post_training_notes

LLM Post-Training 面试笔记

> 面试级别 · 结构化排版 · 保留原始知识点并补入 2025-2026 官方实践锚点

目录

- 1. 为什么需要 Post-Training
- 2. SFT 监督微调
- 3. Reward Model 奖励模型
- 4. RLHF 强化学习对齐
- 5. PPO 算法详解
- 6. DPO 及变体家族
- 7. GRPO 与后继算法
- 8. 推理模型训练范式
- 9. SFT 数据工程
- 10. LoRA 与参数高效微调
- 11. Agent 训练全链路
- 12. Test-Time Compute 与搜索
- 13. 分布式训练与推理优化
- 14. MoE / 长上下文 / Model Merging
- 15. 多模态 Post-Training
- 16. 评测治理
- 17. 面试题库

1. 为什么需要 Post-Training

1.1 Pre-Training 的本质与局限

Pre-trained model 在做且**只在做**一件事：

$$\max_{\theta} \sum_{t} \log P_\theta(x_t \mid x_{<t})$$

直觉解释：

- 给定前文 `context`，让模型把下一个 token 的条件概率尽可能压到真实答案上；
- 训练目标本质上是**语言建模**，而不是"做一个助手""遵守安全边界"或"理解用户真实意图"。

它的训练目标是**续写**，不是**对话**，不是**拒绝**，不是**帮助**。

- 输入"如何制作炸弹"→ 它会认真续写，因为互联网语料里存在相关文本
- 它不是"坏"，它只是**不懂拒绝这个概念**
- 它没有"助手"身份，只有"语言续写机器"身份

正是由于训练目标的本质差异，Pre-trained model 在以下四个核心维度上存在系统性缺失：

缺失能力	具体表现
对话格式	不知道什么时候该停，什么时候该回答
伦理判断	不拒绝有害请求
指令遵循	把"写封邮件"理解成续写邮件相关文章
人类偏好	不知道哪种回答人类更喜欢

1.2 Post-Training 完整链路与工业演进

Post-Training 的完整地图

可以把完整链路理解成一条非常清晰的能力升级路径：

阶段	做什么	解决什么问题
Pre-trained Model	只学习 `next token prediction`	会续写，但不会做助手
SFT	学习高质量问答范式	学会"怎么回答"
RM	学习人类偏好或质量排序	学会"什么是好答案"
RLHF / DPO	在对齐约束下优化输出分布	在不明显跑偏的前提下，把"好答案"概率抬高
Aligned Model	形成最终助手形态	具备 **Helpful / Harmless / Honest** 三类核心属性

一句话记忆：预训练负责"学语言"；Post-training 负责"学角色、学偏好、学行为边界"。

三个时代的工业演进

Post-Training 的工业路线经历了三个明显的演进阶段：

阶段 1（2022–2023）：Base → Instruct

- 核心问题：模型不会"干活"，只会续写

- 解法：SFT + RLHF/DPO
- 代表：InstructGPT / ChatGPT / Claude 2
- 评测重心：MT-Bench、AlpacaEval

阶段 2（2024）：Instruct → Thinking

- 核心问题：模型会干活但不会"深度推理"
- 解法：GRPO + 可验证奖励 + CoT 训练
- 代表：DeepSeek-R1 / OpenAI o1 / Qwen3
- 评测重心：MATH、AIME、LiveCodeBench

阶段 3（2025–2026）：Thinking → Agent

- 核心问题：模型会推理但不会"执行多步任务"
- 解法：Tool-use RL + Planning RL + Memory RL + MCP
- 代表：豆包 Agent / 扣子 / Claude Computer Use
- 评测重心：τ^2-Bench、SWE-Bench、AgentBench

四个核心张力（面试展示深度用）

每个演进阶段背后，贯穿始终的是以下四对核心张力：

张力 1：能力 ↔ 对齐（Alignment Tax）

- 对齐训练让模型更安全，但削弱推理能力
- 解法：并行训练而非串行

张力 2：效率 ↔ 质量

- 推理时多想 → 更好的答案，但 token 成本更高
- 解法：Adaptive Compute，按难度动态分配

张力 3：通用 ↔ 专精

- 全能模型 vs 专家模型
- 解法：Model Merging / MoE / Modular RL

张力 4：在线 ↔ 离线

- Online RL 数据新鲜但贵，Offline DPO 便宜但分布偏移
- 解法：Online Iterative DPO，定期刷新数据

> **面试话术：**
> "Post-training 经历了三个时代：从教模型'怎么说话'（SFT+RLHF），到教模型'怎么深度推理'（GRPO+可验证奖励），到现在教模型'怎么执行多步任务'（Tool-use RL+Agent RL）。

> 每个阶段的核心张力都是能力和对齐之间的平衡——对齐会带来 Alignment Tax，而可验证奖励是目前绕开这个矛盾的最优解。"

官方实践锚点（2025–2026，截至 2026-04 核实）

> 🏭 **工业补充**
>
> - **Qwen3（2025-04-29）**：官方发布 **Hybrid Thinking Modes**，把 **Thinking Mode** 与 **Non-Thinking Mode** 合到同一套模型里；公开模型包含 2 个 MoE 和 6 个 dense 版本，其中 `Qwen3-235B-A22B` 为 **235B 总参数 / 22B 激活参数**，`Qwen3-30B-A3B` 为 **30B 总参数 / 3B 激活参数**。
> - **Qwen3 Post-training**：官方明确写出 **四阶段训练流程：长 CoT cold start reasoning RL thinking mode fusion general RL**。这说明 2025 年后的 reasoning model 不再只是"更长思维链"，而是进入 **预算可控 + 模式可切换 + 更强 agent 能力** 的阶段。
> - **DeepSeek-R1（官方仓库）**：`DeepSeek-R1-Zero` 直接从 base model 做 RL，展示了 **self-verification、reflection、long CoT** 等能力；正式版 `DeepSeek-R1` 则采用 **两阶段 RL + 两阶段 SFT**，并基于 **DeepSeek-V3-Base（671B total / 37B activated / 128K context）**。
> - **面试表达升级**：讲工业演进时，最好把关键词从 **Base Instruct Thinking Agent** 再补成 **Thinking budget control / hybrid reasoning / tool-augmented execution / protocol standardization**。

1.3 Alignment Tax 与 Goodhart's Law

Alignment Tax：对齐是有代价的

现象： DPO/Safety RL 训练后，模型在推理 benchmark 上下降 5~15%。

根本原因：

- 人类标注员偏好"简洁、安全、礼貌"的回答
- DPO 把这个偏好编码进模型
- 推理任务需要"长链思考"
- → DPO 的优化方向和推理能力冲突

具体机制：

- DPO 训练时：
  - 简短答案 → 标注员认为简洁，给高分
  - 长链推理 → 标注员认为啰嗦，给低分
  - 模型学到：长 = 坏，短 = 好
  - → 思维链被主动缩短

- → 数学推理能力下降

表现形式（面试举例）：

- RLHF 之前：能解出复杂数学题（用 2000 token 推理）
- RLHF 之后：强行用 200 token 给出简短答案（可能错误）

工业界的缓解方案：

方案 1：并行而非串行（DeepSeek-R1 的选择）

- 错误做法：GRPO 完成 → 做 DPO → DPO 抹掉推理能力
- 正确做法：GRPO 和偏好信号同时训练，加权融合

$$\text{Loss} = \alpha \cdot \text{Loss}_{\text{GRPO}} + (1-\alpha) \cdot \text{Loss}_{\text{DPO}}$$

α 在推理任务上设为 0.8，让推理信号主导。

方案 2：专门的推理保护机制

- 在偏好数据里加入"推理质量"维度
- 好的推理链（即使很长）→ 标注为好回答
- 改变标注员的评判标准

方案 3：监控推理 benchmark

- DPO 训练中同步看 MATH/AIME 分数
- 一旦下降超过 2% → 立即停止 DPO

Goodhart's Law：指标成为目标时就失效了

原始表述：

> "当一个指标成为目标时，它就不再是好指标。"
> ——英国经济学家 Charles Goodhart

在 Post-training 里的具体体现：

- RM 设计初衷：衡量回答质量
- RM 成为优化目标：模型找到 RM 的盲区走捷径

走捷径的具体方式：

- RM 看到引用就打高分 → 模型塞满"据研究显示..."
- RM 看到长度就打高分 → 模型废话连篇
- RM 看到礼貌词就打高分 → 模型全是"当然！非常感谢您的提问！"

结果：RM 得分飞涨，用户体验骤降。这就是 Reward Hacking 的理论根源。

两者的关系

- **Goodhart's Law** 是病因（哲学层面）
- **Alignment Tax** 是具体症状（工程层面）

Alignment Tax = Goodhart's Law 在"推理能力"上的发病：

- DPO 的指标 = 人类偏好分数
- 人类偏好被"简洁"刷高
- → 模型走捷径：缩短思维链
- → 副作用：推理能力下降

> **面试连接话术：**
> "Goodhart's Law 是 Reward Hacking 的理论根源；Alignment Tax 是它在推理能力上的具体工程表现。解决思路本质上只有两类：要么让奖励变成多维、难以被单点攻击；要么让奖励尽可能可验证，比如数学和代码任务里的客观验证器。"

1.4 工业前沿视角与高频面试追问

Post-Training 的目标正在从"对齐回答"扩展到"对齐执行"

2025–2026 的明显变化：

- 顶级模型已经不再满足于"回答得像助手"；
- 更现实的目标是：**在有限推理预算下，稳定完成推理、工具调用、协议交互和执行闭环**。

一个很重要的工业信号：

- **Qwen3** 官方博客已经把后训练目标明确扩展到 **thinking mode、general RL、agent capabilities、MCP support**；
- **DeepSeek-V3** 官方 README 则说明后训练不只是做偏好对齐，还在把 **R1 的 verification / reflection 模式蒸馏**回标准 LLM。

这意味着什么：

- 早期 Post-Training 更像"把模型训成会说话的助手"；
- 现在的 Post-Training 更像"把模型训成**会分配算力、会守协议、会执行任务**的系统组件"。

资深工程师视角： 所以今天再讲 Post-Training，如果只停留在 `Helpful / Harmless / Honest`，深度已经不够；更完整的表述应该再补上：

- **compute-aware**：按难度分配思考预算；
- **tool-aware**：能稳定调用外部系统；
- **protocol-aware**：能遵循 MCP / function calling / structured output 等接口契约。

高频追问标准答法

Q：为什么 Pre-training 这么强了，还一定要做 Post-Training?

> 因为 Pre-training 学到的是"语料分布下的续写能力"，不是"助手身份下的行为边界"。它可能会说得很流畅，但并不知道什么时候该拒绝、什么时候该澄清、什么叫人类偏好、什么叫工具调用协议。所以 Post-Training 本质上是在把"语言模型"变成"可部署的助手策略"。

Q：Alignment Tax 和 Reward Hacking 有什么关系?

> Reward Hacking 是更一般的现象，意思是模型学会了刷指标而不是完成真实目标。Alignment Tax 可以理解成这个问题在推理任务上的一个具体症状：模型为了迎合"简洁、安全、礼貌"的偏好，主动缩短思维链，导致 reasoning 掉分。所以两者不是并列概念，而是"总问题"和"具体表现"的关系。

---

2. SFT 监督微调

2.1 核心思想与 Labels 掩码设计

用 `<指令，高质量回答>` 配对数据，教模型从"续写机器"变成"对话助手"。

一个最典型的 SFT 样本：

- **System**：你是一个有帮助的 AI 助手
- **User**：帮我写封道歉邮件
- **Assistant**：亲爱的 XX，对于……我深感抱歉……

本质上，SFT 在教模型两件事：

- 什么场景下应该以"助手"的身份回答；
- 在这个身份下，回答应该呈现什么格式、语气和完成度。

Labels 掩码设计（核心考点）

不是所有 token 都参与 Loss 计算！

```
input_ids: [SYSTEM tokens, USER tokens, ASSISTANT tokens]
labels:    [-100,          -100,         ASSISTANT tokens]

# -100 是 PyTorch 的忽略标记
# 只有 ASSISTANT 的回答部分计算 Loss
```

为什么这样设计?

- 目标是让模型学会"怎么回答"，不是学"怎么提问"
- 如果 USER 部分也算 Loss，模型会努力让问题更像训练集，目标混乱

多轮对话的 Labels 设计:

```
# 多轮对话：所有 ASSISTANT 回合都开放，所有 USER/SYSTEM 都掩盖
input_ids: [USER1, ASST1, USER2, ASST2]
labels:    [-100,  ASST1, -100,  ASST2]
```

为什么两个 ASST 都要开放?

- ASST1 教模型：给定 [USER1]，应该怎么回答（早期对话能力）
- ASST2 教模型：给定完整上下文，应该怎么回答（多轮理解能力）
- 只开放 ASST2 = 标注数据用了一半，浪费训练信号

代码实现:

```python
def build_multiturn_labels(messages, tokenizer):
    labels = []
    for message in messages:
        tokens = tokenizer.encode(message["content"])
        if message["role"] == "assistant":
            labels += tokens                    # ✅ 开放，参与 Loss
        else:
            labels += [-100] * len(tokens)      # ❌ 忽略（user/system）
    return labels
```

2.2 SFT 数据质量与 Sequence Packing 工程链

SFT 数据质量问题

用 GPT-4 生成 SFT 数据的问题:

- 存在"模型天花板效应"：学生永远无法超越老师
- 每次蒸馏都是有损压缩：真实人类偏好 → GPT-4 输出 → 你的模型
- 会导致 Model Collapse（模型坍塌）前兆：回答越来越平庸、安全但无聊
- 失去真实人类回答中的多样性和创造力

正确做法: 优先使用真人标注数据，GPT-4 生成数据仅作补充。

Sequence Packing 工程链

Sequence Packing 不是一个单点技巧，而是一条**问题驱动的工程链**。

问题背景：Padding 浪费

传统 Padding 方案：batch 里按最长样本补齐。

```
样本A: [50 tokens ][PAD×3950]  ← 98.7% 是废料
样本B: [2000 tokens][PAD×2000] ← 50% 是废料
样本C: [4000 tokens]            ← 满的
```

GPU 在 PAD token 上做 Attention 计算 = 纯粹的算力浪费，实际 GPU 利用率：30~50%。

核心思想：拼接变长序列

不补 PAD，把多个样本拼接成一条长序列：

```
max_length = 4096

传统：[样本A 50tokens][PAD×4046]  ← 浪费 98%

Packing：
  [样本A][样本B][样本C][样本D][样本E]...
  直到填满 4096 个 token

cu_seqlens（累积序列长度）记录边界：
  [0, 50, 250, 800, 1200, 2400, 4096]
```

吞吐量提升：30%~2x；数据长度越参差不齐，提升越明显。

关键工程问题一：Block Diagonal Mask（样本间污染）

拼接后的序列，Attention 默认跨样本 attend：

```
样本B 的 token 会 attend 到样本A 的 token
→ 不同样本互相"污染"
→ 训练信号被污染
```

解法：Block Diagonal Mask（块对角注意力掩码）

```
可视化（A=样本A内部，B=样本B内部，.=隔断）：
  A A . . . .
  A A . . . .
```

```
. . B B . .
. . B B . .
. . . . C C
. . . . C C
```

每个样本只 attend 自己内部的 token，样本之间完全隔离。

关键工程问题二：Flash Attention 的 cu_seqlens 解法

标准 Flash Attention 不支持 Block Diagonal Mask。Unsloth/Flash Attention v2 的 varlen 版本解法：

```
flash_attn_varlen_func(
    q, k, v,
    cu_seqlens_q = [0, 50, 250, 801, ...],  # 边界信息
    cu_seqlens_k = [0, 50, 250, 801, ...],
    max_seqlen_q = 4096,
    max_seqlen_k = 4096,
)
```

核心洞察：

- 不修改 Attention 矩阵结构（太贵）
- 而是告诉 Flash Attention "在哪里切断"
- Flash Attention 在 SRAM 分块计算时，自然不会跨越 `cu_seqlens` 边界
- → 零额外显存开销，速度和普通 Flash Attention 一样快

关键工程问题三：Sample-level Loss Normalization

每个 packed batch 里的样本数不固定：

```
packed_batch_1: 3 个长样本
packed_batch_2: 20 个短样本
```

默认 token 级平均：short sample 在大 batch 里梯度被稀释 → 训练不稳定。

解法：Sample-level Loss Normalization

```
def packing_loss(logits, labels, cu_seqlens):
    sample_losses = []
    for i in range(len(cu_seqlens) - 1):
        start, end = cu_seqlens[i], cu_seqlens[i+1]
        sample_logits = logits[start:end]
        sample_labels = labels[start:end]
```

```
        valid = sample_labels != -100
        if valid.sum() == 0:
            continue
        loss = cross_entropy(sample_logits[valid], sample_labels[valid])
        sample_losses.append(loss)
    return torch.stack(sample_losses).mean()  # 样本级平均
```

效果： 无论每个 packed batch 里有多少样本，每个样本的期望梯度贡献相等。

Sequence Packing 完整工程链汇总

暴露出来的问题	为什么会出现	对应解法
PAD 浪费算力	不同样本长度差异大，短样本后面全是 padding	把多个样本拼成变长长序列，用 `cu_seqlens` 记录边界
样本间 Attention 污染	拼接后如果不加约束，前一个样本会看到后一个样本的 token	使用 **Block Diagonal Mask** 或 `flash_attn_varlen_func` 做分块注意力
batch 内样本权重波动	一个 packed batch 里到底塞了多少个样本并不固定	使用 **Sample-level Loss Normalization** 保证样本级贡献稳定

> **面试亮点**：真正懂 Packing 的人，不会只说"它能提速"；更完整的回答是："Packing 先解决 padding 浪费，但立刻引出 attention 隔离和 loss 归一化问题，所以它本质上是一条成体系的工程链。"

2.3 PSFT：近端约束与 Entropy Collapse 防护

Entropy Collapse：SFT 的隐患

两个来源：

来源 A（数据侧）：

- SFT 数据格式趋同 → 模型反复看到相同句式 → 输出越来越模板化
- 解法：数据多样性 + 语义去重

来源 B（优化侧，更底层）：

- SFT 的 MLE 目标：最大化 $P(y|x)$
- 极端情况：把所有概率压在训练集见过的 token 上
- → 输出分布的熵（entropy）骤降 → 模型变成"确定性机器"，失去采样多样性
- 这是 MLE 目标本身的副作用，和数据多样性无关

- 数据量越大、epoch 越多，越严重

表现：

- 训练前："这道题答案是 42，因为..."（多样）
- Collapse 后："当然！以下是答案：\n1. ..."（模板化）

PSFT 的解法

在 SFT loss 上加 KL 约束：

$$\mathcal{L}_{\text{SFT}} = -\log P_\theta(y \mid x) \quad \text{（标准 SFT）}$$

$$\mathcal{L}_{\text{PSFT}} = \mathcal{L}_{\text{SFT}} + \alpha \cdot \text{KL}[\pi_\theta | \pi_{\text{ref}}]$$

- π_{ref} = SFT 开始前的预训练模型（冻结不动）
- α = 约束强度，通常 0.01~0.1

为什么有效：

- 预训练模型 π_{ref} 的输出分布比较"宽"（高熵）
- SFT 把分布压窄（低熵）→ KL 散度增大 → $\mathcal{L}_{\text{PSFT}}$ 增大
- → 梯度阻止分布继续收窄
- 用预训练模型的高熵分布作为"锚点"

PSFT vs RLHF 的 KL 约束对比

	PSFT	RLHF
_ref 是谁	预训练模型（冻结）	SFT 后的模型（冻结）
作用阶段	SFT 训练阶段	RL 训练阶段
防止什么	Entropy Collapse	Policy Drift
锚点	预训练分布（高熵）	SFT 分布

> **核心洞察（面试加分）**：后训练不是从 RL 阶段才开始管 policy drift，从 SFT 阶段就应该控制。每个阶段都需要自己的"锚点"。

2.4 诊断监控、训练超参与稳定性边界

SFT 诊断指标（五个）

```
sft_diagnostics = {
    # 指标1：format_diversity（最早预警）
```

```
    "format_diversity": {
        "含义": "输出格式的多样性",
        "量化": """
            outputs = [model.generate(q) for q in eval_questions]
            prefixes = [tokenizer.decode(o[:10]) for o in outputs]
            diversity = len(set(prefixes)) / len(prefixes)
            # diversity < 0.3 → 报警
        """,
        "预警时机": "训练中实时计算，几百步一次",
        "collapse 表现": "80% 的输出以'当然！以下是'开头",
    },

    # 指标2：entropy（最灵敏）
    "entropy": {
        "含义": "输出 token 分布的信息熵",
        "量化": """
            probs = F.softmax(logits, dim=-1)
            entropy = -(probs * probs.log()).sum(dim=-1).mean()
            # entropy 从 6.0 骤降到 2.0 → 报警
        """,
        "预警时机": "每个 step 都能算，最灵敏",
        "collapse 表现": "entropy 持续下降，不再反弹",
    },

    # 指标3：template_leakage（明确的过拟合信号）
    "template_leakage": {
        "含义": "模型是否把 chat template 的 special token 输出",
        "量化": """
            # 检测输出里是否含有 <|assistant|>、<|end|> 等
            leakage_rate = count(special_tokens_in_output) / total
            # leakage_rate > 0.01 → 报警
        """,
        "预警时机": "需要 generate，比 entropy 慢一点",
        "collapse 表现": "输出里出现 <|assistant|> 这样的标记",
    },

    # 指标4：MMLU_score（滞后指标）
    "MMLU_score": {
        "含义": "通用能力是否退化",
        "量化": "跑完整 MMLU benchmark，对比 SFT 前后",
        "预警时机": "需要几小时，collapse 严重后才能看到",
        "用途": "事后确认，不能用来预警",
    },
```

```
    # 指标5：benchmark_scores（问题定位用）
    "benchmark_scores": {
        "含义": "各领域分项表现",
        "量化": "HumanEval（代码）/ GSM8K（数学）/ MT-Bench（对话）",
        "用途": "某领域骤降 → 数据配比问题，不是 collapse",
    },
}
```

报警时序（从快到慢）：

- **最先（训练中实时）：** `entropy` （每 step 可算，collapse 开始就能看到）→ `format_diversity` （每几百 step 采样，灵敏）
- **其次（需要 generate）：** `template_leakage` （发现时 collapse 还未太严重，仍然及时）
- **最后（需要跑完整 benchmark）：** `MMLU / benchmark` （几小时出结果，属于"事后验尸"）

结论： 监控顺序：`entropy → format_diversity → template_leakage → MMLU/benchmark`；早停策略：entropy 骤降 或 diversity < 0.3 → 立即停止。

训练超参与稳定性边界

工业经验的默认起点：

- `epoch`：通常 `1~2`
- `lr`：全量微调多在 `1e-5 ~ 3e-5`，LoRA 常在 `1e-4 ~ 3e-4`
- `scheduler`：cosine decay 是最常见默认值
- `warmup ratio`：通常 `0.03 ~ 0.1`

为什么 epoch 往往不大：

- 大模型不是小模型时代的"多训几轮更稳"；
- 过多 epoch 很容易先出现 format overfitting，而不是 val loss 明显恶化。

更可靠的 early stop 规则：

- 不是只看 `val_loss`；
- 要联动看：benchmark、format diversity、template leakage、slice eval、安全/拒答行为是否漂移。

> **一句话：**SFT 最难调的不是"能不能收敛"，而是"怎么在还没开始模板化之前停下来"。

2.5 Chat Template、掩码策略与工业前沿

Chat Template 与 Tokenizer 边界

为什么这件事在 2025-2026 变得更重要：

- 模型输出越来越依赖统一 chat template、tool schema、structured answer；
- 一旦模板和 tokenizer 的 special token 约定不一致，训练和推理会在最底层发生"看不见的偏移"。

最常见的四类坑：

- **special token 冲突**：模板里写了 `<assistant>`、`<tool_call>`、`<think>`，但 tokenizer 没有把它们注册成真正 special token；结果是模型把这些标记当普通文本学，后续极容易出现 template leakage。
- **BOS / EOS 处理不一致**：训练时自动加 BOS，推理时又手动拼了一次；或训练时 assistant 段尾没有 EOS，推理时却期待 EOS 终止。
- **HF 与 vLLM 模板差异**：HF 侧模板可能由 tokenizer 内部 `chat_template` 渲染；vLLM/serving 侧又拼了另一份模板；最终导致 train/infer 分布不一致。
- **assistant prefix 差异**：训练样本是 `Assistant:`，线上推理却是 `<|assistant|>`；模型会把风格、格式、停止行为学歪。

工程判断标准： 任一训练样本在进入 model 前，必须满足：渲染后的 prompt 唯一确定；特殊 token 序列唯一确定；loss mask 与角色边界完全对齐。

> **面试一句话**：chat template 不是前端展示层问题，而是训练分布定义问题。

Tool Call / Tool Result / Think / Answer 的掩码策略

这类样本最容易犯的错误，就是把所有 token 一股脑都开 loss。更合理的做法是按行为目标来决定：

段落	常见策略	什么时候开 loss	什么时候 mask
`think`	视任务而定	需要显式教会思维链格式时	不希望模型过拟合 verbose CoT 时
`tool_call`	常常开	需要模型学会结构化调用、参数填充	工具调用完全外部生成时
`tool_result`	通常 mask	只有在想让模型学会"读取并压缩工具返回值"时少量开	大多数情况下应 mask，避免模型学"伪造环境结果"
`answer`	必开	最终回复始终是主要监督目标	基本不应 mask

为什么 `tool_result` 往往要 mask：

- 它是环境返回值，不是模型应当生成的内容；

- 如果对 `tool_result` 开 loss，模型容易学会"复述或幻想环境结果"，而不是在真实环境里等待结果。

三种典型训练口径：

- **Assistant-only**：只学最终 answer，最稳，但 tool use 学得慢。
- **Tool-supervised**：`tool_call + answer` 开 loss，适合 agent 初训。
- **Reasoning + Tool + Answer**：`think + tool_call + answer` 都开 loss，能力最强，但也最容易模板化和泄漏。

工业前沿视角（2025–2026）

> 🏭 **工业补充**
>
> **工业界现在对 SFT 的理解更像三层脚手架：**
>
> - **身份脚手架**：用 chat template、system prompt 风格、拒答范式把模型先拉到"助手分布"上；
> - **格式脚手架**：让模型先学会 `tool_call / JSON / <think>...</think> / <answer>...</answer>` 这些结构；
> - **搜索脚手架**：给后续 RL 一个可优化的起点，避免一上来采样全是垃圾轨迹。
>
> **为什么 Qwen3 / DeepSeek 这类 reasoning pipeline 仍然要保留 cold-start SFT：**
>
> - RL 不擅长从"完全不会输出某种格式"开始学；
> - 如果模型压根不会生成 `<think>`、工具 schema 或结构化 answer，RL 的奖励再清楚也很难稳定传回去。
>
> **工程上最容易被忽略的两个点：**
>
> - **chat template 与 tokenizer 边界**：模板符号、special token、assistant 起始标记如果和 tokenizer 不一致，会直接污染 loss mask。
> - **tool call / tool result 的掩码策略**：是否让模型学习生成 `tool_call`；是否要对 `tool_result` 开放 loss；这其实决定了模型学的是"会不会调用工具"，还是"会不会复述工具返回值"。
>
> **一句话总结**：SFT 在 2025-2026 的位置，更像是**把策略分布先拉进可训练区域**；没有这层边界条件，后面的 DPO、GRPO、Trajectory RL 往往只是高成本地放大噪声。

高频追问标准答法

Q：为什么 SFT 时通常只对 Assistant 部分计算 Loss？

> 因为训练目标是让模型学"如何回答"，不是学"如何提问"或"如何重建整段对话"。如果把 User / System 也纳入 loss，模型会把优化压力浪费在复现输入分布上，目标就偏了。这也是为什么 labels 里通常把非 assistant token 置成 `-100`。

Q：Sequence Packing 的真实难点是什么？

> 不是简单拼起来就结束了。它至少有三层问题：padding 浪费、样本间 attention 污染、以及 packed batch 下的 loss 权重波动。能把这三层链条讲清楚，才算真正理解 packing 的工程含义。

3. Reward Model 奖励模型

3.1 核心思想：训练数据、目标函数与 Margin 扩展

为什么需要 RM?

SFT 只能学"示范答案"，无法比较多个答案的好坏。同一个问题可以有多个"都还不错"的回答，SFT 没有比较和判断能力。RM 的职责正是成为一个**自动评委**：输入任意问答对，输出一个代表质量的标量分数。

```
[Question + Answer] → RM → 标量分数（如 0.87）
```

训练数据：不是打分，是排名！

同一个问题，人类标注员给出排名：

```
Question: "解释一下黑洞"
回答A > 回答C > 回答B
```

让人类说"A 比 B 好"远比说"A 得 87 分"更容易、更一致。

为什么用排名不用打分：

- 人与人之间的绝对分数难以统一（我的 8 分可能是你的 6 分）
- 相对排名更稳定、标注一致性更高

训练目标：Bradley-Terry 模型

对于每一对（好回答 y_w，坏回答 y_l）：

$$\mathcal{L} = -\log \sigma\big(r(y_w) - r(y_l)\big)$$

目标：让好回答的得分始终高于坏回答的得分。

基于 **Bradley-Terry 模型**，假设人类选择回答 A 优于 B 的概率为：

$$P(A > B) = \sigma\big(r(A) - r(B)\big)$$

Margin Loss 扩展：Reward Calibration

标准 Bradley-Terry 的局限： 所有偏好对被同等对待，低信心的噪声标注会污染训练。

```
案例A：标注员投票 100:0（绝对一致）→ 高信心
案例B：标注员投票 51:49（几乎相同）→ 低信心，噪声
两个案例施加相同训练信号 → RM 对边界情况判断不准
```

Margin Loss 设计：

$$\mathcal{L} = -\log \sigma\big(r(y_w) - r(y_l) - \text{margin}\big)$$

margin 从标注员信心度估算：

投票比例	margin	含义
100:0	1.0	要求分数差距大
75:25	0.5	中等约束
51:49	0.1	几乎不施加约束

效果： 高信心偏好对 → 强迫 RM 拉开差距；低信心偏好对 → 放松要求，不让噪声主导。这叫 **Reward Calibration（奖励校准）** —— RM 分数不只反映排名，还反映"差距有多大"。

> **面试话术：**
> "标准 RM 把所有偏好对一视同仁，但标注员 51:49 投票的数据是噪声，不应该和 100:0 的数据一样强。Margin Loss 用标注员投票比例构造动态 margin，让 RM 专注于学习高信心的偏好，这是 Reward Calibration 的核心思路。"

3.2 RM 全景图：六种奖励信号与多目标融合

六种奖励信号的统一图谱

在构建奖励模型时，任务的性质直接决定了适用的奖励信号类型：**任务越可验证，越应优先使用客观验证器；任务越开放，则越依赖主观评判。**

可验证性光谱（从低到高）：

```
低（完全开放的创意/对话）←————————————————→ 高（有唯一标准答案的数学/代码）

Judge / Rubric / Pairwise RM —— ORM —— PRM —— Verifier / Executor
```

1. Verifier / Executor（规则验证器 / 代码执行器）

- **机制**：完全确定性的程序，不是神经网络（如数学符号验证器、代码单元测试、规则匹配）。
- **特点**：最客观，直接评判通过/不通过（对/错）。零成本，零偏见，零幻觉，零被 Hack 风险，具备极强的可扩展性。
- **适用**：拥有绝对唯一标准答案的可验证任务。

2. ORM（Outcome Reward Model，结果奖励模型）

- **机制**：训练一个专门对任务最终输出结果进行打分的模型。
- **特点**：比 Verifier 适用范围更广，客观且难以被 Hack。但对于长推理链任务，存在奖励信号稀疏、梯度弱的问题，且包含一定的 RM 固有偏见。
- **适用**：无法用硬规则验证，但最终结果可以明确判断优劣的场景。

3. PRM（Process Reward Model，过程奖励模型）

- **机制**：对推理链的每一步（Step-by-step）进行细粒度打分（例：step_1 正确 +0.3，step_3 错误 -0.5）。
- **特点**：提供极其密集的反馈信号，能够有效指导中间搜索过程，非常适合复杂推理。
- **代价**：步骤级别的训练数据构造与标注困难（通常需要蒙特卡洛 Rollout），标注成本极高且估计存在偏差。
- **适用**：需要复杂推理和中间步骤验证的数学、代码或逻辑任务。

4. Pairwise RM（传统偏好模型）

- **机制**：输入 `(问题, 回答)`，输出一个标量分数，通过 Bradley-Terry pairwise loss 训练。
- **适用**：通用对话质量的综合主观评判。

5. Rubric-based Reward（基于量规的结构化奖励）

- **机制**：预先定义明确、多维度的评分标准（Rubric），将其写进 Judge 的 Prompt，让强模型按条目独立打分，最后加权合并。
  - 示例 Prompt："按以下标准评分（1-10）：准确性（30%）、完整性（30%）、流畅性（20%）、格式（20%）"。
- **特点**：比自由放养的 Judge 更加结构化、稳定、标准一致、可重复且高度可解释（知道具体哪个维度失分）。缺点是 Rubric 的设计需要极强的专业知识。

【面试加分用例】：代码生成任务的 Rubric 设计

```
| 维度 | 满分 | 评分标准 |
|---|---|---|
| 功能正确性 | 0-3 分 | 3=全通过；2=大部分通过但边界失败；1=基本逻辑对但实现错；0=功能错误 |
| 代码质量 | 0-2 分 | 2=清晰规范带注释；1=可读但不清晰；0=难以理解 |
| 执行效率 | 0-2 分 | 2=最优或接近最优时间复杂度；1=可接受；0=效率极差 |
```

总分：加权合并后输入到 RL 作为最终 Reward。

6. Judge（LLM-as-Judge，强模型评判）

- **机制**：利用 GPT-4 等强模型直接作为裁判对回答质量进行综合评估。
- **特点**：最为灵活，覆盖范围最广，适用于完全开放的任务。
- **缺点**：推理成本高，不可重复，且存在各种固有偏见（如位置偏见、偏好冗长回答、自我增强偏见/Sycophancy）。

多目标 Reward 融合与冲突解决

工业场景下的 RLHF 通常无法只追求单一指标，往往需要同时兼顾多个互相冲突的目标。

常见的优化目标： 帮助性/有用性（Helpfulness）、安全性（Safety）、诚实性/准确性（Honesty/Accuracy）、格式合规（Format）、工具使用正确性（Tool-correctness）、简洁流畅性（Efficiency/Fluency）。

典型冲突模式：

- 有用性 ↔ 安全性：用户询问制药或黑客技术，详尽的回答有用，但属于不安全内容。
- 有用性 ↔ 诚实性：模型为了显得"有用"，可能会编造听起来合理但实际上错误的答案（幻觉）。
- 有用性 ↔ 简洁性：模型倾向于输出长篇大论来提升表面有用度（冗长偏见）。
- 准确性 ↔ 流畅性：技术上绝对准确的回答可能生涩难懂，不够流畅。
- 工具调用 ↔ 格式规范：模型专注于复杂的工具调用逻辑时，容易忽略系统要求的输出格式。

四种工业级多目标融合方案：

方案 A：加权线性融合（Weighted Sum，最基础）

$$R_{\text{total}} = w_1 \cdot R_{\text{helpful}} + w_2 \cdot R_{\text{safe}} + w_3 \cdot R_{\text{accurate}}$$

缺陷：权重极难平衡，各目标之间 Scale 不一致；更严重的是，一个维度的高分可能掩盖另一个维度的严重缺陷（例如，格式完美的危险发言依然能拿到高分）。

方案 B：分层约束与门控（Hierarchical Constraint / Safety First）

将安全性等核心底线设为必须满足的"硬约束"，其他目标设为"软目标"。在满足安全的前提下，最大化帮助性：

```
if safe(response) == False:
    reward = -10    # 安全红线直接扣除
else:
    reward = R_helpful + R_accurate
```

应用：优先保证安全，再优化其他指标（Qwen3Guard 设计思路）。

方案 C：课程式/分阶段训练（Curriculum / Phased Training）

解耦冲突目标，按阶段各个击破，避免多目标同时优化导致的训练不稳定：

- Phase 1（早期）：只优化主要目标（如 Helpfulness RL 提升帮助性）
- Phase 2（中期）：引入 Safety RL 确保安全底线
- Phase 3（后期）：加入工具使用效率、格式规范等辅助目标

应用：Qwen3 的 Safety RL 采用此演进思路。

方案 D：Pareto 前沿优化与选择（Pareto Frontier）

放弃粗暴加权，训练多个不同权重配置的模型。在所有目标维度上同时进行多目标优化，要求在不降低任何单一目标性能的前提下提升整体表现。实践中通常结合 Model Merging（模型融合）技术来融合具备不同偏好的模型，或使用 Multi-objective DPO 近似实现。

3.3 RM 防御机制与工程细节

RM Ensemble、Audit 与在线更新

单一奖励模型在 RLHF 过程中极易触发 **Goodhart's Law**：因为单一 RM 的训练数据有限，必然存在盲区，Actor 会迅速找到漏洞进行 Reward Hacking，导致分数虚高但实际质量崩塌。

1. RM Ensemble（多 RM 集成）

同时训练并运行 3~5 个不同的 RM（不同架构、不同训练数据，或专职 RM 如安全 RM、质量 RM、格式 RM）。

融合计算：

- 平均值：$\text{reward} = (RM_1 + RM_2 + RM_3)/3$
- 加权平均：根据各 RM 在验证集上的准确率表现赋予权重
- 保守估计（最小值）：取所有 RM 给出的最低分或中位数

优势：不同 RM 的盲区和偏见往往不重叠且相互抵消。模型极难同时 Hack 所有"评委"，显著降低了漏洞风险，提供更稳健的奖励信号。

工业实践：Anthropic 大量使用多个 RM 并行评估，且倾向于采用**保守估计（取最低分或中位数）而非平均值**，以此强势阻断模型利用单一 RM 漏洞得分的可能。代价是成倍增加了训练时的前向推理开销。

2. RM Audit（定期人工审计）

每隔一段时间，系统自动随机抽取被 RM 打出"极高分"的样本，由人工介入复核——"RM 认为完美的回答，真的完美吗？"类似财务内部审计，专门用于捕获 Actor 最新发明的 Hack 模式

（例如发现了特定的废话触发词能骗取高分），从而针对性修复 RM 漏洞。

3. RM Online Update（奖励模型在线更新）

随着 Actor 模型能力飞速进化，旧版 RM 的判别能力会逐渐跟不上甚至失效。必须定期收集最新 Actor 生成的困难样本（Hard Examples），交由人工重新标注，或利用更强大的闭源模型（如最新版 Claude/GPT-4）重新蒸馏出新的 RM，以确保"裁判"的水平始终高于"运动员"。

RM 训练工程细节

初始化： 工业里常见做法不是从头训练 RM，而是从同族 SFT checkpoint 初始化；这样 reward model 一开始就具备较强的语言理解和助手分布。

偏好对规模： 开放任务通常要足够多的 pair 才能稳住边界；但比"数量更多"更重要的是：难样本比例、边界样本质量、高分样本审计质量。

最常见的离线指标： held-out ranking accuracy、AUC、margin calibration、不同 slice 上的一致性。

Early stopping： 不是看 train loss 一直降就继续；真正需要盯的是 held-out ranking accuracy 和高分样本审计是否开始漂。

Reward Calibration 与量纲统一

多 reward 组合时最容易出的问题：不同 reward 量纲不同；一个 reward 数值大，不代表它更重要，只可能代表它没校准。

常见校准方法： normalize、clipping、quantile clipping、temperature scaling、per-task weights。

一个实际原则： 先把 reward 校到"量纲相近"，再谈业务权重；不要一开始就直接线性相加。

对应英文检索词： `reward calibration`、`reward normalization`、`reward scaling`。

Scalable Verifier Signal

为什么 2025-2026 大家都在强调 scalable verifier signal： 因为单一人类偏好太贵、太慢、太主观；可扩展的 verifier 信号才是 reasoning、code、agent 场景真正能放大的监督来源。

一套更完整的 verifier 分层：

- **规则型**：schema、regex、格式、JSON、XML
- **执行型**：单测、编译、SQL 执行、API 返回
- **环境型**：tool success、页面状态变化、任务完成率
- **代理指标型**：计划完整度、步骤覆盖率、子任务成功率

Judge Bias 与 Debias

最常见的 judge bias： position bias、length bias、self-preference、style bias。

常见去偏手段： swap position、tie 规则、multi-judge ensemble、RBD / randomized bidirectional comparison、对抗提示词与长度控制。

PRM 数据构造成本

PRM 的最大问题不是效果，而是贵： 需要过程级标注；需要 rollout 多路径采样；还要面对偏差-方差权衡。

工程直觉： 路径采样越多，估计越稳，但成本越高；路径采样越少，成本越低，但 step-level score 方差更大。

RM 尺寸选择与 Reward Drift 检测

RM 多大更合适：

- 同规模：表达能力足，但贵
- 略小：最常见，性价比高
- 略大：在 ORM / PRM / Judge 上更稳，但系统成本高

reward drift 的常见信号： 高分样本人工抽查变差、judge disagreement 上升、reward holdout 与线上真实满意度背离。

> **一句话**：reward 漂了，不代表模型更强，通常代表奖励标准开始失真。

Reward Service 化

工业里的 reward 不再只是训练脚本里的一段 forward；更常见的形态是 **reward service**：异步打分、结果缓存、版本化、latency budget、fallback judge / fallback verifier。

为什么服务化重要： rollout 规模上来后，reward 往往会成为吞吐瓶颈；不服务化，就很难把训练平台做大。

3.4 工业前沿视角与高频面试追问

Reward Stack 正在从"单一 RM"升级成"组合栈"（2025–2026）

> 🏭 **工业补充**
>
> **单个 RM 已经越来越不够用：**
>
> - 开放式任务需要 **pairwise RM / judge / rubric**；
> - 数学与代码任务更依赖 **verifier / executor**；
> - 长链推理和 agent 任务又需要 **ORM / PRM** 去处理结果与过程的不同粒度。

> **一个很有代表性的官方信号：**
>
> - Qwen 在 **Qwen2.5-Math-PRM** 官方博客里明确指出：`Qwen2.5-Math-PRM-7B` 在 ProcessBench 上优于不少现有 PRM；更有意思的是，`Qwen2.5-Math-RM-72B` 这个 **ORM** 对识别步骤错误也有相当强的能力，甚至超过了一些开源 PRM。
>
> **这说明的底层逻辑是：** ORM 并不只会"看最终答案对不对"；当 base model 足够强、reward 模型规模足够大时，结果级监督本身也能部分恢复过程判别能力。
>
> **资深工程师视角：** 工业里更好的做法不是执着于"到底 ORM 还是 PRM 更先进"；而是把它们看成不同精度、不同成本、不同延迟的评估器：
>
> - **Verifier**：高精度、低歧义
> - **Judge / Rubric**：覆盖开放任务
> - **ORM**：结果锚点稳定
> - **PRM**：过程信用分配更细
>
> **面试一句话**：2025-2026 的 Reward Model 已经不是一张网络，而是一套**可组合的 reward service**。

高频追问标准答法

Q：为什么 Reward Model 常用 Bradley-Terry，而不是直接做绝对分数回归?

> 因为人类更擅长判断"两个回答谁更好"，不擅长给一个回答打稳定的一致性绝对分数。Pairwise preference 的噪声通常比 pointwise score 更低。Bradley-Terry 把这个偏好关系转成概率建模，更适合真实标注流程。

Q：ORM 和 PRM 到底怎么选?

> 如果任务有清晰最终结果，ORM 更稳、更便宜，也更容易服务化。如果 credit assignment 很难、步骤质量本身就很关键，PRM 更有价值。2025-2026 的工程趋势不是二选一，而是把 ORM / PRM / verifier 做成组合栈。

---

4. RLHF 强化学习对齐

4.1 核心目标函数：Reward Hacking 与 KL 约束

为什么不直接"刷 RM 分数"?

Reward Hacking（奖励黑客）问题：

RM 训练数据里，"带引用的回答"分数普遍较高，模型因此学到了捷径：

- 不管问什么，在回答里塞满"据研究显示……""数据表明……"
- → RM 给高分 ✓，但内容可能是一本正经地胡说八道 ✗

模型不是在变得更好，**它是在学习欺骗裁判**。

本质原因：

```
RM 得分  ≠  真实人类满意度

两者在训练初期高度相关，
但随着模型疯狂优化 RM 分数，
两者会逐渐撕裂。
```

KL 散度：给模型拴一根绳子

解法： 在优化 RM 分数的同时，用 KL 散度约束模型不能跑太远。

KL 散度衡量两个概率分布之间的距离：

$$\mathrm{KL}[\pi_\theta \| \pi_{\mathrm{ref}}] = \sum_y \pi_\theta(y) \cdot \log \frac{\pi_\theta(y)}{\pi_{\mathrm{ref}}(y)}$$

值越大 = 当前模型离原始模型越远；值越小 = 离原始模型越近。

RLHF 完整目标函数

$$\max \; r(x,y) - \beta \cdot \mathrm{KL}\big[\pi_\theta(y \mid x) \| \pi_{\mathrm{ref}}(y \mid x)\big]$$

其中：

符号	含义	优化方向
$r(x,y)$	RM 打分	最大化
π_θ	当前训练的模型	可变
π_{ref}	原始 SFT 模型（冻结）	固定锚点
$\mathrm{KL}(\cdots)$	两个模型输出分布的距离	最小化
β	控制绳子松紧的超参数	β 越大=越保守；β 越小=越激进（风险更高）

4.2 工业级 RLHF 系统架构

四个模型体系（工程噩梦）

模型	角色	是否训练	类比
Actor (π_θ)	生成回答	✅ 训练中	棋手（落子）
Reference (π_{ref})	KL 约束基准	❌ 冻结	原始自己
Reward Model	给回答打分	❌ 冻结	裁判
Critic	估算长期价值	✅ 训练中	棋手脑中的形势判断

4 个模型 × 70B 参数 = 工业界的显存噩梦。

完整平台链路：持续迭代的数据飞轮

完整的工业级 RLHF 训练系统是一个持续迭代的闭环数据飞轮，而不是单次训练过程。核心流水线由七个关键组件构成：

1. Prompt Queue（提示词队列）

- 功能：存储待训练的 prompt 池，并向系统分发。
- 特性：支持按难度或类型分层采样，支持优先级调度（难题优先），支持动态补充新 prompt（Online RLHF）。

2. Rollout Engine（推理/采样引擎）

- 功能：使用当前 Actor 模型生成回答（支持批量并行推理，如每个 prompt 生成 G 个回答）。
- 特性：通常使用 vLLM 或 SGLang 进行加速。输出结果为 $(prompt, response, log_probs)$。

3. Reward / Verifier Stack（奖励计算栈）

- 功能：对生成的 response 进行打分计算。
- 特性：支持异步计算以避免阻塞采样。包含多个组件如：ORM（对最终答案打分）、PRM（对每步推理打分）、Rule Checker（格式与安全检查），以降低 Reward Hacking 风险。输出结果为 $(response, reward_score)$。

4. Advantage Builder（优势计算器）

- 功能：计算训练所需的优势值。
- 特性：PPO 中使用 Critic 估计 Value 并计算 GAE advantage；GRPO/DAPO 中使用组内归一化。支持 Token-level 或 Sequence-level 计算。输出结果为 $(response, advantages, returns)$。

5. Trainer（参数更新训练器）

- 功能：执行模型权重更新。
- 特性：计算 Actor 更新（Policy Loss）和 Critic 更新（Value Loss，仅 PPO）及 KL 惩罚。支持 ZeRO/FSDP/Megatron 分布式训练（3D 并行），配合梯度裁剪与混合精度。

6. Eval Gate（自动评测门控）

- 功能：每隔 N 步自动运行私有评测，监控 reward、KL、entropy 和 length。
- 特性：如果模型性能退化或未通过质量门控，则触发 early stop 或回滚操作。

7. Model Registry（模型注册中心）

- 功能：存储所有版本的模型权重与 checkpoint。
- 特性：支持快速回滚到任意历史版本，并作为 A/B 测试的基础设施。新模型通过 Eval Gate 后方可注册进入下一轮飞轮。

完整平台链路顺序：`Prompt Queue → Rollout Engine → Reward / Verifier Stack → Advantage Builder → Trainer → Eval Gate → Model Registry → Canary / A-B / Rollback`

KL 散度计算与自适应控制

Sampled KL vs Exact KL：

- **Exact KL（精确计算，不可行）：**

$$\mathrm{KL}(\pi_\theta \| \pi_{\mathrm{ref}}) = \sum_y \pi_\theta(y) \cdot \log \frac{\pi_\theta(y)}{\pi_{\mathrm{ref}}(y)}$$

需要对词表大小和序列长度内的所有可能输出求和，计算开销巨大。

- **Sampled KL（实际应用，无偏估计）：**

$$\mathrm{KL} \approx \log \frac{\pi_\theta(a_t)}{\pi_{\mathrm{ref}}(a_t)}$$

只对实际采样到的 token 计算 KL，计算开销仅需 Reference Model 进行一次前向传播。**这也是 RLHF 训练中 Reference Model 必须全程保持在线的原因。**

Adaptive KL Controller（自适应 KL 控制）：

固定的 KL 惩罚系数 β 极难调节：β 太大限制了学习速度（模型几乎不变）；β 太小则约束太弱（导致 KL 爆炸和 Reward Hacking）。训练初期模型变化快需要大 β，后期稳定后小 β 即可。

机制：设定一个目标 KL 值（`target_kl`，通常在 0.01~0.1 之间），根据当前实际 KL 动态调整 β。

算法逻辑（OpenAI InstructGPT 方案）：

- 如果 `current_kl > 1.5 × target_kl` → 说明偏离过大，加强惩罚：$\beta = \beta \times 1.5$
- 如果 `current_kl < target_kl / 1.5` → 说明步子太小，放松惩罚：$\beta = \beta / 1.5$

优势：免去手动调参，自动适应不同训练阶段，保证训练稳定。

常见失败诊断与监控指标

关键监控指标：

- `clipfrac`： PPO 中 ratio 被截断的比例（正常范围在 0.1~0.3 之间）
- `entropy`： 输出分布的策略熵（持续下降预示着模式坍塌）
- `KL`： 与参考模型的距离（持续快速增大预示着模型崩溃）
- `reward`： 平均奖励（理想状态为稳步上升）
- `response_length`： 回答长度（应保持相对稳定，不应单调增减）

六种常见失败模式与诊断方案：

失败模式 1：KL 爆炸（KL Explosion）

- 症状：KL 曲线持续快速增大，突破阈值，模型输出行为异常，与原始模型差异极大。
- 原因：β 太小（惩罚过弱），学习率过大，或 PPO 的 Clip ϵ 过大，奖励信号异常。
- 解法：降低学习率，减小 ϵ，增大 β 或引入 Adaptive KL Controller，检查 advantage 的归一化过程。

失败模式 2：奖励坍塌（Reward Collapse）

- 症状：RM 评分短暂上升或飞涨后骤降，或高分回答在人工抽查中质量极差。
- 原因：Reward Hacking（模型找到了 RM 的漏洞或盲区）。
- 解法：人工抽查高分样本，引入多源奖励（RM Ensemble），加入明确的规则约束（Rule Checker），定期用新数据更新 RM。

失败模式 3：长度漂移（Length Drift）

- 症状： `response_length` 持续单调增加（或减少），模型学会通过"刷长度"来骗取高分。
- 原因：RM 存在隐含的 verbosity bias（认为长回答天然更好）。
- 解法：在奖励中显式加入长度惩罚项（Length Penalty），或参考 SimPO/Dr.GRPO 采用长度归一化机制。

失败模式 4：模式/熵坍塌（Mode/Entropy Collapse）

- 症状：策略熵（Entropy）骤降，所有回答失去多样性，变成相似的死板模板。
- 原因：过度优化 reward，导致模型丧失了探索性。
- 解法：在损失函数中增加 Entropy Bonus（如 PPO 目标里的 $c_e \cdot H$ 项），或在采样时稍微增大温度系数。

失败模式 5：格式崩溃（Format Collapse）

- 症状：模型输出的内容不再遵循 prompt 指定的格式（如 JSON、特定的思维链结构）。
- 原因：RL 过程为追求极致的分数，牺牲了指令中的格式要求。
- 解法：在 Reward 栈中强制加入格式合规奖励，如果不符合格式直接给予大幅度的负分。

失败模式 6：Value 过拟合（Value Overfit，PPO 特有）

- 症状：Critic 网络的 Value Loss 降得极低，但 Actor 的 Policy 却不收敛。
- 原因：Critic 对历史数据产生了过拟合，导致其对新采样的轨迹估值极不准确。
- 解法：比较 Critic 预测值与实际 reward 的相关性，限制 Critic 的更新步数，或加入 Value Clipping。

4.3 RLAIF 与安全对齐

RLAIF 与 Constitutional AI

核心思想：

- 传统 RLHF：人类写偏好标注 → 贵、慢、规模受限
- RLAIF：给模型一部"宪法"（一组原则），让模型自己批判自己的输出

宪法原则示例：

- "回答应该是无害的"
- "不应该协助非法活动"
- "应该诚实，不应该欺骗用户"

成本对比：

- 人类标注 1 条偏好对：约 $1~$5（含质检）
- RLAIF 生成 1 条偏好对：约 $0.001
- 成本差距：1000~5000 倍

核心风险：

风险 A：Sycophancy（谄媚性）

- 模型批判自己时倾向于认为"听起来自信的回答"更好、"更长、更详细的回答"更好；
- → 这些偏好固化进 RM → 螺旋式放大，模型越来越会迎合而不是说真话。

风险 B：Value Lock-in（价值锁定）

- 宪法写于某个时间点，那个时间点的认知偏差 → 永久编码进模型；
- 等发现时，数百万条训练数据已经生产完毕 → 必须从头重来，成本极高。

工业界缓解方案：

- 针对 Sycophancy：宪法里显式加入反谄媚原则，"不要因为回答更长就认为它更好"。
- 针对 Value Lock-in：宪法版本控制 + 定期人工审查；多套宪法并行训练，交叉验证。

- 混合策略（最优实践）：人类标注核心安全数据、高风险场景（贵但精准）；RLAIF 负责通用能力数据、大规模覆盖（便宜但有偏）；人类复核抽样验证 RLAIF 数据质量（兜底）。

安全对齐：Red Teaming 与闭环流水线

安全对齐不是单个 loss，而是一条闭环流水线：

一个完整的工业流程通常是：安全策略定义（policy / constitution / risk taxonomy）→ 种子红队样本库 → 自动攻击扩写（prompt mutation / jailbreak template / 多语言变体）→ 模型对抗生成 → 自动 judge / rule checker 初筛 → 人工审查高风险样本 → 回流到 SFT / DPO / RLAIF / Safety RL → Canary eval → 线上监控与回滚。

关键点： 安全不是"训练前标一遍数据就结束"，它是和评测、上线、回滚绑定在一起的持续过程。

Jailbreak 攻击谱系（面试里按类别讲）：

- **直接型**：明确要求有害内容，如武器、欺诈、违法绕过。
- **角色扮演型**：通过"你现在不是助手，你是小说角色/越狱代理"来绕策略。
- **提示注入型（Prompt Injection）**：在网页、文档、工具返回值里夹带"忽略之前指令"。
- **编码/混淆型**：base64、谐音、拼音、缩写、代码片段、符号分隔等。
- **多语言 / code-switch 型**：英文守住了，换成中英夹杂、小语种、方言后漏掉。
- **工具滥用型**：不靠文本回答有害内容，而是诱导模型去调用高权限工具。

over-refusal 与 helpfulness：安全系统最容易翻车的平衡点

- **under-refusal**：该拒绝的不拒绝，安全事故最严重。
- **over-refusal**：本来是合法请求，但模型因为过度保守全部拒掉。

工业上不能只看 harmful rate，还要同时看：benign prompt 的通过率、边界样本的拒绝精度、多轮澄清后能否恢复正常帮助。

实务结论：最稳妥的设计不是"所有可疑请求都一刀切拒绝"，而是"先分类风险，再决定澄清 / 降权回答 / 拒绝 / 触发安全专模"。

multilingual safety 与 tool-use safety（2025 之后的重点）：

多语言安全不能只在英文安全集上过关，应至少覆盖：中文、中英混输、小语种、谐音/缩写/绕写表达。

工具安全比"生成一段危险文本"更棘手，因为它会产生真实外部动作。常见防线包括：tool allowlist / denylist、参数 schema 校验、destructive action 二次确认、最小权限令牌、工具返回值再审查。

> **面试一句话**：文本安全主要防"说错话"，tool safety 还要防"做错事"。

线上安全评测与回滚（别只讲离线 benchmark）：

- 离线阶段：HarmBench / 私有红队集 / 多语言安全集 / tool-abuse 集
- 上线阶段：shadow traffic、canary release、实时风险告警
- 回滚触发条件：高风险违规率上升、over-refusal 激增、某类工具被异常高频调用、新增 jailbreak 模板穿透率显著升高

一个成熟团队的做法：把安全评测当成和 Win Rate、延迟、成本同级别的发布门槛，而不是事后补丁。

4.4 高级工程话题：异步 RL、资源编排与平台化

PPO mini-batch / multi-epoch 更新机制

PPO 在工程上不是"采样一批 → 只更新一次"，更典型的流程是：rollout 一大批轨迹 → 切成多个 mini-batch → 对同一批数据更新 `K` 个 epoch。

为什么这么做： rollout 极贵，尤其在大模型和 tool-use RL 下；所以希望尽量榨干每一批采样数据的价值。

代价： 同一批数据反复更新越久，和当前策略的偏差越大；stale ratio 会逐渐上升。

Rollout / Trainer 解耦

为什么真实平台一定会拆开：

- rollout 更像高吞吐推理服务；trainer 更像高带宽反向传播服务；两者的资源形态完全不同。

典型分工：

- rollout 绑 `vLLM / SGLang`
- trainer 绑 `FSDP / ZeRO / Megatron`

Async RL / Stale Policy / Freshness Filter

Async RL 的收益： 吞吐更高，GPU 空转更少。

Async RL 的代价： 一部分样本来自旧策略，这就是 stale policy。

常见缓解： freshness filter、importance correction、限制 rollout 允许落后的版本数。

Partial Rollout / Replay Buffer / Off-policy 边界

Partial rollout 为什么出现： 在长轨迹、长 CoT、Agent 场景里，完整 episode 太贵；所以工业里会保留部分轨迹、局部片段、或高价值 replay。

风险： 一旦 off-policy 成分太高，更新会变飘；reward 归因也更容易失真。

更细一点的稳定条件：

- 只保留最近若干版本的样本；
- 对旧样本做 importance correction；
- 高奖励但异常长、异常格式、异常 tool trace 的样本不直接 replay；
- 一旦 judge / verifier 与当前策略分布差太远，就宁可丢样本，不硬吃。

一个典型反例： 把一批旧策略下采出来的长轨迹直接塞回新策略训练；新策略其实已经不再处于那个分布附近；结果就是：ratio 失真 → KL 波动 → advantage 变噪 → 更新方向开始飘。

Actor / Ref / Reward / Critic 资源放置策略

常见放置方式：

- `Actor`：单独 rollout 资源池
- `Reference`：冻结服务化，通常可和 actor 解耦
- `Reward`：服务化或与 rollout 共置
- `Critic`：和 trainer 侧更紧密

一个实用原则： 谁最吃推理吞吐，谁优先和 rollout 引擎绑定；谁最吃反向带宽，谁优先和 trainer 侧绑定。

Wall-clock 经验判断： 当任务变成长 CoT、tool-use、multi-turn agent 时，rollout 往往会占掉总训练时间的大头；实际壁钟时间经常不是由反向传播决定，而是由采样吞吐决定。

训练时长的估算模板（面试用）：

如果面试官问"70B RLHF 大概要训多久"，更稳妥的回答不是给一个死数字，而是先拆：rollout engine 吞吐、平均生成长度、每轮要采多少 prompt、reward/verifier 延迟、trainer 的 global batch 和更新频率，然后再说 wall-clock 会强依赖 rollout，而不是只依赖 GPU 算力。

假设：每轮 rollout N 个 prompt，平均每个 prompt 生成 L 个 token，rollout 吞吐是 R token/s，每轮做 U 次参数更新，那么仅 rollout 时间大致就是：

$$T_{\text{rollout}} \approx \frac{N \cdot L}{R}$$

总 wall-clock 可以粗略看成：

$$T_{\text{total}} \approx T_{\text{rollout}} + T_{\text{reward}} + T_{\text{train}} + T_{\text{eval}}$$

为什么这个模板有价值： 它能把"训多久"从拍脑袋，变成一个依赖 rollout 吞吐、平均长度和 verifier 延迟的可解释估算。

> **面试口述版本：** "如果是 70B 级别的 RLHF / GRPO，我不会先报训练多少小时，而是先拆 rollout token 总量、rollout 引擎吞吐、reward/verifier 延迟、每轮更新频率。很多时候不是 trainer 算得慢，而是 rollout 和 verifier 把 wall-clock 吃掉了。"

OpenRLHF vs veRL 架构对比

OpenRLHF：

- 基于 `Ray + vLLM` 驱动的高性能 RLHF 框架；
- 强调 actor / reward / reference / critic 的资源编排与 async agentic RL。

veRL / verl：

- 支持 `FSDP / FSDP2 / Megatron-LM` 与 `vLLM / SGLang / HF Transformers` 的统一 RL 平台；
- 更强调 HybridFlow、placement、rollout/trainer 解耦和大模型级别训练。

RL 崩溃诊断 Playbook（速查）

症状	优先排查项
KL 爆炸	学习率、beta、clip、reward scale
reward 飙升但样本变差	reward hacking
长度单调漂移	verbosity bias、length penalty、stop token
熵塌缩	entropy bonus、温度、采样多样性
格式崩溃	格式 reward、schema gate、template 一致性
value overfit	critic 更新步数、value clipping、预测值与真实 reward 的相关性

4.5 工业前沿视角与高频面试追问

RLHF 正在平台化，算法只是其中一层（2025–2026）

> 🏭 **工业补充**
>
> **今天工业界说"做 RLHF"时，往往在说一整个平台，而不是一个 PPO 脚本。**
>
> **典型平台化特征：**
>
> - rollout 和 trainer 分离部署；
> - reward / reference / verifier 服务化；
> - checkpoint、eval、registry 纳入统一流水线；
> - 用 **vLLM / SGLang** 承担高吞吐采样；
> - 用 **veRL / OpenRLHF** 这类框架统一管理 actor、trainer、reward、async rollout。
>
> **这和早期 RLHF 的差别：**

> - 早期重点在"loss 对不对"；
> - 现在重点在"loss 对不对 + rollout 吞吐够不够 + 版本治理稳不稳 + 奖励服务能不能在线演进"。
>
> **一个工程判断标准**：如果一个 RLHF 系统还做不到采样/训练解耦、奖励栈可替换、checkpoint 可回滚、eval gate 自动拦截，那它更像实验代码，还不算真正的工业 RLHF 平台。

高频追问标准答法

Q：RLHF 为什么通常要保留 Reference Model?

> 因为 RL 只看 reward 时，策略很容易为了刷分迅速跑飞。Reference Model 提供的是一个"不要偏离太远"的锚点，KL 惩罚本质上就是把新策略拴在旧分布附近。没有这个锚点，Reward Hacking 和模式坍塌会更严重。

Q：RLAIF 能不能彻底替代人类标注?

> 不能完全替代，最多是大幅放大人类标注的覆盖范围。因为 RLAIF 本身会继承 judge model 的偏差，比如 sycophancy、value lock-in、多语言盲区。工业里更现实的做法是：人类定义高风险原则和抽样复核，RLAIF 负责规模化扩写。

---

5. PPO 算法详解

5.1 核心设计：Clip 截断、Advantage 与 GAE

PPO 要解决的问题

RL 训练中，如果每次参数更新"步子太大"：

- 模型会突然变成另一个人
- 之前采样的数据全部作废
- 训练直接崩溃

PPO（Proximal Policy Optimization）的核心目标，就是**在保证策略持续改进的同时，限制每次更新幅度**，避免训练不稳定。

Clip 截断：PPO 的核心机制

```
ratio = π_θ(a|s) / π_old(a|s)  # 新旧策略的概率比值

L_clip = min(
```

```
    ratio * advantage,                    # 原始目标
    clip(ratio, 1-ε, 1+ε) * advantage     # 截断后的目标
)
# ε 通常取 0.2
# 意思是：新旧策略概率比，最多只允许偏离 20%
```

直觉： 无论梯度有多大，每步最多只走这么远。好处是**训练更稳定**，代价是**收敛更慢**。

PPO 不再允许策略因为某一次高 advantage 样本而发生剧烈偏移，而是把优化限制在一个"可信更新区间"内。

Advantage（优势值）

$$\text{advantage} = \text{这个回答实际得分} - \text{Critic 预测的平均得分}$$

- advantage > 0：这个回答比预期好 → 增大这个回答的概率（强化）
- advantage < 0：这个回答比预期差 → 减小这个回答的概率（抑制）

Critic 的作用就是**估算"平均水平基线"**，把原始奖励信号变成相对值，从而降低方差，提高训练稳定性。

也就是说，PPO 更新的不是"奖励高不高"，而是"比当前策略平均水平更好就提高概率，更差就降低概率"。

GAE（Generalized Advantage Estimation）完整推导

为什么需要 GAE

优势值 A_t 的估计存在一个经典矛盾：**无偏但高方差**，或**低方差但有偏差**。

- **方式 A：Monte Carlo（$\lambda = 1$）**
  - $A_t = \sum \gamma^l r_{t+l} - V(s_t)$
  - 优点：无偏差；缺点：方差大（需要完整 episode）
- **方式 B：TD(0)（$\lambda = 0$）**
  - $A_t = r_t + \gamma V(s_{t+1}) - V(s_t)$
  - 优点：低方差，实时计算；缺点：有偏差（依赖 V 的估计质量）

GAE 的作用，就是在二者之间做折中：$\lambda \in [0, 1]$，平衡 bias 和 variance。

GAE 推导过程

先定义 TD 残差（Temporal Difference Error）：

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

其中：r_t 是第 t 步奖励，$V(s_{t+1})$ 是 Critic 对下一状态的估值，$V(s_t)$ 是对当前状态的估值。

GAE 定义为：

$$A_t^{\text{GAE}} = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \cdot \delta_{t+l}$$

展开后是：

$$A_t^{\text{GAE}} = \delta_t + \gamma\lambda \cdot \delta_{t+1} + (\gamma\lambda)^2 \cdot \delta_{t+2} + \cdots$$

于是可以得到递推关系：

$$A_t^{\text{GAE}} = \delta_t + \gamma\lambda \cdot A_{t+1}^{\text{GAE}}$$

这就是代码里要**从后往前递推**的原因。

参数含义

- γ：折扣因子（通常 0.99），控制长期奖励和短期奖励的权重平衡
- λ：GAE 参数（通常 0.95），控制 bias-variance 权衡
  - $\lambda = 0$：纯 TD，低方差高偏差
  - $\lambda = 1$：纯 MC，高方差低偏差
  - $\lambda = 0.95$：工业界默认值，通常是较好的平衡

Schulman 2015 之后，$\lambda = 0.95$ 基本成为 PPO/GAE 的默认工程配置。

代码实现

```python
def compute_gae(rewards, values, gamma=0.99, lam=0.95):
    """
    rewards: (T,)   每步奖励（LLM 通常只有最后一步非零）
    values:  (T+1,) Critic 估值（包括 V(s_{T+1}) = 0）
    """
    T = len(rewards)
    advantages = torch.zeros(T)
    gae = 0.0

    for t in reversed(range(T)):
        delta = rewards[t] + gamma * values[t + 1] - values[t]
        gae = delta + gamma * lam * gae
        advantages[t] = gae

    return advantages
```

LLM 场景下 GAE 的特殊性

LLM 上的 PPO 和传统控制任务不同，一个关键区别是：**奖励通常极度稀疏**。

```
传统 RL：每步都有 reward
LLM：通常只有最后一步有 reward

r_t = 0（t < T）
r_T = RM(response) 或 RM_score
```

因此在中间步骤：

$$\delta_t = 0 + \gamma V(s_{t+1}) - V(s_t)$$

最终步骤：

$$\delta_T = r_T + 0 - V(s_T)$$

这意味着：

- 梯度信号需要从最后一步向前传播
- 中间 token / 中间步骤的 advantage 几乎完全依赖 Critic 的估值
- **Critic 的质量决定了 GAE 信号的质量**
- 这也是 LLM PPO 训练不稳定的根本原因之一

→ **VAPO 的 Value-Pretraining 就是为了提高这一步的质量**，本质就是先把 Value 学稳，再做 PPO。

5.2 PPO 完整目标函数与典型超参数

PPO 的完整 Loss 并不只有一个 clipped policy objective，而是通常由四部分组成：

$$\mathcal{L} = \mathcal{L}_{\text{policy}} - c_v \cdot \mathcal{L}_{\text{value}} + c_e \cdot \mathcal{L}_{\text{entropy}} - \beta \cdot \mathcal{L}_{\text{KL}}$$

也常写成：

$$\mathcal{L} = \mathbb{E}\big[\min(\text{ratio} \cdot A,\ \text{clip}(\text{ratio}, 1-\varepsilon, 1+\varepsilon) \cdot A)\big] - c_v \cdot \mathcal{L}_{\text{value}} + c_e \cdot H(\pi) - \beta \cdot \text{KL}(\pi_\theta \| \pi_{\text{ref}})$$

两种写法本质一致，只是符号习惯略有差别。

Policy Loss（核心）

$$\text{ratio} = \frac{\pi_\theta(a \mid s)}{\pi_{\text{old}}(a \mid s)}$$

$$\mathcal{L}_{\text{policy}} = -\mathbb{E}\big[\min\big(\text{ratio} \cdot A,\ \text{clip}(\text{ratio}, 1-\varepsilon, 1+\varepsilon) \cdot A\big)\big]$$

作用：

- 当新策略比旧策略变化不大时，正常按 advantage 更新
- 当 ratio 偏离过大时，被 clip 截断

- 防止更新过猛，保证训练稳定

这是 PPO 最核心的设计。

Value Loss（训练 Critic）

$$\mathcal{L}_{\text{value}} = \mathbb{E}\big[(V_\theta(s) - V_{\text{target}})^2\big]$$

为了防止 Critic 更新过猛，工程上常加入 **Value Clipping**：

$$V_{\text{clipped}} = V_{\text{old}} + \text{clip}(V_\theta - V_{\text{old}}, -\varepsilon, \varepsilon)$$

$$\mathcal{L}_{\text{value}} = \mathbb{E}\big[\max\big((V_\theta - V_{\text{target}})^2,\ (V_{\text{clipped}} - V_{\text{target}})^2\big)\big]$$

这样可以防止 Value 网络突然震荡，否则会直接污染 advantage 估计。

Entropy Bonus（熵奖励）

$$\mathcal{L}_{\text{entropy}} = -\mathbb{E}\Big[\sum_a \pi(a \mid s) \cdot \log \pi(a \mid s)\Big]$$

加入熵奖励的目的，是防止策略过早坍塌（collapse），鼓励模型保持一定探索性和输出多样性。c_e 通常取 0.01。

- entropy 太低：模型已经过度确定化，容易训练僵死
- entropy 太高：策略过于发散，训练没有收敛

KL 惩罚

$$\mathcal{L}_{\text{KL}} = \text{KL}[\pi_\theta \| \pi_{\text{ref}}]$$

作用： 约束当前策略不要偏离参考模型太远；防止语言模型在 PPO 过程中"训歪"；保持语言质量、可读性和分布稳定性。在 RLHF / LLM PPO 中，参考模型往往就是 SFT 模型，这一项尤其重要。

β 通常 0.02~0.1，且常由 Adaptive KL Controller 动态调整。

典型超参数汇总

超参数	典型值	含义
ε	0.2	clip 范围
c_v	0.5	value loss 权重
c_e	0.01	entropy 权重
β	0.02（动态调节）	初始 KL 系数
γ	0.99	折扣因子
λ	0.95	GAE 参数

5.3 训练监控、工程细节与工业前沿

PPO 训练监控指标

训练 PPO 时，不能只盯 reward，还需要同时观察多个稳定性指标：

```
ppo_metrics = {
    "clipfrac":        "比例超过 clip 范围的 token（0.1~0.3 正常，过高说明更新太猛）",
    "entropy":         "策略熵（过低说明 collapse，过高说明训练无效）",
    "KL":              "当前策略和参考模型的 KL 散度（过大触发 early stop）",
    "reward":          "平均 RM 分数（持续上升说明训练有效）",
    "response_length":"平均回答长度（异常增长说明 length reward hacking）",
    "value_loss":      "Critic 损失（过大说明 Critic 拟合不好）",
    "pg_loss":         "Policy gradient loss（正常波动）",
}
```

这些指标可以理解为 PPO 的"体检面板"：

- `clipfrac` 过高：说明更新幅度过猛
- `entropy` 过低：说明策略塌缩
- `KL` 过大：说明偏离参考模型太远
- `reward` 上升：说明优化方向有效
- `response_length` 异常增长：可能出现 length reward hacking
- `value_loss` 过大：说明 Critic 没学稳
- `pg_loss`：用于观察 policy 优化是否正常波动

PPO 工程补充：Mini-batch、多 Epoch 与 Value 更新稳定性

PPO 的真实工程循环通常是：rollout 一批数据 → 切成多个 mini-batch → 对每个 mini-batch 做 `K` 个 epoch 更新。

为什么要强调这一点： 这直接决定了 stale ratio 风险；也决定了同一批数据到底被"压榨"到什么程度。

value update 最容易踩的坑：

- policy 更新得太快，critic 跟不上
- critic 过拟合旧样本，新的 advantage 开始失真

常见缓解：

- 限制 value update 次数

- value clipping
- 分开调 policy lr 与 value lr
- 定期刷新 rollout，避免同一批数据用太久

一句话总结

PPO 的本质是：用 Clip 限制策略更新幅度，用 Critic 估计基线，用 GAE 在偏差和方差之间折中，再结合 Value Loss、Entropy Bonus 和 KL 惩罚，让策略能够稳定地朝高奖励方向优化。

对于 LLM 来说，PPO 最大的难点不是公式本身，而是：

$$\text{奖励稀疏} + \text{Critic 难训} + \text{KL 约束} + \text{长序列信用分配}$$

这也是为什么 LLM 场景下的 PPO，工程复杂度远高于标准控制任务。

工业前沿视角与高频面试追问（2025–2026）

> 🏭 **工业补充**
>
> **PPO 仍然重要**：它是最经典、最系统的 policy gradient 工程化方案；很多后来算法的稳定性讨论，本质上都在和 PPO 做对照。
>
> **但在 2025-2026 的 reasoning / long CoT / large MoE 场景里，PPO 的代价越来越明显：**
>
> - 需要额外的 Critic；
> - Value 训练本身就容易滞后；
> - 长轨迹下，Critic 误差会层层传导到 advantage。
>
> **为什么长链推理下 Critic 更难：** 价值网络要压缩的是"当前前缀未来还能拿到多少 reward"；一旦任务很长、奖励很稀疏、路径分叉很多，这个函数本身就比 policy 更难拟合。
>
> **资深工程师视角：** 所以 2025-2026 的 reasoning RL 更偏向用 **GRPO / GSPO / REINFORCE++** 减少 Critic 维护成本；只在确实需要细粒度值函数控制时，再回到 PPO。
>
> **一句话**：PPO 不是过时了，而是它从"默认主角"变成了"高成本但解释力很强的基准线"。

高频追问标准答法

Q：为什么 PPO 里一定要有 Advantage，而不是直接拿 reward 更新？

> 因为原始 reward 方差太大，直接更新会非常不稳。Advantage 的作用就是把"绝对分数"变成"相对当前策略平均水平的超额收益"。有了这个基线，策略梯度的方差会小很多。

Q：GAE 的核心直觉是什么？

> GAE 不是简单地把所有未来 reward 相加，而是把一串 TD error 做指数加权累积。λ 越大越接近 Monte Carlo，偏差小但方差大；λ 越小越接近 TD，偏差大但方差小。所以 GAE 本质上是在偏差和方差之间做连续可调的折中。

6. DPO 及变体家族

6.1 核心洞察、损失函数与 RLHF 对比

DPO 的核心洞察与完整推导链

RLHF 的最优解在数学上等价于：

$$\pi^*(y \mid x) \propto \pi_{\text{ref}}(y \mid x) \cdot \exp\big(r(x,y)/\beta\big)$$

既然知道最优解的形式，能不能用偏好数据直接拟合这个最优解，**跳过 RM 训练和 RL 训练**？**答案：可以。**

面试里更完整的推导链应该是：

- 从带 KL 约束的 RLHF 最优策略出发；
- 得到最优策略满足 $\pi^*(y|x) \propto \pi_{\text{ref}}(y|x)\exp\big(\frac{1}{\beta}r(x,y)\big)$；
- 再把隐式 reward 写成策略与 reference 的对数比；
- 最后结合 Bradley-Terry 偏好建模，得到 DPO loss。

> **一个高质量表述：** DPO 不是"拍脑袋写出来的监督学习 loss"；它是带 KL 约束的偏好优化，在特定假设下的闭式替代。

DPO 损失函数

$$\mathcal{L}_{\text{DPO}} = -\log\sigma\Big(\beta \cdot \log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \cdot \log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\Big)$$

- y_w：人类偏好的回答（winner）
- y_l：人类不喜欢的回答（loser）

用人话说：

- 好回答：让当前模型比"原始的我"更愿意生成它 ✓
- 坏回答：让当前模型比"原始的我"更不愿意生成它 ✓

RLHF vs DPO 完整对比

维度	RLHF	DPO
模型数量	4 个	2 个（Actor + Reference）

维度	RLHF	DPO
是否需要 RM	✅ 需要单独训练	❌ 数学消掉了
训练稳定性	不稳定（RL 本身不稳定）	稳定（监督学习）
实现复杂度	高	低（约 20 行核心代码）
数据类型	在线采样（动态）	离线偏好对（静态）
工业界地位	逐渐被替代	目前主流

6.2 Distribution Shift 与 Online Iterative DPO

DPO 的核心缺陷：Distribution Shift

问题： DPO 使用静态的离线偏好数据，而 RLHF 可以实时采样新数据。

```
DPO 训练数据在时间点 T0 收集：
  此时模型能力 = Level 1
  偏好对 (yw, yl) 是针对 Level 1 的输出标注的

训练到时间点 T1：
  模型能力 = Level 3
  但还在用 Level 1 的偏好数据

Level 1 时代的"坏回答 yl"：
  "黑洞是个很黑的洞，什么都进去了出不来"

模型已经不会生成这种幼稚答案了
→ 还在拼命学"不要说这种话"
→ 在一个不存在的问题上浪费训练信号
→ 模型在和一个"不再是自己"的旧自己较劲
```

解法：Online Iterative DPO

```
Step 1: 用当前模型采样新回答
Step 2: 用 RM（或人类）标注新偏好对
Step 3: 用新数据做一轮 DPO
Step 4: 回到 Step 1
```

→ 兼顾简单性和数据新鲜度；Llama3、Qwen2.5 等均采用此策略。

Online Preference Learning 完整闭环（工业视角）：

一个更完整的在线偏好闭环通常是：点赞 / 点踩 / 停留 / 复制 → 构造 pair → replay / 过滤 / 质检 → DPO 或在线偏好优化 → eval gate → canary / rollback。

工业难点： 在线反馈并不天然等于高质量 preference；需要强过滤和校准。

6.3 DPO 变体全家族

随着标准 DPO 在工业界的广泛落地，其在特定场景下的局限性（显存占用大、依赖成对数据、分布偏移等）逐渐显现，学界和工业界演化出了一系列变体。

变体决策框架（Decision Tree）

在实际业务中选择哪种对齐算法，可以通过以下核心决策树快速定位：

Q1：系统是否有条件加载冻结的参考模型（Reference Model）？

- **没有**（显存吃紧）→ 选择 **ORPO** 或 **SimPO**
- **有** → 进入 Q2

Q2：业务收集到的反馈数据是什么类型?

- **Pairwise（成对比较：好 vs 坏）** → 选择标准 **DPO** 或 **IPO**
- **Binary（二元独立信号：好 / 坏 / 不确定）** → 选择 **KTO**
- **Online（在线实时打分反馈）** → 选择 **Online DPO**

Q3：训练数据是否存在严重的分布偏移（Distribution Shift）？

- **是**（静态数据与当前模型能力严重脱节）→ 选择 **Online Iterative DPO**
- **否** → 选择标准的 **Offline DPO** 等离线变体

五种核心变体详解

IPO（Identity Preference Optimization）—— 解决小偏好差距下的梯度消失

痛点： 标准 DPO 的损失函数基于 sigmoid 构建。当偏好对（chosen 和 rejected）之间的质量差距极小，导致两者的概率分布极其接近时，sigmoid 的梯度会无限趋近于 0，导致训练陷入停滞和不稳定。

改进机制： 放弃使用 logsigmoid 映射，直接改用**均方误差（MSE）**。

目标函数：

$$\mathcal{L}_{\text{IPO}} = \mathbb{E}\left[\left(\log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \log \frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)} - \frac{1}{\beta}\right)^2\right]$$

优势： 通过 MSE 强制拟合偏好差距，即使 y_w 和 y_l 的质量极度接近，梯度也依然稳定，不会发生梯度消失现象。

ORPO（Odds Ratio Preference Optimization）—— 告别参考模型，极限省显存

痛点： 标准 DPO 需要同时在显存中保留当前策略模型（π_θ）和被冻结的参考模型（π_{ref}），显存占用直接翻倍，极大地限制了训练的 Batch Size 或模型规模。

改进机制： 彻底移除参考模型，引入赔率比（Odds Ratio），并将 SFT 损失与偏好损失强绑定，同时优化。

目标函数：

首先计算 chosen 和 rejected 的赔率比：

$$OR_\theta(y_w/y_l) = \frac{P(y_w)/(1-P(y_w))}{P(y_l)/(1-P(y_l))}$$

最终的总损失函数为 SFT 损失加上惩罚项：

$$\mathcal{L}_{\text{Total}} = \mathcal{L}_{\text{SFT}} + \lambda \cdot (-\log \sigma(\beta \cdot \log OR_\theta(y_w/y_l)))$$

优势： 单模型单轨运行，显存消耗直接节省 **50%**。

代价： 由于失去了参考模型的 KL 散度约束，模型容易过度放飞自我，偏离原始的语言分布。因此必须依赖 $\mathcal{L}_{\text{SFT}}$ 作为强力锚点来拉扯模型。

KTO（Kahneman-Tversky Optimization）—— 打破成对数据依赖

痛点： DPO 强依赖于 (x, y_w, y_l) 的成对比较数据。但在真实的业务场景中，绝大多数用户反馈是独立的二元信号（如单纯的点个赞 Thumbs up，或踩一下 Thumbs down），极难凑齐完美的成对数据。

改进机制： 不需要 Pair，仅凭 Binary Signal 就能训练。其灵感来源于行为经济学中的**前景理论（Prospect Theory）**——人类对"损失"的敏感度远大于对"收益"的敏感度。

目标函数：

$$\mathcal{L}_{\text{KTO}} = \mathbb{E}\left[1 - \sigma\left(\beta \cdot \left(\log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)} - z_{\text{ref}}\right)\right)\right]$$

（注意：实际计算中会带有权重 λ_y）

核心设计：

- 权重不对称：设置惩罚权重 $\lambda_{\text{lose}} > \lambda_{\text{win}}$，即"惩罚一个坏回答"比"奖励一个好回答"对模型演进更重要。
- z_{ref}：用于估计当前模型与参考模型之间的 KL 散度。

适用场景： 拥有海量廉价的用户点赞/点踩日志，但缺乏高质量人工成对标注的业务线。

SimPO（Simple Preference Optimization）—— 极致简洁的参考无关算法

痛点： DPO 依赖参考模型给出的 `log_prob` 来进行隐式归一化。但在很多情况下，预训练参考模型的 `log_prob` 并不完全与人类认为的"质量"正相关。

改进机制： 不仅去掉了参考模型，更直接使用了**长度归一化的平均 log_prob**，并人为引入了 Margin 机制。

目标函数：

$$\mathcal{L}_{\text{SimPO}} = -\log \sigma\left(\beta \cdot \left(\frac{1}{|y_w|}\log \pi_\theta(y_w|x) - \frac{1}{|y_l|}\log \pi_\theta(y_l|x) - \gamma\right)\right)$$

核心设计：

- **长度归一化（除以 $|y|$）**：强力消除语言模型的长度偏见（长序列的总概率天然低，短序列天然高）。
- **引入 Margin γ**：强制要求模型给出的 chosen 概率不仅要大于 rejected，还必须大出一个 γ 的安全余量，保证偏好区分度。

当前地位： 作为 Reference-free 家族的新星，SimPO 因为其极简的设计和消除长度偏见的特性，在多数基准测试中已经反超标准 DPO。

Online DPO —— 彻底解决分布偏移难题

痛点： 无论是 DPO 还是上述变体，如果是 Offline 离线训练，数据的分布是静态的。当模型训练几步变强后，它自己生成的话术风格已经变了，但仍然在用旧模型生成的数据做对齐，这被称为"分布偏移（Distribution Shift）"。

改进机制： 将对齐做成一个持续迭代的飞轮（Iterative loop）。

- Step 1：用当前最新的策略模型针对 Prompt 池采样生成新的回答。
- Step 2：调用强力的外部 RM 或 LLM-as-Judge 对新回答进行实时打分，构建新的偏好对。
- Step 3：用这批新鲜出炉的数据做一轮 DPO 更新。
- Step 4：循环回 Step 1。

优势： 训练数据永远来自 Current Policy，实现**零分布偏移**，追求最高质量的模型能力上限。

代价： 极度昂贵。每轮更新都需要大量算力去做推理采样和判别打分。工程实践中一般妥协为每隔 100~1000 个 Step 更新一次数据集，而非严格的每步更新。

变体全景对比决策表

变体名称	是否需要 Ref 模型	数据类型要求	解决分布偏移	核心机制与适用场景
标准 DPO	**是**	Pairwise（成对）	否	**通用基线**：算法成熟，生态好，适合算力和数据规范的场景。
IPO	**是**	Pairwise（成对）	否	**应对小偏好差**：解决 DPO 梯度消失问题，适合难辨优劣的极高难度任务。
ORPO	**否**	Pairwise（成对）	否	**极限省显存**：SFT+对齐一步到位，适合显存资源吃紧的平民玩家。
KTO	**是**	Binary（二元）	否	**残缺数据救星**：无需成对数据，完美契合真实产品中只有点赞/点踩反馈的场景。
SimPO	**否**	Pairwise（成对）	否	**防长度偏见**：引入长度归一化与 Margin，追求极致简洁与超越 DPO 的性能。
Online DPO	**是**	Pairwise（生成）	**是**	**追求极致上限**：实时生成实时打分，彻底解决分布偏移，土豪公司冲榜必备。

6.4 高级话题：Implicit Reward、噪声标签与动态 KL

Implicit Reward 诊断

DPO 的一个重要直觉是：

$$r(x, y) = \beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}$$

你可以直接把 policy 相对 ref 的对数优势，看成一个隐式 reward，再去观察：

- chosen / rejected 的分离度
- 哪类样本 reward margin 太小
- 哪类样本被过分放大

噪声标签下的 DPO 稳定化

噪声标签下最常见的补救办法： filtering、cDPO / rDPO、dynamic beta、低置信样本降权。

核心思路： 不要让模糊或错标 pair 在 loss 里拥有和高置信样本同样的力量。

更细一点的适用边界：

- **cDPO / conservative DPO**：更适合偏好标签明显噪声较大、但又不想完全丢弃样本的场景。

- **rDPO / robust DPO**：更适合存在系统性错标或长尾异常 pair 的场景。
- **filtering**：最适合 judge / verifier 已经能较稳定识别坏 pair 的情况。
- **dynamic beta**：适合样本置信度差异很大的数据集。

> **一句话**：噪声标签下最危险的不是"学不到"，而是"把错偏好学得特别认真"。

Dynamic KL / Dynamic Beta 与 -DPO

为什么固定 beta 常常不够：

- 高置信偏好样本适合放松约束；
- 模糊、异常、错标样本更适合收紧约束。

更合理的做法： 做 instance-level beta 或 dynamic KL control；本质上是在让模型对"值得学的样本"和"危险样本"使用不同更新强度。

和 -DPO 的关系：

- dynamic beta 更像是在调"每个样本愿意偏离 reference 多远"；
- ε-DPO 则更像是在目标或间隔设计里显式留出一个稳定区间，避免模型在模糊样本上过度拉开差距。

实战直觉： 高置信偏好样本：可以更激进；模糊、长度极端、judge 分歧大的样本：应该更保守。

-DPO 的更稳妥解释：

- 它的核心思想不是让所有 pair 都被无限制地拉开；
- 而是给偏好优化留一个"安全间隔"或"容忍区间"，避免模型在本来就模糊的 pair 上用力过猛。

为什么这很重要： 在真实偏好数据里，有不少 chosen / rejected 其实差距没那么大；如果还强行把它们推到非常分离，模型反而会学出伪确定性。

更公式化一点的记法：

- 标准 DPO 更像持续推动 preference margin 变大；
- ε-DPO 则相当于给这个 margin 一个"够了就停"的软阈值。

如果要在面试里把 -DPO 讲得更像正式变体： 可以说它本质上是在 preference margin 上引入一个"容忍区间"；当 chosen / rejected 的相对优势已经超过某个阈值后，继续放大这对样本的收益会变小；这样可以减少模型在边界样本和模糊样本上的过拟合倾向。

> **一句话**：`dynamic beta` 解决"不同样本该偏多远"；`ε-DPO` 解决"偏到什么程度就该停"。

什么时候不该只用 DPO

以下场景 DPO 并非最优选择：

- reasoning 任务需要探索；
- tool trajectory 需要环境反馈；
- verifier 驱动任务更适合结果可验证优化；
- agent 长轨迹往往需要过程 credit assignment。

> **一句话**：DPO 擅长离线偏好校正，不擅长代替所有在线探索。

6.5 工业前沿视角与高频面试追问

DPO 更像离线对齐底座，而不是 RLHF 的简单替代品（2025–2026）

> 🏭 **工业补充**
>
> **DPO 的工业定位越来越清晰：** 当你有较稳定的偏好对数据、又不想引入完整 RL 系统时，DPO 是最顺手的对齐底座；但它非常依赖数据分布的新鲜度。
>
> **为什么 Online Iterative DPO 在 2025-2026 又变得重要：** 模型本身在变强；旧的 rejected answer 很快失真；如果 preference dataset 不刷新，DPO 其实是在和"旧自己"较劲。
>
> **更现实的工业组合：**
>
> - 用 **SFT / DPO** 做第一层风格与偏好对齐；
> - 再在高价值任务上追加 online RL 或 verifier-based optimization。
>
> **资深工程师视角：** 不要把 DPO 理解成"更优雅的 RLHF 终局"；更准确的理解是：它是一个**训练稳定、实现成本低、适合离线偏好对齐的基座层**。

高频追问标准答法

Q：DPO 里的 β 到底控制什么？

> β 决定了策略相对 reference 的偏离强度。β 大，说明更保守，更贴近 reference；β 小，说明更激进，更愿意为了偏好数据拉开分布差距。所以它本质上是在控制"对齐强度"和"分布漂移风险"的平衡。

Q：什么时候会优先用 DPO，而不是直接上 RLHF？

> 有比较干净的 preference pair、又想要稳定训练和较低系统复杂度时，DPO 很合适。如果任务更依赖在线探索、可验证奖励或者复杂 credit assignment，RLHF / reasoning RL 更有优势。可以把 DPO 理解成离线偏好对齐的强基线，而不是所有对齐问题的终局。

---

7. GRPO 与后继算法

7.1 GRPO：用组内平均替代 Critic

动机与核心设计

PPO 需要 Critic 模型来估算"平均得分基线"，但 Critic 本身就是一个大模型，带来严重的显存压力。

GRPO 的灵魂问题： 能不能用更简单的方式算出基线，直接省掉 Critic?

核心思路： 对同一个问题，让 Actor 生成 G 个不同回答，用小组平均分做基线：

```
responses = [r1, r2, r3, ..., rG]
rewards   = [RM(r1), RM(r2), ..., RM(rG)]
baseline  = mean(rewards)   # 小组平均分就是基线

advantage_i = r_i - mean(r_1, ..., r_G)
```

不需要 Critic 神经网络实时预测基线，直接采样小组，现场算平均分。

PPO vs GRPO 完整对比

维度	PPO	GRPO
模型数量	4 个	3 个（省掉 Critic）
基线来源	Critic 神经网络	组内平均 RM 分数
显存压力	极大	显著降低（约 25%）
实现复杂度	高	低
训练稳定性	高	同样高
适用场景	通用 RL	可验证奖励任务

GRPO 伪代码

```
for 每个问题 x:
    # 生成 G 个回答（G 通常取 8~16）
    responses = [Actor采样() for _ in range(G)]

    # 每个回答算分
    rewards = [RM打分(r) for r in responses]

    # 组内平均作为基线
    baseline = mean(rewards)
```

```
    for i, response in enumerate(responses):
        advantage = rewards[i] - baseline
        # advantage > 0: 比组内平均好，强化
        # advantage < 0: 比组内平均差，抑制
        update(Actor, advantage)
```

工程陷阱：梯度真空与课程学习

场景 A：题目太难，G 个回答全错

```
rewards   = [0, 0, 0, 0, 0, 0, 0, 0]
advantage = 0 - 0 = 0
→ 没有梯度信号，模型不知道往哪走
```

场景 B：题目太简单，G 个回答全对

```
rewards   = [1, 1, 1, 1, 1, 1, 1, 1]
advantage = 1 - 1 = 0
→ 同样没有梯度信号，白白浪费算力
```

有效训练信号只存在于：同一批回答里，有对有错。

解法：课程学习（Curriculum Learning）

- 训练初期：只喂"中等难度"题目 → 模型有概率答对，有概率答错 → advantage 有正有负，梯度信号丰富
- 随着模型变强：动态调高难度 → 始终维持"部分对、部分错"的状态 → 始终在"跳一跳够得着"的区间

Binary Reward 下的 Advantage 归一化问题

在 `0/1 reward` 下，组内标准差可能非常不稳：全对/全错时方差为 0；少量样本时，std 极小会导致梯度异常放大。

这就是为什么 Dr.GRPO 会谨慎对待 `/std` 归一化：它在某些推理任务上确实会引入额外脆弱性。

更正式的直觉表达：如果组内奖励是 $r_i \in \{0, 1\}$，advantage 常写成：

$$A_i = \frac{r_i - \bar{r}}{\sigma_r + \epsilon}$$

当"全对""全错"或"几乎全一样"时，σ_r 会非常小；这会让 advantage 的数值要么退化成 0，要么被极端放大。

工程上的缓解方式：

- 对极小方差组直接丢弃；
- 只减组均值，不除标准差；
- 对 denominator 加更强的最小值截断。

7.2 REINFORCE++：轻量化策略梯度

背景与动机

- **PPO** 需要 **Critic**，显存占用大，系统复杂
- **GRPO** 需要 **G 个采样**，计算量大
- 自然会问：**有没有不需要 Critic、也不需要多采样的更简单方案？**

答案：REINFORCE++

REINFORCE（最原始的策略梯度）的基本形式：

$$\mathcal{L} = -\mathbb{E}[\log \pi_\theta(a|s) \cdot R]$$

其中 $\pi_\theta(a|s)$ 是策略模型对动作/token 的概率，R 是实际获得的累计回报（或序列级奖励）。

优点： 形式最简单，不需要 Critic。**核心问题：** 方差极大，训练容易不稳定，奖励尺度和序列长度都会影响训练质量。

REINFORCE++ 的五个关键改进

改进 1：Baseline / Group Baseline（降低方差）

把原来的回报 R 改成优势形式 $R \to R - b$：

$$\mathcal{L} = -\mathbb{E}[\log \pi_\theta(a|s) \cdot (R - b)]$$

b 可以是 mini-batch 平均奖励，也可以是 group 内平均奖励——这和 **GRPO 用组内均值做 baseline** 的思想一致。**作用：** 显著降低梯度方差，让训练更稳定。

改进 2：Reward Normalization（奖励归一化）

$$R \to \frac{R - \text{mean}(R)}{\text{std}(R)}$$

或对 $R - b$ 再做标准化。**作用：** 防止不同 batch 的奖励尺度差异过大，避免梯度忽大忽小，提高训练稳定性。

改进 3：Token-level 归一化（消除长度偏差）

将序列级损失按 token / 序列长度做归一化，而不是让长回答天然累积更多梯度。**作用：** 消除长度偏差，避免模型单纯因为回答更长而得到更大更新——这一点和 **DAPO / Dr.GRPO** 的思

路一致。

改进 4：Clip（借鉴 PPO 的稳定性）

对 importance sampling ratio 做 clip，限制策略更新幅度，防止一步走太大。可理解为引入 PPO 风格的稳定机制，但**不需要单独训练 Critic**。**作用：** 防止策略突变，训练更稳，在实践中比纯 REINFORCE 更可靠。

改进 5：Token-level KL Penalty

对每个 token 单独计算相对参考模型的 KL，而不是只看整条序列的 KL。整体目标可以写成：

$$\mathcal{L} = -\mathbb{E}[\log \pi_\theta \cdot (R - b)] + \text{clip 项} - \beta \cdot \mathrm{KL}_{\text{token}}$$

其中 b 是 baseline，β 是 KL 惩罚系数，$\mathrm{KL}_{\text{token}}$ 是 token 级 KL 约束。**作用：** 防止策略偏离 reference 太远，相比序列级 KL 更细粒度、更稳定。

REINFORCE++ 的核心特点与适用场景

相比 PPO / GRPO 的优势：

- **不需要 Critic**：省掉一个价值模型，显存和实现复杂度更低
- **不需要 G 个采样**：每个 prompt 采样 1 个输出就能训练，计算成本低于 GRPO
- **直接使用真实回报 R**：不需要复杂的 advantage estimator，训练链路更直接
- **实现简单**：通常只需要 **Actor + Reference + Reward Model**，很适合资源受限环境

REINFORCE++ 是对多类方法优点的"折中整合"：

- 从 REINFORCE 继承：简单、直接、无需 Critic
- 从 GRPO 借鉴：baseline / 组内均值降方差
- 从 PPO 借鉴：clip 稳定更新
- 从 DAPO / Dr.GRPO 借鉴：token-level 归一化
- 同时加入：token-level KL penalty 来约束策略漂移

> **一句话**：既不想像 PPO 那样维护 Critic，也不想像 GRPO 那样做多采样，那就用 REINFORCE++。

特别适合：

- 资源受限（只有 3 个模型：Actor / Ref / RM）
- 快速原型验证（想尽快跑通训练闭环）
- 工程实现优先（希望训练框架简单、容易调试）
- 短序列任务（因为长序列下原始 REINFORCE 的方差问题会更严重）

工业边界：

- 在 reasoning RL 里，维护高质量 critic 的成本很高；如果 baseline 可以通过更轻量方式得到，REINFORCE++ 会显得很有吸引力。
- 更适合 critic 成本过高、rollout 代价高、更在意工程简洁而不是最完整 value 建模的场景。

7.3 DAPO / VAPO / Dr.GRPO / GSPO 算法详解

DAPO：字节 2025 年对 GRPO 的四个改进

改进 1：Clip-Higher（不对称裁剪）

GRPO 继承 PPO 的对称 Clip：ratio $\in [1-\varepsilon, 1+\varepsilon] = [0.8, 1.2]$，好回答和坏回答的学习速度一样快。

问题： 发现好的推理路径 → 需要快速强化，让模型记住；对称 Clip 限制了好回答的学习速度。

DAPO 的 Clip-Higher：

- 好回答（advantage > 0）：上限提高到 $1+\varepsilon_{\text{high}}$（如 1.5）
- 坏回答（advantage < 0）：下限保持 $1-\varepsilon_{\text{low}}$（如 0.8）
- 不对称范围：$[0.8, 1.5]$ → 好回答学得更快，坏回答抑制保持稳定

改进 2：Dynamic Sampling（动态过滤）

GRPO 的梯度真空问题解法是课程学习（手动控制难度）。DAPO 改为自动过滤：

```
采样 G 个回答后：
if 全对 or 全错 → 丢弃这道题，换一道
只保留"有对有错"的题目
```

效果：不需要手动设计课程，系统自动过滤无效训练信号，不需要人工调难度。

改进 3：Token-level Policy Gradient Loss

GRPO 默认对整个回答算一个 advantage：

```
长回答（2000 token）每个 token 梯度 = advantage/2000
短回答（50 token）每个 token 梯度  = advantage/50
→ 短回答每个 token 梯度是长回答的 40 倍
→ 模型系统性偏爱短回答
```

DAPO 修正：按 token 数量归一化：`token_advantage = advantage / len(response)` → 长短回答每个 token 贡献相等。

改进 4：去掉 KL 散度惩罚

DAPO 发现 KL 约束在推理任务上有害：推理需要模型大幅改变行为，从"直接输出"变成"先 `<think>` 再输出"；KL 约束阻止这种行为变化 → 阻碍推理能力涌现。

DAPO： 直接去掉 KL 项，只保留 Clip-Higher 的比率约束，让模型自由探索推理空间。

VAPO：Value Pretraining 解决基线失真

问题：组内平均分对题目难度一无所知

GRPO 基线 = mean($r_1, \ldots, r_G$)。极端案例：

```
训练初期，模型很弱，采样 8 个回答：7 个错、1 个对

对的：advantage = 1 - 0.125 = +0.875（强烈强化）
错的：advantage = 0 - 0.125 = -0.125（轻微抑制）

惩罚错误的力度 远小于 奖励正确的力度
不对称 → 学习效率低
```

VAPO 解法：预训练 Value 网络

- Step 1：收集（问题，部分推理链，最终结果）数据，不需要人工标注，用验证器自动打分
- Step 2：预训练 Value 网络，输入：问题 + 前 k 步推理，输出：最终答案正确的概率
  - 例："设方程 x+y=10，x-y=2..."（前 2 步）→ Value 网络预测：成功率 85%
- Step 3：用 Value 网络作为基线：$\text{advantage} = r_i - V(s_i)$，$V(s_i)$ 是对当前推理状态的全局估值
  - 简单题 → $V(s) \approx 1.0$ → advantage ≈ 0（不浪费在简单题）
  - 难题 → $V(s) \approx 0.1$ → advantage 有意义（给强信号）

Decoupled GAE：分离过程和结果奖励

传统 GAE 问题：LLM 推理链中，奖励只在最后一步给出，中间步骤 $r_t = 0$，导致早期步骤的 advantage 估计很不准。

VAPO 的 Decoupled GAE：

- 结果奖励（ORM）：只在最后一步
- 过程奖励（PRM）：每步都给信号
- 两套 GAE 分别计算，加权融合：

$$\text{advantage} = \alpha \cdot \text{GAE}_{\text{outcome}} + (1-\alpha) \cdot \text{GAE}_{\text{process}}$$

→ 早期步骤也能得到有意义的梯度信号 → 特别适合长推理链（30 步以上）

Dr.GRPO：三个系统性偏差的修正

偏差 1：样本数量偏差

```
G=8，7 对 1 错：错的 advantage ≈ -3.5
G=4，3 对 1 错：错的 advantage ≈ -1.8
同样"只有一个错"，梯度差了 2 倍
→ 梯度受 G 影响，不可比
```

Dr.GRPO 修正： 去掉 std 归一化，改用固定 reward scale：$\text{advantage}_i = r_i - \text{mean}(r)$（只减均值，不除 std），加采样权重修正：$\text{weight}_i = 1/(G \cdot P(\text{问题被采样到}))$，保证不同 G 下梯度期望值一致。

偏差 2：长度偏差

长回答每个 token 的梯度 << 短回答每个 token 的梯度 → 模型倾向于生成短回答。

Dr.GRPO 修正： 按 token 数量归一化 advantage：`token_advantage_i = advantage_i / len(response_i)` —— 和 DAPO 的 Token-level 思路一致，独立发现。

偏差 3：难度偏差

全对/全错 → advantage 全 0（显性梯度真空）；但"8 个回答里 7 个几乎一样"→ 隐性梯度真空。

Dr.GRPO 修正（比 DAPO 更精细）： 不只判断"全对/全错"，计算 advantage 的方差：

```
if var(advantage) < threshold:
    # 梯度信号无效，丢弃
```

能捕捉到隐性梯度真空，比 DAPO 的二元判断更鲁棒。

GSPO：Sequence 级 RL

问题：Token 级 ratio 的内部不一致

GRPO/DAPO 在 token 级别计算 ratio：$\text{ratio}_t = \pi_\theta(a_t|s_t)/\pi_{\text{old}}(a_t|s_t)$。

同一个回答里：token_1 的 ratio = 1.8（变化很大），token_2 的 ratio = 0.95（几乎没变）——同一个回答被"区别对待"。

副作用： 模型可能在坏回答里插入几个"好 token"来规避 Clip 惩罚 → token 级别的 Reward Hacking。

GSPO 的解法：Sequence 级 ratio

不对每个 token 算 ratio，而是对整个回答算一个 ratio：

$$\text{ratio}_{\text{seq}} = \frac{\pi_\theta(y|x)}{\pi_{\text{old}}(y|x)} = \exp\left(\sum_t \left(\log \pi_\theta(a_t|s_t) - \log \pi_{\text{old}}(a_t|s_t)\right)\right)$$

一个回答只有一个 ratio，要么整体被强化，要么整体被抑制 → 和"回答整体质量"的语义对齐 → 消除了 token 级别的内部不一致。

更正式的对比写法：

- token-level 方法：$r_t = \dfrac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\text{old}}(y_t \mid x, y_{<t})}$，对单 token 波动极敏感
- sequence-level 方法：$R = \dfrac{\pi_\theta(y \mid x)}{\pi_{\text{old}}(y \mid x)}$，更接近整条回答的全局质量

为什么这对 MoE 很关键： token-level ratio 会把路由波动、局部长度偏差、局部异常 token 一起放大；sequence-level ratio 更容易和"回答整体好不好"对齐。

7.4 五代算法综合对比与工程基础设施

五代算法完整对比表（面试必备）

维度	GRPO	DAPO	VAPO	Dr.GRPO	GSPO
提出方	DeepSeek	字节	阿里	独立研究	阿里 Qwen
基线来源	组内均值	组内均值	Value 网络	组内均值（修正权重）	组内均值
Clip 粒度	Token 级对称	Token 级不对称	继承 DAPO	Token 级（修正归一化）	Sequence 级
梯度真空解法	课程学习	Dynamic Sampling	Value 网络规避	方差过滤	Dynamic Sampling
长度偏差	有	修正	修正	修正	从根本消除
KL 约束	有	去掉	去掉	有（可选）	有
核心创新	省掉 Critic	不对称 Clip	Value 预训练	三偏差修正	Sequence 级 ratio
数学严谨性	一般	较好	好	最严谨	好
工程复杂度	低	低	高	低	中
主要使用方	DeepSeek	字节 Seed	阿里 Qwen	学界	阿里 Qwen

一句话区分五代（面试速查）：

- **GRPO**：省掉 Critic，组内平均做基线，工程最简单
- **DAPO**：不对称 Clip + 动态过滤，字节自研，工程友好
- **VAPO**：把 Critic 请回来做 Value 预训练，基线估计更准
- **Dr.GRPO**：数学推导三个偏差（样本/长度/难度），最严谨的修正
- **GSPO**：ratio 从 token 级提升到 sequence 级，消除内部不一致

开源 RL Infra 快照（2025–2026）

- **veRL（ByteDance Seed / volcengine）**：官方仓库把自己定义为 **flexible, efficient, production-ready RL training library**，核心是 **HybridFlow** 编排思想，支持 **FSDP / FSDP2 / Megatron-LM** 训练后端与 **vLLM / SGLang / HF Transformers** rollout 后端，并显式支持 **PPO、GRPO、GSPO、REINFORCE++、DAPO、Dr.GRPO**、多轮 tool calling 与 VLM RL。
- **OpenRLHF**：官方 README 把它定位成基于 **Ray + vLLM + ZeRO-3 + Transformers** 的高性能 RLHF 框架，强调 **Actor / Reward / Reference / Critic 分离部署**、**Hybrid Engine** 共卡调度，以及 **Async Agentic RL**。

> **工程结论：** 2025 之后面试里如果只会讲损失函数，往往不够。更完整的回答是：**算法 = policy update，Infra = rollout scheduling + resource placement + verifier/reward serving + checkpoint/eval gate**。

GRPO / DAPO 可运行级实现框架

一个最小实现骨架至少要包含：

- 当前策略 `log_probs`
- reference `ref_log_probs`
- group rewards
- 组内 baseline / normalized advantage
- policy ratio
- clip
- 可选 KL 附加项

真正能区分熟练度的点不是把公式背出来，而是你能把这几个张量在代码里对齐：`B x G`、token-level / sequence-level、chosen / rejected / reference。

Reward Shaping 实战权重样例

一个常见的 reasoning / agent reward stack 例子：

- `correctness`： `+2.0`
- `format`： `+0.5`
- `efficiency`： `-0.2 * extra_steps`

- `safety`：硬门控或大幅负分
- `length`：落在预算区间内再给小幅奖励

原则：

- correctness 永远是主锚点；
- format / safety 先做 gate；
- efficiency 和 length 更适合做软约束。

Rollout-efficient / Noise-corrected GRPO 变体方向

为什么这类变体在 2025-2026 变重要： rollout 成本越来越高；同时 group-based RL 又很容易被噪声、方差和旧样本污染。

这类方法通常在做三件事：

- **减少无效 rollout**：过滤全对 / 全错 / 极低方差 prompt
- **修正噪声样本**：对低置信 reward、异常长度、异常格式样本降权
- **保留高价值历史样本**：通过 partial rollout / replay / freshness 规则降低重复采样成本

如果面试官追问"具体会怎么做"，可以补三个工程动作：

- **variance filtering**：对组内 reward 方差过低的 prompt 直接跳过
- **confidence-aware weighting**：对 judge / verifier 不稳定样本降权
- **trajectory reuse with freshness bound**：只在 freshness 还可接受时复用高价值轨迹

这类变体真正要解决的是三件事： rollout 太贵；组内信号太噪；旧样本复用容易把训练带偏。

如果要把它讲到"接近论文层"，可以把这类方法统一理解成在优化三个维度：

- **sample efficiency**：少采样也能学到东西
- **signal quality**：尽量减少低方差、低置信、低价值组
- **freshness control**：历史轨迹复用但不过度 off-policy

所以它们并不是另一条完全不同的 RL 路线，更像是：在 GRPO 这个骨架上，把 rollout 成本和噪声问题往工程上压下去。

7.5 工业前沿视角与高频面试追问

Sequence-level 更新正在成为 reasoning RL 主线（2025–2026）

> 🏭 **工业补充**

> **过去很多方法按 token 更新，是因为实现直接、和 PPO 框架兼容**；但 reasoning / MoE / long CoT 场景暴露了 token-level ratio 的两个问题：一条长链里局部 token 噪声会被放大；不同专家路由、不同思考长度会让 token 级 importance ratio 变得非常抖。
>
> **GSPO 的重要性就在这里：** 它把更新粒度抬到 sequence level；在长序列和 MoE 上，更新信号通常更平滑，也更贴近"整条回答质量"的语义。
>
> **工业框架层面的意义：** `veRL`、`OpenRLHF` 这类框架开始原生支持 **PPO / GRPO / GSPO / REINFORCE++ / DAPO / Dr.GRPO**；这意味着 reasoning RL 不再只是论文比较，而是进入了真正的工程试验平台阶段。
>
> **一句话**：2025-2026 的变化不是"GRPO 火了"这么简单；更深的一层是：**更新粒度正在从 token 走向 sequence，从理论技巧走向工程友好。**

高频追问标准答法

Q：GRPO 相比 PPO 的核心价值是什么？

> 它把 Critic 去掉了，改成用组内平均 reward 做基线。这样直接省掉了 value model 的训练和维护成本，系统更轻。对 reasoning task 来说，这很重要，因为长链推理下 Critic 本身就很难学准。

Q：怎么一句话区分 DAPO / VAPO / GSPO？

> **DAPO**：在 GRPO 框架里重点修 token 级长度偏置和采样效率。**VAPO**：把 value pretraining 带回来，增强长链推理里的基线质量。**GSPO**：把优化粒度抬到 sequence level，更适合长 CoT 和大 MoE。

---

8. 推理模型训练范式

8.1 推理增强训练基础：奖励机制与任务分类

传统 RLHF vs 推理增强训练

维度	传统 RLHF	R1/GRPO
奖励来源	人类偏好（主观）	客观验证器（数学/代码/逻辑）
目标	让人觉得你好	答案真的对
天花板	人类标注员水平	理论上无上限，可超越人类
适用任务	通用对话、创意写作	数学、代码、逻辑推理

Outcome-based Reward（结果导向奖励）

- 数学题：答案对了 → +1，答案错了 → 0
- 代码题：代码跑通测试 → +1，编译报错 → 0
- 评委是编译器/数学验证器，不是人类

为什么这样的天花板更高？

- 传统 RLHF 的老师是人类，上限是人类的认知边界
- GRPO 的老师是"真理本身"，理论上可以无限探索

ORM vs PRM

类型	全称	打分对象	特点
ORM	Outcome Reward Model	最终答案	信号稀疏，实现简单
PRM	Process Reward Model	思维链每一步	信号密集，实现复杂，需要步骤级标注

PRM 的优势： 推理到一半如果出错 → 对这一步扣分，而不是等到最后才发现整个推理链是错的 → 更细粒度的训练信号。

Verifiable vs Unverifiable 任务

- **Verifiable Reward（可验证奖励）**← R1/o1 适用：数学、代码、逻辑，有唯一正确答案，机器可验证
- **Unverifiable Reward（不可验证奖励）**← 必须回归人类判断：写诗、咨询、创意写作、商业文案，答案空间无限，好坏因人而异，无法量化

结论：两条路线长期共存，无法互相替代。

8.2 DeepSeek-R1 完整训练流程与 Rejection Sampling SFT

五阶段完整流程

阶段 1：Cold-start SFT

- 少量 CoT 数据（几千条）
- 植入 `<think>...</think>` 思维链格式
- 不做这步，GRPO 采样时模型不会生成思维链，梯度真空

阶段 2：GRPO（推理激活）

- 数学/代码可验证奖励

- 课程学习控制难度
- 模型开始"自己发现"推理路径

阶段 3：Rejection Sampling SFT（自我蒸馏）

- 用当前模型大量采样
- 只保留答对的回答（含完整思维链）
- 用这些"自己的最优解"再做一轮 SFT
- 不存在"天花板"问题（从更强的自己身上学习）

阶段 4：GRPO + 偏好信号融合

- 推理能力（GRPO）+ 安全对齐（偏好信号）并行训练
- 注意：串行容易产生 Alignment Tax（对齐税）
- 对齐可能让模型缩短思维链，推理能力下降

阶段 5：Online Iterative

- 动态采样新数据，防分布偏移
- 模型持续自我进化

Rejection Sampling SFT 为什么没有天花板？

GPT-4 生成 SFT 数据：

- 天花板 = GPT-4 能力（外部固定）
- 信息经过两次有损压缩

Rejection Sampling SFT：

- 数据来自 GRPO 之后涌现的能力
- GRPO 让模型发现了人类没有教过的推理路径
- 天花板随模型能力持续上升

这叫 **Self-improvement（自我提升）**，是 R1 超越人类数学水平的根本原因之一。

8.3 RLVR 理论体系

形式化定义与边界

RLVR（Reinforcement Learning with Verifiable Rewards）的核心命题是：对于存在客观验证器的任务，直接使用客观事实和验证结果作为奖励信号，从而摆脱对人类偏好标注和奖励模型（RM）的依赖。

定义： 给定任务 x、回答 y 和确定性的客观验证函数 V。奖励信号：

$$r(x, y) = V(x, y) \in \{0, 1\}\text{（或连续值）}$$

客观性体现： 数学任务看符号计算验证最终答案；代码任务看单元测试通过率；逻辑任务看形式化推理检验。

与 RLHF 的本质区别：

	RLHF	RLVR
奖励来源	人类主观偏好，高度依赖 RM	客观事实，理论上无法被 Hack
Goodhart's Law 风险	极易受影响（被 Hack）	极低
适用任务	写作、对话等开放任务	数学、代码等闭合任务

核心争议（面试高频考点）：RLVR 真的"学会了推理"吗？

2025 年学界的谨慎共识是：**RLVR 能够显著提升模型在可验证任务（Verifiable Tasks）内的表现，但通用推理能力的涌现仍有争议。**

能力来源： RLVR 无法凭空创造新知识，它实质上是**激活了基础模型（Base Model）在预训练中已经拥有、但不常使用的推理路径**。如果基础模型完全缺乏相关推理能力，RL 将无法"无中生有"。基础模型的预训练质量直接决定了 RL 的上限——这也是为什么 DeepSeek-R1 在 GRPO 之前，必须先用强模型做 Cold-start SFT。

泛化争议： 模型在可验证任务上的提升，是否能跨任务迁移（Task Transfer）到不可验证任务？其输出的推理链是真实的思考（Faithfulness），还是仅仅是对高分模式的记忆搜索与压缩？这仍需在独立的 Held-out 任务上寻找更多证据。

8.4 Overthinking 控制策略与 Thinking Budget 管理

Overthinking / Panic / Self-doubt 三类失控模式

- **Overthinking**：token 成本持续暴涨，但正确率不涨；模型学会了"表演思考"，在简单问题上也生成极长的、无意义的思维链（例如反复输出"让我仔细想想...首先...其次...因此答案是 2"）。
- **Panic**：在预算或时间压力下，模型突然放弃高质量推理，过早收尾。
- **Self-doubt**：长链后半程开始反复自我否定，甚至把原本正确路径推翻。

为什么这些问题重要： 2025-2026 的 reasoning model 已经不是"能不能想"，而是"会不会失控地想、错位地停、后半程崩掉"。

Overthinking 的成因： 在 GRPO 训练中，长思维链通常隐含对应着高奖励，模型学会了"表演思考"来刷长度。**危害：** Token 效率（正确率提升 / Token 数增加）极低；导致推理成本飙

升；极度占用上下文窗口并破坏用户体验。

训练期控制（Training-Time Control）

方案 A：长度惩罚（Length Penalty）

$$\text{reward} = R_{\text{base}} - \alpha \cdot \max(0, \text{len} - \text{budget})$$

逻辑：超出预算长度后线性扣分。**缺陷：** 可能导致模型为了节省 Token 而过度偷懒，截断了真正需要的长链推理。

方案 B：预算感知/超长整形（Budget-aware / Overlong Reward Shaping）

逻辑（DAPO 思路）：如果回答超长，不给负分惩罚，而是直接让 $\text{reward} = 0$（在预算内解出则正常给分）。**优势：** 一种比直接扣分更温和的方案，模型不会故意拉长字数，也不会因为怕扣分而强行截断。

方案 C：格式奖励（Format Reward）

逻辑：设定严格的格式约束（如必须包含 `</think>` 结束标志）。如果未按格式结束，则直接降低或取消 Reward。这能迫使模型主动学习"决定何时停止"。

推理期控制（Test-Time Control）

方案 A：思考预算（Thinking Budget / Max Tokens）

逻辑：直接设定 `max_think_tokens`，达到上限强制截断或强行输出 `</think>`。**缺点：** 容易破坏正常的推理链。

方案 B：难度感知分配（Difficulty-Aware Allocation）

逻辑：根据问题难度动态分配 Token 预算——简单问题给小预算（如 512 Token），复杂问题给大预算（如 4096 Token）。

探测方式：可以引入外部小模型快速分类难度；或者监控模型 Token 概率分布的熵（熵低代表极度自信，应快速回答；熵高代表不确定，允许深度思考）。

方案 C：早停机制（Early Stopping Rules）

逻辑：在生成过程中，如果检测到推理链陷入无意义的循环重复，或者 PRM / 中间步骤置信度判定当前推理已经"足够好"，则强制提前终止思考并输出答案。

Thinking Budget 的关键指标与前沿思路

thinking budget 的关键不是 token 上限本身，而是：哪类题值得多想、想多久收益还在上升、什么时候应该提前停。

更实用的指标： accuracy@budget、tokens-per-solved、stop reason 分布。

Reasoning Path Compression / Extrapolation：

- 更长的链不一定更适合部署；工业上往往先用长链探索把高质量路径找出来，再做 compression，把昂贵路径蒸馏成更短但仍然有效的推理。
- extrapolation 则是在问：训练时见过的推理深度有限；推理时能不能在更长预算下仍然稳定工作。

Budget-conditioned Policy（前沿思路）： 让策略显式感知预算——低预算时学会"快但够用"，高预算时学会"慢但更稳"。这比单纯做硬截断更像真正的产品能力。

Reasoning Budget 的线上控制指标（上线时至少应监控）：

- 平均思考 token
- 简单题的浪费率
- 长链提前截断率
- 每 solved query 的平均 token 成本
- stop reason 分布

8.5 Qwen3：Hybrid Think/Non-Think 统一模式

背景与核心设计

早期的行业痛点在于：以 R1 为代表的"思考模型"推理强但生成极慢、成本高昂；而传统的"指令微调模型"速度快但推理能力弱。Qwen3 提出了将两者合二为一的混合架构（豆包的"思考模式"亦是同理）。

核心设计：一个模型，两种模式。 利用 Special Token 切换状态，用户可以按需在速度和质量之间取得平衡，企业也无需部署两套截然不同的模型架构：

- **开启思考（深度模式）**：包含完整的 `<think>...</think><answer>...</answer>` 流程，适合复杂推理。
- **关闭思考（快速模式）**：直接生成最终答案（即 `<answer>...</answer>`），适合日常快速对话。

训练数据构造、推理预算控制与四阶段管线

训练数据的构造与混合：

- 训练集中同时混合两类数据：将简单题目匹配短思考路径（Short CoT）甚至无思考直接回答的数据；将复杂题目匹配长思考路径（Long CoT）的数据。
- 模型通过对这些数据的学习，掌握了"根据上下文难度按需思考"的能力。

推理预算的动态控制： 在 System Prompt 中加入具体的控制指令（例如：`"请用不超过 500 个 token 的思考过程回答这道题"`，或传入 `/think`、`/no-think` 的系统参数）。经过混合训练的模型，能够极其精准地理解并服从该 Budget 指令，在限定的资源预算内规划并完成最高效的推理闭环。

Qwen3 四阶段训练的每阶段职责：

阶段	名称	职责
阶段 1	long CoT cold start	建立基本长链习惯与格式边界
阶段 2	reasoning-based RL	用可验证奖励把推理路径真正拉出来
阶段 3	thinking mode fusion	把 think / no-think 融进同一模型，避免分裂成两套系统
阶段 4	general RL	校正通用助手行为、格式、agent 能力和安全边界

官方实现补充（Qwen3，截至 2026-04 核实）

- **发布时间**：2025-04-29。
- **Hybrid Thinking**：官方将其定义为同一模型内的 **Thinking Mode** 与 **Non-Thinking Mode** 双模式切换，并强调这种设计有利于 **stable and efficient thinking budget control**。
- **Pre-training 基线**：Qwen3 官方博客写明其预训练数据约 **36T tokens**，覆盖 **119 languages and dialects**，并使用三阶段预训练把上下文扩到 **32K**；后训练版本再提供 **128K** 上下文的大模型族。
- **Post-training 管线**：官方四阶段是 **long CoT cold start → reasoning RL → thinking/non-thinking fusion → general-domain RL**；最后一阶段显式覆盖 **instruction following、format following、agent capabilities**。
- **Agent 方向信号**：官方博客直接写出 **improved agentic capabilities** 与 **stronger MCP support**，说明 reasoning model 与 agent model 在 2025 年已经开始合流。

Thinking Leakage 与 Fusion 阶段意义

Thinking Leakage（思考泄露）： 关闭思考模式时，模型仍然残留 verbose CoT 痕迹；或者 style 上明显像"想过了但没完全关掉"。

Fusion 阶段的意义： 不是锦上添花；而是防止 think / no-think 两种策略相互污染、互相破坏。

8.6 工业前沿视角与高频面试追问

Reasoning Model 已经进入"探索、蒸馏、预算控制"三段式（2025–2026）

> 🏭 **工业补充**
>
> **第一段：探索** —— 用可验证奖励做 RL，让模型自己发现有效推理路径。

> **第二段：蒸馏** —— 把这些长 CoT、verification、reflection 模式蒸馏回更易部署的模型。DeepSeek-V3 官方就明确提到：它把 **R1 的 verification / reflection 模式**蒸馏回标准 LLM，同时控制输出风格与长度。
>
> **第三段：预算控制** —— 让 reasoning 能力变成可配置产品能力，而不是实验室里的"无限思考"。Qwen3 的 **thinking / non-thinking** 双模式，本质上就是把推理能力变成了产品侧的预算接口。
>
> **资深工程师视角：** 真正难的不是让模型"能想"；而是让它在简单题上别过想，复杂题上别想不够，并且上线后还能控制 token 成本和延迟。
>
> **一句话**：今天的 reasoning training，不只是"奖励更聪明的答案"，更是"把探索出来的思考模式压缩成可部署、可控成本的能力"。

高频追问标准答法

Q：为什么说 RLVR 不能"凭空创造知识"?

> 因为 RLVR 优化的是已有能力的调用方式，不是从零注入新知识。如果 base model 在预训练里几乎没有某类推理能力，单靠 verifier reward 也很难让它无中生有。这就是为什么 cold-start SFT 和强 base model 仍然关键。

Q：Qwen3 的 Hybrid Thinking 模式为什么重要?

> 因为它把"会思考"和"可部署"放进了同一模型里。对复杂问题可以开 think mode，对简单问题可以走 non-think mode。这本质上是把 reasoning 能力变成了一个可配置的成本接口。

---

9. SFT 数据工程

9.1 数据质量基础：Length Bias、配方设计与过拟合防护

长样本主导问题（Length Bias）

问题：

```
# PyTorch 默认 Cross Entropy Loss:
batch_loss = sum(所有token的loss) / 所有token数量

# 同一个 batch 里:
# 2000 token 的代码样本 → 贡献 99.5% 的梯度
```

```
# 10 token 的对话样本  → 贡献  0.5% 的梯度
# 模型实际上只在学写代码，对话能力悄悄退化
```

解法：Per-sample Loss Normalization

```python
def per_sample_loss(logits, labels):
    losses = []
    for i in range(batch_size):
        valid_tokens = (labels[i] != -100).sum()      # 非-100的token数
        sample_loss = cross_entropy(logits[i], labels[i]) / valid_tokens
        losses.append(sample_loss)
    return sum(losses) / batch_size  # 每个样本权重相等
```

数据配方与温度采样

问题： 数据量不均衡时，直接均匀采样会导致小数据集过拟合。

解法：温度采样（Temperature Sampling）

```python
dataset_sizes = {
    "dialog":    1_000_000,
    "code":        100_000,
    "tool_use":     10_000,
}

T = 0.7  # T越小→越均匀；T越大→越按原始比例

weights = {name: size ** (1/T) for name, size in dataset_sizes.items()}
total = sum(weights.values())
weights = {name: w/total for name, w in weights.items()}

# T=0.7 时结果：
# dialog:   85%（比均匀多，但不被压制）
# code:     13%（被放大）
# tool_use:  2%（被放大，但不过拟合）
```

T 值选择：

- T → 0：完全均匀（小数据集过拟合）
- T = 1.0：完全按原始比例（小数据集被淹没）
- T = 0.7：折中，LLaMA/Qwen 的默认选择

工业级 SFT 数据配方

- **通用对话** ← 基础对话能力
- **代码** ← 推理能力和精确性
- **工具调用** ← Agent 能力
- **数学推理** ← CoT 格式植入
- **安全拒绝** ← 不能忽略，否则模型什么都答

SFT 的过拟合：Format Overfitting

SFT 的过拟合和普通深度学习不同：

- 普通深度学习过拟合：val loss ↑（明显上升，能被 Loss 曲线发现）
- SFT 过拟合（Format Overfitting）：val loss →（平稳，不上升，Loss 曲线看起来正常），但模型悄悄坏掉了：

```
过拟合前：
  问"推荐一部电影" → "我推荐《星际穿越》，因为..."

过拟合后：
  问"推荐一部电影" → "当然！以下是我的推荐：\n1. ..."
  问"今天心情不好" → "当然！以下是一些建议：\n1. ..."
  问"1+1等于几"    → "当然！以下是答案：\n1. ..."
```

模型学会了模仿训练集的格式和语气，而不是真正的能力。

正确的早停策略：

```
评估指标 = {
    "val_loss":          监控是否上升,
    "MMLU score":        监控通用能力是否退化,
    "format_diversity": 监控回答格式是否趋同,
    "benchmark":         监控实际任务表现,
}
# 任何一个指标退化 → 立即停止
# 通常 1~2 个 epoch 就足够了
```

9.2 数据飞轮与去重去污染 Pipeline

数据飞轮完整闭环（Data Flywheel）

数据飞轮的核心理念是克服静态数据集的上限（一次性收集 → 训练 → 结束）。它通过"训练产生更好的模型 → 更好的模型生成更好的数据 → 更好的数据训练出更强的模型"这一螺旋上升

的机制，实现模型能力的持续自我强化，大幅降低对持续人工标注的依赖。

完整闭环的六个核心阶段：

Step 1：种子数据（Seed Data）—— 飞轮的第一推动力

- 内容：少量（几千到几万条）由人工精心标注的高质量数据。
- 作用：覆盖目标任务的核心场景与能力（如复杂推理、多轮对话、工具使用），作为飞轮启动的基准。

Step 2：模型采样与生成（Sampling）

- 操作：使用当前最优版本（SOTA）的模型，对大量的 prompt 种子生成候选回答。
- 策略：设置较高的温度（如 diversity > 1.0）以保证生成数据的多样性，包括不同的回答路径、思维链（CoT）和工具调用轨迹。

Step 3：自动过滤与打分（Judge / Filter）—— 飞轮的关键门控

- 打分机制：利用极强的外部模型（如 GPT-4）或内部最强模型作为 Judge，从准确性（Accuracy）、完整性、简洁性、格式规范、安全性等多维度进行打分。
- 规则过滤：结合代码/数学的客观验证器（Verifier），只保留执行正确的结果；剔除格式错误或长度异常的数据。
- 筛选：仅保留 Top 30~50% 的高分数据进入训练集；对于处于边界分数的数据，进行人工抽样复核。

Step 4：再训练（Retrain）

- 操作：使用过滤后的高质量生成数据重新训练模型（执行 SFT → RLHF/GRPO → 对齐流程）。

Step 5：评测与验证（Eval）

- 操作：在私有 Benchmark 上评测新模型，结合人工盲测和线上 A/B 测试。
- 决策：如果能力全面提升，则更新为最优模型；如果出现退化，则触发回滚，并重点检查调整 Step 3 的过滤策略。

Step 6：在线数据采集与循环（Loop）

- 操作：将模型部署上线后，收集真实用户反馈（点赞/点踩偏好信号），提取高质量的真实用户对话作为新的 SFT 数据，更新种子数据集。
- 循环：用更强的新模型回到 Step 2 进行下一轮采样，飞轮转速越来越快，数据更加贴近真实分布。

理论局限与挑战（面试考点）：

- **模型崩溃（Model Collapse）**：如果 Judge/Filter 把关不严，噪声数据和 Judge 本身的偏见（Sycophancy/阿谀奉承）会随着飞轮不断累积放大，导致模型性能雪崩。

- **自举天花板（Bootstrap Ceiling）**：如果完全只用模型自身生成的数据，其能力的理论天花板就是模型自身。要突破天花板，必须引入外部客观验证器（数学/代码）或定期引入新的人工复核数据。

工业级 SFT 数据闭环

把 SFT 看成一个真正可运转的工业闭环，更合理的链路是：数据来源 → 去隐私 / 去敏感信息 → 去重 / 去模板化 → 质量打分 → 配方混合 → train/eval split → 训练与诊断 → checkpoint 选择 → 灰度发布 → 回滚。

为什么要强调"闭环"： 因为 2025-2026 的训练数据越来越多来自线上日志、agent 轨迹、强模型蒸馏和用户反馈；如果没有版本化和回滚，这些新增数据一旦带入偏差，后果会比传统离线语料更快放大。

数据去重与去污染 Pipeline

如果训练数据中包含了评测集的题目（即"污染"），会导致 Benchmark 分数虚高，模型真实泛化能力极差。因此，必须建立严密的去重与去污染流水线。

第一部分：数据去重（Deduplication）

去除训练集内部的冗余数据，提升训练效率。

1. 精确去重： 使用 MD5 哈希，完全相同的数据直接去掉一份。

2. 模糊去重（工业级核心：MinHash + LSH）：

- 原理：将文本转换为 n-gram 集合，通过多个哈希函数映射为固定维度的 MinHash 签名。然后利用局部敏感哈希（LSH）快速找到相似文本（哈希碰撞概率高）。
- 优势：时间复杂度为 $O(n)$，而非直接两两比较的 $O(n^2)$，极其适合亿级大数据的去重。
- 阈值设定：Jaccard 相似度 > 0.8（强匹配，直接删除）；0.5~0.8（疑似匹配，人工复核）。

3. 语义去重： 使用 Embedding 模型计算向量相似度（相似度 > 阈值则删除）。成本较高，但能有效去除"换种说法但语义完全相同"的重复数据。

第二部分：多层去污染（Decontamination）

确保训练集与所有公开/私有评测集（如 MMLU, MATH, HumanEval 等）严格隔离。

层 1：n-gram 精确匹配（最快）

- 对训练集和评测集提取 n-gram。若某条训练数据与测试题的 n-gram 重叠率超过安全阈值，直接删除。

层 2：语义相似度检测（更准）

- 针对 MinHash 和 n-gram 漏掉的案例，利用 Embedding 模型和 FAISS 向量检索库，在向量空间内计算语义相似度（如相似度 > 0.85 则删除），捕捉字面不同但题意相同的题目。

层 3：困惑度异常检测（检测间接污染 / Memorization）

- **Min-K% Prob 方法**：在测试集上运行模型，计算每个 token 的生成概率。重点关注概率最小的 K% 的 token（即本来最难预测的部分）的平均概率。
- **诊断**：如果是正常未见过的数据，这部分 token 的概率应该很低；如果这些极难预测的 token 概率异常偏高（即模型对这道题的整体困惑度 Perplexity 异常低），说明模型极大可能在预训练阶段间接"背诵"过这道题，需标记为污染并从评测中排除。

层 4：时间分割（Temporal Split）

- 物理隔离手段。强制要求训练数据的收集时间戳必须**早于**评测题目的发布时间（例如 LiveCodeBench 等动态榜单的核心防作弊机制）。

层 5：私有评测集（Air-gapped）

- 终极防御：将最核心的评测题库完全隐藏不对外公开，从根本上杜绝爬虫污染。

数据版本与来源治理（Data Governance）

在工业界，数据的管理需要像代码一样严谨。

为什么需要数据治理：

- **Debug 溯源**：当模型在特定领域能力突然崩溃或出现严重幻觉时，能精确定位是哪一批次、哪个处理环节的数据导致了"投毒"。
- **合规与版权**：清晰界定数据的商业可用性，追溯版权来源，规避法律风险。

数据血缘（Dataset Lineage）与版本控制： 必须为每一条进入训练池的数据记录详尽的"身世档案"：

- **来源溯源**：是人工标注、模型合成、网页爬取还是知识蒸馏所得？
- **时间戳**：数据的生成或采集时间。
- **处理流水线**：经历了哪些清洗规则、去重算法和 Judge 打分？
- **应用记录**：最终被用于哪个版本（Checkpoint）的 SFT 或 RLHF 训练？

工程实践工具： 每次启动训练任务前，必须对当前数据集打 Tag，并记录数据内容的 Hash 值。业界常用工具：**DVC (Data Version Control)** 或 **W&B Artifacts**，实现像 Git 一样对 TB 级数据集进行精确的版本控制与回溯。

9.3 多域配比、Mixture Scheduler 与自动化质量评估

多域配比与 Mixture Scheduler

一个更实用的混训视角，不是"数据越多越好"，而是通用对话、行业知识、安全拒答、工具调用、偏好风格、多模态 / 长上下文各自的采样权重如何动态变化。

为什么需要 scheduler：

- 不同阶段的模型短板不同；
- 早期可能更缺格式和助手身份；
- 后期可能更缺工具调用、长上下文、agent 轨迹。

一个常见工业做法： 先按大类定义基础 mixture；再根据 slice eval 的退化情况动态调高对应域的采样权重。

阶段化调度（更像真实工程）：

阶段	名称	数据重心
阶段 1	身份与格式定型	通用对话、system style、拒答范式占比更高
阶段 2	能力补齐	代码、数学、工具、多模态、长上下文按短板补齐
阶段 3	行为校正	安全、偏好、agent trace、复杂指令跟随占比上升

按 slice 动态调权的真实逻辑：

- 如果最近 `tool-use` slice 掉得最快，就调高工具轨迹占比；
- 如果 `refusal` 过强，就降低安全拒答配比并补 benign hard negatives；
- 如果 `reasoning` 变短，就提高长 CoT / verifier 任务配比。

> **一句话**：mixture scheduler 不是静态配方，而是根据 slice 退化信号做闭环调度。

自动化质量评估与 Slice 诊断

自动化质量评估通常至少包含三层：

- **规则层**：长度、格式、schema、拒答合规
- **模型层**：LLM-as-a-judge、分类器、质量分桶
- **多样性层**：n-gram、embedding diversity、难度分层

Slice 诊断为什么重要： 线上退化往往不是"模型整体变差"；更常见的是某一类场景单独崩掉，比如：tool use 退化、多轮对话退化、安全拒答过强、reasoning 变短。

落到实操层的阈值化判断：

- **规则过滤**：非法 schema、乱码、过长、空回答直接剔除；
- **judge 过滤**：低于阈值的样本不进训练集；
- **disagreement 过滤**：多个 judge 分歧太大的样本送人工复核或直接降权；
- **难度分桶**：easy / medium / hard 分开统计，避免只看到平均分。

一个常见误区： 只看总平均质量分。更靠谱的是同时看：总体质量、多样性、难度分布、失败类型占比、重点 slice 的退化曲线。

MAPE 数据飞轮

更工程化的数据飞轮可以按 MAPE 视角来讲：

- **Monitor**：日志、失败率、重查询率、工具失败率、慢思考 token 暴涨
- **Analyze**：用 LLM-as-Judge 做错误归因和错误聚类
- **Plan**：决定补数据、补 reward、补规则还是改策略
- **Execute**：自动合成、过滤、重训、灰度、回滚

两个常用指标口径：

$$\varepsilon_r = \frac{n_r}{N} \quad \text{（错误率）}$$

$$\Delta_L = \frac{L_{\text{baseline}} - L_{\text{tuned}}}{L_{\text{baseline}}} \times 100\% \quad \text{（延迟改善率）}$$

> **一句话**：现代数据飞轮的重点不是"转得快"，而是"每转一圈都知道为什么转、哪里出了问题、能不能回滚"。

9.4 工业前沿视角与高频面试追问

数据工程的关键词从"规模"转向"lineage、freshness、可追责"（2025–2026）

> 🏭 **工业补充**
>
> **为什么数据血缘越来越重要：** 2025-2026 的训练数据很多来自强模型生成、verifier 筛选、线上用户交互、红队回流、agent 轨迹。如果没有 lineage，后面出了问题根本定位不到是哪一层"投毒"。
>
> **freshness（新鲜度）成为新的核心变量：** 尤其在 preference、trajectory、reasoning 数据里，模型越强，旧数据就越快过时。
>
> **工业界的更优做法：** 不再只做"一次性清洗"；而是让数据进入：生成 / 采样 → judge / verifier / rule filter → decontamination → lineage tagging → eval gate → 线上反馈回流，形成真正的数据飞轮。
>
> **一句话**：现在的数据工程，不只是"多搞点数据"，而是"让每一条数据都知道它从哪来、为什么被保留、被谁用过"。

高频追问标准答法

Q：数据飞轮最容易在哪一步翻车？

> 最容易翻车的是 judge / filter 这一层。因为一旦高分机制本身有偏，飞轮会把偏差一轮轮放大。所以真正的难点不是"生成更多数据"，而是"过滤逻辑能不能抗偏差、能不能被审计"。

Q：去污染为什么不能只做字符串去重？

> 因为污染很多时候不是完全重复，而是改写、改序、同义重写甚至跨语言复述。所以工业上一般要组合：n-gram overlap、MinHash / LSH、embedding 相似度、temporal split、private eval。只做精确去重，远远不够。

---

10. LoRA 与参数高效微调

10.1 核心原理、训练过程与工程超参

问题背景

全量微调 70B 模型：需要至少 16 张 A100，训练时间数周，成本数十万人民币。

LoRA 核心原理

关键观察： 微调学到的权重变化 ΔW 是一个**低秩矩阵**。

- **全量微调：** 直接学 ΔW（4096 × 4096 = 1670 万参数）
- **LoRA：** 用两个小矩阵近似大矩阵：$\Delta W \approx A \times B$
  - A 的维度：4096 × r（$r = 8$）
  - B 的维度：r × 4096（$r = 8$）
  - 参数量：4096×8 + 8×4096 = 65536 个
  - 压缩比：1670 万 → 6.5 万，**节省 99.6% 参数**

LoRA 训练过程

```
# 冻结原始权重 W，完全不动
W.requires_grad = False

# 只训练两个小矩阵
A = nn.Parameter(torch.randn(d, r))   # 随机初始化
B = nn.Parameter(torch.zeros(r, d))   # ← 初始化为0！关键！

# 前向传播：
output = x @ (W + A @ B)
```

B=0 初始化的原因（高频追问）

如果 A 和 B 都随机初始化：

```
训练第 0 步，还没学任何东西
A @ B = 充满噪声的随机矩阵
output = x @ (W + 随机噪声)
→ 预训练模型的能力被立即破坏
→ 从一个很差的起点开始收敛，极慢甚至不收敛
```

B = 0 的作用：

```
A @ B = 零矩阵（无论 A 是什么）
output = x @ (W + 0) = x @ W
→ 微调起点 = 预训练模型原始状态
→ 从"已知的好起点"开始训练
→ 训练稳定，收敛快
```

这叫：**稳定性初始化（Stable Initialization）**。

LoRA 工业级超参

```python
lora_config = LoraConfig(
    r=8,                # 秩，通常8~64，越大能力越强但显存越多
    lora_alpha=32,      # 缩放系数，通常是r的2~4倍
    target_modules=[
        "q_proj",       # Attention的Query（必加）
        "v_proj",       # Attention的Value（必加）
        "k_proj",       # Key（可选）
        "o_proj",       # Output（可选）
    ],
    lora_dropout=0.05,
)
```

10.2 QLoRA 与 Multi-LoRA Switch

QLoRA：进一步降低显存

QLoRA = LoRA + 4bit 量化基础模型。

效果：

- 原始 72B 模型：约 144GB（bf16）
- QLoRA 后：约 36GB
- 显存压力直接腰斩

适用场景： 8 张 A100（640GB 总显存）训练 72B 模型——没有 QLoRA 几乎不可能。

Multi-LoRA Switch：一模型服务多任务

问题： 多任务场景下，如果把 LoRA 合并进 W，需要多份完整大模型，显存爆炸。

解法： 保持 W 和 LoRA 分离，根据请求动态切换。

```
class MultiLoRAServer:
    def __init__(self, base_model):
        self.W = base_model          # 只加载一次（140GB）
        self.lora_adapters = {}      # 每个任务一个小adapter（几十MB）

    def register(self, task_name, lora_A, lora_B):
        self.lora_adapters[task_name] = (lora_A, lora_B)

    def forward(self, x, task_name):
        A, B = self.lora_adapters[task_name]  # 动态切换
        return x @ (self.W + A @ B)

# 节省：3×140GB → 140GB + 几百MB
```

什么时候合并（Merge）：

- 单一任务，追求推理速度最快
- 不需要动态切换

什么时候不合并：

- 多租户场景（多任务共享一个基础模型）
- A/B 测试不同版本
- 资源受限环境

10.3 LoRA 变体全家族与 MoE 适配策略

标准 LoRA 虽然奠定了参数高效微调（PEFT）的基础，但在实际工业应用中暴露出了一些固有局限：例如高秩（Rank）状态下梯度不稳定、A 矩阵随机初始化导致起步效率低、以及幅度和方向的耦合优化问题。为此，学界和工业界衍生出了丰富的 LoRA 变体家族。

rsLoRA（Rank-Stabilized LoRA）—— 解决高秩梯度不稳定

标准 LoRA 的痛点： 输出缩放公式为 $\text{output} = \text{base} + (AB)\frac{\alpha}{r}$。当设置更大的秩 r 时，缩放因子 $\frac{\alpha}{r}$ 会线性减小，导致梯度信号减弱。因此，高秩 LoRA 的学习率实际上被变相降低了，高秩往往不一定比低秩效果好。

rsLoRA 的改进： 将缩放因子修改为 $\frac{\alpha}{\sqrt{r}}$。

数学原理： 标准 LoRA 的梯度范数正比于 $1/r$，而 rsLoRA 的梯度范数正比于 $1/\sqrt{r}$。这意味着无论 r 多大，梯度的尺度（Magnitude）都能保持一致。

实践效果： 可以安全地使用更大的 r。在 $r > 16$ 的高秩微调场景下，表现极其稳定，显著优于标准 LoRA。

DoRA（Weight-Decomposed LoRA）—— 幅度与方向分离

核心洞察： 全量微调的本质，等于权重矩阵的**幅度（Magnitude）**变化 + **方向（Direction）**变化。任何权重矩阵 W 都可以分解为 $W = m\frac{W}{\|W\|}$（其中 m 是幅度标量，$\frac{W}{\|W\|}$ 是单位方向向量）。标准 LoRA 同时改变两者，导致学习动态相互干扰。

DoRA 的改进： 分别独立更新幅度和方向。

- 幅度 m：作为可学习的标量直接更新（参数量极少，每个输出维度仅一个）。
- 方向：使用 LoRA 低秩近似进行更新。
- **公式：**

$$W_{\text{new}} = (m + \Delta m)\frac{W_0 + AB}{\|W_0 + AB\|}$$

实践效果： 学习模式极度接近全量微调。在同等参数量下，精度显著高于标准 LoRA，特别适合那些需要大幅改变权重特征"方向"的复杂任务。

PiSSA（Principal Singular Values Adaptation）—— SVD 主成分初始化

标准 LoRA 的痛点： A 矩阵服从高斯随机初始化，B 矩阵零初始化。训练初期增量 $\Delta W = AB = 0$，模型实质上是从随机噪声方向开始"摸着石头过河"，起步效率低。

PiSSA 的改进： 使用奇异值分解（SVD）找到最重要的方向来初始化 A 和 B。

- Step 1：对原始权重做 SVD 分解：$W = U\Sigma V^T$
- Step 2：提取前 r 个最大的奇异值/向量，初始化 LoRA 参数：

$$A = U_{:,\,:r}\sqrt{\Sigma_{:r,\,:r}}, \quad B = \sqrt{\Sigma_{:r,\,:r}}V^T_{:,\,:r}$$

- Step 3：冻结原始权重矩阵的剩余残差部分：$W_{\text{residual}} = W - AB$

优势： LoRA 从模型"最重要的方向"开始学习，避免了冷启动，收敛极快，同等步数和秩下效果更好。

缺陷： 初始化阶段需要做一次完整的 SVD（有一定计算开销），且原始权重被修改，无法直接与标准权重参数对比。

OLoRA（Orthonormal LoRA）—— 正交归一化保证

机制： 在 PiSSA 的 SVD 思想基础上，进一步施加约束，保证 LoRA 的基向量始终是正交归一化的。

优势： 训练过程极致稳定，不同维度的梯度绝对不会互相干扰。非常适合需要精确控制每个 LoRA 独立维度特征的精细化调优场景。

LoftQ（LoRA Fine-tuning with Quantization）—— 量化感知初始化

场景痛点： 当我们在 4-bit 量化模型上做微调时，量化本身会带来不可逆的精度损失。而标准 LoRA 初始 $\Delta W = 0$，在训练初期根本无法弥补这部分量化误差。

LoftQ 的改进： 联合优化模型的量化过程与 LoRA 的初始化。

- 目标函数：$\min \|W - Q(W - AB)\|_F$
- 流程：先量化基础模型得到 W_q，计算量化误差 $E = W_{\text{fp}} - W_q$。然后用 SVD 分解该误差（$E \approx AB$），以此初始化 A 和 B。

实践效果： LoRA 从第一步迭代开始就在"专职"补偿量化误差。最终的微调精度远高于从零起步的普通 QLoRA，效果逼近全精度微调。

补充辨析：QLoRA 与 AWQ/GPTQ 的工业定位

QLoRA（服务于训练期）：

- 机制：基础模型采用 NF4（正态分布友好的 4-bit 格式）量化并冻结，LoRA 适配器保持 FP16 精度进行训练。结合 Double Quantization（对量化常数再量化）进一步压缩显存。
- 核心价值：极大降低显存门槛，让单张消费级小显存显卡微调大模型成为可能。

AWQ / GPTQ（服务于推理期）：

- 机制：纯推理加速优化（不支持训练）。AWQ 基于激活感知保护重要通道；GPTQ 基于 Hessian 矩阵做逐层误差补偿。

工业标准 WorkFlow：

训练侧使用 **QLoRA** 微调 → 训练完成后将 LoRA 权重与 Base 模型合并 → 针对合并后的新模型进行 **AWQ** 量化 → 送入 **vLLM** 进行高性能推理部署。

LoRA 变体工业选型指南

变体名称	核心改进机制	推荐适用场景	相比标准 LoRA 的核心优势
标准 LoRA	A 为 Gaussian，B 为 0	通用基准测试，追求实现简单	基础工业标准，生态兼容性最好
rsLoRA	将缩放系数修改为 $\alpha/\sqrt{r}$	**需要高秩（$r > 16$）** 但训练不稳定的复杂任务	梯度稳定，解锁高秩表达能力
DoRA	权重分解，分别更新幅度和方向	同等参数量下**追求极限精度**，需大幅改变原特征	学习轨迹更接近全量微调（Full-FT）
PiSSA	使用 SVD 初始化 A 与 B	算力受限，**希望加速收敛**，缩短训练时间	起步方向极佳，收敛速度显著提升
LoftQ	联合量化误差与 LoRA 初始化	必须在 **INT4 量化模型上微调**，且对精度要求高	量化精度损失更小，补偿能力强
QLoRA	NF4 基础模型 + FP16 LoRA	**显存资源极度受限**的平民玩家 / 消费级显卡	显存占用最小化

MoE 架构下的 LoRA 适配策略

当基础模型是混合专家模型（Mixture of Experts, MoE）时，给包含数十个 Expert FFN 的网络加 LoRA 会面临参数爆炸或专家微调不均的问题。工业界目前有三种主流策略：

1. Dense LoRA（全局共享 LoRA）

- 机制：给所有的专家外挂**同一个**共享的 LoRA。
- 特点：参数量增加极少。但所有专家都被迫学习相同的增量知识。
- 适用：全局任务适应（如单纯改变输出的格式、语气风格），不涉及深层领域知识的改变。

2. Sparse LoRA（专家私有 LoRA）

- 机制：给模型中的 N 个专家分配 N 个**独立**的 LoRA（即各自挂载）。
- 特点：参数量剧增（$N \times \text{LoRA_params}$）。完美保留了 MoE 专家分工的专业化特性。
- 适用：领域知识注入，需要各专家在各自擅长领域（如数学专家、代码专家）继续深化专业能力的场景。

3. Hybrid LoRA（混合 LoRA）

- 机制：共享 LoRA（学全局通用特征）+ 专家私有 LoRA（学特定专业特征）组合使用。
- 特点：极其灵活，表现上限最高，但系统实现最复杂，容易导致路由权重的学习失衡。

10.4 工业前沿视角与高频面试追问

LoRA 正在从训练技巧变成服务层能力（2025–2026）

> 🏭 **工业补充**
>
> **过去大家谈 LoRA，重点是省显存；现在工业界谈 LoRA，重点越来越是服务能力**：adapter 能否热切换；多 LoRA 是否能共存；是否需要 merge；不同租户/场景能否共享同一基础模型。
>
> **为什么 PiSSA / LoftQ / DoRA 这类变体重要：** LoRA 的瓶颈已经不只是"能不能训"；更是"在量化底座上、在更小 rank 下，还能不能训得稳、训得像样"。
>
> **资深工程师视角：**
>
> - 训练侧看 LoRA，关心的是：初始化质量；rank 与稳定性；quantization compatibility。
> - 服务侧看 LoRA，关心的是：merge 开销；adapter 路由；多任务共享与隔离。
>
> **一句话**：LoRA 在 2025-2026 已经不只是 PEFT，而是 **training + serving 一体化设计问题**。

高频追问标准答法

Q：LoRA 里的 `B=0` 初始化为什么这么关键？

> 因为这样训练一开始 `ΔW = BA = 0`，不会破坏原模型输出。这让优化从一个"已知可用的预训练点"开始，而不是从一个随机扰动后的坏点开始。对大模型来说，这个稳定起点非常重要。

Q：什么时候应该 merge LoRA，什么时候不 merge？

> 单任务上线、追求最低延迟时，merge 更合适。多任务共底座、频繁热切换、A/B 测试时，不 merge 更灵活。所以 merge 不是纯训练问题，更是服务架构问题。

---

11. Agent 训练全链路

11.1 Agent RL 基础：奖励设计与可验证性分类

普通 SFT 与 Agent RL 的本质区别

普通 SFT 教工具调用——只给模型看 `<tool_call>` 的格式示例，模型学会"格式上怎么调用"，但不知道：调用时机、工具选择、参数准确性、结果如何利用。

Agent RL 教的正是这四件事，通过与真实环境交互产生奖励信号。

Agent 奖励的维度总览

对话路由：先判断是否进入 Agent 流程

- 先判断是否需要工具调用
- 不需要 → 直接回答（无需 RM，直接用 DPO/SFT）
- 需要 → 进入 Agent 流程

Agent 流程奖励维度：

- Planning 质量 → 子目标完成率
- Tool Use 准确性 → 工具调用准确率
- 效率奖励 → 工具调用次数少但答案正确 → 额外奖励
- Memory 使用 → 相关记忆是否被正确召回
- 最终结果 → ORM 打分

Tool-use RL 四层 Reward：

```python
def agent_reward(trajectory):
    reward = 0.0

    # 层次1：Plan 质量（Planning Reward）
    plan_quality = evaluate_plan(trajectory.plan)
    reward += 0.1 * plan_quality   # 权重小，避免只会规划不会执行

    # 层次2：工具调用过程奖励（Tool-use Reward）
    for step in trajectory.steps:
        if step.tool_call_success:    reward += 0.2  # 成功调用
        if step.got_valid_result:     reward += 0.1  # 拿到有效结果
        if step.correct_tool_chosen:  reward += 0.1  # 选对了工具

    # 层次3：推理质量（Reasoning Reward）
    reasoning_score = evaluate_reasoning(trajectory.reasoning_steps)
    reward += 0.2 * reasoning_score

    # 层次4：最终结果（Outcome Reward，权重最大）
    outcome = verify_final_state(trajectory.final_state)
    reward += 1.0 * outcome

    return reward
```

工具调用的可验证性分类

完全可验证（用 Outcome Reward 就够）：

- 数学计算工具 → 结果对不对，客观
- 代码执行工具 → 测试通过/失败，客观

部分可验证（需要 Tool-use Reward + 轻量 RM）：

- 天气 API → 能验证调用成功，但判断是否下雨需要推理
- 日历修改 → 能验证修改成功，但"改得对不对"需要判断

完全不可验证（完全依赖 RM，有 Goodhart 风险）：

- 写一封措辞得体的邮件
- 给出合适的建议

11.2 Credit Assignment、Planning RL 与效率奖励

Credit Assignment：Agent RL 的核心难题

问题定义：

```
任务：帮我订一张明天去上海的机票

Step1：搜索航班          ← 这步是否导致了失败？
Step2：筛选最优          ← 这步是否导致了失败？
Step3：填写乘客信息      ← 这步是否导致了失败？
Step4：支付              ← 这步是否导致了失败？
最终：失败

问题：无法判断是哪一步的决策导致了最终失败
```

纯 Outcome Reward 的局限： 不知道哪步出错 → 梯度信号稀疏。

Reward Shaping 的副作用： 每步给固定小奖励 → 模型学会"拆得越碎越好"——步骤越多拿越多奖励，但结果不一定完成。

工业界常见解法：

方案 A：稀疏奖励 + PRM —— ORM 给最终结果评分，PRM 对中间步骤评分，两个信号加权融合。

方案 B：子任务分解 —— 把长任务切成可验证的小任务，每个小任务单独给奖励，把稀疏奖励变成密集奖励。

方案 C：LLM-as-Judge —— 用更强的模型对每一步打分，适用于开放式任务，但有被骗风险（Reward Hacking）。

实际方案：三种混合使用。

蒙特卡洛解法（和 PRM 标注同构）：

```
从 Step k 完成后，rollout N 次：
score(Step k) = 任务最终完成的比例

step_value = score(Step k) - score(Step k-1)
→ 只有真正推进任务的步骤才有正 advantage
→ 无法靠"拆步骤"刷奖励
→ 自动对齐过程奖励和结果奖励
```

Planning RL：如何量化 Plan 质量

Plan 质量的量化方式：

方案 A：LLM-as-Judge —— 用强模型评估 Plan 合理性。灵活，但有被 hack 的风险，成本高。

方案 B：Plan 执行成功率（推荐） —— 不直接评估 Plan 质量，而是看"按这个 Plan 执行的成功率"——成功率高 = Plan 质量高。把 Plan 质量和 Outcome 客观挂钩。

方案 C：子任务完成率（Dense Reward） —— 复杂任务分解成子任务，每完成一个子任务给小奖励，把稀疏奖励变成密集奖励。

工业界：方案 B + 方案 C 混合。

Plan 最终不该只看"写得像不像计划"，而应尽量和"是否真的帮助完成任务"绑定。

OTC-GRPO 与效率奖励

为什么只奖励"任务完成"不够： 普通 Agent RL 只奖励"任务完成"，但模型可能用了很多不必要的工具调用才完成任务 → 效率差，成本高。

OTC-GRPO（Optimal Tool Call GRPO）核心思想： 不只奖励完成任务，还惩罚不必要的工具调用。

奖励函数：

$$r = r_{\text{outcome}} - \alpha \cdot \max(0, n_{\text{calls}} - n_{\text{optimal}})$$

- r_{outcome}：任务完成的奖励（0 或 1）
- n_{calls}：实际工具调用次数
- n_{optimal}：完成这个任务的最优调用次数
- α：效率惩罚系数

n_{optimal} 怎么得到：

- 方法 A：人工定义最优路径（成本高）
- 方法 B：用最强模型（GPT-4）生成最优解，作为参考
- 方法 C：在多次成功轨迹中，取调用次数最少的那次

效果： 模型学会了"用最少的工具调用完成任务"，减少不必要的工具调用 30~50%，降低 token 成本。

11.3 轨迹级优化：Trajectory-level RL / DPO

在传统微调中，我们通常只关注单步"一问一答"。但对 Agent 而言，成败在于**一整条执行轨迹（Trajectory）**的规划与演进。Agent 的对齐必须从单步跨越到**轨迹级（Trajectory-level）**。

Trajectory-level SFT 与掩码规则

数据格式：从对话对到完整任务流

```
[TASK] 查明天上海天气并更新日历
[THINK] 我需要先查天气API，再访问日历
[TOOL_CALL] {"tool": "weather_api", "params": {"city": "上海", "date":
"tomorrow"}}
[TOOL_RESULT] {"weather": "晴天", "temp": "15-22°C"}
[THINK] 天气晴，保留室外会议，取消室内计划
[TOOL_CALL] {"tool": "calendar", "action": "update", ...}
[TOOL_RESULT] {"status": "success"}
[ANSWER] 已为您查询天气并更新日历室外会议。
```

Labels Masking 规则（极其重要）：

计算 Loss（模型需要学如何生成的）：

- `[THINK]` 环节：让模型学会如何做正确的推理规划。
- `[TOOL_CALL]` 环节：让模型学会严格按照 JSON Schema 生成正确的工具参数。
- `[ANSWER]` 环节：最终用户回复。

屏蔽 Loss（Mask 掉的，模型不需要学的）：

- `[TOOL_RESULT]` 环节：这是外部 API 或环境返回的真实数据，是"客观存在的设定"，模型不需要去学习预测 API 返回的值是什么。

Trajectory-level DPO：数据构造与目标函数

数据构造的对比维度：

- **普通 DPO**：对比的是同一问题下的单步回答（Chosen: "好回答" vs Rejected: "坏回答"）。
- **轨迹级 DPO**：对比的是完整任务流。
  - **Chosen（成功轨迹）**：`[user_request] → [tool_call_1] → [result_1] → [tool_call_2] → [result_2] → [final_answer_success]`
  - **Rejected（失败轨迹）**：`[user_request] → [wrong_tool_call] → [error] → [retry] → [failed_answer]`

目标函数：

$$\mathcal{L}_{\text{TrajDPO}} = -\log \sigma\Big(\beta \cdot \big(\log P(\text{success_traj}) - \log P(\text{fail_traj})\big)\Big)$$

这里计算的概率 P 是模型生成整条轨迹中所有可学习 Token 概率的累乘。

对比方向设计：

- **Sequence-level（序列级对比）**：直接拿赢了的整条轨迹打败输了的整条轨迹。优点是实现简单，只需关注最终结果；不需要对中间步骤单独人工标注（成本远低于 PRM，且信号比仅打分最终答案的 ORM 更丰富）。
- **Step-level（步骤级对比）**：类似于 PRM 思路，精细对比 W 轨迹和 L 轨迹在第 k 步的具体工具调用差异。

核心工程挑战：

- **轨迹长度极度不一**：可能赢的轨迹只有 5 步，输的轨迹陷入死循环长达 50 步，导致概率乘积差异极大。必须引入 **Per-sample Loss Normalization**（基于长度的归一化）。
- **环境的随机性**：相同的工具调用，昨天成功了，今天可能因为网络超时报错。因此训练时常常需要在模拟沙箱环境中重放（Replay）轨迹。

OTC-GRPO 与分层复合奖励（Hierarchical Reward）

当使用 GRPO 训练 Agent 时，如果仅仅设定"任务完成就给 +1 分"，模型会发现通过无脑疯狂调用各种工具"撞大运"也能偶尔完成任务——导致模型学得极其啰嗦、效率极低。

OTC（Optimal Tool Call）惩罚机制：

$$\text{reward} = \text{task_success_reward} - \lambda \times \text{extra_tool_calls}$$

- $\text{extra_tool_calls} = \max(0, \text{actual_calls} - \text{optimal_calls})$
- λ 是效率惩罚系数（通常在 $0.05 \sim 0.2$ 之间）

分层复合奖励体系（Hierarchical Reward）：

- **层次 1：格式依从（Format Compliance，最底层）** —— 模型输出的工具参数 JSON 是否合法？不合法直接归 0。（R_{format}）
- **层次 2：执行连通（Execution Success）** —— 工具调用发送给 API 后，是否成功执行并返回了有效结果？（R_{exec}）

- **层次 3：任务正确性（Correctness，最重要核心）** —— 最终交还给用户的答案是否真的解决了问题？权重最高。（R_{correct}）
- **层次 4：效率与安全惩罚（Efficiency & Safety Constraint）** —— 是否使用了多余步骤？是否调用了危险操作？（$R_{\text{efficiency}}, R_{\text{safety}}$）

总奖励公式：

$$R = w_1 R_{\text{format}} + w_2 R_{\text{exec}} + w_3 R_{\text{correct}} - w_4 R_{\text{efficiency}} - w_5 R_{\text{safety}}$$

（权重通常设定为 $w_3 \gg w_1, w_2, w_4, w_5$，即确保任务成功是绝对的第一优先级，格式、效率和安全则是辅助约束条件）

11.4 MCP：工具调用统一接口标准

为什么需要 MCP

问题背景： Agent 需要调用各种工具，每种接口格式不同——天气 API 用 REST API，日历工具用 Google Calendar API，数据库用 SQL 查询。模型需要为每种工具单独学习调用格式，新增一个工具就要重新训练 → 不可扩展。

MCP（Model Context Protocol，Anthropic 2024）的核心定位： 给所有工具定义统一的接口格式，模型只需要学会一种调用方式，新工具实现 MCP 接口 → 模型自动会用。

> **类比：MCP 是工具调用的 USB 标准。**

对 Tool-use RL 的影响：

- 没有 MCP：100 个工具需要 100 种格式的训练数据
- 有了 MCP：Reward 设计统一，新工具零额外训练成本

MCP 底层机制与完整调用流程

架构层次：

- MCP Host（主机）：Claude Desktop / 你的 APP
- MCP Client：嵌入在 Host 里，管理连接
- MCP Server：提供工具的服务（天气 API/日历/数据库）

通信协议： 基于 JSON-RPC 2.0；传输层：stdio（本地）或 HTTP+SSE（远程）

完整调用流程（五步）：

Step 1：Server 注册工具

```
{
  "name": "get_weather",
```

```
    "description": "获取指定城市的天气",
    "inputSchema": {
      "type": "object",
      "properties": {
        "city": {"type": "string"},
        "date": {"type": "string"}
      }
    }
  }
```

Step 2：模型生成工具调用

```
<tool_call>
{"name": "get_weather", "input": {"city": "上海", "date": "2026-04-11"}}
</tool_call>
```

Step 3：Client 执行调用 —— MCP Client 解析 JSON，向对应的 MCP Server 发送 `tools/call` 请求。

Step 4：Server 返回结果 —— `{"result": {"temperature": "22°C", "condition": "晴天"}}`

Step 5：结果注入上下文 —— Client 把结果以 `tool_result` 形式注入对话，模型继续生成。

MCP 对 Agent 训练的意义： 统一接口 → 训练数据格式统一；新工具零额外训练 → 数据效率高；工具描述是自然语言 → 模型可以泛化到新工具。

MCP 协议更新与底层机制辨析（截至 2026-04）

2025-03-26 revision： MCP 引入了基于 **OAuth 2.1** 的授权框架，并把旧的 **HTTP+SSE** 传输替换为更通用的 **Streamable HTTP**；同时补充了 tool annotations 等更细粒度的工具描述能力。

2025-06-18 revision： MCP 将 server 明确为 **OAuth Resource Server**，并要求 HTTP 请求中带上协商后的 `MCP-Protocol-Version`；还增加了 **structured tool output** 等对生产落地更关键的能力。

2025-11-25 revision： 继续增强 **OIDC discovery**、增量 scope 同意、默认 **JSON Schema 2020-12** 方言，以及把 `tools` / `toolChoice` 扩展到 sampling 侧，说明协议已经从"工具注册标准"进化为 **Agent I/O 与授权治理标准**。

> **面试表述建议：** 不要只把 MCP 说成"统一 function calling 格式"。更准确的说法是：**MCP 标准化了工具、资源、提示、采样、鉴权与传输，让 Agent 能力从模型私有接口转向跨平台协议接口。**

MCP 权限、鉴权与资源边界： MCP 真正进入生产后，核心问题不只是"能不能调工具"，而是：谁有权限调、能访问哪些 resource、tool scope 是什么、structured output 是否满足协议版本。常见约束：OAuth / scope、resource boundary、protocol versioning。

传统 Function Calling vs MCP 底层机制辨析：

- **传统 Function Calling（OpenAI 风格）**：本质上大模型还是在"吐 Token"，只是通过 SFT 学会了碰巧将生成的 Token 排列组合成了特定的结构化 JSON。各个平台的 Schema 定义各不相同，换个平台就要重新适配。
- **MCP（Model Context Protocol）**：它是协议层的标准化，与平台无关。MCP 将所有工具的定义抽象为了统一的 Schema 格式。对训练的巨大影响：有了 MCP 标准，所有工具调用的 SFT 数据都可以采用统一格式构造，模型只需要学会一种固定的调用范式，未来即便向 MCP 注册了全新工具，模型无需二次微调即可"零样本"自动可用。

Tool Schema / Tool Trace / Tool Result Replay

- **Tool Schema**：定义工具名、参数类型、必填项、返回结构和错误码。
- **Tool Trace**：记录一次 agent 任务里的 user request → reasoning → tool call → tool result → final answer。
- **为什么 replay 重要**：线上 agent 问题不靠日志文本很难定位；必须能回放整个 trace，才能知道是 plan 错了、参数错了还是结果没用好。

Sandbox / Tool Safety / Tool Result Prompt Injection 防护

tool safety 的重点在于防止模型把错误变成真实外部动作。常见防线：

- sandbox、timeout、retry policy
- allowlist / denylist
- destructive action 二次确认
- 审计日志与可回放 trace

tool result prompt injection 防护： tool result 不是天然可信输入；尤其是网页检索结果、用户上传文档、搜索摘要和外部 API 文本，都可能把恶意提示重新喂回模型。

更稳妥的处理方式：

- schema filtering
- field allowlist
- 明确 trust boundary
- 对网页/搜索结果做 sanitization
- 在 prompt 中把 tool result 标成"外部证据"，而不是"系统真理"

> **一句话**：prompt injection 不只会从 user prompt 进来，也会从 tool result 反向渗透进来。

11.5 Memory RL、多 Agent 训练与持续学习

Memory RL：长期记忆的奖励设计

Memory RL 的四维 Reward 体系：

维度 1：写入时机（Write Timing）

- Reward = 写入的记忆被后续任务用到 → +0.5
- Penalty = 没写但后来需要重新查询 → -0.3（重复劳动）
- Penalty = 写了但从没被用到 → -0.1（噪声记忆）

维度 2：写入质量（Write Quality）

- 原始信息 → 写入记忆 → 用记忆完成新任务
- 压缩后任务成功率 / 原始信息直接用的成功率，比值越接近 1，写入质量越高。

维度 3：写后效果（Retrieval Effectiveness，最可验证）

```
写入前：完成同类任务的成功率 = baseline
写入后：完成同类任务的成功率 = new_rate
memory_reward = new_rate - baseline
→ 和 PRM 的蒙特卡洛估计逻辑相同
```

维度 4：生命周期管理（Compression & Forgetting）

- 压缩：压缩前后任务成功率不变 → 压缩成功，给奖励；成功率下降 → 压缩损失了关键信息，惩罚。
- 遗忘：删除后不影响成功率 → 正确遗忘；删除后成功率下降 → 删错了，惩罚。

Memory RL 的训练信号时序问题： "写入一条记忆"的价值，要等到"用到这条记忆"才能评估 → 奖励天然延迟 → 需要更长的训练 episode → 比 Tool-use RL 训练成本高 5~10 倍。

Memory RL 不只是"会不会存"，更重要的是：该不该存、存得好不好、后面能不能检索出来、压缩/遗忘是否正确。

个性化记忆：参数记忆 vs 检索记忆

- **参数记忆**：记在模型参数里，调用便宜，但难改、难删、难审计。
- **检索记忆**：记在外部 memory store，易更新、易删除、易追踪，但调用要依赖检索。

个性化系统里，长期记忆更适合作为可检索外部状态，而不是硬塞进参数。

用户画像、隐私、误记忆回滚（个性化记忆上线后必须考虑）：去标识化、写入策略、删除策略、误记忆纠偏、用户请求清除后的可追踪删除。

记忆评测与删除策略（至少应监控）： 记忆命中率、错误召回率、误记忆率、用户修正后的恢复速度。删除策略：时间淘汰、置信度淘汰、用户显式删除优先、冲突信息覆盖时保留版本历史。

多 Agent RL 与 Non-stationarity

Non-stationarity（非平稳性）问题：

```
同时训练 Leader 和 SubAgent：

第 1 轮：Leader 策略 L1 + SubAgent 策略 S1，配合好
第 100 轮：SubAgent 更新成 S100
      但 Leader 的 L1 是基于 S1 设计的
      L1 + S100 可能完全不兼容
      → 系统性能骤降

本质：对 Leader 来说，SubAgent 是"环境"
      SubAgent 在训练 → 环境在变化
      RL 假设环境平稳，被破坏
      → 训练不稳定，甚至发散
```

三种工业解法：

解法 A：Alternating Training（轮流训练）

- Phase 1：冻结 SubAgent，只训练 Leader（10 步）
- Phase 2：冻结 Leader，只训练 SubAgent（10 步）
- 交替进行，每个阶段对方固定 → 环境平稳
- 代价：收敛慢

解法 B：CTDE（集中训练，分散执行）

- 训练时：Leader 能看到所有 SubAgent 内部状态（全局信息）
- 推理时：每个 Agent 只用自己的局部信息
- 代表算法：MAPPO、QMIX
- 优点：训练信号准确 + 推理时不依赖全局通信

解法 C：分层训练（工程最简单，最接近实际架构）

- Leader 做 RL（Tool-use / Planning / Memory RL）
- SubAgent 做 SFT（用 Leader 生成的好指令训练）
- 两者不同时做 RL → 避免非平稳
- 字节豆包/扣子最可能用的方案

只训练 Leader 的局限： 优势是环境平稳、训练稳定、工程简单；局限是系统天花板 = SubAgent 的能力上限，Leader 再聪明也无法弥补 SubAgent 的能力缺陷。

实际最优工程方案： 分层训练（解法 C）—— Leader RL + SubAgent SFT + 轮流迭代。

Continual RL 与灾难性遗忘

灾难性遗忘的本质：

```
传统 RL 的遗忘路径：
  学日历技能 → 能力存在参数 W 里
  大量训练邮件任务 → W 被更新
  → 日历技能权重被覆盖 → 遗忘

持续运行 Agent 的特殊挑战：
  没有明确的 episode 边界
  任务和任务之间连续发生
  新任务训练 → 旧任务能力被覆盖
```

工业解法（按性价比排序）：

解法 A：能力外部化（Modular RL）★★★★★

- 技能封装成独立 Skill 模块，主模型只学"路由"（什么时候调哪个 skill）
- 新技能直接加文件，不动参数
- → 技能不在参数里，永远不会被遗忘

解法 B：Memory 外部化（RAG-based Memory）★★★★★

- 长期知识存文件，按需检索；参数只存"怎么检索和利用"
- → 永久不遗忘，因为根本不在参数里

解法 C：EWC（Elastic Weight Consolidation）★★★

- 训练新任务时，保护旧任务重要的参数
- 计算"参数重要性矩阵"，梯度更新时给重要参数加权
- 代价：随任务增多，计算开销线性增长

解法 D：Progressive Neural Networks ★★

- 每个新任务新增一列神经网络，旧列冻结
- 代价：模型越来越大，工程不可持续（学术研究用）

工业实践结论： 能力外部化（A）+ Memory 外部化（B）通常是性价比最高的组合；如果还担心持续训练损伤"何时调用"的判断能力，再额外用 **EWC** 保护少量路由相关参数即可。

> **连接 OpenClaw 的面试话术：** "灾难性遗忘的根本原因是把所有能力压缩进参数。我的解法是能力外部化：技能封装成独立 Skill 文件，知识存入外部 Memory，参数只负责路由和检索决策。这就是 OpenClaw 的架构——Skills 文件系统 + MEMORY.md 是天然的 Modular RL 实现。需要防遗忘的只剩下'如何调用'这一层，用 EWC 保护相关参数就够了，大幅降低了 Continual RL 的难度。"

11.6 全链路统一视角与高级补充话题

Agent 全链路训练如何拼起来

入口：先做对话路由 —— 先判断是否需要工具调用：不需要 → 直接回答（DPO / SFT）；需要 → 进入 Agent 流程。

Agent 流程内部的核心能力：

能力模块	本质问题	常见训练信号
Planning	子目标怎么拆、执行顺序怎么排	plan 完成率、子任务成功率、回溯代价
Tool Use	何时调工具、调哪个工具、参数怎么填、结果怎么消费	调用成功率、工具选择准确率、结果利用率
Efficiency	在任务完成前提下，如何减少无效步骤	OTC-GRPO、步骤数惩罚、冗余调用惩罚
Memory	何时写入、写什么、何时检索、如何压缩与遗忘	写入收益、召回命中率、后续任务增益
Outcome	最终任务有没有完成	ORM / Verifier / 最终成功率

Agent 不是"会不会调用工具"这么简单；它本质上是一条从**规划 → 执行 → 记忆 → 结果**的长链决策过程。

训练信号的五层组合方式：

1. **稀疏结果奖励** —— 最终任务是否完成，是最稳定的总锚点。
2. **过程奖励** —— 包括 PRM、子任务奖励、工具调用奖励、Memory 奖励，用来缓解 credit assignment。
3. **偏好优化** —— 典型形式是 **Trajectory-level DPO**，用整条成功轨迹压过失败轨迹。
4. **协议与接口层** —— **MCP** 让工具接口标准化，从而降低新增工具的训练与数据构造成本。
5. **长期稳定性机制** —— Multi-Agent 用分层训练缓解非平稳；Continual RL 用 Skill / Memory 外部化缓解灾难性遗忘。

工业化落地完整组合拳：

- **SFT / DPO** —— 负责基础格式、语言能力和简单路由。
- **Outcome RL** —— 负责最终任务完成率。
- **PRM / 子任务奖励 / Monte Carlo credit assignment** —— 负责把稀疏结果奖励拆回中间步骤。
- **OTC-GRPO** —— 负责效率优化，减少冗余工具调用。
- **Trajectory-level DPO** —— 负责整条轨迹层面的偏好学习。
- **MCP** —— 负责统一工具接口，提升泛化与扩展性。
- **Memory RL** —— 负责长期个性化和跨任务收益。
- **分层多 Agent 训练** —— 负责系统级稳定性。
- **Modular Skill + External Memory + EWC** —— 负责 Continual RL 下的抗遗忘。

一句话总括

Agent RL 的本质，不是教模型"会不会生成一个工具调用 JSON"，而是让模型在真实环境反馈下，学会：什么时候调用；调哪个工具；参数怎么填；结果怎么利用；计划如何制定；记忆何时写入、何时检索；如何高效完成整条任务轨迹；以及如何在长期持续学习中不遗忘旧能力。

GUI / Computer Use / UI-TARS / A2A

GUI / Computer Use 场景的特殊点： 动作空间比纯文本大得多；页面状态是隐式环境；错误动作会产生真实副作用。

常见动作空间： click / type / scroll / drag / hotkey / select / wait。

GUI / Computer Use 完整任务流：

1. 感知当前 screen / DOM / accessibility tree
2. 规划下一步动作
3. 生成 action schema
4. 执行动作
5. 读取新状态
6. 判断是否继续、回退或终止

UI-TARS / Computer Use 这类系统真正难的地方： 不是动作种类够不够多，而是：状态观测是否完整、动作是否可逆、错误点击是否会带来高成本副作用、长轨迹里怎么做恢复和回退。

A2A（Agent-to-Agent）与 MCP 的互补关系：

- **MCP** 更像"模型与工具/资源"的统一协议（I/O 协议）；
- **A2A** 更像"agent 与 agent"之间的能力转交和任务协作协议（协作协议）；
- 两者不是替代关系，而是分别解决"调工具"和"多 agent 协作"。

截至 2026-04 可公开确认的信号： `A2A` 已经由 Google 发起开源协议，目标是让不同框架、不同服务器上的 agent 彼此协作；这意味着 agentic system 的协议层正在分化成两类：`MCP`（模型与工具/资源）、`A2A`（agent 与 agent）。

Agent Eval：Task Success、Trace Quality、Safety

agent eval 至少要看三层： task success、trace quality、safety / policy compliance。

为什么只看最终成功率不够： 有的轨迹虽然成功，但完全不可复用、不可审计、成本极高；这类轨迹不适合生产。

更贴近 real-world agent eval 的专项维度： Task Success、Trace Quality、Cost / Step Count、Recovery Ability、Safety。

常见 benchmark / eval 参照系： AgentBench、SWE-Bench、τ^2-Bench、各团队私有 office / browser / search workflow 任务集。

Search / Deep Research Agent：从检索到多文档综合

Deep Research / Search Agent 的完整链路：

1. 识别用户意图与任务类型
2. 拆成检索子问题
3. 多源检索与重排
4. 跨文档证据对齐
5. verifier / judge 过滤低可信结论
6. 结构化总结与引用归并

为什么它比普通 RAG 难： 它不是"检索后读一遍"；而是要在多轮搜索中不断更新计划，并处理：证据冲突、来源可信度、多跳综合、引用一致性。

一个 deep research agent 往往还要补三层：

- **source planning**：先决定先查哪类来源，什么时候停止继续搜；
- **evidence bookkeeping**：每条结论绑定来源、时间戳、可信度；
- **final synthesis**：不是把检索结果拼起来，而是做跨文档归纳、冲突消解和引用对齐。

Qwen-Agent 给的启发： 长上下文承载证据，工具调用补充信息，计划能力控制检索深度与终止条件——三者不是独立能力，而是有机整体。

> **一句话**：search agent 的核心不是"会搜"，而是"会在多文档、多轮次检索中持续修正计划并合成结论"。

11.7 工业前沿视角与高频面试追问

MCP Revision 之后，Agent 训练更接近真实生产协议（2025–2026）

> 🏭 **工业补充**
>
> **MCP 的意义已经超过"统一工具调用格式"**，它现在同时覆盖：tools、resources、prompts、sampling、auth / authorization、transport。
>
> **为什么这对 Agent 训练很关键：** 训练数据不再只是假想的 function call；而是越来越接近真实生产接口的约束条件：schema 是否合法、是否有权限、资源能否访问、sampling / tool choice 是否满足协议。
>
> **资深工程师视角：** Agent 能力越来越不像"自然语言技巧"，更像"在协议约束下完成规划、调用和状态转移"的系统行为。
>
> **一句话**：2025-2026 的 Agent 训练，正在从"教模型像代理一样说话"，转向"教模型像代理一样遵守真实生产协议"。

高频追问标准答法

Q：Tool-use RL 最难的地方是什么？

> 最难的不是生成一个工具调用格式，而是 credit assignment。最终任务失败时，到底是 plan 错了、工具选错了、参数填错了，还是结果没利用好，这些都要拆清楚。所以 Agent RL 的核心难点是长链决策归因，而不是 JSON 生成。

Q：MCP 为什么会提升 Agent 训练价值？

> 因为它把工具接口从"平台私有格式"变成了"跨平台协议约束"。这样训练出来的能力更有迁移性，新增工具也更容易复用原有策略。本质上，MCP 降低了 agent data construction 和 tool generalization 的成本。

---

12. Test-Time Compute 与搜索

12.1 范式转变：从训练时算力到推理时算力

新旧 Scaling Law 对比

范式	核心资源	典型结论
旧 Scaling Law（Kaplan 2020）	模型参数量、训练数据量	参数越大、数据越多，性能通常越强

范式	核心资源	典型结论
新 Scaling Law（Snell 2024 之后的 reasoning 视角）	推理时 compute、思考 token、搜索深度	同等训练成本下，给模型更多推理算力，可能比单纯放大参数更划算

旧范式： 把尽可能多的智慧"烧进参数"；推理时一次前向传播，越快越好。

新范式： 推理时允许模型"多想一会儿"；用额外的 test-time compute 去换答案质量和可验证正确率。

推理时长控制：三种方案与工业最优解

方案	描述	问题
A：固定预算	所有题 token 数相同	简单题水字数，难题被截断
B：模型自决	模型自己决定何时停	Overthinking Problem，表演性思考
C：外部系统	根据难度动态分配	需要额外难度判断模块

工业界最优：B + C 组合

- C 负责粗粒度控制（设定 token 预算上限）
- B 负责细粒度控制（在预算内自行决定何时输出 `</think>`）
- 这正是 Qwen3 和 Claude 3.7 "思考模式开关"背后的逻辑

Overthinking Problem

模型倾向于：明明 200 token 能解决的题，非要绕弯路想 2000 token。

原因： 训练时"更长的思考链"往往对应"更高的奖励"，模型学会了"表演思考"而不是"高效思考"。

12.2 MCTS + LLM：推理时搜索

为什么需要搜索?

传统推理： 问题 → 线性思考链 → 答案（一条路走到底）

MCTS 新范式： 问题 → 展开多条思考路径（树）→ PRM 评估 → 剪枝深挖 → 答案

核心挑战：搜索空间爆炸

LLM 上做 MCTS 面临的根本困难：

```
AlphaGo MCTS：
  每个节点 = 棋盘状态
  分支数   = 合法落子（约 200）

LLM MCTS：
  每个节点 = 当前思考链状态
  分支数   = 下一个 token 的可能数（约 100,000）

分支数是棋盘的 500 倍 → 直接在 token 级做 MCTS 根本搜不完
```

工业界解法：抬高搜索粒度到步骤

```
❌ token 级 MCTS（不可行）：
  分支数 100,000，深度 2000 token

✅ 步骤级 MCTS（可行）：
  节点   = 每一个完整的推理步骤（"设方程 x+y=10"）
  分支数 = 每步采样 4~8 个候选步骤
  深度   = 10~20 步

搜索空间：100,000^2000 → 8^20，压缩天文数字级别
```

PRM 在 MCTS 中的角色

MCTS 需要两个函数：

- **Policy（策略）：下一步往哪走？** → LLM 本身，生成候选推理步骤
- **Value（价值）：这条路有多有希望？** → PRM！对当前推理链打分 → 分数高的节点优先展开 → 分数低的节点剪枝（早停）

这正是 AlphaProof / o3 被猜测使用的核心机制。

12.3 Self-Verification 与 Scalable Oversight

核心矛盾

当模型能力超越人类后：

- 人类无法判断"这步推理是否正确"
- 模型自己验证自己 → 用有问题的推理验证推理
- → 鸡生蛋问题

解法层级

层级	方法	局限
Level 1	人类直接判断	能力受限，已不够用
Level 2	更强模型辅助	递归问题，强模型谁来验证
Level 3	形式化验证	仅适用于数学，AlphaProof 路线
Level 4	Debate 机制	弱裁判 + 强对抗，放大监督能力
Level 5	Self-Play	纯对抗涌现，不需要外部真理

Debate 机制详解

两个模型就同一问题给出相反论点，让第三方弱裁判判断谁更有说服力。

关键洞察：

- 裁判不需要比辩手更聪明
- 只需要判断"哪方论证更自洽、更难被反驳"
- 辩手为了赢，必须主动暴露对方漏洞
- 对抗压力倒逼推理质量提升

这叫 **Amplification（放大）**：弱监督者 + 强对抗机制 = 超越强监督者的监督质量。

12.4 PRM 训练数据构造

两种方案对比

方案	描述	优点	缺点
LLM-as-Judge	强模型评判每个推理步骤	适用任意任务	成本极高，递归问题
Monte Carlo Rollout	从该步骤采样 N 条后续路径，统计正确率	自动化，无需强模型	仅适用可验证任务

蒙特卡洛估计原理

```
score(Step k) = 从 Step k 继续采样 N 条路径中，最终答案正确的比例

例：Step 3 之后采样 10 条路径
  → 8 条最终答案正确
```

```
→ score(Step 3) = 0.8（此步大概率正确）
→ score(Step 3) = 0.1（此步大概率出错）
```

代表工作：**Math-Shepherd** 论文。

系统性偏差问题与解法

问题： 同一模型采样的 N 条路径，都受同一偏差影响 → 评分系统性失真。

解法：多模型集成采样

- K 个不同模型各采样若干条路径
- 各模型的偏差方向不同
- → 在集成中相互抵消
- → 评分更稳健

12.5 工业前沿视角与高频面试追问

Test-Time Compute Controller 正在从固定预算走向自适应预算（2025–2026）

> 🏭 **工业补充**
>
> **固定预算的问题：** 简单题浪费 token；难题又常常被截断。
>
> **工业界越来越倾向于引入 controller：** 根据题目难度、模型不确定性、局部 verifier 信号、历史成功率动态分配预算。
>
> **典型可组合机制：** self-consistency / majority vote；early stopping；verifier-gated continuation；search depth 自适应；think-mode / non-think-mode 切换。
>
> **资深工程师视角：** TTC 真正难的不是"让模型想久一点"；而是让预算分配和收益近似匹配，也就是：哪些问题值得继续思考；哪些问题应该立刻停。
>
> **一句话**：Test-time compute 的终局不是"无限思考"，而是"按收益分配思考"。

高频追问标准答法

Q：为什么 Test-Time Compute 对数学和代码特别有效？

> 因为这些任务往往存在清晰的中间搜索空间和可验证终点。多给一点推理预算，模型就有机会探索更多路径、做更多自我校验。对纯事实问答，这种额外思考的收益通常就没那么大。

Q：MCTS 里 LLM 和 PRM 分别扮演什么角色？

> LLM 更像 policy，负责提出候选步骤；PRM 更像 value，负责判断当前路径值不值得继续展开。这跟 AlphaGo 的 policy/value 分工非常像，只不过这里的节点换成了推理步骤。

---

13. 分布式训练与推理优化

13.1 分布式训练基础：显存估算、Adam 与并行策略

为什么需要分布式训练

单参数显存： 1 个参数（BF16）= 2 bytes；参数规模换算：1B 参数 = 2GB。

训练显存四大块（BF16 + Adam）：

```
参数：        N × 2 bytes
梯度：        N × 2 bytes
Adam m：      N × 4 bytes（FP32）
Adam v：      N × 4 bytes（FP32）
──────────────────────────────
合计：        N × 12 bytes
```

简单记忆公式： 参数量（B）× 12 = 训练显存（GB）

70B 模型 = 70 × 12 = 840GB（不含激活值）；840GB / 80GB（A100）= 至少需要 11 张 A100。

72B 级别显存估算手算模板：

- 第一步：72B 参数，BF16 权重：参数 144GB + 梯度 144GB + Adam m 288GB + Adam v 288GB = **合计 864GB**
- 第二步：长上下文训练时，激活值很容易再吃掉 100GB+；实际训练还要留出激活值、通信 buffer、CUDA allocator 碎片、checkpoint / optimizer 临时峰值。
- 第三步得出工程结论：如果是全量微调，8 张 80GB 卡远远不够。常见选项只有三种：上 ZeRO-3 / FSDP + TP/PP；改成 QLoRA / LoRA；降模型规模或降上下文长度。

> **面试答法：** 不要只报一个数字，要顺手补一句：理论静态显存只是下界，真实训练还要给激活、buffer 和碎片留余量。

Adam 优化器原理与 FP32 必要性

为什么需要 m 和 v?

- 普通梯度下降：`W = W - lr × gradient`，梯度每步都在变，更新不稳定。

Adam 维护两个"记忆"：

- **一阶动量 m（梯度的指数移动平均）：**

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot \text{gradient}_t, \quad \beta_1 = 0.9$$

作用：平滑梯度方向，防止单步梯度太大。

- **二阶动量 v（梯度平方的指数移动平均）：**

$$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot \text{gradient}_t^2, \quad \beta_2 = 0.999$$

作用：估计梯度幅度，实现自适应学习率。

- **更新公式：** $\text{params} = \text{params} - lr \cdot \hat{m}/(\sqrt{\hat{v}} + \varepsilon)$
  - 梯度大的参数（v 大）→ 步长被缩小 → 保守更新
  - 梯度小的参数（v 小）→ 步长被放大 → 激进更新

为什么 m 和 v 必须用 FP32?

v 存的是梯度的平方：梯度可能很小（0.0001），梯度平方 = 1e-8。BF16 精度不够表示 1e-8 → 直接变成 0 → v = 0 → 除以 √v = 除以 0 → NaN → 训练崩溃。

结论： 参数和梯度用 BF16（省显存），m 和 v 必须用 FP32（数值稳定），优化器状态占训练显存的 66%。

三种并行策略

数据并行（Data Parallelism）：

- 做法：每张卡有完整模型副本，处理不同数据；各自算完梯度 → All-Reduce 汇总 → 各自更新。
- 局限：每张卡要存完整模型，只能加速，不能解决"模型放不下"的问题。
- 代表：PyTorch DDP

张量并行（Tensor Parallelism）：

- 做法：参数矩阵按列切成 N 份，每张卡存一份：`GPU1: X @ W_1 = Y_1`，`GPU2: X @ W_2 = Y_2`，合并：`Y = [Y_1, Y_2]`。
- 优势：解决单层显存不够的问题。代价：每次前向传播需要 GPU 间通信（激活值大小），要求高速连接（NVLink）。
- 代表：Megatron-LM

流水线并行（Pipeline Parallelism）：

- 做法：模型按层切开，每张卡存不同层（GPU1：第 1-20 层；GPU2：第 21-40 层）。
- 问题：流水线气泡（Pipeline Bubble），GPU2 要等 GPU1 算完才能开始。

- 解法：Micro-batch（把 batch 切成小块）——GPU1 处理 micro-batch2 时，GPU2 同时处理 micro-batch1 的结果。
- 代表：DeepSpeed / Megatron

ZeRO：切优化器状态

ZeRO（Zero Redundancy Optimizer）三个阶段：

- **Stage 1：切优化器状态（m 和 v）** —— 每张卡只存 1/N 的 Adam 状态，显存节省 ~4x，通信量和数据并行一样。
- **Stage 2：切优化器状态 + 梯度** —— 显存节省 ~8x，通信量和数据并行一样。
- **Stage 3：切优化器状态 + 梯度 + 参数** —— 显存节省 ~64x，代价是前向传播需要 All-Gather 参数（通信量更大）。

实际计算（70B，ZeRO-3，16 张 A100）：

- 每张卡 = 840GB / 16 = 52.5GB ← 放得下（80GB 卡）
- 还剩 27.5GB 放激活值

代表框架：**DeepSpeed**

3D 并行（工业实践）

训练 70B 模型，64 张 A100 的典型配置：

- 数据并行（DP）：4 路 → 4 组，处理不同数据
- 张量并行（TP）：4 路 → 每组内 4 张卡切参数矩阵
- 流水线并行（PP）：4 路 → 每组内 4 张卡切层
- 4 × 4 × 4 = 64 张卡

通信量层级：

- TP：最频繁（层内通信），要求 NVLink（节点内）
- PP：中等（层间激活值），要求高速网络
- DP：最少（梯度汇总），可用 InfiniBand（节点间）

代表框架：Megatron-LM + DeepSpeed 联合使用

五种并行方式汇总：

并行方式	切什么	解决什么	通信开销	代表框架
数据并行	切数据	加速，不解决显存	梯度 All-Reduce	PyTorch DDP
张量并行	切参数矩阵	单层显存不够	激活值	Megatron-LM
流水线并行	切层	层数太多	层间激活值	DeepSpeed

并行方式	切什么	解决什么	通信开销	代表框架
ZeRO-1/2	切优化器/梯度	优化器状态太大	同数据并行	DeepSpeed
ZeRO-3	切所有	参数也放不下	最大	DeepSpeed

13.2 高级并行与训练系统工程

ZeRO-1/2/3 vs FSDP：原理、通信与显存视角

方案	切分对象	前向时是否临时聚合参数	反向时核心通信	优势	代价
ZeRO-1	Optimizer states	否	梯度 All-Reduce	最容易落地，几乎不改训练图	参数和梯度仍完整驻留
ZeRO-2	Optimizer states + Gradients	否	Reduce-Scatter / All-Gather 梯度	显存比 ZeRO-1 更省	参数仍是完整副本
ZeRO-3	Optimizer states + Gradients + Parameters	是	参数 All-Gather + 梯度 Reduce-Scatter	能把超大模型塞进较少 GPU	通信压力最大，对网络更敏感
FSDP（full-shard）	参数 + 梯度 + Optimizer states	是，按 module 粒度聚合	每层前向 All-Gather、反向 Reduce-Scatter	PyTorch 原生、和现有代码集成自然	wrap 策略复杂，参数重组频繁

ZeRO-3 和 FSDP 的本质相同点： 都是在做 **fully sharded training**，都用"参数只在需要计算时短暂聚合"的思路换显存。

两者常见差异：

- DeepSpeed ZeRO-3 更偏"系统级训练栈"，通常和 pipeline、offload、RL rollout 系统一起谈。
- FSDP 更偏 PyTorch 原生生态，适合在已有 Trainer / DDP 代码上渐进升级。

> **面试一句话：** `ZeRO-3/FSDP` 解决的是"模型放不下"，`TP` 解决的是"单层矩阵算不动"，两者不是替代关系，而是互补关系。

3D/4D 并行拓扑落位：NVLink、InfiniBand、CP/SP/EP

拓扑分配原则：

- TP（Tensor Parallel）：通信最频繁，优先放在**节点内 NVLink / NVSwitch**。
- PP（Pipeline Parallel）：传的是层间激活值，可跨节点，但相邻 stage 之间最好仍在低延迟网络上。
- DP / ZeRO 组：更适合跨节点走 **InfiniBand**，因为通信频率相对最低。

进一步的 4D/5D 扩展维度：

- **SP（Sequence Parallel）**：本质是在 TP 组内部沿序列维度切激活，常用于降低 LayerNorm、Dropout、Residual 路径的激活显存。它通常是 **TP 的补丁**，不是替代 TP 的新范式。
- **CP（Context Parallel）**：面向超长上下文，把长序列本身切到多张卡上。目标是解决 `32K / 128K / 1M` 上下文训练时的激活和注意力开销，服务的是**长上下文训练/推理**问题。
- **EP（Expert Parallel）**：专门给 MoE 用，把不同专家分散到不同设备。主要矛盾是 **All-to-All** 通信，而不是普通 dense 模型里的矩阵分片。

并行维度	切分对象	主要解决问题	最依赖的硬件条件
TP	权重矩阵	单层算不动 / 单层显存不够	NVLink / NVSwitch
PP	模型层	模型太深，单卡装不下整网	低延迟跨卡 / 跨节点网络
DP / ZeRO	数据 / 状态	提升吞吐、摊薄状态显存	InfiniBand
SP	激活（序列维）	TP 下的激活显存过大	TP 组内高速互联
CP	长上下文序列	超长 context 训练与推理	长序列通信优化
EP	专家参数	MoE 专家太多放不下	All-to-All 带宽

BF16 vs FP16 vs FP8：工业界的混合精度选择

FP16： 优点是生态成熟、显存省、很多旧卡支持好；缺点是指数位太少，容易 underflow / overflow，工程上必须配合 **loss scaling**。

BF16： 优点是和 FP32 一样的指数范围，数值稳定性明显更强；缺点是尾数精度略差于 FP16，但对大模型训练通常不是主要问题。只要硬件支持，**BF16 基本是训练默认选项**。

FP8： 优点是进一步降低显存和带宽压力，能显著抬高吞吐；缺点是对 kernel、校准、硬件代际（如 H100/B200）和框架支持要求高。更像"高端栈优化项"，而不是通用默认项。

面试结论： 训练默认优先级通常是：`BF16 > FP16`。FP16 适合"老硬件 / 老栈 / 已经验证过的方案"。FP8 适合"硬件和 kernel 很新，且团队能 profile/回滚"的场景。

RL 训练拓扑：角色拆分与资源放置

很多人会讲 PPO / GRPO 的 loss，却答不清楚 RL 系统到底怎么部署。工业里更常见的是下面这套**角色拆分**：

- **Actor / Policy Worker**：持有当前策略模型，负责真正生成回答或轨迹。
- **Rollout Engine**：承担高吞吐采样，通常直接绑定 **vLLM / SGLang** 这样的推理后端；在 Agent 场景下还要接工具、环境、verifier。
- **Reference Model（Ref）**：提供 KL 约束所需的参考 log-prob，一般冻结，服务化部署即可。
- **Reward Model / Rule Checker / Verifier**：开放任务用 RM；数学/代码任务用 verifier；Agent 场景常常是 **RM + Rule + Executor** 混合栈。
- **Trainer**：真正做反向传播和参数更新，常和 rollout 物理隔离，避免训练抢占生成吞吐。
- **Eval Gate**：每轮更新后跑 benchmark、私有集、安全集，不通过就不晋升 checkpoint。
- **Model Registry**：记录 `policy/ref/rm/verifier` 的版本关系，保证可回滚和可追责。

同步 RL vs 异步 RL：stale policy 怎么解释

同步 RL： 采样时所有 actor 用同一版策略。优点是数据最"新鲜"，on-policy 假设最干净；缺点是最慢，所有 worker 要等最慢的一批。

异步 RL： 一部分 actor 已经在采样旧策略，trainer 那边策略已经更新。优点是吞吐更高，GPU 更不容易空转；缺点是出现 **stale policy**，即"用旧策略生成的数据训练新策略"。

stale policy 的后果： importance ratio 波动变大；KL 和 advantage 的统计更难稳定；在 Agent / long CoT 任务里容易让训练变"快但飘"。

工业缓解手段： 限制 actor 落后版本数（最多落后 1~2 个 policy 版本）；用更频繁的 checkpoint broadcast；只对高价值样本做异步 replay，低价值样本直接丢弃；对异步数据做额外 importance weighting 或 freshness filtering。

Rollout/Trainer 权重同步、Replay Buffer 与 Off-policy 边界

Rollout Engine 与 Trainer 的权重同步：

- trainer 正在更新权重，rollout 仍在用旧权重采样，同步频率过低就会出现 stale sample。
- 常见策略：fixed-step broadcast、versioned checkpoint pull、rollout freshness 阈值。

Partial Rollout / Replay Buffer / Async Train-Infer：

- partial rollout 适合长轨迹或昂贵环境；
- replay buffer 适合保留高价值样本；
- async train-infer 则是在吞吐与 freshness 之间做权衡。

风险控制点： 只 replay 最近若干版本的样本；对高奖励但异常长度 / 异常格式 / 异常 tool trace 样本做额外过滤；一旦当前策略和旧样本分布差太远，就宁可舍弃，不强行重复利用。

一个常见反例： rollout 成本太高，于是把旧轨迹反复拿来训练；结果吞吐是上去了，但 advantage、KL 和 sample freshness 一起变差；最后线上效果反而更差。

Ray / K8s / Scheduler / Fault Tolerance

平台化后训练常见基础设施问题：资源调度、actor/trainer 分布式 placement、checkpoint resume、节点驱逐后的恢复、实验版本隔离。

一个常见组合： Ray 管 actor / task 调度，K8s 管容器、资源池与失败恢复。

- **Ray** 更偏任务图、worker placement、actor 生命周期；
- **K8s** 更偏资源池、容器调度、节点驱逐、镜像与作业恢复。

fault tolerance 常见要点： checkpoint lineage、rollout worker 重启、trainer 中断后 resume、节点 eviction 后恢复到最近一致 checkpoint。

Train-Serve 一体化： rollout 本质上就是高吞吐推理；如果 rollout 还沿用 trainer 风格的执行栈，吞吐会被直接拖垮。

> **一句话**：train 和 serve 不一定是同一套系统，但 rollout 非常像 serve。

13.3 推理优化全链路

KV Cache：用显存换速度

为什么需要 KV Cache？ LLM 推理是自回归的，生成第 k 个 token 时，需要前 k-1 个 token 的 K、V；如果每步都重新计算 → 第 1000 步要算 999 个 token 的 KV → 大量重复计算，极其浪费。

解法： 把每个 token 的 K 和 V 缓存起来，每步只计算新 token 的 KV，复用之前的缓存 → 速度提升几十倍，代价是需要显存存储缓存。

KV Cache 显存公式：

```
def estimate_kv_cache(n_layers, batch_size, seq_len,
                      n_kv_heads, d_head, dtype_bytes=2):
    kv_bytes = (2 * n_layers * batch_size * seq_len
                * n_kv_heads * d_head * dtype_bytes)
    return kv_bytes / (1024**3)  # 转 GB

# Llama-70B, batch=32, seq=4096, GQA后kv_heads=8:
# = 2 × 80 × 32 × 4096 × 8 × 128 × 2 ≈ 10.7GB
```

GQA / MQA 与 KV Cache 节省量： KV cache 公式里真正起关键作用的是 `n_kv_heads`。MQA 所有 query head 共享一组 KV head，节省最大；GQA 多个 query head 共享一个 KV 组，节省明显但保留更多表达能力。GQA / MQA 不只是"模型结构改了一点"，而是在直接压缩 decode 阶段最贵的那块显存和带宽。

PagedAttention：消除 KV Cache 碎片

朴素实现的问题：

```
预先给每个请求分配最大可能的显存：
  请求A：预分配 2048 token，实际只用 100 → 浪费 95%
  请求B：预分配 2048 token，实际只用 200 → 浪费 90%

显存碎片化严重，实际利用率只有 20~30%
→ 同时能处理的请求数极少，吞吐量很低
```

PagedAttention 解法： 类比操作系统虚拟内存分页，KV Cache 不预先分配连续空间，而是分成固定大小的 block（通常 16 个 token）：

```
请求A 生成第 1-16 个 token   → 分配 block_1
请求A 生成第 17-32 个 token  → 分配 block_2（不一定连续）
请求B 生成第 1-16 个 token   → 分配 block_3

块表（Block Table）：
  请求A → [block_1, block_2, block_5, ...]
  请求B → [block_3, block_7, ...]
```

效果： 按需分配，无碎片；显存利用率：20% → 90%+。

Prefix Cache（前缀共享）：

- 多个请求共享相同 System Prompt（200 token）时，朴素实现每个请求重新计算，Prefix Cache 只计算一次，用引用计数共享。
- 引用计数管理：请求1 使用 block_0 → 引用计数 = 1；请求2 使用 block_0 → 引用计数 = 2；请求1 完成 → 引用计数 = 1（不释放）；所有请求完成 → 引用计数 = 0 → 可释放或保留缓存。

OS 虚拟内存类比（面试 Q&A）： OS：物理内存不连续，通过页表映射让程序感觉连续。PagedAttention：KV Cache 块不连续，通过块表让 Attention 正常计算。两者都是按需分配，消除预分配碎片。

Flash Attention：IO 感知的 Attention

标准 Attention 的瓶颈：

```
GPU 存储层级（速度）：
  寄存器 > SRAM（40MB，19TB/s）> HBM（80GB，2TB/s）
  SRAM 比 HBM 快约 10 倍

标准 Attention 多次读写 HBM：
  Step1：读 Q、K → 算 QK^T → 写回 HBM
```

```
  Step2：读 QK^T → 算 Softmax → 写回 HBM
  Step3：读 Softmax、V → 算输出 → 写回 HBM

seq=4096 时，QK^T 矩阵 = 32MB
每次都要在 HBM 和 SRAM 之间搬运 32MB
→ IO 成为瓶颈（Memory-Bound）
```

Flash Attention 核心：Tiling + 融合

核心思想：把 Q、K、V 切成小块（tile），在 SRAM 里一次完成所有计算，只有最终输出写回 HBM。

Online Softmax（解决分块计算 Softmax 的难题）：

维护两个统计量：m（当前看到的最大值，数值稳定）和 l（当前的归一化因子，分母）。

每看到新的 K 块：

$$m_{\text{new}} = \max(m_{\text{old}}, \max(\text{new_block}))$$

$$l_{\text{new}} = l_{\text{old}} \cdot \exp(m_{\text{old}} - m_{\text{new}}) + \sum \exp(\text{new_block} - m_{\text{new}})$$

处理完所有块后，结果和标准 Softmax 数学等价，不会因为分块而损失精度。

反向传播：重计算（Recomputation）

Flash Attention 不存中间注意力矩阵，反向传播时重新跑一遍 Tiling 前向计算，重新得到注意力权重再计算梯度。

权衡：

- 多约 1/3 的计算量（额外前向计算）
- 节省 seq² 的显存（seq=4096 省 ~30GB）
- IO 节省 >> 重计算代价 → 整体更快

这叫 **Activation Recomputation（用计算换显存）**。

Flash Attention 演进：

- v1（2022）：提出 Tiling + Online Softmax，速度 2~4x
- v2（2023）：改进 work partitioning，速度再 2x，工业标准
- v3（2024）：针对 H100 的 FP8 + 异步流水线，速度再 1.5~2x

Flash Attention vs PagedAttention 常见混淆：

	Flash Attention	PagedAttention
解决	Attention 计算的 IO 效率	KV Cache 的显存碎片化

	Flash Attention	PagedAttention
阶段	训练 + 推理都用	只在推理时用
改的	Attention 计算本身	KV Cache 的内存管理

两者互不冲突，vLLM 同时使用两者。

Gradient Checkpointing

原理（用计算换激活值显存）：

- 标准训练：前向传播时存储每一层的激活值，用于反向传播计算梯度，激活值显存 ≈ 几十 GB。
- Gradient Checkpointing：只存一部分"检查点"层的激活值，其他层在反向传播时重新计算前向。

效果： 显存节省 60~70%，计算代价：多约 33% 的前向计算。这和 Flash Attention 的重计算思想完全一致。

量化（Quantization）

量化的基本原理： 把 FP16 参数（2 字节）压缩成 INT8（1 字节）或 INT4（0.5 字节）。

```
基本流程：
  Step1：找范围  W_max = 3.7，W_min = -2.5
  Step2：算 scale = (W_max - W_min) / 255 ≈ 0.0243
  Step3：量化  W_int8 = round((W_float - W_min) / scale)
  Step4：反量化  W_float ≈ W_int8 × scale + W_min

精度损失：
  FP16 有 65536 个值 → INT8 只有 256 个值
  朴素 INT8 量化 → perplexity 上升 10~30%
```

三个核心缺陷：

- 缺陷 1：离群值（Outlier）—— 少数参数值极大（如 50.3），为覆盖离群值 scale 被迫很大 → 正常范围内的参数精度极差。
- 缺陷 2：Per-tensor 粒度太粗 —— 整个矩阵用同一个 scale，不同行/列分布差异大 → 有的行精度差。
- 缺陷 3：激活值比权重更难量化 —— 权重固定可离线分析，激活值随输入变化，分布动态变化，离群值更严重。

GPTQ：逐层误差补偿

量化权重 W_i 时产生误差 δ_i，把 δ_i 补偿到相邻权重 W_j 上，用 Hessian 矩阵决定误差分配方向——把误差分配给"最不敏感"的权重方向。

效果： INT4 量化后精度接近 FP16，需要少量校准数据（128 条样本），量化过程需要几小时（只做一次），推理时和普通 INT4 一样快。

AWQ：保护重要通道

核心洞察： 权重的重要性 = 对应激活值的大小，$W_i \times X_i$ 对输出贡献大（X_i 大）→ W_i 是"重要权重"，量化误差影响大 → 需要高精度。

做法（Per-channel scaling）：

- 分析每个通道的激活值大小
- 对重要通道在量化前做缩放：$W_i' = W_i / s_i$（让重要通道的权重更小）
- 量化后误差更小，推理时用 s_i 还原

优势： 比 GPTQ 更快（无需 Hessian 计算），精度和 GPTQ 相当，目前工业界最常用。

量化粒度：

- Per-tensor（最粗）：整个矩阵一个 scale → 精度最差
- Per-channel（中等）：每行/列一个 scale → 精度明显提升
- Per-group（最细）：每 128 个参数一个 scale → 精度接近 FP16（GPTQ 和 AWQ 默认用 Per-group，工业界标准）

方法	核心思路	精度	速度	适用场景
朴素 INT8	直接线性映射	差	快	不推荐
GPTQ	逐层误差补偿（Hessian）	好	慢（量化时）	追求精度
AWQ	保护重要通道（激活感知）	好	快	工业首选
QLoRA	INT4 基础模型 + FP16 LoRA	很好	中	微调场景

FP8 推理与部署约束： FP8 的主要价值在于降带宽、降显存、提升新硬件吞吐。部署约束包括：kernel 支持、校准策略、与现有 INT4 / BF16 服务链兼容性、数值鲁棒性回归。更细的工程关注点：activation / weight 的校准口径是否一致；哪些 layer 能安全下到 FP8，哪些层必须保留更高精度；回归测试是否覆盖长上下文、多语言、structured output、tool-use / agent 任务。

Speculative Decoding（投机解码）

核心思路：

```
Step1：小模型（7B）猜 K 个 token
       比如猜出 ["今", "天", "天", "气"]
```

```
Step2：大模型（70B）一次性并行验证 K 个位置
        只跑一次前向传播！

Step3：接受/拒绝（关键的数学机制）

Step4：生成最终 token
```

接受/拒绝机制（保证输出分布不变）：

对每个猜测的 token x_i：$q(x_i)$ = 小模型的概率；$p(x_i)$ = 大模型的概率。

接受概率：$\text{accept} = \min(1, p(x_i)/q(x_i))$

- 情况 A：$p \geq q \rightarrow$ accept = 1.0，直接接受（大模型更有把握，无条件接受）
- 情况 B：$p < q \rightarrow$ accept = $p/q < 1$，按概率拒绝（大模型不认可，有一定概率拒绝）

拒绝后的修复：从修正分布采样 $p_{\text{corrected}}(x) = \text{normalize}(\max(0, p(x) - q(x)))$

数学证明： 整个序列的输出分布 = 纯用大模型的分布，完全等价，零精度损失。

加速效果：

$$\text{加速比} \approx \frac{K \times \text{接受率}}{1 + K \times \text{小模型速度}/\text{大模型速度}}$$

- 接受率 80%（小大同系列模型）→ 加速 2~3x
- 接受率 50%（模型差异大）→ 加速 1.2x
- 接受率 0% → 比直接用大模型还慢

关键： 小模型和大模型要"气味相投"，最好是同一模型系列（Llama-7B 猜，Llama-70B 验证）。

Speculative Decoding 的边界与反例（什么时候会变慢）：

- draft 太弱
- target / draft 不同域差异太大
- 接受率太低
- verify 成本超过节省掉的 target step

> **面试一句话**：speculative decoding 的收益不是理论保证，而是接受率驱动的工程结果。

Continuous Batching 与 PD 分离

Continuous Batching：

```
朴素 Batching：
  等凑够一批再处理
  请求A完成 → 等待整个 batch 完成
  → GPU 空转等最慢的请求

Continuous Batching：
  请求A完成 → 立刻填入新请求B
  GPU 持续满载
```

效果： 吞吐量提升 5~10 倍，延迟降低 30~50%。这是 vLLM 成为推理标准框架的核心原因之一。

PD 分离（Prefill-Decode 分离）：

两个阶段的特性完全不同：

```
Prefill（预填充）：
  一次并行处理整个 prompt
  Compute-Bound（计算密集）
  GPU 满负荷

Decode（解码）：
  每步只生成 1 个 token
  Memory-Bound（访存密集）
  GPU 大量空转
```

传统混合部署： 两种模式互相干扰。

PD 分离：

- Prefill 集群：专门处理 prompt（算力强的 GPU，H100）
- Decode 集群：专门生成 token（带宽大的 GPU，A100 或更便宜）
- KV Cache 计算完后通过网络传输给 Decode GPU

工程挑战：

- KV Cache 传输（每请求 ~330MB）→ 需要高速网络
- 负载均衡（Decode GPU 约是 Prefill 的 3~5 倍）
- 请求调度（短 prompt 长输出 vs 长 prompt 短输出）

价值： Decode 集群用便宜 GPU → 大幅降低成本；两个集群各自优化 → 整体利用率提升。

推理优化全链路（面试系统设计用）

按**请求生命周期**来讲：

1. **请求进入系统** —— 先判断是否命中 **Prefix Cache**；如果公共前缀（如 system prompt、工具描述）已经缓存，就直接复用 KV。
2. **Prefill 阶段** —— 请求被路由到 **Prefill 集群**；这里的核心优化是 **Flash Attention**，目标是降低 attention 的 IO 成本。
3. **KV 交接** —— Prefill 结束后，把 KV Cache 传给 Decode 侧；如果采用 **P/D 分离**，这一段尤其依赖高速网络。
4. **Decode 阶段** —— 由 **PagedAttention** 管理 KV block，保证高显存利用率；由 **Continuous Batching** 保持请求持续填充，避免 GPU 空转。
5. **进一步提速** —— 可叠加 **Speculative Decoding**，用"小模型猜 + 大模型验"换吞吐；可叠加 **AWQ / GPTQ / INT4** 等量化，降低显存和带宽压力。
6. **结果返回** —— 将生成结果流式返回用户，同时更新缓存与调度状态。

框架视角：

- **vLLM**：`PagedAttention + Continuous Batching` 是底座，再叠加 `Flash Attention + Prefix Cache`。
- **SGLang**：通常在 vLLM 类底座上做更激进的调度和服务编排优化。
- **TensorRT-LLM**：更偏 NVIDIA 官方推理栈，适合深度绑定特定硬件能力。

vLLM 架构拆解与 SGLang 对比

vLLM 的核心不只是一个 attention kernel，而是一整套推理系统：
KV block + block table / page table + PagedAttention + continuous batching + prefix caching + chunked prefill。**为什么它强**：它把内存管理、调度、缓存复用和吞吐问题一起解决了。

vLLM 2025-2026 工程补充：

- **Automatic Prefix Caching**：官方采用 **hash-based** prefix caching，把每个 KV block 由"当前 block token + 前缀 block hash"唯一标识，本质上是把公共前缀复用问题做成缓存命中问题。
- **Chunked Prefill**：价值不只是"把长 prompt 切块"，更关键是把 **TTFT** 与 **TPOT** 的矛盾暴露给调度器，让超长 prompt 不至于长期阻塞 decode。
- **Disaggregated Prefill**：prefill 更偏 compute-bound，decode 更偏 memory-bound；把两者解耦后，更容易按不同并行策略和实例数独立调优。
- **系统视角总结**：vLLM 真正的面试亮点通常不是单个 kernel，而是 **PagedAttention + scheduling + cache reuse + prefix caching + P/D disaggregation** 形成的复合收益。

SGLang vs vLLM：

- vLLM 更像高吞吐 serving engine。
- SGLang 更强调：结构化程序式生成；更强的编排与控制流表达；agent / tool / structured generation 场景适配。
- SGLang 的特色三层：RadixAttention / 前缀复用思路；结构化生成（agent / tool / constrained output 更自然）；程序式控制流 / DSL 风格（把复杂多轮生成写成可编排的执

行流程）。

> **一句话**：vLLM 更像吞吐底座；SGLang 更像把 serving、structured generation 和 agent orchestration 粘在一起的 runtime。纯 serving 吞吐优先时常看 vLLM，复杂 agent / structured execution 更容易想到 SGLang。

服务指标体系：TTFT / TPOT / TPS

- **TTFT（Time To First Token）**：用户最先感知的延迟。
- **TPOT（Time Per Output Token）**：流式生成阶段的每 token 延迟。
- **TPS / Throughput**：单位时间系统能吞多少 token 或多少请求。

为什么要分开讲： 很多优化只会改善其中一项：chunked prefill 更偏 TTFT；continuous batching 更偏吞吐；GQA / PagedAttention 更偏 decode 阶段 TPOT。

推理优化高频追问 Q&A

Q：Flash Attention 和标准 Attention 输出完全一样吗？

> 完全一样。Online Softmax 数学等价于标准 Softmax，只是计算顺序不同。反向传播用重计算而非存储中间值，结果也完全一致。对模型训练和推理结果没有任何影响。

Q：为什么 Speculative Decoding 能保证输出分布不变？

> 通过接受/拒绝机制。接受概率 = min(1, p/q)，拒绝时从修正分布 normalize(max(0, p-q)) 采样。数学上可以证明整个过程等价于纯用大模型采样，零精度损失。

Q：量化后推理速度一定更快吗？

> 不一定。INT4 量化减少了显存占用（从 Memory-Bound 改善），理论上应该更快。但如果推理已经是 Compute-Bound（计算瓶颈），量化反而需要额外的反量化计算，速度可能不提升甚至下降。需要 profile 确认瓶颈在哪里。

Q：PD 分离的核心收益是什么？

> 两个方面：一是让 Prefill 和 Decode 各自使用最适合的 GPU 类型（计算型 vs 带宽型），降低成本；二是消除两种模式互相干扰，各自优化到极致，整体吞吐量提升。

13.4 工业前沿视角与高频面试追问

PyTorch Native 训练栈正在成型（2025–2026）

> 🏭 **工业补充**
>
> **一个重要变化：** 超大模型训练不再只有 `DeepSpeed + Megatron` 这条主线；**PyTorch Native** 训练栈正在快速成熟。

> **FSDP2（`fully_shard`）的官方改进点：** per-parameter sharding，组合并行更直观；对冻结参数的约束更少；支持 communication-free 的 sharded state dict；内存管理更可预测，不再依赖 FSDP1 那种更笨重的 all-gather 限制策略。
>
> **TorchTitan 的工程信号：** 官方仓库已经把下列能力放到同一平台里：FSDP2、Tensor Parallel、Pipeline Parallel、Context Parallel、activation checkpointing、distributed checkpointing、`torch.compile`、**Float8 / MXFP8** 训练。
>
> **工业含义：** 训练平台正在从"框架拼装"走向"原生栈收敛"；这会直接影响：checkpoint 格式互通；full finetune / LoRA / inference 之间的切换成本；新硬件（H100 / Blackwell）上的 Float8 落地速度。
>
> **一句话**：2025-2026 的训练基础设施，不只是并行策略升级，更是**原生框架、checkpoint、编译器、低精度栈一起收敛**。

高频追问标准答法

Q：ZeRO、TP、PP 的本质区别是什么？

> ZeRO / FSDP 主要是在切训练状态，解决"显存放不下"。TP 是在切单层矩阵，解决"这一层算不动"。PP 是在切模型层，解决"整网太深、单卡放不下全部层"。三者不是替代关系，而是不同维度的并行组合。

Q：为什么 BF16 逐渐取代 FP16 成为训练默认值？

> 因为 BF16 保留了和 FP32 接近的指数范围，极大缓解下溢和上溢问题。对大模型训练来说，数值稳定性往往比多一点尾数精度更重要。只要硬件支持，BF16 基本就是更省心的默认选择。

---

14. MoE / 长上下文 / Model Merging

14.1 MoE 核心原理与路由机制

稀疏激活：核心思想与价值

Dense 模型（普通 Transformer）： 每个 token → 经过所有 FFN 参数，计算量 = 参数量。

MoE 模型： 把 FFN 替换成 N 个"专家"（Expert），每个 token 只激活 Top-K 个专家。

```
比如：N=8，K=2
每个 token 只用 2 个专家
激活参数 = 总参数的 2/8 = 25%
```

核心价值： 参数量增大（更多专家）→ 模型容量更大；计算量不变（只激活 K 个）→ 速度不变。

DeepSeek-V3 实例： 671B 总参数，每次只激活 37B —— 用 671B 的知识容量，37B 的计算开销。

activated params vs total params：为什么 MoE"参数很大但计算不大"

- **total params（总参数）**：把所有专家参数、共享参数、attention 参数全部算上，代表模型的"知识容量上限"。
- **activated params（激活参数）**：某个 token 在一次前向传播里真正用到的参数量，代表实际 FLOPs 和时延更接近的量。

面试最常见误区： `671B` 不是说每次都做 `671B` 的计算；真正和吞吐、时延更相关的是 `37B activated` 这一类数字。

> **一句话**：Dense 模型里 `total params ≈ activated params`；MoE 里两者被故意拉开，这正是稀疏激活的价值所在。

Router 路由机制

结构： 一个小的线性层 + Softmax。

```
router_logits = token_embedding @ W_router
router_probs  = Softmax(router_logits)
top_k_experts = argmax(router_probs, k=2)
```

示例（token"导数"）：

```
专家1（数学）: 0.6  ← 选中
专家2（物理）: 0.3  ← 选中
专家3（文学）: 0.05
专家4（代码）: 0.05
→ 选专家1和专家2处理这个 token
```

Top-K Routing 的完整 forward 逻辑

面试里常从"Router 就是一层线性层"继续追问到完整前向传播链路：

1. 输入 token hidden state：$h_t \in \mathbb{R}^d$
2. Router 打分：$z_t = h_t W_{\text{router}}$
3. Softmax 变成概率：$p_t = \text{softmax}(z_t)$
4. 取 Top-K 专家索引：`idx = topk(p_t, K)`

5. 只保留被选中的 K 个概率，并重新归一化：

$$\tilde{p}_{t,i} = p_{t,i} / \sum_{j \in \text{topK}} p_{t,j}$$

6. dispatch：把 token 发送到对应专家所在设备
7. 专家前向：每个专家各自做 FFN 变换，得到 $e_i(h_t)$
8. combine：输出为 $\sum_{i \in \text{topK}} \tilde{p}_{t,i} \cdot e_i(h_t)$

关键点： Router 只负责"分流"和"加权"，不负责真正建模内容；专家输出最后还要**按 router 权重加权求和**，不是简单拼接。

Router 设计全景：Top-K / Expert Choice / Soft-MoE

- **Top-K routing**：最主流，易实现，稀疏性强。
- **Expert Choice**：让专家反向挑 token，天然更利于负载均衡。每个专家主动选择 Top-K 个 token 处理 → 每个专家负载天然均衡。缺陷：部分 token 可能被多个专家选中（重复计算），部分 token 可能没有专家选中（被忽略），要求固定序列长度，不适合变长推理，主要用于训练，推理时仍用 Token Choice。
- **Soft-MoE**：更连续、更平滑，但稀疏性和成本优势不如硬 Top-K 明显。

14.2 负载均衡、架构设计与训练挑战

Expert Collapse 与三种均衡解法

Expert Collapse（专家坍塌）的问题根源：

```
Expert1 稍微好一点 → Router 多发 token
→ Expert1 训练更充分 → 更好
→ Router 更倾向 Expert1
→ ... 正反馈循环

最终：Expert1 处理 90% token
      其他专家几乎空载
      → 退化成 1 个专家的 Dense 模型
```

解法 A：辅助损失（传统方案）

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{LM}} + \alpha \cdot \mathcal{L}_{\text{balance}}$$

$$\mathcal{L}_{\text{balance}} = N \cdot \sum_i (f_i \times P_i)$$

- f_i = 专家 i 实际接收的 token 比例
- P_i = Router 分配给专家 i 的平均概率

缺陷： α 超参难调；两个 Loss 互相拉扯，训练不稳定；直接干扰语言模型学习目标。

解法 B：Auxiliary-Loss-Free（DeepSeek-V3 创新）

做法：不修改 Loss，而是修改 Router logits。为每个专家维护偏置项 b_i：

$$\text{router_logits_adjusted} = \text{router_logits} + b_i$$

动态调整规则（不参与梯度）：专家 i 超载 → 降低 b_i（让 Router 少选它）；专家 i 空闲 → 提高 b_i（让 Router 多选它）。

优势： 负载均衡和模型性能完全解耦，不干扰语言学习目标。DeepSeek-V3 用此方案，同等 FLOPs 性能更高。

Auxiliary-Loss-Free 的底层逻辑： 传统辅助损失直接把负载均衡目标加进主 loss，会和语言建模目标拉扯。Auxiliary-Loss-Free 不去改总 loss，而是通过 router bias 或统计项动态修正路由倾向，使负载均衡和语言学习解耦。

解法 C：Expert Choice Routing（见 Router 设计全景）

Global vs Local Routing 的权衡：

- **Global routing**：负载更平衡，但调度和通信更难，跨节点 All-to-All 和调度复杂度更高。
- **Local routing**：更易实现，局部一致性更强，更容易和硬件拓扑对齐，但全局利用率未必最佳。

> **一句话**：架构上不是谁更先进，而是谁更适合当前互联拓扑和延迟预算。

MoE 架构设计三种方案

设计 1：标准 MoE（每层都是 MoE）

- 所有 FFN 层替换成 MoE
- 代表：Switch Transformer

设计 2：交替 MoE（隔层 MoE）

- 奇数层：Dense FFN；偶数层：MoE FFN
- 代表：Mixtral 8×7B
- 优势：减少路由开销，工程简单

设计 3：细粒度 MoE + 共享专家（DeepSeek-V3）

- 专家数：256 个细粒度小专家；每次选：Top-8 个专家；+ 1 个共享专家（每个 token 必经）

共享专家的作用： 256 个专家负责专业化处理，1 个共享专家负责通用能力，防止所有专家过度专业化，保留基础语言能力。

细粒度的优势：

- 粗粒度（8 个大专家，选 2 个）：组合数 = C(8,2) = 28 种
- 细粒度（256 个小专家，选 8 个）：组合数 = C(256,8) ≈ 天文数字 → 更丰富的路由组合，更强的表达能力

MoE 训练挑战

All-to-All 通信（主要瓶颈）：

不同 token 路由到不同 GPU 上的专家 → token 需要跨 GPU 传输。通信量 = batch_size × seq_len × hidden_dim。

DeepSeek-V3 的优化：节点内（8 张卡）：NVLink（900GB/s）；节点间（多节点）：InfiniBand（400Gb/s）；尽量控制路由在节点内完成。

Router 和专家的学习率解耦：

- Router 更新太快 → 专家还没训练好就被换掉
- 专家更新太快 → Router 路由决策过时
- 解法：Router lr = 专家 lr × 0.1~0.3，Router 用更小的学习率，更新更保守

DPO 在 MoE 上的梯度噪声：

问题：DPO 的 chosen 和 rejected 前向传播可能激活完全不同的专家 → log_ratio 计算不稳定，梯度噪声大。

解法：DPO 前先做充分 SFT（稳定专家分工）；或用 temperature scaling 稳定路由；或 SFT 阶段固定 Router 不更新。

MoE + RL 的特有不稳定性

RL 会放大奖励驱动下的路由漂移。结果是：热门专家更热，冷门专家更饿，token-level ratio 更抖。

在 DPO / token-level GRPO 里更明显： `chosen` 和 `rejected` 可能激活完全不同的专家；同一个序列里不同 token 的 importance ratio 波动非常大；结果就是 log-ratio 噪声和梯度噪声同时上升。

GSPO 更友好的直觉： 它把优化重心从 token 级比率抬到 sequence 级比率；对长 CoT 或多专家协同的序列，更新信号更平滑；不会因为单个 token 路由波动就把整个序列的梯度扭得很厉害。

> **面试一句话：** `MoE` 的难点从来不只是"算子变稀疏"，而是"稀疏路由把优化问题也变得更脆弱"；`GSPO` 的价值之一，就是减弱这种 token 级抖动。**这也是 GSPO 在 MoE 场景里更有吸引力的原因之一。**

14.3 MoE 推理部署

核心矛盾与三种解法

核心矛盾： 671B 参数需要放在显存里，推理时每步只用 37B 的计算量，其他 634B 参数"占着茅坑不拉屎" → 显存极度浪费。

解法 A：Expert Parallelism（专家并行）

把 256 个专家分散到不同 GPU：

```
GPU1：专家 1-32
GPU2：专家 33-64
...
GPU8：专家 225-256
```

每张 GPU 只存 1/8 的专家参数：671B / 8 = 约 84B 参数每张卡。

代价： token 路由时需要 All-to-All 跨 GPU 通信；推理 batch 小时，通信开销 >> 计算开销 → 推理延迟高。

解法 B：Expert Caching（专家缓存）

观察：热门专家被高频选中（80% 的 token），冷门专家几乎不被选中。

做法：分析历史路由统计，把 Top-K 热门专家常驻每张 GPU 显存。热门专家：本地计算，零通信延迟；冷门专家：按需从其他 GPU 加载。

效果： 80% 的 token 无需跨 GPU 通信，整体延迟大幅降低。思想和 CPU Cache / Prefix Cache 完全一致——把高频访问的数据放在快速存储里。

Expert Cache 的更细规则：

- 热专家常驻在低延迟显存；
- 次热专家做短期缓存；
- 极冷专家按需迁移或远端拉取。
- **经验法则：** cache size 往往要覆盖"高频活跃专家集合"的至少一到两倍波动范围；否则一旦热点切换，就会频繁抖动。

解法 C：活跃/非活跃专家分级量化

- 活跃专家（当前 token 要用）：FP16 精度
- 非活跃专家（当前不用）：INT4 压缩存储

```
671B 模型：
  37B 活跃参数：FP16 = 74GB
```

```
634B 非活跃参数：INT4 = 317GB
合计：391GB（比全 FP16 的 1342GB 少 70%）
```

代价： 非活跃专家被调用时需要反量化。

Expert Parallel 与 All-to-All：为什么它是 MoE 最大工程瓶颈

Dense 模型的主要通信： All-Reduce、Reduce-Scatter、All-Gather。

MoE / EP 的主要通信升级成： All-to-All。

为什么更难：

- 每个 token 可能路由到完全不同的专家；
- 每张卡既要发送自己手里的 token，也要接收其他卡送来的 token；
- 发送量和接收量都随路由分布实时波动。

工程后果：

- 一旦热门专家跨节点，网络抖动会直接放大成尾延迟；
- batch 小时，通信启动开销甚至比真正计算还重；
- 因为 routing 每步都可能变化，缓存和静态编排都更难做。

典型缓解手段：

- 热门专家尽量共置于同一节点；
- 限制跨节点 dispatch 比例；
- 做 capacity control，避免少数专家吸走全部 token；
- 训练时和推理时采用不同的 expert placement 策略。

Dynamic Expert Placement： 推理时最实用的工程思路是热专家常驻、冷专家按需取、placement 根据历史路由分布动态调整。

DeepSeek-V3 推理部署实践

- 32 张 H800，Expert Parallelism：每张卡存 8 个专家；节点内 NVLink 通信（900GB/s）；节点间 InfiniBand（400Gb/s）。
- **Prefill 阶段：** 大 batch 处理，摊薄 All-to-All 通信开销。
- **Decode 阶段：** 热门专家复制到每张卡（Expert Caching）；冷门专家按需加载；+ PD 分离，Decode 集群独立部署。

MoE vs Dense 完整对比

维度	Dense 模型	MoE 模型
参数量	较小	很大（稀疏激活）

维度	Dense 模型	MoE 模型
每步计算量	= 参数量	<< 参数量
训练显存	N × 12 bytes	更大（所有专家）
推理显存	参数量 × 2	所有专家 × 2（但只算 K 个）
负载均衡	不需要	必须处理
通信开销	低	高（All-to-All）
代表模型	Llama/Qwen	DeepSeek-V3/Mixtral

MoE 高频追问 Q&A

Q：MoE 的专家会真正"专业化"吗？

> 会，但不完全按我们预期。实验发现专家会自发形成某种专业分工，但往往不是"数学专家""文学专家"这么明确，更多是一些隐含的语言特征维度。可视化 Router 的路由决策，可以看到相似语义的 token 确实倾向于被路由到相似的专家组合。

Q：为什么 DeepSeek-V3 要用共享专家？

> 防止所有专家过度专业化。如果每个 token 完全由专业化专家处理，模型可能丧失跨领域的通用推理能力。共享专家作为"通用锚点"，每个 token 必经，保留了模型的基础语言能力。

Q：Auxiliary-Loss-Free 为什么比辅助损失更好？

> 核心是解耦。辅助损失直接参与反向传播，和语言模型损失同时优化，两者经常互相拉扯——负载均衡了但模型性能下降，或模型性能好了但负载不均。DeepSeek-V3 的偏置项不参与梯度，纯粹通过统计负载来动态调整，完全不干扰语言学习，实验表明同等 FLOPs 下性能更高。

Q：推理时 Expert Parallelism 和训练时有什么区别？

> 最大区别是 batch size。训练时 batch 大，All-to-All 通信的开销被大量 token 摊薄，通信效率可接受。推理时 batch 小甚至 batch=1，每个 token 都触发跨 GPU 通信，通信延迟成为主要瓶颈。所以推理时需要额外的 Expert Caching 或 PD 分离来缓解。

14.4 长上下文训练

三个核心障碍

障碍 1：显存随序列长度平方增长

Self-Attention 注意力矩阵大小 = seq_len × seq_len × num_heads × dtype_bytes：

```
seq_len = 4096:    4096² × 32 × 2 ≈ 1GB（可接受）
seq_len = 32768:   32768² × 32 × 2 ≈ 64GB（超出单卡）
seq_len = 128000:  128000² × 32 × 2 ≈ 1TB（完全不可能）
```

Flash Attention 把注意力矩阵显存降到 O(n)，但计算量仍然是 O(n²)。

障碍 2：训练数据稀缺

互联网文本大多很短（推文/新闻/博客）。

核心难点：

- 容易构造但没用：给一本书，问"主角叫什么？"→ 第一段就有答案，不需要长上下文。
- 真正有价值：需要综合文档头部+中部+尾部的信息才能回答的问题 → 考验真实的长距离信息整合能力。

障碍 3：RoPE 位置编码的 OOD 外推问题

RoPE 训练时见过位置 0~4095，推理时输入位置 4096~32767 → 从未见过 → OOD。

外推失败的数学根源：

旋转角度：$\text{angle} = m \times \theta_i$，$\theta_i = \text{base}^{-2i/d}$

```
中频维度，训练时：4095 × θ_i ≈ π（半圈）
外推时：32767 × θ_i ≈ 8π（4圈）

三角函数周期性：sin(8π) = sin(0)
→ 位置 32767 的编码 = 位置 0 的编码
→ 模型把远处位置当成开头
→ 注意力机制完全混乱
```

RoPE 外推四种解法

解法 A：Position Interpolation（PI）

核心思想：线性压缩所有位置到训练范围内。

```
训练范围：0~4095，目标长度：32768（8倍）
压缩：新位置 m' = m / 8
位置 32767 → m' ≈ 4095（在训练范围内）
```

代价： 相邻 token 位置差从 1 变成 1/8，近距离位置分辨率下降 8 倍，模型很难区分相邻 token 的位置，必须配合微调才能恢复精度。

解法 B：NTK-aware Scaling

核心洞察：高频维度已有完整周期，不需要太多压缩；低频维度还没有周期，需要大幅压缩；不同维度应该有不同的缩放策略。

做法：修改 base（不修改位置索引）。

$$\theta_i = (10000 \times \alpha)^{-2i/d}, \quad \alpha \approx \left(\frac{L_{\text{new}}}{L_{\text{train}}}\right)^{d/(d-2)}$$

效果： 等效于"拉长"低频维度的周期，高频维度几乎不受影响，比 PI 的分辨率损失更小，不需要微调就能工作（直接外推）→ 通常作为"快速外推"的应急方案。

解法 C：YaRN（当前主流）

综合了 PI 和 NTK 的优点，对不同频率维度用不同策略：

- 高频维度（已有完整周期）：不压缩，直接使用原始位置
- 中频维度（部分完整）：用 NTK 方式轻微调整 base
- 低频维度（几乎没有周期）：用 PI 方式线性压缩

额外：注意力温度缩放——外推时注意力分数分布变宽，加入温度系数 t：

$$\text{Attention} = \text{softmax}\left(\frac{QK^T}{\sqrt{d} \cdot t}\right), \quad t = \sqrt{\log(L_{\text{new}}/L_{\text{train}}) + 1}$$

→ 保持训练时的注意力"尖锐程度"。

解法 D：ABF（Adjusted Base Frequency）—— 工程最简单

直接把 base 从 10000 改大：Llama-3 将 base 从 10000 → 500000。

效果： 旋转频率变慢，周期变长，模型能外推到更长序列。简单粗暴但工程友好，Llama-3 就用这个方案扩展到 8K。

四种方法对比：

方法	核心做法	近距分辨率	是否需要微调	工业使用
PI	压缩位置索引	差	必须	早期
NTK	修改 base	中	可以不用	应急
YaRN	分频率策略	好	少量	当前主流
ABF	直接改大 base	中	需要	工程首选

NTK-aware / YaRN / Dynamic NTK 补充对比： NTK-aware 更像快速外推修正；YaRN 对不同频率维度做差异化处理，兼顾远距离外推和近距离分辨率；Dynamic NTK 更强调按上下文长度动态调整位置编码参数。

长上下文训练数据构造（五种方法）

方法 1：论文/书籍跨章节 QA 对

针对有明确章节结构的长文档，构造跨章节问题：

- 引用关系追踪："论文方法章节提出的假设 X，实验章节哪个结果验证了它？"
- 前后一致性："作者在开头说 X，后来改变了观点吗？"
- 多段落综合："综合论文三个实验，最重要的发现是什么？"

验证真的需要长上下文：用短上下文模型（2048 token）回答，能答对 → 太容易，丢弃；答不对 → 真需要长上下文，保留。

方法 2：主题相关文档拼接

把同一主题的多篇文章拼接：`[强化学习论文1][强化学习论文2][综述]`

构造的问题："这三篇论文中，哪种方法在效率上有共同优势？"→ 必须跨文档整合信息。比随机拼接更有价值（文档间有语义关联）。

方法 3：Needle-in-a-Haystack

在大量无关文本中插入关键信息：

```
[1000 token 无关新闻]
[插入："小明电话：138-0000-1234"]
[继续无关内容... 总共 32768 token]

问题："小明的电话号码是多少？"
```

针的位置：随机分布在文档任意位置。

评测矩阵：

- X 轴：针的位置（0%~100%）
- Y 轴：文档总长度（4K~128K）
- 颜色：准确率
- 好模型：整个矩阵高准确率；差模型：文档中间位置准确率低 → "Lost in the Middle"现象

方法 4：Lost in the Middle 针对性训练

研究发现：模型对文档开头和结尾信息敏感，对文档中间信息经常"忘记"。

针对性构造：专门把关键信息放在文档中间，强制模型必须检索中间信息，训练后各位置注意力更均匀。

方法 5：合成推理链（最高质量）

```
Step1：准备长文档（技术手册 50000 token）
Step2：用 GPT-4 生成跨位置问题：
  "根据第3章的安装步骤和第7章的故障排查，
  出现错误码 E303 应该怎么处理？"
Step3：用 GPT-4 生成带引用的答案：
  "根据第3章第4节（~15000 token 处）步骤5，
  以及第7章第2节（~40000 token 处）故障码表..."
Step4：验证答案真的需要长上下文
```

效果：最高质量，真实考验长距离信息整合。代价：成本高，通常只生成几万条作为种子数据。

长上下文训练工程流程

渐进式上下文扩展（关键）：

不直接跳到目标长度，而是逐步扩展：

```
阶段1：2048 → 4096（主要预训练阶段）
阶段2：4096 → 8192（继续预训练）
阶段3：8192 → 32768（长上下文微调）
阶段4：32768 → 128000（可选）
```

每个阶段：调整 RoPE base（NTK/YaRN）；更换训练数据（逐渐加入更长的文档）；学习率降低（越长越保守）。

为什么渐进式： 直接跳到 128K 会导致训练不稳定，模型需要"逐步适应"更长的上下文。

混合数据配比（防止灾难性遗忘）：

不能只用长文档训练！问题：只用长文档 → 模型忘记短文本处理能力 → 灾难性遗忘（Catastrophic Forgetting）。

长上下文微调阶段的配比：长文档（32K+）50% + 短文档（4K 以内）50%，确保模型同时保持短文本和长文本能力。

Ring Attention：超长序列的分布式方案

问题： 即使有 Flash Attention（O(n) 显存），seq=128K 时单卡显存仍然不够。128K token × hidden_dim × BF16 ≈ 几十 GB，单卡放不下。

Ring Attention 解法： 把超长序列切成 N 段，分布到 N 张 GPU：

```
GPU1：token 0~32767
GPU2：token 32768~65535
```

```
GPU3：token 65536~98303
GPU4：token 98304~131071
```

环形通信（N 轮）：

- 轮 1：每张卡用自己的 Q，attend 自己的 K、V
- 轮 2：每张卡把 K、V 传给下一张卡，用自己的 Q attend 新收到的 K、V
- 轮 3：继续传递...
- 轮 N：完成一圈，每张卡完成完整的 Attention 计算

关键： Q 留在本地，K、V 在环中传递；通信量：每轮 K、V 的大小（不是全量）。

效果： 显存 O(n) → O(n/N)；速度：通信和计算重叠（流水线），几乎无速度损失。这是训练 128K+ 上下文的标准方案。

Context Parallel vs Ring Attention：

- Context Parallel：切的是长序列上下文，目的是缓解长上下文训练的显存和通信压力。
- Ring Attention：更像在超长序列场景下，把 K/V 在多卡间环形传递。两者都服务长序列，但切分维度和通信模式不同。

长上下文 Serving：prefill / chunked prefill / budget

长上下文上线时，真正贵的是：prefill 计算、prefix reuse、chunked prefill 调度、不同请求的预算分配。

Lost-in-the-Middle 与 Chunk-level RM： 长文本里，中间信息最容易被模型和 RM 同时忽视；更实用的评估方式是 chunk-level RM、滑窗评估、位置打散。

> **一句话**：长上下文不是只靠 RoPE 外推撑起来的，更是 serving 系统问题。

长上下文高频追问 Q&A

Q：YaRN 和 PI 的本质区别是什么？

> 处理粒度不同。PI 对所有维度统一做线性压缩，导致高频维度被过度压缩，近距离分辨率变差。YaRN 对不同频率的维度分别处理：高频维度不动（已有完整周期），中频维度轻微调整，低频维度做 PI 压缩。还加入注意力温度缩放，保持训练时的注意力分布形态。

Q：为什么长上下文训练必须混合短文档数据？

> 防止灾难性遗忘。如果只用长文档训练，模型会逐渐忘记处理短文本的能力（权重被长上下文任务覆盖）。混合 50% 短文档确保两种能力同时保持。这和 Continual RL 里"能力外部化防止遗忘"是同一个问题，只是解法不同。

Q：Needle-in-a-Haystack 评测里，为什么模型在文档中间表现最差？

> 这是 Lost in the Middle 现象。模型的注意力机制对序列的开头（primacy effect）和结尾（recency effect）更敏感，对中间部分的注意力权重相对较小。针对性训练（把关键信息放在中间）可以缓解这个问题。

Q：Ring Attention 和普通张量并行有什么区别?

> 切割维度不同。张量并行切的是参数矩阵（按维度切），不同 GPU 处理不同的参数。Ring Attention 切的是序列长度，不同 GPU 处理不同的 token 段。Ring Attention 不修改模型参数，只改变 Attention 的计算方式，可以和张量并行同时使用（3D 并行的一部分）。

14.5 Model Merging

为什么朴素参数平均失败

```
模型 A 某参数：W_A[i][j] = +0.87（学会识别方程结构）
模型 B 同位置：W_B[i][j] = -0.43（学会识别函数调用）

直接平均：(0.87 + (-0.43)) / 2 = 0.22
→ 既不能识别方程，也不能识别函数调用
→ 两个能力都被稀释
```

这叫 **Interference（干涉效应）**。

为什么简单平均会坏： 参数空间不是低维线性平面；不同 task vector 在高维里会出现曲率问题、符号冲突、方向抵消。所以简单平均往往既损失 A 的能力，也损失 B 的能力。

Task Vector：正确的参数融合方式

核心洞察： 微调学到的是"方向"，不是"绝对位置"。

$$\Delta W_A = W_A(\text{微调后}) - W_{\text{base}} \quad \text{（数学方向）}$$
$$\Delta W_B = W_B(\text{微调后}) - W_{\text{base}} \quad \text{（代码方向）}$$
$$\Delta W_C = W_C(\text{微调后}) - W_{\text{base}} \quad \text{（对话方向）}$$
$$W_{\text{merged}} = W_{\text{base}} + \lambda_A \cdot \Delta W_A + \lambda_B \cdot \Delta W_B + \lambda_C \cdot \Delta W_C$$

λ 通过在验证集上 grid search 确定，每个任务独立。

TIES-Merging：解决符号冲突

三步流程：

Step 1 - Trim（剪枝）

绝对值小的参数 = 微调中几乎没动 = 噪声，不是能力，直接置零，避免引入干涉。

Step 2 - Elect Sign（选举符号）

```
某参数位置 [i][j]：
  ΔW_A[i][j] = +0.8
  ΔW_B[i][j] = +0.6
  ΔW_C[i][j] = -0.9  ← 少数派

多数为正 → 该位置采用正方向
```

Step 3 - Merge（选择性合并）

只平均符号一致的参数：

```
mean(+0.8, +0.6) = +0.7  ✓
-0.9 直接丢弃             ✗

对比朴素平均：(0.8 + 0.6 - 0.9) / 3 = +0.17（被大幅稀释）
```

Linear / SLERP / TIES / DARE 全景对比

- **Linear merge**：最简单，成本最低，但最容易被方向冲突伤到。
- **SLERP**：在球面上插值，比直线平均更尊重高维参数空间几何。
- **TIES**：通过 trim + sign consensus 解决符号冲突。
- **DARE**：通过 drop and rescale 降低干扰项，把 merge 做得更稳。

DARE 的底层直觉： 不是所有 task vector 分量都值得保留；一部分其实更像噪声或冲突源。DARE 的思路就是先 drop 掉一部分低价值或高冲突分量，再对剩余分量 rescale，避免整体幅度被削得太狠。

如果面试官追问 DARE 的机制：

- 先看哪些 task vector 分量更像噪声或冲突源；
- 对这部分分量做有控制的丢弃（drop）；
- 再把剩余分量重新缩放（rescale），保证有效信号整体幅度还在；
- 本质上是"降干扰，而不是简单做稀疏化"。

> **一句话**：DARE 的关键不是"多一个 merge 名字"，而是它在解决"怎么把干扰减掉，但又不把有效信号一起抹平"。

Post-Merge SFT（容易被忽略的关键步骤）

TIES-Merging 之后参数融合可能破坏指令遵循格式，需要 1~2% 的 SFT 数据做"缝合训练"，将四个专家的能力在格式层面统一。不需要全量重训，几百步就够。

Merge 后的 Eval / Safety Regression

merge 完成后不能直接上线，至少要复查：benchmark、instruction following、safety、refusal、tool use、hallucination。

> **一句话**：merge 是能力合成，不是质量保证。

Merge 与多目标 Alignment

merge 可以被看作在不同 reward frontier 上取点——比如一个模型更 helpful，一个模型更 safe；merge + 轻量 SFT / DPO 可以在 Pareto front 上找折中点。

Task Vector vs MoE 路由对比

维度	Task Vector	MoE 路由
部署形态	单一模型	多模型
显存占用	1×	N×
推理延迟	无额外开销	路由判断延迟
能力上限	低于专家（有干涉）	等于最强专家
适用场景	边缘部署、资源受限	云端服务、追求极致性能

两者不是竞争关系，是不同场景下的最优选择。

14.6 工业前沿视角与高频面试追问

架构与产品双线收敛（2025–2026）

> 🏭 **工业补充**
>
> **MoE 侧最典型的信号是 DeepSeek-V3：**
>
> - `671B total / 37B activated`；
> - **DeepSeekMoE + MLA + auxiliary-loss-free load balancing + MTP**；
> - 官方还明确写到：预训练用了 **14.8T tokens**；采用 **FP8 mixed precision training**；全过程没有出现不可恢复的 loss spike，也没有回滚。
>
> **这说明一个很重要的工程事实：** 2025-2026 的高效模型，不再只是"MoE 专家变多"；而是 **MoE 稀疏激活 + MLA 降 KV 压力 + MTP 服务推理加速 + FP8 训练栈** 一起协同。
>
> **多模态侧最典型的信号是 Qwen2.5-VL 系列：**

> - 官方明确强调：能直接作为 **visual agent** 做 computer use / phone use；支持 **1 小时以上视频理解**；支持稳定的 **bounding box / point / JSON structured output**。
> - `Qwen2.5-VL-32B-Instruct` 基于 **reinforcement learning** 继续优化得到：更符合人类偏好的回答风格；更强的视觉数学推理；更细粒度的图像理解和视觉逻辑。
>
> **长上下文侧的产品信号：**
>
> - `Qwen2.5-Turbo` 已经把上下文扩到 **1M tokens**；
> - 官方同时强调两件事：不能因为长上下文伤害短文本能力；必须同时优化推理速度，比如用 sparse attention 降低 1M context 下的计算压力。
>
> **一句话**：这一章在 2025-2026 的主旋律，不是单点突破，而是：**架构效率、训练效率、推理效率、产品可用性开始一起收敛**。

高频追问标准答法

Q：为什么说 MoE 是"参数变大了，但计算不一定变大"?

> 因为总参数量和激活参数量在 MoE 里被人为拆开了。模型可以持有很多专家参数，但每个 token 只激活其中 Top-K 个。所以 capacity 变大了，但单次前向的有效 FLOPs 增长没那么夸张。

Q：多模态模型的 hallucination 为什么比纯文本更棘手?

> 因为它不是单纯"说错了"，而是"语言先验压过了视觉证据"。模型可能看都没看清图，就靠语言统计习惯回答。所以多模态对齐必须额外解决 grounding 和 visual evidence utilization。

---

15. 多模态 Post-Training

15.1 视觉编码器：ViT 原理与高分辨率解法

ViT 核心机制

核心思想：把图像切成小块（Patch），每个 Patch 当作一个"视觉 token"

处理步骤如下：

- 输入图像：224×224 像素
- Patch 大小：16×16 像素
- 切成：$(224/16)^2 = 14 \times 14 = 196$ 个 Patch
- 每个 Patch：$16 \times 16 \times 3 = 768$ 个像素值
- → Linear 投影 → 768 维向量
- 最终：图像 → **196 个视觉 token**（每个 768 维）

ViT vs CNN 的本质区别：

维度	CNN	ViT
感受野	局部，层层堆叠才能获取全局信息	Self-Attention，一层即有全局视野
归纳偏置	强（预设相邻像素相关、平移不变）	无（完全从数据学习）
数据适应性	小数据集表现好	大数据集表现更好（VLM 都用 ViT）

位置编码：

每个 Patch 加可学习的位置向量：

\text{patch}_i\text{的输入} = \text{patch}_i\text{_embedding} + \text{pos_embedding}_i

`[CLS] token`：在 196 个 Patch token 前加一个 `[CLS]`，经过 Transformer 后代表整张图的全局特征。

2D RoPE（Qwen-VL 创新）：

给每个视觉 token 分配二维坐标 (row, col)：

$$\text{位置编码} = \text{RoPE_row}(\text{row}) + \text{RoPE_col}(\text{col})$$

优势：

- 模型理解"水平相邻"vs"垂直相邻"
- 对 OCR 和空间推理帮助极大
- tile 切割后空间关系完整保留

高分辨率难题与四种解法

问题根源： 分辨率 1024×1024，patch=16 → 4096 个 Patch，Self-Attention $O(n^2)$ → 计算不可行。

解法 A：更大 Patch（牺牲细节）

patch=32 → token 数减少 4 倍，但细节丢失。

解法 B：动态分辨率（主流，Qwen-VL / InternVL）

- 把高分辨率图切成多个 448×448 的 tile
- 每个 tile 单独过 ViT
- 加缩略图提供全局视野
- 2D RoPE 保留各 tile 的空间位置关系

解法 C：Q-Former 压缩（BLIP-2）

- 用固定 32 个 Query 向量提取信息
- 无论多高分辨率，输出固定 token 数
- 代价：信息瓶颈，细粒度任务弱

解法 D：Linear Attention（降低复杂度）

- $O(n^2) \to O(n)$
- 代价：损失部分全局信息

15.2 视觉语言对齐：CLIP 与 SigLIP

CLIP：对比学习对齐图文空间

训练数据： 4 亿个图文配对

训练目标： 配对图文在向量空间里靠近，不配对图文远离。

流程：

- Image Encoder（ViT）：图1 → v_1，图2 → v_2，...
- Text Encoder（Transformer）：描述1 → t_1，描述2 → t_2，...
- 计算 $N \times N$ 相似度矩阵：对角线（配对）应高，非对角线（不配对）应低

InfoNCE Loss：

$$\mathcal{L}_{\text{image}_i} = -\log \frac{\exp(\text{sim}(v_i, t_i)/\tau)}{\sum_j \exp(\text{sim}(v_i, t_j)/\tau)}$$

$$\mathcal{L}_{\text{total}} = \frac{\mathcal{L}_{\text{图像侧}} + \mathcal{L}_{\text{文字侧}}}{2}$$

零样本分类能力：

不需要专门训练分类器：

- 类别名 → 文字向量：$t_{\text{cat}}, t_{\text{dog}}, t_{\text{car}}$
- 测试图 → 视觉向量：v_{test}
- $\text{class} = \arg\max(\text{sim}(v_{\text{test}}, t_{xxx}))$

ImageNet 零样本准确率 ≈ 76%，完全没见过这些类别的训练数据。

CLIP 的能力边界：

- **擅长：** 图像分类（零样本）、图文检索、粗粒度匹配

- **不擅长：** 细粒度理解、OCR、计数、复杂推理
- **原因：** 训练目标是"整张图和整段文字配对"，不需要理解图像细节 → 这是 VLM 需要在 CLIP 之上继续 Post-Training 的根本原因

SigLIP：CLIP 的改进版（当前主流）

CLIP 的问题： InfoNCE 需要全局 Softmax，batch 越大效果越好，大 batch → 显存爆炸。

SigLIP 的改法（Sigmoid Loss）：

不用 Softmax（需要全局归一化），改用 Sigmoid（每个图文对独立二分类）：

$$\mathcal{L} = -\sum_{i,j} \log(\sigma(y_{ij} \times \text{sim}(v_i, t_j)/\tau))$$

其中 $y_{ij} = +1$（配对）或 -1（不配对）。

优势：

- 不需要全局归一化
- 可以用极大 batch（32768）
- 负样本更多 → 对比更充分 → 效果更好

> 🏭 **工业补充**
>
> 当前 VLM 领域 SigLIP 已基本取代 CLIP，成为视觉编码器对比预训练的主流方案。

15.3 跨模态桥接：三种 Projector 方案对比

三种 Projector 方案分别代表不同的信息压缩哲学：Linear/MLP 保全信息量、Q-Former 主动压缩、Cross-Attention 深度融合。以下逐一展开。

方案 A：Linear / MLP Projector（LLaVA，工业主流）

架构： ViT 输出（768 维）→ MLP（768→4096）→ LLM 输入

两阶段训练流程：

- **Stage 1：预训练 Projector**
  - 冻结：ViT + LLM
  - 训练：Projector
  - 数据：大量图文配对（595K 条）
  - 目标：让 Projector 学会"翻译"视觉特征

- **Stage 2：全量 SFT**
  - 冻结：ViT
  - 训练：Projector + LLM
  - 数据：高质量指令对话（665K 条）
  - 目标：视觉指令遵循

LLaVA-1.5 改进： Linear → MLP（两层 + GELU）

视觉到语言的映射是非线性的，MLP 学到更复杂的映射关系。核心洞察：**"好数据 + 简单架构 > 差数据 + 复杂架构"**。

方案 B：Q-Former（BLIP-2，逐渐退出）

架构： 32 个可学习 Query 向量，通过 Cross-Attention 从 ViT 输出中提取信息：

- Q = Query 向量（32 个）
- $K = V$ = ViT 的视觉 token（N 个）
- 输出：固定 32 个视觉 token → 输入 LLM

三阶段训练（复杂但系统）：

- Stage 1a：视觉语言对比（类 CLIP）
- Stage 1b：图文匹配（二分类）
- Stage 1c：图像引导文字生成
- Stage 2：连接 LLM，下游微调

缺陷： 32 个 token 的信息瓶颈，细粒度任务（OCR/计数）明显弱于 LLaVA。

方案 C：Cross-Attention（Flamingo）

架构： 在 LLM 每一层插入 Cross-Attention：

- Q = 文字 token
- $K = V$ = ViT 的视觉 token
- 文字生成的每一步都受视觉影响

优势： 深度融合，天然支持多图对话，每层都能访问视觉信息。

缺陷： 必须修改 LLM 架构，不能复用已有 LLM，工程复杂度高。

三种方案完整对比

维度	Linear/MLP（LLaVA）	Q-Former（BLIP-2）	Cross-Attention（Flamingo）
视觉 token 数	全量（196+）	固定少量（32）	全量
信息损失	最小	较大	最小
细粒度能力	强	弱	强
工程复杂度	低	中	高
修改 LLM	否	否	是
多图支持	弱	弱	强
工业主流	✅ 是	逐渐被取代	少数用

> 💬 **面试话术**
>
> - **Linear/MLP projector**：便宜、简单，适合先把视觉 token 对齐到 LLM 空间。
> - **Adapter 式桥接**：改动小，适合增量升级已有模型。
> - **Cross-Attention 路线**：更适合高分辨率、多图、多帧视频或 GUI Agent，因为语言流可以更细粒度地读取视觉证据。

15.4 VLM SFT：数据体系与 Mask 策略

SFT 数据类型分类

VLM SFT 数据覆盖七种核心任务类型，从简单到复杂依次为：

类型	输入	输出示例
图像描述（Caption）	图片	"图中有一只橙色的猫坐在蓝色沙发上"
视觉问答（VQA）	图 + "图里有几只猫？"	"图里有 2 只猫"
空间关系理解	图 + "猫在沙发的左边还是右边？"	"猫在沙发的左边"
OCR 与文字理解	含文字的图 + "图中的价格是多少？"	"图中显示价格为 ¥299"
图表理解	折线图 + "2023 年的增长率是多少？"	"2023 年增长率为 23.5%"
多图对比	两张图 + "这两张图有什么不同？"	"第一张图是白天，第二张是夜晚..."

类型	输入	输出示例
视觉推理（VQA11y 方向）	场景图 + 复杂问题	多步推理的详细答案

SFT Labels 掩码策略

VLM SFT 的核心特殊性在于**图像视觉 token 不参与 Loss 计算**：

```
# VLM SFT 的特殊之处：图像视觉 token 不参与 Loss

input_ids = [SYSTEM, USER, <image_tokens>, question, ASSISTANT, answer]
labels    = [-100,   -100, -100×N_visual, -100,     answer_tokens]

# 图像视觉 token 设为 -100
# 原因：模型不需要"学会生成视觉 token"
#       只需要"根据视觉信息生成文字"
```

最核心的原则：

- 视觉 token 通常不参与文本生成 loss；
- 训练重点是"基于视觉证据生成文本"，而不是"重建视觉 token"。

Interleaved image-text 场景下：

- 用户文本通常 mask；
- assistant answer 开 loss；
- 图片占位 token 通常 mask；
- 对结构化定位输出可单独监督。

15.5 幻觉问题：成因、评测与缓解

幻觉的本质与成因

表现：

- 图里明明没有香蕉，VLM 说"图中有一个香蕉"
- 图里的文字是"3月15日"，VLM 说"3月25日"

原因：

- LLM 的语言先验太强
- "猫通常坐在沙发上" → LLM 倾向于说沙发
- 视觉信号不够强，语言偏见覆盖了视觉信息

POPE 评测方法

POPE 设计三种难度梯度来衡量模型的幻觉程度：

- **随机：** 随机选不存在的物体问（容易）
- **流行：** 选常见物体问（中等）
- **对抗：** 选图中已有物体的相邻类别（难）——图里有橙子，问"有苹果吗？"

好模型：对抗条件下 Yes 率不会虚高；差模型：不管图里有没有，倾向于说"有"。

三种缓解方法

方法 A：数据层面（加否定样本）

- "图里有狗吗？"（图里没有狗）→ 正确答案：没有
- 让模型学会说"没有"，不要总往"有"方向猜

方法 B：MM-RLHF（针对幻觉的偏好对齐）

构建幻觉偏好数据：

- chosen：`"图里有 3 个人"`（正确）
- rejected：`"图里有 5 个人"`（幻觉）

用 DPO 训练：偏向如实描述，远离幻觉回答。

方法 C：视觉 CoT（先描述再回答）

- "我看到左边有一个人，中间有两个人……总共 4 个"
- 强迫模型"先看图再说话"
- 减少语言先验的影响
- 幻觉率下降 20～30%
- 和 VQA11y 论文的 A-CoT 思路一致

> 💬 **面试话术**
>
> 视觉 CoT 解决幻觉的原理：强迫模型"先看图再说话"。普通回答时，模型可能直接用语言先验输出答案（语言偏见）。视觉 CoT 要求先描述图像内容，这个描述过程强迫模型真正

> 处理视觉信息，从而抑制了语言先验的影响。实验表明幻觉率下降 20～30%。

15.6 工业实践：Qwen-VL、InternVL 与 Encoder 冻结策略

Qwen-VL 核心创新

动态分辨率（Naive Dynamic Resolution）：

- 原图切成多个 448×448 的 tile
- 每个 tile 单独过 ViT → 1024 个视觉 token
- 加缩略图（全局视图）→ 1024 个全局 token

token 数量随图像复杂度动态变化：

图像 tiles	视觉 token 数
1 tile	2048
4 tile	5120

既有局部细节，又有全局信息。

2D RoPE 对 OCR 的影响：

- tile_2（右上角，row=0～31，col=32～63）：
  - 模型知道它和 tile_1 在同一行（row=0～31）
  - 水平偏移 32（一个 tile 的宽度）
- "￥299" 中的 ￥ 和 299 在同一行，2D RoPE 能表达水平相邻关系 → OCR 准确率大幅提升

训练流程：

- Stage 1：大规模视觉语言预训练（14 亿图文对）
  - 冻结 LLM，训练 ViT + Projector
- Stage 2：多任务 SFT（全量微调）
  - 特殊任务：视觉定位（输出边界框坐标）

InternVL 核心创新

核心洞察： LLM Scaling 有效，视觉编码器为什么不能 Scaling?

InternViT-6B： ViT 扩大到 6B 参数（普通 ViT-L = 300M）

- 细粒度视觉理解显著提升
- OCR 和文档理解任务领先

Dynamic High Resolution（DHR）： 类似 Qwen-VL 的 tile 切割，支持任意分辨率输入。

代价： 6B ViT + 7B LLM = 13B 总参数，推理成本翻倍。

Vision Encoder 冻结 vs 解冻策略

策略	适用场景	代价
冻结	先做对齐，节省显存和训练稳定性	能力上限受限
部分解冻	适配 OCR、文档理解、细粒度视觉任务	中等成本
全解冻	能力上限更高	成本大、遗忘风险高

> 🏭 **工业补充**
>
> 常见做法是**先冻后解**，或只解 projector + 高层 vision block。

Qwen2.5-VL / Qwen2.5-VL-32B 的 RL 与 Visual Agent 路线

`Qwen2.5-VL` 的一个核心工业信号是：不只做图文问答，而是直接面向 **visual agent / computer use / phone use** 场景。`Qwen2.5-VL-32B-Instruct` 则进一步表明多模态模型也开始系统性引入 RL 去优化人类偏好、视觉推理和更细粒度感知能力。

截至 2026-04 的公开锚点：

真正完整可核实的官方路线，仍然主要来自 `Qwen2.5-VL` 与 `Qwen2.5-VL-32B-Instruct`；如果面试官追问更后续的 Qwen 视觉线，比较稳妥的答法是：

> "公开锚点仍以 2.5-VL 系列为主，但可以预期路线会继续沿 visual agent、structured output、多模态 RL 和长视频理解延展。"

关于 `Qwen3.5-VL` 的答法边界：

截至 `2026-04-12`，没有核实到可直接引用的独立官方公开技术博客或仓库路线；更稳妥的面试表达是：

> "公开可确认的工业锚点仍以 Qwen2.5-VL / Qwen2.5-VL-32B 为主；更后续路线可以按 visual agent、RL、structured output 延展理解，但不要把未公开细节说成既定事实。"

这类表述本身也是面试加分点——它说明你知道怎么区分"公开可核实事实"和"合理推断的技术路线"，比硬说一个没有官方锚点的 Qwen3.5-VL 细节要稳得多。

如果面试官继续追问"那你会怎么回答 Qwen3 / Qwen3.5 的视觉路线？"：

> "我会先 anchored 在 Qwen2.5-VL 和 Qwen2.5-VL-32B 的公开事实，再补一句：后续路线大概率沿 visual agent、structured output、多模态 RL、长视频理解继续增强，但没有官方锚点的细节我不会当作确定事实说。"

一个更完整的训练链路口径：

- vision pretrain / connector 对齐
- multimodal SFT
- visual preference / RL 优化
- visual agent / structured action / grounding 强化

15.7 视频理解：帧采样、时序建模与长视频处理

Token 数量爆炸：核心挑战

- 视频 30fps，60 秒 = 1800 帧
- 每帧 196 个视觉 token
- 总计：$1800 \times 196 = 352{,}800$ 个视觉 token
- LLM context window：4096～32768 token
- → 远超 context 限制

三种帧采样策略

策略 A：均匀采样

- 每隔 N 帧取一帧
- 简单，覆盖均匀
- 缺点：静止场景冗余，快速变化场景漏帧

策略 B：关键帧提取（内容感知采样）

计算相邻帧余弦相似度：

$$\text{sim}(v_t, v_{t+1}) > 0.95 \Rightarrow \text{冗余，丢弃}$$

$$\text{sim}(v_t, v_{t+1}) < 0.95 \Rightarrow \text{有变化，保留}$$

- 静止场景：大量冗余被丢弃（保留 5 帧）

- 动作场景：变化丰富，保留多帧（80 帧）
- 自适应，按内容决定密度

策略 C：问题感知采样（最精细）

根据任务类型动态决定采样策略：

问题类型	采样策略
"说了什么"	低采样（1fps）
"何时起跳"	高采样（10fps）
"几种颜色"	全局均匀扫描

先用 LLM 分析问题类型，再决定采样方案。

时序建模

时间位置编码：

frame_t 的所有 token 加 \text{temporal_embedding}(t)，模型知道哪些 token 来自哪一帧。

3D Attention（精细但贵）：

- 空间维度：同一帧内不同位置 attend
- 时间维度：不同帧的同一位置 attend
- 计算量 = 2D 的 T 倍

长视频（>10 分钟）处理

解法 A：分段处理 + 摘要压缩

- 视频切成 N 段 → 每段生成文字摘要 → 把摘要拼接后输入 LLM
- 适合：理解性问题

解法 B：稀疏 + 密集混合采样

- 全局：低频均匀采样（每 10 秒 1 帧）→ 整体结构
- 局部：根据问题定位相关片段 → 高频采样
- 先翻目录，再查具体章节
- 代表：LLaVA-Video（2025）

视频 Post-Training 更完整的流程：

- 先做内容感知采样，把十万级 token 压到可训范围；
- 再做视频理解 SFT，让模型学会时间顺序和问题条件；
- 后续再用 verifier / judge / preference 对齐"看没看对、说没说对、时间顺序对不对"。

一个更实用的视频 verifier 组合：

- OCR / ASR
- 关键帧比对
- 时间区间一致性
- 动作前后状态变化

15.8 多模态 RLHF 与 GUI Agent

为什么多模态 Post-Training 比文本更难

文本任务常常只需要判定"答得好不好"；多模态任务还要额外判定：

- 有没有真的看图
- 有没有看对区域
- 有没有把视觉证据转成正确文字
- 工具动作是否真的在界面上成功执行

> 💬 **一句话总结**
>
> 文本 RM 解决的是"会不会说"，多模态 RM / verifier 还要解决"有没有真的看见、有没有真的执行到位"。

MM-RLHF / MM-DPO 核心思路

样本不再只是 `prompt + answer`，而是：

```
image / video / screenshot + prompt + answer
```

偏好标注要同时比较：

- 文本正确性
- grounding 是否真实
- 幻觉是否严重
- 对 GUI / agent 任务来说，动作序列是否有效

VLM RLHF 的特殊挑战：

- 图像 hallucination 比纯文本 hallucination 更难；
- spatial grounding 更难验证；
- 环境反馈更复杂；
- GUI 动作带来的副作用更强。

因此，**多模态 RLHF 通常更依赖 verifier / executor / environment feedback，而不是只靠 judge。**

多模态 Reward Stack / Verifier Stack

文本任务常见 reward：风格、偏好、格式、正确性。多模态任务还要额外加三类 signal：

Verifier 类型	覆盖能力
Perception verifier	OCR / detection / segmentation / caption consistency
Grounding verifier	边界框、point、region alignment 是否对上
Action verifier	GUI 动作后页面状态是否如预期变化

一个工业上更实用的 reward stack：

- 第 1 层：格式正确（是否输出合法 action / schema）
- 第 2 层：感知对齐（是否看对图、看对区域）
- 第 3 层：步骤有效（动作执行后状态是否朝目标推进）
- 第 4 层：最终完成（任务是否成功）

一个更像真实系统的多模态 reward stack 流程：

- 先过 perception / grounding / schema gate；
- 再优化文本质量、回答风格、步骤效率；
- 最后用环境反馈或任务成功率做最终锚点。

GUI Agent 与 Computer Use

GUI Agent 的动作空间通常包括：

`click` / `double_click` / `scroll` / `type` / `drag` / `select` / `hotkey` / `wait`

GUI / Computer Use 完整任务流：

GUI / Computer Use 不该只当作"多模态里的一个小应用"，因为它实际上把以下问题全串起来了：看屏幕 → 理解结构 → 规划动作 → 执行动作 → 观察结果 → 安全回退。

完整七步任务流：

1. 读取 screen / DOM / accessibility tree
2. 压缩成可用状态表示
3. 规划子目标
4. 生成 action schema
5. 执行动作
6. 观察新状态
7. 判断完成 / 继续 / 回退

为什么这比纯文本 agent 难：

- 环境状态不稳定
- 动作成本真实
- 错一步可能整条轨迹都废掉

常见 action schema： click / type / scroll / drag / hotkey / wait

reward 常见分层： schema 正确 → 执行成功 → 页面状态朝目标推进 → 最终任务完成 → 安全合规

UI-TARS / Visual Agent 的工程难点

UI-TARS 这类 visual agent 的难点不只是"大模型会不会点按钮"，而是：

- 状态压缩是否丢信息
- action schema 是否足够细但不过拟合
- 长轨迹里如何 credit assignment
- 失败动作后是否能恢复
- 不同分辨率 / 不同主题 / 不同语言 UI 上是否泛化

工程上通常要补的能力：

- screen parser
- OCR / accessibility tree
- state diff
- action replay
- safety gate

多模态 QA vs UI-TARS / Visual Agent 的本质区别：

- 多模态 QA 主要是"看对再说对"；
- UI-TARS / visual agent 还要做到"看对 → 想对 → 做对 → 做错后还能恢复"。

这也是为什么 visual agent 的 reward / eval 必须比普通 VLM 更重动作正确性与恢复能力。

15.9 多模态 Post-Training 全链路总图

从视觉表示到可执行 Agent

把多模态大模型训练链条讲清楚时，最好不要只讲"ViT + Projector + LLM"，而是按下面这张总图来组织：

层级	核心内容
1. 视觉表征层	ViT / SigLIP / 大规模 vision encoder
2. 跨模态对齐层	projector / adapter / cross-attention
3. 多模态 SFT 层	图文问答、OCR、文档理解、定位、视频理解
4. 多模态偏好与 RL 层	MM-DPO / MM-RLHF / visual preference / action reward
5. 多模态 Agent 层	GUI / Computer Use / phone use / multimodal tool calling
6. 多模态评测层	perception / grounding / action correctness / safety

> 💬 **一句话总结**
>
> 多模态模型不是"给语言模型塞点图片"，而是一整条从视觉表示到执行闭环的训练链。

VLM Post-Training 完整训练阶段

- **Stage 1：Projector 预训练**
  - 冻结：ViT + LLM；训练：Projector
  - 数据：大量图文配对（低质量但量大）
  - 目标：学习基础视觉特征翻译
- **Stage 2：视觉指令 SFT**
  - 冻结：ViT（通常）；训练：Projector + LLM
  - 数据：VQA / 描述 / OCR / 图表 / 空间推理（高质量）
  - 目标：学会回答各类视觉问题
- **Stage 3：偏好对齐（MM-RLHF / DPO）**
  - 目标：减少幻觉，提升视觉推理准确性
  - 数据：(图, 问题, 好回答, 坏回答) 四元组

- **Stage 4（可选）：视觉 GRPO**
  - 针对可验证视觉任务（计数 / OCR / 数学图表）
  - 用 Outcome Reward 强化推理链
  - = R1 训练范式在视觉领域的应用

多模态评测：Perception / Grounding / Action 三层

如果把多模态评测讲成一个大章节，最清晰的分法通常是三层：

层级	核心问题	典型指标
Perception	有没有看见、看清、看全	OCR 准确率、caption 质量、对象识别
Grounding	文本与区域是否对齐	box / point 命中率、region correctness
Action	动作是否推动任务完成	task success、action success、recovery ability

一个常见误区： 只看 VQA 正确率；但 GUI / agent 场景里，更重要的往往是 grounding 和 action。

多模态系统设计答法（面试模板）

> 💬 **面试话术**
>
> 如果面试官让你"设计一个多模态后训练系统"，一个稳妥回答模板：
>
> 1. 先分 perception / grounding / action 三层目标；
> 2. 数据上分 image / video / GUI / OCR / doc 四类；
> 3. 训练上先做 multimodal SFT，再做 MM-DPO / RL；
> 4. reward 上组合 perception verifier、grounding verifier、task success；
> 5. 最后用多模态私有评测和 visual agent workflow 做上线门控。

15.10 高频追问 Q&A

> ❓ **Q：为什么 LLaVA 比 BLIP-2 更好，尽管 LLaVA 更简单？**
>
> A：关键是信息瓶颈。LLaVA 把所有 196 个视觉 token 全部传给 LLM，信息完整；BLIP-2 的 Q-Former 把视觉信息压缩成 32 个 token，大量细节丢失。在细粒度任务（OCR、计数、空间关系）上，LLaVA 的信息量优势非常明显。"简单架构 + 好数据 > 复杂架构 + 差数据"。

> ❓ **Q：SigLIP 和 CLIP 的本质区别是什么？**
>
> A：Loss 函数不同。CLIP 用 InfoNCE（Softmax），需要整个 batch 的全局归一化，batch 越大效果越好但显存受限。SigLIP 用 Sigmoid（每个图文对独立二分类），不需要全局归一化，可以用极大的 batch size（32768），负样本更充分，效果更好。

> ❓ **Q：2D RoPE 对哪些任务提升最大？**
>
> A：OCR 和空间推理任务。OCR 需要理解文字的水平排列关系，2D RoPE 能表达"同一行的 token 距离近"；空间推理需要知道物体的相对位置，2D RoPE 让模型能计算两个 token 在图像中的行列差，从而推断空间关系。

> ❓ **Q：幻觉问题用视觉 CoT 解决的原理是什么？**
>
> A：强迫模型"先看图再说话"。普通回答时，模型可能直接用语言先验输出答案（语言偏见）。视觉 CoT 要求先描述图像内容，这个描述过程强迫模型真正处理视觉信息，从而抑制了语言先验的影响。实验表明幻觉率下降 20～30%。

> ❓ **Q：视频理解和图像理解最大的工程区别是什么？**
>
> A：Token 数量爆炸和时序建模。图像处理 196 个视觉 token，视频可能需要处理十万量级。解决方案：内容感知帧采样（相似度阈值过滤冗余帧）+ 时间位置编码（让模型理解帧的时序关系）+ 长视频分段处理。

> ❓ **Q：多模态 Verifier 的常见来源有哪些？**
>
> A：OCR 结果校验、目标检测 / grounding 模型、DOM / Accessibility tree、执行器返回值（按钮是否点到、页面是否跳转成功）、强 judge 模型对图文一致性的复核。

---

16. 评测治理

16.1 评测框架与核心 Benchmark

三个评测层次

评测体系从三个维度覆盖模型的综合能力：

层次	关注点	代表 Benchmark
层次 1：能力评测	模型会什么（知识与推理）	MMLU、HumanEval、GSM8K
层次 2：对齐评测	模型是否安全可信	TruthfulQA、HarmBench
层次 3：用户体验评测	用户是否喜欢	Chatbot Arena、MT-Bench

核心 Benchmark 详解

MMLU：

- 内容：57 个学科的多选题（约 15,000 道），覆盖数学/物理/化学/历史/法律/医学等
- 评测方式：4 选 1，选最可能的答案
- 局限：2024 年后严重污染，区分度下降

HumanEval：

- 内容：164 道 Python 编程题，给函数签名和文档，让模型补全代码
- 评测方式：pass@k（k 次生成至少 1 次通过测试用例）
- 局限：样本量太小，顶级模型已接近 90%+，区分度丧失

数学难度演进：GSM8K　MATH　AIME

- GSM8K：小学数学文字题，顶级模型接近满分，已失去区分度
- MATH：竞赛数学，5 个难度级别，Level 5 仍有挑战
- AIME：美国数学邀请赛，每年 15 题，极难
  - 2024 年：GPT-4 约 20%，DeepSeek-R1 约 72%
  - 目前区分顶级推理模型的最重要 benchmark

LiveCodeBench（动态 Benchmark 代表）：

- 核心创新：持续更新——从 LeetCode/Codeforces/AtCoder 持续收集新题，新题发布后立即加入 benchmark，旧题可能被训练后逐渐降权
- 优势：新题发布时间 > 模型训练截止时间，模型绝对没见过 → 彻底解决污染问题

Chatbot Arena（最可信）：

- 设计：真实用户提问（不预设题目），盲测两个匿名模型，用户选更好的那个，Elo 评分系统计算排名
- 为什么最可信：真实用户提问无法提前训练，盲测无品牌偏见，数百万次对战样本充足，多样化场景覆盖
- 局限：成本高，速度慢，不可重复，不适合快速迭代时的日常评测

优秀 Benchmark 的五个核心特性

特性 1：新鲜度（Freshness）

题目发布时间 > 模型训练截止时间，持续更新，旧题降权。代表：LiveCodeBench、LiveBench。

特性 2：抗污染性（Contamination Resistance）

- 方法 A：程序化生成（答案随机，背答案无效）
- 方法 B：开放生成题（无固定答案，用测试用例判断）
- 方法 C：对抗性设计（测真理解 vs 模式匹配）

特性 3：多维度覆盖（Coverage）

- 能力维度：知识/数学/代码/长上下文/多模态/工具使用
- 任务类型：选择题/生成题/交互式
- 难度梯度：基础/进阶/专家级

特性 4：真实场景对齐（Ecological Validity）

- τ^2-Bench：真实电商/航空客服 Agent 场景
- SWE-Bench：真实 GitHub Issue 修复
- WebArena：真实网页操作任务

特性 5：可靠的评测方式（Evaluation Reliability）

评测方式	适用场景	特点
客观题（自动）	MMLU/GSM8K	可重复，无主观性
程序化验证	HumanEval	运行代码看测试用例
LLM-as-Judge	MT-Bench	评测开放生成
人类偏好	Chatbot Arena	最真实但成本最高

Agent 专项评测

评测维度：

- **维度 1：工具调用准确性**——程序化验证（JSON schema 是否正确），指标：Tool Call Accuracy
- **维度 2：任务完成率（最重要）**——可验证任务直接验证结果，不可验证任务用 LLM-as-Judge，指标：Task Completion Rate（TCR）
- **维度 3：效率**——指标：Steps to Completion（越少越好）
- **维度 4：鲁棒性**——注入错误和边界情况，评测模糊指令处理 / 工具错误恢复

代表 Benchmark：

- τ^2-Bench：电商/航空客服真实场景
- AgentBench：多领域综合评测
- SWE-Bench：真实代码修复任务

16.2 自动评测指标：三代演进

第一代：BLEU / ROUGE（字符串匹配）

BLEU（2002，机器翻译）： 计算模型输出和参考答案的 n-gram 重叠率，精确率视角——"我说的有多少是对的？"

ROUGE（2004，文本摘要）： 计算参考答案在模型输出里的覆盖率，召回率视角——"参考答案有多少被我覆盖了？"

- ROUGE-1：单词重叠
- ROUGE-2：双词重叠
- ROUGE-L：最长公共子序列（不要求连续）

共同缺陷：

- 只看字面匹配，不看语义——"非常精彩"≠"十分出色"（BLEU 低，但语义相同）
- 需要参考答案（开放创意任务无法用）
- 鼓励输出短答案（重叠率更高）

第二代：BERTScore（语义向量相似度）

核心突破： 不看字面，看语义向量相似度。

计算步骤：

- Step 1：BERT 编码参考答案和模型输出的每个 token
- Step 2：计算每对 token 的余弦相似度
- Step 3：
  - Precision：每个模型 token 找最相似的参考 token
  - Recall：每个参考 token 找最相似的模型 token
  - F1：综合两者

优势： "非常精彩"和"十分出色"→ BERTScore 高（语义相同），而 BLEU 几乎为 0（字面不同）。

局限： 依赖 BERT 的语义理解能力；仍然需要参考答案；不能评测创意、风格等高级属性。

第三代：LLM-as-Judge（强模型当评委，当前主流）

三种评判模式：

- **模式 A：Pointwise（单答案打分，1-10 分）**——简单，但分数标准不一致
- **模式 B：Pairwise（两答案对比，主流）**——给两个回答，Judge 选更好的，配合 Elo 系统得到全局排名 → Chatbot Arena 的核心设计
- **模式 C：Reference-guided（参考答案辅助）**——提供参考答案，Judge 判断模型回答是否一致，适合有明确正确答案的任务

LLM-as-Judge 的偏见问题与解法：

- **自我偏好（Self-preference）：** Judge 倾向于给自己风格的回答高分（GPT-4 judge → GPT-4 回答得高分）。解法：多个不同 Judge 取平均。
- **位置偏见（Position Bias）：** 倾向于认为第一个答案更好。解法：AB 和 BA 两次评判，结果一致才算。
- **啰嗦偏见（Length Bias）：** 倾向于认为更长的答案更好。解法：明确提示"不要因为长度而加分"。

三代指标完整对比

指标	核心思想	优点	缺点	适用场景
BLEU	n-gram 字面重叠（精确率）	快速可重复	不看语义	机器翻译（已过时）
ROUGE	n-gram 覆盖率（召回率）	快速可重复	不看语义	文本摘要
BERTScore	语义向量相似度	理解同义词	计算贵，仍需参考答案	语义相似任务
LLM-as-Judge	强模型判断	灵活接近人类	有偏见，成本高	开放生成（主流）
Chatbot Arena	真实用户投票 + Elo	最可信	慢，不可重复	综合能力排名

16.3 Benchmark 污染：检测、防治与动态设计

直接与间接污染检测

直接污染检测：

- **n-gram 重叠检测：** 计算训练数据和测试题的 n-gram 重叠率，重叠率 > 阈值 → 疑似污染
- **Min-K% Prob 方法：** 对测试题每个 token 计算模型给出的概率，概率异常高 → 模型"见过"这道题（原理：模型对训练数据的 token 给出更高概率）

间接污染检测：

- **语义相似度检测：** 用 embedding 计算测试题和训练数据的语义相似度，相似度 > 阈值 → 疑似间接污染
- **难度一致性检测：** 模型在某 benchmark 上的表现远超其在等难度其他任务上的表现 → 疑似污染

自动化检测组合：

- n-gram overlap
- embedding similarity
- MinHash / LSH
- watermark / hash
- retrieval guard

> 🔍 **重点**：字符串不重复，不等于没有污染。

RL 阶段污染（2025 年新问题）

问题： GRPO 训练时用了公开 benchmark 题作为训练题 → 直接在评测集上 RL 优化 → 比数据污染更严重（直接优化目标）。

检测： 对比模型在该 benchmark 和等难度新题上的表现，如果 benchmark 远高于新题 → RL 污染。

解法：

- RL 训练时专门使用私有题库
- 不用任何公开 benchmark 作训练题
- 评测用完全隔离的私有测试集

> 💬 **面试话术**
>
> 数据污染是被动的（训练集里恰好有测试题），RL 污染是主动的（直接在测试集上做 RL 优化，梯度明确指向提升该 benchmark 的分数）。后者等于模型专门为这个 benchmark 做了"针对性训练"，分数虚高更严重，也更难被发现。

动态 / Live Benchmark 设计

静态 benchmark 最大的问题是很快会被训练、蒸馏、甚至 RL 过程污染。

更好的 Live Bench 设计：

- 定期注入新题
- 控制题目来源时间戳
- 保留一部分历史集做可比性
- 轮转一部分题目做 freshness

Train / Eval / Holdout / Rotating Canary 隔离流程

更可信的隔离方式是分五层：

1. **train** — 训练数据
2. **offline eval** — 离线评测集
3. **private holdout** — 私有保留集
4. **rotating canary set** — 轮换金丝雀集
5. **online shadow / A-B** — 线上影子流量和 A/B

Rotating canary 的价值： 保持新鲜度；避免模型和团队都"背熟了那一小批 canary"。

16.4 Judge 系统：Prompt 设计、偏差消除与排名体系

Judge Prompt 设计

一个可用的 judge prompt 至少要明确：

- 任务目标
- 比较维度
- tie 规则
- 长度偏差约束
- 输出 schema

> 🔍 **为什么这很重要：** judge prompt 本身就是评测标准的一部分；prompt 写歪，评测就歪。

Judge Bias 去偏方法

最常见偏差：

- position bias（位置偏见）
- length bias（长度偏见）
- self-preference（自我偏好）

去偏方法：

- swap position（交换位置，做两次评判）
- tie policy（两次结果不一致则算平局）
- multi-judge（多个不同 Judge 取平均）
- blind style normalization（风格归一化）

Elo / Bradley-Terry / Arena Ranking

Bradley-Terry 模型： 把 pairwise preference 转成胜率概率建模。

Elo 系统： 把多轮对战结果累计成全局评分。每次对战：赢了得分上升（赢强者得更多分），输了得分下降（输给弱者失更多分）。最终 Elo 分数反映模型的真实综合能力，相比简单统计胜率，Elo 考虑了对手强度，更公平。

> **一句话**：Arena 排名不是简单统计胜率，而是 pairwise ranking 系统。

16.5 工业级评测体系与发布链路

四层完整评测体系

第一层：训练中实时监控

- 每隔几百步自动运行私有题库
- 防止 RL 阶段污染公开 benchmark
- 目的：早发现训练问题，及时干预
- 不用于最终发布决策

第二层：训练后完整评测

公开 benchmark（参考基线）：MMLU / MATH / AIME / LiveCodeBench / MT-Bench

私有 benchmark（质量门控）：

- 业务场景题库（每季度更新 30%）
- Agent 任务库（τ^2-Bench 风格内部版）
- 多模态理解题库

发布门控（必须全部满足）：

- 私有综合分 > 上一版本 +3%
- Agent TCR > 上一版本 +5%

- 幻觉率 < 上一版本 -10%

第三层：人工盲测

- 标注员不知道哪个是新模型，同一问题新旧模型各回答一次
- 统计 Win Rate：> 60% → 可以发布；55%～60% → 需进一步分析；< 55% → 不发布
- 覆盖场景：日常对话 30% / 专业问答 30% / Agent 任务 30% / 创意写作 10%

第四层：线上 A/B 测试

- 5% 流量 → 新模型，95% → 旧模型，用户不知道用的是哪个版本
- 监控行为数据（不是让用户主动投票）：
  - 对话轮数（更多 → 更喜欢）
  - 会话长度（更长 → 更有价值）
  - 次日留存率（更高 → 满意度高）
  - 显式差评率（点踩比例）
- 全量发布条件：关键指标提升 > 阈值 + 无严重安全问题 + 线上延迟不超标

线上指标体系与冲突处理

模型上线至少要看的指标：

win-rate / complaint/unsafe rate / tool fail / latency / cost / retention/revisit

最常见指标冲突：

- 正确率上升，但投诉上升
- tool success 上升，但成本暴涨
- win-rate 上升，但延迟超 SLA

一个现实原则： 发布决策要先看 guardrail，再看收益指标。

SRM（Sample Ratio Mismatch）： A/B 里样本分配比例明显偏离预期，一旦出现，实验本身就不再可信。对应英文检索词：`sample ratio mismatch`。有时候不是模型不好，而是实验设计坏了。

发布链路：Shadow　Canary　Partial　Full　Rollback

一条更稳妥的发布链路：

1. shadow（影子流量）
2. canary（金丝雀）

3. partial rollout（部分发布）
4. full rollout（全量发布）
5. auto rollback（自动回滚）

为什么不能跳步： 离线过线不等于线上稳定。

发布 Guardrail 至少包括：

latency / safety/refusal / hallucination / tool error / complaint / cost

---

工业补充：2025-2026 评测体系演进

> 🏭 **工业补充（2025-2026）**
>
> 评测体系正在从"静态 benchmark 排名"走向"离线 + 在线 + 新鲜度 + 代理任务"四位一体。
>
> - **过去最关心的是分数高不高；**
> - **现在更关心的是这四件事能不能同时成立：**
> - **可比性：** 公开 benchmark 上能横向比较；
> - **新鲜度：** 题目不能被训练集和 RL 流程污染；
> - **任务真实性：** 要有 agent、tool-use、长视频、多轮任务这类更接近真实工作的评测；
> - **发布可靠性：** 离线评测结果能否在 shadow / canary / A/B 中延续。
>
> Qwen 在 **Qwen2.5-Math-PRM** 官方博客里专门强调了 **ProcessBench** 对"步骤级错误识别"的价值，说明 response-level BoN 或最终正确率已经不足以完整描述 reasoning quality。
>
> **资深工程师视角：** 评测不是一个排行榜问题，而是一个**发布门控问题**——benchmark 决定你能不能比较；private / live eval 决定你能不能相信；online A/B 决定你能不能上线。
>
> 2025-2026 的评测体系，目标不是找"榜单冠军"，而是找"在真实流量里最稳的版本"。

---

17. 面试题库

17.1 综合系统设计题

豆包 2.0 完整 Post-Training 方案

目标： 数学推理、代码、多模态、对话四个维度全面超越 GPT-4o

- **阶段 0：基线建立**——4 个专家模型各自在私有测试集上打基线分

- **阶段 1：TIES-Merging**——融合 4 个专家的 Task Vector，benchmark 验证融合效果，如果某个专家引入负干涉，调低其 λ
- **阶段 2：Post-Merge SFT**——少量数据缝合指令格式（几百步），防止融合破坏对话能力
- **阶段 3：GRPO（推理激活）**——数学 + 代码的可验证奖励，课程学习控制难度，避免梯度真空
- **阶段 4：Online DPO + RLAIF**——RLAIF 生产偏好数据（成本控制），人工复核抽样（质量兜底），Online 迭代防分布偏移
- **阶段 5：部署**——云端：MoE 路由，专家各司其职；边缘：保留 TIES-Merged 单模型
- **阶段 6：三层评测体系**
  - Layer 1：公开 benchmark（MATH/HumanEval/MMLU）← 可比性
  - Layer 2：私有测试集（防 Benchmark Contamination）← 真实泛化
  - Layer 3：盲测 Win Rate vs GPT-4o ← 真实体验

> 🔍 **Benchmark Contamination（基准污染）**
>
> GPT-4o 训练集可能包含公开 benchmark，你的模型也可能过拟合了这些 benchmark → 数字好看，真实体验未必好 → 必须用从未公开的私有测试集做最终裁判。

类 R1 推理模型完整训练流程

- **阶段 1：Cold-start SFT**（几千条 CoT 数据）——目的：植入思维链格式，解决 GRPO 的冷启动问题
- **阶段 2：GRPO 推理训练**——奖励：数学/代码可验证结果；技巧：课程学习，动态控制难度，避免梯度真空
- **阶段 3：Rejection Sampling SFT**——用当前模型采样，只保留答对的（含思维链），用"更强的自己"的最优解再训练
- **阶段 4：GRPO + 偏好信号融合（并行，非串行）**——GRPO 保持推理能力，偏好信号负责安全对齐，注意 Alignment Tax 问题
- **阶段 5：Online Iterative DPO**——持续采样新数据，保持数据分布新鲜

文档问答任务 QLoRA 方案

```
# 场景：72B模型，8张A100，10万条文档问答数据

# 1. 模型加载：QLoRA
model = AutoModelForCausalLM.from_pretrained(
    "Qwen2.5-72B",
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.bfloat16,
```

```
)

# 2. LoRA 配置
lora_config = LoraConfig(
    r=16,              # 文档理解任务，r稍大
    lora_alpha=32,
    target_modules=["q_proj", "v_proj", "k_proj", "o_proj"],
    lora_dropout=0.05,
)

# 3. 数据处理（注意：SYSTEM 必须保留在 input_ids 里！）
def process_doc_qa(sample):
    messages = [
        {"role": "system",    "content": "你是文档问答助手"},  # ✅ 保留上下文
        {"role": "user",      "content": f"文档：{sample['doc']}\n问题：
{sample['q']}"},
        {"role": "assistant", "content": sample['answer']},
    ]
    input_ids = tokenize(messages)
    labels = mask_non_assistant(input_ids)  # SYSTEM和USER设为-100
    return input_ids, labels

# 4. 训练配置
training_args = TrainingArguments(
    num_train_epochs=2,                # 不超过2，防格式过拟合
    per_device_train_batch_size=4,
    gradient_accumulation_steps=8,     # 模拟大batch
    evaluation_strategy="steps",
    eval_steps=500,
    load_best_model_at_end=True,       # 自动选最优checkpoint
)
```

17.2 面试题库全集

本节整合所有高频 Q&A，按主题分类，覆盖数据/SFT/Reward、RLHF/DPO/GRPO/推理RL、系统/并行/推理优化、Agent/多模态/系统设计、评测与项目表达五大主题。

数据 / SFT / Reward 类

> ❓ **Q1：SFT 为什么只对 assistant token 算 loss？多轮时所有 ASST 都要开放吗？**

要点：目标是学"如何回答"不是"如何提问"；所有 assistant turn 都开放，否则浪费监督信号；tool message/function result 按训练目标区分——学工具调用格式时 tool_call 部分开放，tool_result 是环境返回通常 mask 掉。

❓ Q2：工业级 SFT 数据工程怎么设计？

要点：数据来源（人工/合成/蒸馏/工具轨迹）→ 去重（MinHash/LSH）→ 质量过滤（perplexity/分类器/规则）→ 质量打分（LLM-as-Judge）→ 任务/长度 mix 配方（温度采样）→ sequence packing → train/eval split（防污染分割）。

主要风险：format overfit（模板化）、长样本主导（Length Bias）、template leakage（泄露 chat template）、benchmark contamination（评测集污染）。

❓ Q3：什么是 PSFT？为什么值得在面试里提？

要点：把 SFT 看作 constant-advantage 的 policy optimization，在监督阶段加入近端约束（KL penalty），限制 policy drift、降低泛化退化和 entropy collapse。

$$\mathcal{L}_{\text{PSFT}} = \mathcal{L}_{\text{SFT}} + \alpha \cdot \text{KL}[\pi_\theta | \pi_{\text{ref}}]$$

面试价值：说明"后训练不是从 RL 才开始管 policy drift——从 SFT 阶段就应该控制，每个阶段都需要自己的锚点"，展示你对 SFT 的深度理解。

❓ Q4：RM 为什么用 pairwise ranking 而不直接打分？

要点：绝对分数跨标注员不稳定（我的 8 分 ≠ 你的 8 分）；相对偏好一致性更高（A 比 B 好）；基于 Bradley-Terry 模型，数学严谨；可扩展到 listwise/Plackett-Luce 排名。

❓ Q5：RM、ORM、PRM、Verifier、Judge 怎么区分？

- **RM**：偏好/质量打分（通用），神经网络，输出标量分数，训练用 BT loss
- **ORM**：Outcome RM，只看最终结果，verifiable 任务首选，零 hack 风险
- **PRM**：Process RM，看中间步骤，数据贵，适合搜索/训练密集信号
- **Verifier**：规则/程序，非神经网络，确定性，代码执行器，最客观
- **Judge**：LLM-as-Judge，最灵活，有偏见，开放任务必用

选型原则：任务越可验证 → 优先 Verifier/ORM；任务越开放 → 只能用 Judge/Rubric/Pairwise。

❓ Q6：开放式任务 vs 可验证任务的奖励设计本质区别？

- **可验证：** rule checker/execution/matching，客观、便宜、可扩展，但 reward sparsity
- **开放式：** pairwise/rubric/LLM-as-judge，覆盖更广，但 bias/hacking 风险

实践混合：verifier + pairwise + rubric + format + cost penalty；分层约束（先满足安全红线，再优化有用性）。

❓ Q7：合成数据（GPT-4 蒸馏）的深层隐患？

- Model Collapse（分布收窄退化）
- 格式过拟合/Sycophancy（模仿语气而非解决问题）
- 能力天花板（永远无法超过老师）

缓解：掺杂人工标注 + 高温度多样性蒸馏 + 语义去重。

❓ Q8：Length Bias 怎么处理？

- 数据端：长度分桶重采样，配平长短数据
- 算法端：Length-normalized DPO（除以 token 数）；SimPO 内置长度归一化；Dr.GRPO 移除 $1/|o|$ 归一化
- RM 端：加长度惩罚项，防止 verbosity bias

❓ Q9：Reward Hacking 怎么防？

奖励源多样化（不只用一个 RM）+ Verifier/规则约束 + 人工抽检 + Reference/KL 约束 + RM Ensemble（难以同时 hack 多个）+ Slice eval（分场景监控）+ RM 定期迭代（更新盲区）。

❓ Q10：Benchmark Contamination 怎么防？

- 训练前：MinHash/LSH 去重，n-gram 匹配
- 训练后：temporal split，fresh benchmark，canary
- 核心 benchmark 分 public/private
- 代码任务用 LiveCodeBench 类 freshness benchmark
- 不只看总分，看 slice 和 pairwise win rate
- RL 阶段：不用公开 benchmark 题作训练题

❓ Q：数据飞轮怎么设计？

种子数据（人工精标）→ 当前最优模型采样（多样化采样）→ Judge/Verifier 过滤（只保留高质量）→ 模型训练（SFT + RLHF）→ 评测验证（私有 benchmark + 盲测）→ 线上部署收集用户反馈 → 更新种子数据，循环。

飞轮加速的关键：Judge 质量 + 采样多样性 + 评测体系。

RLHF / DPO / GRPO / 推理 RL 类

❓ Q11：RLHF 完整流程？为什么 SFT 不够？

SFT → RM → PPO 三阶段。SFT 只提供正向反馈，RL 提供负反馈 + 探索；RL 可突破监督学习性能天花板；SFT 不能学会"什么回答比另一个更好"。

❓ Q12：PPO 完整目标函数？每项作用？

$$\mathcal{L} = \mathbb{E}\left[\min(r_t \cdot A_t,\ \text{clip}(r_t, 1-\varepsilon, 1+\varepsilon) \cdot A_t)\right] - c_v \mathcal{L}_{\text{value}} + c_e H - \beta \cdot \text{KL}$$

- **policy clip**：防更新过猛（核心），$c_v = 0.5$
- **value loss**：训练 Critic
- **entropy bonus**：防过早塌缩，保持多样性，$c_e \approx 0.01$
- **KL 惩罚**：约束不偏离 reference model，$\beta = 0.02 \sim 0.1$

❓ Q13：GAE 为什么重要?

bias-variance tradeoff：$\lambda = 1$ 接近 MC（无偏，高方差），$\lambda = 0$ 接近 TD(0)（有偏，低方差），$\lambda = 0.95$ 是实践中的最优点。

LLM 稀疏 reward 场景下 GAE 特别重要：中间步骤 reward=0，只有最后一步有 reward，GAE 把末尾奖励往前传播，让早期 token 也有梯度。没有 GAE → 早期 token 完全无法学习。

❓ Q14：DPO 怎么从 RLHF 推出来?

KL-regularized RL 最优策略：$\pi^* \propto \pi_{\text{ref}} \cdot \exp(r/\beta)$，反解 r，代入 BT 模型，配分函数 $Z(x)$ 在相减时消去，得到只含 π_θ 和 π_{ref} 的闭式 loss。关键：$Z(x)$ 消去是整个推导最精妙的地方。

❓ Q15：DPO 的 β 作用？核心缺陷?

β 控制对 reference 偏离强度：β 大 → 保守（靠近 ref），β 小 → 激进（远离 ref），实践 $\beta \in [0.05, 0.5]$。

核心缺陷：离线静态数据的 distribution shift——训练数据针对旧模型标注，新模型学到一半后数据分布就错了。解法：Online Iterative DPO（持续采样新数据）。

❓ Q16：GRPO 和 PPO 本质区别?

PPO 依赖 Critic 估 value → 4 个模型 → 显存大；GRPO 用组内平均 reward 当 baseline，省掉 Critic → 3 个模型。

GRPO 适合：verifiable reward 任务（数学/代码）；PPO 适合：通用 RLHF（需要 Critic 估长期价值）。GRPO 的代价：每轮需要采样 G 个回答（推理成本更高）。

❓ Q17：GRPO 为什么会梯度真空?

组内全对或全错 → advantage 全 0 → 无梯度信号。解法：课程学习 + 难度调度；Dynamic Sampling 过滤全对/全错（DAPO）；Value 网络估算基线（VAPO）；方差过滤（Dr.GRPO）。

❓ Q18：DAPO 对 GRPO 做了哪些改进？（字节最高频）

五个改进，必须全部说出：

1. **Clip-Higher**：不对称裁剪（$\varepsilon_{\text{high}} = 0.28$，$\varepsilon_{\text{low}} = 0.2$），好回答学更快，促进探索

2. **Dynamic Sampling**：过滤全对/全错 prompt，自动解决梯度真空
3. **Token-level Loss**：替代 Sample-level Loss，每 token 贡献相等，消除长度偏差
4. **去掉 KL 散度**：推理任务上 KL 约束阻碍推理涌现，让模型自由探索
5. **Overlong Reward Shaping**：超长回答 reward=0（不是 -1，更温和，避免过度惩罚长推理探索）

结果：Qwen2.5-32B 在 AIME 上从 30 提升到 50 分。

❓ Q19：GSPO 为什么比 GRPO 更适合 2025-2026?

Qwen 官方指出 GRPO 的问题：长训 instability（token 级 ratio 内部不一致）+ MoE 模型上 collapse（路由不稳定）。

GSPO 的解法：sequence-level ratio/clipping——一个回答只有一个 ratio，要么整体被强化，要么整体被抑制，语义上和"回答质量"一致。更稳定，更适合大规模训练和 MoE 模型，内置长度归一化，是 Qwen3 推理训练的核心算法。

❓ Q20：DeepSeek R1 核心创新?

GRPO 替代 PPO（省掉 Critic）；纯 RL 训练推理能力（R1-Zero 证明可行）；Cold-start SFT 设计（植入格式）；蒸馏小模型优于直接 RL 小模型；"Aha moment"：模型自发涌现回溯和验证行为。

❓ Q21：R1 为什么需要 Cold-start SFT?

R1-Zero（纯 RL，无 SFT）证明了推理能力可以涌现，但存在问题：可读性差、语言混杂（中英混用）、格式混乱。

Cold-start SFT 的作用：植入 `<think>...</think>` 格式（行为先验）；提供基础推理轨迹示例；缩小 RL 搜索范围；守住输出质量底线。没有 cold-start → GRPO 采样全是无思维链的输出 → 梯度真空。只需要几千条数据，不需要大量标注。

❓ Q22：RLVR 真的"学会推理"了吗？（谨慎回答）

- 至少显著提升 verifiable reasoning 表现，部分跨任务泛化
- 但"通用推理能力"需要区分：task transfer（跨任务迁移）、faithfulness（思维链是否真实反映推理过程）、search vs compression（是在搜索还是在压缩记忆）
- Base model 的 pretrain quality 决定 RL 上限

回答框架："至少在 verifiable 任务上显著提升；base model 的 pretrain 质量决定 RL 上限；通用推理能力的泛化需要更多 held-out 验证。"

❓ Q23：Overthinking 是什么？怎么控制?

定义：模型生成过长的思维链，不比短链效果好，"表演思考"：长度增加，准确率不提升，Token Efficiency = 正确率提升 / token 数 ← 极低。

训练期控制： 长度惩罚（reward -= α × len）；Format reward（限制思维链格式）；Budget-aware reward（按难度设 token 预算）。

推理期控制： Max token 限制；Difficulty-aware allocation（难题分大预算）；Early stop（检测循环推理）；PRM 认为足够好时停止。

❓ Q24：Token-level reward 和 Sequence-level reward 区别?

- Sequence-level（GRPO）：每个 token 相同 advantage
- Token-level（DAPO）：引入 token 级 loss，长度归一化
- Step-level（VAPO + GAE）：精细的步骤级信号
- PRM：提供 step-level 密集反馈，最精细

越细粒度 → 梯度信号越精准，但标注成本越高。

❓ Q25：DPO 变体（ORPO/KTO/SimPO/Online DPO）分别解决什么?

- **ORPO**：去掉 ref model，SFT 和对齐合并（显存省一半）
- **KTO**：不需要配对数据，适合 binary 反馈（标注更简单）
- **SimPO**：reference-free + 内置长度归一化
- **Online DPO**：缓解 distribution shift（持续采样新数据）

决策表：只有点赞数据（binary）→ KTO；没有参考模型/省显存 → ORPO 或 SimPO；偏好对差距小/训练不稳 → IPO；分布偏移严重 → Online DPO；通用场景 → 标准 DPO。

❓ Q26：PRM 训练数据如何构造?

Monte Carlo Rollout：从 Step k 采样 N 条路径，统计最终正确率作为该步 score：$\text{score}(\text{Step}_k) = \text{正确路径数}/N$。多模型集成采样降低系统性偏差（Math-Shepherd）。自动化无需人工标注每一步，前提是任务有客观的最终验证标准。

❓ Q：GRPO 中 G 取多少合适?

G 通常取 8～16。太小：小组平均分不稳定，基线估计不准；太大：计算开销大，收益递减。实践中 DeepSeek-R1 用 G=8。

❓ Q：为什么 R1 训练不能先 GRPO 再 DPO?

Alignment Tax 问题。DPO 优化的是人类偏好（主观），人类标注员偏好简洁的回答，DPO 会让模型缩短思维链。应该并行而非串行：GRPO + 偏好信号同时训练，加权融合。

❓ Q：RLHF 训练中出现 reward collapse 怎么诊断和处理?

诊断： reward 短暂上升后骤降；人工抽查高 reward 回答发现质量差（例如空洞但礼貌）→ Reward Hacking 的信号。

处理： 增加 RM 多样性（集成多个 RM）；加规则约束（安全/格式硬约束）；降低学习率，增大 KL 惩罚；RM 迭代更新（用当前模型的输出重新标注）；人工抽检 + 黑名单机制。

❓ Q：Online DPO 相比 Offline DPO 的代价是什么？

需要额外的 RM（或人类评估）来对新采样的数据进行偏好标注。如果用 RM 标注，引入了 RM 误差；如果用人类标注，成本高、迭代慢。工业界通常用 RM 自动标注 + 定期人工复核。

❓ Q：Per-sample Loss Normalization 和普通 Loss 有什么区别？

普通 mean reduction 把 batch 内所有 token 等权平均，导致长样本（高 token 数）主导梯度。Per-sample normalization 先在样本内归一化，再在 batch 间平均，每个样本权重相等，无论长短。

系统 / 并行 / 推理优化类

❓ Q27：ZeRO-1/2/3 区别？

- **Stage 1**：切 optimizer states（m 和 v）→ 显存节省 ~4x，通信量和 DP 一样
- **Stage 2**：切 optimizer states + gradients → 显存节省 ~8x，通信量和 DP 一样
- **Stage 3**：切 optimizer states + gradients + parameters → 显存节省 ~64x，前向需要 All-Gather 参数，通信量最大

❓ Q28：张量并行为什么放节点内，流水线并行跨节点？

- **张量并行（TP）**：层内并行，切参数矩阵，需要 All-Reduce（高通信量），必须用 NVLink（~900GB/s）→ 只能在同一机器的 GPU 间使用
- **流水线并行（PP）**：层间并行，切层，只需相邻 stage 点对点通信（通信量小），可以容忍跨节点的 InfiniBand 延迟 → 可以跨机器使用

配置原则：TP 放节点内（NVLink），PP 跨节点（InfiniBand）。

❓ Q29：72B 模型显存怎么估？

完整公式：

- 参数（BF16）：$70 \times 2 = 140\text{GB}$
- 梯度（BF16）：$70 \times 2 = 140\text{GB}$
- Adam m（FP32）：$70 \times 4 = 280\text{GB}$
- Adam v（FP32）：$70 \times 4 = 280\text{GB}$
- 小计：840GB

简化记忆：参数量（B）× 12 = 训练显存（GB）→ 70B × 12 = 840GB，需要约 11 张 A100，实际加激活需 16-32 张。激活值取决于 batch size 和 seq*len，另算；KV Cache（推理）：$2 \times L \times B \times S \times n*{\text{kv}} \times d_{\text{head}} \times 2$。

❓ Q30：BF16 和 FP16 区别?

- **BF16**：8 位指数（与 FP32 相同动态范围）+ 7 位尾数（精度低），几乎不会下溢 → 不需要 Loss Scaling → LLM 训练默认选择
- **FP16**：5 位指数（动态范围小）+ 10 位尾数（精度高），容易下溢 → 必须使用 Loss Scaling → 推理时可以用，训练时有风险

❓ Q31：FlashAttention 为什么快?

核心不是数学变了，而是 IO-aware。标准 Attention 的瓶颈：HBM 读写速度（2TB/s）vs SRAM（19TB/s），每一步都要把 N×N 的注意力矩阵写回 HBM。

Flash Attention：Tiling + Kernel Fusion，在 SRAM 里完成所有计算，最后才写回 HBM，节省 10x 以上的 HBM 读写，速度提升 2~10x。

FA3（H100 专项）：FP8 计算 + TMA 异步 + 交错编排，H100 利用率 35% → 75%。

❓ Q32：vLLM PagedAttention 原理?

传统 KV Cache：预分配最大可能长度的显存 → 大量内存碎片（浪费 60-80%）。

PagedAttention（借鉴 OS 虚拟内存分页）：KV Cache 分成固定大小的 block（16 token），Block Table 记录逻辑块 → 物理块映射，支持非连续内存分配，浪费率降至 <4%，配合 Continuous Batching，吞吐提升 5-10x。

❓ Q33：Speculative Decoding 原理？为什么不影响输出质量?

小 Draft 模型生成 K 个候选 token（q 分布），大 Target 模型一次 forward 并行验证 K 个位置（p 分布）。

接受/拒绝机制：$\text{accept} = \min(1, p(x)/q(x))$，拒绝后从修正分布采样：$p_{\text{corrected}}(x) = \text{normalize}(\max(0, p(x) - q(x)))$。

数学证明：整个过程等价于 Target 模型自回归解码，完全不影响输出质量（零精度损失），速度提升：接受率 80% 时约 2-3x。

❓ Q34：KV Cache 为什么重要？PD 分离为什么热?

KV Cache：decode 阶段是 memory-bound，不缓存 KV → 每步重算历史 token → 极度浪费。

PD 分离（热门原因）：Prefill compute-bound（并行处理整个 prompt）vs Decode memory-bound（每步只生成 1 token），混合在一起互相干扰，各自效率都低。分离后：

Prefill 用算力强的 GPU，Decode 用带宽大的 GPU → 成本降低 + 效率提升。

❓ **Q35：MoE Routing Collapse 怎么防?**

Router 偶然偏向少数专家 → 正反馈 → 大量专家闲置。

- **Auxiliary Load Balancing Loss**：$\mathcal{L}_{\text{balance}} = N \times \sum(f_i \times P_i)$，强制均匀分配，但干扰主 Loss
- **DeepSeek-V3 Auxiliary-Loss-Free**：per-expert 偏置项动态调整，不参与梯度，不干扰语言学习，更优雅，效果更好

❓ **Q36：FSDP vs ZeRO vs Megatron 怎么选?**

不是三选一，而是组合：TP + PP + DP + CP/EP + ZeRO/FSDP。选型取决于：模型规模（参数量和层数）、序列长度（是否需要 Sequence Parallel）、是否 MoE（需要 Expert Parallel）、集群互联（NVLink vs InfiniBand）、工程栈（PyTorch 生态 vs Megatron 生态）。

一般原则：TP 放节点内（NVLink），PP 跨节点，ZeRO-3/FSDP 处理 optimizer states。

❓ **Q：SFT 训练多少个 epoch 合适?**

经验规则：数据量越大 epoch 越少。10 万条以上的数据，通常 1~2 个 epoch 足够。超过 2 个 epoch 容易 Format Overfitting（格式过拟合）。监控 benchmark 和 format diversity，而不只看 val loss。

❓ **Q：LoRA 的 r 怎么选?**

任务越复杂、领域差异越大，r 越大。一般规律：对话/通用任务 r=8，代码/文档理解 r=16~32，领域高度特殊 r=64。实践中做 ablation study 最可靠。

❓ **Q：什么情况下 LoRA 权重要合并，什么时候不合并?**

单任务追求推理速度 → 合并（zero overhead）；多任务共享基础模型 → 不合并（Multi-LoRA Switch，节省显存）；A/B 测试不同版本 → 不合并（灵活切换）。

Agent / 多模态 / 系统设计类

❓ **Q37：Function Calling 底层到底是什么?**

模型本质还是在生成 token，只是输出的 token 碰巧是 JSON 格式。

完整流程：模型输出符合 schema 的结构化文本 → parser/orchestrator 变成真实 API 调用 → 结果塞回上下文（tool_result）→ 模型继续生成（可能继续调用工具）→ error handling（工具返回错误时如何处理）。

本质：模型还是在生成 token，只是 token 符合 JSON schema。MCP 统一了这个接口格式，不同工具用同一套调用方式。训练需要：专门构造 function calling SFT 数据，包含工具定义 → 调用 → 结果 → 继续的完整轨迹。

❓ Q38：怎么设计 Agent 的 reward?

分层设计：

- 层次 1 格式依从：工具调用 JSON schema 是否正确 → 硬性要求
- 层次 2 执行连通：工具是否真的执行成功 → 基础要求
- 层次 3 正确性：最终答案是否正确 → 最重要（权重最大）
- 层次 4 效率：是否用了最少的步骤 → OTC-GRPO

Credit Assignment：蒙特卡洛 Rollout 估算每步的贡献，
$\text{score}(\text{Step}_k) = \text{从该步 rollout 后的成功率}$。

❓ Q39：怎么评估 Agent 而非聊天机器人?

核心指标：Task Success Rate（最重要）、Tool Call Accuracy、Steps to Completion（越少越好）、Recovery Ability、Planning Quality（LLM-as-Judge）。

评测框架：τ^2-Bench（电商/航空客服真实场景）、AgentBench（多领域综合）、SWE-Bench（代码修复）。

评测方式：离线 benchmark + sandbox 环境 + 限量 online canary。

❓ Q40：设计 RLHF 训练系统。

数据侧：prompt queue + human preference 标注；模型侧：SFT → RM → PPO/DPO 三阶段。

分布式架构：Actor（训练中）+ Reference（冻结）+ RM（冻结）+ Critic（训练中），4 个模型显存调度（通常不能同时都放在 GPU）。解法：异步 RL（Actor 和 RM 分离部署）。

评估指标：自动（reward 曲线、KL 散度、clipfrac、entropy）+ 人工（Win Rate 新旧模型盲测）。迭代策略：在线采样 → 过滤 → 训练 → 评测 → 循环（数据飞轮闭环）。

❓ Q41：设计 Reasoning Model 训练 Pipeline（R1/o1 风格）。

1. Cold-start SFT（植入 CoT 格式）
2. 构建 verifier/reward stack（数学/代码客观验证器）
3. On-policy reasoning RL（GRPO/DAPO/REINFORCE++），动态难度采样，避免梯度真空
4. Online eval + private set + shadow traffic，持续监控，防止 benchmark 污染
5. Token cost/overthinking 控制（Length penalty + budget-aware reward）
6. 数据飞轮：用更强的模型生成更好的 CoT 数据 → 重新 SFT → 继续 RL

❓ Q42：设计 Agent 场景奖励系统（多步 Tool-use）。

核心难题：Credit Assignment（哪一步导致了最终失败？）。

解法：蒙特卡洛步骤估值，
\text{step_value} = \text{success_rate}(\text{after_step}_k) - \text{success_rate}(\text{be
，只有真正推进任务的步骤才有正 advantage。

分层奖励：格式依从 → 执行连通 → 正确性 → 效率；OTC-GRPO 效率惩罚：reward -= λ × extra_tool_calls。

❓ Q43：模型线上效果退化怎么诊断?

- **信号 1 Reward Hacking**：reward 指标很高，但用户满意度下降 → 人工抽检 reward 高分样本，发现异常模式
- **信号 2 RM 过度优化**：模型输出越来越模式化、奉承式 → RM 定期迭代更新
- **信号 3 特定能力退化**：Slice eval 定位具体场景

Evaluation Pipeline：自动 eval（每轮训练自动跑）+ 人工抽检（每周）+ 线上 A/B（新旧模型对比真实用户行为）。

❓ Q44：长文本 RLHF 面临什么瓶颈?

- **瓶颈 1 $O(N^2)$ 显存爆炸**：4 个模型 × 长上下文 × Attention 矩阵。解法：Sequence Parallel + Ring Attention
- **瓶颈 2 RM "Lost in the Middle"**：长文本的中间部分 RM 注意力不足，评估不准。解法：Chunk 级 RM 滑窗评估，把长文分成 chunk 分别评估，加权合并
- **瓶颈 3 奖励稀疏**：长文本的奖励信号更稀疏。解法：长短混合穿插训练，step-level PRM 提供密集反馈

❓ Q45：设计医疗垂类大模型 Post-Training 流程。

1. 能力目标定义（诊断/咨询/文书/检索）
2. 数据准备（医生标注高质量问答 + 基于医疗知识库合成 + 医疗规则校验）
3. SFT（医疗指令微调）
4. 安全偏好优化（绝对不能给错误诊断）
5. Agent/tool 优化（检索 RAG、药品数据库调用）
6. 评测（correctness/safety/hallucination/citation/tool success）
7. 上线灰度（先专科医生内测，再扩大）

❓ Q46：Llama 4 后训练新思路?

Lightweight SFT → Online RL → Lightweight DPO。

核心发现：重 SFT 和 DPO 过度约束限制了 RL 的探索空间，模型被 SFT 锁定了，RL 很难突破。

解法：SFT 用少量高质量数据（而不是大量数据），给 RL 留足探索空间；DPO 只做轻量微调（不要过拟合偏好数据）；去除 >50% easy 数据（简单数据让模型偷懒）。

❓ Q47：Qwen3 训练范式独特之处？

- Thinking/Non-thinking 双模式统一（一个模型两种模式，用 special token 切换）
- Thinking Budget 可控推理深度（用户指定最大思考 token 数）
- GRPO 进行 RL 训练（可验证任务）
- Safety RL（hybrid reward = 安全 + 有用性，避免 Alignment Tax）
- REINFORCE++ 而非 GRPO（更简单、更省、在大规模训练上更稳定）

面试亮点："Qwen3 解决了'思考模型贵、普通模型弱'的两难困境，用统一模型满足不同场景需求。"

❓ Q48：字节/腾讯/阿里面试风格差异？

- **字节（豆包/Seed）**：先讲场景 → 再讲算法 → 最后讲系统和指标；会追问：数据飞轮、verifier 设计、token 成本控制；关键词：DAPO、veRL、Agent/MCP、评测体系
- **腾讯（混元/元宝）**：先给公式定义 → 补并行/显存 → 补推理性能；会追问：Loss 推导、并行配置、显存计算；关键词：3D 并行、ZeRO/FSDP、FlashAttention、MoE
- **阿里（Qwen/通义）**：先给统一抽象 → 展开到 agent/thinking/framework；会追问：unified thinking/non-thinking、数据飞轮、评测工具；关键词：Qwen3、GSPO、ROLL、Qwen-Agent/MCP

❓ Q49：GRPO PyTorch 中 KL 惩罚怎么设计？

不在 reward 中直接减 KL（会破坏 advantage 归一化的统计意义）。

正确做法：

```
# Step1: 用纯净 reward 完成组内归一化
advantages = (rewards - mean) / std
# Step2: 计算 surrogate policy loss
L_policy = min(ratio × A, clip(ratio) × A)
# Step3: token 级 KL 近似惩罚作为独立附加损失
kl_penalty = exp(log_p - ref_log_p) - (log_p - ref_log_p) - 1
L_kl = β × kl_penalty.mean()
# Step4: 总损失
L = L_policy + L_kl
```

这样 KL 不污染 advantage 的统计性质。

❓ Q50：如果面试官问"你笔记最大短板是什么"？

> 标准回答："主线算法（SFT/PPO/DPO/GRPO）比较完整，我正在补三块工业能力：第一块分布式训练与推理系统（ZeRO/FSDP/Megatron/vLLM 的工程细节）；第二块评测与防污染体系（私有 benchmark 建设、数据飞轮设计）；第三块 Agent/多模态落地（Tool-use RL、MCP 协议、VLM 对齐）。这三块是从'理解算法'到'设计系统'的关键跨越。"
>
> 不要说："基础不够牢"（太笼统）、"数学不好"（真实短板不要暴露）、"什么都不会"（过度谦虚）。

Gap Analysis 补充 Q&A

> ❓ **Q：Test-Time Compute 和训练时 Compute 如何权衡？**
>
> 对于推理密集型任务（数学/代码），增加推理时 compute 往往比等比例增加训练 compute 更高效。但对于知识密集型任务（事实问答），推理时多想帮助有限，还是需要更大的训练语料。

> ❓ **Q：MCTS 的 N（采样数）怎么设置？**
>
> 步骤级 MCTS 通常每步采样 4~8 个候选。太少：搜索不充分，容易陷入局部最优；太多：计算开销指数级增长，收益递减。实践中做 ablation study 确定，通常 4 就够。

> ❓ **Q：TIES-Merging 的 Trim 阈值怎么定？**
>
> 通常保留每个 Task Vector 中绝对值最大的 top-k% 参数（k=20~50）。太激进（k 太小）会丢失有效信号；太保守（k 太大）会引入太多噪声。同样通过验证集 ablation 确定。

> ❓ **Q：RLAIF 生成的偏好数据，如何检测是否有系统性偏差？**
>
> 三种方法：(1) 人工抽样复核，统计标注一致率；(2) 对比 RLAIF 数据和少量人类标注数据的分布差异；(3) 在有标准答案的任务上检验 RLAIF 的判断准确率。

> ❓ **Q：Model Merging 和继续微调相比有什么本质优势？**
>
> 继续微调（Continual Learning）有灾难性遗忘问题——学新任务会遗忘旧任务。Model Merging 在参数空间直接叠加，理论上不存在遗忘。但 Merging 有干涉效应，继续微调有遗忘问题，实践中两者结合（Merge 后再轻量微调）效果最好。

> ❓ **Q：Tool-use RL 和普通 GRPO 最本质的区别是什么？**
>
> 奖励信号的来源不同。普通 GRPO 的奖励来自数学/代码的客观验证，是单步的。Tool-use RL 的奖励来自多步工具调用的最终结果，是序列级的，还涉及 Credit Assignment 问题。

> ❓ **Q：Multi-Agent RL 和 Single-Agent RL 最大的挑战是什么？**

> Non-stationarity（非平稳性）。多个 Agent 同时训练时，对每个 Agent 来说，其他 Agent 是"环境"的一部分，但这个"环境"在不断变化，破坏了 RL 的收敛性假设。解法是 CTDE（集中训练分散执行）或轮流训练。

> ❓ **Q：Memory RL 为什么比 Tool-use RL 训练成本高？**
>
> 奖励延迟更长。Tool-use RL 的奖励可以在单次任务结束时计算。Memory RL 的奖励需要等到"写入的记忆被用到"才能评估，可能跨越多个任务，需要更长的训练 episode。

> ❓ **Q：MCP 对 Agent 训练有什么实际影响？**
>
> 统一了工具调用的 Reward 设计。没有 MCP 时，100 个工具需要 100 种格式的训练数据；有了 MCP，Reward 信号统一，新工具零额外训练成本，大幅降低了 Tool-use RL 的数据需求。

面试加工：高频追问标准答法

> ❓ **Q：为什么 RL 阶段的污染比训练数据污染更严重？**
>
> 数据污染很多时候还是"被动见过题"；RL 污染则是"主动拿 benchmark 做优化目标"，梯度会直接把模型往这个测试集上推。所以它造成的分数虚高通常更严重，也更难发现。

> ❓ **Q：一个成熟团队怎么决定模型能不能上线？**
>
> 不是只看离线 benchmark 排名。至少要同时看：离线公开集 + 私有 / live eval + 安全评测 + shadow / canary / A/B。真正能上线的是"在真实流量里最稳的版本"，不是"榜单上最亮眼的一版"。

> ❓ **Q：为什么 `tool_result` 往往要 mask？**
>
> 因为 `tool_result` 是环境返回值，不是模型应该学会"生成"的内容。如果对它开 loss，模型很容易学会伪造环境结果，或者把外部工具返回当成自己语言建模的一部分。更合理的目标通常是：学会生成 `tool_call`，理解 `tool_result`，但不去复现 `tool_result`。

> ❓ **Q：chat template 和 tokenizer 最容易踩什么坑？**
>
> 最常见的坑是 special token 没真正注册、BOS/EOS 处理不一致、HF 和 serving 侧模板不一致，以及 assistant prefix 不一致。这些问题一旦存在，模型学到的其实是错误的角色边界和停止行为。所以 chat template 本质上不是 UI 层问题，而是训练分布定义问题。

> ❓ **Q：为什么 2025-2026 的 reward 不再是单一 RM？**
>
> 因为不同任务需要的监督来源不一样。开放任务更依赖 judge / rubric / pairwise RM，可验证任务更依赖 verifier / executor，长链推理和 agent 又需要 ORM / PRM 做不同粒度的评估。所以今天的 reward 更像一个组合栈，而不是一张单独的 RM 网络。

❓ Q：ORM 和 PRM 应该怎么组合？

ORM 更适合做结果锚点，稳、便宜、服务化简单；PRM 更适合做过程信用分配，尤其在长链推理和 agent 任务里更有价值；真正工业化的做法通常是 verifier / ORM / PRM 组合，而不是二选一。

❓ Q：rollout 为什么经常比 trainer 更贵？

因为 rollout 本质上是在做大规模高吞吐生成，尤其在长 CoT、tool-use、multi-turn agent 场景里，采样成本会占到总训练时间的大头。trainer 虽然做反向传播，但不一定比海量 rollout 更耗 wall-clock。所以很多 RL 平台真正先优化的是 rollout engine，而不是 loss 本身。

❓ Q：stale sample 为什么会把 RL 训练弄飘？

因为这些样本是旧策略生成的，但你在用新策略更新。一旦策略漂移太多，importance ratio、KL、advantage 统计都会失真。所以 async RL 能提吞吐，但必须控制 freshness，否则就会"快但飘"。

❓ Q：OpenRLHF 和 veRL 分别更像什么？

OpenRLHF 更像基于 Ray + vLLM 的高性能 RLHF 训练框架，强调 actor/reward/reference/critic 编排和 async agentic RL。veRL 更像统一的 RL 平台，强调 HybridFlow、FSDP/FSDP2/Megatron 后端、vLLM/SGLang rollout，以及大模型级别 placement 和 resource mapping。两者都不只是"能跑 PPO"，而是在解决平台化训练问题。

❓ Q：DPO 为什么训练稳，但不一定适合 reasoning / agent 主线？

因为 DPO 本质上是离线偏好对齐，适合稳定地校正风格和行为边界；但 reasoning、tool-use、trajectory 这类任务往往需要在线探索、环境反馈和可验证奖励；所以 DPO 很适合做对齐底座，但不一定能单独承担 reasoning / agent 的主训练目标。

❓ Q：dynamic beta 到底在缓解什么问题？

它在缓解"固定 beta 对所有样本一刀切"的问题。高置信样本可以放松约束，多学一点；模糊、异常、可能错标的样本应该更保守。所以 dynamic beta 的本质，是让不同质量的样本使用不同的分布偏移强度。

❓ Q：为什么 sequence-level 更新在 2025-2026 更重要？

因为在 long CoT 和 MoE 场景里，token-level ratio 噪声非常大，单个 token 的不稳定会被放大。sequence-level 更新更接近"整条回答质量"的语义，也更适合长链和多专家协同。GSPO 之所以重要，本质就在这里。

❓ Q：Qwen3 四阶段为什么不能少 fusion？

> 因为前两阶段主要把 think mode 拉出来，但还没有解决 think / no-think 的共存问题。如果没有 fusion，两个模式很容易相互污染，出现 reasoning leakage、风格不一致、甚至推理预算控制失灵。fusion 阶段本质上是在把探索出来的 reasoning 能力重新整合成统一可部署策略。

> ❓ **Q：Agent 训练为什么不是 function calling?**
>
> function calling 只回答"模型能不能生成一个结构化调用"；Agent 训练真正关心的是：什么时候调、调哪个、参数怎么填、结果怎么利用、失败怎么恢复、整条轨迹是否高效且安全。所以 Agent 问题的核心是长链决策和环境交互，不是 JSON 生成。

> ❓ **Q：tool safety 和 text safety 为什么不是一回事?**
>
> text safety 主要是防模型"说错话"；tool safety 要防模型"做错事"，包括高权限操作、destructive action、参数误填、prompt injection 经由工具结果传播等。所以 tool safety 一定要加上 sandbox、权限、审计和二次确认这类系统约束。

> ❓ **Q：vLLM 为什么不只是一个 kernel，SGLang 又什么时候更顺手?**
>
> vLLM 的核心价值是 `PagedAttention + Continuous Batching + Prefix Cache + Chunked Prefill` 组成的一整套高吞吐推理系统。SGLang 则更强调结构化生成和程序式控制流，适合复杂 agent / tool / structured generation 场景。纯 serving 吞吐优先时常看 vLLM，复杂 agent 编排时更容易想到 SGLang。

> ❓ **Q：DeepSeek-V3 为什么强调 auxiliary-loss-free?**
>
> 因为传统辅助损失会直接和语言建模目标拉扯，可能负载均衡了但模型性能掉了。auxiliary-loss-free 的关键是通过 router bias 或统计项动态调节，不把负载均衡目标直接写进主 loss，能把"负载均衡"和"语言学习"解耦，所以更适合大规模 MoE。

> ❓ **Q：长上下文和记忆不是一回事，边界在哪里?**
>
> 长上下文解决的是"当前窗口里能看多远"；记忆系统解决的是"跨会话、跨任务、长时间维度的信息保存与检索"。前者是上下文容量问题，后者是状态管理问题，不能混为一谈。

评测高频追问

> ❓ **Q：BLEU 分高的翻译一定更好吗?**
>
> 不是。BLEU 只看字面 n-gram 重叠，不看语义。"非常精彩"和"十分出色"BLEU 很低但语义完全相同。而且 BLEU 鼓励短回答（重叠率更高），会系统性地低估流畅的长翻译。现在机器翻译评测已经转向 BERTScore 和人工评测。

> ❓ **Q：LLM-as-Judge 的位置偏见怎么解决?**

> 做两次评判：第一次 A 在前 B 在后，第二次 B 在前 A 在后。两次结果一致（都选同一个）才认为有效；两次结果不同则认为平局（tie）。这样可以消除位置偏见的影响。

> ❓ **Q：私有 benchmark 为什么要每季度更新 30%？**
>
> 防止内部泄露和过拟合。如果题目长期不变，训练过程中可能通过反复评测逐渐"学到"这些题目。更新 30% 保持新鲜度，但保留 70% 维持历史可比性，让你能追踪版本间的能力变化。

> ❓ **Q：Chatbot Arena 的 Elo 系统是什么？**
>
> 来自国际象棋的评分系统。每次对战：赢了得分上升（赢强者得更多分），输了得分下降（输给弱者失更多分）。最终 Elo 分数反映模型的真实综合能力。相比简单统计胜率，Elo 考虑了对手强度，更公平。

17.3 手撕代码速查与显存手算

核心函数：至少要能手写到伪代码级

函数	关键词	面试重点
`pairwise_rm_loss`	`reward_chosen - reward_rejected`、`-log(sigmoid(...))`	为什么 BT 只关心相对分差，不关心 reward 绝对值
`ppo_clip_loss`	`ratio = exp(new_logp - old_logp)`、`min(surr1, surr2)`	为什么 clip 是"限制更新步长"，不是直接限制参数差
`compute_gae`	`delta_t = r_t + gamma * V_{t+1} - V_t`	GAE 是用指数衰减方式累计 TD error，不是简单 reward-to-go
`dpo_loss`	`beta * ((pi_c - ref_c) - (pi_r - ref_r))`	为什么它隐式对应一个最优 reward model
`grpo_loss`	group reward、group mean、normalized advantage	为什么它把 Critic 换成了组内基线
`lora_forward`	`W x + BAx`	为什么 `B=0` 初始化能保证训练开始时不破坏原模型输出
`kv_cache_estimator`	`2 * L * B * S * n_kv_heads * d_head * bytes`	为什么 GQA 会显著降低 KV cache

第一层：核心 Loss 函数

```python
def pairwise_rm_loss(reward_chosen, reward_rejected):
    margin = reward_chosen - reward_rejected
    return -torch.log(torch.sigmoid(margin)).mean()


def ppo_clip_loss(new_logp, old_logp, advantage, eps=0.2):
    ratio = torch.exp(new_logp - old_logp)
    surr1 = ratio * advantage
    surr2 = torch.clamp(ratio, 1 - eps, 1 + eps) * advantage
    return -torch.min(surr1, surr2).mean()


def dpo_loss(pi_c, pi_r, ref_c, ref_r, beta=0.1):
    logits = beta * ((pi_c - ref_c) - (pi_r - ref_r))
    return -torch.nn.functional.logsigmoid(logits).mean()
```

第二层：递推与线性代数模块

```python
def compute_gae(rewards, values, gamma=0.99, lam=0.95):
    advantages = torch.zeros_like(rewards)
    gae = 0.0
    for t in reversed(range(len(rewards))):
        next_value = values[t + 1] if t + 1 < len(values) else 0.0
        delta = rewards[t] + gamma * next_value - values[t]
        gae = delta + gamma * lam * gae
        advantages[t] = gae
    return advantages


def lora_forward(x, W, A, B, alpha, r):
    base = x @ W
    delta = (x @ A @ B) * (alpha / r)
    return base + delta
```

`compute_gae`：能体现你真的理解 advantage 怎么从 reward 和 value 递推出来；
`lora_forward`：能体现你不只是知道"低秩分解"四个字，而是知道它在 forward 里具体长什么样。

第三层：GRPO / DAPO 与 MHA

```
def grpo_loss(logp, old_logp, rewards, group_size, eps=0.2):
    rewards = rewards.view(-1, group_size)
    mean = rewards.mean(dim=1, keepdim=True)
    std = rewards.std(dim=1, keepdim=True) + 1e-6
    adv = ((rewards - mean) / std).reshape(-1)

    ratio = torch.exp(logp - old_logp)
    surr1 = ratio * adv
    surr2 = torch.clamp(ratio, 1 - eps, 1 + eps) * adv
    return -torch.min(surr1, surr2).mean()


def mha_forward(x, Wq, Wk, Wv, Wo, num_heads):
    B, T, D = x.shape
    H = num_heads
    Dh = D // H
    q = (x @ Wq).view(B, T, H, Dh).transpose(1, 2)
    k = (x @ Wk).view(B, T, H, Dh).transpose(1, 2)
    v = (x @ Wv).view(B, T, H, Dh).transpose(1, 2)
    att = (q @ k.transpose(-2, -1)) / (Dh ** 0.5)
    att = torch.softmax(att, dim=-1)
    out = (att @ v).transpose(1, 2).reshape(B, T, D)
    return out @ Wo
```

第四层：RoPE、KV Cache、GRPO+KL、DAPO

```
def rope_apply(x, cos, sin):
    x_even = x[..., 0::2]
    x_odd = x[..., 1::2]
    out_even = x_even * cos - x_odd * sin
    out_odd = x_even * sin + x_odd * cos
    out = torch.empty_like(x)
    out[..., 0::2] = out_even
    out[..., 1::2] = out_odd
    return out


def kv_cache_estimator(n_layers, batch_size, seq_len, n_kv_heads, d_head,
```

```
dtype_bytes=2):
    total = 2 * n_layers * batch_size * seq_len * n_kv_heads * d_head *
dtype_bytes
    return total / (1024 ** 3)


def grpo_loss(logp, old_logp, rewards, group_size, eps=0.2, kl_coef=0.0,
ref_logp=None):
    rewards = rewards.view(-1, group_size)
    mean = rewards.mean(dim=1, keepdim=True)
    std = rewards.std(dim=1, keepdim=True) + 1e-6
    adv = ((rewards - mean) / std).reshape(-1)

    ratio = torch.exp(logp - old_logp)
    surr1 = ratio * adv
    surr2 = torch.clamp(ratio, 1 - eps, 1 + eps) * adv
    loss = -torch.min(surr1, surr2).mean()

    if ref_logp is not None and kl_coef > 0:
        kl = torch.exp(logp - ref_logp) - (logp - ref_logp) - 1
        loss = loss + kl_coef * kl.mean()
    return loss


def dapo_token_loss(logp, old_logp, token_adv, eps_low=0.2, eps_high=0.28):
    ratio = torch.exp(logp - old_logp)
    clipped = torch.where(
        token_adv >= 0,
        torch.clamp(ratio, 1 - eps_low, 1 + eps_high),
        torch.clamp(ratio, 1 - eps_low, 1 + eps_low),
    )
    return -torch.min(ratio * token_adv, clipped * token_adv).mean()
```

`rope_apply`：能证明你真知道位置编码怎么落实到张量旋转；`kv_cache_estimator`：能让你在系统题里快速把"为什么 decode memory-bound"讲实；`grpo_loss` 说明你知道 group reward、组内 baseline、ratio、clip 和可选 KL 是怎么接起来的；`dapo_token_loss` 说明你知道 DAPO 的关键不是"另一个名字"，而是 token-level advantage 和不对称裁剪。

注意力 / 位置编码 / 推理系统

MHA 最小实现： `QKV projection → reshape heads → score → mask → softmax → weighted V → merge heads`，面试重点：维度变化必须说清楚。

RoPE 最小实现： 把偶数/奇数维视作二维平面，按位置角度做旋转，面试重点：为什么它天然支持相对位置信息。

Prefix Cache / KV Cache 估算： 面试重点：区分"减少重复算力"和"占用更多显存"这两个方向。

最实用的练习顺序：

1. 先能口述公式和张量维度
2. 再能不看资料写出 Python 伪代码
3. 最后再练 PyTorch 版本和边界条件

一个现实建议： 面试里真正拉开差距的，不是把代码写到可运行，而是你能一边写一边解释：每个 tensor 代表什么，为什么这样写，哪里最容易数值不稳定。

显存手算模板与典型口径

三类最常用公式：

- 训练静态显存：`params + grads + optimizer states`
- KV Cache：
  2 \times L \times B \times S \times n_{\text{kv_heads}} \times d_{\text{head}} \times \te
- LoRA / QLoRA：`base weights + adapter weights + optimizer states`

典型口径 20 类：

72B BF16 全量微调 / 13B BF16 全量微调 / 32B ZeRO-3 / 72B QLoRA / 70B KV cache / GQA 前后 KV 节省 / PPO 四模型静态显存 / GRPO 三模型静态显存 / TP=4 时单层切分 / PP=4 时层拆分 / EP 下专家参数分布 / Prefill 长 prompt 显存 / Decode batch 扩大时 KV 增长 / INT4 与 BF16 权重差异 / FP8 权重与激活 / Prefix cache 复用收益 / Chunked prefill 对 TTFT 的影响 / 多 LoRA 常驻显存 / Reward model 服务成本 / Rollout 占用与 trainer 占用对比。

最常见手算样例（72B 全量微调）：

$$\text{参数（BF16）：}\ 72 \times 2 = 144\text{GB}$$

$$\text{梯度（BF16）：}\ 72 \times 2 = 144\text{GB}$$

$$\text{Adam } m\text{（FP32）：}\ 72 \times 4 = 288\text{GB}$$

$$\text{Adam } v\text{（FP32）：}\ 72 \times 4 = 288\text{GB}$$

$$\text{静态合计：}\ 864\text{GB}$$

再加激活值、buffer、碎片后，8 张 80GB 卡显然不够。手算不是背数字，而是先拆组成项。

RLHF / GRPO 显存口径：

- PPO 常需要 `policy + ref + reward + critic`
- GRPO 常需要 `policy + ref + reward`

在同等参数规模下，GRPO 至少天然少掉一整套 critic 显存与训练成本。

面试里更稳妥的说法： "我先报静态显存下界，再补激活值、通信 buffer 和碎片上界，最后再说明 rollout 成本是不是主瓶颈。"

Decode 侧手算样例：

若 $L = 80$、$B = 16$、$S = 4096$、n_{\text{kv_heads}}=8、$d_{\text{head}} = 128$、$\text{bytes} = 2$：

$$KV \approx 2 \times 80 \times 16 \times 4096 \times 8 \times 128 \times 2$$

量级会落到数 GB 到十几 GB，非常适合说明：为什么 decode 常是 memory-bound；为什么 GQA、PagedAttention、prefix caching 值钱。

Wall-clock 口算模板：

若平均每轮需要 rollout 10k 个 prompt，平均每个 prompt 生成 1k token，rollout 引擎总吞吐是 200k tok/s：

$$\text{rollout 时间} = \frac{10^4 \times 10^3}{2 \times 10^5} \approx 50\text{s}$$

这还没算 verifier、reward、参数更新和 eval gate。

更完整的 RLHF 平台估算样例：

假设：20k prompts/round，每个 prompt 平均 800 rollout token，rollout 吞吐 250k tok/s，reward/verifier 折合 40s/round，trainer 更新与 eval 再花 30s/round：

$$\text{rollout 时间} \approx \frac{2 \times 10^4 \times 800}{2.5 \times 10^5} \approx 64\text{s}$$

$$\text{单轮总时间} \approx 64 + 40 + 30 \approx 134\text{s}$$

面试结论： 这类系统里，rollout 和 verifier 往往才是 wall-clock 真正的大头；trainer 反而不一定是最慢的部分。

> 💬 **一句话总结**：显存手算决定"能不能放下"，wall-clock 手算决定"值不值得跑"。

17.4 面试表达：项目深挖与失败案例模板

项目深挖模板

一个可直接口述的项目模板（问题 → 方案 → 数据 → reward → eval → 资源 → 失败 → 回滚）：

1. 问题背景
2. 为什么旧方案不够
3. 方案设计
4. 数据来源与筛选
5. reward / verifier / eval
6. 资源与并行
7. 结果指标
8. 失败案例
9. 灰度与回滚

把"指标-资源-权衡"也嵌进去，效果会更像真实项目复盘：

- 训练花了多少卡时
- rollout 占了多少 wall-clock
- 线上延迟和成本变化多少
- 为什么最终选了这个方案，而不是另一个更"理论最好"的方案

失败案例与权衡表达模板

失败案例不要回避，要会这样讲：

"我们一开始用了方案 A；它在指标 X 上很好，但在 Y 上出现了副作用；所以后来切到方案 B，并加了监控/回滚。"

权衡表达模板： 不是"最好"，而是"在当前资源、风险和业务目标下最合适"。

更像真实面试的失败案例说法（RL 训练）：

"我们最开始把更多历史轨迹 replay 回训练里，短期离线分数涨了，但线上 trace quality 明显下降。后来定位到 stale sample 和 reward drift 同时存在，所以把 replay 窗口缩短，并加了 freshness filter 和高分样本人工审计，之后线上效果才稳定下来。"

多模态/Agent 失败案例模板：

"我们一开始把 GUI agent 的 reward 几乎全压在最终 task success 上，结果模型学会了乱点和碰运气。后来把 reward 拆成 schema correctness、action success、state progress、task success 四层，轨迹质量才稳定下来。"

推理训练失败案例模板：

"我们一开始把更多 rollout 轮次都堆到长链推理上，离线 AIME 提升了，但 tokens-per-solved 指标急剧恶化。后来发现是 overthinking 在刷 benchmark，于是加了 budget-aware reward 和 stop reason 监控，才把成本拉回可接受范围。"

更完整的权衡表达句式：

"方案 A 理论上更强，但 rollout 成本和线上延迟都太高；方案 B 离线分低一点，但可监控、可回滚、上线风险更低，所以我们先选 B，再在高价值 slice 上继续迭代。"

这类表达的价值：它说明你不是只会复述成功案例，而是真的做过权衡、踩过坑、也知道怎么回滚。

17.5 参考更新锚点（2025-2026 官方资料）

资源	链接
Qwen GSPO 官方博客	https://qwenlm.github.io/blog/gspo/
DAPO 项目页	https://dapo-sia.github.io/
VAPO 论文	https://arxiv.org/abs/2504.05118
SGLang 官方仓库	https://github.com/sgl-project/sglang
Qwen2.5-VL 官方博客	https://qwenlm.github.io/blog/qwen2.5-vl/
Qwen2.5-VL-32B 官方博客	https://qwenlm.github.io/blog/qwen2.5-vl-32b/
Qwen2.5-Math-PRM 官方博客	https://qwenlm.github.io/blog/qwen2.5-math-prm/
Qwen2.5-Turbo 1M Context 官方博客	https://qwenlm.github.io/blog/qwen2.5-turbo/
PyTorch FSDP2 官方文档	https://docs.pytorch.org/docs/2.8/distributed.fsdp.fully_shard.html
TorchTitan 官方仓库	https://github.com/pytorch/torchtitan
Qwen3 官方博客	https://qwenlm.github.io/blog/qwen3/
DeepSeek-R1 官方仓库	https://github.com/deepseek-ai/DeepSeek-R1
DeepSeek-V3 官方仓库	https://github.com/deepseek-ai/DeepSeek-V3
MCP 规范与变更日志	https://modelcontextprotocol.io/specification/
MCP 2025-03-26 changelog	https://modelcontextprotocol.io/specification/2025-03-26/changelog
MCP 2025-11-25 changelog	https://modelcontextprotocol.io/specification/2025-11-25/changelog

资源	链接
MCP Streamable HTTP 传输	https://modelcontextprotocol.io/specification/2025-11-25/basic/transports
vLLM Automatic Prefix Caching	https://docs.vllm.ai/en/latest/design/prefix_caching/
veRL 官方仓库	https://github.com/volcengine/verl
OpenRLHF 官方仓库	https://github.com/OpenRLHF/OpenRLHF