AI 越强，底层越重要

从哈萨比斯的建议说起，拆开模型、软件与第一性原理

竹小竹8167

费曼式写法 · 由多个 AI 代理撰写与互相审校

导读

从哈萨比斯的建议说起，拆开模型、软件与第一性原理

这本书从一句看似普通的学习建议开始：AI 时代，年轻人仍然要学好数学、物理、计算机。它普通到像长辈劝学，细想却很反直觉。既然 AI 已经能写代码、解题、读论文、做计划，为什么基础知识不是变便宜，而是变更重要？

答案不在“人要和 AI 比聪明”。多数人已经不该把战场放在那里。真正的问题是：AI 的判断要落到真实世界，必须经过软件、接口、工具、权限、反馈和验证。谁看得懂这些底层结构，谁就更知道该让 AI 做什么、怎么做、做到哪里必须停下来检查。

所以这不是一本劝你回去背公式的书。它想讲清一件事：**基础知识不是旧时代的包袱，而是强 AI 时代的操作系统。**数学让你看见结构，物理让你尊重约束，计算机让你理解执行。三者合起来，训练的是穿透表象的能力。

读完这本书，你不一定会立刻多会一个工具。但你应该能更准确地判断：一个限制是真的，还是界面造成的错觉；一个创新是可靠迁移，还是把因果链剪断后的幻觉；一个模型的回答是可以执行的计划，还是只是顺滑的话。

第一章 · 那句看似保守的建议

短视频里那句话并不新鲜：年轻人要学好基础知识，尤其是数学、物理、计算机。新鲜的是，说这句话的人站在 AI 最前沿。他不是在怀旧，也不是在替传统课程打广告。他的潜台词更硬：AI 越强，越需要人懂底层。

这里说的 底层，不是神秘的“高深知识”，而是一个东西真正靠什么运转。比如手机屏幕上有一个按钮，表面看是图标；往下一层是应用代码；再往下一层是操作系统事件；再往下是芯片里的电路。你不必每层都成为专家，但你要知道它们不是魔法，而是一层层约束叠起来的结果。

大白话

大白话说：底层就是“这件事到底靠什么成的”。知道这一点，你才知道哪里能改，哪里不能乱改。

很多人听到“AI 会越来越聪明”，第一反应是：那人还学基础干什么？反正模型会算、会写、会查、会画。这个反应只看见了 AI 的输出，没有看见输出背后的链条。模型给你一段代码，不等于系统就能上线；模型说一个方案可行，不等于接口、权限、成本、延迟都自动消失；模型写出一个漂亮解释，也不等于因果关系真的成立。

所以本书要回答的不是“AI 会不会替代人”这种大问题，而是一个更能落地的问题：**当 AI 已经很会思考时，人为什么还要训练基础判断力？**

表面上是学习建议，里面是能力结构

数学、物理、计算机这三类基础课，表面看像三摞知识点。数学有公式，物理有定律，计算机有语言和系统。可真正有价值的不是背下多少术语，而是它们训练了三种看世界的方式。

数学训练你把问题压缩成结构。一个复杂现象里，哪些量可以忽略，哪些量必须留下，哪些关系能写成函数或约束。物理训练你尊重现实。再漂亮的想法也要问：能量从哪来，摩擦在哪里，延迟和损耗怎么出现。计算机训练你理解可执行性。一个判断要变成动作，必须经过数据、状态、接口、错误处理和反馈。

这三种训练合在一起，就是 AI 时代最稀缺的能力：不被表象牵着走。

AI 的表象很有迷惑性。它说话像人，写作像人，甚至能在很多考试里超过人。于是我们容易把“会回答”误认为“已经把事情做成”。但真实世界里，回答只是第一步。后面还有执行、验证、迭代和承担后果。

越强的工具，越放大基础差距

一个普通锤子，给谁用差别有限。一个复杂机床，差别就大了。懂材料、懂图纸、懂误差的人，能加工出精密零件；不懂的人，只会更快地弄坏东西。AI 更像机床，不像锤子。

它能把你的意图放大，也能把你的误判放大。你若知道问题的下一层结构，就能让它查证、拆解、生成、测试；你若只知道一句口号，它也会很流畅地围绕口号编出一堆看似合理的话。

这就是为什么哈萨比斯的建议听起来保守，实际上很激进。它不是说“回到过去那套学习方法”，而是说：未来工具越强，人的工作越从亲手搬砖，转向判断砖该往哪放、墙能不能承重、图纸有没有错。越到这一步，底层知识越像地基。

本书后面会沿着三层往下拆：第一层，AI 与软件是相辅相成的；第二层，懂下一层才有组合和创新；第三层，第一性原理帮我们分清自由变量和客观约束。拆完之后，你会发现“学好基础”不是一句鸡汤，而是一套面对强 AI 的生存方法。

第二章 · 模型会想，软件会做

把 AI 想成一个很聪明的大脑，很容易。但这个比喻只说对了一半。大脑要改变世界，必须有手、眼睛、工具和工作台。模型再会判断，也不能凭空把文件存到你的硬盘，不能凭空给同事发日程，不能凭空让服务器扩容。它必须通过软件把想法变成动作。

模型，这里指能根据输入生成判断和输出的 AI 系统。软件，这里指那些能稳定执行任务的程序、接口、数据库、权限系统和工作流。前者擅长“想下一步做什么”，后者擅长“真的把这一步做完”。

大白话

大白话说：模型像会出主意的人，软件像能照规则干活的机器。主意再好，没有机器也落不了地；机器再稳，没有主意也只是空转。

过去的软件，大多数时候等人来点按钮。人决定要做什么，软件负责执行。AI 进来以后，顺序变了：模型可以先看上下文，判断用户真正想要什么，再选择哪个工具、传什么参数、怎么检查结果。于是软件从“人手的延伸”，变成了“模型手脚的一部分”。

为什么模型越强，软件越重要

直觉上，模型越强，软件好像越不重要。因为它似乎什么都会。但真实情况相反：模型越强，越会遇到更多真实任务；任务越真实，越离不开可靠软件。

写一段总结，只需要语言能力。订一张机票，就需要搜索航班、比较价格、校验身份、支付、出票、改签规则和通知。写一段代码，只需要生成文本。把代码合进生产系统，就需要测试、构建、权限、回滚、监控和审计。越往真实世界走，越多事情不是“想明白”就结束，而是要被系统稳稳接住。

这也是为什么办公软件、编程工具、数据平台、自动化系统会重新变得关键。它们不再只是给人用的界面，而是给模型调用的能力库。一个模型如果能接入高质量工具，它的能力会突然变大；如果只能困在聊天框里，它再聪明也像一个被绑住手的人。

软件质量决定 AI 的上限

模型调用软件时，最怕两件事：不确定和不可恢复。不确定是指接口说不清楚，输入输出含糊，错误信息像谜语。不可恢复是指一旦做错，没有日志、没有回滚、没有权限边界。人用软件时可以靠经验补洞，模型调用软件时，这些洞会被放大。

所以 AI 时代的软件不是随便包一层 API 就够了。它要更清楚地表达自己能做什么、不能做什么；要把状态暴露出来；要把错误讲人话；要允许模型先试探、再执行、再验证。这样的软件，对人也更好用，但对 AI 尤其关键。

这解释了一个表面奇怪的趋势：模型公司会越来越关心工具，软件公司会越来越关心模型。不是谁吃掉谁，而是两边正在合成一个新东西：会判断、会执行、能反馈的工作系统。

如果你只看模型，就会以为未来全靠提示词。如果你只看软件，就会以为 AI 只是一个新按钮。真正的变化在两者之间：模型把软件组织起来，软件把模型的判断落到现实里。基础知识的重要性，也正藏在这个交界处。你懂软件的底层结构，才知道模型的判断能落在哪里，不能落在哪里。

第三章 · 工具不是外挂，是 AI 的手脚

一个只会聊天的 AI，像坐在玻璃房里的人。它看起来很聪明，能分析、能计划、能提醒你哪里有风险，但它碰不到外面的东西。你让它“把这份材料发给团队”，它最多告诉你应该怎么发。真正让它走出玻璃房的，是工具。

工具调用，就是模型不只生成文字，还能选择一个外部工具并传入参数，让工具替它执行动作。查数据库、读文件、跑测试、发邮件、开工单，都可以是工具调用。

> 大白话
>
> 大白话说：工具调用就是 AI 不再只“说”，而是会“叫软件去做”。

这件事看似只是产品功能，实则改变了模型能力的定义。以前我们问一个模型强不强，主要看它会不会回答问题。以后还要看它能不能把问题拆成动作，能不能选对工具，能不能看懂工具返回的结果，能不能发现执行失败。

工具让智能变成杠杆

假设你有一个非常聪明的实习生，但不给他电脑、账号、资料库和权限。他能做的事很有限。给他一套清楚的工具，他就能查资料、整理表格、写代码、提交报告。人的智力没有变，产出却变大了，因为工具提供了杠杆。

AI 也是这样。一个模型接入日历，就能帮你安排会议；接入代码仓库，就能理解项目；接入测试系统，就能验证修改；接入支付系统，就可能完成交易。每接上一类可靠工具，它能完成的任务边界就往外推一圈。

但杠杆有反面。工具如果设计差，模型会被误导。接口名字含糊，模型可能选错；权限边界模糊，模型可能做过头；返回结果缺少结构，模型可能把失败当成功。工具越多，系统越像一张管网，任何一个阀门标错都可能造成事故。

好工具要让模型容易做对

给人用的软件常常依赖“看一眼就懂”的界面。按钮在这里，红色代表警告，灰色表示不可点。模型没有真正的人眼经验，它更需要明确的结构：这个工具的用途是什么，需要哪些参数，执行前会影响什么，失败时返回什么。

好工具至少要回答四个问题。第一，我能做什么？第二，我需要什么输入？第三，我做完会改变什么状态？第四，怎么证明我做成了？这四个问题越清楚，模型越容易把任务稳定完成。

这就是底层知识的价值。懂软件的人不会只问“AI 能不能帮我处理文档”，他会追问：文档在哪里，格式是什么，权限怎么给，版本如何保存，失败如何重试，执行后怎么验收。这些问题听起来琐碎，却是智能落地的轨道。

如果没有轨道，AI 的能力只能停在建议层。如果轨道足够清楚，它就能进入执行层。所谓“AI 原生软件”，本质不是把聊天框塞进页面，而是把软件能力拆成模型能够理解、调用、验证的一组工具。到这一步，基础工程能力就不再是后台细节，而是智能系统的骨架。

第四章 · 下一层到底是什么

“懂下一层”听起来像一句工程师黑话，其实很朴素。你每天用一个东西，如果只知道它表面怎么点，就只能按说明书走。你一旦知道它下一层靠什么运转，就会发现很多限制不是客观规律，只是默认设计。

下一层，指你当前操作背后的那一层机制。你在 IDE 里看到项目树，下一层可能是文件系统；你在页面上点提交，下一层可能是 HTTP 请求；你在聊天框里让 AI 改代码，下一层可能只是它读取了一堆文本再生成补丁。

大白话

大白话说：下一层就是“这个按钮背后到底发生了什么”。看清它，你就不会被按钮的样子骗住。

源代码首先是文本

很多人写代码时，脑子里有一个很强的“项目”概念。打开一个 IDE 工作区，就像进了一间房。房间里有哪些文件，工具就处理哪些文件；房间外的东西，好像天然不相关。

但从更下一层看，源代码首先是文本文件。AI 编程工具读取代码，本质上也是读文本、建上下文、再输出文本修改。IDE 的工作区是一个方便人的边界，不一定是问题真正的边界。

一旦想通这一点，很多限制就松了。比如两个微服务互相依赖，过去你可能分别打开两个项目，改完 A 再改 B，中间靠人脑记住接口变化。现在如果工具允许，你可以把相关项目放在同一个工作目录里，让 AI 同时看到调用方和被调用方。它不是在“跨项目施法”，只是读到了更多相关文本。

这个例子重要的不是技巧本身，而是思维方式：你把“IDE 工作区”从客观边界降级成了人机界面；把“文本依赖关系”提上来当真正边界。边界一换，解法就变了。

工具首先是接口

另一个常见误解，是把工具等同于它的界面。一个编程助手长在 IDE 里，我们就以为它只能被人坐在电脑前使用。可如果它有 CLI，事情就不一样了。

CLI，就是命令行接口。它让你不用点图形界面，而是用一行命令调用软件能力。只要一个工具有 CLI，它就能被脚本调用，也就可能被另一个 Agent 调用。再往下一层，如果它有 HTTP 接口，就能跨机器、跨系统被调用。

这时你会发现，“桌面软件”只是包装，“可调用能力”才是本体。一个工具能不能被部署、能不能排队、能不能远程调度、能不能接入流水线，取决于它把能力暴露到了哪一层。

懂下一层的人，看到的是可迁移的能力；不懂下一层的人，看到的是固定的产品形态。前者会问“这个能力有没有接口”，后者只会问“这个按钮在哪里”。两种问题，通向两种世界。

所以学习基础不是为了显得专业，而是为了减少幻觉。你越知道下一层是什么，越能分辨哪些限制是真的，哪些只是包装造成的错觉。创新常常不是发明一个全新东西，而是把一个已有能力从旧包装里取出来，接到新的地方。

第五章 · 工作区的墙是怎么消失的

微服务的麻烦，不在“微”，而在“彼此有关”。一个服务改了字段名，另一个服务的调用就可能坏；一个接口多了鉴权，前端、测试、文档都要跟着变。过去我们靠会议、接口文档和人工记忆来同步这些变化。AI 进来以后，问题变成：能不能让工具同时看见这些相互依赖的部分？

微服务，就是把一个大系统拆成许多小服务，每个服务负责一块功能，通过接口互相通信。它的好处是团队能分开开发，坏处是接口一变，很多地方都要跟着动。

大白话

大白话说：微服务像一组分工明确的小店。每家店独立经营，但菜单、送货和收银规则一改，隔壁店也会受影响。

旧边界：一个仓库一个世界

传统开发工具常常把一个代码仓库当作一个世界。打开 A 仓库，就改 A；打开 B 仓库，就改 B。这个边界对人很方便，因为团队、权限、部署通常也是按仓库分的。但对一次跨服务修改来说，它可能是错的边界。

比如订单服务把 `userId` 改成 `accountId`，支付服务、数据同步任务、管理后台都要跟着调整。如果 AI 只能看到订单服务，它也许能把本仓库改得很漂亮，却不知道外面已经坏了。你再让它打开另一个仓库，它又缺少刚才修改的完整上下文。

这不是模型不聪明，而是它看到的世界被切碎了。上下文的边界决定了它能推理的边界。

新边界：一次变更一个世界

如果把相关仓库放在同一个工作目录，情况就变了。AI 可以同时搜索字段名、接口定义、调用方、测试和文档。它能看到“这里改了，那里也要改”。这时候真正的边界不再是仓库，而是一次变更影响到的范围。

工作目录，就是工具当前能读取和修改的文件范围。过去它常常等同于一个项目；在 AI 协作里，它更应该围绕“要解决的问题”来组织。

这不是鼓励把所有代码都扔进一个大坑。权限、性能和噪音仍然要控制。关键是你要知道边界是人为设置的，可以按任务调整。要改一个局部函数，给一个仓库就够；要改跨服务协议，就应该让工具看见协议两端。

同步修改的本质是降低来回成本

跨服务修改最贵的不是敲代码，而是来回确认。A 服务改完，B 服务报错；B 服务补完，测试环境又发现文档没更新；文档改完，客户端 SDK 又漏了字段。每一次来回都在消耗人的注意力。

当 AI 同时看到多个相关项目，它能把这些来回压缩到一次修改里：先找影响面，再改调用链，再跑测试，再根据错误继续补。这就是“懂下一层”带来的现实收益。你不是让 AI 更神，而是给了它更完整的可观察范围。

这里的基础知识包括文件系统、文本搜索、依赖关系、接口契约和测试反馈。它们听起来都不炫，但正是这些东西决定 AI 能不能从“写一段代码”升级为“完成一次变更”。基础越扎实，你越会给 AI 布置一个真实可完成的工作现场。

第六章 · 一个 Agent 调另一个 Agent

当你发现一个 AI 工具有 CLI，脑子里应该亮一下。因为它不再只是你手边的桌面软件，而变成了一个可被别的程序调用的能力。再往前一步，如果它有 HTTP 接口，它甚至不必和你在同一台机器上。

接口，就是一个系统把能力交给外部使用时约定的入口。按钮是一种接口，命令行是一种接口，HTTP API 也是一种接口。接口不同，能组合的方式就不同。

大白话

大白话说：接口像插座。能力本身是电，插座长什么样，决定了谁能接上来、怎么接。

从人调用工具，到工具调用工具

过去我们习惯了“人坐在屏幕前，打开工具，点击按钮”。这是人调用工具。AI Agent 出现以后，多了一种结构：一个 Agent 可以根据任务，去调用另一个工具，甚至另一个 Agent。

Agent，这里指能观察任务、做计划、调用工具、根据结果继续行动的 AI 工作单元。它不只是一次回答，而是一个小循环：看见、判断、行动、检查。

比如一个主 Agent 负责读需求，发现需要改代码，就调用编程工具的 CLI；编程工具改完后，主 Agent 再调用测试命令；测试失败，它把错误交回编程工具继续修。整个过程里，人不需要每一步都点按钮，只需要设置目标、边界和验收标准。

这不是玄学自动化，它仍然建立在很朴素的东西上：命令能不能跑，参数能不能传，输出能不能读懂，错误能不能恢复。所谓 Agent 协作，落到底就是接口协作。

为什么懂接口的人更会创新

不懂接口的人，看到的是产品形态。这个工具是 IDE 插件，那个工具是网页应用，另一个工具是手机 App。懂接口的人，看到的是能力形态：谁能读文件，谁能生成补丁，谁能查数

据，谁能发通知，谁能验证结果。

一旦从能力形态看问题，组合空间就打开了。你可以让一个工具负责搜索，让另一个工具负责修改，让第三个工具负责审查；你可以把昂贵模型用在判断点，把便宜模型用在批量整理；你可以把本地执行和远程推理拆开，让任务在更合适的位置完成。

这里的关键不是“套娃式地让 AI 调 AI”，而是把复杂任务拆成清楚的能力节点。每个节点输入是什么，输出是什么，失败怎么表示，是否会改变外部状态。只要这些边界清楚，Agent 之间就能像软件模块一样组合。

组合不是越多越好

接口越多，系统也越容易混乱。一个 Agent 调另一个 Agent，如果没有清楚边界，可能互相等待、重复工作、把错误越传越远。真正好的组合，通常很克制：把判断、执行、验证分开，把危险操作加上权限，把不可逆动作留给人确认。

这又回到了基础。计算机科学里那些老概念：模块、协议、状态、幂等、日志、回滚，在 AI 时代没有过时。它们只是从后台工程原则，变成了智能系统能否可靠运转的前台条件。AI 越会做事，越需要这些条件给它铺路。

第七章 · 同步到异步：漂亮创新的样子

很多产品创新，回头看都像“本来就该这样”。远程控制 Agent 也是如此。早期思路很自然：手机要控制远程软件，就做一个同步远程工具，把远端屏幕传过来，把本地操作传过去。就像远程桌面，只是换到手机上。

后来有人换了想法：为什么一定要同步？如果用户只是要给 Agent 一个任务、看它的进展、必要时补一句话，也许不需要实时盯着屏幕。用异步消息把意图传过去，让 Agent 自己在远端工作，反而更合适。

同步通信，就是双方必须同时在线、等着对方回应。异步通信，就是一方先留下消息，另一方稍后处理，处理完再回传结果。

大白话

大白话说：同步像打电话，异步像发邮件。打电话要双方都在，发邮件可以各忙各的。

漂亮创新常常来自拆耦合

这里最关键的问题是：实时屏幕共享到底是不是必须的？如果任务是“帮我检查这个部署错误”，Agent 真正需要的是日志、代码、命令行和权限，不一定需要把整个桌面流传给手机。用户真正需要的是状态、结果和少量干预，也不一定需要每秒看画面。

耦合，就是两个东西被绑在一起，一个动另一个就必须跟着动。有些耦合是必要的，比如刹车和车轮；有些耦合只是历史习惯，比如“远程控制”天然等于“同步看屏幕”。

把不必要的耦合拆掉，系统会突然变轻。网络不稳定时，异步消息可以排队；用户离开手机，任务仍可继续；Agent 需要长时间运行，也不必占着一个实时连接。体验从“我远程操作一台电脑”，变成“我委托一个远端工作单元”。

为什么这依赖底层理解

如果你只看表面需求：“手机控制远程软件”，同步远程桌面几乎是唯一答案。因为“控制”这个词把你带进了旧模型：人要看见界面，人要点按钮，远端要实时响应。

但如果你往下一层看，就会把需求拆开：用户要表达意图，Agent 要接收上下文，远端要执行动作，系统要返回状态，关键节点要允许人介入。拆完你会发现，实时画面只是其中一种实现，不是客观规律。

这就是第一性原理的前奏。你把一个产品形态拆成元素和关系，才能判断哪些关系必须存在，哪些关系可以换。同步不是错，异步也不是永远更高级；真正重要的是，你知道为什么在这个场景里异步更合适。

AI 时代会出现很多类似机会。过去为了人操作而设计的同步界面，未必适合 Agent 工作；过去为了单机软件设置的边界，未必适合远程协作；过去为了人工审批设计的流程，未必适合机器先跑、人后验。创新不是把所有旧东西推翻，而是逐个问：这个绑定关系还必要吗？

第八章 · 第一性原理不是口号

“第一性原理”这个词很容易被说空。很多人把它当成“想得很深”的同义词，或者当成商业演讲里的高级词。其实它没有那么玄。它只是在提醒你：别先相信表面的组合方式，先把东西拆到更基本的元素和关系。

第一性原理，就是从最基本、最不容易被再拆的事实出发推理，而不是从习惯、类比或行业默认做法出发。

大白话

大白话说：别先问“别人都怎么做”，先问“这件事到底由哪些东西组成，它们为什么必须这样连在一起”。

第一步：找元素

任何系统都可以先拆成元素。一个 AI 工作流里，元素可能包括用户意图、模型、工具、数据、权限、执行环境、日志和验收标准。一个教育问题里，元素可能包括学生、知识、练习、反馈、时间和动机。先列元素，不急着下结论。

找元素的价值，是防止你被现成包装骗住。一个“AI 助手”听起来是一个东西，拆开后可能包含语言模型、检索系统、工具调用、记忆模块和权限系统。问题到底出在哪一层，只有拆开才看得见。

第二步：看关系

元素本身还不够，关键是它们怎么相互作用。模型读取数据，工具改变状态，日志记录结果，验收标准判断成功。关系错了，元素再强也没用。

比如一个模型能写代码，一个测试系统能跑测试，但两者之间没有自动连接，模型就不知道自己写错了。连接一旦建立，测试失败会变成反馈，反馈又会进入下一轮修改。系统能力不是元素相加，而是关系组织出来的。

反馈回路，就是行动产生结果，结果又影响下一次行动的循环。没有反馈，AI 只是一次性猜测；有了反馈，它才可能逐步接近正确。

第三步：分清约束和自由变量

拆到这里，真正重要的问题出现了：哪些关系必须尊重，哪些只是历史习惯？必须尊重的，叫约束；可以调整的，叫自由变量。

约束，就是你不能靠意愿取消的限制。网络有延迟，权限必须校验，数据会过期，生产系统需要回滚。自由变量，就是你可以选择的部分。界面可以换，调用顺序可以改，同步可以变异步，本地可以变远程。

第一性原理不是让你把一切推倒重来。恰恰相反，它让你更少乱动。因为你知道哪些地方动了会出事，哪些地方不动只是因为过去没人重新想过。

AI 越强，这种拆解越重要。模型会给你很多方案，其中有些很像创新，有些只是换了包装的幻觉。判断它们，不能只看语言是否顺滑，而要看它有没有尊重约束、有没有抓住自由变量。底层训练的意义，就在于你能看懂这张结构图。

第九章 · 哪些能动，哪些不能动

很多失败的创新，不是因为想象力不够，而是因为把不能动的东西当成了能动的东西。反过来，很多真正有价值的创新，也不是因为胆子特别大，而是因为看出某个大家以为不能动的东西，其实只是旧习惯。

第一性原理的用处，就在这里：帮你分清哪些能动，哪些不能动。

不能动的，是客观约束

客观约束不是领导说的规矩，也不是行业里的惯例，而是你绕不开的现实关系。计算机系统里，数据必须有来源；权限必须有边界；分布式系统必然面对网络失败；用户注意力有限；钱花出去就会进入成本结构。

AI 不会取消这些约束。它能帮你写更多代码，但不能让复杂系统不需要测试；它能帮你生成更多方案，但不能让方案不需要验证；它能帮你压缩沟通成本，但不能让责任边界消失。

大白话

大白话说：AI 可以帮你跑得更快，但不能取消地心引力。你跑得越快，越要知道地面在哪里。

这也是为什么强 AI 反而会惩罚基础薄弱的人。过去你写错一个判断，可能只影响一页文档。现在你把错误判断交给 AI 自动执行，它可能影响一批文件、一组客户、一个线上系统。速度越快，约束越重要。

能动的，是自由变量

自由变量常常藏在包装里。过去必须人点按钮吗？不一定，可能 CLI 就能调用。过去必须同步在线吗？不一定，异步队列也能工作。过去必须一个项目一个工具上下文吗？不一定，任务相关文件才是真上下文。过去必须由高级工程师逐行写脚本吗？不一定，AI 可以生成，人负责约束和验收。

迁移，就是把一个能力从原来的使用场景搬到新场景里。迁移能不能成功，取决于你有没有把能力本身和旧包装分开。

比如“写代码”这个能力，从人手里迁移到 AI，并不意味着软件工程消失。它只是把人的部分工作从敲字迁移到定义目标、暴露上下文、设计验证、处理例外。变量动了，约束还在。

好创新像换接头，不像砸机器

如果把系统想成一台机器，坏创新是拿锤子乱砸：这个流程太慢，删掉；这个审核麻烦，跳过；这个测试费时间，不跑。看起来快，实际是在剪断因果链。

好创新更像换接头。它保留必要约束，把不必要的连接方式换掉。比如保留权限审计，但把人工复制粘贴换成工具调用；保留测试反馈，但把人工跑命令换成 Agent 自动执行；保留人类最终责任，但把中间信息整理交给模型。

这就是底层知识带来的创新感。你不是凭感觉说“这里可以自动化”，而是能说明：这个环节的输入输出清楚，失败可检测，副作用可回滚，所以可以交给 Agent；另一个环节涉及不可逆决策、模糊价值判断或高风险权限，所以必须留人确认。

当你能这样说话时，AI 就不再只是一个潮流词。它变成一套可以被接入、限制、放大和验证的能力。你也从工具消费者，变成了系统设计者。

第十章 · AI 时代的底层训练

现在可以回到那句建议了：学好数学、物理、计算机。它不是在说每个人都要成为数学家、物理学家或系统工程师，也不是说你要把大学课本从头啃一遍。它真正指向的是三种底层训练。

数学训练结构感

数学的价值，不只是会算。AI 很会算，计算器更早就会算。数学真正训练的是结构感：从一堆现象里抽出变量，找出关系，判断哪些条件足够，哪些结论不能推出。

变量，就是在问题里会变化、会影响结果的量。结构，就是变量之间稳定的关系。

> 大白话
>
> 大白话说：数学帮你把一团乱麻变成几根线，知道拉哪根会动哪里。

用 AI 时，这种能力尤其重要。模型可以给你十个方案，但你要看出它们其实只在一个变量上变化，其他关键条件都没碰；模型可以写出一个漂亮论证，但你要看出中间偷换了前提。数学训练的不是答案，而是“这个推理有没有漏”。

物理训练约束感

物理让人变得不容易飘。因为物理总是在问：力从哪来，能量到哪去，摩擦在哪里，时间尺度是多少，误差会不会积累。它训练你相信世界有代价。

AI 产品里也有“物理”。模型调用有延迟，搜索有噪音，数据库有一致性问题，用户有耐心上限，自动化有错误成本。这些不是写几句提示词就能消失的东西。

学物理的底层收益，是你会本能地问约束。一个方案听起来很美，你会问：数据够吗？反馈多快？失败怎么发现？成本随规模怎么涨？这比背某个定律更重要。

计算机训练执行感

计算机科学让你知道，一个想法要变成现实，必须经过明确步骤。输入是什么，状态在哪里，接口怎么约定，异常怎么处理，权限怎么限制，结果怎么验证。

执行感，就是你能把“应该做”翻译成“机器如何一步步做”，并知道每一步可能在哪里失败。

AI 时代，执行感会变得更值钱。因为模型会替你生成很多步骤，但它不一定知道你的真实环境。懂计算机的人能把环境暴露给它，把任务拆细，把测试接上，把危险动作拦住。你不是和 AI 比谁更会写代码，而是在设计一个让 AI 不容易做错的系统。

普通人怎么开始

最好的开始不是收藏更多课程，而是带着真实问题往下一层看。你用一个 AI 工具，就问它读到了什么上下文；你让它改文件，就问文件边界在哪里；你让它调用接口，就问输入输出和副作用是什么；你看到一个创新产品，就问它拆掉了哪种旧耦合，又保留了哪些必要约束。

数学、物理、计算机可以慢慢学，但这种追问方式今天就能开始。每次多问一层，你就少被表象控制一点。每次分清一个自由变量和一个客观约束，你就多一点真正的创造空间。

AI 会继续变强，强到很多具体技能都会被重写。但底层能力不会因此贬值。恰恰相反，当答案越来越容易生成，判断答案能不能落地、能不能迁移、有没有违反约束，就成了更核心的工作。

所以“AI 越强，底层越重要”不是一句反技术的话。它是对技术最认真的态度：承认模型会越来越聪明，也承认真实世界仍然由结构、约束和执行组成。学基础，就是学会在这三者之间看路。