

100 个问题系列

100 Questions Series

关于 **Codex** 的 100 个问题



AI磊叔·出品

写《100 个问题系列》丛书，初衷很简单

一方面，我想用一种更有趣的方式去收集资料。

很多主题如果只是零散记笔记，最后很容易变成一堆链接、一堆截图、一堆随手记下来的片段，过段时间自己都懒得回看。换成《100 个问题》这种形式以后，资料会慢慢有结构，阅读、搜索、整理、写作，也都会更有抓手。

另一方面，我也想借这个过程，把自己的思考、判断、踩坑经验沉淀下来。

很多时候，真正值钱的是“我怎么理解它”“我在哪些地方踩过坑”“我后来为什么改了看法”。这些东西如果不主动整理，很快就会散掉。

所以这套丛书，既是在收集资料，也是在逼自己把判断写清楚。希望最后留下来的，不只是一些信息汇总，而是一套套带着思考痕迹的主题文档。

怎么用

你可以把它当成一个可直接阅读的主题文档库来看。

可以拿去读，拿去学习，拿去快速建立对一个主题的整体理解。也可以把里面的问题、结构、资料路径继续拿去蒸馏，变成你自己的文章、课程、清单或者研究框架。

如果你在做 Agent、做内容系统、做知识库，或者在写自己的 skill，这个仓库也可以直接当成内容参考。很多主题拆法、章节组织方式、问题设计方式、以及资料转正文的路径，都可以复用。

《100 个问题系列》丛书库

<https://my.feishu.cn/wiki/FC6ZwnwWWi0dWpke56act61gnUd>

我是磊叔，认识一下

<https://my.feishu.cn/wiki/QWQHwA9uYibmtzkZLJBccaEhnNd>

目 录

一、基础认知与定位

1. Codex 到底是个什么东西
2. Codex 有哪几种形态，普通人该用哪个
3. Codex 和我平时在 ChatGPT 里问代码有啥不一样
4. 桌面版 Codex 和命令行版 Codex 有什么区别
5. 没有代码仓库能不能用 Codex
6. 用 Codex 要不要先买会员
7. Codex 能动我电脑上的其他文件吗
8. Codex 算聊天工具还是编程工具
9. 为什么有人吹 Codex 有人骂它难用
10. Codex 适合从零写还是改现成的
11. 普通人用 Codex 最容易高估哪一点
12. 什么样的活不该交给 Codex
13. 不懂编程能用 Codex 吗
14. 用 Codex 之前最该先搞清楚哪件事
15. Codex 会取代我手动干活吗

二、第一次上手与核心机制

16. 第一次用 Codex 该从哪一步开始
17. 登录方式选哪个，ChatGPT 账号还是 API Key
18. 项目到底是什么，为什么一定要先建项目
19. 项目文件夹该选哪个，有哪些坑
20. Sandbox 权限那三档到底怎么选
21. 推理强度那几档有什么区别
22. 计划模式是什么，什么时候该开

- 23. 侧边栏是干什么的，为什么这么重要
- 24. Diff 是什么，为什么每次都要看
- 25. Thread 是什么，一个项目里能开几个
- 26. 怎么知道一个任务还要跑多久
- 27. 任务跑着跑着没动静了怎么办
- 28. 任务跑完了结果在哪看
- 29. 5 小时滚动窗口到底怎么算的
- 30. 怎么查自己还剩多少额度
- 31. Plus 和 Pro 的额度差得远吗
- 32. 额度烧完了任务会立刻停吗
- 33. 手机上能用 Codex 吗
- 34. 关掉 Codex 任务还在跑吗
- 35. Codex Credits 值得买吗

三、踩坑与反模式

- 36. 为什么 Codex 突然开始原地打转
- 37. 任务跑了半小时最后给我个半成品
- 38. 明明写清楚了 Codex 为什么还是漏指令
- 39. Codex 改着改着把好代码也改坏了
- 40. 为什么简单需求 Codex 非要绕一大圈
- 41. 后台任务是不是在偷偷烧我额度
- 42. 为什么关掉窗口额度还在掉
- 43. 为什么我开的任务最后没结果就消失了
- 44. 同样的任务再跑一次结果完全不一样
- 45. Codex 给的方案为啥和我项目对不上
- 46. Codex 说自己改好了怎么一跑全是报错
- 47. 哪些信号说明这个任务该手动叫停

- 48. 任务失败了我该重试还是改需求
- 49. 为什么大任务拆开反而更容易废
- 50. 怎么避免一个任务把整周额度拖垮
- 51. 用 Codex 最容易翻车的人都是哪几条没做到

四、进阶：Skill·MCP·插件·自动化·操控电脑

- 52. Skill、MCP、插件到底有什么区别
- 53. 怎么找到并安装一个好用的插件
- 54. 自己怎么创建一个 Skill
- 55. Skill 怎么写才能被自动触发
- 56. MCP 什么时候才需要碰
- 57. AGENTS.md 到底要不要写
- 58. 任务干预和任务排队怎么用
- 59. 自动化任务适合干什么
- 60. 操控电脑是怎么回事，安全吗
- 61. 三种浏览器方式到底用哪个
- 62. Worktree 模式什么时候该用
- 63. 多方案预览到底看哪个
- 64. Goal 模式和普通任务有什么区别
- 65. 持久对话流和置顶怎么用
- 66. 共享记忆怎么搭最稳
- 67. 语音输入为什么值得用
- 68. AGENTS.md 有哪些高价值的进阶套路
- 69. 怎么让 Codex 改代码前先讲思路
- 70. 怎么让 Codex 记住我项目的约定
- 71. 哪些小习惯能让结果稳定性翻倍

五、场景实战

- 72. 第一次该拿什么任务练手
- 73. 让 Codex 做网页该怎么提需求
- 74. 让 Codex 做 PPT 该怎么玩
- 75. 让 Codex 做视频靠谱吗
- 76. 让 Codex 处理 PDF 和文档怎么用
- 77. 让 Codex 分析数据和对账怎么做
- 78. 让 Codex 帮我清理电脑靠谱吗
- 79. 让 Codex 写文章和做内容怎么用
- 80. 做完整的 side project 怎么用最划算
- 81. 改 bug 该怎么交给 Codex 最有效
- 82. 加新功能该怎么拆给 Codex
- 83. 老项目敢不敢交给 Codex 重构
- 84. 写测试用例怎么交给 Codex
- 85. 让 Codex 做代码审查靠谱吗
- 86. 团队多人协作能用吗
- 87. 想让 Codex 半自动跑长项目该怎么搭

六、边界、替代与反思

- 88. 什么时候该继续用 Codex 其实已经错了
- 89. 不想买 Plus 有没有别的平替
- 90. Codex 和 Claude Code 怎么选
- 91. 已经在用 Cursor 了还有必要开 Codex 吗
- 92. Codex 的额度会一直涨下去吗
- 93. 为什么 OpenAI 把 Codex 越做越像操作系统
- 94. Codex 会不会很快被新东西替代
- 95. 普通人真的需要 AI agent 写代码吗
- 96. 把整个项目交给 AI 改风险有多大

97. 重度依赖 Codex 会不会让我退化，还算我会写代码吗
98. 免费用户有没有什么省着用的方法
99. 什么时候该回到手动写代码
100. 用了一阵之后最该复盘哪件事，长期最大的收获是什么

七、参考资料

官方文档

社区讨论

横向评测

新闻报道

一、基础认知与定位

1. Codex 到底是个什么东西

很多人第一次听说 Codex，以为就是 ChatGPT 换了个名字，专门用来写代码。这个理解差得有点远。

Codex 是 OpenAI 做的 AI agent，核心能力是"进入一个项目现场，把一件明确的活从头干到尾，交一个能验收的结果"。你给 Codex 一个任务，Codex 会自己读你的文件、自己改、自己跑、自己看报错、再改一轮，跑通了把结果拿给你看。整个过程更像在看一个远端的同事在你项目里操作，会思考、会试错、会回头修正。

和普通对话最大的差别在"动不动手"。普通对话你说一句、它回一段代码，你拿去自己复制、自己跑、自己改，AI 只动嘴。Codex 会真的执行，Codex 给你的是一段跑过、验证过的结果，一段能直接用的产物。

这一点很关键，因为后面所有关于"为什么慢""为什么烧额度""为什么有时会跑偏"的疑问，根子都在这里——Codex 在真的干活，不是在猜。

理解了"动手"这两个字，Codex 是什么就清楚了：Codex 是一个会进入项目现场干活的 AI 助手，干的是落地的事。

2. Codex 有哪几种形态，普通人该用哪个

这是新手最容易绕晕的地方，因为 Codex 现在同时存在四种形态，定位完全不同。

桌面版 Codex 是 OpenAI 官方做的图形客户端。最早只有 macOS 版，2026 年 3 月 4 日 Windows 版上线，现在两个平台都能装。打开是一个完整的应用窗口，左边任务列表、中间对话、右边展示结果。Codex 能读你电脑上的本地文件、能操控你电脑上的应用、内置一个浏览器。这是功能最全、对普通人最友好的一种。

CLI 命令行版跑在终端里，没有图形界面。你在哪个文件夹敲 `codex`，Codex 就在哪个文件夹干活。懂命令行的人喜欢它，因为轻、快、和 `git`、`shell` 配合顺。

IDE 插件装在 VS Code、Cursor 这些编辑器里，给程序员在写代码的时候顺手用。

Codex Web / Cloud 跑在云端，专门处理 GitHub 上的项目，改完直接给你开 PR。Codex 跑在 OpenAI 的服务器上，你电脑关了 Codex 照样干活。

四种形态里，普通人、入门用户、不写代码的人，该选桌面版。桌面版有图形界面，不用敲命令，功能也最全——后面这本书的所有内容，默认说的都是桌面版 Codex，除非专门提到别的形态。

3. Codex 和我平时在 ChatGPT 里问代码有啥不一样

差别不在聪明程度，在动不动手。

你在普通对话框里问"这段代码为什么报错"，模型读完上下文给你一段分析，再贴一段修改后的代码。对不对、跑不跑得通，全靠你自己再去验证。模型自己并不知道 Codex 给的代码执行起来会怎样，因为 Codex 压根没执行过。

Codex 走的是另一条路。Codex 接到任务后，会先把你说的需求翻译成一串动作：读哪些文件、改哪里、装什么依赖、跑什么测试。然后一步一执行，每一步都看结果回来，看到底成了没。成了往下走，没成就回头再改。这个循环 Codex 可能要走好几轮。

带来的差别很具体：普通对话里你常遇到"它给的代码看着对，一粘贴进去就报错"；Codex 给你的东西，至少在 Codex 那个环境里是跑通过的。Codex 还会顺手把牵连到的别的文件一起改了，普通对话做不到这种连贯改动。

代价也有。普通对话几秒就回，Codex 要搭环境、要试错，一个任务跑几分钟到几十分钟都正常。你换来的是结果的可信度，用的是等待时间。

4. 桌面版 Codex 和命令行版 Codex 有什么区别

它俩背后是同一套能力，但用法和门槛差很大。

命令行版没有界面，你打开终端，敲命令和 Codex 交互。好处是轻、快、和 git、shell 脚本、自动化流程配合得很顺，懂命令行的程序员用起来如鱼得水。坏处是门槛高，你得会开终端、会敲命令、看得懂吐出来的日志，完全不碰命令行的人会很痛苦。

桌面版是图形界面，打开就是一个完整的应用窗口。左边能看到所有任务、中间打字对话、右边直接看 Codex 生成的文件、网页、代码改动。Codex 能读本地文件夹、能操控你电脑上的应用、内置浏览器可以直接看效果。所有在命令行里要敲的命令，在桌面版里基本都能点出来或者直接说出来。

对普通人和入门用户来说，桌面版几乎在每一个维度都更友好：不用记命令、看得见正在发生什么、结果直接摆在眼前。桌面版比命令行版好上手得多，符合普通人的使用习惯。

所以结论很简单：你会敲命令行、喜欢终端那种感觉，CLI 适合你；你不是程序员、或者只是不想折腾命令行，直接用桌面版。

5. 没有代码仓库能不能用 Codex

能，而且这是 Codex 相比云端版最大的优势之一。

云端版 Codex 最自然的用法是接一个 GitHub 仓库，授权之后 Codex 拉代码、改完开 PR。没有 GitHub 项目，云端版用起来就别扭，因为云端版本来就是围着仓库转的。

Codex 完全不依赖 GitHub。Codex 的"项目"概念，就是你电脑上的一个普通文件夹。你新建一个项目的时候，让 Codex 选一个文件夹，选完之后这个文件夹里的所有文件，Codex 都能读、能写、能改。文件夹里有没有代码、是不是 git 仓库、要不要传到 GitHub，都不影响 Codex 干活。

这意味着你可以拿 Codex 做各种和"代码仓库"八竿子打不着的事。给 Codex 一个放着体检报告 PDF 的文件夹，让 Codex 帮你分析；给 Codex 一个放着素材的文件夹，让 Codex 帮你剪视频；给 Codex 一个空文件夹，让 Codex 从零帮你做一个网页。这些都不需要 GitHub。

唯一要注意的是选哪个文件夹。千万不要选桌面、下载、系统目录这种乱七八糟的地方，也别选中文路径或带空格的路径。专门建一个干净的工作文件夹，把它当作你的专属工作区。

6. 用 Codex 要不要先买会员

要看你想用得有多深。免费额度能让你体验，但干正事基本得上会员。

免费账号给的额度相当克制，大致就是让你感受一下"哦 agent 是这么回事"，跑一两个稍微像样的任务就见底了。真要把 Codex 当成日常工具，绕不开 ChatGPT Plus 这道坎，每月二十刀。

Plus 给的是一个 5 小时滚动窗口里的额度，窗口里大概能跑相当于几十分钟推理的活，小任务够用，大任务几个就到顶。除了 5 小时窗口还有周限额，连着几天猛用会撞顶。Pro 档更贵但额度大得多，重度用户才需要考虑。

还有一个对中国用户很实际的选项：Codex 不一定要用 OpenAI 官方额度。

Codex 支持接入国产模型，通过一个叫 CC Switch 的工具，可以接智谱 GLM、Kimi、DeepSeek 这些国内厂商，价格便宜很多，还不用梯子。很多人日常就是挂着国产模型在用。

所以门槛其实是这样的：只想试水，免费账号够你玩一圈；想正经用，要么买 Plus，要么走国产模型接入。完全不花钱、还想持续干活，这条路基本走不通。

7. Codex 能动我电脑上的其他文件吗

能动，但有边界，边界由你来定。

Codex 有一个叫 Sandbox（沙箱）的权限系统，分三档。第一档只读，Codex 只能看你项目里的文件，不改东西。第二档是项目可读写，Codex 能改你当前

项目文件夹里的文件，但出不去这个文件夹。第三档是完全放开，整个电脑 Codex 都能碰。日常推荐用第二档，第三档新手别碰。

这意味着什么？只要你把项目限定在一个专门的工作文件夹里，Codex 能改的就只是这个文件夹里的东西，你电脑其他地方的文件 Codex 动不了。你的微信、邮箱、系统配置、别的项目的代码，默认都在 Codex 的边界之外。

但 Codex 还有一个能力会突破这个边界，叫"操控电脑"（Computer Use）。

打开这个功能后，Codex 能像人一样看你的屏幕、点按钮、操作你授权的桌面应用。这个能力很强，也更需要谨慎——涉及付款、账号、删除数据的操作，最好别让 Codex 碰，不用的敏感应用先关掉。

所以答案是：默认 Codex 被关在你给的项目文件夹里，想让 Codex 伸到手以外的应用，得你主动开权限、主动授权。开关始终在你手里。

8. Codex 算聊天工具还是编程工具

严格说算 agent，介于工具和工具人之间。形态上 Codex 套着聊天的外壳，干的却是落地的活。

聊天工具的特征是对完即止，你说一句 Codex 回一句，结束。编程工具的特征是你给 Codex 任务 Codex 给你产物。Codex 两头都沾一点，又两头都不完全是。

Codex 有聊天的外壳：你在对话框里和 Codex 说话，描述任务、给反馈、让 Codex 改。但 Codex 干的活更像工具：Codex 会真的去改文件、跑代码、操控应用、交付一个可验证的结果。中间 Codex 还自己思考、自己决策、自己试错，这部分又像是在和一个同事合作。

OpenAI 自己的定位也越来越清晰：ChatGPT 是入口，Codex 是执行层。你通过对话把任务交进去，但任务真正落地是在 Codex 这个执行层完成的。

OpenAI 内部甚至已经基本用 Codex 替代 ChatGPT 来做工程类工作。

对用户的实际影响是：别用聊天的期待去用 Codex。聊天你可以随便问、随时停、没压力。用 Codex 你得有任务意识，每一次对话都在消耗额度，每一次任务都得有验收标准，不然就是烧钱。

9. 为什么有人吹 Codex 有人骂它难用

这个问题没有标准答案，但有一个很关键的变量：用的人把 Codex 当什么用。

骂得最凶的，往往是把 Codex 当加强版 ChatGPT 用的人。他们丢一个模糊的需求进去，等着 Codex 像对话那样秒回一个完美结果。结果等了几分钟，拿回来一个改得七七八八的东西，要么漏了需求，要么改岔了，自然觉得难用。这类用户的期待和 Codex 实际的工作方式之间，差着一整个“agent 该怎么用”的认知。

吹得最响的，通常是那些会写需求、会拆任务、会盯过程的人。他们知道 agent 需要清晰的输入，知道大任务得拆开，知道跑偏了要及时叫停重来。同样一个 Codex，在他们手里能稳定产出可用的结果，自然觉得好用。

除了使用方式，Codex 本身也确实有版本波动。社区里经常能看到“最近变难用了”“上周还好好的这周就跑不动”这类反馈，这背后是模型版本和系统更新的影响。GPT-5.5 上线之后那阵子，就有不少人反映长任务退化、容易跑偏。这种波动是真实存在的。

一半是用法问题，一半是产品本身还在调；会用的当宝，不会用的当坑，两边都没完全说错。

10. Codex 适合从零写还是改现成的

改现成的，比从零写稳得多。

这个结论可能反直觉，因为很多人觉得“从零写更自由，AI 想怎么发挥怎么发挥”。但 Codex 干活的方式决定了 Codex 更需要上下文，而不是更自由。

改现成代码的时候，项目结构、命名约定、依赖关系、已有的测试和配置文件，这些信息 Codex 都能读到。Codex 在改的时候有参照系，知道改哪里、怎么改才和现有代码风格一致，改完也能拿现有测试去验证。这种场景下 Codex 的输出质量很高，经常让你觉得“这改动我自己写也差不多就这样”。

从零写就完全是另一种局面。你只给一段需求描述，没有现成代码做锚点，Codex 要自己决定项目结构、技术选型、文件组织。每一个决定都可能和你心里想的不一样，而 Codex 又没法读你的心。结果经常是 Codex 给了一版完整的代码，你一看技术栈、目录结构、风格全是 Codex 自己定的，和你预期的差很远。

所以如果你要用 Codex 从零起项目，最好的办法是先手动搭一个最小骨架，哪怕只是空的目录结构加一个 README，让 Codex 在这个骨架上接着写。给 Codex 锚点，Codex 就稳；让 Codex 凭空发挥，Codex 就飘。

11. 普通人用 Codex 最容易高估哪一点

最容易高估的是 Codex 的"自主性"。

很多人带着一种期待打开 Codex：我把需求一丢，Codex 自己就能从头到尾干完，干完给我一个能用的东西。这种期待一般来自各种 demo 视频和宣传文案，里面展示的往往是任务跑通之后那个漂亮的结果，中间那些反复调整、叫停重来、补充说明的过程都被剪掉了。

真实的体验是这样的：Codex 确实能自主完成很多动作，但 Codex 完成得好不好，极度依赖你给 Codex 的输入有多清晰。一个模糊的需求丢进去，Codex 会按 Codex 自己的理解往下跑，跑出来的东西往往和你想的差一截。你以为

Codex"自主"了，其实 Codex 是在按 Codex 自己的猜测自主，而 Codex 的猜测经常不在你的频道上。

真正用得顺的人，都不会把 Codex 当成一个"丢完就不管"的黑盒。他们会把任务拆细、把验收标准讲清楚、跑偏了及时打断重来、跑完认真审一遍结果。他们信任 Codex 的执行能力，但不放任 Codex 自由发挥。

高估自主性带来的直接代价是额度。你丢一个模糊的大需求，Codex 在自己猜的过程中会跑很多轮、试很多方向，每多一轮都是真金白银。最后拿回来一个还不太对的东西，你既赔了额度又得返工。

12. 什么样的活不该交给 Codex

几种活，交过去基本是浪费时间或者烧钱。

需求你自己都说不清的，别丢给 Codex。如果你脑子里对要做的东西还是一团模糊，只有一个大概感觉，Codex 没法替你想清楚需求，Codex 只会按 Codex 自己那团模糊的理解往下跑，结果大概率不对，你还得多花好几轮去纠偏，还不如自己先想明白。

没法验证对错的活，Codex 的优势也用不上。Codex 强在 Codex 能执行、能验证，如果一件事情做完之后没有客观标准判断"成了还是没成"，那 Codex 的

执行能力就用不上。比如“帮我想几个产品方向”“帮我评估这个设计好不好”，这种主观判断类的事情，普通对话就够了。

强依赖你本地敏感环境或者真实账号的活，尤其是涉及钱的，风险太高。让 Codex 帮你操作付款、登录银行账号、改生产数据库，这些事出了岔子没法收。涉及付款、账号设置、删除数据的操作，最好别让 Codex 碰。

你自己完全看不懂、也无法验收的活，Codex 能给你一个结果，但如果你连这个结果对不对都判断不了，那这个结果对你没用。要么全盘接受（风险高），要么逐行核对（比自己干还累）。

13. 不懂编程能用 Codex 吗

能，而且这正是 Codex 相比 CLI 的最大意义之一。

命令行版 Codex 假设你会用终端、看得懂日志、知道怎么处理报错，对完全不碰代码的人门槛很高。Codex 把这些都包进了图形界面，你打字描述需求，Codex 执行，结果直接在右边窗口里展示。整个交互更像在和一个懂技术的同事说话，而不是在操作一个开发工具。

大量非程序员用 Codex 的真实场景已经跑通：做电商的用 Codex 剪视频、拆爆款；做内容的用 Codex 写文章、做封面、做 PPT；普通职员用 Codex 分析

体检报告、清理磁盘、对账。这些都不需要懂编程，只要你把"想让 Codex 干什么"说清楚。

但"不懂编程能用"不等于"什么都不用懂"。你还是要懂几件事：怎么给 Codex 一个清晰的任务、怎么验收 Codex 的结果、什么时候该叫停、哪些操作有风险。这些属于用 agent 的常识，跟编程知识没关系。

一句话总结：Codex 把编程门槛降到了零，但把"会用 agent"的门槛留在了那里。前者 Codex 替你解决了，后者还得你自己跨。

14. 用 Codex 之前最该先搞清楚哪件事

先搞清楚你这个活，做完了"好不好"能不能自己验。

这句话听着简单，但它是用 Codex 顺不顺的分水岭。很多人用得不顺，根子都在这里：他们把任务丢出去，等结果回来，结果回来了却不知道好不好，只能凭感觉说"好像行吧"。这种用法下，Codex 给你的结果好坏全凭运气。

为什么这件事这么关键。Codex 的整个工作模式，是替你把"改-跑-看结果-再改"这个循环跑一遍，最后给你一个在 Codex 那个环境里通过的版本。Codex 能保证的是"在 Codex 那个环境里跑通了"，但不能保证"这个东西真的解决了你的问题"。这两者之间的差距，只能靠你来填。

你能在拿到结果之后清楚判断"这个对了"还是"这个还不对", 用 Codex 就很顺—对了直接用, 不对就告诉 Codex 哪里不对让 Codex 再改。你判断不了, 那就只能要么全盘接受, 要么逐行核对, 哪个都累。

所以用之前, 先问自己: 这事做完了, 我怎么知道成了? 有测试可以跑、有明确的预期行为、有可以对照的参考, 这些都能帮你验。如果连你自己都不知道"成了"长什么样, 先别急着开工, 先把验收标准想清楚。

15. Codex 会取代我手动干活吗

短回答: 会改变你干活的方式, 不会让你什么都不用干。

Codex 擅长的是把一件明确的、可验证的活执行完。你给 Codex 清晰的输入, Codex 给你跑通的结果。这部分劳动—那些重复的、有套路的、需要反复试错的—确实会被 Codex 接走。用了之后很多事自己再不想手动干了。

但 Codex 取代不了的, 是判断。哪个任务该交给 Codex、任务描述得清不清楚、结果对不对、要不要接受、下一步往哪走, 这些判断全都压在你身上。

Codex 越能干, 你判断的权重反而越重, 因为 Codex 的每一次执行都消耗额度、每一次改动都影响项目。

还有一个 Codex 取代不了的, 是你对活本身的理解。你完全不懂一件事, 把活丢给 Codex, Codex 给你一个结果你验收不了, 那这个活对你还是没有真正完

成。Codex 最舒服的用法，是你对这件事有足够的理解，让 Codex 替你把执行的部分扛走，你专心做判断和决策。

所以更现实的期待是：Codex 会让你从一个"动手干活的人"，慢慢变成一个"提需求、验收结果、做决策的人"。这个过程里，你干的活变少了，但你想得更多了。这算不算取代，看你从哪个角度想。

二、第一次上手与核心机制

16. 第一次用 Codex 该从哪一步开始

别急着给 Codex 派活，先做两件事：装好、体检。

下载的渠道只有一条，就是 OpenAI 官方入口。Mac 用户去官网下 dmg，注意分清 Apple Silicon (M 系芯片) 和 Intel 版本，选错了装不上。Windows 用户去 Microsoft Store 搜 Codex，发布者认准 OpenAI；Codex 的 Windows 版是 2026 年 3 月才上线的，对 Windows 11 体验最好，Windows 10 也支持。不要从奇奇怪怪的第三方站点下，那种渠道质量参差不齐。

装完第一次打开，会走一个登录流程。三个选项：ChatGPT 账号登录、API Key 登录、第三方中转站。普通人选第一个，用你的 ChatGPT 账号登进去就行，走的是套餐额度，最省心。API Key 是给开发者和自动化场景用的，新手别碰。

登完 Codex 会问你职业身份 (Engineer / PM / HR 之类)，随便选一个就行，这个只是给 OpenAI 做统计，不影响功能。接着会问要不要开 Sandbox 沙箱模式，一定点 Set up 开启，这样 Codex 跑命令时不会乱动你项目目录之外的东西。

做完这些，别马上丢任务。先随便发一句"你好"，确认 Codex 能正常回复，说明全套配通了。然后再开始真正的项目工作。

第一次接触的最低目标就一条：确认"装对了、登对了、能跑了"。能不能干活先放一边，这一步顺过去，后面才不会埋雷。

17. 登录方式选哪个，ChatGPT 账号还是 API Key

普通用户选 ChatGPT 账号登录，别碰 API Key。这是新手最容易选错的岔路口。

ChatGPT 账号登录走的是你的套餐额度，Plus、Pro 这些会员每个月给你一定的使用量，用完了等下个周期恢复。你不用管花了多少 token、多少钱，OpenAI 都帮你算好了。这种模式对普通人是透明的、可预期的，最省心。

API Key 登录走的是 OpenAI Platform 的按量计费，用多少 token 扣多少钱，和充电话费一个道理。好处是没有固定月费、用多少算多少，坏处是你得盯着消耗，跑偏的任务可能瞬间烧掉一笔钱。这种模式适合开发者、自动化场景、CI/CD 流水线，不适合普通人日常用。

一个判断标准：如果你听到"token""按量计费"这两个词还得愣一下想是什么意思，那就老老实实用 ChatGPT 账号登录。等你用熟了、想精细控制成本了，再去研究 API Key 那条路。

还有第三条路，就是通过 CC Switch 这种工具接国产模型，这条路其实也是走 API Key，但用的是国内厂商的额度。这条路对中国用户更便宜，但属于非官方玩法，留到第六章细讲。

18. 项目到底是什么，为什么一定要先建项目

项目，就是你电脑上的一个文件夹。

Codex 不是单纯的聊天工具，Codex 需要进入一个具体的"项目目录"才能干活。

你新建一个项目的时候，让 Codex 选一个文件夹，选完之后这个文件夹里的所有文件，Codex 都能读、能写、能改。文件夹就是 Codex 的工作边界，也是 Codex 的信任边界。

为什么非要这一步？因为 Codex 的核心能力是"动手"，动手就得有个动手的地方。你让 Codex"帮我改这个网页"，Codex 得知道网页在哪个文件夹、有哪些文件、依赖装在哪。没有一个明确的项目目录，Codex 就像一个被蒙着眼睛扔进陌生办公室的人，不知道从哪下手。

很多人跳过建项目，直接在普通对话框里让 Codex 干活，结果遇到一堆怪事：Codex 说找不到文件、Codex 改的东西不知道存哪了、Codex 跑的命令报莫名其妙的错。这些问题的根子都在“没建项目”。

建项目还有一个好处，是结果可追溯。项目模式下，Codex 生成的所有文件都存在同一个文件夹里，你打开文件夹就能看到全部产物，不会像对话模式那样生成的东西散落在各处、找不到。日常用基本都开项目模式，除非问个临时小问题才用普通对话。

所以正确的打开方式是：先建一个专门的文件夹当工作区，再把每个具体的活当成这个文件夹里的一个项目。

19. 项目文件夹该选哪个，有哪些坑

这个选择比你想象的重要得多，选错了后面全是麻烦。

先说该选什么。专门新建一个干净的文件夹，专门给 Codex 用，里面再按项目分子文件夹。比如 `D:\AI-Codex-Projects\我的网页项目`，或者 Mac 上的 `~/AI-Codex-Projects/我的网页项目`。路径用英文最稳，中文路径和空格在 Windows 上特别容易出毛病。

再说千万别选什么。C 盘根目录、桌面、下载文件夹、系统目录、重要资料文件夹、中文路径、带空格的路径，这些都不能用。

C 盘根目录和系统目录权限敏感，Codex 跑命令容易撞墙。桌面和下载文件夹太乱，里面什么都有，Codex 分不清哪些是你要处理的、哪些是你的私人文件。重要资料文件夹风险在于，万一 Codex 理解错了任务，把你的重要文件改了或删除了，恢复起来很麻烦。中文路径和空格则是技术层面的坑，很多命令行工具对这两种路径处理不好，容易报莫名其妙的错。

还有一个隐形坑：不要选一个正在被别的程序占用的文件夹。比如你正在 VS Code 里编辑这个项目，同时又让 Codex 去改 Codex，两边可能冲突，文件状态会乱。要么关掉编辑器，要么给 Codex 用 worktree 模式（这个后面讲）。记住一条铁律：建一个干净、专用、英文路径的工作文件夹，把它当作你和 Codex 共同的工作区，所有项目都在这里发生。

20. Sandbox 权限那三档到底怎么选

Sandbox 是 Codex 的安全围栏，分三档，选错要么束缚手脚，要么风险失控。

第一档叫只读，Codex 只能看你项目里的文件，不改东西。适合你只想让 Codex 分析、解释、给出建议，但不想让 Codex 动文件的场景。比如“帮我看这个项目的架构合不合理”，这种用只读档最安全。

第二档叫项目可读写，Codex 能改你当前项目文件夹里的文件，但出不去这个文件夹。这是日常推荐档位，绝大多数活都用这档。Codex 能读、能写、能跑

命令、能装依赖，但所有动作都被限制在你的项目目录里，你电脑其他地方的文件 Codex 碰不到。

第三档叫完全访问，整个电脑 Codex 都能碰。这一档新手不要碰，只有在你非常清楚自己在干什么、并且完全信任当前任务的时候才考虑。开了这档相当于把电脑钥匙全交给 Codex，Codex 要是理解错了任务，能搞出大麻烦。

Codex 里这三档对应的是"请求批准 / 自动审核 / 完全访问权限"三个选项。

新手建议保持"请求批准"，意思是任何命令执行前都先问过你，你确认了才执行。这样慢一点，但安全。中间的"自动审核"档，系统会替你判断某些安全操作要不要放行，比"请求批准"省心，又不至于像"完全访问"那样放任。

权限给多大，说到底是"信任 vs 效率"的权衡。权限小，安全但繁琐；权限大，丝滑但风险高。刚开始小权限，等你熟了、信任建立起来了，再逐步放开。一上来就开最大权限是最容易翻车的做法。

21. 推理强度那几档有什么区别

推理强度控制的是 Codex 在回答你之前思考多久。档位越高，Codex 想得越深，但越慢、越烧额度。

通常是四档。低档适合改文案、改颜色、回答小问题这种轻活，几秒就完事。中档适合普通网页、简单 bug、日常开发，这是最常用的档位。高档适合多文件

修改、复杂 bug、重构这种需要深思考的活。最高档适合很难的问题、架构分析、反复修不好的 bug，Codex 会想很久，但往往能想到你意想不到的解法。

很多人犯的错是：所有任务都开最高推理，觉得"越高越聪明"。这是双重浪费。

小任务根本不需要那么深的思考，高档和低档在改个变量名这种事上结果一样，你白等时间。高档消耗的额度远多于低档，你白烧额度。

正确的用法是按任务匹配。改个文字、调个样式，用低档；做个普通功能、修个常规 bug，用中档；只有遇到真正复杂、需要 Codex 深入分析的问题，才上高档。省额度最核心的原则就是这一条：小任务不开最高推理。

如果你拿不准该用哪档，从中档开始最稳。中档是"够用且不浪费"的甜点位，跑完看结果，要么调高要么调低，几次之后就摸清规律了。

顺带说一下当前的主力模型。截至 2026 年中，Codex 的旗舰模型是 GPT-5.5，2026 年 4 月底上线，Plus、Pro、Business、Enterprise 账号都能用，Edu 和 Go 也在官方可用范围里。相比之前偏编程特化的 GPT-5.3-Codex，GPT-5.5 更像一个能规划、能执行、能检查的工程负责人，强在推理和长链路任务。对 Codex 用户来说，有两个关键数字值得记一下：上下文窗口扩展到了 400K tokens（能装下更大的项目和更长的对话），还多了一个 Fast mode（速度约快 1.5 倍，但 token 消耗约 2.5 倍，适合赶时间的任务）。模型和档位会持续更新，具体以你账号里实际能选为准。

22. 计划模式是什么，什么时候该开

计划模式是 Codex 最被低估的功能，也是防翻车最重要的一道闸。

正常模式下，你给个任务 Codex 直接开干。计划模式下，Codex 不立即动手，而是先停下来，整理出一份工作计划：打算怎么做、分几步、会动哪些文件、有什么风险，然后等你确认了再开始。

为什么这个模式这么重要？因为 agent 干活快，快到你来不及反应。你一句话丢进去，Codex 可能已经改了十几个文件，等你发现方向错了，已经改完了一大半，回退都麻烦。计划模式就是给你一个"刹车的机会"，在 Codex 真正动手之前，先看看 Codex 的思路对不对。

跳过计划模式是项目出错的最大原因。一句话的误解，可能让 Codex 改了四十个文件，而如果你先看计划，一句话就能纠正。

什么时候该开？所有稍微复杂的任务都先开计划模式。改个变量名、调个颜色这种小事不用，但凡涉及多文件、新功能、重构、修复复杂 bug，先开计划模式让 Codex 出方案，你审完再让 Codex 执行。

用法很简单，在对话框里打 `/plan`，或者描述任务时说"先给我一个计划"。

Codex 会返回一份分步方案，你看完觉得没问题，再说"开始执行"。觉得有问题，直接在计划阶段提出来改，比写完代码再改省事十倍。

23. 侧边栏是干什么的，为什么这么重要

侧边栏是 Codex 的"作业检查区"，也是 Codex 和普通聊天工具拉开差距的关键。

普通对话里，AI 给你一段文字或者一段代码，你看完就完了。Codex 不一样，Codex 干活的时候，右边那个侧边栏会实时展示 Codex 正在做什么、生成了什么、改了哪里。这个侧边栏就是 Codex 的"工作现场"。

侧边栏能展示的东西很多：Codex 生成的文件、搜索来源、网页预览、图片预览、PDF 预览、内置浏览器、代码改动对比。中间告诉你"Codex 做了什么"，右边让你看"Codex 产出了什么"。

侧边栏最有价值的几个用法里，最该盯紧的是看代码改动。Codex 改完文件后，侧边栏会显示一个 diff，绿色是新增、红色是删除，你能清楚看到 Codex 到底动了哪些行。除此之外，Codex 做了个网页，侧边栏直接打开就能看效果，省得你自己去启动服务器。更妙的是，网页预览时你可以直接在页面上圈出某个按钮，说"这个颜色改深一点"，Codex 能精确收到你的反馈，只改你指的那一处。

很多人刚用 Codex 的时候，眼睛只盯着中间的对话框，把 Codex 当聊天工具用，完全忽略侧边栏。这是巨大的浪费。真正会用的人，眼睛是盯着侧边栏的，

因为那才是 Codex 干活的现场。中间的对话只是你指挥 Codex 的渠道，右边的侧边栏才是 Codex 交作业的地方。

24. Diff 是什么，为什么每次都要看

Diff 就是代码改动的对比，绿色是新增、红色是删除。每次 Codex 完成任务后，第一件事就是看 diff。

为什么这么强调看 diff？因为 Codex 改得很快，快到你不知道 Codex 到底改了什么。如果只看 Codex 给你写的文字总结——“我修改了登录逻辑，优化了样式”——你根本不知道具体动了哪些地方。文字总结可以糊弄，diff 不会骗人。看 diff 能帮你抓出几种问题。最常见的，Codex 改了不该改的地方——你以为让 Codex 改登录页，结果它顺手把首页的样式也动了，这种“顺手改”在 diff 里一眼就能看出来。还有就是删了不该删的东西，有些代码 Codex 觉得“没用”就删了，但其实是你的业务逻辑；再就是莫名其妙多了个新依赖、多了个新文件，这些在 diff 里都很显眼。

有一条硬规矩值得反复说：每次 Codex 完成任务后，先看 diff，再决定要不要接受。不要只看 Codex 的文字总结就放心。看到不理解的改动，先停下来问 Codex“这里为什么这么改”，而不是默认接受。

Diff 还能用来迭代。你在 diff 某一行的旁边留个评论，说"这里改得不对，换种方式"，Codex 能收到你的反馈，只针对那一处再改。这比把整个任务重新描述一遍高效得多。

养成"先看 diff"的习惯，是用 Codex 用得稳的最低门槛。这个习惯能把绝大多数翻车挡在合并之前。

25. Thread 是什么，一个项目里能开几个

Thread 就是一个项目里的一个具体任务对话。一个项目能开很多个 thread。

为什么要分 thread? 因为一个项目里你要干的事通常不止一件。做网页的过程中，你可能一边让 Codex 修首页的 bug，一边让 Codex 加个新功能，一边让 Codex 改样式。这些是三个独立的任务，塞进一个对话里会乱——Codex 分不清你新说的话是接着上一个任务，还是开新任务，上下文会搅在一起。

正确做法是一个明确的任务开一个 thread。修 bug 是一个 thread，加功能是另一个 thread，改样式又是一个。每个 thread 专心干一件事，上下文干净，结果也清晰。项目像一个公司，thread 像公司里的员工，每个员工负责一摊事。

thread 之间可以并行。你布置完一个任务，不用等 Codex 完成，可以新建一个对话直接布置另一个。多个任务只要不是改同一个文件，就不会互相干扰。每个 thread 的进度是独立的，你能分别看到哪个完成了、哪个还在跑。

完成的 thread 可以归档，归档不是删除代码，是把任务列表收起来，保持界面清爽。归档了还能找回来，在设置里取消归档就能还原。如果一个 thread 跑得不好，想换个思路重来，可以 fork 一份去试新方案，原来的还在。

记住一条：不要把所有事情塞进同一个 thread。任务越清楚、thread 越独立，Codex 越稳定。

26. 怎么知道一个任务还要跑多久

这个问题没有精确答案，但有几个信号可以判断。

Codex 跑任务的时候，左边对话列表里那个任务的状态会变化。如果是转圈的图标，说明 Codex 还在干，没结束。如果变成了一个静止的蓝色点，说明任务已经跑完了，你可以去验收。

至于具体还要多久，Codex 不会给你一个倒计时。一个任务快可能几十秒，慢可能十几分钟，差距很大。影响因素有几个：任务本身的复杂度、Codex 要不要装依赖、Codex 要跑几轮试错、当前模型版本的状态。改个变量名这种事，几秒就完；让 Codex 从零做一个完整应用，跑半小时也不奇怪。

如果你觉得 Codex 跑得太久，可以中途去看 Codex 在干什么。点开那个 thread，能看到 Codex 当前正在执行的动作——在读文件、在跑命令、在等什么。如果 Codex 卡在某个动作上很久不动，可能是遇到了问题；如果 Codex 在正常地一步步推进，那就是任务本身需要这么久。

有一种常见现象要警惕：Codex 跑着跑着会进入"原地打转"的状态，反复尝试同一种解法但一直失败。这种情况下等再久也没用，应该手动叫停，换个思路重来。怎么判断 Codex 是不是在打转，看 Codex 的动作日志，如果连续几轮都在做差不多的事，基本就是在绕圈子了。

27. 任务跑着跑着没动静了怎么办

先别慌，分情况处理。

最常见的情况，Codex 真的还在跑，只是慢。有些任务本来就耗时，比如装一堆依赖、跑完整的测试套件、处理大文件。点进 thread 看 Codex 的动作日志，如果还在正常推进，那就是等的事，别打断。

还有一种情况，Codex 在等你批准什么。Codex 在"请求批准"模式下，遇到敏感操作会停下来等你确认，比如要装个新依赖、要改某个文件、要跑某条命令。这时候 Codex 是在问你要不要让 Codex 继续，没卡住。检查一下界面有没有待批准的提示，处理了 Codex 就能往下走。

再有一种，Codex 真的卡住了或者原地打转。判断方法是看动作日志，如果连续几轮都在重复类似的操作、反复报同样的错、或者干脆没有任何动作更新，那就是卡了。这种时候等再久也没用。

卡住了怎么办？最直接的办法是手动叫停，重新描述任务，或者缩小任务范围。

也可以直接停掉这个 thread，开一个新的，把上下文重新整理一遍再给

Codex。要是环境问题（比如依赖装不上、网络不通），就先暂停任务，去把环境收拾干净，解决完再让 Codex 继续。

连续失败时要叫停，不要让 Codex 无限乱试。Codex 有时候会陷入“改一版跑一遍发现还不对再改一版”的死循环，每多一轮都在烧额度，最后大概率还是不行。及时止损比硬等明智。

28. 任务跑完了结果在哪看

结果要在三个地方看，各有各的用处。

中间的对话框最快——Codex 跑完会给你一段文字总结，说做了什么、改了哪里、结果如何。这是最快速的概览，但也是最不可信的部分，文字总结可以糊弄，你得结合别的地方验证。

真正要看的是右边的侧边栏。如果 Codex 改了代码，侧边栏会显示 diff，你能逐行看 Codex 到底动了什么；如果生成了新文件，侧边栏能预览那个文件；

如果做了网页，侧边栏能直接打开网页看效果。侧边栏就是 Codex"交作业的地方"。

最实在的是你项目文件夹本身。Codex 改的所有文件、生成的所有产物，最终都落在你建项目时选的那个文件夹里。打开 Finder 或者资源管理器，去那个文件夹看，所有东西都在那。不管 Codex 文字总结怎么写，文件夹里的真实状态是骗不了人的。

验收的标准流程是这样的：先扫一眼文字总结知道大概，然后认真看 diff 确认改动符合预期，再打开实际产物（网页、文件、图片）确认效果，最后跑一遍你自己的测试或者验证流程。这四步走完，才能说这个任务真的完成了。

千万不要只看文字总结就放心。每次完成后先看 diff，再决定要不要接受。文字总结 + diff + 实际产物 + 你的验证，四重检查一个都不能少。

29. 5 小时滚动窗口到底怎么算的

这是 Plus 用户最绕不清的一件事，搞不懂就总在莫名其妙烧额度。

先讲一个最容易误解的点：Plus 的 5 小时窗口看的不是"5 小时内你能发多少条消息"，看的是"5 小时窗口里你有多少推理计算量"。OpenAI 官方文档的原话是，本地消息和云端任务共享一个 5 小时窗口，窗口里的额度按照推理计算量（reasoning compute）来算，跟消息条数没有直接关系。

社区里有实测数据：Plus 大概是每个 5 小时窗口里有相当于 40 分钟左右的推理时间。Pro 5x 档大概是这个数字的 5 倍。注意，这个"40 分钟"指的是你跑的任务折算成推理计算量，大约等于 40 分钟的推理消耗，跟你实际坐在电脑前的挂钟时间没关系。一个简单的改文字可能几秒就完，几乎不消耗；一个复杂的多文件重构可能一条消息就吃掉好几分钟的推理额度。

这就是为什么很多人觉得"我没发几条消息额度就没了"——你发的几条都是重活，每条消耗大。同样，也有人觉得"我发了挺多条还有额度"——因为都是轻活。

这里还有个"滚动"的坑。窗口并不会"每 5 小时整点刷新一次"，而是滚动计算——你 5 小时前用的那部分额度，要到 5 小时之后才"释放"回来。所以你不会感觉到一个明确的"刷新时刻"，额度是慢慢恢复的。

除了 5 小时窗口，还有一个周限额。社区里有人实测，连续重用几天就会触顶。触顶之后只能等下周恢复。

30. 怎么查自己还剩多少额度

Codex 里能直接看，不用去网页查。

打开 Codex，找到设置里的额度查看入口，能看到当前账号的使用情况——还剩多少、什么时候刷新、速率限制状态。入口一般在左下角设置里，点开就能看到剩余用量。用完了就只能等下个周期自动刷新。

为什么要经常查？因为 Codex 不会在你快用完的时候主动提醒你。Codex 只会等你真的用完了，下一个任务发出去直接报错，告诉你额度不够。这种"突然断电"的体验很糟，尤其是你正跑着一个长任务的时候。

养成一个习惯：开一个稍微大点的任务之前，先扫一眼额度。够就跑，不够就等，或者把任务拆小。这种习惯能避免一半以上的"额度焦虑"。

如果你走的是 API Key 计费（不是普通用户的首选），查的就是余额和消耗，在 OpenAI Platform 的后台看。如果是接的国产模型，在对应厂商的后台看余额。这些非套餐路线的消耗更透明，但也更需要你主动盯着，因为按量计费跑偏了真的会爆账单。

还有一个细节：额度恢复不是瞬时的。有时候你看到"还有 1 小时恢复"，以为到点就能满血，其实恢复是渐进的，刚恢复那会儿额度很少，重活还是跑不动。安排任务的时候留点余量，别卡着点用。

31. Plus 和 Pro 的额度差得远吗

差得远，但大多数人不需要 Pro。

Plus 每月二十刀，给的是一个 5 小时滚动窗口里的推理额度，小任务够用。

Pro 要贵得多，给的是 Plus 的好几倍额度。Pro 有 5x 档（Plus 的 5 倍）

甚至更高档，数字看着吓人，但价格也吓人。Pro 5x 经常会有促销活动，期间额度再翻倍，但这种促销是临时的，过了就回到正常档位。

谁需要 Pro？只有那种把 Codex 当成生产力工具、每天重度使用、Plus 频繁触顶的人。普通人做做网页、改改文件、跑跑小任务，Plus 完全够，上 Pro 是浪费钱。

判断自己要不要升 Pro，最简单的方法是先用 Plus 跑一两周，看你会不会频繁撞额度墙。如果 Plus 的额度对你来说绑着用都跑不完，那 Pro 是浪费；如果你每隔几天就因为额度不够得歇着，那 Pro 才有意义。

还有一个隐形选项：买额外额度。Plus 和 Pro 用户额度用完了，可以单独买额外的 credits 继续用，不用升级套餐。这种适合“偶尔需要猛用一次”的人，比长期订阅 Pro 划算。

对中国用户还有一个特别的省钱路子：接国产模型。通过 CC Switch 接智谱、Kimi、DeepSeek，月费比 Plus 便宜一大截，还不用梯子。这条路是非官方的，有兼容性和稳定性风险，但对预算敏感又想持续用的人来说，是很实在的选项。

32. 额度烧完了任务会立刻停吗

不会立刻停，但会停。

额度检查通常发生在任务开始或者某些关键节点上。你正在跑的任务，大多数情况下会跑完当前这一轮，然后在你发起下一个任务、或者 Codex 要进入下一轮执行的时候，报错告诉你额度不够。

这意味着两件事。第一，正在跑的任务不用太担心被腰斩，大概率能跑完当前阶段。第二，你不能指望"刚好用完最后一滴额度把任务跑完"，因为额度耗尽的时机你控制不了，可能正好卡在你任务的中段。

更麻烦的是后台任务。你开了一个任务没盯着，关掉浏览器去干别的，任务在后台继续跑、继续消耗。等你回来发现额度早烧光了，任务可能跑了一半挂在中间，既没完成又赔了额度。这种"隐形消耗"是 Plus 用户提前烧光的主要原因之一。怎么避免？长任务跑之前先看额度，预估一下够不够。拿不准的，拆成小段跑，每段跑完看一眼额度还剩多少。不要同时开好几个后台任务，那样消耗是叠加的，烧得特别快。

万一任务跑到一半额度真没了，别硬重启。先让 Codex 停在那，等额度恢复一部分，再决定是接着跑还是重新来。强行用仅剩的一点点额度去续跑一个需要大量消耗的任务，往往跑不完还白烧。

33. 手机上能用 Codex 吗

Codex 本身是电脑端的，手机上用的是另一个东西——ChatGPT 手机端的

Codex 里的 Codex 功能。

这两者能力不一样。Codex 跑在你电脑上，能读本地文件、操控你电脑上的应用、用本地环境。手机端的 Codex 里的 Codex 走的是云端，能干的事更接近 Codex Web 那一套，处理 GitHub 项目、跑云端任务，但碰不到你电脑上的本地文件。

不过手机端有一个特别实用的场景：续跑和远程控制。2026 年 5 月 OpenAI 还专门更新过，让 ChatGPT 手机版能直接发起 Codex 任务，包括在 Windows 上跑的 Codex 任务。你在电脑上开了一个长任务，出门了，路上用手机查看进度；Codex 有问题问你，你直接在手机上回复、批准 Codex 的下一步、或者指派新方向。你电脑上的本地环境安安静静在那干活，你人可以自由移动。

这种体验是：任务在你电脑上跑，但你的自由不受限制。手机版和电脑端能完全同步过去的所有会话，比 ChatGPT 的手机版稳定得多。

所以答案是：手机不能跑 Codex 本身，但能用手机端继续参与 Codex 上启动的任务。对长任务、后台任务特别有用。如果你只是想随时随地问点小问题，手机端的 Codex 也能用，只是能力比桌面版弱一截，毕竟碰不到你电脑的东西。

34. 关掉 Codex 任务还在跑吗

分情况，看你跑的是哪种任务。

如果跑的是本地任务（Local 模式或者 Worktree 模式），任务直接依赖你电脑上的 Codex 进程。你把 Codex 完全退出（注意是完全退出，不是最小化），任务就停了。这是本地任务的特点：任务在你的机器上跑，机器或者 Codex 关了，任务就没了。

这里有个坑：很多人以为关掉窗口就是退出 Codex，其实 Codex 默认是最小化到系统托盘（Windows）或者菜单栏（Mac），Codex 还在后台跑。你以为关了，其实没关，任务还在继续，额度还在烧。要真正退出，得去托盘图标右键选 Quit，或者菜单栏里选退出。

如果跑的是云端任务（Cloud 模式），那就不一样了。云端任务跑在 OpenAI 的服务器上，你电脑关机、Codex 退出都不影响，任务照样在云端推进。这是云端模式最大的价值——不依赖你的机器在线。

所以判断“关掉 Codex 任务还在不在跑”，先看你跑的是哪种模式。本地模式，关 Codex 等于停任务；云端模式，关 Codex 任务继续。拿不准的话，最稳的做法是：跑长任务或者要离开电脑的时候，用云端模式；日常短任务用本地模式，用完就退。

养成一个习惯：不用的时候彻底退出 Codex，别让 Codex 最小化在后台偷偷消耗。尤其是 Plus 用户，额度宝贵，被后台任务悄无声息地烧掉最亏。

35. Codex Credits 值得买吗

看你是什么用法，分两种人。

一类人，平时 Plus 额度够用，只是偶尔某几天需要猛用一次。这种人值得买 credits。Plus 的月度额度是固定的，超额了只能等下个月。如果你这个月临时有个大项目要赶，买点 credits 顶一下，比为了这几天升级到 Pro 划算得多。用完就停，下个月回到 Plus 节奏，灵活。

另一类人，几乎每个月都把 Plus 额度用光，经常被额度卡住。这种人不该买 credits，该升 Pro。如果你每个月都要补 credits，累计下来可能比 Pro 还贵，而且体验断断续续——额度一断任务就停，补了 credits 又得重接上下文，效率低。这种情况直接上 Pro，额度池子大，连续干活不卡壳，长期更省心。

判断自己是哪种人，跑一两个月就有数。如果你发现自己这个月买了 credits，下个月又买，连续两三个月都这样，那就是该升 Pro 的信号。如果只是偶发性的需求，credits 是最灵活的补充。

还有一个隐性成本要算：买 credits 之后心理上容易"放开用"，本来该省着用的也开始大手大脚，结果 credits 也烧得快。买之前先问问自己，是真有那么多活要干，还是只是用得更随意了。前者值，后者是白烧钱。

三、踩坑与反模式

36. 为什么 Codex 突然开始原地打转

这是社区里被吐槽最多的一种翻车，几乎每个人都遇到过。

现象是这样的：你给 Codex 一个任务，Codex 开始干，跑了会儿报错，Codex 自己改了一版再跑，又报错，再改，再跑.....这么循环好几轮，每一轮的错误信息都差不多，但就是过不去。你看着 Codex 在那转圈，等了十几分钟还在原地，额度哗哗往下掉。

为什么会这样？因为 agent 遇到失败的时候，Codex 的策略是"换个思路再试"。但有时候 Codex 换的"新思路"其实和上一次差不多，或者 Codex 压根没意识到自己在重复。Codex 没有人类那种"我刚才已经试过这个了，这条路明显不通"的元认知。结果就是 Codex 卡在同一个死胡同里反复撞墙。

社区管这种现象叫"无限乱试"或者 endless loop。Reddit 上甚至有人开帖吐槽，说 Codex 突然变得"几乎不可用"，任务都变成"无尽的循环、漏掉的指令、半成品的修复"。

怎么办？及时叫停。遇到 Codex 打转的时候直接发这一句：

先停一下。请总结目前失败原因，不要继续修改文件。

让 Codex 停下来复盘，比你让 Codex 继续硬试有用得多。复盘完，要么换个完全不同的思路重来，要么缩小任务范围，要么你自己介入判断问题在哪。让 Codex 无限试下去，最后大概率还是不行，但额度肯定烧光。

养成一个习惯：盯着 Codex 的动作日志，发现连续两三轮都在做差不多的事、报差不多的错，立刻叫停，别等。

37. 任务跑了半小时最后给我个半成品

这种翻车比原地打转更隐蔽，也更气人。

现象是：任务跑得很顺，没有报错，没有打转，看起来一直在推进。跑了二十分钟、半小时，你满怀期待去验收，结果 Codex 给你的东西只做了一半——要么功能没全实现，要么某些地方明显没改完，要么代码里有半句注释没写完，像 Codex 中途被拔了电源。

为什么会这样？最常见的是上下文超长。长任务里 Codex 要读很多文件、改很多地方，对话历史越堆越长，长到一定程度，Codex 的注意力就分散了，后面的细节开始漏。另一个原因是任务边界没说清楚——Codex 理解的“完成”和你想的不一样，Codex 觉得“核心功能做完就算完了”，但你觉得“还得处理边界情况、写测试、加文档”。

最阴险的是还有一种：Codex 在中途悄悄"自以为完成了"。这是一个已知问题，Codex 有时候会在后台任务还在跑的时候，就丢给你一个"final answer"，把任务提前结束。你以为 Codex 跑了半小时，其实它可能在第 15 分钟就已经"认为"自己做完了，剩下时间都在干别的。

怎么避免？把验收标准写进任务描述里，明确说"完成的标准是 XXX，缺一样都不算完"，让 Codex 没法提前收工。长任务一定要拆成小段，每段跑完验收一次，别让 Codex 一口气跑半小时。还有，跑完之后别只看 Codex 文字总结里那句"已完成"，要打开 diff、看实际产物，到底完成到什么程度，自己确认。

38. 明明写清楚了 Codex 为什么还是漏指令

这是最让人崩溃的一种翻车：你把要求一条条列得清清楚楚，Codex 还是漏了几条。

你写"1. 改 A, 2. 改 B, 3. 改 C, 4. 加测试, 5. 不要动 D"。Codex 跑完，A 和 B 改了，C 漏了，测试没加，D 还顺手给你改了。你回去看你的描述，明明写得明明白白，Codex 就是不全做到。

为什么？因为 agent 处理长指令的时候，注意力不是均匀分布的。开头几条和结尾几条 Codex 记得牢，中间的容易漏。多条指令堆在一起，Codex 倾向于先抓"Codex 觉得最重要"的几条做，剩下的可能忽略。社区里反馈 Codex 突

然变成 "endless loops, missed instructions, half-finished fixes", 漏指令就是这个 missed instructions。

还有一个原因：指令越多，Codex 越倾向于"先把能快速搞定的做完给你看"，剩下的打算回头再补——然后回头就忘了。Codex 没有故意漏，它的工作记忆就那么大，长列表里中间几条自然就被挤掉了。

怎么治这个毛病？编排上有点讲究：重要指令放开头和结尾，中间放次要的，别堆成一片。指令数量也要克制，一个任务四五条要求是上限，再多就拆成两个任务跑。最有用的是计划模式，让 Codex 先把要做的步骤列出来，你一眼就能看出有没有漏掉哪条，确认了再执行。跑完之后，别忘了拿你的原始要求逐条对一遍，发现漏的让 Codex 补，别默认接受。

最有效的还是计划模式。先让 Codex 出计划，计划里缺了你的某条要求，你立刻就能发现并补上，比等 Codex 跑完再发现漏了强十倍。

39. Codex 改着改着把好代码也改坏了

这种翻车叫"顺手重构"，是头号反模式。

现象是：你让 Codex 改 A 功能，Codex 改完 A，顺手觉得 B 那段代码"不够优雅"，给你重构了；重构完 B，又觉得 C 的命名不规范，给你改了命名；改完命名，引用 C 的地方全得跟着改，D、E、F 一连串全动了。你以为让

Codex 改一个地方，回头一看 Codex 改了二十几个文件，其中一半和你的任务根本没关系。

为什么 Codex 会这样？因为 agent 有一种"优化倾向"。Codex 看到代码，会本能地觉得"这里可以更好"，然后就去改了。Codex 不区分"你让我改的"和"我顺手想改的"，在 Codex 眼里都是改动。很多老项目本身就有 lint 或类型问题，Codex 觉得某些代码"不够优雅"就会顺手重构，可能带来重命名导致引用出错、改全局样式影响其他页面、升级依赖引发兼容问题，甚至删除 Codex 认为无用但实际是业务逻辑的代码。

后果很严重。重命名一个组件，所有引用这个组件的地方都得跟着改，改错一处就报错。改全局样式，可能让别的页面布局崩掉。最阴险的是删代码——Codex 觉得"这段没用"，其实是你的业务逻辑，删了你都不知道哪坏了。

怎么防御？把这段固定的话术写进任务描述或者 AGENTS.md 里：

本次只实现当前功能点。

不要顺手重构无关代码。

不要修改命名、目录结构、依赖版本和全局配置。

如果你发现代码可以优化，请先记录为建议，不要直接修改。

这段话能把绝大多数顺手重构挡住。再加一条更狠的铁律，专门针对删代码：

不要删除你没有创建的文件。遇到报错的代码，修复它，不要删掉它。

这条特别重要。Codex 遇到报错时有一种危险倾向：删掉报错的东西来消除报错，哪怕那东西跟当前任务无关。社区里有人反馈，Codex 在做一个小改动时，把一个无关的模块整个删了，理由是"它在报错"。把"禁止删除你没创建的文件"写进 AGENTS.md，能把这种最阴险的翻车挡住。

最后一道防线：每次跑完看 diff，发现 Codex 改了和任务无关的文件，立刻让 Codex 回退那部分改动，别默认接受。

40. 为什么简单需求 Codex 非要绕一大圈

你让 Codex"在这个页面加个按钮"，听上去多简单的事，Codex 跑完给你改了十几个文件，建了三个新模块，引入了两个新依赖，折腾了十五分钟。你看着 diff 满头问号：我就让你加个按钮，你这是干嘛？

这是 agent 的另一种常见毛病：过度工程化。Reddit 上经常能看到反馈，说 Codex 经常"过度复杂化简单的任务"，哪怕你给了详细的需求文档，Codex 还是把简单的分类、组织工作搞得绕来绕去。

为什么？根子在于 Codex 见过太多"好实践"了。训练数据里全是各种设计模式、最佳实践、架构思想，所以看到任何一个需求，Codex 本能地想"用更专业的方式来做"。加个按钮？Codex 觉得应该封装成可复用组件。做个简单的数据展示？

Codex 觉得应该引入状态管理。Codex 没在故意添乱，它是真心觉得这样"更好"。

更麻烦的是，Codex 不会主动问你"要不要简单点"。它默认你要的是 production-ready 的方案，就照生产标准来做了。但很多时候你只想快速做个原型，或者改个小东西，根本用不着那么讲究。

怎么办？在任务描述里明确说清楚你的预期。常用的话术：

用最简单的方式实现，不要引入新依赖，不要新增模块，不要做超出本需求范围的封装。

或者：

这是一个原型/演示，不需要生产级的架构，能跑就行。

把"简单"这个要求显式说出来，能治大部分过度工程化。另外，看到 Codex 跑完给了一堆你意料之外的文件改动，先别急着接受，问 Codex"为什么需要这些改动，能不能简化"。多数情况下 Codex 能收敛到更小的改动范围。

41. 后台任务是不是在偷偷烧我额度

很可能是。这是 Plus 用户提前烧光额度的头号原因。

现象是这样的：你开了一个任务，跑了一会儿觉得无聊或者要去干别的，关掉浏览器或者切换到别的应用。你以为自己"没在用"，其实任务还在后台跑，每跑一轮都在消耗额度。等你过两小时回来，发现额度掉了一大块，任务可能还挂在半路。

这种"隐形消耗"现象确实存在。有报道说 Codex 的后台进程导致用户"比预期更快地撞到限制"，OpenAI 的工程负责人当时承认这是问题，后续在修。但截至现在，后台任务消耗额度这件事本身没变，变的是界面提示更明显了些。

更阴险的是另一种情况：你开了一个 thread，跑了任务，任务完成了，你以为这个 thread 就"停下"了。但其实如果你开了自动化或者某些持续运行的任务，可能在你看不见的地方继续消耗。

怎么避免？养成几个习惯就行。最关键的一条：不用的时候彻底退出 Codex，注意是真的退出，不是最小化——Windows 要去托盘图标右键选退出，Mac 要在菜单栏里退出。最小化到后台的 Codex 还在跑，额度还在掉。另外，长任务跑之前先看一眼额度，估算够不够撑到任务结束；平时定期查额度，发现掉得比预期快，就排查是不是有后台任务在偷跑。

这里头有个意识得扭过来：Codex 没有"打开才花钱，关掉就免费"这回事。只要 Codex 活着、有任务在跑，就在花钱。不用的时候彻底关掉，是最简单也最有效的省钱办法。

42. 为什么关掉窗口额度还在掉

这是 Q34「关掉 Codex 任务还在跑吗」的另一种痛：你以为自己关了，其实没关。

Codex 默认的行为是最小化到系统托盘（Windows）或者菜单栏（Mac），不是真正退出。你点窗口右上角的叉号，窗口消失了，你以为 Codex 关了，其实 Codex 还在后台活着。后台活着的 Codex，如果你之前跑了任务没结束，那个任务还在继续跑、继续消耗额度。

很多人就是这样烧钱的：开了个任务，等得不耐烦，叉掉窗口去干别的，以为"我都关了应该不烧了"。结果第二天发现额度没了，任务跑了个乱七八糟挂在半路。这种亏吃一次就够肉疼了。

区分"关窗口"和"退出 Codex"的办法。Windows 上，点叉号只是关窗口，要去右下角系统托盘找到 Codex 图标，右键选 Quit 或者退出，才是真的退出。

Mac 上，点红点是关窗口，要在菜单栏里选"退出 Codex"，或者按

Command+Q，才是真的退出。养成一个判断习惯：每次想关的时候，问自己"我是关了窗口，还是退出了程序"，这两个动作差别巨大。

如果你确实想让长任务在后台继续跑，请回到 Q34 看本地模式和云端模式的区别——只有云端模式做得到"关掉电脑任务继续"，本地模式做不到。

43. 为什么我开的任务最后没结果就消失了

这种翻车特别让人火大：你开个任务，跑了一会儿，回头一看，任务没了，没有结果，没有报错，就像从来没存在过。

几种可能。最常见的是任务被意外终止——前面那个已知问题，Codex 在后台任务还在跑的时候丢给你一个"final answer"，把自己结束了。你以为 Codex 还在跑，其实它早就"认为"做完了，悄悄退出了。

也可能是 Codex 崩溃或者被系统杀掉了。本地任务依赖 Codex 进程，Codex 一挂任务就跟着没。如果你电脑内存紧张、系统在清理进程、或者 Codex 本身有 bug 崩了，正在跑的任务会一起消失，连中间结果都来不及保存。

还有一类是环境掉了：登录过期、网络断开、连不上服务器，任务可能直接中断，又没有恢复机制。最后就是额度跑光，跑到一半发现额度不够直接停——这种通常会提示一下，但有时候提示不明显，你以为任务"消失"了。

怎么减少这种事？重要的长任务用云端模式跑，不依赖本地 Codex 进程，Codex 崩了也不影响。任务跑完要立刻验收、立刻把结果存到项目文件夹，别让 Codex 停留在"只在对话里"的状态。登录和网络保持稳定，掉线立刻重连。

另外养成看任务列表的习惯，发现任务莫名其妙没了，就去翻动作日志和系统通知，看有没有报错。

44. 同样的任务再跑一次结果完全不一样

这是 agent 的"特性", 不是 bug, 但你得知道它存在, 不然会被坑。

你今天让 Codex"做个登录页", Codex 给了你一版 A。明天你把任务原封不动再发一遍, Codex 给了你完全不同的 B。A 和 B 都能跑, 但风格、结构、用的库、甚至技术选型都不一样。你懵了: 同一个任务, 怎么差别这么大?

原因是 agent 的输出有随机性。Codex 每次生成的时候, 内部有一定的"采样"成分, 同样的输入会走到不同的生成路径。再加上模型版本可能更新、上下文可能略有不同、Codex 当次的"判断"可能不一样, 结果就是同一个任务每次跑出来都不完全相同。

这个特性带来的麻烦很具体。比如你看到一版很好的结果, 想"再跑一次看看有没有更好的", 结果第二版差很多, 想回到第一版又回不去 (除非你提前存了)。再比如协作场景, 你和同事分别跑同一个任务, 拿到的结果完全对不上, 连"到底用哪版"都没法讨论。

应对办法也很直接。好的结果立刻保存, 别想着"再跑一次说不定更好"——把成果复制到项目文件夹、提交到 git, 定格下来。想对比多个方案, 用多方案预览功能, 让 Codex 一次性给你几个方案对比, 比分别跑三次靠谱得多。对复现性要求高的任务, 把任务描述写到极致详细, 因为描述越模糊, 每次跑差异越大; 描述越精确, 差异越小。

记住一条：agent 的输出是"一次性"的，跑出来的东西不保存就没了。别指望 Codex 能"原样再来一遍"。

45. Codex 给的方案为啥和我项目对不上

这种翻车经常发生在"贴代码"场景：你把一段代码贴给 Codex 让 Codex 改，Codex 改完给你一版，你高高兴兴贴回项目，一跑全是报错。

为什么？因为 Codex 看不到你项目的全貌。你贴给 Codex 的只是那一段代码，Codex 不知道这段代码周围还有什么、依赖什么、调用了什么。Codex 改的时候，只能按 Codex 对"一般代码"的理解来改，跟你项目里这段代码的实际上下文对不上。结果就是 Codex 改的地方和你项目里的别处打架，引用错、类型错、依赖找不到，全是这种问题。

如果你不用项目模式，只是贴代码让 Codex 改，Codex 验证的那个环境和你的真实项目隔了一层，结果的可信度打折扣。这就是为什么要强调用项目模式——让 Codex 进入你的项目目录，读到你所有的文件，Codex 改的时候才知道周围什么样。

怎么避免这种翻车？能用项目模式就用项目模式，让 Codex 读到你项目的全貌，别只让 Codex 看你贴的那一段。如果不得不贴代码（比如项目不方便共享），那就把相关的依赖、import、上下文一起贴上，别只给一段孤零零的代码。

Codex 改完之后，记得在你的项目里实际跑一遍——Codex 能给你"看起来对"的代码，但你项目里跑不跑得通，它自己并不知道（因为它没在你项目里跑过）。最根本的还是项目模式。很多 Codex 任务失败，原因往往不在 Codex 不会写代码，而在 Codex 还没理解项目就被要求直接动手。让 Codex 先读项目，再动手，翻车率会大幅下降。

46. Codex 说自己改好了怎么一跑全是报错

这种翻车的根源是：Codex 跑通的环境，和你跑的环境，不是同一个环境。

Codex 在沙箱里跑通了，那个沙箱有 Codex 自己装的依赖、Codex 自己的运行环境、Codex 自己假设的项目结构。Codex"跑通"是在那个环境里跑通的。

你拿到 Codex 的代码，放到你真实的项目里，你的环境可能不一样：依赖版本不同、操作系统不同、配置不同、别的代码和 Codex 的改动有冲突。结果就是 Codex 说"跑通了"，你一跑全是错。

有真实的吐槽：有人让 Codex 干个简单的活，Codex 连代码库用什么语言都没识别对，一个简单任务直接失败。这种就是典型的"Codex 没真正进到你环境里就开干"。

更麻烦的是，Codex 有时候根本没真的跑，只是"觉得"自己跑通了。Codex 生成完代码，看着代码觉得"这样写应该没问题"，然后告诉你"已完成"。你没验证，拿去跑，发现根本没跑过。

怎么治？关键就一条：让 Codex 在你项目里跑，别让 Codex 在自己的沙箱里自说自话。用项目模式，让 Codex 真的进到你项目目录，这样 Codex 跑通的就是你项目的环境。跑完之后你自己再跑一遍，别全信 Codex 说的"跑通了"。如果报错，就把真实报错信息直接发回给 Codex，让 Codex 看到现场，别让 Codex 自己脑补。还有，如果 Codex 反复在同一类报错上栽跟头，多半是任务描述不够清楚，或者任务本身超出了它的能力，那就考虑拆小或者换思路。

47. 哪些信号说明这个任务该手动叫停

及时叫停是省额度和防翻车的核心技能，比会启动任务更重要。

几个明确的信号，看到任何一个都该停。最典型的就是原地打转——连续几轮报差不多的错、做差不多的尝试，没有实质进展，这种等再久也没用，立刻叫停。

文件数量也能透露问题。你让 Codex 改一个功能，它改了二十几个文件，明显是在顺手重构或者跑偏了，看到 diff 文件数飙升，先暂停看看它到底在干什么。更危险的是 Codex 开始删大量代码——除非你明确让 Codex 重构或者清理，

否则任何删代码的举动都要警惕，Codex 觉得"没用"的代码，往往是你的业务逻辑。

还有几个信号同样要紧。Codex 引入了新依赖、改了 `package.json`、动了配置文件，你都得问一句"为什么需要这个"。任务跑了很久还没完成、也没明确进展，本来几分钟该完的事拖到十几分钟还在继续，多半是陷入循环或者卡住了。最后还有一种尤其要命：Codex 的文字总结和实际 `diff` 对不上。它嘴上说"修改了登录逻辑"，`diff` 里压根没动登录相关的文件——这种说一套做一套的情况，要立刻叫停问清楚。

遇到任何不对劲都可以发这一句：

先停一下。请总结目前做了什么，遇到了什么问题，不要继续修改文件。

让 Codex 停下来汇报，比你让 Codex 继续试更省时间和额度。记住一个原则：宁可多叫停几次重来，也别让 Codex 一个跑偏的任务跑半小时。前者浪费几分钟，后者浪费十几分钟和一大块额度。

48. 任务失败了我该重试还是改需求

这两种应对方式差别很大，选错了会越搞越糟。

先说什么时候该重试。如果失败是偶发性的——比如网络断了、Codex 崩了、某个临时报错——直接重试就行，原任务不用改。这种是环境或者运气问题，再来一次大概率能成。

什么时候该改需求。如果失败是"Codex 试了好几次都做不到"，那就别再当运气问题看了，根子在任务本身：可能是任务太大、描述太模糊，或者超出了 Codex 的能力范围。这种情况下直接重试，多半还是失败，白烧额度。正确的做法是停下来，分析为什么失败，然后调整任务。

调整任务常见就几个方向。最常用的是拆小，把大任务拆成几个小步骤，一步一步来，每步成功率高。再把需求说清楚，把模糊的地方具体化、把验收标准明确化。范围也要收紧，别让 Codex"重构整个模块"，让 Codex"只改这一个函数"。如果 Codex 反复在同一种解法上失败，可能是这条思路本身不适合，那就换一个完全不同的方向。

有一个特别重要的习惯：失败的时候让 Codex 先总结原因，而不是继续试。发一句"先停一下，总结目前失败原因，不要继续修改文件"，让 Codex 给你一个失败分析。这个分析能帮你判断是该重试还是该改需求——如果 Codex 说的是"网络问题""临时错误"，重试；如果 Codex 说的是"这个需求涉及的模块太复杂""依赖的 API 不存在"，那必须改需求。

最忌讳的是"无脑重试"。看到失败就直接点重新运行，不改任何东西，结果大概率还是失败，连续失败几次额度就烧光了。每次失败都先停一下想想为什么，再决定下一步。

49. 为什么大任务拆开反而更容易废

这个反直觉的现象坑过很多人：你把一个大任务"合理地"拆成几个小任务，本以为是稳妥的做法，结果每个小任务都跑得不顺，连起来更乱，还不如一开始让 Codex 整体干。

为什么拆了反而废？最常见的是拆错了边界。你以为 A、B、C 是三个独立的小任务，其实它们之间有耦合。A 改完影响了 B 依赖的东西，B 跑的时候还基于 A 改之前的状态，结果 B 的改动和 A 冲突——这种"拆了但没拆干净"比不拆还糟。

另一个坑是上下文丢失。大任务一次性跑，Codex 从头到尾待在同一个上下文里，记得前面做了什么；拆成小任务分别跑，每个任务都是"新开始"，Codex 不知道前面干过什么，可能重复改动，也可能和前面矛盾。再加上拆得太碎，Codex 做每个小任务时只盯着这一小块，看不到整体，结果每个小块单独看还行，拼起来风格不统一、结构不一致、甚至逻辑冲突。

怎么拆才对？关键原则是按耦合程度拆，别按"功能模块"拆，紧密耦合的部分放一个任务里，不要硬拆。拆完之后还要给 Codex 整体上下文：每个小任务开始前告诉它"前面已经做了什么、现在要做的是什么、后面还要做什么"，让 Codex 有整体感。更稳妥的做法是用计划模式先规划整体，让 Codex 自己提出怎么拆，你看它的拆法合不合理，比你硬拆靠谱。最后记得每拆完一段就验收一段，发现不对立刻调整，别等三个小任务全跑完才发现整体乱了。

拆任务本身没错，错的是"机械拆分"。会拆的人能让复杂任务变简单，不会拆的人能让本来能干的任务变得干不成。

50. 怎么避免一个任务把整周额度拖垮

这是 Plus 用户必须学会的自我保护，不然一个跑偏的任务能让你一周都没法用。

危险是怎么发生的。你开了一个看起来不算大的任务，Codex 跑着跑着陷入循环，每多一轮都在烧额度；你忘了盯着 Codex，去做别的事，半小时回来发现额度掉了一大块；你想止损但又不甘心，再让 Codex 试一次，又烧一波；最后任务没成，额度也快没了，剩下几天只能干瞪眼。

避免的办法，核心是给任务设一道止损线。开任务前先预估成本，看任务复杂度，估算大概要跑几轮、每轮消耗多少，心里有个数。再给任务设个时间上限，比如

跑超过 10 分钟还没结果就强制叫停，不要无脑等。盯动作日志也很关键，发现 Codex 在打转、在试错、在反复失败，立刻停，别指望它自己想明白。

另一头是"拆和省"。大任务一定要拆小，分多次跑，每次跑完看额度还剩多少，够再跑下一段，不够就先等。小任务用低推理强度，省额度；动手之前用计划模式先规划，规划阶段消耗少，规划对了，执行阶段才不浪费。

第三头是"监控"。经常查额度，发现掉得快立刻排查；不用的时候彻底退出 Codex，不让后台任务偷跑；后台同时只跑一个重任务，多个叠加起来消耗是翻倍的。

最根本的心态调整：不要把 Codex 当成"丢进去等结果"的工具，要当成"需要你看着的工具"。Codex 跑任务的时候，你最好在旁边盯着，发现不对立刻介入。完全放手的任务，要么是云端模式（你不怕 Codex 跑偏），要么是你已经验证过 Codex 能稳定完成的 routine 任务。新任务、复杂任务，一定要盯。

51. 用 Codex 最容易翻车的人都是哪几条没做到

把上面这些踩坑读完，你会发现真正救命的招其实就那么几条。社区里翻车最多的人，基本都是这几条底线没守住。整理成一份最低标准清单，照着做能挡住八成以上的灾难。

最要紧的一条是：每次让 **Codex** 动手之前，先给你的项目存一个版本。如果你用 **Git**，就是先提交一次再让 **Codex** 干活；如果你不用 **Git**，至少把整个项目文件夹复制一份留底。**Codex** 改崩了你还能 **reset**、还能 **checkout** 回来，**Codex** 删了关键代码你还能从备份里捞回来。没有这层"撤销键"，让 **Codex** 改项目等于不系安全带上高速。

再有一条是：稍微大点的改动，都用 **Worktree** 模式跑。**Worktree** 相当于给 **Codex** 一本独立的草稿本，让 **Codex** 在副本上大胆改，你的主项目纹丝不动。改坏了直接丢掉副本，零风险。重构、加新功能、不确定结果的尝试，都该用 **Worktree**。

还有一条是：在 **AGENTS.md** 里写死"禁止删除你没创建的文件"。**Codex** 遇到报错的代码，倾向删掉报错的东西来消除报错——哪怕那东西跟当前任务无关。

这条铁律把这种最阴险的行为挡住。

再就是：别让 **Codex** 跑无人监督的大动作。改名、清理、大规模重构、迁移——这类任务风险最高，最容易被 **Codex** 的"优化倾向"带跑，必须全程盯着。

最后：小步快跑，每步都看 **diff**。把大任务拆成小的、边界清楚的变更，每一步跑完立刻看 **diff**。这个习惯能把绝大多数翻车挡在合并之前。

这几条没有任何一条是高深技巧，全是底线操作。社区里那些把 Codex 用得风生水起的人，并没有掌握什么秘密武器，他们只是把这几条做到了骨子里。反过来，社区里翻车最惨的案例，几乎都能从这几条里找到违反的地方。

四、进阶：Skill·MCP·插件·自动化·操控电脑

52. Skill、MCP、插件到底有什么区别

这三个词是新手绕不过去的坎，混在一起根本分不清谁是谁。用大白话讲一遍。

插件是能力安装包。你装一个插件，Codex 就多了一个新能力。比如装了 Presentations 插件，Codex 就会做 PPT；装了 Spreadsheets 插件，Codex 就会处理表格；装了 Chrome 插件，Codex 就能操作你已登录的浏览器。插件解决的是"能不能做某类事"的问题。

Skill 是 workflow 说明书。Skill 的作用不是给 Codex 加新能力，Skill 把"某件事怎么做"的方法固化下来。比如你有一套自己整理数据的标准流程，写成 Skill 之后，下次只要一句话 Codex 就按你的流程来，不用每次重新解释。Skill 解决的是"同类任务怎么做"的问题。

MCP 是连接外部工具的通道。MCP 的作用是让 Codex 能读到外面的数据和工具——你的 GitHub、你的飞书、你的 Notion、你的数据库。MCP 解决的是"连接什么外部资源"的问题。

一句话记忆："怎么做"用 Skill，"连什么"用 MCP，"打包的能力"用插件。

普通人的上手顺序应该是：先玩明白内置能力和插件（这一步就够你做大部分事了），等你有重复性的 workflow 想固化了，再去碰 Skill；等你需要让 Codex 读外部数据了，再去研究 MCP。判断标准很简单：普通写代码不一定需要 MCP；需要让 Codex 访问外部工具或外部数据时才考虑 MCP。

53. 怎么找到并安装一个好用的插件

插件不是越多越好，按需装才对。

Codex 里有个插件页，打开能看到所有可用的插件，分类很清楚。装的方式很简单，找到你想要的，点 Add to Codex 或者 Install，按提示走完授权就行。常见的插件有几大类：浏览器类（Chrome、内置浏览器）、办公类（Documents、Presentations、Spreadsheets）、开发类（GitHub、Computer Use）、内容创作类（HyperFrames、Remotion）。

新手最容易犯的错是“看到就装”。刚开始只用官方已经内置好的插件，不要急着装一堆，后续熟练了再慢慢添加。装太多你用不上的插件，只会让界面变乱、增加上下文消耗（每个插件都占一点），还可能引入兼容性问题。

正确的做法是按需装。你明确知道“我要让 Codex 做 PPT”，那就去装 Presentations；你明确知道“我要让 Codex 读我的 GitHub”，那就装 GitHub。没有明确需求的时候，别为了“以后可能用得上”而装。

还有一个判断标准：先想清楚你要做什么事，再去找对应的插件。而不是反过来——看到一堆插件不知道装哪个。等你明确知道"我要让 Codex 连接某个工具"，再去插件页找。这种"需求驱动"的安装方式，能让你装的每个插件都用得上。

装完插件之后，调用方式是在对话框里用 @ 命令。比如 @Chrome 调用 Chrome 插件，@Computer 调用电脑操控。打一个 @ 就能看到所有已安装的插件列表，选你要用的就行。

54. 自己怎么创建一个 Skill

创建 Skill 不需要懂技术，会用自然语言说话就行。

最简单的创建方式是这样的：你先让 Codex 完整跑一遍你想固化的流程，跑出来的结果你满意，然后跟 Codex 说"把这个流程变成一个 Skill"。Codex 会自动把你刚才的流程整理成一个 SKILL.md 文件保存下来，下次你就能快速调用。

更直接的方式是用 skill-creator。在对话框里输入 \$skill-creator，然后像跟同事说话一样，用自然语言描述你的流程：第一步做什么、第二步做什么、最后输出什么。Codex 会帮你把这个描述整理成结构化的 Skill 文件。这种方式不需要知道技术细节，直接把自己的流程讲出来就行，不需要说得严丝合缝，Codex 能理解，理解不了的地方再交互几轮。

很多人觉得 Skill 这个词听起来很高级、很复杂，不敢碰。其实 Skill 就是个 markdown 文件，里面写清楚"什么场景用、目标是什么、按什么步骤做、输出什么格式"。Skill 的价值在于：把只在"你脑子里"的经验，变成"Codex 能反复执行"的标准流程。

做一件事的流程很重要——先做 A 再做 B，还是先做 B 再做 A，结果差很多。做饭的时候，炒完菜立刻擦灶台，趁油没凝结，比吃完饭再收拾省力得多；做饭的过程中，趁着间隙顺带把后面不再用的餐具清洗了，会减轻饭后的清理负担。Skill 就是把你做事的"流程"沉淀下来，让 Codex 按你这个高效流程来，而不是 Codex 自己随便发挥。

什么时候该做 Skill？标准很明确：只做一次的任务用提示词，经常重复做的任务用 Skill。如果你每个月都要做一次月度汇报，每次都重复解释一遍流程，那就把流程做成 Skill；如果只是临时做一次的事，直接说就行，不用专门做 Skill。

55. Skill 怎么写才能被自动触发

很多人做了 Skill，发现 Skill 从来没有被自动调用过，每次都得手动 `$skill-name` 去喊，觉得 Skill 不好用。问题出在 Skill 的描述上。

Skill 能不能被自动触发，关键在 Skill 的 description（描述）字段。

Codex 决定要不要用某个 Skill，是看你的任务描述和 Skill 的描述匹不匹配。描述写得模糊，永远匹配不上；描述写得清楚，Codex 就能在你说话的时候自动判断"这个任务该用那个 Skill"。

一句话总结：**Codex** 仅根据描述文本来匹配你的任务，以便隐式调用。把关键用例和触发词放在前面；模糊的描述永远不会触发。

什么样的描述是好的？举个对比。坏的描述："这是一个数据分析的 Skill。"—太模糊，Codex 不知道什么场景该用。好的描述："通过读取飞书多维表格里
的回款数据，按产品和渠道分类汇总，标记异常值，输出一份带图表的 Excel
周报。每当我要求'回款对账'或'周报'时触发。"—场景、动作、输出、触发词
都写清楚了。

写描述有几个要点。先把最常见的触发场景放开头，让 Codex 一眼看到这个 Skill 是干什么的；再把触发词写清楚，明确说"每当我说 X 或 Y 时触发"；
输入什么、输出什么也要交代明白，Codex 才知道要读什么、给什么。还有一点容易忽略：避免笼统的词。"数据分析""内容创作"这种太宽泛，要具体到"飞
书回款对账""公众号文章生成"这种窄场景。

自动匹配之外，还有两种方式可以兜底。一种是显式调用，在对话里输入

\$skill-name 直接把 Skill 喊出来，不依赖自动匹配；另一种是 @ 提及，和

引用文件一个语法。这两种方式确保即使自动匹配没生效，你也能手动调用。但前提还是描述写清楚，不然你连该喊哪个 Skill 都不知道。

56. MCP 什么时候才需要碰

普通人大部分场景都不需要碰 MCP。但有几个特定场景，MCP 是绕不过去的。

先说什么时候不需要。如果你只是让 Codex 处理本地文件、做网页、做 PPT、剪视频、分析 PDF，这些用内置能力和插件就够了，不用 MCP。MCP 没有"用了更高级"这一说，它就是个"特定场景才需要"的工具。

什么时候需要？当你要让 Codex 读外面的数据或者用外面的工具的时候。最常见的是几类活：比如你要让 Codex 读你的 GitHub 仓库、创建分支、发起 PR、评论 issue，用 GitHub MCP，Codex 就能直接操作，不用你手动搬运；再比如你要让 Codex 读 Notion、飞书、Linear 里的文档和任务，对应接上 MCP，Codex 就能直接读到，省得你复制粘贴；还有一类是让 Codex 读某个数据库、内部 API 或某个 SaaS 工具的数据，只要对应服务有 MCP，接上就能用。

MCP 干的事就是"给 Codex 接外部工具的插座"。生活化的类比：Codex 是会干活的人，MCP 是给 Codex 接工具的插座，MCP Server 是工具箱，Tool 是具体工具。你要让 Codex 干的活只在 Codex 自己脑子里和本地文件里，不用接插座；你要让 Codex 干的活涉及外面的数据和工具，就得接插座。

如果判断下来你确实需要 MCP，配置不算难。Codex 里有 MCP 管理界面，按提示添加 MCP Server，填好启动命令和凭证就行。常用的 MCP Server 都有现成的，比如 GitHub、Notion、飞书、各种数据库。装完之后，Codex 在对话里就能直接调用这些外部工具。

记住一条：MCP 是"按需启用"的东西，不是"装了显得专业"的东西。没需求别碰，有需求再研究。

57. AGENTS.md 到底要不要写

要写，而且这是把 Codex 用得稳的关键配置之一。

AGENTS.md 是写给 Codex 看的项目规则说明书。每次你在这个项目里开新对话，Codex 都会先读这个文件，知道你是谁、项目是什么、有什么规矩、什么不能动。AGENTS.md 是 Codex 中最被低估的文件。

不写 AGENTS.md 会怎样？每次开新对话，Codex 都"重新认识"你的项目，每次给你的答案都不一样，每次都得你重新解释一遍背景。"没有 AGENTS.md"是让 Codex 只发挥 3% 潜力的坏习惯之首。

写 AGENTS.md 不难，也不用从头写。聪明的做法：用简单的话告诉 Codex 你的项目目标，让 Codex 帮你起草一个 AGENTS.md。你会得到一个结构完整的

文件，然后再根据自己的需要修改。这样比从头写快得多、完整得多、不容易遗漏。

一个好的 `AGENTS.md` 应该包含什么？大致四块内容。开头是背景，一段话讲清你是谁、为什么做这个项目；接着是目标，一段话描述你要的最终状态；然后是约束条件，列成清单，写明硬性规则，比如用什么技术栈、不能用什么、安全边界、输出格式；最后是工作习惯，也是清单，比如“改代码前先确认计划”“不要顺手重构”“把失败经验记录下来”。

放哪里？项目根目录放一个 `AGENTS.md`，整个项目通用；子目录放 `AGENTS.md`，那个子模块的专属规则；用户级（`~/ .codex/AGENTS.md`）放一个，是你所有项目通用的个人规则，比如全局禁止某些危险命令。这几层叠加生效。

对普通人来说，最小的 `AGENTS.md` 只要写清楚三件事就够用：这个项目是做什么的、用什么语言/风格回答你、哪些东西绝对不能动。写一次，后续每次对话都省事。再往里的高阶玩法，放到 Q68「`AGENTS.md` 有哪些高价值的进阶套路」里专门讲。

58. 任务干预和任务排队怎么用

这两个功能是让你在任务跑的过程中保持控制权的关键，被叫做“人机合一的掌控感”。

任务干预 (Steering) 是中途打断 Codex。当你发现 Codex 跑偏了、要撞南墙了，不用等 Codex 跑完，直接开口打断 Codex，给 Codex 新方向。比如 Codex 正在审查一个网页，你看着侧边栏发现某个元素间距不对，直接说"这两个元素间距看着不对，调小一点"。Codex 会立刻收到你的指令，调整当前的工作。

任务排队 (Queuing) 是给 Codex 排后面的活。Codex 正在干 A，你想让 Codex 干完 A 接着干 B，不用等 A 完成再说，现在就可以告诉 Codex"等这个干完之后，把预览链接发到 Slack"。Codex 会把 B 排在 A 后面，A 一完成自动接 B。

两者的区别很清楚："干预"是改变 **Codex** 眼下正在做的事，"排队"是安排 **Codex** 接下来要做的事。

这两个功能为什么重要？因为不掌握这两个功能，你用 Codex 的方式就是"丢任务→等→看结果→再丢任务"，每一步都得等 Codex 完成才能做下一步，效率很低。掌握这两个功能之后，你可以在 Codex 跑的过程中持续介入、持续指挥，像一个真正在协作的同事，而不是一个你扔了任务就够的黑盒。

什么时候用干预？看到 Codex 方向错了立刻用，别等。什么时候用排队？你脑子里已经想好下一步了，但当前这一步还没完，用排队把下一步排上。

还有个小技巧：默认情况下，你在 Codex 跑的时候发新指令，新指令是排队的（按顺序执行）。如果你想插队，让 Codex 立刻处理新指令而不是等当前任务完成，要明确用"引导"功能，告诉 Codex"这个优先"。

59. 自动化任务适合干什么

自动化是把"需要定期做的事"交给 Codex 自己跑，相当于给项目请一个 AI 值班工程师。

适合做成自动化的，是那些有固定节奏、有明确流程、不需要你每次重新想的事。

最典型的是定期汇总类——每天早上让 Codex 整理一下项目状态、每周生成一次周报、每天从某个数据源拉数据做汇总，有人就每天让 Codex 给自己准备一份 AI 行业日报，全网值得关注的资讯第一时间就能看到。还有定期检查类，比如每周检查一次仓库有没有问题、每天查一下回款数据有没有异常、定期监控某个网页或邮箱有没有更新。另外续跑类也很实用，半小时后继续当前线程、每天定时回到同一个对话接着推进工作，对话流自动化会按设定的时间表定期回到同一个对话里继续干活，甚至能根据情况自己调整查岗的频率。

设置自动化不难。跑通一个复杂任务之后，直接告诉 Codex"每天早上 9 点自动执行"或者"每周一检查一次"，Codex 会自动创建自动化任务。

但有一个特别重要的提醒：自动化任务的模型选择器不会继承你活跃对话的设置。

新建的自动化任务会使用面板的默认值，那个默认值可能比你实际想用的模型更慢更便宜。如果你不手动指定模型，原本 7 分钟的任务可能突然跑 40 分钟。

所以创建自动化的时候，一定要显式指定用什么模型、什么推理强度。

自动化是把双刃剑。用得好，自动化让你睡觉的时候活也在干；用得不好，自动化在你看不见的地方跑偏、烧额度、产出垃圾。设置的时候把流程写清楚、把模型指定对、定期检查 Codex 的产出，才能真省心。

60. 操控电脑是怎么回事，安全吗

操控电脑 (Computer Use) 是 Codex 最强、也最需要谨慎的功能。这是

Codex 最被夸的能力，但同时要非常小心。

Computer Use 能干什么？打开你电脑上的应用、点击按钮、读取页面、填写表单——像一个人坐在你电脑前帮你操作。Codex 能操作 Xcode、飞书、Figma、系统设置、iOS 模拟器，任何你装了并且授权的桌面应用，Codex 都能碰。有人用 Computer Use 从零写了一份法律检索报告，自己只完成了登录几个网站的操作，搜索、探究网站结构、点击、整理资料全是 Codex 一次性完成。

为什么这么强？因为 Codex“看”得到你的屏幕。Computer Use 的工作方式是：看屏幕 → 决定点哪里 → 点击 → 等应用响应 → 再看。每一步都基于 Codex

看到的画面，所以 **Codex** 能操作任何有图形界面的应用，不需要那个应用提供 **API**。

但这里有个安全问题必须讲清楚。开了 **Computer Use**，相当于把你电脑钥匙交了一部分给 **Codex**。**Codex** 能碰你授权的所有桌面应用，包括微信、邮箱、浏览器、付费软件、公司工具、私人资料。涉及付款、账号设置、删除数据的操作，最好别让 **Codex** 碰；不用的敏感应用先关掉，不要让 **Codex** 接触到不相关的东西；每次只给 **Codex** 一个描述清楚的任务。

新手用 **Computer Use** 的最低安全标准：第一次用，只让 **Codex** 操作无风险的应用，特别记住不要让 **Codex** 操作社媒账号和微信。等你对 **Codex** 的行为模式有信心了，再逐步放开。

还有三种"操控"方式的区别，普通人最容易搞混。**Computer Use** 权限最大，能碰桌面上所有授权应用，但最慢。**Chrome 扩展** 权限中等，进入你已登录的浏览器，带着你的账号和 **Cookie**，网站会把 **Codex** 的操作当成你本人的操作。应用内浏览器 权限最小，完全隔离，不带你的任何账号信息，最适合本地开发调试。

选择原则：桌面应用相关的事用 **Computer Use**，需要你登录状态的网页任务用 **Chrome 扩展**，本地开发和公开页面调试用应用内浏览器。涉及付款、发布、

提交、删除的操作，不管用哪种方式，都务必你来确认，不要让 Codex 自动执行。

61. 三种浏览器方式到底用哪个

三种方式有重叠，新手特别容易搞混。这里把选择逻辑讲清楚。

应用内浏览器—最隔离、最安全。应用内浏览器是住在 Codex 里的一个全新浏览器，没有你的任何 Cookie、账号、扩展、历史记录，完全干净。适合的场景：打开本地开发服务器（localhost）、调试前端样式、检查不同屏幕尺寸的布局、在页面上直接标注反馈。应用内浏览器做不了需要登录的页面，但本地开发和公开页面调试最合适。

Chrome 扩展—带着你的登录状态。Chrome 扩展不会新开浏览器，Chrome 扩展会直接进入你已经登录的 Chrome，带着你的账号、Cookie、已打开的标签页。适合的场景：需要你登录状态才能进行的任务、操作多个标签页、读取和对比不同页面的信息。典型例子：你已经在浏览器登了 Vercel，让 Codex 去 Vercel Dashboard 看部署状态；你登了飞书云文档，让 Codex 去读某个文档。网站会把 Codex 的操作当成你本人在操作，所以涉及发送、提交、付款的操作，务必你来确认。

Computer Use 操作浏览器——最重、最慢、最广。Codex 用 Computer Use 操作浏览器的时候，看到的是像素和坐标，像在屏幕上盲点，不理解网页结构。适合的场景：必须操作桌面应用、需要在多个应用之间切换、完成某个"最后一步"（比如上传文件、确认弹窗）。对比很清楚：任务全程在浏览器里 → 用 Chrome 扩展，不要用 Computer Use；需要操作桌面应用 → 只能用 Computer Use。

简单记忆：权限从小到大、隔离从强到弱：应用内浏览器 → **Chrome 扩展** → **Computer Use**。能用小权限完成的，别用大权限。本地调试用应用内浏览器，已登录的网页任务用 Chrome 扩展，桌面应用相关的事才动用 Computer Use。还有一个工具叫 Appshots，Appshots 算不上第四种方式，Appshots 是你"指"给 Codex 看的手段。在 Mac 上快速按两下 Command 键（左 CMD + 右 CMD），截取当前最前面的窗口，Codex 会收到截图和页面文字。然后你说"这个报错是什么意思"或者"这个按钮我想移到右边，怎么改"。Appshots 是你用来"指"给 Codex 看的手指，Computer Use / Chrome / 浏览器是 Codex 用来"动手"的工具。

不过上面三种方式都是"看屏幕操作"。有一类网站特别难搞——拼多多后台、电商管理后台、有反爬检测的 SaaS 系统，你用 Computer Use 去操作，会被网站识别为自动化操作直接拒绝访问。这时候就该换第四条路：Playwright。

Playwright 可以理解成"让代码直接操控浏览器内核"，它不靠看屏幕、模拟键鼠，而是直接通过 Chrome 的底层协议去找按钮、填表单、点页面，行为更接近真人。配合"复用你已经登录过的浏览器"这一招，能绕过大部分登录态检测。分工原则很清楚：开放的网站（Google 搜索、文档编辑、X 发帖、Vercel 这类），用 Computer Use 或 Chrome 扩展就够；有反爬保护的后台系统（电商后台、有验证码的页面、检测 WebDriver 的网站），用 Playwright 更稳。

有人用这套思路做过真实的电商批量上架：表格里几千行商品数据，要批量填进拼多多后台。Computer Use 直接被拒，换 Playwright 加复用已登录浏览器，整条链路跑通—读 Excel、分析字段、整理成上架表、自动填写、截图暂停让人确认、批量运行、记录成功失败日志。最后把流程沉淀成 Skill，从"一次性帮我填表"升级成"公司可复用的自动化流程"。这条偏进阶，新手先用三种基础方式，等碰到反爬墙了再研究 Playwright。同样要提醒：Playwright 也是有风险的操作，涉及支付、提交、删除的步骤，务必你确认了再让 Codex 执行。

62. Worktree 模式什么时候该用

Worktree 是给同一个项目额外开一个独立工作副本，相当于给 Codex 一本草稿本，让 Codex 大胆改而不影响你的主项目。

为什么需要 Worktree? 因为本地模式 (Local) 下, Codex 直接在你当前的项目目录里改, 每次改动都影响实际的工作文件。小修小补没问题, 但如果 Codex 要大胆尝试、要重构、要试一个不确定的方案, 直接在主项目上改风险太高—改坏了你的主项目就乱了。

Worktree 模式解决了这个问题。Worktree 在 Git worktree (和同一个仓库绑定的独立工作目录) 里隔离改动, Codex 在单独的分支里工作, 不影响你的主分支。如果跑错了, 直接删除 worktree, 零风险, 你的主项目纹丝不动。

经验法则是: **Worktree** 用于重要工作, **Local** 用于小修小补, **Cloud** 用于长时间运行的自动化任务。三个信任级别, 按任务选择。

什么时候该用 Worktree? 凡是牵连面广、结果不确定的活都建议开。比如让 Codex 做比较大的改动、试一个不确定的方案, 或者要重构、加新功能; 再比如你想同时做多任务、对比多个方案, 每个方案各开一个 worktree 互不干扰。说到底, 只要你喜欢 Codex"放心大胆地试"、最坏了能直接丢掉, 就用 Worktree。

什么时候用 Local 就行? 改个文案、调个颜色、修个小 bug, 这种改动小、可控、你盯着就能完成的, 用 Local 最快最直接, 没必要专门开 worktree。

Worktree 还有一个隐形好处：比较方案。你可以对同一个任务开两个 worktree，让 Codex 用两种不同的思路分别跑，跑完对比哪个好，留下好的，丢掉差的。这种"并行试错"在 Local 模式下做不到，因为两个改动会互相覆盖。普通人刚开始用 Codex，可能觉得 Worktree 听起来很高级、很复杂。其实 Worktree 就是个"草稿本"概念，用了 Git 的人会觉得很自然。就算你不用 Git，把 Worktree 理解成"给 Codex 一个独立的副本去改，改好了再并回主项目"就行。任何稍微大点的改动，都建议用 Worktree，这是最低成本的安全保障。

63. 多方案预览到底看哪个

这是 Codex 一个挺有用的功能：让 Codex 针对同一个任务，一次给你几个不同的实现方案，你挑一个。

新版 Codex 引入了一个预览系统，针对每个任务会生成 2 到 4 个不同的实现选项。每个方案可能用不同的技术选型、不同的结构、不同的取舍。你可以对比之后选一个最合适的，而不是 Codex 给你什么你就接受什么。

看哪个？几个判断维度可以参考。先看哪个更符合你项目的现有风格——如果你的项目用的是 A 技术栈，Codex 给的方案里有一个用 A、一个用 B，选 A 的那个，集成进去最顺。再看改动范围，同样的功能，一个改 5 个文件、一个改

15 个文件，优先选改动小的，引入风险低。另外依赖越少越好，一个不引入新依赖、一个要装新库，选前者，项目更干净。最关键的一条：选你看得懂的，看不懂的方案后续维护不了。

但多方案预览也有坑。一是别陷入选择困难。Codex 给你 4 个方案，你比来比去半小时选不出来，这比直接用第一个方案还浪费时间；多数情况下，第一个或第二个方案就够用，纠结边际效益不大。

还有一点，别都试一遍。有人想"我把 4 个方案都跑了看哪个好"，听着合理，实际很费额度。多方案预览的意义是让你在执行前对比思路，而不是让你每个都实际执行一遍。

正确的用法是：看几个方案的思路描述和关键差异，选一个你认为最合适的，执行 Codex。如果执行结果不满意，再换另一个方案。这种"先看思路选一个，不行再换"的方式，比"每个都跑一遍"高效得多。

64. Goal 模式和普通任务有什么区别

Goal 模式是给一个任务画一条明确的终点线，让 Codex 在一段时间内持续朝终点冲刺。

普通任务是这样的：你说一个需求，Codex 干完就结束，干没干成、干到什么程度，由 Codex 的判断决定。Goal 模式不一样，你除了给需求，还给一个明确的"成功标准"，Codex 会持续朝这个标准努力，直到达成或者明确达不到。

对比很清楚。一个糟糕的 Goal: "把这个 Markdown 文件里的计划实现一下。

"——没有终点，Codex 什么时候算完成全凭 Codex 自己判断，大概率做到一半就"觉得"完成了。

一个优秀的 Goal: "直到所有单元测试全部通过，这个新版本的开发才算完成。

"——终点明确，Codex 知道要跑到什么状态才算完，跑到那个状态之前都不算成功。

Goal 模式干的事是把"持续执行"和"验证器"结合在一起。你来定义想要的结果、何时停止的条件、以及判断 Codex 有没有离终点更近的信号。好用的验证器包括：一套完整的测试用例、一项基准性能测试、一个能稳定复现的 bug、一个验证矩阵、一个必须始终跑通的端到端工作流。

适合用 Goal 模式的场景：代码迁移（从一种语言迁到另一种语言，目标是所有测试通过）、bug 修复（目标是某个 bug 不再复现）、性能优化（目标是某个基准跑得更快）、重构（目标是功能不变但结构更好，所有测试还得过）。

不适合 Goal 模式的场景：需求模糊的事、没有客观验收标准的事、纯创意性的事。这些没有明确的"终点线"，Goal 模式用不上。

有一句话特别值得记住：有野心固然重要，但没有验证机制的野心，就只是在许愿而已。Goal 模式的前提是你能给一个可验证的终点。给得了，Goal 模式能帮你把复杂任务推进到位；给不了，用普通任务模式更合适。

65. 持久对话流和置顶怎么用

持久对话流（durable threads）是 Codex 里一个被低估的功能。

普通对话是“聊完即焚”的：你开一个对话，聊完任务完成，关掉，下次开新的又得重新解释背景。持久对话流不一样，持久对话流能长时间运行、跨多次使用保留工作上下文。Codex 记得你之前做过的决定、你的偏好、当前的进度，下次你回到这个对话，Codex 还在那个状态。

为什么要用持久对话流？因为很多工作是持续推进的，不是一次性完事。你做一个产品，可能要持续几周；你做内容创作，可能每天都有新想法往里加；你监控某些数据，可能要持续跟进。这些场景下，每次开新对话重新解释背景太累，持久对话流让你“接着上次的继续”。

推荐的用法是“置顶”。把那些需要反复推进的持久对话置顶，这些对话就随叫随到。比如一个专属的“幕僚长”对话流，帮 Codex 处理日常杂务；一个负责产品发布的对话流；一个负责审查文档的对话流；一个专门盯外部数据的监控对话流。这些对话跟聊完即焚的闲聊框不一样，这些对话是持久的工作空间。

置顶之后还能用快捷键。按下 `Command-1` 到 `Command-9`，你能瞬间穿越回这些保存好的专属对话流里继续工作。这种"一键回到某个工作上下文"的体验，比每次翻历史记录找对话快得多。

持久对话流配合共享记忆，威力更大。把这些持久的对话"锚定"在一个知识库里——说白了就是建一个存放纯文本文件的文件夹。在最外层放一个 `AGENTS.md`，给 `Codex` 定规矩：当 `Codex` 了解到新情况时，如何更新这个知识库。这样重要的上下文就不会锁死在某一次聊天里，而是写下来，放在下一个对话流能立刻接手的地方。

普通人的起步用法：为你最常做的几类事，各开一个持久对话流，置顶起来。比如一个"日常杂事"、一个"我的项目"、一个"学习笔记"。下次有事，直接去对应的对话流里说，不用每次重新解释你是谁、在做什么。

66. 共享记忆怎么搭最稳

共享记忆是让 `Codex` 跨多次对话保留上下文的机制，有一种最稳的搭法。

为什么要搭共享记忆？因为 `Codex` 默认只有短期记忆——`Codex` 在本次对话里学到的东西，明天就忘。每次新对话，`Codex` 都"重新认识"你和你的项目。如果你反复做类似的事，每次都要重新解释，效率很低。共享记忆就是把这些重复需要的上下文，存在对话之外的地方，让未来的工作能基于这些信息继续推进。

最稳的搭法，是用一个纯文本文件夹当知识库。说白了就是建一个文件夹，里面放 markdown 文件。示例结构：vault 文件夹下分 TODO、people、projects、agent、notes 几个子目录。在最外层放一个 AGENTS.md，给 Codex 定规矩：当 Codex 了解到关于人员、项目、决策、待办的新情况时，如何更新这个知识库。

为什么用纯文本文件夹？纯文本文件夹简单直白，方便你随时查看、修改、移动，能保存很久。团队可以把这个文件夹放在任何云盘里—Git、Dropbox、Google Drive 都行。

还有一点特别重要：别死板地照抄某一种知识库结构。你需要做的是"教"Codex：那些持久的上下文该放在哪、哪些上下文需要保留、什么时候不要瞎折腾去改文件。每个人的工作方式不一样，知识库结构应该贴合你的实际，而不是抄一个"标准模板"。

一份实用的 AGENTS.md 指南里可以写：把这个文件夹当作长期工作记忆区；把笔记整理得有条理，别搞得到处都是碎片记录；准确地把待办、人员、项目、每日总结分类放好；把做过的决定、遇到的卡点、负责人、日期、有用的链接保存下来；如果没有实质性的新进展，不要随意修改文件。

Codex 自己也提供官方的记忆功能，在设置 > 个性化 > 记忆里。这个功能就像系统自带的本地记事本，用来记住你的个人偏好、常用的工作流、经常踩的坑。但这个功能是用来辅助你清晰写下来的上下文的，不是取代它。

等你用得更深，想让记忆系统更聪明，可以往"带索引和语义检索"的方向升级。社区里有进阶玩家用 Obsidian 当记忆库的外壳，加一层本地数据库做关键词检索，再加一层语义检索（按意思找东西，而不是只按原词匹配），配合 Git 做版本留痕、closeout 脚本做自动收尾、audit 脚本做定期体检。这种架构能把记忆库从"一堆文本文件"升级成"可搜索、可审计、能自运转的系统"。不过这套偏极客，搭建成本高，对普通用户来说，先把基础的纯文本知识库用好，等你规矩多得管不过来了，再考虑往这个方向升级。

普通人起步：建一个文件夹，放一个 AGENTS.md，写清楚你最常需要 Codex 记住的几件事。然后用起来，缺什么补什么，慢慢长出自己的知识库。不用一开始就追求完美结构。

67. 语音输入为什么值得用

语音输入听起来像个小功能，但语音输入解决了一个真问题：把脑子里最原始、最粗糙的念头先捕捉下来。

打字的问题是什么？你脑子里有个模糊的想法，但还没完全成型。要把它打成文字，你会本能地“字斟句酌”一边想边组织语言、边过滤边精简。这个组织的过程会丢失很多信息，原本丰富的上下文被你压缩成几句干巴巴的话，AI 拿到的信息量变少了。

语音输入绕过了这个过滤。你对着 Codex“碎碎念”两三分钟，把脑子里的想法一股脑倒出来，效果出奇的好。举个例子：“我记得有个叫 Ben 的人在 Slack 上提过这事儿，细节我忘了，你去帮我找找看。”对于一个会自己搜索、收集上下文并向你汇报的 agent 来说，这种模糊的话就足够 Codex 干活了。

录音转写也是同样的道理。一份未经修饰的会议记录，或者一段口述的计划草案，往往比一份简短的总结更有价值。因为那些粗糙的记录里，保留了你犹豫的语气、强调的重点、还没讲完的灵光一现。这些细节在打字的时候会被你“精简掉”，但恰恰是 AI 理解你真实意图的关键信号。

语音输入配合任务干预，威力更大。一个任务正在跑，你看着侧边栏发现某个元素不对，不用停下来打字，直接开口说“这个调小一点”“这句文案写错了”。这种“边看边说”的交互，比“打字描述问题”流畅得多。

什么时候用语音？想法还很模糊、没完全成型的时候最该用，把它一股脑倒出来再整理；需求复杂、打字要打很长的時候也划算；任务跑的过程中你想随时插话，开口比打字快得多；还有就是你正走动、不方便打字的时候。

Codex 的语音输入可以设成全局键，打开之后不止在 Codex 内部，可以在电脑上任意一个对话框调用。如果你习惯了别的语音输入法（比如豆包），不切换也行，重点是养成“用嘴说”的习惯，而不是事必打字。

68. AGENTS.md 有哪些高价值的进阶套路

Q57 讲了 AGENTS.md 的基础写法。写多了你会发现，有几类规则特别管用，是社区里反复验证过的“高效套路”。这四个套路各有针对性的痛点，可以按需挑着用，不用全上。

第一个套路叫 Confidence Gate（置信度闸门），专治“半成品”。Codex 经常在任务没真正完成时就宣称做完了。你在 AGENTS.md 里写一段“完成定义”，让 Codex 在宣布完成前先自评打分，分数不够就得继续干。模板长这样：

宣布任何任务完成之前，先在三个维度上给自己打分（每项 0-10）：1. 测试证据——有没有跑测试，把输出贴出来；2. 行为等价——有没有验证公开接口和行为没变；3. 清理——有没有删掉死代码、没留下注释掉的代码块。三项都达到 8 分以上才算完成。任何一项低于 8 分就继续干，不要来问我。

这个套路对治半成品特别有效。Codex 为了凑够分数，会主动跑测试、主动验证、主动清理，而不是写完核心功能就停。

第二个套路专治"频繁打断"。长任务里 Codex 会时不时停下来问"要不要继续""下一步做什么",把节奏打断。在 AGENTS.md 里明确写:

持续执行直到任务完成,不要中途停下来问我"要不要继续"。遇到必须确认的事情,先按最合理的方式做,把决策记录下来,等我回来统一审。

这条能让长任务从"一小时停三次"变成"连续跑数小时",省心得多。

第三个套路专治"上下文爆炸"。Codex 跑长任务时,会把命令输出的整页日志塞回上下文,把额度吃光。在 AGENTS.md 里约束命令输出的处理方式:

命令输出超过 50 行的,只保留尾部和关键报错行,不要把整页日志塞回上下文。读大文件这种重活,开一个新的子对话去读,主对话只收结论。

有人实测这条规则把额度消耗砍掉了一半左右。

第四个套路有点反直觉:别让 AGENTS.md 太长。社区里有人做了跨多个 agent 的对照研究,发现过长的 AGENTS.md 反而降低表现——Codex 要花更多注意力去遵守所有规则,反而抓不住重点。正确做法是把 AGENTS.md 当"入口",只放最核心的规则和指引;架构细节、实现约定、repo 结构这些,拆成单独的 markdown 文件,从 AGENTS.md 里引用。AGENTS.md 控制在你能一眼扫完的长度,Codex 反而更听话。

第五个套路是进阶玩法：把习惯拆成多个文件，按场景路由。一个 AGENTS.md 装不下所有规矩的时候，别硬塞，按场景拆成几个文件，让 Codex 工作前先读对应的那份。

举例，你可以这样拆。一份放全局使用习惯：用什么系统（比如 Windows 默认跑 PowerShell）、文件操作用 `-LiteralPath`、不混用 Bash 的 `&&`、执行命令前先确认项目目录。这些看着碎，但决定了 Codex 给的命令能不能直接在你电脑上跑——很多时候 Codex 不是不会做，是不知道你本机的工作环境。另一份放写作和沟通偏好：中文要直接、不要套话、不要硬升华、未实测的工具必须标注“未实测”，让 Codex 写东西时不会写出那种很标准的 AI 腔。再一份放安全边界：不删用户文件、不自动 commit 或 push、发布前先检查 Token 和密钥、不把私有代码发给公网搜索工具。

这种“多文件路由”的好处是：每个文件短、专注、好维护；Codex 按需读对应那份，不会被无关规则干扰。坏处是搭建成本比单个 AGENTS.md 高，适合你已经用了一阵、规矩越来越多的时候再上。新手先把 Q57 的基础 AGENTS.md 写好，等规则塞不下再来这套。

69. 怎么让 Codex 改代码前先讲思路

这是个特别实用的防翻车技巧，比计划模式更轻量，适合"不想走完整 plan 流程、但又不放心让 Codex 直接干"的场景。

正常情况下，你给个任务 Codex 直接开干。Codex 干得很快，等你想介入的时候已经改完了一大半。如果你能在 Codex 动手之前先听听 Codex 的思路，很多翻车都能提前挡住。

怎么让 Codex 先讲思路？最简单的就是直接说："先告诉我你打算怎么做，我确认了你再开始改。"Codex 会返回一份思路描述——打算改哪些文件、用什么方法、分几步。你看完觉得没问题，再说"开始执行"；觉得有问题，直接在思路阶段提出来改。

复杂一点的任务可以用计划模式，打 `/plan`，Codex 会返回一份结构化的计划文档，比单纯讲思路更完整。

更省事的做法是把它写进 `AGENTS.md`，作为 Codex 的默认行为。比如"对于任何涉及多文件的改动，先告诉我思路，我确认了再执行"。这样不用每次都嘱咐，Codex 默认就先讲思路。

跳过计划阶段是项目出错的最大原因。一句话的误解，可能让 Codex 改了四十个文件。先让 Codex 讲思路，你能一句话纠正，比写完代码再改省事十倍。

什么时候该用这个技巧？凡是任务涉及多文件改动、做法上有多种选择而你又想参与决策、或者错了代价比较大的，都必须先讲思路。

什么时候不用？改个变量名、调个样式、问个简单问题，这种确定性强、错了也好回退的事，直接让 Codex 干就行，先讲思路反而啰嗦。

记住一个原则：复杂度越高、不可逆性越强，越要让 Codex 先讲思路。简单可逆的事可以放手让 Codex 干，复杂或者影响面大的事必须先对齐思路。这个习惯能挡住大部分翻车。

70. 怎么让 Codex 记住我项目的约定

让 Codex 记住约定，最稳的办法就是写下来，而不是指望 Codex 自己“记忆”。

Codex 默认只有短期记忆，Codex 在本次对话里学到的东西，新对话就忘。很多人抱怨“我跟 Codex 说过好几次别动那个文件，Codex 还是动”，根本原因就是 Codex 真的“忘”了——你上次说的，存在上次的对话里，新对话 Codex 不知道。

让约定真正被记住，要靠几种“写下来”的机制。

最常用的是项目级 AGENTS.md，把项目的硬规矩写在里面，每次开新对话

Codex 都会读。比如“不要修改 config 目录下的任何文件”“所有新增组件放在 components 目录”“使用我们项目自己的代码风格”。这些写一次，长期生效。

再往上一层是用户级 **AGENTS.md**（在 `~/.codex/AGENTS.md`），把你个人的通用偏好写在这里，对你所有项目都生效。比如“我偏好简洁的代码”“修改前先解释影响”“涉及隐私文件先提醒我”。

如果你有一套固定的工作方式，比如“写文档按这个结构来”“做代码审查按这个清单查”，那就把这套方式做成 **Skill**。**Skill** 是显式的、可调用的，比纯记忆可靠。

Codex 还自带一套记忆功能（设置 > 个性化 > 记忆），算系统级的本地记事本，用来存你的个人偏好、常用 workflow、踩过的坑。但这个功能是辅助性的，不能取代你清晰写下来的上下文。

还有一个特别好的建议：遇到值得记录的经验教训时，告诉 **Codex** 更新

AGENTS.md 或者项目记忆。比如 **Codex** 试了一种方法失败了，你跟 **Codex** 说“把这次失败的原因记到 **AGENTS.md** 里，下次别再试这个方法”。系统会越来越聪明，因为 **Codex** 把这些教训写下来了。

一句话总结：别指望 **Codex**“记住”，要让 **Codex**“读到”。写在 **AGENTS.md**、**Skill**、知识库里的东西，**Codex** 每次都能读到；只存在对话里的东西，下次就没了。重要约定一定要写下来。

71. 哪些小习惯能让结果稳定性翻倍

很多小习惯，单独看都不起眼，叠加起来效果惊人。这里集中讲几个最有效的。

最值得养成的一条：所有复杂任务先开计划模式。计划阶段消耗少，能挡住绝大多数跑偏。养成"但凡复杂点的事先 `/plan`"的习惯，翻车率会大幅下降。

每次跑完别只看 Codex 的文字总结，去 diff 里看 Codex 到底动了什么。看到不对的改动立刻问、立刻回退。这个习惯能把"顺手重构""过度工程化"挡在合并之前。

任务要分清楚边界——一个明确任务一个 thread，不要把多件事塞进一个对话。

任务越独立、上下文越干净，Codex 越稳定。

遇到失败先停再想。发一句"先停一下，总结失败原因"，比无脑重试有用得多。

让 Codex 给你失败分析，你再决定下一步。

还有几条顺手就能做的：好的结果立刻保存，agent 的输出是一次性的，跑出来不保存就没了，满意的成果马上复制到项目文件夹、提交到 git；任务描述带验收标准，明确说"完成的标准是 XXX，缺一样都不算完"，半成品的概率会大幅降低；任务描述也别忘了带约束，"不要动 X""不要引入新依赖""不要顺手重构"，约束越明确，Codex 越收敛。

最后是两个跟额度和成本有关的小规矩。经常查额度，发现掉得快立刻排查，不用的时候彻底退出 Codex，不让后台任务偷跑；按任务配推理强度，小任务用低档省额度，复杂任务才用高档，开任务前花两秒判断该用哪档。

还有一条特别要紧：遇到值得记录的经验立刻写进 `AGENTS.md`，让 Codex 下次别再犯同样的错。系统会越用越聪明。

这些习惯没有任何一条是高深的技巧，全是日常操作层面的小规矩。那些把 Codex 用得特别好的人，靠的并不是什么秘密，靠的就是这些小习惯每天都在做。叠加起来，他们的结果稳定性就是比一般人高得多。

五、场景实战

72. 第一次该拿什么任务练手

别拿重要项目练手，别拿复杂需求练手，拿一个“成了挺好、废了也不亏”的小事开始。

具体拿什么任务练手？推荐让 Codex 做一个简单的 Hello 网页：黑色背景、页面中间显示大字、字体白色、整体水平和垂直居中、只用 HTML 和 CSS。这种任务特别适合练手——它清晰、简单、可验证，Codex 做完了你一眼就能看出对不对；涉及的文件少，不会出乱七八糟的依赖问题；做错了也没成本，改改再来就行。

要明确一点：第一次练手的核心目标是走通“建项目→提任务→看 Codex 执行→看 diff→验收”这套流程，而不是真的做出什么有用的东西。你要感受的是 Codex 怎么工作、你怎么跟它交互、结果在哪里看。这些流程性的体验，用一个最简单的任务就能跑通，没必要用复杂任务给自己增加认知负担。

第一次还有几个小规矩得守住。先用普通对话问个简单问题，确认 Codex 能正常回复、整套环境配通了；然后建一个干净的项目文件夹，别用你已有的重要项目；打开计划模式，让 Codex 先出方案你过一眼；跑完之后认真看 diff、看

实际产物，验收习惯从第一次就要养成。整个过程都别开完全访问权限，保持"请求批准"档位。

有一个观点值得记住：很多 **Codex** 任务失败，原因不在 **Codex** 不会写代码，而在它还没理解项目就被要求直接动手。所以第一次练手，重点是让自己熟悉"怎么和 **Codex** 合作"，而不是检验它的能力。等你把这套流程跑顺了，再上真正的活。

73. 让 **Codex** 做网页该怎么提需求

做网页是 **Codex** 最经典的场景，也是最容易看出"提需求方式决定结果质量"的场景。

新手最常犯的错是需求太模糊。一句"帮我做个网站"，**Codex** 会给你一个完全凭 **Codex** 自己想象的东西，技术栈、风格、结构全是 **Codex** 定的，和你想的差很远。你看着这个东西不满意，又不知道怎么改，就卡住了。

正确的提法是把需求具体化到 **Codex** 能照着做。可以套用一个结构。背景：一段话，你是谁、为什么要做这个网页。目标：一段话，描述最终状态，比如"一个展示我个人作品的 **portfolio** 页面，单页应用，响应式"。约束条件：列表，硬性规则，比如"只用 **HTML** 和 **CSS**，不用框架""配色用深色系""字体用思源

黑体”。完成标准：怎么算完成，比如“在浏览器里打开能看到内容，移动端也能正常显示”。

还有几个能直接提升结果质量的技巧。先说用途和受众——“给我自己用的 portfolio”“给客户看的展示页”“做演示用的原型”，受众不同，Codex 给的结构和风格也差很远，提前讲清楚省得返工。更关键的是，先让 Codex 用图像功能生成概念图。在写 UI 代码之前，先让 Codex 用 gpt-image 生成一两张图模拟页面外观，保存到项目里，然后看着图来写代码——最终的视觉效果，比让 Codex 凭文字描述去设计好得多。还有一点值得一提：做完之后用应用内浏览器打开看效果。让 Codex 在侧边栏的内置浏览器里打开，你能直接看到效果，圈出不对的地方让 Codex 改，比反复打字描述问题高效多了。

做完之后，标准验收流程：看 diff 确认改动、在内置浏览器看实际效果、在移动端尺寸下检查布局、点开每个链接看通不通。这些都过了，这个网页才算真做好。

74. 让 Codex 做 PPT 该怎么玩

做 PPT 是非程序员最能直接受益的场景之一。

开始之前先装 PPT 插件。Codex 默认不一定有 PPT 能力，需要装 Presentations 插件。装完之后，Codex 就能生成可编辑的 PPT 文件——是真的能打开改的演示文稿，不是图片或 PDF。

装好插件，接下来最要紧的是给 Codex 足够的素材。做 PPT 这件事，素材的质量直接决定结果的质量。做年中汇报 PPT，把相关的文件、数据、截图、PDF 都丢给 Codex，它能理解所有这些内容，整合成一份 PPT。你给的素材越丰富、越具体，做出来的东西就越贴合你的需求。

光有素材还不够，需求也得说清楚，这点跟做网页一样。模糊的需求只会得到模糊的结果。讲明白主题、受众、风格、要包含哪些部分、大致几页，尤其要明确风格——商务正式、活泼简洁、科技感强，这些风格词会直接影响 Codex 的设计。

另外得心里有数：AI 一次做出来的 PPT，肯定不会 100% 贴合你的要求。

Codex 做完之后，你直接上手改就行；如果改动比较大，继续给 Codex 提要求让 Codex 改，比你自己从头改快。

要是你做 PPT 比较频繁，还可以用 Skill 沉淀风格。网上有别人做好的 PPT Skill，把 Skill 地址发给 Codex，让 Codex 帮你安装，之后做 PPT 就按那个 Skill 的风格来。如果你自己有固定的 PPT 风格，也可以做成自己的 Skill，每次都按你的风格做，不用重新解释。

装插件只是最基础的路。社区里现在还流行两条更进阶的路线，效果往往比插件生成的 PPT 更好看。一条叫图片型 PPT：先让 Codex 用图像能力生成一张张 16:9 的视觉图片，再把图片铺进 PPT，每一页就是一张完整图片。这条路视觉效果好，但每一页是图片，编辑性差。另一条叫 HTML 型 PPT：不用传统 PPT 软件，直接用 HTML、CSS、JS 写幻灯片，能在浏览器预览、能做动画、能做交互、能用代码控布局，特别适合和 Codex 这种代码执行型工具配合。网上有不少开源的 HTML PPT Skill，搜一下让 Codex 装上就能用。

除了"写 Skill"，还有一种更新、更省事的 Skill 创建方式，叫 Record & Replay（录制回放）。你在 Mac 上把一套操作亲自演示一遍，Codex 在旁边看一遍，自动把流程整理成一个可复用的 Skill。比如你录一个"文章多平台分发"的流程——打开文章、改标题、整理标签、检查封面、切换平台、再处理一遍——Codex 观察后生成 Skill，下次你只要说"用我的文章分发 Skill，原文是这篇，平台是小红书和公众号"，Codex 就按你录的规则处理。

这条路特别适合那些"步骤固定但你写不清楚、做一遍反而清楚"的重复流程：

内容分发、电商选品、报表整理、后台录入。有几个限制要先知道：目前主要是 macOS 支持，Windows 还得等；需要开启 Computer Use 权限；高风险操作（支付、转账、删库）不建议交给它录；页面经常变的网站录出来不稳定。最适合的是步骤稳、页面稳、成功标准明确、风险可控的流程。

做 PPT 最容易踩的坑是期待 Codex 一次完美。Codex 的价值在于给你一个 80% 完成的底稿，你再加工，指望它一次给你能直接用的成品多半会失望；指望它给你一个改起来比自己从头做快得多的底稿，这个期待就靠谱多了。

75. 让 Codex 做视频靠谱吗

能做，而且效果可以很惊艳，但要装对插件、给对素材。

Codex 自己不会直接剪视频，Codex 的视频能力靠插件。两条路：一条是 HyperFrames by HeyGen 插件，专门做动效视频；另一条是 Remotion Studio，用代码生成视频。这两条路的逻辑都是“用代码描述视频”，然后由对应的引擎渲染成真正的视频文件。

有一个特别生动的电商案例。一个做电商的人用 Codex 做了一个“电商剪辑工作流”：拆解爆款、识别和选择素材、自动配音加 BGM、自动对齐字幕。整个过程做成一个 Skill 之后，每次只要输入一个斜杠命令，Codex 就按提前设计好的流程跑，跑完在剪映草稿里能看到剪好的视频。就靠这一个工作流，他说电商素材产出效率翻了 5 倍，单日订单涨了 60%。

这个案例的启示是：视频这件事，“一句话丢给 Codex 就完事”做不成，视频是一套流程。拆解爆款、选素材、配音、字幕，每一步都是独立的任务，组合起来

才是完整的视频工作流。把这个工作流固化成 Skill，下次一键调用，比每次重新解释流程高效得多。

让视频工作稳定输出，有几个要点得守住。素材一定要给齐，视频质量很大程度上取决于你给的素材质量。流程也要拆细，别一句"帮我做个视频"就完事，要拆成"先拆解爆款结构""再选这些素材""按这个节奏配音""字幕按这个格式对齐"。最要紧的是结果一定要验证——Codex 做的视频你一定要看一遍，发现不对的地方让 Codex 改，别默认接受。等这套流程跑通一次，就把它做成 Skill 固化下来，下次直接调用。

视频是 Codex 能力上限很高的一个场景，但门槛也比做网页、做 PPT 高。新手不要一上来就碰视频，先把基础流程跑顺，再往视频这种复杂场景延伸。

76. 让 Codex 处理 PDF 和文档怎么用

这是非程序员用 Codex 最高频的场景之一。

Codex 能干哪些事？分析体检报告、把 PDF 转成 markdown、提取 PDF 里的关键信息、整理一堆文档的内容、对比几份文档的异同。有人直接把家人的体检报告 PDF 扔给 Codex，让 Codex 分析之后梳理一份建议放到飞书文档上，全程不用自己读那些医学数据。还有人经常要把 PDF 转成 MD 文件，这事也交给 Codex 批量处理。

为什么这件事 Codex 做得特别好？因为文档处理是典型的“输入明确、输出可验”的任务。输入是一份或者几份文档，输出是结构化的整理结果，对不对你一眼能看出来。这种任务最能发挥 Codex 的执行能力，又不涉及高风险操作。

提需求的关键是把“输出格式”说清楚。不要只说“帮我分析这份报告”，要说“帮我分析这份报告，按以下结构输出：1. 总体健康状况，2. 异常指标列表，3. 每个异常指标的解释和建议，4. 后续要关注的事项”。输出格式越明确，Codex 给你的东西越有用。

处理多份文档的时候，把所有文档放在同一个项目文件夹里，让 Codex 一次性读完所有，然后做整合。比如对比三份合同的条款差异、把五份会议纪要整理成一份汇总、从十份简历里筛选出符合某个条件的候选人。这些任务手工做要花几小时，交给 Codex 几分钟就完。

文档处理有一个特别适合的场景：批量格式转换和清理。把一堆 PDF 转成 markdown、把一堆 word 文档提取出纯文本、把一堆乱糟糟的文件按某种规则重命名归类。这种重复性高、规则明确的活，Codex 做得又快又准，你手工做反而容易出错。

唯一要注意的是隐私。如果文档里有敏感信息（身份证号、银行卡号、公司机密），让 Codex 处理之前先想清楚。Codex 处理的内容会经过 Codex 的模型，虽然 OpenAI 有隐私政策，但涉及真正机密的内容还是要谨慎。

77. 让 Codex 分析数据和对账怎么做

数据分析和对账是另一个普通人高频受益的场景。

有一个特别典型的对账场景。一个做业务的人，每天上午要去检查昨天的回款数据，看几张表的数据能不能对得上。以前这事人工干，容易忘、容易漏。这个人把这个流程做成了一个 Skill：告诉 Codex 对哪里的账、怎么对、什么算异常。然后设了个定时任务，每天自动跑一次，发现异常就提醒。这个流程从他脑子里挪到了系统里，再也不会忘，再也不会漏。

这个案例揭示了数据对账类任务的关键：你要做的，是把脑子里的对账规则写成 Codex 能照着执行的检查步骤，而不是指望它替你判断什么算异常。你不说，Codex 不知道你要对什么、怎么算异常，自然帮不上忙。

提这类需求，有几个要素得讲清楚。先告诉 Codex 数据在哪里——飞书多维表格、Excel 文件、数据库、还是某个 SaaS 工具，让 Codex 知道去哪读。然后说明规则：哪些字段要对、按什么 key 对、差异在什么范围内算正常、超过多少算异常。再讲清楚发现异常怎么办——是发邮件给你、推送到飞书、写进报告、还是只标记出来等你来看。最后别忘了约定正常情况输出什么，日报、周报、汇总表，什么格式。

涉及外部数据源的时候，可能需要 MCP。比如数据在飞书多维表格里，给

Codex 装飞书的 CLI 或者飞书 MCP，Codex 就能直接读飞书的数据。数据在

Notion 里，装 Notion MCP。数据在本地 Excel，那直接放项目文件夹里就行，不用 MCP。

对账这类任务特别适合做成 Skill 加自动化。Skill 把对账规则固化下来，自动化让 Codex 定时跑。组合起来，就是一个"AI 值班会计"，每天自己查、自己比对、发现异常主动汇报。用了这套之后，那些最耗时耗力的"收集背景资料"的工作往往在你回来之前就做完了，你只需要做最后拍板的决定。

78. 让 Codex 帮我清理电脑靠谱吗

靠谱，而且效果出乎意料地好，但要谨慎。

有个人用 Codex 检查最近装的应用是不是都卸载干净了。卸载软件这件事，很多人都遇到过：明明卸载了，系统里还残留一堆配置文件、缓存、依赖，慢慢把磁盘塞满。让 Codex 检查某个软件卸载干不干净，Codex 会去找这个软件相关的文件和依赖，看看有没有残留。如果有，直接让 Codex 清理。这个人说他在 macOS 上靠这个方法腾出了 20G 空间。

为什么 Codex 适合做这个？因为"找残留文件"这种事，规则明确（找特定名称、特定路径、特定后缀的文件）、可验证（找到的就是找到、没找到就是没找到）、又不涉及危险操作（删除前给你看清单，你确认才删）。这种任务 Codex 能稳定完成，你验收也容易。

但这里有安全边界必须讲清楚。首先，始终用项目可读写档，别开完全访问，把 Codex 限制在它处理的范围里，别让 Codex 整个硬盘乱扫。其次，删除前一定要看清单——让 Codex 列出来要删什么，你看完确认才让 Codex 动手。还有，重要文件先备份，万一 Codex 判断错了要删的东西，你有备份能恢复。最要紧的是，碰到不确定的文件，别盲信 Codex 说它"没用"，多问一句"这是什么文件，删了会有什么影响"，等它给你解释完你再决定。

清理电脑这个场景，涉及删除数据的操作，最好别让 Codex 自动执行，务必你来确认。Codex 特别擅长"找出候选删除项"，但"删不删"这个决策要你来做。这种"Codex 找、你定、Codex 执行"的协作模式，是处理有风险任务的标准做法。Codex 把最耗时的"找"做了，你做最关键的"判断"，Codex 再执行你确认过的操作。既发挥了 Codex 的效率，又保住了你的控制权。

79. 让 Codex 写文章和做内容怎么用

内容创作是 Codex 的强项之一。

有一个内容创作者的例子特别典型。这个人目前的内容输出基本靠 Codex 搞定。前几天发的一篇公众号文章，新号跑出 1000 浏览量，算是个小爆款。这篇文章从大纲到初稿，再到封面、配图，全部由 Codex 完成。这个人只是在完成之后做了一些细节上的调整。

这个人还有一套自动化的内容生产流程。每天 Codex 会给这个人准备一份 AI 行业日报，全网有哪些值得关注的资讯第一时间看到。然后还有一个选题

Brief 的自动化任务：Codex 每天读取日报内容，找出其中可以写的选题，改成需求 Brief 放在这个人的知识库里。那篇爆款文章，选题就是 Codex 给这个人做的。

这种用法揭示了内容创作的关键：真正省力气的做法，是让 Codex 把"找选题、做调研、出初稿、配图"这些耗时的体力活扛下来，你只做"选哪个选题、改哪里、发不发"的判断。那位作者的做法就是这样，只在完成以后做一些细节调整，效果还不错。

让内容创作稳定输出，有几个要点。最要紧的是给 Codex 一份风格指引——你的写作风格、目标读者、内容调性，写进 AGENTS.md 或者做成 Skill，Codex 每次都按你的风格来；网上也有别人做好的写作 Skill，可以直接用。光有风格还不够，输入也得给足，Codex 写得好不好很大程度取决于你给的素材和背景，先把选题、调研、素材都准备好（这些也可以让 Codex 做），再让 Codex 写正文。还有一点心态得摆正：AI 一次出来的东西不会 100% 贴合你的要求，你要期待的是它给你 80% 的底稿，你再精修，而不是指望一次完美。

内容创作还有一个适合做成自动化的点：定期监控和选题。每天让 Codex 扫一遍你关注的信息源，找出值得写的内容，整理成选题清单放在你的知识库。等你

坐下要写的时候，选题已经有了，调研也做了一半，你直接进入"写和改"的环节，效率高得多。

80. 做完整的 side project 怎么用最划算

side project（个人小项目）是 Codex 的甜区。

为什么 side project 特别适合？因为 side project 通常是一个人做、规模可控、需求自己定、错了也没大代价。这种场景下，Codex 的能力上限完全够用，Codex 的额度消耗也在可控范围，你能体会到"AI 替我把大部分活干了"的爽感。

最划算的用法是按"六步法"推进。完整流程是这样的：需求拆解→制定计划→小步实现→测试→代码审查→提交与复盘。每一步都有对应的提示词模板，逐步推进，比一句话丢个大需求靠谱得多。

第一步需求拆解。把你想做的东西写清楚，七个要素：背景、要解决什么、相关文件、不能动什么、完成标准、需要哪些测试、有哪些风险。第二步制定计划。用计划模式让 Codex 出方案，你审完确认。第三步小步实现。不要让 Codex 一口气把整个项目做完，分模块、分功能，一步一步来，每步完成立刻验证。第四步测试。让 Codex 写测试用例，跑通了才算这一步完成。第五步代码审查。推荐两轮审查：第一轮让 Codex 自审，把明显问题先暴露出来；第二轮你人工

审，重点看边界条件、安全问题、是否误删、业务逻辑。第六步提交和复盘。改完提交到 git，记录这次遇到的问题、总结好用的 Prompt、更新 AGENTS.md。

side project 还有一个特别适合的工作模式：用 ChatGPT + Codex 组合。

先用 ChatGPT 想清楚，再用 Codex 进项目执行。ChatGPT 是顾问，帮你理清思路、做技术选型、讨论方案；Codex 是实习生，进项目把活落地。两者配合，比单独用任何一个都强。

省额度，有几个习惯值得养成。复杂任务先开计划模式，规划阶段消耗少，规划对了执行阶段不浪费；小任务用低推理强度就行，复杂任务才上高档；不要一次让 Codex 扫整个项目，能指定文件就指定文件；另外别反复大范围重构，那种操作最烧额度。还有一条经验：拿不准的时候先让 Codex 讲思路，你确认了再让 Codex 执行，比一头扎进去撞墙划算。

side project 是用 Codex 最好的"练习场"。你用 side project 把整套流程跑顺，熟悉了 Codex 的工作方式，再上真正的生产项目，心里就有底了。想让 Codex 半自动跑长项目，Q87 会专门讲那套"半自动工作流"怎么做。

81. 改 bug 该怎么交给 Codex 最有效

改 bug 是 Codex 最能体现价值的场景之一，但提需求的方式决定成败。

最有效的提法是这样的：把完整的报错信息给 Codex，加上能复现这个 bug 的步骤，加上相关的代码文件。标准做法是把报错信息直接发给 Codex，让 Codex 自己分析。Codex 能读到完整的报错栈、能跑到你的项目里复现、能改完再跑一遍验证。这种"Codex 在自己环境里真的跑通"的能力，是改 bug 场景里 Codex 比普通对话强得多的地方。

新手最常犯的错是只描述 bug 不给信息。一句"我的登录页报错了"，Codex 只能凭空猜。Codex 不知道报什么错、不知道你的代码长什么样、不知道怎么复现，给的建议也是泛泛的，解决不了你的实际问题。

正确的提法是把这几样东西凑齐。先把完整报错给它——控制台输出、错误信息、报错栈直接复制，或者截图（Codex 支持图片输入）。再讲清楚复现步骤：你是怎么触发这个 bug 的，点哪里、输什么、做了什么操作。还要指一下相关文件，告诉 Codex bug 大概在哪些文件、哪些函数里，缩小它的搜索范围。最后别漏了环境信息，什么操作系统、什么浏览器、什么版本，这些有时候很关键。

改 bug 还有一个特别适合的模式：让 Codex 先复现，再修。让 Codex 先在你的项目里跑一遍，触发那个 bug，确认 Codex 能复现。然后再让 Codex 修。修完再跑一遍，确认 bug 没了。这种"先复现、再修、再验证"的循环，比"Codex 给你一段代码你拿去试"靠谱得多——因为 Codex 在 Codex 自己的环境里已经验证过了。

遇到难修的 bug，推荐用 Goal 模式。给一个明确的终点："直到这个 bug 不再复现，修复才算完成。"Codex 会持续朝这个目标努力，试各种方法，每试一种都验证一遍，直到 bug 真的消失。

改 bug 最容易踩的坑是"修了一个引入两个"。Codex 改完 bug，你可能没注意 Codex 顺手改了别的东西，结果 bug 修了但引入了新问题。防御办法还是那条：每次跑完看 diff，发现 Codex 改了和 bug 无关的文件，立刻问、立刻回退。

82. 加新功能该怎么拆给 Codex

加新功能是 side project 里最常见的需求，标准做法是"小步实现"。

新手最容易犯的错是一句话丢个大功能。"帮我的网站加个用户系统"，Codex 听到这种需求，会按 Codex 自己的理解从零给你搭一套，登录、注册、权限、数据库、session 管理，全都做。结果大概率和你现有的项目结构对不上，引入一堆新依赖，还可能动到你不想动的地方。

正确的拆法是按"用户能感知的功能单元"拆，不按"技术模块"拆。比如"加用户系统"这个大功能，应该拆成：第一步加注册页（用户能填表单提交），第二步加登录页（用户能登录退出），第三步加权限控制（登录用户能看到特定内容），

第四步加用户资料页（用户能查看和编辑自己的信息）。每一步都是一个“用户能感知的功能单元”，每一步都能独立验收。

每一步怎么执行？先用计划模式让 **Codex** 出方案，你审完确认。然后明确告诉它“本次只实现这一步，不要顺手做下一步”，把它框在当前这个功能单元里。

做完立刻验收，确认这一步真的完成了，再进入下一步。每步完成提交一次 **git**，留个版本节点，后面想回退也方便。

有一个反模式要警惕：让 **Codex** 顺手重构无关代码。**Codex** 在加新功能的时候，可能觉得现有代码“不够优雅”，顺手重构一下，结果引入各种意外。防御句式写进任务描述里：“本次只实现当前功能点，不要顺手重构无关代码，不要修改命名、目录结构、依赖版本和全局配置。”

加功能还要注意一个点：回归测试。改了首页按钮，也可能影响复用同一组件的其他页面。新功能加进去之后，不光要测新功能本身，还要测原有的功能有没有被影响。这种“回归测试”是最容易被忽略的——你盯着新功能，忘了确认 **Codex** 有没有顺手搞坏别的。让 **Codex** 跑一遍完整的测试套件，或者你手动点一遍核心流程，确认没引入新问题。

83. 老项目敢不敢交给 **Codex** 重构

敢，但要极其谨慎，而且一定要用 **Worktree** 模式。

老项目重构是 Codex 能力上限很高的场景，也是最容易翻车的场景。老项目通常代码多、历史包袱重、耦合复杂、没准还有一堆历史遗留问题。让 Codex 在上面做大改动，风险天然就高。

老项目有几个特有的坑得提前知道。首先，老项目本身就可能有 lint 或类型问题，别让 Codex 顺手重构——它一看到这些"问题"就想"修一下"，一修就牵连出一堆改动，把本来能跑的项目改崩。其次，老项目的历史包袱要尊重，有些代码看着"烂"，却是当年为了某个特殊原因写的，改了可能引入隐蔽的 bug。最要紧的是，老项目的耦合比你想象的复杂，改一个地方，可能影响你完全没想到的别的地方。

正确的做法有一套。一定要用 Worktree 模式，给 Codex 一个独立的工作副本去改，你的主项目纹丝不动；它改完了你审，审过了再并回主项目，改坏了直接丢掉 worktree，零风险。每次只让 Codex 改一小块，别"重构整个模块"，而是"只把这一个函数拆清楚""只把这一个组件抽出来"，改动范围越小，引入风险越低。每步都得跑完整测试，老项目能改的前提是有测试，没有测试的老项目，每改一步都得你手动验证核心流程。另外，看着 diff——Codex 改的每一行你都要看，发现它顺手改了无关代码立刻回退。

老项目重构还有一个适合用 Goal 模式的场景：代码迁移。把一个内部工具从 Python 迁移到 Rust。建好新目录，设定目标："直到所有单元测试全部通过，

这个新版本的开发才算完成。"Codex 持续往这个目标努力，每改一处都跑测试，直到所有测试通过。

老项目重构最重要的心态调整：慢就是快。不要急着让 Codex 一次重构一大块，宁可慢一点、稳一点，每步都验证。有一句话值得记住——交付不是结束，复盘才是下一次效率提升的开始。重构完一块，复盘一下：哪里翻车了、哪里 Codex 做得好、下次怎么拆更合理。这种复盘能让后续的重构越来越顺。

84. 写测试用例怎么交给 Codex

写测试是 Codex 的强项，写测试被推荐为稳定输出的关键技巧之一。

为什么写测试特别适合交给 Codex？因为测试的特点就是"明确输入、明确预期、验证输出"，这种规则清晰、可自动验证的任务，正是 agent 最擅长的。

Codex 能读懂你的代码、推断边界条件、生成各种测试用例，然后自己跑一遍验证哪些通过哪些失败。

最有效的提法是：先让 Codex 读懂要测的代码，再让 Codex 生成测试。不要直接说"给这个函数写测试"，而是说"读这个函数的实现，找出所有边界条件，为每个边界条件写一个测试用例，跑一遍确认都能通过"。这种提法能让 Codex 真正理解代码、找全边界，而不是机械地生成几个常规用例。

还有一个反向用法：先写测试，再让 Codex 实现。这种思路特别适合新功能。

你先告诉 Codex"这个功能要满足这些测试用例"，让 Codex 看着测试去实现功能。测试是明确的、可验证的，Codex 实现的代码只要能让所有测试通过，就算完成。这种"测试驱动"的方式，比"实现完再补测试"稳定得多，

Superpowers Skill 就是这种思路——先澄清需求、写规格、做计划、用 TDD 推进。

写测试有几个坑要避免。别让 Codex 只写"happy path"测试——它倾向于写"正常情况能跑通"的测试，可真正有价值的是边界测试，比如空输入、超长输入、非法输入、并发情况。明确让 Codex"重点测边界条件"，不然给你的测试覆盖不了真实场景。还有一个隐蔽的坑：测试和实现可能一起跑偏。有时候 Codex 写测试按自己的理解，写实现又按另一个理解，结果测试通过了但功能是错的——防御办法是先确认测试用例本身对不对，再让 Codex 实现。另外，测试也要看 diff，你不看就接受，可能引入一些没意义的测试、漏掉关键测试，扫一眼它写的测试，确认覆盖了你想覆盖的场景就行。

写测试是那种"前期投入、长期受益"的事。第一次让 Codex 写测试可能比直接让 Codex 实现慢，但有了测试之后，后续每次改动都能跑测试验证，回归问题大幅减少。写测试被列为稳定输出的关键技巧之一，理由就在这里。

85. 让 Codex 做代码审查靠谱吗

靠谱，而且有些方面比人审更细，但要会用。

Codex 做代码审查，干的事就是让 Codex 读一组代码改动，找出其中的问题。

推荐两轮审查的做法。第一轮让 Codex 自审，Codex 输出一份问题清单，把

明显的 bug、边界问题、安全隐患先暴露出来。这一步相当于"让写代码的

Codex 自己挑自己的错"，往往能挡住低级问题。第二轮人工审，你看着

Codex 列出来的清单和 diff，重点判断那些 Codex 发现的问题是不是真问题、有没有遗漏。

这种"一个写、一个挑错"的模式，特别推荐用于高风险场景：涉及登录、支付、权限、数据库、鉴权的改动，建议用第二个模型做交叉审查。第一个模型写代码，第二个模型审代码，两个模型从不同角度看，比一个模型自己写自己审靠谱得多。

Codex 做代码审查擅长什么？边界条件它能想到各种边界输入，提醒你哪里没处理；安全问题，像常见的注入、越权、敏感信息泄露，它能识别出来；明显 bug，比如空指针、数组越界、逻辑错误，这些它也能发现；还有风格和一致性问题，命名不一致、不符合项目约定，它也能指出。

Codex 做代码审查的局限也要清楚。业务逻辑它看不出——你的代码是不是符合业务需求，它没法判断，因为它不懂你的业务，这部分得你来审。架构合理性它的判断也有限，能看出局部问题，但对整体架构是不是合理，它说的不一定靠谱。

还有一点，它可能误报，说"这里有问题"，其实是你故意这么写的，它不知道上下文。所以它的报告你要判断，不能全信。

代码审查还有一个特别适合的场景：在合并前审一遍。让 Codex 读 diff，列出问题清单，你逐条判断。这比你自己逐行读 diff 高效，因为人的注意力有限，盯着 diff 看久了会漏掉问题，而 Codex 能从头到尾保持一致的细致度。但永远记住那条硬规矩：不要让 AI 自动合并 PR。Codex 能审，能发现问题，但"改不改、合不合并"的决策永远是你做的。Codex 的审查是辅助，不是替代你的判断。

86. 团队多人协作能用吗

能用，但和单人用法不一样，有一些特别注意点。

先说项目文件夹的协作。Codex 工作在一个本地文件夹里，如果团队多人都要用它处理同一个项目，最稳的方式是 Git。每个人在自己电脑上跑 Codex，改完提交到 Git，互相通过 Git 协作。千万别让多个人同时用 Codex 改同一个文件夹——文件状态会冲突，改动会互相覆盖。

更关键的是 AGENTS.md 在团队里的价值。团队场景下，把团队的代码规范、技术栈约定、不能动的文件、提交规范都写进 AGENTS.md，放在项目根目录。

这样不管团队里谁打开这个项目用 **Codex**，**Codex** 都按同一套规矩来，输出风格统一。**AGENTS.md** 本身也提交到 **Git**，团队所有人共享同一份规则。

Skill 同样可以团队复用。团队里某个成员把某个工作流做成了 **Skill**，共享给其他人就行——**Skill** 就是文件夹里的 markdown 文件，提交到 **Git** 仓库就能让所有人用。比如团队统一一个“代码审查 **Skill**”，所有人审查代码的时候都按这个 **Skill** 来，审查质量自然统一。

另外两点也得心里有数。额度是按账号算的，团队多人用 **Codex**，每个人用自己的 **ChatGPT** 账号，额度独立；想统一管理额度，可以考虑 **Team** 或者 **Business** 套餐，**Business** 套餐有团队管理功能，但具体套餐和额度会变，以官方页面为准。还有敏感数据要小心——团队场景下让 **Codex** 处理的数据，可能涉及客户数据、商业机密、内部系统，处理之前要想清楚数据会经过 **Codex** 的模型，符合公司的数据安全政策；真正机密的内容，谨慎处理或者干脆不用 **Codex**。

团队协作的核心原则是：**Codex** 是个人工具，团队协作靠 **Git** 和共享配置文件。每个人用自己的 **Codex** 干自己的活，通过 **Git** 合并工作成果，通过共享的 **AGENTS.md** 和 **Skill** 保持一致性。不要试图把 **Codex** 当成“团队共享的工具”，**Codex** 的设计不是为多人同时操作同一个文件夹准备的。

87. 想让 Codex 半自动跑长项目该怎么搭

很多人听完自动化的故事，以为"完全自动化"就是丢一个 prompt 给 Codex 然后去睡觉。社区里真做过的人会告诉你：这种玩法风险极高，真正的"半自动"是另一套东西。

被多人验证过的半自动工作流，由三样东西撑起来：一份任务清单、一个监工机制、频繁的版本留底。

任务清单是你用 ChatGPT 先把需求拆成结构化的清单，写进一个文件（比如叫 `task-list.md`），当作 Codex 的"进度账本"。Codex 知道全局要做什么、做到哪一步了，照着清单往下推进。

监工机制（社区里俗称 `watchdog`，可以理解成"监工"）负责周期性检查清单进度，把跑偏的 Codex 拽回正轨。Codex 长任务里最容易偏航，光有清单不够，还得有东西盯着它别走神。

版本留底就是每完成一步就给你的项目存一个版本快照（用 Git 就是 `commit` 一次，不用 Git 就复制一份留底）。万一某一步跑崩了，你能回退到上一个已知良好的状态，不至于全军覆没。

这套循环能跑到什么程度？社区里有实测：数小时无人值守、跨多文件边实现边测试的循环，是可达成的，有人用这套跑过 8 小时的长任务。但现实的边界也

要说清楚。消耗大，一次数小时自主跑可能吃掉周额度的一成多。会偏航，所以才需要监工机制。风险高，无人监督的 Codex 是最危险的。

所以"完全自动"这个词得理解对：真正的目标，是搭一套"任务清单 + 监工 + 频繁版本留底"的半自动循环，让你能在全程盯着的情况下持续推进，同时保留随时介入和回退的能力，而不是"一个 prompt 然后睡觉"。比那种"全交给 AI"的幻想，这种半自动循环诚实得多，也实操得多。

这一套偏进阶，如果你刚上手，先把前面 Q72 到 Q86 的基础场景跑顺，再考虑半自动。普通用户不一定要碰它，知道有这么个方向就行。

六、边界、替代与反思

88. 什么时候该继续用 Codex 其实已经错了

真正会用一个工具的人，不光知道什么时候用 Codex，更知道什么时候不该用。

Codex 也有边界，过了边界硬用，只会烧钱受气。

最常见的是把它当聊天机器人。你只是想问个概念、聊个思路、要一段通用代码片段，这种事丢给普通 ChatGPT 对话就行，几秒就回，不烧额度，让 Codex 等几分钟纯属浪费。一个简单的判断标准：一般任务用普通对话，要处理本地文件或项目才动用 Codex。

再有一种，是拿它干没有验收标准的事。"帮我评估这个设计好不好""帮我想几个产品方向""帮我看看这个市场怎么样"，这类主观判断的活，Codex 的执行能力派不上用场，结果只是用更慢、更贵的方式做了一个普通对话能做的事。

更麻烦的是把它当作"替你想清楚需求"的工具。你脑子里对要做的东西还一团模糊，就丢给它跑，Codex 只会按自己那套模糊理解往下走，结果大概率不对。它没法替你想清楚要什么，它只会替你猜。

最要紧的是高风险且不可逆的事。让 Codex 操作付款、改生产数据库、删重要文件、动银行账号，这些一旦出错没法收。凡是涉及付款、账号设置、删除数据的操作，最好别让 Codex 碰。

判断该不该用 Codex，可以顺着这个思路捋一遍：这事到底要不要 Codex"动手执行"？做完了你能不能验收？万一错了代价大不大？几条都想清楚了，再决定是动用 Codex，还是换个工具、自己上手。

89. 不想买 Plus 有没有别的平替

有，而且对中国用户来说，平替可能是更优解，不是凑合。

最主流的平替是接国产模型。Codex 不一定要用 OpenAI 官方额度，Codex 支持通过 CC Switch 这种工具接入国产模型——智谱 GLM、Kimi、DeepSeek 是最常用的几家。月费比 Plus 便宜一大截，还不用梯子，国内直连就能用。

很多人日常就是挂着国产模型在用。有人切到 Kimi，说 Kimi 是国内接入 Codex 效果最好的模型之一，原生支持多模态，能识别图片和视频，做 PPT、表格、网页生成审美也很好；有人挂智谱 GLM，说 GLM-4.7 是国产编程模型第一梯队；有人用 DeepSeek，图的就是价格最低，适合预算敏感的人。

但接国产模型这条路，有几个前提得先讲清楚。最根本的一条是，这是非官方玩法，不属于 OpenAI 官方功能，所以模型兼容性、稳定性、隐私、费用、工具

调用的支持程度，都得看第三方。配置这一关也比直接登 ChatGPT 账号麻烦得多——得用 CC Switch 这种桥接工具，配 API Key、设路由，不是点一下就完事。功能上同样可能有差异，OpenAI 自家模型在 Codex 上的某些能力（比如某些插件、某些工具调用），国产模型不一定完全兼容。还有一句务实的提醒：重要项目先用测试仓库跑通，再往生产项目里上。

CC Switch 这个工具值得多说两句。CC Switch 把“来回改配置文件”这件事图形化了。以前想切换不同模型，得手动改配置文件里的接入地址和密钥，存盘重启。CC Switch 把这套流程变成了托盘里点一下就完事。CC Switch 还能统一管理 MCP、Skills，统计用量和消耗。对玩多家国产模型的人来说，CC Switch 几乎是必装的工具。

除了接国产云端模型，Codex 现在的格局更大了：它已经支持直接接入任意开源模型——Ollama 本地模型、LM Studio、vLLM 这些都能接进来。换句话说，你甚至可以在自己电脑上跑一个本地小模型，零成本用 Codex 的界面和工作流。更骚的是配置方式：不用自己手改配置文件，直接跟 Codex 说一句话——“把 Ollama 配成 Codex 自定义模型，模型用 qwen3:8b，Base URL 用 localhost:11434”——Codex 会自己去读 config.toml、加配置、检查语法、告诉你要不要重启。全程你不碰一行配置代码。让 Codex 配 Codex 自己，这种递归感很极客。

这种“什么模型都能接”的开放性，带来一种新的省钱思路：分层处理。便宜的本地小模型或者国产轻量模型，放在检索、改小文件、跑批处理这类简单任务上，响应快还省钱；复杂规划和难活再交给强模型。生产环境本来就该这么分层——简单任务用本地小模型，复杂推理才动用贵的云端大模型，整体成本结构会灵活很多。

接国产模型或开源模型适合什么人？预算敏感、不想折腾梯子、愿意花点时间做配置、能接受非官方玩法的兼容性风险。如果你追求最稳、最省心、最完整的功能，还是 ChatGPT Plus 直接登录。两条路各有代价，看你更在乎什么。

90. Codex 和 Claude Code 怎么选

Codex 和 Claude Code 是同一个生态位的东西，定位相似但风格不同。重度用户很多是两个装着用，根据场景切换。

先说定位差异。Claude Code 是 Anthropic 出的命令行 agent，跑在终端里。

Codex 有 Codex、CLI、IDE 插件、云端多种形态，对普通人更友好的是 Codex。两者都能读你的项目、改文件、跑命令、交付结果，核心能力是重合的。

再说各自的优势。有一句总结很到位：Claude Code 啃复杂代码库更稳，

Codex 写 Python 数据脚本和前端样式比较顺。Claude Code 因为是纯命令行，跟终端、git、shell 脚本配合更顺，程序员用起来如鱼得水。Codex 因

为是图形界面，能开多个项目并行跑、内置浏览器看效果、能操控电脑，对非程序员和不碰命令行的人更友好。

能力上的差异。横评显示，Claude Code 在 SWE-bench Verified（一个主流的软件工程基准测试）上领先，复杂多文件工作输出质量更高。Codex 在 token 效率上表现好，做同样的任务消耗更少。这意味着 Claude Code 更适合复杂、对质量要求高的任务，Codex 更适合追求性价比的场景。

社区里有一种比喻很到位：**Codex** 像个靠谱的中级工程师，听话、快、少翻车；**Claude Code** 更像个能跟你讨论架构的资深，更聪明、更有想法，但也更贵、更慢、偶尔跑偏。Codex 在可靠性、失败频率、沙箱体验、每日额度这些"日常干活"的维度上口碑更好；Claude Code 在架构推理、长会话记忆、深度调试这些"啃硬骨头"的维度上更强。一个被反复提到的差别是：Codex 适合"明确任务交给它执行"，Claude Code 适合"模糊需求一起探索"。

有个有意思的细节。Codex 做 code review 的时候，会给每个问题标优先级，这一点社区很喜欢。但 Codex 很少解释"为什么是这个优先级"，透明度不如 Claude。这也是两者风格差异的一个缩影：Codex 偏执行、少解释；Claude 偏推理、多讨论。

那普通人到底该怎么选？头一个看你会不会用命令行：完全不想碰终端，Codex 几乎是唯一选择；习惯命令行，Claude Code 体验更纯粹。再看做的事的类型

—写代码、改项目、做技术活，两者都行，看你对哪个模型更信任；而做内容、做 PPT、处理办公文档、操控电脑这些非编程的事，Codex 因为有图形界面和各种插件，能力强一截。预算也得算进去，Claude Code 要 Anthropic 的订阅或 API，Codex 要 OpenAI 的订阅或国产模型接入，两边价格不一样，国产模型接入两边都支持。

重度用户的真实做法是：两个都装着用。啃复杂代码库、做严肃的工程任务，回 Claude Code；做前端、调样式、做内容、处理办公事务，用 Codex。说到底这无关站队，只是根据任务挑合适的工具。

91. 已经在用 Cursor 了还有必要开 Codex 吗

看你用 Cursor 干什么。Cursor 和 Codex 是两种不同的东西，不是替代关系。

Cursor 是一个完整的 IDE（代码编辑器），Cursor 的 AI 能力嵌在编辑器里。你写代码的时候，Cursor 能帮你补全、帮你改、帮你查。Cursor 的强项是"在你写代码的过程中实时辅助"，是一种嵌入式的体验。

Codex 是一个独立的 agent，Codex 能进入你的项目、独立完成任务、交付结果。Codex 的强项是"你把一个完整的活交给 Codex，Codex 替你从头干到尾"。这种模式下，你不用全程盯着代码，Codex 自己跑。

有一种用法很好：用 **Cursor** 做日常编码和局部修改，用 **Codex** 做任务推进和工程交付。这两者是互补关系，不必二选一。日常你在 **Cursor** 里写代码，需要 **AI** 帮你补全、改某个函数、解释某段逻辑，用 **Cursor** 顺手；当你有一个明确的任务要交给 **AI** 独立完成—做个新功能、修个复杂 bug、重构一个模块—把任务丢给 **Codex**，让 **Codex** 替你跑。

还有一个差异是能力范围。**Cursor** 的能力主要在编码相关的事，**Cursor** 能做的事都围绕"代码编辑"展开。**Codex** 的能力更广，**Codex** 能做 PPT、处理 PDF、操控电脑、做视频、剪素材、分析数据，这些 **Cursor** 都做不了。如果你除了写代码还有大量非编码的事要处理，**Codex** 的价值更明显。

判断要不要同时用，可以问自己两件事。一件是你的工作是不是只有写代码—是，**Cursor** 可能就够；不是，**Codex** 能帮你处理一堆 **Cursor** 处理不了的事。另一件是你更偏爱"边写边让 **AI** 辅助"，还是"把整个任务丢给 **AI** 跑"—前者 **Cursor** 体验更好，后者 **Codex** 更合适。

有一个观点值得记住：工具站队是让 **Codex** 只发挥 3% 潜力的习惯之一。根据身份而不是任务选择工具，是错的。该用什么工具，由当前的任务决定，不由"我是哪个工具的人"决定。

92. **Codex** 的额度会一直涨下去吗

这个问题没人能给确定答案，但可以从趋势里看出一些方向。

从历史看，额度是越来越宽松的。早期的 Codex 额度非常紧，免费档几乎没法用，Plus 也很容易撞顶。随着模型成本下降、用户规模上来、竞争加剧，OpenAI 在持续放松额度。ChatGPT Pro 5x 套餐给到 Plus 的 5 倍额度，还经常有临时加额的活动（比如某段时间 2 倍额度提升）。整体趋势是往更宽松走。

但额度放松不等于“无限”。agent 类任务的成本天然比普通对话高，每跑一轮都要消耗算力，这部分成本是真实的。OpenAI 不可能无限放松，否则商业模式撑不住。社区里一直有用户呼吁更透明的额度系统、更可视化的用量预测，说明现在的额度管理还不算完美，用户焦虑依然存在。

未来额度怎么走，还看几个变量。最直接的是竞争，Claude Code、Cursor 这些对手在后面追，OpenAI 想留住用户，额度就得保持吸引力。模型推理成本如果继续降，额度自然也有放松的空间。规模也在帮忙——Sam Altman 在 2026 年 4 月透露 Codex 已经有约 400 万周活跃用户，规模效应能摊薄成本。还有商业化策略这一层要警惕，OpenAI 可能把更多能力塞进更高档的套餐里，看着基础额度放松了，高级能力却悄悄变贵。

对普通用户来说，与其猜未来额度怎么变，不如建立“不依赖大量额度”的使用习惯。小步用、按需用、盯消耗、不浪费。这种习惯不管额度怎么变都不会吃亏。

反之，如果你养成“反正额度多就乱用”的习惯，等哪天额度政策变了，你会非常不适应。

93. 为什么 OpenAI 把 Codex 越做越像操作系统

这是个有意思的观察。Codex 现在能管文件、能装应用（插件）、能定时任务（自动化）、能记住你的习惯（记忆）、能控制其他应用（操控电脑）。这些能力加起来，Codex 确实越来越像一个“AI 操作系统”。

为什么会这样？OpenAI 团队的思路透露了逻辑。OpenAI 认为，我们在电脑上做的大部分工作，都和代码息息相关：执行终端命令、浏览网页、调用 API、导出文档、响应事件、触发自动化。当 Codex 延伸到这些领域，Codex 给人的感觉就不再只是编程助手，而是能帮你搞定各种电脑工作的“全能打工人”。

这个方向的逻辑是合理的。如果你有一个能执行任务的 AI，把 Codex 限制在“只写代码”是浪费。让 Codex 能管文件、能操作应用、能定时跑任务、能记住你的偏好，Codex 的价值会指数级放大。一个真正能干活的 AI 助手，自然会演化成一个“操作系统”——因为 Codex 需要调度各种资源来完成任务。

对用户来说，这是好事也是隐忧。好事是，你越来越多的事可以交给 Codex，效率提升明显。隐忧是，Codex 对你的电脑、你的数据、你的工作流介入越来越深，依赖越来越重。那些“幕僚长”对话流、“产品发布”对话流、自动化任务

定时跑、知识库持续更新——这些场景里，Codex 已经深度嵌入他们的日常工作，离开 Codex 会很不适应。

这种"AI 操作系统化"还在早期，未来会演化到什么程度不好说。但趋势是明确的：Codex 不满足于只做编程工具，Codex 在往"你电脑上的 AI 总管"方向走。这个方向最终能不能成、用户买不买账，要看 Codex 在便利和风险之间能不能找到平衡。

94. Codex 会不会很快被新东西替代

可能会。但要担心的，未必是"另一个 Codex"这类同形态竞品——更值得警惕的是更深层的范式变化。

短期内，Codex 这个形态（本地装一个 AI 应用、能读文件、能操控电脑、能跑任务）大概率会持续存在。这个形态解决了真实需求，用户也有粘性，OpenAI 自己也在持续投入。Sam Altman 透露的 400 万周活数字说明这个产品已经有规模，不会被轻易放弃。

但中长期看，几个变量可能改变格局。最显眼的是模型本身——如果模型变得足够强、足够便宜，"本地装一个独立应用"的必要性可能下降，更多能力会直接嵌进操作系统、嵌进浏览器、嵌进你用的各种工具里。交互方式也值得留意，现在你和 Codex 之间主要靠打字，要是语音、视觉、意图理解成熟到一定程度，

交互的形态可能完全变样。更外层还有平台策略这一层，Apple、Microsoft 这些操作系统厂商自己在做 AI 能力，一旦把 AI 深度集成进系统，第三方 AI 应用的空间就会被挤压。

对用户来说，怎么应对这种不确定性？头一条是别把鸡蛋放一个篮子里。Codex 是好工具，但你的核心能力——理解问题、拆解任务、判断结果——应当是工具无关的，换了工具照样能用，这才算真能力。再者，要把功夫下在方法论上：具体的命令、界面、配置项会变，但“怎么给 AI 清晰的任务”“怎么验收结果”“怎么拆解复杂任务”这些方法是通用的。也别因为已经摸熟了 Codex，就对新东西关上门——新工具出来的时候试一试，往往有意外收获。

根据身份而不是任务选择工具是错的。同理，根据“我是哪个工具的用户”而拒绝尝试新工具也是错的。工具会变，你解决实际问题的能力不能被某个工具绑死。

95. 普通人真的需要 AI agent 写代码吗

这个问题没有统一答案，取决于你想解决什么问题。

先说不需要的情况。如果你完全不想理解代码、不想理解技术、只想让 AI 替你把所有事做完，那 agent 对你的价值有限。因为 agent 给你的结果你得能验收，验收不了的话，AI 做得再好对你也没用。Codex 能提高效率，但不能替你做判断。如果你连判断都做不了，agent 帮不到你。

再说需要的情况。你有具体想解决的问题——做个个人网站、做个小工具、自动化处理一些数据、分析一些文档——这些事以前要么得自己学编程、要么得找程序员帮忙。现在有了 `agent`，你不需要成为程序员，但你需要能描述清楚需求、能验收结果。这种“不写代码但能用 AI 做出东西”的能力，对普通人来说是非常实用的。

那些非程序员的案例很有说服力。做电商的用 `Codex` 剪视频、做内容创作的用 `Codex` 写文章、做业务的用 `Codex` 对账、普通人用 `Codex` 分析体检报告。这些人人都不是程序员，但他们用 `Codex` 解决了真实的、具体的问题。对这些人来说，AI `agent` 早就成了手边一件再普通不过的工具，该用就用。

更深层的问题是：AI `agent` 让“会编程”这件事的门槛发生了变化。以前会编程意味着能写代码，现在“会编程”更多意味着能和 AI 协作把事情做出来——能描述需求、能验收结果、能判断对错、能迭代优化。这种“新会编程”的能力，对普通人来说越来越值得掌握。

所以普通人需不需要 AI `agent` 写代码，跟“是不是程序员”关系不大，更要紧的是你有没有想用代码解决的问题，愿不愿意学着用 AI 去解决。有，就值得用；没有，先找问题再说。

96. 把整个项目交给 AI 改风险有多大

很大，而且比你以为的更大。这是高风险场景。

风险来自好几个方面。最先是 **Codex** 看不到全局——即便它能读你整个项目，注意力也是局部的。它改 **A** 的时候，常常意识不到 **A** 还牵连着 **B**、**C**、**D**，改完 **A** 把那几处搞坏了，它自己还不知道。更阴险的是它会顺手重构：觉得某段代码"不够优雅"就动手，重命名一个组件，所有引用都得跟着改，改错一处就报错；改全局样式，别的页面布局跟着崩；最要命的是删代码，它觉得"没用"的，往往是你的业务逻辑。还有认知上的错位——**Codex** 觉得核心功能跑通就算完事，你却还要处理边界情况、写测试、补文档，结果到手的是个半成品。再加上回退成本高，它一改二十个文件，你发现方向错了，回退常常比从头来还麻烦。

防御这些风险的办法其实很全。头一条，别让 **Codex** 一次改一大块——六步法、小步实现，都是为了把改动范围圈住。配套的是 **Worktree** 模式：给它一个独立副本去改，主项目不动，改坏了直接丢。每一步还要看 **diff**，发现它顺手改了无关代码立刻回退。指令上要写防御性的话，比如"本次只实现当前功能点，不要顺手重构无关代码，不要修改命名、目录结构、依赖版本和全局配置"。最后别忘了回归测试，改完不只测新功能，还要把原有功能过一遍，看有没有被牵连。

最根本的心态是：把整个项目全权交给 **Codex** 去改，是最危险的做法。哪怕你对 **Codex** 再信任，哪怕 **Codex** 历史表现再好，这种"放手让 **Codex** 干"的用

法早晚会翻车。正确的做法是"Codex 改一块、你看一块、确认一块、再让 Codex 改下一块"。慢一点，但稳。有一句话值得反复记：AI 改得很快，但你不知道 Codex 改了什么，也无法放心交付。

97. 重度依赖 Codex 会不会让我退化，还算我会写代码吗

这两个问题其实是一回事：工具替你做了越来越多事之后，你的能力是涨了还是废了，你这个人还算是"会"那个事的人吗。

先说退化。退化的风险是真实的。如果你所有事都丢给 Codex，自己不思考、不判断、不学习，长期下来你的某些能力确实会萎缩。就像 GPS 出现之后，很多人的方向感变差了一因为你不再需要自己记路、认路。同理，如果你所有代码都让 Codex 写、所有判断都让 Codex 做，你的代码能力和判断能力可能会变弱。

但这种"退化"不是必然的，取决于你的用法。那些把 Codex 用得很好的人，他们并没有退化，反而在进化。原因在于他们用的是"协作模式"——他们出思路、Codex 执行、他们验收、Codex 再优化。这种过程里，他们学到了新东西，他们的判断力、拆解问题的能力、对代码的理解，都在提升。反之，"外包模式"下把所有事丢给 Codex、自己不参与，那确实会退化。

判断自己到底是协作还是外包，有几个信号可以照一照。你能不能解释 Codex 给你的代码？能解释，说明你理解了，是协作；解释不清，多半只是接受了结果。Codex 出错的时候你能不能发现？能发现，说明你还在判断；发现不了，那就是完全依赖。还有一条更长远：你从 Codex 身上有没有学到东西？学到了，说明你在成长；什么都没学到，说明你在被动接受。

再说"算不算会写代码"这个身份焦虑。本来就会写代码的人，Codex 只是个更聪明的工具，就像 IDE 的自动补全、Stack Overflow 的答案一样，是辅助，你当然还是"会写代码"。不太会写代码、主要靠 Codex 做事的人，你能描述需求、能验收结果、能判断对错、能迭代优化，但你不一定能独立从零写出这些代码——这种情况算不算"会写代码"，要看你怎么定义。如果定义是"能独立产出代码"，那你不算；如果定义是"能让代码被做出来解决你的问题"，那你算。

这个定义之争本身没那么重要。重要的是它揭示了一个现实："会写代码"这件事的含义正在变化。以前"会写代码"意味着掌握语法、能动手写；现在"会写代码"越来越多地意味着能和 AI 协作、能把问题拆解清楚、能验收结果。这是一种新的能力，不比前者低级，只是不同。那些非程序员用 Codex 做出了真实可用的东西——网页、PPT、视频、数据分析流程——他们大概率没法独立写出这些代码，但他们解决了真实的问题。他们算不算"会写代码"不重要，他们用工具把事情做成了，这才是重点。

所以更健康的心态是：不要把“会不会写代码”当成一个身份标签去捍卫。你能不能解决问题、能不能让 AI 帮你做出想要的东西、能不能持续学习和迭代——这些比“算不算会写代码”重要得多。

防御退化的几个习惯也顺便说一下。头一条，重要的判断得自己做——Codex 给方案你审，Codex 给代码你看懂，Codex 给建议你权衡。再一条，保留“手动”的能力——重要的事偶尔自己动手做一遍，别让肌肉记忆完全退化。还可以把 Codex 当老师而不只是工具，让 Codex 解释为什么这么写、为什么这么改。最后，把注意力放在方法论上——具体的命令、配置会变，但“怎么拆解任务、怎么验收结果、怎么和 AI 协作”这些方法论会跟着你一辈子。

工具让某些能力变得不那么必要，同时让另一些能力变得更值钱。你的方向感可能变差了，但你能去更多地方；你的代码手感可能退化了，但你能做更复杂的东西。这种演化是好是坏，看你怎么用 Codex。

98. 免费用户有没有什么省着用的方法

免费额度少，但有些方法能让你在有限额度里做更多事。

最直接的省法，是把最重的活分摊给免费的外部工具。比如做大量调研、查资料这种事，用免费的 ChatGPT 普通对话、Perplexity、Kimi 就够了，不必动

Codex。把它那点额度留给那些"必须 Codex 执行"的任务—改你本地文件、操控你电脑、跑你项目里的代码。

更划算的是接国产模型扩充额度。CC Switch 加上智谱、Kimi、DeepSeek，月费比 Plus 便宜一大截，有些还带免费额度。把不重要的任务切到国产模型上跑，省着用 OpenAI 的额度；国产模型在 Codex 上的兼容性虽不完美，日常任务基本够用。

任务本身的档位也要会挑。改个变量、问个小问题、做个小调整，这种就用最低推理强度，几秒就完，消耗最少，别什么事都开最高档，那是浪费。动手之前，最好先在计划模式里把路想清楚—规划阶段消耗少，方向对了执行阶段才不浪费；直接把大任务丢出去让 Codex 乱跑，跑偏了烧的额度更多。

执行中要舍得喊停。发现 Codex 跑偏、打转、改不对，立刻叫停，别让 Codex 无限试下去，每多一轮都是额度。日常还要养出一套"最省"的工作流：能用项目模式就别让 Codex 扫整个项目，能指定文件就别让 Codex 猜，能用 Skill 固化流程就别每次重新解释。这些小习惯叠起来，能省不少。最后一句容易被忽视：好结果跑出来，立刻复制出来、提交到 git。免费额度换来的成果不存下来就没了，别让 Codex 停留在"只在对话里"。

免费额度的核心策略是"精打细算"。每个任务都想想有没有更省的办法，每次用都想想这次是不是真的需要 Codex。养成这种习惯，免费额度能撑很久；不养成这种习惯，给你多少都不够。

99. 什么时候该回到手动写代码

会用工具的标志是知道什么时候不用工具。手动写代码在以下场景仍然有不可替代的价值。

比如你需要深度理解某段代码的时候。让 Codex 写完你直接拿去用，和你自己写完、弄懂每一行，是两回事。那些重要的、核心的、经常要改的代码，自己写一遍才能让你真正吃透它，这种理解在后续维护、调试、扩展的时候会救你。

再比如要做特别精细、特别讲究的活。Codex 擅长"做到差不多"，却不太擅长"做到极致"。一个性能极致优化的模块、一条算法的关键路径、一个对细节苛求的交互，自己动手更靠谱——给它 80 分容易，让 Codex 给到 100 分很难。

涉及核心业务逻辑和关键安全的地方更要自己上手。权限和安全的核心模块、支付的关键逻辑、生产数据库的操作，这些都不该全权交给 Codex。这些位置一旦出错代价太大，必须攥在自己手里。

还有你正在学习的时候。学习阶段，自己写、自己错、自己调，是建立理解的过程，让 Codex 替你写，你学不到东西。这时候 Codex 更适合当陪练——用来解释、答疑、对比方案，而不是替你交作业。

最后一种情况，是 Codex 反复做不对的事。如果你试了好几次，它在某个特定任务上就是卡住，那多半是这个任务超出了它的能力，或者这件事需要一种它给不了的判断。这种时候自己动手往往更快，别和它较劲。

判断该不该手动，问自己：这事重要吗？重要到自己应该完全理解？Codex 做出来的东西我能验收吗？Codex 做的够不够好？如果答案是"重要、应该理解、我能验收但不够好"，自己动手。

手动写代码谈不上落后，反倒是一种该被保留的能力。工具再强，你的核心能力也不能完全外包给工具。有一句话值得记住：Codex 能提高效率，但不能替你做判断。判断的核心，恰恰是你通过手动做事积累出来的。

100. 用了一阵之后最该复盘哪件事，长期最大的收获是什么

最该复盘的是：你能不能解释 Codex 给你做的每一个东西。

这句话听着简单，做起来会暴露很多问题。很多人用了一阵 Codex，做了不少事，但你问 Codex"这段代码为什么这么写"，Codex 答不上来；"这个功能是

怎么实现的", Codex 答不上来; "为什么用这个方案不用那个", Codex 答不上来。如果答不上来, 说明你在"外包"而不是"协作", 你在被工具推着走, 而不是驾驭工具。

复盘的具体做法可以这样展开。挑你最近做的几个项目逐个走一遍, 每个项目问自己: Codex 做了什么? 为什么这么做? 有没有更好的做法? 如果 Codex 没做对, 问题出在哪? 你能回答这几个问题, 说明你理解了; 答不上来, 说明你只是接受了。

顺便把有效的任务描述方式沉淀下来。哪些提法效果好? 哪些会跑偏? 形成你自己的"任务描述模板", 好用的 Prompt 不要每次重新写。用了一阵, 你应该也更清楚哪些规矩需要明确、哪些流程需要固化—把这些更新到 AGENTS.md 和 Skill 里, 让 Codex 的表现更稳定。还有你踩过的坑, 哪些翻车重复出现、根因是什么、怎么避免, 记录下来, 下次别再踩同样的坑。尝试、失败、记录失败、不再重复同样的错误, 这是 AI 辅助工作中最重要的习惯。额度使用也顺便评估一下, 哪些花得值、哪些是浪费, 据此调整后续用法。

复盘的核心目的是从"被工具推着走"变成"驾驭工具"。那些把 Codex 用得特别好的人, 差别往往不在用得多久, 倒在复盘得更勤上—他们知道每次为什么成功、为什么失败、下次怎么调整。这种持续迭代的能力, 比任何具体技巧都重要。

而长期下来最大的收获，其实也就是这种持续复盘长出来的能力——你和 AI 协作把事情做出来的能力。效率会提升，东西也能做出来，但这些是表象。真正值钱的是几层更基础的能力。比如拆解问题——要让 AI 听懂你的需求，你得先把模糊的想法拆成清晰的步骤，这种拆解能力反过来让你自己思考问题也更清楚。再比如描述需求——把脑子里的东西用语言精确地表达出来，让另一个“智能”能照着做。还有验收结果——判断一个结果对不对、好不好、是不是真的解决了问题，这种能力让你在面对 AI 输出的时候不被带着走。最后是迭代优化，第一次做不对，调整需求、调整思路、再来一次。

这些能力合起来，就是让你从“动手干活的人”变成“提需求、验收结果、做决策的人”。这种转变谈不上让你变懒，反倒让你升了一层——你不再被重复性劳动困住，精力可以放在更有价值的判断和创造上。

还有一个更隐性的收获：你建立了一种对 AI 的“健康信任”。你既不盲目相信 AI（知道 AI 会跑偏、会顺手改错、会给半成品），也不盲目排斥 AI（知道 AI 能做很多事、能提效、能解决真实问题）。你清楚地知道 AI 的边界在哪、你的边界在哪，在合适的场景用 AI、在不合适的场景不用。这种判断力，是用任何教程都学不到的，只能靠真正用一阵、踩几次坑、复盘几轮才能长出来。

有一句话特别值得作为收尾：当你把 **Codex** 当成一个可以配置、可以持续改进的队友来对待，它的效果最好。这话反过来也成立：当你把 Codex 当成队友，

你自己也会成为一个更好的"使用者"—更会提需求、更会验收、更会协作、更会判断。这种相互的成长，是长期用下来最值钱的收获。

工具会换，模型会变，但和 AI 协作把事情做出来的能力，会跟着你很久。这是用 Codex 这件事，能给你的最大的东西。

七、参考资料

官方文档

- <https://openai.com/index/introducing-the-codex-app/>
- <https://developers.openai.com/codex/>
- <https://developers.openai.com/codex/app>
- <https://developers.openai.com/codex/app/computer-use>
- <https://apps.microsoft.com/detail/9plm9xgg6vks>
- <https://developers.openai.com/codex/changelog>
- <https://developers.openai.com/codex/integrations/github>
- <https://agents.md/>
- <https://help.openai.com/en/articles/11369540-using-codex-with-your-chatgpt-plan>
- <https://chatgpt.com/codex/pricing/>
- <https://developers.openai.com/codex/learn/best-practices>
- <https://developers.openai.com/blog/run-long-horizon-tasks-with-codex>

社区讨论

- <https://www.reddit.com/r/codex/>
- <https://www.reddit.com/r/codex/comments/1rn6wpf/>
- <https://www.reddit.com/r/codex/comments/1ue9moq/>

- <https://www.reddit.com/r/codex/comments/1tlff44/>
- <https://www.reddit.com/r/codex/comments/1to7zhc/>
- <https://www.reddit.com/r/codex/comments/1tjfxcf/>
- <https://www.reddit.com/r/codex/comments/1seg8di/>
- <https://www.reddit.com/r/codex/comments/1r9619s/>
- <https://www.reddit.com/r/codex/comments/1u97eq0/>
- <https://www.reddit.com/r/codex/comments/1t7wx3g/>
- <https://www.reddit.com/r/codex/comments/1omdoev/>
- <https://www.reddit.com/r/codex/comments/1rci2qz/>
- <https://www.reddit.com/r/codex/comments/1tf4s07/>
- <https://www.reddit.com/r/codex/comments/1sx66wr/>
- <https://www.reddit.com/r/codex/comments/1t3ffxe/>
- <https://www.reddit.com/r/codex/comments/1rajwne/>
- <https://www.reddit.com/r/codex/comments/1t6iulo/>
- <https://www.reddit.com/r/codex/comments/1rdhtzt/>
- <https://www.reddit.com/r/codex/comments/1u9h4ja/>
- <https://www.reddit.com/r/codex/comments/1tnqsw8/>
- <https://www.reddit.com/r/codex/comments/1u979hw/>
- <https://community.openai.com/t/understanding-the-new-codex-limit-system-after-the-april-9-update/1378768>
- <https://community.openai.com/t/codex-usage-forecasting-transparency-for-pro-plus-plans-clarification-needed/1365130>
- <https://community.openai.com/t/gpt-5-5-seems-to-be-degraded/1381700>
- <https://github.com/openai/codex/issues>

横向评测

- <https://www.reddit.com/r/vibecoding/comments/1t10zmw/>
- <https://www.reddit.com/r/ClaudeCode/comments/1sk7e2k/>
- <https://www.reddit.com/r/cursor/comments/1u864tu/>
- https://www.reddit.com/r/AI_Agents/comments/1slczzz/
- <https://www.reddit.com/r/vibecoding/comments/1r6htpy/>
- <https://www.reddit.com/r/cursor/comments/1qz8rof/>
- <https://www.cosmicjs.com/blog/claude-code-vs-codex-vs-cursor>
- <https://composio.dev/content/claude-code-vs-openai-codex>
- <https://www.morphllm.com/codex-pricing>
- <https://www.developersdigest.tech/blog/ai-coding-tools-pricing-2026>

新闻报道

- <https://9to5mac.com/2026/05/29/chatgpt-for-ios-can-now-start-codex-work-on-windows/>
- <https://www.techrepublic.com/article/news-openai-codex-windows-app-launch/>
- <https://www.itpro.com/business/business-strategy/openai-codex-developer-agent-just-got-a-big-update>
- <https://openai.com/zh-Hans-CN/index/running-codex-safely/>
- <https://openai.com/zh-Hans-CN/index/building-codex-windows-sandbox/>