3. 中训练

继续预训练：受控领域适配、稳定性工程与系统兼容性

Table of contents

1 概览

你们团队拿到一个公开可用的基座模型。它能解释 Python、总结文档、在通用 benchmark 上表现不错。于是团队把它接到公司的内部代码库，目标是把它做成“代码助手”：理解私有 API、解释内部服务依赖、根据 runbook 生成修复建议。上线前的演示看起来很顺利；上线后的真实流量却暴露出完全不同的问题：模型会把内部 API 名字补全错、会把废弃配置当成现行约定、会在多文件上下文里失去仓库层级感。它不是不会写代码，而是不会“站在这个代码库的分布里思考”。

这时，prompting 往往只能修表面：你可以把 system prompt 写得更细、塞更多 few-shot，也可以上检索，把 README 和接口文档喂给模型。它们都有效，但都没有真正改变模型内部的先验。如果模型对仓库里的命名习惯、目录结构、调用模式和内部术语没有参数化表示，它就会持续把“看起来像代码”的模式误当成“这个仓库里的代码”。这就是中训练（**mid-training / continued pretraining, CPT**）存在的原因。

中训练不是重新训练一个模型，而是对已经得到的基座模型做受控的继续预训练：目标函数常常仍是下一个 token 预测，但训练分布、数据混合、学习率、恢复方式、回归门控和后续兼容性要求都变了。它让模型向目标领域移动，但不会自动保证移动过程稳定，更不会自动保证移动后仍然适合聊天、工具使用或安全对齐。

从工程角度看，CPT 的价值主要体现在四类场景：

- 模型需要参数化地掌握新领域知识，而不仅仅是在推理时临时检索；
- 领域文本的结构和语言习惯与通用互联网语料差异很大，例如法规、病历、论文、日志、代码仓库；
- 分词和上下文先验需要调整，例如高频术语被切得过碎、长文档结构学习不足；
- 你希望后续 **SFT**、偏好优化或 **RL** 站在一个更接近目标分布的起点上。

但一旦继续训练，风险也同时进入系统：灾难性遗忘、稳定性鸿沟、分布错配、分词器不一致、**checkpoint** 恢复方式不一致、后续对齐兼容性变差。因此，中训练不是“多喂一点数据”这么简单，而是一种典型的系统工程任务：你要在专业化收益与通用能力保全之间做受控权衡。

> **Note**
>
> 本章的中心问题
> 一旦基座模型已经预训练完成，工程师该如何在不破坏既有能力的前提下，把它适配到新的领域、语言、上下文长度或任务分布？

> **Tip**
>
> 贯穿案例：内部代码助手
> 本章会反复使用一个统一案例：一个通用代码模型要被适配到企业私有代码库。我们会用它串起“什么时候该做 CPT、如何配数据、为什么会掉通用能力、什么时候值得扩 tokenizer、以及 CPT 如何影响后续 SFT / DPO / RL”。

1.1 中训练在生命周期中的位置

> **Important**
>
> 本节先回答几个关键问题：
> 1. 现代 LLM 的生命周期为什么不能只分成“训练”和“推理”两步？
> 2. 中训练在整个模型生命周期中扮演什么角色？
> 3. 为什么“先判断缺口类型，再选训练杠杆”是比“先上微调”更稳的工程顺序？

中训练是连接“通用预训练”和“任务对齐”的桥梁。预训练负责让模型学会一个广义语言世界；中训练负责把它带到更接近目标分布的位置；后训练再把模型行为塑形成可部署的产品形态。把这三步混为一谈，会直接导致方法选错、预算浪费，甚至把该由 SFT 解决的行为问题误交给 CPT，反过来把该由 CPT 解决的知识问题误交给提示工程。

如果把整个系统抽象成“缺口诊断 $\rightarrow$ 选择最便宜但有效的杠杆”，可以写成一个非常工程化的目标：

$$a^* = \arg\min_{a \in \mathcal{A}} C(a) \quad \text{s.t.} \quad Q(a; g) \geq \tau, \quad R(a) \leq \rho$$

其中：

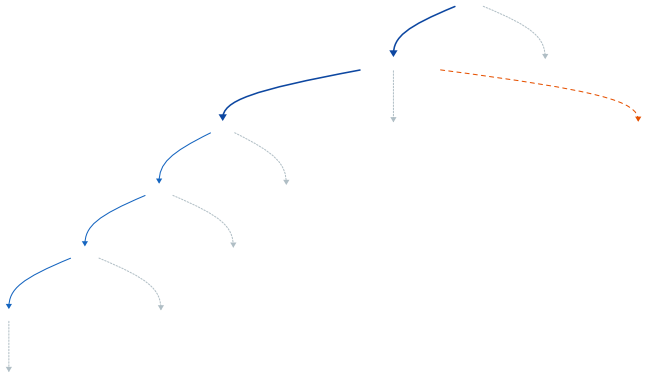

Figure 1: 图 1 · LLM 生命周期与中训练的位置

- $\mathcal{A} = \{\text{Prompt}, \text{RAG}, \text{SFT}, \text{CPT}, \text{DPO}, \text{RL}, \text{Distill}\}$ 是可选干预集合；
- g 是“缺口向量”，例如知识缺口、行为缺口、搜索缺口、实时性缺口、成本缺口；
- $C(a)$ 是实施成本；
- $Q(a; g)$ 是该方法对当前缺口的预期修复效果；
- $R(a)$ 是引入的回归或系统风险。

这个公式并不追求精确可解，它的作用是提醒你：选方法是一个受约束的系统优化问题。

从工程实践看，不同阶段改动的对象并不相同：

- 预训练主要塑造基础表示空间和通用语言先验；
- 中训练主要改变模型面对某类分布时的参数化知识、术语习惯和文档结构先验；
- 后训练主要塑造交互行为、偏好排序、安全边界、工具使用模式；
- 推理系统决定在固定模型下如何花预算换质量、换延迟或换吞吐。

这也意味着，一个成熟的工程回答通常不是“我要微调一下”，而是先定义缺口：

1. 如果缺的是知识，例如模型不懂合同条款、不熟悉内部 API、不理解医学缩写，优先考虑 CPT 或外部记忆；
2. 如果缺的是行为，例如不会按 JSON 输出、不会按客服语气回答、不会调用工具，优先考虑 SFT；
3. 如果缺的是偏好或长程策略，例如多步推理质量不稳、回答优劣排序不稳定，再考虑 DPO 或 RL；
4. 如果缺的是成本与延迟，就不该再加训练，而该做蒸馏、量化、服务优化。

> **Note**
>
> 把贯穿案例放进生命周期里看
> 对“内部代码助手”来说，预训练给了模型通用编程能力；中训练负责把它带到“这个仓库”的语言世界；SFT 再让它学会按团队规定输出补丁、解释变更、调用检索或代码搜索工具；后续偏好学习或 RL 才会去优化“哪种修复建议更可靠、哪种工具轨迹更省成本”。

1.2 干预选择：何时用 Prompt、RAG、SFT、CPT、RL

> **Important**
>
> 本节先回答几个关键问题：
> 1. 什么是 CPT？它与预训练和 SFT 有何不同？
> 2. CPT、SFT、提示工程和 RAG，分别适合解决什么问题？
> 3. 什么时候 continued pretraining 是正确解，什么时候它反而是过度训练？
> 4. 如果你有一个开源 7B 模型和 5 万份临床笔记，应该做 CPT、SFT，还是两者都做？
> 5. 中训练能否用来引入新语言、新领域，甚至新模态？它的边界在哪里？

CPT 解决的是“参数化知识与分布先验不足”，不是“交互行为不对”。它沿用预训练阶段的 next-token 目标，但把模型继续拉向新的数据分布。因此，CPT 最适合处理的是领域知识、术语密度、文档结构、长上下文习惯和部分词表问题；如果问题只是输出格式、角色语气、工具调用协议或安全拒答边界，CPT 通常不是第一选择。

1.2.1 CPT 与预训练、SFT 的本质区别

形式上看，预训练和 **CPT** 用的都是下一个 token 预测；区别不在损失函数，而在训练分布、学习率和目标范围。预训练是在极大、极杂的通用分布上获得基础能力；CPT 则是在一个更窄、更有偏向性的分布上做受控继续训练。

而 **SFT** 则不同。SFT 的核心不是让模型更懂某个语言世界，而是让模型在给定用户输入时学会一种行为映射：什么情况下直接回答、什么情况下调用工具、什么格式才算合格输出。SFT 的训练样本通常是示范对话或结构化轨迹，而不是大规模原始文档。

因此可以用一句话区分三者：

- 预训练：学会世界的大体统计结构；
- **CPT**：把“世界”改成更靠近目标领域的世界；
- **SFT**：在这个世界里学会按要求行动和表达。

1.2.2 一个更可执行的诊断框架

把“该选哪种杠杆”写成一个简化诊断器，会更接近真实工程流程。设：

- g_k：知识缺口（knowledge gap）
- g_b：行为缺口（behavior gap）

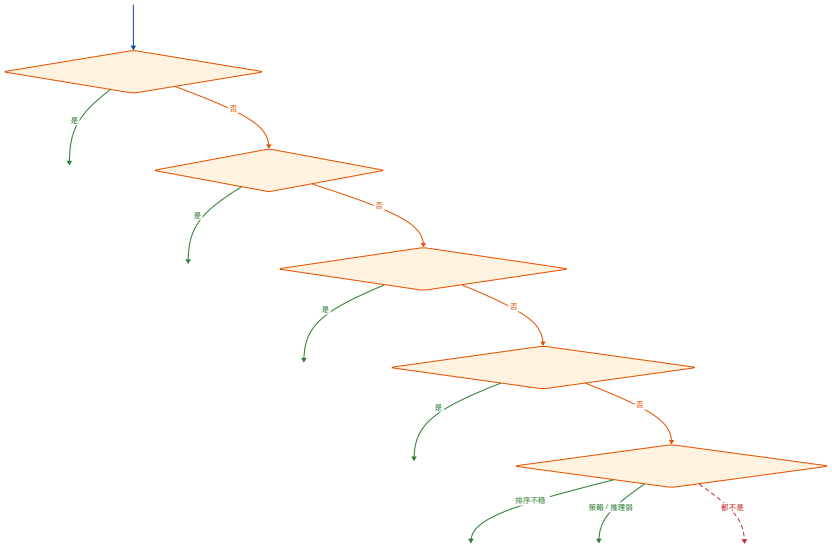

Figure 2: 图 2 ：决策树：什么时候用 Prompt、RAG、SFT、CPT、RL？

- g_f：知识新鲜度缺口（freshness gap）
- g_t：工具使用 / 轨迹缺口（tool / trajectory gap）
- g_r：搜索与长程策略缺口（reasoning / search gap）

那么一个非常实用的白板判断是：

$$\text{choose} = \begin{cases} \text{Prompt} & \text{if } g_k, g_b, g_r \text{ 都不高，只是触发不稳} \\ \text{RAG} & \text{if } g_f \text{ 高，知识必须外部更新} \\ \text{SFT} & \text{if } g_b \text{ 或} g_t \text{ 高} \\ \text{CPT} & \text{if } g_k \text{ 高，且有大规模领域语料} \\ \text{RL} & \text{if } g_r \text{ 高，且任务可验证} \end{cases}$$

当然，真实系统往往是组合拳而不是单选题，但这个框架至少能防止一个常见错误：把所有问题都看成“再训一下就好”。

1.2.3 什么时候该用 CPT，什么时候不该

Table 1: CPT 决策速查表

信号	判断	原因
模型在目标领域的 **perplexity** 明显高于通用语料	做 CPT	模型尚未站在该领域分布上
高频术语、实体名、缩写和引用格式经常出错	做 CPT	术语共现和文档结构先验不足
输入是法规、病历、论文、长报告、日志、代码仓库等结构化长文档	做 CPT	长文档组织方式需要参数化学习
尝试过 prompting 或少量 SFT，模型“说话更像了”但事实和术语仍不稳	做 CPT	表层行为可学，但知识先验仍缺
希望后续 SFT / RL 站在更接近任务分布的初始化上	做 CPT	降低下游对齐的知识填补负担
只需要模型把已有能力更稳定地触发出来	不做 CPT	用 Prompt / few-shot 即可
知识更新很快，参数化存储不如检索可维护	不做 CPT	优先 RAG / 外部记忆
问题是输出格式、函数调用 schema、角色语气、拒答边界	不做 CPT	这是 SFT 的工作
没有足够干净的领域文档，只有少量高质量示范	不做 CPT	数据不足以支撑 CPT

信号	判断	原因
部署预算紧张，无法承担 CPT 算力、评估和再对齐成本	不做 CPT	CPT 是昂贵的高杠杆手术

经典判断题：如果模型在法律文档上的困惑度很高，但通用性能不错，优先修的是 CPT。困惑度高说明模型没有站在法律文本分布上，术语共现、引文格式和长文档组织方式都不稳；这不是简单多写几个提示就能补上的。

1.2.4 一个临床笔记的工程案例

你有一个开源 7B 模型和 5 万份临床笔记，目标是做医疗问答助手。一个成熟的回答不该是“直接 SFT”或“直接 CPT”，而应当分层判断。

首先，这批数据的形态更像原始领域文档，而不是已经整理好的问答示范，因此它天然更适合做 CPT，而不是直接当 SFT 数据。其次，医疗领域高度依赖术语、缩写、实体和文档结构先验，模型即使会一般问答，也未必具备稳定的临床语言习惯。更稳妥的路径通常是：

1. 先做数据治理：脱敏、去模板、去重、过滤明显错误样本；
2. 先做 **CPT**：让模型获得临床术语和病历结构先验；
3. 再做少量高质量 **SFT**：把模型塑形成问答、摘要、分诊建议等明确产品行为；
4. 最后做安全与评估：医疗领域里，“似懂非懂”往往比直接拒答更危险。

所以这个问题的好答案通常是：两者都做，但顺序不是随意的，而是 **CPT** 打底，**SFT** 定形。

1.2.5 新语言与新模态的边界

CPT 也常被用来适配新语言或新模态，但边界必须说清楚。

- 新语言：如果 base model 已经有对应脚本或字节级分词能力，只是语料暴露不足，那么 CPT 很有效；
- 新词表 / 新脚本：可能需要分词器扩展，甚至 embedding resize；
- 新模态：如果 base model 本身没有视觉、音频或其他模态的输入通路，单纯 CPT 并不能凭空获得新模态能力，通常还需要新增编码器、adapter 或跨模态对齐层。

这类边界意识很重要，因为它能区分“继续预训练可以解决的问题”和“需要改架构的问题”。

> **Note**
>
> 把贯穿案例放进决策树里看
> 对内部代码助手来说，如果问题只是“模型知道 API，但总是忘记用固定 patch schema 输出”，应先做 SFT；如果问题是“模型根本不熟悉仓库中的内部函数和目录结构”，才值得上 CPT；如果知识频繁变化，检索工程文档可能比参数化记忆更可维护。

2 CPT 到底改变了什么：目标不变，分布改变，先验重写

> **!** Important
>
> 本节先回答几个关键问题：
>
> 1. 为什么 continued pretraining 明明还在做 next-token prediction，模型行为却会发生显著变化？
> 2. “目标不变，分布改变”到底改变了什么：知识、表示空间，还是行为接口？
> 3. 为什么有些问题应该交给 CPT，有些问题却必须留给 SFT、DPO 或 RL？
> 4. 从数学上看，CPT 与预训练到底相同在哪里，又不同在哪里？

对一个自回归模型来说，目标函数可以保持不变，但只要训练分布从通用网页切换到法律文本、病历、论文、日志或内部代码库，最优参数就会随之改变。模型学到的不只是新事实，更是“什么词更常见、什么结构更正常、什么续写更像这个领域”。

目标函数保持不变，但最优参数会变

把自回归语言建模写成统一形式，可以得到：

$$\mathcal{L}_{AR}(\theta; D) = -\mathbb{E}_{x\sim D} \sum_{t=1}^{|x|} \log p_\theta(x_t \mid x_{<t})$$

如果把某个训练分布 D 对应的最优参数记为：

$$\theta^*(D) = \arg\min_\theta \mathcal{L}_{AR}(\theta; D)$$

那么从预训练到中训练，并不是把目标换掉，而是把 D 从更宽的通用分布 D_{web}，换成更窄、更偏向目标领域的 D_{domain} 或混合分布 D_{mix}。只要分布改变，$\theta^*(D)$ 就会改变。这就是为什么 **“还是 next-token loss”**，模型却会变得越来越像律师、医生、审计员或内部代码库维护者。

CPT 常被一句话概括为“继续用下一个 token 训练”，但这句话很容易掩盖真正重要的工程事实：目标函数不变，并不意味着系统行为不变。当训练分布从通用网页、论坛、百科，切换到法规、病历、论文或内部代码库时，模型会重估大量条件概率：

- 术语与术语之间的共现关系；
- 某类实体出现的先验频率；
- 文档的局部结构和长程组织方式；
- 哪些 token 序列在新世界里“更自然”。

可以把数据混合写成一个简单的形式：

$$D_{mix} = \lambda D_{domain} + (1-\lambda) D_{replay}$$

而 CPT 的目标仍然是：

$$\mathcal{L}_{CPT}(\theta) = -\mathbb{E}_{x\sim D_{mix}} \sum_{t=1}^{|x|} \log p_\theta(x_t \mid x_{<t})$$

这里真正决定模型被拉向哪里的，不是公式长什么样，而是 D_{mix} 里每一类数据各占多少。

如果你想把“领域收益”和“通用保留”显式写成一个双目标问题，也可以写成：

$$\max_\theta \ \Delta M_{domain}(\theta) \quad \text{s.t.} \quad \Delta M_{general}(\theta) \geq -\tau_g, \qquad \Delta M_{safety}(\theta) \geq -\tau_s$$

这就是中训练在工程上的真实约束：你不是单纯最大化领域能力，而是在一组回归边界内最大化领域增益。

一个直观例子：同样是“续写”，世界已经变了

设两个上文分别来自两个不同分布：

- 通用网页：`The service supports multiple environments, including ...`
- 企业内部仓库：`The billing worker writes retries to /svc/retry_queue and calls ...`

在前一个分布里，模型更可能接出“常见教程式解释”；在后一个分布里，模型需要学会内部路径、私有 API、团队约定和仓库层级。两者都叫“续写”，但条件概率表已经不是同一张表。

同样的现象也发生在法律和医疗场景里：

- 合同文本里的 token 更偏向条款、例外、引文和长程交叉引用；
- 病历文本里的 token 更偏向缩写、时间序列、实验室指标和临床风格；
- 代码库文本里的 token 更偏向路径、配置键、函数族和仓库内部共现模式。

因此，中训练不是“再背一点知识”，而是把模型的条件概率地形重新雕刻成更接近目标领域的地形。

CPT、SFT、DPO / RL 分别在改什么

把几个常见训练阶段并排比较，会更清楚：

方法	主要优化目标	主要改变什么	典型数据形态	典型问题
预训练 / CPT	自回归 next-token prediction	知识先验、术语共现、表示空间、长文档习惯	原始文档、代码、长文本	“模型根本不懂这个语言世界”
SFT	监督式行为映射	输出格式、角色语气、工具模板、对话接口	指令对话、工具轨迹、示范答案	“模型知道，但不会按要求做”
DPO / ORPO	相对偏好目标	回答质量倾向、偏好排序	chosen / rejected 偏好对	“答案都能写，但优劣不稳”

方法	主要优化目标	主要改变什么	典型数据形态	典型问题
RL	期望奖励最大化	搜索、探索、长程策略	rollout + reward / verifier	“任务需要试错、规划和可验证反馈”

- CPT 改的是模型相信什么世界更常见；
- SFT 改的是模型在这个世界里该怎么回答用户；
- DPO / RL 改的是模型在多个可能行为里更偏向哪一种。

为什么这件事在真实系统里重要

当一个团队说“模型会答错内部 API，但 JSON schema 也不稳”，这其实是两个层面的问题：

1. 它不认识这个仓库的分布，这是 CPT 或外部记忆要解决的事；
2. 它不会按产品约束输出，这是 SFT 或推理约束要解决的事。

如果把这两个问题混成一个问题，最常见的结果就是：用错误的训练阶段去修错误的缺口，最后既没有学会新领域，也把已有行为底盘练坏。

3 数据工程：领域语料、mixture、general replay、packing

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 什么样的数据真正适合继续预训练：原始文档、QA 对、工具轨迹，还是网页抓取混合物？
> 2. 为什么 general replay 不是一个小技巧，而是中训练最关键的稳定性杠杆之一？
> 3. 为什么 mixture 不是“配料表”，而是一个直接控制领域收益与遗忘风险的旋钮？
> 4. packing 明明是在优化吞吐量，为什么却会改变梯度看到的数据拓扑？

大多数 **CPT** 的成败，首先是一个数据工程问题。
loss 只是一个表面接口；真正决定模型会被拉向哪里的，是你给它看的语料分布、子域权重、通用回放比例、文档边界策略，以及这些东西随训练进程如何变化。

3.1 领域语料长什么样

CPT 的最佳数据通常不是精心写好的对话答案，而是领域原始文档：法规、判例、合同、病历、论文、规范、代码库文件、日志、注释、设计文档。原因很直接：CPT 想学习的是这个领域的语言世界本身，而不是用户—助手之间的行为映射。

这类数据在工程上有三个价值：

1. 它保留了自然分布中的术语与共现关系；
2. 它保留了真实的文档结构，例如章节层级、引文、表格上下文、代码文件边界；

3. 它更便于大规模收集和持续扩充。

当然，这也意味着数据清洗更重要。网页抓取残留、模板页脚、重复样本、导航条、OCR 错误、拼接失败都会直接进入参数更新。很多团队以为自己在做“领域适配”，实际上是在教模型记住模板噪声。

3.2 通用数据与领域数据的混合比例

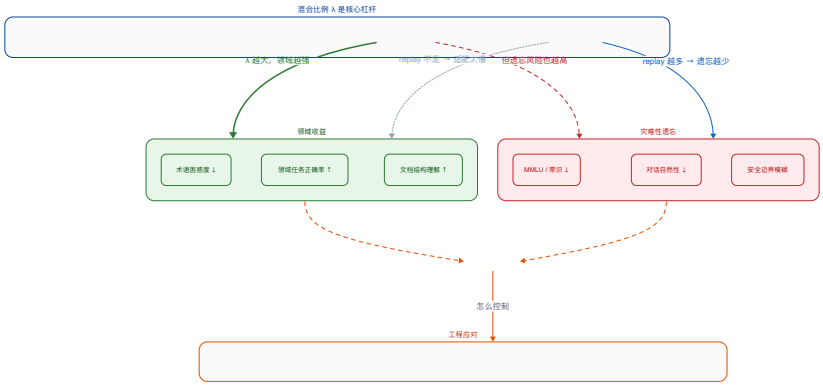

Figure 3: 图 3 · 领域适配 vs 通用能力保持：混合比例是杠杆，不是细节

把混合分布写成：

$$D_{mix} = \sum_{i=1}^{K} \alpha_i D_i, \qquad \sum_{i=1}^{K} \alpha_i = 1, \quad \alpha_i \geq 0$$

其中至少有一个 D_i 是目标领域数据，另一个常见分量是 D_{replay}。于是，调 **mixture** 并不是“调味”，而是在改变模型下一次参数更新看到的世界组成。这也是为什么很多团队会把“80/20”写进经验配方：它不是神圣数字，但它提醒你 replay 不是事后补丁，而是训练分布本身的一部分。

perplexity 常被用来判断模型是否真正站在某个分布上。给定领域语料 $x = (x_1, \ldots, x_T)$，其平均负对数似然为：

$$\ell(x;\theta) = -\frac{1}{T} \sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t})$$

对应的困惑度为：

$$\mathrm{PPL}(x;\theta) = \exp\left(\ell(x;\theta)\right)$$

如果模型在目标领域上的 PPL 明显高于通用语料，同时实体、术语和结构错误频发，这通常是一个更偏向 CPT 的信号，而不是单纯的 prompting 或 SFT 问题。

所谓 **general replay**，就是在喂入新领域语料的同时，保留一部分原本更通用、更多样的训练数据。它的作用并不仅仅是“别忘了老知识”，而是同时承担三件事：

- 降低灾难性遗忘，避免模型过快脱离原来的通用表示；
- 缓冲优化冲击，让参数更新不要完全被窄分布主导；
- 维持后续对齐基础，保留通用问答、常识推理和对话行为的底盘。

一个常见的起始配方是 **80%** 领域数据 / **20%** 通用回放，但它不是法则，只是工程上一个保守起点。真正该怎么调，要看两个方向的信号：

- 如果领域指标涨得慢，可以逐步提高领域占比；
- 如果通用评估掉得快，应当提高 replay、降低学习率、改善数据质量，必要时减小训练步数。

策略	做法	优点	风险
固定混合	全程固定 80/20、90/10 等配方	简单、稳定、易复现	可能不是最优点
课程学习	前期 replay 更高，后期逐步增加领域占比	前期更稳，后期更专	调参更复杂，收敛更慢
子域自适应采样	对 loss 高的子域临时加权	能补齐最薄弱子域	容易让整体分布抖动
回归门控驱动调整	评估退化时自动上调 replay 或降 LR	更工程化，适合生产	需要持续评估和自动化基础设施

对多数团队来说，比较稳的顺序是：先从固定混合起步，再根据回归门控决定是否需要课程学习或自适应权重。

3.3 Packing：吞吐优化与文档边界风险

CPT 常常处理海量长短不一的文档。如果不做任何拼接，短文档会带来大量 padding，严重浪费 GPU。于是工程上常见做法是把多个文档拼到一个固定上下文窗口里，这就是 **document packing**。它的数学形式和工程含义，我们会在后面的“训练拓扑与系统实现”一节详细展开；这里先记住一句话：

> Packing 提高的是吞吐量，但如果文档边界处理不当，它也会把不存在的跨文档模式写进参数里。

> **Note**
>
> 把贯穿案例放进数据分布里看
> 对内部代码助手来说，最有价值的 CPT 数据不是“如何回答代码问题”的问答对，而是仓库里的真实语言世界：源文件、接口文档、变更说明、runbook、注释、配置模板。只有这样，模型才会学到“这个代码库的条件概率长什么样”。

4 评估，门控与停止准则

> **Important**
>
> 本节先回答几个关键问题：
> 1. 一次中训练 run 到底该看哪些指标，才算真正“在变好”？
> 2. regression gates 应该怎样设计，才能在模型开始漂移时及时刹车？
> 3. checkpoint 选择应该找最后一个点、最高领域分，还是 Pareto 最优点？
> 4. 什么情况下应该继续跑，什么情况下应该果断停止？

中训练不是“训练完再评估”，而是“边训练、边评估、边决定是否继续投入 **token**”。一个 run 是否成功，不是由单一领域指标决定，而是由一组带约束的多目标判断共同决定：领域收益有没有上升，通用能力有没有跌穿底线，行为和安全有没有出现不可接受的回归。

4.1 评估集的构成

中训练最常见的评估错误，是把领域 benchmark 当作唯一信号。更稳的做法是从一开始就把 eval suite 分成四类：

$$\mathcal{E} = \{\mathcal{E}_{domain}, \mathcal{E}_{general}, \mathcal{E}_{behavior}, \mathcal{E}_{safety}\}$$

- $\mathcal{E}_{domain}$：法律、医疗、内部代码、长文档等目标任务；
- $\mathcal{E}_{general}$：常识问答、通用推理、基础写作；
- $\mathcal{E}_{behavior}$：结构化输出、工具模板、对话可用性；
- $\mathcal{E}_{safety}$：拒答、敏感边界、极端输入稳定性。

然后给 stop criteria 一个明确形式。一个很实用的门控方式是：

$$\text{stop if} \quad \Delta M_{general} < -\tau_g \ \text{ or } \ \Delta M_{safety} < -\tau_s \ \text{ or } \ \Delta M_{domain} < \epsilon_d \ \text{for } N \text{ consecutive evals}$$

也就是说，“不再值得继续烧 **token**”和“已经不允许继续回归”应当是两个独立的停止条件。

4.2 checkpoint 选择：找 Pareto 前沿，而不是找最后一个点

如果把一次 run 在第 t 次评估时的领域、通用、安全指标记为 $(M_d(t), M_g(t), M_s(t))$，那么一个更工程化的做法是先定义 **Pareto** 前沿：

$$\mathcal{P} = \left\{ t \;\middle|\; \nexists s,\ M_d(s) \geq M_d(t),\ M_g(s) \geq M_g(t),\ M_s(s) \geq M_s(t) \text{ 且至少一项严格更好} \right\}$$

直觉上，$\mathcal{P}$ 里的 checkpoint 都是不被其他点“全面支配”的候选。然后你再在这条前沿上选择最适合产品约束的点，而不是机械地取：

- 最后一个 checkpoint；
- 领域分数最高的 checkpoint；
- 或者训练 loss 最低的 checkpoint。

对内部代码助手来说，这尤其重要：一个仓库级 pass@k 最高的点，可能恰好也是通用问答和拒答行为最差的点。把“最高分”错当成“最佳 **checkpoint**”，是中训练里最贵的误判之一。

4.3 门控：让评估进入训练闭环

regression gates 指的不是训练后做一次总评，而是把一组小型回归集变成训练中的持续监控器。一个实用的门控集至少应覆盖四类信号：

- 通用能力：如常识问答、通用推理、基础写作；
- 领域能力：你真正想提升的法律、医疗、代码或长文档任务；
- 行为能力：指令跟随、结构化输出、工具模板是否退化；
- 安全边界：拒答、敏感内容边界、极端输入稳定性。

如果要把 gate 写成一个“可以排序 checkpoint 的分数”，一个简单而实用的形式是：

$$J(t) = w_d \widetilde{\Delta M_d}(t) - w_g \widetilde{\mathrm{Reg}_{general}}(t) - w_b \widetilde{\mathrm{Reg}_{behavior}}(t) - w_s \widetilde{\mathrm{Reg}_{safety}}(t)$$

这里的 $\tilde{\cdot}$ 表示归一化后的指标。它不是为了“把一切变成一个神奇数字”，而是为了让 checkpoint 选择从“拍脑袋看几张表”变成一个可审计的过程。

4.4 停止准则

你做了 20B token 的 CPT，MMLU 掉了 5 个点；继续跑到 50B token 后，分数又恢复了一部分。这样的 U 形曲线通常可以这样解释：

- 前期模型快速吸收新分布，通用表示先被挤压；
- 随着 replay 持续注入、学习率下降、优化逐步平稳，模型重新找到一部分平衡；
- 但“部分恢复”并不代表一定会恢复到起点，更不代表这是最优 checkpoint。

所以停止标准不应是“既然已经回了一些，不如再等等看”，而应是事先定义：

- 最低可接受通用分数；
- 最小领域收益阈值；
- 连续 N 次评估无改善则停止；
- 单位 **token** 收益是否还值得继续投入。

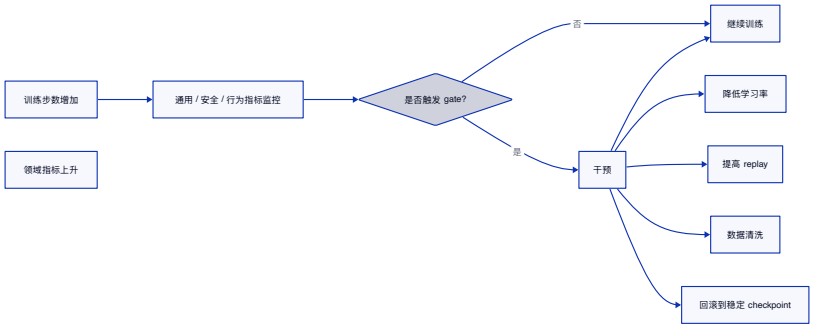

Figure 4: 图 3 · 稳定性鸿沟控制图：监控、门控与干预

把 stop policy 写成显式规则，会比“再等等看”稳得多。一个典型形式是：

$$\text{stop if} \quad \Delta M_{general} < -\tau_g \ \text{ or } \ \Delta M_{safety} < -\tau_s \ \text{ or } \ \Delta M_{domain} < \epsilon_d \ \text{ for } N \text{ consecutive evals}$$

其中：

- τ_g：通用能力最大可接受退化；
- τ_s：安全能力最大可接受退化；
- ϵ_d：最小领域收益阈值；
- N：连续多少次评估无改善就停止。

这套写法的价值不在于“把训练形式化得很漂亮”，而在于把资源分配从直觉问题变成显式的运营策略。

现在你知道怎么判断训练是否健康，看什么指标，怎么选 checkpoint，什么叫该停。接下来我们讲稳定性时，明确“训练坏了”怎么定义。

5 稳定性工程：学习率、恢复方式、梯度控制与回滚

! Important

本节先回答几个关键问题：

1. 什么是 CPT 中的稳定性鸿沟 (stability gap)？为什么它几乎是一个常见现象？
2. 开始 CPT 后通用基准突然下跌，应该按什么顺序排查？
3. 学习率、数据分布和梯度范数在稳定性里分别扮演什么角色？
4. 什么是 regression gates，为什么它比“训练完再看”更重要？
5. 遇到 U 形曲线时，应该如何设置停止标准，而不是盲目继续烧算力？

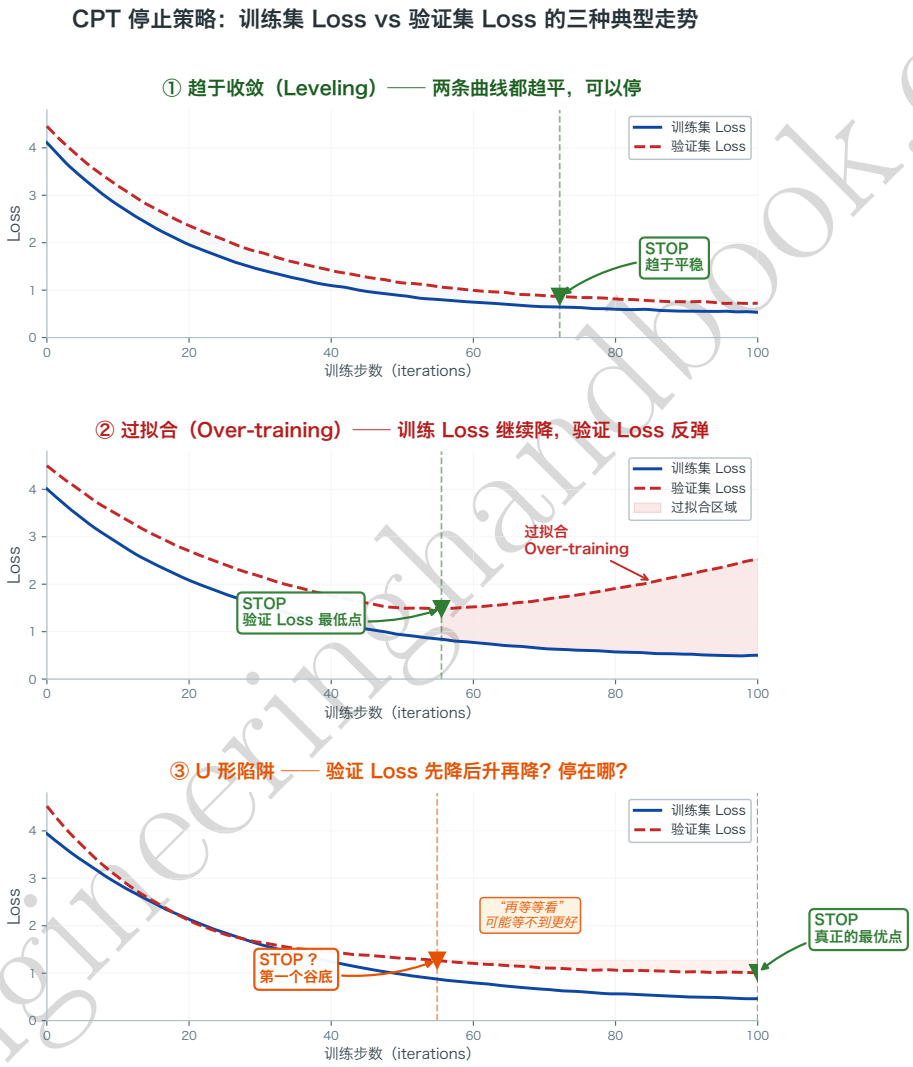

Figure 5: U 形曲线：通用基准先降后升 vs 领域 Loss 持续下降

CPT 的早期退化并不罕见，但必须被主动管理。稳定性问题之所以危险，不是因为模型一定会掉分，而是因为很多 run 的退化在前期看起来“只是暂时的”，团队于是继续训练，最后把原本可回滚的问题烧成了不可逆的分布漂移。

5.1 稳定性鸿沟到底是什么

所谓 **stability gap**，指的是模型一开始进入新分布训练后，领域 **loss** 下降很快，但通用能力先明显下滑。之后它可能部分恢复，也可能不恢复；真正的挑战在于，你不能靠希望来区分这两种情况。

可以把它写成两个随训练步数 t 变化的指标：

- 领域指标：$M_d(t)$
- 通用指标：$M_g(t)$

那么“稳定性鸿沟”最简单的形式就是：

$$\Delta_{stab}(t) = M_g(t) - M_g(0)$$

在很多真实 run 中，$M_d(t)$ 单调上升，而 $\Delta_{stab}(t)$ 在前期显著为负，后期才可能部分回升。这就是工程里常见的 **U** 形恢复轨迹。

它发生的原因，通常是多个因素叠加：

- 学习率不匹配：对一个已收敛的大模型来说，继续训练的可用 LR 往往比初始预训练更小；
- 分布切换过猛：领域数据太窄、变化太剧烈，模型被快速拽向一个局部子空间；
- **replay** 不足：旧分布没有足够权重，无法对冲新分布梯度；
- 梯度噪声上升：脏数据、重复模板、少量极端 batch、错误拼接都会放大不稳定；
- 长上下文与 **packing** 的副作用：更长的序列、错误的边界处理、序列并行配置问题都会让优化更脆。

5.2 学习率与 warm-up：为什么 CPT 比预训练更怕“手太重”

在中训练里，学习率往往是第一优先级的稳定性旋钮。原因并不神秘：CPT 不是从随机初始化开始学，而是在一个已经形成结构的参数空间里继续移动。移动得太快，旧表示会被粗暴改写；移动得太慢，领域适配又不充分。

一个常见但不严谨的做法，是直接沿用预训练阶段的峰值学习率或衰减策略。更稳妥的做法通常是：

- 使用更低的峰值 LR；
- 给继续训练单独设计 warm-up；
- 在早期密集评估，确认没有出现大幅回归；
- 必要时缩短单次训练窗口，多做 checkpoint 比较，而不是一口气跑完。

5.3 一个可执行的调试顺序

如果开始 CPT 后通用基准急剧下降，一个工程上可执行的排查顺序通常是：

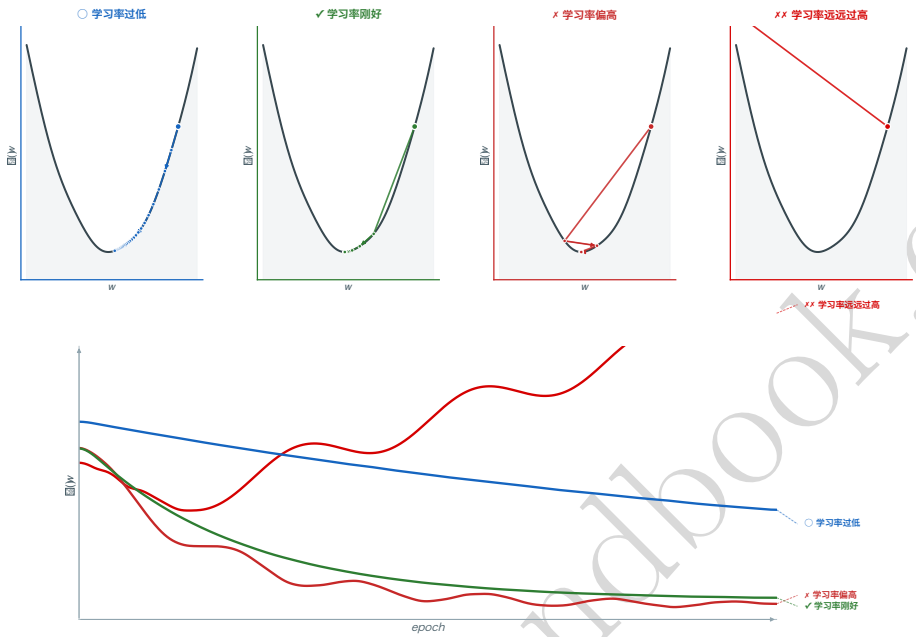

Figure 6: CPT 学习率策略：太高 / 刚好 / 太低的三种结局

1. 先看学习率与 **warm-up**：这是最常见的罪魁；
2. 再看 **replay** 比例：领域占比是否过激；
3. 检查数据质量：去重、去模板、去错分样本；
4. 检查 **packing** 与边界处理：是否引入了跨文档污染；
5. 查看梯度范数与 **loss** 曲线：判断是否有异常 batch 或优化器不稳；
6. 必要时加参考正则：例如与基座 logits 的 KL 约束，限制表示漂移；
7. 回滚并缩小实验面：在更小预算上复现问题，别在全量 run 上继续赌博。

如果要把“参考正则”写得更具体一点，一种常见形式是：

$$\mathcal{L}_{total} = \mathcal{L}_{CPT} + \beta\, \mathbb{E}_{x\sim D_{replay}}\big[\mathrm{KL}(p_{\theta_0}(\cdot \mid x) \,\|\, p_{\theta}(\cdot \mid x))\big]$$

其中 θ_0 是基座模型参数，θ 是当前 CPT 参数。这个项并不是 CPT 的默认组成部分，但在高风险场景里，它能作为一种“别离开太远”的软约束。

5.4 梯度控制、异常 batch 与训练中回滚

除了学习率和 mixture，另一个低估很严重的稳定性旋钮是梯度控制。最常见的做法包括：

- **gradient clipping**：限制异常 batch 的梯度爆发；
- **anomaly detection**：在 loss、梯度范数、长度分布上设置异常阈值；
- **rollback**：一旦 gate 被击穿，回滚到最近稳定 checkpoint。

梯度裁剪最常见的形式是：

$$g_t \leftarrow g_t \cdot \min\left(1, \frac{c}{\|g_t\|_2}\right)$$

其中 c 是设定的最大梯度范数。如果再加一个简单的异常规则，例如：

$$\text{rollback if } \|g_t\|_2 > \mu_g + k\sigma_g$$

你就得到了一条很朴素但常常很有用的工程守则：不要和异常 **batch** 赌运气。

5.5 一个带回放与门控的 CPT 训练骨架

```
# 伪代码：带 replay、packing、KL 参考约束与 regression gate 的 CPT 训练循环

for step in range(T):
    batch = sample(D_domain) if rand() < p_domain else sample(D_replay)
    x = pack_sequences(batch, seq_len=L, add_eos=True)

    logits = model(x[:, :-1])
    loss_nll = cross_entropy(logits, x[:, 1:])

    if use_ref_kl:
        with torch.no_grad():
            ref_logits = ref_model(x[:, :-1])
        loss_kl = kl_divergence(ref_logits, logits)
        loss = loss_nll + beta * loss_kl
    else:
        loss = loss_nll

    loss.backward()
    clip_grad_norm_(model.parameters(), max_norm=grad_clip)
    optimizer.step()
    optimizer.zero_grad()

    if step % eval_every == 0:
        metrics = run_regression_suite()
        if gate_triggered(metrics):
            rollback_or_tune(lr="down", replay="up", data="clean")
```

> **Note**
>
> 把贯穿案例放进稳定性里看
> 内部代码助手常见的失败不是“loss 不降”，而是“仓库任务上升、通用对话和安全行

为下降”。如果团队只看代码任务 pass@k，而不把通用问答、结构化输出和拒答放进 regression gates，就会把一个原本可回滚的 run，烧成一个只会“说仓库黑话”的模型。

稳定性工程的目标，并不是保证“指标永远不掉”，而是保证一旦开始掉，你知道该看哪里、该调什么、该不该继续投入算力。

6 分词器扩展：高风险结构性改动

! Important

本节先回答几个关键问题：

1. 什么时候应该在 CPT 期间扩展 tokenizer，而不是继续沿用原有词表？
2. 哪些领域最容易受到 token 碎片化影响：科学记数法、代码 token、多语言字符，还是法律引文？
3. 新增 token 的 embedding 应该怎么初始化，才能把风险降到最低？
4. 什么是“训练不足的 token”，它为什么会在扩词后频繁出现？
5. 分词器一旦扩展，训练、评估和部署链路为什么必须一起升级？

分词器扩展是一种高收益、高风险的“checkpoint 手术”，而不是常规动作。它可以显著降低领域术语的碎片化，节省上下文预算，改善建模效率；但同时也会改动 embedding 矩阵、序列长度分布、packing 效率、日志解析和服务链路。如果没有足够强的收益信号，最稳的默认做法通常仍是“不扩词表”。

6.1 什么时候值得扩展 tokenizer

一个可操作的判断方式，是统计目标领域的 **fragmentation rate**（碎片率）。对某个术语 u，令原分词器产生的子词序列长度为 $k(u)$，则可以定义：

$$r(u) = k(u)$$

如果对一个带权术语集 $\mathcal{V}_{domain}$ 按出现频率 $f(u)$ 取期望，则平均碎片率可写成：

$$\bar{r} = \frac{\sum_{u \in \mathcal{V}_{domain}} f(u)\, r(u)}{\sum_{u \in \mathcal{V}_{domain}} f(u)}$$

当大量高频、高价值术语的 $r(u)$ 长期偏高，例如：

- 生物医学中的药名、蛋白名、试验编号；
- 法律领域中的条款编号、引文格式；
- 代码中的 API 名、路径、配置键；
- 科学文献中的公式符号、单位写法、科学记数法；
- 多语言场景中的新脚本、罕见字符组合；

模型会同时承担两种成本：

1. 上下文成本：本来一个概念可以占 1 个 token，现在占了 4 到 8 个；
2. 学习成本：模型必须跨多个碎片重建一个稳定概念，训练和解码都更困难。

只有当这种碎片化足够频繁、足够贵，并且真实影响质量时，扩词才值得进入路线图。

6.2 不扩词表时你还能做什么

分词器扩展不是唯一答案。在很多场景下，更便宜的替代手段已经足够：

- 继续做领域 CPT，让模型学会在旧分词下也能稳定理解术语；
- 在推理系统中通过检索和模板减少复杂术语的自由生成；
- 在服务端做规范化，例如把常见别名映射成统一表达。

因此，分词器扩展应当被看作“当碎片化已成为明显瓶颈时的工程升级”，而不是美化词表的常规动作。

6.3 扩词是一台手术，不是一行配置

假设旧词表大小为 V，embedding 维度为 d，旧 embedding 矩阵为：

$$E_{old} \in \mathbb{R}^{V \times d}$$

现在新增 k 个 token，则新的 embedding 矩阵变成：

$$E_{new} \in \mathbb{R}^{(V+k) \times d}$$

如果模型使用 tied embeddings，那么输出头 W_{lm} 也需要同步扩容。对新 token u_{new}，一种更稳的初始化方式是用它原有子词拆分对应向量的均值：

$$E_{new}(u_{new}) = \frac{1}{m} \sum_{j=1}^{m} E_{old}(s_j)$$

其中 $(s_1, \ldots, s_m)$ 是旧分词器下对该术语的子词分解。和纯随机初始化相比，这种“折叠式初始化”更像把旧表示压缩成一个新位置，通常更平滑。

6.4 “训练不足的 token”问题

扩词后最常见的失效模式，就是 **undertrained tokens**。它的本质是：虽然新 token 已经进入词表，但相关样本在训练早期出现频率仍然不足，导致新增 embedding 行没有获得足够梯度更新，于是表现为：

- 生成时不稳定；
- 输出时偶尔像随机噪声；
- 在真正的领域上下文里反而比旧分词更差。

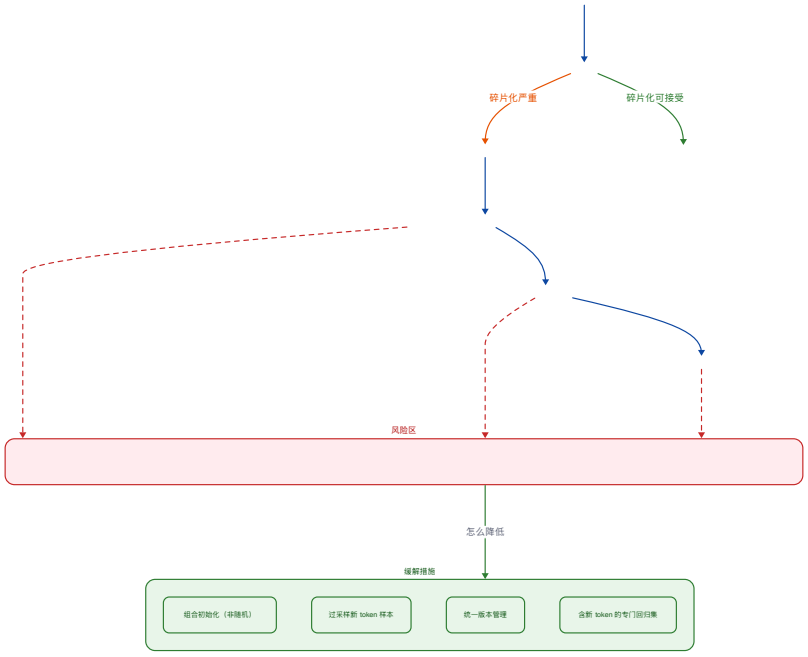

Figure 7: 图 4 · 分词器扩展与 checkpoint 手术

缓解方法通常包括：

- 对包含新 **token** 的样本做过采样；
- 给扩词后的模型一个单独 **warm-up** 阶段；
- 控制新增 **token** 数量，只扩那批真正高频且高价值的术语；
- 更密集地评估含新 **token** 的典型任务，而不是只看总体 loss。

6.5 分词器一致性

一旦分词器扩展，训练、评估、推理、日志分析、缓存、离线数据预处理等所有链路都必须统一更新。否则最容易出现的不是“性能略差”，而是样本和 **checkpoint** 根本不兼容：

- 同一句话在训练和推理时被切成不同 token；
- embedding index 对不上；
- 预处理缓存失效；
- 线上长度预算与离线估算不一致。

所以，分词器扩展不仅是一个建模问题，还是一个版本管理和部署一致性问题。

> **Note**
>
> 把贯穿案例放进分词器手术里看
> 内部代码助手里最容易触发扩词的是高频 API 名、配置键、路径片段和内部服务缩写。但这类 token 也最容易在 serving 侧出问题：一旦训练端和线上检索或缓存还在用旧 tokenizer，同一个函数名的切分就会前后不一致，模型“线下好、线上怪”的问题几乎必然出现。

7 梯度到底从哪里来？三种学习拓扑

我们第一次接触中训练、SFT、DPO 或 RL 时，会把注意力放在“数据长什么样”上：文档、对话、偏好对、工具轨迹。这样的直觉并不算错，但还不够深。

真正决定模型如何更新参数的，不只是样本内容本身，而是这几个更基础的问题：

1. 模型在每一个位置 到底看到了什么上下文；
2. 序列里的哪些位置 真的参与了损失函数；
3. 训练样本是 静态给定的，还是 由当前策略自己生成的。

Packing、Masking、Rollout 之所以重要，不是因为它们是三种“最流行的训练技巧”，而是因为它们分别控制了梯度形成链路中最上游的三个变量：

- 上下文拓扑（context topology）
- 监督拓扑（supervision topology）
- 数据生成拓扑（data-generation topology）

如果这三个环节里任何一个设计错了，训练依然会收敛，但模型很可能在学错东西。很多训练故障表面上像“数据质量一般”或“loss 不太稳”，本质上其实是 梯度来源错了。

> **Important**
>
> 本节先回答几个关键问题：
>
> 1. 为什么 Packing、Masking、Rollout 不在同一个抽象层级上，却要放在一起讲？
> 2. 为什么说它们不是简单地“改一点训练细节”，而是在改梯度的来源？
> 3. 在 CPT、SFT、DPO 和 RL 里，哪些 token 真正拿到梯度？
> 4. 为什么同样一段文本，换一种训练拓扑，模型学到的东西会完全不同？

7.1 一个统一框架：梯度到底从哪里来

先把不同训练方法放进一个统一的数学框架里。设模型参数为 ( )，训练序列为 (s = (s_1, s_2, ..., s_T))，则很多自回归训练都可以抽象成：

$$\mathcal{L}(\theta) = \mathbb{E}_{s\sim q}\left[\sum_{t=1}^{T} w_t\, \ell_t(\theta; s_{\le t})\right]$$

其中：

- (q) 表示 训练序列的分布；
- (s_{ t}) 表示第 (t) 个位置能看到的 上下文前缀；
- (w_t) 表示这个位置的 损失权重；
- (_t) 表示第 (t) 个位置对应的局部损失。

于是梯度就是：

$$g(\theta) = \nabla_\theta \mathcal{L}(\theta) = \nabla_\theta \mathbb{E}_{s\sim q}\left[\sum_{t=1}^{T} w_t\, \ell_t(\theta; s_{\le t})\right]$$

这个公式的意义非常大：训练中的参数更新，归根结底来自三件事。

1. 样本分布 (q)

你到底在从什么样的序列里采样？

- 是来自语料库的静态文本？
- 是领域文档与通用回放混合后的序列？
- 还是当前策略自己 rollout 出来的轨迹？

2. 上下文前缀 (s_{ t})

同一个 token，如果放在不同的上下文里被预测，它给出的梯度不会一样。

对 language model 来说，损失通常写成条件概率形式：

$$\ell_t(\theta; s_{\le t}) = -\log p_\theta(s_{t+1} \mid s_{\le t})$$

也就是说，模型不是在学习“孤立 token 是什么”，而是在学习“在这个前缀下，下一个 token 应该是什么”。所以只要前缀变了，梯度就会变。

3. 损失权重 (w_t)

不是每个 token 都必须被优化。某些位置可以参与损失，某些位置可以只是上下文但不参与监督。只要 (w_t) 变了，参数更新的来源也就变了。

7.2 三种最关键的“梯度入口”

从上面的统一框架看，Packing、Masking、Rollout 分别对应三种不同的上游控制：

- **Packing**：主要改的是 (s_{ t})，也就是每个位置的上下文；
- **Masking**：主要改的是 (w_t)，也就是哪些位置真的进损失；
- **Rollout**：主要改的是 (q)，也就是训练样本是谁生成的。

所以，更准确的表述不是“这三种方法改变梯度最多”，而是：

> 它们分别控制了梯度形成链路里最根本的三个变量：上下文、监督位置和样本来源。

这就是为什么它们值得放在一起讲。

> **Note**
>
> 一个更直观的例子
> 把训练想成老师批作业，会更容易理解这三种拓扑。
> **Packing**
> 相当于你把几份不同学生的作业钉在一起交给老师。老师会默认这些页面前后相连。于是，题目前后文被改写了。
> **Masking**
> 相当于你告诉老师：“只批最后一页，不用批前面草稿。”于是，不是所有内容都参与评分。
> **Rollout**
> 相当于你不再给老师固定答案，而是让学生现场解题，再按最终成绩给分。于是，作业是谁写出来的，以及怎么给分，都变了。
> 这个比喻并不严格，但它很好地对应了三者的本质差别：
> - Packing 改的是作业的排布；
> - Masking 改的是批改范围；
> - Rollout 改的是作业的生成方式。

7.3 Packing：改变上下文拓扑

Packing 到底是什么

Packing 本身不是一个新的优化目标，而是一种 数据布局策略。它最常见于预训练和 continued pretraining (CPT)：为了减少 padding 浪费，把多个较短文档拼进同一个固定长度的上下文窗口里。

设原始文档集合为：

$$D^{(i)} = (x_1^{(i)}, x_2^{(i)}, \dots, x_{n_i}^{(i)})$$

Packing 操作可以写成：

$$z = \mathrm{Pack}\left(D^{(i_1)}, D^{(i_2)}, \dots, D^{(i_k)}\right) = [D^{(i_1)}, \langle eos \rangle, D^{(i_2)}, \langle eos \rangle, \dots]$$

拼好之后，训练目标通常还是标准的 next-token loss：

$$\mathcal{L}_{\mathrm{pack}}(\theta) = -\sum_{t=1}^{T-1} \log p_\theta(z_{t+1} \mid z_{\le t}, A)$$

其中 (A) 是可选的 attention mask，用来控制文档边界是否允许跨越。

为什么它会改变梯度

关键不在于 loss 公式变没变，而在于 上下文变了。

如果模型原本是在单文档模式下训练，那么文档 B 的开头，只会在“文档 B 的起点”这个语境里被预测。可一旦你把文档 A 和文档 B pack 在一起，文档 B 的开头就会在“文档 A 的尾部之后”被预测。

也就是说，Packing 改的是：

$$s_{\le t}$$

它改变了每个 token 的条件上下文。

对语言模型来说，这种变化非常大，因为训练目标本来就是条件式的：

$$-\log p_\theta(x_t \mid x_{<t})$$

同一个 (x_t)，只要 (x_{<t}) 变了，梯度就不是同一个梯度。

例子：两个完全不相关的文档被拼在一起

假设有两篇短文档：

- 文档 A：患者主诉胸痛，持续两小时。
- 文档 B：本合同自签署之日起生效。

Packing 之后，序列可能是：

$$[\text{患者主诉胸痛，持续两小时。}, \langle eos \rangle, \text{本合同自签署之日起生效。}, \langle eos \rangle]$$

从吞吐量角度看，这很好：几乎没有 padding 浪费。
但从学习信号角度看，如果边界处理不好，模型会看到一种并不存在的统计关系：

“患者主诉胸痛，持续两小时。”后面很自然地接“本合同自签署之日起生效。”

这显然不是你想让模型学习的结构。

工程含义

Packing 的收益是 吞吐量，代价是 上下文拓扑被重写。因此它不是一个“纯粹的系统优化技巧”，而是一个会直接影响梯度的学习设计。

常见故障模式

- 文档边界泄漏（boundary leakage）
- EOS 分隔不一致
- attention mask 设计错误
- 长度分布变化导致 loss 统计和训练动态变化

Packing 改的是 token 在什么前缀下被预测。

7.4 Masking：改变监督拓扑

Masking 到底是什么

如果说 Packing 解决的是“样本怎么摆进去”，那么 Masking 解决的是“哪些位置算数”。

在监督微调（SFT）里，我们常常把一整段对话拼成一个序列，但并不希望所有 token 都参与损失。很多场景下，我们只希望模型学习 assistant 的输出，而不是要求它去“复述用户输入”。

设训练序列为：

$$s = (s_1, s_2, \dots, s_T)$$

定义一个 loss mask：

$$m_t \in \{0, 1\}$$

其中：

- $(\mathrm{m\_t} = 1)$：第 (t) 个位置参与损失；
- $(\mathrm{m\_t} = 0)$：第 (t) 个位置不参与损失。

那么训练目标可以写成：

$$\mathcal{L}_{\text{mask}}(\theta) = -\sum_{t=1}^{T-1} m_{t+1} \log p_\theta(s_{t+1} \mid s_{\le t})$$

为什么它会改变梯度

Masking 改的是每个位置的损失权重，也就是：

$$w_t = m_t$$

如果某个位置的 $(\mathrm{m\_t} = 0)$，那么该位置虽然存在于序列中，仍然作为上下文被模型看到，但它 不会贡献任何梯度。

这意味着同一段文本，只要 mask 方式变了，模型被要求负责的部分就变了。

例子：一条 **SFT** 对话样本

```
<system> 你是企业客服助手
<user> 帮我写一封退款邮件
<assistant> 当然可以，下面是一封简洁专业的退款邮件：
...
```

在典型的 assistant-only SFT 里：

- `system` token：作为上下文，但不参与损失；
- `user` token：作为上下文，但不参与损失；
- `assistant` token：参与损失。

换句话说，模型会利用 system 和 user 的内容来预测 assistant 的输出，但不会因为“没有把用户的话重复一遍”而被惩罚。

为什么这件事影响很大

因为它实际在定义：

> 模型到底要为谁负责。

- 在 CPT 里，几乎所有 token 都参与 next-token 预测；
- 在典型 SFT 里，只有 assistant token 参与监督；
- 在某些工具轨迹训练里，可能只监督工具调用和最终答案；
- 在结构化输出任务里，甚至可能只对 JSON body 的某些字段计算损失。

因此，Masking 改的不是“样本格式细节”，而是 监督边界。

常见故障模式

- mask 错位，导致 user token 也被监督；
- 模板拼接错误，assistant 第一段没被算进 loss；
- 多轮对话里部分回答被监督，部分回答漏掉；
- 工具调用 token 与自然语言 token 的监督策略不一致。

> **Masking** 改的是哪些位置的误差真的能进入梯度。

7.5 Rollout：改变数据生成拓扑

Rollout 到底是什么

Rollout 和前两者的本质差别在于：训练样本不再完全来自一个静态数据集，而是由当前策略模型自己生成。

设输入 prompt 为 (x)，策略模型为 $(\ \_\ )$，模型生成的轨迹为：

$$y = (y_1, y_2, \ldots, y_T) \sim \pi_\theta(\cdot \mid x)$$

然后环境、verifier 或奖励模型为这个完整轨迹给出回报：

$$R(x, y)$$

训练目标是最大化期望奖励：

$$J(\theta) = \mathbb{E}_{y \sim \pi_\theta(\cdot|x)}[R(x, y)]$$

一个典型的策略梯度形式可以写成：

$$\nabla_\theta J(\theta) \approx \mathbb{E}\left[\sum_{t=1}^{T} A_t \, \nabla_\theta \log \pi_\theta(y_t \mid x, y_{<t})\right]$$

其中 (A_t) 可以理解为 advantage 或 reward 相关的加权项。

为什么它会改变梯度

在监督学习里，训练分布 (q) 通常来自静态语料库；而在 rollout 里，训练序列是从当前策略 (_) 中采样出来的。也就是说，样本分布本身与参数有关：

$$q = q_\theta$$

这和前两种拓扑有本质区别：

- Packing 改的是上下文；
- Masking 改的是监督位置；
- Rollout 改的是 训练样本的来源机制。

例子：数据库查询与解释

假设任务是：

```
用户：去数据库里找出上个月 GMV 最高的 10 个商品，并给出解释。
```

如果是 SFT，你会给模型一条示范答案，让它模仿。
如果是 rollout，模型必须自己完成完整轨迹：

1. 写 SQL；
2. 调工具查询数据库；
3. 读取返回结果；
4. 写出解释；
5. 接受 verifier 或奖励函数评分。

奖励可能来自：

- SQL 是否可执行；
- 查询结果是否正确；
- 最终文字解释是否与返回数据一致；

- 是否调用了正确工具。

这里已经没有一个固定的“标准答案 token 序列”可供逐字模仿。模型必须先自己生成轨迹，梯度才会沿着那条轨迹回流。

为什么它影响更大

因为 rollout 把训练从“在固定数据上拟合”变成了“在自己的行为上优化”。这会带来几个连锁后果：

- 模型必须探索；
- 监督信号往往延迟到序列结束后才出现；
- credit assignment 变难；
- reward 噪声会通过整条轨迹传回梯度；
- 当前策略改变后，未来看到的样本也会改变。

Rollout 改的是样本由谁生成，以及奖励如何沿着轨迹回流到参数。

7.6 如何改变梯度

很多读者会自然地问：学习率、batch size、optimizer、temperature、采样比例这些东西，不也非常重要吗？

当然重要。但这些变量更像是在调节：

- 梯度有多大；
- 更新有多快；
- 噪声有多强；
- 训练有多稳。

而 Packing、Masking、Rollout 更像是在决定：

- 梯度 来自哪些样本；
- 梯度 在什么上下文下形成；
- 梯度 归属于哪些位置；
- 样本是 静态给定还是 策略自己探索出来。

这就是两者的差别：

- 学习率是在调 更新强度；
- 拓扑是在定 梯度支持集与来源机制。

因此，说这三种拓扑重要，不是因为它们“唯一重要”，而是因为它们处在 梯度形成链路的最上游。

维度	Packing	Masking	Rollout
核心问题	样本怎么排进上下文窗口	哪些位置参与损失	样本是谁生成的
主要改变的数学对象	(s_{ t})	(w_t)	(q)
是否需要探索	不需要	不需要	需要
监督是否立即可见	是	是	往往延迟

维度	Packing	Masking	Rollout
典型场景	预训练、CPT	SFT	RL、Agent training
它改写的拓扑	上下文拓扑	监督拓扑	数据生成拓扑

> **Packing** 决定 **token** 在什么前缀下被预测，**Masking** 决定哪些 **token** 的误差进入梯度，**Rollout** 决定训练样本由谁生成。

现在回头看常见训练阶段，会更清楚：

方法	典型数据形态	直接参与梯度的对象	学到的主要东西
CPT	原始文档、代码、长文本	几乎所有 token	参数化知识、领域分布、文档结构
SFT	对话示范、工具轨迹	通常是 assistant token	行为格式、指令跟随、工具使用模板
DPO / ORPO	chosen / rejected 偏好对	两条回答的相对偏好差异	回答质量倾向、偏好排序
RL	prompt + rollout + 奖励	策略采样轨迹上的 token	搜索、探索、长程策略改进

这张表背后的重点不是术语，而是：

> 不同训练阶段的差别，不只是“数据长什么样”，而是梯度到底从哪里来。

当训练结果不符合预期时，不要只问”数据对不对”，还要继续追问：模型在这些位置看到了什么上下文？哪些位置真的参与了损失？样本到底来自静态语料，还是来自当前策略自己的行为？很多所谓”模型学坏了”的问题，不是优化器太差，也不是 loss 不够先进，而是 Packing、Masking、Rollout 这三个梯度来源里至少有一个答案错了。

8 怎么把整套东西放进训练系统：并行、恢复、版本绑定与可审计

> **!** Important
>
> 本节先回答几个关键问题：
> 1. CPT 在系统上到底是如何从 pretrained checkpoint 恢复的？
> 2. data parallel、tensor / model parallel、sequence parallel 在中训练里各自解决什么问题？
> 3. 为什么分词器版本、数据版本、restore mode 必须和 checkpoint 绑定管理？
> 4. 什么样的 pipeline 才算可回滚、可复现、可审计？

中训练是一次“再配置过的继续训练”，而不是把旧脚本换一批数据继续跑。
系统实现决定了实验能不能复现、checkpoint 能不能回滚、回归能不能定位、线上行为能不能

追溯。很多团队以为自己在讨论模型，最后真正把 run 拉开的，却是恢复方式、数据版本、并行栈和日志治理。

8.1 怎么恢复，恢复什么

CPT 几乎总是从一个已有的 pretrained checkpoint 开始，但“恢复 checkpoint”有两种含义：

1. 只恢复模型权重：最常见于拿开源模型继续训练，此时优化器状态通常不可用，需要重新初始化优化器与 scheduler；
2. 恢复完整训练状态：如果是在自有预训练流水线上继续训练，可以同时恢复优化器、调度器和随机状态，让 CPT 更像“中断后续跑”。

把训练状态写得严格一点，可以把一次恢复看成对下面这个状态集合的选择：

$$\mathcal{S} = \{\theta, \phi_{opt}, \psi_{sched}, \xi_{rng}, \tau_{tok}, \nu_{data}\}$$

其中：

- θ：模型权重；
- ϕ_{opt}：优化器状态；
- ψ_{sched}：学习率调度状态；
- ξ_{rng}：随机数状态；
- τ_{tok}：分词器版本；
- ν_{data}：数据版本与采样配置。

于是两种常见恢复方式可写成：

$$\mathcal{R}_{weights} = \{\theta\} \qquad \mathcal{R}_{full} = \{\theta, \phi_{opt}, \psi_{sched}, \xi_{rng}, \tau_{tok}, \nu_{data}\}$$

这两种方式在工程含义上不同。只恢复模型权重时，你必须更小心学习率和 warm-up，因为优化器对当前参数几乎没有历史感知；而恢复完整状态时，虽然更连续，但你也要确认新分布与旧优化状态是否匹配。

8.2 weights-only 与 full-state 的操作差异

这个区别在真实项目里常常被低估。**weights-only resume** 和 **full-state resume** 不是同一个 **recipe**。

- **weights-only**：更像“拿一个成品模型重新开一个新实验”；
- **full-state**：更像“在原训练轨道上继续跑”。

因此，当只恢复权重时，工程上通常需要：

- 更低的初始 LR；
- 重新 warm-up；
- 更密的 early eval；
- 把它当成一个新 run，而不是“原 run 的后半段”。

而当恢复 full-state 时，工程上更需要警惕：

- 旧 optimizer 的动量是否适合新分布；
- scheduler 是否会把 LR 带到一个不适合 CPT 的区间；
- 数据版本和分词器是否保持严格一致。

8.3 怎么并行

对大模型而言，CPT 通常沿用预训练期的分布式训练栈，只是预算更小、run 更短、评估更密。常见角色如下：

- **Data Parallel / FSDP / ZeRO**：把样本批次拆到多卡，分摊参数、梯度或优化器状态；
- **Tensor Parallel**：把单层张量切到多卡，适合超大模型；
- **Pipeline Parallel**：把层切到不同设备，适合非常深的模型；
- **Sequence Parallel**：在长上下文或大 batch 条件下分担序列维度压力；
- **Activation Checkpointing**：用更多重算换更低显存。

具体内容，请翻阅第二章。

在中训练里，它们不是“越多越好”，而是由模型规模、上下文长度和集群结构共同决定。CPT 常见的工程判断是：既然 **run** 短、评估密，就要尽量降低系统复杂度，避免为了极限吞吐引入过多并行层级。

8.4 怎么绑定版本，怎么保证可复现、可审计、可回滚

1. 数据版本与分词器版本必须跟 checkpoint 绑定。否则模型回归时无法定位到底是数据变了还是代码变了。
2. checkpoint 频率要比预训练更密。因为 CPT 的目标不是长期收敛，而是找到一个领域收益与通用退化平衡更好的点。
3. 评估必须和 checkpoint 同步。没有相邻 checkpoint 的评估，很难解释 U 形曲线。
4. 保持一个不可变的 base checkpoint。这样你总能回到清洁起点，而不是在多个“半适配”的分支上迷路。
5. I/O 管线要跟上。** 很多中训练瓶颈不是算力，而是长文档读取、去重、packing 和数据加载本身。

9 CPT 与后续 SFT / DPO / RL 的兼容性

! Important

本节先回答几个关键问题：

1. 为什么中训练会影响后续的 instruction tuning、RLHF、DPO 和 Agentic RL，而不是一个独立的前置步骤？
2. 什么样的 CPT 会帮助后续 alignment，什么样的 CPT 会拖后腿？
3. 为什么过度领域化会伤害后续对齐训练？
4. 给定一个真实产品需求，如何在 prompting → RAG → SFT → DPO →

RL → CPT → 蒸馏之间做决策？

5. 为什么“先做一个更懂领域的模型”仍然不等于“已经做好了产品对齐”？

CPT 改变的是模型的起点分布，而后续 **alignment** 改变的是这个起点上的行为。如果中训练把模型带到了一个更贴近目标任务的分布，SFT 和 RL 往往更容易学；但如果 CPT 把模型拉得过窄、过于模板化、过于远离通用对话分布，那么后续对齐反而会更吃力。

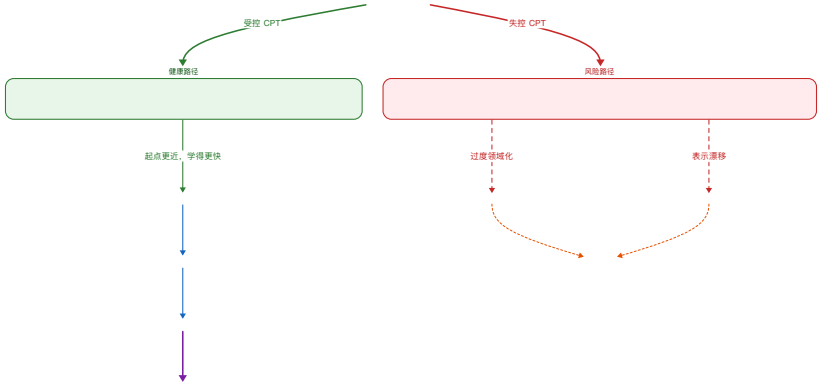

Figure 8: 图 5 · CPT 与后续 SFT / DPO / RL 的兼容性

9.1 为什么 CPT 会影响后续 SFT / RL

后训练方法并不是凭空创造能力，它们更像是在已有能力上做重加权或行为塑形。一个模型如果在目标领域、长上下文、代码痕迹、工具文本或数学表述上已经有一定先验，那么：

- **SFT** 更容易把这些能力塑形成稳定行为；
- 偏好优化 更容易在多个“都不差”的回答之间学到好坏排序；
- **RL / Agentic RL** 更容易从 rollout 中获得有用的奖励信号，因为候选解本身已经更像样。

反过来，如果 base model 对任务分布几乎没有任何先验，让 RL 直接去学，常常会遇到奖励稀疏、探索成本高、rollout 太差的问题。也就是说，**CPT** 可以提高后续 **RL** 的可学性，但不能替代 **RL** 本身。

9.2 两条最常见的失败路径

CPT 到底怎样把后续对齐变难?

失败路径 **1**：过度领域化 → 对话退化 → **SFT** 债务上升

如果领域分布过窄、replay 太低，模型会开始把一切都看成法律、医疗或代码问题。其后果不是单一 benchmark 下降，而是整个聊天分布开始退化：

- 普通用户问题也被回答得过于术语化；
- 输出风格更像文档续写而不是助手回复；
- 结构化输出和工具模板被领域文本风格冲掉。

从系统角度看，这意味着你把本来应该做“轻量 SFT”的问题，变成了“先用 SFT 把模型拉回能聊天的状态，再做真正的产品行为监督”。这就是 **SFT debt（SFT 债务）**。

失败路径 **2**：表示空间漂移 $\rightarrow$ **rollout** 脆弱 $\rightarrow$ **RL** 不稳定

即使表面指标没那么差，CPT 也可能通过表示漂移增加后续 RL 的脆弱性。直觉上，如果模型在 CPT 中被高度模板化数据强行拉向一个狭窄子空间，那么 RL rollout 会出现几个问题：

- 中间推理步骤更僵硬，探索空间更小；
- verifier 能给出的高奖励轨迹更少，奖励更稀疏；
- 小的参数更新更容易造成风格塌缩或工具使用波动。

如果要把这种“离得太远”的风险写成一个简化量，可以在某个固定聊天分布 D_{chat} 上测：

$$\mathrm{Drift}_{chat}(\theta_{cpt}) = \mathbb{E}_{x\sim D_{chat}}\left[\mathrm{KL}(p_{\theta_{base}}(\cdot \mid x) \,\|\, p_{\theta_{cpt}}(\cdot \mid x))\right]$$

它不必作为唯一指标，但可以帮助你理解一个原则：领域涨得再多，如果它伴随聊天分布和对齐分布上的大幅漂移，后续成本可能反而更高。

9.3 一个实用的决策顺序

给定产品需求时，较稳的决策顺序通常是：

1. 先试 **prompting / system design**：如果能力本来就在，只是没被稳定触发，不要急着训练；
2. 知识变化快时优先 **RAG**：把事实留在外部记忆里，比写进参数更可维护；
3. 行为格式不稳时做 **SFT**：输出 schema、角色风格、工具模板都是这一层的事；
4. 回答质量排序不稳时做 **DPO / RLHF**；
5. 需要搜索、长程策略和 **verifier** 优化时做 **RL**；
6. 根本缺的是领域分布和参数化知识时，再做 **CPT**；
7. 质量够了、成本过高时，最后做蒸馏或压缩。

这个顺序看起来像流程图，但它背后其实是一条成本原则：先动最便宜、最贴近缺口的杠杆。

> **Note**
>
> 把贯穿案例放进兼容性里看
> 对内部代码助手来说，CPT 可能让模型更熟悉仓库，但不会自动让它学会“先检索、再解释、最后按团队 patch 模板输出”。这些仍然是 SFT 的工作。相反，如果 CPT 让模型开始把普通用户问题也回答成内部 runbook 风格，后续对齐的代价会立刻上升。

10 工程案例

> **!** Important
>
> 本节先回答几个关键问题：
>
> 1. 域内任务涨分、通用能力掉分时，最典型的根因是什么？
> 2. 为什么分词器扩展经常在离线指标正常、线上行为异常时才暴露问题？
> 3. aggressive learning rate 会怎样把一次本来可控的 CPT 变成训练发散？
> 4. 为什么“领域数据更多了”并不总是等于“系统整体更好了”？
> 5. 哪些失败一开始看起来像小 bug，最后却直接进入梯度？

下面这些不是抽象概念，而是中训练项目里非常常见的工程故事。每个故事都用同一个结构说明：症状 $\longrightarrow$ 根因 $\longrightarrow$ 修复。

案例 **1**：法律任务涨了，通用问答却崩了

症状：法律检索和合同摘要明显提升，但开放域问答、MMLU 和一般聊天质量同时下降。

根因：训练配方几乎完全由法律文档组成，replay 比例过低，模型在短时间内被窄分布拉偏。团队只盯着领域 benchmark，直到产品灰度后才发现通用回归。

修复：提高通用回放比例，降低学习率，把通用与安全评估放进回归门控，并将 checkpoint 选择标准从“领域最高分”改成“领域收益 / 通用回归的 Pareto 最优点”。

案例 **2**：扩词后离线 **loss** 正常，线上代码生成却更差

症状：新增了一批代码 API token 后，训练 loss 看起来正常，但线上生成经常出现奇怪的半截标识符和不稳定补全。

根因：新 token 虽然加入词表，但相关样本频率不足，embedding 行训练不足；同时，部分离线评估仍然使用旧 tokenizer，导致训练—评估—部署不一致。

修复：统一分词器版本，对含新 token 的样本做阶段性过采样，单独增加包含 API 名和路径的回归集，并在 serving 侧清理所有旧分词器缓存。

案例 **3**：做完医疗 **CPT** 后，模型更懂术语了，却不会和用户说话了

症状：模型在病历术语、实体识别和医疗长文档摘要上显著提升，但多轮对话体验变差，回答风格更像文档续写而不是助手回复。

根因：团队把 CPT 当成了产品化步骤，忽略了中训练只学分布、不学交互行为。模型被推向医疗文档风格，却没有再经过足够的 SFT 把它拉回聊天分布。

修复：在 CPT 后补充高质量医疗问答和摘要类 SFT，并单独评估对话行为、拒答边界和结构化输出质量。

案例 **4**：训练在前几亿 **token** 看起来正常，后面突然发散

症状：前期 loss 平稳下降，随后某一阶段 loss、梯度范数和通用评估同时恶化，后续 checkpoint 全部变差。

根因：学习率设得过高，加上一个数据子集存在错误拼接和重复模板，导致模型在少量异常 batch 上出现大的优化冲击。团队因为 checkpoint 间隔过大，没能及时定位拐点。

修复：缩短 checkpoint 与评估间隔，降低峰值 LR，增加 gradient clipping，重新清洗异常子集，并在数据加载阶段增加长度、字符集和模板检测。

案例 **5**：领域任务提升了，但后续 **RL** 反而更难扩

症状：中训练后代码与数学 corpus 上的 loss 下降了，但后续基于 verifier 的 RL 提升幅度不如预期，rollout 风格也更僵硬。

根因：CPT 数据高度模板化，模型学到了较强的局部格式偏好，却没有学到足够多样的中间解空间，导致后续探索空间变窄。

修复：提高数据多样性，混入更自然的推理和工具文本，在 CPT 阶段保留足够 replay，并在 RL 前先用 SFT 恢复更稳定的交互与中间步骤格式。

案例 **6**：**packing bug** 让一个文档的引用“流血”到另一个文档

症状：法律模型在摘要时开始把上一份合同的引文片段带到下一份合同里，像是在“跨文档记忆”。

根因：document packing 时只做了简单拼接，没有在边界上使用足够明确的 EOS / attention 约束，模型把跨文档衔接当成了真实统计模式。

修复：增加显式边界、修正 attention mask、重新构建 cross-document contamination regression set，并回滚到未受污染的 checkpoint。

案例 **7**：同样的 **base weights**，外部复现却和内部 **run** 完全不一样

症状：团队拿到同一个基座模型重新做 CPT，结果与原团队内部 run 的稳定性和收敛点明显不同，怀疑“是不是数据有问题”。

根因：内部 run 恢复了 optimizer / scheduler / RNG 的 full state，而外部复现只恢复了 model weights；两边实际上跑的不是同一个 recipe。

修复：显式记录 restore mode，把 weights-only resume 当作一个新实验重新调 LR / warm-up，并在实验元数据中把 tokenizer、data mix、optimizer state 一起版本化。

案例 **8**：领域化做得很好，拒答边界却悄悄松了

症状：领域模型在专业问题上更自信，但对不该回答的问题也更愿意硬答，安全评估直到后期才发现劣化。

根因：replay 太低，安全相关的通用底盘被削弱；同时 regression gates 里没有单独的 safety slice，团队只监控了领域任务和通用 QA。

修复：提升 replay，把安全评估加入硬门控，并预留 post-CPT 的对齐工作，而不是假设“领域能力上涨会自动带来更好行为”。

11 本章小结

CPT 应该被当成一项受约束的工程变更，下列原则值得被当成实践清单：

1. 先判断是不是知识缺口，再决定是否动用 **CPT**。不要用继续预训练去解决本应由 SFT、RAG 或 prompting 解决的问题。

2. 数据分布比原始数据量更重要。领域数据再多，如果充满模板、重复和脏样本，也只会把错误放大。
3. 默认带 **general replay**。只在极少数确有把握的场景下，才尝试几乎纯领域化的配方。
4. 学习率要比预训练更保守。对已经成型的表示空间，剧烈更新往往弊大于利。
5. **packing** 是吞吐优化，不是免费午餐。文档边界处理不当，会把不存在的模式写进参数里。
6. 分词器扩展要克制。只有在高频术语碎片化已成为明显质量或成本瓶颈时，才值得动它。
7. **checkpoint**、数据版本、分词器版本必须绑定管理。否则回归无法定位。
8. 评估要双向看。既看领域收益，也看通用、安全、行为和后续对齐能力。
9. **CPT** 不是产品化终点。很多系统在 CPT 之后仍然需要 SFT、偏好优化和推理层控制。
10. 能用更便宜的方法，就不要先上训练。对工程团队来说，最低成本的有效解始终优先。
11. 把 **restore mode** 当成实验变量。 weights-only 和 full-state resume 需要不同的 LR / warm-up 与复现预期。
12. 给每次 **run** 一个明确的 **stop policy**。不要把“也许再跑一会儿就恢复”当成训练策略。

11.1 问题小结

1. 什么是 **continued pretraining（CPT）**？它与预训练和 **SFT** 有何不同？

CPT 是在一个已经预训练好的基座模型上，继续使用下一个 token 目标做训练，但训练分布换成更靠近目标领域的数据。它和预训练的主要区别不在损失函数，而在数据分布、训练预算和稳定性要求；它和 SFT 的区别在于，CPT 学的是参数化知识和领域先验，SFT 学的是行为映射和输出格式。

2. 什么时候该用 **CPT**，而不是提示工程、**RAG** 或 **SFT**？

当问题是术语、实体、长文档结构和领域分布本身不在模型里时，用 CPT；当能力已潜伏、只是没被稳定触发时，先用 prompting；当知识更新快且需要可追溯时，用 RAG；当模型知道事实但不会按要求输出时，用 SFT。

3. 模型在法律文档上困惑度很高，但通用性能不错，优先修哪一层？

优先考虑 CPT，因为这说明模型没有很好建模法律文本分布。SFT 可以改变回答方式，但通常不能弥补术语共现、引文格式和长文档结构先验的缺失。

4. 为什么 **same objective, different distribution** 会在实践中产生巨大差异？

因为模型学习的是条件概率分布。目标函数不变，但如果样本分布从通用网页换成法规、病历或代码库，模型就会重新估计哪些 token、结构和实体更常见，从而改变参数化知识和语言先验。

5. **general replay** 为什么重要？

replay 是在领域适配时保护通用能力的最低成本手段。它既减少灾难性遗忘，也能缓和新分布带来的优化冲击，还能保留后续 SFT 和 alignment 所依赖的通用底盘。

6. 如何设计 **CPT** 的 **regression suite** 和 **stop criteria**？

至少包含四类评估：领域、通用、行为、安全。stop criteria 不应只看“领域涨没涨”，而应同时包含通用 / 安全回归阈值、领域收益停滞窗口，以及单位 token 收益是否还值得继续投入。

7. **CPT** 中的稳定性鸿沟是什么？

稳定性鸿沟指模型在进入新分布训练后，领域指标上升，但通用能力先明显下跌的现象。它通常由过高学习率、过窄分布、replay 不足和脏数据共同造成；应对方式是更保守的 LR、更稳的 mixture、更密的评估和明确的回滚策略。

8. 什么时候应该扩展 **tokenizer**？

当目标领域里存在大量高频、高价值、但被严重碎片化的 token，例如药名、法律引文、代码标识符或新语言脚本，而且这种碎片化已经明显影响上下文成本和质量时，才值得扩词。否则，优先考虑直接用 CPT 学习旧分词下的表示。

9. 新增 **token** 的 **embedding** 怎么初始化更稳？

一般会用组成该 token 的旧 sub-token embedding 的均值或组合初始化，而不是纯随机初始化。这样可以让新 token 从一个更接近原语义位置的起点开始训练，降低早期噪声和不稳定。

10. **weights-only resume** 和 **full-state resume** 有什么操作差异？

weights-only 更像在已有参数上重新开一个新实验，通常需要重新 warm-up 和更保守的 LR；full-state 更像原训练轨道的延续，但也更依赖旧 optimizer / scheduler 状态与新分布的兼容性。两者不应被视为同一个 recipe。

11. **CPT**、**SFT**、**DPO** 和 **RL** 的核心差别是什么？

核心差别在于数据构造和梯度来自哪里。CPT 对几乎所有 token 做 next-token 学习；SFT 主要在 assistant token 上施加监督；DPO 比较 chosen 和 rejected 的相对偏好；RL 优化策略采样出来的 rollout 的期望奖励。

12. 为什么中训练会影响后续 **alignment** 和 **RL**？

因为 CPT 改变了模型的初始化分布。一个更贴近目标领域的模型，会让后续 SFT、DPO 和 RL 更容易学；但如果 CPT 过度领域化、损伤通用对话或安全边界，后续对齐的成本反而会上升。

13. 如何 **debug** 一个“领域 **perplexity** 变好了，但工具使用和结构化输出变差”的模型？

先判断问题是不是行为回归而非知识回归：检查 SFT 模板是否被 CPT 冲掉、behavior eval 是否缺失、replay 是否过低；然后核查分词器/ schema 兼容性、packing 污染以及是否需要在 CPT 后追加针对工具轨迹的 SFT。

14. 给定产品需求，如何在 **prompting** $\rightarrow$ **RAG** $\rightarrow$ **SFT** $\rightarrow$ **DPO** $\rightarrow$ **RL** $\rightarrow$ **CPT** $\rightarrow$ **distill** 之间做决策？

我会先定义目标行为和约束，再诊断缺口类型。能不训练就先不训练；知识需要外部更新时优先 RAG；行为格式问题优先 SFT；偏好排序问题用 DPO / RLHF；长程搜索与 verifier 优化问题用 RL；只有当模型真的缺领域知识和分布先验时才上 CPT；最后如果质量够了但成本过高，再做蒸馏和压缩。

中训练的难点，从来不只是“继续训练”四个字，而是如何让继续训练仍然可控、可回滚、可解释、可接入后续系统。从本章的角度看，一个成熟的 AI 工程师至少要掌握四个判断：

- 什么时候面对的是知识缺口，而不是行为缺口；
- 如何通过 replay、mixture、packing 和门控做受控领域适配；
- 为什么 tokenizer、checkpoint 和训练拓扑会影响实际稳定性；
- 为什么 CPT 的价值不止在本身，还在于它如何为后续 SFT、DPO、RL 和部署留下更好的起点。

当你能够用这些判断回答“如何在不破坏既有能力的前提下适配新领域”时，你就真正掌握了 CPT 作为一项工程系统。

下一章我们介绍如何把中训练完成的基础模型变成助手。