决定）
from itertools import permutations
locs8 = [np.float32(0)] * 8
for i, seg in enumerate(np.split(sq, 8)):
    a = np.float32(0)
    for v in seg: a = np.float32(a + v)
    locs8[i] = a
res = set()
for order in permutations(locs8):
    acc = np.float32(0)
    for v in order: acc = np.float32(acc + v)
    res.add(float(acc))
print(f"8 个局部和的 40320 种合并次序，得到 {len(res)} 个不同的 FP32 结果，最大相差 {(max(res)-min(res))/ulp:.0f} 个 ULP")
```

输出：

```
一组一行：24 MiB，1 个 kernel；切 8 段：32.1 MiB，3 个 kernel，任务数 1024 → 8192
切  1 段：与精确值相差  -110.9 个 ULP
切  2 段：与精确值相差   -42.9 个 ULP
切  4 段：与精确值相差   -32.9 个 ULP
切  8 段：与精确值相差    -8.9 个 ULP
切 16 段：与精确值相差    -5.9 个 ULP
切 32 段：与精确值相差    +0.1 个 ULP
8 个局部和的 40320 种合并次序，得到 4 个不同的 FP32 结果，最大相差 3 个 ULP
```

对照正文：24 MiB 变成 32.1 MiB、1 个 kernel 变成 3 个；一组从头加到尾差约 111 个 ULP，切 8 段约 9 个；40,320 种次序得到 4 个不同的结果。

容易混淆的地方

1. **PagedAttention 首先是一种显存管理方式**。第 75 讲的分页是“KV 怎么存”，这一讲的 kernel 是“分页存的 KV 怎么读”。两者合在一起，才是 vLLM 论文里的 PagedAttention。
2. **Flash-Decoding 不是给 prefill 用的**。prefill 有很多查询，按查询切分就足够并行了；Flash-Decoding 解决的是查询太少时的并行度问题。
3. **切段合并不是近似**。合并用的是在线 softmax 的精确公式，结果和一次算完相同。
4. **切开归约不只影响速度**。多出来的并行任务要用多读一遍输入、多几个 kernel 来换；段数和合并次序还会改变浮点结果，让同一个请求在不同 batch 里算出不同的输出。

回顾

decode 时每个请求只有 1 个查询，要读完上下文里所有的 k、v，注意力完全受带宽限制。KV Cache 按块散落在显存里，kernel 按块表一块一块地读：第 i 个 token 在物理块 块表[i ÷ B] 的第 i mod B 个位置，块内连续，带宽仍能用好。按“请求 × KV 头”分工时，批小、上下文长的情况只有寥寥几个工作单元，大部分 SM 闲着；Flash-Decoding 沿上下文切段并行，最后按在线 softmax 的公式合并，批大小 1、上下文 32K 的一层从约 661 μs 降到约 40 μs。

沿归约维切开是 decode 增加并行度的通用手段，RMSNorm 也一样：切成 8 段让任务数多 8 倍，读写从 24 MiB 增到 32.1 MiB，kernel 从 1 个变成 3 个。浮点加法不满足结合律，段数和合并次序都会改变结果，要让结果与 batch 无关，就得固定段数和次序。

这两讲反复说到“工作单元”“SM”“片上存储”。GPU 上的程序到底是怎么组织的？一个 kernel 里成千上万个线程是怎么分工的？下一讲：**CUDA 编程模型**。

第86讲 CUDA 编程模型

13 神经元与线性层　83 算子融合　84 FlashAttention

前三讲反复出现了“kernel”“工作单元”“片上存储”“SM”这些词：融合是把几个 kernel 合成一个，FlashAttention 让一块查询留在片上，Flash-Decoding 要让 132 个 SM 都有活干。这一讲把它们背后的东西讲清楚：在 GPU 上写程序，到底是怎么组织的。

NVIDIA GPU 的编程模型叫 CUDA。它的核心思想只有一句话：**写一段只处理一小份数据的代码，让成千上万个线程同时执行它，每个线程根据自己的编号决定处理哪一份**。这一讲先用最简单的向量加法看清“线程、块、网格”三层结构，再看它们怎么对应到硬件上，最后用矩阵乘法看共享内存为什么重要。

一个线程，算一个元素

先从最简单的例子开始：两个长度为 n 的向量相加，c = a + b。在 CPU 上，我们写一个循环，从第 0 个元素算到第 n − 1 个。在 GPU 上，我们只写“一个元素怎么算”：

```
__global__ void add(const float* a, const float* b, float* c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;    // 我是第几个线程?
    if (i < n) c[i] = a[i] + b[i];                    // 越界的线程什么也不做
}

// CPU 端启动：每块 256 个线程，块数向上取整
add<<<(n + 255) / 256, 256>>>(a, b, c, n);
```

`__global__` 表示这是一个 kernel：在 GPU 上执行、由 CPU 启动的函数。启动时 `<<<块数, 每块线程数>>>` 指定要开多少个线程，GPU 会让这么多个线程**同时**执行这同一段代码。每个线程的代码一模一样，区别只在于几个内置变量的值不同：

- `threadIdx.x`：我在自己的块里是第几个线程（0 到 blockDim.x − 1）；
- `blockIdx.x`：我所在的块是第几个块；
- `blockDim.x`：每个块有几个线程（这里是 256）。

于是 `i = blockIdx.x * blockDim.x + threadIdx.x` 就是这个线程在全体线程里的编号，它负责第 i 个元素。

线程、块、网格

这三层结构是 CUDA 的骨架：

- **线程**（thread）：执行 kernel 代码的最小单位，有自己的寄存器。
- **线程块**（block）：一组线程，最多 1024 个。同一个块里的线程可以通过**共享内存**交换数据，也可以互相等待（同步）。
- **网格**（grid）：一次 kernel 启动的所有块。

n = 1000、每块 256 个线程时，要 $\lceil 1000 / 256 \rceil = 4$ 个块，一共 1024 个线程。最后 24 个线程的 i 是 1000 到 1023，超出了向量的范围，`if (i < n)` 让它们什么也不做。比如块 1 的第 5 个线程负责第 $1 \times 256 + 5 = 261$ 个元素；块 3 的第 231 个线程负责第 999 个，也就是最后一个元素。

块和网格都可以是二维、三维的，处理矩阵时很方便。一个 M × N 的矩阵，每个线程负责一个元素，块取 16 × 16 个线程，网格就是 $\lceil M / 16 \rceil \times \lceil N / 16 \rceil$ 个块。线程用两个下标定位：

$$\text{row} = \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y}, \qquad \text{col} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$$

20 × 30 的矩阵、16 × 16 的块，网格是 2 × 2 个块；块 (1, 1) 里的线程 (2, 13) 负责第 18 行、第 29 列。

下面的演示把线程网格画在矩阵上。每种颜色是一个块，粗框是矩阵的范围，框外的灰色格子是越界的线程。点一个格子，看它由哪个块的哪个线程负责。

线程网格怎么盖住一个矩阵

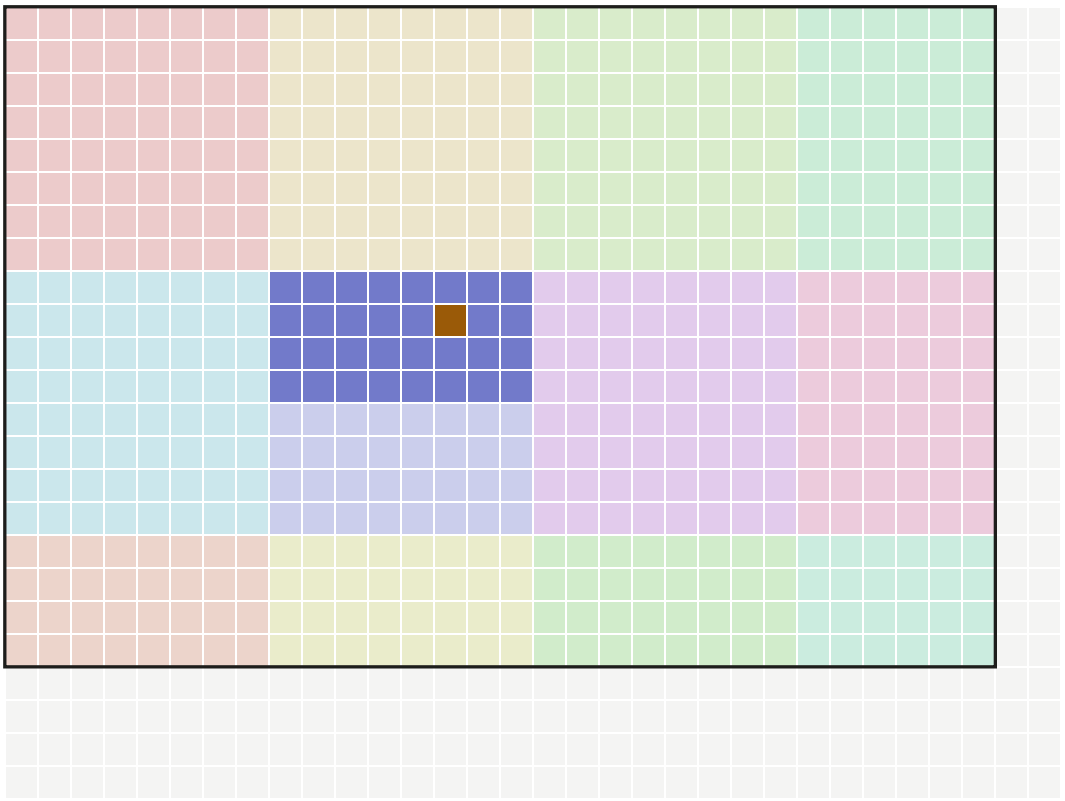
⇆ 对照

矩阵 **20** 行

矩阵 **30** 列

每块（行 × 列）：

8 × 8 | 4 × 16 | 16 × 16 | 1 × 32



网格 gridDim = (3, 4)，共 12 个块；每块 blockDim = (8, 8) = 64 个线程（2 个 warp）；一共 768 个线程，168 个越界（灰色），什么也不做。
选中的格子 (9, 13)：blockIdx = (1, 1)，threadIdx = (1, 5)
row = 1 × 8 + 1 = 9，col = 1 × 8 + 5 = 13
深色框出的是和它同一个 warp 的 32 个线程（块内编号 13，第 0 个 warp）：它们同时执行同一条指令。1 × 32 的块里，一个 warp 正好是一整行，读相邻的 32 个元素，显存访问最整齐。

图 86-1　线程网格映射到矩阵。默认 20 × 30 的矩阵、8 × 8 的块：网格 3 × 4 = 12 个块、768 个线程，其中 168 个越界。深色是和选中线程同一个 warp 的 32 个线程。

对应到硬件上

为什么要分“块”这一层？因为它对应着硬件。

GPU 由很多个 SM（流式多处理器）组成，H100 有 132 个。每个 SM 有自己的计算单元（包括做矩阵乘法的 Tensor Core）、寄存器和共享内存。kernel 启动后，GPU 把网格里的块一个个分配给 SM：**一个块整个放在一个 SM 上运行，直到结束**；一个 SM 可以同时运行好几个块，只要寄存器和共享内存够用。

这带来两个重要的性质：

1. **同一个块里的线程能合作**。它们在同一个 SM 上，可以通过这个 SM 的共享内存交换数据，可以用 `__syncthreads()` 互相等待。
2. **不同的块之间不能直接合作**。块的执行顺序没有保证，它们可能在不同的 SM 上同时运行，也可能先后运行。一个块不能等另一个块的结果。所以上一讲 Flash-Decoding 要把各段的结果先写回显存，再用**另一个 kernel** 合并。

块要足够多，所有 SM 才都有活干。上一讲批大小为 1 时只有 8 个块（工作单元），132 个 SM 里大部分闲着，就是这个原因。

再往下一层，SM 实际执行时，把一个块里的线程按编号每 **32 个**编成一组，叫一个 **warp**。同一个 warp 的 32 个线程在同一时刻执行**同一条指令**，只是作用在各自的数据上（这种方式叫 SIMT，单指令多线程）。由此有两条写 kernel 的基本规则：

- **避免分支发散**。如果同一个 warp 里一半线程走 `if`、一半走 `else`，两条路径只能先后执行，速度减半。
- **合并访存**。同一个 warp 的 32 个线程如果读的是显存里相邻的 32 个元素，硬件能把它们合成一次大的读取；如果读的地址七零八落，就要分成很多次。所以矩阵按行存放时，让相邻的线程处理同一行里相邻的列。

三种存储

线程能用的存储，和上一讲的“显存 / 片上”对应起来：

- **寄存器**：每个线程私有，最快。kernel 里的局部变量通常就放在这里。
- **共享内存**：每个块私有，在 SM 的片上，块内所有线程都能读写，几乎和寄存器一样快，但容量只有几十到两百多 KB。
- **全局内存**：就是显存（HBM），所有线程都能访问，容量大，但慢得多。kernel 的输入输出都在这里。

写高性能 kernel 的核心问题，几乎总是：**怎么让数据尽量少地从全局内存读，尽量多地在寄存器和共享内存里复用**。融合（第 83 讲）是这样，FlashAttention（第 84 讲）是这样，下面的矩阵乘法也是这样。

> **要点**
> CUDA 程序是一个 kernel 加一次启动配置：网格里有很多块，块里有很多线程，每个线程用 blockIdx、threadIdx 算出自己负责的数据。一个块整个跑在一个 SM 上，块内线程可以用共享内存合作，块与块之间不能直接合作。线程按 32 个一组（warp）执行同一条指令。

矩阵乘法：共享内存的用处

$C = A \cdot B$，A、B 都是 $K \times K$ 的方阵（第 13 讲）。最直接的写法：每个线程算 C 的一个元素，自己从显存读 A 的一整行、B 的一整列，做 K 次乘加。

问题是重复读取。A 的第 r 行，要被算 C 第 r 行的所有 K 个线程各读一遍；B 的每一列也一样。一共要从显存读 $2K^3$ 个元素。K = 4096、BF16 时，约 **274.9 GB**，按 3.35 TB/s 要约 82 ms；而计算本身只有 $2K^3 \approx 137$ GFLOP，用 Tensor Core 只要约 0.14 ms。

分块解决了这个问题。让一个线程块负责 C 的一个 $T \times T$ 小块。它沿着 K 方向一段一段地前进：每一段，块里的线程一起把 A 的一个 $T \times T$ 小块和 B 的一个 $T \times T$ 小块搬进共享内存，`__syncthreads()` 等大家都搬完，然后每个线程从共享内存里取数，累加自己那个元素的部分和；再搬下一段。

这样，每个从显存读进来的元素，在共享内存里被 T 个线程复用。从显存读的总量降到原来的 1/T：

- 块大小 1（不分块）：274.9 GB，约 82 ms；
- 块大小 16：17.2 GB，约 5.1 ms；
- 块大小 64：4.3 GB，约 1.3 ms。

块越大，复用越多，但共享内存和寄存器装得下的块有限。真实的矩阵乘法库（cuBLAS、CUTLASS）用好几层分块：线程块级的大块放在共享内存，warp 级的小块放在寄存器里交给 Tensor Core，再加上 L2 缓存的复用，最终能让矩阵乘法跑在算力的上限附近。这也是为什么第 68 讲可以把大矩阵乘法看作“计算受限”。

一个线程块在 SM 上住不住得下

上一节的矩阵乘法 kernel 能跑多快，要看 SM 里同时住着多少个线程块。线程块要在 SM 上**驻留**，必须同时分到三样东西：线程槽、寄存器、共享内存。哪一样不够，块就只能排队等前面的块结束。

拿一个实际尺寸的矩阵乘 kernel 算：每个线程块负责 128 × 128 的输出块，沿 K 每次搬 64；A 块和 W 块（BF16）各 16 KiB，FP32 累加器 64 KiB，三者都放在共享内存里，共 96 KiB。每块 256 个线程，每线程用 128 个寄存器。一个 H100 的 SM 最多 2048 个线程、65,536 个寄存器、228 KB（233,472 字节）共享内存、32 个线程块。每类资源能放几块，等于总量除以每块的需求，向下取整：

资源	SM 总量	每块需求	放得下
线程槽	2048	256	8
寄存器	65,536	256 × 128 = 32,768	2
共享内存	233,472 B	96 KiB + 1 KiB 预留	2
块数上限	32	1	32

取最小值，SM 上驻留 2 个块，16 个 warp。驻留的 warp 数占 SM 上限（64 个）的比例叫**占用率**（occupancy），这里是 25%。

改哪一项才有用，要看哪一项最紧。比如把 64 KiB 的累加器从共享内存挪进寄存器：每块的共享内存降到 32 KiB，能放 6 块了；可每个线程要多存 128 × 128 ÷ 256 = 64 个 FP32 值，寄存器从 128 个涨到 192 个，寄存器只允许 1 块，反而更少。要让更多块驻留，得同时压低寄存器和共享内存，或者把块做小。

占用率为什么重要：在途请求

为什么要多驻留几个 warp？因为读显存要等很久（这里取 600 ns），而访存请求都是驻留的 warp 发出的。一个请求在等的时候，别的请求要把接口占满，这叫**延迟隐藏**。第 61 讲的 Little 定律给出了需要多少在途字节：带宽 × 延迟。H100 是 3.35 TB/s × 600 ns ≈ 2.01 MB，平摊到 132 个 SM，每个 SM 要保持约 **15.2 KB** 在途。

驻留的 warp 能提供多少？一个 warp 的一条加载指令让 32 个线程各取 16 字节，共 512 字节；每个 warp 最多挂 4 条没完成的加载；16 个 warp 合计 **32 KiB**，是需求的 2.15 倍，够用。

但这个结论会翻转：如果延迟涨到 1000 ns，每个 warp 只能挂 2 条加载，每个 SM 就要 25.4 KB 在途，16 个 warp 只能提供 16 KiB，只有需求的 0.65 倍。这时有效带宽只有接口的六成多，接口再快也用不上。要么把块做小换更多驻留块，要么改用异步批量复制（Hopper 的 TMA），一个线程一次请求一整块，在途字节就不再取决于 warp 数。

块多大：读得少，但住得少

再看块的大小对显存读写的影响。拿 Qwen3-8B 的一个 FFN 投影：C = A·W，A 是 [1024, 4096]，W 是 [4096, 12288]，BF16。三份矩阵各读写一次只要 128 MiB。可如果每个输出块各自读自己的输入，N 方向有 N/t 个列块，每个都要把 A 读一遍；M 方向有 M/t 个行块，每个都要把 W 读一遍：

$$\text{读写量} = 2MK \cdot N/t + 2KN \cdot M/t + 2MN$$

输出块	局部存储（K 步 32）	读写量	算术强度
32 × 32	8 KiB	6168 MiB	15.9 FLOP/byte
64 × 64	24 KiB	3096 MiB	31.8 FLOP/byte
128 × 128	80 KiB	1560 MiB	63.0 FLOP/byte

块边长翻倍，输入重读减半；可局部存储涨得更快，因为累加器随 t^2 增长。三种块的算术强度都远低于 H100 的峰值算力 ÷ 带宽 ≈ 295 FLOP/byte（第 62 讲的拐点），只靠这一级分块越不过拐点，剩下的复用要靠 L2 和更大的块。

块变大还有一个代价：同时驻留的块变少，在途请求变少，实际拿到的带宽可能下降。读写量减半，时间不一定减半：如果大块只拿到小块四成的带宽，时间反而是原来的 0.504 ÷ 0.4 ≈ 1.26 倍。所以选块要先按容量排除放不下的，再按读写量缩小范围，最后实测。

bank 冲突：一列数据挤进一个 bank

共享内存不是一块整体，而是分成 32 个 **bank**，每个 bank 每个周期只能给出一个 32 位的字，字地址除以 32 的余数就是它所在的 bank。一个 warp 的 32 个线程同时读 32 个地址：落在 32 个不同的 bank，一轮就完成；落在同一个 bank，要排 32 轮。

32 × 32 的 FP32 块按行存放，读一行没问题；读一列时，相邻两行的同一列相差 32 个字，全部落在同一个 bank 里，要 **32 轮**。在每行末尾补一个字（padding），行跨度变成 33，同一列相邻行的 bank 号依次加一，32 个请求分到 32 个 bank，**1 轮**完成，只多占 3.1% 的空间。

也可以不多占空间，只改存放位置：把第 r 行第 c 列放到第 c ⊕ r 列（⊕ 是按位异或）。读一列时，32 个 lane 取到的列号互不相同，同样一轮完成。CUTLASS 把这叫 **swizzle**。TMA 这类整块搬移的硬件不能在行间插 padding，所以用 swizzle。

搬和算错开：双缓冲与 warp 分工

最后看一个 K 块从显存搬进来、再被矩阵单元用掉的过程。按 H100 一个 SM 分到的份额（显存带宽的 1/132 约 25.4 GB/s，矩阵算力的 1/132 约 7.5 TFLOP/s），一个 K 块的 A、W 共 32 KiB，搬进来约 **1.29 μs**；块乘 2,097,152 FLOPs，算完约 **0.28 μs**。搬比算慢 4.6 倍，因为这个块每搬 1 字节只做 64 次运算。

先搬后算，四块要 4 × (1.29 + 0.28) = 6.28 μs。用两个缓冲槽：算槽 A 里的块时，搬下一块进槽 B，交替使用。总时间是

$$T = t_{搬} + (n-1) \times \max(t_{搬}, t_{算}) + t_{算} = 1.29 + 3 \times 1.29 + 0.28 = 5.44\ \mu s$$

只少了 13%：搬移首尾相接，计算单元只有约两成时间在干活。这时再加第三个槽也没用（CUTLASS 把槽数叫 stage 数），只能减少每块要搬的字节，比如把块做大。

Hopper 上的 kernel 常让 warp 分工（**warp specialization**）：一个 warp 只负责发异步复制（生产者），其余 warp 只做矩阵计算（消费者）。两边靠共享内存里的**栅栏**交接：每个槽一对栅栏，“满”栅栏由复制完成触发，消费者等它；“空”栅栏由计算结束触发，下一次复制等它。FlashAttention-3 就是这样安排的：一个 warpgroup 搬 K、V，两个 warpgroup 交替做矩阵乘和 softmax。

用代码复现

GPU 代码要在 GPU 上才能跑。为了看清楚线程怎么分工，下面用 Python 模拟：先算向量加法里几个线程负责的下标，再算二维网格里一个线程负责的行列；然后用 NumPy 实现分块的矩阵乘法，统计从“显存”读了多少个元素；最后估算 K = 4096 时不同块大小的读取量。

```python
import numpy as np

# 1. 一维：n 个元素的向量加法，每个线程负责一个元素。用 Python 模拟 GPU 上的“网格 → 块 → 线程”
n, blockDim = 1000, 256
gridDim = (n + blockDim - 1) // blockDim                # 向上取整：要几个块才能盖住 n 个元素
print(f"n = {n}，每块 {blockDim} 个线程 → {gridDim} 个块，共 {gridDim * blockDim} 个线程，{gridDim * blockDim - n} 个线程越界、什么也不做")
for blockIdx, threadIdx in [(0, 0), (1, 5), (3, 231), (3, 255)]:
    i = blockIdx * blockDim + threadIdx                 # 每个线程算出自己负责的全局下标
    print(f"  blockIdx = {blockIdx}，threadIdx = {threadIdx:3d} → i = {i:4d}{' （越界，跳过）' if i ⩾ n else ''}")

# 2. 二维：矩阵按 16 × 16 的块切开，线程 (ty, tx) 负责第 row 行、第 col 列
M, N, BX = 20, 30, 16
by, bx, ty, tx = 1, 1, 2, 13
row, col = by * BX + ty, bx * BX + tx
print(f"{M}×{N} 的矩阵、16×16 的块 → 网格 {(M + 15) // 16}×{(N + 15) // 16} 个块；块 ({by}, {bx}) 里的线程 ({ty}, {tx}) 负责 ({row}, {col})")

# 3. 矩阵乘法 C = A·B（K = 4096 的方阵，BF16）：每个线程算 C 的一个元素，要从显存读多少
def matmul_tiled(A, B, T):
    """按 T×T 分块：一个块的线程先把 A、B 的一小块搬进共享内存，大家一起用，再搬下一块。返回结果和从“显存”读的元素个数。"""
    n = A.shape[0]; C = np.zeros((n, n)); loads = 0
    for i in range(0, n, T):
        for j in range(0, n, T):                        # 每个 (i, j) 是一个线程块，负责 C 的一个 T×T 小块
            for k in range(0, n, T):
                a, b = A[i:i + T, k:k + T], B[k:k + T, j:j + T]   # 搬进共享内存
                loads += a.size + b.size
                C[i:i + T, j:j + T] += a @ b
    return C, loads
A, B = np.random.default_rng(0).standard_normal((2, 8, 8))
for T in [1, 2, 4]:
    C, loads = matmul_tiled(A, B, T)
    print(f"8×8 矩阵乘法，块大小 {T}：结果正确 {np.allclose(C, A @ B)}，从显存读了 {loads} 个元素")
K = 4096
for T in [1, 16, 64]:
    reads = 2 * K**3 // T * 2                           # 每个元素被读 K/T 次；BF16 每个 2 字节
    print(f"K = {K}，块大小 {T:2d}：从显存读 {reads / 1e9:7.1f} GB，按 3.35 TB/s 需要 {reads / 3.35e12 * 1e3:7.2f} ms")
```

```
print(f"计算量 2K³ = {2 * K**3 / 1e9:.0f} GFLOP，按 989 TFLOPS 只要 {2 * K**3 / 989e12 * 1e3:.2f} ms")
```

输出：

```
n = 1000，每块 256 个线程 → 4 个块，共 1024 个线程，24 个线程越界、什么也不做
  blockIdx = 0，threadIdx =   0 → i =    0
  blockIdx = 1，threadIdx =   5 → i =  261
  blockIdx = 3，threadIdx = 231 → i =  999
  blockIdx = 3，threadIdx = 255 → i = 1023（越界，跳过）
20×30 的矩阵、16×16 的块 → 网格 2×2 个块；块 (1, 1) 里的线程 (2, 13) 负责 (18, 29)
8×8 矩阵乘法，块大小 1：结果正确 True，从显存读了 1024 个元素
8×8 矩阵乘法，块大小 2：结果正确 True，从显存读了 512 个元素
8×8 矩阵乘法，块大小 4：结果正确 True，从显存读了 256 个元素
K = 4096，块大小  1：从显存读   274.9 GB，按 3.35 TB/s 需要   82.05 ms
K = 4096，块大小 16：从显存读    17.2 GB，按 3.35 TB/s 需要    5.13 ms
K = 4096，块大小 64：从显存读     4.3 GB，按 3.35 TB/s 需要    1.28 ms
计算量 2K³ = 137 GFLOP，按 989 TFLOPS 只要 0.14 ms
```

前五行是向量加法：4 个块、1024 个线程，24 个越界，块 3 的第 255 个线程算出的下标 1023 越界，跳过。第六行是二维网格的例子，和上文一致。接下来三行是 8 × 8 的分块矩阵乘法：结果都正确，块大小从 1 到 4，读取量从 1024 个元素降到 256 个，正好是 1/T。最后四行是 K = 4096 时的估算。

第二段代码复现这一讲后半部分的数字：线程块的驻留与占用率、在途字节、块大小与读写量、bank 冲突，以及双缓冲的时间。

```python
# 1. 一个线程块能不能驻留：每类资源各允许几块，取最小（H100 一个 SM）
SM = {"线程槽": 2048, "寄存器": 65536, "共享内存": 233472, "块数上限": 32}
def resident(threads, regs, smem):
    need = {"线程槽": threads, "寄存器": threads * regs, "共享内存": smem + 1024, "块数上限": 1}
    per = {k: SM[k] // need[k] for k in SM}
    n = min(per.values())
    return per, n, n * threads // 32 / 64
per, n, occ = resident(256, 128, 96 * 1024)                # 128×128 输出块、K 步 64：A、W 块各 16 KiB + FP32 累加器 64 KiB
print("每类资源允许的块数：", per, f"→ 驻留 {n} 块，占用率 {occ:.0%}")

# 2. 在途字节够不够隐藏 600 ns 的显存延迟（Little 定律：带宽 × 延迟）
need = 3.35e12 * 600e-9 / 132
have = n * 8 * 4 * 512                                     # 每块 8 个 warp × 每 warp 4 条未完成加载 × 512 字节
print(f"每个 SM 需要 {need/1e3:.1f} KB 在途，驻留的 warp 能提供 {have/1024:.0f} KiB（{have/need:.2f} 倍）")
need2, have2 = 3.35e12 * 1000e-9 / 132, n * 8 * 2 * 512
print(f"延迟 1000 ns、每 warp 只挂 2 条：需要 {need2/1e3:.1f} KB，只有 {have2/1024:.0f} KiB（{have2/need2:.2f} 倍）")

# 3. 输出块越大，输入重读越少，但局部存储越大（C = A·W，M=1024，K=4096，N=12288，BF16，k=32）
M, K, N, k = 1024, 4096, 12288, 32
for t in [32, 64, 128]:
    smem = 2 * t * k * 2 + 4 * t * t
    traffic = 2 * M * K * N / t + 2 * K * N * M / t + 2 * M * N
    print(f"{t:>3}×{t:<3}：局部存储 {smem/1024:3.0f} KiB，接口读写 {traffic/2**20:5.0f} MiB，算术强度 {2*M*K*N/traffic:4.1f} FLOP/byte")

# 4. 共享内存 bank：32 个 lane 读一列，行跨度 32 个字时全落在同一个 bank
for stride in [32, 33]:
    banks = {(r * stride) % 32 for r in range(32)}
    print(f"行跨度 {stride}：用到 {len(banks)} 个 bank，要 {32 // len(banks)} 轮")
print("异或 swizzle（第 r 行第 c 列放到 c^r 列）：用到", len({(r * 32 + (0 ^ r)) % 32 for r in range(32)}), "个 bank")

# 5. 双缓冲：每块搬 1.29 μs、算 0.28 μs，4 块
tm, tc, n4 = 1.29, 0.28, 4
print(f"串行 {n4*(tm+tc):.2f} μs；双缓冲 {tm + (n4-1)*max(tm, tc) + tc:.2f} μs；计算单元忙的比例 {n4*tc/(tm + (n4-1)*max(tm, tc) + tc):.0%}")
```

输出：

```
每类资源允许的块数： {'线程槽': 8, '寄存器': 2, '共享内存': 2, '块数上限': 32} → 驻留 2
块，占用率 25%
每个 SM 需要 15.2 KB 在途，驻留的 warp 能提供 32 KiB（2.15 倍）
延迟 1000 ns、每 warp 只挂 2 条：需要 25.4 KB，只有 16 KiB（0.65 倍）
 32×32 ：局部存储    8 KiB，接口读写  6168 MiB，算术强度 15.9 FLOP/byte
 64×64 ：局部存储   24 KiB，接口读写  3096 MiB，算术强度 31.8 FLOP/byte
128×128：局部存储   80 KiB，接口读写  1560 MiB，算术强度 63.0 FLOP/byte
行跨度 32：用到 1 个 bank，要 32 轮
行跨度 33：用到 32 个 bank，要 1 轮
异或 swizzle（第 r 行第 c 列放到 c^r 列）：用到 32 个 bank
串行 6.28 μs；双缓冲 5.44 μs；计算单元忙的比例 21%
```

对照正文：驻留 2 块、占用率 25%；在途 32 KiB 是需求 15.2 KB 的 2.15 倍，延迟变长后只有 0.65 倍；三种块的读写量 6168、3096、1560 MiB；行跨度 32 要 32 轮，33 或 swizzle 只要 1 轮；串行 6.28 μs、双缓冲 5.44 μs。

容易混淆的地方

1. **CUDA 的“块”和 PagedAttention 的“块”不是一回事。**这里的块（block）是一组线程；第 75 讲的块是 KV Cache 的一段存储。英文里都叫 block，要看上下文。

2. **线程多不等于快。**线程只是把工作切开的方式。如果每个线程都在等显存，开再多线程也快不了；性能取决于数据怎么在各级存储之间流动。

3. **共享内存要手动管理。**它不像 CPU 的缓存那样自动工作，什么时候搬什么数据进来、什么时候同步，都要在 kernel 里写清楚。这正是写 CUDA kernel 难的地方，也是下一讲 Triton 要简化的地方。

4. **占用率高不等于快，低也不一定慢。**占用率的作用是提供足够的在途请求去隐藏延迟；在途字节已经够了，再提高占用率没有收益。反过来，块做大省了读写，却可能因驻留块变少而拿不到带宽。

5. **双缓冲只能藏住较短的那一段。**搬比算慢时，重叠之后的时间由搬移决定，加更多的槽也没用。

回顾

CUDA 的程序是一个 kernel：写清楚“一个线程做什么”，启动时指定网格有几个块、每块几个线程，所有线程执行同一段代码，用 blockIdx、blockDim、threadIdx 算出自己负责的数据，越界的线程跳过。一个块整个跑在一个 SM 上，块内线程通过共享内存合作、可以同步；块与块之间不能直接合作。线程每 32 个一组成为 warp，同时执行同一条指令，所以要避免分支发散、让相邻线程读相邻的数据。存储分寄存器、共享内存、全局内存三级；矩阵乘法按 T × T 分块、在共享内存里复用，从显存读的数据降到 1/T，K = 4096 时从 274.9 GB 降到 4.3 GB（T = 64）。

写真实的 kernel 还要算 SM 的资源：128 × 128 的矩阵乘块每块要 96 KiB 共享内存、每线程 128 个寄存器，H100 的一个 SM 只住得下 2 块，占用率 25%，在途字节仍是需求的 2.15 倍。块越大输入重读越少（64 × 64 要 3096 MiB，128 × 128 只要 1560 MiB），但住得下的块越少。读一列时用 padding 或 swizzle 避开 bank 冲突；双缓冲让搬和算错开，可搬比算慢时，总时间仍由搬移决定。

分块解决了“从显存读多少”。可一个矩阵乘法真的能跑到峰值吗？Tensor Core 怎么被喂进数据，矩阵怎么分给 132 个 SM，什么形状会让大部分 SM 闲着？下一讲：**矩阵乘法 Kernel 与 Tensor Core**。

第87讲 矩阵乘法 Kernel 与 Tensor Core

重点

60 GPU 架构　62 Roofline 模型　86 CUDA 编程模型

一个大模型的前向，九成以上的计算是矩阵乘法：每层的 q、k、v、o 四个投影，FFN 的三个矩阵，最后的输出层。第 60 讲说过，H100 的 BF16 峰值 989 TFLOPS 来自 Tensor Core，只有组织成矩阵乘法的计算才用得上它；第 86 讲又讲了矩阵乘法 kernel 的第一个关键：把 A、B 分块搬进共享内存反复用，从显存读的量降到原来的 1/T。

这一讲接着往下走。有了分块，一个矩阵乘法就能跑到峰值了吗？不一定。同样是 4096 维的 K，1408 × 1408 的矩阵乘法和 1536 × 1536 的，后者计算量只多 19%，用时却是前者的两倍；decode 时 16 个请求乘一个 4096 × 1024 的权重，GPU 只用上了峰值的百分之一。这一讲讲清楚这些现象：Tensor Core 是怎么被喂进数据的，矩阵怎么分给 132 个 SM，什么形状会让 GPU 闲着，以及 cuBLAS、CUTLASS 这些库是怎么对付的。

三层分块：线程块、warp、Tensor Core 指令

第 86 讲的分块只有一层：一个线程块负责 C 的一个 T × T 小块，每个线程算其中几个元素。真实的矩阵乘法 kernel 有三层，每一层对应一级存储：

- **线程块级**：一个线程块负责 C 的一大块，比如 128 × 128。它沿着 K 方向一段一段地前进，每段把 A 的 128 × 32、B 的 32 × 128 搬进**共享内存**。
- **warp 级**：块里的几个 warp（每个 32 个线程）各分到其中一小块，比如 64 × 64，结果累加在 warp 自己的**寄存器**里。
- **指令级**：warp 把自己那一小块再拆成 Tensor Core 指令能处理的形状。Ampere（A100）上的指令是 `mma.sync.m16n8k16`：一个 warp 一次算 16 × 8 的输出，沿 K 走 16 步，即 16 × 16 的 A 乘 16 × 8 的 B。

Hopper（H100）换了一套更大的指令 `wgmma`：四个 warp 组成一个 warpgroup，一次算 64 × N 的输出（N 最大 256），K 方向一次 16；A、B 可以直接从共享内存读，不必先搬进寄存器，而且是**异步**的，发出指令后可以去做别的事。配合 TMA（张量内存加速器），线程块只要告诉硬件“把这块搬进共享内存”，搬运也由硬件异步完成。这样，搬数据和算数据就能完全重叠起来：第 86 讲讲的双缓冲，在 Hopper 上变成了几段流水，一段在算，几段在搬。

为什么要这么多层？因为每往下一层，存储更小、更快：显存 80 GB、3.35 TB/s；共享内存每个 SM 最多 228 KB；寄存器每个 SM 256 KB，延迟最低。数据每被搬上一层，就在这一层被尽量多地复用，再往上一层只需要很少的流量。分块的形状就是在这几层的容量、复用次数和指令的形状之间找平衡。

把矩阵分给 132 个 SM

现在看一个完整的矩阵乘法 C[M, N] = A[M, K] · B[K, N] 怎么跑。输出切成 Mt × Nt 的块，一共 ⌈M/Mt⌉ × ⌈N/Nt⌉ 个，每个块交给一个线程块。H100 有 132 个 SM，我们假设每个 SM 同一时刻算一个块（真实 kernel 有时能放两个，道理一样）。块数超过 132 时，先上 132 个，算完再上下一批，每一批叫一**波**（wave）。

这里藏着两种浪费。

补零（tile quantization）：M 或 N 不是块大小的整数倍时，边上的块只有一部分是有用的，但 Tensor Core 照样按整块算。M = 1000、块 128 时，要算 8 × 128 = 1024 行，浪费 2.3%，不大。

波次（wave quantization）：块数不是 132 的整数倍时，最后一波只有一部分 SM 有活干，其余的空等。这一项可能很大：

- 1408 × 1408：每边 11 块，一共 121 块，一波就能装下，132 个 SM 里 121 个在算，利用率 91.7%。
- 1536 × 1536：每边 12 块，一共 144 块，第一波 132 块，第二波只有 12 块，另外 120 个 SM 闲着。两波的时间算了 1.09 波的活，利用率 54.5%。

计算量只多了 19%，时间却多了一倍。这就是**波次悬崖**：块数刚过 132 的整数倍，时间就跳一级。下面的演示画出了每一波里有多少个 SM 在干活，可以选几个预设，也可以自己拖 M、N、K：

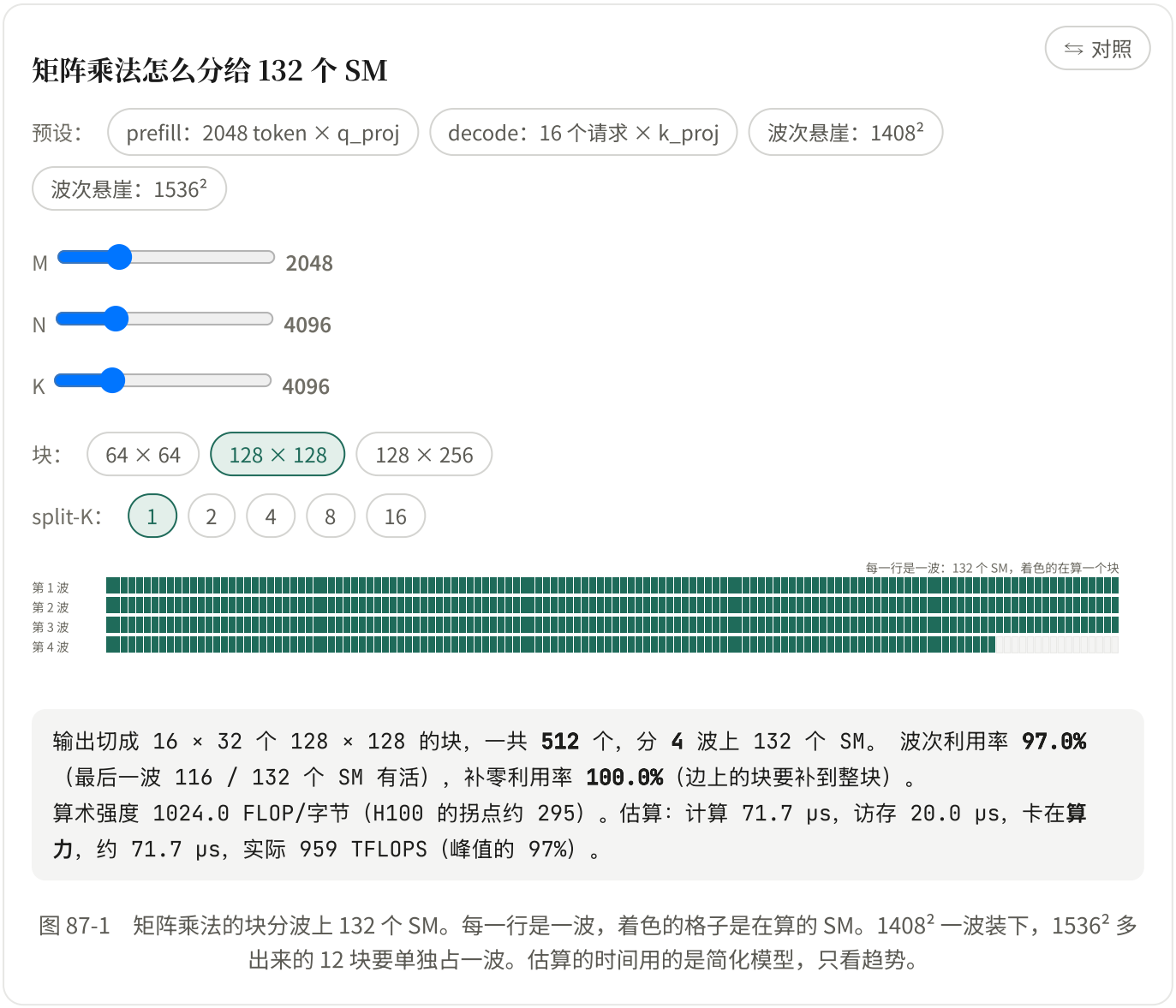


图 87-1　矩阵乘法的块分波上 132 个 SM。每一行是一波，着色的格子是在算的 SM。1408^2 一波装下，1536^2 多出来的 12 块要单独占一波。估算的时间用的是简化模型，只看趋势。

decode：块太少，SM 闲着

再看 decode。batch 里有 16 个请求，每个请求一个 token，乘 Llama-3-8B 的 k_proj（4096 → 1024）：M = 16，N = 1024，K = 4096。

先看算术强度（第 62 讲）。计算 2MNK，读的主要是权重 2KN 字节，强度约等于 M：16 FLOP/字节。H100 的拐点是 989 ÷ 3.35 ≈ 295 FLOP/字节，16 远在左边，这一步**受带宽限制**。理想情况下，时间就是把 8.5 MB（权重 8.4 MB，加上很小的输入和输出）读一遍：约 2.6 µs。

可是按普通的分块，输出只有 16 × 1024：块大小 64 × 64 时，只有 1 × 16 = 16 个块，132 个 SM 里只有 16 个在干活。受带宽限制的 kernel 也需要足够多的 SM 同时发出读请求，才能把显存带宽跑满（第 86 讲说的“在途请求”）；16 个 SM 远远不够。

解决办法是 split-K：既然 M、N 方向切不出足够的块，就沿 K 方向切。K = 4096 切成 8 段，每个块只算 512 长的一段，得到一个部分和；8 段的部分和最后再加起来。块数变成 128，几乎占满所有 SM。代价是多了部分和的读写和一次归约，但比起让 116 个 SM 闲着，划算得多。第 85 讲的 Flash-Decoding 是同一个思路：decode 注意力的查询只有一行，就沿 KV 的长度方向切开。

顺带一提，M = 16 时 Tensor Core 本身也是浪费的：`wgmma` 的输出一次就是 64 行，只有 16 行有用，四分之三的乘加白算了。但这不要紧：这一步受带宽限制，Tensor Core 本来就在等数据。decode 的矩阵乘法要优化的是读，不是算。

用代码算一遍

把上面的简化模型写成代码，几种形状一起算：每波的时间按一个 SM 算完一块计，访存时间按读 A、B、写 C 计，活跃的块少于一半的 SM 时，假设带宽按比例打折。

```
import math
SM, PEAK, BW = 132, 989e12, 3.35e12            # H100 SXM: SM 数、BF16 稠密峰值、显存带宽

def gemm(M, N, K, Mt=128, Nt=128, split=1):
    tiles = math.ceil(M / Mt) * math.ceil(N / Nt) * split
    waves = math.ceil(tiles / SM)
    t_comp = waves * 2 * Mt * Nt * (K / split) / (PEAK / SM)      # 每波的时间 = 一个 SM 算完一块
    nbytes = 2 * (M * K + K * N + M * N) + (8 * M * N * split if split > 1 else 0)
    t_mem = nbytes / (BW * min(1, min(tiles, SM) / (SM / 2)))     # 活跃的块少于 66 个时带宽打折
    t = max(t_comp, t_mem)
    return tiles, waves, tiles / (waves * SM), t, 2 * M * N * K / t / 1e12

print("形状 M × N × K (块)              块数  波数  波次利用率  时间(μs)  TFLOPS")
for name, args in [("2048 × 4096 × 4096", (2048, 4096, 4096)),
                   ("1408 × 1408 × 4096", (1408, 1408, 4096)),
                   ("1536 × 1536 × 4096", (1536, 1536, 4096)),
                   ("16 × 1024 × 4096 (64×64) ", (16, 1024, 4096, 64, 64)),
                   ("同上, split-K = 8", (16, 1024, 4096, 64, 64, 8))]:
    tiles, waves, eff, t, tf = gemm(*args)
    print(f"{name:<28}{tiles:>6}{waves:>6}{eff:>10.1%}{t * 1e6:>10.1f}{tf:>8.0f}")
```

形状 M × N × K（块）	块数	波数	波次利用率	时间（μs）	TFLOPS
2048 × 4096 × 4096	512	4	97.0%	71.7	959
1408 × 1408 × 4096	121	1	91.7%	17.9	907
1536 × 1536 × 4096	144	2	54.5%	35.8	539
16 × 1024 × 4096（64×64）	16	1	12.1%	10.5	13
同上，split-K = 8	128	1	97.0%	2.9	47

逐行看：

- **2048 × 4096 × 4096**：prefill 时 2048 个 token 乘 q_proj。512 块，4 波，最后一波 116 / 132，利用率 97%。强度上千，受算力限制，模型估出 959 TFLOPS。真实的 cuBLAS 在这种形状上通常能到峰值的七八成：模型没有算流水的启动和收尾、指令发射和同步的开销。
- **1408^2 和 1536^2**：一波和两波，时间 17.9 对 35.8 μs，正好两倍，实际算力从 907 掉到 539 TFLOPS。
- **16 × 1024 × 4096**：16 个块，时间 10.5 μs，只有 13 TFLOPS，峰值的 1.3%，比理想的 2.6 μs 慢了 4 倍。
- **split-K = 8**：128 个块，2.9 μs，接近只读一遍的 2.6 μs。

要点

矩阵乘法 kernel 有三层分块：线程块（共享内存）、warp（寄存器）、Tensor Core 指令。分块之后，输出的块按“波”分给 SM：块数刚超过 SM 数的整数倍会多出一波（波次悬崖）；块太少时大部分 SM 闲着，连带宽都跑不满，要沿 K 切开（split-K）。

库是怎么对付的

上面这些问题，矩阵乘法库都要处理：

- **按形状选配置**。cuBLAS 内部有很多套 kernel（不同的块大小、流水段数、是否 split-K），根据 M、N、K 用启发式规则或预先测好的表挑一套。CUTLASS 是 NVIDIA 开源的 C++ 模板库，把这些配置都做成模板参数，推理框架可以为自己常见的形状专门实例化。Triton（下一讲）则用自动调优：同一个 kernel 写好几种配置，第一次运行时逐个试，记住最快的。
- **Stream-K**。与其把输出切成块、按波分配，不如把所有块的“K 方向的每一段”加起来，平均分给 132 个 SM，每个 SM 分到的活一样多，最后再把跨 SM 的部分和合起来。这样就没有最后一波的空闲了。CUTLASS 里有这种调度方式。

- **推理引擎里的特化**。decode 的 M（batch 里的请求数）每一步都在变，引擎会把它补齐到几个固定的档位（第 89 讲的 CUDA Graph 也要这样做），每个档位用调好的 kernel。很小的 M 有时直接用专门的矩阵 × 向量 kernel，不走 Tensor Core。
- **形状本身也可以调**。模型设计时把隐藏维度、FFN 宽度、词表大小取成 64 或 128 的倍数，就是为了避免补零；Llama 的词表 128,256 = 128 × 1002 就是一例。

容易混淆的地方

- **“用上了 Tensor Core”不等于“跑到了峰值”**。块的形状、波次、流水、数据搬运，任何一项没做好，Tensor Core 都会在等。
- **矩阵大一点不一定更快，也不一定更慢**。同一个 kernel，矩阵从 1408 长到 1536，时间翻倍；再长到 2048，块数 256，2 波，利用率 97%，时间和 1536 一样。性能曲线是阶梯形的，不是平滑的。
- **decode 的矩阵乘法慢，不是因为 Tensor Core 不够快**。它受带宽限制，要优化的是让足够多的 SM 一起读，以及少读（量化，第 90 讲起）。

回顾

矩阵乘法 kernel 有三层分块：线程块把 A、B 的一段搬进共享内存，warp 在寄存器里累加自己的小块，Tensor Core 指令一次算一个小矩阵乘加（A100 是一个 warp 算 16 × 8，H100 的 wgmma 是四个 warp 算 64 × N，配合 TMA 异步搬运）。输出的块按波分给 132 个 SM：1408^2 一波装下，1536^2 多一波、时间翻倍；decode 时 16 × 1024 的输出只有 16 个块，只用了峰值的 1.3%，沿 K 切 8 份后接近只读一遍权重的理想时间。库用按形状选配置、Stream-K、档位特化来对付这些问题。

写一个这样的 kernel，要自己管三层分块、共享内存、同步和流水，用 CUDA 写非常繁琐。有没有办法只描述“一个块怎么算”，其余交给编译器？下一讲：**Triton 入门**。

第88讲 Triton 入门

83 算子融合　86 CUDA 编程模型

第 86 讲用 CUDA 写了向量加法，看上去不难。可一旦要写一个真正快的 kernel，事情就多了：矩阵乘法要手动把数据搬进共享内存、安排每个线程搬哪几个元素、在合适的地方同步、让相邻的线程读相邻的地址、调用 Tensor Core 的专用指令、让搬下一块数据和算这一块重叠起来……一个高性能的矩阵乘法 kernel，CUDA 代码动辄几百上千行。

Triton 是 OpenAI 在 2021 年开源的一种写 GPU kernel 的语言，嵌在 Python 里。它的思路是把抽象的层级往上提一层：**我们只描述“一个块的数据怎么算”，块里的线程怎么分工，交给编译器**。这一讲用三个例子入门：融合的 SwiGLU 激活、RMSNorm、矩阵乘法。

从“一个线程”到“一个块”

CUDA 的视角是一个线程：我是第几个线程，我算哪一个元素。Triton 的视角是一个**程序**（program）：我是第几个程序，我处理哪一**块**元素。一个 Triton 程序大致对应 CUDA 的一个线程块，但我们不用管块里的线程。

在 Triton 里，变量通常不是一个数，而是一整块数（一个小张量）。比如 `tl.arange(0, 256)` 就是 [0, 1, …, 255] 这 256 个数；`tl.load` 一次读进一整块；加减乘除都是对整块逐元素进行的。编译器拿到这样的代码，自己决定用多少个线程、每个线程处理几个元素、怎么合并访存、要不要用共享内存。

例一：融合的 SwiGLU 激活

第 83 讲说，`F.silu(g) * u` 逐个算子执行要读写显存 8 遍，融合成一个 kernel 只要 3 遍。用 Triton 写出这个融合 kernel：

```python
import triton
import triton.language as tl

@triton.jit
def silu_mul(g_ptr, u_ptr, y_ptr, n, BLOCK: tl.constexpr):
    pid = tl.program_id(0)                          # 我是第几个程序
    offs = pid * BLOCK + tl.arange(0, BLOCK)        # 我负责的一整块下标
    mask = offs < n                                 # 最后一块可能越界
    g = tl.load(g_ptr + offs, mask=mask)            # 一次读进一整块
    u = tl.load(u_ptr + offs, mask=mask)
    y = g * tl.sigmoid(g) * u                       # 中间结果只在片上
    tl.store(y_ptr + offs, y, mask=mask)            # 只写最终结果

def silu_and_mul(g, u):                             # g、u 是 GPU 上的 PyTorch 张量
    y = torch.empty_like(g)
    n = g.numel()
    grid = (triton.cdiv(n, 1024),)                  # 网格：要几个程序
    silu_mul[grid](g, u, y, n, BLOCK=1024)
    return y
```

逐行看：

- `@triton.jit` 表示这是一个 Triton kernel，第一次调用时编译成 GPU 代码。
- `tl.program_id(0)` 是这个程序的编号，相当于 CUDA 的 `blockIdx.x`。
- `offs` 是一整块下标：程序 pid 负责第 pid × BLOCK 到 pid × BLOCK + BLOCK − 1 个元素。
- `mask` 标出哪些下标没越界。`tl.load` 和 `tl.store` 带上 mask，越界的位置不读不写。CUDA 里每个线程写一个 `if (i < n)`，这里对整块用一个 mask。
- `BLOCK: tl.constexpr` 表示 BLOCK 是编译时的常数（`tl.arange` 的长度必须是编译时已知的 2 的幂）。
- 启动时 `silu_mul[grid](...)`，方括号里是网格，也就是要开几个程序。

n = 1000、BLOCK = 256 时，网格是 ⌈1000 / 256⌉ = 4 个程序；最后一个程序的 offs 是 768 到 1023，mask 只有前 232 个为真。

例二：RMSNorm，一个程序处理一行

RMSNorm（第 36 讲）要对每一行求平方和，是一个归约。用 Triton 写，可以让一个程序处理一整行：整行读进来，求和、归一化、乘权重，写回去，中间结果都在片上。

```
@triton.jit
def rmsnorm(x_ptr, w_ptr, out_ptr, d, eps, BLOCK: tl.constexpr):
    row = tl.program_id(0)                          # 一个程序负责一行
    cols = tl.arange(0, BLOCK)                      # BLOCK ≥ d，取 2 的幂
    mask = cols < d
    x = tl.load(x_ptr + row * d + cols, mask=mask, other=0.0).to(tl.float32)
    r = x * tl.rsqrt(tl.sum(x * x) / d + eps)       # tl.sum：整块求和
    w = tl.load(w_ptr + cols, mask=mask)
    tl.store(out_ptr + row * d + cols, r * w, mask=mask)
```

网格就是行数（token 数）。`other=0.0` 让越界的位置读成 0，不影响平方和。`tl.sum` 是块内归约，CUDA 里要自己写 warp 内的规约和共享内存里的规约，这里一行搞定。计算前转成 FP32，是为了平方和不溢出、不丢精度。

例三：矩阵乘法

最后看矩阵乘法。第 86 讲说过，快的矩阵乘法要分块：每个线程块负责 C 的一个小块，沿 K 方向一段一段地把 A、B 的小块搬进片上，复用之后再搬下一段。用 Triton 写，结构几乎就是这句话的直译：

```python
@triton.jit
def matmul(a_ptr, b_ptr, c_ptr, M, N, K,
           stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
           BM: tl.constexpr, BN: tl.constexpr, BK: tl.constexpr):
    pid_m, pid_n = tl.program_id(0), tl.program_id(1)    # 二维网格：负责 C 的哪一块
    rm = pid_m * BM + tl.arange(0, BM)
    rn = pid_n * BN + tl.arange(0, BN)
    rk = tl.arange(0, BK)
    acc = tl.zeros((BM, BN), dtype=tl.float32)           # 累加器
    for k in range(0, K, BK):                            # 沿 K 方向一块一块地走
        a = tl.load(a_ptr + rm[:, None] * stride_am + (k + rk)[None, :] *
stride_ak)
        b = tl.load(b_ptr + (k + rk)[:, None] * stride_bk + rn[None, :] *
stride_bn)
        acc += tl.dot(a, b)                              # 小块矩阵乘法，编译器交给
Tensor Core
    tl.store(c_ptr + rm[:, None] * stride_cm + rn[None, :] * stride_cn, acc)
```

（为了简洁，这里假设 M、N、K 都是块大小的整数倍，省略了 mask。）`rm[:, None]` 和 `rn[None, :]` 是 NumPy 式的广播，拼出一个 BM × BK 的二维下标块；stride 是每一维相邻元素在内存里隔了多少个元素。`tl.dot` 是块与块的矩阵乘法，编译器把它变成 Tensor Core 指令，并自动安排共享内存。

块大小选多少最快，取决于矩阵的形状和硬件。Triton 提供 `@triton.autotune`：列出几组候选的块大小、warp 数（`num_warps`）、流水线级数（`num_stages`，即提前搬几块数据，让搬运和计算重叠），第一次运行时逐个试，选最快的那组记下来。

下面的演示把代码和它处理的数据块并排放在一起。向量模式下点一行看那个程序的 offs 和 mask；矩阵乘法模式下点 C 的一块选程序，拖动 k 看每次循环取了 A、B 的哪两块。

Triton：一个程序处理一整块

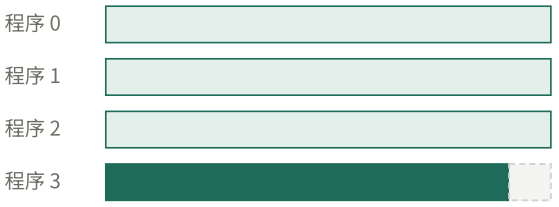
⇆ 对照

向量：融合的 SwiGLU　矩阵乘法

n = **1000**

BLOCK：　128　256　512

```
@triton.jit
def silu_mul(g_ptr, u_ptr, y_ptr, n, BLOCK: tl.con
    pid = tl.program_id(0)
    offs = pid * BLOCK + tl.arange(0, BLOCK)
    mask = offs < n
    g = tl.load(g_ptr + offs, mask=mask)
    u = tl.load(u_ptr + offs, mask=mask)
    y = g * tl.sigmoid(g) * u
    tl.store(y_ptr + offs, y, mask=mask)
```



每行一个程序（256 个位置），灰色 = mask 为假

网格 = ⌈1000 / 256⌉ = 4 个程序（总位置 1024 个，24 个被 mask 掉）。点左边的一行选程序。
程序 3：offs = 3 × 256 + [0, 1, …, 255] = [768, …, 1023]；mask = offs < 1000 → 232 个为真，后 24 个不读不写
一个程序就是 CUDA 里的一个线程块，但我们只写“这一块数据做什么”，块里的线程怎么分工、warp、合并访存，都由编译器处理。

图 88-1　Triton 的程序与数据块。左边高亮的代码行，作用在右边高亮的数据块上：向量模式里一个程序处理一整块元素，最后一块用 mask 盖住越界的位置；矩阵乘法里一个程序负责 C 的一块，沿 K 方向取 A、B 的小块相乘累加。

要点

Triton 让我们以“一块数据”为单位写 kernel：用 program_id 确定自己负责哪一块，用 arange 生成整块下标，用 mask 处理越界，load、整块计算、store。线程分工、合并访存、共享内存、Tensor Core 都由编译器处理，写融合 kernel 和分块矩阵乘法变得像写 NumPy。

候选怎么挑：先排除，再估算，最后实测

`@triton.autotune` 会把每组候选都编译、跑一遍。候选多了，搜索本身就很贵，所以编译器和工程师一般按下面的顺序缩小范围（第 86 讲的块大小分析就是其中一步）：

步骤	要填的内容	依据
固定目标	输入形状、精度、允许的误差、每种形状的调用频数	比较的是同一项数学工作、同一种负载
列出候选	块大小、循环次序、缓冲槽数（`num_stages`）、warp 数、融合范围	编译器和硬件是否支持
检查可行性	共享内存、寄存器、线程数、对齐、数据依赖	编译的资源报告和正确性检查
估计时间	计算量、实际读写量、启动和布局转换	各项资源的服务时间和关键路径
实测并选择	代表形状的耗时、波动，以及放在完整算子链里的表现	按实际负载的总成本选，相近的候选都保留

实测的差别往往比估算大。同一个矩阵转置程序，在 RTX PRO 6000 上用 16 × 16 的块约 7.9 μs，用 32 × 32 约 2.9 μs，快 2.7 倍：块的边长翻倍，元素数变成四倍，存储布局和线程协作也一起变了。

要看调用的比例，而不是单个形状。某个算子有 A、B 两种输入形状，原实现都要 10 μs。新实现处理 A 只要 5 μs，处理 B 却要 20 μs。设 A 占全部调用的比例为 p，新实现的平均时间是 $5p + 20(1 - p) = 20 - 15p$ μs，要比 10 μs 快，需要 $p > 2/3$。只在形状 A 上测，会以为快了一倍；放到真实的调用比例里，可能反而变慢。所以调优可以按形状分别选实现，也可以按实际调用频数选一个总成本最低的。

搜索也要回本。如果一次搜索要 10 分钟，选出来的实现每次调用省 5 μs，要调用 600 s ÷ 5 μs = **1.2 亿次**才收回。每秒调用 1 万次约 3.3 小时，每秒 100 次要 14 天。所以优先调那些调用频繁、会长期使用的形状。

动态形状：通用、分桶，还是逐个特化

Triton 的 kernel 是按形状和 `tl.constexpr` 参数即时编译的。推理时每一步的 token 数都在变，运行时有三种选择：**通用**程序按当前形状计算下标和边界；**分桶**把输入补齐到少数几个代表形状，复用这些形状的程序；**特化**为每个常见形状单独编译一份。越利用已知的形状，程序越快，但要编译和保存的程序也越多。

算一个例子：10 次 FFN 为一组，8 次 256 行、1 次 1536 行、1 次 2048 行，共 5632 行。用 512 和 2048 两个桶，实际要算 8192 行，多算 45%。在 RTX PRO 6000（BF16 峰值 503.8 TFLOP/s）上，设三种程序分别发挥峰值的 20%、40%、50%，准备时间（编译、调优）分别为

100、400、900 ms：

策略	准备	每组	说明
通用	100 ms	16.9 ms	按实际 5632 行算
分桶	400 ms	12.3 ms	多算 45%，但效率翻倍，仍比通用快
特化	900 ms	6.8 ms	每个形状一份程序

跑 r 组的总时间是 准备 + r × 每组。每组越快的方案准备越久，所以最省时的方案随 r 变化：前 65 组通用最省，第 66 组起分桶，第 91 组起特化。只跑 70 组时，分桶最快（1.26 s），特化反而最慢（1.37 s）。如果三种程序都已经编译并缓存，准备时间为零，特化从第一组起就最快。所以运行时的选择同时取决于形状的分布、要跑多少次，以及程序是否已经在缓存里。

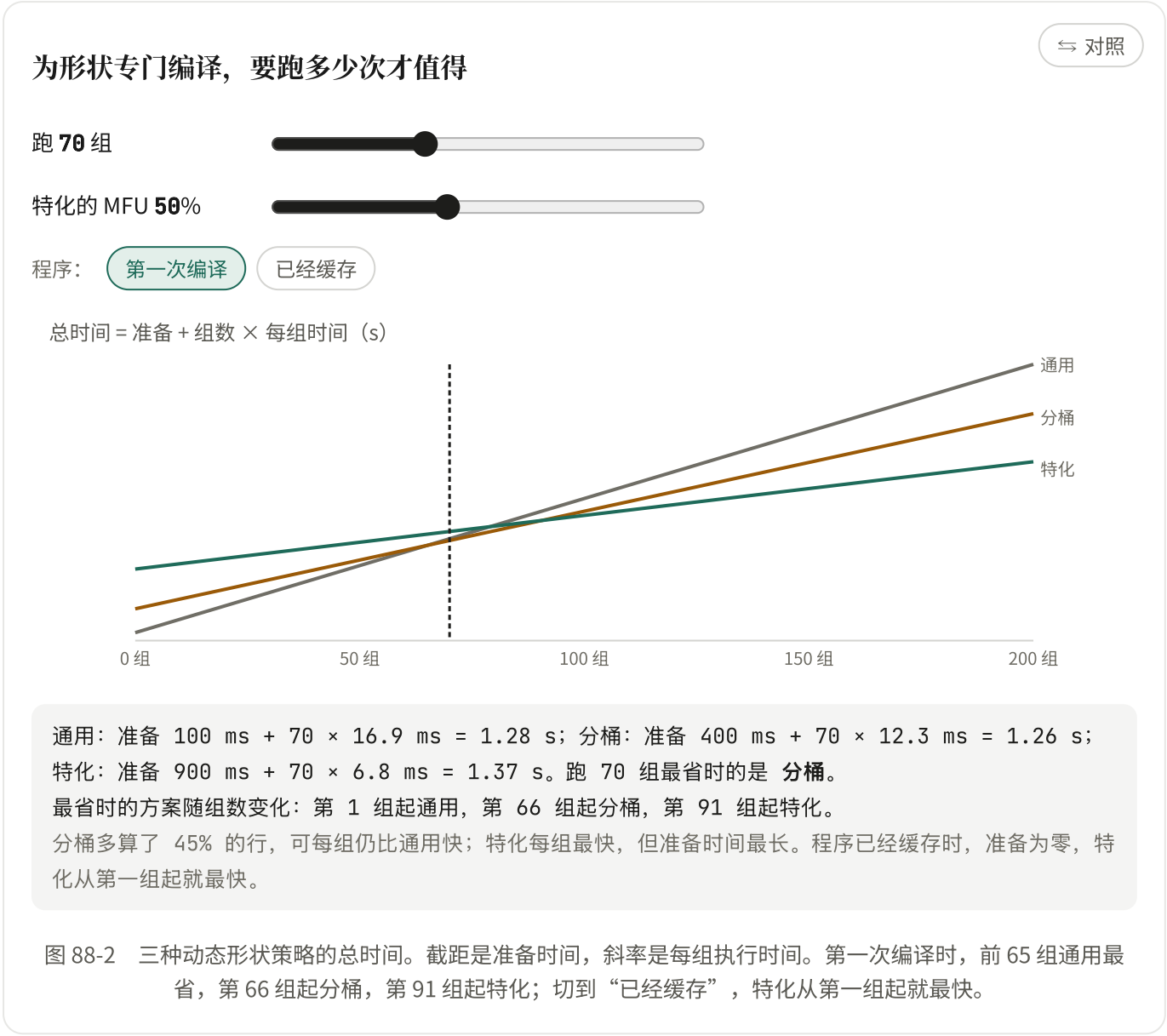


图 88-2　三种动态形状策略的总时间。截距是准备时间，斜率是每组执行时间。第一次编译时，前 65 组通用最省，第 66 组起分桶，第 91 组起特化；切到“已经缓存”，特化从第一组起就最快。

用代码复现

这里没有 GPU，我们用 NumPy 模拟 Triton 的执行：写两个小函数模拟带 mask 的 `tl.load` 和 `tl.store`，然后把上面三个 kernel 的逻辑原样写一遍，用循环依次“启动”网格里的每个程序（在 GPU 上它们是同时运行的），和直接用 NumPy 算的结果对比。

```python
import numpy as np

# 用 NumPy 模拟 Triton 的几个基本操作：一个“程序”一次处理一整块数据
def load(x, offs, mask, other=0.0):
    return np.where(mask, x[np.where(mask, offs, 0)], other)          # 越界的位置不读，填 other
def store(x, offs, val, mask):
    x[offs[mask]] = val[mask]

# 1. 融合的 SwiGLU 激活：y = silu(g) * u。每个程序处理 BLOCK 个元素
def silu_mul_kernel(pid, g, u, y, n, BLOCK):
    offs = pid * BLOCK + np.arange(BLOCK)                 # 这个程序负责的下标（一整块）
    mask = offs < n                                       # 最后一块可能越界
    gv, uv = load(g, offs, mask), load(u, offs, mask)
    store(y, offs, gv / (1 + np.exp(-gv)) * uv, mask)
    return int(mask.sum())

n, BLOCK = 1000, 256
g, u = np.random.default_rng(0).standard_normal((2, n)); y = np.zeros(n)
grid = (n + BLOCK - 1) // BLOCK
valid = [silu_mul_kernel(pid, g, u, y, n, BLOCK) for pid in range(grid)]   # GPU 上这几个程序是同时运行的
print(f"n = {n}, BLOCK = {BLOCK} → {grid} 个程序，各自处理的有效元素：{valid}")
print("结果正确：", np.allclose(y, g / (1 + np.exp(-g)) * u))

# 2. RMSNorm：每个程序处理一行（BLOCK 要不小于行长 d），整行留在片上算平方和
def rmsnorm_kernel(row, x, w, out, d, BLOCK, eps=1e-5):
    offs = row * d + np.arange(BLOCK); mask = np.arange(BLOCK) < d
    xv = load(x, offs, mask)
    r = xv / np.sqrt((xv * xv).sum() / d + eps)           # 越界的位置读成 0，不影响平方和
    store(out, offs, r * load(w, np.arange(BLOCK), mask), mask)
rows, d = 3, 10
x = np.random.default_rng(1).standard_normal(rows * d); w = np.linspace(0.5, 1.5, d); out = np.zeros_like(x)
for row in range(rows): rmsnorm_kernel(row, x, w, out, d, BLOCK=16)
X = x.reshape(rows, d)
print("RMSNorm 结果正确：", np.allclose(out.reshape(rows, d), X / np.sqrt((X**2).mean(1, keepdims=True) + 1e-5) * w))

# 3. 矩阵乘法：程序 (pid_m, pid_n) 负责 C 的一个 BM×BN 小块，沿 K 方向一块一块地取 A、B 相乘累加
def matmul_kernel(pid_m, pid_n, A, B, C, BM, BN, BK):
    rm, rn = pid_m * BM + np.arange(BM), pid_n * BN + np.arange(BN)
    acc = np.zeros((BM, BN))                              # 累加器在寄存器里
    for k in range(0, A.shape[1], BK):
```

```
        rk = k + np.arange(BK)
        acc += A[np.ix_(rm, rk)] @ B[np.ix_(rk, rn)]     # tl.dot：交给 Tensor Core
    C[np.ix_(rm, rn)] = acc
A, B = np.random.default_rng(2).standard_normal((2, 64, 64)); C = np.zeros((64,
64))
grid = (64 // 32, 64 // 16)
for pm in range(grid[0]):
    for pn in range(grid[1]): matmul_kernel(pm, pn, A, B, C, 32, 16, 16)
print(f"64×64 矩阵乘法，BM = 32、BN = 16 → 网格 {grid}，共 {grid[0] * grid[1]} 个程序；
结果正确：{np.allclose(C, A @ B)}")
```

输出：

```
n = 1000，BLOCK = 256 → 4 个程序，各自处理的有效元素：[256, 256, 256, 232]
结果正确： True
RMSNorm 结果正确： True
64×64 矩阵乘法，BM = 32、BN = 16 → 网格 (2, 4)，共 8 个程序；结果正确：True
```

第一行：n = 1000、BLOCK = 256，4 个程序，前三个各处理 256 个元素，最后一个只有 232 个有效，和演示一致。第二、三行说明融合的 SwiGLU 和 RMSNorm 结果正确。最后一行：64 × 64 的矩阵乘法，每个程序负责 32 × 16 的一块，网格 (2, 4) 共 8 个程序，结果正确。

如果有一张 NVIDIA 或 AMD 的 GPU，装好 PyTorch 和 Triton，就可以直接运行前面那几段真正的 Triton 代码。

到了大模型里

PyTorch 编译器生成的就是 Triton。`torch.compile` 的后端 TorchInductor，会把计算图里能融合的运算融合起来，生成 Triton kernel（第 83 讲）。很多人每天都在用 Triton，只是没有亲手写。

推理引擎里的 Triton kernel。vLLM 里有不少 kernel 是用 Triton 写的，比如 MoE 的融合 kernel（第 41 讲的专家计算）、一些注意力的实现、各种量化相关的 kernel。SGLang 也一样。Triton 写起来快、改起来方便，新模型、新结构出来时，往往先用 Triton 实现。

不止 NVIDIA。Triton 也支持 AMD 的 GPU，同一份 kernel 代码可以在两家的硬件上编译运行，这是它很大的优势。

极限性能仍然要更底层的工具。对矩阵乘法、注意力这些最关键的 kernel，CUDA 配合 CUTLASS 等库能更精细地控制硬件的新特性，FlashAttention-3 就是这样写的。也有一些新的工具（比如 CUTLASS 的 Python 接口 CuTe DSL、TileLang）试图在 Triton 的易用和 CUDA 的可控之间找平衡。

第二段代码复现“候选怎么挑”和“动态形状”的数字：调用比例的临界值、三种策略的每组时间与交点，以及调优搜索的回本次数。

```
# 1. 新实现对形状 A 快一倍、对形状 B 慢一倍：A 的调用占比 p 超过多少才划算
old, new_a, new_b = 10, 5, 20                          # μs
p = (new_b - old) / (new_b - new_a)
print(f"新实现平均 = {new_a}p + {new_b}(1 - p)，比原来的 {old} μs 快的条件：p >
{p:.3f}")

# 2. 动态形状：通用 / 分桶 / 逐形状特化。10 次 FFN 为一组：8 次 256 行、1 次 1536 行、1 次
2048 行
H, F, peak = 4096, 12288, 503.8e12                     # RTX PRO 6000 BF16 稠密峰值
real = 8 * 256 + 1536 + 2048
padded = 8 * 512 + 2048 + 2048                         # 两个桶：512、2048
plans = {"通用": (0.100, real, 0.20), "分桶": (0.400, padded, 0.40), "特化": (0.900,
real, 0.50)}
T = {k: 6 * H * F * rows / (peak * mfu) for k, (prep, rows, mfu) in plans.items()}
print(f"实际 {real} 行，分桶后执行 {padded} 行（多算 {padded/real-1:.0%}）")
for k, (prep, rows, mfu) in plans.items():
    print(f"{k}：准备 {prep*1e3:.0f} ms，每组 {T[k]*1e3:.1f} ms")
best = lambda r: min(plans, key=lambda k: plans[k][0] + r * T[k])
switch = [r for r in range(1, 200) if best(r) != best(r - 1)]
print("最省时的方案：", "，".join(f"第 {r} 组起 {best(r)}" for r in [1] + switch))
print("跑 70 组：", "、".join(f"{k} {plans[k][0] + 70*T[k]:.2f} s" for k in plans))

# 3. 自动调优搜索 10 分钟，每次调用省 5 μs，要调用多少次才收回
calls = 600 / 5e-6
print(f"要 {calls:.1e} 次调用；每秒 1 万次约 {calls/1e4/3600:.1f} 小时，每秒 100 次约
{calls/100/86400:.0f} 天")
```

输出：

```
新实现平均 = 5p + 20(1 - p)，比原来的 10 μs 快的条件：p > 0.667
实际 5632 行，分桶后执行 8192 行（多算 45%）
通用：准备 100 ms，每组 16.9 ms
分桶：准备 400 ms，每组 12.3 ms
特化：准备 900 ms，每组 6.8 ms
最省时的方案： 第 1 组起 通用，第 66 组起 分桶，第 91 组起 特化
跑 70 组： 通用 1.28 s、分桶 1.26 s、特化 1.37 s
要 1.2e+08 次调用；每秒 1 万次约 3.3 小时，每秒 100 次约 14 天
```

对照正文：p > 2/3；分桶多算 45%；每组 16.9、12.3、6.8 ms；第 66 组起分桶、第 91 组起特化；70 组时分桶最快；搜索要 1.2 亿次调用才回本。

容易混淆的地方

1. **Triton 的“程序”不是线程**。一个程序处理一整块数据，背后是一个线程块里的几十到几百个线程。`num_warps` 决定这个块用几个 warp。

2. **它和 NVIDIA 的 Triton 推理服务器不是一个东西**。NVIDIA 有一个推理服务框架也叫 Triton（Triton Inference Server），它是部署模型的服务，和这里写 kernel 的 Triton 语言毫无关系，只是重名。

3. **块的形状是编译时常数**。`tl.arange` 的长度、`tl.dot` 的块大小都必须在编译时确定，而且一般是 2 的幂。行长 d 不是 2 的幂时，就取不小于它的 2 的幂，再用 mask 盖住多出来的部分。

4. **单个形状上的加速不等于整体加速**。一个实现在某个形状上快、在另一个形状上慢，要按实际的调用比例算平均时间。

5. **最快的程序不一定最省时间**。特化程序每次最快，但编译和调优的准备时间要靠足够多的调用来摊。

回顾

CUDA 以线程为单位写 kernel，要自己管下标、共享内存、同步和访存模式，写快的 kernel 很繁琐。Triton 把视角提到“一块数据”：program_id 决定负责哪一块，arange 生成整块下标，mask 处理越界，load、整块计算、store；线程怎么分工、共享内存、Tensor Core 都交给编译器。融合的 SwiGLU、按行处理的 RMSNorm、分块的矩阵乘法，都只要十几行。torch.compile 生成的就是 Triton，推理引擎里也有大量 Triton kernel。

挑选实现时，先按资源排除放不下的候选，再估算，最后在真实的调用比例下实测：新实现对形状 A 快一倍、对 B 慢一倍时，A 的占比要超过 2/3 才划算。动态形状有通用、分桶、特化三种做法，准备越久的方案每次越快，本例前 65 组通用最省，第 66 组起分桶，第 91 组起特化；程序已经缓存时特化从一开始就最快。

kernel 写得再快，每启动一次，CPU 都要花几微秒准备和发射。decode 一步只有几毫秒，却要启动几百个小 kernel，启动开销会占多少？怎么消除它？下一讲：**CUDA Graph 与调度开销**。

第89讲 CUDA Graph

选读

68 Prefill 与 Decode　81 请求生命周期　83 算子融合　86 CUDA 编程模型

到目前为止，我们估算 decode 一步的时间，只看 GPU：读一遍权重、读一遍 KV Cache，除以带宽（第 68 讲）。可 GPU 上的每个 kernel，都是 CPU 一个一个发射过去的。第 83 讲数过，一层 Transformer 里有十几个 kernel，32 层加起来就是好几百个。如果 CPU 发射一个 kernel 要几微秒，几百个就是好几毫秒，和 GPU 真正计算的时间差不多了。

这一讲讲这种“CPU 跟不上 GPU”的情况什么时候会发生，以及推理引擎怎么用 CUDA Graph 消除它。

CPU 发射，GPU 执行

在 PyTorch 里写 `y = x @ w`，背后发生的是：Python 解释器执行这一行，PyTorch 检查张量的形状、类型、设备，选出合适的 kernel，准备好参数，再通过 CUDA 驱动把这个 kernel 放进 GPU 的任务队列。这一整套在 CPU 上要花几微秒到十几微秒。这种逐个算子执行的方式，叫 **eager 模式**。

好在发射是**异步**的：CPU 把 kernel 放进队列就返回，不等 GPU 算完，接着发射下一个。GPU 从队列里一个接一个地取出来执行。只要 CPU 发射得比 GPU 执行得快，CPU 就会一路跑在前面，队列里总有活，GPU 不会停下来，发射开销被完全藏住了。

反过来，如果 GPU 执行一个 kernel 的时间比 CPU 发射一个还短，GPU 算完这一个，下一个还没发过来，只能空等。这时一步的时间就由 CPU 决定了：**一步 ≈ kernel 数 × 每次发射的 CPU 时间**，GPU 算得再快也没用。

什么时候 CPU 会跟不上

代码里用一个粗略的模型估算了几种情况（批大小 1 的 decode；每层 4 个矩阵乘法按读权重的时间算，另有 11 个约 2 μs 的小 kernel；假设 eager 模式下 CPU 发射一个 kernel 约 8 μs，这些数字只代表量级）：

- **Llama-3-8B，1 张卡**：481 个 kernel，GPU 计算约 5.18 ms；CPU 发射全部要 481 × 8 μs ≈ 3.85 ms，比 GPU 快，发射被藏住了，eager 一步约 5.19 ms，GPU 几乎一直在忙。

- **Llama-3-8B，8 张卡张量并行**（第 13 篇）：权重分到 8 张卡上，每张卡的矩阵乘法只剩八分之一，每层又多了 2 次卡间通信，一共 545 个 kernel，GPU 只需约 **1.93 ms**；可 CPU 发射要 545 × 8 μs ≈ 4.36 ms，eager 一步约 **4.40 ms**，GPU 有一半多的时间在空等。
- **Llama-3.2-1B，1 张卡**：小模型，241 个 kernel，GPU 约 1.09 ms，eager 一步约 2.08 ms，GPU 只忙了一半。

规律是：**模型越小、并行的卡越多、批越小，每个 kernel 越短，CPU 越容易成为瓶颈**。而这些正是延迟敏感的在线服务常见的情况。GPU 越来越快，CPU 却没有快那么多，这个问题只会越来越明显。

CUDA Graph：录一次，放很多次

decode 的每一步，执行的 kernel 序列几乎一模一样：同样的几百个 kernel，同样的顺序，只是输入的数据不同。既然如此，何必每一步都让 CPU 重新走一遍 Python、PyTorch、驱动的流程?

CUDA Graph 就是为此设计的。它分两个阶段：

1. **捕获**（capture）：让 CPU 照常执行一遍这一步，但 kernel 不真正执行，而是被记录下来：哪些 kernel、什么顺序、什么参数、读写哪些显存地址，组成一张“图”。
2. **回放**（replay）：之后每一步，CPU 只需要一次调用，把整张图交给 GPU。GPU 按图一个接一个地执行这几百个 kernel，中间没有 CPU 参与。

回放一张图，CPU 只花几十微秒。上面的例子里，8 卡张量并行时一步从约 4.40 ms 降到约 **1.95 ms**，1B 模型从约 2.08 ms 降到约 1.11 ms，GPU 几乎一直在忙。

下面的时间线把 CPU 和 GPU 画成两条线。eager 模式下，CPU 那条线上每一小段是发射一个 kernel，GPU 那条线上每一小段是执行一个 kernel；GPU 线上的空白就是在等 CPU。换一换模型，调一调 CPU 发射的时间看看。

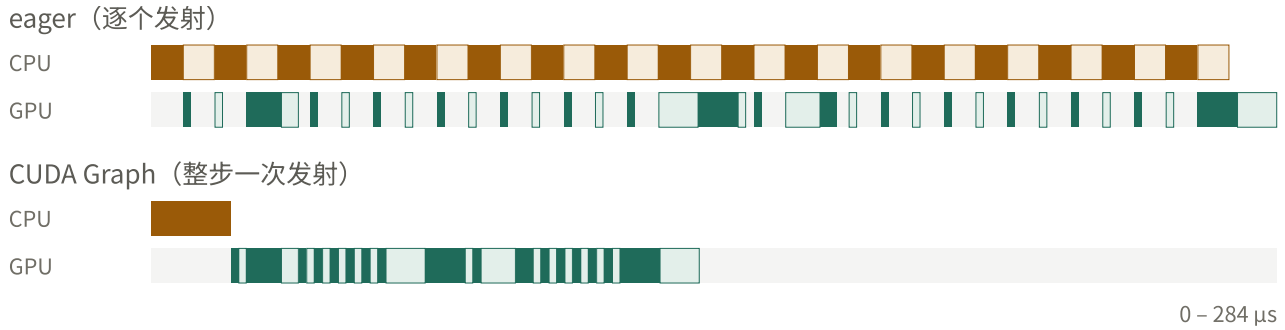
CPU 发射，GPU 执行：谁在等谁

⇆ 对照

Llama-3-8B，1 张卡 | Llama-3-8B，8 卡张量并行 | Llama-3.2-1B，1 张卡

看前 2 层 | 看整步

CPU 发射一个 kernel = **8** µs



一步 545 个 kernel，GPU 真正在算 1.93 ms；CPU 逐个发射要 545 × 8 µs = 4.36 ms。
eager 一步 4.40 ms（GPU 忙 44%）；CUDA Graph 一步 1.95 ms（GPU 忙 99%）。
CPU 发射得比 GPU 执行得慢：GPU 那一行出现很多空隙，一步的时间由 CPU 决定。

图 89-1　CPU 发射与 GPU 执行。默认是 Llama-3-8B 8 卡张量并行的前 2 层：eager 模式下 GPU 每算完一个短 kernel 就要等 CPU 发下一个；CUDA Graph 一次发射整步，GPU 的 kernel 首尾相接。

要点

eager 模式下，CPU 逐个发射 kernel，每个要几微秒。GPU 的 kernel 很短时（小模型、多卡并行、小批量的 decode），CPU 发射不过来，GPU 就在 kernel 之间空等。CUDA Graph 把一整步的 kernel 序列录成一张图，之后每步只发射一次，GPU 连续执行，启动开销几乎消失。

代价：形状和地址都要固定

CUDA Graph 录下的是具体的 kernel 和具体的显存地址。回放时不能改变这些东西，这带来几个限制：

输入要放在固定的地址。每一步的输入（token、位置、块表）不能新建张量，而要先拷贝到捕获时用的那块“静态”缓冲区里，再回放。输出也写在固定的位置。

形状要固定。一张图对应一个固定的批大小。可连续批处理里，每一步的批大小都在变（第 74 讲）。vLLM 的做法是预先为一组批大小（比如 1、2、4、8、16、24、32……直到几百）各捕获一张图；实际的批大小是 13 时，就补齐到 16，用 16 的那张图，多出来的 3 个位置算了也不用。

不能有 CPU 端的判断。图里不能有“根据中间结果，CPU 决定下一步调哪个 kernel”的逻辑。所以一步里依赖动态信息的部分要放在图外面。

占显存，也要时间。每张图都要预留它用到的中间缓冲区，捕获几十张图，启动时要多花几十秒，也要占一些显存。vLLM 启动日志里的 “Capturing CUDA graphs” 就是在做这件事；调试时可以用 `enforce_eager` 关掉它，看看差别。

prefill 通常不用。prefill 的 token 数每次都不同，变化范围又大，不适合按形状预先捕获。好在 prefill 的 kernel 都很大，GPU 时间远大于发射时间，本来也不需要。vLLM 的新版本还把模型切成几段，注意力（形状最不固定）留在图外，其余部分用图，这叫分段的 CUDA Graph。

固定地址的代价：输入复制

上面说输入要先拷到捕获时的静态缓冲。这次拷贝要多久，会不会把省下的时间吃掉？算一个例子：普通执行时 GPU 计算 20 µs，再加上等主机逐个提交的 20 µs，一共 40 µs；改成图重放，重放加准备元数据只要 5 µs，原本能省 15 µs。

可新输入要先拷到图记录的地址。拷贝读一遍源、写一遍目标，按 RTX PRO 6000 的显存带宽 1792 GB/s：

- 输入 [256, 4096] 的 BF16 是 2 MiB，拷贝约 **2.3 µs**，图执行共 27.3 µs，比普通执行少 12.7 µs。
- 输入 [2048, 4096] 是 16 MiB，拷贝约 **18.7 µs**，图执行共 43.7 µs，**比普通执行还慢**。

省下的主机时间是固定的 15 µs，拷贝时间却随输入变大。两者相等时 $2X \div B = 15$ µs，临界输入 $X \approx 12.8$ MiB。所以选哪些操作放进一张图，要同时比较省下的提交时间和多出来的拷贝时间。更好的做法是让上游算子直接写进图的固定缓冲，这次拷贝就没了。



图 89-2　图重放省的是主机提交时间（固定），多出来的是输入复制（随输入变大）。默认 2 MiB 输入净省 12.7 µs；调到 2048 行（16 MiB），复制 18.7 µs，图重放反而更慢。临界输入约 12.8 MiB。

GPU 越快，越要看主机

第 66 讲的时间线里，GPU 等主机的时间一直存在。把它写成流水来算：一共 100 段工作，每段主机准备 20 µs、GPU 计算 20 µs。严格串行要 4000 µs；主机准备下一段时 GPU 算当前段，变成流水，只要 2020 µs。

现在把 GPU 计算缩短到 5 µs。主机仍然每 20 µs 才准备好一段，总时间是 2005 µs，只少了 15 µs：GPU 越快，等得越久。只有把主机准备也压到 5 µs，才降到 505 µs。流水的节拍由较慢的一侧决定，这和第 86 讲的双缓冲是同一个规律。

vLLM 在 2024 年 9 月的 v0.6.0 版本就遇到了这个问题。发布前的剖析显示：单张 H100 跑 Llama-3-8B 时，一步里 API 服务器占 33%，调度占 29%，GPU 执行只占 38%。原因是 API 服务器和推理引擎在同一个 Python 进程里，两者争用 GIL（Python 的全局解释器锁）。v0.6.0 做了三件事：把 API 服务器拆到单独的进程、用消息队列通信；多步调度，一次调度连续跑几步；输出处理改成异步，和下一步的执行重叠。8B 的吞吐提高到原来的 2.7 倍，70B 提高到 1.8 倍。这些改动没有碰任何 kernel，也没有换硬件。

能复用的不只是提交序列，还有提交之前的准备。推理算子库 FlashInfer 把注意力分成 plan（按本步每个请求的长度和 KV 的位置选 kernel、划分任务、安排工作区，并把这些元数据拷到 GPU）和 run（按计划执行）两步。同一步里 36 层的请求长度和 KV 布局都一样，可以共用一份 plan。在 RTX PRO 6000 上的对照中，从每层都 plan 改成只 plan 一次，36 层注意力的时间从约 2.0 ms 降到 0.72 ms，主机到 GPU 的拷贝从 144 次降到 4 次，GPU 上的 kernel 数没变。反过来，请求长度变了就必须重新 plan，沿用旧计划会算错。

怎么计时才对

最后补一个容易出错的地方：同一次调用，在不同的位置计时，得到的是三种不同的时间。设 CPU 用 3 μs 提交全部任务，GPU 依次做 H2D 8 μs、kernel 20 μs、D2H 4 μs：

- 在异步调用的前后读 CPU 时钟，测到的是**提交时间**：3 μs。
- 用 CUDA event 在 GPU 的任务流里打点，测到的是**GPU 执行时间**：kernel 20 μs。
- 等结果回到 CPU 才停表，测到的是**结果可用的时间**：第 35 μs。

把 kernel 优化到 5 μs，快了 4 倍，结果却是在第 20 μs 可用，整次调用只快了 1.75 倍，剩下的 15 μs 是提交和拷贝。所以报告加速比时，要说清楚是从哪里计到哪里。用 profiler 把主机调用、GPU 计算和拷贝画在同一条时间线上，才能看清时间花在哪一段。

其他省 CPU 时间的办法

CUDA Graph 解决的是“发射 kernel”的开销。一步里 CPU 上还有别的事：调度器决定这一步跑哪些请求、准备输入张量、采样后处理（第 81 讲）。推理引擎也在想办法压缩或藏住它们：

- **算子融合**（第 83 讲）直接减少了 kernel 的数量。
- **torch.compile** 把模型编译成更少、更大的 kernel，还可以和 CUDA Graph 一起使用。
- **CPU 和 GPU 重叠**：GPU 在执行第 n 步时，CPU 就开始为第 n + 1 步做调度和输入准备，而不是等第 n 步结束。vLLM 和 SGLang 都有这类“异步调度”或“重叠调度”的机制。
- **把 Python 挪出热路径**：调度逻辑、输入准备尽量用向量化的张量操作，甚至用 C++ 实现。

用代码复现

下面的代码按上面的粗略模型，模拟 CPU 逐个发射、GPU 按顺序执行的过程：GPU 开始一个 kernel 的时刻，是“它被发射完”和“GPU 做完上一个”中较晚的那个。分别算 eager 和 CUDA Graph 下一步的时间。

```
# decode 一步（批大小 1）里的 kernel：CPU 逐个发射，GPU 按顺序执行。粗略模型，数字是假设的量级
BW = 3.35e12                                                # H100 显存带宽
SMALL, LAUNCH, GRAPH = 2e-6, 8e-6, 20e-6                    # 小 kernel 在 GPU 上约 2 μs；eager 模式 CPU 发射一个 kernel 约 8 μs；回放一张图约 20 μs

def kernels(d, dff, L, kv_dim, vocab, tp=1):
    """每层：4 个矩阵乘法（读权重，时间 = 字节 ÷ 带宽），11 个小 kernel（归一化、RoPE、注意力、激活、相加……）；多卡时每层多 2 次 all-reduce。"""
    ks = []
    for _ in range(L):
        for rows, cols in [(d + 2 * kv_dim, d), (d, d), (2 * dff, d), (d, dff)]:
            ks.append(max(SMALL, rows * cols * 2 / tp / BW))
        ks += [SMALL] * 11 + ([10e-6] * 2 if tp > 1 else [])
    return ks + [vocab * d * 2 / tp / BW]                    # 最后的输出层

def step(ks, graph):
    cpu, gpu = (GRAPH, 0.0) if graph else (0.0, 0.0)
    for g in ks:
        if not graph: cpu += LAUNCH                          # eager：CPU 发射完这个 kernel，GPU 才能开始它
        gpu = max(gpu, cpu) + g
    return gpu

print("模型 | kernel 数 | GPU 真正在算的时间 | eager 一步 | CUDA Graph 一步")
for name, cfg in [("Llama-3-8B，1 张卡", (4096, 14336, 32, 1024, 128256, 1)),
                  ("Llama-3-8B，8 张卡张量并行", (4096, 14336, 32, 1024, 128256, 8)),
                  ("Llama-3.2-1B，1 张卡", (2048, 8192, 16, 512, 128256, 1))]:
    ks = kernels(*cfg)
    busy, e, g = sum(ks), step(ks, False), step(ks, True)
    print(f"{name} | {len(ks)} | {busy * 1e3:.2f} ms | {e * 1e3:.2f} ms (GPU 忙 {busy / e:.0%}) | {g * 1e3:.2f} ms (GPU 忙 {busy / g:.0%}) ")
```

输出：

```
模型 | kernel 数 | GPU 真正在算的时间 | eager 一步 | CUDA Graph 一步
Llama-3-8B，1 张卡 | 481 | 5.18 ms | 5.19 ms（GPU 忙 100%） | 5.20 ms（GPU 忙 100%）
Llama-3-8B，8 张卡张量并行 | 545 | 1.93 ms | 4.40 ms（GPU 忙 44%） | 1.95 ms（GPU 忙
99%）
Llama-3.2-1B，1 张卡 | 241 | 1.09 ms | 2.08 ms（GPU 忙 52%） | 1.11 ms（GPU 忙 98%）
```

三行分别对应上面的三种情况。Llama-3-8B 单卡时，eager 和 CUDA Graph 几乎一样（5.19 ms 对 5.20 ms），因为 CPU 发射本来就跑在前面；8 卡并行和 1B 模型时，eager 的 GPU 只忙了 44% 和 52%，用 CUDA Graph 后接近 100%。

这个模型里每个数字都是假设的量级：实际的发射开销取决于 CPU、框架和驱动，从两三微秒到十几微秒不等；小 kernel 的执行时间也取决于具体的实现。真实的情况要用 Profiler 看时间线（第 66 讲）。

第二段代码复现这一讲后半部分的数字：主机准备与 GPU 计算的流水、三种计时位置，以及图重放的输入复制与临界大小。

```python
# 1. 主机准备和 GPU 计算流水：100 段，谁慢谁决定节拍
def pipe(host, gpu, n=100): return host + (n - 1) * max(host, gpu) + gpu
print(f"串行 {100*(20+20)} μs；流水 {pipe(20, 20)} μs；GPU 提速到 5 μs：{pipe(20, 5)}
μs；主机也到 5 μs：{pipe(5, 5)} μs")

# 2. 一次调用的三种计时：提交 3 μs，H2D 8 μs，kernel 20 μs，D2H 4 μs
for k in [20, 5]:
    print(f"kernel {k:>2} μs：CPU 提交用 3 μs，GPU 计时 {k} μs，结果在第 {3 + 8 + k +
4} μs 可用")
print(f"kernel 快 4 倍，整次调用只快 {(3+8+20+4)/(3+8+5+4):.2f} 倍")

# 3. CUDA Graph 要先把新输入复制到图记录的地址：复制要读一遍、写一遍
BW, saved = 1792e9, 15e-6                          # RTX PRO 6000 显存带宽；图重放省下的
主机准备时间
for rows in [256, 2048]:
    x = rows * 4096 * 2
    copy = 2 * x / BW
    print(f"输入 [{rows}, 4096] ({x/2**20:.0f} MiB) ：复制 {copy*1e6:4.1f} μs，图执行
共 {(5e-6 + copy + 20e-6)*1e6:.1f} μs（普通执行 40 μs）")
print(f"输入小于 {saved * BW / 2 / 2**20:.1f} MiB 时，图重放才划算")
```

输出：

```
串行 4000 µs；流水 2020 µs；GPU 提速到 5 µs：2005 µs；主机也到 5 µs：505 µs
kernel 20 µs：CPU 提交用 3 µs，GPU 计时 20 µs，结果在第 35 µs 可用
kernel  5 µs：CPU 提交用 3 µs，GPU 计时 5 µs，结果在第 20 µs 可用
kernel 快 4 倍，整次调用只快 1.75 倍
输入 [256, 4096]（2 MiB）：复制  2.3 µs，图执行共 27.3 µs（普通执行 40 µs）
输入 [2048, 4096]（16 MiB）：复制 18.7 µs，图执行共 43.7 µs（普通执行 40 µs）
输入小于 12.8 MiB 时，图重放才划算
```

对照正文：流水 2020 µs，GPU 提速后 2005 µs，主机也提速后 505 µs；kernel 快 4 倍、整次调用只快 1.75 倍；2 MiB 输入图执行 27.3 µs，16 MiB 输入 43.7 µs，临界约 12.8 MiB。

容易混淆的地方

1. **CUDA Graph 不让 kernel 本身变快**。它省的是 kernel 之间的空隙。如果 GPU 本来就没有空等（大模型、大批量、prefill），用不用 CUDA Graph 差别不大。
2. **“GPU 利用率 100%”不代表 GPU 在全力计算**。`nvidia-smi` 显示的利用率，只表示这段时间里有 kernel 在运行，哪怕是一个只用了几个 SM 的小 kernel。看 GPU 有没有空等，要看 Profiler 的时间线。
3. **异步发射意味着 CPU 上测的时间不准**。CPU 发射完就返回，在 Python 里用计时器量 `y = x @ w`，量到的只是发射时间。要测 GPU 上的真实时间，需要先同步（`torch.cuda.synchronize()`），或者用 CUDA 事件计时。
4. **图重放不总是更快**。新输入要先拷到图的固定地址，输入大到一定程度（本例约 12.8 MiB），拷贝时间就超过了省下的提交时间。
5. **kernel 的加速比不等于调用的加速比**。提交、拷贝和同步不随 kernel 变快；计时的起止位置不同，测到的是不同的东西。

回顾

GPU 上的每个 kernel 都由 CPU 发射。eager 模式下，每发射一个要经过 Python、框架和驱动，花几微秒；发射是异步的，只要 CPU 跑在 GPU 前面，这些开销就被藏住了。可是模型小、多卡并行、批量小时，每个 kernel 只要一两微秒，CPU 发射不过来，GPU 就在 kernel 之间空等：Llama-3-8B 8 卡张量并行时，GPU 只需 1.93 ms 的一步，eager 要 4.40 ms。CUDA

Graph 把一整步的 kernel 序列捕获成一张图，之后每步只发射一次，一步降到约 1.95 ms。代价是输入地址和形状要固定，所以要按一组批大小分别捕获、实际批大小向上补齐，prefill 一般不用。

主机和 GPU 是一条流水，节拍由较慢的一侧决定：GPU 从 20 µs 提速到 5 µs，主机仍要 20 µs，100 段工作只从 2020 µs 降到 2005 µs。vLLM v0.6.0 把 API 服务器拆出进程、多步调度、异步输出处理，没改任何 kernel，吞吐就提高到 2.7 倍。图重放的输入要拷到固定地址，输入超过约 12.8 MiB 时拷贝就吃掉了全部收益。

第 11 篇到这里结束了。我们看到，推理快不快，最后往往落在“读写了多少字节”上：权重、KV Cache、中间结果。既然读写的字节决定了速度，能不能让每个数本身占的字节少一点？把 16 位的权重换成 8 位、4 位，速度和显存会怎样，精度又会损失多少？下一讲进入第 12 篇：**量化基础**。

AI INFRA · HANDBOOK

第 十二 篇

模型压缩与加速

收录 9 讲 · 第 90—98 讲

第90讲 量化基础

63 低精度计算　68 Prefill 与 Decode　71 显存账本

第 11 篇的结论可以归成一句话：推理快不快，往往取决于从显存里读了多少字节。decode 的每一步要把 16 GB 的权重读一遍（第 68 讲），一张 80 GB 的卡装下权重之后，剩下的才能放 KV Cache（第 71 讲）。

既然字节数这么关键，一个很自然的想法是：**让每个数少占几个比特**。权重现在用 BF16 存，每个数 16 比特；如果每个数只用 8 比特、甚至 4 比特，权重就小了一半、四分之三，读得也快了一倍、几倍。这就是**量化**（quantization）。第 12 篇讲模型压缩，这一讲先从最基本的问题开始：怎么用 4 个比特表示 2.31 这样一个数?

把实数放到格点上

4 个比特只能表示 16 个不同的值。有符号的 4 位整数（INT4）能表示 −8 到 7；为了正负对称，通常只用 −7 到 7 这 15 个整数。要用它们表示一组实数，就得先把实数按比例“缩放”到这个范围里。

做法分三步：

1. **算 scale**：找出这组数里绝对值最大的那个，让它对应到最大的整数 7。scale = max|x| ÷ 7。
2. **量化**：每个数除以 scale，四舍五入成整数，q = round(x ÷ scale)。存下来的就是这些整数 q，加上一个 scale。
3. **反量化**：要用的时候，乘回去，$\hat{x} = q \times \text{scale}$。

$\hat{x}$ 和原来的 x 不完全一样，差别就是**量化误差**。因为取整最多差 0.5，所以误差不超过 scale ÷ 2。

一般地，b 比特的对称量化，最大的整数是 2^(b−1) − 1：8 比特是 127，4 比特是 7。

算一个例子

取 8 个权重：0.82, −1.47, 0.05, 2.31, −0.66, 1.12, −0.29, 0.94。

INT8：最大的绝对值是 2.31，scale = 2.31 ÷ 127 = 0.01819。0.82 ÷ 0.01819 = 45.08，取整得 45；反量化 45 × 0.01819 = 0.8185，误差不到 0.002。8 个数的量化结果是 [45, −81, 3, 127, −36, 62, −16, 52]，最大误差 0.0077，确实不超过 scale ÷ 2 = 0.0091。

INT4：scale = 2.31 ÷ 7 = 0.33，q = [2, −4, 0, 7, −2, 3, −1, 3]，平均误差 0.0725。格点稀了 18 倍，误差也大了 20 倍。

位宽再往下，误差急剧变大：INT3 平均误差 0.1362，INT2 只剩 −1、0、1 三个值，8 个数里有 5 个变成了 0，平均误差 0.59，基本不能用了。

非对称量化。如果这组数不是围绕 0 对称的（比如都是正数），对称量化会浪费一半的格点。非对称量化把 [最小值, 最大值] 映射到 [0, 2^b − 1]，多存一个**零点** z，表示实数 0 对应哪个整数：scale = (max − min) ÷ (2^b − 1)，q = round(x ÷ scale) + z，反量化 $\hat{x}$ = (q − z) × scale。上面 8 个数用 INT4 非对称量化，scale = 0.252，z = 6，平均误差 0.064，比对称的 0.0725 小一点，因为 16 个格点都用上了。

下面的演示可以一步步看量化的过程。改一改位宽、改一改数字，看格点和误差怎么变。

量化：除以 scale，取整，再乘回去

⇆ 对照

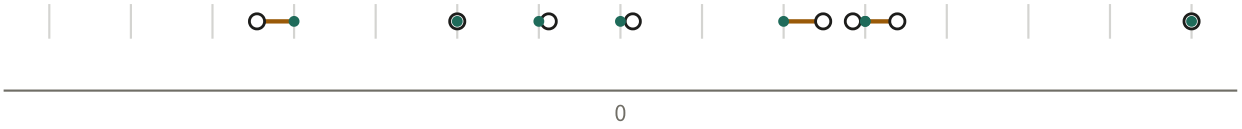
8 个数（可以改，例如把 2.31 改成 12 试试离群值）

0.82, -1.47, 0.05, 2.31, -0.66, 1.12, -0.29, 0.94

位宽 = **4** bit

对称 | 非对称（带零点）

8 个共用一个 scale | 每 4 个一组



x	0.82	-1.47	0.05	2.31	-0.66	1.12	-0.29	0.94
x / scale	2.48	-4.45	0.15	7.00	-2.00	3.39	-0.88	2.85
q（取整）	**2**	**-4**	**0**	**7**	**-2**	**3**	**-1**	**3**
x̂ = q · scale	0.660	-1.320	0.000	2.310	-0.660	0.990	-0.330	0.990
误差	0.160	-0.150	0.050	0.000	0.000	0.130	0.040	-0.050

scale = 0.3300，q 的范围 [-7, 7]（4 bit，16 个格点）
平均误差 0.0725；每个数存 4 bit，另外每组存一个 16 bit 的 scale。
数轴上竖线是能表示的值（格点），空心圆是原来的数，实心点是反量化后的值。最大的那个数决定了 scale：它越大，格点越稀，其他数的误差越大。

图 90-1　量化与反量化。数轴上的竖线是这个位宽能表示的值，空心圆是原来的数，实心点是反量化后的值，它们之间的连线就是误差。把 2.31 改成 12，看一个离群值怎么让所有数的误差变大；再切换到“每 4 个一组”。

一个离群值毁掉一组数

scale 由最大的那个数决定，这是量化最大的弱点。把上面的 2.31 换成 12.0（一个**离群值**，outlier），INT4 的 scale 变成 12 ÷ 7 = 1.714，格点间隔 1.7，0.82、−0.66、0.05、−0.29 都被量化成了 0。除了 12 本身，其余 7 个数的平均误差从不到 0.1 变成了 **0.49**。

大模型的权重和激活里恰恰有不少离群值（第 91、92 讲细说）。应对的第一个办法是**缩小共用一个 scale 的范围**：

- **逐张量**（per-tensor）：整个权重矩阵共用一个 scale。最省，但最怕离群值。

- **逐通道**（per-channel）：权重矩阵的每一行（每个输出通道）一个 scale。
- **逐组**（per-group）：每一行再切成若干组，每组比如 128 个数，一个 scale。

上面的例子分成两组、每 4 个一组：离群值所在的组 scale 是 1.714，另一组只有 0.16，另一组的数又能被精细地表示了。其余 7 个数的平均误差从 0.49 降到 **0.17**。

代价是要多存 scale。每 128 个 4 比特的权重配一个 16 比特的 scale，平均每个权重多 16 ÷ 128 = 0.125 比特，实际是 4.125 比特。组越小，精度越好，存 scale 的开销越大，128 是常用的折中。

> **要点**
> 量化用 scale 把一组实数映射到少数几个整数格点上：q = round(x ÷ scale)，用时 $\hat{x}$ = q × scale，误差不超过 scale ÷ 2。scale 由组里最大的数决定，离群值会让格点变稀、其他数的误差变大，所以要让更少的数共用一个 scale（逐通道、逐组）。

量化换来了什么

对 Llama-3-8B（约 80.3 亿个参数）：

- **BF16**：16.06 GB，decode 一步读一遍约 4.79 ms；
- **INT8**：8.03 GB，约 2.40 ms；
- **INT4**（每组 128 个）：4.14 GB，约 1.24 ms。

INT4 让权重小了将近四分之三。省下的显存可以放更多的 KV Cache、服务更多的请求；读权重的时间也少了将近四分之三，小批量的 decode（卡在读权重上）能快好几倍。一个 70B 的模型，BF16 要 140 GB，两张卡才放得下；INT4 只要 35 GB 左右，一张卡就够了。

量化什么，算的时候用什么

量化有两种用法，加速的原理不同：

只量化权重（weight-only，比如 W4A16：权重 4 比特，激活 16 比特）。权重以 INT4 存在显存里，kernel 读进片上以后，先反量化成 BF16，再和 BF16 的激活做矩阵乘法。计算还是 BF16 的，但从显存读的字节少了四分之三。decode 是带宽受限的，所以收益明显；prefill 是计算受限的，几乎没有收益，反量化还多了一点计算。第 91 讲的 GPTQ、AWQ 就是这一类。

权重和激活都量化（比如 W8A8）。矩阵乘法直接用 INT8 或 FP8 的硬件单元来算，第 63 讲说过，它们的算力是 BF16 的两倍。这样 prefill 这种计算受限的阶段也能加速。难点是激活里的离群值，第 92 讲讲 SmoothQuant 怎么处理它。

另外，量化可以在训练完以后直接做，叫**训练后量化**（PTQ）：拿一小批校准数据，算出 scale，几分钟到几小时就能完成，这是推理里最常用的方式。也可以在训练时就模拟量化误差，让模型适应它，叫**量化感知训练**（QAT），效果更好，但要重新训练。

用代码复现

下面的代码实现了对称和非对称量化，用上面 8 个数演示不同位宽的误差、离群值的影响和分组的效果，最后算出 Llama-3-8B 在不同位宽下的大小和读一遍的时间。

```python
import numpy as np

x = np.array([0.82, -1.47, 0.05, 2.31, -0.66, 1.12, -0.29, 0.94])   # 8 个权重（FP32）

def quant_sym(x, bits):
    """对称量化：scale = max|x| / (2^(bits-1) - 1)，q = round(x / scale)，反量化 x̂ = q · scale"""
    qmax = 2 ** (bits - 1) - 1
    scale = np.abs(x).max() / qmax
    q = np.clip(np.round(x / scale), -qmax, qmax).astype(int)
    return q, scale, q * scale

def quant_asym(x, bits):
    """非对称量化：把 [min, max] 映射到 [0, 2^bits - 1]，多一个零点 z"""
    qmax = 2 ** bits - 1
    scale = (x.max() - x.min()) / qmax
    z = int(np.round(-x.min() / scale))
    q = np.clip(np.round(x / scale) + z, 0, qmax).astype(int)
    return q, scale, z, (q - z) * scale

q, s, xh = quant_sym(x, 8)
print(f"INT8 对称：scale = 2.31 / 127 = {s:.5f}")
print("  q  =", q.tolist())
print("  x̂ =", np.round(xh, 4).tolist())
print(f"  最大误差 {np.abs(x - xh).max():.4f}（不超过 scale / 2 = {s / 2:.4f}）")
for bits in [8, 4, 3, 2]:
    q, s, xh = quant_sym(x, bits)
    print(f"INT{bits} 对称：q = {q.tolist()}，平均误差 {np.abs(x - xh).mean():.4f}")
q, s, z, xh = quant_asym(x, 4)
print(f"INT4 非对称：scale = {s:.4f}，零点 z = {z}，q = {q.tolist()}，平均误差 {np.abs(x - xh).mean():.4f}")

# 一个离群值会拉大 scale，让其他数都变粗
y = x.copy(); y[3] = 12.0
q, s, yh = quant_sym(y, 4)
print(f"把 2.31 换成 12.0 后 INT4：scale = {s:.3f}，q = {q.tolist()}，其余 7 个数的平均误差 {np.abs(np.delete(y - yh, 3)).mean():.4f}")

# 分组：每 4 个数一组各算各的 scale
g = y.reshape(2, 4); sc = np.abs(g).max(1, keepdims=True) / 7
gh = np.clip(np.round(g / sc), -7, 7) * sc
print(f"每 4 个一组：两组的 scale = {np.round(sc.ravel(), 3).tolist()}，其余 7 个数的平均误差 {np.abs(np.delete((g - gh).ravel(), 3)).mean():.4f}")

# Llama-3-8B 的权重，不同位宽下的大小（每 128 个权重存一个 FP16 的 scale）和 decode 一步读
```

```
权重的时间
N = 8.03e9
for name, bits in [("BF16", 16), ("INT8", 8), ("INT4（每组 128 个）", 4 + 16 /
128)]:
    gb = N * bits / 8 / 1e9
    print(f"{name}: {gb:5.2f} GB，读一遍 ≈ {gb / 3.35:.2f} ms")
```

输出：

```
INT8 对称：scale = 2.31 / 127 = 0.01819
  q  = [45, -81, 3, 127, -36, 62, -16, 52]
  x̂ = [0.8185, -1.4733, 0.0546, 2.31, -0.6548, 1.1277, -0.291, 0.9458]
  最大误差 0.0077（不超过 scale / 2 = 0.0091）
INT8 对称：q = [45, -81, 3, 127, -36, 62, -16, 52]，平均误差 0.0036
INT4 对称：q = [2, -4, 0, 7, -2, 3, -1, 3]，平均误差 0.0725
INT3 对称：q = [1, -2, 0, 3, -1, 1, 0, 1]，平均误差 0.1362
INT2 对称：q = [0, -1, 0, 1, 0, 0, 0, 0]，平均误差 0.5900
INT4 非对称：scale = 0.2520，零点 z = 6，q = [9, 0, 6, 15, 3, 10, 5, 10]，平均误差
0.0640
把 2.31 换成 12.0 后 INT4：scale = 1.714，q = [0, -1, 0, 7, 0, 1, 0, 1]，其余 7 个数的
平均误差 0.4904
每 4 个一组：两组的 scale = [1.714, 0.16]，其余 7 个数的平均误差 0.1692
BF16：16.06 GB，读一遍 ≈ 4.79 ms
INT8：  8.03 GB，读一遍 ≈ 2.40 ms
INT4（每组 128 个）：  4.14 GB，读一遍 ≈ 1.24 ms
```

前四行是 INT8 的完整过程：scale、整数 q、反量化的值、最大误差。接着是 INT8 到 INT2 的对比和 INT4 非对称的结果。然后是离群值的两行：12.0 让其余 7 个数的平均误差变成 0.4904，分组后降到 0.1692。最后三行是 Llama-3-8B 的权重大小。

容易混淆的地方

1. **量化误差小不等于模型效果好。**每个数的误差只是起点，真正关心的是模型的输出变了多少、在评测上掉了几分。有的权重误差大一点无所谓，有的一点误差就会放大。第 91 讲正是从“哪些误差更重要”出发的。
2. **存成 INT4 不等于用 INT4 计算。**W4A16 存的是 INT4，算的是 BF16；它省的是读取的字节，不是计算。
3. **“4 比特”的实际大小比四分之一多一点。**要算上 scale（和零点）的开销，以及通常不量化的部分（比如嵌入层、输出层、归一化的权重）。

回顾

量化用更少的比特表示一个数：选一个 scale 把一组实数缩放到整数范围里，取整存下来，用的时候乘回去。对称量化 scale = max|x| ÷ (2^(b−1) − 1)，误差不超过 scale ÷ 2；非对称量化多一个零点，能用满所有的格点。8 个示例权重用 INT8 量化，最大误差 0.0077；INT4 平均误差 0.0725；INT2 基本不能用。离群值会拉大 scale、毁掉同组的其他数，所以要逐通道、逐组量化，每组 128 个时实际约 4.125 比特。Llama-3-8B 的权重从 BF16 的 16.06 GB 降到 INT4 的 4.14 GB，decode 读一遍从 4.79 ms 降到 1.24 ms。

逐组量化能缓解离群值，但 4 比特的格点终究很粗，直接四舍五入（RTN）在一些模型上会掉不少精度。能不能在取整的时候“聪明”一点，让误差落在不那么重要的地方？下一讲：**权重量化：GPTQ 与 AWQ**。

第91讲 权重量化：GPTQ 与 AWQ

13 神经元与线性层　90 量化基础

上一讲的量化方法，说到底就是“除以 scale、四舍五入”，叫 RTN（round-to-nearest）。8 比特时它几乎不损失精度；可降到 4 比特、3 比特，很多模型的效果会明显下降。

这一讲的两种方法，GPTQ 和 AWQ，都还是把权重存成 4 比特整数加 scale，格式和 RTN 一样，区别只在于**怎么决定每个权重取整成哪个整数**。它们的出发点是同一个观察：我们真正在乎的不是权重本身变了多少，而是**这一层的输出**变了多少。

该盯住的是输出

一个线性层 $y = x \cdot W$（第 13 讲）。量化以后权重变成 $\hat{W}$，输出变成 $x \cdot \hat{W}$。输出的误差是

$$x \cdot W - x \cdot \hat{W} = \Sigma_j \, x_j \cdot (W_j - \hat{W}_j) \quad （W_j \text{ 是 } W \text{ 的第 } j \text{ 行，对应第 } j \text{ 个输入通道}）$$

第 j 行权重的误差，被第 j 个输入通道的激活 x_j 乘了一下，再加到输出上。如果某个输入通道的激活特别大，这一行权重的一点点误差，在输出里就会被放大很多倍。

大模型恰恰有这样的通道：在 Transformer 的激活里，少数几个通道的数值长期比其他通道大几十倍（第 92 讲还会细看）。RTN 对每一行一视同仁，这几行的误差和别的行一样大，却被放大得最多，成了输出误差的主要来源。

所以，**量化需要知道激活长什么样**。GPTQ 和 AWQ 都要先拿一小批**校准数据**（几百条普通文本）跑一遍模型，记录每一层的输入 X，再根据 X 来量化这一层的权重。

我们用一个人造的层来比较：64 个输入通道、64 个输出，256 个 token 的校准数据，其中 4 个输入通道是离群通道，激活大 20 倍。4 比特、每个输出通道一个 scale 的 RTN，输出的相对误差是 10.56%。

下面的演示是一个更小的层（16 个通道，第 5 号是离群通道）。上面一排是每个输入通道的平均激活，下面一排是每一行权重的量化误差乘以激活，也就是它对输出误差的贡献。

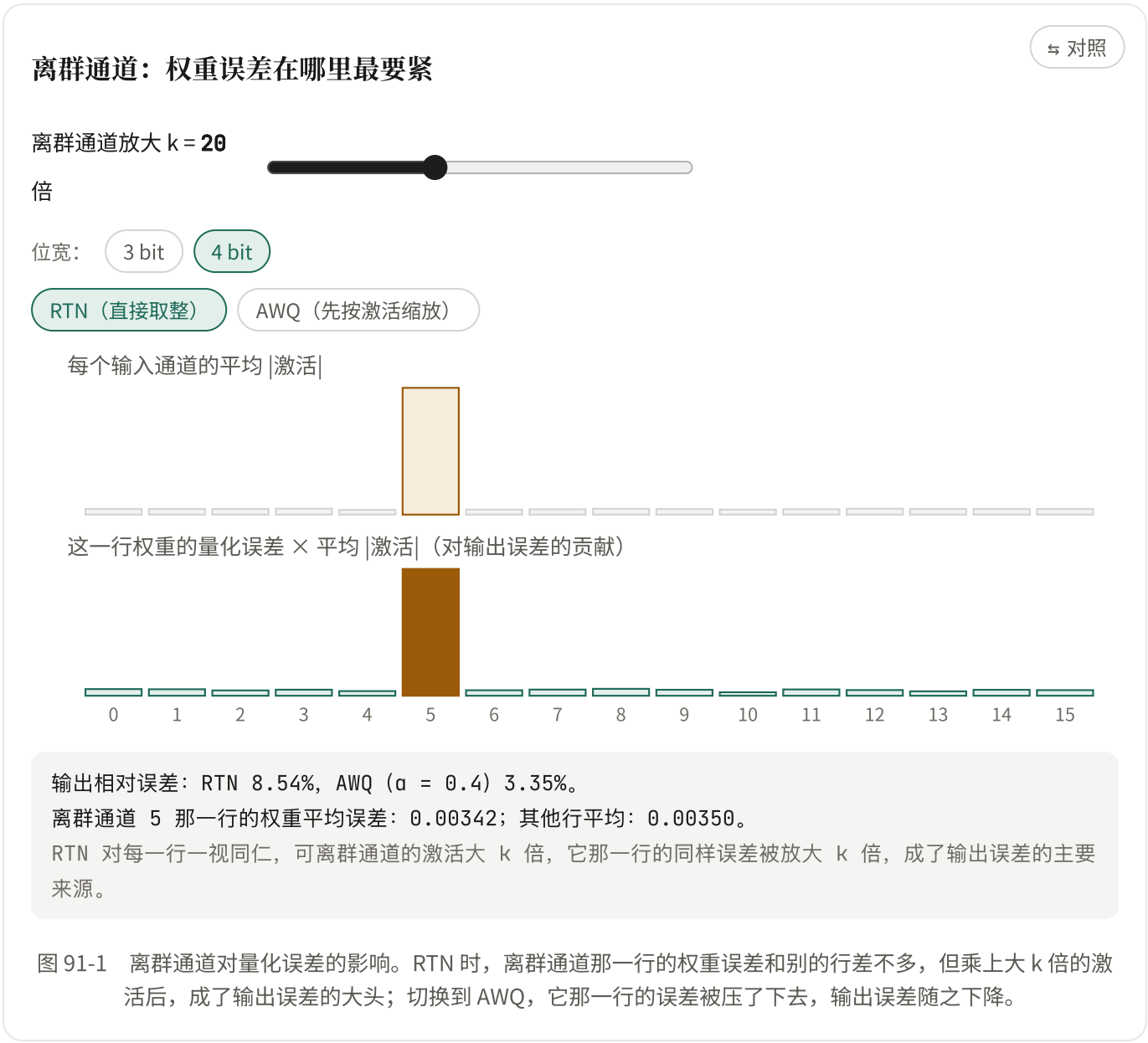


图 91-1　离群通道对量化误差的影响。RTN 时，离群通道那一行的权重误差和别的行差不多，但乘上大 k 倍的激活后，成了输出误差的大头；切换到 AWQ，它那一行的误差被压了下去，输出误差随之下降。

AWQ：给重要的行“放大”再量化

AWQ（Activation-aware Weight Quantization，2023 年）先问：哪些权重最重要？答案是激活大的输入通道对应的那些行。论文发现，只要把大约 1% 最重要的权重保持 FP16 不量化，效果就能恢复很多。可是一个矩阵里混着两种精度，对 GPU kernel 很不友好。

AWQ 用了一个等价变换来达到类似的效果。对第 j 个输入通道，取一个放大系数 $s_j > 1$：

$$x \cdot W = (x / s) \cdot (s \cdot W), \qquad \text{也就是 } x_j \text{ 除以 } s_j，\text{第 } j \text{ 行权重乘以 } s_j$$

数学上完全相等。现在量化的是放大后的 $s \cdot W$。为什么这样误差会变小？同一个输出通道（同一列）的 scale 由这一列绝对值最大的数决定，把第 j 行放大一点，通常不会改变这个最大值，所以 scale 基本不变，取整误差的绝对大小也基本不变；可是反量化后要除以 s_j，这一行的误差就缩小成了原来的 $1/s_j$。重要的行误差变小了，代价是：放大得太多会抬高 scale，其他行的误差变大。

s_j 怎么定？AWQ 让它和激活的大小挂钩：$s_j = (\text{平均}|x_j|)^{\alpha}$，在 0 到 1 之间搜索 α，选让输出误差最小的那个。我们的例子里，最好的 α 是 0.2，输出误差从 10.56% 降到 **6.34%**。离群通道那 4 行的权重误差从 0.0046 降到 0.0024，其余行从 0.0046 升到 0.0056：用一点不重要的误差，换来重要的误差减半。

x 要除以 s，这个除法不用真的做：它可以合并进前一个运算里，比如前面 RMSNorm 的权重（第 36 讲）直接除以 s。所以 AWQ 量化后的模型，推理时没有任何额外开销。

GPTQ：取整之后，让别的权重来补偿

GPTQ（2022 年）换了一个角度。一个权重四舍五入以后产生了误差，能不能**调整还没量化的其他权重**，把这个误差抵消掉一部分?

回到输出误差 $\Sigma_j x_j \cdot (W_j - \hat{W}_j)$。如果第 1 个输入通道和第 2 个输入通道的激活是相关的（比如 x_2 常常约等于 x_1 的一半），那么第 1 行取整多出来的误差，可以通过把第 2 行往反方向调一点来抵消。真实的激活各通道之间确实有很强的相关性，这种补偿就有了用武之地。

GPTQ 的做法是：**一行一行地量化**。量化第 i 行时，先四舍五入得到它的误差，再按一个由校准数据算出的矩阵，把这个误差分摊到后面还没量化的行上，然后再量化下一行。这个矩阵来自 $H = X^{T}X$（输入的二阶统计量）的逆，它告诉我们“第 i 个通道的误差，用第 k 个通道来补偿，最划算的量是多少”。数学推导来自更早的 OBS（Optimal Brain Surgeon）方法，GPTQ 的贡献是把它做得足够快：利用 H^{-1} 的 Cholesky 分解，所有输出通道一起处理，一个 1750 亿参数的模型几个 GPU 小时就能量化完。

实践中还有一个技巧：**先量化最重要的行**（H 的对角线最大的，也就是激活最大的通道），这样它们的误差有最多的行来补偿，自己却不会被别人的补偿再改动。

我们的例子里，GPTQ 的输出误差是 **5.30%**，比 AWQ 还小。有意思的是，它的**权重**平均误差反而最大（0.016，RTN 只有 0.0046）：为了补偿，很多权重被有意地调离了原来的值。这正好说明，权重误差不是该优化的目标。

要点

权重量化该最小化的是输出误差，而输出误差取决于激活：激活大的通道，它那一行权重的误差被放大得最多。AWQ 用等价缩放把这些重要的行放大后再量化，让它们的相对误差变小；GPTQ 逐行量化，把每一行的取整误差按 $H = X^\top X$ 的逆摊给还没量化的行去补偿。两者都只需要一小批校准数据，存储格式和 RTN 一样。

位宽越低，差别越大

代码也比较了 3 比特和 8 比特：

- **8 比特**：RTN 0.58%，AWQ 0.36%，GPTQ 0.30%。都很小，RTN 就够用。
- **4 比特**：RTN 10.56%，AWQ 6.34%，GPTQ 5.30%。
- **3 比特**：RTN 23.99%，AWQ 14.71%，GPTQ 13.07%。

格点越粗，“取整取到哪里”的选择越重要，聪明的方法优势越大。这也是为什么 8 比特量化很少需要 GPTQ、AWQ，而 4 比特及以下几乎离不开它们。

到了大模型里

INT4 权重量化是部署的常见选择。Hugging Face 上大量模型都有 GPTQ、AWQ 的 4 比特版本，通常每组 128 个权重一个 scale（第 90 讲）。vLLM、SGLang 可以直接加载它们，用专门的 W4A16 kernel（比如 Marlin）在读进片上之后反量化、再做 BF16 的矩阵乘法。小批量 decode 时速度能提高两三倍。

效果损失因模型而异。参数量大的模型通常更“耐”量化，4 比特几乎看不出差别；小模型、训练数据特别多的模型（每个参数“记”的东西更多）损失更明显。量化后一定要在自己关心的任务上评测（第 51 讲），不能只看困惑度。

更低的位宽。3 比特、2 比特需要更复杂的方法，比如把几个权重一起用一个码本向量表示（向量量化），或者先对权重做随机旋转让分布更均匀再量化。它们能把模型压得更小，但推理 kernel 也更复杂。

校准数据的影响。校准数据最好和实际的使用场景相近。用纯英文的数据校准，再拿去处理中文或代码，效果可能会差一些。

用代码复现

下面的代码构造了上面那个 64 × 64 的层（激活各通道之间有相关性，4 个离群通道），分别实现 RTN、AWQ（搜索 α）、GPTQ（先量化激活大的行，逐行补偿），比较权重误差和输出误差，以及离群通道那几行的误差，最后对比 3 比特和 8 比特。

```python
import numpy as np
rng = np.random.default_rng(0)

# 一个线性层 y = x · W，W 的形状 [输入 64，输出 64]。校准数据 X：256 个 token，其中 4 个输入通道是“离群通道”，数值大 20 倍
d = 64
W = rng.standard_normal((d, d)) * 0.05
X = rng.standard_normal((256, d)) @ (np.eye(d) + 0.3 * rng.standard_normal((d, d)))
# 各通道之间有相关性，像真实的激活
X[:, [3, 17, 40, 58]] *= 20

def rtn(W, bits=4):
    """直接四舍五入（RTN），每个输出通道（W 的每一列）一个 scale"""
    qmax = 2 ** (bits - 1) - 1
    s = np.abs(W).max(0, keepdims=True) / qmax
    return np.clip(np.round(W / s), -qmax, qmax) * s

def err(Wq):                                          # 我们真正关心的：输出变了多少（相对误差）
    return np.linalg.norm(X @ W - X @ Wq) / np.linalg.norm(X @ W)

def awq(W, bits=4):
    """AWQ：输入通道 j 的权重先乘 s_j 再量化，计算时激活除以 s_j（结果不变）。s_j = (平均|x_j|)^α，搜索最好的 α"""
    best = None
    for a in np.linspace(0, 1, 11):
        s = np.abs(X).mean(0) ** a
        Wq = rtn(W * s[:, None], bits) / s[:, None]          # 等价于 (X / s) · Q(W · s)
        if best is None or err(Wq) < best[0]: best = (err(Wq), a, Wq)
    return best

def gptq(W, bits=4, damp=0.01):
    """GPTQ：一行一行（一个输入通道一个输入通道）地量化，把这一行的误差按 H⁻¹ 摊到还没量化的行上"""
    qmax = 2 ** (bits - 1) - 1
    s = np.abs(W).max(0) / qmax                                # 和 RTN 一样的 scale
    H = X.T @ X; H += damp * np.mean(np.diag(H)) * np.eye(d)
    p = np.argsort(-np.diag(H))                                # 先量化最重要的行（激活大的通道），让后面的行来补偿
    H, W = H[p][:, p], W[p].copy()
    U = np.linalg.cholesky(np.linalg.inv(H)).T                 # H⁻¹ 的 Cholesky 分解（上三角）
    Q = np.zeros_like(W)
    for i in range(d):
        Q[i] = np.clip(np.round(W[i] / s), -qmax, qmax) * s
```

```
        e = (W[i] - Q[i]) / U[i, i]
        W[i + 1:] -= np.outer(U[i, i + 1:], e)              # 后面的行替它“补偿”误差
    return Q[np.argsort(p)]                                  # 换回原来的行顺序

w_err = lambda Wq: np.abs(W - Wq).mean()
print(f"RTN  4 bit: 权重平均误差 {w_err(rtn(W)):.5f}, 输出相对误差 {err(rtn(W)):.2%}")
e, a, Wa = awq(W)
print(f"AWQ  4 bit: 最好的 α = {a:.1f}, 权重平均误差 {w_err(Wa):.5f}, 输出相对误差
{e:.2%}")
Wg = gptq(W)
print(f"GPTQ 4 bit: 权重平均误差 {w_err(Wg):.5f}, 输出相对误差 {err(Wg):.2%}")
out = [3, 17, 40, 58]; norm = [j for j in range(d) if j not in out]
for name, Wq in [("RTN", rtn(W)), ("AWQ", Wa), ("GPTQ", Wg)]:
    print(f"  {name:4s}: 离群通道那 4 行的权重平均误差 {np.abs(W - Wq)
[out].mean():.5f}, 其余行 {np.abs(W - Wq)[norm].mean():.5f}")
for bits in [3, 8]:
    print(f"{bits} bit: RTN {err(rtn(W, bits)):.2%}, AWQ {awq(W, bits)[0]:.2%}, GPTQ
{err(gptq(W, bits)):.2%}")
```

输出：

```
RTN  4 bit: 权重平均误差 0.00455, 输出相对误差 10.56%
AWQ  4 bit: 最好的 α = 0.2, 权重平均误差 0.00540, 输出相对误差 6.34%
GPTQ 4 bit: 权重平均误差 0.01604, 输出相对误差 5.30%
  RTN : 离群通道那 4 行的权重平均误差 0.00460, 其余行 0.00455
  AWQ : 离群通道那 4 行的权重平均误差 0.00244, 其余行 0.00559
  GPTQ: 离群通道那 4 行的权重平均误差 0.00465, 其余行 0.01680
3 bit: RTN 23.99%, AWQ 14.71%, GPTQ 13.07%
8 bit: RTN 0.58%, AWQ 0.36%, GPTQ 0.30%
```

前三行是 4 比特时三种方法的结果：输出误差 10.56%、6.34%、5.30%，GPTQ 的权重误差反而最大。中间三行把权重误差拆成离群通道那 4 行和其余的行：AWQ 把离群行的误差压到了 0.0024；GPTQ 离群行的误差和 RTN 差不多（它们最先量化，没有被补偿），但其余的行被大幅调整，替它们抵消了误差。最后两行是 3 比特和 8 比特。

容易混淆的地方

1. **GPTQ、AWQ 不是新的数据格式**。量化完的模型还是 INT4 加 scale，推理时的计算和 RTN 量化的模型一样；它们只是更聪明地选了每个整数。
2. **“需要校准数据”不等于“需要训练”**。它们只做前向，记录激活，不做反向传播，几分钟到几小时就能完成。

3. **权重量化不加速计算受限的部分**。W4A16 省的是读权重的带宽，decode 小批量时收益最大；prefill 和大批量 decode 是计算受限的，几乎不会变快。要让计算本身变快，需要激活也量化，这是下一讲的话题。

回顾

直接四舍五入的 RTN 在 4 比特以下损失明显。GPTQ 和 AWQ 都盯住输出误差而不是权重误差，用一小批校准数据记录每层的输入。AWQ 发现激活大的通道对应的权重最重要，用 $x \cdot W = (x/s) \cdot (s \cdot W)$ 的等价缩放把这些行放大后再量化，让它们的相对误差缩小，s 可以合并进前一层，没有额外开销。GPTQ 逐行量化，把每一行的取整误差按 $H = X^{\top}X$ 的逆摊给还没量化的行去补偿。人造的例子里，4 比特时输出误差从 RTN 的 10.56% 降到 AWQ 的 6.34%、GPTQ 的 5.30%，位宽越低差别越大。

权重可以在部署前慢慢量化，激活却是每次推理时才产生的。要让矩阵乘法用 INT8 来算，激活也必须量化，可激活里的离群通道比权重里的还要极端。怎么办？下一讲：**激活量化与 SmoothQuant**。

第92讲 激活量化与 SmoothQuant

13 神经元与线性层　90 量化基础　91 权重量化：GPTQ 与 AWQ

上一讲的 GPTQ、AWQ 只量化权重，矩阵乘法仍然用 BF16 来算。它们让 decode 读权重更快，却没有让**计算**更快。prefill、大批量的 decode 是计算受限的（第 68、70 讲），要加速它们，得让矩阵乘法本身用更低的精度来算：INT8 的 Tensor Core 算力是 BF16 的两倍。这就要求**权重和激活都量化**，叫 W8A8。

权重是固定的，可以离线慢慢量化；激活是每次推理时才算出来的，而且大模型的激活里有一类非常棘手的东西：**离群通道**。这一讲讲为什么它让激活量化变得困难，以及 SmoothQuant 怎么用一个简单的等价变换绕过它。

激活里的离群通道

2022 年的 LLM.int8() 论文发现：模型规模超过六七十亿参数以后，Transformer 每一层的输入激活里，会出现少数几个**固定的通道**，数值比其他通道大几十倍甚至上百倍。而且它们几乎在每个 token 上都出现，就是那几个通道。权重里则没有这么极端的情况。

我们造一个类似的例子：一层的输入 X 有 256 个 token、64 个通道，其中 3 个通道是离群通道。普通通道的最大值约 3.0，离群通道的最大值约 178。

为什么不能逐通道量化激活

第 90 讲说过，对付离群值的办法是缩小共用 scale 的范围。对激活来说，最自然的是每个**通道**一个 scale，把离群通道单独处理。可惜，这在矩阵乘法里行不通。

看 $Y = X \cdot W$ 的一个元素：$Y_{ik} = \Sigma_j X_{ij} \cdot W_{jk}$。量化后，$X_{ij} \approx a \cdot qx_{ij}$，$W_{jk} \approx b \cdot qw_{jk}$，我们希望用整数做乘加，最后再乘一次 scale：

$$Y_{ik} \approx a \cdot b \cdot \Sigma_j\, qx_{ij} \cdot qw_{jk}$$

这要求 a、b 在求和的过程中**不随 j 变化**，才能提到求和号外面。所以：

- 激活的 scale 可以随 i（token）变：**每个 token 一个 scale**；

- 权重的 scale 可以随 k（输出通道）变：**每个输出通道一个 scale**。

而“激活每个通道一个 scale”，scale 随 j 变，提不出去，整数矩阵乘法就没法用了。

于是只能每个 token 一个 scale。可每个 token 里都有那几个离群通道，scale 由它们决定（一个 token 里离群通道的值常有几十上百，scale 就是它除以 127），普通通道的值只在 ±3 以内，只能落在少数几个格点上。代码里，直接 W8A8 的激活整体误差只有 1.52%（被离群通道的大数撑着），**普通通道的激活误差却高达 17.40%**，输出误差 1.70%。在真实模型里，这会让效果明显下降。

LLM.int8() 的做法是把离群通道拆出来，用 FP16 单独算，其余的用 INT8 算。效果很好，但一个矩阵乘法拆成两个、精度又不同，速度反而可能变慢。

SmoothQuant：把难度迁移给权重

SmoothQuant（2022 年）注意到：激活难量化，是因为它的通道之间差别太大；权重好量化，因为它的通道之间差别不大。能不能把激活的“难”匀一部分给权重?

用的还是上一讲 AWQ 那个等价变换：

$$X \cdot W = (X / s) \cdot (s \cdot W), \qquad \text{第 } j \text{ 个通道：激活除以 } s_j\text{，权重的第 } j \text{ 行乘以 } s_j$$

离群通道的 s_j 取得大，激活这一列就被压小了；对应的权重行被放大。SmoothQuant 给出的取法是

$$s_j = \max|X_j|^{\alpha} \div \max|W_j|^{(1-\alpha)}$$

α 叫**迁移强度**。$\alpha = 0$ 时几乎不动激活；$\alpha = 1$ 时激活每个通道的最大值都被压成 1，所有的不均匀都转移到了权重上；$\alpha = 0.5$ 时，平滑后激活和权重第 j 个通道的最大值相等，都是 $\sqrt{(\max|X_j| \cdot \max|W_j|)}$，两边一样“难”。

代码扫了几个 α，看平滑后激活和权重的“最大值 ÷ 中位数”（越接近 1，通道之间越均匀，越好量化）和 W8A8 的输出误差：

- $\alpha = 0$：激活 59.8、权重 1.0，输出误差 1.75%。激活太难。
- $\alpha = 0.5$：激活 7.7、权重 7.7，输出误差 **0.50%**。两边一样难，误差最小。
- $\alpha = 1$：激活 1.0、权重 59.8，输出误差 1.88%。权重太难。

SmoothQuant 的论文里，大部分模型取 α = 0.5 左右效果最好。

下面的演示画出了平滑前后每个通道的最大值（对数坐标）。从 α = 0 拖到 1，看离群通道的高度怎么从激活那边“搬”到权重那边。

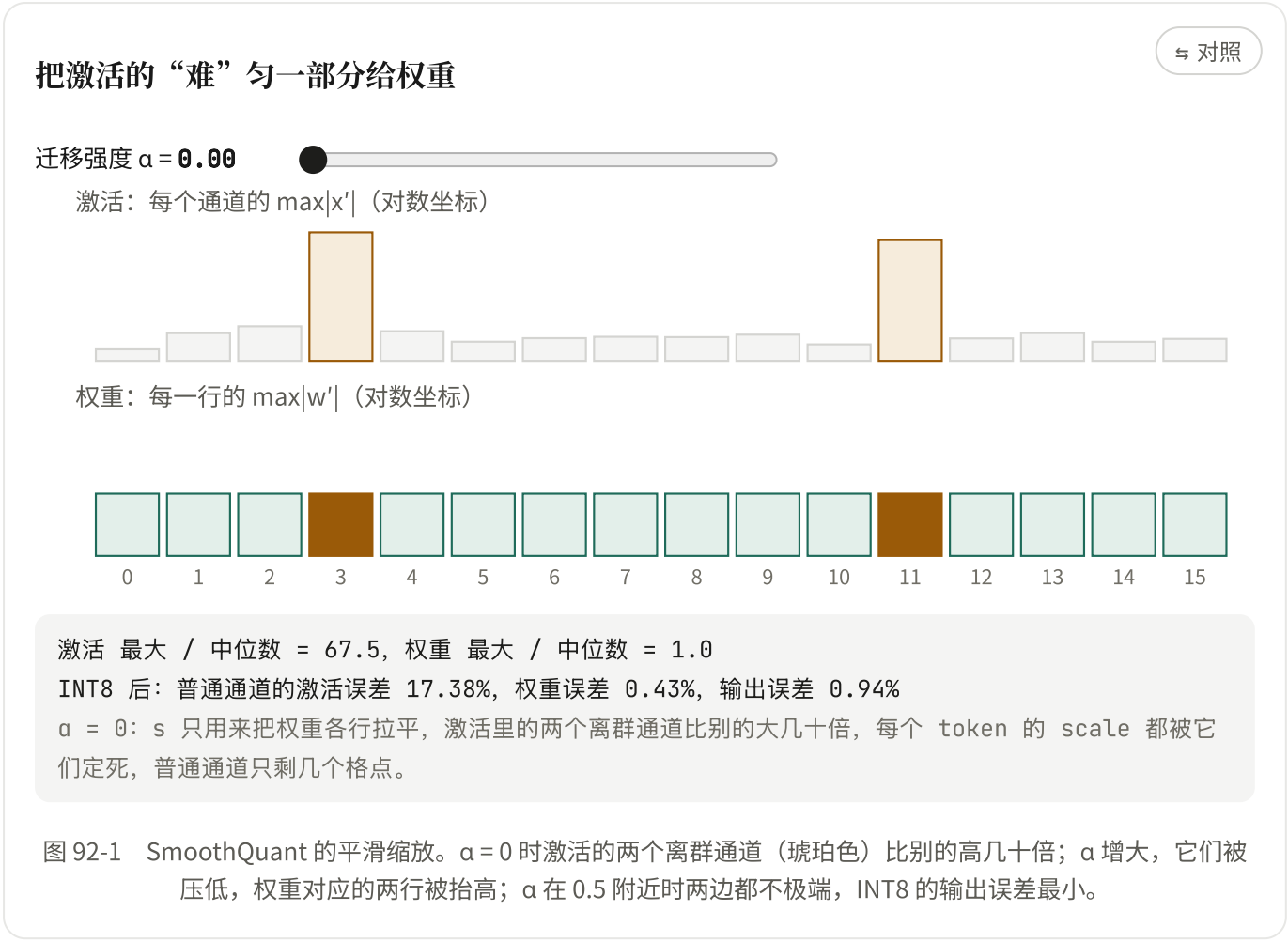


图 92-1　SmoothQuant 的平滑缩放。α = 0 时激活的两个离群通道（琥珀色）比别的高几十倍；α 增大，它们被压低，权重对应的两行被抬高；α 在 0.5 附近时两边都不极端，INT8 的输出误差最小。

和 AWQ 一样，激活除以 s 不用真的在推理时做：s 在离线时用校准数据算好，合并进前一层（比如前面 RMSNorm 的权重），权重乘以 s 也离线做完。推理时没有任何额外开销，矩阵乘法照常用 INT8。

要点

激活里有少数固定的离群通道，而整数矩阵乘法只允许激活每个 token 一个 scale，离群通道就把普通通道的精度挤没了。SmoothQuant 用 $X \cdot W = (X/s) \cdot (s \cdot W)$ 的等价变换，$s_j = \max|X_j|^{\alpha} \div \max|W_j|^{(1-\alpha)}$，把激活的不均匀按比例迁移给权重，α 取 0.5 左右时两边都好量化。s 合并进前一层，推理时没有开销。

静态还是动态

激活的 scale 什么时候算？有两种做法：

- **静态**：用校准数据提前算好每层激活的 scale，推理时直接用。最快，但遇到和校准数据差别大的输入，可能截断。
- **动态**：推理时对每个 token 现算 scale（求这一行的最大绝对值）。多一点计算，但更稳。现在的实现多用每个 token 动态的 scale，开销可以融合进前一个 kernel（第 83 讲）。

到了大模型里

FP8 替代了大部分 INT8 的场景。在 H100 这一代 GPU 上，FP8 的 Tensor Core 算力和 INT8 一样，而浮点格式天生更能容忍数值范围大的分布（下一讲）。所以今天的 W8A8 部署，大多用的是 FP8，而不是 INT8。但离群通道的问题并没有消失，FP8 只是更耐得住一些；到了 4 比特，它又成了大问题。

更激进的办法：旋转。要把激活也量化到 4 比特（W4A4），平滑往往不够。一类方法（比如 QuaRot、SpinQuant）在激活和权重之间插入一个**正交矩阵**（比如 Hadamard 矩阵）：$X \cdot W = (X \cdot R) \cdot (R^{\intercal} \cdot W)$。正交矩阵把每个离群通道的“能量”均匀地摊到所有通道上，离群值就消失了。它和平滑一样是等价变换，大部分旋转也可以合并进相邻的权重里。

KV Cache 也是激活。注意力里的 K、V 也是推理时算出来的激活，也有离群通道的问题，第 94 讲会专门讲。

用代码复现

下面的代码构造了一个有 3 个离群通道的激活，先看直接 W8A8 的误差（激活每个 token 一个 scale，权重每个输出通道一个 scale），再扫描 SmoothQuant 的 α，最后对照“激活逐通道量化”这种硬件做不了的方式。

```python
import numpy as np
rng = np.random.default_rng(0)

# 一层的输入 X（256 个 token × 64 个通道），其中 3 个通道是离群通道，数值大 50 倍；权重 W 没有离群值
d = 64
X = rng.standard_normal((256, d)); X[:, [5, 30, 47]] *= 50
W = rng.standard_normal((d, d)) * 0.05

def q8(A, axis):
    """INT8 对称量化，沿 axis 求最大值：axis=1 → 每行（每个 token）一个 scale；axis=0 → 每列一个 scale"""
    s = np.abs(A).max(axis, keepdims=True) / 127
    return np.round(A / s) * s

def rel(a, b): return np.linalg.norm(a - b) / np.linalg.norm(a)
Y = X @ W
print(f"每个通道的最大 |x|：普通通道约 {np.abs(np.delete(X, [5, 30, 47], 1)).max(0).mean():.1f}，离群通道约 {np.abs(X[:, [5, 30, 47]]).max(0).mean():.1f}")

# W8A8：激活每个 token 一个 scale，权重每个输出通道一个 scale——两者都能在矩阵乘法之后再乘回去
Xq, Wq = q8(X, 1), q8(W, 0)
print(f"直接 W8A8：激活误差 {rel(X, Xq):.2%}，普通通道的激活误差 {rel(np.delete(X, [5, 30, 47], 1), np.delete(Xq, [5, 30, 47], 1)):.2%}，输出误差 {rel(Y, Xq @ Wq):.2%}")

# SmoothQuant：s_j = max|X_j|^α / max|W_j|^(1-α)，X' = X / s，W' = s · W，X'W' = XW
for a in [0.0, 0.25, 0.5, 0.75, 1.0]:
    s = np.abs(X).max(0) ** a / np.abs(W).max(1) ** (1 - a)
    Xs, Ws = X / s, W * s[:, None]
    print(f"α = {a:.2f}：激活最大值 / 中位数 = {np.abs(Xs).max() / np.median(np.abs(Xs).max(0)):5.1f}，"
          f"权重最大值 / 中位数 = {np.abs(Ws).max() / np.median(np.abs(Ws).max(1)):5.1f}，输出误差 {rel(Y, q8(Xs, 1) @ q8(Ws, 0)):.2%}")

# 对照：激活每个“通道”一个 scale。这个 scale 落在求和的维度上，没法在矩阵乘法之后再乘回去
print(f"对照：激活逐通道量化（硬件做不了）输出误差 {rel(Y, q8(X, 0) @ Wq):.2%}")
```

输出：

```
每个通道的最大 |x|：普通通道约 3.0，离群通道约 178.1
直接 W8A8：激活误差 1.52%，普通通道的激活误差 17.40%，输出误差 1.70%
α = 0.00：激活最大值 / 中位数 =  59.8，权重最大值 / 中位数 =   1.0，输出误差 1.75%
α = 0.25：激活最大值 / 中位数 =  21.5，权重最大值 / 中位数 =   2.8，输出误差 0.71%
α = 0.50：激活最大值 / 中位数 =   7.7，权重最大值 / 中位数 =   7.7，输出误差 0.50%
α = 0.75：激活最大值 / 中位数 =   2.8，权重最大值 / 中位数 =  21.5，输出误差 0.76%
α = 1.00：激活最大值 / 中位数 =   1.0，权重最大值 / 中位数 =  59.8，输出误差 1.88%
对照：激活逐通道量化（硬件做不了）输出误差 0.99%
```

第一行是离群的程度：普通通道最大约 3.0，离群通道约 178.1。第二行是直接 W8A8：普通通道的激活误差 17.40%。中间五行是不同 α 的结果，$\alpha = 0.5$ 时激活和权重的“最大值 ÷ 中位数”都是 7.7，输出误差最小（0.50%）。最后一行是逐通道量化激活的对照：误差 0.99%，还不如平滑后的 W8A8，而且硬件做不了。

容易混淆的地方

1. **SmoothQuant 不改变模型的数学**。平滑前后 X·W 完全相等，变的只是“哪一边更难量化”。
2. **它和 AWQ 用的是同一个变换，目的不同**。AWQ 只量化权重，放大重要的行是为了让这些行的相对误差变小；SmoothQuant 两边都量化，是为了在激活和权重之间平衡难度。
3. **激活的离群通道不是噪声**。它们对模型的输出很重要（去掉它们，模型效果会大幅下降），所以不能简单地截断，只能想办法“安置”它们。

回顾

要让矩阵乘法本身用低精度计算，激活也必须量化。大模型的激活里有少数固定的离群通道，比普通通道大几十上百倍；而整数矩阵乘法只允许激活每个 token 一个 scale、权重每个输出通道一个 scale，离群通道就让普通通道只剩几个格点，例子里普通通道的激活误差达到 17.40%。SmoothQuant 用 $X \cdot W = (X/s) \cdot (s \cdot W)$，取 $s_j = \max|X_j|^{\alpha} \div \max|W_j|^{(1-\alpha)}$，把激活的不均匀迁移一部分给权重；$\alpha = 0.5$ 时两边一样均匀，W8A8 的输出误差从 1.70% 降到 0.50%。s 合并进前一层，推理没有额外开销。

INT8 是整数，格点均匀分布。还有一类低精度格式是浮点数：同样 8 比特，格点在 0 附近密、远处疏，更适合神经网络里大小悬殊的数值。H100 支持 FP8，新一代的 GPU 还支持 FP4。它们是怎么表示数的，用起来要注意什么？下一讲：**FP8 与 FP4**。

第93讲 FP8 与 FP4

重点

63 低精度计算　90 量化基础　92 激活量化与 SmoothQuant

前三讲的量化，都是把实数映射成**整数**：INT8、INT4。整数的格点是等间隔的，0.01 附近和 100 附近的间隔一样大。可是神经网络里的数大多聚在 0 附近，偶尔有几个很大（第 90、92 讲的离群值），等间隔的格点对它们并不友好：小数附近不够密，大数附近又浪费。

浮点数正好相反：0 附近格点密，越往外越疏，每个数的**相对**误差差不多。BF16 就是一种浮点格式。H100 这一代 GPU 开始支持 8 位的浮点数 **FP8**，计算速度是 BF16 的两倍；更新的 Blackwell 系列（B200 等）又加入了 4 位的 **FP4**。这一讲看看这些格式是怎么表示数的，以及为什么 FP4 必须和“分组缩放”一起用。

8 个比特怎么拼成一个浮点数

一个浮点数由三部分组成：**符号** S（1 位）、**指数** E、**尾数** M。它表示的值是

$$(-1)^S \times (1 + M / 2^m) \times 2^{(E - bias)}$$

m 是尾数的位数，bias 是一个固定的偏移，让指数可以取负值。指数决定数的“量级”（是 2 的几次方），尾数决定在这个量级里的精细位置。指数全为 0 时是**次正规数**，$(-1)^S \times (M / 2^m) \times 2^{(1 - bias)}$，用来表示非常接近 0 的数。

8 位可以有不同的分法，FP8 有两种：

- **E4M3**：4 位指数、3 位尾数，bias = 7。最大 448，最小的正数 $2^{-9} \approx 0.00195$。尾数 3 位，所以每个 2 的幂之间有 8 个格点：1 到 2 之间是 1, 1.125, 1.25, ⋯, 1.875，间隔 0.125。
- **E5M2**：5 位指数、2 位尾数，bias = 15。最大 57344，最小的正数约 1.5×10^{-5}。范围大得多，但每个 2 的幂之间只有 4 个格点，精度更粗。

E4M3 精度更好，推理时权重和激活一般都用它；E5M2 范围更大，主要用在训练时数值范围很大的梯度上。

FP4 只有一种常用的分法，**E2M1**：2 位指数、1 位尾数。它一共只能表示 8 个非负值：**0, 0.5, 1, 1.5, 2, 3, 4, 6**。

下面的演示可以点每一位，看它表示的值；下面的数轴（对数坐标）画出了这种格式能表示的所有正数。

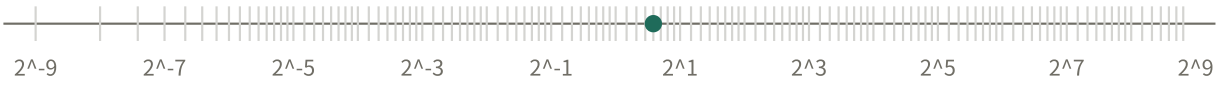
8 位、4 位的浮点数长什么样

⇆ 对照

FP8 E4M3 | FP8 E5M2 | FP4 E2M1

0	0	1	1	1	1	0	0

符号 1 位 · 指数 4 位 · 尾数 3 位（点一位翻转）



`(1 + 4/8) × 2^(7 - 7) = 1.5`
能表示 127 个非负值，最大 448，最小的正数 0.001953125。
数轴是对数坐标：每个 2 的幂之间格点数一样多，所以格点的“相对”间隔处处差不多——小数附近格点密，大数附近格点疏。整数格式则是等间隔的。

分组缩放：16 个数量化成 FP4

16 个共用一个 scale | 每 8 个一组

x	0.03	-0.12	0.07	2.4	-0.04	0.18	-0.09	0.02	0.11	-0.06	0.14
x ÷ scale → FP4	**0**	**-0.5**	**0**	**6**	**0**	**0.5**	**0**	**0**	**0.5**	**0**	**0.5**
x̂	0.000	-0.200	0.000	2.400	0.000	0.200	0.000	0.000	0.200	0.000	0.200

`scale = 组内最大 |x| ÷ 6：0.4000。除离群值以外的 15 个数，相对误差 56.3%，有 9 个被量化成了 0。`
第 4 个数 2.4 是离群值。整体一个 scale 时，它把 scale 撑到 0.4，其余的数除以 scale 后都不到 0.6，只能变成 0 或 ±0.5。

图 93-1　上：FP8、FP4 的位布局，点一位翻转，看值怎么算；数轴是对数坐标，每个 2 的幂之间格点数一样多，所以相对精度处处相近。下：16 个数用 FP4 量化，一个离群值让整体 scale 下的其他数几乎都成了 0 或 ±0.5，分组后好得多。

为什么浮点更适合神经网络

整数格式 INT8 的格点是 −127 到 127 乘以一个 scale，等间隔。E4M3 的格点在对数坐标上均匀：从 2^{-6} 到 2^{8}，每段 8 个格点。同样是 8 位，INT8 在大数附近更精细，E4M3 在小数附近更精细、能覆盖的范围大得多。

神经网络的权重和激活大多是“重尾”的：大部分数很小，少数很大。代码造了 4096 个这样的数，用一个整体的 scale 量化：

- **INT8**：相对误差 3.22%；**FP8 E4M3**：2.44%。
- **INT4**：51.25%；**FP4 E2M1**：34.02%。

同样的位数，浮点格式都更好，4 位时差别尤其大：INT4 只有 15 个等间隔的格点，scale 被最大的数撑开以后，大量小数都被量化成了 0；FP4 在 0 附近还有 0.5、1 这样的格点。

不过 34% 的误差显然还是不能用。FP4 只有 8 个非负值，单靠格式本身扛不住一整个张量的数值范围。

FP4 离不开分组缩放

办法还是第 90 讲的分组：每一小组数配一个自己的 scale。到了 4 位，组要分得很小，scale 本身也要尽量便宜。目前有两种标准做法：

- **MXFP4**（开放计算项目 OCP 的 Microscaling 标准）：每 **32** 个数一组，scale 用 8 位的 E8M0 存，也就是只能是 2 的幂。好处是乘 scale 只要改指数，硬件实现简单；代价是 scale 只能取 2 的幂，最多会浪费将近一个二进制位的精度。平均每个数 4 + 8 ÷ 32 = **4.25** 位。
- **NVFP4**（NVIDIA 在 Blackwell 上推出）：每 **16** 个数一组，scale 用 FP8 E4M3 存，可以取非 2 的幂的值；整个张量再配一个 FP32 的总系数。组更小、scale 更精细，平均每个数 4 + 8 ÷ 16 = **4.5** 位。

对那 4096 个数：INT4 每 32 个一组，误差 15.09%；MXFP4 15.91%；NVFP4 降到 **9.25%**。

关键是，这些分组的 scale **由硬件直接支持**：Blackwell 的 Tensor Core 在做 FP4 矩阵乘法时，自动把每组的 scale 乘进去。所以分组再细，也不会拖慢计算。这和第 92 讲说的“激活的 scale 必须能提到求和号外面”不矛盾：硬件把 K 方向切成了 16 或 32 一段，每段内部用同一个 scale，段与段之间的累加用高精度完成。

要点

FP8 有 E4M3（精度好，最大 448）和 E5M2（范围大，最大 57344）两种，推理多用 E4M3。浮点格式的格点在对数坐标上均匀，比同位数的整数更适合大多是小数、偶有大数的分布。FP4（E2M1）只有 0、0.5、1、1.5、2、3、4、6 八个非负值，必须配合由硬件支持的细粒度分组缩放：MXFP4 每 32 个数一个 2 的幂的 scale，NVFP4 每 16 个数一个 FP8 的 scale。

换来的速度

第 63 讲说过，Tensor Core 的算力随精度翻倍：

- **H100**：FP8 的稠密算力约 1979 TFLOPS，是 BF16（989 TFLOPS）的两倍。
- **Blackwell**：FP4 的算力又是 FP8 的两倍。

对计算受限的 prefill 和大批量 decode，矩阵乘法用 FP8 就能快将近一倍；对带宽受限的小批量 decode，权重从 16 位变成 8 位、4 位，读得也快一倍、三倍多。显存方面，Llama-3-8B 用 FP8 是 8 GB 左右，NVFP4 约 4.5 GB。

到了大模型里

FP8 推理已经很成熟。vLLM、SGLang、TensorRT-LLM 都支持 FP8 的权重和激活（W8A8），常见的 scale 粒度是权重每个输出通道一个、激活每个 token 一个（动态计算）。对大多数模型，FP8 的效果和 BF16 几乎看不出差别，是 H100 上部署的常见选择。

FP8 训练也在普及。DeepSeek-V3 用 FP8 训练：激活每 1×128 个数一组、权重每 128×128 一块，各配一个 scale，再把累加放在更高的精度里做。这种细粒度的分组思路，和 MXFP、NVFP4 一脉相承。

FP4 主要用在 Blackwell 上。有的模型直接发布 FP4 的权重（比如 OpenAI 的 gpt-oss 用 MXFP4 存 MoE 专家的权重），也有工具把 BF16 模型训练后量化成 NVFP4。4 位的误差毕竟大，通常要配合第 91、92 讲的方法（校准、平滑、旋转），或者在训练时就让模型适应它。

KV Cache 也能用 FP8。下一讲专门讲。

用代码复现

下面的代码按格式的定义列出 E4M3、E5M2、E2M1 能表示的所有值，然后用一组重尾分布的数，比较整数和浮点、整体 scale 和分组 scale 的量化误差。

```python
import numpy as np

def fp_values(e, m, bias, ieee_inf):
    """列出一种小浮点格式能表示的全部非负有限值：符号 1 位，指数 e 位，尾数 m 位"""
    vals = set()
    for E in range(2 ** e):
        for M in range(2 ** m):
            if ieee_inf and E == 2 ** e - 1: continue                  # E5M2：指数全 1 留给 inf / NaN
            if not ieee_inf and E == 2 ** e - 1 and M == 2 ** m - 1 and e > 2: continue   # E4M3：只有全 1 是 NaN
            vals.add(M / 2 ** m * 2.0 ** (1 - bias) if E == 0 else (1 + M / 2 ** m) * 2.0 ** (E - bias))
    return np.array(sorted(vals))

E4M3, E5M2, E2M1 = fp_values(4, 3, 7, False), fp_values(5, 2, 15, True), fp_values(2, 1, 1, False)
for name, v in [("FP8 E4M3", E4M3), ("FP8 E5M2", E5M2), ("FP4 E2M1", E2M1)]:
    print(f"{name}：{len(v)} 个非负值，最大 {v.max():g}，最小的正数 {v[1]:g}")
print("FP4 E2M1 的全部非负值：", E2M1.tolist())
print("E4M3 在 1 附近的间隔：", (E4M3[(E4M3 ≥ 1) & (E4M3 ≤ 2)]).tolist())

def to_grid(x, grid):                                          # 取最近的可表示值（带符号）
    g = np.concatenate([-grid[::-1], grid])
    return g[np.abs(x[..., None] - g).argmin(-1)]

# 一组“像权重/激活”的数：大多数很小，少数很大（重尾分布）
rng = np.random.default_rng(0)
x = rng.standard_t(3, 4096) * 0.02
rel = lambda xh: np.linalg.norm(x - xh) / np.linalg.norm(x)
amax = np.abs(x).max()

def int_q(x, bits, group):
    qm = 2 ** (bits - 1) - 1; g = x.reshape(-1, group)
    s = np.abs(g).max(1, keepdims=True) / qm
    return (np.round(g / s) * s).ravel()

def fp_q(x, grid, group, scale_kind):
    g = x.reshape(-1, group); a = np.abs(g).max(1, keepdims=True)
    if scale_kind == "fp32": s = a / grid.max()
    elif scale_kind == "e8m0": s = 2.0 ** np.ceil(np.log2(a / grid.max()))
# MXFP4：scale 只能是 2 的幂
    else:
# NVFP4：scale 本身用 E4M3 存，外加一个整体的 FP32 系数
        t = amax / (grid.max() * E4M3.max()); s = to_grid(a / grid.max() / t, E4M3) * t
```

```
    return (to_grid(g / s, grid) * s).ravel()

print("同一组 4096 个数（重尾分布）的相对误差：")
print(f"  INT8，整体一个 scale：{rel(int_q(x, 8, 4096)):.2%}")
print(f"  FP8 E4M3，整体一个 scale：{rel(fp_q(x, E4M3, 4096, 'fp32')):.2%}")
print(f"  INT4，整体一个 scale：{rel(int_q(x, 4, 4096)):.2%}")
print(f"  FP4 E2M1，整体一个 scale：{rel(fp_q(x, E2M1, 4096, 'fp32')):.2%}")
print(f"  INT4，每 32 个一组：{rel(int_q(x, 4, 32)):.2%}")
print(f"  MXFP4（每 32 个一组，scale 是 2 的幂）：{rel(fp_q(x, E2M1, 32,
'e8m0')):.2%}")
print(f"  NVFP4（每 16 个一组，scale 用 E4M3）：{rel(fp_q(x, E2M1, 16,
'e4m3')):.2%}")
print(f"每个数平均占几位：MXFP4 = 4 + 8/32 = {4 + 8 / 32}，NVFP4 = 4 + 8/16 = {4 + 8 /
16}")
```

输出：

```
FP8 E4M3：127 个非负值，最大 448，最小的正数 0.00195312
FP8 E5M2：124 个非负值，最大 57344，最小的正数 1.52588e-05
FP4 E2M1：8 个非负值，最大 6，最小的正数 0.5
FP4 E2M1 的全部非负值：  [0.0, 0.5, 1.0, 1.5, 2.0, 3.0, 4.0, 6.0]
E4M3 在 1 附近的间隔：  [1.0, 1.125, 1.25, 1.375, 1.5, 1.625, 1.75, 1.875, 2.0]
同一组 4096 个数（重尾分布）的相对误差：
  INT8，整体一个 scale：3.22%
  FP8 E4M3，整体一个 scale：2.44%
  INT4，整体一个 scale：51.25%
  FP4 E2M1，整体一个 scale：34.02%
  INT4，每 32 个一组：15.09%
  MXFP4（每 32 个一组，scale 是 2 的幂）：15.91%
  NVFP4（每 16 个一组，scale 用 E4M3）：9.25%
每个数平均占几位：MXFP4 = 4 + 8/32 = 4.25，NVFP4 = 4 + 8/16 = 4.5
```

前三行是三种格式的值的个数和范围（E4M3 的 127 个非负值里，扣掉了唯一的 NaN）；第四行是 FP4 的全部 8 个非负值；第五行是 E4M3 在 1 到 2 之间的格点。后面是量化误差的比较：整体 scale 时浮点都比整数好；分组以后，NVFP4 的 9.25% 最好。最后一行是两种 FP4 方案每个数平均占的位数。

容易混淆的地方

1. **FP8 有两种，不能混用。**E4M3 和 E5M2 的位含义不同，存储和计算时要明确用哪一种。说“FP8 推理”时，一般指 E4M3。

2. **FP4 不是“FP8 再砍一半”那么简单**。8 个非负值的格式，离开分组缩放几乎不能用；它的效果很大程度上取决于组的大小和 scale 的格式。
3. **硬件不支持，就没有加速**。在不支持 FP8 计算的 GPU（比如 A100）上，FP8 权重只能先转回 BF16 再算，省的是显存和带宽，不是计算；FP4 的计算加速同样只在 Blackwell 这类新硬件上才有。

回顾

浮点数用符号、指数、尾数表示：(−1)^S × (1 + M/2^m) × 2^(E − bias)。FP8 有 E4M3（最大 448、精度较好）和 E5M2（最大 57344、范围大）两种；FP4 E2M1 只有 0、0.5、1、1.5、2、3、4、6 八个非负值。浮点的格点在对数坐标上均匀，比同位数的整数更适合重尾分布：整体 scale 下，FP4 的误差 34.02%，INT4 是 51.25%。FP4 必须配合细粒度的分组缩放，MXFP4 每 32 个数一个 2 的幂的 scale，NVFP4 每 16 个数一个 E4M3 的 scale，后者误差降到 9.25%，而且由 Blackwell 的 Tensor Core 直接支持。H100 上 FP8 的算力是 BF16 的两倍，Blackwell 上 FP4 又翻一倍。

权重和激活都可以用低精度了。还有一大块显存是 KV Cache：它决定能同时服务多少个请求，decode 时还要每一步都读一遍。把它也压缩成 8 位、4 位，会怎样？下一讲：**KV Cache 量化**。

第94讲 KV Cache 量化

69 KV Cache　71 显存账本　85 Decode 注意力 Kernel　90 量化基础　93 FP8 与 FP4

第 71 讲的显存账本说：一张 80 GB 的卡跑 Llama-3-8B，权重占 16 GB，留给 KV Cache 的约 49 GiB，能放约 40 万个 token。每个 token 的 KV Cache 是 128 KiB（BF16）。如果每个请求有 4096 个 token，最多只能同时服务 98 个请求。

前几讲压缩了权重。可在并发高、上下文长的场景里，KV Cache 往往比权重还大：98 个请求的 KV 就有 49 GiB，是权重的三倍。而且 decode 的每一步都要把所有请求的 KV 读一遍（第 85 讲）。这一讲把 KV Cache 也量化，看它能换来多少并发和吞吐，又要注意什么。

字节减半，并发翻倍

KV Cache 的大小 = token 数 × 每个 token 的字节数。池子的大小固定，每个 token 的字节数减半，能放的 token 就翻倍：

- **BF16**：每个 token 128 KiB，池子放 403,865 个 token，4096 token 的请求 98 个。
- **FP8**：64 KiB，807,731 个 token，197 个请求。
- **INT4**（每 128 个数配一个 FP16 的 scale）：33 KiB，1,566,508 个 token，382 个请求。

并发多了，速度会不会变慢？代码估算了满载时 decode 一步的耗时：BF16 98 个请求时约 20.5 ms，FP8 197 个请求时约 20.6 ms，几乎一样。原因是满载时 decode 一步的时间主要花在读 KV 上（98 × 4096 个 token 的 KV 有 49 GiB，比权重的 16 GB 大得多）。KV 的字节减半，同样的时间能读两倍的请求，所以 **TPOT 几乎不变，吞吐翻倍**：从 4781 token/s 到 9573 token/s。INT4 时吞吐又翻一倍，约 18,564 token/s。

注意力 kernel 读进片上之后，先把 FP8 或 INT4 的 KV 反量化成 BF16 再计算（第 85 讲），从显存读的字节少了，计算方式不变。

下面的计算器可以调请求长度，看每种精度能放多少个请求，以及满载时的 TPOT 和吞吐。

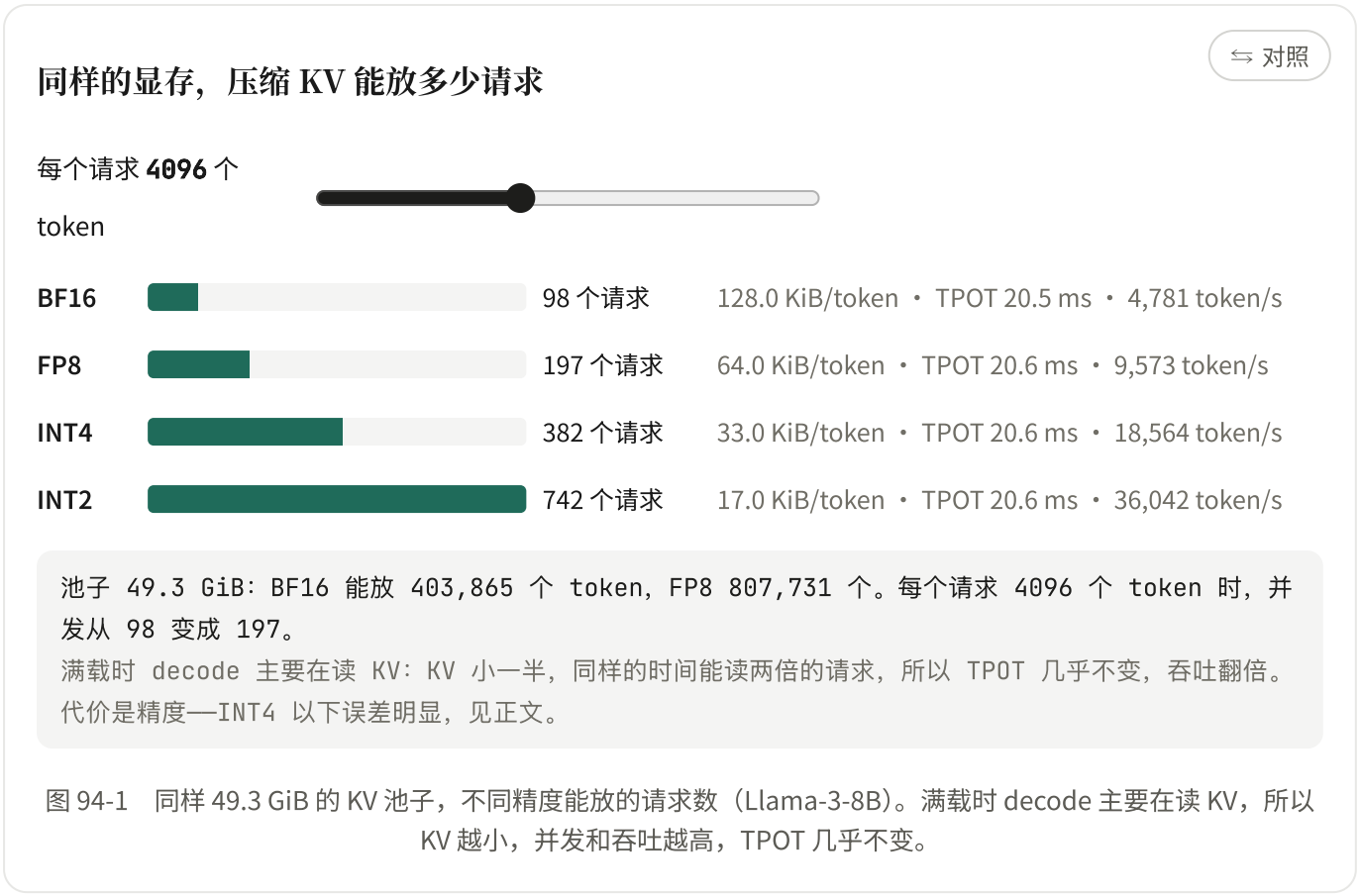


图 94-1　同样 49.3 GiB 的 KV 池子，不同精度能放的请求数（Llama-3-8B）。满载时 decode 主要在读 KV，所以 KV 越小，并发和吞吐越高，TPOT 几乎不变。

要点

KV Cache 量化让每个 token 的 KV 字节数变小，同样的显存能放更多 token、服务更多请求。满载的 decode 主要花在读 KV 上，所以字节减半时 TPOT 几乎不变、吞吐翻倍。它对长上下文、高并发的场景收益最大。

精度：K 比 V 难

KV 量化的误差会进入注意力：K 的误差让打分变了，softmax 之后的权重跟着变；V 的误差直接进入加权求和。研究者发现，K 和 V 的数值分布很不一样：

- **K 有离群通道**。和第 92 讲的激活类似，K 向量的少数几维（固定的那几个通道）在几乎所有 token 上都特别大。一般认为这和 RoPE（第 37 讲）转得慢的那些维度有关。
- **V 比较均匀**，没有明显的离群通道。

这就决定了 scale 该怎么分组。代码造了一个类似的例子：512 个 token、每个头 128 维，K 的两个通道大 15 倍，V 正常，比较注意力输出的误差：

- FP8（K、V 各一个整体的 scale）：3.87%。
- INT8，每个 token 一个 scale：5.00%。

- **INT4，K、V 都每个 token 一个 scale：64.71%。**K 的每个 token 向量里都有那两个大通道，scale 被它们撑开，其余 126 维几乎都被量化成 0。
- **INT4，K 每个通道一个 scale、V 每个 token 一个：18.84%。**离群通道单独一个 scale，其余通道的精度保住了。

这个“K 按通道、V 按 token”的分法来自 KIVI 等论文。和第 92 讲不同，KV Cache 不需要做整数矩阵乘法，读进片上以后先反量化，所以 scale 沿哪个方向分都可以。

不过 18.84% 的误差仍然不小，4 位以下的 KV 量化还需要更多技巧：比如最近的一小段 token 保持高精度（它们往往最受关注，而且按通道量化需要攒够一组 token 才能算 scale），或者对 K 先做旋转（第 92 讲）把离群值摊平。

到了大模型里

FP8 KV Cache 已经是常规选项。vLLM、SGLang、TensorRT-LLM 都支持把 KV Cache 存成 FP8（vLLM 的参数 `kv_cache_dtype="fp8"`），scale 可以是每层一个，也可以从校准数据里算出。对大多数模型，FP8 KV 的效果损失很小，是提高并发最省事的办法。

INT4、更低的位宽还在发展中。它们的压缩比更高，但需要细致的分组和专门的注意力 kernel，效果也更依赖模型，部署前要认真评测，尤其是长上下文的任务（第 51 讲）。

和其他省 KV 的办法叠加。GQA、MLA（第 39、40 讲）从模型结构上减少 KV 的数量；分页和前缀缓存（第 75、76 讲）减少浪费和重复；KV 量化减少每个数的位数。它们可以叠加使用。另外，KV 也可以搬到 CPU 内存或其他机器上（第 14 篇），量化后搬运的字节也少了。

量化 KV 和量化权重是独立的。可以只量化 KV，权重仍用 BF16；也可以 FP8 权重配 FP8 KV。KV 量化主要提高并发和长上下文的能力，权重量化主要省下权重的显存和读取时间。

用代码复现

下面的代码先按第 71 讲的池子算出不同精度能放多少 token 和请求，以及满载时 decode 一步的时间和吞吐；再造一个 K 有离群通道的例子，比较几种量化方式的注意力输出误差。

```python
import numpy as np

# 1. 同样 49.3 GiB 的 KV Cache 池子（第 71 讲），不同精度能放多少个 token、多少个 4096 token 的请求
POOL = 49.3 * 2**30
ELEMS = 2 * 32 * 8 * 128                                    # Llama-3-8B 每个 token 的 K、V 元素数（65536 个）
N, NE, L, d, F, BW = 8.03e9, 128256 * 4096, 32, 4096, 989e12, 3.35e12
print("精度 | 每个 token | 池子能放的 token | 4096 token 的请求 | 满载时 TPOT | 吞吐")
for name, bits in [("BF16", 16), ("FP8", 8), ("INT4（每 128 个一个 FP16 scale）", 4 + 16 / 128)]:
    per = ELEMS * bits / 8
    toks = int(POOL // per); b = toks // 4096
    t = max((2 * (N - NE) + 4 * L * d * 4096) * b / F, (N * 2 + per * 4096 * b) / BW)   # decode 一步（上下文都按 4096 算）
    print(f"{name} | {per / 1024:6.1f} KiB | {toks:9,d} | {b:4d} | {t * 1e3:5.1f} ms | {b / t:7,.0f} token/s")

# 2. 精度：K 里有几个“离群通道”（固定的几维总是很大），V 没有
rng = np.random.default_rng(0)
T, dh = 512, 128
K = rng.standard_normal((T, dh)); K[:, [7, 70]] *= 15
V = rng.standard_normal((T, dh))
Q = rng.standard_normal((16, dh)); Q[:, [7, 70]] *= 0.2

def attn(K, V):
    s = Q @ K.T / np.sqrt(dh); p = np.exp(s - s.max(1, keepdims=True)); p /= p.sum(1, keepdims=True)
    return p @ V
def q(A, axis, qm=7):                                       # 对称量化（默认 INT4），axis=1：每个 token 一个 scale；axis=0：每个通道一个 scale
    s = np.abs(A).max(axis, keepdims=True) / qm
    return np.clip(np.round(A / s), -qm, qm) * s
def fp8(A):                                                 # 粗略模拟 E4M3：保留 3 位尾数（整体一个 scale 把最大值放到 448）
    s = np.abs(A).max() / 448; x = A / s
    e = np.floor(np.log2(np.maximum(np.abs(x), 2.0**-6)))
    return np.round(x / 2**e * 8) / 8 * 2**e * s

O = attn(K, V)
rel = lambda o: np.linalg.norm(o - O) / np.linalg.norm(O)
print(f"FP8（K、V 各一个 scale）：注意力输出误差 {rel(attn(fp8(K), fp8(V))):.2%}")
print(f"INT8，K、V 都每个 token 一个 scale：{rel(attn(q(K, 1, 127), q(V, 1, 127))):.2%}")
print(f"INT4，K、V 都每个 token 一个 scale：{rel(attn(q(K, 1), q(V, 1))):.2%}")
```

```
print(f"INT4, K 每个通道一个 scale、V 每个 token 一个 scale: {rel(attn(q(K, 0), q(V,
1))):.2%}")
```

输出：

```
精度 | 每个 token | 池子能放的 token | 4096 token 的请求 | 满载时 TPOT | 吞吐
BF16 |  128.0 KiB |   403,865 |   98 |  20.5 ms |   4,781 token/s
FP8 |   64.0 KiB |   807,731 |  197 |  20.6 ms |   9,573 token/s
INT4（每 128 个一个 FP16 scale） |   33.0 KiB | 1,566,508 |  382 |  20.6 ms |  18,564
token/s
FP8（K、V 各一个 scale）：注意力输出误差 3.87%
INT8，K、V 都每个 token 一个 scale：5.00%
INT4，K、V 都每个 token 一个 scale：64.71%
INT4，K 每个通道一个 scale、V 每个 token 一个 scale：18.84%
```

表格是池子的容量和满载时的速度：FP8 的并发和吞吐是 BF16 的两倍，TPOT 几乎不变。下面四行是注意力输出的误差：FP8 3.87%，INT8 5.00%；INT4 时 K 按 token 量化误差高达 64.71%，改成按通道量化降到 18.84%。

容易混淆的地方

1. **KV 量化不让单个请求变快多少。**并发低时，decode 主要在读权重，KV 小一半对 TPOT 影响不大；它的收益在于同样的显存能服务更多请求、支持更长的上下文。
2. **K 和 V 应该区别对待。**K 有离群通道，适合按通道分组；V 适合按 token 分组。用同一种方式处理，低位宽时会损失很多。
3. **FP8 的 KV 不需要 FP8 的计算单元。**注意力读进 FP8 的 KV 后反量化成 BF16 计算，所以在不支持 FP8 计算的 GPU 上，FP8 KV Cache 同样能省显存和带宽。

回顾

高并发、长上下文时，KV Cache 往往比权重还大，而且 decode 每一步都要读一遍。把 KV 存成 FP8，每个 token 从 128 KiB 变成 64 KiB，49.3 GiB 的池子能放的 4096 token 请求从 98 个变成 197 个；满载时 decode 主要在读 KV，TPOT 几乎不变（20.5 对 20.6 ms），吞吐从 4781 翻倍到 9573 token/s。精度上，K 有固定的离群通道，V 比较均匀：INT4 时 K 按 token 量化误差高达 64.71%，按通道量化降到 18.84%。FP8 KV 已经是常规选项，更低的位宽还需要更多技巧。

量化让每个 token 的 KV 小了一半。可上下文一长，减半也不够：128K 的上下文，FP8 的 KV 也有 8 GiB。生成下一个 token 时，真的需要前面每一个 token 的 K、V 吗？下一讲：**KV Cache 压缩与驱逐**。

第95讲 KV Cache 压缩与驱逐

选读

42 长上下文技术　69 KV Cache　71 显存账本　94 KV Cache 量化

上一讲把 KV Cache 存成 FP8，每个 token 的缓存从 128 KiB 减到 64 KiB，并发翻倍。可上下文一长，减半也不够看：Llama-3-8B 的一条 128K 上下文的请求，BF16 的 KV 是 16 GiB，FP8 也有 8 GiB，一张 80 GB 的卡放不下几条。

量化是让每个 token 存得更省；另一条路是**不存所有的 token**。生成下一个 token 时，它真的需要前面 13 万个 token 的 K、V 吗？如果注意力只集中在其中一小部分上，其余的能不能扔掉？这一讲用 Qwen3-0.6B 做一次真实的实验，看几种“扔法”的效果。

先算一下能省多少

```
per_token = 2 * 32 * 8 * 128 * 2            # Llama-3-8B：K、V × 32 层 × 8 个 KV 头 × 128 维 × 2 字节
print(f"每个 token {per_token / 1024:.0f} KiB")
for ctx in (8_192, 32_768, 131_072):
    print(f"上下文 {ctx:>7,}：一条请求的 KV {ctx * per_token / 2**30:5.1f} GiB；只留 4096 个：{4096 * per_token / 2**30:.2f} GiB，是原来的 {4096 / ctx:.1%}")
import math
for d in (0.539, 0.134, 0.013):
    print(f"损失多 {d:.3f}，困惑度变成原来的 {math.exp(d):.2f} 倍")
```

```
每个 token 128 KiB
上下文   8,192：一条请求的 KV   1.0 GiB；只留 4096 个：0.50 GiB，是原来的 50.0%
上下文  32,768：一条请求的 KV   4.0 GiB；只留 4096 个：0.50 GiB，是原来的 12.5%
上下文 131,072：一条请求的 KV  16.0 GiB；只留 4096 个：0.50 GiB，是原来的 3.1%
损失多 0.539，困惑度变成原来的 1.71 倍
损失多 0.134，困惑度变成原来的 1.14 倍
损失多 0.013，困惑度变成原来的 1.01 倍
```

只留 4096 个 token 的 KV，一条 128K 请求的缓存从 16 GiB 降到 0.5 GiB，是原来的 3%。缓存的大小不再随上下文增长，decode 每一步读的 KV 也只有这么多（第 71 讲：长上下文时，decode 的时间大半花在读 KV 上）。代价是什么，要看扔掉的是哪些。

注意力集中在哪里

实验是这样的：取一段 512 个 token 的讲义，前 448 个当作“已经在缓存里的上下文”，后 64 个当作“接下来要生成的部分”（尾巴）。先完整地跑一遍，看尾巴里的每个 token 把注意力分给了上下文的哪些位置；然后按几种策略只保留上下文里的 B 个位置，再算尾巴每个 token 的预测损失（负对数似然，第 16 讲），和全部保留时比较。损失多得越少，说明扔掉的东西越不重要。

```
$ .venv/bin/python bench/kv_evict.py                                    ⇆ 对照

Qwen3-0.6B，512 个 token：前 448 个是上下文，评估后 64 个。注意力 (28, 16, 512, 512)
尾巴的注意力里，落在上下文前 4 个 token 上的比例：63.5%（它们只占上下文的 0.9%）
落在上下文最近 64 个 token 上的比例：15.2%

不驱逐（448 个全留）：尾巴的平均损失 3.316
缓存预算 B | 只留最近 B 个 | 汇聚点 + 最近 | H2O | SnapKV | 随机 B 个
32 | +4.492（丢了第 0 个） | +0.539 | +0.570 | +0.504 | +0.878
64 | +3.528（丢了第 0 个） | +0.312 | +0.386 | +0.342 | +0.753
128 | +3.414（丢了第 0 个） | +0.134 | +0.118 | +0.151 | +4.723（丢了第 0 个）
256 | +2.814（丢了第 0 个） | +0.096 | +0.013 | +0.034 | +0.447
（表里是比“不驱逐”多出来的损失，越小越好；H2O、SnapKV 一半预算给最近的 token，一半按分数挑）
```

图 95-1　Qwen3-0.6B 上的驱逐实验：开头 4 个 token 独自收了尾巴 63.5% 的注意力；只留最近的 token、丢掉开头，损失暴涨；留下开头再加最近的，32 个就够用。

第一个数最惊人：**尾巴的注意力里，63.5% 落在上下文的前 4 个 token 上**，而它们只占 448 个位置的 0.9%。最近的 64 个 token 拿到 15.2%，剩下的 380 个位置分剩下的约五分之一。

第 42 讲讲过这个现象，叫**注意力汇聚点**（attention sink）：softmax 要求每一行的权重加起来是 1，很多时候当前 token 并不需要看前面的什么内容，模型就学会了把“用不掉”的注意力倒给开头那几个 token。它们本身的内容不重要，可它们的 K、V 承担了“吸收多余注意力”的作用。

四种扔法

只留最近 B 个。最直接的想法：越近的越重要，像滑动窗口一样只留最后 B 个。结果是灾难：B = 32 时损失多 4.49，B = 256（留了一半多）还多 2.81。原因就是丢了开头。汇聚点没了，本该倒给它们的注意力只能分给别的 token，每一层的输出都偏了。

汇聚点 + 最近（StreamingLLM）。永远保留开头 4 个，其余预算给最近的 token。B = 32 时损失只多 0.54，B = 128 时 0.13。只是多留了 4 个 token，效果天差地别。

H2O（heavy-hitter oracle）。不按位置，按“谁收到的注意力多”来留：把上下文里每个位置从所有查询那里收到的注意力加起来，分数高的就是“重击者”（heavy hitter）。预算的一半给最近的 token，另一半给分数最高的。开头的汇聚点分数最高，自然被留下了。B = 256 时损失只多 0.013，几乎无损。

SnapKV。只用上下文最后 32 个查询（观察窗口）的注意力来打分：离生成最近的那些查询看的地方，最能预测接下来要看的地方。再做一次池化，选中一个位置时把它的邻居也带上，保住完整的词。B = 32 时损失最小（0.50），B = 256 时 0.034。

随机作为对照：B = 128 时损失多 4.72，因为随机抽的 128 个里恰好没有第 0 个 token；B = 32、64、256 时抽中了第 0 个，损失就小得多。这个“意外”很说明问题：**汇聚点比其他所有位置加起来都重要**。

下面的演示画出了上下文每个位置收到的注意力（对数刻度），强调色是每种策略留下的位置：

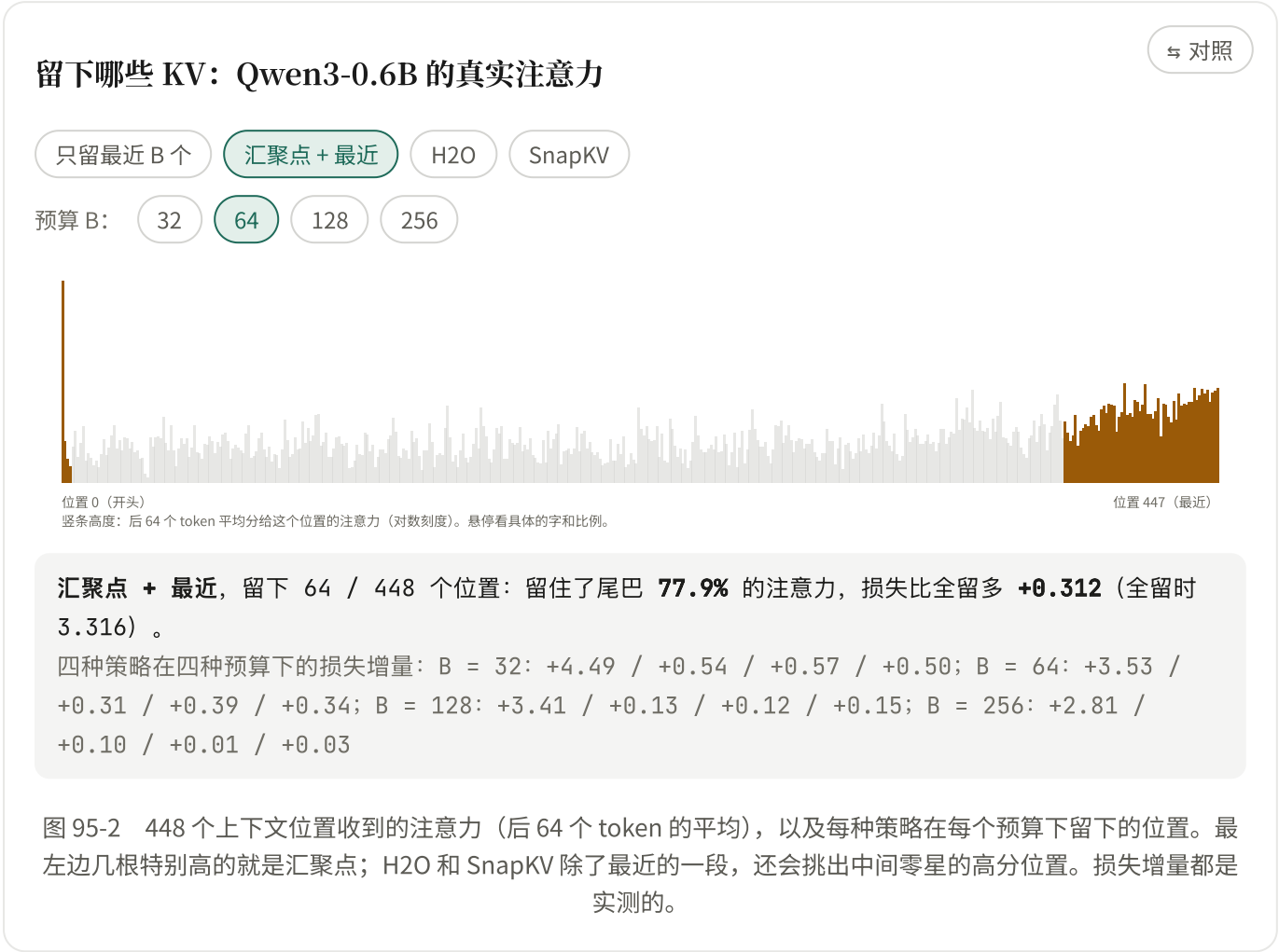


图 95-2　448 个上下文位置收到的注意力（后 64 个 token 的平均），以及每种策略在每个预算下留下的位置。最左边几根特别高的就是汇聚点；H2O 和 SnapKV 除了最近的一段，还会挑出中间零星的高分位置。损失增量都是实测的。

几个损失增量换成困惑度更直观：多 0.539 是困惑度变成 1.71 倍，多 0.134 是 1.14 倍，多 0.013 只是 1.01 倍。448 个只留 256 个时，H2O 几乎没有损失；只留 32 个（7%）时，最好的策略也让困惑度高了七成。

要点

注意力高度集中：开头几个“汇聚点”收走大部分注意力，最近的 token 次之。驱逐时汇聚点必须保留；在此之上，按累计注意力（H2O）或观察窗口的注意力（SnapKV）挑重要的 token，比只按位置留更好。

这个实验的局限

这只是一个小实验，有几处比真实方法更苛刻，也有几处更宽松：

- **所有层、所有头共用一份保留名单**。真实的 H2O、SnapKV 是每层、每个 KV 头各选各的，不同的头关注的地方不同。我们用一份名单（四维注意力掩码实现）是为了能直接算出真实的损失，代价是每个头都只能看同一批位置。
- **H2O 是简化版**。真实的 H2O 在生成的每一步都更新累计分数、逐步驱逐；我们只用上下文内部的注意力打一次分。
- **只测了 64 个 token 的尾巴、一段讲义**。驱逐的真正风险在于：**扔掉的 token 再也找不回来**。如果后面的问题恰好要用到上下文中间某个早被扔掉的细节（比如“大海捞针”测试里藏的那句话），任何策略都救不了。语言建模的损失看起来很好，不代表长文问答也好。

到了真实系统里

- **和分页 KV 的冲突**。vLLM 的 PagedAttention 按块管理 KV（第 75 讲），一块 16 个 token，所有头一起分配。按头、按 token 驱逐会在块里留下一个个洞，块表和 kernel 都要改。所以主流引擎默认不做逐 token 的驱逐；更常见的是模型结构本身带滑动窗口（如 Mistral 7B 的滑动窗口注意力、Gemma 2 隔层使用的局部注意力），引擎为它专门管理。
- **SnapKV 更适合 prefill 很长、生成较短的场景**。比如长文档问答：读完提示后一次性压缩 KV，之后的 decode 只用压缩后的缓存。它只在 prefill 结束时选一次，和引擎的配合比 H2O 简单。
- **按层分配预算**。PyramidKV 等工作发现低层的注意力比较分散、高层比较集中，于是低层多留、高层少留，同样的总预算效果更好。

- **扔掉之外，还可以挪走**。把不常用的 KV 从显存挪到 CPU 内存或 SSD，需要时再读回来（第 113 讲）。这样没有信息损失，代价是读回来的时间。

容易混淆的地方

- **驱逐和量化可以叠加**。量化让每个 token 小一半，驱逐让 token 少几倍，两者独立。
- **汇聚点不是“重要的内容”**。它们通常是开头的特殊符号或第一个词，内容无关紧要；重要的是它们在注意力里的“位置”。换成别的 token 放在开头，模型一样把注意力倒给它。
- **驱逐不省 prefill 的计算**。prefill 时所有 token 都要参与注意力（SnapKV 正是用 prefill 的注意力来选），省的是之后 decode 的显存和读 KV 的时间。

回顾

只留 4096 个 token 的 KV，128K 上下文的缓存能从 16 GiB 降到 0.5 GiB。在 Qwen3-0.6B 上，尾巴 63.5% 的注意力落在开头 4 个 token 上：只留最近的 token、丢掉它们，留 256 个也要多 2.81 的损失；保留汇聚点再加最近的，32 个只多 0.54。H2O 按累计注意力挑、SnapKV 按观察窗口挑，B = 256 时损失只多 0.013、0.034。驱逐的代价是扔掉的信息找不回来，而且和分页 KV 的块管理有冲突，所以主流引擎默认不做，更多用在长文档读完再答的场景。

到这里，压缩讲的都是“每一步读得更少”。可 decode 还有一个更根本的限制：一步只能生成一个 token，每一步都要把权重完整地读一遍。能不能一步生成好几个 token？下一讲：**投机解码**。

第96讲 投机解码

24 自回归生成　68 Prefill 与 Decode　69 KV Cache　73 解码策略

自回归生成（第 24 讲）有一个根本的限制：一步只能生成一个 token，因为下一个 token 要等上一个出来才能算。而第 68 讲算过，小批量的 decode 一步几乎全花在读权重上，Llama-3-8B 一步约 4.83 ms，算力只用了不到 1%。

这里藏着一个机会。代码算了一下：让大模型一次前向处理 **1 个** token 约 4.83 ms，处理 **5 个**也是 4.83 ms，**32 个**还是 4.83 ms。因为读权重的时间是固定的，多处理几个 token 只是多做一点计算，而计算单元本来就闲着。

可是生成的时候，我们不知道接下来的 5 个 token 是什么，没法一次喂进去。**投机解码**（speculative decoding）的想法是：**先找一个便宜的办法把接下来几个 token 猜出来，再让大模型一次前向把它们全部验证一遍**。猜对了几个，就一步前进几个。

先猜，再验证

做法需要两个模型：要用的**大模型**（目标模型），和一个同词表的**小模型**（草稿模型），比如用 Llama-3.2-1B 给 Llama-3.1-70B 打草稿。每一轮：

1. **起草**：小模型照常自回归地生成 k 个 token，比如 k = 4。小模型很小，这 4 步加起来也只相当于大模型一步的零头。
2. **验证**：把这 4 个草稿 token 接在已有文本后面，让大模型做**一次**前向。因果注意力（第 27 讲）保证，这一次前向会同时给出 5 个位置的预测：草稿第 1 个 token 之前那个位置应该接什么、第 1 个之后应该接什么……一直到第 4 个之后。
3. **对答案**：从前往后比较。以贪心解码（第 73 讲）为例：草稿第 1 个 token 和大模型在那个位置的预测一样，就接受，看下一个；遇到第一个不一样的，就停下，把这个位置换成大模型的预测，后面的草稿全部作废。如果 4 个全对，大模型在最后一个位置的预测也白送一个。

所以每一轮至少前进 1 个 token（最坏情况：第一个就猜错，用大模型的预测，和不用投机解码一样），最多前进 k + 1 个。被作废的草稿在 KV Cache 里留下的那几个位置，回退掉就行。

结果和大模型自己一个个生成完全一样：每个留下来的 token，都是大模型在那个位置认可的。投机解码不改变输出，只改变速度。

一轮能前进几个

设小模型每个草稿 token 被接受的概率是 α（**接受率**），并且假设各个位置相互独立。一轮前进的 token 数：前面连续接受了几个，再加上大模型给的 1 个。第 1 个被接受的概率 α，前 2 个都被接受 α^2……平均下来

$$E = 1 + \alpha + \alpha^2 + \cdots + \alpha\text{^}k = (1 - \alpha\text{^}(k+1)) \div (1 - \alpha)$$

$\alpha = 0.8$、$k = 4$ 时，$E = 3.362$：一轮平均前进 3.36 个 token。代码模拟了 10 万轮，平均 3.361 个，和公式一致。

一轮的耗时是大模型一步，加上小模型 k 步。设小模型一步的耗时是大模型的 c 倍，一轮就是 1 + k·c 个大模型步。所以

$$\text{加速比} = E \div (1 + k \cdot c)$$

$c = 5\%$ 时，代码列出了不同 α 和 k 的加速比：

- $\alpha = 0.6$：$k = 4$ 最好，1.92 倍。猜得不准，多猜也是浪费。
- $\alpha = 0.8$：$k = 4$ 时 2.80 倍，$k = 8$ 时达到 3.09 倍，再往后反而下降。
- $\alpha = 0.9$：k 越大越好，$k = 10$ 时 4.57 倍。

接受率是关键：小模型和大模型越“像”，α 越高，能放心地多猜几个。k 不是越大越好：猜得越多，后面的 token 被接受的概率越小（要前面全对），而小模型的成本却是线性增长的。

下面的演示模拟了几轮草稿和验证，绿色是被接受的草稿，红色是第一个猜错的位置，蓝色是大模型给出的 token；下面是加速比随 k 的曲线。调一调 α、k 和小模型的成本。

⇆ 对照

小模型先猜，大模型一次验证

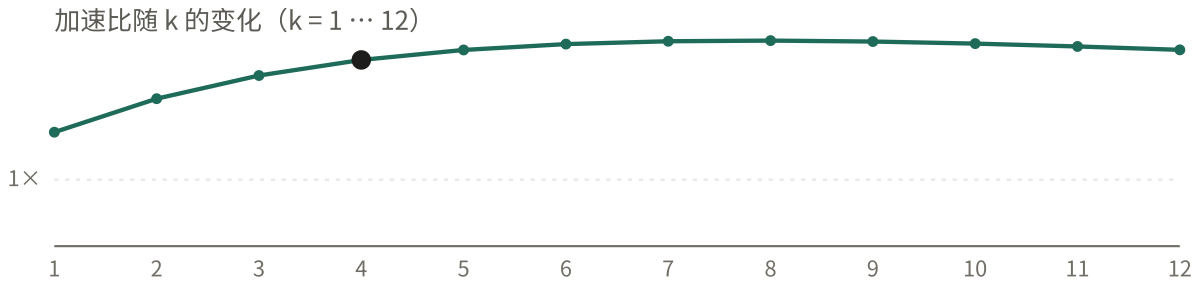
接受率 α = **0.80**

每轮猜 k = **4** 个

小模型一步 = 大模型的 **5%**

第 1 轮	投	机	×	·	解	+3
第 2 轮	码	先	让	×	小	+4
第 3 轮	模	型	猜	几	个	+5
第 4 轮	×	·	·	·	词	+1
第 5 轮	，	再	让	×	大	+4
第 6 轮	模	×	·	·	型	+2
第 7 轮	一	次	把	它	们	+5
第 8 轮	都	验	×	·	证	+3

已生成：投机解码先让小模型猜几个词，再让大模型一次把它们都验证



每轮期望产出 `E = (1 - 0.80^5) / (1 - 0.80) = 3.36` 个；一轮耗时 `= 1 + 4 × 0.05 = 1.20` 个大模型步

加速比 `= 3.36 ÷ 1.20 =` **2.80×**（这组参数下最好的 `k = 8`，3.09×）。上面 8 轮实际产出 27 个，平均 3.38 个/轮。

绿色：小模型猜对、被接受；红色 ✗：第一个猜错的位置，后面的（灰点）全部作废；蓝色：大模型在这个位置自己给出的 token——所以每轮至少前进 1 个。

图 96-1　投机解码。每一轮小模型猜 k 个，大模型一次验证：从第一个猜错的位置截断，并补上自己的 token，所以每轮前进 1 到 k + 1 个。下图是加速比随 k 的变化，α 越高，最好的 k 越大。

要点

decode 卡在读权重上，大模型一次前向处理几个 token 和处理 1 个几乎一样快。投机解码让小模型先猜 k 个 token，大模型一次前向验证，接受到第一个错误为止，并在那里补上自己的预测。每

轮期望前进 (1 − α^(k+1)) ÷ (1 − α) 个，加速比 E ÷ (1 + k · c)。输出与大模型单独生成完全一样。

采样时也不改变结果：拒绝采样

贪心解码时，“一样就接受”很直观。可如果用温度、Top-p 采样（第 73 讲），大模型每个位置给出的是一个概率分布 p，小模型给出的是 q，怎么判断“接受”，又保证最终的分布不变?

投机解码的论文（2022–2023 年，Google 和 DeepMind 分别独立提出）给出了一个精巧的办法，叫**拒绝采样**：

1. 小模型按 q 抽出草稿 token x。
2. 以概率 min(1, p(x) / q(x)) 接受它。大模型也觉得 x 很可能（p(x) ⩾ q(x)），就一定接受；大模型觉得 x 没那么可能，就按比例接受。
3. 如果拒绝，从分布 max(0, p − q)（归一化后）里重新抽一个，作为这个位置的 token。

可以证明，这样得到的 token 的分布**恰好是 p**。代码验证了一下：大模型 p = [0.5, 0.3, 0.15, 0.05]，小模型 q 是均匀的 [0.25, 0.25, 0.25, 0.25]，抽 20 万次，输出的分布是 [0.5, 0.3, 0.15, 0.05]，和 p 一模一样。接受率 0.699，和理论值 Σ min(p, q) = 0.700 相符。

所以 α 本质上是两个分布的重合程度：小模型越像大模型，重合越多，接受率越高。

到了大模型里

什么时候有用。投机解码的前提是“大模型一次处理多个 token 几乎不多花时间”，也就是 decode 卡在带宽上。批量越大，大模型的 decode 越接近计算受限（第 70 讲），多验证的那几个 token 就不再免费，加速比随之下降，大批量时甚至会变慢。所以它最适合延迟敏感、并发不高的场景，比如单个用户的对话、代码补全、智能体的长链条调用。

草稿从哪里来。单独的小模型要和大模型用同一个词表，还要占显存、要单独部署。更轻量的办法有：从输入里找现成的片段当草稿（比如修改代码时，输出大段重复输入，叫 n-gram 或提示查找，几乎零成本）；给大模型加几个小的预测头（下一讲的 Medusa、EAGLE）；或者模型训练时就带上多 token 预测（DeepSeek-V3 的 MTP 模块，可以直接用来打草稿）。

推理引擎的支持。vLLM、SGLang 都支持多种投机解码方式，和连续批处理、分页结合在一起。验证时被拒绝的草稿 token 占用的 KV 位置要回退，调度器也要为每个请求多预留 k 个位置。

接受率依赖任务。写代码、改写文本这类“可预测”的输出，α 很高；开放式创作、高温采样，α 低。实际部署要在真实流量上测加速比（第 82 讲）。

用代码复现

下面的代码先用第 68 讲的方法算出大模型一次前向处理不同个数 token 的耗时；再验证每轮期望产出的公式、算出不同 α、k 的加速比；最后用拒绝采样抽 20 万次，验证输出分布和大模型相同。

```python
import numpy as np

# 1. 验证 k+1 个 token 和 decode 1 个 token 差不多一样快（Llama-3-8B, H100, 上下文 1000）
N, NE, L, d, F, BW, KV = 8.03e9, 128256 * 4096, 32, 4096, 989e12, 3.35e12, 131072
def fwd_ms(n, ctx=1000):
    return max((2 * (N - NE) + 4 * L * d * ctx) * n / F, (N * 2 + KV * (ctx + n)) / BW) * 1e3
print(f"大模型前向：1 个 token {fwd_ms(1):.2f} ms，5 个 token {fwd_ms(5):.2f} ms，32 个 token {fwd_ms(32):.2f} ms")

# 2. 每猜一个 token 被接受的概率是 α，一次猜 k 个：一轮平均产出 E = (1 - α^(k+1)) / (1 - α) 个 token
def expected(a, k): return (1 - a ** (k + 1)) / (1 - a)
rng = np.random.default_rng(0)
def simulate(a, k, rounds=100_000):                    # 模拟：从第一个被拒绝的位置截断，再加上大模型自己给出的 1 个
    acc = rng.random((rounds, k)) < a
    n_ok = np.where(acc.all(1), k, acc.argmin(1))
    return (n_ok + 1).mean()
print(f"α = 0.8、k = 4：公式 {expected(0.8, 4):.3f} 个/轮，模拟 {simulate(0.8, 4):.3f} 个/轮")

# 3. 加速比：一轮的耗时 = 小模型猜 k 次（每次是大模型一步的 c 倍）+ 大模型验证一次
c = 0.05
print("加速比（小模型一步的耗时是大模型的 5%）：")
for a in [0.6, 0.8, 0.9]:
    sp = [expected(a, k) / (1 + k * c) for k in range(1, 11)]
    print(f"  α = {a}：k = 1..10 → {' '.join(f'{s:.2f}' for s in sp)}，最好的 k = {int(np.argmax(sp)) + 1}")

# 4. 拒绝采样：按 min(1, p/q) 接受小模型的猜测，拒绝时从 max(0, p - q) 里重新抽——结果的分布和大模型一模一样
p = np.array([0.5, 0.3, 0.15, 0.05])                   # 大模型的分布
q = np.array([0.25, 0.25, 0.25, 0.25])                 # 小模型的分布
n = 200_000
x = rng.choice(4, n, p=q)
keep = rng.random(n) < np.minimum(1, p[x] / q[x])
resid = np.maximum(p - q, 0); resid /= resid.sum()
out = np.where(keep, x, rng.choice(4, n, p=resid))
print(f"接受率 {keep.mean():.3f}（理论 Σ min(p, q) = {np.minimum(p, q).sum():.3f}）；输出的分布 {np.round(np.bincount(out) / n, 3).tolist()}，大模型 {p.tolist()}")
```

输出：

```
大模型前向：1 个 token 4.83 ms，5 个 token 4.83 ms，32 个 token 4.83 ms
α = 0.8、k = 4：公式 3.362 个/轮，模拟 3.361 个/轮
加速比（小模型一步的耗时是大模型的 5%）：
  α = 0.6：k = 1..10 → 1.52 1.78 1.89 1.92 1.91 1.87 1.82 1.77 1.71 1.66，最好的 k = 4
  α = 0.8：k = 1..10 → 1.71 2.22 2.57 2.80 2.95 3.04 3.08 3.09 3.08 3.05，最好的 k = 8
  α = 0.9：k = 1..10 → 1.81 2.46 2.99 3.41 3.75 4.01 4.22 4.38 4.49 4.57，最好的 k = 10
接受率 0.699（理论 Σ min(p, q) = 0.700）；输出的分布 [0.5, 0.3, 0.15, 0.05]，大模型 [0.5, 0.3, 0.15, 0.05]
```

第一行：1 个、5 个、32 个 token 的前向都约 4.83 ms。第二行：公式 3.362，模拟 3.361。接下来三行是加速比的表格。最后一行是拒绝采样的验证。

容易混淆的地方

1. **投机解码不降低质量。**无论贪心还是采样，输出的分布都和大模型单独生成完全一样（不考虑浮点误差）。它和量化、蒸馏这类“用质量换速度”的方法不同。
2. **它不减少计算量，反而增加了。**被拒绝的草稿白算了，小模型也要算。它是用闲着的算力换延迟，所以在算力本来就紧张的大批量场景里不划算。
3. **加速比不等于平均接受的 token 数。**E 是每轮前进的 token 数，还要除以一轮的耗时 1 + k·c；小模型越慢，抵消得越多。

回顾

小批量 decode 卡在读权重上，大模型一次前向处理 1 个和 32 个 token 都约 4.83 ms。投机解码让小模型先猜 k 个 token，大模型一次前向验证所有位置，接受到第一个错误为止，再补上自己的预测，所以每轮前进 1 到 k + 1 个，结果与大模型单独生成完全一样。每轮期望前进 E = (1 − α^(k+1)) ÷ (1 − α)，加速比 E ÷ (1 + k·c)：α = 0.8、k = 4、c = 5% 时约 2.80 倍。采样时用拒绝采样：以 min(1, p/q) 接受，拒绝时从 max(0, p − q) 重抽，输出分布恰好是 p。它最适合并发不高、对延迟敏感的场景。

单独的草稿模型要和大模型同词表、要占显存、要单独维护，而且它只看得到文字，看不到大模型内部已经算好的东西。能不能直接让大模型“顺便”多猜几个 token？下一讲：**EAGLE 与 Medusa**。

第97讲 EAGLE 与 Medusa

选读

27 因果掩码　33 输出层　96 投机解码

上一讲的投机解码要一个单独的草稿模型。它带来不少麻烦：要和大模型用同一个词表；要占显存、单独部署；很多大模型根本没有合适的“小弟”。更可惜的是，草稿模型从头开始读文字，大模型已经算好的那些丰富的内部表示，它一点也用不上。

这一讲的两种方法都换了一个思路：**不要单独的小模型，在大模型自己身上接几个轻量的预测头来打草稿**。Medusa 一次并行地猜后面好几个位置，EAGLE 在大模型的“特征”层面一步步地猜。它们还一起推广了一个重要的技巧：一次验证一整棵候选树。

Medusa：多长几个头

第 33 讲说过，大模型最后一层的隐藏状态，经过输出层（LM head）变成下一个 token 的概率。这个隐藏状态里，其实已经包含了不少关于“再往后几个 token”的信息。

Medusa（2024 年）在最后一层的隐藏状态上，额外接几个小小的**预测头**（一两层的小网络）：第 1 个头猜下一个 token 之后的那个（第 $t+2$ 个），第 2 个头猜第 $t+3$ 个，以此类推。原来的输出层照常预测第 $t+1$ 个。一次前向，就同时得到了后面好几个位置的猜测，不需要额外的模型，也不需要额外的前向。

训练这些头很便宜：大模型的参数冻结不动，只用一些数据训练这几个小头（Medusa-1）；也可以和大模型一起微调，猜得更准（Medusa-2）。

问题是，越往后的头猜得越不准：它要跳过中间的几个 token 直接猜，而中间那几个 token 到底是什么，它并不知道。我们假设四个头只看第 1 名时，猜对的概率分别是 0.60、0.45、0.35、0.28（示意数字），按上一讲的方法，一轮平均只前进 **1.99** 个 token。

一次验证一棵树

改进的办法是：**每个头不只取第 1 名，而是取前几名**。第 1 名不对的时候，正确答案常常在第 2 名。假设各个头的前 2 名里包含正确答案的概率是 0.78、0.62、0.52、0.44。

可是前 2 名组合起来就不是一条链了，而是一棵**树**：第 1 个位置有 a、b 两个候选，每个候选后面第 2 个位置又有两个候选……深度 3 的树一共 2 + 4 + 8 = 14 个候选 token。要一个个地验证，就失去了意义。

树形注意力（tree attention）让这 14 个候选一次前向全部验证完：把它们排成一排送进大模型，但不用普通的因果掩码（第 27 讲），而是用一个特殊的掩码，**每个候选只看得见前文和自己在树上的祖先**。比如节点“ab”（第 1 个位置取第 1 名、第 2 个位置取第 2 名）能看见“a”，看不见“b”，也看不见同一层的“aa”。这样，每个候选得到的预测，和它所在的那条路径单独送进去时完全一样。

验证完后，从根开始，沿着大模型认可的方向往下走，能走多深就接受多少个。代码里的例子：大模型认可的路径是“第 1 个位置第 1 名、第 2 个位置第 2 名”，在树里走到“ab”，接受 2 个，这一轮前进 3 个；如果只用一条链（每层只取第 1 名），在第 2 个位置就停了，只前进 2 个。

按示意数字算：深度 2 的树（6 个候选）平均每轮前进 2.26 个，深度 3（14 个候选）2.52 个，深度 4（30 个候选）2.63 个。候选越多，收益越小，而验证的成本（大模型一次多处理多少个 token）越来越大。实际的实现会根据每个头各个名次的命中率，挑出一棵“性价比”最高的、不对称的树。

下面的演示画出了深度 3 的候选树和它的注意力掩码。点上面的按钮选大模型在每个位置认可的是第几名，看这一轮能前进几个；点树上的节点，看它在掩码里能看见谁。

一次验证一整棵候选树

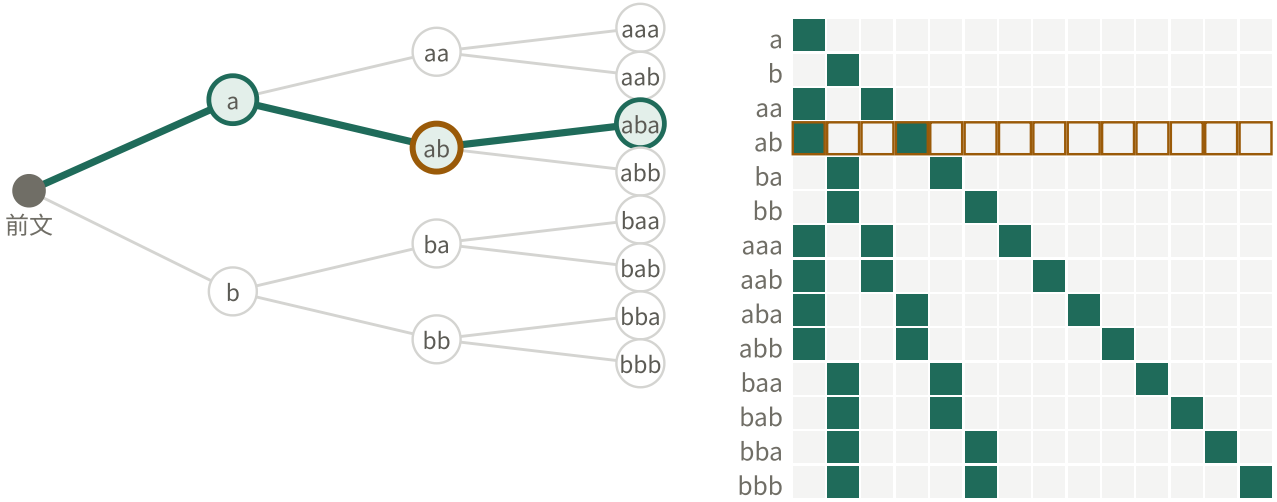
⇆ 对照

树（每层前 2 名） 链（每层只取第 1 名）

大模型在第 1 个位置认可： 第 1 名 第 2 名 都不是

大模型在第 2 个位置认可： 第 1 名 第 2 名 都不是

大模型在第 3 个位置认可： 第 1 名 第 2 名 都不是



14 个候选 token 一次送进大模型。选中的节点 **ab** 在注意力里只看得见前文 和 a（掩码里它那一行）。
大模型认可的路径：a → b → a。沿树走到 aba，接受 3 个，这一轮前进 **4** 个 token。
树里同时摆着每层的前 2 名，大模型认可第 2 名时也能接着走；代价是要一起验证 14 个候选。

图 97-1　树形验证。左：每层取前 2 名的候选树，绿色是被接受的路径；右：树形注意力掩码，每一行是一个候选，只看得见自己的祖先。切换到“链”，对比大模型认可第 2 名时的差别。

要点

Medusa 在大模型最后一层的隐藏状态上接几个预测头，一次前向同时猜后面好几个位置，不需要单独的草稿模型。每个头取前几名组成候选树，用“只看得见祖先”的树形注意力掩码，一次前向验证整棵树，沿大模型认可的路径接受最长的一段。

EAGLE：在特征层面一步步地猜

Medusa 的头是“跳着猜”的，第 3 个头要在不知道第 t + 1、t + 2 个 token 的情况下猜第 t + 3 个，所以越往后越不准。

EAGLE（2024 年）回到了一步一步地猜，但猜的不是 token，而是大模型的**特征**（最后一层、输出层之前的隐藏状态）。它的草稿模型只是一个很小的网络（大约一层 Transformer），输入是大模型当前位置的特征，加上刚刚确定的那个 token 的嵌入，输出是下一个位置的特征；再用大模型自己的输出层，把这个特征变成 token 的概率。然后把猜出的 token 和特征接回去，继续猜下一个。

这样做有两个好处：

- **特征比 token 好猜**。隐藏状态是连续的、平滑的，比离散的 token 更有规律，小网络更容易学。
- **知道上一步选了哪个 token**。把上一步的 token 一起喂进去，消除了“上一步其实有好几种可能”带来的不确定性。这正是 Medusa 的头做不到的。

论文报告，EAGLE 的接受率明显高于 Medusa。后续的 EAGLE-2 根据草稿模型自己的置信度**动态地**决定候选树的形状（有把握的分支多展开，没把握的少展开）；EAGLE-3 改用大模型好几层的特征，并改进了训练方式，加速比进一步提高。EAGLE 系列是目前推理引擎里最常用的投机解码方法之一。

训练时就学会多猜几个：MTP

还有一种做法是在**训练大模型的时候**就让它学会预测后面好几个 token，叫**多 token 预测**（MTP）。DeepSeek-V3 在主模型之外加了一个 MTP 模块，训练时用来提供更多的学习信号；推理时，这个模块正好可以拿来打草稿。DeepSeek-V3 的技术报告说，它对第二个 token 的预测接受率在 85% 到 90% 之间，生成速度提高到约 1.8 倍。

到了大模型里

推理引擎的支持。vLLM、SGLang 都支持 EAGLE（包括 EAGLE-3）、Medusa、MTP 和单独的草稿模型等多种方式，树形注意力也集成在它们的注意力 kernel 里。很多开源模型会同时发布配套的 EAGLE 草稿头。

和批大小的关系没有变。这些方法和上一讲一样，依赖“大模型一次多处理几个 token 几乎免费”。树越大，每轮要验证的 token 越多；批大小大时，这些额外的 token 不再免费，所以引擎会根据负载调小候选树，甚至关掉投机解码。

草稿头要和模型匹配。预测头或 EAGLE 的草稿网络是针对某一个大模型训练的，换了模型（哪怕只是微调过）就要重新训练，否则接受率会下降。

用代码复现

下面的代码按上面的示意数字，计算链和不同深度的树平均每轮前进的 token 数；构造一棵深度 2 的小树，打出它的树形注意力掩码；最后演示沿树接受最长路径。

```
import numpy as np
from itertools import product

# 1. Medusa：在大模型最后一层的隐藏状态上接 4 个小头，第 i 个头猜“往后第 i+1 个 token”。
#    假设各个头猜对的概率（示意数字）：只看第 1 名 / 看前 2 名
top1 = [0.60, 0.45, 0.35, 0.28]
top2 = [0.78, 0.62, 0.52, 0.44]
def accepted(ps):                                   # 一轮期望前进几个：1 +（前 1 个都对）+（前 2 个都对）+ …
    return 1 + sum(np.prod(ps[:i + 1]) for i in range(len(ps)))
print(f"一条链（每个头取第 1 名，4 个候选 token）：平均每轮前进 {accepted(top1):.2f} 个")
for depth in [2, 3, 4]:
    nodes = sum(2 ** i for i in range(1, depth + 1))
    print(f"一棵树（每个头取前 2 名，深度 {depth}，{nodes} 个候选 token）：平均每轮前进 {accepted(top2[:depth]):.2f} 个")

# 2. 树形注意力：把整棵树的候选 token 排成一排，一次前向验证。每个候选只能看见自己的祖先（和已有的前文）
paths = [p for depth in [1, 2] for p in product([0, 1], repeat=depth)]      # 深度 2、每层 2 个分支：6 个节点
names = {p: "".join("ab"[x] for x in p) for p in paths}
mask = np.array([[1 if q == p[:len(q)] else 0 for q in paths] for p in paths])
print("节点：", [names[p] for p in paths], "（a、b 是第 1 名、第 2 名；“ab”= 第一位取第 1 名、第二位取第 2 名）")
print("树形注意力掩码（行看列，1 = 看得见）：")
for p, row in zip(paths, mask):
    print(f"  {names[p]:>2s}  " + " ".join(map(str, row)))

# 3. 验证：大模型在每个位置的预测，从根开始沿着树往下走，能走多深就接受多少
truth = (0, 1)                                      # 假设大模型认可的是“第一位第 1 名、第二位第 2 名”
depth = next((d for d in range(2, 0, -1) if truth[:d] in names), 0)
print(f"大模型认可的路径是 {names[truth]}：在树里找到长度 {depth} 的路径，这一轮前进 {depth + 1} 个 token（链只取第 1 名，只能接受 1 个，前进 2 个）")
```

输出：

```
一条链（每个头取第 1 名，4 个候选 token）：平均每轮前进 1.99 个
一棵树（每个头取前 2 名，深度 2，6 个候选 token）：平均每轮前进 2.26 个
一棵树（每个头取前 2 名，深度 3，14 个候选 token）：平均每轮前进 2.52 个
一棵树（每个头取前 2 名，深度 4，30 个候选 token）：平均每轮前进 2.63 个
节点： ['a', 'b', 'aa', 'ab', 'ba', 'bb']  （a、b 是第 1 名、第 2 名；“ab”= 第一位取第 1 名、第二位取第 2 名）
树形注意力掩码（行看列，1 = 看得见）：
    a  1 0 0 0 0 0
    b  0 1 0 0 0 0
   aa  1 0 1 0 0 0
   ab  1 0 0 1 0 0
   ba  0 1 0 0 1 0
   bb  0 1 0 0 0 1
大模型认可的路径是 ab：在树里找到长度 2 的路径，这一轮前进 3 个 token（链只取第 1 名，只能接受 1 个，前进 2 个）
```

前四行：一条链 1.99 个，树的深度 2、3、4 时分别是 2.26、2.52、2.63 个，候选数是 6、14、30。中间是 6 个节点的树形注意力掩码：比如“ab”这一行，只有“a”和它自己是 1。最后一行是验证的例子：树能接受到“ab”，比链多前进 1 个 token。

容易混淆的地方

1. **这些方法也不改变输出**。和上一讲一样，最终留下的 token 都经过大模型验证，结果和大模型单独生成相同（采样时配合拒绝采样或类似的规则）。
2. **Medusa 的头和 MoE 的专家不是一回事**。预测头是额外接在最后的小网络，用来猜后面的 token；MoE 的专家（第 41 讲）是模型主体的一部分。
3. **树形注意力不是新的注意力机制**。它只是换了一个掩码，让多条候选路径挤在一次前向里而互不干扰，计算方式和普通的注意力一样。

回顾

单独的草稿模型要同词表、占显存，还用不上大模型的内部表示。Medusa 在大模型最后一层的隐藏状态上接几个预测头，一次前向同时猜后面好几个位置，但越往后越不准。每个头取前几名组成候选树，用“只看得见祖先”的树形注意力掩码一次验证整棵树，沿大模型认可的路径接受最长的一段：示意数字下，链平均每轮前进 1.99 个，深度 3 的树 2.52 个。EAGLE 用一个很小的网络在特征层面一步步地猜，并把上一步的 token 喂回去，接受率更高；DeepSeek-V3 的 MTP 模块在训练时就学会了多猜一个，推理时直接用来打草稿。

量化、投机解码都在不改模型大小的前提下加速。另一条路是直接得到一个更小的模型：让小模型向大模型学习，或者把大模型里不重要的部分剪掉。下一讲：**蒸馏与剪枝**。

第98讲 蒸馏与剪枝

选读

16 损失函数　47 监督微调 SFT　91 权重量化：GPTQ 与 AWQ

量化让每个参数占的比特变少，投机解码让每一步多出几个 token，它们都不改变模型有多少个参数。另一条路是直接得到一个**更小的模型**：参数少了，显存、计算、带宽全都跟着省。

从头训练一个小模型当然可以，但效果往往不如同样大小、却“向大模型学过”的模型。这一讲讲两种办法：**蒸馏**（distillation），让小模型学习大模型的输出；**剪枝**（pruning），把大模型里不重要的部分剪掉。实际中它们经常一起用：先剪，再蒸馏，把剪掉造成的损失补回来。

蒸馏：学老师的“分寸”

第 16 讲的交叉熵损失，训练时只告诉模型“正确答案是哪个”。比如“一只可爱的___”，标准答案是“猫”，模型只知道“猫”对、其他都错。

一个训练好的大模型（**老师**）给出的却是一个分布。代码里的示意：老师觉得“猫”0.744、“狗”0.166、“老虎”0.082，“汽车”“桌子”“苹果”都只有千分之几。这个分布里藏着标准答案没有的信息：狗和老虎也说得通，汽车完全不行。Hinton 等人在 2015 年把这叫作“暗知识”（dark knowledge）。**蒸馏**就是让小模型（**学生**）去拟合老师的整个分布，而不只是那一个标准答案。

为了让这些小概率的信息更明显，蒸馏时常把老师和学生的 logits 都除以一个**温度** T（第 73 讲）再做 softmax。T = 4 时，老师的分布变成“猫”0.336、“狗”0.231、“老虎”0.194，“汽车”等也有 0.08 左右，各个词之间的相对关系一目了然。损失是两个分布之间的 KL 散度 KL(p老师 ‖ p学生)。

两种训练信号的差别，看梯度最清楚。学生现在的打分把“老虎”排得比“狗”高：

- **只用标准答案**：梯度是 p − y = [−0.526, 0.064, 0.288, 0.106, 0.039, 0.029]。它只说“提高猫，压低其他所有词”，“狗”和“汽车”一起被压低。
- **蒸馏**（T = 4）：梯度是 (p学生 − p老师) ÷ T = [−0.024, −0.022, 0.004, 0.02, 0.012, 0.01]。它说“提高猫，**也提高狗**；稍微压低老虎；压低汽车、桌子、苹果”。

每个样本都带着这么丰富的信号，学生用更少的数据就能学得更好。

大模型的蒸馏有两种做法。一种是上面这样按 token 拟合老师的分布（logit 蒸馏），要求学生和老师用同一个词表，训练时还要跑老师的前向。另一种更简单：让老师生成大量的回答，学生直接在这些回答上做 SFT（第 47 讲），叫**数据蒸馏**。DeepSeek 用 R1 生成的约 80 万条数据，微调出了一系列 Qwen 和 Llama 的小模型，让它们也学会了长思维链（第 49 讲）。Meta 的 Llama 3.2 的 1B、3B 模型，在预训练时也用到了更大的 Llama 3.1 模型的输出。

剪枝：哪些权重可以不要

剪枝是把一部分权重设成 0，或者整块地去掉。核心问题是：剪哪些？

最朴素的想法是剪掉**绝对值最小**的权重。可第 91 讲说过，权重的重要性还取决于它乘的激活：激活大的输入通道，对应的权重哪怕很小，也可能对输出影响很大。2023 年的 **Wanda** 方法就用 $|w| \times \|x_j\|$（权重大小乘以对应输入通道激活的大小）作为重要性，只需要一小批校准数据，不用再训练。

代码比较了一个 64 × 64 的层，校准激活里有 4 个离群通道，每种方法都剪掉一半权重：

- **按 $|w|$ 剪**：输出误差 27.12%。离群通道那几行的权重和别的行差不多大，被剪掉了不少。
- **按 $|w| \cdot \|x\|$ 剪（Wanda）**：10.72%。离群通道的权重被保住了。

剪了能变快吗：稀疏的结构

把一半权重设成 0，模型就快一倍吗？不一定。这取决于 0 的分布方式：

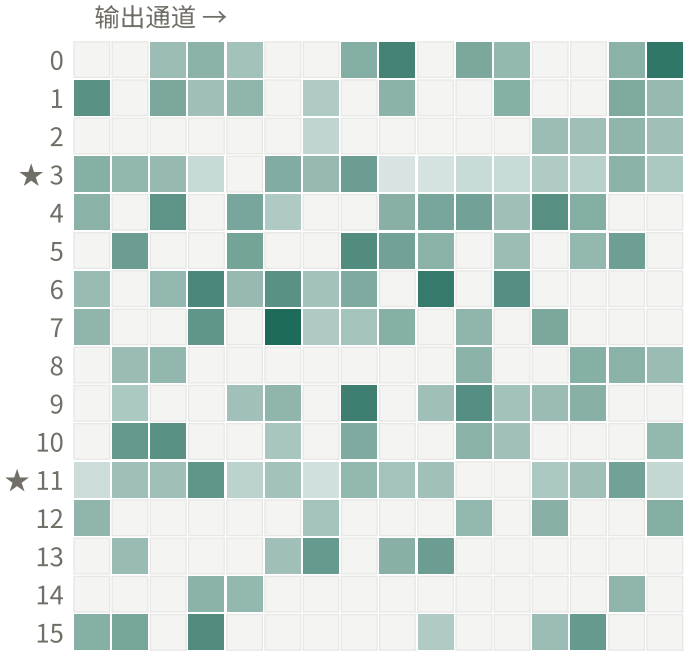
- **非结构化**：0 零散地分布在矩阵里。矩阵的形状没变，普通的 GPU 矩阵乘法照样要把这些 0 读进来、乘一遍，几乎不会变快。除非稀疏到 90% 以上，用专门的稀疏格式和 kernel 才有收益。
- **半结构化 2:4**：沿输入方向，每 4 个连续的权重里恰好保留 2 个。这种规则的稀疏，NVIDIA 从 Ampere（A100）开始有专门的**稀疏 Tensor Core** 支持，只存非零值和它们的位置，理论上矩阵乘法快一倍、权重小将近一半。代价是约束更强，误差 13.92%，比非结构化大一些。
- **结构化**：整行、整列、整个注意力头，甚至整层地去掉。矩阵直接变小，任何硬件都能变快、变省。但一次去掉的东西多，误差也最大：去掉一半输入通道，误差 23.43%。通常要配合再训练（最好是蒸馏）来恢复。

下面的演示是一个 16 × 16 的缩小版。切换剪枝方式，看哪些权重被剪掉、误差多少、硬件能不能因此变快。

剪掉一半权重：剪哪些、怎么剪

⇆ 对照

原始 | 非结构化：按 |w| | 非结构化：按 |w| · ‖x‖ | 2:4 | 结构化：去掉整行



保留 128 / 256 个权重，输出相对误差 11.12%。
同样零散，速度上和按 |w| 剪一样没有好处；但它保住了离群通道（★）那两行的权重，误差小得多。颜色深浅是 |w| 的大小，灰色是被剪掉的；★ 是激活大 10 倍的输入通道。

图 98-1　剪掉一半权重的几种方式。颜色深浅是权重的大小，灰色是剪掉的，★ 是激活大 10 倍的输入通道。按 |w| · ‖x‖ 剪能保住这些通道；2:4 和结构化剪枝的形状规则，硬件才能真正加速。

要点

蒸馏让小模型拟合大模型输出的整个分布，学到“哪些答案也说得通”，比只学标准答案的信号丰富得多；大模型常用的是数据蒸馏，让老师生成训练数据。剪枝去掉不重要的权重，重要性要结合激活来判断（|w| · ‖x‖）；只有 2:4 半结构化或整块的结构化剪枝，才能在硬件上真正变快，结构化剪枝通常要再用蒸馏恢复效果。

先剪，再蒸馏

把两者结合起来，就是一条得到小模型的实用路线：

1. 从一个大模型出发，用校准数据估计每个注意力头、每个 FFN 神经元、每一层的重要性；

2. 按重要性做结构化剪枝，比如把 FFN 的宽度砍掉一部分，或者去掉若干层，得到一个更小的模型；
3. 用原来的大模型当老师，对剪完的模型做蒸馏，只用很少的数据（远少于从头训练）就能恢复大部分效果。

NVIDIA 的 Minitron 系列就是这样做的，比如从 Llama 3.1 8B 剪枝、蒸馏得到 4B 的模型。它比从头训练同样大小的模型省下了大量训练算力。

到了大模型里

蒸馏是小模型的标配。今天很多小尺寸的开源模型，都在训练中用到了同家族大模型的输出，或者大模型生成的数据。对推理来说，这意味着在同样的延迟和成本下，能用上一个更聪明的小模型。

剪枝在推理部署中用得不如量化多。非结构化剪枝难以加速；2:4 稀疏的实际加速通常不到理论的两倍，效果损失也比同等压缩的量化更明显；结构化剪枝需要再训练，更像是“训练一个新模型”。所以部署现成的模型时，量化（第 90–94 讲）是首选；剪枝和蒸馏更多地用在**模型厂商制作小模型**的环节。

和 MoE 的关系。MoE（第 41 讲）每个 token 只激活一部分专家，可以看作一种“动态的结构化稀疏”。也有工作对 MoE 剪掉不常用的专家，或者把专家合并，以降低显存。

用代码复现

下面的代码先用示意的老师和学生打分，对比只用标准答案和蒸馏时的梯度；再在一个有离群通道的线性层上，比较几种剪掉一半权重的方式的输出误差。

```python
import numpy as np

# 1. 蒸馏：老师的输出分布里，除了“正确答案”，还有“哪些词也说得通”的信息
words = ["猫", "狗", "老虎", "汽车", "桌子", "苹果"]
t_logits = np.array([6.0, 4.5, 3.8, 0.5, 0.2, 0.1])    # 老师看到“一只可爱的___”时的打分（示意）
def softmax(z, T=1.0): e = np.exp((z - z.max()) / T); return e / e.sum()
for T in [1, 4]:
    print(f"温度 T = {T}：老师的分布 " + "  ".join(f"{w} {p:.3f}" for w, p in zip(words, softmax(t_logits, T))))
s_logits = np.array([3.0, 1.0, 2.5, 1.5, 0.5, 0.2])    # 学生现在的打分
T = 4
pt, ps = softmax(t_logits, T), softmax(s_logits, T)
print(f"只用标准答案“猫”训练：损失 {-np.log(softmax(s_logits)[0]):.3f}，梯度 (p - y) = {np.round(softmax(s_logits) - np.eye(6)[0], 3).tolist()}")
print(f"蒸馏 (T = 4)：KL(老师 ‖ 学生) = {np.sum(pt * np.log(pt / ps)):.3f}，梯度 (p学生 - p老师) / T = {np.round((ps - pt) / T, 3).tolist()}")

# 2. 剪枝：一个线性层（64 → 64），校准激活里有 4 个离群通道。剪掉一半的权重，看输出误差
rng = np.random.default_rng(0)
d = 64
X = rng.standard_normal((256, d)); X[:, [3, 17, 40, 58]] *= 10
W = rng.standard_normal((d, d)) * 0.05                  # y = X · W，W 的第 j 行对应第 j 个输入通道
err = lambda Wp: np.linalg.norm(X @ W - X @ Wp) / np.linalg.norm(X @ W)

def prune_by(score, keep_mask_fn):
    return W * keep_mask_fn(score)
def unstructured(score, ratio=0.5):                     # 全局（每一列内）去掉分数最小的一半
    thr = np.quantile(score, ratio, axis=0, keepdims=True); return score > thr
def two_four(score):                                    # 2:4：沿输入方向每 4 个里保留分数最大的 2 个
    s = score.reshape(d // 4, 4, d); r = s.argsort(1).argsort(1)
    return (r >= 2).reshape(d, d)

mag = np.abs(W)                                         # 只看权重大小
wanda = np.abs(W) * np.linalg.norm(X, axis=0)[:, None]  # Wanda：权重大小 × 这个输入通道激活的大小
print(f"非结构化 50%，按 |w|：输出误差 {err(prune_by(mag, unstructured)):.2%}")
print(f"非结构化 50%，按 |w|·‖x‖ (Wanda)：输出误差 {err(prune_by(wanda, unstructured)):.2%}")
print(f"2:4 半结构化，按 |w|·‖x‖：输出误差 {err(prune_by(wanda, two_four)):.2%}")
ch = np.argsort(np.linalg.norm(X, axis=0) * np.linalg.norm(W, axis=1))[: d // 2]
# 结构化：整行（整个输入通道）去掉
```

```
Wc = W.copy(); Wc[ch] = 0
print(f"结构化，去掉一半输入通道：输出误差 {err(Wc):.2%}（矩阵直接变成 32 × 64）")
```

输出：

```
温度 T = 1：老师的分布 猫 0.744  狗 0.166  老虎 0.082  汽车 0.003  桌子 0.002  苹果
0.002
温度 T = 4：老师的分布 猫 0.336  狗 0.231  老虎 0.194  汽车 0.085  桌子 0.079  苹果
0.077
只用标准答案“猫”训练：损失 0.746，梯度（p - y）= [-0.526, 0.064, 0.288, 0.106, 0.039,
0.029]
蒸馏（T = 4）：KL(老师 ‖ 学生) = 0.082，梯度（p学生 - p老师）/ T = [-0.024, -0.022,
0.004, 0.02, 0.012, 0.01]
非结构化 50%，按 |w|：输出误差 27.12%
非结构化 50%，按 |w|·‖x‖（Wanda）：输出误差 10.72%
2:4 半结构化，按 |w|·‖x‖：输出误差 13.92%
结构化，去掉一半输入通道：输出误差 23.43%（矩阵直接变成 32 × 64）
```

前两行是老师在 T = 1 和 T = 4 时的分布，温度高时小概率词的信息更明显。接着两行是梯度：只用标准答案时“狗”被压低（+0.064），蒸馏时“狗”被提高（−0.022）。最后四行是剪枝：按 |w| 27.12%，Wanda 10.72%，2:4 13.92%，结构化 23.43%。

容易混淆的地方

1. **蒸馏不是压缩一个已有的模型。**它训练的是另一个（通常从别处初始化的）小模型，只是用大模型当老师。剪枝才是在原模型上动刀。
2. **稀疏不等于快。**零散的 0 在普通的矩阵乘法里照样要算；只有硬件支持的稀疏格式（2:4）或者真正变小的矩阵（结构化）才能加速。
3. **剪枝后的误差可以靠训练补回来。**Wanda 这类方法追求“不再训练就能用”；结构化剪枝则默认要再训练，单看剪完那一刻的误差并不公平。

回顾

蒸馏让学生拟合老师的整个输出分布（常除以温度 T），学到“哪些答案也说得通”：只用标准答案时“狗”和“汽车”一起被压低，蒸馏时“狗”被提高、“汽车”被压低。大模型多用数据蒸馏，让老师生成训练数据。剪枝去掉不重要的权重，重要性要结合激活：剪掉一半时，按 |w|

输出误差 27.12%，按 $|w|\cdot\|x\|$ 10.72%。非结构化稀疏难以加速，2:4 半结构化有稀疏 Tensor Core 支持，结构化剪枝让矩阵真正变小，通常再用蒸馏恢复效果。部署现成模型首选量化，剪枝和蒸馏更多用在制作小模型上。

第 12 篇到这里结束，第三章“单机推理”也告一段落。我们一直在一张卡上想办法：显存装得下、算得够快。可 DeepSeek-V3 这样 6710 亿参数的模型，一张卡根本装不下；即使装得下，一张卡也撑不住大量的请求。模型和请求要分到多张卡、多台机器上，卡与卡之间就要不停地交换数据。下一讲进入第四章：**为什么需要多卡**。

AI INFRA · HANDBOOK

第 十三 篇

分布式与集合通信

收录 11 讲 · 第 99—109 讲

第99讲 为什么需要多卡

41 MoE 混合专家　68 Prefill 与 Decode　71 显存账本　90 量化基础

第三章的所有讨论都发生在一张卡上：Llama-3-8B 的权重 16 GB，一张 80 GB 的 H100 绰绰有余，剩下的空间放 KV Cache（第 71 讲）。可是今天常用的大模型，很多比它大一个甚至两个数量级。从这一讲开始的第四章，我们要把模型和请求分到多张卡、多台机器上。

这一讲先回答最基本的问题：**为什么一张卡不够？**原因有三个：放不下、算不快、扛不住。

放不下

先算权重。参数量乘以每个参数的字节数（第 90 讲）：

- **Llama-3-8B**：BF16 16 GB，FP8 8 GB。
- **Llama-3.1-70B**：BF16 141 GB，FP8 71 GB。
- **Llama-3.1-405B**：BF16 812 GB，FP8 406 GB。
- **DeepSeek-V3**（6710 亿参数，第 41 讲的 MoE）：BF16 1342 GB，原生的 FP8 权重也有 671 GB。

一张 H100 是 80 GB。70B 的模型用 BF16 就已经放不下了；405B 和 DeepSeek-V3 即使用 FP8，也要好几张卡才装得下权重。

再加上 KV Cache。每个 token 的 KV：Llama-3.1-70B 有 80 层，每个 token 320 KiB；405B 有 126 层，504 KiB；DeepSeek-V3 用了 MLA（第 40 讲），每层每个 token 只存 576 个数，61 层加起来只有 68.6 KiB。32 个请求、每个 8192 个 token 时，70B 的 KV 有 86 GB，405B 有 135 GB。

把权重和 KV 加起来，按每张卡 90% 可用：70B BF16 至少要 **4 张** H100，405B 要 14 张，DeepSeek-V3（BF16）要 19 张。这还没算激活和框架本身的开销。

算不快

就算放得下，速度也是问题。第 68 讲说过，decode 每一步要把权重从显存读一遍。H100 的带宽是 3.35 TB/s：

- 70B 的 BF16 权重，读一遍约 **42 ms**，单个请求最多每秒 24 个 token，勉强能用；
- 405B 读一遍约 242 ms，每秒 4 个 token，用户等不起。

如果把权重分到 8 张卡上，每张卡只读自己的八分之一，**8 张卡同时读**，一步就只要 70B 的 5.3 ms、405B 的 30.3 ms。多卡不只是为了装下，也是为了把“读权重”这件事并行起来：8 张卡的总带宽是一张卡的 8 倍。

prefill 是计算受限的，道理一样：一个几万 token 的长提示，一张卡要算好几秒，分到 8 张卡上就快得多。

扛不住

第三个原因是流量。一个线上服务每秒可能要处理成百上千个请求。第 82 讲的压测告诉我们，一张卡的容量是有限的，超过饱和点，延迟就失控了。这时要做的是**多复制几份**：同一个模型在很多组卡上各跑一份，请求分摊到各组上。

这三个原因对应着不同的做法：放不下、算不快，就要把一**个模型切开**，放到几张卡上合作完成一次计算；扛不住，就把**整个模型复制**几份，各自独立地服务。后面几讲会看到，“切开”的方式有好几种：按层切、按矩阵切、按专家切、按序列切，每种都要卡与卡之间交换数据，交换的代价取决于卡与卡之间的连接（第 64 讲）。

下面的演示可以选模型、精度、显卡，调并发和上下文长度，看至少要几张卡。

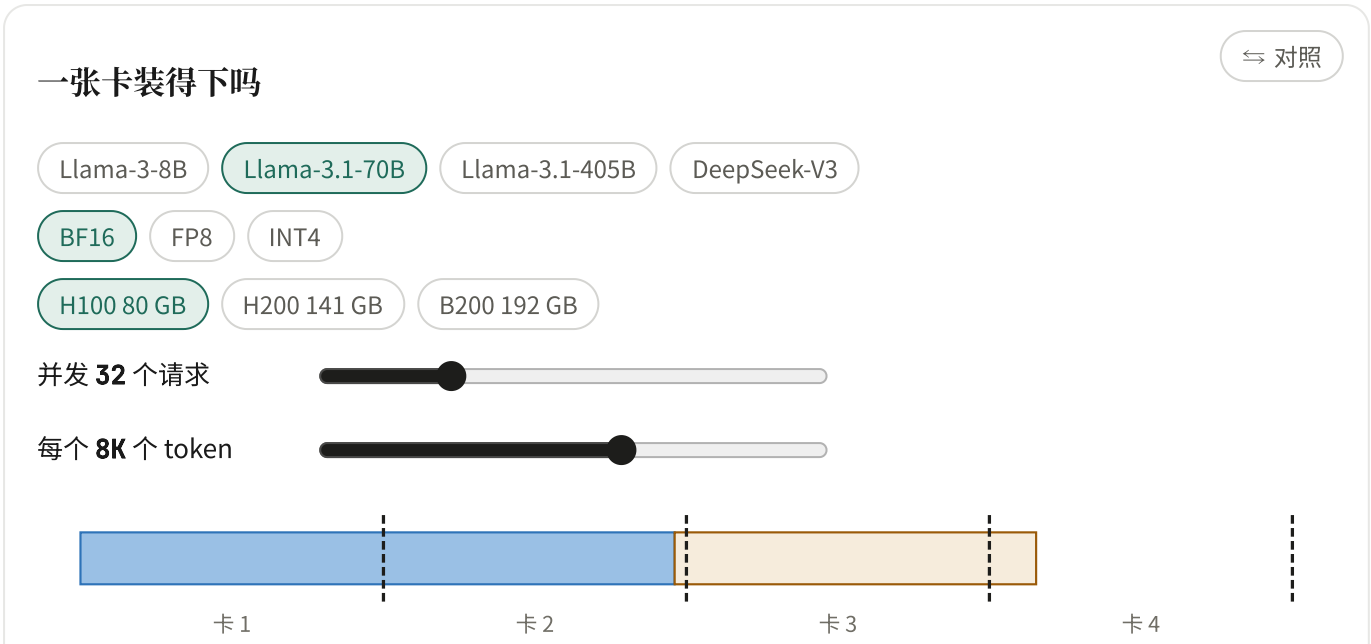


图 99-1　模型权重（蓝）和 KV Cache（琥珀）加起来要占多少显存，竖线是每张卡的容量。默认是 Llama-3.1-70B BF16、32 个 8K 的请求：至少 4 张 H100。

要点

一张卡不够的原因有三个：权重和 KV Cache 放不下；decode 每步要读一遍权重，单卡带宽决定了速度上限；流量超过一张卡的容量。前两个要把一个模型切到多张卡上，各读、各算一部分；第三个要把模型复制成多份。切开以后，卡与卡之间必须交换数据。

训练更需要多卡

训练对显存的要求比推理高得多。混合精度训练加 Adam 优化器（第 19 讲），每个参数要存：BF16 的权重 2 字节、梯度 2 字节、FP32 的主权重 4 字节、Adam 的两个状态各 4 字节，一共约 16 字节，还没算激活。

- Llama-3-8B：128 GB，一张 H100 都放不下，至少 2 张；
- Llama-3.1-70B：1130 GB，至少 16 张。

而训练的计算量更是巨大（第 46 讲），实际的预训练动辄用上千、上万张卡。分布式训练的方法（数据并行、ZeRO、张量并行、流水并行）是在训练里发展起来的，推理借用了其中的很多。第 103 讲会讲训练特有的 ZeRO。

换更大的卡不行吗

显卡的显存确实在变大：H200 有 141 GB，B200 有 192 GB。用 H200，70B 的 FP8 版本一张卡就放得下了。但模型也在变大，而且单卡的带宽和算力终究有限：405B 的模型即使放进一张卡，decode 一步也要读几百 GB。

所以业界的方向是把**更多的卡连得更紧**。一台 DGX H100 服务器里有 8 张卡，用 NVLink 互连，卡与卡之间每个方向约 450 GB/s；服务器之间用 InfiniBand 等网络连接，每张卡约 50 GB/s，慢了一个数量级。NVIDIA 的 GB200 NVL72 把 72 张卡放进同一个 NVLink 域，就是为了让更多的卡之间都能高速通信（第 65 讲的超节点）。哪种切法放在哪一层连接上，是第 109 讲的话题。

用代码复现

下面的代码算出四个模型的权重、每个 token 的 KV，以及 32 个 8K 请求时至少要几张 H100；再算单卡和 8 卡读一遍权重的时间；最后估算训练时参数相关的显存。

```python
import math
# 几个模型：参数量，每个 token 的 KV Cache 字节数（BF16）
MODELS = {
    "Llama-3-8B":     (8.03e9,  2 * 32 * 8 * 128 * 2),       # 32 层、8 个 KV 头、每头 128 维
    "Llama-3.1-70B":  (70.6e9,  2 * 80 * 8 * 128 * 2),       # 80 层
    "Llama-3.1-405B": (405.9e9, 2 * 126 * 8 * 128 * 2),      # 126 层
    "DeepSeek-V3":    (671e9,   61 * (512 + 64) * 2),        # 61 层，MLA 每个 token 每层存 576 个数（第 40 讲）
}
GPU, USABLE = 80e9, 0.9                                      # H100 80 GB，按 90% 可用
print("模型 | 权重 BF16 / FP8 | 每 token KV | 32 个请求 × 8K token 的 KV | BF16 至少几张卡")
for name, (n, kv) in MODELS.items():
    w16, w8 = n * 2, n * 1
    kv_total = kv * 32 * 8192
    cards = math.ceil((w16 + kv_total) / (GPU * USABLE))
    print(f"{name} | {w16 / 1e9:6.0f} / {w8 / 1e9:4.0f} GB | {kv / 1024:5.1f} KiB | {kv_total / 1e9:6.1f} GB | {cards}")

# 算力：decode 每一步要把权重读一遍；单卡带宽 3.35 TB/s
for name in ["Llama-3.1-70B", "Llama-3.1-405B"]:
    n = MODELS[name][0]
    t = n * 2 / 3.35e12 * 1e3
    print(f"{name}：一张卡读一遍 BF16 权重 {t:.0f} ms（如果放得下）→ 单个请求最多 {1000 / t:.0f} token/s；8 张卡各读 1/8 → {t / 8:.1f} ms")

# 训练：每个参数约 16 字节（BF16 权重 2 + 梯度 2 + FP32 主权重 4 + Adam 两个状态 8），还没算激活
for name in ["Llama-3-8B", "Llama-3.1-70B"]:
    n = MODELS[name][0]
    print(f"训练 {name}：参数相关的状态 {n * 16 / 1e9:.0f} GB，至少要 {math.ceil(n * 16 / (GPU * USABLE))} 张 H100")
```

输出：

```
模型 | 权重 BF16 / FP8 | 每 token KV | 32 个请求 × 8K token 的 KV | BF16 至少几张卡
Llama-3-8B |     16 /    8 GB | 128.0 KiB |   34.4 GB | 1
Llama-3.1-70B |    141 /   71 GB | 320.0 KiB |   85.9 GB | 4
Llama-3.1-405B |    812 /  406 GB | 504.0 KiB |  135.3 GB | 14
DeepSeek-V3 |   1342 /  671 GB |  68.6 KiB |   18.4 GB | 19
Llama-3.1-70B：一张卡读一遍 BF16 权重 42 ms（如果放得下）→ 单个请求最多 24 token/s；8 张卡各读 1/8 → 5.3 ms
Llama-3.1-405B：一张卡读一遍 BF16 权重 242 ms（如果放得下）→ 单个请求最多 4 token/s；8 张卡各读 1/8 → 30.3 ms
训练 Llama-3-8B：参数相关的状态 128 GB，至少要 2 张 H100
训练 Llama-3.1-70B：参数相关的状态 1130 GB，至少要 16 张 H100
```

表格是四个模型的显存账：70B 要 4 张卡，405B 要 14 张，DeepSeek-V3 的 BF16 版本要 19 张（它的 KV 很小，权重是大头）。接着两行是读权重的时间：70B 单卡 42 ms，8 卡 5.3 ms；405B 单卡 242 ms，8 卡 30.3 ms。最后两行是训练：8B 要 128 GB，70B 要 1130 GB。

容易混淆的地方

1. **“放得下”只是最低要求**。卡数刚好装下权重时，留给 KV Cache 的空间很少，并发上不去。实际部署通常会用比最低要求更多的卡。
2. **多卡不是免费的加速**。切开的模型每算一层，卡与卡之间都可能要交换数据；交换的时间如果和计算差不多，多卡的收益就被吃掉了。后面几讲的核心就是算清楚这笔账。
3. **复制和切分可以叠加**。一个大模型先切到 8 张卡上组成一组，再复制成很多组，是最常见的部署方式。

回顾

一张卡不够，有三个原因：放不下（70B 的 BF16 权重 141 GB，405B 812 GB，DeepSeek-V3 671 GB 的 FP8，再加上 KV Cache）；算不快（decode 每步读一遍权重，70B 单卡 42 ms，分到 8 张卡上同时读只要 5.3 ms）；扛不住（流量超过单卡容量）。前两个要把模型切开，第三个要把模型复制。训练每个参数约要 16 字节，比推理更依赖多卡。卡与卡之间的连接有快有慢：同一台服务器里的 NVLink 每个方向约 450 GB/s，跨服务器的网络约 50 GB/s。

把模型切开以后，几张卡要合作完成一次计算：有时要把各自的部分结果加起来，有时要把各自的一块拼成整体，有时要互相交换一部分数据。这些固定的通信模式有专门的名字。下一讲：**集合通信原语**。

第100讲 集合通信原语

99 为什么需要多卡

上一讲说，大模型要切到多张卡上。切开以后，几张卡要合作完成一次计算，就得交换数据。好在这些交换的“形状”很固定，归纳起来只有几种：大家的结果要加起来，大家的一块要拼起来，大家要互相交换一部分……这几种模式叫**集合通信**（collective communication），每一种叫一个**原语**。

NVIDIA 的 NCCL、AMD 的 RCCL、华为的 HCCL 等通信库，提供的就是这几个原语的高效实现。PyTorch 的 `torch.distributed` 也是对它们的封装。后面讲到的每一种并行方式，最后都会落到这几个原语上。这一讲用一个 4 张卡的小例子，把它们一个个看清楚。

设定

4 张卡，编号 0 到 3（通信库里叫 **rank**）。每张卡上有一个长度为 4 的向量，切成 4 段，每段一个数。为了看清数据从哪来，让卡 r 的数据是 [10r, 10r+1, 10r+2, 10r+3]：十位数就是它来自哪张卡。

- 卡 0：[0, 1, 2, 3]
- 卡 1：[10, 11, 12, 13]
- 卡 2：[20, 21, 22, 23]
- 卡 3：[30, 31, 32, 33]

一对多、多对一

Broadcast（广播）：把一张卡（比如卡 0）的数据复制给所有卡。之后每张卡都是 [0, 1, 2, 3]。用来分发同一份东西，比如启动时的初始权重。

Reduce（归约）：所有卡的数据**按位置相加**（也可以取最大值等），结果只放在一张卡上。4 张卡按位置相加是 [0+10+20+30, …] = [60, 64, 68, 72]，放在卡 0。

大家都要结果：All-Reduce

All-Reduce：按位置相加，**每张卡都拿到**完整的和。之后 4 张卡都是 [60, 64, 68, 72]。

它是分布式计算里最常用的原语。张量并行（第 105 讲）把一个矩阵乘法拆到几张卡上，每张卡算出一部分和，要用 All-Reduce 加起来，所有卡才能接着算下一层；数据并行训练（第 103 讲）每张卡算出自己那批数据的梯度，要 All-Reduce 求和，所有卡才能用同样的梯度更新参数。

切开和拼起来：Reduce-Scatter 与 All-Gather

Reduce-Scatter：也是按位置相加，但结果切成 p 段，卡 r 只拿第 r 段。之后卡 0 是 [60]、卡 1 是 [64]、卡 2 是 [68]、卡 3 是 [72]。每张卡只负责和的一部分。

All-Gather：每张卡手里有一段，拼起来，让每张卡都有完整的。如果每张卡分别有 [60]、[64]、[68]、[72]，All-Gather 之后每张卡都是 [60, 64, 68, 72]。

注意这两步连起来：先 Reduce-Scatter，再 All-Gather，结果恰好就是 All-Reduce。

$$\text{All-Reduce} = \text{Reduce-Scatter} + \text{All-Gather}$$

这不只是一个巧合。下一讲会看到，最高效的 All-Reduce 实现就是这么做的；而且很多场景里，我们其实只需要其中一半。比如 ZeRO（第 103 讲）只需要每张卡拿到自己负责的那部分梯度，用 Reduce-Scatter 就够了，省掉一半通信；用到参数时，再用 All-Gather 临时拼出完整的。

互相交换：All-to-All

All-to-All：每张卡把自己的数据切成 p 段，第 j 段发给卡 j。卡 r 收到每张卡的第 r 段。之后卡 0 是 [0, 10, 20, 30]（每张卡的第 0 段），卡 1 是 [1, 11, 21, 31]……把 4 张卡的数据排成一个 4 × 4 的矩阵，All-to-All 就是把它**转置**。

和前面几个不同，All-to-All 里每对卡之间交换的数据都不一样，没有求和，也没有复制。它是 MoE 专家并行（第 108 讲）的核心：每张卡上的 token 要去的专家分散在各张卡上，每张卡要给每张卡发一批不同的 token。

下面的演示可以切换六种原语，对比通信前后每张卡手里的数据。颜色表示数据来自哪张卡，深色的格子是相加的结果。



图 100-1　六种集合通信原语，4 张卡。颜色表示数据来自哪张卡，深色是多张卡相加的结果。对比 All-Reduce 与 Reduce-Scatter、All-Gather，以及像转置一样的 All-to-All。

要点

集合通信有几种固定的模式：Broadcast 一对多复制，Reduce 多对一求和，All-Reduce 求和后人人都有，Reduce-Scatter 求和后每人一段，All-Gather 每人一段拼成完整，All-to-All 每对卡交换不同的数据。All-Reduce = Reduce-Scatter + All-Gather。

每张卡要收发多少

通信要花时间，时间取决于数据量。设每张卡的数据量是 n（字节），p 张卡：

- All-Gather：每张卡已有 1/p，还缺另外 p − 1 段，至少要收 $(p-1)/p \cdot n$。
- Reduce-Scatter：每张卡要算出 1/p 的和，需要其他 p − 1 张卡在这一段上的数据，至少也是 $(p-1)/p \cdot n$。
- All-Reduce：等于前两者相加，至少 $2(p-1)/p \cdot n$。
- All-to-All：每张卡留下自己的 1/p，其余 $(p-1)/p \cdot n$ 发出去，也收进同样多。

4 张卡时，All-Reduce 每张卡至少要收发 1.5n，其他三个各 0.75n。有意思的是，卡数越多，$(p-1)/p$ 越接近 1，**每张卡的通信量几乎不随卡数增长**。这是因为好的实现会让所有卡同时收发，把任务分摊开（下一讲）。

到了大模型里

通信库。NCCL 是 NVIDIA GPU 上的事实标准，PyTorch、vLLM、SGLang、Megatron 都用它。它会根据卡之间的连接（NVLink、PCIe、InfiniBand）自动选择算法和路径。

哪个原语出现在哪里。推理里最常见的是张量并行的 All-Reduce（每层两次，第 105 讲）和 MoE 的 All-to-All（每层两次，第 108 讲）；训练里还有数据并行和 ZeRO 的 All-Reduce、Reduce-Scatter、All-Gather。

点对点通信。除了集合通信，还有最简单的 Send/Recv：一张卡发、一张卡收。流水并行（第 106 讲）把一层的输出传给下一段的卡，用的就是它；PD 分离（第 14 篇）把 KV Cache 从 prefill 的卡传到 decode 的卡，也是点对点。

用代码复现

下面的代码用 NumPy 模拟 4 张卡上的六种原语，打出每种原语之后每张卡的数据，验证 Reduce-Scatter 加 All-Gather 等于 All-Reduce，并列出每张卡至少要收发的数据量。

```python
import numpy as np

p = 4                                                    # 4 张卡（rank 0-3），每张卡上有一个长度 4 的向量，切成 4 段
data = [10 * r + np.arange(p) for r in range(p)]         # 卡 r 的数据：[10r, 10r+1, 10r+2, 10r+3]，十位数 = 来自哪张卡
def show(name, xs):
    print(f"{name:15s}" + "  ".join(f"卡{r}:{x.tolist() if x is not None else '—'}" for r, x in enumerate(xs)))

show("初始", data)
show("Broadcast", [data[0].copy() for _ in range(p)])                  # 卡 0 的数据复制给所有卡
total = sum(data)                                                      # 按位置相加：[60, 64, 68, 72]
show("Reduce", [total] + [None] * (p - 1))                             # 相加，结果只在卡 0
show("All-Reduce", [total.copy() for _ in range(p)])                   # 相加，所有卡都有结果
rs = [total[r:r + 1] for r in range(p)]
show("Reduce-Scatter", rs)                                             # 相加的结果切成 p 段，卡 r 拿第 r 段
show("All-Gather", [np.concatenate(rs) for _ in range(p)])             # 每张卡各一段，拼成完整的给所有卡
show("All-to-All", [np.array([data[j][r] for j in range(p)]) for r in range(p)])
# 卡 j 的第 r 段发给卡 r

print("Reduce-Scatter 之后再 All-Gather，结果和 All-Reduce 相同：",
np.array_equal(np.concatenate(rs), total))
n = 1.0                                                                # 每张卡的数据量记为 n，看每张卡要收发多少
print(f"每张卡至少要收的数据量 (p = {p}) ：All-Gather {(p - 1) / p * n:.2f}n，Reduce-Scatter {(p - 1) / p * n:.2f}n，"
      f"All-Reduce {2 * (p - 1) / p * n:.2f}n，All-to-All {(p - 1) / p * n:.2f}n")
```

输出：

```
初始             卡0:[0, 1, 2, 3]  卡1:[10, 11, 12, 13]  卡2:[20, 21, 22, 23]  卡3:
[30, 31, 32, 33]
Broadcast      卡0:[0, 1, 2, 3]  卡1:[0, 1, 2, 3]  卡2:[0, 1, 2, 3]  卡3:[0, 1, 2,
3]
Reduce         卡0:[60, 64, 68, 72]  卡1:—  卡2:—  卡3:—
All-Reduce     卡0:[60, 64, 68, 72]  卡1:[60, 64, 68, 72]  卡2:[60, 64, 68, 72]  卡
3:[60, 64, 68, 72]
Reduce-Scatter 卡0:[60]  卡1:[64]  卡2:[68]  卡3:[72]
All-Gather     卡0:[60, 64, 68, 72]  卡1:[60, 64, 68, 72]  卡2:[60, 64, 68, 72]  卡
3:[60, 64, 68, 72]
All-to-All     卡0:[0, 10, 20, 30]  卡1:[1, 11, 21, 31]  卡2:[2, 12, 22, 32]  卡3:
[3, 13, 23, 33]
Reduce-Scatter 之后再 All-Gather，结果和 All-Reduce 相同： True
每张卡至少要收的数据量（p = 4）：All-Gather 0.75n，Reduce-Scatter 0.75n，All-Reduce
1.50n，All-to-All 0.75n
```

每一行是一种原语之后各卡的数据，和演示一致。All-to-All 那一行：卡 0 拿到的是每张卡的第 0 段 [0, 10, 20, 30]。最后两行是验证和通信量。

容易混淆的地方

1. **All-Reduce 和 Reduce 不同**。Reduce 的结果只在一张卡上，All-Reduce 每张卡都有。
2. **All-Gather 和 All-to-All 不同**。All-Gather 后每张卡的数据都一样（完整的拼接）；All-to-All 后每张卡的数据都不一样（各自收到的那一列）。
3. **集合通信要所有卡一起参加**。只要有一张卡还没算完、没调用这个原语，其他卡都得等着。所以负载不均衡时，最慢的那张卡决定了所有卡的速度。

回顾

多张卡合作计算时，要用固定的几种模式交换数据：Broadcast（一对多复制）、Reduce（求和到一张卡）、All-Reduce（求和后每张卡都有）、Reduce-Scatter（求和后每张卡一段）、All-Gather（每张卡一段拼成完整）、All-to-All（每对卡交换不同的数据，像转置）。All-Reduce 等于 Reduce-Scatter 加 All-Gather。每张卡至少要收发的数据量：All-Reduce $2(p-1)/p \cdot n$，其余三种 $(p-1)/p \cdot n$，几乎不随卡数增长。张量并行用 All-Reduce，ZeRO 用 Reduce-Scatter 和 All-Gather，MoE 用 All-to-All。

“每张卡至少收发 $2(p-1)/p \cdot n$”是理论下限。最直接的实现，所有卡把数据发给卡 0 求和、卡 0 再发回去，卡 0 要收发 $2(p-1)\cdot n$，卡越多越慢。怎么组织通信才能达到下限？下一讲：**Ring 与 Tree All-Reduce**。

第101讲 Ring 与 Tree All-Reduce

100 集合通信原语

上一讲说，All-Reduce 每张卡至少要收发 $2(p-1)/p \cdot n$ 的数据。最直接的实现却远远达不到：所有卡把数据发给卡 0，卡 0 加起来，再发回给所有卡。卡 0 要依次收 $p-1$ 份、发 $p-1$ 份，一共 $2(p-1)\cdot n$；其他卡大部分时间在等。卡越多，卡 0 越忙，时间随卡数线性增长。

问题出在**负担全压在一张卡上**，其他卡的链路都闲着。这一讲讲两种让所有卡、所有链路一起干活的组织方式：**Ring**（环）和 **Tree**（树），以及它们各自适合什么情况。

Ring：大家排成一个圈

把 p 张卡连成一个环：卡 0 → 卡 1 → 卡 2 → … → 卡 $p-1$ → 卡 0。每张卡只和下一张卡说话。每张卡的数据切成 p 块。

第一阶段：Reduce-Scatter，$p-1$ 步。第 1 步，卡 i 把自己的第 i 块发给下一张卡，下一张卡把它加到自己的第 i 块上。第 2 步，每张卡把刚刚加过的那一块继续往下传，对方再加上自己的……每一块沿着环走了 $p-1$ 步，经过了所有的卡，每张卡都往上加了一份。$p-1$ 步之后，**每张卡恰好有一块是所有卡的总和**。

第二阶段：All-Gather，$p-1$ 步。每张卡把自己那块完整的结果传给下一张卡，对方直接覆盖；再继续往下传。$p-1$ 步之后，每张卡都有了全部 p 块完整的和。

用上一讲的 4 张卡走一遍（卡 r 的数据是 $[10r, 10r+1, 10r+2, 10r+3]$，最终结果应是 $[60, 64, 68, 72]$）：

- Reduce-Scatter 3 步之后：卡 0 的第 1 块是 64，卡 1 的第 2 块是 68，卡 2 的第 3 块是 72，卡 3 的第 0 块是 60，各自恰好有一块完整的和。
- All-Gather 3 步之后：4 张卡都是 $[60, 64, 68, 72]$。

下面的演示可以一步一步地看。琥珀色是这一步正在发出的块，深色是已经是最终结果的块。

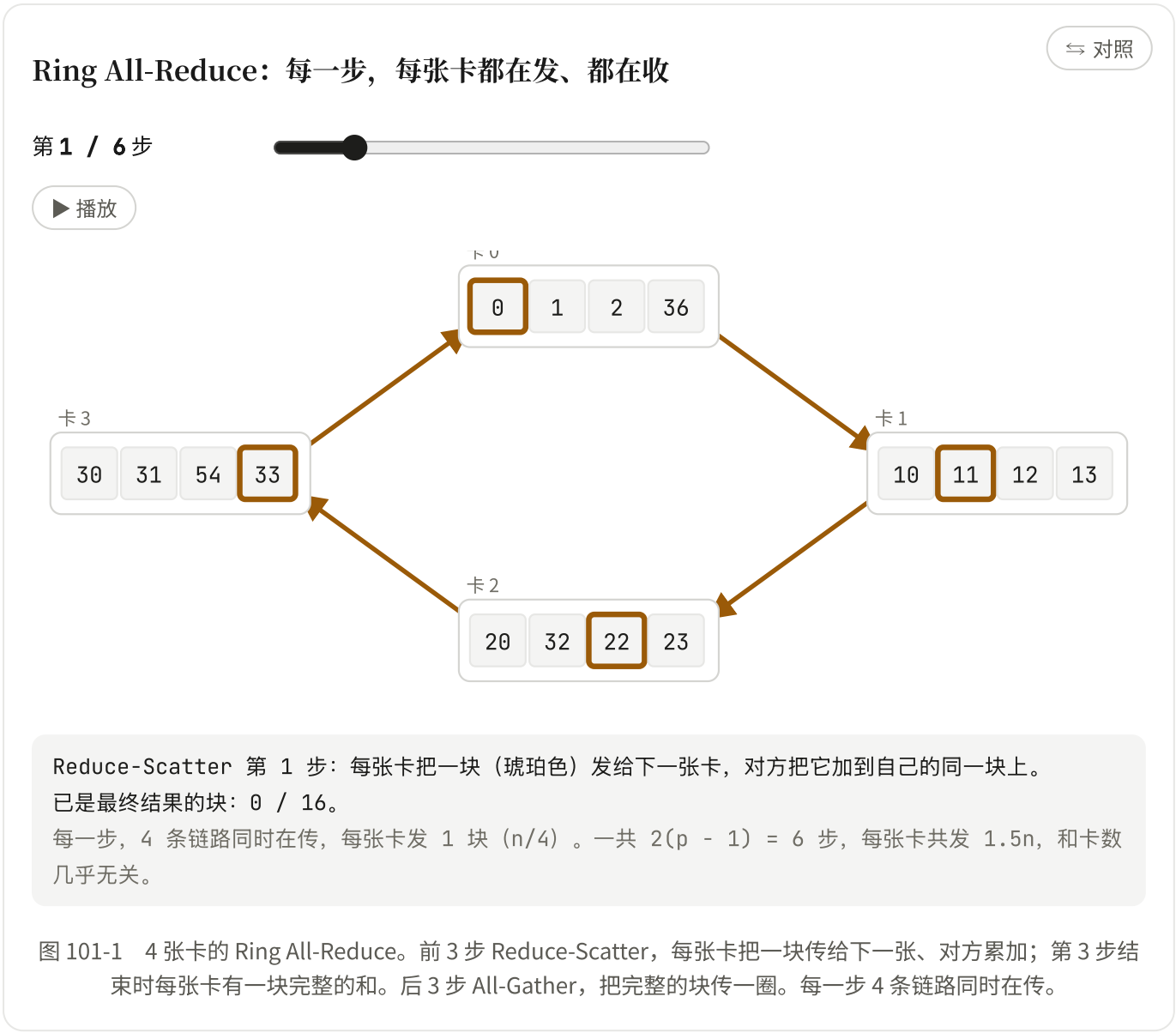


图 101-1　4 张卡的 Ring All-Reduce。前 3 步 Reduce-Scatter，每张卡把一块传给下一张、对方累加；第 3 步结束时每张卡有一块完整的和。后 3 步 All-Gather，把完整的块传一圈。每一步 4 条链路同时在传。

数一数每张卡发了多少：$2(p-1)$ 步，每步发一块，一块是 n/p，一共 $2(p-1)/p \cdot n$，正好是上一讲说的下限。而且每一步里，所有 p 条链路**同时**在传，没有谁闲着，也没有谁特别忙。4 张卡时每张卡发 $1.5n$，朴素做法里卡 0 要收发 $6n$。

两个代价：延迟和带宽

估算通信时间，要区分两种代价（第 102 讲会细讲）：

- **延迟** α：每发一次消息的固定开销，不管消息多小，都要几微秒（启动传输、同步、经过交换机……）。
- **带宽** β：每秒能传多少字节。传 m 字节，要 m/β 的时间。

一步的耗时 $\approx \alpha +$ 这一步的数据量 $\div \beta$。于是：

$$\text{朴素：}2(p-1)\cdot(\alpha+n/\beta)\qquad \text{Ring：}2(p-1)\cdot(\alpha+n/(p\cdot\beta))$$

Ring 的**带宽项** $2(p-1)/p\cdot n/\beta$ 几乎不随卡数增长，非常好；可它的**延迟项** $2(p-1)\cdot\alpha$ 随卡数线性增长。

代码按 $\alpha = 5\ \mu s$、$\beta = 50$ GB/s（跨机网络的量级）算了几种情况：

- **1 GB 的数据**（比如训练时同步梯度）：8 张卡时朴素做法 280 ms，Ring 35 ms；512 张卡时朴素做法要 20 秒，Ring 只要 45 ms。数据大，带宽项占主导，Ring 几乎完美。
- **64 KB 的数据**（比如推理时张量并行同步一层的输出）：512 张卡时 Ring 要 5.1 ms，几乎全是延迟（$2\times 511\times 5\ \mu s$）。数据小，延迟项占主导，Ring 的步数太多了。

Tree：用树来压缩步数

Tree（树形）All-Reduce 的思路是减少步数：把卡组织成一棵二叉树，数据从叶子往根逐层相加（Reduce），再从根往叶子逐层广播（Broadcast）。树的深度是 $\log_2 p$，所以只要大约 $2\cdot\log_2 p$ 步。512 张卡时，Ring 要 1022 步，树只要 18 步。

朴素的树有个问题：根节点和靠近根的节点要收发整份数据，带宽利用不好。实际的实现会把数据切成很多小块，像流水线一样一块一块地沿着树往上、往下送，让带宽项也接近 $2n/\beta$。NCCL 用的是**双二叉树**（double binary tree）：两棵互补的二叉树，各处理一半的数据，让每张卡在一棵树里当叶子、在另一棵树里当中间节点，负担均衡。

代码里，64 KB、512 张卡时，树约 **0.09 ms**，比 Ring 的 5.1 ms 快了五十多倍；1 GB 时两者都在 35 到 45 ms 之间，差不多。

要点

朴素的 All-Reduce 把负担压在一张卡上，时间随卡数线性增长。Ring 让 p 张卡排成环，先 p−1 步 Reduce-Scatter 再 p−1 步 All-Gather，每步所有链路同时传 n/p，带宽项达到下限、几乎不随卡数增长，但要 2(p−1) 步，延迟项随卡数线性增长。Tree 只要约 $2\cdot\log_2 p$ 步，适合小消息、多卡数。NCCL 会根据消息大小和拓扑自动选择。

到了大模型里

NCCL 自动选。NCCL 内置了 Ring、Tree 等多种算法，会根据卡数、消息大小、拓扑自动选择，也可以用环境变量（`NCCL_ALGO`）强制指定。大消息多用 Ring，小消息、跨很多台机器时多用 Tree。

机器内部另有捷径。一台 8 卡的 H100 服务器里，所有卡都接在 NVSwitch 上，任意两张卡之间都有高速通路。NVSwitch 还支持在交换芯片里直接做加法（NVLink SHARP），All-Reduce 的数据不用在卡之间绕一圈，进一步减少了步数和流量。InfiniBand 交换机也有类似的网内计算（SHARP）。

分层的通信。跨很多台机器时，常常先在每台机器内部做一次 Reduce-Scatter（走快速的 NVLink），再在机器之间对各自的一份做 All-Reduce（走网络），最后在机器内部 All-Gather。这样慢速网络上只传 1/8 的数据。

推理的通信大多是小消息。decode 时张量并行每层同步的只是几十个 token 的隐藏状态，几十 KB，延迟是主要矛盾。这也是为什么推理引擎里有专门为小消息优化的 All-Reduce 实现（比如 vLLM 的自定义 All-Reduce），并且尽量把张量并行限制在一台机器内部（第 109 讲）。

用代码复现

下面的代码模拟 4 张卡的 Ring All-Reduce，打出每一步之后各卡的数据，统计每张卡发了多少；再按 α-β 模型比较朴素、Ring、Tree 三种做法在不同消息大小、不同卡数下的耗时。

```python
import numpy as np, math

# 1. Ring All-Reduce：4 张卡连成环（0→1→2→3→0），数据切成 4 块。先 p-1 步 Reduce-Scatter，再 p-1 步 All-Gather
p = 4
buf = [10 * r + np.arange(p) for r in range(p)]          # 和第 100 讲一样：卡 r 的数据 [10r, …, 10r+3]
sent = [0] * p
for s in range(p - 1):                                    # Reduce-Scatter：第 s 步，卡 i 把第 (i - s) 块发给下一张卡，对方加到自己的那块上
    msgs = [(i, (i - s) % p, buf[i][(i - s) % p]) for i in range(p)]
    for i, c, v in msgs: buf[(i + 1) % p][c] += v; sent[i] += 1
    print(f"Reduce-Scatter 第 {s + 1} 步：" + "  ".join(f"卡{r}:{buf[r].tolist()}" for r in range(p)))
for s in range(p - 1):                                    # All-Gather：第 s 步，卡 i 把已经算完的第 (i + 1 - s) 块发给下一张卡，对方直接覆盖
    msgs = [(i, (i + 1 - s) % p, buf[i][(i + 1 - s) % p]) for i in range(p)]
    for i, c, v in msgs: buf[(i + 1) % p][c] = v; sent[i] += 1
    print(f"All-Gather     第 {s + 1} 步：" + "  ".join(f"卡{r}:{buf[r].tolist()}" for r in range(p)))
print(f"每张卡一共发了 {sent[0]} 块 = 2(p-1)/p · n = {2 * (p - 1) / p:.2f}n；朴素做法（都发给卡 0 再发回）卡 0 要收发 {2 * (p - 1)}n")

# 2. 耗时估算：每一步的耗时 = α（启动延迟）+ 这一步的数据量 ÷ β（单向带宽）
alpha, beta = 5e-6, 50e9                                  # 示意：α = 5 μs，β = 50 GB/s（一张卡的跨机网卡）
def naive(n, p): return 2 * (p - 1) * (alpha + n / beta)                # 卡 0 依次收、依次发
def ring(n, p):  return 2 * (p - 1) * (alpha + n / p / beta)
def tree(n, p):  return 2 * math.ceil(math.log2(p)) * alpha + 2 * n / beta   # 流水化的树：延迟项 log p，带宽项约 2n
print("数据量 | 卡数 | 朴素 | Ring | Tree")
for n in [64e3, 1e9]:
    for pp in [8, 64, 512]:
        print(f"{n / 1e6:7.3f} MB | {pp:3d} | {naive(n, pp) * 1e3:9.3f} ms | {ring(n, pp) * 1e3:8.3f} ms | {tree(n, pp) * 1e3:8.3f} ms")
```

输出：

```
Reduce-Scatter 第 1 步：卡0:[0, 1, 2, 36]  卡1:[10, 11, 12, 13]  卡2:[20, 32, 22, 23]
卡3:[30, 31, 54, 33]
Reduce-Scatter 第 2 步：卡0:[0, 1, 56, 36]  卡1:[10, 11, 12, 49]  卡2:[30, 32, 22,
23]  卡3:[30, 63, 54, 33]
Reduce-Scatter 第 3 步：卡0:[0, 64, 56, 36]  卡1:[10, 11, 68, 49]  卡2:[30, 32, 22,
72]  卡3:[60, 63, 54, 33]
All-Gather     第 1 步：卡0:[60, 64, 56, 36]  卡1:[10, 64, 68, 49]  卡2:[30, 32, 68,
72]  卡3:[60, 63, 54, 72]
All-Gather     第 2 步：卡0:[60, 64, 56, 72]  卡1:[60, 64, 68, 49]  卡2:[30, 64, 68,
72]  卡3:[60, 63, 68, 72]
All-Gather     第 3 步：卡0:[60, 64, 68, 72]  卡1:[60, 64, 68, 72]  卡2:[60, 64, 68,
72]  卡3:[60, 64, 68, 72]
每张卡一共发了 6 块 = 2(p-1)/p · n = 1.50n；朴素做法（都发给卡 0 再发回）卡 0 要收发 6n
数据量 | 卡数 | 朴素 | Ring | Tree
  0.064 MB |   8 |     0.088 ms |    0.072 ms |    0.033 ms
  0.064 MB |  64 |     0.791 ms |    0.633 ms |    0.063 ms
  0.064 MB | 512 |     6.418 ms |    5.113 ms |    0.093 ms
1000.000 MB |   8 |   280.070 ms |   35.070 ms |   40.030 ms
1000.000 MB |  64 |  2520.630 ms |   40.005 ms |   40.060 ms
1000.000 MB | 512 | 20445.110 ms |   45.032 ms |   40.090 ms
```

前六行是 Ring 的每一步，和演示一致：Reduce-Scatter 3 步之后每张卡各有一块完整的和，All-Gather 3 步之后全部完成。第七行：每张卡发了 6 块，即 1.5n。表格是耗时：大消息时 Ring 远好于朴素做法；小消息、卡多时 Tree 远好于 Ring。

容易混淆的地方

1. **Ring 的“环”是逻辑上的**。它不要求物理上真的连成一个圈，只要相邻两张卡之间能通信。NCCL 会根据实际的连接（NVLink、PCIe、网卡）安排环的顺序，尽量让每一跳都走最快的路径。
2. **“带宽最优”不等于“最快”**。Ring 的带宽项达到了下限，但延迟项很大；消息小的时候，延迟项占主导，Tree 更快。
3. **All-Reduce 的时间和卡数的关系，取决于消息大小**。大消息时几乎不随卡数增长；小消息时，Ring 随卡数线性增长，Tree 按 log p 增长。

回顾

朴素的 All-Reduce 让一张卡收发 $2(p-1)\cdot n$，时间随卡数线性增长。Ring All-Reduce 把卡排成环，数据切成 p 块：先 $p-1$ 步 Reduce-Scatter（每块沿环走一圈、被每张卡加一次），再 $p-1$ 步 All-Gather；每步所有链路同时传 n/p，每张卡共发 $2(p-1)/p \cdot n$，达到下限。但它要 $2(p-1)$ 步，延迟项随卡数增长：1 GB、512 张卡时 Ring 约 45 ms，64 KB 时却要 5.1 ms。Tree All-Reduce 只要约 $2\cdot\log_2 p$ 步，64 KB、512 张卡时约 0.09 ms。NCCL 会根据消息大小和拓扑自动选择，机器内部还有 NVSwitch 的网内计算。

这一讲我们已经用到了 α 和 β 来估算时间。下一讲把这个方法系统化：对任意一次通信，怎么快速估出它要多久，判断它是卡在延迟还是带宽上？下一讲：**通信代价模型**。

第102讲 通信代价模型

重点

62 Roofline 模型　68 Prefill 与 Decode　101 Ring 与 Tree All-Reduce

第 68 讲用一个简单的模型估算了计算的时间：耗时 ≈ max(计算量 ÷ 算力, 读写字节 ÷ 带宽)。它让我们不用跑程序，就能判断一个运算是卡在算力还是带宽上（第 62 讲的 Roofline）。

通信也需要这样一个模型。上一讲已经用过它：**α-β 模型**。这一讲把它讲完整：它的两个参数是什么意思，怎么判断一次通信卡在哪里，以及用它估算两个真实的场景：推理时的张量并行，和训练时的梯度同步。

一条消息要多久

从一张卡向另一张卡发 m 个字节，耗时

$$T = \alpha + m / \beta$$

- α（**延迟**）：不管消息多小都要付的固定代价。包括通信库准备和启动传输、硬件建立连接、数据经过交换机、对方确认收到……通常是几微秒到十几微秒。
- β（**带宽**）：链路每秒能传多少字节。H100 机内的 NVLink 每个方向约 450 GB/s；跨机的 InfiniBand NDR 每张网卡 400 Gb/s，约 50 GB/s。

这一讲用 α = 3 μs、β = 450 GB/s 代表机内 NVLink，α = 6 μs、β = 50 GB/s 代表跨机的 InfiniBand。β 是硬件规格的量级，α 是一个示意的量级（真实的 α 还包括通信库的软件开销，因实现而异），实际要以测量为准。

分界点：α·β

α 和 β 哪个重要，取决于消息有多大。两项相等时，α = m/β，也就是

$$m^* = \alpha \cdot \beta$$

比 m^* 小的消息，时间主要花在延迟上；比 m^* 大的消息，主要花在传数据上。

- NVLink：$m^* = 3\ \mu s \times 450\ GB/s \approx 1.35\ MB$。
- InfiniBand：$m^* = 6\ \mu s \times 50\ GB/s \approx 300\ KB$。

代码列出了几个消息大小。在 NVLink 上发 8 KB，3.0 μs，99.4% 是延迟，实际带宽只有 2.7 GB/s，不到标称的百分之一；发 512 KB，4.1 μs，延迟仍占 72.5%；发 16 MB，38.6 μs，实际带宽 415 GB/s，接近标称。

这和第 62 讲的 Roofline 是同一个道理：Roofline 里，算术强度低于拐点就卡在带宽上；这里，消息小于 α·β 就卡在延迟上。**小消息时，再高的带宽也用不上。**

下面的计算器可以选链路、调 α、β、消息大小和卡数。上面是三种 All-Reduce 算法的耗时（灰色是延迟项，绿色是带宽项），下面是单条消息的实际带宽随消息大小的变化，竖线是 α·β。

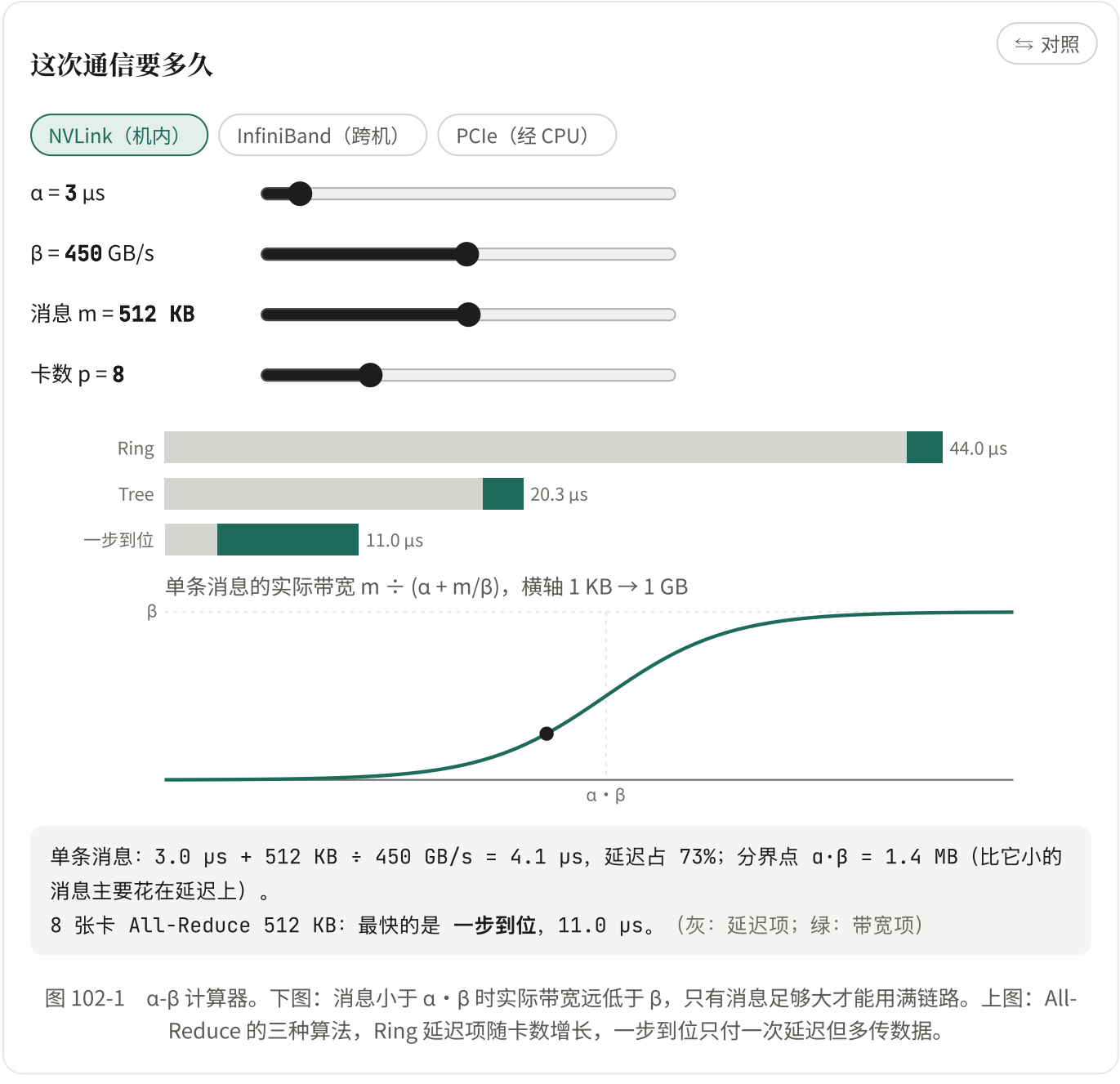


图 102-1　α-β 计算器。下图：消息小于 α · β 时实际带宽远低于 β，只有消息足够大才能用满链路。上图：All-Reduce 的三种算法，Ring 延迟项随卡数增长，一步到位只付一次延迟但多传数据。

从一条消息到一次集合通信

一次集合通信由很多步组成，每一步都是一条（或很多条同时进行的）消息。把每一步的 $\alpha + m/\beta$ 加起来，就是整体的估算。上一讲的 Ring All-Reduce：

$$T_{ring} = 2(p-1)\cdot(\alpha + m/(p\cdot\beta)) \qquad = 2(p-1)\cdot\alpha\text{（延迟项）} + 2(p-1)/p\cdot m/\beta\text{（带宽项）}$$

两项的平衡点和单条消息不同：Ring 有 $2(p-1)$ 次 α，延迟项被放大了。另一种思路是**一步到位**：在 NVSwitch 这样任意两张卡都直连的拓扑里，每张卡同时把另外 $p-1$ 张卡的数据读过来，自己加起来。只付一次 α，代价是每张卡要收 $(p-1)\cdot m$ 的数据。小消息时它最快。

例一：推理时的张量并行

Llama-3.1-70B 用 8 张 H100 做张量并行（第 105 讲），decode 一步，32 个请求。每一层要做 **2 次 All-Reduce**（注意力之后一次、FFN 之后一次），每次同步的是 32 个 token 的隐藏状态：$32 \times 8192 \times 2$ 字节 $\approx$ **524 KB**。80 层，一步就是 160 次。

- 用 Ring：每次约 44 μs，一步 **7.05 ms**，其中 6.72 ms 是延迟项（14 次 $\alpha \times 160$）。
- 用一步到位：每次约 11.2 μs，一步 **1.78 ms**。

第 99 讲算过，70B 的权重分到 8 张卡上，每张卡读一遍只要约 5.3 ms。用 Ring 时，通信比计算还慢；换成一步到位，通信降到计算的三分之一左右。**推理的通信大多是小消息，延迟才是主要矛盾**。这正是 vLLM、SGLang 为张量并行写了专门的小消息 All-Reduce 的原因，也是为什么张量并行几乎总是放在一台机器内部：跨机的 α 更大、β 更小。

例二：训练时的梯度同步

Llama-3-8B 的数据并行训练，16 台机器（第 103 讲），每一步要 All-Reduce 16 GB 的 BF16 梯度，走 InfiniBand：

- Ring：约 **0.60 s**，几乎全是带宽项。16 GB 远大于 $\alpha\cdot\beta$，延迟可以忽略。

这时要做的是减少数据量（比如梯度用更低的精度）、或者让通信和计算重叠（反向传播算完一层的梯度，就开始同步这一层，同时算下一层），而不是去优化延迟。

要点

一次通信的时间 $T = \alpha + m/\beta$。消息小于 $\alpha \cdot \beta$ 时主要花在延迟上（NVLink 约 1.35 MB，InfiniBand 约 300 KB），实际带宽远低于标称；大于它时才接近带宽上限。推理的张量并行是大量小消息，延迟是主要矛盾，要减少步数、留在机内；训练的梯度同步是大消息，带宽是主要矛盾，要减少数据量、和计算重叠。

测出来的带宽怎么读

NVIDIA 的 `nccl-tests` 是测集合通信性能的标准工具，它报告两种带宽：

- **算法带宽**（algbw）= 数据量 ÷ 时间。直观，但不同原语、不同卡数之间不好比较。
- **总线带宽**（busbw）= 算法带宽 × 一个和原语有关的系数。对 All-Reduce，系数是 $2(p-1)/p$，也就是把“每张卡实际要传的数据量”算进去。

例二里，16 GB 用 0.60 s，算法带宽约 26.7 GB/s；乘以 $2 \times 15/16$，总线带宽约 50 GB/s，正好等于链路的 β。所以看 `nccl-tests` 的结果时，**拿总线带宽和硬件的链路带宽比**，就知道通信库把链路用到了几成。

用代码复现

下面的代码按 α-β 模型，列出 NVLink 和 InfiniBand 上不同消息大小的耗时、延迟占比和实际带宽；然后估算上面的两个例子；最后演示算法带宽和总线带宽的换算。

```python
# α-β 模型：发一条 m 字节的消息，耗时 T = α + m / β。下面的 α 是示意的量级（含通信库的软件开销），β 是单向带宽
LINKS = {"NVLink（H100 机内）": (3e-6, 450e9), "InfiniBand NDR（跨机，每卡一张网卡）": (6e-6, 50e9)}

for name, (a, b) in LINKS.items():
    print(f"{name}：α = {a * 1e6:.0f} μs，β = {b / 1e9:.0f} GB/s，分界点 α·β = {a * b / 1e3:,.0f} KB")
    for m in [8e3, 512e3, 16e6, 1e9]:
        t = a + m / b
        print(f"  {m / 1e6:9.3f} MB：{t * 1e6:10.1f} μs，其中延迟占 {a / t:5.1%}，实际带宽 {m / t / 1e9:6.1f} GB/s")

def ring_allreduce(m, p, a, b):                          # 第 101 讲：2(p-1) 步，每步 m/p
    return 2 * (p - 1) * (a + m / p / b)

# 例 1：推理，Llama-3.1-70B 张量并行 8 卡（机内 NVLink），decode 一步，32 个请求
a, b = LINKS["NVLink（H100 机内）"]
m = 32 * 8192 * 2                                       # 每次同步 32 个 token 的隐藏状态（d = 8192，BF16）
t1 = ring_allreduce(m, 8, a, b)
print(f"例 1：每次 All-Reduce {m / 1e3:.0f} KB，Ring 约 {t1 * 1e6:.1f} μs；每层 2 次、80 层 = {160 * t1 * 1e3:.2f} ms/步"
      f"（延迟项 {160 * 14 * a * 1e3:.2f} ms）")
t1b = a + 7 * m / b                                      # 一步到位：每张卡通过 NVSwitch 同时读其余 7 张卡的数据，自己相加
print(f"  改用一步到位的 All-Reduce：每次约 {t1b * 1e6:.1f} μs，每步 {160 * t1b * 1e3:.2f} ms（多传了数据，但只付一次 α）")
# 例 2：训练，Llama-3-8B 的 BF16 梯度 16 GB，16 台机器之间做数据并行（每台机器出一份，走 InfiniBand）
a, b = LINKS["InfiniBand NDR（跨机，每卡一张网卡）"]
m = 8.03e9 * 2
t2 = ring_allreduce(m, 16, a, b)
print(f"例 2：16 GB 梯度、16 个节点 Ring All-Reduce 约 {t2:.2f} s（带宽项 {2 * 15 / 16 * m / b:.2f} s）")
# 算法带宽与总线带宽（nccl-tests 的两种报告方式）
algbw = m / t2
print(f"nccl-tests 的读法：算法带宽 = 数据量 ÷ 时间 = {algbw / 1e9:.1f} GB/s；总线带宽 = 算法带宽 × 2(p-1)/p（p = 16）= {algbw * 2 * 15 / 16 / 1e9:.1f} GB/s，接近链路的 β")
```

输出：

```
NVLink（H100 机内）：α = 3 μs，β = 450 GB/s，分界点 α·β = 1,350 KB
       0.008 MB:        3.0 μs，其中延迟占 99.4%，实际带宽    2.7 GB/s
       0.512 MB:        4.1 μs，其中延迟占 72.5%，实际带宽  123.7 GB/s
      16.000 MB:       38.6 μs，其中延迟占  7.8%，实际带宽  415.0 GB/s
    1000.000 MB:     2225.2 μs，其中延迟占  0.1%，实际带宽  449.4 GB/s
InfiniBand NDR（跨机，每卡一张网卡）：α = 6 μs，β = 50 GB/s，分界点 α·β = 300 KB
       0.008 MB:        6.2 μs，其中延迟占 97.4%，实际带宽    1.3 GB/s
       0.512 MB:       16.2 μs，其中延迟占 36.9%，实际带宽   31.5 GB/s
      16.000 MB:      326.0 μs，其中延迟占  1.8%，实际带宽   49.1 GB/s
    1000.000 MB:    20006.0 μs，其中延迟占  0.0%，实际带宽   50.0 GB/s
例 1：每次 All-Reduce 524 KB，Ring 约 44.0 μs；每层 2 次、80 层 = 7.05 ms/步（延迟项
6.72 ms）
   改用一步到位的 All-Reduce：每次约 11.2 μs，每步 1.78 ms（多传了数据，但只付一次 α）
例 2：16 GB 梯度、16 个节点 Ring All-Reduce 约 0.60 s（带宽项 0.60 s）
nccl-tests 的读法：算法带宽 = 数据量 ÷ 时间 = 26.7 GB/s；总线带宽 = 算法带宽 × 2(p-1)/p（p
= 16）= 50.0 GB/s，接近链路的 β
```

前十行是两种链路上的单条消息：小消息几乎全是延迟，大消息接近标称带宽。接着是例一：Ring 每步 7.05 ms，一步到位 1.78 ms。然后是例二：0.60 s。最后一行是带宽的换算：总线带宽 50.0 GB/s。

容易混淆的地方

1. **标称带宽常是双向的。**“NVLink 900 GB/s”是 H100 每张卡所有 NVLink 的双向总和，单向约 450 GB/s；“400 Gb/s 的网卡”单位是比特，约 50 GB/s。代入模型前要换算成单向的字节每秒。
2. **α 不只是物理延迟。**光在线缆里走几米只要几十纳秒，α 的大头是软件和协议：通信库的调度、kernel 的启动、同步。这也是为什么优化通信库能显著降低 α。
3. **模型是估算，不是测量。**拥塞、多个通信同时进行、拓扑不对称都会让实际时间偏离模型。它的用处是快速判断量级和瓶颈，具体数字要用 `nccl-tests` 或 Profiler 测（第 66 讲）。

回顾

α-β 模型把一次通信的时间写成 $T = \alpha + m/\beta$：α 是每条消息的固定延迟，β 是链路带宽。分界点 α·β（NVLink 约 1.35 MB，InfiniBand 约 300 KB）以下卡在延迟上，以上卡在带宽上。一次集合通信把每一步的耗时加起来，Ring All-Reduce 是 $2(p-1)\cdot(\alpha + m/(p\beta))$。70B 张量并行 8 卡

decode 时，每步 160 次 524 KB 的 All-Reduce，Ring 要 7.05 ms（6.72 ms 是延迟），一步到位只要 1.78 ms；8B 训练同步 16 GB 梯度，16 个节点约 0.60 s，几乎全是带宽。`nccl-tests` 的总线带宽可以直接和链路带宽比较。

有了估算通信的工具，就可以分析各种并行方式了。先从训练讲起：数据并行最简单，每张卡一份完整的模型；可第 99 讲算过，8B 模型训练就要 128 GB 的状态，一张卡都放不下。怎么把这些状态切到多张卡上？下一讲：**数据并行与 ZeRO/FSDP**。

第103讲 数据并行与 ZeRO

19 优化器　20 训练循环　99 为什么需要多卡　100 集合通信原语　102 通信代价模型

第 99 讲算过，训练一个模型，每个参数要存约 16 字节：BF16 的权重 2 字节、梯度 2 字节，FP32 的主权重 4 字节，Adam 的两个状态（第 19 讲）各 4 字节。Llama-3-8B 就是 128 GB，一张 80 GB 的卡放不下。

训练最常用的并行方式是**数据并行**，可它恰恰要求每张卡放一份完整的状态。这一讲先讲数据并行怎么工作，再讲 ZeRO 怎么在不改变计算结果的前提下，把这些状态切到多张卡上，以及它要付出多少通信。PyTorch 的 FSDP 就是 ZeRO 最彻底的那一级。

数据并行：每人一份模型，各算一批数据

数据并行的想法很简单：N 张卡上各放一份完整的模型，把一个大批次的数据平均分成 N 份，每张卡用自己那份数据做前向、反向，算出梯度。

可是各卡算出的梯度不一样（数据不同），直接各自更新，模型就分叉了。所以在更新之前，要用 All-Reduce（第 100 讲）把所有卡的梯度加起来、取平均。因为损失是对所有样本求平均的，各份数据的梯度取平均，**正好等于用整个大批次算出的梯度**。代码用一个线性回归验证了这一点：4 张卡各用 16 条数据算梯度再平均，和用 64 条数据一次算出的完全相同。

之后每张卡用同样的梯度做同样的更新，模型始终保持一致。

数据并行的计算是完美切分的：N 张卡，每张只算 1/N 的数据。可显存一点没省：**每张卡都存着完整的 16 字节/参数**。8B 模型每张卡要 128.5 GB，放不下。

ZeRO：把冗余的状态切开

2019 年微软的 ZeRO（Zero Redundancy Optimizer）注意到：数据并行里，N 张卡存的状态**完全一样**，这是 N 倍的冗余。能不能每张卡只存 1/N，需要的时候再去拿？

它分三级，一级比一级切得多：

ZeRO-1：切优化器状态。优化器状态（FP32 主权重和 Adam 的两个状态，12 字节/参数）是大头。让每张卡只负责 1/N 的参数的更新：卡 r 只存第 r 段参数的优化器状态。更新时，梯度还是要 All-Reduce，但每张卡只更新自己那一段，然后用 **All-Gather** 把更新后的各段参数拼回完整的模型。8 张卡时，每张卡从 128.5 GB 降到 **44.2 GB**。

ZeRO-2：再切梯度。既然卡 r 只更新第 r 段，它也只需要第 r 段的梯度。把梯度的 All-Reduce 换成 **Reduce-Scatter**（第 100 讲）：求和之后，每张卡只拿自己那一段。梯度也只需存 1/N。每张卡降到 **30.1 GB**。

ZeRO-3：连参数也切。每张卡平时只存 1/N 的 BF16 权重。前向算到某一层时，用 All-Gather 临时拼出这一层的完整权重，算完就扔掉；反向时再拼一次。每张卡降到 **16.1 GB**，正好是 16Ψ/N。PyTorch 的 **FSDP**（Fully Sharded Data Parallel）就是 ZeRO-3 的实现。

代码验证了 ZeRO 的切分更新不改变结果：每张卡只更新自己那一段参数，再拼起来，和不切分时一步更新的结果完全相同。

下面的演示可以选阶段、调卡数，看每张卡上三块状态各占多少。



图 103-1　ZeRO 的三级切分。从数据并行到 ZeRO-3，优化器状态、梯度、参数依次被切成 N 份（虚线框的部分），每张卡的显存越来越小；卡数加倍，被切的部分减半。

要多付多少通信

切开状态，代价是通信。设参数量为 Ψ，BF16 下一份梯度或参数是 2Ψ 字节。用第 100 讲的结论，All-Reduce 每张卡收发约 2 × 2Ψ，Reduce-Scatter 和 All-Gather 各约 2Ψ：

- **数据并行**：梯度 All-Reduce，约 2 × 2Ψ。
- **ZeRO-1、ZeRO-2**：梯度 Reduce-Scatter（2Ψ）+ 更新后的参数 All-Gather（2Ψ），一共还是约 2 × 2Ψ，**和数据并行一样多**。All-Reduce 本来就等于 Reduce-Scatter 加 All-Gather，ZeRO 只是把这两半拆开，中间插进“只更新自己那段”。
- **ZeRO-3**：前向 All-Gather 参数（2Ψ），反向再 All-Gather 一次（2Ψ），梯度 Reduce-Scatter（2Ψ），一共约 3 × 2Ψ，是数据并行的 **1.5 倍**。

对 8B 模型，每步每张卡的通信从 32 GB 增加到 48 GB。用第 102 讲的估算，跨机网络上这是零点几秒到一秒的量级。实际的实现会把通信和计算**重叠**起来：算第 k 层时，提前 All-Gather 第 k + 1 层的参数；反向算完一层的梯度，就开始 Reduce-Scatter 它，同时算下一层。只要计算时间足够长，通信大部分可以被藏起来。

> **要点**
> 数据并行每张卡放一份完整的模型、算 1/N 的数据，梯度用 All-Reduce 求平均，结果和单卡大批次相同，但每卡要存完整的 16 字节/参数。ZeRO 去掉这种冗余：ZeRO-1 切优化器状态，ZeRO-2 再切梯度（All-Reduce 拆成 Reduce-Scatter + All-Gather，通信不变），ZeRO-3/FSDP 连参数也切（用到时 All-Gather，通信变成 1.5 倍）。

显存里还有激活

ZeRO 切的是和参数有关的状态。训练时还有一大块是**激活**：前向每一层的中间结果要留到反向时用（第 18 讲）。它和批大小、序列长度成正比，长序列训练时往往比参数状态还大。ZeRO 管不了激活，要用别的办法：

- **激活重算**（activation checkpointing）：前向时只保留少数几层的输出，反向时再把中间的重新算一遍，用计算换显存（第 104 讲）。
- **张量并行、序列并行**（第 105、107 讲）：把激活本身也切到多张卡上。

所以大规模训练通常把 ZeRO/FSDP 和其他并行方式组合起来用。

到了大模型里

训练框架的标配。DeepSpeed 实现了 ZeRO 的三级；PyTorch 的 FSDP（以及新版的 FSDP2）实现了 ZeRO-3；Megatron-LM 里有“分布式优化器”，相当于 ZeRO-1。中小规模的模型（几十亿参数），只用 FSDP 往往就够了。

推理一般不用 ZeRO-3。推理没有梯度和优化器状态，只有权重。如果像 ZeRO-3 那样把权重切开、每次用到再 All-Gather，decode 的每一步都要把整个模型的权重在卡之间传一遍：8B 模型每步每卡要收约 14 GB，即使走 NVLink 也要 30 ms 左右，比读一遍权重（4.8 ms）还慢得多。所以推理用的是张量并行、专家并行这些“不搬权重、只搬激活”的切法（第 105、108 讲）。

强化学习里训练和推理挨在一起。第 49 讲的 RL 训练，一边用推理引擎生成回答，一边用训练框架更新参数，权重要在两种切分方式之间来回转换。这也是现在训练框架和推理引擎结合得越来越紧的原因之一。

用代码复现

下面的代码先验证数据并行的梯度平均等于全量梯度、ZeRO 的切分更新等于不切分；再算出 8B 模型在 8 张卡上各级 ZeRO 每张卡的显存和每步的通信量。

```python
import numpy as np

# 1. 数据并行：4 张卡各拿一份数据算梯度，All-Reduce 取平均 = 用全部数据算一次梯度
rng = np.random.default_rng(0)
X, y = rng.standard_normal((64, 8)), rng.standard_normal(64)
w = np.zeros(8)
grad = lambda w, X, y: 2 * X.T @ (X @ w - y) / len(y)          # 线性回归的均方误差梯度
g_full = grad(w, X, y)
g_dp = sum(grad(w, X[i::4], y[i::4]) for i in range(4)) / 4     # 每张卡 16 条数据，All-Reduce 后求平均
print("4 卡梯度求平均 = 全部数据的梯度：", np.allclose(g_full, g_dp))

# 2. ZeRO：每张卡只负责 1/4 的参数的优化器状态。Reduce-Scatter 拿到自己那段梯度，只更新自己那段，再 All-Gather 拼回完整参数
p, lr = 4, 0.1
w_ref = w - lr * g_full                                          # 对照：普通的 SGD 一步（这里没用动量，只演示切分）
shards = np.split(np.arange(8), p)
w_parts = [w[s] - lr * (sum(grad(w, X[i::4], y[i::4]) for i in range(4)) / 4)[s] for s in shards]   # 卡 r 只更新第 r 段
print("ZeRO 切分更新再 All-Gather = 不切分：", np.allclose(np.concatenate(w_parts), w_ref))

# 3. 训练 Llama-3-8B，每张卡的显存（只算和参数有关的状态）：BF16 权重 2Ψ + BF16 梯度 2Ψ + FP32 主权重和 Adam 两个状态 12Ψ
Psi, N = 8.03e9, 8
stages = {
    "数据并行（每卡一份完整的）": 2 * Psi + 2 * Psi + 12 * Psi,
    "ZeRO-1（切优化器状态）":     2 * Psi + 2 * Psi + 12 * Psi / N,
    "ZeRO-2（再切梯度）":         2 * Psi + (2 * Psi + 12 * Psi) / N,
    "ZeRO-3 / FSDP（再切参数）":  16 * Psi / N,
}
comm = {"数据并行（每卡一份完整的）": 2, "ZeRO-1（切优化器状态）": 2, "ZeRO-2（再切梯度）": 2, "ZeRO-3 / FSDP（再切参数）": 3}
for k, v in stages.items():
    print(f"{k}：每卡 {v / 1e9:6.1f} GB；每步每卡通信约 {comm[k]} × 2Ψ = {comm[k] * 2 * Psi / 1e9:.0f} GB")
```

输出：

```
4 卡梯度求平均 = 全部数据的梯度：  True
ZeRO 切分更新再 All-Gather = 不切分：  True
数据并行（每卡一份完整的）：每卡   128.5 GB；每步每卡通信约 2 × 2Ψ = 32 GB
ZeRO-1（切优化器状态）：每卡    44.2 GB；每步每卡通信约 2 × 2Ψ = 32 GB
ZeRO-2（再切梯度）：每卡    30.1 GB；每步每卡通信约 2 × 2Ψ = 32 GB
ZeRO-3 / FSDP（再切参数）：每卡    16.1 GB；每步每卡通信约 3 × 2Ψ = 48 GB
```

前两行是两个验证。后四行是显存和通信：从数据并行的 128.5 GB 降到 ZeRO-3 的 16.1 GB；通信前三者都是 32 GB，ZeRO-3 是 48 GB。

容易混淆的地方

1. **ZeRO 不改变计算结果**。它只是改变了状态存在哪里、什么时候通信，每一步的更新和普通的数据并行完全相同。
2. **ZeRO-3 和张量并行都在“切参数”，方式不同**。ZeRO-3 存的时候切开、算的时候拼回完整的矩阵，每张卡仍然做完整的计算（只是数据不同）；张量并行连计算也切开，每张卡只算矩阵乘法的一部分（下一讲）。
3. **卡数增加，没切的部分不会变小**。ZeRO-1 时，权重和梯度每卡始终是 4Ψ；只有切开的部分随卡数减少。要压到最小，就得用 ZeRO-3。

回顾

数据并行让每张卡用一份完整的模型算 1/N 的数据，梯度用 All-Reduce 求平均，结果等于单卡大批次，但每张卡要存约 16 字节/参数的完整状态，8B 模型 128.5 GB。ZeRO 去掉这种冗余：ZeRO-1 切优化器状态（44.2 GB），ZeRO-2 再切梯度（30.1 GB），ZeRO-3/FSDP 连参数也切（16.1 GB）。ZeRO-1、2 的通信和数据并行一样，ZeRO-3 是 1.5 倍，靠和计算重叠来掩盖。激活要靠重算或其他并行方式。推理不用 ZeRO-3，因为每步搬一遍权重太慢。

ZeRO 切的是和参数有关的状态，可训练时显存里还有一大块是激活，序列一长，它比参数状态还大。激活到底有多大、怎么省？16 字节/参数这个数又是怎么来的？下一讲：**混合精度训练与激活重计算**。

第104讲 混合精度训练与激活重计算

10 浮点数与精度　18 反向传播　84 FlashAttention　103 数据并行与 ZeRO

上一讲算过训练一个 8B 模型的“状态”：BF16 的参数和梯度、FP32 的主参数、Adam 的两份滑动平均，一共 16 字节/参数，128.5 GB；ZeRO-3 切到 8 张卡上，每卡 16.1 GB。看起来一张 80 GB 的卡绰绰有余。

可这只是一半。训练时还有一大块显存是**激活**：前向每一层的中间结果，要留到反向传播时用（第 18 讲）。上一讲最后提了一句，它“长序列训练时往往比参数状态还大”。这一讲把它算清楚：8B 模型、8192 长的序列、batch 只有 1 条，激活要多少？然后讲两件事：16 字节/参数这个数是怎么来的（混合精度训练），以及怎么用多算一点来换少存很多（激活重计算）。

混合精度：用 BF16 算，用 FP32 记

第 10 讲讲过 FP32、FP16、BF16 的位布局：BF16 和 FP32 一样有 8 位指数，范围从 10^{-38} 到 10^{38}，但只有 7 位尾数，有效数字两到三位；FP16 尾数多 3 位，指数只有 5 位，最大 65504。

Tensor Core 算 BF16 比 FP32 快得多（H100 上 989 对 67 TFLOPS，第 60 讲），数据也小一半。所以前向、反向的矩阵乘法都用 BF16。但有两样东西不能用 BF16：

```python
import numpy as np

def bf16(x):
    """把 FP32 的低 16 位清零，得到 BF16 能表示的数（向零舍入，演示用）。"""
    return (np.asarray(x, dtype=np.float32).view(np.uint32) &
np.uint32(0xFFFF0000)).view(np.float32)

g = 2e-8                                                # 一个很小的梯度
print(f"梯度 {g:.0e}：FP16 存成 {float(np.float16(g)):.1e}，BF16 存成
{float(bf16(g)):.2e}")
S = 1024                                                # loss 放大 1024 倍，梯度跟着放大
print(f"放大 {S} 倍再存 FP16：{float(np.float16(g * S)):.3e}，用 FP32 除回来
{float(np.float32(np.float16(g * S)) / S):.2e}")
print(f"FP16 最大 {float(np.finfo(np.float16).max):.0f}，最小的正规数
{float(np.finfo(np.float16).tiny):.1e}；BF16 的范围和 FP32 一样，约 1e-38 到 3e38")

w32, w16, upd = np.float32(1.0), bf16(1.0), np.float32(1e-4)    # 一个权重，每步更新
1e-4
for _ in range(1000):
    w32 = np.float32(w32 + upd)
    w16 = bf16(w16 + upd)
print(f"权重 1.0 每步加 1e-4、加 1000 步：FP32 得到 {float(w32):.4f}，BF16 得到
{float(w16):.4f}（1 附近 BF16 的间隔是 {2 ** -7}）")
```

```
梯度 2e-08：FP16 存成 0.0e+00，BF16 存成 1.99e-08
放大 1024 倍再存 FP16：2.050e-05，用 FP32 除回来 2.00e-08
FP16 最大 65504，最小的正规数 6.1e-05；BF16 的范围和 FP32 一样，约 1e-38 到 3e38
权重 1.0 每步加 1e-4、加 1000 步：FP32 得到 1.1000，BF16 得到 1.0000（1 附近 BF16 的间隔
是 0.0078125）
```

第一，FP16 的梯度会下溢。2×10^{-8} 的梯度，FP16 存成 0：它最小的数在 6×10^{-8} 左右（再往下是精度越来越差的非正规数）。早期用 FP16 训练时要做**损失缩放**（loss scaling）：把 loss 乘上 1024 或更大，梯度跟着放大，存成 FP16 不会丢；更新前再用 FP32 除回来。放大倍数太大又会上溢（超过 65504 变成无穷大），所以要动态调整：一出现无穷大就减半、跳过这一步。BF16 的范围和 FP32 一样，2×10^{-8} 存得好好的，不需要损失缩放。这是今天大模型训练普遍用 BF16 的主要原因。

第二，权重的更新会被吞掉。权重 1.0，每步更新 10^{-4}（学习率乘梯度通常就这么小）。BF16 在 1 附近相邻两个数的间隔是 $2^{-7} \approx 0.0078$，$1 + 0.0001$ 舍入后还是 1：加 1000 步，还是 1.0000，而 FP32 得到 1.1。所以优化器手里要有一份 **FP32 的主参数**（master weights）：更新在 FP32 上做，再转成 BF16 给下一步的前向用。

于是每个参数要存：BF16 参数 2 字节、BF16 梯度 2 字节、FP32 主参数 4 字节、Adam 的两份 FP32 滑动平均 8 字节，一共 **16 字节**。这就是上一讲那个数。更进一步的 FP8 训练（DeepSeek-V3 就用了 FP8 混合精度训练）让矩阵乘法的输入也降到 8 位，主参数和优化器状态仍然是高精度的（FP8 的格式见第 93 讲）。

激活有多大

前向时，每一层要为反向留下哪些东西？第 18 讲的反向传播说过：算一个运算的梯度，往往要用到它前向时的输入。一层 Transformer 里，线性层要留输入，LayerNorm 要留输入和统计量，GELU 要留输入，dropout 要留掩码，注意力要留 Q、K、V，还有 softmax 之后那个 [头数, s, s] 的注意力矩阵。

Korthikanti 等人 2022 年的论文（Megatron-LM 团队）把这些一项项加起来，对 GPT 结构、16 位激活，每层是：

$$\text{每层激活} = s\,b\,h\left(34 + \frac{5\,a\,s}{h}\right)\ \text{字节}$$

s 是序列长度，b 是 micro-batch 大小，h 是隐藏维度，a 是头数。括号里的 34 来自和 s 成正比的那些张量（各种 [s, b, h] 和 [s, b, 4h]）；5as/h 来自注意力分数、softmax 结果、dropout 掩码这些 [a, s, s] 的矩阵，它和 s^2 成正比。Llama 的 FFN 是 SwiGLU、没有 dropout，系数会有些不同，但量级一样。

用计算换显存：激活重计算

全部重计算（activation checkpointing）：前向时每层只留下它的输入（2sbh 字节），中间结果全部扔掉；反向算到这一层时，从留下的输入把这一层的前向重新跑一遍，得到中间结果，再算梯度。显存从“所有层的全部激活”降到“所有层的输入 + 一层的激活”。代价是每层多跑一次前向：训练本来是前向 2N + 反向 4N = 6N（第 35 讲），现在是 8N，**多算 33%**。

选择性重计算：仔细看那个公式，5as/h 这一项在 s 大的时候是大头，可它对应的运算（QK^T、softmax、乘 V）计算量并不大。只重算这一部分，就能去掉和 s^2 成正比的那一项，只多花一点计算。

FlashAttention 天然就是选择性重计算。第 84 讲说过，FlashAttention 不把 [s, s] 的注意力矩阵写进显存，只存每行的最大值和求和，反向时按块重新算。用了它，5as/h 这一项本来就不存在了。所以今天的训练框架里，“选择性重计算”和“用 FlashAttention”在显存上的效果基本一样。

用公式算一下 Llama-3-8B、b = 1 时，32 层加起来：

```
h, a, L = 4096, 32, 32                                  # Llama-3-8B：隐藏维度、头数、层数
b = 1                                                   # micro-batch
print("序列长度 | 不重算 | FlashAttention / 选择性重算 | 全部重算 | 选择性重算多算")
for s in (2048, 8192, 32768):
    lin, att = 34 * s * b * h, 5 * a * s * s * b    # 每层：和 s 成正比的部分；注意力分数那一项
    none, sel, full = L * (lin + att), L * lin, L * 2 * s * b * h + lin
    print(f"{s:>6} | {none / 1e9:7.1f} GB | {sel / 1e9:6.1f} GB | {full / 1e9:5.2f} GB | {4 * s / (3 * (24 * h + 4 * s)):.1%}")
print(f"参数、梯度、优化器状态：16 字节 × 80.3 亿 = {16 * 8.03e9 / 1e9:.1f} GB（ZeRO-3 切到 8 卡是 {16 * 8.03e9 / 8 / 1e9:.1f} GB）")
```

```
序列长度 | 不重算 | FlashAttention / 选择性重算 | 全部重算 | 选择性重算多算
  2048 |    30.6 GB |    9.1 GB |  0.82 GB | 2.6%
  8192 |   380.1 GB |   36.5 GB |  3.29 GB | 8.3%
 32768 |  5643.6 GB |  146.0 GB | 13.15 GB | 19.0%
参数、梯度、优化器状态：16 字节 × 80.3 亿 = 128.5 GB（ZeRO-3 切到 8 卡是 16.1 GB）
```

- s = 2048：不重算 30.6 GB，用 FlashAttention 9.1 GB，全部重算只要 0.82 GB。
- s = 8192：不重算要 380 GB，比参数状态的 128.5 GB 还大三倍。用 FlashAttention 后是 36.5 GB，加上 ZeRO-3 的 16.1 GB，一张 80 GB 的卡勉强放得下；全部重算只要 3.3 GB。
- s = 32768：用 FlashAttention 也要 146 GB，一张卡放不下；全部重算 13 GB。这时要么全部重算，要么用张量并行、序列并行把激活切到多张卡上（下一讲起）。

选择性重计算多花的计算随 s 增长：s = 2048 时只多 2.6%，s = 32768 时多 19%，因为序列越长，注意力在整层计算里的比例越大。

下面的演示可以换模型、调序列长度、micro-batch 和张量并行度，看三种策略的显存：

激活占多少显存：序列越长越吓人

⇆ 对照

Llama-3-8B　Llama-3-70B

不重算　FlashAttention / 选择性重算　全部重算

序列长度 s　8K

micro-batch b　1

张量并行 t　1



Llama-3-8B，s = 8192，b = 1，张量并行 1 路：每卡激活 **36.5 GB**，计算多 **8.3%**。灰色是参数、梯度、优化器状态（16 字节/参数，ZeRO-3 切到 8 路数据并行后每卡 16.1 GB）。
每层里，和 s 成正比的部分 34·s·b·h = 1.14 GB，注意力分数那一项 5·a·s²·b = 10.74 GB：s 大了以后它是大头，而且随 s² 涨。

图 104-1　每卡的显存：灰色是参数、梯度、优化器状态（ZeRO-3 切到 8 卡），琥珀色是激活。不重算时，激活随序列长度的平方增长；FlashAttention 去掉平方项；全部重计算只留每层的输入，代价是多算 33%。虚线是一张 80 GB 的卡。

要点

混合精度训练用 BF16 做前向和反向的矩阵乘法，用 FP32 记主参数和优化器状态，合计 16 字节/参数。激活和序列长度成正比，注意力矩阵那一项还和序列长度的平方成正比；FlashAttention 或选择性重计算去掉平方项，全部重计算只留每层的输入，多算约 33%。

到了真实系统里

- **不是全有或全无。**全部重计算可以只对一部分层做：显存还有余量，就少重算几层，少花点计算。Megatron-LM、PyTorch 的 `torch.utils.checkpoint`、FSDP 都提供按层开关的选项。
- **重算省下的显存可以换成更大的 micro-batch。**batch 大了，矩阵乘法更大，GPU 的利用率更高（第 87 讲），有时能把重算多花的时间赚回来。
- **激活还可以挪到 CPU 内存。**前向算完把激活拷出去，反向前再拷回来，和计算重叠。PCIe 带宽有限，适合计算很重的层。

- **推理不需要这些**。推理只做前向，不留激活给反向；推理的显存大头是权重和 KV Cache（第 71 讲）。

容易混淆的地方

- **重计算省的是激活，不是参数状态**。参数、梯度、优化器状态要靠 ZeRO 来切；两者通常一起用。
- **重计算多算的是一次前向，不是一倍**。6N 变 8N，是多 33%，不是多 100%。
- **BF16 不需要损失缩放，FP16 需要**。原因是指数位：BF16 有 8 位，和 FP32 一样；FP16 只有 5 位。

回顾

混合精度训练用 BF16 算、用 FP32 记：FP16 会让 2×10^{-8} 的梯度下溢成 0（要靠损失缩放），BF16 不会；BF16 的权重每步加 10^{-4}，加 1000 步还是 1，所以要 FP32 的主参数，合计 16 字节/参数。激活按每层 $sbh(34 + 5as/h)$ 字节估算，Llama-3-8B 序列 8192 时不重算要 380 GB，FlashAttention 去掉平方项后 36.5 GB，全部重计算只要 3.3 GB、多算 33%。

可序列再长，或者模型再大，一张卡上的激活和权重都还是放不下。有没有办法让每张卡只存一部分权重、只算一部分激活？下一讲：**张量并行**。

第105讲 张量并行

13 神经元与线性层　28 多头注意力　31 前馈网络　39 MQA 与 GQA　100 集合通信原语　102 通信代价模型

第 103 讲的 ZeRO-3 把权重切开存，可算的时候还要拼回完整的矩阵，推理时每一步都要搬一遍权重，太慢。**张量并行**（tensor parallelism，TP）换了一个思路：把一个矩阵乘法本身拆开，**每张卡只用自己那一块权重，算出结果的一部分**，卡与卡之间只交换小得多的激活。

这是推理里最常用的模型切分方式。它最早由 2019 年的 Megatron-LM 系统化地提出。这一讲先看一个矩阵乘法有哪两种切法，再看怎么把 FFN 和注意力切得巧妙，让每层只通信两次。

一个矩阵乘法的两种切法

$Y = X \cdot A$（第 13 讲），X 是 [token 数, d_in]，A 是 [d_in, d_out]。要拆到两张卡上：

按列切 A：卡 0 拿 A 的前一半列，卡 1 拿后一半列。每张卡用完整的 X 乘自己那一半，得到 Y 的一半列：$Y = [X \cdot A_0 \ , X \cdot A_1 \ ]$。每张卡算出的都是**完整的值**，只是只有一部分列。

按行切 A：卡 0 拿 A 的前一半行，卡 1 拿后一半行，对应地 X 也按列切开。每张卡算 $X_i \cdot A_i$，得到的是一个和 Y 形状相同的矩阵，但只是**部分和**：$Y = X_0 \cdot A_0 + X_1 \cdot A_1$，要加起来（All-Reduce）才是完整的 Y。

FFN：先按列，再按行

FFN（第 31 讲）是两个矩阵乘法，中间夹一个激活函数：$Z = \mathrm{ReLU}(X \cdot A) \cdot B$。（Llama 用的 SwiGLU 多一个门控矩阵，切法相同。）

Megatron 的切法是：**A 按列切，B 按行切**。

1. 每张卡用完整的 X 乘 A_i，得到中间结果的第 i 组列。
2. ReLU 是逐元素的，每张卡对自己那几列做就行，**不需要通信**。
3. 中间结果的第 i 组列，正好就是 B 按行切时第 i 张卡需要的输入。每张卡算 $Y_i \cdot B_i$，得到 Z 的部分和。
4. 一次 All-Reduce 把部分和加起来，每张卡都得到完整的 Z。

整个 FFN 只在最后通信一次。

如果反过来，A 按行切，X·A 在每张卡上只是部分和，而 ReLU 是非线性的：ReLU(部分和$_0$) + ReLU(部分和$_1$) ≠ ReLU(部分和$_0$ + 部分和$_1$)。所以在 ReLU 之前就得先 All-Reduce，FFN 要通信两次。

代码用 4 张卡验证了“先列后行”的结果和单卡完全一样，也验证了“A 按行切”时先做 ReLU 再相加是错的。下面的演示用两张卡和小整数，一步步地看。

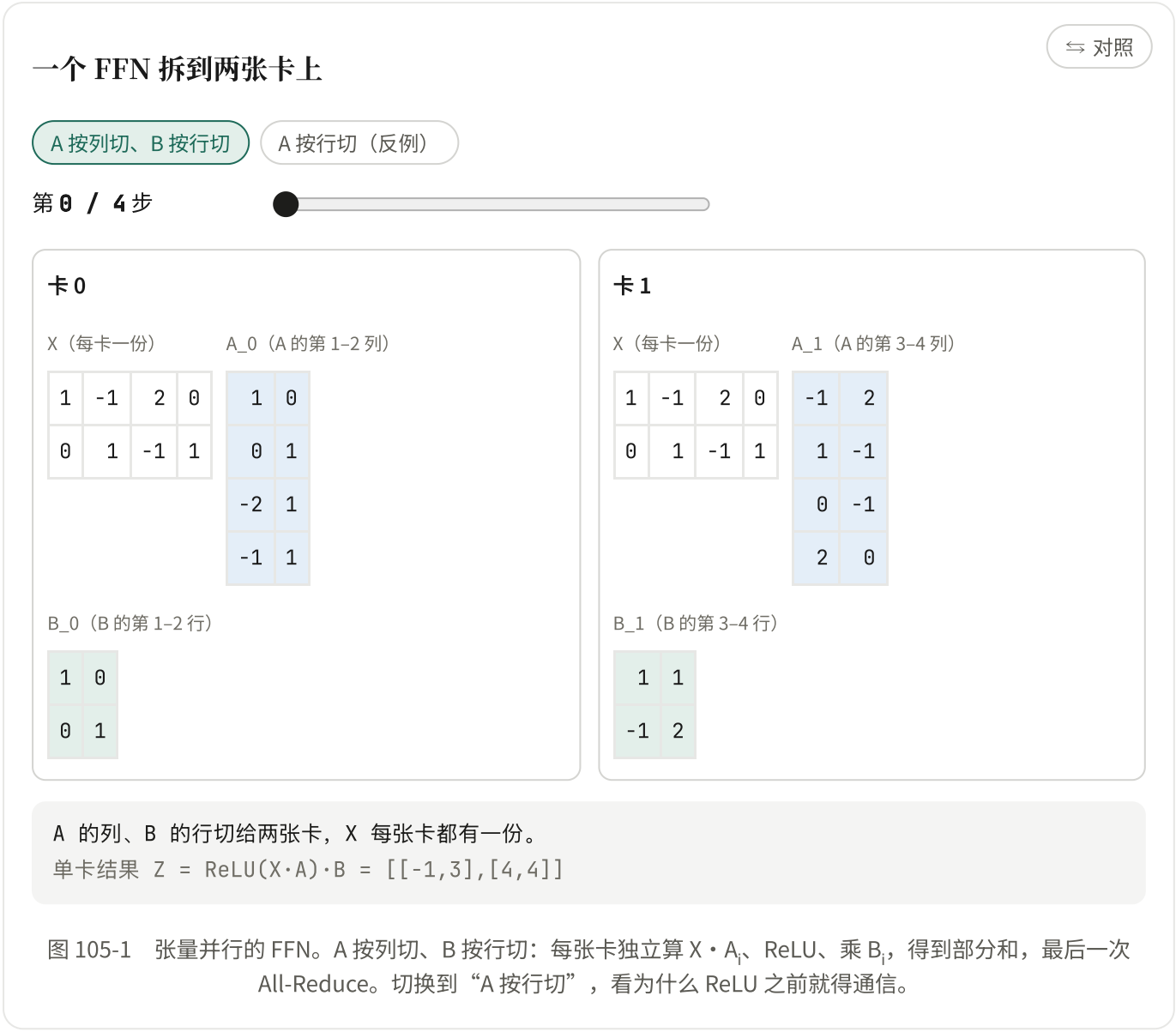


图 105-1　张量并行的 FFN。A 按列切、B 按行切：每张卡独立算 $X \cdot A_i$、ReLU、乘 B_i，得到部分和，最后一次 All-Reduce。切换到“A 按行切”，看为什么 ReLU 之前就得通信。

注意力：按头切

多头注意力（第 28 讲）本来就是好几个独立的头。张量并行就**按头切**：

1. Q、K、V 的投影矩阵**按列切**，每张卡拿到其中几个头对应的列。每张卡算出自己那几个头的 q、k、v。
2. 每个头的注意力计算只用到自己的 q、k、v，各卡独立完成，不需要通信。
3. 输出投影 W_O **按行切**，每张卡用自己那几个头的输出乘 W_O 对应的行，得到部分和。
4. 一次 **All-Reduce**。

和 FFN 一模一样：先列后行，最后通信一次。所以**每个 Transformer 层的前向只需要 2 次 All-Reduce**：注意力之后一次，FFN 之后一次。

按头切还有一个好处：**KV Cache 也自然地切开了**。每张卡只存自己那几个头的 K、V，KV Cache 的显存也分成了 t 份，并发能力随卡数增加。

不过有个限制：Llama-3.1-70B 用 GQA（第 39 讲），只有 8 个 KV 头。TP = 8 时每张卡正好分到 1 个 KV 头；TP 再大，KV 头不够分，只能在几张卡之间复制，KV Cache 就不再随卡数减少了。

要点

张量并行把每个矩阵乘法拆到 t 张卡上：FFN 的第一个矩阵按列切、第二个按行切，注意力的 QKV 按头（按列）切、输出投影按行切。每张卡只用自己那块权重独立计算，每层前向只在注意力和 FFN 之后各做一次 All-Reduce。KV Cache 也随头一起切开。

收益与代价

每张卡只存、只读 1/t 的权重，decode 时读权重的时间降到 1/t。可每层 2 次 All-Reduce 的次数不随 t 减少，每次同步的数据量也不小。代码算了 Llama-3.1-70B、32 个请求 decode 一步（用第 102 讲的 NVLink 数字和一步到位的 All-Reduce）：

- **TP = 1**：每卡 141 GB（放不下），读权重 42.1 ms。
- **TP = 2**：70.5 GB，读 21.0 ms，通信 0.67 ms，合计 21.7 ms。
- **TP = 4**：35.2 GB，读 10.5 ms，通信 1.04 ms，合计 11.6 ms。
- **TP = 8**：17.6 GB，读 5.3 ms，通信 1.78 ms，合计 **7.0 ms**。

卡数翻倍，读权重的时间减半，通信却在增加，所占的比例从 3% 涨到了 25%。如果 TP 跨出一台机器，走 InfiniBand，α 更大、β 小了九倍，通信很快就会超过计算。所以**张量并行通常只在一台机器内部做**（最多 8 张卡），更大的规模要和其他并行方式组合（第 106、109 讲）。

到了大模型里

推理引擎的默认选择。vLLM 和 SGLang 用 `tensor_parallel_size`（`--tp`）设置张量并行。70B 级别的模型在 8 张 H100 上用 TP = 8，是最常见的部署之一。模型代码里的 `ColumnParallelLinear`、`RowParallelLinear` 就是这一讲的两种切法。

嵌入层和输出层。词表有十几万，嵌入表和输出层（第 33 讲）也很大，通常按词表切开（vocab parallel），每张卡负责一部分词，最后再合并。

通信和计算重叠。每层的 All-Reduce 要等这一层算完才能开始，下一层又要等它结束，很难完全藏起来。一些实现会把 All-Reduce 和下一步的 RMSNorm 融合，或者把一个批次切成两半交替计算和通信，尽量减少等待。

训练里也一样。训练时反向传播也要对应地做 All-Reduce，每层前向 2 次、反向 2 次。训练的激活更大，第 107 讲的序列并行会进一步优化它。

用代码复现

下面的代码先用 NumPy 验证 FFN 的“A 按列、B 按行”切法与单卡结果一致，并演示 A 按行切时先做 ReLU 是错的；再估算 Llama-3.1-70B 在不同 TP 卡数下 decode 一步的读权重时间和通信时间。

```python
import numpy as np
rng = np.random.default_rng(0)

# 1. 一个 FFN: Z = ReLU(X·A)·B。A 按列切、B 按行切到 t 张卡上，只需要最后一次 All-Reduce
d, dff, t = 8, 32, 4
X = rng.standard_normal((5, d)); A = rng.standard_normal((d, dff)); B =
rng.standard_normal((dff, d))
relu = lambda v: np.maximum(v, 0)
Z = relu(X @ A) @ B
A_parts, B_parts = np.split(A, t, axis=1), np.split(B, t, axis=0)       # 卡 i: A 的
第 i 组列，B 的第 i 组行
partial = [relu(X @ A_parts[i]) @ B_parts[i] for i in range(t)]          # 每张卡独
立算，不用通信
print("A 按列切、B 按行切，各卡部分和 All-Reduce 后 = 原结果: ",
np.allclose(sum(partial), Z))
print("每张卡上: X·A_i 的形状", (X @ A_parts[0]).shape, "，部分和的形状",
partial[0].shape)
# 反过来: A 按行切，X·A 只能得到部分和，ReLU 之前必须先加起来——多一次通信
bad = [X[:, i * 2:(i + 1) * 2] @ A[i * 2:(i + 1) * 2] for i in range(t)]
print("如果 A 按行切: ReLU(部分和) 之和 ≠ ReLU(总和): ", not np.allclose(sum(relu(b)
for b in bad), relu(X @ A)))

# 2. Llama-3.1-70B decode 一步（32 个请求），张量并行 t 张卡: 每卡读的权重变少，通信次数不变
W, L, dm, BW = 141e9, 80, 8192, 3.35e12
a, beta = 3e-6, 450e9                                          # NVLink（第 102 讲的示意数
字），用一步到位的 All-Reduce
m = 32 * dm * 2
print("TP 卡数 | 每卡权重 | 读权重 | 通信 (160 次 All-Reduce)  | 合计")
for tp in [1, 2, 4, 8]:
    read = W / tp / BW
    comm = 0 if tp == 1 else 2 * L * (a + (tp - 1) * m / beta)   # 每卡同时读其余
tp-1 张卡的部分和
    print(f"{tp:5d} | {W / tp / 1e9:6.1f} GB | {read * 1e3:5.1f} ms | {comm *
1e3:5.2f} ms | {(read + comm) * 1e3:5.1f} ms")
```

输出:

```
A 按列切、B 按行切，各卡部分和 All-Reduce 后 = 原结果:  True
每张卡上：X·A_i 的形状 (5, 8)，部分和的形状 (5, 8)
如果 A 按行切：ReLU(部分和) 之和 ≠ ReLU(总和):  True
TP 卡数 | 每卡权重 | 读权重 | 通信 (160 次 All-Reduce)  | 合计
     1 |  141.0 GB |  42.1 ms |  0.00 ms |  42.1 ms
     2 |   70.5 GB |  21.0 ms |  0.67 ms |  21.7 ms
     4 |   35.2 GB |  10.5 ms |  1.04 ms |  11.6 ms
     8 |   17.6 GB |   5.3 ms |  1.78 ms |   7.0 ms
```

前三行是验证：4 张卡的部分和加起来等于原结果；每张卡的中间结果是 5 × 8（32 列里的 8 列）；A 按行切时先 ReLU 再相加不等于正确结果。表格是 70B 的估算：TP = 8 时一步约 7.0 ms，其中通信 1.78 ms。

容易混淆的地方

1. **张量并行要求每张卡同时参与每一层**。每层都要 All-Reduce，任何一张卡慢了，所有卡都要等。所以它要求卡之间的连接又快又稳，适合机内。
2. **它不减少每张卡要算的 token 数**。每张卡都处理全部的 token，只是每个 token 只算一部分权重。这和数据并行（每张卡处理一部分 token）正好相反。
3. **TP 不能超过头数的约束**。按头切要求头数能被 t 整除；KV 头数小于 t 时，KV 头要复制，KV Cache 的切分效果打折扣。

回顾

张量并行把每个矩阵乘法拆到多张卡上，每张卡只用自己那块权重计算。矩阵按列切得到完整值的部分列，按行切得到部分和。FFN 先按列、再按行，中间的激活函数各卡独立做，最后一次 All-Reduce；注意力按头切 QKV、按行切输出投影，也是最后一次 All-Reduce。每层前向 2 次 All-Reduce，KV Cache 也随头切开。Llama-3.1-70B、TP = 8 时，每卡权重 17.6 GB，decode 一步读权重 5.3 ms、通信 1.78 ms；卡数越多，通信占比越高，所以张量并行一般限制在一台机器内部。

张量并行每层都要通信，跨机就太慢了。还有一种切法：不拆矩阵，而是把不同的层放到不同的卡上，卡与卡之间只在层的交界处传一次激活。下一讲：**流水并行**。

第106讲 流水并行

18 反向传播　32 Transformer Block　100 集合通信原语　105 张量并行

上一讲的张量并行每层都要 All-Reduce，适合连接很快的机内。可 405B 这样的模型，一台 8 卡的机器装不下，必须跨机。跨机的网络慢了一个数量级，每层都通信就太慢了。

流水并行（pipeline parallelism，PP）换一种切法：不拆矩阵，而是**按层切**。一个 80 层的模型放到 4 张卡上，卡 0 放第 1 到 20 层，卡 1 放第 21 到 40 层……数据从卡 0 流到卡 3，就像工厂的流水线。卡与卡之间只在交界处传一次激活，而且是点对点的 Send/Recv（第 100 讲），通信量很小。

这一讲讲流水并行的核心问题：**气泡**，以及训练和推理里它分别是什么样子。

只有一份数据时，大家轮流闲着

假设只有一批数据。卡 0 算完自己的 20 层，把结果传给卡 1；这时卡 1 才开始算，卡 0 就闲着了；卡 1 算完传给卡 2……前向走到卡 3，再开始反向（第 18 讲），梯度又从卡 3 一路传回卡 0。

任何时刻只有一张卡在干活，其他卡都在等。4 张卡的利用率只有 25%，空等的部分叫**气泡**（bubble）。

切成微批次

解决办法是把一个批次切成 m 个**微批次**（micro-batch），让它们像流水线上的零件一样依次进入：卡 0 算完微批次 0 交给卡 1，马上开始算微批次 1；卡 1 算微批次 0 的同时，卡 0 在算微批次 1……流水线填满以后，所有卡都在同时工作。

最直接的调度是 **GPipe**（2019 年）：所有微批次先依次做完前向，再依次做完反向，最后统一更新参数。气泡只出现在流水线“灌满”和“排空”的时候。设前向一段耗时 1、反向耗时 2，代码模拟了 4 段的情况：

- m = 1：总时长 12，气泡占 75%。
- m = 4：总时长 21，气泡 42.9%。

- m = 16：总时长 57，气泡 15.8%。

气泡占比有一个简洁的公式：

$$\text{气泡占比} = (p - 1) \div (m + p - 1)$$

p 是段数，m 是微批次数。微批次越多，灌满和排空的那一小段相对越短。所以流水并行要求 m 远大于 p。

1F1B：少存一些激活

GPipe 有个问题：所有微批次的前向都做完了才开始反向，每张卡要同时存着 **m 份**激活，等着反向用。m 越大气泡越小，激活却越多，显存吃不消。

1F1B（one forward, one backward）调度改了顺序：第 s 段卡先做 p − s − 1 个前向“热身”，之后每做一个前向，就紧接着做一个反向，交替进行。一个微批次的反向做完，它的激活就可以释放。这样每张卡同时存的激活**最多 p 份**，和 m 无关。

代码里 m = 16 时，GPipe 的卡 0 要同时存 16 份激活，1F1B 只要 4 份；而两者的总时长、气泡占比完全一样。

下面的甘特图画出了每张卡在每个时刻做什么。蓝色是前向，琥珀色是反向，灰色是气泡。调一调段数和微批次数，切换两种调度。

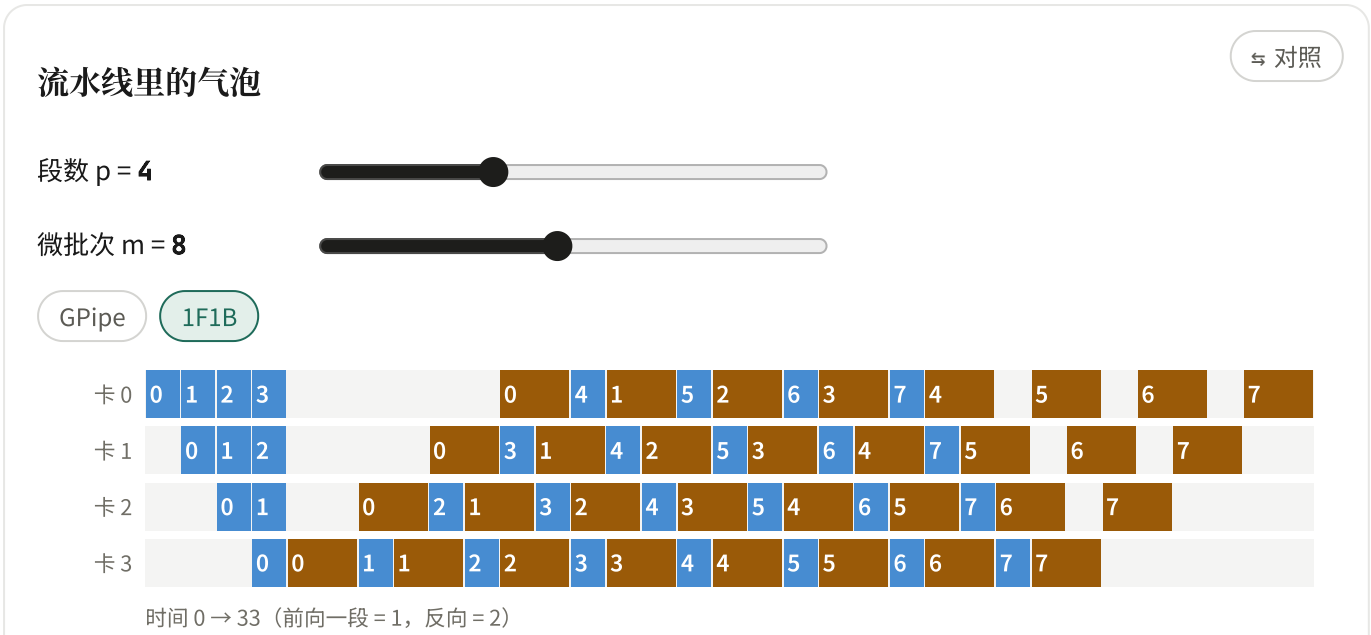


图 106-1　流水并行的调度。前向（蓝）从卡 0 流向最后一张卡，反向（琥珀）流回来；灰色是空等的气泡。微批次越多，气泡占比越小；1F1B 和 GPipe 的总时长一样，但每张卡同时存的激活少得多。

要点

流水并行按层把模型切成 p 段，段之间只点对点传激活，通信量小，适合跨机。代价是气泡：一个批次切成 m 个微批次，气泡占比 (p − 1)/(m + p − 1)，要让 m 远大于 p。1F1B 调度让前向、反向交替进行，每张卡同时存的激活从 m 份降到最多 p 份。

推理里的流水并行

推理没有反向，也不用等一整个批次结束再更新参数。连续批处理（第 74 讲）源源不断地有请求进来，可以把正在运行的请求分成几组，像微批次一样依次送进流水线：卡 0 算第 1 组的这一步时，卡 1 在算第 2 组的上一步……只要组数不少于段数，流水线就一直是满的，**吞吐**几乎没有损失。

可是**延迟**不一样。对单个请求来说，它的每个 token 都要依次走过所有的段，一段一段地读权重、计算。Llama-3.1-70B 放在 4 张卡上：

- **流水并行**：一个请求 decode 一步，要依次走完 4 段，每段读 1/4 的权重，加起来还是读了一遍全部权重，约 **42.1 ms**。
- **张量并行**：4 张卡同时各读 1/4，约 **10.5 ms**，再加上通信。

流水并行让模型装得下、让吞吐上得去，但**不能缩短单个请求的延迟**；张量并行能。这是两者在推理中最大的区别。

流水并行的好处是通信极少：段与段之间每步只传 32 个 token 的隐藏状态，约 524 KB，跨机走 InfiniBand 也只要 16 μs 左右。

到了大模型里

和张量并行组合。典型的做法是机内用张量并行、机间用流水并行。比如 Llama-3.1-405B 用两台 8 卡的机器：每台机器内部 TP = 8，两台之间 PP = 2。vLLM、SGLang 都支持这样组合（`--tp 8 --pp 2`）。第 109 讲会系统地讨论这种映射。

训练里的更多花样。大规模训练常用更复杂的调度来压缩气泡：比如每张卡放好几段不相邻的层（交错式 1F1B，interleaved），或者把反向拆成“算输入的梯度”和“算权重的梯度”两部分分开安排（零气泡流水线）。DeepSeek-V3 的训练用了它们自己设计的 DualPipe，让前向、反向和通信相互重叠。

层数要能均分。每段的层数和计算量最好相等，否则最慢的那一段决定了整条流水线的速度。第一段有嵌入层、最后一段有输出层，有时要少分几层来平衡。

用代码复现

下面的代码模拟 GPipe 和 1F1B 两种调度，统计 4 段、不同微批次数时的总时长、气泡占比和第 0 段同时存的激活份数；最后比较推理时流水并行和张量并行的单请求延迟，以及段与段之间的通信量。

```
# 流水并行：模型按层切成 p 段（每段一张卡），一个批次切成 m 个微批次。前向一段耗时 1，反向耗时
2
def order(s, p, m, sched):
    """第 s 段卡上的执行顺序。GPipe：先做完所有前向，再做所有反向；1F1B：先做 p-s-1 个前向热
身，然后一前一后交替"""
    if sched == "GPipe":
        return [("F", i) for i in range(m)] + [("B", i) for i in range(m)]
    warm = min(p - s - 1, m); seq = [("F", i) for i in range(warm)]
    f, b = warm, 0
    while b < m:
        if f < m: seq.append(("F", f)); f += 1
        seq.append(("B", b)); b += 1
    return seq

def simulate(p, m, sched):
    end, free, peak = {}, [0.0] * p, [0] * p
    seqs = [order(s, p, m, sched) for s in range(p)]
    pos = [0] * p
    while any(pos[s] < len(seqs[s]) for s in range(p)):
        for s in range(p):
            if pos[s] == len(seqs[s]): continue
            kind, i = seqs[s][pos[s]]
            dep = ("F", s - 1, i) if kind == "F" and s > 0 else ("B", s + 1, i) if
kind == "B" and s < p - 1 else ("F", s, i) if kind == "B" else None
            if dep and dep not in end: continue                          # 依赖还没算
完，这张卡先等
            start = max(free[s], end[dep] if dep else 0)
            end[(kind, s, i)] = free[s] = start + (1 if kind == "F" else 2)
            pos[s] += 1
            done_f = sum(1 for k in seqs[s][:pos[s]] if k[0] == "F"); done_b =
pos[s] - done_f
            peak[s] = max(peak[s], done_f - done_b)                  # 这张卡同时存着
几个微批次的激活
    T = max(free)
    return T, 1 - p * m * 3 / (p * T), peak[0]

p = 4
print(f"p = {p} 段: ")
for m in [1, 4, 8, 16]:
    for sched in ["GPipe", "1F1B"]:
        T, bubble, pk = simulate(p, m, sched)
        print(f"  m = {m:2d}, {sched:5s}: 总时长 {T:4.0f}, 气泡占 {bubble:5.1%} (公式
(p-1)/(m+p-1) = {(p - 1) / (m + p - 1):5.1%}) , 第 0 段最多同时存 {pk} 份激活")

# 推理：Llama-3.1-70B 放在 4 张卡上，decode 一个请求，比较流水并行和张量并行
read = 141e9 / 3.35e12 * 1e3                                     # 单卡读一遍全部权重的时间
```

```
print(f"一个请求 decode 一步：流水并行 4 段要依次走完 = {read:.1f} ms；张量并行 4 卡同时读 ≈ {read / 4:.1f} ms（再加通信）")
print(f"段与段之间每步只传 32 个 token 的隐藏状态：{32 * 8192 * 2 / 1e3:.0f} KB，跨机传一次约 {(6e-6 + 32 * 8192 * 2 / 50e9) * 1e6:.0f} μs")
```

输出：

```
p = 4 段：
  m =  1，GPipe：总时长   12，气泡占 75.0%（公式 (p-1)/(m+p-1) = 75.0%），第 0 段最多同时存 1 份激活
  m =  1，1F1B ：总时长   12，气泡占 75.0%（公式 (p-1)/(m+p-1) = 75.0%），第 0 段最多同时存 1 份激活
  m =  4，GPipe：总时长   21，气泡占 42.9%（公式 (p-1)/(m+p-1) = 42.9%），第 0 段最多同时存 4 份激活
  m =  4，1F1B ：总时长   21，气泡占 42.9%（公式 (p-1)/(m+p-1) = 42.9%），第 0 段最多同时存 4 份激活
  m =  8，GPipe：总时长   33，气泡占 27.3%（公式 (p-1)/(m+p-1) = 27.3%），第 0 段最多同时存 8 份激活
  m =  8，1F1B ：总时长   33，气泡占 27.3%（公式 (p-1)/(m+p-1) = 27.3%），第 0 段最多同时存 4 份激活
  m = 16，GPipe：总时长   57，气泡占 15.8%（公式 (p-1)/(m+p-1) = 15.8%），第 0 段最多同时存 16 份激活
  m = 16，1F1B ：总时长   57，气泡占 15.8%（公式 (p-1)/(m+p-1) = 15.8%），第 0 段最多同时存 4 份激活
一个请求 decode 一步：流水并行 4 段要依次走完 = 42.1 ms；张量并行 4 卡同时读 ≈ 10.5 ms（再加通信）
段与段之间每步只传 32 个 token 的隐藏状态：524 KB，跨机传一次约 16 μs
```

前八行是调度的模拟：模拟出的气泡占比和公式完全一致；m 大于 p 以后，1F1B 存的激活始终是 4 份，GPipe 随 m 增加。最后两行是推理：流水并行一步 42.1 ms，张量并行约 10.5 ms；段间每步只传 524 KB。

容易混淆的地方

1. **流水并行不减少单个请求的计算路径。**每个 token 仍然要依次经过所有的层，只是这些层分布在不同的卡上。它提高的是同时处理多份数据的能力。
2. **微批次不改变训练的结果。**所有微批次的梯度累加起来再更新，等价于用整个批次算一次梯度（第 20 讲的梯度累加）。
3. **气泡公式假设每段耗时相同。**段与段不均衡时，实际的气泡更大。

回顾

流水并行按层把模型切成 p 段，段之间只点对点地传激活，通信量小，适合跨机。只有一份数据时，各段轮流工作，气泡很大；把批次切成 m 个微批次依次送入，气泡占比降到 $(p-1)/(m+p-1)$：4 段时 m = 1 为 75%，m = 16 为 15.8%。1F1B 调度让前向和反向交替，每段同时存的激活最多 p 份。推理时连续批处理能让流水线保持满载，吞吐不受影响，但单个请求的延迟不会缩短：70B 放 4 卡，流水并行一步 42.1 ms，张量并行约 10.5 ms。实际部署常常机内张量并行、机间流水并行。

张量并行和流水并行切的都是模型。还有一个维度没有切：序列。上下文长到几十万、上百万个 token 时，一张卡连一个请求的激活和 KV Cache 都放不下。序列怎么切？下一讲：**序列并行与上下文并行**。

第107讲 序列与上下文并行

选读

42 长上下文技术　84 FlashAttention　85 Decode 注意力 Kernel　100 集合通信原语　105 张量并行

前两讲切的都是模型：张量并行切矩阵，流水并行切层。可是还有一种情况，模型本身不大，**序列**却很长。第 42 讲讲过，今天的模型支持几十万甚至上百万 token 的上下文。Llama-3-8B 一个 100 万 token 的请求，光 KV Cache 就有 **122 GiB**，一张卡根本放不下；prefill 的注意力计算量随长度平方增长，一张卡要算很久。

这一讲讲沿着序列切开的两种做法：训练里配合张量并行的**序列并行**，以及把一条长序列分到多张卡上算注意力的**上下文并行**。

序列并行：张量并行里没切开的那部分

先说一个容易被忽略的问题。第 105 讲的张量并行切开了注意力和 FFN 的计算，但每一层里还有 RMSNorm、残差相加这些部分没有切：每张卡都对**完整的**激活做同样的计算，存同样的一份。训练时这些激活要留到反向用，t 张卡就存了 t 份一模一样的东西。

2022 年 Megatron 的**序列并行**（sequence parallelism，SP）把这部分沿序列切开：每张卡只对 1/t 的 token 做 RMSNorm 和残差。进入注意力或 FFN 之前，用 **All-Gather** 把完整的序列拼回来；算完之后，原本的 All-Reduce 换成 **Reduce-Scatter**，求和的同时就按序列切开。

第 100 讲说过，All-Reduce = Reduce-Scatter + All-Gather，所以通信总量**一点没变**，只是拆成两半放在了不同的位置，而这部分激活的显存降到了 1/t。这几乎是免费的优化，训练里和张量并行一起默认开启。

上下文并行：一条序列分给几张卡

序列并行只处理了逐 token 的部分，注意力本身还是要看完整的序列。当一条序列长到一组张量并行都放不下时，就要把**注意力也沿序列切开**，这叫**上下文并行**（context parallelism，CP）。

把序列切成 P 段，卡 i 存第 i 段 token 的 Q、K、V。难点在于，第 i 段的查询要和**所有**段的键做注意力，而别的段的 K、V 在别的卡上。

Ring Attention（2023 年）的做法是让 K、V 在卡之间沿环转一圈：

1. 第 1 步，每张卡用自己的 Q 和自己的 K、V 算一块注意力；同时把自己的 K、V 发给下一张卡。
2. 第 2 步，每张卡收到了上一张卡的 K、V，用自己的 Q 和它算一块；再把它传下去。
3. 如此 P 步，每张卡的 Q 见过了所有段的 K、V。

每一步只算出注意力的一部分，怎么合并？就是第 84 讲的**在线 softmax**：每张卡为自己的查询维护最大值 m、指数和 l、加权和 acc，每来一块就更新一次，最后输出 acc / l，和一次算完完全相同。代码用 4 张卡、32 个 token 验证了这一点，带不带因果掩码都对。

传 K、V 的同时可以计算当前这一块，只要每块的计算时间比传输时间长，通信就被完全藏住了。序列越长，每块的计算量（随长度平方增长）相对传输量（线性增长）越大，越容易藏住。

因果掩码带来的不均衡

加上因果掩码（第 27 讲）以后出现了一个问题：每个 token 只看它前面的 token。如果按顺序切，卡 0 拿最前面的 8 个 token，它们前面几乎没有别的 token，大部分块都被掩码挡掉了，计算量很少；卡 3 拿最后 8 个 token，要看前面所有的 token，计算量最大。代码数出来的各卡计算量是 **36、100、164、228**，最忙的卡是最闲的六倍多。所有卡都要等最慢的那张，效率很低。

解决办法是**之字形切分**（zigzag）：把序列切成 2P 段，卡 i 拿第 i 段和倒数第 i 段。每张卡都分到一段靠前的（活少）和一段靠后的（活多），忙闲搭配。代码里，之字形切分后各卡的计算量都是 **132**，完全均衡，结果也不变。

下面的演示画出了 4 张卡在 4 步中各自算哪一块。行是查询，列是键，颜色表示归哪张卡，深色是这一步正在算的块。

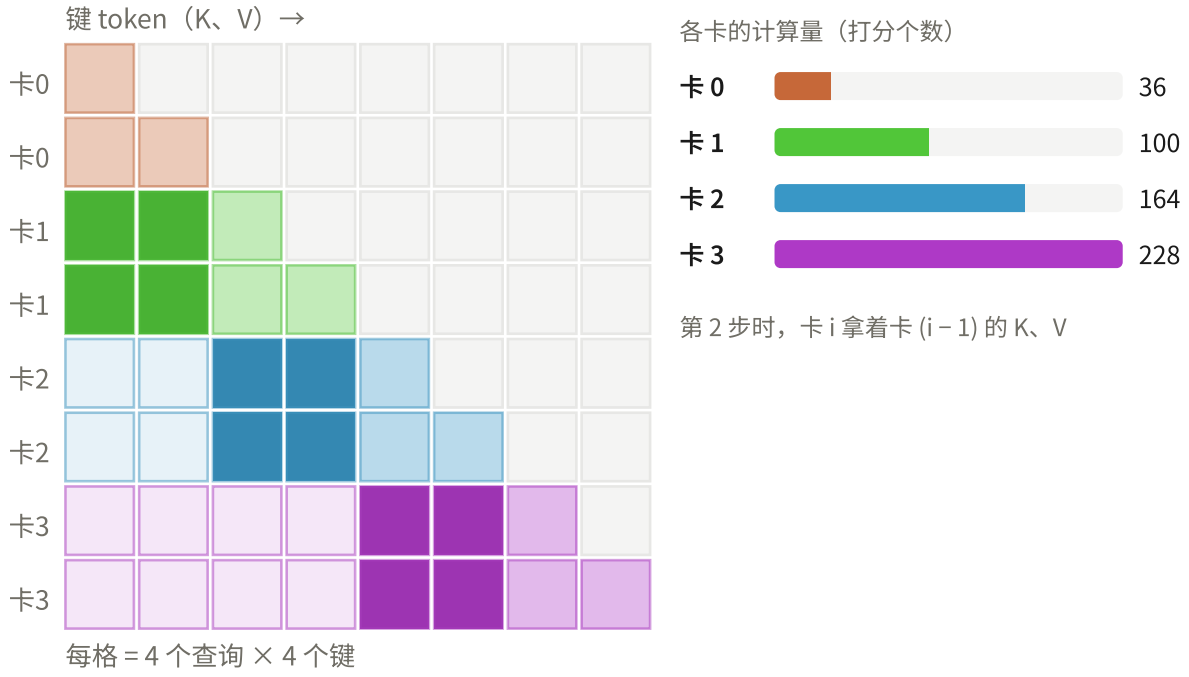
K、V 沿着环转一圈

⇆ 对照

第 2 / 4 步

因果掩码 | 不带掩码

按顺序切 | 之字形切



深色是这一步各卡正在算的块：每张卡用自己的 Q 和刚收到的 K、V 算一块，同时把这份 K、V 传给下一张卡。

各卡计算量：36、100、164、228，最多是最少的 6.3 倍。

按顺序切时，卡 0 的 token 在最前面，大部分键都被掩码挡掉，很快就闲了；卡 3 的 token 在最后，要看所有的键，最忙。

图 107-1　Ring Attention。每一步各卡用自己的 Q 和手里的 K、V 算一块，同时把 K、V 传给下一张卡，4 步后都看过了全部的键。加上因果掩码时，按顺序切会让后面的卡最忙；之字形切分让各卡的计算量相同。

要点

序列并行把张量并行中没切开的逐 token 运算（RMSNorm、残差）沿序列切开，All-Reduce 换成 Reduce-Scatter + All-Gather，通信不变、激活显存降为 1/t。上下文并行把注意力也沿序列切开：Ring Attention 让 K、V 沿环传 P 步，每步用在线 softmax 累加，传输和计算重叠；因果掩码下要用之字形切分来平衡负载。

另一种思路：按头换着切

DeepSpeed 的 **Ulysses** 用了不同的办法：注意力之前，用一次 **All-to-All**（第 100 讲）把“每张卡有一段序列的所有头”换成“每张卡有完整序列的一部分头”；这样每张卡就能对自己那几个头做完整的注意力（可以直接用 FlashAttention）；算完再用一次 All-to-All 换回来。它的通信量和头数有关，并行度不能超过头数（GQA 的 KV 头更少）；Ring Attention 没有这个限制。实际系统常把两者组合使用。

到了大模型里

训练长上下文模型离不开它。训练 128K 甚至 1M 的上下文时，激活和注意力的计算都太大，Megatron、DeepSpeed 等框架都提供上下文并行（或 Ulysses），和张量、流水、数据并行组合起来用。

推理的长上下文 prefill。一个几十万 token 的 prefill，在一组卡上要算好几秒甚至更久。上下文并行能把它分到更多的卡上，缩短 TTFT。推理引擎正在陆续支持这种方式。

decode 时 KV Cache 也可以按序列切。100 万 token 的 KV 有 122 GiB，按序列分到 8 张卡上，每张 15 GiB。decode 时，每张卡用新 token 的 q 和自己那段 KV 算出部分的 (m, l, acc)，再在卡之间合并，这和第 85 讲的 Flash-Decoding 是同一个思路，只是从一张卡里的多个 SM 扩展到了多张卡。

通信量不小。按顺序传 K、V，每张卡每层每步要传自己那段的 K、V：100 万 token 分 8 卡、Llama-3-8B，每层约 488 MiB。所以上下文并行通常放在 NVLink 这样的快速连接上，并依赖计算和传输的重叠。

用代码复现

下面的代码实现了 Ring Attention：4 张卡、32 个 token，K、V 沿环传 4 步，用在线 softmax 累加，分别验证不带掩码和带因果掩码时与完整注意力的结果一致；比较按顺序切和之字形切的各卡计算量；最后算 100 万 token 的 KV Cache 和每层要传的数据量。

```python
import numpy as np
rng = np.random.default_rng(0)

# 1. Ring Attention：序列切成 P 段，卡 i 存第 i 段的 Q、K、V。K、V 沿环传 P 步，每步用在线 softmax（第 84 讲）累加
P, n, d = 4, 32, 8
Q, K, V = rng.standard_normal((3, n, d))
def full_attn(causal):
    s = Q @ K.T / np.sqrt(d)
    if causal: s = np.where(np.tril(np.ones((n, n))) > 0, s, -np.inf)
    e = np.exp(s - s.max(1, keepdims=True)); return e / e.sum(1, keepdims=True) @ V

def ring_attn(causal, chunks):
    """chunks[i] = 卡 i 负责的 token 下标。第 t 步，卡 i 拿到的是卡 (i - t) 的 K、V"""
    out = np.zeros_like(Q); work = [0] * P
    for i in range(P):
        qi = chunks[i]; m = np.full(len(qi), -np.inf); l = np.zeros(len(qi)); acc = np.zeros((len(qi), d))
        for t in range(P):
            kj = chunks[(i - t) % P]
            s = Q[qi] @ K[kj].T / np.sqrt(d)
            if causal: s = np.where(qi[:, None] ⩾ kj[None, :], s, -np.inf)
            work[i] += int(np.isfinite(s).sum())                          # 这一块里真正要算的打分个数
            if not np.isfinite(s).any(): continue
            m_new = np.maximum(m, s.max(1)); sc = np.exp(m - m_new); e = np.exp(s - m_new[:, None])
            l = l * sc + e.sum(1); acc = acc * sc[:, None] + e @ V[kj]; m = m_new
        out[qi] = acc / l[:, None]
    return out, work

contig = np.split(np.arange(n), P)                                        # 按顺序切：卡 0 拿最前面的 8 个 token
half = np.split(np.arange(n), 2 * P)
zigzag = [np.concatenate([half[i], half[2 * P - 1 - i]]) for i in range(P)]   # 之字形：卡 i 拿第 i 段和倒数第 i 段
o, _ = ring_attn(False, contig); print("Ring Attention（不带掩码）= 完整注意力：", np.allclose(o, full_attn(False)))
o, w1 = ring_attn(True, contig); print("Ring Attention（因果掩码）= 完整注意力：", np.allclose(o, full_attn(True)), "；按顺序切，各卡的计算量：", w1)
o, w2 = ring_attn(True, zigzag); print("之字形切分，结果一样：", np.allclose(o, full_attn(True)), "；各卡的计算量：", w2)

# 2. 为什么要切序列：Llama-3-8B 一个 100 万 token 的请求
kv = 1_000_000 * 131072
print(f"100 万 token 的 KV Cache：{kv / 2**30:.0f} GiB；上下文并行 8 卡，每卡 {kv / 8 /
```

```
2**30:.0f} GiB")
print(f"每卡每步要传给下一张卡的 K、V（每层）：{1_000_000 / 8 * 2 * 8 * 128 * 2 /
2**20:.0f} MiB")
```

输出：

```
Ring Attention（不带掩码）= 完整注意力： True
Ring Attention（因果掩码）= 完整注意力： True ；按顺序切，各卡的计算量： [36, 100, 164,
228]
之字形切分，结果一样： True ；各卡的计算量： [132, 132, 132, 132]
100 万 token 的 KV Cache：122 GiB；上下文并行 8 卡，每卡 15 GiB
每卡每步要传给下一张卡的 K、V（每层）：488 MiB
```

前三行是验证和计算量：两种切法的结果都正确，按顺序切的计算量是 36、100、164、228，之字形切是四个 132。最后两行是 100 万 token 的规模：KV 122 GiB，8 卡各 15 GiB，每层每步传 488 MiB。

容易混淆的地方

1. **“序列并行”这个词有好几种用法**。Megatron 的序列并行专指张量并行里切逐 token 运算的那个优化；也有人把上下文并行、Ulysses 都叫序列并行。读文档时要看它具体切了什么。
2. **上下文并行不改变注意力的结果**。它和 FlashAttention 一样，靠在线 softmax 精确合并，不是近似。
3. **之字形切分只在因果掩码下需要**。双向注意力（比如编码器）每个查询都看所有的键，按顺序切就已经均衡了。

回顾

序列很长时，激活和 KV Cache 一张卡放不下，注意力的计算量也随长度平方增长。Megatron 的序列并行把张量并行里没切开的 RMSNorm、残差沿序列切开，All-Reduce 换成 Reduce-Scatter + All-Gather，通信不变、激活显存降到 1/t。上下文并行把注意力也沿序列切开：Ring Attention 让 K、V 沿环传 P 步，每步用在线 softmax 累加，结果和完整注意力相同；因果掩码下，按顺序切的计算量是 36、100、164、228，之字形切分后均为 132。Ulysses 用 All-to-All 在“按序列切”和“按头切”之间切换。

到这里，稠密模型能切的维度都切过了：数据、矩阵、层、序列。MoE 模型还多了一个维度：专家。DeepSeek-V3 有 256 个路由专家，每个 token 只去其中 8 个。专家怎么分到各张卡上，token 又怎么找到自己的专家？下一讲：**专家并行与 All-to-All**。

第108讲 专家并行与 All-to-All

重点

41 MoE 混合专家　100 集合通信原语　102 通信代价模型　105 张量并行

第 41 讲的 MoE（混合专家）让每个 token 只经过少数几个专家：DeepSeek-V3 每层有 256 个路由专家，每个 token 选 8 个。总参数 6710 亿，每个 token 只用到约 370 亿。

计算量小了，可**参数一个也没少**。一层 256 个专家，每个专家约 4400 万参数，一层就是 113 亿；58 个 MoE 层，加起来六千多亿。这么多专家怎么放？如果用张量并行，把每个专家都切成几份，每个专家本来就很小，切开后的矩阵乘法更小，GPU 根本跑不满。

更自然的办法是**专家并行**（expert parallelism，EP）：**把不同的专家放在不同的卡上**，每张卡放完整的几个专家。这一讲看 token 怎么找到自己的专家，用的正是第 100 讲的 All-to-All，以及专家并行特有的难题：负载不均衡。

专家分到各张卡上

举一个小例子：4 张卡，8 个专家，卡 g 放专家 2g 和 2g + 1。注意力层和其他部分照常（比如每张卡用数据并行各处理自己的一批 token）。每张卡上有 8 个 token，每个 token 由路由器（第 41 讲）选出 top-2 个专家。

问题是，卡 0 上的某个 token 可能选了专家 5 和专家 6，它们在卡 2 和卡 3 上。token 得过去找专家。

一个 MoE 层的流程是：

1. **路由**：每张卡对自己的 token 算路由打分，选出每个 token 的 top-k 个专家。
2. **分发**（dispatch）：每张卡把每个 token 的副本发到它选中的专家所在的卡。卡 i 发给卡 j 的是一批**不同的** token，每对卡之间都可能有数据，这正是 **All-to-All**。
3. **专家计算**：每张卡用自己的专家，处理收到的所有 token。
4. **合并**（combine）：再做一次 All-to-All，把结果发回 token 原来的卡，按门控权重加起来。

代码模拟了这个过程，验证两次 All-to-All 之后的结果和不分布式时完全一样。它还打出了**分发矩阵**：第 i 行第 j 列是卡 i 发给卡 j 的 token 副本数。

负载不均衡

路由是模型自己学的，并不保证均匀。例子里路由器偏爱专家 0 和 1，各专家收到的 token 数是 18、17、7、7、4、5、5、1。专家 0、1 都在卡 0 上，卡 0 一共要处理 35 个副本，而平均每张卡只有 16 个。

这在专家并行里是个大问题：第 2 次 All-to-All 要等所有卡都算完，**最忙的那张卡决定了这一层的速度**。卡 0 的工作量是平均的两倍多，其他卡算完就只能等。第 41 讲的负载均衡损失、DeepSeek-V3 的无辅助损失均衡（调整路由偏置），在训练时就是为了让专家的负载尽量均匀。

下面的演示可以调 top-k 和路由的偏斜程度，看每个 token 去了哪里、分发矩阵和每张卡的负载。



图 108-1　专家并行的分发。每个 token 旁的彩色数字是它选中的专家；分发矩阵是每对卡之间要传的 token 副本数（对角线留在本卡）。路由越偏斜，越多 token 挤到同一张卡上，所有卡都要等它。

要点

专家并行把不同的专家放到不同的卡上，每个 MoE 层用两次 All-to-All：分发把 token 副本发到专家所在的卡，合并把结果送回原来的卡加权求和。每对卡之间传的数据都不一样，量取决于路由；最忙的卡决定整层的速度，所以负载均衡至关重要。

通信要多少

每张卡每层要发出的数据量 ≈ token 数 × top-k × 隐藏维度 × 每个数的字节数（留在本卡的不用发）。估算 DeepSeek-V3 的 decode：每张卡 128 个 token，top-8，d = 7168，分发用 FP8（1 字节），合并用 BF16（2 字节）：

- 每层分发约 **7.3 MB**，合并约 **14.7 MB**；
- 如果几乎都要跨机、走 50 GB/s 的网卡，每层约 0.44 ms，58 个 MoE 层约 **26 ms**。

这对 decode 来说是很重的负担（一步往往只有几十毫秒的预算）。所以 MoE 的推理系统在通信上下了大功夫：

- **限制跨机的范围**。DeepSeek-V3 训练时规定每个 token 最多发往 4 个节点，减少跨机流量。
- **先走机内，再走机间**。跨机的数据先在本机内用 NVLink 汇集，再用网卡统一发出，让每个数据只跨一次机。
- **计算和通信重叠**。把一批 token 切成两半，一半在做 All-to-All 的时候，另一半在做注意力或专家计算。
- **专门的通信库**。DeepSeek 开源的 DeepEP 为 MoE 的分发、合并写了专门的 kernel，decode 用低延迟的版本，prefill 用高吞吐的版本。

DeepSeek-V3 是怎么部署的

DeepSeek-V3 的技术报告给出了它的推理部署：prefill 的最小单元是 4 台机器、32 张卡，专家部分用 EP32；decode 的最小单元是 40 台机器、**320 张卡**，用 EP320，几乎每张卡只放一个专家。

为什么 decode 要用这么多卡？decode 每步每张卡只有少量 token，专家的矩阵乘法很小，卡在读权重上（第 68 讲）。专家分得越散，每张卡要读的专家权重越少；同时，同一时刻在系统里运行的 token 越多，每个专家收到的 token 越多，算术强度越高。用大量的卡、每卡一个专家，正是在这两者之间找到平衡。

为了应对负载不均衡，系统还会**复制热门专家**：统计一段时间里各专家的负载，把最忙的专家多复制几份放到别的卡上，分摊它的 token（冗余专家）。

到了大模型里

推理引擎的支持。vLLM、SGLang 都支持专家并行（比如 `--enable-expert-parallel`），并集成了 DeepEP 等通信库，可以把注意力部分的数据并行和专家部分的专家并行组合起来（常叫作“DP 注意力 + EP”）。

专家并行适合大规模的高速互联。All-to-All 让每对卡都要通信，对网络的要求比 All-Reduce 更苛刻。GB200 NVL72 这样 72 张卡共享 NVLink 的超节点，让一整个 EP 组的 All-to-All 都走高速链路，对 MoE 模型的推理特别有利（第 65、109 讲）。

和张量并行比较。稠密模型用张量并行，每层两次 All-Reduce；MoE 模型的专家部分用专家并行，每层两次 All-to-All。专家并行不拆矩阵，每个专家的计算保持完整，GPU 更容易跑满；代价是负载不均衡和更复杂的通信。

用代码复现

下面的代码模拟 4 张卡、8 个专家、top-2 的专家并行：路由、第 1 次 All-to-All 分发（打出分发矩阵）、各卡的专家计算、第 2 次 All-to-All 合并，验证结果与不分布式的计算一致，并统计每个专家的负载；最后估算 DeepSeek-V3 decode 时每层的通信量和专家的规模。

```python
import numpy as np
rng = np.random.default_rng(1)

# 1. 4 张卡，8 个专家（卡 g 放专家 2g、2g+1），每张卡上有 8 个 token，每个 token 选 top-2 个专家
G, E, T, k, d = 4, 8, 8, 2, 6
x = rng.standard_normal((G, T, d))                          # 每张卡上的 token
logits = rng.standard_normal((G, T, E)) + np.array([1.0, 0.8, 0, 0, 0, 0, 0, -0.5])
# 路由偏爱前两个专家
top = np.argsort(-logits, -1)[..., :k]                      # 每个 token 选的专家
gate = np.take_along_axis(np.exp(logits), top, -1); gate /= gate.sum(-1, keepdims=True)
W = rng.standard_normal((E, d, d)) * 0.3                    # 每个专家是一个小线性层（示意）
home = lambda e: e // 2                                     # 专家 e 在哪张卡上

# 不分布式的参考结果
ref = np.einsum("gtk,gtkd→gtd", gate, np.einsum("gtd,gtkde→gtke", x, W[top]))

# 第 1 次 All-to-All（分发）：卡 g 把每个 token 的副本发到它选中专家所在的卡
send = np.zeros((G, G), int); inbox = [[] for _ in range(G)]
for g in range(G):
    for t in range(T):
        for j in range(k):
            e = top[g, t, j]; send[g, home(e)] += 1; inbox[home(e)].append((g, t, j, e))
print("分发矩阵（第 i 行第 j 列 = 卡 i 发给卡 j 的 token 副本数）：")
for g in range(G): print("  卡", g, send[g].tolist())
load = np.bincount(top.ravel(), minlength=E)
print("每个专家收到的 token 数：", load.tolist(), f"；最忙的卡 {send.sum(0).max()} 个，平均 {send.sum(0).mean():.0f} 个")

# 各卡上的专家算完，第 2 次 All-to-All（合并）：结果发回 token 原来的卡，按门控权重加起来
out = np.zeros_like(x)
for dst in range(G):
    for g, t, j, e in inbox[dst]:
        out[g, t] += gate[g, t, j] * (x[g, t] @ W[e])
print("两次 All-to-All 之后的结果 = 不分布式的结果：", np.allclose(out, ref))

# 2. DeepSeek-V3 decode 的通信量（估算）：每张卡 128 个 token，top-8，d = 7168；分发用 FP8（1 字节），合并用 BF16（2 字节）
B, K, D, L_moe = 128, 8, 7168, 58
disp, comb = B * K * D * 1, B * K * D * 2
t_layer = (disp + comb) / 50e9                              # 假设几乎都要跨机，走 50 GB/s 的网卡
print(f"每层每卡：分发 {disp / 1e6:.1f} MB，合并 {comb / 1e6:.1f} MB；按 50 GB/s 约
```

```
{t_layer * 1e3:.2f} ms；58 个 MoE 层共 {t_layer * L_moe * 1e3:.0f} ms")
expert = 3 * D * 2048                                    # 一个专家：gate、up、down
三个矩阵（中间维度 2048）
print(f"一个专家 {expert / 1e6:.0f}M 参数；一层 256 个专家 {256 * expert / 1e9:.1f}B；
分到 32 张卡，每卡每层 8 个专家")
```

输出：

```
分发矩阵（第 i 行第 j 列 = 卡 i 发给卡 j 的 token 副本数）：
  卡 0 [8, 2, 4, 2]
  卡 1 [7, 4, 4, 1]
  卡 2 [13, 2, 1, 0]
  卡 3 [7, 6, 0, 3]
每个专家收到的 token 数： [18, 17, 7, 7, 4, 5, 5, 1] ；最忙的卡 35 个，平均 16 个
两次 All-to-All 之后的结果 = 不分布式的结果： True
每层每卡：分发 7.3 MB，合并 14.7 MB；按 50 GB/s 约 0.44 ms；58 个 MoE 层共 26 ms
一个专家 44M 参数；一层 256 个专家 11.3B；分到 32 张卡，每卡每层 8 个专家
```

前五行是分发矩阵：卡 2 发给卡 0 的最多（13 个），因为路由偏爱卡 0 上的专家 0、1。接着是各专家的负载：最忙的卡 35 个，平均 16 个。然后是正确性验证。最后两行是 DeepSeek-V3 的规模：每层分发 7.3 MB、合并 14.7 MB，58 层约 26 ms；一个专家 44M 参数，一层 11.3B。

容易混淆的地方

1. **专家并行只切 MoE 层。**注意力、嵌入等部分还要用别的方式切（数据并行、张量并行），所以专家并行几乎总是和别的并行方式组合在一起。
2. **All-to-All 的数据量取决于路由。**它不像 All-Reduce 那样固定；同一个模型、同样的批大小，换一批输入，各卡之间的流量就不同。
3. **负载均衡不只是训练的事。**训练时的均衡策略让平均负载均匀，但推理时某一批输入仍可能集中在少数专家上；推理系统还要靠冗余专家等手段应对。

回顾

MoE 每个 token 只用少数专家，但参数一个也不少，DeepSeek-V3 一层 256 个专家就有 113 亿参数。专家并行把不同的专家放到不同的卡上，每个 MoE 层两次 All-to-All：分发把 token 副本发到专家所在的卡，合并把结果送回加权求和。路由不均匀时，最忙的卡决定整层的速度：4

卡、8 专家的例子里，最忙的卡 35 个副本，平均只有 16 个。DeepSeek-V3 decode 每卡 128 个 token 时，每层分发 7.3 MB、合并 14.7 MB，推理系统靠限制跨机、分层传输、计算通信重叠和专门的通信库（DeepEP）来应对；它的 decode 用 320 张卡、EP320 部署，并复制热门专家。

到这里，我们有了五种并行方式：数据、张量、流水、上下文、专家。真实的系统要把它们组合起来，放到具体的硬件上：哪种并行放在 NVLink 上，哪种放在网络上？下一讲：**并行策略与拓扑映射**。

第109讲 并行策略与拓扑映射

重点

99 为什么需要多卡　102 通信代价模型　103 数据并行与 ZeRO　105 张量并行　106 流水并行　107 序列与上下文并行　108 专家并行与 All-to-All

前面几讲给了我们五种并行方式：数据并行（DP）、张量并行（TP）、流水并行（PP）、上下文并行（CP）、专家并行（EP）。真实的系统几乎从不只用一种，而是把它们组合起来：比如 TP = 8、PP = 2、DP = 4，一共 64 张卡。

组合起来还不够，还要回答一个具体的问题：**这 64 张卡，哪些卡组成一个 TP 组，哪些组成一个 PP 组？**卡与卡之间的连接是分层的：同一台机器里的卡用 NVLink 相连，快；不同机器之间走网络，慢了一个数量级。同样的并行配置，分组的方式不同，性能可以差好几倍。这一讲把前面的通信分析汇总起来，讲清楚怎么把并行方式映射到拓扑上。

互联是分层的

以 H100 的集群为例（数字是第 102 讲用的量级）：

- **机内**：一台 DGX H100 有 8 张卡，通过 NVSwitch 两两全连，每张卡每个方向约 450 GB/s，延迟几微秒。
- **机间**：每张卡配一张 400 Gb/s 的 InfiniBand 网卡，约 50 GB/s，延迟更大，还要经过交换机。

带宽差了 9 倍。更大的系统还有更多层：同一个机柜、同一个网络交换层、跨数据中心……越往外越慢。

每种并行的通信特点

把前几讲的结论放在一起：

并行方式	每一步的通信	频率	能不能和计算重叠
张量并行 TP	All-Reduce，每层 2 次	最高	很难，下一层要等它
上下文并行 CP	沿环传 K、V，每层	高，量大	可以，靠重叠
专家并行 EP	All-to-All，每个 MoE 层 2 次	高，量大且不均匀	部分可以
流水并行 PP	段间点对点，每步几次	低，量小	容易
数据并行 DP	推理时没有；训练时每步同步一次梯度	最低	容易

由此得出映射的原则：**通信越频繁、越需要同步等待的并行，放在越快的互联上**。TP 放在最里层（NVLink），CP、EP 尽量也在 NVLink 域内；PP 和 DP 放在外层，跨机器。

算一个例子

两台 8 卡的 H100 机器，一共 16 张卡，部署 Llama-3.1-70B（BF16 141 GB），每个微批次 32 个请求。代码估算了几种方案（TPOT = 读权重 + TP 通信 + 段间传输）：

- **TP = 8（机内）× 2 份**：每份模型在一台机器里，TP 走 NVLink。TPOT **7.0 ms**，其中通信 1.78 ms；全集群约 9083 token/s。
- **TP = 16（跨两台机器）**：每张卡只放 8.8 GB 权重，读权重快了一倍，可每层的 All-Reduce 都要走网络，TP 通信 31.95 ms，TPOT **34.6 ms**，吞吐只有 925 token/s。
- **TP = 8（机内）× PP = 2（跨机）**：机内 TP、机间 PP。段间每步只传一次 524 KB，TPOT 7.1 ms，和第一种几乎一样，而且每张卡只占 8.8 GB 权重，KV Cache 的空间更大。
- **TP = 8 但每组跨两台机器 × 2 份**：卡数和配置都和第一种一样，只是分组的方式错了：每个 TP 组有 4 张卡在机器 0、4 张在机器 1。All-Reduce 被迫走网络，TP 通信 16.38 ms，TPOT **21.6 ms**，是第一种的三倍。
- **TP = 4（机内）× 4 份**：单个请求慢一些（TPOT 11.6 ms），但份数多，全集群吞吐约 11,071 token/s，比 TP = 8 还高。

最后一个例子说明了另一个权衡：**TP 越大，单个请求越快，但通信占比越高；TP 越小，总吞吐可能越高，但延迟变大，每份的 KV Cache 空间也更少**。选哪个，取决于服务对延迟的要求（第 82 讲的 SLO）。

下面的演示可以选 TP、PP 和放法，看 16 张卡怎么分组，以及 TPOT 和吞吐。颜色相同的卡属于同一份模型（和同一段）。



图 109-1　并行方式映射到两台 8 卡机器。TP 组放在机内时，每层的 All-Reduce 走 NVLink；故意让 TP 组跨机，同样的配置 TPOT 变成三倍。PP 跨机影响很小。

要点

互联是分层的：机内 NVLink 快，机间网络慢一个数量级。按通信的频率和同步程度从里到外排：TP（每层两次 All-Reduce）放在最里层 NVLink，CP、EP 尽量在 NVLink 域内，PP（段间少量点对点）和 DP 放在跨机网络上。同样的配置，分组方式错了，性能可以差好几倍。

训练里的“4D 并行”

大规模训练把四五种并行全用上。Meta 的 Llama 3 技术报告描述了 405B 模型的训练：从里到外依次是张量并行、上下文并行、流水并行、数据并行（FSDP），正是按通信频率从高到低排列的。TP 在机内，CP 也尽量在机内，PP 跨机，DP 在最外层，梯度同步可以和反向传播重叠。

训练框架（Megatron、TorchTitan 等）的配置里，往往有一个参数决定这几个维度的嵌套顺序；卡的编号顺序和物理拓扑对齐，内层的维度自然落在同一台机器里。

超节点改变了什么

上面的原则有一个前提：NVLink 域只有 8 张卡。超过 8，就只能跨机。

NVIDIA 的 GB200 NVL72 把 72 张卡放进一个 NVLink 域，任意两张卡之间都是 NVLink 的带宽（第 65 讲的 scale-up）。这对两类负载影响最大：

- **更大的 TP**。70B 甚至更大的稠密模型，TP 可以超过 8，延迟进一步降低。
- **专家并行**。MoE 的 All-to-All 是所有卡两两通信，最怕慢速链路。第 108 讲说 DeepSeek-V3 的 decode 用 320 张卡、靠 InfiniBand 和大量通信优化；在 72 卡的 NVLink 域里，一整组 EP 的 All-to-All 都走高速链路，简单得多，也快得多。

更大的高速互联域，让原本必须放在慢速网络上的并行维度，可以搬到快速链路上。这正是超节点的价值，也是第四章从互联角度讲推理的原因。下一篇会接着讲：在这样的集群上，推理系统还有哪些专门的设计。

到了大模型里

推理引擎的参数。vLLM、SGLang 用 `--tp`、`--pp`、`--dp`、`--enable-expert-parallel` 等参数组合并行方式；多机部署时，通常每台机器内部做 TP，机器之间做 PP 或 DP。

让编号和拓扑对齐。通信库按 rank 的编号建立通信组。启动多机任务时，要保证同一个 TP 组的 rank 落在同一台机器上，否则就会出现上面“TP 组跨机”的情况，而且不会报错，只是悄悄地变慢。

先测再定。这一讲的数字是模型估算。实际选方案时，要用第 82 讲的压测方法，在目标负载下比较几种配置的 goodput。

用代码复现

下面的代码按 α-β 模型和第 68 讲的读权重时间，估算两台 8 卡机器上几种并行方案的每卡权重、TPOT（拆成读权重和 TP 通信）和全集群吞吐。

```
# 2 台机器 × 8 张 H100，部署 Llama-3.1-70B（BF16 141 GB，80 层，d = 8192），每份模型每个微批次 32 个请求
W, L, D, BW, GPUS = 141e9, 80, 8192, 3.35e12, 16
NVL = (3e-6, 450e9); IB = (6e-6, 50e9)                    #  (α, β)：机内 NVLink、跨机 InfiniBand（第 102 讲的示意数字）
m = 32 * D * 2                                            # 每次 All-Reduce / 段间传输的数据：32 个 token 的隐藏状态

def allreduce(t, link):
    if t == 1: return 0.0
    a, b = link
    return a + (t - 1) * m / b if link is NVL else 2 * (t - 1) * (a + m / t / b)
# 机内一步到位，跨机用 Ring

def plan(tp, pp, tp_cross_node):
    replicas = GPUS // (tp * pp)
    per_gpu = W / (tp * pp)
    read = W / tp / BW                                    # 一个 token 依次走完 pp 段，每段读 W/(tp·pp)，合计 W/tp
    comm = 2 * L * allreduce(tp, IB if tp_cross_node else NVL)
    p2p = 0 if pp == 1 else (pp - 1) * (IB[0] + m / IB[1])   # 段间点对点（这里都按跨机算）
    step = read + comm + p2p                              # 一个请求的 TPOT
    thru = replicas * pp * 32 / step                      # 流水线里同时有 pp 个微批次
    return per_gpu, step, thru, comm

print("方案 | 每卡权重 | TPOT（读权重 + TP 通信 + 段间） | 全集群吞吐")
for name, tp, pp, cross in [("TP=8（机内）× 2 份", 8, 1, False), ("TP=16（跨两台机器）", 16, 1, True),
                            ("TP=8（机内）× PP=2（跨机）", 8, 2, False), ("TP=8 但每组跨两台机器 × 2 份", 8, 1, True),
                            ("TP=4（机内）× 4 份", 4, 1, False)]:
    w, step, thru, comm = plan(tp, pp, cross)
    fit = "" if w + 20e9 < 72e9 else "（放不下：还要留 KV Cache）"
    print(f"{name} | {w / 1e9:5.1f} GB{fit} | {step * 1e3:5.1f} ms（TP 通信 {comm * 1e3:5.2f} ms） | {thru:6,.0f} token/s")
```

输出：

```
方案 | 每卡权重 | TPOT（读权重 + TP 通信 + 段间） | 全集群吞吐
TP=8（机内）× 2 份 |  17.6 GB |   7.0 ms（TP 通信  1.78 ms） |  9,083 token/s
TP=16（跨两台机器） |   8.8 GB |  34.6 ms（TP 通信 31.95 ms） |    925 token/s
TP=8（机内）× PP=2（跨机） |   8.8 GB |   7.1 ms（TP 通信  1.78 ms） |  9,062 token/s
TP=8 但每组跨两台机器 × 2 份 |  17.6 GB |  21.6 ms（TP 通信 16.38 ms） |  2,958 token/s
TP=4（机内）× 4 份 |  35.2 GB |  11.6 ms（TP 通信  1.04 ms） | 11,071 token/s
```

五行对应上面的五种方案：TP 留在机内（第一、三、五行）通信都很少；TP 跨机（第二、四行）通信时间远超读权重的时间。

容易混淆的地方

1. **并行的“大小”和“放法”是两件事**。TP = 8 说的是切成几份，放在哪 8 张卡上是另一个决定，后者同样关键。
2. **更多的卡不一定更快**。TP = 16 比 TP = 8 每卡的权重少了一半，却因为跨机通信慢了五倍。增加并行度之前，先算通信。
3. **延迟和吞吐要分开看**。小的 TP 加更多份数，吞吐可能更高，但每个请求的 TPOT 更长。选方案要看服务的目标是哪一个。

回顾

真实的系统把多种并行方式组合起来，还要决定哪些卡组成一组。互联是分层的：机内 NVLink 约 450 GB/s，机间网络约 50 GB/s。按通信的频率和同步程度从里到外安排：TP 最里层，CP、EP 尽量在 NVLink 域内，PP、DP 在外层跨机。两台 8 卡机器跑 70B：TP = 8 机内 × 2 份，TPOT 7.0 ms；TP = 16 跨机，34.6 ms；TP = 8 但每组跨机，21.6 ms；TP = 8 × PP = 2（PP 跨机）几乎不受影响。Llama 3 的训练也按 TP、CP、PP、DP 从里到外排列。GB200 NVL72 这样的超节点把 NVLink 域扩大到 72 张卡，让 TP、EP 能用上更大的高速域。

第 13 篇讲完了把模型切到多张卡上的各种方法。下一篇把视角放到推理集群：多卡推理时延迟和吞吐到底怎么随并行度变化，prefill 和 decode 要不要分到不同的卡上，KV Cache 怎么在机器之间传递？先从最常用的那一种说起。下一讲：**推理中的张量并行**。

AI INFRA · HANDBOOK

第 十四 篇

分布式推理与集群

收录 7 讲 · 第 110—116 讲

第110讲 推理中的张量并行

70 TTFT、TPOT 与吞吐　102 通信代价模型　105 张量并行　109 并行策略与拓扑映射

第 13 篇讲了把模型切到多张卡上的各种方法。从这一讲开始的第 14 篇，我们站在推理服务的角度，看这些方法用起来到底怎样，以及一个推理集群还需要哪些专门的设计。

先从最常用的张量并行说起。第 105 讲说明了它怎么切；这一讲回答部署时最常被问的问题：**模型放得下以后，还要不要加卡做张量并行？加了以后，延迟能降多少，吞吐又会怎样？**答案取决于你更在乎哪一个。

一步要多久

第 70 讲有一个单卡 decode 的模型：一步 ≈ max(计算 ÷ 算力, (权重 + KV) ÷ 带宽)。张量并行 t 张卡时，每张卡只读 1/t 的权重、1/t 的 KV（KV 按头切开了），也只算 1/t；但每层要多做 2 次 All-Reduce（第 105 讲），用第 102 讲的 NVLink 数字估算：

$$\text{TPOT}(t) \approx \text{单卡耗时} \div t + 2 \times \text{层数} \times \text{All-Reduce}(\text{批大小} \times d \times 2\ \text{字节})$$

代码按这个模型算了 Llama-3-8B 和 Llama-3.1-70B（上下文 2048）：

Llama-3.1-70B，批 64：

- TP = 1、2：放不下（每卡要存的权重加 KV 超过 72 GB）。
- TP = 4：TPOT 15.3 ms，每张卡每秒 1043 个 token。
- TP = 8：TPOT 10.0 ms，每张卡每秒 803 个 token。

卡数翻倍，延迟只降到原来的三分之二左右，每张卡的吞吐却降了约 23%。

Llama-3-8B，批 64：一张卡就放得下，TPOT 9.9 ms、每卡 6450 tok/s；TP = 8 时 TPOT 降到 2.0 ms，每卡却只有 4094 tok/s，少了三分之一以上。

为什么每张卡变慢了

读权重和计算是按 t 分摊的，这部分完美地随卡数缩小；通信却不是。每层 2 次 All-Reduce 的**次数**不随 t 变化，每次的**数据量**（批大小 × d）也不随 t 减少，延迟项（α）更是一分不少。t 越大，计算越短，通信所占的比例越高：70B、批 64 时，TP = 8 的 10.0 ms 里约三成是通信。

而且批越小，这个问题越明显。批 1 时，70B 从 TP = 4 到 TP = 8，TPOT 从 11.1 ms 降到 5.8 ms，还算可观；8B 批 1 时，TPOT 从 4.9 ms 降到 0.8 ms，但每张卡的吞吐从 205 降到 154。

下面的演示可以调模型、批大小和上下文，看 TP = 1、2、4、8 时的 TPOT（柱子）和每张卡的吞吐（折线）。

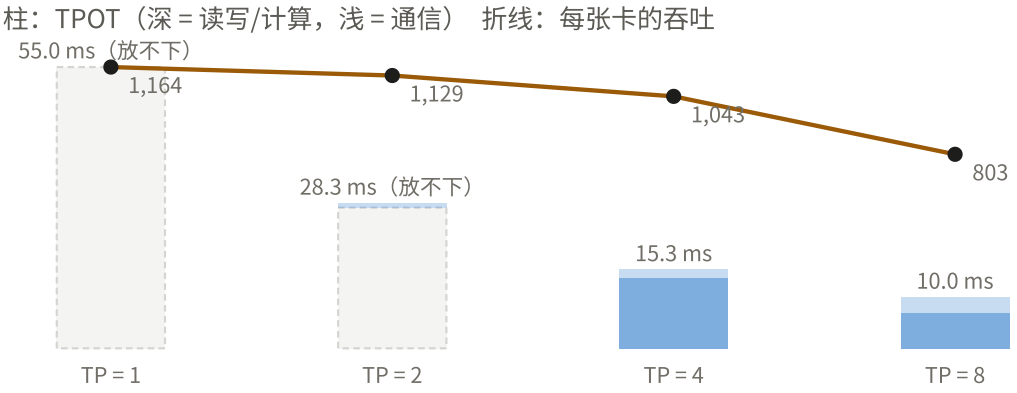
卡越多越快吗

⇆ 对照

Llama-3-8B　Llama-3.1-70B

批大小 b = **64**

上下文 **2048**



从 TP = 4 到 TP = 8：TPOT 15.3 → 10.0 ms（快了 1.5 倍，卡多了 2 倍）；每张卡的吞吐 1,043 → 803 token/s。
TP = 8 时通信占 31%，每张卡显存（权重 + KV）23.0 GB。
TP 用更多的卡换更低的延迟：权重和计算按比例分摊，但每层的通信不随卡数减少，所以卡越多，每张卡的效率越低。

图 110-1　张量并行度与延迟、每卡吞吐。柱子是 TPOT（浅色部分是通信），折线是每张卡每秒的 token 数。TP 越大，延迟越低，但通信占比越高，每张卡的效率越低。

要点

张量并行用更多的卡换更低的延迟。权重、KV 和计算按卡数分摊，但每层 2 次 All-Reduce 不随卡数减少，所以延迟的下降比卡数的增加慢，每张卡的吞吐随卡数下降。模型放得下以后，加 TP 是

为了满足延迟要求，不是为了提高吞吐。

那该怎么选

同样 8 张卡，跑 8B 模型可以有两种方式：一份 TP = 8，或者 8 份 TP = 1（数据并行，每张卡独立服务）。按上面的数字（批 64），后者的总吞吐是 8 × 6450 ≈ 51,600 tok/s，前者是 8 × 4094 ≈ 32,750 tok/s。可前者每个用户的 TPOT 是 2.0 ms，后者是 9.9 ms。

所以选择的顺序通常是：

1. **先保证放得下**：权重加上足够的 KV Cache 空间，决定了 TP 的下限。70B 的 BF16 至少要 TP = 4。
2. **再看延迟目标**：如果在下限的 TP 上，TPOT 已经满足 SLO（第 116 讲），就不要再加；不满足，才加大 TP。
3. **剩下的卡用来复制**：多出来的卡做多份副本（数据并行），提高总吞吐。

还有一个常被忽略的好处：**TP 越大，每份副本的 KV Cache 空间越大**。70B 用 TP = 8 时每张卡只放 17.6 GB 权重，剩下的空间都给 KV，一份副本能同时服务的请求更多、上下文更长。

prefill 的情况不同

prefill 是计算受限的，TP 把计算分摊到 t 张卡上，TTFT 的改善很明显。但 prefill 时每次 All-Reduce 的数据是整段提示的隐藏状态，很大。Llama-3.1-70B 一个 8192 token 的提示：

- TP = 2：计算 621 ms + 通信 48 ms = 669 ms；
- TP = 4：310 + 74 = 385 ms；
- TP = 8：155 + 90 = 245 ms。

通信从 48 ms 增加到 90 ms，因为消息大，用的是 Ring，每张卡要传的数据随 t 接近 2 倍的消息量，而计算在减半。实际实现会把 All-Reduce 和下一层的计算重叠起来，部分藏住它。

到了大模型里

张量并行要留在 NVLink 域内。第 109 讲说过，TP 跨机会让通信时间暴涨。8 卡服务器上，TP 最多 8；GB200 NVL72 这样的超节点上，TP 可以更大（第 115 讲）。

MLA 模型的特殊情况。DeepSeek 的 MLA（第 40 讲）把所有头的 KV 压缩成一个共享的潜向量，没法按头切开。用张量并行时，每张卡都要存一份完整的 KV Cache，TP 越大浪费越多。所以 SGLang 等推理引擎为它设计了“DP 注意力”：注意力部分每张卡各自处理不同的请求（数据并行），只有 MoE 部分用专家并行（下一讲）。

小模型往往不用 TP。8B 以下的模型一张卡就放得下，延迟也通常够用，多卡时更常见的是多副本加负载均衡（第 114 讲）。

用代码复现

下面的代码按上面的模型，计算 8B 和 70B 在批 1 和批 64 时，TP = 1、2、4、8 的 TPOT 和每卡吞吐，并标出放不下的情况；最后算 70B 一个 8192 token 提示的 TTFT。

```
# 多卡张量并行时一步的耗时：每卡读 1/t 的权重和 KV、算 1/t 的计算量，再加每层 2 次 All-Reduce（NVLink，第 102 讲）
F, BW, a, beta = 989e12, 3.35e12, 3e-6, 450e9
MODELS = {"Llama-3-8B": (8.03e9, 0.53e9, 32, 4096, 131072), "Llama-3.1-70B": (70.6e9, 1.05e9, 80, 8192, 327680)}
CTX = 2048                                          # 每个请求的上下文按 2048 个 token 算

def allreduce(m, tp):                               # 小消息用一步到位，大消息用 Ring，取快的那个
    return min(a + (tp - 1) * m / beta, 2 * (tp - 1) * (a + m / tp / beta))

def tpot(model, tp, b):
    N, Nemb, L, d, kv = MODELS[model]
    flops = (2 * (N - Nemb) + 4 * L * d * CTX) * b / tp
    bytes_ = (N * 2 + kv * CTX * b) / tp
    comm = 0 if tp == 1 else 2 * L * allreduce(b * d * 2, tp)
    return max(flops / F, bytes_ / BW) + comm, comm

for model in MODELS:
    print(f"{model}（上下文 {CTX}）：")
    for b in [1, 64]:
        row = []
        for tp in [1, 2, 4, 8]:
            t, c = tpot(model, tp, b)
            fits = MODELS[model][0] * 2 / tp + MODELS[model][4] * CTX * b / tp < 72e9
            row.append(f"TP={tp}: {t * 1e3:5.1f} ms{'' if fits else '（放不下）'}，每卡 {b / t / tp:6,.0f} tok/s")
        print(f"  批 {b:2d}：" + " | ".join(row))

# prefill：Llama-3.1-70B 一个 8192 token 的提示，计算受限，TP 把计算分摊，但每次 All-Reduce 的消息很大
N, Nemb, L, d, kv = MODELS["Llama-3.1-70B"]
for tp in [2, 4, 8]:
    comp = (2 * (N - Nemb) * 8192 + 4 * L * d * 4096 * 8192) / F / tp
    comm = 2 * L * allreduce(8192 * d * 2, tp)
    print(f"70B prefill 8192 token，TP={tp}：计算 {comp * 1e3:4.0f} ms + 通信 {comm * 1e3:3.0f} ms = TTFT {(comp + comm) * 1e3:4.0f} ms")
```

输出：

```
Llama-3-8B（上下文 2048）：
  批  1：TP=1：   4.9 ms，每卡    205 tok/s | TP=2：   2.6 ms，每卡    190 tok/s |
TP=4：   1.4 ms，每卡    177 tok/s | TP=8：   0.8 ms，每卡    154 tok/s
  批 64：TP=1：   9.9 ms，每卡  6,450 tok/s | TP=2：   5.2 ms，每卡  6,121 tok/s |
TP=4：   2.9 ms，每卡  5,524 tok/s | TP=8：   2.0 ms，每卡  4,094 tok/s
Llama-3.1-70B（上下文 2048）：
  批  1：TP=1：  42.3 ms（放不下），每卡     24 tok/s | TP=2：  21.7 ms，每卡     23
tok/s | TP=4：  11.1 ms，每卡     23 tok/s | TP=8：   5.8 ms，每卡     21 tok/s
  批 64：TP=1：  55.0 ms（放不下），每卡  1,164 tok/s | TP=2：  28.3 ms（放不下），每卡
1,129 tok/s | TP=4：  15.3 ms，每卡  1,043 tok/s | TP=8：  10.0 ms，每卡    803 tok/s
70B prefill 8192 token，TP=2：计算  621 ms + 通信  48 ms = TTFT  669 ms
70B prefill 8192 token，TP=4：计算  310 ms + 通信  74 ms = TTFT  385 ms
70B prefill 8192 token，TP=8：计算  155 ms + 通信  90 ms = TTFT  245 ms
```

前六行是 decode：TP 越大，TPOT 越短，每卡吞吐越低。最后三行是 prefill：TTFT 从 669 ms 降到 245 ms，但通信从 48 ms 涨到 90 ms。

容易混淆的地方

1. **“扩展效率”是相对卡数说的**。TP = 8 比 TP = 4 快了 1.5 倍，卡数却翻了倍，扩展效率约 77%。说“TP 让推理变快”时，要说清楚是延迟变快，而每张卡的效率在下降。
2. **批大小会改变结论**。批很大时，KV 的读取和计算占主导，通信相对变小，TP 的效率会好一些；批很小时，延迟项占主导，效率更差。
3. **放不下时，TP 不是唯一的选择**。也可以量化（第 90 讲）让模型变小，或者用流水并行跨机（第 106 讲）。

回顾

张量并行让每张卡分摊权重、KV 和计算，延迟随卡数下降；但每层 2 次 All-Reduce 不随卡数减少，延迟下降得比卡数增加慢，每张卡的吞吐随之降低。Llama-3.1-70B、批 64：TP = 4 时 TPOT 15.3 ms、每卡 1043 tok/s，TP = 8 时 10.0 ms、803 tok/s。选择的顺序是：先让模型放得下，再加 TP 满足延迟目标，剩下的卡做多副本提高吞吐。prefill 时 TP 对 TTFT 帮助很大（70B 8192 token 从 669 ms 到 245 ms），但每次 All-Reduce 的消息也大。MLA 模型的 KV 不能按头切，通常用 DP 注意力。

稠密模型的多卡推理，瓶颈常在 NVLink 上的 All-Reduce。MoE 模型不一样：它的专家要分到几十上百张卡上，跨机的 All-to-All 成了主角。为什么说 MoE 推理的瓶颈常在网络？下一讲：**MoE 推理与大规模专家并行**。

第111讲 MoE 推理与大规模专家并行

重点

41 MoE 混合专家　62 Roofline 模型　102 通信代价模型　108 专家并行与 All-to-All

第 108 讲讲了专家并行怎么工作：专家分到各张卡上，每个 MoE 层用两次 All-to-All 把 token 送过去、再把结果收回来。这一讲算一笔账：在 DeepSeek-V3 这样的大规模 MoE 推理里，时间到底花在哪里？

答案可能有些出乎意料：**专家的计算很快，慢的是网络**。这一讲用一个粗略的模型说明为什么，以及推理系统为此做了哪些努力。

专家算得很快

DeepSeek-V3 的每个专家约 4400 万参数（中间维度 2048，d = 7168）。用 EP = 64 部署时，每张卡每层只放 256 ÷ 64 = 4 个专家。decode 时每张卡有 128 个 token，每个 token 选 8 个专家，负载均衡时每张卡每层收到约 128 × 8 = 1024 个 token 副本。

这 4 个专家的 FP8 权重约 176 MB，从显存读一遍约 0.053 ms；1024 个 token 的计算量在 FP8 下更快。所以每层的专家计算约 **0.053 ms**，非常快。

网络很忙

再看通信。每个 token 要把隐藏状态（7168 维）复制 8 份发给 8 个专家（FP8，1 字节），算完再收回 8 份结果（BF16，2 字节）：每个 token 每层约 8 × 7168 × 3 ≈ 172 KB。每张卡 128 个 token，一层就是约 22 MB，绝大部分要发到别的卡上。

如果卡之间走 50 GB/s 的 InfiniBand，每层的 All-to-All 约 **0.45 ms**，是专家计算的八倍多。58 个 MoE 层累积下来，一步约 31 ms，其中通信占了 26 ms 左右。

代码扫了几种网络带宽（每卡 128 个 token，EP = 64；假设非专家部分每步 2 ms）：

- **25 GB/s**：一步约 53 ms（通信和计算重叠后），每卡 2405 tok/s。
- **50 GB/s**：约 28 ms，每卡 4560 tok/s。
- **200 GB/s**：约 9.2 ms，每卡 13,892 tok/s。
- **450 GB/s**（NVLink 量级）：约 5.7 ms，每卡 22,371 tok/s。

网络带宽从 50 提到 450 GB/s，吞吐提高了将近 5 倍。而换更快的 GPU，几乎没有帮助，因为 GPU 本来就大部分时间在等数据。

网络的“屋顶”

为什么会这样？可以用第 62 讲 Roofline 的思路看。专家计算对每个 token 是 8 个专家 × 2 × 4400 万 ≈ 7 亿次运算，网络流量是 172 KB，**每传 1 字节网络数据，只有约 4096 次运算**。而 H100 的 FP8 算力除以 50 GB/s 的网络，是约 39,580 次/字节。前者比后者小了将近 10 倍：这类负载的“网络算术强度”太低，必然卡在网络上。

更直接的看法是：**每个 token 每步要在网络上走约 10.0 MB**（58 层 × 172 KB），这和批大小、EP 规模都无关。所以每张卡的吞吐有一个硬上限：

$$\text{每卡吞吐} \leqslant \text{网络带宽} \div \text{每个 token 的流量} \approx 50\ \text{GB/s} \div 10.0\ \text{MB} \approx 5011\ \text{tok/s}$$

换成 450 GB/s，上限变成约 45,100 tok/s。这和稠密模型很不一样：稠密模型加大批大小可以摊薄读权重的成本，MoE 的通信却是按 token 计的，摊不薄。

下面的演示可以调网络带宽、每卡 token 数、EP 规模，以及是否重叠。

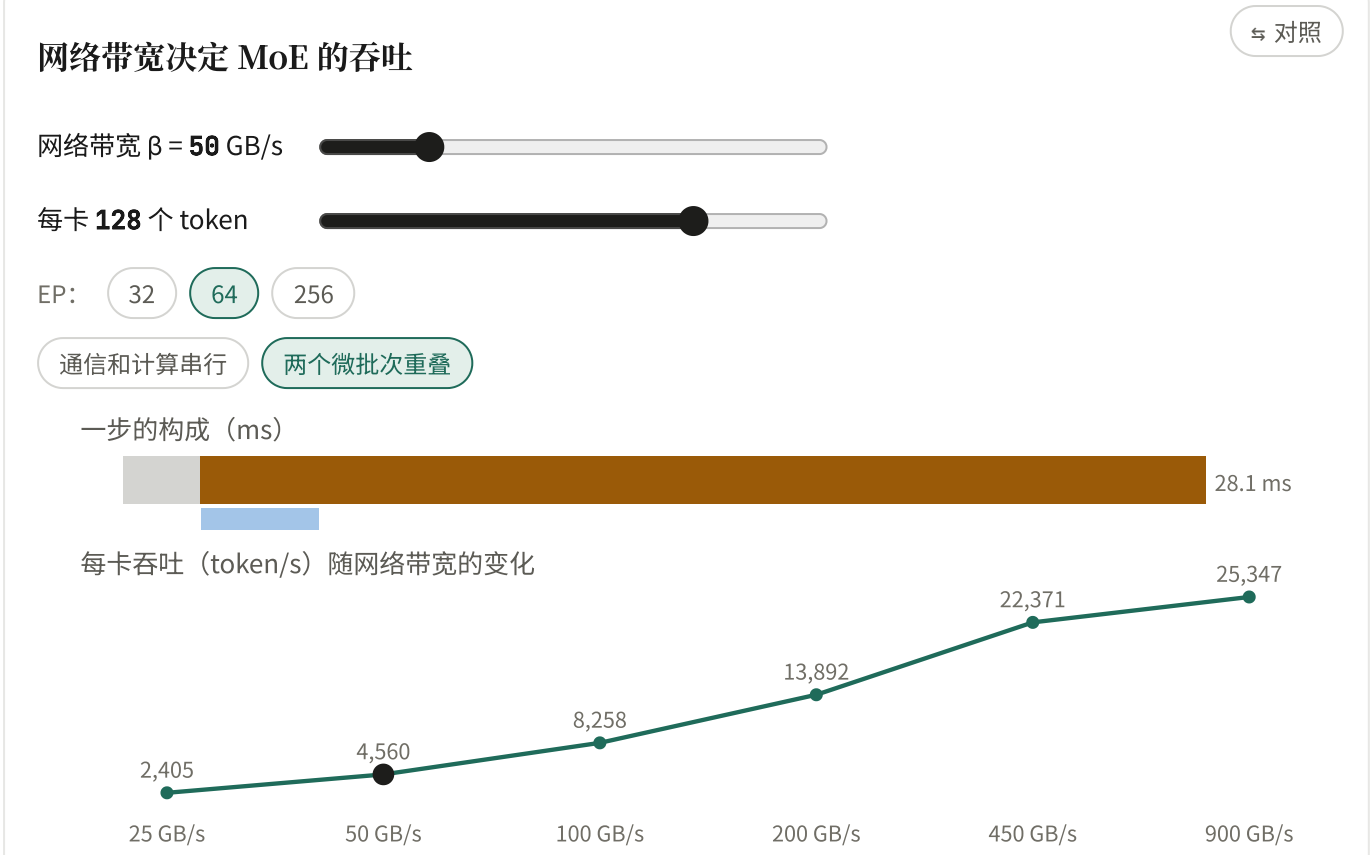


图 111-1　MoE decode 一步的构成与每卡吞吐（粗略模型）。专家计算（蓝）很短，All-to-All（琥珀）占大头；网络带宽从 50 GB/s 提高到 450 GB/s，吞吐提高近 5 倍。

要点

大规模专家并行时，每张卡上只有几个专家，专家计算很快；每个 token 每层却要把 top-k 份隐藏状态送出、收回，DeepSeek-V3 规模下每个 token 每步约 10 MB 的网络流量，和批大小无关。每卡吞吐的上限约等于网络带宽除以这个流量，所以 MoE 推理的瓶颈常在网络。

为什么不少用几张卡

既然通信这么贵，为什么不把专家集中在少数几张卡上？首先是**放不下**：EP = 8 时每张卡每层要放 32 个专家，58 层的专家权重加上非专家部分约 99 GB，超过了 80 GB。其次，专家集中了，每张卡每层要读的专家权重就多，专家计算从 0.053 ms 变成 0.42 ms，同样不快。

而且减少 EP 规模，并不能减少每个 token 的网络流量：只要专家不在本卡上，就得发出去。真正能减少流量的，是让更多的通信**留在快速链路上**，或者**减少每次传的字节**。

推理系统的对策

更快的互联。把 EP 组放进 NVLink 域里，All-to-All 就从 50 GB/s 变成了几百 GB/s。8 卡的服务器只能容纳很小的 EP；GB200 NVL72 这样 72 张卡共享 NVLink 的超节点，可以把整个 EP 组都放在高速域内（第 115 讲）。

减少字节。分发时用 FP8 传（我们已经这样算了）；有的系统合并时也用更低的精度。DeepSeek-V3 训练时还限制每个 token 最多发到 4 个节点，减少跨机的流量。

计算和通信重叠。把每张卡的 token 分成两个微批次，一个在做 All-to-All 时，另一个在算注意力或专家。代码里的“重叠”就是取两者中较大的那个。DeepSeek 的推理系统正是这样做的。不过，当通信远大于计算时，重叠也只能藏住一小部分。

专门的通信库。DeepEP 等库为 MoE 的分发、合并写了专门的 kernel：decode 用低延迟的版本（直接通过 RDMA 读写对方显存，减少 α），prefill 用高吞吐的版本。

冗余专家与负载均衡。第 108 讲说过，最忙的卡决定整层的速度。复制热门专家、动态调整专家的放置，让每张卡收到的 token 尽量相等。

一层 MoE 里的两次等待

前面把通信当成一个整体。把一层 MoE 拆开，可以看到决定耗时的不是平均，而是最慢的那一份。下图是一种常见的放法：服务器 A 的 4 张卡做注意力（各自保存自己请求的 KV），服务器 B 的 4 张卡各放四分之一的路由专家。

一层 MoE：两次 All-to-All 夹着专家计算

对照

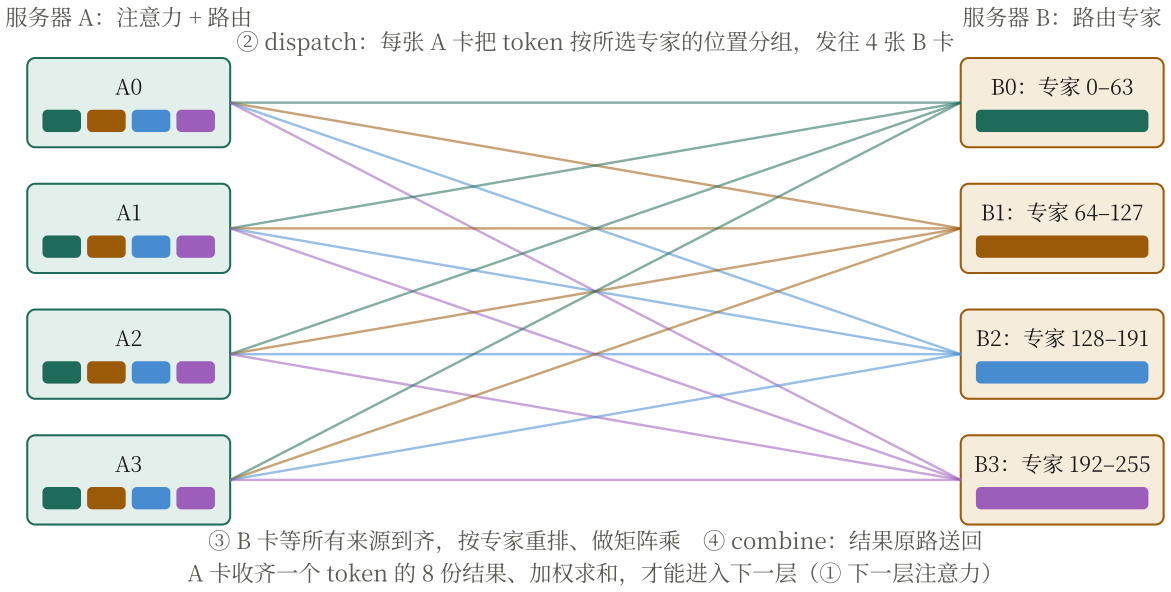


图 111-2　一层 MoE 的执行顺序。A 卡算完注意力和路由，按每个 token 所选专家的位置把隐藏状态分组发出（dispatch，4 × 4 的 All-to-All，颜色表示去哪张 B 卡）；B 卡收齐后做专家计算；结果原路送回（combine），A 卡收齐一个 token 的 8 份结果才能进入下一层。

这条路径上有两处必须等的地方：B 卡要等所有来源的输入到齐才开始算；A 卡要等一个 token 的 8 份结果都回来才能求和。所以一张 B 卡收得多、算得慢，会同时推迟它自己的计算、它的返回，以及所有在等它结果的 A 卡。

算一个具体的例子：1024 个 token、每个选 8 个专家，每行 8 KiB（4096 维 BF16）。dispatch 一共 64 MiB，combine 也是 64 MiB。每张卡经自己的 50 GB/s 网卡收发。

- **均衡**：每个专家组收 2048 行（16 MiB）。dispatch 0.336 ms，专家计算 0.156 ms，combine 0.336 ms，三段串行 **0.827 ms**。
- **热点**：分布变成 [5120, 1024, 1024, 1024]，总字节数一样，组 0 却要收 40 MiB。三段都变成原来的 2.5 倍，共 **2.068 ms**。
- **同样的热点，两池同在一台 HGX（NVLink 450 GB/s）**：两次交换都压到 0.093 ms，共 0.577 ms。这时最长的一段变成热点卡 0.391 ms 的计算，再加带宽已经没用，要减少热点卡的工作。

跨服务器时，两次交换占了总时间的八成。H100 算一行专家只要约 76 ns，网卡传一行 8 KiB 却要约 164 ns。

把一批拆成两个微批次，第一个在算的时候第二个在 dispatch，三段可以流水：均衡时从 0.827 ms 降到 **0.581 ms**。流水的节奏由最长的一段（这里是通信）决定。但重叠也有代价：网卡的 DMA 和同时运行的矩阵乘法会争用显存带宽，如果因此 dispatch 和 combine 各慢约一半，流水就回到 0.827 ms，通信确实重叠了，整层却没有变快。

最忙的那张卡

上面的热点是人为设的。实际中热点从哪里来？EP 越大，问题越严重吗？

还是 1024 个 token 各选 8 个专家，共 8192 次分派，256 个专家平均每个 32 行。设专家 0 比较热，收到平均值 4 倍的 128 行，其余均分给另外 255 个专家。专家按编号均分到各张卡：

- **EP8**：每卡 32 个专家，热点卡约 1108 行，只比平均的 1024 行多 8%。热点被同卡的其他 31 个专家冲淡了。
- **EP32**：每卡 8 个专家，热点卡约 349 行，比平均的 256 行多 36%。
- **EP256**：每卡只剩这一个专家，128 行，是平均 32 行的 **4 倍**。专家之间的不均，原样变成了卡之间的不均。

从 EP8 到 EP256，每卡的平均行数缩小到 1/32，最忙卡的行数却只从 1108 降到 128。整层按最忙的卡计时，EP256 中一张平均的卡有四分之三的时间在等。

没有热点也会这样。即使每个 token 完全随机地选专家，各专家的行数也会在 32 附近随机波动；一张卡上专家越多，波动越能互相抵消，卡越多，越可能出现一张特别忙的卡。模拟 300 批：EP8 时最忙的卡平均只比每卡平均多 4%，EP64 多 21%，EP256 多 **53%**。

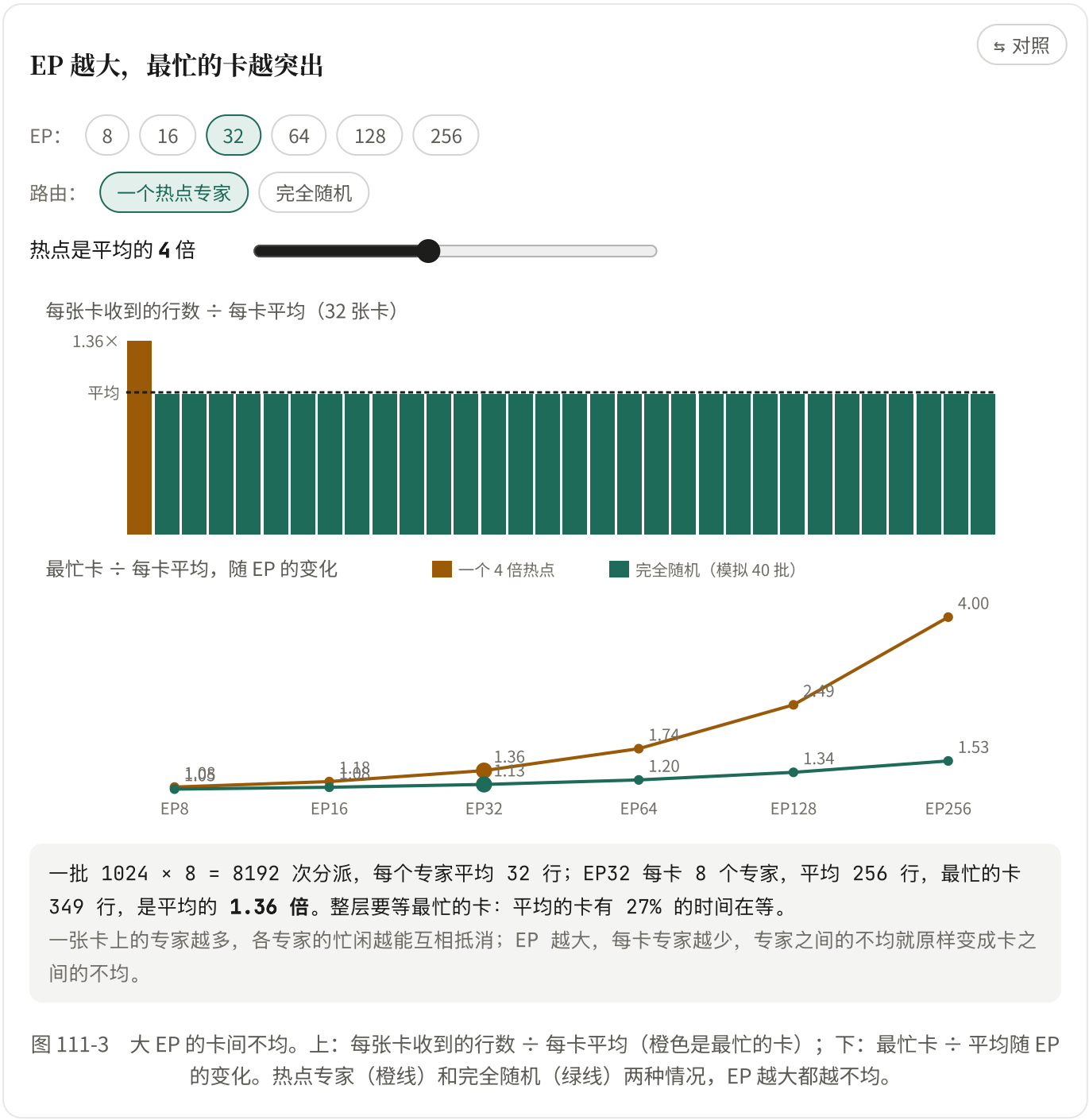


图 111-3　大 EP 的卡间不均。上：每张卡收到的行数 ÷ 每卡平均（橙色是最忙的卡）；下：最忙卡 ÷ 平均随 EP 的变化。热点专家（橙线）和完全随机（绿线）两种情况，EP 越大都越不均。

这就是 DeepSeek 的 decode 部署（EP144，每卡只放 2 个路由专家）要额外放 32 个**冗余专家**、并按负载动态调整的原因。

冗余专家多久回本

冗余专家就是给热门专家多放几份副本，让空闲的卡也能分担它的任务。只要副本的权重相同、合并时用对路由权重，就不改变“每个 token 选分数最高的 8 个专家”这条规则，模型的输出不变。vLLM、SGLang 的 EPLB（专家负载均衡）就是按路由统计调整副本的位置。

副本不是免费的：它占显存，复制也要时间。算一个例子：一台 HGX H100 上，一批 128 个 token 都选中卡 0 上的 8 个热点专家，共 1024 次分派。每个专家（Qwen3-235B-A22B 的规模）36 MiB。把其中 7 个各复制一份到卡 1 至卡 7，卡 0 只剩 576 次分派；它是读显存受限的，每批从约 0.550 ms 降到 0.309 ms，省约 0.24 ms。

- **同一台服务器内**复制，走 NVLink，7 份共约 **0.62 ms**，第 **3** 批就回本；热点持续 16 批，净省约 3.2 ms。
- **跨服务器**复制，走 50 GB/s 网卡，约 **5.32 ms**，要到第 **23** 批才回本；热点只持续 16 批的话，反而净亏约 1.5 ms。

所以副本策略要预测的是“热点还会持续多久”。在机器内，复制几乎不花时间，限制副本数的是显存（每卡每层多一个 36 MiB 的专家，94 层就是 3.3 GiB，挤掉的是 KV）；跨机器复制贵，频繁地追着热点搬副本反而亏。研究系统 CRAFT 的做法是按层分配副本预算，把显存花在最拥挤的层上。

专家放在 CPU 还是 GPU

还有一种更“省卡”的思路：显存放不下全部专家，就把大部分专家留在 CPU 内存里。这时有两条路：把被选中的专家权重搬到 GPU 上算；或者把输入传给 CPU，让 CPU 用本地的权重算，再把结果传回来。前者花链路带宽换 GPU 算力，后者花 CPU 算力换更少的流量。KTransformers 走的是后一条路。

哪条路快，取决于同一个专家这一批收到多少个 token。用 KTransformers 论文的实验机算：一颗 Xeon 8452Y（同插槽内存带宽实测 220 GB/s，AVX-512 kernel 实测 1.8 TFLOP/s，AMX 矩阵指令 kernel 21.3 TFLOP/s），一张 A100 40GB PCIe（按峰值 50% 计），PCIe 交接 25 GB/s。8 个专家，每个 36 MiB：

- **每个专家只收到 1 个 token**：CPU 读一遍本地的 288 MiB 权重约 1.4 ms，加上传输共 **1.46 ms**；搬到 GPU 光传权重就要约 12 ms，共 **12.5 ms**。专家复用少时，避免搬大块权重最重要。
- **每个专家收到 128 个 token**：计算量涨了 128 倍。AVX-512 的 CPU 要 22.2 ms，不如搬到 GPU 的 12.5 ms；换成 AMX，CPU 只要 2.57 ms，还是 CPU 快。

交点：用 AVX-512 时，每个专家超过约 **71** 个 token 就该搬到 GPU；用 AMX 时要超过约 **688** 个。同一台机器、同一批请求，CPU 用哪套指令就决定了专家该放在哪一侧。这也说明大内存 + 少量 GPU 的机器适合批小的场景：并发一高，CPU 的算力先用完。

prefill 好一些

prefill 一次处理很多 token，注意力和专家的计算量都大得多，通信相对没那么突出，也更容易被重叠藏住。所以 DeepSeek-V3 的 prefill 用 32 张卡、decode 用 320 张卡（第 108 讲），两个阶段的部署完全不同。这也是下一讲 PD 分离的一个重要动机。

用代码复现

下面的代码按上面的粗略模型，估算 DeepSeek-V3 规模的 decode：先扫网络带宽，看每层的专家计算和通信时间、每步的耗时和每卡吞吐；再看 EP 规模和每卡 token 数的影响（标出放不下的情况）；最后算每个 token 的网络流量、吞吐上限和“网络算术强度”。

```
# DeepSeek-V3 式 MoE 的 decode，粗略模型：注意力部分各卡数据并行处理自己的 B 个 token，专家部分用 N 卡专家并行
D, TOPK, EXPERTS, LAYERS = 7168, 8, 256, 58
EXPERT_BYTES = 3 * D * 2048                                  # 一个专家约 44M 参数，FP8 每个 1 字节
F8, BW = 1979e12, 3.35e12                                    # H100 的 FP8 算力、显存带宽
ATTN_MS = 2.0                                                # 非专家部分（注意力、共享专家等）每步每卡的耗时，示意值

def step_ms(B, N, beta, alpha=8e-6, overlap=False):
    per_gpu = max(1, EXPERTS // N)                           # 每卡每层放几个专家
    toks = B * TOPK                                          # 负载均衡时，每卡每层收到 B·8 个 token 副本
    expert = max(per_gpu * EXPERT_BYTES / BW, 2 * EXPERT_BYTES * toks / F8)
# 读专家权重 vs 算
    comm = 2 * alpha + B * TOPK * D * 3 * (N - 1) / N / beta
# 分发 FP8 + 合并 BF16，留在本卡的不传
    layer = max(expert, comm) if overlap else expert + comm
    return ATTN_MS + LAYERS * layer * 1e3, expert * 1e3, comm * 1e3

B, N = 128, 64
print(f"每卡 {B} 个 token、EP = {N}（每卡每层 {EXPERTS // N} 个专家）：")
for beta in [25e9, 50e9, 100e9, 200e9, 450e9, 900e9]:
    t, e, c = step_ms(B, N, beta); to, _, _ = step_ms(B, N, beta, overlap=True)
    print(f"  网络带宽 {beta / 1e9:4.0f} GB/s：每层专家 {e:.3f} ms、通信 {c:.3f} ms；一步 {t:5.1f} ms（重叠后 {to:5.1f} ms），每卡 {B / to * 1e3:6,.0f} tok/s")

print("EP 规模的影响（50 GB/s，重叠）：")
for N in [8, 32, 64, 256]:
    for B in [32, 128]:
        t, e, c = step_ms(B, N, 50e9, overlap=True)
        mem = max(1, EXPERTS // N) * EXPERT_BYTES * LAYERS + 17e9          # 专家权重 + 约 17 GB 的非专家权重（FP8）
        note = "" if mem < 70e9 else f"（每卡 {mem / 1e9:.0f} GB，放不下）"
        print(f"  EP = {N:3d}、每卡 {B:3d} 个 token：每层专家 {e:.3f} ms、通信 {c:.3f} ms，每卡 {B / t * 1e3:6,.0f} tok/s{note}")

# 网络的“屋顶”：每个 token 每层要送出 8 份、收回 8 份隐藏状态，不管 B、N 多大都免不了
per_tok = LAYERS * TOPK * D * 3                                  # 字节
print(f"每个 token 一步的网络流量 {per_tok / 1e6:.1f} MB → 每卡吞吐上限 = β ÷ {per_tok / 1e6:.1f} MB：50 GB/s 时 {50e9 / per_tok:,.0f} tok/s，450 GB/s 时 {450e9 / per_tok:,.0f} tok/s")
print(f"专家计算每传 1 字节只有 {2 * EXPERT_BYTES * TOPK / (TOPK * D * 3):,.0f} 次运算；H100 的 FP8 算力 ÷ 50 GB/s 网络 = {F8 / 50e9:,.0f} 次/字节")
```

输出：

```
每卡 128 个 token、EP = 64（每卡每层 4 个专家）：
  网络带宽   25 GB/s：每层专家 0.053 ms、通信 0.883 ms；一步  56.3 ms（重叠后  53.2
ms），每卡  2,405 tok/s
  网络带宽   50 GB/s：每层专家 0.053 ms、通信 0.450 ms；一步  31.1 ms（重叠后  28.1
ms），每卡  4,560 tok/s
  网络带宽  100 GB/s：每层专家 0.053 ms、通信 0.233 ms；一步  18.6 ms（重叠后  15.5
ms），每卡  8,258 tok/s
  网络带宽  200 GB/s：每层专家 0.053 ms、通信 0.124 ms；一步  12.3 ms（重叠后   9.2
ms），每卡 13,892 tok/s
  网络带宽  450 GB/s：每层专家 0.053 ms、通信 0.064 ms；一步   8.8 ms（重叠后   5.7
ms），每卡 22,371 tok/s
  网络带宽  900 GB/s：每层专家 0.053 ms、通信 0.040 ms；一步   7.4 ms（重叠后   5.0
ms），每卡 25,347 tok/s
EP 规模的影响（50 GB/s，重叠）：
  EP =   8、每卡  32 个 token：每层专家 0.421 ms、通信 0.112 ms，每卡  1,212 tok/s（每卡
99 GB，放不下）
  EP =   8、每卡 128 个 token：每层专家 0.421 ms、通信 0.401 ms，每卡  4,849 tok/s（每卡
99 GB，放不下）
  EP =  32、每卡  32 个 token：每层专家 0.105 ms、通信 0.123 ms，每卡  3,511 tok/s
  EP =  32、每卡 128 个 token：每层专家 0.105 ms、通信 0.443 ms，每卡  4,625 tok/s
  EP =  64、每卡  32 个 token：每层专家 0.053 ms、通信 0.124 ms，每卡  3,473 tok/s
  EP =  64、每卡 128 个 token：每层专家 0.053 ms、通信 0.450 ms，每卡  4,560 tok/s
  EP = 256、每卡  32 个 token：每层专家 0.013 ms、通信 0.126 ms，每卡  3,445 tok/s
  EP = 256、每卡 128 个 token：每层专家 0.046 ms、通信 0.455 ms，每卡  4,512 tok/s
每个 token 一步的网络流量 10.0 MB → 每卡吞吐上限 = β ÷ 10.0 MB：50 GB/s 时 5,011 tok/s，
450 GB/s 时 45,100 tok/s
专家计算每传 1 字节只有 4,096 次运算；H100 的 FP8 算力 ÷ 50 GB/s 网络 = 39,580 次/字节
```

前七行：带宽从 25 GB/s 到 900 GB/s，每层通信从 0.883 ms 降到 0.040 ms，专家计算始终是 0.053 ms。中间八行：EP = 8 放不下；EP 从 32 到 256，每卡吞吐几乎不变（都约 4500 tok/s），因为网络流量按 token 计。最后两行是网络的屋顶：50 GB/s 时每卡最多约 5011 tok/s；每传 1 字节只有 4096 次运算，远低于 39,580。

这个模型非常粗略：它假设负载完全均衡、通信没有拥塞、所有专家流量都走同一种链路，非专家部分用一个固定的 2 ms 代替。但“专家快、网络慢”的结论是可靠的。

第二段代码复现这一讲后半部分的数字：热点专家和随机路由下的卡间不均，冗余专家的回本批数，专家放在 CPU 还是 GPU 的交点，以及一层 MoE 三段的时间。

```python
import numpy as np
# 1. 同一个热点专家，EP 越大，卡间越不均：256 个专家、1024 个 token 各选 8 个，专家 0 收到平均值 4 倍的 128 行
n, k, E = 1024, 8, 256
rows = np.full(E, (n * k - 128) / (E - 1)); rows[0] = 128
for ep in [8, 32, 256]:
    per_card = rows.reshape(ep, -1).sum(1)              # 专家按编号均分到 ep 张卡
    print(f"EP{ep:<3}：热点卡 {per_card.max():6.0f} 行，平均 {per_card.mean():5.0f} 行，最忙 ÷ 平均 = {per_card.max()/per_card.mean():.2f}")

# 2. 没有热点、完全随机地选专家，最忙的卡也会比平均多：模拟 300 批
rng = np.random.default_rng(0)
for ep in [8, 32, 64, 256]:
    r = []
    for _ in range(300):
        pick = np.argpartition(rng.random((n, E)), k, axis=1)[:, :k]   # 每个 token 随机选 8 个不同专家
        cnt = np.bincount(pick.ravel(), minlength=E).reshape(ep, -1).sum(1)
        r.append(cnt.max() / cnt.mean())
    print(f"随机路由 EP{ep:<3}：最忙卡平均比每卡平均多 {np.mean(r)*100-100:.0f}%")

# 3. 冗余专家多少批回本：8 个热点专家挤在卡 0，把其中 7 个各复制一份到卡 1-7
expert = 3 * 4096 * 1536 * 2                                    # Qwen3-235B-A22B 一个专家，BF16 36 MiB
t0 = 921e6 / 1675e9                                             # 卡 0 每批读写约 921 MB（显存受限），有效带宽 1675 GB/s
save = t0 - t0 * 576 / 1024                                      # 复制后卡 0 只剩 1024 次分派中的 576 次
for name, bw in [("同一台服务器 (NVLink 450 GB/s) ", 450e9), ("跨服务器 (网卡 50 GB/s) ", 50e9)]:
    copy = 7 * (expert / bw + 5e-6)
    print(f"{name}：复制 {copy*1e3:.2f} ms，第 {int(copy // save) + 1} 批回本；热点持续 16 批净省 {(16*save-copy)*1e3:+.1f} ms")

# 4. 专家放在 CPU 算，还是把权重搬到 GPU 算：8 个专家，每个收到 m 个 token
P, h = 3 * 4096 * 1536, 4096
def cpu(m, flops): return 8 * (2 * 5e-6 + 2 * m * h * 2 / 25e9 + max(2 * m * P / flops, expert / 220e9))
def gpu(m): return 8 * (5e-6 + expert / 25e9 + max(2 * m * P / 156e12, expert / 778e9))
for m in [1, 128]:
    print(f"每专家 {m:>3} 个 token：CPU(AVX-512) {cpu(m, 1.8e12)*1e3:5.2f} ms，CPU(AMX) {cpu(m, 21.3e12)*1e3:5.2f} ms，搬到 GPU {gpu(m)*1e3:5.2f} ms")
for name, f in [("AVX-512", 1.8e12), ("AMX", 21.3e12)]:
    m = next(m for m in range(1, 5000) if gpu(m) < cpu(m, f))
    print(f"{name}：每专家超过 {m - 1} 个 token，搬到 GPU 更快")
```

```
# 5. 一层 MoE 的三段：dispatch → 专家计算 → combine（两台服务器：4 张卡做注意力，4 张卡各放 1/4 的专家）
vec, flop_row, eff = 8192, 6 * 4096 * 1536, 494.7e12            # 每行 8 KiB（BF16），每行专家计算量，H100 有效算力
for name, recv, bw in [("均衡，跨服务器", [2048] * 4, 50e9), ("热点，跨服务器", [5120, 1024, 1024, 1024], 50e9),
                       ("热点，同一台机器 NVLink", [5120, 1024, 1024, 1024], 450e9)]:
    disp = max(recv) * vec / bw                # 最忙的接收方决定 dispatch，combine 同理
    comp = max(recv) * flop_row / eff
    print(f"{name}：dispatch {disp*1e3:.3f} + 计算 {comp*1e3:.3f} + combine {disp*1e3:.3f} = {(2*disp+comp)*1e3:.3f} ms")
d_, c_ = 2048 * vec / 50e9, 2048 * flop_row / eff
print(f"均衡时拆成两个微批次流水：{(2*d_ + c_)/2*1e3 + max(d_, c_)/2*1e3:.3f} ms（原来 {(2*d_+c_)*1e3:.3f} ms）")
```

输出：

```
EP8  ：热点卡  1108 行，平均  1024 行，最忙 ÷ 平均 = 1.08
EP32 ：热点卡   349 行，平均   256 行，最忙 ÷ 平均 = 1.36
EP256：热点卡   128 行，平均    32 行，最忙 ÷ 平均 = 4.00
随机路由 EP8  ：最忙卡平均比每卡平均多 4%
随机路由 EP32 ：最忙卡平均比每卡平均多 13%
随机路由 EP64 ：最忙卡平均比每卡平均多 21%
随机路由 EP256：最忙卡平均比每卡平均多 53%
同一台服务器（NVLink 450 GB/s）：复制 0.62 ms，第 3 批回本；热点持续 16 批净省 +3.2 ms
跨服务器（网卡 50 GB/s）：复制 5.32 ms，第 23 批回本；热点持续 16 批净省 -1.5 ms
每专家   1 个 token：CPU(AVX-512)  1.46 ms，CPU(AMX)  1.46 ms，搬到 GPU 12.51 ms
每专家 128 个 token：CPU(AVX-512) 22.23 ms，CPU(AMX)  2.57 ms，搬到 GPU 12.51 ms
AVX-512：每专家超过 71 个 token，搬到 GPU 更快
AMX：每专家超过 688 个 token，搬到 GPU 更快
均衡，跨服务器：dispatch 0.336 + 计算 0.156 + combine 0.336 = 0.827 ms
热点，跨服务器：dispatch 0.839 + 计算 0.391 + combine 0.839 = 2.068 ms
热点，同一台机器 NVLink：dispatch 0.093 + 计算 0.391 + combine 0.093 = 0.577 ms
均衡时拆成两个微批次流水：0.581 ms（原来 0.827 ms）
```

对照正文：热点卡从 EP8 的 1.08 倍到 EP256 的 4 倍；随机路由时 EP8 多 4%、EP256 多 53%；机内复制第 3 批回本、跨机第 23 批；交点 71 和 688 个 token；三段的 0.827、2.068、0.577 ms 和流水后的 0.581 ms。

容易混淆的地方

1. **MoE 省的是计算，不是通信**。每个 token 只激活 8 个专家，计算量小；可正因为专家分布在各处，每个 token 都要跨卡旅行。稠密模型的张量并行每层同步的是整个批次的隐藏状态，MoE 每个 token 要传 top-k 份。

2. **加大批次救不了网络**。批越大，网络流量同比例增加，每卡吞吐的上限不变。加大批次只能摊薄读专家权重的成本。

3. **网络带宽要看每张卡的**。一台机器有 8 张网卡，总带宽 400 GB/s，但每张卡分到的是 50 GB/s；All-to-All 时每张卡都在同时收发，看的是每张卡的带宽。

4. **EP 越大不等于越均衡**。EP 越大，每卡的专家越少，专家之间的忙闲越不能互相抵消，最忙卡相对平均越突出。大 EP 一般要配合冗余专家和动态负载均衡。

5. **重叠通信不一定变快**。拆成微批次后通信和计算能并行，但它们会争用显存带宽；争用严重时，重叠后的总时间和不重叠一样。

回顾

大规模专家并行时，每张卡每层只放几个专家，专家计算很快（EP = 64 时每层约 0.053 ms）；每个 token 每层却要送出 8 份、收回 8 份隐藏状态，DeepSeek-V3 规模下每个 token 每步约 10.0 MB 的网络流量，和批大小、EP 规模都无关。走 50 GB/s 的网络时，每层通信约 0.45 ms，是专家计算的八倍多，每卡吞吐上限约 5011 tok/s；带宽提高到 450 GB/s，上限约 45,100 tok/s。专家计算每传 1 字节网络数据只有约 4096 次运算，远低于 H100 FP8 算力与网络带宽之比，所以瓶颈在网络。对策是更快的互联（NVLink 域、超节点）、更少的字节、计算和通信重叠、专门的通信库和负载均衡。

把一层拆开看，dispatch、专家计算、combine 三段都由最忙的那份决定：跨服务器时均衡分布 0.827 ms，热点分布 2.068 ms。EP 越大越不均，一个 4 倍热点专家在 EP8 只多 8%，在 EP256 就是 4 倍；完全随机的路由在 EP256 也会让最忙的卡多 53%。冗余专家能分担热点，机内复制第 3 批回本，跨机要第 23 批。专家也可以留在 CPU 上算，每个专家的 token 数超过交点（AVX-512 约 71 个、AMX 约 688 个）才值得搬到 GPU。

prefill 和 decode 在 MoE 推理里的瓶颈如此不同，所需的卡数、并行方式也不同。稠密模型也有类似的问题：第 77 讲说过，prefill 插进来会干扰 decode。能不能干脆把两者放到不同的机器上？下一讲：**PD 分离**。

第112讲 PD 分离

68 Prefill 与 Decode　70 TTFT、TPOT 与吞吐　77 Chunked Prefill　82 推理压测实践
110 推理中的张量并行　111 MoE 推理与大规模专家并行

第 68 讲就说过，prefill 和 decode 是两种性质完全不同的工作：prefill 一次处理成百上千个 token，计算受限；decode 每步每个请求只处理 1 个 token，带宽受限。可到目前为止，它们一直挤在同一张卡上。第 77 讲的分块 prefill 让两者和平共处，但只是缓解，并没有消除互相干扰：带着 prefill 块的那一步，总比纯 decode 的步要长。

PD 分离（prefill-decode disaggregation）的想法很直接：**把 prefill 和 decode 放到不同的卡、甚至不同的机器上**。prefill 实例算完一个请求的提示，把它的 KV Cache 传给 decode 实例，由后者接着生成。这一讲看它解决了什么、付出了什么，以及什么时候值得这么做。

混在一起的问题

回顾混合部署的两个问题：

互相干扰。用分块 prefill（每步预算 2048 个 token），一步里有 prefill 块时，耗时从 5 ms 左右变成 30 ms 左右（第 77 讲）。正在 decode 的请求，每遇到这样一步，就卡一下。负载越高，长提示越多，卡顿越频繁，TPOT 的尾延迟越差。

没法各取所需。prefill 喜欢大算力，并行方式上偏好能把计算摊开的方式（比如更大的张量并行）；decode 喜欢大带宽和大显存（放更多 KV、更大的批），第 111 讲的 MoE decode 甚至要几百张卡做专家并行。混在一起时，只能选一套折中的配置。

分开之后

PD 分离的系统由三部分组成：

1. **prefill 实例**：只做 prefill。可以用大的批、大的 token 预算，追求计算效率；不用担心打断谁。
2. **decode 实例**：只做 decode。每一步都是纯粹的 decode，耗时平稳，TPOT 可预测。
3. **KV 传输**：prefill 完成后，把这个请求的 KV Cache 从 prefill 实例的显存传到 decode 实例的显存，通常走 RDMA 网络（下一讲细说）。

两边的卡数可以独立调整，比如 1 台 prefill 配 3 台 decode（记作 1P3D），按负载的特点决定比例；两边也可以用不同的并行方式，甚至不同型号的卡。

模拟一下

代码模拟了 4 张 H100 跑 Llama-3-8B：请求按泊松过程到达，提示 1000 到 3000 个 token，输出 100 到 300 个。比较三种部署：

- **混合部署**：4 张卡各自做 prefill 和 decode（分块 prefill，预算 2048）；
- **PD 分离 1P+3D**：1 张卡做 prefill，3 张做 decode；
- **PD 分离 2P+2D**：各 2 张。

每秒 12 个请求时：

- **混合部署**：TTFT 中位 36 ms、P99 70 ms；token 间隔中位 5.1 ms，但 **P99 23.7 ms、最长 32.7 ms**。
- **1P+3D**：token 间隔 P99 5.5 ms、最长 5.6 ms，非常平稳；但只有一张卡做 prefill，长提示要排队，**TTFT 的 P99 涨到 120 ms**。
- **2P+2D**：token 间隔 P99 5.8 ms、最长 5.9 ms；TTFT P99 77 ms，和混合部署相当。

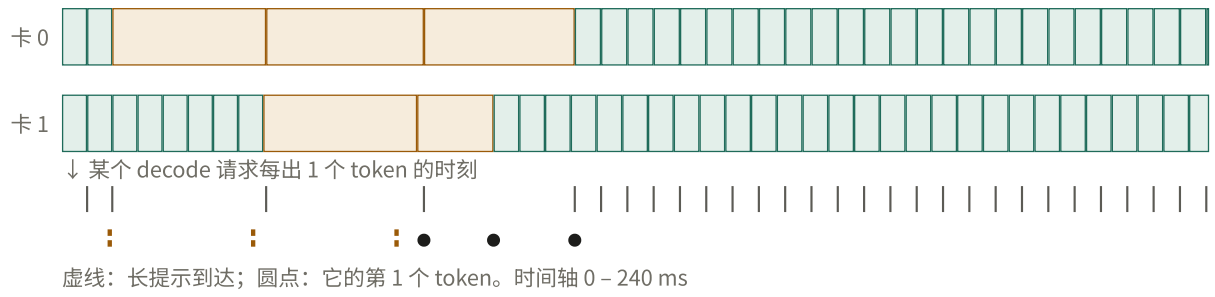
三种部署的吞吐几乎一样（约 2360 tok/s）。PD 分离换来的是 decode 的**平稳**：token 间隔的尾延迟从 23.7 ms 降到 6 ms 以内。比例选错了（1P+3D），prefill 成了瓶颈，TTFT 变差。

下面的演示是一个更小的场景：两张卡，16 个请求在 decode，先后来了三个长提示。

同样两张卡：混在一起，还是分开

⇆ 对照

混合部署（分块 prefill） | PD 分离（1 P + 1 D）



虚线：长提示到达；圆点：它的第 1 个 token。时间轴 0 – 240 ms

被跟踪的 decode 请求：token 间隔中位 5.5 ms，最长 **33.0 ms**。三个长提示的 TTFT：66 ms、50 ms、37 ms。
混合部署时，长提示的分块挤进 decode 的步里（琥珀色），这些步变长，正在 decode 的请求就卡一下。

图 112-1　混合部署与 PD 分离。混合部署时，长提示的分块挤进 decode 的步里（琥珀色），被跟踪的请求每隔一阵就卡一下；PD 分离后 decode 卡的每一步都很短。代价是 prefill 卡要排队：后到的长提示 TTFT 变长。

要点

PD 分离把 prefill 和 decode 放到不同的实例上，prefill 完成后把 KV Cache 传给 decode 实例。decode 不再被 prefill 打断，token 间隔平稳；两边可以独立选择卡数、并行方式和硬件。代价是一次 KV 传输，以及要把 prefill 和 decode 的卡数比例配好，否则一边会成为瓶颈。

代价：KV 要搬一次

一个 2000 token 的提示，Llama-3-8B 的 KV Cache 有 **262 MB**，经 50 GB/s 的网络传输约 **5.2 ms**。和 prefill 本身（约 30 ms）相比不大，而且可以**逐层传输**：prefill 算完第 1 层，就开始传第 1 层的 KV，同时算第 2 层，传输大部分能和计算重叠。

可模型越大、提示越长，KV 越多：Llama-3.1-70B 每个 token 320 KiB，一个 32K 的提示就是 10 GB，要传零点几秒。所以 PD 分离依赖高带宽的 RDMA 网络，也让 KV 小的模型（GQA、MLA，第 39、40 讲）更占优势。

比例怎么配

prefill 和 decode 的负载比，取决于请求的形状：提示越长、输出越短，prefill 越重；反之 decode 越重。代码里 1P+3D 的 prefill 成了瓶颈，2P+2D 刚好。实际系统会根据流量动态调整 P 和 D 的数量，甚至让一部分实例在两种角色之间切换。

DistServe（2024 年）的论文把这个问题表述为：在给定的 TTFT 和 TPOT 目标下，**最大化每张卡的 goodput**（第 82 讲）。因为两种 SLO 分别由两类实例负责，可以分别优化。

交接的细节：搬多久，占多少地方

上面的模拟只给了一个总数。把一次交接拆开看，它有三件事要算清楚：字节数、两端的内存、谁先谁后。

字节数。换一个更长的请求：Qwen3-8B（36 层、8 个 KV 头、头维度 128，BF16），输入 8192 个 token。每个 token 的 KV 是 2 × 36 × 8 × 128 × 2 字节 = 144 KiB，整个输入是 **1.125 GiB**。两台服务器之间按一张 200 Gbit/s 的网卡、实际 25 GB/s 算，纯传输约 **48.3 ms**；每次传输的启动开销按 5 μs 算，只占万分之一。PD 交接是一次搬一大块，拼的是带宽，不是延迟。换成 DeepSeek-V3 的 MLA（每 token 每层只存 512 + 64 个数），同样 8192 个 token 只有 549 MiB，传输时间减半。

两端同时占内存。最简单的做法是整份双缓冲：传输结束之前，prefill 一端要保留完整的 KV，decode 一端也要先分配同样大的接收空间。传输期间这份状态占了两份内存，GQA 一共 2.25 GiB。分块传输能缩短重叠的时间：decode 端确认收到哪一块，prefill 端就释放哪一块。

谁先谁后。decode 端不能一看到数据开始到达就读：要等 prefill 端发出“写完了”的完成标记，才能开始用这份状态；执行权交给 decode 之后，prefill 端才能释放源缓冲。标记本身不搬数据，只规定“先写完、后使用”的顺序。

持续到达时的总流量。每秒 8 次这样的交接要约 9.7 GB/s；一个 100 GbE 端口（每方向 12.5 GB/s）能撑住平均流量，但每份要 96.6 ms 才传完。每秒 16 次就要约 19.3 GB/s，超过了端口的能力：即使两个池的计算都有空闲，网络队列也会一直变长。换成 MLA，同一个端口能撑住每秒 16 次。所以交接是否成为瓶颈，由每份的字节数和链路带宽一起决定。

不同的卡做不同的阶段

PD 分离在同一种卡上，收益主要是延迟稳定。如果手里有两种卡，它还能提高吞吐，因为每种卡可以只做自己擅长的阶段。

设想一个具体的部署：一台服务器有 4 张 A100 80GB，另一台有 4 张 H20 96GB，跑 Qwen3-8B，每秒来 3.5 个请求，每个输入 8192 个 token、输出 1025 个 token（带推理过程的请求，输出比较长）。两种卡的峰值很不一样：

卡	BF16 稠密算力	显存带宽
A100 80GB SXM	312 TFLOP/s	2039 GB/s
H20 96GB	148 TFLOP/s	4096 GB/s

H20 的算力不到 A100 的一半，带宽却是它的两倍。实际 kernel 达不到峰值，这里两项都按 50% 算。

prefill 一个请求要多久。8192 个 token 的 prefill 是计算受限的：线性层 2 × 69.5 亿参数 × 8192，加上因果注意力，共约 133.6 TFLOPs。A100 要 **0.856 s**，H20 要 **1.805 s**。

decode 一个请求占多少 GPU 秒。decode 按批 32 算，一步要读 15.1 GB 权重和 32 个请求的 KV（平均上下文 8192 + 512 个 token），带宽受限。A100 一步约 55 ms，H20 约 27 ms。一个请求 1024 步，一步由 32 个请求分摊，A100 占 **1.764 GPU 秒**，H20 占 **0.878 GPU 秒**。

两种卡正好相反：A100 的 prefill 快一倍多，H20 的 decode 快一倍。

共置。每张卡两个阶段都做，A100 每请求占 0.856 + 1.764 = 2.62 GPU 秒，H20 占 2.68。八张卡合计每秒 **3.02** 个请求，比到达的 3.5 个少，队列会一直变长。

PD 分离。4 张 A100 做 P，每秒能 prefill 4 ÷ 0.856 = 4.67 个请求；4 张 H20 做 D，每秒能 decode 4 ÷ 0.878 = 4.55 个。链路每秒能交接 25 GB/s ÷ 1.125 GiB ≈ 20.7 份。整体的上限是三者中最小的：

$$\mu = \min(\mu_P, \mu_D, \text{链路带宽} \div \text{每请求 KV}) = \min(4.67, 4.55, 20.7) = 4.55 \text{ 请求/s}$$

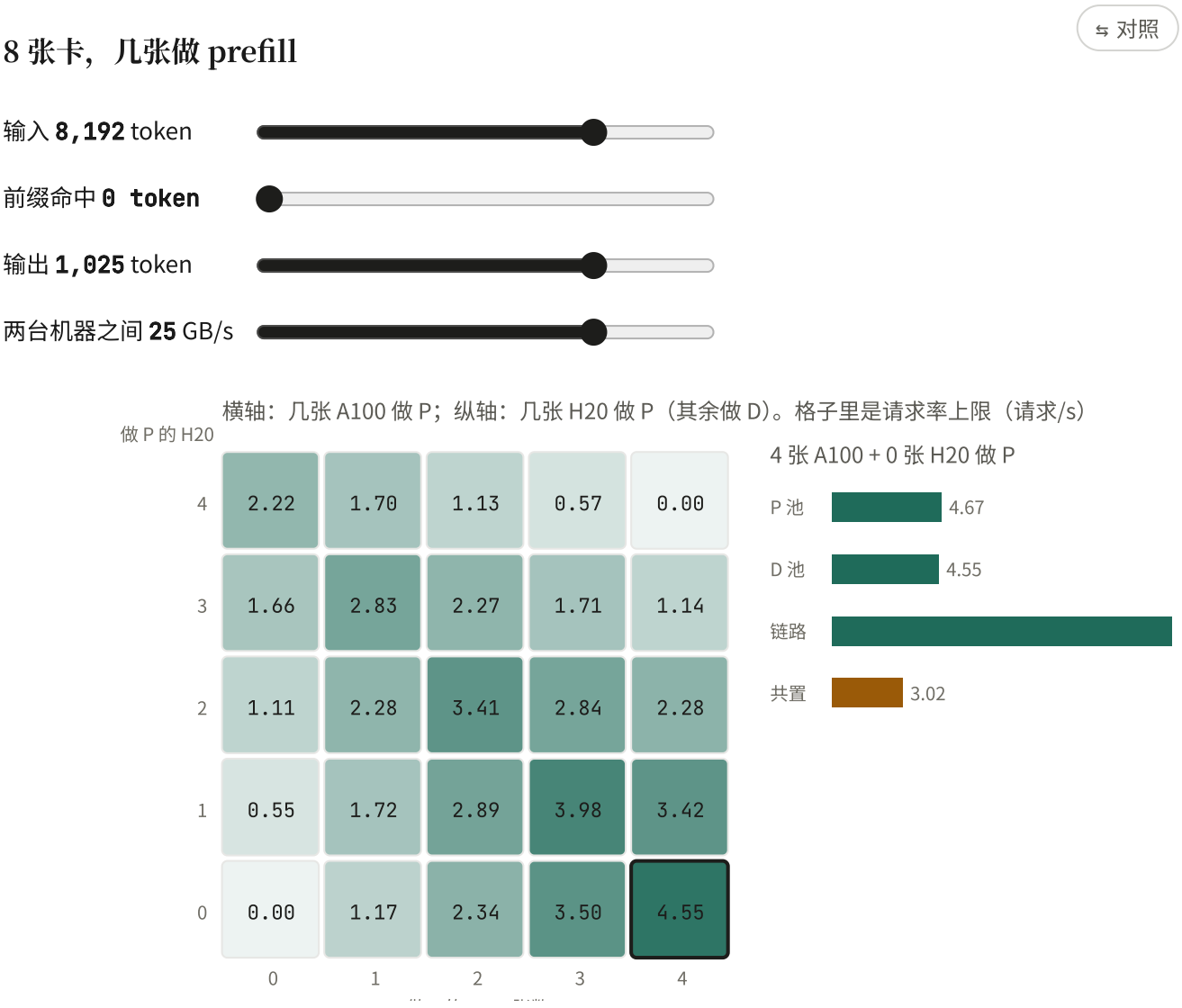
是共置的 1.51 倍，够处理每秒 3.5 个请求。计算量一点没少，变的只是谁来做。角色对调（H20 做 P、A100 做 D）则只有 **2.22** 请求/s，不到正确分工的一半。

链路什么时候会成为瓶颈？4.55 请求/s 时，GQA 的 KV 需要链路至少 **5.5 GB/s**，MLA 只要 **2.6 GB/s**。高于这个值，再加带宽不会提高吞吐。

同一种卡呢？ 8 张 A100 共置是 3.05 请求/s；分成 3 张做 P、5 张做 D，P 池 3.50、D 池 2.83，上限反而降到 **2.83**。每个请求消耗的 GPU 秒没变，卡数又只能按整数分，总有一边的能力闲着。对比一下第 77 讲的分块 prefill：A100 decode 一步 55 ms 里，矩阵计算只用了约 4 ms，剩下的算力足够把 prefill 切成小块捎带进去。如果捎带的计算能完全藏在读显存的时间里，八张卡的上限只受 decode 限制，是 8 ÷ 1.764 = **4.53** 请求/s。所以在同构集群上，追求吞吐先用分块 prefill；PD 分离的价值是两个阶段各选各的并行方式和批大小，并彻底隔开彼此的排队。H20 decode 一步本来就短，空闲的算力很少，分块捎带帮不上忙，异构集群的收益来自硬件本身的差异，分块替代不了。

请求一变，配比就变

最优的配比不是固定的。下面的演示可以改输入长度、前缀命中、输出长度和链路带宽，看 25 种分法各自的上限。



每个请求：A100 prefill 0.856 s、decode 1.764 GPU 秒；H20 prefill 1.805 s、decode 0.878 GPU 秒。每请求交接 KV 1.125 GiB，链路每秒最多 20.7 份。
选中格：min(P 4/0.856 + 0/1.805 = 4.67, D 0/1.764 + 4/0.878 = 4.55, 链路 20.70) = **4.55 请求/s**。最优 4 张 A100 + 0 张 H20 做 P：4.55 请求/s，是共置 3.02 的 1.51 倍。
A100 算力强，适合 prefill；H20 带宽强，适合 decode。前缀命中让 P 的工作变少，卡往 D 挪；输出变短让 D 的工作变少，卡往 P 挪。点格子看它的三项。

图 112-2　4 张 A100 + 4 张 H20 的 25 种分法（Qwen3-8B，效率按峰值 50%）。每格是 min(P 池, D 池, 链路) 的请求率上限，橙框是最优配比。默认的 8K 输入、1025 输出时，4 张 A100 做 P、4 张 H20 做 D 最好，4.55 请求/s。

前缀命中。前 6144 个 token 已经有缓存，P 只需处理 2048 个。A100 的 prefill 从 0.856 s 降到 0.238 s，4 张 A100 做 P 太多了。最优配比变成 **2 张 A100 做 P**，另外 2 张 A100 和 4 张 H20 做 D，上限升到 **5.69** 请求/s。注意 decode 一端没有缓存，交接的 KV 仍是完整的 1.125 GiB。

输出变短。恢复完整输入，输出只有 129 个 token（普通对话）。D 几乎不再是瓶颈，最优配比变成 **4 张 A100 + 3 张 H20 做 P**，只留 1 张 H20 做 D，上限 **6.33** 请求/s。这时共置已经有 5.84 请求/s，分离只多了 8%。

三种请求下，最合适的 P 卡数分别是 4、2、7 张。前缀命中让 P 的工作变少，把卡推向 D；输出变短让 D 的工作变少，把卡推向 P。推理模型的输出越长，D 的比重越大，H20 这类带宽强、算力弱的卡越有价值。实际系统因此要在两个时间尺度上调度：请求到达时选进哪个队列，负载分布慢慢变化时调整两个池的大小。

到了大模型里

大规模部署的常见架构。DeepSeek-V3 的推理就是 PD 分离的：prefill 用 32 卡的单元，decode 用 320 卡的单元（第 108 讲），两边的并行方式完全不同。月之暗面的 Mooncake 系统也是以 PD 分离和 KV Cache 为中心设计的。vLLM、SGLang 都提供 PD 分离的部署方式，配合 NIXL、Mooncake Transfer Engine 等传输库。

不是所有场景都需要。规模小、提示短、对 TPOT 尾延迟不敏感的服务，混合部署加分块 prefill 已经够好，PD 分离反而增加了系统的复杂度（两类实例、KV 传输、路由）。PD 分离的收益在**大规模、长提示、严格的 TPOT 要求**下最明显。

和前缀缓存的配合。多轮对话里，上一轮的 KV 在 decode 实例上，下一轮的 prefill 却在 prefill 实例上做。要想命中缓存，就得把 KV 存在两边都能访问的地方，或者把请求路由到有缓存的实例（第 113、114 讲）。

用代码复现

下面的代码模拟 4 张 H100、Llama-3-8B 的三种部署：混合部署（分块 prefill）、PD 分离 1P+3D 和 2P+2D，在每秒 6 个和 12 个请求下，统计 TTFT、token 间隔（第 2 个 token 以后）的中位数、P99 和最大值，以及吞吐；最后算一个 2000 token 提示的 KV 传输时间。每步耗时用第 68 讲的估算。

```
import math
# 4 张 H100 跑 Llama-3-8B。混合部署：4 张卡各自既做 prefill 也做 decode（分块 prefill，每步预算 2048）；
# PD 分离：P 张卡只做 prefill，其余只做 decode，prefill 完的 KV 经 50 GB/s 网络传给 decode 卡
N, NE, L, d, W, KV, F, BW = 8.03e9, 128256 * 4096, 32, 4096, 16.06e9, 131072, 989e12, 3.35e12

def step_time(work):                                    # work: [(新 token 数, 前面已有的 token 数)]，第 68 讲的估算
    flops = sum(2 * (N - NE) * c + 4 * L * d * c * (p + c / 2) for c, p in work)
    return max(flops / F, (W + KV * sum(p + c for c, p in work)) / BW)

def lcg(seed):
    while True: seed = (seed * 1103515245 + 12345) % 2**31; yield seed / 2**31

def workload(rate, n=400):
    r, t, out = lcg(7), 0.0, []
    for _ in range(n):
        t += -math.log(1 - next(r)) / rate
        out.append(dict(at=t, p=1000 + int(next(r) * 2000), o=100 + int(next(r) * 200)))
    return out

def run(reqs, n_prefill):
    gpus = [dict(kind="mixed" if n_prefill == 0 else ("prefill" if g < n_prefill else "decode"), free=0.0, q=[], run=[]) for g in range(4)]
    pending = sorted(reqs, key=lambda q: q["at"]); handoff = []          # handoff: (到达 decode 卡的时间, 请求)
    for q in reqs: q.update(done=0, gen=0, times=[])
    while pending or handoff or any(g["q"] or g["run"] for g in gpus):
        g = min(gpus, key=lambda g: g["free"]); now = g["free"]
        while pending and pending[0]["at"] <= now:                       # 新请求：混合部署给最空闲的卡；分离时给 prefill 卡
            q = pending.pop(0)
            cand = [x for x in gpus if x["kind"] in ("mixed", "prefill")]
            min(cand, key=lambda x: len(x["q"]) + len(x["run"]))["q"].append(q)
        for h in [h for h in handoff if h[0] <= now]:                    # KV 传到了：放进 decode 卡
            handoff.remove(h); min([x for x in gpus if x["kind"] == "decode"], key=lambda x: len(x["run"]))["run"].append(h[1])
        work, budget = [], (2048 if g["kind"] == "mixed" else 8192)
        if g["kind"] in ("mixed", "decode"):
            for q in g["run"]: work.append((q, 1)); budget -= 1
        if g["kind"] in ("mixed", "prefill"):
            for q in g["q"]:
```

```
                if budget <= 0: break
                c = min(budget, q["p"] - q["done"]); work.append((q, c)); budget -=
c
        if not work:                                                    # 没活：等
下一个事件
            nxt = [x["at"] for x in pending[:1]] + [h[0] for h in handoff] +
[x["free"] for x in gpus if x is not g and (x["q"] or x["run"])]
            g["free"] = max(now + 1e-4, min(nxt)) if nxt else now + 1e-3; continue
        dt = step_time([(c, q["done"] + q["gen"]) for q, c in work]); g["free"] =
now + dt
        for q, c in work:
            if q["done"] < q["p"]:
                q["done"] += c
                if q["done"] < q["p"]: continue
                g["q"].remove(q); q["gen"] = 1; q["times"].append(g["free"])  #
prefill 完成，出第一个 token
                if g["kind"] == "prefill": handoff.append((g["free"] + q["p"] * KV
/ 50e9, q))
                else: g["run"].append(q)
            else:
                q["gen"] += 1; q["times"].append(g["free"])
                if q["gen"] == q["o"]: g["run"].remove(q)
    return reqs

def pct(xs, p): xs = sorted(xs); return xs[min(len(xs) - 1, int(p / 100 *
len(xs)))]
for rate in [6, 12]:
    for name, P in [("混合部署 4 卡", 0), ("PD 分离 1P+3D", 1), ("PD 分离 2P+2D",
2)]:
        rs = run(workload(rate), P)
        ttft = [(q["times"][0] - q["at"]) * 1e3 for q in rs]
        gaps = [(b - a) * 1e3 for q in rs for a, b in zip(q["times"][1:],
q["times"][2:])]   # 第 2 个 token 以后的间隔
        span = max(q["times"][-1] for q in rs) - rs[0]["at"]
        print(f"每秒 {rate} 个请求，{name}: TTFT 中位 {pct(ttft, 50):5.0f} / P99
{pct(ttft, 99):5.0f} ms；"
              f"token 间隔 中位 {pct(gaps, 50):4.1f} / P99 {pct(gaps, 99):5.1f} / 最
大 {max(gaps):5.1f} ms；吞吐 {sum(q['o'] for q in rs) / span:5.0f} tok/s")
print(f"一个 2000 token 的提示，KV 有 {2000 * KV / 1e6:.0f} MB，经 50 GB/s 网络传输约
{2000 * KV / 50e9 * 1e3:.1f} ms")
```

输出：

```
每秒 6 个请求，混合部署 4 卡：TTFT 中位    35 / P99    65 ms；token 间隔 中位  5.0 / P99
5.2 / 最大  32.2 ms；吞吐  1204 tok/s
每秒 6 个请求，PD 分离 1P+3D：TTFT 中位    33 / P99    83 ms；token 间隔 中位  5.0 / P99
5.2 / 最大   5.2 ms；吞吐  1204 tok/s
每秒 6 个请求，PD 分离 2P+2D：TTFT 中位    33 / P99    75 ms；token 间隔 中位  5.1 / P99
5.4 / 最大   5.5 ms；吞吐  1203 tok/s
每秒 12 个请求，混合部署 4 卡：TTFT 中位    36 / P99    70 ms；token 间隔 中位  5.1 /
P99  23.7 / 最大  32.7 ms；吞吐  2359 tok/s
每秒 12 个请求，PD 分离 1P+3D：TTFT 中位    38 / P99   120 ms；token 间隔 中位  5.2 /
P99   5.5 / 最大   5.6 ms；吞吐  2362 tok/s
每秒 12 个请求，PD 分离 2P+2D：TTFT 中位    35 / P99    77 ms；token 间隔 中位  5.4 /
P99   5.8 / 最大   5.9 ms；吞吐  2360 tok/s
一个 2000 token 的提示，KV 有 262 MB，经 50 GB/s 网络传输约 5.2 ms
```

每秒 6 个请求时差别还不大，只有混合部署的最长间隔（32.2 ms）明显偏高。每秒 12 个请求时，混合部署 token 间隔的 P99 涨到 23.7 ms，PD 分离始终在 6 ms 以内；1P+3D 的 TTFT P99 是 120 ms，2P+2D 是 77 ms。

这个模拟不考虑 KV 传输和 prefill 的重叠、显存上限和抢占，负载均衡也很简单，只用来说明趋势。

下面的代码复现异构集群的配比：由峰值和 50% 的效率推出两种卡在两个阶段的时间，算共置、PD 最优配比、角色对调、同构集群和两种请求变化下的上限。

```
# Qwen3-8B (BF16) 在 A100 与 H20 上：两个阶段每个请求各占多少 GPU 秒，PD 分离后怎么配比
L, d, ffn, nq, nkv, hd, vocab = 36, 4096, 12288, 32, 8, 128, 151936
lin = L * (d * (nq + 2 * nkv) * hd + nq * hd * d + 3 * d * ffn)   # 每层线性层参数 × 层数
W = 2 * (lin + vocab * d)                                          # decode 每步读的权重（含输出层）
KV = 2 * L * nkv * hd * 2                                          # 每 token 的 KV：144 KiB
GPU = {"A100": (312e12, 2039e9), "H20": (148e12, 4096e9)}          # BF16 稠密算力、显存带宽（峰值）
EFF, BATCH = 0.5, 32                                               # 实际只发挥峰值的 50%；decode 批 32

def need(gpu, prompt, cached, out):            # 一个请求在这张卡上的 prefill 秒数、decode GPU 秒数
    flops, bw = GPU[gpu][0] * EFF, GPU[gpu][1] * EFF
    new = prompt - cached
    f = 2 * lin * new + 4 * nq * hd * L * (prompt**2 - cached**2) / 2   # 线性层 + 因果注意力
    step = (W + BATCH * (prompt + out / 2) * KV) / bw                   # 一步 decode：读权重 + 32 个请求的 KV
    return f / flops, (out - 1) * step / BATCH

def best(prompt, cached, out, link=25e9):
    a, h = need("A100", prompt, cached, out), need("H20", prompt, cached, out)
    co = 4 / sum(a) + 4 / sum(h)                                        # 共置：每张卡两个阶段都做
    grid = {(i, j): min(i / a[0] + j / h[0], (4 - i) / a[1] + (4 - j) / h[1], link / (prompt * KV))
            for i in range(5) for j in range(5)}                        # i 张 A100、j 张 H20 做 P，其余做 D
    (i, j), mu = max(grid.items(), key=lambda kv: kv[1])
    return a, h, co, grid, i, j, mu

print(f"每步读权重 {W/1e9:.1f} GB，每 token KV {KV/1024:.0f} KiB，8192 token 的 KV {8192*KV/2**30:.3f} GiB")
a, h, co, grid, i, j, mu = best(8192, 0, 1025)
print(f"A100：prefill {a[0]:.3f} s，decode {a[1]:.3f} GPU 秒；H20：prefill {h[0]:.3f} s，decode {h[1]:.3f} GPU 秒")
print(f"共置（8 张卡都做两个阶段）：{co:.2f} 请求/s")
print(f"PD 最佳：{i} 张 A100 + {j} 张 H20 做 P，{mu:.2f} 请求/s；角色对调：{grid[(0, 4)]:.2f} 请求/s")
print(f"链路要至少 {mu * 8192 * KV / 1e9:.1f} GB/s 才不成为瓶颈（MLA 的 549 MiB 只要 {mu * 549 * 2**20 / 1e9:.1f} GB/s）")
# 同构：8 张 A100
pa, da = a
```

```
print(f"8 张 A100：共置 {8/(pa+da):.2f}，3P5D {min(3/pa, 5/da):.2f}，分块捎带的理想上限
{8/da:.2f} 请求/s")
for name, args in [("前缀命中 6144", (8192, 6144, 1025)), ("输出只有 129", (8192, 0,
129))]:
    a, h, co, grid, i, j, mu = best(*args)
    print(f"{name}：最佳 {i} 张 A100 + {j} 张 H20 做 P，{mu:.2f} 请求/s（共置
{co:.2f}）")
```

输出：

```
每步读权重 15.1 GB，每 token KV 144 KiB，8192 token 的 KV 1.125 GiB
A100：prefill 0.856 s，decode 1.764 GPU 秒；H20：prefill 1.805 s，decode 0.878 GPU 秒
共置（8 张卡都做两个阶段）：3.02 请求/s
PD 最佳：4 张 A100 + 0 张 H20 做 P，4.55 请求/s；角色对调：2.22 请求/s
链路要至少 5.5 GB/s 才不成为瓶颈（MLA 的 549 MiB 只要 2.6 GB/s）
8 张 A100：共置 3.05，3P5D 2.83，分块捎带的理想上限 4.53 请求/s
前缀命中 6144：最佳 2 张 A100 + 0 张 H20 做 P，5.69 请求/s（共置 4.90）
输出只有 129：最佳 4 张 A100 + 3 张 H20 做 P，6.33 请求/s（共置 5.84）
```

对照正文：A100 的 prefill 0.856 s、decode 1.764 GPU 秒，H20 为 1.805 s、0.878 GPU 秒；共置 3.02，PD 4.55，对调 2.22 请求/s；8 张 A100 时共置 3.05、3P5D 2.83、分块捎带的理想上限 4.53；前缀命中后 2 张 A100 做 P 得 5.69，输出变短后 7 张卡做 P 得 6.33。

容易混淆的地方

1. **PD 分离不提高单个请求的速度**。prefill 和 decode 的计算量一点没少，还多了一次传输。它改善的是延迟的**稳定性**和资源的**匹配度**。

2. **分块 prefill 和 PD 分离不矛盾**。分块 prefill 是“混在一起时减少干扰”，PD 分离是“干脆分开”。PD 分离的 prefill 实例上，长提示依然可以分块，以便和 KV 传输流水起来。

3. **PD 分离的 TTFT 不一定更好**。prefill 实例不够时，请求要排队，TTFT 反而变差；KV 传输也会让第 2 个 token 来得晚一点。

4. **同构集群上，PD 分离不提高吞吐**。每个请求消耗的 GPU 秒不变，卡数还只能按整数分。提高吞吐要靠分块 prefill 把空闲的算力用起来；PD 分离换来的是隔离。异构集群不同，每种卡做擅长的阶段，吞吐能提高一半。

回顾

prefill 计算受限、decode 带宽受限，混在一起时，prefill 会打断 decode，两者也没法各自选择最合适的配置。PD 分离把它们放到不同的实例上，prefill 完成后把 KV Cache 传给 decode 实例。模拟的 4 卡、每秒 12 个请求下，混合部署 token 间隔 P99 23.7 ms、最长 32.7 ms；2P+2D 的 PD 分离 P99 5.8 ms，TTFT 相当，吞吐不变；1P+3D 则 prefill 不够，TTFT P99 涨到 120 ms。代价是 KV 传输（2000 token 的 8B 模型约 262 MB、5.2 ms，可逐层重叠）和配比的调优。它在大规模、长提示、严格 TPOT 要求的场景收益最大。

换一个更具体的场景：Qwen3-8B 的 8K 请求每次交接 1.125 GiB、在 25 GB/s 链路上约 48.3 ms，传输期间两端各占一份。4 张 A100 加 4 张 H20 时，共置只有 3.02 请求/s；A100 做 prefill、H20 做 decode 是 4.55 请求/s，对调后只有 2.22。最优配比随请求变化：前缀命中时 2 张卡做 P，输出变短时 7 张卡做 P。

PD 分离让 KV Cache 不再只待在一张卡的显存里：它要在机器之间传递，还可能被存下来给以后的请求复用。显存放不下的 KV 能不能放到 CPU 内存、SSD 里？从哪里读回来比重新计算更划算？下一讲：**KV Cache 传输与分层存储**。

第113讲 KV Cache 传输与分层存储

69 KV Cache　75 PagedAttention　76 Prefix Cache　94 KV Cache 量化　112 PD 分离

在第三章里，KV Cache 一直待在一张卡的显存里：请求开始时分配，结束时释放（第 75 讲），前缀缓存让它多留一会儿（第 76 讲）。到了集群里，它开始“流动”起来：

- PD 分离（上一讲）要把它从 prefill 实例的显存**传到** decode 实例的显存；
- 显存放不下的前缀缓存，可以**挪到** CPU 内存、SSD，甚至远端的存储里，等需要时再读回来；
- 一台机器算过的前缀，可以被**另一台机器**拿去用。

这一讲先算一笔最基本的账：一段已经算过的 KV，从哪一层存储读回来，比重新计算更快？然后看 KV 在这些层级之间流动时要注意什么。

读回来，还是重算

一段 n 个 token 的前缀，如果它的 KV 存在某个地方，用的时候有两个选择：

- **读回来**：耗时 = n × 每 token 的 KV 字节数 ÷ 存储到显存的带宽。
- **重新 prefill**：耗时 ≈ n × 2 × 激活参数 ÷ 算力（prefill 是计算受限的，第 68 讲；这里只算线性层）。

两者都和 n 成正比。所以比较的是**每个 token**：读它的 KV 要多久，重算它要多久。两者相等时的存储带宽叫**分界带宽**：

$$\text{分界带宽} = \text{每 token 的 KV 字节数} \div \text{每 token 的重算时间}$$

比分界带宽快的存储，读回来更划算；比它慢的，不如重算。代码算了三个模型：

- **Llama-3-8B**：每 token KV 128 KiB，重算 15.2 µs，分界带宽 **8.6 GB/s**。
- **Llama-3.1-70B**：320 KiB，140.5 µs，分界带宽 **2.3 GB/s**。
- **DeepSeek-V3**：MLA 每 token 只有 68.6 KiB，重算 37.4 µs（FP8），分界带宽 **1.9 GB/s**。

模型越大，每个 token 算得越久；KV 越小（GQA、MLA），读得越快。这两点都让分界带宽变低，**连很慢的存储都值得用来存 KV**。

存储的层级

再看各层存储的带宽（示意的量级，按一台 8 卡服务器）：

存储	到 GPU 的带宽	容量
本卡显存 HBM	约 3.35 TB/s	约 50 GB（扣掉权重后）
同机其他卡（NVLink）	约 450 GB/s	约 400 GB
CPU 内存（PCIe）	约 50 GB/s	1–2 TB
别的机器（RDMA 网络）	约 50 GB/s	整个集群
本机 NVMe SSD	约 10 GB/s	几十 TB
远端分布式存储	约 3 GB/s	几乎无限

拿 Llama-3-8B 一段 8000 token 的前缀算一下（KV 共 1.05 GB，重算约 121 ms）：从 CPU 内存读回约 21 ms，从别的机器读约 21 ms，从本机 SSD 读约 105 ms，都比重算快；从远端存储读要 350 ms，就不如重算了。换成 70B，重算要 1124 ms（单卡计；多卡张量并行时两者按同样比例缩小），连远端存储（874 ms）都比重算快。

容量上，CPU 内存比显存大几十倍：2 TB 的内存能存 Llama-3-8B 的约 1500 万个 token 的 KV，DeepSeek-V3 的约 2800 万个。显存里放不下的前缀缓存，挪到内存里，命中率就能高得多。

下面的计算器可以选模型、调前缀长度，看每一层的读回时间和重算时间。

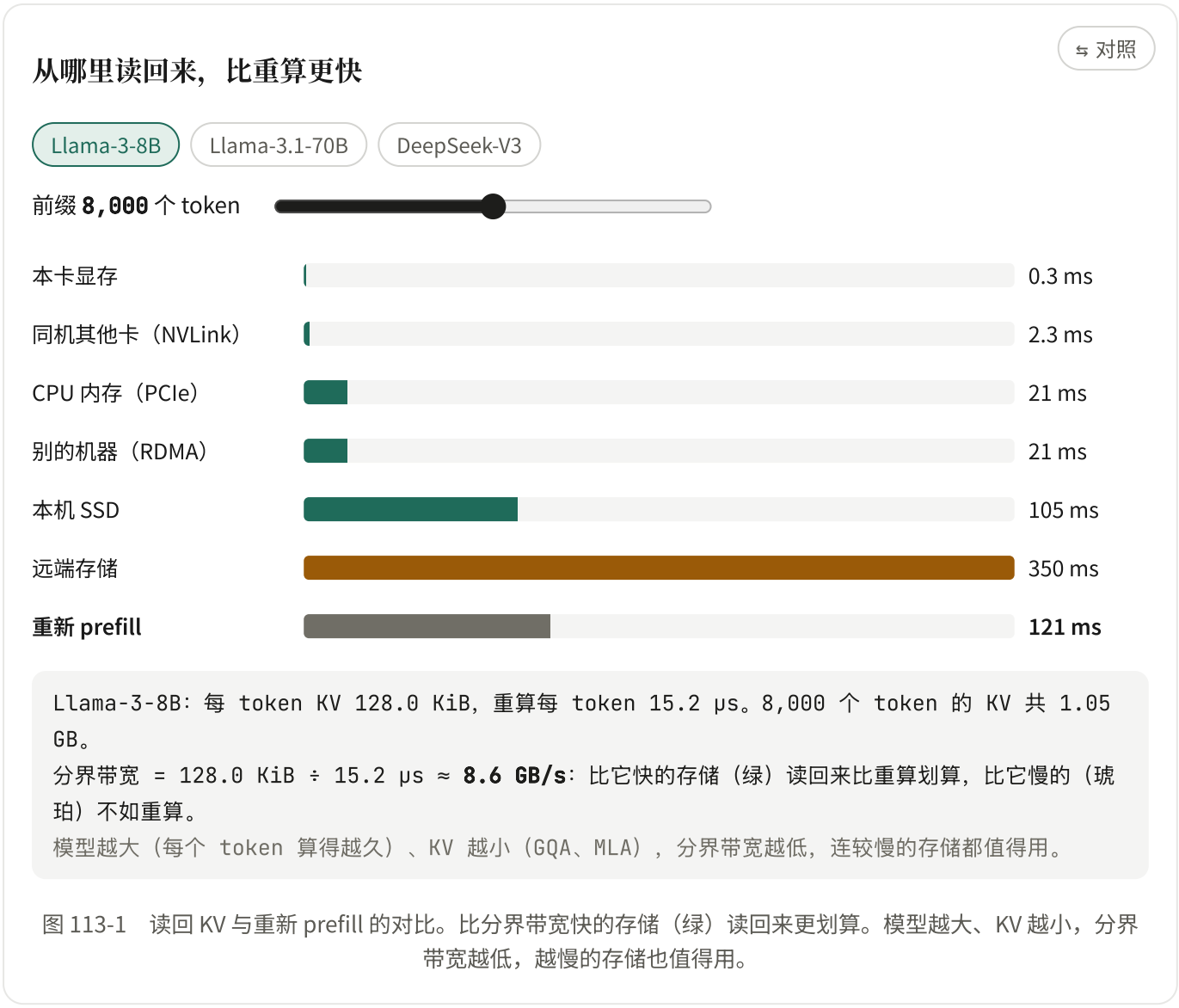


图 113-1　读回 KV 与重新 prefill 的对比。比分界带宽快的存储（绿）读回来更划算。模型越大、KV 越小，分界带宽越低，越慢的存储也值得用。

要点

一段算过的前缀，读回它的 KV 还是重算，取决于存储带宽是否高于分界带宽 = 每 token KV ÷ 每 token 重算时间。Llama-3-8B 约 8.6 GB/s，70B 约 2.3 GB/s，DeepSeek-V3 约 1.9 GB/s：CPU 内存、RDMA 网络、甚至 SSD 都比重算快。所以 KV Cache 可以从显存扩展到内存、SSD 和整个集群，成为一个分层的缓存。

流动时要注意什么

上面的账只算了带宽，真实的系统还要处理几件事：

和计算重叠。读回一段前缀，不必等全部读完才开始算：先读第 1 层的 KV，第 1 层就可以开始算新 token，同时读第 2 层。PD 分离的传输也一样，prefill 每算完一层就开始传这一层（上一讲）。做得好的话，传输的时间大部分被藏住了。

传输的粒度。分页的块只有 16 个 token（第 75 讲），一层一块只有几十 KB，一块一块地传，α 的开销太大（第 102 讲）。传输时要把很多块合并成大的传输，或者用支持“分散-聚集”的 RDMA 操作。

不要绕道 CPU。GPU 之间、GPU 和网卡之间的数据，最好直接走 GPUDirect RDMA，不经过 CPU 内存中转，否则要多拷贝一次，还会抢 PCIe 的带宽。

谁有哪段前缀。KV 散落在各台机器的显存、内存、SSD 里，就需要一个“目录”：记录每段前缀（按第 76 讲的块哈希）存在哪里。请求来了，先查目录，再决定是读回、还是重算，以及把请求发给谁（下一讲）。

淘汰策略。每一层都会满，按 LRU 等策略把冷的数据往下一层挪，最后丢弃。热门的前缀（比如常用的系统提示）留在上层。

存多久：容量和复用间隔

上面的账只问了“读回来快不快”。可一份 KV 要能读回来，前提是它还在。缓存要多大，取决于它要保存多久。

换一个具体的例子：Qwen3-8B（每 token 144 KiB），一张 A100，按峰值 50% 计，重新 prefill 一段 8192 token 的前缀要 0.856 s（第 112 讲）。如果这张卡一直在做没有命中的 8K prefill，每 0.856 s 就产生一份 1.125 GiB 的 KV，产生速率

$$r = 1.125\ \text{GiB} \div 0.856\ \text{s} \approx 1.41\ \text{GB/s}$$

注意，r 正是上一节的**分界带宽**：每 token 的 KV 字节数除以每 token 的重算时间。

所有新 KV 都写进某一级、按先后淘汰时，容量为 C 的一级大约能保留最近 $C \div r$ 秒内产生的 KV：HBM（扣掉权重约 63.6 GB）约 **45 s**，每卡 256 GiB 的主机内存约 **195 s**，一块 3.84 TB 的 SSD 约 **2721 s**（45 分钟）。反过来，同一前缀两次使用的间隔是 T，要让间隔 T 以内的复用都能命中，这一级的容量至少要

$$C \geqslant r \times T$$

T 由谁发起下一轮决定。阿里云一项基于线上请求记录的研究发现：面向个人用户的对话里，80% 的复用发生在 10 分钟以内；面向企业的 API 调用里，80% 的复用发生在 10 秒以内。10 秒只要 14.1 GB，HBM 就够；10 分钟要约 **847 GB**，超过了每卡的主机内存，要加上 SSD。

容量越大，命中率涨得越慢。不是每份 KV 都会被再用。Mooncake 公开的 Kimi 线上一小时采样记录里，按 LRU 淘汰，缓存从 1000 块（每块 512 token）加到 50,000 块，命中率只从 30% 升到 50%；容量不限也只有 51%。这份记录中超过一半的块从未被再次使用，另一些块却被访问上万次。所以加容量之前，要先看命中率曲线是不是已经平了。

读回来：多少次才回本，怎么和计算重叠

保存也要花时间。把一份 KV 存下来，将来复用 k 次，比每次都重算更快的条件是

$$T_{写} + k \times T_{读} < k \times T_{重算}$$

经 25 GB/s 的链路，写、读各约 48.3 ms，复用一次就回本。链路降到 2 GB/s，读一次要 604 ms，要复用 3 次才回本。降到 10 GbE 的 1.25 GB/s，读一次 966 ms，已经比重算的 856 ms 还慢，怎么复用都不划算。临界点就是 r：链路比 GPU 产生 KV 的速度还慢，读回来就不如重算。

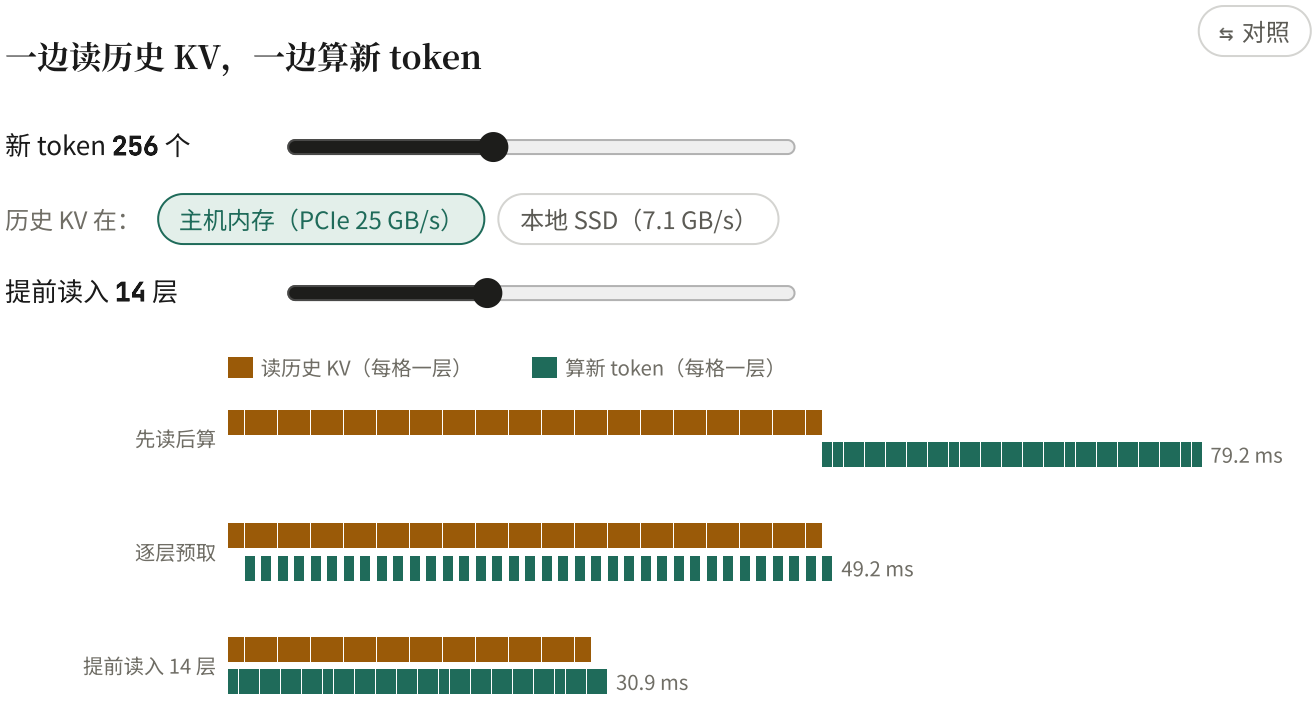
读和算重叠。命中之后，只需计算 256 个新 token，约 30.9 ms。历史 KV 在主机内存里，整份读回要 48.3 ms，先读后算共 **79.2 ms**，读比算还长。

Transformer 是一层一层算的，第 i 层只用第 i 层的 KV。所以可以**逐层预取**：GPU 算第 i 层时，PCIe 在读后面的层。可每层读要 1.34 ms、算只要 0.86 ms，每层的计算都在等读取，总时间约 **49.2 ms**，仍由读取决定。

再进一步：在这个请求开始之前，利用上一个 batch 的执行时间，先把前面若干层读进显存里预留的缓冲区。缓冲区要装下的，正是读比算多出来的字节：

$$\text{缓冲} = \text{链路带宽} \times (T_{读} - T_{算}) = 25\ \text{GB/s} \times (48.3 - 30.9)\ \text{ms} \approx 436\ \text{MB}，即 14\ 层$$

提前读入 14 层，剩下 22 层的读取被计算完全盖住，总时间降到计算本身的 **30.9 ms**。新 token 达到约 400 个时，计算就长过读取，不需要缓冲。CachedAttention 在 LLaMA-13B 上实测，逐层预取让 prefill 缩短 35%，再加 15 层的缓冲区缩短 61%。



读全部 1.125 GiB：48.3 ms（每层 1.34 ms）；算 256 个新 token：30.9 ms（每层 0.857 ms）。读比算多 17.5 ms，要提前读入 B·(T_读 - T_算) = 436 MB，即 **14 层**，读取才能被完全盖住。

Transformer 一层一层地算，第 i 层只用第 i 层的 KV，所以读和算可以错开。提前读入的层可以利用前一个 batch 的执行时间，或请求还在排队的时间。

图 113-2　命中 8K 前缀后，读历史 KV（橙）和算 256 个新 token（绿）的时间线。先读后算 79.2 ms；逐层预取 49.2 ms，每层都在等读取；提前读入 14 层，总时间就是计算本身的 30.9 ms。换成 SSD，读一层要 4.73 ms，提前读入的层数要多得多。

用排队时间从 SSD 预取。SSD 读一层要 4.73 ms，是算一层的 5.5 倍，逐层预取只能盖住一小部分。可以利用排队时间：请求还在队列里时，调度器已经知道它要哪份前缀，先把它从 SSD 读到主机内存；只要排队时间长过约 170 ms，这次读取就不在关键路径上。主机内存装得下多少份，就为队列最前面的多少个请求预取；换出时，先换出下次使用最晚的。这比只看过去访问的 LRU 好得多：CachedAttention 回放 ShareGPT 的多轮对话，按队列预取和换出时命中率 86%，LRU 只有 58%。

SSD：带宽够，寿命不够

SSD 的读写带宽都比 r 高，写入 1.41 GB/s 也只占 PCIe 的一小部分。真正的限制是**写入寿命**。数据中心 SSD 常标“1 DWPD”：五年内每天可以写满一次。3.84 TB 的盘平均每秒只能写 44.4 MB，只有 r 的 3.1%。如果把所有新 KV 都写进 SSD，五年的寿命不到两个月就用完了。

所以写入 SSD 之前要先判断值不值得：比如只写已经被复用过的前缀，或者还没结束的多轮会话。前面看到超过一半的块从来不会再用，这样的准入判断能省下大部分写入。

到了大模型里

推理引擎的支持。vLLM 支持把 KV 卸载到 CPU 内存，并通过 LMCache 等插件接入更大的存储；SGLang 的 HiCache 把前缀缓存分成显存、内存、存储三层。它们都配合 NIXL、Mooncake Transfer Engine 这类传输库，在 GPU、CPU、网卡之间高效地搬运 KV。

以 KV 为中心的架构。月之暗面的 Mooncake 把整个集群的 CPU 内存和 SSD 组成一个分布式的 KV 缓存池，PD 分离的 prefill、decode 实例都从这个池子里读写。它的出发点就是这一讲的结论：对长上下文的对话服务，存起来复用比重算划算得多。

API 的缓存价格。DeepSeek 等厂商的 API 提供“上下文缓存”（DeepSeek 称为硬盘缓存），命中缓存的输入 token 按低得多的价格计费（第 72、76 讲）。MLA 让每个 token 的 KV 很小，存在 SSD 上也划算，这正是它能把缓存做到磁盘上的原因。

用代码复现

下面的代码列出各层存储的带宽，对三个模型算出每 token 的 KV 大小、重算时间和分界带宽，并比较一段 8000 token 的前缀从各层读回和重算的时间；最后算 2 TB 的 CPU 内存能存多少 token。

```
# 一段已经算过的前缀，再用到时：从某一层存储把它的 KV 读回来，还是重新 prefill？
# 存储层级（带宽是示意的量级；容量按一台 8 卡服务器估）
TIERS = [("本卡显存 HBM", 3350e9, "约 50 GB（扣掉权重）"), ("同机其他卡 NVLink", 450e9,
"约 400 GB"),
         ("CPU 内存 经 PCIe", 50e9, "1-2 TB"), ("别的机器 经 RDMA 网络", 50e9, "整个集
群"),
         ("本机 NVMe SSD", 10e9, "几十 TB"), ("远端分布式存储", 3e9, "几乎无限")]
# 模型：每个 token 的 KV 字节数，重新 prefill 每个 token 的计算时间（计算受限，2 × 激活参数
÷ 算力）
MODELS = {"Llama-3-8B": (131072, 2 * 7.5e9 / 989e12),
          "Llama-3.1-70B": (327680, 2 * 69.5e9 / 989e12),
          "DeepSeek-V3（MLA，FP8 计算）": (61 * 576 * 2, 2 * 37e9 / 1979e12)}
n = 8000                                                    # 一段 8000 token 的前缀（比
如一份长文档或长对话历史）
for name, (kv, t_tok) in MODELS.items():
    print(f"{name}：每 token KV {kv / 1024:.1f} KiB，重算每 token {t_tok * 1e6:.1f}
μs → 分界带宽 {kv / t_tok / 1e9:.1f} GB/s")
    print(f"   {n} token：KV {n * kv / 1e9:.2f} GB，重新 prefill 约 {n * t_tok *
1e3:.0f} ms；读回来：" +
          "，".join(f"{t[0].split()[0]} {n * kv / t[1] * 1e3:.0f} ms" for t in
TIERS))

# 一台机器的 CPU 内存能存多少段前缀
dram = 2e12
print(f"2 TB 的 CPU 内存能存 Llama-3-8B 的 {dram / 131072 / 1e6:.0f}M 个 token，
DeepSeek-V3 的 {dram / (61 * 576 * 2) / 1e6:.0f}M 个 token")
```

输出：

```
Llama-3-8B：每 token KV 128.0 KiB，重算每 token 15.2 μs → 分界带宽 8.6 GB/s
   8000 token：KV 1.05 GB，重新 prefill 约 121 ms；读回来：本卡显存 0 ms，同机其他卡 2
ms，CPU 21 ms，别的机器 21 ms，本机 105 ms，远端分布式存储 350 ms
Llama-3.1-70B：每 token KV 320.0 KiB，重算每 token 140.5 μs → 分界带宽 2.3 GB/s
   8000 token：KV 2.62 GB，重新 prefill 约 1124 ms；读回来：本卡显存 1 ms，同机其他卡 6
ms，CPU 52 ms，别的机器 52 ms，本机 262 ms，远端分布式存储 874 ms
DeepSeek-V3（MLA，FP8 计算）：每 token KV 68.6 KiB，重算每 token 37.4 μs → 分界带宽 1.9
GB/s
   8000 token：KV 0.56 GB，重新 prefill 约 299 ms；读回来：本卡显存 0 ms，同机其他卡 1
ms，CPU 11 ms，别的机器 11 ms，本机 56 ms，远端分布式存储 187 ms
2 TB 的 CPU 内存能存 Llama-3-8B 的 15M 个 token，DeepSeek-V3 的 28M 个 token
```

每个模型两行：第一行是分界带宽，第二行是 8000 token 前缀的读回时间和重算时间。8B 时 SSD（105 ms）勉强比重算（121 ms）快，远端存储（350 ms）不如重算；70B 和 DeepSeek-V3 时，所有层级都比重算快。最后一行是内存的容量。

第二段代码复现后半部分的数字：KV 产生速率和各级能保留多久，复用间隔对应的容量，保存再取回的回本次数，逐层预取的三种时间线，以及 SSD 的写入额度。

```python
# Qwen3-8B 的 8192 token 前缀（1.125 GiB），一张 A100（按峰值 50%：156 TFLOP/s）
V, layers = 8192 * 144 * 1024, 36
recompute = 0.856                                   # 在 A100 上重新 prefill 这 8192 个 token（第 112 讲）
r = V / recompute                                   # GPU 连续 prefill 时，每秒产生的 KV 字节数
print(f"KV 产生速率 r = {r/1e9:.2f} GB/s")

# 1. 存多久：容量 C 至少要 r × 复用间隔 T
for name, cap in [("HBM（扣掉权重）", 63.6e9), ("主机内存（每卡 256 GiB）", 256 * 2**30), ("本地 SSD", 3.84e12)]:
    print(f"{name}：能保留最近 {cap / r:,.0f} 秒产生的 KV")
for T in [10, 600]:
    print(f"复用间隔 {T} 秒：至少要 {r * T / 1e9:,.1f} GB")

# 2. 保存再取回，什么时候比重算快：T_写 + k·T_读 < k·T_重算
for bw in [25e9, 2e9, 1.25e9]:
    w = rd = V / bw
    k = next((k for k in range(1, 100) if w + k * rd < k * recompute), None)
    print(f"链路 {bw/1e9:5.2f} GB/s：读一次 {rd*1e3:6.1f} ms，" + (f"复用 {k} 次就回本" if k else "怎么复用都不如重算"))

# 3. 命中后只算 256 个新 token：先读后算、逐层预取、提前读入几层
t_new = (2 * 6.95e9 * 256 + 4 * 32 * 128 * 36 * (8448**2 - 8192**2) / 2) / 156e12
t_read = V / 25e9                                   # 从主机内存经 PCIe 读回
per_r, per_c = t_read / layers, t_new / layers
serial = t_read + t_new
layerwise = per_r * layers + per_c                  # 每层的计算都在等这一层读完，最后一层读完再算一层
buf = 25e9 * (t_read - t_new)                       # 读比算多出来的字节，要提前读进缓冲
pre = -(-buf // (V / layers))
print(f"计算 {t_new*1e3:.1f} ms；先读后算 {serial*1e3:.1f} ms；逐层预取 {layerwise*1e3:.1f} ms")
print(f"提前读入 {buf/1e6:.0f} MB（{pre:.0f} 层），总时间降到计算本身的 {t_new*1e3:.1f} ms")
print(f"新 token 至少 {256 * t_read / t_new:.0f} 个，计算就能完全盖住从主机内存的读取")

# 4. SSD 的寿命：1 DWPD（五年内每天写满一次）
budget = 3.84e12 / 86400
print(f"SSD 每秒平均只能写 {budget/1e6:.1f} MB，是 r 的 {budget/r*100:.1f}%")
```

输出：

```
KV 产生速率 r = 1.41 GB/s
HBM（扣掉权重）：能保留最近 45 秒产生的 KV
主机内存（每卡 256 GiB）：能保留最近 195 秒产生的 KV
本地 SSD：能保留最近 2,721 秒产生的 KV
复用间隔 10 秒：至少要 14.1 GB
复用间隔 600 秒：至少要 846.7 GB
链路 25.00 GB/s：读一次   48.3 ms，复用 1 次就回本
链路  2.00 GB/s：读一次  604.0 ms，复用 3 次就回本
链路  1.25 GB/s：读一次  966.4 ms，怎么复用都不如重算
计算 30.9 ms；先读后算 79.2 ms；逐层预取 49.2 ms
提前读入 436 MB（14 层），总时间降到计算本身的 30.9 ms
新 token 至少 401 个，计算就能完全盖住从主机内存的读取
SSD 每秒平均只能写 44.4 MB，是 r 的 3.1%
```

对照正文：r 约 1.41 GB/s；HBM、主机内存、SSD 分别能保留约 45 s、195 s、2721 s；10 秒间隔要 14.1 GB、10 分钟要约 847 GB；链路 2 GB/s 要复用 3 次才回本，1.25 GB/s 不如重算；先读后算 79.2 ms、逐层预取 49.2 ms、提前读入 14 层后 30.9 ms；SSD 每秒只能写 44.4 MB，是 r 的 3.1%。

容易混淆的地方

1. **分界带宽只是第一道判断**。它没算读的延迟、和其他传输的争抢、以及读回时显存要先腾出空间。真正决定是否读回，还要看当时的负载。

2. **CPU 内存的带宽受 PCIe 限制**。CPU 内存本身很快，但到 GPU 要经过 PCIe，一张卡大约 50 GB/s；多张卡同时读时，还可能共享 CPU 的内存带宽。

3. **把 KV 存下来不改变模型的输出**。读回来的 KV 和重算的完全一样（第 76 讲的前缀缓存同理），只要前缀逐 token 相同。

4. **缓存够大不等于命中率够高**。命中率随容量增长很快就饱和，很多 KV 根本不会再用。容量按 $C \geqslant r \times T$ 估计，再看命中率曲线。

5. **SSD 的瓶颈常常不是带宽，而是写入寿命**。按 1 DWPD 算，一块盘每秒只能写几十 MB，写入前要做准入判断。

回顾

在集群里，KV Cache 要在 prefill 和 decode 实例之间传递，也可以从显存卸载到 CPU 内存、SSD 和远端存储。一段算过的前缀，读回还是重算，看存储带宽是否高于分界带宽 = 每 token KV ÷ 每 token 重算时间：Llama-3-8B 约 8.6 GB/s，70B 约 2.3 GB/s，DeepSeek-V3 约 1.9 GB/s。8B 的 8000 token 前缀从 CPU 内存读回约 21 ms，重算约 121 ms。模型越大、KV 越小，越慢的存储也值得用。KV 流动时要逐层与计算重叠、合并成大的传输、走 GPUDirect RDMA，并维护一个“谁有哪段前缀”的目录。

缓存还要回答“存多久”：容量至少是 KV 产生速率 r × 复用间隔，r 就是分界带宽（Qwen3-8B 在 A100 上约 1.41 GB/s）；命中率随容量很快饱和。读回时逐层预取，再提前读入读比算多出来的那部分（8K 前缀约 14 层），读取就能被计算完全盖住。SSD 带宽够，但 1 DWPD 的寿命只允许写入约 3% 的新 KV。

有了分布在各处的 KV 缓存，新的问题来了：一个请求进来，该发给哪一台机器？发给有它前缀缓存的那台，能省掉 prefill；可如果大家的前缀都在同一台上，那台就会被挤爆。下一讲：**缓存感知路由与负载均衡**。

第114讲 路由与负载均衡

76 Prefix Cache　78 调度与抢占　82 推理压测实践　113 KV Cache 传输与分层存储

一个推理服务通常有很多个实例（每个实例是一份完整的模型，可能占一张卡，也可能占一组卡）。请求进来，先到一个**路由器**（也叫网关、负载均衡器），由它决定发给哪个实例。

传统的 Web 服务里，这件事很简单：哪个实例闲就发给哪个，或者干脆轮着发。可大模型推理有一个特殊之处：**每个实例里都有前缀缓存**（第 76 讲）。同一个会话的第二轮、同一个应用的系统提示，如果发到上次处理过它的实例，前缀就能命中，省掉一大段 prefill；发到别的实例，就得从头算。

于是路由面临一个两难：**为了命中缓存，应该把相关的请求集中到同一个实例；为了负载均衡，又应该把请求分散开**。这一讲用一个模拟比较几种策略，看怎么兼顾两头。

一个模拟

我们构造一个典型的负载：

- 12 个应用，每个有一段 2000 token 的系统提示。应用的热度很不均匀，第 1 个最热门（热度按 1、1/2、1/3……递减）。
- 300 个用户会话，每个属于某个应用，聊 1 到 8 轮。每一轮的输入是“系统提示 + 这个会话之前的全部历史 + 新问题”，每轮新增 300 token。
- 一共 1288 个请求，各会话交错到达。

4 个实例，每个实例的前缀缓存能放 6 万个 token（显存有限，按 LRU 淘汰）。前缀只能从头连续命中：系统提示命中了，才可能接着命中第 1 轮的历史，依此类推。

每个实例的**工作量**记为：要 prefill 的 token 数，加上 decode 的部分（200 个输出 token，粗略地按每个相当于 20 个 prefill token 计）。我们看两个指标：前缀缓存的**命中率**（命中的 token 占所有输入 token 的比例），和**负载不均衡**（最忙实例的工作量是平均的几倍）。

五种策略

轮询：依次发给 4 个实例。负载最均匀（1.01 倍），命中率 71.6%。命中主要来自系统提示：12 段系统提示很快就在每个实例里都有了一份；可同一个会话的下一轮多半被发到了别的实例，历史部分命中不了。

最空闲：发给当前工作量最少的实例。负载完美（1.00 倍），命中率和轮询差不多（71.4%），原因一样。

按会话固定：同一个会话的请求总发给同一个实例（按会话编号取模）。会话历史基本都能命中，命中率升到 84.8%；但不同会话的长度不同，负载有些不均（1.12 倍）。缓存容量有限，热门会话多的实例上，旧的历史会被挤掉。

按应用固定：同一个应用的请求总发给同一个实例。系统提示和会话历史都在一处，命中率 92.1%；可应用的热度很不均匀，最热门的应用所在的实例被压垮了：工作量是平均的 1.86 倍。整个集群的速度被这一个实例拖住。

缓存感知（限负载）：对每个请求，先找出负载不超过平均 1.2 倍的实例，再在其中选**前缀命中最多**的那个（一样多时选更空闲的）。命中率 96.8%，负载 1.12 倍。热门应用的请求先集中在一个实例上，它一旦过载，新的请求就溢出到别的实例，在那里也建立起缓存。

下面的演示可以切换策略，调实例数和缓存容量。柱子是各实例的总工作量，下方列出了五种策略的对比。



图 114-1　路由策略的模拟。轮询和最空闲负载最均匀，但同一会话的请求散落各处，历史命中不了；按应用固定命中率高，但热门应用压垮了一个实例；缓存感知的路由在负载上限内选命中最多的实例，两头兼顾。

要点

大模型推理的路由要兼顾前缀缓存的命中率和负载均衡：把相关请求集中到同一实例能命中缓存，但会造成热点；均匀分散能平衡负载，但缓存命中不了。缓存感知路由在负载不超过某个上限的实例中，选前缀命中最多的那个，模拟里命中率 96.8%、最忙实例只比平均多 12%。

路由器怎么知道谁有缓存

缓存感知路由要知道每个实例里缓存了哪些前缀。常见的做法有两种：

- **路由器自己记账**：路由器为每个实例维护一棵近似的前缀树（第 76 讲的基数树），记录发过去的请求的前缀。它不精确（实例那边可能已经淘汰了），但足够好，而且不需要和实例通信。SGLang 的路由器就是这样做的。
- **实例上报**：实例把自己缓存里的块哈希（第 76 讲）定期或实时地报告给一个中心的索引，路由器查询这个索引。更准确，但要多一些通信。vLLM 生态的一些路由组件（如 llm-d）采用这种方式。

无论哪种，匹配的单位都是前缀：从头开始比，能连续匹配多长。

负载怎么衡量

“负载”也不止一种量法：排队的请求数、正在运行的请求数、KV Cache 的使用率、预计还要算多少 token……前缀命中率高的请求，prefill 的工作量小，同样一个请求，发到不同实例，带来的负载也不同。好的路由器会把“发到这里要算多少”算进去：一个命中了 90% 前缀的实例，即使队列稍长，可能也比一个完全不命中的空闲实例更快。

还要注意**突发**。热门应用在某个时刻突然涌入大量请求，缓存感知路由会先把它们都送到有缓存的那个实例。负载上限（这里是平均的 1.2 倍）就是为了防止这种情况：超过上限就溢出。

有缓存的实例很忙时

上面的模拟用“负载不超过平均 1.2 倍”这条规则防止热点。把这个问题算细一点：有缓存的实例 A 正在排队，空闲的实例 B 没有缓存，请求发给谁?

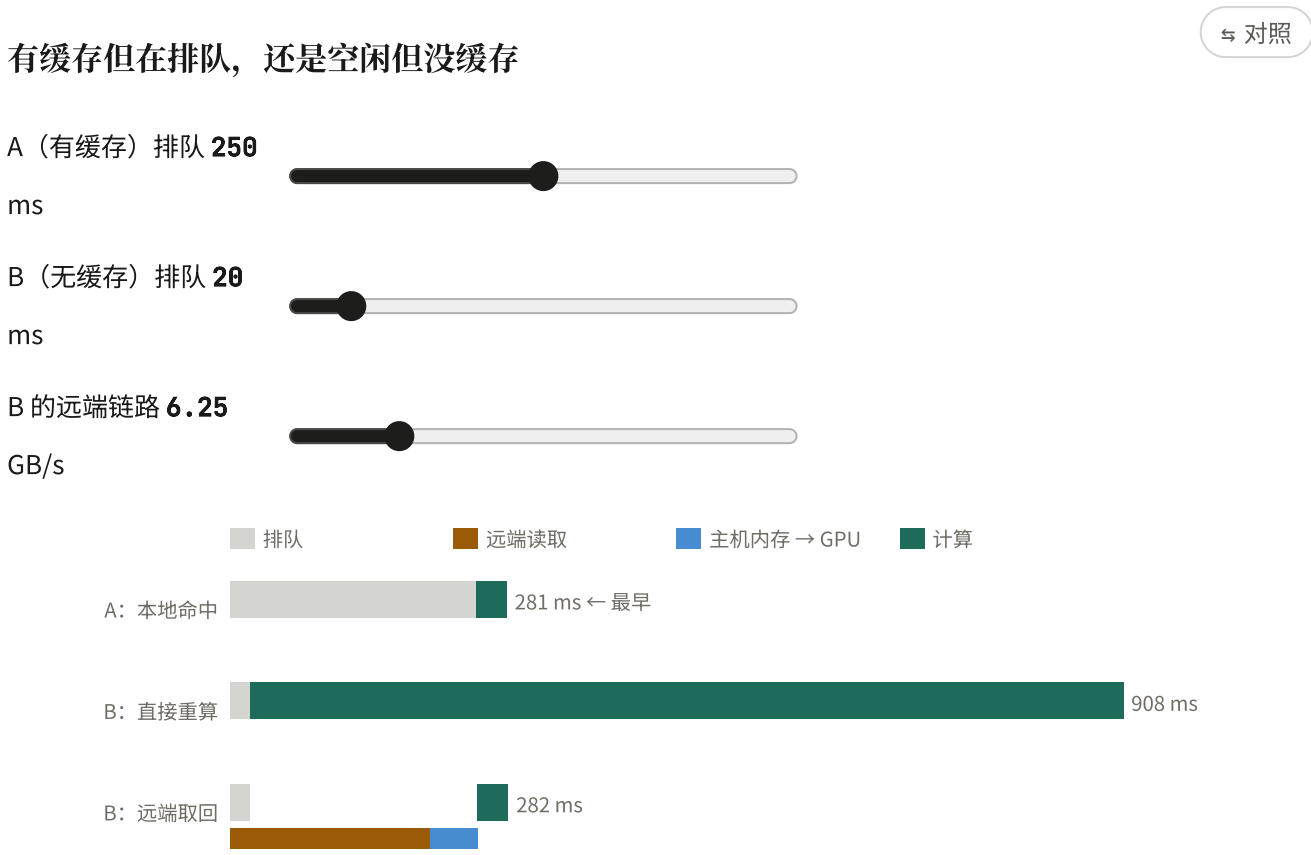
只看命中率或只看负载都不够，要比的是**首 token 什么时候出来**。设 GPU 最早能开始算的时刻是 Q（排队），状态准备好的时刻是 R（取回 KV 的时间），状态好了之后还要算 C。如果取回可以和排队同时进行，

$$\text{首 token} = \max(Q, R) + C$$

依赖关系很重要：Q = 80 ms、取回 60 ms、计算 10 ms，取回和排队同时进行只要 **90 ms**；等 GPU 空了才开始取回就要 **150 ms**。同样的计算、同样的字节，差了 60 ms。所以好的系统在请求还在排队时就开始取回它要的 KV。

算一个具体的例子：Qwen3-8B 的请求带 8192 token 的前缀和 256 个新 token，两个 A100 实例。A 的显存里有这份前缀，但要排队 250 ms；B 只要等 20 ms，但没有缓存。全部重算 8448 个 token 约 888 ms，命中后只算 256 个约 30.9 ms。B 还可以从远端存储取回前缀：查找 10 ms，经网络读到主机内存，再经 PCIe（25 GB/s）搬到 GPU，两段串行。

路径	首 token
A：本地命中	250 + 30.9 ≈ **281 ms**
B：直接重算	20 + 888 ≈ **908 ms**
B：经 50 GbE（6.25 GB/s）远端取回	约 **282 ms**
B：经 200 GbE（25 GB/s）远端取回	约 **138 ms**



首 token = max(排队 Q，状态就绪 R) + 剩余计算 C。B 取回：R = 10 + 193.3 + 48.3 = 251.6 ms，与 B 的排队同时进行。B 取回要比 A 先出首 token，远端链路至少要 1.125 GiB ÷ (A 的排队 - 10 - 48.3) ms ≈ **6.30 GB/s**。

只看“谁有缓存”会把请求送去排长队；只看“谁空闲”会白白重算 888 ms。路由要比较的是预计的完成时间。

图 114-2　同一个请求的三条路径。A 有缓存但排队 250 ms；B 空闲但要重算 888 ms，或者从远端取回再算。取回和 B 的排队同时进行；远端链路够快时，B 取回最早出首 token。

重算几乎总是最慢的。B 要靠取回赶上 A：扣掉命中后的 30.9 ms 计算，状态必须在 250 ms 内就绪；再扣掉查找的 10 ms 和 PCIe 的 48.3 ms，远端读取只剩约 192 ms，所以远端链路至少要 6.30 GB/s。50 GbE 的 6.25 GB/s 略低于此，200 GbE 远高于此。这个阈值随 A 的排队时间变化，调一调演示里的滑块就能看到。

持续到达时还要看总流量。200 GbE 时单个请求取回最快，但每秒 16 次取回要约 19.3 GB/s，已经用掉这条链路 77% 的能力，突发时取回本身要排队。

路由器以为的缓存不一定还在。路由器一般根据实例上报的缓存事件来判断谁有缓存，事件可能延迟，缓存可能已经被淘汰。把 A 的排队改成 80 ms，设缓存只有概率 p 真的还在：在的话 111 ms，不在就只能重算，968 ms。p = 0.9 时平均 196.5 ms，看起来很好；但 10% 的请求要 968 ms，p99 就是 968 ms。要让 p99 也好，命中的把握要到 99% 以上。均值和尾延迟，要分开看。

优化谁的时间。把一个请求挪到空闲实例，它自己快了，可能让其他请求更晚完成。决定路由之前，要先确定目标是这个请求的响应时间，还是所有请求一起的完成时间。

到了大模型里

PD 分离时的路由。PD 分离的系统里，路由器要分两步决定：prefill 发给哪个 prefill 实例（看前缀缓存），decode 交给哪个 decode 实例（看 KV 空间和负载）。如果 KV 存在共享的缓存池里（上一讲的 Mooncake 等），prefill 实例可以从池子里读回别处算过的前缀，路由的约束就松了很多。

多卡、多机房。大规模服务还要考虑地理位置、模型版本、LoRA 适配器（第 50 讲）在哪些实例上加载，等等。缓存感知只是路由要考虑的因素之一。

会话粘性的副作用。按会话固定的路由简单有效，很多服务在用。它的问题是：实例故障或扩缩容时，大量会话的缓存会失效；某些超长会话会让一个实例特别忙。所以通常要配合负载上限一起用。

用代码复现

下面的代码生成上面的负载，实现五种路由策略和每个实例的 LRU 前缀缓存，统计命中率和负载不均衡的程度。

```
import random
from collections import OrderedDict
random.seed(0)

# 负载：12 个应用，各有一段 2000 token 的系统提示（热度不均）；300 个用户会话，每个会话属于一个应用，聊 1-8 轮，
# 每轮新增 300 token（问题 + 上一轮的回答）。请求的前缀 = 系统提示 + 这个会话之前的全部历史
APPS, SESS, SYS, TURN, OUT = 12, 300, 2000, 300, 200
weights = [1 / (a + 1) for a in range(APPS)]                          # 第 1 个应用最热门
sessions = [(random.choices(range(APPS), weights)[0], random.randint(1, 8)) for _ in range(SESS)]
reqs = [(s, t) for s, (_, turns) in enumerate(sessions) for t in range(turns)]
random.shuffle(reqs); reqs.sort(key=lambda r: r[1])                   # 各会话交错到达，同一会话的轮次按顺序

def blocks(s, t):
    """请求的前缀拆成若干“块”（键，token 数）：系统提示一块，之前每一轮一块。前缀缓存只能从头连续命中"""
    app = sessions[s][0]
    return [(("sys", app), SYS)] + [(("turn", s, i), TURN) for i in range(t)]

def simulate(policy, n_inst=4, cap=60_000):
    caches = [OrderedDict() for _ in range(n_inst)]                   # 每个实例一个 LRU 前缀缓存（容量按 token 计）
    load = [0.0] * n_inst; hit_tok = all_tok = 0
    for k, (s, t) in enumerate(reqs):
        bl = blocks(s, t)
        def hits(i):                                                  # 在实例 i 上能从头连续命中多少 token
            h = 0
            for key, n in bl:
                if key in caches[i]: h += n
                else: break
            return h
        avg = sum(load) / n_inst
        if policy == "轮询": i = k % n_inst
        elif policy == "最空闲": i = min(range(n_inst), key=lambda j: load[j])
        elif policy == "按会话固定": i = s % n_inst
        elif policy == "按应用固定": i = sessions[s][0] % n_inst
        else:                                                         # 缓存感知：命中最多的实例，但负载不能超过平均的 1.2 倍
            ok = [j for j in range(n_inst) if load[j] <= 1.2 * avg + 1] or list(range(n_inst))
            i = max(ok, key=lambda j: (hits(j), -load[j]))
        h = hits(i); total = sum(n for _, n in bl)
```

```
        hit_tok += h; all_tok += total
        load[i] += (total - h) + OUT * 20                          # 工作量：要
prefill 的 token + decode（按每个输出 token 约等于 20 个 prefill token 估）
        for key, n in bl + [(("turn", s, t), TURN)]:               # 用过的块放到
LRU 的最新端；这一轮的结果也存进去
            caches[i].pop(key, None); caches[i][key] = n
        while sum(caches[i].values()) > cap: caches[i].popitem(last=False)
    return hit_tok / all_tok, max(load) / (sum(load) / n_inst)

print(f"{len(reqs)} 个请求，4 个实例，每个实例的前缀缓存 6 万 token")
for p in ["轮询", "最空闲", "按会话固定", "按应用固定", "缓存感知（限负载）"]:
    hr, imb = simulate(p)
    print(f"  {p:9s}：缓存命中率 {hr:5.1%}，最忙实例的工作量是平均的 {imb:.2f} 倍")
```

输出：

```
1288 个请求，4 个实例，每个实例的前缀缓存 6 万 token
  轮询         ：缓存命中率 71.6%，最忙实例的工作量是平均的 1.01 倍
  最空闲        ：缓存命中率 71.4%，最忙实例的工作量是平均的 1.00 倍
  按会话固定      ：缓存命中率 84.8%，最忙实例的工作量是平均的 1.12 倍
  按应用固定      ：缓存命中率 92.1%，最忙实例的工作量是平均的 1.86 倍
  缓存感知（限负载）：缓存命中率 96.8%，最忙实例的工作量是平均的 1.12 倍
```

五行对应五种策略，和演示一致：缓存感知（限负载）的命中率最高（96.8%），负载只比轮询差一点（1.12 倍）；按应用固定虽然命中率也高，最忙的实例却是平均的 1.86 倍。

第二段代码复现“有缓存的实例很忙时”的数字：四条路径的首 token、B 取回追平 A 需要的带宽、取回与排队串行和并行的差别，以及缓存只以概率 p 存在时的均值和慢路径。

```python
# 首 token 时间 = max(GPU 排队 Q, 状态就绪 R) + 剩余计算 C（取回可以和排队同时进行）
# Qwen3-8B 的 8192 token 前缀 + 256 个新 token, A100 按 156 TFLOP/s
V = 8192 * 144 * 1024                                   # 前缀的 KV: 1.125 GiB
def prefill(new, ctx):                                  # 线性层 + 因果注意力（第 112 讲）的计算时间
    return (2 * 6.95e9 * new + 4 * 32 * 128 * 36 * (ctx**2 - (ctx - new)**2) / 2) / 156e12
full, hit = prefill(8448, 8448), prefill(256, 8448)     # 全部重算 / 命中后只算 256 个新 token
lookup, pcie = 0.010, V / 25e9                          # 远端查找 10 ms；主机内存到 GPU 经 PCIe 25 GB/s

def first(Q, R, C): return max(Q, R) + C
print(f"A: 本地命中, 排队 250 ms     首 token {first(0.250, 0, hit)*1e3:5.0f} ms")
print(f"B: 空闲 20 ms, 直接重算     首 token {first(0.020, 0, full)*1e3:5.0f} ms")
for name, bw in [("50 GbE (6.25 GB/s) ", 6.25e9), ("200 GbE (25 GB/s) ", 25e9)]:
    R = lookup + V / bw + pcie                          # 先经网络到主机内存，再经 PCIe 到 GPU，两段串行
    print(f"B: 从远端取回, {name}  首 token {first(0.020, R, hit)*1e3:5.0f} ms")
budget = 0.250 - lookup - pcie                          # B 要追平 A，远端读取最多能用的时间
print(f"B 取回要追平 A，远端链路至少 {V / budget / 1e9:.2f} GB/s")
print(f"等 GPU 空了才开始取回（串行）：{(0.080 + 0.060 + 0.010)*1e3:.0f} ms；和排队同时进行: {first(0.080, 0.060, 0.010)*1e3:.0f} ms")

# 路由器以为 A 有缓存，但只有概率 p 真的还在（事件延迟、已被淘汰）：A 排队 80 ms
ok, miss = 0.080 + hit, 0.080 + full
for p in [0.5, 0.9, 0.99]:
    print(f"p = {p:.2f}: 平均 {(p*ok + (1-p)*miss)*1e3:5.1f} ms；但 {1-p:.0%} 的请求要 {miss*1e3:.0f} ms")
```

输出：

```
A: 本地命中, 排队 250 ms     首 token   281 ms
B: 空闲 20 ms, 直接重算     首 token   908 ms
B: 从远端取回, 50 GbE (6.25 GB/s)   首 token   282 ms
B: 从远端取回, 200 GbE (25 GB/s)   首 token   138 ms
B 取回要追平 A，远端链路至少 6.30 GB/s
等 GPU 空了才开始取回（串行）：150 ms；和排队同时进行: 90 ms
p = 0.50: 平均 539.3 ms；但 50% 的请求要 968 ms
p = 0.90: 平均 196.5 ms；但 10% 的请求要 968 ms
p = 0.99: 平均 119.4 ms；但 1% 的请求要 968 ms
```

对照正文：281、908、282、138 ms；追平需要 6.30 GB/s；串行 150 ms、并行 90 ms；p = 0.9 时平均 196.5 ms，但 10% 的请求要 968 ms。

容易混淆的地方

1. **命中率不是唯一目标**。按应用固定的命中率很高，但热点让整体延迟变差。路由要优化的是整体的吞吐和延迟（第 82 讲的 goodput），命中率只是手段。

2. **轮询不是“没有缓存”**。系统提示这类所有请求共享的前缀，在每个实例里都会被缓存；轮询损失的是会话历史这类“只属于少数请求”的前缀。

3. **缓存容量会改变结论**。缓存很大时，几乎什么都能存下，策略之间的差距变小；缓存很小时，集中路由的价值更大。在演示里调一调缓存容量就能看到。

4. **命中缓存不等于更快**。有缓存的实例可能在排长队。要比较的是首 token = max(排队, 状态就绪) + 剩余计算；空闲实例从够快的远端取回，可能比在有缓存的实例上排队更早。

5. **平均命中率高，尾延迟不一定好**。路由器以为的缓存有一定概率已经不在，这部分请求要整段重算，决定了 p99。

回顾

推理服务的路由器要决定每个请求发给哪个实例。把相关请求集中起来能命中前缀缓存，但会造成热点；均匀分散能平衡负载，但缓存命中不了。模拟的 4 个实例、1288 个请求里：轮询命中率 71.6%，负载 1.01 倍；按会话固定 84.8%、1.12 倍；按应用固定 92.1%，但最忙实例是平均的 1.86 倍；缓存感知路由在负载不超过平均 1.2 倍的实例里选命中最多的，命中率 96.8%、负载 1.12 倍。路由器可以自己维护近似的前缀树，也可以让实例上报缓存的块哈希；负载的衡量要考虑命中后实际要算多少。

算得更细时，路由比较的是预计的首 token 时间 max(Q, R) + C。8K 前缀的例子里，有缓存但排队 250 ms 的实例 281 ms 出首 token，空闲实例重算要 908 ms，从 200 GbE 的远端取回只要 138 ms；远端链路要追平本地命中，至少需要 6.30 GB/s。缓存预测有失误时，均值可能很好，p99 却由重算的慢路径决定。

从张量并行、专家并行、PD 分离到 KV 的流动和路由，这一篇反复出现同一个主题：卡与卡之间的通信越快，能做的事情越多。超节点把几十上百张卡放进同一个高带宽域，它对哪类推理负载的收益最大？下一讲：**超节点上的推理**。

第115讲 超节点上的推理

重点

65 超节点　102 通信代价模型　109 并行策略与拓扑映射　110 推理中的张量并行　111 MoE 推理与大规模专家并行　113 KV Cache 传输与分层存储

第 109 讲给出了一条原则：通信越频繁的并行，放在越快的互联上。可快速互联的范围是有限的：一台 HGX H100 服务器只有 8 张卡，第 9 张卡就要走慢了九倍的网卡。第 65 讲介绍了**超节点**（supernode）：把几十甚至几百张卡放进同一个高带宽域（scale-up 域），任意两张卡之间都有接近机内的带宽。

于是很自然地会想：域变大了，推理是不是就全面变快了？答案没有这么简单。这一讲用同一套估算方法，逐项算三类负载：放得进 8 卡的稠密模型、放不进 8 卡的稠密模型、大规模 MoE。你会看到，有的负载在超节点上反而更慢；有的负载变快，但原因不是“通信变快”，而是“显存变多”。最后再看超节点内部的拓扑怎么选、多出来的显存能不能借给别人，以及它的代价。

先看超节点里是怎么连的

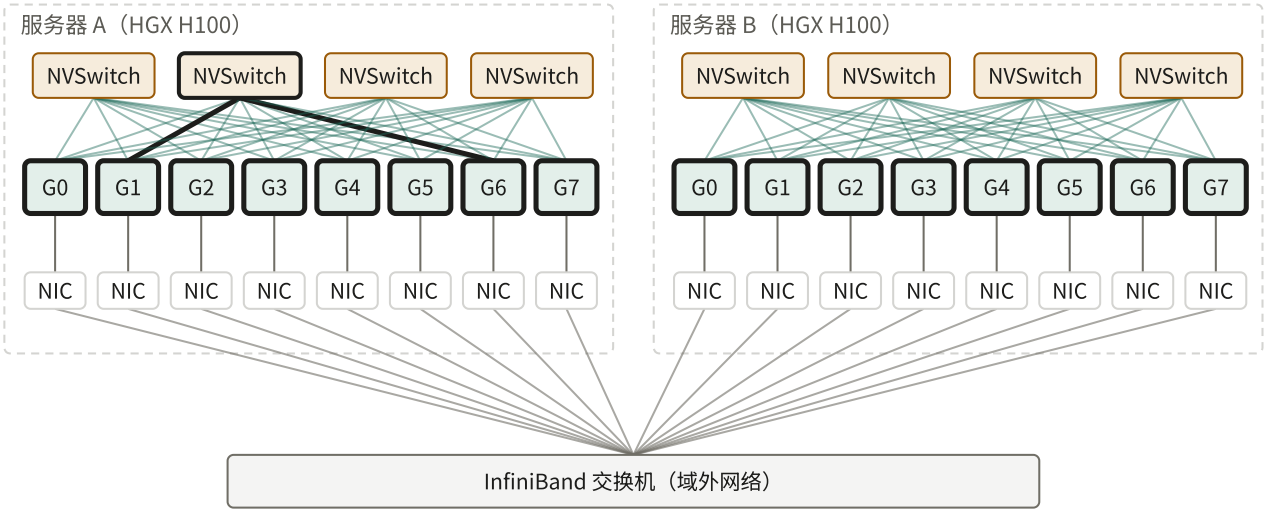
动手算之前，先把要比较的几种连法看清楚。下面的图画了四种公开的系统，可以点部件看说明，切换“路径”看一次传输经过哪些芯片。

超节点里是怎么连的

⇆ 对照

系统： 两台 8 卡服务器 | GB200 NVL72 | CloudMatrix384 | TPU v4 立方体

路径： 同一台机器：A.G1 → A.G6 | 跨机器：A.G3 → B.G5



每台服务器是一个 8 卡的 NVLink 域；两台之间只能走网卡

H100 SXM：80 GB HBM3，显存带宽 3.35 TB/s，BF16 稠密算力约 989 TFLOP/s。18 条第四代 NVLink，每条每方向 25 GB/s，合计每方向 450 GB/s。

路径（加粗的线）：A.G1 → NVSwitch → A.G6：两条 NVLink 加一次交换，每卡单向 450 GB/s，实测每次传输的固定延迟约 0.8 µs。 点图中任意部件看说明。

图 115-1　四种系统的连接方式（示意，数字取公开规格，以官方资料和论文为准）。两台 8 卡服务器：机内走 NVSwitch，机间只能走网卡；GB200 NVL72：72 张卡经机柜背板接到 18 颗 NVLink 交换芯片；CloudMatrix384：384 个 NPU 经两级 UB 交换互连；TPU v4：芯片直连邻居组成环面，立方体之间经光路交换机。

先看**两台 8 卡服务器**。每台机器里，8 张 H100 的 18 条 NVLink 接到 4 颗 NVSwitch 上，任意两张卡之间都是满速的 450 GB/s（每方向）。但服务器 A 的卡要找服务器 B 的卡，数据得先经 PCIe 到网卡，再经 InfiniBand 交换机到对方网卡：带宽降到 50 GB/s，单向延迟从约 0.8 µs 涨到约 3.8 µs。这两个延迟数字取自 MSCCL++ 论文在“每节点 8 张 H100 + 每卡一张 400 Gbit/s ConnectX-7”平台上的实测，这一讲都用它们。

GB200 NVL72 把 18 个计算托盘（每个 2 颗 Grace CPU + 4 张 Blackwell GPU）和 9 个交换托盘（每个 2 颗 NVLink Switch 芯片）装进一个机柜，用背板铜缆全部连起来。每张 GPU 的 18 条 NVLink 分别接到 18 颗交换芯片上，所以 72 张卡中任意两张都只隔一次交换。8 卡服务器里“同一台机器”才有的待遇，扩大到了整个机柜。

CloudMatrix384 走的是同一个方向，但规模更大，所以交换分两级：48 个节点，每个节点 8 个昇腾 910C 和 4 颗鲲鹏 CPU，先接到节点里 7 颗一级 UB 交换芯片；一级交换再用光模块接到 7 个二级子平面（每个 16 颗交换芯片）。全系统 448 颗交换芯片，换来的是 384 个 NPU 在同一张网上。

TPU v4 是另一条路：不用交换芯片，每颗芯片用 6 条链路直接连前后左右上下的邻居，64 颗组成一个 4 × 4 × 4 的立方体，每一维首尾相连，叫**三维环面**（torus）。立方体之间再用光路交换机拼起来。这条路线便宜，但数据可能要穿过好几颗中间芯片，后面会算它对推理的影响。

还要分清两个概念：**推理实例**（一个模型副本，用几张卡）和**高带宽域**（哪些卡连得紧）。一个 72 卡的域可以放一个 72 卡的实例，也可以放 9 个 8 卡实例。超节点只是给了“需要时组成更大实例”的可能，用不用、怎么用，要看负载。

估算一步的方法

这一讲全部用一套估算，前几讲都用过：decode 一步的时间 ≈ 每张卡要从显存读的字节 ÷ 显存带宽（第 70 讲），加上卡与卡之间的通信。通信按第 102 讲的 α-β 模型，但这里要把**轮数**算清楚：Ring All-Reduce 要走 2(p − 1) 轮，每轮付一次固定延迟 α，再传 m/p 字节。所以

$$T_{AR} = 2(p-1) \times (\alpha + m / (p \cdot \beta))$$

第 102 讲用的 α 是示意的量级；这一讲换成实测值：NVLink 每轮 α = 0.822 μs。跨服务器的 All-Reduce 牵涉通信库的分层算法，不好按轮数推，直接用公开的 nccl-tests 记录：两台 HGX H100、16 个进程、16 KiB 消息的 All-Reduce 实测 **32.74 μs**。

GPU 一律按 H100 算，只改卡的连法。这样比出来的差别，完全来自互联。

场景一：放得进 8 卡的稠密模型

先看一个已经能放进一台服务器的模型：Qwen3-32B（64 层，d = 5120，64 个查询头、8 个 KV 头，头维度 128），给一个用户续写，他的会话已经有 128K（131,072 个）token 的上下文。

一张卡放不下。BF16 权重约 65.5 GB，其中每步要读的部分（除去只查表的嵌入层）是 **64.0 GB**；KV 每个 token 是 2 × 64 层 × 8 头 × 128 × 2 字节 = **256 KiB**，128K 个 token 就是 32 GiB，约 34.4 GB。加起来远超 80 GB，至少要 TP2。

TP8，一台服务器内。权重按 8 份切，每卡 8.0 GB；KV 按头切，8 个 KV 头正好每卡一个，每卡 4.29 GB。每卡一步读 12.3 GB，÷ 3.35 TB/s = **3.67 ms**。通信：每层 2 次 All-Reduce，64 层共 128 次；每次的消息只有一个 token 的隐藏状态，5120 × 2 = 10 KiB。代入公式，14 轮 × 0.822 μs ≈ 11.55 μs，128 次共 **1.48 ms**。一步 **5.15 ms**。

现在你多了一台服务器。直觉是：卡翻倍，TP16，读显存的时间减半，应该更快。

TP16，跨两台服务器。权重每卡降到 4.0 GB。但 KV 只有 8 个头，16 张卡分不开，只能每个头在两张卡上各存一份，每卡读的 KV 还是 4.29 GB。读显存只从 3.67 ms 降到 **2.48 ms**。通信则是 128 次跨机的 All-Reduce，每次 32.74 μs，共 **4.19 ms**。一步 **6.67 ms**，比 TP8 **慢了 1.5 ms**。

TP16，在一个 16 卡的 NVLink 域里。这正是超节点给的条件：16 张卡都在 NVLink 交换网络里（DGX H100 的 NVLink Switch System 最多能连 256 张 H100）。All-Reduce 换成域内的 Ring：30 轮 × 0.822 μs ≈ 24.7 μs，128 次共 **3.16 ms**。一步 **5.64 ms**，仍然比单台服务器的 TP8 慢。

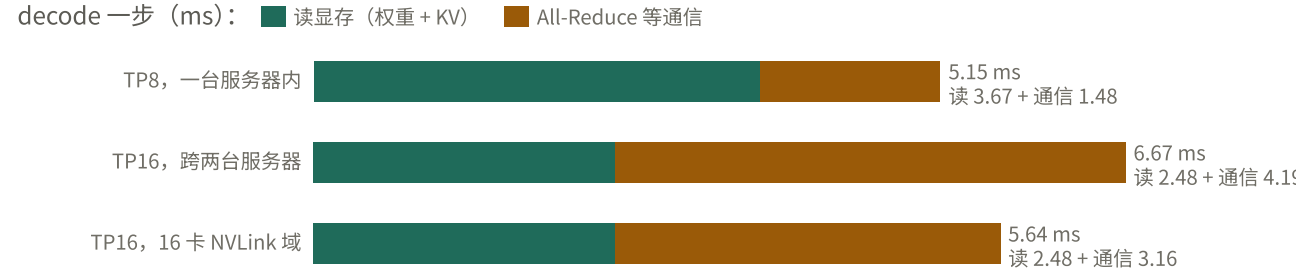
把 TP 组放大到 16 张卡

⇆ 对照

负载： Qwen3-32B，一个 128K 会话 | Llama-3.1-405B，32 个 8K 会话

NVLink 每轮 **0.822** µs

跨机 All-Reduce **32.74** µs



每卡：TP8 读权重 8.00 GB + KV 4.29 GB；TP16 读权重 4.00 GB + KV 4.29 GB（只有 8 个 KV 头，KV 不再减少）。一次 All-Reduce：TP8 环 14 轮 11.55 µs，16 卡域内环 30 轮 24.7 µs，跨机 32.74 µs；每步 128 次。

TP16 要追平 TP8，每次 All-Reduce 必须 ≤ (5.15 - 2.48) ms ÷ 128 = **20.9 µs**。现在：域内 仍然更慢，跨机 仍然更慢。

把 NVLink 每轮延迟往小调，相当于用更好的算法或在交换芯片里做归约（轮数变少）；这比把带宽翻倍有用得多。

图 115-2　稠密模型把 TP 组从 8 卡扩到 16 卡。Qwen3-32B：读显存只少了 1.2 ms（KV 头不够分），All-Reduce 的轮数却翻了一倍，域内的 TP16 也比 TP8 慢；拖动两个滑块找它追平 TP8 的条件。Llama-3.1-405B：从 TP8 × PP2 换成 TP16，一步少了约三成。

为什么超节点也救不了它？拆开看有两个原因。

第一，**KV 头的数量限制了切分**。TP 能把权重切到 1/16，KV 却只能切到 1/8。在 128K 这样的长上下文下，KV 占了每卡读取量的三分之一以上，这一部分不随卡数下降。

第二，**小消息的通信，耗时几乎全是轮数 × 延迟**。一次 All-Reduce 的 11.55 µs 里，真正传数据的带宽项只有约 40 ns。所以把 NVLink 带宽翻倍，每步只省 2.5 µs；把每轮延迟减半，每步能省 0.74 ms。卡数从 8 到 16，Ring 的轮数从 14 变成 30，延迟项跟着翻倍。超节点给了更多的卡、更大的带宽，但这类负载缺的不是带宽。

让结论翻转的条件。TP16 要追平 TP8，128 次 All-Reduce 加起来不能超过 5.15 − 2.48 = 2.67 ms，也就是每次 ⩽ **20.9 µs**。域内 Ring 的 24.7 µs 和跨机实测的 32.74 µs 都不满足。能满足的办法是减少轮数：例如一步到位的算法（第 102 讲），或 NVSwitch 在交换芯片里直接做归约（NVIDIA 叫 NVLink SHARP），轮数不再随卡数增长。vLLM、SGLang 为张量并行专门写小消息的 All-Reduce，也是这个原因。

> **要点**
> 放得进 8 卡的稠密模型，把 TP 组再扩大，收益很小甚至为负：KV 头数限制了 KV 的切分，小消息 All-Reduce 的耗时由轮数 × 延迟决定，带宽再高也帮不上。对这类负载，大的高带宽域更大的价值是灵活地放更多实例，而不是把一个请求切得更碎。

场景二：放不进 8 卡的稠密模型

换成 Llama-3.1-405B（126 层，d = 16384，8 个 KV 头）。BF16 权重约 812 GB，8 张 80 GB 的卡装不下，至少要 16 张。负载是 32 个会话一起 decode，每个会话 8K 上下文。

没有超节点：TP8 × PP2。TP 不能跨机（场景一刚算过跨机的代价），只能每台机器里 TP8，两台机器之间做流水并行（第 106 讲）。每卡放 1/16 的权重（50.5 GB）和 1/16 的 KV（8.46 GB）。问题在于：一个 token 要**先**走完第一台机器上的 63 层，**再**走第二台上的 63 层，两段的读显存时间是相加的。一步读显存 **35.18 ms**，加上通信共 **39.13 ms**。

有超节点：TP16。16 张卡在一个 NVLink 域里，直接 TP16：每卡的权重一样是 50.5 GB，KV 却翻倍到 16.91 GB（又是 8 个 KV 头不够 16 张卡分）。但一个 token 只走一段，读显存 **20.11 ms**；通信是 252 次 30 轮的 Ring，消息变大（32 × 16384 × 2 = 1 MiB），共 **7.32 ms**。一步 **27.43 ms**，比 TP8 × PP2 少了约三成。

这正是第 106 讲说过的：流水并行能让放不下的模型放下，也能靠多个微批次把吞吐补回来，但**缩短不了单个请求的延迟**；张量并行能。超节点让本来只能用 PP 的大模型可以改用 TP，TPOT 明显下降。注意收益比“读显存减半”小：KV 存了两份，All-Reduce 也多了 3.4 ms。

场景三：大规模 MoE

第 111 讲已经算过，MoE 的专家并行（EP）每层要做两次 All-to-All，跨机时卡在网卡上。超节点让 All-to-All 走 NVLink，通信自然快了。但这一节要讲一个更重要、也更容易被忽略的收益：**容量**。

每张卡放什么

以 DeepSeek-V3 为例（FP8 权重，61 层，前 3 层稠密，后 58 层各有 256 个路由专家，每个 token 选 8 个；注意力是 MLA，每个 token 每层缓存 576 个数）。按 DeepSeek 推理系统的做法部署：**注意力按数据并行**（每张卡服务自己的一批会话，注意力权重每卡一份），**路由专家按专家并行**均分到 S 张卡上。一个实例占满 S 张卡，S 就是 EP 组的大小。

每张卡上的显存分四块：

- **共享部分 17.1 GB**：注意力、前 3 层稠密 FFN、共享专家、路由器、嵌入表和输出层，每卡各存一份。
- **路由专家 654 GB ÷ S**：S 越大，每卡分到的专家越少。
- **2 GiB 工作区**。
- **剩下的全部给 KV**。一个 8K 上下文的会话，KV 是 8192 × 61 × 576 × 2 字节 = **0.576 GB**。剩下的显存能放几个会话，这张卡就能同时服务几个用户。

S = 8 时，每卡专家 81.7 GB，加上共享部分已经 **98.9 GB**，一台 8 卡服务器根本放不下（和第 111 讲的结论一致）。S = 16 时每卡权重 58.0 GB，剩下的显存只够 **34** 个会话；S = 64 时每卡权重 27.3 GB，能放 **87** 个；S = 256 时 **101** 个。

一步多长

每张卡一步要读：共享部分、本卡全部专家（batch 很大，本卡的专家几乎都会被选中）、本卡全部会话的 KV。接着做 58 层、每层两次的 All-to-All。

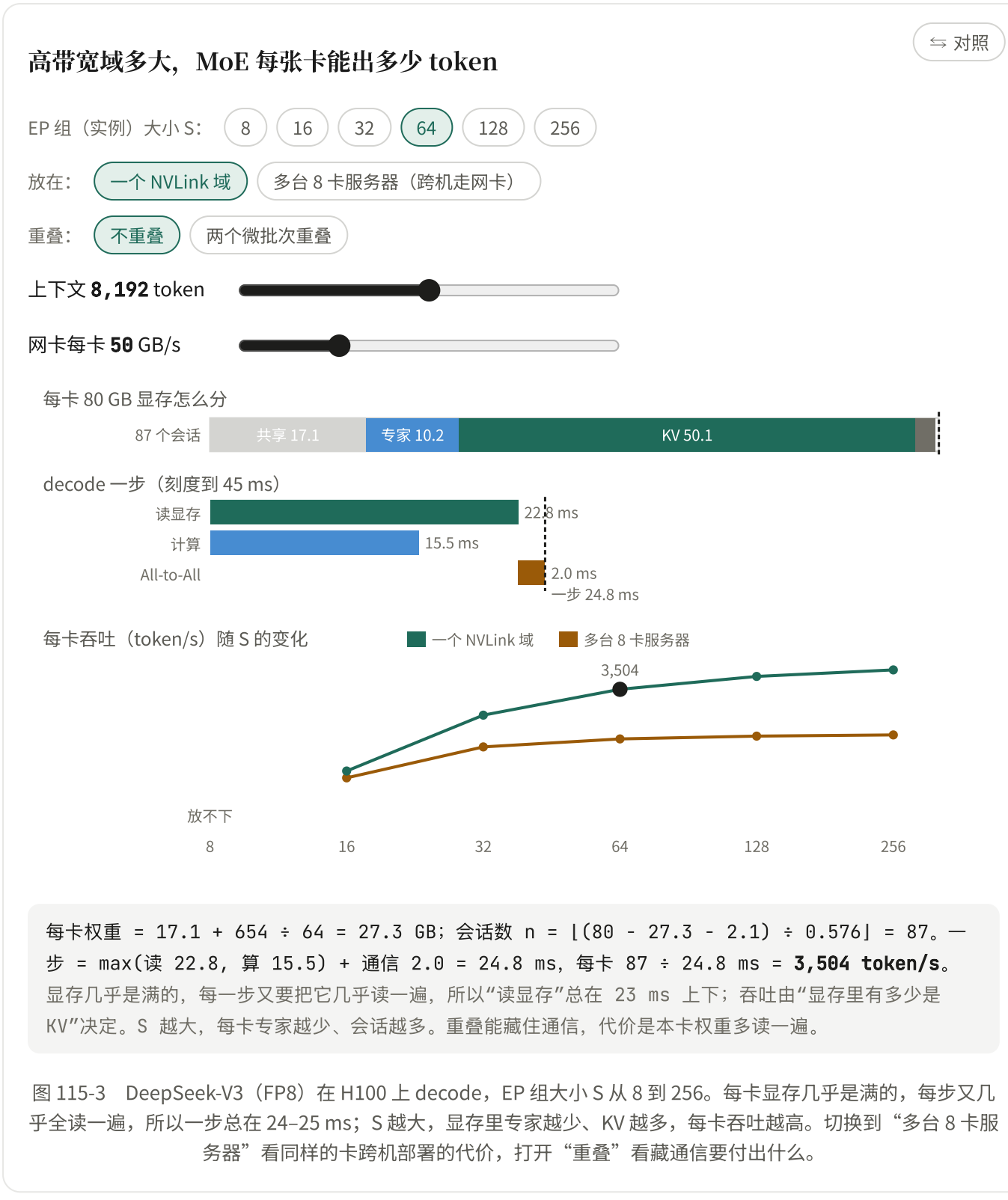


图 115-3　DeepSeek-V3（FP8）在 H100 上 decode，EP 组大小 S 从 8 到 256。每卡显存几乎是满的，每步又几乎全读一遍，所以一步总在 24–25 ms；S 越大，显存里专家越少、KV 越多，每卡吞吐越高。切换到“多台 8 卡服务器”看同样的卡跨机部署的代价，打开“重叠”看藏通信要付出什么。

代码算出的结果（8K 上下文，域内 All-to-All 走 450 GB/s 的 NVLink）：

EP 组大小 S	每卡权重	每卡会话	读显存	计算	通信	一步	每卡吞吐
8	98.9 GB	放不下					
16	58.0 GB	34	22.9 ms	6.0 ms	0.8 ms	23.7 ms	1436 tok/s
32	37.6 GB	70	23.0 ms	12.4 ms	1.6 ms	24.6 ms	2850 tok/s
64	27.3 GB	87	22.8 ms	15.5 ms	2.0 ms	24.8 ms	3504 tok/s
128	22.2 GB	96	22.9 ms	17.1 ms	2.2 ms	25.1 ms	3831 tok/s
256	19.7 GB	101	23.0 ms	18.0 ms	2.3 ms	25.3 ms	3996 tok/s

看“读显存”这一列：不管 S 多大，都在 23 ms 上下。原因很简单：每张卡的显存都被塞满了（专家少了，就多放 KV），而 decode 每一步要把显存里几乎所有的东西读一遍，80 GB ÷ 3.35 TB/s ≈ 24 ms。**一步的时间几乎是固定的，变的是这 80 GB 里有多少是 KV**。专家权重读一遍，服务的是所有会话；KV 读一遍，每份只服务一个会话。S 从 16 到 64，每卡专家从 40.9 GB 降到 10.2 GB，腾出来的显存放下了 2.6 倍的会话，每卡吞吐提高约 2.4 倍。

再往上，收益很快变小：S 从 64 到 256，每卡只多了 14 个会话。专家已经只占很小一块，剩下的是每卡都要存的 17.1 GB 共享部分，它不随 S 变化。

还有一点：**单个用户的速度和 S 无关**。一个会话独占时，关键路径是读一遍共享部分、每层读一个专家、读自己的 KV，约 5.96 ms 一步（168 tok/s），S = 16 还是 256 都一样。超节点让 MoE 服务更多的人，而不是让每个人更快。

同样 64 张卡，分在 8 台服务器上

现在回答最初的问题：超节点到底省了多少？把同样的 64 张卡换成 8 台 8 卡服务器。每张卡要和其余 63 张卡交换数据，只有 7 张在本机，**56/63 的字节要走 50 GB/s 的网卡**。每卡每层要发出 87 × 8 × 7168 × 3 字节 ≈ 15 MB（分发用 FP8，合并用 BF16），58 层的 All-to-All 从 2.0 ms 涨到 **15.9 ms**，一步 38.7 ms，每卡吞吐从 3504 降到 **2248 tok/s**。

看上去超节点快了 1.56 倍。但真实系统不会傻等通信：第 111 讲说过，DeepSeek 把每张卡的 batch 切成两个微批次，一个在做 All-to-All 时，另一个在计算。重叠后，跨机部署的一步变成 31.6 ms，每卡 **2754 tok/s**。

重叠不是免费的。decode 是读显存受限的，两个微批次各自做一遍矩阵乘法，本卡的权重（27.3 GB）就要读两遍。所以在超节点里，重叠反而亏：同样打开重叠，域内一步从 24.8 ms 变成 30.9 ms，吞吐降到 2815 tok/s，不如不重叠的 3504 tok/s。

把两边最好的方案比一比：跨机部署靠重叠能做到 2754 tok/s，超节点不重叠就有 3504 tok/s，差约 1.27 倍。**超节点省掉的，除了通信时间本身，还有为了藏住通信付出的代价**：多读一遍权重、把 batch 切小、更复杂的调度。

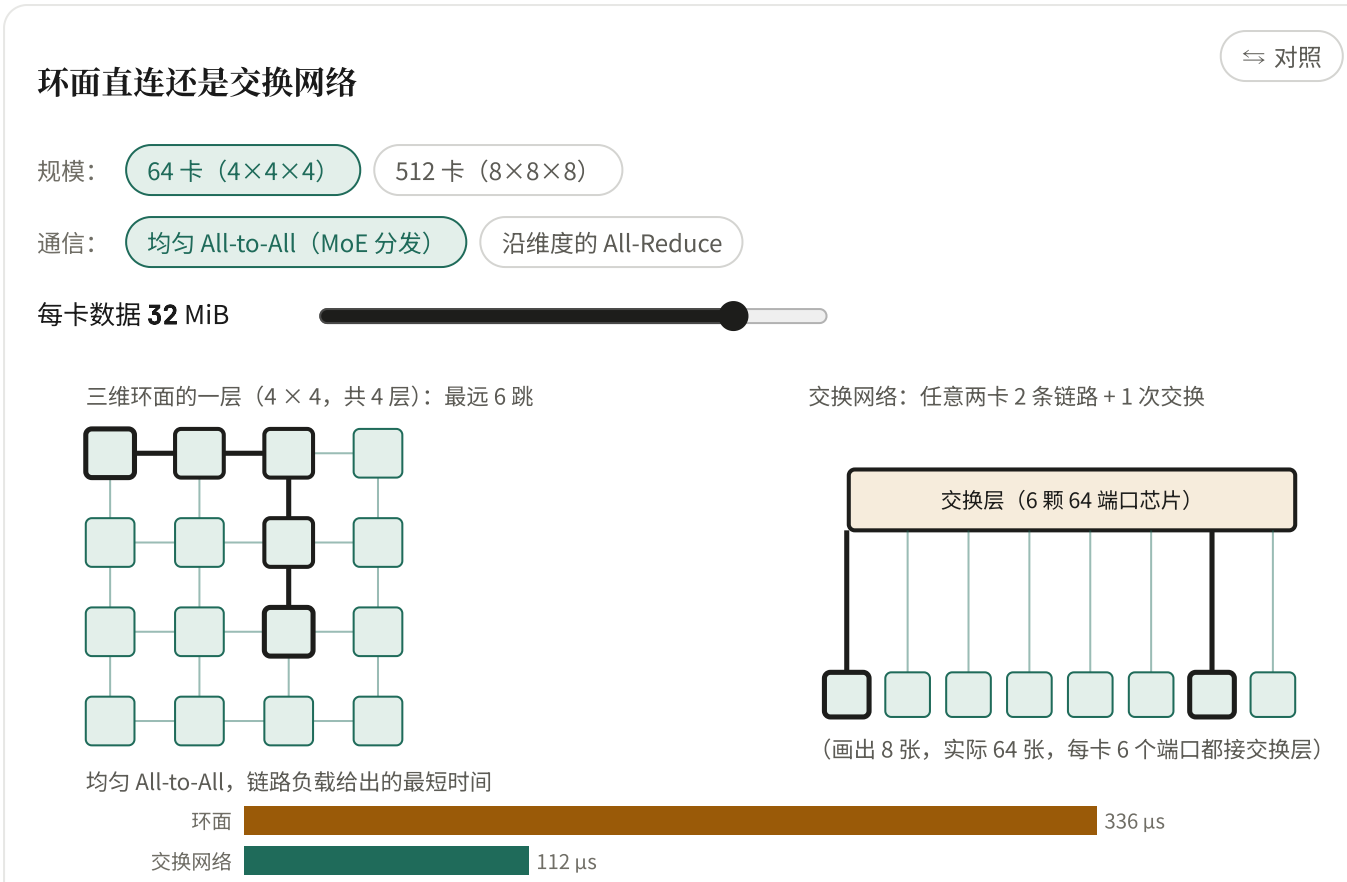
要点

大规模 MoE 是超节点收益最大的负载，但收益有两种来源。一是容量：域越大，每卡分到的专家越少，腾出的显存放更多会话；decode 一步的时间几乎固定（显存读一遍），每卡吞吐正比于会话数。二是通信：All-to-All 不再挤网卡，不需要靠重叠和多读一遍权重去藏它。单个用户的速度与域的大小无关。

超节点里面：交换还是直连

前面默认域内任意两张卡等距，像 NVL72、CloudMatrix384 那样经交换芯片相连。TPU 的环面不一样：数据要经过中间芯片的路由器一跳一跳地走。这对推理有什么影响?

给两种拓扑同样的条件：每张卡 6 个端口，每端口每方向 50 GB/s（TPU v4 的配置）。环面里每个端口接一个邻居；交换网络里每个端口接交换芯片。



环面：每份数据平均走 $\bar{h} = 3 \times 1 = 3$ 跳，所有链路共承载 $3 \times 64 \times M$，摊到 384 条有向链路 → $T \geq \bar{h} \cdot M/(6\beta) = 3 \times 32\ \text{MiB} \div (6 \times 50\ \text{GB/s}) = 336\ \mu\text{s}$。交换网络：每卡 6 个端口各发 $M/6$ → 112 µs，快 3 倍。

最远一对卡：环面经过 5 颗中间芯片，交换网络 1 颗；按每颗 200 ns，环面每次多等 0.8 µs。硬件：环面 192 条链路、不需要交换芯片；交换网络 384 条链路，外加 6 颗交换芯片。

环面便宜、规则通信快；交换网络贵，但任意两卡等距，目的地任意的 All-to-All 快得多，规模越大差距越大。

图 115-4　64 张卡（4 × 4 × 4）或 512 张卡（8 × 8 × 8），每卡 6 个 50 GB/s 端口。沿维度的 All-Reduce 两种拓扑一样快；均匀的 All-to-All（MoE 的分发）在环面上平均要走 3 跳（512 卡时 6 跳），链路负载按跳数放大，交换网络快 3 倍（6 倍）。

规则的集合通信，环面不吃亏。All-Reduce 可以拆成沿 x、y、z 三个方向的环，每一步只在邻居之间传，6 个端口同时工作，带宽和交换网络一样。TPU 的编译器正是把模型的切分维度对到环面的维度上。

目的地任意的 All-to-All，环面吃亏。每张卡要给其余 63 张卡各发一份，环面里每份数据平均要走 3 跳，每多一跳就多占一条链路。64 张卡各发 32 MiB，所有链路共要承载 $3 \times 64 \times 32$ MiB，摊到 384 条有向链路上，每条 16 MiB，至少 **336 µs**；交换网络每卡 6 个端口各发 1/6，只要 **112 µs**。512 张卡时环面平均 6 跳，差距变成 **6 倍**（671 µs 对 112 µs）。

延迟也随跳数增加。环面里最远的两张卡相距 6 跳，中间经过 5 颗芯片；交换网络只经过 1 颗。按每颗芯片 200 ns 算（开放标准 UALink 2.0 给交换芯片定的空载延迟目标），每次传输多等 0.8 μs。场景一说过，小消息的通信几乎全是延迟，decode 每层都要同步，这一点在每一步里会累积起来。

代价在另一边：64 卡的环面只要 192 条链路，不需要任何交换芯片；交换网络要 384 条链路加 6 颗 64 端口的交换芯片，规模超过一颗芯片的端口数就要再加一级（CloudMatrix384 用了 448 颗）。所以，以规则集合通信为主的负载，环面更划算；大规模 MoE 推理以 All-to-All 为主，交换网络更合适。这也是 NVL72 和 CloudMatrix384 都选了交换网络的原因之一。

多出来的显存：借给别人

场景三说明，超节点的价值很大一部分在于容量。顺着这个思路再走一步：域里有的卡显存很紧，有的卡还有空闲，能不能把空闲的借过来？比如把暂时不活跃的会话的 KV 放到别的卡或 CPU 内存上（第 113 讲），需要时再读回来。CloudMatrix384 的 UB 就把 CPU 的内存也接进了同一张网，可以当作一个大的共享池。

借来的内存能用多快，除了链路带宽，还取决于**同时在途的请求数**。读远端内存时，一个请求从发出到拿到结果要一个往返时间 L；如果最多只能有 u 个请求同时在途，每个取 q 字节，那每 L 秒最多拿回 u·q 字节：

$$\text{可用带宽} = \min(\text{链路带宽},\ u \cdot q / L)$$

这就是排队论里的 Little 定律。按网卡 RDMA 算：往返 $L = 2 \times 3.76\ \mu s$，每次读 256 字节（Qwen3-32B 一个 KV 头在一层、一个 token 上的 K 向量），128 个请求在途时只有 **4.4 GB/s**，不到链路的十分之一；要用满 50 GB/s，需要 **1469** 个请求同时在途。域内的 NVLink 往返更短，但带宽也大得多，要用满 450 GB/s 需要 **2890** 个在途请求。

所以远端内存要用好，访问方式比带宽更重要：一次搬大块、批量发起，而不是一个 token 一个 token 地零碎读。还要看读的频率：等用户输入下一轮的会话，KV 可以长时间放在远端，恢复时一次取回；正在 decode 的会话每一步都要读全部 KV，放在远端会持续占用带宽，不如取回本地。

代价和局限

功率和散热。一台 DGX H100（8 张卡）的系统功率上限约 10.2 kW。风冷机柜一般在 40 kW 上下就到头，只放得下 3 台。72 张卡挤进一个机柜，整柜功率到了 100 kW 以上，必须用液冷。

交换芯片成本跳升。一颗交换芯片的端口数有限，卡数一旦超过它能接的数量，就要整整多加一级交换。成本不是随卡数平滑上升，而是在边界处跳一下。

故障影响的范围变大。一个 64 卡的 MoE 实例里，任何一张卡、一根线出问题，整个实例都要停下来，等恢复后重做。实例越大，依赖正常工作的设备就越多。实际系统要准备冗余专家、备用卡和快速重建通信组的能力，CloudMatrix384 的论文和 DeepSeek 的部署都花了不少篇幅在这上面。

软件要跟上。场景一已经看到，算法的轮数决定了小消息通信的耗时；拓扑感知的通信库、按新拓扑设计的并行方式和调度，都是用好超节点的前提。把为 8 卡服务器写的部署方式原样搬过去，很多收益拿不到。

用代码复现

第一段代码复现场景一、场景二：Qwen3-32B 的 TP8 与两种 TP16，Llama-3.1-405B 的 TP8 × PP2 与 TP16，以及 TP16 追平 TP8 的条件和对带宽、延迟的敏感性。参数量由模型配置算出。

```python
# 稠密模型 decode 一步 = 每卡要读的字节 ÷ 显存带宽 + 每层 2 次 All-Reduce（第 70、110 讲的模型）
HBM = 3.35e12                           # H100 显存带宽，B/s
A_NV, B_NV = 0.822e-6, 450e9            # NVLink：每轮延迟（MSCCL++ 论文实测）、每卡单向带宽
A_IB, B_IB = 3.76e-6, 50e9              # InfiniBand：单向延迟（同一论文实测）、每卡网卡带宽
AR_2SRV = 32.74e-6                      # 两台 HGX H100、16 卡、16 KiB All-Reduce 的 nccl-tests 实测

def ring_ar(m, p):                      # 域内 Ring All-Reduce：2(p-1) 轮，每轮每卡传 m/p 字节
    return 2 * (p - 1) * (A_NV + m / p / B_NV)

def dense(L, d, nq, nkv, hd, ffn, vocab):   # 每步要读的权重字节（BF16，不含嵌入表）和每 token 的 KV 字节
    attn = d * (nq + 2 * nkv) * hd + nq * hd * d
    return 2 * (L * (attn + 3 * d * ffn) + vocab * d), 2 * L * nkv * hd * 2

def step(cfg, tp, pp, batch, ctx, ar):
    L, d, nkv = cfg[0], cfg[1], cfg[3]
    w, kv_tok = dense(*cfg)
    kv = batch * ctx * kv_tok / min(tp, nkv) / pp   # KV 按头切，头不够分时每个头存多份
    local = pp * (w / tp / pp + kv) / HBM           # 一个 token 依次走过 pp 段，每段读本段的权重和 KV
    comm = 2 * L * ar(batch * d * 2, tp) + (pp - 1) * (A_IB + batch * d * 2 / B_IB)
    return local, comm, w / tp / pp / 1e9, kv / 1e9

QWEN32 = (64, 5120, 64, 8, 128, 25600, 151936)
L405 = (126, 16384, 128, 8, 128, 53248, 128256)
print(f"Qwen3-32B 每步读权重 {dense(*QWEN32)[0]/1e9:.1f} GB，每 token KV {dense(*QWEN32)[1]/1024:.0f} KiB")
rows = [("Qwen3-32B TP8，一台服务器", QWEN32, 8, 1, 1, 131072, ring_ar),
        ("Qwen3-32B TP16，跨两台服务器", QWEN32, 16, 1, 1, 131072, lambda m, p: AR_2SRV),
        ("Qwen3-32B TP16，一个 NVLink 域", QWEN32, 16, 1, 1, 131072, ring_ar),
        ("405B TP8×PP2，两台服务器", L405, 8, 2, 32, 8192, ring_ar),
        ("405B TP16，一个 NVLink 域", L405, 16, 1, 32, 8192, ring_ar)]
for name, cfg, tp, pp, b, ctx, ar in rows:
    loc, com, w, kv = step(cfg, tp, pp, b, ctx, ar)
    print(f"{name}：每卡权重 {w:.1f} + KV {kv:.2f} GB，读 {loc*1e3:.2f} + 通信 {com*1e3:.2f} = {(loc+com)*1e3:.2f} ms")
loc16 = step(QWEN32, 16, 1, 1, 131072, ring_ar)[0]
loc8, com8 = step(QWEN32, 8, 1, 1, 131072, ring_ar)[:2]
print(f"TP16 要追平 TP8，每次 All-Reduce 须 ≤ {(loc8 + com8 - loc16) / 128 * 1e6:.1f} µs")
```

```
m = 10240                                   # Qwen3-32B 一次 All-Reduce 的消息：5120 × 2 字节
print(f"TP8 一次 All-Reduce {ring_ar(m, 8)*1e6:.2f} μs，其中带宽项只有 {14 * m / 8 / B_NV * 1e9:.0f} ns")
print(f"NVLink 带宽翻倍，每步省 {128 * 14 * m / 8 / B_NV / 2 * 1e6:.1f} μs；每轮延迟减半，每步省 {128 * 14 * A_NV / 2 * 1e3:.2f} ms")
```

输出：

```
Qwen3-32B 每步读权重 64.0 GB，每 token KV 256 KiB
Qwen3-32B TP8，一台服务器：每卡权重 8.0 + KV 4.29 GB，读 3.67 + 通信 1.48 = 5.15 ms
Qwen3-32B TP16，跨两台服务器：每卡权重 4.0 + KV 4.29 GB，读 2.48 + 通信 4.19 = 6.67 ms
Qwen3-32B TP16，一个 NVLink 域：每卡权重 4.0 + KV 4.29 GB，读 2.48 + 通信 3.16 = 5.64 ms
405B TP8×PP2，两台服务器：每卡权重 50.5 + KV 8.46 GB，读 35.18 + 通信 3.95 = 39.13 ms
405B TP16，一个 NVLink 域：每卡权重 50.5 + KV 16.91 GB，读 20.11 + 通信 7.32 = 27.43 ms
TP16 要追平 TP8，每次 All-Reduce 须 ≤ 20.9 μs
TP8 一次 All-Reduce 11.55 μs，其中带宽项只有 40 ns
NVLink 带宽翻倍，每步省 2.5 μs；每轮延迟减半，每步省 0.74 ms
```

对照正文：Qwen3-32B 的三行是 5.15、6.67、5.64 ms；405B 从 39.13 ms 降到 27.43 ms；TP16 要追平 TP8，每次 All-Reduce 不能超过 20.9 μs；一次 All-Reduce 的带宽项只有 40 ns，所以带宽翻倍只省 2.5 μs，延迟减半能省 0.74 ms。

第二段代码复现场景三：DeepSeek-V3 在不同 EP 组大小下每卡的权重、会话数、一步的构成和吞吐，64 张卡分在 8 台服务器上的情况，单个会话独占时的速度，以及两个微批次重叠的效果。

```
# DeepSeek-V3（FP8 权重）decode：注意力按数据并行（每卡服务自己的会话），256 个路由专家均分到域内 S 张卡（EP = S）
HBM, CAP, WS = 3.35e12, 80e9, 2 * 2**30          # 显存带宽、容量、每卡预留的工作区（2 GiB）
F8, BF = 1979e12, 989e12                          # FP8、BF16 稠密算力
A_NV, B_NV, A_IB, B_IB = 0.822e-6, 450e9, 3.76e-6, 50e9
d, L, MOE, E, K, CTX = 7168, 61, 58, 256, 8, 8192
attn = d * 1536 + 1536 * 128 * 192 + d * 576 + 512 * 128 * 256 + 128 * 128 * d   # MLA 一层的参数
expert = 3 * d * 2048
shared = L * attn + 3 * 3 * d * 18432 + MOE * (expert + E * d) + 129280 * d     # 每卡各存一份的部分（不含嵌入表）
routed = MOE * E * expert                                                       # 路由专家，分到 S 张卡
kv = CTX * L * 576 * 2                            # 一个会话的 KV（MLA：每层每 token 576 个 BF16）
flops_tok = 2 * (shared + MOE * K * expert) / F8 + L * 2 * 128 * (576 + 512) * CTX / BF   # 一个 token 的计算时间（秒）

def run(S, per_server=None):
    w = shared + 129280 * d + routed / S          # 每卡权重（再加上嵌入表）
    n = int((CAP - w - WS) // kv)                 # 剩下的显存全给 KV：每卡能放几个会话
    if n < 1: return w, 0, None
    read = (shared + routed / S + n * kv) / HBM  # 每步读：共享部分 + 本卡全部专家（batch 大，几乎都被选中）+ 全部 KV
    comp = n * flops_tok
    byte = n * K * d * 3                          # 每层发出：每 token 8 份，分发 FP8 + 合并 BF16
    if per_server is None:                        # 整个 EP 组在一个 NVLink 域里
        a2a = 2 * A_NV + byte * (S - 1) / S / B_NV
    else:                                         # 分散在多台服务器：发往别的服务器的那部分走网卡
        inside, outside = (per_server - 1) / (S - 1), (S - per_server) / (S - 1)
        a2a = 2 * A_IB + max(byte * inside / B_NV, byte * outside / B_IB)
    t = max(read, comp) + MOE * a2a
    return w, n, (t, read, comp, MOE * a2a)

print(f"每卡共享权重 {(shared + 129280*d)/1e9:.1f} GB，路由专家共 {routed/1e9:.0f} GB，一个 {CTX} token 会话的 KV {kv/1e9:.3f} GB")
for S in [8, 16, 32, 64, 128, 256]:
    w, n, r = run(S)
    if r is None: print(f"域 {S:>3} 卡：每卡权重 {w/1e9:5.1f} GB，放不下"); continue
    t, rd, cp, cm = r
    print(f"域 {S:>3} 卡：每卡权重 {w/1e9:5.1f} GB，{n:3d} 个会话，一步 {t*1e3:4.1f} ms"
          f"（读 {rd*1e3:4.1f}、算 {cp*1e3:4.1f}、通信 {cm*1e3:4.1f}），每卡
```

```
{n/t:5.0f} tok/s")
w, n, (t, rd, cp, cm) = run(64, per_server=8)
print(f"64 卡分在 8 台服务器：{n} 个会话，一步 {t*1e3:.1f} ms（通信 {cm*1e3:.1f}），每卡
{n/t:.0f} tok/s")
one = (shared + MOE * expert + kv) / HBM + MOE * 2 * 2 * A_NV   # batch 1：读共享部
分、每层一个专家、自己的 KV
print(f"单个会话独占时：一步约 {one*1e3:.2f} ms，即 {1/one:.0f} tok/s，与 S 无关")
# 两个微批次重叠：通信可以藏在另一半的计算后面，但每个微批次各读一遍本卡权重
for S, ps in [(64, None), (64, 8)]:
    w, n, (t, rd, cp, cm) = run(S, ps)
    rd2 = rd + (shared + routed / S) / HBM
    t2 = max(rd2, cp, cm) + 2 * MOE * (2 * (A_NV if ps is None else A_IB))
    print(f"{'一个 NVLink 域' if ps is None else '8 台服务器'}，两个微批次重叠：一步
{t2*1e3:.1f} ms，每卡 {n/t2:.0f} tok/s")
```

输出：

```
每卡共享权重 17.1 GB，路由专家共 654 GB，一个 8192 token 会话的 KV 0.576 GB
域   8 卡：每卡权重  98.9 GB，放不下
域  16 卡：每卡权重  58.0 GB，  34 个会话，一步 23.7 ms（读 22.9、算  6.0、通信  0.8），每
卡  1436 tok/s
域  32 卡：每卡权重  37.6 GB，  70 个会话，一步 24.6 ms（读 23.0、算 12.4、通信  1.6），每
卡  2850 tok/s
域  64 卡：每卡权重  27.3 GB，  87 个会话，一步 24.8 ms（读 22.8、算 15.5、通信  2.0），每
卡  3504 tok/s
域 128 卡：每卡权重  22.2 GB，  96 个会话，一步 25.1 ms（读 22.9、算 17.1、通信  2.2），每
卡  3831 tok/s
域 256 卡：每卡权重  19.7 GB，101 个会话，一步 25.3 ms（读 23.0、算 18.0、通信  2.3），每
卡  3996 tok/s
64 卡分在 8 台服务器：87 个会话，一步 38.7 ms（通信 15.9），每卡 2248 tok/s
单个会话独占时：一步约 5.96 ms，即 168 tok/s，与 S 无关
一个 NVLink 域，两个微批次重叠：一步 30.9 ms，每卡 2815 tok/s
8 台服务器，两个微批次重叠：一步 31.6 ms，每卡 2754 tok/s
```

和正文的表格一致：S = 8 放不下；S = 64 时 87 个会话、一步 24.8 ms、每卡 3504 tok/s；同样的卡分在 8 台服务器上，通信 15.9 ms，吞吐 2248 tok/s；重叠后跨机 2754 tok/s，而域内重叠反而从 3504 降到 2815 tok/s。

第三段代码复现拓扑和远端内存的两组数字。

```python
# 拓扑：64 / 512 张卡，每卡 6 个端口、每端口每方向 50 GB/s；均匀 All-to-All，每卡共发 32 MiB
M, PORT = 32 * 2**20, 50e9
for k in [4, 8]:
    n = k ** 3
    avg_hops = 3 * sum(min(i, k - i) for i in range(k)) / k   # 三维环面：每一维平均要走的跳数之和
    torus = avg_hops * n * M / (n * 6) / PORT                 # 所有数据经过的链路总量，摊到 6n 条有向链路
    switch = M / 6 / PORT                                     # 交换网络：每卡 6 个端口各发 1/6
    print(f"{n:>3} 卡：环面最远 {3*(k//2)} 跳、平均 {avg_hops:.0f} 跳，All-to-All ≥ {torus*1e6:.0f} µs；交换网络 ≥ {switch*1e6:.0f} µs")

# 远端读取：每个请求 q 字节、往返 L 秒、最多 υ 个请求在途 → 可用带宽 = min(链路带宽, υ·q/L)
q = 256
for name, L, link in [("网卡 RDMA", 2 * 3.76e-6, 50e9), ("NVLink 域内", 2 * 0.822e-6, 450e9)]:
    need = -(-link * L // q)
    print(f"{name}：128 个在途时 {128*q/L/1e9:.1f} GB/s；要用满 {link/1e9:.0f} GB/s 需 {need:.0f} 个在途请求")
```

输出：

```
 64 卡：环面最远 6 跳、平均 3 跳，All-to-All ≥ 336 µs；交换网络 ≥ 112 µs
512 卡：环面最远 12 跳、平均 6 跳，All-to-All ≥ 671 µs；交换网络 ≥ 112 µs
网卡 RDMA：128 个在途时 4.4 GB/s；要用满 50 GB/s 需 1469 个在途请求
NVLink 域内：128 个在途时 19.9 GB/s；要用满 450 GB/s 需 2890 个在途请求
```

64 卡的 All-to-All 环面至少 336 µs、交换网络 112 µs；512 卡时 671 µs 对 112 µs。远端读取在 128 个请求在途时只有 4.4 GB/s，用满网卡要 1469 个在途请求。

容易混淆的地方

1. **超节点不是更快的 GPU**。它改变的是卡与卡之间的连接。单卡的显存带宽不变，一步“把显存读一遍”的时间就不变；单个用户的 decode 速度基本不受影响。
2. **带宽高不等于通信快**。decode 的 All-Reduce 是几 KiB 的小消息，耗时由轮数 × 每轮延迟决定。超节点让更多卡能参与同一个 TP 组，但卡越多轮数越多，Qwen3-32B 的 TP16 在域内也比 TP8 慢。

3. **MoE 在超节点上变快，主要原因不只是通信**。域越大，每卡专家越少，腾出的显存放更多会话，吞吐跟着会话数走。如果只看通信，会低估“容量”这一项，也会高估“重叠”能弥补的程度。
4. **“72 卡一个域”不等于“72 卡一个模型”**。域可以切成多个实例；它的价值是需要时能组成大实例。
5. **公开规格和估算不等于实测**。这一讲的延迟取自公开的测量，带宽取标称值，模型忽略了排队、负载不均等因素。数字用来比较量级和找出瓶颈，部署前要用实际的负载测。

回顾

超节点把几十上百张卡放进同一个高带宽域。GPU 都按 H100 算，只改连法：

- **放得进 8 卡的稠密模型**几乎不受益，甚至更慢。Qwen3-32B 一个 128K 会话，TP8 一步 5.15 ms；跨两台服务器的 TP16 是 6.67 ms，16 卡 NVLink 域里的 TP16 也有 5.64 ms。原因是 KV 头只能切到 8 份，而小消息的 All-Reduce 由轮数 × 延迟决定。每次 All-Reduce 要降到 20.9 μs 以下，TP16 才能追平。
- **放不进 8 卡的稠密模型**明显受益。Llama-3.1-405B 从 TP8 × PP2 换成域内 TP16，一步从 39.13 ms 降到 27.43 ms。
- **大规模 MoE** 收益最大，来源有两个。一是容量：DeepSeek-V3 的 EP 组从 16 卡扩到 64 卡，每卡会话从 34 个增加到 87 个，吞吐从 1436 提高到 3504 tok/s。二是通信：同样 64 张卡分在 8 台服务器上，不重叠时只有 2248 tok/s，靠重叠也只能做到 2754 tok/s。

超节点内部，交换网络比环面更适合 All-to-All；多出来的显存可以借给别人，但远端读取要有足够多的在途请求。代价是功率和液冷、交换芯片的成本、更大的故障影响范围，以及软件的适配。

我们已经知道了怎么让一个推理集群跑得更快。最后一个问题回到业务：给定每秒多少请求、延迟要满足什么要求，到底需要多少张卡？下一讲：**SLO 与容量规划**。

第116讲 SLO 与容量规划

70 TTFT、TPOT 与吞吐　72 推理成本模型　77 Chunked Prefill　82 推理压测实践　110 推理中的张量并行

第四章讲了很多让推理集群跑得更快的方法。最后回到一个很实际的问题，也是部署大模型服务时一定要回答的问题：**每秒有多少请求、延迟要满足什么要求，需要多少张卡？**卡少了，延迟超标、用户流失；卡多了，大把的钱花在闲置的 GPU 上。

这一讲把前面的知识串起来，做一次**容量规划**：先把延迟要求写成 SLO，再算一个实例在 SLO 内能服务多少请求，最后算出要几个实例、花多少钱。

先把要求写清楚：SLO

第 82 讲引入了 SLO（服务等级目标）。对一个对话服务，常见的 SLO 是：

- **TTFT**：P99 不超过某个值，比如 1 秒。用户发出问题后，最慢的那 1% 也要在 1 秒内看到第一个字。
- **TPOT**：P99 不超过某个值，比如 30 ms（约每秒 33 个 token，比人阅读的速度快得多）。
- 还有可用性等其他目标，这里不展开。

SLO 决定了一个实例能“满载”到什么程度：第 70 讲说过，同时 decode 的请求越多，吞吐越高，每个请求的 TPOT 也越长。**TPOT 的上限，就决定了一个实例最多能同时服务多少请求。**

一个实例能服务多少请求

以 Llama-3-8B、一张 H100 为一个实例，沿用第 72 讲的假设：实际效率是理论值的 60%。请求的输入 1000 个 token，输出 300 个。

一个实例每秒能服务 λ 个请求，要同时满足三件事：

1. **prefill 占时间**。每个请求的 1000 个输入 token 要 prefill，这是计算受限的（第 68 讲）。每秒 λ 个请求，prefill 就占掉 $\lambda \times 1000 \times$（每个 token 的计算时间）这么多比例的 GPU 时间，记为 f。decode 只能用剩下的 $1 - f$。

2. **Little 定律**。排队论里有一个简单而普遍的关系：系统里同时存在的请求数 = 到达率 × 每个请求停留的时间。每个请求要 decode 300 步，每步 TPOT，所以同时在 decode 的请求数 b = λ × 300 × TPOT。
3. **TPOT 由 b 决定**。b 个请求一起 decode 一步要 t(b)（第 70 讲），可 decode 只能用 1 − f 的时间（第 77 讲的分块 prefill 把 prefill 插在 decode 之间），所以实际的 TPOT = t(b) ÷ (1 − f)。

从小到大试 λ，直到 TPOT 超过上限（或 KV Cache 放不下）为止。TPOT 上限 30 ms 时，代码算出：一个实例最多每秒 **15.3 个**请求，此时同时约 136 个请求在 decode，prefill 占掉约 39% 的时间，实际 TPOT 29.7 ms。验证一下 Little 定律：15.3 × 300 × 29.7 ms ≈ 136。

需要几个实例

一个实例不能真的跑到上限。第 82 讲说过，接近饱和点时，排队会让延迟急剧上升；流量还会有波动。所以要留**余量**，比如让每个实例只跑到上限的 70%：

$$\text{实例数} = \lceil \text{总请求率} \div (\text{单实例最大请求率} \times \text{目标使用率}) \rceil$$

每秒 50 个请求：50 ÷ (15.3 × 0.7) ≈ 4.7，向上取整，**5 个实例**，也就是 5 张 H100。按 2.5 美元/小时的示例租金，每百万个请求的 GPU 成本约 69 美元。

再看几种变化：

- **TPOT 上限收紧到 15 ms**：同时 decode 的请求只能有约 43 个，每实例每秒 9.7 个，需要 **8 张卡**，成本约 111 美元/百万请求。延迟要求严一倍，成本贵了六成。
- **输入变长到 4000 token**：prefill 占的时间多了四倍，KV 也大了，每实例每秒只有 4.0 个，需要 **18 张卡**，成本约 250 美元/百万请求。
- **流量涨到每秒 200 个**：单实例能力不变，需要 19 张卡，成本基本不变（规模大了，取整的浪费比例更小）。

下面的计算器可以调这些参数。



图 116-1　容量计算器（Llama-3-8B、H100、60% 效率）。TPOT 上限决定一个实例能同时 decode 多少请求，输入长度决定 prefill 占多少时间；两者一起决定单实例能服务的请求率，再按目标使用率算出需要的卡数和成本。

要点

容量规划先把延迟要求写成 SLO，再算一个实例在 SLO 内的最大请求率：TPOT 上限限制了同时 decode 的请求数，Little 定律把它和请求率联系起来（$b = \lambda \times$ 输出长度 $\times$ TPOT），prefill 还要占掉一部分时间。实例数 = 总请求率 ÷ (单实例最大请求率 × 目标使用率)，向上取整。

模型之外：要测，也要看峰值

上面的计算是**估算**，用来快速判断量级、比较方案。真正上线前，要用第 82 讲的方法实测：在目标负载下逐级加压，测出每个实例在 SLO 内的 goodput，用实测值代替模型算出的 λ。

还要考虑流量的**形状**：

- **峰值，不是平均值**。对话服务的流量有明显的日夜周期，峰值常是平均的两三倍。容量要按峰值规划，或者配合自动扩缩容。
- **请求的分布**。输入、输出长度不是固定的。长尾的超长请求会占用大量的 KV 和 prefill 时间，要单独评估，或者用不同的实例池处理。

- **扩容要时间**。新实例要下载、加载几十 GB 的权重，捕获 CUDA Graph（第 89 讲），往往要几十秒到几分钟。自动扩缩容要提前一点触发，指标常用排队长度、KV Cache 使用率，而不只是 GPU 利用率。

省钱的杠杆

当容量规划的结果太贵时，可以回头看第三、四章的各种方法，它们最终都体现在“单实例的最大请求率”上：

- **前缀缓存**（第 76 讲）：系统提示命中缓存，prefill 的时间大幅减少，f 变小，λ 变大。
- **量化**（第 90–94 讲）：权重和 KV 变小，decode 一步更快，同样的 TPOT 下能放更大的批。
- **投机解码**（第 96 讲）：在低并发、TPOT 要求严的场景，降低延迟。
- **PD 分离**（第 112 讲）：prefill 和 decode 各自按需配卡，TPOT 更稳，余量可以留得更少。
- **调整 SLO 本身**：TPOT 从 15 ms 放宽到 30 ms，成本降了近四成。和业务一起确认真正需要的延迟，往往是最便宜的优化。

启动和积压：扩容要多快

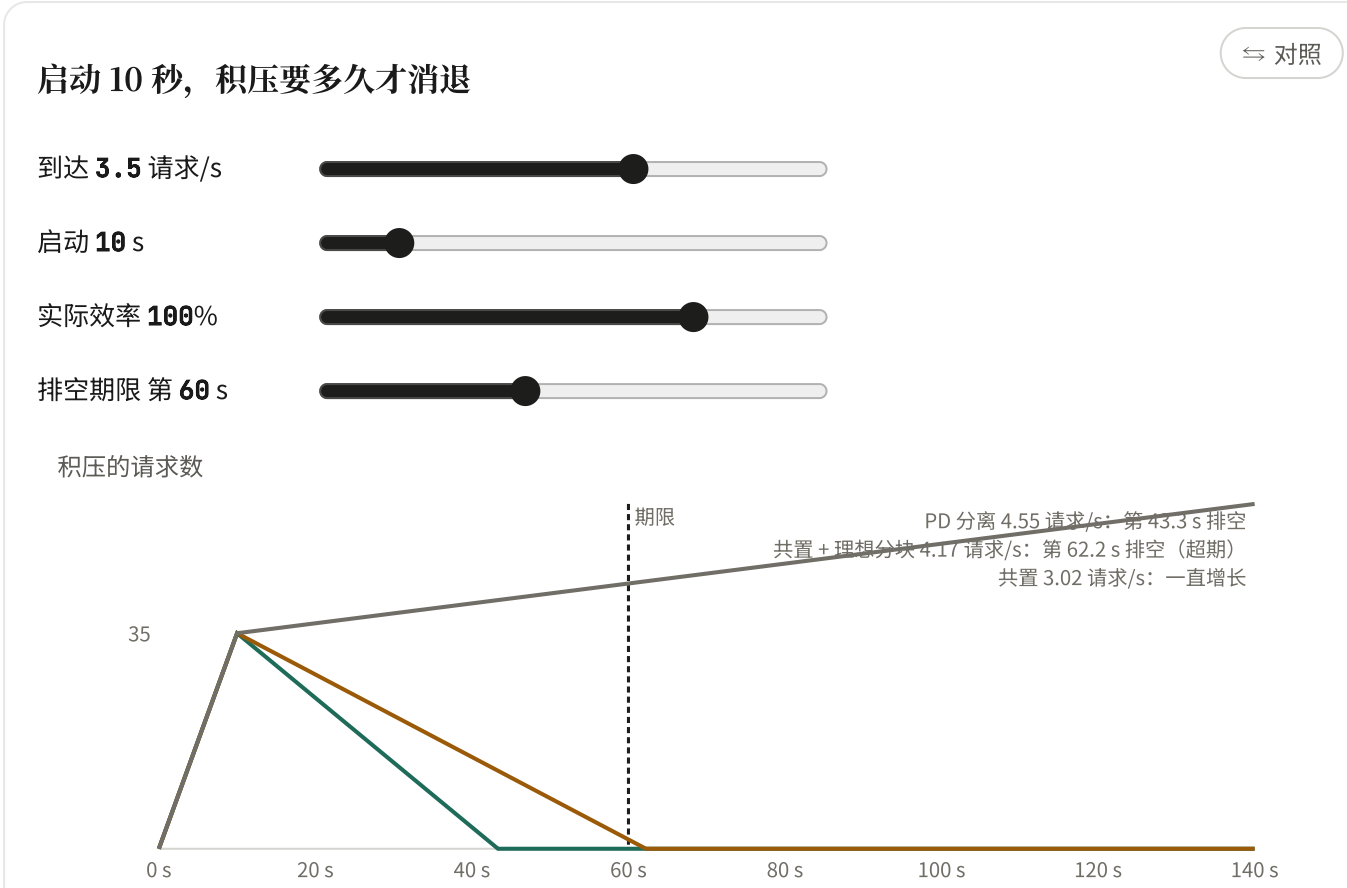
前面算的都是稳态：实例已经在跑。扩容、发布新版本或故障恢复时，新实例要先启动，而请求还在不断到来。

启动要做的事。新实例要创建进程、初始化分词器、读取并切分权重、编译、探测 KV 容量、捕获 CUDA Graph（第 89 讲）。有的准备换来的是之后每一步更快：比如多花 10 s 捕获执行图，每步省 0.2 ms，要 **50,000 步**才持平；每步 32 个请求，相当于约 160 万个输出 token。这笔账按执行的步数算，实例活得越久越划算。

积压怎么消退。设到达率 λ，启动期间服务完全停摆，启动用了 T_s 秒，积压就是 λ·T_s。就绪后服务率是 μ，每秒新来 λ 个，真正能用来消化积压的只有 μ − λ：

$$\text{排空时间} = \lambda \cdot T_s \div (\mu - \lambda)$$

拿第 112 讲的 8 卡异构集群算：每秒 3.5 个请求，启动 10 s，积压 35 个。PD 分离的 4.55 请求/s 每秒只能消化 1.05 个，从开始启动算起第 **43.3 s** 排空；共置加理想的分块捎带是 4.17 请求/s，要到第 **62.2 s**；不分块的共置只有 3.02，低于到达率，积压一直增长。



启动 10 s 积压 λ·T_s = 35.0 个；排空时间 = λ·T_s ÷ (μ - λ)。要在第 60 s 前排空，μ ≥ λ + λ·T_s ÷ (期限 - T_s) = **4.20 请求/s**。
排空的速度是服务率减去到达率的“余量”，不是服务率本身。余量只有 8% 时，效率稍差一点就会超期。

图 116-2　启动 10 s 后三种部署的积压曲线（每秒 3.5 个请求）。下降的斜率是服务率减去到达率；PD 分离第 43.3 s 排空，共置加理想分块第 62.2 s，共置一直积压。把效率调到 90%，PD 也会超过 60 s 的期限。

反过来也能求最低服务率：要在第 60 s 前排空，扣掉启动的 10 s，剩下 50 s 要消化 35 个积压，同时处理新到的请求：

$$\mu \geqslant 3.5 + 35 \div 50 = 4.20 \text{ 请求/s}$$

PD 分离的 4.55 只比它高约 8%。如果实际效率只有估算的九成（4.09 请求/s），稳态还撑得住，排空却要到第 68.8 s，超了期限。这就是第 82 讲说“不能按上限规划”的另一面：余量不只是为了扛波动，也决定了出事之后恢复得多快。

在线迁移。计划内地把一个 decode 实例挪走，可以先在后台复制 KV，源端照常生成，最后短暂停一下补上新增的部分。一张 H20 上 32 个请求的 KV 约 41.1 GB，decode 每秒只追加约 0.172 GB。经 25 GB/s 的网卡约 **1.66 s** 追上源端，经 6.25 GB/s 的 50 GbE 约 **6.76 s**。只要复制

带宽远高于状态增长的速度，追赶时间几乎只由初始的状态大小决定。

软件准备是下限。把 Qwen3-8B 从 TP4 切到 TP8，要经网络搬约 16.4 GB。50 GbE 要 2.62 s，200 Gbit/s 网卡 0.66 s，相差四倍；但如果建通信组、准备执行图、恢复接收请求还要串行地花 9 s，总时间是 11.6 s 和 9.7 s，只差 17%。即使八张卡在同一台机器、走 NVLink、搬数据只要 0.02 s，总时间也还有 9.0 s。这类时间靠更快的链路省不掉，只能靠软件：提前建好通信组、预先捕获执行图、让准备和传输重叠。

故障的影响范围。张量并行组或专家并行组丢了一张卡，整个实例上的请求都会停；其他独立的副本不受影响。所以实例越大，一次故障停掉的服务能力越多（第 115 讲）。恢复还要分清两件事：用户已经收到了哪些 token（要可靠地记录下来，例如预写日志），哪些 KV 已经保存。如果 1025 个输出都记下了，但 KV 只保存了输入部分，恢复时把前 1024 个输出当作一次 prefill 补回去就能接着生成，不必重新采样，用户看到的输出不会变。

按合格的请求算成本

上面的成本按“每百万个请求”算。更准确的分母是**满足质量和时限的请求**：

$$\text{有效成本} = \text{统计期间的总成本} \div \text{合格完成的请求数}$$

还是 8 卡异构集群，按每小时 8 元算：满载时共置每千个请求 0.74 元，PD 分离 0.49 元。可如果 PD 分离只有四分之一的请求满足时限，每千个合格请求就涨到 **1.95** 元，比共置还贵。超时、失败、重做的请求都算在分子里、不算在分母里。提前退出的 Agent 少生成了 token，但没有完成任务，也不能算一次合格的完成。

到了大模型里

多模型、多租户。一个平台常常同时服务很多个模型和客户，不同客户的 SLO 和优先级不同。容量规划要分别评估，再考虑共享资源时的隔离和抢占（第 78 讲）。

按 token 计价的反推。API 服务按每百万 token 收费（第 72 讲）。容量规划算出每百万请求、每百万 token 的 GPU 成本，就能判断定价是否覆盖成本、留出多少利润。

大模型的实例更大。70B 的实例是 4 到 8 张卡（第 110 讲），DeepSeek-V3 的 decode 实例是几百张卡（第 108 讲）；方法完全一样，只是“一个实例”的单位变大，单实例的最大请求率要按整组卡来测。

用代码复现

下面的代码实现了上面的容量模型：对给定的输入、输出长度和 TPOT 上限，从小到大试单实例的请求率 λ，用不动点迭代求出同时 decode 的请求数 b，检查 TPOT 和显存；再按 70% 的使用率算出实例数和每百万请求的成本。

```python
import math
# 容量规划：给定流量和 SLO，需要多少个实例。沿用第 72 讲的假设：实际效率 60%，H100 租金 2.5 美元/小时（示例）
F, BW, EFF, PRICE = 989e12, 3.35e12, 0.6, 2.5
M = dict(N=8.03e9, Ne=525e6, L=32, d=4096, kv=131072)        # Llama-3-8B，一个实例 = 一张 H100

def tpot(b, ctx):                                            # b 个请求一起 decode 一步（第 70 讲）
    flops = (2 * (M["N"] - M["Ne"]) + 4 * M["L"] * M["d"] * ctx) * b
    return max(flops / F, (M["N"] * 2 + M["kv"] * ctx * b) / BW) / EFF

def plan(qps, inp, out, slo_tpot, util=0.7):
    """一个实例每秒最多能服务多少请求 λ，同时满足 TPOT 和显存。
    prefill 是计算受限的，每秒占掉 f = λ·inp·c 的 GPU 时间；decode 只能用剩下的 (1 - f)，所以实际 TPOT = t(b) ÷ (1 - f)。
    同时在 decode 的请求数 b 由 Little 定律决定：b = λ × out × TPOT。"""
    ctx, c = inp + out / 2, 2 * (M["N"] - M["Ne"]) / F / EFF
    best = None
    for k in range(1, 2000):
        lam = k * 0.1
        f = lam * inp * c
        if f ⩾ 1: break
        b = 1.0
        for _ in range(50): b = lam * out * tpot(b, ctx) / (1 - f)       # 不动点迭代求 b
        real = tpot(b, ctx) / (1 - f)
        if real > slo_tpot or M["kv"] * (inp + out) * b > 50e9: break
        best = (lam, b, real)
    lam, b, real = best
    n = math.ceil(qps / (lam * util))                        # 每个实例只跑到 70%，留余量应对波动（第 82 讲）
    cost = n * PRICE / (qps * 3600) * 1e6                    # 每百万个请求的 GPU 成本
    return b, real, lam, n, cost

print("每秒请求 | 输入/输出 | TPOT 上限 | 满载时同时 decode 的请求 | 实际 TPOT | 每实例每秒请求 | 需要实例 | 每百万请求成本")
for qps, inp, out, slo in [(50, 1000, 300, 0.030), (50, 1000, 300, 0.015), (50, 4000, 300, 0.030), (200, 1000, 300, 0.030)]:
    b, t, lam, n, cost = plan(qps, inp, out, slo)
    print(f"{qps:4d} | {inp}/{out} | {slo * 1e3:3.0f} ms | {b:5.0f} | {t * 1e3:5.1f} ms | {lam:5.1f} | {n:3d} | ${cost:6.0f}")
b, t, lam, n, cost = plan(50, 1000, 300, 0.030)
print(f"第一行的含义：一个实例最多每秒 {lam:.1f} 个请求，此时同时有约 {b:.0f} 个在 decode（Little 定律：{lam:.1f} × 300 × {t * 1e3:.1f} ms），"
      f"TPOT {t * 1e3:.1f} ms；按 70% 使用率，每秒 50 个请求要 {n} 个实例")
```

输出：

```
每秒请求 | 输入/输出 | TPOT 上限 | 满载时同时 decode 的请求 | 实际 TPOT | 每实例每秒请求 | 需要实例 | 每百万请求成本
  50 | 1000/300 |  30 ms |   136 |  29.7 ms |  15.3 |   5 | $    69
  50 | 1000/300 |  15 ms |    43 |  14.9 ms |   9.7 |   8 | $   111
  50 | 4000/300 |  30 ms |    35 |  29.5 ms |   4.0 |  18 | $   250
 200 | 1000/300 |  30 ms |   136 |  29.7 ms |  15.3 |  19 | $    66
第一行的含义：一个实例最多每秒 15.3 个请求，此时同时有约 136 个在 decode（Little 定律：15.3 × 300 × 29.7 ms），TPOT 29.7 ms；按 70% 使用率，每秒 50 个请求要 5 个实例
```

四行对应上面的四种情况：基准 5 张卡；TPOT 收紧到 15 ms 要 8 张；输入变长到 4000 要 18 张；流量变成每秒 200 个要 19 张。最后一行解释了第一行的数字，并用 Little 定律验证了它。

第二段代码复现“启动和积压”和“按合格的请求算成本”的数字：三种部署的排空时刻、满足期限的最低服务率、执行图的回本步数、在线迁移和重配置的时间，以及有效成本。

```python
# 1. 启动期间的积压：到达率 λ，启动 Ts 秒，就绪后服务率 μ > λ，积压 λ·Ts 要 λ·Ts/(μ-λ) 秒排空
lam, Ts, deadline = 3.5, 10, 60                    # 第 112 讲的 8 卡异构集群：每秒 3.5 个请求
for name, mu in [("PD 分离", 4.55), ("共置 + 理想分块捎带", 4.17), ("共置", 3.02), ("PD，效率只有九成", 4.55 * 0.9)]:
    if mu ⩽ lam: print(f"{name} ({mu:.2f} 请求/s) ：服务率低于到达率，积压一直增长"); continue
    drain = lam * Ts / (mu - lam)
    print(f"{name} ({mu:.2f} 请求/s) ：积压 {lam*Ts:.0f} 个，从开始启动算起第 {Ts + drain:.1f} 秒排空")
print(f"要在第 {deadline} 秒前排空，服务率至少 {lam + lam*Ts/(deadline - Ts):.2f} 请求/s")

# 2. 启动时多花的准备（比如捕获 CUDA Graph）要多少步才收回
extra, saved = 10, 0.2e-3
print(f"启动多花 {extra} s、每步省 {saved*1e3} ms：{extra/saved:,.0f} 步才持平（批 32 约 {extra/saved*32/1e6:.1f} 百万个 token）")

# 3. 在线迁移一个 decode 实例：后台复制 41.1 GB 的 KV，源端还在以每秒 0.172 GB 增长
kv0, grow = 41.1e9, 1166 * 144 * 1024              # H20 每秒 1166 次 decode，每次追加一个 token 的 KV
for name, bw in [("200 Gbit/s 网卡 (25 GB/s) ", 25e9), ("50 GbE (6.25 GB/s) ", 6.25e9)]:
    print(f"{name}：{kv0/(bw-grow):.2f} s 追上源端")
for name, move in [("50 GbE", 16.4e9 / 6.25e9), ("200 Gbit/s 网卡", 16.4e9 / 25e9), ("机内 NVLink", 4.7e9 / 300e9)]:
    print(f"TP4→TP8 重配置，{name}：搬数据 {move:.2f} s + 软件准备 9 s = {move + 9:.1f} s")

# 4. 按合格请求算成本：8 张卡每小时 8 元
for name, mu, ok in [("共置满载", 3.02, 1), ("PD 满载", 4.55, 1), ("PD，但只有四分之一的请求满足时限", 4.55, 0.25)]:
    print(f"{name}：每千个合格请求 {8 / (mu * ok * 3600) * 1000:.2f} 元")
```

输出：

```
PD 分离（4.55 请求/s）：积压 35 个，从开始启动算起第 43.3 秒排空
共置 + 理想分块捎带（4.17 请求/s）：积压 35 个，从开始启动算起第 62.2 秒排空
共置（3.02 请求/s）：服务率低于到达率，积压一直增长
PD，效率只有九成（4.09 请求/s）：积压 35 个，从开始启动算起第 68.8 秒排空
要在第 60 秒前排空，服务率至少 4.20 请求/s
启动多花 10 s、每步省 0.2 ms：50,000 步才持平（批 32 约 1.6 百万个 token）
200 Gbit/s 网卡（25 GB/s）：1.66 s 追上源端
50 GbE（6.25 GB/s）：6.76 s 追上源端
TP4→TP8 重配置，50 GbE：搬数据 2.62 s + 软件准备 9 s = 11.6 s
TP4→TP8 重配置，200 Gbit/s 网卡：搬数据 0.66 s + 软件准备 9 s = 9.7 s
TP4→TP8 重配置，机内 NVLink：搬数据 0.02 s + 软件准备 9 s = 9.0 s
共置满载：每千个合格请求 0.74 元
PD 满载：每千个合格请求 0.49 元
PD，但只有四分之一的请求满足时限：每千个合格请求 1.95 元
```

对照正文：PD 第 43.3 s 排空、共置加分块第 62.2 s、效率九成时第 68.8 s；最低服务率 4.20 请求/s；50,000 步持平；迁移 1.66 s 和 6.76 s；重配置 11.6、9.7、9.0 s；每千个合格请求 0.74、0.49、1.95 元。

容易混淆的地方

1. **GPU 利用率不是容量的好指标**。decode 时 GPU 可能显示“100% 忙”，算力却只用了一小部分（第 89 讲）。容量要看 SLO 下的请求率，而不是利用率。

2. **余量不是浪费**。接近饱和点时延迟会急剧变差，余量是为了让 P99 守得住。目标使用率定多少，取决于流量的波动和 SLO 的严格程度。

3. **平均延迟满足不代表 SLO 满足**。SLO 通常针对 P99，容量规划和压测都要看尾延迟。

4. **排空积压靠的是余量，不是服务率**。排空时间 = $\lambda \cdot T_s \div (\mu - \lambda)$。服务率只比到达率高一点时，稳态看着没问题，一次重启或扩容就要很久才能恢复。

5. **链路再快，也省不掉软件准备的时间**。重配置、扩容时，建通信组、准备执行图常常比搬数据更久。

回顾

容量规划先把延迟要求写成 SLO（比如 TTFT P99、TPOT P99），再算一个实例在 SLO 内能服务的最大请求率：TPOT 的上限限制了同时 decode 的请求数，Little 定律（$b = \lambda \times$ 输出长度 $\times$ TPOT）把它和请求率联系起来，prefill 还要占掉一部分 GPU 时间。Llama-3-8B、H100、每秒 50 个请求（输入 1000、输出 300）：TPOT $\leqslant$ 30 ms 时每实例 15.3 个/s，按 70% 使用率要 5 张卡；TPOT $\leqslant$ 15 ms 要 8 张；输入 4000 要 18 张。上线前要实测 goodput，按峰值规划，并考虑扩容的时间。前缀缓存、量化、PD 分离和放宽 SLO，都是降低成本的杠杆。

容量规划还要看非稳态：启动 T_s 秒留下 $\lambda \cdot T_s$ 个积压，排空时间是 $\lambda \cdot T_s \div (\mu - \lambda)$。8 卡异构集群每秒 3.5 个请求、启动 10 s，PD 分离第 43.3 s 排空，要在 60 s 内排空至少需要 4.20 请求/s，效率差一成就超期。成本的分母应该是合格完成的请求：只有四分之一合格时，0.49 元会变成 1.95 元。

第四章到这里结束。我们从单卡的一次前向，一路讲到了几百张卡的推理集群。最后一章，我们动手把这些原理串起来：从最简单的逐 token 生成开始，一步步写出一个迷你推理引擎，每个版本加一种优化，看它到底快了多少。下一讲：**迷你推理引擎 v1：带 KV Cache 的逐 token 生成**。

AI INFRA · HANDBOOK

第 15 篇

手写迷你推理引擎

收录 9 讲 · 第 117—125 讲

第117讲 一个推理引擎的组成

68 Prefill 与 Decode　74 Continuous Batching　75 PagedAttention　81 请求生命周期

前四章把推理引擎拆成了一个个零件来讲：KV Cache（第 69 讲）、continuous batching（第 74 讲）、PagedAttention（第 75 讲）、量化（第 90 讲）、投机解码（第 96 讲）。每个零件都讲清楚了原理和收益，可它们拼在一起是什么样子，第 81 讲只画了一张模块图。

这一篇换一种方式：我们从一个空目录开始，**亲手写出一个推理引擎**。它叫 miniinfer，用真实的 Qwen3-0.6B 模型，最后一版约 400 行 Python。每一讲加一个零件，每一讲结束时它都能跑，而且生成的每个 token 都和 HuggingFace transformers 完全一样。

这一讲不写代码，先把全局看清楚：一个推理引擎由哪些部分组成，我们按什么顺序把它们写出来，以及怎么判断写对了。

一个请求要经过哪些地方

先跟着一个请求走一遍。用户发来一句 “The capital of France is”，想让模型接着写。

1. **分词**：分词器把文字切成 token 编号，这句话是 5 个 token（第 21 讲）。
2. **进队列**：引擎为这个请求建一个记录，记下它的 token、最多生成多少个、现在处于什么状态，放进等待队列。
3. **调度**：每一步，调度器决定这一步 batch 里有谁：已经在生成的请求各算一个 token，有空位就放新请求进来。
4. **准备 KV 的位置**：新请求要为它的提示分配 KV Cache 的空间；正在生成的请求，下一个位置如果超出了已分配的空间，要再分一块。
5. **前向**：把这一步所有请求要算的 token 拼成一批，送进模型。新请求是整段提示（prefill），老请求是一个 token（decode）。模型一层一层地算，注意力从 KV Cache 里读每个请求自己的上下文，同时把新 token 的 K、V 写进去。
6. **采样**：每个请求取最后一个位置的打分，选出下一个 token。
7. **更新状态**：新 token 追加到请求里；遇到结束符或达到长度上限的请求离开 batch，释放它的 KV 空间。
8. 回到第 3 步，直到所有请求都结束；最后把 token 还原成文字。

这八步就是推理引擎的全部骨架。前四章讲过的优化，都是在其中某一步上做文章：continuous batching 改的是第 3、7 步，分页改的是第 4 步，量化改的是第 5 步用的权重，投机解码改的是第 5、6 步。

架构：五个部分

把上面的八步按“谁负责”归类，推理引擎可以分成五个部分。下图是 miniinfer 最后一版的样子，切换版本按钮可以看它是怎么一步步长出来的，点模块看它在哪个文件、做什么。

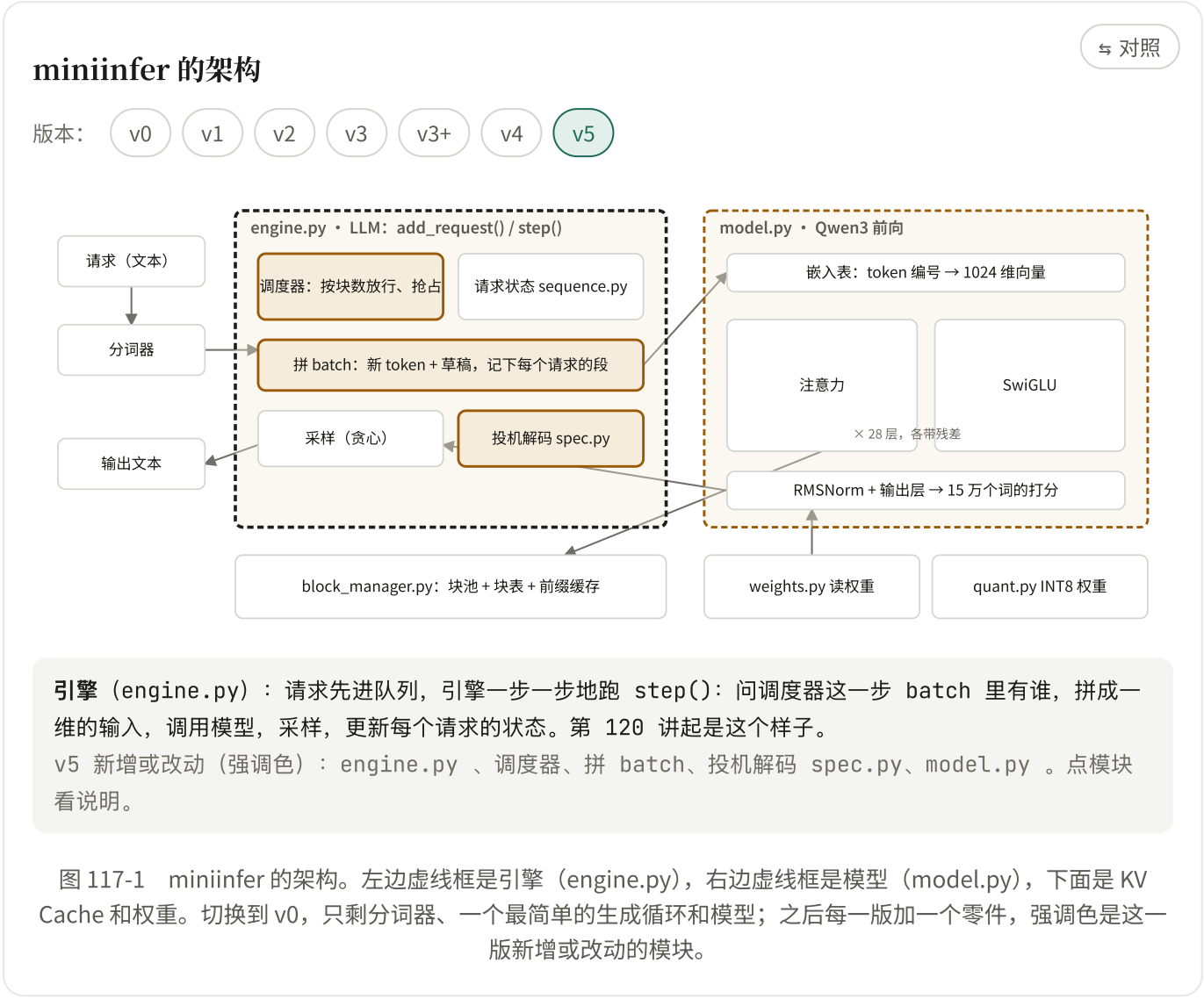


图 117-1　miniinfer 的架构。左边虚线框是引擎（engine.py），右边虚线框是模型（model.py），下面是 KV Cache 和权重。切换到 v0，只剩分词器、一个最简单的生成循环和模型；之后每一版加一个零件，强调色是这一版新增或改动的模块。

模型执行（model.py、weights.py、config.py）：把权重文件读进来，按 Qwen3 的结构一层一层地算。这是唯一碰矩阵运算的地方，其余部分都是“组织工作”。

状态（kv_cache.py，之后换成 block_manager.py）：每个请求已经算过的 K、V 存在哪里、怎么读写。

调度（scheduler.py、sequence.py）：每一步谁进 batch，谁等待，谁被抢占。

内存管理（block_manager.py）：KV 的空间怎么分配、回收、共享。

加速手段（quant.py、spec.py）：在不改变上面结构的前提下，让每一步更省或者一步干更多的活。

路线：六个版本

我们按依赖顺序写，每一版只解决上一版最明显的一个毛病：

版本	讲	解决什么问题	主要改动
v0	第 118 讲	先让模型跑起来：从权重文件算出第一个 token	新写 config.py、weights.py、model.py、engine.py
v1	第 119 讲	v0 每生成一个 token 都把整段重算一遍	新增 kv_cache.py；模型分出 prefill 和 decode
v2	第 120 讲	一次只服务一个请求，硬件大部分时间闲着	新增 sequence.py、scheduler.py；引擎改成 step() 循环
v3	第 121、122 讲	每个请求按最大长度预留 KV，放不下几个请求；相同的前缀重复计算	新增 block_manager.py，取代 kv_cache.py
v4	第 123 讲	权重太占内存	新增 quant.py
v5	第 124 讲	decode 每步只出一个 token	新增 spec.py

最后在第 125 讲复盘：时间花在哪里，和工业级引擎差在哪里，作为下一篇解读 nano-vllm 的引子。

每一版都是仓库里一个完整的目录（`v0` 到 `v5`），后一版从前一版复制过来再改。想看一讲到底改了什么，`diff -r v1 v2` 就行；文章里每讲也会用差异视图把改动标出来。

为什么这样选

模型：Qwen3-0.6B。它是一个结构完整的现代大模型：RMSNorm、RoPE、GQA（16 个查询头、8 个 KV 头）、SwiGLU，和第 35–39 讲讲的一样；28 层，每层 1024 维，权重约 1.5 GB，一台笔记本就放得下。用真实模型，生成的是有意义的文字，量化和投机解码的效果也看得出

来。

设备：CPU，FP32。这一篇的数字都是在一台 Apple M2（16 GB 内存）上用 CPU 跑出来的。选 FP32 不是为了快，而是为了**能和 HuggingFace 逐 token 比对**：同样的精度、同样的算法，两边算出的每个 token 都应该一模一样，任何一处写错都会立刻暴露。代价是速度：后面每一讲的绝对数字都比 GPU 慢一两个数量级，我们关心的是版本之间的相对变化，以及这些变化能不能用前面讲的原理解释。

只用最基本的工具。模型前向只用 torch 的矩阵乘法、逐元素运算和 softmax，不用 `torch.nn` 的模块，也不用 FlashAttention；分词直接用模型自带的 tokenizer.json。这样每一行代码都能讲清楚在算什么。CUDA kernel、CUDA Graph、张量并行这些“工业级”的部分，留到第 16、17 篇读 nano-vllm、vLLM 时再看。

> **要点**
> 推理引擎的骨架是一个循环：调度器决定这一步 batch 里有谁，为它们准备好 KV 的空间，把这一步要算的 token 拼成一批送进模型，采样出下一个 token，更新每个请求的状态。前四章讲的各种优化，都是在这个循环的某一步上做文章。

搭好环境

仓库在课程目录下的 `miniinfer/`。从零开始搭，只要三条命令：建一个虚拟环境装依赖，下载模型，生成 HuggingFace 的参考答案。

```
cd miniinfer && python3 -m venv .venv && .venv/bin/pip install torch transformers
safetensors huggingface_hub
```

```
.venv/bin/python -c "from huggingface_hub import snapshot_download as d;
d('Qwen/Qwen3-0.6B', local_dir='models/Qwen3-0.6B')"
```

```
.venv/bin/python tests/hf_ref.py
```

transformers 只在测试里用来生成参考答案，引擎本身不依赖它。目录结构如下：

```
miniinfer/
  models/Qwen3-0.6B/   config.json、model.safetensors、tokenizer.json
  v0/ … v5/            每个版本一个完整的引擎
  tests/hf_ref.py      用 HuggingFace 生成参考答案（4 条提示，各 32 个 token，贪心）
  tests/check.py       用某个版本生成，和参考答案逐 token 比对
  bench/               每一讲的压测脚本
  outputs/             脚本实际运行的输出（文章里的数字都来自这里）
```

怎么判断写对了

写引擎最怕“看起来能跑，其实算错了”：注意力的掩码差一位、RoPE 的位置从 1 开始数、KV 写错了位置，生成的文字往往还挺通顺，只是和正确的模型不一样。所以我们从第一版开始就定一个硬标准：

同样 4 条提示，各贪心生成 32 个 token，每个 token 都必须和 HuggingFace 一样。

4 条提示有英文、代码、中文、数字序列，覆盖不同的分词和不同的上下文长度。`tests/hf_ref.py` 用 transformers 生成参考答案存成 JSON；`tests/check.py v2` 用 v2 的引擎跑同样的提示，逐 token 比对。每一讲写完都要先过这一关，再测速度。下面是最后一版的检查结果：

```
$ .venv/bin/python tests/check.py v5                                    ⇆ 对照

一致: 'The capital of France is' → ' Paris. The capital of Italy is Rome. The capital
of Spain is Madrid. The capital of China is Beijing. The capital of Japan is Tokyo.
The capital'
一致: 'def fibonacci(n):' → '\n    if n == 0:\n        return 0\n    elif n == 1:\n
return 1\n    else:\n        return fibonacci(n'
一致: '一个推理引擎由哪几部分组成？' → '它如何运作？  请提供一个例子，说明如何使用它来解决一个实际问
题。\n答案:  问题: 某公司需要为员工'
一致: '1, 2, 3, 4, 5,' → ' 6, 7, 8, 9, 10, 11, 12, 13, 14,'
v5: 4/4 条与 HuggingFace 逐 token 一致
```

图 117-2　v5（带投机解码）的检查结果：4 条提示都和 HuggingFace 逐 token 一致。

只有 v4（量化）例外：它有意改变了权重的数值，生成结果允许不同，我们会量化“差了多少”。

回顾

推理引擎的骨架是一个循环：分词 → 进队列 → 调度 → 准备 KV 空间 → 前向 → 采样 → 更新状态，直到所有请求结束。按职责分成五个部分：模型执行、状态、调度、内存管理和加速手段。我们用 Qwen3-0.6B，在 CPU 上以 FP32 运行，从 v0 写到 v5，每一版解决一个问题，每一版都必须和 HuggingFace 逐 token 一致。

第一件事是让模型跑起来：一个 1.5 GB 的权重文件，怎么变成 “Paris” 这个词？下一讲：**从权重文件到第一个 token**。

第118讲 从权重文件到第一个 token

33 输出层　34 前向传播　36 Pre-Norm 与 RMSNorm　37 RoPE 旋转位置编码　39 MQA 与 GQA　117 一个推理引擎的组成

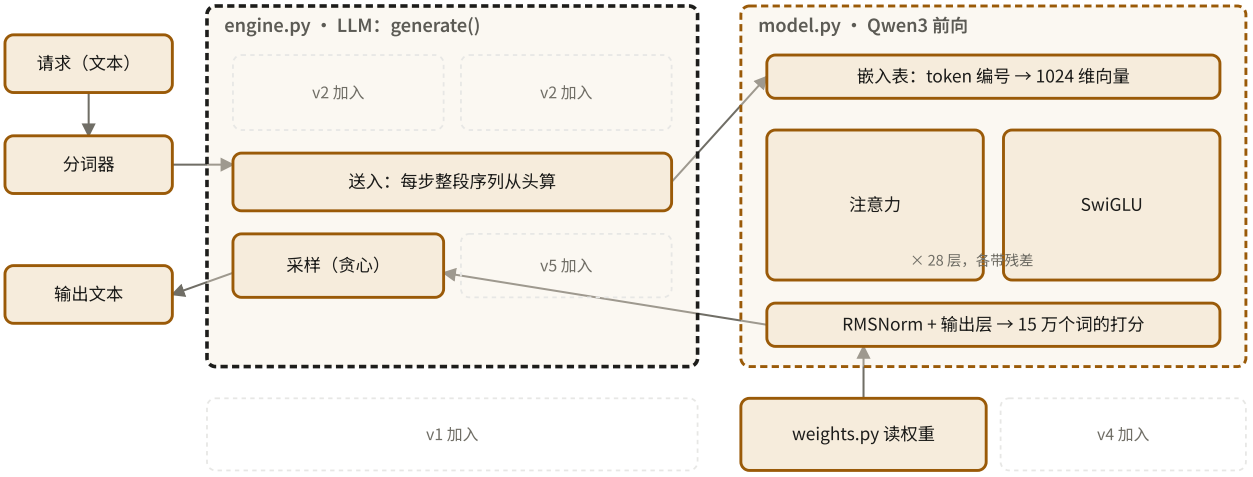
我们手里有一个 1.5 GB 的文件 `model.safetensors`，和一句提示 “The capital of France is”。这一讲的目标只有一个：**不用 transformers，自己写代码，让这个文件算出下一个词是 “Paris”**，而且接下来生成的 32 个 token 和 HuggingFace 一个不差。

这就是 v0。它很慢（下一讲就会看到慢在哪里），但它是后面所有版本的地基：模型前向写对了，之后的 KV Cache、批处理、分页都只是在“怎么组织调用”上做文章。

miniinfer 的架构

⇆ 对照

版本：v0 v1 v2 v3 v3+ v4 v5



引擎（engine.py）：一个请求一个请求地生成。v0 每步把整段序列送进模型；v1 先 prefill 整个提示，之后每步只送一个新 token（第 118、119 讲）。

v0 新增或改动（强调色）：engine.py 、请求（文本）、分词器、输出文本、送入、采样（贪心）、model.py 、嵌入表、注意力、SwiGLU、RMSNorm + 输出层 → 15 万个词的打分、weights.py 读权重。点模块看说明。

图 118-1　v0 只有四个文件：config.py 读模型的尺寸，weights.py 读权重，model.py 写前向，engine.py 写生成循环。分词直接用模型自带的 tokenizer.json。

权重文件里有什么

先打开文件看一眼。safetensors 是一种很简单的格式：文件头是一段 JSON，记录每个张量的名字、形状、数据类型和在文件里的位置，后面是连续存放的原始数据。

```
$ .venv/bin/python bench/inspect_weights.py                    ⇆ 对照

311 个张量，共 752 M 个参数，数据类型 BF16
  lm_head.weight                                  [151936, 1024]
  model.embed_tokens.weight                       [151936, 1024]
  model.layers.0.input_layernorm.weight           [1024]
  model.layers.0.mlp.down_proj.weight             [1024, 3072]
  model.layers.0.mlp.gate_proj.weight             [3072, 1024]
  model.layers.0.mlp.up_proj.weight               [3072, 1024]
  model.layers.0.post_attention_layernorm.weight [1024]
  model.layers.0.self_attn.k_norm.weight          [128]
  model.layers.0.self_attn.k_proj.weight          [1024, 1024]
  model.layers.0.self_attn.o_proj.weight          [1024, 2048]
  model.layers.0.self_attn.q_norm.weight          [128]
  model.layers.0.self_attn.q_proj.weight          [2048, 1024]
  model.layers.0.self_attn.v_proj.weight          [1024, 1024]
  model.norm.weight                               [1024]
```

图 118-2　Qwen3-0.6B 的权重：311 个张量，BF16。这里只列出第 0 层和层外的张量，其余 27 层的名字和形状都一样。

对照第 34 讲的完整前向，每个名字都能对上号：

- `embed_tokens`：嵌入表，[151936, 1024]，每个 token 一行 1024 维（第 22 讲）。
- 每层 4 个注意力投影：`q_proj` 是 [2048, 1024]，`k_proj`、`v_proj` 是 [1024, 1024]，`o_proj` 是 [1024, 2048]。注意 q 的输出是 2048 维 = 16 个头 × 128，而 k、v 只有 1024 维 = 8 个头 × 128：这就是 GQA（第 39 讲），两个查询头共用一组 K、V。还要注意 16 × 128 = 2048 比隐藏维度 1024 大，Qwen3 的头维度不等于 d ÷ 头数。
- `q_norm`、`k_norm`：只有 128 个数。这是 Qwen3 特有的：在 RoPE 之前，对每个头的 q、k 各做一次 RMSNorm，让注意力打分更稳定。
- 每层 3 个 FFN 矩阵：`gate_proj`、`up_proj` 是 [3072, 1024]，`down_proj` 是 [1024, 3072]，这是 SwiGLU（第 38 讲）。
- 两个 RMSNorm 的缩放（各 1024 个数），加上最后的 `model.norm`。

- `lm_head`：输出层，[151936, 1024]。它和嵌入表一模一样（`tie_word_embeddings`），文件里存了两份，所以 752 M 个参数里有 1.56 亿是重复的，模型实际约 6 亿参数，这就是“0.6B”。

PyTorch 的线性层把权重存成 [输出, 输入]，所以 $y = x \cdot W^{T}$。后面的代码里到处是 `x @ W.T`，就是这个原因。

---

config.py 和 weights.py：读进来

层数、头数这些数从 `config.json` 读，代码里不写死任何一个尺寸，换一个同结构的模型（比如 Qwen3-1.7B）也能直接用。

v0/config.py 共 33 行 ⇆ 对照

```python
"""模型配置：从 config.json 读出前向计算需要的几个数。"""
import json
from dataclasses import dataclass


@dataclass
class Config:
    vocab_size: int
    hidden_size: int          # d，每个 token 的向量长度
    intermediate_size: int    # FFN 中间层的宽度
```

1 **需要的只有这几个数** 第 6-17 行

前向计算只需要 10 个数：词表大小、隐藏维度 d、FFN 宽度、层数、查询头数、KV 头数、头维度、RoPE 的 θ、RMSNorm 的 ε，以及结束符的编号。

用 `dataclass` 而不是直接用字典，是为了在代码里写 `c.num_kv_heads` 这样有名字的属性，写错名字会立刻报错。

2 **从 config.json 对应过来** 第 19-33 行

← 上一步 | 看整个文件 | 下一步 →

v0/weights.py 共 13 行 ⇆ 对照

```python
"""从 safetensors 文件读出全部权重，按名字放进一个字典。"""
import torch
from safetensors.torch import load_file


def load_weights(model_dir, dtype=torch.float32, device="cpu"):
    raw = load_file(f"{model_dir}/model.safetensors")
    w = {name: t.to(dtype=dtype, device=device) for name, t in raw.items()}
    # Qwen3-0.6B 的输出层和嵌入表是同一份权重（config.json 里 tie_word_embeddings 为 true
    # 这个文件里只有一份，按 lm_head 没有的话，直接用嵌入表
```

1 读出来，转成需要的精度 第 6–8 行

`safetensors.torch.load_file` 把整个文件读成一个“名字 → 张量”的字典。文件里是 BF16，这里统一转成 FP32：精度和 HuggingFace 的参考答案一致，才能逐 token 比对（第 117 讲）。

FP32 的权重有 752 M × 4 字节 ≈ 3 GB，16 GB 内存的笔记本放得下。

2 共用的输出层 第 9–13 行

← 上一步 | 看整个文件 | 下一步 →

model.py：一层一层地算

这是这一讲的主体。整个 Qwen3 的前向只有 59 行，用到的运算只有矩阵乘、逐元素运算、softmax 和查表。左边是代码，右边逐段解释；点步骤或者“下一步”，左边会高亮对应的行。

v0/model.py 共 59 行 Qwen3 的 ⇆ 对照

```python
"""Qwen3 的前向计算：只用 torch 的基本运算，一层一层写出来。"""
import torch


def rms_norm(x, weight, eps):
    # 每个向量除以自己的均方根，再乘可学习的缩放（第 36 讲）
    var = x.pow(2).mean(-1, keepdim=True)
    return x * torch.rsqrt(var + eps) * weight

```

1 **RMSNorm** 第 5-8 行

每个向量除以自己的均方根，再乘一组可学习的缩放（第 36 讲）。`mean(-1, keepdim=True)` 沿最后一维（1024 维）求平均，保留这一维方便广播。

用 `rsqrt`（平方根的倒数）再相乘，而不是相除，是常见的写法；ε 防止除以零。

2 **RoPE：按位置旋转** 第 11-19 行

3 **注意力：q、k、v 投影** 第 26-31 行

4 **q、k 的 RMSNorm，然后 RoPE** 第 32-34 行

5 **GQA：把 K、V 复制给多个查询头** 第 35-36 行

6 **打分、掩码、softmax、加权求和** 第 37-41 行

7 **SwiGLU** 第 43-46 行

8 **整个前向** 第 48-59 行

← 上一步 | 看整个文件 | 下一步 →

注意

写模型前向时，最容易出错的四个地方：RoPE 的配对方式（前后两半，而不是相邻两维）；GQA 的复制方式（`repeat_interleave`，而不是 `repeat`）；Qwen3 在 RoPE 之前对 q、k 做的 RMSNorm；位置从 0 开始数。这四处任何一处写错，模型都还能输出看似通顺的文字，只有和参考答案逐 token 比对才能发现。

---

engine.py：生成循环

有了前向，生成就是一个循环：把当前的全部 token 送进模型，取最后一个位置打分最高的 token，接到后面，再来一遍。

v0/engine.py 共 32 行 ⇆ 对照

```python
"""v0: 没有 KV Cache 的生成。每生成一个 token，都把整段序列从头算一遍。"""
import torch
from tokenizers import Tokenizer
from config import Config
from weights import load_weights
from model import Qwen3


class LLM:
    def __init__(self, model_dir, device="cpu", dtype=torch.float32):
```

1 **把三样东西装起来** 第 10-15 行

配置、分词器、模型。分词器用 `tokenizers` 库直接读模型自带的 `tokenizer.json`，和 HuggingFace 用的是同一份规则。

2 **每生成一个 token，整段重算一遍** 第 17-25 行

3 **给测试用的接口** 第 27-33 行

← 上一步 | 看整个文件 | 下一步 →

跑起来

```
$ cd v0 && ../.venv/bin/python run.py
生成 8 个 token：1.56 s，平均每个 195 ms
生成 32 个 token：6.95 s，平均每个 217 ms
'The capital of France is Paris. The capital of Italy is Rome. The capital of Spain is
Madrid. The capital of China is Beijing. The capital of Japan is Tokyo. The capital'
```

图 118-3　v0 的第一次运行。模型接着写出了 “Paris”，还顺着往下列了几个国家的首都。

模型说对了。再用 `tests/check.py` 和 HuggingFace 逐 token 比一比：

```
$ .venv/bin/python tests/check.py v0
一致：'The capital of France is' → ' Paris. The capital of Italy is Rome. The capital
of Spain is Madrid. The capital of China is Beijing. The capital of Japan is Tokyo.
The capital'
一致：'def fibonacci(n):' → '\n    if n == 0:\n        return 0\n    elif n == 1:\n
return 1\n    else:\n        return fibonacci(n'
一致：'一个推理引擎由哪几部分组成？' → '它如何运作？  请提供一个例子，说明如何使用它来解决一个实际问
题。\n答案：  问题：某公司需要为员工'
一致：'1, 2, 3, 4, 5,' → ' 6, 7, 8, 9, 10, 11, 12, 13, 14,'
v0：4/4 条与 HuggingFace 逐 token 一致
```

图 118-4　v0 和 HuggingFace 逐 token 一致。四条提示覆盖英文、代码、中文和数字序列。

四条全部一致：我们 59 行的前向，和 transformers 的实现算出了同样的结果。

它慢在哪里

上面生成 32 个 token 用了约 7 秒，平均每个 200 多毫秒。看起来每个 token 的耗时差不多，那是因为提示很短。把提示加长到 199 个 token、生成 64 个，每一步的耗时就看得出变化了（这个压测下一讲会完整展示）：第 1 步约 419 ms，第 64 步约 724 ms，64 步一共约 34 秒。

原因就在生成循环里：第 n 步要算 199 + n 个 token 的前向，越往后越长。总计算量大约是 64 × 230 ≈ 15,000 个 token 的前向，而真正“新”的只有 199 + 64 = 263 个。

前面的 token 每一步都在重算，结果却和上一步一样。能不能把算过的东西存下来？第 68、69 讲已经给出了答案：把每层的 K、V 存起来，新 token 只需要和存好的 K、V 做注意力。

回顾

v0 用四个文件让 Qwen3-0.6B 跑了起来：config.py 读模型的 10 个尺寸，weights.py 把 311 个张量读进字典，model.py 用 59 行写出完整的前向（RMSNorm、RoPE、q/k 归一化、GQA、因果注意力、SwiGLU、残差），engine.py 用一个循环做贪心生成。和 HuggingFace 比对，4 条提示的 32 个 token 全部一致。

它的毛病是每生成一个 token 都要把整段序列重算一遍：199 个 token 的提示，第 64 步要算 263 个 token，耗时从 419 ms 涨到 724 ms。下一讲：**KV Cache**。

第119讲 KV Cache

重点

68 Prefill 与 Decode　69 KV Cache　118 从权重文件到第一个 token

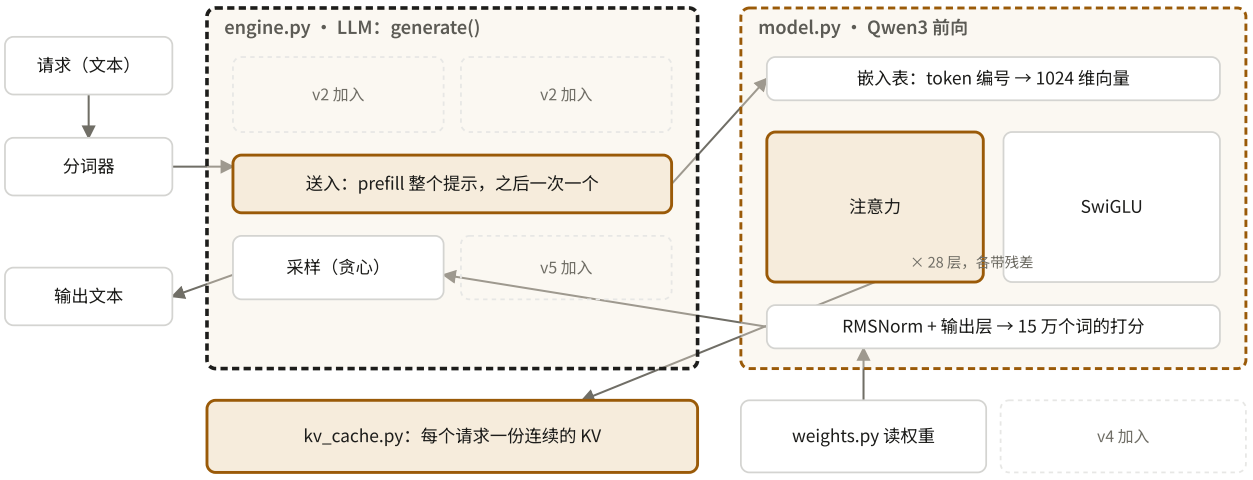
v0 每生成一个 token，都把整段序列从头算一遍。199 个 token 的提示生成 64 个 token，第 1 步 419 ms，第 64 步 724 ms，一共 34 秒；其中绝大部分计算是在重复上一步已经做过的事。

第 69 讲给出了解决办法：每层把算过的 K、V 存起来，新 token 只需要算自己的 q、k、v，再和存好的 K、V 做注意力。这一讲把它写进 miniinfer。改动不大：新增一个 18 行的 `kv_cache.py`，模型的注意力多两行，生成循环分出 prefill 和 decode 两种调用。

miniinfer 的架构

⇆ 对照

版本：v0 v1 v2 v3 v3+ v4 v5



引擎（engine.py）：一个请求一个请求地生成。v0 每步把整段序列送进模型；v1 先 prefill 整个提示，之后每步只送一个新 token（第 118、119 讲）。

v1 新增或改动（强调色）：engine.py 、送入、model.py 、注意力、kv_cache.py。点模块看说明。

图 119-1　v1 新增了 KV Cache（下方），改动了模型的注意力和引擎的生成循环。

为什么只存 K 和 V 就够了

先确认一件事：新 token 的计算到底依赖前面 token 的哪些东西?

沿着一层 Transformer 看。RMSNorm、q/k/v 投影、RoPE、SwiGLU 都是**逐位置**的：位置 p 的输出只依赖位置 p 的输入。只有注意力会跨位置：位置 p 的查询 q_p 要和所有 ⩽ p 位置的键 k_j 做点积，再对值 v_j 加权求和。而这些 k_j、v_j 一旦算出来就不会再变（因果掩码保证了后面的 token 不影响前面的计算）。

所以，位置 p 的新 token 需要的“历史”，恰好就是每一层前面所有位置的 K 和 V。把它们存下来，新 token 走完 28 层，每层只多读一次缓存。这就是 KV Cache，第 69 讲的推导这里不再重复。

数据结构：一块按最大长度分好的连续内存

最简单的存法：每个请求一开始就按“提示长度 + 最多生成多少个”分配好一整块，形状是 [层数, 最大长度, KV 头数, 头维度]，K、V 各一份。生成时往里写，用到哪里读到哪里。

Qwen3-0.6B 每个 token 的 KV 有多大？2（K 和 V）× 28 层 × 8 个 KV 头 × 128 维 × 4 字节（FP32）：

```
$ .venv/bin/python bench/kv_size.py
FP32: 每个 token 224 KiB；一个 2048 token 的请求 448 MiB
BF16: 每个 token 112 KiB；一个 2048 token 的请求 224 MiB
```

⇆ 对照

图 119-2　Qwen3-0.6B 的 KV Cache：FP32 下每个 token 224 KiB，一个 2048 token 的请求 448 MiB。GQA 让 KV 头只有查询头的一半，KV 也随之减半。

v1/kv_cache.py 共 18 行 ⇆ 对照

```python
"""v1: 一个请求一份连续的 KV Cache。按最大长度一次分配好，生成时往后追加。"""
import torch


class KVCache:
    def __init__(self, cfg, max_len, dtype=torch.float32, device="cpu"):
        shape = (cfg.num_layers, max_len, cfg.num_kv_heads, cfg.head_dim)
        self.k = torch.zeros(shape, dtype=dtype, device=device)
        self.v = torch.zeros(shape, dtype=dtype, device=device)
```

1 **一次分配好** 第 5-9 行

形状 [28, 最大长度, 8, 128]。第一维是层号，这样第 i 层读写时用 `self.k[i]` 取出这一层的那一片。

一开始就按最大长度分配，是因为生成过程中不知道每一步会不会结束；预先分好，写的时候就不用扩容和复制。代价是：请求提前结束时，后面没用到的空间也被占着。这个毛病在第 121 讲解决。

2 **写：新 token 的 K、V 放到它们的位置** 第 11-14 行

3 **读：前 end 个位置的全部 K、V** 第 16-18 行

← 上一步 | 看整个文件 | 下一步 →

模型：注意力多两行

注意力的改动只有两处：算出新 token 的 k、v 之后，**先写进缓存，再从缓存读出全部**，后面的计算照旧；掩码不再是 T × T 的下三角，而是“查询位置 p 只看键位置 ≤ p”。下面是和 v0 的差别：

v1/model.py 对比 v0/model.py：+10 行 / −7 行 ⇆ 对照

```
      ····· 省略 23 行没变的代码
24            self.cfg, self.w = cfg, w
25
26 +      def attention(self, i, x, pos, cache):
26 -      def attention(self, i, x, pos):
27            c, w, p = self.cfg, self.w, f"model.layers.{i}.self_attn."
28            T = x.shape[0]
      ····· 省略 4 行没变的代码
33            k = rms_norm(k, w[p + "k_norm.weight"], c.rms_eps)
34            q, k = rope(q, pos, c.rope_theta), rope(k, pos, c.rope_theta)
35 +          cache.write(i, pos, k, v)                                  # 新 token 的 K、V
36 +          end = int(pos[-1]) + 1
37 +          k, v = cache.read(i, end)                                  # 取出从第 0 个位置到
38            rep = c.num_heads // c.num_kv_heads                        # GQA：每组 K、V 给
39            k, v = k.repeat_interleave(rep, dim=1), v.repeat_interleave(rep, dim=1)
40            scores = torch.einsum("qhd,khd→hqk", q, k) / c.head_dim ** 0.5
41 +          mask = torch.arange(end, device=x.device)[None, :] ≤ pos[:, None]   # 位置
38 -          mask = torch.ones(T, T, dtype=torch.bool, device=x.device).tril()   # 因果掩
42            scores = scores.masked_fill(~mask, float("-inf"))
43            out = torch.einsum("hqk,khd→qhd", scores.softmax(-1), v).reshape(T, -1)
      ····· 省略 6 行没变的代码
50
51        @torch.no_grad()
52 +      def forward(self, tokens, start, cache):
53 +          # tokens: 这次新来的 [T] 个 token，从位置 start 开始；它们之前的 K、V 已经在 cach
49 -      def forward(self, tokens):
50 -          # tokens: [T] 的 token 编号。返回每个位置对下一个 token 的打分 logits: [T, vocab
54            c, w = self.cfg, self.w
55 +          pos = torch.arange(start, start + len(tokens), device=tokens.device)
52 -          pos = torch.arange(len(tokens), device=tokens.device)
56            x = w["model.embed_tokens.weight"][tokens]
57            for i in range(c.num_layers):
58                p = f"model.layers.{i}."
59 +              x = x + self.attention(i, rms_norm(x, w[p + "input_layernorm.weight"],
56 -              x = x + self.attention(i, rms_norm(x, w[p + "input_layernorm.weight"],
60                x = x + self.mlp(i, rms_norm(x, w[p + "post_attention_layernorm.weight"
61 +          x = rms_norm(x[-1:], w["model.norm.weight"], c.rms_eps)   # 只需要最后一个位置
58 -          x = rms_norm(x, w["model.norm.weight"], c.rms_eps)
62            return x @ w["lm_head.weight"].T
```

图 119-3　model.py 和 v0 的差别。绿底是新增，红底是删掉。注意力多了写缓存、读缓存，掩码改成按位置比较；forward 多了起始位置和缓存两个参数，最后只取一个位置的打分。

v1/model.py 第 26–62 行 / 共 62 行 ⇆ 对照

```
26      def attention(self, i, x, pos, cache):
27          c, w, p = self.cfg, self.w, f"model.layers.{i}.self_attn."
28          T = x.shape[0]
29          q = (x @ w[p + "q_proj.weight"].T).view(T, c.num_heads, c.head_dim)
30          k = (x @ w[p + "k_proj.weight"].T).view(T, c.num_kv_heads, c.head_dim)
31          v = (x @ w[p + "v_proj.weight"].T).view(T, c.num_kv_heads, c.head_dim)
32          q = rms_norm(q, w[p + "q_norm.weight"], c.rms_eps)        # Qwen3 在 RoPE 之
33          k = rms_norm(k, w[p + "k_norm.weight"], c.rms_eps)
34          q, k = rope(q, pos, c.rope_theta), rope(k, pos, c.rope_theta)
```

1 先写，再读 第 34–37 行

RoPE 之后的 k 和原始的 v 写进缓存。注意存的是**旋转之后**的 k：旋转只和它自己的位置有关，存旋转后的结果，以后就不用再转一次。

`end` 是这一批里最后一个 token 的位置加一。读出来的 k、v 是 [end, 8, 128]，包含了历史和这一次的新 token。

2 **掩码按位置比较** 第 40–43 行

3 **位置从 start 开始** 第 52–55 行

4 **只算最后一个位置的输出层** 第 61–62 行

← 上一步 | 看整个文件 | 下一步 →

引擎：prefill 一次，之后每步一个

v1/engine.py 第 10–28 行 / 共 36 行 ⇆ 对照

```
10 class LLM:
11     def __init__(self, model_dir, device="cpu", dtype=torch.float32):
12         self.cfg = Config.load(model_dir)
13         self.tok = Tokenizer.from_file(f"{model_dir}/tokenizer.json")
14         self.model = Qwen3(self.cfg, load_weights(model_dir, dtype, device))
15         self.device, self.dtype = device, dtype
16 
17     def generate(self, prompt, max_new_tokens=32):
18         ids = self.tok.encode(prompt).ids
```

1 **为这个请求分配缓存** 第 17–20 行

最大长度 = 提示长度 + 最多生成的数量。第一步要送进模型的是整个提示，这一步就是 **prefill**。

2 **一次 forward，一个新 token** 第 21–23 行

3 **之后每步只送一个** 第 24–27 行

← 上一步 | 看整个文件 | 下一步 →

先验证，再测速度

```
$ .venv/bin/python tests/check.py v1
一致: 'The capital of France is' → ' Paris. The capital of Italy is Rome. The capital of Spain is Madrid. The capital of China is Beijing. The capital of Japan is Tokyo. The capital'
一致: 'def fibonacci(n):' → '\n    if n == 0:\n        return 0\n    elif n == 1:\n return 1\n    else:\n        return fibonacci(n'
一致: '一个推理引擎由哪几部分组成？' → '它如何运作？  请提供一个例子，说明如何使用它来解决一个实际问题。\n答案:  问题：某公司需要为员工'
一致: '1, 2, 3, 4, 5,' → ' 6, 7, 8, 9, 10, 11, 12, 13, 14,'
v1: 4/4 条与 HuggingFace 逐 token 一致
```

图 119-4　v1 和 HuggingFace 逐 token 一致：缓存的读写、位置、掩码都对了。

再用和 v0 一样的设置比一比：199 个 token 的提示，生成 64 个，记录每一步 forward 的耗时。

```
$ .venv/bin/python bench/v0_vs_v1.py
v0: 提示 199 个 token
  第 1 步    419 ms，第 2 步    417 ms，第 64 步    724 ms，64 步共 34.12 s
v1: 提示 199 个 token
  第 1 步    357 ms，第 2 步     63 ms，第 64 步     66 ms，64 步共  4.55 s
```

图 119-5　v0 每步都在重算，第 64 步比第 1 步慢了七成；v1 只有第 1 步（prefill）慢，之后每步约 65 ms，64 步共 4.55 s，是 v0 的 7.5 分之一。

逐项看这组数：

- **第 1 步两者差不多**（419 ms 与 357 ms）：都是把 199 个 token 完整算一遍。v1 稍快，因为输出层只算了最后一行。
- **第 2 步起分道扬镳**：v0 还要算 200 个 token，417 ms；v1 只算 1 个 token，63 ms。
- **v1 的 decode 每步几乎不变**（63 → 66 ms）：每步的计算量只和 1 个 token 有关，注意力多读的 K、V 从 200 个涨到 263 个，在 CPU 上只占很小一部分。
- **v0 越来越慢**：第 64 步要算 263 个 token，724 ms。

一个 token 的 decode 要 63 ms，而 199 个 token 的 prefill 只要 357 ms，也就是每个 token 不到 2 ms。为什么 decode 一个 token 这么“贵”？因为无论算 1 个还是 199 个 token，每一步都要把全部权重（FP32 约 3 GB）从内存读一遍；token 越少，读一遍权重摊到每个 token 上的代价越大。这就是第 70 讲说的：decode 是读内存受限的。

要点

KV Cache 把“每步重算整段”变成“每步只算一个新 token”：v1 生成 64 个 token 从 34 s 降到 4.55 s。改动集中在注意力的两行（先写缓存，再读全部）和位置（从 start 开始）。剩下的问题是，decode 每一步都要把全部权重读一遍，却只为一个 token 服务。

回顾

v1 给每个请求按“提示 + 最大输出”分配一块连续的 KV Cache（FP32 下每个 token 224 KiB），注意力里先把新 token 的 K、V 写进去，再读出全部历史；生成循环分成一次 prefill 和之后每步一个 token 的 decode。和 HuggingFace 仍然逐 token 一致，64 个 token 从 34 s 降到 4.55 s，decode 每步稳定在 65 ms 左右。

可这 65 ms 里，绝大部分时间花在把 3 GB 权重读一遍，只为一个请求算出一个 token。如果同时有 8 个请求，能不能让它们共用这一遍读取？下一讲：**Continuous Batching**。

第120讲 Continuous Batching

重点

70 TTFT、TPOT 与吞吐　74 Continuous Batching　119 KV Cache

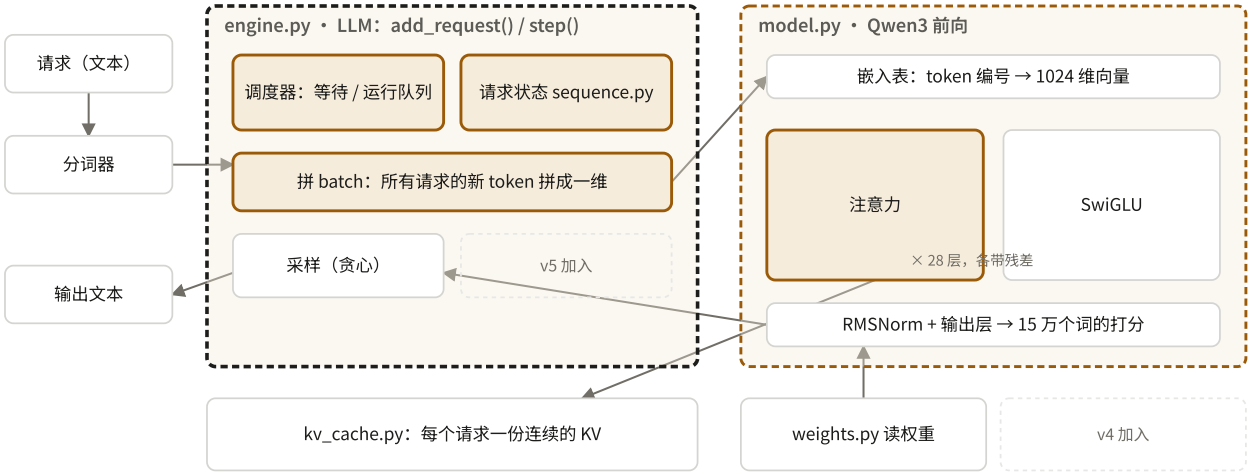
v1 的 decode 每步约 65 ms，几乎全花在把 3 GB 的权重从内存读一遍上，而这一遍读取只为一个请求算出一个 token。第 70 讲算过：decode 是读内存受限的，batch 里多放几个请求，读一遍权重就能同时为它们各算一个 token，每步的耗时增加得很少。

v1 的 `generate(prompt)` 一次只能接一个请求。这一讲把引擎改成真正的“服务”：请求随时进队列，引擎一步一步地跑，**每一步由调度器决定 batch 里有谁**；生成完的请求立刻离开，空出的位置马上给等着的请求，这就是第 74 讲的 continuous batching。

⇆ 对照

miniinfer 的架构

版本：v0 v1 v2 v3 v3+ v4 v5



引擎（engine.py）：请求先进队列，引擎一步一步地跑 `step()`：问调度器这一步 batch 里有谁，拼成一维的输入，调用模型，采样，更新每个请求的状态。第 120 讲起是这个样子。

v2 新增或改动（强调色）：engine.py 、调度器、请求状态 sequence.py、拼 batch、model.py 、注意力。点模块看说明。

图 120-1　v2 新增了请求状态（sequence.py）和调度器（scheduler.py），引擎从 generate() 一个循环改成 add_request() + step()，模型的注意力按请求分段计算。

设计：从“一个循环”到“一步一步”

v1 的结构是“一个请求从头跑到尾”。要同时服务多个请求，得把这个循环拆开：

- **请求要有自己的状态**。每个请求处在不同的阶段：有的还在排队，有的刚进来要做 prefill，有的已经生成了 50 个 token。每个请求自己记住 token 序列、已经算进缓存的位置数、状态和它的 KV Cache。这就是 `Sequence`。
- **每一步做什么由调度器决定**。调度器维护两个队列：等待中的和运行中的。每一步，运行中的请求各 decode 一个 token；batch 里还有空位，就从等待队列放新请求进来做 prefill。
- **引擎只负责“执行一步”**。`step()`：问调度器要这一步的 batch，把所有请求要算的 token 拼成一批送进模型，采样，更新每个请求，生成完的交给调度器移出。

模型这一侧的问题是：一个 batch 里的请求长度各不相同，prefill 的请求一次送进几十个 token，decode 的只有一个。常见的做法是**把所有请求这一步的 token 首尾相接拼成一维**，再记下每个请求占其中哪一段。线性层、RMSNorm、SwiGLU 都是逐 token 的，拼起来直接一次算完，batch 的收益就在这里；只有注意力要按段分开，每个请求只和自己的上下文做注意力。这种“拼成一维”的格式叫 varlen（变长），vLLM、nano-vllm 都是这样组织 batch 的。

用一个例子看拼出来的样子：请求 A 正在 decode，已有 20 个 token；请求 B 刚进来，提示 6 个 token。

```
tokens = [A 的第 20 个 token,  B 的 6 个提示 token]      共 7 个
pos    = [19,                  0, 1, 2, 3, 4, 5]
segs   = [(0, 1, A 的缓存),   (1, 7, B 的缓存)]          每个请求占 tokens 的哪一段
```

模型算完，A 取第 0 个位置的打分，B 取第 6 个位置的打分，各自得到下一个 token。

sequence.py：一个请求的全部状态

v2/sequence.py 共 30 行 ⇆ 对照

```python
"""一个请求在引擎里的全部状态。"""
from enum import Enum, auto
from itertools import count


class Status(Enum):
    WAITING = auto()     # 还没进 batch
    RUNNING = auto()     # 在 batch 里，每步生成一个 token
    FINISHED = auto()    # 生成完了

```

1 **三种状态** 第 6-9 行

等待（还没进 batch）、运行（在 batch 里，每步生成一个 token）、结束。下一版加上抢占后，运行中的请求还可能退回等待。

2 **要记的东西** 第 15-22 行

3 **这一步要送进模型的 token** 第 28-30 行

← 上一步 | 看整个文件 | 下一步 →

scheduler.py：每一步谁进 batch

v2/scheduler.py 共 46 行 ⇆ 对照

```python
"""v2: continuous batching 的调度器。每一步都重新决定 batch 里有谁。"""
from collections import deque
from sequence import Status


class Scheduler:
    def __init__(self, max_num_seqs=8, max_num_tokens=2048, static=False, kv_budget=
        self.max_num_seqs = max_num_seqs          # batch 里最多几个请求（同时占几份 KV Ca
        self.max_num_tokens = max_num_tokens      # 一步最多送进模型多少个 token（限制 pre
        self.static = static                      # True：静态批处理，等整批都结束才放新请求
```

1 几个上限 第 7-13 行

`max_num_seqs`：batch 里最多几个请求，也就是同时占着几份 KV Cache。`max_num_tokens`：一步最多送进模型多少个 token，防止一下子放进好几个长提示，让这一步变得很慢（第 77 讲的思路）。

`static` 打开时是静态批处理，用来对比。`kv_budget` 限制 KV 一共能放多少个位置，第 121 讲对比分页时用。

2 **先排运行中的请求** 第 18-21 行

3 **再放新请求进来** 第 22-36 行

4 **结束：立刻让出位置** 第 39-43 行

← 上一步 | 看整个文件 | 下一步 →

model.py：拼起来的 batch，分段的注意力

v2/model.py 对比 v1/model.py：+17 行 / −9 行 ⇆ 对照

```
      ····· 省略 23 行没变的代码
24            self.cfg, self.w = cfg, w
25
26 +      def attention(self, i, x, pos, segs):
27 +          # x: batch 里所有请求的新 token 拼在一起 [N, d]；segs: 每个请求占其中哪一段、用哪份
26 -      def attention(self, i, x, pos, cache):
28            c, w, p = self.cfg, self.w, f"model.layers.{i}.self_attn."
29            T = x.shape[0]
      ····· 省略 4 行没变的代码
34            k = rms_norm(k, w[p + "k_norm.weight"], c.rms_eps)
35            q, k = rope(q, pos, c.rope_theta), rope(k, pos, c.rope_theta)
36 +          out = torch.empty_like(q)
37 +          for a, b, cache in segs:                                  # 注意力按请求分开算
38 +              out[a:b] = self.attend(i, q[a:b], k[a:b], v[a:b], pos[a:b], cache)
39 +          return out.reshape(T, -1) @ w[p + "o_proj.weight"].T
40 +
41 +      def attend(self, i, q, k, v, pos, cache):
42 +          c = self.cfg
43            cache.write(i, pos, k, v)                                 # 新 token 的 K、V
44            end = int(pos[-1]) + 1
      ····· 省略 2 行没变的代码
47            k, v = k.repeat_interleave(rep, dim=1), v.repeat_interleave(rep, dim=1)
48            scores = torch.einsum("qhd,khd→hqk", q, k) / c.head_dim ** 0.5
49 +          mask = torch.arange(end, device=q.device)[None, :] ≤ pos[:, None]   # 位置
41 -          mask = torch.arange(end, device=x.device)[None, :] ≤ pos[:, None]   # 位置
50            scores = scores.masked_fill(~mask, float("-inf"))
51 +          return torch.einsum("hqk,khd→qhd", scores.softmax(-1), v)
43 -          out = torch.einsum("hqk,khd→qhd", scores.softmax(-1), v).reshape(T, -1)
44 -          return out @ w[p + "o_proj.weight"].T
52
53        def mlp(self, i, x):
      ····· 省略 3 行没变的代码
57
58        @torch.no_grad()
59 +      def forward(self, tokens, pos, segs):
60 +          # tokens、pos: batch 里所有请求的新 token 和它们的位置，拼成一维 [N]
61 +          # segs: [(起点, 终点, 缓存), ...]，第 j 个请求的 token 是 tokens[起点:终点]
52 -      def forward(self, tokens, start, cache):
53 -          # tokens: 这次新来的 [T] 个 token，从位置 start 开始；它们之前的 K、V 已经在 cach
62            c, w = self.cfg, self.w
55 -          pos = torch.arange(start, start + len(tokens), device=tokens.device)
63            x = w["model.embed_tokens.weight"][tokens]
64            for i in range(c.num_layers):
65                p = f"model.layers.{i}."
66 +              x = x + self.attention(i, rms_norm(x, w[p + "input_layernorm.weight"],
59 -              x = x + self.attention(i, rms_norm(x, w[p + "input_layernorm.weight"],
67                x = x + self.mlp(i, rms_norm(x, w[p + "post_attention_layernorm.weight"
68 +          last = torch.tensor([b - 1 for _, b, _ in segs], device=x.device)
```

```
69 +          x = rms_norm(x[last], w["model.norm.weight"], c.rms_eps)  # 每个请求只需要最后
61 -          x = rms_norm(x[-1:], w["model.norm.weight"], c.rms_eps)   # 只需要最后一个位置
70            return x @ w["lm_head.weight"].T
```

图 120-2　model.py 和 v1 的差别：forward 的参数从“起始位置 + 一份缓存”换成“每个 token 的位置 + 每个请求的段”；注意力拆成两部分，attention 做投影，attend 对一个请求做注意力；最后每个请求取自己最后一个位置的打分。

v2/model.py　第 26–52 行 / 共 70 行　　⇆ 对照

```
26      def attention(self, i, x, pos, segs):
27          # x: batch 里所有请求的新 token 拼在一起 [N, d]；segs: 每个请求占其中哪一段、用哪份缓
28          c, w, p = self.cfg, self.w, f"model.layers.{i}.self_attn."
29          T = x.shape[0]
30          q = (x @ w[p + "q_proj.weight"].T).view(T, c.num_heads, c.head_dim)
31          k = (x @ w[p + "k_proj.weight"].T).view(T, c.num_kv_heads, c.head_dim)
32          v = (x @ w[p + "v_proj.weight"].T).view(T, c.num_kv_heads, c.head_dim)
33          q = rms_norm(q, w[p + "q_norm.weight"], c.rms_eps)         # Qwen3 在 RoPE 之前
34          k = rms_norm(k, w[p + "k_norm.weight"], c.rms_eps)
```

1 **投影：整个 batch 一起算**　第 26–35 行

x 是 [N, 1024]，N 是这一步所有请求的新 token 加起来的数（上面的例子 N = 7）。q、k、v 投影、q/k 归一化、RoPE 都是逐 token 的，整个 batch 一次算完。RoPE 用的是每个 token 自己的位置 `pos`，所以 A 的 token 转 19 个位置的角度，B 的转 0–5。

2 **注意力：按请求分开**　第 36–39 行

3 **一个请求的注意力**　第 41–51 行

← 上一步　　看整个文件　　下一步 →

engine.py：add_request 和 step

v2/engine.py 第 12–58 行 / 共 58 行 ⇆ 对照

```
12 class LLM:
13     def __init__(self, model_dir, device="cpu", dtype=torch.float32, **sched_args):
14         self.cfg = Config.load(model_dir)
15         self.tok = Tokenizer.from_file(f"{model_dir}/tokenizer.json")
16         self.model = Qwen3(self.cfg, load_weights(model_dir, dtype, device))
17         self.device, self.dtype = device, dtype
18         self.scheduler = Scheduler(**sched_args)
19         self.log = []                                  # 每一步的 (prefill 请求数, d
20
21     def add_request(self, prompt_ids, max_new_tokens):
```

1 **加请求：只是放进队列** 第 21–24 行

`add_request` 建一个 Sequence 交给调度器就返回，不做任何计算。真正的服务里，网络请求随时到来，调用的就是这个函数。

2 **拼 batch** 第 26–36 行

3 **一次前向，所有请求** 第 39–41 行

4 **更新每个请求** 第 42–46 行

5 **generate：把请求都放进去，跑到没有活为止** 第 48–55 行

← 上一步 | 看整个文件 | 下一步 →

先验证

```
$ .venv/bin/python tests/check.py v2                                    ⇆ 对照
一致: 'The capital of France is' → ' Paris. The capital of Italy is Rome. The capital
of Spain is Madrid. The capital of China is Beijing. The capital of Japan is Tokyo.
The capital'
一致: 'def fibonacci(n):' → '\n    if n == 0:\n        return 0\n    elif n == 1:\n
return 1\n    else:\n        return fibonacci(n'
一致: '一个推理引擎由哪几部分组成？' → '它如何运作？  请提供一个例子，说明如何使用它来解决一个实际问
题。\n答案：  问题：某公司需要为员工'
一致: '1, 2, 3, 4, 5,' → ' 6, 7, 8, 9, 10, 11, 12, 13, 14,'
v2: 4/4 条与 HuggingFace 逐 token 一致
```

图 120-3　v2 一次放进 4 个请求一起跑，结果仍然和 HuggingFace 逐 token 一致。把 4 个请求拼成一批、分段做注意力，没有改变任何一个请求的计算。

这个检查有分量：4 个请求的提示长度各不相同，第一步是 4 个 prefill 拼在一起，之后是 4 个 decode 拼在一起。只要位置、段的边界、缓存的对应关系有一处错，就会有请求的输出不一样。

测一测：16 个请求

16 个请求同时到达，每个要生成的 token 数从 8 到 96 不等，一共 792 个 token。比较三种跑法：一个一个跑（`max_num_seqs=1`，相当于 v1）、静态批处理（8 个一批，整批结束才换下一批）、continuous batching（8 个槽，有空就补）。

```
$ .venv/bin/python bench/batching.py                                    ⇆ 对照
输出长度:  [64, 64, 8, 32, 96, 64, 64, 32, 64, 32, 96, 16, 96, 16, 32, 16] 共 792 个
token
一个一个跑:  54.7 s,  14.5 token/s, 792 步, 平均每步 1.0 个请求
静态批处理（8 个一批）:  35.3 s,  22.4 token/s, 192 步, 平均每步 4.1 个请求
continuous batching（8 个槽）:  28.8 s,  27.5 token/s, 160 步, 平均每步 5.0 个请求
```

图 120-4　同样 792 个 token：一个一个跑 792 步；静态批处理 192 步；continuous batching 160 步，吞吐是一个一个跑的 1.9 倍。

先看步数。一个一个跑，每步只出一个 token，要 792 步。静态批处理分两批，每批要跑到这批里最长的请求结束：两批里都有要生成 96 个 token 的请求，所以各要约 96 步，共 192 步。continuous batching 只要 160 步。

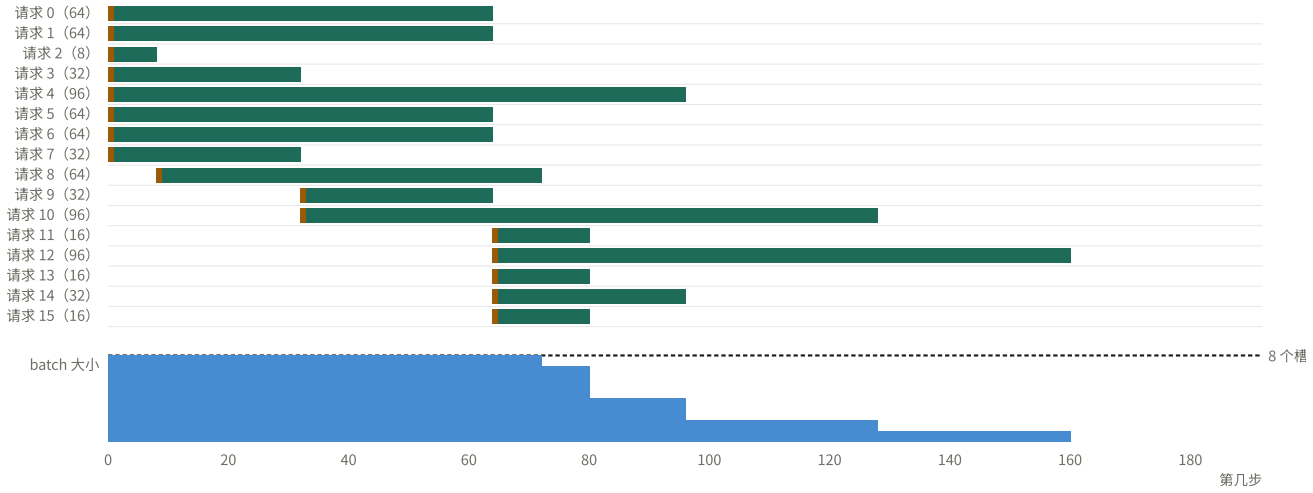
再看每一步的耗时：一个一个跑约 69 ms 一步，continuous batching 约 180 ms 一步，每步的请求平均 5 个。每步的请求是 5 倍，每步的耗时只是 2.6 倍，所以吞吐从 14.5 提高到 27.5 token/s。这就是第 70 讲的结论在 CPU 上的样子：多出来的请求共享了读权重的那一遍。

下面是两种批处理每一步 batch 里实际有谁（来自同一次运行的记录）：

每一步 batch 里有谁

⇆ 对照

运行： 静态批处理（8 个一批） continuous batching（8 个槽）



`continuous batching（8 个槽）：160 步，平均每步 5.0 个请求；8 个槽 × 160 步 = 1280 个“槽·步”，用了 792 个，空着 488 个（38%）。`

琥珀色：这一步做 prefill；绿色：decode。括号里是这个请求要生成的 token 数。数据来自引擎的实际运行记录。

图 120-5　静态批处理：第一批里短的请求早早结束，它们的槽一直空到这一批最长的请求（96 个 token）结束，才放第二批进来。continuous batching：请求 2（8 个 token）结束后，下一步请求 8 就补了进来。

切换两种运行对比：静态批处理 192 步 × 8 个槽里，空着 48%；continuous batching 空着 38%。continuous batching 还有不少空槽，是因为 16 个请求很快全都进了 batch，最后只剩几个长请求在跑，没有新请求可以补。真实的服务里请求源源不断地到来，这部分空槽会被填上。

要点

continuous batching 的关键是“每一步重新决定 batch”：运行中的请求各 decode 一个 token，有空位就放新请求进来 prefill，生成完的立刻离开。模型这边把所有请求的新 token 拼成一维，线性层一次算完，只有注意力按请求分段。

还剩什么问题

v2 给每个请求按“提示 + 最大输出”预留一整份 KV Cache。KV 的空间有限时，能同时运行的请求数就由这个“最大值”决定，而不是由实际用了多少决定。比如每个请求声明最多生成 512 个 token、实际只生成了 50 个，预留的空间九成是空的，却挡住了后面的请求。下一讲的压测会看到，KV 空间紧张时，这个毛病直接限制了吞吐。

回顾

v2 把引擎拆成三部分：Sequence 记每个请求的状态（关键是 `num_computed`），Scheduler 每一步决定 batch（运行中的先 decode，有空位再放新请求 prefill），`step()` 把这一步所有请求的 token 拼成一维送进模型，注意力按请求分段。和 HuggingFace 仍然逐 token 一致。16 个请求、792 个 token，continuous batching 用 160 步、27.5 token/s，是一个一个跑的 1.9 倍，比静态批处理少用 32 步。

每个请求按最大长度预留 KV，空间不够时放不下几个请求；而且这块空间必须是连续的。下一讲：**分页 KV 管理**。

第121讲 分页 KV 管理

重点

75 PagedAttention　78 调度与抢占　120 Continuous Batching

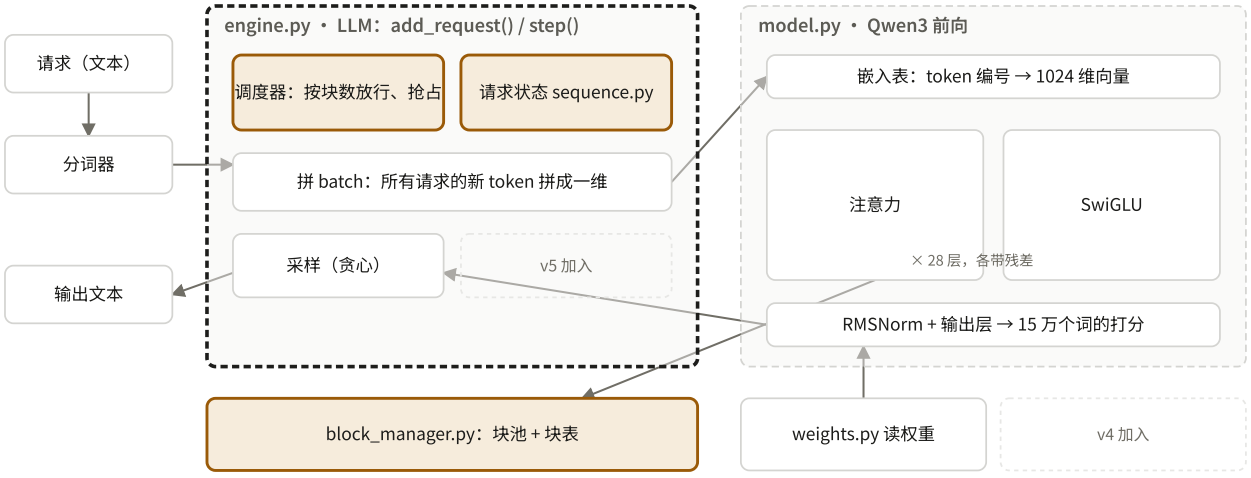
v2 给每个请求按“提示 + 最大输出”预留一整块连续的 KV Cache。请求刚进来时，它只用了提示那一小段，后面的空间要等它慢慢生成才用得上；可这块空间从一开始就被占住了，别的请求用不了。KV 的空间一紧张，能同时跑的请求数就由“最大长度”而不是“实际用了多少”决定。

第 75 讲的 PagedAttention 给出了办法：把 KV 的空间切成固定大小的块，所有请求共用一个块池；每个请求用一张**块表**记下自己的 KV 放在哪些块里，用到哪里拿到哪里。这一讲把它写进 miniinfer，顺带处理分页带来的新问题：块用完了怎么办。

miniinfer 的架构

⇆ 对照

版本：v0 v1 v2 v3 v3+ v4 v5



引擎（engine.py）：请求先进队列，引擎一步一步地跑 step()：问调度器这一步 batch 里有谁，拼成一维的输入，调用模型，采样，更新每个请求的状态。第 120 讲起是这个样子。

v3 新增或改动（强调色）：调度器、请求状态 sequence.py、block_manager.py。点模块看说明。

图 121-1　v3 用 block_manager.py 取代了 kv_cache.py；调度器改成按空闲块数放行，块不够时抢占。模型的代码一行都没改。

设计：块池、块表、和一个“看起来连续”的视图

三样东西：

- **块池**（KVPool）：一开始就分配好一大块内存，形状 [层数, 块数, 块大小, KV 头数, 头维度]，K、V 各一份。块大小取 16 个位置，Qwen3-0.6B 在 FP32 下一个块（28 层合计）是 16 × 224 KiB = 3.5 MiB。
- **块表**：每个请求一个整数列表。第 j 项是“我的第 j 个 16 位置的段，放在池子里的第几块”。位置 p 在第 `table[p // 16]` 块的第 `p % 16` 格。
- **块管理器**（BlockManager）：一个空闲块的列表。请求进来时按提示长度拿块；decode 时位置跨进新块，再拿一块；请求结束，块还回来。

用一个例子看块表：块大小 16，请求 A 有 20 个 token，请求 B 有 40 个。

```
块池：   [ 0 ][ 1 ][ 2 ][ 3 ][ 4 ][ 5 ][ 6 ][ 7 ] …
          A    B    A    B    B
A 的块表 = [0, 2]       位置 0-15 在块 0，16-19 在块 2 的前 4 格
B 的块表 = [1, 3, 4]    位置 0-15、16-31、32-39
```

A 的两个块在池子里并不相邻，中间夹着 B 的块，这正是分页的意义：不需要连续的空间。

模型怎么办？注意力要的是“从位置 0 到现在的全部 K、V”。我们给每个请求包一个小对象 `SeqKV`，它和 v1 的 `KVCache` **接口完全一样**（`write(layer, pos, k, v)` 和 `read(layer, end)`），内部按块表把位置换算成块和格子。这样 model.py 一行都不用改：模型只管调用 write 和 read，不知道背后是一整块还是很多小块。

block_manager.py：分页的部分

这个文件里和前缀缓存有关的部分（allocate 里算哈希、查表的几行，以及 register）留到下一讲，这一讲只看分页本身。

v3/block_manager.py 共 119 行 ⇆ 对照

```python
"""v3：分页的 KV Cache。所有请求共用一个块池，每个请求用一张块表记下自己的 KV 在哪些块里。"""
import torch



class KVPool:
    """一大块预先分配好的显存，切成 num_blocks 个块，每块放 block_size 个位置的 K、V。"""

    def __init__(self, cfg, num_blocks, block_size, dtype=torch.float32, device="cpu
        shape = (cfg.num_layers, num_blocks, block_size, cfg.num_kv_heads, cfg.head_
        self.k = torch.zeros(shape, dtype=dtype, device=device)
```

1 **块池：一次分配好** 第 5-12 行

[28, 块数, 16, 8, 128]，K、V 各一份。压测里用 20 块（320 个位置），只有 70 MiB；真实服务会把扣掉权重之后的显存几乎全部分给块池（第 99 讲的显存账本）。

2 **SeqKV：位置换算成块和格子** 第 15-25 行

3 **写：每个位置写到它的格子** 第 27-30 行

4 **读：按块表取出来，拼成连续的** 第 32-39 行

5 **块管理器的状态** 第 42-54 行

6 **拿一块** 第 56-65 行

7 **请求进来：为它要算的 token 分配块** 第 67-87 行

8 **decode 前：要进新块就再拿一块** 第 89-96 行

9 **还块** 第 113-119 行

← 上一步 | 看整个文件 | 下一步 →

scheduler.py：按块放行，块不够就抢占

分页之后，调度器要回答的问题变了：v2 问的是“KV 空间够不够预留一整份”，现在问的是“空闲块够不够”。还多了一种情况：请求进来时块是够的，可它一边生成一边要新块，几个请求一起长，池子可能在半路用完。这时要有人让出来，这就是第 78 讲的**抢占**。

v3/scheduler.py 对比 v2/scheduler.py：+37 行 / −20 行 ⇆ 对照

```
+ """v3：调度器按块的数量决定放谁进来；块不够时，把最晚进来的请求抢占回等待队列（之后重算）。"""
- """v2：continuous batching 的调度器。每一步都重新决定 batch 里有谁。"""
  from collections import deque
  from sequence import Status
      …… 省略 1 行没变的代码

  class Scheduler:
+     def __init__(self, block_manager, max_num_seqs=8, max_num_tokens=2048):
+         self.bm = block_manager
+         self.max_num_seqs = max_num_seqs
+         self.max_num_tokens = max_num_tokens
-     def __init__(self, max_num_seqs=8, max_num_tokens=2048, static=False, kv_budge
-         self.max_num_seqs = max_num_seqs        # batch 里最多几个请求（同时占几份 KV
-         self.max_num_tokens = max_num_tokens    # 一步最多送进模型多少个 token（限制 pr
-         self.static = static                    # True：静态批处理，等整批都结束才放新请
-         self.kv_budget = kv_budget              # KV Cache 最多能放多少个位置（None 表
-         self.kv_reserved = 0                    # 已经分出去的位置数：每个请求按“提示 +
          self.waiting, self.running = deque(), []
+         self.num_preempted = 0
+         self.num_cached = 0                         # 一共有多少个位置是前缀缓存命中、没

      def add(self, seq):
      …… 省略 1 行没变的代码

      def schedule(self):
+         # 1. 正在运行的请求各 decode 一个 token；下一个位置要进新块时先拿块，拿不到就抢占
+         batch = []
+         for seq in list(self.running):
+             if seq not in self.running:             # 已经在本轮被抢占了
+                 continue
+             while not self.bm.append_slot(seq):
+                 victim = self.running[-1]           # 抢占最晚进来的那个，把它的块让出来
+                 self.preempt(victim)
+                 if victim is seq:
+                     break
+             if seq in self.running:
+                 batch.append(seq)
-         # 1. 正在运行的请求，这一步各 decode 一个 token
-         batch = list(self.running)
          budget = self.max_num_tokens - len(batch)
+         # 2. 有空位、有空闲块，就从等待队列放新请求进来做 prefill
+         while self.waiting and len(batch) < self.max_num_seqs:
-         # 2. 有空位就从等待队列里放新请求进来，这一步做它们的 prefill
-         can_admit = not (self.static and self.running)
-         while can_admit and self.waiting and len(batch) < self.max_num_seqs:
              seq = self.waiting[0]
+             need = self.bm.blocks_needed(len(seq.tokens))
+             if len(seq.tokens) > budget or need > self.bm.num_free():
```

```
-             if seq.num_prompt > budget:          # 这一步的 token 预算不够了，下一步再说
                  break
-             need = seq.num_prompt + seq.max_new_tokens
-             if self.kv_budget is not None and self.kv_reserved + need > self.kv_bu
-                 break                            # KV Cache 的空间不够预留一整份了
-             self.kv_reserved += need
              self.waiting.popleft()
+             seq.num_computed = self.bm.allocate(seq)  # 前缀缓存命中的部分不用再算
+             self.num_cached += seq.num_computed
              seq.status = Status.RUNNING
              self.running.append(seq)
              batch.append(seq)
+             budget -= len(seq.tokens) - seq.num_computed
-             budget -= seq.num_prompt
          return batch

+     def preempt(self, seq):
+         # 抢占：释放它所有的块，放回等待队列最前面；它已经生成的 token 留着，下次连同提示一起重
+         self.running.remove(seq)
+         self.bm.release(seq)
+         seq.num_computed, seq.status = 0, Status.WAITING
+         seq.cache = None
+         self.waiting.appendleft(seq)
+         self.num_preempted += 1
+
      def finish(self, seq):
          seq.status = Status.FINISHED
          self.running.remove(seq)
+         self.bm.release(seq)
+         seq.cache = None
-         seq.cache = None                         # 释放这个请求的 KV Cache
-         self.kv_reserved -= seq.num_prompt + seq.max_new_tokens

      def has_work(self):
  …… 省略 1 行没变的代码
```

图 121-2　scheduler.py 和 v2 的差别：放行条件从“预留得下一整份”换成“空闲块够放下这次要算的 token”；运行中的请求每步先保证有块可写，拿不到就抢占；新增 preempt。

v3/scheduler.py 共 63 行 ⇆ 对照

```python
"""v3: 调度器按块的数量决定放谁进来；块不够时，把最晚进来的请求抢占回等待队列（之后重算）。"""
from collections import deque
from sequence import Status


class Scheduler:
    def __init__(self, block_manager, max_num_seqs=8, max_num_tokens=2048):
        self.bm = block_manager
        self.max_num_seqs = max_num_seqs
```

1 运行中的请求：先保证有地方写 第 18-30 行

对每个运行中的请求，先调用 `append_slot` 保证下一个位置有块可写。拿不到，就抢占运行队列里**最晚进来**的请求，把它的块还回池子，再试一次。

如果最晚进来的就是自己，那就抢占自己，这一步不再给它排 decode。最先进来的请求永远不会被抢占，所以总有请求在往前走，不会出现大家互相抢、谁都完不成的局面。

2 **新请求：空闲块够才放进来** 第 31-45 行

3 **抢占：还块，回去重排** 第 47-54 行

← 上一步 | 看整个文件 | 下一步 →

引擎的改动只有几行：建块池和块管理器，请求进 batch 时包一个 SeqKV 交给模型。

v3/engine.py 对比 v2/engine.py：+11 行 / −7 行 ⇆ 对照

```
+ """v3：分页的 KV Cache。KV 存在一个共用的块池里，请求按需拿块；还可以打开前缀缓存。"""
- """v2：continuous batching。请求先进队列，引擎一步一步地跑，每一步由调度器决定 batch 里有谁。
  import torch
  from tokenizers import Tokenizer
      …… 省略 1 行没变的代码
  from weights import load_weights
  from model import Qwen3
+ from block_manager import KVPool, SeqKV, BlockManager
- from kv_cache import KVCache
  from sequence import Sequence
  from scheduler import Scheduler
      …… 省略 1 行没变的代码

  class LLM:
+     def __init__(self, model_dir, device="cpu", dtype=torch.float32,
+                  num_blocks=256, block_size=16, prefix_cache=False, **sched_args):
-     def __init__(self, model_dir, device="cpu", dtype=torch.float32, **sched_args)
          self.cfg = Config.load(model_dir)
          self.tok = Tokenizer.from_file(f"{model_dir}/tokenizer.json")
          self.model = Qwen3(self.cfg, load_weights(model_dir, dtype, device))
          self.device, self.dtype = device, dtype
+         self.pool = KVPool(self.cfg, num_blocks, block_size, dtype, device)
+         self.bm = BlockManager(num_blocks, block_size, prefix_cache)
+         self.scheduler = Scheduler(self.bm, **sched_args)
-         self.scheduler = Scheduler(**sched_args)
          self.log = []                                    # 每一步的 (prefill 请求数,

      …… 省略 8 行没变的代码
          tokens, pos, segs = [], [], []
          for seq in batch:
+             if seq.cache is None:                        # 刚进 batch：块已经由调度器
+                 seq.cache = SeqKV(self.pool, seq.block_table)
-             if seq.cache is None:                        # 刚进 batch：分配一份按最大
-                 seq.cache = KVCache(self.cfg, seq.num_prompt + seq.max_new_tokens,
              new = seq.new_tokens()
              segs.append((len(tokens), len(tokens) + len(new), seq.cache))
              pos += range(seq.num_computed, seq.num_computed + len(new))
              tokens += new
+         n_prefill = sum(1 for s in batch if len(s.new_tokens()) > 1)
-         n_prefill = sum(1 for s in batch if s.num_computed == 0)
          self.log.append((n_prefill, len(batch) - n_prefill))
          logits = self.model.forward(torch.tensor(tokens, device=self.device),
      …… 省略 2 行没变的代码
          for seq, t in zip(batch, nxt):
              seq.num_computed = len(seq.tokens)           # 这一步送进去的 token 都已经
+             self.bm.register(seq)                        # 新装满的块登记进前缀缓存
              seq.tokens.append(t)
              if t == self.cfg.eos_token_id or seq.num_generated >= seq.max_new_toker
```

```
…… 省略 13 行没变的代码
```

图 121-3　engine.py 和 v2 的差别。分配 KV 从引擎挪到了调度器（分配块），引擎只把块表包成 SeqKV；每步算完多一行登记前缀缓存（下一讲）。

先验证，再故意把池子开小

```
$ .venv/bin/python tests/check.py v3
一致: 'The capital of France is' → ' Paris. The capital of Italy is Rome. The capital of Spain is Madrid. The capital of China is Beijing. The capital of Japan is Tokyo. The capital'
一致: 'def fibonacci(n):' → '\n    if n == 0:\n        return 0\n    elif n == 1:\n return 1\n    else:\n        return fibonacci(n'
一致: '一个推理引擎由哪几部分组成？' → '它如何运作？　请提供一个例子，说明如何使用它来解决一个实际问题。\n答案：　问题：某公司需要为员工'
一致: '1, 2, 3, 4, 5,' → ' 6, 7, 8, 9, 10, 11, 12, 13, 14,'
v3: 4/4 条与 HuggingFace 逐 token 一致
```

图 121-4　v3 和 HuggingFace 逐 token 一致。

这个检查用的块池很大，一次抢占都没发生。抢占的代码路径也要验证：把块池开到只有 6 块（96 个位置），4 个请求各要 40 个左右的位置，一定会有人被抢占、重算：

```
$ .venv/bin/python tests/check.py v3 "{'num_blocks': 6, 'prefix_cache': True}"
一致: 'The capital of France is' → ' Paris. The capital of Italy is Rome. The capital of Spain is Madrid. The capital of China is Beijing. The capital of Japan is Tokyo. The capital'
一致: 'def fibonacci(n):' → '\n    if n == 0:\n        return 0\n    elif n == 1:\n return 1\n    else:\n        return fibonacci(n'
一致: '一个推理引擎由哪几部分组成？' → '它如何运作？　请提供一个例子，说明如何使用它来解决一个实际问题。\n答案：　问题：某公司需要为员工'
一致: '1, 2, 3, 4, 5,' → ' 6, 7, 8, 9, 10, 11, 12, 13, 14,'
运行中发生了 2 次抢占
v3: 4/4 条与 HuggingFace 逐 token 一致
```

图 121-5　块池只有 6 块时，运行中发生了 2 次抢占，被抢占的请求重算之后接着生成，4 条输出仍然和 HuggingFace 逐 token 一致。

被抢占的请求要把提示和已经生成的 token 一起重算一遍，算出来的 K、V 和被扔掉的那份完全一样，所以输出不受影响。这也说明“重算”在正确性上是安全的：KV 只是一种缓存，扔掉了总可以从 token 重新算出来。

测一测：同样只有 320 个位置

还是第 120 讲的 16 个请求，这次把 KV 的空间限制在 320 个位置（20 个块）。v2 按“提示 + 最大输出”整份预留；v3 按块按需分配。

```
$ .venv/bin/python bench/paging.py                                  ⇆ 对照

v2 整份预留:  46.5 s,  17.0 token/s, 240 步, 平均每步 3.3 个请求
v3 按块分配:  34.7 s,  22.8 token/s, 167 步, 平均每步 4.7 个请求, 抢占 9 次
```

图 121-6　KV 空间只有 320 个位置时：v2 平均每步只能放 3.3 个请求，240 步、17.0 token/s；v3 平均每步 4.7 个，167 步、22.8 token/s，期间抢占 9 次。

v2 每个请求要预留“十一二个 token 的提示 + 8 到 96 个输出”这么多位置，320 个位置只够同时放 3 到 4 个请求，8 个槽有一半空着。v3 进来时只要提示的 1 个块，8 个槽很快坐满；它们一起往下生成，块用完了就抢占最晚进来的，等有空再重算。抢占了 9 次，多做了 9 次 prefill，但每步多跑的 1.4 个请求远远抵得过，吞吐提高了 34%。

如果 KV 的空间足够大（比如第 120 讲那样不设限），两者就没有区别了：v2 预留的空间本来就够用。分页的价值，在 KV 空间成为瓶颈时才显现出来，而真实的服务几乎总是如此：显存扣掉权重之后剩下的那部分，决定了能同时服务多少个请求。

要点

分页把“每个请求一整块连续的 KV”换成“共用的块池 + 每个请求一张块表”。请求只按现在要算的 token 拿块，decode 每跨进一个新块再拿一块，结束时还回去。代价是池子可能在半路用完，需要抢占：最晚进来的请求还回所有的块，回到队首，之后连同已生成的 token 一起重算。

回顾

v3 新增 block_manager.py：KVPool 是一大块按 [层, 块, 16, 头, 维] 分好的内存；BlockManager 管空闲块和引用计数；SeqKV 按块表把位置换算成块和格子，接口和 v1 的 KVCache 一样，所以模型不用改。调度器按空闲块数放行，运行中的请求拿不到新块时抢占最晚进来的那个，被抢占的请求回到队首重算。和 HuggingFace 仍然逐 token 一致，块池只有 6 块、发生抢占时也一样。KV 只有 320 个位置时，吞吐从 v2 的 17.0 提高到 22.8 token/s。

块表还带来一个新的可能：两个请求的提示开头一样（比如同一段系统提示），它们的前几个块里存的 K、V 也一模一样。能不能让第二个请求直接用第一个请求的块？下一讲：**前缀缓存**。

第122讲 前缀缓存

76 Prefix Cache　121 分页 KV 管理

真实的服务里，大量请求的开头是一样的：同一个应用的系统提示，同一段多轮对话的历史，同一份被反复提问的文档。v3 的每个请求都把自己的提示从头算一遍，哪怕前面 300 个 token 和上一个请求一字不差。

第 76 讲讲过前缀缓存的原理：KV 只取决于“这个位置之前的全部 token”，开头相同，前几个块里的 K、V 就一模一样，后来的请求可以直接用。有了 v3 的块表，这件事变得很自然：**让后来的请求的块表，直接指向已经算好的块**。这一讲在 block_manager.py 里加上它，大约 30 行。

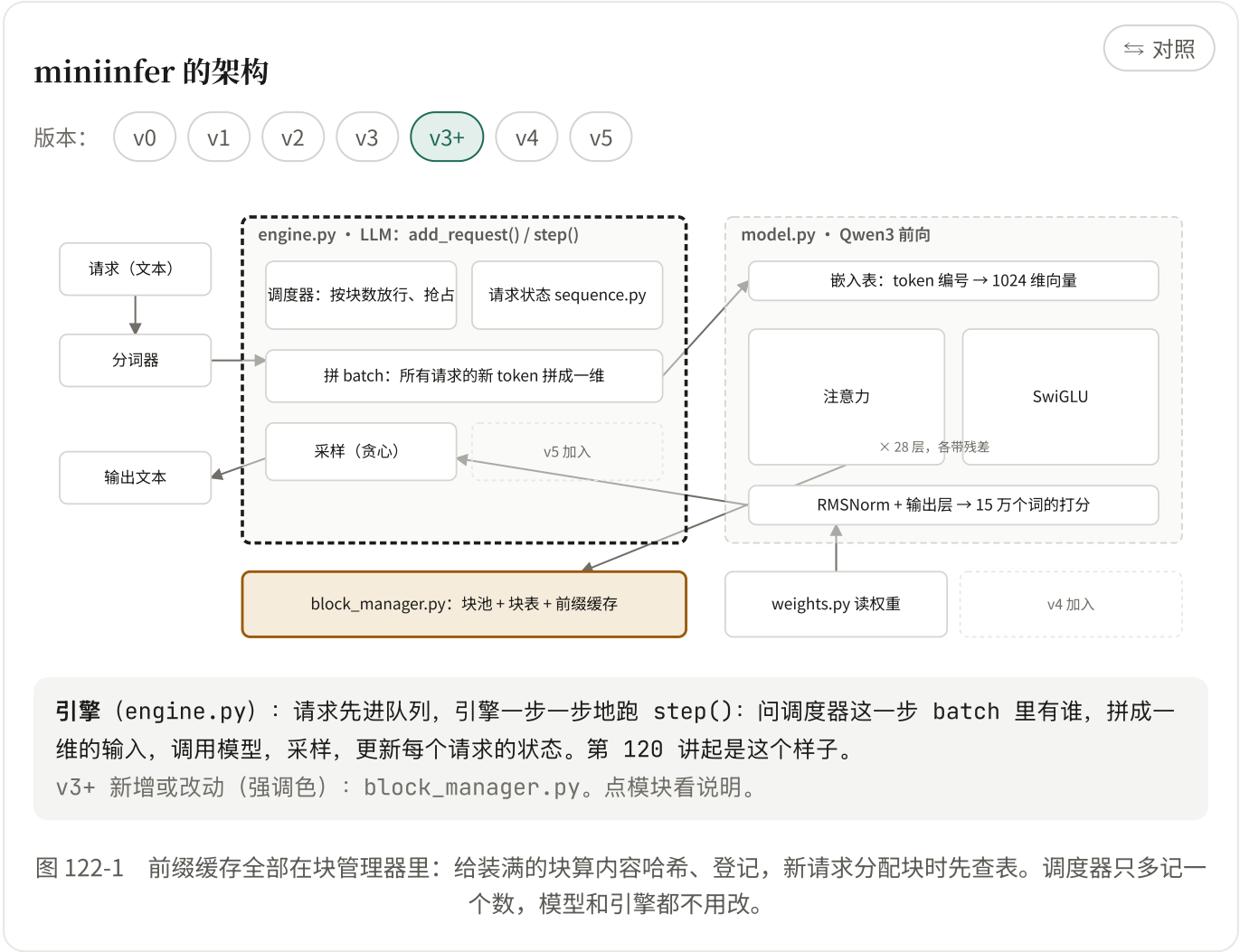


图 122-1　前缀缓存全部在块管理器里：给装满的块算内容哈希、登记，新请求分配块时先查表。调度器只多记一个数，模型和引擎都不用改。

设计：怎么认出“同一个块”

要复用一个块，得能快速判断“这个块里存的，就是我要的 KV”。直接比 KV 的数值太慢，我们比它的**来源**：一个块的 KV 由“它之前的全部 token + 它自己的 16 个 token”唯一决定。

所以给每个装满的块算一个哈希：

```
h(第 0 块) = hash(无,          第 0 块的 16 个 token)
h(第 1 块) = hash(h(第 0 块),  第 1 块的 16 个 token)
h(第 2 块) = hash(h(第 1 块),  第 2 块的 16 个 token)
```

每一块的哈希都把前一块的哈希算进去，链式地连到开头。这样两个块哈希相同，就说明它们的整个前缀都相同，而不只是这 16 个 token 碰巧一样（第 76 讲的链式块哈希）。

块管理器维护一张表：哈希 → 块号。剩下的规则：

1. **只登记装满的块**。没装满的块，后面还会写入，内容没定下来。
2. **算完才登记**。块的 K、V 要真正写进去之后，才能给别人用；所以登记放在每一步 forward 之后。
3. **新请求从头逐块查**，连续命中就继续，第一次没命中就停：前缀缓存只能从开头连续地复用。
4. **一个块可以被多个请求共用**，用引用计数记有几个人在用。
5. **没人用的块，内容先留着**。它回到空闲列表，但哈希表里的登记不删，后来的请求还能命中；只有它被拿去装别的内容时，登记才作废。这让空闲列表自然地变成了一个按释放先后淘汰的缓存。
6. **不复用含最后一个 token 的块**。新请求总得算出最后一个位置的打分，才能得到下一个 token；如果它的提示全部命中，最后一个 token 也就没有被“算”过。所以最后一个 token 所在的块总是自己算。

block_manager.py：前缀缓存的部分

v3/block_manager.py 第 42–119 行 / 共 119 行 ⇆ 对照

```
42  class BlockManager:
43      """管块的分配和回收。前缀缓存（第 122 讲）也在这里：内容相同的整块可以被多个请求共用。"""
44
45      def __init__(self, num_blocks, block_size, prefix_cache=False):
46          self.block_size = block_size
47          self.free = list(range(num_blocks))          # 空闲块的编号
48          self.ref = [0] * num_blocks                  # 每个块被几个请求用着
49          self.prefix_cache = prefix_cache
50          self.hash_of = [None] * num_blocks           # 块里存的是哪段内容（只记装满的块）
51          self.block_of = {}                           # 内容的哈希 → 块编号
```

1 两张表 第 45–51 行

`block_of`：哈希 → 块号，用来查“这段前缀算过没有”。`hash_of`：块号 → 哈希，用来在块被重新使用时，把它的旧登记删掉。`prefix_cache` 开关默认关闭，方便对比。

2 **拿块时作废旧内容** 第 56–62 行

3 **逐块算链式哈希** 第 70–75 行

4 **命中：直接用别人的块** 第 76–82 行

5 **没命中：拿新块** 第 83–87 行

6 **算完之后登记** 第 98–111 行

7 **还块：计数归零也不删内容** 第 113–119 行

← 上一步 | 看整个文件 | 下一步 →

调度器只多了一行统计，引擎在每步之后多调用一次 `register`，上一讲的差异图里已经能看到。

先验证

前缀缓存最怕的是“命中了不该命中的块”：两个请求只是某一块的 16 个 token 相同、前面不同，却共用了块，后面的请求就会读到别人的 KV，悄悄地生成不同的结果。上一讲的小块池检查已经打开了前缀缓存；下面的压测还会比较打开和关闭时的输出。

测一测：共用系统提示的 8 个请求

8 个请求共用一段很长的系统提示（约 330 个 token），后面各接一个不同的问题。先跑第 1 个，它的提示块留在块池里；再一起跑后 7 个，比较前缀缓存打开和关闭。

```
$ .venv/bin/python bench/prefix.py                                  ⇆ 对照

前缀缓存关闭：后 7 个请求的提示共 2356 个 token，命中 0 个，实际 prefill 2356 个，用时 5.5 s
前缀缓存打开：后 7 个请求的提示共 2356 个 token，命中 2240 个，实际 prefill 116 个，用时 2.4 s
两种情况的输出完全一致： True
```

图 122-2　后 7 个请求的提示共 2356 个 token。打开前缀缓存后，2240 个位置直接命中，只 prefill 了 116 个，用时从 5.5 s 降到 2.4 s；两种情况的输出完全一致。

逐项看：

- **命中 2240 个位置** = 7 个请求 × 20 个块 × 16。系统提示加上 “Question: ” 共约 330 个 token，前 20 个满块（320 个位置）对所有请求都一样，全部命中。
- **只 prefill 116 个 token**：每个请求剩下的十几个 token（第 21 块里的系统提示尾巴和它自己的问题）。
- **时间少了一半多，而不是少 95%**：prefill 省掉了，但每个请求还要 decode 8 个 token，这部分时间没变；而且命中的请求在 decode 时，注意力仍要读完整的 330 多个位置的 K、V。
- **输出完全一致**：共用的块里存的 K、V，和自己算出来的一模一样。

第 76 讲提过的多轮对话也是同样的道理：第二轮的输入是“第一轮的全部 + 新问题”，第一轮结束时它的块都还在池子里（引用计数归零，但登记还在），第二轮进来就能命中。

要点

前缀缓存建在块表之上：给装满、算完的块按“前一块的哈希 + 本块的 token”算链式哈希并登记；新请求从头逐块查，连续命中的块直接放进自己的块表、引用计数加一，这些位置不用再算。没人用的块内容先留着，被重新拿去用时才作废登记。

回顾

前缀缓存全部写在块管理器里：链式块哈希保证“哈希相同 = 整个前缀相同”；只登记装满并且已经算完的块；新请求从头逐块查，连续命中就复用，最后一个 token 所在的块总是自己算；引用计数让一个块被多个请求共用，归零的块保留内容、被覆盖时才作废。8 个共用系统提示的请求，后 7 个的 prefill 从 2356 个 token 降到 116 个，用时从 5.5 s 降到 2.4 s，输出完全一致。

到这里，KV 的组织已经比较完整了。剩下最大的一块内存是权重：FP32 下线性层就有 1.6 GB 多。能不能用更少的位数存权重？下一讲：**权重量化**。

第123讲 权重量化

90 量化基础　91 权重量化：GPTQ 与 AWQ　122 前缀缓存

到 v3 为止，miniinfer 的权重全是 FP32：28 层的线性层加起来 1680 MiB，再加上嵌入表和输出层（各 593 MiB），约 2.8 GiB。在 GPU 上，权重占的显存越多，留给 KV 块池的就越少，能同时服务的请求也越少（第 99 讲）；decode 每一步又要把权重读一遍，权重越大，每一步越慢（第 70 讲）。

第 90 讲讲过量化：用更少的位数存权重，计算时再换算回来。这一讲实现最常见的一种：**只量化权重的 INT8，每个输出通道一个 scale**（权重 8 位、激活保持高精度，常写作 W8A16）。新增一个 35 行的 `quant.py`，模型里把 `x @ W.T` 换成一个 `linear(x, W)` 函数。

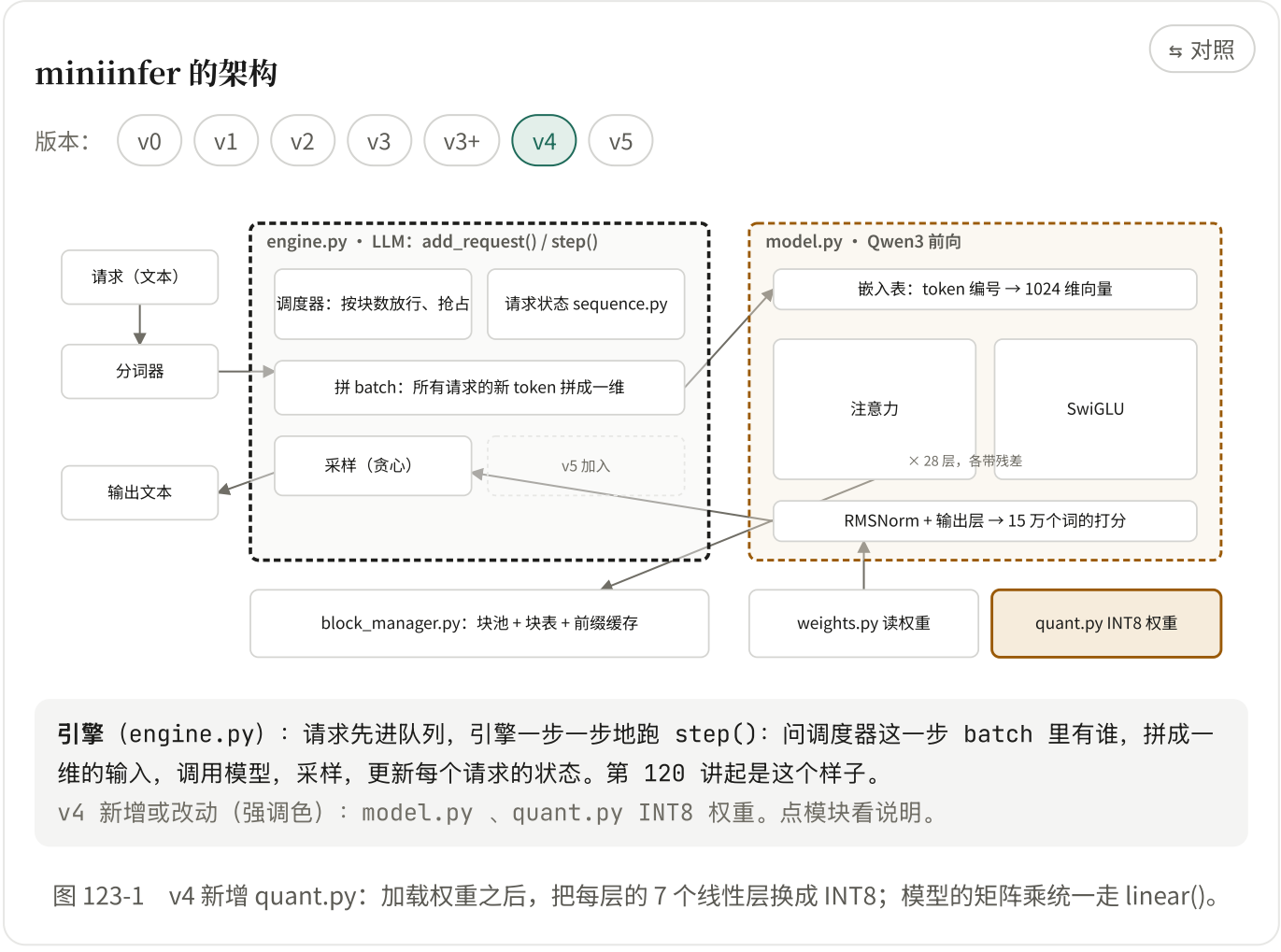


图 123-1　v4 新增 quant.py：加载权重之后，把每层的 7 个线性层换成 INT8；模型的矩阵乘统一走 linear()。

设计：量化什么，怎么量化

量化哪些权重。只量化每层的 7 个线性层（q、k、v、o 投影和 FFN 的 3 个矩阵），它们占了 28 层权重的绝大部分。嵌入表是查表，不做矩阵乘，量化它省的是内存而不是计算；输出层直接决定 15 万个词的打分，对误差最敏感；RMSNorm 的缩放只有几千个数。这三样都保持原样。

一个 scale 管多少个数。INT8 只有 −127 到 127 这 255 个整数（对称量化，不用 −128），要用一个 scale 把它们映射回实数。scale 管得越细，误差越小，存 scale 的开销越大（第 90 讲）。这里每个**输出通道**（权重矩阵的每一行）一个 scale：q_proj 是 [2048, 1024]，就有 2048 个 scale，平均每 1024 个权重多存 4 字节，开销 0.4%。

按行取 scale 还有一个好处：矩阵乘的结果可以先算、后缩放。y 的第 j 个输出 = x · W[j] = x · (q[j] × s_j) = (x · q[j]) × s_j。所以

$$y = x \cdot W^{\top} = (x \cdot q^{\top}) \odot s \qquad s \text{ 是每个输出通道的 scale，} \odot \text{ 按列相乘}$$

不用先把整个矩阵还原成 FP32，只需要在矩阵乘之后，对每一列乘一个数。

每个通道的 scale 怎么定。最简单的对称量化：s = 这一行的最大绝对值 ÷ 127，每个权重除以 s 再四舍五入。最大的那个数正好映射到 ±127，其余的数按比例落在中间（第 90 讲的 RTN，round-to-nearest）。

quant.py

v4/quant.py 共 35 行 ⇆ 对照

```python
"""v4: 只量化权重的 INT8（W8A16 的思路）。每个输出通道（权重矩阵的每一行）一个 scale。"""
import torch



class QuantWeight:
    """一个线性层的 INT8 权重：q 是 [输出, 输入] 的 int8，scale 是每一行的缩放 [输出]。"""

    def __init__(self, w):
        amax = w.abs().amax(dim=1).clamp(min=1e-8)          # 每一行的最大绝对值
        self.scale = (amax / 127).float()                   # 对称量化：[-amax, amax]
```

1 量化一个矩阵 第 5-11 行

对每一行求最大绝对值 `amax`，scale = amax ÷ 127。权重除以自己那一行的 scale，四舍五入，截到 [−127, 127]，存成 int8。

`clamp(min=1e-8)` 防止某一行全是 0 时除以零。

2 **占多少字节** 第 13-14 行

3 **linear：先乘整数矩阵，再乘 scale** 第 17-21 行

4 **量化整个模型** 第 24-35 行

← 上一步 | 看整个文件 | 下一步 →

模型的改动是机械的：每一处 `x @ w[...].T` 换成 `linear(x, w[...])`，一共 7 处。输出层保持原来的写法。

v4/model.py 对比 v3/model.py：+7 行 / −6 行 ⇆ 对照

```
"""Qwen3 的前向计算：只用 torch 的基本运算，一层一层写出来。"""
import torch
+ from quant import linear


…… 省略 23 行没变的代码
        c, w, p = self.cfg, self.w, f"model.layers.{i}.self_attn."
        T = x.shape[0]
+       q = linear(x, w[p + "q_proj.weight"]).view(T, c.num_heads, c.head_dim)
+       k = linear(x, w[p + "k_proj.weight"]).view(T, c.num_kv_heads, c.head_dim)
+       v = linear(x, w[p + "v_proj.weight"]).view(T, c.num_kv_heads, c.head_dim)
-       q = (x @ w[p + "q_proj.weight"].T).view(T, c.num_heads, c.head_dim)
-       k = (x @ w[p + "k_proj.weight"].T).view(T, c.num_kv_heads, c.head_dim)
-       v = (x @ w[p + "v_proj.weight"].T).view(T, c.num_kv_heads, c.head_dim)
        q = rms_norm(q, w[p + "q_norm.weight"], c.rms_eps)        # Qwen3 在 RoPE 之
        k = rms_norm(k, w[p + "k_norm.weight"], c.rms_eps)
…… 省略 2 行没变的代码
        for a, b, cache in segs:                                  # 注意力按请求分开算
            out[a:b] = self.attend(i, q[a:b], k[a:b], v[a:b], pos[a:b], cache)
+       return linear(out.reshape(T, -1), w[p + "o_proj.weight"])
-       return out.reshape(T, -1) @ w[p + "o_proj.weight"].T

    def attend(self, i, q, k, v, pos, cache):
…… 省略 11 行没变的代码
    def mlp(self, i, x):
        w, p = self.w, f"model.layers.{i}.mlp."
+       gate, up = linear(x, w[p + "gate_proj.weight"]), linear(x, w[p + "up_proj.w
+       return linear(torch.nn.functional.silu(gate) * up, w[p + "down_proj.weight"
-       gate, up = x @ w[p + "gate_proj.weight"].T, x @ w[p + "up_proj.weight"].T
-       return (torch.nn.functional.silu(gate) * up) @ w[p + "down_proj.weight"].T

    @torch.no_grad()
…… 省略 12 行没变的代码
```

图 123-2　model.py 和 v3 的差别：只是把 7 个线性层的矩阵乘换成 linear()，计算的结构一点没变。

结果和 HuggingFace 不再完全一样

量化有意改变了权重的数值，生成结果不再保证和 HuggingFace 一致。先看检查的结果：

```
$ .venv/bin/python tests/check.py v4
一致: 'The capital of France is' → ' Paris. The capital of Italy is Rome. The capital
of Spain is Madrid. The capital of China is Beijing. The capital of Japan is Tokyo.
The capital'
一致: 'def fibonacci(n):' → '\n    if n == 0:\n        return 0\n    elif n == 1:\n
return 1\n    else:\n        return fibonacci(n'
不一致: '一个推理引擎由哪几部分组成? ' → '它如何运作?  请提供一个例子，说明如何使用它来解决一个实际
问题。\n答案:  1. 逻辑推理引擎由'
一致: '1, 2, 3, 4, 5,' → ' 6, 7, 8, 9, 10, 11, 12, 13, 14,'
v4: 3/4 条与 HuggingFace 逐 token 一致
```



图 123-3　v4 的检查：4 条里 3 条仍然逐 token 一致；中文那条在中途开始不同。

```
$ .venv/bin/python bench/quant_diverge.py
'The capital of France is': 32 个 token 全部相同
'def fibonacci(n):': 32 个 token 全部相同
'一个推理引擎由哪几部分组成? ': 前 25 个 token 相同，第 26 个开始不同（HuggingFace '问题: 某',
INT8 '1. '）
'1, 2, 3, 4, 5,': 32 个 token 全部相同
```



图 123-4　不一致的那一条，前 25 个 token 和 HuggingFace 相同，第 26 个开始走上了另一条路。

第 26 个 token 处，FP32 的模型选了“问题”，INT8 的模型选了“1.”。这时两个候选的打分本来就很接近，权重的微小误差让排名翻了过来；一旦选了不同的 token，后面的上下文就不同了，生成的内容自然越走越远。这不说明量化后的模型变差了，“1.”接在“答案:”后面同样通顺。

“逐 token 一致”这个标准对量化太苛刻了，要换一种衡量方法：**让两个模型读同一串 token，比较每一步的打分**。下面让 FP32 的模型先生成 128 个 token，再让两个模型都按这串 token 一步一步地读，看每一步的最高分 token 是否相同，以及两个概率分布的 KL 散度（第 16 讲）有多大。

```
$ .venv/bin/python bench/quant.py
线性层权重：FP32 1680 MiB → INT8 421 MiB（3.99 倍）
128 步里最高分 token 相同的比例 100.0%，平均 KL 散度 0.0011
FP32：单请求生成 64 个 token 用时 4.27 s
INT8：单请求生成 64 个 token 用时 5.11 s
```

⇆ 对照

图 123-5　线性层权重从 1680 MiB 降到 421 MiB；128 步里最高分的 token 全部相同，平均 KL 散度 0.0011；但在 CPU 上，INT8 版本反而慢了约 20%。

内存：3.99 倍。线性层从 1680 MiB 降到 421 MiB，多出来的 0.01 是 scale 的开销。嵌入表和输出层（各 593 MiB，FP32）没有量化，整个模型从约 2.8 GiB 降到约 1.6 GiB。

精度：几乎没有损失。128 步里每一步的最高分 token 都相同，平均 KL 散度 0.0011：两个模型给出的概率分布几乎重合。每通道 INT8 对 Qwen3-0.6B 这样的模型足够精确；第 91 讲说过，真正难量化的是有离群激活的大模型，或者 4 位这样更低的位宽。

速度：反而慢了。单请求生成 64 个 token，FP32 用 4.27 s，INT8 用 5.11 s。原因就在 `linear()` 那一行：每次矩阵乘之前，都要把 int8 的矩阵转换成一份临时的 FP32 矩阵。转换本身要读 1 字节、写 4 字节，再由矩阵乘读一遍这 4 字节，比直接读 FP32 的权重还多了一道工序。

量化要变快，前提是**计算时直接读 int8**：在 GPU 上，一个融合的 kernel 把 int8 权重从显存读进片上，在寄存器里换算成 BF16 再做乘加（第 83、91 讲），显存只读了四分之一（相对 FP32）或一半（相对 BF16）的字节，decode 才能跟着快起来。这需要专门写 kernel，第 16 篇看 nano-vllm 和 vLLM 时会碰到。我们这一版拿到的是内存的收益：同样的内存，可以把省下来的约 1.2 GiB 给块池，多放约 360 个块。

要点

每通道的 INT8 权重量化：每一行一个 scale = 最大绝对值 ÷ 127，矩阵乘时先算 $x \cdot q^{\top}$ 再按列乘 scale。线性层内存降到四分之一，Qwen3-0.6B 的打分几乎不变。但在没有专门 kernel 时，每次计算都要把 int8 转回浮点，速度反而变慢；量化的速度收益要靠“直接读 int8 的 kernel”才能兑现。

回顾

v4 新增 quant.py：加载权重后，把每层 7 个线性层量化成 INT8，每个输出通道一个 scale；模型的矩阵乘统一走 `linear()`，量化时先算整数矩阵乘、再按列乘 scale。线性层从 1680 MiB 降到 421 MiB；4 条检查里 3 条仍然逐 token 一致，读同一串 token 时 128 步的最高分 token 全部相同，KL 散度 0.0011。在 CPU 上因为每次都要把 int8 转回浮点，单请求反而慢了约 20%。

到这里，引擎每一步 decode 仍然只为每个请求出一个 token。能不能一步多出几个？下一讲：**投机解码**。

第124讲 投机解码

96 投机解码　97 EAGLE 与 Medusa　121 分页 KV 管理

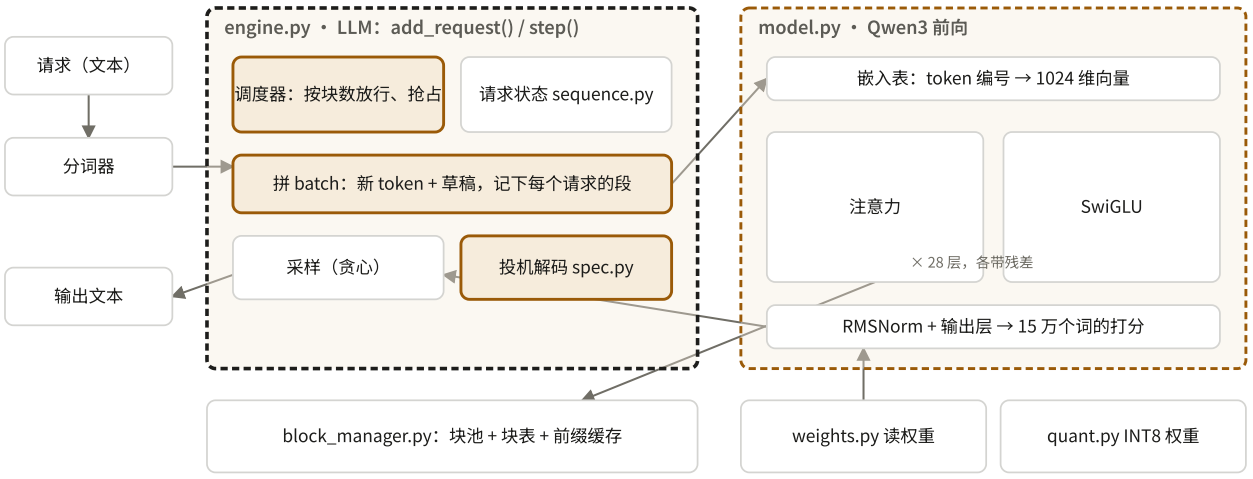
到 v4 为止，每个请求每一步 decode 只出一个 token。第 119 讲测过，decode 一个 token 要 63 ms，而 prefill 199 个 token 只要 357 ms：一次前向多算几个 token，时间涨得很少，因为大头是把权重读一遍。

第 96 讲的投机解码就是利用这一点：先用一个很便宜的办法**猜**接下来的几个 token（草稿），再让模型一次前向把它们全部**验证**一遍。猜对了几个，这一步就多出几个 token；猜错了，最多只损失这一步多算的那几个位置。在贪心采样下，它生成的结果和不投机时**完全一样**。

miniinfer 的架构

⇆ 对照

版本：v0 v1 v2 v3 v3+ v4 v5



引擎（engine.py）：请求先进队列，引擎一步一步地跑 step()：问调度器这一步 batch 里有谁，拼成一维的输入，调用模型，采样，更新每个请求的状态。第 120 讲起是这个样子。

v5 新增或改动（强调色）：engine.py 、调度器、拼 batch、投机解码 spec.py、model.py 。点模块看说明。

图 124-1　v5 新增 spec.py（起草和验证），引擎的 step() 在 decode 请求后面接上草稿一起送进模型，模型多返回几个位置的打分。

设计：草稿从哪里来

第 96 讲的标准做法是用一个小模型起草。我们用一个更简单、完全不需要第二个模型的办法：**n-gram 查找**（也叫 prompt lookup decoding）。

思路是：很多生成任务会大段地重复已有的内容。改写代码时，大部分代码原样抄过来；总结文档时，会引用原文的短语；多轮对话里，会重复上一轮提过的名字。所以，看看当前序列结尾的几个 token，**在前面出现过没有**；出现过，就把那次出现之后紧跟着的几个 token 当作草稿。

```
已有的 token：  … for i, v in enumerate(values):  …  total += v  …  for i, v in
结尾 3 个：                                                          [for, i, v]
前面出现过，后面跟着：                          [in, enumerate, (, values]          ← 草稿
（k = 4）
```

先找结尾 3 个 token，找不到就退到 2 个、1 个；从近往远找，越近的上下文越可能相关。整个过程只是在列表里做比较，比一次前向便宜几个数量级。

设计：一次前向怎么验证

草稿接在最后一个真实 token 后面，一起送进模型。设最后一个真实 token 是 t，草稿是 d_1 d_2 d_3 d_4，送进去的就是 5 个 token：t d_1 d_2 d_3 d_4，位置连续。模型对这 5 个位置各给出一个“下一个 token”的贪心预测 p_0 … p_4：

- p_0 是看完 t 之后的预测。如果 p_0 = d_1，说明草稿第一个猜对了。
- p_1 是看完 t d_1 之后的预测。如果前面猜对了，而且 p_1 = d_2，第二个也猜对了。
- 依此类推，直到第一个不一致的位置。

假设 d_1、d_2 对了，d_3 错了：这一步接受 d_1、d_2，再加上 p_2（模型在“看完 t d_1 d_2”之后真正想要的 token），一共新增 **3 个** token。全猜错也有 p_0，至少和不投机一样多出 1 个。

为什么结果和不投机完全一样？因为每个被接受的 token，都是模型在“同样的前缀”下的贪心选择，和一步一步生成时选的完全相同（第 96 讲）。

KV 怎么办。这一步把 5 个位置的 K、V 都写进了缓存。被接受的位置（t、d_1、d_2）的 K、V 是正确的；被拒绝的位置（d_3、d_4）写进去的是错的。不需要专门去删：把 `num_computed` 只推进到被接受的位置，那两格就算作“没写过”，下一步会被新的 token 覆盖。这就是投机解

码的“回滚”，在分页的块里只是一个数的事。

spec.py

v5/spec.py 共 27 行 ⇆ 对照

```
1  """v5：投机解码。用 n-gram（prompt lookup）起草：在已有的 token 里找和结尾相同的片段，把它后面
2
3
4  def propose(tokens, k=4, max_n=3):
5      """从最长的 n-gram 开始找：结尾的 n 个 token 之前出现过，就把那次出现之后的 k 个 token 当
6      for n in range(max_n, 0, -1):
7          if len(tokens) ⩽ n:
8              continue
9          tail = tokens[-n:]
10         for start in range(len(tokens) - n - 1, -1, -1):      # 从近往远找
```

1 propose：找草稿 第 4–15 行

n 从 3 退到 1：取结尾 n 个 token，从近往远在前面找一样的片段，找到就返回它后面的最多 k 个 token。都找不到就没有草稿，这一步退化成普通的 decode。

这里用的是最朴素的线性查找，每步 O(序列长度 × n)。序列很长时可以用哈希表记录每个 n-gram 最近出现的位置，做到常数时间。

2 **verify：接受到哪里** 第 18–27 行

← 上一步 | 看整个文件 | 下一步 →

模型和调度器的小改动

模型原来每个请求只返回最后一个位置的打分，现在要返回“最后一个真实 token + 每个草稿”的位置。forward 多一个参数 `out_idx`，指定要哪些位置的打分。

v5/model.py 对比 v4/model.py：+4 行 / −3 行 ⇆ 对照

```
      …… 省略 57 行没变的代码
58
59        @torch.no_grad()
60 +      def forward(self, tokens, pos, segs, out_idx=None):
60 -      def forward(self, tokens, pos, segs):
61            # tokens、pos: batch 里所有请求的新 token 和它们的位置，拼成一维 [N]
62            # segs: [(起点, 终点, 缓存), ...]，第 j 个请求的 token 是 tokens[起点:终点]
      …… 省略 4 行没变的代码
67                x = x + self.attention(i, rms_norm(x, w[p + "input_layernorm.weight"],
68                x = x + self.mlp(i, rms_norm(x, w[p + "post_attention_layernorm.weight"
69 +          if out_idx is None:                                          # 默认每个请求只要最
70 +              out_idx = [b - 1 for _, b, _ in segs]
71 +          x = rms_norm(x[torch.tensor(out_idx, device=x.device)], w["model.norm.weigh
69 -          last = torch.tensor([b - 1 for _, b, _ in segs], device=x.device)
70 -          x = rms_norm(x[last], w["model.norm.weight"], c.rms_eps)  # 每个请求只需要最后
72            return x @ w["lm_head.weight"].T
```

图 124-2　model.py 和 v4 的差别：forward 可以指定返回哪些位置的打分，默认仍是每个请求的最后一个位置。

调度器给 decode 的请求拿块时，要多留草稿的位置；块管理器的 `append_slot` 相应地多一个参数。

v5/scheduler.py 对比 v4/scheduler.py：+4 行 / −3 行 ⇆ 对照

```
      …… 省略 4 行没变的代码
  5
  6   class Scheduler:
  7 +     def __init__(self, block_manager, max_num_seqs=8, max_num_tokens=2048, num_dra
  7 -     def __init__(self, block_manager, max_num_seqs=8, max_num_tokens=2048):
  8           self.bm = block_manager
  9 +         self.num_draft = num_draft                  # 每步最多给每个请求起草几个 token
 10           self.max_num_seqs = max_num_seqs
 11           self.max_num_tokens = max_num_tokens
      …… 省略 11 行没变的代码
 23               if seq not in self.running:             # 已经在本轮被抢占了
 24                   continue
 25 +             while not self.bm.append_slot(seq, self.num_draft):
 24 -             while not self.bm.append_slot(seq):
 26                   victim = self.running[-1]           # 抢占最晚进来的那个，把它的块让出来
 27                   self.preempt(victim)
      …… 省略 2 行没变的代码
 30               if seq in self.running:
 31                   batch.append(seq)
 32 +         budget = self.max_num_tokens - len(batch) * (1 + self.num_draft)
 31 -         budget = self.max_num_tokens - len(batch)
 33           # 2. 有空位、有空闲块，就从等待队列放新请求进来做 prefill
 34           while self.waiting and len(batch) < self.max_num_seqs:
      …… 省略 30 行没变的代码
```

图 124-3　scheduler.py 和 v4 的差别：保证块表能放下“下一个位置 + 草稿”，token 预算也按 1 + 草稿数来算。

engine.py：起草、验证、回滚

v5/engine.py 第 35–70 行 / 共 82 行 ⇆ 对照

```
35      def step(self):
36          batch = self.scheduler.schedule()
37          tokens, pos, segs, out_idx, drafts = [], [], [], [], []
38          for seq in batch:
39              if seq.cache is None:
40                  seq.cache = SeqKV(self.pool, seq.block_table)
41              new = seq.new_tokens()
42              # decode 的请求：在已有 token 里找草稿，接在最后一个 token 后面一起送进去
43              draft = propose(seq.tokens, self.num_draft) if len(new) == 1 and self.nu
```

1 给 decode 的请求起草 第 41–46 行

只有 decode 的请求（这一步只送 1 个 token）才起草；prefill 的请求照常。草稿不能超过这个请求还能生成的数量，否则会多生成。

2 **要哪些位置的打分** 第 47–51 行

3 **一次前向** 第 55–57 行

4 **逐个请求验证** 第 58–62 行

5 **回滚：只推进到被接受的位置** 第 63 行

6 **追加 token，检查是否结束** 第 64–70 行

← 上一步 | 看整个文件 | 下一步 →

先验证：投机不改变结果

```
$ .venv/bin/python tests/check.py v5
一致: 'The capital of France is' → ' Paris. The capital of Italy is Rome. The capital of Spain is Madrid. The capital of China is Beijing. The capital of Japan is Tokyo. The capital'
一致: 'def fibonacci(n):' → '\n    if n == 0:\n        return 0\n    elif n == 1:\n return 1\n    else:\n        return fibonacci(n'
一致: '一个推理引擎由哪几部分组成？' → '它如何运作？  请提供一个例子，说明如何使用它来解决一个实际问题。\n答案:  问题：某公司需要为员工'
一致: '1, 2, 3, 4, 5,' → ' 6, 7, 8, 9, 10, 11, 12, 13, 14,'
v5: 4/4 条与 HuggingFace 逐 token 一致
```

图 124-4　v5（每步最多 4 个草稿）和 HuggingFace 逐 token 一致：起草和验证只改变了“一步出几个 token”，没有改变“出哪些 token”。

v5 默认不打开量化：我们要验证投机解码本身不改变结果，量化会带来另一种差异，混在一起就分不清了。

测一测：抄得多，猜得准

投机解码的收益完全取决于草稿猜得准不准。用两种请求对比：一种是“给一个函数加上类型标注，其余不变”，答案大部分是在抄输入的代码；另一种是自由地写一段说明。

```
$ .venv/bin/python bench/spec.py
改写代码（大量照抄），草稿 0 个：128 个 token 用了 127 步 decode，接受草稿 0 个，平均每次前向 1.00 个 token，用时 9.4 s
改写代码（大量照抄），草稿 4 个：128 个 token 用了 34 步 decode，接受草稿 93 个，平均每次前向 3.66 个 token，用时 6.6 s
  两种方式的输出完全一致：True
自由生成，草稿 0 个：128 个 token 用了 127 步 decode，接受草稿 0 个，平均每次前向 1.00 个 token，用时 9.2 s
自由生成，草稿 4 个：128 个 token 用了 53 步 decode，接受草稿 74 个，平均每次前向 2.37 个 token，用时 8.0 s
  两种方式的输出完全一致：True
```

图 124-5　改写代码：128 个 token 只用了 34 步 decode，接受了 93 个草稿，平均每次前向 3.66 个 token，用时从 9.4 s 降到 6.6 s。自由生成：53 步，平均每次前向 2.37 个 token，用时从 9.2 s 降到 8.0 s。两种情况的输出都和不投机时完全一致。

步数大幅减少。改写代码时，n-gram 草稿 128 个 token 里猜对了 93 个，decode 从 127 步降到 34 步。连自由生成也有 2.37 个 token 一步：这个小模型在贪心解码下很容易重复自己说过的话，n-gram 正好抓住了这一点。

时间没有按步数同比例减少。步数降到约四分之一，时间只降到七成。原因是每一步不再只算 1 个 token，而是 5 个；在 CPU 上，多算 4 个 token 并不是免费的（不像在 GPU 上那样，decode 时算力大量闲置）。第 96 讲的加速比公式里，这对应“验证 k 个草稿的代价”这一项；CPU 上这一项比 GPU 上大得多，所以同样的接受率，CPU 上的加速比更小。

猜不准时也不亏多少。草稿全错时，这一步仍然出 1 个 token，只是多算了几个位置。草稿在 batch 很大时会更“贵”：每个请求多出的 k 个位置都要算，batch 已经让硬件很忙时，多算的部分就不再藏在读权重的时间里了。这也是工业引擎在高负载时会自动减少、甚至关闭投机的原因。

要点

投机解码 = 便宜地起草 + 一次前向验证。n-gram 起草在已有 token 里找和结尾相同的片段，把后面的几个 token 当草稿；验证时逐个比较草稿和模型的贪心预测，接受到第一个不一致处，再加上模型自己的那个 token。被拒绝的草稿位置只要不推进 num_computed 就自动作废。贪心采样下结果和不投机完全一样。

回顾

v5 新增 spec.py：`propose` 用 n-gram 在已有 token 里找草稿，`verify` 按模型在每个位置的贪心预测决定接受到哪里。引擎在 decode 请求后面接上草稿一起送进模型，模型按 `out_idx` 返回需要的几个位置的打分；回滚只需要把 `num_computed` 推进到被接受的位置。和 HuggingFace 仍然逐 token 一致。改写代码时平均每次前向出 3.66 个 token，用时从 9.4 s 降到 6.6 s；自由生成也有 2.37 个。

到这里，miniinfer 已经有了一个推理引擎的主要零件。它离 vLLM 这样的工业级引擎还差多远？时间到底花在了哪里？下一讲：**时间花在哪里**。

第125讲 时间花在哪里

66 性能分析　82 推理压测实践　85 Decode 注意力 Kernel　89 CUDA Graph　124 投机解码

六个版本写完，miniinfer 有了一个推理引擎的主要零件：手写的模型前向、KV Cache、continuous batching、分页和前缀缓存、INT8 量化、投机解码，全部约 400 行 Python，每一版都和 HuggingFace 逐 token 一致（量化版除外，差异也量化过了）。

这一讲回头看三件事：每一版到底换来了什么；现在的时间花在哪里；它和 vLLM 这样的工业级引擎还差多远，差在哪些地方。最后一个问题的答案，就是第 16、17 篇要读的内容。

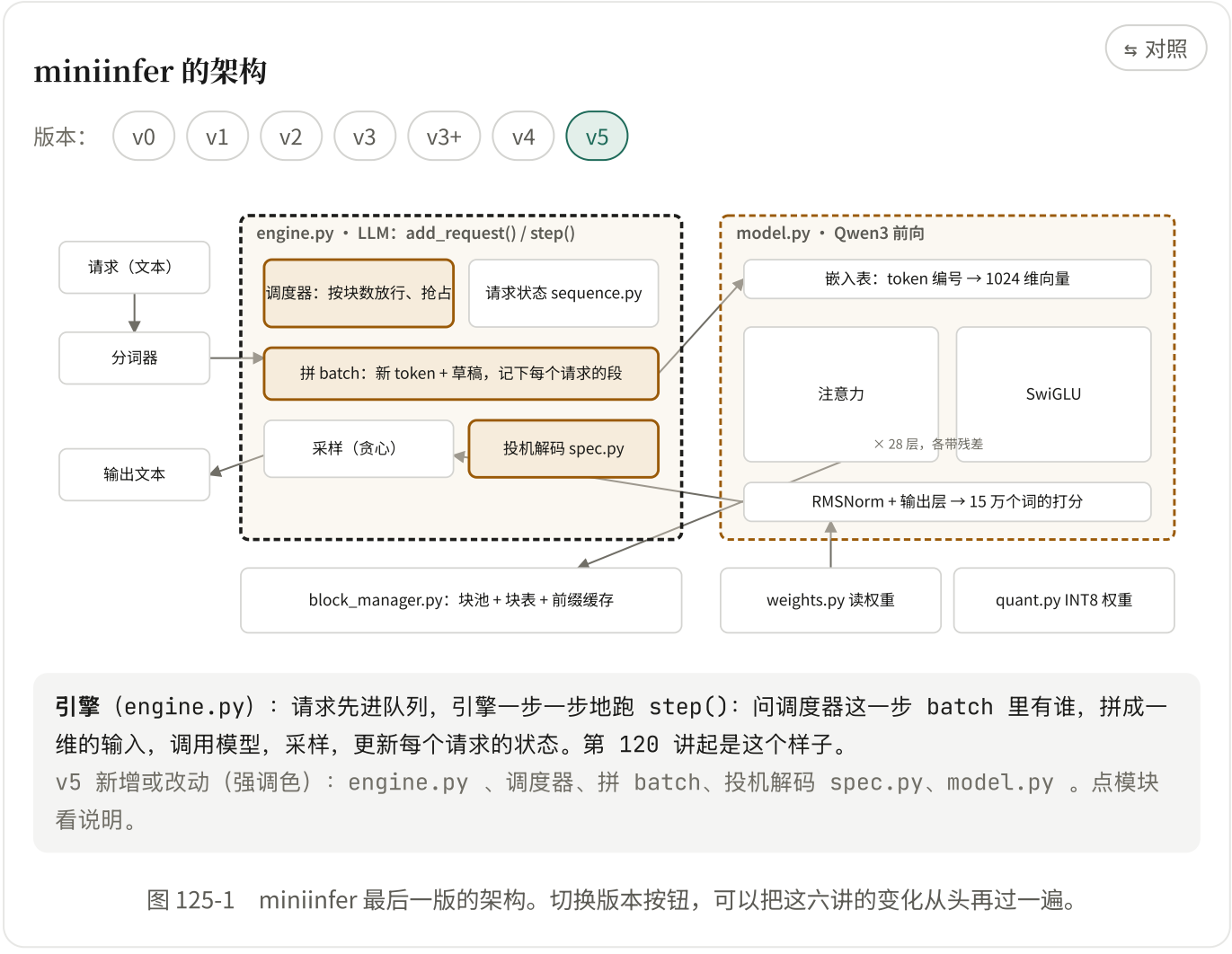


图 125-1　miniinfer 最后一版的架构。切换版本按钮，可以把这六讲的变化从头再过一遍。

每一版换来了什么

版本	改了什么	实测（Apple M2，CPU，FP32）	讲
v0	手写前向，每步整段重算	199 个 token 的提示生成 64 个：34.1 s	106
v1	KV Cache	同样的任务 4.55 s，快 7.5 倍；decode 每步约 65 ms	107
v2	continuous batching	16 个请求、792 个 token：27.5 token/s，一个一个跑的 1.9 倍	108
v3	分页 KV、抢占	KV 只有 320 个位置时：22.8 token/s，v2 只有 17.0	109
v3+	前缀缓存	共用系统提示的 7 个请求：prefill 从 2356 个 token 降到 116 个，5.5 s → 2.4 s	110
v4	INT8 权重量化	线性层 1680 MiB → 421 MiB；128 步最高分 token 全部相同；CPU 上慢了约 20%	111
v5	n-gram 投机解码	改写代码：每次前向 3.66 个 token，9.4 s → 6.6 s；输出不变	112

代码量是这样长起来的：

```
$ .venv/bin/python bench/loc.py                                                    ⇆ 对照

v0: 119 行  config.py 30  engine.py 28  model.py 50  weights.py 11
v1: 140 行  config.py 30  engine.py 32  kv_cache.py 14  model.py 53  weights.py 11
v2: 230 行  config.py 30  engine.py 52  kv_cache.py 14  model.py 60  scheduler.py 40
sequence.py 23  weights.py 11
v3: 336 行  block_manager.py 100  config.py 30  engine.py 56  model.py 60  scheduler.py
56  sequence.py 23  weights.py 11
v4: 365 行  block_manager.py 100  config.py 30  engine.py 59  model.py 61  quant.py 25
scheduler.py 56  sequence.py 23  weights.py 11
v5: 407 行  block_manager.py 100  config.py 30  engine.py 76  model.py 62  quant.py 25
scheduler.py 57  sequence.py 23  spec.py 23  weights.py 11
```

图 125-2　每个版本的代码行数（不算空行）。v0 只有 119 行；最大的一次增长是 v3 的块管理器（100 行），它同时承担了分页、抢占时的回收和前缀缓存。

回头看这张表，有两件事值得记住。第一，**每一版的收益都有前提**：KV Cache 在生成越长时收益越大；continuous batching 要有足够多的请求同时在跑；分页在 KV 空间紧张时才显出优势；前缀缓存要请求真的共用前缀；投机解码要草稿猜得准。脱离负载谈优化，很容易得出错误

的结论。第二，**量化在我们这里没有变快**，这不是量化的问题，而是我们没有一个直接读 int8 的 kernel。下面会看到，这是 miniinfer 和工业引擎差距最大的地方。

时间花在哪里

用 v5（关掉投机）跑 8 个请求、各生成 64 个 token，把每一步拆开计时：

```
$ .venv/bin/python bench/breakdown.py                    ⇆ 对照

8 个请求各生成 64 个 token：共 13.1 s，64 步
  注意力（每个请求单独算，含读写 KV）：3.1 s（24%）
  线性层、归一化等其余张量运算：10.0 s（76%）
  Python 侧：调度、拼 batch、采样：26 ms（0.2%）
单请求 64 个 token：HuggingFace 4.15 s，miniinfer v5 4.76 s
```

图 125-3　8 个请求的 decode：注意力占 24%，线性层等其余张量运算占 76%，Python 侧的调度、拼 batch 和采样只占 0.2%。单请求时 miniinfer 和 HuggingFace transformers 的速度相当。

线性层占 76%。每一步要把约 2.8 GiB 的 FP32 权重读一遍，CPU 的内存带宽决定了这一部分的下限。在 GPU 上，这部分换成 BF16（字节减半）、读 HBM（带宽高一两个数量级），再加上 INT8 或 FP8 的 kernel，能再快几倍。

注意力占 24%。这一部分比它“应该”占的多。原因有两个，都是我们为了代码清楚而付的代价：一是按请求用 Python 循环逐个算，8 个请求就是 8 次小的矩阵运算；二是 `SeqKV.read` 每层、每步都把这个请求的块**复制**成一块连续的内存，再交给注意力。工业引擎用一个 PagedAttention kernel（第 85 讲）一次处理所有请求，直接按块表去读，不复制。

Python 侧只占 0.2%。在我们这里，一步要 200 ms，调度和拼 batch 的几十微秒可以忽略。可是到了 GPU 上，一步 decode 只要几毫秒，同样的 Python 开销就会占到可观的比例。第 89 讲提到的 vLLM v0.6.0 的剖析里，API 服务器和调度加起来一度占了一步的六成。所以工业引擎要把调度放到单独的进程、和 GPU 计算异步地重叠，还要用 CUDA Graph 把一步里几百个 kernel 的发射合成一次。

和 HuggingFace 差不多。单请求生成 64 个 token，HuggingFace 4.15 s，miniinfer 4.76 s。transformers 的 `generate` 也是带 KV Cache 的逐 token 生成，在 CPU 上两者做的事几乎一样；miniinfer 慢一点，主要是分页带来的 KV 复制。我们的优势在多请求时才显现：HuggingFace 的 `generate` 没有 continuous batching，也没有分页和前缀缓存。

要点

在 CPU 上，miniinfer 的时间主要花在线性层（76%），其次是注意力（24%），Python 的调度几乎可以忽略。换到 GPU 上，线性层会快得多，注意力里“逐请求循环 + 复制 KV”的写法和 Python 侧的开销就会变成主要矛盾，这正是工业引擎花大力气解决的地方。

离工业级引擎还差什么

把 miniinfer 的每个模块，和工业引擎的做法放在一起：

模块	miniinfer 的做法	工业引擎的做法	原理讲	在哪里读
注意力	Python 循环逐个请求；按块表把 KV 复制成连续内存	一个 kernel 处理整个 batch：prefill 用 FlashAttention 的变长版本，decode 直接按块表读 KV	75、76	第 127 讲
写 KV	用下标张量逐位置写	一个小 kernel，按“槽位映射”一次写完整个 batch 的 K、V	76	第 127 讲
kernel 发射	每个运算一次发射，CPU 上不明显	CUDA Graph 按 batch 大小分档预先捕获	79	第 127、129 讲
量化	int8 → FP32 再做矩阵乘	融合的 W8A16 / FP8 kernel，直接读低位宽的权重	80–83	第 129 讲
调度	每步先 decode 再 prefill，整个提示一次 prefill	统一的 token 预算，长提示分块 prefill，和 decode 混在同一步	69	第 129 讲
前缀缓存	进程内的链式哈希	稳定的哈希（可跨进程）、基数树（SGLang 的 RadixAttention）、跨机器的缓存池	68、101	第 129、131 讲
投机解码	n-gram，贪心验证	小模型、EAGLE、MTP 起草；树形验证；支持随机采样的接受规则	85、86	第 129 讲
采样	只有贪心	温度、top-k、top-p、惩罚项、约束解码，在 GPU 上批量完成	65、71	第 129、131 讲
服务	一个 Python 函数	HTTP / OpenAI 兼容接口、流式输出、调度和执行分进程、异步重叠	72、79	第 129、130 讲
多卡	不支持	张量并行、专家并行、PD 分离	93–100	第 127、130、132 讲

表里的每一行，都可以在 miniinfer 上动手做一版。几个适合练手的方向：

- **分块 prefill**：在调度器里给长提示设一个 token 预算，一步只算提示的一段，下一步接着算。要改的是 `Sequence.new_tokens()` 的切法和调度器的预算，模型不用改（第 77 讲）。
- **top-p 采样**：把引擎里的 `argmax` 换成第 73 讲的采样；投机解码的验证规则也要随之换成第 96 讲的接受-拒绝采样，才能保持分布不变。

- **流式输出**：让 `step()` 返回这一步每个请求新增的 token，外面用一个生成器把它们逐个吐出去。
- **按块表直接读的注意力**：不再复制 KV，而是对每个块分别算打分，用第 84、85 讲的在线 softmax 合并。这是通往 PagedAttention kernel 的第一步。

回顾

miniinfer 用约 400 行 Python 实现了一个推理引擎的骨架：v0 手写前向，v1 用 KV Cache 把 64 个 token 从 34 s 降到 4.55 s，v2 的 continuous batching 让吞吐提高到 1.9 倍，v3 的分页在 KV 紧张时再提高 34%，前缀缓存省掉了 95% 的重复 prefill，v4 把线性层的内存降到四分之一，v5 的投机解码在照抄型任务上让每次前向出 3.66 个 token。在 CPU 上，时间 76% 花在线性层、24% 在注意力，Python 的调度可以忽略。

和工业级引擎相比，最大的差距在 kernel（注意力、写 KV、量化矩阵乘）、发射开销（CUDA Graph）和服务化（异步调度、流式接口）。下一篇换一个角度：读一个真实的开源小引擎 nano-vllm，看它用一千多行代码怎么把这些补上。下一讲：**nano-vllm 的整体结构**。

AI INFRA · HANDBOOK

第 16 篇

解读 nano-vllm

收录 3 讲 · 第 126—128 讲

第126讲 nano-vllm 的整体结构

117 一个推理引擎的组成　120 Continuous Batching　121 分页 KV 管理　122 前缀缓存

第 15 篇的 miniinfer 是为了讲清楚而写的：只用 torch 的基本运算，在 CPU 上逐 token 对得上 HuggingFace。它和真正能用的引擎差在哪里，第 125 讲列了一张表。这一篇换个方向：读一个**真实可用、又小到能一口气读完**的开源引擎，看它怎么把那张表上的几项补上。

我们读的是 nano-vllm（作者 Xingkai Yu，MIT 许可）。它用约 1200 行 Python 实现了 vLLM 的核心：分页 KV、前缀缓存、continuous batching、分块 prefill、CUDA Graph、张量并行，接口也和 vLLM 一样。按它 README 里的测试（RTX 4070 笔记本 GPU，Qwen3-0.6B，256 个请求，输入、输出长度各在 100–1024 之间随机），vLLM 是 1361.84 token/s，nano-vllm 是 1434.13 token/s。这组数字是作者测的，本站没有复现。

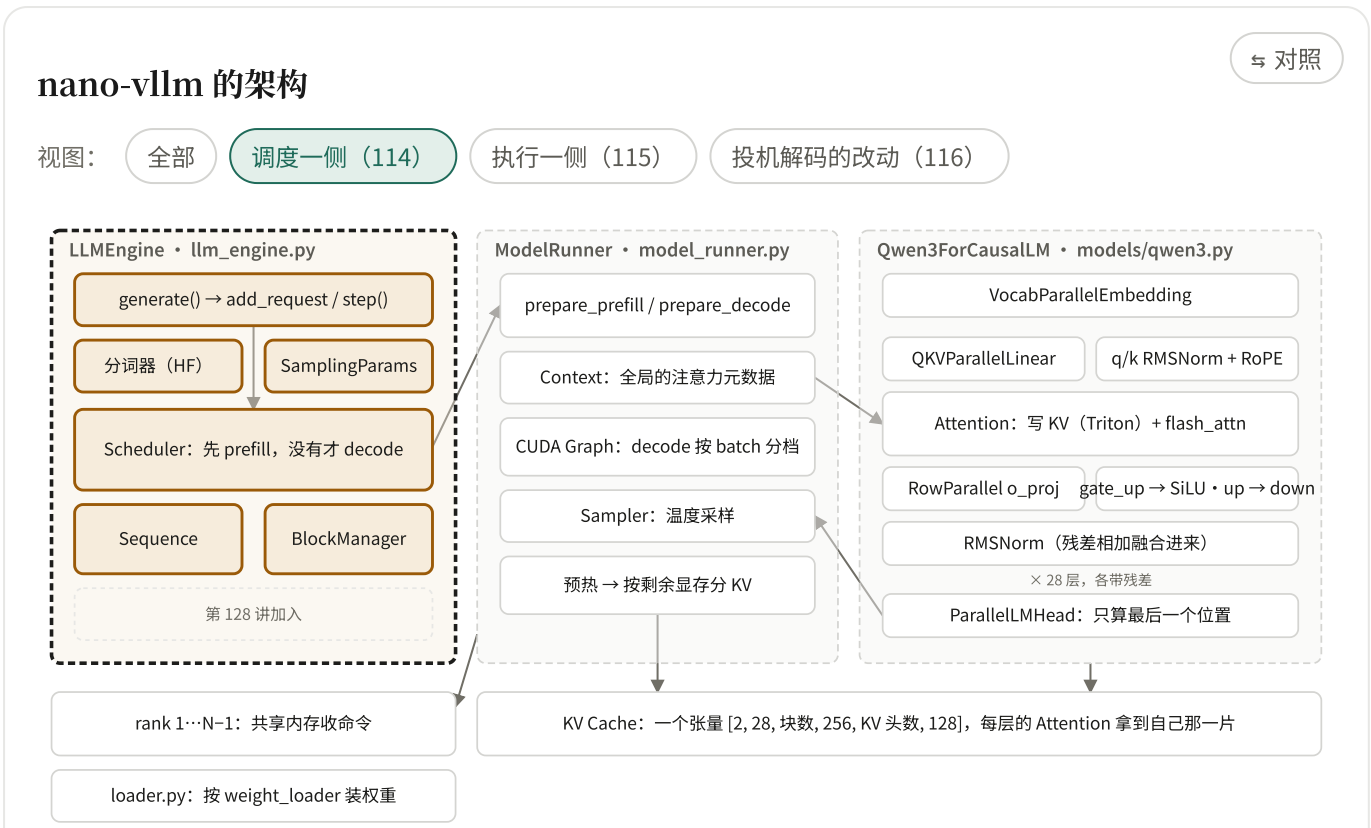
这一讲读调度一侧：请求怎么进来、调度器每一步挑谁、块管理器怎么分块和做前缀缓存。这些全是普通的 Python 代码，和 miniinfer 的 v2、v3 一一对应。下一讲读执行一侧：模型怎么在 GPU 上跑、FlashAttention 和 CUDA Graph 怎么接进来、张量并行怎么切。

> **注意**
> nano-vllm 依赖 NVIDIA GPU（flash-attn、Triton、NCCL）。本站的机器没有 GPU，这一篇**只读源码，没有运行它**；文中的行数统计只读源码文本。读的是 2026 年 4 月 26 日的提交 `bb823b3`，之后的版本行号可能不同。

拿到同一份代码：

```
cd miniinfer && git clone https://github.com/GeeeekExplorer/nano-vllm
third_party/nano-vllm
cd third_party/nano-vllm && git checkout bb823b3
```

全局：三块，两个进程边界



LLMEngine · llm_engine.py（engine/llm_engine.py）：引擎：在主进程里。generate() 把请求放进调度器，然后反复调用 step()：问调度器这一步跑谁 → 交给执行器跑模型、采样 → 把新 token 交回调度器更新状态。

miniinfer 里对应：engine.py 的 LLM 类。点模块看说明。

图 126-1　nano-vllm 的架构。左边的引擎在主进程里，跑调度；中间的执行器每张卡一个，拼输入、跑模型、采样；右边是模型。强调色是这一讲读的部分。点模块可以看它在哪个文件、对应 miniinfer 的什么。

和 miniinfer 比，最大的结构差别是**调度和执行分开了**。miniinfer 的 `engine.step()` 一个函数里既问调度器、又拼 batch、又跑模型；nano-vllm 把它拆成三段：

1. **调度器**（`Scheduler`）只决定这一步跑哪些请求、每个请求算几个 token，顺带管块。它只碰 Python 对象，不碰张量。
2. **执行器**（`ModelRunner`）拿到这份名单，把它变成 GPU 上的张量（token、位置、写 KV 的槽位、块表），跑模型，采样，返回每个请求的新 token。
3. **引擎**（`LLMEngine`）把两者串起来，再把新 token 交回调度器更新状态。

拆开的原因是张量并行：有 N 张卡时，执行器有 N 份，每张卡一个进程；调度只需要一份，放在主进程里。调度器做完决定，rank 0 的执行器把“要跑哪些请求”广播给其余的执行器，大家一起算同一步。第 127 讲会看到这条广播是怎么做的。

整个仓库的文件：

```
nanovllm/
├── llm.py, config.py, sampling_params.py    入口和配置
├── engine/
│   ├── llm_engine.py       引擎：generate() 和 step()
│   ├── sequence.py         一个请求的状态
│   ├── scheduler.py        调度器
│   ├── block_manager.py    块管理、前缀缓存
│   └── model_runner.py     执行器（第 127 讲）
├── models/qwen3.py         模型结构（第 127 讲）
├── layers/                 注意力、线性层、归一化、RoPE、采样等（第 127 讲）
└── utils/                  读权重、全局的注意力元数据
```

逐模块数一下行数，和 miniinfer 的最后一版对照：

```
$ .venv/bin/python bench/nano_loc.py                    ⇆ 对照

nano-vllm  miniinfer v5  模块
      114            30  入口、配置
       65            23  请求状态
       80            57  调度器
      102           100  块管理、前缀缓存
      256            76  执行器（拼 batch、跑模型）
      479            62  模型
       66             –  注意力、写 KV
        9             –  采样
       24            11  读权重
        –            25  量化
        –            23  投机解码
     1195           407  合计（不含 __init__.py）
```

图 126-2　nano-vllm 和 miniinfer v5 每个模块的行数（不算空行）。调度一侧（请求状态、调度器、块管理）两者差不多大；多出来的 800 行几乎都在执行一侧：执行器 256 行、模型 479 行、注意力 66 行。

这张表本身就说明了一件事：**调度的逻辑不需要很多代码**。请求状态、调度器、块管理器加起来 nano-vllm 是 247 行，miniinfer 是 180 行。真正多出来的，是让模型在 GPU 上跑得快的那部分：模型层要支持张量并行切分（每种线性层都有自己的切法和装权重的方法），执行器要管

CUDA Graph、KV 显存的估算和多进程通信。

一个请求走一圈：llm_engine.py

nano-vllm 的用法和 vLLM 一样：

```
llm = LLM("/path/to/Qwen3-0.6B", enforce_eager=True, tensor_parallel_size=1)
outputs = llm.generate(["introduce yourself"], SamplingParams(temperature=0.6,
max_tokens=256))
```

`LLM` 只是 `LLMEngine` 的别名。引擎的全部代码：

third_party/nano-vllm/nanovllm/engine/llm_engine.py 第 15–90 行 / 共 90 行 ⇆ 对照

```
15 class LLMEngine:
16
17     def __init__(self, model, **kwargs):
18         config_fields = {field.name for field in fields(Config)}
19         config_kwargs = {k: v for k, v in kwargs.items() if k in config_fields}
20         config = Config(model, **config_kwargs)
21         Sequence.block_size = config.kvcache_block_size
22         self.ps = []
23         self.events = []
```

1 配置：把关键字参数筛进 Config 第 17–21 行

只挑出 `Config` 里有的字段，其余的忽略。`Config` 的默认值值得记一下：每步最多 16384 个 token、最多 512 个请求、上下文最长 4096、用 90% 的显存、**每块 256 个位置**（miniinfer 是 16）。块大小为什么这么大，下一讲讲 flash-attn 时解释。

`Sequence.block_size` 是类属性，所有请求共用。

2 **每张卡一个执行器** 第 22–31 行

3 **分词器和调度器** 第 32–35 行

4 **add_request：变成一个 Sequence 放进等待队列** 第 43–47 行

5 **step：调度 → 执行 → 更新** 第 49–55 行

6 **generate：跑到所有请求结束** 第 60–90 行

← 上一步 | 看整个文件 | 下一步 →

请求状态：sequence.py

third_party/nano-vllm/nanovllm/engine/sequence.py 共 83 行 ⇆ 对照

```python
from copy import copy
from enum import Enum, auto
from itertools import count

from nanovllm.sampling_params import SamplingParams


class SequenceStatus(Enum):
    WAITING = auto()
    RUNNING = auto()
```

1 **三种状态** 第 8-11 行

和 miniinfer 一样：等待、运行、结束。被抢占的请求回到等待。

2 **字段** 第 18-31 行

3 **块的辅助函数** 第 55-65 行

4 **跨进程时只发必要的字段** 第 72-83 行

← 上一步 | 看整个文件 | 下一步 →

调度器：scheduler.py

nano-vllm 的调度规则可以概括成一句话：**能 prefill 就只 prefill，没有可 prefill 的才 decode**。一步要么全是 prefill，要么全是 decode，不混在一起。

这和 miniinfer 不同。miniinfer 从 v2 起，每一步先让运行中的请求各 decode 一个 token，再用剩下的预算放新请求进来 prefill，两种混在同一步（第 120 讲）。两种做法的取舍：

	miniinfer：混在一步	**nano-vllm：prefill 优先，分开跑**
新请求的首 token	要和 decode 分预算	最快：有空间就立刻 prefill
正在生成的请求	每一步都前进	有新请求进来的那一步，全部停一下
一步的形状	长短混杂	整齐：要么全是长段，要么每个请求一个 token
对执行一侧的要求	注意力要同时处理两种	prefill 和 decode 各用一个专门的 kernel；decode 可以录成 CUDA Graph

最后一行是 nano-vllm 这样选的主要原因：decode 步的形状非常规整（每个请求正好一个 token），才能用 CUDA Graph 预先录好（第 127 讲）。vLLM V1 走的是另一条路：把 prefill 和 decode 统一成“每个请求这一步算几个 token”，混在同一步里，靠一个能处理变长查询的注意力 kernel 兼顾两者（第 77 讲的分块 prefill，第 129 讲）。

third_party/nano-vllm/nanovllm/engine/scheduler.py 共 92 行 ⇆ 对照

```python
from collections import deque

from nanovllm.config import Config
from nanovllm.engine.sequence import Sequence, SequenceStatus
from nanovllm.engine.block_manager import BlockManager


class Scheduler:

    def __init__(self, config: Config):
```

1 **两个队列和一个块管理器** 第 10-17 行

等待队列和运行队列都是 `deque`。块管理器按执行器量出来的块数建块池。

2 **prefill：看等待队列的第一个** 第 29-41 行

3 **分块 prefill：只有第一个请求能被切开** 第 42-43 行

4 **分块，记下这一步算几个** 第 44-52 行

5 **有 prefill 就直接返回** 第 54-55 行

6 **decode：块不够就抢占** 第 57-73 行

7 **保持运行队列的顺序** 第 71-73 行

8 **抢占：放回等待队列的最前面** 第 75-79 行

9 **postprocess：记账，接上新 token** 第 81-92 行

← 上一步 | 看整个文件 | 下一步 →

块管理器：block_manager.py

third_party/nano-vllm/nanovllm/engine/block_manager.py 共 120 行 ⇆ 对照

```python
from collections import deque
import xxhash
import numpy as np

from nanovllm.engine.sequence import Sequence


class Block:

    def __init__(self, block_id):
```

1 **一个块的元数据** 第 8-23 行

每个物理块一个 `Block` 对象：块号、引用计数、哈希、以及它装的 token。miniinfer 用几个并列的数组（`ref`、`hash_of`）存同样的信息。

多存一份 `token_ids` 是为了防哈希碰撞，见第 4 步。

2 **空闲队列、已用集合、哈希表** 第 28-33 行

3 **链式哈希：xxhash64** 第 35-41 行

4 **拿一块、还一块** 第 43-56 行

5 **can_allocate：能命中几块，块够不够** 第 58-73 行

6 **allocate：命中的共用，其余拿新块** 第 75-92 行

7 **deallocate：倒着还** 第 94-101 行

8 **decode 时是否要新块** 第 103-108 行

9 **hash_blocks：算完之后登记** 第 110–120 行

← 上一步 | 看整个文件 | 下一步 →

把调度一侧和 miniinfer 对照：

	miniinfer v3+	nano-vllm
块大小	16	256（flash-attn 分页 KV 的要求，第 127 讲）
一步里	decode 和 prefill 混在一起	只 prefill 或只 decode
分块 prefill	没有	有，只用于超过一步预算的长提示
前缀缓存的哈希	Python `hash`，单进程内稳定	xxhash64，可跨进程；命中后再比一遍 token
不复用的块	含最后一个 token 的块	最后一块（效果相同）
抢占	运行队列末尾，释放后重算	相同
空闲块的淘汰	先还先用，倒着还	相同
块数	手动指定	启动时按剩余显存算出

块大小 256 有代价：前缀缓存的粒度变粗了，两个请求要有 256 个 token 完全相同的前缀才能共用一块；每个请求最后一块平均浪费约 128 个位置。第 122 讲的例子（共用约 330 个 token 的系统提示）在 nano-vllm 里只能命中 1 块（256 个位置），miniinfer 能命中 20 块（320 个位置）。

要点

nano-vllm 把一步拆成三段：调度器（只碰 Python 对象，决定这一步跑谁、算几个 token、管块）、执行器（把名单变成 GPU 张量、跑模型、采样，每张卡一个）、引擎（串起两者）。调度规则是 prefill 优先、一步只做一种，这让 decode 步的形状规整到可以录成 CUDA Graph。块管理和前缀缓存的逻辑和 miniinfer v3 几乎一样，只是块大 16 倍、哈希稳定、命中后多比一次 token。

回顾

nano-vllm 约 1200 行，调度一侧（请求状态、调度器、块管理器）只占 247 行，和 miniinfer 的 180 行差不多，逻辑也几乎一一对应：等待和运行两个队列、按块数放行、块不够就抢占最晚进来的、链式哈希做前缀缓存、引用计数共用块、倒着还块让前缀留得更久。不同的地方有三处：一步只做 prefill 或只做 decode；超长提示可以分块 prefill；块大小是 256。

多出来的约 800 行都在执行一侧。下一讲：**nano-vllm 多做了什么**——FlashAttention、写 KV 的 Triton kernel、CUDA Graph 和张量并行是怎么接进来的。

第127讲 nano-vllm 多做了什么

84 FlashAttention　85 Decode 注意力 Kernel　89 CUDA Graph　105 张量并行　126 nano-vllm 的整体结构

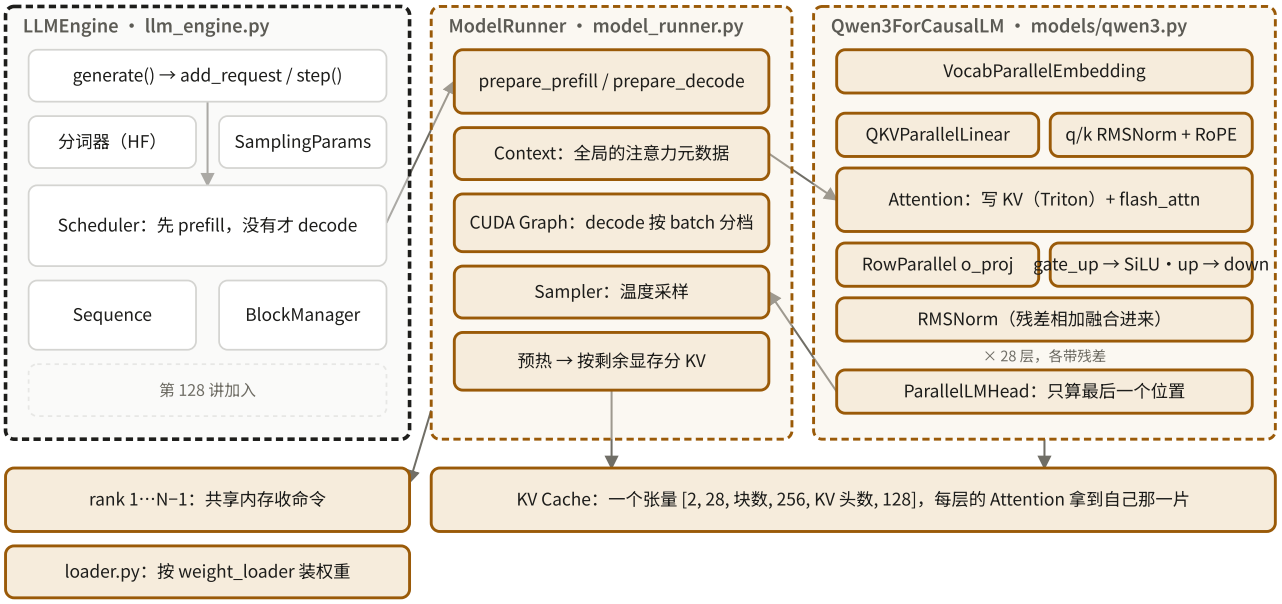
第 125 讲测过，miniinfer 在 CPU 上的时间 76% 花在线性层、24% 在注意力；同时也指出，换到 GPU 上，注意力里“逐请求循环 + 复制 KV”的写法、每个运算一次的 kernel 发射，会变成主要矛盾。上一讲看到，nano-vllm 比 miniinfer 多出来的约 800 行，几乎都在执行一侧。这一讲逐项读这部分：

1. **启动**：怎么装权重、怎么算出能放多少个 KV 块；
2. **拼输入**：一步的请求怎么变成 GPU 上的几个数组；
3. **注意力**：写 KV 的 Triton kernel，和两个 flash-attn 函数；
4. **CUDA Graph**：decode 怎么按 batch 大小分档录图、重放；
5. **张量并行**：每种线性层怎么切，进程之间怎么同步；
6. 几个小的优化：合并矩阵、融合逐元素运算、在 GPU 上采样。

nano-vllm 的架构

⇆ 对照

视图： 全部 | 调度一侧（114） | 执行一侧（115） | 投机解码的改动（116）



`LLMEngine · llm_engine.py`（`engine/llm_engine.py`）：引擎：在主进程里。`generate()` 把请求放进调度器，然后反复调用 `step()`：问调度器这一步跑谁 → 交给执行器跑模型、采样 → 把新 token 交回调度器更新状态。

`miniinfer 里对应：engine.py 的 LLM 类。点模块看说明。`

图 127-1　强调色是这一讲读的部分：执行器、模型的各层、KV Cache 和张量并行的工作进程。

注意

和上一讲一样，本站没有 GPU，这一讲只读源码、没有运行 nano-vllm（提交 `bb823b3`）。下面的交互演示按源码里的公式计算，不是运行结果。

启动：ModelRunner 的构造函数

third_party/nano-vllm/nanovllm/engine/model_runner.py 第 15–48 行 / 共 257 行 ⇆ 对照

```python
class ModelRunner:


    def __init__(self, config: Config, rank: int, event: Event | list[Event]):
        self.config = config
        hf_config = config.hf_config
        self.block_size = config.kvcache_block_size
        self.enforce_eager = config.enforce_eager
        self.world_size = config.tensor_parallel_size
        self.rank = rank
        self.event = event
```

1 **每张卡一个进程，先连成一组** 第 26–27 行

用 NCCL 建一个进程组（第 100 讲的集合通信库），地址写死成本机的 2333 端口，所以只支持单机多卡。每个进程绑定自己那张卡（rank 号就是卡号）。

2 **在 GPU 上直接建模型** 第 28–31 行

3 **装权重，建采样器** 第 32–33 行

4 **预热 → 分 KV → 录图** 第 34–37 行

5 **恢复默认设置** 第 38–39 行

6 **张量并行：共享内存当信箱** 第 41–48 行

← 上一步 | 看整个文件 | 下一步 →

装权重：一个参数一个 weight_loader

HuggingFace 的权重文件里，q、k、v 是三个矩阵，gate、up 是两个矩阵；nano-vllm 的模型里，它们分别合成了一个 `qkv_proj` 和一个 `gate_up_proj`（一次矩阵乘算出三样或两样，少发射几个 kernel）。张量并行时，每张卡还只要其中自己那一片。所以“把文件里的张量拷进参数”这件事，每种层的做法都不一样。

nano-vllm 的办法是：每个参数身上挂一个 `weight_loader` 函数，由定义这个参数的层自己说明怎么装。读权重的代码只负责把名字对上：

third_party/nano-vllm/nanovllm/utils/loader.py 共 28 行 ⇆ 对照

```python
import os
from glob import glob
import torch
from torch import nn
from safetensors import safe_open


def default_weight_loader(param: nn.Parameter, loaded_weight: torch.Tensor):
    param.data.copy_(loaded_weight)
```

1 **逐个读张量** 第 13-16 行

`packed_modules_mapping` 由模型类提供（`models/qwen3.py` 第 187–193 行）：`q_proj` → (`qkv_proj`, "q")，`gate_proj` → (`gate_up_proj`, 0)，等等。

2 **被合并的权重：换个名字，带上“第几份”** 第 17-24 行

3 **其余的权重** 第 25-28 行

← 上一步 | 看整个文件 | 下一步 →

QKV 合并矩阵的 `weight_loader` 是最绕的一个，看懂它，其余的都好懂：

third_party/nano-vllm/nanovllm/layers/linear.py 第 96–128 行 / 共 156 行 ⇆ 对照

```python
class QKVParallelLinear(ColumnParallelLinear):

    def __init__(
        self,
        hidden_size: int,
        head_size: int,
        total_num_heads: int,
        total_num_kv_heads: int | None = None,
        bias: bool = False,
    ):
```

1 **每张卡上的合并矩阵有多大** 第 106–112 行

Qwen3-0.6B 有 16 个查询头、8 个 KV 头，每头 128 维。不切时合并矩阵的输出维是 (16 + 2 × 8) × 128 = 4096。两张卡时，`ColumnParallelLinear` 把输出维除以 2：每张卡 2048 行，放 8 个查询头、4 个 K 头、4 个 V 头。

2 **q、k、v 各放在哪几行** 第 114–125 行

3 **这张卡取文件里的哪一片** 第 126–128 行

← 上一步 | 看整个文件 | 下一步 →

KV Cache 能放多少块

third_party/nano-vllm/nanovllm/engine/model_runner.py 第 91–121 行 / 共 257 行 ⇆ 对照

```
91      def warmup_model(self):
92          torch.cuda.empty_cache()
93          torch.cuda.reset_peak_memory_stats()
94          max_num_batched_tokens, max_model_len = self.config.max_num_batched_tokens,
95          seq_len = min(max_num_batched_tokens, max_model_len)
96          num_seqs = min(max_num_batched_tokens // seq_len, self.config.max_num_seqs)
97          seqs = [Sequence([0] * seq_len) for _ in range(num_seqs)]
98          for seq in seqs:
99              seq.num_scheduled_tokens = seq_len
```

1 **预热：用最大的 batch 跑一次** 第 91–101 行

构造最坏情况的 prefill：一步能放的 token 数（默认 16384）和最长上下文（默认 4096）取小，凑出几个这么长的假请求，跑一遍。目的不是结果，而是让 PyTorch 的显存分配器记下**激活的峰值**。

2 **量显存** 第 106–109 行

3 **一块多大，能放几块** 第 110–113 行

4 **一个大张量，每层挂两个视图** 第 115–121 行

← 上一步 | 看整个文件 | 下一步 →

这一步替代了 miniinfer 里手动指定的 `num_blocks=256`。它让 nano-vllm 在不同的卡上都能“尽量用满显存”，这也是第 99 讲说的：权重和激活之外的显存，越多留给 KV，能同时服务的请求就越多。

拼输入：prepare_prefill 和 prepare_decode

miniinfer 的注意力是“按请求循环”的：`segs` 记下每个请求占哪一段，Python 循环里一个一个算。GPU 上要的是**一个 kernel 处理整个 batch**，所以执行器要把这一步的全部请求描述成几个一维数组，交给 kernel：

- `input_ids`、`positions`：所有请求这一步的 token 和位置，拼成一维。
- `cu_seqlens_q`、`cu_seqlens_k`：前缀和，第 i 个请求的查询是 `[cu_q[i], cu_q[i+1])`，键的个数是 `cu_k[i+1] - cu_k[i]`（含已缓存的部分）。这是 flash-attn 变长接口的约定。
- `slot_mapping`：每个新 token 的 K、V 写到块池的哪个槽位。槽位号 = 块号 × 块大小 + 块内偏移，把整个块池看成一个一维的数组。
- `block_tables`：每个请求的块表，补齐成一个矩形（短的行补 −1）。
- `context_lens`：decode 时每个请求有多少个位置的 K、V。

⇆ 对照

一步的输入：slot_mapping、cu_seqlens 和块表

prefill：两个新请求 | prefill：前缀命中 | 分块 prefill 第二段 | decode：三个请求 | decode：补齐到图的档位

A 有 6 个提示 token，B 有 3 个，都是第一次进来。调度器已经给 A 分了块 5、2，给 B 分了块 7（块不必连续）。这一步把 9 个 token 拼成一维送进去；K、V 刚算出来就紧挨着放在一起，所以注意力直接用这一批的 K、V，不需要块表。



input_ids	[A0, A1, A2, A3, A4, A5, B0, B1, B2]	这一步送进模型的 token（这里用“请求名+位置”表示）
positions	[0, 1, 2, 3, 4, 5, 0, 1, 2]	RoPE 用的位置
cu_seqlens_q	[0, 6, 9]	查询的前缀和：第 i 个请求的查询是 [cu[i], cu[i+1])
cu_seqlens_k	[0, 6, 9]	键的前缀和：每个请求的键包括已缓存的部分
slot_mapping	[20, 21, 22, 23, 8, 9, 28, 29, 30]	每个新 token 的 K、V 写到哪个槽位
block_tables	None	只有查询少于键（有缓存的部分）时才需要

图 127-2　五种情况下拼出来的数组。为了看清，块大小取 4（nano-vllm 是 256）。深色是这一步写入 K、V 的槽位，浅色是已经在缓存里、注意力要读的槽位。

third_party/nano-vllm/nanovllm/engine/model_runner.py 第 123–193 行 / 共 257 行 ⇆ 对照

```
123     def prepare_block_tables(self, seqs: list[Sequence]):
124         max_len = max(len(seq.block_table) for seq in seqs)
125         block_tables = [seq.block_table + [-1] * (max_len - len(seq.block_table)) fo
126         block_tables = torch.tensor(block_tables, dtype=torch.int32, pin_memory=True
127         return block_tables
128
129     def prepare_prefill(self, seqs: list[Sequence]):
130         input_ids = []
131         positions = []
```

1 **块表补齐成矩形** 第 123-127 行

每个请求的块数不同，补 −1 到最长的那个，变成一个 [请求数, 最多块数] 的 int32 张量。`pin_memory` + `non_blocking`：先放进锁页内存，再异步拷到 GPU，CPU 不用等拷完。

2 **prefill：每个请求算 [start, end) 这一段** 第 138-148 行

3 **预热时没有块表** 第 149-150 行

4 **逐块算出写入的槽位** 第 151-161 行

5 **有缓存的部分才传块表** 第 162-163 行

6 **放进全局的 Context** 第 164-170 行

7 **decode：每个请求一个 token** 第 172-188 行

8 **每个请求一个温度** 第 190-193 行

← 上一步 | 看整个文件 | 下一步 →

注意力：一个 Triton kernel，两个 flash-attn 函数

third_party/nano-vllm/nanovllm/layers/attention.py 共 75 行 ⇆ 对照

```python
import torch
from torch import nn
import triton
import triton.language as tl

from flash_attn import flash_attn_varlen_func, flash_attn_with_kvcache
from nanovllm.utils.context import get_context


@triton.jit
```

1 写 KV 的 kernel：一个 token 一个程序 第 10-30 行

启动 N 个程序（N 是这一步的 token 数），第 idx 个程序负责第 idx 个 token：读出它的槽位，**如果是 −1 就直接返回**；否则把这个 token 的 K（KV 头数 × 头维度 = 1024 个数，单卡时）整段读进来，写到块池里第 slot 个位置；V 同样。

块池按 [块, 块内位置, KV 头, 头维度] 存，第 slot 个位置的数据正好从 slot × D 开始，连续 D 个数。所以一个 token 的写入就是一段连续的拷贝，访存是合并的（第 86 讲）。

2 **检查内存布局，然后启动** 第 33-40 行

3 **先写，再算** 第 59-63 行

4 **prefill：变长的 FlashAttention** 第 64-70 行

5 **decode：按块表读 KV** 第 71-74 行

← 上一步 | 看整个文件 | 下一步 →

几个读代码时容易忽略的点：

为什么要自己写 Triton kernel 写 KV。用 PyTorch 写就是一行 `k_cache.view(-1, D)[slot_mapping] = k`。问题出在 −1：CUDA Graph 补齐出来的空位，槽位填的是 −1（下面会看到），而 PyTorch 的下标里 −1 表示“最后一个”，空位的垃圾数据就会写进块池的最后一个槽位，把别人的 KV 覆盖掉。要跳过 −1，在 PyTorch 里得先用布尔掩码挑出有效的行，而布尔掩码选出多少行要等 GPU 算完才知道，CPU 必须同步等待，这在 CUDA Graph 里是不允许的。Triton kernel 里一个 `if slot == -1: return` 就解决了，形状固定，可以录进图里。顺带，K 和 V 在同一个 kernel 里写完，少一次发射。

为什么块大小是 256。`Config` 里有一条 `assert kvcache_block_size % 256 == 0`。这是 flash-attn 分页 KV 接口（传 `block_table` 的用法）的要求：块大小必须是 256 的倍数。上一讲说过这让前缀缓存的粒度变粗；vLLM 的注意力后端支持更小的块（早期版本默认 16，第 129 讲读的提交里默认 64）。

输出层只算需要的位置。`ParallelLMHead`（`layers/embed_head.py` 第 56–66 行）在 prefill 时用 `cu_seqlens_q[1:] - 1` 取出每个请求最后一个位置的向量，只算这几行的打分。decode 时每个请求本来就只有一个位置。这和 miniinfer v1 起的做法一样（第 119 讲）。

CUDA Graph：decode 按 batch 分档录图

第 89 讲讲过 CUDA Graph 解决的问题：decode 一步要发射几百个 kernel，每个都很小，CPU 发射的速度跟不上 GPU 执行的速度，GPU 就在 kernel 之间空等。把一整步提前录成一张图，之后 CPU 只发射一次。

图有一个硬限制：录的时候张量的形状、地址都固定了，重放时只能换内容，不能换形状。而 decode 每一步的 batch 大小都在变。nano-vllm 的办法是**分档**：为几个固定的 batch 大小各录一张图，运行时挑不小于实际 batch 的最小那一档，多出来的行补成空位。

decode 用哪一张图

⇆ 对照

这一步 decode 的请求数 37

max_num_seqs 512



一共 36 张图（1、2、4、8，再每 16 一档到 512），从大到小依次录制，共用一个显存池

37 个请求 → 用 batch **48** 的图，补 11 行空位（slot_mapping 填 -1、context_lens 填 0），多算了 23% 的行。decode 时线性层是读权重受限的，多算几行几乎不花时间；省下的是几百次 kernel 发射。

图 127-3　nano-vllm 录图的档位：1、2、4、8，再每 16 一档到 512，一共 36 张图。实际 batch 用不小于它的那一档，多出来的行写 KV 时跳过（槽位 −1）、注意力时长度为 0。把 max_num_seqs 调成 24 这类不是 16 倍数的值，可以看到源码没有处理的一个边界情况。

third_party/nano-vllm/nanovllm/engine/model_runner.py 第 195–257 行 / 共 257 行 ⇆ 对照

```
195     @torch.inference_mode()
196     def run_model(self, input_ids: torch.Tensor, positions: torch.Tensor, is_prefill
197         if is_prefill or self.enforce_eager or input_ids.size(0) > 512:
198             return self.model.compute_logits(self.model(input_ids, positions))
199         else:
200             bs = input_ids.size(0)
201             context = get_context()
202             graph = self.graphs[next(x for x in self.graph_bs if x >= bs)]
203             graph_vars = self.graph_vars
```

1 **固定的输入缓冲区** 第 222–233 行

按最大的 batch（最多 512）建好一组缓冲区：token、位置、槽位、上下文长度、块表（按最长上下文能有的块数）、输出。录图和重放都用这同一组张量，图里记住的是它们的地址。

2 **档位** 第 234–236 行

3 **从大到小依次录** 第 238–248 行

4 **记下缓冲区** 第 250–257 行

5 **什么时候不用图** 第 196–198 行

6 **重放：拷进缓冲区，跑图，取前 bs 行** 第 199–212 行

← 上一步 | 看整个文件 | 下一步 →

这里读出一个源码没有处理的边界情况：档位从 16 开始每 16 一档，到 `max_num_seqs` 为止。如果 `max_num_seqs` 设成 24，档位就是 1、2、4、8、16，batch 为 17–24 时 `next(x for x in self.graph_bs if x >= bs)` 找不到档位，会抛出 `StopIteration`。默认的 512 是 16 的倍

数，碰不到。改法是在档位列表最后补上 `max_bs` 本身。读开源代码时，留意这种“默认配置下碰不到”的分支，是找 bug 最常见的地方。

张量并行：线性层怎么切，进程之间怎么同步

四种线性层

第 105 讲的 Megatron 切法：一对线性层，第一个**按列切**（每张卡算一部分输出），第二个**按行切**（每张卡用自己那部分输入算出一个部分和），最后 All-Reduce 把部分和加起来。中间的逐元素运算（激活函数）各卡独立做，不用通信。

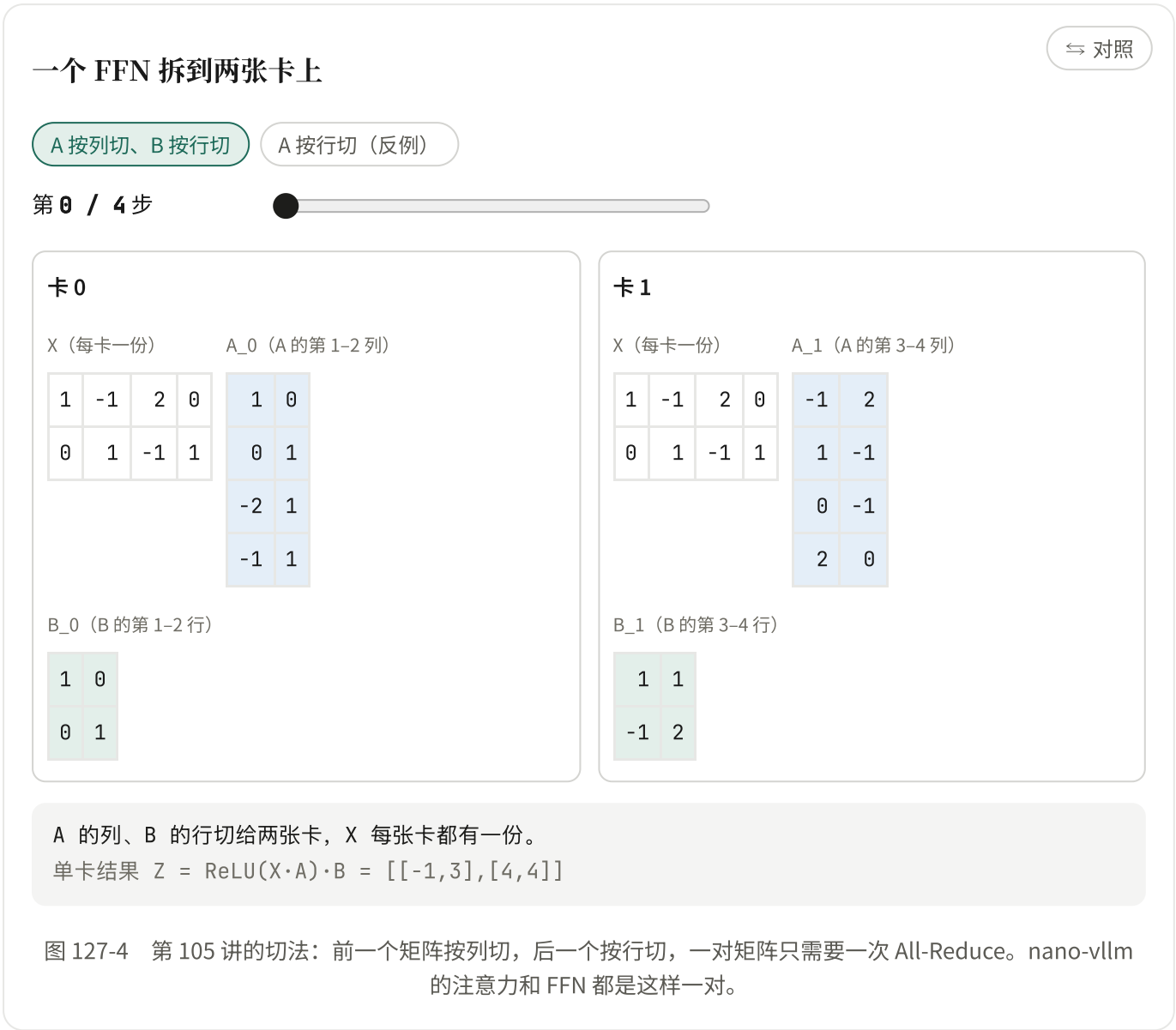


图 127-4　第 105 讲的切法：前一个矩阵按列切，后一个按行切，一对矩阵只需要一次 All-Reduce。nano-vllm 的注意力和 FFN 都是这样一对。

nano-vllm 的 `layers/linear.py` 和 `layers/embed_head.py` 正好把这几种情况各写成一个类：

类	用在哪里	怎么切	通信
`QKVParallelLinear`	q、k、v 投影（合并）	按列（按头）	无
`MergedColumnParallelLinear`	gate、up（合并）	按列	无
`RowParallelLinear`	o 投影、down	按行	All-Reduce
`VocabParallelEmbedding`	嵌入表	按词表	All-Reduce
`ParallelLMHead`	输出层	按词表	gather 到 rank 0

third_party/nano-vllm/nanovllm/layers/linear.py 第 131–156 行 / 共 156 行 ⇆ 对照

```python
class RowParallelLinear(LinearBase):

    def __init__(
        self,
        input_size: int,
        output_size: int,
        bias: bool = False,
    ):
        tp_size = dist.get_world_size()
        super().__init__(divide(input_size, tp_size), output_size, bias, 1)
```

1 **按行切：每张卡只要一部分输入维** 第 133-140 行

o 投影的输入是 16 个头拼起来的 2048 维。两张卡时，每张卡只有自己 8 个头的注意力输出（1024 维），所以权重也只要对应的 1024 列（PyTorch 的权重是 [输出, 输入]，切的是第 1 维）。

2 **装权重：取自己那一段输入维** 第 142-150 行

3 **部分和，All-Reduce** 第 152-156 行

← 上一步 | 看整个文件 | 下一步 →

third_party/nano-vllm/nanovllm/layers/embed_head.py 共 66 行 ⇆ 对照

```python
import torch
from torch import nn
import torch.nn.functional as F
import torch.distributed as dist

from nanovllm.utils.context import get_context


class VocabParallelEmbedding(nn.Module):
```

1 **嵌入表按词表切** 第 34-42 行

15 万个词按编号切成 tp 段，每张卡存一段。查表时，不在自己这段的 token 先换成 0 号（随便查一行），查完再乘一个掩码清成 0；All-Reduce 相加之后，每个 token 正好得到它所在那张卡查出的向量。

2 **输出层：只算需要的行，gather 到 rank 0** 第 56-66 行

← 上一步 | 看整个文件 | 下一步 →

数一下通信：每层两次 All-Reduce（o 投影、down 投影），28 层就是 56 次，加上嵌入表一次、输出层一次 gather，decode 一步 58 次集合通信。每次的数据量是 batch × 1024 个 BF16 数，decode 时很小，主要是延迟（第 102 讲的 α 项）。第 110 讲说过，这正是推理中张量并行最怕慢的互联的原因。

进程之间：共享内存里的“方法名 + 参数”

张量并行的计算需要每张卡在同一时刻执行同一步。nano-vllm 的控制方式非常直接：

third_party/nano-vllm/nanovllm/engine/model_runner.py 第 61–89 行 / 共 257 行 ⇆ 对照

```
61      def loop(self):
62          while True:
63              method_name, args = self.read_shm()
64              self.call(method_name, *args)
65              if method_name == "exit":
66                  break
67
68      def read_shm(self):
69          assert self.world_size > 1 and self.rank > 0
```

1 **其余 rank：一个死循环** 第 61–66 行

等命令，执行，直到收到 `exit`。

2 **读命令** 第 68–74 行

3 **写命令** 第 76–83 行

4 **call：rank 0 先广播，再自己执行** 第 85–89 行

← 上一步 | 看整个文件 | 下一步 →

这种做法简单，代价是**每一步都要在 Python 里序列化、反序列化一次请求列表**，而且每张卡都重复拼一遍输入。vLLM 也是 rank 0 调度、广播给各个 worker，但用的是专门的消息队列，并且尽量只发这一步的增量（第 129 讲）。

几个小的优化

合并矩阵。q、k、v 三个投影合成一个矩阵乘，gate、up 合成一个。矩阵乘的总计算量不变，但 kernel 数从 5 个降到 2 个，读一遍输入就够了。

融合逐元素运算。`RMSNorm`、`SiluAndMul`、`RotaryEmbedding` 的 forward 都加了 `@torch.compile`：PyTorch 把一串逐元素运算编译成一个 kernel，中间结果不写回显存（第 83 讲的算子融合）。其中 RMSNorm 还把**残差相加**融合了进来：

third_party/nano-vllm/nanovllm/layers/layernorm.py 第 28–40 行 / 共 50 行 ⇆ 对照

```
28      @torch.compile
29      def add_rms_forward(
30          self,
31          x: torch.Tensor,
32          residual: torch.Tensor,
33      ) -> tuple[torch.Tensor, torch.Tensor]:
34          orig_dtype = x.dtype
35          x = x.float().add_(residual.float())
36          residual = x.to(orig_dtype)
```

1 一次读写完成“加残差 + 归一化” 第 28–40 行

输入是这一层子模块的输出 x 和残差 residual，返回归一化后的 x 和新的残差 x + residual。不融合时，残差相加要读两个张量、写一个，归一化再读一个、写一个；融合后读两个、写两个。decode 时这些都是读显存受限的运算，少读写一遍就快一截。

对应的，模型的每层不再写 `x = x + h`，而是把残差一路传下去（`models/qwen3.py` 第 146–159 行）。

← 上一步 | 看整个文件 | 下一步 →

在 GPU 上采样。`Sampler` 只有一行核心代码：概率除以一个服从指数分布的随机数，取最大值。

$$\text{argmax}_i \; p_i / E_i, \qquad E_i \sim \text{Exp}(1) \text{ 相互独立} \iff \text{按概率 } p_i \text{ 抽出 } i$$

原因是：E_i / p_i 服从参数为 p_i 的指数分布，几个相互独立的指数分布里，第 i 个最小的概率恰好是 $p_i / \Sigma p_j = p_i$。所以“除以 E_i 取最大”就是“按概率抽样”，而且整个 batch 在 GPU 上一次完成，不用把 15 万维的概率拷回 CPU，也不用 `torch.multinomial`。

nano-vllm 不允许温度为 0（`SamplingParams` 里有 assert），也就不支持纯贪心；想要贪心效果，可以给一个极小的温度。它也没有 top-k、top-p、重复惩罚。

要点

nano-vllm 执行一侧做的事：启动时用预热量出激活峰值，把剩余显存全部切给 KV 块；每一步把请求描述成 cu_seqlens、slot_mapping、块表等一维数组，写 KV 用一个能跳过 −1 的 Triton kernel，prefill 和 decode 各用一个 flash-attn 函数直接按块表读 KV；decode 按 batch 分档录 CUDA Graph，补齐的行靠 −1 和长度 0 变成空操作；张量并行用列切、行切一对线性层，每层两次 All-Reduce，rank 0 通过共享内存广播命令。

回顾

和 miniinfer 对照，nano-vllm 在执行一侧补上了第 125 讲表里的这几项：

第 125 讲的差距	nano-vllm 的做法	代码
注意力逐请求循环、复制 KV	prefill 用变长 FlashAttention，decode 按块表读，一个 kernel 处理整个 batch	`layers/attention.py`
写 KV 逐位置下标	Triton kernel 按 slot_mapping 一次写完，跳过 −1	`layers/attention.py`
每个运算一次发射	decode 分 36 档录 CUDA Graph；合并矩阵；torch.compile 融合逐元素运算	`engine/model_runner.py`
块数手动指定	预热量峰值，剩余显存全给 KV	`engine/model_runner.py`
只有贪心	GPU 上的温度采样	`layers/sampler.py`
不支持多卡	张量并行，共享内存广播命令	`layers/linear.py`、`engine/model_runner.py`

没有补上的：量化、投机解码、top-p 等采样参数、在线服务和流式输出。下一讲动手把第 124 讲的投机解码移植进 nano-vllm，看一个真实的引擎里，加一个功能要动多少地方。下一讲：**动手改 nano-vllm**。

第128讲 动手改 nano-vllm

96 投机解码　124 投机解码　126 nano-vllm 的整体结构　127 nano-vllm 多做了什么

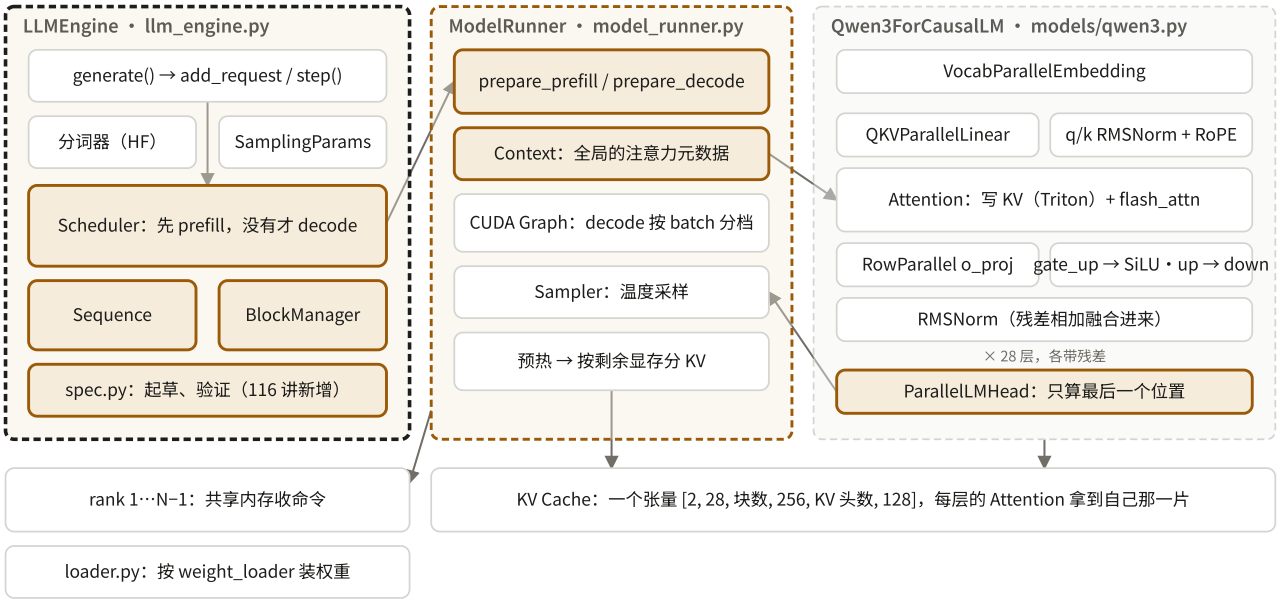
前两讲读完了 nano-vllm。读懂一个引擎最好的检验，是往里面加一个它没有的功能。这一讲把第 124 讲的 n-gram 投机解码移植进去。

在 miniinfer 里，投机解码新增了 spec.py，另外改了 4 个文件：引擎里起草和验证，模型多返回几个位置的打分，调度器和块管理器多留几个位置。到了 nano-vllm，同样的想法要穿过调度器、块管理器、请求的序列化、执行器、Context、输出层、CUDA Graph 的判断：新增 spec.py，另外改 7 个文件。这一讲的重点不在代码量，而在**每一处为什么要改**：一个功能在真实引擎里碰到的，往往是你没想到的那些地方。

nano-vllm 的架构

⇆ 对照

视图：　全部　调度一侧（114）　执行一侧（115）　投机解码的改动（116）



LLMEngine · llm_engine.py（engine/llm_engine.py）：引擎：在主进程里。generate() 把请求放进调度器，然后反复调用 step()：问调度器这一步跑谁 → 交给执行器跑模型、采样 → 把新 token 交回调度器更新状态。

miniinfer 里对应：engine.py 的 LLM 类。点模块看说明。

图 128-1　强调色是要改的模块：调度器（起草、验证、回滚）、请求状态（带上草稿）、块管理器（一次多要几个位置、回滚后退块）、执行器（新的拼输入函数）、Context 和输出层（每个位置都要打分），以及新增的 spec.py。

注意

本站没有 GPU，这一讲的修改**没有运行验证过**，只做了语法检查。下文最后给出在 GPU 上验证的方法和脚本。读的时候请把它当成一份经过仔细推敲、但还没跑过的补丁：逻辑上的每一步都对照了源码，但真正的 bug 往往要跑起来才看得见。

先想清楚六个问题

动手之前，先回答六个设计问题。每个问题的答案都对应 nano-vllm 里一处具体的约束。

1. **草稿放在哪里？** 放在 `Sequence` 的一个新字段 `draft` 里，**不放进 `token_ids`**。`token_ids` 牵动太多东西：`num_tokens`、已生成的数量、前缀缓存的哈希（`seq.block(i)` 直接切 `token_ids`）。草稿随时可能被拒绝，放进去再删，就得保证这期间没有任何代码读到它们。单独放，验证完再把接受的部分 `append_token` 进去，其余代码看到的永远是真实的 token。

2. **验证走哪条路？** decode 的那条路（`flash_attn_with_kvcache` + CUDA Graph）假定每个请求正好 1 个查询。验证要送“最后一个真实 token + k 个草稿”，每个请求 1 + k 个查询。nano-vllm 里正好有一条现成的路能处理“查询比键少、键在块池里”：**带前缀缓存的 prefill**（第 127 讲）。验证就是一次这样的 prefill：已缓存的是前面的全部 token，这一步算最后一个真实 token 和草稿。所以新写一个 `prepare_verify`，拼出 cu_seqlens 和块表，走 `flash_attn_varlen_func`。代价是验证步不能用 CUDA Graph。

3. **输出层算哪些位置？** 原来 prefill 只算每个请求最后一个位置的打分。验证要的是每个送入位置的打分（第 j 个位置的打分用来验证第 j + 1 个草稿）。在 `Context` 里加一个标志 `all_logits`，输出层看到它就不再只取最后一个位置。

4. **有温度时怎么接受？** miniinfer 只有贪心，“草稿等于贪心预测”就接受。nano-vllm 只有温度采样。好在 n-gram 草稿是**确定的**（草稿分布在 d 上的概率是 1），这时第 96 讲的接受-拒绝规则会简化成一个非常简单的形式：

> 在每个位置照常采样 x ~ p；x = d 就接受 d，否则输出 x，丢弃后面的位置

为什么分布不变：第 96 讲的规则是以 $\min(1, p(d) / q(d)) = p(d)$ 的概率接受；拒绝时从 $(p - q)_+$ 归一化后的分布里重新采样，也就是 p 去掉 d 之后再归一化。而“采样 x ~ p，看 x 是否等于 d”：x = d 的概率正好是 p(d)；x ≠ d 时，x 的分布正好是 p 去掉 d 再归一化。两者完全相同。

所以 `verify` 函数一个字都不用改：把“贪心预测”换成“采样出的 token”就行。nano-vllm 的采样器对每个位置独立加噪声，正好满足要求。

5. 块怎么管？草稿要占位置，可能跨进新的块。原来的 `can_append` / `may_append` 假定每步只多一个 token（第 126 讲），要改成“要放下 len + k 个位置，还差几块”。块不够时，**先放弃草稿，再考虑抢占别人**：草稿只是可能的加速，不值得把别的请求赶出去。验证之后，被拒绝的草稿如果独占了末尾的块，把块还回去，让块表重新等于请求的实际块数（`prepare_decode` 用块表的最后一块算写入槽位，多一块就写错地方）。

6. 回滚和前缀缓存怎么协调？和 miniinfer 一样，回滚只是让 `num_cached_tokens` 只推进到有效的位置。但 nano-vllm 的 `hash_blocks` 用 `num_cached_tokens + num_scheduled_tokens` 决定登记哪些块，所以验证之后要先把 `num_scheduled_tokens` 改成**有效的位置数**，再登记；而且登记必须在接受的 token 追加进 `token_ids` **之后**，否则算哈希时块里的 token 还不全。

还有两处不改会出错、却很容易漏掉的地方，后面讲到时再说：跨进程序列化请求时要带上草稿；`run_model` 判断要不要用 CUDA Graph 的那个条件。

改了哪些地方

```
$ .venv/bin/python bench/nano_spec_diff.py
```

⇆ 对照

```
  +1   -0  config.py
 +17   -4  engine/block_manager.py
 +38   -3  engine/model_runner.py
 +42  -11  engine/scheduler.py
  +3   -2  engine/sequence.py
 +27   -0  engine/spec.py（新文件）
  +1   -1  layers/embed_head.py
  +3   -2  utils/context.py
+132  -23  合计 8 个文件
```

图 128-2　每个文件增删了多少行（只比较源码文本）。新增 132 行，其中 27 行是新文件 spec.py；改动最大的是调度器和执行器。

修改后的文件放在 `miniinfer/nano_spec/nanovllm/`，目录结构和 nano-vllm 相同，覆盖过去就能用。

配置、Context、输出层：各一两行

nano_spec/nanovllm/config.py 对比 third_party/nano-vllm/nanovllm/config.py：+1 行 / −0 行 ⇆ 对照

```
      …… 省略 16 行没变的代码
17        kvcache_block_size: int = 256
18        num_kvcache_blocks: int = -1
19 +      num_spec_tokens: int = 0           # 每步最多给每个请求起草几个 token，0 表示不投机
20
21        def __post_init__(self):
      …… 省略 5 行没变的代码
```

图 128-3　Config 多一个字段 num_spec_tokens，默认 0 表示不投机。LLMEngine 会把同名的关键字参数筛进来，所以用法是 LLM(path, num_spec_tokens=4)，引擎的代码不用改。

nano_spec/nanovllm/utils/context.py ⇆ 对照

对比 third_party/nano-vllm/nanovllm/utils/context.py：+3 行 / −2 行

```
      …… 省略 12 行没变的代码
13        context_lens: torch.Tensor | None = None
14        block_tables: torch.Tensor | None = None
15 +      all_logits: bool = False           # 验证草稿时：每个送入位置都要打分，而不只是每个请求
16
17    _CONTEXT = Context()
      …… 省略 2 行没变的代码
20        return _CONTEXT
21
22 + def set_context(is_prefill, cu_seqlens_q=None, cu_seqlens_k=None, max_seqlen_q=0, r
21 - def set_context(is_prefill, cu_seqlens_q=None, cu_seqlens_k=None, max_seqlen_q=0, r
23        global _CONTEXT
24 +      _CONTEXT = Context(is_prefill, cu_seqlens_q, cu_seqlens_k, max_seqlen_q, max_se
23 -      _CONTEXT = Context(is_prefill, cu_seqlens_q, cu_seqlens_k, max_seqlen_q, max_se
25
26    def reset_context():
      …… 省略 2 行没变的代码
```

图 128-4　Context 多一个 all_logits 标志，set_context 多一个参数，默认 False，原有的调用都不受影响。

nano_spec/nanovllm/layers/embed_head.py ⇆ 对照

对比 third_party/nano-vllm/nanovllm/layers/embed_head.py：+1 行 / −1 行

```
      …… 省略 55 行没变的代码
 56       def forward(self, x: torch.Tensor):
 57           context = get_context()
 58 +         if context.is_prefill and not context.all_logits:
 58 -         if context.is_prefill:
 59               last_indices = context.cu_seqlens_q[1:] - 1
 60               x = x[last_indices].contiguous()
      …… 省略 6 行没变的代码
```

图 128-5　输出层：验证草稿时（is_prefill 且 all_logits），每个送入位置都算打分。

请求状态：草稿要跟着请求跨进程

nano_spec/nanovllm/engine/sequence.py ⇆ 对照

对比 third_party/nano-vllm/nanovllm/engine/sequence.py：+3 行 / −2 行

```
      …… 省略 26 行没变的代码
 27           self.is_prefill = True
 28           self.block_table = []
 29 +         self.draft = []                    # 这一步的草稿 token：只在一步之内存在，不进 t
 30           self.temperature = sampling_params.temperature
 31           self.max_tokens = sampling_params.max_tokens
      …… 省略 41 行没变的代码
 73       def __getstate__(self):
 74           last_state = self.last_token if not self.is_prefill else self.token_ids
 75 +         return (self.num_tokens, self.num_prompt_tokens, self.num_cached_tokens, se
 74 -         return (self.num_tokens, self.num_prompt_tokens, self.num_cached_tokens, se
 76
 77       def __setstate__(self, state):
 78 +         self.num_tokens, self.num_prompt_tokens, self.num_cached_tokens, self.num_s
 77 -         self.num_tokens, self.num_prompt_tokens, self.num_cached_tokens, self.num_s
 79           if isinstance(last_state, list):
 80               self.token_ids = last_state
      …… 省略 4 行没变的代码
```

图 128-6　Sequence 多一个 draft 字段；序列化时也带上它。

第二处改动很容易漏掉。第 126 讲说过，张量并行时 rank 0 每一步把请求列表 pickle 之后发给其他卡，`__getstate__` 只发必要的字段，decode 时连 `token_ids` 都不发。如果不把 `draft` 加进去，其他卡上的请求 `draft` 是空的：rank 0 送进模型的是 1 + k 个 token，其他卡只送 1 个，张量形状对不上，All-Reduce 会卡住或报错。**单卡测试完全发现不了这个 bug，一开张量并行就出问题。**

块管理器：一次多要几个位置，回滚后退块

nano_spec/nanovllm/engine/block_manager.py ⇆ 对照

对比 third_party/nano-vllm/nanovllm/engine/block_manager.py：+17 行 / −4 行

```
      …… 省略 100 行没变的代码
101           seq.block_table.clear()
102
103 +     def num_missing_blocks(self, seq: Sequence, extra: int) → int:
104 +         # 要放下 len(seq) + extra 个位置（extra 是草稿数），块表还差几块。extra = 0 时就是
105 +         need = (len(seq) + extra + self.block_size - 1) // self.block_size
106 +         return max(0, need - len(seq.block_table))
107 +
108 +     def can_append(self, seq: Sequence, extra: int = 0) → bool:
109 +         return len(self.free_block_ids) ⩾ self.num_missing_blocks(seq, extra)
103 -     def can_append(self, seq: Sequence) → bool:
104 -         return len(self.free_block_ids) ⩾ (len(seq) % self.block_size == 1)
110
111 +     def may_append(self, seq: Sequence, extra: int = 0):
112 +         for _ in range(self.num_missing_blocks(seq, extra)):
106 -     def may_append(self, seq: Sequence):
107 -         if len(seq) % self.block_size == 1:
113               seq.block_table.append(self._allocate_block())
114
115 +     def trim(self, seq: Sequence):
116 +         # 回滚之后：末尾只装了被拒草稿的块还给空闲列表，让块表重新等于 seq.num_blocks 块
117 +         while len(seq.block_table) > seq.num_blocks:
118 +             block_id = seq.block_table.pop()
119 +             self.blocks[block_id].ref_count -= 1
120 +             if self.blocks[block_id].ref_count == 0:
121 +                 self._deallocate_block(block_id)
122 +
123       def hash_blocks(self, seq: Sequence):
124           start = seq.num_cached_tokens // self.block_size
      …… 省略 9 行没变的代码
```

图 128-7　原来的“长度除以块大小余 1 就要一块新的”换成一般的“要放下 len + extra 个位置还差几块”；新增 trim，回滚后把多出来的块还回去。

nano_spec/nanovllm/engine/block_manager.py 第 103–121 行 / 共 133 行 ⇆ 对照

```
103     def num_missing_blocks(self, seq: Sequence, extra: int) -> int:
104         # 要放下 len(seq) + extra 个位置（extra 是草稿数），块表还差几块。extra = 0 时就是原
105         need = (len(seq) + extra + self.block_size - 1) // self.block_size
106         return max(0, need - len(seq.block_table))
107
108     def can_append(self, seq: Sequence, extra: int = 0) -> bool:
109         return len(self.free_block_ids) >= self.num_missing_blocks(seq, extra)
110
111     def may_append(self, seq: Sequence, extra: int = 0):
```

1 还差几块 第 103-106 行

放下 len(seq) + extra 个位置要 ⌈(len + extra) ÷ 块大小⌉ 块，减去块表里已有的。extra = 0 时：块表原来覆盖前 len − 1 个位置，如果 len 除以块大小余 1，最后一个 token 正好在新的一块里，差 1 块，否则差 0 块，和原来的判断完全一样。

2 **can_append / may_append 多一个参数** 第 108-113 行

3 **trim：退还只装了被拒草稿的块** 第 115-121 行

← 上一步 | 看整个文件 | 下一步 →

调度器：起草、验证、回滚

nano_spec/nanovllm/engine/scheduler.py

⇆ 对照

对比 third_party/nano-vllm/nanovllm/engine/scheduler.py：+42 行 / −11 行

```
      …… 省略 3 行没变的代码
 4    from nanovllm.engine.sequence import Sequence, SequenceStatus
 5    from nanovllm.engine.block_manager import BlockManager
 6 +  from nanovllm.engine.spec import propose, verify
 7
 8
      …… 省略 6 行没变的代码
15            self.block_size = config.kvcache_block_size
16            self.block_manager = BlockManager(config.num_kvcache_blocks, config.kvcache
17 +          self.num_spec_tokens = config.num_spec_tokens
18 +          self.num_accepted = 0                       # 一共接受了多少个草稿 token
19            self.waiting: deque[Sequence] = deque()
20            self.running: deque[Sequence] = deque()
      …… 省略 37 行没变的代码
58                return scheduled_seqs, True
59
60 +          # decode (可以带草稿)
57 -          # decode
61            while self.running and len(scheduled_seqs) < self.max_num_seqs:
62                seq = self.running.popleft()
63 +              seq.draft = self.propose(seq, self.max_num_batched_tokens - num_batched
64 +              while not self.block_manager.can_append(seq, len(seq.draft)):
65 +                  if seq.draft:
66 +                      seq.draft = []                      # 块不够：先放弃草稿，再考虑抢占别
67 +                  elif self.running:
60 -              while not self.block_manager.can_append(seq):
61 -                  if self.running:
68                        self.preempt(self.running.pop())
69                    else:
      …… 省略 1 行没变的代码
71                        break
72                else:
73 +                  seq.num_scheduled_tokens = 1 + len(seq.draft)
74 +                  num_batched_tokens += seq.num_scheduled_tokens
67 -                  seq.num_scheduled_tokens = 1
75                    seq.is_prefill = False
76 +                  self.block_manager.may_append(seq, len(seq.draft))
69 -                  self.block_manager.may_append(seq)
77                    scheduled_seqs.append(seq)
78            assert scheduled_seqs
      …… 省略 1 行没变的代码
80            return scheduled_seqs, False
81
82 +      def propose(self, seq: Sequence, budget: int) -> list[int]:
83 +          # 草稿数不超过：设定的上限、这个请求还能生成的数量减一、这一步剩下的 token 预算减一
84 +          k = min(self.num_spec_tokens, seq.max_tokens - seq.num_completion_tokens -
85 +          return propose(seq.token_ids, k) if k > 0 else []
86 +
```

```
87        def preempt(self, seq: Sequence):
88            seq.status = SequenceStatus.WAITING
      ····· 省略 3 行没变的代码
92
93        def postprocess(self, seqs: list[Sequence], token_ids: list[int], is_prefill: b
94 +          i = 0
95 +          for seq in seqs:
96 +              draft, seq.draft = seq.draft, []
97 +              pred = token_ids[i:i + 1 + len(draft)]    # 这个请求的 1 + 草稿数 个位置上
98 +              i += 1 + len(draft)
99 +              if is_prefill and seq.num_cached_tokens + seq.num_scheduled_tokens < se
100 +                 self.block_manager.hash_blocks(seq)   # 分块 prefill 的中间一段：不出
101 +                 seq.num_cached_tokens += seq.num_scheduled_tokens
102 +                 seq.num_scheduled_tokens = 0
103 +                 continue
104 +             new = verify(draft, pred)                 # 没有草稿时就是 [pred[0]]，和原
105 +             self.num_accepted += len(new) - 1
106 +             finished = False
107 +             for n, token_id in enumerate(new, 1):
108 +                 seq.append_token(token_id)
109 +                 if (not seq.ignore_eos and token_id == self.eos) or seq.num_comple
110 +                     finished = True
111 +                     break
112 +             if not is_prefill:
113 +                 # 回滚：KV 有效的只有最后一个真实 token 和被接受（并且追加了）的草稿
114 +                 seq.num_scheduled_tokens = min(n + 1, len(new))
115 +             self.block_manager.hash_blocks(seq)       # 要在追加之后：被接受的草稿所在的
82  -          for seq, token_id in zip(seqs, token_ids):
83  -              self.block_manager.hash_blocks(seq)
116             seq.num_cached_tokens += seq.num_scheduled_tokens
117             seq.num_scheduled_tokens = 0
118 +           if finished:
86  -           if is_prefill and seq.num_cached_tokens < seq.num_tokens:
87  -               continue
88  -           seq.append_token(token_id)
89  -           if (not seq.ignore_eos and token_id == self.eos) or seq.num_completion_
119                 seq.status = SequenceStatus.FINISHED
120                 self.block_manager.deallocate(seq)
121                 self.running.remove(seq)
122 +           else:
123 +               self.block_manager.trim(seq)
```

图 128-8　scheduler.py 的改动：decode 时起草并按草稿数要块，新增 propose，postprocess 改成逐请求验证、追加、回滚。

nano_spec/nanovllm/engine/scheduler.py 第 60–123 行 / 共 123 行 ⇆ 对照

```
60          # decode（可以带草稿）
61          while self.running and len(scheduled_seqs) < self.max_num_seqs:
62              seq = self.running.popleft()
63              seq.draft = self.propose(seq, self.max_num_batched_tokens - num_batched_
64              while not self.block_manager.can_append(seq, len(seq.draft)):
65                  if seq.draft:
66                      seq.draft = []                          # 块不够：先放弃草稿，再考虑抢占别人
67                  elif self.running:
68                      self.preempt(self.running.pop())
```

1 **每个 decode 的请求先起草** 第 62-63 行

用 n-gram 在这个请求已有的 token 里找草稿（第 124 讲）。起草是纯 Python 的列表比较，在调度器里做，执行器不用知道草稿是怎么来的。

2 **块不够：先放弃草稿，再抢占** 第 64-71 行

3 **这一步算 1 + 草稿数个位置** 第 72-77 行

4 **草稿数的三个上限** 第 82-85 行

5 **取出这个请求的那几个预测** 第 94-98 行

6 **分块 prefill 的中间一段：照旧** 第 99-103 行

7 **验证，追加** 第 104-111 行

8 **回滚：有效的位置数** 第 112-114 行

9 **先追加，再登记，再推进** 第 115-117 行

10 **结束，或者退还多余的块** 第 118-123 行

执行器：新的拼输入函数，和一个容易漏的条件

nano_spec/nanovllm/engine/model_runner.py ⇆ 对照

对比 third_party/nano-vllm/nanovllm/engine/model_runner.py：+38 行 / −3 行

```
      ····· 省略 187 行没变的代码
188           return input_ids, positions
189
190 +     def prepare_verify(self, seqs: list[Sequence]):
191 +         # 带草稿的 decode：每个请求送入“最后一个真实 token + 草稿”，按带前缀缓存的 prefill
192 +         input_ids = []
193 +         positions = []
194 +         cu_seqlens_q = [0]
195 +         cu_seqlens_k = [0]
196 +         max_seqlen_q = 0
197 +         max_seqlen_k = 0
198 +         slot_mapping = []
199 +         for seq in seqs:
200 +             start = len(seq) - 1                        # 最后一个真实 token 的位置
201 +             ids = [seq.last_token] + seq.draft
202 +             end = start + len(ids)
203 +             input_ids.extend(ids)
204 +             positions.extend(range(start, end))
205 +             cu_seqlens_q.append(cu_seqlens_q[-1] + len(ids))
206 +             cu_seqlens_k.append(cu_seqlens_k[-1] + end)
207 +             max_seqlen_q = max(len(ids), max_seqlen_q)
208 +             max_seqlen_k = max(end, max_seqlen_k)
209 +             for p in range(start, end):
210 +                 slot_mapping.append(seq.block_table[p // self.block_size] * self.b
211 +         block_tables = self.prepare_block_tables(seqs)
212 +         input_ids = torch.tensor(input_ids, dtype=torch.int64, pin_memory=True).cu
213 +         positions = torch.tensor(positions, dtype=torch.int64, pin_memory=True).cu
214 +         cu_seqlens_q = torch.tensor(cu_seqlens_q, dtype=torch.int32, pin_memory=Tr
215 +         cu_seqlens_k = torch.tensor(cu_seqlens_k, dtype=torch.int32, pin_memory=Tr
216 +         slot_mapping = torch.tensor(slot_mapping, dtype=torch.int32, pin_memory=Tr
217 +         set_context(True, cu_seqlens_q, cu_seqlens_k, max_seqlen_q, max_seqlen_k, s
218 +         return input_ids, positions
219 +
220       def prepare_sample(self, seqs: list[Sequence]):
221 +         temperatures = [seq.temperature for seq in seqs for _ in range(1 + len(seq
191 -         temperatures = [seq.temperature for seq in seqs]
222           temperatures = torch.tensor(temperatures, dtype=torch.float32, pin_memory=
223           return temperatures
      ····· 省略 1 行没变的代码
225       @torch.inference_mode()
226       def run_model(self, input_ids: torch.Tensor, positions: torch.Tensor, is_prefi
227 +         if get_context().is_prefill or self.enforce_eager or input_ids.size(0) > 5
197 -         if is_prefill or self.enforce_eager or input_ids.size(0) > 512:
228               return self.model.compute_logits(self.model(input_ids, positions))
229           else:
      ····· 省略 13 行没变的代码
243
244       def run(self, seqs: list[Sequence], is_prefill: bool) -> list[int]:
```

```
245 +         if is_prefill:
246 +             input_ids, positions = self.prepare_prefill(seqs)
247 +         elif any(seq.draft for seq in seqs):
248 +             input_ids, positions = self.prepare_verify(seqs)
249 +         else:
250 +             input_ids, positions = self.prepare_decode(seqs)
215 -         input_ids, positions = self.prepare_prefill(seqs) if is_prefill else self.p
251           temperatures = self.prepare_sample(seqs) if self.rank == 0 else None
252           logits = self.run_model(input_ids, positions, is_prefill)
      ····· 省略 40 行没变的代码
```

图 128-9　model_runner.py 的改动：新增 prepare_verify；温度按位置展开；run_model 判断是否用 CUDA Graph 的条件改成看 Context；run 按有没有草稿选择拼输入的函数。

nano_spec/nanovllm/engine/model_runner.py 第 190–255 行 / 共 292 行 ⇆ 对照

```
190     def prepare_verify(self, seqs: list[Sequence]):
191         # 带草稿的 decode：每个请求送入“最后一个真实 token + 草稿”，按带前缀缓存的 prefill 那
192         input_ids = []
193         positions = []
194         cu_seqlens_q = [0]
195         cu_seqlens_k = [0]
196         max_seqlen_q = 0
197         max_seqlen_k = 0
198         slot_mapping = []
199         for seq in seqs:
```

1 **每个请求：最后一个真实 token + 草稿** 第 199-208 行

起点是最后一个真实 token 的位置 len − 1，这一步算 [len − 1, len − 1 + 1 + k)。查询有 1 + k 个，键有 len + k 个（前面的全部，加上这一步的）。和 `prepare_prefill` 处理“前缀缓存命中”的情况一模一样，只是这里的“已缓存”是前面全部的 token。

没有草稿的请求（这一步没找到草稿）也走这里，就是 1 个查询，混在一起没有问题。

2 **逐个位置算槽位** 第 209-210 行

3 **一定传块表，打开 all_logits** 第 211-218 行

4 **温度按位置展开** 第 220-223 行

5 **容易漏的条件：看 Context，而不是参数** 第 226-228 行

6 **三选一** 第 244-250 行

← 上一步 | 看整个文件 | 下一步 →

怎么验证

在有 NVIDIA GPU 的机器上：克隆 nano-vllm 并切到 `bb823b3`，按它的 README 装好依赖，把 `miniinfer/nano_spec/nanovllm/` 下的 8 个文件覆盖到同名位置，然后运行 `miniinfer/nano_spec/check_spec.py`：

```
python check_spec.py ~/huggingface/Qwen3-0.6B 0
```

```
python check_spec.py ~/huggingface/Qwen3-0.6B 4
```

第一次关闭投机，第二次每步最多 4 个草稿，三个请求各生成 256 个 token，温度取 10^{-5}（效果接近贪心）。第二次运行会打印吞吐、接受的草稿数，以及每个请求和不投机时“前多少个 token 相同”。

为什么不直接要求“完全相同”：在 miniinfer 里，我们在 CPU 上用 FP32，投机和不投机走的是同一段注意力代码，结果逐 token 相同（第 124 讲）。在 GPU 上，验证步走 `flash_attn_varlen_func`，普通 decode 走 `flash_attn_with_kvcache` 和 CUDA Graph，batch 的形状也不同，BF16 下打分会有末位的差异。两个候选的打分很接近时，选择可能翻转，之后的内容就会不同（第 123 讲量化时见过同样的现象）。所以合理的预期是：大多数请求全部相同，少数在很靠后的位置开始不同；如果从前几个 token 就开始不同，那就是移植有 bug。

还应该测的：

- **张量并行**：`LLM(path, num_spec_tokens=4, tensor_parallel_size=2)`，检验草稿的序列化。
- **块很紧的时候**：把 `gpu_memory_utilization` 调小，让抢占和“先放弃草稿”的分支真正跑到。
- **前缀缓存**：同一个提示跑两遍，第二遍应当命中；投机产生的块登记不能出错。
- **长度上限和结束符**：`max_tokens` 取一个很小的数，检查生成的 token 数正好是它。

在 GPU 上能快多少

第 124 讲在 CPU 上，照抄型任务每次前向出 3.66 个 token，用时只降到七成：CPU 上多算 4 个位置并不便宜。GPU 上小 batch 的 decode 是读权重受限的，多算几个位置几乎不加时间，所以同样的接受率，加速比会更接近“每次前向出几个 token”。但这份移植有两个新的代价：验证步不能用 CUDA Graph，每步多了几百次 kernel 发射的开销（第 89 讲）；调度器每步要在 Python 里为每个请求做一次 n-gram 查找。batch 很大时，decode 本身已经接近计算受限，多算的草稿位置不

再免费，投机反而可能变慢。vLLM 的做法是把草稿的验证也录进 CUDA Graph，并在负载高时减少草稿数（第 129 讲）。这些都要实测才知道，本站没有数字。

要点

把投机解码移植进 nano-vllm：草稿单独放在请求的 draft 字段里，不进 token_ids；验证复用“带前缀缓存的 prefill”那条路（变长 FlashAttention + 块表），输出层每个位置都打分；n-gram 草稿是确定的，“每个位置照常采样、等于草稿就接受”就是精确的接受-拒绝采样；块不够先放弃草稿再抢占，验证后退还多余的块；回滚时先追加被接受的 token，再按有效位置数登记哈希、推进 num_cached_tokens。另有两处不改就出错：序列化要带上草稿（张量并行），判断用不用 CUDA Graph 要看 Context。

回顾

同一个投机解码，miniinfer 里新增 1 个文件、改 4 个，nano-vllm 里新增 1 个、改 7 个，一共新增 132 行。多出来的改动几乎都来自 nano-vllm 为了快而做的设计：调度和执行分开（草稿要在两边都看得见）、多进程张量并行（草稿要序列化）、decode 专用的 kernel 和 CUDA Graph（验证要换一条路走）、prefill 只算最后一个位置的输出层（要加开关）、按块登记的前缀缓存（回滚要和登记的顺序配合）。这也是读大型引擎源码时的常态：一个功能的核心逻辑往往只有几十行，难的是它和已有的每一项优化怎么共存。

第 16 篇到这里结束。第 17 篇去读真正的工业级引擎：vLLM 怎么把 prefill 和 decode 统一调度、怎么做投机解码和量化 kernel，SGLang 的 RadixAttention 和零开销调度。下一讲：**vLLM 的架构**。

AI INFRA · HANDBOOK

第 17 篇

主流推理引擎

收录 5 讲 · 第 129–133 讲

第129讲 vLLM 的架构

重点

74 Continuous Batching　75 PagedAttention　76 Prefix Cache　77 Chunked Prefill
81 请求生命周期　126 nano-vllm 的整体结构　127 nano-vllm 多做了什么

第 16 篇读的 nano-vllm 约 1200 行，是 vLLM 核心思想的一个小样。这一讲读 vLLM 本身。规模先摆出来：

```
$ .venv/bin/python bench/engine_loc.py                    ⇆ 对照

nano-vllm    vLLM  SGLang  文件
       80    3569    6176  调度器
      102     823     721  块 / 前缀缓存
       65     401    3204  请求状态
      235    2368    2460  执行器

vLLM vllm/v1/ 整个目录: 170876 行
SGLang srt/managers/ 整个目录: 43063 行
nano-vllm 全部: 1197 行
```

图 129-1　三个引擎几个关键文件的行数（不算空行）。vLLM 的调度器一个文件就有 3569 行，是 nano-vllm 整个仓库的三倍；vllm/v1/ 一个目录就有 17 万行。SGLang 只统计了取下来的几个目录。

17 万行不可能逐行读，也没有必要。这一讲的读法是：**抓住一个请求的路径，在路径上的每个关键点停下来，看它和 nano-vllm、miniinfer 的做法有什么不同、为什么不同**。多出来的代码大部分是在支持更多的模型、硬件、并行方式和功能（多模态、LoRA、PD 分离、几十种注意力后端），核心的骨架并不大。

注意

本篇（第 129–133 讲）只读源码，没有运行这些引擎（本站没有 GPU）。读的是 2026 年 11 月 9 日 main 分支上的 vLLM（提交 `821851e`，Apache-2.0 许可）和 SGLang（提交 `48da6bd`，Apache-2.0 许可），只用 `git sparse-checkout` 取下了要读的几个目录。这两个项目改得很快，几个月后行号、甚至函数名都可能不一样，但这里讲的结构变化很慢。

取下同样的代码：

```
cd miniinfer/third_party
git clone --filter=blob:none --sparse https://github.com/vllm-project/vllm && cd
vllm
git checkout 821851e && git sparse-checkout set --no-cone '/vllm/v1/'
'/vllm/engine/' '/vllm/config/' '/vllm/entrypoints/llm.py'
'/vllm/entrypoints/openai/'
```

三个进程



图 129-2　vLLM V1 的进程结构。点“下一步”看一个请求经过哪些模块，点模块看它在哪个文件。

nano-vllm 只有一个主进程（张量并行时再加几个 worker），调度、分词、执行模型都在主进程里轮流做。vLLM V1 把它们拆到三类进程里：

1. **API 服务器进程**：HTTP 接口、分词、反分词、流式输出。这些都是纯 CPU 的 Python 代码，而且量很大：一个高并发的服务每秒要处理成千上万个 token 的反分词。
2. **EngineCore 进程**：只做调度和驱动执行。它是一个死循环：收请求 → 调度 → 让 worker 执行 → 更新状态 → 发输出。
3. **Worker 进程**：每张卡一个，执行模型。

为什么拆？第 89 讲引过 vLLM 团队对 v0.6.0 的剖析：API 服务器和调度一度占了一步时间的六成，GPU 大部分时间在等 Python。拆成多个进程以后，分词、反分词和调度跑在不同的 CPU 核上，互不阻塞；Python 的全局解释器锁（GIL）也只锁住各自的进程。进程之间用 ZMQ 传消息、msgpack 序列化；EngineCore 里收和发又各放在一个后台线程，主循环只管调度。

third_party/vllm/vllm/v1/engine/core.py 第 1794–1873 行 / 共 3143 行 ⇆ 对照

```
1794      def run_busy_loop(self):
1795          """Core busy loop of the EngineCore."""
1796          while self._handle_shutdown():
1797              # 1) Poll the input queue until there is work to do.
1798              self._process_input_queue()
1799              # Publish request counts before and after GPU step to ensure freshness.
1800              self._maybe_publish_request_counts()
1801              # 2) Step the engine core and return the outputs.
1802              self._process_engine_step()
```

1 **EngineCore 的主循环** 第 1794–1805 行

两件事交替：处理输入队列里的请求（新请求、取消、控制命令），执行一步。和 nano-vllm 的 `generate()` 循环是一回事，只是这里永不退出，请求随时从外面来。

2 **没活干时阻塞等待** 第 1820–1853 行

3 **执行一步，把输出放进输出队列** 第 1855–1873 行

← 上一步 | 看整个文件 | 下一步 →

third_party/vllm/vllm/v1/engine/core.py 第 713–743 行 / 共 3143 行 ⇆ 对照

```
713     def step(self) -> tuple[dict[int, EngineCoreOutputs], bool]:
714         """Schedule, execute, and make output.
715
716         Returns tuple of outputs and a flag indicating whether the model
717         was executed.
718         """
719         # Check for any requests remaining in the scheduler - unfinished,
720         # or finished and not yet removed from the batch.
721         if not self.scheduler.has_requests():
```

1 调度 第 721–723 行

和 nano-vllm 一样的三段：调度 → 执行 → 更新。`schedule()` 返回一个 `SchedulerOutput`，下面细讲。

2 **执行：非阻塞地发出去，再等结果** 第 724–732 行

3 **更新状态** 第 736–743 行

← 上一步 | 看整个文件 | 下一步 →

启动：KV Cache 有多大，是量出来的

请求进来之前，EngineCore 要先把 worker 准备好。顺序和 nano-vllm 一样（第 127 讲）：建模型、装权重、量显存、分 KV、录 CUDA Graph；不同的是，这些步骤由 EngineCore 进程通过执行器指挥所有 worker 一起做，而“能分多少个块”的决定，是在 EngineCore 里统一做出的。

third_party/vllm/vllm/v1/engine/core.py 第 273–407 行 / 共 3143 行 ⇆ 对照

```
273     def _initialize_kv_caches(self, vllm_config: VllmConfig) -> KVCacheConfig:
274         start = time.time()
275
276         # register all kvcache specs in enginecore process.
277         register_all_kvcache_specs(vllm_config)
278
279         # Get all kv cache needed by the model
280         kv_cache_specs = self.model_executor.get_kv_cache_specs()
281
```

1 **问 worker：每一层要什么样的 KV** 第 279-280 行

每个 worker 返回自己模型里每一层的 KV 规格（KVCacheSpec）：普通的全注意力层、滑动窗口层、MLA 层、Mamba 这类状态空间层的规格各不相同。混合架构的模型（一部分层是滑动窗口、一部分是全注意力）就是在这里被识别出来的。

2 **非因果的层：关掉分块 prefill 和前缀缓存** 第 288-302 行

3 **量显存** 第 336-339 行

4 **统一决定块数** 第 349-363 行

5 **分配 KV，再编译、录图** 第 404-407 行

← 上一步 | 看整个文件 | 下一步 →

调度：没有 prefill 阶段，也没有 decode 阶段

第 126 讲看到，nano-vllm 一步要么全做 prefill、要么全做 decode。vLLM V1 的调度器开头有一段注释，说清了它的做法：

third_party/vllm/vllm/v1/core/sched/scheduler.py 第 716–742 行 / 共 3934 行 ⇆ 对照

```
716     def schedule(self, throttle_prefills: bool = False) -> SchedulerOutput:
717         self.current_step += 1
718         # NOTE(woosuk) on the scheduling algorithm:
719         # There's no "decoding phase" nor "prefill phase" in the scheduler.
720         # Each request just has the num_computed_tokens and
721         # num_tokens_with_spec. num_tokens_with_spec =
722         # len(prompt_token_ids) + len(output_token_ids) + len(spec_token_ids).
723         # At each step, the scheduler tries to assign tokens to the requests
724         # so that each request's num_computed_tokens can catch up its
```

1 只有两个数 第 718–727 行

调度器里没有“prefill 阶段”和“decode 阶段”。每个请求只有两个数：`num_computed_tokens`（KV 已经算好的位置数）和 `num_tokens_with_spec`（提示 + 已生成 + 草稿）。每一步，调度器给请求分配 token，让前者去追后者。

这一句话就统一了几件事：新请求的 prefill 差额是整个提示；分块 prefill 是差额太大、一步只给一部分；前缀缓存是 `num_computed_tokens` 一开始就不是 0；decode 的差额是 1；投机解码的差额是 1 + 草稿数。

2 这一步的账本 第 729–742 行

← 上一步 | 看整个文件 | 下一步 →

一步的 token 预算怎么分

⇆ 对照

max_num_batched_tokens 1024

long_prefill_token_threshold 512

下一步 → 从头开始

第 1 步将要分出的 token（总宽 = 预算 1024）



用掉 1024 / 1024

请求	状态	num_computed / num_tokens_with_spec	上一步分到	这一步将分到
A	decode	339 / 340	—	1
B	decode + 2 个草稿	519 / 522	—	3
C	prefill 中	512 / 3000	—	512
D	等待	0（缓存） / 800	—	508
E	第 2 步到达	128（缓存） / 200	—	—

每个请求要的是“差额” = num_tokens_with_spec - num_computed_tokens：decode 是 1，带草稿是 1 + 草稿数，prefill 是剩下的提示长度。长提示每步最多分 long_prefill_token_threshold 个，其余留给后面的请求。

图 129-3　统一的 token 预算（简化版）。A、B 在 decode（B 带 2 个草稿），C 在分块 prefill，D、E 在等待。每步先给运行中的请求补差额，再放新请求，直到预算用完。演示里把 long_prefill_token_threshold 设成了 512；vLLM 的默认值是 0（不限），调成“不限”看长提示 C 怎么把整步的预算吃光。

一步的调度分两轮。第一轮是运行中的请求：

third_party/vllm/vllm/v1/core/sched/scheduler.py 第 809–1041 行 / 共 3934 行 ⇆ 对照

```
809         # First, schedule the RUNNING requests.
810         req_index = 0
811         while req_index < len(self.running) and token_budget > 0:
812             request = self.running[req_index]
813             if input_budget ⩽ draft_slots:
814                 break
815
816             if (
817                 request.num_output_placeholders > 0
```

1 **先排运行中的请求** 第 809-814 行

按运行队列的顺序，预算用完就停。运行中的请求里既有在 decode 的，也有在分块 prefill 的（提示还没算完的请求也在运行队列里）。

2 **差额，受三个上限约束** 第 855-873 行

3 **分不到就跳过，而不是停下** 第 907-925 行

4 **要块：allocate_slots** 第 927-938 行

5 **块不够就抢占** 第 946-996 行

6 **记账** 第 1012-1023 行

7 **草稿** 第 1025-1041 行

← 上一步 | 看整个文件 | 下一步 →

第二轮是等待中的请求（第 1065 行起，500 多行，大部分在处理 PD 分离、多模态、LoRA 等情况）。主干和 nano-vllm 一样：查前缀缓存（`get_computed_blocks`，第 1246 行），差额 = 总长 − 命中的部分（第 1356 行），长提示同样受上限约束（第 1382 行），`allocate_slots` 拿块（第 1497 行），拿不到就停（第 1523 行）；拿到了就移进运行队列，`num_computed_tokens` 设成命中的位置数（第 1609–1637 行）。

和 nano-vllm 相比，最关键的差别是：**新请求的 prefill 和老请求的 decode 在同一步里**。一步里可能有 30 个请求各 decode 1 个 token，加上一个新请求 prefill 了 1500 个 token。这对执行一侧提出了要求：注意力 kernel 必须能处理“每个请求查询长度不同”的 batch，CUDA Graph 也要能录这种混合的步。vLLM 的注意力后端统一用“变长查询 + 块表”的接口（第 127 讲的 `flash_attn_varlen_func` 就是这种接口），CUDA Graph 用两种模式：整步录（只适合纯 decode 这类规整的步）和分段录（注意力之外的部分录成图，注意力单独执行，混合的步也能用）。

KV 管理：块池、哈希、LRU

third_party/vllm/vllm/v1/core/kv_cache_utils.py 第 703–733 行 / 共 1284 行 ⇆ 对照

```
703 def hash_block_tokens(
704     hash_function: Callable[[Any], bytes],
705     parent_block_hash: BlockHash | None,
706     curr_block_token_ids: Sequence[int],
707     extra_keys: tuple[Any, ...] | None = None,
708 ) -> BlockHash:
709     """Computes a hash value corresponding to the contents of a block and
710     the contents of the preceding block(s). The hash value is used for
711     prefix caching. We use LRU cache for this function to avoid recomputing
712     hash values for the same block contents
```

1 **块哈希：父块哈希 + 本块 token + 额外的键** 第 703–733 行

和第 122 讲、nano-vllm 一样是链式哈希。多了一个 `extra_keys`：LoRA 适配器的编号、多模态输入（图片）的哈希、缓存的“盐”。两个请求的 token 一样，但用了不同的 LoRA，或者同一位置是不同的图片，KV 就不同，不能共用。

默认的哈希函数是 SHA-256（`vllm/config/cache.py` 的 `prefix_caching_hash_algo`）：慢一点，但碰撞的概率可以忽略，而且结果跨进程、跨机器稳定，KV 可以在多台机器之间共享（第 113 讲）。nano-vllm 用 xxhash 再比一遍 token，是另一种取舍。

← 上一步 | 看整个文件 | 下一步 →

third_party/vllm/vllm/v1/core/kv_cache_utils.py 第 254–275 行 / 共 1284 行 ⇆ 对照

```
254 class FreeKVCacheBlockQueue:
255     """This class organizes a list of KVCacheBlock objects to a doubly linked
256     list of free blocks. We implement this class instead of using Python
257     builtin deque to support removing a block in the middle of the queue
258     in O(1) time. To close the performance gap to the builtin deque which is
259     implemented in C++, this class does not allocate any Python objects when
260     manipulating the linked list. Instead, this class manipulates the
261     prev_free_block and next_free_block attributes of the given blocks.
262
263     The queue is ordered by block ID in the beginning. When a block is allocated
```

1 **空闲块是一个双向链表** 第 254–275 行

nano-vllm 的空闲块是 `deque`，前缀命中一个躺在空闲队列中间的块时，`deque.remove` 要 O(n)。vLLM 自己写了一个双向链表，直接改块对象上的前后指针，从中间摘掉一块是 O(1)，而且不创建任何 Python 对象。

顺序就是淘汰顺序：最久没用的在前面；同一个请求释放的块，链尾的块（含更多 token 的哈希）在前。这和 nano-vllm、miniinfer 倒着还块的道理一样：让开头的块在缓存里留得更久。

← 上一步 | 看整个文件 | 下一步 →

third_party/vllm/vllm/v1/core/block_pool.py 第 798–837 行 / 共 936 行 ⇆ 对照

```
798     def free_blocks(
799         self, ordered_blocks: Iterable[KVCacheBlock], evict_first: bool = False
800     ) -> None:
801         """Free a list of blocks. The blocks should be ordered by their
802         eviction priority, where the first block will be evicted first.
803
804         Args:
805             ordered_blocks: A list of blocks to free ordered by their eviction
806                 priority.
```

1 **还块时分两类** 第 812–833 行

引用计数减到 0 的块：没有哈希的（内容不会被命中）放到空闲链表的**前面**，最先被重新使用（刚用过，还在 GPU 的缓存里）；有哈希的放到**后面**，按 LRU 留着等命中。

miniinfer 和 nano-vllm 不区分这两类，可能先把一个还能被命中的块覆盖掉。

← 上一步 | 看整个文件 | 下一步 →

`KVCacheManager.allocate_slots`（`v1/core/kv_cache_manager.py` 第 401 行起）的注释里有一张图，把一个请求的块分成几段：已缓存的、这一步命中的、外部（PD 分离时从别的机器搬来的）、这一步要算的、为投机解码预留的。和 miniinfer 相比，多出来的两段正好对应第 14 篇讲的 PD 分离和 KV 传输。

投机解码：和我们的回滚是同一个式子

vLLM 的投机解码有十几种起草方式（n-gram、EAGLE、MTP、独立的小模型等，在 `v1/worker/gpu/spec_decode/`），但调度器里的处理很简单。草稿只是让 `num_tokens_with_spec` 多了几个，验证后被拒的草稿这样回滚：

third_party/vllm/vllm/v1/core/sched/scheduler.py 第 2387–2410 行 / 共 3934 行 ⇆ 对照

```
2387            req_index = req_indices[req_id]
2388            generated_token_ids = (
2389                sampled_token_ids[req_index] if sampled_token_ids else []
2390            )
2391
2392            scheduled_spec_token_ids = (
2393                scheduler_output.scheduled_spec_decode_tokens.get(req_id)
2394            )
2395            if scheduled_spec_token_ids and (
```

1 **接受了几个** 第 2392-2401 行

worker 返回这个请求这一步生成的 token（被接受的草稿加上采样的那个）。生成数减去每步必然采样的数量，就是接受的草稿数，其余是被拒的。

2 **回滚** 第 2406-2410 行

← 上一步 | 看整个文件 | 下一步 →

不同在执行一侧：验证在 worker 里用拒绝采样（第 96 讲），支持任意温度；起草器有自己的 KV Cache 时，`allocate_slots` 的 `num_lookahead_tokens` 为它预留位置；为了让整步仍能用 CUDA Graph，调度器会把没有草稿的 decode 请求也补齐到 1 + k 个 token（第 1358–1380 行）。第 128 讲的移植里，验证步放弃了 CUDA Graph，正是这里要解决的问题。

结构化输出：语法在哪里编译，掩码在哪里加

第 80 讲讲了约束解码的原理，第 131 讲会看 SGLang 的做法。vLLM 把它分成三个时机，都由 EngineCore 里的 `StructuredOutputManager`（`v1/structured_output/__init__.py`）管理：

third_party/vllm/vllm/v1/structured_output/__init__.py 第 134–205 行 / 共 548 行 ⇆ 对照

```
134     def grammar_init(self, request: "Request") → None:
135         if request.structured_output_request is None:
136             return
137
138         sampling_params = request.sampling_params
139         assert sampling_params is not None
140         structured_outputs = sampling_params.structured_outputs
141         assert structured_outputs is not None
142         request_backend = structured_outputs._backend
```

1 **一个引擎只用一种后端** 第 144–151 行

xgrammar、guidance（llguidance）、outlines、lm-format-enforcer 四种后端，第一次遇到结构化输出的请求时才初始化，之后整个引擎都用它。

2 **语法在后台线程里编译** 第 196–205 行

← 上一步 | 看整个文件 | 下一步 →

请求进来时状态是 `WAITING_FOR_STRUCTURED_OUTPUT_GRAMMAR`。调度器每一步检查（`scheduler.py` 第 3685–3697 行）：语法编译好了，就改成普通的 `WAITING`，进入正常的调度；还没好，这一步跳过它，其他请求照常跑。

掩码在每一步计算：上面 `EngineCore.step()` 第 724–725 行，调度结果交给执行器之后，主线程马上调 `get_grammar_bitmask`，让后端按每个请求当前的状态算出“下一个 token 允许哪些”，写成一个位图（每个 token 一位，用 32 位整数存）。这一步和 GPU 的前向同时进行，前向算完、采样之前，worker 把位图展开、把不允许的 token 的打分设成 −∞。采样出 token 后，`accept_tokens`（第 515 行）推进每个请求的状态机。

投机解码和结构化输出同时打开时，还要多一步：草稿 token 也必须符合语法，`trim_spec_tokens`（第 358 行）把不符合语法的草稿截掉，免得验证一个注定被拒的 token。

异步调度：让 CPU 的工作藏在 GPU 后面

即使拆成了多个进程，EngineCore 里“调度 → 等 GPU → 处理结果”仍然是串行的：GPU 算的时候 CPU 在等，CPU 调度的时候 GPU 在等。decode 一步只有几毫秒时，调度和处理结果的那一两毫秒就占了可观的比例。

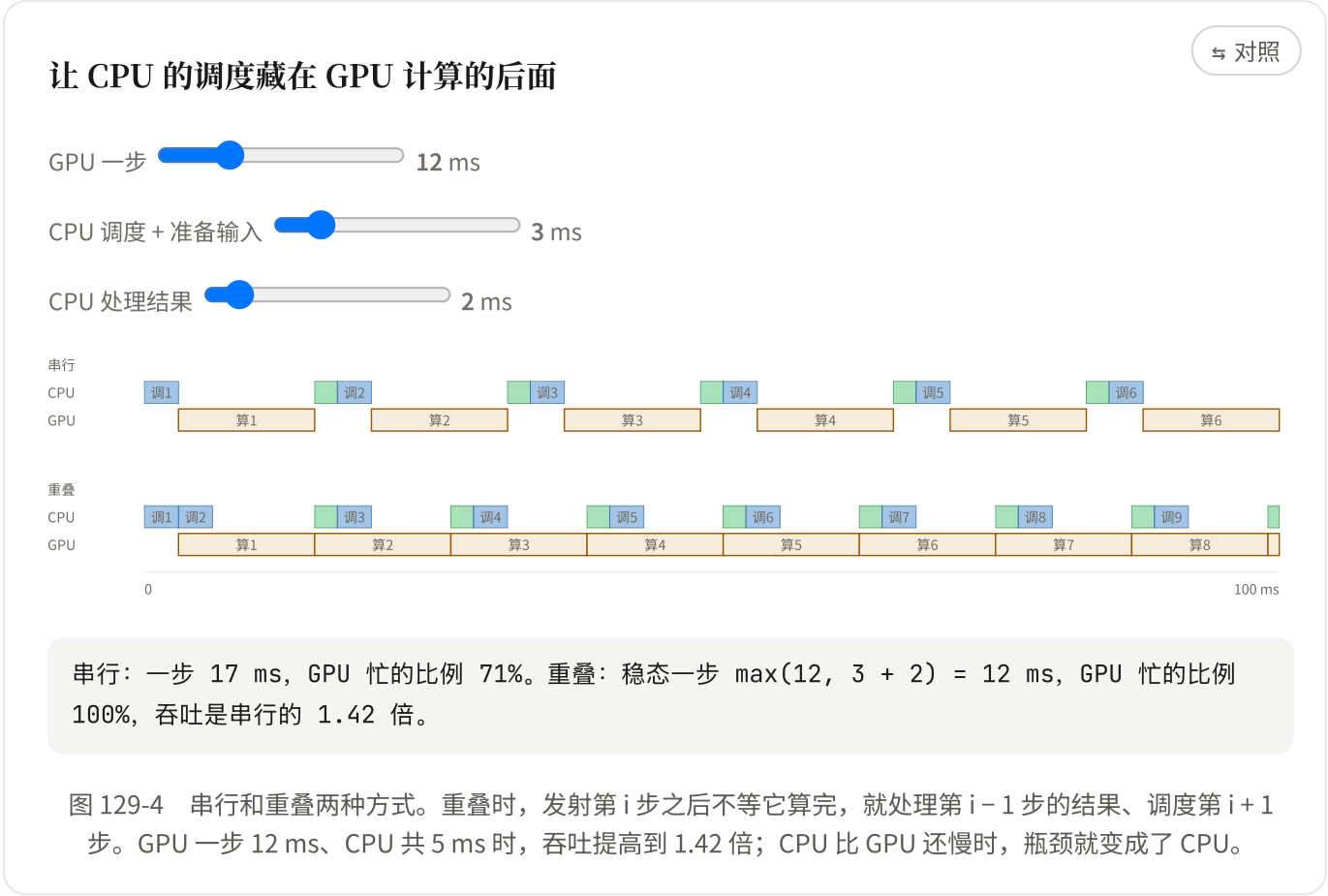


图 129-4　串行和重叠两种方式。重叠时，发射第 i 步之后不等它算完，就处理第 i − 1 步的结果、调度第 i + 1 步。GPU 一步 12 ms、CPU 共 5 ms 时，吞吐提高到 1.42 倍；CPU 比 GPU 还慢时，瓶颈就变成了 CPU。

异步调度（`SchedulerConfig.async_scheduling`，这个提交里默认打开，设成 False 才关掉）时，调度器在不知道上一步采样结果的情况下就开始调度下一步。难点是：下一步要送进模型的 token 正是上一步刚采样出来的，调度器还不知道它是什么。vLLM 的办法是**占位**：

third_party/vllm/vllm/v1/core/sched/async_scheduler.py 共 45 行 ⇆ 对照

```python
# SPDX-License-Identifier: Apache-2.0
# SPDX-FileCopyrightText: Copyright contributors to the vLLM project

import vllm.envs as envs
from vllm.platforms import current_platform
from vllm.v1.core.sched.output import SchedulerOutput
from vllm.v1.core.sched.scheduler import Scheduler


class AsyncScheduler(Scheduler):
```

1 **先记下“会多出几个 token”** 第 18–41 行

调度完一步，每个 decode 的请求将会多出 1 个采样的 token，再加上这一步的草稿数。调度器不知道它们是什么，先给 `num_output_placeholders` 加上这个数，下一步就把它们当作“已有的 token”去调度。真正的 token 由 worker 自己保存，下一步直接从 GPU 上取，不经过调度器。草稿同理，用 −1 占位，worker 里再填上真实的。

代价在调度器的其他地方：比如请求已经到了长度上限，调度器要晚一步才知道（第 816–830 行专门处理这种情况，避免多跑一步）。

← 上一步 | 看整个文件 | 下一步 →

SGLang 更早就把同样的机制作为默认（它称之为“零开销调度”），第 131 讲细看。

SchedulerOutput：只发增量

调度器和 worker 在不同的进程里，每一步都要把调度结果发过去。nano-vllm 每步把整个请求列表 pickle 一次（第 127 讲）。vLLM 的 `SchedulerOutput`（`v1/core/sched/output.py` 第 220 行）把请求分成两类：

- **新请求**（`scheduled_new_reqs`）：发完整的信息，提示的全部 token、采样参数、块表。worker 收到后缓存起来。
- **老请求**（`scheduled_cached_reqs`）：只发这一步的变化，新增的块号、这一步算几个 token。

worker 里的 `GPUModelRunner` 维护一个**持久的 batch**：每个请求在固定的大张量里占一行（token、块表等；有的放在显存里，有的放在 GPU 能直接访问的锁页内存里，见 `v1/worker/gpu/states.py`），每一步只按增量修改其中几行，而不是像 nano-vllm 那样每步从 Python 列表重新拼出全部输入张量。batch 很大（几百个请求）时，这一步省下的 CPU 时间很可观。

要点

vLLM V1 = 三类进程（前端做分词和反分词、EngineCore 做调度、worker 做执行）+ 统一的 token 预算调度（每个请求只有“已算”和“应有”两个数，prefill、分块、前缀缓存、decode、投机都是补差额）+ 分页 KV（SHA-256 链式哈希，双向链表做 LRU）+ 异步调度（用占位让调度和执行重叠）+ 只发增量的 SchedulerOutput 和持久 batch。核心思想和 nano-vllm、miniinfer 一脉相承，多出来的代码主要在支持更多的模型、硬件和功能。

回顾

一个请求在 vLLM V1 里的路径：API 服务器分词 → ZMQ 发给 EngineCore → 调度器在 token 预算内给它分配 token（查前缀缓存、按块分配 KV，块不够就抢占最晚进来的请求）→ 执行器把 SchedulerOutput 的增量广播给每张卡 → worker 更新持久 batch、跑模型、采样 → 调度器接上新 token、回滚被拒的草稿、释放结束的请求 → 输出发回前端，反分词，流式返回。

和 nano-vllm 比，最根本的变化是调度不再区分 prefill 和 decode：这要求注意力 kernel 能处理变长查询，CUDA Graph 能录混合的步，也换来了更平稳的 decode 延迟。

这一讲停在“一个 EngineCore”的边界上。真实的服务往往有多个 API 服务器、跨多个节点的多个引擎副本，PD 分离时还有两种实例之间的 KV 搬运。下一讲把视野放大到整个服务：一条 curl 请求怎么在多节点的 vLLM 里找到它的引擎、跑完、返回。下一讲：**vLLM 的分布式服务**。

第130讲 vLLM 的分布式服务

108 专家并行与 All-to-All　111 MoE 推理与大规模专家并行　112 PD 分离
113 KV Cache 传输与分层存储　114 路由与负载均衡　127 nano-vllm 多做了什么　129 vLLM 的架构

第 129 讲读的是“一个 EngineCore”：一个调度器、一个执行器、几张卡。真实的线上服务往往大得多：模型要跨好几张卡做张量并行；为了吞吐，同一个模型要部署好几个副本，分布在多台机器上；做 PD 分离时，prefill 和 decode 还是两组不同的实例，KV 要在它们之间搬运。

这一讲把视野放大到整个服务，跟着一条 curl 请求走一遍：它先到哪台机器、怎么在几个副本里挑一个、在副本里怎么分给几张卡、MoE 模型的副本之间为什么要“锁步”、PD 分离时 KV 怎么从一个实例到另一个实例。每一层都对应第 14 篇讲过的一个原理（数据并行、专家并行、PD 分离、KV 传输、缓存感知路由），这一讲看它们在 vLLM 里是怎么写出来的。

> **参考**
> 这一讲的讲法参考了 Aleksa Gordić 的博客 *Inside vLLM: Anatomy of a High-Throughput LLM Inference System*（2025 年 8 月，基于 vLLM 提交 `42172ad`）。我们读的是 2026 年 11 月的提交 `821851e`，一年多里不少细节变了：只有 MoE 模型的副本才锁步运行，负载均衡的打分加入了在途请求数和 KV 压力，博客示例里的 `SharedStorageConnector` 也不在了。下文遇到时会指出来，这本身就是读源码时要注意的事：博客和文档会过时，代码才是准的。和前几讲一样，本讲只读源码，没有运行。

第一层：一个副本里的几张卡

先回到单个副本。第 129 讲提到，EngineCore 通过执行器（Executor）让 worker 跑模型。一个副本用 4 张卡做张量并行时，执行器是 `MultiprocExecutor`：每张卡一个 worker 进程，EngineCore 每一步把调度结果广播给它们。

这件事 nano-vllm 也做过（第 127 讲）：rank 0 把“方法名 + 参数”pickle 后写进 1 MiB 的共享内存，再用事件通知其他进程。vLLM 的做法是同一个思路的工程化版本：

third_party/vllm/vllm/v1/executor/multiproc_executor.py 第 471–540 行 / 共 1294 行 ⇆ 对照

```
471     def collective_rpc(  # type: ignore[override]
472         self,
473         method: str | Callable,
474         timeout: float | None = None,
475         args: tuple = (),
476         kwargs: dict | None = None,
477         non_block: bool = False,
478         unique_reply_rank: int | None = None,
479         kv_output_aggregator: KVOutputAggregator | None = None,
```

1 **一次“集体调用”** 第 471–491 行

`collective_rpc` 让所有 worker 执行同一个方法（`execute_model`、`sample_tokens` 等都是通过它调用的）。`unique_reply_rank` 指定只要哪个 worker 的返回值。

2 **KV connector 的输出要汇总** 第 493–510 行

3 **写进广播队列** 第 512–516 行

4 **只从需要的 worker 读回复** 第 518–533 行

← 上一步 | 看整个文件 | 下一步 →

谁来回复？第 683–697 行的 `_get_output_rank` 给出了答案：**最后一个流水线阶段里张量并行编号为 0 的那个 worker**。只有最后一个流水线阶段算出了打分，而同一阶段里各张卡的结果相同，取第一张就够了。worker 那边是一个死循环（第 1188–1218 行）：从广播队列取出调用，执行，如果自己就是指定回复的那一个，就把结果写进自己的回复队列。

和 nano-vllm 相比，差别在细节上：环形队列让 EngineCore 可以连续发出几步（流水线并行、异步调度时需要），不用等上一步的读者读完；回复只走一条队列；所有 worker 共享同一个广播写入，而不是 pickle 一次、通知 N 次。

第二层：多个副本

模型放进 4 张卡以后，想要更高的吞吐，就部署更多副本：数据并行（DP）。博客里的例子是两台 8 卡的机器、4 个副本、每个副本张量并行 4 卡。在 vLLM 里这样启动（两台机器各运行一次）：

```
# 节点 A：带 API 服务器，副本 0、1
vllm serve <model> --tensor-parallel-size 4 --data-parallel-size 4 --data-parallel-size-local 2 \
  --data-parallel-start-rank 0 --data-parallel-address <节点 A 的 IP> --data-parallel-rpc-port 13345
# 节点 B：只跑引擎，副本 2、3
vllm serve <model> --tensor-parallel-size 4 --data-parallel-size 4 --data-parallel-size-local 2 \
  --data-parallel-start-rank 2 --data-parallel-address <节点 A 的 IP> --data-parallel-rpc-port 13345 --headless
```

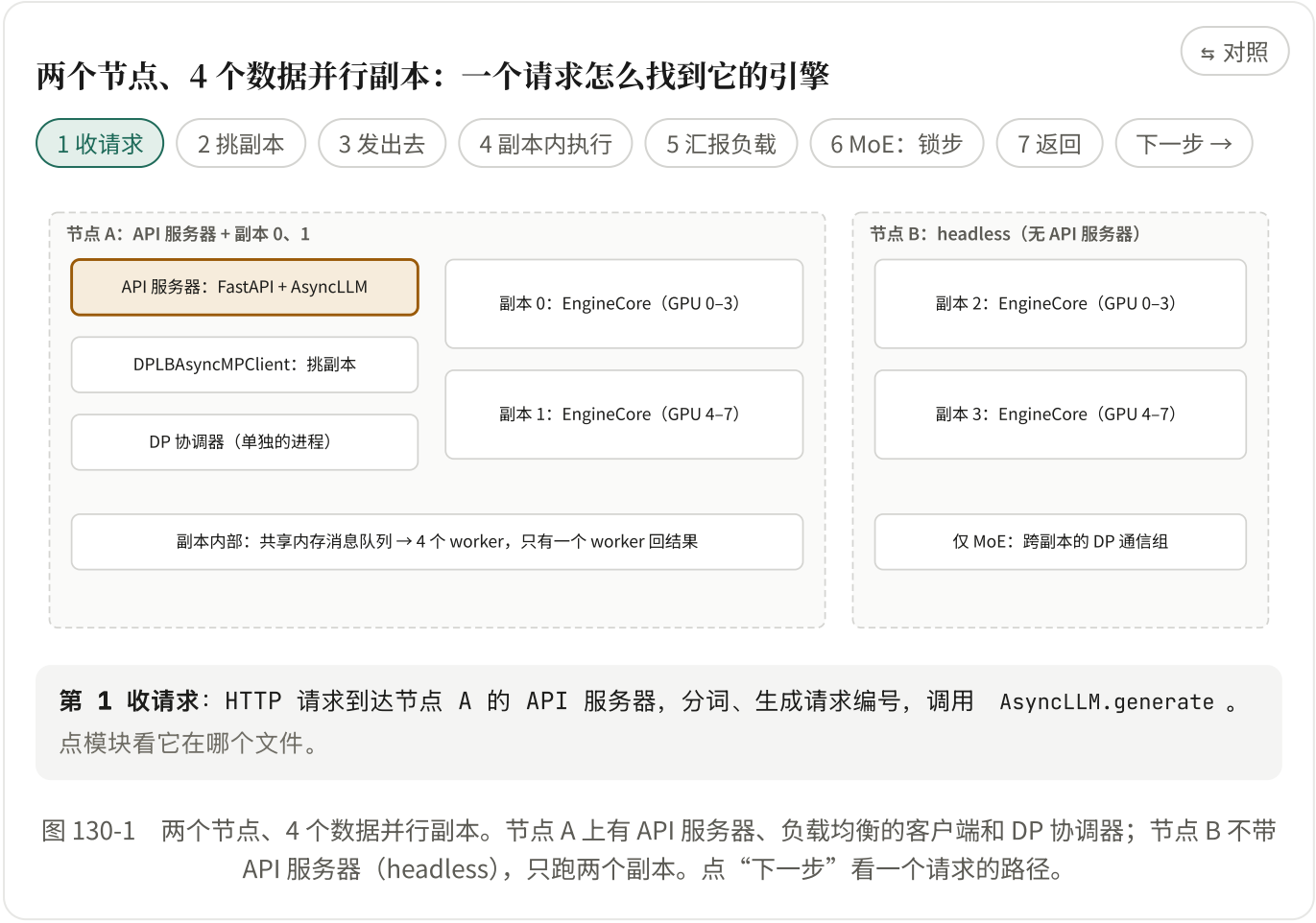


图 130-1　两个节点、4 个数据并行副本。节点 A 上有 API 服务器、负载均衡的客户端和 DP 协调器；节点 B 不带 API 服务器（headless），只跑两个副本。点“下一步”看一个请求的路径。

非 MoE 和 MoE 是两回事

博客写作时，vLLM 的数据并行副本一律“锁步”运行（下一节讲锁步是什么）。现在的代码把两种情况分开了：

third_party/vllm/vllm/v1/engine/core.py 第 1718–1728 行 / 共 3143 行 ⇆ 对照

```
1718            parallel_config.data_parallel_index = dp_rank
1719            if data_parallel and vllm_config.model_config.is_moe:
1720                # Set data parallel rank for this engine process.
1721                parallel_config.data_parallel_rank = dp_rank
1722                engine_core = DPEngineCoreProc(*args, **kwargs)
1723            else:
1724                # Non-MoE DP ranks are completely independent, so treat like DP=1.
1725                # Note that parallel_config.data_parallel_index will still reflect
1726                # the original DP rank.
```

1 **MoE 模型：DPEngineCoreProc** 第 1719–1722 行

MoE 模型的专家层要跨副本做专家并行（第 108、111 讲）：每一步前向里都有 All-to-All，所有副本必须一起参加。所以 MoE 模型的每个副本是一个 `DPEngineCoreProc`，它们之间有一个通信组，步调一致。

2 **非 MoE 模型：彼此独立** 第 1723–1728 行

← 上一步 | 看整个文件 | 下一步 →

前端用哪种客户端

API 服务器里的 `AsyncLLM` 通过一个“引擎核心客户端”和 EngineCore 通信（第 129 讲）。数据并行时，选哪种客户端决定了谁来做负载均衡：

third_party/vllm/vllm/v1/engine/core_client.py 第 153–186 行 / 共 2252 行 ⇆ 对照

```python
    def make_async_mp_client(
        vllm_config: VllmConfig,
        executor_class: type[Executor],
        log_stats: bool,
        client_addresses: dict[str, Any] | None = None,
        client_count: int = 1,
        client_index: int = 0,
        renderer: BaseRenderer | None = None,
    ) -> "AsyncMPClient":
```

1 **外部负载均衡** 第 171–177 行

`--data-parallel-external-lb`：每个副本配一个自己的 API 服务器，由 vLLM 之外的负载均衡器（比如 Kubernetes 的 Service）在副本之间分请求。适合“一个 Pod 一个副本”的大规模 MoE 部署。

2 **内部负载均衡（默认）** 第 178–182 行

3 **单副本** 第 183–186 行

← 上一步 | 看整个文件 | 下一步 →

第三层：挑哪个副本

内部负载均衡要回答一个问题：新请求来了，发给哪个副本？前端自己不知道各副本有多忙，需要有人告诉它。这个人就是 **DP 协调器**，一个单独的进程：

third_party/vllm/vllm/v1/engine/coordinator.py 第 23–54 行 / 共 464 行 ⇆ 对照

```
23 class DPCoordinator:
24     """Coordinator process used for data-parallel deployments (DP>1).
25
26     Intermediates between multiple DP engine rank processes and one or more
27     front-end API server processes.
28
29     * Collects stats from each DP engine (currently just waiting and running
30       queue lengths), and publishes these to all front-ends for use in
31       load-balancing decisions.
```

1 **收集负载，发布给前端** 第 29–31 行

每个副本把自己的等待数、运行数（以及 KV 占用率）发给协调器，协调器合并后发布给所有前端。发布的间隔最小 100 ms（`min_stats_update_interval_ms`）。

2 **请求“波次”（MoE 才用）** 第 33–49 行

← 上一步 | 看整个文件 | 下一步 →

前端订阅协调器的发布（`core_client.py` 第 1599–1674 行的 `run_engine_stats_update_task`）：它总是把积压的消息读空，只保留最新的一份快照，存在 `self.lb_engines` 里。每来一个请求，`add_request_async`（第 1676–1687 行）先挑副本，再把请求发过去。挑副本的代码：

third_party/vllm/vllm/v1/engine/core_client.py 第 1818–1879 行 / 共 2252 行 ⇆ 对照

```
1818     def get_core_engine_for_request(self, request: EngineCoreRequest) → EngineIdent
1819         if request.resumable:
1820             self.streaming_reqs.add(request.request_id)
1821         else:
1822             # Also removes a session on its final (end-of-stream) request.
1823             self.streaming_reqs.discard(request.request_id)
1824
1825         if (engine := self.reqs_in_flight.get(request.request_id)) is not None:
1826             # This is for streaming-input session affinity.
1827             return engine
```

1 **同一个会话发回同一个副本** 第 1825–1827 行

流式输入的会话（请求分几次送进来）要一直发给同一个副本，它的 KV 在那里。

2 **逐个副本打分** 第 1829–1843 行

3 **基础分：快照和在途数取大** 第 1844–1853 行

4 **KV 压力罚分** 第 1854–1861 行

5 **挑中之后，先在本地记一笔** 第 1864–1873 行

6 **记下请求去了哪里** 第 1875–1879 行

← 上一步 | 看整个文件 | 下一步 →

这几条规则各自防的是什么，用一个小模型看得最清楚：

⇆ 对照

前端怎么在 4 个副本之间分请求

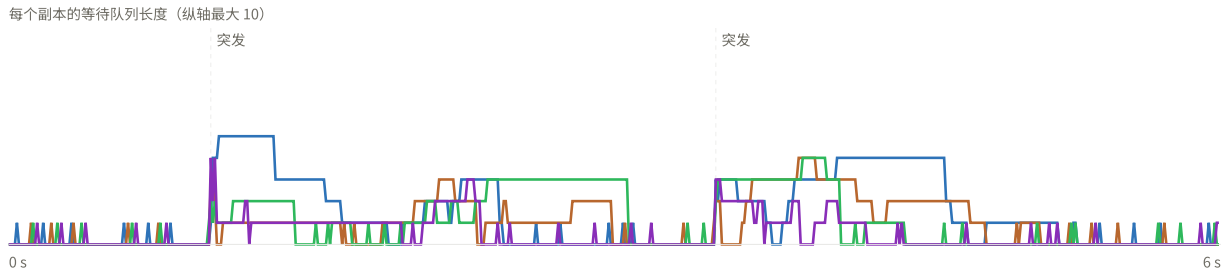
只看快照　现在的 vLLM（三条全开）

☑ 本地计数

☑ 在途下限

☑ KV 压力罚分

快照间隔 100 ms



副本	分到的请求	最长等待队列
副本 0	42	5
副本 1	43	4
副本 2	44	4
副本 3	51	4

平均排队 **124 ms**，95% 分位 **650 ms**（从到达到开始运行；179/180 个请求在 6 s 内开始运行）。

图 130-2　4 个副本、两阵突发的简化模型（不是 vLLM 的实测，规则见组件说明）。点“只看快照”：同一份快照里最空的副本会在 100 ms 内吃下整阵突发，平均排队 497 ms；打开“本地计数”就降到 123 ms。再把快照间隔拉长，看两种设置各自怎么变化。

在这个模型里，作用最大的是**本地计数**：它消除了“一份旧快照被用很多次”带来的扎堆。在途下限和 KV 罚分的作用在这里不明显，因为模型只有一个前端、快照从不被重置，各副本的请求组成也差不多；它们针对的是多个前端同时分请求、快照和路由决策互相赛跑、个别副本 KV 吃紧这些真实部署里的情况。这也是读这类代码的一个经验：每一条看起来多余的规则，背后往往对应一个线上出过的问题，去翻它的提交记录（第 133 讲）就能找到。

和第 114 讲的缓存感知路由是什么关系

这里的负载均衡只看“忙不忙”，不看“谁有我的前缀缓存”。第 114 讲讲的缓存感知路由，在 vLLM 里通常放在更外面的一层（多个 vLLM 实例前面的路由器，例如 llm-d 或 SGLang 的路由器），因为它需要知道各实例缓存了哪些块（vLLM 可以通过 KV 事件把这些信息发布出去）。两层可以叠加：外层按缓存挑实例，实例内部再按负载挑副本。

第四层：MoE 副本的锁步

对稠密模型，副本选好就结束了。MoE 模型多了一个约束：**所有副本必须一起前向**。专家层的 All-to-All 是一个集合通信，每个副本都必须参加；只要有一个副本在算，其余副本就算没有请求，也要跑一个“空批次”陪着，否则正在算的副本会卡在 All-to-All 上等。

third_party/vllm/vllm/v1/engine/core.py 第 2745–2830 行 / 共 3143 行 ⇆ 对照

```
2745     def run_busy_loop(self):
2746         """Core busy loop of the EngineCore for data parallel case."""
2747         # Loop until process is sent a SIGINT or SIGTERM
2748         while self._handle_shutdown():
2749             # 1) Poll the input queue until there is work to do.
2750             was_running = self.engines_running
2751             self._process_input_queue()
2752             # Publish request counts before and after GPU step to ensure freshness.
2753             self._maybe_publish_request_counts()
```

1 **收请求，发布负载** 第 2749–2753 行

和普通的 EngineCore 一样（第 129 讲），先处理输入队列，再执行一步。发布负载的数字给协调器用。

2 **没活也要陪跑** 第 2774–2792 行

3 **大家都空了吗** 第 2794–2797 行

4 **一个波次结束** 第 2799–2817 行

5 **一个波次开始** 第 2818–2830 行

← 上一步 | 看整个文件 | 下一步 →

“大家都空了吗”这个问题本身也有代价：每次都要做一次跨副本的 All-Reduce。现在的代码不再每一步都问：

third_party/vllm/vllm/config/parallel.py 第 966–991 行 / 共 1443 行 ⇆ 对照

```
966     def sync_dp_state(
967         self, dp_group: ProcessGroup, has_unfinished: bool, pending_pause: bool
968     ) -> tuple[bool, bool]:
969         """Combined all-reduce for DP state synchronization.
970 
971         Uses a single SUM all-reduce on a 2-element tensor:
972           [0] = 1 if this rank has unfinished work, else 0.
973                 SUM > 0 ≡ logical OR across ranks → any rank has work.
974           [1] = 1 if this rank has a pending pause request, else 0.
```

1 一次 All-Reduce 问两件事 第 969–983 行

一个两元素的张量做求和：第 0 个元素是“我有没有未完成的请求”，求和大于 0 就说明有人在忙；第 1 个元素是“我是否想暂停”（比如要重新加载权重），求和等于副本数才算全体同意。两件事合成一次通信。

2 在 CPU 上做 第 984–991 行

← 上一步 | 看整个文件 | 下一步 →

调用它的 `_has_global_unfinished_reqs`（`core.py` 第 2834 行起）只在一个波次的第 1 步和之后每 `dp_sync_interval` 步（默认 16）真正同步一次，其余步直接当作“还有人在忙”。代价是：所有副本都空闲后，最多还要多陪跑十几个空批次才停下来；换来的是正常运行时每 16 步才同步一次，而不是每一步都多一次跨机器的通信。

第五层：PD 分离时，KV 从哪里来

第 112 讲讲了 PD 分离的原理，第 113 讲讲了 KV 的传输和分层存储。在 vLLM 里，这些都通过一个接口接进来：KV connector。它的接口文件开头就把全部方法列了出来：

third_party/vllm/vllm/distributed/kv_transfer/kv_connector/v1/base.py

⇆ 对照

第 3–40 行 / 共 824 行

```
3  """KVConnectorBase_V1 Class for Distributed KV Cache & Hidden State
4  communication in vLLM v1
5
6  The class provides the following primitives:
7      Scheduler-side: runs in the scheduler, binds metadata, which
8      is used by the worker-side to load/save KV cache.
9          get_num_new_matched_tokens() - get number of new tokens
10             that exist in the remote KV cache. Might be called multiple
11             times for a given request and should be side-effect free.
12         update_state_after_alloc() - update KVConnector state after
```

1 调度器一侧 第 7–23 行

在 EngineCore 进程里，和调度器一起工作：

- `get_num_new_matched_tokens`：这个请求在外部（另一个实例、CPU 内存、远端存储）还有多少个 token 的 KV。可能被调用多次，必须没有副作用。
- `update_state_after_alloc`：调度器为这些外部 token 分好块之后，告诉 connector 往哪些块里写。
- `request_finished`：请求结束时决定块是马上释放，还是由 connector 负责稍后释放（等对方把 KV 取走），并可返回 KV 的位置信息。

此外还有 `build_connector_meta`（第 602 行）：每一步调度结束时，把这一步要加载、要保存的信息打包进 SchedulerOutput，随调度结果一起发给 worker。

2 worker 一侧 第 25–39 行

← 上一步 | 看整个文件 | 下一步 →



图 130-3　PD 分离时一个请求的路径。prefill 实例和 decode 实例都是普通的 vLLM，只是配置了不同角色的 KV connector。

调度器一侧的调用点在 `scheduler.py` 里，等待队列放一个新请求进来时：

third_party/vllm/vllm/v1/core/sched/scheduler.py 第 1170–1190 行 / 共 3934 行 ⇆ 对照

```
1170                        # Present a block-aligned local hit to the connector so
1171                        # a strictly longer remote hit can supersede a local
1172                        # sub-block tail without racing its copy-on-write.
1173                        partial_tail = num_new_local_computed_tokens % self.block_si
1174                        block_aligned_local = (
1175                            num_new_local_computed_tokens - partial_tail
1176                        )
1177                        ext_tokens, load_kv_async = (
1178                            self.connector.get_num_new_matched_tokens(
```

1 **问 connector：外面还有多少个 token 的 KV** 第 1170-1181 行

先在本地前缀缓存里查到命中的部分（按块对齐），再问 connector：除了本地命中的，**外部**还能提供多少个 token 的 KV，以及要不要异步加载。PD 分离时，decode 实例在这里得知“整个提示的 KV 都在 prefill 实例那边”。

2 **还不知道就先跳过** 第 1185-1190 行

← 上一步 | 看整个文件 | 下一步 →

接下来，`allocate_slots` 的 `num_external_computed_tokens` 参数（第 129 讲那张块布局图里的 `ext_comp` 段）为这些外部 token 分好块，`connector.update_state_after_alloc`（第 1544 行）记下要写进哪些块；异步加载的请求进入“等远端 KV”的状态，加载完成后才进入运行队列。worker 一侧，在 GPU worker 里由一个小类包着：

third_party/vllm/vllm/v1/worker/gpu/kv_connector.py 第 62–166 行 / 共 194 行 ⇆ 对照

```
62 class ActiveKVConnector(KVConnector):
63     def __init__(
64         self, vllm_config: VllmConfig, kv_caches_dict: dict[str, torch.Tensor]
65     ):
66         self.vllm_config = vllm_config
67         self.kv_connector = get_kv_transfer_group()
68         # Register kv caches with KV Connector if applicable.
69         self.kv_connector.register_kv_caches(kv_caches_dict)
70         self.kv_connector.set_host_xfer_buffer_ops(copy_kv_blocks)
```

1 **把块池交给 connector** 第 66–70 行

初始化时，把每一层 KV Cache 的张量注册给 connector。传输后端（例如 NIXL）要知道这些显存的地址，才能直接用 RDMA 读写。

2 **前向之前** 第 75–96 行

3 **前向之后** 第 136–160 行

← 上一步 | 看整个文件 | 下一步 →

具体的传输方式是不同的 connector 实现，都在 `vllm/distributed/kv_transfer/kv_connector/v1/` 下：`nixl/`（NVIDIA 的 NIXL，GPU 之间点对点 RDMA）、`mooncake/`（Mooncake 的传输引擎）、`lmcache_mp_connector.py`（LMCache 的 KV 缓存层）、`offloading_connector.py`（卸载到 CPU 内存，即第 113 讲的分层存储）、`multi_connector.py`（同时用几种），以及一个用来理解接口的最简实现 `example_connector.py`（把 KV 存成本地文件）。博客里用的 `SharedStorageConnector` 在这个提交里已经不存在了，作用相同的是 `example_connector.py`。

合起来：一条 curl 的全程

把五层串起来，一条请求在“两个节点、4 个 MoE 副本、每副本 4 卡”的部署里：

1. **API 服务器**（`entrypoints/openai/completion/serving.py` 的 `create_completion`）收到请求，`AsyncLLM.generate` 分词、建输出流。
2. **挑副本**：`DPLBAsyncMPClient.get_core_engine_for_request` 用协调器的快照、本地计数、在途数和 KV 压力打分，选中比如副本 2；本地快照里副本 2 的等待数加一。
3. **发过去**：经 ZMQ 发到节点 B 上副本 2 的输入线程，放进输入队列。如果副本组正处于暂停，副本 2 醒来并通知协调器，协调器广播 START_DP_WAVE 叫醒其余副本。
4. **调度**：副本 2 的调度器为它分配 token 和块（第 129 讲）；PD 分离时先问 connector 外部有多少 KV。
5. **执行**：`MultiprocExecutor` 把调度结果写进共享内存广播队列，4 个 worker 一起前向；MoE 的专家层和其他 3 个副本做 All-to-All（没有请求的副本跑空批次陪着）；输出 rank 把采样结果写回。
6. **更新**：调度器接上新 token、判断结束；副本把新的负载数字发给协调器，协调器发布给前端。
7. **返回**：副本 2 的输出线程把结果发回节点 A，`AsyncLLM` 的输出任务反分词、分发到这个请求的输出流，FastAPI 流式推给客户端。

要点

vLLM 的分布式服务是一层套一层：副本内部，MultiprocExecutor 用共享内存广播队列驱动几张卡，只让一个 worker 回结果；副本之间，稠密模型彼此独立，MoE 模型锁步运行（空闲副本跑空批次陪着做 All-to-All，每 16 步 All-Reduce 一次判断是否全体空闲，用“波次”管理启停）；前端在副本之间分请求，用协调器每 100 ms 发布的负载快照，加上本地计数、在途下限和 KV 压力罚分；PD 分离通过 KV connector 接入，调度器一侧回答“外部有多少 KV、写进哪些块、何时释放”，worker 一侧负责加载和保存。

回顾

这一讲从一个副本的几张卡，一路放大到多节点、多副本和 PD 分离。几个要记住的取舍：回复只走一个 worker，省掉重复的序列化；稠密模型的副本互不通信，MoE 的副本必须锁步；负载均衡依赖 100 ms 一次的旧快照，靠本地计数和在途数补偿；全局同步每 16 步一次，用少量空跑

换掉大量通信；异步的 KV 加载挪到前向之后，不挡关键路径。这些细节和一年前的博客已经不同，读源码时要以手上的那个提交为准。

下一讲换一个引擎：SGLang 怎么用一棵基数树做前缀缓存，怎么让调度和执行重叠。下一讲：**SGLang 的架构**。

第131讲 SGLang 的架构

76 Prefix Cache　80 约束解码　89 CUDA Graph　122 前缀缓存　129 vLLM 的架构

SGLang 和 vLLM 是今天用得最多的两个开源推理引擎。它最初来自一篇论文（Zheng 等，*SGLang: Efficient Execution of Structured Language Model Programs*），要解决的问题是：一个 LLM 应用往往不是一次调用，而是一串调用（多轮对话、思维链、先生成再检查、按 JSON 格式输出），这些调用之间共享大量的前缀，输出还要满足格式约束。论文提出了两样东西：**RadixAttention**（用一棵基数树做前缀缓存，自动复用任意请求之间的公共前缀）和**压缩的有限状态机**（加速约束解码）。后来的版本又加上了**零开销调度**（CPU 调度和 GPU 计算重叠）。

这一讲读这三样，以及 SGLang 的整体结构。读的是 2026 年 11 月 9 日 main 分支的提交 `48da6bd`（Apache-2.0 许可），只取下了 `srt/managers`、`srt/mem_cache`、`srt/model_executor`、`srt/constrained` 等几个目录。和上一讲一样，只读源码，没有运行。

三个进程



图 131-1　SGLang 的进程结构：主进程（HTTP 和分词）、每张卡一个调度器进程（调度和执行模型在同一个进程里）、一个反分词进程。点“下一步”看一个请求的路径。

和 vLLM 一样，SGLang 也把分词、调度、反分词放在不同的进程里。最大的不同在中间：vLLM 的 EngineCore 只有一份，调度结果广播给各个 worker 进程；SGLang 是**每张卡一个调度器进程**，调度器和执行模型的 `TpModelWorker` 在同一个进程里。rank 0 从 ZMQ 收到请求后，用 `broadcast_pyobj` 发给其余的 rank（`managers/scheduler_components/request_receiver.py`），每个 rank 各自跑一遍同样的调度，得到同样的结果，然后各自执行。

这样做省掉了“调度结果每步跨进程发给 worker”这一步：调度器直接在本进程里调用模型。代价是每张卡都重复做一遍调度的 CPU 工作，并且必须保证各 rank 的调度结果完全一致。

RadixAttention：前缀缓存是一棵树

第 122 讲、nano-vllm 和 vLLM 的前缀缓存都是“块哈希”：把序列切成固定大小的块，每块按“前面所有块 + 本块”算哈希，只能整块地命中。SGLang 换了一个数据结构：**基数树**（radix tree，压缩的前缀树）。

- 每个节点存一段 token（`key`）和这些 token 的 KV 在内存池里的槽位（`value`）。
- 从根到某个节点的路径，拼起来就是一段完整的前缀。
- 一个节点的孩子按第一个 token 区分，同一段公共前缀只存一次。



图 131-2　依次送入 5 个请求，看树怎么长。请求 2 和请求 1 在“北京”处分开，匹配停在节点中间，节点被劈成两段；请求 5 是请求 1 加上回答，整条路径都能复用。每次都会对比：按每块 4 个 token 的块哈希能命中多少。

third_party/sglang/python/sglang/srt/mem_cache/radix_cache.py 第 251–280 行 / 共 ⇆ 对照

```
251 class TreeNode:
252     counter = 0
253
254     def __init__(self, id: Optional[int] = None, priority: int = 0):
255         self.children = defaultdict(TreeNode)
256         self.parent: TreeNode = None
257         self.key: RadixKey = None
258         self.value: Optional[torch.Tensor] = None
259         self.lock_ref = 0
```

1 一个节点 第 254–261 行

`children`：按孩子的第一个 token（页大小大于 1 时是第一页）索引。`key`：这个节点的 token 段。`value`：这段 token 的 KV 槽位，一个整数张量。`lock_ref`：有几个正在运行的请求在用这个节点，大于 0 就不能淘汰。`last_access_time`：LRU 淘汰用。

2 **被淘汰的节点** 第 274–279 行

← 上一步 | 看整个文件 | 下一步 →

匹配：从根往下走，停在中间就劈开

third_party/sglang/python/sglang/srt/mem_cache/radix_cache.py 第 658–707 行 / 共 ⇆ 对照

```
658     def _match_prefix_helper(self, node: TreeNode, key: RadixKey):
659         access_time = time.monotonic()
660         node.last_access_time = access_time
661
662         child_key = key.child_key(self.page_size)
663
664         value = []
665         while len(key) > 0 and child_key in node.children.keys():
666             child = node.children[child_key]
```

1 沿着树往下走 第 658–668 行

从根开始，用请求剩余 token 的第一个找孩子；找到了，算这个孩子的 key 和请求剩余部分的公共前缀有多长。路过的节点都刷新访问时间。

2 **停在节点中间：劈开** 第 669–673 行

3 **整段匹配：继续往下** 第 674–682 行

4 **_split_node** 第 684–707 行

← 上一步 | 看整个文件 | 下一步 →

插入、加锁、淘汰

third_party/sglang/python/sglang/srt/mem_cache/radix_cache.py 第 559–623 行 / 共 ⇆ 对照

```
559     def evict(self, params: EvictParams) → EvictResult:
560         if self.disable:
561             return EvictResult()
562
563         start_time = time.perf_counter()
564         num_tokens = params.num_tokens
565         leaves = list(self.evictable_leaves)
566         eviction_heap = [
567             (self.eviction_strategy.get_priority(node), node) for node in leaves
```

1 **淘汰：从没人用的叶子里挑最久没访问的** 第 559–587 行

需要腾出 num_tokens 个槽位时，把所有可淘汰的叶子（没有孩子、没人在用）放进一个堆，按淘汰策略（默认 LRU）一个个弹出：把它的槽位还给内存池，从树上删掉。删掉之后，如果父节点也变成了没人用的叶子，就把父节点也放进堆里。所以淘汰总是从树的末梢往根部推进：公共前缀（靠近根的节点）被很多请求访问过，最后才会被淘汰。

2 **加锁：一路锁到根** 第 589–602 行

3 **解锁** 第 604–623 行

← 上一步 | 看整个文件 | 下一步 →

插入（第 712–767 行的 `_insert_helper`）和匹配几乎一样：沿树往下走，在分叉处劈开节点，最后把没匹配上的部分作为一个新的叶子挂上去。请求结束时，它的全部 token（提示 + 输出）都插入树中，KV 的槽位交给树管理，而不是还给内存池。下一轮对话进来时，就能匹配上整个上一轮。

内存：槽位是一个个 token，不是块

基数树的 `value` 是逐个 token 的槽位编号，这背后是 SGLang 的两级内存池（`mem_cache/memory_pool.py`）：

third_party/sglang/python/sglang/srt/mem_cache/memory_pool.py 第 381–478 行 / 共 ⇆ 对照

```
381 class ReqToTokenPool:
382     """A memory pool that maps a request to its token locations."""
383
384     enable_mamba_extra_buffer_lazy: bool = False
385     # Extra pre-allocation headroom (reserved for in-transfer decode requests).
386     # 0 for a plain pool; the decode-flavored pool (DecodeReqToTokenPool) sets a
387     # positive value. Declared here so callers can read it without getattr.
388     pre_alloc_size: int = 0
389     # Class default: some decode pools borrow another __init__ (see
```

1 **ReqToTokenPool：每个请求一行** 第 412–423 行

一个形状为 [请求数上限, 最长上下文] 的 int32 张量 `req_to_token`：第 r 行第 i 列，是第 r 个请求第 i 个 token 的 KV 在 KV 池里的槽位号。它相当于块表，只是粒度是 token：每个 token 单独记一个槽位，不要求连续。第 0 行留空，CUDA Graph 补齐出来的空请求都指向它，读写无害（和 nano-vllm 用 −1 的思路一样）。

2 **分一行** 第 432–465 行

3 **还一行** 第 467–478 行

← 上一步 | 看整个文件 | 下一步 →

第二级是 KV 池（`token_to_kv_pool`）：一个按槽位编号的大张量，存每一层每个 token 的 K、V。分配器在 `mem_cache/allocator` 里。打开 `page_size` 大于 1 时，分配、匹配和淘汰都按页对齐，粒度变粗，但有的注意力后端要求按页存（和 nano-vllm 的 256、vLLM 的块是同一

类约束）。

把两种前缀缓存放在一起比较：

	块哈希（vLLM、nano-vllm、miniinfer）	基数树（SGLang）
命中粒度	整块（16–256 个 token）	1 个 token（或一页）
查找	逐块算哈希、查表	沿树往下比较 token
结构维护	哈希表 + 空闲队列	劈开、插入节点；锁要一路加到根
淘汰	LRU 的空闲块队列，倒着还让前缀留得久	从叶子开始的 LRU，公共前缀自然最后被淘汰
跨机器共享	哈希可以跨进程稳定，便于做全局的 KV 池	需要额外给页算哈希（节点上的 `hash_value`）
额外开销	每块一次哈希	树的 Python 操作，前缀很多时匹配较慢

两者在实际的命中率上差得并不多（块足够小时），差别更多在工程上：基数树天然适合“多轮对话、思维链分叉”这类前缀呈树状的负载；块哈希更简单，也更适合跨机器的 KV 共享。

缓存感知的调度

有了前缀缓存，调度器选谁先 prefill 也会影响命中率。SGLang 的 `SchedulePolicy`（`managers/schedule_policy.py`）提供了几种策略：

third_party/sglang/python/sglang/srt/managers/schedule_policy.py ⇆ 对照

第 167–260 行 / 共 1826 行

```
167 class CacheAwarePolicy(Enum):
168     """Scheduling policies that are aware of the tree cache."""
169
170     LPM = "lpm"  # longest prefix match
171     DFS_WEIGHT = "dfs-weight"  # depth-first search weighting
172     HRRN = "hrrn"  # highest response ratio next, token-based aging
173     SHORTEST_PREFILL_FIRST = "shortest-prefill-first"
174
175
```

1 **两类策略** 第 167–182 行

缓存感知的：lpm（最长前缀匹配优先）、dfs-weight（按树的深度优先顺序，让同一棵子树下的请求挨着跑）、hrrn、最短 prefill 优先。**不感知缓存**的：fcfs（先来先到）、lof（最长输出优先）、随机等。

2 **不管哪种策略，都先做匹配** 第 216–228 行

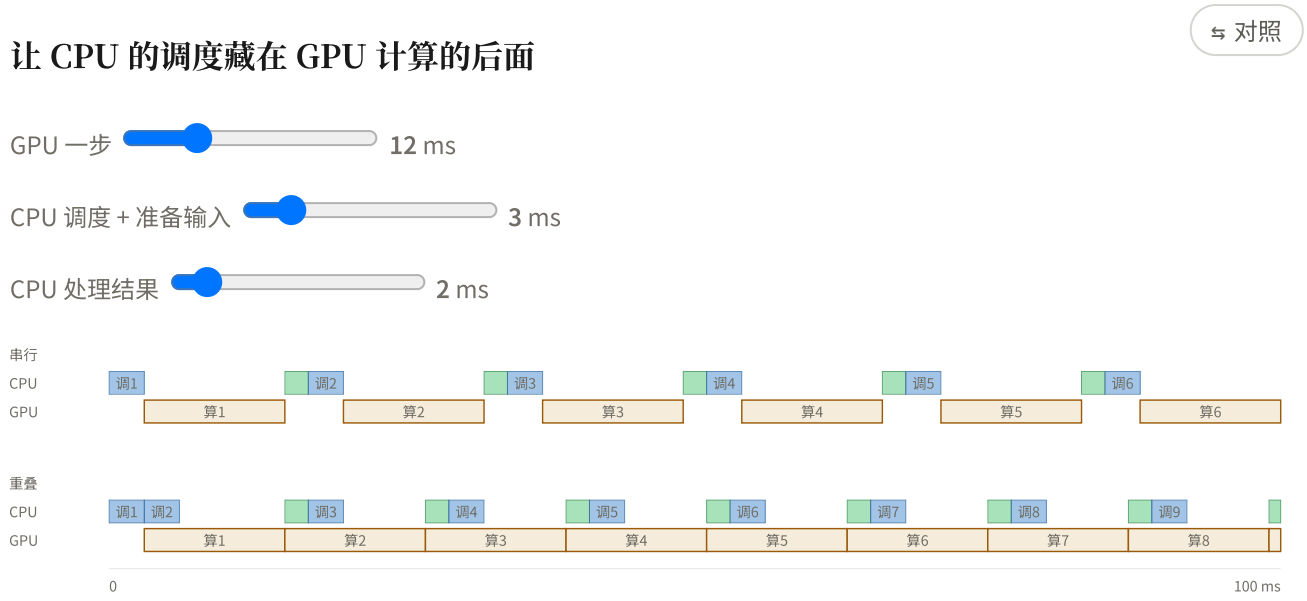
3 **lpm：命中越长越先** 第 234–241 行

← 上一步 | 看整个文件 | 下一步 →

两个值得注意的细节：等待队列超过 128 个请求时，lpm 会退化成 fcfs（`_determine_active_policy`，第 312 行），因为对每个请求做匹配再排序太慢了；这个提交里默认的策略是 fcfs（`srt/arg_groups/fields/schedule.py`），缓存感知的排序要显式打开。缓存感知的调度能提高命中率，但会让前缀不常见的请求等得更久，默认值偏向公平。

零开销调度：发射这一步，再处理上一步

第 129 讲看到，vLLM 用占位的办法让调度和执行重叠。SGLang 的调度循环更直接地体现了这个思路：



串行：一步 17 ms，GPU 忙的比例 71%。重叠：稳态一步 max(12, 3 + 2) = 12 ms，GPU 忙的比例 100%，吞吐是串行的 1.42 倍。

图 131-3　调度和计算重叠（和图 129-4 是同一个模型）。SGLang 称之为“零开销调度”：只要 CPU 一步的工作比 GPU 一步快，CPU 的时间就完全藏在 GPU 后面。

third_party/sglang/python/sglang/srt/managers/scheduler.py 第 2116–2221 行 / 共 6(⇆ 对照

```
2116     def event_loop_normal(self):
2117         """A normal scheduler loop."""
2118         while True:
2119             if self.gracefully_exit:
2120                 break
2121             self.increment_scheduler_iterations()
2122
2123             # Receive requests
2124             self.ingest_requests()
```

1 **普通的循环** 第 2116–2148 行

收请求 → 决定下一批 → 执行 → 处理结果，严格串行。和 nano-vllm 的 `step()` 一样。

2 **重叠的循环：一个结果队列** 第 2151–2161 行

3 **先决定、发射这一步** 第 2187–2200 行

4 **再处理上一步** 第 2205–2208 行

5 **有时要先处理上一步** 第 2174–2185 行

6 **采样放在最后** 第 2213–2216 行

← 上一步 | 看整个文件 | 下一步 →

重叠有一个隐含的前提：调度第 i + 1 步时还不知道第 i 步采样出的 token。SGLang 让 GPU 上的输入直接引用第 i 步的输出张量（token 不回到 CPU 再送下去），所以 i + 1 步的输入在 GPU 上自然就是对的。这和 vLLM 的占位是同一个思路的两种写法。重叠模式默认打开，`--disable-overlap-schedule` 可以关掉。

调度规则：prefill 优先，可以混合

`get_next_batch_to_run`（`managers/scheduler.py` 第 4158 行）的主干是：先把上一步的 prefill 批并入运行中的 batch；然后尝试组一个新的 prefill 批（`get_new_batch_prefill`）；组得出来就跑 prefill，组不出来才跑 decode（`update_running_batch`，KV 不够时会“撤回”一部分请求，相当于抢占）。这和 nano-vllm 的 prefill 优先一样。打开 `--enable-mixed-chunk` 后，分块 prefill 的批可以把运行中的 decode 请求也带上（`mix_with_running`，第 4701 行），一步里混合两种，这又接近 vLLM 的做法了。

约束解码：每一步一个词表掩码

第 80 讲讲过约束解码的原理：按语法（JSON Schema、正则、EBNF）维护一个状态机，每一步算出“下一个 token 可以是哪些”，把其余 token 的打分设成 $-\infty$。SGLang 支持 xgrammar、outlines、llguidance 几个后端（`srt/constrained/`），默认用 xgrammar：

third_party/sglang/python/sglang/srt/constrained/xgrammar_backend.py

⇆ 对照

第 78–200 行 / 共 497 行

```python
class XGrammarGrammar(BaseGrammarObject):
    def __init__(
        self,
        matcher: GrammarMatcher,
        vocab_size: int,
        ctx: CompiledGrammar,
        override_stop_tokens: Optional[Union[List[int], int]],
        key_string: Optional[str] = None,
        grammar_stats: Optional[GrammarStats] = GrammarStats(),
```

1 **接受一个 token** 第 97–109 行

每个请求一个 `GrammarMatcher`，记着语法走到了哪里。采样出 token 后调用 `accept_token` 推进状态。不被接受说明前面的掩码有 bug，直接报错。

2 **回退** 第 111–113 行

3 **算掩码，加到打分上** 第 123–147 行

4 **跳跃式前进** 第 168–200 行

← 上一步 | 看整个文件 | 下一步 →

要点

SGLang = 每张卡一个调度器进程（调度和执行同进程，rank 0 收请求再广播）+ RadixAttention（前缀缓存是一棵基数树，逐 token 命中，节点加锁防淘汰，从叶子开始 LRU 淘汰）+ 两级内存池（请求 → token 槽位，槽位 → KV）+ 零开销调度（发射这一步之后再处理上一步，CPU 藏在 GPU

后面）+ 基于位图的约束解码。默认是 prefill 优先、fcfs，缓存感知的调度和混合分块要显式打开。

回顾

SGLang 和 vLLM 走到了相近的地方：都分了进程、都有分页（或逐 token）的 KV 和前缀缓存、都让调度和执行重叠、都支持投机解码和约束解码。差别主要在几处取舍：前缀缓存用树还是用哈希；调度结果是广播给 worker，还是每张卡各算一遍；prefill 和 decode 默认分开跑还是混着跑。读懂这些取舍背后的原因，比记住哪个引擎有哪个功能更有用，因为功能会互相借鉴，取舍的道理不会变。

下一讲把视野放宽：TensorRT-LLM、LMDeploy、llama.cpp 这些引擎各自的定位，以及 PD 分离、大规模专家并行在这些引擎里是怎么落地的。下一讲：**其他引擎与规模化部署**。

第132讲 其他引擎与规模化部署

108 专家并行与 All-to-All　111 MoE 推理与大规模专家并行　112 PD 分离　113 KV Cache 传输与分层存储　114 路由与负载均衡　129 vLLM 的架构　130 vLLM 的分布式服务　131 SGLang 的架构

第 129、131 讲读了 vLLM 和 SGLang 的单机结构，第 130 讲读了 vLLM 怎么组成一个多节点的在线服务。这一讲做两件事：一是看看其他几个常见引擎各自的定位，知道什么场景该选什么；二是看第 14 篇讲过的大规模专家并行在真实的引擎里由哪几样东西拼起来（PD 分离和多副本服务在第 130 讲已经读过）。

注意

这一讲关于各引擎特点的描述是定性的，来自各项目的公开文档和本篇读过的源码，不是本站测出来的性能排名。这些项目更新很快，选型时请以官方文档和自己的压测为准（第 133 讲讲怎么压测）。

几个常见的引擎

⇆ 对照

按场景看引擎

在线服务（数据中心 GPU） | 共享前缀多（多轮、Agent） | 大规模 MoE（DeepSeek 类）

显存紧、要 4 位量化 | 本地 / 端侧 / Mac | 国产加速卡

引擎	实现	硬件	最适合	特点
vLLM	Python + CUDA/Triton kernel	NVIDIA、AMD、Intel、TPU，昇腾等靠插件	通用的在线服务；模型支持最全	统一 token 预算调度、分页 KV、异步调度；KV connector 接 PD 分离
SGLang	Python + CUDA/Triton kernel	NVIDIA、AMD，昇腾等	前缀复用多的负载（多轮、Agent、结构化输出）；DeepSeek 类 MoE 大规模部署	RadixAttention、零开销调度、约束解码；DP 注意力 + 专家并行
TensorRT-LLM	C++ 运行时 + Python（PyTorch 后端）	只有 NVIDIA	在 NVIDIA 新卡上压榨性能，FP8 / FP4 低精度	NVIDIA 官方的 kernel 和低精度支持最早到位；部署链路较重
LMDeploy	C++（TurboMind）+ PyTorch 引擎	NVIDIA；PyTorch 引擎支持昇腾等	4 位权重量化、KV 量化下的高吞吐	TurboMind 的 W4A16 kernel；书生（InternLM）系列支持好
llama.cpp	C/C++，GGUF 格式	CPU、Apple Metal、CUDA、Vulkan 等几乎一切	本地、端侧、单用户或低并发	多种 k-quant 量化；Ollama、LM Studio 等以它为内核
MLC-LLM	TVM 编译	手机、浏览器（WebGPU）、各类 GPU	要编译到特殊平台（移动端、网页）	用编译器为每种硬件生成 kernel

要 continuous batching、分页 KV、前缀缓存、OpenAI 兼容接口、多卡并行。四个都具备；vLLM 和 SGLang 社区最大、模型最新；只用 NVIDIA 新卡、追求极致时看 TensorRT-LLM。

图 132-1　六个引擎的定位。选一个部署场景，强调色的是适合的引擎，下方说明原因。

按“它们在第 125 讲那张差距表上各自押注在哪一行”来理解，比记功能列表更有用：

TensorRT-LLM（NVIDIA，Apache-2.0）押注在 **kernel**。它最初的思路是把模型“编译”成一个针对具体 GPU、具体 batch 形状优化好的引擎（TensorRT 的老路），后来改成以 PyTorch 为主的后端，调度部分（in-flight batching，即 continuous batching；分页 KV；分块 prefill；投机解码）和 vLLM 大同小异。它的优势在 NVIDIA 自家写的 kernel 和低精度支持：新一代 GPU 上的 FP8、FP4 矩阵乘、融合的注意力和 MoE kernel，往往在这里最先成熟。代价是只支持 NVIDIA 的卡，部署链路也更重。

LMDeploy（书生 InternLM 团队）有两个引擎：C++ 写的 TurboMind，和一个纯 PyTorch 的引擎。TurboMind 押注在**量化**：4 位权重量化（AWQ 等）和 KV Cache 量化下的矩阵乘、注意力 kernel 是它的强项。第 123 讲我们亲手验证过，量化要快起来，必须有直接读低位宽权重的 kernel；TurboMind 就是这类 kernel 做得好的例子。PyTorch 引擎更容易扩展到新模型和昇腾等其他硬件。

llama.cpp（C/C++）押注在**能在任何地方跑**。它用自己的 GGUF 格式存权重，内置十几种量化格式（如 Q4_K_M 这类 k-quant，按小块分组、每组带缩放），在 CPU（AVX、ARM NEON）、Apple 的 Metal、CUDA、Vulkan 上都有 kernel。它的目标是本地和端侧：单用户、低并发、内存紧，关心的是单请求的延迟和占用，而不是几百个请求的吞吐。Ollama、LM Studio 这类桌面工具都以它为内核。它也有 continuous batching 和多槽位的服务模式，但不是为大规模并发设计的。

MLC-LLM 押注在**编译器**：用 TVM 为每种硬件生成 kernel，因此能部署到手机、浏览器（WebGPU）这类别的引擎去不了的地方。

此外，各家国产加速卡厂商通常为 vLLM、SGLang 提供插件或分支（例如昇腾的 vllm-ascend），也有自己的推理服务（例如昇腾的 MindIE）。选型时先看自己的硬件上哪些引擎有官方支持。

怎么快速判断一个引擎的定位

看三样东西：它的 kernel 是自己写的、调用的库，还是编译器生成的（决定在什么硬件上快）；它的调度器是 Python 还是 C++、有没有分页 KV 和前缀缓存（决定高并发时的吞吐）；它的服务层有没有 OpenAI 兼容接口、流式输出、监控指标（决定上线的成本）。第 117–125 讲的 miniinfer 在这三样上都是最简单的版本，用它当参照系，任何引擎的取舍都能看出来。

PD 分离与编排层

第 130 讲已经完整读过：vLLM 的数据并行副本、DP 协调器和负载均衡，以及 PD 分离用的 KV connector（调度器一侧回答“外部有多少 KV、写进哪些块、何时释放”，worker 一侧负责加载和保存；NIXL、Mooncake、LMCache 等是不同的实现）。SGLang 的做法类似：用 `disaggregation_mode` 区分 prefill 和 decode 实例，传输后端可以选 mooncake、nixl 等。

在引擎之上，还需要一层编排：把请求先送到 prefill 实例，再把 decode 交给合适的 decode 实例，按负载伸缩两类实例的数量（第 116 讲的容量规划）。这一层常见的有 NVIDIA 的 Dynamo（后端可以是 vLLM、SGLang 或 TensorRT-LLM）、基于 Kubernetes 的 llm-d（以 vLLM 为后端），以及 SGLang 自己的路由器。第 114 讲的缓存感知路由也在这一层实现。

大规模专家并行：几样东西拼起来

第 111 讲讲过 DeepSeek 这类 MoE 模型的大规模部署：注意力部分用数据并行（每张卡处理不同的请求），专家部分用专家并行（每张卡放几个专家），两次 All-to-All 把 token 送到专家所在的卡再送回来（第 108 讲）。落到引擎里，是这几样东西拼起来：

要解决的问题	做法	在源码里
注意力用数据并行，各卡 batch 不一样大	每一步各卡同步 batch 大小，空闲的卡也要跑一个空 batch 陪着做 All-to-All	vLLM：`--data-parallel-size`、`v1/worker/gpu/dp_utils.py`；SGLang：DP 注意力（`managers/scheduler_components/dp_attn.py`）
All-to-All 要快	专门的通信库（如 DeepSeek 开源的 DeepEP），prefill 用高吞吐的模式，decode 用低延迟的模式	两个引擎都把它作为 MoE 层的通信后端之一
专家负载不均（第 108 讲）	EPLB：统计每个专家的负载，把热门专家复制到多张卡，周期性地重排	vLLM：`v1/worker/gpu/eplb_utils.py`
通信时 GPU 空等	把一个 batch 切成两个微批，一个在通信时另一个在计算	vLLM：`v1/worker/ubatching.py`（2 个微批）；SGLang：two-batch overlap（`managers/overlap_utils.py` 等）
prefill 和 decode 互相干扰	PD 分离，prefill 和 decode 各用不同的并行配置	第 130 讲

这些单独看都不复杂，难在组合：DP 注意力要求各卡步调一致，微批重叠要求注意力和通信的时间配得上，EPLB 重排专家时不能打断服务，PD 分离要求两边的块大小、KV 布局能对上。这也是为什么大规模 MoE 部署往往有一份专门的、写死了各项参数的“部署方案”，而不是几个开关就能打开。

要点

选引擎先看它押注在哪：TensorRT-LLM 押 kernel（NVIDIA 新卡上的低精度），LMDeploy 押量化 kernel，llama.cpp 押“哪里都能跑”，MLC-LLM 押编译器，vLLM 和 SGLang 押通用的高并发服务。规模化部署是在引擎上“接”出来的：PD 分离靠 KV connector 接口（第 130 讲），大规模专家并行靠 DP 注意力、All-to-All 通信库、EPLB 和微批重叠拼起来，再由上层的编排系统调度实例。

回顾

TensorRT-LLM、LMDeploy、llama.cpp、MLC-LLM 和 vLLM、SGLang 的调度骨架大同小异，区别在押注的方向：kernel、量化、硬件覆盖面还是编译器。PD 分离和多副本服务在 vLLM 里怎么实现，第 130 讲已经读过。大规模专家并行则是 DP 注意力、DeepEP 这类通信库、EPLB 和微批重叠的组合。

第五章的最后一讲，回到方法：面对 17 万行的 vLLM，怎么找到入口、怎么定位一个功能、怎么压测，怎么开始贡献代码。下一讲：**怎么读大型引擎的源码**。

第133讲 怎么读一个大型引擎的源码

66 性能分析　70 TTFT、TPOT 与吞吐　82 推理压测实践　125 时间花在哪里　129 vLLM 的架构　130 vLLM 的分布式服务　131 SGLang 的架构

第五章从一个 119 行的 v0 开始，写到 400 行的 miniinfer，再读 1200 行的 nano-vllm，最后读了 17 万行的 vLLM 和体量相当的 SGLang。这最后一讲把一路上用到的读法整理出来：面对一个大型引擎，怎么找入口、怎么定位一个功能、怎么验证自己的理解、怎么开始改它。

一、先有一张地图，再进代码

读大型代码库最常见的失败，是从目录树的第一个文件开始往下读。17 万行里，和“一个请求怎么被处理”直接相关的只占一小部分；其余的是模型实现、硬件适配、各种可选功能。所以第一步是**带着一张地图进去**，地图就是你已经懂的那个小引擎。

第五章其实一直在用这张地图。把四份代码的同一个模块放在一起：

模块	miniinfer v5	nano-vllm	vLLM V1	SGLang
请求状态	`sequence.py`	`engine/sequence.py`	`v1/request.py`	`managers/schedule_batch.py`（`Req`）
调度器	`scheduler.py`	`engine/scheduler.py`	`v1/core/sched/scheduler.py`	`managers/scheduler.py`、`schedule_policy.py`
KV 块与前缀缓存	`block_manager.py`	`engine/block_manager.py`	`v1/core/kv_cache_manager.py`、`block_pool.py`	`mem_cache/radix_cache.py`、`memory_pool.py`
主循环	`engine.py` 的 `step()`	`engine/llm_engine.py` 的 `step()`	`v1/engine/core.py` 的 `step()`	`managers/scheduler.py` 的 `event_loop_overlap()`
拼输入、跑模型	`engine.py` + `model.py`	`engine/model_runner.py`	`v1/worker/gpu/model_runner.py`	`managers/tp_worker.py`、`model_executor/model_runner.py`
注意力	`model.py` 的 `attend()`	`layers/attention.py`	`v1/attention/backends/`	`layers/attention/`（本篇未取下）
采样	`argmax`	`layers/sampler.py`	`v1/worker/gpu/sample/`	`layers/sampler.py`（本篇未取下）
投机解码	`spec.py`	（第 128 讲移植）	`v1/worker/gpu/spec_decode/`	`speculative/`（本篇未取下）

有了这张表，读 vLLM 的调度器时，你知道它一定要回答 miniinfer 的调度器回答过的那几个问题：这一步跑谁、每个请求算几个 token、块不够怎么办、请求什么时候结束。3569 行里，能对上这几个问题的就是主干，其余的（PD 分离、多模态、LoRA、流水线并行……）第一遍可以全部跳过。

二、顺着一个请求走，而不是顺着目录走

第 129–131 讲都是这样读的：找到入口（`LLM.generate` 或 HTTP 接口），找到主循环（`EngineCore.run_busy_loop`、`Scheduler.event_loop_overlap`），然后只追一个请求：它在哪里被分词、被放进哪个队列、在哪一行被调度、它的 KV 块在哪里分配、在哪里被送进模

型、新 token 在哪里接上、在哪里结束。每一站只读到“知道这一站做了什么”为止，不往旁边展开。

几个实用的办法：

- **读注释里的设计说明**。大项目的核心函数开头往往有一段说明设计的注释（第 129 讲调度器开头那段“没有 prefill 阶段”的注释，比后面 300 行代码更重要）。
- **读数据结构，再读函数**。先看 `SchedulerOutput`、`Request`、`TreeNode` 这些结构里有哪些字段，函数做的事情大多就是在改这些字段。
- **读测试**。单元测试是最好的用法示例：它会构造一个最小的场景（几个请求、几个块），断言调度的结果。vLLM 的调度器测试在 `tests/v1/core/` 下。
- **读提交历史**。一段看不懂的代码，用 `git log -L` 或 `git blame` 找到引入它的提交，再看对应的 PR 描述：大多数 PR 会写清楚为什么要这样改、解决了什么问题、压测数字是多少。

三、要定位一个功能，从配置项搜起

想知道某个功能是怎么实现的，最快的入口往往是它的**配置项**：每个用户能打开的功能都有一个参数，参数从命令行一路传到用它的地方。比如想知道第 129 讲的“长 prefill 上限”是怎么起作用的：

```
$ .venv/bin/python bench/trace_flag.py long_prefill_token_threshold
```

⇆ 对照

```
“long_prefill_token_threshold”出现在 3 个文件里，共 21 处：

vllm/config/scheduler.py（7 处）
     90  long_prefill_token_threshold: int = Field(default=0, ge=0)
    102  long_prefill_token_threshold is nonzero."""
    313  "long_prefill_token_threshold",
    344  self.long_prefill_token_threshold = 0
         …还有 3 处

vllm/engine/arg_utils.py（3 处）
    618  long_prefill_token_threshold: int =
SchedulerConfig.long_prefill_token_threshold
   1796  **scheduler_kwargs["long_prefill_token_threshold"],
   2708  long_prefill_token_threshold=self.long_prefill_token_threshold,

vllm/v1/core/sched/scheduler.py（11 处）
    631  long_prefill_threshold = self.scheduler_config.long_prefill_token_threshold
    770  # `long_prefill_token_threshold` exists to stop a long prefill from
    775  long_prefill_token_threshold = (
    776  self.scheduler_config.long_prefill_token_threshold
         …还有 7 处
```

图 133-1　在 vLLM 源码里搜一个配置项（只搜文本）。它出现在三个文件里：config/scheduler.py 定义它（默认 0，表示不限）和校验它，engine/arg_utils.py 把命令行参数传进来，v1/core/sched/scheduler.py 用它。读最后一个文件里的那 11 处，就知道这个功能的全部行为。

这条“定义 → 传递 → 使用”的链在所有大型项目里都存在。读的顺序反过来：先读定义处的文档字符串（它说清了意图和默认值），再直接跳到使用处，中间的传递可以略过。

四、验证你的理解：跑起来，打日志，做剖析

读懂了一段代码，要用运行来验证。对推理引擎，有几种由浅入深的办法：

1. **最小配置跑一遍**：一张卡、一个小模型（比如第五章一直在用的 Qwen3-0.6B）、关掉 CUDA Graph（vLLM 的 `enforce_eager=True`，nano-vllm 同名），这样所有代码都按 Python 的顺序执行，能打断点、能加打印。

2. **打开调试日志**：vLLM 用环境变量 `VLLM_LOGGING_LEVEL=DEBUG`；在调度器里临时打印每一步的 `num_scheduled_tokens`，就能看到第 129 讲那张 token 预算图的真实版本（就像 miniinfer 的 `bench/batching.py` 画出的甘特图）。
3. **剖析**：用 PyTorch Profiler 或 Nsight Systems 抓一段 decode（第 66 讲），看每一步里 CPU 在做什么、GPU 在做什么、中间有没有空隙。第 125 讲在 CPU 上做的“时间花在哪里”，在 GPU 上就要靠这些工具。
4. **对照实验**：只改一个参数（打开、关闭前缀缓存；草稿数 0 和 4；关闭重叠调度），比较输出和速度。第五章每一版的 `bench/` 脚本都是这个思路。

五、压测：先想清楚要测什么

第 82 讲讲了推理压测的方法，读引擎源码时它是检验理解的最后一关：你认为某个改动会让什么变快，压测要能测出来。几个容易踩的坑：

- **离线吞吐和在线服务是两回事**。离线（一次给全部请求，测总 token/s）看的是满载时的吞吐；在线（请求按一定的速率到达，测 TTFT、TPOT 的分布，第 70 讲）看的是延迟和 SLO。vLLM 的 `vllm bench throughput` 和 `vllm bench serve` 分别对应这两种。
- **随机的提示测不出前缀缓存**。随机 token 之间没有公共前缀，前缀缓存毫无作用；要测前缀缓存，就要构造共享前缀的负载（第 122 讲的系统提示实验）。
- **控制输出长度**。不控制时，不同的引擎、不同的采样参数会在不同的地方遇到结束符，生成的 token 数不一样，吞吐没有可比性。压测时通常忽略结束符、固定输出长度（nano-vllm 的 `bench.py` 就用了 `ignore_eos=True`）。
- **先预热**。第一次请求会触发编译、录 CUDA Graph、分配显存，计时要从预热之后开始。
- **扫一个区间，而不是测一个点**。并发数（或到达速率）从小到大扫一遍，画出吞吐和延迟的曲线，才能看出系统在哪里饱和（第 82 讲）。

六、开始贡献

读懂之后，最好的检验是改它，第 128 讲就是一次小小的练习。给 vLLM、SGLang 这样的项目贡献代码，通常的路径是：

1. **从小的开始**：文档、测试、带有“good first issue”标签的问题、你自己在使用中遇到并能复现的 bug。

2. **大的改动先讨论**：新功能、改接口、改调度策略，先开一个 issue 或 RFC，说明动机、设计和预期的收益，和维护者达成一致再写代码。
3. **按项目的规矩来**：装好 pre-commit 做格式和静态检查，跑相关的单元测试，新代码配上测试。
4. **用数字说话**：性能相关的 PR 附上压测的设置和前后对比（硬件、模型、并发、输入输出长度），正确性相关的附上能复现问题的最小例子。
5. **保持 PR 小而聚焦**：一个 PR 只做一件事，更容易被审阅和合入。

要点

读大型引擎：带着小引擎的地图进去，只追一个请求的路径，每一站只读到知道它做了什么为止；先读设计注释和数据结构，再读函数；用测试和提交历史理解“为什么”；定位功能从配置项搜起（定义 → 传递 → 使用）；用最小配置运行、调试日志、剖析和对照实验验证理解；压测分清离线和在线、控制输出长度、预热、扫区间；贡献从小处开始，大改先讨论，用数字说话。

回顾：第五章走过的路

篇	做了什么	得到了什么
第 15 篇	手写 miniinfer：v0 前向、v1 KV Cache、v2 continuous batching、v3 分页和前缀缓存、v4 量化、v5 投机解码	一个推理引擎的每个零件从哪里来、为什么需要，以及每一个都用实测数字检验过
第 16 篇	读 nano-vllm，并把投机解码移植进去	同样的思想在 GPU 上怎么实现：变长 FlashAttention、写 KV 的 kernel、CUDA Graph、张量并行；一个功能在真实引擎里要碰到多少地方
第 17 篇	读 vLLM、SGLang，概览其他引擎和规模化部署	工业级引擎的结构和取舍：进程拆分、统一 token 预算、基数树前缀缓存、调度和执行重叠、KV connector

到这里，整个课程从“一个神经元怎么算”走到了“一个 17 万行的推理引擎怎么读”。后面的路，是挑一个你关心的方向（kernel、调度、量化、分布式），在真实的系统里继续往下挖。

AI INFRA · HANDBOOK

第 18 篇

适配原理：把模型跑通

收录 8 讲 · 第 134–141 讲

第134讲 换一块卡要重做什么

60 GPU 架构　86 CUDA 编程模型　126 nano-vllm 的整体结构　129 vLLM 的架构

前五章的每一行代码、每一个数字，背后都默认站着一块 NVIDIA 的卡：第 86 讲的线程网格是 CUDA 的，第 84 讲的 FlashAttention 来自一个只编译 CUDA 的库，第 101 讲的 All-Reduce 走 NCCL。现在把这些东西搬到另一块卡上：一台装了 8 张国产加速器的机器，手里是同一份 Qwen3 权重和同一个 vLLM。

第一个问题不是“快不快”，而是“从哪里开始”。这一讲把模型和芯片之间的东西拆成九层，说清楚每一层是谁的责任、换卡时哪些能原样沿用、哪些有对应物但要翻译、哪些必须从头重写，以及这些工作该按什么顺序做。后面七讲是这张图上的几层各自展开，第 20 篇是把这张图套到海光 DCU 上。

九层：从模型文件到驱动

我们从一个具体的调用开始往下看。一个用户请求进来，vLLM 最终要算出 `x @ w`（第 137 讲会逐层追这一次调用）。这一次矩阵乘法要穿过下面这些东西：



图 134-1　从模型到芯片的九层。先看 NVIDIA 这一列：每层都有现成的东西，所以“换卡要重做什么”这个问题在 CUDA 上不存在。点任意一层看它负责什么；切换到另外两条路线，看同一层变成了什么、工作量是“沿用”“翻译”还是“重写”。

自上而下过一遍，顺便说清每层的职责边界——边界划在哪里，决定了换卡时谁必须改。

模型代码。`Qwen3ForCausalLM` 这样的类，只描述张量怎么流动：先 RMSNorm，再投影，再注意力。它不关心张量在哪块卡上。所以这一层是最该原样复用的——除非代码里写死了 `device="cuda"` 或者 `.cuda()`，这类写法在第 136 讲会专门清点。

推理引擎或训练框架。vLLM、SGLang、Megatron。它们管的是调度、KV 块、批处理、分布式的组织方式（第 10 篇、第 13 篇）。这些逻辑和硬件无关，但引擎里有一层**平台抽象**：设备叫什么名字、用哪个注意力后端、用哪个通信器、一块卡有多少显存。这一层是厂商插件接进来的地方（第 147 讲）。

深度学习框架。PyTorch。它要回答一个很具体的问题：`x @ w` 这个调用，最终执行哪一份机器码？答案由**分发器**（dispatcher）给出：按张量的设备类型和数据类型，查表找到注册在这个设备上的算子实现。换卡时这一层有两种接法——沿用 CUDA 的分发键，或者注册一个全新的设备——第 137 讲细讲。

图编译。`torch.compile` 把 Python 代码捕获成一张计算图，做融合、常量折叠，再交给后端生成代码。NVIDIA 上默认的后端是 Inductor + Triton。有的硬件更依赖这一层，因为它要求一次把整张图交给硬件（第 138 讲）。

算子库。cuBLAS 管矩阵乘，cuDNN 管卷积，CUTLASS 是写矩阵乘 kernel 的模板库，FlashAttention 是一个专门的注意力库。它们的共同点是：针对某一代硬件的指令形状、缓存容量、线程组宽度，手工调到接近峰值。这意味着它们**不可移植**——不是语法上不能编译，而是调好的参数换一块卡就不再是最优（第 143 讲）。

kernel 语言与编译器。CUDA C++ 和 nvcc，往下是 PTX 和机器码。Triton 是另一条路：用 Python 写块级的 kernel，由它的编译器生成各家后端的代码（第 88 讲）。这一层是“手写 kernel”的落脚点，也是迁移时最容易卡住的地方（第 136、144 讲）。

集合通信库。NCCL 实现 All-Reduce、All-Gather、All-to-All（第 100 讲），而且知道机内 NVLink、机间网卡的拓扑，会据此选算法（第 102 讲）。每家都有自己的对应物（第 146 讲）。

运行时与驱动。设备枚举、显存分配、流和事件、kernel 发射。这一层是纯粹的厂商实现，没有复用的余地；它决定了上面所有层能不能工作。

工具。`nvidia-smi` 看卡的状态，Nsight Systems 看时间线，Nsight Compute 看单个 kernel。调优全靠它们，所以这一层缺了的话，第 19 篇讲的所有分析方法都无从下手（第 142 讲）。

层与层之间的接口，决定了换卡时改动能不能被挡住：

接口	上面的人看到什么	下面的人要提供什么	换卡时谁改
模型 ↔ 引擎	`forward(input_ids, positions)`	一个能执行前向的 runner	都不改
引擎 ↔ 平台层	`current_platform.get_attn_backend_cls()` 等几十个方法	一个 Platform 子类	厂商写插件
框架 ↔ 算子	`torch.matmul(x, w)`	注册在这个设备上的 kernel	厂商注册算子
算子 ↔ 运行时	`hipLaunchKernel` / `cudaLaunchKernel`	发射、同步、分配显存	厂商实现运行时

要点

换卡这件事，之所以不是“重写整个栈”，是因为上面四层（模型、引擎、框架的 Python 接口、图的表示）都站在接口的上方；真正要重做的是接口下方的实现。工作量的大小，取决于厂商选择在哪一层接入。

量一下：和硬件绑定的代码有多少

“上面几层能复用”这句话要落到数字上。第 16 篇读过的 nano-vllm 是个好标本：一千多行，功能完整，依赖关系清楚。把它的源码按上面的分层扫一遍，数出每一行里和 NVIDIA 绑定的写法：

```
$ .venv/bin/python bench/port_inventory.py

nano-vllm（bb823b3）共 1195 行代码（不含空行和注释）
和 NVIDIA 绑定的写法，按层分类（一行只算进第一个匹配的类）：

【设备与运行时】23 行，分布在 1 个文件：model_runner.py
    换卡后：换成新设备的接口；ROCm 式的后端沿用 torch.cuda，PrivateUse1 式的要改成新设备名
    例 nanovllm/engine/model_runner.py:27  torch.cuda.set_device(rank)
    例 nanovllm/engine/model_runner.py:30  torch.set_default_device("cuda")
【通信】6 行，分布在 3 个文件：model_runner.py、embed_head.py、linear.py
    换卡后：通信后端换成厂商的集合通信库
    例 nanovllm/layers/embed_head.py:41  dist.all_reduce(y)
    例 nanovllm/layers/linear.py:155  dist.all_reduce(y)
【第三方 CUDA 算子库】3 行，分布在 1 个文件：attention.py
    换卡后：找厂商移植版；没有就得自己实现
    例 nanovllm/layers/attention.py:6  from flash_attn import flash_attn_varlen_func,
flash_attn_with_kvcache
    例 nanovllm/layers/attention.py:67  o = flash_attn_varlen_func(q, k, v,
【自写的 GPU kernel】13 行，分布在 1 个文件：attention.py
    换卡后：要有能用的 Triton 后端；没有就改写成厂商的 kernel 语言
    例 nanovllm/layers/attention.py:3  import triton
    例 nanovllm/layers/attention.py:4  import triton.language as tl
【图捕获】7 行，分布在 1 个文件：model_runner.py
    换卡后：换成厂商的图捕获接口，或者干脆先关掉
    例 nanovllm/engine/model_runner.py:37  self.capture_cudagraph()
    例 nanovllm/engine/model_runner.py:57  del self.graphs, self.graph_pool
【精度与数据类型】0 行，分布在 0 个文件：
    换卡后：确认硬件原生支持；不支持的要换类型或换算子
    （代码里一处也没有：数据类型来自模型的 config.json，问题藏在运行时）

合计 52 行，占全部代码的 4.4%
```

图 134-2　把 nano-vllm 搬到非 NVIDIA 的卡上要动的代码（只读源码文本统计，不运行）。1195 行里和 NVIDIA 绑定的有 52 行，占 4.4%。

52 行，4.4%。这个数字容易让人乐观，所以要马上接一句反直觉的：**这 52 行的改动量很小，它们背后的工作量很大。**

逐类看背后是什么：

- 23 行“设备与运行时”，改法确实只是换个名字（`torch.cuda.set_device` → 新设备的对应方法），前提是**框架已经认识这块卡**。这个前提背后是整个 PyTorch 的后端移植：几千个算子的实现、一个显存分配器、流和事件的语义。

- 3 行“第三方 CUDA 算子库”（`from flash_attn import ...`）是整件事里最重的一条。换卡后这一行要么指向厂商的移植版，要么你自己实现一个变长 FlashAttention。第 84 讲算过它为什么必须手写：注意力矩阵不能落地，要分块在线 softmax。
- 13 行“自写的 GPU kernel”（Triton 写的写 KV 的 kernel，第 127 讲）需要这块卡有可用的 Triton 后端；有后端也不等于性能对得上，为 NVIDIA 调的块大小和 warp 数要重调（第 144 讲）。
- 7 行“图捕获”（CUDA Graph，第 89 讲）可以先关掉，代价是 decode 的每一步多出几百次 kernel 发射的开销；在下发开销占比高的硬件上，这个代价可能是几成的吞吐。
- 6 行“通信”里的 `dist.all_reduce` 本身不用改，改的是初始化时的后端名字和它背后的通信库。
- “精度与数据类型”一行都没匹配到，但它恰恰是第 139 讲整讲在处理的问题：BF16 还是 FP16 来自模型的 `config.json`，换卡后它们在数值上的差别才显形。

所以适配工作的总量，不由仓库里有多少行带 `cuda` 的代码决定，而由**这些行背后的库在新硬件上成熟到什么程度**决定。同一个 vLLM，在一块软件栈完整的卡上可能两天跑通，在一块算子库刚起步的卡上可能要两个月。

两种接法，两种工作量分布

把 StackMap 切到另外两条路线，同一张图会换一副面孔。两条路线的区别不在硬件好坏，而在**选择在哪一层和 CUDA 生态对齐**。



图 134-3　兼容 GPGPU 路线（以 AMD 的 ROCm 为代表，海光 DCU 也走这条路）。高亮的三层是工作量的大头：框架重新编译、算子库重写或重调、kernel 语言靠工具翻译加手改。

兼容路线：提供一套和 CUDA 一一对应的运行时接口（HIP），提供一个把 CUDA 源码翻成它的工具（hipify），把 PyTorch 用这套接口重新编译一遍。这样做的回报很直接：PyTorch 对外仍然报告 `cuda` 设备，`torch.cuda.is_available()` 返回真，上面四层的 Python 代码几乎一行不用改。代价是你继承了 CUDA 的全部概念，包括那些和硬件细节绑死的概念——线程组宽度就是最典型的一个，CUDA 的 warp 是 32 个线程，AMD 和海光的波前（wavefront）是 64 个，任何假设“32”的代码都会算错（第 135、136 讲）。



图 134-4　NPU 路线（以华为昇腾的 CANN 为代表）。高亮的三层是和 GPGPU 路线最不一样的地方：图模式成了主路径，kernel 语言是另一套编程模型，运行时接口完全不同。

重写路线：硬件本身就不是 SIMT 的，kernel 里要显式安排数据在片上缓冲之间的搬运，CUDA 的 kernel 没法翻译，只能按厂商的语言重写。框架这一层用 PyTorch 的 `PrivateUse1` 机制注册一个新设备名（第 137 讲），`.cuda()` 变成 `.npu()`，所有写死 `cuda` 的地方都要改。换来的好处是不必背 CUDA 的历史包袱，而且这类硬件往往更吃“整图下发”，图编译那一层能做更激进的融合（第 138 讲）。

两条路线的工作量分布，也就是 StackMap 里“沿用 / 翻译 / 重写”的那一列：兼容路线把工作量压在算子库和 kernel 上，上层基本不动；重写路线在每一层都要出力，但每一层都能按自己的硬件重新设计。

注意

“兼容”不等于“不用改”。hipify 能翻译的是 API 名字，翻不动三样东西：内联汇编（PTX 是 NVIDIA 的指令集）、假设线程组宽度是 32 的写法、CUDA 独有的库（cuDNN 的某些算子、cuSPARSE 的某些格式）。第 136 讲会逐行看这三类。

按什么顺序做

拿到一块新卡，最常见的失败方式是一上来就调性能：还没确认输出是对的，就在量 kernel 耗时，最后发现快了三成的那个 kernel 算错了。可行的顺序是三段，每段有明确的验收标准：

阶段	做什么	验收标准	对应的讲
跑通	装好驱动和工具包，框架能看见卡；模型能前向一次，不报错	`单卡单请求能生成 token`，缺失算子都有去处	136、137、140
跑对	和一个可信的参考比对输出	逐层误差在阈值内，贪心生成逐 token 一致，评测分数不掉	139
跑快	找瓶颈、改 kernel、调通信、开图模式	达到目标 TPOT 和吞吐，和理论上限的差距可解释	142–147

这个顺序里有两条不能换：**“跑对”一定在“跑快”之前**，因为优化几乎每一步都在改变数值（融合改变累加顺序，低精度改变舍入，第 139 讲），没有基线就无法判断是优化引入的错误还是本来就有的误差；**“缺失算子有去处”一定在“跑对”之前**，因为一个悄悄回退到 CPU 的算子既会让性能面目全非，也可能掩盖精度问题（第 140 讲）。

第三阶段的内部也有顺序，第 142 讲会用 profiler 的时间线来讲：先看下发开销（CPU 是不是跟不上），再看访存，再看计算，最后看通信。顺序的理由是性价比——关掉一个不必要的同步、开上图模式，这类改动一行代码就能换来几成，而重写一个 kernel 要几天。

什么时候这张图会翻转

上面说“模型代码和引擎能复用”，这是针对当下主流的稠密模型和标准注意力说的。有三种情况会让结论变样：

模型用了专有算子。DeepSeek-V3 的 MLA（第 40 讲）需要一个把压缩、还原和注意力融在一起的 kernel；MoE 的 All-to-All（第 108 讲）在 DeepEP 这样的库里和网卡深度绑定。这类模型在新硬件上的“适配”，实际上是把几个最关键的 kernel 重写一遍，上层代码复用的比例反而最低。

负载偏向 decode。decode 的每一步都是小矩阵乘加大量访存（第 68 讲），kernel 本身很短，发射开销和同步开销占比很高。这时候图模式、下发路径的长度、甚至 Python 侧的代码效率都会直接变成吞吐差异，而这些都在框架和运行时那两层——不在算子库里。第 141 讲会在本机的

Apple MPS 上实测这件事：同一个模型，只是去掉每层一次“把一个数拷回 CPU”的同步，decode 每步就从 55 ms 降到 49 ms。

要多卡。单卡跑通和八卡跑通是两件事。通信库的拓扑感知、P2P 能不能用、自定义 All-Reduce 在什么消息大小下才划算（第 146 讲），这些问题在单卡阶段完全看不见。

回顾

换一块卡，要重做的不是整个栈，而是九层里接口下方的实现。上面四层（模型、引擎、框架接口、图表示）设计上就和硬件无关，所以 nano-vllm 一千多行里只有 4.4% 需要改；但这 4.4% 背后是框架后端、算子库、kernel 编译器、通信库这几件大工程，它们的成熟度才是适配周期的真正决定因素。两条技术路线的差别在于选择在哪一层对齐 CUDA 生态：兼容路线让上层几乎不改、把压力集中在算子库和 kernel，重写路线每层都要出力、但能按自己的硬件重新设计。工作顺序是跑通、跑对、跑快，三段不能颠倒。

下一讲往硬件里面看一层。同样是“加速器”，GPGPU 和 NPU 的计算单元、线程组宽度、片上存储组织得很不一样——而这些差异会直接改变第 62 讲那张 Roofline 图的形状，以及一个算子该怎么写。一块卡的波前是 64 个线程而不是 32 个，对 kernel 来说到底意味着什么？

第135讲 国产与 NVIDIA 硬件差异

重点

62 Roofline 模型　84 FlashAttention　86 CUDA 编程模型　134 换一块卡要重做什么

上一讲把软件栈拆成了九层，其中算子库和 kernel 那两层要重写。重写之前得先知道：这块卡和 H100 到底哪里不一样？“不一样”不是指快慢——参数表上的峰值算力和带宽只是两个数——而是指**同一个算子在这块卡上该怎么组织**。

这一讲回答三件事：两条硬件路线（GPGPU 和 NPU）的内部结构差在哪；线程组宽度从 32 变成 64、片上存储从 228 KB 变成 64 KB，对一个矩阵乘或注意力 kernel 具体意味着什么；这些差异怎么改变第 62 讲那张 Roofline 图上各类算子的落点。

两条路线：一块卡里面的分工

一块加速器里面有什么

对照

GPGPU（SIMT）　NPU / DSA（显式搬运）



显存 HBM：容量几十到几百 GB，带宽几 TB/s。第 62 讲的 Roofline 里，它的带宽是那条斜线的斜率。
编程模型：一个 kernel 启动成千上万个线程，每 w 个线程编成一个线程组（NVIDIA 的 warp 是 32，AMD 和海光的波前是 64），组内所有线程同一拍执行同一条指令。搬数据也由线程自己写：从显存读到寄存器，或者读到共享内存再彼此共享。

图 135-1　GPGPU 路线的卡内结构（示意图，不按比例）。点任意部件看它负责什么；切到另一条路线，看同一件事由谁来做。

GPGPU 这条路线上，NVIDIA 的 SM、AMD 的 CU、海光 DCU 的计算单元，分工是一样的：**一个 kernel 启动成千上万个线程，线程按固定宽度编成线程组，组内所有通道同一拍执行同一条指令**（第 86 讲的 SIMT）。搬数据的代码由线程自己写：谁读哪一段显存、要不要先搬进共享内存给同块的线程共用，都写在 kernel 里。矩阵乘法的峰值算力来自专门的矩阵核心，它按固定形状吃矩阵块（第 87 讲）。



图 135-2　NPU / DSA 路线的卡内结构（示意图，不按比例）。核的数量比 GPGPU 少一个量级，但每个核里有明确分工的三类单元和几级软件管理的缓冲。

NPU（也叫 DSA，领域专用架构）这条路线放弃了 SIMT。一个核里是三类单元各管一摊：**标量单元**管控制流和地址，**向量单元**管逐元素运算和归约，**矩阵单元**管矩阵乘；数据进出片上缓冲由专门的**搬运单元**负责，kernel 里要显式写“把这块搬进来、算、搬回去”，并自己把搬和算流水起来。写这类 kernel 的感觉更接近写 DMA 驱动，而不是写并行循环。

两者的差别带来两个直接后果。

一是**代码不能翻译，只能重写**。CUDA kernel 的核心是 `threadIdx` 和 `blockIdx`：每个线程根据自己的编号算出负责哪个元素。NPU 上没有这个概念，所以第 136 讲说的 hipify 这类工具在这条路线上完全派不上用场。

二是**“把卡占满”的办法不同**。GPGPU 靠块数：132 个计算单元要 132 个以上的线程块才能都有活干，块数不够就得 split-K（第 87 讲）。NPU 的核只有几十个，一个核的矩阵单元却更大，所以并行度要靠“每个核处理一个足够大的块”获得；块太小时，吃亏的不是空闲的核，而是搬运和算没法重叠起来。

线程组宽度：32 和 64 差在哪

GPGPU 路线内部最常踩的坑，是线程组宽度。NVIDIA 的 warp 是 32 个线程；AMD 和海光的波前（wavefront）是 64 个。海光 vLLM 插件里的小 M 矩阵乘 kernel 开头就写着 `constexpr std::int32_t kWaveSize = 64;`（第 151 讲会读这个 kernel）。宽度不同，四处要改：

跨通道归约的步数。一个线程组内做求和（softmax 的分母、RMSNorm 的平方和），用的是跨通道移位指令，步数是 $\log_2 w$：w = 32 要 5 步，w = 64 要 6 步。写死 5 次循环的代码在 64 宽度的卡上只归约了一半的通道，结果是错的——而且不会报错。

掩码的类型。投票和表决类指令返回每条通道一位的掩码：32 宽度放得进一个 `uint32_t`，64 宽度必须用 `uint64_t`。CUDA 代码里到处是 `unsigned` 的掩码，这是 hipify 翻不动、只能手改的一类。

每条通道持有的累加器个数。一条矩阵指令算一块固定形状的输出，这块输出分摊在组内所有通道的寄存器里。16×16 的输出有 256 个元素：32 条通道时每条拿 8 个，64 条通道时每条拿 4 个。

块内的线程数怎么凑。一个线程块常用 256 个线程：在 32 宽度上是 8 个线程组，在 64 宽度上是 4 个。于是“每个线程组负责输出块的哪一片”这件事要重新分，同步的粒度也跟着变。

下面的演示把这些量放在一起。默认是一个 decode 场景：M = 16 个 token，库给的输出块是 64 × 128。

⇆ 对照

线程组宽度和片上缓冲怎么限制分块

线程组宽度 w： 32（warp） 64（波前）

矩阵指令： 16×8×16 16×16×16 32×32×16

M（这一步的 token 数）= **16**

输出块 Mt × Nt = **32 × 128**

K 步长 = **64**

片上存储 = **160** KB

输出块 32 × 128，每格是一条 16×8 的矩阵指令（深色 = 这一步真的有 token，浅色 = 补零）



一条指令 16×8×16
一个线程组 32 条通道
每通道 4 个累加器（FP32）
一块要 32 条指令覆盖
一块的 A+B 分块 20 KB

M = 16，输出块 32 行：这一步**只有 16 行是真的**，块里 50% 的行在算补的零（M = 16 对指令行数 16 来说正好对齐）。
每条通道持有 4 个 FP32 累加器：指令形状不变、通道数从 32 变 64，每通道的累加器就少一半，寄存器压力减半，但要凑满一条指令需要的数据翻倍。
片上存储 160 KB ÷ 一块 20 KB：单缓冲能放 **8** 块，双缓冲（一边搬一边算）能放 **4** 块。

图 135-3　线程组宽度、矩阵指令形状、输出块和片上容量的相互制约。深色格子是这一步真的有 token 的行，浅色是补零；右侧列出每条通道的累加器数和一块分块的字节数。把 M 拖到 16、输出块拖到 256 × 256，看浪费掉多少。

同样的账用代码算一遍，顺便把它和 Roofline、FlashAttention 的块大小串起来。前两行参数照抄第 60 讲用的 H100 官方规格，后两行是按两条路线的量级设的参数——**各代卡的具体规格以官方文档为准，把这几个数换掉，后面所有结论跟着变**：

```python
# 每行：名字、峰值算力（BF16 稠密，TFLOPS）、显存带宽（TB/s）、计算单元数、线程组宽度、每单元片上存储（KB）
# H100 用第 60 讲一直用的官方规格；另外两行是按两条路线的量级设的参数，换成你手上卡的官方规格即可
cards = [
    ("H100 SXM", 989, 3.35, 132, 32, 228),
    ("GPGPU 示例", 1300, 5.3, 304, 64, 64),
    ("NPU 示例", 400, 1.6, 24, 0, 512),
]
print("拐点（FLOP/字节）= 峰值算力 ÷ 带宽；decode 一步 M 个 token 的线性层，强度约等于 M")
for name, tf, tb, cu, w, smem in cards:
    reach = min(tf, 16 * tb)                        # M = 16 时能达到的 TFLOPS
    print(f"{name}：拐点 {tf / tb:.0f}，M=16 最多跑到 {reach:.0f} TFLOPS，占峰值 {reach / tf:.1%}")

print("\n一条 16×16×16 的矩阵指令，每条通道要持有几个 FP32 累加器")
for w in (32, 64):
    print(f"  线程组宽度 {w}：{16 * 16 // w} 个")

print("\nM=16 的 decode，输出块越大、补的零越多")
for mt in (16, 64, 128, 256):
    print(f"  输出块 {mt} 行：有效 {min(16, mt)} 行，浪费 {1 - min(16, mt) / mt:.1%}")

print("\nFlashAttention 的 K/V 块能开多大（head_dim 128，BF16，K、V、Q 各一份）")
for name, tf, tb, cu, w, smem in cards:
    print(f"  {name}：片上 {smem} KB → {smem * 1024 // (3 * 128 * 2)} 行")
```

```
拐点（FLOP/字节）= 峰值算力 ÷ 带宽；decode 一步 M 个 token 的线性层，强度约等于 M
H100 SXM：拐点 295，M=16 最多跑到 54 TFLOPS，占峰值 5.4%
GPGPU 示例：拐点 245，M=16 最多跑到 85 TFLOPS，占峰值 6.5%
NPU 示例：拐点 250，M=16 最多跑到 26 TFLOPS，占峰值 6.4%

一条 16×16×16 的矩阵指令，每条通道要持有几个 FP32 累加器
  线程组宽度 32：8 个
  线程组宽度 64：4 个

M=16 的 decode，输出块越大、补的零越多
  输出块 16 行：有效 16 行，浪费 0.0%
  输出块 64 行：有效 16 行，浪费 75.0%
  输出块 128 行：有效 16 行，浪费 87.5%
  输出块 256 行：有效 16 行，浪费 93.8%

FlashAttention 的 K/V 块能开多大（head_dim 128，BF16，K、V、Q 各一份）
  H100 SXM：片上 228 KB → 304 行
  GPGPU 示例：片上 64 KB → 85 行
  NPU 示例：片上 512 KB → 682 行
```

最后一组数值得停一下。片上存储从 228 KB 掉到 64 KB，FlashAttention 一次能装下的 K/V 行数从 304 掉到 85——这意味着扫一遍 4096 长的上下文，分块次数从 14 次变成 49 次，每次都要重做一轮在线 softmax 的更新（第 84 讲的 `FlashTileSize` 演示里能看到这个关系）。所以“把 FlashAttention 移植过来”这件事，不只是把语法翻译一遍，分块参数要按新的容量重新选。

Roofline 的形状变了

第 62 讲用拐点把算子分成两类：算术强度在拐点左边的受带宽限制，右边的受算力限制。拐点 = 峰值算力 ÷ 带宽，所以换卡就是换拐点。

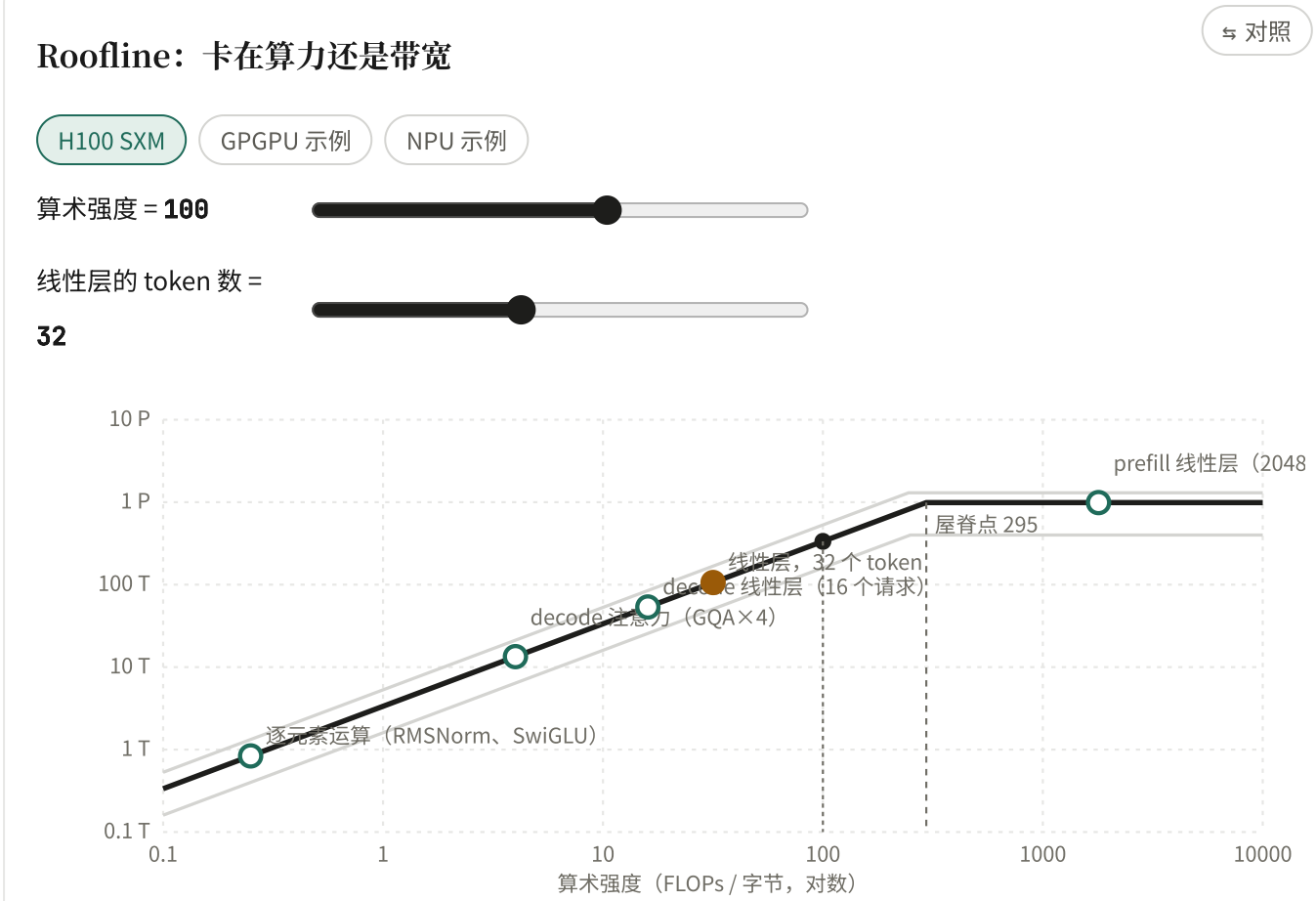


图 135-4　同一组算子落在三条路线的 Roofline 上。切换卡看拐点怎么动；拖算术强度和线性层的 token 数，找出这块卡上 decode 开始受算力限制的批大小。三组参数的来源见上面的代码注释，非 NVIDIA 的两行按官方规格替换。

有意思的是上面代码算出的那三个拐点：295、245、250。它们差别不大——因为峰值算力和带宽往往一起涨。这就引出一个反直觉的结论：**换一块卡，算子“卡在哪一边”这件事通常不变。** decode 的线性层在三块卡上都只能跑到峰值的 5% 到 7%，因为它的算术强度约等于批大小，而批大小远小于任何一个拐点。换卡不改变这个事实，第 68 讲的结论（decode 受带宽限制）在新硬件上照样成立。

真正变的是三件更具体的事：

绝对带宽。decode 一步的时间下限 = 要读的权重字节数 ÷ 带宽。带宽从 3.35 TB/s 到 5.3 TB/s，同一个模型的 TPOT 下限直接差 1.6 倍，和算力无关。

达到上限的难度。受带宽限制的 kernel 要跑满带宽，需要足够多的计算单元同时在途发出读请求（第 86 讲）。计算单元数、每单元能在途多少字节、L2 的组织方式都不一样，所以“理论带宽的几成”这个效率因子换卡后要重新测——第 67 讲强调过它，第 142 讲会用 profiler 量它。

哪些算子有现成实现。Roofline 只说上限，不说有没有人把这个上限做出来。注意力在 H100 上有 FlashAttention 的多代实现，换一块卡可能只有一个 Triton 版本，实测离上限很远。这是适配阶段最常见的差距来源，而它在 Roofline 图上完全看不见。

> **注意**
> 参数表上的峰值算力有两个容易混的点：一是稀疏算力（NVIDIA 的 Tensor Core 规格常给 2:4 稀疏下的数字，是稠密值的两倍），二是不同数据类型（FP16 / BF16 / FP8 / INT8 的峰值依次翻倍）。比较两块卡时，要确认比的是同一种类型、同样是稠密值。第 63 讲列过各精度的关系。

片上存储和互联

除了线程组宽度，还有两处差异会改变算子设计。

片上存储的容量和层次。GPGPU 上是“寄存器 + 共享内存 + L2”三级，共享内存由块内线程共享，容量几十到几百 KB。NPU 上是几级软件管理的缓冲（L1 大缓冲、紧贴矩阵单元的 L0 A/B/C），容量往往更大，但**必须显式填充**：没有“自动缓存命中”这回事，搬运的时机和粒度都写在代码里。这解释了为什么 NPU 更依赖图模式（第 138 讲）——编译器拿到整张图，才能在图的尺度上安排搬运和复用；逐个算子下发时，每个算子都得自己把数据搬进来再搬出去。

互联拓扑。第 102 讲算过，All-Reduce 的代价由每轮固定开销 α 和带宽 β 决定，而算法的选择取决于拓扑：机内全连接（每张卡到每张卡都有直连）时 Ring 和一步到位都可行；如果是环状或立方体直连，跳数就要算进延迟（第 115 讲的 `TorusVsSwitch`）。换卡时这部分完全重来：通信库换了、拓扑换了、P2P 能不能用也换了，第 146 讲专门讲。

什么时候这些差异会反过来

上面说的“NPU 核少块大、GPGPU 核多块小”，有两种负载会让结论反转。

极小 batch 的 decode。一步只有十几个 token，矩阵乘的 M 很小。这时候矩阵单元越大越吃亏：指令形状要求 M 补齐到 16 或 32 的倍数，库选的输出块可能是 256 行，于是 94% 的行在算补的零（上面代码的最后一组数）。第 151 讲会看到海光为此专门写了 $M \leqslant 64$ 的 kernel，绕开

库的大分块。**矩阵单元大是优势还是负担，取决于 M。**

长上下文的注意力。prefill 一个 128K 的上下文，注意力的计算量按序列长度的平方涨，算术强度很高，这时拼的是峰值算力和片上存储容量，片上缓冲大的设计反而占优：块开得大、扫的轮数少、在线 softmax 的更新次数少。

通信密集的 MoE。第 111 讲的结论是大 EP 的瓶颈常在网络。这时候卡内的差异几乎不起作用，决定吞吐的是互联带宽和 All-to-All 的实现质量。

要点

换卡后要重新算的，不是“算子卡在算力还是带宽”（这个结论通常不变），而是三组硬件常数：线程组宽度（决定归约步数、掩码类型、累加器分配）、矩阵指令形状与对齐（决定小 M 时补多少零）、片上存储容量（决定分块和注意力的块大小）。把这三组数填进 kernel 的分块公式，就知道该怎么重新选参数。

回顾

两条硬件路线的分工不同：GPGPU 用 SIMT，线程自己搬数据，靠很多小块占满卡；NPU 把标量、向量、矩阵单元分开，数据靠搬运单元显式进出片上缓冲，靠大块喂饱少数几个核。GPGPU 路线内部最具体的差异是线程组宽度 32 对 64，它改变归约步数、掩码类型、每通道累加器数和块内线程的分组方式。片上存储容量决定了分块和 FlashAttention 的块大小：从 228 KB 到 64 KB，一次能装的 K/V 从 304 行掉到 85 行。Roofline 的拐点在不同卡上反而接近，所以算子受带宽还是受算力限制这件事通常不变，变的是绝对带宽、达到上限的难度，以及有没有人把这个上限实现出来。

硬件差异讲完了，下一讲回到代码：手上有一份写满 `threadIdx`、`__shfl_xor` 和 PTX 内联汇编的 CUDA kernel，哪些能用工具自动翻译过去，哪些必须人工重写？

第136讲 编程模型与 CUDA 迁移

86 CUDA 编程模型　88 Triton 入门　134 换一块卡要重做什么　135 国产与 NVIDIA 硬件差异

上一讲的结论里有一条很具体：这块卡的波前是 64 条通道，不是 32 条。现在把这条结论放回代码里。手上是一个推理引擎，里面有几十个 `.cu` 文件，几千行 kernel，到处是 `threadIdx`、`__shfl_xor_sync`、`cublasGemmEx`，还有几段 PTX 内联汇编。

这一讲回答三个问题：CUDA 生态把人锁在哪里；“兼容”和“重写”两条路线各自要付出什么；一份 CUDA 代码里，哪些行能被工具自动翻译，哪些行工具能换名字但语义会出错，哪些行只能从头重写。

锁定不在语言，在三样东西

“CUDA 生态锁定”这句话常被说成“代码是用 CUDA 写的”，这不准确。CUDA C++ 本身就是 C++ 加上几个扩展（`__global__`、`threadIdx`、`<<< >>>`），换一套名字并不难。真正难搬的是另外三样：

一是调好的库。cuBLAS 里每种形状该用多大的分块、cuDNN 里每种卷积该用哪个算法、FlashAttention 里针对 Ampere 和 Hopper 分别写的两套 kernel——这些是人多年调出来的参数和代码，不是接口。接口换名字是一天的事，参数重调是几个月的事。

二是写进 kernel 的硬件常数。线程组宽度 32、共享内存 228 KB、矩阵指令 `m16n8k16`、L2 的大小。这些数字散落在每一个高性能 kernel 的分块逻辑里，而且往往没有名字，就是一个字面量（第 135 讲的 `% 32`）。

三是框架之外的那一圈工具。Nsight 的时间线、CUDA Graph、NVTX 的打点、`nvidia-smi` 的输出格式、依赖 `torch.cuda` 的第三方库。工程习惯和自动化脚本全长在这些工具上。

对应到上一讲的分层：第一样在算子库层，第二样在 kernel 层，第三样在工具层。模型和引擎那两层基本不锁。

兼容路线：hipify 能做到哪一步

兼容路线的做法是提供一套和 CUDA 一一对应的接口（HIP），再提供一个源到源的翻译工具。ROCm 的 HIPIFY 有两个版本：`hipify-perl` 按正则替换，不需要能编译就能跑，适合粗扫；`hipify-clang` 用真正的 C++ 前端解析，替换更准，但要求代码能被编译器完整解析（AMD 的 HIPIFY 文档对两者的定位就是这样区分的）。

工具翻完不等于可用。HIPIFY 的文档自己也写了这一点：迁移之后要做代码审查，把不支持的库和写法换掉，再测正确性和性能。下面逐行看一段 kernel，每行标出属于哪一类处理。

一段 CUDA kernel 逐行翻译

⇆ 对照

全部 16 行 | 只看要人工处理的

`#include <cuda_runtime.h>`	`#include <hip/hip_runtime.h>`	自动
`#include <cub/cub.cuh>`	`#include <hipcub/hipcub.hpp>`	自动
`cudaMalloc(&buf, n * 4);`	`hipMalloc(&buf, n * 4);`	自动
`cudaMemcpyAsync(d, h, n, cudaMemcpyHostToDevice, s);`	`hipMemcpyAsync(d, h, n, hipMemcpyHostToDevice, s);`	自动
`int lane = threadIdx.x % 32;`	`int lane = threadIdx.x % 32;`	要确认
`for (int i = 16; i > 0; i >>= 1)` `v += __shfl_xor_sync(0xffffffff, v, i);`	`for (int i = 32; i > 0; i >>= 1)` `v += __shfl_xor(v, i);`	要确认
`unsigned mask = __ballot_sync(0xffffffff, pred);`	`uint64_t mask = __ballot(pred);`	要确认
`__syncwarp();`	`// 波前内天然同步，按架构决定是否需要`	要确认
`asm volatile("mov.u32 %0, %%laneid;" : "=r"(id));`	`// PTX 内联汇编：整段重写`	重写
`wmma::fragment<...> a_frag;` `wmma::mma_sync(c, a, b, c);`	`__builtin_amdgcn_mmac_f32_16x16x16bf16(...)`	重写
`kernel<<<grid, 256, smem, stream>>>(p);`	`hipLaunchKernelGGL(kernel, grid, 256, smem, stream, p);`	自动
`cublasGemmEx(h, ..., CUDA_R_16BF, ...);`	`hipblasGemmEx(h, ..., HIPBLAS_R_16B, ...);`	要确认
`cudnnConvolutionForward(...);`	`// 查目标平台有没有对应算子，或改用其他实现`	重写
`ncclAllReduce(sb, rb, n, ncclBfloat16, ncclSum, comm, s);`	`rcclAllReduce(sb, rb, n, rcclBfloat16, rcclSum, comm, s);`	自动

`cudaFuncSetAttribute(kernel, cudaFuncAttributeMaxDynamicSharedMemorySize, 128*1024);`	`hipFuncSetAttribute(...) // 上限按本卡容量改`	要确认
`nvcc -arch=sm_90 ...`	`编译器 --offload-arch=gfx938 ...`	要确认

要确认（名字能换，语义要人工核对）：工具不会改这个 32，因为它只是个整数。但这块卡的波前是 64 条通道，写死 32 会把一个波前当成两个 warp 用：同步的粒度、归约的范围全错。应改成 warpSize 或按架构的常量。

这 16 行里：自动 6 行、要人工确认 7 行、必须重写 3 行。真实项目里自动的比例高得多，但剩下那些行往往集中在最关键的 kernel 里。

图 136-1　一段 RMSNorm kernel 的缩影逐行翻译。点任意一行看它为什么属于那一类；“只看要人工处理的”按钮过滤掉自动替换的行。代码是按两边公开的接口对应关系写的示意，不是某个真实项目里的片段。

挑三行细说。

`int lane = threadIdx.x % 32;`。工具不会碰它——在工具眼里这就是一个整数。但在 64 条通道的卡上，这一行把一个波前当成两个 warp 用：`lane` 的取值范围只到 31，而硬件上同一拍执行的是 64 条通道。于是下面所有“同一个 warp 内不需要同步”的假设都不成立。正确做法是用 `warpSize` 这个内建变量，或者按架构定义一个常量。AMD 的 HIP 移植文档把这件事写成一条明确的告诫：不要假设 warp 是 32 或 64，设备代码用 `warpSize`，主机侧通过设备属性查询。

跨通道归约的步数。 `for (int i = 16; i > 0; i >>= 1)` 是 32 条通道的折半归约，5 步。64 条通道要从 32 开始，6 步。少一步的后果是：每条通道手里只是半个波前的和，RMSNorm 的分母偏小，输出整体偏大——而且程序不报错、不崩溃，生成的文本看起来只是“质量差一点”。这类错误在第 139 讲的逐层比对里会第一时间暴露出来，这也是为什么“跑对”必须有一套机械的检查手段，不能靠看输出顺不顺。

表决掩码的宽度。 `__ballot_sync` 返回每条通道一位：32 条通道刚好塞进 `unsigned`，64 条通道必须用 64 位整数。沿用 `unsigned` 会静默丢掉编号 32 到 63 的通道。

> **注意**
> 这三类问题的共同点是**静默**：编译通过、运行不报错、输出有值但是错的。迁移时要主动去搜这些模式（`% 32`、`0xffffffff`、`>> 1` 的归约循环、`unsigned mask`），不能等着测试告诉你。

真实项目里这几类各有多少

上面是构造的例子。看一份真实的、已经迁移完成的代码：海光的 vLLM 插件里带 kernel 的那几个目录（第 20 篇会详细读它，这里只做文本统计）。

```
$ .venv/bin/python bench/migrate_scan.py                                    ⇆ 对照

扫了 11 个文件、4250 行（不算空行）：vllm_hcu/csrc、vllm_hcu/utils、
vllm_hcu/model_executor/layers/mhc.py
   47 处  沿用 CUDA 的名字（torch 的 CUDA 接口、at::cuda::、CUDACheck）
         例 vllm_hcu/csrc/hcu_cache_kernel.cu:1033  const hipStream_t stream =
at::cuda::getCurrentCUDAStream();
   36 处  换成 hip 前缀的运行时 API
         例 vllm_hcu/csrc/hcu_cache_kernel.cu:887  hipStream_t stream =
at::hip::getCurrentHIPStream();
    3 处  HIP 的发射宏 hipLaunchKernelGGL
         例 vllm_hcu/utils/bf16_fp32_mmac.py:278
hipLaunchKernelGGL((gemm_nt_bf16_fp32_splitk<4, 64, 64, 8>), grid,
   11 处  架构内建函数 __builtin_amdgcn_* / __builtin_hcu_*
         例 vllm_hcu/utils/bf16_fp32_mmac.py:45  reg_c =
__builtin_amdgcn_mmac_f32_16x16x16bf16(
    2 处  矩阵核心指令 mmac
         例 vllm_hcu/utils/bf16_fp32_mmac.py:45  reg_c =
__builtin_amdgcn_mmac_f32_16x16x16bf16(
    4 处  按架构分支的宏（__gfx…__、HIP_DEVICE_COMPILE）
         例 vllm_hcu/utils/bf16_fp32_mmac.py:44  #if defined(__gfx936__) ||
defined(__gfx928__)
    3 处  写死 64 的线程组宽度
         例 vllm_hcu/utils/bf16_fp32_mmac.py:87  constexpr std::int32_t kWaveSize =
64;
```

图 136-2　在海光 vLLM 插件的 C++/HIP 源码里数“CUDA 翻译痕迹”（只读文本的正则统计，提交 81cbe8f，Apache-2.0；不编译也不运行）。

这份统计有三个值得注意的地方。

“沿用 CUDA 的名字”比“换成 hip 前缀”还多（47 处对 36 处）。原因是 PyTorch 的这类构建对外仍然自称 CUDA：拿当前流的函数叫 `at::cuda::getCurrentCUDAStream()`，守卫叫 `OptionalCUDAGuard`，连 setup.py 里注册扩展用的都是 `CUDAExtension`。底层编译成的是 HIP 代码，上层的名字一个没改。这正是兼容路线最大的收益：调用方不需要知道自己在哪块卡上。

真正和架构绑定的只有十几处：11 处架构内建函数（`__builtin_amdgcn_*`、`__builtin_hcu_*`）、4 处按架构代号分支的宏、3 处写死 64 的线程组宽度。它们全部集中在一个文件——那个专门为 decode 写的小 M 矩阵乘 kernel（第 151 讲）。

这十几处恰好是性能最关键的地方。这是迁移工作的典型形状：九成的行机械替换，剩下一成在最热的几个 kernel 里，而那一成决定了性能。

重写路线：没有线程编号的编程模型

如果硬件不是 SIMT 的（第 135 讲的 NPU 路线），上面整套办法都不适用。那类 kernel 的写法不是“我是第几号线程、我负责哪个元素”，而是：

1. 声明几块片上缓冲，说明它们多大、放什么；
2. 让搬运单元把显存里的一块数据搬进缓冲（搬运途中可能还要做补齐、转置、按矩阵单元要求的排布重排）；
3. 让矩阵单元或向量单元对缓冲里的数据做运算；
4. 把结果搬回显存；
5. 把这几步排成流水：这一块在算的时候，下一块已经在搬，缓冲要开双份。

这种模型里没有 `threadIdx`，自然也没有东西可以从 CUDA 翻译过来。所以这条路线上厂商要做的事多得多：一套 kernel 语言、一个算子库、一套图编译（第 138 讲），还要把 PyTorch 的几千个算子逐个实现一遍（第 137 讲）。换来的是编译器能看到完整的数据搬运计划，可以在图的尺度上做融合和复用。

还有第三种选择，而且在实践中越来越重要：**用能生成多种后端代码的 kernel 语言**。Triton 的 kernel 写的是“一个程序处理一整块数据”（第 88 讲），不写线程编号，所以同一份 Python 代码可以由不同后端编译到不同架构。Triton 项目自己维护 NVIDIA 和 AMD 两个后端，其他厂商也在接。这条路的代价是：能表达的东西比手写汇编少，而且为一种硬件调好的块大小、流水级数到另一种硬件上要重调（第 144 讲）。

对应关系表：哪些库有对应物

迁移前先查一遍依赖，看每个库在目标平台上有什么。兼容路线的对应关系大致是这样（名字以 ROCm 这条路线为例，其他厂商各有自己的命名）：

CUDA 侧	作用	兼容路线的对应物	注意
CUDA Runtime	设备、显存、流、发射	HIP Runtime	接口一一对应，`cuda` 换 `hip`
cuBLAS / cuBLASLt	矩阵乘	rocBLAS / hipBLASLt	接口对应；默认选的分块不同，小 M 时差别很大（第 151 讲）
cuDNN	卷积等	MIOpen	覆盖的算子和格式不完全一致
CUTLASS	写矩阵乘的模板库	Composable Kernel、厂商自研模板库	概念相近，模板参数和指令形状都要重写
CUB / Thrust	并行原语	hipCUB / rocThrust	模板参数里的线程组宽度要改
NCCL	集合通信	RCCL	接口照抄，PyTorch 里后端名仍写 `nccl`（第 146 讲）
FlashAttention	注意力	厂商移植版或 Triton 实现	分块参数按片上容量重调（第 135 讲）
Nsight Systems / Compute	时间线、单 kernel 分析	rocprof 一类的工具	能力对应，输出格式和字段名不同（第 142 讲）
nvcc	编译器	基于 Clang 的厂商编译器	架构代号从 `sm_90` 变成 `gfx938` 这样的串

这张表里真正要留意的是“注意”那一列：**有对应物不代表能直接换**。rocBLAS 有 `hipblasGemmEx`，调用能跑通，但它内部按哪一代硬件调过、默认选什么分块，和 cuBLAS 完全是两套结论。第 151 讲会看到一个具体后果：decode 时 $M \leqslant 64$，库选了 256×256 的分块，于是不管 N 多大都要约 135 μs，海光只能自己写一个专用 kernel 绕过去。

为什么会有人写内联汇编

迁移时最让人头疼的是内联汇编（`asm volatile("..." )` 里的 PTX）。既然这么难搬，为什么还有人写？因为有三类事情用 C++ 表达不出来，或者表达出来编译器生成的代码不够好：

- **拿到硬件状态**：当前通道号、当前时钟、某个特殊寄存器的值。
- **精确控制指令**：让编译器一定用某一条指令（某种舍入模式的乘加、某种宽度的加载），而不是由它自己挑。
- **异步搬运的细节**：从显存直接搬进片上存储、绕过某级缓存、提前预取。这类指令往往比 C++ 的接口早出现一两代。

海光那个小 M kernel 里的 `make_buffer_resource` 和 `__builtin_hcu_raw_buffer_load_lds` 就属于第三类：手工拼出一个描述“一段显存”的四元组，再让搬运指令按它把数据直接送进片上存储（第 151 讲会逐行读）。这类代码在任何架构上都要重写，但它也解释了为什么高性能 kernel 不可能“写一次到处跑”——最后那几个百分点的性能，就藏在这些和架构绑死的指令里。

专栏

迁移完成后还有一件容易忘的事：**构建系统**。架构代号（`sm_90` → `gfx938`）、编译选项、一份二进制包含哪几个架构的机器码、JIT 编译的缓存位置，都要改。海光那个 kernel 用的是 PyTorch 的 `load_inline` 在运行时即时编译（第 151 讲），好处是不用预编译所有架构，代价是第一次调用会卡住几十秒——在服务里要放进预热阶段。

迁移该按什么顺序做

把上面的内容整理成一个可执行的顺序：

步	做什么	为什么在这一步
1	先不迁 kernel：把模型跑在框架的通用算子上（`torch.matmul`、`F.softmax`）	先拿到一个能跑、能比对的基线，哪怕很慢。第 141 讲在 MPS 上就是这样起步的
2	清点代码里和 NVIDIA 绑定的写法（设备、通信、第三方库、自写 kernel、图捕获）	第 134 讲的清单给出工作量的全貌和优先级
3	跑翻译工具，处理掉机械替换的部分	这部分占行数的九成，先清掉，剩下的才看得清
4	逐个检查“要人工确认”的行：线程组宽度、掩码、同步、共享内存容量、架构代号	这些是静默错误的来源，必须人工过一遍
5	重写“工具无能为力”的部分：内联汇编、矩阵核心指令、专有库调用	到这一步才动最难的代码，而且有基线可比
6	对齐精度（第 139 讲），再调性能（第 142 讲起）	顺序不能反

第 4 步容易被跳过，因为跑起来“看着是对的”。一个实用的办法是在这一步之前先把第 139 讲的逐层比对脚本准备好：改一个 kernel，就跑一次比对，立刻知道这一层的输出有没有偏。

回顾

CUDA 的锁定不在语言上，而在三处：调好参数的算子库、写进 kernel 的硬件常数、框架之外的那一圈工具。兼容路线提供一一对应的接口和翻译工具，让九成的行机械替换掉，代价是继承了 CUDA 的硬件假设——线程组宽度、掩码宽度、同步语义、共享内存容量，这四类是静默错误的高发区。重写路线面对的是另一种编程模型：没有线程编号，数据搬运要显式安排，CUDA 代码无从翻译，但编译器能在图的尺度上优化。真实项目的统计印证了这个形状：海光插件的 kernel 代码里，和架构强绑定的只有十几处，却全部落在最热的那个矩阵乘 kernel 里。

kernel 的事先放一放。往上一层：就算 kernel 都有了，PyTorch 怎么知道这块卡存在？`x @ w` 这句 Python 调用，是怎么找到注册在这块卡上的那份实现的？缺了一个算子又会发生什么？

第137讲 框架怎么接入新硬件

118 从权重文件到第一个 token
134 换一块卡要重做什么
136 编程模型与 CUDA 迁移

miniinfer 的模型文件里有一句 `x @ w[p + "q_proj.weight"].T`（第 118 讲）。这一句在 CPU 上能跑，在 Apple MPS 上能跑，换成一块国产加速器也应该能跑——前提是 PyTorch “认识”这块卡。

认识是什么意思？具体说就是：`x.device` 能报出这块卡，`x @ w` 能找到一份在这块卡上执行的实现，`x.to(device)` 能把数据搬上去，显存能分配和回收，流和事件能同步。这一讲把这条路径拆开，看每一站是谁的责任；然后用本机的 PyTorch 真的打印出分发表，注册一份自己的实现，看调用被转到哪里；最后看缺一个算子时有哪几种结局。

x @ w 的六站



图 137-1　沿用 CUDA 分发键的接法（ROCm 式，海光 DCU 走这条路）。点“下一站”逐站走，或点任意一站看它做了什么。

关键的一站是第三站：**分发器**（dispatcher）。PyTorch 的每个算子（`aten::mm`、`aten::add`、`aten::_softmax` 等几千个）都有一张表，表的行是**分发键**（dispatch key）。键有三类：设备键（CPU、CUDA、MPS、XPU、PrivateUse1……）、功能键（Autograd、Autocast、量化、稀疏……）、以及兜底的复合键。一次调用进来，分发器按张量的设备和当前打开的功能算出要走哪一行，然后调用那一行注册的函数。

所以“让 PyTorch 认识一块新卡”，技术上就是**在这些表里填出一列**。填法有两种。

沿用 CUDA 键。把 PyTorch 整个用兼容接口（HIP）重新编译一遍，注册时还是注册到 CUDA 键上。好处立竿见影：`torch.cuda.is_available()` 为真，`tensor.cuda()` 可用，`torch.cuda.synchronize()`、`torch.cuda.mem_get_info()` 全部照旧，上层几万行 Python 代码一个字都不用改。第 150 讲会看到海光的 vLLM 插件把平台声明成

`device_type="cuda"`、`dispatch_key="CUDA"`，正是因为底下这份 PyTorch 就是这样编出来的。代价是语义上的别扭：报错信息里写着 CUDA，环境变量叫 `CUDA_VISIBLE_DEVICES`，但机器里根本没有 NVIDIA 的卡。

注册新设备（PrivateUse1）。PyTorch 预留了一个叫 `PrivateUse1` 的键给第三方后端用。厂商把它改名（`torch.utils.rename_privateuse1_backend("npu")`），自动生成 `Tensor.npu()`、`Tensor.is_npu`、`Module.npu()` 这些接口，然后把自己的算子实现逐个注册到这个键上。PyTorch 官方教程里给的步骤就是这几步，外加一句重要的提醒：注册算子的同时**要注册回退**，否则没实现的算子会直接报错。



图 137-2　注册新设备的接法（PrivateUse1 式，华为昇腾的 torch_npu 走这条路）。前两站和上一张图完全一样，从第三站开始分叉。

两种接法的取舍：

	沿用 CUDA 键	注册新设备
上层代码	几乎不改	所有 `cuda` 字样要改
第三方库（写死 `torch.cuda` 的）	直接能用	要打补丁或等上游支持
语义	名字和实际硬件不符	名字干净
要实现的范围	算子 + 库，运行时接口对齐 CUDA	算子 + 分配器 + 流 + 事件 + 拷贝 + 工具链
适合	硬件是 GPGPU、编程模型接近 CUDA	硬件不是 SIMT，或希望长期自成体系

真的打印一次分发表

上面的描述可以在本机验证。下面这段脚本在 CPU 和 Apple MPS 上真实运行：打印 `aten::mm` 的分发表、查几个算子在各键上有没有实现、给 MPS 换一份 `mm` 的实现看调用落到哪里、再注册一个新设备名。

```
$ .venv/bin/python bench/dispatch_probe.py                                  ⇆ 对照

一、aten::mm 注册在哪些分发键上（只列设备键和回退键）
  CPU                  RegisterCPU_0.cpp:3246
  MPS                  RegisterMPS_0.cpp:12779
  Meta                 _meta_registrations.py:53
  SparseCPU            RegisterSparseCPU_0.cpp:1319

二、几个算子在各个键上有没有实现（有 = 这个键上注册了一份真正的实现）
  算子                                                  CPU     MPS    CUDA      复合
  aten::mm                                              有       有       —       —
  aten::_scaled_mm                                      有       —       —       —
  aten::_int_mm                                         有       有       —       —
  aten::_scaled_dot_product_flash_attention             —       —       —       —
  aten::rms_norm                                        —       —       —       有
  CUDA 一列全是空：本机这份 PyTorch 没有编进 CUDA 实现，分发表里就没有这个键的条目。
  rms_norm 一行三个设备键都是空，但它能在任何设备上跑：它是复合算子，最后一列说明它被拆成 pow、
mean、rsqrt、mul 再分发。

三、分发是按设备键查表：给 MPS 换一份 mm 的实现，看调用被转到哪里
  注册之后 x @ w：进了我们的实现 1 次，输入形状 ((2, 3), (3, 4))，输出 (2, 4)，每个元素都是 3:
True
  注销之后再算一次：进我们的实现 1 次（没有再进），结果仍然对：True
  一块新卡的接入就是这件事的规模化：几千个算子逐个注册到自己的键上。

三之二、注册一个全新的设备名（PrivateUse1）
  rename + 生成接口之后，Python 侧多出来的东西：Tensor.mychip → True ，Tensor.is_mychip →
True ，Module.mychip → True
  但 torch.empty(2, 2, device='mychip') 还不能用：显存分配器、流、事件、拷贝都要在 C++ 里实
现。
  这就是 ROCm 式接法（沿用 CUDA 键）和 PrivateUse1 式接法的分界：前者白拿 CUDA 那一套，后者从零
铺。

四、缺算子时的两种结局（以 MPS 上没有的 aten::_scaled_mm 为例）
  直接调用： The operator 'aten::_scaled_mm' is not currently implemented for the MPS
device
  打开 PYTORCH_ENABLE_MPS_FALLBACK=1 后，PyTorch 会把它悄悄搬到 CPU 上算完再搬回来，
  正确性没问题，但一次调用要多两趟拷贝——第 140 讲量过这笔账。
```

图 137-3　在本机（PyTorch 2.15，CPU + MPS）真实跑出的分发表与注册实验。

逐段读这份输出。

第一段是 `aten::mm` 的注册位置：CPU 在 `RegisterCPU_0.cpp`，MPS 在 `RegisterMPS_0.cpp`，Meta 在 Python 里，稀疏张量还有单独一行。这就是一块卡接进来后要填的那一列——Register*你的设备*.cpp，几千行注册代码。

第二段有两个值得记住的现象。一是 `aten::_scaled_mm`（FP8 矩阵乘）只有 CPU 有、MPS 没有：这正是第 140 讲要处理的算子缺口。二是 `aten::rms_norm` 三个设备键全是空，但它在任何设备上都能跑——因为它注册在**复合键**上：PyTorch 把它拆成 `pow`、`mean`、`rsqrt`、`mul` 这些基础算子，再各自分发。

这第二点对适配很重要：**PyTorch 的几千个算子并不都要实现**。复合算子只要底下的基础算子齐了就自动可用。真正要厂商逐个实现的是那些“原子”算子，加上为了性能而特意覆盖的融合算子。这也解释了第 141 讲的一个结果：miniinfer 整个 Qwen3 前向只用到 31 种 aten 算子，其中要在设备上真正执行的 23 种——一个模型的算子面比“几千个”小得多（第 140 讲会细讲怎么统计）。

第三段是一次真实的注册和分发：我们用 `torch.library.Library("aten", "IMPL")` 给 MPS 键挂上自己的 `mm`，然后算 `x @ w`，调用进了我们的 Python 函数；注销之后再算一次，又回到 PyTorch 自带的实现。C++ 里的 `TORCH_LIBRARY_IMPL(aten, MPS, m)` 做的是同一件事。一块新卡的接入，就是这件事重复几千遍。

第四段是新设备名：改名并生成接口之后，`Tensor.mychip`、`Tensor.is_mychip`、`Module.mychip` 都出现了，但 `torch.empty(2, 2, device="mychip")` 还不能用——因为显存分配器、流、事件、拷贝都要在 C++ 里实现。这几行输出就是两条路线成本差异的最小演示。

缺一个算子，有四种结局

适配中最常见的状况不是“这块卡不行”，而是“这块卡的第 731 个算子没有”。



图 137-4　缺一个算子时的四条岔路。前两条是框架替你决定的，后两条要自己动手；第 140 讲会用实测数字比较后三条。

报错是默认行为，也是最该保留的行为：问题暴露在当场，不会悄悄变慢或变错。

回退到 CPU 是框架提供的兜底。PyTorch 的 MPS 后端用环境变量 `PYTORCH_ENABLE_MPS_FALLBACK=1` 打开，PrivateUse1 式的后端要自己注册一个 `fallback`。它的代价不是“慢一点”这么含糊——第 140 讲在本机量到：一次 FP8 矩阵乘，回退到 CPU 是 22 ms，留在设备上用现有算子组合是 1.5 ms，差十几倍。更麻烦的是它打断了设备上的流水：前后的算子都要等这次往返结束。

用现有算子组合是适配期最划算的办法：把缺的算子按定义拆成已有的算子。缺 FP8 矩阵乘就先反量化成 BF16 再乘，缺融合的 RMSNorm 就用 `pow`、`mean`、`rsqrt`、`mul` 写出来（这其实就是 miniinfer 的做法，第 118 讲）。数据留在设备上，正确性有保证，只是省不下本该省的带宽。

自己写 kernel 留给那些真正占时间的算子。第 83 讲的 Amdahl 定律在这里直接可用：一个算子占整个前向 2% 的时间，把它写到无穷快，整体也只快 2%。

要点

适配一个模型的现实路径是：先让每个缺失算子都有“组合”或“回退”的去处，把模型跑通跑对；再用 profiler 找出哪几个算子真正占时间（第 142 讲），只给它们写 kernel。顺序颠倒的话，很容易花一周优化一个占 1% 的算子。

自定义算子怎么挂进来

厂商写好的 kernel 要让 PyTorch 和推理引擎用上，还要过一道“注册成算子”的手续。现在的标准做法是注册**自定义算子**：给它一个名字（`myvendor::fused_rmsnorm`）、一个签名、一份实现，再加一份只算形状的“假实现”（fake / meta 实现）。假实现是给 `torch.compile` 用的——图编译时要推断每个节点输出的形状和类型，但不能真的去设备上算一遍（第 138 讲）。

海光的插件里就能看到这个模式：BF16 转 FP32 的矩阵乘用 `direct_register_custom_op` 注册成 `hcu_linear_bf16_fp32`，同时给了一个 `_linear_bf16_fp32_fake`，只按输入形状返回一个空张量（第 151 讲会逐行读）。vLLM 的插件机制里还有一类“算子插件”入口，专门用来在引擎启动时导入并注册这些算子（第 147、150 讲）。

注意

注册自定义算子时最容易出错的是**声明它改了什么**（`mutates_args`）。一个原地写 KV Cache 的算子如果声明成“不改任何输入”，图编译会认为它没有副作用，可能把它优化掉或者重排顺序，结果是 KV 写丢。这类 bug 在 eager 模式下看不见，开图模式才炸。

回顾

PyTorch 认识一块新卡，技术上是在每个算子的分发表里填出一列。两种填法：沿用 CUDA 键（重新编译一份 PyTorch，上层不改，名字别扭），或注册新设备（PrivateUse1，名字干净，分配器、流、事件、拷贝都要自己实现）。本机的实验把这条路径坐实了：`aten::mm` 在 CPU 和 MPS 各有一行注册，`_scaled_mm` 在 MPS 上没有，`rms_norm` 三个设备键都空着却照样能跑（它是复合算子），给 MPS 换一份 `mm` 的实现后调用立刻转向我们的代码。缺算子有四条岔路——报错、回退 CPU、用现有算子组合、自己写 kernel——按“先跑通、再按占比优化”的顺序选。

分发器是“一次调用一次查表”，一步 decode 有几千次这样的调用。有的硬件对这种逐个算子下发的方式很不友好，它希望一次拿到整张图。下一讲看图模式：`torch.compile` 怎么接一个新后端，整图下发到底省下了什么。

第138讲 图模式与编译

89 CUDA Graph　137 框架怎么接入新硬件

上一讲的分发器是“一次调用、一次查表、一次下发”。一步 decode 要下发多少次？拿 miniinfer 跑的 Qwen3-0.6B 数一数：28 层，每层几十个算子，一步两千多次（第 141 讲会给出准确数字：2221 次要在设备上算的调用）。每一次都要走分发、算形状、准备参数、交给驱动。

第 89 讲讲过这件事在 NVIDIA 卡上的后果和解法（CUDA Graph）。这一讲从适配的角度重看它：为什么有的硬件几乎非图模式不可；`torch.compile` 怎么接一个新后端；一个图被切成几十段会损失什么；静态形状和动态形状各自的代价。

下发开销是怎么变成瓶颈的

⇆ 对照

一步 decode：逐算子下发和整图下发

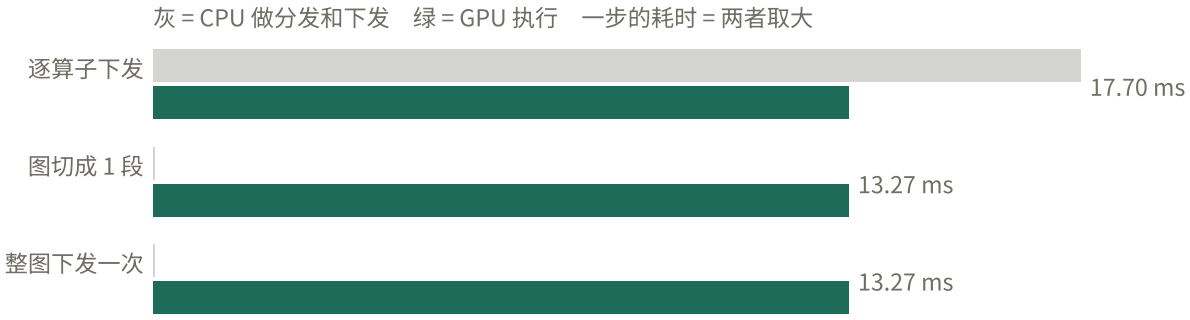
层数 = **28**

每层算子数 = **79**

图被切成 **1** 段

每次下发 CPU **8** µs

每个算子 GPU **6** µs



一步 28 层 × 79 个算子 = **2212** 个算子。逐算子下发时 CPU 要花 2212 × 8 µs = **17.70 ms**，GPU 的活是 13.27 ms：**GPU 在等 CPU**，这一步的时间由下发决定。
整图下发只付一次入口开销 20 µs，一步 13.27 ms，相比逐算子快 **1.33 倍**；切成 1 段时 13.27 ms（段入口 1 × 20 µs，中断处还有约 0 个算子退回逐算子）。

图 138-1　一步 decode 的 CPU 下发开销和 GPU 执行时间（简化模型，默认值按 Qwen3-0.6B 的 28 层 × 79 个算子）。调“每次下发 CPU 时间”，看 GPU 什么时候开始等 CPU；调“图被切成几段”，看中断的代价。

默认参数下，CPU 要花 2212 次 × 8 µs ≈ 17.7 ms 做下发，GPU 的活是 13.3 ms——GPU 在等 CPU。这不是假设：第 141 讲在 Apple MPS 上实测，一步 decode 49 ms，而把下发全算上也只有十几毫秒，说明 MPS 上单个算子的执行时间比这个简化模型里的 6 µs 大；但在带宽更高、算子更快的卡上，比例会反过来，下发就成了主角。

新硬件上这个问题往往更严重，原因有三个：

每次下发的路径更长。运行时还不成熟时，一次 kernel 发射可能多几层封装、多一次参数拷贝、多一次状态检查。

同步点更多。移植初期的实现里常有“为了保险起见”的同步。第 141 讲量到一个具体数字：MPS 上下发一个很小的加法平均 5.4 μs，而把一个数从设备拷回 CPU（`int(tensor)`）平均 141.2 μs——相差 26 倍。miniinfer 的 v1 每层做一次这种拷贝，28 层就是 28 次。

算子更碎。缺少融合算子时，一个 RMSNorm 要拆成四五个算子（第 137 讲的复合算子），下发次数成倍增加。

---

torch.compile 的三段，和后端挂在哪

`torch.compile` 做三件事：**捕获**（Dynamo 把 Python 字节码翻成一张 FX 图）、**变换**（AOTAutograd 做函数化、分解成更基础的算子）、**生成代码**（后端把图编译成可执行的东西）。厂商要接的是第三段。

一个后端就是一个函数：收到图和示例输入，返回一个可调用对象。PyTorch 文档里的自定义后端就是这个形状。我们在本机写了一个最小的后端，它什么优化都不做，只统计图的大小，然后原样执行：

bench/port_compile.py 第 1–25 行 / 共 64 行 ⇆ 对照

```
1  """第 138 讲：给 torch.compile 写一个最小的"自定义后端"，看它拿到了几张图、每张图多大、什么时候
2  后端只是一个函数：收到 FX 图（GraphModule）和示例输入，返回一个可调用对象。真实的厂商后端在这里把
3  v1 的模型每层有一次 int(pos[-1])，port/ 去掉了它；分别编译，比较图的数量。在 CPU 上运行。"""
4  import os, sys, importlib
5  import torch
6  import torch._dynamo as dynamo
7  import logging
8  torch._logging.set_logs(dynamo=logging.ERROR)          # 关掉"重新编译次数到上限"之类的警
9  ROOT = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

1 **这个脚本在做什么** 第 1-4 行

给 `torch.compile` 写一个最小的自定义后端，统计它拿到了几张图、每张图多大、什么时候重新编译。真实的厂商后端在同一个位置把图交给自己的图编译器。

2 **后端函数** 第 10-16 行

3 **按版本加载模型** 第 18-25 行

← 上一步 | 看整个文件 | 下一步 →

跑出来的结果是这一讲最重要的证据：

```
$ .venv/bin/python bench/port_compile.py
```

⇆ 对照

```
v1: 一共编译了 27 张图，每张图的节点数 [6, 72, 17, 8, 72, 17] …
   prefill，5 个 token：新编译 26 张图
   decode 第 1 步：新编译 1 张图
   decode 第 2 步：新编译 0 张图
   decode 第 3 步：新编译 0 张图
   explain：25 张图，24 处图中断
   第一处中断的原因：Unsupported Tensor.item() call with capture_scalar_outputs=False
port: 一共编译了 2 张图，每张图的节点数 [3319, 3320]
   prefill，5 个 token：新编译 1 张图
   decode 第 1 步：新编译 1 张图
   decode 第 2 步：新编译 0 张图
   decode 第 3 步：新编译 0 张图
   explain：1 张图，0 处图中断
默认（按形状特化）：共 3 张图  prefill 8 个 token：+1，prefill 16 个 token：+1，prefill 24 个 token：+0，prefill 8 个 token：+0，decode：+1，decode：+0，decode：+0
dynamic=True（一次编译，形状当变量）：共 3 张图  prefill 8 个 token：+1，prefill 16 个 token：+1，prefill 24 个 token：+0，prefill 8 个 token：+0，decode：+1，decode：+0，decode：+0
```

图 138-2　同一个 Qwen3 前向，两种写法下 torch.compile 的编译结果（本机 CPU 上真实运行）。

v1 编译出 27 张图，每张只有几十个节点（6、72、17、8、72、17……）；`explain` 报告 25 张图、24 处图中断，第一处中断的原因写得很清楚：`Unsupported Tensor.item() call with capture_scalar_outputs=False`。这就是 `int(pos[-1])` 那一行——它要求把一个张量里的值变成 Python 整数，而 Dynamo 在捕获时并不知道这个值是多少，只能在这里停下来，把图切断、回到 Python 执行一步，再重新开始捕获下一段。28 层里每层切两次，就是二十几张小图。

port 版编译出 2 张图，每张 3319、3320 个节点：整个 28 层前向是一张完整的图。差别只在 `model.py` 里的一行——`end` 由调用方以 Python 整数传进来，而不是从张量里取（第 141 讲会用 `CodeDiff` 看这处改动）。

要点

图中断（graph break）不是性能的小瑕疵，而是“有没有图模式”的开关。一个被切成 27 段的前向，对需要整图才能优化的硬件来说，等于回到了逐算子执行：段与段之间不能融合、不能跨段规划片上缓冲、不能一次下发。把图中断清零，往往是适配一块依赖图模式的硬件时最先要做的事。

常见的图中断来源，和它们的改法：

中断来源	典型写法	改法
把张量变成 Python 值	`int(x[-1])`、`x.item()`、`if x.sum() > 0:`	把这个量从外面以 Python 数传进来；或打开 `capture_scalar_outputs` 让它变成符号量
依赖张量值的控制流	`if seq_len > 512: ...`	改成和值无关的形式，或把两个分支分别编译
调用不认识的 C 扩展	第三方库的函数	注册成自定义算子（第 137 讲），图里就是一个节点
打印和日志	`print(x.shape)`	热路径里去掉

静态形状和动态形状

图编译的第二个坑是形状。第 88 讲的 `SpecializeCost` 演示讲过这笔账：按形状特化能生成最好的代码，但每换一个形状就要重新编译一次。

推理的形状变化很具体：**decode 的形状稳定**（每步 1 个 token，batch 由调度器决定，vLLM 会把它凑到几个固定档位，第 127 讲的 `GraphBucket`），**prefill 的长度每次都不同**（提示多长就是多长）。

上面脚本的最后两行是这个实验：连续 prefill 8、16、24、8 个 token，再 decode 三步。默认模式下一共编译 3 张图——第一次 prefill（长度 8）编一张按形状特化的图，第二次（长度 16）发现形状变了，重新编译一张**把长度当符号变量**的图，之后长度 24 和回到 8 都直接复用；decode（长度 1）再编一张。这是 Dynamo 的自动动态策略：先赌形状固定，发现变了就退一步，把这一维变成变量。显式写 `dynamic=True` 得到同样的 3 张图。

> **注意**
> 这个“赌一次再退一步”的策略对服务很重要：如果每个新长度都要编译一次，一个在线服务的前几百个请求会被编译拖慢，而且编译缓存会把内存吃光。接新硬件时要确认厂商的图编译器支持符号形状；只支持静态形状的后端，就必须像 vLLM 那样把形状分档（padding 到档位），用少量几张图覆盖所有情况。

为什么有的硬件更依赖图模式

同样是“整图下发比逐算子好”，不同硬件的程度差得很远。差别来自三件事。

片上缓冲要显式管理。第 135 讲说过，NPU 路线上数据进出片上缓冲是写在代码里的。逐算子执行时，每个算子都得自己把输入搬进来、结果搬回去——因为它不知道下一个算子是谁。有了整张图，编译器可以让一个算子的输出留在片上，直接给下一个算子用。省下的是显存往返，这在 RMSNorm、SwiGLU 这类带宽受限的算子上就是成倍的差别（第 83 讲的融合账本）。

下发路径更贵。整图下发把几千次下发压缩成一次，省下的 CPU 时间是按比例的。

编译器需要全局信息才能排流水。搬运和计算的重叠、双缓冲的安排，都要知道后面还有什么活。逐算子下发时每个算子各自为政，流水排不起来。

这三件事叠起来，就是为什么这类硬件的厂商往往把图模式做成**主路径**而不是优化项：PyTorch 侧接一个自己的 `torch.compile` 后端，把整张图交给厂商的图引擎，由它做融合、内存规划和下发。

GPGPU 路线上的做法不同，更接近 vLLM 在 NVIDIA 上的组合：**分段编译 + 图捕获**。注意力这类带可变长度、要读块表的算子单独留在外面，其余部分按段编译；再用 CUDA Graph 把一步录成一张图，消掉下发开销（第 89、127 讲）。海光的插件就走这条路：平台类里 `support_static_graph_mode` 返回真，图包装器直接复用 vLLM 的 `CUDAGraphWrapper`（第 150 讲会读这几行）。

适配时怎么用这一讲

接一块新卡，图模式相关的工作可以按这个顺序推进：

1. **先用 eager 跑通跑对**。图模式会掩盖问题：图中断、形状特化、融合都会改变执行顺序和数值。基线必须是 eager。
2. **数图中断**。`torch._dynamo.explain(fn)(args)` 直接给出图的数量、中断的数量和每处中断的原因。目标是把热路径上的中断清到零，像上面从 27 张图降到 2 张。
3. **确认形状策略**。后端支持符号形状就打开动态，不支持就分档并记录每档一张图，算一下编译时间和内存。
4. **最后开图捕获**。确认数值和 eager 一致（第 139 讲的比对），再量下发开销省了多少（第 142 讲的时间线）。

注意

开图模式之后，一类新错误会出现：原地写 KV Cache 的算子如果没有正确声明副作用，图可能把它优化掉或重排（第 137 讲）。症状是生成的前几个 token 对，后面开始胡言乱语——因为缓存里的内容不对。排查办法是关掉图模式对比一次，再逐段打开。

回顾

一步 decode 有两千多次算子下发，这笔 CPU 开销在新硬件上常常更大，因为下发路径更长、同步点更多、算子更碎。`torch.compile` 把图交给后端，厂商只要实现一个 `(gm, example_inputs) → 可调用对象` 的函数；但能不能拿到一张完整的图，取决于代码里有没有图中断。本机的实验给出了这个对比：miniinfer 的 v1 因为每层一次 `int(pos[-1])` 被切成 27 张小图，去掉这一行后整个前向是一张 3319 个节点的图。形状方面，Dynamo 会先按形状特化、发现变化后把那一维变成符号量，四个长度的 prefill 最终只用两张图。依赖显式搬运的硬件把图模式当主路径，GPGPU 路线更多用分段编译加图捕获。

到这里“跑通”所需的东西差不多齐了：框架认识这块卡，算子有去处，图能编出来。下一讲处理一个更难受的问题——模型跑起来了，输出却和 CPU 上不一样。到底是哪一层开始偏的?

第139讲 精度对齐

重点

10 浮点数与精度　36 Pre-Norm 与 RMSNorm　63 低精度计算　137 框架怎么接入新硬件

模型在新卡上跑起来了，第一次生成的文本是这样的：

```
'.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\
n'
```

没有报错，没有崩溃，权重也是同一份。在 CPU 上同样的代码、同样的提示生成的是 `' Paris. The capital of Italy is Rome. …'`。这一讲讲怎么从这种状态定位到具体的一行代码。

上面这段输出不是编造的：本机把 miniinfer 搬到 Apple MPS 上、把精度切到 FP16 时真的会这样（第 141 讲是完整的移植过程）。查下来问题出在 RMSNorm 的一行，和 MPS 没有关系，但只有换到低精度的硬件上才会暴露。下面就用这个真实的例子走一遍方法。

先定标准：什么叫“一致”

逐位相同（bitwise identical）这个标准在换卡时**不成立**，而且不应该作为目标。原因有三个，都和硬件有关：

累加顺序不同。浮点加法不满足结合律：`(a + b) + c` 和 `a + (b + c)` 的结果可以差一个最低位。一个矩阵乘法的 K 方向怎么切、分几个线程组归约、用不用 split-K，决定了加法的顺序（第 87 讲）。换一块卡就是换一套分块策略，顺序必然不同。

中间精度不同。BF16 的矩阵乘通常用 FP32 累加，但“FP32 累加”在不同硬件上可能是不同的实现：有的用全精度 FP32，有的用 TF32 一类的压缩格式。

同一个数学函数的实现不同。`exp`、`rsqrt`、`tanh` 在不同硬件上用不同的近似多项式或查表，最后一位常常不一样。softmax 里每个元素都要过一次 `exp`（第 6 讲），这点差别会被带进后面所有层。

所以要换一套标准，分三级：

级别	检查什么	怎么判断
张量级	每一层输出和参考的相对误差	相对误差 = ‖a − b‖ ÷ ‖b‖，在阈值内；逐层看，找第一个越界的层
token 级	贪心生成的 token 序列	和参考逐 token 比对（miniinfer 的 `tests/check.py` 一直这么做）
任务级	评测分数	几个数据集上的分数不低于参考（允许小幅波动）

三级各有各的盲区，所以要一起用。张量级能定位到层，但“误差 2e-2 算不算大”没有绝对答案；token 级直观，但贪心生成有放大效应——某一步两个候选词打分只差 0.001，一点误差就让它们换位，之后整段话都不同，看起来像灾难，实际上模型质量没变；任务级最接近用户感受，但一次评测要跑几十分钟，没法用来定位代码。

要点

换卡后的精度目标不是“逐位一致”，而是“误差来源都能解释”。每一处差异要么归因到已知的硬件行为（累加顺序、超越函数实现），要么归因到一个具体的 bug。解释不了的差异，就是还没找到的 bug。

逐层比对：找第一个偏离的层

方法很直接：同一个提示，在参考实现和待查实现上各走一遍前向，把每一层的输出都留下来，逐层比相对误差，找第一个越过阈值的层。参考实现用 CPU FP32——它慢，但确定性好，而且和 HuggingFace 的输出能对上（第 118 讲建立的基线）。

miniinfer 的 `port/model.py` 为此加了一个参数：传一个列表进去，每层的输出就依次存进去。

port/model.py 对比 v1/model.py：+16 行 / −8 行 ⇆ 对照

```
+ """移植版的 Qwen3 前向：和 v1 一样只用 torch 的基本运算；改了两处，让它在 GPU 类的后端上不做
- """Qwen3 的前向计算：只用 torch 的基本运算，一层一层写出来。"""
  import torch

  ····· 省略 1 行没变的代码

  def rms_norm(x, weight, eps):
      # 每个向量除以自己的均方根，再乘可学习的缩放（第 36 讲）
+     # 平方和先升到 FP32 再算：FP16 最大只能表示 65504，而 Qwen3 第 0 个位置的激活能到 6500
+     # 平方就是 4000 多万，在 FP16 里直接变成 inf（第 139 讲）。FP32 下这两行和 v1 完全一样
+     xf = x.float()
+     var = xf.pow(2).mean(-1, keepdim=True)
+     return (xf * torch.rsqrt(var + eps)).to(x.dtype) * weight
-     var = x.pow(2).mean(-1, keepdim=True)
-     return x * torch.rsqrt(var + eps) * weight


  ····· 省略 13 行没变的代码

          self.cfg, self.w = cfg, w

+     def attention(self, i, x, pos, end, cache):
-     def attention(self, i, x, pos, cache):
          c, w, p = self.cfg, self.w, f"model.layers.{i}.self_attn."
          T = x.shape[0]

  ····· 省略 5 行没变的代码

          q, k = rope(q, pos, c.rope_theta), rope(k, pos, c.rope_theta)
          cache.write(i, pos, k, v)                                    # 新 token 的 K、V
+         # end 由调用方用 Python 整数传进来。v1 在这里写的是 int(pos[-1]) + 1：
+         # 在 GPU 上这一步要等设备把这个数算完再拷回主机，28 层就停 28 次（第 141 讲量到一次约
-         end = int(pos[-1]) + 1
          k, v = cache.read(i, end)                                    # 取出从第 0 个位置到
          rep = c.num_heads // c.num_kv_heads                          # GQA：每组 K、V 给

  ····· 省略 11 行没变的代码


      @torch.no_grad()
+     def forward(self, tokens, start, cache, trace=None):
-     def forward(self, tokens, start, cache):
          # tokens: 这次新来的 [T] 个 token，从位置 start 开始；它们之前的 K、V 已经在 cach
+         # trace: 传一个列表进来，就把每一层的输出依次存进去（精度对齐用，第 139 讲）
          c, w = self.cfg, self.w
+         end = start + len(tokens)
+         pos = torch.arange(start, end, device=tokens.device)
-         pos = torch.arange(start, start + len(tokens), device=tokens.device)
          x = w["model.embed_tokens.weight"][tokens]
          for i in range(c.num_layers):
              p = f"model.layers.{i}."
+             x = x + self.attention(i, rms_norm(x, w[p + "input_layernorm.weight"],
-             x = x + self.attention(i, rms_norm(x, w[p + "input_layernorm.weight"],
              x = x + self.mlp(i, rms_norm(x, w[p + "post_attention_layernorm.weight'
```

```
67 +             if trace is not None:
68 +                 trace.append(x)
69           x = rms_norm(x[-1:], w["model.norm.weight"], c.rms_eps)   # 只需要最后一个位置
70           return x @ w["lm_head.weight"].T
```

图 139-1　为了做逐层比对，前向函数多了一个 trace 参数（顺带把 end 从张量里取值改成了参数，那处改动是第 138、141 讲的主题）。

脚本在本机真实跑了五组：MPS 的 FP32、BF16、FP16，CPU 的 BF16，以及 FP16 配上“有问题的那版 RMSNorm”。

```
$ .venv/bin/python bench/port_align.py
```

⇆ 对照

提示 'The capital of France is', 5 个位置；参考：CPU FP32。每层的误差 = 最差位置的相对误差
各层输出的最大绝对值（CPU FP32）：7 8 6544 6543 6542 6542 6540 6539 6535 6529 6528 6524 6522 6520 6516 6514 6513 6514 6519 6528 6534 6542 6547 6553 6558 6560 6320 1451

mps float32 fixed：逐 token 生成 32 个，前 32 个和参考一致；下一个 token 的 top-1 相同
  最差位置：2e-06 2e-06 2e-06 2e-06 2e-06 2e-06 2e-06 2e-06 3e-06 3e-06 3e-06 3e-06 2e-06 3e-06 3e-06 3e-06 3e-06 2e-06 2e-06 2e-06 2e-06 2e-06 2e-06 2e-06 2e-06 2e-06 2e-06 9e-06
  整个张量：1e-06 8e-06
  生成：' Paris. The capital of Italy is Rome. The capital of Spain is Madrid. The capital of China is Beijing. The capital of Japan is Tokyo. The capital'

mps bfloat16 fixed：逐 token 生成 32 个，前 32 个和参考一致；下一个 token 的 top-1 相同
  最差位置：2e-02 2e-02
  整个张量：9e-03 1e-02 2e-03 2e-03 2e-03 2e-03 2e-03 2e-03 1e-03 5e-04 5e-04 8e-04 1e-03 1e-03 2e-03 2e-03 2e-03 2e-03 2e-03 1e-03 2e-03 3e-03 3e-03 4e-03 5e-03 5e-03 3e-03 2e-02
  生成：' Paris. The capital of Italy is Rome. The capital of Spain is Madrid. The capital of China is Beijing. The capital of Japan is Tokyo. The capital'

cpu bfloat16 fixed：逐 token 生成 32 个，前 5 个和参考一致；下一个 token 的 top-1 相同
  最差位置：2e-02 2e-02 2e-02 2e-02 2e-02 2e-02 2e-02 1e-02 2e-02
  整个张量：9e-03 1e-02 2e-03 2e-03 2e-03 2e-03 2e-03 2e-03 1e-03 5e-04 5e-04 8e-04 1e-03 1e-03 2e-03 2e-03 2e-03 2e-03 2e-03 1e-03 2e-03 2e-03 3e-03 4e-03 5e-03 5e-03 3e-03 2e-02
  生成：' Paris. The capital of France is also the capital of the Republic of France. The capital of France is also the capital of the European Union. The capital of'

mps float16 naive：逐 token 生成 32 个，前 0 个和参考一致；下一个 token 的 top-1 不同
  最差位置：4e-03 3e-03 4e-03 1e+00 3e+00
  整个张量：2e-03 2e-03 6e-04 4e-03 5e-03 7e-03 7e-03 9e-03 1e-02 1e-02 1e-02 1e-02 2e-02 2e-02 2e-02 3e-02 3e-02 4e-02 5e-02 6e-02 7e-02 9e-02 1e-01 1e-01 2e-01 2e-01 2e-01 3e+00
  生
成：'.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n.\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n'

mps float16 fixed：逐 token 生成 32 个，前 32 个和参考一致；下一个 token 的 top-1 相同
  最差位置：3e-03 2e-03 2e-03 2e-03 2e-03 2e-03 3e-03 2e-03 3e-03 3e-03 3e-03 3e-03 3e-

```
03 3e-03 3e-03 3e-03 3e-03 3e-03 3e-03 3e-03 3e-03 3e-03 3e-03 2e-03 2e-03 2e-03 2e-03
7e-03
  整个张量: 1e-03 1e-03 8e-05 2e-04 2e-04 3e-04 5e-04 8e-04 8e-04 1e-03 1e-03 1e-03 9e-
04 6e-04 7e-04 9e-04 1e-03 9e-04 8e-04 1e-03 9e-04 9e-04 8e-04 1e-03 9e-04 1e-03 1e-03
7e-03
  生成: ' Paris. The capital of Italy is Rome. The capital of Spain is Madrid. The
capital of China is Beijing. The capital of Japan is Tokyo. The capital'
```

图 139-2　逐层误差（本机真实运行，提示 “The capital of France is”，参考是 CPU FP32）。每层给两个数：按位置分别算再取最差的那个位置，以及整个张量一起算。

把它画成曲线更容易读：

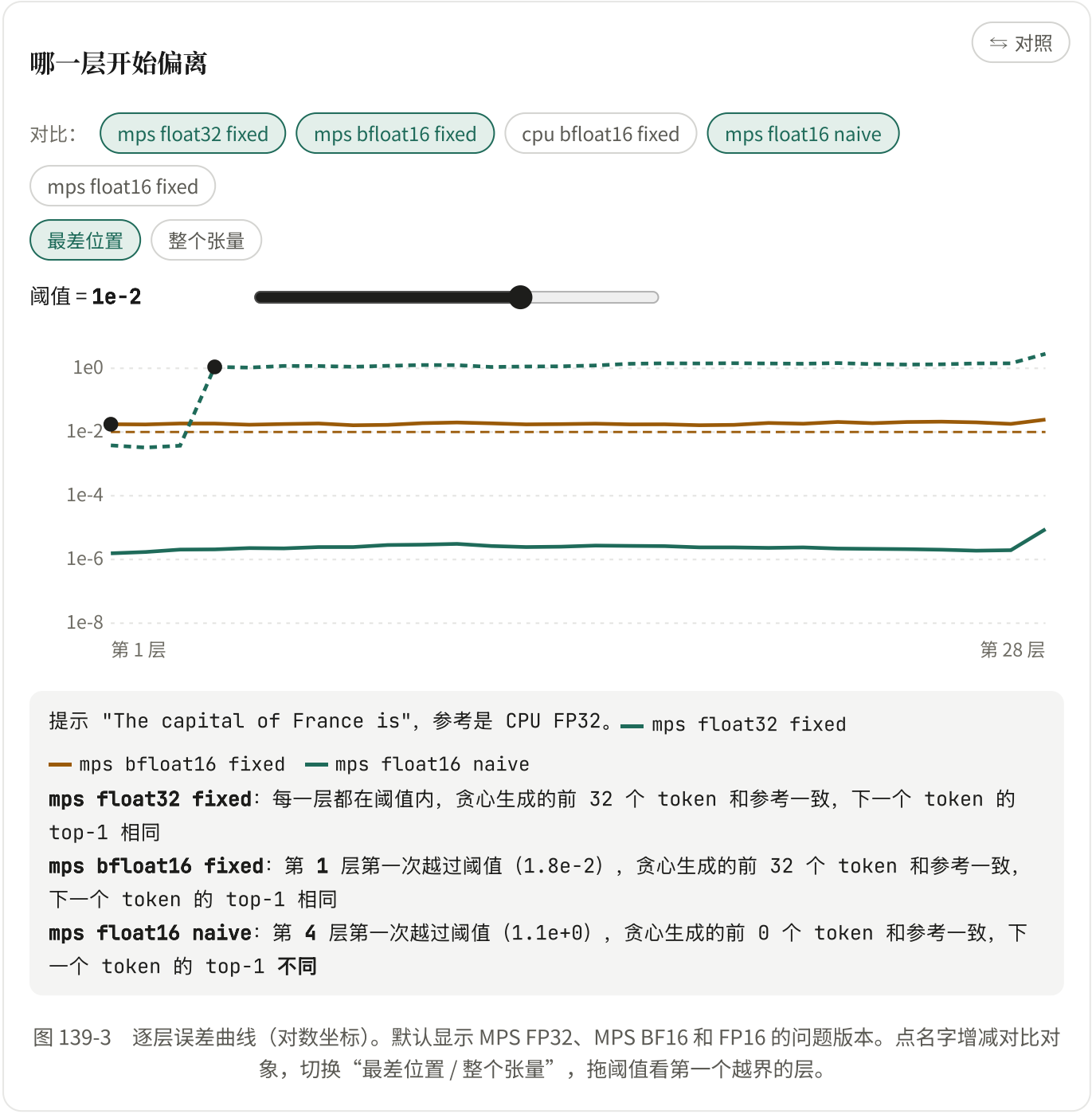


图 139-3　逐层误差曲线（对数坐标）。默认显示 MPS FP32、MPS BF16 和 FP16 的问题版本。点名字增减对比对象，切换“最差位置 / 整个张量”，拖阈值看第一个越界的层。

读这张图的四个结论：

MPS FP32 的误差稳定在 1e-6 量级，28 层下来没有累积。这是“同一种精度、不同硬件”的正常状态：误差来自累加顺序和超越函数的实现，每层引入一点，但不放大。这种情况下 token 级检查会完全一致——实测 4 条提示、每条 32 个 token，和 HuggingFace 逐 token 相同。

BF16 的误差是 2e-2 量级，而且不随层数增长。BF16 只有 8 位尾数（第 10 讲），相对误差天生在 1e-2 到 1e-3 这个量级，这是精度本身的代价，不是 bug。重点是它**不随层数放大**——曲线是平的。贪心生成也完全一致。

问题版本的 FP16 曲线从第 4 层起直接跳到 1e0。相对误差 1.0 意味着输出和参考已经毫无关系。这就是定位结果：**第 4 层**。

同一组数据的“最差位置”和“整个张量”差别很大。问题版本的整张量误差在第 4 层只有 4e-3，看起来无关痛痒；而按位置分别算时，最差的那个位置是 1.0。原因是只有 5 个位置中的 4 个坏了，而那个好的位置范数很大，把平均拉了回来。**比对时一定要按位置（或按 token）分别算再取最差**，否则会漏掉这类局部失效。

为什么是第 4 层：一个真实的 bug

定位到第 4 层之后，往层里面看。把这一层的输入打印出来，最大绝对值是 6544。

Qwen3 的激活值里有少数几个通道特别大（第 91 讲讲过这种离群值，量化方法都要对付它）。6544 这个数在 FP16 里完全放得下——FP16 的最大值是 65504。问题出在下一步：RMSNorm 要算平方的均值。

$$\mathrm{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{d}\sum_i x_i^2 + \varepsilon}} \odot w$$

6544 的平方是 4.28×10^7 ，**超过了 FP16 的上限 65504**。于是 `x.pow(2)` 得到 `inf`，均值是 `inf`，`rsqrt(inf + eps)` 是 0，输出整层变成 0——再往后，残差连接（第 29 讲）让它还能传下去，但这一层的贡献全丢了。

v1 的写法是这样的，在 FP32 和 BF16 上都没事：

```
var = x.pow(2).mean(-1, keepdim=True)
return x * torch.rsqrt(var + eps) * weight
```

BF16 为什么没事？因为 BF16 和 FP32 的指数位一样多（8 位），能表示到 10^{38} ，只是尾数少。FP16 的指数只有 5 位，上限 65504。**同样是 16 位，一个会溢出、一个不会**——第 10 讲的位布局在这里直接决定了成败。

改法是把平方和这一步升到 FP32 再算（`port/model.py` 里的 `rms_norm`）：

```
def rms_norm(x, weight, eps):
    xf = x.float()                                  # 先升到 FP32：平方不会溢出
    var = xf.pow(2).mean(-1, keepdim=True)
    return (xf * torch.rsqrt(var + eps)).to(x.dtype) * weight
```

改完之后 FP16 的曲线落到 3e-3，而且比 BF16 更低（FP16 尾数 10 位，比 BF16 的 7 位多）；贪心生成的 32 个 token 和参考完全一致。这一处改动在 LayerDrift 里可以直接对比：勾上 `mps float16 naive` 和 `mps float16 fixed` 两条线。

> **注意**
> 这类“溢出型”错误有三个特征，值得记住：换精度才出现；在残差网络里不会崩溃，只是某层失效；整张量的平均误差看不出来。所以**换精度之后必须重跑逐层比对**，不能因为“上一个精度是对的”就跳过。现实里的高发位置是所有要做平方或求和的地方：RMSNorm、注意力打分、softmax 的分母、loss。

其他常见的偏离来源

把这几年适配中反复出现的来源列一遍，按定位的难度从易到难：

算子级的实现差异。某个算子在新卡上的实现有 bug 或语义不同（边界处理、NaN 的传播、`index_put` 的重复下标）。定位办法是单算子对比：同样的输入喂给 CPU 和新卡的同一个算子，比输出。第 140 讲统计的算子清单正好可以拿来做这件事——逐个算子跑一遍对比，比在整个模型里猜要快得多。

RoPE 的角度。角度表是用 `arange` 和幂运算算出来的（第 37 讲），有的实现里用 FP16 存角度，长上下文时位置 10 万以上的角度精度不够，注意力就偏了。症状是短提示正常、长提示崩坏。

掩码和 −inf。因果掩码把分数设成 `-inf`（第 27 讲）。有的硬件上 `-inf` 参与乘法会产生 NaN，有的实现用一个很大的负数（−65504）代替，在 FP16 里再减一点就溢出。症状是某些长度下出现 NaN。

归约的范围写错。第 136 讲讲的线程组宽度问题：归约少走一步，结果偏小。症状是误差稳定在某个倍数上（例如所有值都偏大约 1.4 倍），而不是随机抖动。

权重加载。权重在新环境里被转换过一次（BF16 → FP16、转置、分片），某一层的权重错了。症状是某一层的误差突然很大、其他层正常——和上面第 4 层的例子在曲线上很像，所以两者要一起排查：先确认权重的字节和参考一致，再看计算。

KV Cache 的写入和读取。写错位置或读少了一段。症状很特别：第一个 token 对，后面逐渐跑偏（第 141 讲的移植里要重点测这条）。

阈值定多少

逐层比对的阈值不能拍脑袋，按精度定：

待查实现的精度	相对误差的量级	建议阈值	越界意味着
FP32	1e-7 到 1e-6	1e-5	一定有 bug
TF32 一类	1e-3	1e-2	可能有 bug
BF16	1e-3 到 1e-2	5e-2	可能有 bug
FP16	1e-4 到 1e-3	1e-2	可能有 bug
FP8 / INT8 权重量化	1e-2	按量化方法单独定基线	和量化基线比，而不是和 FP32 比

两条实用规则。一是**和同精度的参考比**：待查实现是 BF16 的，参考也要跑一遍 CPU 的 BF16，这样分离出“精度的代价”和“硬件的差异”。本机这组数据正好能看到这件事：MPS BF16 和 CPU BF16 的曲线几乎重合（整张量 2e-3 量级），说明差异来自 BF16 本身，不来自 MPS。二是**看形状，不只看数值**：平的曲线说明每层各自引入一点误差，正常；指数上升的曲线说明误差在被放大，要查是哪一层开始放大的；突然跳一个量级的台阶说明那一层有 bug。

量化模型是个例外，要单独说：量化本来就改变数值（第 90 讲），所以它的参考不是 FP32 的原模型，而是**同一套量化方法在参考平台上的输出**。miniinfer 的 v4 就是这样做的——第 123 讲里 INT8 版本和 HuggingFace 不再逐 token 一致，文章里写明了在第几个 token 开始分叉、为什么。

把比对做成例行检查

精度对齐不是一次性的工作。每改一个 kernel、每开一个融合、每打开一次图模式，都可能改变数值。所以要把它做成脚本，和压测一样例行跑：

1. 固定几个提示（短、长、中文、代码各一条），固定贪心解码。

2. 参考答案存成文件（miniinfer 的 `tests/hf_ref.json`），不要每次重新生成。
3. 每次改动后跑两级检查：逐层误差（快，几秒）和 token 级比对（慢一点，但能发现累积效应）。
4. 上线前跑一次任务级评测。
5. 把阈值写进脚本，越界就失败——而不是打印一个数字让人用眼睛判断。

专栏

海光在插件仓库里把这件事做成了公开的验证记录：`docs/validation/` 下每个配置一份，记录冻结的环境、硬件、后端、并行拓扑和各数据集的分数，并明确写着“模型已注册或存在配置，不代表精度验证已经通过”、“诊断配置不能计为精度验收通过”。这个态度值得抄：**适配到什么程度，要有可追溯的记录，而不是一句“跑通了”**。

回顾

换卡后的精度目标是“误差来源都能解释”，而不是逐位一致——累加顺序、中间精度、超越函数的实现都会带来最低位的差别。检查分三级：逐层相对误差（定位到层）、token 级比对（看累积效应）、任务级评测（接近用户感受）。逐层比对时要按位置分别算再取最差，否则会漏掉局部失效：本机那个 FP16 的例子里，整张量误差只有 4e-3，而最差位置已经是 1.0。定位到第 4 层之后，原因是 RMSNorm 在 FP16 里算平方和溢出——激活值 6544 的平方超过了 FP16 的上限 65504，而 BF16 因为指数位更多完全没事。阈值按精度定，并且要和同精度的参考比，把“精度的代价”和“硬件的差异”分开。

还有一类问题比精度更早出现：某个算子在这块卡上根本没有。下一讲把一个模型要用的算子完整清点一遍，看缺口有多大、每个缺口该用哪种办法补。

第140讲 算子缺口：盘点与自写

83 算子融合　84 FlashAttention　137 框架怎么接入新硬件　139 精度对齐

“这块卡支持 PyTorch 吗？”这个问题没法用是或否回答。PyTorch 有几千个算子，一块新卡的后端实现了其中一部分；而一个具体的模型只用到其中很小一部分。两个集合的差集，就是**算子缺口**。

这一讲做三件事：把一个模型真正用到的算子完整数出来（不靠猜，靠在分发器那一层截住每次调用）；对照目标平台的支持情况，列出缺口；对每个缺口在三种办法里选一种——用现有算子组合、回退到 CPU、自己写 kernel——并用实测数字说明这三种的代价差多少。

怎么把算子数清楚

最可靠的办法是在分发器那一层记账：模型跑一次前向，每一个落到 aten 的调用都记一笔。PyTorch 提供了 `TorchDispatchMode`，在它的作用域里，所有算子调用都会先经过我们的函数。

脚本很短：进入这个模式，跑一次 prefill 和一步 decode，把算子名和次数记成一个计数器。顺便把“回退到 CPU”的警告也捕获下来——这类警告平时会被日志淹没，而它恰恰是性能问题的源头。

```
$ .venv/bin/python bench/port_ops.py                                    ⇆ 对照
一、Qwen3-0.6B 在 MPS 上用到 31 种 aten 算子；一步 decode 调用 3932 次，其中视图类 1711 次，要在设备上算的 2221 次（23 种）
   要算的：mul×590、add×225、mm×197、pow×169、mean×113、rsqrt×113、arange×85、div×84、reciprocal×56、_to_copy×56、cos×56、sin×56、sub×56、cat×56、index_put_×56、clone×56、bmm×56、le×28、bitwise_not×28、masked_fill×28、_softmax×28、silu×28、index×1
   视图类：permute×477、view×448、unsqueeze×448、slice×169、select×112、expand×56、lift_fresh×1
   回退到 CPU 的算子：0 个
二、推理引擎常用的几个算子在 MPS 上（torch 2.15.0）：
   支持     bf16 矩阵乘
   支持     FP8 类型转换
   不支持   FP8 矩阵乘 _scaled_mm：NotImplementedError
   支持     INT8 矩阵乘 _int_mm
   支持     MoE 分组矩阵乘 _grouped_mm
   支持     注意力 scaled_dot_product_attention
   不支持   FlashAttention 内核 _scaled_dot_product_flash_attention：NotImplementedError
   不支持   FP64：TypeError
三、补上 FP8 矩阵乘（M=16, K=N=4096，模拟 decode 时的一个线性层）：
     22273 μs    相对误差 3.78e-02    回退到 CPU
      1459 μs    相对误差 3.78e-02    组合：反量化 + BF16 矩阵乘
       526 μs    相对误差 2.88e-03    对照：不量化，BF16 矩阵乘
    （相对误差 = ‖结果 - FP32 精确值‖ ÷ ‖FP32 精确值‖；前两种的误差来自 FP8 量化本身，两者应当接近）
```

图 140-1　Qwen3-0.6B 在 Apple MPS 上的算子清单、几个常用算子的支持情况，以及补上缺失算子的两种办法（本机真实运行，PyTorch 2.15）。

第一段输出就把“几千个算子”这个印象校正了：

- **一共只用到 31 种 aten 算子**。一个标准的稠密 Transformer 推理，用到的算子面就这么小。
- **一步 decode 调用 3932 次，其中 1711 次是视图类**（`view`、`permute`、`unsqueeze`、`slice`、`select`、`expand`）。视图算子只改形状、步长这些元数据，不读写数据、不在设备上启动 kernel——所以**它们不需要厂商实现 kernel**，而且在数“下发次数”时要把它们和真正要算的分开。
- **真正要在设备上算的是 2221 次、23 种**。这 23 种就是适配一个 Qwen3 类模型的最小算子集。

要点

盘点算子时要把三类东西分开：视图类（不需要 kernel）、复合算子（会被拆成基础算子，第 137 讲）、原子算子（必须有一份实现）。只有第三类才构成缺口。一个模型的最小算子集通常是几十种，不是几千种。

有了清单，就能排优先级。调用次数多不等于占时间多：`mul` 被调用 590 次，但每次都是逐元素运算；`mm` 只有 197 次，却几乎占掉一步的全部时间（一步要读一遍 596 M 个参数的权重，第 141 讲量过这笔账）。下面这张清单按“估计的时间占比”排序，可以切换成按调用次数排：

一个模型要用哪些算子，缺了怎么办

⇆ 对照

按时间占比排 | 按调用次数排 | 只看要动手的

mm（矩阵乘）	62%	197 次	已支持
bmm（批矩阵乘，注意力打分与加权）	9%	56 次	已支持
mul / add / sub / div（逐元素）	8%	955 次	已支持
pow + mean + rsqrt（RMSNorm 拆开）	6%	395 次	可组合
index_put_（写 KV Cache）	5%	56 次	已支持
_softmax	3%	28 次	已支持
cos / sin / cat（RoPE）	2%	168 次	可组合
silu（SwiGLU 的激活）	2%	28 次	已支持
masked_fill / le / bitwise_not（因果掩码）	1%	84 次	可组合
_scaled_mm（FP8 矩阵乘）	–	– 次	需自写
_scaled_dot_product_flash_attention	–	– 次	需自写
分组矩阵乘（MoE 专家）	–	– 次	需自写

mm（矩阵乘）（已支持）：一步要读一遍除嵌入表以外的全部权重，时间几乎都在这里。任何平台都会优先把它做好；要关心的是它在小 M 时选的分块对不对（第 143、151 讲）。

清单里已支持 6 类、可用现有算子组合 3 类、需要自写或替代 3 类。调用次数是本机 MPS 上一步 decode 的真实统计，时间占比是按口径估的，用来排优先级。

图 140-2　Qwen3-0.6B 一步 decode 的算子清单。调用次数是上面脚本的真实统计；时间占比按文中口径估算，用来排优先级（精确占比要用 profiler 量，第 142 讲）。点一行看这个算子缺了该怎么办；“只看要动手的”过滤出需要组合或自写的。

对照平台：缺口在哪

清单的另一半是平台支持情况。查法有两种：用 `torch._C._dispatch_has_kernel_for_dispatch_key` 逐个查（第 137 讲用过），或者直接试着跑一次。上面脚本的第二段是后者——挑几个推理引擎常用的算子，在 MPS 上各跑一次：

算子	MPS 上	对推理意味着什么
BF16 矩阵乘	支持	主路径可用
FP8 类型转换	支持	能存成 FP8，但……
FP8 矩阵乘 `_scaled_mm`	**不支持**	FP8 量化没法原生算（第 145 讲）
INT8 矩阵乘 `_int_mm`	支持	INT8 量化路径可用
MoE 分组矩阵乘 `_grouped_mm`	支持	MoE 可用
`scaled_dot_product_attention`	支持	有一个融合注意力的入口
FlashAttention 内核	**不支持**	上面那个入口在 MPS 上走的是别的实现
FP64	**不支持**	任何用到双精度的代码都要改

这张表的形状很典型：**基础算子齐全，低精度和融合算子有缺口**。适配一块新卡时，缺口几乎总是集中在三个地方——量化相关的矩阵乘、融合的注意力、MoE 相关的分组算子。这也正是第 19 篇几讲要分别处理的内容。

> **注意**
> “支持”有两个层次，不要混。一是**有实现**（能跑出正确结果），二是**有好实现**（接近硬件上限）。`scaled_dot_product_attention` 在 MPS 上能跑，但它不是 FlashAttention；INT8 矩阵乘能跑，但不代表它跑在矩阵核心的 INT8 峰值上。盘点阶段先确认第一层，调优阶段（第 142 讲起）再确认第二层。

三种补法，差多少

缺一个算子有三种补法，第 137 讲列过。现在用实测数字比较它们。例子是 FP8 矩阵乘：M = 16、K = N = 4096，相当于 decode 时的一个线性层。参考答案用 FP32 在 CPU 上算。

上面脚本的第三段给出了结果：

办法	耗时	相对 FP32 精确值的误差	评价
回退到 CPU	22273 µs	3.78e-2	正确，但慢了十几倍，而且打断设备上的流水
用现有算子组合（反量化成 BF16 再乘）	1459 µs	3.78e-2	数据留在设备上，误差和回退完全一样
对照：不量化，直接用 BF16 权重	526 µs	2.88e-3	最快、最准，但权重占的显存是 FP8 的两倍

三行数字里有三个结论。

回退到 CPU 的代价是 15 倍，不是“一点点”。22 ms 里绝大部分是两趟拷贝和 CPU 的计算。更糟的是它的连带效应：设备上的流水被打断，前后的算子都要等这次往返结束；如果这个算子在每层都出现，28 层就是 28 次往返。所以回退只适合两种场景——模型加载、预热这类一次性的路径，或者“先让它跑起来”的临时状态。

组合的误差和原生实现一样。这一点容易被怀疑：把 FP8 反量化成 BF16 再乘，会不会更不准？不会。误差来自**量化本身**（把权重压成 FP8 丢掉的那些位），而不是来自用什么精度去乘它们。两种办法的误差都是 3.78e-2，完全一致。所以组合是适配期的默认选择：正确性有保证，数据留在设备上。

组合省不下它本该省的东西。对照那一行是关键：不量化的 BF16 矩阵乘只用 526 µs，比“FP8 组合”的 1459 µs 还快，误差还小一个量级。为什么？因为 FP8 量化的全部意义在于**少读一半的字节**（第 93 讲），而组合的做法是先把 FP8 展开成 BF16 再乘——读的字节少了，但多了一次展开，而且乘法仍然按 BF16 算。于是省下的带宽被反量化的开销吃掉了。

> **要点**
> 用现有算子组合，能保住正确性，保不住收益。量化、融合、稀疏这些优化的收益都来自“硬件原生支持”：没有原生 FP8 矩阵乘，FP8 只省显存容量，不省带宽，也不省时间。所以这类算子最终必须自己写 kernel（或者等厂商提供）——这也解释了为什么厂商算子库里最先补的总是量化矩阵乘，第 152 讲会看到海光支持的量化方法清单。

什么时候值得自己写

第三种办法最贵，所以要算一下划不划算。用第 83 讲的 Amdahl 定律：

$$\text{整体加速} = \frac{1}{(1-p) + p/s}$$

其中 p 是这个算子占一步的时间比例，s 是它自身的加速倍数。把清单里的算子代进去：

算子	占一步	写到两倍快	写到五倍快	结论
矩阵乘	62%	1.45×	1.9×	值得，而且是唯一真正值得的
注意力（bmm + softmax 融合掉）	12%	1.06×	1.11×	长上下文时占比会涨到几成，那时才值得（第 84 讲）
RMSNorm + RoPE 融合	8%	1.04×	1.07×	单独做不值得；但它能顺带减少几百次下发，在下发受限的卡上收益更大
写 KV Cache	5%	1.03×	1.04×	分页 KV 下必须自写（没有现成算子能按块表散写），但不是为了快

表里最后一行提醒一件事：**自写 kernel 的理由不总是性能**。有些算子是“没有替代品”才要写的——按块表散写 KV、按块表读 KV 做注意力（第 85 讲），这些在标准算子里根本没有对应物。

顺序上，推荐这样推进：

1. 盘点清单，把缺口分成“挡住跑通”和“只影响性能”两类。
2. 挡住跑通的，先用组合或回退补上（回退要记成技术债，写进待办）。
3. 跑对（第 139 讲），拿到正确的基线。
4. 用 profiler 量真实占比（第 142 讲），按占比决定写哪几个 kernel。
5. 每写完一个 kernel，重跑逐层比对——自写 kernel 是精度问题的主要来源。

还有一类缺口：模型结构

算子清单是按“标准稠密模型”数出来的。换一个模型，缺口会变样：

- **MoE 模型**（第 41 讲）多出路由、分组矩阵乘、All-to-All 三类需求，后者还和通信库绑在一起（第 146 讲）。

- **MLA 模型**（第 40 讲）要么用“正常还原”的写法（算子都是现成的，但显存和带宽都浪费），要么用吸收后的融合 kernel（快，但必须自己写）。第 152 讲会看到海光为 MLA 准备了多个后端，并按条件选。
- **混合架构**（第 42 讲的线性注意力、Mamba 类层）需要扫描类算子，这类算子在新平台上几乎总是缺的，而且不容易用现有算子高效组合。
- **多模态**多出图像预处理、视觉编码器里的卷积和插值算子。

所以“这块卡支持哪些模型”这个问题，比“支持 PyTorch 吗”具体得多，答案要按模型一个个列。海光的插件仓库就是这么组织的：`tests/models/` 下每个 checkpoint 一份配置，写明它的并行拓扑、注意力后端、MoE 后端和 KV Cache 设置（第 149 讲）。

回顾

算子缺口要用数据说，不能靠猜。在分发器那一层记账，Qwen3-0.6B 一步 decode 的 3932 次调用里 1711 次是不需要 kernel 的视图类，真正要算的是 23 种、2221 次——一个模型的最小算子集只有几十种。对照平台支持情况，缺口集中在量化矩阵乘、融合注意力和 MoE 分组算子三处。三种补法的实测对比给出了明确的取舍：回退 CPU 慢 15 倍且打断流水，只适合一次性路径；用现有算子组合能保住正确性（误差和原生实现完全相同），但保不住收益——没有原生 FP8 矩阵乘的话，FP8 只省显存不省时间；自己写 kernel 按占比决定，矩阵乘值得，占比几个百分点的算子不值得，除非它没有替代品。

第 18 篇的原理到这里讲完了。下一讲动手：把 miniinfer 真的搬到本机的 Apple MPS 上，走一遍设备抽象、算子盘点、精度比对和性能对比，看这些原理落到代码里是什么样子。

第141讲 实操 · 把 miniinfer 移植到非 CUDA

119 KV Cache　137 框架怎么接入新硬件　139 精度对齐　140 算子缺口：盘点与自写

前七讲的原理，这一讲全部落到代码上。目标是把第 15 篇写的 miniinfer 搬到一块非 NVIDIA 的加速器上，真实运行、真实比对、真实计时。

本机没有国产卡，但有一块能用的非 CUDA 加速器：Apple M2 的 GPU，PyTorch 通过 MPS 后端支持它。它和一块国产 GPGPU 的处境相似——不是 CUDA，算子库不全，低精度支持有缺口，profiler 工具另一套。所以这一讲的流程、踩的坑和验收标准都可以照搬；只有最后调 kernel 那一步，各家的工具不同。

起点选 **v1**（KV Cache 版，第 119 讲）：它只有五个文件、160 行，prefill 和 decode 两条路径都有，足够暴露移植的问题，又不会被调度、分页这些东西干扰。

第一步：把设备收到一个地方

v1 的代码里，设备这件事散在三处：`LLM.__init__` 的默认参数 `device="cpu"`、`load_weights` 的参数、`KVCache` 的参数。移植的第一件事不是改它们，而是**先建一个地方专管设备**，之后所有和“哪块卡”相关的决定都走这里。

port/device.py 共 31 行 ⇆ 对照

```python
"""设备抽象：引擎里所有和“哪块卡”有关的地方都走这里，换后端只改这一个文件（第 141 讲）。"""
import torch


def pick_device(name=None):
    # 没指定时按 CUDA（含 ROCm、海光 DCU 这类沿用 cuda 设备名的后端）→ MPS → CPU 的顺序挑
    if name:
        return torch.device(name)
    if torch.cuda.is_available():
        return torch.device("cuda")
```

1 **挑一块设备** 第 5-13 行

顺序是 CUDA → MPS → CPU。注意第一个分支：`torch.cuda.is_available()` 在一块海光 DCU 上也会返回真——那份 PyTorch 是用 HIP 编译的，对外仍自称 cuda（第 137 讲）。所以这一行代码同时覆盖了 NVIDIA 和走 ROCm 式接法的国产卡；走 PrivateUse1 式接法的硬件要在这里加一个分支（`torch.npu.is_available()` 一类）。

2 **同步** 第 16-21 行

3 **这块卡支持这种精度吗** 第 24-31 行

← 上一步 | 看整个文件 | 下一步 →

然后引擎只改四行：设备从参数里拿到后统一用 `self.device`，并在建模型之前先检查精度。

port/engine.py 对比 v1/engine.py：+11 行 / −6 行 ⇆ 对照

```
+ """移植版引擎：在 v1 的基础上，把设备从写死的 "cpu" 换成 device.py 挑出来的后端。"""
- """v1：带 KV Cache 的生成。prefill 一次算完整个提示，之后每步只算一个新 token。"""
  import torch
  from tokenizers import Tokenizer
  …… 省略 2 行没变的代码
  from model import Qwen3
  from kv_cache import KVCache
+ from device import pick_device, supports


  class LLM:
+     def __init__(self, model_dir, device=None, dtype=torch.float32):
+         self.device = pick_device(device)
+         if not supports(self.device, dtype):
+             raise ValueError(f"{self.device.type} 不支持 {dtype}")
-     def __init__(self, model_dir, device="cpu", dtype=torch.float32):
          self.cfg = Config.load(model_dir)
          self.tok = Tokenizer.from_file(f"{model_dir}/tokenizer.json")
+         self.model = Qwen3(self.cfg, load_weights(model_dir, dtype, self.device))
+         self.dtype = dtype
-         self.model = Qwen3(self.cfg, load_weights(model_dir, dtype, device))
-         self.device, self.dtype = device, dtype


      def generate(self, prompt, max_new_tokens=32):
  …… 省略 2 行没变的代码
          new = list(ids)                                  # 第一步（prefill）送入整个提示
          for _ in range(max_new_tokens):
+             start = len(ids) - len(new)
+             logits = self.model.forward(torch.tensor(new, device=self.device), star
+             nxt = int(logits[-1].argmax())               # 每步只在这里把一个数拷回 CPU
-             logits = self.model.forward(torch.tensor(new, device=self.device), len
-             nxt = int(logits[-1].argmax())               # 贪心：取打分最高的 token
              ids.append(nxt)
              new = [nxt]                                  # 之后（decode）每步只送入刚生成的
  …… 省略 11 行没变的代码
```

图 141-1　引擎的改动。device 的默认值从写死的 cpu 变成 None（由 device.py 决定），权重和 KV Cache 都用挑出来的设备；generate 里把 start 提成一个变量，为下一步做准备。

要点

移植的第一条经验：**先把“设备”这件事收敞口**。第 134 讲统计过，nano-vllm 里和设备、运行时绑定的写法有 23 行，散在一个文件里。如果不先收口，这 23 行会在后面每一次调试里反复出现。

第二步：跑通，和参考答案逐 token 比

改完就能跑。第 118 讲建立的检查脚本原样可用——它用同一组提示、贪心解码，和 HuggingFace 存下来的参考答案逐 token 比对：

```
$ .venv/bin/python tests/check.py port "{'device':'mps'}"
一致: 'The capital of France is' → ' Paris. The capital of Italy is Rome. The capital of Spain is Madrid. The capital of China is Beijing. The capital of Japan is Tokyo. The capital'
一致: 'def fibonacci(n):' → '\n    if n == 0:\n        return 0\n    elif n == 1:\n return 1\n    else:\n        return fibonacci(n'
一致: '一个推理引擎由哪几部分组成？' → '它如何运作？  请提供一个例子，说明如何使用它来解决一个实际问题。\n答案:  问题：某公司需要为员工'
一致: '1, 2, 3, 4, 5,' → ' 6, 7, 8, 9, 10, 11, 12, 13, 14,'
port: 4/4 条与 HuggingFace 逐 token 一致
```

图 141-2　移植版在 Apple MPS 上的正确性检查（FP32，本机真实运行）。

四条提示、每条 32 个 token，全部与 HuggingFace 逐 token 一致。这个结果说明三件事：MPS 上这 23 种算子的实现都是对的；FP32 下的数值差异（1e-6 量级，第 139 讲量过）不足以改变贪心选择；KV Cache 的写入和读取在新后端上没有走偏。

> **注意**
> 能跑通 FP32 不等于能跑通别的精度。下面第四步就会看到 FP16 的翻车。移植的检查要按“每个精度各跑一遍”来做，而不是跑通一次就宣布完成。

第三步：算子盘点，找出不支持的算子

跑通之后先别调性能，而是把这个模型用到的算子完整数一遍，确认没有东西在悄悄回退到 CPU。办法是第 140 讲的 `TorchDispatchMode`：

```
$ .venv/bin/python bench/port_ops.py                                    ⇆ 对照
一、Qwen3-0.6B 在 MPS 上用到 31 种 aten 算子；一步 decode 调用 3932 次，其中视图类 1711 次，要在设备上算的 2221 次（23 种）
   要算的：mul×590、add×225、mm×197、pow×169、mean×113、rsqrt×113、arange×85、div×84、reciprocal×56、_to_copy×56、cos×56、sin×56、sub×56、cat×56、index_put_×56、clone×56、bmm×56、le×28、bitwise_not×28、masked_fill×28、_softmax×28、silu×28、index×1
   视图类：permute×477、view×448、unsqueeze×448、slice×169、select×112、expand×56、lift_fresh×1
   回退到 CPU 的算子：0 个
二、推理引擎常用的几个算子在 MPS 上（torch 2.15.0）：
   支持     bf16 矩阵乘
   支持     FP8 类型转换
   不支持   FP8 矩阵乘 _scaled_mm：NotImplementedError
   支持     INT8 矩阵乘 _int_mm
   支持     MoE 分组矩阵乘 _grouped_mm
   支持     注意力 scaled_dot_product_attention
   不支持   FlashAttention 内核 _scaled_dot_product_flash_attention：NotImplementedError
   不支持   FP64：TypeError
三、补上 FP8 矩阵乘（M=16, K=N=4096，模拟 decode 时的一个线性层）：
     22273 μs    相对误差 3.78e-02    回退到 CPU
      1459 μs    相对误差 3.78e-02    组合：反量化 + BF16 矩阵乘
       526 μs    相对误差 2.88e-03    对照：不量化，BF16 矩阵乘
    （相对误差 = ‖结果 - FP32 精确值‖ ÷ ‖FP32 精确值‖；前两种的误差来自 FP8 量化本身，两者应当接近）
```

图 141-3　算子清单、MPS 的支持情况、以及补上缺失算子的两种办法（本机真实运行）。

对这次移植，三个结论：

31 种算子，回退到 CPU 的 0 个。这条最重要——它意味着后面量到的性能数字是真的在设备上跑出来的。如果这里不是 0，所有性能结论都要作废。

一步 decode 要在设备上算 2221 次。这个数字解释了第 138 讲的下发开销：两千多次调用，每次都要走分发、算形状、下发。

MPS 缺 FP8 矩阵乘和 FlashAttention 内核。对 miniinfer 的当前版本不影响（它用 `bmm` + `softmax` 手写注意力，用 FP32/FP16 权重），但如果要把第 123 讲的量化版或第 84 讲的 FlashAttention 搬过来，这就是缺口。

第四步：精度，一个真实的坑

FP32 跑通了，换 FP16 试试——这是移植里必然要走的一步，因为 FP32 在任何加速器上都是浪费（权重大一倍、带宽多读一倍）。

结果是第 139 讲开头那段输出：生成一串句号和换行。逐层比对直接给出了位置：

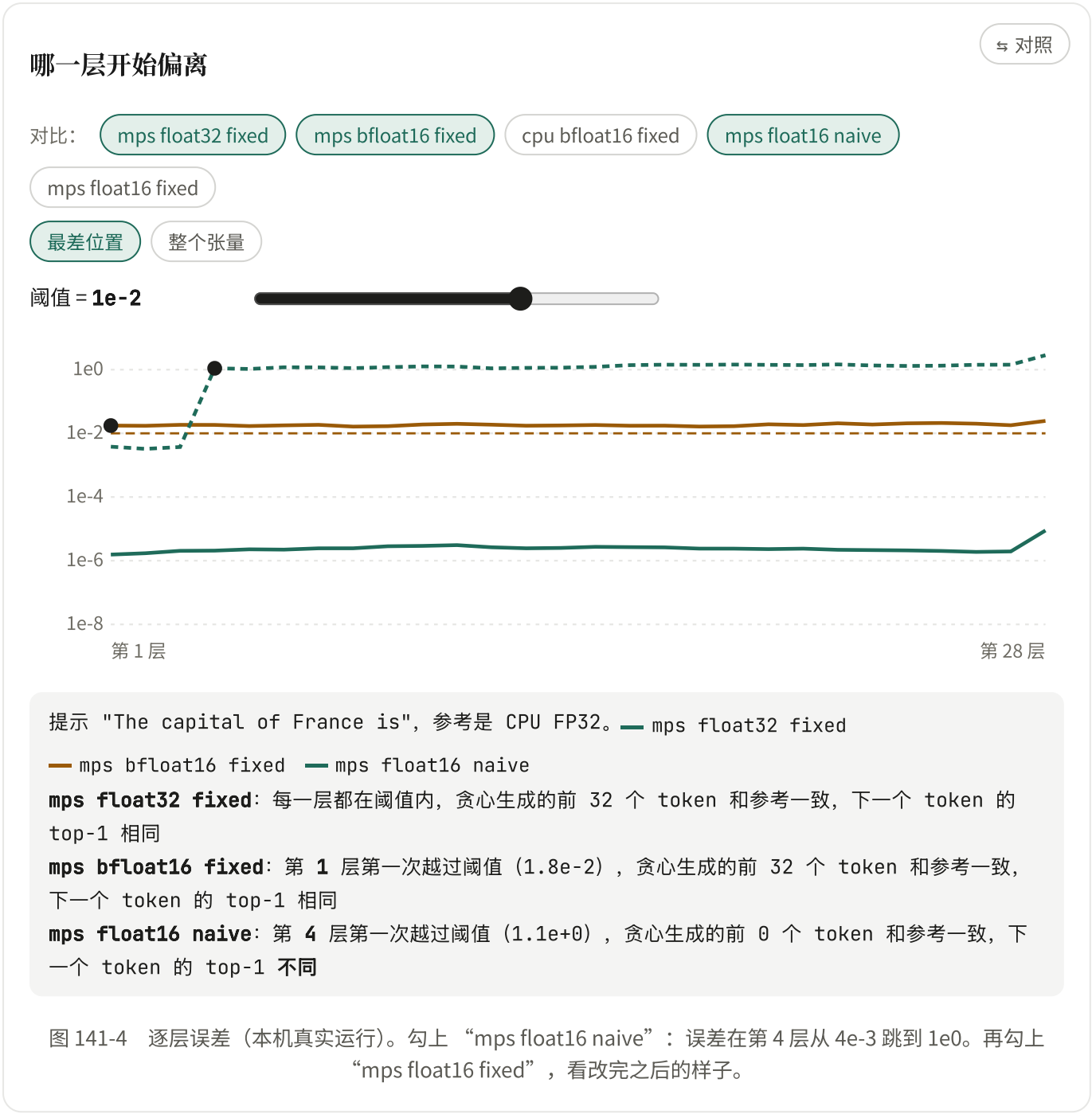


图 141-4　逐层误差（本机真实运行）。勾上 “mps float16 naive”：误差在第 4 层从 4e-3 跳到 1e0。再勾上 “mps float16 fixed”，看改完之后的样子。

原因在第 139 讲讲透了：RMSNorm 的平方和在 FP16 里溢出（激活值 6544 的平方是 4.28×10^7，超过 FP16 的上限 65504）。改法是把平方和升到 FP32：

port/model.py 对比 v1/model.py：+16 行 / −8 行 ⇆ 对照

```
+ """移植版的 Qwen3 前向：和 v1 一样只用 torch 的基本运算；改了两处，让它在 GPU 类的后端上不做
- """Qwen3 的前向计算：只用 torch 的基本运算，一层一层写出来。"""
  import torch

  ····· 省略 1 行没变的代码
  def rms_norm(x, weight, eps):
      # 每个向量除以自己的均方根，再乘可学习的缩放（第 36 讲）
+     # 平方和先升到 FP32 再算：FP16 最大只能表示 65504，而 Qwen3 第 0 个位置的激活能到 6500
+     # 平方就是 4000 多万，在 FP16 里直接变成 inf（第 139 讲）。FP32 下这两行和 v1 完全一样
+     xf = x.float()
+     var = xf.pow(2).mean(-1, keepdim=True)
+     return (xf * torch.rsqrt(var + eps)).to(x.dtype) * weight
-     var = x.pow(2).mean(-1, keepdim=True)
-     return x * torch.rsqrt(var + eps) * weight


  ····· 省略 13 行没变的代码
          self.cfg, self.w = cfg, w

+     def attention(self, i, x, pos, end, cache):
-     def attention(self, i, x, pos, cache):
          c, w, p = self.cfg, self.w, f"model.layers.{i}.self_attn."
          T = x.shape[0]
  ····· 省略 5 行没变的代码
          q, k = rope(q, pos, c.rope_theta), rope(k, pos, c.rope_theta)
          cache.write(i, pos, k, v)                                    # 新 token 的 K、V
+         # end 由调用方用 Python 整数传进来。v1 在这里写的是 int(pos[-1]) + 1：
+         # 在 GPU 上这一步要等设备把这个数算完再拷回主机，28 层就停 28 次（第 141 讲量到一次约
-         end = int(pos[-1]) + 1
          k, v = cache.read(i, end)                                    # 取出从第 0 个位置到
          rep = c.num_heads // c.num_kv_heads                          # GQA：每组 K、V 给
  ····· 省略 11 行没变的代码

      @torch.no_grad()
+     def forward(self, tokens, start, cache, trace=None):
-     def forward(self, tokens, start, cache):
          # tokens: 这次新来的 [T] 个 token，从位置 start 开始；它们之前的 K、V 已经在 cach
+         # trace: 传一个列表进来，就把每一层的输出依次存进去（精度对齐用，第 139 讲）
          c, w = self.cfg, self.w
+         end = start + len(tokens)
+         pos = torch.arange(start, end, device=tokens.device)
-         pos = torch.arange(start, start + len(tokens), device=tokens.device)
          x = w["model.embed_tokens.weight"][tokens]
          for i in range(c.num_layers):
              p = f"model.layers.{i}."
+             x = x + self.attention(i, rms_norm(x, w[p + "input_layernorm.weight"],
-             x = x + self.attention(i, rms_norm(x, w[p + "input_layernorm.weight"],
              x = x + self.mlp(i, rms_norm(x, w[p + "post_attention_layernorm.weight"
```

```
67 +             if trace is not None:
68 +                 trace.append(x)
69           x = rms_norm(x[-1:], w["model.norm.weight"], c.rms_eps)    # 只需要最后一个位置
70           return x @ w["lm_head.weight"].T
```

图 141-5　model.py 的三处改动：RMSNorm 的平方和升到 FP32（这一步修掉了 FP16 的溢出）；end 改成由调用方传入（下一步会看到它值多少钱）；forward 多一个 trace 参数，用来逐层比对。

改完之后 FP16 和 BF16 都和参考逐 token 一致，FP16 的逐层误差（3e-3）还比 BF16（2e-2）更小。

第五步：性能，以及一行代码值多少

现在量速度。同一个 199 token 的提示（和第 119 讲同一条），生成 64 个 token，分别计时 prefill 和每步 decode：

```
$ .venv/bin/python bench/port_speed.py                                    ⇆ 对照

decode 一步要读的权重: 596 M 个参数, FP32 2.38 GB, FP16/BF16 1.19 GB
版本     设备    精度          prefill(199)   decode 每步    64 步合计    生成的前 8 个 token
和 CPU FP32 一致吗
v1     cpu   float32          355 ms     64.9 ms    4.44 s    一致
v1     mps   float32          170 ms     55.5 ms    3.67 s    一致
port   mps   float32          141 ms     49.3 ms    3.25 s    一致
port   mps   float16          130 ms     32.9 ms    2.20 s    一致
port   mps   bfloat16         196 ms     33.1 ms    2.28 s    一致
port   cpu   bfloat16        3683 ms     50.6 ms    6.87 s    一致
MPS 上下发一个很小的 add: 平均 5.2 μs
MPS 上int(张量): 等设备算完再拷回一个数: 平均 139.9 μs
```

图 141-6　CPU 与 MPS、三种精度下的 prefill 与 decode 耗时（本机 Apple M2，真实运行；每次计时前后都做了设备同步）。

逐行读这张表。

CPU FP32 → MPS FP32：prefill 355 ms → 170 ms，decode 每步 65 ms → 56 ms。prefill 快了一倍（它受算力限制，GPU 的算力更高），decode 只快 14%。decode 为什么没跟上？因为它受带宽限制（第 68 讲），而 M2 的 CPU 和 GPU **共用同一块内存**——统一内存架构下，换到 GPU 并不能换来更高的带宽。这个现象在独立显卡上不会出现，但它恰好演示了一件事：**decode 的速度由带宽决定，不由谁在算决定**。

v1 原样搬到 MPS → port 版：decode 每步 56 ms → 49 ms，快了 13%。两个版本的数学完全一样，差别只有 `model.py` 里那一行：v1 写 `end = int(pos[-1]) + 1`，port 版把 `end` 作为参数传进来。表最后两行解释了这 7 ms：MPS 上下发一个很小的加法平均 5.2 µs，而把一个数从设备拷回 CPU（`int(tensor)`）平均 139.9 µs——相差 27 倍。28 层每层一次，就是 28 × 140 µs ≈ 3.9 ms；再加上它每次都打断设备上正在排队的 kernel 流水，实际代价更大。

> **要点**
> 这是移植性能问题的典型形状：**不是哪个 kernel 慢，而是 CPU 和设备之间的同步太频繁**。这类问题改起来几乎没有成本（把一个数从外面传进来），收益却是百分之十几。调优时先找这类同步点（`int(tensor)`、`.item()`、`tensor.cpu()`、`if tensor > 0`、打印张量），再去看 kernel，顺序和第 142 讲一致。

FP32 → FP16：decode 每步 49.3 ms → 32.9 ms，快 1.5 倍。理论上应该快 2 倍：decode 一步要读 596 M 个参数的权重（脚本第一行算出来的），FP32 是 2.38 GB，FP16 是 1.19 GB，字节减半。实际只有 1.5 倍，差的那部分是下发开销和没跑满的带宽——按这两个数算，FP32 下达到的有效带宽是 2.38 GB ÷ 49.3 ms ≈ 48 GB/s，FP16 是 1.19 GB ÷ 32.9 ms ≈ 36 GB/s。精度越低，要读的字节越少，固定开销的占比就越高，所以效率反而下降。这是第 67 讲“实测与理论的差距”在一块新硬件上的具体体现。

BF16 的 prefill 比 FP16 慢（196 ms 对 128 ms），decode 却一样快（33 ms）。prefill 受算力限制，这个差别说明 MPS 上 BF16 的矩阵乘走的路径不如 FP16 高效；decode 受带宽限制，两者字节数相同，所以没差别。**同样是 16 位，在同一块卡上也可能不等价**——选精度时要实测，不能只看位数。

CPU BF16 的 prefill 是 3665 ms，比 CPU FP32 慢 10 倍。这是全表最反直觉的一行。原因是本机 CPU 没有原生的 BF16 矩阵乘指令，PyTorch 只能把 BF16 转成 FP32 再算，每次矩阵乘都多两次类型转换。**低精度在没有原生支持的硬件上会变慢，而不是变快**——第 145 讲会把这件事展开讲，它也是适配国产卡时最常见的误判之一。

还差什么

到这里，一个能跑、跑对、知道瓶颈在哪的移植版完成了。离“能上线”还差这些，每一项对应后面的讲：

还缺什么	为什么现在没做	对应的讲
分页 KV、连续批处理、调度	这些是引擎层的逻辑，和硬件无关，v3 已经有了，直接套上来即可	121、122
融合算子（RMSNorm、RoPE、SwiGLU）	现在每个都是几个基础算子拼出来的，一步多上千次下发	83、144
FlashAttention 类的注意力	MPS 缺这个内核；长上下文下现在的写法会让注意力矩阵落地	84、143
图模式 / 图捕获	去掉同步点之后整个前向已经能编成一张图（第 138 讲量过：3319 个节点、0 处中断），但还没接后端	138
量化	MPS 缺 FP8 矩阵乘；INT8 有 `_int_mm`，可以试	145
多卡	本机只有一块 GPU，通信这一层完全没碰	146
profiler 下的逐 kernel 分析	下一讲的主题	142

在一块真实的国产卡上，这张表还要多两行：**厂商的算子库接进来**（矩阵乘换成厂商的库，注意力换成厂商移植的 FlashAttention），以及**引擎插件**（把这些选择告诉 vLLM 这样的引擎，而不是只在自己的小引擎里用，第 147 讲）。

专栏

这次移植的全部改动：新增 `port/device.py`（31 行），`engine.py` 改 4 处，`model.py` 改 3 处。总共不到 50 行——和第 134 讲那个“4.4%”的统计是一致的。但要注意这个数字的前提：miniinfer 本来就只用基础算子，没有 CUDA 专有库、没有自写 kernel、没有通信。真实引擎的那 4.4% 里，每一行背后都是一个库。

回顾

一次完整的移植，顺序是：把设备收进一个模块（顺手把同步和精度探测也包进去）→ 跑通并和参考答案逐 token 比对 → 盘点算子确认没有东西回退到 CPU → 逐个精度跑逐层比对（FP16 在 RMSNorm 上溢出，升到 FP32 解决）→ 量性能并先解决同步点。本机的实测给出了几个可以带走的结论：decode 的速度由带宽决定，统一内存下换到 GPU 也快不了多少；一行 `int(pos[-1])` 值 13% 的 decode 吞吐；FP16 只快 1.5 倍而不是 2 倍，因为固定开销占比上升；CPU 上的 BF16 比 FP32 慢 10 倍，因为没有原生支持。

第 18 篇到此结束——模型跑通了、跑对了。第 19 篇开始讲怎么跑快。第一个问题是：在一块没有 Nsight 的卡上，怎么知道时间花在哪里？

AI INFRA · HANDBOOK

第 19 篇

调优原理：把模型跑快

收录 6 讲 · 第 142—147 讲

第142讲 在新硬件上做性能分析

66 性能分析　67 理论 vs 实测　141 实操 · 把 miniinfer 移植到非 CUDA

上一讲的移植版在 MPS 上跑到了每步 33 ms。这个数字是快还是慢？第 141 讲已经算过一次下限：一步要读 1.19 GB 的权重，按机器的内存带宽，下限远低于 33 ms。差的部分在哪儿，得量。

第 66 讲讲过在 NVIDIA 卡上怎么读 Nsight 的时间线。换一块卡，工具名字和界面全变了，但要回答的问题没变。这一讲给出一套和工具无关的做法：时间线上该看哪几件事；按什么顺序找瓶颈；以及在本机用 PyTorch 的 profiler 真实量一遍，看它能告诉我们什么、不能告诉我们什么。

时间线上看三样东西

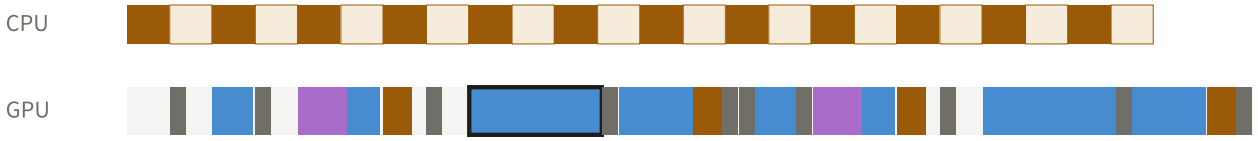
厂商 profiler 的输出形式各不相同（有的是时间线，有的只有汇总表），但只要能拿到“哪个 kernel、什么时候开始、持续多久”，就能看出三件事。

一步 decode 的时间线

⇆ 对照

看前 2 层　看整步

eager　CUDA Graph



蓝：矩阵乘法　紫：注意力　灰：逐元素　琥珀：通信　空白：GPU 在等

0 – 0.21 ms

选中：第 0 层 **gate_up_proj**（矩阵乘法），24.9 µs，读写 58.72 MB，实际带宽 2.36 TB/s（峰值 3.35）。点时间线上的 kernel 换一个看。
整步：3.09 ms，GPU 忙 91%，空等 0.27 ms；矩阵乘法 1.69 ms，逐元素 0.49 ms，通信 0.35 ms，注意力 0.30 ms。
小 kernel 之间的空白：GPU 算完了，下一个 kernel 还没被 CPU 发射过来（第 89 讲）。

图 142-1　一步 decode 的时间线（模拟数据，模型同第 66 讲：Llama-3-8B、TP=4）。两行：上面是 CPU 在发射 kernel，下面是设备在执行。点一个 kernel 看它的耗时和实际带宽；切换 eager / 图捕获看空隙的变化。

第一样：两行之间的关系。CPU 这一行是发射，设备那一行是执行。设备那一行**有空隙**，说明它在等 CPU——瓶颈在下发，不在硬件。设备那一行**排满**，说明硬件在满负荷干活，这时候才该去看单个 kernel 快不快。

第二样：最长的那几个 kernel。按耗时排序，看前五个占了多少。decode 一步里排在前面的应该是几个大矩阵乘；如果排在前面的是类型转换、拷贝、转置这类算子，说明有东西写得不对。

第三样：空隙的位置。规律的、每层一次的空隙，通常是同步点（把一个数拷回主机、等一次事件）。一次很长的空隙，通常是主机侧在做慢操作（Python 里的循环、重新分配显存、JIT 编译一个 kernel）。

注意

换到一块新卡时，profiler 本身就是要适配的东西之一（第 134 讲的第九层）。常见的坑：时间线里只有 CPU 侧的时间、没有设备侧；kernel 名字是内部编号，对不上源码里的算子；时间戳的两个时钟没对齐，CPU 和设备的时间线错位。所以 profiler 的数字要和一个你自己写的、只用计时器的粗测互相印证——下面就是这么做的。

按什么顺序找

有了数据，接下来是顺序问题。四类瓶颈——下发开销、访存、计算、通信——该先看哪个？答案取决于负载：

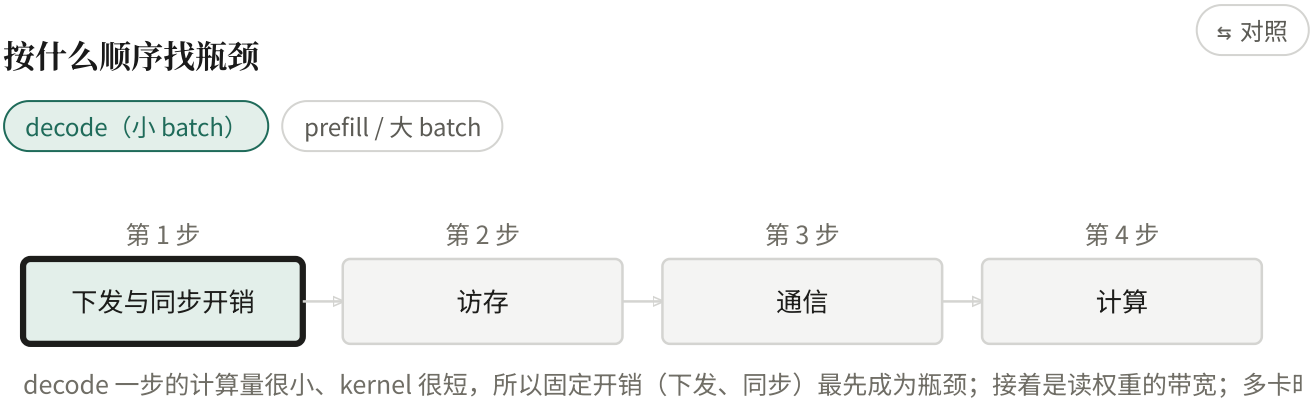


图 142-2　decode 和 prefill 两种负载下找瓶颈的推荐顺序。点一类看它的症状、确认办法和常见改法；切换负载看顺序怎么变。

decode 先看下发开销，理由是性价比：去掉一个同步点、开一次图捕获，改动小、收益大，而且这类问题在新硬件上普遍存在（第 141 讲那一行 `int(pos[-1])` 就是 13%）。prefill 反过来：kernel 本身就很长，下发可以忽略，先看算力有没有用上。

两种负载在真实服务里是混在一起的（第 77 讲的分块 prefill），所以要分开量：一次只跑纯 decode，一次只跑长 prefill。混在一起量出来的平均值，既不能解释 TTFT 也不能解释 TPOT。

在本机真的量一遍

下面这份脚本在 MPS 上做三件事：用 profiler 按算子汇总，用显式同步按段汇总，再量两个固定开销。

```
$ .venv/bin/python bench/port_profile.py                                  ⇆ 对照

一、按算子汇总（4 步平均，上下文 199 token，FP16，合计 25.4 ms）
     毫秒      占比      次数  算子
  16.06  63.3%    844  aten::copy_
   3.62  14.3%     56  aten::bmm
   0.95   3.8%    590  aten::mul
   0.39   1.5%    225  aten::add
   0.37   1.4%    197  aten::mm
   0.33   1.3%    225  aten::pow
   0.32   1.3%    760  aten::_to_copy
   0.26   1.0%    477  aten::permute
   0.24   0.9%     56  aten::einsum
   0.20   0.8%    113  aten::mean
   2.61  10.3%         其余 36 类合计

二、按段汇总（每段边界都做了同步，所以一步 61.0 ms 比不加同步时长）
  28.97 ms  47.5%  注意力（QKV 投影、两次 RMSNorm、RoPE、写读 KV、打分、输出投影）
  13.19 ms  21.6%  FFN（SwiGLU 的三个矩阵乘）
  18.79 ms  30.8%  其余（层间的两次 RMSNorm、残差、输出层、Python 侧的循环）

三、固定开销和一处写法差异
  下发一个很小的 add：平均 3.6 μs
  int(张量)：等设备算完再拷回一个数：平均 158.1 μs
  RMSNorm：整个张量升 FP32：平均 37.0 μs
  RMSNorm：只有归约用 FP32：平均 25.9 μs
  两种写法的最大差值 3.551，输入量级 342，相对差约 1.0%（FP16 的舍入）
  一步 decode 调用 RMSNorm 113 次：按上面的差值，一步省下约 1.3 ms（约 4% 的一步时间）
```

图 142-3　Qwen3-0.6B 一步 decode 在 Apple M2 的 GPU 上的分析（FP16，上下文 199 token，本机真实运行）。

这份输出里有三个值得琢磨的地方。

第一段：`aten::copy_` 占了 63%。按算子汇总的第一名不是矩阵乘，而是拷贝，844 次。这就是上一节说的“排在前面的是拷贝类算子，说明有东西写得不对”的真实样本。追下去，这些拷贝来自两处：RMSNorm 里为了防溢出做的 `x.float()` 和 `.to(dtype)`（第 139 讲），以及 RoPE 里的类型转换。113 次 RMSNorm、28 次 RoPE，每次两三个拷贝，就是八百多次。

第二段：注意力 43%、FFN 23%、其余 34%。这个比例和第一段的算子汇总不矛盾，但角度不同：按算子看，时间散在拷贝里；按段看，时间集中在注意力。两种视角要一起用——按算子能发现“哪类操作有问题”，按段能回答“改哪一部分最值”。

要点

在异步后端上，**不显式同步的分段计时是错的**。设备上的 kernel 是排队执行的，CPU 侧的某一次调用会在队列积压时被阻塞，于是这段时间被记到“恰好等到的那次调用”头上，而不是真正慢的那次。这一段的代码在每段边界都调了 `synchronize()`：代价是一步从 33 ms 涨到 54 ms（同步本身要钱），好处是比例可信。第 66 讲在 CUDA 上强调的也是同一件事。

第三段：两个固定开销和一个写法对比。下发一个小 kernel 3.6 µs，把一个数拷回主机 158 µs，相差 44 倍——这个比例在不同硬件上不一样，但量级关系是普遍的，所以“热路径上不要把数据拷回主机”是条通用规则。

最后两行是一次具体的优化判断。RMSNorm 的两种写法：把整个张量升到 FP32（现在的写法，37.0 µs），或者只让归约在 FP32 里做（25.9 µs）。后者快 11 µs，一步调用 113 次，省 1.3 ms，约占一步的 4%；两者数值差 1%，是 FP16 的舍入，在第 139 讲的阈值内。

4% 值不值得改？这是个取舍，不是定论：第 83 讲的 Amdahl 定律说它只能带来 1.04 倍；而它要改的是一处和数值安全直接相关的代码（溢出就是在这里修掉的）。我们这里选择不改，把它记在待办里，等到真正受带宽限制、而且有了融合的 RMSNorm 算子时一起处理。**把判断过程写下来，比改或不改更重要**——第 153 讲的清单里有一项就是“每个被放弃的优化都要记下理由”。

判断单个 kernel 够不够好

定位到某个 kernel 之后，还要回答“它还有多少空间”。不管用哪家的工具，判据只有两条，而且都只需要三个数：这个 kernel 的耗时、它读写了多少字节、它做了多少次乘加。

$$\text{实际带宽} = \frac{\text{读写字节}}{\text{耗时}}, \qquad \text{实际算力} = \frac{2MNK}{\text{耗时}}$$

把两者分别和官方的带宽、峰值算力比，就知道这个 kernel 卡在哪一边、离上限多远（第 62、67 讲）。几条经验阈值：受带宽限制的 kernel 做到标称带宽的 80% 以上就接近极限了，60% 以下说明访存模式有问题（不连续、跨步、没对齐）；受算力限制的矩阵乘做到峰值的 70% 以上算好，30% 以下通常是分块或对齐的问题（第 143 讲）。

厂商工具里的字段名各不相同，但大多能对应到这几件事：

要回答的问题	工具里常见的字段	看出什么
访存效率	内存吞吐、L2 命中率、访存事务数	实际带宽占标称的几成；事务数远多于理论值说明访存不连续
占用率	每个计算单元驻留的线程组数、寄存器用量、片上存储用量	哪项资源先用完（第 86 讲的 `TileOccupancy`）
指令构成	矩阵指令占比、标量/向量指令占比	矩阵核心有没有被喂满；标量指令过多说明地址计算或控制流太重
等待原因	停顿分类（等访存、等同步、等依赖）	哪一类等待最多，决定改访存还是改流水

有一点要留意：**占用率高不等于快**。一个把寄存器用得很省、驻留了很多线程组的 kernel，可能因为每个线程手里的累加器太少而反复读写片上存储。第 86、87 讲算过这笔账，第 143 讲会在矩阵单元的形状上再碰到它。

没有好 profiler 的时候

新硬件上常遇到的情况是工具还不好用。有四种办法，都只依赖计时器：

分段计时加同步。就是上面第二段的做法：在你关心的边界前后同步再计时。能回答“哪一段占时间”，代价是同步本身的开销会让总时间变长。

删一段再量（差分法）。把某一段跳过去（比如把 FFN 换成一个形状相同的空操作），量总时间的变化。它的好处是不需要工具支持；坏处是数学变了，只能用于定位，不能用于验证。

改一个参数看斜率。这是最可靠的归因办法，因为它不依赖任何内部可见性。把权重精度从 FP32 换成 FP16：如果时间几乎减半，就是受访存限制；如果几乎不变，就是受下发或算力限制。把上下文从 200 加到 2000：如果每步时间按比例涨，注意力就是主项；如果几乎不变，就是权重读取占主导。第 141 讲表里的几行其实都是这种实验。

数 kernel 和算理论值。一步有多少次算子调用（第 140 讲的办法），乘上单次下发的开销，就得到下发开销的下限；要读多少字节除以官方带宽，得到访存的下限。两个下限哪个更接近实测，瓶颈就在哪边。

专栏

第 67 讲提过“实测和理论的差距”这个习惯：每量到一个数，先算出它的理论下限，再看差多少倍。在新硬件上这个习惯更重要，因为你对这块卡没有直觉。本机的例子：FP16 下一步 32.9 ms、要读 1.19 GB，有效带宽 36 GB/s；换成 FP32 是 2.38 GB、49.3 ms，有效带宽 48 GB/s。两个数都

远低于这台机器的内存带宽，而且**精度越低效率越差**——这就把“固定开销占比上升”这件事量出来了。

调优的记录该怎么写

性能工作最容易丢的不是数字，而是上下文。一条记录至少要有这些字段，缺了任何一项，几周后就复现不出来：

字段	例子
硬件与软件版本	Apple M2，PyTorch 2.15，MPS 后端
模型与精度	Qwen3-0.6B，FP16 权重
负载	上下文 199 token，batch 1，纯 decode，64 步
测的是什么	每步 forward 的时间，前后都同步，预热 4 步
结果	32.9 ms/步
理论下限与差距	1.19 GB ÷ 带宽；实测有效带宽 36 GB/s
这次改了什么	`end` 由参数传入，去掉每层一次的主机同步
相比上一次	49.3 → 32.9 ms（含换精度），其中去同步贡献约 13%

海光的插件仓库里把这件事做成了公开的 `docs/validation/` 目录：每个配置一份记录，写明冻结的环境、硬件、后端、并行拓扑和各数据集的分数（第 139、149 讲）。做适配时抄这个格式就够用了。

回顾

在一块陌生的卡上做性能分析，第一步是从时间线上看三样东西：CPU 和设备两行的关系（有空隙就是下发瓶颈）、最长的几个 kernel（如果是拷贝类算子就说明写法有问题）、空隙的位置（规律的空隙是同步点）。找瓶颈的顺序按负载定：decode 先看下发和同步，再看访存，最后才是算力；prefill 反过来。本机的实测演示了这套做法的结果：按算子汇总第一名是 844 次拷贝（占 63%，来自防溢出的类型转换），按段汇总注意力占 43%；异步后端上分段计时必须显式同步，否则时间会被记到错误的调用上；一次主机同步 158 μs，是一次小 kernel 下发的 44 倍。工具不好用时，靠分段计时、差分、改一个参数看斜率、数 kernel 算下限这四招也能定位。

瓶颈定位完，接下来就是动手改最热的那部分。下一讲处理推理里最热的两个算子：矩阵乘和注意力。为什么厂商库给的分块在 decode 时反而最糟？

第143讲 矩阵乘法与注意力的适配

重点

84 FlashAttention　85 Decode 注意力 Kernel　87 矩阵乘法 Kernel 与 Tensor Core　135 国产与 NVIDIA 硬件差异

上一讲把瓶颈定位到了最热的那几个算子。推理里最热的永远是两个：矩阵乘法（权重投影，一步要把全部权重读一遍）和注意力。这一讲讲它们在一块新卡上为什么需要重做，以及怎么重做。

先把结论摆在前面，它很反直觉：**厂商的矩阵乘库在 decode 时往往是最糟的选择**。不是因为库写得差，而是因为库默认的那套分块是为训练和 prefill 那种大矩阵调的，decode 的形状（M 只有几十）恰好落在它最不擅长的区间。下面用海光插件源码注释里给出的真实数字算这笔账。

矩阵单元的三个硬约束

第 135 讲说过矩阵核心按固定形状吃数据。这件事对 kernel 提出三个要求。

指令形状与对齐。一条指令算 im × in 的输出、沿 K 走 ik 步。输出块的行数必须补齐到 im 的倍数、列数补齐到 in 的倍数，K 方向也要对齐。海光那个小 M kernel 的约束条件写得很直白：`M 在 [1, 64]`、`N 只支持 256 / 512 / 1024 / 2048`、`K 必须是 128 的倍数`（`vllm_hcu/utils/bf16_fp32_mmac.py` 的 `supported()`，提交 81cbe8f）。这不是偷懒，而是把对齐要求显式写成了入口检查：不满足就走别的实现，而不是悄悄算错。

数据排布。矩阵核心要求参与运算的数据在寄存器或片上存储里按特定方式摆放（哪条通道拿哪几个元素）。所以 kernel 里常见两件事：把权重**预先转置**成 kernel 想要的方向，以及按 16 字节对齐做向量化加载。海光的代码里两种布局都有：一个环境变量控制权重按 `[in_features, out_features]` 还是 `[out_features, in_features]` 存，入口处先判断布局再决定走哪条路（`vllm_hcu/utils/bf16_fp32_gemm.py`）。注意力的 KV Cache 也有同样的问题，插件的文档里要求用 FlashAttention 后端时设 `VLLM_KV_CACHE_LAYOUT=HND`，并提醒混合注意力的模型不要全局强制这个布局。

每条通道持有多少累加器。输出块的元素分摊在线程组的通道里，通道数不同，每条通道手里的累加器个数就不同（第 135 讲的 16×16 指令：32 条通道每条 8 个，64 条每条 4 个）。这个数字直接决定寄存器压力和一个计算单元能驻留几个块。

⇆ 对照

线程组宽度和片上缓冲怎么限制分块

线程组宽度 w： 32（warp） 64（波前）

矩阵指令： 16×8×16 16×16×16 32×32×16

M（这一步的 token 数）= **16**

输出块 Mt × Nt = **32 × 128**

K 步长 = **64**

片上存储 = **160** KB

输出块 32 × 128，每格是一条 16×8 的矩阵指令（深色 = 这一步真的有 token，浅色 = 补零）



一条指令 16×8×16
一个线程组 32 条通道
每通道 4 个累加器（FP32）
一块要 32 条指令覆盖
一块的 A+B 分块 20 KB

M = 16，输出块 32 行：这一步**只有 16 行是真的**，块里 50% 的行在算补的零（M = 16 对指令行数 16 来说正好对齐）。
每条通道持有 4 个 FP32 累加器：指令形状不变、通道数从 32 变 64，每通道的累加器就少一半，寄存器压力减半，但要凑满一条指令需要的数据翻倍。
片上存储 160 KB ÷ 一块 20 KB：单缓冲能放 **8** 块，双缓冲（一边搬一边算）能放 **4** 块。

图 143-1　同一块输出在不同指令形状、不同线程组宽度下的账。把 M 拖到 16，输出块拖到 256 × 256，看有多少行在算补的零；再把输出块调成 16 × 64。

decode 的小 M：库为什么会选错

现在算那笔账。场景取海光插件注释里的例子：一个 lm_head 分片，权重 32224 × 4096（BF16）。decode 时 M ⩽ 64。

```python
# 海光插件源码注释里的一组数（vllm_hcu/model_executor/layers/skinny_lm_head.py 开头，提交 81cbe8f）：
# lm_head 分片 32224 × 4096 的 BF16 权重，M ≤ 64 属于“纯读权重”的形状；
# hipBLASLt 在这里选 256×256 的宏块，M=16 时 285 μs；只沿 N 分块的专用 kernel 153 μs，M=64 时 190 μs。
N, K = 32224, 4096
bw_w = N * K * 2                                  # BF16 权重的字节数：decode 时要读的主要数据
print(f"权重 {N} × {K} BF16 = {bw_w / 1e6:.0f} MB（decode 时每步都要整读一遍）")
for name, m, us in [("库选的 256×256 宏块", 16, 285), ("只沿 N 分块的专用 kernel", 16, 153), ("只沿 N 分块的专用 kernel", 64, 190)]:
    print(f"  {name}，M = {m:2d}：{us} μs → 实际带宽 {bw_w / (us * 1e-6) / 1e12:.2f} TB/s，"
          f"实际算力 {2 * m * N * K / (us * 1e-6) / 1e12:.1f} TFLOPS")

print("\n同一个 M = 16 的输出，两种分块的账")
for mt, nt in [(256, 256), (16, 64)]:
    rows = -(-16 // mt) * mt                      # 行方向补齐到块的倍数
    blocks = -(-16 // mt) * -(-N // nt)
    print(f"  输出块 {mt:3d} × {nt:3d}：补到 {rows:3d} 行（有效 16，浪费 {1 - 16 / rows:4.0%}），"
          f"{blocks:3d} 块，矩阵指令算的格子 {blocks * mt * nt / 1e6:.1f} M（有用的只有 {16 * N / 1e6:.1f} M）")
```

```
权重 32224 × 4096 BF16 = 264 MB（decode 时每步都要整读一遍）
  库选的 256×256 宏块，M = 16：285 μs → 实际带宽 0.93 TB/s，实际算力 14.8 TFLOPS
  只沿 N 分块的专用 kernel，M = 16：153 μs → 实际带宽 1.73 TB/s，实际算力 27.6 TFLOPS
  只沿 N 分块的专用 kernel，M = 64：190 μs → 实际带宽 1.39 TB/s，实际算力 88.9 TFLOPS

同一个 M = 16 的输出，两种分块的账
  输出块 256 × 256：补到 256 行（有效 16，浪费  94%），126 块，矩阵指令算的格子 8.3 M（有用的只有 0.5 M）
  输出块  16 ×  64：补到  16 行（有效 16，浪费   0%），504 块，矩阵指令算的格子 0.5 M（有用的只有 0.5 M）
```

三行数字说明了全部问题。

这个形状**受带宽限制**：要读 264 MB 权重，算的只有 $16 \times 32224 \times 4096 \times 2 \approx 4.2$ GFLOP，算术强度约等于 M = 16，远在拐点左边（第 135 讲算过）。所以唯一该看的指标是**实际带宽**：库选的分块跑出 0.93 TB/s，专用 kernel 跑出 1.73 TB/s——**快了 1.9 倍，而计算量一模一样**。

为什么库会慢？因为 256 × 256 的宏块在行方向上补了 94% 的零。补零不仅浪费矩阵单元的格子（算了 8.3 M 个格子，有用的只有 0.5 M），更关键的是它改变了访存模式：一个 256 行的块要为那 16 行真实数据准备 256 行的空间和流水，片上存储和寄存器都按 256 行配，于是一个计算单元能驻留的块数变少，在途的读请求也变少——而受带宽限制的 kernel 恰恰需要尽可能多的在途请求才能把带宽喂满（第 86 讲）。

专用 kernel 的做法是**只沿 N 分块，M 方向一个块吃完**：块的行数就等于 M（或 16 的倍数），一行零都不补，块数从 126 变成 504，刚好让更多计算单元同时发出读请求。

这解释了另一条注释里更极端的数字：DeepSeek V4 的压缩投影在 decode 时每卡 M ⩽ 64，`hipBLASLt 选了 256×256 的块，不管 N 多大都要约 135 μs`（`vllm_hcu/utils/bf16_fp32_mmac.py` 开头）。“不管 N 多大”是关键——时间完全由分块的固定代价决定，和真实工作量脱钩了。于是海光为 M ⩽ 64、N ∈ 2048、K 是 128 倍数的这一类形状写了专门的 kernel，第 151 讲会逐行读它。

要点

这就是第 87 讲“小 M 矩阵乘”问题在一块新卡上的样子，而且比在 NVIDIA 上更严重：cuBLAS 为 decode 形状准备了多套 kernel 和启发式规则，一个年轻的库往往只有为大矩阵调好的那几套。**适配时要专门测一遍 decode 的形状**（M = 1、8、16、32、64），而不是只测大矩阵的 TFLOPS——后者好看，前者才是服务里真正跑的。

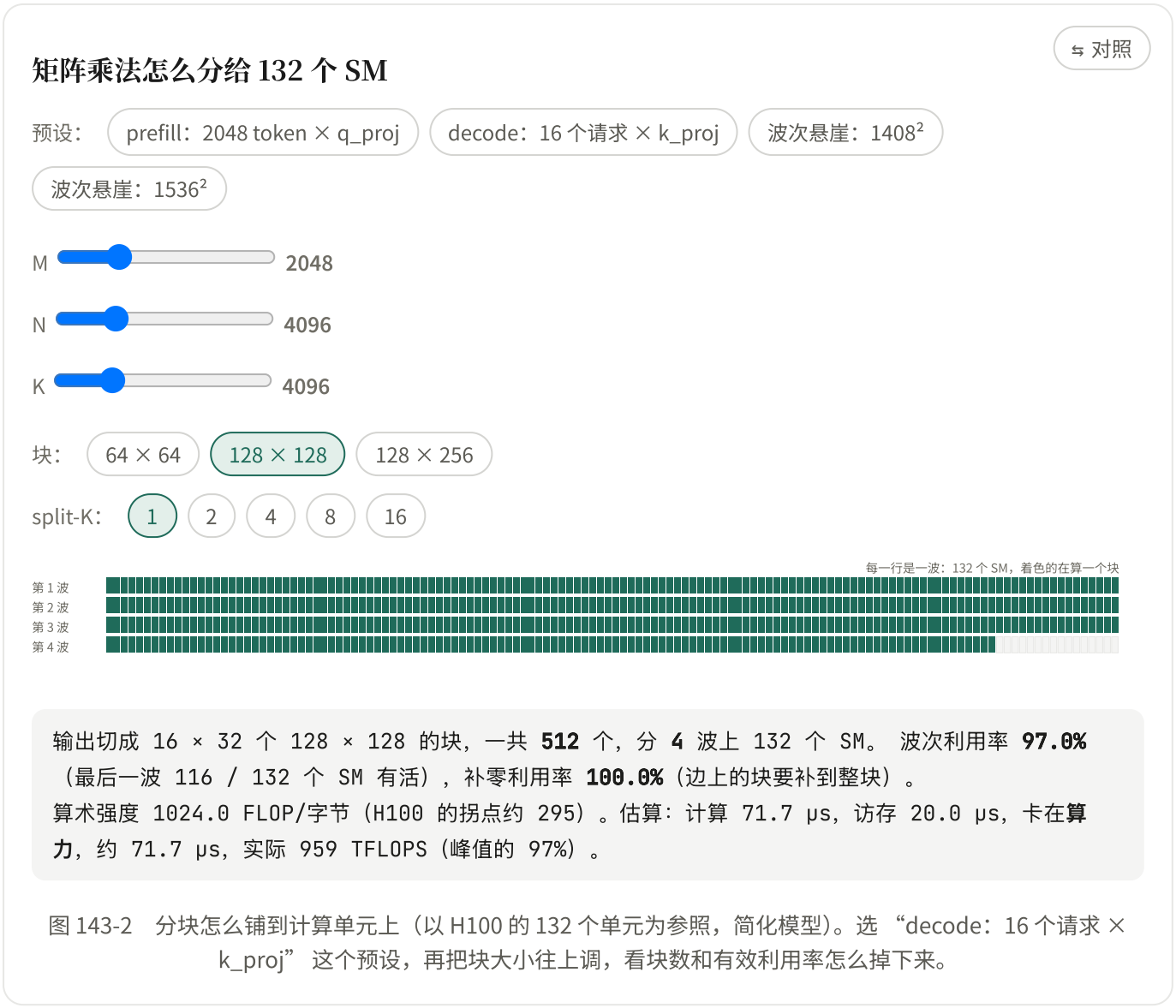


图 143-2　分块怎么铺到计算单元上（以 H100 的 132 个单元为参照，简化模型）。选 “decode：16 个请求 × k_proj” 这个预设，再把块大小往上调，看块数和有效利用率怎么掉下来。

什么时候该自己写、怎么选形状

把上面的逻辑整理成可操作的判断：

形状	谁更合适	理由
M ⩾ 512（prefill、训练）	厂商库	大块复用充分，库里为这类形状调过
64 < M < 512	先测库，再决定	过渡区，和库的启发式规则有关
M ⩽ 64（decode）	往往要专用 kernel	受带宽限制，库的大分块补零太多
N 很大、M 很小（lm_head、MoE 路由）	专用 kernel	纯读权重，只沿 N 分块最优
K 特别大（压缩投影、MLA）	考虑 split-K	块数不足时沿 K 切开凑并行度（第 87 讲）

写专用 kernel 时，形状的选法有几条经验：输出块的行数取 max(M 补齐到指令行数, 指令行数)，不要更大；列数按“一个块能驻留、且块数足够盖住所有计算单元”来定；K 步长按片上存储容量定，留出双缓冲的余量（第 135 讲的 `WaveTile` 可以直接试）。

注意力：三种实现，按条件选

注意力的适配比矩阵乘复杂，因为 prefill 和 decode 要的是两种不同的 kernel（第 84、85 讲）。

prefill 要 FlashAttention 式的实现：注意力矩阵不落地，分块扫过 K、V，在线更新 softmax 的 m、l 和累加器。它的移植难点不在算法，而在分块参数——块大小由片上存储容量决定，而容量换一块卡就变了：



图 143-3　FlashAttention 的块大小受片上存储容量约束。把容量从 228 KB 调到 64 KB，看能放的 Q 行数、扫描次数和读写量怎么变（第 84 讲的模型）。

decode 要 Flash-Decoding 式的实现：查询只有一行（或几行），要沿 KV 的长度方向切开并行，最后归约（第 85 讲）。在新卡上这一步常常是性能缺口最大的地方，因为它的并行度来源和 prefill 完全不同。

兜底是“打分矩阵落地”的朴素实现：`bmm` + `softmax` + `bmm`，就是 miniinfer 的做法。正确，但长上下文下显存和带宽都吃不消。

所以厂商插件里通常不是一个注意力实现，而是**一组后端加一套选择顺序**。海光插件的平台类里就写着这个顺序（`vllm_hcu/platforms/hcu.py` 的 `_get_backend_priorities`，提交 81cbe8f）：稀疏注意力的模型先试稀疏 MLA 的两个后端；MLA 模型先试 FlashMLA，退而用

Triton 版的 MLA；普通的 MHA / GQA 模型先试 FlashAttention，退而用 Triton 版。每个候选都要先通过一次 `validate_configuration` 检查（头数、块大小、数据类型、这块卡的架构支不支持），不通过就换下一个，全都不通过就报错而不是悄悄降级。第 152 讲会读这段代码。

> 注意
> 注意力适配里最容易踩的两个坑。一是**块大小和 KV 分页的块大小耦合**：注意力 kernel 支持的块大小有限（例如只支持 16、32、64），而引擎的 KV 块大小是配置项，两者不匹配时要么报错要么性能很差——海光插件专门有一个 `update_block_size_for_backend` 来调和这件事。二是**布局**：K、V 在显存里按“头优先”还是“序列优先”摆，不同后端的要求不同，混合注意力的模型里可能同时存在两种要求。

适配这两个算子的顺序

1. 先用框架的通用实现（`torch.matmul` + 朴素注意力）跑通跑对，拿到基线（第 141 讲）。
2. 换上厂商库和厂商移植的 FlashAttention，重新跑精度比对（第 139 讲）——库换了，累加顺序就变了。
3. 按真实负载测形状矩阵：prefill 的几个长度、decode 的 M = 1 到 64、几种上下文长度。每个格子记下实际带宽或实际算力占上限的比例。
4. 只给“占比高且离上限远”的格子写专用 kernel。decode 的小 M 几乎总是第一个。
5. 写完之后再跑一遍逐层比对和 token 级比对。

> 专栏
> 这个顺序里第 3 步最容易被跳过，但它是全部判断的依据。做成一张表：行是形状，列是实现（库、厂商 kernel、自写 kernel），格子里是耗时和实际带宽。表一摆出来，“该写哪个 kernel”就不用争论了——海光插件里那两个专用 kernel（lm_head 的和压缩投影的），注释里写的正是这张表上的几个格子。

回顾

矩阵单元给 kernel 三个硬约束：指令形状与对齐、数据排布、每条通道持有的累加器数。decode 的小 M 形状上，厂商库默认的大分块会补掉九成以上的行，于是实际带宽只有专用 kernel 的一半——海光插件注释里的 lm_head 例子是 0.93 TB/s 对 1.73 TB/s，1.9 倍的差距而计算量完全相同；另一处注释更极端：M ⩽ 64 时库选的 256 × 256 分块“不管 N 多大都要约

135 μs”。注意力要按 prefill 和 decode 分别准备实现，块大小随片上容量重选，并且做成一组后端加一套带校验的选择顺序。适配顺序是先通用实现拿基线、再换库、再按形状矩阵测、最后只给值得的格子写 kernel。

自己写 kernel 就要选语言。下一讲比较三条路：Triton、新兴的块级 kernel 语言、厂商自己的 C++ 模板库和 kernel 语言，用同一个 RMSNorm 把几种写法摆在一起看。

第144讲 用 Triton 与厂商语言写 kernel

83 算子融合　88 Triton 入门　136 编程模型与 CUDA 迁移　143 矩阵乘法与注意力的适配

上一讲的结论是有些 kernel 必须自己写。写之前要选语言，而这个选择不只是“用哪种语法”：它决定了这份代码能跑在几种硬件上、写一遍要多久、能逼近硬件上限的几成。

这一讲把同一个算子——RMSNorm——用四种方式写出来摆在一起，并在本机把其中两种真实跑起来量出数字。RMSNorm 是个好例子：数学只有三步（平方求均值、开倒数平方根、乘权重和缩放），但它受带宽限制，所以“写法”带来的差别全部来自访存和下发，没有算力的因素干扰。

四种写法

同一个 RMSNorm，四种写法

⇆ 对照

框架算子组合　Triton　厂商 kernel 语言　块级 DSL

```
def rms_norm(x, w, eps):
    var = x.pow(2).mean(-1, keepdim=True)     # 两个 kernel：pow、mean
    return x * torch.rsqrt(var + eps) * w     # 再三个：rsqrt、两次乘
```

能跑在哪：所有后端（只要基础算子齐，第 137 讲）　**工作量**：几分钟
性能：本机 decode 形状 25.9 μs，有效带宽 0.3 GB/s
要自己操心的：没有。代价是这一个数学操作被摊成五个 kernel，张量被读写五遍，而且下发五次。带宽受限的算子这样写等于白扔几倍的时间。

图 144-1　同一个 RMSNorm 的四种写法。切换标签看代码和它要自己操心什么。Metal 那一份是本机真实编译运行的；Triton 和块级 DSL 的片段没有在本机运行（本机没有这两个后端），代码按各自文档的常见形式写。

四种写法的分水岭有两条。

第一条在**要不要写线程**。框架算子组合和 Triton 都不写线程编号：前者连块都不管，后者以“一个程序处理一整块数据”为单位（第 88 讲）。厂商 kernel 语言要写到通道级：谁读哪一列、归约分几步、什么时候同步。

第二条在**搬运是不是显式的**。Triton 里 `tl.load` / `tl.store` 的背后，编译器自己决定要不要经过片上存储；块级 DSL（TileLang 这一类）要求你声明缓冲、写出搬运语句。多一层控制，换来的是能表达显式搬运硬件所需要的调度（第 135 讲的 NPU 路线）。

在本机真的跑一遍

下面这份脚本在 Apple M2 的 GPU 上对比三种写法：基础算子组合、组合但只让归约升精度、以及一个手写的融合 kernel。本机的“厂商 kernel 语言”是 Metal，它的结构和 HIP、CUDA 是同构的——一个线程组处理一行，组内先做 SIMD 组归约，再跨组归约，然后写回。

bench/port_kernel.py 第 13–43 行 / 共 85 行 ⇆ 对照

```
13 #include <metal_stdlib>
14 using namespace metal;
15 kernel void rms_norm_f32(
16     device const float* x   [[buffer(0)]],
17     device const float* w   [[buffer(1)]],
18     device float* out       [[buffer(2)]],
19     constant uint& d        [[buffer(3)]],
20     constant float& eps     [[buffer(4)]],
21     uint row      [[threadgroup_position_in_grid]],
```

1 kernel 的入口和共享内存 第 15-24 行

三个指针（输入、权重、输出）加两个常量（维度、eps）。后面三个参数由硬件填：这个线程组负责第几行、我是组里第几条通道、组里一共多少条通道。`threadgroup float partial[32]` 是共享内存，用来放各个 SIMD 组的部分和。

CUDA / HIP 里这几样东西一一对应：`blockIdx`、`threadIdx`、`blockDim`、`__shared__`。换平台时要改的是名字，不是结构。

2 **边读边累加平方** 第 25-28 行

3 **跨组归约和同步** 第 29-36 行

4 **写回** 第 37-39 行

5 **编译和它报告的硬件常数** 第 41-43 行

← 上一步 | 看整个文件 | 下一步 →

```
$ .venv/bin/python bench/port_kernel.py                    ⇆ 对照

编译出的 kernel：SIMD 宽度 32，一个线程组最多 1024 个线程

decode（1 行），d = 1024，FP32：
      微秒        相对误差          有效带宽   写法
    25.9    4.2e-08        0.3 GB/s   基础算子组合（5 个 kernel）
    22.3    4.2e-08        0.4 GB/s   组合 + 归约用 FP32
     5.6    4.2e-08        1.5 GB/s   手写融合 kernel（1 个）

prefill（199 行），d = 1024，FP32：
      微秒        相对误差          有效带宽   写法
   122.2    5.4e-08       13.3 GB/s   基础算子组合（5 个 kernel）
   125.4    5.4e-08       13.0 GB/s   组合 + 归约用 FP32
    11.5    5.2e-08      141.3 GB/s   手写融合 kernel（1 个）

大 batch（4096 行），d = 1024，FP32：
      微秒        相对误差          有效带宽   写法
  3181.6    5.3e-08       10.5 GB/s   基础算子组合（5 个 kernel）
  3169.5    5.3e-08       10.6 GB/s   组合 + 归约用 FP32
   394.3    4.8e-08       85.1 GB/s   手写融合 kernel（1 个）
```

图 144-2　三种写法在三种形状上的实测（Apple M2 的 GPU，FP32，参考答案是 CPU 上的 FP64，本机真实运行）。

四个结论。

融合 kernel 在 decode 形状上快 4.6 倍（25.9 μs → 5.6 μs）。这个形状只有一行、4 KB 数据，时间几乎全是固定开销：组合写法要下发五个 kernel，融合写法只下发一个。这正是第 142 讲说的 decode 阶段的典型瓶颈。

在 prefill 形状上快 10.6 倍（122.2 μs → 11.5 μs）。这里数据量够大（199 行 × 1024 列 × 4 字节 ≈ 800 KB），固定开销被摊薄，差距来自访存：组合写法要把张量读写五遍，融合写法一遍。按第 83 讲的融合账本，这个比例应该在 5 倍左右，实测 10 倍——多出来的部分是中间张量的分配和逐元素 kernel 本身的效率。

“有效带宽”这一列的两个反常值要解释。prefill 行的 141.3 GB/s 高于这台机器的内存带宽，因为 800 KB 的数据留在缓存里了，根本没走内存；换成 4096 行（32 MB，放不进缓存）后降到 85.1 GB/s，这个数才是接近内存带宽的真实值。**量带宽时一定要确认数据量超过缓存**，否则量到的是缓存带宽——这是性能测试里最常见的自欺。

三种写法的误差都在 5e-8 量级。融合 kernel 在片上用 FP32 累加，和组合写法的数值行为一致。顺带回答了第 142 讲留的那个问题：“组合 + 归约用 FP32”这个微优化在 decode 形状上只快了 14%（25.9 → 22.3 μs），而写一个融合 kernel 快 4.6 倍——**要改就改到位，不值得在组合写法里做微调**。

> **要点**
> 这个实验可以照搬到任何一块新卡上，而且不需要厂商的 profiler：选一个带宽受限的算子，写三个版本（组合、Triton、手写），在三种形状（decode 一行、prefill 几百行、超过缓存的大 batch）上各量一次，算出有效带宽。拿到的这张表直接回答两件事——这块卡的融合收益有多大、手写值不值得。

怎么选

把四种写法放进一张决策表：

情况	选什么	理由
适配初期，先要跑通	框架算子组合	零成本、正确性有保证（第 140 讲）
逐元素、归一化、激活、RoPE 这类带宽受限的算子	Triton（有后端）或厂商语言	融合收益大、逻辑简单，写一次能用很久
注意力	优先找厂商移植版，其次 Triton	算法复杂（在线 softmax、分页读 KV），自己从零写容易出错
矩阵乘的常规形状	厂商库	库里为这些形状调过（第 143 讲）
矩阵乘的 decode 小 M 形状	厂商语言手写	要控制分块、对齐和矩阵指令，Triton 的抽象够不着最后那几成（第 151 讲）
要跨多家硬件维护一份代码	Triton 或块级 DSL	牺牲一点性能换可移植性
硬件要显式搬运（NPU 路线）	厂商语言或块级 DSL	Triton 的模型里表达不出搬运调度

一个常被忽略的成本是**维护**。手写 kernel 要跟着架构代号走：海光那个小 M kernel 里就有三套分支（`gfx928`、`gfx936` 用一条内建指令，`gfx938` 用另一条，其他架构直接陷阱），每出一代新卡都要加一个分支（第 151 讲）。Triton 的代码不用改，但调好的参数要重调。所以“写几个 kernel”这件事要控制数量：只写占时间的那几个，其余交给库和框架。

> **注意**

自写 kernel 是精度问题的主要来源（第 139 讲）。三条纪律：入口处把假设写成检查（形状、对齐、数据类型不满足就走回退路径，像第 143 讲那个 `supported()` 函数那样）；归约和同步的每一步都对照参考实现验证；每个新 kernel 配一个单算子对比测试，而不是只在整个模型里测。

Triton 在新硬件上的现实

Triton 常被当成“写一次、到处跑”的答案，这个期待要打两个折扣。

第一个折扣是有没有后端。Triton 的编译器要为每种架构实现一个后端（指令选择、寄存器分配、共享内存布局）。NVIDIA 和 AMD 的后端由项目自己维护，其他厂商要么自己接、要么等。海光的容器镜像里确实装了 Triton（第 148 讲会看到构建脚本里 `das-install triton==3.6.0`），说明这条路是通的。

第二个折扣是参数要重调。同一份 Triton 代码，块大小、`num_warps`、流水级数在不同架构上的最优值不同——因为片上存储容量、线程组宽度、矩阵指令形状都变了（第 135 讲）。常见做法是用自动调优（autotune）在目标卡上扫一遍参数，把结果缓存下来；代价是第一次运行很慢，所以要在服务的预热阶段做掉。

还有一类现实选择：**用 Triton 写非矩阵乘的算子，用厂商库和手写 kernel 覆盖矩阵乘和注意力**。海光插件的注意力后端里就能看到这个分工——有 Triton 版的注意力后端作为兜底，也有厂商移植的 FlashAttention 作为首选（第 152 讲）。

回顾

同一个 RMSNorm 的四种写法，差别在两条分水岭：要不要写线程编号，搬运是不是显式的。本机的实测给出了融合的真实收益：手写 Metal kernel 在 decode 形状上比基础算子组合快 4.6 倍（固定开销从五次下发降到一次），在 prefill 形状上快 10.6 倍（张量从读写五遍降到一遍），而三种写法的数值误差都在 5e-8 量级。量带宽时要让数据量超过缓存，否则会量到缓存带宽（本机 800 KB 的形状量出 141 GB/s，32 MB 的形状才是 85 GB/s）。选语言按算子分工：带宽受限的小算子用 Triton 或厂商语言融合，矩阵乘常规形状交给库，decode 的小 M 和注意力按情况手写或用厂商移植版，跨硬件维护一份代码时才牺牲性能换可移植性。

写 kernel 的工具齐了，下一讲回到数据类型：这块卡原生支持哪几种精度？不支持的时候，“用 FP8 省一半带宽”这件事还成不成立？

第145讲 低精度与量化的取舍

63 低精度计算　90 量化基础　93 FP8 与 FP4　140 算子缺口：盘点与自写

第 12 篇讲过一长串压缩方法：INT8 权重量化、AWQ、SmoothQuant、FP8、KV Cache 量化。它们的收益都基于一个前提——硬件原生支持这种数据类型。换一块卡，这个前提可能不成立，而不成立的后果不是“收益小一点”，而是**完全相反**：第 141 讲量到 CPU 上的 BF16 比 FP32 慢 10 倍。

这一讲把这件事量清楚：各类硬件对各种精度的支持分几档；没有原生 kernel 时替代办法能拿到什么、拿不到什么；量化算子适配要改哪些地方。

三档支持，不是两档

“支持”这个词在精度问题上必须分三档，少分一档就会做错判断。

这块卡上，这种精度能不能换来速度

⇆ 对照

能不能用　能省下什么

	FP32	TF32 一类	BF16	FP16	FP8	INT8	FP4 / I
成熟 GPGPU	原生	原生	原生	原生	原生	原生	原生
较新的国产 GPGPU	原生	只能存	原生	原生	原生	原生	只能
NPU / DSA	只能存	不支持	原生	原生	只能存	原生	只能

成熟 GPGPU × FP8：原生支持：矩阵单元直接吃这种类型，省下的字节同时变成省下的时间。按官方文档确认峰值算力和支持的缩放粒度。

FP8：权重和 KV 减半的主力（第 93 讲）。关键问题是有没有原生的 FP8 矩阵乘：没有的话只能反量化成 BF16 再算，省显存但不省带宽也不省时间。

图 145-1　硬件 × 精度的支持矩阵。三列硬件是按路线设的参数，不是某款具体型号——按手上卡的官方文档改这张表。点格子看这种组合该怎么办；切换“能省什么”看同一张表的另一面。

原生支持：矩阵单元直接吃这种类型。少读的字节同时变成少花的时间。

能存不能算：有这个数据类型（能转换、能存显存），但矩阵单元不认它，kernel 里必须先反量化。第 140 讲在 MPS 上遇到的 FP8 就是这一档：`x.to(torch.float8_e4m3fn)` 能跑，`torch._scaled_mm` 不存在。

完全不支持：连类型都没有，只能换一种类型。

第二档最危险，因为它看起来是支持的：模型能加载、显存占用确实降了一半、输出也正确。只有量速度时才发现没快，甚至更慢。

本机实测：六种做法

下面这份脚本把 decode 的一个线性层（M = 16 个 token、权重 4096 × 4096）用六种精度方案各跑一遍，量耗时、要读的权重字节、实际带宽和误差。

```
$ .venv/bin/python bench/port_precision.py                                  ⇆ 对照

decode 的一个线性层：M = 16，权重 4096 × 4096
      微秒        读权重         有效带宽        相对误差   做法
  1186.6      67 MB        57 GB/s    1.1e-06  FP32 权重（基线）
   491.4      34 MB        68 GB/s    3.6e-04  FP16 权重，FP16 算
   544.7      34 MB        62 GB/s    2.9e-03  BF16 权重，BF16 算
  4971.1      17 MB         3 GB/s    8.7e-03  INT8 权重 + 反量化成 FP16 再算（W8A16）
  3922.3      17 MB         4 GB/s    1.2e-02  INT8 权重 + INT8 输入，用 _int_mm（W8A8）
  2055.4      17 MB         8 GB/s    2.7e-02  FP8 权重 + 反量化成 BF16 再算

同一件事换个问法：每种做法相对 FP32 基线快了多少、权重小了多少
  FP32 权重（基线）：1.03× 速度，权重是 FP32 的 1.00 倍
  FP16 权重，FP16 算：2.47× 速度，权重是 FP32 的 0.50 倍
  BF16 权重，BF16 算：2.24× 速度，权重是 FP32 的 0.50 倍
  INT8 权重 + 反量化成 FP16 再算（W8A16）：0.24× 速度，权重是 FP32 的 0.25 倍
  INT8 权重 + INT8 输入，用 _int_mm（W8A8）：0.30× 速度，权重是 FP32 的 0.25 倍
  FP8 权重 + 反量化成 BF16 再算：0.59× 速度，权重是 FP32 的 0.25 倍

本机不支持的：torch._scaled_mm（原生 FP8 矩阵乘）、FP4。缺原生 kernel 时，低精度只省显存容量，
省不下带宽，也省不下时间——上表里反量化那几行就是证据（第 140、145 讲）。
```

图 145-2　同一个线性层的六种精度方案（Apple M2 的 GPU，M = 16，权重 4096 × 4096，参考答案是 CPU 上的 FP64，本机真实运行）。

这张表要横着读，一行一行地看它兑现了什么。

FP32 → FP16：2.51 倍。这是“原生支持”该有的样子：权重字节减半，时间也差不多减半（甚至更多，因为固定开销不变而工作量减半）。误差 3.6e-4，远在第 139 讲的阈值内。

FP16 和 BF16 不等价（491 µs 对 545 µs）。字节数相同，速度差 11%，误差差一个量级（3.6e-4 对 2.9e-3）。原因在第 10 讲的位布局：BF16 尾数 7 位，FP16 尾数 10 位。**选哪个要看模型在哪一步会溢出**——第 139 讲那个 RMSNorm 的例子说明 FP16 的 5 位指数是真实风险，所以主流权重格式仍是 BF16。

INT8 的两种做法都比 FP32 基线慢（4971 µs 和 3922 µs，即 0.23× 和 0.29×）。权重只有 17 MB，比 FP32 的 67 MB 少四分之三，却慢了四倍。这就是“能存不能算”的代价被量化出来了：

- W8A16 那一行的写法是 `w8.half() * sw.half()`——它**先把整个权重反量化成一个新的 FP16 张量**，再做矩阵乘。这一步要写 34 MB、再读 34 MB，比原来直接读 67 MB 还多。
- W8A8 那一行用了 `_int_mm`（MPS 上有这个算子），但 INT8 的输出是 INT32，还要乘回两个缩放系数，又是几次全张量的逐元素运算。

FP8 反量化成 BF16：0.58×。比 INT8 的两种做法好一些（只多一次转换），但仍然比不量化慢。第 140 讲那组数（22273 µs 回退 CPU、1459 µs 组合、526 µs 不量化 BF16）说的是同一件事。

> **要点**
> 量化的收益来自“少读字节”，**而少读的字节只有在 kernel 内部被直接喂进矩阵单元时才变成少花的时间**。把反量化写在矩阵乘外面（物化一个高精度权重张量），等于把省下的带宽又花回去，还要再加一次写入。所以在没有原生低精度矩阵乘的硬件上，量化只剩一个收益：**省显存容量**（能放更大的模型、更多的 KV）。这个收益也是真的，但要在这个前提下重新算部署账（第 71、99 讲）。

那量化在新卡上到底怎么用

按三档支持分别给方案。

原生支持这种类型：直接用，但要核对三件事。一是**缩放粒度**：按张量、按通道、按分组（第 91 讲），硬件的量化矩阵乘通常只支持其中几种，不支持的粒度要么换方法，要么在 kernel 外补算。二是**累加精度**：INT8 乘积累加在 INT32 里不会溢出，FP8 累加在 FP16 还是 FP32 里对精度影响很大。三是**支持的形状**：量化 kernel 的对齐要求往往比普通矩阵乘更严（第 143 讲）。

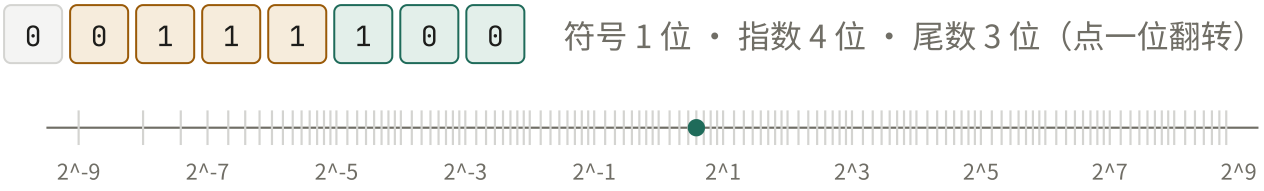
能存不能算：有两条路。短期路线是把反量化**融进 kernel**——读 INT8 权重、在片上转成 BF16、立刻喂给矩阵单元，这样少读的字节真的变成少花的时间（这正是 miniinfer v4 在第 123 讲做的事，也是所有 W8A16 kernel 的做法）。长期路线是等硬件和库支持原生类型。决定走哪条，看这个模型在这块卡上到底是缺显存还是缺带宽。

完全不支持：换类型。FP8 的 checkpoint 在没有 FP8 的卡上，可以转成 INT8（两者都是 8 位，精度特性不同但量级相当，第 93 讲），代价是要重新做校准、重新验精度。

⇆ 对照

8 位、4 位的浮点数长什么样

FP8 E4M3 | FP8 E5M2 | FP4 E2M1



`(1 + 4/8) × 2^(7 - 7) = 1.5`
能表示 127 个非负值，最大 448，最小的正数 0.001953125。
数轴是对数坐标：每个 2 的幂之间格点数一样多，所以格点的“相对”间隔处处差不多——小数附近格点密，大数附近格点疏。整数格式则是等间隔的。

分组缩放：16 个数量化成 FP4

16 个共用一个 scale | 每 8 个一组

x	0.03	-0.12	0.07	2.4	-0.04	0.18	-0.09	0.02	0.11	-0.06	0.14
x ÷ scale → FP4	**0**	**-0.5**	**0**	**6**	**0**	**0.5**	**0**	**0**	**0.5**	**0**	**0.5**
x̂	0.000	-0.200	0.000	2.400	0.000	0.200	0.000	0.000	0.200	0.000	0.200

`scale = 组内最大 |x| ÷ 6：0.4000。除离群值以外的 15 个数，相对误差 56.3%，有 9 个被量化成了 0。`
第 4 个数 2.4 是离群值。整体一个 scale 时，它把 scale 撑到 0.4，其余的数除以 scale 后都不到 0.6，只能变成 0 或 ±0.5。

图 145-3　FP8 和 FP4 的位布局（第 93 讲）。换硬件时要确认支持的是哪一种 FP8（E4M3 还是 E5M2）、缩放是整体一个还是按组一个——这两件事不匹配时 checkpoint 没法直接用。

把这件事放回部署账

量化在一块卡上到底能带来什么，可以在动手之前就算出来。decode 每步的时间下限 = 要读的权重字节 ÷ 带宽（第 68 讲）：

```
# 一块卡上，权重精度决定 decode 每步的时间下限：下限 = 要读的权重字节 ÷ 带宽
P = 8.03e9                          # Llama-3-8B 的参数量（配置见大纲，第 35 讲）
for bw in (3.35e12, 1.6e12):        # 两种带宽：示例值，换成手上卡的官方规格
    print(f"\n带宽 {bw / 1e12:.2f} TB/s")
    print(f"{'精度':<8}{'权重':>9}{'每步下限':>10}{'每秒 token':>12}  有原生 kernel 时 / 只能存不能算时")
    for name, bytes_per in [("BF16", 2), ("FP8", 1), ("INT4", 0.5)]:
        w = P * bytes_per
        t = w / bw
        # 只能存不能算：权重要先反量化成 BF16 物化出来，读写各一遍 BF16 权重，再读一遍 BF16 做乘法
        t_bad = (w + 2 * P * 2 * 2) / bw
        print(f"{name:<8}{w / 1e9:>7.1f} GB{t * 1e3:>9.2f} ms{1 / t:>12.0f}  {t * 1e3:.2f} ms / {t_bad * 1e3:.2f} ms")
```

```
带宽 3.35 TB/s
精度              权重       每步下限     每秒 token  有原生 kernel 时 / 只能存不能算时
BF16        16.1 GB     4.79 ms          209  4.79 ms / 23.97 ms
FP8          8.0 GB     2.40 ms          417  2.40 ms / 21.57 ms
INT4         4.0 GB     1.20 ms          834  1.20 ms / 20.37 ms

带宽 1.60 TB/s
精度              权重       每步下限     每秒 token  有原生 kernel 时 / 只能存不能算时
BF16        16.1 GB    10.04 ms          100  10.04 ms / 50.19 ms
FP8          8.0 GB     5.02 ms          199  5.02 ms / 45.17 ms
INT4         4.0 GB     2.51 ms          399  2.51 ms / 42.66 ms
```

最后一列是这张表的重点：有原生 kernel 时，FP8 把每步从 4.79 ms 砍到 2.40 ms；只能存不能算时，三种精度的每步时间挤在 20 到 24 ms 之间——**彼此的差别几乎消失了，而且都比不量化慢四五倍**。位宽越低，这一列反而越平，因为主导项变成了那次反量化的读写，和权重本身的大小关系不大了。

这张表也给出了适配时的决策顺序：先问“有没有原生 kernel”，再问“反量化在 kernel 内部还是外部”，最后才问“用哪种量化方法”。把顺序倒过来做（先挑方法再适配）是常见的弯路。

量化算子适配要改什么

把一套量化方法搬到新卡上，要动的地方比“换一个矩阵乘”多。按 vLLM 这类引擎的结构，有四处：

权重加载时的重排。量化权重在 checkpoint 里是按某种打包格式存的（4 位两个一字节、按 kernel 要求交错排列）。换卡后 kernel 想要的排列不同，要在加载时重排。这一步出错的症状很典型：输出完全是乱的（不是误差大，是毫无关系），因为权重的每一位都对错位了。

缩放系数的形状和布局。按通道的缩放是一个长度为 N 的向量，按分组的缩放是一个 [N, K/group] 的矩阵；kernel 要求它连续、对齐、也许还要转置。

反量化的位置。在矩阵乘内部（最优）、矩阵乘前（物化一个高精度权重，最差）、还是矩阵乘后（INT8 的输出乘回缩放）。这决定了收益能不能拿到。

回退路径。某个形状不满足 kernel 的对齐要求、或者某一层的量化配置特殊（第一层和最后一层常常保留高精度），要有一条能用的后路。

这四处在海光插件里都能找到对应的代码：平台类里列着支持的量化方法名单（`supported_quantization`，启用 23 种、注释掉 4 种，第 150 讲会读这段），`model_executor/layers/quantization/` 下按方法各有一个实现文件，还有一个 `verify_quantization` 在启动时检查并替换配置（例如用 AWQ 时强制打开 Triton 版的实现）。第 152 讲会把这些展开。

> **注意**
> 量化模型的精度验收不能和 FP32 原模型比（第 139 讲）。正确的参考是**同一套量化方法在参考平台上的输出**：同样的 checkpoint、同样的缩放粒度，在一块成熟的卡上跑出来的结果。两边都掉同样多，说明是量化方法本身的代价；只有这块卡上掉得更多，才是适配的问题。

KV Cache 的精度是另一笔账

权重量化省的是“每步都要读一遍权重”的带宽，KV Cache 量化省的是“每步都要读一遍历史 KV”的带宽和显存（第 94 讲）。它在新卡上有两个额外问题。

写入和读取都要支持这种类型。KV 是每步追加写、每步全量读的。写的时候要量化（算缩放、转类型），读的时候要在注意力 kernel 内部反量化。所以 KV 量化不是一个算子的事，而是**写 KV 的 kernel 和注意力 kernel 都要支持**。

布局更复杂。FP8 的 KV 常常不是简单地“把 BF16 换成 FP8”，而是一种打包布局：把缩放系数和数据交错存在一起，让注意力 kernel 一次读到。海光插件的说明里就有这样一条：对稀疏 MLA 的后端，平台会把用户写的 `--kv-cache-dtype fp8_e4m3` 转成内部的 `fp8_ds_mla` 格式，并明确说这是该后端专用的打包布局、不是所有注意力后端通用的格式。

所以 KV 量化的适配要按“注意力后端 × KV 类型”的组合来验，而不是只验一个开关。

回顾

精度支持要分三档：原生支持（少读的字节变成少花的时间）、能存不能算（只省显存容量）、完全不支持（只能换类型）。中间那一档最容易误判，因为模型能跑、显存确实降了。本机的实测把代价量了出来：FP16 比 FP32 快 2.51 倍（原生支持该有的样子），而 INT8 的两种做法反而只有 0.23× 和 0.29× 的速度，FP8 反量化是 0.58×——因为反量化被写在矩阵乘外面，物化出的高精度权重把省下的带宽又花了回去。要拿到量化的时间收益，反量化必须融进 kernel 内部。量化算子的适配要动四处：权重重排、缩放的形状与布局、反量化的位置、回退路径；KV Cache 量化还要求写 KV 和注意力两个 kernel 都支持，而且常常有专用的打包布局。

单卡的事到这里讲完了。下一讲上多卡：通信库换了一家，拓扑也不一样，第 13 篇算过的那些 All-Reduce 代价还成立吗?

第146讲 集合通信库与多卡互联

100 集合通信原语　101 Ring 与 Tree All-Reduce　102 通信代价模型　110 推理中的张量并行

单卡的适配做完，上多卡。第 13 篇算过张量并行每层两次 All-Reduce 的代价，用的是 α-β 模型：一次通信的时间等于每轮固定开销 α 加上搬运时间 m/β（第 102 讲）。换一块卡，这两个参数都变，拓扑也可能变——于是“TP8 比 TP4 划算吗”这类结论要重算。

这一讲讲四件事：通信库这一层换了什么、没换什么；拓扑怎么改变算法的选择；为什么厂商还要在库之外自己写一个 All-Reduce；以及通信和计算怎么重叠。

库换了，接口基本没换

好消息是这一层的适配成本最低。原因是 PyTorch 的分布式接口（`dist.all_reduce`、`dist.all_gather`、进程组）把具体的库挡在了后面，而各家的库都照着 NCCL 的接口做。

兼容路线上甚至连名字都不改。海光插件的平台类里，分布式后端写的是 `dist_backend: str = "nccl"`（`vllm_hcu/platforms/hcu.py`，提交 81cbe8f），而底下跑的是 RCCL。从调用方看，初始化进程组时写 `backend="nccl"`、模型里写 `dist.all_reduce(y)`，一行都不用动（第 134 讲统计 nano-vllm 的通信相关代码只有 6 行，就是这个道理）。

重写路线上要多做一步：厂商的通信库注册成 PyTorch 的一个新后端，初始化时写 `backend="hccl"` 这样的名字。模型代码里的 `dist.all_reduce` 仍然不用改，但所有写死 `"nccl"` 的地方要改——引擎、启动脚本、文档。

真正要重新量的是参数：

要量的	怎么量	为什么不能照抄
每轮固定开销 α	发一条很小的消息（几 KB），量端到端时间	下发路径、同步机制、驱动实现都不同
有效带宽 β	发一条很大的消息（几百 MB），用字节除时间	标称带宽和有效带宽的比例各家不同
分界消息大小 α·β	两个量出来的数相乘	比它小的消息主要花在延迟上（第 102 讲）
卡间是不是全连接	查拓扑接口，或两两测带宽	决定能用哪些算法

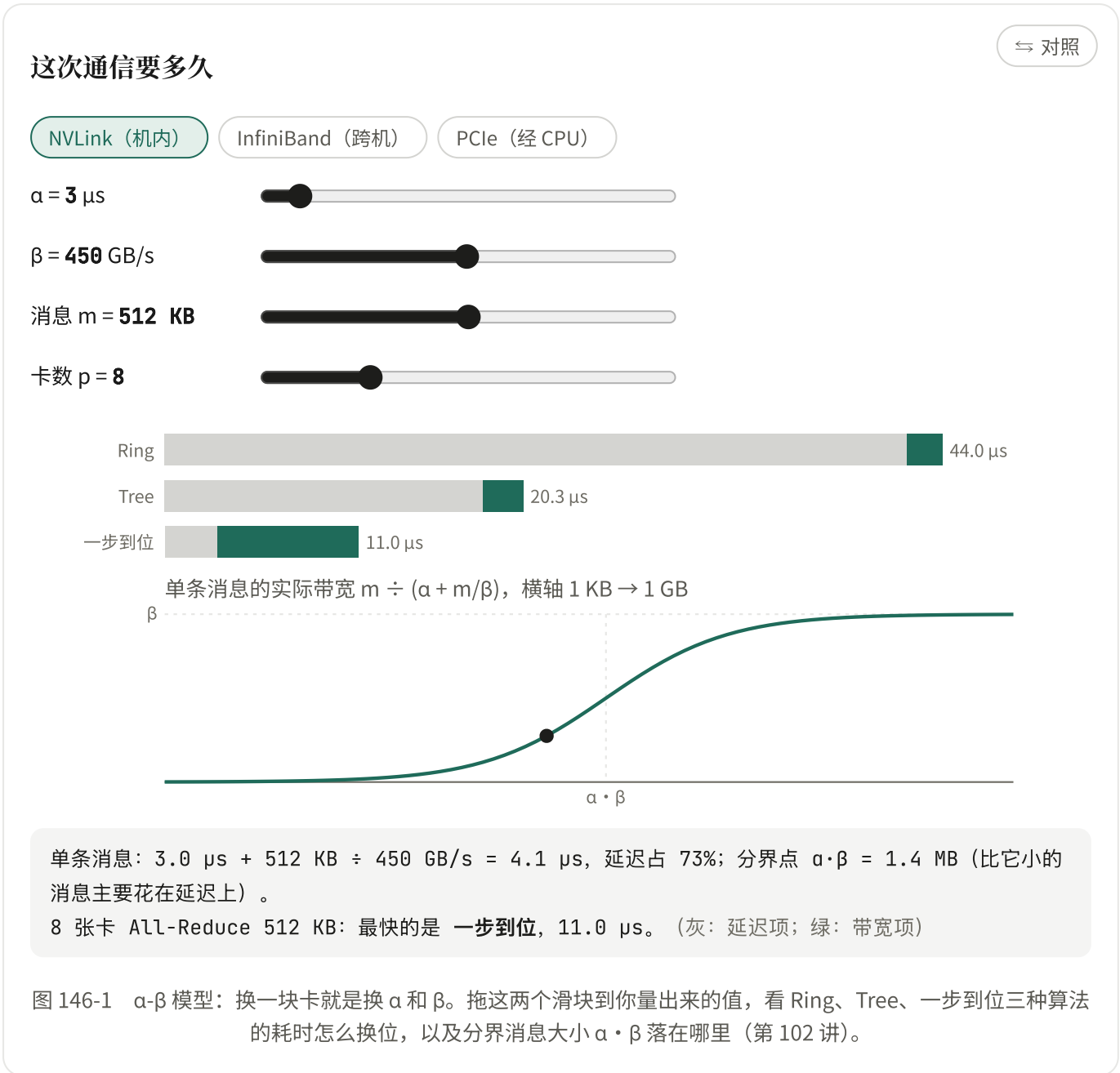


图 146-1　α-β 模型：换一块卡就是换 α 和 β。拖这两个滑块到你量出来的值，看 Ring、Tree、一步到位三种算法的耗时怎么换位，以及分界消息大小 α · β 落在哪里（第 102 讲）。

拓扑决定能用哪些算法

第 102 讲的三种算法在不同拓扑上的可用性不同，这一点换卡时最容易忽略。

全连接（每张卡经交换芯片到每张卡都有直连）：Ring、Tree、一步到位都能用。小消息时一步到位最快（只有一轮固定开销），大消息时 Ring 最省带宽。

环形或网格直连（只有相邻卡有直连）：远端要多跳，跳数直接乘进 α。一步到位用不了——每张卡没法同时和所有卡通信。

跨机：α 大、β 小，所以轮数越少越好；通常用两阶段——先机内 Reduce-Scatter / All-Gather，再机间一轮（第 102、115 讲）。

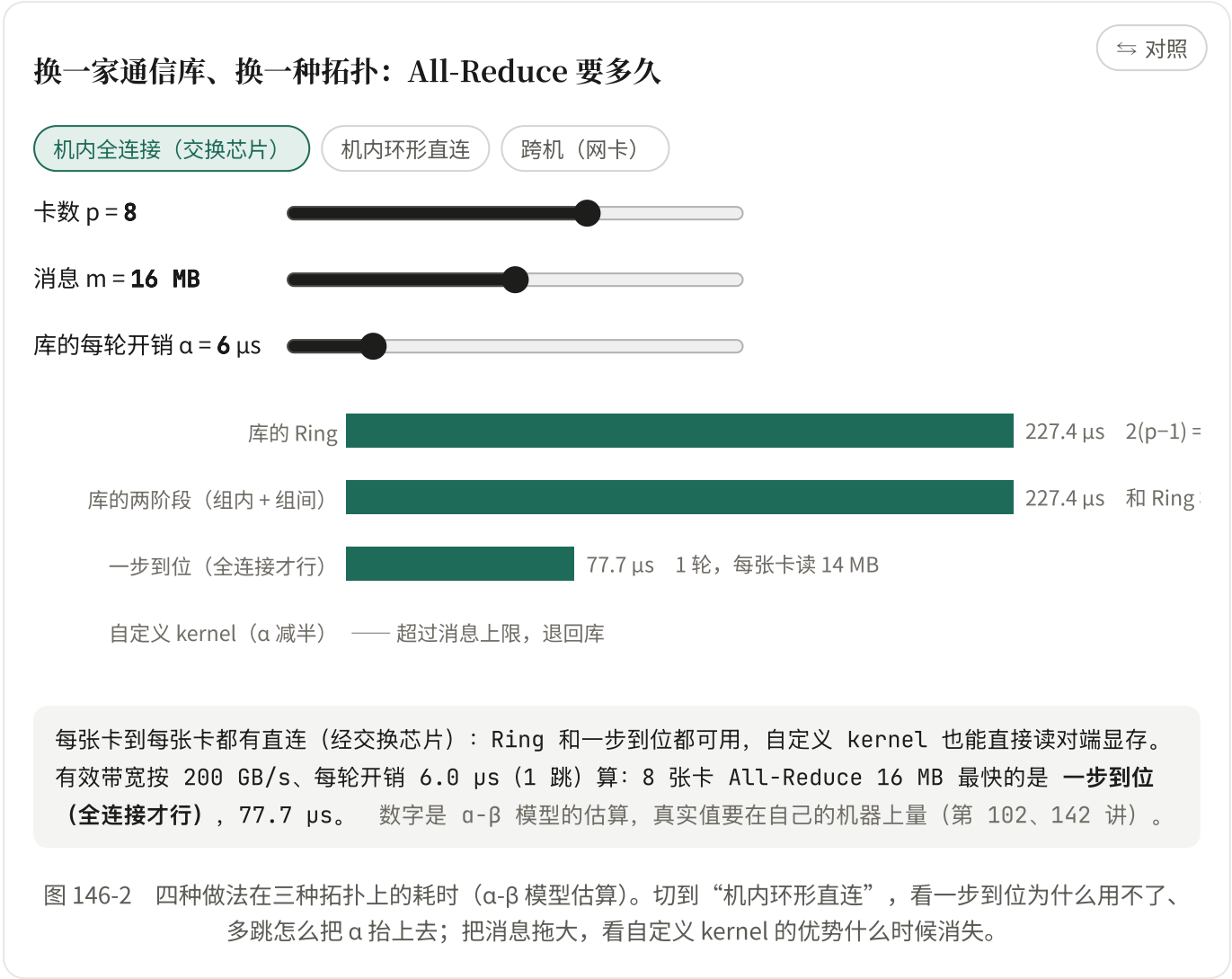


图 146-2　四种做法在三种拓扑上的耗时（α-β 模型估算）。切到“机内环形直连”，看一步到位为什么用不了、多跳怎么把 α 抬上去；把消息拖大，看自定义 kernel 的优势什么时候消失。

实践中还有一件事要查：**P2P 能不能用**。两张卡之间能不能直接读对方的显存，取决于硬件、驱动和虚拟化配置。能用时，自定义 All-Reduce 这类“直接读对端显存”的实现才成立；不能用时，所有通信都要经过主机内存，α 和 β 都会差一个量级。海光插件里就有这一步检查：平台类

的 `is_fully_connected` 通过管理接口逐对查询链路类型和跳数，要求跳数为 1 且类型符合直连，否则就关掉自定义 All-Reduce 并打一条警告（`vllm_hcu/platforms/hcu.py`）。

为什么还要自己写 All-Reduce

库已经有了，为什么 vLLM 和各家插件还都带一个自定义 All-Reduce？因为 decode 的消息很小，而小消息上库的固定开销占比太高。

算一下量级。张量并行度 8、隐藏维度 4096、BF16、一步 decode 的 batch 是 32：一次 All-Reduce 的消息是 $32 \times 4096 \times 2 = 256$ KB。按 $\alpha = 6\ \mu s$、$\beta = 200$ GB/s，Ring 要 $2 \times 7 = 14$ 轮，固定开销就是 84 μs，而搬运时间只有 $2 \times 7 \times 32\ \text{KB} \div 200\ \text{GB/s} \approx 2.2\ \mu s$。**97% 的时间花在每轮的固定开销上**。一层两次 All-Reduce、28 层，就是五六毫秒——比整步的计算时间还长。

自定义 All-Reduce 的做法是绕过库的那套通用机制：把几张卡的缓冲区通过 IPC 互相映射，一个 kernel 里直接读对端显存、就地归约，用标志位做同步。省下的是库的下发路径和多轮同步。

海光插件里的实现（`vllm_hcu/csrc/custom_all_reduce.cuh`、`distributed/device_communicators/custom_all_reduce.py`，提交 81cbe8f）把这类 kernel 的工程约束写得很清楚，值得逐条看，因为每一条都是一个适配要点：

- **只支持 2、4、6、8 卡**（`_SUPPORTED_WORLD_SIZES`）。卡数不在名单里就退回库。
- **消息有上限**，默认 8 MiB；超过就退回库——大消息上库的 Ring 更省带宽。
- **不是全连接时上限压到 512 KB**（源码里写成 `32 * 8192 * 2`），并打一条提示说正在用 PCIe 版的自定义 All-Reduce，性能不好就加 `--disable-custom-all-reduce`。
- **输入字节数必须是 16 的倍数**，而且要求张量基本连续（`should_custom_ar`）。
- **一阶段还是两阶段按消息大小和卡数切换**：全连接时，4 卡以内消息小于 512 KB、8 卡以内小于 256 KB 用一阶段，否则用两阶段（`REDUCE_CASE` 宏里的条件）。PCIe 版的两个阈值小得多（1 MB 和 64 KB）。
- **块数有上限**：`kMaxBlocks = 36`，而 ROCm 这条路径上默认只用 16 个块。源码注释解释了为什么不用更多：NCCL 的 kernel 也只占少量计算单元，块太多会在互联总线上互相争抢。

> **要点**
> 自定义 All-Reduce 不是“更快的 All-Reduce”，而是**在一段很窄的区间里更快**：卡数在名单内、消息小于上限、链路是直连、张量对齐。这四个条件之外它要么不可用、要么更慢。所以适配时它

是一个条件分支，而不是一个替换——写代码时一定要把退回库的那条路留好，并且在压测里覆盖两条路。

同样的思路也用在别的原语上。插件里还有一个 IPC 版的 All-Gather，注释里给出了动机：`NCCL all-gather 在 decode 的消息大小上每次调用约 55 μs`，而 MoE 的 dispatch 每层都要付一次（`vllm_hcu/distributed/device_communicators/ipc_all_gather.py`，提交 81cbe8f）。它的实现是一个 Triton kernel：每个程序把自己那块行直接推进对端的 IPC 缓冲，置一个标志位，再等对端的标志位，然后从自己的缓冲里取出对端的数据。注释里还交代了三个细节——每个程序自带调用计数器所以能被图捕获、两个缓冲槽按调用奇偶交替、**一个程序就是一个波前所以写入有序**。最后这一条正是第 135 讲那个“波前 64”的硬件常数，在通信代码里也要用到。

通信和计算怎么重叠

通信优化的另一半是把它藏起来。第 115 讲讲过超节点上的重叠，这里说适配时能做的三件事。

用单独的流。通信 kernel 放在一条独立的流上，计算在另一条，靠事件同步。前提是硬件和运行时支持多流并发，而且通信 kernel 不会把计算单元占满——上面那个“只用 16 个块”的设定就是为了给计算留资源。

按微批次切开。把一步的 batch 切成两半，第一半的通信和第二半的计算重叠（MoE 的 dispatch / combine 最常这么做，第 111 讲）。这是引擎层的改造，不需要改通信库。

把通信融进算子。例如矩阵乘的输出一边算一边发（Reduce-Scatter 融进 GEMM）。收益最大，工程量也最大，而且要重新验精度（累加顺序变了）。

重叠能不能生效，要在时间线上看（第 142 讲）：通信和计算的条带是否真的并行，还是一前一后。常见的失败原因有两个——中间有一次隐式同步（把数据拷回主机），或者两条流其实被硬件串行化了。

多卡适配的顺序

步	做什么	验收
1	单卡跑通跑对（第 18 篇）	逐 token 一致
2	量 α、β、P2P、拓扑	四个数字写进记录
3	两卡张量并行跑通，和单卡比对输出	逐 token 一致（允许累加顺序带来的 1e-3 量级差异，第 139 讲）
4	扩到 4、8 卡，量 TPOT 和吞吐随卡数的变化	和 α-β 模型的预测对得上，差异能解释（第 110 讲）
5	打开自定义 All-Reduce，两条路都压测	在它适用的区间内更快，之外自动退回
6	做重叠，在时间线上确认	通信条带和计算条带真的并行

第 3 步最容易出问题，而且症状和第 139 讲那类精度 bug 很像：两卡的输出和单卡差很多。原因通常不是通信库算错，而是**切分和归约的位置不对**——权重按行还是按列切、All-Reduce 放在哪一层之后（第 105 讲）。排查办法是关掉通信、在一张卡上模拟两卡的切分，先把切分本身验对。

回顾

通信库这一层的接口适配成本最低：兼容路线上连后端名字都仍叫 `nccl`，模型代码里的 `dist.all_reduce` 不用改；重写路线要注册一个新后端，但调用方照旧。真正要重新量的是四个数：每轮固定开销 α、有效带宽 β、两者的乘积（小消息的分界线）、以及卡间是不是全连接。拓扑决定能用哪些算法：非全连接时一步到位用不了、跳数要算进 α。decode 的 All-Reduce 消息只有几百 KB，97% 的时间花在每轮固定开销上，所以厂商要在库之外自己写 All-Reduce；海光的实现把适用区间写成了显式条件（2–8 卡、消息 ⩽ 8 MiB、全连接、16 字节对齐，非全连接时上限压到 512 KB），区间之外退回库。它们的 IPC All-Gather 注释里还给出了一个具体动机：NCCL 的 All-Gather 在 decode 消息大小上每次约 55 μs。

第 19 篇的原理还剩一块：这些选择（注意力后端、量化方法、通信方式、专用 kernel）怎么告诉 vLLM 这样的引擎，而且不改它的源码？下一讲讲硬件插件。

第147讲 推理引擎的硬件插件

重点

129 vLLM 的架构　137 框架怎么接入新硬件　143 矩阵乘法与注意力的适配　146 集合通信库与多卡互联

前面几讲攒下了一堆与硬件相关的决定：用哪个注意力后端、矩阵乘走库还是走专用 kernel、支持哪些量化方法、All-Reduce 走库还是走自定义实现、块大小取多少、要不要开图捕获。现在的问题是：**怎么把这些决定告诉 vLLM，而又不改它的代码**。

为什么不能改？第 17 篇读过 vLLM 的规模——17 万行，几周一个版本。在上游代码里加一个 `if is_my_hardware()` 分支，意味着每次上游更新都要重新合并一遍；而厂商需要支持的模型和特性比上游合并的节奏更快。所以主流做法是**插件**：引擎留出入口，厂商在自己的包里实现，安装之后引擎自动发现。

这一讲讲插件机制本身：引擎留了哪些入口、插件要实现哪些接口、哪些必须实现、不改上游代码怎么替换掉上游的模块，以及版本怎么对齐。第 150 讲会用这套框架逐行读海光的插件。

引擎留的三类入口

vLLM 用 Python 标准的 entry point 机制。插件在自己的 `setup.py` 里声明入口，引擎启动时扫描已安装的包，把每一类入口都加载一遍。按官方文档，和硬件适配相关的有两类：

- **平台插件**（`vllm.platform_plugins`）：入口是一个函数，返回平台类的全限定名；这块卡在当前环境不可用时返回 `None`。
- **通用插件**（`vllm.general_plugins`）：用来注册模型类、导入自定义算子。

文档里还强调了一条实现要求：**入口函数必须可重入**——它可能被调用多次（引擎有多个进程，每个进程都要加载插件），所以重复调用不能出问题。这一点在第 150 讲会看到海光是怎么处理的（进程内的注册表 + 失败锁存）。

插件要实现什么

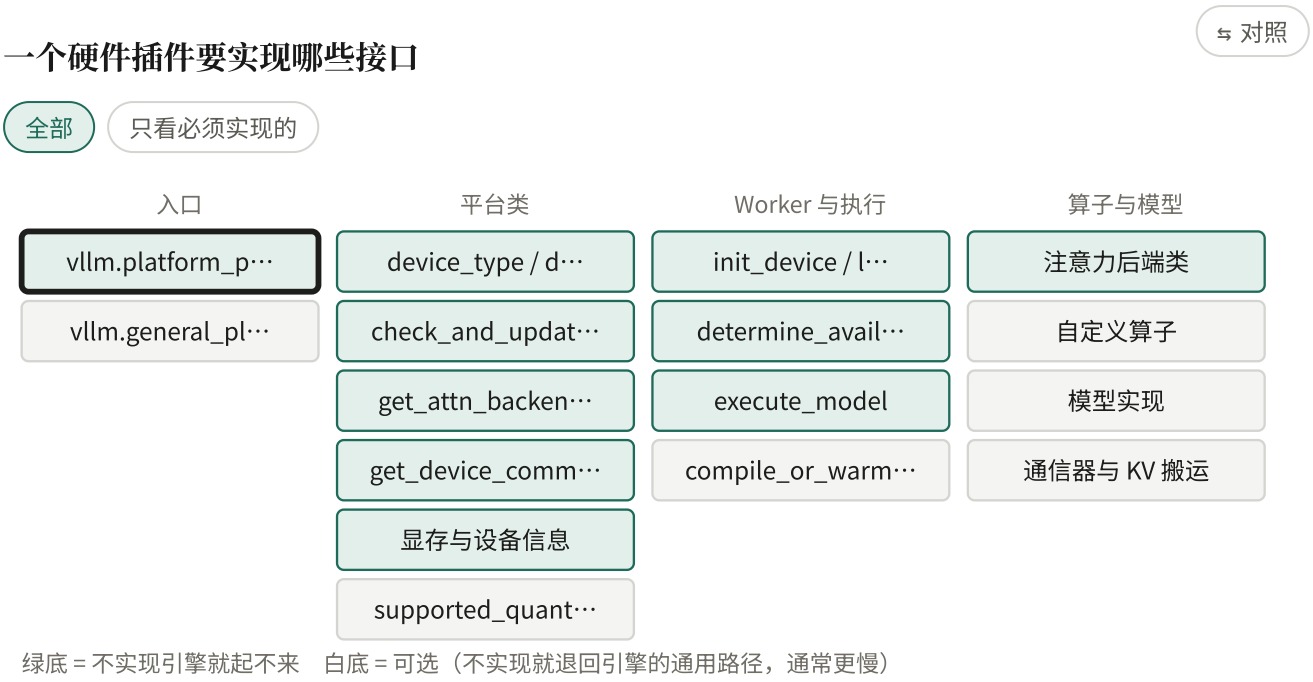


图 147-1　一个硬件插件的接口清单，按四层分组（内容按 vLLM 公开文档的插件机制整理）。绿底是不实现引擎就起不来的；点任意一项看它替换了引擎里的什么。

四层的分工可以这样理解：

入口只做注册，不做实际工作——它要足够轻，不能在这里 import torch 或者初始化设备，否则会在插件发现阶段就把一堆东西拽进来。

平台类是引擎问硬件问题的地方。它继承引擎的 `Platform` 基类，回答一长串问题：设备名是什么、分发键是什么、有几张卡、每张卡多少显存、支持哪些量化方法、该用哪个注意力后端、该用哪个通信器、用哪个 worker 类。按官方文档，最少要实现的是 `device_type`、`device_name`、`check_and_update_config`、`get_attn_backend_cls`、`get_device_communicator_cls`；其中 `check_and_update_config` 最关键，因为worker 类就是在这里指定的。

Worker 与执行是真正跑模型的地方。文档列出的基本方法是：`init_device`、`initialize_cache`、`load_model`、`get_kv_cache_layer_configs`、`determine_available_memory`、`initialize_from_config`、`execute_model`；另外按需实

现休眠唤醒、图模式、投机解码、LoRA、数据并行的空步。

算子与模型是性能的落点：注意力后端类、自定义算子、特殊模型实现、通信器、PD 分离的 KV 搬运。

> **要点**
> 这张清单有一个重要的性质：**引擎的核心逻辑一项都不在里面**。调度、token 预算、KV 块管理、前缀缓存、请求生命周期（第 129 讲读过的那些）全部由上游负责，插件只回答“在这块卡上这件事怎么做”。所以插件的工作量和引擎的规模无关，只和硬件的差异有关——这也是插件机制能成立的原因。

沿用 CUDA 键的插件能省掉多少

第 137 讲讲的两条接法，在插件这一层的差别很大。

沿用 CUDA 键（海光 DCU 这条路）：平台类里 `device_type = "cuda"`、`dispatch_key = "CUDA"`、通信后端叫 `nccl`、设备可见性环境变量用 `CUDA_VISIBLE_DEVICES`。于是引擎里所有“如果是 CUDA 就……”的分支自动走对，worker 可以直接继承引擎自带的 GPU worker，图捕获可以直接复用引擎的 CUDA Graph 包装器。插件要做的事缩小成“替换掉那些性能不够或语义不同的部分”。

注册新设备（昇腾这条路）：引擎里每一处判断设备类型的地方都要有自己的分支，worker、注意力后端、KV Cache 的管理、图模式几乎都要重写一遍。插件的体量会大得多。

两者的共同点是：**上游代码都不用改**。差别只在插件自己要写多少。

不改上游，怎么替换上游的模块

插件机制能注册新东西（平台、模型、算子），但有些适配需求是“要改上游某个函数的行为”。比如上游的调度器里有一处判断，在这块卡上要换一个阈值；或者上游的某个算子实现用了这块卡不支持的指令，要整体换掉。

这类需求有三种做法，代价依次递增：

一是把它变成配置。如果上游已经留了配置项（块大小、注意力后端、自定义算子开关），平台类的 `check_and_update_config` 里改配置就够了。这是最该优先走的路。

二是运行时替换。在引擎还没导入某个模块之前，先安排好“当它被导入时，用我的实现替换掉”。Python 的导入机制允许在 `sys.meta_path` 上装一个查找器，拦住指定的模块名；也可以在模块导入之后替换它的某个属性（函数、类的方法）。这就是常说的猴子补丁，但做法上有讲究：

- 只拦**精确的模块名**，不要按前缀匹配，否则会误伤无关模块；
- 每个补丁只应用一次，并记录状态（幂等）；
- 失败要**锁存并抛出**，不能悄悄回退到未打补丁的状态——否则会出现“一半打了补丁、一半没打”的诡异状态；
- 要能**报告**：打了哪些补丁、目标是什么、成功还是失败。

三是 fork 上游。维护成本最高，一般只在等不及上游合并时作为过渡。

海光插件选的是第二条路，并且把上面四条做成了明确的机制：一个精确匹配模块名的导入协调器、一个进程内的补丁注册表（记录进程角色、目标符号、状态和失败详情）、以及一份显式排好序的补丁清单。它的 README 里写明两条原则：**安装和启动不会重写 vLLM 的包文件，也不创建源码软链接；必需功能不兼容时失败并保留诊断状态，不回退到重写源码**。第 150 讲会逐行读这套机制。

> **注意**
> 运行时替换的最大风险是**版本错配**：上游改了被替换函数的签名或语义，补丁还按老的写，结果可能是跑错而不是报错。所以这类插件必须有一道版本检查的关卡——在任何补丁注册之前，先确认装的引擎版本在支持的范围内，不在就拒绝启动。

版本对齐

插件夹在引擎和硬件栈之间，两边都在动：引擎几周一个版本，驱动和工具包几个月一个版本，算子库跟着走。所以一个可用的插件必须声明清楚“我支持什么”。

一个完整的版本矩阵包括：驱动版本、工具包版本、PyTorch 版本、引擎版本、插件版本，以及若干算子库的版本。检查的时机要足够早——在打任何补丁、初始化任何设备之前，而且要**只读元数据、不导入被检查的包**（导入会触发一堆副作用）。

匹配的严格程度要分情况。版本号里的主次补丁号通常要求一致（接口可能变），而预发布、开发版、本地构建后缀可以放宽（同一条发布线上的不同构建）。海光插件的做法正是这样：要求已安装的 vLLM 的版本号在主次补丁这一级和目标版本一致，允许 `rc`、`dev`、本地构建后缀不

同；并且在文档里明确提醒——**通过版本检查不代表 PyTorch、工具包和算子的二进制接口已验证兼容**。第 149 讲会把这个矩阵的每一层展开。

一个插件的代码规模

抽象讲完，看一个真实插件的体量。下面这份统计只读源码文本，不运行、不导入：

```
$ .venv/bin/python bench/das_loc.py                                    ⇆ 对照

vllm-plugin-das（提交 81cbe8f）：818 个源文件（.py .cu .cuh .cpp .h），255134 行（不算空行）
   115629  tests
    41325  vllm_hcu/model_executor
    36330  vllm_hcu/patch
    23517  vllm_hcu/v1
    15804  vllm_hcu/models
     7580  vllm_hcu/distributed
     3037  vllm_hcu/ops
     3013  tools
     2371  vllm_hcu/csrc
     1684  vllm_hcu/platforms
     1553  vllm_hcu/runtime_compat
      944  vllm_hcu
      891  examples
      634  vllm_hcu/utils
setup.py 里登记的入口：
  vllm.platform_plugins   hcu = vllm_hcu:hcu_platform_plugin
  vllm.general_plugins    hcu_model = vllm_hcu:hcu_platform_register_model
  vllm.general_plugins    hcu_ops = vllm_hcu:hcu_platform_register_ops
运行时补丁文件（vllm_hcu/patch/ 下以 patch_ 开头的文件）：platform/core_fix 24 个，
platform/framework_opt 21 个，worker/core_fix 42 个，worker/framework_opt 35 个，
worker/op_opt 73 个，共 195 个
整模块替换（module_exchange.py）：16 对，例如
  vllm.model_executor.layers.fused_moe.modular_kernel
    → vllm_hcu.model_executor.layers.fused_moe.modular_kernel
  vllm.model_executor.layers.linear
    → vllm_hcu.model_executor.layers.linear
  vllm.v1.attention.backends.mla.sparse_swa
    → vllm_hcu.v1.attention.backends.mla.sparse_swa
HCUPlatform.device_name = "hip"
HCUPlatform.device_type = "cuda"
HCUPlatform.dispatch_key = "CUDA"
HCUPlatform.dist_backend = "nccl"
HCUPlatform.device_control_env_var = "CUDA_VISIBLE_DEVICES"
supported_quantization：启用 23 种，注释掉 4 种（mxfp4、bitsandbytes、modelopt_fp4、
gpt_oss_mxfp4）
源码里出现的架构代号：gfx928（2 处）、gfx936（10 处）、gfx938（87 处）、gfx950（6 处）
把线程组宽度写成 64 的地方：3 处
```

图 147-2　海光 vLLM 插件（HYGON-AI/vllm-plugin-das，提交 81cbe8f，Apache-2.0）的规模与结构（只读源码文本统计）。

几个数字值得先放在这里，第 20 篇会逐项展开：

- 整个插件 25 万行，其中测试占 11.5 万行——**测试比实现还多**，这对一个要跟着两边版本跑的项目是必需的。
- 实现部分里，算子与层（`model_executor`）4.1 万行最大，补丁机制（`patch`）3.6 万行第二，worker 与注意力（`v1`）2.4 万行第三。
- 平台类只有 1684 行，但它是整个插件的入口和总开关。
- 补丁一共 195 个文件，分成平台阶段和 worker 阶段，每个阶段再分“修正”“框架适配”“算子优化”三类。
- 整模块替换 16 对，都是“当上游的某个模块被导入时，换成插件的同名实现”。

专栏

对比一下第 16 篇读的 nano-vllm：1195 行就实现了一个能跑的引擎。而一个**只做硬件适配**的插件是 25 万行。这个对比说明了工业级推理系统的成本结构：核心算法不大，大的是“在真实硬件、真实模型、真实版本组合上让它一直对”的那部分。

回顾

把一块新卡接进推理引擎，标准做法是插件而不是改上游：引擎用 entry point 机制留出平台插件和通用插件两类入口，插件实现一个平台类（回答设备名、分发键、显存、注意力后端、通信器、worker 类等问题）、一个 worker（设备初始化、加载权重、量显存、执行一步）、以及注意力后端和自定义算子。引擎的调度、KV 管理、请求生命周期一项都不需要插件参与，这是插件机制成立的根本原因。沿用 CUDA 分发键的硬件能白拿引擎里所有 CUDA 分支，插件体量小得多。要改上游行为时，优先改配置，其次用精确匹配、幂等、失败锁存、可报告的运行时替换，最后才考虑 fork；而运行时替换必须配一道版本检查的关卡。海光插件的规模统计给出了这件事的真实成本：25 万行，其中测试 11.5 万行，补丁 195 个文件。

原理部分到此结束。接下来第 20 篇把这两篇的每一条都落到一块具体的国产卡上：海光 DCU 的软硬件栈长什么样，CUDA 代码怎么迁过去，插件里那些决定是怎么写的，以及一个模型上线要走完哪些验收。

AI INFRA · HANDBOOK

第 20 篇

案例：海光 DCU

收录 6 讲 · 第 148—153 讲

第148讲 海光 DCU 的软硬件栈

134 换一块卡要重做什么
135 国产与 NVIDIA 硬件差异
143 矩阵乘法与注意力的适配
147 推理引擎的硬件插件

前两篇讲的是通用原理。这一篇把它们落到一块具体的国产卡上：海光的 DCU（深度计算单元）。

选它作为案例有两个实际原因。一是它走的是第 134 讲说的**兼容 GPGPU 路线**，和 ROCm 一脉相承，所以前两篇讲的每一条都能对上号，差异也看得清楚。二是它有**可以逐行读的开源代码**：海光把 vLLM 插件开源在 GitHub（`HYGON-AI/vllm-plugin-das`，Apache-2.0），25 万行，平台类、补丁机制、专用 kernel、量化和通信的选择全在里面。

> **注意**
> 这一篇的读法先说清楚。代码部分读的是 `miniinfer/third_party/vllm-plugin-das`，固定在提交 `81cbe8f`，**只读不运行**——跑它需要海光的卡和配套的 DTK 环境，本站没有。所有性能数字只引用两个来源：源码里的注释，以及官方公开文档；每处都注明出处。没有出处的数字一律不写。硬件规格（算力、显存、带宽、卡间互联）**以官方产品文档为准**，本篇只讲结构和软件栈，组件里的硬件参数都做成可输入的（第 135 讲）。

两栏对照：海光的栈和 CUDA 的栈



图 148-1　海光 DCU 的九层软件栈。切到 NVIDIA（CUDA）那一列逐层对照，看每一层的对应物和换卡时的工作量；点任意一层看说明。

自上而下说清每一层是什么，以及这一层的名字在海光这边叫什么。

硬件与驱动。DCU 是 GPGPU 架构的加速卡，编程模型和 CUDA 同构：线程、线程块、线程组、共享内存、矩阵核心。驱动之上有厂商的管理工具 `hy-smi`（对应 `nvidia-smi`，看卡的状态和占用）。插件里查卡间链路拓扑时用的是 `amdsmi` 这一类管理接口——逐对查询两张卡之间的跳数和链路类型，要求跳数为 1 才认为是直连（`vllm_hcu/platforms/hcu.py` 的 `is_fully_connected`，提交 81cbe8f，第 146 讲讲过它的用途）。

DTK（深算工具包）。对应 CUDA Toolkit：编译器、HIP 运行时、基础数学库都在里面，默认装在 `/opt/dtk`。插件的构建脚本印证了这几件事——环境变量用 `ROCM_PATH`，版本号从 `$ROCM_PATH/.info/rocm_version` 读出来并写进 wheel 的版本后缀（`setup.py`），CI 的环境配置里检查的 DTK 发布线是 `26.04` 和 `26.10`（`.github/workflows/configs/hcu-runner-environment.json`）。容器构建脚本里还能看到编译器是单独安装的一个组件（脚本名 `hcu_llvm_installer.sh`，按大版本号选择），以及一个单独覆盖进 DTK 的数学库包。

PyTorch 的 DCU 版。按 ROCm 的做法用 HIP 重新编译，对外仍然自称 `cuda` 设备。这是整条路线最省事的一环，第 149 讲会专门讲它带来的便利和别扭之处。

DAS 与 OpenDAS。这一层是框架和组件的适配层，对应 ROCm 生态里那一堆库加工具：注意力（FlashAttention、FlashMLA）、MoE 通信（DeepEP）、矩阵乘模板库、算子库、kernel 语言工具链等。插件的容器构建脚本里一次性列出了它依赖的组件（`docker/Dockerfile`）：`triton==3.6.0`、`flash_attn`、`lightop`、`lmslim`、`deepgemm`、`aiter`、`deep_ep`、`boltops`、`causal_conv1d`、`flash_mla`、`flash_kda`、`fastsafetensors`、`nixl`、`lmcache`、`mooncake_transfer_engine`、`vllm-router`，最后装 `tilelang`。这张清单比任何描述都直观：**适配一块卡的工作量，一大半在这些组件上**。

引擎与插件。vLLM 本身是上游的，加上 `vllm_hcu` 这个插件。插件用 entry point 注册，再用运行时补丁替换上游的部分模块（第 147 讲的机制，第 150 讲逐行读）。

> **专栏**
> 官方资料里对这一层的描述（光合开发者社区、DAS 的发布说明）还提到：OpenDAS 开源了数十个组件库，LightOp 算子库包含 w4a8、w8a8 等量化算子；DAS 的某个版本标注支持 vLLM 0.11 和 SGLang 0.5.7，以及 FP8、PD 分离、大规模专家并行。这些数字和版本随发布变化很快，**以官方最新文档为准**；而我们手上这份插件源码的目标引擎版本是 vLLM 0.28.1（`vllm_hcu/version.py`），比上面那个版本号更新——这正说明版本矩阵要按仓库里的声明核对，不能照抄文档（第 149 讲）。

卡里面：波前 64 和 MMAC

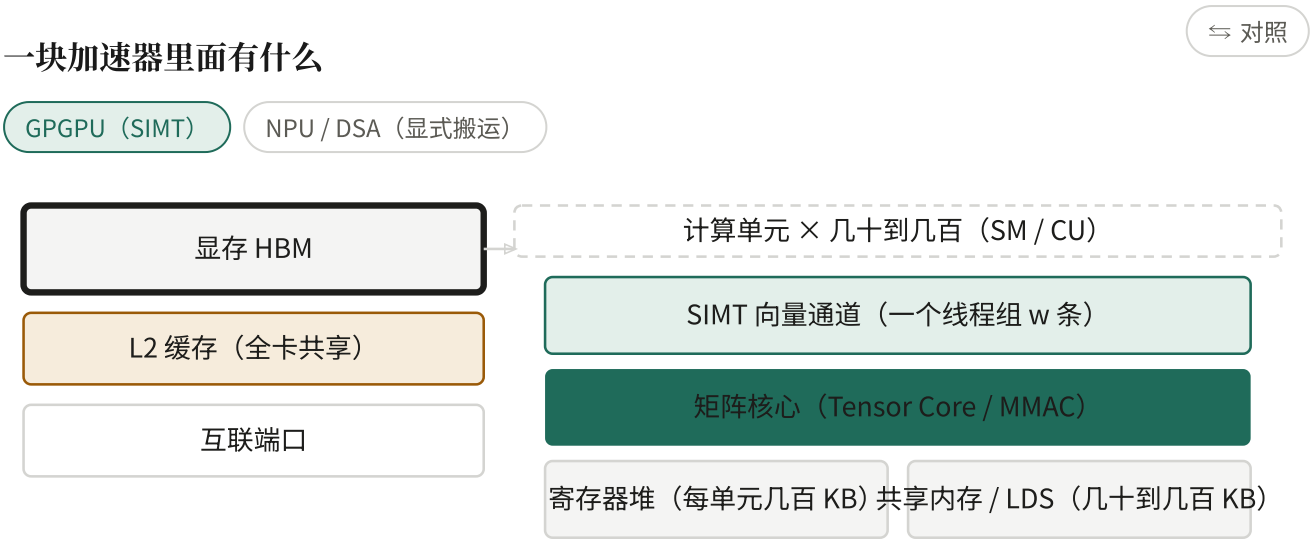


图 148-2　GPGPU 路线的卡内结构（示意图）。DCU 属于这一类：SIMT 的向量通道 + 矩阵核心 + 寄存器 + 共享内存，卡级有 L2、显存和互联端口。具体的单元数量、容量和带宽以官方文档为准。

从插件源码里能确认两个硬件常数，它们正是第 135 讲强调的那两类。

一个线程组（波前）是 64 条通道。证据是那个小 M 矩阵乘 kernel 里的 `constexpr std::int32_t kWaveSize = 64;`（`vllm_hcu/utils/bf16_fp32_mmac.py`，提交 81cbe8f），以及通信代码里一句设计说明：`一个程序就是一个波前，所以这次释放操作把它的所有写入都排好了序`（`vllm_hcu/distributed/device_communicators/ipc_all_gather.py`）。后一句很有意思——它把“波前宽度”当作同步语义的依据，而不只是性能参数。

矩阵核心指令叫 MMAC。kernel 里按架构分支调用的内建函数是：

```
#if defined(__gfx936__) || defined(__gfx928__)
  reg_c = __builtin_amdgcn_mmac_f32_16x16x16bf16(...)
#elif defined(__gfx938__)
  reg_c = __builtin_hcu_mmac_f32_16x16x16_bf16_lit_lts(...)
#endif
```

（`vllm_hcu/utils/bf16_fp32_mmac.py` 的 `hcu_mmac_bf16`，提交 81cbe8f。）从名字能读出指令的形状：一次吃 16×16×16，输入 BF16、累加 FP32——正是第 135 讲那个“16×16×16”的例子。所以在 64 条通道的硬件上，每条通道持有 256 ÷ 64 = 4 个 FP32 累加器。

架构代号是 gfx 加数字。整份源码里出现的代号用脚本数一遍：

```
$ .venv/bin/python bench/das_loc.py
```

⇆ 对照

```
vllm-plugin-das（提交 81cbe8f）：818 个源文件（.py .cu .cuh .cpp .h），255134 行（不算空行）
   115629  tests
    41325  vllm_hcu/model_executor
    36330  vllm_hcu/patch
    23517  vllm_hcu/v1
    15804  vllm_hcu/models
     7580  vllm_hcu/distributed
     3037  vllm_hcu/ops
     3013  tools
     2371  vllm_hcu/csrc
     1684  vllm_hcu/platforms
     1553  vllm_hcu/runtime_compat
      944  vllm_hcu
      891  examples
      634  vllm_hcu/utils
setup.py 里登记的入口：
  vllm.platform_plugins   hcu = vllm_hcu:hcu_platform_plugin
  vllm.general_plugins    hcu_model = vllm_hcu:hcu_platform_register_model
  vllm.general_plugins    hcu_ops = vllm_hcu:hcu_platform_register_ops
运行时补丁文件（vllm_hcu/patch/ 下以 patch_ 开头的文件）：platform/core_fix 24 个，
platform/framework_opt 21 个，worker/core_fix 42 个，worker/framework_opt 35 个，
worker/op_opt 73 个，共 195 个
整模块替换（module_exchange.py）：16 对，例如
  vllm.model_executor.layers.fused_moe.modular_kernel
    → vllm_hcu.model_executor.layers.fused_moe.modular_kernel
  vllm.model_executor.layers.linear
    → vllm_hcu.model_executor.layers.linear
  vllm.v1.attention.backends.mla.sparse_swa
    → vllm_hcu.v1.attention.backends.mla.sparse_swa
HCUPlatform.device_name = "hip"
HCUPlatform.device_type = "cuda"
HCUPlatform.dispatch_key = "CUDA"
HCUPlatform.dist_backend = "nccl"
HCUPlatform.device_control_env_var = "CUDA_VISIBLE_DEVICES"
supported_quantization：启用 23 种，注释掉 4 种（mxfp4、bitsandbytes、modelopt_fp4、
gpt_oss_mxfp4）
源码里出现的架构代号：gfx928（2 处）、gfx936（10 处）、gfx938（87 处）、gfx950（6 处）
把线程组宽度写成 64 的地方：3 处
```

图 148-3　插件源码的规模、结构，以及里面出现的架构代号与硬件常数（只读源码文本统计，提交 81cbe8f）。

统计结果里的几项，配合前面的内容读：

- **架构代号**出现 `gfx928`、`gfx936`、`gfx938`、`gfx950`，其中 `gfx938` 出现 87 处，远多于其他——说明这份代码主要针对这一代卡调过。代号和具体产品型号的对应关系以官方文档为准，但代码里的分布直接告诉你**哪一代是当前的主力**。
- **把线程组宽度写成 64 的地方有 3 处**，全部在那个小 M kernel 里。第 136 讲说过这个数字是静默错误的高发区，这里它被写成了具名常量而不是裸字面量。
- **`supported_quantization` 启用 23 种、注释掉 4 种**。被注释掉的是 `mxfp4`、`bitsandbytes`、`modelopt_fp4`、`gpt_oss_mxfp4`——第 152 讲会看这张名单意味着什么。
- **平台类 1033 行、补丁 195 个文件、整模块替换 16 对**：第 150 讲的主题。

各层对应的讲

这一篇后面五讲按软件栈的层次展开，和前两篇的对应关系如下：

层	海光这边是什么	对应原理讲	本篇哪一讲
硬件与驱动	GPGPU、波前 64、MMAC、gfx 代号、hy-smi	135	本讲
工具包	DTK（编译器、HIP 运行时、数学库）	134、136	149
框架	PyTorch 的 DCU 版，仍自称 cuda	137	149
引擎接入	vllm-plugin-das 的平台类与运行时补丁	147	150
矩阵乘	hipBLASLt、MMAC、小 M 专用 kernel	143	151
注意力 / 量化 / 通信	后端选择顺序、量化名单、RCCL 与自定义 All-Reduce	143–146	152
上线	环境、跑通、精度、性能、多卡、服务化	全部	153

回顾

海光 DCU 走兼容 GPGPU 路线，九层软件栈逐层都有对应物：驱动和 `hy-smi` 对应驱动和 `nvidia-smi`，DTK 对应 CUDA Toolkit（装在 `/opt/dtk`，版本从 `.info/rocm_version` 读，CI 里检查 26.04 和 26.10 两条发布线），PyTorch 的 DCU 版沿用 `cuda` 设备名，DAS / OpenDAS 这一层提供注意力、MoE 通信、量化算子等组件（插件的容器脚本里列了十几个），

引擎用 vLLM 加上开源插件。卡内的两个硬件常数从源码里得到确认：波前是 64 条通道（而且被用作同步语义的依据），矩阵核心指令是 MMAC，形状 16×16×16、BF16 输入 FP32 累加，按 `gfx928` / `gfx936` / `gfx938` 分支调用不同的内建函数。

下一讲走第二层：手上一份 CUDA 代码，怎么迁到 DCU 上；为什么迁完之后 `torch.cuda` 还能用；以及驱动、DTK、PyTorch、引擎、插件这五层的版本该怎么对。

第149讲 从 CUDA 到 DCU：迁移与版本矩阵

136 编程模型与 CUDA 迁移　137 框架怎么接入新硬件　147 推理引擎的硬件插件　148 海光 DCU 的软硬件栈

接着上一讲往下一层。手上有一份 CUDA 代码，或者一个装满 `torch.cuda` 调用的推理引擎，怎么搬到 DCU 上？搬完之后为什么 `torch.cuda.is_available()` 还返回真？以及一个更现实的问题：驱动、DTK、PyTorch、引擎、插件、算子库这六层的版本，怎么才能对上。

代码迁移：对应第 136 讲的三类

DCU 走的是兼容路线，所以第 136 讲那张逐行表直接适用：

一段 CUDA kernel 逐行翻译

⇆ 对照

全部 16 行 | 只看要人工处理的

`#include <cuda_runtime.h>`	`#include <hip/hip_runtime.h>`	自动
`#include <cub/cub.cuh>`	`#include <hipcub/hipcub.hpp>`	自动
`cudaMalloc(&buf, n * 4);`	`hipMalloc(&buf, n * 4);`	自动
`cudaMemcpyAsync(d, h, n, cudaMemcpyHostToDevice, s);`	`hipMemcpyAsync(d, h, n, hipMemcpyHostToDevice, s);`	自动
`int lane = threadIdx.x % 32;`	`int lane = threadIdx.x % 32;`	要确认
`for (int i = 16; i > 0; i >>= 1)` `v += __shfl_xor_sync(0xffffffff, v, i);`	`for (int i = 32; i > 0; i >>= 1)` `v += __shfl_xor(v, i);`	要确认
`unsigned mask = __ballot_sync(0xffffffff, pred);`	`uint64_t mask = __ballot(pred);`	要确认
`__syncwarp();`	`// 波前内天然同步，按架构决定是否需要`	要确认
`asm volatile("mov.u32 %0, %%laneid;" : "=r"(id));`	`// PTX 内联汇编：整段重写`	重写
`wmma::fragment<...> a_frag;` `wmma::mma_sync(c, a, b, c);`	`__builtin_amdgcn_mmac_f32_16x16x16bf16(...)`	重写
`kernel<<<grid, 256, smem, stream>>>(p);`	`hipLaunchKernelGGL(kernel, grid, 256, smem, stream, p);`	自动
`cublasGemmEx(h, ..., CUDA_R_16BF, ...);`	`hipblasGemmEx(h, ..., HIPBLAS_R_16B, ...);`	要确认
`cudnnConvolutionForward(...);`	`// 查目标平台有没有对应算子，或改用其他实现`	重写
`ncclAllReduce(sb, rb, n, ncclBfloat16, ncclSum, comm, s);`	`rcclAllReduce(sb, rb, n, rcclBfloat16, rcclSum, comm, s);`	自动

`cudaFuncSetAttribute(kernel, cudaFuncAttributeMaxDynamicSharedMemorySize, 128*1024);`	`hipFuncSetAttribute(...) // 上限按本卡容量改`	要确认
`nvcc -arch=sm_90 ...`	`编译器 --offload-arch=gfx938 ...`	要确认

要确认（名字能换，语义要人工核对）：工具不会改这个 32，因为它只是个整数。但这块卡的波前是 64 条通道，写死 32 会把一个波前当成两个 warp 用：同步的粒度、归约的范围全错。应改成 warpSize 或按架构的常量。

这 16 行里：自动 6 行、要人工确认 7 行、必须重写 3 行。真实项目里自动的比例高得多，但剩下那些行往往集中在最关键的 kernel 里。

图 149-1　CUDA 代码迁到 HIP 的三类处理（第 136 讲的表）。对 DCU 来说，第 5、6、7 行尤其要注意——它的波前是 64 条通道，而不是 32。

对 DCU 这块卡，三类里各挑最该留意的：

自动替换的部分就是运行时 API 和库名。实践中跑一遍 hipify 类工具，九成的行就过去了（第 136 讲在插件源码里数过：`hip` 前缀的运行时调用 36 处，而沿用 CUDA 名字的 47 处）。

要人工确认的头号项是**波前 64**。第 148 讲从源码里确认了这个常数。所有 `% 32`、`0xffffffff` 的掩码、从 16 开始折半的归约循环都要改。这类错误不报错、不崩溃，只是结果偏小（第 139 讲的那类静默错误）。

必须重写的是内联汇编和矩阵核心指令。DCU 的矩阵指令是 MMAC，而且**同一个形状在不同代上的内建函数名不同**：`gfx928` / `gfx936` 用 `__builtin_amdgcn_mmac_f32_16x16x16bf16`，`gfx938` 用 `__builtin_hcu_mmac_f32_16x16x16_bf16_lit_lts`（第 148 讲引过这段）。所以“迁到 DCU”这件事，在 kernel 层面还要再按代分支一次。

构建这一侧也有对应关系：`nvcc -arch=sm_90` 变成 DCU 编译器加 `--offload-arch=gfx938` 这样的参数；`CUDA_HOME` 的角色由 `ROCM_PATH` 承担，默认指向 `/opt/dtk`。插件的 `setup.py` 里能看到这套东西的真实样子：注册扩展用的仍是 PyTorch 的 `CUDAExtension`，而编译时 PyTorch 会把 HIP 平台的宏塞进每条编译命令（源码注释里专门提到它会把这些宏从构建日志里隐去）。

为什么 torch.cuda 照样能用

这是 DCU 适配里最省事的一环，也最容易让人困惑：机器里没有 NVIDIA 的卡，但 `torch.cuda.is_available()` 返回真，`tensor.cuda()` 能用，`torch.cuda.synchronize()` 能用。

原因在第 137 讲讲过：PyTorch 的 DCU 版沿用 ROCm 的做法，把整个后端用 HIP 重新编译，**注册到 CUDA 这个分发键上**。于是：

- 分发器查表时，设备类型是 `cuda`，查的是 CUDA 那一行，而那一行里注册的是为 DCU 编译出的实现；
- Python 侧的 `torch.cuda` 命名空间照旧；
- C++ 侧 `at::cuda::getCurrentCUDAStream()` 这类接口照旧；
- 环境变量用 `CUDA_VISIBLE_DEVICES`。

插件的平台类把这些事实写成了六行声明（`vllm_hcu/platforms/hcu.py`，提交 81cbe8f）：

```
_enum = PlatformEnum.ROCM
device_name: str = "hip"
device_type: str = "cuda"
dispatch_key: str = "CUDA"
ray_device_key: str = "GPU"
dist_backend: str = "nccl"
device_control_env_var: str = "CUDA_VISIBLE_DEVICES"
```

这段声明的每一行都在替引擎回答一个问题：PyTorch 侧的设备名用什么（`cuda`）、算子分发走哪个键（`CUDA`）、日志里怎么称呼这块卡（`hip`）、Ray 调度时资源叫什么（`GPU`）、分布式后端名字写什么（`nccl`，底下是 RCCL）、限制可见设备用哪个环境变量。`_enum` 取 `PlatformEnum.ROCM` 还有一个额外效果：引擎里凡是“如果是 ROCm 平台就……”的既有分支自动生效，插件白拿这些处理。

要点

这套做法的收益用第 134 讲的话说就是：**上面四层几乎不改**。模型文件不改，引擎的 Python 代码不改，PyTorch 的接口不改，第三方库里写死 `torch.cuda` 的地方也能用。代价是名字和实际硬件不符——报错信息里写着 CUDA，环境变量叫 `CUDA_VISIBLE_DEVICES`，新人上手时要先接受这件事。

还有一处要留意：`device_count` 不能直接用 `torch.cuda.device_count()`。插件自己实现了一个无状态的版本（`_rocm_device_count_stateless`），因为它要在 `CUDA_VISIBLE_DEVICES` 被改写之后仍然给出正确的卡数——这类“看起来一样、细节不一样”的地方，正是插件存在的理由。

版本矩阵：六层都要对

迁移之外，DCU 适配中最耗时间的其实是版本。六层各有自己的节奏，而它们必须成套。



图 149-2　六层版本的检查规则（按插件源码里的检查逻辑整理）。逐层换版本，看哪一层先被拒绝、报什么问题。版本号是示例，实际以官方文档和仓库里的声明为准。

从源码里能读到的真实规则（提交 81cbe8f）：

插件 ↔ 引擎。 `vllm_hcu/compatibility.py` 在任何补丁注册之前检查已安装的 vLLM：读发行包的元数据（**不导入 vLLM**，避免副作用），用 PEP 440 的规则解析版本号，要求 epoch 和 `(major, minor, patch)` 与 `__vllm_target_version__` 完全一致，允许预发布、开发版、后

发布和本地构建后缀不同。不通过就抛错，而且这个失败会被**锁存**——第二次调用插件入口时直接把同一个错误再抛一遍，不会让一部分补丁打上、另一部分没打。

版本号里带着来源。`vllm_hcu/version.py` 里的目标版本串是 `0.28.1rc1.dev491+g462fdb097.das.462fdb0.dtk2604`，另外两个常量分别记着上游 vLLM 的提交和 DAS 分支的提交。这种写法把“这份插件是对着哪一次上游提交审过的”固定在了代码里，比写在文档里可靠。

环境检查分三种严格程度。CI 的环境配置（`.github/workflows/configs/hcu-runner-environment.json`）里，Python 版本用 `exact`（只接受 3.10.12 或 3.12.12），`torch` 和 `torch.hip` 用 `prefix`（`2.11.0+`、`6.3.` 或 `7.2.`），DTK 用 `release_line`（26.04、26.10 两条线，线内的补丁号和 `-` / `+` 后缀都接受）。**不同层的容忍度不同**，这是做版本检查时值得抄的设计。

算子库按标签从厂商归档装。容器脚本里每个组件都用 `das-install <包名> torch<版本标签>` 装（第 148 讲列过那张清单），注释里说明了为什么不硬编码完整包名：换 DTK 或 torch 版本时不用改构建脚本。还有一句很具体的提醒——`tilelang` 要放在最后装，因为安装真包会替换掉前面为了满足依赖写进去的占位包。

自检工具。插件带一个 `vllm-hcu-doctor` 命令，只读元数据和源码、做只读诊断；加 `--no-arm --json` 时连补丁都不注册。文档里同时写着：两种模式都**不能代替**模型精度或设备执行的验证。

> **注意**
> 版本检查通过不等于能跑对。插件的 README 里把这件事说得很直接：通过版本关卡不代表 PyTorch、DTK 和算子的二进制接口已经验证兼容，也不保证任意上游 vLLM 的 wheel 都能用；模型已注册或存在配置，不代表精度验证已经通过。换句话说，**版本矩阵挡掉的是“肯定不行”的组合，而不是保证“一定行”**。第 153 讲的清单里，精度和性能的验收是单独的两关。

从零到跑起来的顺序

把这一讲和上一讲的内容拼成一条实际路径：

步	做什么	怎么确认这一步过了
1	装驱动，`hy-smi` 能看到所有卡	卡数、显存、温度正常
2	装与驱动配套的 DTK，设 `ROCM_PATH`	编译器能编一个最小 kernel；`.info/rocm_version` 的值在支持的发布线上
3	装 PyTorch 的 DCU 版	`torch.cuda.is_available()` 为真，`torch.version.hip` 的前缀对；跑一次矩阵乘
4	装引擎和算子库（用厂商归档的版本）	import 不报符号错误
5	装插件，跑自检	`vllm-hcu-doctor` 全绿；版本检查通过
6	起一个小模型的服务	能生成 token（第 153 讲起才是精度和性能的验收）

第 2 步和第 3 步的配对关系最容易出错：PyTorch 的构建里写着它编译时用的 HIP 版本，和 DTK 必须成套。症状是 import torch 就报符号找不到，或者能 import 但一调 kernel 就崩。

回顾

迁到 DCU 的代码工作就是第 136 讲那三类：运行时 API 和库名机械替换；波前 64 相关的写法要人工确认（掩码、归约步数、块内分组）；内联汇编和矩阵指令必须重写，而且 MMAC 的内建函数名按 `gfx928` / `gfx936` / `gfx938` 还要再分支。构建侧 `ROCM_PATH` 取代 `CUDA_HOME`、架构代号从 `sm_90` 变成 `gfx938` 这类串。PyTorch 的 DCU 版把实现注册在 CUDA 分发键上，所以 `torch.cuda` 整套接口照样能用；插件的平台类用六行声明把这件事告诉引擎（`device_type="cuda"`、`dispatch_key="CUDA"`、`dist_backend="nccl"`、`_enum=PlatformEnum.ROCM` 等）。版本上六层要成套，插件在打任何补丁之前先只读元数据检查引擎版本（主次补丁号一致、预发布后缀放宽），CI 对 Python 用精确匹配、对 torch 用前缀、对 DTK 用发布线匹配，算子库一律从厂商归档按标签安装。

版本对上、服务起来之后，就该看插件内部了。下一讲打开 `vllm-plugin-das`：平台类怎么声明自己、运行时补丁怎么做到不改上游一行代码、以及那道版本关卡在代码里长什么样。

第150讲 读 vllm-plugin-das：平台与运行时补丁

129 vLLM 的架构
147 推理引擎的硬件插件
148 海光 DCU 的软硬件栈
149 从 CUDA 到 DCU：迁移与版本矩阵

第 147 讲讲了插件机制该长什么样，这一讲读一份真的。

> **注意**
> 读的是 `miniinfer/third_party/vllm-plugin-das`，提交 `81cbe8f`，Apache-2.0 许可，**只读不运行**。行号按这个提交。这个项目跟着 vLLM 的版本走，几个月后函数名和行号都可能变，但下面讲的几个结构变得很慢。

取下同样的代码：

```
cd miniinfer/third_party
git clone https://github.com/HYGON-AI/vllm-plugin-das && cd vllm-plugin-das && git checkout 81cbe8f
```

先看规模和结构：

```
$ .venv/bin/python bench/das_loc.py                                    ⇆ 对照

vllm-plugin-das（提交 81cbe8f）：818 个源文件（.py .cu .cuh .cpp .h），255134 行（不算空行）
   115629  tests
    41325  vllm_hcu/model_executor
    36330  vllm_hcu/patch
    23517  vllm_hcu/v1
    15804  vllm_hcu/models
     7580  vllm_hcu/distributed
     3037  vllm_hcu/ops
     3013  tools
     2371  vllm_hcu/csrc
     1684  vllm_hcu/platforms
     1553  vllm_hcu/runtime_compat
      944  vllm_hcu
      891  examples
      634  vllm_hcu/utils
setup.py 里登记的入口：
  vllm.platform_plugins   hcu = vllm_hcu:hcu_platform_plugin
  vllm.general_plugins    hcu_model = vllm_hcu:hcu_platform_register_model
  vllm.general_plugins    hcu_ops = vllm_hcu:hcu_platform_register_ops
运行时补丁文件（vllm_hcu/patch/ 下以 patch_ 开头的文件）：platform/core_fix 24 个，
platform/framework_opt 21 个，worker/core_fix 42 个，worker/framework_opt 35 个，
worker/op_opt 73 个，共 195 个
整模块替换（module_exchange.py）：16 对，例如
  vllm.model_executor.layers.fused_moe.modular_kernel
    → vllm_hcu.model_executor.layers.fused_moe.modular_kernel
  vllm.model_executor.layers.linear
    → vllm_hcu.model_executor.layers.linear
  vllm.v1.attention.backends.mla.sparse_swa
    → vllm_hcu.v1.attention.backends.mla.sparse_swa
HCUPlatform.device_name = "hip"
HCUPlatform.device_type = "cuda"
HCUPlatform.dispatch_key = "CUDA"
HCUPlatform.dist_backend = "nccl"
HCUPlatform.device_control_env_var = "CUDA_VISIBLE_DEVICES"
supported_quantization：启用 23 种，注释掉 4 种（mxfp4、bitsandbytes、modelopt_fp4、
gpt_oss_mxfp4）
源码里出现的架构代号：gfx928（2 处）、gfx936（10 处）、gfx938（87 处）、gfx950（6 处）
把线程组宽度写成 64 的地方：3 处
```

图 150-1　插件的规模、入口、补丁清单和平台声明（只读源码文本统计，提交 81cbe8f）。

25 万行里，和“怎么接进引擎”直接相关的只有三处：三个 entry point（`vllm_hcu/__init__.py`，138 行）、平台类（`platforms/hcu.py`，1033 行）、补丁机制（`patch/`，195 个补丁文件加几个基础模块）。这一讲读这三处，第 151、152 讲读算子和后端。

三个入口，一条纪律：轻

`setup.py` 里登记的三个入口（上面统计的第二段）对应第 147 讲说的两类：一个平台插件（`hcu = vllm_hcu:hcu_platform_plugin`）、两个通用插件（注册模型、注册算子）。

third_party/vllm-plugin-das/vllm_hcu/__init__.py 第 14–61 行 / 共 138 行 ⇆ 对照

```
14 from __future__ import annotations
15
16 import importlib
17 import threading
18
19 from vllm_hcu.lightop_env import configure_lightop_environment
20
21 configure_lightop_environment()
22
23 from vllm_hcu.compatibility import (
```

1 导入要足够轻 第 14–26 行

模块顶层只做两件事：配置算子库的环境变量，导入兼容性检查。注意这里**没有 import torch，也没有 import vllm**。

原因在文件开头的说明里：插件发现阶段只能“布置好回调”，不能把实现拽进来。如果这时候把 torch 和 vLLM 都导入了，后面的“整模块替换”就来不及安排——目标模块已经先被加载了（第 147 讲说的那条纪律）。

2 **保住进程角色** 第 41–54 行

3 **两个阶段** 第 57–61 行

← 上一步 | 看整个文件 | 下一步 →

入口函数本身只有十几行，但处理了一个很具体的现实问题：

third_party/vllm-plugin-das/vllm_hcu/__init__.py 第 64–98 行 / 共 138 行 ⇆ 对照

```
64 def _ensure_platform_plugin_ready() -> None:
65     """Fail closed after a platform-initialization error was latched."""
66
67     if _PLATFORM_INIT_FAILURE is None:
68         return
69     error = _PLATFORM_INIT_FAILURE
70     if isinstance(error, VllmCompatibilityError):
71         raise VllmCompatibilityError(
72             "HCU platform compatibility initialization previously failed: "
73             f"{error}"
```

1 **失败要锁存** 第 64–78 行

如果平台初始化失败过一次，后面任何一次调用都直接把同一个错误抛出来，而不是重试。理由是补丁可能只打了一半——重试会在一个半残的注册表上继续操作，产生更难查的问题。第 147 讲说的“失败锁存”在代码里就是这几行。

2 **为什么失败时还返回类路径** 第 81–98 行

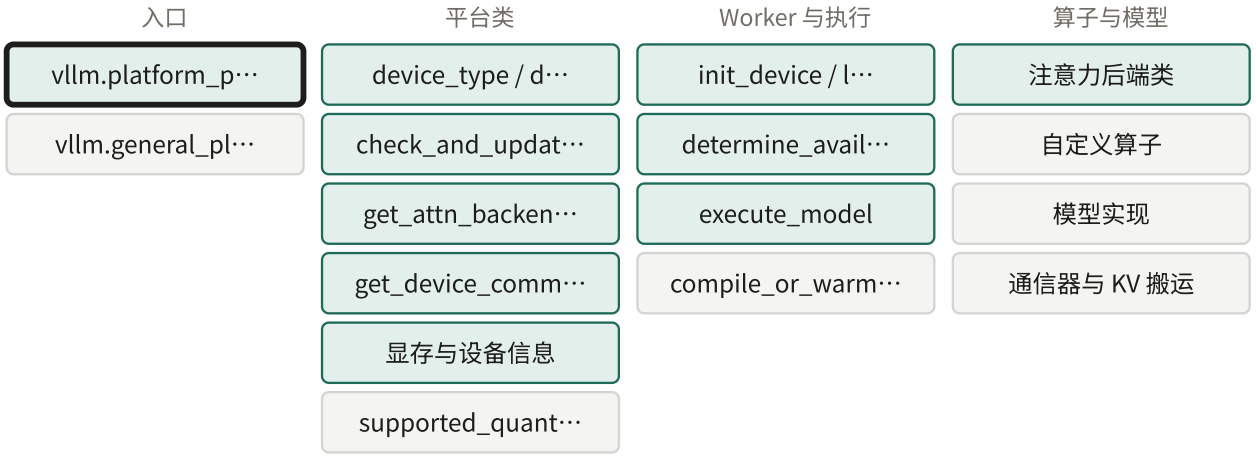
← 上一步 | 看整个文件 | 下一步 →

平台类：七行声明替引擎回答七个问题

一个硬件插件要实现哪些接口

⇆ 对照

全部　只看必须实现的



绿底 = 不实现引擎就起不来　白底 = 可选（不实现就退回引擎的通用路径，通常更慢）

入口 · `vllm.platform_plugins`（必须实现）：一个函数，返回平台类的全限定名（这块卡不可用时返回 `None`）。引擎启动时按 `entry point` 发现它，所有进程都会加载一次。这是整个插件唯一的“注册”动作——不改引擎一行代码。

图 150-2　插件要实现的接口清单（第 147 讲的图）。海光的实现里，绿底那些在 platforms/hcu.py 里；点任意一项看它的职责。

third_party/vllm-plugin-das/vllm_hcu/platforms/hcu.py 第 231–244 行 / 共 1033 行 ⇆ 对照

```
231 class HCUPlatform(Platform):
232     #这个地方会管理custom_ops
233     _enum = PlatformEnum.ROCM
234     device_name: str = "hip"
235     device_type: str = "cuda"
236     dispatch_key: str = "CUDA"
237     ray_device_key: str = "GPU"
238     dist_backend: str = "nccl"
239     device_control_env_var: str = "CUDA_VISIBLE_DEVICES"
```

1 设备身份 第 231–239 行

七行声明，第 149 讲逐行解释过：平台枚举取 ROCM（白拿引擎里所有 ROCm 分支）、PyTorch 侧设备类型是 `cuda`、分发键是 `CUDA`、日志里叫 `hip`、Ray 的资源名是 `GPU`、分布式后端写 `nccl`（底下是 RCCL）、限制可见设备用 `CUDA_VISIBLE_DEVICES`。

第 232 行还留着一句中文注释（“这个地方会管理 custom_ops”）——真实项目的代码不会处处精致，读源码时要习惯这种痕迹。

2 Ray 的三个环境变量 第 240–244 行

← 上一步 | 看整个文件 | 下一步 →

平台类的其余部分回答的是第 147 讲清单里的问题，挑几个和前面讲过的内容对得上的：

方法	它回答什么	对应的讲
`get_attn_backend_cls` / `get_valid_backends`	用哪个注意力后端（一组候选 + 逐个校验 + 回退）	143、152
`get_device_communicator_cls`	用哪个通信器	146、152
`use_custom_allreduce` / `is_fully_connected`	要不要开自定义 All-Reduce，拓扑是不是全连接	146
`supported_quantization` / `verify_quantization`	支持哪些量化方法，启动时怎么校验和改写	145、152
`check_and_update_config` / `apply_config_platform_defaults`	按硬件改配置，指定 worker 类，调自定义算子开关	147
`update_block_size_for_backend`	把 KV 块大小调到注意力后端支持的值	143
`supports_fp8`	这一代卡支不支持 FP8（实现里是“是不是 gfx938”）	145、152
`get_static_graph_wrapper_cls`	图捕获用哪个包装器（直接复用 vLLM 的 CUDA Graph 包装器）	138
`get_current_memory_usage` / `get_device_total_memory`	量显存，决定 KV 块数	129

`supports_fp8` 的实现只有一行：返回这块卡是不是 `gfx938`。这是第 145 讲那张“硬件 × 精度”表在代码里的样子——**精度支持是按架构代号判断的，不是按型号名字**。

运行时补丁：精确、幂等、失败锁存

插件的难点不在平台类，而在“怎么改上游的行为而不改上游的代码”。`patch/` 这 3.6 万行就是干这件事的。机制分三样。

整模块替换。当上游的某个模块被导入时，用插件里的同名实现替换它。统计里有 16 对，例如上游的 `vllm.model_executor.layers.linear` 换成 `vllm_hcu.model_executor.layers.linear`。文档里还给了使用约定：**调用方应该照常写 `vllm.*` 的导入路径**，由协调器在上游模块加载前解析到插件的实现——这样插件代码和上游代码长得一样，升级时好比对。

导入后回调。模块正常加载，加载完成后立刻调用插件的一个函数，由它替换模块里的某个函数或类方法。195 个补丁文件大多是这一类。

配置改写。平台类的 `check_and_update_config` 里直接改配置，不碰代码（第 147 讲说的第一选择）。

替换的入口是一个装在 `sys.meta_path` 上的查找器：

third_party/vllm-plugin-das/vllm_hcu/patch/import_coordinator.py

⇆ 对照

第 334–416 行 / 共 822 行

```
334     def find_spec(
335         self,
336         fullname: str,
337         path: object = None,
338         target: ModuleType | None = None,
339     ) -> importlib.machinery.ModuleSpec | None:
340         # Import machinery calls this concurrently for unrelated modules.  The
341         # exact dictionary lookup ensures ancestors and descendants are ignored.
342         with self._lock:
```

1 只拦精确的模块名 第 340–345 行

第一件事就是精确查表：注册的是完整模块名，祖先和子孙模块一律不管。开头的文件说明里强调了这一点——旧的做法会替换 `builtins.__import__` 并扫描祖先名字，这个版本不这么干。

为什么重要？按前缀匹配会误伤无关模块，而且会在一个还没完成初始化的模块上触发回调，产生极难复现的问题。

2 **已经打过补丁的模块禁止重载** 第 347–365 行

3 **正在打补丁时禁止重入导入** 第 367–381 行

4 **失败过的补丁：拒绝导入，不悄悄回退** 第 383–396 行

5 **最后才是真正的替换或挂回调** 第 398–416 行

← 上一步 | 看整个文件 | 下一步 →

补丁的**顺序**也是显式写出来的。平台阶段的分发器里有一个按顺序排列的适配器清单，注释解释了为什么顺序重要：如果某些目标模块在插件发现之前就被导入了，协调器会**按清单顺序立即应用**这些回调；而 KV 相关的适配必须排在 MoE 协调器包装之前，否则基类构造链看到的缓存尺寸就不对（`vllm_hcu/patch/platform/__init__.py` 的 `_ORDERED_FRAMEWORK_ADAPTERS`）。

这段代码还有一处值得学的做法：注册这一批替换时用了一个“注册批次”的上下文（`registration_batch`），注释说明了意图——第三方的导入只能发生在这一批开始之前或完整可见之后，**不能观察到中间状态**。

版本关卡在代码里的样子

第 149 讲讲过规则，这里看实现：

third_party/vllm-plugin-das/vllm_hcu/compatibility.py 第 74–158 行 / 共 166 行 ⇆ 对照

```
74 def inspect_vllm_compatibility() -> VllmCompatibility:
75     """Inspect the installed vLLM metadata without importing vLLM itself."""
76
77     # Keep platform-plugin discovery importable under ``python -S``.  The
78     # platform probe latches a missing runtime dependency before any patch
79     # mutation, while normal compatibility checks still use packaging's full
80     # PEP 440 parser.
81     from packaging.version import InvalidVersion, Version
82
83     try:
```

1 只读元数据，不导入 vLLM 第 74–83 行

函数的文档字符串就说明了意图：检查已安装的 vLLM 元数据，而不导入它。导入会触发一长串副作用（注册算子、初始化平台），而这些副作用恰恰是我们想在检查通过之后才允许发生的。

PEP 440 的解析器在函数内部才导入，注释给的理由是让插件发现阶段在 `python -S` 这类受限环境下也能 import。

2 **vLLM 没装：返回“不兼容”而不是抛错** 第 90–102 行

3 **匹配到主次补丁这一级** 第 123–136 行

4 **关卡：不兼容就抛错** 第 150–158 行

← 上一步 | 看整个文件 | 下一步 →

这份代码里可以带走的四条

一、入口要轻。插件发现阶段不 import torch、不 import 引擎，否则后面的替换来不及安排。

二、**失败要响**。版本不符、补丁失败，都要明确失败并保留诊断状态，绝不静默回退到上游实现。半打补丁的状态比起不来更糟。

三、**替换要精确、幂等、可报告**。精确匹配模块名，每个补丁只应用一次并记录状态，提供一个报告接口（这个项目里是 `patch_report()`，能列出进程角色、目标符号、补丁状态、失败详情和特性开关）。

四、**把上游的真实行为写进注释**。那段“平台插件会被调用两次”的注释，价值比代码本身还高——它是别人调试了很久才得到的知识。

> 专栏
>
> 对比一下第 16 篇的 nano-vllm：1195 行，没有任何这类机制，因为它只在一种硬件上跑、也不需要跟着别人的版本走。这套补丁机制的全部复杂度，都来自两个约束：**不能改上游**，和**上游在动**。做任何长期维护的适配层，这两条都会出现。

回顾

海光插件的接入面很小：三个 entry point、一个平台类、一套补丁机制。入口模块只有 138 行，顶层不导入 torch 和 vLLM，先布置好替换再让实现加载；入口函数会把失败锁存，而且在第一次调用时即使检查失败也返回类路径——因为被审计的那版 vLLM 会把平台插件调用两次，第一次在一个吞异常的探测里。平台类用七行声明回答设备身份（ROCM 枚举、`cuda` 设备、`CUDA` 分发键、`nccl` 后端），其余方法回答注意力后端、通信器、量化名单、显存、块大小、图包装器等问题，`supports_fp8` 的实现就是判断架构代号是不是 `gfx938`。补丁机制的四道守卫是这一讲的重点：只拦精确模块名、已打补丁的模块禁止重载、正在打补丁时禁止重入导入、失败过的补丁拒绝导入而不是静默回退；顺序和注册批次都显式写出来。版本关卡只读元数据、不导入 vLLM，匹配到主次补丁这一级，放宽预发布和本地构建后缀。

下一讲钻进算子：这块卡上 decode 的矩阵乘为什么要自己写 kernel，那个 $M \leqslant 64$ 的 kernel 是怎么写的。

第151讲 DCU 上的矩阵乘：hipBLASLt 与小 M kernel

87 矩阵乘法 Kernel 与 Tensor Core　135 国产与 NVIDIA 硬件差异　143 矩阵乘法与注意力的适配　150 读 vllm-plugin-das：平台与运行时补丁

第 143 讲用海光插件注释里的数字算过一笔账：decode 时 M ⩽ 64，`hipBLASLt` 选 256 × 256 的宏块，lm_head 那个形状上实际带宽只有 0.93 TB/s，而只沿 N 分块的专用 kernel 能跑到 1.73 TB/s。这一讲把那个专用 kernel 打开，逐行看它是怎么写的。

这是全篇最硬的一讲，但它值得读完：一个真实的、为具体硬件写的矩阵乘 kernel，把前面讲过的所有硬件常数（波前 64、MMAC 的 16×16×16、片上存储、对齐要求、架构分支）全用上了。

注意

读的是 `third_party/vllm-plugin-das/vllm_hcu/utils/bf16_fp32_mmac.py`，提交 `81cbe8f`，**只读不运行**。性能数字只引用文件开头的注释。

这个 kernel 解决什么形状

third_party/vllm-plugin-das/vllm_hcu/utils/bf16_fp32_mmac.py 第 1–23 行 / 共 354 行 ⇆ 对照

```
1  # SPDX-License-Identifier: Apache-2.0
2  # SPDX-FileCopyrightText: Copyright (c) 2026 Hygon Information Technology Co., Ltd.
3  """Small-M BF16xBF16 → FP32 NT GEMM on HCU MMAC.
4
5  Ported from sglang-das ``deepseek_v4/linear_bf16_fp32.cuh``. The DeepSeek V4
6  compressor projections run with M ⩽ 64 tokens per rank during decode, where
7  hipBLASLt picks a 256x256 tile and takes ~135us regardless of N. These kernels
8  cover M in [1, 64], N in {256, 512, 1024, 2048} and K % 128 == 0.
9  """
```

1 **问题陈述** 第 3–9 行

注释写得像一份工单：这段代码从 SGLang 的 DCU 分支移植过来；DeepSeek V4 的压缩投影在 decode 时每卡 M ⩽ 64；在这个区间 `hipBLASLt` 选 256×256 的分块，**不管 N 多大都要约 135 µs**；所以这些 kernel 覆盖 M ∈ [1, 64]、N ∈ {256, 512, 1024, 2048}、K 是 128 的倍数。

“不管 N 多大”这半句是整件事的关键（第 143 讲算过）：时间和工作量脱钩了，说明代价全在分块的固定开销上。一个为大矩阵调好的库，在小 M 上会退化成这样。

2 **把约束写成常量** 第 18–23 行

← 上一步 | 看整个文件 | 下一步 →

架构分支：同一个形状，三种写法

third_party/vllm-plugin-das/vllm_hcu/utils/bf16_fp32_mmac.py 第 31–80 行 / 共 354 ⇆ 对照

```
31
32 typedef __hip_bfloat16 bf16_t;
33 typedef _Float16 hcu_half4_t __attribute__((__vector_size__(8)));
34 typedef short hcu_bf16_mfma_t __attribute__((__vector_size__(8)));
35 typedef float hcu_float4_t __attribute__((__vector_size__(16)));
36 typedef std::uint32_t hcu_uint32x4_t __attribute__((ext_vector_type(4)));
37
38 struct HcuBf16x8 {
39   hcu_half4_t data[2];
```

1 **向量类型** 第 31-40 行

先定义几个向量类型：4 个半精度打包成一个 8 字节的向量、8 个 BF16 作为矩阵指令的输入、4 个 FP32 作为累加器、4 个 32 位整数作为缓冲描述符。

为什么累加器正好是 4 个 FP32？第 135 讲算过：16×16 的输出有 256 个元素，摊到 64 条通道上，每条通道 4 个。这个数字不是选的，是硬件定的。

2 **MMAC：按架构选内建函数** 第 42-57 行

3 **手工拼一个缓冲描述符** 第 59-70 行

4 **异步搬运：显存直接进片上存储** 第 72-80 行

← 上一步 | 看整个文件 | 下一步 →

主 kernel：一块 16 行，波前沿 K 切

third_party/vllm-plugin-das/vllm_hcu/utils/bf16_fp32_mmac.py 第 82–141 行 / 共 35 ⇆ 对照

```
82  // One 16-row x kNPerBlock-col tile per block; waves split K and reduce in LDS.
83  template <int kWarps, int kNPerBlock>
84  __global__ void gemm_nt_bf16_fp32(
85      const bf16_t* __restrict__ a, const bf16_t* __restrict__ b,
86      float* __restrict__ out, std::int32_t m, std::int32_t n, std::int32_t k) {
87    constexpr std::int32_t kWaveSize = 64;
88    const std::int32_t block_x = blockIdx.x;
89    const std::int32_t block_y = blockIdx.y;
90    const std::int32_t thread = threadIdx.x;
```

1 **分块策略：一句话** 第 82-87 行

注释把策略说完了：**一个线程块负责 16 行 × kNPerBlock 列的一块，块里的几个波前沿 K 方向分工，最后在共享内存里归约。**

对照第 143 讲的结论：M 方向不补零（16 行正好是矩阵指令的行数），并行度靠 N 方向的块数和 K 方向的波前数凑出来。这正是“只沿 N 分块 + 沿 K 切波前”的实现。

2 **每条通道负责什么** 第 88-97 行

3 **主循环：读 8 个、算两条 MMAC** 第 99-112 行

4 **波前之间折半归约** 第 114-132 行

5 **写回：一条通道写 4 个数** 第 134-140 行

← 上一步 | 看整个文件 | 下一步 →

⇆ 对照

线程组宽度和片上缓冲怎么限制分块

线程组宽度 w： 32（warp） 64（波前）

矩阵指令： 16×8×16 16×16×16 32×32×16

M（这一步的 token 数）= **16**

输出块 Mt × Nt = **32 × 128**

K 步长 = **64**

片上存储 = **160** KB

输出块 32 × 128，每格是一条 16×8 的矩阵指令（深色 = 这一步真的有 token，浅色 = 补零）



一条指令 16×8×16
一个线程组 32 条通道
每通道 4 个累加器（FP32）
一块要 32 条指令覆盖
一块的 A+B 分块 20 KB

M = 16，输出块 32 行：这一步**只有 16 行是真的**，块里 50% 的行在算补的零（M = 16 对指令行数 16 来说正好对齐）。
每条通道持有 4 个 FP32 累加器：指令形状不变、通道数从 32 变 64，每通道的累加器就少一半，寄存器压力减半，但要凑满一条指令需要的数据翻倍。
片上存储 160 KB ÷ 一块 20 KB：单缓冲能放 **8** 块，双缓冲（一边搬一边算）能放 **4** 块。

图 151-1　把演示的参数调成这个 kernel 的设定：线程组宽度 64、矩阵指令 16×16×16、M 拖到 16 或 64、输出块选 16 × 64，再对比 256 × 256 时的补零比例（第 135、143 讲）。

一个被关掉的优化，和它的注释

源码里还有一个 split-K 的变体（第 87 讲的 split-K：沿 K 切开凑并行度），用双缓冲、异步搬运、原子加归约。但它被一个宏关掉了，注释写明了原因：

```
// The current DTK toolchain emits an empty gfx938 code object when the split-K raw
// LDS load path is present in this JIT translation unit. Keep it out of device
// compilation until it can be built as an independent extension.
#define VLLM_HCU_ENABLE_MMAC_SPLITK 0
```

翻译过来：当前的 DTK 工具链在这个即时编译单元里遇到 split-K 那条“直接搬进共享内存”的路径时，会给 `gfx938` 生成一个空的代码对象；所以先把它从设备编译里排除，等能作为独立扩展构建时再说。

这条注释值得单独说，因为它展示了适配工作里最真实的一面：**一个写好的优化，因为工具链的一个问题而暂时关掉**。处理方式有三点可取——用宏整段关闭而不是删掉代码；注释里写清症状（空的代码对象）、条件（在这个即时编译单元里）和重新启用的条件（作为独立扩展构建）；以及关掉之后主路径仍然完整可用。第 142 讲说过“把判断过程写下来”，这就是一个范例。

入口：检查、路由、即时编译

third_party/vllm-plugin-das/vllm_hcu/utils/bf16_fp32_mmac.py 第 242–300 行 / 共 3 ⇆ 对照

```
242 torch::Tensor linear_bf16_fp32_mmac(torch::Tensor x, torch::Tensor w) {
243   TORCH_CHECK(x.dim() == 2 && w.dim() == 2, "x and w must be 2-D");
244   TORCH_CHECK(x.is_cuda() && w.is_cuda(), "x and w must be CUDA tensors");
245   TORCH_CHECK(x.device() == w.device(), "x and w must be on the same device");
246   TORCH_CHECK(x.scalar_type() == torch::kBFloat16 &&
247                   w.scalar_type() == torch::kBFloat16,
248               "x and w must be BF16");
249   TORCH_CHECK(x.is_contiguous() && w.is_contiguous(),
250               "x and w must be contiguous");
```

1 **C++ 侧的检查** 第 243–257 行

进 kernel 之前把所有假设检查一遍：两个张量都是二维、都在设备上、同一个设备、都是 BF16、都连续、K 维对得上、M 在 [1, 64]、N 是那四个值之一、K 是 128 的倍数。

这些检查不是防御性编程的样板，而是**契约**：下面的 kernel 完全按这些前提写的（一次读 8 个元素、按 16 行分块、不处理非连续布局）。任何一条不满足，结果就是错的而不是慢的。

2 **准备输出和网格** 第 258–273 行

3 **发射** 第 285–294 行

← 上一步 | 看整个文件 | 下一步 →

Python 侧的两层把它接进引擎：

third_party/vllm-plugin-das/vllm_hcu/utils/bf16_fp32_mmac.py 第 303–354 行 / 共 3 ⇆ 对照

```
303 @functools.lru_cache(maxsize=1)
304 def _jit_module() -> Any:
305     import torch.utils.cpp_extension
306
307     return torch.utils.cpp_extension.load_inline(
308         name="vllm_hcu_linear_bf16_fp32_mmac",
309         cpp_sources="",
310         cuda_sources=_SOURCE,
311         extra_cflags=["-O3"],
```

1 **即时编译，只编一次** 第 303-314 行

用 PyTorch 的 `load_inline` 在运行时编译这段源码，外面套一个只缓存一个结果的装饰器，所以一个进程里只编一次。

第 136 讲提过这种做法的取舍：不用预编译所有架构，代价是第一次调用要等几十秒——在服务里必须放进预热阶段（第 153 讲的清单里有这一项）。

2 **问硬件要架构代号** 第 317-321 行

3 **Python 侧的 supported()** 第 323-347 行

4 **对外的函数** 第 350-354 行

← 上一步 | 看整个文件 | 下一步 →

调用方：三层回退

这个 kernel 不是直接被模型调用的，而是放在一个三层回退的链条里（`vllm_hcu/utils/bf16_fp32_gemm.py`）。顺序是：

1. **布局不对就走原生 torch**。如果权重按 `[in_features, out_features]` 存（插件的一个选项），直接用 `torch.mm` 并要求 FP32 输出。
2. **一长串前置条件**：环境变量没关、`torch.version.hip` 不是 None、两个张量都是 BF16、都在设备上、都是二维。任何一条不满足，记下原因、退回 `torch.mm`。
3. **小 M 走 MMAC kernel**：开关打开且 `supported()` 通过时调上面那个 kernel。注释里就写着 `Decode-sized M: hipBLASLt picks a 256x256 tile here (~135us/call)`。调用失败就把这条路**永久关掉**（一个进程内的标志位），记一条警告，退到 hipBLASLt。
4. **再失败退到 `torch.mm`**，同样是永久关掉加一条警告。

最后，整个函数通过 `direct_register_custom_op` 注册成一个自定义算子（`hcu_linear_bf16_fp32`），还配了一个只按形状返回空张量的假实现——第 137 讲讲过这个假实现是给图编译用的。

> **要点**
> 这条链条的设计值得抄：**每一层都有明确的前置条件和失败处理，失败只警告一次（而不是每步刷日志），失败之后永久降级（而不是每次都重试一个已知坏掉的路径）**。适配代码的质量，很大程度上就体现在这种“出错之后怎么办”的处理上。

另一个专用 kernel：lm_head

同一个思路还用在 lm_head 上（`vllm_hcu/model_executor/layers/skinny_lm_head.py`），但实现换成了 Triton。文件开头的注释给了第 143 讲算过的那组数：`M ≤ 64 时这个投影是纯粹的权重流；hipBLASLt 在这里选 256×256 的宏块、在 gfx938 上达到约 0.93 TB/s；这个 kernel 只沿 N 分块、M 方向一个块吃完，达到 1.4-1.7 TB/s（Step5 的 lm_head 分片 32224 × 4096：M=16 时 285 → 153 μs，M=64 时 190 μs）`。

两个 kernel 的对比正好印证第 144 讲的选择逻辑：

	压缩投影的 kernel	lm_head 的 kernel
语言	HIP C++（手写，带架构分支和内建指令）	Triton
形状约束	M ⩽ 64，N 取 256 / 512 / 1024 / 2048，K % 128 = 0	M ⩽ 64，K % 128 = 0，N 任意
分块	16 行 × 16 列，波前沿 K 切	按 M 分三档选 BM、BN 和 warp 数
开关	环境变量 + `supported()`	环境变量 `STEP5_SKINNY_LM_HEAD`
为什么这样选	要控制矩阵指令、搬运和归约的每一步	逻辑简单（纯流式读权重），Triton 够用且好维护

同一个问题（小 M 矩阵乘），两种语言，按复杂度分工——这比“全部手写”或“全部 Triton”都更实际。

回顾

这个 kernel 是前面所有硬件知识的落点。它解决的形状是 decode 的小 M（M ⩽ 64），而库在这个区间退化成“不管 N 多大都要约 135 μs”。它的策略是一个块吃 16 行（不补零）、沿 N 分块、块里的波前沿 K 分工最后在共享内存里折半归约。实现里用到的每一个常数都来自硬件：波前 64、MMAC 的 16×16×16、每条通道 4 个 FP32 累加器、一次读 8 个 BF16（16 字节对齐）、按 `gfx928` / `gfx936` / `gfx938` 分支选内建函数、不支持的架构编译出一条陷阱指令。工程上有三处可以直接抄：把假设写成两层检查（Python 侧用来路由、C++ 侧作为保险）、三层回退且失败后永久降级只警告一次、以及用宏关掉一个因工具链问题暂时不能用的优化并把症状和重启条件写进注释。另一个专用 kernel（lm_head）用 Triton 写，说明语言选择要按算子的复杂度分工。

矩阵乘讲完了。下一讲把注意力、量化和通信这三块在 DCU 上的选择一起看：后端按什么顺序挑、支持哪些量化方法、自定义 All-Reduce 在什么条件下才用，以及 PD 分离怎么搭。

第152讲 DCU 上的注意力、量化与通信

143 矩阵乘法与注意力的适配　145 低精度与量化的取舍　146 集合通信库与多卡互联　150 读 vllm-plugin-das：平台与运行时补丁

矩阵乘讲完了，剩下三块：注意力后端怎么挑、支持哪些量化方法、多卡通信怎么走。这三件事在插件里的共同特征是**没有唯一答案，只有一套选择规则**——按模型、按架构代号、按消息大小、按配置选。这一讲把这些规则读出来。

> **注意**
> 读的还是 `third_party/vllm-plugin-das`，提交 `81cbe8f`，只读不运行。配置和命令行的例子引自仓库里的文档（`docs/`、`README.zh-CN.md`），数字引自源码注释。

注意力：五个后端，一套优先级

插件注册了五个注意力后端（`vllm_hcu/platforms/hcu.py` 的 `register_attention_backends`）：普通注意力的 FlashAttention 版和 Triton 版，MLA 的 FlashMLA 版和 Triton 版，以及稀疏 MLA 的版本。注册时有一个细节：**只在这个后端没有被覆盖过时才注册**（`if not backend.is_overridden()`），也就是说用户或第三方指定的选择优先于插件的默认。

选择的优先级写在一个函数里：

third_party/vllm-plugin-das/vllm_hcu/platforms/hcu.py 第 151–177 行 / 共 1033 行 ⇆ 对照

```
151 @cache
152 def _get_backend_priorities(
153     use_mla: bool,
154     use_sparse: bool,
155     prefer_triton_mla: bool = False,
156 ) -> list[AttentionBackendEnum]:
157     """Get HCU backend priorities; validation filters unavailable kernels."""
158     if use_sparse:
159         return [
```

1 三个输入决定一切 第 151–157 行

函数的输入只有三个布尔量：是不是 MLA 模型、是不是稀疏注意力、要不要优先用 Triton 版的 MLA。文档字符串里交代了分工：**这里只给优先级，可用性由后面的校验来过滤。**

2 **稀疏 MLA：两个候选** 第 158–162 行

3 **MLA：FlashMLA 优先，Triton 兜底** 第 164–173 行

4 **普通 MHA / GQA：FlashAttention 优先** 第 174–177 行

← 上一步 | 看整个文件 | 下一步 →

优先级之外还有两道机制，正是第 143 讲说的“一组候选 + 校验 + 回退”：

逐个校验。 `get_valid_backends` 把候选逐个拿出来，调用它的 `validate_configuration`（传入设备能力、头数、块大小、数据类型等），返回“不可用的原因”列表；`ImportError` 也被当成一条原因（算子库没装）。全部不可用时**抛错**，并把每个候选的原因打印出来——而不是悄悄降级到一个能跑但很慢的实现。

块大小的调和。 `update_block_size_for_backend` 处理第 143 讲提到的那个坑：注意力 kernel 支持的块大小和引擎的 KV 块大小要匹配。代码里能看到两处具体的调整（注释写得很清楚）：稀疏 MLA 的模型里还有一个索引器注意力层，上游可能先发现它并选一个 16 token 的块，所以要把管理器的块保持在至少 64 且 64 对齐；而 DeepSeek-V4 的稀疏后端要求 256 token 的 kernel 块，它的压缩缓存还要求管理器的块能被每个压缩比整除——于是用最小公倍数把块大小对齐上去，并打一条警告说明改了。

要点

这两段代码回答了一个常见问题：“为什么我在新卡上设的 `--block-size` 不生效？”因为块大小不是一个自由参数，而是**注意力后端、KV 布局、压缩比共同约束的结果**。适配时要把这些约束写进代码（并打日志说明改了什么），而不是写进文档让用户自己对。

还有一处和 KV 布局有关的规则，第 143 讲引过：插件的文档建议 MHA / GQA 模型用 FlashAttention 后端时设 `VLLM_KV_CACHE_LAYOUT=HND`，但同时提醒——**混合注意力的模型里可能有不支持这个布局的后端，这时候要取消设置，让 vLLM 去解析一个共同支持的布局**；MLA 等后端应按对应模型的配置来，不要全局强制。

量化：23 种启用，4 种注释掉

这块卡上，这种精度能不能换来速度

⇆ 对照

能不能用　能省下什么



成熟 GPGPU × FP8：原生支持：矩阵单元直接吃这种类型，省下的字节同时变成省下的时间。按官方文档确认峰值算力和支持的缩放粒度。

FP8：权重和 KV 减半的主力（第 93 讲）。关键问题是有没有原生的 FP8 矩阵乘：没有的话只能反量化成 BF16 再算，省显存但不省带宽也不省时间。

图 152-1　硬件 × 精度的支持矩阵（第 145 讲）。DCU 属于“较新的国产 GPGPU”那一行；FP8 的可用性在插件里是按架构代号判断的。

平台类里列着一张支持名单（第 148 讲的统计：启用 23 种，注释掉 4 种）。读这张名单能看出几件事：

覆盖面以“已有的开源量化格式”为主：AWQ 和 GPTQ 两家（各带一个 marlin 变体和一个自动解析的别名）、`compressed-tensors`、`fp8`、`fbgemm_fp8`、`gguf`、`quark`、`torchao`、`modelopt` 系列，以及厂商自己的几种（`deepseek_v4_fp8`、`fp8_per_tensor`、`fp8_per_block`、`online`、`slimquant_w4a8` 及其两个变体）。

被注释掉的四种是 `mxfp4`、`bitsandbytes`、`modelopt_fp4`、`gpt_oss_mxfp4`——都和 4 位浮点有关（第 93 讲）。注释掉而不是删掉，说明是“暂时不支持”而不是“永不支持”。

启动时会改写配置。`verify_quantization` 里有一段：用 AWQ 时如果没打开 Triton 版的 AWQ 实现，就打一条警告并自己把那个环境变量设上。这是第 147 讲说的“优先改配置”的具体例子——不改上游代码，只改配置。

FP8 的可用性按架构代号判断。`supports_fp8` 的实现是一行：返回这块卡是不是 `gfx938`。这就是第 145 讲那张三档支持表落到代码里的样子——**同一家的卡，不同代之间的精度支持可能不一样**，所以判断要按架构代号，不能按厂商或型号系列。

量化方法的实现文件在 `model_executor/layers/quantization/` 下，一个方法一个文件（`compressed_tensors` 的几种组合、`int8_runtime`、`lightop_fp8_runtime`、`lightop_autoawq`、`slimquant_w4a8` 及其 DeepGEMM 变体等）。另外补丁清单里 `worker/op_opt` 那一类有 70 多个文件，其中一批就是量化算子的替换（`patch_compressed_tensors_*`、`patch_w8a8_utils`、`patch_input_quant_fp8`、`patch_auto_awq` 等）——对应第 145 讲说的“量化算子适配要改四处”里的权重重排、缩放布局和反量化位置。

还有一处第 145 讲讲过的细节在文档里有明确说明：对稀疏 MLA 后端，平台会把用户写的 `--kv-cache-dtype fp8_e4m3` **转成内部的 `fp8_ds_mla` 格式**，并强调这是该后端专用的打包布局，不是所有注意力后端通用的 cache 格式。代码里就是那一行 `cache_config.cache_dtype = "fp8_ds_mla"` 加一条说明日志。

通信：RCCL、自定义 All-Reduce、IPC All-Gather

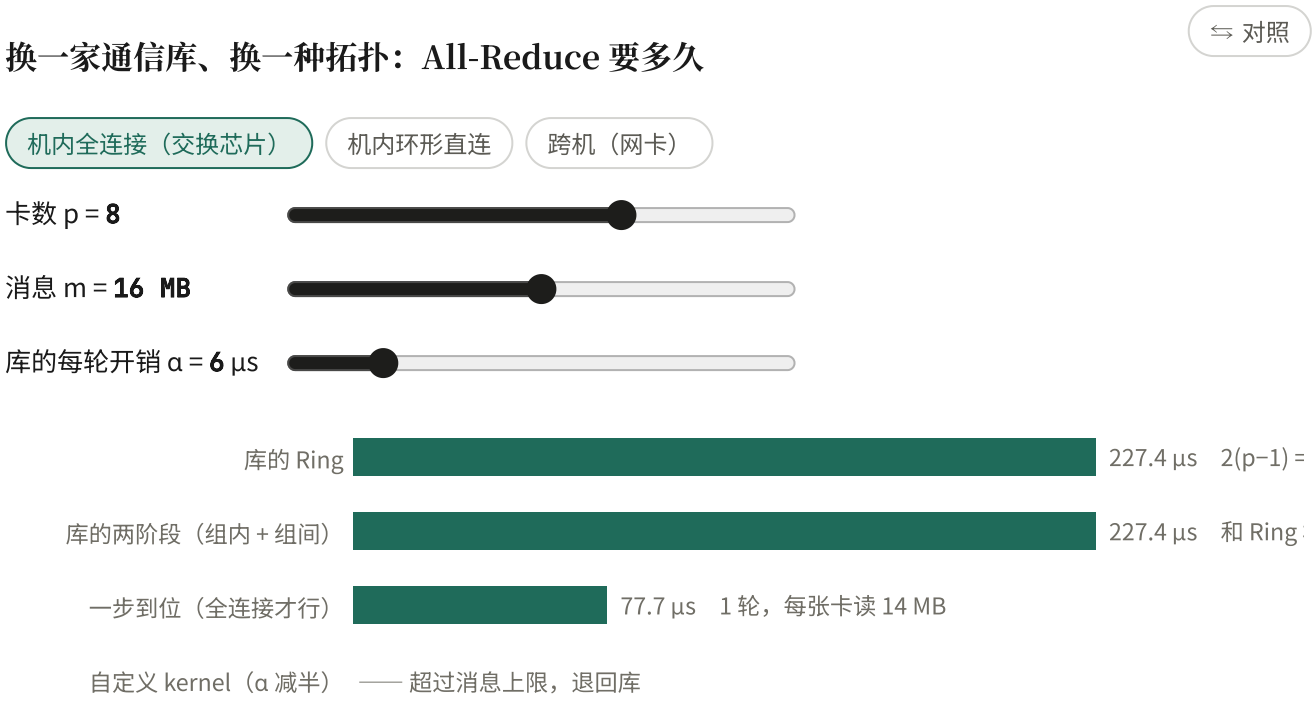


图 152-2　四种 All-Reduce 做法在三种拓扑下的耗时（第 146 讲的 α-β 估算）。插件的自定义实现对应“自定义 kernel”那一行，它的适用区间由下面这些条件决定。

通信这一层的三件事，第 146 讲都讲过原理，这里对应到代码：

库：RCCL，但名字叫 nccl。平台类里 `dist_backend = "nccl"`（第 149、150 讲），所以引擎和模型代码一行不改。

通信器：直接复用 vLLM 的 CUDA 通信器。`get_device_communicator_cls` 返回的是上游的 `CudaCommunicator` 类路径——又一次“沿用 CUDA 路线”的红利。

自定义 All-Reduce：一个条件分支。第 146 讲列过它的适用区间，这里把源码里的条件再按顺序排一遍（`distributed/device_communicators/custom_all_reduce.py` 和 `csrc/custom_all_reduce.cuh`）：

条件	值	不满足时
卡数	2、4、6、8	打日志，禁用
消息大小	默认上限 8 MiB	退回库
链路	要求全连接（逐对查跳数为 1、类型为直连）	非全连接时上限压到 512 KB，并提示可以加 `--disable-custom-all-reduce`
对齐	字节数是 16 的倍数、张量基本连续	退回库
一阶段还是两阶段	全连接时：≤4 卡且消息 < 512 KB，或 ≤8 卡且消息 < 256 KB 用一阶段；否则两阶段。PCIe 版的两个阈值是 1 MB 和 64 KB	由宏里的条件自动选
块数	上限 36，这条路径上默认 16	—

“默认只用 16 个块”这一条配着源码里的注释读很有意思：注释说用 36 个块在作者试过的那些卡上最好或接近最好，并且提到 NCCL 的 kernel 也只占少量计算单元，猜测原因是块太多会在互联总线上争抢。而 ROCm 这条路径把默认值降到 16——**同一份代码，不同平台的最优并行度不同**，这正是第 143 讲说的“参数要按硬件重调”。

IPC All-Gather：为 MoE 的小消息写的。第 146 讲引过它的动机注释：NCCL 的 All-Gather 在 decode 的消息大小上每次约 55 μs，而 MoE 的分发每层都要付一次。它的实现是一个 Triton kernel，设计说明里有三条值得记住的约束：每个程序自带调用计数器（所以标志位单调递增、能被图捕获）、两个缓冲槽按调用奇偶交替（一张卡只能在收到所有对端的第 k 次标志后才开始第 k+1 次）、**一个程序就是一个波前所以写入有序**。最后这条是用硬件常数换掉了一次显式同步。

还有一条工程纪律写在同一段注释里：**所有卡必须对每一次调用走相同的路径**（自定义还是回退），而这个决定只依赖全局的尺寸信息和公共的行布局。否则就会出现一部分卡在等自定义 kernel 的标志位、另一部分卡在库里等，直接死锁。

PD 分离：两套 connector

第 112 讲讲过 PD 分离的原理，第 130 讲读过 vLLM 的 KV connector 接口。插件里实现了两套（`distributed/kv_transfer/kv_connector/v1/`）：一套基于 Mooncake（配套的补丁里还恢复了一个旧版本的 KV 布局接口，注释说明是为了通过 Mooncake 的校验），一套是厂商自己

的实现（`du_swift_connector`，另有一个数据并行版本）。仓库的 `docs/` 下有三份使用说明，`examples/` 下有代理脚本。

从文档里的启动命令能看出 PD 分离在这块卡上的真实形态（引自 `docs/pd分离使用.md`）：P 和 D 两侧各起一个 `vllm serve`，用 `--kv-transfer-config` 指定 connector、角色（`kv_producer` / `kv_consumer`）、缓冲大小、端口和代理地址；中间有一个代理进程转发请求。文档里还点出了几个实际问题：跨机时两侧都要打开“多机”开关；支持非对称切分（P 和 D 的并行度不同，要在配置里写明 D 侧的 TP 和 PP）；服务重启时**代理也要重启**。

这些细节是第 112、114 讲那些架构图在一块真实国产卡上的落地形态：原理一样，但每个参数都要按环境配。

> **专栏**
>
> `docs/` 下这些文档的写法也值得一提：环境变量、启动命令、拓扑组合一字不漏地列出来，连网卡名字和 IP 都在。这类文档不好看，但对适配工作最有用——因为多卡、跨机的问题九成来自环境配置，而不是代码。第 153 讲的清单会把这些整理成可勾选的条目。

三块合起来：一个模型的配置长什么样

插件的 `tests/models/` 下每个 checkpoint 一份 YAML 配置，写明它的并行拓扑、注意力后端、MoE 后端、KV cache 设置和量化方法。拿仓库里一个 MoE 模型的配置为例（`docker/image-citest-conf-hcu.yaml`）：注意力后端写 `FLASH_ATTN`，KV cache 类型 `fp8_e4m3`，KV 布局 `HND`，量化 `slimquant_marlin`，投机解码用 MTP 起草 3 个 token，张量并行 1，还要求打开 V2 的 model runner。

这张配置把这一篇的内容串了起来：**注意力后端（第 143 讲）、KV 精度与布局（第 145 讲）、量化方法（第 145 讲）、投机解码（第 96 讲）、并行度（第 110 讲）各选一个值，组合在一起才是“这个模型在这块卡上的跑法”**。而文档里反复强调的那句话也在这里生效：模型已注册或有配置，不代表精度验证已经通过；换一种量化、换一个拓扑、换一次构建，都要重新验。

回顾

注意力在 DCU 上是五个后端加一套优先级：稀疏 MLA 两个候选、MLA 先 FlashMLA 后 Triton、普通注意力先 FlashAttention 后 Triton；每个候选都要通过一次配置校验，全部不可用时抛错而不是降级；关掉自定义算子的开关会让 MLA 整体退回 Triton 版，避免一半自定义一半通用。块大小不是自由参数，而是被注意力后端、索引器层和压缩比共同约束的结果，插件用最小公倍数把它对齐并打日志。量化支持 23 种、注释掉 4 种 4 位浮点相关的方法，FP8 的可用性按架构代号（是不是 `gfx938`）判断，稀疏 MLA 的 KV 精度会被转成专用的打包布局。通信沿用 `nccl` 这个名字和上游的 CUDA 通信器，自定义 All-Reduce 是一个带六项条件的分支（卡数名单、8 MiB 上限、全连接、16 字节对齐、一阶段/两阶段阈值、块数 16），IPC All-Gather 为 MoE 的小消息而写、用波前宽度换掉一次同步，并要求所有卡走相同路径。PD 分离有两套 connector，文档里把环境变量、启动命令和重启顺序都列了出来。

案例读到这里，所有零件都看过了。最后一讲把它们串成一条路：从拿到卡到模型上线，每一步做什么、验收标准是什么。

第153讲 模型上线 DCU 的完整清单

141 实操 · 把 miniinfer 移植到非 CUDA　142 在新硬件上做性能分析

149 从 CUDA 到 DCU：迁移与版本矩阵　152 DCU 上的注意力、量化与通信

第六章的最后一讲把二十讲的内容压成一条可执行的路线：从机器上架到服务上线，哪些事必须做、每一步怎么算过关。

清单的价值不在于“别忘了什么”，而在于**每一项都有一个能机械判断的验收标准**。适配工作里最常见的失败不是某一步做不出来，而是某一步“看起来过了”就往下走——跑通了但有算子在回退、输出看着正常但某一层已经失效、压测很快但没预热、多卡能跑但两条通信路径里有一条会死锁。

清单

从拿到卡到上线：可勾选清单

⇆ 对照

全部 | 只看未完成 | 清空勾选

一 环境　0 / 5

☐ 驱动装好，管理工具能看到所有卡	卡数、显存、温度正常；连续跑一小时无掉卡	第 148 讲
☐ 工具包版本在支持的发布线上	读版本文件，和插件/引擎声明的发布线比对	第 149 讲
☐ 框架能看见卡，能跑一次矩阵乘	设备数正确；一次 BF16 矩阵乘结果正确	第 137 讲
☐ 算子库按厂商归档的版本安装	每个库能 import，且版本与目标组合匹配	第 149 讲
☐ 插件自检通过	诊断命令全绿；版本关卡通过	第 150 讲

二 跑通　0 / 4

☐ 单卡单请求能生成 token	给一个短提示，生成 32 个 token 不报错	第 141 讲
☐ 算子清单里没有回退到 CPU 的算子	在分发器层记账跑一次前向，回退计数为 0	第 140 讲
☐ 缺失算子都有明确去处	每个缺口标注：组合 / 自写 / 等厂商，并记进待办	第 140 讲
☐ 每种要用的精度各跑通一次	FP32 / BF16 / FP16（以及量化）各生成一次	第 139 讲

三 精度　0 / 5

☐ 逐层误差在阈值内	以同精度的参考平台为基准，按位置取最差值，每层都在阈值内	第 139 讲
☐ 贪心生成逐 token 一致	固定几个提示（短/长/中文/代码），和参考逐 token 比对	第 139 讲
☐ 任务级评测分数不低于参考	几个数据集上的分数记录在案，和参考平台比	第 139 讲
☐ 量化模型和“同一套量化方法的参考输出”比	不和 FP32 原模型比，和参考平台的同方法输出比	第 145 讲
☐ 每改一个 kernel 就重跑比对	把比对做成脚本，阈值写进脚本，越界即失败	第 144 讲

四 性能　0 / 6

☐ 算出每一项的理论下限	每步要读的字节 ÷ 带宽；矩阵乘的 2MNK ÷ 峰值	第 142 讲

☐	热路径上没有多余的主机同步	数一遍每步的同步次数；对比去掉前后的耗时	第 141 讲
☐	按形状矩阵测过矩阵乘	decode 的 M=1…64 和 prefill 的几个长度各测一遍，记实际带宽占比	第 143 讲
☐	注意力按 prefill / decode 分别验过	两种负载各量一次；长上下文单独测	第 143 讲
☐	图模式的收益和风险都量过	数图中断；开图捕获前后比对数值和耗时	第 138 讲
☐	放弃的优化记下了理由	每条写明占比、预期收益、为什么不做	第 142 讲

五 多卡　0 / 4

☐	量过 α、β、是否全连接、P2P 是否可用	小消息和大消息各测一次，拓扑逐对查	第 146 讲
☐	两卡结果和单卡比对过	逐 token 一致（允许累加顺序带来的小差异）	第 146 讲
☐	扩到目标卡数，曲线能解释	TPOT 和吞吐随卡数的变化与 α-β 模型的预测对得上	第 110 讲
☐	自定义通信的两条路都压测过	适用区间内更快；区间外自动退回库且不死锁	第 152 讲

六 服务化　0 / 5

☐	预热覆盖了即时编译和图捕获	首个真实请求的延迟和稳态接近	第 151 讲
☐	在线压测扫过一遍区间	按到达率扫，记 TTFT、TPOT 分布和 goodput	第 82 讲
☐	容量按 SLO 算过	给定流量和延迟要求，算出实例数并留余量	第 116 讲
☐	版本组合和配置冻结并记录	驱动/工具包/框架/引擎/插件/算子库版本 + 完整启动参数存档	第 149 讲
☐	回滚方案和降级路径准备好	上一组可用版本可一键切回；关掉自定义算子仍能跑	第 150 讲

驱动装好，管理工具能看到所有卡
怎么验：卡数、显存、温度正常；连续跑一小时无掉卡
为什么在这一步：掉卡、显存读数异常这类问题要在最前面排掉，否则后面每个结论都不可信。　（第 148 讲）
已完成 0 / 29；勾选只存在你自己的浏览器里。

图 153-1　上线清单，六个阶段 29 项。点一项看“怎么验”和“为什么在这一步”；点方框或再点一次勾选完成（只存在你自己的浏览器里）。

六个阶段的分界线是这样定的，每条分界都对应一个“不往回走”的承诺：

环境 → 跑通：版本组合固定下来了。之后再换任何一层版本，精度和性能两关都要重走（第 149 讲）。

跑通 → 精度：缺失算子都有去处，而且没有东西在悄悄回退到 CPU。这一条是后面所有数字可信的前提（第 140 讲）。

精度 → 性能：有了一个可信的基线。优化几乎每一步都改变数值，没有基线就分不清“优化引入的错误”和“本来就有的误差”（第 139 讲）。

性能 → 多卡：单卡的瓶颈都解释清楚了。单卡没弄明白就上多卡，通信的问题和计算的问题会纠缠在一起（第 142 讲）。

多卡 → 服务化：并行配置定了，两条通信路径都压过。

服务化 → 上线：版本、配置、压测结果都冻结并记录；回滚路径验证过。

三个最容易跳过的环节

二十讲写下来，有三件事值得单独强调，因为它们看起来不紧急，但跳过的代价最大。

第一，算子回退的检查。一个回退到 CPU 的算子，本站实测慢 15 倍，而且打断设备上的流水（第 140 讲）。更麻烦的是它会掩盖问题：性能数据面目全非，精度问题也可能被它遮住。做法很简单——在分发器那一层记一次账，确认回退计数为 0（第 141 讲的脚本可以直接改用）。

第二，按精度分别验。本站那个 FP16 的例子最有说服力：FP32 下四条提示全部逐 token 一致，换到 FP16 立刻生成一串句号——原因是 RMSNorm 的平方和溢出，和硬件没关系，但只在这个精度暴露（第 139 讲）。所以“跑通”和“跑对”都要按精度逐个来，而不是验一种推广到全部。

第三，把判断过程记下来。性能工作里丢的不是数字，是上下文：为什么选了这个分块、为什么放弃了那个优化、为什么这个版本组合不能动。第 142 讲给过记录的字段表，第 151 讲那个被宏关掉的 split-K 变体是个好范例——症状、条件、重启用条件全写在注释里。

> **要点**
> 清单里有一类项特别容易被当成“形式”：预热、回滚、版本冻结。它们的共同点是**只在出问题时才体现价值**，而新硬件上出问题的概率比成熟平台高。即时编译一次要几十秒（第 151 讲），落在第一个真实请求上就是一次超时；版本没冻结，三个月后复现不出来当初的结果。

回到那张分层图



图 153-2　九层软件栈（第 134 讲的图，切到海光那一列）。清单里的六个阶段，其实是自下而上把这九层逐层点亮的过程。

把清单和这张图对上，能看出适配工作的形状：

阶段	点亮哪几层	典型耗时的来源
环境	驱动、运行时、工具链	版本错配；每换一次组合都要重来
跑通	框架、算子库	缺失算子的处理；少数模型要插件提供实现
精度	算子库、kernel	低精度下的溢出、累加顺序、自写 kernel 的 bug
性能	kernel、图编译、运行时	小 M 矩阵乘、注意力后端、下发与同步开销
多卡	通信库、互联	拓扑探测、自定义通信的适用区间
服务化	引擎与插件	预热、配置组合、SLO 下的容量

第 134 讲说“要重做的是接口下方的实现”，现在可以把这句话说得更具体：**要重做的工作量集中在算子库和 kernel 两层，而要花掉的时间往往集中在环境和服务化两头**。前者是技术问题，后者是组合问题——而组合问题没有捷径，只能靠记录和自动化压住。

这一章的几个数字

二十讲里本站真实跑出来、或从公开资料引用的数字，挑出最能带走的几个：

数字	含义	出处
4.4%	nano-vllm 里和 NVIDIA 绑定的代码占比；但这 4.4% 背后是几个大工程	第 134 讲（本站统计）
31 种 / 2221 次	Qwen3-0.6B 一步 decode 用到的 aten 算子种数、要在设备上算的调用次数	第 140 讲（本站实测）
27 张图 → 2 张图	一行 `int(pos[-1])` 让 `torch.compile` 把前向切成 27 段；去掉后是一张 3319 节点的图	第 138 讲（本站实测）
1.0 / 4e-3	FP16 溢出时第 4 层的最差位置误差 / 整张量误差——差了两个量级	第 139 讲（本站实测）
150 μs 对 3.6 μs	一次主机同步对一次小 kernel 下发（约 44 倍）	第 141、142 讲（本站实测）
4.6 倍 / 10.6 倍	手写融合 kernel 相对基础算子组合的加速（decode / prefill 形状）	第 144 讲（本站实测）
0.23×	没有原生低精度矩阵乘时，INT8 方案相对 FP32 基线的速度	第 145 讲（本站实测）
0.93 → 1.73 TB/s	lm_head 小 M 形状上，库的 256×256 分块对专用 kernel 的实际带宽	第 143、151 讲（引自插件源码注释）
约 135 μs	压缩投影在 M ⩽ 64 时，库选的分块“不管 N 多大”都要的时间	第 143、151 讲（引自插件源码注释）
约 55 μs	NCCL 的 All-Gather 在 decode 消息大小上每次调用的耗时	第 146、152 讲（引自插件源码注释）
25 万行 / 11.5 万行	海光 vLLM 插件的总行数 / 其中测试的行数	第 147、150 讲（本站统计）

专栏

这张表里前七行是在一台 MacBook 上用 Apple MPS 跑出来的。换到任何一块真实的国产卡上，数值都会不同，但**现象和方法不会变**：同步比下发贵一两个数量级、融合在带宽受限的算子上值几倍、没有原生 kernel 的低精度会变慢、库的大分块在小 M 上退化、按位置取最差才能发现局部失效。这也是第六章的写法——原理在能动手的硬件上验证，案例在只读的真实代码里对照。

整个课程到这里

第一章从向量和矩阵乘法开始，第二章把 Transformer 和现代架构拆开，第三、四章讲单机与分布式推理，第五章手写了一个推理引擎、再读了工业级引擎，第六章把模型搬到另一块芯片上。

如果要用一句话串起来：**上面的每一层都在给下面的层下订单，而下面的每一层都在用自己的物理约束改写上面的设计**。注意力的数学决定了要读多少字节，带宽决定了 decode 的下限，矩阵单元的形状决定了 kernel 怎么分块，而一块卡的波前是 64 而不是 32 这件小事，会一直传上去，变成一行必须手改的归约循环。

接下来往哪走，取决于你关心哪一层：想往下，就挑一个算子在你手上的硬件上写到接近上限；想往上，就挑一个真实负载做一次完整的容量规划和上线复盘。两条路都会回到同一个习惯——**先算理论值，再量实测，解释清楚两者的差距**。