2. 预训练

从数据系统、数据混合到规模化训练。

Table of contents

1 概览 **4**

2 数据管线与质量 **6**
- 2.1 数据不是一个文件夹，而是一套可审计系统 6
- 2.2 一条典型的预训练数据管线 7
- 2.3 名义 token 预算与有效 token 预算 7
- 2.4 面向 Common Crawl 的过滤策略 8
- 2.5 去重不是附属步骤，而是核心能力保护 8
- 2.6 常见去重策略：精确哈希、MinHash，再到语义相似度 9
- 2.7 模型逐字记忆了训练片段，如何诊断与修复 12
- 2.8 数据治理：许可证、PII 与审计追踪 13
- 2.9 模型驱动过滤与治理：质量分类器与安全分类器是同一类杠杆 13
- 2.10 质量分类器与两阶段过滤 14
- 2.11 构建毒性过滤器时，最怕的不是漏掉几个脏样本，而是误删整类高价值文本 . . 14
- 2.12 本节结论 14

3 数据混合与分布 **15**
- 3.1 模型看到的不是世界，而是加权后的世界 15
- 3.2 为什么“更多代码”可能反而伤害对话质量 16
- 3.3 如何为通用 LLM 设计数据混合 17
- 3.4 领域性能上升、通用基准下降，通常说明了三件事 18
- 3.5 课程学习：不是“由易到难”，而是“按训练阶段重新分配预算” 19
- 3.6 如何估算一个新数据源的边际价值 19
- 3.7 本节结论 20

4 计算与扩展 **20**
- 4.1 预训练为什么贵：四个主旋钮同时在转 21
- 4.2 一个足够好用的训练 FLOPs 估算式 22
- 4.3 Chinchilla 改变了什么：不是“越大越好”，而是“别把大模型喂得太少” . . 23
- 4.4 参数翻倍、上下文翻倍，分别会发生什么 24
- 4.5 一台机器为什么不够：预训练天然会走向分布式 24
- 4.6 并行策略地图：DP、TP、PP、ZeRO/FSDP 与序列并行 25

4.7 Megatron 3D 并行 vs ZeRO-3/FSDP：不是替代关系，而是不同层次的问题 . . . 27
4.8 激活检查点：用重算换显存 . . . 28
4.9 混合精度：为什么 BF16 成了主流起点 . . . 28
4.10 用 roofline 看系统瓶颈：你的训练到底是在算，还是在等 . . . 28
4.11 GPU utilization 很高，但 tokens/s 很差：通常不是算子，而是有效 token 比例出了问题 . . . 28
4.12 一个粗略的 H100 估算：7B 模型、1T token 要多少卡、多久、多少钱 . . . 29
4.13 本节结论 . . . 30

5 训练配方与监控 30
5.1 训练配方不是“超参数列表”，而是稳定性边界 . . . 30
5.2 典型学习率调度：warmup 后再衰减，而不是开局就冲顶 . . . 31
5.3 一条现代预训练配方通常长什么样 . . . 31
5.4 Operator 视角：一张训练看板至少要看什么 . . . 32
5.5 Loss 中途飙升 3 倍又慢慢恢复，怎么排查 . . . 33
5.6 恢复语义：如何证明一次 resumed run 等价于一次 uninterrupted run . . . 33
5.7 预训练中最常见的五类故障模式 . . . 34
5.8 探测集不是“大 benchmark 缩小版”，而是训练期的早期预警系统 . . . 34
5.9 训练 loss 下降，但下游分数停滞，通常说明模型在“学容易学的东西” . . . 35
5.10 梯度范数：最廉价、也最常被忽视的地震仪 . . . 35
5.11 何时停止预训练：不是“loss 不降了”，而是“继续训练还值不值” . . . 35
5.12 本节结论 . . . 36

6 评估与下游影响 36
6.1 困惑度衡量的是“平均惊讶程度”，不是“产品完成度” . . . 37
6.2 为什么更低的困惑度不保证更好的指令跟随 . . . 39
6.3 一个“loss 更低、产品更差”的具体案例 . . . 40
6.4 预训练如何约束中训练、SFT 与推理系统 . . . 40
6.5 基座模型代码弱，应该在预训练修，还是在后面修 . . . 41
6.6 为什么模型 A 困惑度更低，模型 B 下游任务更强 . . . 41
6.7 从预训练走向中训练与后训练 . . . 42
6.8 本节结论 . . . 42

7 本章小结 43
7.1 问题小结 . . . 43

8 参考资料 45

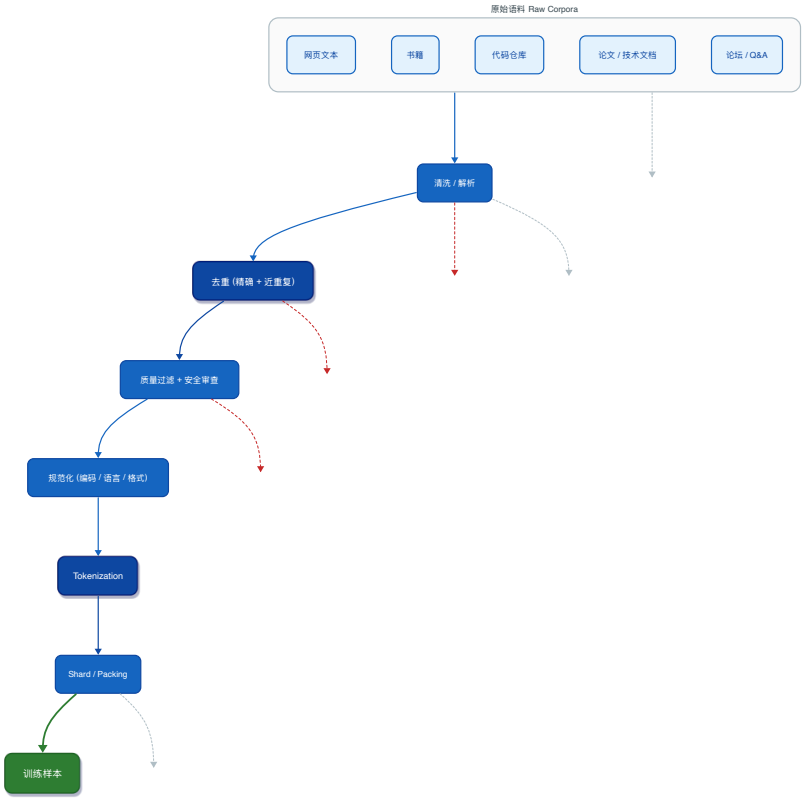

Figure 1: **数据漏斗——从原始语料到可训练** token

1 概览

> **Important**
>
> 本章讨论的不是“模型会不会回答问题”，而是一个更底层的问题：
> 一个大语言模型的参数，究竟是怎样被训练出来的？
> 更准确地说，本章要回答的是五类工程问题：
>
> 1. 你准备让模型看见什么数据，又怎样保证这些数据值得看？
> 2. 你怎样决定不同数据源的比例，让模型学到你真正想要的能力结构？
> 3. 在给定算力预算下，参数、token、上下文长度和精度应该怎样配平？
> 4. 一次持续数周甚至数月的训练，怎样才能稳定、不发散、可监控、可回滚？
> 5. 你怎么知道模型是真的变强了，而不是只是在某个验证集上把 loss 压低了？

想象一个非常真实的值班场景：团队启动了一次计划运行 18 天的预训练任务，目标是在 512 张 H100 上，把一个 $N = 7\text{B}$ 的稠密 decoder-only 模型训练到 $D = 10^{12}$ token。按粗估公式 $C \approx 6ND$，理论训练计算量约为 4.2×10^{22} FLOPs。第 7 天，训练 loss 突然 spike 到原来的 $2.8\times$；第 14 天，validation perplexity 继续下降，但内部工具使用 probe 却开始回落。此时真正的问题不再是“next-token prediction 是什么”，而是：你是否理解自己正在训练的整个系统。

> **贯穿全章的运行实例：一个假设性的 7B × 1T 训练项目**
>
> 为让抽象讨论始终落在同一个工程对象上，本章会反复引用一个假设案例：
>
> - 模型：7B dense decoder-only Transformer
> - 数据：1T nominal token；总体 mixture 为网页与通用文档 68%、书籍 12%、代码 10%、学术 6%、对话与论坛 4%
> - 硬件：512× H100，BF16 训练，目标 MFU $\eta \approx 0.38$
> - 训练计划：全局有效 batch 约 8M token/step，对应约 1.2×10^5 step
> - 运行事件：一次 benchmark contamination 惊报、一次 loss spike、一次 checkpoint resume 语义不一致排查
>
> 这个例子不是在“模拟一家具体公司”，而是把大多数大模型团队都会碰到的约束压缩到一个可追踪的对象里：数据池、预算、分布式拓扑、训练看板，以及最后的下游能力。

> **本章你可以反复思考的问题**
>
> 1. 名义 **token budget** 和 **effective token budget** 有何区别？
> 2. 你如何证明一次 **resumed run** 在语义上等价于一次 **uninterrupted run**？
> 3. **GPU utilization** 很高而 **tokens/s** 很差时，你先查什么？

在第一章里，我们把 LLM 看成一个 next-token predictor：给定前文，预测下一个词元。这个定义解释了模型推理时在做什么。而预训练解释的是：这样一个预测器，为什么会逐渐获得语言能力、知识压缩能力、迁移能力，以及后来被塑造成“助手”的可能性。

现代 decoder-only LLM 的预训练，通常仍然围绕同一个极其朴素、却极其强大的目标函数

展开：自回归语言建模。设词元序列为 $x_1, \ldots, x_T$，模型参数为 θ，则训练目标是最小化真实下一个词元的负对数似然：

$$\mathcal{L}_{\text{CLM}} = -\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t})$$

这就是为什么预训练通常被称为自监督学习：监督信号不是人工标注，而是文本自身提供的。训练时使用 teacher forcing，模型在位置 t 看到的是真实前缀 $x_{<t}$；推理时，它看到的是自己刚刚生成的前缀。因此，预训练优化的是条件分布拟合，而不是直接优化“像一个有帮助的助手那样回答”。GPT-3 把这种大规模自回归预训练路线推到了一个新的规模阶段，也让“先训一个强 base model，再做后续塑形”成为主流范式 (Brown et al., 2020)。

同样重要的是：预训练产物通常是 **base model**，而不是 **assistant**。
预训练负责学会“世界上通常会出现什么”；中训练负责把分布往特定领域推进；后训练负责让模型在与人交互时表现得更可用、更可控、更一致。能力边界主要在预训练阶段确定，行为接口则主要在后续阶段被塑造。

从工程角度看，预训练不是“把很多文本喂给模型”这么简单。更接近事实的说法是：它是一条从原始语料到基础模型的长链路，而链路的每一段都会改变最终结果。

如果把上面的链路换成更偏 operator 视角的签名图，可以得到图 2-1。

本章因此不再按“先讲一点数学、再讲一点系统”的顺序组织，而是按一条更符合工程决策的路径来展开：

1. 数据管线与质量：模型到底看到了什么；
2. 数据混合与分布：模型以什么比例看到了这些东西；
3. 计算与扩展：你能否在预算内把训练真正跑起来；
4. 训练配方与监控：它会不会稳定收敛，以及出了问题如何定位；
5. 评估与下游影响：这些预训练决策怎样约束中训练、后训练与推理系统。

从这里开始，本章会沿着三条互相缠绕的主线前进：数据系统 $\longrightarrow$ 预算配平 $\longrightarrow$ 基础设施。如果你能把这三条线同时看见，后面的 loss、PPL、MFU、checkpoint、污染审计就不再是散点，而会变成同一套系统的不同观测面。

读完本章后，你应该能够把“预训练”重新理解为一句完整的话：

> 预训练是在固定算力预算下，用可追溯、可筛选、可混合的大规模 **token** 流，训练一个尽可能强、尽可能稳、尽可能可泛化的条件语言模型。

2 数据管线与质量

本节先想这些问题

1. 详述预训练数据管线：数据源 → 过滤 → 去重 → 分词 → 混合。
2. 为什么“数据质量 > 数据数量”是最重要的预训练原则？
3. 在预训练前，你会对 Common Crawl 数据集应用哪些过滤？
4. 解释训练-训练去重与训练-测试污染——各自为什么重要？
5. 你如何构建一个检测和防止基准测试污染的管线？
6. 你发现模型逐字记忆了训练数据片段——如何诊断和修复？
7. 预训练中哪些数据治理实践很重要（许可证、PII、审计追踪）？
8. 如何大规模处理训练数据中的 PII？
9. 常见的去重策略有哪些（精确哈希、MinHash、嵌入相似度）？
10. 你需要创建一个分类器来过滤预训练语料中的有毒内容。你使用什么架构？如何处理误删有价值数据的假阳性？
11. “质量分类器”（如基于 Wikipedia vs 随机网页训练的分类器）在预训练数据管线中的作用是什么？Llama 3 是如何处理的？

2.1 数据不是一个文件夹，而是一套可审计系统

预训练里最常见、也最危险的错觉，是把“数据”想成一个静态资产：仿佛你已经有了几百 GB 或几 TB 的文本，剩下的事只是把它送进 dataloader。现实里，预训练数据更像一套持续运转的数据系统。它必须回答至少四个问题：

1. 这些文本从哪里来，许可证和使用边界是什么？
2. 这些文本在被模型看到之前经历了哪些过滤、修改与剔除？
3. 这些文本是否和验证集、基准集或敏感数据发生了重叠？
4. 如果模型后来出现记忆化、偏见或能力缺口，能不能反向追溯到具体数据桶？

因此，好的预训练数据工程从来不只是“收集更多文本”，而是要让数据具备三种性质：高质量、可追溯、可调配。

这也是为什么“数据质量 > 数据数量”会成为预训练里最重要的原则之一。海量噪声文本当然也能训练出一个会续写的模型，但它更可能学到的是模板残留、SEO 垃圾页、近重复内容、导航栏语言和格式噪声。CCNet 的一个核心结论就是：Common Crawl 这类超大规模网页语料极有价值，但前提是必须通过语言识别、去重和高质量过滤，把真正值得保留的部分抽取出来

(Wenzek et al., 2019)。The Pile 则从另一个方向说明了同一件事：数据多样性和质量会共同决定模型跨领域泛化的上限（Gao et al., 2020)。

2.2 一条典型的预训练数据管线

一条可工作的预训练数据管线，通常至少包含下面九步：

1. 数据采集与解析
   抓取网页、书籍、论文、代码仓库、论坛、文档站点等原始语料，并把 HTML、PDF、代码仓库文件等格式解析成统一文本表示。
2. 文本规范化
   统一 Unicode、空白、换行、控制字符和编码异常；剔除脚本、样式、cookie banner、导航模板等非正文噪声。
3. 语言识别与文档边界修复
   判断语言，去除混杂或错误编码文本，修复被错误拼接的页面片段，保证“文档”这个单位本身有意义。
4. 质量过滤
   去掉极短、极长、结构异常、标点/数字比例异常、重复行异常的文本；加入基于参考高质量语料的分类器或打分器。
5. 安全、隐私与合规过滤
   处理 PII、受限许可证文本、明显敏感内容、高风险站点和不可用于训练的来源。
6. 去重
   先做精确去重，再做近重复去重，必要时补做行级或段级去重。
7. 分词前审计
   检查 tokenizer 在关键领域上的碎片率、特殊字符处理和长度分布；必要时回退修改数据规范化或 tokenizer 设计。
8. 分桶、采样与混合准备
   按来源、语言、领域、质量等级切分数据桶，为后续 mixture 设计和 annealing 做准备。
9. 留出集与污染审计
   在最终训练前，对 benchmark、验证集、红队集、内部探测集做重叠检查和 contamination audit。

这条流水线里，任何一步都不是“可有可无的卫生工作”。它们决定的是：模型究竟在用昂贵的参数容量学习知识模式，还是在浪费容量拟合噪声残留。

2.3 名义 token 预算与有效 token 预算

预训练计划里经常写“本次训练共看见 D_{nominal} 个 token”。但真正对参数更新形成高价值信号的 token 数，更接近下面这个工程近似：

$$D_{\text{eff}} \approx D_{\text{nominal}}(1 - r_{\text{dup}})(1 - r_{\text{pad}})\rho_{\text{parse}}\rho_{\text{quality}}$$

其中，r_{dup} 是重复或近重复比例，r_{pad} 是 packing / padding 浪费比例，ρ_{parse} 是解析成功率，ρ_{quality} 则是高质量、可学习 token 的占比。它不是严格定律，但非常适合工程判断：计划 **token** 预算不是有效 **token** 预算，训练信号会在数据系统里层层漏损。

在贯穿全章的 7B 示例里，如果 nominal token 是 1T，而去重、padding、解析失败和低质量文本共同把有效系数压到 0.72，那么模型真正获得的有效预算只有约 7.2×10^{11} token。对一个原本就靠 compute-optimal 配比吃饭的训练计划来说，这种偏差足以把“刚好够训练”推成“明显训练不足”。

2.4 面向 Common Crawl 的过滤策略

Common Crawl 是预训练最常见的原始来源之一，因为它规模巨大、覆盖面广、更新快；但它也是最典型的“高价值、高清洗成本”数据源。对 Common Crawl 这类网页数据，至少应该考虑下表中的过滤维度：

过滤维度	典型做法	解决的问题
HTML/模板剥离	boilerplate removal、正文提取	去掉导航栏、页脚、广告、脚本残留
语言识别	fastText 类语言 ID、段落级回退	避免多语言混杂和错判
长度与结构规则	文档长度、句长、换行模式、重复行比例	去掉异常拼接页、列表页、模板页
字符统计规则	标点比例、数字比例、异常字符比例	拦截乱码页、日志页、SEO 垃圾页
参考语料相似度	与 Wikipedia / 高质量语料的分类或打分	优先保留更像“可引用文本”的文档
域专用过滤	code/math/doc 页面定制提取与规则	降低代码页、教程页、论坛页的误杀
安全与 PII 过滤	站点规则、正则/NER、模型分类器	移除高风险个人信息和敏感样本
去重	文档级、行级、近重复去重	提高 token 利用率，减少记忆化

CCNet 的做法很有代表性：先从 Common Crawl 中提取单语文本，再做语言识别、去重，并使用“与高质量参考语料接近程度”的过滤思想，提高最终保留语料的平均质量（Wenzek et al., 2019）。这类方法的关键，不在于规则本身，而在于规则组合：启发式规则适合高召回地扫掉噪声，模型分类器适合对边界样本做更细的排序。

2.5 去重不是附属步骤，而是核心能力保护

“训练-训练去重”和“训练-测试污染”经常被混为一谈，但它们解决的是不同层次的问题。

训练-训练去重

训练集内部的重复，会造成三个直接后果：

- 浪费 token 预算：同一内容被模型反复观看，等于昂贵算力在重复购买同一条信息；

- 提高记忆化风险：大量近重复样本会让模型更容易逐字复述，而不是抽象泛化；
- 扭曲学习优先级：高频模板会在梯度里被过度放大，挤占更稀缺但更有价值的模式。

Lee et al. 系统展示了去重的收益：对标准语料做更彻底的去重，不仅能显著降低逐字记忆，还能以更少的训练步数达到同等甚至更好的效果，并减少训练集与验证集的重叠 (Lee et al., 2022)。

训练-测试污染

污染则是另一个问题：当 benchmark 样本或其高度相似变体进入训练集后，评估分数会被高估。污染不一定表现为“原样出现同一题”，也可能表现为：

- 题目被改写，但事实结构相同；
- benchmark 中的代码片段、题解、论坛讨论进入训练数据；
- evaluation prompt 与训练语料只有轻微表面差异。

因此，训练-训练去重保护的是泛化质量和 **token** 效率；训练-测试污染排查保护的是评估可信度。两者都必须做，而且不能相互替代。

> ⚠ **运行实例：一次被公共题解污染的 benchmark 惊报**
>
> 在 7B 示例里，团队在约 0.62T token 时发现一个代码 benchmark 分数异常跃升，涨幅远高于相邻 probe。最终定位到的问题不是“模型突然顿悟”，而是训练 mixture 混入了公开题解镜像站、论坛讨论串和近重复教程页。虽然 benchmark 原题文本没有一字不差地出现，但解法片段、测试用例解释与讨论镜像已经足够污染评估。
> 修复动作不是把这个分数记成“阶段性胜利”，而是三步同时做：1. 按 benchmark family 做 denylist 与近重复审计；2. 对题解站、教程聚合站、论坛镜像站单独降权或剔除；3. 对后续 checkpoint 强制执行 contamination audit，并把污染样本族加入持续监控。

2.6 常见去重策略：精确哈希、MinHash，再到语义相似度

预训练里的去重，最好理解成一个由便宜到昂贵、由高精度到高召回的级联系统，而不是单一算法。
设规范化后的文档为 d，规范化算子为 $N(\cdot)$，则整条链路的目标可以写成：

$$\mathrm{dedup}(d_i, d_j) = \mathbf{1}\big[\mathrm{sim}(N(d_i), N(d_j)) \geq \tau\big]$$

其中 $\mathrm{sim}(\cdot,\cdot)$ 可以是“完全相等”、Jaccard 相似度、段级覆盖率，或者嵌入余弦相似度；τ 则是该层的判重阈值。工程上常见做法不是一次性用最贵的相似度，而是采用分层过滤：

$$C_{\text{exact}} \ll C_{\text{MinHash}} \ll C_{\text{segment}} \ll C_{\text{embed}}$$

这里 C 表示单位样本对比成本。换句话说，先用最便宜的方法扫掉最容易识别的重复，再把昂贵方法留给高风险、小规模、边界模糊的集合。

1. 精确哈希
   对规范化后的文档直接做哈希：

$$h_i = H(N(d_i))$$

   若

$$h_i = h_j$$

   则把 d_i, d_j 视为精确重复，只保留一份。这里的关键不在哈希函数名字本身，而在规范化是否充分：空白、大小写、Unicode 形式、HTML 残留、换行风格如果没有先统一，很多“看起来一样”的文档在哈希层仍会被当成不同文档。
   这一层的优点是速度极快、实现简单、误判率极低；缺点也同样明确：它只能捕捉

$$N(d_i) = N(d_j)$$

   这种严格相等，抓不住“只改了标题”“多了一段免责声明”“镜像站改写了少量模板”的近重复。

2. 近重复去重（**shingling + MinHash/LSH**）
   对每个文档构造 k-shingle 集合：

$$S_k(d) = \{s_1, s_2, \dots, s_m\}$$

   其中每个 s_t 是长度为 k 的 token n-gram 或字符 n-gram。两个文档的近重复程度通常先用 Jaccard 相似度表示：

$$J(d_i, d_j) = \frac{|S_k(d_i) \cap S_k(d_j)|}{|S_k(d_i) \cup S_k(d_j)|}$$

   直接计算所有文档对的 J 在大规模语料上代价太高，因此通常用 MinHash 构造签名。对第 r 个随机置换 π_r，定义：

$$m_r(d) = \min_{s \in S_k(d)} \pi_r(s)$$

   MinHash 的核心性质是：

$$\Pr[m_r(d_i) = m_r(d_j)] = J(d_i, d_j)$$

   也就是说，如果我们用 R 个独立哈希函数得到签名向量，那么 Jaccard 相似度可由碰撞比例近似估计：

$$\hat{J}(d_i, d_j) = \frac{1}{R}\sum_{r=1}^{R} \mathbf{1}[m_r(d_i) = m_r(d_j)]$$

为了避免对所有文档对逐一比较签名，通常再叠加 LSH (Locality-Sensitive Hashing)。若把 $R = br$ 个签名分成 b 个 band、每个 band 含 r 行，则某一对文档成为候选对的概率近似为：

$$P_{\text{cand}}(J) = 1 - (1 - J^r)^b$$

这条式子很有工程意义：它解释了为什么 LSH 会形成一个“软阈值”——当 J 低于某个区域时，候选概率迅速下降；当 J 足够高时，候选概率迅速接近 1。也正因为此，MinHash + LSH 是大规模网页语料近重复去重的主力方案：它在召回高相似重复和控制两两比较成本之间给出了非常实用的折中。

3. 行级 / 段级去重

很多高重复文本的问题，并不发生在“整篇文档几乎一样”，而发生在局部段落被反复复制。教程、法律条款、模板页、README、FAQ、代码注释、站点页脚都属于这一类。

这时更有效的对象不是整篇文档，而是段集合或行集合。设文档被切成段集合

$$P(d) = \{p_1, p_2, \dots, p_n\}$$

可以定义段级覆盖率，例如：

$$\text{cover}(d_i \to d_j) = \frac{\sum_{p \in P(d_i)} |p| \cdot \mathbf{1}\left[\exists q \in P(d_j),\ \text{sim}(p, q) \geq \tau_p\right]}{\sum_{p \in P(d_i)} |p|}$$

这里 $|p|$ 可以取段长（token 数），τ_p 是段级相似阈值。这个定义是有方向的：$\text{cover}(d_i \to d_j)$ 高，表示 d_i 中很大一部分内容都能在 d_j 里找到对应片段；这对检测“长文档中嵌入大量模板段”尤其有效。

实际系统里，段级去重往往采用两阶段策略：先对段做局部哈希或 MinHash 候选召回，再对候选段做更精细的 token overlap 比较。它的价值在于：即使整篇文档的 Jaccard 不高，也能发现“有 40% 内容其实是复制来的”这类污染。

4. 嵌入相似度回查

最昂贵的一层通常是语义嵌入。设编码器 $f_\phi(\cdot)$ 将文档映射到向量空间，并做 L_2 归一化：

$$e(d) = \frac{f_\phi(d)}{\|f_\phi(d)\|_2}$$

则两个文档的语义相似度可写为余弦相似度：

$$s_{\cos}(d_i, d_j) = e(d_i)^\top e(d_j)$$

若

$$s_{\cos}(d_i, d_j) \geq \tau_e$$

则把它们视作语义近邻，再进入精排模型或人工审计。
这一层的优点是：它能抓住“措辞变化较大但语义几乎相同”的样本，因此非常适合做 **benchmark contamination audit**、内部保留题审计、敏感文档泄漏排查等高风险小集合问题。缺点同样明显：编码和 ANN 检索成本高，且阈值选择更脆弱，容易把“主题相近但不应去重”的文档误合并。因此，嵌入相似度通常不作为全量主去重器，而更适合作为高风险回查层。

把这四层放在一起看，会更容易理解预训练去重的工程本质：
精确哈希负责清掉完全重复，**MinHash** 负责清掉大规模近重复，段级方法负责清掉局部复制污染，嵌入回查负责审计高风险语义重叠。
真正成熟的去重系统，追求的不是某个单点相似度最“聪明”，而是在固定算力预算下最大化：

$$\frac{\text{被移除的冗余 token}}{\text{去重成本}}$$

同时尽量降低两类错误：
- 假阴性：该删没删，重复内容继续污染训练；
- 假阳性：不该删却删了，把稀有但高价值的文本误杀掉。

这也是为什么去重不是一个纯算法选择题，而是一个相似度定义、阈值选择、成本预算与风险控制共同作用的系统设计题。

2.7 模型逐字记忆了训练片段，如何诊断与修复

“记忆化”如果只停留在抽象讨论层面，工程上很难处理。更有效的视角是：把它当成一种可诊断的系统故障。

先回答“怎么发现”：

- 对高风险私有文本、许可证敏感文本和长独特字符串做 extraction tests；
- 对训练集中出现频率异常高的句子、模板和长 span 建立 canary 检测；
- 对被投诉的输出进行逆向检索，看它是否能在训练源中找到高度相似文档；
- 对特定来源桶统计重复率、长公共子串率和 memorization-prone 模式。

再回答“怎么修”：

- 加强近重复去重和行级去重；
- 降低高重复来源桶的 mixture 权重；
- 移除模板化站点、聚合页、镜像站；
- 对私密或敏感语料做更强的排除与 PII 清理；
- 必要时回滚并重训受污染阶段的模型，而不是只在后训练里“靠安全提示堵漏洞”。

一个经验上很重要的判断是：
如果模型能稳定地、逐字地复述长片段，那通常不是“模型太强了”，而是数据系统某处出了问题。

2.8 数据治理：许可证、PII 与审计追踪

预训练数据治理的底线，不是“尽量少出事”，而是要让每一份高风险数据都能被解释、被追踪、被删除、被复盘。

至少需要三类治理机制：

许可证治理

你需要知道每个数据源的来源、抓取时间、使用许可、是否允许训练、是否允许再分发、是否允许商用。否则，模型一旦走向产品化，法律风险会在最糟糕的时候追上来。

PII 治理

大规模处理 PII 通常不能只靠一种方法。更现实的做法是多级联动：

- 基于站点和来源的粗过滤：先避开明显高风险源；
- 基于规则和模式的粗筛：邮箱、手机号、身份证号、地址等；
- 基于命名实体识别（NER）或轻量分类器的精筛：人名、组织、地址、账户信息等上下文敏感实体；
- 对高风险切片做人工 spot check 和抽样复核。

这里要格外警惕“低假阴性但高假阳性”的系统：如果你把大量医学、法律或企业文档都误删掉，数据质量会在你不注意的时候塌掉。PII 过滤必须看分桶召回率和误杀率，而不是只看单一总体分数。

审计追踪

一个可出版、可复现、可运营的预训练项目，应该能回答：

- 某个样本来自哪个源？
- 它经过了哪些过滤器？
- 为什么被保留、被降权或被删除？
- 它最后进入了哪个数据桶、以什么采样权重进入训练？

最实用的做法通常是为每个文档保留最小但足够的 lineage 信息：源 ID、抓取快照、许可证标签、语言标签、质量分数、去重状态、PII/安全标签、最终分桶与采样权重。没有这条线，后续所有“排障”都只能靠猜。

2.9 模型驱动过滤与治理：质量分类器与安全分类器是同一类杠杆

到这一步，可以把质量分类器、毒性过滤器、PII 检测器放在同一个工程抽象下理解：它们本质上都是 **learned gates on the data pipeline**。它们不会取代规则系统，但会在启发式规则之后，重新塑造“哪些文本值得进入训练分布”的边界。差别只在于目标函数：质量分类器更关心信息密度与可学习性，安全分类器更关心风险暴露与误召回成本。

从训练目标看，这类过滤器都在改变一个事实：进入经验风险的样本分布本身。也就是说，它们不是“预处理插件”，而是训练分布的前置参数。

2.10 质量分类器与两阶段过滤

质量分类器的作用，不是取代启发式规则，而是把“这段文本是否像高质量可学习文本”这件事，变成一个可训练、可排序、可调阈值的问题。

最经典的思路之一，是把高质量参考语料（如 Wikipedia、经过精选的文档集合）与随机网页语料区分开来，训练一个分类器或打分器，让管线优先保留更像“值得被引用”的文本。这类思想在 CCNet 与后续很多预训练管线里都非常常见 (Wenzek et al., 2019)。

实践中，一个稳妥的过滤架构往往是两阶段的：

1. 快筛阶段：启发式规则 + 轻量模型
   用 fastText、DistilRoBERTa 等快速模型先做大规模打分，保证吞吐。
2. 精排阶段：更强但更贵的模型或抽样人工校验
   只对边界样本、关键领域桶或高风险来源做精排和 spot check，控制假阳性。

Llama 3 的公开技术报告展示了这种思路的现代版本：Meta 不仅使用模型化质量过滤，还使用知识分类器来识别网页内容所属的信息类型，并对网页中过度代表的类别做下采样；同时，它们也用小模型和 Llama 2 标注信号来构建质量、代码、数学推理等领域分类器，以帮助确定更合理的数据混合 (Grattafiori et al., 2024)。这说明质量分类器的价值已经不止于“删脏数据”，它还直接参与了数据混合设计。

2.11 构建毒性过滤器时，最怕的不是漏掉几个脏样本，而是误删整类高价值文本

如果你要为预训练语料构建有毒内容过滤器，架构上通常有三个现实选择：

- 轻量文本分类器：吞吐高，适合全量扫描；
- 更强的编码器分类器：准确率更高，但成本更大；
- 分层系统：轻量模型做第一层，高风险或低置信样本再交给更强模型。

真正难的地方，不是选择模型名字，而是选择误差形状。例如，医学、法律、犯罪报道、心理健康求助、代码注释和社会新闻，往往同时包含大量“敏感词”和大量高价值知识。如果阈值过严，模型会失去处理现实世界复杂文本的能力；如果阈值过松，安全和产品风险会累积到后续阶段。

因此，毒性过滤器必须做切片评估：分别看新闻、医学、法律、论坛、代码、小说等桶的误杀率，并保留可回溯样本，持续手工复审边界案例。预训练数据治理的成熟度，恰恰体现在这种“不迷信总体准确率”的耐心上。

2.12 本节结论

可以把本节浓缩成四句话：

1. 预训练数据不是文件堆，而是一套可追溯的数据系统。
2. **Common Crawl** 很值钱，但只有在足够强的清洗、过滤和去重之后才值钱。
3. 训练集内部去重保护泛化与 **token** 效率；训练-测试污染排查保护评估可信度。
4. 质量分类器、**PII** 过滤、许可证治理和审计追踪，不是合规附属品，而是模型质量的一部分。

预训练数据不是一个文件夹，而是一套经过治理、去重、可审计的 token 生产系统。而当高质量数据池准备好之后，下一个问题就不再是“有没有数据”，而是“模型最终看到的是哪一种加权后的世界”。这就进入数据混合。

3 数据混合与分布

本节先想这些问题

1. 为什么训练数据混合很重要？举一个添加更多代码数据反而损害对话质量的例子。
2. 你如何为通用 LLM 设计数据混合？
3. 你添加了领域特定数据后领域性能提高但通用基准下降——发生了什么？
4. 预训练中的“课程学习”（curriculum learning）是什么？
5. 你如何估算向混合中添加新数据源的边际价值？在运行完整预训练之前可以跟踪哪些代理指标？

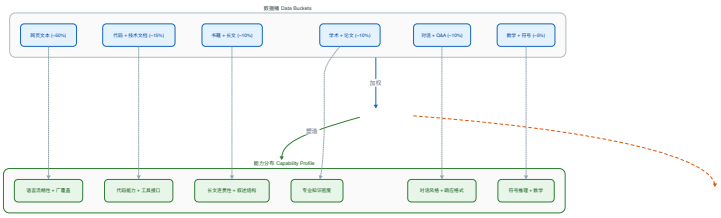

Figure 2: 数据混合塑造能力结构

3.1 模型看到的不是世界，而是加权后的世界

经过清洗、过滤和去重之后，预训练的下一个关键决策不是“是否开始训练”，而是“以什么比例训练”。这一步通常被低估，但它直接决定模型最终更像什么。

设有多个数据源 $D_1, D_2, \ldots, D_n$，对应采样权重为 $w_1, w_2, \ldots, w_n$，那么训练时来自某一数据源的概率可以写成：

$$p(D_i) = \frac{w_i}{\sum_j w_j}$$

这个式子很简单，但它揭示了预训练最重要的事实之一：

> 模型最终学到的不是客观世界，而是加权后的世界。

网页文本在真实世界里可能无处不在，但如果你让它在训练里占据压倒性权重，模型就会更像“互联网续写器”；如果你显著提高书籍、论文和规范文档的比例，模型通常会获得更好的长文结构感和更稳定的书面表达；如果你上调代码和数学推理语料，结构化输出与形式推理往往会改善，但普通对话风格可能会被拖偏。

The Pile 这类多源高质量语料的价值，就在于它展示了：数据多样性本身是一种能力设计 (Gao et al., 2020)。

如果把 mixture 写成优化目标的一部分，它的工程含义会更清楚。令 π_i 表示第 i 个数据桶的采样概率，$\sum_i \pi_i = 1$，则整体经验风险可以写成：

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} \pi_i \, \mathbb{E}_{x \sim D_i} \left[-\log p_\theta(x) \right]$$

对应的梯度也是加权求和：

$$\nabla_\theta \mathcal{L}(\theta) = \sum_{i=1}^{n} \pi_i \, \mathbb{E}_{x \sim D_i} \left[-\nabla_\theta \log p_\theta(x) \right]$$

这就是为什么调 mixture 不是“换一道配菜”，而是在直接改变梯度期望本身。

为了把“mixture 改动 $\rightarrow$ 能力改动”压缩成一张记忆卡，可以用下面这张表：

mixture 变化	语言流畅性	代码能力	知识密度	风格稳定性	典型风险
网页占比上升	中性到提升	中性	下降或分散	波动变大	噪声、模板、SEO 文本
书籍 / 规范文档占比上升	提升	中性	提升	更稳定	覆盖面收窄
代码占比上升	风格更技术化	显著提升	结构知识提升	口语稳定性下降	对话风格漂移
学术占比上升	更正式	对数学 / 论文总结有利	提升	更稳定但更书面	普通对话变硬
对话 / 论坛占比上升	交互感更强	中性	取决于质量	口语更自然	事实质量不稳

3.2 为什么“更多代码”可能反而伤害对话质量

这件事最容易被误解。很多人会说：“代码也是高质量文本，为什么多一点代码会伤害模型？”答案不是“代码有害”，而是：代码语料会改变模型对‘什么样的续写更常见’的内部先验。

设想这样一个例子：

你原本训练一个通用对话模型，网页、书籍、文档、论坛、代码都比较均衡。后来为了提升编码能力，你把代码占比从 8% 提到 30%，而没有同步增加总 token 预算。结果可能出现三类变化：

1. 风格迁移
   模型更偏好短、紧、结构化、列表化甚至带代码围栏的回答，普通问答变得像“技术文档片段”。

2. 语义偏置
   用户问一个开放性问题时，模型更容易往“可执行方案”“伪代码”“函数接口”方向收缩，而不是给出人类沟通友好的解释。

3. 容量竞争
   在固定参数与固定 token 预算下，模型花更多容量去拟合代码分布，自然会减少对普通自然语言细腻表达的建模。

所以，代码不是不能多，而是必须带着目标配比和产品目标来多。如果你的产品是代码助手，这可能是正确决策；如果你的产品是客服、教育或企业知识助手，这个配比就未必合适。

3.3 如何为通用 LLM 设计数据混合

对通用 LLM，更稳健的 mixture 设计流程通常不是“凭经验设百分比”，而是遵循下面五步：

第一步：先定义目标能力，而不是先定义来源

你想要的是更强的聊天、代码、数学、检索理解、长文总结，还是跨语言覆盖？不同目标对应的最优 mixture 完全不同。
先定义能力向量，再映射到数据桶，而不是先搜集到什么就喂什么。

第二步：按功能划桶，而不只是按来源划桶

来源划分（网页、书籍、代码、论文）很有效，但还不够。更可操作的是再按功能切分：

- 通用知识与百科
- 高质量长文与正式写作
- 代码与技术文档
- 数学与符号推理
- 多语言文本
- 任务形式丰富的问答/说明类文本

这样做的好处是：当某项能力表现不足时，你能更准确地调权，而不是粗糙地说“多加一点网页”。

第三步：先用小模型做 **mixture** 试验，再决定大模型预算

Llama 3 的报告公开了一个非常值得借鉴的思路：先通过知识分类与缩放实验，在小模型上比较不同 mixture，再把结果外推到更大模型 (Grattafiori et al., 2024)。这比“直接拿大模型赌一个配比”要稳健得多，也便宜得多。

第四步：用切片验证，而不是只看总 **loss**

一个通用模型的好坏，不能只看单一验证集的整体 loss。你至少需要对以下切片分别观测：

- 通用网页与百科
- 代码
- 数学/推理
- 长文写作

- 多语言
- 安全/敏感领域文本

只有这样，你才能知道一个新 mixture 到底是在“整体提升”，还是在用某一类数据牺牲另一类能力。

第五步：给后续 **CPT** 留出空间

通用预训练阶段不是解决所有专业能力缺口的唯一机会。很多看似应该在预训练里狂加的数据，更适合在中训练阶段用更可控的方式补。
一个成熟的设计不是“什么都在预训练阶段做完”，而是知道哪些能力应当早做，哪些能力应当后移。

把这个框架落到 7B 运行实例里，可以得到一个更具体的设计草图：

数据桶	占比	主要收益	主要风险
网页与通用文档	68%	覆盖广、知识面大	噪声高、重复多
书籍	12%	长文结构、书面表达	领域覆盖偏差
代码	10%	代码与结构化推理	对话风格技术化
学术	6%	知识密度、正式表达	文风变硬
对话与论坛	4%	交互语气、问答形式	事实质量不稳

这类表的价值不在于具体百分比，而在于它强迫你把每一份 **token** 预算都映射到能力收益和副作用。

3.4 领域性能上升、通用基准下降，通常说明了三件事

当你加入领域数据后发现领域指标提升、通用指标下降，这往往不是偶然，而是至少发生了下面三件事中的一种：

1. 分布偏移

模型把更多容量和更多训练步数花在了更窄的领域分布上，于是对通用分布的拟合速度被拖慢，甚至被挤压。

2. 评估目标改变了

你的领域 benchmark 可能更接近新增数据，而通用 benchmark 离它更远。于是模型对真实产品价值未必变差，但对旧有基准的适应性下降了。

3. 数据质量或 **tokenization** 不友好

有时问题并不在“领域数据太多”，而在“领域数据很碎”。例如术语被 tokenizer 切得过细、文档充满模板重复、公式或表格解析质量差，那么你虽然加了领域数据，真正提供给模型的高价值信息却没有想象中那么多。

修复这类问题，通常有四种路径：

- 降低领域数据权重；
- 增加总 token 预算，而不是只改比例；

- 引入通用回放（general replay）保持底座分布；
- 把领域强化从预训练阶段后移到 CPT。

这里最重要的判断是：
你是在做通用底座，还是在做偏专业底座。
混合比例没有脱离产品目标的“客观正确值”。

3.5 课程学习：不是“由易到难”，而是“按训练阶段重新分配预算”

预训练中的课程学习，常被说成“先学简单，再学复杂”。这个说法太粗了。对预训练更实用的理解是：

> 课程学习是在训练的不同阶段，改变数据分布、序列长度或样本难度，以换取更好的收敛效率与最终能力。

常见的课程化方式有三种：

质量优先课程

训练早期先用更干净、更高质量的文本，让模型尽快建立稳定语言统计；中后期再逐步加入覆盖更广的通用文本。这种做法的收益，通常是早期 loss 更稳、更可预测。

长度课程

先用较短序列训练，提高吞吐；待基础模式学稳后，再拉长上下文窗口。这在长上下文训练里尤其常见，因为序列长度会直接影响训练成本。

域 **annealing**

在训练后期，以较低学习率上调某些高价值桶的比例，例如高质量代码、数学推理或长文文档，以换取特定能力上的边际提升。Llama 3 报告了利用少量高质量代码与数学数据进行 annealing 的做法，并强调不把常用 benchmark 的训练集直接塞进 annealing 数据，以避免评估失真（Grattafiori et al., 2024）。

课程学习的边界也必须讲清楚：
它不是“总能变好”的万能技巧。课程一旦设计错误，本质上就是在错误的阶段重新分配宝贵的 token 预算。

3.6 如何估算一个新数据源的边际价值

真正成熟的预训练团队，不会因为“这个数据源看起来不错”就直接把它扔进大训练。更稳妥的做法是先估算它的边际价值。一个好用的评估框架至少包括以下代理指标：

1. 接受率与净增益

新数据源经过质量过滤、PII 过滤、许可证筛查和去重之后，还剩多少有效 token？
如果一个源表面上有 500B token，过滤后只剩 20B，还高度重复，那它的真实边际价值就很有限。

2. 去重后独特信息密度

看近重复率、长公共子串比例、模板占比。很多看起来“规模很大”的新源，真正新增的独特信息很少。

3. tokenizer 友好度

看碎片率、平均 token 长度、关键术语切分情况。对代码、法律、医学、表格混排文本，这一点尤其关键。

4. 小模型 proxy 实验

在小模型或短跑训练里，比较加入与不加入该数据源后的：

- validation loss 变化；
- 领域切片 loss 变化；
- 小型下游 benchmark 变化；
- 安全与记忆化风险变化。

5. annealing 试验

对已有中途 checkpoint，用很小学习率在该源上做短程 annealing，看它是否能带来低成本、可重复的收益。Llama 3 也明确把 annealing 当成判断小型领域数据价值的工具 (Grattafiori et al., 2024)。

判断一个数据源值不值得，不是问“它大不大”，而是问：

> 它是否在可接受风险下，提供了足够多此前没有、且与你目标能力高度相关的 **token**。

3.7 本节结论

1. 数据混合本质上是能力设计，不是仓库管理。
2. 更多某类数据会改变模型的内部先验，因此“更多代码”“更多数学”永远不是无条件的好事。
3. 领域数据抬高领域指标、拉低通用指标，通常说明分布偏移、容量竞争或 **tokenizer**/数据质量问题正在发生。
4. 最可靠的 **mixture** 设计方式，不是拍脑袋，而是先做小模型实验、切片评估和边际价值估计。

而一旦 mixture 定下来，问题就从“训练哪一种世界”变成“用多少 FLOPs 去训练这个世界”。这就进入参数、token 与 compute budget 的配平。

4 计算与扩展

! 本节先想这些问题

1. 预训练的主要成本驱动因素是什么（参数、上下文、token 数、精度）？
2. 解释 Chinchilla 缩放定律——参数与 token 的计算最优比例是什么？
3. 粗略估算：训练一个 70B 模型在 2T token 上需要多少 FLOPs？
4. 模型参数翻倍大约会使每个 token 的 FLOPs 怎样变化？

5. 解释数据并行、张量并行和管线并行。各在什么时候使用？
6. 什么是 ZeRO / FSDP？它如何减少内存？
7. 什么是激活检查点（activation checkpointing）？权衡是什么？
8. 解释混合精度训练（BF16/FP16）。为什么 BF16 在 LLM 中占主导地位？
9. 什么是“屋顶线模型”（roofline model）？它如何帮助推理 GPU 利用率？
10. 估算预训练一个 7B 参数模型在 1T token 上所需的最少 H100 GPU 数量和时钟时间。说明你的假设。
11. 什么是序列并行？它如何配合张量并行进行长上下文训练？
12. 解释 Megatron-LM 的 3D 并行与 DeepSpeed ZeRO-3 的区别。各在什么时候使用？

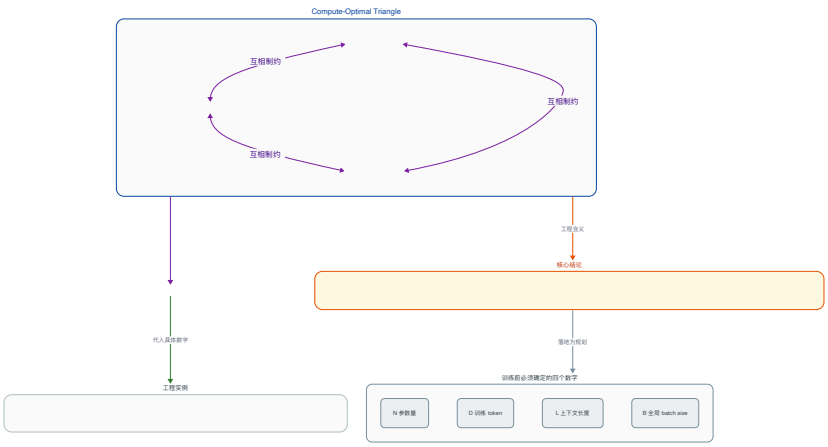

Figure 3: 计算三角——N×D×C 的相互制约

4.1 预训练为什么贵：四个主旋钮同时在转

预训练的昂贵，不来自单一因素，而来自四个旋钮同时变化：

1. 参数规模 N：决定模型容量，也决定权重、梯度和优化器状态的内存；
2. 训练 **token** 数 D：决定模型总共能看见多少内容；

3. 上下文长度 L：直接影响 attention 成本、激活内存和打包效率；
4. 数值精度与训练状态：影响显存、带宽、稳定性和有效吞吐。

如果只盯着参数量，很容易误判。一个 70B 模型和一个 7B 模型，差异不只是“十倍参数”，还包括更大的优化器状态、更重的通信压力、更难的并行切分，以及通常更高的训练 token 需求。与此同时，上下文长度一旦增加，attention 和激活开销也会快速上升。

从系统角度看，真正拉高账单的不只是理论 FLOPs，还包括：

- 通信同步；
- padding 与 packing 浪费；
- dataloader 和存储带宽；
- checkpoint 写盘；
- 故障恢复与重跑；
- 为了稳定性留出的保守配置。

所以，讨论“预训练成本”时，不能只问“模型多大”，而要问：你打算把它训到什么程度，用什么长度训，训多久，以及系统效率有多高。

4.2 一个足够好用的训练 FLOPs 估算式

对稠密 decoder-only Transformer，训练 FLOPs 的一个常用粗估是：

$$\text{FLOPs} \approx 6ND$$

其中：

- N 是参数量；
- D 是训练 token 总数。

这是一个工程上非常好用的数量级公式。它忽略了不少常数项和系统开销，但足够帮你做预算级判断。

例如：

70B 模型，2T token

$$6 \times 70 \times 10^9 \times 2 \times 10^{12} = 8.4 \times 10^{23} \text{ FLOPs}$$

也就是说，量级上大约是 8.4×10^{23} **FLOPs**。

7B 模型，1T token

$$6 \times 7 \times 10^9 \times 10^{12} = 4.2 \times 10^{22} \text{ FLOPs}$$

量级上约为 4.2×10^{22} **FLOPs**。

要特别强调：这只是理想化训练计算量，还没有把通信、重算、checkpoint、空转和输入管线瓶颈算进去。现实 wall-clock 往往比“用理论 FLOPs 直接倒推”更保守。

更接近 operator 视角的 wall-clock 估算通常写成：

$$T_{\text{wall}} \approx \frac{6ND}{n_{\text{gpu}} P_{\text{peak}} \eta_{\text{MFU}}}$$

其中 n_{gpu} 是 GPU 数，P_{peak} 是单卡理论峰值吞吐，η_{MFU} 是有效 Model FLOP Utilization。若把每一步实际消费的 token 数记作 $B^{\text{tok}}_{\text{global}}$，则训练步数可粗估为：

$$S \approx \frac{D}{B^{\text{tok}}_{\text{global}}}$$

在 7B 示例里，若 $D = 10^{12}$ token、$B^{\text{tok}}_{\text{global}} \approx 8.4 \times 10^6$ token/step，则总步数约为

$$S \approx 1.19 \times 10^5 \text{ steps}$$

若使用 512 张 H100、$P_{\text{peak}} \approx 989$ TFLOPS（BF16 峰值）且 $\eta_{\text{MFU}} \approx 0.38$，则理想化纯计算 wall-clock 大约是几天量级；再叠加 I/O、checkpoint、故障恢复和 padding 损失，真实 wall-clock 往往会进一步拉长。

4.3 Chinchilla 改变了什么：不是“越大越好”，而是“别把大模型喂得太少”

Kaplan et al. 的 scaling law 说明，语言模型的 loss 会随着模型大小、数据量和计算量的增加而呈现平滑的幂律下降（Kaplan et al., 2020）。这给了行业一个非常重要的信心：扩大规模通常会带来收益，而且收益变化是可预测的。

但 Hoffmann et al. 进一步指出，很多大型模型其实都训练不足：参数堆得很大，却没有看到足够多的 token（Hoffmann et al., 2022）。这就是 Chinchilla 的核心修正。它关心的不是“参数越大越好”，而是：

> 在固定计算预算下，参数规模和训练 **token** 应该怎样配平，才最划算。

类似找对象：时间和精力都是固定预算，不能只想着找一个“配置很高”的人，还得看你们有没有足够的相处、了解和磨合。最划算的，不是标准越高越好，而是在有限投入下，把“人有多合适”和“相处有多充分”配平。

在 Chinchilla 论文的实验设定下，一个非常常见的经验近似是：
compute-optimal 区域大约对应每个参数 **20** 个左右的训练 **token**。

这个数字极有名，但更重要的是理解它的边界：

- 它是经验规律，不是自然常数；
- 它依赖数据质量、优化器、训练配方和架构细节；
- 它是一个很好的起点，但不保证对所有后续模型都精确成立。

Llama 3 的公开报告就显示，随着数据质量、配方和目标能力变化，超过“经典 Chinchilla 最优点”之后，继续增加高质量 token 仍可能带来收益（Grattafiori et al., 2024）。因此，工程上最成熟的态度不是盲信某个比例，而是把 Chinchilla 当成预算规划的第一版基线。

4.4 参数翻倍、上下文翻倍，分别会发生什么

对稠密模型而言，在保持大体相同架构比例时：

- 参数翻倍：每 token 的前向/反向计算量大致也会近似翻倍，权重与优化器状态内存也近似线性上升；
- 训练 **token** 翻倍：总训练成本近似翻倍；
- 上下文长度翻倍：attention 成本会显著增加，激活内存更快膨胀，长序列训练吞吐明显下降；
- 精度从 **FP32** 降到 **BF16/FP16**：显存与带宽压力下降，但数值稳定性管理会变得更重要。

这就是为什么预训练规划本质上不是一个二维表，而是一个四维权衡：
参数、**token**、上下文、精度缺一不可。

4.5 一台机器为什么不够：预训练天然会走向分布式

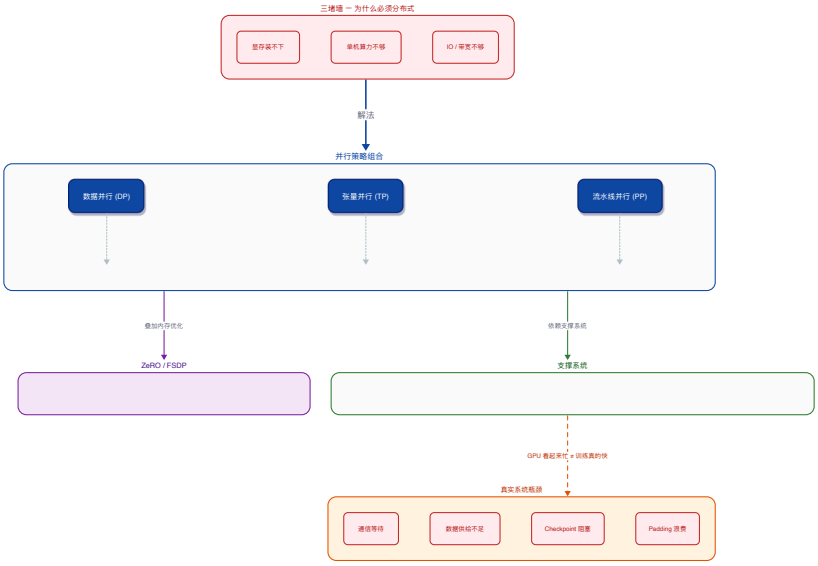

Figure 4: 分布式预训练拓扑

只要模型进入十亿参数级，单机就会很快碰到三堵墙：

1. 显存墙：装不下参数、梯度、优化器状态和激活；
2. 时间墙：单机训练 wall-clock 长到不可接受；
3. 带宽墙：单机读取、缓存和处理数据的能力跟不上目标吞吐。

于是，预训练问题很自然就从“怎么写一个 loss”转成“怎么把模型拆开、把状态拆开、把训练拆开”。

4.6 并行策略地图：DP、TP、PP、ZeRO/FSDP 与序列并行

可以把 parallelism 理解成：同一个训练问题，沿不同维度拆给多张 **GPU** 一起做。
设全局 batch 为 B，序列长度为 L，隐藏维度为 H，参数量为 P，设备数为 n。单卡训练主要受三类约束：

- 状态内存：参数、梯度、优化器状态，量级大致与 P 同阶；
- 激活内存：中间张量，量级常近似与 $B \times L \times H$ 成正比；
- 通信开销：设备之间同步参数、梯度或激活所花的时间。

五类并行，本质上就是在这三种瓶颈之间做交换。一个很实用的记法是：

DP 拆样本，**TP** 拆矩阵，**PP** 拆层，**ZeRO** 拆状态，**SP** 拆序列。

并行方式	切分什么	主要解决什么	主要代价
数据并行（DP）	样本	提高吞吐	梯度同步
张量并行（TP）	单层内部矩阵	单卡装不下单层算子	通信频繁
流水线并行（PP）	网络深度上的层段	单卡/单机装不下整模	pipeline bubble、调度复杂
ZeRO / FSDP	参数、梯度、优化器状态	降低冗余显存	通信更多、实现更复杂
序列并行（SP）	序列维度上的激活/操作	长上下文训练的激活压力	需要与 TP 协同设计

下面用一个统一例子来理解：
假设你要用 **8** 张 **GPU** 训练一个 **70B** 模型，全局 **batch** 为 $B = 1024$，上下文长度 $L = 8192$。
不同并行策略的区别，不是“谁更高级”，而是你到底在拆哪一个维度。

数据并行（**DP**）

DP 最直观：每张卡都放一份完整模型，只把样本分开。
如果有 n 张卡，那么每张卡处理

$$B_{\text{local}} = \frac{B}{n}$$

个样本；算完本地梯度 g_i 后，再做一次同步：

$$g = \frac{1}{n} \sum_{i=1}^{n} g_i$$

这就像 8 个厨师按照同一份菜谱，各做一批菜，最后再统一口味。
它的优点是简单、稳定、吞吐容易扩展；缺点也很直接：每张卡都必须装得下完整模型副本。如

果 70B 模型连一张卡都放不下，那么 DP 再多也没用。DP 解决的是“样本太多”，不是“模型太大”。

张量并行（**TP**）

TP 解决的是“单层都塞不下”。
以线性层

$$Y = XW$$

为例，如果输出维度太大，可以把权重矩阵按列切开：

$$W = [W_1, W_2, \dots, W_n]$$

于是第 i 张卡只算自己的那一块：

$$Y_i = XW_i$$

最后再把各卡结果拼接或归约回来。
直觉上，DP 是“每个人做一整道题的不同样本”，TP 则是“几个人一起做同一道大题的不同小问”。
它特别适合 attention 和 FFN 里那些超大的 GEMM，但代价是：每一层都要通信。所以 TP 很吃高速互联；卡间带宽不够时，算子本身没变慢，系统却会被通信拖住。

流水线并行（**PP**）

PP 解决的是“整模装不下单机”，做法是按层深切开。
如果模型有 80 层，分成 4 段，那么每段大约负责 20 层。第 1 组 GPU 跑前 20 层，第 2 组接着跑 21–40 层，以此类推。
它很像工厂流水线：不是每个人都造整辆车，而是有人装底盘、有人装车门、有人装电路。

问题在于流水线不会天然满载。为了让后面的 stage 不空等，通常要把一个 batch 再切成多个 micro-batch。若流水线有 p 段，micro-batch 数为 m，则 bubble 比例近似为：

$$\text{bubble} \approx \frac{p-1}{m+p-1}$$

所以 micro-batch 太少，前后段就会频繁“发呆”；micro-batch 太多，又会增加调度复杂度和激活保存压力。PP 的难点不在概念，而在把流水线真正填满。

ZeRO / FSDP

ZeRO / FSDP 不是在拆样本、拆矩阵或拆层，而是在拆训练状态本身。
传统 DP 最浪费的地方是：每张卡都保存完整参数、完整梯度、完整优化器状态。若把它们分别记为 P, G, O，那么单卡状态内存近似是：

$$M_{\text{naive}} \approx P + G + O$$

而在理想分片下，n 张卡每张只保留其中一部分，单卡大致变成：

$$M_{\text{shard}} \approx \frac{P+G+O}{n}$$

这就像以前 8 个人每人都背一整套工具箱，现在改成每人背其中一部分，用到时再互相传。
它显著降低了冗余显存，但代价是更多的 all-gather、reduce-scatter 和更复杂的实现语义。
FSDP 可以看作这一思路在现代框架里的主流工程化形式：平时分片保存，需要算某层时再临时聚齐。

序列并行（**SP**）

当上下文从 4K 拉到 32K、64K 甚至更长时，真正爆炸的常常不是参数，而是激活。
若激活张量写成

$$A \in \mathbb{R}^{B\times L\times H}$$

那么仅从形状上就能看出，长度 L 一变大，内存会线性上升。SP 的想法是：既然 TP 已经在“宽度”上拆了 hidden dimension，那我们也可以在“长度”上拆，把序列维度分到多张卡上。若按序列均匀切到 n 张卡，则单卡激活近似降为：

$$\frac{B\times L\times H}{n}$$

直觉上，它像是把一卷很长的书卷分给 8 个人，每人只读其中一段，而不是每个人都摊开整卷。
SP 往往和 TP 一起使用，因为长上下文训练里，真正贵的是“宽度上的大矩阵”和“长度上的大激活”同时存在。TP 主要减轻单层算子压力，SP 主要减轻长序列激活压力。

把这五种策略放在一起看，就会更清楚：
并行训练不是单纯“多加几张卡”，而是决定你究竟在样本、矩阵、层、状态还是序列上动刀。

4.7 Megatron 3D 并行 vs ZeRO-3/FSDP：不是替代关系，而是不同层次的问题

很多人第一次接触这些技术时，会问：“Megatron 的 3D 并行和 ZeRO-3 到底谁更好？”
更准确的答案是：它们解决的问题并不完全相同。

Megatron-LM 的“3D 并行”通常指 DP + TP + PP 的组合，用来把计算图和模型结构本身拆开 (Shoeybi et al., 2019)。
ZeRO-3 / FSDP 则是把训练状态本身拆开，减少冗余副本 (Rajbhandari et al., 2019)。

因此：

- 当单层都装不进一张卡时，你通常必须引入 TP；
- 当整模装不进单机时，PP 往往也会进入方案；
- 当完整副本太浪费时，ZeRO-3/FSDP 极有价值；
- 真正的大训练里，经常是这些策略组合使用，而不是二选一。

4.8 激活检查点：用重算换显存

activation checkpointing 的核心思想很直接：前向传播时不保存全部中间激活，而是在反向传播时按需重算一部分。这样做能显著降低显存压力，让更大的 batch、更长的上下文或更大的模型成为可能。

权衡也很明确：

- 优点：省显存，常常是训练能否启动的关键；
- 代价：增加额外计算，拉长单步时间。

这类技术很像工程上的“折中阀门”——它不会让训练更便宜，但它经常能让本来无法训练的配置变成可训练。

4.9 混合精度：为什么 BF16 成了主流起点

LLM 训练几乎不再使用全程 FP32。原因很简单：显存和带宽都太贵。混合精度训练通过让大部分张量使用 16-bit 表示，显著降低内存占用和传输成本。

在 BF16 与 FP16 之间，现代大模型训练通常更偏好 BF16，原因不是它“更先进”，而是它更容易稳定。BF16 保留了与 FP32 相同的指数范围，因此在大规模训练中更不容易因为数值范围不足而出现溢出或下溢。
工程上的含义是：你可以把更多精力放在训练本身，而不是花大量时间对抗数值事故。

4.10 用 roofline 看系统瓶颈：你的训练到底是在算，还是在等

roofline model 提供了一个很有效的视角：把 kernel 或训练任务的性能上限，看成由两条天花板共同决定——计算峰值和内存带宽峰值。某段工作负载如果算术强度不高，就更可能受内存带宽限制；如果算术强度很高，就更可能逼近计算峰值。

这对预训练很有帮助，因为它能把“GPU 利用率不高”进一步拆开看：

- 是算子本身不够饱和？
- 是 HBM 带宽卡住了？
- 是通信在拖慢？
- 还是数据根本喂不饱？

很多训练系统看起来 GPU 很忙，但真实 MFU (Model FLOP Utilization) 并不高；而 roofline 视角能帮助你识别：瓶颈究竟在算、在存、还是在通信。

4.11 GPU utilization 很高，但 tokens/s 很差：通常不是算子，而是有效 token 比例出了问题

在 7B 示例里，某次训练看板显示 GPU utilization 持续高于 90%，但实际 tokens/s 比计划低了近 28%。最终定位到的问题不是 matmul 太慢，而是三件事叠加：packing 粒度差、shard 不均导致 dataloader 抖动，以及恢复后 data cursor 错位带来的重复读取。更有效的 operator 指标不是单独看 utilization，而是看：

$$\rho_{\text{effective}} = \frac{\text{useful tokens/s}}{\text{loaded tokens/s}}$$

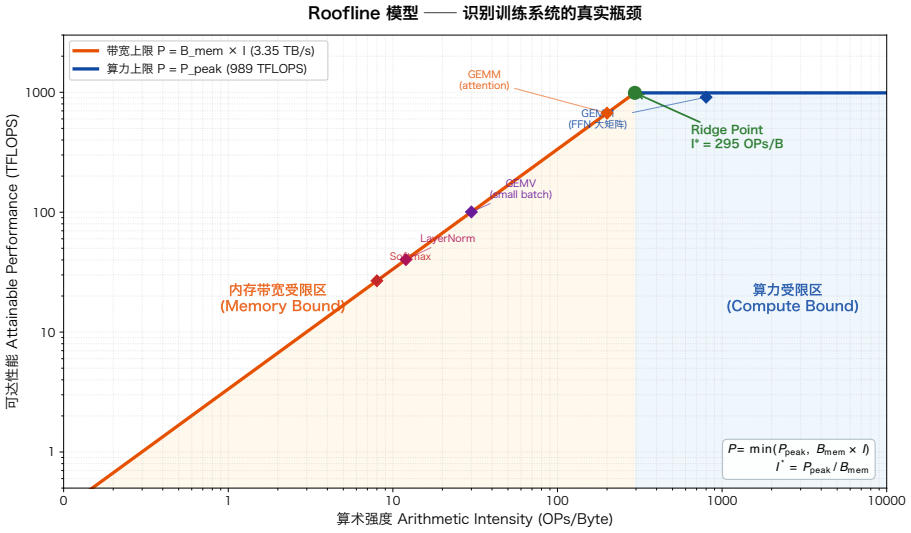

Figure 5: Roofline 模型——识别训练系统的真实瓶颈

当 $\rho_{\text{effective}}$ 明显下降时，优先排查的通常是：1. packing 质量与 padding 浪费；2. dataloader / storage 带宽；3. shard 分配与 worker 负载均衡；4. 是否发生了重复读取或 stale sample replay。

总之，关注有效 token 预算和有效 tokens/s

4.12 一个粗略的 H100 估算：7B 模型、1T token 要多少卡、多久、多少钱

还是用前面的粗估：

假设你训练一个 7B 稠密模型、总 token 为 1T。再假设单张 H100 在端到端训练中能达到大约 350–450 TFLOPS 的有效吞吐（注意，这是工程假设，不是硬件峰值），那么：

- 256 张 H100：纯计算时间约 4.2–5.4 天
- 512 张 H100：纯计算时间约 2.1–2.7 天
- 1024 张 H100：纯计算时间约 1.1–1.4 天

把这件事换算成钱，关键不是“多少张卡”，而是“总卡时（GPU-hours）”。上面几种配置对应的总卡时大致都在 2.6 万–3.3 万 H100-hour 量级。若按 $3–$4 / H100-hour 粗估，那么这次训练的纯 GPU 成本大约在 $80k–$130k。注意，256、512、1024 张卡的纯 GPU 账单通常不会差出一个数量级，因为总 FLOPs 基本没变；更多卡主要是在换更短的 wall-clock。

但这还只是“理想纯算时间”。现实 wall-clock 往往还要再加上：

- 数据加载与 packing；
- 通信与同步；

- checkpoint 保存；
- 故障恢复；
- 安全余量和效率损失。
- 以及存储、网络、失败重跑等额外开销。

因此，粗估 GPU 数量、时钟时间和训练费用时，一定要把系统效率假设写明白。否则，所有估算都会显得比现实乐观。

4.13 本节结论

1. 预训练成本由参数、**token**、上下文和精度共同决定。
2. $6ND$ 是很好用的第一版 **FLOPs** 估算式，但现实 **wall-clock** 总会更贵。
3. **Chinchilla** 的核心不是一个神奇常数，而是“别把大模型喂得太少”。
4. 大模型训练从根上就是分布式系统问题；**TP**、**PP**、**ZeRO/FSDP**、**SP** 解决的是不同层次的瓶颈。
5. 做预算时，不仅要算理论 **FLOPs**，更要算系统效率。

而当预算被定义出来之后，下一步就不是继续写更复杂的公式，而是让这些 FLOPs 在真实集群上稳定落地。这就把问题带到训练配方与监控。

5 训练配方与监控

! 本节先想这些问题

1. LLM 预训练的典型学习率调度是什么？
2. Loss 在训练中途飙升 3 倍然后缓慢恢复——你如何排查？
3. 预训练中最常见的 5 种故障模式是什么（不稳定、污染、记忆化、分布偏移、安全退化）？
4. 你如何设置“探测集”来监控预训练期间的质量？
5. 训练 Loss 在下降但下游评估分数停滞——出了什么问题？
6. 何时决定停止预训练？你关注哪些信号？
7. 什么是梯度范数监控？梯度范数突然飙升意味着什么训练问题？你应该怎么做？

5.1 训练配方不是“超参数列表”，而是稳定性边界

到了大规模预训练阶段，训练配方的作用已经不是“微调一点效果”，而是决定这次训练是不是能稳定活过第一周、第二周，以及整个月。配方中任何一个环节——学习率、global batch、warmup 长度、weight decay、精度格式、梯度裁剪、检查点频率——都可能改变训练是否发散、是否出现长时间 loss 振荡、是否在一次集群故障后无法恢复。

因此，预训练配方更像一个稳定性边界设计问题。你真正想构建的不是“一个最好看的 config 文件”，而是：

- 可以规模化运行；
- 失稳时能被尽快发现；
- 出错后能被回滚；
- 变化原因可被解释；
- 训练结束后可复盘的系统。

5.2 典型学习率调度：warmup 后再衰减，而不是开局就冲顶

现代 LLM 预训练里，最常见的学习率调度仍然是：

1. **warmup**：从较小学习率平滑上升到峰值；
2. 主训练阶段：维持一段稳定区间或直接开始缓慢衰减；
3. 衰减阶段：常见是 cosine decay 或 linear decay，直到训练结束。

为什么 warmup 几乎总是必要？因为训练早期同时具备三个危险条件：

- 参数还没进入稳定区域；
- 动量统计还不可靠；
- 大 batch 与低精度下更容易出现数值振荡。

一开始就用很大学习率，loss spike、梯度爆炸、NaN 和不可逆发散常常来得非常快。
这也是为什么大模型训练里，学习率调度不只是“收敛速度技巧”，它首先是风险管理工具。

5.3 一条现代预训练配方通常长什么样

虽然不同团队的默认值不同，但一条相对稳健的现代预训练配方，往往会包含以下组件：

- 优化器：AdamW 仍然是最稳妥的主流起点。它把 weight decay 与梯度更新解耦，对大规模自适应优化更友好 (Loshchilov & Hutter, 2017)。
- 数值格式：BF16 混合精度，辅以关键统计和主权重的稳定维护。
- 学习率调度：短 warmup + 长衰减，常以 token 而不是 epoch 为单位。
- 梯度裁剪：限制异常 batch 带来的单步爆炸。
- 激活检查点：用重算换显存，提升可训练规模。
- 全局 **batch** 设计：通过 DP、梯度累计和微批大小共同决定。
- 打包与数据顺序控制：减少 padding，避免训练 token 和“有效 token”偏差过大。
- **checkpoint** 与回滚机制：确保训练出问题时不是“从头再来”。

重要的是：这些组件不是彼此独立的。
例如，你调整了 global batch，学习率未必还能不变；你切换了上下文长度，激活内存和吞吐会同时改变；你换了精度，稳定性边界也会变化。好的配方从来不是一串孤立开关，而是一组互相耦合的系统参数。

对于 global-norm clipping，一个常见写法是：

$$\tilde{g} = g \cdot \min\left(1, \frac{\tau}{\|g\|_2}\right)$$

其中 τ 是 clipping threshold。这个公式很朴素，但它揭示了训练配方的一个核心思想：很多“高级技巧”本质上是在控制异常步对整个优化轨迹的破坏半径。

5.4 Operator 视角：一张训练看板至少要看什么

如果把训练配方翻译成值班工程师每天真正盯的东西，一张合格的看板至少应包含下表：

指标	健康信号	红旗信号	常见根因	第一动作
train loss	平滑下降	突然 spike / 持续振荡	LR、坏 batch、恢复错误	对齐时间线
val PPL	稳定下降或平台	无改善 / 与 train 分叉	过拟合、污染、mixture 失衡	看切片验证
grad norm	带噪但有界	突然飙升或周期性抖动	溢出、极端样本、LR 过高	先保命，再回放
tokens/s	接近计划值	持续低于计划	packing 差、I/O 饥饿、通信	看 step breakdown
MFU	稳定在目标区间	utilization 高但 MFU 低	kernel / 通信 / 数据喂不饱	对照 roofline
checkpoint age	周期性刷新	太久未落盘或恢复异常	存储抖动、写盘阻塞	检查恢复路径
slice eval	关键切片同步改善	某域持续回落	mixture、tokenizer、污染	先定位到桶

在 7B 示例里，真正有效的看板不是“图很多”，而是这些指标能否被同一时间线对齐：loss spike 是不是恰好发生在恢复之后？tokens/s 下降是不是恰好发生在 packing 策略改变之后？没有统一时间线，看板只是漂亮噪声。

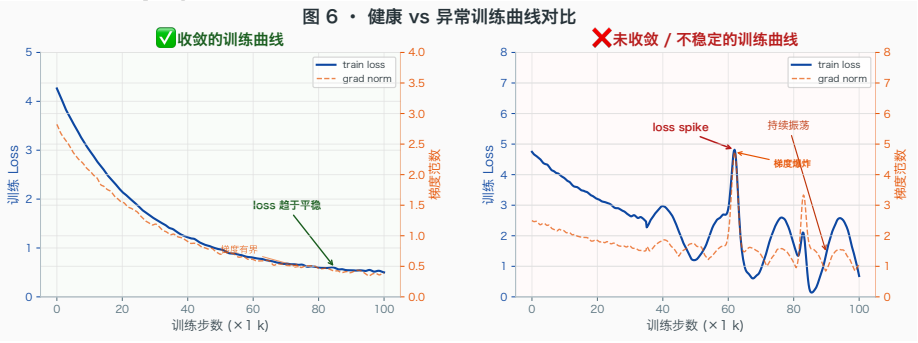

Figure 6: 健康 vs 异常训练曲线对比

5.5 Loss 中途飙升 3 倍又慢慢恢复，怎么排查

这是预训练里非常典型、也非常棘手的故障。它说明问题可能是暂时性的，但不代表可以忽视。更稳妥的排查路径通常是：

第一步：对齐时间线

先确认 spike 发生时，系统里是否有任何“刚刚改了什么”：

- 学习率是否刚切换阶段；
- 序列长度是否刚变；
- mixture 是否刚重配；
- 是否刚恢复自 checkpoint；
- 是否有节点重启、通信异常或数据分片切换。

很多故障不是“突然发生”，而是配置或运行状态变化刚好在那一刻显化。

第二步：看梯度范数与数值日志

如果 loss spike 同时伴随 grad norm 突然飙升，优先怀疑：

- 学习率过高；
- 坏 batch；
- 精度溢出；
- 梯度同步异常。

如果 grad norm 正常，但 loss 异常，优先看数据内容、标签错位、packing bug 或恢复逻辑。

第三步：回放异常 **batch**

如果你能定位到那一步的 batch，就尝试离线重放。
很多隐藏很深的问题——例如 tokenizer 对某类文本切分异常、某个 shard 中混入了破损样本、某类极长样本造成 attention 数值问题——只有在回放时才会暴露。

第四步：判断它是偶发还是结构性

- 只发生一次，后面恢复：可能是坏样本、节点抖动或临时通信问题；
- 周期性发生：多半与学习率调度、特定数据桶、checkpoint 恢复流程有关；
- 越来越频繁：说明系统正在接近失稳边界，不能因为“它还能恢复”就继续赌。

工程上最危险的误判，是把“恢复了”误当成“没事”。

5.6 恢复语义：如何证明一次 resumed run 等价于一次 uninterrupted run

一次训练在 step t 的“完整状态”从来不只是参数 θ_t。更完整地，它至少包含：

$$\mathcal{S}_t = (\theta_t, m_t, v_t, \mathrm{rng}_t, \pi_t, \sigma_t)$$

其中：

- m_t, v_t 是 Adam 一阶 / 二阶动量；

- rng_t 是随机数状态；
- π_t 表示数据游标、shard 顺序、packing 状态与样本重排位置；
- σ_t 表示学习率调度器、loss scaler、梯度累计状态等。

因此，“恢复成功”不等于“权重文件能读出来”。更严格的语义等价要求是：在相同输入 shard 和相同随机状态下，resume 之后的若干步 loss、grad norm 和更新方向应与 uninterrupted run 仅存在浮点级微小偏差。

> ⚠ 运行实例：一次不完整恢复导致的隐蔽退化
>
> 在 7B 示例里，某次 checkpoint 恢复后，loss 很快回到原水平，因此最初被误判为“恢复成功”。但几百 step 之后，grad norm 噪声显著放大，tokens/s 也略有下降。最终定位到的问题是：权重和优化器状态恢复了，但 data cursor 与梯度累计状态没有完整恢复，导致后续若干 step 的样本顺序与有效 batch 语义发生偏移。
> 修复原则只有一个：把 **resume semantics** 当成一等系统特性，而不是训练脚本的附属功能。

5.7 预训练中最常见的五类故障模式

把故障按“归因层”来分，比按“表面现象”来分更有效。最常见的五类故障模式如下：

故障模式	典型表象	根因层
不稳定	loss spike、NaN、grad norm 爆炸	优化与数值
污染	benchmark 异常好看、离线评估失真	数据与评估
记忆化	逐字复述长片段、隐私泄漏	数据与去重
分布偏移	某域变强、通用退化，或相反	数据混合
安全退化	敏感回答更激进、拒答边界漂移	数据治理与后续对齐接口

这个表背后的核心思想是：
先判断问题属于数据、优化、系统还是评估，再决定排障路径。
否则，很容易在错误层面上花很多时间。

5.8 探测集不是“大 benchmark 缩小版”，而是训练期的早期预警系统

训练期间的评估，不能只靠一个 validation loss。你需要一个小而稳的“探测集”系统，用来提前暴露不同方向上的质量变化。

一个好的探测集，通常具有五个特点：

1. 规模小，成本低：可以高频运行，不拖慢训练；
2. 覆盖关键切片：至少包含通用知识、代码、数学、长上下文、语言/领域切片、安全边界；
3. 污染风险低：最好是内部构造、严格去重、可追踪来源；
4. 信号稳定：不要过度依赖温度敏感、评分波动大的指标；
5. 与产品相关：不是只测“学术上看起来重要”的任务，也测“产品上真正在意”的 proxy。

最好的探测集设计，往往不是追求覆盖一切，而是覆盖那些一旦退化你就真的会后悔没有早点发现的能力。

5.9 训练 loss 下降，但下游分数停滞，通常说明模型在“学容易学的东西”

这是预训练里非常容易误判的一类情况。loss 继续下降，说明模型仍在更好地拟合训练分布；但下游分数不涨，说明这些新增拟合未必转化为你关心的能力。

常见原因有四种：

1. 验证集与下游任务分布不一致
   你在“网页平均文本”上继续变好，不代表在代码、数学、长文或工具使用代理任务上会同步变好。
2. 新增 **token** 的信息密度不高
   模型可能在反复学习更多“容易预测的 token”，比如模板化网页、重复短语、形式结构，而不是学到新的知识模式。
3. **tokenizer** 或上下文设置限制了收益传导
   某些领域即使加入更多数据，如果切分很碎、长度被截断严重，收益也很难真正转化为能力。
4. 已经进入边际收益递减区间
   这不是说不能继续训练，而是说“继续花钱”的理由必须更明确。

所以，loss 从来不是答案本身，它只是在告诉你：模型仍在学习；至于学到的是不是你想要的东西，要靠切片评估与探测集来判断。

5.10 梯度范数：最廉价、也最常被忽视的地震仪

梯度范数监控的价值，在于它比很多高层指标更早暴露风险。grad norm 的突然飙升，通常意味着以下某类问题之一：

- 学习率超出当前稳定边界；
- 混合精度溢出或数值异常；
- 某个 batch 异常，导致梯度极端；
- 训练状态恢复不一致；
- 数据分布突然切换，引入了极端难样本。

应对上，通常遵循“先保命，再定位”的顺序：

1. 先触发梯度裁剪或自动保护；
2. 保存并标记异常状态；
3. 检查是否与 LR、序列长度、数据桶切换同步发生；
4. 回放异常 batch；
5. 必要时回滚到安全 checkpoint，再做小步试探。

如果你只能多加一个训练监控图，很多时候最值得加的不是第十张 fancy dashboard，而是一张可靠的 **grad norm over time**。

5.11 何时停止预训练：不是“loss 不降了”，而是“继续训练还值不值”

预训练停止，几乎从来不是单信号决策。更合理的停止标准，通常同时看：

- 计划 token budget 是否已经用完；

- 验证 loss 是否明显放缓；
- 关键切片是否已基本平台期；
- 探测集是否不再提升，甚至出现回归；
- 继续训练的单位成本，是否已经高于可接受边际收益；
- 数据是否开始重复使用过多。

换句话说，停止训练并不等于“模型学不动了”，而是一个更现实的工程判断：

> 在当前预算、当前数据和当前目标下，继续训练是否仍然值得。

5.12 本节结论

1. 训练配方是稳定性边界，不是一组孤立超参数。
2. **warmup** + 衰减是主流，不是习惯动作，而是大规模训练的风险控制。
3. **loss spike**、**grad norm** 飙升、下游停滞都必须放回“数据—优化—系统—评估”四层里定位。
4. 探测集的使命不是替代大评估，而是尽早发现你最不希望发生的退化。
5. 停止预训练是成本—收益决策，而不是盯着一条 **loss** 曲线等它贴地。

> **AI 工程师的 60 秒检查顺序**
>
> 1. 先看 **train loss / val PPL / grad norm** 是否在同一时间点发生异常。
> 2. 再看 **tokens/s / MFU / checkpoint age** 是否同步异常，从而区分“优化问题”还是“系统问题”。
> 3. 然后看 **slice eval**，判断退化是全局的，还是只发生在代码、数学、长文等特定桶。
> 4. 最后才决定是 继续训练、回滚、降学习率，还是冻结数据与恢复路径做复盘。

训练配方是一条稳定性边界，而监控则是告诉你是否还在边界之内的仪表盘。即便训练过程看起来稳定，也不代表模型正朝着正确的产品方向演化。下一节的问题因此变成：这些监控数字究竟能说明什么，又不能说明什么。

6 评估与下游影响

> **本节先想这些问题**
>
> 1. 困惑度（perplexity）衡量什么？它有什么局限性？
> 2. 更低的困惑度并不总意味着更好的指令跟随——为什么？
> 3. 预训练的选择如何约束中训练、SFT 和推理？

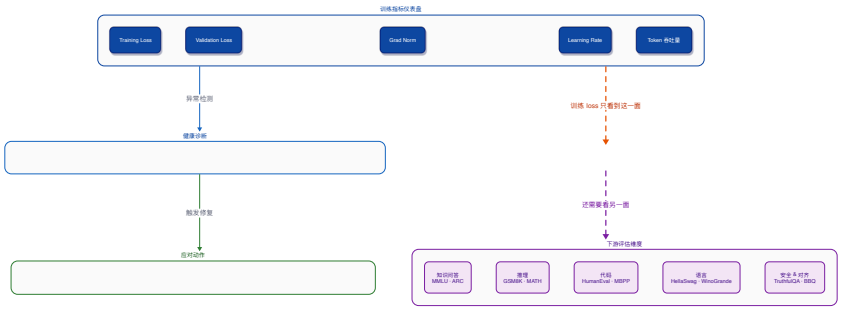

Figure 7: 训练监控与下游效果——loss ↓　能力 ↑

4. 你的基座模型在代码方面较弱——应该在预训练中修复还是之后？

5. 你预训练了两个架构相同但数据混合不同的模型。模型 A 困惑度更低但模型 B 在下游任务上得分更高。如何解释？

6.1 困惑度衡量的是“平均惊讶程度”，不是“产品完成度”

设验证集一共有 N 个待预测 token，真实序列为 $x_1, \dots, x_N$。模型在第 i 个位置给真实 token 的条件概率是

$$p_i = p_\theta(x_i \mid x_{<i})$$

那么平均 token loss，也就是平均负对数似然，可写成：

$$\ell = -\frac{1}{N}\sum_{i=1}^{N} \log p_i$$

困惑度（perplexity, PPL）就是把这个平均 loss 再做一次指数变换：

$$\mathrm{PPL} = \exp(\ell)$$

把两式合起来看，会更直观：

$$\mathrm{PPL} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N} \log p_i\right) = \left(\prod_{i=1}^{N} \frac{1}{p_i}\right)^{1/N}$$

这说明 PPL 本质上是真实 **token** 概率倒数的几何平均。
如果模型总能给真实下一个 token 很高的概率，那么 ℓ 会低，PPL 也会低；如果模型经常“猜偏”，给真实 token 的概率很小，那么 ℓ 会高，PPL 也会高。

一个生动的理解方式是：
PPL 像是在问——模型每走一步，平均相当于还在几个候选项里犹豫？

例如，如果模型在很多位置上给真实 token 的概率大约都是 0.2，那么

$$\ell \approx -\log 0.2, \qquad \mathrm{PPL} \approx 5$$

可以粗略理解为：模型平均像是在 5 个差不多可能的选项里做选择。
如果真实 token 的平均概率只有 0.05，那么

$$\mathrm{PPL} \approx 20$$

这时模型就更“困惑”，因为它平均像是在 20 个候选里摸索。

可以把它想成填词。

看到“回首向来萧瑟处，归去，也无风雨也无 _____”时，一个像样的模型会把“晴”的概率打得很高；
而一个较差的模型可能在“晴、寒、声、情”等词之间犹豫，甚至给出不太合适的续写。
前者的惊讶程度低，PPL 低；后者的惊讶程度高，PPL 高。

所以，PPL 确实是预训练阶段非常重要的基础指标，因为它：

- 便宜；
- 稳定；
- 可以高频计算；
- 与 next-token prediction 的训练目标直接对应。

但它的局限性也必须讲清楚。

1. 困惑度依赖 **tokenizer**

不同 tokenizer 下，同一文本会被切成不同的 token 序列，所以 N 变了、每个 p_i 的定义也变了，PPL 就不再处在同一个比较基准上。
一个 tokenizer 若把术语切得更碎，模型也许只是多做了几次较容易的局部预测，PPL 变化未必代表语言建模能力真的更强。

2. 困惑度依赖评估分布

如果 PPL 是在网页验证集上算的，那么它衡量的是模型对“网页文本分布”的拟合程度，而不是对所有任务分布的统一能力。
网页上的 PPL 更低，不等于代码、数学、法律文书或真实产品请求上的效果也同步更好。
换句话说，PPL 测的是某个分布上的

$$\mathbb{E}_{x\sim D_{\mathrm{val}}}[-\log p_\theta(x)]$$

而不是“所有你在乎的能力”的总和。

3. 困惑度不直接测行为

它不直接测：

- 指令跟随；
- 工具调用；
- JSON 结构化输出；
- 安全边界；
- 冗长、礼貌、风格一致性；
- 多轮对话体验。

原因也很简单：PPL 只关心“真实 token 有没有被赋高概率”，并不关心“这个回答作为产品行为是不是好答案”。
一个模型完全可能在验证集上 PPL 更低，却在真实产品里更爱啰嗦、更不会调工具，或者更不符合你想要的回答风格。

如果把预训练评估写成更贴近产品的形式，可以把真实关心的风险记为：

$$R_{\text{prod}}(\theta) = \sum_{k=1}^{K} \alpha_k R_k(\theta), \qquad \sum_{k=1}^{K} \alpha_k = 1$$

其中 R_k 可以分别对应代码、长文、工具使用、事实性、安全性或特定行业任务。
validation PPL 只是在测某个开发分布上的负对数似然；它并不直接最小化 R_{prod}。

因此，更准确的说法不是“PPL 没用”，而是 **PPL** 是训练期最重要的体温计之一，但它不是整份体检报告。

6.2 为什么更低的困惑度不保证更好的指令跟随

这个问题的核心在于：
预训练优化的是**“如何续写真实文本”**，而不是**“如何服从用户意图”**。

一个 base model 即使拥有更低的 PPL，也可能：

- 不知道该在何时拒答；
- 不知道什么叫“按照 JSON schema 回答”；
- 不知道何时调用工具；
- 不知道什么样的答案对用户更有帮助；
- 更倾向于“像互联网上常见文本那样继续写”。

而指令跟随、偏好对齐、安全边界和工具行为，大多是在后训练阶段通过 SFT、偏好优化、RL 或约束解码进一步塑造的。

因此，更低 PPL 常常意味着更强的底层语言建模能力，但不自动意味着更好的行为接口。这个区分非常重要，因为它决定了问题要去哪里修：

- 能力缺口，通常更早修；
- 行为缺口，通常更后修。

6.3 一个“loss 更低、产品更差”的具体案例

下面给一个合成但工程上很常见的例子。两次训练使用相同的 7B 架构、相同的总 token 预算，但 mixture 不同：

模型	开发集 PPL ↓	代码 pass@1 ↑	工具使用 success ↑	指令跟随 win-rate ↑	主要 mixture 变化
Model A	7.12	39.4	61.8	67.1	代码 + 正式技术文档权重上调
Model B	7.34	36.8	69.5	74.2	mixture 更均衡，保留更多对话与通用文档

为什么会出现这种“PPL 更低，产品更差”的反直觉现象？因为 A 在开发分布上学到了更容易预测的形式结构，却把对话行为、工具触发边界和自然指令风格推向了更窄的区域。换句话说，A 更会拟合那套验证文本，但 B 更接近真实产品风险 R_{prod}。

这也是为什么预训练评估里必须同时保留：1. 基础分布指标：loss / PPL；2. 切片指标：代码、数学、长文、多语言；3. 产品 proxy：工具使用、结构化输出、instruction following、安全边界。

6.4 预训练如何约束中训练、SFT 与推理系统

预训练不是后续阶段的“前置工序”而已，它会持续约束后面每一层。

tokenizer 约束

如果预训练 tokenizer 让代码、医学术语、表格符号或多语言字符切得过碎，那么后续 CPT、SFT、RAG 乃至推理成本都会受到影响。
这不是后训练能轻松补救的。

上下文长度与位置方案约束

如果预训练只在 8K 窗口上训过，后面再想稳定支持 128K，不仅是推理系统问题，也会触及位置编码、长度外推、KV 缓存和长上下文分布的问题。
后移可以做，但代价会变大。

数据混合约束

如果 base model 在代码、数学或长文一致性上很弱，后训练通常只能在已有底座上“拉齐表现”，很难凭空补出深层统计能力。

架构与系统约束

你选了什么架构、并行方式、精度和训练栈，都会继续影响：

- 中训练成本；
- 适配器训练方式；
- 推理系统的 KV 内存；
- 长上下文扩展的可行性；
- 工具调用时的结构化输出稳定性。

一句话概括就是：

> 预训练决定后面还能走多远、修多快、花多少钱。

6.5 基座模型代码弱，应该在预训练修，还是在后面修

这是一个非常典型、也非常实用的判断题。一个好用的经验规则是：

如果缺的是“统计能力”和“分布覆盖”，尽量前移修

例如：

- 模型不理解代码语法结构；
- 常见 API、库、语言习惯都很生；
- 代码上下文一长就崩；
- 看不懂自然语言与代码混排文本。

这类问题说明 base model 在代码分布上的覆盖不足，最好在预训练或至少在 CPT 阶段补，因为你缺的是底层语言建模能力。

如果缺的是“行为接口”，可以后移修

例如：

- 会写代码，但不会按照工具格式输出；
- 能完成函数体，但解释太差；
- 能写，但不遵守代码审查模板；
- 不会在不确定时先提问或先测试。

这类问题更接近指令跟随、交互风格和工具使用，通常可以在 SFT、偏好优化甚至推理时约束里修。

最糟糕的决策，是把底层能力缺口误判成后训练问题。
那样你会在一个不够强的底座上反复修接口，最后发现怎么调行为都不够自然。

6.6 为什么模型 A 困惑度更低，模型 B 下游任务更强

当两个架构相同、但数据混合不同的模型出现“A 的 PPL 更低，B 的下游任务更强”时，最常见的解释有四种：

1. 验证分布与任务分布不一致

A 更擅长你拿来算 PPL 的分布；B 更擅长你真正关心的任务分布。
这不是矛盾，而是评估维度不同。

2. B 的 mixture 更贴近目标能力

例如，B 可能在代码、数学、长文或专业文档上的占比更合理，因此即使总 PPL 不占优，真正的任务表现仍然更强。

3. tokenizer 或样本长度差异改变了学习速度

有些 mixture 在相同 token 预算下，实际提供给模型的“有效学习信号”更强；有些 mixture 则充满更容易预测却信息密度更低的 token。
PPL 下降速度并不总能代表“能力增长速度”。

4. A 的评估可能被污染或过拟合该验证分布

如果 A 的训练数据与验证集高度接近，它的 PPL 更好看并不奇怪；但这不等于它更有泛化性。

因此，工程上更成熟的说法不是“到底应该相信 PPL 还是 benchmark”，而是：

> 你必须确认你在比较的是不是同一个分布、同一个 **tokenizer**、同一个目标能力。

6.7 从预训练走向中训练与后训练

到这里，可以把预训练重新定义为一句非常精确的话：

预训练是在大规模 **token** 分布上，训练一个通用条件语言模型的过程。

它的产出主要有两层：

1. 能力底座：语言统计、知识压缩、模式迁移、长程依赖、结构理解；
2. 参数初始化：为中训练、SFT、偏好优化、工具使用训练和推理系统约束提供可塑的起点。

这也正是为什么后续章节必须分开讨论中训练与后训练：

- 中训练主要处理分布覆盖；
- 后训练主要处理行为接口。

如果把这两者混在一起，很多决策都会失焦：该前移修的能力被后移了，该后移修的行为却在底座阶段强行解决，最后既贵又不稳。

6.8 本节结论

1. 困惑度是预训练最重要的基础指标之一，但不是产品质量总分。
2. 更低 **PPL** 不保证更好的指令跟随，因为两者优化目标并不相同。
3. 预训练会持续约束 **tokenizer**、上下文、架构、底座能力和后续训练成本。
4. 底层能力缺口尽量前移修；行为与接口问题更适合后移修。
5. 同架构模型出现 **“PPL** 更低但下游更差**”**，往往是在提醒你：分布、目标和评估没有对齐。

预训练结束时，你拿到的不是“已经完成的助手”，而是一组被数据系统、预算约束、训练稳定性和评估纪律共同塑造出来的参数。

7 本章小结

预训练不是“把很多文本喂给模型”，而是在固定预算下，构建一套能持续把高质量 token 变成可泛化参数的系统。如果只记住本章的五个判断，可以记住下面这五句：

1. 数据系统的质量，决定模型的上限能否被真正接近。
2. 数据混合不是仓储问题，而是能力设计问题。
3. 预训练的预算规划，必须同时平衡参数、**token**、上下文和精度。
4. 大模型训练首先是分布式系统问题，其次才是“把优化器调漂亮”的问题。
5. **loss** 和困惑度只回答“模型有没有继续拟合分布”，不能单独回答“模型是否更适合产品”。

   LLM 的能力不是从一个 **loss** 函数里直接长出来的；它是被数据系统、算力预算、分布式基础设施、训练配方和评估纪律共同决定的。

预训练会不会被替代？在今天 AutoML 已经很强的情况下，AI 工程师还会存在吗？

笔者认为这个问题里，其实混着两件事。

第一件事是：谁来训练模型。
第二件事是：谁来为模型负责。

很多人把它们混为一谈，所以才会觉得，只要自动化越来越强，人就会退出。其实未必。自动化最擅长替代的，是那些目标已经被定义清楚、反馈已经被量化清楚、出了问题也不需要谁真正负责的工作。它会越来越擅长做实验、调参数、跑 A/B test、筛掉坏配置、保留好配置。只要一件事足够像搜索，agent 迟早会比人更适合做。

但预训练不是纯搜索。

预训练更像是在塑造一个脑子。不是因为它神秘，而是因为它昂贵、不可逆，而且后果会累积。你不是在调一个局部指标，你是在决定一个模型将如何表示世界。数据怎么进来，损失怎么定义，什么能力被鼓励，什么偏差被放大，哪些错误会在几万亿 token 之后固化下来——这些都不是“把实验跑完”就能回答的问题。这些问题最后都指向判断。而判断的核心，不是优化，而是责任。

所以我不觉得“训练脑子”这件事会消失。恰恰相反，它会一直存在，而且会一直是高门槛、高责任、也因此高回报的工作。因为真正稀缺的，从来不是会跑训练的人，而是知道什么时候该继续、什么时候该停，知道模型坏在哪儿，也愿意对结果负责的人。

真正会被替代的，是另一类 ML 工作：那些不需要理解 domain adaptation、不需要理解 failure mode、也不需要承担后果的训练流程。它们当然重要，但它们更像流水线。只要流程足够稳定，agent 接管只是时间问题。

这是技术进步常见的路径：机器先拿走那些不需要品味的工作，再逼着人去做那些真正需要判断的部分。最后留下来的，不是“操作机器的人”，而是“决定机器该成为什么的人”。

7.1 问题小结

1. 什么是预训练？
预训练是在海量未标注文本上，用自监督目标训练一个通用条件语言模型的过程。它让模型先学会语言和世界的统计结构，再为后续中训练与后训练提供底座。

2. 为什么 **LLM** 可以不用人工标注数据？
因为训练标签来自文本本身。给定前缀，真实的下一个 token 就是监督信号，所以 next-token prediction 天然属于自监督学习。

3. 什么是 **next-token prediction objective**？
就是让模型在每个位置根据历史 token 预测下一个 token，并最小化交叉熵损失。它的核心形式是 $P(x_t \mid x_{<t})$。

4. 为什么需要海量数据？
因为语言和知识分布非常复杂。模型如果要学到语法、文体、世界知识、代码结构和推理模式，就必须在足够多样的语料上反复看到这些现象。

5. 预训练教会了模型什么？
它教会模型语言流畅性、概念共现、文档结构、知识模式和一定程度的推理结构，但不会自动教会它像一个产品化助手那样行动。

6. 大规模训练数据如何构建？
通常要经过采集、解析、规范化、质量过滤、PII 与许可证处理、去重、分词、分片和污染审计，最后再进入 mixture 和训练系统。

7. 什么是 **data mixture**？
是不同数据源在训练中被采样的加权分布。它直接决定模型最终更像什么，因此本质上是能力设计，不是仓储管理。

8. 数据质量为什么比单纯数据量更重要？
因为参数容量和 compute 都很贵。高噪声、重复、模板化内容会浪费容量，也会提高记忆化与污染风险。

9. scaling laws 的工程意义是什么？
它告诉你扩大模型、数据和计算通常会带来可预测收益，但收益递减，因此必须在预算内找到更优配比，而不是只盯参数规模。

10. compute budget 为什么是硬约束？
因为训练成本大致同时受参数量、训练 token、上下文长度、精度和系统效率约束。预算不足时，模型可能不是“太小”，而是“没喂够、没跑够”。

11. 一个现代预训练管线长什么样？
从原始语料出发，经过解析、过滤、去重、分词、分片、数据混合，再进入分布式训练、checkpoint、验证和 contamination audit，最终产出 base model。

12. 数万亿 **token** 怎样被高效处理？
通常先预分词并写成高吞吐 shard，训练时按 rank 稳定读取，配合 packing、缓存和异步预取，避免在线 tokenization 和随机 IO 成为瓶颈。

13. 为什么需要大规模分布式基础设施？
因为单机通常放不下模型和训练状态，也无法在合理时间内完成训练。必须用 DP、TP、PP、ZeRO/FSDP 等并行策略把内存与计算拆开。

14. 如何监控训练稳定性？
至少看训练 loss、验证 PPL、grad norm、tokens/s、MFU、数据输入延迟和 checkpoint 健康度，并对异常 batch、恢复点和切片评估保持可回放能力。

15. 预训练如何影响下游 **alignment** 和 **inference**？
它会决定 tokenizer 成本、底座能力、上下文扩展难度、后续微调成本和推理系统效率。很多后续阶段的问题，其实源头都在预训练设计里。

下一章会继续沿着这个逻辑往下走：当通用预训练结束后，如何通过中训练 / 继续预训练（**CPT**），把模型进一步推向特定领域、特定任务分布和更具体的能力边界。

8 参考资料

1. Brown, T. B., et al. (2020). *Language Models are Few-Shot Learners.* NeurIPS. https://arxiv.org/abs/2005.14165

2. Kaplan, J., et al. (2020). *Scaling Laws for Neural Language Models.* https://arxiv.org/abs/2001.08361

3. Hoffmann, J., et al. (2022). *Training Compute-Optimal Large Language Models.* https://arxiv.org/abs/2203.15556

4. Raffel, C., et al. (2020). *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer.* JMLR. https://arxiv.org/abs/1910.10683

5. Wenzek, G., et al. (2019). *CCNet: Extracting High Quality Monolingual Datasets from Web Crawl Data.* https://arxiv.org/abs/1911.00359

6. Gao, L., et al. (2020). *The Pile: An 800GB Dataset of Diverse Text for Language Modeling.* https://arxiv.org/abs/2101.00027

7. Lee, K., et al. (2022). *Deduplicating Training Data Makes Language Models Better.* ACL. https://aclanthology.org/2022.acl-long.577/

8. Loshchilov, I., & Hutter, F. (2017). *Decoupled Weight Decay Regularization.* https://arxiv.org/abs/1711.05101

9. Shoeybi, M., et al. (2019). *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism.* https://arxiv.org/abs/1909.08053

10. Rajbhandari, S., et al. (2019). *ZeRO: Memory Optimizations Toward Training Trillion Parameter Models.* https://arxiv.org/abs/1910.02054

11. Grattafiori, A., et al. (2024). *The Llama 3 Herd of Models.* https://arxiv.org/abs/2407.21783