把黑箱拆开：手搓一个大语言模型

从零亲手造一个会说话的 LLM，破解它凭什么'像'有智能的迷

王建硕

面向对世界有好奇心的人 · 费曼式写法 · 由多个 AI 代理撰写与互相审校

导读

从零亲手造一个会说话的 LLM，破解它凭什么'像'有智能的迷

都说它是**黑箱**：喂它一句话，它就写代码、答问题、安慰你，流利得像藏着一个心灵，于是人们说它"快有意识了"，说没人真懂它在想什么。这本书要做的事很简单——把盖子撬开，让你自己往里看。看进去你会有点扫兴：里面没有魔法，没有小人，只有一堆矩阵乘法（把数字排成表格，按固定规则相乘再相加）和一个笨到好笑的目标——**猜下一个词**。

对，就这么个目标。给它半句话，它拼命猜下一个字该是什么，猜错了就挨罚、微调一下，猜上亿亿次。全书的悬念就是这句话：一个只会"猜下一个词"的程序，凭什么最后看起来像懂了？

这本书带你走的路

我们从最朴素的地方起步：怎么把"文字"变成程序能算的数字，怎么让相近的词在坐标里靠在一起；再看神经网络（一个能被数据反复捏形状、去逼近任意规律的大函数）怎么学，看它的心脏——注意力（读一个词时，自动决定该多看前文哪几个词）怎么让机器"抓重点"。然后是训练：所谓学习，不过是**沿着一面山坡一小步一小步往下走**，找最低点。走到后面，你会撞见真正有意思的问题：为什么模型一大就突然"开窍"？它一本正经地胡说（我们叫幻觉）是从哪冒出来的？它到底是真懂了，还是一个极其高明的模仿者？

压轴是动手：**两百行代码，手搓一个能说话的迷你 GPT**，再把开源模型请进你自己的电脑里跑起来。到这一步，它对你就不再是黑箱——是你亲手拼起来的东西。

怎么读，读完你会得到什么

你可以从第一章顺流而下，前一块是后一块的地基；也可以哪章标题勾住你就从哪进，每章都尽量自成一段。我不堆大词，每个术语第一次出现就当场用大白话说清；宁可给你一个精准的类比，也不塞一句正确的废话。

> 读完，你会换上一副新眼镜：**既不神化它，也不贬低它**——知道哪些是真本事，哪些是错觉。

而当有人再压低声音说"AI 快有意识了、没人能懂"，你不会慌。因为你亲手把它搭过一遍，你清楚那底下是什么。**拆开过，就不再怕了。**

第一章 · 它其实只在干一件事：猜下一个词

打开手机，在输入框里敲下"今天天气很"，别急着按下一个键——看一眼键盘上方那三个灰色的候选词。它们大概是"好""热""冷"。你还没想好要说什么，手机已经替你把可能的下一个词摆了出来。

这个每天被你用几百次、从没多看一眼的小功能，就是一整个大语言模型的核心。不是"很像",不是"简化版"——是同一件事。**ChatGPT 干的事，和你手机输入法联想下一个词，本质上是一模一样的动作。**区别只在于:它猜得准得离谱,而且能一个接一个地猜下去,猜出整篇文章。

这话听着像在贬低它。恰恰相反。这一章我想让你先牢牢抓住这个"笨"目标,因为整本书要拆的谜题就藏在这里:一个只会猜下一个词的机器,凭什么会写代码、会翻译、会讲笑话、看起来还挺懂事?

它到底在猜什么

先把话说准。我们平时说的 大语言模型(LLM,Large Language Model),拆开就是"大的、关于语言的、模型"。**"模型"你就理解成一个函数**——一头进去、另一头出来的那种加工机器;"语言"是说它加工的是文字;"大"是说这台机器内部有海量可以调的旋钮(后面几章你会看到具体是几千亿个)。

那它这个函数,输入是什么,输出是什么?

> 大白话
>
> 输入:到目前为止的一串文字(比如"今天天气很")。
> 输出:下一个词最可能是什么。
> 就这么一进一出。整个大语言模型,你可以先粗暴地理解成这一个动作。

但这里有个关键细节,很多人第一次听会愣一下:它给的**不是**一个词,而是一张"概率清单"。面对"今天天气很",它内部算出来的东西长这样:

> `好 → 35%`
>
> `热 → 22%`
>
> `冷 → 15%`
>
> `不错 → 9%`
>
> `……(剩下几万个词分掉剩下的概率)`

注意最后一行。它不是在"好/热/冷"三个里挑,而是给它认识的**每一个词**都打了个分——包括"香蕉""薛定谔""的""。"。只不过"香蕉"这种在这儿分到的概率低到近乎为零。所以更准确的说法是:语言模型就是这样一台机器,你喂它一段文字,它吐出一整张"下一个词该是谁"的概率表,覆盖它词表里的所有词。

你可能会问:那它怎么就决定说"好"了?——它掷了个骰子。一个被这张概率表加了权的骰子:"好"占了 35% 的面,所以最容易被掷中,但偶尔也会掷出"冷"。**这就是为什么同一个问题问它两次,答案会不一样。**这个"加权掷骰子"的机制,是它既能干活又会胡说八道的根源,我们留到第十章专门拆。这里你只要记住:模型算的是概率,选词是另一步。

一个词一个词,怎么就成了一篇文章

光猜一个词,顶多是个高级输入法。它怎么写出一整段的?

答案简单到有点狡猾:**把猜出来的词接回去,再猜下一个。**循环。

我们在脑子里跑一遍。假设它开了个头,要续写一句话:

1. 输入"今天天气很" → 猜出"好" → 现在句子变成"今天天气很好"
2. 把"今天天气很好"整个再喂回去 → 猜出"," → "今天天气很好,"
3. 再喂回去 → 猜出"适合" → "今天天气很好,适合"
4. 再喂 → "出门" → "今天天气很好,适合出门"
5. ……一直到它猜出一个特殊的"该停了"的信号为止

看清楚这个循环了吗?它**每一步都只做同一件事**——看着已有的全部文字,猜一个词。它没有"打腹稿",没有先想好整句话再往外蹦。它就像一个只能看着已经写下的字、往后接一个字的接龙玩家,接完再看,再接。这种"拿自己的输出当下一次的输入"的干法,有个专门的名字叫自回归(autoregressive)——"自"是用自己刚生成的,"回归"是再塞回去接着算。你不用记这个词,记住"写一句,接一个,再回头读一遍全文"这个画面就够了。

大白话

所以当 ChatGPT 一个字一个字往外蹦的时候,你看到的不是它在"打字",而是它真的在**一次一个词**地做几百上千次预测。屏幕上的流畅,是几百次"猜下一个词"接起来的。

凭什么是"猜下一个词"?

你可能觉得这个目标土得掉渣。这么简单的活,凭什么值得造几千亿个旋钮的庞然大物?

反过来想:要想每一次都猜准下一个词,机器被逼着必须搞懂什么?

看这几个填空,你替机器猜猜看:

- "水在 100 度会___" —— 要填"沸腾",它得懂点物理常识。
- "他把钥匙插进锁孔,轻轻一转,门___" —— 要填"开了",它得懂事情的因果顺序。
- "这道菜我等了一个小时才上,还是凉的,服务真是___" —— 要填"差",它得听出这是讽刺,不是夸奖。
- "法国的首都是___" —— 要填"巴黎",它得记住世界的事实。

看出门道了吗?**"猜下一个词"这个目标,像一张巨大的网,顺手把常识、因果、语气、事实全都网了进来。**因为一个词填得准不准,常常取决于你懂不懂前面那句话到底在说什么。你想让机器在千千万万种句子里都猜得准,就等于逼着它把这些东西全学会——不是我们教它常识,是"猜准词"这个死目标把常识从它身上逼出来的。

这是全书第一个、也是最重要的一个反直觉:**智能不是被直接设计进去的,是被一个笨目标"逼"出来的副产品。**没人给模型写过"讽刺=表面夸实际贬"这条规则。是它为了在海量含讽刺的句子里把下一个词猜对,自己在那几千亿旋钮里"拧"出了识别讽刺的能力。至于这个逼

出来的过程具体怎么发生——它读了多少文字、又是怎么根据猜错来调旋钮的——是第七、第八章的活,这里先按下不表。

这不是魔法,是被规模放大的模式

本书有一条暗线,从这一章就要埋下:**它没有魔法,也没有意识。**它不"想"表达什么,不"知道"自己在说什么,甚至不知道"巴黎"是个能让人去旅游的地方。它只是发现,在人类写下的天文数字般的文字里,"法国的首都是"后面,跟着"巴黎"这个模式出现得压倒性地多——于是它把这个模式,连同亿万个别的模式,压进了自己的旋钮里。

它是一台被人类文字喂大的、猜下一个词的**模式机器**。仅此而已——但正是这个"仅此而已",能长出让你惊掉下巴的东西。这中间的落差,就是这本书要一层层拆开的黑箱。

> 所以本质上,它就是一个填空高手:看着前面的字,猜后面的字,一个接一个,循环到底。你觉得它在思考,其实它在猜。

那么下一个问题自然就来了:机器不认字,"今天天气很"这几个汉字,是怎么变成它能计算的东西的?毕竟函数吃的是数字,不是文字。这就是下一章要拆的第一颗螺丝——**怎么把文字剪成小块、再给每块编上号。**

第二章 · 文字怎么变成数字：切词与编号

你现在正读着这行字。你的眼睛扫过一个个方块汉字，毫不费力地知道"苹果"是一个东西、"跑"是一个动作。可计算机不认识字。它只认识数字——准确说，只认识由 0 和 1 堆出来的数字。所以在上一章我们说"LLM 只干一件事：猜下一个词"之前，有个更土的问题必须先解决：**那个"词"，到底是怎么变成一串它能算的数字的？**

这一章我们把这个"词"——也就是 token（词元）——彻底拆开看。它是整本书里最不起眼、却最无处不在的东西：模型的输入是它，输出是它，计费按它算，上下文长度按它量，训练数据按它数。把它看清楚了，后面所有章节的单位才算统一了。

先把句子剪成积木块

假设你要把一句话交给一台只认识数字的机器。最笨的办法是给每个字一个编号：中文常用字几千个，那就 0 到几千号。英文更省，二十六个字母加标点，几十个号搞定。听起来行得通，但很快会出问题——后面会讲为什么。先记住这个动作本身有个名字。

分词（Tokenization），就是把一段连续的文字，切成一小块一小块的过程。切出来的每一块，叫一个 token（词元）——你可以把它想象成语言的"积木块"。一块可能是一个完整的词，也可能只是半个词、一个字、甚至一个空格、一个标点。

> 大白话
>
> token 不是"词"。别被翻译骗了。它是模型处理文字的最小单位，是人为切出来的一块砖，不一定对应你直觉里的"一个词"。

切完之后，第二步简单得可笑：给每一种不同的积木块，发一个固定的编号。所有编号凑成一张大表，叫 词表（vocabulary）——本质就是一本字典，左边是积木块，右边是它的编号。于是"苹果"可能变成 `15043`，"跑"变成 `892`。模型自始至终看到的都不是字，而是这一串编号。

> 一句 `"我爱苹果"`，经过分词器（tokenizer）先被切成 `["我", "爱", "苹果"]`，再查词表换成编号 `[2769, 4263, 15043]`。喂给模型的，就是最后这串数字。模型吐出来的，也是数字，比如 `892`，再反查词表变回 `"跑"` 给你看。**整个"理解语言"的过程，两头都是查表翻译，中间全是数字运算。**

顺便说个直观的换算：在主流模型里，一个 token 大约对应**四分之三个英文单词**，或者**一个左右的汉字**。所以"100 个 token"大概是一段七八十字的中文，或七十五个单词的英文——也就两三句话。这个比例先放心里，待会儿要用。

为什么不整词一个编号，也不单字一个编号

现在回到刚才那个"很快会出问题"的坑。切词有两个极端方案，两个都糟。

第一个极端：**一个完整的词发一个编号**（英文里叫 word-level）。听起来最自然，但英语里光是常见词就有几十万，还不算 `running` / `runs` / `ran` 这种同一个词的各种变形，每个都得单独占一个号。词表会膨胀到几十上百万。更致命的是未登录词（OOV, out-of-vocabulary）——就是"表里没有的词"。用户打一个词表里没收录的新词、拼错的词、生造的词，机器直接傻眼：这块积木不在我字典里，没编号，处理不了。而语言天天在造新词，这个窟窿永远堵不完。

第二个极端：**一个字符发一个编号**（character-level）。英文就 26 个字母，词表小到可怜，也永远不会遇到"表里没有"的字符。听着完美？代价是把语言切得太碎了。`apple` 五个字母，模型得自己从 `a-p-p-l-e` 这五块毫无意义的碎片里，重新拼出"苹果"这个概念。一句话本来十几个词，现在变成几十上百个字符，模型要处理的积木块数量暴涨，还得花大力气去学"哪些字母凑一起才算个意思"。太累，太浪费。

大白话

整词太胖：词表爆炸，还怕遇到没见过的词。单字太瘦：永远不怕生词，但把话剁成了肉末，模型累死。理想方案得在胖和瘦之间。

折中的聪明活：切成"子词"

真正被广泛采用的，是介于两者之间的 子词分词（subword tokenization）。核心思想一句话：**常见的词整块保留，罕见的词拆成更小的常见零件。**

怎么决定哪些该整块、哪些该拆？不靠人拍脑袋定规则，而是让算法在海量文本里数数。有一套经典办法叫 BPE（Byte Pair Encoding，字节对编码），它的逻辑朴素得像小孩搭积木。我们拿一个迷你语料现场演示一遍：

> 假设全部训练文本只有四个词，出现次数是：`low` 5 次、`lower` 2 次、`newest` 6 次、`widest` 3 次。第一步，把每个词拆成单个字符。然后反复做同一件事：**统计哪两个相邻块一起出现得最频繁，就把它们合并。**数一遍，`e` 和 `s` 挨着出现 9 次（newest 里 6 次、widest 里 3 次），全场最频繁——合并成 `es`；再数，`es` 和 `t` 挨着 9 次——合并成 `est`；接着 `l` 和 `o` 合并成 `lo`，`lo` 再和 `w` 合并成 `low`……几轮下来，`est` 和 `low` 都成了独立的积木块。于是再来个没见过的新词 `lowest`，不用慌，直接切成 `low` + `est` 两块现成的零件。

真实的 BPE 干的就是这件事，只不过语料是几百亿词、合并几万次。那些高频组合——常见的词根、词缀、整词——自然而然长成了独立的积木块，而没长成的稀有组合，就还是一堆小碎片。

结果就是：`apple` 这种天天见的词，会作为一整块留在词表里，一个编号搞定。而一个生僻词，比如 `tokenization`，可能被切成 `token` + `ization` 两块；再冷门些的、模型从没整块见过的词，会被拆得更碎，比如 `[" un", "believ", "ability"]` 这样。**这就是为什么生僻词会被拆碎——不是机器歧视它，而是它没常见到"配得上"一个自己的编号。**

这套机制有个漂亮的副作用：OOV（表里没有的词）问题被彻底干掉了。因为无论你打出多离谱的词，最坏情况也能一路拆到单个字符（甚至单个字节），而字符和字节的数量是有限且固定的，全都在表里。**再没有"这块积木我不认识"的时刻了。**任何词都能被表达，代价只是拆得碎一点。代价也体现在价格上：越生僻的表达，拆出的 token 越多，而模型的一切开销都按 token 计。

一个 token 里，到底装了多少信息

现在到了本章最值得停下来想一想的地方。一个 token，看起来不过是词表里的一个编号，能有多大信息量？惊人地大——只要你算一笔账。

主流大模型的词表大约是**十万个 token**。也就是说，模型每生成一个字块，本质上是在做一道**十万选一的选择题**：从十万种可能的下一块里，挑出"对的"那一块。而信息论告诉我们，从 N 个等可能的选项里确定一个，所携带的信息量是 $\log_2 N$ 个比特（bit）——计算机信息的最小单位，一次"是或否"的回答。十万个选项，$\log_2(100000) \approx$ **16.6 比特**。

16.6 比特是什么概念？抛硬币，猜正反，猜对一次得到 1 比特信息。**模型每吐出一个 token，相当于连续猜对十六七次硬币的正反面。**它回答你一个问题，轻轻松松写了 2000 个 token——那就是连续猜对三万三千多次硬币，次次都中。换成另一杆秤：一副洗匀的扑克牌有 52! 种排列，从中猜中你手里那一把的顺序，约等于 226 比特——也只值模型说上十四个 token。

当然，严格说这十万种选择不是等可能的——"今天天气很"后面接"好"的概率远高于接"氢弹"，所以平均每个 token 实际携带的信息没有 16.6 比特那么满。但反过来说，正因为模型面对的是**真正的十万选一**，而不是从两三个候选里挑顺眼的，它每一次选择都可能出乎你的意料——这正是它"会说人话"的数学地基：它的表达空间，大得近乎自由。

> 大白话
>
> 别把"一个编号"看扁了。编号本身是死的，但"从十万个编号里选中这一个"这个动作，一次就顶你连猜对十六次硬币。一段话几千个 token，就是几万个这样的抉择连成串——信息密度就是这么堆出来的。

再往上叠一层，尺度会更吓人。训练一个大模型要"读"多少 token？GPT-3 是 3000 亿个，后来几代模型是**十几万亿个**。十几万亿 token 是多少文字？一本书大约 15 万个 token，十几万亿 token 相当于**一亿本书**——而人类有史以来出版过的全部书籍，估计也就一亿三千万种上下。换句话说，这些模型在几个月的训练里"读"掉的文字，和整个人类文明几千年来写下的书的总量，是同一个量级。而你自己呢？一个人从识字到八十岁，听、说、读、写全算上，一辈子经手的文字折成 token 大约是**十亿量级**——不到训练集的万分之一。它为什么显

得什么都懂一点？因为它"见过"的话，比你多出四五个数量级，而这"见过"的每一次，都是一次十万选一的练习。

这笔账还有一个务实的结果：**token 成了这个行业通用的度量衡。**你在 API 账单上看到的费用，按 token 计；模型"一次能看多长"，按 token 量；生成速度的快慢，按"每秒多少 token"算。下次看到"上下文 128k tokens"，别再脑补成"12.8 万个字"——它是 12.8 万块积木，英文大约是一本两百页的书，中文大约是十几万字、一部长篇小说的篇幅。整个行业的成本、速度、能力的标尺，都钉在这块小小的积木上。

这解释了几件你可能纳闷过的事

为什么模型对某些字符拼写会犯迷糊？比如数 `strawberry` 里有几个 `r` 会出错。因为它压根没逐个字母地看这个词——它看到的是两三块子词积木，字母藏在积木内部，它不一定拆得开。

为什么中文和英文"计费"感觉不一样？主流大模型按 token 数收费和计算。英文一个常见词往往就一个 token，而中文因为字符集庞大、组合多，常常是一个汉字占一个 token，生僻字还会被拆成多个字节 token。**所以同样一段话，中文消耗的 token 数往往比你以为的多。**这不是玄学，就是分词器切出来的块数不同。

所以本质上，它就是……

所谓"让机器读懂文字"，第一步毫无浪漫可言：把连绵的语言剪成一堆预先编好号的积木块，再把号码串成一列数字递进去。机器从头到尾没见过一个"字"，它见到的只是 `[2769, 4263, 15043]` 这样的整数序列，然后对这串整数做运算，再吐出一个整数，我们再翻译回字给自己看。

可这里有个新问题：编号 `15043`（苹果）和编号 `892`（跑）之间，除了大小不同，没有任何关系。`15043` 不比 `892` 更"甜"，也不更"红"。注意，这跟前面说的"信息量惊人"并不矛盾——惊人的是"从十万里选中这一个"这个动作，而不是门牌号本身。门牌号是死的、任意的，纯粹是查表查出来的：它完全不携带"意思"。可我们明明希望机器知道"苹果"和"香蕉"沾点边、和"跑"八竿子打不着。**光有门牌号不够，还得给每个词一个能表达"含义"的坐标。**这，就是下一章的事。

第三章 · 让词有了'意思'：语义坐标系

上一章我们把句子剪成了积木块，每块贴上一个编号。可编号这东西，冷酷得没有一点道理：`猫` 是 5182，`狗` 是 9047，`桌子` 是 5183。就因为 5183 紧挨着 5182，桌子就比狗更像猫吗？当然不是。编号只是一个户口本上的名字，它不携带任何"意思"。

可我们明明知道：猫和狗是一类的（都是毛茸茸会叫的动物），"高兴"和"愉快"几乎是同义词，"巴黎之于法国"就像"东京之于日本"。这些关系客观存在，只是编号完全丢掉了它们。于是本章要解决的就是这一件事：**怎么给每个词一个"有意思"的身份，让含义变成机器能算的东西。**

把词从"名字"升级成"坐标"

换个思路。别再用一个孤零零的编号代表一个词，改用**一串数字**——一个坐标。

先从我们能想象的二维说起。假设我造一张地图，横轴叫"是不是动物"，纵轴叫"体型大小"。那么：

- `猫` 可能落在 (0.9, 0.2)：很像动物，体型小。
- `狗` 落在 (0.9, 0.4)：一样很像动物，体型稍大。
- `桌子` 落在 (0.05, 0.6)：几乎不是动物，中等大小。

看出门道了吗？在这张地图上，猫和狗紧挨着，桌子被甩到老远。**"意思相近"变成了"坐标相近"**——两个词像不像，现在可以用它们在地图上的距离来量，而距离是纯数学，机器最擅长算。这就是整章的核心魔术。

> 大白话
>
> 编号是"你叫几号"，坐标是"你站在哪"。前者只能比较相等或不相等，后者能比较远近、方向、甚至做加减。我们要的，正是后面这些。

这串代表一个词的坐标，就是 词向量（word embedding）——说白了，就是"用一长串数字给一个词定的位置"。`embedding` 这个词英文原意是"嵌入"，意思是把词嵌进一个多维空

间里，让它有个落脚点。

二维不够用，那就几百维

两根轴显然不够。"动物性"和"体型"能分开猫和桌子，可分不开"高兴"和"悲伤"（它们动物性都是 0、体型都无意义）。词与词之间的差别维度太多了：褒义还是贬义、抽象还是具体、正式还是口语、是不是国家名、是不是颜色……

怎么办？加轴。真实的语言模型里，每个词的坐标不是 2 个数，而是几百到上万个数。GPT-2 用的是 768 维，后来的大模型常见到几千维。也就是说，每个词都活在一个几百上千维的空间里，占据一个精确的位置。

> 大白话
>
> "768 维"听着吓人，其实一点都不玄。它就是一串 768 个小数，比如 `[0.12, -0.44, 0.03, ……]`。维度多，只是因为要区分的"意思的侧面"太多了，两三个轴根本装不下。你没法在脑子里画出 768 维的图，但机器算距离时，768 个数和 2 个数用的是同一个公式，一点不费劲。

这里要破除一个常见的误解：**没有哪根轴是人手工指定"这根管动物性、那根管褒贬"的。**我上面说的横轴纵轴只是为了让你有画面感。真实模型里,这几百个数字是训练时自己长出来的（下一章讲神经网络怎么"长",第七章讲怎么"训练"），每根轴具体代表什么,往往说不清、也不需要说清。我们只在乎一件事:**意思相近的词,坐标最终会凑到一起。**

机器凭什么知道两个词像？靠"看它们跟谁做邻居"

凭空给每个词编几百个数,数从哪来?靠的是一条朴素到近乎粗暴的规律,语言学上叫 分布假设(distributional hypothesis)——一句话:**一个词的意思,由它经常和哪些词一起出现来决定。**

"你是谁,看你跟谁混。"扫过海量文本你会发现,`猫` 和 `狗` 周围反复出现 `喂`、`可爱`、`养`、`叫`;而 `桌子` 周围是 `搬`、`木头`、`放`。既然猫和狗的"邻居圈"高度重合,模型就把它俩的坐标往一块儿挪。整个训练过程,本质就是不停微调每个词的坐标,让"经常做邻居的词坐标靠近,老死不相往来的词坐标拉远"。

大白话

注意,机器从头到尾没人告诉它"猫是一种动物"。它只是数了无数遍谁和谁一起出现,坐标就自己排布成了我们眼中"有意义"的样子。意思不是被灌进去的,是从"共同出现的统计规律"里自己浮出来的。这一点,是破除"AI 懂词义"神秘感的关键。

压轴戏:意思居然能做加减法

词一旦变成坐标,一件很反直觉的事就发生了:**它们能像向量一样加减,而且加减的结果有意义。**最著名的例子是:

> `国王 − 男人 + 女人 ≈ 王后`

把它当成地图上的位移来读就通了。从"男人"走到"国王",你走的是某个方向、某段距离——这段位移大致可以理解为"加上王室身份、加上统治者这层意思"。现在你站在"女人"的位置上,朝**同一个方向、走同样的距离**,落脚点会非常接近"王后"。

说人话:*国王之于男人,正如王后之于女人*这句关系,在坐标空间里变成了一段可以平移的箭头。既然"性别差异"是一个固定方向,"王室身份"是另一个固定方向,那含义就真的能被拆成一根根方向来加减。类似的还有 `巴黎 − 法国 + 日本 ≈ 东京`(把"这个国家的首都"当成一个可平移的方向)。

这不是被人为设计出来的彩蛋,而是"用邻居定坐标"这套办法自然而然的副产品——含义的规律,自己在几百维空间里排成了整齐的几何。**到这一步,"意思"这个虚头巴脑的东西,被彻底翻译成了坐标、距离和方向,全是机器算得动的数学。**

(小声提醒:`≈` 是"约等于",不是精确相等。真实模型里结果是"离王后最近的那个词",偶尔也会翻车指向别的词——但方向对得惊人,足以说明问题。而且这类经典例子多来自早期专门的词向量模型 Word2Vec,今天大模型内部的词向量更复杂,但"坐标承载意思"这个内核完全一样。)

一个还没解决的破绽

本章我们让每个词有了固定坐标。可"意思相近坐标就近"有个致命漏洞:**同一个词,坐标却是死的。**

想想"苹果"。在"我啃了一口苹果"里它是水果,在"苹果发布了新手机"里它是公司。可到目前为止,无论哪句话,"苹果"用的都是同一串坐标——它没法根据上下文变脸。一个词的真正含义,明明得看它周围站着谁才能定啊。

所以本质上,这一章我们只做了一件事:**把每个词钉在一张几百维地图上的固定位置,让"意思"第一次变成了能算距离、能做加减的坐标。**但这只是每个词的"出厂默认含义"。怎么让"苹果"在读到"手机"时自己挪一挪、变成"那家公司"的意思?——那要靠让词与词互相"看一眼",这正是第五章那个全书心脏级别的机制。在此之前,下一章我们得先搞清楚:那个能把这些坐标揉来揉去、还能自己长出规律的"神经网络",到底在算什么。

第四章 · 神经网络到底在算什么：会拟合的函数

上一章你已经见过一件近乎魔法的事：把"国王"减掉"男人"再加上"女人"，落点竟然离"王后"很近。含义变成了能加减的坐标。但一个问题被我们悄悄跳过了：**那些坐标，是谁算出来的？**词向量不是天上掉下来的，它背后有台机器在把数字搓来搓去。这台机器，就是本章的主角——神经网络。

你八成听过它的传说：模仿人脑、由"神经元"组成、能"学习"能"思考"。现在把这层皮撕掉。神经网络（neural network）跟大脑的关系，大概就跟"热狗"跟"狗"的关系一样大——名字是历史遗留的浪漫，实物是另一回事。它不思考，不理解，也没有小人在里面开会。它只是一个巨大的、可以调的**函数**。

先把"函数"这个词还给你

函数听起来吓人，其实你天天在用。函数（function）就是一台"进去一个东西、出来一个东西"的机器：你喂给它输入，它吐给你输出。自动售货机是函数（投币+按钮→一罐可乐），汇率换算是函数（100 人民币→约 14 美元），温度换算 `F = C × 1.8 + 32` 也是函数。

关键在于：一个函数里往往藏着几个**可以拧的旋钮**。温度换算里那个 `1.8` 和 `32` 就是旋钮。如果我不告诉你摄氏和华氏怎么换，只丢给你一堆对照数据——0 度对 32、100 度对 212、37 度对 98.6——你能不能反过来把 `1.8` 和 `32` 这两个旋钮的值试出来？能。**这就是神经网络干的唯一一件事：给定一堆"输入→正确输出"的例子，把自己身上的旋钮拧到能拟合这些例子。**

大白话

所谓"训练一个神经网络"，翻译成人话就是：有一个带着几亿个旋钮的公式，我拿海量例子去反复微调这些旋钮，直到这公式喂进输入能吐出对的输出。没有学习的灵光，只有拧旋钮。

拆开看：一层就是一次矩阵乘法

那这个"带旋钮的公式"长什么样？核心零件简单到你会失望：**一堆乘法和加法**。

假设输入是三个数 $[x_1, x_2, x_3]$——比如上一章那种词向量的一小截。我想把它变成两个数的输出。我要做的，就是给每个输入配一组"权重"（就是旋钮），加权求和：

$$out_1 = w_{11}\cdot x_1 + w_{12}\cdot x_2 + w_{13}\cdot x_3 + b_1$$

$$out_2 = w_{21}\cdot x_1 + w_{22}\cdot x_2 + w_{23}\cdot x_3 + b_2$$

那些 `w` 是权重（weight）——每个输入该被看重几分，就是可拧的旋钮；那些 `b` 是偏置（bias）——一个不管输入是啥都要加上的基准值，好比调音时的"底噪"。当你把成千上万个这样的加权求和整整齐齐码在一起，数学家给它起了个名字叫矩阵乘法（matrix multiplication）。别被吓到，它的本质就是刚才那一堆"乘一乘、加起来"，只是用一个紧凑的记号一次性写完几百万个。

> 大白话
>
> 矩阵乘法不是什么高深操作，它就是"批量的加权求和"。你有一排输入数字，有一张旋钮表，把它们对着乘一遍加起来，得到一排新数字。GPU 之所以为 AI 而生，就是因为它一秒能做天文数字次这种乘加。

光会乘加还不够：得把线掰弯

但只有矩阵乘法，会撞上一堵墙。你把一次加权求和的结果，再喂给下一次加权求和……数学上有个残酷的事实：**一堆线性变换叠起来，等于一个线性变换**。加权求和的加权求和，还是加权求和。这意味着你哪怕堆一百层，本事和一层一模一样——只能画直线。而真实世界几乎没有一件事是直线：房价和面积不是直线，一个词该不该接在另一个词后面更不是直线。

怎么破？在每层加权求和之后，塞一个**非线性**的小动作，把直线"掰弯"。最常用的那个简单到离谱，叫 ReLU（读作"瑞露"，全称 Rectified Linear Unit）：

> `ReLU(x) = 如果 x 大于 0 就保留 x，否则就变成 0`

就这么一句"负数一律拍平成零"。别小看它。正是这个不起眼的拐点，让网络有了"分段"的能力——不同区间可以有不同的斜率。**无数个这样的小拐点拼在一起，就能逼近任何弯弯曲曲的曲线。**这不是比喻，而是一条被证明过的定理（叫万能逼近定理（universal approximation theorem））：只要旋钮够多，这种"乘加+掰弯"的结构，能以任意精度拟合几乎任何函数。

大白话

打个比方：矩阵乘法给你一根根笔直的木条，非线性给你在木条上折弯的能力。光有直木条你只能搭个方框；能折弯之后，你用足够多的折线就能拼出任何形状的轮廓——一只猫、一条心电图、语言里的规律，都是"形状"。

把它串起来：一个会认数字的例子

在脑子里跑一遍。假设我要认手写数字：输入是一张 28×28 的小图，摊平成 784 个数（每个数是一个像素的深浅）。

1. 第一层：784 个数进去，做一次矩阵乘法，比如压成 128 个数，再每个过一遍 ReLU 掰弯。
2. 第二层：128 个数进去，再乘一次、再掰一次，压成 10 个数。
3. 这 10 个数分别代表"像 0 的程度""像 1 的程度"……哪个最大，就猜它是几。

整个过程没有任何一步在"看"这张图、在"想"这是什么。从头到尾，只是一串数字流进来，经过几轮"乘加、掰弯、乘加、掰弯"，另一串数字流出去。**它不认识数字，它只是被一堆例子把旋钮拧到了——碰巧对这些像素组合会吐出对的答案。**

那"智能"到底藏在哪

你可能会追问：既然只是乘加和掰弯，那让 GPT 能写诗、能答题的本事在哪？答案不在结构里，而在**旋钮的具体数值**里。结构是死的、笨的、通用的；一个刚出厂、旋钮全是随机数

的网络，吐的是纯粹的乱码。它和训练好的 GPT，零件一模一样，差别只在于那几千亿个旋钮被拧成了什么值。

> 大白话
>
> 所以"模型"这个词，物理上就是一大坨数字——那几千亿个权重的值。你下载一个开源大模型，下载的不是什么灵魂，就是一个几百 GB 的旋钮设定文件。

这里也埋着本章最该破除的迷思：网络里没有"概念"、没有"知识"以人能读懂的方式存着。所谓"它懂了语法"，物理上只是某些旋钮恰好被拧到了一个让语法规律能被拟合出来的组合。它不是学会了规则，而是长成了一个"碰到这种输入就吐出那种输出"的形状。

那么问题只剩最后一个，也是整本书最硬的一个：这么多旋钮，到底是**怎么**被拧到对的值上的？总不能真靠人手一个个试——几千亿个旋钮，试到宇宙热寂也试不完。这里有一套精巧的"自动拧旋钮"的办法，叫训练。它怎么知道该往哪拧、拧多少？那是第七章"蒙眼下山"的故事了。

但在那之前，我们得先看看这台"乘加+掰弯"的机器里，最要命也最漂亮的一个零件：它怎么让句子里的词彼此看一眼、决定谁该听谁的。所以本质上，神经网络从来不是一个会思考的大脑，**它只是一个大到吓人、却笨到骨子里的可调函数——真正让它显得聪明的，是下一章那个叫"注意力"的机关。**

第五章 · 灵魂登场：注意力，让词互相看一眼

上一章你把神经网络看穿了：它不思考，只是个巨大的、可以调参的函数，输入一串数字，吐出一串数字。可这里有个大窟窿。假设我们要处理这句话：

> "我把苹果放桌上，它还是热的。"

"它"指谁？苹果，还是桌子？你一眼就知道是苹果——因为"热"跟苹果搭得上（刚烤过的、烫手的苹果），跟桌子搭不上。你不是孤立地看"它"这个字，你是**回头瞟了一眼前面的词**，才确定它的意思。

但第三章里我们给每个词分配的那套坐标，是**死的**。"它"这个词的坐标，写在《说文解字》里就那样，管你前面放的是苹果还是冰块。一个只会查固定坐标的函数，永远搞不清"它"到底指谁。这就是那个大窟窿：**词的意思是随上下文变的，而固定坐标给不了这个变化。**

本章要登场的这块拼图，就是专门来补这个窟窿的。它叫注意力（Attention）——说白了，就是让每个词在算自己意思的时候，**能回头看一眼句子里别的词，并决定该重点听谁的**。这一个机制，是今天所有大模型的心脏。拆开它，你就拆开了整台机器最关键的那一下。

先看一场会议

想象一个开会的场景。轮到"它"这个词发言，它要搞清楚自己是谁。它不能瞎猜，得问问在座的其他词。于是它环视全场，心里揣着一个问题：**"谁是可以'热'的东西？"**

在座每个词——"苹果""桌上""放"——都举着一块牌子，上面写着自己"能提供什么信息"。"苹果"的牌子上写着"我是个能烤热的实物"；"桌子"的牌子写着"我是个平面家具"。"它"拿着自己的问题，挨个对照这些牌子，发现"苹果"的牌子跟自己的问题最对得上。于是它主要听"苹果"的，几乎不理"桌子"。

就这么个动作。注意力干的事，从头到尾就是这场会议：**一个词带着问题，去匹配别的词举的牌子，谁匹配得好就多听谁的。**把这句话记住，下面所有术语都只是它的严格版本。

Q、K、V：三张从同一个词里长出来的东西

现在把会议里的三样东西起个正式名字。它们的英文缩写你到处都会见到，其实平平无奇：

- 查询（Query，Q）：一个词**要问的问题**。上面例子里"它"心里那句"谁是可以热的东西？"，就是它的 Q。
- 键（Key，K）：一个词**举出来招揽别人的牌子**，用来被别人的问题匹配。"苹果"举的"我是能烤热的实物"，就是它的 K。
- 值（Value，V）：一个词**真正要交出去的内容**。别人决定听它之后，它实际递过去的那份信息，就是它的 V。

> 大白话
>
> 牌子（K）和内容（V）为什么要分成两个？打个比方：K 是商品的**标签**，V 是**商品本身**。你在货架上是靠标签找东西的（匹配），但你买回家用的是商品本身（信息）。用标签来搜、用商品来用——分开，才能"招牌写得好听"和"货真价实"各管各的。

关键的一点，别被三个名字唬住：Q、K、V **都是从同一个词的坐标里算出来的**。还记得第四章的函数吗？拿"苹果"那串坐标，分别乘上三个不同的矩阵（就是三套可调的旋钮），就得到"苹果"的 Q、K、V 三串新数字。同一个词，戴上三顶不同的帽子：作为提问者时用 Q，作为被匹配的招牌时用 K，作为被取用的内容时用 V。

那三个矩阵里的数字从哪来？没人手写，全是**训练时自己学出来的**——怎么学，是第七章"下山"的活，这里先记住：这些旋钮不是设计的，是练出来的。

一个词的意思，是怎么被"重算"出来的

把会议拆成能在脑子里跑一遍的四步。就以"它"这个词为例，看它怎么算出自己的新意思：

1. **打分（对牌子）**：拿"它"的 Q，去和句子里**每一个**词的 K 做一次点乘。点乘你可以理解成"方向对不对得上"的打分——两串数字越同向，分越高。于是"它"对"苹果"打了高分，对"桌子"打了低分。
2. **归一（分蛋糕）**：把这些分数过一遍 Softmax——它的作用就是把一堆高低不齐的分数，压成一组加起来正好等于 1 的比例，像切蛋糕。比如变成"苹果 0.8、桌子 0.05、放 0.05……"。这组比例，就是"它"这一刻的**注意力权重**：注意力该往哪撒、撒多少。

3. **加权求和（按比例拿货）**：按这组比例，去把每个词的 V 加起来。0.8 份"苹果"的 V，加 0.05 份"桌子"的 V……混成一串新数字。
4. **这串新数字，就是"它"的新意思**——一个被上下文重新算过、已经**掺进了大量"苹果"信息**的"它"。

大白话

看明白发生了什么吗？进去的时候，"它"还是那个谁都指的、空洞的"它"；出来的时候，它已经变成"一个热乎乎、指向苹果的它"。**固定坐标那个窟窿，就是这样被补上的：词的意思不再是查表查来的死值，而是每次都看着上下文现算的活值。**这就是注意力这一下的全部魔力——其实一点都不魔法，就是打分、切蛋糕、按比例拿货。

它同时在替所有词做这件事

上面只跟着"它"走了一遍，但机器是**并行**地替句子里每个词都做一遍的。"苹果"也在提它的问题、"放"也在提它的问题——每个词都当一次提问者，环视全场，重算一遍自己。所以一句 `n` 个词的话进去，出来还是 `n` 个词，但每个词都吸收了它该吸收的上下文。

这也解释了一件你可能早就好奇的事：为什么大模型能"并行"训练、比老式逐字处理的网络快那么多？因为这一堆打分，本质就是**一次大矩阵乘法**——把所有词的 Q 排成一摞、所有词的 K 排成一摞，一乘，全部两两之间的分数一次算完。GPU 最擅长的就是这个。注意力之所以能撑起今天的大模型，一半是因为它聪明，另一半是因为它**算得起**。

当然也有代价：`n` 个词两两打分，是 `n×n` 张分数表。句子长一倍，这张表大四倍。这就是为什么"上下文能塞多长"是个真金白银的成本问题——这笔账，留到第十二章讲"记忆"时再算。

一个必须提的细节：不许偷看后面

还记得全书第一章那个核心动作吗——**猜下一个词**。模型是从左往右一个个往外蹦词的。那么当它在预测第 5 个词时，它**只能看前 4 个词**，绝不能偷看第 6、第 7 个——因为那些还没生成，现实里根本不存在。

所以在打分那一步，会人为地把"看向后面的词"那些分数，强行摁成负无穷（过完 Softmax 就变成 0，一点都分不到）。这个动作叫 掩码（Masking），就是拿张纸把后文盖住、只让每个词看到自己和左边。别小看这一盖：正是它逼着模型学会"只凭前文预测下一个词"，而这恰恰是整本书从第一页起就在说的那件唯一的事。

所以本质上它就是……

注意力，剥到底，就是一句话：**让每个词带着自己的问题（Q）去匹配所有词的招牌（K），按匹配度分配注意力，再按这个比例把大家的内容（V）搅拌成自己的新意思。**没有理解、没有意图，只有打分、切蛋糕、按比例混合——三步小学算术，被并行地、成千上万次地叠加起来。

但你可能已经嗅到一点不对劲：一个词就**只能带一个问题**吗？"它"想问"谁是热的"，可能同时还想问"谁是我指代的那个名词"——一个问题显然不够用。而且，光靠这么一层"互相看一眼"，真能堆出会写代码、会讲笑话的庞然大物？

下一章，我们就让每个词**同时问好几个问题**（这叫多头注意力），再把这样的注意力层**摞成几十层高的塔**——那座塔，就是 Transformer。心脏已经在跳了，接下来是给它接上整副身体。

第六章 · 把注意力堆成塔：Transformer 全貌

上一章结束时，你脑子里应该刚跑完一场"开会"：一句话里的每个词，都在扭头看别的词，决定该多听谁的。`the animal didn't cross the street because it was too tired` 里的 `it`，靠注意力把目光投向 `animal` 而不是 `street`。很妙。但也留了个尾巴：一次注意力，真的够吗？一个词只能"看一眼"，它想同时关心的东西可不止一件。

这一章，我们把那个心脏拿出来，看它怎么被复制、被加固、被叠成一座塔——最后从塔顶掉下来的，是一串"下一个词是什么"的概率。这座塔，就是 Transformer（变换器），如今几乎所有大模型的骨架。

一个头不够看：多头注意力

回到那句话。`it` 该看谁，其实不是一个问题，而是好几个问题叠在一起：

- 语法上，`it` 指代哪个名词？（该看 `animal`）
- 情绪/状态上，谁"累了"？（还是 `animal`，但这是另一条线索）
- 句子结构上，`because` 引出的从句和主句怎么挂钩？

如果只有一套注意力，它得把这些关心的对象糊成一团平均值——就像开会时你只有一双耳朵，三个人同时说话，你听到的是一锅粥。解决办法笨得可爱：**那就配好几双耳朵**。

这就是 多头注意力（Multi-Head Attention）。所谓"头"，就是一整套独立的注意力计算——各有自己的一组 Q、K、V 权重（还记得吧：Query 是"我想找什么"，Key 是"我有什么标签"，Value 是"我能提供什么内容"）。

> 大白话
>
> 把"头"想成会议室里派出的几名助理。每人拿着不同的任务清单去听会：一号只记"谁指代谁"，二号专盯"谁在描述谁的状态"，三号管"时间先后"。他们各听各的、各记各的，最后回到你面前，把几份笔记拼在一起。你一下子拿到了好几个角度的信息，而不是一锅粥。

典型的模型会配多少个头？GPT-2 的中等版本用 12 个头，大模型常见 16、32 乃至更多。关键是，这些头**并行**算——它们不排队，同时开工。每个头输出一份结果向量，然后把这些向量首尾拼接起来，再过一个矩阵把维度压回原样。于是"一个词"就同时携带了十几个视角的信息。

你可能会担心：头一多，计算量是不是暴涨？不会。诀窍是把每个头做"瘦"：如果词向量是 768 维、有 12 个头，那每个头只在 `768 / 12 = 64` 维的小空间里干活。总活儿量和一个大头差不多，但被切成了 12 份各管一摊。省钱，还更聪明。

叠得越高越糊涂：残差连接来兜底

一个头也好、多个头也罢，一层注意力能做的事有限。真正的威力来自**把这样的层堆很多层**：GPT-2 small 有 12 层，GPT-3 有 96 层，更大的模型上百层。每一层都在前一层的理解之上，再加一点更抽象的关系。

但堆深了会出事。你把一个词向量往上传，经过几十层反复揉搓、乘来乘去，最初那点信息很容易被冲刷得面目全非——就像一句话在几十个人之间口耳相传，传到最后完全变味。更麻烦的是训练：第七章会讲到，训练要把错误从塔顶一路"反推"回塔底，塔太高，这个信号一层层衰减，传到底下几乎归零，底层就学不动了。

Transformer 用一个朴素到你会想"就这？"的办法解决它：残差连接（residual connection）。

> 大白话
>
> 每一层不直接输出"我算出的新结果"，而是输出"原来的东西 + 我这一层做的改动"。用公式说就是 `输出 = 输入 + 这一层算出来的东西` 。这条把输入原样加回去的线，叫"跳线"（skip connection）。

为什么这一招这么关键？两个原因，都很实在：

- **信息不丢。**每一层顶多是在原稿上"批注修改"，而不是重新誊抄。哪怕某一层暂时啥有用的都没算出来，原稿也原样传下去了，不会越传越糊。
- **训练信号能穿透。**那条加回去的跳线，给反推的错误信号修了一条"直达电梯"，可以绕过中间层直接下到底层。于是几十上百层也带得动。

可以说，没有残差连接，就没有"深"的 Transformer。它是让塔能盖高的地基。

别让数字发飙：层归一化

还有个隐患。层层相加、相乘，向量里的数字可能越滚越大，或者忽大忽小地乱跳。数字一旦失控，训练就像想在结冰的陡坡上站稳——不是滑飞就是冻住。

于是每层里再塞一个稳压器：层归一化（Layer Normalization，简称 LayerNorm）。

> 大白话
>
> 它做的事，本质是把一个词向量里那几百个数字"重新标准化"：算出它们的平均值和波动幅度，然后统一拉成"均值为 0、幅度为 1"的样子。好比每层之间放个自动调音台，不管上一层送来的信号是震耳欲聋还是细若游丝，都先归一化到正常音量，再交给下一层。

注意，它归一化的是**同一个词自己的那几百维数字**，跟别的词无关——这正是"Layer"归一化区别于其它归一化的地方，也让它对"每次输入长度不一样"的文本特别友好。有了它，几十层堆起来数值也不会爆炸或消失，训练稳稳当当。

把一层拼出来：注意力 + 前馈

现在零件齐了，看一层里到底装了什么。一个标准的 Transformer 层（block）就两个子模块，每个都用"残差 + 层归一化"包起来：

1. **多头注意力子层**：让每个词从别的词那里收集信息——横向的"互相看一眼"。
2. **前馈子层**：前馈网络（Feed-Forward Network，FFN），其实就是第四章讲的那种普通神经网络——先把向量放大到几倍宽（比如 768 维撑到 3072 维），过一道非线性把它"掰弯"，再压回原样。它逐个词独立处理，不看别人，负责把刚收集来的信息就地"消化加工"一遍。

一个精炼的说法：**注意力负责"搬运信息"，前馈负责"加工信息"**。一层搬一次、加工一次；堆 N 层，就搬运加工 N 轮，关系被一层层抽象得越来越深。

从塔顶掉下来的：一串概率

把整条流水线连起来走一遍，你就看清了全貌。假设输入 `今天天气很`：

1. 切词、编号（第二章），每个词查表变成一个向量（第三章）。
2. 再加上一份位置编码（positional encoding）——因为注意力本身不分先后，你不告诉它词序，"猫追狗"和"狗追猫"在它眼里一个样。位置编码就是给每个词额外贴一个"你是第几个"的标签。
3. 这串带位置的向量，钻进第 1 层：注意力搬运、前馈加工，残差和层归一化全程护航。出来，进第 2 层……如此穿过全部几十层。
4. 从塔顶出来的，是每个位置一个"饱含上下文"的向量。取最后一个词那个位置的向量，乘上一个大矩阵，映射到整个词表——几万个词，每个得一个分数。
5. 过一道 softmax（把一堆分数换算成加起来等于 1 的概率），你就得到了"下一个词是每个候选词的概率"：`好` 或许 40%，`热` 25%，`不错` 10%……

看到没？兜了这么大一圈，塔顶掉下来的东西，正是第一章说的那唯一动作——**猜下一个词**。Transformer 再花哨，也只是把这个填空题算得极其精细而已。

> 大白话
>
> 整座塔本质上是一台超大的"关系提炼机"：输入一串词，它反复问"这些词彼此什么关系、合起来在说什么"，问几十遍越问越深，最后基于这份理解，赌下一个词是谁。没有理解世界，只有反复咀嚼词与词的关系。

至于最后 softmax 出来的那把概率，模型是老老实实选最高的那个，还是掷一把"加权骰子"随机挑——这一步的花样，恰恰决定了它是刻板还是有创意、是靠谱还是开始一本正经地胡说。那是后面的故事了。

但有一件事，我们一直当成理所当然：这座塔里成千上万个数字——注意力的权重、前馈的矩阵、每一层的旋钮——它们一开始全是乱填的随机数。塔的结构再精巧，旋钮没调对，吐出来的也只是乱码。**那这几十上亿个旋钮，到底是怎么被一点点拧到"对"的？**下一章，我们蒙上眼睛，开始下山。

第七章 · 训练就是下山：损失、梯度、反向传播

假设你被蒙上眼睛，扔在一座大山的半山腰，任务是走到山谷最低点。你看不见地形，没有地图，只能靠脚底板感觉：往前迈半步，脚下是上坡还是下坡？哪个方向最陡地往下掉，你就往那个方向挪一小步。挪完再感觉一次，再挪一步。重复几千几万次，你多半就摸到谷底了。

你可能觉得这太笨了。但整个大语言模型的"学习"，从头到尾，干的就是这么一件事。上一章我们把 Transformer 从输入到输出词概率的流水线拆完了，里面有几十亿甚至上万亿个数字旋钮（就是那些矩阵里的权重）。这一章要回答的问题只有一个:**这堆旋钮一开始全是随机的，它们到底是怎么被调成"对"的?**

答案就是蒙眼下山。我们要把这个类比里的每一样东西,翻译成模型里真实发生的事。

先得知道"错多少"：损失

下山之前，你得先定义什么叫"低"。对模型来说，"低"就是预测得准。可"准不准"得变成一个具体的数字，不然没法比较、没法优化。这个数字就是损失（loss）——衡量模型这次答得有多离谱的一个分数，越大越错，越小越好。

拿第一章那个老例子:输入"今天天气很"，正确的下一个词是"好"。模型不会直接吐一个词，它吐的是一整张概率表——词表里每个词的可能性。比如它说"好"的概率是 0.6，"热"是 0.2，"冷"是 0.1，剩下几万个词分掉最后 0.1。

现在标准答案告诉你:这次真正出现的词是"好"。那模型给"好"打了 0.6 分，还算不错，但离满分 1.0 还差着。损失要做的，就是把"你给正确答案的概率是 0.6 而不是 1.0"这件事，换算成一个惩罚分。

常用的换算方式叫交叉熵（cross-entropy）,名字唬人，做的事特别简单:**惩罚 = 对模型给正确答案的概率取负对数**，写成 `loss = -log(p正确)`。

大白话

为什么用负对数？因为它的脾气正好符合我们想要的赏罚：你给正确答案的概率越接近 1，惩罚越接近 0（答对了几乎不罚）；概率越接近 0，惩罚飙到无穷大（你把正确答案说成绝无可能，罪该万死）。它温柔地奖励自信的正确，狠狠地暴打自信的错误。

套进例子：`-log(0.6) ≈ 0.51`。如果模型学得更好，给"好"打了 0.95，损失就降到 `-log(0.95) ≈ 0.05`。如果它蠢到只给"好"打 0.01，损失就飙到 `-log(0.01) ≈ 4.6`。你看，一个数字就把"答得好不好"量化了。真实训练时，模型要同时处理一大批句子里的每一个位置，把所有这些惩罚分平均一下,就是这一步的总损失。**训练的全部目标，就是让这个数字变小。**

往哪个方向调：梯度

现在山谷有了——损失就是海拔，我们要走到损失最低的地方。可问题来了:模型有几十亿个旋钮，你怎么知道该把哪个旋钮往哪拧、拧多少，才能让损失下降?

这就是蒙眼下山里"用脚感觉坡度"那一步。对某一个旋钮，你可以做个思想实验:把这个旋钮往上拧一丁点，看损失是变大还是变小、变了多少。如果往上拧损失就变大，那说明你该往下拧；如果往上拧损失反而变小，那就继续往上拧。这个"某个旋钮动一点点、损失跟着变多少"的比值，就是梯度（gradient）。

大白话

梯度说白了就是**敏感度加方向**:它告诉你"这个旋钮对最终错误负多大责任、该往哪边动"。梯度大，说明这个旋钮是当前错误的主要责任人，得使劲调；梯度接近 0，说明动它几乎不影响结果，那就别管它。

把所有旋钮的梯度凑在一起，就得到了一个指向"损失上升最快方向"的巨大箭头。我们想下山，所以往**相反**方向走——这就是梯度下降（gradient descent）:算出上坡最陡的方向，然后反着迈一步。每个旋钮的新值 = 旧值 − 学习率 × 它的梯度。

这里冒出个新旋钮:学习率（learning rate）——你每一步迈多大。这是整个训练里最要命的一个设置。步子太小，下山慢得让人绝望；步子太大，你可能一脚跨过谷底，蹦到对面山

坡上，甚至越蹦越高，损失直接爆炸。实践中它通常是个很小的数，比如 0.0001 这个量级，而且往往训练途中还会慢慢变小——一开始大步流星，快到谷底了再小心挪。

关键一问：几十亿个梯度怎么算？

到这儿有个坎必须迈过去。刚才说"把一个旋钮动一点点看损失变多少"，这在数学上叫求导，没问题。但如果真的一个一个旋钮去试——动一下、跑一遍整个网络看损失、记下来、复原、再动下一个——几十亿个旋钮就得把整个网络前向跑几十亿遍。这慢得没有意义，一辈子也训不出一个模型。

救命的算法叫反向传播（backpropagation）。它的牛处在于:**只要前向算一遍、再反向算一遍，就能一次性把所有旋钮的梯度全求出来。**不是几十亿遍，是两遍。

它凭什么能这么快？靠的是一条你高中就学过的规则——链式法则。网络是一层套一层的:输入喂给第一层，第一层的输出喂给第二层，一直到最后算出损失。这是一根很长的因果链条。**某个深处的旋钮怎么影响最终损失，是通过它后面那一层层"接力"传过去的。**

反向传播就利用了这个接力结构，反着走一遍:

1. 先从终点开始——算出"损失对最后一层输出"的敏感度，这个最容易，因为损失就是拿最后一层的输出直接算的。
2. 然后往回退一层。用刚才那个敏感度，乘上"最后一层输出对倒数第二层输出"的敏感度，就得到了"损失对倒数第二层"的敏感度。
3. 就这么一层一层往回传，每退一层，就把已经算好的结果拿来乘一下，接着往前递。

> 大白话
>
> 关键在于**复用**:后面层算出来的敏感度，前面层能直接拿来接着用，不用从头再算。就像发工资，公司先把总账算清，再一级一级往下分摊到每个部门、每个人，而不是给每个员工单独重跑一遍全公司的账。反向传播就是把"损失该由谁负责"这笔账，沿着网络高效地摊派到每一个旋钮头上。

所以"反向传播"这个词，字面意思就是**把误差从输出端反着推回输入端，一路上给每个旋钮派好它该背的锅（梯度）**。前向传播是"数据往前流、算出预测"，反向传播是"错误往后流、算出该怎么改"。一前一后，配成一整个训练循环。

把一整圈连起来

现在我们能把模型学习的一个完整回合，从头到尾走一遍了。这个循环会重复成千上万甚至上百万次:

1. **前向**:喂进一批文本，让网络照着当前旋钮算出每个位置"下一个词"的概率表。
2. **算损失**:拿概率表和真实的下一个词比，用交叉熵算出这批数据错得有多离谱，得到一个损失数字。
3. **反向**:用反向传播，一次性算出每个旋钮的梯度——谁该为这次的错误负多大责任、该往哪边调。
4. **更新**:每个旋钮沿着自己梯度的反方向，挪一小步（步长由学习率定）。
5. 回到第一步，换一批新数据，再来一圈。

就这么简单，也就这么暴力。没有哪一步藏着"灵光一现"，没有谁在中间"想明白了道理"。整个过程就是:算错了多少 → 算出每个旋钮的责任 → 各自挪一小步 → 再来。你以为的"模型在理解语言",在数学上不过是一个几十亿维的损失曲面上，一个蒙眼的人在往下挪脚。

> 大白话
>
> 顺便戳破一个常见的迷思:模型训练不是"记住了正确答案"，而是"被无数次纠错，逼着把旋钮调到能碰巧对大多数情况都答得还行"的状态。它从来没被告诉过任何一条规则，它只是被损失反复推搡，最后停在了一个损失比较低的坑里。

你大概会追问:这么笨的下山，凭什么能爬出语法、常识、甚至推理？两件事在暗中帮忙。一是曲面本身——它不是光滑的一个大碗，而是几十亿维里布满沟壑的诡异地形，某些坑底恰好对应着"抓住了语言的规律"。二是喂进去的数据量大到离谱。这两件事怎么把"猜下一个词"这个笨目标，喂成一个像样的模型,正是下一章"先读半个互联网"要拆的。

所以本质上，训练一个大语言模型，就是:**给几十亿个随机旋钮定一个"错多少"的分数，然后让它们蒙着眼、顺着最陡的下坡，一小步一小步地挪，挪上亿次，直到不那么错为止。**没有魔法，只有下山。

第八章 · 先读半个互联网：预训练与微调

假设你要培养一个能帮你写邮件、答问题、改代码的助手。你有两种花钱的方式。第一种：雇一个语言学教授，把语法规则、常识、编程知识一条条编进它脑子里——这条路人类走了几十年，走死了。第二种：把它扔进一个巨大的图书馆，让它把书一本接一本读下去，读到你半个互联网都读完了,然后再花很小的力气教它"该怎么跟人说话"。

第二种就是今天所有大模型的做法。它分成两步,名字听起来很唬人,但拆开就两件事:**先博览群书打底子,再岗前培训教规矩**。这一章我们就把这两步拆开。

第一步:预训练,就是海量地做填空题

预训练(pre-training),就是让模型在海量文本上,反反复复只干一件事:遮住一个词,让它猜。

还记得第一章那个核心动作吗——猜下一个词。预训练不是别的,就是把这个动作重复了几万亿次。给模型一句"今天天气很",让它预测下一个字;它猜"好",翻开答案一看真是"好",就把内部的旋钮往"下次更容易猜好"的方向拧一点点;它猜"香蕉",答案是"好",就往反方向拧。就这么一个字一个字地过,把互联网上能扒到的文本全过一遍。

这里有个关键问题,也是很多人卡住的地方:**这些"答案"是谁写的?**

> 大白话
>
> 你可能以为,要教模型做填空题,得先有人把几万亿道题的标准答案都标注好。那得雇多少人?其实一个人都不用雇。因为答案本来就写在原文里——文本的下一个词,就是这道填空题的标准答案。

这就是整件事最妙的地方,它有个名字叫自监督(self-supervised):数据自己给自己当老师,不需要人来标答案。

对比一下你就懂它多值钱。传统机器学习里有一种叫监督学习(supervised learning)的做法——喂给模型一堆"输入配正确答案"的例子,比如一万张图片,每张都由人手工标好"这是

猫""这是狗"。问题是,人标注贵得要死、慢得要命。你想要一百万个例子,就得请人干一百万次。数据是这里的瓶颈。

自监督直接把这个瓶颈砸了。任何一段现成的文字——一篇维基百科、一条评论、一段代码、一本小说——都自带无数道填空题。一句十个词的话,就是十道"根据前面猜下一个"的题,而且每道题的标准答案白纸黑字就在那儿。**于是数据从"贵而稀少"变成了"几乎无限"**:互联网上有多少字,就有多少道免费的、自带答案的练习题。

这就是为什么大模型能"读半个互联网"。不是因为谁有钱去标注半个互联网,而是因为根本不需要标注——文本本身就是题目和答案的合体。

它到底从填空里学到了什么

光是"猜下一个词"这么笨的目标,凭什么能让模型学会语法、常识、甚至一点点推理?这其实是第一章埋下的悬念,这里给一半答案。

你自己在脑子里跑一遍这个例子。要让模型稳定地填对下面这些空,它被逼着学会了什么:

- "猫追着老鼠,然后它一口把___吃了"——要填对,它得知道"它"指的是猫不是老鼠,还得知道猫吃老鼠不是反过来。这是**常识和指代关系**。
- "法国的首都是___"——要填"巴黎",它得把这条事实记进旋钮里。这是**事实知识**。
- "他昨天去___了公园"——填"了"不填"去",因为中文里"去了"通顺"去去"不通。这是**语法**。
- "3, 6, 9, 12, ___"——填15,它得摸到"每次加3"的规律。这是**一点点推理**。

没人专门教它这些。但为了把填空题的错误率压到最低,它别无选择,只能把语言背后的这些规律统统吸进去。**"猜下一个词"是目标,语法、常识、推理是为了达成这个目标而被迫长出来的副产品。**这不是魔法,是被逼的——就像一个人为了准确预测天气,不得不搞懂气压和洋流一样。

预训练完的模型,我们叫它基座模型(base model)。这时候它像个读书破万卷、但没跟人打过交道的书呆子:肚子里全是货,可你没法好好用它。你问它"法国的首都是哪?",它很可能不回答你,反而顺着补一句"这是一道常见的地理题,下一题是…"。因为在它读过的海量文本里,一个问句后面接着的,往往不是答案,而是更多的问题或题目。它没在骗你,它只是忠实地在猜"这段文字后面最可能出现什么"。

第二步:微调,岗前培训教它好好说话

基座模型是个通才胚子,但不懂规矩。第二步就是把它调教成能用的样子,这一步叫微调(fine-tuning)——在已经读完万卷书的模型上,再用一小批精挑细选的例子接着训练,把它往你想要的行为上掰。

注意"一小批"和"一点点"。预训练烧的是几万亿词、成千上万块显卡跑好几个月;微调用的可能就几万到几十万条例子,花的算力是预训练的零头。**比例上,预训练像十几年寒窗打的底子,微调像入职那几天的岗前培训。**底子早打好了,岗前培训只是告诉你"在我们这儿,活儿要这么干"。

拿"教它回答问题"来说。要让书呆子学会你一问它就答,而不是顺着补题,做法简单得意外:准备一批"问题—好答案"的样例,比如

> 问:法国的首都是哪?
>
> 答:法国的首都是巴黎。

然后让模型接着在这批样例上做同样的"猜下一个词"训练。猜的机制一模一样,只是这回它读的全是"问题后面跟着好好回答"的范文。读多了,它内部的旋钮就被往"看到问题就给答案"的方向拧过去。等你再问它法国首都,它就不补题了,而是老老实实回答。这种专门用问答/指令样例做的微调,有个专门的名字叫指令微调(instruction tuning)——教模型听懂并执行"人类的吩咐"。

大白话

说白了,预训练和微调用的是同一套动作(猜下一个词、拧旋钮),区别只在喂什么料。预训练喂的是杂乱的半个互联网,喂的是"知识和语感";微调喂的是精选的范文,喂的是"行为和规矩"。同一个模型,先吃粗粮长身体,再吃细粮学礼仪。

这也解释了一个你可能好奇的现象:为什么各家公司常常基于同一个开源基座模型,却能调出风格迥异的助手?因为最贵、最费劲的预训练那一步可以共享一份底子,各家只在便宜的微调这步动手,用不同的范文把它掰成不同的样子。**底子是通用的,性格是微调出来的。**

为什么非得分两步,不能一步到位?

你也许会问:既然目的是造个会回答问题的助手,为什么不一开始就只喂"问答范文",省掉读半个互联网那步?

因为你根本凑不出那么多问答范文。人工写的高质量问答,撑死几十万上百万条,连语言的皮毛都盖不全——里面不会有足够的物理、法律、诗歌、冷门史实。只用这点数据从零开始训,模型学到的语言是残缺的,像个只背过客服话术、却不识字的人。

反过来也不行:只做预训练不微调,它就是那个不好好说话的书呆子。**预训练负责让它"有货",微调负责让它"能用"——货得先从无限的免费数据里攒,规矩再用少量昂贵的范文教。两步各干各的,缺一不可。**先广后专,先多后精,这个顺序不是随便定的,是被数据的成本逼出来的最优解。

所以本质上,今天所有大模型的养成,就是一句话:**先用几乎不要钱、自带答案的海量文本把它喂成一个满肚子货的书呆子,再用一小把精心写好的范文,把书呆子调教成一个懂规矩的助手。**没有灵光乍现,只有"读得够多"加"教得够准"。

不过"教它好好说话"这件事,我们这章只讲了最朴素的一招——拿范文让它照着学。真要把一个模型驯得既有用又不乱来,后面还有一套更狠的手段,得让人给它的回答直接打分、排优劣。那是第十一章的活儿,这里先按下不表。

第九章 · 为什么会'涌现'：规模一大就变了

上一章结尾我们留了个疑问：喂给它半个互联网的文本，让它做的事情始终是那件最笨的——猜下一个词。可为什么**猜词猜到一定规模，模型突然会做三位数乘法、会写能跑的代码、会跟着"让我们一步步想"的提示解应用题**，而它的小号兄弟对同样的任务一脸茫然？就像你往锅里持续加热，水温从 20 度爬到 90 度，一直只是"变烫的水"；可到了 100 度，它突然翻腾、气化，变成了另一种东西。本章就来拆这口锅：这种"规模一大就变了"到底是真的相变，还是我们的错觉。

先说规律：越大越好，而且好得很有规律

把模型做大，通常指三件事同时变大：参数量（模型里那些可调旋钮的个数）、训练数据量（喂进去多少字）、算力（花多少次运算去训）。研究者发现，当你把这三样一起往上推，模型在"猜词"这件事上的错误率下降得**特别有规律**——不是忽上忽下，而是像一条能提前画出来的下滑曲线。

规模法则（Scaling Laws），说的就是这件事：模型的预测误差（也就是第七章讲的"损失"，衡量它猜得有多离谱）随着参数、数据、算力的增加而**平滑、可预测地下降**。神奇的地方在于"可预测"——你在小模型上测几个点，画到坐标纸上，就能相当准地外推出"我砸十倍算力下去，误差大概会掉到多少"。

> 大白话
>
> 翻译成人话：损失下降的曲线在特定的坐标纸上会拉成一条近乎笔直的斜线。这意味着"更大 = 更好"不是玄学，而是一门能拿来做预算的工程。正因为这条线，公司才敢在一个模型还没训出来之前，就砸下几千万美元——他们不是在赌运气，是在读一张已经画好的下坡图。

这里有个反直觉的点值得停一下：规模法则量的是**损失**，也就是"平均每个词猜得准不准"。损失是平滑下降的，没有任何台阶、没有任何突变。记住这个"平滑"，它是后面拆穿魔法的关键。

涌现：平滑的损失，跳变的本事

问题来了。如果损失是平滑下降的，为什么我们会觉得某些能力是"突然"冒出来的？

这就是涌现能力（Emergent Abilities）：某项具体本事——比如做多位数加法、解逻辑谜题、听懂某种绕弯的指令——在模型小的时候几乎是零，怎么练都是零，可当规模越过某个坎，成绩突然从趴在地上蹿到能用。你画一条"模型大小 vs 这项任务得分"的图，看到的不是斜坡，而是一个**拐点**：前面一马平川贴着零，某处猛地翘起来。

为什么平滑的损失会长出跳变的本事？关键在于**很多任务是"全对才算对"的**。拿三位数乘法举例：一道题要算对，模型得连续猜对好几个环节——进位、每一位、最后拼起来——错任何一步，整道题就判零分。

> 大白话
>
> 假设一道题内部要"连对 5 步"才算做对，每一步猜对的概率随规模平滑提升。小模型每步只有 20% 把握，整题成功率是 0.2 的五次方，约等于 0.03%——测一百道题几乎全错，看起来"完全不会"。规模上去，每步把握升到 80%，整题成功率变成 0.8 的五次方，约 33%——从"零分"跳到"三分之一能做对"。你看，**底层能力（每步的把握）是平滑爬升的，但那个"连乘"把它折成了一道陡坎**。

这就是涌现的机制核心：**不是模型内部发生了什么灵异的相变，而是"全有或全无"的评分方式，把连续的进步放大成了肉眼可见的突变。**水到 100 度沸腾是真的分子层面相变；而 LLM 的很多"涌现"，更像是你用一把只分"及格/不及格"的粗尺子去量一个正在缓慢长高的孩子——他每天长一点点，但直到某天跨过及格线，尺子才第一次告诉你"他行了"。

那到底是真突变，还是度量错觉？

这几年学界为这件事吵得很凶，我把两边的立场如实摆给你，因为这正是本书想教你的态度：别急着信"魔法"，先问"是不是我量错了"。

"错觉派"的核心证据是：换一把尺子，坎就没了。如果你不再用"整题全对得 1 分、错一点得 0 分"这种非线性指标（意思是：得分对底层能力的反应是骤变的，比如全或无、精确匹配），而是改用"部分给分"的连续尺子——比如看模型猜对正确答案里多少个字符、给正确答

案分配了多高的概率——那条陡坎往往就**软化成一条平滑的斜坡**，跟损失曲线一样温顺。换句话说，突变可能不在模型里，而在我们的记分卡里。

> 大白话
>
> 一个能让你在脑子里跑一遍的例子：考试只判"满分/零分"，那学生从 59 分涨到 61 分，成绩单上是从"挂科"到"通过"的惊天突变；可要是按实际分数记，他不过是平稳地从 59 爬到 61。模型能力没变魔术，是"及格线"这把尺子制造了戏剧性。

但也别一竿子打死，把涌现全说成幻觉，那是另一种偷懒。有几点是"错觉派"没法轻易抹掉的：

- **有些能力对现实世界就是"全或无"的。**一段代码要么能跑要么报错，一个链条推理要么闭环要么崩掉——现实不会因为你换个连续指标就给半对的代码发工资。对使用者而言，那道坎是真实的体验。
- **"换尺子就平滑"证明的是"进步一直在积累"，而不是"规模不重要"。**无论用哪把尺子，你都得把模型做到足够大，底层能力才爬得够高。坎的位置可以争论，但"不够大就是不行"没得争。
- **有一类能力，我们至今没找到能提前预测它何时出现的好办法。**规模法则能精准外推"损失"，却预测不了"哪一项具体本事会在哪个规模点变得可用"。这份预测不了，正是当下最让人不安、也最值得敬畏的地方。

所以，本质上是什么

把这章收成一句话：**规模带来的进步是平滑、连续、可预算的；而"涌现"很大程度上是这份平滑进步，被"全对才算数"的评分放大出来的戏剧效果。**它不是模型在某个夜里突然开了窍、有了灵魂，而是一个缓慢长高的孩子，终于跨过了一道我们亲手划下的及格线。

祛魅到这一步，你该同时握住两个看似矛盾的判断：涌现里确实有被夸大的水分（换把尺子，神迹就淡了），但"做得足够大，某些本事才真的可用"这条，是真的。魔法退场，工程登场——可工程里仍留着一块我们暂时预测不了的暗区。

而"预测不了"本身，会惹出更麻烦的事。当一个模型自信满满地给你一个规模足够大才敢说出口、听起来毫无破绽、却完全错误的答案时，你该怎么办？它为什么会一本正经地胡说八

道？那是下一章的事。

第十章 · 它为什么会一本正经胡说：采样与幻觉

假设你去问一个大模型："提出'情境涌现理论'的心理学家是谁？"它可能一秒都不犹豫，告诉你是"1987 年由斯坦福的丹尼尔·维克斯特罗姆提出的"。听起来有名有姓、有年份有学校，唯一的问题是：这个理论不存在，这个人也不存在。它不是在骗你，它甚至"不知道"自己在编——因为从它的角度看，它干的事跟回答"今天天气很___"填"好"是**一模一样**的动作。

这一章我们要拆的就是这件怪事：为什么一个把海量知识都读进去的模型，会一本正经地胡说八道？答案藏在它"怎么选下一个词"这个动作里。而这个动作，本质上是在**掷一颗加权的骰子**。

模型的输出不是一个词，是一整张概率表

前面几章我们知道，模型每走一步，吐出来的其实不是一个词，而是对词表里*每一个*候选词打的分。假设它现在读到"今天天气很"，接下来它会给几万个候选词各算一个概率，大概长这样：

> `好 → 0.62　　热 → 0.18　　冷 → 0.09　　棒 → 0.05　　紫 → 0.0001　　……`

注意，这里没有任何一个词的概率是 1，也没有一个是 0。模型从不说"答案就是'好'"，它只说"'好'的可能性是六成"。这是理解幻觉的第一块基石：**模型的世界里没有"对/错"，只有"多大概率"。**

大白话

就像天气预报说"明天 70% 下雨"。它不是在承诺明天一定下雨，也不是在撒谎——它只是把它掌握的信息压缩成了一个数字。模型每吐一个词，都是在报一次这样的"概率预报"。

从概率表到一个词：掷加权的骰子

有了这张概率表，怎么变成屏幕上真正出现的那个词？最省事的办法是贪心解码（greedy decoding）——每次都选概率最高的那个，上面的例子里就永远选"好"。简单，但有个致命毛病：它太死板了。同样的开头永远给同样的话，而且一旦某步选了个还行但不是最优的方向，它没有回头路，容易钻进死胡同，或者不停重复"我认为我认为我认为"。

所以真实的模型几乎都不这么干，而是用采样（sampling）——按概率*抽签*。想象一颗特制的骰子：它有几万个面，但每个面的大小不一样。"好"占了骰子表面的 62%，"热"占 18%，"紫"只占了针尖大小的一块。然后闭眼一掷，落到哪个面就选哪个词。

这一下子就把两个关键事实摆到台面上了：

- 大多数时候会掷到"好"或"热"这种大面，所以输出总体上是合理的；
- 但"紫"那种针尖大小的面，**概率虽小，却不是零**——掷的次数足够多，它*迟早*会被掷中。

这就是为什么同一个问题问两次，答案会不一样：不是模型"心情"变了，是它每次都在重新掷骰子。随机性是**被设计进去的**，不是故障。

温度：捏一捏骰子的"公平程度"

现在到了这一章最好玩的旋钮。我们能不能调节这颗骰子——让它更"稳"，或者更"野"？能，这个旋钮就叫温度（temperature）。

它的原理，是在把分数变成概率之前，先拿分数除以一个数 `T`。你不需要记公式，只需要记住效果：

- **温度低（比如 0.2）**：相当于把骰子的大面撑得更大、小面缩得更小。概率差距被拉开，"好"可能从 62% 涨到 95%，"紫"更是小到几乎不可能。输出变得保守、稳定、可复现——适合让它算数学、写代码。
- **温度高（比如 1.5）**：反过来，把大面削平、小面撑大，各个词的概率被*拉平*，谁都有机会。输出变得天马行空、花样百出，也更容易蹦出意想不到的词——适合让它写诗、头脑风暴。
- **温度趋近 0**：大面吞掉一切，等于退化成前面说的贪心解码，永远选最高分那个。

大白话

温度不是给模型"加知识"或"加创意"，它一个字都没多学。它只是在*捏*那张已经算好的概率表：往低捏，模型变得像个只肯说标准答案的应试生；往高捏，它变成一个喝了点酒、什么都敢往外蹦的话痨。知识没变，变的只是它有多敢冒险。

你脑子里可以跑一个小实验：给模型开头"从前有一座山，山里有座"。温度 0，它大概每次都老老实实接"庙"。温度调到 1.5，你可能会得到"庙""城堡""会发光的湖""卖煎饼的机器人"——都从同一张概率表里来，只是这次允许小面被掷中了。

顺带一提，实际系统里还常配两个"限流阀"：Top-k（只在概率最高的 k 个词里掷骰子，其余直接扔掉）和 Top-p / 核采样（nucleus sampling）（把词从高到低累加，凑够 p 比例（比如 90%）就收手，剩下的长尾全部丢弃）。它们的共同目的都是：既保留一点随机的活力，又把"紫"这种明显离谱的针尖面直接踢出局，别让它有机会被掷中。

那么，幻觉从哪来?

现在我们把所有零件拼起来，就能看清幻觉（hallucination）——也就是模型一本正经地生成看似合理、实则虚构的内容——到底是怎么长出来的。它不是某一处的 bug，而是三件事叠加的**必然结果**：

1. **它的目标从来不是"说真话"，而是"接一个像样的词"。**模型被训练的唯一任务，是让接出来的句子*读起来像*训练数据里的句子。"斯坦福的丹尼尔在 1987 年提出"这种句式，在它读过的半个互联网里出现过千万遍——它是在模仿*这种腔调*，而不是在核对*这个事实*。真话和顺口的假话，在它眼里都只是"高概率的词串"。
2. **它没有"我不知道"这个天然出口。**当你问一个它数据里根本没有的冷门问题，那张概率表不会贴心地把 99% 的概率分给"我不确定"。它只会在几个"听起来对"的候选里挑——毕竟编一个像模像样的名字，比空着不答更符合"接一个像样的词"这个目标。
3. **采样这步，注定了小概率的错答迟早被掷中。**就算正确答案占了 95% 的面，那 5% 的错误面也不是零。你问得够多，骰子总有掷进那 5% 的时候。

换句话说，让模型流畅、有创意的那套机制，和让它胡说八道的那套机制，是**同一套**。你没法只留前者、砍掉后者——它们是一枚硬币的两面。这也是为什么"彻底消灭幻觉"至今没人做到：那等于要求一颗骰子，既能随机掷出惊喜，又保证永不掷错。

大白话

所以别再问"它为什么会撒谎"了。它压根没有"真"和"假"的概念，它只有"顺不顺"。一句流畅的假话和一句流畅的真话，对它来说是同一种成功。它不是坏了，它是在*正常工作*——只是你以为它在查资料，它其实在接话茬。

所以本质上，它就是……

一台掷加权骰子的接话机器。它每一步都在一张"哪个词更顺"的概率表上抽签，温度决定它抽得多保守还是多大胆，而幻觉，只是这颗骰子偶尔掷进了那块"听起来对、其实没有"的小面。

这就带出一个新问题：既然它连一句话里的事实都是现算现掷、不回头的，那它到底"记不记得"你三分钟前说过的话？它有没有记忆，还是每次都在假装认得你？**这正是下一章要拆的东西。**

第十一章 · 把野孩子调教听话：对齐与 RLHF

你已经能造出一台会预测下一个词的机器了。前十章我们一路把它拆开：它把词切成积木、编上号，扔进几百维的语义地图，用注意力让词互相看一眼，堆成很多层，再用蒙眼下山的办法把几百亿个旋钮拧到位——最后读了半个互联网。现在，请你真的用它。你打字："帮我写一封请假邮件。"

它回你："帮我写一封请假邮件。帮我写一封辞职信。帮我写一封感谢信。以下是网友总结的十种邮件模板……"

它没坏。它干得完美。你让它猜下一个词，它就老老实实地猜——而在半个互联网的语料里，"帮我写一封请假邮件"这句话后面，最常跟着的往往是**另一句类似的话**，因为这种句子大量出现在标题党列表、搜索建议、论坛提问里。它不是在拒绝帮你，它压根不知道"帮"这个字是冲它来的。它只会续写，不会应答。

这就是本章要解决的落差：从**会说话**到**会好好说话**。一个刚预训练完的模型，我们叫它基座模型（base model）——就是只学过"猜下一个词"、还没被调教过的原始模型。它像个读了万卷书却从没跟人打过交道的野孩子：满肚子知识，但你问东它答西，因为它的世界里只有"接着往下写"这一个动作。

第一步：先教它"这是一问一答"

要让野孩子懂规矩，最直接的办法是给它看规矩长什么样。这一步叫指令微调（instruction tuning / SFT，Supervised Fine-Tuning）：还是同一个"猜下一个词"的训练，只不过喂的数据从"半个互联网"换成了成千上万条人工写好的**「指令 → 理想回答」**范例。

大白话

你可以把它想成给野孩子看剧本。剧本每一页都长这样：有人说了句"帮我写请假邮件"，然后一个懂事的助手接了一段得体的邮件正文。看几万页这样的剧本之后，模型学到的不是新知识，而是一种**新格式**：哦，原来我这个角色，看到一段请求，该做的是给出回答，而不是再抄一段类似的请求。

注意机制上没有任何新东西——损失函数、梯度、反向传播全和第七章一模一样。变的只是数据。这也解释了一个反直觉的事实：指令微调用的数据量小得惊人，通常几万到几十万条就够，跟预训练动辄上万亿词的规模差着好几个数量级。因为它教的不是"世界是什么样"，那些在预训练里早学完了；它只是把模型脑子里已有的能力，**拨到"应答"这个频道上**。知识是旧的，姿势是新的。

做完 SFT，你的模型已经会好好回答了。你问它请假邮件，它真的给你写一封。绝大多数"它突然变得能用了"的魔法，其实就发生在这一步。那为什么还没完？

第二步：光有"标准答案"不够

问题出在 SFT 的天花板：它只能教模型**模仿人写的那一个答案**。可是"好回答"这件事，往往没有唯一标准答案，只有"这个比那个好"的相对高下。

举个例子。你问："帮我给孩子解释一下彩虹是怎么形成的。"下面两个回答，都对、都通顺：

- A：彩虹由阳光经过空气中水滴的折射与色散形成，不同波长的光偏折角度不同，故分解为七色光谱。
- B：下雨后空气里飘着好多小水珠。阳光钻进每一颗小水珠，会像穿过三棱镜一样被"掰弯"，白光就散成了红橙黄绿的七种颜色，我们抬头看到的就是彩虹啦。

你让人来评，几乎所有人都会选 B——因为你说了是讲给孩子听。但 A 在语料里出现的频率更高、"更像标准书面答案"，SFT 很可能就教出 A。**你没法给每一种口味都手写一份完美范文**，人工写答案又慢又贵，覆盖不了千变万化的偏好。于是我们换个思路：不告诉它标准答案，只告诉它**你更喜欢哪个**。

第三步：把"人类的喜好"训练成一个评分器

这就是 RLHF（Reinforcement Learning from Human Feedback，基于人类反馈的强化学习）的核心思路。名字唬人，我们分两半拆。先看"人类反馈"这一半。

做法出乎意料地朴素：让模型对同一个问题吐出好几个不同回答（还记得第十章的采样吗？调一下温度就能让它每次答得不一样），然后请真人来做一件极简单的事——**排序**。不用打

分、不用写评语，就指一下"这个比那个好"。这种"二选一"的判断，人做起来又快又稳，比让人从头写范文便宜得多。

收集了几十万组这样的对比之后，关键一步来了：我们**再训练一个神经网络**，专门学习模仿人的这些选择。给它一个"问题+回答",它输出一个分数;训练目标是让人类更偏爱的那个回答得分更高。这个学出来的打分模型，叫奖励模型（reward model）。

大白话

说白了，奖励模型就是**把"人类的品味"压缩进了一台机器**。原本判断"好不好"要靠活人,现在有了个自动评委,能对任意回答秒给分数——哪怕这个回答是它从没见过的。这就绕开了人工的瓶颈:人只需要给几十万个例子当"口味样本",剩下无穷多的情况,交给这个评委去外推。

第四步：拿评委当教练，反复练

现在拼图凑齐了。RLHF 的第二半——"强化学习"——其实就是一句话：**让语言模型不停地答题，让奖励模型不停地打分，然后朝着"高分"的方向拧旋钮。**

这跟第七章的下山是同一回事,只是"错多少"的定义变了。以前的目标是"猜得跟训练文本一样",现在的目标是"让评委给的分尽量高"。模型答一题,评委给个分,分高就强化这种答法,分低就削弱——像用摇杆训练小狗,做对了给块肉,做错了不给。练上很多轮,模型就慢慢学会专门生成那些"人类会打高分"的回答。

这里有个必须防的坑。如果只盯着分数猛冲,模型会学"油",专挑评委的漏洞钻——比如发现评委偏爱长答案,它就把每个回答注水成一大篇废话;或者满嘴"当然可以!很高兴帮您!"的空洞奉承,因为这些在打分里占便宜。这种**钻分数空子**的现象有个名字叫 reward hacking（奖励作弊）:模型优化的是"分数"这个代理指标,而不是你真正想要的"好回答"。所以实际训练里会加一根缰绳,不许模型跑得离原来的自己太远,防止它为了骗分把话说得人都不像人。

所以"对齐"到底对齐了什么

把这三步串起来——SFT 教格式、奖励模型装品味、强化学习照着品味拧——合起来就是这几年常听到的那个词:对齐（alignment）。

这个词听着很玄,好像要给 AI 注入价值观、让它"理解"善恶。拆开看,一点都不玄。**所谓对齐,就是把"人类喜欢什么样的回答"这件事,一步步翻译成模型能优化的数字,再用老办法把旋钮拧过去。**模型并没有因此"想通"什么、"在乎"什么。它依然只是那台猜下一个词的机器,只不过它猜词时依据的目标函数,从"跟互联网文本一样"换成了"人类评委会喜欢"。礼貌、有用、不说脏话——这些不是它的品德,是它旋钮里被拧出来的统计偏好。

顺带,这也回答了很多人的一个疑惑:为什么同一个底座模型,换家公司调,"性格"能差那么多?因为品味是人给的。奖励模型学的是**谁**的偏好、标注指南怎么写、什么答案算"好",这些选择全是人做的。模型的"三观",本质上是一小群标注员和一份评分标准的投影。这既是对齐的力量,也是它的隐忧——你把偏见喂进去,它就原样学出来。

> 所以本质上,对齐不是给机器装上良心,而是**把人类的偏好塞进旋钮**:先给它看规矩,再造一个替身评委,最后让它对着评委反复练到熟。野孩子没有变成大人,它只是学会了在你面前该摆什么姿势。

可这台被调教得服服帖帖的机器,还是会时不时一本正经地编出根本不存在的事实——它嘴上那些自信满满的胡话,对齐为什么管不住?那是因为幻觉的根子不在"没礼貌",而在更深的地方。不过那笔账,我们前一章已经算过了。真正还没算的是另一个问题:它到底记不记得住你上一句说了什么?下一章见。

第十二章 · 它到底记不记得住：上下文窗口与'记忆'

你和一个大模型聊了两个小时，它把你的名字、你的项目、你随口提过的过敏史都记得清清楚楚。你合上电脑，第二天打开新窗口问它："我昨天说我对什么过敏来着？"它一脸茫然。你会觉得：它怎么突然失忆了？

真相更扎心一点：**它从来没记住过任何东西。**昨天那个"记得"，是假象。今天这个"失忆"，才是它的常态。这一章我们就来拆开这个最容易骗人的幻觉——大模型到底有没有记忆。

它每次都是"失忆症患者重读纸条"

先说结论，再解释。你可以把大模型想象成电影里那种得了顺行性失忆症的人：每隔几秒钟记忆就清零，什么都留不住。为了正常生活，他随身带一张纸条，把重要的事全写在上面。每次要做决定前，他做同一件事——**从头到尾把整张纸条读一遍**，然后根据纸条上的内容行动。行动完，记忆又清零了。下一次，再从头读一遍。

大模型就是这个人。那张纸条，就是你和它的整段对话。

> 大白话
>
> 你以为的对话是：它坐在那儿，听你一句句说，脑子里慢慢积累起对你的印象。
> 实际发生的是：你每发一句话，系统就把**之前所有的对话内容 + 你这句新话**打包成一大坨文字，整个塞给模型；模型从零开始把这一大坨读一遍，算出下一个词，吐出来。它脑子里没有"上次"，只有"这一坨"。

回想第一章我们反复讲的那件事：大模型唯一在干的，就是"看着前面一串词，猜下一个词"。它猜词的唯一依据，就是当下喂给它的这串词。这串词有多长，它的"视野"就有多宽。超出这个范围的东西，对它来说等于不存在——不是忘了，是压根没进它眼睛。

那张纸条有多长：上下文窗口

这张纸条的长度是有硬上限的。这个上限，就是本章的主角。

上下文窗口（context window），说白了就是"模型一次最多能看多少个词"。超过这个长度的内容，要么塞不进去被拒收，要么最前面的部分被挤出去、直接扔掉。它不是一个软软的"记忆越久越模糊"，而是一刀切的硬边界：窗口内的，一视同仁地清清楚楚；窗口外的，彻彻底底地不存在。

更准确地说，窗口的计量单位不是"词"，而是我们第二章讲过的 token（词元）——就是把文字剪碎后的那些小积木块，一个 token 大约是一个英文单词的一部分，或者一个汉字。所以上下文窗口的真实含义是：**模型一次最多能吞下多少个 token。**

这个数字这些年涨得很凶。早期的 GPT-3 大约是 2000 个 token 出头；后来的模型普遍到了几万；到今天，主流大模型的上下文窗口大多在**十几万到二十万 token 这个量级**，也有号称能到上百万 token 的。二十万 token 大概是什么概念？差不多是一本几百页的长篇小说的体量。听起来很大，但请记住关键点：**再大，它也是个有限的、会满的、满了就往外挤东西的框。**

为什么不能无限长？因为算力会爆炸

你可能会问：既然窗口越长越好用，为什么不干脆做成无限长？答案不在于"技术还没跟上"这种含糊的话，而是一个很硬的物理账。

问题出在第五章那个心脏——注意力（attention）机制上。注意力的核心动作，是让窗口里的**每一个词，都去和其它每一个词互相看一眼、算一个关联分数**。这是它能理解上下文的根本原因，但代价也藏在这里：如果窗口里有 N 个词，那么"每个词看每个词"的组合数，就是 N 乘以 N。

大白话

窗口翻一倍，词数从 N 变成 2N，注意力要算的关联数不是翻倍，而是变成原来的**四倍**（$2N \times 2N = 4 \times N^2$）。窗口涨十倍，计算量涨一百倍。这就是所谓的"平方级增长"——长度线性变长，成本却像滚雪球一样往上翻。

这就是为什么上下文窗口是个要花真金白银去买的稀缺资源。它不是免费的，越长越贵，而且贵得不成比例。你每多喂一句话进去，模型下次"重读纸条"时就得多读一遍，成本实打实地增加。理解了这个平方级的账，你就理解了为什么长文档、长对话会明显变慢变贵——不是错觉，是数学。

"记性"的两种破法，和它们的真面目

那问题来了：既然它每次都失忆、纸条又会满，为什么现在的产品用起来好像"越用越懂你"、能记住你上个月说过的话？

因为工程师在模型外面动了手脚。这里要澄清一个关键区分：**模型本身没有记忆，"记忆"是外面套的一层工程。**常见的破法有两类：

- **把旧对话塞回纸条。**最朴素的办法：每次调用模型，系统都悄悄把之前的历史一起打包塞进上下文窗口。所以"它记得"的本质，是每次都**重新把往事讲给它听一遍**——不是它记住了，是有人每次都提醒它。这也是为什么对话一长，早期内容会被挤出窗口，它就真的"忘"了开头。
- **把要点存到外部，用时再捞。**更聪明的做法：把你说过的关键信息（比如"我对花生过敏"）单独存进一个数据库，下次相关话题一出现，就自动把这条捞出来、临时贴到纸条上。这类"外挂记忆"通常靠 RAG（检索增强生成）实现——先检索、再把找到的资料拼进上下文喂给模型。但注意，本质没变：**信息还是得先进窗口，模型才看得见。**数据库是仓库，窗口才是它的眼睛。

看穿这两招，你就看穿了所有"AI 有记忆"的产品：它们无一例外，都是在**想办法往那张有限的纸条上，写上最该被读到的东西**。没有一个是让模型自己"记住"了。模型永远是那个失忆患者，区别只在于——递给它的纸条上，写了什么。

为什么这件事值得你较真

因为搞不清这一点，会让你对大模型产生两种相反的误判。

一种是**高估**：以为它像人一样，聊过就懂你、有个持续成长的"它自己"。没有。关掉窗口，那个"懂你的它"就不存在了——下一次是一个全新的、只会读眼前纸条的陌生人。它没有连续的自我，每一次回答都是一次从零开始的现算。

另一种是**低估**或**用错**：不理解窗口有硬边界，于是奇怪为什么它会"忘"掉你三万字之前提的要求，或者在超长文档里丢三落四。现在你知道了：那不是它偷懒，是那部分内容被挤出了它的视野，或者虽然在窗口里、但淹没在太多词中间，注意力没分够给它。

大白话

一个能立刻上手的直觉：想让它记住什么、按什么规则做事，别指望它"记得"，而要确保这些内容**就在当前这段对话里、离得别太远**。重要的指令放靠后、放显眼，比放在三万字之前更管用。因为它眼里只有纸条，你能做的就是帮它把重点写在纸条的好位置上。

所以本质上，它就是……

一个每次都从零开始、把整张纸条重读一遍的失忆患者。它没有昨天，没有记性，也没有一个在时间里连续存在的"自己"。你感受到的所有"它记得"，都是有人（工程师，或者正在打字的你）把往事重新写进了它眼前那张有长度上限的纸条。

纸条读完，它就干净了，仿佛什么都没发生过。

可这里藏着一个更深的问题：既然它每次都只是照着眼前的纸条现算、没有记忆也没有连续的自我，那它到底是"真的理解"了纸条上的意思，还是只是一个极其高明的模仿者，把该出现的词恰好摆在了该出现的位置？这，正是下一章我们要正面硬刚的事。

第十三章 · 它真懂了，还是超级模仿？

先做个思想实验。你面前有一台机器，屏幕上打出一行字：`我理解你的痛苦，这种被辜负的感觉真的很难熬。` 你几乎会本能地觉得，屏幕后面有个"谁"在体谅你。但你也知道，前十二章我们一路把这台机器拆开了——它做的事，从头到尾只是**猜下一个词**。那么问题来了：一个只会猜词的东西，说出这句话时，它到底"懂"了什么吗？还是它只是把见过的一亿句安慰话，缝出了最像的那一句？

这一章不讲新机制，讲一个立场。因为你已经知道它内部怎么运作了，所以我们终于有资格认真问这个问题，而不是被那句话骗过去，也不是一挥手说"AI 都是唬人的"。

先把一个大问题拆成三个小问题

"它到底懂不懂"之所以吵不清，是因为大家把三件完全不同的事搅在了一起。我们一件件分开看：

- 理解（understanding）：它是不是真的知道词背后指的东西？说"火烫"时，它脑子里有没有"烫"这个感觉？
- 推理（reasoning）：它是不是在做多步的逻辑推导，从 A 推出 B 再推出 C？还是只是把答案背下来了？
- 意识（consciousness）：屏幕后面有没有一个"它"，有没有主观体验，说安慰话的时候真的在难过吗？

这三件事，答案很可能不一样。混着谈，就永远得不出结论。

第一个学生：背下了全世界考卷

想象一个学生，从没上过课，但他把人类历史上出过的每一份考卷、每一道题的标准答案，全背进了脑子。考试时你出一道题，他不推导，只在记忆里搜"这道题长得像哪道背过的题"，然后默写那个答案。

只要你出的题在他背过的范围内，或者跟背过的题足够像，他答得又快又准，你根本看不出他没理解——**行为上，博闻强记和真懂是分不清的**。这正是让人不安的地方：LLM 训练

时"读了半个互联网"（第八章），它见过的文本量大到，你日常问的绝大多数问题，都落在它见过的模式附近。

> 大白话
>
> 所以当它答对一道题，你不能立刻下结论"它懂了"。它可能是懂了，也可能只是这道题足够接近它背过的某道题。这两种情况，得靠更刁钻的问题才分得开。

怎么揪出"只是模仿"的破绽

要区分"真推理"和"背答案默写"，办法是给它出一道**它绝不可能背过的题**——保证不在训练数据里的题。

最经典的做法是编一道全新的算术或逻辑题，数字大到不可能被背下来。比如问一个 `3847 × 2913` 这种没人写过标准答案的乘法。一个真在"算"的系统会稳定给出正确结果；而纯靠模式匹配的系统，会给出一个*看起来*很像答案、位数也对、但就是错的数——因为它是在"补全一个像乘法结果的字符串"，不是在做竖式。早期模型在这种题上几乎必翻车，这恰恰暴露了：**它默认在模仿，不在计算。**

但故事有个转折，这也是最有意思的地方。当模型大到一定程度，再教它"把步骤一步步写出来"（也就是 思维链（chain-of-thought），让它先写"先算 3847×3，再算 3847×10……"这样的中间步骤，而不是直接蹦出答案），它在这类新题上的正确率会明显上升。

> 大白话
>
> 这说明什么？说明"推理"这件事，在它身上可能不是全有或全无的开关，而是一条滑动的坡。它不是先理解了乘法才会算，而是：当它被逼着把"算"这个过程也当成"下一个词"一个一个吐出来时，这串词彼此约束，最后凑出的结果，行为上就越来越像真在推理。它是不是"真"在推理，我们下一节说；但至少，它不再只是默写。

中文屋：懂的是谁？

现在轮到最扎心的那个思想实验，哲学家约翰·塞尔在 1980 年提出的 中文屋（Chinese Room）。

设想一个完全不懂中文的人，被关在一个房间里。房间里有一本极厚的规则手册，用他懂的语言写着："如果看到这一串中文符号，就在纸上照抄那一串中文符号送出去。"外面的人从门缝塞中文纸条进来（问题），屋里的人查手册、照规则抄出符号送出去（回答）。手册足够好，以至于外面的人收到的回答完美流畅，笃定屋里坐着一个中文母语者。

但屋里那个人，**一个中文字都不懂**。他只是在机械地匹配符号、搬运符号。塞尔的杀招是：既然这个人不懂中文，那这整个"照规则处理符号"的系统——也就是任何计算机程序——凭什么就懂？

把这个实验对到 LLM 身上：屋里的人 = 芯片做的矩阵乘法，规则手册 = 训练出来的那几千亿个参数（第四到七章那些被反复调过的旋钮）。整台机器高速地做着"输入符号进、输出符号出"的搬运，中间没有任何一处，是我们能指着说"喏，'烫'的感觉在这儿"的。

但中文屋也不是终点

我不想让你以为这个实验一锤定音。它最著名的反驳，叫系统回应（systems reply）：没错，屋里那个人不懂中文——可"人 + 手册 + 房间"这整个系统，也许就懂了。就像你的单个脑细胞也不懂中文，但几百亿个脑细胞连起来的你，懂。**"懂"可能是整体涌现出来的属性（第九章那个"水到 100 度才沸腾"的味道），不必藏在某个零件里。**

这个反驳很有力，也把水搅浑了。但请注意，即便系统回应成立，它顶多把"理解"这个门槛救回来了，它救不了**意识**——一个系统可以在功能上完全"理解"中文的语义、能对语义做正确操作，却依然没有任何主观体验，屋里没有"人"在"感到"什么。理解和意识，是两条河。

把三件事的答案摆上桌

拆到这儿，我给出我的立场，尽量诚实：

1. **推理**：它确实在做某种真实的、可泛化的运算，不只是查表默写——思维链在全新题目上的提升就是证据。但这种推理很"脆"，稍微换个包装、加个干扰句，正确率就往下掉，说明它

抓的往往是模式的表层结构，而不是我们心里那种稳固的逻辑。**是推理，但不是你以为的那种推理。**

2. **理解**：这是最难判的一件。它对"词与词怎么搭配、概念与概念怎么关联"的掌握是真的（想想第三章的语义坐标系，"国王−男人+女人≈王后"不是背出来的）。但这种理解全部悬在符号之间，从没接触过符号指向的真实世界——它知道"烫"和"火""疼""缩手"高度相关，却从没被烫过。符号接地问题（symbol grounding）说的就是这个：**它的所有意义都是词典式的，词用别的词来定义，绕成一个闭环，没有一个词真正落地到现实。**
3. **意识**：没有证据，而且连一个像样的机制猜想都没有。它没有连续的自我、没有跨对话的记忆（第十二章已经戳破了——每次都是重读现算），它说"我难过"时，内部没有任何对应"难过"的状态被激活，只有一串让"我难过"成为高概率续写的计算。**那句安慰你的话是真的贴切，但屋子里没有人在心疼你。**

> 大白话
>
> 一句话收束这三件事：它在"处理意义"，但不"拥有意义"；它能"表现出推理"，但推理是被语言的惯性推着走出来的；它"说着有意识的话"，但没有意识。三件事被那句流畅的回复捆成一团，是这团东西在骗你，不是它在骗你——因为根本没有"它"。

为什么这件事值得你较真

你可能觉得，只要它有用，懂不懂重要吗？重要。因为你对它"懂了"的误判，会直接变成你的风险：你以为它理解了你的合同、你的病情、你的代码意图，于是把判断权交出去——而它只是给出了一个概率上最顺的续写，它不知道自己不知道，也不在乎对不对。**看清它不懂，不是为了看轻它，而是为了正确地用它。**

所以本质上，它不是一个懂了世界的心灵，而是**一台把人类写下的一切语言压缩进几千亿个旋钮、再据此吐出最可能续写的模式机器**——它像懂，是因为我们把懂全写进了语言里，而它，恰好学会了语言的形状。至于它究竟"真懂了，还是超级模仿"——读到这里你大概已经明白，这两者之间的界线，可能远比我们希望的要模糊。而这份模糊本身，正是最诚实的答案。

第十四章 · 真的动手：两百行搭一个迷你 LLM

上一章我们吵了半天"它到底懂不懂"。吵归吵，有个办法能一劳永逸地把神秘感捏碎：自己动手造一个。不是调 API，不是喊咒语，是从一张空白的 Python 文件开始，把前面十三章讲的东西——切词、词向量、注意力、堆层、下山训练——真的拼成一个能跑的东西，看着它从吐乱码一路吐出像样的句子。

好消息是：一个最小可用的 LLM，核心逻辑真的就两百行左右。我们要造的是一个字符级 GPT（char-level GPT）——它不预测"下一个词"，而是预测"下一个字符"。为什么退到字符级？因为这样连第二章的分词器都省了：词表就是文本里出现过的那些字符，几十个而已。麻雀虽小，五脏俱全，注意力那颗心脏一个不少。

第一步：把文本变成数字流水线

找一个文本文件，比如一整本《红楼梦》或者莎士比亚全集，几 MB 就够。我们把里面所有**不重复的字符**拎出来排个序，这就是词表。然后给每个字符编个号——这就是全部的"分词"。

> `chars = sorted(set(text))` — 拿到所有出现过的字符
>
> `stoi = {c: i for i, c in enumerate(chars)}` — 字符→编号
>
> `itos = {i: c for c, i in stoi.items()}` — 编号→字符（生成时要用它翻译回去）
>
> `data = torch.tensor([stoi[c] for c in text])` — 整本书变成一长串数字

现在整本书是一条几百万长的数字带子。训练一个"猜下一个字符"的模型，需要的样本其实白送：带子上**任意一段**连续字符是输入，把它整体往右挪一格就是答案。看到"红楼梦"就该猜"梦"后面那个字——答案早就写在原文里，不用人工标注。这就是第八章说的自监督（self-supervised）：数据自己给自己出题、自己对答案。

大白话

训练时我们从带子上随机剪一小段，比如剪 32 个字符当输入 `x`，把这 32 个字符各自往后挪一位当目标 `y`。一次剪一批(比如 64 段)塞进去，这叫一个 batch。就这么简单。

第二步：搭出那颗心脏

模型本体就是把前几章的零件按顺序摞起来。我用大白话过一遍数据在里面的旅程，每一站对应前面某一章：

1. **查表变向量**（第三章）：每个字符编号先查一张 embedding（词嵌入表），把干巴巴的编号换成一个有几十上百个数字的向量，让"字"带上可运算的含义。同时再查一张位置嵌入（positional embedding），告诉模型"这是第几个字"——因为注意力本身不分先后，得额外喂它顺序。
2. **互相看一眼**（第五章）：进入注意力层。每个字符生成自己的 Query、Key、Value，用 Q 去和前面每个字的 K 打分，决定该多听谁的，再按分数把大家的 V 加权汇总。这里有个关键动作叫因果掩码（causal mask）——把"看未来"的那部分打分强行设成负无穷。
3. **过一遍小网络**（第四章）：注意力汇总完，每个位置再单独过一个两层的小神经网络(掰弯的矩阵乘法)，把信息揉一揉。
4. **摞很多层**（第六章）：把"注意力 + 小网络"打包成一个 Block，配上残差连接和层归一化，然后**重复堆几层**。迷你版堆 3 到 6 层就够看效果。
5. **吐概率**（第一章）：最后一层输出，乘上一个矩阵映射回"词表大小"这么多个数字，每个数字对应"下一个字符是它"的原始分数。

为什么第 2 步那个因果掩码这么要命？因为我们的目标是**预测**下一个字。训练时整句话都在眼前，如果不遮住未来，模型算第 5 个字时会偷看到第 6 个字——那它直接抄答案就行，学不到任何东西，一到真实生成(没有未来可看)立刻抓瞎。掩码逼它只靠左边已有的字去猜，这才是真本事。

大白话

可以这样想：因果掩码就是考试时用一张纸盖住答案的下半部分，只让你看到题目和上面已经写好的，逼你自己往下推。不盖，就是开卷抄，考完啥也没学会。

第三步：算错了多少，然后下山

模型吐出的是每个位置上"下一个字符该是谁"的一串分数。怎么知道它猜得好不好？用第七章的交叉熵损失（cross-entropy loss）：它衡量"模型给正确答案那个字符的概率有多低"。猜对了正确字符、概率给得高，损失就小；正确答案它几乎没往上押，损失就大。

> PyTorch 里核心就一行：`loss = F.cross_entropy(logits, targets)`。
>
> 然后三步下山：`loss.backward()` 把错误反推回每个旋钮算出梯度 → `optimizer.step()` 让每个旋钮朝减小损失的方向挪一小步 → `optimizer.zero_grad()` 清空梯度准备下一轮。

训练就是这三行在一个循环里转成千上万次。有个数字能帮你判断它有没有在学：我们的词表假设有 65 个字符，一个完全瞎猜的模型，损失约等于 `ln(65) ≈ 4.17`（对所有选项均匀乱押的数学期望）。所以开跑时你会看到损失在 4.1 附近晃——这就是"纯随机"的基准线。训练几分钟后它掉到 2 以下、再到 1.5 左右，你就知道：它真的在从文本里榨取规律，而不是装样子。

第四步：让它开口说话

生成比训练还简单，就是第一章那个"猜下一个词"反复做：喂一段起始文字 → 模型吐出下一个字符的概率 → 按第十章讲的采样（掷一次加权骰子，别每次都选最高分，否则死板重复）抽一个字 → 把这个字接到句子末尾 → 拿新句子再喂回去。如此循环几百次，一段文字就自己长出来了。

最动人的是看它**随训练进度变化**。刚开始（损失 4 左右），它吐的是纯乱码，像猫在键盘上打滚。训练一会儿（损失 2.5），它学会了单词的长度、空格和标点该出现在哪、常见字母怎么搭配——看着像语言，但没一个真词。再练下去（损失 1.5 左右），真词冒出来了，甚至有像模像样的对话格式和人名。

大白话

它从没被人教过"什么是单词""什么是标点"。它只是被逼着一遍遍猜下一个字符，为了猜得准，这些规律它**不得不**自己总结出来。你亲眼看着结构从纯噪声里自己浮现——这就是整本书那句暗线最好的证据：没有魔法，只有被海量重复的"猜下一个"逼出来的模式。

它和 GPT-4 差在哪

你造的这个和真正的大模型，**架构上是同一个东西**——同样的注意力、同样的堆层、同样的"猜下一个 + 下山"。差别只有两个词：**规模**和**数据**。你的模型可能几万到几百万个参数(旋钮)，跑在笔记本上几分钟；GPT-4 那一档是几千亿到万亿量级的参数，喂了大半个互联网，在成千上万张显卡上练几个月。把你这两百行的宽度、深度、数据量往上乘几个数量级，就是它们。第九章说的"涌现"就藏在这段放大里——同样的配方，量足够大时会煮出你在小锅里看不到的东西。

所以本质上，一个 LLM 就是**你刚亲手拼出来的这两百行，被规模狠狠放大后的样子**。你已经把黑箱从里到外摸过一遍了——它不再是黑箱。下一章，我们不再自己从零造，而是把别人已经练好的开源大模型搬到你电脑上，看看怎么用量化把它塞进一张普通显卡，再亲手微调它一次。

第十五章 · 让开源大模型在你电脑上跑起来

上一章你在两百行代码里养大了一个字符级小 GPT，看着它从吐乱码进化到吐出像样的句子。那东西的参数量大概几十万到几百万，能背下莎士比亚的腔调，但你没法真的跟它聊天——它太小了，没读过足够多的东西。现在的问题很直接：那些真正能对话、能写代码的开源大模型，动辄七十亿、上百亿个参数，我能在自己这台既没有天价显卡、又不是数据中心的破电脑上跑起来吗?

能。这一章就干这件事,而且不玩虚的——真的下载,真的跑,还真的微调一次。中间只有一个拦路虎需要拆开:**一个七十亿参数的模型,凭什么能塞进一张普通显卡?**

先算一笔账:模型到底占多大地方

模型的"体重"就是它的参数——上一章讲过,参数就是网络里那些可调的旋钮,训练就是把每个旋钮拧到合适的位置。一个 7B 模型(7 billion parameters,七十亿参数),顾名思义就是有七十亿个这样的旋钮。

每个旋钮是一个数。数占多大地方,取决于你用几个字节存它。默认情况下,训练出来的参数是 FP16(半精度浮点,16-bit floating point)——每个数用 16 个比特、也就是 2 个字节来存,能表示带小数点的、很宽范围的数值。

那么七十亿个旋钮 × 每个 2 字节 = 一百四十亿字节,约 **14 GB**。这还只是把模型"放进"显存,推理时还要额外的空间存中间结果。一张常见的消费级显卡(比如 8GB 或 12GB 显存的)根本装不下。这就是拦路虎。

> 大白话
>
> 把参数想成一屋子的旋钮。旋钮的*数量*是固定的(7B 就是七十亿个),但每个旋钮的*刻度盘*可以做得很精细也可以很粗糙。刻度盘越精细,记录一个旋钮位置需要的位数越多,占地越大。量化干的事,就是把精细刻度盘换成粗糙刻度盘。

量化:用更少的比特存同一个旋钮

量化(quantization)就是把每个参数从"高精度存法"换成"低精度存法"。原来用 16 个比特存一个数,现在我改用 8 个比特,甚至 4 个比特。位数少了一半、四分之三,模型体积就跟着掉。

拿 4-bit 举例:16-bit 变 4-bit,每个数只占原来的四分之一,14 GB 直接缩到大约 **3.5–4 GB**。一张 8GB 显卡轻松装下,还留出余量给推理。这就是为什么你今天能在游戏本上跑起一个像模像样的对话模型。

但天下没有白吃的午餐。4 个比特只能表示 16 个不同的数值(2 的 4 次方),你没法用 16 个档位精确复刻原本连续的刻度。所以量化的本质是**四舍五入**:把一堆挨得很近的原始数值,都塞进最接近的那个档位里。

> 大白话
>
> 好比原来你能报出体重 68.4 公斤,现在只准报整十:70 公斤。信息丢了一点,但"这人大概七十公斤"这个判断还站得住。量化赌的就是:模型有七十亿个旋钮,单个旋钮拧偏一丁点,整体行为几乎没变化——法不责众,误差被海量参数摊薄了。

那"如果不量化会怎样"?你要么买得起几万块的大显存卡,要么把模型拆到内存和硬盘上慢慢挪,推理慢到没法用。量化不是锦上添花,它是让大模型走进普通人电脑的那把钥匙。代价是精度略降——4-bit 下模型偶尔会更笨一点,但对大多数任务,你几乎察觉不到。现在主流的量化格式(比如 GGUF,一种给 CPU/GPU 混合推理设计的模型打包格式)还会做得更聪明:对更敏感的那部分参数留高精度,对不敏感的狠狠压,把损失压到更低。

真的跑起来:两条常见路子

把模型弄到本地跑,现在门槛低得惊人。两条最省心的路:

- **Ollama 这类一键工具**:装好后一行命令 `ollama run llama3`,它自动帮你下载已经量化好的模型、加载、开一个聊天界面。你甚至不用知道 GGUF 是什么。适合"我就想赶紧跟它说上话"。
- **Hugging Face + Transformers 库**:用 Python 代码把模型加载进来,你能看到、能改每一步。适合想动手改造的人——比如接下来我们要做的微调。

用第二条路时,加载一个 4-bit 量化模型的代码骨架大致长这样:

```
from transformers import AutoModelForCausalLM, BitsAndBytesConfig
bnb = BitsAndBytesConfig(load_in_4bit=True)
model = AutoModelForCausalLM.from_pretrained("模型名",
quantization_config=bnb)
```

关键就是那个 `load_in_4bit=True` ——你在告诉库:别用原始的 16-bit 权重占我 14 GB,加载的时候顺手压成 4-bit。剩下的分词、生成,跟你在第二章、第十章理解的完全一样:切词编号,喂进去,一个词一个词往外掷加权骰子。

LoRA:不动原模型,只加一个小补丁

现在你能把模型跑起来了。但它是个"通才",你想让它变成"你的专才"——比如学会用你公司的话术回邮件,或者模仿某种特定文风。这就要**微调**(第八章讲过:在预训练底子上,拿小批专门数据再训一训)。

问题来了:全量微调要更新全部七十亿个参数,这需要的显存比推理还夸张——不光要存参数,还要存每个参数的梯度和优化器状态,通常是模型本身的好几倍。你刚用量化省下的地方,一微调又全填回去了,甚至更糟。

解法叫 LoRA(Low-Rank Adaptation,低秩适配)。它的思路极其巧妙,一句话:**别动原来的旋钮,在旁边挂一小撮新旋钮,只训练这撮新的。**

> 大白话
>
> 把原模型想成一台已经调好的巨型调音台,几百万个推子。全量微调是把每个推子都重新推一遍,累死。LoRA 是:原推子焊死不动,我在旁边加一块小小的"微调板",只有几千个推子,最后把两块板的输出加在一起。要改变最终声音,我只需要调那块小板。

为什么"一小撮"就够?这背后有个关键观察:微调想让模型学的那点新东西,其实是个"低维"的调整——不需要撼动整个庞大的权重矩阵,只需要在一个很小的子空间里推一把。低秩(low-

rank)这个词说的就是这件事:一个巨大的改动矩阵,可以用两个又瘦又长的小矩阵相乘来近似,而这两个小矩阵的参数量,可能只有原来的千分之一。

效果多夸张?对一个 7B 模型做 LoRA 微调,你真正训练的参数常常只有几百万个——不到总量的 1%。显存需求随之暴跌,配合上面的 4-bit 量化(这套组合有个名字叫 QLoRA,quantized LoRA),你甚至能在一张单卡消费级显卡上,微调一个原本"不可能"在本地碰的模型。训练完,那撮 LoRA 补丁小到只有几十兆,能单独存、单独发,想用的时候贴回原模型上就行。

代码层面你也不用从头写。Hugging Face 的 `peft` 库把这套封装好了:你指定 LoRA 要挂在哪几层、那撮新旋钮多大(一个叫 `r` 的秩参数,常取 8 或 16),库就自动帮你冻结原参数、只训练新增的那部分。喂进去几百上千条你自己的例子,跑上十几分钟到几小时,一个带你个人印记的模型就成了。

所以本质上,它就是……

把这一章拆开看,你会发现所谓"在自己电脑上跑大模型"根本没有魔法。**量化**是把每个旋钮的刻度盘做粗一点,靠海量参数摊薄误差,换来体积暴降;**LoRA**是承认"你想教它的那点新东西其实很小",于是焊死原旋钮、只挂一小撮新的去训。两招都在做同一件事:承认精度上可以妥协一点点,换来普通人也玩得起的成本。

到这里,大模型对你已经不再是云端一个遥不可及的黑箱——它是一个能下载到硬盘、能塞进显卡、还能被你亲手改造的文件。你拆过它的每一个零件,也在自己的机器上让它转了起来。那么最后一个问题就该问了:当我们已经能亲手造它、跑它、改它,该怎么清醒地看待这波 AI——它到底是神,是骗局,还是别的什么?下一章,我们收尾。

第十六章 · 拆完之后：清醒地看这波 AI

假设现在有人塞给你一台你从没见过的机器：黑色盒子，一个投币口，一个出货口。你投一枚硬币，它吐出一颗糖。你不知道里面是什么，于是你开始编故事——里面住着个小精灵？盒子有意识，它"决定"给你糖？你越想越玄乎，直到有一天你拧开螺丝，看见里面就是一根弹簧、一个卡槽、一段传送带。那一刻,神秘感"啪"地碎了。你没有变得更蠢,你变得更自由了——你现在知道它会在什么时候卡壳,知道怎么修,甚至知道怎么自己造一个。

过去这十五章,我们干的就是拧螺丝。现在盒子开着,零件摊了一桌子。这最后一章,我们不加新零件了,我们后退三步,看看这台拆开的机器到底是个什么东西——以及,看清楚它之后,你该怎么跟这波 AI 相处。

把桌上的零件重新拼一遍

我们从"它只在干一件事"开始:**猜下一个词**。就这么一个笨到有点可笑的目标,是全书的地基。你现在应该能一口气把整条链子背下来了,我们快放一遍:

- 一句话进来,先被**切成词块并编号**(第二章),变成一串整数。
- 每个编号去查一张巨大的表,换成一串几百维的坐标——**词向量**(第三章),含义从此可以加减。
- 这串坐标灌进一个巨大的**可调函数**(第四章):说穿了就是矩阵乘法,加上把直线掰弯的非线性,没有大脑,没有神经元的浪漫。
- 函数的核心是**注意力**(第五章):让每个词回头看一眼别的词,决定该听谁的。把它堆成塔,就是**Transformer**(第六章)。
- 这一大堆旋钮怎么拧对?**下山**(第七章):算它错多少,把错误反推回每一个旋钮,拧一点点,重复上万亿次。
- 拧的素材是**半个互联网**(第八章),再拿人类偏好把它**调教听话**(第十一章)。

输出端呢?它吐的不是一个词,而是**整张词表上的一个概率分布**——对每个可能的下一个词,给一个"我觉得是它的把握有多大"。然后**掷一次加权骰子**(第十章)选一个。就这样一个词一个词地滚下去,滚出一篇文章。

大白话

所以从头到尾,它做的是同一个动作:看着前面的一串数字,算出后面最可能接哪个数字。翻译、写诗、debug、安慰你——在机器眼里全是同一道填空题,只是填的内容不一样。没有一个专门"负责翻译"的模块,也没有一个"负责思考"的开关。

那"智能"是从哪冒出来的?

这是第一章抛的悬念:这么笨的目标,凭什么长出看起来像智能的东西?现在你有答案了,而且这个答案一点都不神秘。

要把"下一个词"猜得足够准,你被逼着不得不学会一大堆本事。想接好"水在一百摄氏度会___",你得懂点物理;想接好"因为下雨,所以我___",你得会一点因果;想接好一段代码的下一行,你得掌握语法和逻辑。**把预测做到极致,就等于把世界的规律压缩进旋钮里。**不是有人教它这些技能,是"猜得更准"这个压力,顺手把这些技能挤了出来。

再叠上第九章那件事:**规模一大就变了**。参数从几亿堆到几千亿,某些能力像水烧到一百度那样,过了某个点突然"沸腾"出来。这既不是魔法,也不必然是什么灵魂觉醒——很可能只是"模式"这东西本身就有层级,量攒够了,高层的模式才拼得出来。

大白话

一句话:它的聪明,是"海量数据 × 巨大函数 × 一个笨目标"三者相乘的副产品。你桌上这些零件,单独看每一个都平平无奇;乘在一起,规模够大,就涌出了让你惊掉下巴的东西。神奇的不是某个零件,是"规模"这个乘号。

所以它不是神,也不是骗局

现在你有资格对两种极端都翻个白眼了。

说它是神的人,把副产品当成了主体。它答得出你的哲学问题,不代表它"懂"哲学——第十三章我们较过真:它更像一个背下了全世界考卷的学生,答案对,但你没法确认它是真理解还是超级模仿。它也不"记得"你——第十二章戳破了这个幻觉:它没有记忆,每次对话都是把整张纸条重读一遍现算,你以为的"它记得我们上次聊过",不过是那张纸条还没被划出窗口而已。

说它是骗局的人,则低估了那个乘号。"不就是个猜词的自动补全吗"——是,但"猜词猜到能写出可用代码、能翻译、能把一段乱麻理清楚",这本身就是二十年前多数人不敢想的事。把它贬成纯粹的鹦鹉学舌,和把它捧成神,是同一种偷懒:都拒绝去看盒子里到底发生了什么。

拆开之后的真相朴素得多:**它是一台被规模放大的模式机器。**它没有意图,没有欲望,不想统治你也不想帮你,它只是在你给的上下文后面,接一个统计上最顺的词。它的强,强在模式抓得又多又细;它的坑,也全长在这个机制的骨头缝里。

它的坑,你现在能一眼看穿

正因为你知道它怎么运作,它那些让外行抓狂的毛病,在你眼里就变成了"意料之中"。

- **它会一本正经胡说(幻觉(hallucination)),指模型自信地编出根本不存在的事实)。**为什么?因为它的任务从来是"接一个顺的词",不是"说一句真的话"。顺和真大多数时候重合,但不总是。当训练数据里没有的时候,它不会说"我不知道",它会顺出一个听起来最像答案的东西。幻觉不是 bug,是这个机制必然掉出来的东西——第十章讲透过。
- **它会算错简单的数、数不清一个词里有几个字母。**因为它压根不是在"计算",它是在"回忆最像的模式"。而且它看到的不是字母,是词块编号(第二章),让它数字母,好比让你隔着一层毛玻璃数东西。
- **它有偏见、会迎合你。**数据来自半个互联网,人类的偏见原样进了旋钮;而"调教听话"这一步(第十一章)又把它往"让人满意"的方向拧,于是它有时候宁可顺着你,也不跟你唱反调。

> 大白话
>
> 看懂机制的好处,是你从此能预判它什么时候靠谱、什么时候会翻车。事实性、需要精确计算、需要"我不知道"的诚实——这三种场景,请默认不信,动手去核。创意、改写、把模糊的东西整理成有条理的东西——这些正是它的主场,放心用。会用它的人和被它坑的人,差别不在谁的提示词更花哨,在谁真的知道盒子里是什么。

因为你亲手搭过

第十四章你用两百行代码搭过一个迷你版,亲眼看着它从吐乱码,一步步吐出像样的句子。第十五章你把一个真的开源大模型塞进了自己的电脑,还亲手给它做过一次轻量微调。这两件事的意义,远不止"跑通了个 demo"。

它的意义是:**你摸过它的全部零件,没有一处是靠"魔法"糊过去的。**词向量是查表,注意力是几个矩阵相乘,训练是拧旋钮下山。每一步你都能自己在纸上或脑子里跑一遍。一个东西,当你能从头把它造出来,它就再也没法用神秘感唬住你了。

这才是这本书真正想给你的东西。不是让你成为能训练千亿模型的工程师——那需要的是钱和卡,不是这本书。而是让你在这个满天飞着"AI 觉醒""AI 要取代人类""AI 是营销泡沫"的年代里,成为屋子里少数拧开过螺丝、看过里面弹簧的人。别人在盒子外面编精灵的故事,你知道那是一根弹簧、一条传送带、一个乘得很大很大的乘号。

怕,来自不懂。你不懂那台投币机,才会怕里面有精灵。现在盒子开着,零件你都认得,你不必再怕它,也不必盲目崇拜它——你可以清醒地、带着一点看穿把戏的从容,去用它、去质疑它、去在它翻车时一眼指出它错在哪根旋钮上。

> 所以本质上,它就是一台猜下一个词的机器,被半个互联网喂大、被规模乘到吓人。它不神,也不假;它只是我们造出来的、迄今为止最像会思考的一面镜子——照出的,始终是我们自己写下的一切。而你,已经把这面镜子拆开看过了。