

从调 API 到训模型

从 AI 应用工程师到应用架构师、再到能训专用模型的算法岗，一条递进的路

竹小竹8167

费曼式写法 · 由多个 AI 代理撰写与互相审校

导读

从 AI 应用工程师到应用架构师、再到能训专用模型的算法岗，一条递进的路

先别问要学哪些课、考哪些证。先想象一个场景：

你接手了一个上线三个月的 AI 客服系统。它大体能用，但有三件事悬着：账单每个月翻倍地涨；有一类用户问题答错率居高不下，提示词改了七次都没根治；老板上周问了一句「我们能不能不用别人的模型，自己搞一个」。

面对这三件事，不同段位的人看到的是完全不同的东西。刚入行的人看到三个孤立的麻烦；这本书想让你看到一张连起来的地图：账单问题是**成本架构**问题，答错率问题的尽头可能是**提示词到顶、该动模型了**，而老板那个问题的答案取决于你对**数据、微调和算力账**的判断——三件事串在一条路上。

这本书讲的就是这条路：**从会调 API 的 AI 应用工程师，到能设计整个系统的应用架构师，再到能训练专用模型的算法岗**。我们不罗列工具清单——工具半年一换。我们讲每一级台阶上那些不变的权衡：一次调用里每个旋钮在拧什么；提示词为什么该当接口契约来写；上下文、工具、评估、可靠性、成本这些系统层的真问题；以及那个分水岭判断——什么时候提示词到顶了、该微调了；再往后，数据怎么造、模型怎么训、自训的账怎么算。

每一章只讲透一件事，章节之间是递进的：这一章的结尾，就是下一章的问题。你可以从中间任何一章进来——如果你是架构师候选，直接从第六章的评估开始也未尝不可；但顺读会多一层「原来这些坑都连着」的乐趣。

读者画像很明确：写过代码、调过模型 API、想再上一层楼的人。不讲数学推导，不堆名词，每个概念都当场用大白话讲清楚。读完你能回答一个问题：**面对任何一个 AI 系统的决策点，该问哪些问题、用什么换什么。**

路从最小的地方开始：一次 API 调用。

第一章 · 台阶在哪里

先问你一个问题：同样一个需求——「给我们的客服系统加一个 AI 自动回复」——一个 AI 应用工程师和一个 AI 应用架构师，拿到的反应有什么不一样？

应用工程师想的是：**怎么把它做出来**。调哪个 API、提示词怎么写、回复格式怎么解析、错了怎么重试。这些是「实现」层的问题，答案大多有对错。

架构师想的是另一组问题：**这个东西该不该这样做**。客服场景里答错的代价是什么？是答非所问让人烦，还是报错退款金额造成真金白银的损失？需要多高的正确率才够上线？用贵的大模型还是便宜的小模型，省下的钱够不够覆盖错误率的上升？这个问题到底是提示词能解决的，还是要喂更多上下文，还是——到了某个点——应该自己训一个专用模型？

大白话

大白话：工程师回答「怎么做」，架构师回答「做不做、做成什么样、用什么换什么」。后者没有标准答案，只有权衡。

三个岗位，三种判断力的范围

把这条路上的三个台阶摆在一起看：

- **AI 应用工程师**：把模型接进产品。判断力覆盖「一次调用」——提示词、参数、解析、重试。产出是可用的功能。
- **AI 应用架构师**：设计整个 AI 系统。判断力覆盖「一个系统」——多个模型怎么编排、数据怎么流、成本怎么算、可靠性怎么保证、什么时候该微调。产出是能在真实世界长期跑、扛得住流量和意外的系统。
- **算法岗（训专用模型）**：当现成模型不够用，自己造一个。判断力覆盖「一个模型本身」——数据怎么造、训练怎么配、训出来的模型到底变好了没有、值不值那笔算力钱。产出是别人训不出来的能力。

注意一个规律：台阶越高，**判断力的范围越大，但每一级都建立在下一级之上**。没见过提示词怎么在线上崩掉的人，设计不出可靠的系统；没算过推理成本账的人，判断不了「自己训

一个模型」到底划不划算。所以这不是三条平行的职业路线，而是一级一级叠上去的。

为什么差的不是「会的工具多」

很多人以为升级就是学更多东西：学 LangChain、学向量数据库、学 PyTorch、学 CUDA……工具清单越列越长，人越来越焦虑。

但工具是结果，不是原因。架构师和工程师真正的差别在于：**架构师脑子里有一张「代价地图」**。他知道每个选择背后牺牲了什么。用更大的模型，牺牲的是钱和延迟；用更复杂的智能体编排，牺牲的是可调试性；自己训模型，牺牲的是时间和维护成本，换来的是效果、成本和数据主权。

大白话

大白话：工具半年一换，权衡十年不变。这本书只讲权衡，工具顺带提。

贯穿全书的一个问题

这本书要回答的问题只有一个：

一个现在只会调 API 的应用工程师，怎么一步步长出「这个系统该这样设计、这个模型该这样训」的判断力？

我们的走法是顺着判断力范围的扩大走：先吃透**一次调用**里每个旋钮到底在拧什么（第二章）；再把它当成**接口设计**来对待（第三章）；然后把视野扩大到**上下文、工具、评估、可靠性、成本**这四到八章的系统层面——到这里，你已经具备应用架构师的完整视野。第九章是一个分水岭：什么时候提示词这条路走到头了？答案指向**数据、微调、自训专用模型**（十到十二章），也就是算法岗的核心。最后两章把整条线收拢成一张能力地图。

每一章只讲透一件事，并且回答上一章留下的疑问。现在，从最小的一次调用开始。

第二章 · 把一次调用用明白

一个 AI 应用工程师的日常，看起来就是一行代码：把提示词发给 API，拿回一段文本。但这一行代码里藏着四五个旋钮，绝大多数人从头到尾没拧过，只是用了默认值。架构师和工程师的第一个分水岭就在这里：**你能不能说出每个旋钮拧动时，模型内部发生了什么。**

token：模型眼中的「字」不是字

模型读的不是汉字或单词，而是 token——把文本切成的小片段，每个片段对应词表里的一个编号。切法很务实：常见词整个算一个 token，生僻词拆成几块。英文里 `unbelievable` 可能被切成 `un`、`believ`、`able` 三段；中文里一个汉字通常占一到两个 token。

为什么工程师必须在乎这个？因为**计费、限速、窗口长度，全都按 token 算**，不按字算。你估算成本时用「这段文字大概三百字」是错的；而且切分有反直觉的后果：模型做算术差，部分原因就是「12345」可能被切成奇怪的几段，它从没把「12345」当成一个整体数字见过。

大白话

大白话：token 是模型的「像素」。你看的是一张图，它看的是一格一格的色块。账也是按色块算的。

上下文窗口：一块会会计费的桌面

上下文窗口（context window）是模型一次能看到的全部文本——你的系统提示、对话历史、塞进去的资料，加起来不能超过这个上限。它是模型唯一的工作记忆：窗口之外的东西，模型一个字都不知道。

两个工程后果常被忽视。第一，**窗口大不等于用得好**：塞进二十万字，模型对中间部分的注意力会变稀，答案质量常常反而下降，这就是所谓的「迷失在中间」现象。第二，**窗口里的**

每个 token 每次请求都要重新付一遍钱。多轮对话里历史越滚越长，如果不做裁剪，第十轮对话的成本可能是第一轮十倍。

温度：不是「创造力」，是「掷骰子的歪斜程度」

模型生成文本的方式，本质是每一步算出下一个 token 的概率分布，然后从中抽样。温度（temperature）拧的是这个分布的形状：温度低，概率最高的那几个 token 几乎垄断选择权，输出稳定、趋同；温度高，长尾上的怪词也有机会被抽中，输出多变、也容易跑偏。

大白话

大白话：温度不是「聪明程度」也不是「创造力」，是骰子的公平程度。温度 0 像灌了铅的骰子，永远落在最可能的选项上。

工程含义很直接：**凡是要进程序、要被解析的输出，温度调低**——你要的是可重复，不是惊喜。文案头脑风暴可以调高。线上事故里有一大类，就是有人把温度留在默认值，结果同样的输入昨天能解析、今天解析炸了。

结构化输出：把「求你了」变成「保证」

应用工程师最常干的事是让模型输出 JSON 给程序用。早年的做法是提示词里写「请只输出 JSON，不要别的」——这是请求，不是保证，模型随时可能送你一句「好的，以下是结果：」然后整个解析器崩溃。

现在的主流 API 提供 结构化输出（structured output）：你把 JSON Schema 传给接口，模型在生成时被**约束解码**——每走一步，不允许出现在语法上走不通的 token 会被直接屏蔽。这不是「模型更听话了」，而是采样时被硬约束在合法空间里，格式错误从物理上不可能发生。

架构上的意义：可靠性问题一旦能被机制保证，就不要靠提示词祈祷。这条原则在第七章还会回来。

这一章的账

回头看，一次调用里真正属于你的决策有：塞多少上下文（成本与质量的权衡）、温度定多少（稳定性与多样性的权衡）、用什么输出约束（祈祷与保证的权衡）。把这三个权衡说清楚，你就已经超过大多数「会调 API」的人了。

但旋钮拧得再好，提示词本身写得一团糟也没用。下一章的问题是：提示词到底是个什么东西？答案是一一它不是咒语，是一份接口契约。

第三章 · 提示词是接口设计

上一章我们把一次调用的旋钮讲清楚了。但旋钮之外，还有一个更大的东西：提示词本身。网上流传的提示词技巧——「你是世界顶级专家」「深呼吸，一步步来」「答对了给你二十美元小费」——让这个领域看起来像巫术。这一章要破除的就是这个印象：**提示词不是咒语，它是一份接口契约。**

咒语的失效方式

咒语式提示词有三个共同的死法。

第一，**只对某个版本有效**。「你是顶级专家」在 GPT-4 某一代上可能让输出略微认真一点，换个模型版本就毫无作用甚至起反作用——因为它的机制是碰巧拨动了训练数据里的某种统计关联，而这种关联每次训练都会重洗。

第二，**不可回归**。你改了第五行的措辞，回答变好了还是变坏了？不知道。没有评估，改提示词就是在黑暗里拧旋钮（第六章会专门解决这个）。

第三，**不可维护**。一个塞了八百行「不要这样、不要那样」的提示词，每条禁令都是某次线上事故留下的创可贴，没人敢删，也没人知道哪条还在起作用。

大白话

大白话：靠撞大运写出来的提示词，就像靠试出来的偏方——可能有效，但你不知道对什么有效、为什么有效、什么时候失效。

契约式写法：角色、输入、输出、边界

把提示词当成一份你和模型之间的接口契约来写，结构就清楚了。一份能进生产环境的提示词通常有四块：

- **角色与任务**：一句话说清要干什么。「你是客服系统的回复生成器，负责根据工单内容起草回复。」不是吹捧，是限定职责范围。

- **输入的字段约定**：明确告诉模型会收到什么。用分隔符或 XML 风格的标签把变量包起来，如 `<ticket>...</ticket>`，模型对结构化分隔的遵循度远高于裸文本拼接。
- **输出的格式约定**：能走结构化输出就走结构化输出（第二章），走不了就给出精确的格式示例。一个具体的例子，胜过三段形容词。
- **边界与兜底**：明确「不知道时怎么办」。比如「工单信息不足以回答时，输出 `escalate=true`，不要编造政策细节」。这一条省掉的是幻觉造成的真金白银。

契约思维和咒语思维的分水岭在于：契约的每一条款都回答「这条违约了会怎样」。如果删掉某一句不会让任何输出变差，那它就是噪音，删掉它——长提示词不仅贵，还会稀释模型对关键指令的注意力。

例子：少样本提示是契约里的附件

少样本提示（few-shot prompting）是在提示词里放几个「输入 → 正确输出」的样例。它的机制不是「教模型学会新技能」，而是**把输出的分布钉在你展示的那个格式和风格上**——模型极其擅长模仿模式，样例就是最明确的模式。

所以样例的选择标准是覆盖面，不是数量：三个覆盖不同情况的样例，好过十个雷同的。边缘情况（输入为空、输入自相矛盾、用户骂人的情况）尤其值得各给一个样例，因为这些正是模型自由发挥时最容易翻车的地方。

提示词也要过代码评审

契约的最后一个属性是**可管理**：提示词应该进版本库、有变更记录、过评审、配套评估用例。改一行提示词和改一行代码同级对待——因为它们对系统行为的影响是同级的。

大白话

大白话：提示词是这个系统里唯一用自然语言写的代码。是代码，就要有代码的纪律。

到这里，「问什么、怎么问」讲完了。但很快你会发现一个更扎心的事实：模型答得不好，常常不是问得不对，而是**它手里根本没有那份资料**。下一章进入上下文工程——窗口里塞什么，比怎么问更重要。

第四章 · 上下文工程

第二章说过，上下文窗口是模型唯一的工作记忆。那么一个明显的推论是：模型答不出来，十有八九不是它不会，而是**需要的那份资料此刻不在窗口里**。公司内部的政策文档、用户三个月前的工单、昨天晚上更新的价格表——模型训练时一个都没见过。把对的资料在对的时刻塞进窗口，这件事有个名字：上下文工程（context engineering）。它是 AI 应用架构师日常花精力最多的一件事。

RAG：开卷考试，不是背书考试

最经典的上下文装配方案是 RAG（检索增强生成）：不靠模型背知识，而是提问时先去资料库里**检索**出相关的几段，贴进提示词，让模型照着手边的资料回答。就像把闭卷考试改成开卷考试——模型不用记住整本手册，只要会查、会照着答。

检索这一步怎么运作？主流做法是**向量检索**：把每段资料用一个嵌入模型压缩成一串数字（向量），语义相近的段落，数字也相近。提问时把问题也压成向量，找数字最接近的几段。它的美妙和局限都在同一个地方：它抓的是「语义像不像」，不是「答案在不在」。用户问「退款多久到账」，语义最像的段落可能是「退款政策总则」，而真正写着「三个工作日」的那条细则排在第八名，被截断扔了。

大白话

大白话：向量检索像一个只按「主题」找书的图书管理员——你要找的是某页上的一个数字，它给你抱来五本同主题的书。数字在哪页，得靠更细的切分和排序。

检索质量是上限，切分是第一战场

RAG 系统的质量上限由检索决定：资料没检索到，模型再聪明也只能编。而检索质量的第一个战场出人意料地朴素——**怎么把文档切成小块**。切太大，向量被多个主题稀释，检索不准；切太小，答案的上下文被拦腰砍断，检索到了也读不懂。

实用的原则有两条。一是**按文档自身的结构切**（按章节、条款、表格边界），而不是死板地每五百字一刀。二是**给每块附上它的出处标题**——「3.2 退款细则：……」比裸文本好检索得多，因为问题往往和标题语义相近。

再往上，是**重排序**（rerank）：先用便宜的向量检索粗筛出五十段，再用一个更精细的模型给这五十段按「真的能回答这个问题吗」重排，取前五。粗筛保召回、重排保精度，这是用一层便宜的快换一层贵的准。

记忆：窗口装不下的，用结构换

多轮对话里另一个问题：聊了五十轮，最早的轮次早被挤出窗口了，模型却还要记得用户说过「我对花生过敏」。常见的解法分三层，按成本递增：

- **滚动摘要**：定期让模型把历史对话压缩成一段摘要，塞进窗口。便宜，但摘要是有损压缩，细节会丢。
- **事实抽取**：把「用户花生过敏」这类关键事实抽出来存进结构化字段，每轮必带。精确，但要预先想清楚「什么算关键事实」。
- **检索式记忆**：把历史全部存起来，每轮按当前问题去检索相关旧对话。最灵活，也最贵、最容易检索出噪音。

选哪层，取决于「记错的代价」：过敏信息必须走事实抽取，闲聊偏好用滚动摘要就够了。

上下文装配：架构师真正设计的东西

把这一章收拢：一个 AI 应用的每一次模型调用之前，都有一段看不见的装配代码在跑——查权限、检索资料、拼摘要、带事实、截断历史，最后组装出那一次的上下文。**这段装配逻辑才是系统的核心资产**，提示词只是它的最后一行。

但模型光会「看资料、答问题」还不够，真实的系统要它去**做事**：查订单、发邮件、调接口。下一章，给模型接上手和脚。

第五章 · 给模型接上手和脚

到目前为止，我们的模型只会说，不会做。它能告诉你「这个订单应该退款」，但退款动作本身，还得人点按钮。这一章讲的是跨越这条线的机制——也是「聊天机器人」和「智能体」之间真正的分界。

工具调用：模型写申请单，程序去执行

工具调用（function calling）的机制，说穿了朴素得让人意外：你把系统里可供使用的函数（查订单、算运费、发邮件）按 JSON Schema 的格式登记给模型；模型在生成时如果发现「我需要先查一下订单」，就输出一段结构化数据——函数名加参数——而不是普通文本。**然后轮到你的程序干活**：解析这段数据，真的去执行那个函数，把结果塞回上下文，模型接着往下说。

注意这里一个常被误解的事实：**模型自己什么都不能执行**。它只能「提议」调用。执行权限、参数校验、超时、失败重试，全部在你的代码里。这不是缺陷，恰恰是安全边界所在——把「想做什么」的决定权给模型，把「能不能做」的否决权留在程序里。

大白话

大白话：模型是个只能写申请单的员工，盖章执行的永远是你的程序。审批流设计得好，员工再聪明也闯不了大祸。

智能体循环：一个 while 循环而已

把工具调用串起来，就得到了 智能体（agent）。它的核心骨架简单到可以写在黑板上：

1. 把任务和可用工具交给模型；
2. 模型输出「下一步动作」——要么调用工具，要么给出最终答案；
3. 如果是工具调用，程序执行，把结果追加进上下文；
4. 回到第 1 步，直到模型给出最终答案或撞上限。

所谓「会自己干活的 AI」，就是这个循环加上一堆工程护栏。神奇感消失了，但工程问题浮出来了：这个循环**会放大错误**。单次调用准确率 95% 的模型，连走十步，全对的概率只剩约六成。步数越多，复利式衰减越狠。这就是为什么「演示里惊艳、上线就翻车」是智能体的常态——演示是三步，生产是三十步。

护栏设计：架构师在智能体上的真正工作

既然错误会复利累积，工程重心就不是「让每一步更聪明」，而是「让错误可被拦截」。几条久经考验的做法：

- **限制步数和预算**：硬性上限（比如最多十五步、最多花掉多少 token），撞限就停下来交给 人。没有上限的循环是定时炸弹。
- **工具分级**：只读工具（查询、搜索）放开用；写操作分两级——可逆的（存草稿）自动执行，不可逆的（发钱、发邮件给外部）必须人工确认。
- **验证步骤**：关键动作后加一个「检查结果的合理性」的步骤，由模型或规则校验。比如转账后核对余额变化。
- **全程留痕**：每一步的输入输出落日志。智能体出错时，没有轨迹就等于没有现场。

什么时候不该用智能体

这是本章最重要的判断：**能用固定流程写死的，不要用智能体**。如果任务的步骤是确定的（先查订单、再算金额、再退款），写成普通代码，让模型只在真正需要判断的环节出场。智能体的自由度是成本、延迟、不可靠性的总和，只该花在「步骤事先无法穷举」的任务上。架构师的功力，体现在把 80% 的流程写死、把 20% 的判断留给模型。

到这里，系统会看、会说、会做了。下一个问题随之而来：你怎么知道它做得**好不好**？改了一行提示词，是进步了还是退步了？没有评估，前面所有的工程都是盲修。下一章，评估。

第六章 · 没有评估就没有工程

前面五章建起来的系统，现在面临一个朴素的问题：你改了提示词、换了模型、调了检索，然后呢？打开聊天框试三个例子，觉得「好像变好了」？这就是绝大多数 AI 应用的真实状态——**凭感觉迭代**。传统软件有测试，AI 应用的对应物是 评估 (evals)。一句话概括这一章：没有评估，前面所有章节的东西你都只是在碰运气。

为什么「试试感觉不错」不成立

传统代码改坏了，测试会红。AI 系统改坏了，表面风平浪静——它照样流畅地输出，只是边缘情况的答案悄悄变蠢了。而你手工试的那三个例子，恰好都是最容易的例子。模型的输出是分布，不是函数：同样的输入跑两次都可能不同，「试了三次都对」和「通过率 99%」之间隔着几百个没测过的输入。

更隐蔽的是**回归不可见**：为了让模型学会处理「用户骂人」的情况，你在提示词里加了一条规则，骂人场景修好了——但这条规则同时让正常投诉的回复变得生硬。没有对照评估，这种拆东墙补西墙你永远不会发现。

评估集：不是题库，是资产的快照

评估的第一步是造一个 评估集：一批「输入 + 期望表现」的样例。它从哪里来？最好的来源是**线上真实数据**——用户真实问过的问题、系统真实答砸过的案例。每出一次线上事故，就把那个输入收进评估集。久而久之，评估集成了这个系统最值钱的资产：它是「我们的用户到底在意什么」的快照，比任何通用 benchmark 都懂你的业务。

大白话

大白话：通用 benchmark 考的是「别人家的孩子」，评估集考的是「你家的孩子」。换模型、改提示词时，你关心的是后者。

规模上有个反直觉的事实：**起步时五十个精心挑选的样例，好过五千个随手抓的**。因为评估集的价值在于覆盖面——每个已知的失败模式、每类典型输入至少占一席——而不在于数

量。样例要持续维护：太容易的删掉（已经 100% 通过的样例不提供信息量），新失败模式随时补入。

判分：模型评模型，人评模型评得好不好

判分（「这个回答算好吗」）比造题难。三种办法，按成本递增：

- **程序化判分**：输出有确定答案时用——JSON 能不能解析、关键字段对不对、答案里有没有出现禁用词。便宜、绝对可靠，但只覆盖结构化任务。
- **模型判分**：让另一个（通常更强的）模型按评分细则给回答打分，即「LLM 当裁判」。便宜、可扩展，能评开放式回答；代价是它有自己的偏见——偏爱长的、偏爱措辞自信的、偏爱自己生成的风格。所以评分细则要写得像合同：列出具体的扣分项，而不是「给 1 到 5 分」。
- **人工判分**：最贵最慢，但它是**校准器**——定期抽样人工评一批，用来检验「LLM 裁判的分数和人类判断一致吗」。不一致，就先修裁判，再信它的分数。

工程上的组合通常是：每次改动跑程序化判分 + LLM 裁判（几分钟、几块钱），每周人工抽样校准一次。

评估驱动的迭代节奏

有了评估集和判分器，迭代从碰运气变成科学实验：改一个变量，跑全量评估，看总分和每个失败类别的变化，决定留下还是回滚。这就是第三章说的「提示词可回归」的落地方案。

架构师视角下，评估还有一个更深的作用：**它是全书的枢纽**。后面第八章的成本账（便宜模型掉多少分）、第九章的微调决策（微调到底提升了吗）、第十二章的自训模型（训出来的模型到底变好了没有）——每一个重大决策的度量衡都是评估。不会造评估的人，在后面的每一步都是盲人。

下一章换一个方向：系统内部做对了还不够，它还得在外部世界的混乱里活下来——超时、限流、模型提供商宕机。可靠性工程。

第七章 · 让系统扛住线上

AI 应用的可靠性工程，一半是老的分布式系统知识，一半是 AI 独有的新坑。老的部分（重试、超时、熔断）教科书上都有，这一章只讲 AI 特有的部分——因为恰恰是新坑，最容易在半夜把你叫醒。

AI 依赖的三个新失败模式

传统后端挂掉的方式很干脆：超时、报错、返回 500。AI 依赖的挂法要暧昧得多：

第一，概率性正确。同样的输入，这一分钟答对，下一分钟答错。你不能用「跑通一次」来验证修复，只能靠第六章的评估跑通过率。

第二，静默变质。模型提供商会更新模型——同一个 API 名字，底下的权重换了。你什么都没改，系统行为变了。没有版本钉死和定期评估，你连「什么时候开始变差的」都查不出来。

第三，输出违反约定。你的下游代码期待一个 JSON，模型却输出了「我很抱歉，我无法……」。格式错误在 AI 系统里是常态，不是异常。

大白话

大白话：传统接口坏了会喊疼，AI 接口坏了会微笑。可靠性工程的第一原则就是别信它的微笑。

防线一：验证与修复，而非祈祷

既然输出违反约定是常态，就不要假设输出合法。每次模型返回后，先过一道**程序化验证**：JSON 能解析吗？必填字段在吗？数值在合理范围吗？验证不过，有两条路：**重试**（同一个请求再来一次，概率性错误重试往往就好）或者**修复式追问**（把错误信息喂回去：「你的输出缺了 amount 字段，请修正后重新输出」——模型很擅长这类自我修正）。第二章说的约束解码能从源头消掉一大半格式错误，这是机制优于祈祷的又一个例子。

防线二：降级链，而不是单点

模型提供商会限流、会抖动、会整个宕机。架构上必须预设一条**降级链**：首选模型超时或报错，切到备选提供商的同档模型；备选也挂了，降到更小更便宜的模型（质量下降但服务还在）；全挂，落到**确定性兜底**——一段写死的文案加人工接管入口。

降级链的关键不在「有备选」，而在**事前演练**：备选通道如果你从来没跑过流量，它大概率在真出事的那天也是坏的（密钥过期、配额为零）。成熟的团队会常态化让 1% 的流量走备选通道，既保持它「是热的」，也持续拿到两个模型的对照数据。

防线三：监控要盯分布，不只盯报错

传统监控盯错误率和延迟。AI 系统还得盯**分布漂移**：回答的平均长度是不是突然变了？拒答率是不是上升了？某个工具被调用的频率是不是异常？这些指标不变红，但它们的变化往往先于用户投诉。再配合抽样质检——每天随机抽一批真实对话，走一遍评估流程（第六章的裁判在这里复用）——你就有了系统「健康度」的连续曲线，而不是等用户来告诉你。

防线四：成本护栏

可靠性还包括「别把公司烧穿」。一个循环失控的智能体（第五章）、一个被恶意刷的公开入口，一晚上可以烧掉几个月的预算。所以按用户、按功能设 token 配额与消费告警，是可靠性工程的一部分，不是财务的事后算账。

到这里，系统能在混乱里活下来了。下一章把账算细：活下来之外，怎么活得便宜——成本与延迟的经济学。

第八章 · 算清楚钱和毫秒

很多 AI 项目不是死于效果，是死于账单。演示阶段没人算成本——一天几百次调用，几美元的事。上线后流量涨三个数量级，财务拿着账单来问的时候，才有人回头算这笔账。架构师的基本功，是在写第一行代码之前就把这笔账算出来。

账单的结构：输入便宜是假象

模型 API 按 token 计费，**输入和输出的单价不同，输出通常贵好几倍**。原因是生成是自回归的——每产生一个 token 都要把整个模型跑一遍，而读输入可以并行。这个定价结构直接塑造架构：

- **让模型少说话**。「只输出结论，不要解释」这种约束，省钱效果立竿见影。一个爱写三段解释的回答，成本可能是「输出一个 JSON」的十倍。
- **输入端的大头是重复**。每次请求都带着同样的系统提示、同样的检索资料，这些重复输入是缓存的主战场。

把账算细的公式很简单：单次成本 = （输入 token 数 × 输入价 + 输出 token 数 × 输出价）。但真正的功夫在于搞清楚「我的平均一次请求到底是多少 token」——去日志里量，别拍脑袋。量的结果经常让人吓了一跳：那个你以为「很轻」的功能，因为塞了整份检索资料，单次成本是预期的八倍。

缓存：同一笔钱不付第二次

主流提供商都支持**前缀缓存**：如果本次请求的开头部分和上一次完全相同，这部分输入按折扣价计费（甚至接近免费）。工程含义：**把不变的内容放在提示词最前面**，把变动的（用户问题、当次检索结果）放在最后。顺序反了，缓存全部失效——这是「排列顺序值多少钱」的直接体现。

再往上是应用层缓存：同一个用户问题（或其嵌入相近的问题）之前答过且答案仍有效，直接返回旧答案，根本不调用模型。FAQ 类场景里，这一层常常能挡掉三到五成的调用。

模型路由：贵的脑子只用在难的题上

旗舰模型和轻量模型的单价能差一两个数量级，但绝大多数请求根本不需要旗舰。**模型路由**用一个便宜的前置判断（可以是小模型，也可以是规则）把请求分流：简单的走小模型，只有复杂的上大模型。这和一个团队的分工一模一样——你不会让资深专家去回复「你们几点上班」。

路由的底气来自第六章的评估：你得先知道「小模型在这类请求上的通过率是 97%」，才敢把流量切过去。没有评估的路由是赌博，有评估的路由是算术。

延迟：用户感受到的那一面

成本账的另一面是时间账。生成的延迟大致正比于输出 token 数——又是「让模型少说话」。两个工程手段值得知道：**流式输出**（边生成边推给前端，用户感知延迟从「全部生成完」变成「第一个字出现」）和**批处理接口**（非实时任务如夜间跑报告，用离线批量 API，价格通常打对折）。实时交互用流式换取体验，离线任务用批处理换取折扣，各取所需。

当省钱省到尽头

把这一章的手段全用上——缓存、路由、少说话、批处理——成本能砍掉大半。但有些场景会发现：剩下的成本来自「每次都要用大模型处理本可以固化下来的模式」。当同一个模式被调用几百万次，一个新的选项开始进入视野：**把大模型的本事蒸馏进一个自己训的小模型**，推理成本降一个数量级。这是第九章到第十二章的主题。但在掏钱训练之前，先得回答一个更根本的问题：你怎么知道提示词这条路真的走到头了？

第九章 · Prompt 到顶的信号

这一章是全书的分水岭。前八章的所有手段——提示词、上下文、工具、评估——都不改变模型本身，只是在模型外面做文章。从这一章起我们要动模型本身了。所以必须先把门槛讲清楚：**到底什么信号说明「外面做文章」这条路走到头了？**动模型很贵，动错了更贵。

一个核心区分：缺知识，还是缺行为

模型答得不好，原因只有两大类，而它们的解法完全不同：

缺知识：模型不知道你公司的退款政策，因为训练数据里没有。解法是把知识在推理时喂给它——RAG、上下文工程，第四章已经讲过。知识会过时、会更新，放在上下文里随时换，最灵活。

缺行为：知识模型都有，但它**做的样子**不对。输出格式总不符合你们系统的古怪约定；语气总是不像你们的品牌；某类边界情况反复判断错，提示词里加了规则、给了例子，好两天又犯。这是「行为模式」问题——你要改的不是它知道什么，而是它的**习惯**。习惯写在模型的权重里，改习惯就得动权重，也就是微调。

大白话

大白话：不知道的事，查资料（RAG）；改不掉的毛病，改造本人（微调）。把知识问题误当行为问题去微调，是最常见也最贵的错误——你会把一个明天就过期的政策，焊死进模型的权重里。

五个到顶的信号

具体什么信号说明该认真考虑微调了？五个，按可信度排序：

1. **提示词越来越长、越来越脆**。几百行的提示词，全是补丁，改一处崩三处——说明你在用自然语言描述一个本该用权重承载的行为模式。
2. **少样本示例塞不下**。要把边界情况讲全需要五十个例子，上下文装不下、装下也付不起钱——说明这个行为的复杂度超过了提示词的带宽。

3. **评估集上某类错误率压不下去**。注意前提：有评估集。没有评估数据支撑的「感觉提示词到顶了」，一律是错觉。
4. **成本或延迟被逼到墙角**。第八章的尽头：每次调用都靠大模型加超长提示词维持行为，量一大就烧钱。微调后的小模型可以把行为「内置」，提示词缩回几十行。
5. **一致性要求极高**。比如合规场景，输出格式必须 99.99% 稳定。提示词给的是统计意义上的稳定，微调能把行为钉得更死。

反向的刹车：什么时候千万别微调

同样重要的是刹车清单。出现以下情况，微调基本是错的投资：

- **问题其实是知识缺失**——回去做 RAG。
- **评估集都没有**——你既无法判断微调有没有用，也无法防止它把别的能力训坏。先补第六章的课。
- **数据凑不齐几百条高质量样例**——微调的效果上限由数据质量决定，凑不出好数据，训出来的是退步。这是下一章的主题。
- **需求还在快速变**——行为每周都在改，用提示词迭代是分钟级，微调迭代是天级。敏捷期用提示词，稳定后再固化。

架构师的判断框架

把这一章浓缩成一句决策口诀：**知识进上下文，行为进权重；先用提示词探路，用评估确认到顶，再用微调固化**。这个顺序不能反——微调是「把探明的路铺成柏油」，在探明之前铺路，只是把弯路修得更贵。

确认该微调之后，真正的硬仗不是训练本身，而是训练之前的那一步：数据。业内的说法是，微调项目九成的工作量在数据上。下一章看看这九成到底在干什么。

第十章 · 数据是训练的九成

第九章的结论是：确认该微调之后，九成的工作量在数据上。这一章看这九成是什么。先记住一句业内被反复验证的话：**微调项目的质量上限由数据决定，训练只是把它兑现**。训练代码可以照抄开源脚本，超参数有默认值兜底，唯独数据——只有你知道你的业务里「好」长什么样。

数据长什么样：输入、输出、以及「为什么这样算好」

微调数据的单位是**样例**：一条「输入 → 理想输出」。对客服系统，就是「工单原文 → 一份完美的回复草稿」。注意「理想」两个字——训练数据的本质定义是：**你希望模型在类似情况下做出的行为**。模型不会分辨你喂的是示范还是凑合，它一律照单全收，内化成习惯。喂它一百条「还行的回复」，你就得到一个稳定输出「还行」的模型。

大白话

大白话：微调像教徒弟——你示范什么水平，徒弟就学到什么水平。徒弟不会自动超越师傅。

数量：比你想的少；质量：比你想的狠

新人最容易犯的错是把微调当成「越多越好」的数据工程。实际上，行为塑造类的微调，**几百到几千条精修的样例往往就够**——因为大模型预训练时已经学完了通用能力，微调只是调行为习惯，不是从零教语言。真正决定成败的是每条样例的质量：输出是否真的无可挑剔？格式是否逐字一致？是否覆盖了线上真实出现的输入分布？

一个经验法则：**宁要三百条每条的输出你都愿意直接发给客户的样例，不要三万条没人逐条看过的**。后者里的 5% 脏数据（格式错乱、答案错误、语气不对）足以让模型学会一堆坏习惯，而且你事后很难查出来是哪个坏习惯来自哪批数据。

数据从哪来：三条路，风险递增

- **线上真实数据改写**：把真实工单配人工修过的理想回复。分布最真实，是黄金来源；成本是人工改写的时间。
- **大模型合成 + 人工筛**：让强模型批量生成候选，人来淘汰和修正。速度快十倍，但要警惕两件事：合成数据会放大强模型的腔调（满是「首先、其次、总而言之」），以及**把评审当成了走过场**——人一旦开始批量点「通过」，合成数据就退化成垃圾。
- **纯自动合成**：没人筛。只配用于无关紧要的原型验证。

切分与泄漏：考试题不能出现在练习册里

训练之前，要把样例切成训练集和**留出集**——留出的部分绝不参与训练，只用于训完后检验「模型在没见过的输入上表现如何」。这是机器学习的常识，但有一个 AI 时代特有的坑：**近重复泄漏**。一条工单稍微改写几个字同时进了训练集和留出集，留出集的成绩就虚高——模型不是会了，是背过。

和第六章接上：评估集与训练集是两套账本

这里要和第六章对齐：线上评估集是「业务在乎什么」的快照，训练数据是「希望模型学会的行为」。两者的关系是——评估集里的失败案例指明了需要什么样的训练数据，但评估集本身**绝不能混进训练集**，否则你永远失去了度量衡。数据的纪律，到这里已经和算法的纪律一样重了。

数据备齐，终于可以谈训练了。下一章：SFT、LoRA、偏好对齐——这些名词背后其实只有一个动作。

第十一章·微调实操

数据备齐（第十章），该训练了。这一章把训练环节的三个高频名词——SFT、LoRA、偏好对齐——讲到「你能和别人争论方案」的程度。先说一个能消除大半神秘感的事实：**这三种方法在做的都是同一个动作**——拿一批数据，调整模型的权重，让「理想输出」在这种输入下出现的概率变高。区别只在于调多少权重、用什么标准算「好」。

SFT：照着示范改习惯

SFT（监督微调，supervised fine-tuning）是最基础的一步：把你的「输入 → 理想输出」样例喂给模型，每遇到模型自己的输出和理想输出不一致，就微调权重，把理想输出的概率往上推一点。几万次这样的推拉之后，模型的行为习惯就被掰成了你示范的样子。

理解 SFT 的关键是理解它的边界：**它只能教「怎么做」，教不了「新知识」**。把公司政策原文当训练数据灌进去，模型不会因此记住政策——它学到的是「这种文字的腔调」，具体事实照样记错、照样幻觉。这就是为什么第九章说知识进上下文、行为进权重。用 SFT 灌知识是最经典的误用。

LoRA：不动本体，只加一层「小抄」

直接调整模型的全部权重（全量微调）有两个现实问题：贵（几十亿参数全部参与训练，显存开销巨大），险（容易把通用能力训坏，即「灾难性遗忘」）。LoRA（低秩适配）是个精巧的绕行：**冻结原模型的全部权重，只在关键位置插入一小组新增的小矩阵，训练时只调这些小矩阵**。

为什么有效？因为「把一个通用模型调成某个垂直行为」所需的改变，在数学上是低维的——不需要动几十亿个参数，几百万个就够表达这个「差量」。工程上的好处非常实在：显存需求降一个数量级（消费级显卡也能训）；训练快；而且**原模型权重一个字节没动**——训坏了，把小矩阵扔掉就是，还能给不同业务各挂一个 LoRA「插件」切换使用。

大白话：全量微调是给房子重新装修，LoRA 是只换窗帘和摆设。绝大多数「换风格」的需求，换窗帘就够了，而且不满意随时换回来。

偏好对齐：没有标准答案时，用「哪个更好」来教

SFT 需要每条样例有唯一的理想输出。但很多场景不存在唯一标准答案——「回复的语气要温暖但不谄媚」这种东西没法写成一条示范。这时候用**偏好对齐**：给模型看同一个输入的两个回答，只标注「A 比 B 好」。DPO（直接偏好优化）这类方法直接用这种成对比较来调整权重，让「像 A 那样」的输出概率升高、「像 B 那样」的降低。

偏好数据有个绝妙的低成本来源：**你的线上系统每天都在产生**。人工客服把 AI 草稿改成终稿——草稿 vs 终稿，就是一对现成的「较差 vs 较好」。把这条数据管道接上，系统的每一次人工修正都在为下一次训练攒料。

第一次微调的实战顺序

1. 先拿五十条精选数据在现成平台（或开源脚本加一张显卡）上跑一次 LoRA，目标不是效果，是**跑通全流程**：数据格式、训练、导出、部署、评估。
2. 用留出集（第十章）和线上评估集（第六章）各评一遍——特别注意**通用能力有没有退化**，这是微调最常见的暗伤。
3. 有效，再把数据量加上去；没效，先怀疑数据质量，最后才怀疑超参数。十次里有九次，问题在数据。

微调走到这一步，你已经有算法岗的手感了。下一章上最后一个台阶：什么时候不微调别人的模型，而是自己从头（或从基座）训一个专用的？

第十二章 · 自训一个专用模型

第十一章的微调动的是别人的开源模型。这一章谈最后一步：**从基座开始，训练一个真正属于你的专用模型**。先泼一盆冷水：绝大多数团队永远不需要做这件事。所以本章的前半部分是「什么时候值得」，后半部分才是「怎么做」。顺序不能反——这个决定做错，烧的是六位数起步的真钱。

什么时候值得自训：三个硬条件，缺一不可

1. **任务定义稳定且高频**。需求还在变，训出来就过时；调用量不够大，摊不回训练成本。自训的经济学本质是：把每次调用都在付的「大模型理解你意图」的税，一次性前置成训练成本。量不够大，税前置就不划算。
2. **通用模型在该任务上有结构性的天花板**。注意是「结构性」——不是「GPT 偶尔答错」，而是「你的任务依赖的信号，通用模型的训练数据里根本没有」。典型例子：工业设备的振动信号分类、你们内部十年积累的代码审查记录、某种小众语言的法律文书。这类数据世界上只有你有，这是自训唯一不可替代的理由。
3. **数据主权或延迟的硬约束**。数据不能出内网，或者延迟预算小到不可能走云端 API——这时候自训自部署不是优化，是准入条件。

大白话

大白话：值得自训的情形，说到底只有一种——你手里有别人没有的数据，而且这个数据对应的需求又大又稳。缺了「独有数据」这一条，前面九章的手段几乎总是更便宜。

基座选择：别从随机权重开始

「自训」在今天几乎不意味着从零训练。从零预训练一个有竞争力的模型要烧掉几百万到上亿美元的算力，那是巨头游戏。现实路径是**站在开源基座的肩膀上**：选一个尺寸合适的开源模型（参数量按任务复杂度选，能用 7B 就别上 70B），用你的数据做继续预训练或深度微调。基座已经学会了语言和推理的通用能力，你只注入它缺的领域信号。

训练流水线：和第十一章同构，只是放大

好消息：概念上没有任何新东西——数据、训练、留出集评估，就是第十、十一章那套，只是每个环节放大了一个量级。真正新增的是**工程纪律**：

- **小规模先行**：先用 1% 的数据和最小的模型跑通全流水线，验证数据管线、评估指标都对，再上量。直接全量开训，等于没测试就上线。
- **checkpoint 纪律**：训练中定期存档，并且**每个存档都过评估**——训练后期模型常常开始过拟合，最好的 checkpoint 往往不是最后一个。
- **失败的预算**：训练任务崩溃、loss 发散是常态不是意外，预算里要留重跑的钱。

算力账：一次性支出 vs 永续租金

自训的账要和 API 的账对着算。API 是永续租金：每次调用都在付钱，量越大越贵。自训是一次性大额支出加少量推理成本：训练花一笔（租 GPU 按周计），之后每次推理只花自托管的电费级成本，量大时单价可以比 API 低一两个数量级。交叉点在哪里，取决于你的调用量、模型尺寸和团队维护成本——把三个数代进去算，别凭感觉。

别忘了账的隐藏项：**维护**。基座模型在进步，开源社区每半年刷新一次，你的专用模型如果不持续迭代，两年后就是古董。自训买到的不只是模型，是一份长期的维护责任。

走完这一步意味着什么

能独立完成「判断该自训 → 选基座 → 备数据 → 跑流水线 → 评估 → 部署」全链路的人，就是市场上说的算法岗里能落地的那一类。你已经站在这条路的技术顶端了。剩下两章换个视角：技术之外，架构师每天在做什么决定，以及这条路具体怎么走。

第十三章 · 架构师的系统设计

技术章节到此结束。这一章回答一个更日常的问题：**AI 应用架构师的工作日到底长什么样？**答案可能会让期待「画架构图」的人意外：架构师的大部分时间，花在三种决定上——结构、边界、时机。

结构的决定：编排，而不是堆叠

真实的 AI 系统很少是「一个模型回答一切」，而是多个不同档位、不同职责的模型加传统代码的编排。架构师设计的结构，常见形态是**分层**：便宜的小模型做分流、意图识别、格式转换这些「体力活」；旗舰模型只处理真正需要推理的环节；确定性代码（规则、校验、数据库）把守所有不能错的关口。

分层的原则在前面各章已经分别出现过，这里把它们收拢成一条：**确定性优先于模型，小模型优先于大模型，检索优先于记忆，提示词优先于微调**。每一层升级都意味着成本和复杂度的跃迁，架构师的工作就是挡住不必要的跃迁。一个「只用了一个模型加一个提示词」的设计，往往比「五个智能体协同」的设计更能体现功力——简单是省出来的，不是偷懒出来的。

边界的决定：AI 管什么，人管什么，代码管什么

每个 AI 系统里都有一条三权分立的边界要划：

- **代码管确定性的事**：金额计算、权限校验、流程流转。能用 if 写清楚的，绝不让模型投票。
- **模型管判断与生成**：理解意图、起草内容、在非结构化信息里找答案。
- **人管不可逆的事**：第五章的工具分级在这里放大成组织设计——哪些动作永远要人点头，出错时谁接手，责任怎么界定。

这条边界划错是架构师最严重的失误：让模型管了不该管的（自动审批大额退款），是事故；让代码硬写本该交给模型的（拿正则解析用户意图），是返工。划对了，系统里每个部件都在自己最可靠的位置上。

时机的决定：什么时候做，什么时候忍

架构决策里最被低估的是时机。同样的方案，时机不同，对错相反：

- **什么时候把临时方案固化**：提示词里的补丁攒到什么程度该微调（第九章五个信号）；脚本跑到什么量级该变成正式服务。
- **什么时候容忍不完美**：评估通过率 92% 够不够上线？答案从来不是技术问题——它取决于答错那 8% 的代价落在谁身上。架构师要能把这个代价翻译成业务语言，和产品、运营一起拍板，而不是躲在「技术上还不完美」后面无限延期。
- **什么时候拥抱新东西**：模型半年迭代一代。架构师要有纪律地跟踪（自己的评估集就是最好的新模型试金石——新模型发布，跑一遍评估，分数说话），也要有纪律地**不追**：生产系统的稳定收益，常常大于新版模型 3% 的提升。

一个隐藏的职责：让团队不依赖你

最后是架构师最不像技术的技术职责：**把判断力沉淀成系统，而不是沉淀成「问你」**。评估集、降级链、成本看板、上线检查清单——这些东西的共同作用是让团队在你不在场时也能做出八十分以上的决定。一个凡事都要亲自把关的架构师不是资产，是瓶颈。

到这里，书的技术主线和岗位画像都完整了。最后一章回到你身上：这条路具体怎么走，每一步怎么验证自己走到了。

第十四章 · 走这条路

最后一章，把整本书折成一张可以自己打勾的路标。不讲新东西，只回答三个问题：每一级台阶的「毕业标准」是什么？怎么练最快？以及——转算法岗这件事，有哪些没人明说的真相。

台阶一：应用工程师的毕业标准

你不再以「调通了」为完成标准。看到一个 API 调用，你条件反射地问：温度多少、为什么；输出有没有机制级的格式保证；这次调用多少 token、花多少钱；提示词改一个字，拿什么验证没改坏。**毕业标志：你负责的功能，线上出问题时你能说出「它在哪一环、为什么」，而不是「我重新试试」。**（对应第二、三章）

台阶二：应用架构师的毕业标准

视野从一次调用扩到一个系统。毕业标志是四个「有」：

- **有评估**：你的系统有一个持续维护的评估集，任何改动跑它说话；
- **有降级**：任何一个外部依赖挂掉，系统都有预演过的退路；
- **有账**：你说得出每个功能的单次成本和延迟构成，以及量翻十倍后谁先崩；
- **有边界**：系统里哪些归代码、哪些归模型、哪些归人，是你白纸黑字划出来的，不是事故划出来的。

（对应第四到八章、十三章）这四个「有」，比任何职位 title 都更能证明你是架构师——因为它们每一个都对应一次「没有它时烧过的钱或出过的锅」。

台阶三：算法岗的毕业标准

毕业标志：**独立完成过一次「数据 → 训练 → 评估 → 部署」的闭环，并且能讲清每一步的失败模式。**数据泄漏怎么防、训坏了先看数据还是先看超参、哪个 checkpoint 上线为什么、自训和 API 的账怎么对——这些问题你答得出第一手的答案，而不是背下来的答案。（对应第九到十二章）

怎么练最快：一个反常识的建议

最快的练法不是上课，是**把一个真实的小系统推到足够大的流量，让它自然地把你逼过每一级台阶**。流量是最好的老师：量一小，所有问题都能靠人肉糊住；量一大，成本逼你学第八章，事故逼你学第七章，评估需求逼你学第六章，提示词到顶逼你学第九章。选一个你真心在乎的小场景——哪怕是给自己做的工具——然后想办法让它被真用。

大白话

大白话：不栽跟头的学习是消费，栽了跟头并且复盘的学习才是投资。流量替你制造跟头。

转算法岗的三个真相

最后说几句关于「转岗算法（训模型）」的实话，招聘启事上不会写：

1. **应用背景是优势，不是前科**。算法岗最缺的不是会推公式的人，是知道「这个模型上线后会被怎么用」的人。你从应用走来，对数据分布、评估、线上失败模式的直觉，是纯算法背景的人要花几年补的。
2. **别从零补数学竞赛路线，从微调项目切入**。先用第十一、十二章的路径做出两三个真实微调项目，在项目中按需补数学（梯度、损失、过拟合的直觉），比先啃一年教材再碰代码有效得多。数学是按需取用的工具，不是入场券。
3. **你的作品集就是评估集和数据集**。面试算法岗时，一个「我从线上事故攒出的五百条评估集加三轮微调迭代记录」的仓库，胜过十张课程证书。它证明的不是你学过，是你训出来的东西真的变好了——这是这本书反复说的度量衡，也是这个岗位唯一认的货币。

回到最初的问题

第一章问：只会调 API 的人，怎么长出架构师的判断力？现在答案完整了：判断力的范围一级一级扩大——从一次调用，到一个系统，到模型本身；每一级的通行证都是同一个东西：**能度量、能解释、能算账**。工具会一直换，这三样不会。

路已经画完。剩下的那段，得你自己走了。