《Codex 蓝皮书：从入门到架构大师》

主理人: Hunk Wu (X: @ai_pmer)

🌐 English PDF Version | 🌐 English Online Version

🧭 目录 (Table of Contents)

🔌 关联开源项目

- **[飞书助理 (Codex Feishu Sentinel)]**：蓝皮书 Ch.08 官方参考工程。专为 Codex 开发者打造的飞书助理，支持日报自动汇总推送、CI 熔断移动端警报与手机端一键审批。

Ch.01 告别手写代码：Vibe Coding 时代的产品心智

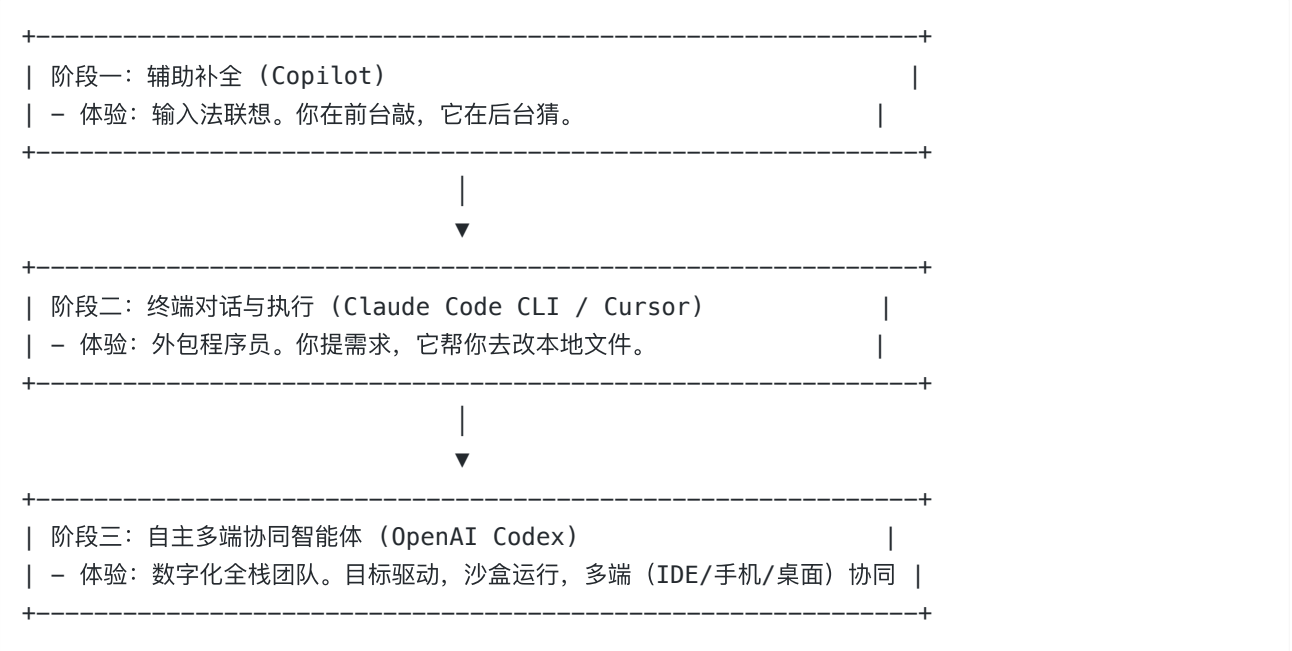
> 🎯 ***具体工程麻烦***：*开发者把 AI 当作高级自动补全，手动写样板代码累死累活，或者任由 AI 自由发挥导致架构完全失控。*
> 💡 ***可运行实战代码与落地收益***：*掌握 3 阶段演进心智模型；带走「边界与验收标准 PRD 控制模板」，彻底告别低效打字。*
> ⚡ ***社交传播 / 截图金句***：*“人类不应该再手写一行无意义的模板代码。你的手应该用来握紧方向盘，而不是推手推车。”*

在 “实战产品说” 与很多读者聊天时，我发现大家最容易陷入一个思维陷阱：拼命去记 AI 编程工具的快捷键和命令，像背 API 手册一样去学习 AI 编程。

醒醒吧！在最新的 AI 智能体时代（如 OpenAI Codex 时代），代码编写本身的门槛已经被拉低到几乎为零。这被称为 **“Vibe Coding”（氛围感编程）** 时代——你只需要关注产品的核心逻辑、商业闭环与用户体验，剩下的脏活累活全部交给智能体。

本章将帮你重构心智，理清从传统的“代码打字员”向“AI 编排架构师”跃升的底层逻辑。

1.1 编程浪潮的演进：我们在哪里？



很多人还在用第一阶段或第二阶段的工具，但其实 Codex 已经把我们推到了第三阶段：

- **多端联动**：本地用 CLI 跑编译，人在外面通过手机 ChatGPT 实时监控部署，甚至开启 Desktop Computer Use 让 AI 去操作浏览器进行 UI 测试。
- **强推理核心**：最新的 5.5 系列模型具备长链规划能力。智能体能够自主解决打包冲突、依赖报错等复杂的工程问题，超越了简单的代码补全。

1.2 核心心智转变：从“过程控制”到“边界控制”

做产品经理（PM）最懂的一点是：你给开发派需求，千万不要写“第一步点这里，第二步查那个表”，那是外行指导内行。你应该写清楚 **PRD（产品需求文档）** 的验收标准和边界约束。

用 Codex 也是一样的道理。以前你用 Prompt 教 AI 怎么写 for 循环（过程驱动）；现在你要当架构师，只负责三件事：

1. **定义终态 (Goal)**：我们要解决用户的什么问题？
2. **确立红线 (Constraints)**：哪些代码绝对不能碰？安全密钥怎么隔离？
3. **自动化验证 (Validation)**：写好测试脚本，让 Codex 运行它来向你证明代码写对了。

> *[!IMPORTANT]*
>
> ***主理人硬核公式***
>
> *成功发布 = 清晰的目标 \(Goal\) \+ 严苛的约束 \(Constraints\) \+ 自动化的测试 \(Validation\)*

1.3 你的新护城河：编排力与商业认知

如果手写代码不再是壁垒，那么程序员和产品经理的护城河是什么？

在 pmer.cn 上我分享过这个观点：**壁垒在于你对业务边界的定义，以及你对用户同理心的理解。**

你不需要再花三天去调一个 CSS 布局或配置 Webpack；你只需要知道，如何用 Codex 编排出稳定的架构，然后迅速把产品丢到市场上，验证用户是否愿意买单。

在接下来的章节中，我们将正式开启这套“跨端编排”的产品实战之旅。

Ch.02 跨端掌控：Codex 多端生产力矩阵搭建

> 🎯 ***具体工程麻烦***：*新手急着和 AI 对话写代码，本地 Node/Rust 依赖错乱、权限没开对，导致终端疯狂报错、白白烧掉几万 Token。*
> 💡 ***可运行实战代码与落地收益***：*提供无需安装 Rust 的 npm CLI 零踩坑安装脚本，macOS 辅助功能与屏幕录制权限核验命令，三端联调 checklist。*
> ⚡ ***社交传播 / 截图金句***：*“AI 编程第一步，先配稳你的指挥舱。别让本地依赖报错吃光你的宝贵 Token。”*

工欲善其事，必先利其器。在“实战产品说”中，我常强调一个原则：**AI Native 开发的第一步，是把你的“指挥舱”配置得足够稳定。** 很多新手急着去跟 AI 聊天写代码，结果因为本地环境不匹配、权限没开对，导致 AI 在终端疯狂报错、浪费 Token。

本章将带你一步步配置 Codex 的多端生产力矩阵，包括 CLI 客户端、Desktop App 以及 ChatGPT 移动端的联动桥接。

2.1 CLI 客户端（Rust 核心）安装与避坑

Codex CLI 的核心是用 Rust 编写的（codex-rs），运行效率极高；但 OpenAI 通过 npm 提供了官方分发包，**作为用户你不需要安装 Rust 工具链**，只需要 Node.js 18+（推荐 20+）。

1. 基础环境检查与安装

在终端运行：

```
# 检查 Node.js（要求 18+，推荐 20+）
node --version

# 方式 A：官方一键脚本（macOS / Linux）
curl -fsSL https://chatgpt.com/codex/install.sh | sh

# 方式 B：npm 全局安装
npm install -g @openai/codex

# 方式 C：Homebrew
brew install --cask codex
```

Windows 用户可以通过 PowerShell 安装：

```
powershell -ExecutionPolicy ByPass -c "irm https://chatgpt.com/codex/install.ps1 | iex"
```

安装完成后，验证：

```
codex --version
```

2. 认证方式选择：ChatGPT 账号 vs API Key

Codex CLI 提供两种认证方式，**官方默认推荐 ChatGPT 账号登录**：

- **方式 A（推荐 / 默认）**：直接在终端运行 `codex`，浏览器会自动弹出 ChatGPT 登录页面。**ChatGPT Plus（$20/月）、Pro、Team、Business、Edu、Enterprise 套餐都已经包含 Codex 用量额度**，对独立开发者是性价比最高的选择。
- **方式 B（按量计费 / CI 场景）**：使用 OpenAI API Key。适合 CI/CD 自动化场景，或订阅额度跑满后兜底。

```
# 在你的 ~/.zshrc 或 ~/.bashrc 中写入
export OPENAI_API_KEY="sk-proj-xxxxxx..."
```

3. 账单熔断防爆配置

AI 暴走可能会在几小时内刷爆你的卡。**Codex CLI 本身没有“每任务预算”或“每分钟请求数”这类环境变量**，真正的防爆控制路径有三条：

1. **OpenAI 后台硬上限（最重要）**：在 OpenAI 平台后台为该 API Key 设置月度 Usage Limit 硬限制（建议初学者设为 $50）。即使 AI 发生无限循环，也能保证你的资金绝对安全。
2. **`config\.toml` ** 中的 Token 控制**：在 `\~/\.codex/config\.toml` 写入：

```
# 每次工具输出的 token 上限，防止误读超大文件
tool_output_token_limit = 12000

# 主动压缩历史的阈值，避免上下文无限膨胀
model_auto_compact_token_limit = 64000
```

3. **沙盒与审批策略**：通过 `sandbox\_mode`、`approval\_policy` 限制 Codex 的危险动作（详见 Ch.05）。

> 💡 ***主理人避坑小贴士**：如果你用的是 API Key 模式，**务必去 OpenAI 后台把月度额度（Usage Limit）写死**——这是最后一道保险，比任何环境变量都靠谱。*

2.2 桌面端（Desktop App）与 Computer Use 权限配置

> ***前置说明**：Codex 的 Computer Use（让 AI 操作你的桌面应用）**目前仅支持 macOS**，欧洲经济区（EEA）、英国和瑞士因合规暂未开放。Windows / Linux 用户暂不可用。*

为了让 Codex 能够自主操作你的屏幕进行测试，你需要安装 Codex Desktop，并授予其操作系统底层权限。

1. 下载与插件安装

1. 从 openai.com/codex 下载并安装 Codex Desktop App(macOS)。
2. 启动 App，在 **「Settings → Computer Use」** 中点击 **Install** 安装 Computer Use **插件**（这是个单独的 plugin，默认未启用）。

2. 系统权限授权

在 macOS **「系统设置 → 隐私与安全性」** 中，授予 Codex 两项权限：

- **辅助功能 (Accessibility)**：允许 Codex 模拟鼠标点击和键盘输入。
- **屏幕录制 (Screen Recording)**：允许 Codex 截取屏幕图像输入给视觉模型进行识别。

```
[macOS 系统设置] -> [隐私与安全性]
  ├─ 辅助功能 (Accessibility)  ──> 勾选 [Codex] (启用模拟鼠标/键盘)
  └─ 屏幕录制 (Screen Recording) ──> 勾选 [Codex] (允许 Vision 分析)
```

---

3. 能力边界（重要）

Computer Use 的安全边界由 macOS 系统层 + Codex App 内的 GUI 白名单共同决定：

- 你需要在 Codex App 设置中**显式批准每一个 Computer Use 可以操作的应用**（首次使用某 App 时会弹窗询问，可勾选 “Always allow”）。
- 出于安全原因，Codex **无法用 Computer Use 自动操作终端、Codex 自身、或系统级管理员授权弹窗**。

2.3 手机端（ChatGPT App）监控桥接

2.3 手机端（ChatGPT App）监控桥接

人在户外时，希望本地或云端的 Codex 任务能把状态推送到手机。**注意：Codex 本身没有内置 “任务失败时自动推送到微信/飞书” 的字段**——本节展示的是用 **GitHub Actions + Webhook 网关 + 飞书/企微机器人** 自行拼装的方案。

1. 整体思路

```
[Codex 任务 (本地/云端)] → [GitHub Actions 或自定义脚本] → [Webhook 网关] → [飞书/企微/Telegram 机器人]
```

```
                                                    ↓
                                              [手机端收到推送]
```

具体的 GitHub Actions 配置与 Webhook 中转脚本见 Ch.08。

2. 手机端 ChatGPT App 中查看 Codex 任务

如果你用 **ChatGPT 账号登录** Codex，你的云端 Codex 任务会自动同步到手机 ChatGPT App，可以随时查看进度、发送追加指令。这是 OpenAI 官方提供的能力，不需要额外配置。

3. 双向交互（自定义实现）

当你在咖啡馆喝咖啡时，本地的 Codex 运行到了部署步骤，通过 Ch.08 展示的网关脚本，飞书或企微机器人会收到卡片推送。你只需要在群中回复 `1`，中转网关解析后向本地 Codex 进程发送批准信号（通过文件信号、HTTP 回调或 Codex 的 hooks 机制），继续执行发布。

Ch.03 破局云端孤岛：沙盒调试与本地环境深度穿透

> 🎯 ***具体工程麻烦***：*AI 在隔离沙盒跑测试时连不上本地 Docker 的 PostgreSQL/Redis，报错 `Connection refused`，智能体抓瞎。*
> 💡 ***可运行实战代码与落地收益***：*提供 SSH / Ngrok 端口反向映射脚本、`host.docker.internal` 端口配置、3 步网络连通性排查流程。*
> ⚡ ***社交传播 / 截图金句***：*“AI 报错连不上 localhost？不是代码写错了，是你没给隔离沙盒开网络通道。”*

使用 Codex 运行数据库测试时，常见问题之一是智能体报错 `Connection refused to ``localhost:5432`，即便本地已通过 Docker 启动了 PostgreSQL 服务。

这就是 **沙盒隔离（Sandbox Isolation）** 带来的典型壁垒。为了系统的安全和环境的纯净，Codex 默认是在云端独立的虚拟化容器中执行代码的。这意味着，AI 眼中的 `localhost` 是它自己的虚拟隔离环境，而不是你的 Mac 电脑本身。

本章我们聊聊怎么破局，打通云端沙盒与你本地宿主机的网络通道。

3.1 沙盒网络壁垒：为什么 `localhost` 连不上?

如下图所示，默认状态下，沙盒与本地宿主机（Your Mac）是网络阻断的：

```
+-----------------------------------+                  +-----------------------------------+
|     云端沙盒 (Sandbox)      |                  |    宿主机 (Local Mac)     |
| - App Code                  |    (隔离屏障)    | - Docker container        |
| - localhost:5432 (空无一人) ----[ X ]-----------> | - PostgreSQL (port: 5432) |
+-----------------------------------+                  +-----------------------------------+
```

如果我们要让沙盒里的代码能正常读写本地的数据库，或者调通本地跑着的微服务，我们必须在“网络防火墙”上钻开一个孔，也就是进行**端口映射与反向隧道穿透 (Reverse Tunneling)**。

3.2 实战：使用 SSH / Ngrok 建立反向隧道

我们以最常见的场景为例：**让云端沙盒里的 Codex 连接本地宿主机 Docker 里的 PostgreSQL 数据库。**

方法一：使用 SSH 反向端口转发 (推荐)

如果你拥有一个公网中转服务器（VPS），你可以利用 SSH 自带的端口转发机制，将本地的数据库端口安全地挂载到云端。

在本地终端运行：

```
# 将本地的 5432 (Postgres) 转发到公网中转机的 54320 端口
ssh -R 54320:localhost:5432 user@your-public-vps.com -N
```

接着，配置 Codex 沙盒中的环境变量，让其直接访问公网服务器的映射端口：

```
export DATABASE_URL="postgresql://postgres:password@your-public-vps.com:54320/dev_db"
```

方法二：使用 Ngrok 穿透本地端口 (零配置极速方案)

如果你没有公网服务器，`ngrok` 是最快的方式。

在宿主机运行：

```
# 启动 TCP 隧道映射本地 PostgreSQL 端口
ngrok tcp 5432
```

终端会输出如下穿透地址：

`Forwarding tcp://0\.tcp\.ngrok\.io:12345 \-\> localhost:5432`

将该动态地址配置到你的 AGENTS.md 或是 Codex Cloud 的环境变量中：

```
export DATABASE_URL="postgresql://postgres:password@0.tcp.ngrok.io:12345/dev_db"
```

> 💡 ***小提示***：*ngrok 免费版每次重启地址会变；如果做长期开发，建议要么用付费的固定 TCP 端口，要么换 Cloudflare Tunnel / frp 等替代方案。*

3.3 目录映射与环境变量同步

除了网络，文件和密钥也需要顺畅流转。

1. 密钥文件的安全同步规则

千万不要让 Codex 自动把 `\.env` 文件同步到公共云端。我们必须在本地的 `\.gitignore` 中加入 `\.env`，并在项目的 AGENTS.md 中添加硬约束：

```
## 🛑 Hard Constraints
- Never sync or commit files matching *.env.
- Use `src/config.ts` as the unified wrapper to fetch environment configurations.
```

2. 沙盒临时缓存清理

为了避免沙盒缓存导致的"灵异 Bug"（比如旧的依赖包未清除），我们可以让 Codex 在每次启动测试前自动运行清理命令。在 AGENTS.md 的开发指令中写入：

```
## 💻 Developer Commands
- **Clean Run**: `rm -rf node_modules/.cache && npm run test`
```

通过这一系列的配置，云端沙盒不再是与世隔绝的孤岛。它就像是你本地电脑的外延，能够流畅、安全地读取各种本地数据与服务，让 Vibe Coding 的丝滑度再上一个台阶。

Ch.04 目标驱动：用“边界与断言”驾驭推理型智能体

> 🎯 ***具体工程麻烦***：*写太长 Prompt 像教大厨切土豆丝导致模型受限；一句话太模糊又导致 AI 胡编乱造改出 10 个 Bug。*
> 💡 ***可运行实战代码与落地收益***：*目标驱动 Specs 标准模板（目标 + 前提 + 验收断言 + 禁区红线）；传统 Prompt 与 Specs 真实测试对比。*
> ⚡ ***社交传播 / 截图金句***：*“教大厨切土豆丝只会把菜炒糊。给 AI 下指令，定死输入输出与验收断言才是专业做法。”*

在以 GPT-5.5 等强推理模型为核心的 Codex 时代，传统的过程式 Prompt 工程已不再适用。推理模型具备内生规划（Internal Planning）空间，过细的执行步骤会限制其规划效率。

本章将分享如何在实际开发中，用“产品 Specs”的方式去驱使 Codex。

4.1 核心逻辑：别教米其林大厨怎么切菜

面对具备强推理能力的智能体，过细的指令往往适得其反：

❌ *“请你拿起菜刀，把土豆切成 2mm 的细丝，然后把锅烧热，倒入 15g 花生油，下锅翻炒 3 分钟，最后放 3g 盐。”*

这叫**“过程驱动”**。不仅累，而且很容易因为火候不同而把菜烧焦。

更高效的协作模式是 **目标驱动**：

✅ *“我需要一盘香脆可口的土豆料理作为牛排的配菜。要求：热量控制在 200 卡内，不能使用黄油，并且必须在 15 分钟内出锅。”*

通过给出 **目标 (Goal)**、**红线 (Constraints)** 和 **验证标准 (Validation)**，将具体的执行细节交给智能体推理并实现。

4.2 目标驱动的 Markdown 结构规范

当你在本地或云端向 Codex 发起编码任务时，不要堆砌毫无章法的对话，使用以下标准 Markdown 格式：

```
# 🎯 Goal
[描述你希望达到的最终状态。例如：实现一个支持 Github 登录并能存储用户偏好设置的路由。]

# 🛑 Constraints (红线约束)
- [安全红线，如：绝对不能将密钥明文写入代码。]
- [技术选型限制，如：必须使用原生的 CSS Grid，不能使用 Tailwind。]
- [代码防污染，如：禁止修改 /src/legacy 目录下的任何文件。]

# 🧪 Validation Specs (验证标准)
- [自动化测试，如：运行 npm run test:unit 必须 100% 通过。]
- [边界行为，如：当输入为空时，接口必须返回 400 Bad Request 且带有 JSON 错误提示。]
```

4.3 真实案例对比：传统 Prompt vs 目标驱动 Specs

假设我们要写一个 **“带 Redis 限流的 API 代理服务”**。

❌ 传统过程驱动型 Prompt

> *“请帮我用 Express 写一个 API 代理。首先引入 express 和 express-rate-limit。然后配置 rate-limit，设置 windowMs 为 15 分钟，max 为 100 次。接着写一个路由 `/api/proxy`，使用 axios 请求第三方 API `https://api\.github\.com`。如果请求成功就返回数据，如果失败就返回 500 错误。注意在请求头里带上 Authorization Bearer Token。”*

✅ 目标驱动 Specs

```
# 🎯 Goal
实现一个 Express API 代理路由，代理所有发往 GitHub API 的请求。

# 🛑 Constraints
- 必须使用 Redis 作为限流器的数据源，禁止使用内存型限流，以支持多实例部署。
- 代理请求的超时时间（Timeout）必须硬性限制在 3000ms 以内，防止挂起主线程。
- 严禁将 GitHub Token 写入代码或日志中，必须从 `process.env.GH_TOKEN` 安全读取。

# 🧪 Validation Specs
- 在 1 分钟内发送超过 60 次请求时，代理必须返回 429 Too Many Requests。
- 当代理请求发生超时或网络错误时，必须返回 504 Gateway Timeout，并带有结构化 JSON 响应。
```

4.4 Codex 对 Specs 的推理过程分析

当你把这套规范扔给 Codex 后，它的内部思维链（Chain of Thought）会这样运转：

```
graph TD
    A[解析 Specs 目标] --> B{分析 Constraints}
    B -->|硬约束: Redis 限流| C[决定引入 ioredis 和 rate-limit-redis]
    B -->|硬约束: 3s 超时| D[配置 Axios/Fetch 的 timeout 参数]
    B -->|安全约束: 密钥隔离| E[引入 dotenv 并编写类型声明]
    C & D & E --> F[规划代码结构并编写实现]
    F --> G{比对 Validation Specs}
    G -->|模拟 429 场景| H[编写 Redis 模拟测试用例]
    G -->|模拟 3s 超时| I[编写延迟 API Mock 并运行单元测试]
    H & I --> J[输出最终代码与测试报告]
```

你会发现，Codex 甚至会自动处理 Redis 连接重试、超时的捕获等健壮性逻辑——而这些在以前，是需要你写上百字去千叮咛万嘱咐的。

把逻辑规划权让渡给 AI，把验收标准牢牢攥在自己手里。 这就是 AI 时代最高效的人机协同法则。

Ch.05 制定 CAP 协议：构建项目专属的 AGENTS.md 规则层

> 🎯 ***具体工程麻烦**：AI 修好 1 个 Bug 带出 3 个新 Bug；在同一报错上连续重试 10 次烧钱自旋；擅自安装杂乱外部包或手写假 TODO。*

> 💡 ***可运行实战代码与落地收益***：*生产级 `AGENTS.md` 规则层模板；失败 2 次强制熔断机制；禁止假代码占位与依赖失控红线。*
> ⚡ ***社交传播 / 截图金句***：*“没有规则约束的 AI 就是脱缰野马。一份 AGENTS.md，让智能体像十年老架构师一样遵守工程规范。”*

在实际做产品的过程中，最怕遇到的一种开发情况是：修好了一个 Bug，却顺手带出了三个新 Bug；或者新写了一个功能，结果把团队约定的代码风格破坏得一塌糊涂。

在人机协同开发中，Codex 如果没有边界约束，也会变成一个“破坏性极强”的勤奋员工。为了给它戴上缰绳，我们需要在项目根目录下建立 ** `AGENTS\.md` **。

这就是我们的 **“智能体协作宪法” (Codex Collaboration Protocol, CAP)**。

5.1 为什么我们需要 `AGENTS\.md`？

`AGENTS\\\.md` 是 OpenAI Codex 官方约定的指令文件名**（不是用户自取的）。Codex 在启动时会从当前工作目录开始，**逐级向上扫描**并合并以下文件，作为系统级上下文的一部分：

1. `\~/\.codex/AGENTS\\\.override\\\.md`（最高优先级，个人覆盖）
2. `\~/\.codex/AGENTS\\\.md`（全局指令）
3. 项目根目录的 `AGENTS\\\.md`（团队约定）
4. 当前工作目录的 `AGENTS\\\.md`（最具体的局部约定）

它的核心价值在于：

1. **启动即恢复记忆**：每次 Codex 启动时，第一件事就是扫描 `AGENTS\\\.md`，瞬间拾起整个项目的技术栈、文件结构和协作约束，不需要你在对话里反复唠叨。
2. **代码防腐层**：明确定义哪些文件是“只读/禁止触碰”的，引导 Codex 不去擅自重构核心安全模块（如登录鉴权、数据库 Schema）。
3. **约束命令执行**：规定只能使用特定的测试和部署命令，降低 AI 误跑破坏性脚本的概率。

5.2 `AGENTS\.md` 的核心四大版块

```
# 项目指纹 (Project Fingerprint)
- 告诉 AI 这是一个什么样的项目，核心技术栈是什么。

# 开发常用命令 (Commands)
- 明确指出编译、测试、迁移数据库的命令，不要让 AI 瞎猜。

# 架构与编码规范 (Styles & Patterns)
- 规定文件存放目录、大小限制及必用的设计模式。

# 智能体安全红线 (Hard Rules)
- 绝对的禁区。一旦触碰，Codex 必须立刻中止并请求人工确认。
```

5.3 实战：一个生产级的 `AGENTS\.md` 模板

以下是一个典型的全栈 SaaS 项目的 `AGENTS\.md` 规范：

```
# 🤖 Project: Aurora SaaS Core (AGENTS.md)

## 🧬 Project Fingerprint
- **Stack**: Next.js 15 (App Router), TypeScript, Prisma ORM, TailwindCSS.
- **Database**: PostgreSQL on Supabase.
- **Auth**: Next-Auth v5.

## 💻 Developer Commands
- **Install**: `npm install` (Only run if package.json has changed)
- **Dev Server**: `npm run dev`
- **Lint Code**: `npm run lint`
- **Run Tests**: `npm run test`
- **DB Migration**: `npx prisma migrate dev` (Never run in production branch)

## 🎨 Styles & Architecture Patterns
- **Directory Structure**:
  - Components: Keep UI components inside `@/components/ui/` (Shadcn styled).
  - Business Logic: Custom React hooks must go to `@/hooks/`.
  - API Routes: Next.js Route Handlers go to `src/app/api/.../route.ts`.
- **Formatting**:
  - Keep components modular. If a component exceeds 200 lines, decompose it.
  - All API routes must implement Zod schema validation for request body.
  - Return HTTP 400 for validation errors, 500 for internal uncaught errors.

## 🛑 Agent Boundary & Hard Rules
- **READ-ONLY Directories**:
  - Never modify files inside `src/app/api/auth/[...nextauth]` (OAuth Core).
  - Never alter `prisma/schema.prisma` without explicit human confirmation.
- **PR Rules**:
  - Before declaring a feature complete, run `npm run test` and `npm run lint`.
  - If tests fail, rollback the change immediately and report the error logs.
- **Security Check**:
  - Never commit raw `.env` files or API Keys. Use environment variables.
```

5.4 软约束与硬约束的边界

需要特别澄清的一点是：** `AGENTS\\\.md` **** 里写的规则本质上是“软约束”**——它是注入到模型上下文里的自然语言指令，模型（GPT-5.5 / GPT-5.3-Codex）会**尽量遵守，但没有运行时硬性 enforcement**。

如果你要做“宁可错杀也不放过”的硬性拦截，必须搭配 Codex 的三层运行时控制：

1. **沙盒模式 (Sandbox Mode)**：在 `\~/\.codex/config\.toml` 中设置 `sandbox\_mode = \&\#34;read\-only\&\#34;` 或 `\&\#34;workspace\-write\&\#34;`，从 OS 层面限制文件写入范围。
2. **审批策略 (Approval Policy)**：通过 `approval\_policy` 让 Codex 在执行危险命令前必须人工确认。

3. **Hooks 机制**：在 `config\.toml` 中配置 `\[\[hooks\]\]`，可以在 Codex 调用某个工具（如文件写入）前自动触发拦截脚本。

一个典型的硬约束配置示例：

```
# ~/.codex/config.toml
sandbox_mode = "workspace-write"
approval_policy = "on-request"

# 项目内默认沙盒收紧到只能读写当前 worktree
project_root_markers = [".git"]
```

软约束 + 硬约束的组合，才能让你放心地把大段业务开发逻辑委托给 Codex，而不用时刻盯着它是否改坏了底层核心模块。

Ch.06 思维纠偏：如何像技术总监一样透视推理过程

> 🎯 ***具体工程麻烦***：*面对强推理模型只能干等；当 AI 第一步假设走偏时，它会顺着错误连续深陷，搞乱代码库。*
> 💡 ***可运行实战代码与落地收益***：*终端 TUI 实时透视 CoT 思考步骤方法；3 种死循环特征识别表；配套 Chrome 扩展实战工程（`examples/ch06-chrome-extension`）。*
> ⚡ ***社交传播 / 截图金句***：*“别让 AI 蒙头狂奔 20 分钟才告诉你走偏了。看懂思考日志，在它跑偏的第一步一键拽回。”*

在传统开发中，管理初级程序员时，最担心的场景是其默默闭门造车，最终交付一堆偏离业务逻辑的代码，甚至导致系统崩溃。

在使用强推理模型（GPT-5.5）驱动的 Codex 时，这种情况同样存在。虽然 AI 拥有强大的推理能力，但一旦它的前置假设出错，它就会顺着错误的逻辑一路狂奔，甚至陷入自我纠错的“无限循环”。

本章教你如何穿透 Codex 的**推理过程**，在它偏离航线时，像一个资深技术总监一样精准介入。

6.1 为什么要看模型的推理过程?

强推理模型与普通大模型的最大区别在于：它在输出最终代码前，会先在内部“打草稿”进行推理和自我模拟。Codex TUI 会在交互界面上**以推理摘要（Reasoning Summary）的形式**展示这个过程。

```
[用户需求] ──> 1. 解析目标与限制 ──> 2. 规划步骤 ──> 3. 运行测试 ──> 4. 自我修正 ──> [最终输出]
                  └──(在 TUI 中显示为 Reasoning Summary，即你的“监考视窗”)──┘
```

如果你只看最终结果，你只能“被动接受”。如果你学会监控推理摘要，你就能在步骤 2 或 3 发现它的逻辑漏洞，提前干预，避免浪费你的 API Token 和时间。

> ⚠️ *注意：完整的“思维链 (Chain of Thought)”原始内容是 OpenAI 内部的，不会对用户暴露。TUI 中看到的是模型自己生成的**摘要**——已经足够用来判断它的方向是否走偏。*

6.2 如何在 TUI 中观察并解读推理过程

两种典型用法：

1. 交互模式（TUI）

直接运行 `codex`，进入 TUI 后，Codex 会自动在主面板的“Reasoning”分栏里实时滚动它的推理摘要。

如果你要查看更详细的 turn 信息，可以使用斜杠命令：

```
# TUI 中输入以下斜杠命令
/diff       # 查看当前 turn 的代码变更
/review     # 让另一个 Codex 子 agent 审查最近的变更
/copy       # 复制最后一次响应到剪贴板
```

2. 非交互模式（脚本化分析）

如果想做日志化分析，运行：

```
# 非交互模式，输出 JSONL 流，可管道给日志处理脚本
codex exec --json "<your task>" > task.jsonl
```

JSONL 中会包含 `reasoning` 事件、工具调用、模型回复等结构化数据，方便你做监控告警或回放。

实时推理摘要示例

当 Codex 收到“修复 Redis 限流器连接超时”的任务时，Reasoning 面板会输出类似下方的思考流：

```
[Reasoning Summary - 示意]
- User wants to fix Redis rate limiter connection timeout.
- Checking existing implementation in src/lib/redis.ts...
- I see `redis = new Redis()` without retryStrategy.
- If Redis is down, it hangs the Express app. This violates the 3000ms timeout
constraint in AGENTS.md.
- Plan:
  1. Add `maxRetriesPerRequest: null` and `enableReadyCheck: false`.
  2. Implement custom retryStrategy up to 3 times.
  3. Write mock connection test to verify.
```

6.3 识别 AI 陷入的典型“死循环”

在AI 项目实操经历中，我总结了 AI 容易陷入的三个死循环，一旦看到推理摘要中出现以下特征，必须立刻介入：

1. 无限 npm install 循环 (The Dependency Loop)

- **现象**：AI 试图使用某个新库，运行安装报错；它在 CoT 里决定更换版本再次安装，又报错；接着它试图安装另一个同类库……
- **CoT 特征**：`Error: Cannot resolve dependency \.\.\. Running npm install \-\-legacy\-peer\-deps \.\.\.` 重复出现 3 次以上。

2. 代码重构自毁循环 (The Regression Loop)

- **现象**：AI 修改了 A 文件导致单元测试 B 失败；它去修改 B 测试，结果导致 C 模块报错；它又去改 C，结果 A 又坏了。
- **CoT 特征**：不断在两三个文件之间往返修改，并且测试通过率反复在 80% 和 90% 之间横跳。

6.4 介入三部曲：打断、修正与接管

当发现 AI 走偏或陷入死循环时，不要坐以待毙。请按照以下步骤进行干预：

第一步：果断打断（`Ctrl \\\+ C`）

在终端直接按下 `Ctrl \\\+ C`。这会立刻中止 Codex 的当前任务，阻止它继续消耗 Token。

第二步：点对点纠偏（直接对话）

打断后，Codex 会回到 TUI 等待新指令。**Codex CLI 并没有专门的 `refine` 子命令**——你只需要直接输入下一条指令，明确指出它思考过程中的逻辑盲区：

```
# 在 TUI 中继续输入：
你刚才试图安装 axios-retry，但本项目禁止安装任何第三方 HTTP 重试库。请使用原生的 AbortController 来实现超时重试，并把这条规则追加到 AGENTS.md。
```

如果想接着上一次会话继续，也可以用：

```
codex resume      # 在新终端中恢复之前的会话
```

第三步：人手接管与回滚

如果 AI 已经把代码改得面目全非，不要试图让它自己改回去。直接运行 Git 命令回滚，并在 `AGENTS\.md` 中追加一条硬性红线：

```
# 撤销 AI 刚才的错误修改
git checkout -- src/lib/redis.ts
```

然后在 AGENTS.md 中追加：

```
- 禁止为任何简单的网络超时问题引入外部重试依赖库。
```

6.5 章节实操配套：Codex Web Copilot (Chrome 扩展沙盒样例)

为了让读者直观体验如何使用 Codex 思考链引导与 Anti-Loop 护栏进行浏览器插件开发，本项目配套提供了开箱即用的轻量开源样例：

👉 **源码沙盒目录**：examples/ch06-chrome-extension

核心亮点：

1. **纯原生 Manifest V3**：零打包依赖，直接在 Chrome 浏览器中「加载已解压的扩展程序」即可 1 分钟开箱体验；
2. **严格 CSP 护栏**：在 `AGENTS.md` 中严禁内联脚本与 `eval()`，展示 AI 智能体如何在最严苛的浏览器安全沙盒下编写高可用代码；
3. **自动化测试守卫**：执行 `npm test` 自动验证 MV3 规范与脚本语法。

Ch.07 视觉闭环：Desktop Computer Use 自动巡检与设计还原

> 🎯 ***具体工程麻烦***：*传统前端 UI 还原依赖肉眼查像素，响应式漏看弹窗遮挡；纯命令行测试无法覆盖真实浏览器渲染与点击。*
> 💡 ***可运行实战代码与落地收益***：*安全限制操作框配置；Figma 设计稿与本地网页自动截图比对走查流程；UI 视觉巡检真实录屏复现。*
> ⚡ ***社交传播 / 截图金句***：*“写完前端还在肉眼查像素？让 AI 自己打开浏览器量尺寸、点按钮，截图标注哪里不合规。”*

在传统的 UI 还原度走查中，最耗费产品经理和前端时间的是“像素眼”校对：

“这个按钮好像往左偏了 4 像素。”

“这个弹窗在 iPad 尺寸下会变形遮挡。”

在 OpenAI Codex 生态中，通过 **Computer Use (计算机操作能力)**，智能体不仅能编写代码，还能“动用眼和手”直接操作你的 macOS桌面，打开浏览器、操作开发者工具，进行视觉效果校对。

本章教你如何配置并操纵 Codex Desktop 进行 UI 的自动化视觉还原。

7.1 安全第一：沙盒边界与屏幕操作框限制

让 AI 操作你的屏幕是一件具有安全风险的事情。为了防止 Codex 因为误判乱点你的本地微信或删除系统文件，必须设置**应用层面的访问白名单。**

1. 通过 GUI 设定应用边界（重要）

Codex 的 Computer Use 边界**不是通过 JSON 配置文件**控制的，而是通过 Codex App 内的 GUI：

- 首次使用 Computer Use 操作某个 App（如 Google Chrome）时，Codex 会弹窗请求授权；
- 在弹窗中可以选择 **“Just this once”**（仅本次） 或 **“Always allow”**（每次都允许）；
- 已授权列表可以在 **「Codex Settings → Computer Use → Allowed Apps」** 中随时撤销。

> 💡 ***建议***：*UI 走查任务中，把白名单收紧到只勾选 **Google Chrome**、**iOS Simulator** 等开发相关 App，避免 Codex 跑去操作你的微信、邮件等隐私应用。*

2. 坐标定位机制解析

Codex 主要是通过“截屏 -> OCR/视觉分析 -> 返回目标 x, y 坐标 -> 执行点击”的闭环在操作你的电脑。

```
[屏幕截图 (Screenshot)] ──> [视觉模型识别] ──> [获取元素像素坐标 (x:450, y:230)] ──> [点击/拖拽]
```

由于截屏和点击是**真实的系统级操作**，所以 Codex 无法操作终端、Codex 自身或系统级管理员授权弹窗——这是出厂硬限制。

7.2 视觉驱动的 UI 走查实战：Figma 还原对比

这是一个非常典型且实用的工作流：**让 Codex 自主对比 Figma 设计图截图与浏览器渲染出的页面，并自动修改 CSS 进行还原。**

🎯 目标 (Goal)

让 Codex 对比本地网页 `/auth/login` 与设计师给的 `figma\_login\_mockup\.png`，自动调平页面中登录卡片的边距和字体大小。

🛑 约束 (Constraints)

- 只能通过修改 `src/app/login/page\.tsx` 的 Tailwind Class 来修正样式。
- 禁止修改 DOM 结构。

🧪 验证与自动化执行 Specs

把以下 Specs 直接粘贴到 Codex TUI 中作为 prompt，并在 prompt 里 `@Computer` 或 `@Chrome` 唤起 Computer Use：

```
@Chrome 请按以下步骤完成 UI 还原走查：

# 🎯 Goal
Compare and align browser rendering with figma_login_mockup.png.

# 🛑 Constraints
- Only use Tailwind utility classes in src/app/login/page.tsx.
- Do not change the DOM structure.

# 🚀 Execution Steps
1. Open Google Chrome and navigate to http://localhost:3000/auth/login.
2. Take a screenshot of the login card region.
3. Compare against /assets/figma_login_mockup.png and report layout differences.
4. Identify spacing discrepancy (Figma shows 32px padding-top, current implementation
has 16px).
5. Edit Tailwind classes in src/app/login/page.tsx, reload and verify.
```

> 💡 ***进阶用法***：*Codex 最近新增的* ***Appshots*** *功能（双击 Command 键），可以一键把前台窗口的截图 + 文本上下文发送给 Codex 线程，省去手动截图粘贴的步骤。*

7.3 Codex 自动走查的概念流程（示意）

当你执行上述任务时，Codex 的 Computer Use 模块会在 TUI 中展示类似下方的执行流程（**以下为概念示意，非真实日志格式**，实际你看到的是 Codex 的推理摘要 + 工具调用记录）：

```
> Running visual review for /auth/login
> Step 1: Opening Google Chrome on http://localhost:3000/auth/login...
> Step 2: Taking screenshot. Saved to /tmp/screenshot_v1.png
> Step 3: Calling vision model for image comparison.
    Analysis: "Login card header text 'Welcome Back' font size is too small (approx
16px),
               should be 24px (text-2xl) based on figma mockup. Card padding-top is
insufficient."
> Step 4: Modifying src/app/login/page.tsx:
    Target: Replace `className="text-base pt-4"` with `className="text-2xl pt-8"`
> Step 5: Refreshing Chrome...
> Step 6: Taking validation screenshot. Saved to /tmp/screenshot_v2.png
> Step 7: Vision check: "Layout matches mockup. Visual diff is within 1.5% tolerance."
> Task completed successfully.
```

7.4 最佳实践：响应式多终端巡检

除了单一尺寸的对比，你还可以让 Codex 快速切分屏幕尺寸进行“断点走查”：

```
@Chrome
# 📱 Mobile Viewport Inspection
1. Open DevTools in Chrome.
2. Simulate mobile viewport (iPhone 15 Pro: 393 x 852).
3. Verify that the login card does not overflow horizontally.
4. If the submit button falls below the screen fold, adjust padding-bottom to keep it
visible.
```

通过这一闭环，独立开发者再也不用在修改 CSS 后，手动缩放浏览器、拿手机真机反复刷新了。Codex 能够自主完成 90% 的页面微调和跨端兼容测试，释放你的全部视觉精力。

Ch.08 移动看护工作流：全天候离线编排实战

> 🎯 ***具体工程麻烦***：*开发者被困在工位看滚动编译日志；离开工位后 CI 挂了或高危发布卡住，整个团队进度中断。*
> 💡 ***可运行实战代码与落地收益***：*GitHub Actions 失败推送工作流；飞书 Webhook 告警卡片 JSON；手机远程一键审批部署交互流。*
> ⚡ ***社交传播 / 截图金句***：*“下班不盯屏幕，任务照常推进。CI 报错手机秒收飞书卡片，人在地铁上一键完成上线审批。”*

独立开发者与产品经理的核心追求除了高效率，还有工作的自由度。盯在电脑前查看智能体编译滚动日志的方式并不高效。

本章我们将搭建一套 **移动看护工作流**：当本地或云端服务器上的 Codex 进行自动化重构时，一旦遭遇编译失败或触发高危部署授权，微信、飞书或 Telegram 即可实时接收通知卡片，支持直接通过手机进行远程审批或干预。

> ⚠️ ***重要说明***：*Codex 本身**没有内置“任务失败时自动推送到飞书/微信”的字段**。本章展示的是利用 **GitHub Actions + Webhook 网关 + 机器人** 自行拼装的方案——所有官方零件都是真实可用的，但中转网关需要你自己部署。*

> *顺带一提：如果你用 ChatGPT 账号登录 Codex，你的云端 Codex 任务会**自动同步到手机 ChatGPT App**，这是 OpenAI 官方提供的能力，可作为本套自定义方案的简化平替。*

8.1 移动看护链条的整体架构

我们通过以下管道将本地/云端的 Codex Agent 接入你的手机：

```
[云端 Codex Agent] ──(Webhook)──> [中转网关 (你自部署的 Node 服务)] ──> [手机端微信/飞书]
       ▲                                                                        │
       └──────────────────(手机打字回复 "Approve / Stop") ──────────────────────┘
```

8.2 实战：GitHub Actions 失败推送与 Webhook 配置

当 Codex 在沙盒中运行构建或测试时，我们将构建日志通过 GitHub Actions 抓取，并触发通知。

1. GitHub Actions 自动化流水线配置 (`\.github/workflows/codex\-watchdog\.yml`)

在项目根目录下，我们编写如下工作流配置文件，当构建失败时，直接向手机推送包含 CoT 思考链关键节点的摘要：

```yaml
name: Codex Agent Watchdog

on:
  push:
    branches: [ main ]
  workflow_dispatch:

jobs:
  agent-build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: 20

      - name: Run Codex Validation (exec mode)
        id: run_agent
        env:
          OPENAI_API_KEY: ${{ secrets.OPENAI_API_KEY }}
        run: |
          # 使用 Codex exec 非交互模式跑验证任务
          npm ci
          npm run test || echo "STATUS=failed" >> $GITHUB_ENV

      - name: Push Fail Notice to Mobile
        if: env.STATUS == 'failed'
        run: |
          curl -X POST -H "Content-Type: application/json" \
            -d '{
              "event": "build_failed",
              "repo": "${{ github.repository }}",
              "commit": "${{ github.sha }}",
              "message": "Codex sandbox testing failed. Attention required on phone."
            }' \
            ${{ secrets.MOBILE_WEBHOOK_URL }}
```

> 💡 *在 CI 场景下使用 Codex 时，认证方式推荐用 API Key（通过 GitHub Secrets 注入 `OPENAI\_\_\_API\_\_\_KEY`）。ChatGPT 账号 OAuth 不适合无人值守的 CI 流程。*

8.3 户外移动端双向交互与审批

接收到失败通知只是第一步。更高级的玩法是，直接在手机端对 Codex 进行远程指令干预。

1. 场景：生产环境部署审批

当 Codex 跑通了所有的测试，准备将代码合并进 `main` 分支并发布到 Vercel 时，它会暂停并向你的 飞书 或微信群发送卡片：

```
🚨 [Codex Auth Requested]
Project: pmer-cn-saas
Action: Deploy to production (Vercel)
Change summary: Implemented Stripe subscription webhook in /api/stripe.
Tests: 12 passed, 0 failed.
[回执指令]: 回复 "1" 批准发布，回复 "0" 打断并回滚。
```

2. 实现交互的服务器端中转脚本 (Node.js 极简版)

假设在 `pmer\.cn` 的中转网关部署一个极简的接收端脚本，它会解析你手机端发出的微信或 飞书 命令，并通过远程控制端口（SSH / Codex Port）向下游的 Agent 实例发送信号：

```
// File: gateway.js (部署在你的 VPS 上)
const express = require('express');
const { exec } = require('child_process');
const app = express();
app.use(express.json());

// 接收来自微信/飞书的回复通知
app.post('/api/mobile-reply', (req, res) => {
  const { userMessage, user } = req.body;

  // 仅允许主理人 hunkwu 远程控制
  if (user !== 'hunkwu') {
    return res.status(403).json({ error: 'Unauthorized' });
  }

  if (userMessage === '1') {
    // 写入信号文件，下游 Codex hooks 监听此文件以批准部署
    exec('echo "approved" > /tmp/codex_deploy_signal', (err) => {
      if (err) return res.status(500).send('Error triggering deploy');
      res.json({ reply: '🚀 部署已批准，生产环境正在上线！' });
    });
  } else if (userMessage === '0') {
    // 强制终止 Codex 进程并回滚代码
    exec('pkill -f codex && git checkout -- .', (err) => {
      res.json({ reply: '🛑 部署已中止，代码已安全回滚至 HEAD！' });
    });
  } else {
    res.json({ reply: '⚠️ 无效指令，请回复 1 (批准) 或 0 (中止)' });
  }
});

app.listen(8080, () => console.log('Mobile gateway listening on port 8080'));
```

8.4 官方参考实现：飞书助理 (Codex Feishu Sentinel)

为了避免开发者重复手写 Webhook 网关与长连接轮询逻辑，本项目配套开源了开箱即用的官方参考工程：**plugins-codex-feishu（飞书助理）**。

它将上述架构全面产品化，具备三大核心形态：

1. **日常巡检与值班日报（极速上手）**：
   - 自动聚合本地 Git 提交与工作区动态，生成图文并茂的进展日报，免去每天下班写进展报告的痛点。
   - 默认通过飞书 Bot 仅推送到开发者个人私聊，3 分钟即可完成从零到一的验证。
2. **移动端警报与双向审批（离线编排）**：
   - 当云端/本地 Codex 执行自动化测试遇挫、或遇到需要人工授权的高危部署时，飞书第一时间震动弹窗通知卡片。
   - 手机端直接点击或打字回复指令，即可远程触发「批准发布」或「一键回滚中断」。
3. **沉淀与进阶知识库（团队协同）**：
   - 支持将结构化项目报告写回飞书 Docx 云文档、同步至 Bitable 多维表格，实现代码进展向团队业务知识库的无缝沉淀。

> 💡 ***3 分钟极速配置**：该工程内置了飞书官方应用清单 `feishu_app_manifest.json`，在飞书开放平台直接点击「导入应用清单」即可自动开通所有必要权限与长连接，免去繁琐的手工勾选。*

8.5 实战产品说心法：把自由留给自己

很多同行在用 AI 编程时，把自己活成了一个“人体测试机”和“Git 提交工具人”：AI 改完了，人去点刷新；AI 报错了，人复制报错发给 AI。

移动看护工作流的本质，是把人从“即时等待”中抽离出来。

通过将测试断言（Validation Specs）托付给 GitHub Actions，将异常通知托付给移动 Webhook，将最终签字权（Deploy Approval）掌握在手机端。你才能真正做到“人在喝咖啡，产品在自动进化”。

Ch.09 架构复苏：混乱遗留系统的全景解析与渐进式解耦

> 🎯 ***具体工程麻烦**：接手几万行无文档、无测试的屎山系统，改一行崩三处，重构不敢碰，盲目重写又交不了差。*
> 💡 ***可运行实战代码与落地收益**：全景逆向拓扑生成 Prompt（提取 ER 图与路由）；接口行为快照基准测试（Golden Master）；微创解耦三步法。*
> ⚡ ***社交传播 / 截图金句**：“面对几十万行没文档的遗留系统，别冲动推倒重来。先让 AI 画出地图、补上防崩测试，再做微创手术。”*

在独立开发或承接外包项目时，接手前人留下、无文档且无单测的「屎山」系统，远比从零构建更令人棘手，任何微小改动都伴随着潜在风险。

针对此类项目，并不建议盲目推倒重来。借助 Codex 的全景分析与推理能力，我们可以对遗留系统实施渐进式解耦与重构。本章将介绍如何指挥 Codex 进行项目分析与分层解耦。

9.1 第一步：全景逆向，建立代码拓扑图

接手混乱项目的首要任务是**画出地图**。别自己去翻代码，让 Codex 帮你进行项目逆向工程。

1. 逆向生成数据库依赖关系

如果项目使用了 Prisma 或 SQL，你可以让 Codex 直接根据数据库配置文件逆向出 Mermaid 实体关系图（ERD）。

在本地向 Codex 派发任务 Specs：

```
# 🎯 Goal
分析当前项目的数据库配置，生成一个包含核心表结构与外键关联的 Mermaid ERD 图。

# 🛑 Constraints
- 只分析 /prisma/schema.prisma 或 /src/db/models/ 下的文件。
- 排除临时表和第三方元数据表。

# 🧪 Validation Specs
- 在 /docs/database_topology.md 中输出渲染正确的 Markdown Mermaid 代码块。
```

Codex 会自动解析表之间的关系，并生成直观的拓扑图，这比你人肉去翻数据库关系要快上百倍。

9.2 第二步：安全红线，编写行为基准测试

重构遗留代码的铁律是：**先保证旧行为不被改坏，再动刀子。**

我们需要让 Codex 自动为现有的关键接口（例如购物车结算、OAuth 登录回调）编写**行为基准测试**，以此作为重构过程中的安全红线。

实战：向 Codex 下达基准测试指令

```
# 🎯 Goal
为 src/pages/api/checkout.ts 接口编写单元测试，捕捉当前的请求行为与返回格式。

# 🛑 Constraints
- 不要修改 src/pages/api/checkout.ts 中的任何逻辑。
- 使用本地沙盒中的 Jest / Vitest 运行测试。

# 🧪 Validation Specs
- 至少覆盖三种场景：商品正常结算、库存不足报错、未登录拦截。
- 确保测试运行通过率 100%。
```

有了这些基准测试（也就是 AI 的验收警戒线），后续的任何重构改动一旦破坏了历史逻辑，都会立刻在沙盒的编译测试中被拦截。

9.3 第三步：微创手术，实施渐进式解耦

拥有了“地图”（拓扑图）和“防线”（基准测试）后，我们就可以指挥 Codex 实施重构了。

1. 解耦胖控制器 (Fat Controllers)

假设我们有一个 500 行的 API 路由，里面混杂了“鉴权、折扣计算、发送邮件、扣库存、记录日志”等一堆逻辑。我们可以让 Codex 进行“逻辑抽离”：

```
# 🎯 Goal
将 src/pages/api/checkout.ts 中的“折扣计算逻辑”抽离到单独的服务类
src/services/discountService.ts 中。

# 🛑 Constraints
- 保证外部调用接口入参与出参不变。
```

```
- 严禁影响 checkout.ts 的其他核心逻辑。

# 🧪 Validation Specs
- 运行 npm run test，确保之前编写的结算基准测试 100% 通过。
- 运行 npm run lint 无任何语法错误。
```

2. 局部验证与安全回滚

当 Codex 开始重构时，它会在沙盒中执行修改并自动运行测试。如果测试失败，它会在 CoT 思维链中判断是自己的重构逻辑写错了，还是基准测试写错了。

通过这种"抽取逻辑 -> 运行测试 -> 报错回滚 -> 重新调整"的微创重构循环，你可以在短短几个小时内，把一个几万行、无法维护的遗留系统重构为清晰、模块化、带测试覆盖的现代系统。

不要害怕屎山，用 Specs 和测试做你的盾牌，把手写重构的工作让渡给 Codex。

Ch.10 商业实战：2小时跑通 Next.js + Stripe 商业级 MVP

> 🎯 ***具体工程麻烦***：*想做独立付费产品，在登录鉴权、数据库模型、Stripe Webhook 验签和部署上折腾两周，热情耗尽还没上线。*
> 💡 ***可运行实战代码与落地收益***：*完整可运行代码库 `examples/ch10-saas-mvp`（Next.js 15 + Supabase + Stripe 订阅）；Stripe CLI 本地支付闭环调试命令。*
> ⚡ ***社交传播 / 截图金句***：*"独立开发最核心的里程碑不是架构多完美，而是收到第一笔付款。2 小时搞定全套付费闭环。"*

作为独立开发者（Indie Hacker）或微型创业团队，你最核心的里程碑不是"完美架构"，应该是"收到第一笔付款"。很多人把时间浪费在了反复配置脚手架上，迟迟无法上线。

本章我们以极客速战速决的风格，教你如何指挥 Codex，在 2 小时内利用 `Next\.js 15 \(App Router\) \+ Supabase \(PostgreSQL\) \+ Stripe` 搓出一个具有完整支付与会员权限闭环的 SaaS MVP。

> 📦 ***配套实战源码***：*examples/ch10-saas-mvp —— 完整可运行的订阅制 AI 翻译工具（TransFlow），自带 CAP 协议 `AGENTS.md`，`npm install && npm run build` 已验证通过。*

10.1 初始化骨架与数据库 Schema 设计

我们的目标是做一款订阅制的 AI 翻译工具。首先，使用 Next.js 16 的 `create\-next\-app` 初始化项目，它会**默认生成 AGENTS.md**，你可以在此基础上锁死边界。接着，让 Codex 生成核心的数据库模型。

1. 编写 Prisma Schema (Prisma 实体建模)

向 Codex 下达目标 Specs，让其编写数据库模型：

```
# 🎯 Goal
编写 Prisma 数据库模型，支持用户表（User）、订阅表（Subscription）与翻译记录表
（TranslationRecord）。

# 🛑 Constraints
- 数据库驱动使用 PostgreSQL（连接 Supabase）。
- 订阅状态必须是 Enum 类型，包含 ACTIVE, CANCELED, EXPIRED。
```

```
# 🧪 Validation Specs
- 运行 `npx prisma validate` 无错误提示。
```

Codex 会自动输出标准的 `schema\.prisma` 模型，包含合理的级联删除（Cascade）和索引（Indexes）：

```
// File: prisma/schema.prisma
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

enum SubscriptionStatus {
  ACTIVE
  CANCELED
  EXPIRED
}

model User {
  id             String               @id @default(uuid())
  email          String               @unique
  createdAt      DateTime             @default(now())
  subscription   Subscription?
  records        TranslationRecord[]
}

model Subscription {
  id             String               @id @default(uuid())
  userId         String               @unique
  stripeSubId    String               @unique
  status         SubscriptionStatus
  priceId        String
  currentPeriodEnd DateTime
  user           User                 @relation(fields: [userId], references: [id],
onDelete: Cascade)
}

model TranslationRecord {
  id             String               @id @default(uuid())
  userId         String
  sourceText     String
  translatedText String
  createdAt      DateTime             @default(now())
  user           User                 @relation(fields: [userId], references: [id],
onDelete: Cascade)
}
```

10.2 集成 Stripe 订阅与 Webhook 监听

支付系统的核心是**回调安全**。当用户支付成功后，Stripe 的服务器会向你的 Next.js 服务发送一个 Webhook 请求。我们需要 Codex 帮我们快速编写这套验证与状态流转逻辑。

实战：向 Codex 下达 Webhook 路由编写 Specs

```
# 🎯 Goal
实现 Next.js 16 App Router 风格的 Stripe Webhook 路由处理程序 `/api/webhooks/stripe`。

# 🛑 Constraints
- 必须使用 `stripe.webhooks.constructEvent` 验证请求签名的真实性，防止伪造攻击。
- 当接收到 `checkout.session.completed` 或 `invoice.payment_succeeded` 事件时，更新用户的订阅状态。
- 必须使用最新稳定的 Stripe API 版本（截稿时为 2026-04-22.dahlia）。
- 严禁对响应体进行普通 JSON 解析，Stripe 验证要求使用原始 body 字符串。

# 🧪 Validation Specs
- 编写对应的自动化 Mock 请求测试，验证签名失败时返回 400，成功时返回 200。
- 真实订阅周期必须从 Stripe 响应中读取，不得硬编码。
```

Codex 会在沙盒中自主寻找正确的处理方案（例如如何读取 Next.js 16的 `request\.text\(\)` 获取 Raw Body 进行验证），并输出符合规范的代码：

```
// File: src/app/api/webhooks/stripe/route.ts
import { NextResponse } from 'next/server';
import Stripe from 'stripe';
import { prisma } from '@/lib/prisma';

const stripe = new Stripe(process.env.STRIPE_SECRET_KEY!, {
  // 使用截稿时最新稳定的 API 版本；上线前查阅 Stripe Changelog 确认
  apiVersion: '2026-04-22.dahlia',
});

export async function POST(req: Request) {
  const body = await req.text();
  const signature = req.headers.get('stripe-signature')!;

  let event: Stripe.Event;

  try {
    event = stripe.webhooks.constructEvent(
      body,
      signature,
      process.env.STRIPE_WEBHOOK_SECRET!
    );
  } catch (err: any) {
    return NextResponse.json({ error: `Webhook Error: ${err.message}` }, { status: 400
});
  }
```

```
  // 状态流转处理
  if (event.type === 'checkout.session.completed') {
    const session = event.data.object as Stripe.Checkout.Session;
    const stripeSubId = session.subscription as string;
    const customerEmail = session.customer_details?.email!;

    // 生产实践：从 Stripe 拉取真实的订阅对象，拿到准确的 current_period_end
    const subscription = await stripe.subscriptions.retrieve(stripeSubId);

    await prisma.subscription.upsert({
      where: { stripeSubId },
      update: {
        status: 'ACTIVE',
        currentPeriodEnd: new Date(subscription.current_period_end * 1000),
      },
      create: {
        stripeSubId,
        status: 'ACTIVE',
        priceId: subscription.items.data[0]?.price.id || 'default',
        currentPeriodEnd: new Date(subscription.current_period_end * 1000),
        user: { connect: { email: customerEmail } }
      }
    });
  }

  return NextResponse.json({ received: true });
}
```

> 💡 ***避坑提醒***：*原始版本里很多教程会硬编码 `currentPeriodEnd = now\(\) \\\+ 30 天`，这在跨年订阅、试用期、Coupon 折扣等场景下**会算错**。务必从 `stripe\\\.subscriptions\\\.retrieve` 拿真实数据。*

10.3 本地联调：用 Stripe CLI 通过支付闭环

Stripe 在本地调试需要使用 Stripe CLI 进行 Webhook 转发。我们可以利用 Ch.03 学到的沙盒网络穿透技术，配置本地调试：

1. 在本地运行 Stripe 转发：

```
stripe listen --forward-to localhost:3000/api/webhooks/stripe
```

2. 将控制台给出的 `whsec\_xxx` 写入本地的 `\.env` 中，让 Codex 运行单元测试。

通过这种"Specs 定义接口 -> AI 编码 -> 沙盒校验 -> Stripe 真实联动测试"的方式，独立开发者可以把原本需要折腾两天的第三方集成工作缩短到半小时内。

商业 MVP 的价值在于速度。用最牢靠的边界规则约束 AI，换取最极致的上线时间。

Ch.11 触角延伸：Expo 跨端原生 App 开发与云端打包

> 🎯 ***具体工程麻烦***：*做完 Web 想做 App，但 Xcode 证书、Android Gradle、Cocoapods 依赖地狱让人崩溃，环境配三天打不出包。*
> 💡 ***可运行实战代码与落地收益***：*完整可运行代码库 `examples/ch11-expo-mobile`（Expo SDK 57 + Expo Router + NativeWind）；零本地配置云端打包 `eas build`。*
> ⚡ ***社交传播 / 截图金句***：*“不用配本地 Xcode 和 Android Studio，借助云端打包与 AI 报错自愈，一个人也能轻松交付双端原生应用。”*

做完网页版 SaaS 后，很多独立开发者希望能将触角延伸到移动端。但在传统的移动端开发（React Native / Flutter）中，最耗费时间的往往是复杂的本地开发环境配置：iOS 证书管理、Android Gradle 报错、Cocoapods 冲突，这些环境地狱常常让人望而却步。

我坚信 **“云端开发与打包（EAS）是独立开发者做原生 App 的唯一解”**。结合 Codex 的智能编译报错排查，你可以完全跳过本地 Xcode/Android Studio 的繁琐配置，直接打包出可以上架的原生 App。

本章教你如何让 Codex 自主搞定这一切。

> 📦 ***配套实战源码***：*examples/ch11-expo-mobile —— 完整可运行的 Expo SDK 57 工程（Expo Router `src/app` 路由 + NativeWind + EAS 三档打包配置），自带 CAP 协议 `AGENTS.md`，已通过 `npx expo lint`（零错误）与 `npx expo-doctor`（20/20）验证。*

11.1 Expo 项目极速初始化与模拟器映射

为了能使用 EAS（Expo Application Services）进行云端免配置打包，我们首先初始化一个标准的 Expo 项目。

1. 编写 App 初始化 Specs

给 Codex 下达 Specs 指令，生成标准的 Expo TS 项目：

```
# 🎯 Goal
初始化一个使用 TypeScript 的 React Native Expo 项目。

# 🛑 Constraints
- 使用最新稳定的 Expo SDK（当前为 SDK 56），随之搭配最新版 Expo Router 实现基于文件系统的路由。
- 引入 NativeWind 作为 Tailwind 风格的样式方案。
- 目录约定使用 src/ 前缀（即 src/app/ 作为路由根）。

# 🧪 Validation Specs
- 运行 `npx expo lint` 无错误。
- 运行 `npx expo-doctor` 检查依赖版本对齐。
```

Codex 会自动拉取最新的 Expo 模板并生成基础目录结构：

```
+-- src/
|   +-- app/
|   |   +-- index.tsx          # APP 首页
|   |   +-- _layout.tsx        # 全局路由导航布局
|   +-- components/
|   +-- hooks/
```

```
+-- app.json                    # Expo 核心配置文件
+-- package.json
```

11.2 解决移动端顽疾：原生依赖冲突

React Native 开发最怕升级或引入带 Native 模块的第三方库（例如相机、文件系统访问）。这常常导致 iOS Podfile 或 Android build.gradle 报错崩溃。

在 Vibe Coding 模式下，当 Codex 执行自动化依赖安装时，如果遇到此类原生报错，我们应引导它采用下述排查策略。

实战：如何让 Codex 自主排查 Podfile 报错

如果 Codex 在沙盒中编译 iOS 原生模块失败，它会在推理摘要中产生类似提示：

`\[Error\] \[Cocoapods\] Auto\-linking failed for react\-native\-reanimated\.`

此时**直接在 TUI 中继续输入纠偏指令**（无需任何特殊子命令，详见 Ch.06）：

```
请改用 `npx expo install react-native-reanimated` 重新安装该依赖，它会自动适配当前的 Expo SDK 版本。在本项目中，禁止使用普通的 `npm install` 安装任何带原生代码的第三方包。请把这条规则补充到 AGENTS.md。
```

> 💡 ***主理人心法***：*Expo SDK 最强的地方就在于其自带的版本对齐机制（`npx expo install`）。任何时候只要原生包报错，逼着 Codex 使用 expo 内置指令覆盖安装，90% 的版本冲突都会被自动抹平。*

11.3 EAS Build 云端打包与证书自动化

在传统的 App 打包流程中，申请苹果开发者证书、生成 Provisioning Profile 往往会卡住新手几天。现在，通过 EAS，我们只需要给 Codex 提供开发者账号，让其在云端全自动解决。

1. 配置 `eas\.json` (EAS 云端打包配置文件)

让 Codex 生成生产与测试的云端打包环境配置：

`eas\.json` 配置文件：

```
{
  "cli": {
    "version": ">= 9.0.0"
  },
  "build": {
    "development": {
      "developmentClient": true,
      "distribution": "internal"
    },
    "preview": {
      "distribution": "internal"
    },
    "production": {
      "ios": {
        "simulator": false
      }
```

```
    }
  }
}
```

2. 让 Codex 监考云端构建日志

在本地终端触发 EAS 构建任务，并把日志保留下来供 Codex 分析：

```
# 启动 EAS iOS 打包任务，把日志写到本地文件
eas build --platform ios --profile production --non-interactive 2>&1 | tee eas-build.log
```

构建结束（或失败）后，直接把日志喂给 Codex：

```
# 把构建日志交给 Codex 做诊断
codex exec "分析 eas-build.log，定位失败原因，并给出修复方案。如果是 Provisioning Profile 不匹配或证书失效，告诉我具体要运行哪些 eas credentials 命令。"
```

> ⚠️ ***澄清边界***：*Codex **不会自动登录你的 Apple Developer 后台帮你重发证书**——证书签发与轮换是 EAS（通过 `eas credentials`）和 Apple 之间的事。Codex 能做的是：读懂 EAS 的报错日志、识别证书过期 / Provisioning Profile 不匹配等典型问题、指引你运行正确的修复命令、自动生成补丁脚本。*

最终 EAS 会返回一个安装二维码，你只需用手机扫码即可直接安装测试版 App。

原生开发不再需要笨重的 IDE。用 Expo + EAS 让打包上云，让 Codex 成为你移动开发的副驾驶。

Ch.12 终局思考：独立开发者如何打造自动化商业飞轮

> 🎯 ***具体工程麻烦***：*花 99% 精力写代码，只花 1% 精力找客户，产品上线无人问津；一人微型创业无法兼顾开发与获客。*
> 💡 ***可运行实战代码与落地收益***：*自动化营销脚本（业务事件触发日报推送、社媒动态分发）；核心数据看板模板；一人公司商业运转流水线。*
> ⚡ ***社交传播 / 截图金句***：*“代码写得再漂亮，没人用就是精美的垃圾。让 AI 不仅帮你写代码，更帮你转动商业获客的轮盘。”*

在“实战产品说”微信公众号和 pmer.cn 上，我写过很多关于“独立开发与副业变现”的文章。我发现，很多开发者最容易走进的死胡同是：**把 99% 的精力用来优化代码，却只花 1% 的精力去寻找用户和真实需求。**

代码写得再优雅、架构配置得再完美，只要没人用，它就是一个精美的摆设。在 AI 时代，我们不仅要让 Codex 帮我们“生产产品”，更要让它帮我们“转动商业轮盘”。

作为全书的终局，我们来聊聊如何用 AI 实现自动化营销与独立增长。

12.1 “一人 SaaS 公司”的自动化流量管道

一个健康的独立项目，其流量获取（SEO、社交媒体、冷启动邮件）应该和它的代码库一样，是一套自动运转的流水线。

实战：让 Codex 自主维护 SEO 博客与 Sitemap

我们可以写一段 Node.js 脚本，让 Codex 每天根据最新的热点关键词，自动抓取行业文章、总结成高质量的博文，并更新静态路由与 Sitemap（网站地图），以此获取长尾流量。

在项目中建立自动生成站点地图的脚本 Specs：

```
# 🎯 Goal
实现一个自动遍历 Next.js 静态文件路由并重新生成 /sitemap.xml 文件的脚本 `scripts/generate-sitemap.js`。

# 🛑 Constraints
- 必须包含 /blog 目录下所有新生成的 markdown 文章路径。
- 修改频率（changefreq）和权重（priority）符合 Google 抓取规范。

# 🧪 Validation Specs
- 运行 `node scripts/generate-sitemap.js`，生成的 XML 文件在浏览器中解析正常，没有任何格式缺失。
```

Codex 会自动输出标准的解析脚本：

```
// File: scripts/generate-sitemap.js
const fs = require('fs');
const path = require('path');

const BASE_URL = 'https://pmer.cn';
const pagesDir = path.join(__dirname, '../src/app');
const blogDir = path.join(__dirname, '../content/blog');

function getBlogSlugs() {
  if (!fs.existsSync(blogDir)) return [];
  return fs.readdirSync(blogDir)
    .filter(file => file.endsWith('.md'))
    .map(file => `/blog/${file.replace('.md', '')}`);
}

function generate() {
  const staticPages = ['/', '/auth/login', '/features'];
  const blogPages = getBlogSlugs();
  const allUrls = [...staticPages, ...blogPages];

  const xml = `<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  ${allUrls.map(url => `
  <url>
    <loc>${BASE_URL}${url}</loc>
    <changefreq>daily</changefreq>
    <priority>${url === '/' ? '1.0' : '0.8'}</priority>
  </url>`).join('')}
</urlset>`;

  fs.writeFileSync(path.join(__dirname, '../public/sitemap.xml'), xml);
  console.log('✅ Sitemap.xml generated successfully!');
}
```

```
generate();
```

12.2 对接业务数据，获取每日商业战报

为了让你对钱的动向足够敏感，你可以让 Codex 写一个极简的战报脚本，每天早上拉取 Stripe 昨天的收益和 Supabase 的新增用户数，直接通过 Webhook 发送到你的手机。

每日数据通报脚本 (Node.js)

```
// File: scripts/daily-report.js
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();
const axios = require('axios');

async function sendReport() {
  // 获取昨日新增用户数
  const yesterday = new Date(Date.now() - 24 * 60 * 60 * 1000);
  const userCount = await prisma.user.count({
    where: { createdAt: { gte: yesterday } }
  });

  // 获取激活的订阅数
  const activeSubs = await prisma.subscription.count({
    where: { status: 'ACTIVE' }
  });

  const reportText = `📊 【实战产品战报】\n昨日新增注册用户: ${userCount} 人\n当前总活跃订阅: ${activeSubs} 个\n—— 继续加油！`;

  // 微信/飞书 机器人推送
  await axios.post(process.env.MOBILE_WEBHOOK_URL, {
    msgtype: 'text',
    text: { content: reportText }
  });
}

sendReport();
```

将其挂载到服务器的 crontab 上（每天 8:00 触发），就构成了你最低成本的"业务监控大屏"。

12.3 终局心法：AI 原生时代的唯一壁垒

当任何人都能在两小时内写出几万行代码、打包出原生 App、甚至挂接好自动化流量管道时，**技术本身彻底商品化了**。在这个“代码泛滥”的新时代，独立开发者和产品经理真正的终极壁垒应该是：

- **你对用户痛点深刻的同理心（User Empathy）**。
- **你在具体行业中沉淀多年的业务认知（Domain Knowledge）**。

- **你将 AI 原生工具（如 Codex）化为己用、快速落地并形成商业闭环的执行力。**

不要做纯粹的打字员，去做那个定义问题、手握缰绳并解决真实痛点的人。

感谢你读完《Codex 蓝皮书》，收藏+转发、github star～

现在，请在你的根目录下，配置好 `AGENTS\.md`，运行你的第一行 `codex` 指令，创造属于你的商业故事。

Ch.13 前沿瞭望：2026 Codex 生态全景升级

> 🎯 ***具体工程麻烦***：*工具迭代极快，旧命令作废（Codex 桌面端退役并入 ChatGPT 代码模式）、模型换代（GPT-5.6）、CLI 变更不知所措。*
> 💡 ***可运行实战代码与落地收益***：*2026 迁移命令作战清单（winget/brew 安装、CLI 0.14x 变更配置、Plugins 插件系统对接）；版本兼容检查脚本。*
> ⚡ ***社交传播 / 截图金句***：*“工具每三个月换一轮血，心智才是护城河。这里没有空洞新闻，只有告诉你哪条命令已作废的作战地图。”*

前面十二章建立的，是一套不依赖具体版本的编排方法论。但方法论要落地，就必须踩在真实的工具地面上。2026 年的 Codex 生态发生了四次结构性地震：**桌面端大合并、模型大换代、插件生态成型、安全能力独立成军**。本章逐条拆解这些变化，并给出具体的迁移命令与配置。

13.1 桌面端大合并：Codex App 并入 ChatGPT 客户端

2026 年 7 月，独立的 Codex 桌面应用正式退役，全部能力并入 **ChatGPT 桌面客户端**，以「Codex 代码模式（Code Mode）」的形态存在。免费、Plus 与企业版用户均可使用。

迁移动作：

```
# macOS：直接下载 ChatGPT.dmg，拖入 Applications
# Windows：使用 winget 安装
winget install OpenAI.ChatGPT
```

登录后点击左侧边栏的 **Codex** 标签页即可进入代码模式。需要特别注意的两点：

1. **账号 vs API Key**：使用 ChatGPT 账号登录可获得完整能力（云端任务、跨端同步、Computer Use）；使用 API Key 登录仅有本地基础编码能力。
2. **独立组件不受影响**：Codex CLI、VS Code 插件、Codex Cloud 均为独立产品，继续正常演进。本书 Ch.02 的多端矩阵依然成立，只是「桌面 App」这一格换成了 ChatGPT 客户端。

新增能力速览：

- 内置浏览器升级：地址栏直接搜索浏览历史，Chrome 扩展可引用当前打开的标签页。
- **多仓库审查（Multi-repo Review）**：多文件夹项目可在一个视图中查看所有仓库的变更行数并逐一审查 Diff，不再需要来回切换。
- **Sites**：在客户端内直接创建、部署、管理托管 Web 项目，配合 Annotations 实现「指哪改哪」的原位编辑。

13.2 模型大换代：GPT-5.6 Terra 与 Luna 接管

2026 年 8 月 31 日起，`GPT-5.4` 与 `GPT-5.4 mini` 在 Codex（ChatGPT 登录态）中正式退役。官方指定的替代关系：

退役模型	继任模型	定位
`gpt-5.4`	`gpt-5.6-terra` (GPT-5.6 Terra)	主力深度推理模型
`gpt-5.4-mini`	`gpt-5.6-luna` (GPT-5.6 Luna)	轻量快速模型（Guardian 审批等场景）

迁移检查清单：

```
# 1. 检查你的配置文件中是否硬编码了旧模型
grep -rn "gpt-5.4" ~/.codex/config.toml .

# 2. 在 config.toml 中显式升级默认模型
```

```
# ~/.codex/config.toml
model = "gpt-5.6-terra"
# 轻量后台任务（如自动审查）可单独指定
[profiles.guardian]
model = "gpt-5.6-luna"
```

> ⚠️ *注意：使用 API Key 认证的会话仍可继续调用 GPT-5.4 系列，但 ChatGPT 登录态必须切换。所有定时任务（Automations）、工作区默认配置、自定义 Agent 都要逐一排查。*

13.3 CLI 0.14x：必须知道的决定性变更

Codex CLI 已进入 `0.14x` 时代（截至 2026 年 8 月最新稳定版 `0.147.0`）。以下变更直接影响本书前文的所有命令示例：

```
npm install -g @openai/codex@latest
```

1. `--full-auto` 正式移除

```
# ❌ 旧写法（已报错）
codex exec --full-auto "修复所有 lint 错误"

# ✅ 新写法：用沙盒模式表达自治级别
codex exec --sandbox workspace-write "修复所有 lint 错误"
```

2. Hooks 引擎转正（Stable）

Ch.05 中提到的「拦截层」现在是一等公民。可在 `config.toml` 中直接配置 `SessionStart` / `Stop` 钩子，并能观测 MCP 工具调用、`apply_patch` 与长时间运行的 Bash 会话：

```
# ~/.codex/config.toml
[hooks]
session_start = "bash scripts/bootstrap-env.sh"
stop = "npm run lint --silent"
```

3. Agent Plugins 与插件市场

插件体系从实验走向正式，支持本地、个人、工作区与远程四层目录，并可安装面向 Amazon Bedrock、Claude Code 的第三方市场：

```
# 插件管理统一入口
codex plugin list
codex plugin marketplace add https://plugins.example.com/catalog.json
codex plugin install security-workbench
```

在对话中通过 `@plugin` 提及即可自动注入插件上下文（MCP / App / Skill）。

4. 子智能体（Subagents）与多智能体编排

CLI 原生支持派生拥有独立上下文窗口的子智能体，可将大任务切分为并行流水线：

```
> 派生两个子智能体：一个为 src/api 补充集成测试，
  另一个并行重构 src/hooks 的状态管理，最后汇总 Diff 给我审查。
```

配合 `git worktree` 隔离，多个智能体可在互不污染的工作区并行作业——这正是 Ch.04「目标驱动」思想的官方实现。

5. Guardian 自动审批与 `--approve-for-me`

高危操作不再只有「人肉点确认」一条路。新的 `--approve-for-me` 标志可将审批请求路由给 Guardian 子智能体（由 GPT-5.6 Luna 驱动）自动评审，符合策略的操作自动放行，越界操作拦截并给出理由：

```
codex --approve-for-me "升级依赖并跑通测试"
```

这与本书 Ch.08 的「移动端审批网关」互为补充：低风险交给 Guardian，高风险才推到你的手机。

6. MCP 2026-07-28 协议

支持分页发现、多轮请求与非阻塞服务器启动。旧的 `.mcp.json` 配置继续兼容，但新写的 MCP Server 建议按 2026-07-28 规范实现。

13.4 Skills 生态：把重复流程沉淀为资产

如果说 AGENTS.md 是项目的「通用法律」，**Skills 就是某一类任务的「专项流程」**：PR Review、飞书文档整理、PPT 生成、CI 修复、安全扫描……任何会重复三次以上的流程，都值得沉淀为 Skill。

```
# Skill 的标准目录结构（放在 ~/.codex/skills/ 或插件包内）
skills/
└── pr-review/
    └── SKILL.md      # 带 YAML frontmatter 的流程说明书
```

Skills 可与插件一起发布到市场，供团队统一安装。这实现了 Ch.05 CAP 协议的「跨项目复用」：协议约束行为，Skill 固化流程。

13.5 Codex Security 与 Daybreak：安全能力独立成军

2026 年 8 月，OpenAI 推出面向防御方的 **Daybreak** 双层访问体系：

- **Daybreak Blue**：通用模型（GPT-5.6 Sol），覆盖漏洞发现、安全代码审查、检测工程、事件响应、恶意软件分析与补丁验证——绝大多数防御性安全工作从这里开始。
- **Daybreak Red**：专用训练模型（GPT-5.6 Cyber），用于经明确授权的漏洞复现、利用验证、渗透测试与红队行动，需单独审批开通。

配合 **Codex Security 插件与 CLI**，安全工作流全面产品化：

```
# 在工作台中跑一次深度扫描
codex plugin install codex-security
# 或在 CI 中批量扫描
codex-security scan --deep --report sarif > results.sarif
```

实战红线（与 Ch.05 的沙盒边界一脉相承）：

1. 在**隔离环境**中工作，显式定义授权范围（engagement scope）。
2. 使用最小权限的 Permission Profiles。
3. 为跨越沙盒边界的动作配置 **Auto-review**，让 Guardian 先行评审。

13.6 本章迁移清单（TL;DR）

```
# ① 桌面端：弃用旧 Codex App，改装 ChatGPT 客户端
winget install OpenAI.ChatGPT        # Windows
# macOS 下载 ChatGPT.dmg

# ② CLI 升级到 0.147.0+
npm install -g @openai/codex@latest

# ③ 模型切换到 GPT-5.6 系列（8月31日前必须完成）
grep -rn "gpt-5.4" ~/.codex/ .       # 全面排查

# ④ 替换已移除的 --full-auto
codex exec --sandbox workspace-write "<task>"

# ⑤ 启用 Guardian 自动审批
codex --approve-for-me "<task>"

# ⑥ 安装插件市场与安全插件
codex plugin marketplace add <catalog-url>
codex plugin install codex-security
```

13.7 终局不变量

工具层面的结论只有一条：**凡是写死在脚本里的版本号、命令行标志、模型代号，都是技术债**。把它们集中到 `config.toml` 与 `AGENTS.md` 中管理，让迁移成本收敛到「改一行配置」。

而穿越所有版本周期仍然成立的，正是本书反复强调的三件事：目标驱动而非步骤驱动、边界先行而非事后追责、人机分工而非人被 AI 牵着走。工具会换血，心智不打折。
