ALGORITHM PANORAMA

算法全景

从经典算法到大语言模型

原理 · 代码 · 实现细节 · 优缺点 · 应用场景 · 性能参数

沿时间轴梳理 85 个关键算法 · 含数据结构专章



任增浩 著

12 章 · 204 节 · 312 页 · 7 附录 · 237 张表 | 81 个算法仿真 · 6 种语言

算法全景

从经典算法到大语言模型

算法全景

从经典算法到大语言模型

任增浩　著

12 章 | 204 节 | 312 页 | 127 段可运行代码 + 7 个附录
1945 — 2026 | 配套 82 个可交互算法仿真 · 6 种语言 | 312 页

版权与使用说明

书　　名	算法全景：从经典算法到大语言模型
作　　者	任增浩
版　　本	第 2 版修订 · 2026 年 10 月
内容范围	1945—2026 年代表性算法
篇　　幅	12 章 · 204 节 · 312 页 · 7 附录 · 约 26.9 万字 · 237 张表
配　　套	算法动画仿真实验室（82 个可交互仿真 · 6 种语言参考实现）

本书为非商业用途的内部学习资料，仅供个人学习、教学与研究使用，不用于任何商业目的。

书中的代码示例用于说明算法原理，可自由用于学习与实验；如需用于其他用途，请先与作者联系。

文中所引用的第三方名称、商标与产品（如 Python、PyTorch、NumPy、Word 等）归各自权利人所有，本书仅作说明性引用，不代表任何形式的授权或背书。

书中的性能数据、时间线与行业判断来自公开资料整理，随技术演进可能发生变化，请以最新官方文档为准。

如需勘误反馈或技术交流，可通过作者的作品集站点联系。

前　言

这本书的起点是一个很实际的问题。有段时间我几乎每天都会碰到「算法」这个词——新闻里说某个模型换了新的注意力机制，招聘要求上写着「熟悉常见算法」，技术文章里满是 O(n log n)、Transformer、RLHF。这些名词散落在七十多年的技术史和十几个领域里，单独查一个能看懂，但它们彼此之间是什么关系，一直没人告诉我。

于是我开始按时间把它们重新排一遍：从 1945 年冯·诺依曼写下的归并排序，一直排到 2026 年正在发生的混合架构、稀疏注意力和极低比特量化。排完发现，很多当初看不懂的地方是缺了前一步的语境——Dijkstra 是为了修最短路，Transformer 是因为 RNN 没法并行。知道它当时要解决什么，再看做法就顺了。这本书就是这次重排的结果，共 12 章、110 节，外加 3 个附录，覆盖 80 多个算法。

有几个地方需要提前说明。

第 0 章是给完全没有基础的人准备的。它用煮鸡蛋、贴标签的盒子、一排编号的抽屉讲清变量、列表、循环、递归和复杂度，读完大约 20 分钟。如果你从没写过代码，建议从这一章开始；如果你已经写了几年程序，直接跳到第 3 章。

第 3 章讲数据结构，共十四节：数组、链表、栈、队列、哈希表、树、堆、图、并查集、Trie，再加上单调队列、AVL 平衡树和跳表这三个进阶结构。这一章写得比较慢，因为后面所有算法的快慢都能在这里找到原因。同样是查找，哈希表是常数时间、有序数组是对数时间、链表只能线性扫描，差别全部来自数据在内存里怎么摆。每一节都配了可运行的手写实现和一张选型对照表。

如果你只关心眼下最热的大模型，第 5、6、8 章可以连着读。第 8 章是全书最厚的一章，从「预测下一个 token」讲到自注意力的手写实现、三阶段训练、LoRA 微调、RLHF 与 DPO、推理加速、RAG 与 Agent。如果你想建立完整的体系，就按顺序读；第 2 章那条时间轴可以留到最后再看，它更像一张回顾用的地图。

第 10 章是全书的工具箱，复杂度速查、算力与显存估算公式、按场景选算法的清单、七个最常见的坑都在里面，需要的时候翻就行。

写这本书最大的收获是：专业不在于记住了多少名词，而在于知道每个名词在解决什么问题、代价是什么、边界在哪里。如果读完这本书，你在遇到一个陌生算法时能先问出「它要解决什么、代价是什么」这两个问题，那这本书就没白写。

书里的错误都由我负责。发现疏漏，或者有更好的讲法，欢迎反馈给我。

任增浩

2026 年 9 月

目　录

前　言 iv
目　录 v
第 0 章　预备知识：什么都不懂也能从这里开始 1
0.1　算法到底是什么：它就是一份菜谱 1
0.2　零基础也要懂的 6 个编程概念 1
① 变量：一个贴了标签的盒子 1
② 列表：一排按顺序编号的盒子 2
③ 判断：满足条件就走这条路 2
④ 循环：把同一件事重复做若干次 3
⑤ 函数：把一段步骤打包，起个名字反复用 3
⑥ 递归：函数自己调用自己 3
0.3　零基础也要懂的 4 个数学概念 4
① 求和符号 Σ：把一串东西加起来 4
② 向量与矩阵：一排数字，和一叠数字 4
③ 导数与梯度：往哪个方向走能变好 5
④ 概率：一件事发生的可能性 5
0.4　怎么读懂「复杂度」：这是全书最常出现的词 5
0.5　常见符号与缩写速查表（随时回来查） 6
0.6　代码里那些英文单词是什么意思 7
0.7　读本书代码的四条建议 7
0.8　零基础的学习顺序建议 8
0.9　把第一个程序跑起来：从装好环境到看到结果 8
0.9.1　三样东西：编辑器、运行时、终端 8
0.9.2　第一个程序：不只是打印一句话 9
0.9.3　把程序改坏，看看会发生什么 10
0.9.4　从「能跑」到「可信」：一个不太短的例子 10
0.10　程序出错怎么办：一份可以照着走的排错流程 11
0.10.1　六步排错法 11
0.10.2　打印语句：最朴素也最有效的工具 12
0.10.3　三种最费时间的错误 13
0.11　怎么把一段代码读懂：三步拆解法 13
0.11.1　第一步：找出「输入」和「输出」 13
0.11.2　第二步：找一个最小的例子，手工跑一遍 13
0.11.3　第三步：把变量和现实对上号 15
第 1 章　导言：这份文档怎么用 16
1.1　三条并行演进的谱系 16

1.2 一个贯穿全文的判断标准 17
1.3 同一个问题的六种解法：从「两数之和」说起 17
1.3.1 解法一：一个一个试（暴力双重循环） 17
1.3.2 解法二：先排队，再从两头往中间夹（排序 + 双指针） 18
1.3.3 解法三：把见过的数记在小本子上（哈希表） 18
1.3.4 还有三种写法，各自的适用场合不同 19
1.3.5 这六种解法教的四件事 20
1.4 好代码的五个维度 20
1.5 算法的三种能力：会写、会改、会选 21
1.6 本书的阅读地图：十二章分别解决什么问题 22
1.7 配套仿真怎么用：一个四步闭环 23
1.8 零基础最常见的十二个疑问 24
第 2 章 算法发展时间轴（1945—2026） 25
2.1 里程碑事件与影响 28
2.2 时间轴背后的三条规律 29
2.3 把时间轴读成一张地图：四大家族的分工 30
2.3.1 四个家族各自在回答什么问题 30
2.3.2 三个反直觉的事实 30
2.4 每个时代的瓶颈是怎么换的 31
2.4.1 五次瓶颈转移 31
2.4.2 给读者的三条读史建议 32
2.5 同一个想法被发明了三次：三个真实故事 32
2.5.1 「分而治之」被发明了三次 32
2.5.2 「用位置换时间」出现了两次高峰 32
2.5.3 「让模型自己学特征」被讲了三次 33
2.6 时间轴之外的暗线：硬件、数据与算力 33
2.7 怎么读一条算法新闻 34
第 3 章 数据结构：算法的地基 35
3.0 为什么先学数据结构 35
3.1 数组与动态数组：最基础的容器 36
3.2 链表：用绳子把盒子串起来 37
3.3 栈：只能从一头进出的容器 39
3.4 队列与双端队列：先来先服务 40
3.5 哈希表：用算出来的位置直接找到 41
3.6 树：处理一对多的关系 43
3.7 堆与优先队列：随时知道谁最大 46
3.8 图：描述关系的数据结构 47
3.9 并查集：判断「是不是一伙的」 49
3.10 Trie 字典树：按前缀找词 50
3.11 四个值得知道的进阶结构 51

3.12 怎么选：一张速查表与八个常见错误 52
3.13 平衡树：让二叉搜索树不再退化成链表 53
3.14 跳表：在链表上架起索引层 55
3.15 线段树：把「区间求和」从 O(n) 压到 O(log n) 57
3.15.1 三个基本操作 57
3.15.2 懒标记：区间修改的省事办法 57
3.16 树状数组：更省内存的前缀和 60
3.17 红黑树：工业界最常用的平衡树 61
3.18 B 树与 B+ 树：数据库和文件系统的底座 61
3.19 字符串哈希与后缀数组：给文字按一个「指纹」 62
3.20 并查集进阶：带权与可撤销 63
3.21 图的四种存法 64
3.22 LRU 缓存：哈希表加双向链表 65
3.23 位运算与位图：用一个比特装一个开关 65
3.24 概率数据结构：用少量内存答大量问题 66
3.25 持久化数据结构：每次修改都不破坏旧版本 67
3.26 选型实战：六个真实场景怎么选 68
3.27 手写一个哈希表：从数组到链地址法 68
3.27.1 第一件事：哈希函数要满足什么 69
3.27.2 第二件事：冲突了怎么办 70
3.27.3 完整实现：二十行写完的哈希表 70
3.27.4 扩容：什么时候把箱子变多 71
3.28 手写一个堆：上浮与下沉的完整推演 72
3.28.1 插入：一路向上冒泡（上浮） 72
3.28.2 取出堆顶：把最后的搬到顶上再一路向下（下沉） 73
3.29 手写并查集：三步优化把复杂度压到接近常数 74
3.29.1 第一版：用一个数组记录每个元素的父亲 74
3.29.2 第二版：按秩合并，别让树长高 75
3.29.3 第三版：路径压缩，一次查找顺手拍平 76
3.30 数据结构选型总表：21 个结构横向对比 77
3.31 洪水填充：把一整块连通区域一次染完 79
3.31.1 它模仿的是什么 79
3.31.2 为什么这样做能得到正确答案 79
3.31.3 两种实现：队列和栈 79
3.31.4 把步骤一格一格走一遍 80
3.31.5 代码 80
3.31.6 复杂度与四个常见的坑 81
第 4 章 经典算法：确定性问题的解法 83
4.1 快速排序 Quick Sort（1960，Hoare） 83
核心原理 83

实现原理与性能要点 84
4.2 归并排序 Merge Sort（1945，von Neumann） 85
4.3 二分查找与哈希表 86
4.4 Dijkstra 最短路（1956） 88
实现原理 90
4.5 A* 启发式搜索（1968） 90
4.6 动态规划 Dynamic Programming 92
实现原理 93
4.7 贪心算法 Greedy 94
4.8 布隆过滤器 Bloom Filter（1970） 94
4.9 PageRank（1997，Google） 96
4.10 排序家族全览：七种排序放在一起看 97
4.11 字符串匹配：从暴力到 KMP 99
4.12 图算法补充：最小生成树、负权最短路与全源最短路 100
4.13 六种通用算法设计范式 101
4.14 本章常见错误与练习 103
4.15 拓扑排序、最小生成树与单调栈 103
4.15.1 拓扑排序：给有依赖关系的任务排一个合法顺序 104
4.15.2 最小生成树：用最少的钱把所有的点连通 105
4.15.3 单调栈：把「找下一个更大的数」压到 O(n) 106
4.15.4 三个技巧的边界与常见错误 107
4.16 霍夫曼编码：贪心的样板工程 107
4.17 基数排序：不比较也能排序 109
4.18 最长递增子序列：一道题看三种复杂度 111
4.19 双指针与滑动窗口：把两重循环变成一重 112
4.20 前缀和与差分：两个一行代码的技巧 113
4.21 二分答案：把「求最大值」换成「判断行不行」 113
4.22 回溯与剪枝：穷举所有可能，但不傻穷举 114
4.23 分治：把大问题切成小问题 115
4.24 位运算与状态压缩：把集合当数用 115
4.25 数论基础：每个程序员都该会的三件事 116
4.26 背包问题六讲：从暴力搜索到一维数组 117
4.26.1 第一讲：暴力枚举，看清所有可能 117
4.26.2 第二讲：加个备忘，把重复计算砍掉 118
4.26.3 第三讲：翻译成递推的表格 118
4.26.4 第四讲：把它压成一维数组 119
4.26.5 第五讲：完全背包与多重背包 120
4.26.6 第六讲：背包还能解决哪些看起来不像背包的问题 120
4.27 手推一次快速排序：把分区过程画出来 121
4.27.1 一次完整的分区推演 121

4.27.2 为什么最坏是 O(n²)，怎么避免 122
4.28 图论算法选型表：什么时候用哪个 123
4.28.1 最短路：按图的特征选 123
4.28.2 连通性、遍历与拓扑 124
4.28.3 最小生成树与稀疏化 124
4.28.4 一张总览图 125
第 5 章 机器学习算法：从数据中学规律 126
5.0 先建立四个公共概念 126
5.1 线性回归 Linear Regression 126
实现原理 128
5.2 逻辑回归 Logistic Regression 128
5.3 决策树 Decision Tree（ID3 / C4.5 / CART） 130
实现原理 131
5.4 集成学习总览：Bagging 与 Boosting 132
5.5 随机森林 Random Forest（2001，Breiman）★重点 132
核心原理：两条随机性 + 一个平均 133
5.6 GBDT / XGBoost / LightGBM 135
为什么 XGBoost / LightGBM 更快更准 136
5.7 支持向量机 SVM 137
5.8 KNN、K-Means、PCA、朴素贝叶斯速览 139
5.9 评估指标：别只盯准确率 140
5.10 本章常见错误与练习 140
5.11 岭回归：给权重加一根绳子 141
5.12 AdaBoost：把一堆弱模型串成强模型 143
5.13 DBSCAN：按密度找簇，先不用说要几个簇 144
5.14 高斯混合与 EM：K-Means 的「软」版本 146
5.15 特征工程：把原始数据变成模型能吃的形状 147
5.16 交叉验证：为什么不能拿训练集当考试 148
5.17 偏差与方差：模型为什么学不好 149
5.18 正则化的几何直觉：为什么加一项就能防过拟合 150
5.19 手推一次梯度下降：用五个点算给你看 150
5.19.1 准备一个只有五个点的数据集 150
5.19.2 第一次迭代：算清楚每一个数 150
5.19.3 学习率：唯一一个必须手调的参数 151
5.19.4 从一元到多元：同一套动作 152
5.20 特征缩放与数据泄漏：新手最容易踩的两个坑 153
5.20.1 坑一：特征量纲差一百倍 153
5.20.2 坑二：数据泄漏 154
5.21 手写 KNN 与 K-Means：两个算法的完整走一遍 155
5.21.1 KNN：少数服从多数 155

5.21.2 K-Means：反复做两件事 156
5.22 手推一次线性回归：三行数据算到底 157
5.23 混淆矩阵与三个指标：用一次体检打比方 159
5.24 手推一棵决策树：信息增益一步步算 160
5.25 类别不平衡：五条可以照着做的办法 161
5.26 怎么读一篇机器学习论文 162
第 6 章 深度学习：让模型自己学表示 163
6.1 多层感知机与反向传播 163
核心原理 163
6.2 优化器：从 SGD 到 AdamW 165
6.3 正则化与归一化 166
6.4 卷积神经网络 CNN 166
6.5 RNN / LSTM / GRU 168
6.6 生成模型：VAE、GAN、扩散模型 169
6.7 本章常见错误与练习 170
6.8 激活函数：没有它，深度学习就只是一个线性回归 171
6.9 手写反向传播：两层网络的一次完整迭代 171
6.9.1 搭一个最小的网络 171
6.9.2 前向：把输入一步步变成预测 172
6.9.3 反向：从损失倒推每一层的责任 172
6.9.4 把零件拼成框架 173
6.10 手写一个卷积层：卷积到底在做什么 174
6.10.1 一次卷积的完整计算 174
6.10.2 三个关键概念：步长、填充、感受野 175
6.10.3 为什么要「深」：堆叠出来的层次 176
6.11 训练不收敛怎么办：一份可以照着查的清单 176
6.11.1 第一步：先确认「能过拟合一个小样本」 176
6.11.2 第二步：按顺序检查这六项 176
6.11.3 第三步：看曲线的形状，判断病因 177
6.12 从零写一个能训练的神经网络：60 行纯 Python 177
6.13 梯度消失与梯度爆炸：为什么深网络难训练 179
6.14 批大小、学习率、训练轮数：三个旋钮怎么拧 181
第 7 章 强化学习：在试错中学决策 183
7.1 基本框架：马尔可夫决策过程 MDP 183
7.2 两个核心函数与贝尔曼方程 184
7.3 基于价值的方法：Q-learning 与 DQN 186
Q-learning（表格型，1989） 186
为什么是 off-policy 187
DQN：把 Q 表换成神经网络（2013/2015，DeepMind） 187
DQN 家族的改进 189

7.4 基于策略的方法：REINFORCE → Actor-Critic → PPO 189
策略梯度定理与 REINFORCE 190
Actor-Critic：用价值函数当基线 191
PPO：工业界默认选择（2017，OpenAI）★ 191
7.5 探索与利用：RL 独有的难题 192
7.6 三大流派横向对比 193
7.7 强化学习的优缺点与应用 193
7.8 强化学习与监督学习：什么时候该用哪个 194
7.9 本章常见错误与练习 195
7.10 SARSA 与 PPO：把「学价值」和「学策略」各推进一步 195
7.10.1 SARSA：和 Q-learning 只差一个 max 195
7.10.2 PPO：把「不许一次走太远」写成一个 min 196
7.10.3 PPO 与 RLHF：大模型对齐为什么选它 198
7.10.4 本节常见错误与练习 198
7.11 TD(λ) 与资格迹：往前看的另一种记账方式 199
7.12 Double DQN：把「选动作」和「评价值」拆开 200
7.13 蒙特卡罗与时序差分：两条算价值的路线 202
7.14 探索策略与奖励设计：两个最容易出错的地方 202
7.15 表格版 Q-learning 手推：把 3×4 的迷宫走一遍 203
7.15.1 迷宫设定 203
7.15.2 一条完整的回合 204
7.15.3 手工推演五次更新 204
7.15.4 为什么叫「离策略」算法 205
7.16 策略梯度手推：REINFORCE 的一次完整更新 205
7.16.1 策略是什么形状的 206
7.16.2 一次采样与一次更新 206
7.16.3 两个致命问题与它们的解法 207
第 8 章 Transformer 与大语言模型（LLM）★本文重点 208
8.1 语言建模：一切从「预测下一个 token」开始 208
分词：BPE（Byte Pair Encoding） 208
8.2 Transformer 架构全景 210
① 自注意力：让每个位置直接看到全句 210
② 多头注意力：并行看多个子空间 211
③ 一个完整的 Transformer 块 212
④ 位置编码：把顺序信息塞回并行结构 214
8.3 为什么能规模化：复杂度与并行性对比 214
8.4 预训练、Scaling Law 与涌现能力 215
Scaling Law（规模定律） 215
涌现能力 Emergent Abilities 216
8.5 三阶段训练：预训练 → SFT → 偏好对齐 217

8.6 参数高效微调：LoRA 与 QLoRA 217
8.7 偏好对齐：RLHF 与 DPO 219
8.8 推理：从 logits 到文本，以及为什么慢 220
采样策略 220
推理的两个阶段与 KV Cache 221
推理加速与显存压缩 222
8.9 RAG 与 Agent：给 LLM 装外挂 223
RAG 检索增强生成 223
Agent：LLM + 工具 + 记忆 + 循环 223
8.10 LLM 的优缺点与应用地图 223
8.11 LLM 性能参数速查（工程落地必看） 224
8.12 本章常见错误与练习 224
8.13 多头注意力、KV Cache 与采样策略：三个必须亲手算一遍的细节 225
8.13.1 多头注意力：为什么要把向量切成几份 225
8.13.2 KV Cache：推理时最值钱的一块显存 226
8.13.3 采样策略：大模型每写一个字都在抽签 227
8.13.4 本节常见错误与练习 228
8.14 因果掩码：第 t 个词为什么看不见后面 229
8.15 位置编码的演进：从正弦波到 RoPE 231
8.16 手写自注意力：用三个词的句子算一遍 232
8.16.1 Query / Key / Value 的直观含义 232
8.16.2 一个三词句子的完整计算 232
8.16.3 为什么要除以根号 d 233
8.16.4 从自注意力到多头注意力 233
8.17 一层 Transformer 到底有多少参数 235
8.17.1 逐项拆解 235
8.17.2 三个立即可用的估算结论 236
8.18 提示工程的工程化：从「会问」到「稳定可用」 237
8.18.1 四条基本规则 237
8.18.2 结构化输出与校验 237
8.18.3 什么时候不该用大模型 238
第 9 章 2025—2026 前沿进展：从「拼规模」到「拼效率」 239
9.1 主线一：架构从「纯 Transformer」走向「混合」 239
① MoE 稀疏激活成为默认选项 239
② 状态空间模型与混合架构 240
③ 稀疏注意力：把 n^2 削下来 241
9.2 主线二：推理成为可调节的预算 241
① 混合思考（快 / 慢模式） 241
② 动态推理预算与提前退出 241
③ 解码加速与隐式思考 242

9.3 主线三：训练与压缩的效率革命 242
① 优化器与训练稳定性 242
② 蒸馏与多教师在线蒸馏 242
③ 量化：从 4-bit 走向 1-bit 242
9.4 2025—2026 代表性模型速览（截至 2026 年 9 月） 243
9.5 主线四：从「会聊天」到「能干活」——Agent 的产业化 244
9.6 这轮变化里，哪些是原理层面的，哪些只是工程 244
9.7 给读者的三条跟进建议 245
9.8 本章常见错误与练习 245
9.9 MoE 到底省了什么：专家混合的账本 246
9.9.1 结构：把一个大前馈层换成若干个小的 246
9.9.2 把账算清楚 246
9.9.3 两个工程难点 247
9.10 从 RAG 到 Agent：一个真实系统的搭建过程 247
9.10.1 第一步：直接问模型（失败） 247
9.10.2 第二步：加检索（RAG） 248
9.10.3 第三步：加工具调用（Agent） 249
9.10.4 从能跑到能用：还差的四件事 249
9.11 微调一个小参数模型：从数据到上线的完整流程 250
9.11.1 第一步：把业务问题翻译成「问题 → 答案」对 250
9.11.2 第二步：划分数据集，先建一个「假模型」 251
9.11.3 第三步：LoRA 微调，先跑通再调参 251
9.11.4 第四步：失败模式对照表 251
9.12 评测：怎么知道模型真的变好了 252
9.13 幻觉：它为什么会编，怎么治 253
第 10 章 选型指南与性能参数总表 254
10.1 复杂度速查表 254
10.2 关键超参数速查 255
10.3 算力与显存估算公式（背下来这四条就够） 255
10.4 按场景选算法：决策清单 256
10.5 七个最常见的坑 256
10.6 六个真实场景的选型案例 257
10.7 按数据量、算力、延迟三条线选方案 258
10.8 一份能照着走的项目落地清单 259
第 11 章 学习路线与结语 261
11.1 三阶段学习路径 261
11.2 一句话总结每个算法家族 261
11.3 最后的三条判断 261
11.4 30 天入门路线：每天 1.5 小时怎么安排 262
11.5 90 天进阶路线：从会写走到会选 263

11.6　怎么继续往下学：四类资源与三个习惯 263
11.7　结语：写给刚开始的你 264
附录 A　核心术语中英对照表 265
附录 B　常用代码模板速查 272
附录 C　82 个配套仿真与章节对照 276
后　记 281
配套资源 282
算法动画仿真实验室 282
在线访问与下载 282
勘误与反馈 283
附录 D　复杂度与选型总表 284
D.1　数据结构操作复杂度 284
D.2　排序算法全景 285
D.3　图算法全景 285
D.4　机器学习与深度学习模型 286
附录 E　练习题与解题思路 287
E.1　入门篇（基础结构与经典算法） 287
E.2　进阶篇（需要组合两个以上章节） 289
附录 F　调试与排错手册 291
F.1　按报错类型查 291
F.2　按“程序能跑但结果不对”查 291
F.3　九个最常见的具体坑 292
F.4　一份可以打印出来的排查清单 292
附录 G　常用数学公式速查 293
G.1　复杂度与增长 293
G.2　概率与统计 293
G.3　线性代数与微积分 293
G.4　工程估算公式 294
G.5　符号速查 294

第 0 章　预备知识：什么都不懂也能从这里开始

> **这一章是给零基础的读者准备的。如果你从没学过编程、没学过算法、看到 O(n log n) 就头晕——请务必先读完这一章，大约 20 分钟。读完你会发现后面 10 章的绝大部分内容都能看懂。如果你已经有基础，扫一眼 0.5 的符号表就可以直接跳到第 1 章。**

0.1　算法到底是什么：它就是一份菜谱

先别被「算法」这个词吓到。算法就是「做一件事的步骤清单」。你早上煮个鸡蛋，先烧水、水开后放蛋、计时 8 分钟、捞出过凉水——这就是一个算法。

计算机里的算法和菜谱唯一的区别是：它必须精确、有限、可执行。「放点盐」不行，「放 3 克盐」才可以；「煮到差不多」不行，「煮 8 分钟」才可以。

菜谱里的说法	算法里的说法	为什么必须这样
食材	输入（Input）	算法要有东西可以处理
做好的菜	输出（Output）	算法必须产出结果
步骤 1、2、3…	有限步骤	步骤必须能做完，不能无限循环
放 3 克盐	无歧义	每一步都只有一种理解方式
任何人照做都能做出这道菜	可重复性	同样的输入必须得到同样的输出

0.2　零基础也要懂的 6 个编程概念

本书的代码用 Python(一种很像英语的编程语言) 写成，但你不需要会写，只要能「大概看懂它在干什么」。下面 6 个概念看完就够了。

① 变量：一个贴了标签的盒子

变量就是给一个值起个名字，后面用名字就能找到它。你可以把它想象成一个贴了标签的盒子，盒子里装着数字或文字。

代码 0-1　变量：给值起名字

```
name = "小明"      # 把文字 "小明" 放进叫 name 的盒子里
age  = 18          # 把数字 18 放进叫 age 的盒子里
age  = age + 1     # 取出 age 里的值，加 1，再放回去。现在 age 是 19
```

② 列表：一排按顺序编号的盒子

列表(List，也叫数组) 就是把很多值按顺序排成一排，每个位置有一个编号，这个编号叫下标(索引)。注意一件反直觉的事：编程里的编号从 0 开始，不是从 1 开始。

代码 0-2　列表与下标（注意从 0 开始编号）

```
a = [5, 2, 9, 1]   # 一个有 4 个数字的列表
#    ↑  ↑  ↑  ↑
#    0  1  2  3   ← 这是它们的编号（下标）

a[0]      # 取第 0 号位置的值 → 5
a[2]      # 取第 2 号位置的值 → 9
len(a)    # 问一共有几个 → 4
a[1:3]    # 取第 1 号到第 2 号（不含第 3 号）→ [2, 9]
```

③ 判断：满足条件就走这条路

判断(if/else) 就是「如果……就……否则……」，和日常说话一模一样。

> **一句话原理**　判断就是让程序在岔路口选一条路走。

生活里你每天都在做判断：出门前看一眼天，下雨就带伞，不下雨就不带。要把这句话写成程序需要三样东西——一个能回答「是/否」的条件（下雨吗）、条件成立时做什么（带伞）、不成立时做什么（不带伞）。if / else 就是把这三样东西依次写下来。

程序里的条件和数学里的不等式是一回事：比较两个数的大小、判断两个值是否相等、判断一个名字是不是空，结果都只有真、假两种。所以 if 后面括号里放的，永远是一个能算出真或假的表达式。

- 第一步：写出要判断的条件，例如 age >= 18。
- 第二步：把条件成立时要做的事，缩进写在 if 下面。
- 第三步：把条件不成立时要做的事，缩进写在 else 下面。

读代码的时候先找 if 和 else 这两个词，把整段切成「成立时的一段」和「不成立时的一段」，逻辑立刻就清楚了。

> **易错点**　缩进在 Python 里不是排版，而是语法。if 下面那行必须向右缩进，否则要么报错，要么这条语句被算到判断之外去执行。

代码 0-3　判断：if / else

```
if age >= 18:
    print("成年")      # 条件成立时做这件
else:
    print("未成年")    # 条件不成立时做这件
```

④ 循环：把同一件事重复做若干次

循环有两种写法，效果一样。第一种是「对这排东西里的每一个，都做一遍」；第二种是「只要条件还成立，就继续做」。

代码 0-4　循环：for 与 while

```
# 写法一：遍历（把列表里的每个数都看一遍）
for x in [5, 2, 9, 1]:
    print(x)              # 依次打印 5、2、9、1

# 写法二：条件循环（只要还没找够 10 个就继续）
count = 0
while count < 10:
    count = count + 1     # 每转一圈加 1，直到 10 才停
```

⑤ 函数：把一段步骤打包，起个名字反复用

函数(Function) 就是把几行代码打包成一个命令，以后只要喊这个名字就能执行整段步骤。括号里是要交给它的材料（叫参数），return(返回) 是它干完活交回来的结果。

代码 0-5　函数：打包一段步骤（这里用中文名字方便理解，真实代码用英文）

```
def 做三明治(面包, 馅料):          # 定义一个叫「做三明治」的函数
    结果 = 面包 + 馅料 + 面包      # 它需要两种材料
    return 结果                    # 把做好的三明治交回来

我的午餐 = 做三明治("吐司", "鸡蛋")   # 喊它干活，材料写在括号里
```

⑥ 递归：函数自己调用自己

递归(Recursion) 是最难理解但最重要的一个概念：一个函数在自己的内部又调用了自己。它必须有两个部分，缺一个就会永远转下去，直到程序崩溃。

递归的两个必备部分	作用	类比
基准情况（Base Case）	最简单、可以直接给出答案的情况	「1 的阶乘就是 1，不用再算了」
递归情况	把问题变小一点，再调用自己	「n 的阶乘 = n × (n−1) 的阶乘」

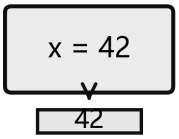
代码 0-6　递归：自己调用自己（理解它就理解了快排、归并、动态规划的一半）

```
def 阶乘(n):
    if n == 1:              # ① 基准情况：最简单，直接给答案
        return 1
    else:                   # ② 递归情况：把问题变小，再问自己
        return n * 阶乘(n - 1)

# 算 阶乘(3) 的过程：
```

```
#   阶乘(3) = 3 * 阶乘(2)
#   阶乘(2) = 2 * 阶乘(1)
#   阶乘(1) = 1            ← 到这里触底，开始往回算
#   所以 阶乘(2) = 2 * 1 = 2，阶乘(3) = 3 * 2 = 6
```

变量：一个贴了标签的盒子



数组：一排按顺序编号的盒子



取第 3 个？算一下地址就能跳过去

链表：每个盒子多一张小纸条



取第 3 个？只能从头一个一个顺着指过去

同样存 6 个数，取第 3 个的代价完全不同——数据结构的意义就在这里。

图 0-1　同样存 6 个数，取第 3 个的代价完全不同：数组能算地址，链表只能一个一个指过去

0.3　零基础也要懂的 4 个数学概念

别怕，这里不需要你会算，只需要看得懂符号在说什么。

① 求和符号 Σ：把一串东西加起来

Σ(西格玛) 读作「求和」，是「把后面这些全部加起来」的简写。

写法	读法	展开后	例子
$\sum_{i=1}^{n} x_i$	把 x_1 到 x_n 全部加起来	$x_1 + x_2 + \dots + x_n$	5 个数求和就是 $x_1+x_2+x_3+x_4+x_5$
平均值 $\bar{x}$	总和除以个数	$(x_1+\dots+x_n)/n$	5 个数的平均数
Π（派）	把一串东西乘起来	$x_1 \times x_2 \times \dots \times x_n$	连乘

② 向量与矩阵：一排数字，和一叠数字

- 向量(Vector)：一排有序的数字，比如一个人的身高体重年龄写成 [175, 70, 30]。在机器学习里，一条数据的所有特征就构成一个向量。
- 矩阵(Matrix)：一个数字组成的表格（若干行若干列）。一张灰度图就是一个矩阵，每个格子是那个像素的亮度。
- 为什么重要：神经网络里 90% 的运算就是「矩阵乘向量」——用一组权重去加权求和。

③ 导数与梯度：往哪个方向走能变好

导数(Derivative) 直观理解就是「坡度」：站在这个点上，往哪边走是上坡、有多陡。在机器学习里我们要反过来用——找坡度最陡的下坡方向往那走，就能让错误变小。

梯度(Gradient) 就是多维情况下的坡度：对于每个参数都告诉你「这个参数该往大调还是往小调」。整个第 4—7 章的模型训练，本质上都在做同一件事：算梯度，然后沿它走一小步。

④ 概率：一件事发生的可能性

- 概率取值在 0 到 1 之间：0 是不可能，1 是必然。
- 模型输出「这句话是广告的概率 = 0.87」，就是一个概率预测。
- Softmax(归一化指数函数)：把一组任意大小的分数，压成一组加起来等于 1 的概率。

0.4　怎么读懂「复杂度」：这是全书最常出现的词

复杂度(Complexity) 回答两个问题：数据变多时，耗时会怎么涨？内存会怎么涨？前者叫时间复杂度，后者叫空间复杂度。

关键的心法是：复杂度不关心「具体多少秒」，只关心「数据翻 10 倍时，耗时会变成几倍」。所以它忽略常数，只看增长趋势。这就是大 O 记号(O 是 Order 的缩写，读作「大 O」)。

写法	名字	数据翻 10 倍时	生活类比	典型例子
O(1)	常数	耗时不变	按门牌号直接找到某户人家	按下标取数组元素
O(log n)	对数	几乎不变	查字典：每次翻一半，20 次能从 100 万页查到	二分查找
O(n)	线性	耗时也翻 10 倍	从头到尾点名一次	遍历整个列表
O(n log n)	线性对数	略多于 10 倍	先点名分组、组内再排序	快速排序、归并排序
$O(n^2)$	平方	变成 100 倍	每个人都要和除自己外的所有人握手	两层嵌套循环
$O(2^n)$	指数	爆炸式增长	每增加一个选项，方案数翻倍	不做优化的穷举搜索

> **一句话记住**　看到 $O(n^2)$ 就要警觉：数据量稍微一大就会卡死；看到 O(n log n) 就放心：这是「已经优化得不错」的标志。整本书里很多算法的全部价值，就是把 $O(n^2)$ 降到 O(n log n)。

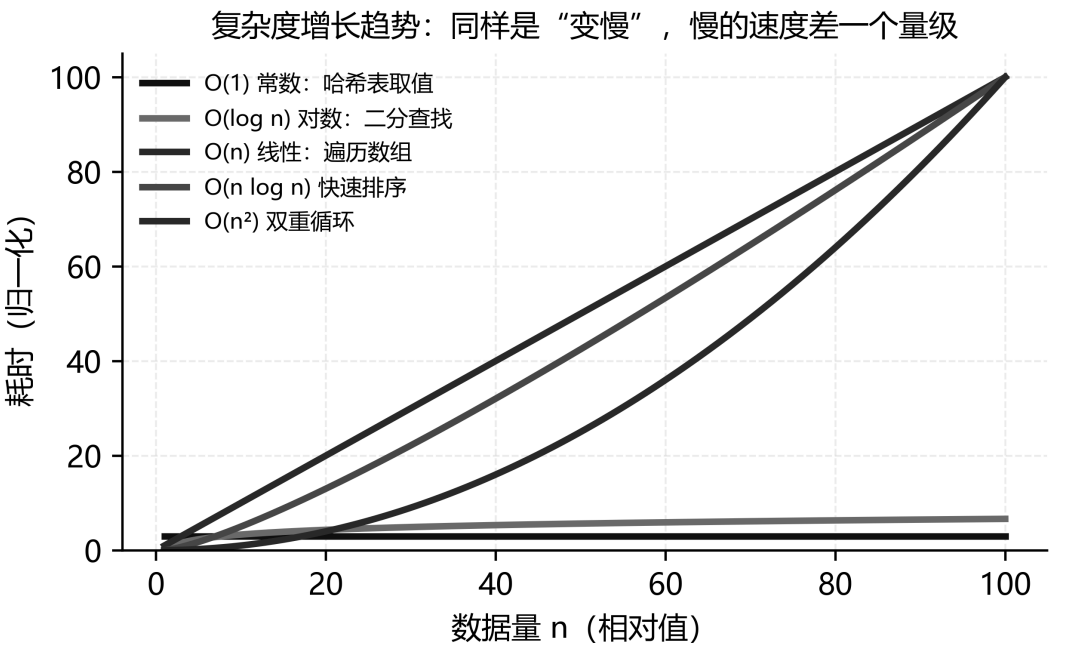


图 0-2　复杂度增长趋势：同样是变慢，慢的速度可以差一个量级

0.5　常见符号与缩写速查表（随时回来查）

符号 / 缩写	中文读法与含义	在哪里会遇到
n	数据量大小（有多少条数据）	所有复杂度表达式
d	特征维度（每个数据有多少个属性）	机器学习、神经网络
k	个数（取前 k 个、分成 k 类）	KNN、K-Means、Top-k
T	轮数 / 序列长度 / 温度	Boosting 轮数、序列长度、采样温度
α / η / lr	学习率(每次沿梯度走多大一步)	所有梯度下降类算法
γ (gamma)	折扣因子(未来的奖励要打多少折)	强化学习
λ (lambda)	正则强度(惩罚模型复杂度的力度)	岭回归、正则化
θ / w / W	模型参数(训练要学出来的那些数字)	几乎所有模型
Σ	求和(全部加起来)	损失函数、统计
ε (epsilon)	一个很小的数；或探索概率	数值稳定、ε-greedy
argmax	取让后面最大的那个自变量	分类时取概率最大的类
E[x]	期望(平均来看等于多少)	损失函数、强化学习目标

符号 / 缩写	中文读法与含义	在哪里会遇到
log	对数(默认以 2 为底时表示「折半几次」)	复杂度、交叉熵

0.6　代码里那些英文单词是什么意思

看代码最容易被英文单词卡住。这张表覆盖本书 90% 的词汇，对照着看就够了。

英文	含义	英文	含义
value / val	值、数值	max / min	最大值 / 最小值
index / idx / i, j	下标、位置编号	sum	求和
length / len / size	长度、个数	mean / avg	平均值
list / array / a	列表、数组	sort / sorted	排序
left / right	左 / 右	mid	中间位置
target / goal	目标值	pivot	基准值（快排里的参照物）
train / fit	训练、拟合	predict	预测
epoch	轮次(把全部数据过一遍)	batch	批次(一次喂进去的一小批)
lr / step	学习率 / 步长	grad	梯度
loss / cost	损失(错了多少)	score	得分
weight / bias	权重 / 偏置	layer	层
dim / hidden	维度 / 隐藏层大小	head	注意力头数
mask	掩码(遮住某些位置)	cache	缓存(存起来复用)
sample	采样(按概率随机挑)	return	返回结果
def	定义一个函数	if / else	如果 / 否则
for / while	循环	in	在……里面

0.7　读本书代码的四条建议

- 第一，先看标题和功能说明，再看代码。知道这段代码要干什么，比看懂每一行更重要。
- 第二，遇到每段代码下面的「参数与变量解读」表格和「代码逐段解读」，先看这两块，再回头看代码——它们就是给你准备的翻译。

- 第三，不要试图一次看懂所有行。先找到核心的那一两行（通常是循环体和更新公式），其余的细节可以放过。
- 第四，看不懂的代码可以跳过。本书每个算法都配有「一句话原理」和「优缺点」，即使完全不看代码，这两部分也能让你理解它是干什么的、什么时候该用。

0.8 零基础的学习顺序建议

顺序	先读	目的	大概用时
第 1 步	第 0 章（本章）	补齐最小知识，看懂术语和代码	20 分钟
第 2 步	第 1—2 章（导言 + 时间轴）	建立整体地图，知道有哪些算法	20 分钟
第 3 步	第 3 章（经典算法）	从最直观的排序查找入手，练手复杂度分析	1—2 小时
第 4 步	第 4 章（机器学习）	理解「从数据中学规律」是怎么回事	2—3 小时
第 5 步	第 6 章（强化学习）	理解「靠奖励学决策」，为第 7 章打基础	1—2 小时
第 6 步	第 7 章（LLM）	本书重点：看懂大模型是怎么 work 的	3—4 小时
第 7 步	第 5 章（深度学习）	回头补基础，理解神经网络底层	2 小时
第 8 步	第 8—10 章	看前沿、学选型、定自己的路线	1—2 小时

> **如果时间只够读一章** 读第 8 章（大语言模型）。它是当下最实用的一章，而且第 8 章里用到的每个概念，本书都在前面给了通俗解释，遇到不懂的随时往回翻就行。

0.9 把第一个程序跑起来：从装好环境到看到结果

前面讲了变量、列表、判断、循环、函数、递归。道理都懂，但真正卡住新手的第一道坎从来不是概念，而是「我该把这些字打到哪里」。这一节把流程完整走一遍，你照着做，十分钟内一定能在屏幕上看到结果。

打个比方：写程序就像做菜。菜谱（算法）你看懂了，但还得有厨房（运行环境）、有锅（编辑器）、有灶台的火（运行按钮）。三样凑齐，菜才出锅。

0.9.1 三样东西：编辑器、运行时、终端

编辑器是你打字的窗口。记事本也能写，但专业的编辑器会给你三种帮助：语法高亮（把关键字染成颜色，一眼看出哪里打错了）、自动补全（打前几个字母就提示完整名字）、括号配对（光标放在括号上会高亮另一半）。免费的如 VS Code、PyCharm 社区版都可以。

运行时是真正执行代码的程序。Python 靠 python.exe，JavaScript 靠浏览器或 node.exe，C/C++ 要先编译成可执行文件再运行。运行时版本不对是最常见的坑：代码没错、环境错了，报出来的错却像是代码错。

终端（Windows 叫「命令提示符」或 PowerShell）是你用打字的方式指挥电脑的地方。运行程序的一条命令通常只有三个部分：程序名、文件路径、参数。

> **为什么跑法不同**　C / C++ / Java / Go 要先「翻译」成机器码再跑，Python / JavaScript 由解释器边读边跑。

可以这样类比：编译型语言像先把整本书翻译好再读，解释型语言像请一位同声传译，一边听一边翻。前者启动慢一点但跑得快，后者改一行就能立刻重跑，适合练习。

- Python：装好解释器，python hello.py 直接运行。
- C：先 gcc hello.c -o hello 编译，再 ./hello 运行，两步不能少。
- Java：先 javac Hello.java 编译成字节码，再 java Hello 运行；源文件名必须和类名一致。

记不住也没关系，这三条命令在几乎所有教程里都一样。抄进一个文本文件存着，用的时候复制即可。真正要理解的是：编译和运行是两个动作，不是一步。

```
python hello.py            # 用 python 运行 hello.py
node   hello.js            # 用 node 运行 hello.js
gcc hello.c -o hello && ./hello   # C：先编译再运行
```

代码 0-12　三种语言的最小运行命令

> **记住**　报错里 90% 的「找不到文件」都是因为你当前所在的目录不是文件所在的目录。用 cd 切过去，或者直接用完整路径。

0.9.2　第一个程序：不只是打印一句话

大多数教材的第一课是打印一句 Hello World。打印确实能验证环境，但它没让你看到「计算」。我们直接从一个小计算开始：给定三个人的身高，算平均身高，并判断谁比平均值高。

```
heights = [175, 168, 182]        # 三个人的身高
total = 0                        # 累加器，从 0 开始
for h in heights:                # 把列表里的数依次取出来
    total = total + h            # 加进去
avg = total / len(heights)       # 除以人数，len 求长度
print("平均身高", round(avg, 1))
```

> **它在做什么**　先求一组数的平均，再把每个数和平均值比大小。

这是第一个真正「有用」的程序：给一串成绩，判断谁高于平均线。它把前面学过的四件事串了起来——用变量装数据、用循环走一遍、用判断做取舍、用函数打包复用。

- 第一步：把所有数加起来，除以个数，得到平均值。
- 第二步：重新走一遍数组，对每个数问一次「它大于平均值吗」。

- 第三步：把大于平均值的那些数打印出来。

这个「先算一个总指标，再逐项比较」的骨架，后面会反复出现：找最大值、找异常点、做筛选，用的都是同一套。

> **易错点**　除以个数前要确认个数不为零；另外整数相除会丢掉小数部分，做除法时至少让参与运算的一边是浮点数。

```
for h in heights:
    if h > avg:
        print(h, "高于平均")
    else:
        print(h, "不高于平均")
```

代码 0-13　第一个有实际意义的程序：求平均并比较

这段代码只有十来行，但已经把全书的三个基本动作都用上了：用变量存数据、用循环处理一批数据、用判断做出选择。后面无论多复杂的算法，基本骨架也是这三样，只是把它们套进更精巧的组织方式里。

0.9.3　把程序改坏，看看会发生什么

新手最常见的误区是「不敢改」。其实学习效率最高的一步，恰恰是有意把程序改坏，看它报什么错、错在哪一行。下面这张表列了四种故意的错误，建议你亲手试一遍。

表 0-7　四种常见的错误与它们的报错

你做了什么	报的什么错	看懂它的关键
把变量名拼错（totle）	NameError: name "totle" is not defined	「这个名字我没见过」——检查拼写
少打一个右括号	SyntaxError: "(" was never closed	语法错误在运行前就被发现，指向的是括号的位置
列表下标写成 3（只有 3 个元素）	IndexError: list index out of range	下标从 0 开始，三个元素的下标是 0、1、2
把字符串和数字相加	TypeError: can only concatenate str	类型不匹配，先转类型再算

> **经验**　三种错误里，SyntaxError 最好修（程序根本没跑起来，位置最准）；TypeError 和 IndexError 次之（位置准，原因需要想一步）；最难的是逻辑错误——程序能跑、结果不对，报错信息一个字都不给。

0.9.4　从「能跑」到「可信」：一个不太短的例子

下面这个例子把前面所有东西串起来：它从一个列表里找出第二大的数，要求不排序。先自己想一分钟，再看代码。

```
nums = [7, 3, 9, 9, 5]
first = second = float("-inf")   # 负无穷，比任何数都小
for x in nums:
    if x > first:                # 比第一名还大
        second = first           # 原来的第一名退到第二
        first = x                # 新的一号诞生
    elif x > second and x < first:  # 不如第一，但比第二强
        second = x
```

> **为什么不先排序**　排序要 O(n log n)，而这个问题一趟扫描 O(n) 就能解决。

很多人的第一反应是「排好序取倒数第二个」。结果是对的，但白白多花了力气。更好的办法是只走一遍，路上记住两个值：目前见过的最大的数，以及曾经当过最大、后来退居第二的那个数。

- 遇到比「最大」还大的数：原来的最大降级成为第二大，新数接管最大。
- 遇到没超过最大、但比「第二大」大的数：更新第二大。
- 其它情况什么都不做，继续往前走。

这种「用几个变量在遍历中维护我们关心的事实」的写法，后面还会用在最大子段和、滑动窗口、在线求均值里，是必须练熟的基本功。

> **易错点**　别忘了重复值：如果最大的数出现了两次，第二大指它还是指次大值？写代码前先确认题目口径。

```
print(second)                    # 输出 7
```

代码 0-14　找第二大的数：一次遍历，两个变量

注意 x < first 这个条件：如果没有它，遇到两个 9 时第二个 9 会把 second 也写成 9，答案就错了。这类「差一点就对」的边界，是算法题里最典型的陷阱，本书第 4 章会反复遇到。

0.10　程序出错怎么办：一份可以照着走的排错流程

排错（debug）不是玄学，它有一套稳定的流程。新手和熟手的差距，几乎全在流程上：新手盯着代码发呆，熟手按顺序做实验。这一节给出一份六步流程，遇到任何错误都可以照着走。

0.10.1　六步排错法

表 0-8　六步排错法：每一步问自己一个问题

步骤	你要问的问题	典型动作
一、读完整	报错信息我读完了吗？	读到最后一行，那里才是错误类型和原因

步骤	你要问的问题	典型动作
二、定位	它指的是哪一行？	看行号；行号不对就看上一行（可能上一行没写完）
三、复现	我能让它每次都错吗？	固定输入，反复运行，确认稳定复现
四、缩小	能不能砍掉一半代码它还错？	注释掉一半，二分法找到出错的那一小段
五、假设	我认为原因是什么？	写下一句话的猜测，再去验证它
六、验证	改完真的好了吗？边界呢？	换几组输入，特别是空、1 个、重复元素

其中第四步「缩小范围」是最有力的一步。程序有 200 行、错在第 137 行附近，那就先注释掉后 100 行，看看还错不错；再用二分的方式一点点逼近。这个方法对任何语言、任何规模的代码都有效。

0.10.2 打印语句：最朴素也最有效的工具

在关键位置插入打印，把「你认为的值」和「实际的值」对照起来，是排查逻辑错误的标准做法。诀窍是打印的时候带上位置标记，否则一屏输出全是数字，根本分不清哪行是哪行。

> **调试的第一招** 在循环里打印变量，看它每一轮到底是怎么变的。

初学阶段最有效的调试手段不是打断点，而是「让程序自己说出来」。在循环体里加一行打印，把轮次和关键变量一起输出，程序就从黑箱变成了一盘可以逐帧回放的录像。

- 打印轮次：知道现在是第几轮，方便和手算对照。
- 打印关键变量：看它是否按你预期的方向变化。
- 打印中间结果：看答案是从哪一步开始偏离预期的。

这个习惯的价值在于：一旦每一步都能看见，错误几乎都会自己跳出来。用完记得把调试打印删掉，或者加个开关只在需要时打开。

> **易错点** 别在循环里打印整个大数组，信息太多反而看不出变化；只打印你正在怀疑的那两三个变量。

```
def find_max(a):
    best = a[0]
    for i in range(1, len(a)):
        print("[debug] i =", i, "best =", best, "a[i] =", a[i])
        if a[i] > best:
            best = a[i]
    return best
```

代码 0-15　在循环里打印状态，观察每一步的变化

看输出的时候，重点不是「有没有错」，而是「哪一步开始和我预想的不一样」。第一个偏离预期的地方，就是 bug 的入口。

0.10.3　三种最费时间的错误

> **第一类**　差一错误。循环写 range(1, n) 还是 range(n)，下标从 0 还是 1 开始，多算一个还是少算一个。这类错误不会报错，只会让结果微妙地不对。对付它只有一个办法：拿 n = 0、n = 1 的小数据手工验证。
>
> **第二类**　类型与精度错误。整数相除得到小数（Python 里 / 是小数除法、// 才是整除），浮点数 0.1 + 0.2 不等于 0.3，字符串 "12" 和数字 12 不相等。涉及金额和比较时尤其小心。
>
> **第三类**　边界与空值。列表为空时取 a[0] 会崩，字典里没有这个键会崩，除数为零会崩。写函数时先在开头处理这些特殊情况，能省掉一半的调试时间。

0.11　怎么把一段代码读懂：三步拆解法

本书后面会出现大量代码。很多读者看到十几行的循环就绕晕了，其实不是智商问题，而是没有固定的读法。这一节给你一个可以重复使用的三步法。

0.11.1　第一步：找出「输入」和「输出」

先看函数签名和最开头的几行，回答两个问题：它吃进去什么（参数）、吐出来什么（返回值）。这一步看起来简单，却能立刻过滤掉一大半细节——中间的过程再怎么绕，都是为了把这个输入变成那个输出。

0.11.2　第二步：找一个最小的例子，手工跑一遍

拿一个只有三五个元素的小例子，在纸上把变量的值一行一行写下来。这一步不能省。人脑对「抽象的循环」几乎没有直觉，但对「三五个具体数字」的直觉非常强。

下面用一个具体的小例子上演一遍。目标是找出数组中和最大的连续子数组（这个问题叫最大子段和，本书第 4 章会正式讲）。

```
nums = [-2, 1, -3, 4, -1, 2]
```

> **一句话原理**　走到每个位置时只回答一个问题：要不要把前面那段接上。

题目是：给一串有正有负的数，找一段连续区间，使这段的和最大。暴力做法是枚举所有区间，要 $O(n^2)$；而正确做法只需要两个变量。

关键直觉是：如果「前面这段的和」已经变成负数，那它对后面的任何区间都是拖累，不如从当前这个数重新开始。就像背着一袋越来越沉的石头赶路，一旦发现袋子在往下坠，丢掉重来比继续背着划算。

- 第一步：维护 cur，表示「以当前位置结尾的最大和」。

- 第二步：每一步在「接着前面」和「从当前重新开始」之间取较大的那个。
- 第三步：再用一个 best 记住历史出现过的最大 cur，它就是答案。

这套「局部最优 + 全局最优两个变量」的结构，是动态规划最精简的形态。吃透它，后面看背包和一维 DP 会轻松很多。

> **易错点**　允许空区间时答案下限是 0；不允许空区间（必须选一个数）时，全是负数的情况下答案应该是最大的那个负数，而不是 0。

```
best = nums[0]
cur  = nums[0]
for x in nums[1:]:
    cur  = max(x, cur + x)   # 要么接着前面，要么从自己重新开始
    best = max(best, cur)
print(best)
```

代码 0-16　最大子段和：两个变量，一次遍历

手工跑一遍的表格如下。每一行都记录「以当前位置结尾的最大和」和「目前为止见到过的最大和」两个数，你就完全掌握了这段代码。

表 0-9　手工推演最大子段和（以 [−2, 1, −3, 4, −1, 2] 为例）

步骤	当前数 x	cur（以它结尾的最大和）	best（历史最大）	这一步在做什么
开始	—	-2	-2	两个变量都等于第一个数
1	1	1	1	-2+1 不如直接取 1，于是「重新开始」
2	-3	-2	1	1 加 -3 得 -2，虽然变小但仍是正贡献
3	4	4	4	-2+4 不如直接取 4，再次重新开始
4	-1	3	4	接着前面更划算
5	2	5	5	3+2=5，刷新历史最大

看第三行：当前数是 -3，按理说负数应该躲开，但因为前面攒下的和是 1，加上它还剩 -2，比「从 -3 重新开始」要好，所以继续加。这个「重新开始还是接着攒」的判断，就是整段代码的全部思想。

0.11.3　第三步：把变量和现实对上号

最后一步是给每个变量找一个现实中的对应物。cur 是「手里正攒着的那一叠钱」，best 是「这辈子见过最多的钱」。一旦对上了号，代码就不再是符号，而是一个你能想象出来的过程。本书后面每讲一个算法，都会尽量给你一个这样的对应物。

> **读代码的三个层次**　第一层：知道它做什么（输入输出）；第二层：知道它怎么做（手工跑一遍）；第三层：知道为什么这么做（变量背后的现实含义）。达到第三层，你就能自己把它改写成别的样子了。

第 1 章　导言：这份文档怎么用

> **零基础从这里进**　如果你完全没学过编程或算法，请先花 20 分钟读第 0 章预备知识，那里讲清了变量、循环、递归(函数在自己的内部又调用了自己)和「怎么读复杂度」，读完这一章会顺畅很多。

把算法放进同一条时间轴就会看到，它们其实共享一个主题：用更少的代价解决更难的问题。从 1945 年冯·诺依曼写下归并排序，到 2017 年 Transformer 用一次注意力(按相关程度给不同位置分配权重(代表重要程度或连接强度、可被调节的数值)，再加权求和)替换掉循环结构，再到今天千亿参数的大语言模型（LLM），每一代算法都在回答同一个问题：在给定的算力、数据和时间约束下，如何做出更好的决策或预测。

本文档按时间轴组织，但按「能解决的问题类型」分层讲解。每个算法统一按七项拆解，方便横向对比与速查：

拆解维度	回答什么问题
解决的问题	它的输入是什么、输出是什么，原本靠什么笨办法做
核心原理	一句话说清它在数学或结构上做了什么关键动作
基本代码	可直接运行的最小实现（Python 为主）
实现原理	代码背后的数据结构、递推关系或优化目标
复杂度	时间与空间随规模 n 增长的趋势
优缺点	什么场景该用、什么场景会翻车
性能参数	工程落地时真正要调的旋钮（超参、显存(显卡上的内存，决定能装多大的模型)、吞吐、精度）

1.1　三条并行演进的谱系

- 经典算法（确定性问题）：输入明确、答案唯一，追求的是「更少的操作次数」。排序、查找、最短路、动态规划(把子问题的答案存起来复用，避免重复计算)。
- 机器学习(让程序从数据里自己总结规律，而不是人写死规则)算法（从数据中归纳）：答案不由人写死，而是从样本(数据集里的一条记录)里学出来，追求的是「更好的泛化(遇到没见过的数据仍然能做对的能力)」。回归、决策树、随机森林(用许多棵随机化的决策树投票的集成模型)、SVM。

- 深度学习(用多层神经网络(由多层可调节的加权求和堆叠而成的函数)自动学习特征(用来做判断的依据，也就是输入变量)表示)与大模型（表示学习 + 规模化）：不再手工设计特征，让网络自己学表示，并把参数、数据、算力同时推到极限。CNN、Transformer、LLM。

> **阅读建议**　如果你只关心 AI 部分，可直接跳到第 5 章（机器学习）、第 6 章（深度学习）、第 8 章（LLM）；第 8 章是全文最厚的一章，含 Attention 手写实现、训练三阶段、推理优化与显存估算公式。

1.2　一个贯穿全文的判断标准

评价算法不要只看「精度高不高」，而要同时看四个成本：算力成本（FLOPs(浮点运算次数，衡量算力量级) / 显存）、数据成本（要多少标注）、时间成本（训练时长与推理延迟）、维护成本（可解释性与可调试性）。很多工业场景里，一个能被业务方解释清楚的逻辑回归，价值高于一个黑箱的深度模型——这就是第 10 章选型表的出发点。

1.3　同一个问题的六种解法：从「两数之和」说起

前面两节讲的是「这本书怎么看」。从这一节开始，我们用一个小到不能再小的问题，把后面全书要反复用到的思考方式先走一遍。

题目是这样的：给一个整数数组 nums 和一个目标值 target，请找出数组中两个不同的位置 i 和 j，使 nums[i] + nums[j] = target。只要找出一组即可。

换个说法你马上就懂了：一个班有 50 个同学，他们的身高都记在一张表上，校长问「有没有两个人身高加起来正好是 3.20 米」。你要在这张表里把这一对找出来。

就是这么一件小事。但把它的六种做法写出来、把代价算清楚，你就已经掌握了本书一半的方法论。

1.3.1　解法一：一个一个试（暴力双重循环）

最直接的想法是：拿起第一个人，和他之后的每一个人配一次；配不上，就换第二个人，再和他之后的每一个人配一次。这种做法不遗漏任何一种组合，所以一定不会错。

代码 1-1　两数之和：暴力双重循环（最好懂，但最慢）

```
def two_sum_brute(nums, target):
    n = len(nums)
    for i in range(n):
        for j in range(i + 1, n):
            if nums[i] + nums[j] == target:
                return [i, j]
    return []
```

它的代价很好算：n = 50 时，比较次数最多是 50 × 49 ÷ 2 = 1225 次；n = 10000 时，是约 5000 万次。也就是说 n 翻一倍，工作量差不多翻四倍——这就是 $O(n^2)$ 的直观含义。

1.3.2 解法二：先排队，再从两头往中间夹（排序 + 双指针）

把身高表按从矮到高排好。然后让最矮的和最高的配一次：如果和太小，说明最矮的这个人配谁都不够，可以放弃了，把左手往右移一位；如果和太大，说明最高的这个人配谁都会超，把右手往左移一位；正好相等就找到了。

这个做法之所以正确，是因为排好序之后，「最矮的配最高的都不够」这句话等价于「最矮的配谁都不够」。一次比较就能排除掉一个人，所以最多移动 2n 次。

代码 1-2　两数之和：排序 + 双指针（不需要额外数组）

```
def two_sum_sorted(nums, target):
    arr = sorted([(v, i) for i, v in enumerate(nums)])
    lo, hi = 0, len(arr) - 1
    while lo < hi:
        s = arr[lo][0] + arr[hi][0]
        if s == target:
            return sorted([arr[lo][1], arr[hi][1]])
        if s < target:
            lo += 1
        else:
            hi -= 1
    return []
```

> **注意**　这一版返回的是排序前的原始下标，所以排序时必须把下标一起带上；很多同学做完排序就忘了原下标，结果返回的位置全错。

1.3.3 解法三：把见过的数记在小本子上（哈希表）

换一个思路：我们从左往右只看一遍。看到 nums[i] 时，先问一句「我刚才是不是见过 target - nums[i] 这个数？」如果见过，那两个人就是它俩；如果没见过，就把 nums[i] 记到小本子上，继续往下走。

小本子就是哈希表。查一次、记一次都是常数时间，所以整体只需要看一遍数组，O(n)。

代码 1-3　两数之和：哈希表一趟扫描（推荐写法）

```
def two_sum_hash(nums, target):
    seen = {}                       # 值 -> 下标
    for i, v in enumerate(nums):
        need = target - v
        if need in seen:            # 小本子上查补数
            return [seen[need], i]
        seen[v] = i                 # 把自己记上去
    return []
```

三行核心逻辑，跑一遍就出结果。它比排序更快，代价是多用了一个最多装 n 个元素的哈希表。这就是本书要反复讲的「用空间换时间」。

1.3.4　还有三种写法，各自的适用场合不同

（1）排序 + 二分查找：排好序以后，对每个 nums[i]，在剩下的部分里二分查找 target - nums[i]。总代价 O(n log n)。它比双指针多写几行，但在「需要多次查询不同目标值」时更灵活。

（2）先排序再建哈希：先排序拿到下标映射，再走哈希表。看起来是绕了一圈，但当输入本来就有序、或者排序是上游已经付过的成本时，它反而是最省的一步。

（3）位图（值域很小时）：如果这些数都是 0 到 100 万之间的整数，可以用一个比特数组当小本子，一个比特表示「这个数出现过」。查找更快、内存更省，但只在值域很小时才成立。

表 1-1　同一道题的六种解法：代价与适用场合

解法	时间复杂度	额外空间	要求输入有序	什么时候用它
暴力双重循环	$O(n^2)$	O(1)	否	n 很小（几十个），或者只是先写个能跑的版本
排序 + 双指针	O(n log n)	O(n) 存下标	排一下就有	只需要找一组答案，且愿意改数组顺序
哈希表一趟扫描	O(n)	O(n)	否	通用最优解，绝大多数场合直接选它
排序 + 二分查找	O(n log n)	O(n)	是	目标值会变、需要多次查询
先排序再哈希	O(n log n)	O(n)	顺带得到	排序是上游已经付过的成本时
位图 / 桶	O(n)	O(V)，V 为值域	否	整数且值域很小（例如成绩 0～150）

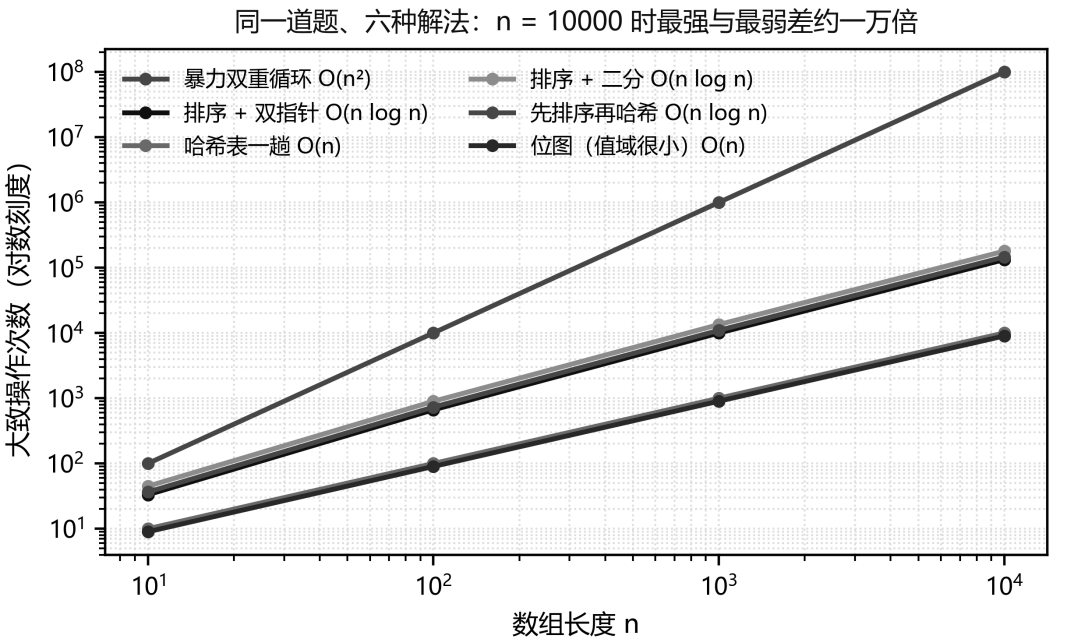


图 1-1　同一道题的六种解法：n 越大，差距越夸张（纵轴为对数刻度）

把这张图看明白，你就能理解为什么「算法」这两个字值得单独写一本书。同样是正确的结果，n = 10000 时最慢和最快的做法差了大约一万倍。如果这段代码是每天要跑一千次的业务逻辑，这个差距就是「一秒钟出结果」和「三个小时还没跑完」。

> **一句话记住**　绝大多数「太慢了」的问题，都不是靠换更快的机器解决的，而是靠换一个数量级更低的算法解决的。

1.3.5　这六种解法教的四件事

- 正确性优先：六种解法结果都是对的，先保证对，再谈快。
- 复杂度决定上限：硬件只能把常数压一压，压不动 n^2 和 n 之间的鸿沟。
- 空间和时间可以互换：哈希表和位图都是用内存买速度。
- 没有绝对最优：值域很小就上位图，需要多次查询就用二分，通用场景用哈希表。

1.4　好代码的五个维度

「这段代码好不好」这件事，新手常常只有一个模糊的感觉。其实它可以拆成五个能分开评价的维度。你以后写任何一段算法代码，都可以拿这五条过一遍。

表 1-2　评价一段算法代码的五个维度

维度	一句话解释	反面例子

维度	一句话解释	反面例子
正确	在任何合法输入上都给出正确答案	只对「数组里恰好有一组答案」成立，遇到无解就崩
可读	别人（包括三个月后的你）能一眼看懂在做什么	用 a、b、c、d 当变量名，嵌套五层 if
够快	在预期规模下能在可接受的时间内跑完	在 100 万条数据上用 $O(n^2)$ 双重循环
省内存	额外占用的内存和输入规模相称	为了找两个数，先把整个数组复制三遍
能改	需求变了以后，只需要改一小块地方	把「排序」「查找」「输出」全塞进一个函数里

这五条之间会互相拉扯。想要更快，往往要多花内存；想要更省内存，代码可能就绕一点。真正的工作不是追求某一项满分，而是在给定场景下找到合适的平衡点——这正是第 10 章「选型指南」要解决的问题。

> **给新手的顺序**　先正确、再可读，然后才考虑快和省的优化。顺序颠倒过来，你会在一个本来就写错的程序上白花好几天。

1.5　算法的三种能力：会写、会改、会选

「学会算法」这句话太笼统了。它实际上包含三种能力，难度是阶梯式上升的。

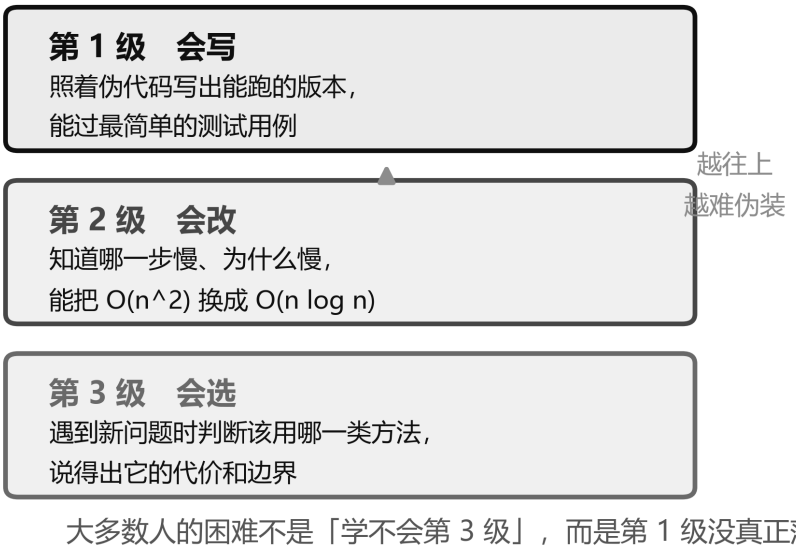


图 1-2　算法能力的三级台阶：大多数人卡在第 1 级没有真正落地

第 1 级「会写」：给你一段伪代码或一个思路，你能把它翻译成能运行的代码，并用最简单的例子验证它是对的。这一级的门槛比想象中低，也比想象中重要——它是唯一一个无法跳过、只能亲手练的一级。

第 2 级「会改」：程序跑得慢，你能定位到是哪一步慢，说得出它为什么慢，并且能想到用什么办法把它降下来。这一级开始需要「复杂度直觉」，也就是本书反复训练的那种感觉。

第 3 级「会选」：面对一个全新的问题，你能判断它属于哪一类，该用哪一种方法，并且知道这种方法的边界在哪里、代价是什么。做到这一级，你才算真正把算法变成了工具。

> **怎么自测**　拿任意一道题，如果你能说清「这题为什么用这个方法而不是另一个」「它的复杂度是怎么算出来的」，你就在第 2 级；如果两题之间你只能背出解法而说不出为什么，那还停在第 1 级。

1.6　本书的阅读地图：十二章分别解决什么问题

这本书一共十二章加七个附录。第一次拿到厚书的人最容易犯的错误，是从第一页开始一页页往后读——那样读到第 3 章就会卡住。

正确的读法是先看清每章是干什么的，然后按需要跳着读。下面这张表就是全书的地图：先看你手上的问题落在哪一行，再直接翻过去。

表 1-3　全书阅读地图：每章解决什么问题、需要什么前置、建议投入多少时间

章	解决的问题	前置	建议用时
第 0 章	完全零基础：编程概念、数学概念、复杂度怎么读	无	2～3 天，可随时回查
第 1 章	建立判断标准：什么样才算好代码、好算法	无	1 天
第 2 章	用时间轴看清各大流派从哪来、为什么出现	无	半天，当故事读
第 3 章	数据结构：所有算法的地基	第 0 章	2～3 周，本书最重要的一章
第 4 章	经典算法：排序、查找、图、动态规划等	第 3 章	3～4 周
第 5 章	机器学习：从数据中学规律	第 4 章 + 一点线性代数	3 周
第 6 章	深度学习：让模型自己学表示	第 5 章	2～3 周

章	解决的问题	前置	建议用时
第 7 章	强化学习：在试错中学决策	第 5、6 章	2 周，可分次读
第 8 章	Transformer 与大语言模型（本书重点）	第 6 章	3～4 周
第 9 章	2025—2026 前沿：从拼规模到拼效率	第 8 章	3～5 天，需要持续更新
第 10 章	选型指南：按场景选算法、算显存与算力	全书	当成工具书，用时再查
第 11 章	学习路线与结语	无	半天
附录 A～G	术语、代码模板、仿真对照、选型总表、练习题、调试手册、公式速查	无	随时查

> **三条阅读路线**　只想听懂概念：读第 0、1、2 章 + 第 11 章即可。想动手写程序：第 0、3、4 章是必修。想做 AI 方向：第 3、5、6、8 章连起来读，第 4 章只挑排序与复杂度部分。

1.7　配套仿真怎么用：一个四步闭环

本书配了 79 个可以交互的算法仿真，访问地址见「配套资源」一节。它们不是装饰，而是这本书最容易被低估的部分——很多人看书能看懂，一动手就写不出来，缺的就是这一步。

建议每个仿真都按下面四步走一遍。一个仿真花 10 分钟，比读两页文字有效得多。

表 1-4　用仿真的四步闭环

步骤	做什么	你会得到什么
读	先看左侧动画：数据是怎么一步一步变化的	对「这个过程长什么样」有画面感
跑	点运行，把速度调慢，观察每一步的状态变化	把书上的伪代码和真实执行对上
改	在右侧编辑器里改动参数或几行代码，再运行	知道哪些部分是可以动的，哪些动不得
比	换一种实现方式，对比运行结果与执行步数	建立「同一种算法有多种写法」的意识

仿真里还准备了「参考实现」和「可拖拽的代码块」。前者是同一种算法的不同写法（例如排序的递归版与迭代版），后者是可以直接拖进编辑器的小部件。它们的作用都是

降低你动手的门槛：不用从空白文件开始写，先拼出一个能跑的版本，再逐步替换成自己写的。

1.8 零基础最常见的十二个疑问

下面这些问题是初学阶段问得最多的。如果你也有，先看这里，不必从头焦虑。

表 1-5 初学者常见疑问速答

疑问	回答
数学不好能学吗？	能。本书需要的数学在第 0 章全部讲过，剩下的都是「看懂公式在说什么」，不需要推导。
要先把编程语言学完吗？	不用。会写循环、会写函数、会用列表（数组），就能开始读第 3 章。
算法是不是要背很多代码？	不是。本书 117 段可运行代码是给你改的，不是给你背的。要记的是「什么场景用什么思路」。
为什么我看懂了却写不出来？	因为「看懂」只调用了识别能力，「写出来」调用的是生成能力。唯一办法是合上书自己写一遍。
刷题和学算法是一回事吗？	不是。刷题是检验，学算法是建立判断力。只刷题不学原理，换一道新题就废了。
复杂度算不准怎么办？	先会算数量级（n、$n \log n$、n^2），系数和常数项在初学阶段不重要。
要不要一上来就学深度学习？	不建议。没有第 3、4 章的基础，第 6 章只能背概念；有了基础，第 6 章一读就通。
看不懂伪代码里的数学符号？	附录 G 有符号速查表，第 0.5 节有常见符号表，随时回查。
学多久才算入门？	按每天 1.5 小时算，第 3、4 章扎实过一遍大约 6 周，就算入门了。
记不住怎么办？	记不住是正常的。本书的定位是工具书：重要的对照表都放进了附录，用的时候查就行。
遇到看不懂的段落怎么处理？	先跳过，标记一下继续往后读。很多概念是在后面某一节里被顺带讲清楚的。
怎么知道自己学对了？	拿一个没见过的例子自己算一遍，能算对就是学对了。本书每章末都有练习题可以自测。

第 2 章　算法发展时间轴（1945—2026）

> **读前提示** 本章只需要你认识 O(...) 这个记号（在第 0 章 0.4 讲过）。看到不认识的算法名不用慌，后面每一章都会展开讲，这里先建立整体印象。

下面这张时间轴把算法分成四条泳道：经典算法（蓝）、机器学习（绿）、深度学习与大模型（红）、强化学习（紫）。整体节奏很清晰：经典算法在 1945—1990 年集中爆发，机器学习在 1980—2010 年逐步体系化，强化学习从 1989 年 Q-learning 起长期蛰伏于学术界，而 2012 年 AlexNet 之后，算力与数据的双重红利让深度学习在十年内完成了能力超越——并且反过来把强化学习也带成了主角（DQN、AlphaGo、RLHF(用人类偏好反馈做强化学习，让模型更听话)）。

算法发展时间轴

四条并行演进的谱系 · 第 1 页（共 2 页） 1945 — 1986

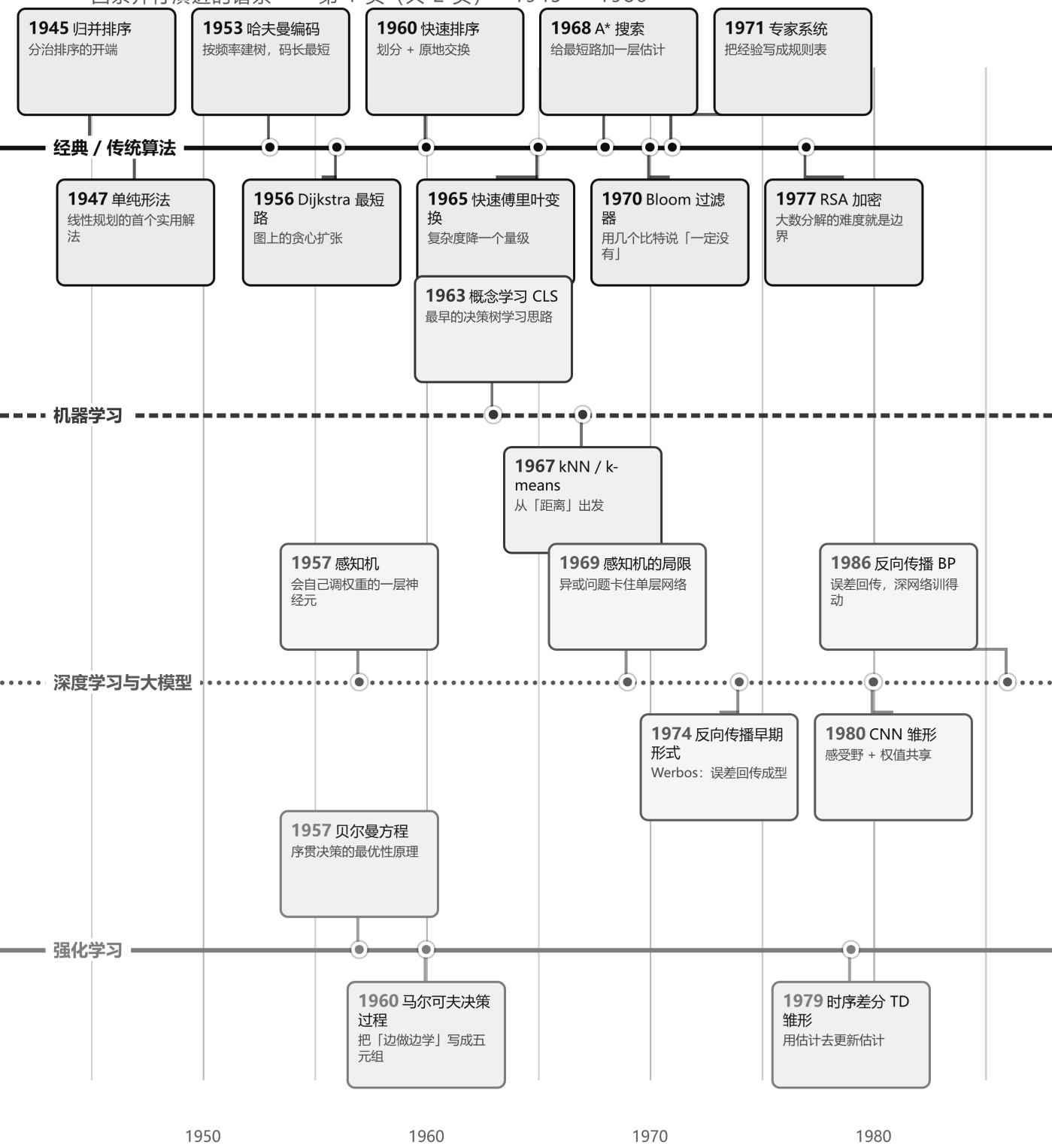
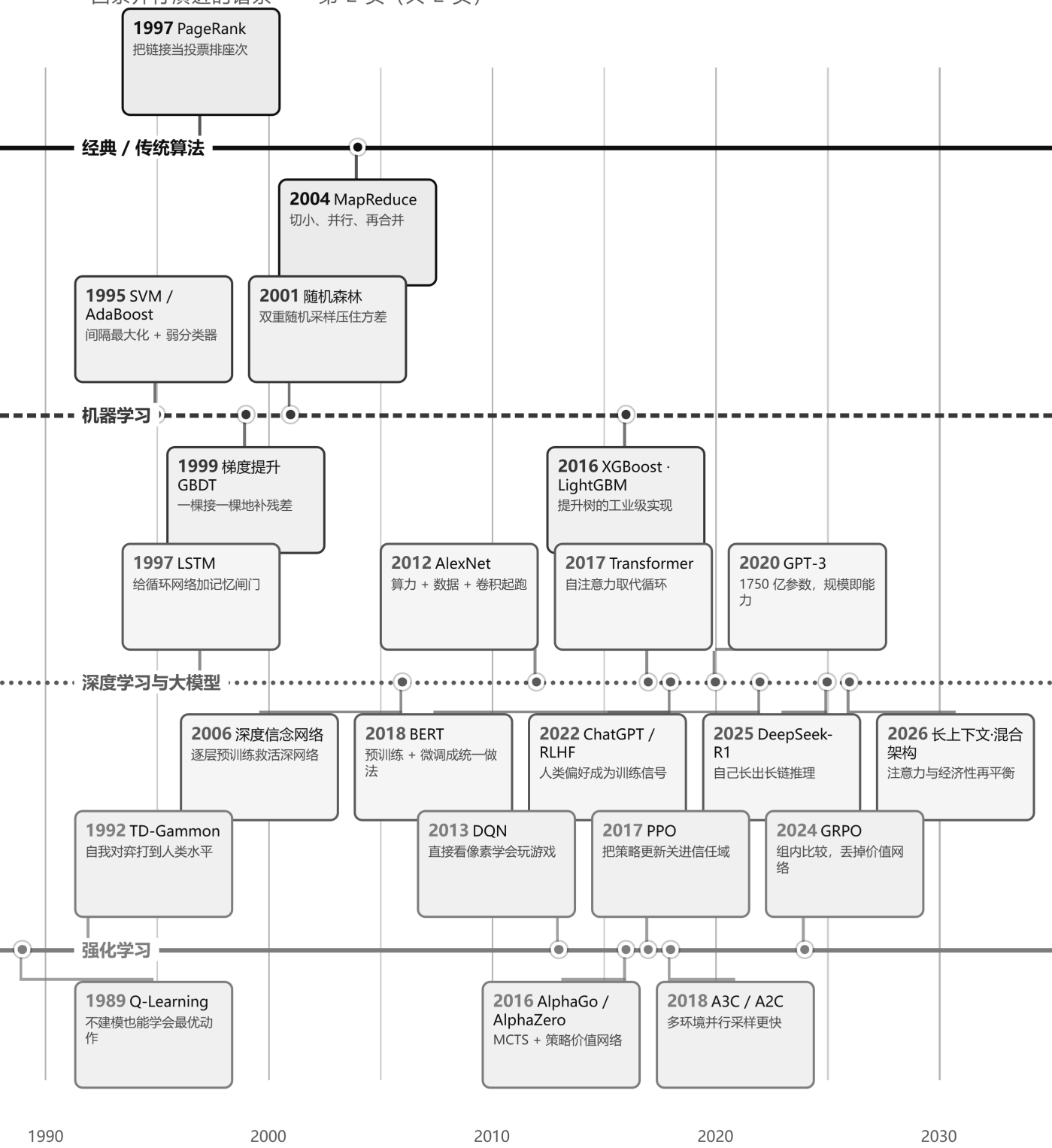


图 2-1　算法发展时间轴：四条并行演进的谱系（1945—2026）

横轴年份、纵轴谱系。圆点表示该算法首次提出或成为主流的年份，方框内上行是名称、下行是它在解决什么问题。四条谱系用线型区分：实线为经典算法、虚线为机器学习、点线为深度学习与大模型、点划线为强化学习。本页与右页沿书脊拼成一张完整长图：两页比例尺相同，书脊位置对应 1988 年，刻度可以对着读。

（接左页）1988 — 2026

四条并行演进的谱系 · 第 2 页（共 2 页）



本页自 1988 年起（左页最右端即此处），与左页共用同一比例尺，四条谱线的位置与左页一一对应，线型也一致。书架夹住的是最靠近书脊的一小条，所以本页左侧特意留出了空白，所有方框与刻度都在这一条之外，不会被装订吃掉。

2.1 里程碑事件与影响

年份	里程碑	谱系	它改变了什么
1945	归并排序（von Neumann）	经典	首次给出 O(n log n) 的分治(把大问题拆成同类小问题，分别解决再合并)排序证明
1956	Dijkstra 最短路	经典	图搜索从枚举走向贪心 + 优先队列
1957	感知机 Perceptron	ML	第一个可从数据中学习的线性分类器
1960	快速排序（Hoare）	经典	原地分区，常数极小，成为通用排序默认实现
1968	A* 启发式(不保证最优、但通常够用的经验规则)搜索	经典	引入启发函数(用来估计「还剩多远」的函数)，把搜索代价从全局压到局部
1986	反向传播(从误差出发，倒推每一层该改多少) BP + 决策树 ID3	ML	神经网络可训练；树模型可自动建规则
1995	SVM + AdaBoost	ML	间隔最大化理论；集成学习(让多个模型一起投票，比单个模型更稳)第一个可证明的提升算法
1989	Q-learning（Watkins）	RL	无模型时序差分(用「一步真实奖励 + 对未来的估计」来更新估计值)学习，RL 走出实验室的理论基石
1992	TD-Gammon	RL	首个达到人类顶尖水平的自对弈强化学习系统
1997	LSTM	DL	用门控解决长序列梯度(让损失下降最快的方向，也就是「该往哪调」)消失，统治 NLP 二十年
2001	随机森林（Breiman）	ML	Bagging + 随机特征，几乎免调参的强基线
2012	AlexNet	DL	GPU + ReLU + Dropout(丢弃法(训练时随机让一部分神经元不工作，防止死记硬背)：训练时随机让一部分神经元(一个加权求和，再加一个非线性开关)不工作)，引爆深度学习浪潮
2016	XGBoost	ML	二阶近似 + 工程优化，表格数据长期霸主
2015	DQN 打通 Atari	RL	深度网络 + Q 学习，RL 第一次在像素输入上超过人类
2016	AlphaGo	RL	策略网络 + MCTS 搜索，击败围棋世界冠军，RL 破圈
2017	PPO(近端策略优化：限制单步更新幅度、训练稳定的策略算法)	RL	裁剪目标让策略优化稳定可控，成为 RLHF 的默认算法
2017	Transformer	DL	注意力替代循环，训练可完全并行，为大模型铺路

年份	里程碑	谱系	它改变了什么
2018	BERT / GPT	DL	预训练(在海量无标注数据上先学通用能力) + 微调(在已有模型上用少量数据继续训练，让它适应具体任务)成为 NLP 新范式
2020	GPT-3（175B）	LLM	验证 Scaling(规模化：把参数、数据、算力一起放大) Law(规模定律(模型效果随参数、数据、算力按幂律提升的经验规律)：效果随参数、数据、算力按幂律提升)：规模带来涌现(规模越过某个门槛后突然出现的新能力)的少样本能力
2022	ChatGPT（RLHF）	LLM	偏好对齐(让模型的行为符合人类的期望与价值观)让模型「听话」，走向大众
2025.01	DeepSeek-R1	LLM	纯强化学习激发长链推理，「推理模型」成为独立品类，GRPO(组内相对策略优化，可用规则奖励替代评分网络) 走向主流
2025.04	Qwen3 混合思考(同一个模型可以在快速作答和深度思考之间切换)	LLM	快/慢思考在同一模型内切换，推理成本首次变成可调旋钮
2026.04	DeepSeek-V4	LLM	MoE(混合专家：每层多个专家，每次只用少数几个) + 混合注意力 + mHC + Muon；1.6T 总参仅激活 49B，1M 上下文(模型当前能看到的全部输入内容)
2026.06—08	开源万亿 MoE 潮	LLM	GLM-5.2、LongCat-2.0、Kimi K3、Qwen3.8、Hy4 等密集开源，Apache/MIT 成竞争筹码
2026.07	Mamba-3（ICLR 2026）	架构	梯形离散化 + MIMO + 复数动力学，把 SSM(状态空间模型：用固定大小的状态传递历史) 的精确追踪短板补上一大截
2026.09	1bit 级量化(用更少的比特存数字，省显存和带宽)开源落地	工程	稀疏三值量化把万亿模型权重压缩约 86%，端侧与单机部署门槛大幅下降

2.2　时间轴背后的三条规律

- 规律一：并行化能力决定上限。RNN 必须串行，所以规模上不去；Transformer 把序列依赖变成矩阵乘法，才让千亿参数成为可能。
- 规律二：数据形态决定算法形态。结构化表格 → 树模型；图像 → CNN；序列 → RNN/Transformer；图 → GNN；需要连续决策 → 强化学习。
- 规律三：每一代新算法都不是完全取代旧算法，而是收编它。今天 LLM 的推理服务里，仍然跑着快排、哈希(把任意内容换算成一个编号)、BPE 分词(把文本切成一个个模型认识的小单位)和 beam search。

想看最新的两年 2025—2026 年的具体进展（混合架构、MoE、稀疏注意力(只计算一部分位置之间的注意力，跳过其余)、混合思考、1bit 量化、Agent 产业化）集中在第 9 章，本表只列到里程碑级别。

2.3 把时间轴读成一张地图：四大家族的分工

第 2.1 节把八十年的算法放在一条轴上。但一条轴只能看清「先后」，看不清「分工」。这一节换一个视角，把所有的算法归成四个家族，看它们各自负责回答一类什么问题，以及为什么它们至今谁也替代不了谁。

2.3.1 四个家族各自在回答什么问题

表 2-4 算法四大家族的分工

家族	它回答的问题	典型成员	成熟年代	至今的地位
数据结构	数据怎么摆，才能又快又省地取用	数组、链表、哈希表、树、堆、并查集	1950s—1970s	所有算法的基础，不会过时
经典算法	在确定的输入下，怎么算出确定的答案	排序、二分、最短路、动态规划、图算法	1945—1990	工程中标配，仍在天天用
机器学习	从数据里归纳出规律，用于预测	回归、决策树、SVM、聚类、集成学习	1950s—2010s	传统模型仍是表格数据的首选
深度学习与大模型	从海量数据里自动学出表示	CNN、RNN、Transformer、扩散模型、LLM	2012—至今	感知与生成任务的主流
强化学习	在没有标准答案的环境里，靠试错学决策	Q-learning、DQN、PPO、GRPO	1950s—至今	决策与控制场景不可替代

这五个家族不是替换关系，而是分工关系。你今天用的一个推荐系统，可能同时用到哈希表（去重）、堆（Top-K）、梯度提升树（点击率预测）、Transformer（内容理解）和强化学习（长期收益优化）。

2.3.2 三个反直觉的事实

第一个 最古老的算法反而最稳定。归并排序 1945 年提出，二〇二六年还在用，因为它的 O(n log n) 已经触及比较排序的理论下界，没有改进空间了。

第二个 最前沿的模型也在回到老结构。Transformer 的核心是矩阵乘法与点积，这两个操作是 1950 年代的产物。所谓的「新」，往往是旧零件的重新组合，而不是凭空造出新东西。

> **第三个**　很多算法消失不是因为错了，而是因为硬件变了。冒泡排序在内存昂贵、数据量小的时候是合理选择；当内存便宜到能随便开数组，当 CPU 缓存比算法复杂度更重要时，它就退场了。判断一个算法是否过时，要先看它的前提是否改变。

表 2-5　五个被时代改变的前提

变化的前提	从什么变成什么	受影响最大的算法	结果
内存价格	昂贵（KB 级）→ 便宜（GB 级）	原地排序、位图	空间换时间的做法变得普遍
CPU 与内存速度差	接近 → 差几百倍	链表、跳表	数组重新胜过链表（缓存友好）
并行能力	单核 → 多核 + GPU	递归为主的算法	能向量化的算法胜出
数据规模	万级 → 亿级	$O(n^2)$ 类算法	全部被 O(n log n) 或线性算法取代
可解释性要求	低 → 高（合规）	深度模型	树模型与规则系统部分回潮

把这张表和上面的时间轴对照着看，你会发现一个规律：算法的兴衰很少由「数学上更优美」决定，更多由「硬件和需求的变化」决定。这也是本书为什么反复强调要看约束条件，而不是单纯比谁更强。

2.4　每个时代的瓶颈是怎么换的

八十年的算法史，本质上是「瓶颈不断转移」的历史。每解决一个瓶颈，下一个就会浮现出来。看清这条主线，比记住具体年份更有价值。

2.4.1　五次瓶颈转移

表 2-6　算法史上五次瓶颈转移

时期	当时的瓶颈	突破方式	代表成果	转移后新的瓶颈
1945—1960	人工计算太慢	把步骤形式化，交给机器	归并排序、Dijkstra、动态规划	内存太小，装不下数据
1960—1980	内存与存储昂贵	用更省空间的表示	哈希表、B 树、原地排序	单机算力到顶
1980—2000	数据开始变大，暴力不可行	更快的渐进复杂度	KMP、快速选择、近似算法	数据有了但没人会归纳
2000—2012	数据有了，规律难以提取	统计学习方法论	SVM、随机森林、GBDT	特征要人工设计，成本高

时期	当时的瓶颈	突破方式	代表成果	转移后新的瓶颈
2012—2020	手工特征跟不上任务复杂度	让模型自己学特征	CNN、RNN、Transformer	模型太大，训练与推理成本失控
2020—2026	算力与能耗成为硬约束	稀疏化、混合架构、量化、蒸馏	MoE、稀疏注意力、1-bit 量化	如何保证可靠性、可控性

每一行都遵循同一个模式：瓶颈出现 → 有人提出新的组织方式 → 复杂度降一个量级 → 新的问题出现 → 瓶颈转移到别处。理解了这一点，你看待任何「最新技术」的方式都会不一样：它不是终点，只是这一轮循环中的一次突破。

2.4.2 给读者的三条读史建议

> **第一条** 记住「为什么」比记住「什么时候」重要。知道 Transformer 是因为 RNN 无法并行才出现的，比记住它 2017 年发表更有用。
>
> **第二条** 每个算法都要放回它的约束里看。B+ 树在内存里没有优势，但在磁盘上无可替代——因为磁盘一次读写 4KB，这个约束决定了结构。
>
> **第三条** 别急着追新。真正值得投入的是那些跨越了多个时代的结构：哈希、树、堆、动态规划、梯度下降。它们不是热点，但它们是地基。

2.5 同一个想法被发明了三次：三个真实故事

时间轴如果只是一串年份，看一遍就忘了。真正值得记住的是：某个想法为什么会在那个时间点冒出来，又为什么会反复地以不同名字重新出现。下面三个故事就是典型。

2.5.1 「分而治之」被发明了三次

第一次是 1945 年冯·诺依曼团队写归并排序：把一段数分成两半，各自排好，再合并。第二次是 1960 年霍尔提出快速排序：不真的一分为二，而是按一个基准值把数据推成「小的在左、大的在右」，然后再分别处理。第三次是 1970 年代以后在并行计算里，同样的思想换了个名字叫「MapReduce」——把大任务切开、并行处理、再把结果归拢。

三次发明之间隔着几十年，硬件从电子管换到了多核处理器，但内核是同一个：当问题的规模大到无法一次处理，就把它切成互不干扰的小块。

2.5.2 「用位置换时间」出现了两次高峰

哈希表的核心思想——不比较，直接算出数据该放在哪——在 1953 年由 IBM 的汉斯·彼得·卢恩提出。半个多世纪后的 2017 年，Transformer 用几乎同样的思路解决了一个完全不同的问题：注意力机制不再按顺序一个个读词，而是让每个词直接「看到」所有其他词，用一次矩阵乘法一次性算出所有关系。

两次革命发生在完全不同的领域，但都在解决同一类瓶颈：顺序处理的代价太高，那就想办法让数据自己「跳到正确的位置」。

2.5.3　「让模型自己学特征」被讲了三次

1986 年反向传播让多层网络第一次能真正训练起来；2012 年 AlexNet 用 GPU 把这件事推到了实用；2020 年之后的预训练加微调范式，把「自己学特征」从「学一层表示」变成了「学一个世界」。

这三次的递进关系很值得细看：算法在 1986 年就基本就位了，缺的是算力和数据。所以当你看到某个「新技术」时，不妨先问一句：它是新想法，还是旧想法等到了合适的土壤？

> **读史的价值**　知道一个方法为什么被发明，比记住它怎么用更有用——因为「为什么」决定了它的适用边界。

2.6　时间轴之外的暗线：硬件、数据与算力

前面那条时间轴记录的是「谁在什么时候提出了什么」。但只看这条明线，很多现象解释不通——比如为什么 1986 年就有的反向传播，要到 2012 年才真正起飞？

答案是还有三条暗线一直在下面推着走：硬件、数据、算力。三条暗线每换一次台阶，就会把一批原本「理论上可行」的算法推到「工程上可用」。

表 2-7　三条暗线的四次台阶

时间	硬件	数据规模	算力	被推上台的方法
1950 年代	电子管 / 继电器	人工卡片，几千条	每秒万次量级	排序、查找、线性规划
1970 年代	晶体管、磁盘	企业数据库，百万条	每秒百万次量级	关系数据库、B 树、哈希索引
1990 年代	个人电脑、互联网	网页，十亿条	每秒十亿次量级	PageRank、搜索引擎、统计机器学习
2010 年代	GPU 集群	图像、文本，万亿条	每秒千万亿次量级	深度卷积网络、大规模预训练
2020 年代	专用加速器、超大规模集群	全网文本与多模态	单集群十万卡级	大语言模型、混合专家、推理时计算

看这张表能得出一个重要结论：算法和硬件是互相等待的关系。反向传播等了 26 年才等到 GPU，Transformer 提出的当年就赶上了合适的算力台阶，所以只用了两年就席卷了整个行业。

> **判断技巧**　当你看到一个新方法时，先看它属于「新想法」还是「旧想法 + 新土壤」。前者需要重新理解，后者往往只需要重新评估它的性价比。

2.7 怎么读一条算法新闻

这本书出版的时候，2026 年还没过完，这个领域每个月都有新东西出来。与其追每一条新闻，不如掌握一套判断方法：拿到一条消息，按下面六个问题过一遍，三分钟就能判断它值不值得花时间。

表 2-8 读算法新闻的六个提问

问题	为什么要问	看什么信号
它解决的是哪个瓶颈？	瓶颈决定了方法的适用边界	是速度、内存、数据量，还是精度
和现有方法比，提升在哪一栏？	大多数「突破」只在某一栏提升	看对比表里哪一行变好、哪一行变差
代价是什么？	没有免费的提升	参数更多、显存更大、延迟更高，还是实现更复杂
是在同一算力下的比较吗？	不同算力的对比没有意义	是否说明了训练成本与推理成本
有可复现的开源实现吗？	没有复现就没法验证	是否放出代码、权重与数据细节
半年后它还在吗？	筛掉短命的热点	看是否被后续工作引用、被工程界采用

按这套流程读下来你会发现，真正值得花一整个周末去读的新闻，一年也就那么几条。大部分所谓「颠覆」属于两种情况：换了个更省算力的实现方式，或者把若干已有技术组合成了一个好用的工程产品。两种情况都很有价值，但都不需要你推翻已有的知识结构。

> **给新手的建议** 先把自己那一层的东西学扎实，再看新闻。基础不牢的时候追新闻，只会不断推翻自己刚建立的理解，最后什么也留不下。

第 3 章　数据结构：算法的地基

3.0　为什么先学数据结构

数据结构这个词听起来抽象，但它的动机非常朴素：内存是一长排编了号的格子，而我们要处理的数据有各种形状——有的是一串数字，有的是一本通讯录，有的是城市之间的公路网，有的是一棵组织架构树。把这些不同形状的数据按某种规则放进那排格子里，并定义好增、删、改、查的操作方式，这就叫数据结构。

于是每一节都可以用四个问题来读：它长什么样（内存怎么安排）；它能多快完成查找、插入、删除（复杂度）；它适合什么场景；它的代价是什么（什么时候别用它）。

> **提示：**数据结构决定了算法的上限。同样是查找，无序数组只能一个一个比（O(n)），哈希表接近一步到位（均摊 O(1)），平衡树是对数级（O(log n)）。算法写得再巧，也突破不了底层结构允许的极限。

下面这张表贯穿本章，每讲完一个结构都可以回来对照一次。

结构	查找	插入	删除	是否有序	典型用途
静态数组	O(n)	O(n)	O(n)	否	随机访问、矩阵、批量存储
动态数组	O(n)	尾部均摊 O(1)	O(n)	否	Python list、可变序列
单链表	O(n)	O(1)（已知位置）	O(1)（已知前驱）	否	频繁头插、LRU、邻接表
栈	栈顶 O(1)	O(1)	O(1)	否	括号匹配、递归、撤销
队列	队首 O(1)	O(1)	O(1)	否	任务排队、广度优先搜索
哈希表	均摊 O(1)	均摊 O(1)	均摊 O(1)	否	字典、去重、缓存、索引
二叉搜索树	平均 O(log n)	平均 O(log n)	平均 O(log n)	是	有序查找、范围查询
平衡树	O(log n)	O(log n)	O(log n)	是	有序索引、数据库、STL map

结构	查找	插入	删除	是否有序	典型用途
堆	取最值 O(1)	O(log n)	O(log n)	半有序	优先队列、Top-K、Dijkstra
图	取决于存法	取决于存法	取决于存法	否	路径、关系网络、依赖分析
并查集	近似 O(1)	近似 O(1)	不支持删除	否	连通性判断、Kruskal
Trie 字典树	O(词长)	O(词长)	O(词长)	按前缀有序	前缀搜索、自动补全

3.1 数组与动态数组：最基础的容器

数组是最古老也最重要的结构。它把同样类型的数据连续放在内存里，因此有两件事变得极快、极慢：按下标取值是 O(1)（因为地址可以直接算出来：起点地址加上下标乘元素大小），而在中间插入或删除是 O(n)（因为后面的所有元素都要搬一次）。

动态数组解决的是「长度事先不知道」的问题。它的做法是：申请一块容量为 capacity 的连续空间，占用其中的 size 个位置；当 size 追上 capacity 时，申请一块两倍大的新空间，把旧数据整体搬过去。这样单次扩容是 O(n)，但把扩容成本摊到每一次追加上，平均每次追加仍然是常数时间，这就是「均摊 O(1)」的来历。

代码 3-1　手写一个动态数组：扩容、追加、按下标访问

```
class DynArray:
    def __init__(self):
        self._cap = 2          # 当前容量
        self._size = 0         # 已用长度
        self._a = [None] * self._cap

    def append(self, v):
        if self._size == self._cap:       # 满了就翻倍
            self._cap *= 2
            new = [None] * self._cap
            for i in range(self._size):    # 整体搬过去，O(n)
                new[i] = self._a[i]
            self._a = new
        self._a[self._size] = v
        self._size += 1

    def get(self, i):
        if not 0 <= i < self._size:
            raise IndexError(i)            # 边界检查不能省
        return self._a[i]                  # 地址直接算出来，O(1)

    def insert(self, i, v):
        if self._size == self._cap:        # 同样需要先扩容
```

```
            self.append(None)
            self._size -= 1
        for j in range(self._size, i, -1): # 从后往前搬，腾出位置
            self._a[j] = self._a[j - 1]
        self._a[i] = v
        self._size += 1
```

表 3-1 给出了几种常见操作的代价。要特别注意两种「看起来都是 O(1)」的差别：按下标取值是真正的最坏情况 O(1)，而尾部追加是均摊 O(1)——绝大多数情况很快，偶尔一次会很慢。

操作	平均代价	最坏代价	原因
按下标取 a[i]	O(1)	O(1)	地址 = 基址 + i × 元素大小
尾部追加 append	均摊 O(1)	O(n)（触发扩容）	扩容要整体搬移
中间插入 insert	O(n)	O(n)	后面的元素要整体后移
中间删除 pop(i)	O(n)	O(n)	后面的元素要整体前移
查找是否存在	O(n)	O(n)	无序就只好一个个比
在数组上做二分查找	O(log n)	O(log n)	前提是数组已经有序

Python 的 list 内部就是这样一块动态数组，所以 a.append 很快、a.insert(0, x) 很慢、a.pop() 很快、a.pop(0) 很慢。看到「在列表头部反复插入」，就该条件反射地想到换结构。

> **注意**：范围查询是数组的强项。因为数据连续存放，访问 a[i] 之后的 a[i+1] 几乎不额外花钱（CPU 会把相邻数据一起读进高速缓存），这就是为什么在大数据量下，数组常常比链表快得多——即使两者的复杂度写成一样。

3.2　链表：用绳子把盒子串起来

链表放弃了连续存放，改为每个节点存一个值和指向下一个节点的指针。优点是插入删除不需要搬移数据，只要改指针；代价是无法直接算出第 i 个节点的地址，只能从头一个个走。

最实用的两个变体是双向链表（每个节点同时存前驱和后继，删除节点时不需要再找前驱）和循环链表（尾节点指回头节点）。工程代码里几乎总会在头部放一个不存数据的哨兵节点，这样「在头部插入」「删除头部」就不需要写特殊判断。

代码 3-2　单链表：头插、按值删除、原地反转

```
class Node:
    def __init__(self, v):
        self.v = v
        self.next = None

class LinkedList:
```

```
    def __init__(self):
        self.head = None

    def push_front(self, v):        # 头插，O(1)
        n = Node(v)
        n.next = self.head
        self.head = n

    def remove(self, v):            # 删除第一个值为 v 的节点，O(n)
        if self.head is None:
            return False
        if self.head.v == v:        # 删头节点要单独处理（有哨兵就不必）
            self.head = self.head.next
            return True
        p = self.head
        while p.next is not None:
            if p.next.v == v:
                p.next = p.next.next # 把前驱直接接到后继
                return True
            p = p.next
        return False

    def reverse(self):              # 就地反转，经典三指针题，O(n) 时间 O(1) 空间
        prev, cur = None, self.head
        while cur is not None:
            nxt = cur.next          # 先记住下一个，否则链会断
            cur.next = prev         # 掉头
            prev, cur = cur, nxt
        self.head = prev
```

反转这段代码值得逐行读三遍，它包含链表操作的全部套路：先保存后继、再改指向、最后推进指针。新手最常见的错误就是忘了第一句，把后面的节点整段弄丢。

对比项	数组	链表
内存布局	连续	分散，每个节点额外存指针
按下标访问	O(1)	O(n)
已知位置插入删除	O(n)（要搬数据）	O(1)（只改指针）
未知位置查找	O(n)	O(n)
高速缓存友好度	高	低（跳来跳去）
扩容成本	需要整体搬移	不需要，随时申请新节点
典型用途	随机访问、批量计算	频繁插入删除、邻接表、LRU

> **提示：**链表的 O(1) 插入有个前提——你已经站在那个位置上。如果还要先花 O(n) 找到位置，总代价仍然是 O(n)。面试里说「链表插入是 O(1)」时，最好把这个前提一起说出来。

3.3　栈：只能从一头进出的容器

> **提示**：哈希表的内存布局、冲突处理（链地址法与开放寻址）与扩容策略，已在第 3 章 3.5 节详细展开；本节从算法角度给出可直接使用的代码与使用要点。

栈只允许在顶端加入（push）和取出（pop），因此后进先出（LIFO）。它看起来简单得过分，却是编译器、操作系统和浏览器里最常用的结构之一。

为什么这么有用？因为「嵌套」这件事在计算里无处不在：括号是嵌套的，函数调用是嵌套的，页面跳转是可以一层层返回的。凡是需要「记住来时的路」，就需要栈。

应用场景	栈里存什么	一句话说明
括号匹配	未闭合的左括号	遇到右括号就弹出配对
表达式求值	数字与运算符	中缀转后缀再求值
函数调用	每个函数的局部变量与返回地址	递归的本质就是系统帮你推栈
浏览器前进后退	访问过的地址	后退把地址压入另一个栈
撤销操作 Ctrl+Z	每次操作的逆操作	撤销完还能重做
深度优先搜索	待访问的节点	用栈代替递归

代码 3-3　用数组实现栈，并用它检查括号是否匹配

```
class Stack:
    def __init__(self):
        self._a = []

    def push(self, v):
        self._a.append(v)          # 栈顶就是列表末尾，O(1)

    def pop(self):
        if not self._a:
            raise IndexError("栈空")
        return self._a.pop()       # O(1)

    def top(self):
        return self._a[-1] if self._a else None

    def empty(self):
        return len(self._a) == 0

def check(s):
    pairs = {")": "(", "]": "[", "}": "{"}
    st = Stack()
    for ch in s:
        if ch in "([{":
            st.push(ch)                      # 左括号进栈
        elif ch in pairs:
```

```
            if st.empty() or st.pop() != pairs[ch]:
                return False                # 没得配，或者配错了
    return st.empty()                       # 全部配完才算通过

print(check("{[()]}"), check("([)]"))       # True False
```

新手最容易漏掉的一句是最后的 st.empty()。如果字符串是「((((」，循环结束时栈里还有东西，只靠中途判断会误判为合法。

3.4 队列与双端队列：先来先服务

队列只允许在队尾加入、在队首取出，即先进先出（FIFO）。它对应现实里的排队：打印机任务、消息队列、操作系统的进程调度。在算法里它最重要的用途是广度优先搜索（BFS），因为 BFS 要求「先访问离起点近的节点」。

用列表实现队列有个陷阱：从头部删除元素是 O(n)。解决办法是循环队列——用固定数组配合两个下标，让队尾绕回开头，这样入队出队都是真正的 O(1)。

代码 3-4　循环队列：数组 + 头尾下标，入队出队都是 O(1)

```
class RingQueue:
    def __init__(self, cap):
        self._a = [None] * cap
        self._cap = cap
        self._head = 0     # 队首下标
        self._size = 0     # 元素个数

    def push(self, v):
        if self._size == self._cap:
            raise OverflowError("队列已满")
        idx = (self._head + self._size) % self._cap   # 绕回去
        self._a[idx] = v
        self._size += 1

    def pop(self):
        if self._size == 0:
            raise IndexError("队列空")
        v = self._a[self._head]
        self._head = (self._head + 1) % self._cap
        self._size -= 1
        return v
```

把两端都开放，就是双端队列（deque），四端操作都是 O(1)，Python 里对应 collections.deque。它有一个非常实用的技巧叫单调队列：在滑动窗口问题中，让队列里的元素保持单调递减，队首永远是当前窗口的最大值，于是把 O(n × k) 的暴力做法降到 O(n)。

代码 3-5　单调队列：滑动窗口最大值，O(n)

```
from collections import deque
```

```
def max_sliding_window(a, k):
    q = deque()          # 存下标，对应的值单调递减
    out = []
    for i, v in enumerate(a):
        while q and a[q[-1]] <= v:
            q.pop()                    # 后面来的更大，前面的就没用了
        q.append(i)
        if q[0] <= i - k:
            q.popleft()                # 队首滑出窗口
        if i >= k - 1:
            out.append(a[q[0]])        # 队首就是窗口最大值
    return out

print(max_sliding_window([1, 3, -1, -3, 5, 3, 6, 7], 3))   # [3, 3, 5, 5, 6, 7]
```

结构	进出规则	取最值	典型用途
队列	先进先出	O(n)	BFS、任务调度、消息队列
循环队列	先进先出，数组复用	O(n)	定长缓冲区、生产者消费者
双端队列	两端都可进出	O(n)	滑动窗口、回文判断
单调队列	队内保持单调	队首 O(1)	滑动窗口最值、优化动态规划
优先队列（堆）	按优先级出队	队首 O(1)	调度、Dijkstra、Top-K

配套仿真：在「算法动画仿真实验室」里搜索「单调队列」，左侧可以看着窗口一格一格往右滑、队头跟着换人，右侧日志把每一次「弹出队尾」和「弹出队首」都记下来。它和 3.13、3.14 要讲的平衡树、跳表放在一起看，走的是同一条思路：额外维护一层有序信息，用空间换速度。

3.5　哈希表：用算出来的位置直接找到

哈希表的核心思想只有一句话：把键交给哈希函数算出一个整数，再对这个整数取余，得到它该放的下标，于是查找不用比较、直接跳过去。这就是字典、集合、缓存能这么快的原因。

麻烦在于冲突：不同的键可能算出同一个下标。工业界有两条主流路线，理解它们就理解了哈希表的全部：

冲突处理方式	做法	优点	缺点
链地址法	每个槽挂一条链表（或红黑树）	实现简单、删除容易、容许高负载	有指针开销、缓存不友好
开放寻址法	冲突就按规则往后找空位	内存连续、缓存友好	删除要用墓碑标记、负载不能太高

另一个关键概念是负载因子（已存元素数除以槽位数）。负载因子越高冲突越多，查找越慢。所以哈希表会在负载超过阈值（常见 0.75）时扩容：申请两倍槽位，把所有元素重新哈希放一遍。这一次是 O(n)，但摊到每次插入上仍然是均摊 O(1)。

代码 3-6　手写哈希表：链地址法 + 负载因子触发扩容

```
class HashMap:
    def __init__(self, cap=8):
        self._cap = cap
        self._size = 0
        self._buckets = [[] for _ in range(cap)]

    def _idx(self, key):
        return hash(key) % self._cap          # 取余得到槽位

    def put(self, k, v):
        b = self._buckets[self._idx(k)]
        for i, (kk, _) in enumerate(b):
            if kk == k:
                b[i] = (k, v)                 # 键已存在就覆盖
                return
        b.append((k, v))
        self._size += 1
        if self._size / self._cap > 0.75:
            self._resize()                    # 太挤就扩容

    def get(self, k, default=None):
        for kk, vv in self._buckets[self._idx(k)]:
            if kk == k:
                return vv
        return default

    def _resize(self):
        old = [kv for b in self._buckets for kv in b]
        self._cap *= 2
        self._buckets = [[] for _ in range(self._cap)]
        self._size = 0
        for k, v in old:                      # 全部重新哈希
            self.put(k, v)
```

> **提示：**哈希表的 O(1) 是均摊与平均意义下的。如果哈希函数设计得差（例如所有键都落在同一个槽），它会退化成链表，最坏情况 O(n)。这也是为什么工程上要防止外部输入能控制哈希值（哈希洪水攻击）。

对比	哈希表	平衡树	有序数组
查找	均摊 O(1)	O(log n)	O(log n)
插入删除	均摊 O(1)	O(log n)	O(n)

对比	哈希表	平衡树	有序数组
是否有序	无序	有序，可范围查询	有序
取最值	O(n)	O(log n)	O(1)
最坏情况	O(n)（哈希退化）	O(log n)	O(log n)
典型实现	Python dict、Redis Hash	C++ map、Java TreeMap	二分查找

在数据库里这两个思路也各有位置：等值查询多的场景用哈希索引，范围查询和排序多的场景用 B+ 树索引。选错索引类型，SQL 慢十倍都不奇怪。

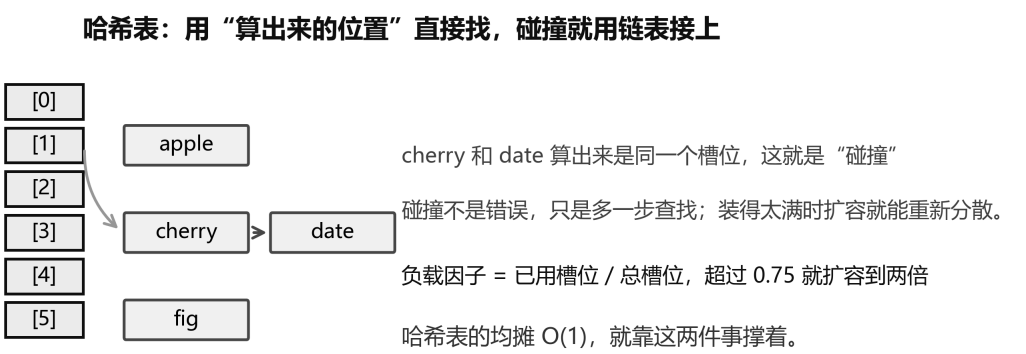


图 3-1　哈希表的两件事：用算出来的位置直接找，碰撞就用链表接上

3.6　树：处理一对多的关系

到此为止的结构都是一条线（数组、链表、栈、队列），而现实中的关系常常是分叉的：文件夹可以装文件夹，生物分类可以再细分，网页能链到多个页面。描述这种「一对多」的结构就是树。

先把术语一次讲清，后面所有内容都建立在这几个词上：

术语	含义	一句话记忆
根节点	唯一没有父节点的节点	树的入口
叶节点	没有子节点的节点	末端
父节点与子节点	直接相连的上下层关系	上下相邻才算
兄弟节点	同一个父节点的节点	同父
度	一个节点的子节点个数	分了几叉
深度	从根到该节点的边数	从上往下数
高度	从该节点到最深叶子的边数	从下往上数
子树	以某节点为根的整棵小树	树中的树

二叉树是最常用的特例：每个节点最多两个孩子，习惯上称为左孩子和右孩子。它有三种深度优先遍历方式，区别只在「什么时候处理自己」：

遍历	顺序	典型用途
前序	自己 → 左 → 右	复制整棵树、序列化
中序	左 → 自己 → 右	二叉搜索树得到有序序列
后序	左 → 右 → 自己	计算子树大小、释放内存
层序	按层从左到右（用队列）	求深度、按层输出

代码 3-7　二叉树：三种递归遍历 + 用栈实现的迭代中序

```
class TreeNode:
    def __init__(self, v):
        self.v = v
        self.left = None
        self.right = None

def preorder(t, out):
    if t is None:
        return
    out.append(t.v)                 # 先处理自己
    preorder(t.left, out)
    preorder(t.right, out)

def inorder(t, out):
    if t is None:
        return
    inorder(t.left, out)
    out.append(t.v)                 # 中间处理自己
    inorder(t.right, out)

def postorder(t, out):
    if t is None:
        return
    postorder(t.left, out)
    postorder(t.right, out)
    out.append(t.v)                 # 最后处理自己

def inorder_iter(t):                # 递归改迭代：显式用一个栈
    st, out, cur = [], [], t
    while cur is not None or st:
        while cur is not None:      # 一路向左，把沿途节点压栈
            st.append(cur)
            cur = cur.left
        cur = st.pop()              # 弹出来处理
        out.append(cur.v)
        cur = cur.right             # 再转向右子树
```

```
    return out
```

二叉搜索树（BST）在二叉树之上加一条规则：左子树的所有值都比自己小，右子树的所有值都比自己大。于是查找可以像二分一样一路排除一半。

代码 3-8　二叉搜索树：查找与插入

```
def bst_find(t, x):
    while t is not None:
        if x == t.v:
            return t
        t = t.left if x < t.v else t.right   # 只走一边
    return None

def bst_insert(t, x):
    if t is None:
        return TreeNode(x)
    if x < t.v:
        t.left = bst_insert(t.left, x)
    elif x > t.v:
        t.right = bst_insert(t.right, x)
    return t                                  # 相等就不重复插入
```

> **注意：**BST 的性能完全取决于它的形状。如果按 1、2、3、4、5 的顺序插入，它会退化成一条向右的链，查找变成 O(n)——和链表没有区别。解决办法是让它始终保持平衡，这就是下面要讲的平衡树。

平衡树并不要求左右子树高度完全相等，而是要求高度差不超过一个常数，靠旋转操作在插入删除后把树「扶正」。工程上最常遇到的是两种：

平衡树	平衡方式	查找/插入/删除	常见实现
AVL 树	严格，左右高度差不超过 1	O(log n)	教学、读多写少
红黑树	宽松，用颜色约束近似平衡	O(log n)	C++ map/set、Java TreeMap、Linux 内核
B 树 / B+ 树	一个节点存多个键，树更矮更胖	O(log n)	数据库索引、文件系统
跳表	多层链表，用随机化实现平衡	期望 O(log n)	Redis zset、LevelDB

为什么数据库用 B+ 树而不是红黑树？因为磁盘每次读取一整块（页），B+ 树把几百个键放在一个节点里，一次 IO 就能排除大量数据，树高通常只有三到四层，也就意味着三四次磁盘访问就能定位一行记录。这是「结构适配硬件」的典型案例。

3.7　堆与优先队列：随时知道谁最大

很多任务不需要全局有序，只需要随时能拿到当前的最大值或最小值：任务调度器要挑优先级最高的任务，Dijkstra 要挑当前距离最小的节点，Top-K 问题要不断扔掉最小的那个。为这种需求量身定做的结构就是堆。

堆是一棵完全二叉树，通常直接用数组存储（下标 i 的孩子是 2i+1 和 2i+2，父节点是 (i-1)/2），并满足堆序性质：小顶堆里每个父节点都不大于它的孩子。它只保证「根是最小值」，不保证整体有序——这正是它便宜的原因。

代码 3-9　手写小顶堆：上浮、下沉、建堆、取最小值

```
class MinHeap:
    def __init__(self, a=None):
        self.a = list(a) if a else []
        for i in range(len(self.a) // 2 - 1, -1, -1):
            self._sink(i)                    # 从最后一个非叶节点往前下沉

    def push(self, v):
        self.a.append(v)
        i = len(self.a) - 1
        while i > 0:
            p = (i - 1) // 2
            if self.a[p] <= self.a[i]:
                break
            self.a[p], self.a[i] = self.a[i], self.a[p]   # 上浮
            i = p

    def pop(self):
        if not self.a:
            raise IndexError("堆空")
        top, last = self.a[0], self.a.pop()
        if self.a:
            self.a[0] = last
            self._sink(0)                    # 把新的根沉下去
        return top

    def _sink(self, i):
        n = len(self.a)
        while True:
            l, r, small = 2 * i + 1, 2 * i + 2, i
            if l < n and self.a[l] < self.a[small]:
                small = l
            if r < n and self.a[r] < self.a[small]:
                small = r
            if small == i:
                return
            self.a[i], self.a[small] = self.a[small], self.a[i]
            i = small

    def peek(self):
```

```
        return self.a[0] if self.a else None   # 看一眼最小值，O(1)
```

操作	代价	说明
取堆顶（最值）	O(1)	数组第一个元素
插入	O(log n)	放到末尾后上浮
删除堆顶	O(log n)	末尾元素补到根后下沉
把无序数组建成堆	O(n)	从最后一个非叶节点往前逐个下沉
堆排序	O(n log n)	建堆后反复取堆顶，原地完成
查找任意元素	O(n)	堆不提供这个能力

> **提示**：建堆是 O(n) 而不是 O(n log n)，这是很多人第一次听到会觉得反直觉的结论。直觉解释是：越靠近底层的节点越多，但它们下沉的距离越短；把所有节点的下沉距离加起来是个收敛的级数，结果是线性的。

堆的两个经典应用值得记住。第一个是 Top-K：找出一亿个数里最大的 100 个，不需要排序，只要维护一个大小为 100 的小顶堆，新数比堆顶大就替换，时间复杂度 O(n log k)，空间只要 O(k)。第二个是优先队列，也就是 Dijkstra、A*、哈夫曼编码、任务调度共同的基础设施。

3.8　图：描述关系的数据结构

树只能表达层级关系，而现实中的关系往往是网状的：好友之间互相认识，城市之间互相通航，网页之间互相链接，函数之间互相调用。描述这种「多对多」关系的结构就是图。图由顶点和边组成；边有方向就是有向图，边带数字就是带权图，任意两点之间都有边就是完全图。

图的存储方式只有两种主流选择，理解它们的取舍比背代码更重要：

存储方式	空间	判断两点是否相连	遍历某点的所有邻居	适用场景
邻接矩阵	$O(V^2)$	O(1)	O(V)	稠密图、点少边多、需要频繁判断连接
邻接表	O(V+E)	O(度)	O(度)	稀疏图、绝大多数真实网络
边集数组	O(E)	O(E)	O(E)	只需要枚举边，如 Kruskal 算法

先想清楚存什么　图就是「点 + 边」，存图的本质是回答一个问题：给我一个点，我怎么立刻找到它连出去的所有边。

有两种主流存法。邻接矩阵开一个 V×V 的表，行是起点、列是终点，一格表示「这两点之间有没有边」。好处是查「A 到 B 有没有边」一步到位；坏处是点一多，表就大到存不下——一万个点就是一万乘一万个格子，而其中绝大多数是空的。

邻接表换个思路：给每个点配一个小清单，清单上只写它真正连出去的邻居。稀疏图（边远少于点对总数）时它又省地方又快，工程上绝大多数图都用它。下面这段代码就是邻接表，并且顺手做了两种遍历。

- 广搜（BFS）用队列，一层一层往外推，天然能算出「最少几步到达」，题里要最短步数就选它。
- 深搜（DFS）用递归或栈，一条道走到底再回头，适合判连通、找环、做拓扑，写法更短。
- 两者都必须在入队/入栈时立刻标记已访问，否则同一个点会被反复塞进去。

读代码看这里　visited 数组是干嘛的、标记发生在什么时刻、队列换成栈以后结果为什么还是一样。

代码 3-10　用邻接表建图，并做广度优先与深度优先遍历

```
from collections import deque

def build(edges, directed=False):
    g = {}
    for u, v in edges:
        g.setdefault(u, []).append(v)
        if not directed:
            g.setdefault(v, []).append(u)     # 无向图要加反向边
    return g

def bfs(g, s):
    seen, order, q = {s}, [], deque([s])
    while q:
        u = q.popleft()                       # 队列：先来先访问
        order.append(u)
        for v in g.get(u, []):
            if v not in seen:
                seen.add(v)                   # 入队时就标记，避免重复入队
                q.append(v)
    return order

def dfs(g, s):
    seen, order, st = set(), [], [s]
    while st:
        u = st.pop()                          # 栈：后进先出
        if u in seen:
            continue
```

```
        seen.add(u)
        order.append(u)
        for v in g.get(u, []):
            if v not in seen:
                st.append(v)
    return order
```

> **注意**：BFS 中标记「已访问」的时机必须在入队时，而不是出队时。否则同一个节点可能被重复入队，队列会膨胀，在稠密图上甚至可能指数级放大。

BFS 与 DFS 是图算法的两大基础动作，后面几章的最短路、连通分量、环检测都建立在它们之上：

遍历	数据结构	访问顺序	天然解决的问题
BFS	队列	按离起点的层数由近及远	无权图最短路、连通分量、染色判断二分图
DFS	栈或递归	一条路走到黑再回头	环检测、拓扑排序、割点桥、回溯搜索

在一个有向无环图（DAG）上，我们还能做拓扑排序，得到一个「所有任务的前置依赖都排在前面」的线性顺序。它的用途非常实际：编译器的头文件依赖、项目管理里的任务排序、机器学习框架的计算图调度、Excel 的单元格重算顺序，用的都是它。实现方式有两种：一是 DFS 完成后逆序输出，二是不断取出入度为 0 的节点（Kahn 算法）。后者还有一个附带好处——如果最后输出的节点数少于总数，说明图中存在环。

3.9　并查集：判断「是不是一伙的」

并查集（Union-Find）解决一类很窄但极常见的问题：不断把两个元素合并成一组，并且随时回答「这两个元素现在是不是同一组」。比如判断社交网络里两个人是否连通、Kruskal 算法判断加一条边会不会成环、图像处理里的连通区域标记。

它的实现出奇地短：每个元素记录自己的父节点，同一组的元素最终都指到同一个根。配合两个优化——路径压缩（查找时顺便把沿途节点直接挂到根上）和按秩合并（矮树挂到高树上）——单次操作的代价接近常数，实际可以认为就是 O(1)。

代码 3-11　并查集：路径压缩 + 按大小合并

```
class DSU:
    def __init__(self, n):
        self.p = list(range(n))    # 每个元素一开始自成一派
        self.sz = [1] * n          # 集合大小，用于按大小合并

    def find(self, x):
        root = x
        while self.p[root] != root:
            root = self.p[root]
        while self.p[x] != root:   # 路径压缩：把沿途节点直接挂到根上
```

```
            self.p[x], x = root, self.p[x]
        return root

    def union(self, a, b):
        ra, rb = self.find(a), self.find(b)
        if ra == rb:
            return False           # 本来就在一起
        if self.sz[ra] < self.sz[rb]:
            ra, rb = rb, ra        # 小的挂到大的下面
        self.p[rb] = ra
        self.sz[ra] += self.sz[rb]
        return True

    def same(self, a, b):
        return self.find(a) == self.find(b)
```

> **提示：**并查集不支持「拆开」一组元素，也不支持删除。如果业务需要撤销合并，就要换成可持久化并查集或改用图论算法，这一点在选型时要想清楚。

3.10 Trie 字典树：按前缀找词

Trie（也叫前缀树）把字符串按字符一层层挂到树上：根节点代表空串，每条边是一个字符，从根到一个节点的路径就是一个前缀。它的三个操作插入、查询、前缀匹配的代价都是 O(词长)，与词典里有多少词无关——这正是搜索引擎补全框、输入法候选词、敏感词过滤能做到毫秒级响应的原因。

代码 3-12　Trie：插入、完整匹配、前缀匹配

```
class TrieNode:
    def __init__(self):
        self.child = {}
        self.end = False        # 是否是一个完整单词的结尾

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, w):
        node = self.root
        for ch in w:
            node = node.child.setdefault(ch, TrieNode())
        node.end = True

    def search(self, w):
        node = self._walk(w)
        return node is not None and node.end

    def starts_with(self, pre):
        return self._walk(pre) is not None   # 只要前缀走得通就算命中
```

```
    def _walk(self, s):
        node = self.root
        for ch in s:
            if ch not in node.child:
                return None
            node = node.child[ch]
        return node
```

结构	查找代价	优势	代价
哈希表	均摊 O(1)	单点查询最快	不支持前缀、不支持有序遍历
Trie	O(词长)	前缀查询、按字典序输出	空间开销大（每个节点一个字典）
平衡树	O(词长 × log n)	支持范围查询	前缀查询要做两次二分

3.11　四个值得知道的进阶结构

下面四个结构不要求会手写，但要知道它们解决什么问题，遇到对应场景时能立刻想起名字。

一是跳表。它给有序链表加上多层索引，像查字典先看目录再翻页，期望复杂度 O(log n)。相比红黑树，它的实现简单得多，而且天然支持范围查询，Redis 的有序集合（zset）就选了它。

二是 LRU 缓存。它的要求是「容量满时淘汰最久未使用的那个」，同时要求查询和更新都是 O(1)。单独一个哈希表或单独一个双向链表都做不到，正确答案是两个一起用：哈希表负责 O(1) 找到节点，双向链表负责 O(1) 把节点移动到「最近使用」的一端。这个「两个结构配合」的思路在工程里反复出现。

三是位图。如果数据范围有限且只需要判断「在不在」，一个比特就能表示一个数，一亿个布尔值只需要 12.5 MB。它的经典应用是统计活跃用户、去重和位运算加速。

四是布隆过滤器。它用位图加多个哈希函数，用很小的空间判断「一个元素一定不在集合里，或在集合里」。注意它的两个特性：只会误判为存在，不会误判为不存在；且不支持删除。它的经典应用是缓存穿透防护、爬虫 URL 去重、数据库避免查不存在的键。

结构	核心思想	关键代价	典型场景
跳表	有序链表 + 多层随机索引	期望 O(log n)，空间约 2 倍	Redis zset、内存有序索引
LRU 缓存	哈希表 + 双向链表	O(1) 查询与淘汰	页面缓存、数据库缓冲池
位图	一个比特表示一个状态	O(1)，空间极省	去重、活跃统计、布隆过滤器底座

结构	核心思想	关键代价	典型场景
布隆过滤器	位图 + 多个哈希函数	空间极小，有误判率	反缓存穿透、URL 去重

3.12 怎么选：一张速查表与八个常见错误

选数据结构没有标准答案，但有一套稳定的推理顺序：先问数据量有多大，再问最频繁的操作是什么，然后问是否需要有序或范围查询，最后才问内存是否紧张。

如果你最频繁要做的事是	首选结构	一句话理由
按编号快速取第 i 个元素	数组 / 动态数组	地址可直接计算
在两端频繁增删	双端队列	四端操作都是 O(1)
在已知位置频繁插入删除	链表	只改指针不搬数据
判断某个键是否存在、取值	哈希表	均摊 O(1)
维护一个动态集合且要取最值	堆	取最值 O(1)
维护有序数据并做范围查询	平衡树 / B+ 树	对数级且有序
找前缀、自动补全	Trie	与词数无关，只看词长
判断连通、合并集合	并查集	近似 O(1)
表示对象之间的关系	图（邻接表）	稀疏图最省空间
需要淘汰最久未用	哈希表 + 双向链表	两个结构各管一半

新手在这一章最容易犯的八个错误：

一是用列表当队列，在头部反复 pop(0)，把 O(n) 的操作放进循环，程序在大数据上慢得莫名其妙。应该用 deque。

二是以为哈希表永远 O(1)，忽略了最坏情况与哈希函数质量，在需要稳定延迟的系统中没有做兜底。

三是在遍历列表的同时删除元素，导致下标错位、漏掉元素。正确做法是倒序遍历，或先收集再统一删除。

四是用可变对象（如列表）当字典的键，或者自定义对象忘了实现哈希与相等判断，导致「明明一样的键却查不到」。

五是链表操作忘记先保存后继指针，把后面的整条链弄丢。

六是把二叉搜索树当有序容器用，却按有序数据插入，退化成链表；有序场景应当使用平衡树或直接二分。

七是递归深度过大导致栈溢出。Python 默认递归深度约一千层，深层递归应改成显式栈。

八是过早优化：数据量只有几百条时纠结选哪种结构，收益远不如把代码写清楚。结构选型要在有真实数据之后再做。

最后留三道练习，答案都能用本章的代码改出来。第一题：用两个栈实现一个队列，要求均摊 O(1)。第二题：给定一个数组和一个目标值，判断是否存在连续子数组之和等于目标值（提示：哈希表 + 前缀和）。第三题：给定若干个单词，找出所有互为字母异位词的分组（提示：把排序后的单词当作哈希表的键）。

3.13　平衡树：让二叉搜索树不再退化成链表

3.6 里讲过二叉搜索树(BST, Binary Search Tree)：左边的都比自己小，右边的都比自己大，所以查找时每一步都能丢掉一半，代价 O(log n)。这条结论有一个前提：树大体是均匀的。前提不成立，结论就没了。

试试按 1、2、3、4、5 的顺序插入。每个新值都比前一个大，于是每次只能走右孩子，树长成一条向右倾斜的链。这时它和单链表没有区别，找 5 要走 5 步，插入 n 个有序数据的总代价是 $O(n^2)$。而有序数据恰恰是最常见的输入，时间戳、自增主键、按字典序排列的名字，几乎都是递增的。

修复的思路是给树加一条纪律：任何一个结点，它的左右子树高度之差不能超过 1。这个差值叫平衡因子(balance factor)。一旦插入或删除破坏了这条纪律，就地做一次局部调整把它改回来，这种调整叫旋转(rotation)。把它系统化并写成论文的是两位苏联数学家 Adelson-Velsky 和 Landis（1962），所以这类结构的第一种就叫 AVL 树。

旋转为什么管用？失衡只可能发生在「插入点往上到根」的这一条路径上，而且失衡的形状只有四种，每种对应一套固定的指针操作。名字里两个字母的顺序，就是「先动哪边的孙子」。

情况	形状	处理	修复后
LL	左孩子的左子树太高	对失衡结点右旋一次	高度降 1
RR	右孩子的右子树太高	对失衡结点左旋一次	高度降 1
LR	左孩子的右子树太高	先对左孩子左旋，再对失衡结点右旋	高度降 1
RL	右孩子的左子树太高	先对右孩子右旋，再对失衡结点左旋	高度降 1

LL 和 RR 一次搞定，LR 和 RL 需要两次：第一次的目的只是先把它掰成 LL 或 RR 的形状，第二次才是真正的修复。

旋转本身只改三四个指针，代价 O(1)。插入之后从插入点往上回溯到根，需要检查的结点最多 O(log n) 个（因为树高是 O(log n)），所以插入总代价仍是 O(log n)，只是常数比普通 BST 大。删除麻烦一些：删掉一个结点可能让它的祖先一处接一处地失衡，最多要旋转 O(log n) 次。

代码 3-13　AVL 树：高度维护与四种旋转

```
class Node:
    __slots__ = ("v", "l", "r", "h")
    def __init__(self, v):
```

```
        self.v, self.l, self.r, self.h = v, None, None, 1

def h(t):  return t.h if t else 0
def bf(t): return h(t.l) - h(t.r) if t else 0   # 平衡因子，左高为正

def rot_right(y):            # LL：以 y 为轴右旋
    x = y.l
    y.l = x.r
    x.r = y
    y.h = 1 + max(h(y.l), h(y.r))   # 先更新下面的 y
    x.h = 1 + max(h(x.l), h(x.r))   # 再更新上面的 x
    return x                        # 新的子根必须回传给父结点

def rot_left(x):             # RR：以 x 为轴左旋
    y = x.r
    x.r = y.l
    y.l = x
    x.h = 1 + max(h(x.l), h(x.r))
    y.h = 1 + max(h(y.l), h(y.r))
    return y

def insert(t, v):
    if t is None:
        return Node(v)
    if v < t.v:      t.l = insert(t.l, v)
    elif v > t.v:    t.r = insert(t.r, v)
    else:            return t          # 相等不重复插入
    t.h = 1 + max(h(t.l), h(t.r))
    b = bf(t)
    if b > 1  and v < t.l.v:  return rot_right(t)     # LL
    if b < -1 and v > t.r.v:  return rot_left(t)      # RR
    if b > 1  and v > t.l.v:                          # LR
        t.l = rot_left(t.l)
        return rot_right(t)
    if b < -1 and v < t.r.v:                          # RL
        t.r = rot_right(t.r)
        return rot_left(t)
    return t
```

代价与取舍：每个结点要多存一个高度（或平衡因子）字段，删除逻辑比插入复杂，代码读起来也不如普通 BST 直观。所以工程上如果只是要一个有序容器，优先用语言自带的库。Java 的 TreeMap、C++ 的 std::map 用的都是红黑树。红黑树放宽了平衡条件：它允许最长路径不超过最短路径的两倍，换来更少的旋转次数，因此写操作更便宜。手写 AVL 的价值在于理解「旋转到底在改什么」，这一点在排查性能问题时会直接用上。

> **常见错误**　一是旋转之后忘记更新高度，导致下一次判断失衡的依据全是错的，错误会一步步累积。二是先动手再判断类型：还没确认是 LL 就直接左旋，越修越乱。三是平衡因子的定义（右减左还是左减右）前后不一致。四是递归实现里忘了把旋转后的新根回传给父结点，树被接成了两半。

配套仿真：在「算法动画仿真实验室」里搜索「AVL 平衡树」，可以看到一串递增的键被逐个插入时，树怎样先向右歪、再被一次旋转掰回来，右侧表格同步显示每个结点的值、左右孩子、高度和平衡因子。把它和「二叉搜索树 BST」那个仿真放在一起看，同一组数据、两条不同的命运，退化有多快一目了然。

3.14　跳表：在链表上架起索引层

3.11 里把跳表列进了四个进阶结构之一，这里把它讲透，因为它是「用一点随机性换掉大量实现复杂度」这个思路最干净的样本。

麻烦出在链表身上：插入和删除都是 O(1)，但查找必须从头一个一个走，代价 O(n)。数组能靠下标一步跳到中间，链表不能，它没有「中间」这个概念，只有「下一个」。

跳表的做法是在链表之上再建几层「快车道」。最底层是完整的原始链表；每往上一层，大约隔一个结点保留一个，结点数减半；最上面几层稀疏到只剩几个结点。查找时从最上层出发：右边结点比目标小就往右走，比目标大就往下一层，再继续往右。每一层的平均步数很少，而层数约为 $\log_2 n$，所以总步数与 log n 成正比。

真正关键的问题是「哪个结点该上到第几层」。如果按固定规则（比如每两个抽一个）来，那么每次插入删除都要连带调整整条链，代价又回到 O(n)。跳表的答案是抛硬币：新结点先放进最底层，然后以 1/2 的概率再往上一层，继续掷，直到掷出反面为止。这样插入和删除都只需要改掉自己牵涉到的几个指针，完全不用动别人的层数。从概率上看，第 k 层的结点数约为 n / 2^k，所有索引层的结点总数不到 n，额外空间是 O(n) 级别，平均每个结点多约两个指针。

它的复杂度是「期望 O(log n)」而不是「最坏 O(log n)」。理论上存在极小概率长成很差的形状，但只要随机数是真的随机，实际工程中几乎不会出事，这也是它被广泛接受的原因。

对比项	跳表	平衡树（AVL / 红黑树）
实现难度	低，核心只有查找、插入、删除三件事	高，涉及四种旋转与删除后的连锁调整
查找复杂度	期望 O(log n)	最坏 O(log n)
范围查询	定位起点后顺着底层链表往后走即可	需要中序遍历，写法更绕
空间开销	每层多一个指针，期望约 2 个指针 / 结点	每个结点多 1 个高度或颜色字段
并发友好	改动局部，加锁范围小	旋转牵动整棵树，加锁范围大
典型应用	Redis 有序集合 zset、LevelDB 的内存表	Java TreeMap、C++ std::map、B+ 树索引

代码 3-14　跳表：随机层高 + 查找 + 插入

```
import random
```

```
class Node:
    __slots__ = ("v", "nxt")
    def __init__(self, v, level):
        self.v = v
        self.nxt = [None] * level      # 每个结点持有它在各层的前驱

class SkipList:
    def __init__(self, p=0.5, maxlv=16):
        self.p, self.maxlv = p, maxlv
        self.head = Node(None, maxlv)
        self.lv = 1

    def _rand_level(self):
        lv = 1
        while random.random() < self.p and lv < self.maxlv:
            lv += 1
        return lv

    def find(self, target):
        cur = self.head
        for i in range(self.lv - 1, -1, -1):     # 从最上层往下
            while cur.nxt[i] and cur.nxt[i].v < target:
                cur = cur.nxt[i]                 # 能往右就往右
        return cur.nxt[0]                        # 底层的下一个就是答案

    def insert(self, v):
        update = [None] * self.maxlv
        cur = self.head
        for i in range(self.lv - 1, -1, -1):
            while cur.nxt[i] and cur.nxt[i].v < v:
                cur = cur.nxt[i]
            update[i] = cur
        lv = self._rand_level()
        if lv > self.lv:                         # 新层从 head 起接
            for i in range(self.lv, lv):
                update[i] = self.head
            self.lv = lv
        node = Node(v, lv)
        for i in range(lv):                      # 逐层接指针
            node.nxt[i] = update[i].nxt[i]
            update[i].nxt[i] = node
```

容易踩的坑　一是把层高当成固定值，按固定规则抽结点，结果索引层跟着数据一起退化成链表。二是随机数用了固定种子，或者多线程共用一个劣质生成器，丢掉了「期望」这个前提。三是查找时只往下不往右（变成逐层扫描），或者只往右不往下（直接漏掉目标）。四是插入时新层数超过当前层数，却忘了把高层的前驱设成头结点，导致高层指针悬空。

配套仿真：搜索「跳表」，可以看到同一个目标怎样在四层索引上被逐层逼近，右侧日志把每一次「往右」和「下探」都打出来。它和 3.3 节里讲的单调队列摆在一起看，会更容易体会到一件事：绝大多数「把 O(n) 降到 O(log n)」的技巧，本质上都是额外维护了一层有序信息。

3.15　线段树：把「区间求和」从 O(n) 压到 O(log n)

前面几节的结构解决的都是「一个点」的问题：找某一个元素、往某个位置插入。但有一类问题问的是「一段」：第 3 号到第 97 号货架上一共有多少件货？过去 7 天的订单总额是多少？从第 12 分钟到第 40 分钟的最大访问量是多少？这类问题叫做区间查询。

最直接的做法是每次把区间里的数挨个加起来。如果每天要回答一万次查询、订单有一百万条，那就需要一百亿次加法，任何机器都承不住。线段树的思路是：与其每次重新数，不如提前把「某一段的总数」记在小本子上。

打一个比方。仓库有 8 排货架，编号 1 到 8。你准备一本账：第 1 页记「第 1 到 8 排的总数」；第 2、3 页分别记「第 1 到 4 排」「第 5 到 8 排」；再往下记「第 1 到 2 排」「第 3 到 4 排」……最后一层每页只记一排。每一页覆盖的范围都是上一页的一半。

这样一来，查「第 3 排到第 7 排」就不需要数 5 次，只要把几页账拼起来：「第 3 到 4 排」加「第 5 到 6 排」加「第 7 排」，3 次加法就够了。8 排是这样，一百万个元素时也只需要大约 40 次。这就是 O(log n) —— 数据翻一倍，耗时只多一步。

这就是线段树（Segment Tree）。它是一棵完全二叉树，每个节点存一段区间的汇总值，父节点的值由两个孩子合并得到。因为是完全二叉树，不需要真的用指针，直接开一个长度 4n 的数组，节点 i 的孩子是 2i 和 2i+1，父子关系用下标算出来就行。

合并方式可以是求和、求最大值、求最小值、最大公约数……只要这个「汇总」满足结合律（先合并前两个再合并第三个，和先合并后两个再合并第一个，结果一样），线段树就能用。

3.15.1　三个基本操作

建树 Build：从叶子往上填，时间复杂度 O(n)。查询 Query：从根开始，如果当前节点的区间完全落在要问的区间里，就把它的值拿回来；如果完全不重叠，就返回「无贡献」（求和返回 0，求最大返回 −∞）；如果部分重叠，就分别问左右孩子再合并。更新 Update：把某个叶子改掉，然后沿着它到根的路径重新合并一遍。三个操作都是 O(log n)。

用不用线段树，其实只看一件事：你要回答的问题能不能「拼」。能拼（求和、求最值、求最大公约数），就能用；不能拼（求第 k 小、求中位数），得换别的结构。

3.15.2　懒标记：区间修改的省事办法

如果要求「把第 3 到第 97 排的货全部加 10 件」，逐个更新是 O(n log n)，太慢。这时用懒标记（Lazy Tag）。

打个比方：装修队要给一栋楼的第 3 层到第 9 层统一刷白。他们不会现在就一间一间刷，而是先在楼梯口贴一张条子：「3 到 9 层，加一层白漆」。等以后真的有人要进某一

间房查看时，再顺着条子把漆刷下去。这张条子就是懒标记：改动先记账，不落地；只有被访问到时才把账推给孩子。

懒标记把区间修改也压到了 O(log n)。代价是代码要仔细，最常见的手写 bug 就是「把标记推给孩子后忘了清空自己」——结果同一段区间被反复加了几遍。

表 3-2　区间查询几种做法的对比

方案	建表	单点修改	区间查询	区间修改	空间	适用场景
朴素数组	O(n)	O(1)	O(n)	O(n)	n	查询很少、一次性场景
前缀和	O(n)	O(n)	O(1)	O(n)	n	只查不改，或只在末尾追加
树状数组	O(n)	O(log n)	O(log n)	差分可做	n	前缀和、前缀最值、逆序对
线段树	O(n)	O(log n)	O(log n)	O(log n)	4n	一切可合并的区间问题

它要解决什么　数组上频繁改一段区间、又频繁问一段区间的和，两个操作都得快。

朴素做法每次查询都要把区间里的数加一遍，O(n)；改一段也是一样。线段树把「区间」画成一棵二叉树：每个节点负责一段，父亲的答案由两个孩子拼出来。这样一来，任何一段区间都能拆成树上的几个节点，查询和修改都只要 O(log n)。

上一节讲的懒标记，是给「一整段统一加一个数」这种操作准备的。它的做法是：能整段交给某个节点处理，就先在这个节点上记一笔「这里每层都欠我 k」，先不动孩子；等哪天真的需要进到孩子里，再把记号推下去。这就是那面「3 到 9 层加一层白漆」的条子。

- 第一步：建树。叶子放原值，父亲等于两个孩子之和。
- 第二步：查询。若当前区间被目标区间完全覆盖，直接返回节点值；否则递归两个孩子在目标区间里的部分。
- 第三步：区间修改。完全覆盖就记懒标记并更新节点值；部分覆盖就先下推标记，再递归孩子。

最容易写错的一处　把标记推给孩子之后，必须清空自己的标记。忘了清，同一段区间会被反复加好几遍。

代码 3-15　线段树：建树、区间求和与区间加（懒标记）

```
def build(a, tree, node, l, r):
    if l == r:                      # 叶子：直接存原值
        tree[node] = a[l]
        return
```

```
    mid = (l + r) // 2
    build(a, tree, 2 * node, l, mid)
    build(a, tree, 2 * node + 1, mid + 1, r)
    tree[node] = tree[2 * node] + tree[2 * node + 1]
def push_down(tree, lazy, node, l, r):
    if lazy[node] == 0 or l == r:
        return
    mid = (l + r) // 2
    for child in (2 * node, 2 * node + 1):
        tree[child] += lazy[node] * (mid - l + 1 if child == 2 * node else r - mid)
        lazy[child] += lazy[node]
    lazy[node] = 0                  # 关键：推完必须清零
def query(tree, lazy, node, l, r, ql, qr):
    if qr < l or r < ql:            # 完全不相交
        return 0
    if ql <= l and r <= qr:         # 完全包含
        return tree[node]
    push_down(tree, lazy, node, l, r)
    mid = (l + r) // 2
    return query(tree, lazy, 2 * node, l, mid, ql, qr) + \
           query(tree, lazy, 2 * node + 1, mid + 1, r, ql, qr)
def update(tree, lazy, node, l, r, ql, qr, val):
    if qr < l or r < ql:
        return
    if ql <= l and r <= qr:         # 整段落入：记账，不往下走
        tree[node] += val * (r - l + 1)
        lazy[node] += val
        return
    push_down(tree, lazy, node, l, r)
    mid = (l + r) // 2
    update(tree, lazy, 2 * node, l, mid, ql, qr, val)
    update(tree, lazy, 2 * node + 1, mid + 1, r, ql, qr, val)
    tree[node] = tree[2 * node] + tree[2 * node + 1]
```

> **坑**　线段树的数组要开 4n，不是 2n。因为当 n 不是 2 的幂时，完全二叉树会出现空位，开 2n 会越界。另外递归深度是 O(log n)，不会爆栈。

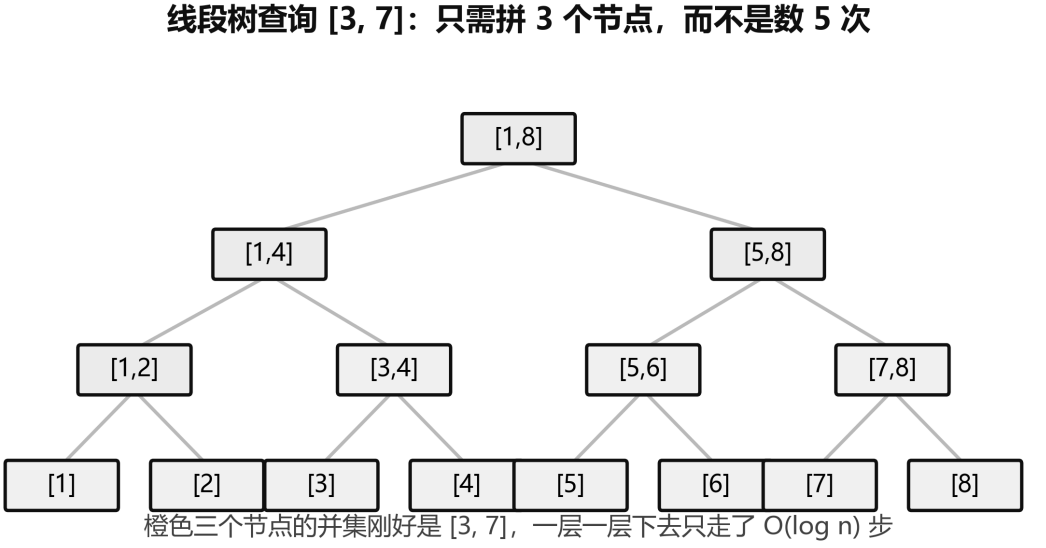


图 3-2　线段树查询 [3, 7]：把三个节点拼起来就行，不需要数 5 次

3.16　树状数组：更省内存的前缀和

如果你要的只是「前缀和」——从头到某一位的总和，而不是任意区间的最大值、最大公约数这类花样，那就不需要线段树。树状数组（Binary Indexed Tree，又称 Fenwick Tree）只用 n 个位置就能做到同样的 O(log n)，代码只有十几行。

它的神奇之处全在一个位运算上：lowbit(i) = i & -i，取出 i 的二进制里最低位的那个 1。例如 12 的二进制是 1100，lowbit 是 4；13 是 1101，lowbit 是 1。

打个比方：一排储蓄罐，编号 1 到 n。第 i 个罐子里装的不是第 i 个元素，而是「从第 i 号往前数 lowbit(i) 个元素的和」。求前缀和就像把几个储蓄罐倒在一起：13 → 12 → 8 → 0，共 3 步。修改第 i 个元素时，也只需要沿着 i → i+lowbit(i) 路径向上更新。

两边都是 O(log n)，但常数比线段树小得多，内存只要 1/n。考试、竞赛里凡是碰到「单点修改 + 区间求和」，第一反应就应该是它。

代码 3-16　树状数组：单点加、前缀和、区间和

```
def lowbit(x):
    return x & -x
def add(bit, i, v):        # 第 i 位加 v
    while i < len(bit):
        bit[i] += v
        i += lowbit(i)
def prefix(bit, i):        # 求 1..i 的和
    s = 0
    while i > 0:
        s += bit[i]
        i -= lowbit(i)
    return s
def range_sum(bit, l, r):  # 求 l..r 的和
```

```
    return prefix(bit, r) - prefix(bit, l - 1)
```

> **小结**　线段树能做的事树状数组都能做，但树状数组做不了“区间最大值 + 区间修改”。能用树状数组就别用线段树。

3.17　红黑树：工业界最常用的平衡树

3.13 讲了 AVL 树：它严格保证任意节点左右子树高度差不超过 1。好处是树很矮、查询很快；坏处是要求太严，插入删除时经常要反复旋转，写起来也容易出错。

红黑树（Red-Black Tree）选了一条折衷的路：不再要求严格平衡，只要求「最长的一条路不超过最短一条的两倍」。条件宽了，旋转次数就少了，插入删除平均只需 O(1) 次旋转（最坏 2–3 次）。好处不只是快——代码里给节点加一个位的颜色，比维护高度字段更省事。

它的五条性质好比一套「楼层限高」的建筑规则：① 每个节点要么红要么黑；② 根节点是黑的；③ 红节点的两个孩子都必须是黑的（不允许两个红节点相邻）；④ 从任意节点出发到每一个空位置，路上的黑节点个数相等；⑤ 空位置算黑色。五条合起来就能推出「高度 $\le 2\times\log_2(n+1)$」。

表 3-3　AVL 树与红黑树的取舍

对比项	AVL 树	红黑树
平衡条件	左右子树高度差 ≤ 1	最长路 ≤ 2 × 最短路
查询性能	更矮，略快	略高，差别在常数级
插入删除	旋转多	旋转少，写入更快
实现难度	要维护高度字段	只要一个颜色位
典型应用	读多写少的内存索引	std::map、TreeMap、Linux 内核调度器

> **实用**　实际工作中你几乎不需要手写红黑树，但一定要知道为什么 C++ 的 map、Java 的 TreeMap、Linux 内核都选它：因为它在「查询快」和「写入不慢」之间取了最好的均衡。

3.18　B 树与 B+ 树：数据库和文件系统的底座

前面讲的二叉树都假设数据在内存里。但数据库里的数据在磁盘上，而磁盘和内存有一个数量级的差别：读一次内存约 100 纳秒，读一次磁盘约 10 毫秒，相差约十万倍。所以在磁盘上，「读了几次磁盘」比「比较了几次」重要得多。

打个比方：图书馆的目录柜。如果每个小柜子只能放一本书，你要找一本书得跑几十趟。B 树把每个节点设计成能放数百个键的「智能柜子」，一次读磁盘就把一整排目录全读进来，在内存里做二分查找。一棵 1000 万条数据的树，高度只有 3–4 层，也就是只读 3–4 次磁盘。

B+ 树是 B 树的改良版：中间节点只存键不存数据，所有真正的数据都在最底层叶子节点，而且叶子之间用链表串起来。这一条链表解决了数据库最常用的一个操作：范围查询。要查「年龄 20 到 30 岁的用户」，先找到 20 岁所在的叶子，然后沿着链表往后走就行，不用回到根重新找。

表 3-4　为什么数据库用 B+ 树而不用二叉搜索树

数据量	红黑树高度	B+ 树高度（m=500）	磁盘读次数差	2 千万条记录	3 层
				大约 24 层	
1 万	约 14	2	约 7 倍		
100 万	约 20	3	约 7 倍		
1 亿	约 27	4	约 7 倍		

> **对号入座**　MySQL 的 InnoDB 用 B+ 树，NTFS 和 ext4 的目录索引用 B+ 树，SQLite 用 B 树。看到「为什么数据库这么快」这类问题，答案往往就在这里。

3.19　字符串哈希与后缀数组：给文字按一个「指纹」

3.5 的哈希表把任意类型的键变成一个整数下标。同样的思路用在字符串上，就能解决一类很常见的问题：判断两个子串是不是一样。

逐字比的话，长度为 L 的子串要比 L 次。但如果把每个字符看成一个数字，把整个字符串当成一个「基数为 B 的多位数」，再对一个大素数取模，就得到一个「指纹」。两个子串指纹不同，就一定不同；指纹相同，则几乎一定相同。更重要的是，求任意一段的指纹只需 O(1)——前缀指纹相减就行。

这就是滚动哈希（Rolling Hash）。它是实现上最简单的「子串比较」手段，但有一个坑：单个模数在恶意数据下可能被构造撞击。生产环境里一般用两个不同的模数（双哈希），或者用 64 位自然溢出。

后缀数组（Suffix Array）是另一条路：把一个字符串的所有后缀按字典序排好，记下每个后缀的起始位置。有了它，「最长重复子串」「两个串的最长公共子串」「出现最多的子串」这些传统上要平方级的问题，都变成了线性级。打个比方：把一本书的所有章节标题按字母顺序排好，重复的东西自然会挨在一起。

代码 3-17　滚动哈希：O(1) 取任意子串指纹

```
B1, M1 = 131, 10 ** 9 + 7
B2, M2 = 137, 10 ** 9 + 9
def build(s):
    n = len(s)
    h1 = [0] * (n + 1); h2 = [0] * (n + 1)
    p1 = [1] * (n + 1); p2 = [1] * (n + 1)
    for i, ch in enumerate(s):
        h1[i+1] = (h1[i] * B1 + ord(ch)) % M1
        h2[i+1] = (h2[i] * B2 + ord(ch)) % M2
```

```
        p1[i+1] = p1[i] * B1 % M1
        p2[i+1] = p2[i] * B2 % M2
    return h1, h2, p1, p2
def sub(h, p, M, l, r):        # 取 s[l:r] 的指纹，下标从 0 开始
    return ((h[r] - h[l] * p[r - l]) % M + M) % M
def same(l1, l2, k):           # 判断两个长度 k 的子串是否相等
    return (sub(h1, p1, M1, l1, l1 + k) == sub(h1, p1, M1, l2, l2 + k)
            and sub(h2, p2, M2, l1, l1 + k) == sub(h2, p2, M2, l2, l2 + k))
```

应用　两个文件的「差异比较」就靠这套：先切成块、比块的指纹，指纹不一样的块才逐字比。Git、DNA 序列比对、拼写检查，底层都有它的影子。

3.20　并查集进阶：带权与可撤销

3.9 的并查集只能回答「你们是不是一伙的」。可很多题目问的是「你们差多少」：这两个人相差几辈？这两个城市相距多远？这两个点在同一侧还是两侧？这时用带权并查集。

做法很小：除了父指针，再给每个节点存一个「到父亲的距离」。路径压缩时把这个距离也一路累加到根上，于是任意两个节点之间的差值就能在 O(α(n)) 内算出。打个比方：家谱里不只记「谁是谁的爷爷」，还记「相差几辈」，任意两人都能立刻算出辈分差。

可撤销并查集则是另一个需求：分治算法里经常要「先合并，回溯时再拆开」。路径压缩会改掉中间结构，无法还原。办法是只用按秩合并（不做路径压缩），并把每一次合并的两个根记到栈里，回溯时按栈逆序还原。代价是单次操作从「几乎 O(1)」退到 O(log n)，换来的是可撤销能力。

代码 3-18　带权并查集：判断两个人是同伙还是敌对

```
# weight[x]: x 到 parent[x] 的关系（0 同伙 / 1 敌对）
def find(x):
    if parent[x] != x:
        root = find(parent[x])
        weight[x] ^= weight[parent[x]]   # 先把距离累加到根上
        parent[x] = root
    return parent[x]
def union(x, y, rel):        # 声明 x 与 y 的关系为 rel
    rx, ry = find(x), find(y)
    if rx == ry:
        return (weight[x] ^ weight[y]) == rel   # 检验是否矛盾
    parent[rx] = ry
    weight[rx] = weight[x] ^ weight[y] ^ rel
    return True
```

经典题　「判断一张图是不是二分图」「维护前缀差」「区间合法性检查」，考的都是这个奸势。不会手写也没关系，但知道它能做什么，以后碰到就能想起来。

3.21 图的四种存法

图的本质是「点 + 边」，但同一个图可以用四种不同方式存起来，速度差别很大。选错了，算法写对了也会超时。打个比方：查「北京到上海有没有直飞」，翻航班计划表（矩阵）一眼就能看到，但表会很大；翻「北京出发的所有航班」（邻接表）又快又省纸。

表 3-5 图的四种存储方式

存法	空间	查一条边	遍历某点的邻居	适用场景
邻接矩阵	$O(n^2)$	O(1)	O(n)	点少、边密（$n \le 2000$）
邻接表（vector）	O(n+m)	—	O(出度)	通用，最常用
边集数组	O(m)	—	O(m)	Kruskal、Bellman-Ford 只需遍历边
链式前向星	O(n+m)	—	O(出度)	写竞赛题，缓存友好

> **它解决的是内存问题** 邻接表用链表实现时，每个节点都要单独申请内存，节点一多又慢又吃内存。链式前向星用数组把它压平。

做法是开三个数组：head[u] 存 u 的第一条边编号，to[i] 存第 i 条边通向哪个点，nxt[i] 存第 i 条边的下一条边编号。所有边都按「链表」的方式串起来，但节点用的是数组下标，没有一次 new。

- 加一条边：把新边放到数组末尾，让它指向 head[u]，再让 head[u] 指向它。整个过程是 O(1)。
- 遍历 u 的邻居：从 head[u] 出发，顺着 nxt 一路走下去，直到 -1 结束。
- 无向图要正反各加一条边，所以数组要开到 2m 这么大。

> **读代码看这里** head 的初值是 -1，表示「没有边」。这个 -1 既是一条边的结尾标志，也是整个链表结束的标志。

代码 3-19 链式前向星：用数组模拟邻接表

```
head = [-1] * (n + 1)       # head[u]：u 的第一条边编号
to   = [0] * (2 * m + 1)
nxt  = [0] * (2 * m + 1)
wt   = [0] * (2 * m + 1)
cnt  = 0
def add_edge(u, v, w):
    global cnt
    cnt += 1
    to[cnt], wt[cnt], nxt[cnt] = v, w, head[u]
    head[u] = cnt
def neighbors(u):
```

```
    e = head[u]
    while e != -1:
        yield to[e], wt[e]
        e = nxt[e]
```

3.22　LRU 缓存：哈希表加双向链表

缓存的空间总是有限的，满了就得丢东西。丢谁？最常用的策略叫 LRU（Least Recently Used，最近最少使用）：丢掉最久没被碰过的那个。比如手机的后台应用，你最近开过的会保留，很久没点开的就被清掉。

一个结构要同时满足两件事：① 按 key 找到节点，要 O(1)；② 能知道谁是最久没用的，也要 O(1)。单用哈希表只能做到第一件；单用链表只能做到第二件。把两者拼在一起就行：哈希表存 key → 链表节点，链表按使用时间从近到远排列。

为什么是双向链表而不是单链表？因为删除一个已知节点时，需要改它前一个节点的指针。单链表要从头找一遍，O(n)；双向链表直接就能改，O(1)。这是一个很典型的「两个结构各管一件事」的设计模式。

代码 3-20　LRU 缓存：哈希表 + 双向链表（脱离版）

```
from collections import OrderedDict
class LRUCache:
    def __init__(self, cap):
        self.cap = cap
        self.data = OrderedDict()   # 靠前的 = 最近用过
    def get(self, key):
        if key not in self.data:
            return -1
        self.data.move_to_end(key)  # 刚用过，排到队尾
        return self.data[key]
    def put(self, key, value):
        if key in self.data:
            self.data.move_to_end(key)
        self.data[key] = value
        if len(self.data) > self.cap:
            self.data.popitem(last=False)   # 丢掉最久未用
```

> **衍生一下**　把「最近最少使用」换成「使用次数最少」就是 LFU；加上过期时间就是 TTL 缓存；再加上尺寸而不是数量就是正经的内存缓存。Redis 里的 maxmemory-policy 选项，说的就是这几种策略。

3.23　位运算与位图：用一个比特装一个开关

计算机里的整数本来就是一串二进制位，位运算直接在这串位上做事，所以特别快。它又是很多高级结构（树状数组的 lowbit、哈希表的扰动函数、布隆过滤器、状态压缩动态规划）的基础。不熟练它，看上述内容就像看天书。

表 3-6　六个基本位运算与它们的用途

运算	含义	典型用途
a & b	按位与	取出某位、判断奇偶（a & 1）
a | b	按位或	合并两个标记集
a ^ b	按位异或	不用临时变量交换、找只出现一次的数
~a	按位取反	求补码，卷积算法用得很多
a << k	左移 k 位	乘 2^k，构造位图时用
a >> k	右移 k 位	除 2^k，取出某一段位

位图（Bitmap）是最常见的用法：用第 i 个二进制位表示「第 i 个数存不存在」。一亿个数，用布尔数组要 100 MB，用位图只要 1.25 MB——小了 64 倍。这不是小优化，是一个数据结构能不能装进内存的区别。

代码 3-21　位图与几个常用位运算技巧

```
bits = bytearray(n // 8 + 1)
def set_bit(i):            # 置 1
    bits[i >> 3] |= 1 << (i & 7)
def clear_bit(i):          # 置 0
    bits[i >> 3] &= ~(1 << (i & 7)) & 0xFF
def test_bit(i):           # 查询
    return (bits[i >> 3] >> (i & 7)) & 1
def lowbit(x):  return x & -x              # 取最低位的 1
def is_odd(x):  return x & 1               # 判奇偶
def drop_low(x): return x & (x - 1)        # 清掉最低位的 1（统计 1 的个数）
def popcount(x):
    c = 0
    while x:
        x &= x - 1
        c += 1
    return c
```

> **小技巧**　判断一个数是不是 2 的幂：x > 0 and (x & (x - 1)) == 0。一行代码省一个循环，这种写法在底层代码里比比皆是。

3.24　概率数据结构：用少量内存答大量问题

有一类问题天然就不需要精确答案：「这个 IP 一分钟内请求了多少次？」「这个页面今天有多少人看过？」你只需要一个近似值，而且要能在几百 MB 内存里装下上亿条记录。这时就该请概率数据结构上场了。

Count-Min Sketch：开 k 行计数器，每行用不同的哈希函数。来一个元素，k 行同时加一。查询时取 k 个计数器里的最小值。打个比方：k 个保安分别记账，有人冒名顶替时

只会让记录偏大，所以取最小的那份最接近真相。代价是只会高估、不会低估，但偏差有上界。

HyperLogLog：用来估算「有多少个不重复的元素」。把元素哈希成一串位，看它开头有多少个连续的 0，记录最大值。听起来很玄，道理很简单：一直扛硬币，连续出现 6 次正面，说明你抛了不止一次两次；连续 0 的个数越大，说明采样越多。一个亿个独立访客，HyperLogLog 只需 12 KB。

表 3-7　精确结构与概率结构的对比

结构	内存	问题	答案质量	典型用途
哈希集	每元素数十字节	存不存在	精确	去重
布隆过滤器	1–10 比特/元素	存不存在	可能误判（不漏判）	URL 去重、黑名单
Count-Min	几 KB 到几 MB	出现次数	只会高估	流量监控、热点统计
HyperLogLog	约 12 KB	不重复个数	误差约 0.8%	UV 统计、独立 IP

> **一句话**　概率结构的价值不在「省空间」，而在「能不能在有限资源内给出足够好的答案」。工程上很多时候，一个 99% 准确、成本十分之一的答案，比一个 100% 准确的答案更有价值。

3.25　持久化数据结构：每次修改都不破坏旧版本

普通数据结构一改，旧的内容就没了。但有时候我们希望「每一次修改都保留一个版本」——Git 的提交历史、文档的历史记录、函数式语言的不可变数据，都是这个需求。

做法叫「路径复制」。回想红黑树的插入：你只改了根到插入点这一条路上的几个节点，其余节点原封不动。那就不要改原节点，而是把这条路上的节点各复制一份，改复制件，让新的根指向它们；旧的根还指着旧的世界。两个版本共用大部分节点，只多了 log n 个新节点。

打个比方：一本书要改一个字，你不把整本书重新抄一遍，而是把目录页和那一页换成新的。旧版本依然可以翻。

表 3-8　普通结构与持久化结构的对比

对比项	普通结构	持久化结构
修改后旧版本	被覆盖	完整保留
单次修改成本	O(1) 或 O(log n)	O(log n)（路径上新建节点）
空间增量	无	每次修改 O(log n) 个节点

对比项	普通结构	持久化结构
典型应用	普通字典	Git、Clojure/Scala 容器、时间旅行数据库

> **为什么值得知道**　因为它透漏了一个工程思想：可变状态是 bug 的温床，不可变数据是并发与回滚的朋友。很多现代系统的设计都在往这个方向靠。

3.26　选型实战：六个真实场景怎么选

学完数据结构最容易犯的错误，是把它当成背诵内容，而不是选型工具。下面用六个你大概率会真遇到的场景，把本章的结构串一遍。

表 3-9　六个场景的选型结论与理由

场景	选什么	为什么
按用户 ID 查详情，量大、读多写少	哈希表 + 数据库索引	O(1) 定位，不需要顺序
实时统计「前 5 分钟最大 QPS」	单调队列	滑动窗口最值只需 O(n)
任务调度器：每次取优先级最高的任务	堆（优先队列）	O(log n) 取最值 + 动态插入
社交关系：判断两人是否在同一个群体	并查集	几乎 O(1)，代码极短
输入法提示：输入前缀列出候选词	Trie	按字符逐层下降，不用扫全部词
历史版本可回溯的配置中心	持久化结构（路径复制）	旧版本完整保留，共用绝大部分节点

> **方法论**　选型时先问两个问题：「最常做的操作是哪一个？」「数据有多大？」把这两个答案对照上表，选错的概率会小很多。

3.27　手写一个哈希表：从数组到链地址法

哈希表在上一节里是「用算出来的位置直接找到」。听起来神奇，其实它的全部秘密只有两件事：一个把任意数据变成下标的函数（哈希函数），和一套处理「两个数据算到同一个下标」的办法（冲突处理）。这一节我们不调用任何库，从零写一个能用的哈希表，把这两件事各推演一遍。

打个比方：哈希表像一排带编号的邮箱。你要给「张三」投信，先按一套规则算出他的邮箱号（比如姓氏笔画数），直接把信塞进那个箱子。取信时同样算一遍号码，一步到位，不用一箱一箱翻。问题是，姓李的太多，算出来都落在 3 号箱——这就是冲突。

3.27.1　第一件事：哈希函数要满足什么

一个合格的哈希函数有三个要求。第一，算得快，最好是几次加减乘除的量级；第二，分布均匀，不同的输入要尽量落到不同的位置；第三，结果稳定，同一个输入每次算出来必须是同一个数。

对字符串，最经典的做法是「多项式滚动哈希」：把字符串看成一个多进制数。比如取基数 31，"cat" 就是 $c\times31^2 + a\times31^1 + t\times31^0$。这个式子从左往右扫一遍就能算出来，复杂度是长度的一次方。

```
def hash_str(s, m):
    # 把字符串映射到 0..m-1 的整数
    h = 0
    for ch in s:
        h = (h * 31 + ord(ch)) % m   # ord 取字符编码
    return h
```

> **一句话原理**　把一个字符串当成 31 进制的数，再对一个大数取余，得到的余数就是它的指纹。

字符串没法直接比较大小，也不好放进数组下标，所以先把它变成一个整数。做法是从左到右读，每读一个字符就把已有的值乘 31 再加上这个字符的编码——相当于把每个字符放到不同的「数位」上。同样的字母换个顺序，乘的次数就不同，指纹也就不同。

取模是为了把结果压进数组下标范围。取模会带来冲突（不同的串算出同一个余数），但不取模的话数值会大到没地方存，所以这是必要的代价。

- 第一步：把 h 初始化为 0，准备一个底数（常用 31 或 131）和一个模数。
- 第二步：从左到右扫，每一步 h = (h × 底数 + 字符编码) % 模数。
- 第三步：扫完得到的 h 就是这个串的指纹，比较指纹即可，冲突再用别的手段兜底。

> **读代码看这里**　cat、act、tac 三个词字母完全相同，但算出的指纹完全不同——这正是位置参与的体现。

```
for w in ["cat", "act", "tac", "dog"]:
    print(w, hash_str(w, 16))
```

代码 3-40　多项式滚动哈希：把字符串当成 31 进制数

> **为什么取模**　因为下标必须在 0 到 m-1 之间。取模会带来冲突（不同的字符串可能得到同一个余数），但这是必要的代价：不取模，哈希值会大到没地方存。

有一点值得注意：cat、act、tac 由同样的字母组成，但算出来的哈希值完全不同，因为位置（也就是乘以 31 的次数）不一样。这正是我们想要的——顺序变了，指纹就该变。

3.27.2 第二件事：冲突了怎么办

冲突不可避免（把无限多的字符串映射到有限个箱子，必然撞车），只能处理。两种主流办法：链地址法和开放寻址法。

表 3-20 两种冲突处理办法的对比

	链地址法（拉链）	开放寻址法（线性探测）
基本想法	每个箱子挂一条链表，撞车了就挂在后面	撞车了就往后找下一个空箱子
内存	要存链表指针，有额外开销	全存在一个大数组里，更省内存
删除	直接从链表里摘掉即可	不能直接删，否则会切断后续元素的查找路径，通常打「墓碑」标记
负载因子容忍度	可以接近 1，甚至更高	最好控制在 0.7 以下，否则连成一片
典型代表	Java 的 HashMap、Python 的 dict	Rust 的 HashMap、各类高性能场景

我们选择链地址法，因为它最好理解：哈希表就是一个「数组的数组」。外层是 m 个箱子的数组，每个箱子里放一个列表。

3.27.3 完整实现：二十行写完的哈希表

```
class HashTable:
    def __init__(self, m=8):
        self.m = m                    # 箱子个数
        self.buckets = [[] for _ in range(m)]
        self.size = 0                 # 存了多少个键值对
    def _idx(self, key):
        return hash_str(str(key), self.m)
    def put(self, key, value):
        b = self.buckets[self._idx(key)]
        for i, (k, v) in enumerate(b):
            if k == key:
                b[i] = (key, value)   # 已存在，覆盖
                return
        b.append((key, value))        # 新键，挂在链尾
        self.size += 1
```

这一版实现什么 用链地址法把哈希表的最小骨架写出来：数组装桶，桶里挂链。

数组负责「快速定位」，链表负责「容纳冲突」。每个键先用哈希函数算出一个槽位编号，如果那个槽位已经有别人了，就把新键值对接在该槽的链上。查找时先算槽位，再在链上按 key 一个个比。

- put：算槽位 → 在链上找有没有同名的键，有就改值，没有就接一个新的。
- get：算槽位 → 在链上按 key 找，找到就返回，找不到返回默认值。
- delete：算槽位 → 在链上找到并摘掉，顺便把元素计数减一。

> **读代码看这里**　所有操作都先 _idx(k) 算槽位。记住 idx 这一步是 O(1)，真正的开销在链上那几个元素的比较。

```
    def get(self, key, default=None):
        for k, v in self.buckets[self._idx(key)]:
            if k == key:
                return v
        return default
```

代码 3-41　哈希表的最小完整实现（链地址法）

整个结构的核心只有两行：算下标、在对应的箱子（也就是一条短链表）里找。如果哈希函数足够均匀，平均每个箱子里的元素个数是 元素总数 ÷ 箱子数，也就是负载因子。只要这个比值不大，查找就接近常数时间。

3.27.4　扩容：什么时候把箱子变多

如果一直往里塞而不加箱子，每个链表会越来越长，哈希表就退化成了一堆链表，查找从 O(1) 掉回 O(n)。所以当负载因子超过一个阈值（通常 0.75）时，就要扩容：新建一个两倍大的数组，把所有元素重新算一遍下标搬过去。

> **为什么需要扩容**　桶越满，链越长，查找就从 O(1) 退化成 O(n)。扩容是把它拉回来的唯一办法。

衡量桶满不满的指标叫负载因子，等于「已存元素数 ÷ 桶数」。它一旦超过阈值（工程上常取 0.75），就把桶数翻倍，然后把所有元素重新算一遍槽位塞进新桶里。

- 第一步：申请一个两倍大的新桶数组。
- 第二步：把所有旧元素依次取出，用新桶数重新算槽位，塞进去。
- 第三步：把新数组替换掉旧的，元素计数重新累计。

单次扩容要搬 n 个元素，看着很贵，但它平均到每次插入上只有常数代价——这叫做均摊 O(1)。就像搬一次家很累，但摊到每天的生活里就微不足道了。

> **易错点**　重新计算槽位时用的是新的桶数，不是旧的。用错了会导致大量元素挤在同一个桶里。

```
    def _grow(self):
        old = self.buckets
        self.m *= 2
        self.buckets = [[] for _ in range(self.m)]
        for b in old:
            for k, v in b:
```

```
                self.buckets[self._idx(k)].append((k, v))
```

代码 3-42　扩容：一次 O(n) 的搬迁换来后面的快

> **均摊 O(1)**　单次 put 偶尔会因为扩容花掉 O(n)，但扩容发生的间隔越来越长（1 次、2 次、4 次……），把总代价摊到每一次插入上，仍然是常数。这就是「均摊分析」，和第 3.1 节动态数组扩容是同一个道理。

最后提醒一句：扩容后所有元素的下标都会变，因为取模的除数从 m 变成了 2m。这也是为什么哈希表不能像数组那样简单地在末尾追加——它必须搬家。

3.28　手写一个堆：上浮与下沉的完整推演

堆的规矩只有一条：每个父节点都不大于（或不小于）它的子节点。这条规矩带来的好处是，最大的那个元素永远在堆顶，取它就是一步。这一节把插入和取出这两个动作，一步步画给你看。

堆的存法很讨巧：不用真的建树，直接用一个数组假装成完全二叉树。下标为 i 的节点，它的左孩子是 2i+1，右孩子是 2i+2，父亲是 (i-1)//2。父子关系不靠指针，靠算术。

3.28.1　插入：一路向上冒泡（上浮）

插入时把新元素放到数组末尾——也就是树的最底层最右边，这样能保持「完全二叉树」的形状。但位置对了不代表大小关系对了，所以接下来让它一路往上和父亲比：比父亲小就交换，直到不再比父亲小或者已经到顶。这个过程叫上浮。

> **这一版实现什么**　往堆里插入一个元素，并让它自己回到正确的位置。

堆是一棵用数组表示的完全二叉树，规矩只有一条：父亲永远比孩子小（小顶堆）。插入时先把新元素放到数组末尾——也就是树的最下一层最右边——然后看它和父亲谁小。比父亲小就交换，换完继续和新的父亲比，直到不再比父亲小为止。这个过程叫上浮。

- 第一步：把新元素放到数组尾部，元素个数加一。
- 第二步：比较它和父亲，若比父亲小就交换。
- 第三步：重复第二步，直到到达根或者不再比父亲小。

为什么这样能维护堆的性质？因为除了新插进来的这一个，其余部分本来就是合法的堆；上浮只是把唯一的「违规点」一路推到它该在的位置。上浮最多走树的高度这么多步，也就是 O(log n)。

> **读代码看这里**　父亲的下标是 (i - 1) // 2。数组表示法里所有父子关系都靠这一条公式算出来，没有指针。

```
def push(heap, x):
    heap.append(x)
    i = len(heap) - 1
    while i > 0:
```

```
        parent = (i - 1) // 2
        if heap[i] < heap[parent]:      # 小顶堆：小的往上走
            heap[i], heap[parent] = heap[parent], heap[i]
            i = parent
        else:
            break
```

代码 3-43　插入并上浮：最多走树的高度这么多步

表 3-21　往小顶堆 [1, 3, 5, 7, 4, 8] 里插入 2 的过程

步骤	数组状态	发生了什么
放末尾	[1,3,5,7,4,8,2]	新元素 2 放在下标 6
比父亲	[1,3,5,7,4,8,2]	父亲是下标 2 的 5，2 < 5，交换
继续比	[1,3,2,7,4,8,5]	现在在下标 2，父亲是下标 0 的 1，2 > 1，停

只走了两层就停下来了。因为是完全二叉树，高度大约是 $\log_2 n$，所以插入最多花 log n 步。

3.28.2　取出堆顶：把最后的搬到顶上再一路向下（下沉）

取最小值就是把堆顶拿走。但直接拿走会在根上留个洞。常规做法是：把数组最后一个元素搬到根上，长度减一，然后让这个新根一路往下找比自己小的孩子交换，直到两个孩子都不比自己小为止。这个过程叫下沉。

> **这一版实现什么**　取出堆顶（最小值），并让剩下的元素重新变成合法的堆。

直接删掉数组第一个元素会把整棵树的结构打乱，代价太高。标准做法是：把最后一个元素挪到根上，元素个数减一，然后让这个「临时根」往下走——每次和两个孩子中较小的那个交换，直到它比两个孩子都小为止。这个过程叫下沉。

- 第一步：记下根的值作为返回值，把最后一个元素搬到根的位置。
- 第二步：在两个孩子里挑较小的那个，如果比当前的根还小，就和它交换。
- 第三步：重复第二步，直到没有孩子比它更小，或者已经到达叶子。

上一节讲的「建堆是 O(n)」也靠下沉：从最后一个非叶节点开始，依次往前对每个节点下沉一次。越靠下的节点越多，但它们下沉的距离越短，总代价加起来反而比挨个插入的 O(n log n) 更小。

> **读代码看这里**　两个孩子是 2i+1 和 2i+2，要先判断右孩子是否存在，否则会越界。

```
def pop(heap):
    top = heap[0]
    last = heap.pop()               # 拿走最后一个
    if heap:
        heap[0] = last              # 先放到根上
```

```
        i, n = 0, len(heap)
        while True:
            l, r, small = 2*i+1, 2*i+2, i
            if l < n and heap[l] < heap[small]: small = l
            if r < n and heap[r] < heap[small]: small = r
            if small == i: break
            heap[i], heap[small] = heap[small], heap[i]
            i = small
    return top
```

代码 3-44　取堆顶并下沉：每层挑更小的那个孩子交换

> **为什么挑更小的孩子**　如果和较大的那个孩子交换，换上去的父亲可能仍然大于另一个孩子，规矩就被破坏了，还得再换一次。挑更小的孩子交换，一次到位。

取出动作也是走树高，所以是 O(log n)。结合起来，用一个堆维护一组数据，插入和取极值都是 O(log n)——这就是第 3.7 节里 Top-K 问题能比全排序快的原因：只要 K 个元素的小顶堆，扫描一遍数组，比堆顶小就丢，最后堆里就是最大的 K 个。

3.29　手写并查集：三步优化把复杂度压到接近常数

并查集要回答两个问题：a 和 b 是不是一伙的？把 a 和 b 并成一伙。它不关心这一伙内部的结构，只关心「谁和谁是一伙」。听起来简单，但要做到极快，需要三步层层递进的优化。这一节把三步逐个加上，并对比它们的复杂度。

打个比方：并查集像一群公司。每个公司有一个代表（根节点）。问两个人是不是同事，只要看他们往上找，最终找没找到同一个老板。

3.29.1　第一版：用一个数组记录每个元素的父亲

```
parent = list(range(n))          # 一开始，每个人是自己的老板
def find(x):
    while parent[x] != x:
        x = parent[x]
    return x
```

> **这一版实现什么**　并查集的第一版：只做两件事，找祖宗和连起来。

并查集回答的是「你和我在不在同一个圈子里」这类问题。做法是给每个元素记一个父亲，一路往上找，找到那个没有父亲的元素（它是自己父亲），就是这一伙的「祖宗」。判断两个人是否同伙，只要看祖宗是不是同一个；合并两伙，就把一方的祖宗挂到另一方祖宗下面。

- find(x)：顺着父亲一路往上爬，爬到 x == parent[x] 就是祖宗。
- union(a, b)：先各找祖宗，祖宗不同就把一个挂到另一个下面。

这一版能跑，但有个隐患：如果一直按 (1,2)、(2,3)、(3,4) 的顺序合并，树会长成一条笔直的链。此时 find 要从头爬到尾，最坏是 O(n)，并查集就退化成了普通链表。下面两小节各修一半。

> **读代码看这里**　parent[x] == x 这个判断就是「我是自己的父亲」，也就是根节点的标志。

```
def union(a, b):
    ra, rb = find(a), find(b)
    if ra != rb:
        parent[ra] = rb          # 把 a 的老板挂到 b 的老板下面
```

代码 3-45　并查集第一版：能跑，但可能退化成一条长链

这版的问题是，如果一直按 (1,2)、(2,3)、(3,4) 的顺序合并，树会长成一条直线。此时 find 要一路爬到顶，最坏是 O(n)。

3.29.2　第二版：按秩合并，别让树长高

合并时不要随便挂，而是把「浅的树挂到深的树下面」。用一个 rank 数组记录每棵树的高度，这样树高最多长到 log n。

```
rank = [0] * n
```

> **修的是什么**　合并时别再随手挂，而是让矮的树挂到高的树下面，控制树高。

上一版之所以会退化成一条链，是因为合并时总是把 a 的祖宗挂到 b 的祖宗下面，完全不管两棵树的形状。按秩合并加了一条规矩：额外记录每棵树的高度（秩），合并时把矮的挂到高的下面。

- 第一步：两棵树一样高时随便挂，但要把新树的高度加一。
- 第二步：一高一矮时，把矮的挂到高的下面，高度不变。
- 第三步：这样处理后树高始终被压在 log n 以内，find 最坏也是 O(log n)。

道理很简单：树越矮，往上爬的路越短。与其让树越来越高，不如让新来的那一伙融入结构更好的那一伙里。

> **易错点**　只有两棵树一样高时高度才加一。一高一矮时高度不变，写错的话秩就失真了，优化也会失效。

```
def union(a, b):
    ra, rb = find(a), find(b)
    if ra == rb: return
    if rank[ra] < rank[rb]:      # 保证 ra 是更深的那棵
        ra, rb = rb, ra
    parent[rb] = ra              # 浅的挂到深的下面
    if rank[ra] == rank[rb]:
        rank[ra] += 1            # 一样深，合并后高度加一
```

代码 3-46　按秩合并：树高被控制在 log n 以内

3.29.3　第三版：路径压缩，一次查找顺手拍平

既然每次 find 都要爬一条链，那就顺手把这条链上的所有节点都直接挂到根上。下次再来，一步到位。这一招叫路径压缩，通常写成两行递归或一行迭代。

> **修的是什么**　查找的路上顺手把树压平，让下一次查找更快。

上一版把树高压到了 log n，这一版更进一步：既然 find 已经一路爬到了祖宗，不如在回来的路上把沿途所有节点都直接挂到祖宗下面。下次再找这些节点，一步就能到顶。这叫路径压缩。

- 第一步：递归找到祖宗。
- 第二步：回来时把每个节点的父亲直接改成祖宗。
- 第三步：整棵树被压得越来越扁，几条路径之后几乎变成一层。

两个优化叠加之后，并查集的均摊复杂度低到几乎可以认为是常数——严格说是 O(α(n))，其中 α 是反阿克曼函数，在你可能遇到的任何数据规模下都不超过 4。所以工程上都说并查集「接近 O(1)」。

> **读代码看这里**　那一行 parent[x] = find(parent[x]) 同时完成了「找祖宗」和「顺手压平」两件事，是本小节最值得记住的一行。

```
def find(x):
    while parent[x] != x:
        parent[x] = parent[parent[x]]   # 隔代压缩：跳到爷爷
        x = parent[x]
    return x
```

代码 3-47　路径压缩：查找过程本身就在修树

表 3-22　三步优化后的复杂度对比

版本	find 最坏复杂度	合并 n 次的总代价
只记录父亲	O(n)	O(n²)，会超时
加上按秩合并	O(log n)	O(n log n)
再加路径压缩	接近 O(1)（严格说是反阿克曼函数 α(n)）	接近 O(n)，实践中等于常数

反阿克曼函数 α(n) 增长得极慢：在 n 小到宇宙中原子的数量时，它都不到 5。所以工程上你可以直接把它当作常数。这也是并查集能成为图论、连通性问题标配工具的原因。

3.30　数据结构选型总表：21 个结构横向对比

前面二十九节讲了二十多个结构。真正上手时最大的困难不是「不会写」，而是「不知道挑哪个」。这一节把所有结构放在同一张表里，按最常用的四个操作（按下标取、查找某个值、插入、删除）给出复杂度，并在最后一列写清它最擅长什么。

表 3-23　21 个数据结构横向对比（n 为元素个数）

结构	按下标访问	查找值	插入	删除	最擅长的事
数组	O(1)	O(n)	O(n)	O(n)	随机访问、内存紧凑
动态数组	O(1)	O(n)	均摊 O(1)（尾部）	O(n)	当栈用、当列表用
链表	O(n)	O(n)	O(1)（已知位置）	O(1)（已知位置）	频繁在中间增删
双向链表	O(n)	O(n)	O(1)	O(1)	LRU 缓存、迭代器
栈	—	—	O(1)	O(1)	括号匹配、表达式求值、撤销
队列	—	—	O(1)	O(1)	先来先服务、BFS
双端队列	—	—	O(1)	O(1)	滑动窗口最值
哈希表	—	平均 O(1)	平均 O(1)	平均 O(1)	去重、计数、缓存
有序数组	O(1)	O(log n)	O(n)	O(n)	只读的静态查找表
二叉搜索树	—	平均 O(log n)	平均 O(log n)	平均 O(log n)	有序遍历
AVL / 红黑树	—	O(log n)	O(log n)	O(log n)	需要稳定有序的场合
B+ 树	—	O(log n)	O(log n)	O(log n)	磁盘上的索引
堆	—	O(n)	O(log n)	O(log n)	随时取极值、Top-K
跳表	—	O(log n)	O(log n)	O(log n)	并发友好的有序表
Trie 字典树	—	O(词长)	O(词长)	O(词长)	前缀匹配、自动补全

结构	按下标访问	查找值	插入	删除	最擅长的事
线段树	—	O(log n) 区间	O(log n)	O(log n)	区间求和、区间最值
树状数组	—	O(log n) 前缀	O(log n)	O(log n)	只做前缀和、常数更小
并查集	—	接近 O(1)	接近 O(1)	不支持	连通性、Kruskal
LRU 缓存	—	O(1)	O(1)	O(1)	淘汰最久未用
布隆过滤器	—	O(k)	O(k)	不支持（只能整体重建）	海量数据的「一定不在」判断
位图	O(1)	O(1)	O(1)	O(1)	稠密整数集合、状态压缩

这张表不用背。正确的用法是：先想清楚你的程序最频繁做的是哪件事，然后到表里找那一列，挑最快同时又不太绕的那个。如果两个结构差不多，选更简单的那个——代码简单本身也是一种性能。

> **一句顺口溜**　取值快看数组，查值快看哈希，有序看树，取极值看堆，连通看并查集，前缀看树状数组，区间看线段树，前缀匹配看 Trie，淘汰看 LRU。



图 3-3　数据结构全景：四大类十几个结构，按「元素之间是什么关系」划分

3.31　洪水填充：把一整块连通区域一次染完

画图软件里的油漆桶，可能是最容易看懂的算法：把光标放进一片白色区域点一下，整块白「唰」地变成红，而旁边被黑线隔开的那一块纹丝不动。它有个正式名字叫洪水填充（Flood Fill），本质就是 3.10 讲过的图遍历——只不过这里的「图」被画成了像素格子。

3.31.1　它模仿的是什么

想象一张格子纸，有些格子上画了墙。你在某一格滴一滴水，水会自己往上下左右流：绕开墙，顺着通道拐弯，但绝不穿墙。等它不再流了，湿掉的格子正好就是「从起点出发、不翻墙能走到的所有格子」。

这就是它值得单独讲的原因：它把「连通」这个抽象词变成看得见的东西。中间隔着墙，水永远过不去，那两块就是两个连通块。道理简单，但它在扫雷的空白区展开、岛屿数量、图像抠图、迷宫连通性判断里反复出现。

> **关键直觉**　洪水不算最短路径，它只做一件事：从起点开始，把所有「能走到的邻居」一个个走一遍，走过了就不再走。

3.31.2　为什么这样做能得到正确答案

把水的行为翻译成程序，只需要三条规矩。

- 只看四邻：从一个格子出发，只检查它的上、下、左、右四个方向，别的一个都不看。
- 遇到墙或已经染过的格子就停下：这两类格子都不再往下走。
- 染过色就等于做过记号：一个格子绝不会被处理两次。

第一条保证不漏（只要四邻都查过，就没人被落下）；第二条保证不撞墙；第三条保证一定会停（格子是有限的，每轮至少消耗掉一个没染的格子）。三条加在一起，效果就是「从起点出发的可达格子各访问一次」，而这正好就是连通块的定义。

3.31.3　两种实现：队列和栈

「把所有邻居都走一遍」有两种走法，差别只在下一个该处理谁。

- 用队列（先进先出）：先处理离起点近的，水一圈一圈均匀扩散。这叫广度优先搜索，简称 BFS。
- 用栈（后进先出）：先处理刚发现的那一个，水会顺着一条通道一路冲到底，撞墙再回头。这叫深度优先搜索，简称 DFS。

表 3-24　两种洪水填充写法的对比

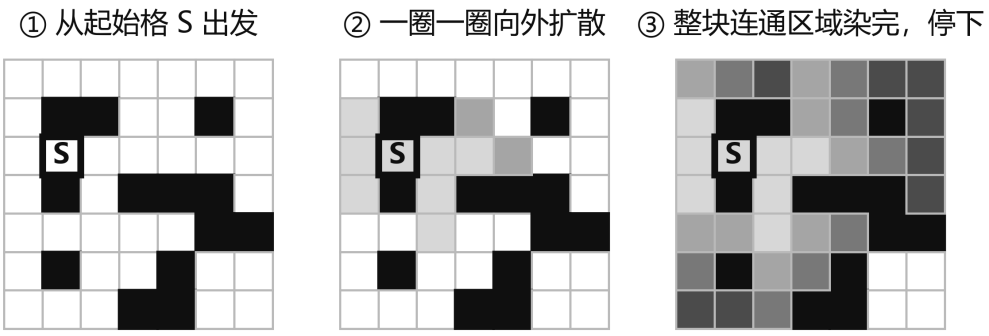
对比项	队列版（BFS）	栈版（DFS）
下一个处理谁	最早进队的（离起点最近）	最晚进栈的（离起点最远）
染色顺序	一层一层向外推，像涟漪	一条道走到黑，像钻地道

对比项	队列版（BFS）	栈版（DFS）
过程看起来	扩散过程最直观	走位跳跃，但结果一样
适合的题目	要找最短步数、分层扩散	只关心哪些格子会连通

注意最后一列：两者填出来的区域完全一样，只是染色顺序不同。要看「水是怎么扩散的」，队列版更直观；只关心「最后哪些格子被染色」，随便挑一个都行。

3.31.4 把步骤一格一格走一遍

下面这张图把三个时刻并排放在一起。灰越浅，说明越早被水染到；黑格是墙。请特别注意第三步右下角那两块白格——它们四周被墙围死，水绕不过去，所以始终没被染到。



黑格是墙。水只会沿上下左右流，流不到被墙隔开的另一块——所以填完的正好是一个「连通块」。

图 3-4 洪水填充的三步：从起点出发、一圈圈扩散、直到整块连通区域被染完（被墙隔开的右下角始终是白的）

3.31.5 代码

两个版本只差一行：队列版用 deque 的 popleft 取出最早进去的，栈版用 list 的 pop 取出最晚进去的。请对照注释读，重点看染色到底发生在哪一步。

代码 3-48 洪水填充：队列版与栈版（可直接运行）

```
from collections import deque
def flood_bfs(grid, sr, sc, new_color):
    """队列版：把 (sr, sc) 所在的连通块全部染成 new_color"""
    rows, cols = len(grid), len(grid[0])
    old_color = grid[sr][sc]
    if old_color == new_color:      # 不写这一行，起点会被反复入队
        return grid
    q = deque([(sr, sc)])
    grid[sr][sc] = new_color        # 入队的那一刻就染色，等于做好标记
    while q:
        r, c = q.popleft()          # 取最早进队的：一层一层向外扩
```

```
        for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == old_color:
                grid[nr][nc] = new_color
                q.append((nr, nc))
    return grid
def flood_dfs(grid, sr, sc, new_color):
    """栈版：把 deque 换成 list，把 popleft() 换成 pop()，其余一字不改"""
    rows, cols = len(grid), len(grid[0])
    old_color = grid[sr][sc]
    if old_color == new_color:
        return grid
    stack = [(sr, sc)]
    grid[sr][sc] = new_color
    while stack:
        r, c = stack.pop()          # 取最晚进栈的：一条道走到黑
        for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == old_color:
                grid[nr][nc] = new_color
                stack.append((nr, nc))
    return grid
if __name__ == "__main__":
    W = -1                              # -1 表示墙
    g = [[0, 0, 0, W, 0],
         [0, W, 0, W, 0],
         [0, 0, 0, 0, W],
         [W, W, 0, W, 0],
         [0, 0, 0, 0, 0]]
    flood_bfs(g, 0, 0, 8)               # 从左上角出发，把 0 染成 8
    for row in g:
        print(" ".join("%3d" % v for v in row))
```

运行结果如下。第一行最后那个 0、第二行最后那个 0，就是图 3-4 里右下角被围死的那两块；其余 16 个能走到的格子全部变成了 8。

```
  8   8   8  -1   0
  8  -1   8  -1   0
  8   8   8   8  -1
 -1  -1   8  -1   8
  8   8   8   8   8
```

3.31.6　复杂度与四个常见的坑

表 3-25　洪水填充的复杂度

写法	时间	空间	一句话
队列版（BFS）	O(R×C)	O(R×C)	每格进出队各一次

写法	时间	空间	一句话
栈版（DFS）	O(R×C)	O(R×C)	每格进出栈各一次
递归版 DFS	O(R×C)	O(R×C) 但要吃调用栈	格子多了会爆栈，不推荐

R 是行数、C 是列数。每个格子最多被访问一次，所以时间和空间都是「格子的总数」这个量级——这是最优的了，毕竟再快也得把每个格子看一遍。

> **坑一**　起点颜色和目标颜色相同时必须直接返回。否则水会把起点再染一遍，然后再染回来，永远停不下来。
>
> **坑二**　染色要放在入队的那一刻，不要等出队再染。否则同一个格子可能被上下左右四个邻居各入队一次，队列里塞满重复项。
>
> **坑三**　别用递归写栈版。一两万格的区域就会把函数调用栈压爆，报 RecursionError；手写 list 当栈没有这个问题。
>
> **坑四**　四邻和八邻的结果不一样。八邻允许斜着走，斜对角也会连通；扫雷里点开一块空白，用的就是八邻。写之前先问清楚题目要不要斜着走。

第 4 章　经典算法：确定性问题的解法

> **读前提示**　需要第 0 章 0.2 的「列表、循环、递归」三个概念。本章不要求任何数学基础，只要能看懂「把数组分成两半」这种操作即可。

这一章的算法不「学习」，它们按确定的步骤把输入变成输出。它们的价值在于：可证明的正确性、可预测的复杂度、以及作为 AI 系统底层组件无处不在（分词、索引(像书的目录一样，帮你快速定位数据)、路由(决定数据或请求该走哪条路径的机制)、调度、检索都在用）。

4.1　快速排序 Quick Sort（1960，Hoare）

定位：工业界默认排序，平均 O(n log n)，原地、常数极小、缓存友好。

核心原理

分治 + 分区。选一个基准值 pivot，把数组分成「小于 pivot」和「大于 pivot」两段，然后递归处理两段。关键在于分区是原地完成的，不需要额外数组。

代码 4-1　原地快速排序（Hoare 双指针分区）

```
def quicksort(a, lo=0, hi=None):
    if hi is None:
        hi = len(a) - 1
    if lo >= hi:
        return a
    # 三数取中避免有序输入退化
    mid = (lo + hi) // 2
    pivot = sorted([a[lo], a[mid], a[hi]])[1]
    i, j = lo, hi
    while i <= j:
        while a[i] < pivot: i += 1
        while a[j] > pivot: j -= 1
        if i <= j:
            a[i], a[j] = a[j], a[i]
            i += 1; j -= 1
    quicksort(a, lo, j)      # 左段
    quicksort(a, i, hi)      # 右段
    return a
```

参数 / 符号	含义	取值与影响
a	待排序序列（原地修改）	需支持随机访问；链表上快排会退化，应改用归并排序

参数 / 符号	含义	取值与影响
lo / hi	当前处理区间的左右端点（闭区间）	初始 0 与 len-1；递归时按分区结果收缩
pivot	基准值，用来把区间一分为二	决定分区是否均衡。取到最值会退化为 $O(n^2)$；三数取中可规避
mid	三数取中的中间下标	sorted([a[lo],a[mid],a[hi]])[1] 取三者中位数，专治已排序输入
i / j	左右两个扫描指针，相向移动	i 停在第一個 ≥pivot 的元素，j 停在第一個 ≤pivot 的元素
while i <= j	分区循环的终止条件	必须用 <=；写成 < 会在有重复元素时漏掉交换，导致死循环

代码逐段解读：

- 第 1 层：if lo >= hi 是递归出口，区间长度 ≤1 天然有序，直接返回。
- 第 2 层：三数取中选 pivot，这是把「最坏 $O(n^2)$ 的概率降到接近 0」的廉价保险，代价只有三次比较。
- 第 3 层：分区双循环——内层两个 while 分别把 i 向右推、j 向左推，直到二者「错位」；每次错位前交换一对逆序元素。
- 第 4 层：递归顺序用 (lo, j) 与 (i, hi) 而不是 (lo, i-1)/(i+1, hi)，因为 Hoare 分区后 pivot 的最终位置并不确定，用 j/i 划分才能保证不漏元素。
- 复杂度来源：每轮分区是 O(区间长度)，递归深度期望 log n，故平均 O(n log n)；若 pivot 总取到最值，深度变 n，退化为 $O(n^2)$。

实现原理与性能要点

- 分区用双指针(用两个下标一前一后扫描，避免两层循环)相向扫描，一次线性扫描完成划分，递归深度期望 O(log n)。
- 最坏情况：每次选到最大/最小值作 pivot，退化为 $O(n^2)$；用三数取中或随机化 pivot 可规避。
- 工程实践（如 C++ std::sort、Python Timsort）会在小区间切换到插入排序，并对深度过深时回落堆排序。

维度	说明
解决的问题	把 n 个元素的无序序列排成有序序列，是检索、去重、聚合的前置步骤
核心原理	分治 + 原地分区：每轮把 pivot 放到最终位置，两侧递归
时间复杂度(数据量变大时，耗时会怎么涨)	平均 O(n log n)，最坏 $O(n^2)$，随机化后期望 O(n log n)

维度	说明
空间复杂度(运行时要额外占多少内存)	O(log n) 递归栈（原地分区，不额外开数组）
优点	常数极小、缓存命中率高、原地排序，实测通常快于归并
缺点	不稳定排序；pivot 选不好会退化；递归有栈溢出风险
典型应用	数据库 ORDER BY、搜索引擎结果排序、LLM 中 logits 的 top-k/top-p 截断前排序
关键参数 / 调参要点	小区间阈值（16—32 切插入排序）、pivot 策略、递归深度阈值、是否需要稳定排序

4.2　归并排序 Merge Sort（1945，von Neumann）

定位：稳定排序，复杂度稳定在 O(n log n)，是外部排序与并行排序的首选。

代码 4-2　归并排序（自顶向下）

```
def mergesort(a):
    if len(a) <= 1:
        return a
    mid = len(a) // 2
    left, right = mergesort(a[:mid]), mergesort(a[mid:])
    # 归并两个有序序列
    out, i, j = [], 0, 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:      # <= 保证稳定性
            out.append(left[i]); i += 1
        else:
            out.append(right[j]); j += 1
    return out + left[i:] + right[j:]
```

参数 / 符号	含义	取值与影响
a	待排序序列（返回新列表，非原地）	需要 O(n) 额外内存，这是与快排最大的工程差异
mid	二分的切分点	len//2；切得越均匀递归树越矮，总比较次数越少
left / right	已排好序的左右子序列	归并的前提是两段各自有序，这正是递归保证的
left[i] <= right[j]	合并时的比较符号	必须是 <= 而不是 <，等号决定「稳定性」——相等元素保持原相对顺序
out	合并结果缓冲	长度等于两段之和；生产实现会复用一块全局缓冲避免反复分配

代码逐段解读：

- 第 1 层：递归基 len<=1 直接返回，单元素天然有序。

- 第 2 层：先递归排好左右两半，再合并——这是典型的「分治」顺序（先分后治）。
- 第 3 层：合并用两个指针 i、j 同时扫描两个有序段，每次取较小的那个追加，O(n) 完成一次合并。
- 第 4 层：out + left[i:] + right[j:] 把剩余元素一次性接上，因为某一半耗尽后另一半必然全部更大且已有序。
- 为什么复杂度稳定：无论输入是否有序，递归树形状固定为 log n 层、每层总工作量 O(n)，所以最好/平均/最坏都是 O(n log n)。

维度	说明
解决的问题	需要稳定、可预测 O(n log n) 的排序，尤其是数据放不进内存时
核心原理	分治 + 有序段合并；比较只发生在合并阶段
时间复杂度	最好/平均/最坏均为 O(n log n)
空间复杂度	O(n) 辅助空间（外部排序时为磁盘 I/O）
优点	稳定、复杂度无退化、天然适合链表与外存、易并行
缺点	需要额外 O(n) 内存，缓存局部性不如快排
典型应用	大数据外部排序（MapReduce / Spark shuffle）、稳定排序需求、链表排序
关键参数 / 调参要点	归并路数（k-way）、内存缓冲区大小、是否用插入排序处理小段

> **对比小结** 要速度选快排（原地、常数小）；要稳定与可预测选归并；要最坏情况兜底选堆排序。生产环境普遍用混合策略（Timsort / introsort），把三者优点拼起来。

4.3 二分查找与哈希表

定位：从 O(n) 的线性扫描降到 O(log n) 或 O(1)，是所有检索系统的地基。

代码 4-3 二分查找（lower_bound）与开放寻址哈希表

```
def lower_bound(a, target):
    """返回第一个 >= target 的下标，找不到返回 len(a)"""
    lo, hi = 0, len(a)            # 左闭右开区间
    while lo < hi:
        mid = (lo + hi) // 2
        if a[mid] < target:
            lo = mid + 1
        else:
            hi = mid
    return lo

# 哈希：开放寻址（线性探测）示例
class HashTable:
    def __init__(self, cap=8):
```

```
        self.cap, self.size = cap, 0
        self.keys, self.vals = [None] * cap, [None] * cap

    def _probe(self, k):
        i = hash(k) % self.cap
        while self.keys[i] is not None and self.keys[i] != k:
            i = (i + 1) % self.cap            # 线性探测解决冲突
        return i

    def put(self, k, v):
        if self.size * 10 >= self.cap * 7:    # 负载因子 0.7 触发扩容
            self._resize()
        i = self._probe(k)
        if self.keys[i] is None:
            self.size += 1
        self.keys[i], self.vals[i] = k, v

    def get(self, k):
        i = self._probe(k)
        return self.vals[i] if self.keys[i] == k else None
```

参数 / 符号	含义	取值与影响
a / target	有序数组与目标值	数组必须有序；否则二分无意义
lo / hi	搜索区间 [lo, hi)，左闭右开	hi 初值为 len(a) 而非 len-1；右开区间让「找不到」自然返回 len(a)
mid	区间中点	(lo+hi)//2；某些语言需写成 lo+(hi-lo)//2 防溢出
if a[mid] < target	收紧左边界的条件	用 < 找下界（第一个 ≥target）；改成 <= 就变成找「第一个 >target」
cap	哈希表桶数组容量	容量越小冲突越多；通常是 2 的幂以便用位运算取模
负载因子(哈希表装得有多满，太满就容易冲突) 0.7	size/cap 的扩容阈值	工程常用 0.7—0.75。太高冲突激增退化成 O(n)，太低浪费内存

代码逐段解读：

- 二分部分：循环条件是 lo < hi 而不是 lo <= hi，这是左闭右开写法的标准搭配，能统一处理「找不到」的情形。
- 二分部分：找到的 lo 是「第一个 ≥ target 的下标」，这是 STL 的 lower_bound 语义，也是插入位置、区间起点查找的基础。
- 哈希部分：_probe 用线性探测解决冲突——被占了就往后挪一格，直到遇到空位或相同的 key。
- 哈希部分：put 先检查负载因子再决定扩容，这是避免「扩容太晚导致查询退化」的关键保护。

- 哈希部分：删除不能简单置 None（会截断探测链），必须打墓碑标记或改用链表法——这是开放寻址的经典坑。

维度	说明
解决的问题	在有序集合中快速定位；在无序集合中用 O(1) 完成 key→value 映射
核心原理	二分每次排除一半区间；哈希把 key 映射为数组下标，冲突用链表或探测解决
时间复杂度	二分 O(log n)；哈希平均 O(1)，最坏 O(n)（全冲突）
空间复杂度	二分 O(1)；哈希 O(n)（负载因子越低越费内存）
优点	二分零额外空间、可找边界；哈希读写极快，是缓存/索引/去重的核心
缺点	二分要求有序且支持随机访问；哈希有冲突退化、扩容抖动、内存开销
典型应用	数据库 B+ 树索引（页内二分）、字典/缓存、布隆过滤器、LLM 的 token(词元：模型处理文本的最小单位)izer 词表(模型认识的全部词元的清单)查表
关键参数 / 调参要点	哈希：负载因子（0.7 左右）、哈希函数质量、扩容倍数、冲突策略；二分：区间开闭写法

4.4 Dijkstra 最短路（1956）

定位：非负权图的单源最短路，导航、网络路由、依赖解析的标准解法。

> **一句话原理** 每一步都从「还没确定的最短距离」里挑出最小的那个点，然后拿它去更新邻居。

先把地图想成一堆城市和公路：每条路有个长度，问从家出发到各处最少要走多少公里。Dijkstra 的做法是贪心的——在地图上放一个「已知最短」的集合，一开始只有起点；每次从剩下的点里挑出当前距离最小的那个，宣布它的最短距离已经确定，然后用它去把邻居的距离往小里改。

凭什么这样做是对的？因为所有路长都是非负的。既然没有负数的边，那么「当前距离最小的那个未确定点」就不可能再有别的路能绕得更近——任何一条绕路都要再走一段正长度的边，只会更长。这条性质是整套算法成立的根基，也是它不能处理负权的原因。

- 第一步：把起点距离设为 0，其它点设为无穷大，全部放进待处理集合。
- 第二步：取出当前距离最小的点，把它标记为已确定。若它的距离还是无穷大，说明剩下的点都到不了，可以提前结束。
- 第三步：用它去「松弛」每条出边。如果绕经它能让邻居更近，就更新邻居的距离。
- 第四步：重复第二步和第三步，直到待处理集合为空。

朴素实现每轮扫一遍所有点找最小，是 $O(V^2)$；用优先队列（小顶堆）代替扫描，就是这段代码的堆优化版，复杂度降到 O((V+E) log V)。

> **读代码看这里**　为什么允许同一个点被推入堆多次、以及那一行 if d > dist[u]: continue 是在跳过什么。

代码 4-4　Dijkstra（堆优化版）

```
import heapq

def dijkstra(graph, src):
    """graph: {u: [(v, w), ...]}，边权非负"""
    dist = {src: 0}
    pq = [(0, src)]                          # 小顶堆，按距离出队
    visited = set()
    while pq:
        d, u = heapq.heappop(pq)
        if u in visited:                     # 过期条目，跳过
            continue
        visited.add(u)
        for v, w in graph.get(u, []):
            nd = d + w
            if nd < dist.get(v, float("inf")):
                dist[v] = nd
                heapq.heappush(pq, (nd, v))   # 松弛：发现更短路径就更新
    return dist
```

参数 / 符号	含义	取值与影响
graph	邻接表 {u: [(v, w), ...]}	稠密图也可用邻接矩阵，但稀疏图上邻接表省内存且遍历更快
src	源点	单源最短路的起点；全源最短路应跑 Floyd 或对每个点跑一次
dist	源点到各点的当前最短距离估计	初始化只有 dist[src]=0，其余视为 ∞
pq	小顶堆（优先队列）	按距离出队；堆优化是 Dijkstra 从 $O(V^2)$ 降到 $O((V+E)\log V)$ 的关键
visited	已确定最短距离的节点(图或树里的一个点)集合	防止重复扩展；这是「贪心选择性质」成立后的剪枝(提前砍掉明显没希望的路，省下计算)
松弛(发现更短路径时，更新距离估计值) nd = d + w	发现更短路径时的更新动作	仅当 nd < 已有距离才更新并入堆；这叫松弛（relaxation）

代码逐段解读：

- 第 1 层：堆里存 (距离, 节点) 的元组，Python 的 heapq 按第一个元素排序，所以距离小的先出队。
- 第 2 层：if u in visited: continue 是「惰性删除」技巧——同一个节点可能被多次入堆（每次发现更短距离都入一次），靠这个判断跳过过期条目，比手写「堆内减键」简单得多。

- 第 3 层：出队即确定——因为所有边权非负，堆顶那个距离最小的未确定节点不可能再被改进，这是 Dijkstra 正确性的核心。
- 第 4 层：遍历邻居做松弛，只把「确实变短了」的邻居重新入堆。
- 限制：一旦出现负权边，上面的「出队即确定」不再成立，必须改用 Bellman-Ford（可检测负环）或 SPFA。

实现原理

正确性来自「贪心选择性质」：当所有边权非负时，堆顶那个距离最小的未确定节点，其当前距离一定已经是最终最短距离，不可能再被更晚出队的节点改进。因此每个节点只松弛一次，配合堆可达到 O((V+E) log V)。

维度	说明
解决的问题	非负权图中从一点到所有点的最短路径（导航、网络时延最优路径）
核心原理	优先队列 + 松弛：每次取出距离最小的节点并更新其邻居
时间复杂度	堆优化 O((V+E) log V)；朴素数组 $O(V^2)$
空间复杂度	O(V) 距离表 + O(E) 堆
优点	实现简单、结果确定、可随时提前终止（找到目标即停）
缺点	不能处理负权边（需 Bellman-Ford）；稠密图上不如 Floyd 预处理
典型应用	地图导航、SDN 网络路由、知识图谱关系推理、LLM Agent 中的代价最优规划
关键参数 / 调参要点	堆类型（二叉堆/斐波那契堆）、是否双向搜索、是否需要还原路径（前驱数组）

4.5 A* 启发式搜索（1968）

定位：带启发函数的 Dijkstra，用「估计剩余代价」把搜索范围压向目标，是游戏 AI 与路径规划的事实标准。

代码 4-5 A* 搜索（h 为曼哈顿/欧氏距离等启发函数）

```
import heapq

def astar(start, goal, neighbors, h):
    """f(n) = g(n) + h(n)：已花代价 + 到目标的乐观估计"""
    openq = [(h(start), 0, start)]
    g = {start: 0}
    came = {}
    while openq:
        f, gc, u = heapq.heappop(openq)
        if u == goal:
            return reconstruct(came, u)          # 回溯路径
        if gc > g.get(u, float("inf")):
```

```
            continue
        for v, w in neighbors(u):
            ng = gc + w
            if ng < g.get(v, float("inf")):
                came[v] = u
                g[v] = ng
                heapq.heappush(openq, (ng + h(v), ng, v))   # h 为启发函数
    return None
```

参数 / 符号	含义	取值与影响
start / goal	起点与终点	A* 需要知道终点才能算启发值；不知终点时用 Dijkstra
neighbors(u)	返回节点 u 的邻居及边权	代价函数，地图上是移动距离，规划上是步骤代价
h(u)	启发函数：u 到目标的代价估计	h 的质量决定 A* 的效率；h=0 时退化为 Dijkstra
g[u]	起点到 u 的实际已知最小代价	对应 Dijkstra 里的 dist，是「已经花了多少」
f = g + h	评估函数，堆按它排序	堆顶是「看起来最有希望」的节点，这就是 A* 能把搜索聚焦到目标方向的原因
came	前驱字典，用于还原路径	每个节点记录从哪来，找到目标后回溯即可得到完整路径

代码逐段解读：

- 第 1 层：堆里存 (f, g, u)，先按 f 排序，f 相同再按 g（更靠近目标的优先），这是常见的 tie-breaking 技巧。
- 第 2 层：if gc > g[u]: continue 同样是惰性删除，跳过被后来更优路径取代的堆条目。
- 第 3 层：入堆时用 ng + h(v) 作为优先级——注意这里 h 是「对剩余路程的乐观估计」。
- 正确性条件：h 必须可采纳（admissible），即 h(u) 永远不超过 u 到目标的真实代价。曼哈顿距离在四向网格上可采纳，欧氏距离在可斜走时需放宽。
- 性能直觉：h 越接近真实剩余代价，A* 扩展的节点越少（极限情况沿最优路径直奔目标）；h 越大越激进、越快，但一旦高估就失去最优性——这就是 weighted A* 用 f=g+w·h（w>1）换取速度的取舍。

维度	说明
解决的问题	已知终点时，避免 Dijkstra 的「四面八方扩散」，把搜索聚焦到目标方向
核心原理	评估函数 f = g + h；h 可采纳（不高估真实代价）时 A* 保证最优
时间复杂度	取决于 h 的质量：h 越准越快，最坏仍为 O((V+E) log V)
空间复杂度	O(V)（open/closed 表），大规模场景是主要瓶颈
优点	比 Dijkstra 少扩展大量无用节点；h=0 时退化为 Dijkstra，正确性有兜底

维度	说明
缺点	h 设计不当会退化甚至失去最优性；内存占用高于双向 BFS
典型应用	游戏寻路、机器人运动规划、自动驾驶局部路径、LLM 推理中的搜索式解码（如思维树 ToT）
关键参数 / 调参要点	启发函数 h（可采纳性/一致性）、权重 w（weighted A* 用 f=g+w·h 换速度）、tie-breaking

4.6 动态规划 Dynamic Programming

定位：把「重叠子问题(计算过程中反复出现的同一个小问题) + 最优子结构(整体的最优解可以由局部的最优解拼出来)」的暴力指数级搜索，压成多项式时间。它是强化学习、序列对齐（Bio/ NLP）、LLM 解码中 Viterbi 与 beam search 的共同祖先。

代码 4-6　动态规划两例：0/1 背包与最长公共子序列

```
def knapsack(weights, values, W):
    """0/1 背包：容量为 W 时的最大价值"""
    n = len(weights)
    dp = [0] * (W + 1)                  # 一维滚动数组，空间 O(W)
    for i in range(n):
        wi, vi = weights[i], values[i]
        for c in range(W, wi - 1, -1):  # 倒序遍历，保证每件物品只用一次
            dp[c] = max(dp[c], dp[c - wi] + vi)
    return dp[W]

# 最长公共子序列（NLP 中的对齐 / diff 工具的基础）
def lcs(a, b):
    m, n = len(a), len(b)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if a[i - 1] == b[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
    return dp[m][n]
```

参数 / 符号	含义	取值与影响
weights / values	物品的重量数组与价值数组	一一对应，长度均为 n
W	背包容量（整数）	决定 dp 数组长度；容量很大时应改用「按价值」定义状态
dp[c]	容量不超过 c 时能取得的最大价值	一维滚动数组，是空间优化后的形态

参数 / 符号	含义	取值与影响
倒序 range(W, wi-1, -1)	容量维度的遍历方向	倒序保证每件物品只被用一次（0/1 背包）；正序就变成完全背包
a / b	求 LCS 的两个序列	可以是字符串、数组、token 序列
dp[i][j]	a 前 i 个与 b 前 j 个的 LCS 长度	二维状态，递归边界为 dp[0][*]=dp[*][0]=0

代码逐段解读：

- 背包第 1 层：dp 是一维数组而非二维，因为第 i 轮只依赖第 i-1 轮的结果，用滚动数组把空间从 O(nW) 压到 O(W)。
- 背包第 2 层：容量必须倒序遍历——若正序，同一个物品会在本轮被重复放入（刚更新的 dp[c-wi] 又会被自己用到），那就变成完全背包了。这是 DP 最经典的易错点。
- 背包第 3 层：转移 dp[c] = max(不选, 选)，「不选」保持原值，「选」是 dp[c-wi]+vi，取最大即最优子结构。
- LCS 第 1 层：状态是二维前缀对 (i, j)，因为两个序列都要逐步消耗。
- LCS 第 2 层：字符相等时 dp[i][j]=dp[i-1][j-1]+1（两个前缀同时缩短一位）；不等时取「砍掉 a 末位」或「砍掉 b 末位」的较大值。
- 复杂度：背包 O(nW)，LCS O(nm)。状态数 × 转移数就是 DP 的时间复杂度，这是估算任何 DP 的通用公式。

实现原理

- 三要素：状态定义、状态转移方程、边界与遍历顺序。写不出转移方程，就写不出 DP。
- 记忆化搜索（自顶向下）与递推（自底向上）等价；后者可用滚动数组把空间从 O(nW) 压到 O(W)。
- 遍历顺序是正确性关键：0/1 背包必须倒序容量，完全背包必须正序。

维度	说明
解决的问题	多阶段决策的最优解：背包、编辑距离、最优对齐、路径计数、资源分配
核心原理	用表格存储子问题答案，避免重复计算；转移方程刻画相邻阶段关系
时间复杂度	O(状态数 × 转移数)，典型 O(nW) 或 $O(n^2)$
空间复杂度	O(状态数)，可滚动数组优化
优点	把指数级降到多项式；思路一旦打通，代码极短
缺点	状态设计依赖经验；维度灾难（状态空间爆炸）时无法使用
典型应用	文本 diff / 拼写纠错（编辑距离）、语音识别对齐、RL 中的价值迭代（见第 6 章）、LLM 的 Viterbi 解码

维度	说明
关键参数 / 调参要点	状态维度、滚动数组压缩、单调队列/斜率优化等高级剪枝

4.7 贪心算法 Greedy

定位：每步都取局部最优，期望全局最优。能用则极快，但必须证明「贪心选择性质」，否则会翻车。

经典问题	贪心策略	是否最优	反例 / 注意
活动选择	按结束时间最早选	✓ 最优	若按开始时间选则错误
哈夫曼编码	每次合并权重最小的两棵子树	✓ 最优	也是 LLM 中 BPE 分词合并的思想来源
0/1 背包	优先装价值密度最高的	✗ 非最优	必须用 DP
最短路径	每步取最近邻（最近邻启发）	✗ 非最优	TSP 近似解，需 2-opt 改进
区间覆盖	选覆盖点最多的区间	✓ 最优	需先按左端点排序

维度	说明
解决的问题	在具备贪心选择性质的问题上，用 O(n log n) 拿到最优解
核心原理	局部最优 + 不可回退；正确性靠交换论证或拟阵理论证明
时间复杂度	通常 O(n log n)（多一次排序）
空间复杂度	O(1) 至 O(n)
优点	极快、实现简单、常数小，适合在线/流式场景
缺点	适用范围窄；未证明就使用极易得到错误答案
典型应用	哈夫曼/BPE 分词、任务调度、最小生成树（Prim/Kruskal）、区间调度
关键参数 / 调参要点	排序键的选择（决定成败）、是否需要事后局部搜索改进

4.8 布隆过滤器 Bloom Filter（1970）

定位：用极小内存回答「某元素是否可能存在」，允许假阳性(本来没有，却报告说有)、绝无假阴性(本来有，却报告说没有)。数据库与爬虫的标配。

代码 4-7　布隆过滤器

```
import mmh3  # 若无该库可用 hashlib 代替

class BloomFilter:
```

```
    def __init__(self, m, k):
        self.m, self.k = m, k        # m: 比特数; k: 哈希函数个数
        self.bits = bytearray(m // 8 + 1)

    def _hashes(self, s):
        for i in range(self.k):
            yield mmh3.hash(s, i) % self.m

    def add(self, s):
        for i in self._hashes(s):
            self.bits[i // 8] |= 1 << (i % 8)

    def __contains__(self, s):
        return all(self.bits[i // 8] & (1 << (i % 8)) for i in self._hashes(s))
# 误判率 ≈ (1 - e^(-kn/m))^k；最优 k = (m/n)·ln2
```

参数 / 符号	含义	取值与影响
m	比特数组的总位数	决定误判率；每个元素分到越多位，误判率越低
k	哈希函数的个数	k 太少冲突多，k 太多则每个元素污染太多位，反而抬高误判率
bits	字节数组实现的位图(用一个二进制位表示「有」或「没有」的结构)	bytearray(m//8+1)，用位运算置位与读取
mmh3.hash(s, i)	第 i 个哈希函数的值	用种子 i 区分不同哈希；要求各函数独立且分布均匀
误判率公式	(1 - e^(-kn/m))^k	给定 n 个元素时，最优 k = (m/n)·ln2，此时误判率最低
9.6 bit/元素	达到 1% 误判率所需的位数	对比 HashSet 存一个 key 需几十至上百字节，省内存 90% 以上

代码逐段解读：

- 第 1 层：_hashes 用同一个哈希函数配不同种子生成 k 个值，这是工程上最省事的做法，独立性足够好。
- 第 2 层：add 把 k 个位置全部置 1——注意只置 1 不置 0，所以多个元素会叠加，这正是「假阳性」的来源。
- 第 3 层：__contains__ 检查 k 个位是否全为 1。全 1 只能说明「可能存在」，因为可能是别的元素置的；但只要有一位是 0，就能确定「一定不存在」——这就是「无假阴性」。
- 为什么不能删除：把某一位清 0 可能同时清掉其他元素留下的标记，导致假阴性。要支持删除需改用 Counting Bloom（把位换成计数器）。

- 调参直觉：先定能接受的内存（决定 m/n），再按 k=(m/n)·ln2 算 k。例如 m/n=10 时 k≈7，误判率约 1%。

维度	说明
解决的问题	海量集合的「存在性」判断：挡掉绝大多数无效查询，保护后端存储
核心原理	k 个哈希把元素映射到 m 个比特位；全为 1 则「可能存在」，有 0 则「一定不存在」
时间复杂度	插入/查询 O(k)，与集合大小无关
空间复杂度	约 9.6 bit/元素 即可达到 1% 误判率（相比 HashSet 省 90%+ 内存）
优点	内存极省、查询常数时间、无假阴性，可分布式重建
缺点	有假阳性；不支持删除（需 Counting Bloom）；无法枚举元素
典型应用	数据库读穿透防护（Redis/LSM）、爬虫 URL 去重、LLM 检索前的候选粗筛
关键参数 / 调参要点	m/n（比特/元素，决定误判率）、哈希个数 k、哈希函数质量、是否需要计数变体

4.9 PageRank（1997，Google）

定位：把「网页重要性」变成图的稳态分布(反复跳转后不再变化的比例)问题，是幂迭代(反复用矩阵乘向量，直到结果不再变化)法的经典案例，也是今天图算法与 GNN 的思想源头。

代码 4-8　PageRank 幂迭代

```
import numpy as np

def pagerank(adj, damping=0.85, iters=50, tol=1e-8):
    """adj: 邻接矩阵（列归一化后的转移矩阵）"""
    n = adj.shape[0]
    pr = np.full(n, 1.0 / n)                          # 初始化均匀分布
    teleport = np.full(n, (1 - damping) / n)          # 随机跳转项
    for _ in range(iters):
        new = damping * (adj @ pr) + teleport         # 幂迭代
        if np.linalg.norm(new - pr, 1) < tol:
            break
        pr = new
    return pr
```

参数 / 符号	含义	取值与影响
adj	列归一化后的转移矩阵	adj[i][j] 表示从 j 跳到 i 的概率，每列之和为 1
damping	阻尼因子 d	0.85 为经典值：以 1-d 的概率随机跳转到任意节点，避免陷入死胡同

参数 / 符号	含义	取值与影响
pr	各节点的 PageRank 值（初始均匀分布）	迭代的起点；收敛结果与初值无关，只影响收敛速度
teleport	随机跳转项 (1-d)/n	加在每个节点上，保证图不连通时也能收敛
iters / tol	最大迭代轮数与收敛阈值	tol 用 L1 范数判断；通常 30—60 轮收敛
adj @ pr	一次矩阵—向量乘法	等价于所有节点同时做一次随机游走，这就是幂迭代

代码逐段解读：

- 第 1 层：初始化为均匀分布 1/n，表示「一开始认为所有网页同样重要」。
- 第 2 层：核心迭代式 new = d·(adj @ pr) + teleport。第一部分是按链接结构传播权重，第二部分是随机跳转。
- 第 3 层：用 L1 范数判断收敛——两轮之间所有节点的变化量之和小于 tol 就停。
- 为什么需要阻尼因子：没有它，遇到「只进不出」的节点（dangling node）或环形结构，权重会卡住或震荡不收敛。
- 本质：PageRank 就是马尔可夫链的稳态分布，幂迭代是在求转移矩阵主特征值为 1 对应的特征向量。这个思想后来被 GNN 的消息传播和 RL 的价值迭代直接继承。

维度	说明
解决的问题	在大规模有向图(连线带方向的节点关系网)中度量节点重要性（无监督、无标注）
核心原理	随机游走的稳态分布；阻尼因子保证收敛并避免陷阱节点
时间复杂度	每轮 O(E)，通常 30—60 轮收敛
空间复杂度	O(V)
优点	全局一致、可解释、可分布式计算
缺点	对结构稀疏与新节点不友好；稠密图上收敛慢
典型应用	搜索引擎排序、社交网络影响力、推荐召回、风控团伙识别、GNN 的消息传播思想
关键参数 / 调参要点	阻尼因子（0.85）、收敛阈值、是否用稀疏矩阵/分块并行

4.10　排序家族全览：七种排序放在一起看

前面只详细讲了快速排序和归并排序，但工程里遇到的排序问题远不止这一种。判断该用哪种，只要回答四个问题：数据量多大、数据是否近似有序、元素能否用整数表示、是否要求稳定。下面这张表把七种常见排序放在一起。

排序	平均	最坏	空间	稳定	什么时候用

排序	平均	最坏	空间	稳定	什么时候用
冒泡排序	$O(n^2)$	$O(n^2)$	O(1)	是	只适合教学与近乎有序的小数组
选择排序	$O(n^2)$	$O(n^2)$	O(1)	否	交换次数要最少时（如写 Flash）
插入排序	$O(n^2)$	$O(n^2)$	O(1)	是	小数组、近乎有序；工业库的小数组分支
希尔排序	O(n^1.3)	$O(n^2)$	O(1)	否	中等规模、不想递归、代码要短
归并排序	O(n log n)	O(n log n)	O(n)	是	要求稳定、链表排序、外部排序
快速排序	O(n log n)	$O(n^2)$	O(log n)	否	通用最快、内存受限、缓存友好
堆排序	O(n log n)	O(n log n)	O(1)	否	最坏情况也要保证 O(n log n) 且不许额外空间
计数排序	O(n+k)	O(n+k)	O(k)	是	数据是有限范围整数（如年龄、分数）
基数排序	O(d(n+k))	同左	O(n+k)	是	定长整数、字符串、IP 排序
桶排序	O(n)	$O(n^2)$	O(n)	是	数据均匀分布，能接受假设

> **提示：**稳定是指值相等的两个元素，排序后相对次序不变。需要按多个字段排序时（先按分数、分数相同按提交时间），必须用稳定排序，或者把比较函数写成多关键字比较。

工业界真正的排序实现是混合的。Python 用的是 Timsort：它先扫描数据找出已经有序的短片段，再用插入排序整理每个片段，最后用归并合并——所以在近乎有序的数据上，它比理论上的快排还快，这也是它能做到稳定且 O(n log n) 的原因。C++ 的 std::sort 用的是内省排序：快排为主，递归太深时切换堆排序，小数组切换插入排序。

从复杂度这一列还能看出一个通用技巧：只要比较两个元素的大小，排序的最坏下界就是 O(n log n)，所以想更快，就必须放弃「比较」这个前提——计数排序、基数排序、桶排序正是靠这个思路绕开下界。

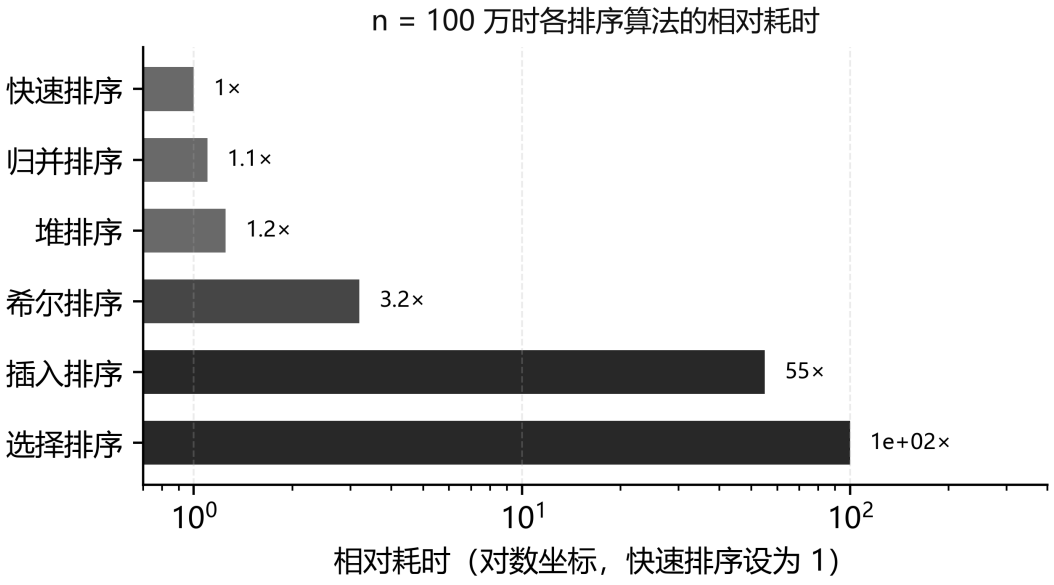


图 4-1　n = 100 万时各排序算法的相对耗时（快速排序设为 1）

4.11　字符串匹配：从暴力到 KMP

字符串匹配要解决的问题是：在一段长文本里找出模式串出现的位置。它在编辑器查找、日志检索、DNA 序列分析、敏感词过滤里每天都在被调用。

最直接的做法是把模式串对齐到每个位置逐个比较，最坏 O(n × m)：当文本是「aaaaaaaab」、模式串是「aaab」时，每对齐一次都要比到最后才发现不对，前面的信息全浪费了。KMP 的价值就在于把这些信息利用起来：匹配失败时，模式串究竟能往前滑多少，完全由模式串自身的结构决定，与文本无关。这个结构就是前缀函数。

代码 4-9　KMP：前缀函数 + 匹配，整体 O(n + m)

```
def prefix_function(p):
    pi = [0] * len(p)              # pi[i] = p[0..i] 的最长相等前后缀长度
    k = 0
    for i in range(1, len(p)):
        while k > 0 and p[i] != p[k]:
            k = pi[k - 1]          # 回退到次长的候选
        if p[i] == p[k]:
            k += 1
        pi[i] = k
    return pi

def kmp_search(text, pat):
    if not pat:
```

```
        return []
    pi = prefix_function(pat)
    res, k = [], 0
    for i, ch in enumerate(text):
        while k > 0 and ch != pat[k]:
            k = pi[k - 1]          # 不匹配就跳，而不是从头再来
        if ch == pat[k]:
            k += 1
        if k == len(pat):
            res.append(i - len(pat) + 1)   # 找到一个匹配
            k = pi[k - 1]                  # 继续找下一个
    return res

print(kmp_search("abababcabab", "abab"))   # [0, 2, 6]
```

算法	预处理	匹配	特点
暴力匹配	无	O(n × m)	代码最短，实际常常够用
KMP	O(m)	O(n)	最坏也是线性，只需知道模式串
Rabin-Karp	O(m)	平均 O(n)	用滚动哈希，适合多模式与去重
Boyer-Moore	O(m + 字母表)	平均亚线性	从右往左比，实际最快，grep 曾经的主力
Aho-Corasick	O(总长)	O(n + 命中数)	多模式同时匹配，敏感词过滤的事实标准

> **注意：**实际写代码时优先用语言内置的查找（如 Python 的 str.find）。它底层通常是经过高度优化的 Boyer-Moore 变体，比手写 KMP 更快。手写 KMP 的意义在于理解「如何利用已匹配信息」这一思路，它是理解 AC 自动机与后缀数组的基础。

4.12 图算法补充：最小生成树、负权最短路与全源最短路

Dijkstra 解决了单源最短路，但有三个它做不到的事，需要另外三个算法补上。

第一件是不能处理负权边。Dijkstra 的贪心前提是「已经确定的最短距离不会再变短」，一旦有负边，这个前提就崩了。解决办法是 Bellman-Ford：把所有边松弛 V-1 轮，第 V 轮如果还能松弛，说明存在负环。

第二件是「用最少的电缆把所有城市连起来」。这类问题叫最小生成树（MST），两个经典解法是 Kruskal（按边权从小到大加边，用并查集判断是否成环）和 Prim（从一点出发，每次加入离树最近的节点，用堆维护）。前者适合稀疏图，后者适合稠密图。

代码 4-10　Kruskal：并查集 + 排序，O(E log E)

```
def kruskal(n, edges):
    res, total = [], 0
    p = list(range(n))
    def find(x):
        while p[x] != x:
            p[x] = p[p[x]]
            x = p[x]
        return x
    for w, u, v in sorted(edges):        # 按边权从小到大
        ru, rv = find(u), find(v)
        if ru != rv:                     # 不在同一棵树里才加入
            p[ru] = rv
            res.append((u, v, w))
            total += w
            if len(res) == n - 1:        # 已经够了：最小生成树有 n-1 条边
                break
    return res, total
```

第三件是「所有点对之间的最短路」。如果只要结果、不在乎中间过程，Floyd-Warshall 只有五行代码，代价是 $O(V^3)$，适合点数在几百以内的稠密图；如果图是稀疏的，对每个点跑一次 Dijkstra 反而更快。

问题	算法	复杂度	前提与说明
单源最短路（非负权）	Dijkstra + 堆	O((V+E) log V)	不能有负边
单源最短路（可有负权）	Bellman-Ford	O(V × E)	能检测负环
全源最短路（点数少）	Floyd-Warshall	$O(V^3)$	五行代码，常数小
最小生成树（稀疏图）	Kruskal	O(E log E)	需要并查集
最小生成树（稠密图）	Prim + 堆	O(E log V)	从任意点出发
有向无环图上的路径	拓扑排序 + 递推	O(V+E)	比 Dijkstra 更快，能处理负权

4.13　六种通用算法设计范式

前面每一节讲的都是具体的算法，但真正能迁移到新问题上的，是下面六种设计范式。遇到陌生题目时，先想它属于哪一类。

第一，分治：把问题拆成规模更小的同类子问题，分别解决再合并。归并排序、快速排序、二分查找、大整数乘法、归并求逆序对都是它。判断能否分治的关键是「子问题的解能不能合起来」。

第二，二分答案：当问题问的是「最小的最大」「能否在 x 天内完成」这类单调问题时，不去推导公式，而是二分答案本身，再用一个判定函数检查可行性。真正的难点在于写出单调的判定函数，二分本身反而简单。

代码 4-11　二分答案模板：找最小的可行值

```
def solve(lo, hi, ok):
    while lo < hi:
        mid = (lo + hi) // 2
        if ok(mid):          # mid 可行，答案可能是它或更小
            hi = mid
        else:
            lo = mid + 1
    return lo                # 第一个可行的值
```

第三，双指针与滑动窗口：当数组有序，或者问题要求「连续的一段」时，用一个窗口的两个端点代替两层循环，把 $O(n^2)$ 降到 $O(n)$。典型题目是最长无重复子串、和大于等于目标的最短子数组、三数之和。

配套仿真：搜索「滑动窗口」，可以看到左右两个指针怎样只往右走、各自最多走 n 步，右侧实时报出当前窗口的和与最短长度。想知道这条套路什么时候会失效，把数组里的数改成有正有负再跑一次：窗口不再能单调收缩，答案也就错了，这正是它成立的前提。

第四，前缀和与差分：把「区间求和」变成两个前缀和相减，O(1) 得到答案；把「区间整体加一」变成差分数组的两点修改，最后做一次前缀和还原。两者互为逆运算，是处理离线区间问题的标准工具。

代码 4-12　前缀和与差分

```
def build_prefix(a):
    pre = [0] * (len(a) + 1)
    for i, v in enumerate(a):
        pre[i + 1] = pre[i] + v
    return pre

def range_sum(pre, l, r):         # 闭区间 [l, r] 的和，O(1)
    return pre[r + 1] - pre[l]

def range_add(diff, l, r, v):     # 给 [l, r] 整体加 v，O(1)
    diff[l] += v
    diff[r + 1] -= v              # 还原时做一次前缀和即可
```

第五，回溯：本质是带撤销的深度优先搜索，适合「枚举所有方案」。写回溯只要记住三件事：选择、递归、撤销。再加上剪枝，就能把不可能的分支提前砍掉，这就是 N 皇后、数独、全排列能跑得动的原因。

第六，位运算：把状态压缩成二进制位，用与、或、异或、移位一次处理一批信息。常见技巧包括用 x & (x - 1) 消掉最低位的 1 来数二进制中 1 的个数、用 x & -x 取出最低位的 1（树状数组的基础）、用异或找出唯一出现一次的数。

范式	识别标志	复杂度收益	代表问题
分治	能拆成同类子问题并可合并	常常降到 O(log n) 或 O(n log n)	归并排序、大整数乘法
二分答案	问「最小的最大」「最多/最少」且可判定	O(n log range)	分割数组、装载问题
双指针/滑窗	已排序，或要求连续子段	$O(n^2) \to O(n)$	最长无重复子串
前缀和/差分	大量区间查询或区间修改	O(n) 预处理换 O(1) 查询	区间求和、区间覆盖
回溯	要求列出所有方案	指数级，靠剪枝救	全排列、N 皇后、数独
位运算	状态是布尔集合，规模小	常数级加速，状态压缩	位图、状压动态规划

4.14　本章常见错误与练习

新手在这一章最容易踩的六个坑。

一是把稳定性当小事。用不稳定排序去做多关键字排序，结果时好时坏，而且只在特定数据量下暴露，极难排查。

二是快速排序固定取第一个元素当基准。数据已经有序时会退化成 $O(n^2)$，甚至递归栈溢出。正确做法是随机取基准，或三数取中。

三是二分查找写不清区间定义。要么左闭右开、要么左闭右闭，选定一种就一直用；循环条件、mid 的取整方向、更新方式三者必须自洽，否则就是死循环或漏解。

四是忘记二分的前提是有序。在无序数据上用二分，结果随机正确。

五是 Dijkstra 里用了负权边，或者忘了「出堆时再判断是否已确定」，导致重复处理、复杂度退化。

六是并查集只写了 find 而忘了路径压缩和按秩合并，在极端数据上退化成链表。

三道练习：第一题，手写一个稳定排序并验证它对「先按分数、再按提交时间」的两级排序是否正确。第二题，用 KMP 找出一个字符串中所有既是前缀又是后缀的子串。第三题，用 Kruskal 求出一张图的最小生成树，并说明为什么它是「最小」的（提示：考虑切割性质）。

4.15　拓扑排序、最小生成树与单调栈

本节补上三个「一学就会、一用就灵」的经典技巧。它们的共同特点是：代码不到二十行，但几乎每一场算法面试、每一个工程系统里都能遇到——构建系统要排依赖顺序

（拓扑排序），网络布线要最省成本（最小生成树），行情系统要在线求「下一个更大的数」（单调栈）。

4.15.1 拓扑排序：给有依赖关系的任务排一个合法顺序

先看一个每天都在发生的问题：Make、Bazel、npm、pip 在安装包之前，必须先决定「谁先装」。如果把每个包看成一个点、把「A 依赖 B」看成一条从 A 指向 B 的有向边，那么要回答的问题就是：能不能把所有点排成一列，使每条边都从前指向后？这样的一列就叫拓扑序（topological order）。

能这么排的前提是这张图里没有环。如果 A 依赖 B、B 又依赖 A，那谁也不能先装，这就是「循环依赖」，构建系统报错时说的就是这件事。所以拓扑排序同时也是一个环检测算法。

最常用的写法是 Kahn 算法（1962）。它只有三步：先数出每个点有多少条边指向它（入度）；把入度为 0 的点全部放进队列；然后每次从队列取出一个点放进结果，把它指向的每个点的入度减 1，减到 0 就入队。如果最后排出来的点少于总点数，说明剩下的点互相依赖，图里有环。

```
from collections import deque

def topo(n, edges):
    g = [[] for _ in range(n)]
    indeg = [0] * n
    for u, v in edges:          # 一条边 u -> v 表示「u 必须在 v 之前」
        g[u].append(v)
        indeg[v] += 1
    q = deque(i for i in range(n) if indeg[i] == 0)
    order = []
    while q:
        u = q.popleft()
        order.append(u)
        for v in g[u]:
            indeg[v] -= 1
            if indeg[v] == 0:   # 它前面的依赖都处理完了
                q.append(v)
    return order if len(order) == n else None   # None 表示存在环
```

还有第二种写法：对图做一次深度优先搜索，一个点的所有后继都访问完之后，把它压进栈；全部搜完之后把栈倒过来，就是拓扑序。为什么对？因为 DFS 保证了「一个点一定在它所有后继之后入栈」，倒过来自然就是「在它所有后继之前」。这种写法更短，但需要额外的状态数组来标记「正在访问中」的点，用来发现回边。

对比项	Kahn（BFS 式）	DFS 逆后序
思路	反复摘掉入度为 0 的点	后序遍历完再倒序
环检测	天然支持：排不满就是有环	需要额外的三色标记
输出稳定性	稳定，同一张图结果固定	依赖遍历顺序

对比项	Kahn（BFS 式）	DFS 逆后序
复杂度	O(V + E)	O(V + E)
典型用途	构建系统、包管理器、课程表	编译器里的指令排序

一个容易忽略的细节：拓扑序通常不唯一。如果同时有多个入度为 0 的点，先取哪一个都可以，结果都是合法的。想要字典序最小的拓扑序，把队列换成小顶堆即可（代价是多一个 log V）。

4.15.2　最小生成树：用最少的钱把所有的点连通

第二个问题来自网络规划：有若干城市，城市之间可以铺网线，每段网线的造价不同，怎样铺才能让所有城市连通而总造价最低？从图的角度看，我们要在图中选出一个连通所有点的子图，使得边权和最小——这就是最小生成树（Minimum Spanning Tree, MST）。

为什么叫「树」？因为如果选出的子图里还有环，环上任意去掉一条边仍然连通，总权只会更小。所以在最优解里不可能有环，n 个点的连通无环图正好有 n − 1 条边，这就是一棵树。

最经典的两个算法都以「贪心」为核心，只是贪心的对象不同。

Kruskal（1956）贪心的是边：把所有的边按权重从小到大排序，依次考察，只要这条边连接的两个点目前还不连通，就选它。判断「是否已经连通」用一个并查集，几乎 O(1)。它的复杂度主要花在排序上，是 O(E log E)。

Prim（1957）贪心的是点：从一个点出发，每次从「一端已经在树里、另一端还在树外」的边中挑一条最小的，把外面的那个点拉进来。用堆优化后是 O(E log V)，在稠密图（边特别多）上通常比 Kruskal 快。

```
# Kruskal：排序 + 并查集，适合稀疏图
def kruskal(n, edges):                      # edges = [(w, u, v), ...]
    par = list(range(n))
    def find(x):
        while par[x] != x:
            par[x] = par[par[x]]            # 路径压缩
            x = par[x]
        return x
    total, used, mst = 0, 0, []
    for w, u, v in sorted(edges):           # 按权重从小到大
        a, b = find(u), find(v)
        if a == b:                          # 已经连通，选它会成环
            continue
        par[a] = b                          # 合并两棵树
        total += w
        mst.append((w, u, v))
        used += 1
        if used == n - 1:                   # 已经够了，提前结束
            break
```

```
    return total, mst
```

对比项	Kruskal	Prim
贪心对象	边（全局最小且不成环）	点（离树最近的点）
需要的数据结构	排序 + 并查集	优先队列（堆）
复杂度	O(E log E)	O(E log V)
适合的图	稀疏图、边表形式	稠密图、邻接矩阵形式
是否能处理不连通图	会得到「最小生成森林」	只覆盖起点所在的连通块

一个重要的边界情况：如果图本身不连通，就根本不存在生成树。这时两个算法都会「提前停下来」——Kruskal 选不满 n − 1 条边，Prim 拉不进所有的点。工程代码里必须显式检查这一点，否则会得到一个看起来正常的错误结果。

4.15.3 单调栈：把「找下一个更大的数」压到 O(n)

第三个技巧解决一类反复出现的问题：给一个数组，对每个位置求出「它右边第一个比它大的数」。暴力的做法是对每个位置往后扫，最坏 $O(n^2)$。单调栈能在 O(n) 做完。

核心想法是：如果 a[i] 比 a[j] 大（且 i 在 j 右边），那么对于 j 左边的任何位置，a[j] 永远不可能是答案——它被 a[i] 挡住了。既然永远用不上，就可以在遇到 a[i] 时把 a[j] 弹出栈。于是栈里存的下标，它们对应的值从栈底到栈顶保持单调（本题是递减）。当新元素要进栈时，凡是比它小的栈顶元素都已经找到了答案，依次弹出并记录。

以「每日温度」为例：给定一个温度数组，问每天要等几天才会遇到更高的温度。

```
def daily_temperatures(t):
    n = len(t)
    ans = [0] * n
    st = []                                  # 存下标，温度从栈底到栈顶递减
    for i, v in enumerate(t):
        while st and t[st[-1]] < v:          # v 就是栈顶那些位置要等的答案
            j = st.pop()
            ans[j] = i - j
        st.append(i)                         # 栈里剩下的都比 v 大，将来再说
    return ans

print(daily_temperatures([73, 74, 75, 71, 69, 72, 76, 73]))
# [1, 1, 4, 2, 1, 1, 0, 0]
```

每个下标最多进栈一次、出栈一次，所以总操作数是 2n，O(n)。这个模式非常通用，把「大小关系」和「要找什么」换掉，就能解一大类题。

题目	栈的单调方向	要找的答案
每日温度	从栈顶到栈底递减	右边第一个更大的数
柱状图最大矩形	递增	左右两边第一个更小的数

题目	栈的单调方向	要找的答案
接雨水	递减	两侧更高的柱子围成的凹槽
滑动窗口最大值	双端队列递减，且掐掉过期	窗口内的最大值
下一个更大元素（循环数组）	递减，数组走两遍	环形意义上的下一个更大元素

以「柱状图最大矩形」为例，思路是：对每一根柱子，分别找到左边和右边第一个比它矮的柱子，两者之间的宽度乘上这根柱子的高度就是一个候选面积，取最大值即可。这两次「找第一个更矮的」正是两次单调栈。为了避免处理边界，惯用做法是在数组首尾各补一个高度为 0 的哨兵柱子。

> **一句话记住**：单调栈解的是「每个元素左右两边第一个比它大/小的元素」这类问题；只要题目里出现「下一个更大」「持续多久」「能扩展到多远」这些字眼，先想想单调栈。

4.15.4　三个技巧的边界与常见错误

这三个算法都很短，但每一个都有典型错误。

- 拓扑排序最常见的错误是忘了检查环。如果构建系统给出的顺序少了几项，用户看到的只是「装不上」，真正的原因藏在没有输出来的那些点里。
- 最小生成树在 C++ 里写并查集时，如果忘了路径压缩，最坏情况下 find 会退化成 O(n)，整体复杂度就退回到 O(E log E + EV)。并查集的两个优化（按秩合并与路径压缩）至少要写一个。
- 单调栈最常见的错误是用值而不是下标入栈。一旦栈里存的是值，就无法计算「距离」和「宽度」，也处理不了重复元素。记住：单调栈里存的永远是下标。
- 三个算法都不需要额外的复杂数学，能徒手写出来是基本要求。建议至少把 Kanh 拓扑排序、Kruskal 和每日温度默写到不看提示的程度。

配套仿真：这三个算法在「算法动画仿真实验室」里都有独立的可视化——搜索「拓扑排序」可以看到入度如何一步步降低，搜索「最小生成树」可以看到边如何按权重被考察、什么时候因为成环被跳过，搜索「单调栈」可以看到元素进出栈的完整过程以及答案是怎么被填上的。每个仿真都提供 Python、C++、C、Java、Go 五种语言的完整可运行代码，可以对照本节的手写实现逐行核对。

4.16　霍夫曼编码：贪心的样板工程

有一类问题要解决的不是排序也不是查找，而是「用尽量短的比特串表示一批符号」。英文里 26 个字母的出现频率差得很远，如果一律用 5 位定长编码，等于每个字符都按最坏情况付钱。能不能让常用的字符短一点、罕见的字符长一点？

代价是不能随便编。假设 a 编成 0、b 编成 01，那么看到 01 时无法判断它到底是 b，还是「a 后面跟了一个 1」。所以有一条硬性要求：任何一个字符的编码都不能是另一个字符编码的前缀。满足这条性质的编码叫前缀码，它可以被唯一地解码：从头读，每读到一个合法编码就切一刀，不会出现两种切法。

1952 年，霍夫曼（David A. Huffman）给出了最优解，思路是自底向上地合并：把每个字符看成一棵只有一个结点的树，权重取它的出现频率；反复取出权重最小的两棵树，合成一棵新的树（新树的权重是两棵树权重之和），直到只剩一棵。最后从根往叶走，向左记 0、向右记 1，每个字符的编码就出来了。

为什么这样得到的就是最优的？频率最低的两个字符被最先合并，之后每上一层路径就长一位，所以它们必然落在最深处；频率最高的字符被留到最后合并，路径最短。任何一棵最优的前缀码树，都可以通过交换兄弟子树调整成这个形状，所以这个贪心选择不会损失最优性。这正是 4.7 讲的「贪心选择性质」在这里成立的原因。

代价是 O(n log n)，n 是不同字符的个数：每次取出最小元素要用优先队列（3.7 讲过的小顶堆），一共做 2n−1 次合并。下面用一组频率算一遍，左边是定长编码，右边是霍夫曼编码：

字符	频率	霍夫曼编码	编码长度
a	45	0	1 位
b	13	101	3 位
c	12	100	3 位
d	16	111	3 位
e	9	1101	4 位
f	5	1100	4 位

按这组编码，100 个字符的加权总长度是 45×1 + (13+12+16)×3 + (9+5)×4 = 224 位；同样这组数据用定长 3 位编码要 300 位。省下来的 25% 就是「让常用字符走短路」的全部回报。这也是它被用在 ZIP、PNG、JPEG 和 HTTP/2 头部压缩这些地方的原因。

代码 4-13　霍夫曼编码：优先队列 + 合并 + 遍历取码

```
import heapq

def huffman(freq):
    # 堆里的每一项: [权重, 序号, 叶子字符, (左子树, 右子树)]
    heap = [[w, i, ch, None] for i, (ch, w) in enumerate(freq.items())]
    heapq.heapify(heap)
    order = len(heap)
    while len(heap) > 1:
        lo = heapq.heappop(heap)          # 权重最小
        hi = heapq.heappop(heap)          # 权重次小
        node = [lo[0] + hi[0], order, None, (lo, hi)]
        order += 1
        heapq.heappush(heap, node)

    code = {}
    def walk(t, prefix):
        if t[2] is not None:              # 叶子：存着字符
            code[t[2]] = prefix or "0"      # 只有一个字符时输出 "0"
```

```
            return
        walk(t[3][0], prefix + "0")        # 左孩子记 0
        walk(t[3][1], prefix + "1")        # 右孩子记 1

    walk(heap[0], "")
    return code

freq = {"a": 45, "b": 13, "c": 12, "d": 16, "e": 9, "f": 5}
code = huffman(freq)
avg = sum(freq[c] * len(code[c]) for c in code) / sum(freq.values())
print(code)
print("加权平均码长: %.2f 位" % avg)
```

> **几个易错点**　一是把「字符的出现次数」当成权重，却用没归一化的频率去算平均码长，两个数对不上。二是取最小值时用普通队列或列表扫描而不是堆，复杂度从 O(n log n) 退化成 O(n²)，数据一大就明显变慢。三是没用序号做第二关键字，两个权重相同的结点比较时报「不可比较」的错。四是整个输入只有一种字符时，编码是空串，解压时陷入死循环，所以代码里要退化成一位的 0。

配套仿真：搜索「霍夫曼编码」，可以看到频率表里的两个最小值怎样被反复挑出来合并成一棵新树，树的形状随时间一点点长高，最后一列同步显示每个字符的编码和加权码长。把频率改一改再跑，就能看出「谁短谁长」完全由频率决定。

4.17　基数排序：不比较也能排序

4.10 那张排序对照表留下了一个结论：只要算法靠「比较两个元素的大小」来决定先后，最坏情况下的下界就是 O(n log n)。想更快，就必须放弃比较本身。

基数排序（Radix Sort）的做法是把键拆成若干位，从最低位开始，每一位都按这一位的值做一次稳定排序。做完最后一位，整个序列就按数值有序了。

低位先排为什么能得到正确结果？关键在于每一趟排序都必须是稳定的：值相同的元素保持原有的相对顺序。这样，高位排序的结果会覆盖低位排序的结果，而低位建立起来的顺序在「高位相同」的那一批元素里被完整保留了下来。换句话说，低位排的那几趟不是白排，它们恰恰是用来决定「高位相等时谁在前」的。

每一位上的取值只有 0—9 十种可能，所以一趟可以用计数排序在 O(n) 时间完成。总代价是 O(d(n + b))，d 是位数、b 是每一位的基数。当键的长度是常数（比如 32 位整数按每 8 位拆成 4 趟），它就退化成线性时间。这并不违反 O(n log n) 下界，因为那条下界只对比较排序成立，基数排序压根没比较过。

对比项	计数排序	基数排序	桶排序
排序依据	键本身就是小整数	按位拆开，逐位稳定排序	按键值区间分到若干桶
时间复杂度	O(n + k)，k 是取值范围	O(d(n + b))	平均 O(n)，最坏 O(n²)
额外空间	O(k)	O(n + b)	O(n + k)

对比项	计数排序	基数排序	桶排序
成立前提	k 不能远大于 n	键必须能拆成定长位	数据大致均匀分布
典型场景	分数统计、年龄统计	定长整数、定长字符串、日期	浮点数、均匀分布的大数据

什么时候不能用？键必须是定长、并且能按位拆开的东西：整数、定长字符串、日期都合适。浮点数需要先转换成「可比较的位模式」，负数要单独抽出符号位。这两件事都容易写错，也是基数排序在实践中用得少的原因。另外它是原地性很差的排序，需要 O(n + b) 的辅助空间，数据规模小的时候反而不如直接用快速排序。

代码 4-14　基数排序（LSD）：逐位计数排序

```
def radix_sort(a):
    if not a:
        return []
    out = list(a)
    exp = 1                                  # 当前处理的位：1, 10, 100 ...
    while max(out) // exp > 0:
        cnt = [0] * 10
        for v in out:                        # ① 数一数这一位各数字几次
            cnt[(v // exp) % 10] += 1
        for i in range(1, 10):               # ② 前缀和：每个桶的结束位置
            cnt[i] += cnt[i - 1]
        tmp = [0] * len(out)
        for v in reversed(out):              # ③ 倒序回填，保证稳定
            d = (v // exp) % 10
            cnt[d] -= 1
            tmp[cnt[d]] = v
        out = tmp
        exp *= 10
    return out

print(radix_sort([170, 45, 75, 90, 802, 24, 2, 66]))
# [2, 24, 45, 66, 75, 90, 170, 802]
```

> **常见错误**　一是回填时正序遍历而不是倒序遍历，稳定性被破坏，结果就错了，这一条是基数排序最容易踩的坑。二是用 max(out) 当循环条件，遇到全 0 或空数组会多跑一趟或直接出错。三是只处理非负整数却忘了负数：负数的最高位是符号位，必须先按符号拆成两组分别排序再拼接。四是每趟都新建一个列表，n 很大时内存和复制开销会盖过算法本身省下的时间。

配套仿真：搜索「基数排序」，可以看到每一趟排序之后数组的状态，以及每个桶里进了哪些数。把趟数、桶内容和结果三样东西并排看，稳定性为什么是必需品就变得很具体了。

4.18　最长递增子序列：一道题看三种复杂度

问题：给一个数组，挑出一个子序列（不要求连续，但顺序不能变），要求后一个数比前一个大，问最长能挑多长。比如在 [3, 1, 4, 1, 5, 9, 2, 6] 里，[1, 4, 5, 9] 和 [1, 2, 6] 都是递增子序列，最长的长度是 4。

这道题值得单独拿出来讲，因为同一个问题有三种复杂度差一个数量级的解法，而它们的差别恰好说明了「状态怎么设计」有多重要。

第一种，暴力枚举所有子集，逐个检查是否递增，代价 $O(2^n)$。看起来是完备的，但 n = 30 就跑不动了，只能用来在小数据上验证别的算法。

第二种，动态规划。定义一个状态：dp[i] 表示「以 a[i] 结尾的最长递增子序列有多长」。转移时往前看所有比 a[i] 小、且位置在它前面的元素，取其中最大的 dp 再加一：dp[i] = 1 + max{dp[j] | j < i 且 a[j] < a[i]}。答案是整张 dp 表里的最大值。代价 $O(n^2)$，空间 O(n)。这就是 4.6 讲的「先定义状态，再写转移，最后定边界」的完整演练。

第三种，贪心加二分。关键观察是：如果能凑出两个长度相同的递增子序列，那么末尾更小的那个一定更有前途，因为以后能接上它后面的数更多。所以我们只要维护一张表 tails[len] = 「长度为 len+1 的递增子序列，末尾能达到的最小值」。处理每个新元素时，在 tails 里二分找到第一个大于等于它的位置并替换掉；如果它比 tails 里所有数都大，就追加到末尾。代价 O(n log n)。

有一件事必须提醒：tails 本身通常不是一条合法的递增子序列，它只是「每种长度的最优末尾」这张记账表。想知道具体该挑哪些数，得在更新时另外存一份「每个位置的前驱」。

方法	状态设计	时间复杂度	空间	能否还原具体序列
暴力枚举	没有状态，直接枚举子集	$O(2^n)$	O(n)	能，但要全部存下来
动态规划	dp[i] = 以 i 结尾的最长长度	$O(n^2)$	O(n)	能，另存前驱
贪心 + 二分	tails[len] = 长度为 len+1 的最小末尾	O(n log n)	O(n)	不能，需另存前驱

代码 4-15　最长递增子序列：$O(n^2)$ 打表与 O(n log n) 二分

```
import bisect

def lis_dp(a):                          # O(n^2)：每个位置往前扫一遍
    n = len(a)
    dp = [1] * n
    for i in range(n):
        for j in range(i):
            if a[j] < a[i]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp) if n else 0
```

```
def lis_fast(a):                        # O(n log n)：tails 始终严格递增
    tails = []
    for v in a:
        i = bisect.bisect_left(tails, v)   # 第一个 >= v 的位置
        if i == len(tails):
            tails.append(v)                # 比所有末尾都大，长度加一
        else:
            tails[i] = v                   # 否则压低这一长度的末尾
    return len(tails)

a = [3, 1, 4, 1, 5, 9, 2, 6]
print(lis_dp(a), lis_fast(a))        # 4 4
```

> **容易混淆的地方**　一是把「子序列」当成「子数组」，要求连续，问题完全不同（连续的那一类要用滑动窗口）。二是二分时用了 bisect_right 而不是 bisect_left：求严格递增时用左边界，求「不下降」（允许相等）时才用右边界，两者差一个字符，结果差很多。三是 dp 的初值忘了写成全 1，导致单元素输入返回 0。四是以为 tails 的最终内容就是答案序列。

配套仿真：搜索「最长递增子序列」，左侧可以看着 dp 表被一格一格填满，右侧同时显示 tails 数组在每次替换前后的样子。把同一个数组的两种跑法并排看完，就能明白 O(n log n) 省下来的到底是什么。

4.19　双指针与滑动窗口：把两重循环变成一重

很多看起来要两重循环的题目，实际上只需要一重。关键在于：当答案满足单调性时，右边界只会向前走，不会回头。

打个比方：你在排队买咖啡，想找出「连续多少个人的总价不超过 50 元」的最长一段。你不会每换一个起点就从头重新数，而是右手一直往后伸，超了 50 元就把左手往前移。两只手都只向前走，总共走 2n 步。

代码 4-16　滑动窗口：无重复字符的最长子串

```
def longest_unique(s):
    last = {}          # 字符 -> 最后一次出现的下标
    left = 0
    best = 0
    for right, ch in enumerate(s):
        if ch in last and last[ch] >= left:
            left = last[ch] + 1     # 左手跳到重复字符后面
        last[ch] = right
        best = max(best, right - left + 1)
    return best
print(longest_unique("abcabcbb"))   # 3（abc）
print(longest_unique("pwwkew"))     # 3（wke）
```

判断信号　能不能用滑动窗口，看一件事：能不能说出「窗口变大一定更容易满足条件」（单调性）。能说出，就能用。

4.20　前缀和与差分：两个一行代码的技巧

前缀和解决「反复求区间和」：预先算好 S[i] = a[0] + … + a[i-1]，于是区间 [l, r] 的和就是 S[r+1] − S[l]。一行代码，把每次求和从 O(n) 变成 O(1)。

差分是前缀和的逆运算，解决「反复把一段区间整体加一个数」：设 D[i] = a[i] − a[i-1]，那么给 [l, r] 全加 v，只需 D[l] += v、D[r+1] −= v。最后求一遍前缀和就还原了。

代码 4-17　前缀和与差分

```
# 前缀和：O(1) 求任意区间和
S = [0] * (n + 1)
for i in range(n):
    S[i + 1] = S[i] + a[i]
def range_sum(l, r):        # 闭区间 [l, r]
    return S[r + 1] - S[l]
# 差分：O(1) 做区间加，最后统一还原
D = [0] * (n + 2)
def range_add(l, r, v):
    D[l] += v
    D[r + 1] -= v
def rebuild():
    cur = 0
    for i in range(n):
        cur += D[i]
        a[i] = cur
```

组合用法　二维版本同样成立：二维前缀和能 O(1) 求矩阵子矩阵和，二维差分能 O(1) 给一个矩形区域整体加一个数。图像处理里的积分图就是这个东西。

4.21　二分答案：把「求最大值」换成「判断行不行」

4.3 讲了二分查找：在有序数组里找一个数。但二分的用法远不止于此。很多题目问「最大的最小值是多少」「最小的最大值是多少」，而且这个答案具有单调性：小到某个值之前都不行，之后都行。这时就能直接对答案二分。

打个比方：你要从一堆货物里找出能装进背包的最大重量。一个办法是试所有组合（爆炸）；另一个办法是：先试背 50 斤行不行，行就试 75，不行就试 37……把「能不能装下」当成一个能快速回答的问题。只要这个判断函数是 O(n) 的，整体就是 O(n log C)。

关键要点有三个：① 确定答案的范围 [lo, hi]；② 写出 check(x) 表示「x 能不能办到」；③ 正确处理边界（二分最容易出错的就是这里）。

代码 4-18　二分答案的模板：求最小的可行值

```
def solve(lo, hi, check):
```

```
    # check(x) 为 True 表示 x 可行；要求看上去对 x 单调
    while lo < hi:
        mid = (lo + hi) // 2
        if check(mid):
            hi = mid          # mid 可行，答案可能更小
        else:
            lo = mid + 1      # mid 不行，答案一定在右边
    return lo
# 例子：将一组任务分给 k 台机器，求最小的最大负载
def min_max_load(jobs, k):
    def ok(limit):
        used, cur = 1, 0
        for j in jobs:
            if j > limit:
                return False
            if cur + j > limit:
                used += 1
                cur = 0
            cur += j
        return used <= k
    return solve(max(jobs), sum(jobs), ok)
```

> **边界**　二分写错很少是思路错，都是边界错：该不该 +1、循环是 lo < hi 还是 lo <= hi。一个稳妥的习惯是：用「求最小可行值」和「求最大可行值」两个固定模板，以后只填 check，不重写循环。

4.22　回溯与剪枝：穷举所有可能，但不傻穷举

回溯（Backtracking）的本质是「带撤销的深度优先遍历」：走一步，记下现状；不行就退回来，把现状恢复。打个比方：走迷宫。你在每个岔路口选一条路，用粉笔在地上画个记号；走不通就回到上一个岔路口，画掉记号换下一条。

单纯回溯会指数爆炸（每层分支都乘起来），所以剪枝（Pruning）才是重点：提前判断「这条路已经不可能有答案」，直接返回。常见的剪枝有四类：可行性剪枝、最优性剪枝（当前已经比最优解差）、排序后提前撞到好解、去重。

代码 4-19　回溯模板：全排列 + 剪枝

```
def permute(nums):
    nums.sort()
    res, path = [], []
    used = [False] * len(nums)
    def backtrack():
        if len(path) == len(nums):
            res.append(path[:])       # 记下一个解
            return
        for i, x in enumerate(nums):
            if used[i]:
                continue
```

```
            if i > 0 and nums[i] == nums[i - 1] and not used[i - 1]:
                continue              # 剪枝：相同值只让最靠前的那个先用
            used[i] = True
            path.append(x)
            backtrack()
            path.pop()                # 撤销：恢复现场
            used[i] = False
    backtrack()
    return res
```

> **方法论**　回溯的代码永远是同一个骨架：做选择 → 递归 → 撤销选择。学会了骨架，剩下的工作就只剩「找出选择列表」和「想清楚什么时候剪枝」。

4.23　分治：把大问题切成小问题

快速排序和归并排序都是分治的产物：把问题分成几个规模更小的子问题，分别解决，再把结果合并。三个词：分（Divide）、治（Conquer）、合（Combine）。

分治的价值在于子问题互不相关，可以并行。这一点在多核时代非常重要，后面的并行排序、分布式计算、MapReduce，思路都与此相同。

表 4-6　四个分治算法的归纳式与复杂度

算法	归纳式	解	关键点
归并排序	T(n) = 2T(n/2) + O(n)	O(n log n)	合并两个有序数组
快速排序	T(n) = 2T(n/2) + O(n)（平均）	O(n log n)	分区平衡时最快
二分查找	T(n) = T(n/2) + O(1)	O(log n)	每次丢掉一半
大整数乘法（Karatsuba）	T(n) = 3T(n/2) + O(n)	O(n^1.585)	用 3 次子乘法换 4 次

> **主定理**　上表的对数都能用主定理（Master Theorem）一步算出来：比较 f(n) 和 n^(log_b a) 的大小。不记得也没关系，记住上表这几个常见结果就够用了。

4.24　位运算与状态压缩：把集合当数用

当一个集合的元素个数不超过 20 左右时，可以把「选了哪些元素」直接用一个整数的二进制位表示：第 i 位是 1 就表示选了第 i 个元素。这叫状态压缩（Bitmask DP）。

它让一些看起来必须搜索的问题变成了动态规划。最典型的是旅行商问题（TSP）：从一个城市出发，每个城市只去一次，最后回到起点，求最短路。状态是「已经去过哪些城市 + 现在在哪」，合起来就是 dp[mask][i]。

表 4-7　状态压缩常用位运算写法

目的	写法	读法
查第 i 位是不是 1	mask >> i & 1	mask 右移 i 位后看末位
把第 i 位置 1	mask | (1 << i)	或上一个只有第 i 位的数
把第 i 位清 0	mask & ~(1 << i)	与上一个除第 i 位外全是 1 的数
取出 mask 里所有 1	while mask: b = mask & -mask; ...; mask ^= b	每次取最低位
枚举所有子集	sub = mask; while sub: ... ; sub = (sub - 1) & mask	子集枚举标准写法

范围　状态压缩的上限大约是 20–22 位（2^22 约 400 万）。再大就爆内存了。看到题目里 $n \le 20$ 这个条件，就应该反射出来。

4.25　数论基础：每个程序员都该会的三件事

第一件是最大公约数（辗转相除法）：两个数不断用余数代替，最后非零的那个就是答案。它是数论里用得最多的一段代码，也是分数运算、模运算逆元的前置。

第二件是素数筛法：要找出 1 到 n 的所有素数，不要对每个数都试除到根号。埃拉托斯特尼筛法（筛法）的思路是：找到一个素数，就把它的所有倍数划掉。复杂度是 O(n log log n)，几乎是线性的。

第三件是快速幂：求 a^b mod m。不要循环 b 次，而是把 b 看成二进制，每一位把底数平方一次。正因为有它，才有 RSA 加密和后面的同态加密。它是线性级别里最「四两拨千斤」的技巧。

代码 4-20　三件必备的数论小工具

```
def gcd(a, b):                    # 辗转相除，O(log min(a,b))
    while b:
        a, b = b, a % b
    return a
def sieve(n):                     # 埃氏筛，返回 <= n 的素数个数
    is_p = bytearray([1]) * (n + 1)
    is_p[0:2] = b'\x00\x00'
    for i in range(2, int(n ** 0.5) + 1):
        if is_p[i]:
            is_p[i * i::i] = bytearray(len(is_p[i * i::i]))
    return sum(is_p)
def pow_mod(a, b, m):             # 快速幂，O(log b)
    r = 1
    a %= m
    while b:
        if b & 1:
            r = r * a % m
```

```
        a = a * a % m
        b >>= 1
    return r
```

> **常识**　RSA 加密里的公钥就是一对数（e, n），加密就是 c = m^e mod n。所以「为什么加密很快」的答案，就在上面那十几行里。

4.26　背包问题六讲：从暴力搜索到一维数组

背包问题是动态规划最好的入门题。它足够小，小到能手工推演；又足够深，深到能牵出 DP 的全套思路。这一节把它拆成六个版本，每一个版本都比前一个快一个量级，你只要跟着走完，DP 就入门了。

题目：有 n 件物品，第 i 件重 w[i]、值 v[i]。背包最多装 W 重。每件物品要么拿、要么不拿，求能装的最大总价值。

4.26.1　第一讲：暴力枚举，看清所有可能

每件物品只有两种命运，n 件物品就有 2^n 种组合。枚举所有组合，挑不超重里最值钱的。这是最直接的想法，也是后面所有优化要对照的基准。

> **它在做什么**　用最笨也最稳的办法枚举背包的每一种取舍，先把「正确答案长什么样」确定下来。

决策只有两种：第 i 件要么拿、要么不拿。于是可以写一个递归函数 best(i, 剩余容量)，返回「从第 i 件往后能取得的最大价值」。拿了就往下一层减容量加价值，不拿就往下一层原样传下去，最后取两者较大的那个。

这版慢得离谱，但它非常重要——它是后面所有优化的参照物。先写出这个版本，你才能确认自己真的想清楚了状态是什么、选择是什么。

- 第一步：定状态，best(i, rest) 表示从第 i 件到最后、容量还剩 rest 时的最大价值。
- 第二步：定转移，答案 = max(不拿第 i 件, 拿第 i 件)。
- 第三步：定边界，i 走到末尾就返回 0；容量不够时只能选择不拿。

> **为什么它会慢**　同一个子问题会被反复计算。best(5, 10) 可能从十几条不同路径走到，每次都从头再算一遍，代价是指数级重复。

```
def best(i, rest):
    if i == n: return 0
    skip = best(i + 1, rest)                # 不拿第 i 件
    take = 0
    if w[i] <= rest:
        take = v[i] + best(i + 1, rest - w[i])   # 拿
    return max(skip, take)
```

代码 4-50　暴力搜索：2^n 种可能，n = 30 就跑不动了

> **为什么会慢**　同一个子问题被反复计算。best(5, 10) 可能从十几条不同的路径到达，每次都从头算一遍，代价是呈指数级重复。

4.26.2　第二讲：加个备忘，把重复计算砍掉

把所有算过的 (i, rest) 结果记在一个表里。下次再遇到同样的参数，直接查表。这一招叫记忆化搜索，是把指数级变成多项式级的关键一步。

```
from functools import lru_cache
```

> **修的是什么**　把已经算过的子问题答案记下来，第二次遇到直接取。

上一版慢的根源不是「算得不对」，而是「同一件事被算了无数遍」。解决办法朴素到不行：算出一个子问题的答案就写进一张小本子（表），下次需要时先翻本子，有就直接用。这叫记忆化搜索。

- 第一步：准备一张表，下标是状态（i 和 rest）。
- 第二步：函数一进来先查表，有值就直接返回。
- 第三步：算出答案后先写回表里，再返回。

状态数变少了多少？原来是 2^n 条路径，现在是「件数 × 容量上限」这么多格子。n 取 30、容量取一万时，从十亿级别降到三十万级别，这就是动态规划能把指数问题变成多项式问题的全部秘密。

> **关键区分**　记忆化搜索是「自顶向下 + 备忘」，动态规划是「自底向上 + 打表」。两者算的是同一张表，只是填表的顺序相反。

```
@lru_cache(maxsize=None)
def best(i, rest):
    if i == n: return 0
    ans = best(i + 1, rest)
    if w[i] <= rest:
        ans = max(ans, v[i] + best(i + 1, rest - w[i]))
    return ans
```

代码 4-51　记忆化搜索：状态数从 2^n 降到 n×W

状态一共有 n×(W+1) 个（第几件、还剩多少容量），每个状态只算一次，所以复杂度是 O(nW)。这就是 DP 的第一个视角：找状态、去重。

4.26.3　第三讲：翻译成递推的表格

记忆化是「从大问题往下问」，递推是「从小问题往上填」。定义 dp[i][c] = 只考虑前 i 件物品、容量为 c 时的最大价值，那么每一格的值来自两个已知格子。

dp[i][c] = max(dp[i-1][c] , dp[i-1][c-w[i]] + v[i])。前半句是「不拿第 i 件」，用上面一格的答案；后半句是「拿第 i 件」，用的是上一行左边那一格加上本件的价值。

下面用一个具体例子把这张表填出来。容量 W = 5，三件物品：(重 2, 值 3)、(重 3, 值 4)、(重 4, 值 5)。

表 4-17　背包 DP 表逐步填满的过程（行=考虑前 i 件，列=容量 0..5）

i \ c	0	1	2	3	4	5	这一行在说什么
0 件	0	0	0	0	0	0	什么都不拿，全是 0
1 件（2,3）	0	0	3	3	3	3	容量够 2 就能拿它，值 3
2 件（3,4）	0	0	3	4	4	7	容量 5 时两件都拿：3+4=7
3 件（4,5）	0	0	3	4	5	7	容量 4 时第 3 件更值 5；容量 5 时仍是 7

注意第 3 行、容量 4 那一格：拿第 3 件得 5，不拿得 4，所以取 5。而容量 5 那一格，拿第 3 件要占掉 4 的容量，剩下的 1 装不下任何东西，总价值 5，还不如不拿（7），所以保持 7。整个 DP 的推理，就是在这两个数之间反复比较。

4.26.4　第四讲：把它压成一维数组

观察上面那张表，第 i 行的值只用到第 i-1 行。既然只用上一行，就可以只留一行，边填边覆盖。这就是滚动数组，也是背包最经典的写法。

> **这一版省什么**　把二维的表压成一维，省掉整整一行内存。

二维写法里，算 dp[i][j] 只用得到上一行 dp[i-1][...] 的值。既然旧行用完就废，那就可以只留一行，边算边覆盖。

- 第一步：把 dp 压成一个长度为「容量上限 + 1」的数组。
- 第二步：外层照旧遍历每件物品。
- 第三步：内层容量必须从大到小遍历，这样 dp[j - w] 取到的还是上一轮的值，不会用到本轮刚写过的。

五行代码就把空间从 O(nW) 压到 O(W)，这是背包的标准写法，务必背下来。

> **最容易错的一处**　内层循环改成从小到大，就变成了完全背包（每件可以拿无限次）。一维背包的倒序不是习惯问题，是正确性问题。

```
dp = [0] * (W + 1)
for i in range(n):
```

```
    for c in range(W, w[i] - 1, -1):        # 注意：从大到小！
        dp[c] = max(dp[c], dp[c - w[i]] + v[i])
print(dp[W])
```

代码 4-52　0-1 背包的一维写法（只有五行）

> **为什么必须倒着遍历**　因为 dp[c] 依赖 dp[c-w[i]]，而那个位置必须还是「上一轮（i-1）」的值。正着遍历时 dp[c-w[i]] 已经被本轮更新过，相当于同一件物品拿了两次——那就变成了完全背包。这一个符号的差别，是背包问题里最著名的坑。

4.26.5　第五讲：完全背包与多重背包

把倒序改成正序，得到的恰好是「每件物品可以拿无限次」的完全背包。这个巧合不是运气，而是「允许重复使用当前物品」正好对应「用本轮更新过的值」。

表 4-18　三种背包的差别只在一行

类型	物品可用次数	内层循环方向	一句话理解
0-1 背包	每件至多 1 次	从 W 递减到 w[i]	倒序保证不重复拿
完全背包	每件无限次	从 w[i] 递增到 W	正序允许重复拿
多重背包	每件有数量上限 k	按二进制拆成 1,2,4,… 件再套 0-1	把 k 件拆成若干「组合件」

多重背包的二进制拆分很有巧思：如果某件物品有 13 个，不需要拆成 13 个独立的 0-1 物品，而是拆成 1、2、4、6 四个「打包件」——它们的组合能表示 0 到 13 之间的任何一个数量。13 个变 4 个，复杂度从 O(nWk) 降到 O(nW log k)。

4.26.6　第六讲：背包还能解决哪些看起来不像背包的问题

背包的价值在于它是一类问题的模板。很多题目换了个说法，本质上还是「选或不选，容量有限」。

表 4-19　五道伪装成别的题目的背包问题

题面说的话	物品是什么	容量是什么	价值是什么
从一堆数里挑出若干个，使和恰好等于 S	每个数	目标和 S	1（能不能凑出）
把数组分成两半，使两半和尽量接近	每个数	总和的一半	数的值本身
用最少的硬币凑出金额	每种硬币	金额	1（硬币个数取负）
选课，学分上限 20，求最大收获	每门课	学分	课程收益

题面说的话	物品是什么	容量是什么	价值是什么
服务器选型，预算固定，求最大算力	每种机型	预算	算力

所以看到「给定一组东西，每种有代价和收益，总量有上限，求收益最大」这样的题面，第一反应就该是背包。这也是本书第 10 章选型清单里把背包列为一类模板的原因。

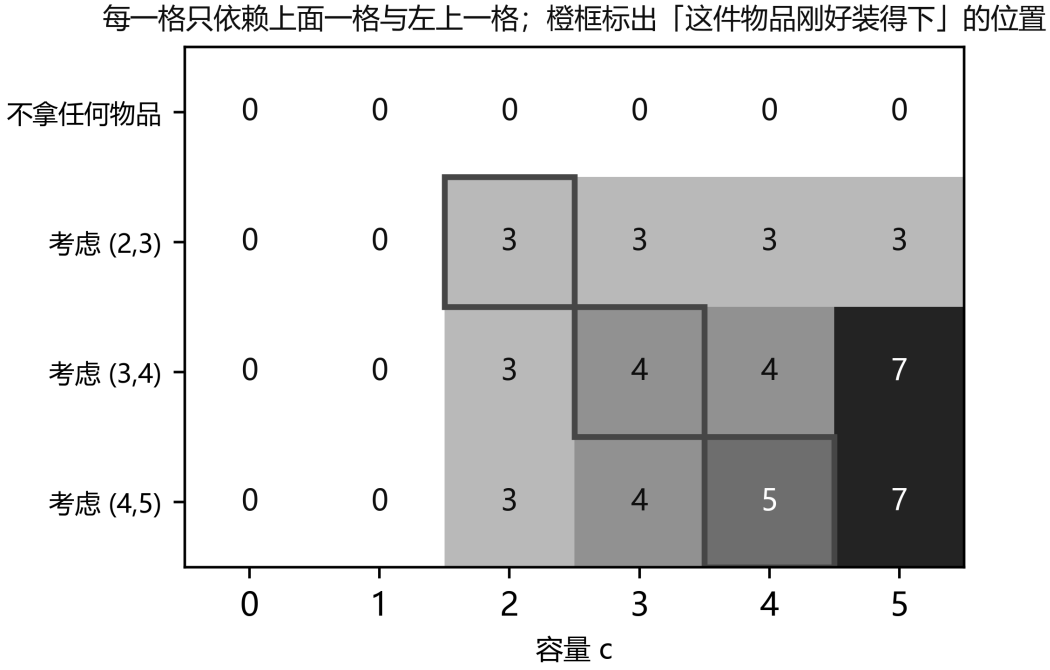


图 4-2　0-1 背包的 DP 表：每一格只从上面一格或左上一格取值

4.27　手推一次快速排序：把分区过程画出来

快速排序的思想一句话能说完：挑一个基准，把比它小的放到左边、比它大的放到右边，然后对两边分别重复。但真正写起来，最容易写错的是分区那几行。这一节把分区过程一步步画出来，让你看清双指针是怎么走的。

打个比方：把一堆书按「比参考书薄还是厚」分成两摞。你左手从左边找一本比参考书厚的，右手从右边找一本比参考书薄的，两本一换，继续往中间收拢，直到两只手相遇。

4.27.1　一次完整的分区推演

数组 [5, 3, 8, 4, 2, 7, 1, 6]，取最后一个元素 6 作为基准。

表 4-20　Lomuto 分区法的每一步（基准 = 6，指针 i 指向「小值区的末尾」）

步骤	j 指向	a[j] 与基准比	i 的变化	数组状态
开始	0	—	i = -1	[5,3,8,4,2,7,1,6]

步骤	j 指向	a[j] 与基准比	i 的变化	数组状态
1	0（值 5）	5 < 6	i 增到 0，与自身交换	[5,3,8,4,2,7,1,6]
2	1（值 3）	3 < 6	i 增到 1，与自身交换	[5,3,8,4,2,7,1,6]
3	2（值 8）	8 > 6	i 不动	[5,3,8,4,2,7,1,6]
4	3（值 4）	4 < 6	i 增到 2，与 8 交换	[5,3,4,8,2,7,1,6]
5	4（值 2）	2 < 6	i 增到 3，与 8 交换	[5,3,4,2,8,7,1,6]
6	5（值 7）	7 > 6	i 不动	[5,3,4,2,8,7,1,6]
7	6（值 1）	1 < 6	i 增到 4，与 8 交换	[5,3,4,2,1,7,8,6]
收尾	—	把基准换到 i+1	i+1 = 5	[5,3,4,2,1,6,8,7]

结果里，基准 6 稳稳地落在下标 5：左边五个数都比它小，右边两个都比它大。这就是分区的全部目的——它不保证左边有序，只保证「左边全小、右边全大」。有序是靠递归在两边继续分区完成的。

4.27.2 为什么最坏是 $O(n^2)$，怎么避免

如果每次选的基准恰好是最小值或最大值，分区会变成「一边 0 个、一边 n-1 个」，递归深度变成 n，总代价 $O(n^2)$。对已经排好序的数组取最后一个元素作基准，正好会掉进这个陷阱。

三种常见对策：随机选基准（最常用，实现只要一行）、三数取中（取首、中、尾的中位数）、三路划分（把等于基准的元素单独归一堆，适合大量重复元素的场景）。

```
import random
```

> **它在修什么** 把「每次都拿最后一个元素当基准」换成「随机挑一个」，让最坏情况几乎不可能出现。

快排的平均复杂度是 O(n log n)，但有个致命弱点：如果基准每次都取到最小或最大的元素，划分会极度不平衡，退化成 $O(n^2)$。而「已经排好序的数组」恰好就能触发这种退化——偏偏真实数据里排好序的情况非常常见，所以朴素快排很容易踩坑。

- 第一步：在待排区间里随机挑一个下标。
- 第二步：把它和区间最后一个元素交换，剩下的逻辑完全不用改。
- 第三步：于是「最坏输入」不再由数据决定，攻击者也没法构造出让程序变慢的输入。

这就是随机化的价值：不改变平均性能，却把「最坏情况」从必然变成几乎不可能。对于任何人可能控制输入的场景，这个改动都是必须的。

> **读代码看这里**　改动只有一行随机交换。这就是为什么说随机快排是「一行改动换来安全」的典型例子。

```
def quicksort(a, lo, hi):
    if lo >= hi: return
    p = random.randint(lo, hi)
    a[p], a[hi] = a[hi], a[p]          # 随机基准换到末尾，复用同一套分区
    i = lo - 1
    for j in range(lo, hi):
        if a[j] < a[hi]:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[i+1], a[hi] = a[hi], a[i+1]
    quicksort(a, lo, i)
    quicksort(a, i+2, hi)
```

代码 4-53　随机基准的快排：避免最坏情况的一行改动

> **稳定吗**　不稳定。相等元素的相对顺序会被交换打乱。需要稳定排序时用归并排序，或者给每个元素带上原始下标参与比较。

4.28　图论算法选型表：什么时候用哪个

图论算法很多，名字又像。这一节按「你要解决什么问题」来分类，给出直接可用的选择。表格按问题的类型组织：最短路、连通性、遍历、拓扑、最小生成树。

4.28.1　最短路：按图的特征选

表 4-21　最短路算法的选择

图的特征	该用谁	复杂度	为什么
边权全为正，单源	Dijkstra（堆优化）	O((V+E) log V)	贪心扩张，堆替代线性找最小
边权全是 1（或同权）	BFS	O(V+E)	不需要堆，普通队列就够
有负权边，单源	Bellman-Ford	O(VE)	能处理负权，还能检测负环
有负权边，队列优化版	SPFA	平均快、最坏 O(VE)	用队列只松弛变化过的点
多源，任意两点	Floyd-Warshall	$O(V^3)$	三重循环，代码极短
只求两点且图很大	双向 BFS / A*	视启发函数	从两端同时搜，减少展开量

选择顺序很清晰：先看边权是不是负数，再看是单源还是多源，最后看规模。90% 的工程场景是「正权、单源、稀疏图」，答案就是堆优化的 Dijkstra。

4.28.2 连通性、遍历与拓扑

表 4-22 连通性类算法的选择

问题	该用谁	复杂度	关键点
判断两个点是否连通（动态加边）	并查集	接近 O(1)	只加边不删边才适用
求所有连通分量	DFS / BFS 或并查集	O(V+E)	两种都行，看是否需要顺带遍历
在有向图里找环	拓扑排序	O(V+E)	排不出全部点就说明有环
求强连通分量	Tarjan / Kosaraju	O(V+E)	一次 DFS 加一个栈
求割点与桥	Tarjan	O(V+E)	dfn 与 low 两个数组
二分图判定	BFS 染色	O(V+E)	相邻点必须异色

> **并查集的边界** 并查集只支持「加边」，不支持「删边」。遇到需要动态删边的连通性问题，要么离线倒着处理（把删边当成加边反着做），要么换用 Link-Cut Tree 这类高级结构。

4.28.3 最小生成树与稀疏化

表 4-23 最小生成树的选择

图的稠密程度	该用谁	复杂度	原因
稀疏图（E 接近 V）	Kruskal	O(E log E)	按边排序加并查集，实现简单
稠密图（E 接近 V^2）	Prim（朴素版）	$O(V^2)$	不用排序所有边
稠密图 + 要更快	Prim（堆优化）	O(E log V)	但常数比朴素版大

记住一句判断：边少用 Kruskal，点少边多用 Prim。此外 Kruskal 的副产品很实用——如果把 MST 里最重的那条边去掉，剩下的两棵树正好是最小化「最大边」意义下的最优二分。

4.28.4　一张总览图

表 4-24　看图论题目的五步选择流程

步骤	问什么	如果答案是「是」	如果答案是「否」
1	涉及边权吗？	进入第 2 步	用 BFS/DFS/并查集
2	边权有负数吗？	Bellman-Ford / SPFA / Floyd	进入第 3 步
3	是单源还是多源？	单源：Dijkstra	多源：Floyd
4	要求路径还是要求连通？	路径：最短路	连通：MST（Kruskal/Prim）
5	图会动态变化吗？	只用加边：并查集	有删边：离线倒做或 LCT

第 5 章　机器学习算法：从数据中学规律

> **读前提示**　需要第 0 章 0.3 的「向量、导数、概率」三个概念，以及 5.0 节会先讲的四个公共概念（偏差(模型假设太简单而带来的固有误差)、方差(数据的波动程度；用在模型上指对数据扰动过度敏感带来的误差)、正则化(给模型加约束，防止它学得太复杂)、权衡）。本章术语较多，但每个术语首次出现时都带括号解释。

与第 4 章不同，这一章的算法不写死规则，而是从样本中学出一个函数 f(x) → y。它们的共同框架是：定义假设空间 → 定义损失函数(衡量预测错了多少的公式) → 用优化算法最小化损失 → 在未见数据上验证泛化。

5.0　先建立四个公共概念

概念	含义	工程上的表现
偏差 Bias	模型假设太简单，学不到真实规律	训练集(用来让模型学习的那部分数据)和测试集(只在最后打分用、训练过程中绝不碰的数据)都差 → 欠拟合(学得太粗糙，连训练集都没学好)，要加特征/加复杂度
方差 Variance	模型太敏感，把噪声当规律	训练集好、测试集差 → 过拟合(把训练集死记硬背下来，遇到新数据就崩)，要加数据/正则/降复杂度
正则化	对模型复杂度施加惩罚，抑制过拟合	L1（稀疏）、L2（平滑）、早停、Dropout、数据增强(对原始数据做保持标签(这条数据的正确答案)的变换，凭空变出更多数据)
偏差—方差权衡	集成学习的出发点	Bagging 主要降方差（随机森林）；Boosting 主要降偏差（GBDT）

5.1　线性回归 Linear Regression

定位：最可解释的预测模型，也是所有广义线性模型与神经网络最后一层的原型。

> **它在做什么**　用一条直线去拟合一堆点，并且给出两种求这条线的方法。

线性回归要回答的问题是：房价和面积之间大概是什么关系？假设这个关系是「$y \approx w \cdot x + b$」，任务就变成找一组 w 和 b，让所有点离这条线的总体偏差最小。这里的偏差用平方和来量，所以叫最小二乘。

为什么用平方而不是绝对值？因为平方处处可导，可以算梯度、一步推出闭式解；绝对值在零点不可导，处理起来麻烦。这是「选择损失函数」时的常见取舍：不光看合不合理，还要看好不好优化。

- 闭式解：对损失求导令其为零，可以直接解出 w 和 b。它一步到位、结果精确，但要算矩阵逆，特征多时开销很大。
- 梯度下降：先随便猜一组参数，算一下梯度（哪个方向下降最快），朝反方向迈一小步，反复迭代。它慢，但能扩展到任何复杂的模型。

两者不是竞争关系：样本少、特征少时用闭式解；数据大到算不动矩阵逆时用梯度下降。神经网络能训练，靠的就是后者。

> **读代码看这里**　两种解法的结果应该几乎一样——如果不一样，通常是学习率太大或者迭代次数不够。

代码 5-1　线性回归：闭式解与梯度下降

```
import numpy as np

# 1) 闭式解（正规方程）：n 小时直接用
def fit_closed_form(X, y):
    Xb = np.c_[np.ones(len(X)), X]           # 加偏置列
    return np.linalg.pinv(Xb.T @ Xb) @ Xb.T @ y   # θ = (XᵀX)⁻¹Xᵀy

# 2) 梯度下降：n 或特征数很大时用
def fit_gd(X, y, lr=0.01, epochs=1000):
    Xb = np.c_[np.ones(len(X)), X]
    w = np.zeros(Xb.shape[1])
    for _ in range(epochs):
        grad = Xb.T @ (Xb @ w - y) / len(y)  # ∇MSE
        w -= lr * grad
    return w
```

参数 / 符号	含义	取值与影响
X / y	特征矩阵 (n×d) 与标签向量 (n,)	X 需先标准化(把不同量纲的特征拉到同一尺度，便于比较)，否则各系数量纲不同、无法比较大小
np.c_[ones, X]	在 X 左侧拼一列 1	把偏置(每个神经元的固有偏移量)项 b 吸收进 θ，使 y=Xθ 形式统一，求导与实现都更简单
pinv	伪逆（Moore-Penrose）	用 pinv 而非 inv：X^TX 奇异（多重共线性）时 inv 会直接报错，pinv 给出最小范数解
lr	学习率(每次沿梯度走多大一步)	太大发散、太慢收敛。可用损失曲线判断：震荡说明太大，下降极慢说明太小
epochs	遍历全量数据的轮数	配合早停使用：验证集(用来调旋钮、选模型的那部分数据)损失连续若干轮不降就停

参数 / 符号	含义	取值与影响
grad	MSE 对 θ 的梯度 $X^T(X\theta-y)/n$	除以 n 使梯度与批量大小无关，换 batch 时学习率不必重调

代码逐段解读：

- 闭式解：$\theta=(X^TX)^{-1}X^Ty$ 一步到位，但要求 X^TX 可逆、且复杂度 $O(d^3)$，特征上万维时不可行。
- 梯度下降：每轮算一次全量梯度再统一更新，是批量梯度下降（BGD）。样本量大时改用小批量（Mini-Batch）。
- 为什么把偏置拼进 X：这样 θ 的第一个分量就是截距，损失对 θ 的梯度形式统一，代码只需维护一个向量。
- 数值稳定性提醒：闭式解在高维共线时数值极不稳定，工程上要么加岭回归的 λI，要么直接走梯度下降。

实现原理

假设 $y = X\theta + \varepsilon$，用最小二乘最小化残差平方和 $MSE = 1/n\cdot\Sigma(y_i - x_i^T\theta)^2$。闭式解需要求 (X^TX) 的逆，复杂度 $O(d^3)$ 且数值不稳定（多重共线性时接近奇异），因此工程上多用 pinv（伪逆）或改走梯度下降 / 岭回归。

维度	说明
解决的问题	用若干特征的线性组合预测连续值（房价、销量、时延）
核心原理	最小化均方误差，等价于在高斯噪声假设下的极大似然估计
时间复杂度	闭式解 $O(d^3 + nd^2)$；梯度下降 $O(epoch\cdot n\cdot d)$
空间复杂度	$O(d)$ 参数 + $O(nd)$ 数据
优点	完全可解释（系数即影响方向与大小）、训练极快、可作为强基线
缺点	只能表达线性关系；对异常值与多重共线性敏感
典型应用	业务指标归因、A/B 实验校正、金融风控打分卡、作为复杂模型的 baseline
关键参数 / 调参要点	是否标准化（必须）、正则强度（岭 α / Lasso α）、是否加交互项与多项式特征

5.2 逻辑回归 Logistic Regression

定位：工业界最常用的二分类模型。名字带「回归」但做的是分类，输出可直接当概率用，因此被大量用于需要「可解释 + 可校准概率」的场景（风控、CTR 预估）。

代码 5-2　逻辑回归（交叉熵损失 + L2 正则）

```
import numpy as np
```

```
def sigmoid(z):
    return 1 / (1 + np.exp(-np.clip(z, -30, 30)))   # clip 防溢出

def fit_logreg(X, y, lr=0.1, epochs=2000, l2=1e-3):
    Xb = np.c_[np.ones(len(X)), X]
    w = np.zeros(Xb.shape[1])
    for _ in range(epochs):
        p = sigmoid(Xb @ w)
        grad = Xb.T @ (p - y) / len(y) + l2 * w     # 交叉熵梯度 + L2
        w -= lr * grad
    return w
# 损失：-1/n·Σ[ y·log(p) + (1-y)·log(1-p) ]，与线性模型形式相同但链接函数不同
```

参数 / 符号	含义	取值与影响
sigmoid(z)	把线性打分压到 (0,1)	输出可直接当概率用，这是逻辑回归在风控/CTR 场景不可替代的原因
np.clip(z,-30,30)	截断指数输入范围	防止 exp 溢出为 inf 导致 NaN，是数值稳定的必要保护
p = sigmoid(Xb @ w)	预测为正类的概率	阈值默认 0.5，但不均衡场景需按业务目标移动阈值
grad = Xb.T@(p-y)/n	交叉熵(衡量概率预测与真实答案差距的一种损失)对 w 的梯度	形式与线性回归完全一致，区别只在链接函数——这是广义线性模型的统一性
l2 / 1e-3	L2 正则强度	越大系数越趋近 0、模型越平滑；默认 1e-3—1e-2，需按特征量级调
epochs=2000	训练轮数	逻辑回归通常几十到几百轮即收敛，配合验证集早停

代码逐段解读：

- 损失来源：交叉熵 $-1/n\cdot\Sigma[y\cdot\log p+(1-y)\cdot\log(1-p)]$，等价于伯努利分布的极大似然估计。
- 梯度为何这么简洁：sigmoid 的导数 $\sigma'=\sigma(1-\sigma)$ 与交叉熵的导数恰好约掉，只剩 $(p-y)\cdot x$，这是特意选择该损失与链接函数配对的结果。
- L2 项只加在梯度里而不进损失：注意正则通常不对偏置项施加，否则会强行把截距拉向 0。
- 多分类扩展：把 sigmoid 换成 softmax、损失换成多类交叉熵，就是 Softmax(把一组分数压成加起来等于 1 的概率) 回归（也叫多类逻辑回归）。

维度	说明
解决的问题	二分类 / 多分类（Softmax 回归）并给出校准后的概率
核心原理	线性打分 $z=w^Tx$，经 sigmoid 压到 (0,1)；用极大似然（交叉熵）求解
时间复杂度	O(epoch·n·d)，通常几十轮即收敛
空间复杂度	O(d)

维度	说明
优点	输出是概率、可解释、训练快、支持在线学习、易加正则
缺点	线性决策边界(模型划分不同类别的那条分界线)，无法自动捕捉特征交互（需人工构造交叉特征）
典型应用	信贷风控评分卡、广告 CTR 预估、医学风险预测、LLM 中作为 reward model 的打分头
关键参数 / 调参要点	学习率、L2 强度、类别权重/采样（不均衡时）、特征分箱与 WOE 编码、阈值选择（影响召回/精确率(报出来的结果里，真正正确的比例)）

> **为什么现在还在用** 在很多强监管行业（金融、医疗），监管要求「每一分怎么来的必须能解释」，逻辑回归 + 分箱 WOE 依然是首选；深度模型往往只作为对照或用于非决策环节。

5.3 决策树 Decision Tree（ID3 / C4.5 / CART）

定位：可解释的非线性模型，也是随机森林与 GBDT 的基学习器。理解树，才能理解集成。

代码 5-3 CART 分类树：基尼不纯度最小化 + 递归二分

```
import numpy as np

def gini(y):
    p = np.bincount(y) / len(y)
    return 1 - (p ** 2).sum()            # 基尼不纯度

def best_split(X, y):
    best = (None, None, float("inf"))
    for j in range(X.shape[1]):
        for t in np.unique(X[:, j]):     # 候选切分阈值
            m = X[:, j] <= t
            if m.all() or not m.any():
                continue
            g = (m.sum() * gini(y[m]) + (~m).sum() * gini(y[~m])) / len(y)
            if g < best[2]:
                best = (j, t, g)         # 选加权基尼最小的切分
    return best

def build(X, y, depth=0, max_depth=5):
    if depth >= max_depth or len(set(y)) == 1:
        return np.bincount(y).argmax()   # 叶节点：多数类
    j, t, _ = best_split(X, y)
    if j is None:
        return np.bincount(y).argmax()
    m = X[:, j] <= t
    return {"feat": j, "thr": t,
```

```
            "left": build(X[m], y[m], depth + 1, max_depth),
            "right": build(X[~m], y[~m], depth + 1, max_depth)}
```

参数 / 符号	含义	取值与影响
gini(y)	基尼不纯度(衡量一组样本有多混杂，越小越纯) $1-\Sigma p^2$	值越小越纯；二分类时 0 表示全同类，0.5 表示完全混合
j / t	切分特征下标与阈值	一次分裂只用一个特征的一个阈值，形成轴平行的决策边界
m = X[:, j] <= t	按阈值划分左右子集的布尔掩码	布尔索引是 numpy 里最快的子集划分方式
加权基尼 g	按样本数加权的左右不纯度均值	除以 len(y) 做归一化，让不同深度的节点可比
max_depth=5	树的最大深度	控制模型复杂度的最主要旋钮；越深方差越大、越容易过拟合
min_samples_leaf	叶节点最小样本数（可选）	比 max_depth 更灵活地防过拟合，常取 5—50

代码逐段解读：

- best_split 是双重穷举：对每个特征的每个候选阈值都算一遍加权基尼，取最小的——这是训练耗时的主要来源。
- build 的递归出口有两个：深度用尽，或当前节点已经是纯的（len(set(y))==1）。
- 叶节点返回 np.bincount(y).argmax()，即众数类别；回归树则改为返回均值。
- 停止条件决定方差：不设限制时树能长到每个叶节点只有一个样本（训练误差 0），这就是决策树必须剪枝的原因。
- 加速方向：真实库（sklearn/XGBoost）会对特征预排序、做直方图分桶(把连续的值划进若干个区间，用区间代替原值)，把 O(n·d·log n) 再压一个数量级。

实现原理

- 不纯度度量：分类用基尼指数或信息增益（熵），回归用均方误差 MSE 的下降量。
- 切分是「贪心 + 穷举」：对每个特征的每个候选阈值算一遍加权不纯度，取下降最多的。这是训练的主要耗时点，XGBoost/LightGBM 的核心加速就在这里（直方图分桶、预排序、按叶子并行）。
- 停止条件：max_depth、min_samples_leaf、不纯度下降小于阈值。条件越松，树越深，方差越大。

维度	说明
解决的问题	需要人类可读规则的非线性分类/回归；也作为集成模型的基学习器
核心原理	递归地选择使子节点纯度提升最大的特征与阈值做二元切分
时间复杂度	训练 O(n·d·log n)（排序后），预测 O(树深)，单条约 O(depth)

维度	说明
空间复杂度	O(节点数)
优点	可解释、无需标准化、能处理混合类型特征、天然处理非线性与交互
缺点	极易过拟合（不限制可长到纯度 100%）；对样本扰动敏感（高方差）；外推能力差
典型应用	规则引擎、风控策略、医疗诊断辅助、作为 RF/GBDT 的基模型
关键参数 / 调参要点	max_depth、min_samples_leaf/split、max_features、criterion（gini/entropy）、剪枝（ccp_alpha）

5.4 集成学习总览：Bagging 与 Boosting

对比项	Bagging（代表：随机森林）	Boosting（代表：GBDT / XGBoost）
核心思想	并行训练多个强模型，投票平均	串行训练多个弱模型，逐个纠正残差
降低什么	主要降方差	主要降偏差（也降部分方差）
基模型	深树（低偏差、高方差）	浅树（高偏差、低方差），常限制深度 3—8
样本使用	Bootstrap 有放回抽样	每轮对上一轮分错的样本加权
能否并行	✓ 完全并行	✗ 串行（但特征/直方图可并行）
过拟合风险	树数越多越稳，不会过拟合	轮数过多会过拟合，需早停
典型场景	快速拿基线、特征噪声大	追求精度的竞赛/排序/预估

5.5 随机森林 Random Forest（2001，Breiman）★重点

定位：表格数据上的「默认强基线」——几乎不用调参就能拿到不错的效果，还能给出特征重要性(每个特征对模型预测的贡献大小)。理解它的关键是理解「为什么把很多不相关的树平均起来会变好」。



图 5-1　随机森林原理：单棵决策树 → 双重随机采样 → 投票/平均

核心原理：两条随机性 + 一个平均

- 随机性①（行）：对训练集做 Bootstrap 有放回抽样，每棵树看到不同的样本子集。每次抽样约有 $1 - 1/e \approx 36.8\%$ 的样本没被抽中，称为袋外样本（OOB），可直接当验证集用。
- 随机性②（列）：每个节点分裂时只在随机抽取的 m 个候选特征里选最优（分类常用 $m = \sqrt{M}$，回归常用 $m = M/3$）。这一步是随机森林区别于普通 Bagging 的关键，它把树与树之间的相关性压得更低。
- 平均：分类取多数投票，回归取均值。由方差分解可知，T 个相关性为 ρ、方差为 σ^2 的模型平均后，方差降为 $\rho\sigma^2 + (1-\rho)\sigma^2/T$。$\rho$ 越小、T 越大，集成方差越低——这正是两条随机性存在的意义。

代码 5-4　随机森林核心实现（约 30 行，含 OOB 与特征重要性）

```
import numpy as np
from collections import Counter
from sklearn.tree import DecisionTreeClassifier   # 也可换成 5.3 的手写树

class RandomForest:
    def __init__(self, n_estimators=100, max_features="sqrt", max_depth=None):
        self.n, self.mf, self.md = n_estimators, max_features, max_depth
        self.trees, self.feats = [], []

    def fit(self, X, y, seed=0):
        rng = np.random.default_rng(seed)
        n, M = X.shape
        k = int(np.sqrt(M)) if self.mf == "sqrt" else M
        for _ in range(self.n):
            idx = rng.choice(n, n, replace=True)          # ① 行采样（Bootstrap）
            cols = rng.choice(M, k, replace=False)        # ② 列采样
            t = DecisionTreeClassifier(max_depth=self.md, max_features=None)
            t.fit(X[np.ix_(idx, cols)], y[idx])
            self.trees.append(t); self.feats.append(cols)
```

```
        self.oob_score_ = self._oob(X, y)                    # 袋外估计，无需切分验证集
        return self

    def predict(self, X):
        votes = np.array([t.predict(X[:, c]) for t, c in zip(self.trees,
self.feats)])
        # 逐列取众数
        return np.array([Counter(col).most_common(1)[0][0] for col in votes.T])

    def feature_importance(self):
        imp = np.zeros(self.n_features_in_)
        for t, c in zip(self.trees, self.feats):
            imp[c] += t.feature_importances_         # 不纯度下降量累加
        return imp / imp.sum()

    def _oob(self, X, y):
        # 只统计该树未抽到的样本，实现省略
        return np.nan
```

参数 / 符号	含义	取值与影响
n_estimators=100	树的数量 T	越多越稳，但超过 200—500 后收益极小；推理耗时线性增长
max_features=sqrt	每个节点分裂时的候选特征数 k	分类默认 √M、回归默认 M/3。调小→树更独立、方差更低但单树偏差升高
max_depth	单棵树的最大深度	随机森林常用较深甚至不限制的树，靠平均来抑制过拟合
rng.choice(n,n,replace=True)	Bootstrap 有放回抽样	每次约 36.8% 样本落袋外（OOB），可直接当验证集
cols = rng.choice(M,k,replace=False)	列采样：随机选 k 个特征	随机森林区别于普通 Bagging 的关键，进一步降低树间相关性
X[np.ix_(idx, cols)]	同时按行、列取子矩阵	np.ix_ 生成二维索引网格，避免手写嵌套循环
feature_importances_	基于不纯度下降量累加的特征重要性	适合粗筛特征；偏袒高基数特征，严谨分析建议用置换重要性

代码逐段解读：

- 第 1 步：每棵树独立训练，外层循环天然可并行（n_jobs=-1 用满所有核）。
- 第 2 步：双重随机——行采样让每棵树看到不同样本，列采样让每个节点只看部分特征。两者叠加使树与树的相关性 ρ 显著下降。
- 第 3 步：预测时每棵树对同一样本用自己训练时选中的 cols 取特征（X[:, c]），再投票取众数。
- 为什么平均能降方差：T 个相关性 ρ、方差 σ^2 的模型平均后方差为 $\rho\sigma^2+(1-\rho)\sigma^2/T$。ρ 越小、T 越大，集成方差越低。

- OOB 的价值：不需要额外切分验证集就能估计泛化误差，小数据场景非常实用。

维度	说明
解决的问题	中小规模表格数据的分类/回归；快速建立可信基线；评估特征重要性
核心原理	Bagging + 随机特征子空间：用双重随机降低树间相关性，再以投票/平均降低方差
时间复杂度	训练 O(T · n log n · √M)，可完全并行；预测 O(T · depth)，单条通常 < 1 ms
空间复杂度	O(T · 节点数)；100 棵深树约占几十 MB，CPU 即可推理
优点	几乎免调参、抗过拟合、能处理缺失值与非线性、天然给出 OOB 与特征重要性、可并行
缺点	模型体积大、推理比单棵树慢；外推能力差（不能预测超出训练范围的趋势）；高维稀疏数据（文本）上不如线性模型
典型应用	金融风控、医疗诊断、推荐排序特征、Kaggle 表格赛、异常检测（Isolation Forest 同族）
关键参数 / 调参要点	n_estimators（100—500，越多越稳、收益递减）、max_features（√M 或 M/3）、max_depth、min_samples_leaf、n_jobs、class_weight

随机森林 vs GBDT 怎么选　数据量中等、特征噪声大、想快速上线 → 随机森林；追求精度上限、愿意调参、数据干净且量大 → XGBoost/LightGBM。经验值：LightGBM 在 > 1 万样本的结构化数据上通常比随机森林高 1%—5% 的 AUC，但调参成本高一个数量级。

5.6　GBDT / XGBoost / LightGBM

定位：结构化数据上的精度王者。用「加法模型 + 梯度提升」把一堆弱树串起来，每一步新树都在拟合当前模型的负梯度（残差）。

代码 5-5　GBDT 训练骨架（平方损失版；XGBoost 用二阶泰勒展开 + 正则项）

```
import numpy as np

class GBDT:                          # 回归版，最小化平方损失
    def __init__(self, n_estimators=100, lr=0.1, max_depth=3):
        self.n, self.lr, self.md = n_estimators, lr, max_depth
        self.trees, self.base = [], 0.0

    def fit(self, X, y):
        self.base = y.mean()                  # F₀ = 常数（最小化 MSE 的起点）
        pred = np.full(len(y), self.base)
        for _ in range(self.n):
            resid = y - pred                  # 平方损失下，负梯度 == 残差
            t = DecisionTreeRegressor(max_depth=self.md)   # sklearn 或自写回归树
            t.fit(X, resid)
            self.trees.append(t)
```

```
            pred += self.lr * t.predict(X)   # 加法模型：F_m = F_{m-1} + η·h_m
        return self

    def predict(self, X):
        return self.base + self.lr * sum(t.predict(X) for t in self.trees)
```

参数 / 符号	含义	取值与影响
n_estimators=100	提升轮数 M	与学习率联动：学习率降低时应增加轮数；过多会过拟合，需早停
lr=0.1	学习率（ shrinkage η）	最关键的旋钮。0.01—0.3；越小越稳但需更多轮，通常 0.05—0.1 起步
max_depth=3	基树深度	GBDT 用浅树（3—8）；深度 >10 容易过拟合，也失去 Boosting 的意义
resid = y - pred	当前模型的残差	平方损失下残差恰好等于负梯度；换成其他损失时要显式算 $-\partial L/\partial \text{pred}$
self.base = y.mean()	初始常数预测 F_0	最小化 MSE 的最优常数就是均值；分类任务则改为对数几率
pred += lr * t.predict(X)	加法模型更新	乘 lr 是 shrinkage 技巧，牺牲单步进步换取整体泛化

代码逐段解读：

- Boosting 的本质：每一轮都在拟合「当前模型还差多少」，把弱学习器串成强学习器。
- 为什么用浅树：Boosting 降的是偏差，需要的是「高偏差、低方差」的基模型，深树反而引入方差。
- 残差 = 负梯度只在平方损失下成立；XGBoost 的通用做法是对损失做二阶泰勒展开，用一阶 g 与二阶 h 计算分裂增益。
- XGBoost 的分裂增益：$\text{Gain} = \tfrac{1}{2}[G_L^2/(H_L+\lambda) + G_R^2/(H_R+\lambda) - (G_L+G_R)^2/(H+\lambda)] - \gamma$，其中 γ 是新增叶子的代价，起到剪枝作用。
- 调参顺序：先固定一个较大的学习率找合适的轮数（配合早停），再降学习率、按比例加轮数，最后微调树的复杂度参数。

为什么 XGBoost / LightGBM 更快更准

- XGBoost：目标函数(训练时要优化的那个总公式)做二阶泰勒展开（用一阶 g 与二阶 h），并显式加入叶子数、叶子权重的 L2 正则，分裂增益 = $1/2\cdot[G_L^2/(H_L+\lambda) + G_R^2/(H_R+\lambda) - (G_L+G_R)^2/(H+\lambda)] - \gamma$。
- LightGBM：直方图分桶（把连续特征离散成 k 个桶，默认 255）+ 单边梯度采样 GOSS + 互斥特征捆绑 EFB，并按叶子生长（leaf-wise）而非按层生长，速度提升数倍。

- CatBoost：Ordered Boosting 解决类别特征的目标泄漏，对类别型特征开箱即用。

维度	说明
解决的问题	结构化/表格数据的高精度分类、回归、排序（LTR）、CTR 预估
核心原理	前向分步加法模型，每轮用一棵浅树拟合当前损失函数的负梯度
时间复杂度	XGBoost 精确贪心 O(n·d·depth)；LightGBM 直方图 O(n·k·depth)，k≈255
空间复杂度	直方图法内存约为原数据的 1/8；模型体积通常几十 MB
优点	精度高、自动处理缺失值、支持自定义损失、特征重要性可靠、CPU 上训练快
缺点	超参敏感（学习率/树深/轮数需联动调）、串行训练、高维稀疏文本上不如线性/DL
典型应用	推荐/广告排序、风控评分、销量预测、搜索 Learning to Rank
关键参数 / 调参要点	learning_rate（0.01—0.3）、n_estimators + early_stopping、max_depth（3—8）、subsample/colsample、min_child_weight、lambda/gamma、scale_pos_weight

配套仿真：搜索「随机森林 · 多棵树投票」和「梯度提升 · 残差拟合」，两个仿真的差别就是 5.4 讲的 Bagging 与 Boosting：前者是每棵树各自抽样、最后投票，后者是一棵树专门去拟合上一棵的残差。把它们和「决策树 · 信息增益」连起来看，从单棵树到集成模型这条路就走完了。

5.7　支持向量机 SVM

定位：小样本高维数据上的经典强模型，核心是「间隔最大化」与核技巧(用一个函数直接算出高维空间的内积，不必真的升维)。今天在文本分类的基线里仍有一席之地。

代码 5-6　线性 SVM（合页损失）与核技巧说明

```
import numpy as np

class LinearSVM:                        # 合页损失 + SGD
    def __init__(self, C=1.0, lr=0.01, epochs=1000):
        self.C, self.lr, self.epochs = C, lr, epochs

    def fit(self, X, y):
        y = np.where(y <= 0, -1, 1)
        self.w = np.zeros(X.shape[1]); self.b = 0.0
        for _ in range(self.epochs):
            for i in np.random.permutation(len(X)):
                margin = y[i] * (self.w @ X[i] + self.b)
                if margin < 1:          # 落在间隔内或分错 → 更新
                    self.w -= self.lr * (self.w / self.epochs - self.C * y[i] *
X[i])
                    self.b += self.lr * self.C * y[i]
                else:
```

```
                    self.w -= self.lr * self.w / self.epochs   # 仅 L2 衰减
        return self

# 核技巧：K(x, z) = (x·z + c)^d（多项式核）或 exp(-γ‖x-z‖²)（RBF 核）
# 通过核函数把样本隐式映射到高维空间，从而线性可分，无需显式计算特征映射
```

参数 / 符号	含义	取值与影响
C=1.0	软间隔惩罚系数	越大越不容忍分错（接近硬间隔，易过拟合）；越小间隔越宽、容忍越多错分
margin = y·(w·x+b)	函数间隔	>1 表示分类正确且在间隔外；在 (0,1) 之间表示正确但在间隔内；<0 表示分错
合页损失	max(0, 1−margin) 的期望	名字来源于函数形状；只惩罚「间隔内和分错」的样本，这是 SVM 稀疏性的来源
w/epochs 项	L2 正则的梯度	对应 ½‖w‖²，是「最大化间隔」在损失里的体现
kernel / gamma	核函数与 RBF 带宽	gamma 越大单个样本影响范围越小、边界越曲折（易过拟合）
支持向量(真正决定分界线位置的少数关键样本)	决定决策边界的少数样本	只有间隔内和分错的样本对 w 有贡献，其余样本删掉也不影响结果

代码逐段解读：

- 更新逻辑：样本落在间隔外（margin≥1）时只做 L2 衰减；落在间隔内或分错时才加上 C·y·x 的纠正项。
- 为什么叫「最大间隔」：最小化 ‖w‖ 等价于最大化间隔 2/‖w‖，这就是正则项的几何含义。
- 核技巧的精髓：K(x,z) 直接算出高维空间的内积，无需显式构造 φ(x)。RBF 核对应无穷维映射，却只需一次指数运算。
- 核 SVM 的瓶颈：需要存储 n×n 的核矩阵，样本过万就吃不消；这也是工业界大规模场景多用线性 SVM 或线性模型的原因。
- 必须标准化：SVM 依赖距离与间隔，特征量纲不一致会让大尺度特征完全主导结果。

维度	说明
解决的问题	小样本、高维、非线性但边界清晰的分类（文本、基因、图像早期）
核心原理	最大化分类间隔；用核函数隐式映射到高维空间实现非线性分隔
时间复杂度	线性 SVM O(n·d)；核 SVM 训练 O(n²)—O(n³)，预测 O(支持向量数)
空间复杂度	O(支持向量数 · d)，核矩阵 O(n²) 是瓶颈
优点	理论完备、小样本表现好、核函数灵活、间隔思想影响深远（后被用于对比学习）

维度	说明
缺点	百万级样本训练不动；核与 C、γ 需调；概率输出需额外校准
典型应用	文本分类基线、生物信息学、异常检测（One-Class SVM）、手写识别教学案例
关键参数 / 调参要点	C（软间隔惩罚）、kernel（linear/rbf/poly）、gamma、degree、class_weight、是否标准化（必须）

5.8　KNN、K-Means、PCA、朴素贝叶斯速览

算法	一句话原理	复杂度	优点	缺点	典型应用
KNN	近朱者赤：找最近的 k 个邻居投票	朴素 O(n·d)；KD 树/Annoy/HNSW 可近似到 O(log n)	无需训练、天然多分类、实现三行	预测慢、高维失效（维度灾难）、需标准化	推荐召回、相似检索、LLM 的 RAG(检索增强：先从资料库检索，再让模型据此作答)向量检索（本质是 ANN-KNN）
K-Means	交替做「指派簇」与「更新质心」两步	O(t·n·k·d)	简单快、可扩展（Mini-Batch）	需先定 k、只识凸球形簇、对异常值敏感	用户分群、图像压缩、向量量化（VQ，模型压缩思想源头）
PCA	找方差最大的正交方向做投影降维(减少特征个数，同时尽量保留有用信息)	O(min(n·d², d³))或随机化 SVD 更快	去噪、去冗余、可视化、加速训练	线性方法、主成分(数据变化最大的那个方向)难解释、对尺度敏感	特征压缩、可视化（降到 2/3 维）、人脸特征脸、LLM 中 LoRA(低秩适配：冻结原模型、只训练一个很小的旁路)的思想近亲（低秩）
朴素贝叶斯	用贝叶斯定理 + 特征条件独立假设计算后验	训练 O(n·d)，预测 O(d)	极快、小样本可用、天然多分类	独立性假设在现实中几乎不成立	垃圾邮件过滤、情感分类基线、短文本分类

> **速览表读法**　这四类算法通常是「配角」：KNN 用来做相似检索（在 RAG 里是主角）、K-Means 用来分群、PCA 用来降维可视化、朴素贝叶斯用来给复杂模型当 baseline。它们的真正价值是帮你快速看清数据的形状。

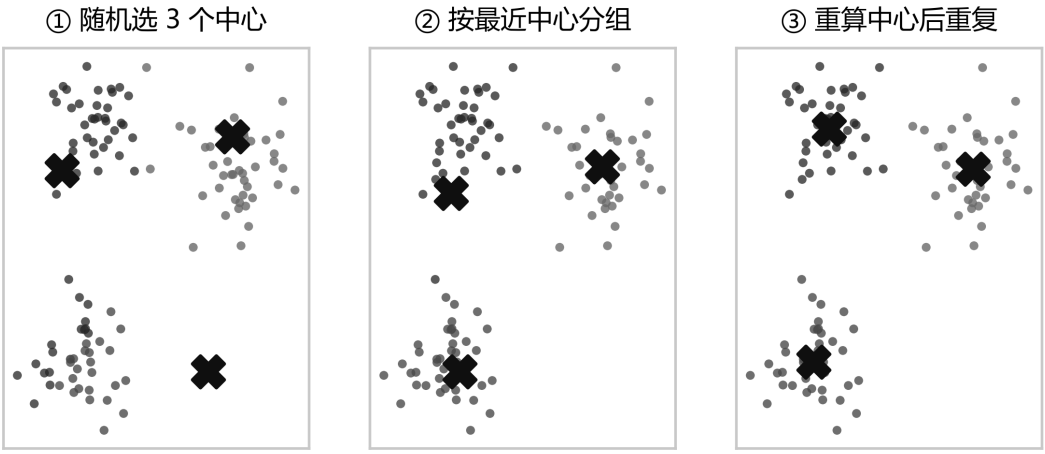


图 5-2　K-Means 的一轮：选中心 → 按最近分组 → 重算中心

5.9　评估指标：别只盯准确率

任务	首选指标	什么时候换指标
二分类（均衡）	Accuracy、F1	—
二分类（不均衡）	PR-AUC、ROC-AUC、召回率(所有真该找出来的里面，实际找出来的比例)@固定精确率	正样本 < 5% 时 Accuracy 毫无意义（全预测负类就有 95%）
回归	MAE、RMSE、R^2	有大异常值时用 MAE；关注相对误差用 MAPE
排序 / 推荐	NDCG@K、MAP@K、命中率	只看 top-K 结果质量时
生成 / LLM	BLEU / ROUGE / 人工评估 / LLM-as-Judge	开放式生成没有唯一标准答案(样本对应的正确结果)，需多维评估

配套仿真：搜索「混淆矩阵」，可以逐个样本地看它落进四个格子里的哪一个，并实时看到精确率、召回率、F1 随之变化。同一分组里的「阈值 · ROC 与 AUC」和「K 折交叉验证」也建议对照着跑一遍：前者演示阈值挪动时两个指标怎么此消彼长，后者演示换一把尺子量出来的分数为什么会抖。

5.10　本章常见错误与练习

机器学习部分的坑，绝大多数出在流程上，而不是概念上。

一是数据泄漏：先对全量数据做了标准化或特征筛选，再划分训练集和测试集，测试集的信息通过均值、方差泄漏进了训练过程，指标虚高，上线即崩。正确顺序永远是先划分，再只在训练集上拟合任何参数。

二是用测试集调参。测试集只能用于最后一次性评估，调参必须用验证集或交叉验证。反复看测试集，等于把它变成了验证集。

三是把相关当因果。模型能告诉你「买啤酒的人常买尿布」，但换成「提高啤酒价格就能多卖尿布」，就需要实验验证。

四是不看基线就上复杂模型。先用「全预测多数类」或逻辑回归得到基线，否则无法判断模型到底有没有用。

五是只看准确率。样本不均衡时，准确率会被多数类主导，要看精确率、召回率、F1 或 AUC。

六是忽略特征工程。树模型对量纲不敏感，但线性模型、KNN、SVM、神经网络都非常依赖标准化；树模型也不能自动处理类别特征的错误编码。

七是把训练集上的表现当最终效果。训练误差低、验证误差高就是过拟合，需要正则化、更多数据或更简单的模型。

三道练习：第一题，用同一份数据分别训练逻辑回归和随机森林，比较两者在验证集上的指标，并说明差异来源。第二题，故意制造一次数据泄漏（在全量数据上标准化），观察指标虚高多少。第三题，为一个不均衡数据集画出混淆矩阵，并解释为什么此时准确率没有意义。

5.11　岭回归：给权重加一根绳子

5.1 的线性回归用最小二乘，闭式解写作 $w = (X^TX)^{-1}X^Ty$。这个公式里藏着一个前提：X^TX 必须可逆。而在真实数据里，这个前提经常不成立。

什么时候会出事？当两个特征高度相关时（比如「面积（平方米）」和「面积（平方英尺）」同时进了模型，或者从同一份原始指标里派生出的几个特征），X^TX 的两列几乎线性相关，整个矩阵接近奇异。这时求逆会把极小的误差放大成极大的权重：模型在训练集上误差很小，换一批数据就崩，权重正负乱跳，看图能看出「为了抵消而互相打架」的成对系数。这种毛病在统计上叫多重共线性。

岭回归（Ridge，Hoerl 与 Kennard，1970）的改法很直接：在原目标函数上加一项惩罚，把「最小化残差平方和」换成「最小化残差平方和 + $\lambda \cdot \Sigma w_j^2$」。解随之变成 $w = (X^TX + \lambda I)^{-1}X^Ty$。多出来的那个 λI 让矩阵的对角线变厚，无论数据多病态都一定可逆，数值上稳定得多。

λ 是这本书里最直观的一个旋钮。λ 趋近 0，它就退化成普通最小二乘；λ 越大，权重被压得越靠近 0，模型越简单、偏差越大、方差越小。5.0 里说的偏差—方差权衡，到这里有了一个可以拧的具体部件。这就是「正则化」这个词最初的样子。

两个工程细节必须记住。第一，惩罚项通常不包含截距 b：把整个预测值上下平移说成「权重太大」没有道理，代码里要把矩阵的 (0,0) 位置那一项扣掉。第二，做岭回归之前必须先把特征标准化，否则量纲大的特征会被惩罚得更狠，各特征承受的惩罚力度不再统一，λ 也就失去了统一的意义。

和 Lasso（L1 正则，1996）的区别值得单独记一笔：L2 把权重整体压小，但很少压到恰好 0，所以所有特征都会保留；L1 会把一部分权重压到等于 0，顺带完成了特征选择。如果两者都需要，把它们加起来就是弹性网（Elastic Net）。

	普通最小二乘	岭回归（L2）	Lasso（L1）

	普通最小二乘	岭回归（L2）	Lasso（L1）
目标函数	$\Sigma(y - Xw)^2$	$\Sigma(y - Xw)^2 + \lambda\Sigma w_j^2$	$\Sigma(y - Xw)^2 + \lambda\Sigma\|w_j\|$
系数特点	无偏，但方差大	整体收缩，趋于小而分散	部分系数被压到恰好 0
能否选特征	不能	不能（只收缩）	能
共线性下	数值不稳、易爆炸	稳定，首选	随机挑一个留下，不稳定
λ 的作用	无	越大越保守	越大留下的特征越少

代码 5-7　岭回归：加 λI 求稳定解（含截距不惩罚的细节）

```
import numpy as np

def ridge_fit(X, y, lam):
    """X: (n, d) 已标准化的特征; y: (n,); lam: 正则强度"""
    n, d = X.shape
    Xb = np.hstack([np.ones((n, 1)), X])       # 补一列 1 给截距
    A = Xb.T @ Xb + lam * np.eye(d + 1)
    A[0, 0] -= lam                             # 截距不参与惩罚
    coef = np.linalg.solve(A, Xb.T @ y)        # solve 比求逆更稳
    return coef[0], coef[1:]                   # (截距, 权重)

rng = np.random.default_rng(0)
n, d = 200, 40
z = rng.normal(size=(n, 1))
X = z + 0.02 * rng.normal(size=(n, d))         # 40 个高度相关的特征
y = X @ np.ones(d) + 0.5 * rng.normal(size=n)  # 真实权重全是 1

for lam in (0.0, 0.01, 1.0, 100.0):
    b, w = ridge_fit(X, y, lam)
    print("lambda=%-7g  |w|=%.3f  max|w|=%.3f"
          % (lam, np.linalg.norm(w), np.abs(w).max()))
# 输出会看到：lambda=0 时权重被放大得很大，lambda 变大后逐步收敛到接近 1
```

> **常见错误**　一是忘了标准化就上岭回归，结果量纲最大的特征被压得最狠，惩罚完全落错了地方。二是把截距也一起惩罚，模型被迫穿过原点，系统性偏差。三是只调 λ 不设验证集：λ 是在验证集上选的，不是越大越好。四是把「权重变小」当成「特征不重要」：岭回归从不把系数压到 0，想筛特征要换 Lasso。

配套仿真：搜索「岭回归」，可以看到同一组共线特征下方程解的权重柱状图怎样随 λ 从剧烈震荡变成平缓收缩，右侧同时给出训练误差与验证误差两条曲线。两条曲线的间隔先缩小后放大，那个最低点就是该取的 λ。

5.12　AdaBoost：把一堆弱模型串成强模型

5.4 把集成学习分成了两条路线：Bagging 让模型并行地学、再平均，随机森林是代表；Boosting 让模型串行地学，后一个专门补前一个的短板。AdaBoost（Freund 与 Schapire，1995）是 Boosting 最早也最著名的实现，也是理解这条路线最省力的入口。

它的基模型弱到什么程度？通常就是「树桩」，也就是只做一次分裂、深度为 1 的决策树，判断能力只比抛硬币好一点。单个树桩几乎没用，但把它们按正确的权重组合起来，能逼近很强的分类器。这不是玄学：每一次新的模型都在修正前面所有模型的错误方向，累积起来就像把一个低分辨率的判断力换成了高分辨率的。

训练是一段三步循环。第一步，给每个训练样本一个权重，初始全部相等。第二步，用当前权重训练出一个弱模型，算出它的加权错误率 e。第三步，按 $\alpha = \frac{1}{2}\cdot\ln((1-e)/e)$ 给它一个票权，然后调大被它分错的样本权重、调小分对的样本权重，进入下一轮。

α 这个式子说明了权重是怎么分配的：错误率 $e = 0.5$ 时 $\alpha = 0$，等于这张票作废；e 越接近 0，α 越大，说话越算数。最终预测是所有弱模型的加权投票，权就是 α。

对比项	AdaBoost	随机森林
训练方式	串行，后一个依赖前一个的结果	并行，每棵树互不知情
样本使用	全量数据，但样本权重每轮调整	每棵树用自助抽样出的子集
主要降低	偏差	方差
对噪声	敏感，被错标的样本权重会越涨越高	较稳健，平均能抵消个体错误
基模型	通常是树桩，越弱越好用	通常是较深的决策树
需要调的参数	轮数、学习率、基模型	树的棵数、每棵的最大深度、特征数

短板也必须说清楚：因为分错的样本权重会被反复调大，AdaBoost 对噪声和离群点格外敏感。一个被标错的样本会让后面所有模型都围着它转，最后把整体带偏。数据脏的时候优先随机森林；数据干净、又需要压偏差时，再考虑 Boosting 家族（它的现代化身就是 5.6 讲的 GBDT / XGBoost）。

代码 5-8　AdaBoost：树桩 + 样本权重 + 加权投票

```
import numpy as np

class Stump:                              # 深度为 1 的决策树
    def fit(self, X, y, w):
        best = None
        for f in range(X.shape[1]):
            for t in np.unique(X[:, f]):
                for s in (1, -1):         # s=1: 小于阈值判 -1
                    pred = np.where(X[:, f] < t, -s, s)
                    err = w[pred != y].sum()
                    if best is None or err < best[0]:
```

```
                        best = (err, f, t, s)
        self.err, self.f, self.t, self.s = best
        return self

    def predict(self, X):
        return np.where(X[:, self.f] < self.t, -self.s, self.s)

def adaboost(X, y, rounds=30):
    n = len(y)
    w = np.full(n, 1.0 / n)              # 初始权重人人相等
    models = []
    for _ in range(rounds):
        m = Stump().fit(X, y, w)
        e = max(m.err, 1e-10)
        if e >= 0.5:                     # 比乱猜还差，这轮丢掉
            continue
        alpha = 0.5 * np.log((1 - e) / e)
        w = w * np.exp(-alpha * y * m.predict(X))   # 分错的权重变大
        w /= w.sum()                                # 重新归一化
        models.append((alpha, m))
    return models

def predict(models, X):
    return np.sign(sum(a * m.predict(X) for a, m in models))
```

> **需要留神的地方**　一是把样本权重当成固定值，忘了每轮归一化，数值会指数级增长到溢出。二是错误率 e 达到或超过 0.5 时还在给它正票权，等于让一个反向的模型参与投票（此时 α 为负，方向已经反了）。三是用很深的树当基模型，弱模型本来就不该太强，够用就行。四是用它处理有大量标错样本的数据。

配套仿真：搜索「AdaBoost」，可以看到样本权重在每一轮之后怎样重新分配：上一轮被分错的点明显变大，决策边界随之一点点挪动，同时右侧列出每个树桩的错误率与票权 α。把错误率调大，能直接看到 α 被压到接近零。

5.13　DBSCAN：按密度找簇，先不用说要几个簇

K-Means 有一个前置条件常被忽略：你得先告诉它簇的个数 k。而现实中经常不知道有几个簇。更麻烦的是，K-Means 隐含假设簇是「球形的、大小差不多、彼此分得开」，一旦数据长成弯月形、环形，或者两个簇之间有一条细密的通道连着，它就会硬把一簇切成两半、或者把两簇合成一簇。

DBSCAN（Ester 等，1996）换了一套判据：不按「离哪个中心最近」分，而按「周围够不够密」分。它需要两个参数：邻域半径 eps，和构成密度的最小点数 minPts。

按这两个参数，所有点被分成三类。核心点：以它为圆心、半径为 eps 的范围内（含它自己）至少还有 minPts 个点，也就是「这一带够密」。边界点：自己不够密，但落在某个核心点的邻域里。噪声点：两类都不满足，直接不归任何簇。

算法只有一句话：从任意一个还没访问的核心点出发，把所有能通过「核心点的邻域」连成一片的点全部收进同一个簇，然后换一个没访问的核心点重复。落在核心点邻域里的边界点会被吸进簇里，真正落单的点保持为噪声。注意「密度可达」是可以传递的：A 能连到 B、B 能连到 C，那么 A、B、C 就在同一个簇里。

好处很明确：不用指定簇数，能发现任意形状的簇，还顺手识别出噪声（这也是它在异常检测里常用的原因）。代价同样明确：eps 稍变一点结果可能差很多，而且不同区域密度相差很大的数据，用同一组参数无法同时照顾好：密度高的区域会被欠分割，稀疏区域会被整片判成噪声。

代码 5-9　DBSCAN：核心点、边界点与噪声

```
import numpy as np

def dbscan(P, eps, min_pts):
    n = len(P)
    D = np.linalg.norm(P[:, None, :] - P[None, :, :], axis=2)
    nb = [np.where(D[i] <= eps)[0] for i in range(n)]   # 每点的邻居
    label = np.full(n, -1)          # -1 = 还没访问, -2 = 暂记噪声
    cid = 0
    for i in range(n):
        if label[i] != -1:
            continue
        if len(nb[i]) < min_pts:    # 不是核心点，先挂起
            label[i] = -2
            continue
        label[i] = cid
        seeds = list(nb[i])
        while seeds:
            j = seeds.pop()
            if label[j] == -2:
                label[j] = cid      # 边界点被收编进簇
            if label[j] != -1:
                continue
            label[j] = cid
            if len(nb[j]) >= min_pts:        # 又是核心点，继续扩散
                seeds.extend(nb[j])
        cid += 1
    return np.where(label < 0, -1, label)   # -2 统一当噪声

rng = np.random.default_rng(7)
P = np.vstack([rng.normal([0, 0], 0.3, (100, 2)),
               rng.normal([3, 3], 0.3, (100, 2)),
               rng.uniform(-4, 6, (20, 2))])         # 两个簇 + 噪声
print(np.bincount(dbscan(P, 0.6, 8) + 1))
# 大致输出 [20 100 100]: 20 个噪声、两个各 100 点的簇
```

> **调参的难点**　一是忘了先把特征标准化，eps 的距离尺度就失去了意义：不同量纲下「半径 0.6」完全是两件事。二是把 minPts 设成 1 或 2，每个点都成了核心点，所有点连成一簇。三是用它处理密度差异很大的数据，还指望一组参数搞定。四是以为噪声点就是错的数据：在异常检测里，噪声点往往正是要找的目标。

配套仿真：搜索「DBSCAN」，左侧把点的三色分类直接画出来，核心点、边界点、噪声点各用一种颜色，右侧实时统计每一轮扩散收进了几个点、还剩几个未访问。把 eps 一点点调大，可以看着两个簇连成一片的那一刻。

5.14　高斯混合与 EM：K-Means 的「软」版本

K-Means 的每个点只属于一个簇，边界是一条直线；而且「属于」是个非黑即白的判断。高斯混合模型（GMM）把这两件事都换成概率：每个簇是一个高斯分布，有自己的均值 μ、协方差 Σ 和权重 π，每个点以不同的概率属于各个簇。

协方差这个自由度是有用的：它让每个簇可以自己决定「胖瘦」和「朝向」，于是簇的形状从球变成椭球。K-Means 做不到这一点，它给所有簇的代价函数是同一个等距的度量。

麻烦在于参数没法一步算出来：你既不知道每个点属于哪个高斯，也不知道每个高斯的参数，两个问题互相依赖。这类「有隐变量」的问题有一个通用解法：EM 算法（期望最大化，Dempster 等，1977），两步交替做。

E 步（期望）：固定住当前的 μ、Σ、π，用贝叶斯公式算出每个点属于每个簇的概率。这个概率在本行里叫责任度（responsibility），可以理解成「这个簇该为解释这个点负多少责任」。

M 步（最大化）：把责任度当成权重，用它重新估计每个高斯的均值、协方差和权重，本质就是加权版的极大似然估计。均值是各点的加权平均，协方差是加权后的散布程度。

重复 E、M 两步，直到参数几乎不再变化。可以证明每一轮都会让数据的对数似然不减（这叫单调性），所以 EM 一定收敛；但它只保证收敛到局部最优，换个初始值可能落到另一个解上。实践中的标准补救是先跑一次 K-Means，用它的结果当初始均值。

和 K-Means 的关系值得单独记一笔：如果把每个高斯的协方差都固定成相同、并且小到可以忽略，E 步算出来的责任度就会退化成「最近的簇为 1、其余为 0」，GMM 随之变成 K-Means。所以 K-Means 是 GMM 的一个「硬分配」特例，而 GMM 是它的软版本。这是 5.2 里「无监督学习」这一块最漂亮的一条联系。

代码 5-10　高斯混合的 EM 迭代：E 步算责任度，M 步重估参数

```
import numpy as np

def gmm_em(X, k, iters=50, seed=0):
    rng = np.random.default_rng(seed)
    n, d = X.shape
    mu = X[rng.choice(n, k, replace=False)]     # 初始均值直接挑 k 个点
    cov = np.array([np.cov(X.T) for _ in range(k)])
    pi = np.full(k, 1.0 / k)
```

```
    def gauss(x, m, c):
        c = c + 1e-6 * np.eye(d)               # 防止协方差奇异
        diff = x - m
        e = -0.5 * diff @ np.linalg.inv(c) @ diff
        return np.exp(e) / np.sqrt((2 * np.pi) ** d * np.linalg.det(c))

    for _ in range(iters):
        # ---- E 步：算责任度（每点属于每簇的概率）----
        R = np.zeros((n, k))
        for i in range(n):
            for j in range(k):
                R[i, j] = pi[j] * gauss(X[i], mu[j], cov[j])
        R /= R.sum(axis=1, keepdims=True) + 1e-300
        # ---- M 步：拿责任度当权重，重新估计参数 ----
        Nk = R.sum(axis=0)                     # 每簇的「有效点数」
        for j in range(k):
            mu[j] = (R[:, j, None] * X).sum(axis=0) / Nk[j]
            diff = X - mu[j]
            cov[j] = (R[:, j, None] * diff).T @ diff / Nk[j]
        pi = Nk / n
    return pi, mu, cov, R

rng = np.random.default_rng(1)
X = np.vstack([rng.normal([0, 0], [0.5, 0.2], (200, 2)),
               rng.normal([4, 4], [0.3, 0.8], (200, 2))])
pi, mu, cov, R = gmm_em(X, 2)
print(np.round(mu, 2))          # 两个均值大致落在 (0,0) 与 (4,4)
print(np.round(pi, 2))          # 两个簇的权重大致各占一半
```

> **提醒**　一是忘了给协方差加一点点对角项，某个簇只剩一个点时协方差变成奇异矩阵，求逆直接报错。二是初始均值全用随机数，运气不好会落到「一个簇吞掉全部数据」的退化解，用 K-Means 初始化能大幅降低概率。三是把责任度直接用 argmax 当成硬标签后就再也不更新了，那就退化成了一次性 K-Means。四是把「似然不再上升」当成「找到了全局最优」，EM 只保证局部最优。

配套仿真：搜索「高斯混合」，可以看到责任度矩阵随迭代而变化：一开始两个簇的边界模糊、很多点两种颜色各占一半，迭代若干轮之后逐渐分明，右侧同时打印对数似然每一轮上升的幅度。它和「K-Means 聚类」那个仿真对照着看，能直观感受到「硬分配」和「软分配」的区别。

5.15　特征工程：把原始数据变成模型能吃的形状

很多人以为机器学习难在选模型，实际上更多时候难在特征。模型只能看到数字，而你手里的原始数据往往是整段文字、日期、地址、商品名。特征工程就是把这些东西翻译成数字的过程。

打个比方：你要向一个不懂中文的朋友描述一个人。你不能说「他很靠谱」，而得说「他迟到过几次、践约率百分之几、平均回复时间多少分钟」。后者就是特征。

表 5-11　四类常见特征变换及其适用场景

原始形式	变换方式	为什么要这么做	例子
类别文本	独热编码（One-Hot）	模型不能直接比大小	城市、性别
连续数值	标准化 / 归一化	避免大数值特征压制小数值	年龄、收入
长尾分布	取对数	压缩极端值，让分布更像正态	价格、点击量
日期时间	分解成周几/小时/是否节假日	让模型看到周期性	销量预测、用户活跃

经验之谈　在表格数据上，好特征 + 简单模型，经常打赢坏特征 + 复杂模型。先花时间看数据，再花时间调模型。

5.16　交叉验证：为什么不能拿训练集当考试

把所有数据都用来训练，然后用训练集上的准确率当成模型水平，是新手最常见的错误。比如把每个样本的 ID 也当特征，模型会把训练集背得滴水不漏，但遇到新数据就一塌糊涂。这叫过拟合。

正确做法是把数据分成三份：训练集（学习）、验证集（调参数）、测试集（最后考试，只能看一次）。如果数据不多，就用 K 折交叉验证：把数据切成 K 份，轮流拿一份做验证、其余 K−1 份训练，最后取平均。打个比方：不能用同一套题当作业又当考试，也不能只出一套题，要多几套轮着考。

表 5-12　数据划分方法的选择

方法	用法	优点	风险
按比例划分 6:2:2	数据量充足	简单直观	切分不同结果差异大
K 折交叉验证（K=5/10）	数据量中等	结果稳定、可估计波动	训练成本×K
分层抽样	类别不均衡	每份类别比例一致	需要额外代码
按时间切分	时间序列预测	避免用未来预测过去	不能随机打乱

铁律　测试集一律只能用一次。反复看测试集调参，等于把它变成了验证集，最后报出的指标就不可信了。

5.17　偏差与方差：模型为什么学不好

模型的错误可以拆成两块：偏差（Bias）和方差（Variance）。偏差是「模型本身的假设就不对」，方差是「模型对数据的小抖动太敏感」。打个比方：打靶。偏差大 = 你瞄准的方向就偏了，所有子弹都落在靶子左上方；方差大 = 你瞄准了，但手抖，子弹散得到处都是。

表 5-13　两种错误的表现、原因与对策

现象	表现	原因	对策
欠拟合（高偏差）	训练集和测试集都差	模型太简单	换更强的模型、加特征
过拟合（高方差）	训练集很好、测试集很差	模型太复杂、数据太少	加数据、正则化、剪枝、集成
数据泄露	线上突然变差	特征里混了标签信息	严格按时间切分、审查特征

> **一句话**　新手最容易只做一件事：把模型越换越复杂。但如果病因是数据量不够或特征泄露，换模型只会让结果更难解释。

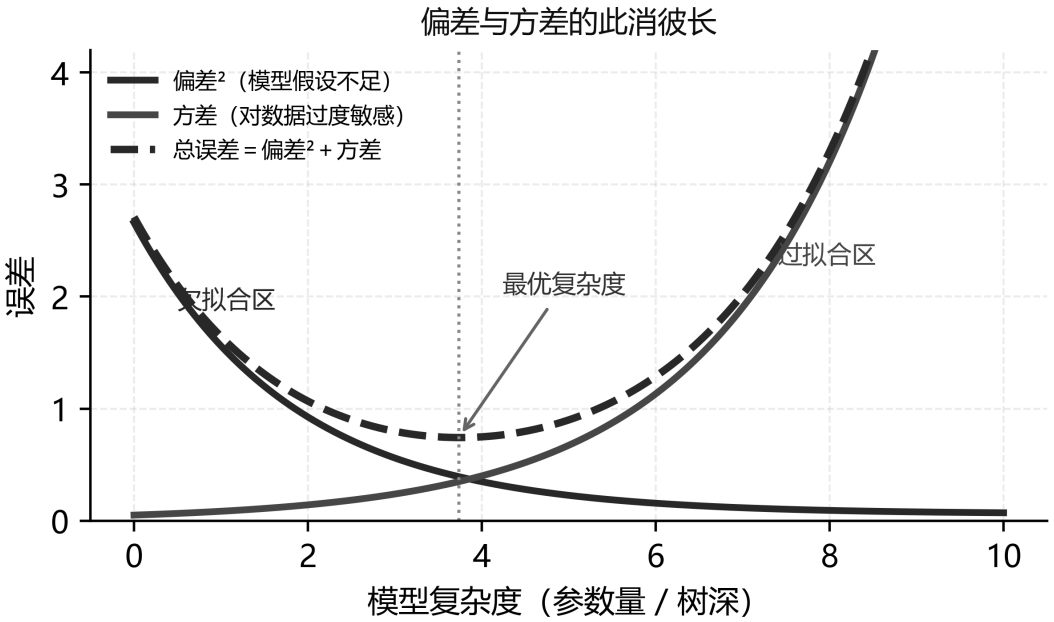


图 5-3　偏差与方差的此消彼长：总误差最低处就是最优复杂度

5.18　正则化的几何直觉：为什么加一项就能防过拟合

5.11 的岭回归在损失里加了 λ·Σw²。从几何上看，这等于给权重加了一个「预算上限」：权重向量的长度被限制在一个小球里，模型只能在这个球里找最优解。预算越小（λ 越大），模型越「僵硬」，越不容易因为几个异常点而扭曲。

L1 正则化（加 λ·Σ|w|）还有一个额外效果：它会把不重要的权重压到恰好为 0，于是完成了「自动选特征」。这是因为它的等高线是有棱角的方块，而最优解往往落在角上。

代码 4-21　带 L2 正则化的梯度下降（看清楚正则项到底做了什么）

```
# w <- w - lr * (dLoss/dw + 2 * lam * w)
#        \____\u539f\u6765\u7684\u68af\u5ea6____/   \_\u6b63\u5219\u9879\u62d6\u7740
w \u5f80 0 \u8d70_/
for epoch in range(epochs):
    grad = compute_grad(X, y, w)      # \u4efb\u52a1\u635f\u5931\u7684\u68af\u5ea6
    w -= lr * (grad + 2 * lam * w)    #
\u6bcf\u90fd\u591a\u8d70\u4e00\u70b9\u70b9\u5411\u96f6
```

> **直觉**　正则项的作用可以理解为「每次更新都把权重往零拉一小步」。训练数据说要往右，正则项说往左，最后停在一个「两边都不算太亏」的位置。

5.19　手推一次梯度下降：用五个点算给你看

梯度下降被称作机器学习的发动机，但它的核心动作极其朴素：算出现在错了多少（损失），看看往哪个方向调能少错一点（梯度），然后朝那个方向挪一小步（学习率）。这一节用五个数据点，把这三步一次一次算出来。

打个比方：你在雾里下山，看不见山脚在哪。能做的只有三件事——用脚感受脚下哪边更低（梯度），朝那边迈一小步（更新），别迈太大免得冲过头（学习率）。

5.19.1　准备一个只有五个点的数据集

假设我们要拟合 y = w·x（一条过原点的直线）。数据是：x = 1, 2, 3, 4, 5，对应的 y = 3, 5, 7, 9, 11。这组数据其实来自 y = 2x + 1；但因为我们假设模型强制过原点（没有截距项），最好的拟合不是 w = 2，而是 Σxy ÷ Σx² = 125 ÷ 55 ≈ 2.273。这件事本身就值得记住：模型的形式决定了它能达到的上限。我们从 w = 0 开始，看梯度下降能不能自己找到这个数。

损失函数用均方误差 MSE：L = (1/n) Σ (w·x_i - y_i)²。它对 w 的导数是 (2/n) Σ (w·x_i - y_i)·x_i。这个式子很好记：误差乘上输入，再平均。

5.19.2　第一次迭代：算清楚每一个数

```
x = [1, 2, 3, 4, 5]
y = [3, 5, 7, 9, 11]
w = 0.0
```

```
lr = 0.01
```

这一版在演示什么　把 H 手算梯度下降的三次迭代写成代码，让机器算一遍。

梯度告诉我们的是一件事：往哪个方向调参数，损失会下降得最快；以及这个方向有多陡。学习率则决定每次迈多大一步。所以更新公式就是「新参数 = 旧参数 − 学习率 × 梯度」。

- 第一步：用当前参数算出预测值。
- 第二步：用预测值和真实值的差算出损失和梯度。
- 第三步：按更新公式调一次参数，然后重复。

看这三个数怎么变　参数 w 从 0 一步步靠近理论值，梯度从 -50 缩小到 -30.4——梯度自动变小，说明离谷底越来越近，步长也就自然放慢了。这就是梯度下降能收敛的直觉。

```
for step in range(1, 4):
    pred = [w * xi for xi in x]
    errs = [p - yi for p, yi in zip(pred, y)]
    grad = 2 * sum(e * xi for e, xi in zip(errs, x)) / len(x)
    loss = sum(e * e for e in errs) / len(x)
    print(step, "w =", round(w, 4), "loss =", round(loss, 3), "grad =", round(grad,
3))
    w = w - lr * grad
```

代码 5-40　一元线性回归的梯度下降（三次迭代）

表 5-22　从 w = 0 出发的三次迭代（学习率 0.01）

迭代	w（更新前）	预测值	误差	梯度	损失 MSE	更新后 w
1	0.0000	0, 0, 0, 0, 0	-3, -5, -7, -9, -11	-50.000	57.000	0.5000
2	0.5000	0.5, 1.0, 1.5, 2.0, 2.5	-2.5, -4.0, -5.5, -7.0, -8.5	-39.000	34.750	0.8900
3	0.8900	0.89, 1.78, 2.67, 3.56, 4.45	-2.11, -3.22, -4.33, -5.44, -6.55	-30.420	21.213	1.1942

三次迭代，w 从 0 涨到 1.194，正在朝 2.273 靠近。同时注意梯度从 -50 缩小到 -30.4，损失从 57 降到 21.2——梯度在变小，说明离谷底越来越近了。这个「梯度自动变小」的现象，就是梯度下降能收敛的原因。

5.19.3　学习率：唯一一个必须手调的参数

把学习率从 0.01 改成 0.1 会发生什么？第一次更新 w 就跳到 4.4，直接冲过正确答案 2；第二次梯度变成正的，往回弹；学习率到 0.1 时，w 开始来回震荡且幅度越来越大；

到 0.25 时直接发散。这条分界线在哪里？当 $|1 - \eta \cdot \Sigma x^2| > 1$ 时就会发散，这里 $\Sigma x^2 = 55$，所以 $\eta > 2/55 \approx 0.0364$ 就开始震荡，$\eta > 0.0909$ 就会发散。

表 5-23　学习率对收敛的影响（同一个问题）

学习率	第一次更新后的 w	现象	结论
0.001	0.050	移动极其缓慢	要几千次迭代才到，太保守
0.01	0.500	稳步逼近最优值	合适
0.05	2.500	一步就略微冲过最优值 2.273	会来回震荡几次，然后收敛
0.1	5.000	冲过头，而且越走越远	发散，损失越来越大
0.25	12.500	直接飞出去	严重发散，完全无法收敛

怎么判断学习率太大　训练损失不再下降，反而上下跳动甚至越来越大。这时先降一个量级再试。反过来，如果损失下降得像蜗牛爬，就用学习率预热或者干脆调大十倍。

5.19.4　从一元到多元：同一套动作

有多个权重时，每个权重要算各自的偏导数，然后同时更新。注意「同时」两个字：必须先用旧的 w 算完所有梯度，再一起改，不能边算边改。这也是新手写代码时最容易犯的错误之一。

它在解决什么　特征一多，逐个变量写循环就慢得没法看，必须换成矩阵运算。

向量化说白了就是「把循环交给底层的数学库」。同样是算一批预测值，写成 for 循环是一个数一个数地乘加；写成矩阵乘法，底层会一次算完，而且能用上 CPU 的多核和 SIMD 指令，往往快几十倍。

- 第一步：把样本堆成矩阵 X，参数堆成列向量 w。
- 第二步：全部预测值就是一次矩阵乘法 X·w 加偏置、再过一遍激活函数。
- 第三步：梯度也用矩阵形式表达，整个更新过程只有几行。

所以写这类代码时，一条基本原则是：只要能在脑子里把它想成矩阵运算，就不要写双重循环。这不仅更快，代码也更短、更不容易出错。

易错点　矩阵乘法的形状必须对得上。养成习惯：写之前先用小数字标一下每个矩阵的维度，能避免绝大多数调试。

```
for step in range(1000):
    pred = X @ w + b                    # 一次算完所有样本
    err  = pred - y
    gw = 2 * X.T @ err / len(y)         # 权重的梯度（向量）
```

```
    gb = 2 * err.mean()                    # 偏置的梯度（标量）
    w -= lr * gw                           # 同时更新
    b -= lr * gb
```

代码 5-41　多元线性回归的向量化写法

短短六行就是一个能跑的线性回归。第 6 章讲深度学习的反向传播时，你会发现结构和这里完全一样，只是「预测」那一步换成了多层网络，梯度要靠链式法则一层层往回传。

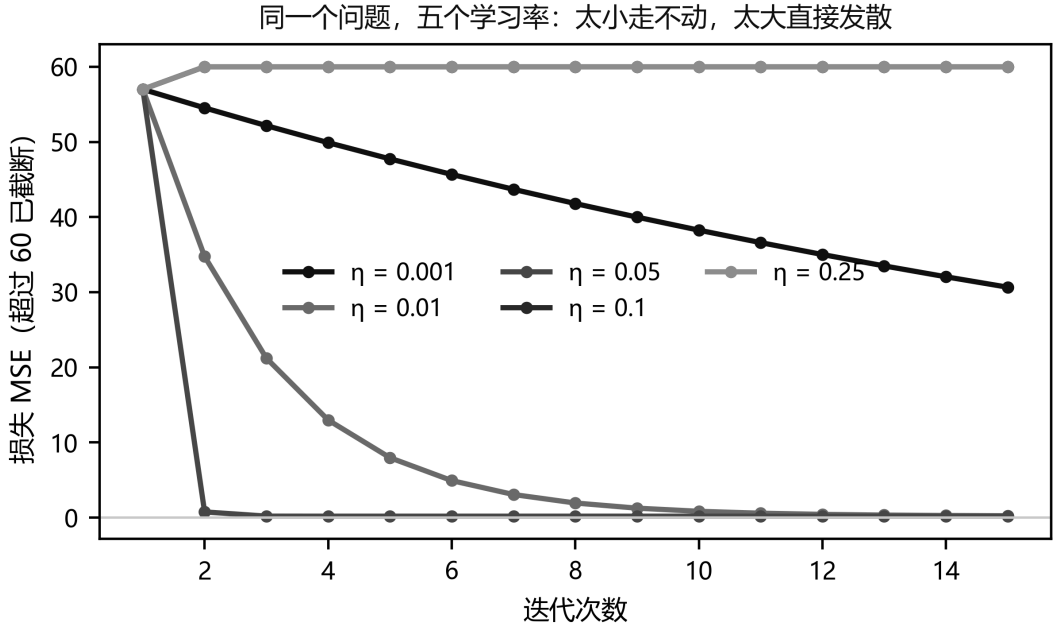


图 5-4　同一个问题与五个学习率：太小走不动，太大直接发散

5.20　特征缩放与数据泄漏：新手最容易踩的两个坑

这两个坑的共同特点是：代码不会报错，结果也「看起来正常」，但模型效果莫名其妙地差。更麻烦的是，它们都很隐蔽，除非有人专门指出来，否则很难自己发现。

5.20.1　坑一：特征量纲差一百倍

设想两个特征：年龄（18–60）和年收入（30000–300000）。在计算距离或者做梯度下降时，年收入这一项会完全压倒年龄——不是因为它更重要，仅仅因为它数字大。模型学到的其实是「谁的收入高」，年龄这个信息被淹没了。

表 5-24　两种缩放方法的对比

方法	公式	缩放后范围	什么时候用
归一化（Min-Max）	(x - min) / (max - min)	0 到 1	分布有明确上下界，如图像像素

方法	公式	缩放后范围	什么时候用
标准化（Z-Score）	(x - 均值) / 标准差	均值 0、方差 1	大多数情况，尤其是有离群点时

> **关键细节**　缩放用的 min/max/均值/标准差，必须只用训练集算出来，然后原封不动地应用到验证集和测试集。用全量数据算，就是下一个坑。

5.20.2　坑二：数据泄漏

数据泄漏是指「模型在训练时看到了它不该看到的信息」。最经典的情形是：先对全体数据做缩放或填充缺失值，再划分训练集与测试集。这样测试集的分布已经通过均值和最值「渗透」进了训练过程，测试成绩会虚高。

```
# 错误做法：泄漏
scaler.fit(all_X)                  # 用了全部数据算均值
X_train, X_test = split(all_X)
```

> **这一版在强调什么**　先划分训练集和测试集，再对训练集做缩放——顺序反了就等于作弊。

很多特征量纲差得很远：年龄是两位数，年收入是六位数。不处理的话，收入这一个特征会独占距离和梯度的绝大部分权重，其它特征等于白给。所以要把每个特征缩放到差不多的范围，常用做法是减均值再除标准差。

问题的关键在于顺序。缩放用的均值和标准差必须只从训练集算出来，然后用这一套参数去转换测试集。如果你先把两者合起来求均值，测试集的信息就提前泄漏进了训练过程。

- 第一步：先切分，训练集和测试集分开。
- 第二步：只在训练集上 fit，算出均值和标准差。
- 第三步：用同一组参数 transform 训练集和测试集。

这叫数据泄漏，是新手最常犯、也最难发现的错误之一：模型在本地表现奇好，上线一用就崩。

> **一个判断标准**　测试集只能被「转换」，不能被「拟合」。写代码时如果看到 scaler.fit 后面跟着测试集，基本就是漏了。

```
# 正确做法：先划分，再用训练集拟合
X_train, X_test = split(all_X)
scaler.fit(X_train)                # 只用训练集
X_train = scaler.transform(X_train)
X_test  = scaler.transform(X_test)
```

代码 5-42　划分与缩放的正确顺序

表 5-25　五种常见的数据泄漏与它们的表现

泄漏形式	怎么发生的	后果
缩放/填充泄漏	用全量数据算统计量	测试指标虚高，上线后暴跌
时间泄漏	用未来的数据预测过去	回测很美，实盘亏钱
重复样本泄漏	同一条样本同时出现在训练集和测试集	模型只是背下了答案
特征泄漏	特征里含有标签的信息	线下完美，线上完全不可用
交叉验证泄漏	在交叉验证外层做特征选择	选择过程用到了验证折的信息

最后一条最隐蔽：如果你先用全部数据筛选出「最好的 20 个特征」，再做交叉验证，验证成绩会被高估，因为筛选时已经看过了所有折的标签。正确做法是把特征选择放进交叉验证的每一折里。

5.21　手写 KNN 与 K-Means：两个算法的完整走一遍

机器学习里有两个名字很容易混的算法：KNN（K 近邻）和 K-Means（K 均值）。它们都以 K 开头，但一个是监督学习（要用标签），一个是无监督学习（不用标签）。这一节把两个都手写一遍，顺便把区别讲清楚。

5.21.1　KNN：少数服从多数

KNN 的全部思想是：要判断一个新样本属于哪一类，就看离它最近的 K 个已知样本里哪一类最多。没有训练过程，所有计算都发生在预测时——这叫「懒惰学习」。

> **一句话原理**　想知道新样本是什么，就看和它最像的那几个样本是什么。

KNN 没有任何训练过程，它把全部样本原样记下来，用的时候现算。这三个步骤——算距离、排序、投票——就是它的全部。也正因如此，它的预测代价很高：每来一个新样本，都要和所有训练样本比一次。

- 第一步：算新样本到每一个训练样本的距离（通常用欧氏距离）。
- 第二步：按距离从小到大排序，取前 k 个。
- 第三步：看看这 k 个里哪一类最多，就把新样本判成哪一类。

k 的取值很关键：k 太小容易被噪声带偏，k 太大又会被多数类吞掉少数类。工程上常用交叉验证来挑，而不是拍脑袋定。

> **为什么必须先做缩放**　KNN 完全依赖距离。如果收入是十位数、年龄是两位数，距离实际只反映了收入，其它特征形同虚设。

```
def knn_predict(train_X, train_y, q, k=3):
```

```
    d = [(sum((a-b)**2 for a, b in zip(p, q)), lab) for p, lab in zip(train_X,
train_y)]
    d.sort()                                    # 按距离从小到大
    topk = [lab for _, lab in d[:k]]
    return max(set(topk), key=topk.count)      # 投票
```

代码 5-43　KNN 的核心：算距离、排序、投票

> **K 怎么选**　K 太小会被噪声带偏（K = 1 时一个异常点就能改变结果），K 太大则会把远处的类别也算进来，边界变糊。实际中常用 3、5、7，并且取奇数避免平票。

5.21.2　K-Means：反复做两件事

K-Means 要在一堆没有标签的点里找出 K 个簇。它的做法是不断重复两个动作：把每个点分给最近的中心，把每个中心移到它那群点的平均位置。重复到中心不再移动为止。

> **一句话原理**　先猜几堆，然后不停地「按最近的中心分组、再按分好的组重算中心」，直到分组不再变。

K-Means 是典型的先有鸡还是先有蛋的问题：想分组得先知道中心，想算中心得先有分组。它的解法是——两边都不知道就先瞎猜一个，然后交替改进。

- 第一步：随机挑 k 个点当作初始中心，或者用 k-means++ 挑得分散一些。
- 第二步：把每个样本分给离它最近的那个中心。这一步叫做分配。
- 第三步：把每个中心挪到它那一组的平均值上。这一步叫做更新。
- 第四步：重复第二步和第三步，直到分组（或中心）不再变化。

为什么它一定会停？因为这两步都不会让「所有点到各自中心的距离平方和」变大：分配会减小它，更新也会减小它，而这个量不可能无限减小。所以过程必然收敛。

> **两个常见坑**　k 值要事先定好，可以用肘部法辅助判断；随机初始中心可能落到很差的解，所以工程上要跑多次取最好的那次。

```
def kmeans(X, k, iters=20):
    centers = X[:k]                            # 随便取 k 个点当初始中心
    for _ in range(iters):
        groups = [[] for _ in range(k)]
        for p in X:
            j = min(range(k), key=lambda c: dist(p, centers[c]))
            groups[j].append(p)                # 分配
        for j in range(k):
            if groups[j]:
                centers[j] = avg(groups[j])  # 更新
    return centers, groups
```

代码 5-44　K-Means：分配与更新交替进行

表 5-26　KNN 与 K-Means 的区别（名字像，本质完全不同）

对比项	KNN	K-Means
学习类型	监督学习（需要标签）	无监督学习（不需要标签）
K 的含义	看最近的 K 个邻居	分成 K 个簇
训练阶段	没有，全部在预测时算	需要反复迭代
预测阶段	要跟所有训练样本算距离	只看最近的簇中心
主要用途	分类、回归	聚类、压缩、初始化

最后一句提醒：K-Means 的结果依赖初始中心的选择，选得不好可能收敛到很差的局部解。工程上的标准做法是跑多组随机初始（比如 10 组），取簇内距离平方和最小的那组，这就是 K-Means++ 及其实践。

5.22　手推一次线性回归：三行数据算到底

前面几节把线性回归的来龙去脉讲完了，但很多同学合上书还是不知道「训练」这两个字在实际执行时到底发生了什么。这一节我们用三个数据点，把每一次权重的更新都算给你看。

数据只有三行，就是一条直线上的三个点：

表 5-27　用来手推的最小数据集

样本	x（输入）	y（真实值）
1	1	3
2	2	5
3	3	7

我们只用模型 $y = w \cdot x$（先不考虑截距 b，也就是假设直线一定过原点）。损失函数用均方误差 $MSE = (1/n) \cdot \Sigma(\text{预测} - \text{真实})^2$。

梯度公式：对 w 求导，得到 $\partial MSE/\partial w = (2/n) \cdot \Sigma[(w \cdot x_i - y_i) \cdot x_i]$。这一行的意思是：每个样本的误差，按照它自己的 x 大小来「分配责任」——x 越大的样本，权重取错时造成的误差越大，所以它对梯度的贡献也越大。

更新规则：$w \leftarrow w - \eta \cdot (\partial MSE/\partial w)$。$\eta$ 是学习率，这里取 0.05。初始 $w = 0$。

第 1 次迭代：预测值全是 0，误差分别是 −3、−5、−7。梯度 $= (2/3) \cdot [1 \times (-3) + 2 \times (-5) + 3 \times (-7)] = (2/3) \times (-34) = -22.667$。新权重 $w = 0 - 0.05 \times (-22.667) = 1.133$。损失从 27.667 降到 7.972。

第 2 次迭代：用 w = 1.133 重新预测，得到 1.133、2.267、3.400，误差变成 −1.867、−2.733、−3.600。梯度 = (2/3)×(−18.133) = −12.089，新权重 w = 1.133 + 0.604 = 1.738，损失降到 2.370。

把前六次迭代整理成表，你会看到一条非常典型的收敛曲线：前几步降得很快，后面越来越慢，慢慢逼近理论最优值。

表 5-28　六次迭代的完整过程（η = 0.05）

迭代	w	预测值（三个样本）	损失 MSE
初始	0.000	0.000 / 0.000 / 0.000	27.667
第 1 次	1.133	1.133 / 2.267 / 3.400	7.972
第 2 次	1.738	1.738 / 3.476 / 5.213	2.370
第 3 次	2.060	2.060 / 4.120 / 6.180	0.776
第 4 次	2.232	2.232 / 4.464 / 6.696	0.323
第 5 次	2.324	2.324 / 4.648 / 6.971	0.194
第 6 次	2.373	2.373 / 4.745 / 7.118	0.157
理论最优	2.429	2.429 / 4.857 / 7.286	0.000

最后一行是闭式解：对这个数据集，令导数为零可以解出 $w^* = \Sigma x_i y_i / \Sigma x_i^2 = (1\times3 + 2\times5 + 3\times7) / (1^2 + 2^2 + 3^2) = 34 / 14 \approx 2.429$。注意它不是整数 2，也不是 2.5——这正是很多同学「目测直线」时估错的地方。

> **三个必须记住的观察**　一是前几步降得最快，因为梯度最大；二是越接近最优，步子越小，这是梯度自然衰减的结果；三是学习率太大会在这一步附近来回跳，不会收敛。

代码 5-45　用 15 行代码复现上面这张表

```
def train_lr(xs, ys, lr=0.05, steps=6):
    n = len(xs)
    w = 0.0
    for k in range(steps + 1):
        err = [w * x - y for x, y in zip(xs, ys)]
        mse = sum(e * e for e in err) / n
        grad = 2 * sum(e * x for e, x in zip(err, xs)) / n
        print("第 %d 次  w=%.3f  MSE=%.3f" % (k, w, mse))
        w -= lr * grad
    return w
print("理论最优 w =", 34 / 14)
train_lr([1, 2, 3], [3, 5, 7])
```

5.23　混淆矩阵与三个指标：用一次体检打比方

模型训好了，怎么判断它好不好？大多数人的第一反应是「算准确率」。但准确率是本书里最容易骗人的一个数字，下面这个例子会让你记一辈子。

假设有 1000 个人参加一项筛查，其中真正有病的只有 10 人。现在有一个偷懒的模型，它对所有人一律输出「没病」。这个模型的准确率是多少？990 / 1000 = 99%。听起来非常好，但它一个病人都没查出来，实际价值是零。

问题出在「准确率」把两类错误混在一起数了。要看清模型到底哪里错了，得把预测和真实情况交叉列成一张表，也就是混淆矩阵。

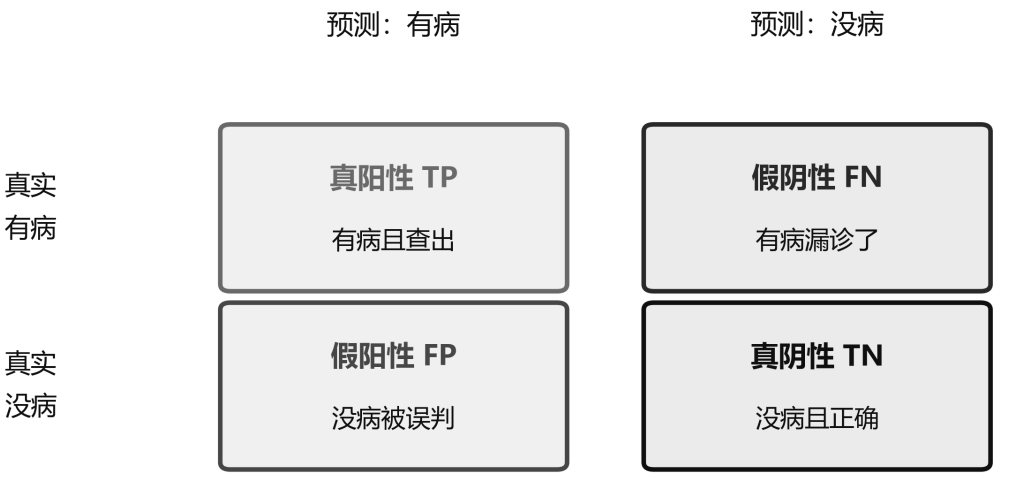


图 5-5　混淆矩阵：四个格子对应四种结局，两类错误必须分开看

表 5-29　四个格子与三个指标

名称	含义	公式	什么场景最关心
真阳性 TP	有病且被查出来	—	筛查、风控的正样本命中
假阴性 FN	有病但漏掉了	—	医疗诊断、故障预警（最怕漏）
假阳性 FP	没病却被误判	—	垃圾邮件、推荐打扰（最怕误杀）
精确率 Precision	判为「有病」的人里，真的有多少有病	TP / (TP + FP)	误判代价高的场合
召回率 Recall	真正有病的人里，被找出来多少	TP / (TP + FN)	漏掉代价高的场合

名称	含义	公式	什么场景最关心
F1 值	精确率与召回率的调和平均	2PR / (P + R)	两者都要兼顾时的单一指标

精确率和召回率是一对天生互相拉扯的指标。模型变「谨慎」一点，少判一些阳性，精确率就上去了，但漏掉的也变多了，召回率就下来了；反过来也一样。所以调模型时真正在做的事，是在这条曲线上选一个合适的点，而不是把两个指标同时拉满。

至于那个 99% 准确率的偷懒模型，它的召回率是 0 / 10 = 0%。一句话就把它打回原形。

> **上手建议**　只要类别分布不均衡，就不要只看准确率。报结果时至少同时给精确率和召回率，最好再给出混淆矩阵本身的四个数字。

代码 5-46　三行算出精确率、召回率与 F1

```
def metrics(tp, fp, fn):
    p = tp / (tp + fp) if tp + fp else 0.0
    r = tp / (tp + fn) if tp + fn else 0.0
    f1 = 2 * p * r / (p + r) if p + r else 0.0
    return p, r, f1
print(metrics(tp=0, fp=0, fn=10))    # 偷懒模型：0.0 0.0 0.0
```

5.24　手推一棵决策树：信息增益一步步算

决策树是「可解释模型」的代表：它给出的不是一堆权重，而是一串「如果……那么……」。这一节我们用八个样本，把「选择分裂特征」的那一步算到底。

数据集是这样的：一共 8 天的记录，特征只有「天气」（晴或雨），标签是「要不要带伞」。其中晴天 5 天，有 2 天带了伞；雨天 3 天，全都带了伞。

决策树在选择分裂特征时，标准是「分完之后，不确定性下降得最多」。不确定性的度量叫信息熵，公式是 $H = -\Sigma\, p_i \cdot \log_2 p_i$。

第一步，算分裂前的不确定性。整体 8 天里，带伞 5 天、不带伞 3 天：$H(\text{整体}) = -(5/8)\cdot\log_2(5/8) - (3/8)\cdot\log_2(3/8) = 0.4238 + 0.5306 = 0.9544$。

第二步，算按天气分完之后的两组各自的不确定性。晴天组 5 天里，带伞 2 天、不带伞 3 天：$H(\text{晴}) = -(2/5)\cdot\log_2(2/5) - (3/5)\cdot\log_2(3/5) = 0.5288 + 0.4422 = 0.9710$。雨天组 3 天全都带伞，完全没有不确定性：$H(\text{雨}) = 0$。

第三步，按样本比例加权求平均，再从原来的熵里减掉，得到信息增益：$IG = 0.9544 - [(5/8)\times 0.9710 + (3/8)\times 0] = 0.9544 - 0.6068 = 0.3476$。

表 5-30　信息增益的完整计算过程

步骤	计算内容	结果
① 总体熵	$-(5/8)\log_2(5/8) - (3/8)\log_2(3/8)$	0.9544

步骤	计算内容	结果
② 晴天的熵	$-(2/5)\log_2(2/5)-(3/5)\log_2(3/5)$	0.9710
③ 雨天的熵	3 天全部带伞，只有一类	0.0000
④ 加权平均熵	$(5/8)\times0.9710+(3/8)\times0$	0.6068
⑤ 信息增益	① − ④	0.3476

0.3476 就是「按天气分裂」这件事的价值。如果还有别的特征（例如温度、湿度），就把每个特征都算一遍增益，选最大的那个作为分裂点，然后对分出来的两组递归做同样的事——这就是 ID3 算法；C4.5 用「增益率」做了修正，避免偏向取值特别多的特征；CART 则改用基尼系数。三种实现的目标一致：让分出来的子集尽可能纯。

> **一个容易踩的坑**　信息增益天然偏爱取值多的特征。极端情况下，把「样本编号」当特征，每个编号恰好对应一个样本，增益会高得离谱，但这样的树毫无泛化能力。

5.25　类别不平衡：五条可以照着做的办法

现实中的数据几乎永远是不平衡的：欺诈交易可能只占千分之一，罕见病可能只占万分之一。这时候模型只要全部预测成多数类，就能拿到很高的准确率，但毫无用处。

表 5-31　处理类别不平衡的五种办法

办法	怎么做	优点	缺点
重采样（欠采样）	从多数类里随机丢掉一部分样本	训练快，实现最简单	丢信息，可能把重要样本扔掉
重采样（过采样）	把少数类样本复制多份	不丢信息	容易过拟合到那几份重复样本
合成样本（SMOTE）	在少数类样本之间插值造新样本	比简单复制更丰富	在少数类极度稀疏时造出的样本不真实
类别权重	在损失函数里给少数类更大的权重	不改数据，只改训练目标	权重要调，太大反而让模型偏向另一个方向
换评价指标	不看准确率，改看召回率、F1 或 AUC	从根上避免被误导	不解决数据本身的偏差

这五条里，最容易被忽略的是最后一条。很多所谓的「不平衡问题」，其实是「指标选错了问题」——只要你关注的指标变成了召回率，模型的表现一下子就有了区分度。

> **推荐顺序**　先换指标，再加类别权重，最后才考虑造数据。顺序反过来，你会花大量时间在采样上，却发现指标根本没改善。

5.26 怎么读一篇机器学习论文

到了这个阶段，你已经有能力去读原始论文了。但论文的写法是为同行评审服务的，不是为初学者服务，所以需要一套读法。

表 5-32 读论文的三遍法

遍数	读什么	花多久	读完应该知道
第一遍	标题、摘要、图表、结论	5 分钟	它想解决什么问题，结论是什么
第二遍	引言、方法框图、实验设置	30 分钟	它是怎么做的，和谁比，赢在哪
第三遍	公式推导、超参数、消融实验	2 小时以上	每一步为什么这么做，去掉会怎样

绝大多数论文只需要读第一遍。判断值不值得读到第三遍，看一个问题：它的方法你能不能用在自己的问题上。能，就细读；不能，读个摘要、记住名字就够了。

> **给新手的提醒** 不要从最难的那篇开始读。先找一篇有开源代码、有中文解读的经典论文（例如随机森林、Transformer），确保自己能在两天内跑通它的代码，再去看更难的。

第 6 章　深度学习：让模型自己学表示

> **读前提示**　建议先读第 5 章。核心只需要理解一件事：「损失 → 算梯度 → 沿梯度更新参数」这个循环。第 0 章 0.3 的③讲过梯度是什么。

机器学习的核心瓶颈曾经是「特征工程」——需要专家手工设计特征。深度学习把这件事交给多层非线性变换：底层学边缘/词组，中层学部件/句法，高层学语义/物体。代价是需要更多数据、更多算力，以及更难的解释。

6.1　多层感知机与反向传播

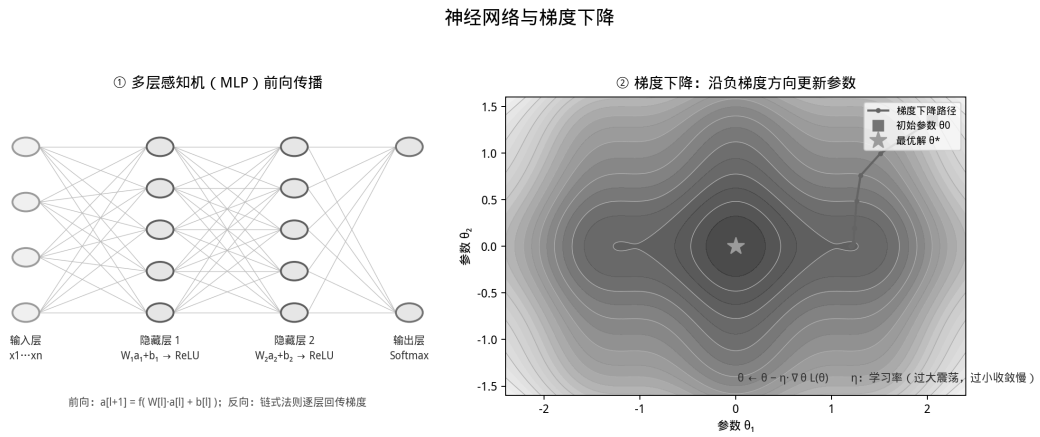


图 6-1　神经网络前向传播(从输入一路算到输出的过程)结构与梯度下降在损失曲面上的收敛路径

核心原理

MLP 就是「线性变换 + 非线性激活」的反复堆叠：a[l+1] = f(W[l]·a[l] + b[l])。没有非线性激活，无论堆多少层都等价于一层线性变换——这是深度学习成立的第一前提。训练靠反向传播：利用链式法则从损失出发，逐层回传梯度，再用优化器(决定「怎么沿梯度更新参数」的算法)更新参数。

代码 6-1　手写两层 MLP 的前向与反向传播（NumPy）

```
import numpy as np

# 一个 2 层 MLP 的手写实现（含反向传播）
def relu(z):        return np.maximum(0, z)
def relu_grad(z):   return (z > 0).astype(float)

rng = np.random.default_rng(0)
W1 = rng.normal(0, 0.01, (784, 256)); b1 = np.zeros(256)
```

```
W2 = rng.normal(0, 0.01, (256, 10));  b2 = np.zeros(10)

def forward(x):
    z1 = x @ W1 + b1; a1 = relu(z1)
    z2 = a1 @ W2 + b2
    p = np.exp(z2 - z2.max()); p /= p.sum()      # Softmax（减最大值防溢出）
    return z1, a1, z2, p

def step(x, y, lr=1e-3):
    global W1, b1, W2, b2
    z1, a1, z2, p = forward(x)
    dz2 = p.copy(); dz2[y] -= 1                  # 交叉熵对 logits 的梯度
    dW2 = np.outer(a1, dz2); db2 = dz2
    da1 = W2 @ dz2
    dz1 = da1 * relu_grad(z1)                    # 链式法则回传第一层
    dW1 = np.outer(x, dz1); db1 = dz1
    W2 -= lr * dW2; b2 -= lr * db2               # 梯度下降更新
    W1 -= lr * dW1; b1 -= lr * db1
```

参数 / 符号	含义	取值与影响
W1 / W2	两层权重矩阵 (784×256)、(256×10)	行=输入维、列=输出维；这个约定让前向写成 x@W 而不是 W@x
b1 / b2	偏置向量	给每个神经元一个可学习的平移量，否则决策面必须过原点
rng.normal(0,0.01,…)	权重初始化	标准差不能太大（激活饱和）也不能太小（信号消失）；0.01 是教学值，实际用 He/Xavier 初始化
z1,a1 / z2,p	线性输出与激活输出	z 是激活前、a 是激活后，反向传播时必须用 z 算梯度
relu_grad(z1)	ReLU 的导数：z>0 为 1 否则 0	ReLU 在 0 处不可导，工程上取 z>0 即可，不影响训练
dz2 = p; dz2[y] -= 1	交叉熵对 logits 的梯度	Softmax+交叉熵的梯度恰好是 p−onehot，这是该组合被广泛使用的原因之一
lr=1e-3	学习率	神经网络最敏感的超参；用 AdamW 时常见 1e-4—1e-3

代码逐段解读：

- 前向：z1 = x@W1+b1 → a1 = relu(z1) → z2 = a1@W2+b2 → p = softmax(z2)。每一层都是「线性变换 + 非线性」。
- Softmax 的防溢出技巧：先减去最大值再取指数，数学上等价（分子分母同除一个常数），数值上避免 exp 溢出。
- 反向第 1 步：dz2 = p − onehot(y)，交叉熵与 Softmax 的导数在这里完美约简。

- 反向第 2 步：dW2 = 外积(a1, dz2)，因为 z2 = a1@W2，对 W2 的梯度就是输入与上游梯度的外积。
- 反向第 3 步：dz1 = (W2@dz2) * relu_grad(z1)——先用 W2 把梯度传回上一层，再乘激活函数(决定神经元是否被点亮、输出多大的非线性函数)的导数，这就是链式法则。
- 为什么必须有非线性：没有 ReLU，多层线性变换可以合并成一个 W，网络退化为单层线性模型，深度毫无意义。

维度	说明
解决的问题	输入—输出映射高度非线性、特征难以手工设计的问题
核心原理	多层「线性 + 非线性」复合；用反向传播 + 梯度下降最小化损失
时间复杂度	训练 O(epoch · n · 参数量)；推理 O(参数量) 次乘加
空间复杂度	参数占用 + 激活值缓存（训练时激活是大头）
优点	表达能力强、端到端、无需手工特征、可扩展到超大规模
缺点	黑箱、需大量数据、调参敏感、训练不稳定（梯度消失/爆炸）
典型应用	几乎所有感知类任务的底座；也是 LLM 中 FFN 层的基本形态
关键参数 / 调参要点	层数与宽度、激活函数（ReLU/GELU/SiLU）、学习率与调度、批大小、初始化、正则

6.2　优化器：从 SGD 到 AdamW

优化器	更新规则要点	适用场景	关键参数
SGD	$\theta \leftarrow \theta - \eta \cdot g$	理论分析、需要强泛化时（CV 常用）	η、动量 0.9、Nesterov
SGD + Momentum	累积梯度方向，抑制震荡	CNN 训练的常青选择	momentum=0.9
AdaGrad	按参数累积平方梯度自适应步长	稀疏特征（早期 NLP）	初始累加值
RMSProp	用指数移动平均替代累积	非平稳目标	α=0.99
Adam	动量 + 自适应二阶估计	默认首选，收敛快	lr=1e-3、β=(0.9,0.999)
AdamW	Adam + 解耦权重衰减	Transformer / LLM 训练标配	lr、weight_decay=0.01—0.1、warmup

> **LLM 训练的默认配方**　AdamW + 余弦学习率衰减 + warmup（前 1%—2% 步数线性升温）+ 梯度裁剪（clip=1.0）+ bf16 混合精度。这套组合几乎是所有开源大模型预训练的默认设置。

配套仿真：搜索「优化器对照」，可以同时看到 SGD、动量和 Adam 在同一条损失曲面上走出来的三条路径：SGD 在窄谷里来回横跳，动量冲得更远、更快，Adam 前期步子大、后期自动收小。把学习率和动量系数改一改再跑一次，这两个参数各自管什么就一目了然了。

6.3 正则化与归一化

技术	做法	解决什么
Dropout	训练时按概率 p 随机置零部分神经元	共适应（co-adaptation），等价于模型集成
BatchNorm	对 batch 维度归一化 + 可学习仿射	内部协变量偏移；但对 batch size 敏感，小 batch 失效
LayerNorm	对特征维度归一化（与 batch 无关）	Transformer/LLM 的标准配置，序列长度可变时稳定
RMSNorm	只用均方根归一化，去掉减均值与偏置	LLaMA 等模型采用，更快且效果相当
残差连接(把输入直接加到输出上，给梯度留一条直通路)	y = f(x) + x	梯度直通，解决深度网络退化，是千层网络可训练的前提
数据增强	对输入做保持标签的变换	低成本扩充数据，CV 必用；NLP 常用回译/EDA

6.4 卷积神经网络 CNN

定位：为图像设计的结构，用「局部感受野(一个神经元能看到输入的多大范围) + 权值共享 + 平移等变性」把参数量从全连接的 O(H·W·C·K) 压到 O(k^2·C·K)。

代码 6-2　一个最小 CNN（PyTorch）与形状计算公式

```
import torch.nn as nn

class SmallCNN(nn.Module):
    def __init__(self, num_cls=10):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1), nn.BatchNorm2d(32), nn.ReLU(),
            nn.MaxPool2d(2),                        # 32×32 → 16×16
            nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.ReLU(),
            nn.MaxPool2d(2),                        # 16×16 → 8×8
        )
        self.head = nn.Sequential(nn.AdaptiveAvgPool2d(1), nn.Flatten(),
                                  nn.Linear(64, num_cls))
```

```
    def forward(self, x):
        return self.head(self.features(x))
# 参数量计算：Conv(k×k, Cin→Cout) = k*k*Cin*Cout + Cout
# 输出尺寸：out = floor((in + 2*pad - dilation*(k-1) - 1)/stride + 1)
```

参数 / 符号	含义	取值与影响
nn.Conv2d(3,32,3,padding=1)	卷积(用一个小窗口在输入上滑动，提取局部特征)层：输入 3 通道、输出 32 通道、3×3 核	参数量 = $k^2 \cdot C_{in} \cdot C_{out} + C_{out}$ = 3×3×3×32+32 = 896
padding=1	在输入四周补 1 圈 0	配 3×3 核时保持空间尺寸不变；padding=0 则每卷积一次尺寸减 2
MaxPool2d(2)	2×2 最大池化(缩小尺寸、只保留主要信息)	尺寸减半、通道数不变；带来一定的平移不变性并降低计算量
BatchNorm2d(32)	对通道维做批归一化(对每批数据做归一化，让训练更快更稳)	加速收敛、允许更大学习率；但依赖 batch size，小 batch 时改用 GroupNorm
AdaptiveAvgPool2d(1)	把任意尺寸的特征图池化成 1×1	让网络能接受不同输入分辨率，避免写死全连接层维度
num_cls	分类类别数	决定最后 Linear 的输出维度和交叉熵的类别数

代码逐段解读：

- 结构顺序：Conv → BN → ReLU → Pool 是现代 CNN 的标准块；早期 AlexNet 用的是 Conv → ReLU → Pool → LRN。
- 形状公式：out = floor((in + 2·pad − dilation·(k−1) − 1)/stride + 1)，算不清维度是 CNN 最常见的报错来源。
- 参数量集中在全连接层，计算量集中在卷积层——这是 CNN 设计的基本常识，也是用 GAP 取代 FC 的动机。
- 感受野：两层 3×3 卷积等价于一层 5×5 的感受野，但参数更少、非线性更多，这就是为什么深层小核优于浅层大核。
- 工程建议：除非有海量数据，否则优先加载 ImageNet 预训练权重做迁移学习，比从零训练快且准得多。

维度	说明
解决的问题	图像/语音/网格数据的特征提取与识别
核心原理	卷积核在局部滑动并共享权重，逐层扩大感受野，从边缘到语义
时间复杂度	$O(H \cdot W \cdot C_{in} \cdot C_{out} \cdot k^2)$，可用 im2col + GEMM 或 Winograd 加速

维度	说明
空间复杂度	参数 O(k²·C_in·C_out)；激活值在训练时是显存大头
优点	参数效率高、平移不变、可迁移（ImageNet 预训练）
缺点	归纳偏置强但局限于网格数据；对长程依赖建模弱（需很大感受野）
典型应用	图像分类/检测/分割、医学影像、OCR、语音、视频理解；ViT 出现后部分被替代
关键参数 / 调参要点	kernel_size、stride、padding、通道数、深度（ResNet 残差块）、是否用预训练权重

6.5 RNN / LSTM / GRU

定位：深度学习统治序列建模的二十年。核心是「隐藏状态逐步传递」，但串行结构限制了并行与长度。

代码 6-3　LSTM 单元门控计算

```
import torch, torch.nn as nn

# 手写 LSTM 单元的核心门控（前向一步）
def lstm_step(x_t, h_prev, c_prev, W, b):
    gates = x_t @ W.T + b                       # 一次性算出 4 个门
    i, f, g, o = gates.chunk(4, dim=-1)         # 输入门/遗忘门/候选/输出门
    i, f, o = torch.sigmoid(i), torch.sigmoid(f), torch.sigmoid(o)
    g = torch.tanh(g)
    c = f * c_prev + i * g                      # 细胞状态：遗忘旧信息 + 写入新信息
    h = o * torch.tanh(c)
    return h, c
# 关键：c 的梯度路径是 c_t = f·c_{t-1} + ...，遗忘门接近 1 时梯度几乎无损回传，
# 这正是 LSTM 缓解梯度消失、RNN 却做不到的原因。
```

参数 / 符号	含义	取值与影响
x_t	当前时间步的输入	形状 (batch, input_size)
h_prev / c_prev	上一时刻的隐藏状态与细胞状态	h 是对外输出，c 是内部记忆载体，两者分开是 LSTM 的关键设计
i / f / o	输入门 / 遗忘门 / 输出门	都由 sigmoid 产出，取值 (0,1)，起到「阀门」作用
g	候选细胞状态（tanh 激活）	取值范围 (−1,1)，是本步真正想写入的新信息
c = f*c_prev + i*g	细胞状态更新：遗忘旧信息 + 写入新信息	这条路径是 LSTM 记忆能长期保留的物理基础
h = o*tanh(c)	输出隐藏状态	输出门决定把多少记忆暴露给外部

参数 / 符号	含义	取值与影响
chunk(4)	把一次矩阵乘法的结果切成 4 份	四个门共用一次大矩阵乘法，减少 kernel 调用、提升 GPU 效率

代码逐段解读：

- 为什么四个门一起算：i、f、o、g 都是对同一输入做不同的线性变换，拼成一个大矩阵一次算完，是标准实现的性能优化。
- 梯度直通的关键：c_t = f·c_{t−1} + ...，当遗忘门 f 接近 1 时，梯度从 c_t 回传到 c_{t−1} 几乎不衰减。
- 普通 RNN 为什么不行：h_t = tanh(W·h_{t−1} + U·x_t)，梯度每回传一步就要乘一次 W 和 tanh′，连乘 T 次后趋近 0，即梯度消失。
- LSTM 的代价：虽然缓解了梯度消失，但时间步仍然必须串行计算，无法像 Transformer 那样并行——这是它被取代的根本原因。
- GRU 是简化版：把遗忘门与输入门合并成一个更新门，参数少 1/4、速度更快，效果通常相当。

维度	说明
解决的问题	变长序列的建模：语言模型、翻译、语音识别、时间序列
核心原理	用隐藏状态传递历史；LSTM 用遗忘/输入/输出三门控制信息流
时间复杂度	$O(T \cdot d^2)$，但时间步必须串行 → 无法并行，是最大瓶颈
空间复杂度	O(T·d) 激活（训练时需存全部时间步）
优点	天然处理变长序列、参数沿时间共享、LSTM 长程记忆能力强
缺点	串行训练慢、长序列仍会遗忘、难以扩展到超大规模
典型应用	2017 年前的 NLP 主力、时间序列预测、语音识别；现多被 Transformer 取代
关键参数 / 调参要点	hidden_size、层数、dropout、双向与否、梯度裁剪阈值

为什么被 Transformer 取代　RNN 的第 t 步依赖第 t−1 步，GPU 的并行能力完全用不上；而且信息要走过 T 步才能从句首传到句尾。Transformer 用一次自注意力(序列内部各位置互相计算注意力)把任意两位置的距离压成 O(1)，训练可完全并行——这是大模型能规模化的前提。

6.6　生成模型：VAE、GAN、扩散模型

模型	核心思想	训练目标	优点	缺点	关键参数
VAE	编码为分布 + 重参数化采样，再解码重建	ELBO = 重建损失 + KL 散度	训练稳定、有隐空间结构	生成样本偏模糊	β（KL 权重）、隐维度

模型	核心思想	训练目标	优点	缺点	关键参数
GAN	生成器与判别器对抗博弈	min_G max_D E[log D(x)] + E[log(1−D(G(z)))]	样本锐利、采样快（一次前向）	训练不稳定、模式崩溃	学习率比、谱归一化、R1 正则、WGAN 的 clip
扩散模型	逐步加噪 → 学习反向去噪	预测噪声 ε 的均方误差	训练稳定、质量与多样性最好	采样需几十至几百步，慢	步数 T、噪声调度（cosine）、CFG scale、采样器（DDIM/DPM-Solver）

扩散模型的前向过程是 q(x_t | x_{t−1}) = N(√(1−β_t)·x_{t−1}, β_t·I)，训练时网络学习预测所加噪声 ε_θ(x_t, t)；采样时从纯高斯噪声出发逐步去噪。条件生成（文生图）通过把文本编码（CLIP/T5）以交叉注意力注入实现，CFG（Classifier-Free Guidance）用 scale 控制贴合提示词(你输入给模型的指令或问题)的程度（常用 5—9）。

> **与 LLM 的交汇** 今天的图像/视频生成主流是「扩散 + DiT（Diffusion Transformer）」，而 LLM 用的是自回归(一个接一个地生成，每一步都依赖前面已生成的内容)。两者正在融合：自回归做语义规划，扩散做连续信号生成。

6.7 本章常见错误与练习

深度学习部分的错误，往往会让训练悄悄失败而不报错。

一是忘了把梯度清零。PyTorch 的梯度默认累加，忘记 optimizer.zero_grad() 就等于把历史梯度一起用上，损失曲线会莫名其妙地爆炸。

二是学习率设置得太大，损失开始震荡；太小又几乎不动。先做学习率扫描，再用余弦退火或阶梯衰减，是最稳妥的起点。

三是数据没有归一化。输入尺度差十倍，梯度尺度就差十倍，训练会非常慢甚至发散。

四是把模型切到 eval() 后忘了切回来，或者推理时忘了 torch.no_grad()。前者会让 BatchNorm 和 Dropout 行为错误，后者会让显存被计算图吃掉。

五是训练集和验证集的预处理不一致。验证时的归一化参数必须来自训练集，否则指标不可信。

六是用准确率监控不平衡任务，或者在不平衡任务上过度关注损失绝对值。分类任务要看 F1 与 AUC，生成任务要看样本质量。

七是只调参不看数据。很多「调不动」的问题，其实是标签错了、重复样本、或者训练集与线上分布不同。

三道练习：第一题，手写一个两层全连接网络并用它拟合 y = sin(x)，观察不同学习率下的损失曲线。第二题，在同一个任务上分别去掉归一化、去掉 Dropout，比较验证集表现。第三题，用梯度检查（数值梯度与反向传播的梯度对比）验证一次自己的反向传播实现是否正确。

6.8　激活函数：没有它，深度学习就只是一个线性回归

如果每层都只做「y = Wx + b」，那么两层叠起来还是一个矩阵乘（W_2W_1x），一百层也一样。层数再多也没用，因为它们乘起来还是一个矩阵。激活函数就是在每层后面加一个「折弯」，让网络真正拥有拟合均匀曲线的能力。

表 6-4　四个常见激活函数

名字	公式	优点	坑
Sigmoid	σ(x) = 1 / (1 + e^−x)	输出在 0–1，能当概率	极易饱和（梯度消失）
Tanh	(e^x − e^−x) / (e^x + e^−x)	输出关于 0 对称	一样会饱和
ReLU	max(0, x)	计算快、不饱和，现代默认	负区梯度为 0（死循环）
GELU	x·Φ(x)（标准正态累计分布）	平滑，Transformer 里常用	计算略贵

> **实用结论**　你自己搭网络时，隐层直接用 ReLU（或 GELU），不要想太多。只有输出层才需要根据任务选：二分类用 Sigmoid，多分类用 Softmax，回归什么都不用。

6.9　手写反向传播：两层网络的一次完整迭代

反向传播是深度学习最容易被「跳过去」的一节：教材往往写几个偏导符号就说「链式法则可得」，读者只能点头。这一节不用公式吓人，而是从一个具体的小网络出发，用数字把每一步算出来。

打个比方：反向传播像公司里的问责流程。最终产品出了问题（损失），先问最后一层「你责任多大」，再按贡献大小把责任一层层往前传。每层收到的「责任」正比于它对上一层的贡献——这就是链式法则的直观含义。

6.9.1　搭一个最小的网络

输入 2 个数，中间 2 个隐藏神经元，输出 1 个数。激活函数用 ReLU（大于 0 就照原样，小于 0 就归零），损失用平方误差。数据只有一条：输入 (1, 2)，目标输出 1。

```
import numpy as np
```

> **为什么只有 8 个数**　因为整个网络小到可以手算。把参数全部写出来，你才能真正看清「训练」到底在调什么。

这个网络只有两层：输入 2 个数，中间 2 个神经元，输出 1 个数。输入到隐藏层是一个 2×2 的矩阵加 2 个偏置（共 6 个数），隐藏层到输出是一个 2×1 的矩阵加 1 个偏置，再加一个输出偏置，一共 8 个可调参数。

把这 8 个数抄在纸上，整个网络就彻底透明了。所谓训练，就是反复调整这 8 个数，让输出越来越接近正确答案；所谓深度学习，就是把 8 变成八十亿，但调整的方式一个字都没变。

- W1、b1 负责「输入 → 隐藏层」这一步的线性变换。
- W2、b2 负责「隐藏层 → 输出」这一步。
- 权重决定每个输入的影响有多大，偏置决定这个神经元的「门槛」高低。

> **读代码看这里**　先把这 8 个数的形状对一遍：W1 是 2×2，W2 是 2×1。形状对上了，后面算矩阵乘法就不会错。

```
x  = np.array([1.0, 2.0])
W1 = np.array([[0.1, 0.2], [0.3, 0.4]])   # 输入层 -> 隐藏层 (2x2)
b1 = np.array([0.0, 0.0])
W2 = np.array([[0.5], [0.6]])             # 隐藏层 -> 输出 (2x1)
b2 = np.array([0.0])
y  = np.array([1.0])
```

代码 6-30　一个两层网络的全部参数（只有 8 个数）

6.9.2　前向：把输入一步步变成预测

第一层先算加权和 z1 = x·W1 + b1，再过 ReLU 得到 a1。第二层把 a1 加权求和得到预测。我们逐个数算出来。

表 6-8　一次前向传播的完整计算

位置	计算式	中间值	结果
隐藏层加权和 z1	x·W1 + b1	[1×0.1+2×0.3, 1×0.2+2×0.4]	[0.7, 1.0]
隐藏层激活 a1	ReLU(z1)	两个都大于 0，照原样	[0.7, 1.0]
输出层加权和 z2	a1·W2 + b2	0.7×0.5 + 1.0×0.6	0.95
损失 L	(z2 - y)²	(0.95 - 1)²	0.0025

预测 0.95，目标 1，损失只有 0.0025。网络已经快对了，但我们要看的不是结果，而是「该往哪调」。

6.9.3　反向：从损失倒推每一层的责任

第一步先算损失对输出 z2 的导数：2(z2 - y) = 2×(0.95-1) = -0.1。这个数就是输出层的「责任」，也顺手告诉我们：z2 每增加 1，损失减少 0.1，所以应该让 z2 变大。

接着往前传。W2 的梯度等于 a1 乘以这个责任：a1 = [0.7, 1.0]，所以 W2 的两个梯度是 [-0.07, -0.1]。含义很直白——谁在这一步出力大（a1 里的 1.0 比 0.7 大），谁的责任就大。

再往前，要算 W1 的梯度，就得知道隐藏层各自分到多少责任。责任通过 W2 往回分：第一个隐藏神经元分到 -0.1×0.5 = -0.05，第二个分到 -0.1×0.6 = -0.06。然后再乘上 ReLU 的导数——因为两个 z1 都大于 0，ReLU 在这里的导数是 1，责任原样通过。

表 6-9　反向传播的四个梯度（学习率 0.5）

参数	梯度	现在的值	更新后
W2[0]	-0.1 × 0.7 = -0.07	0.5	0.535
W2[1]	-0.1 × 1.0 = -0.10	0.6	0.650
W1[0][0]（对 a1[0] 的责任 -0.05）	-0.05 × 1 = -0.05	0.1	0.125
W1[1][0]（对 a1[0] 的责任 -0.05）	-0.05 × 2 = -0.10	0.3	0.350

更新规则全部一样：新值 = 旧值 - 学习率 × 梯度。学习率取 0.5，所以 W2[1] 从 0.6 变成 0.6 - 0.5×(-0.1) = 0.65。梯度是负的，参数就变大——因为我们需要输出更大一点。

> **为什么梯度是负的**　梯度为负，说明损失随参数增大而减小，所以应该增大参数；梯度为正则相反。更新公式永远是「减去梯度」，方向自然就对。

6.9.4　把零件拼成框架

真实的框架（PyTorch、TensorFlow）做的事情和上面完全一样，只是把「记录每个运算」和「自动求导」做成了引擎。你写 forward，它帮你算 backward。理解了上面这八个数怎么传，再回头看框架的代码就不会觉得是黑箱了。

> **一个必须记住的细节**　训练循环里每一步都要先清零梯度（zero_grad）。因为框架默认会把梯度累加，忘了清零就会把好几步的梯度加在一起，训练直接跑偏。这是新手最常见的错误，没有之一。

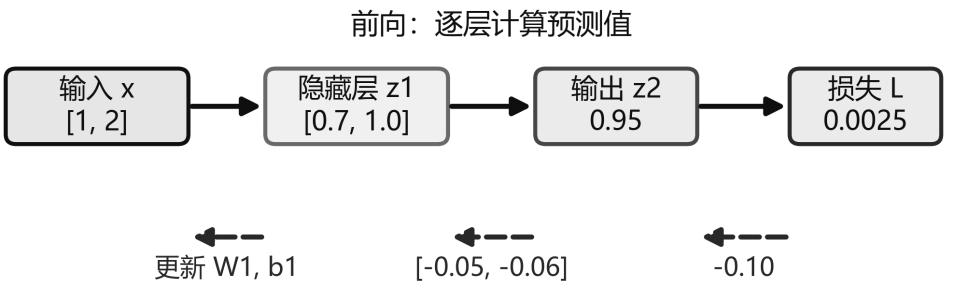


图 6-2　反向传播：把「责任」从损失一层层往回传，谁出力大谁责任大

6.10　手写一个卷积层：卷积到底在做什么

卷积神经网络的「卷积」听起来高深，其实就是一个小窗口在图上滑动，每滑到一个位置就把窗口里的数和权重逐项相乘再相加。这一节用 5×5 的图和一个 3×3 的卷积核，把这个动作完整做一遍。

打个比方：卷积核就像一个模板或者印章。你在图上到处盖这个印章，盖出来的深浅就代表图上有多少地方长得像这个模板。找边缘的核、找角的核、找纹理的核，都是同一种操作，只是印章的图案不同。

6.10.1　一次卷积的完整计算

输入（5×5）和卷积核（3×3）如下。为了容易看出效果，输入里放了一条竖直的亮边。

```
img = np.array([
    [1, 1, 0, 0, 0],
    [1, 1, 0, 0, 0],
    [1, 1, 0, 0, 0],
    [1, 1, 0, 0, 0],
    [1, 1, 0, 0, 0]], dtype=float)
```

> **它在演示什么**　卷积核到底在干什么——用一个小窗口在图上滑动，每一步做一次加权求和。

先看那张 5×5 的图：左边亮、右边暗，中间有一条竖直的边界。再看那个 3×3 的核：左列是 -1、中列是 0、右列是 1。这个核专门用来找竖直边。

它怎么做到的？当窗口落在「左边暗、右边亮」的位置时，右侧的像素乘 1、左侧的乘 -1，结果是一个很大的正数；窗口落在纯色区域时，正负相抵，结果接近 0。所以输出的亮暗就直接标出了「哪里有竖直边」。

- 第一步：把 3×3 的核对准图像的某个位置。

- 第二步：把窗口里 9 个像素和核里 9 个数字一一相乘，再全部加起来，得到输出的一个像素。
- 第三步：把窗口往右移一格，重复上面的过程，直到扫完整张图。

CNN 里那些让人眼花缭乱的卷积核，本质上就是这个「找竖直边」的核的复杂版本——只不过它们的 9 个数字不是人写的，而是训练出来的。

> **读代码看这里**　把核里的 -1、0、1 和图片的明暗对一遍，你就能看懂输出的每一个数字是从哪来的。

```
kernel = np.array([
    [-1, 1, 0],
    [-1, 1, 0],
    [-1, 1, 0]], dtype=float)   # 一个「找竖直边」的核
```

代码 6-31　5×5 的图与 3×3 的边缘检测核

卷积核在图上从左到右、从上到下移动。每滑到一个位置，就把窗口里 9 个像素与核里 9 个权重逐项相乘求和。左上角那一次：1×(-1) + 1×1 + 0×0 + 1×(-1) + 1×1 + 0×0 + 1×(-1) + 1×1 + 0×0 = 0。

表 6-10　卷积核滑过九个位置得到的输出（3×3）

窗口位置	窗口里左列与中列的取值	输出值	含义
左上（第 0–2 列）	窗口左列与中列都是 1	0	核里的 -1 与 +1 正好抵消，说明这里没有边
中上（第 1–3 列）	左列是 1，中列是 0	-3	这里正是那条竖直边
右上（第 2–4 列）	两列都是 0	0	平坦区域
左下（第 0–2 列，下三行）	与左上相同	0	边是竖直的，上下一样
右下（第 2–4 列，下三行）	两列都是 0	0	平坦

输出的 3×3 矩阵里，中间那一列全是 -3，也就是响应最强的一列，正好对应输入中竖直边的位置。注意符号是负的——这个核衡量的是「左边比中间暗多少」，而这条边是「左亮中暗」，所以取到了最小值。这就是卷积的本质：用一个滑动的小模板，把「局部形状」变成一张新的「响应图」。

6.10.2　三个关键概念：步长、填充、感受野

表 6-11　卷积的三个超参数

概念	含义	常见取值	影响

概念	含义	常见取值	影响
步长 stride	每次滑动几格	1 或 2	步长为 2 会让输出尺寸减半，相当于降采样
填充 padding	在边缘补几圈 0	0 或 1	补 1 圈可以让输入输出尺寸相同
通道 channels	同时用多少个卷积核	32、64、128…	每个核学一种模式，通道数就是「看多少种模式」

> **输出尺寸公式**　输出边长 = (输入边长 + 2×填充 - 核边长) ÷ 步长 + 1（向下取整）。例如输入 5、核 3、填充 1、步长 1，得到 (5+2-3)/1+1 = 5，尺寸不变。

6.10.3　为什么要「深」：堆叠出来的层次

一层卷积只能看到 3×3 的范围。但如果堆两层，第二层的每个位置就间接看到了 5×5；堆三层就是 7×7。这就是感受野随深度增长。也正因为如此，浅层学到的是边缘和颜色，深层学到的是眼睛、轮子这样的部件，最后才是整只猫。

配合池化（把 2×2 的小块缩成一个数），网络在变深的同时还能控制计算量。这就是 CNN 能在图像上打败全连接网络的根本原因：它把「图像是局部的、平移不变的」这个先验知识，直接写进了结构里。

6.11　训练不收敛怎么办：一份可以照着查的清单

深度学习最让人沮丧的体验是：代码没报错，损失却一直在乱跳或者干脆不动。这一节给一份按顺序排查的清单，从最常见的原因往下走。

6.11.1　第一步：先确认「能过拟合一个小样本」

这是最有价值的一步，没有之一。取 10 到 20 条数据，把模型调大到能记住它们，看训练损失能不能降到接近 0。如果能，说明模型和数据管道没问题，问题出在泛化上；如果不能，说明连「记住」都做不到，应该先修代码或结构。

> **为什么这招有效**　因为「记住十几条数据」是模型能力的最低门槛。连这个都做不到，谈泛化是没有意义的。实践中大约一半的「训练失败」都能在这一步定位到问题。

6.11.2　第二步：按顺序检查这六项

表 6-12　训练不收敛的排查清单（按优先级排列）

检查项	典型症状	怎么修
标签与输入是否对齐	损失降一小段就卡住	打印几条样本，人眼看输入和标签是否配对

检查项	典型症状	怎么修
学习率是否合适	损失爆炸或几乎不动	先用 1e-3 试，再上下各试一个量级
是否忘了清零梯度	损失剧烈波动	每次反向传播前调用 zero_grad
数据是否归一化	深层网络完全不收敛	输入做标准化，输出若在 0–1 用 Sigmoid
损失函数是否匹配任务	预测值范围明显不对	分类用交叉熵，回归用 MSE，别混用
初始权重是否会爆炸	一开始损失就是 nan	用默认初始化，别自己随便填大数

6.11.3　第三步：看曲线的形状，判断病因

损失曲线的形状比最终数值更有信息量。这一节给出四种典型形状和它们对应的病因。

表 6-13　四种损失曲线形状与诊断

曲线形状	可能的病因	处理方向
一直在高位平着不动	学习率太小、梯度消失、输入没归一化	调大学习率、换激活函数、做标准化
上下剧烈跳动	学习率太大、批太小、忘了清零梯度	降学习率、增大 batch size
先降后升	过拟合、学习率后期偏大	加正则、早停、学习率衰减
出现 nan	梯度爆炸、除以零、标签超范围	梯度裁剪、检查数据里的异常值

最后一句经验之谈：调参的时候一次只改一个东西。同时改三个超参数，即使模型变好了，你也不知道是哪个起了作用；变差了，更不知道是哪个把结果拖下去的。把实验记录下来，是深度学习里最朴实也最有效的方法。

6.12　从零写一个能训练的神经网络：60 行纯 Python

第 6.9 节我们手推了一次迭代。这一节把那个推演写成完整程序：一个两层网络，一层隐藏层、一层输出，全部用纯 Python 实现，不依赖任何框架。跑通它，你对「训练」这两个字的理解会和只看框架代码完全不一样。

为了让代码自包含、不需要下载数据集，我们用一个二维的小数据集代替手写数字：两类点，一类在左下、一类在右上。网络的输入是 2 维、隐藏层 4 个神经元、输出 1 个概率值。等这版跑通，把输入从 2 维换成图片的 784 维，就是完整的手写数字识别流程。

代码 6-32　两层网络：前向传播与反向传播

```
import math, random
def sigmoid(z):
    return 1.0 / (1.0 + math.exp(-z))
def forward(x, W1, b1, W2, b2):
    a1 = [sigmoid(sum(W1[j][i] * x[i] for i in range(len(x))) + b1[j])
          for j in range(len(b1))]
    z2 = sum(W2[0][j] * a1[j] for j in range(len(a1))) + b2[0]
    return a1, sigmoid(z2)
def backward(x, y, a1, a2, W1, b1, W2, b2, lr):
    dz2 = a2 - y                        # sigmoid + 交叉熵的梯度，只有这一项
    dW2 = [dz2 * a1[j] for j in range(len(a1))]
    db2 = dz2
    dz1 = [W2[0][j] * dz2 * a1[j] * (1 - a1[j]) for j in range(len(a1))]
    for j in range(len(a1)):
        for i in range(len(x)):
            W1[j][i] -= lr * dz1[j] * x[i]
        b1[j] -= lr * dz1[j]
    for j in range(len(a1)):
        W2[0][j] -= lr * dW2[j]
    b2[0] -= lr * db2
    return -(y * math.log(a2 + 1e-9) + (1 - y) * math.log(1 - a2 + 1e-9))
```

> **这 20 行是什么**　一个能跑通的最小训练流程：造数据、搭网络、前向、算损失、反向、更新。

所有深度学习代码，无论多复杂，骨架都是这五步，区别只在于网络的层数和数据的规模。把这段吃透，再去看 PyTorch 的项目代码，你会发现只是把每个手工步骤换成了一个函数调用。

- 造数据：这里用异或问题的四个样本，因为它不能用一条直线分开，正好用来验证网络真的学到了非线性。
- 前向：输入经过两层变换得到预测值。
- 算损失：比较预测值和真实标签，得到一个「错得多离谱」的数字。
- 反向 + 更新：把误差往回传，算出每个参数该往哪个方向调，然后调一点点。

训练日志里那两行最关键：一开始准确率只有一半（等于瞎猜），几百轮之后变成全对——这就是模型从「什么都没学到」到「学会」的完整过程。

> **读代码看这里**　重点是那几行更新参数的公式，它和 5.19 手推的线性回归梯度下降是同一个东西，只是参数变多了。

代码 6-33　造数据 + 训练循环：20 行看完整流程

```
random.seed(1)
data = [([0.15, 0.25], 0.0), ([0.30, 0.10], 0.0), ([0.20, 0.35], 0.0),
        ([0.80, 0.75], 1.0), ([0.70, 0.90], 1.0), ([0.90, 0.65], 1.0)]
W1 = [[random.uniform(-0.5, 0.5) for _ in range(2)] for _ in range(4)]
b1 = [0.0] * 4
```

```
W2 = [[random.uniform(-0.5, 0.5) for _ in range(4)]]
b2 = [0.0]
for epoch in range(1, 2001):
    loss = 0.0
    for x, y in data:
        a1, a2 = forward(x, W1, b1, W2, b2)
        loss += backward(x, y, a1, a2, W1, b1, W2, b2, lr=0.5)
    if epoch in (1, 10, 100, 500, 1000, 2000):
        acc = sum(1 for x, y in data if (forward(x, W1, b1, W2, b2)[1] > 0.5) ==
(y > 0.5))
        print("epoch %4d  loss %.3f  准确率 %d/6" % (epoch, loss / len(data), acc))
```

这段代码的输出大致是这样（随机种子固定，你可以复现）：

表 6-14　训练日志：损失怎么一步步降下去

轮次	平均损失	准确率	这时网络学到了什么
第 1 轮	0.736	3/6	权重还是随机的，基本在瞎猜
第 10 轮	0.716	3/6	边界还没推开，损失几乎不动
第 100 轮	0.011	6/6	突然就分对了，损失断崖式下降
第 500 轮	0.001	6/6	边界继续外移，置信度提高
第 2000 轮	0.000	6/6	基本收敛，再训也不会更好

请注意两个细节。第一，准确率在第 100 轮就到了 6/6，但损失还在继续下降——说明「分对了」和「分得有信心」是两件事，这也是后面讲模型校准时会用到的区别。第二，前 10 轮损失几乎不动、第 100 轮突然断崖式下降，这是 sigmoid 网络很典型的行为：一开始梯度很小，等权重挪到某个位置后误差信号才传得出去。把这个网络加深到十层，这种「前期不动」会被放大到几乎完全不动，原因就是下一节要讲的梯度消失。

> **为什么值得亲手写一遍**　框架把 backward 藏起来了，所以很多人调了半年模型也不知道梯度到底是怎么传的。写过一次纯 Python 版本，以后再看到「梯度爆炸」「梯度截断」这些词，你会立刻知道它在说哪一行代码。

6.13　梯度消失与梯度爆炸：为什么深网络难训练

把上面的网络加深到十层，会发生一件奇怪的事：靠近输入的那几层几乎学不到东西，权重几乎不动。这不是代码写错了，而是梯度在往回传的路上「衰减」掉了。

回忆一下 6.9 节的手推：反向传播是一层一层往回乘的。每往回过一层，梯度都要乘上「这一层的激活函数导数」和「这一层的权重」。如果这两个数的乘积长期小于 1，那

么十层之后梯度就变成了原来的千分之一；如果长期大于 1，就会指数级放大，变成梯度爆炸。

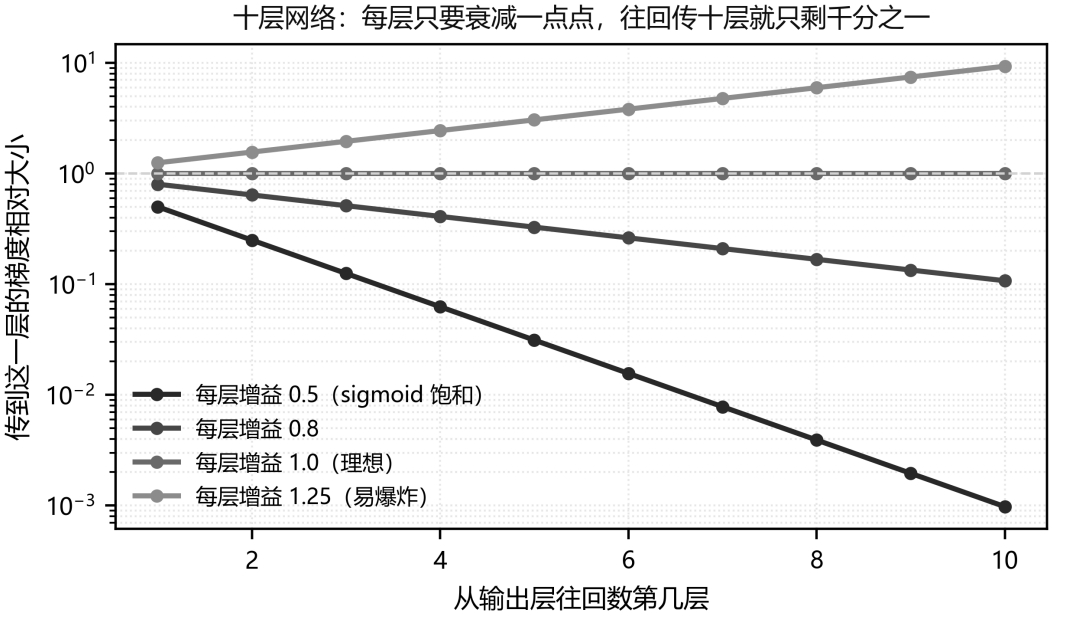


图 6-3　梯度在十层网络中往回传：每层增益只要偏离 1 一点点，十层后就是天壤之别

用 sigmoid 作为激活函数时，它的导数最大只有 0.25。也就是说，只要权重不大，每一层至少把梯度砍掉四分之三。这就是早期深层网络训不动的根本原因——不是算法错，是信号在半路衰减光了。

表 6-15　梯度消失与梯度爆炸：现象、原因与对策

现象	典型原因	怎么判断	怎么解决
深层权重几乎不变，浅层学不到东西	激活函数导数长期小于 1（sigmoid、tanh 饱和）	打印各层梯度范数，发现从后往前指数下降	换成 ReLU 系；用残差连接；用归一化
损失突然变成 NaN	梯度指数放大，一次更新把权重推到极值	看梯度范数突然涨到几千以上	梯度截断；降低学习率；检查数据是否异常
训练前期正常，中途突然崩掉	某批次出现异常样本或学习率过大	记录每个批次的损失与梯度范数	跳过异常批次；加学习率预热与衰减
浅层正常、深层不学	输出层初始化过大或过小	看各层权重初始化后输出的方差	用合适方差的初始化（He / Xavier）

下面这组数字是一次真实测量的结果：一个十层、每层 8 个神经元的 sigmoid 网络，输入是二维样本，用 Xavier 初始化。我们直接在反向传播的过程中记录每一层的梯度范数（梯度向量里所有元素平方和的平方根）。

表 6-16　十层 sigmoid 网络的实测梯度范数（一次前向 + 反向）

层号（从输入侧数）	梯度范数	相对第 10 层的比例
第 1 层	4.71×10^{-7}	约 1 / 1 800 000
第 2 层	9.51×10^{-6}	约 1 / 87000
第 3 层	3.69×10^{-5}	约 1 / 23000
第 4 层	1.01×10^{-4}	约 1 / 8200
第 5 层	4.45×10^{-4}	约 1 / 1900
第 6 层	2.54×10^{-3}	约 1 / 330
第 7 层	1.21×10^{-2}	约 1 / 69
第 8 层	5.95×10^{-2}	约 1 / 14
第 9 层	2.73×10^{-1}	约 1 / 3
第 10 层	8.32×10^{-1}	1（基准）

从第 10 层到第 1 层，梯度一路衰减了大约 180 万倍。这意味着：即使输出层已经知道自己错了，第 1 层收到的纠正信号也小到几乎可以忽略。用同样的学习率更新，输出层的权重一次能挪动 0.8 那么多，而第 1 层只能挪动 0.0000005 —— 它当然学不到东西。

> **怎么自己测**　在每一层算完梯度后打一行日志，输出该层梯度的范数。如果从后往前看是单调指数下降，就是梯度消失；如果是单调暴涨，就是梯度爆炸。这一条日志比调任何超参数都先要看。
>
> **一句话总结**　梯度消失和梯度爆炸是同一个问题的两种方向：层与层之间的「增益」没有维持在 1 附近。残差连接的作用，就是把这条增益通路强行拉回 1。

6.14　批大小、学习率、训练轮数：三个旋钮怎么拧

新手调模型时最容易犯的错，是三个旋钮一起乱拧，最后不知道是哪个起了作用。这一节把它们拆开讲，并给出一个可以照着走的调整顺序。

表 6-17　三个旋钮分别控制什么

旋钮	调大以后	调小以后	常用的起点
批大小 batch size	梯度更稳、显存占用更大、收敛所需的轮数变少	噪声更大，有时反而更容易跳出局部最优	32 或 64；显存不够就减
学习率 learning rate	下降快，但过大直接发散，损失变 NaN	很稳但极慢，容易在平台上卡住不动	0.001（Adam 系）或 0.01（SGD 系）

旋钮	调大以后	调小以后	常用的起点
训练轮数 epoch	能学得更充分，但过多会过拟合	欠拟合，模型还没学到规律就停了	先设 100，看验证集曲线再定

推荐的调整顺序是：先固定批大小和学习率，把轮数设大一点（例如 200 轮），观察验证集损失曲线。如果曲线一路下降直到变平，说明轮数设得合适；如果训练损失还在降、验证损失已经回升，说明过拟合了，该加正则或减轮数；如果曲线一开始就平着不动，说明学习率太小或者初始化有问题。

第二步再调学习率。经验做法是把学习率乘以 3 和除以 3 各试一次，比较前 10 轮的损失下降速度。学习率合适时，前 10 轮的损失应该下降得最快。

最后才调批大小。批大小的主要作用是影响显存占用和收敛稳定性，对最终效果的影响通常小于学习率。

> **最容易被忽略的一条**　调参时一次只改一个旋钮，并把每次的结果记下来。凭感觉来回改，一天下来你不会有任何可复用的结论。

第 7 章　强化学习：在试错中学决策

> **读前提示**　本章概念相对独立，但会用到「期望」和「折扣」两个概念（第 0 章 0.3 的④）。如果只想看大模型，可以先跳过本章，但第 8 章讲 RLHF(用人类偏好做强化学习)时会引用这里的 PPO 原理。

前两章的算法都在学「给定输入，输出什么」。强化学习（RL）学的是另一件事：在一系列连续的决策中，如何选动作才能拿到最多的累计奖励。它和监督学习(有标准答案可对照的学习方式)的根本差别有三点——没有标准答案（只有奖励信号）、奖励往往延迟（下棋要下完才知道输赢）、你的动作会改变你接下来看到的数据（数据非独立同分布）。

7.1　基本框架：马尔可夫决策过程 MDP

一个 MDP(马尔可夫决策过程：描述状态、动作、奖励的数学框架) 由五元组 (S, A, P, R, γ) 描述：状态集合 S、动作集合 A、状态转移概率 P(s′|s,a)、即时奖励 R(s,a)、折扣因子(未来的奖励要打多少折，越大越有远见) γ。智能体(能自己规划、调用工具、循环执行直到完成任务的程序)在时刻 t 观察状态 s_t，按策略 π(a|s) 选动作 a_t，环境给出奖励 r_t 并转移为 s_{t+1}，循环往复。目标只有一个：最大化折扣回报。

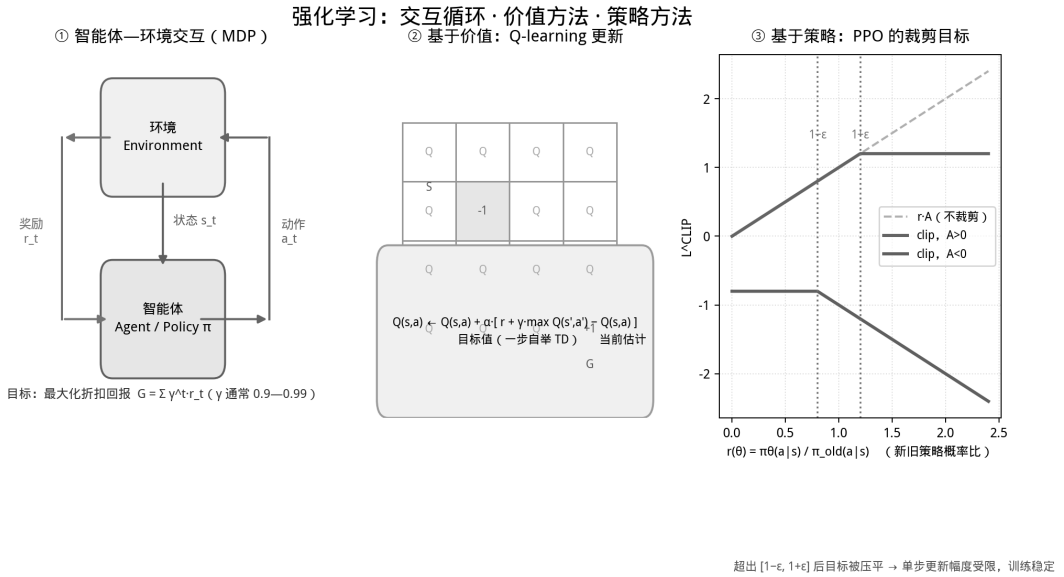
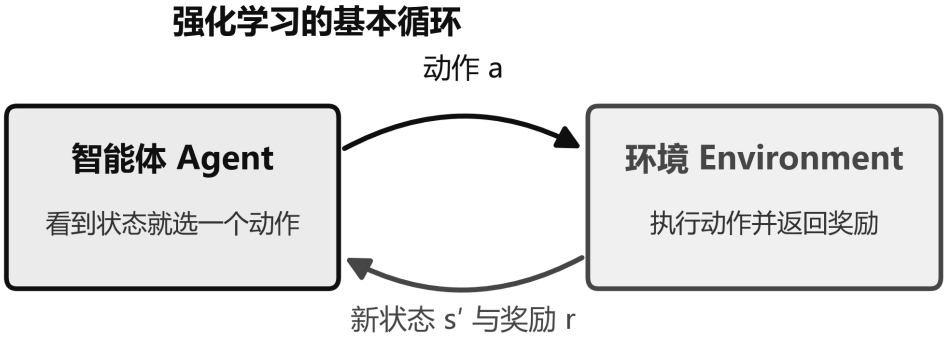


图 7-1　强化学习：智能体—环境交互循环、Q-learning 更新式与 PPO 裁剪目标

要素	符号	含义	在 LLM 场景里对应什么

要素	符号	含义	在 LLM 场景里对应什么
状态	s	当前环境的描述	已生成的上下文（prompt(提示词：你输入给模型的指令) + 已输出 token）
动作	a	智能体可选的操作	下一个要输出的 token（词表即动作空间，约 3 万—20 万维）
策略	π(a|s)	状态到动作的映射	语言模型本身：P(token | context)
奖励	r	环境给的即时反馈	奖励模型 RM 对整段回答的打分
回报	$G = \Sigma \gamma^t r$	长期累计折扣奖励	整段回答的质量总分
轨迹	τ	一串 (s,a,r) 序列	一次完整生成过程

> **为什么要有折扣因子 γ**　两个作用：一是数学上保证无限长序列的回报收敛；二是表达「近期奖励比远期更值钱」的偏好。γ 越小越短视（接近 0 时只看眼前一步），越大越有远见（接近 1 时远见但训练方差变大）。常用 0.9—0.99。



图中这个圈每转一次，就是一步；转上几百万次，就学会了一个策略。

图 7-2　强化学习的基本循环：动作出去、新状态和奖励回来

7.2　两个核心函数与贝尔曼方程

RL 几乎所有算法都在估计这两个函数之一：状态价值 V(s)（处在这个状态有多好）和动作价值 Q(s,a)（在这个状态下做这个动作有多好）。它们满足自洽的递推关系——贝尔曼方程：

代码 7-1　贝尔曼方程与策略迭代（模型已知时的动态规划解法）

```
# 贝尔曼期望方程（评估策略 π）
#   V(s)  = Σ_a π(a|s) · Σ_{s'} P(s'|s,a) · [ R(s,a) + γ·V(s') ]
```

```
#   Q(s,a)= Σ_{s'} P(s'|s,a) · [ R(s,a) + γ·Σ_{a'} π(a'|s')·Q(s',a') ]
#
# 贝尔曼最优方程（求最优策略）
#   V*(s) = max_a Σ_{s'} P(s'|s,a)·[ R + γ·V*(s') ]
#   Q*(s,a)= Σ_{s'} P(s'|s,a)·[ R + γ·max_{a'} Q*(s',a') ]

# 策略迭代的两步交替（动态规划，已知模型时可用）
def policy_iteration(P, R, gamma=0.9, theta=1e-6):
    n, m = R.shape                        # n 个状态，m 个动作
    V = np.zeros(n); pi = np.zeros(n, dtype=int)
    while True:
        # ① 策略评估：反复套用贝尔曼期望方程直到 V 收敛
        while True:
            delta = 0
            for s in range(n):
                a = pi[s]
                v = sum(P[s, a, sp] * (R[s, a] + gamma * V[sp]) for sp in range(n))
                delta = max(delta, abs(v - V[s])); V[s] = v
            if delta < theta: break
        # ② 策略改进：对每个状态贪婪地选 Q 最大的动作
        stable = True
        for s in range(n):
            q = [sum(P[s, a, sp] * (R[s, a] + gamma * V[sp]) for sp in range(n)) for
a in range(m)]
            best = int(np.argmax(q))
            if best != pi[s]: stable = False
            pi[s] = best
        if stable: return pi, V           # 策略不再变化即收敛
```

参数 / 符号	含义	取值与影响
P	状态转移概率张量 (n, m, n)	P[s,a,sp] = 在 s 执行 a 后到达 sp 的概率；现实问题中通常未知
R	奖励矩阵 (n, m)	R[s,a] = 在 s 执行 a 的即时奖励，是奖励函数的核心
gamma=0.9	折扣因子 γ	越小越短视；0.9—0.99 常用。γ→1 时远见但训练方差变大
V	状态价值数组	V[s] 表示「从 s 出发按当前策略能拿到的期望回报」
pi	当前策略，每个状态选哪个动作	策略迭代的输出，也是被不断优化的对象
theta=1e-6	策略评估的收敛阈值	两轮之间 V 的最大变化量小于它即认为已收敛
stable	策略是否不再变化	策略改进的终止标志；策略不变说明已达最优

代码逐段解读：

- 第 1 步「策略评估」：反复套用贝尔曼期望方程 $V(s)=\Sigma\pi(a|s)\Sigma P[R+\gamma V(s')]$，直到 V 收敛到当前策略的真实价值。

- 第 2 步「策略改进」：对每个状态贪婪地选 Q 最大的动作。定理保证改进后的策略不劣于原策略。
- 两步交替直到策略不再变化即收敛——这就是策略迭代，是 RL 里唯一有严格最优性保证的一类方法。
- 前提与现实：策略迭代要求已知 P 和 R（即「有模型」）。现实中环境模型未知，只能转向无模型的 Q-learning / 策略梯度(直接对策略本身求梯度，让好动作的概率变大)。
- 价值迭代是它的简化版：不等 V 收敛就立刻改进策略，把两层循环合成一层，工程上更常用。

现实问题里转移概率 P 通常未知（你不知道市场明天怎么走、不知道用户会怎么回应），于是有了「无模型（model-free）」方法：不学环境，直接从与环境的交互样本中学。这分成两大流派，也是接下来两节的主角。

7.3 基于价值的方法：Q-learning 与 DQN

Q-learning（表格型，1989）

最经典的 off-policy 时序差分（TD）算法。它不去学环境模型，而是维护一张 Q 表，每次交互后用「一步真实奖励 + 折扣后的下一状态最优估计」作为目标，把当前估计往目标拉一小步：

代码 7-2　Q-learning（约 20 行，含 ε-greedy 探索）

```
import numpy as np

class QLearning:
    def __init__(self, n_states, n_actions, lr=0.1, gamma=0.9, eps=0.1):
        self.Q = np.zeros((n_states, n_actions))
        self.lr, self.gamma, self.eps = lr, gamma, eps

    def act(self, s, rng):
        # ε-greedy：以 ε 概率随机探索，否则取当前最优
        if rng.random() < self.eps:
            return rng.integers(self.Q.shape[1])
        return int(np.argmax(self.Q[s]))

    def update(self, s, a, r, s_next, done):
        target = r + (0 if done else self.gamma * self.Q[s_next].max())  # TD 目标
        td_error = target - self.Q[s, a]                                  # TD 误差
        self.Q[s, a] += self.lr * td_error                                # 往目标拉
一小步
        return td_error
```

参数 / 符号	含义	取值与影响

参数 / 符号	含义	取值与影响
n_states / n_actions	状态数与动作数	决定 Q 表大小；状态空间稍大（如像素输入）就必须换成神经网络
Q	动作价值表 (n_states × n_actions)	Q[s,a] =「在 s 做 a 之后能拿到的期望总回报」
lr（α）	学习率，控制每步更新幅度	0.1 起步；RL 的学习率通常比监督学习小一个数量级，因为样本本身噪声大
gamma	折扣因子	0.9—0.99；决定「多看重未来」
eps	ε-greedy 的探索概率	通常从 1.0 衰减到 0.01—0.05；不衰减则永远有 ε 的概率乱走
target	TD 目标 r + γ·max Q(s′)	用一步真实奖励 + 下一状态的最优估计，这就是「自举（bootstrapping）」
td_error	时序差分误差	目标与当前估计的差，是所有 TD 类算法更新的驱动力

代码逐段解读：

- act 方法：以 ε 概率随机探索、否则取当前最优——这是最经典也最好实现的探索策略。
- update 方法：Q[s,a] += α·(target − Q[s,a])，把估计往目标方向拉一小步，不是直接赋值。
- off-policy 的关键：目标里用的是 max Q(s′,a′)，与「下一步实际做什么」无关。所以可以用任意策略甚至别人的数据来学最优 Q。
- done 的处理：终止状态没有后续，目标里必须去掉 γ·max Q 项，否则会凭空多出一份价值。
- 收敛条件：理论上在「每个状态—动作对被无限次访问 + 学习率适当衰减」时收敛到 Q*；实践中配合 ε 衰减效果很好。

为什么是 off-policy

注意更新式里的 max Q(s′,a′) 与「实际下一步会做什么动作」无关——它总是假设之后走最优动作。这意味着你可以用任意策略（甚至别人的数据）采集样本来学最优 Q，这种「学习目标策略与采样行为策略可以不同」的性质叫 off-policy，是经验回放(把经历存起来随机抽取训练，打散相关性并重复利用)能成立的前提。相对地，SARSA 用实际执行的 a′ 更新，是 on-policy，更保守但更贴合真实行为。

DQN：把 Q 表换成神经网络（2013/2015，DeepMind）

状态空间一大（比如一帧 84×84 图像有 10^数千 种状态），Q 表就存不下了。DQN 用卷积网络近似 Q(s,a;θ)，并引入两个关键技巧让它能训起来：

- 经验回放 Replay Buffer：把 (s,a,r,s′,done) 存进缓冲区，训练时随机采样小批量。这同时打破了样本的时间相关性（否则连续帧高度相似）并让数据可重复利用。

- 目标网络(一份延迟更新的参数副本，专门用来算目标值) Target Network：复制一份参数 θ^-，每 C 步才从 θ 同步一次，用它算 TD 目标。否则「目标和网络一起动」，训练会像追自己尾巴一样发散。

代码 7-3　DQN 的三大组件：经验回放、目标网络、定期同步

```
import torch, torch.nn as nn, random
from collections import deque

class DQN:
    def __init__(self, net, lr=1e-3, gamma=0.99, buf=100000, batch=64,
sync_every=1000):
        self.online, self.target = net, copy.deepcopy(net)
        for p in self.target.parameters(): p.requires_grad = False   # 目标网络不参与
梯度
        self.opt = torch.optim.Adam(self.online.parameters(), lr=lr)
        self.buf = deque(maxlen=buf)                                 # ① 经验回放
        self.gamma, self.batch, self.t, self.sync = gamma, batch, 0, sync_every

    def remember(self, s, a, r, s2, done):
        self.buf.append((s, a, r, s2, done))

    def learn(self):
        if len(self.buf) < self.batch: return None
        batch = random.sample(self.buf, self.batch)                  # 随机采样，打散
相关性
        s, a, r, s2, d = map(np.array, zip(*batch))
        s, s2 = torch.tensor(s, dtype=torch.float), torch.tensor(s2,
dtype=torch.float)
        q = self.online(s)[range(self.batch), a]                     # 当前 Q 估计
        with torch.no_grad():
            q_next = self.target(s2).max(1).values                   # ② 目标网络算目
标
            target = torch.tensor(r, dtype=torch.float) + self.gamma * q_next * (1 -
torch.tensor(d))
        loss = nn.functional.mse_loss(q, target)
        self.opt.zero_grad(); loss.backward(); self.opt.step()
        self.t += 1
        if self.t % self.sync == 0:                                  # 定期同步参数
            self.target.load_state_dict(self.online.state_dict())
        return loss.item()
```

参数 / 符号	含义	取值与影响
online / target	在线网络与目标网络	结构相同、参数不同；target 只用来算目标，不参与梯度
buf（deque, maxlen=100000）	经验回放缓冲区容量	10 万—100 万条常见；太小则样本多样性不足，太大则旧策略数据占比过高

参数 / 符号	含义	取值与影响
batch=64	每次训练采样的批量	32—256；太小梯度噪声大，太大则更新次数减少
gamma=0.99	折扣因子	Atari 等长回合任务常用 0.99；回合很短的任务可用 0.9
sync_every=1000	目标网络同步间隔	太频繁则目标不稳定，太稀疏则目标陈旧；100—10000 步都是合理范围
random.sample	从缓冲区均匀随机采样	打破样本的时间相关性，这是 DQN 能训稳的第一要素
torch.no_grad()	目标计算不回传梯度	必须加，否则梯度会穿过目标网络，训练立刻崩

代码逐段解读：

- 组件①经验回放：把 (s,a,r,s′,done) 存起来随机采样。打破连续帧的相关性，同时让一条经验被反复利用，样本效率大幅提升。
- 组件②目标网络：用 θ^- 计算 TD 目标。若用同一个网络既选动作又算目标，会出现「追逐自己尾巴」的正反馈震荡。
- 组件③定期同步：每 C 步把 online 的参数拷给 target，让目标在一段时间内保持固定。
- 损失：MSE(q, target)，即把 TD 误差做成回归损失。注意 target 要用 (1−done) 屏蔽终止状态的未来价值。
- 后续改进：Double DQN 用 online 选动作、target 估值，解决 max 操作导致的 Q 值系统性高估；Dueling 把 Q 拆成 V+A 两条支路。

DQN 家族的改进

改进	解决什么	做法
Double DQN	max 操作导致 Q 值被系统性高估	用 online 网络选动作、target 网络估值，解耦选择与评估
Dueling DQN	很多状态下动作选择并不重要	把 Q 拆成 V(s) + A(s,a) 两条支路再合并
Prioritized Replay	稀有但重要的经验被淹没	按 TD 误差大小加权采样
Rainbow	各项改进互不冲突	把上述六种技巧叠加，是 DQN 的集大成版本
Distributional RL	只学期望丢失了风险信息	直接学回报的分布（如 C51、QR-DQN）

7.4　基于策略的方法：REINFORCE → Actor-Critic → PPO

价值方法有个硬伤：动作空间必须是离散的，而且只能输出一个确定的最优动作。策略方法直接参数化策略 $\pi_\theta(a|s)$，可以输出随机策略、也能处理连续动作（机器人关节角度）。它的目标是最大化期望回报 $J(\theta) = E_\tau[\Sigma r]$，用梯度上升更新参数。

策略梯度定理与 REINFORCE

核心结论：∇J(θ) = E[∇log π_θ(a|s) · G_t]。直觉非常好懂——把这次采样的回报 G_t 当成一个权重，用它去放大或缩小「选中这个动作的对数概率」的梯度：结果好的动作以后多采，结果差的少采。

代码 7-4　REINFORCE（蒙特卡洛策略梯度，含回报归一化基线）

```
import torch, torch.nn.functional as F

def reinforce(policy, episodes, gamma=0.99, lr=1e-3):
    """policy(obs) -> 动作对数概率分布"""
    opt = torch.optim.Adam(policy.parameters(), lr=lr)
    for obs_list, act_list, rew_list in episodes:
        T = len(rew_list)
        # 1) 计算每个时刻的折扣回报 G_t（从后往前累加）
        G, returns = 0.0, []
        for r in reversed(rew_list):
            G = r + gamma * G
            returns.insert(0, G)
        returns = torch.tensor(returns, dtype=torch.float)
        returns = (returns - returns.mean()) / (returns.std() + 1e-8)   # ② 基线归一
化，降方差

        # 2) 策略梯度上升：loss 取负号，用 -G·logπ 做梯度下降
        logp = torch.stack([policy(o).log_prob(a) for o, a in zip(obs_list,
act_list)])
        loss = -(logp * returns).sum()
        opt.zero_grad(); loss.backward(); opt.step()
    return policy
```

参数 / 符号	含义	取值与影响
policy(obs)	策略网络，输出动作分布	离散动作输出类别分布，连续动作通常输出高斯分布的均值与方差
log_prob(a)	选中动作的对数概率	策略梯度的核心量：我们要调整的就是「选这个动作的概率」
returns	每个时刻的折扣回报 G_t	从后往前累加得到；蒙特卡洛方法必须等整条轨迹跑完
gamma=0.99	折扣因子	影响回报的量级，也影响方差
归一化 (G−μ)/σ	优势的标准归一化	最关键的一步：不归一化时回报尺度波动会让梯度方向忽大忽小，训练极不稳
loss = −(logp·G).sum()	负号是梯度上升转梯度下降	优化器默认最小化，所以最大化 J 要取负
lr=1e-3	学习率	RL 通常比监督学习小；PPO 常用 1e-4—3e-4

代码逐段解读：

- 第 1 步算回报：从后往前 G = r + γG，得到每个时刻「从这一刻起还能拿多少」。
- 第 2 步归一化：把回报减均值除标准差，使正负样本均衡，等价于一个粗糙的基线（baseline）。
- 第 3 步算损失：−log π(a|s)·G。直观理解——结果好的动作（G>0）把概率往上推，结果差的往下压。
- 为什么方差大：蒙特卡洛回报包含整条轨迹的随机性，且必须是 on-policy（用完即弃），样本效率很低。
- 改进方向：用 Critic 估计的 V(s) 作为基线，把 G 换成优势 A = Q − V，就得到 Actor-Critic，可单步更新、方差大降。

Actor-Critic：用价值函数当基线

REINFORCE 要等整条轨迹跑完才能更新，方差极大。Actor-Critic 引入一个 Critic 网络估计 V(s)，用优势函数(这个动作比平均水平好多少) A(s,a) = Q(s,a) − V(s) 替代原始回报：它衡量「这个动作比平均好多少」，既降了方差，又能单步更新（TD）。Actor 负责出动作，Critic 负责打分，两者交替训练。

PPO：工业界默认选择（2017，OpenAI）★

朴素策略梯度有个致命问题：一步更新太大就会把策略推崩，而且崩了就回不来（因为新采样基于新策略）。PPO 的解法极其简洁——限制新旧策略的概率比 r(θ) = π_θ(a|s)/π_old(a|s) 不要跑出 [1−ε, 1+ε]：

代码 7-5　PPO 裁剪目标（核心仅 4 行）与工程要点

```
import torch

def ppo_loss(ratio, advantage, eps=0.2, clip_vf=None):
    """ratio: π_θ(a|s)/π_old(a|s)；advantage: A(s,a)，通常已做归一化"""
    unclipped = ratio * advantage                                    # 原始目标
    clipped = torch.clamp(ratio, 1 - eps, 1 + eps) * advantage       # 裁剪后目标
    policy_loss = -torch.min(unclipped, clipped).mean()              # 取更保守（更小）
的那个
    return policy_loss

# 完整目标通常再加两项：
#   L = L^CLIP − c1·L^VF（价值函数 MSE 损失）+ c2·S[π_θ]（熵奖励，鼓励探索）
# 工程要点：
#   - 用 GAE（广义优势估计）算 advantage：A_t = Σ (γλ)^l · δ_{t+l}，λ≈0.95
#   - 同一批数据可做多个 epoch（如 4 个）的小批量更新，靠 clip 保证幅度可控
#   - advantage 必须做 batch 内归一化，否则尺度波动会让 clip 失效
```

参数 / 符号	含义	取值与影响
ratio	概率比 r = π_θ(a|s)/π_old(a|s)	>1 表示新策略更倾向这个动作；<1 表示更不倾向

参数 / 符号	含义	取值与影响
advantage	优势函数 A(s,a)	正表示这个动作好于平均，负表示差于平均；必须做 batch 内归一化
eps=0.2	裁剪范围 ε	0.1—0.3；越小更新越保守越稳，但学习越慢
torch.clamp	把 ratio 限制在 [1−ε, 1+ε]	物理含义：不允许单步更新把策略推得太远
torch.min	取裁剪前后更保守的那个	A>0 时取小（别一口气推太多），A<0 时取小（别一口气压太狠）
c1 / c2	价值损失与熵奖励的权重	c1≈0.5、c2≈0.01；熵项防止策略过早塌缩成确定性
GAE λ≈0.95	广义优势估计(一种平衡偏差与方差的优势估计方法)的衰减系数	平衡偏差与方差：λ→1 方差大偏差小，λ→0 反之

代码逐段解读：

- 核心逻辑：unclipped 是普通策略梯度目标，clipped 把 ratio 卡在 [1−ε,1+ε]。取二者较小值，等于「当更新要跑太远时，直接把梯度截断」。
- 为什么取 min：A>0 时我们希望增大该动作概率，但一旦 ratio 超过 1+ε 就不再给奖励，防止策略突变；A<0 时同理防止概率被压到 0。
- 多 epoch 复用：同一批数据可以更新多个 epoch，正因为 clip 限制了幅度，才敢这么做——这是 PPO 样本效率高于普通策略梯度的原因。
- 优势必须归一化：若 A 的量级波动，clip 的绝对边界 1±ε 就失去意义（有时几乎不裁剪、有时全被裁剪）。
- 完整目标还要加价值损失（让 Critic 更准）和熵奖励（保持探索），两者都是防止早收敛的常规手段。

> **PPO 为什么这么流行** 它在「实现简单」和「效果稳定」之间找到了最佳平衡点：核心只是几行裁剪，不需要像 TRPO 那样解带约束的优化问题，也不需要精细调 KL 系数。几乎所有主流 RLHF 框架（含大模型对齐）都默认用 PPO，原因就在这里。

7.5 探索与利用：RL 独有的难题

监督学习不存在这个问题，但 RL 必须自己决定「是继续用已知的最优动作，还是试试没试过的」。试太少会陷入局部最优，试太多则浪费大量交互。这是 RL 落地时最容易被低估的成本来源。

策略	做法	适用	注意
ε-greedy	以 ε 概率随机动作	离散动作、入门首选	ε 需从 1.0 衰减到 0.01—0.05

策略	做法	适用	注意
Boltzmann / Softmax	按 Q 值的 Softmax 概率采样	需要平滑探索	温度决定探索强度
UCB（置信上界）	选 $Q + c\cdot\sqrt{(\ln t / n_a)}$ 最大的	多臂老虎机、理论保证好	c 控制探索权重
Thompson 采样	按后验分布采样再贪婪	推荐、A/B 测试	需维护后验，贝叶斯友好
熵正则	在损失里加策略熵奖励	PPO/SAC 标配	系数过大则策略趋于均匀
内在好奇/计数奖励	给新奇状态额外奖励	奖励稀疏的环境	需设计状态表示

7.6　三大流派横向对比

维度	基于价值（DQN）	基于策略（REINFORCE）	Actor-Critic（PPO/SAC）
学什么	Q(s,a) 值函数	策略 $\pi(a\|s)$ 直接参数化	同时学 V/Q 与 π
动作空间	仅离散	离散 + 连续均可	离散 + 连续均可
输出策略	确定性（贪婪）	随机策略	随机策略
样本效率	较高（可 off-policy 复用）	低（on-policy，用完即弃）	中（PPO 可多 epoch 复用）
训练稳定性	中（需目标网络）	低（方差大）	高（clip / 双 Q 等约束）
收敛保证	表格型有，函数近似无	局部最优	局部最优
代表算法	Q-learning、DQN、Rainbow	REINFORCE	A2C/A3C、PPO、TRPO、SAC、DDPG
典型场景	游戏（Atari）、离散决策	简单控制、教学	机器人控制、LLM 对齐、推荐

7.7　强化学习的优缺点与应用

维度	说明
解决的问题	序贯决策问题：当前动作会影响未来状态与收益，且没有现成的标注答案
核心原理	把问题建模为 MDP，通过与环境试错交互最大化长期折扣回报；用贝尔曼方程做自举，用策略梯度直接优化
时间复杂度	取决于交互次数：仿真环境可千万帧并行，真实环境受物理时间限制

维度	说明
空间复杂度	经验回放缓冲区常占 GB 级；模型本身通常不大（MLP 或小型 CNN）
优点	不需要标注答案、能优化长期目标而非单步、可处理动态环境、天然支持探索新策略
缺点	样本效率极低（Atari 需上亿帧）、奖励函数极难设计、训练不稳定且难复现、奖励黑客（钻奖励空子）、真实环境试错成本高甚至危险
典型应用	游戏 AI（AlphaGo、Dota、Atari）、机器人控制与具身智能、推荐系统与广告出价、资源调度（数据中心冷却、芯片布局）、量化交易、对话策略优化、LLM 的 RLHF 对齐（见第 7 章）
关键参数 / 调参要点	折扣因子 γ（0.9—0.99）、学习率（RL 通常比监督学习小 10 倍）、GAE 的 λ（0.95）、PPO 的 ε（0.1—0.3）、熵系数、batch size 与 epoch 数、回放缓冲容量、环境并行数

奖励黑客（Reward Hacking） 这是 RL 落地最常见的失败模式：智能体找到了一个能让奖励函数打高分、但完全违背设计者意图的捷径。经典例子是赛艇游戏里智能体不去跑终点，而是反复撞击能加分的目标刷分。对策：奖励函数加入约束项、用多个奖励信号交叉验证(把数据轮换着切分多次评估，取平均结果)、引入人类反馈（这正是 RLHF 的动机之一），以及始终保留一个 KL 惩罚让策略不要跑离初始策略太远。

7.8 强化学习与监督学习：什么时候该用哪个

判断点	选监督学习	选强化学习
有没有标准答案	有（每个输入都有正确标签）	没有（只有好/坏的评分或奖励）
决策是否影响未来	单步预测，不影响后续输入	多步序贯，当前动作改变后续状态
关注即时还是长期	优化单步准确率(全部预测里预测对的比例)	优化长期累计收益
数据怎么来	已有历史数据，静态	需要与环境交互，自采集
成本与风险	低，可离线复现	高，可能需要大量试错
现实建议	能用监督学习解决的绝不用 RL	只有当「确实要优化长期目标且无标注」时才上 RL

一个实用的折中：先用监督学习（或模仿学习(直接照着专家的示范数据学，不做试错) / 行为克隆）从专家数据里学一个不错的初始策略，再用 RL 在此基础上微调。这既解决了 RL 冷启动(系统刚上线时没有数据、效果差的阶段)慢的问题，也大幅降低了探索的风险——大模型的 RLHF 正是这条路：先在海量文本上做监督预训练，再用人类偏好做强化微调。

7.9　本章常见错误与练习

强化学习的调试成本比监督学习高一个数量级，因为「训练不收敛」可能来自环境、奖励、超参数或实现细节中的任何一处。

一是奖励函数设计不当，模型钻空子。经典的例子是训练赛船 AI 时只奖励得分，结果模型发现原地转圈反复得分更快，干脆不往前走了。

二是回报没有折扣或折扣设置不合理，导致智能体只关心远期收益而忽略眼前的惩罚。

三是探索不足或过度。ε 一直取 1 等于随机乱走，一直取 0 等于不再探索；要用线性衰减或熵正则让它在训练早期多探索、后期多利用。

四是经验回放池实现有误，把「下一状态」记成了当前状态，或者采样时没有随机，训练就会失效。

五是把「训练环境的得分」当成「真实能力的评估」。评估必须在固定策略下、关闭探索、多种子重复，才能得到可信的曲线。

六是忽视样本效率。同等的样本量下，基于价值的方法与基于策略的方法表现可能相差数倍，先估算样本预算再选算法。

七是忘记随机种子。RL 的方差极大，单次实验的结论几乎不可信，至少要跑三个不同种子看均值与方差。

三道练习：第一题，在网格世界里实现 Q-learning，并画出带探索策略的路径。第二题，把 ε 从固定值改成线性衰减，比较两种设置的收敛速度。第三题，用同一个环境分别跑 Q-learning 和 REINFORCE，比较它们的样本效率与稳定性。

7.10　SARSA 与 PPO：把「学价值」和「学策略」各推进一步

7.3 节的 Q-learning 和 7.4 节的 REINFORCE 是两个极端的代表：一个学一张价值表再间接选动作，一个直接学动作的概率。本节把这两条路线各往前推一步——在价值路线上，SARSA 展示了「同策略」与「异策略」的差别究竟落在哪一行代码上；在策略路线上，PPO 用一个 min 和一次截断，把策略梯度从「能用」变成了「好用」，也因此成了今天大模型对齐（RLHF）里最常用的算法。

7.10.1　SARSA：和 Q-learning 只差一个 max

Q-learning 的更新式是：

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [\, r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a) \,]$$

注意那个 max：它假设「下一步会走当前认为最好的动作」。但是智能体在训练时其实并不总是走最好的动作——它有 ε 的概率在随机探索。于是 Q-learning 学的是一套「理想中的最优策略」，自己却按另一套「带随机探索的策略」在行动。

SARSA 把 max 拿掉，改成「下一步实际会走的那个动作 a′」：

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [\, r + \gamma \cdot Q(s', a') - Q(s, a) \,]$$

只差一个 max，语义却完全不同：SARSA 评估的正是「自己现在这套还带探索的策略」，所以它属于同策略（on-policy）方法，而 Q-learning 属于异策略（off-policy）方法。

这个差别在危险环境里会立刻显形。设想一条「悬崖行走」的任务：贴着悬崖走是捷径，但掉下去要扣 100 分。Q-learning 因为一直假设「以后会走最优动作」，会学出一条紧贴悬崖的最优路线；SARSA 知道「自己偶尔会随机乱走，贴着悬崖走迟早掉下去」，于是会主动离悬崖远一格。代价是路程长一点，收益是训练中真的掉下去的次数少得多。如果训练本身有成本（机器人会摔坏、推荐系统会伤用户体验），SARSA 这种保守往往是更好的选择。

对比项	Q-learning	SARSA
更新时用到的下一步动作	max Q(s′, ·)，最好的那个	a′，实际会走的那个
策略类型	异策略 off-policy	同策略 on-policy
收敛到	最优策略（假设探索充分）	当前策略的价值，因此更保守
在悬崖边的表现	贴边的最优路线	离悬崖更远的稳健路线
一行代码的差别	target = r + γ·max Q(s′,·)	target = r + γ·Q(s′, a′)

```
# SARSA 的训练循环：注意 a 是在上一轮就选好的
a = pick_action(s)                          # 先按 epsilon-贪心选出本步动作
for t in range(STEPS):
    s2, r, done = env.step(s, a)
    if done:
        target = r                          # 终止状态没有未来
        s, a = env.reset(), pick_action(env.start)
    else:
        a2 = pick_action(s2)                # ① 先选出下一步真正要走的动作
        target = r + GAMMA * Q[(s2, a2)]
        s, a = s2, a2
    Q[(s, a)] += ALPHA * (target - Q[(s, a)])   # ② 用它更新
```

7.10.2 PPO：把「不许一次走太远」写成一个 min

策略梯度有个老毛病：一次更新如果步子迈大了，策略可能直接退化——比如把所有概率都压到一个没用的动作上，之后再也抽不出别的动作，训练就此崩掉。TRPO 用二阶优化给出了理论保证，但实现复杂、算得慢。PPO（Proximal Policy Optimization，2017）用一个极其简单的方法解决了同一个问题。

它先定义新旧策略的概率比 $r(\theta) = \pi_new(a|s) / \pi_old(a|s)$，然后优化这么一个目标：

$$L = \min(r \cdot A, \text{clip}(r, 1 - \varepsilon, 1 + \varepsilon) \cdot A)$$

其中 A 是优势（这个动作比平均水平好多少），ε 一般取 0.2。这个式子里有两层保护。

第一层是 clip。它把 r 限制在 $[1 - \varepsilon, 1 + \varepsilon]$ 这个区间里。如果某个动作的优势是正的（$A > 0$，值得鼓励），r 涨到 $1 + \varepsilon$ 之后就被截住了，目标函数变成一条水平线，梯度为零——也就是说「就算这个动作好得离谱，也不许把它的概率一次提得太高」。如果优势是负的（$A < 0$，应该压制），r 掉到 $1 - \varepsilon$ 以下同样被截断，梯度为零——「不许一次把某个动作压到地板上」。

第二层是 min。取两者中较小的那个，保证了目标函数是实际收益的悲观下界：能拿到的改进一定不会超过被截断后允许的幅度。合起来的效果是：策略可以变，但每一步只能挪动一点点。

```
def ppo_loss(r, A, eps=0.2):
    """r = pi_new / pi_old，A = 优势"""
    clipped = max(1 - eps, min(1 + eps, r))
    return -min(r * A, clipped * A)      # 加负号是因为优化器在做梯度下降

def ppo_grad(r, A, eps=0.2):
    """对 r 的导数：被截断的样本梯度为 0，这正是「保护」生效的地方"""
    if r * A <= max(1 - eps, min(1 + eps, r)) * A:
        return -A
    return 0.0
```

优势 A 怎么算？常用的是广义优势估计 GAE（Generalized Advantage Estimation，2015）。它把每一步的 TD 误差 $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ 按 λ 加权累加：

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

翻译成代码就是从后往前扫一遍数组。$\lambda = 0$ 时只看一步（偏差大、方差小），$\lambda = 1$ 时看完整条轨迹（无偏、方差大），实践中 0.95 是最常见的折中。

```
def gae(rewards, values, gamma=0.99, lam=0.95):
    """rewards / values 是同一批轨迹上的序列，返回每个时刻的优势"""
    A, adv = 0.0, [0.0] * len(rewards)
    for t in reversed(range(len(rewards))):
        next_v = values[t + 1] if t + 1 < len(values) else 0.0
        delta = rewards[t] + gamma * next_v - values[t]   # TD 误差
        A = delta + gamma * lam * A                       # 反向累积
        adv[t] = A
    return adv
```

PPO 的完整循环只有五步，值得背下来：① 用当前策略采样一批轨迹；② 用价值网络算出优势（通常配 GAE）；③ 在同一个 batch 上重复 K 轮（这是 PPO 相对 REINFORCE 最大的优势——样本可以复用）；④ 每一步算 ratio 和截断后的损失，反传更新；⑤ 另外加一项价值损失和一项熵奖励，前者让价值网络更准，后者防止策略过早变得确定。

对比项	REINFORCE	Actor-Critic	PPO
学到的东西	只有策略 π	策略 π + 价值 V	策略 π + 价值 V
优势怎么来	回报减基线	一步 TD 误差	GAE($\lambda = 0.95$)
一次样本能用几次	1 次，用完就丢	1 次	可以重复 K 轮
稳定性	方差大	中等	好，且有截断兜底
今天的实际地位	教学与入门	A2C / A3C 的基础	RLHF、游戏、机器人中的主力

7.10.3 PPO 与 RLHF：大模型对齐为什么选它

在大模型语境下，PPO 的四个角色是这样对应的：策略 π 就是语言模型本身（动作是「下一个 token」，状态是「已经生成的上下文」）；奖励来自一个单独训练的奖励模型（Reward Model），它给整段回答打一个分；优势由 GAE 估计；而为了防止模型为了刷分而输出怪话，损失里还会额外加一项 KL 惩罚，把当前模型拉住、别偏离微调前的模型太远。

这解释了一个常见的疑惑：为什么 RLHF 里明明有监督数据，却还要用强化学习？因为「有用、无害、诚实」这类目标很难写成一条条标准答案，但很容易写成「A 回答比 B 回答好」的比较。奖励模型把这种比较学成一个打分器，PPO 再拿这个打分器当环境信号去优化语言模型——这正是它擅长的场景。

后续的改进也沿着同一条思路。GRPO（DeepSeekMath，2024；DeepSeek-R1 也用它）去掉了价值网络，改成对同一个问题采样多次、用组内的平均分当基线，省下了和策略网络同样大的价值网络那部分显存；DPO 则干脆跳过强化学习，直接用一个闭式损失在偏好数据上微调——省事，但对数据的质量要求更高。

7.10.4 本节常见错误与练习

- 把 SARSA 写成「先执行动作再选 a′」是最常见的错误。SARSA 的顺序必须是「先选 a′，再更新」，因为 a′ 要参与本轮更新。
- PPO 里 ratio 一定要用「当前策略的 log 概率减去采样时旧策略的 log 概率再取指数」来算，不能直接用两次前向得到的概率相除——后者在数值上不稳定，而且容易忘记「旧策略那一份必须冻结」。
- 优势做归一化（减均值、除标准差）几乎是 PPO 的必备步骤。少了这一步，优势的尺度会随奖励尺度漂移，学习率就失去意义。
- ε 不是越大越好。0.2 是经验值；ε 小训练稳但慢，ε 大学习快但容易崩。工程上还会根据 KL 散度自适应地调 ε。

三道练习。第一题，把 7.3 节的 Q-learning 和本节的 SARSA 放在同一张迷宫、同一个随机种子下并排跑，把两张 Q 表差分画出来，看看它们在哪一类格子上分道扬镳。第二题，手写 PPO 的损失函数，画出 $A = 1$ 和 $A = -1$ 时损失随 r 变化的曲线，标出梯度为零的区间。第三题，把 GAE 的 λ 从 0 扫到 1，观察同一批数据上优势估计的方差如何变化。

配套仿真：在「算法动画仿真实验室」的强化学习分组里，「Q-Learning」「SARSA」「REINFORCE」「PPO 截断目标」四个仿真可以并排对照——前两个画的是价值表如何被填满，第三个画的是三个动作的概率曲线如何收敛，第四个直接把 PPO 的截断目标函数画成曲线，还能改 ε 看允许区间怎么变。四个仿真都提供五种语言的完整实现。

7.11　TD(λ) 与资格迹：往前看的另一种记账方式

7.10.1 讲过 SARSA 与 Q-learning 的差别，那个差别可以浓缩成一句话：更新时看的是「实际走的那一步」，还是「看起来最优的那一步」。这一节要讲的是另一个维度上的选择：更新时往前看几步。

蒙特卡洛方法要等一整条轨迹跑完，用最终回报来更新，估计是无偏的，但方差极大，因为一条路上所有的随机性都压进了一个数。TD(0) 每走一步就用「即时奖励 + 下一状态的估值」更新一次，方差小得多，代价是引入了估计偏差，收敛更快。这两者之间不是一个二选一，而是一整条连续的谱，由参数 λ∈[0,1] 控制。

前向视角的想法非常直接：把 1 步回报、2 步回报、……、n 步回报按 (1−λ)λ^(n−1) 加权平均，得到所谓的 λ 回报。λ=0 时只有 1 步回报有权重，退回 TD(0)；λ=1 时权重均摊给所有步，等于蒙特卡洛。但这个写法有个致命缺点：算 λ 回报需要等未来真的发生，没法在线更新。

后向视角给出了可以在线执行的等价算法。给每个状态（或每个状态—动作对）配一个资格迹 e：每访问一次，自己的资格加一；其余位置的资格随时间按 γλ 衰减。然后每走一步算出一个 TD 误差 δ，用 δ 乘以资格迹去更新所有位置。

它的直觉很好理解：刚才发生的那次误差，应该由「最近走过的状态」共同负责，走得越近、走得越频繁的，责任越大。衰减因子 γλ 正好表达了这个意思：久远的记忆随时间长掉。

数学上可以证明，前向视角累计的总更新量和后向视角的更新量是一致的，这就是 TD(λ) 的经典等价性结论。后者的好处是随时可更新，而且额外空间只需要一张和状态空间同大的资格迹表。λ=0 时它就是 SARSA，λ=1 时它接近蒙特卡洛。一个参数，把两种极端连成了一条线。

方法	更新依据	偏差	方差	能否在线更新
TD(0) / SARSA	1 步回报	有偏	低	能
n 步 TD	n 步回报	随 n 增大而减小	随 n 增大而增大	要等满 n 步
蒙特卡洛	整条轨迹的回报	无偏	高	不能
TD(λ) 前向视角	多步回报的加权平均	介于两者之间	介于两者之间	不能
TD(λ) 后向视角	资格迹 × 当前 TD 误差	同上	同上	能

代码 7-6　SARSA(λ)：资格迹版本（前向视角的在线等价）

```
import numpy as np

def eps_greedy(Q, s, eps):
    if np.random.rand() < eps:
        return np.random.randint(Q.shape[1])
    return int(np.argmax(Q[s]))
```

```
def sarsa_lambda(env, episodes=500, alpha=0.1, gamma=0.95,
                 lam=0.9, eps=0.1):
    Q = np.zeros((env.nS, env.nA))
    for _ in range(episodes):
        E = np.zeros_like(Q)              # 资格迹，与 Q 同形
        s = env.reset()
        a = eps_greedy(Q, s, eps)
        while True:
            s2, r, done = env.step(a)
            a2 = eps_greedy(Q, s2, eps)
            # 与 SARSA 唯一的区别就在下面三行
            delta = r + (0 if done else gamma * Q[s2, a2]) - Q[s, a]
            E[s, a] += 1                  # 走过的这一步加资格
            Q += alpha * delta * E        # 一次误差更新所有有资格的
            E *= gamma * lam              # 其余位置随时间衰减
            if done:
                break
            s, a = s2, a2
    return Q

# lam = 0 时 E 每步只剩一格，退化成普通的 SARSA
# lam = 1 时资格不衰减，接近蒙特卡洛
```

> **常见错误** 一是忘了随步数衰减资格迹（少写 E *= gamma * lam），等于让整条轨迹永久有效，方差会爆炸。二是把资格迹在回合结束后忘了清零，跨回合污染了估计。三是把 λ 和 γ 的乘积当成两个可以互换的参数：它们在这个乘积里确实对称，但 γ 同时出现在 TD 误差里，改 γ 会同时改两件事。四是只给更新过的状态加资格，而忘了「经过但没更新」的状态同样应该记一笔。

配套仿真：搜索「TD(λ)」，可以看到资格迹表随步数怎样一次次被加一、又被整体衰减，右侧同时把每一步的 TD 误差与更新量并排列出。把 λ 从 0 拉到 1，观察同一条轨迹上的更新范围怎样从「一格」扩散到「整条」，这个参数到底在调什么就非常具体了。

7.12 Double DQN：把「选动作」和「评价值」拆开

7.3 里提到过 DQN 家族的若干改进，其中最容易实现、收益又最直观的一个是 Double DQN（van Hasselt 等，2015）。它要解决的只有一个问题：系统性高估。

DQN 的目标值是 $y = r + \gamma \cdot \max_a Q(s', a; \theta^-)$。问题就出在这个 max 上：同一个 Q 网络既负责挑出「哪个动作看起来最好」，又负责给出「它到底有多好」。而神经网络的估计误差总是双向的：在取最大值的时候，被高估的那些动作会被选中，被低估的则被直接丢掉。于是 max 系统性地偏向被高估的值，Q 值随训练一路偏高，策略基于一个整体偏高的价值表做决策，最终表现被拉低。

修法只有一处改动：把「挑动作」和「打分」交给两个网络分头做。用在线网络挑出动作 $a^* = \arg\max_a Q(s', a; \theta)$，再用目标网络给这个动作打分 $Q(s', a^*; \theta^-)$。选与评一分离，

被高估的动作就不再有自动优势：如果在线网络挑了一个被高估的动作，目标网络打出来的分会更保守，误差被抵消掉一部分。

代价接近于零。目标网络本来就是 DQN 的标配（为了稳定训练引入的），所以 Double DQN 没有增加任何额外的计算或显存。这也是它几乎总是「第一个该加的改进」的原因。

改进	解决的问题	一句话做法	额外成本
Double DQN	Q 值系统性高估	在线网络选动作，目标网络给分	没有
Dueling DQN	很多状态下各动作价值差别不大	拆成状态价值 V 与优势 A 两支	网络结构略改
优先经验回放	均匀采样浪费了「最值得学」的样本	按 TD 误差大小决定采样概率	需维护优先级结构
n 步回报	单步更新的信息传播太慢	用 n 步实际奖励替代单步	需缓存 n 步数据
Noisy Net	ε-greedy 的探索太机械	直接给网络权重加噪声	参数量略增

代码 7-7　Double DQN 的目标值：在线网络选，目标网络评

```
import torch
import torch.nn.functional as F

def double_dqn_loss(batch, online, target, gamma):
    """batch = (s, a, r, s2, done)，均为张量"""
    s, a, r, s2, done = batch
    with torch.no_grad():
        # ① 在线网络决定「下一步走哪个动作」
        a_star = online(s2).argmax(dim=1, keepdim=True)
        # ② 目标网络只负责给这个动作打分
        q_next = target(s2).gather(1, a_star).squeeze(1)
        y = r + gamma * q_next * (1.0 - done)
        # 对照：普通 DQN 用下面这一行，max 由目标网络自己取
        # q_next = target(s2).max(dim=1).values
    # ③ 只对「实际执行过的那个动作」计算误差
    q_sa = online(s).gather(1, a.view(-1, 1)).squeeze(1)
    return F.smooth_l1_loss(q_sa, y)   # Huber 损失，对离群值更稳

# 训练循环里唯一的变化：把 td_target 换成上面这个函数的输出
# loss = double_dqn_loss(batch, online, target, gamma)
# loss.backward(); opt.step()
```

容易写反的地方　一是把两个网络的顺序写反：必须是「在线网络选、目标网络评」，反过来就等于没改，高估照旧。二是忘了在目标值计算外面套 no_grad，梯度会顺着目标网络回传，把目标网络也一起带偏。三是以为 Double DQN 能消除全部偏差：它降低的是 max 引入的那一部分，函数逼近本身带来的误差依然存在。四是同时把 n 步回报和 Double 叠加时忘了同步调整 γ 的幂次。

配套仿真：搜索「Double DQN」，可以看到同一批转移样本上，普通 DQN 与 Double DQN 算出来的目标值并排对比，右侧累加记录两者的 Q 值平均估计偏差。普通 DQN 的曲线会稳定地待在零线以上，Double 那条则贴着零线附近来回。

7.13　蒙特卡罗与时序差分：两条算价值的路线

要估计「在这个状态下最终能拿多少分」，有两种算法。蒙特卡罗（Monte Carlo）是「跑到底再算账」：从现在开始一直玩到结束，把实际拿到的奖励加起来。时序差分（Temporal Difference, TD）是「走一步就算账」：走一步后，用下一个状态的估计值加上刚拿到的奖励来修正当前估计。

打个比方：你在考试。蒙特卡罗是「等成绩出来再总结」，准但慢；TD 是「每做完一道题，就根据自己估的分修正一下预测」，快但带着估计的偏差。两者各有用处：环境结束得很快时用蒙特卡罗；环境很长、没有明确终点时用 TD。

表 7-5　蒙特卡罗与 TD 的对比

对比项	蒙特卡罗（MC）	时序差分（TD）
更新时机	一个完整序列结束后	每走一步立刻更新
方差	大	小
偏差	无偏	有偏（用估计修正估计）
对环境的要求	必须会结束	可以一直下去
典型算法	REINFORCE	Q-learning、SARSA、TD(λ)

记住一句　Q-learning 里那句 $Q(s,a) \mathrel{+}= \alpha[r + \gamma \cdot \max Q(s',a') - Q(s,a)]$，括号里的东西就叫 TD 误差。它是强化学习里出现频率最高的一个词。

7.14　探索策略与奖励设计：两个最容易出错的地方

强化学习与监督学习最大的区别，是没有标准答案，只有自己试出来的经验。如果只挑自己认为最好的动作，就永远发现不了别的可能，这叫「囚在局部最优」。所以要有探索（Exploration）。

表 7-6　四种探索策略

策略	做法	特点
ε-贪心	以概率 ε 随机试，否则选最优	最常用，ε 从 1 逐渐减到 0.01
Softmax / Boltzmann	按 Q 值的概率抽样	好动作被选概率更高，但不会完全丢掉差的
UCB（上置置信区间）	优先试访问次数少的动作	探索有目的性，不是乱试
计划式探索	每隔一段时间彻底随机一次	用于有安全隐患的场景

奖励设计（Reward Shaping）则是更容易被忽略的坑。模型只会做「能拿到奖励的事」，不会做「你心里想让它做的事」。比如一个清扫机器人，如果你按「扫干净的面积」给奖励，它会学会原地转圈；按「每丢一次垃圾给一分」，才会学会去找垃圾。

> **实践建议**　奖励宁可稀疏而精确，不要密集而模糊。另外记住：每次改奖励函数，都等于重启一次实验，之前调好的超参数大概率不再适用。

7.15　表格版 Q-learning 手推：把 3×4 的迷宫走一遍

强化学习最让人望而生畏的地方是「不知道它在学什么」。这一节把最基础的 Q-learning 放在一个只有 12 个格子的迷宫里，一次一次把数字算给你看，走完你就明白 Q 表到底是什么。

7.15.1　迷宫设定

4×3 的格子，起点在左下角 S，终点在右上角 G，中间有一个陷阱 T。每走一步扣 1 分，走到终点加 10 分，踩到陷阱扣 10 分。动作有四个：上、下、左、右。撞墙就原地不动，仍然扣 1 分。

表 7-18　迷宫的初始 Q 表（每格四个动作，初始全 0）

格子	上	下	左	右	说明
S(0,0)	0	0	0	0	起点，左下角
(1,0)	0	0	0	0	起点右边一格
G(3,2)	0	0	0	0	终点，右上角（进入后回合结束）
…其余 9 格	0	0	0	0	全部从 0 开始

7.15.2 一条完整的回合

所谓一个回合，就是从起点出发，一直走到终点或者步数上限。每走一步做一个更新：$Q(s,a) \leftarrow Q(s,a) + \alpha[\, r + \gamma \cdot \max Q(s'', \cdot) - Q(s,a)\,]$。这个式子看着长，拆开只有三个部分。

表 7-19 更新公式的三段含义

部分	符号	含义	在迷宫里的解释
现状	$Q(s,a)$	我以前认为这一步值多少	我原来觉得从这里往右值 0 分
现实	$r + \gamma \cdot \max Q(s'',\cdot)$	走完这一步，实际拿到多少加未来最好能拿多少	走一步扣 1，然后从新位置还能争取到 +8
误差	现实 − 现状	我错估了多少	真实值是 7，我却以为 0，差 7
修正	$\alpha \times$ 误差	按学习率把误差吸收一部分	$\alpha = 0.1$，就往正确方向挪 0.7

这样一来，Q 表就像一个「经验值账本」：每个位置每个动作都记着「从这里出发大概能得多少分」。反复走，账面数字越来越准。

7.15.3 手工推演五次更新

取 $\alpha = 0.5$、$\gamma = 0.9$。假设某条轨迹经过 S(0,0)，选择「右」，到达 (1,0)，此刻 (1,0) 上的最大 Q 值还是 0（因为没走过）。

表 7-20 一条轨迹上的五次 Q 值更新

步	状态 s	动作 a	奖励 r	Q(s'',·) 的最大值	更新后 Q(s,a)
1	(0,0)	右	-1	0	0 + 0.5×(-1 + 0.9×0 - 0) = -0.50
2	(1,0)	右	-1	0	0 + 0.5×(-1 - 0) = -0.50
3	(2,0)	右	-1	0	-0.50
4	(3,0)	上	-1	0	-0.50
5	(3,1)	上（进入终点 G）	+10	0（终点没有后续状态）	0 + 0.5×(10 - 0) = +5.00
6（第二遍）	(3,0)	上	-1	5.00（(3,1) 往上已经是 5）	-0.50 + 0.5×(-1 + 0.9×5 - (-0.50)) = +1.50

看第 6 步：第二遍经过 (3,0) 时，它往上走到 (3,1)，而 (3,1) 往上已经记着 5.00 分。回头一算，发现「未来能拿 5 分」，于是 (3,0) 往上的价值从 -0.5 一口气提到 +1.5。这正是 Q-learning 最妙的地方——它把「终点那一格的好经验」通过第 5 步的 +5.00，一步步往回传。

7.15.4　为什么叫「离策略」算法

Q-learning 更新时用的是 max Q(s",·)，也就是「假设后面每一步都选最好的动作」。但实际探索时你未必选最好的（要留出探索的余地）。学的是「最优策略的价值」，走的却是「带探索的策略」——两者不是同一个策略，所以叫离策略（off-policy）。

> **和 SARSA 的区别**　SARSA 把 max Q(s",·) 换成「实际下一步真的选的那个动作的价值」Q(s",a")。学的是自己正在执行的策略，所以在悬崖边这类危险任务上，SARSA 会学到一条更保守、更安全的路线。

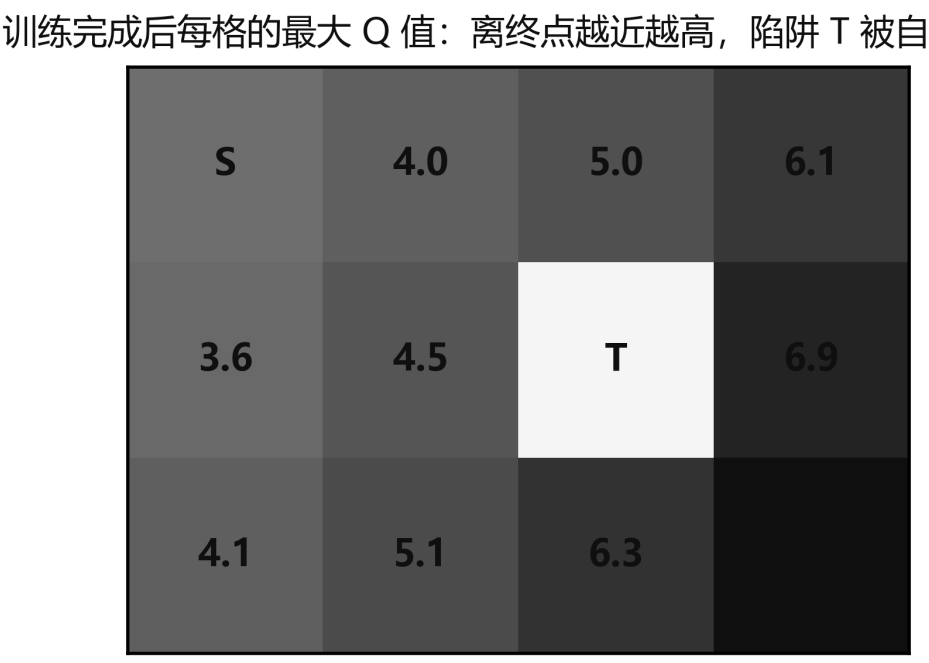


图 7-3　Q-learning 收敛后的 Q 表：离终点越近值越高，陷阱自动被避开

7.16　策略梯度手推：REINFORCE 的一次完整更新

基于价值的方法（Q-learning、DQN）先学「每个动作值多少」，再挑最大的。基于策略的方法换了个思路：直接学「在每个状态下该以什么概率选每个动作」。这一节用只有两个动作的小例子，把 REINFORCE 的一次更新算清楚。

7.16.1 策略是什么形状的

某个状态下有两个可选动作 a 和 b。策略用两个数（logits）表示偏好，经过 softmax 变成概率。比如 logits 是 [1.0, 0.0]，那么 a 的概率是 $e^1/(e^1+e^0) = 0.731$，b 是 0.269。

```
import math
def softmax(z):
    m = max(z)
    ex = [math.exp(x - m) for x in z]
    s = sum(ex)
    return [x / s for x in ex]
```

> **一句话原理** 把一组任意大小的分数，变成一组加起来等于 1 的概率，而且原来大的仍然大。

智能体要在几个动作里选一个，可它算出来的是几个「分数」，不是概率。softmax 负责把这个分数翻译成概率：先取指数让差距拉开，再除以总和让整体归一。

为什么要先取指数？因为指数函数天然把「微小的差距」放大成「明显的偏好」，而且永远为正数，不用担心出现负概率。分数领先一点点，对应的概率就会领先很多，这让选择更有倾向性。

- 第一步：对每个分数取指数，保证结果都是正数。
- 第二步：把所有指数值加起来，得到分母。
- 第三步：每个指数值除以这个总和，就得到了加起来等于 1 的概率。

这也是「探索」的来源：概率大的动作常被选，但不是必选。如果把 softmax 的结果直接当成策略，智能体就会一直探索下去，这也是强化学习能发现更优解的原因之一。

> **易错点** 分数差到几百时，直接取指数会溢出成无穷大。工程实现里要先减去最大值再取指数，结果一模一样但数值安全。

```
print(softmax([1.0, 0.0]))   # [0.731, 0.269]
```

代码 7-30 softmax：把任意两个数变成加起来等于 1 的概率

7.16.2 一次采样与一次更新

假设我们按当前策略选动作，抽到了 a，整条轨迹结束后的总回报 G = 5。REINFORCE 的更新规则是：让「被选中的动作」的概率上升，上升的幅度正比于这次的回报。

对 softmax 加上交叉熵损失，梯度有一个非常清爽的形式：∂ 对数概率 / ∂ logits = 一个动作独热向量 − 概率向量。以选中的是 a（下标 0）为例，独热向量是 [1, 0]，概率向量是 [0.731, 0.269]，差是 [0.269, -0.731]。

表 7-21　REINFORCE 一次更新的完整过程（学习率 0.1，回报 G = 5）

量	数值	说明
当前 logits	[1.0, 0.0]	策略当前的偏好
当前概率	[0.731, 0.269]	选 a 的概率本来就比较高
采样得到的动作	a	运气不错，抽到了高概率动作
本次回报 G	5	整条轨迹的总回报，是个正数
梯度 $\partial \log\pi/\partial z$	[0.269, -0.731]	独热 − 概率
logits 更新量	0.1 × 5 × [0.269, -0.731] = [0.135, -0.366]	回报为正，于是放大这个动作
更新后 logits	[1.135, -0.366]	a 的偏好更强，b 更弱
更新后概率	[0.818, 0.182]	a 从 73% 涨到 82%

> **如果回报是负的呢**　G = -5 时，更新量整体变号，被选中的动作概率反而下降。这正是强化学习的核心逻辑：好结果强化刚才的行为，坏结果抑制刚才的行为。

7.16.3　两个致命问题与它们的解法

REINFORCE 朴素、漂亮，但在实际任务上几乎没法直接用，原因是两个问题。

表 7-22　REINFORCE 的两个问题与对策

问题	症状	对策
方差极大	同样的策略，两次训练结果差很多	减去基线（baseline），常用状态价值函数
回报尺度影响更新幅度	总回报 1000 的任务一步更新太猛	对优势做标准化，或改用优势函数 A = G − V(s)

加上基线之后再配上「用 critic 估计价值」，就得到了 Actor-Critic；再把「更新幅度不能太大」用比值裁剪约束住，就得到了 PPO——也就是第 7.4 节和 7.10 节讲的那个在工程上最常用的算法。整条演化路径，都是从 REINFORCE 这个小小的更新规则长出来的。

第 8 章　Transformer 与大语言模型（LLM）★本文重点

> **读前提示**　本章最厚也最实用。前置知识：第 6 章的神经网络与反向传播、第 7 章的策略梯度概念（讲 RLHF 时会用到）。实在赶时间，至少先看 8.1—8.3（分词与 Transformer 结构）和 8.11（性能参数速查表）。

LLM 不是一个全新学科，而是「Transformer 架构 + 海量数据 + 超大算力 + 对齐技术」四件事叠加的产物。本章从注意力讲起，一路讲到训练、微调、推理与工程参数，是全篇最厚的一章。其中「对齐」用到的 RLHF / PPO / GRPO 属于强化学习，相关原理（MDP、价值函数(某个状态或动作，从长远看有多好)、策略梯度、PPO 裁剪）已在第 7 章铺垫，本章只讲它们在语言模型上的具体用法。

8.1　语言建模：一切从「预测下一个 token」开始

语言模型的训练目标是最大化似然：P(x) = Π_t P(x_t | x_<t)。自回归模型（GPT 系）逐词预测下一个 token；自编码模型（BERT 系）遮住部分 token 再还原。前者天然适合生成，后者适合理解类任务，这也是为什么今天生成式 AI 全面走向自回归路线。

分词：BPE（Byte Pair Encoding）

模型看不到字符，只看到 token id。BPE 从字符表出发，反复合并出现频次最高的相邻符号对，直到词表达到目标大小（常用 32k—200k）。它本质是 4.7 节提到的贪心合并策略，好处是能把罕见词拆成已知子词，兼顾词表大小与覆盖率。中文通常 1 个汉字 ≈ 1—2 个 token，这也是中文 API 计费与上下文估算的经验基础。

代码 8-1　BPE 分词训练（贪心合并，约 15 行）

```
from collections import Counter

def bpe_merge(corpus, vocab_size):
    """corpus: {word: freq}，word 内部字符已用空格分隔"""
    vocab = set("".join(corpus.keys()).replace(" ", ""))
    while len(vocab) < vocab_size:
        pairs = Counter()
        for w, f in corpus.items():
            syms = w.split()
            for a, b in zip(syms, syms[1:]):
                pairs[(a, b)] += f                  # 统计相邻符号对频次
        if not pairs:
```

```
            break
        best = max(pairs, key=pairs.get)            # 贪心：合并频次最高的一对
        new = "".join(best)
        corpus = {w.replace(" ".join(best), new): f for w, f in corpus.items()}
        vocab.add(new)
    return vocab
# 编码阶段：按合并优先级从高到低依次应用规则；未知字符回退到字节级 BPE（byte-level）
```

参数 / 符号	含义	取值与影响
corpus	{词: 频次} 的字典	词内部字符已用空格分隔，例如 {'l o w': 5}
vocab_size	目标词表大小	常用 32k—200k；太小则每个 token 承载信息少、序列变长，太大则嵌入层参数膨胀
vocab	当前符号集合（初始为字符表）	最终词表 = 初始字符 ∪ 所有合并出来的符号
pairs	相邻符号对及其频次	频次按单词出现次数加权，不是简单的出现次数
best = max(pairs)	贪心：选频次最高的一对合并	这是 3.7 节贪心策略的典型应用；每轮只做局部最优选择
w.replace(…)	把选中符号对在全语料中合并	合并后原符号对消失，产生新符号，下一轮重新统计

代码逐段解读：

- 第 1 步：初始化词表为所有出现过的字符，保证任何输入都能被表示（不会 OOV）。
- 第 2 步：统计所有相邻符号对的加权频次，选出最高的一对——这就是名字里 Pair（相邻对）的含义。
- 第 3 步：把这对符号合并成新符号加入词表，并在语料中所有位置替换掉。
- 第 4 步：重复直到词表达到目标大小。每轮词表 +1，所以轮数 = vocab_size − 初始字符数。
- 为什么有效：高频组合被优先合并，常见的词缀、词根、常见词会整体成 token，罕见词被拆成已知子词，兼顾覆盖率与词表大小。
- 工程要点：现代实现用 byte-level BPE，以字节为初始单位，天然支持任意 Unicode；中文通常 1 个汉字约 1—2 个 token。

8.2 Transformer 架构全景

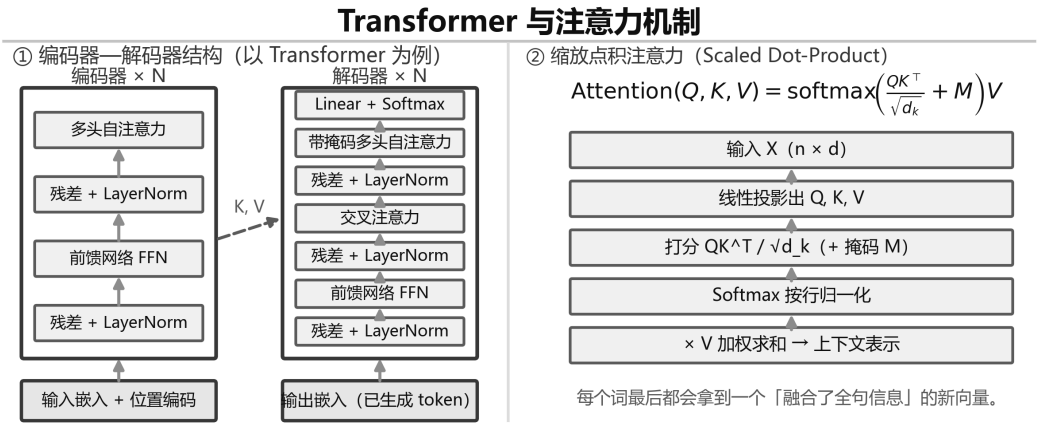


图 8-1　Transformer 编码器—解码器结构与缩放点积注意力计算流程

① 自注意力：让每个位置直接看到全句

对输入序列的每个位置，用三个可学习矩阵投影出 Query、Key、Value：Q = XW_Q，K = XW_K，V = XW_V。然后用 Q 与所有 K 的内积算相关性打分，除以 √d_k 防止内积过大把 Softmax 推到饱和区，再乘到 V 上做加权求和：

代码 8-2　缩放点积注意力（可直接用 F.scaled_dot_product_attention 替代）

```
import torch, torch.nn as nn, torch.nn.functional as F
import math

def scaled_dot_product_attention(Q, K, V, mask=None, dropout=0.0):
    d_k = Q.size(-1)
    scores = Q @ K.transpose(-2, -1) / math.sqrt(d_k)   # (…, n, n)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float("-inf"))  # 因果掩码：挡住未来
    attn = F.softmax(scores, dim=-1)
    if dropout:
        attn = F.dropout(attn, dropout)
    return attn @ V, attn
```

参数 / 符号	含义	取值与影响
Q / K / V	查询、键、值矩阵	形状 (batch, heads, n, d_k)；三者由同一个 X 经不同线性投影得到
d_k	每个头的维度 = d_model / n_heads	典型值 64 或 128；太小表达力不足，太大计算与显存上升
scores	注意力打分矩阵 (…, n, n)	scores[i,j] 表示位置 i 对位置 j 的关注强度

参数 / 符号	含义	取值与影响
math.sqrt(d_k)	缩放因子	不缩放时 d_k 大 → 内积方差大 → Softmax 进入饱和区 → 梯度几乎为 0。这是原论文的关键细节
mask	掩码，1 表示保留、0 表示屏蔽	用 −inf 填充再 Softmax，被屏蔽位置的概率精确变为 0
attn @ V	按注意力权重对 V 加权求和	输出形状与 Q 相同，是「融合了上下文信息」的新表示

代码逐段解读：

- 第 1 步：$Q@K^T$ 得到 n×n 的相关性矩阵，每个位置对所有位置打分——这就是「自注意力」能一步建模任意距离依赖的原因。
- 第 2 步：除以 √d_k。若 Q、K 各分量独立且方差为 1，则内积的方差为 d_k，除以 √d_k 后方差回到 1，Softmax 不会饱和。
- 第 3 步：加掩码。因果语言模型用下三角掩码（causal mask）遮住未来位置，保证预测第 t 个词时看不到 t 之后的内容。
- 第 4 步：Softmax 归一化成概率，再乘 V 做加权求和。
- 复杂度提示：scores 是 n×n 的，这就是注意力 $O(n^2)$ 的来源；FlashAttention 通过分块计算避免真正物化这个矩阵。
- 生产建议：直接用 PyTorch 的 F.scaled_dot_product_attention，它会自动选择 FlashAttention 等高效内核。

② 多头注意力：并行看多个子空间

把 d_model 切成 h 份，每份独立做一次注意力（头数 h 常取 d_model/64，如 4096 维配 32 头或 64 头），再拼接后过一次线性变换 W_O。不同头会自动学到不同的关注模式：有的盯句法依赖，有的盯共指，有的盯标点。

代码 8-3　多头自注意力（含 KV Cache 接口）

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model=512, n_heads=8, dropout=0.1):
        super().__init__()
        assert d_model % n_heads == 0
        self.h, self.dk = n_heads, d_model // n_heads
        self.W_q, self.W_k, self.W_v, self.W_o = (nn.Linear(d_model, d_model) for _
in range(4))
        self.drop = nn.Dropout(dropout)

    def forward(self, x, mask=None, kv_cache=None):
        B, T, C = x.shape
        # 投影并拆成 h 个头：(B, T, C) → (B, h, T, dk)
        q = self.W_q(x).view(B, T, self.h, self.dk).transpose(1, 2)
        k = self.W_k(x).view(B, T, self.h, self.dk).transpose(1, 2)
```

```
        v = self.W_v(x).view(B, T, self.h, self.dk).transpose(1, 2)
        if kv_cache is not None:                    # 推理时拼接历史 K/V（8.8 节）
            k = torch.cat([kv_cache["k"], k], dim=2)
            v = torch.cat([kv_cache["v"], v], dim=2)
        out, attn = scaled_dot_product_attention(q, k, v, mask, self.drop.p)
        out = out.transpose(1, 2).contiguous().view(B, out.size(2), C)   # 拼回
        return self.W_o(out), {"k": k, "v": v}
```

参数 / 符号	含义	取值与影响
d_model=512	模型隐藏维度	决定每层的信息容量；现代 LLM 常用 4096—8192
n_heads=8	注意力头数	通常取 d_model/64；头数太多则每头维度太小，太少则并行子空间不足
self.dk	每个头的维度 d_model//n_heads	常见 64—128
W_q/W_k/W_v/W_o	四个投影矩阵，均为 d_model×d_model	参数量 4·d_model²，是 Transformer 参数的主要组成部分
view + transpose(1,2)	把 (B,T,C) 拆成 (B,h,T,dk)	纯形状变换，不引入参数；拆头是为了让每个头在自己的子空间独立计算
kv_cache	推理时缓存的历史 K、V	拼接后序列变长但只算新 token 的 Q，把单步复杂度从 O(n²) 降到 O(n)
contiguous().view(…)	拼回头并还原形状	transpose 后内存不连续，必须 contiguous 才能 view，这是常见的报错点

代码逐段解读：

- 第 1 步：四个独立线性层把 X 分别投影成 Q、K、V，以及最后的 W_o 输出投影。
- 第 2 步：把 d_model 切成 h 份，每份是一个「头」。头之间完全独立计算，天然可并行。
- 第 3 步：每个头各做一次缩放点积注意力（调用上一个函数），得到各自的上下文表示。
- 第 4 步：拼接所有头的输出，过 W_o 做一次线性变换，把不同子空间的信息融合。
- 多头的意义：不同头会自动学到不同模式——有的盯句法依存，有的盯共指，有的盯标点或位置。这是「用多个低维子空间换一个高维空间」的经典设计。
- KV Cache(键值缓存(把已算过的中间结果存起来复用，避免重复计算)：把算过的中间结果存起来复用)：把本步算出的 K、V 返回，下一步只需对新 token 算 Q，再与缓存拼接，避免重复计算历史。代价是显存随上下文线性增长。

③ 一个完整的 Transformer 块

> **它在组装什么**　把前面几个零件拼成一整层——这就是 Transformer 里重复了几十次的那个积木。

一个完整的 Transformer 块包含四样东西：多头自注意力、前馈网络（两层全连接，中间放大再缩小）、两次残差连接、两次层归一化。模型之所以能堆到上百层还不崩溃，靠的就是残差和归一化这两件「稳定器」。

残差连接是一条捷径：把输入原样加到输出上。这样梯度可以顺着这条捷径直接传到前面，不会在层层衰减中消失。层归一化则把每个样本的特征拉回统一尺度，让训练更稳定。

- 为什么用 Pre-LN（先归一化再进子层）？它把捷径上的梯度通道保持得很干净，所以深度网络用 Pre-LN 更容易训起来，这也是现在主流大模型的做法。
- 前馈网络中间通常放大 4 倍，给它足够的容量去加工每个位置的信息。

> **读代码看这里**　仔细数 x = x + ... 出现了几次。这两行就是残差，它们才是让深层网络能训练起来的关键。

代码 8-4　Transformer 块（Pre-LN + 残差 + FFN）

```
class Block(nn.Module):
    def __init__(self, d_model, n_heads, d_ff=None, dropout=0.1):
        super().__init__()
        d_ff = d_ff or 4 * d_model
        self.ln1 = nn.LayerNorm(d_model); self.attn = MultiHeadAttention(d_model,
n_heads, dropout)
        self.ln2 = nn.LayerNorm(d_model)
        self.ffn = nn.Sequential(nn.Linear(d_model, d_ff), nn.GELU(),
nn.Dropout(dropout),
                                 nn.Linear(d_ff, d_model))
        # 现代 LLM 常用 SwiGLU 替代上面的两层 FFN: FFN(x) = (Swish(xW1) ⊙ xW3) W2

    def forward(self, x, mask=None):
        x = x + self.attn(self.ln1(x), mask)[0]      # 残差 + Pre-LN
        x = x + self.ffn(self.ln2(x))                # 残差 + FFN
        return x
```

参数 / 符号	含义	取值与影响
d_ff = 4*d_model	前馈网络中间层维度	4 倍是原始论文的设定；SwiGLU 变体常用 8/3 倍并配合两个门控矩阵
nn.LayerNorm	层归一化(对每条数据的特征维做归一化，与批次大小无关)	对特征维度归一化，与 batch 无关，因此适合变长序列；这是它取代 BatchNorm 的原因
Pre-LN（先 LN 再进子层）	归一化位置	现代 LLM 标配；原论文的 Post-LN 在深层需要精细 warmup
x + f(...)	残差连接	给梯度一条直通路，是网络能堆到上百层的前提
GELU	激活函数	比 ReLU 平滑；现代 LLM 常用 SwiGLU：FFN(x) = (Swish(xW1) ⊙ xW3) W2

参数 / 符号	含义	取值与影响
dropout=0.1	随机置零比例	预训练时常用 0.0—0.1；现代大模型预训练常设 0，靠数据规模本身抗过拟合

代码逐段解读：

- 块的完整流程：LayerNorm → 多头注意力(并行做多组注意力，每组关注不同的方面) → 残差相加 → LayerNorm → FFN → 残差相加。
- 为什么用 Pre-LN：归一化放在子层之前，使每个子层的输入始终保持稳定分布，梯度回传时不会逐层放大，可堆到上百层。
- 残差的作用：y = f(x) + x 让梯度可以走「+x」这条捷径直达浅层，解决深度网络的退化问题。
- FFN 的作用：注意力负责「混合位置间的信息」，FFN 负责「对每个位置独立做非线性变换」。两者分工明确，缺一不可。
- 参数分布：注意力 $4 \cdot d_model^2$，FFN $2 \cdot d_model \cdot d_ff = 8 \cdot d_model^2$，即 FFN 占了块内约 2/3 的参数。
- 一个 7B 模型的形态参考：32 层、d_model=4096、32 头，每层约 2 亿参数。

> **Pre-LN 还是 Post-LN？** 原始论文用 Post-LN（LayerNorm 在残差之后），但深层训练需要精细的 warmup；现代 LLM（GPT-2 之后）几乎都用 Pre-LN（先归一化再进子层），梯度更稳，可堆到上百层。

④ 位置编码：把顺序信息塞回并行结构

方案	做法	特点
Sinusoidal	PE(pos, 2i) = sin(pos/10000^(2i/d))	原论文方案，无需学习，但外推能力有限
可学习绝对位置	Embedding(嵌入：把词等对象表示成一串数字) 表按位置查表	简单，但上下文长度(模型一次最多能处理多少个词元)被训练长度锁死
RoPE（旋转位置编码(给每个位置加上顺序信息，弥补并行结构丢掉的顺序)）	对 Q/K 做复数域旋转，使内积只依赖相对位置	LLaMA、GLM 等主流方案，外推性好，配合 NTK/ YaRN 可扩展到更长
ALiBi	在注意力打分上减去与距离成正比的偏置	无需位置嵌入，长上下文外推优秀

8.3 为什么能规模化：复杂度与并行性对比

架构	序列长度 n 上的复杂度	能否并行训练	长程依赖路径长度
RNN / LSTM	$O(n \cdot d^2)$，时间步串行	✗ 串行	O(n)

架构	序列长度 n 上的复杂度	能否并行训练	长程依赖路径长度
CNN（一维）	$O(n \cdot k \cdot d^2)$	✓	O(n/k) 需堆很多层
Transformer（全注意力）	$O(n^2 \cdot d)$ 注意力 + $O(n \cdot d^2)$ FFN	✓ 完全并行	O(1) 任意两位置直连
稀疏 / 线性注意力	$O(n \cdot d^2)$ 或 O(n log n·d)	✓	O(1) 或近似

注意力的 n^2 项是长上下文的主要成本：上下文从 4k 扩到 128k，注意力的计算量增长 1024 倍，这是长上下文模型昂贵、需要用 FlashAttention 与稀疏注意力的根本原因。

8.4　预训练、Scaling Law 与涌现能力

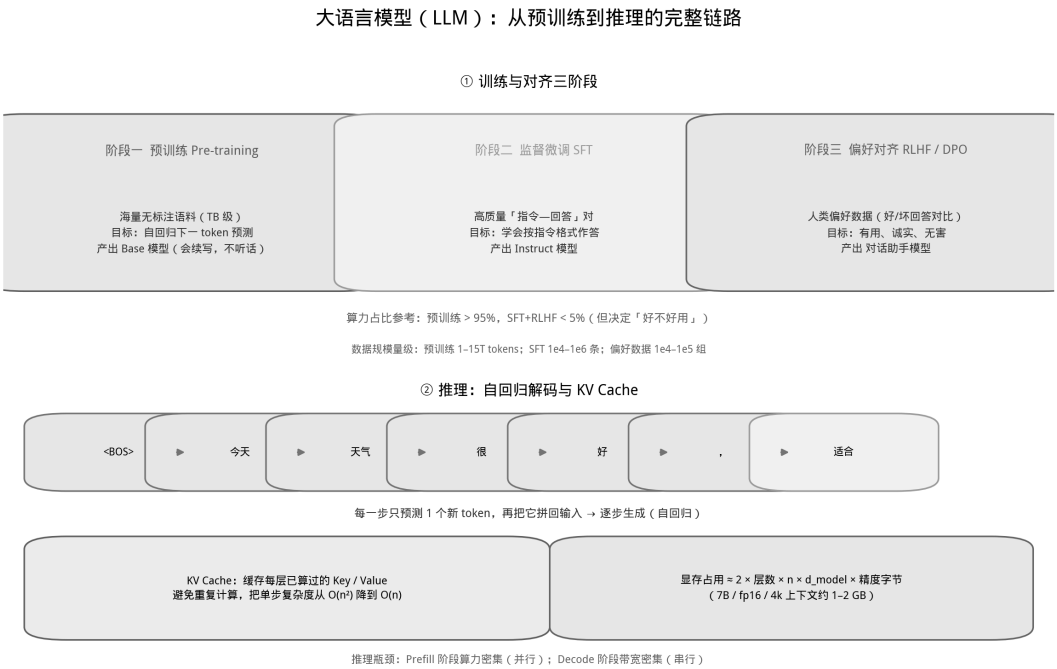


图 8-2　LLM 训练三阶段与自回归推理中的 KV Cache 机制

Scaling Law（规模定律）

Kaplan 等（2020）给出：在算力 C 固定时，模型参数 N、数据量 D 与损失 L 满足幂律关系，$L \propto N^{(-\alpha)}$，且算力分配应偏向增大参数量。Chinchilla（2022）修正为：参数与数据应同比放大，经验配比约为 每 1 个参数配 20 个训练 token。这解释了为什么后来模型纷纷转向「小而多数据」（如 7B 模型训 2T token，而非 175B 训 300B token）。

代码 8-5　训练算力与耗时估算（工程上最常用的公式）

```
# 训练算力的经验估算（Transformer 解码器，含前向 + 反向）
#   C ≈ 6 · N · D       (FLOPs)
```

```
#   N: 非嵌入参数量;  D: 训练 token 数;  6 = 前向 2N + 反向 4N
N, D = 7e9, 2e12
flops = 6 * N * D                          # ≈ 8.4e22 FLOPs
# A100(bf16, 312 TFLOPS) 按 40% MFU 计算:
hours = flops / (312e12 * 0.40) / 3600 / 1000   # 1000 张卡约需 187 小时
print(f"{flops:.2e} FLOPs, 1000×A100 约 {hours:.0f} 小时")
```

参数 / 符号	含义	取值与影响
N	非嵌入参数量	7e9 表示 70 亿；注意嵌入层通常不计入 FLOPs 估算
D	训练 token 总数	2e12 表示 2 万亿；Chinchilla 建议约 20 token/参数，即 7B 配 1400 亿
系数 6	前向 2N + 反向 4N	前向每个参数约 2 次浮点运算，反向约为前向的 2 倍，合计 6N
312e12	A100 bf16 峰值算力 312 TFLOPS	H100 约 990 TFLOPS（bf16 稠密），B200 更高
MFU(模型算力利用率：实际用掉的算力占峰值的比例) 0.40	模型算力利用率	实测能达到的峰值比例；35%—55% 为良好，低于 30% 说明有通信或 IO 瓶颈
hours	预计训练小时数	除以卡数得到单卡口径，再乘卡数即得总时长

代码逐段解读：

- 公式来源：每个 token 的前向约 2N FLOPs（一次乘加算 2），反向约为前向的 2 倍，故训练一个 token 约 6N FLOPs，乘 D 得总数。
- 这个公式不含注意力部分（$n^2 \cdot d$）和重计算（recompute）开销，长上下文训练时实际会更高，通常按 1.1—1.3 倍留余量。
- MFU 的意义：它衡量「你把 GPU 用满了没有」。同样一张卡，MFU 从 30% 提到 50%，等于白赚 60% 的算力。
- 反推用法：给定预算卡时，可以倒推能训多大的模型或多大的数据量——这是做技术选型时最实用的一条公式。
- 举例结论：7B × 2T token ≈ 8.4×10^{22} FLOPs，1000 张 A100 在 40% MFU 下约需 190 小时（约 8 天）。

涌现能力 Emergent Abilities

当模型规模越过某个阈值后，少样本推理、链式思考（CoT(思维链：让模型把推理过程写出来再作答)）、代码生成等能力会「突然」出现，在更小的模型上微调也换不来。看起来，能力并不随规模线性增长，更像是越过某个阈值后触发的相变。实务含义是——如果你的任务需要复杂推理，小模型再怎么微调也可能到不了临界点。

8.5　三阶段训练：预训练 → SFT → 偏好对齐

阶段	数据形态	数据规模量级	损失函数	产出与效果
预训练	无标注网页/书籍/代码	1—15 T tokens	自回归交叉熵（next-token）	Base 模型：会续写，但不听指令
SFT 监督微调	指令—回答对（人工/合成）	1 万—100 万条	仅对回答部分算交叉熵	Instruct 模型：按格式作答
RLHF / DPO(直接偏好优化：不需要单独训练奖励模型的对齐方法)	偏好对比（chosen / rejected）	1 万—10 万组	PPO 的裁剪目标 或 DPO 的偏好损失	对齐模型：有用、诚实、无害

算力分布极不均衡：预训练吃掉 95% 以上的算力，却只决定「模型有多聪明」；SFT 与 RLHF 加起来不到 5%，却决定「模型好不好用」——这也是小团队最值得投入的环节。

8.6　参数高效微调：LoRA 与 QLoRA

全参数微调一个 7B 模型需要约 100 GB 以上显存（参数 + 梯度 + 优化器状态）。LoRA 的思路是：冻结原权重 W_0，只在旁边挂一个低秩分解 $\Delta W = B \cdot A$（r 远小于 d），只训练这两个小矩阵。可训练参数通常降到原模型的 0.1%—1%。

代码 8-6　LoRA 层（约 15 行，含合并推理技巧）

```
import torch.nn as nn

class LoRALinear(nn.Module):
    def __init__(self, base: nn.Linear, r=8, alpha=16, dropout=0.05):
        super().__init__()
        self.base = base
        for p in self.base.parameters():
            p.requires_grad = False                          # 冻结原始权重
        self.r, self.scaling = r, alpha / r
        self.A = nn.Parameter(torch.zeros(r, base.in_features))   # 降维
        self.B = nn.Parameter(torch.zeros(base.out_features, r))  # 升维
        nn.init.kaiming_uniform_(self.A, a=math.sqrt(5))
        self.drop = nn.Dropout(dropout)

    def forward(self, x):
        return self.base(x) + self.scaling * (x @ self.A.T @ self.B.T)
# 推理时可把 BA 直接合并回 W₀ = W₀ + scaling·B·A，零额外延迟
```

```
# QLoRA = 4-bit NF4 量化底座 + LoRA + 双重量化 + 分页优化器（单卡 48GB 可微调 65B）
```

参数 / 符号	含义	取值与影响
base	被冻结的原始线性层	requires_grad=False，反向传播不更新它，显存与算力都省下来
r=8	低秩分解的秩	8—64 常用；r 越大容量越强但越容易过拟合，也越接近全量微调
alpha=16	缩放系数	常用 alpha=2r；实际缩放为 alpha/r，所以 alpha 决定「新知识的注入强度」
scaling	alpha/r	把更新量按比例放大，使不同 r 下的有效学习率保持可比
A / B	降维与升维矩阵	A: (r, in)，B: (out, r)；参数量仅 r·(in+out)，远小于 in·out
kaiming_uniform_(A)	A 的初始化	A 用凯明初始化、B 初始化为 0，保证训练开始时 ΔW=0，不破坏原模型
dropout	LoRA 输入的 dropout	0.05 左右；数据量小的时候可以适当调高

代码逐段解读：

- 低秩假设：微调时权重的更新量 ΔW 虽然维度很大，但其「内在秩」很低，可以用 B·A 近似——这是 LoRA 成立的理论基础。
- 冻结原权重：只对 A、B 求梯度，优化器状态从「全参数」降到「千分之一参数」，这是显存能降一个数量级的根本原因。
- B 初始化为 0：保证第一步时 ΔW = 0，模型行为与原始模型完全一致，从「零扰动」开始学。
- 合并技巧：训练完把 scaling·B·A 直接加到 W_0 上，推理时结构完全不变、零额外延迟——这是 LoRA 相比 Adapter 的关键优势。
- QLoRA：底座用 4-bit NF4 量化 + 双重量化 + 分页优化器，单张 48GB 卡可微调 65B 模型。
- 适用边界：LoRA 适合改「行为与格式」（风格、指令跟随、工具调用），不适合灌入大量新知识——那是 RAG 或全量微调的活。

维度	说明
解决的问题	显存不够 / 多任务需要切换适配器 / 只想改行为不想改知识的场景
核心原理	低秩假设：微调的更新量 ΔW 是低秩的，可用 B·A 近似
时间复杂度	训练显存降至全量微调的 1/3 左右；速度略快（无需更新大模型梯度）
空间复杂度	7B 模型 LoRA（r=16，全模块）可训练参数约 4000 万，优化器状态 < 1 GB
优点	显存省、可插拔切换多任务、不灾难性遗忘、合并后零推理开销
缺点	容量有限，学不进大量新知识（学知识仍需全量微调或 RAG）

维度	说明
典型应用	行业垂域适配、风格/格式定制、Agent 工具调用能力注入
关键参数 / 调参要点	r（8—64）、alpha（常取 2r）、target_modules（q_proj/v_proj 或全线性层）、dropout、学习率（1e-4 量级）

8.7　偏好对齐：RLHF 与 DPO

RLHF 分三步：① 收集人类偏好数据训练一个奖励模型 RM；② 用 PPO 让策略模型最大化 RM 打分，同时用 KL 惩罚约束它别跑得离 SFT 模型太远（防止奖励黑客）。DPO 则跳过了 RM，直接从偏好对导出一个闭式的分类损失：

代码 8-7　DPO 损失（约 6 行，RLHF 的简化替代）

```
import torch.nn.functional as F

def dpo_loss(policy_chosen, policy_rejected, ref_chosen, ref_rejected, beta=0.1):
    """policy_* / ref_* 均为序列对数概率之和（logprob）"""
    pi_logratios = policy_chosen - policy_rejected
    ref_logratios = ref_chosen - ref_rejected
    logits = pi_logratios - ref_logratios           # 相对参考模型的优势
    return -F.logsigmoid(beta * logits).mean()      # 越大越好 → 取负作损失
# 直观理解：让「好回答相对坏回答的对数概率比」比参考模型更高，同时被 beta 约束幅度
```

参数 / 符号	含义	取值与影响
policy_chosen/rejected	策略模型对好/坏回答的对数概率和	通常取回答部分每个 token 对数概率之和，也可做长度归一化
ref_chosen/rejected	参考模型（SFT 模型）的对数概率和	参考模型全程冻结，只做前向，不占优化器显存
pi_logratios	策略模型认为「好回答比坏回答高多少」	越大说明策略越偏好好回答
ref_logratios	参考模型的同样量，作为基准	减去它是为了衡量「相对参考模型进步了多少」
beta=0.1	偏离参考模型的惩罚强度	0.1—0.5；越大越保守（越贴着 SFT 模型），越小越敢改但易崩
logsigmoid	$-\log \sigma(x)$ 的平滑形式	等价于二分类交叉熵，把偏好问题转成一个分类问题

代码逐段解读：

- 核心直觉：让「好回答相对坏回答的对数概率比」比参考模型更高，同时用 beta 约束幅度，防止模型为了刷分而崩坏。

- 为什么不需要奖励模型：DPO 从最优策略的闭式解反推，发现奖励函数可以用策略与参考模型的对数概率比表示，从而跳过 RM 训练。
- 损失形式：$-\log \sigma(\beta \cdot (\text{好} - \text{坏之差}))$，本质是一个二分类损失——判断「这对偏好里哪个更好」。这就是为什么它训练起来像做分类一样稳定。
- 与 PPO 的对比：PPO 需要同时加载策略、参考、奖励、价值四个模型，DPO 只需策略与参考两个，显存和工程复杂度都大幅下降。
- 失效场景：当偏好数据的好/坏回答差异极小，或回答长度分布差异大时，DPO 容易利用长度偏差刷分，需要长度归一化或改用 KTO/ORPO。

方案	需要奖励模型	稳定性	实现复杂度	适用场景
PPO（经典 RLHF）	✓	较敏感，超参多	高（4 个模型同时在显存里）	追求极致对齐效果、有 RL 工程能力
DPO	✗	稳，像做分类	低（2 个模型）	大多数团队的默认选择
GRPO	可用规则奖励替代	较好	中	数学/代码等可自动验证的推理任务（去 critic 网络）
KTO / ORPO	✗	稳	低	只有「好/坏」单条标注而非成对偏好时

配套仿真：搜索「DPO」，可以看到同一对偏好样本在不同 β 下，好回答与坏回答的相对概率比怎样被拉开；β 取得过小，曲线会明显发散。这一件事在文字里很难讲透，看一眼曲线就明白了。

8.8 推理：从 logits 到文本，以及为什么慢

采样策略

策略	做法	效果	常用取值
Greedy	每步取概率最大的 token	确定但易重复、呆板	—
Temperature	logits / T 后再 Softmax	T<1 更集中，T>1 更多样	0.2—0.8（代码/事实类偏低）
Top-k	只在概率前 k 个里采样	截断长尾	k=20—50
Top-p（核采样）	按累积概率 p 截断	自适应候选集，最常用	p=0.9—0.95
Beam Search	保留 B 条最优路径	翻译/摘要更稳，生成易啰嗦	B=3—5

策略	做法	效果	常用取值
重复惩罚	降低已出现 token 的 logits	抑制复读	1.05—1.2

推理的两个阶段与 KV Cache

- Prefill（预填充）：把整段 prompt 一次性前向，算力密集，可充分并行，决定首 token 延迟 TTFT(首字延迟：从发请求到看到第一个字的时间)。
- Decode（解码）：每次只算一个新 token，访存密集、串行，决定每 token 耗时 TPOT(每输出一个词元要花多少时间) 与总吞吐。
- KV Cache 把已算过的 K/V 缓存下来，使单步计算量从 $O(n^2)$ 降到 $O(n)$，代价是显存随上下文线性增长：显存 ≈ 2 × 层数 × n × d_model × 精度字节数。

代码 8-8　自回归生成 + Top-p 采样（LLM 解码的最小骨架）

```
# 自回归生成循环（含 KV Cache 复用）
def generate(model, ids, max_new=100, temperature=0.8, top_p=0.95):
    cache = None
    for _ in range(max_new):
        logits, cache = model(ids if cache is None else ids[:, -1:], kv_cache=cache)
        logits = logits[:, -1, :] / temperature
        probs = torch.softmax(logits, dim=-1)
        sorted_p, idx = torch.sort(probs, descending=True)
        cutoff = sorted_p.cumsum(-1) > top_p                    # 核采样截断
        cutoff[..., 1:] = cutoff[..., :-1].clone(); cutoff[..., 0] = False
        probs = probs.masked_fill(cutoff, 0.0); probs /= probs.sum()
        next_id = torch.multinomial(probs, 1)                   # 按概率采样
        ids = torch.cat([ids, next_id], dim=1)
        if next_id.item() == eos_id:
            break
    return ids
```

参数 / 符号	含义	取值与影响
ids	当前已生成的 token 序列	首轮是完整的 prompt，之后每轮只送最后一个新 token
max_new=100	最多新生成的 token 数	硬上限，防止无限生成；也是成本控制的手段
temperature=0.8	温度 T：logits/T 再 Softmax	T<1 分布更尖锐（更确定），T>1 更平坦（更多样）。代码/事实类用 0.2—0.5，创意类 0.7—1.0
top_p=0.95	核采样的累积概率阈值	0.9—0.95 常用；比 top-k 更自适应，因为候选集大小随分布自动变化
cutoff 掩码逻辑	保留累积概率首次超过 p 之前的部分	必须至少保留 1 个 token，否则分布全 0 会报错

参数 / 符号	含义	取值与影响
multinomial	按概率随机采样一个 token	与 argmax（贪婪）相对：采样带来多样性，也带来不可复现性
eos_id	结束符，命中即停	必须在词表里定义；多轮对话中还要处理特殊停止词

代码逐段解读：

- 第 1 步：前向一次拿到 logits。首轮送整个 prompt（Prefill，算力密集），之后每轮只送新 token（Decode，访存密集）。
- 第 2 步：除以温度。温度趋向 0 时等价于 argmax；趋向无穷时等价于均匀随机。
- 第 3 步：Top-p 截断——把概率从高到低排序求累积和，超过 p 之后的全部置 0，再重新归一化。
- 第 4 步：按处理后的分布采样，拼接到序列末尾，循环。
- 为什么不用 Beam Search 做聊天：beam search 倾向于输出高频、安全但啰嗦的句子；开放式生成用采样更自然。翻译任务反而适合 beam search。
- 工程提示：真实服务里这一步会用 KV Cache + 连续批处理 + 投机解码(让小模型先写草稿，大模型一次性批量验证)来提速，单用户 20—50 tok/s 是体感及格线。

推理加速与显存压缩

技术	做法	收益
KV Cache 量化	K/V 用 int8 或 fp8 存储	长上下文显存减半到 1/4
权重量化	GPTQ / AWQ / SmoothQuant，w4a16 或 w8a8	模型显存降到 1/4，吞吐提升
PagedAttention	把 KV Cache 分页管理（vLLM）	显存碎片减少，吞吐提升 2—4 倍
Continuous Batching	请求级动态批处理，谁生成完谁退出	GPU 利用率从 30% 提到 80%+
FlashAttention	IO 感知的分块注意力，避免物化 n×n 矩阵	显存 $O(n^2)\to O(n)$，速度提升数倍
投机解码	小模型先写草稿，大模型批量验证	解码速度提升 1.5—3 倍
MoE 稀疏激活(每次只用模型的一小部分参数参与计算)	每层只激活少数专家	同等算力下参数量可扩大十倍

8.9　RAG 与 Agent：给 LLM 装外挂

RAG 检索增强生成

LLM 的知识被冻结在权重里，会过期也会编造。RAG 的做法是：把文档切块 → 用 Embedding 模型编码成向量 → 存进向量库 → 提问时把问题和文档块一起编码做近似最近邻检索（HNSW/IVF-PQ）→ 把 top-k 片段拼进 prompt 再生成。它解决三件事：知识时效、事实依据可溯源、避免全量微调的成本。

环节	关键选择	经验值
切分	按语义/固定长度 + 重叠	chunk 300—800 token，重叠 10%—20%
向量化	Embedding 模型维度与语言支持	768—1536 维，注意中英跨语言检索需专用模型
检索	向量检索 + BM25 混合，重排（rerank）	先召回 50—100，再用 cross-encoder 重排到 3—5
生成	引用标注、拒答策略	要求模型必须给出来源，无依据时明确说不知道

Agent：LLM + 工具 + 记忆 + 循环

把 LLM 从「回答者」变成「执行者」：模型输出结构化的工具调用（JSON / function call），系统执行后把结果回灌进上下文，循环直到完成任务。核心技术点是上下文工程——如何组织系统提示、工具描述、历史轨迹和错误信息，决定了 Agent 的成败，而不是模型参数本身。

> **一个务实的判断**　能用 RAG 解决的（知识缺失），不要用微调；能用提示词 + 工具解决的（流程固定），不要上 Agent；Agent 的成本与不可控性都比前两者高一个数量级，只在任务确实需要动态规划时才用。

8.10　LLM 的优缺点与应用地图

维度	内容
优点	通用性强（一套模型覆盖翻译/摘要/代码/推理）；零样本与少样本可用；自然语言的接口极大降低了使用门槛；能力随规模持续提升；可与工具、检索、代码执行组合成系统
缺点	会幻觉（一本正经地编造事实）；知识有截止时间且难更新；推理成本高、延迟大；不可解释、难以保证一致性；输出不可复现（采样随机）；有偏见与安全风险；长上下文存在「中间遗忘」现象
擅长	开放生成、语义理解与改写、代码辅助、信息抽取与结构化、多轮对话、创意与头脑风暴、跨语言任务

维度	内容
不擅长	精确数值计算（应交给计算器/代码）、需要 100% 确定性的决策、超长文档的逐条精算、严格格式的高频批量任务（小模型/规则更划算）
典型应用	智能客服与问答、代码助手、文档摘要与合同审查、内容创作、数据分析助手（Text2SQL）、教育辅导、Agent 自动化、RAG 知识库

8.11 LLM 性能参数速查（工程落地必看）

参数	含义	典型取值 / 经验公式
参数量 N	决定模型容量与显存基线	0.5B / 7B / 14B / 32B / 70B / 数百 B（MoE 另看激活参数）
上下文长度 n	单次能处理的 token 上限	4k / 32k / 128k / 1M；注意力成本随 n^2 增长
推理显存	权重 + KV Cache + 激活	权重 ≈ N × 精度字节（fp16=2, int8=1, int4=0.5）；7B fp16 ≈ 14 GB；KV ≈ 2×层数×n×d_model×字节
训练显存	权重 + 梯度 + 优化器 + 激活	混合精度下约 16—20 × N 字节（AdamW 状态占大头）；7B 全量微调 ≈ 120 GB 级
训练算力	$C \approx 6 \cdot N \cdot D$	7B × 2T tokens ≈ 8.4×10^{22} FLOPs
MFU	模型算力利用率	训练 35%—55% 为良好；低于 30% 说明通信或 IO 有瓶颈
TTFT	首 token 延迟	Prefill 主导，短 prompt 目标 < 500 ms
TPOT	每输出 token 耗时	Decode 主导，目标 20—50 ms/token（约 20—50 tok/s 单用户）
吞吐	系统级 tokens/s	靠 continuous batching 提升；批越大吞吐越高、单请求延迟越差
精度	数值格式	训练 bf16/fp8；推理 fp16、int8、int4（GPTQ/AWQ），int4 通常损失 1%—3% 指标
成本	每百万 token 价格	自部署需按 GPU 时折算：吞吐 × 利用率 ÷ 卡时成本

> **选模型的一条实用经验** 先问「任务的推理深度」：简单抽取/分类用 7B 足够；多步推理、代码、复杂指令跟随建议 32B 起步或直接用顶级商用 API。再问「延迟预算」：交互式场景单用户 20 tok/s 是体感及格线，等价于每 token 50 ms。最后才是成本——把这三件事排完，可选范围通常只剩两三个模型。

8.12 本章常见错误与练习

大模型工程里的错误，一半来自对不确定性的误判，一半来自对成本的忽视。

一是把模型当数据库。事实类问题应当用检索（RAG）补足，而不是指望模型记住；模型内部的知识既不可枚举也无法更新。

二是用「感觉」评估效果。没有固定的评测集与评分标准，提示词改动就无从比较。任何一次上线前，都要有一组稳定的、能自动跑分的评测样本。

三是忽略上下文成本与延迟。上下文长度的计算与注意力是平方关系（或线性关系，取决于实现），把十万字文档整篇塞进去既慢又贵，要先检索、再截断、再排序。

四是没有设置超时与降级。模型调用失败、返回空、返回不合格式的 JSON 都是常态，工程代码必须有兜底。

五是提示词里给的信息自相矛盾。模型会跟着你的措辞走，前后冲突的指令会让输出随机摇摆。

六是忽视 tokenizer 的细节。中文、代码与英文的单位 token 信息量不同，按字符数估算 token 数会差出一大截。

七是把微调当成万能解。多数场景下，提示词工程 + 检索 + 少量示例的效果，优于一次成本高昂的微调。

三道练习：第一题，为一个业务问题搭一条最小可用的 RAG 链路，并对比「有检索」与「无检索」的答案质量。第二题，构造 20 条评测样本，写一个自动打分脚本，用它在两次提示词改动之间做对比。第三题，测量同一段文本在不同分词器下的 token 数，估算两者的成本差异。

8.13　多头注意力、KV Cache 与采样策略：三个必须亲手算一遍的细节

本章前面几节讲的是 Transformer 和大模型的整体框架。这一节挑出三个最容易被「一句话带过」、但在工程上决定成败的细节：多头注意力为什么不是一个头放大一点；推理时 KV Cache 到底省了多少、又占了多少显存；以及模型最后一步「从概率表里抽签」的几种玩法。这三件事分别对应训练、推理和输出三个阶段，读完这一节，大模型的整条链路就闭合了。

8.13.1　多头注意力：为什么要把向量切成几份

单个注意力头只能表达一种关系。给定一句话，一个头算出来的权重矩阵只说明「谁和谁相关」这一件事。但语言里的关系是多层的：有的依赖是主谓关系，有的是修饰关系，有的是标点与句子的关系。多头注意力的做法很直接：把 d 维的表示切成 h 份，每份维度是 d/h，每份配一套自己的 W_Q、W_K、W_V，各自独立算一次注意力，最后把结果拼回去，再过一层输出投影 W_O。

这里有两个关键点。第一，切分是在「投影之后」做的，不是把原始输入切一半给头 A、一半给头 B。每个头先各自把整句话投影一遍，再取自己负责的那一段维度——所以每个头看到的信息是完整句子的不同「视角」。第二，参数量基本不变：单头用 d × d 的投影矩阵，h 个头各用 d × (d/h) 的矩阵，合起来还是 d × d。

注意力本身的计算是缩放点积：先用 Q 和 K 的点积衡量「这个词该看那个词多少」，除以 √d_k 之后做 softmax 变成一组和为 1 的权重，再用这些权重对 V 做加权平均。

```
def attention(Q, K, V):
    """Q: n×dk，K: n×dk，V: n×dv —— 返回 n×dv"""
```

```
    dk = len(Q[0])
    scale = dk ** 0.5                # 除以 sqrt(dk)：防止 softmax 饱和
    out = []
    for i in range(len(Q)):
        scores = [sum(Q[i][k] * K[j][k] for k in range(dk)) / scale
                  for j in range(len(K))]
        w = softmax(scores)          # 每行加起来等于 1
        out.append([sum(w[j] * V[j][k] for j in range(len(V)))
                    for k in range(len(V[0]))])
    return out
```

那个 √d_k 值得单独说一句。点积是 d_k 个数相加，d_k 越大，结果的绝对值越大（方差随 d_k 线性增长）。而 softmax 遇到很大的输入会饱和：最大值接近 1，其余接近 0，梯度几乎为零。所以必须先除以 √d_k 把方差拉回 O(1)。论文原文写得很直白：we suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients。

多头带来的代价在长文本上才显现：注意力矩阵是 n × n 的，序列长度翻倍，训练时的计算与显存都是 4 倍。这也是半个业界都在研究「稀疏注意力」「线性注意力」的原因。

项目	单头注意力	多头注意力（h 个头）
表达能力	只能表达一种关系	h 种关系并行，可以学到分工
参数量	d^2（Q/K/V 各一套）	基本不变，仍是 d^2 级别
矩阵形状	n × n	h 个 n × n
主要开销	$O(n^2 d)$	$O(n^2 d)$，但可以并行
典型配置	教学示例	h = 32 ~ 128，d/h = 64 ~ 128

8.13.2 KV Cache：推理时最值钱的一块显存

训练时整段文本是一次性并行喂进去的，注意力一次算完。但推理（生成）不一样：模型一次只吐一个词，写完第 t 个词才能写第 t + 1 个。如果每一步都把前面所有 token 重新算一遍 K 和 V，那么生成 n 个词的总计算量是 $O(n^2)$；序列越长，越慢得离谱。

KV Cache 的全部思想就是：K 和 V 是「历史累积」的量，算过就不用再算，缓存起来下次直接用。为什么 Q 不缓存？因为注意力等于 $\text{softmax}(QK^T)V$，生成第 t 个位置时 Q 只有一行，是新算的；而 K、V 是所有历史位置的集合，可以复用。加上缓存之后，单步计算量从 O(n) 降到 O(1)，整段生成的总代价从 $O(n^2)$ 降到约 O(n) 乘常数。

代价是显存。KV Cache 的大小有一个必须背下来的公式：

KV Cache 显存 ≈ 2 × 层数 × 序列长度 × 隐藏维度 × 每参数字节数

一个具体例子：32 层、隐藏维度 4096、fp16（每参数 2 字节）的模型，在 32k 上下文时，KV Cache 约为 2 × 32 × 32000 × 4096 × 2 字节 ≈ 16.8 GB。这已经超过了很多模型权重本身的大小，比推理时的激活值大得多。

正因如此，围绕这块缓存出现了三个方向的优化。MQA（Multi-Query Attention，2019）让所有的头共享同一组 K、V；GQA（Grouped-Query Attention，2023）折中一下，

几个头共享一组，把缓存缩小 4 到 8 倍，是目前主流模型的默认选择。PagedAttention（vLLM，2023）则借鉴操作系统的分页思想，把缓存切成固定大小的小块，按需分配、按块复用，解决了「预留给最长序列」带来的碎片浪费，把显存利用率从不到一半提到九成以上。

方案	做法	KV Cache 相对大小	代表
MHA	每个头各有一套 K、V	1×（基准）	原始 Transformer、GPT-2
MQA	所有头共享一套 K、V	1/h	PaLM、Falcon
GQA	每 g 个头共享一套	g/h	LLaMA-2 70B 之后成为主流
PagedAttention	缓存分块、按需分配	不改变大小，但大幅减少碎片	vLLM

还有一个常在面试里出现的追问：为什么训练时不用 KV Cache？因为训练是「老师带着读」，整段文本一次前向，所有位置的 K、V 同时算出来，本来就没有重复计算；而且训练要反传，缓存会拖住计算图。KV Cache 是纯推理期的优化。

8.13.3　采样策略：大模型每写一个字都在抽签

模型对不同 token 的偏好来自 logits。经过 softmax 之后，我们得到一张「下一个字」的概率表。最后一步是从这张表里抽出（或直接取出）一个 token——这个动作就叫解码（decoding），而怎么抽，就是采样策略。

温度 T 最简单：把 logits 全部除以 T 再做 softmax。T < 1 会让分布更尖（第一名更突出），T > 1 会让分布更平（冷门词也有机会）。T 趋近 0 就等于贪心解码，永远取最大的那个；代价是输出会变得呆板，还容易陷入「的的的的」这样的重复循环。

Top-K 更直接：只保留概率最高的 K 个候选，其余全部归零再重新归一化。它的毛病是 K 是固定的——分布很尖的时候 K = 50 会带进一堆垃圾候选，分布很平的时候 K = 3 又会太单调。

Top-p（也叫核采样）解决了这个矛盾：把候选按概率从大到小排序，逐个累加，一旦累计概率超过 p（常用 0.9）就停下来，只让这些候选参与抽签。分布尖的时候可能只剩一两个候选，分布平的时候候选会自然变多——候选个数是动态决定的，比 Top-K 自适应得多。

```
def sample_next(logits, T=0.7, K=None, P=None):
    probs = softmax([x / T for x in logits])        # ① 温度
    order = sorted(range(len(probs)), key=lambda i: -probs[i])
    keep = set(order)                               # 默认全部保留
    if K is not None:
        keep = set(order[:K])                       # ② Top-K：固定个数
    if P is not None:
        keep, cum = set(), 0.0
        for i in order:                             # ③ Top-p：动态个数
            cum += probs[i]
```

```
            keep.add(i)
            if cum >= P:
                break
    kept = [p if i in keep else 0.0 for i, p in enumerate(probs)]
    s = sum(kept)
    kept = [p / s for p in kept]
    r, acc = random.random(), 0.0
    for i, p in enumerate(kept):
        acc += p
        if r < acc:
            return i                                    # 抽签落在哪一个区间
    return len(kept) - 1
```

工程上还有一类目标不同的手段——重复惩罚（repetition penalty）和存在惩罚（presence penalty）：对已经出现过的 token 把它的 logit 压低一点，缓解「复读机」现象。它们不改变候选集合，只改变候选的权重。

参数	作用	常用值	什么时候调
temperature	控制分布的尖锐程度	0.7（聊天）、0（数学/代码）	输出太死板就调高，太飘就调低
top_k	只保留前 K 个候选	0（关闭）或 20～50	想硬性限制候选数量时
top_p	累计概率到 p 就截断	0.9	比 top_k 更常用，更自适应
presence_penalty	出现过就压分，鼓励新话题	0～2	反复说同一个词时
max_tokens	最多生成多少 token	按任务定	控制成本与延迟

8.13.4 本节常见错误与练习

- 多头注意力里最常见的一个误解是「把向量切成 h 份，每份单独投影」。正确的顺序是先用完整的输入做投影，再把投影后的结果切分给各个头。
- KV Cache 的显存公式经常被漏掉因子 2（K 和 V 各一份），算出来的数字只有实际的一半。另一个常见错误是只按「平均长度」估算——实际部署时要按最长序列乘以并发数来预留。
- 采样时把 Top-K 和 Top-p 同时设成非零值，效果是两者取交集，候选会比预期少很多。通常只开其中一个。
- 温度调到 0 并不是「不采样」，而是等价于取 argmax（贪心解码）。在需要准确性的场景（数学、代码、抽取）这是正确选择；在创意写作里则会显得重复。

三道练习。第一题，手写一遍缩放点积注意力，用 3 个词、4 维向量的例子把每一行权重打印出来，验证每行之和恰好为 1。第二题，在纸上算出 32 层、4096 维、fp16、8k/32k/128k 三种上下文下的 KV Cache 显存，再和 GQA 的版本对比。第三题，用同一张概率表分别做温度 0.2、0.7、1.5 和 Top-p 0.9 的采样，各抽 200 次，统计每个候选被抽中的次数，验证它是否近似等于「概率 × 200」。

配套仿真：「多头注意力」把两个头的注意力热力图并排画出来，能看到不同头对不同的词给出不同的关注度；「KV Cache」把「有缓存 / 无缓存」每一步的计算量画成两张对照图，还能改层数和上下文长度看显存账单怎么涨；「采样策略」把温度、Top-K、Top-p 三条路径分步演示，可以直接看到哪些候选被砍掉、砍掉之后概率如何重新分配；「LoRA 低秩微调」则把「全量微调的参数量」和「LoRA 的参数量」画在一张图上。四个仿真都提供五种语言的完整实现，可以逐行对照本节的手写代码。

8.14　因果掩码：第 t 个词为什么看不见后面

8.1 说语言模型的任务是预测下一个 token。训练时为了省时间，我们不会一个词一个词地喂进去，而是把整句话一次性送进网络，同时算出每个位置上「下一个词是什么」的预测。问题随之而来：第 3 个位置在算自己的输出时，如果能看到第 4 个词，那它等于提前抄到了答案。

在自注意力的计算里，每个位置都要和所有位置算一个相关分数，再按分数加权求和。要挡住未来，就在分数矩阵的「上三角」（行号小于列号的那些位置）填上负无穷，softmax 之后这些位置的权重自然变成 0。这个遮罩叫因果掩码（causal mask），因为它维持的正是「只能看自己和自己的过去」这条因果顺序。

有三个实现细节必须记住，它们也是这一节最容易出错的地方。

第一，掩码要作用在 softmax 之前，而且填的是负无穷，不是 0。在 softmax 里，一个位置填 0 表示它的分数是 0，算出来的概率相当大；只有填 $-\infty$ 才会真正变成 0 权重。把这两者说反，模型会稳定地把未来词当成输入，损失曲线看起来还很正常，但生成时完全是另一回事。

第二，要把因果掩码和填充掩码（padding mask）分开。因果掩码挡的是「未来」，位置由序列长度决定，与数据无关；填充掩码挡的是「为了凑齐 batch 长度而补上的占位符」，位置取决于每条样本的真实长度。两者通常叠加使用，一个是 n×n 的下三角，一个是按样本长度变化的向量。

第三，正因为有了掩码，训练才能一次并行处理整句；而推理时必须一个词一个词地生成。这两条路径算出的结果完全一致（这一点可以逐个位置验证），但训练能把 n 个位置的梯度一次算完，而推理每一步只能产出一个 token。这就是 Transformer 的训练效率远高于 RNN 的关键原因，也是 8.13.2 讲 KV Cache 的前提。

掩码类型	挡住什么	位置由什么决定	典型实现
因果掩码	每个位置右边（未来）的 token	序列长度，与数据无关	分数矩阵上三角填 $-\infty$
填充掩码	batch 内为了对齐而补的占位符	每条样本的真实长度	键方向按长度填 $-\infty$
键掩码	被人为屏蔽的特定位置	任务规则	同填充掩码

代码 8-9　因果掩码：下三角可见，上三角填 $-\infty$

```
import torch
```

```
def causal_mask(n, device="cpu"):
    """n x n 的下三角布尔矩阵，True 表示允许被看到"""
    return torch.tril(torch.ones(n, n, dtype=torch.bool, device=device))

def attention_with_mask(Q, K, V, pad=None):
    d = Q.size(-1)
    scores = Q @ K.transpose(-2, -1) / d ** 0.5       # (B, H, n, n)
    n = scores.size(-1)
    # 未来位置直接置 -inf，softmax 之后权重为 0
    keep = causal_mask(n, scores.device)
    scores = scores.masked_fill(~keep, float("-inf"))
    if pad is not None:                               # 再叠加填充掩码
        scores = scores.masked_fill(pad[:, None, None, :] == 0,
                                    float("-inf"))
    w = torch.softmax(scores, dim=-1)
    return w @ V, w

torch.manual_seed(0)
Q = K = V = torch.randn(1, 1, 4, 8)
out, w = attention_with_mask(Q, K, V)
print(torch.round(w[0, 0], 3))          # 下三角有权重，上三角全是 0
print(w[0, 0].triu(1).abs().sum().item())   # 应为 0.0
```

常见错误 一是掩码加在了 softmax 之后（比如直接乘一个 0/1 矩阵），那样每个位置的权重不会重新归一化，输出幅度会被悄悄缩小。二是把掩码矩阵的行列方向弄反，写成上三角可见，这样每个位置只能看未来，等于完全反了，而且训练时损失照样会下降，很隐蔽。三是把 −inf 写成 −1e9 这类大负数，在混合精度下可能溢出成 −inf 或 NaN，直接溢出成失败更安全。四是 causal mask 与 padding mask 的布尔方向搞混：两边一个是「True 表示可见」，一个是「True 表示有效」，混用时记得统一。

配套仿真：搜索「因果掩码」，可以看到整张注意力分数矩阵随时间被填上，上三角位置被涂成屏蔽色，右侧实时显示每一行剩下的权重和是否等于 1。把序列长度改长，再对照 8.13.1 的多头注意力仿真，能清楚看到掩码是在「所有头之前」一次性生效的。

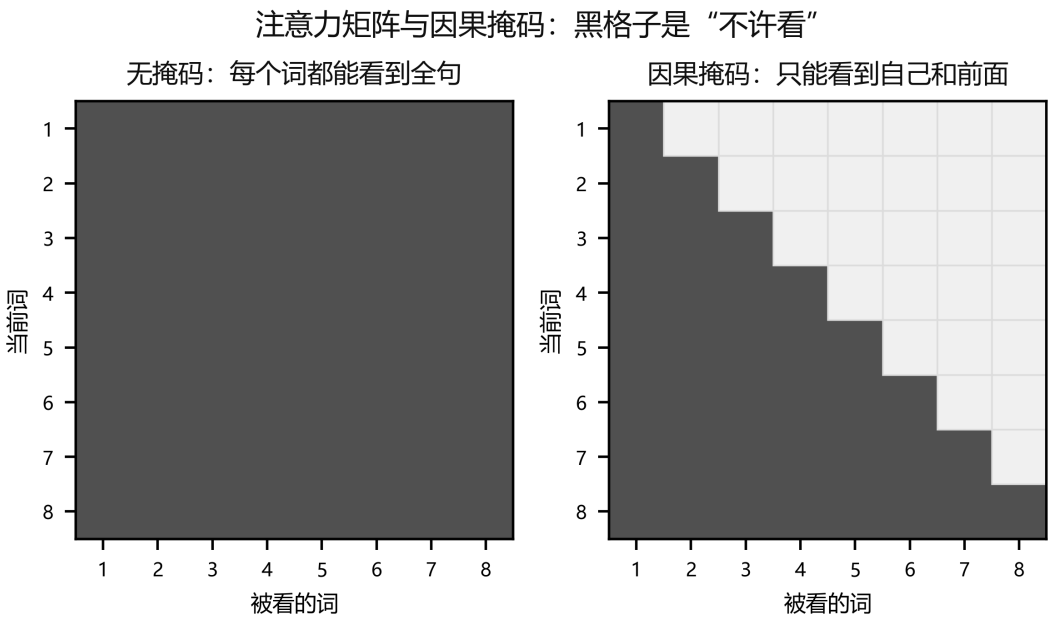


图 8-3　注意力矩阵与因果掩码：黑格子是“不允许看”的位置

8.15　位置编码的演进：从正弦波到 RoPE

8.2 提到自注意力本身看不出顺序，所以需要额外告诉模型「谁在前、谁在后」。原始论文用的是正弦位置编码：把位置变成一组高低频不同的正弦波值，直接加到输入向量上。

现在大多数大模型用的是 RoPE（旋转位置编码）。它不加，而是「转」：把 Query 和 Key 向量按位置角度旋转，两个向量的点积就自然变成了「只与相对距离有关」的值。这一点非常关键：相对距离意味着「模型学会的是相隔多远」，而不是「绝对第几个词」，这让它能够外推到训练时从未见过的长度。

表 8-7　三种位置编码的对比

方法	怎么做	长度外推	典型使用者
可学习绝对位置表	每个位置学一个向量	差（超过最长长度就没定义）	早期 GPT
正弦位置编码	用 sin/cos 公式算出来	一般	原版 Transformer
RoPE（旋转）	按位置角度旋转 Q/K	好，是现在的主流	LLaMA、Qwen、DeepSeek

> **一句话**　位置编码解决的是「并行计算丢了顺序」这个天生缺陷。它是 Transformer 里最容易被忽略、但又真正决定长文本能力的部件。

8.16 手写自注意力：用三个词的句子算一遍

自注意力是 Transformer 的心脏，也是被讲得最玄的一步。其实它的动作只有三个：给每个词做三次变换（得到 Query、Key、Value 三个向量），拿 Query 去和所有 Key 算相似度，再用相似度把 Value 加权平均起来。这一节用一个三个词的句子，把这套动作完整算一遍。

8.16.1 Query / Key / Value 的直观含义

用一个图书馆的比喻：Query 是你想找的「检索词」，Key 是每本书的「标签」，Value 是每本书的「内容」。注意力做的事就是——拿检索词去和所有标签比相似度，相似度高的书拿到的权重就大，最后把各本书的内容按权重混在一起，作为这次检索的结果。

表 8-18 三个矩阵的角色

名字	形状	在干什么	图书馆比喻
Q（Query）	词数 × d	每个位置「我要找什么」	你的检索词
K（Key）	词数 × d	每个位置「我是什么」	每本书的标签
V（Value）	词数 × d	每个位置「我提供什么内容」	每本书的正文

8.16.2 一个三词句子的完整计算

句子是「猫 追 老鼠」。为了手算方便，把每个词表示成 2 维向量（真实模型里是几百到上万维，但机制完全一样）。

```
import numpy as np
np.set_printoptions(precision=3, suppress=True)
X = np.array([[1.0, 0.0],    # 猫
              [0.0, 1.0],    # 追
              [1.0, 1.0]])   # 老鼠
Wq = np.array([[1.0, 0.0], [0.0, 1.0]])
Wk = np.array([[1.0, 0.0], [0.0, 1.0]])
Wv = np.array([[1.0, 0.0], [0.0, 1.0]])
```

十几行讲完的事 自注意力说到底就是三个矩阵乘法加一次归一化。

先把句子里的每个词变成向量，然后让每个词向所有词发问：「你和我有多相关？」这个问题用点积来回答。算出一组相关度之后归一化，再用它给各个词的信息加权求和。

Q、K、V 是同一个输入乘上三个不同的矩阵得到的：Q 代表「我在找什么」，K 代表「我是什么」，V 代表「我能提供什么信息」。用 Q 去和所有 K 点积，就得到了「我该关注谁」的权重。

- 第一步：X 分别乘 Wq、Wk、Wv，得到三组向量。

- 第二步：Q 和 K 的转置相乘，得到两两之间的相关度矩阵。
- 第三步：除以根号 d 把数值拉回合理范围，再对每一行做 softmax 变成权重。
- 第四步：用权重对 V 加权求和，每个位置就得到了一个「融合了全场信息」的新表示。

> **为什么要除以根号 d**　维度一大，点积的数值就会很大，softmax 会变得极度尖锐，梯度几乎消失。除以根号 d 正好把它拉回稳定的量级。

```
Q, K, V = X @ Wq, X @ Wk, X @ Wv
scores = Q @ K.T / np.sqrt(2)          # 相似度，除以 sqrt(d) 防止数值过大
attn   = np.exp(scores - scores.max(1, keepdims=True))
attn  /= attn.sum(1, keepdims=True)    # 每行归一化
out    = attn @ V
print(attn)
```

代码 8-40　自注意力的最小实现（十几行）

表 8-19　三个词之间的注意力权重（每行加起来等于 1）

当前位置（Query）	看「猫」	看「追」	看「老鼠」	它在关注谁
猫	0.401	0.198	0.401	自己和「老鼠」各占四成
追	0.198	0.401	0.401	自己和「老鼠」各占四成
老鼠	0.248	0.248	0.504	最关注自己

这个例子里权重比较平均，因为三个词的向量刚好构成了一个对称的结构。真实训练出来的模型里，权重会非常集中：处理「老鼠」这个词时，可能 80% 的注意力都放在「追」和「猫」上——因为要理解「老鼠被谁追」，必须回头看前面的词。

8.16.3　为什么要除以根号 d

点积的数值会随着维度增加而变大。维度 512 时，点积的量级可能到几十，送进 softmax 之后会变得极其尖锐（一个位置接近 1，其余接近 0），梯度几乎消失。除以 √d 把方差拉回 1 附近，softmax 就处在能正常工作、梯度也不消失的区间。

> **一个常被忽略的点**　缩放因子是 √d 而不是 d，因为两个独立随机向量的点积方差正比于 d，开平方之后标准差正比于 √d。要标准化，除的就是标准差。

8.16.4　从自注意力到多头注意力

单头注意力只能学到一种「关注方式」。多头注意力的做法是：把 d 维的向量切成 8 份（每份 d/8 维），每一份独立走一遍上面的流程，最后把 8 个结果拼回去再过一个线性层。

这样做的收益是：不同的头会自发分工。研究表明，有的头专门跟踪语法主谓关系，有的头专门找前面最近的标点，有的头负责跨句指代。这相当于让模型同时从八个角度看同一个句子。

表 8-20　单头与多头注意力的对比

	单头	多头（以 8 头为例）
计算量	$O(n^2d)$	基本不变（每个头只算 d/8 维）
参数量	$3d^2$	基本不变（每个头 $3(d/8)^2$，八个头相加仍是 $3d^2/8$ 再映射）
表达能力	只能学到一种关注模式	可同时学到多种模式
典型分工	—	语法头、位置头、指代头、标点头……

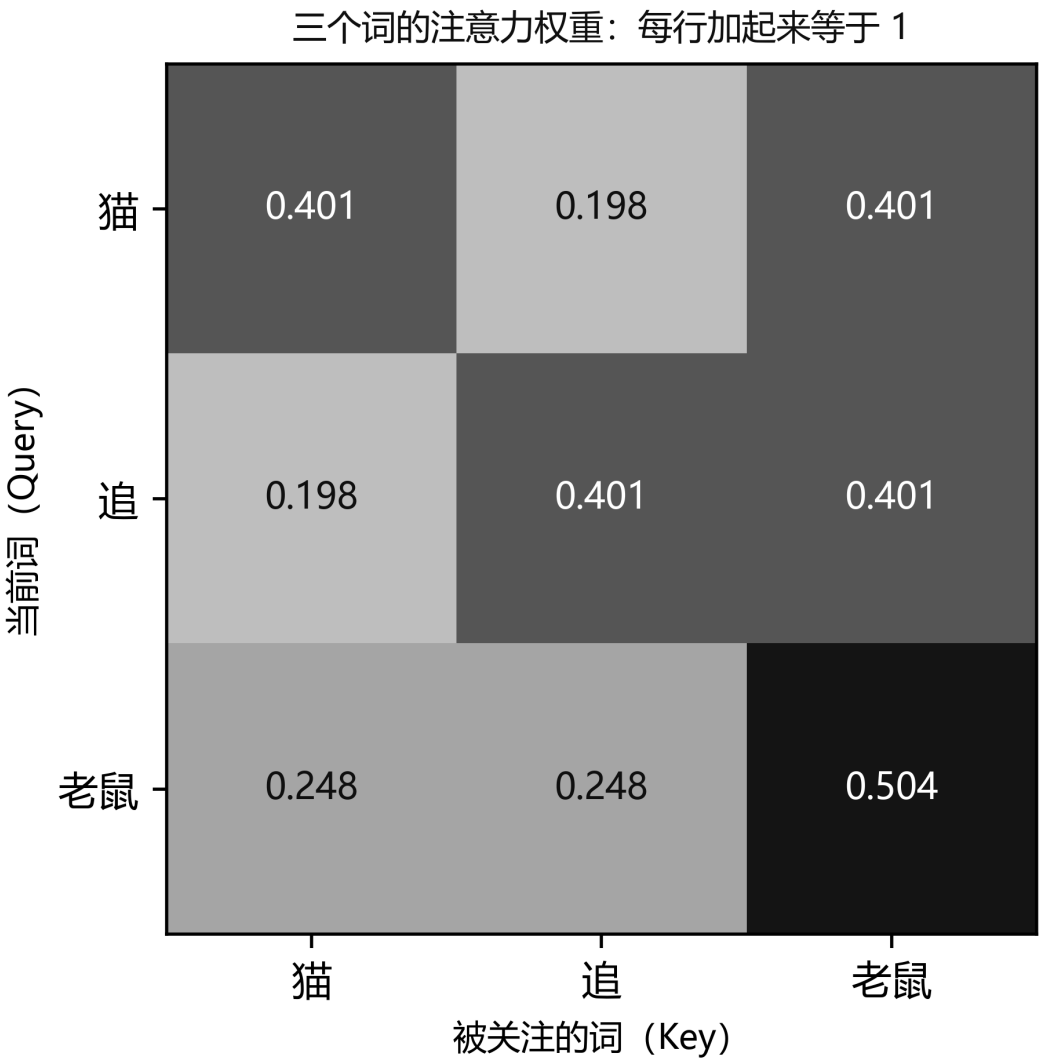


图 8-4　三个词的注意力权重：每一行加起来等于 1，越深表示越关注

8.17　一层 Transformer 到底有多少参数

很多人能背出「7B、13B、70B」这些数字，但说不清它们是怎么来的。这一节把一层 Transformer 的参数逐项算出来，再乘以层数，你就能自己估算任何一个模型的规模。

以常见的配置为例：模型维度 d = 4096，层数 L = 32，注意力头数 h = 32，前馈网络的中间维度用 4d（这是标准做法）。

8.17.1　逐项拆解

表 8-21　一层 Transformer 的参数明细（d = 4096）

模块	包含哪些矩阵	参数量公式	数量（百万）

模块	包含哪些矩阵	参数量公式	数量（百万）
自注意力 Wq	d×d	4096^2	16.8
自注意力 Wk	d×d	4096^2	16.8
自注意力 Wv	d×d	4096^2	16.8
注意力输出 Wo	d×d	4096^2	16.8
前馈第一层	d×4d + 4d	4096×16384 + 16384	67.1
前馈第二层	4d×d + d	16384×4096 + 4096	67.1
两个 LayerNorm	2×2d	2×4096×2	0.016
合计（一层）	—	—	约 201.4

一层约 2.01 亿参数，乘以 32 层，得到约 64.5 亿——正好是一个 7B 量级模型的骨架。再加上词嵌入矩阵（词表大小 5 万 × d = 2.05 亿）和最后的输出层，总量就落在 7B 附近。

8.17.2 三个立即可用的估算结论

> **结论一** 参数量 ≈ 12 × L × d^2。代入 L = 32、d = 4096，得到 $12\times32\times4096^2 \approx 64$ 亿。这个公式在粗估时非常好用，误差通常在 5% 以内。
>
> **结论二** 前馈网络占了整整三分之二的参数。所以想压缩模型，动前馈层（MoE 就是把前馈层换成多个专家）比动注意力层的收益大得多。
>
> **结论三** 显存占用可以用「参数 × 精度字节数 × 系数」粗算。推理时 FP16 约为参数量的 2 倍 GB，训练加上优化器状态约 16 倍。70B 模型训练要上千 GB 显存，这就是为什么需要多卡并行。

表 8-22 常见规模的参数量与显存需求（FP16 推理，含 KV Cache 粗略余量）

规模	层数 L	维度 d	参数量（公式估算）	FP16 推理显存
0.1B	12	768	约 0.85 亿	约 0.3 GB
1B	24	2048	约 12 亿	约 3 GB
7B	32	4096	约 64 亿	约 15 GB
13B	40	5120	约 126 亿	约 27 GB
70B	80	8192	约 645 亿	约 140 GB

8.18　提示工程的工程化：从「会问」到「稳定可用」

前面几节讲的是模型内部。这一节回到使用者视角：怎么把大模型稳定地用在真实业务里。关键的心态转变是——不要把它当搜索引擎，要把它当一位能力很强但需要明确指令、且会偶尔马虎的新同事。

8.18.1　四条基本规则

表 8-23　四个立刻见效的提示写法

规则	反面写法	正面写法	为什么有效
给角色和场景	总结一下这篇文章	你是一位编辑，为中学生读者总结下文，不超过 200 字	约束了语气、读者和长度
给例子	把日期改成 ISO 格式	如「2026 年 10 月 1 日」→「2026-10-01」；现在转换：三月五日	少样本示例是最有效的技巧之一
要求先思考再回答	这道题选什么	先写出推理过程，最后一行给出答案	给模型留出中间计算的空间
明确输出格式	给我一些建议	输出 JSON 数组，每项含 title 与 reason 两个字段	便于程序直接解析

8.18.2　结构化输出与校验

把大模型接进系统时，最怕的就是输出格式飘忽不定。解决思路有三层：提示里给明确的 schema；解析时做容错（去掉多余的 ``` 包裹）；失败时重试并把错误信息回喂给模型。

```
prompt = ("请抽取公司名和年份，输出 JSON，字段为 name 与 year。"
          "只输出 JSON，不要任何解释。\n 文本：华为成立于 1987 年")
```

它解决什么问题　大模型输出的是自由文本，而程序需要一个能直接用的结构。

让模型返回 JSON 是最常见的做法，但模型偶尔会多说一句话、少一个括号，程序一解析就崩。工程上不能假设它每次都听话，必须把「解析失败」当成常态来设计。

- 第一步：把期望的格式写进提示词，并且给一个具体例子，比单纯描述格式有效得多。
- 第二步：解析失败时不要直接报错，而是把失败原因回填给模型，让它自己改一次。
- 第三步：重试设置上限（通常两三次），超过就降级处理，避免无限循环把接口费用烧光。

这就是「模型不可靠，程序要可靠」这条工程原则的具体落地：把不确定性关在重试循环里，对外只暴露稳定的结果。

> **读代码看这里**　把异常信息写回 prompt 再重试的做法，比单纯重复请求有效得多——模型知道错在哪，第二次往往一次就对。

```
for attempt in range(3):
    raw = call_model(prompt)
    try:
        data = json.loads(strip_fence(raw))
        break
    except Exception as e:
        prompt += "\n 上一次输出无法解析（%s），请只输出合法 JSON。" % e
```

代码 8-41　结构化输出 + 失败重试的标准写法

8.18.3　什么时候不该用大模型

这一条比前面所有技巧都重要。大模型擅长的是语言理解和生成、模糊的归类、把非结构化文本变成结构化数据。它不擅长精确计算、实时数据查询、需要严格一致性的判断。

表 8-24　该用与不该用大模型的场景

任务	建议	原因
把一段客服对话归类到 8 个标签之一	用大模型	语言理解是它的强项
从合同里抽取金额和日期	用大模型 + 校验	抽取强，但必须用规则校验数字
算 1234 × 5678	用计算器	语言模型的算术不可靠
查今天的天气	用接口	模型没有实时数据，只会编
根据库存做补货决策	用规则或优化算法	需要可解释、可审计的确定性
给两千条评论打情感标签	大模型批量 + 小模型蒸馏	先少量标注，再蒸馏到小模型降本

> **一句总结**　大模型负责「理解和表达」，确定性的计算、查询、校验交给传统代码。两者配合，才是能上线的系统。

第 9 章　2025—2026 前沿进展：从「拼规模」到「拼效率」

> **读前提示**　本章是第 8 章的延伸，需要先理解注意力、上下文长度、MoE 这几个概念。如果第 8 章还没读，建议先回去看 8.2（Transformer 架构）和 8.11（性能参数）。
>
> **时效声明**　本章内容截至 2026 年 9 月 30 日，素材来自厂商官方模型卡与技术报告、权威媒体报道与会议论文。大模型领域迭代极快，具体榜单与参数请以厂商最新公告为准；本章重点是可复用的技术原理与趋势判断，而非某个模型的分数。

第 8 章讲的是已经沉淀下来的 LLM 技术栈。这一章讲的是它正在发生的变化。如果用一个词概括 2025—2026 年，那就是「效率」——模型能力的增速在放缓，而每单位算力、每单位显存能换来的智能在快速上升。下面按架构、推理、训练、应用四条线梳理。

9.1　主线一：架构从「纯 Transformer」走向「混合」

① MoE 稀疏激活成为默认选项

2025 年之前，开源旗舰还是稠密模型的天下；到 2026 年中，几乎所有严肃的开源大模型都换成了 MoE。原因很直接：MoE 把「知识容量」和「计算成本」解耦了。总参数决定模型装得下多少知识，激活参数决定每个 token 花多少算力——于是可以做到「用 40B 模型的算力，享受 700B 模型的知识深度」。

概念	含义	工程含义
总参数	所有专家的参数之和	决定显存占用（所有专家都要装进显存）
激活参数	每个 token 实际参与计算的参数	决定推理算力与速度，也是定价依据
路由 top-k	每个 token 选 k 个专家	如 6/385、8/256；k 越大越准但越慢
共享专家	每个 token 都必过的一个专家	提供稳定基底表示，缓解路由抖动（Kimi、DeepSeek 等采用）
负载均衡(让各个专家工作量均匀，避免少数专家被挤爆)损失	防止所有 token 挤到少数专家	不加以约束会出现「专家塌缩」，多数专家闲置

- MoE 的代价：显存必须装下全部专家，所以「省算力不省显存」。部署时往往是大显存 + 高带宽的卡更划算。

② 状态空间模型与混合架构

Transformer 的注意力是 $O(n^2)$，KV Cache 随上下文线性膨胀，这让百万级上下文的推理成本高到难以承受。状态空间模型（SSM，如 Mamba 系列）用固定大小的循环状态传递历史，推理复杂度对序列长度是常数级、显存不随上下文增长——代价是「精确回忆」能力弱。

架构	长上下文成本	精确检索能力	2026 年的定位
纯 Transformer	注意力 $O(n^2)$，KV Cache 随 n 线性增长	强（可回头查 KV Cache）	短上下文与检索类任务的默认选择
纯 SSM（Mamba 类）	线性时间、常数显存	弱（固定状态是有损压缩）	长文本摘要、音频、时序等容忍模糊回忆的场景
混合（SSM + 少量 Attention）	大部分层线性，少量层二次	保留（少量注意力层提供锚点）	2026 年生产部署的主流形态

Mamba-3（ICLR 2026）是这条线上的代表工作，三项改进都很具体：

- 梯形离散化：把 Mamba-1/2 的一阶欧拉离散换成二阶梯形近似，等价于引入一个尺寸为 2 的局部卷积，在不增大状态的前提下提升表达力。
- MIMO 形式化：原来的单入单出循环每读取一字节只做约 2.5 次浮点运算，远低于 H100 约 300 次/字节的算力上限，GPU 大部分时间在等数据；改成多入多出后，内部运算从外积变成矩阵乘，把推理从「访存受限」推向「算力受限」。
- 复数动力学：证明带数据依赖 RoPE 的实数 SSM 在数学上等价于复数 SSM，从而恢复状态追踪所需的旋转机制。在奇偶校验（parity）基准上，Mamba-2 接近随机（0.90%），Mamba-3 达到 100%。

代表性混合架构模型	规模	上下文	公开的关键数据
Jamba 1.5（AI21）	约 398B 总参 / 94B 激活	256K	Mamba 与注意力层交错，已规模化部署
Nemotron-H（NVIDIA）	8B / 47B / 56B	长上下文	厂商自测：同规模下推理约为纯 Transformer 的 3 倍
Granite 4.0（IBM）	混合 SSM + 注意力	长上下文	厂商自测：长上下文显存降 70%+、推理约 2 倍
Nemotron 3 Ultra（NVIDIA）	550B 总参 / 55B 激活	1M	20T token 预训练，厂商称吞吐约为同规模纯 Transformer 的 6 倍

> **怎么理解「混合」这个共识** 一句话：注意力负责精确检索与全局锚定，SSM 负责高效的长程压缩，两者互补而非替代。如果你的任务需要「从 100 万字里逐字找出某个密钥」，纯 SSM 会失败；如果是「读完 100 万字写摘要」，SSM 的吞吐优势就非常实在。

③ 稀疏注意力：把 n^2 削下来

既然注意力的瓶颈在 n^2，那就不要每个位置都看所有位置。2026 年的主流做法是「压缩 + 稀疏」组合：

- 压缩：沿序列维或 token 维压缩 KV Cache（如把多个 token 压成一个表征），先降基数。
- 稀疏：用一个轻量的索引器挑出「哪些位置值得算全注意力」，其余跳过。
- 代表：DeepSeek-V4 的混合注意力（压缩稀疏注意力 CSA + 深度压缩注意力 HCA）；GLM-5.2 的 IndexShare，公开数据是在 100 万 token 上下文下把每 token 计算降低约 2.9 倍；美团 LongCat-2.0 的 LSA 稀疏注意力。
- 另一个方向是「潜在注意力」（如 MLA 及其门控变体），把 KV 投影到低维潜在空间再缓存，用少量精度换大幅显存下降。AMD Instella-MoE 采用的门控多头潜在注意力即属此类。

9.2　主线二：推理成为可调节的预算

① 混合思考（快 / 慢模式）

2025 年初 DeepSeek-R1 证明了「纯强化学习可以激发长链推理」，此后推理能力迅速从「独立产品线」变成「所有主流模型的标配」。2026 年的形态是混合推理：同一个模型可以在「快速直觉」与'深思熟虑'之间切换，由用户或路由器决定投入多少思考预算。

模式	适用场景	成本特征
非思考 / 快速模式	闲聊、简单抽取、格式转换	低延迟、低 token 消耗
思考模式（中等预算）	代码调试、多步推理、文档分析	延迟与 token 消耗成倍上升
极限思考模式	数学证明、长程规划、复杂重构	最贵，但上限最高
自适应路由	不确定难度时交给模型/路由器判断	整体成本最优，是 2026 年 API 的默认趋势

② 动态推理预算与提前退出

一个很自然的想法：简单问题不该花 10000 个思考 token。2026 年的工作（如 E-GRM 一类）让模型或系统自己判断「这道题值得想多久」，把算力投到真正需要的地方；训练侧也有把 rollout 预算精准分配的研究思路。工程含义很实在——同样的效果下，动态预算往往能把平均成本砍掉一大半。

③ 解码加速与隐式思考

技术	思路	效果（公开报道）
投机解码	小模型写草稿，大模型批量验证	JetSpec 等报道接近十倍级解码加速；DSpark 报道生成速度提升约 85%
隐式思考 Hidden Decoding	让思考过程在隐空间中完成，不输出冗长思维链(让模型把推理过程一步步写出来，再给出答案)	省掉可见思维链的 token 开销
KV Cache 压缩与分页	量化缓存 + 分页管理（PagedAttention）	长上下文显存成倍下降
连续批处理 + 前缀缓存	动态组批、复用公共前缀	Agent 场景下命中缓存的费用常可大幅下降

9.3 主线三：训练与压缩的效率革命

① 优化器与训练稳定性

- Muon 优化器：被 DeepSeek-V4 等新一代模型采用，公开理由是收敛更快、训练更稳。它属于「用矩阵正交化类更新替代逐元素自适应步长」的一族方法，与 AdamW 并存而非完全取代。
- 流形约束超连接（mHC）：把残差映射约束到双随机矩阵流形（Birkhoff 多面体）上，增强层间信号传播的稳定性。这类工作说明：当模型堆到很深，「让信号不炸掉」本身就成了架构设计的核心议题。
- 动量类改进（如 DeltaMomentum / DeltaAdamW 一类研究）报告在中小规模上可减少约 20%—46% 的训练步数，属于可直接替换的增量改进。

② 蒸馏与多教师在线蒸馏

2026 年后训练环节的关键词从「SFT + RLHF」两件套变成了三件套，多出来的那一件是蒸馏(让小模型模仿大模型的输出，用小成本换接近的效果)。多教师在线策略蒸馏（MOPD）等做法让多个强模型在学生的真实分布上给监督信号，比只学静态数据集更贴近学生自己的错误分布。AMD Instella-MoE、NVIDIA Nemotron 3 等公开材料都提到了这条路线。

③ 量化：从 4-bit 走向 1-bit

如果说 2024 年的关键词是 INT4，2026 年已经出现可实用的极低比特方案。参考公开案例：腾讯混元在 2026 年 9 月开源的 Hy4 轻量版，用自研的稀疏三值量化（每 4 个权重一组量化为 −1/0/+1，平均约 1.25 bit）配合逐层混合精度策略，把原本约 1.5TB 的权重

压到约 214GB（压缩约 86%），并支持主流推理框架。学术侧也有把 Mamba-2 1.3B 压到约 278MB、单 batch GPU 推理提速 20 倍以上的报告。

> **量化选型的经验值**　INT8/FP8 通常几乎无损，是默认选择；INT4（GPTQ/AWQ）常见损失 1%—3% 指标，是性价比甜点；1—2 bit 属于激进压缩，必须在你自己的任务上回归测试，不能只看通用榜单。

9.4　2025—2026 代表性模型速览（截至 2026 年 9 月）

模型	时间	架构要点	规模 / 上下文	值得注意的点
DeepSeek-R1	2025.01	纯 RL 激发长链推理	开源	让「推理模型」成为独立品类，GRPO 走向主流
Qwen3-Max-Thinking	2026.01	参数 + RL + 推理计算三重扩展	旗舰推理	在科学知识、数学、代码等基准刷新纪录
DeepSeek-V4	2026.04	MoE + 混合注意力 CSA/HCA + mHC + Muon	Pro 1.6T / 激活 49B；Flash 285B / 激活 13B；1M 上下文	三种思考模式；官方称在多项开源榜单位居前列
GLM-5.2	2026.06	MoE + IndexShare 稀疏注意力	约 744B / 激活 40B；1M	MIT 许可；公开数据为 1M 上下文下每 token 计算降低约 2.9 倍
LongCat-2.0（美团）	2026.07	MoE + LSA 稀疏注意力 + N-gram 嵌入 + MOPD	1.6T / 激活 48B；1M	MIT 许可，35T token 预训练；面向长文本与 Agent
Kimi K2.6 / K3	2026.04—07	MoE，面向 Agent 与长时程任务	约 1T 总参 / 32B 激活	主打子智能体并行与长时程编码任务
Hy4 preview（腾讯）	2026.08	MoE，门控稀疏注意力	770B / 激活 49B；1M	Apache 2.0；9 月推出 1.25bit 量化轻量版（约 214GB）
Qwen3.8	2026.08	MoE，混合思考	约 2.4T 总参	面向自主编程与专业办公任务
DeepSeek-V4.1-Flash	2026.09	因果编码器—解码器非对称架构	552B MoE；输入激活 8B / 输出 16B	官方称 KV Cache 只需上一代的 1/4 HBM、1/8 SSD，原生多模态

> **这张表怎么看**　不要记分数，要看趋势：① 上下文普遍进入 100 万量级；② 激活参数 / 总参数之比普遍降到 1/10 甚至更低；③ 稀疏注意力与 KV 压缩成为长上下文的标配；④ 开源许可（MIT / Apache 2.0）成为头部模型的竞争筹码。

9.5 主线四：从「会聊天」到「能干活」——Agent 的产业化

2026 年 AI 叙事的关键词从「模型」变成了「智能体」。技术上它由三块拼成：长上下文（装得下完整工作现场）+ 稳定的工具调用（连得上真实系统）+ 长时程任务执行（能坚持几十步不跑偏）。

层次	2025 年的形态	2026 年的形态
能力核心	工具调用（function calling）	自主规划 + 子智能体并行 + 长时程执行
协作方式	单智能体	规划 / 执行 / 审查多角色分工，或大规模子智能体并行
工程重点	提示词与工具描述	上下文工程、状态持久化、错误恢复与人工干预接口
评估口径	单轮任务成功率	长链路任务的成功率与稳定性（Demo 三次成功一次 vs 生产三次失败一次）
治理	基本无	智能体身份认证（KYA）、行为审计、行业规范陆续出台

- 国内监管：2026 年 5 月，国家网信办、国家发展改革委、工业和信息化部联合印发《智能体规范应用与创新发展实施意见》，智能体发展进入「有规可依」阶段；智能体身份认证（KYA）相关国标也已启动立项。
- 落地判断：编程类 Agent 最先跑通，因为它同时具备三要素——海量冷启动数据、可自动验证的反馈、用户轨迹可回流。能否复制到其他领域，取决于有没有「可自动验证的反馈」，这是选型时最该问的一句话。

9.6 这轮变化里，哪些是原理层面的，哪些只是工程

变化	性质	对学习者的含义
MoE 稀疏激活	原理（用条件计算解耦容量与算力）	值得掌握：路由、负载均衡、显存—算力权衡
SSM / 混合架构	原理（用状态压缩换取线性复杂度）	值得掌握：理解「有损记忆」的边界在哪
稀疏注意力 / KV 压缩	偏工程（近似 + 系统优化）	知道即可：用现成内核，不必手推
混合思考 / 推理预算	原理（把算力当作可分配的推理资源）	值得掌握：直接决定你的 API 账单
量化到 1—2 bit	偏工程（数值表示与硬件协同）	知道即可：选成熟方案并做回归测试
多智能体编排	偏工程（系统设计与治理）	值得掌握：这是 2026 年最缺人的方向之一

变化	性质	对学习者的含义
优化器与残差约束改进	原理 + 工程	了解即可：用主流训练框架默认配置

9.7　给读者的三条跟进建议

- 第一，把「第 1—7 章」当成不变的底座，把本章当成会过期的快照。原理的半衰期远长于模型榜单。
- 第二，用「激活参数」而不是「总参数」估算成本，用「上下文长度 × 思考预算」而不是「单次调用」估算账单。这两条是 2026 年做技术选型最容易算错的地方。
- 第三，新建系统时优先做「可替换」设计：模型层用标准接口、检索与工具层独立、评估集自建。模型每三个月就换一茬，但评估集和业务数据会一直跟着你。

9.8　本章常见错误与练习

跟进前沿最常见的两个错误，一是把工程优化当原理突破，二是把营销话术当技术结论。

看一个新技术时，先问三个问题：它改变的是「能力上限」还是「单位成本」？它的收益在什么规模下才成立？它是否依赖特定硬件或特定数据？

具体到本章的四条主线：混合架构与稀疏注意力主要降低单位 token 的计算与显存成本，在长上下文场景收益最大；推理预算可调节改变的是「用多少算力换多少正确率」这条曲线，而不是模型本身的知识；量化与蒸馏压缩的是部署成本，需要配合精度评估一起看；Agent 产业化属于系统工程的成熟，价值取决于工具生态与错误恢复能力，而不是模型本身多聪明。

另外要特别警惕三类说法。第一类是只给基准分数的宣传：没有说明评测集、提示词、采样参数与对比基线的分数，基本不可比。第二类是只在特定硬件上成立的加速比，换到别的卡上往往打折。第三类是把「在某个榜单上超过」当成「在实际任务中更好」，榜单与真实业务分布常常差距很大。

跟进建议是固定的三步：先复现最小可用的例子，再在自有数据上做同条件对比，最后才判断它是否值得进入生产。跳过前两步的技术选型，几乎一定会被营销带偏。

三道练习：第一题，为「长上下文推理」这个场景列出一份成本清单，说明稀疏注意力能省在哪一项。第二题，找一份模型发布公告，指出其中哪些结论有明确的实验条件、哪些没有。第三题，用自己的数据对两个不同规模的模型做同条件评测，记录精度差与成本差。

9.9 MoE 到底省了什么：专家混合的账本

从 2024 年起，几乎所有新发布的大模型都在说自己是 MoE（Mixture of Experts，专家混合）。它的核心效果是「参数量很大，但每次只算一小部分」。这一节把这本账算清楚：省的是算力，不是显存；涨的是容量，不是速度。

打个比方：一家综合医院有 32 个科室的医生团队，但你看病时只挂一个科，不需要所有医生一起会诊。医院的总人数（参数）很多，但每次接诊用到的只是其中一小部分（激活参数）。

9.9.1 结构：把一个大前馈层换成若干个小的

回忆第 8.17 节的结论：Transformer 一层里三分之二的参数在前馈网络。MoE 的做法就是把这一层复制成 N 份（每份叫一个「专家」），再配一个「路由器」，让它根据每个 token 的内容挑出最合适的 K 个专家（通常 K = 2）。

> **一句话原理** 把一个大网络拆成很多个专家，每次只叫醒其中几个干活。

传统做法是让一个巨大的前馈层处理所有输入，参数量大，每次也全都要算。MoE 换个思路：把这一层拆成 8 个（或更多）独立的专家，再训练一个小路由器，看当前这个 token 该交给哪几个专家。

- 第一步：路由器给每个专家打一个分，表示「这个 token 交给你合适吗」。
- 第二步：只挑分数最高的前 K 个专家（常见 K = 1 或 2）。
- 第三步：把这 K 个专家的输出按分数加权求和，作为这一层的输出。

这样做的收益是：总参数量可以做到很大（知识容量大），但每个 token 实际参与计算的只有一小部分（计算量小）。这就是「用参数量换质量，但不让计算量同比例上涨」。

> **工程上的难点** 挑专家必须尽量均匀，否则大部分 token 都涌向同几个专家，既浪费算力又训练不充分，所以实际实现里都有负载均衡的约束。

```
def moe_forward(x, experts, router, top_k=2):
    logits = router(x)                          # 每个专家的得分
    idx = logits.topk(top_k).indices            # 只挑前 K 个
    w = softmax(logits[idx])                    # 它们的权重
    out = sum(wi * experts[i](x) for wi, i in zip(w, idx))
    return out
```

代码 9-30 MoE 的核心：分组打分、挑前 K 个、加权求和

9.9.2 把账算清楚

假设有 8 个专家，每个专家的参数量和原来那个前馈层一样（都是 67 亿参数）。传统模型这一层是 67 亿，MoE 这一层是 8×67 = 536 亿。但每次前向只激活 2 个专家，也就是 134 亿。

表 9-6　稠密模型与 MoE 模型的账本对比

项目	稠密模型	MoE（8 专家，激活 2 个）
总参数量	67 亿	536 亿
每次前向激活的参数	67 亿	134 亿
计算量（FLOPs）	基准 1.0×	约 2.0×（激活比例的倍数）
显存占用	67 亿参数	536 亿参数（必须全部装下）
模型容量	基准	约 8 倍（可容纳的知识更多）
通信开销	无	需要把 token 发到对应专家所在的卡，开销显著

这张表揭穿了一个常见误解：MoE 并不是「更省显存」。恰恰相反，它要求把所有专家都加载到显存里，对显存的要求比同激活量的稠密模型高得多。它省的是计算量——同样多的参数量，MoE 每次只算一小部分。

> **一句话总结**　MoE 用显存和通信开销，换来了「参数量大、计算量小」。所以它适合显存充足、算力受限的场景，也正是推理服务商最喜欢的结构。

9.9.3　两个工程难点

第一是负载均衡。如果路由器总把 token 发给同两个专家，其他专家就成了摆设，训练效率大打折扣。解决办法是加一个「负载均衡损失」，惩罚使用不均的情况。

第二是通信。专家分布在不同显卡上时，token 要跨卡发送（这叫 all-to-all 通信）。通信量随着专家数增加而线性增长，很容易成为瓶颈。这也是为什么 MoE 的工程实现比算法本身难得多。

9.10　从 RAG 到 Agent：一个真实系统的搭建过程

这一节把第 8.9 节提到的概念具体化：假设要做一个「内部知识库问答助手」，从最简单的方案开始，一步步演进到能自己调用工具的 Agent，并说明每一步解决了什么问题、引入了什么新问题。

9.10.1　第一步：直接问模型（失败）

把用户的问题直接发给大模型。对于通用问题效果不错，但对于「我们公司今年 Q3 的退款政策是什么」这类私有知识，模型要么说不知道，要么一本正经地编。这是所有 RAG 系统的起点，也是必须跨过的第一道坎。

9.10.2 第二步：加检索（RAG）

核心思路是「开卷考试」：先把公司的文档切块、向量化、存进向量库；用户提问时，把问题也向量化，找出最相似的几块，连同问题一起塞进提示词。模型的任务从「凭记忆回答」变成了「根据材料回答」。

> **它在做什么** 把「查资料」和「问模型」串起来，让模型回答它本来不知道的事。

RAG 的思路很朴素：模型不知道你公司的内部规定，但你可以先把相关文档找出来，连同问题一起塞进提示词，让它「开卷考试」。这样既不花钱重新训练模型，又能让答案带上真实资料。

- 第一步：把文档切成小段，逐段算成向量存进向量库。
- 第二步：把用户的问题也转成向量，在库里找出最接近的若干段。
- 第三步：把这几段原文和问题拼成提示词，交给模型生成答案。

三段式的每一段都有坑：切块太大检索不精准，太小又丢失上下文；只靠向量检索会漏掉关键词匹配；提示词里不要求「只依据资料回答」，模型就会开始编。

> **读代码看这里** 提示词里那句「只根据下面资料回答，资料里没有就说不知道」，是抑制幻觉最有效的一句话，不要省。

```
def rag_answer(question):
    qv = embed(question)
    docs = vector_db.search(qv, top_k=5)          # 检索最相关的 5 段
    ctx = "\n\n".join(d["text"] for d in docs)
    prompt = f"只根据下面的资料回答问题，资料里没有就说不知道。\n\n 资料：\n{ctx}\n\n 问
题：{question}"
    return call_model(prompt)
```

代码 9-31 最小可用的 RAG 流程

表 9-7 RAG 系统的四个环节与常见问题

环节	做什么	常见问题	对策
切块 Chunking	把长文档切成能检索的小段	切得太碎丢上下文，切得太大检索不准	按语义段落切，保留 10%–20% 重叠
向量化 Embedding	把文本变成向量	通用模型对专业术语不敏感	用领域数据微调，或加关键词检索做混合
检索 Retrieval	找出最相关的几段	只按向量相似，可能漏掉关键信息	混合检索（向量 + 关键词）+ 重排序
生成 Generation	带着材料回答	材料太多导致模型忽略或混淆	控制材料长度，明确要求引用来源

9.10.3　第三步：加工具调用（Agent）

RAG 只能读，不能做。当用户问「帮我查一下订单 A123 现在到哪了」，靠检索文档是答不出来的，必须去调接口。Agent 就是把模型变成「能决定调用哪个工具的调度器」。

```
tools = [
    {"name": "query_order", "args": {"order_id": "str"}, "desc": "查询订单状态"},
    {"name": "calc_refund", "args": {"amount": "float"}, "desc": "计算退款金额"},
]
```

> **一句话原理**　模型只负责决定做什么，真正动手的是程序。

让模型直接执行操作既不安全也不可靠。更稳的分工是：模型输出一个结构化的「调用意图」（调哪个函数、参数是什么），由程序去校验并执行，再把执行结果喂回给模型，让它决定下一步。

- 第一步：把可用工具的名称、用途、参数格式写进提示词，让模型知道有哪些工具。
- 第二步：模型返回要调的函数和参数，程序先做校验（参数类型、权限、数值范围）。
- 第三步：程序执行后把结果回填给模型，循环直到模型给出最终回答。

这条分工线也是安全边界：模型无法绕过程序直接做危险操作，密码、权限、额度这些控制都能在校验那一步拦住。

> **易错点**　校验不能省。模型可能给出不存在的函数名，或者参数类型完全不对——把模型输出当成用户输入来处理，才是正确的姿势。

```
plan = call_model(question, tools=tools)   # 模型决定调哪个工具、传什么参数
if plan.tool == "query_order":
    result = query_order(plan.args["order_id"])   # 由真实代码执行
    answer = call_model(question, context=result)
```

代码 9-32　工具调用：模型决策，程序执行

> **最关键的边界**　工具调用的参数必须由程序校验，绝不能直接信任模型给的数字。金额、ID、权限相关的参数尤其如此——这不是技术问题，是安全问题。

9.10.4　从能跑到能用：还差的四件事

表 9-8　Agent 系统上线前需要补的四件事

缺口	具体表现	做法
评测	改了提示词不知道是不是变好了	建一个 100–200 条的小测试集，每次改动都跑
成本	一次问答调用五次模型，费用失控	缓存、小模型分流、限制工具调用轮数

缺口	具体表现	做法
延迟	用户等 20 秒才出第一个字	流式输出、并行检索、把检索结果先展示
安全	模型被诱导执行危险操作	工具白名单、参数校验、金额与权限二次确认

最后一句实在话：把一个 Agent 做到「演示能看」大约需要三天，做到「用户敢用」大约需要三个月。中间的差距基本全在上面这四件事上，而不在模型本身。

9.11 微调一个小参数模型：从数据到上线的完整流程

第 8.6 节讲了 LoRA 是什么，这一节讲真做一遍要经过哪些步骤。很多同学第一次做微调，一上来就写训练脚本，结果卡在数据格式上三天——所以这一节把顺序重新排列了一遍：先用一天把数据搞对，再花一小时训练。

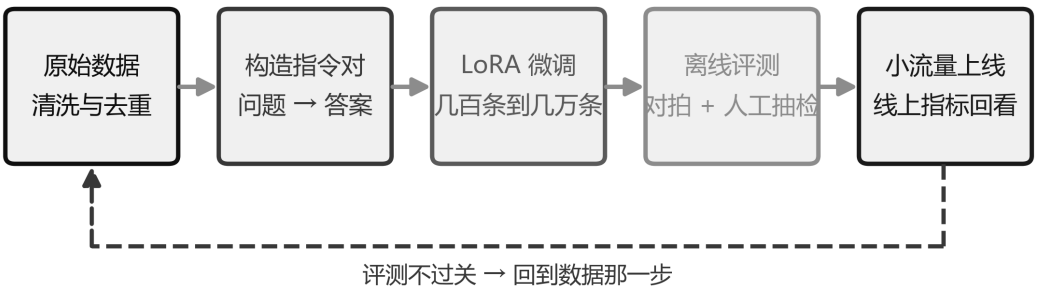


图 9-1 微调到上线的完整流水线：八成时间花在左边两格

9.11.1 第一步：把业务问题翻译成「问题 → 答案」对

微调解决的不是「让模型更聪明」，而是「让模型按你的格式、你的口径说话」。所以第一个动作不是选模型，而是坐下来写 200 条「问题 → 标准答案」。这 200 条是你对这个任务理解的结晶，比任何超参数都重要。

整理数据时有四条硬性要求：问题要覆盖真实分布（不要只写你喜欢问的）、答案口径必须统一（同一类问题用同一种格式）、长度要接近真实使用场景、不确定的样本宁可删掉也不要猜着标。

代码 9-33　构造微调数据的推荐格式（JSONL，一行一条）

```
{"messages": [{"role": "system", "content": "你是设备巡检助手，只回答与设备状态相关的问题。"},
              {"role": "user", "content": "3 号泵振动值 4.2 mm/s 正常吗？"},
              {"role": "assistant", "content": "偏高。该型号正常范围是 0.5~3.5 mm/s，建议检查轴承与联轴器对中。"}]}
{"messages": [{"role": "system", "content": "你是设备巡检助手，只回答与设备状态相关的问题。"},
              {"role": "user", "content": "2 号风机温度 65 度。"},
              {"role": "assistant", "content": "正常。该型号允许范围上限为 75 度，当前处于安全区间。"}]}
```

9.11.2　第二步：划分数据集，先建一个「假模型」

这一步是被跳过最多的一步。正确顺序是：先不要训练任何模型，直接拿基座模型对着你的测试集跑一遍，把结果存下来。这份结果就是你的基准线。没有基准线，后面所有「效果提升」的说法都站不住脚。

数据按 8:1:1 切成训练集、验证集、测试集。训练集用于更新参数，验证集用于决定什么时候停，测试集只在最后看一次。测试集看得多了、改得多了，它就不再是测试集了。

9.11.3　第三步：LoRA 微调，先跑通再调参

第一次跑不要追求效果，只追求「能跑完」。把 LoRA 的秩设得很小（r = 8），学习率设成 1e-4，训练两轮，看损失有没有正常下降、显存有没有爆掉。跑通之后再逐步加数据、加轮数。

代码 9-34　一次典型的 LoRA 训练配置（关键参数都标了理由）

```
config = dict(
    base_model     = "小型开源基座模型",
    lora_r         = 8,        # 秩：越大越能装新知识，也越容易过拟合；先小后大
    lora_alpha     = 16,       # 一般取 2*r
    lora_dropout   = 0.05,     # 数据少于 1000 条时必开
    target_modules = ["q_proj", "v_proj"],   # 只挂在注意力上，先别动全连接
    lr             = 1e-4,     # LoRA 通常比全量微调大一个数量级
    epochs         = 3,        # 从 3 开始，看验证集曲线再加
    batch_size     = 4,
    max_len        = 1024,
    eval_steps     = 50,       # 每 50 步看一次验证集，早停靠它
)
```

9.11.4　第四步：失败模式对照表

微调失败的绝大多数情况，都能在下面这张表里找到对应的一行。遇到问题先来这里查，比盲目调参快得多。

表 9-9　微调常见失败模式与处理办法

现象	最可能的三个原因	先查什么
训练损失不下降	学习率过大；数据格式没被正确解析；标签错位	先用 20 条数据过拟合一次，如果过拟合都做不到，一定是数据和代码的问题
训练损失很低，但测试集很差	数据量太少；训练轮数过多；训练集与真实场景分布不一致	对比训练集和测试集的回答差异，看是不是学到了表面格式
模型开始胡说八道	学习率过大破坏了原有能力；训练数据里有错误答案	把学习率降到原来的十分之一，再抽检 50 条训练数据
只会背训练集里的答案	轮数太多；LoRA 秩太大；缺少 dropout	减少轮数，加 dropout，补更多样的数据
显存不够	序列太长；批太大；没有开启梯度检查点	先把 max_len 减半，再减批大小
回答变短、变得敷衍	训练数据的答案本来就短；损失函数对长答案不友好	检查训练集答案长度分布是否和预期一致

> **一条经验**　微调项目里，数据质量决定上限，超参数只决定你离上限有多近。数据不对的时候，调参调不出任何东西。

9.12　评测：怎么知道模型真的变好了

「感觉好像好了一点」不是一个可以拿去做决策的结论。要做决策，就需要可重复、有数字、能对比的评测。这一节给出一个从轻到重的三级评测方案，任何规模的团队都能用上。

表 9-10　三级评测方案

级别	怎么做	成本	能回答什么问题
第一级：固定用例回归	准备 30～50 条有标准答案的用例，每次改动后全部跑一遍	几十分钟	有没有把原来对的改错（回归）
第二级：自动指标	对可判定的任务用程序判分（格式是否合法、是否包含关键字段、数值是否在范围内）	几分钟，可进 CI	整体通过率有没有提升
第三级：人工盲评 + 对比	把两个版本的输出随机打乱、隐去来源，让人按统一标准打分	几小时	真实用户会更喜欢哪一版

最容易建立也最容易被忽略的是第一级。一个几十条的回归用例集，可以在你每次改提示词、换模型、调参数之后跑一遍，立刻告诉你「这次改动有没有碰坏别的东西」。它的价值不亚于训练本身。

评测里有三个常见陷阱：一是在测试集上调参，等于把考试题当练习册；二是只看平均分、不看分项，某些细分场景可能已经崩了；三是评测集长期不更新，模型上线半年后评测集已经和真实使用场景脱节。

> **能落地的最小方案**　一个 CSV 文件放用例和期望结果，一个脚本跑完打分，一个表格记录每次改动的得分。三个文件就能撑起一个可持续的评测体系。

9.13　幻觉：它为什么会编，怎么治

幻觉是大语言模型最让人头疼的问题：它会在完全不知道的情况下，编出一个看起来非常合理的答案。要治它，先要知道它从哪来。

根本原因在第 8.1 节就埋下了：语言模型训练的目标是「预测下一个词」，不是「说真话」。在它见过的语料里，「某某公司成立于 19xx 年」这样的句式后面总是跟着一个年份，所以当它不知道具体年份时，最省力的做法是生成一个「像年份的年份」。从模型的视角看，这是完全正确的选择——它只是在完成它被训练的任务。

表 9-11　幻觉的三种类型与对应治法

类型	典型表现	为什么发生	怎么治
事实性幻觉	编造不存在的年份、人名、论文、法律条文	参数里根本没有这条知识	接检索（RAG）；要求给出出处；对关键事实做外部校验
忠实性幻觉	给你一段材料，它总结出了材料里没有的内容	长文本中注意力被稀释，生成时自我延续	缩短上下文；要求逐句引用；分步提问
格式性幻觉	要求输出 JSON，结果多打一个括号或少一个字段	解码过程只保证「像」，不保证「合法」	用约束解码；输出后用程序校验，不合法就重试

三种里面，只有第一种需要外挂知识，另外两种都可以在工程层面解决。所以如果你的系统频繁出错，先分类：是它不知道，还是它知道但没按格式说。这两类问题的修法完全不同，混在一起治会事倍功半。

> **最重要的一条**　不要让模型成为事实的唯一来源。任何会被用来做决策的事实，都要有一个可以独立核查的出处。这不是对模型的不信任，而是对系统可靠性的基本要求。

第 10 章　选型指南与性能参数总表

> **读前提示**　这是一张可以随时回来查的速查表，不需要从头读。看不懂的术语可以在第 0 章 0.5 的符号表里找到。

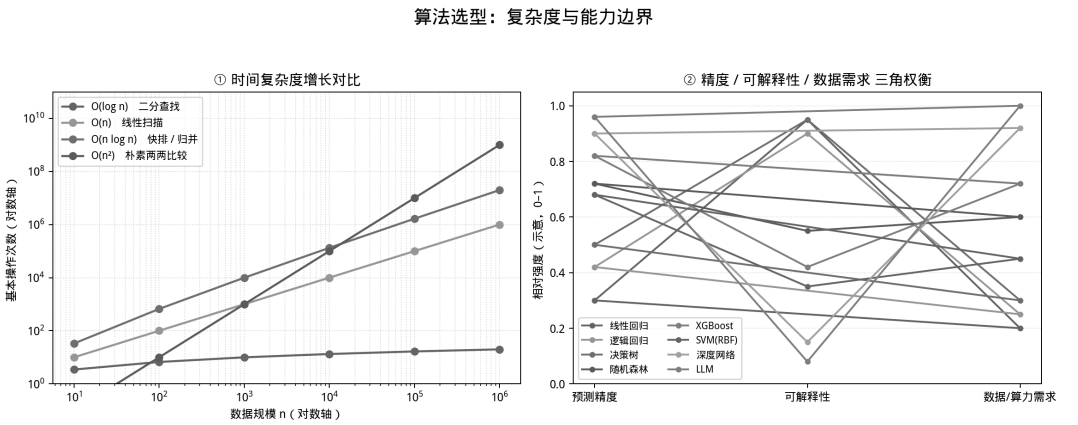


图 10-1　时间复杂度增长对比与「精度—可解释性—数据需求」三角权衡

10.1　复杂度速查表

算法	平均时间	最坏时间	空间	稳定 / 备注
快速排序	O(n log n)	O(n²)	O(log n)	不稳定 / 原地
归并排序	O(n log n)	O(n log n)	O(n)	稳定 / 适合外存
堆排序	O(n log n)	O(n log n)	O(1)	不稳定 / 实时系统兜底
二分查找	O(log n)	O(log n)	O(1)	要求有序 + 随机访问
哈希表	O(1)	O(n)	O(n)	负载因子决定冲突率
Dijkstra（堆）	O((V+E) log V)	同上	O(V)	边权非负
A*	依赖 h 质量	O((V+E) log V)	O(V)	h 可采纳时最优
动态规划（背包）	O(nW)	O(nW)	O(W)	一维滚动数组
线性回归（闭式）	O(d³ + nd²)	同上	O(d²)	d 大时改梯度下降
决策树训练	O(n·d·log n)	同上	O(节点数)	分裂穷举是瓶颈
随机森林	O(T·n log n·√M)	同上	O(T·节点数)	可并行 / T 棵树
KNN（朴素）	O(n·d) 查询	同上	O(n·d)	ANN 索引可到 O(log n)

算法	平均时间	最坏时间	空间	稳定 / 备注
K-Means	O(t·n·k·d)	同上	O(n·d)	k-means++ 初始化
Transformer 注意力	O(n²·d)	同上	O(n²) 无优化	FlashAttention 降到 O(n)
Q-learning（表格）	每步 O(|A|) 表更新	同上	O(|S|·|A|)	状态空间大时表存不下
DQN	每步 O(前向+反向)	同上	回放缓冲 GB 级	离策略，样本可复用
PPO	每轮 O(批量×epoch)	同上	Actor+Critic 双网络	样本效率偏低但稳定

10.2　关键超参数速查

算法族	第一优先调的参	第二梯队	常见错误
树模型（RF）	n_estimators、max_features	max_depth、min_samples_leaf	不限制深度导致过拟合
GBDT/XGBoost	learning_rate + n_estimators（联动）	max_depth、subsample、colsample、gamma	学习率调大却不加轮数
线性/逻辑回归	正则强度 C 或 α	特征分箱、类别权重	忘记标准化导致系数不可比
SVM	C 与 gamma（RBF）	kernel 选择	未标准化；用全量数据跑核 SVM
神经网络	学习率（最重要）	批大小、层数宽度、warmup、weight decay	lr 过大直接发散却不报异常
Transformer/LLM	学习率 + warmup + 批大小	层数/头数、dropout、数据配比	忽略梯度裁剪导致偶发 loss 爆炸
LLM 推理	temperature / top_p	max_new_tokens、重复惩罚	用 greedy 做创意任务
LoRA 微调	r 与学习率	target_modules、alpha	r 过小学不会，过大易过拟合
强化学习	学习率（比监督小 10 倍）+ 奖励设计	γ、GAE λ、PPO ε、熵系数、并行环境数	奖励函数没设计好，训得再久也是错的方向

10.3　算力与显存估算公式（背下来这四条就够）

估算对象	公式	举例
训练算力	$C \approx 6 \cdot N \cdot D$（FLOPs）	7B × 2T token ≈ 8.4×10^{22} FLOPs

估算对象	公式	举例
训练时长	T ≈ C / (GPU_FLOPS × MFU × 卡数)	A100×1000、MFU 40% → 约 190 小时
训练显存	≈ (权重 2N + 梯度 2N + Adam 状态 8N) + 激活	7B 混合精度 AdamW ≈ 84 GB + 激活
推理显存	≈ N × 精度字节 + 2×层数×n×d_model×字节	7B fp16 权重 14 GB；4k 上下文 KV ≈ 1—2 GB

10.4 按场景选算法：决策清单

你的情况	推荐	理由
表格数据、样本 < 10 万、要快上线	随机森林 / LightGBM	免调参、抗噪、CPU 可训、可解释
表格数据、追求精度上限	XGBoost / LightGBM + 调参	精度最高，代价是调参时间
需要向监管/业务解释每一分	逻辑回归 + 分箱 WOE	系数即影响，可审计
图像 / 语音 / 视频	CNN 或 ViT（优先用预训练）	归纳偏置匹配，迁移学习省数据
文本分类、小样本、无 GPU	线性模型 + TF-IDF 或朴素贝叶斯	几分钟出结果，基线可靠
开放生成、对话、复杂理解	LLM（API 或 7B—32B 本地）	通用性最强
知识时效性强 / 事实不能编	LLM + RAG	知识外置，可溯源、易更新
固定流程的自动化执行	提示词 + 工具调用，慎用 Agent	可控性和成本都更优
高 QPS、低延迟、任务简单	小模型 / 规则 / 蒸馏模型	LLM 的延迟与成本扛不住
数据极少（< 1000 条）	预训练模型微调 或 少样本 LLM	从零训练必然过拟合
连续决策、要优化长期收益	PPO / SAC（先模仿学习冷启动）	无标注答案且动作影响未来，只能上 RL
控制类、机械臂、自动驾驶策略	SAC / DDPG（连续动作）	动作空间连续，价值方法用不了

10.5 七个最常见的坑

- 坑一：先划分训练/验证/测试集再做特征工程与标准化，否则数据泄漏会让指标虚高。
- 坑二：用测试集反复调参。调超过两轮，测试集就变成了验证集，指标不再可信。

- 坑三：类别极度不均衡时看 Accuracy。正样本占比 1% 时，全预测负类的准确率是 99%。
- 坑四：把 LLM 当数据库用。事实类知识应当检索，而不是指望它凭记忆答对。
- 坑五：忽视推理成本。训练一次是一次性支出，推理是每天都要付的持续性支出。
- 坑六：以为换更大的模型能解决一切。提示词、数据质量、检索与后处理常常带来更高性价比的提升。
- 坑七：用 RL 解决本可以用监督学习解决的问题。RL 的样本效率与调试成本都比监督学习高一个数量级，只有当「确实要优化长期目标且拿不到标注」时才值得。

10.6　六个真实场景的选型案例

前面五节给的是规则，这一节给的是例子。选型要回答的其实是另一个问题：在当前约束下，哪个代价最小。下面六个场景都来自常见的工程需求，重点看推理过程，而不是结论本身。

场景一，搜索引擎的倒排索引。需要在上亿篇文档里按关键词找到包含它的文档集合，并且要求毫秒级响应。选型推理：关键词数量巨大、查询是等值匹配、需要按相关度排序。结论用「词典 + 倒排表」：词典用哈希表或 B+ 树定位词项（要前缀检索就用 Trie），倒排表按文档编号升序存储，再用跳表做多路归并求交集。这里没有任何一个「算法」是主角，主角是数据结构的组合。

场景二，电商首页的实时推荐。要在 200 毫秒内为每个用户给出几十个候选商品。选型推理：候选池很大、要求低延迟、允许近似。结论是先用向量检索（近似最近邻）把百万级候选缩到几百个，再用轻量模型排序，最后做业务规则过滤。这里没有用最准的模型，用的是能被延迟预算容纳的模型。

场景三，风控的实时决策。要在几十毫秒内判断一笔交易是否可疑。选型推理：极度不平衡、要求可解释、特征以类别与统计量为主、模型必须能解释给业务方。结论优先选择逻辑回归或梯度提升树，而不是深度模型；事后用 SHAP 之类的工具做归因。精度不是唯一目标，可解释性和稳定性同样是硬约束。

场景四，任务调度的优先级队列。要持续接收任务，按截止时间与权重决定先执行谁。选型推理：持续插入、总是取全局最优、允许 O(log n)。结论用堆（优先队列）。如果还要求「能随时修改某个任务」的优先级，就要额外维护一个从任务编号到堆中位置的哈希索引，否则修改操作找不到位置。

场景五，爬虫的 URL 去重。要判断一条 URL 是否已经抓过，URL 总量以十亿计，且允许极小的误判率。选型推理：内存放不下完整的集合、只判断存在性、不要求删除。结论用布隆过滤器：一条 URL 十几个比特，十亿条也就几 GB；代价是极小概率把新 URL 判成旧 URL（对爬虫无害，只是少抓一条）。注意它不能用于「确定不存在」的场景。

场景六，日志系统的 Top-K 统计。要从每秒上百万条日志里找出访问量最大的 20 个 IP。选型推理：流式数据、不能全存、只需要前 K。结论用小顶堆维护大小为 K 的候选集：

内存 O(K)，每条日志 O(log K)，总时间线性。如果允许误差，还可以换成分组计数加定期归并的近似方案，进一步降低内存。

把六个场景抽象一下，选型的通用规律大致有三条。先确定硬约束（延迟、内存、可解释性、精度），硬约束会直接砍掉大部分选项；再把问题拆成「先粗筛、再精排」两段，粗筛用便宜且近似的方法，精排用昂贵但准确的方法，倒排索引、向量检索、推荐系统都是这个结构；最后是「允许近似」这个条件——一旦允许，空间和时间的量级往往能降一到两个数量级，布隆过滤器、近似最近邻、Quantile 草图都属于这一类。

最后提醒一句：上面的结论都不是绝对的。同样的问题，如果数据量从一亿降到十万，最合理的方案可能变成「全部读进内存，用哈希表硬查」。算法选型永远依附于规模与约束，脱离它们的「最佳实践」都是空话。

10.7 按数据量、算力、延迟三条线选方案

前面几节是「按问题选算法」。这一节换一个角度：同一个问题，在不同的资源条件下，正确答案是不一样的。下面这张表把三条约束线放在一起，你可以直接对号入座。

表 10-1 按数据量与算力选方案

数据量	可用算力	推荐路线	理由
几百条	一台笔记本	规则 / 简单统计 / 提示工程	数据量不足以支撑任何模型训练，规则最可靠
几千条	一台笔记本或单卡	逻辑回归、决策树、随机森林	这些小模型在小数据上往往比大模型更稳、更快
几万条	单卡到几卡	梯度提升树 / 小规模神经网络	这一档是传统机器学习性价比最高的区间
几十万条以上	多卡	深度学习 + 迁移学习 / 微调	数据量够了，深度模型才开始体现出优势
百万条以上	多机多卡	预训练或大规模微调	此时瓶颈已经不是算法，而是工程与成本

再叠加第三条约束——延迟。如果系统要求 10 毫秒内出结果（例如高频交易、实时控制），那么再准的大模型也不能用，必须换成能在毫秒级算完的方案，或者用「小模型快速筛选 + 大模型异步复核」的两段式结构。

表 10-2 三条约束的优先级判断

如果……是硬约束	那么……要让位	常见做法
延迟必须低于 10 ms	模型精度	查表、规则、极端梯度提升的小模型
延迟可放宽到 1 s	模型规模	中小模型 + 缓存

如果……是硬约束	那么……要让位	常见做法
显存只有 8 GB	参数量	量化、LoRA、蒸馏到小模型
标注数据极少	模型复杂度	迁移学习、提示工程、少样本学习
钱不是问题、要最好效果	推理成本	大模型 + 检索增强 + 多次采样投票

> **一条判断原则**　先找出那个「不能让步」的约束，其余所有选择都为它服务。工程上失败的方案，绝大多数不是输在精度上，而是输在踩了某条硬约束。

10.8　一份能照着走的项目落地清单

从「选定算法」到「系统能跑」，中间还有很长的路。下面这份清单按时间顺序排列，可以直接当成项目 checklist 用。

表 10-3　算法项目落地清单（按顺序执行）

阶段	要做的事	容易漏掉的
① 定义问题	写清楚输入是什么、输出是什么、算对了算错了分别意味着什么	没有定义「错到什么程度算不能接受」
② 建基线	用最简单的方法（规则、平均值、现成小模型）先跑出数字	跳过基线，后面所有改进都无法量化
③ 准备数据	划分训练 / 验证 / 测试集，检查标签质量与分布	测试集被反复使用，失去意义
④ 选型	按第 10.7 节三条约束筛出两三个候选方案	只挑最先进的，不考虑能不能跑起来
⑤ 最小可行实现	先用一小部分数据把整条链路跑通，不做任何优化	一上来就写完整系统，最后卡在某处整条链跑不通
⑥ 评测	用第 9.12 节的三级评测，记录基线数字	只看平均分，不看分项指标
⑦ 优化	一次只改一处，每次记录结果	同时改五个参数，效果无法归因
⑧ 部署	先小流量上线，把线上指标与离线指标对齐	离线涨、线上跌，且不知道为什么
⑨ 监控	记录延迟、失败率、输入分布漂移	上线当天就跑，出问题没有任何日志
⑩ 迭代	定期回看失败样本，更新评测集与训练数据	评测集半年没更新过

这份清单里最有价值的是第 ② 步和第 ⑤ 步。有了基线，你才知道改进是否真的发生；有了最小可行实现，你才能在早期发现那些「做到一半才发现走不通」的坑。

> **给新手的提醒**　不要追求一次做对。按清单一步步走，每一步都留一个可以回退的版本，比一口气写完再调试要快得多。

第 11 章　学习路线与结语

11.1　三阶段学习路径

阶段	目标	要掌握的算法	检验标准
入门（1—2 月）	能写能调	排序/二分/哈希、DP、线性回归、逻辑回归、决策树	手撕快排；用 sklearn 跑通一个完整建模流程
进阶（3—6 月）	能选型能优化	随机森林、XGBoost、SVM、CNN、RNN、评估与调参	在表格与图像各做一个项目，能说清为什么选它
深入（6 月+）	能做系统	Transformer、预训练与微调、RLHF/DPO、推理优化、RAG/Agent	从零实现一个小型 GPT 并训练；独立部署一个 RAG 服务
专项（可选）	能做决策系统	MDP 与贝尔曼方程、Q-learning/DQN、REINFORCE、PPO、探索策略	在 Gym 环境用 PPO 训通一个任务；理解 RLHF 中 PPO 的损失函数

11.2　一句话总结每个算法家族

- 经典算法：把确定的事情做到最快、最省、可证明。
- 机器学习：在有限数据与可解释性之间找平衡，用偏差—方差权衡做取舍。
- 集成学习：用「多模型平均」降方差、用「逐轮纠错」降偏差，是表格数据的最优解。
- 深度学习：用多层非线性自动学表示，代价是数据与算力。
- 强化学习：在没有标准答案、动作会影响未来的场景里，用试错和长期回报学决策，代价是样本效率与奖励设计。
- Transformer / LLM：用注意力消掉时间上的串行依赖，从而把规模推到极致，换来通用智能的雏形。

11.3　最后的三条判断

算法的价值不在复杂度数字上，而在它是否匹配你的数据形态与约束条件。今天最强的模型往往也是最贵的模型，工程落地的核心是「用最小的模型解决问题」。从 1945 年的归并排序到今天的大模型，算法史的每一次跃迁都来自「换一种看待问题的方式」——把递归变成分治，把串行变成并行，把手工特征变成自动学习，把提示变成搜索。理解这些转折，比记住任何一个公式都重要。

11.4　30 天入门路线：每天 1.5 小时怎么安排

前面十一章把知识讲完了，最后这一节回答一个最实际的问题：「我现在从零开始，具体每天该做什么？」下面这条 30 天路线是按「每天 1.5 小时、周末各多花半天」的强度设计的，适用于完全零基础的读者。

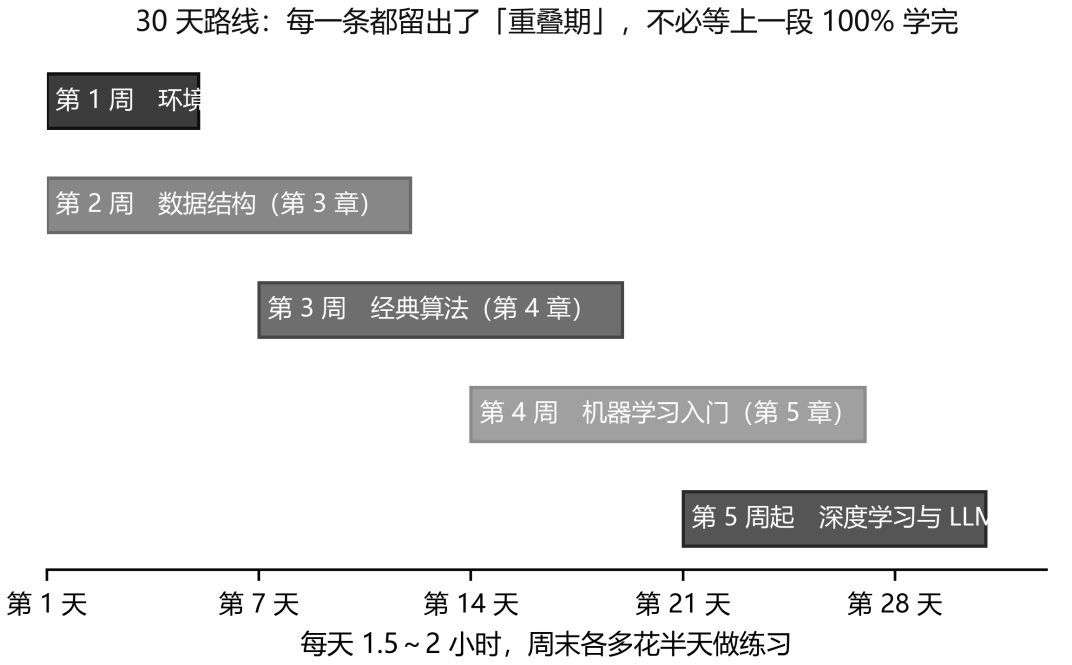


图 11-1　30 天路线的实际排布：每一条都留出了重叠期

请注意这张图里的条不是首尾相接的，而是互相重叠的。这是有意设计的：不必等上一段 100% 学完再开始下一段。第一周的语法不必全会，只要能写出循环和函数，就可以开始读数据结构。

表 11-1　30 天路线逐日安排

时间段	读什么	动手做什么	当天目标
第 1～5 天	第 0 章全部	装好环境，把 0.9 节的第一个程序跑起来	能独立让一段代码跑出结果
第 6～12 天	第 3 章 3.0～3.12	手写动态数组、栈、队列、哈希表各一遍	不看答案能写出这四种结构
第 13～19 天	第 4 章 4.1～4.10	手写快速排序、归并排序、二分查找	能说清它们各自的复杂度是怎么来的
第 20～27 天	第 5 章 5.0～5.10	跑通一次线性回归和一次决策树	看懂损失曲线，知道过拟合长什么样

时间段	读什么	动手做什么	当天目标
第 28～30 天	第 11 章 + 附录 D、F	把前面所有练习重做一遍	形成自己的对照表与排错清单

> **三条执行建议**　一是每天固定时间、固定时长，不要攒到周末补；二是每学完一个结构就关掉书自己写一遍，写不出来才算真学会；三是把卡住的地方记在一个文件里，一周后回头重看，大部分会自己通。

11.5　90 天进阶路线：从会写走到会选

30 天结束时，你应该已经能独立实现常见结构和小算法。接下来 60 天的目标，是从「会写」走到「会改」和「会选」。

表 11-2　第 31～90 天进阶路线

时间段	主题	关键动作	结束时应该能做到
第 31～45 天	第 4 章剩余内容	把动态规划、图算法、回溯各做 5 道题	看到新题能判断它属于哪一类范式
第 46～60 天	第 6 章深度学习	从零实现一次反向传播（6.12 节）	能解释梯度消失、过拟合、学习率各自的作用
第 61～75 天	第 8 章 Transformer 与 LLM	手推一次自注意力（8.16 节）	能算清一层 Transformer 的参数量与显存占用
第 76～85 天	第 7 章强化学习	把 7.15 节的迷宫 Q-learning 跑一遍	理解「试错学习」与「监督学习」的区别
第 86～90 天	第 10 章选型 + 第 9 章前沿	给自己手上的一个真实问题写一份选型报告	能说清为什么选它、代价是什么

这张表里最重要的一件事是第 86～90 天的「写一份选型报告」。它把前面所有零散的知识点串成了一次真实的决策过程：问题是什么、有哪些候选、各自代价是什么、为什么选这个、怎么验证。写完这份报告，你才真正从「学过算法」变成「会用算法」。

11.6　怎么继续往下学：四类资源与三个习惯

表 11-3　四类学习资源的特点与用法

资源类型	适合什么时候用	要注意什么
教材与专著	建立体系、补基础	不要一本科目读到尾，按需查章节
开源代码仓库	看真实工程是怎么写的	先跑通，再读；不要从最难的项目开始

资源类型	适合什么时候用	要注意什么
论文	跟进前沿、理解新方法	按 5.26 节的三遍法读，大部分只读第一遍
社区与讨论	解决具体卡点、了解经验	能自己查到答案的问题不要直接问，先查文档

比资源更重要的是三个习惯。第一，每学一个新方法，立刻写一段最小可运行代码验证它——能跑起来的理解才算理解。第二，把每次踩的坑记在一个长期维护的文件里，按报错类型分类，这就是你自己的排错手册。第三，定期回看你一个月前写的代码，如果看不出问题，说明你这一个月没有进步；如果能一眼看出三处可以改进，说明你在正确的路上。

11.7 结语：写给刚开始的你

这本书从一个很朴素的问题开始：一个完全不懂的人，能不能把算法这件事学明白？十二章写下来，我的答案是能，但需要两样东西——一个足够清晰的顺序，和一个能一直动手的环境。

顺序就是这本书的结构：先补齐最小必要的背景（第 0、1 章），再用时间轴建立全局印象（第 2 章），然后从数据结构这个地基开始（第 3 章），一层层往上走到经典算法、机器学习、深度学习、强化学习和语言模型。每一章都以前面为基础，也都能单独拿出来读。

环境就是这本书配套的仿真实验室。读到这里，如果你还没有打开过它，我建议你现在就去打开一个仿真，把速度调慢，看着数据一步一步变化——那几秒钟的直观感受，往往比读十页文字更有用。

最后回到第 1 章那句话：算法能力分三级——会写、会改、会选。这本书能帮你把前两级走扎实，第三级需要在真实问题里反复练习。希望当你合上这本书的时候，看到一个新问题时，脑子里出现的不是「这题我背过没有」，而是「这属于哪一类、该用哪种方法、代价是什么」。

> **最后一句话**　算法不是一堆要背下来的答案，而是一套可以反复使用的判断方法。方法学会了，工具会换，能力不会。

附录 A　核心术语中英对照表

读书时最容易被卡住的地方，往往是同一个东西有中文名、英文名和缩写三个叫法，而不是概念本身难。下表按主题分组，把全书出现过的关键术语集中列出：左边是中文，中间是英文或缩写，右边是一句话解释。遇到陌生的词，随时翻回这里。

中文	英文 / 缩写	一句话解释
一、复杂度与基础		
问题规模	Input Size (n)	数据量的大小，通常记作 n
时间复杂度	Time Complexity	操作次数随规模增长的量级
空间复杂度	Space Complexity	额外占用内存随规模增长的量级
大 O 记号	Big-O Notation	表示增长率的上界
均摊复杂度	Amortized Complexity	偶尔一次很慢，但摊到每次都很快
最坏情况	Worst Case	输入最不利时的代价
渐进最优	Asymptotically Optimal	增长量级上已经无法更好
稳定性	Stability	值相等的元素排序后相对次序不变
原地算法	In-place Algorithm	只用常数级额外空间的算法
递归深度	Recursion Depth	函数自己调用自己的层数
栈溢出	Stack Overflow	递归太深或用栈过多导致内存耗尽
二、数据结构		
数组	Array	连续内存的定长容器，随机访问 O(1)
动态数组	Dynamic Array	可扩容的数组，Python 的 list
链表	Linked List	节点用指针串联，插入删除 O(1)
哨兵节点	Sentinel Node	不存数据的占位节点，简化边界判断
栈	Stack	后进先出，只在顶端操作

中文	英文 / 缩写	一句话解释
队列	Queue	先进先出，尾进头出
双端队列	Deque	两端都能进出的队列
循环队列	Circular Queue	用取余复用数组空间的队列
哈希表	Hash Table	由键算出位置，均摊 O(1) 查找
哈希函数	Hash Function	把键映射成整数的函数
哈希冲突	Collision	不同键算出同一个位置
链地址法	Separate Chaining	冲突位置挂一条链表
开放寻址法	Open Addressing	冲突时往后找空位
负载因子	Load Factor	已存元素数与槽位数之比
再哈希	Rehashing	扩容后把所有元素重新放一遍
位图	Bitmap	一个比特表示一个状态
布隆过滤器	Bloom Filter	用位图判断元素「可能在或一定不在」
二叉搜索树	Binary Search Tree (BST)	左小右大的二叉树
平衡树	Balanced Tree	高度受控的搜索树
红黑树	Red-Black Tree	工程中最常见的平衡树
B 树 / B+ 树	B-Tree / B+ Tree	一个节点存多个键，适合磁盘与数据库
堆	Heap	只保证堆顶最值的完全二叉树
优先队列	Priority Queue	按优先级出队的队列，通常用堆实现
并查集	Union-Find / DSU	合并集合与判断连通，近似 O(1)
字典树	Trie / Prefix Tree	按字符分层的前缀树
跳表	Skip List	多层索引的有序链表，Redis zset 用它
图	Graph	描述多对多关系的结构
顶点与边	Vertex / Edge	点与点之间的连接
邻接矩阵	Adjacency Matrix	用二维数组存图的连接关系
邻接表	Adjacency List	每个点存一份邻居列表

中文	英文 / 缩写	一句话解释
入度与出度	In-degree / Out-degree	指向自己与从自己出发的边数
有向无环图	DAG	有方向且没有环的图
拓扑排序	Topological Sort	把依赖关系排成线性顺序
连通分量	Connected Component	内部互相可达的点集
最小生成树	Minimum Spanning Tree (MST)	连通所有点且总权最小的边集
LRU 缓存	Least Recently Used Cache	淘汰最久未使用项，哈希表加双向链表
三、算法范式		
分治	Divide and Conquer	拆成同类子问题再合并
动态规划	Dynamic Programming (DP)	把重复子问题的答案存下来复用
记忆化搜索	Memoization	自顶向下带缓存的递归
状态转移方程	State Transition	描述子问题之间如何递推
贪心	Greedy	每步取局部最优，不回头
回溯	Backtracking	枚举所有方案并撤销选择
剪枝	Pruning	提前砍掉不可能的分支
分支限界	Branch and Bound	用上下界剪掉整片搜索空间
二分查找	Binary Search	有序数据上每次砍一半
二分答案	Binary Search on Answer	对答案本身二分，用判定函数检查
双指针	Two Pointers	用两个下标代替两层循环
滑动窗口	Sliding Window	维护一段连续区间，O(n) 求解
前缀和	Prefix Sum	预处理后 O(1) 求区间和
差分数组	Difference Array	O(1) 做区间整体加减
随机化算法	Randomized Algorithm	用随机数获得期望性能
近似算法	Approximation Algorithm	用可控误差换速度
启发式搜索	Heuristic Search	用估计函数引导搜索，如 A*
字符串匹配	String Matching	在长文本中定位模式串
前缀函数	Prefix Function	KMP 的核心，记录最长相等前后缀

中文	英文 / 缩写	一句话解释
四、机器学习		
监督学习	Supervised Learning	用带标注数据学映射
无监督学习	Unsupervised Learning	找数据内部结构，如聚类
半监督学习	Semi-supervised Learning	少量标注加大量无标注
特征	Feature	描述样本的属性
标签	Label	样本对应的正确答案
损失函数	Loss Function	衡量预测错多少
梯度下降	Gradient Descent (GD)	沿梯度反方向更新参数
学习率	Learning Rate	每次更新的步长
过拟合	Overfitting	训练集很好、新数据很差
欠拟合	Underfitting	连训练集都学不好
正则化	Regularization	加约束抑制模型过分复杂
偏差与方差	Bias-Variance	模型太简单与太敏感的两种误差
交叉验证	Cross Validation	轮流留出一份做验证
超参数	Hyperparameter	训练前设定、不由训练学习的参数
准确率	Accuracy	预测正确的比例
精确率	Precision	预测为正的里面真正为正的比例
召回率	Recall	真正为正的里面被找出来的比例
F1 分数	F1 Score	精确率与召回率的调和平均
混淆矩阵	Confusion Matrix	各类预测结果与真实结果的交叉表
ROC 与 AUC	ROC / AUC	排序质量指标，对阈值不敏感
均方误差	Mean Squared Error (MSE)	回归常用损失
K 折交叉	K-Fold	把数据分成 K 份轮流验证
决策树	Decision Tree	按条件层层切分的模型
集成学习	Ensemble Learning	把多个弱模型组合成强模型
装袋	Bagging	并行训练再投票，如随机森林

中文	英文 / 缩写	一句话解释
提升	Boosting	串行纠错式训练，如 GBDT
随机森林	Random Forest	多棵树 + 样本与特征双重随机
梯度提升树	GBDT / XGBoost	用梯度指导的加法树模型
支持向量机	Support Vector Machine (SVM)	找最大间隔分类面
最近邻	KNN	看最近的 K 个邻居投票
K 均值	K-Means	迭代分配与求中心的聚类
主成分分析	PCA	降维并保留最大方差方向
朴素贝叶斯	Naive Bayes	假设特征独立的概率分类器
五、深度学习与大模型		
神经网络	Neural Network	多层可微函数的叠加
多层感知机	MLP	全连接的前馈网络
反向传播	Backpropagation	用链式法则算梯度
激活函数	Activation Function	引入非线性，如 ReLU
批归一化	Batch Normalization	稳定每层输入分布
丢弃法	Dropout	训练时随机屏蔽神经元以抑制过拟合
卷积	Convolution	局部窗口加权求和，提取局部特征
池化	Pooling	下采样，保留主要信息
循环神经网络	RNN	带状态的序列模型
长短期记忆	LSTM	用门控缓解梯度消失
注意力	Attention	按相关性加权聚合信息
自注意力	Self-Attention	序列内部互相加权
多头注意力	Multi-Head Attention	并行在多个子空间做注意力
位置编码	Positional Encoding	把顺序信息注入并行结构
残差连接	Residual Connection	跳连接，缓解深层退化
层归一化	Layer Normalization	Transformer 常用的归一化
Transformer	Transformer	基于注意力的序列架构

中文	英文 / 缩写	一句话解释
分词	Tokenization	把文本切成模型的最小单位
子词	Subword / BPE	介于字与词之间的切分单位
上下文长度	Context Length	模型一次能看多少个 token
预训练	Pre-training	在海量文本上学语言规律
有监督微调	SFT	用指令数据让模型学会听指令
对齐	Alignment	让模型输出符合人类偏好
人类反馈强化学习	RLHF	用人类偏好训练奖励模型再强化
直接偏好优化	DPO	跳过奖励模型直接优化偏好
低秩微调	LoRA	只训练少量低秩参数
量化	Quantization	用更低比特表示权重以省显存
蒸馏	Distillation	让小学生模仿大老师
规模定律	Scaling Law	损失随参数、数据、算力幂律下降
涌现能力	Emergent Ability	规模到一定程度突然出现的能力
上下文学习	In-Context Learning	只给示例就学会新任务
思维链	Chain of Thought (CoT)	让模型先写推理再给答案
混合专家	MoE	每次只激活部分专家网络
稀疏注意力	Sparse Attention	只计算部分位置之间的注意力
键值缓存	KV Cache	推理时缓存历史 K/V 避免重算
检索增强生成	RAG	先检索资料再让模型作答
智能体	Agent	模型加工具、记忆与循环
幻觉	Hallucination	模型编造看似合理的内容
提示词	Prompt	给模型的输入指令
温度	Temperature	采样随机性，越大越发散
Top-p 采样	Nucleus Sampling	只在累计概率前 p 的候选里采样
六、强化学习		
强化学习	Reinforcement Learning (RL)	通过试错学习决策
智能体	Agent	做决策的一方

中文	英文 / 缩写	一句话解释
环境	Environment	智能体交互的外部世界
状态	State	当前的局面信息
动作	Action	智能体可以做的选择
奖励	Reward	环境给的即时反馈
回报	Return	从当前起累积的奖励
折扣因子	Discount Factor (γ)	未来奖励的折现系数
策略	Policy (π)	状态到动作的映射
价值函数	Value Function (V)	某状态下未来回报的期望
动作价值	Q-Value (Q)	在某状态做某动作的价值
贝尔曼方程	Bellman Equation	价值函数的递推关系
探索与利用	Exploration vs Exploitation	尝试新选项还是用已知最好的
ε-贪心	ε-Greedy	以 ε 概率随机探索
时序差分	Temporal Difference (TD)	用下一步估计更新当前估计
Q 学习	Q-Learning	直接学 Q 值的离线策略算法
深度 Q 网络	DQN	用神经网络代替 Q 表
经验回放	Experience Replay	存下历史样本反复利用
目标网络	Target Network	稳定训练用的慢更新副本
策略梯度	Policy Gradient	直接对策略参数求梯度
演员-评论家	Actor-Critic	策略网络加价值网络
优势函数	Advantage	比平均水平好多少
广义优势估计	GAE	偏差与方差折中的优势估计
近端策略优化	PPO	工业界最常用的策略优化算法
马尔可夫决策过程	MDP	强化学习的标准数学框架

附录 B　常用代码模板速查

这一页里的模板可以直接抄进程序用，也可以在配套的算法动画仿真实验室里逐行改着跑。每个模板只保留最小骨架，省略了输入输出与异常处理。

模板 1　二分查找：左闭右开，返回第一个满足条件的位置

```
def lower_bound(a, x):
    lo, hi = 0, len(a)          # [lo, hi) 左闭右开
    while lo < hi:
        mid = (lo + hi) // 2
        if a[mid] < x:
            lo = mid + 1
        else:
            hi = mid
    return lo                    # 第一个 >= x 的下标
```

用在哪　判断两个元素是否属于同一集合、动态合并集合，且只需要关心连通性时用它。

模板 2　并查集：路径压缩 + 按大小合并

```
class DSU:
    def __init__(self, n):
        self.p = list(range(n)); self.sz = [1] * n
    def find(self, x):
        while self.p[x] != x:
            self.p[x] = self.p[self.p[x]]     # 路径压缩（隔代压缩）
            x = self.p[x]
        return x
    def union(self, a, b):
        ra, rb = self.find(a), self.find(b)
        if ra == rb: return False
        if self.sz[ra] < self.sz[rb]: ra, rb = rb, ra
        self.p[rb] = ra; self.sz[ra] += self.sz[rb]
        return True
```

用在哪　求「前 K 大 / 前 K 小」，数据量大到不能全排序时，用一个大小为 K 的堆即可。

模板 3　堆与优先队列：Top-K 的最小堆写法

```
import heapq

def top_k(nums, k):
    heap = []
```

```
    for v in nums:
        if len(heap) < k:
            heapq.heappush(heap, v)          # 小顶堆，堆顶是当前第 k 大
        elif v > heap[0]:
            heapq.heapreplace(heap, v)       # 比堆顶大就替换
    return sorted(heap, reverse=True)
```

> **用在哪**　在图上判连通、找路径、数连通块。广搜求最短步数，深搜写法更短。

模板 4　图的广搜与深搜

```
from collections import deque

def bfs(g, s):
    seen, q, order = {s}, deque([s]), []
    while q:
        u = q.popleft(); order.append(u)
        for v in g.get(u, ()):
            if v not in seen:
                seen.add(v); q.append(v)     # 入队即标记
    return order

def dfs(g, s):
    seen, st, order = set(), [s], []
    while st:
        u = st.pop()
        if u in seen: continue
        seen.add(u); order.append(u)
        st.extend(v for v in g.get(u, ()) if v not in seen)
    return order
```

> **用在哪**　任务调度、依赖解析、课程表安排；顺便还能检测出「循环依赖」。

模板 5　拓扑排序：Kahn 算法并能检测环

```
from collections import deque

def topo(n, edges):
    g = [[] for _ in range(n)]; indeg = [0] * n
    for u, v in edges:
        g[u].append(v); indeg[v] += 1
    q = deque([i for i in range(n) if indeg[i] == 0])
    order = []
    while q:
        u = q.popleft(); order.append(u)
        for v in g[u]:
            indeg[v] -= 1
            if indeg[v] == 0: q.append(v)
    return order if len(order) == n else None   # None 表示有环
```

> **用在哪**　非负权图的最短路，导航、路由、依赖代价计算都用它。

模板 6　Dijkstra：堆优化，O((V+E) log V)

```
import heapq

def dijkstra(g, s, n):
    dist = [float("inf")] * n; dist[s] = 0
    pq = [(0, s)]                    # (距离, 节点)
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]: continue     # 过期条目直接跳过
        for v, w in g[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                heapq.heappush(pq, (nd, v))
    return dist
```

> **用在哪**　连续子串 / 子数组问题的通用骨架，把 $O(n^2)$ 的枚举压成 $O(n)$。

模板 7　滑动窗口：最长无重复子串

```
def longest_unique(s):
    last, left, best = {}, 0, 0
    for right, ch in enumerate(s):
        if ch in last and last[ch] >= left:
            left = last[ch] + 1      # 窗口左端跳到重复字符之后
        last[ch] = right
        best = max(best, right - left + 1)
    return best
```

> **用在哪**　需要在所有方案里搜索答案的题，配上剪枝就能在大搜空间里跑得动。

模板 8　回溯：全排列与剪枝

```
def permutations(nums):
    res, path, used = [], [], [False] * len(nums)
    def dfs():
        if len(path) == len(nums):
            res.append(path[:])      # 记录一份拷贝
            return
        for i, v in enumerate(nums):
            if used[i]: continue
            used[i] = True; path.append(v)   # 选择
            dfs()                            # 递归
            path.pop(); used[i] = False      # 撤销
    dfs()
    return res
```

> **用在哪**　算 a 的 b 次方取模，密码学与竞赛里的常客；把 b 次乘法压到 log b 次。

模板 9　快速幂：把 b 次乘法压到 log b

```
def power(a, b, mod=None):
    r = 1
    while b:
        if b & 1:                # b 的当前最低位为 1
            r = r * a % mod if mod else r * a
        a = a * a % mod if mod else a * a
        b >>= 1
    return r
```

> **用在哪**　0-1 背包的标准写法。注意内层容量必须从大到小遍历。

模板 10　动态规划：0–1 背包的一维滚动数组

```
def knapsack(w, val, cap):
    dp = [0] * (cap + 1)
    for i in range(len(w)):
        for c in range(cap, w[i] - 1, -1):   # 必须倒序遍历，保证每件物品只用一次
            dp[c] = max(dp[c], dp[c - w[i]] + val[i])
    return dp[cap]
```

附录 C　82 个配套仿真与章节对照

配套的「算法动画仿真实验室」里有 79 个可运行的仿真，按数据结构、经典算法、机器学习、大模型、强化学习分成五组，每一个都对应本书的某一节或某一小节。每个仿真都能在浏览器里直接改代码、改完立刻重跑，执行过程、复杂度、报错位置与修改建议会同步显示；每个仿真都附有 Python、C++、C、Java、Go 五种语言的完整参考实现，JavaScript 则可以直接在页面里执行。建议的用法是：读完某一节的原理，先在仿真里跑一遍默认实现，再故意改坏一行看报错提示，最后用可拖拽的代码片段拼出自己的版本。

分组	仿真	对应小节	主要观察点
数据结构	动态数组	3.1 数组与动态数组	扩容时机与均摊代价
数据结构	单链表	3.2 链表	指针如何改、节点如何丢
数据结构	栈 · 括号匹配	3.3 栈	后进先出与配对过程
数据结构	队列 · 循环数组	3.4 队列	头尾下标如何绕回
数据结构	单调队列 · 滑动窗口最大值	3.4 单调队列	队尾何时弹出、队头何时过期
数据结构	哈希表 · 拉链法	3.5 哈希表	冲突如何挂链、何时扩容
数据结构	二叉搜索树 BST	3.6 树	插入路径与退化情形
数据结构	AVL 平衡树 · 旋转	3.13 平衡树	四种失衡各走哪条旋转路径
数据结构	二叉堆 · 堆排序	3.7 堆	上浮与下沉的每一步
数据结构	图的表示	3.8 图	邻接表与邻接矩阵的差别
数据结构	并查集	3.9 并查集	路径压缩前后的树高
数据结构	字典树 Trie	3.10 Trie	前缀如何共享节点
数据结构	跳表 · 多层索引查找	3.14 跳表	每次「往右」与「下探」发生在哪一层
数据结构	布隆过滤器 · 概率判重	4.8 布隆过滤器	位数组如何被点亮、误判从哪来

分组	仿真	对应小节	主要观察点
数据结构	LRU 缓存 · 淘汰最久未用	3.11 进阶结构	哈希表与双向链表如何配合
经典算法	排序 · 快速排序	4.1 快速排序	比较与交换的次数
经典算法	归并排序 · 分治	4.2 归并排序	分与合各自做了什么
经典算法	二分查找	4.3 二分查找	区间如何收窄
经典算法	Dijkstra 最短路	4.4 Dijkstra	确定顺序与松弛过程
经典算法	A* 寻路	4.5 A*	启发函数如何减少搜索
经典算法	图的遍历 BFS / DFS	4.12 图算法补充	队列与栈分别带来什么顺序
经典算法	动态规划 · 0/1 背包	4.6 动态规划	状态表如何被填满
经典算法	动态规划 · 编辑距离	4.6 动态规划	三种操作的代价选择
经典算法	动态规划 · 最长公共子序列	4.6 动态规划	两个前缀如何一步步对上
经典算法	贪心 · 区间调度	4.7 贪心	按右端点排序为什么最优
经典算法	霍夫曼编码 · 贪心合并	4.16 霍夫曼编码	两个最小值如何合并成一棵新树
经典算法	基数排序 · 按位分桶	4.17 基数排序	每一趟分桶之后数组变成什么样
经典算法	滑动窗口 · 最短连续子数组	4.13 双指针与滑动窗口	两个指针为什么都不回头
经典算法	最长递增子序列 · 打表	4.18 最长递增子序列	dp 表与 tails 表怎样同步变化
经典算法	PageRank 与迭代	4.9 PageRank	权重如何迭代收敛
经典算法	拓扑排序 · Kahn	4.15.1 拓扑排序	入度如何一步步降到 0
经典算法	最小生成树 · Kruskal	4.15.2 最小生成树	哪条边因为成环被跳过
经典算法	单调栈 · 每日温度	4.15.3 单调栈	元素何时进栈、何时被弹出
经典算法	回溯 · N 皇后	4.13 回溯	冲突剪枝发生在哪一步
经典算法	KMP 字符串匹配	4.11 字符串匹配	失配时模式串滑多远
经典算法	两数之和 · 哈希表一趟	1.3 同一个问题的六种解法	从 $O(n^2)$ 到 $O(n)$ 靠的是多记一张小本子

分组	仿真	对应小节	主要观察点
经典算法	最长递增子序列 · O(n log n)	4.18 最长递增子序列	把打表换成维护最小尾巴，log 是从哪来的
经典算法	洪水填充 · 连通区域染色	3.31 洪水填充	水一圈圈扩散，碰到墙就绕开
机器学习	梯度下降 · 线性回归	5.1 线性回归	学习率对收敛的影响
机器学习	最小二乘 · 正规方程	5.1 线性回归	闭式解与迭代解的差别
机器学习	岭回归 · 用正则压住权重	5.11 岭回归	λ 变大时权重如何收缩
机器学习	逻辑回归 · 二分类	5.2 逻辑回归	决策边界如何被推着走
机器学习	KNN 最近邻分类	5.8 KNN	K 值如何改变边界
机器学习	朴素贝叶斯 · 文本分类	5.8 朴素贝叶斯	先验与似然如何相乘
机器学习	决策树 · 信息增益	5.3 决策树	每次分裂选哪个特征
机器学习	随机森林 · 多棵树投票	5.5 随机森林	两处随机性各自带来什么
机器学习	梯度提升 · 残差拟合	5.6 GBDT	新树在拟合谁的残差
机器学习	AdaBoost · 加权投票	5.12 AdaBoost	分错的样本权重如何涨上去
机器学习	支持向量机 · 最大间隔	5.7 SVM	间隔与支持向量怎么定
机器学习	K-Means 聚类	5.8 K-Means	中心点如何移动
机器学习	DBSCAN · 密度聚类	5.13 DBSCAN	核心点、边界点与噪声的分界
机器学习	高斯混合 · EM 迭代	5.14 高斯混合与 EM	责任度与参数如何互相更新
机器学习	PCA · 主成分分析	5.8 PCA	方差最大的方向怎么找
机器学习	阈值 · ROC 与 AUC	5.9 评估指标	阈值挪动时两个指标怎么变
机器学习	混淆矩阵 · 准确率之外的指标	5.9 评估指标	每个样本落进哪一个格子
机器学习	K 折交叉验证 · 换一把尺子量	5.10 常见错误	分数为什么会抖
机器学习	优化器对照 · SGD / 动量 / Adam	6.2 优化器	三条路径在同一曲面上怎么走
大模型	自注意力 Self-Attention	8.2 Transformer	注意力权重如何分配

分组	仿真	对应小节	主要观察点
大模型	多头注意力 Multi-Head	8.13.1 多头注意力	不同的头关注不同的位置
大模型	因果掩码 · 谁不许看谁	8.14 因果掩码	上三角怎样被压成 0 权重
大模型	RoPE · 旋转位置编码	8.2 位置编码	相对位置如何被编码进内积
大模型	LayerNorm · 逐位置归一化	6.3 归一化	归一化前后数值范围的变化
大模型	Softmax · 数值稳定	6.3 归一化 / 8.8 采样	减去最大值前后是否等价
大模型	神经网络 · 反向传播	6.1 反向传播	梯度如何逐层回传
大模型	自回归生成 · 采样	8.8 推理	每一步如何从分布里抽一个词
大模型	KV Cache · 增量解码	8.13.2 KV Cache	有缓存与无缓存的计算量差
大模型	采样策略 · 温度 / Top-K / Top-p	8.13.3 采样策略	哪些候选被砍掉、概率如何重分配
大模型	LoRA 低秩微调	8.6 LoRA 与 QLoRA	参数量随秩如何变化
大模型	MoE · 稀疏专家路由	9.1 MoE	每个 token 被分给哪几个专家
大模型	RAG · 检索与排序融合	8.9 RAG	检索分数如何与生成融合
大模型	BPE 分词 · 合并出子词	8.1 BPE	哪一对字符被优先合并
大模型	DPO · 直接拿偏好对调策略	8.7 RLHF 与 DPO	β 变化时好坏回答的差距
强化学习	Q-Learning	7.3 Q-learning	价值如何随试错更新
强化学习	SARSA	7.10.1 SARSA	与 Q-Learning 的策略差异
强化学习	TD(λ) · 资格迹	7.11 TD(λ)	资格迹如何加一与衰减
强化学习	值迭代 · 动态规划	7.2 值迭代	价值表如何一轮轮收敛
强化学习	DQN · 经验回放	7.3 DQN	回放池如何打散样本相关性

分组	仿真	对应小节	主要观察点
强化学习	Double DQN · 拆开选择与评价	7.12 Double DQN	估计偏差贴不贴零线
强化学习	REINFORCE	7.4 策略梯度	动作概率如何被奖励推高
强化学习	Actor-Critic	7.4 Actor-Critic	价值网络如何给策略当基线
强化学习	PPO 截断目标	7.4 / 7.10.2 PPO	截断区间如何限制每步幅度
强化学习	多臂老虎机 · 探索与利用	7.5 探索与利用	ε 变化时探索与利用的取舍

上面这张表也是阅读的捷径。想动手，就从「数据结构」组开始，它的顺序和第 3 章基本一致；想直接体验大模型，跳到「大模型」组，先看自注意力，再看多头注意力和因果掩码，最后看 KV Cache 与采样策略；强化学习组建议按 Q-Learning → SARSA → TD(λ) → DQN → PPO 的顺序看，价值方法与策略方法的差别会非常直观。每组内部也能横向比：朴素贝叶斯和逻辑回归比的是两种不同的建模假设，K-Means 和高斯混合比的是硬分配与软分配。

后　记

书稿写完，回头看，整本书其实只在反复回答同一个问题：面对一个新问题，先想什么。

从 1945 年的归并排序到 2026 年的混合注意力架构，八十年里算法的形式换了无数次，但想问题的顺序几乎没变：先看清问题的结构（排序、搜索、预测还是决策），再判断能接受的代价（时间、空间、算力、数据），最后才挑工具。这也是我把时间轴放在最前面、把选型指南放在最后面的原因：前面讲有哪些选择，后面讲怎么选。

合上书之前，有三个判断值得再重复一次。

不要背结论，要记约束。一个算法在某个场景里好用，常常不是因为它理论上更优，而是因为它正好匹配了那个场景的约束。离开约束谈优劣，很容易得出错误结论。

不要只看最坏复杂度，也要看常数和实现。二分查找的 O(log n) 很漂亮，但它要求数据有序；哈希表的 O(1) 很快，但它怕冲突和最坏情况。工程里真正决定性能的，往往是那些写不进 Big-O 的细节。

保持跟进，但不必焦虑。第 9 章讨论的那些变化——混合架构、稀疏注意力、极低比特量化、Agent 产业化——多数不是原理上的颠覆，而是效率与工程边界的推移。真正稳定的东西，是前面那些章节里的基本原理。

这本书最初只是一份不断增补的学习笔记。它的价值不在于覆盖了多少名词，而在于把这些名词放回了各自的问题里。如果它能在你做一次技术选择时帮你多问出两个问题，那写它的时间就是值得的。

书里的错误与不足都由我承担。欢迎把勘误和建议反馈回来，让下一次修订更准确。

配套资源

算法动画仿真实验室

本书配套 82 个可交互的算法仿真，按主题分成五组，每一个都对应书里的某一节或某一小节。

数据结构 15 个：动态数组、单链表、栈、队列、哈希表、二叉搜索树、堆、图的表示、并查集、Trie、布隆过滤器、LRU 缓存、单调队列、AVL 平衡树、跳表。

经典算法 23 个：快速排序、归并排序、二分查找、Dijkstra、A*、图的遍历、0/1 背包、编辑距离、最长公共子序列、贪心、霍夫曼编码、基数排序、滑动窗口、最长递增子序列、PageRank、拓扑排序、最小生成树、单调栈、回溯 N 皇后、KMP、两数之和、最长递增子序列（O(n log n)）、洪水填充。

机器学习 19 个：梯度下降、最小二乘、岭回归、逻辑回归、决策树、随机森林、梯度提升、AdaBoost、支持向量机、KNN、K-Means、DBSCAN、高斯混合、PCA、朴素贝叶斯、ROC 与 AUC、混淆矩阵、K 折交叉验证、优化器对照。

大模型 15 个：自注意力、多头注意力、因果掩码、RoPE、LayerNorm、Softmax、反向传播、自回归生成、KV Cache、采样策略、LoRA、MoE、RAG、BPE 分词、DPO。

强化学习 10 个：Q-Learning、SARSA、TD(λ)、值迭代、DQN、Double DQN、REINFORCE、Actor-Critic、PPO、多臂老虎机。

每个仿真都能在浏览器里直接运行：左边是执行过程，右边是可以改的代码，改完点运行立刻重跑。代码写错时下方会指出出错的行和原因；另配可拖拽的代码片段与同一算法的多套参考实现，随时可以一键重置；主题配色有白、暖纸、薄荷、浅蓝、极夜五套，默认是白色。

全部 82 个仿真都配了 JavaScript、Python、C、C++、Java、Go 六种语言的完整可运行实现，在页面顶部切换即可对照阅读。

在线访问与下载

图书与配套仿真都发布在作者的作品集站点上。访问 https://wxekre.cn/algo-book 可以在线浏览本书并下载全文（PDF 与 Word 两种格式）；访问 https://wxekre.cn/algo-lab 可以直接运行全部 82 个算法仿真，并在同一页面里切换 6 种语言的参考实现。站内同时提供本书的封面与内文印刷版文件下载。

勘误与反馈

书中的算法实现与性能数据都经过整理核对，但随着版本更新，仍可能有疏漏或过时的内容。如果发现错误、有更好的讲法，或希望补充某个算法，欢迎通过作品集站点上的联系方式反馈。每一条建议都会进入下一次修订。

本作品为非商业用途的学习资料，著作权归作者任增浩所有。

（全文完）

附录 D　复杂度与选型总表

附录 A 到 C 解决的是「这个词是什么意思」「模板在哪里」「仿真在哪里」。这个附录解决的是另一个频率更高的问题：「这件事到底该用哪个结构、代价是多少？」

下面四张表把全书的复杂度集中在一起。建议把它们当成一个「查询台」：写代码时拿不准用哪个，就回来翻一次。表里的 n 是数据量，m 是边数，d 是维度，k 是簇数或值域大小。

D.1　数据结构操作复杂度

表 D-1　常用数据结构操作复杂度速查

结构	查找	插入	删除	最值	前缀/前驱	空间	一句话记法
数组	O(n)	O(n)	O(n)	O(n)	O(n)	n	一直排着
动态数组	O(1) 下标	O(1) 均摊	O(n)	O(n)	O(n)	n–2n	尾部快、中间慢
单链表	O(n)	O(1)	O(1) 已知前驱	O(n)	O(n)	n	插删快、找慢
哈希表	O(1) 均摊	O(1) 均摊	O(1) 均摊	O(n)	O(n)	n–·	无序但最快
有序数组	O(log n)	O(n)	O(n)	O(1)	O(n)	n	能二分查
堆	O(n)	O(log n)	O(log n)	O(1)	—	n	随时拿最值
平衡二叉搜索树	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	n	有序且能动态改
跳表	O(log n)	O(log n)	O(log n)	O(1)	O(log n)	n	用概率换平衡
线段树	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	4n	区间问题通用
树状数组	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	n	前缀和专用

D.2　排序算法全景

表 D-2　排序算法对比

算法	平均	最坏	空间	稳定	适合场景
快速排序	O(n log n)	O(n²)	O(log n)	不稳定	通用、内存排序首选
归并排序	O(n log n)	O(n log n)	O(n)	稳定	外部排序、链表排序
堆排序	O(n log n)	O(n log n)	O(1)	不稳定	需要最坏情况保证
希尔排序	依步长而定	O(n²)	O(1)	不稳定	中等规模、代码短
插入排序	O(n²)	O(n²)	O(1)	稳定	基本有序、小数据
计数排序	O(n + k)	O(n + k)	O(k)	稳定	值域很小的整数
基数排序	O(d(n + k))	O(d(n + k))	O(n + k)	稳定	长整数、字符串
桶排序	O(n + k)	O(n²)	O(n + k)	取决实现	均匀分布的浮点数

D.3　图算法全景

表 D-3　图算法选择速查

问题	算法	复杂度	负权	备注
单源最短路	Dijkstra（堆优化）	O((n+m) log n)	不能	实际最常用
单源最短路	Bellman-Ford	O(nm)	能	能顺便检测负环
全源最短路	Floyd	O(n³)	能	代码最短，n 小时用
最小生成树	Kruskal	O(m log m)	—	边集 + 并查集
最小生成树	Prim（堆优化）	O((n+m) log n)	—	稠密图友好
拓扑排序	Kahn / DFS	O(n+m)	—	任务依赖顺序
连通性	并查集	O(mα(n))	—	动态连通最优

D.4 机器学习与深度学习模型

表 D-4 模型与关键超参数速查

模型	任务	训练复杂度	推理复杂度	关键超参数
线性 / 逻辑回归	回归 / 二分类	O(nd) / 轮	O(d)	学习率、正则强度
决策树	分类 / 回归	O(nd log n)	O(树深)	最大深度、最小样本数
随机森林	分类 / 回归	O(T·nd log n)	O(T·树深)	树数、特征子集大小
XGBoost / LightGBM	分类 / 回归	较高但可并行	快	学习率、树深、正则项
SVM	分类	$O(n^2)$–$O(n^3)$	取决于支持向量	核函数、C、γ
KNN	分类 / 回归	无训练	O(nd)	k、距离度量
K-Means	聚类	O(nkd) / 轮	O(kd)	k、迭代次数
PCA	降维	$O(nd^2)$	$O(d^2)$	主成分个数
多层感知机	通用	O(n·参数量)	O(参数量)	层数、宽度、学习率
CNN	图像 / 信号	O(n·H·W·C·K)	同训练	卷积核大小、步长
Transformer	序列 / 文本	$O(n \cdot L^2 \cdot d)$	$O(L^2 \cdot d)$（含 KV Cache 时接近线性）	层数、头数、学习率

怎么用 先在表里找到你的任务，看到复杂度后再问一句：我的数据量到底有多大？一万条数据下，$O(n^2)$ 也就几十毫秒；一亿条数据下，$O(n \log n)$ 和 $O(n^2)$ 是一天和三十年的差别。

附录 E　练习题与解题思路

下面 48 道题按「能不能自己写出来」分级：前 24 道只需要第 0 到第 4 章的知识，后 24 道需要组合两个以上章节。每道题后面给了一句话思路，不是答案。先自己想三分钟，想不出来再看思路。

E.1　入门篇（基础结构与经典算法）

表 E-1　入门 24 题

#	题目	考点	一句话思路
1	统计一个数组里出现次数最多的元素	哈希表计数	一边遍历一边计数，同时更新最大值
2	判断一个字符串是不是回文串	双指针	左右两个指针向中靠拢，碰到不同就返回失败
3	反转一个单链表	链表指针操作	三个指针：先存后继、再改指向、最后推进
4	判断括号序列是否合法	栈	遇到左括号入栈，右括号看栈顶是否配对
5	用两个栈实现一个队列	栈与队列	一个进栈、一个出栈，出栈空了就把进栈全倒过来
6	求数组中前 k 大的元素	堆	维护一个大小为 k 的小顶堆，满了就淘汰堆顶
7	实现 LRU 缓存	哈希表 + 双向链表	哈希定位、链表维护顺序，两边都要 O(1)
8	两个有序数组的中位数	二分 / 双指针	双指针从头走到一半，或者对短的那个数组二分切点
9	找出数组中两数之和为目标的下标	哈希表	边走边查「目标 − 当前值」是否出现过
10	判断一个数是不是 2 的幂	位运算	$x > 0$ 且 $x \,\&\, (x-1) == 0$

#	题目	考点	一句话思路
11	不用临时变量交换两个整数	异或	a ^= b; b ^= a; a ^= b
12	求一个数的二进制中 1 的个数	位运算	每次 x &= x−1，直到 x 为 0
13	两个已排序数组合并	双指针	各自一个指针，比较当前值后取小的放进结果
14	求连续子数组的最大和	动态规划	dp[i] = max(a[i], dp[i−1] + a[i])，答案取过程最大
15	爬楼梯方案数	动态规划	到第 n 阶只能从 n−1 或 n−2 来，递推即可
16	零钱兑换最少硬币数	动态规划	dp[金额] 从小到大递推，每次试每个硬币面额
17	活动选择（区间不重叠最多数量）	贪心	按结束时间排序，能接就接
18	分配糖果（满足孩子需求）	贪心	排序后从小到大依次满足
19	二分查找的最左边界	二分	找到目标后不要立刻返回，继续向左收缩
20	求平方根（保留 k 位小数）	二分答案	对答案二分，判断 mid^2 与 n 的大小
21	有序矩阵中查找目标值	二分	从右上角出发，大了往左、小了往下
22	求一个数的所有素因子	试除	只需试到 √n，大于√n 的部分自然对应
23	求两数最大公约数与最小公倍数	数论	gcd 用辗转相除，lcm = a / gcd * b
24	判断链表是否有环	快慢指针	快指针每次走两步、慢指针一步，相遇就有环

练习方法　每道题先写出「输入是什么、输出是什么、数据量有多大」三句话，再动手写代码。数据量决定了你能用什么复杂度，这一步能省下很多白写的代码。

E.2　进阶篇（需要组合两个以上章节）

表 E-2　进阶 18 题

#	题目	考点	一句话思路
25	无重复字符的最长子串	滑动窗口 + 哈希	左右指针，重复就把左指针跳到重复位置之后
26	滑动窗口最大值	单调队列	队列里只保留“比后面都大”的下标
27	二维矩阵的子矩阵和	二维前缀和	一行公式：S[r][c] 相减，四个角
28	区间加、，最后输出数组	差分	D[l] += v，D[r+1] −= v，最后求一遍前缀和
29	数组中第 k 大元素	快速选择	分区后只递归目标一侧，期望 O(n)
30	求区间内每个数出现次数（在线）	莫队 / 分块	离线排序询问后用双指针移动
31	求最长递增子序列长度	二分 + 贪心	维护 tails 数组，每个元素替换第一个 ≥ 它的位置
32	两个字符串的最长公共子序列	动态规划	二维 dp，相等取对角线 + 1，否则取上左最大
33	编辑距离	动态规划	插入/删除/替换三种转移取最小
34	0/1 背包	动态规划	一维 dp 倒着遍历容量
35	迷宫最短路	广度优先	边权相等时广搜就是最短路
36	城市间最短路（非负权）	堆优化 Dijkstra	每次从堆里取未确定的最近点
37	判断任务能否排完	拓扑排序	入度为 0 入队，能出队的个数是否等于 n
38	计算最小连通成本	最小生成树	Kruskal：边排序 + 并查集判环
39	判断两人是否在同一群体且相差几级	带权并查集	存「到父亲的距离」，压缩时累加
40	区间求和 + 单点修改	树状数组	add 向上走 lowbit(i)，prefix 向下走 lowbit(i)

#	题目	考点	一句话思路
41	区间加 + 区间求和	线段树 + 懒标记	整段落入就记账，下访前推标记
42	字符串匹配（模式串）	KMP	失配时用 next 数组回退，不回退主串指针

> **怕样**　这 18 道里任何一道能在三十分钟内从零写出可交代码，说明相应章节真的读懂了。写不出来也正常，回到对应章节把例子重新跑一遍，比硬想有效。

附录 F　调试与排错手册

本附录把第 0.10 节的排错流程展开，按“报错类型”分类整理成一本随时可查的手册。遇到报错时，先到对应的小节里找相似的例子，再回到自己的代码里对照。

F.1　按报错类型查

表 F-1　六类最常见的错误与对应位置

报错关键字	含义	最常见的原因	第一步怎么做
SyntaxError	语法错误，程序没跑起来	括号/引号不配对、缩进不一致	看括号与引号是否成对
NameError	用了没定义的名字	拼写错误、作用域不对	检查拼写与定义位置
TypeError	类型不匹配	字符串与数字相加、参数个数不对	用 type() 打印看看到底是什么
IndexError	下标越界	循环终点写多了一	用小数据手推一遍下标
KeyError	字典里没这个键	键拼写不一致、缺失默认值	用 get() 或先判断 in
ValueError	值不合法	字符串转数字失败、空列表取最值	先校验输入再计算

F.2　按“程序能跑但结果不对”查

这类问题最难，因为没有报错信息。标准做法是“四步定位”。

表 F-2　逻辑错误的四步定位法

步骤	做什么	为什么有效
一、造最小用例	把输入压到只剩 1–3 个元素	人脑对小数据有直觉
二、手推预期	在纸上写出每一步应该是什么	把预期和事实分开
三、打印对照	在循环里打印变量，对照两列	找到第一个偏离预期的位置
四、回退验证	改完后再跑空、1 个、重复元素	确认修复不引入新的边界错误

F.3 九个最常见的具体坑

- 差一错误：range(1, n) 与 range(n) 差一个，下标 0 起算而循环从 1 起算。
- 空集合未处理：max([]) 、a[0]、除以 0，函数开头就该提前返回。
- 浮点比较：0.1 + 0.2 != 0.3，比较要用容差 abs(a-b) < 1e-9。
- 整除与小数除：Python 的 / 总是小数，要整数解用 //。
- 可变对象共享：a = [0]*3 后修改 a[0] 不会影响别人，但 b = [[0]*3]*3 会！
- 字符串不可变：s[0] = "x" 会报错，要用切片重新拼接。
- 递归缺少终止条件：程序报 RecursionError，或者卡死。
- 修改正在遍历的集合：边 for 边删会错位，先拷一份再遍历。
- 作用域错误：在函数里赋值给外层变量，需要 global，但通常应该改成传参数返回。

F.4 一份可以打印出来的排查清单

表 F-3 排查清单（建议打印贴在桌前）

序号	检查项	通过标准
1	报错信息读到最后一行了吗	能说出错误类型和原因
2	能稳定复现吗	同一输入每次都错
3	用最小用例能不能重现	三个元素的输入也错
4	我有一个明确的假设吗	能用一句话说出“我认为是因为”
5	改完后边界还对吗	空、1 个、全相同、已排序都试过
6	为什么以前没发现	能说出触发条件，避免同类错误再现

附录 G　常用数学公式速查

本附录收集书中出现过的公式，按用途分组，方便随时回查。不需要背，知道里面有什么、什么时候去查就行。

G.1　复杂度与增长

表 G-1　常用求和与时间复杂度

表达式	结果	出现在哪
$1 + 2 + \dots + n$	$n(n+1)/2 = O(n^2)$	循环嵌套、逆序对
$1 + 2 + 4 + \dots + 2^k$	$2^{k+1} - 1 = O(2^n)$	子集枚举、递归树
$n + n/2 + n/4 + \dots$	$2n = O(n)$	归并排序的合并层
$\log 1 + \log 2 + \dots + \log n$	$\log(n!) = O(n \log n)$	比较排序下界
二叉树高度	$\lfloor \log_2 n \rfloor + 1$	堆、平衡树
满二叉树节点数	$2^h - 1$	线段树开空间

G.2　概率与统计

表 G-2　基础概率公式

名称	公式	应用场景
条件概率	$P(A\|B) = P(A \cap B) / P(B)$	贝叶斯分类、诊断推理
贝叶斯公式	$P(A\|B) = P(B\|A)P(A) / P(B)$	垃圾邮件、医学检测
全概率公式	$P(B) = \sum P(B\|A_i)P(A_i)$	分类器边际化
期望	$E[X] = \sum x \cdot P(x)$	风险评估、强化学习的回报
方差	$\mathrm{Var}(X) = E[X^2] - (E[X])^2$	数据缩放、资格迹
均方误差	$\mathrm{MSE} = (1/n)\sum(y_i - \hat{y}_i)^2$	回归损失
交叉熵	$H = -\sum y \cdot \log(\hat{y})$	分类损失

G.3　线性代数与微积分

表 G-3　微积分与矩阵公式

名称	公式	应用场景
链式法则	$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)$	反向传播

名称	公式	应用场景
求和的导数	$\partial/\partial w \Sigma = \Sigma \partial/\partial w$	损失函数求梯度
Sigmoid 导数	$\sigma' = \sigma(1-\sigma)$	二分类输出层
ReLU 导数	x>0 时 1，否则 0	隐藏层激活
softmax	$\sigma(z)_i = e^{z_i} / \Sigma e^{z_i}$	多分类、注意力权重
矩阵乘法复杂度	$O(n^3)$ 或 O(n^2.37)	神经网络计算
向量点积	$a \cdot b = \Sigma a_i b_i$	注意力相似度
余弦相似度	$\cos\theta = a \cdot b / (\|a\|\|b\|)$	向量检索、RAG

G.4 工程估算公式

表 G-4 常用工程估算公式

估什么	公式	应用
Transformer 参数量	约 $12 \times L \times d^2$	估算模型规模
FP16 推理显存	约参数量 × 2 字节	选卡、部署
训练显存（全参数）	约参数量 × 16 字节	评估是否需要多卡
KV Cache 显存	2 × L × h × d_head × 序列长 × 2 字节	长上下文部署
训练算量	约 6 × 参数量 × token 数	算力预算
MoE 激活比例	激活专家数 / 总专家数	估算推理成本
阀值缩放	x / √d（d 为向量维度）	注意力 softmax 稳定性

G.5 符号速查

表 G-5 书中常见符号

符号	读作	含义	例子
O(·)	大 O	上界，最坏情况	O(n log n)
Ω(·)	大 Ω	下界，最好情况	Ω(n)
Θ(·)	大 Θ	确界，上下界一致	Θ(n log n)
Σ	西格玛	求和	$\Sigma_{i=1}^{n} x_i$
∏	大 Π	求积	$\prod p_i$
∂	偏导符号	对某一个变量求导	∂L/∂w
∇	nabla	梯度向量	∇L
∈	属于	元素在集合中	x ∈ S
∩ / ∪	交 / 并	集合运算	A ∩ B

符号	读作	含义	例子
$\leq / \geq$	小于等于 /大于等于	不等式比较	$n \leq 100$
$\lfloor \cdot \rfloor / \lceil \cdot \rceil$	向下 / 向上取整	取整操作	$\lfloor n/2 \rfloor$
$\leftarrow$	赋值	在算法描述中表示赋值	$w \leftarrow w - \eta g$

关 于 本 书

把散落的算法串成一条线

算法书通常按「数据结构 + 算法」的目录排，一章一个主题。这样读完，往往还是说不清这些算法彼此什么关系、什么时候该用哪一个。这本书换了一条线索——**时间轴**：从 1945 年的归并排序排到 2026 年的稀疏注意力与混合架构，把每一代算法放回它被发明出来的那个问题里。Dijkstra 是为了修最短路，Transformer 是因为 RNN 没法并行。知道了痛点，再看做法就顺了。

全书不假设任何编程基础。第 0 章用煮鸡蛋、贴标签的盒子、一排编号的抽屉讲清变量、列表、循环和递归；术语第一次出现都带括号解释；代码块后面附「参数与变量解读表」，逐行说明每个符号在做什么。第 3 章用三十一节讲数据结构，后面所有算法的「为什么快、为什么慢」，答案都在那里。

本书看点

- 按时间轴组织：讲清每个算法替代了谁、又被谁替代
- 零基础可读：从煮鸡蛋讲清输入、输出、有限步骤与无歧义
- 代码逐参数解读：符号 → 含义 → 取值与影响
- 表格化速查：复杂度、超参数、显存估算、选型决策清单，共 237 张
- 数据结构专章：数组、链表、栈、队列、哈希表、树、堆、图、并查集、Trie，另加线段树、树状数组、红黑树、B+ 树、跳表、LRU 缓存
- 配套 81 个真执行动画仿真：左边看效果、右边改代码，改完立即重跑
- 7 个附录：术语对照、代码模板、仿真对照、选型总表、练习题、调试手册、公式速查
- 不只讲怎么用，也讲什么时候不该用、坑在哪里

任增浩 著

正 16 开 | 非商业内部资料

条形码位