面向零基础学习者的系统入门指南

我的AI笔记
WO DE AI BI JI

Codex
零基础学习手册

从第一次使用，到项目实战与交付落地

真实项目驱动的
Codex学习路径

我的AI笔记 × 前哨科技

版本与使用说明

版本	V2.1-F
正文范围	第 1—40 章
版本状态	正式稿
出品	我的 AI 笔记 × 前哨科技

修订原则

完整保留 V2.0-F 已经通过 Review 的正文、代码示例、章节结构和教学结论，仅调整配套资料、外部依赖和综合案例的定位话术；未列入修订范围的内容不压缩、不改写。

使用提示

产品界面、命令参数和第三方工具版本可能随时间更新，实际操作时以正文标注日期及对应官方文档为准。

配套资料说明

本书大部分练习由读者按照正文，从空白项目或自己的现有项目逐步完成。随书轻量配套资料包括一份《Codex 实用提示词与模板合集》，以及第 17—36 章持续使用的 team-task-board 基础练习项目。第 1—16 章按照正文逐步创建项目；第 19 章使用正文提供的固定错误示例完成调试训练；第 37—40 章为企业工单系统综合案例拆解，不要求下载或运行同名项目。

目录

第 1 章：Codex 是什么，它能帮你完成什么

一、起始语

你可能已经习惯用 AI 回答问题：

> 帮我写一段网页代码。
> 解释一下这个报错。
> 给我一个任务管理工具的制作方案。

在普通聊天模式中，AI 通常先给出文字、建议或代码片段。接下来，你还要自己创建文件、找到代码应该放置的位置、运行项目、处理报错，再确认结果是否符合要求。

Codex 把这件事向前推进了一步。

它可以围绕一个真实项目查看文件、搜索内容、修改代码、运行命令和检查结果，再把实际变化交给你审查。OpenAI 将 Codex 定义为帮助用户编写、审查和交付代码的编码智能体。OpenAI Help Center

本章暂时不讲下载安装，也不要求你编写代码。我们先解决三个最基础的问题：

> Codex 是什么？
> 它适合完成什么任务？
> 哪些决定仍然必须由人负责？

二、主体

1.1 Codex 是什么

Codex 是 OpenAI 的编码智能体。

“编码”说明它主要面向软件、网页、脚本和其他技术成果；“智能体”说明它不只生成一段回答，还可以围绕一个目标连续采取行动。

例如，你想修改一个已有网页的搜索功能。Codex 可以先查看项目结构，找到相关文件和现有逻辑，再完成修改、运行检查，并汇报改动和仍未验证的部分。

理解 Codex 时，可以先记住这句话：

> **Codex 能够进入被授权的工作环境，围绕真实项目完成一组可以检查的技术动作。**

它最终交付的通常不只是一段回答，还可能包括：

- 创建或修改后的文件；
- 项目的实际运行结果；
- 测试和检查记录；
- 修改前后的代码差异；
- 可以继续审查的候选成果。

1.2 它和普通 AI 对话有什么区别

Codex 与普通 AI 对话并非互相替代，它们适合处理的问题不同。

任务	普通 AI 对话	Codex
解释概念、讨论方案	很适合	可以，但通常没有必要
生成一段示例代码	很适合	可以
查看一个完整项目	依赖你提供材料	可以读取被授权的项目
修改多个关联文件	通常需要你手工衔接	可以在项目中连续处理
运行命令和测试	普通聊天通常不操作本地项目	可以在相应环境中执行
检查实际差异	需要你自行整理	可以展示真实修改
准备可审查的工程成果	需要较多人工衔接	是 Codex 的典型用途

可以用一句话概括：

> 普通 AI 对话更擅长帮助你思考和获得答案；Codex 更擅长在项目中推进任务并交付实际变化。

当前 ChatGPT 桌面应用在 ChatGPT 下提供 Chat 和 Work，并将 Codex 作为并列的独立视图：Chat 用于快速交流，Work 用于较长的研究和成品任务，Codex 专门面向软件开发与技术工作。OpenAI Help Center

1.3 Codex 通常怎样完成一项任务

一项 Codex 任务通常会经历这样的过程：

> 理解目标 → 查看项目 → 找到相关内容 → 修改或创建文件 → 运行检查 → 汇报结果

以本书后面要制作的“AI 学习导航页”为例，Codex 可能会：

1. 查看当前文件夹里已经有什么；
2. 判断需要创建哪些页面文件；
3. 写入页面内容和样式；
4. 检查文件之间的引用是否正确；
5. 根据你的反馈继续修改；
6. 告诉你实际改变了哪些内容。

这里展示的是整体过程，不要求你现在记住一套复杂的操作方法。

怎样描述任务、怎样要求 Codex 先查看项目、怎样检查修改证据，会在后面的相应章节中分别学习。

现在只需要建立一个认识：

> **Codex 的价值来自围绕真实对象持续行动，而不只是生成一段看起来合理的内容。**

1.4 Codex 最适合完成哪些任务

Codex 最典型的应用集中在软件开发和技术执行领域。

理解已有项目

它可以帮助查找项目入口、主要模块、功能位置和数据流，让使用者更快进入陌生代码库。

开发或修改功能

例如：

- 创建一个页面；
- 为工具增加搜索；
- 调整表单和页面交互；
- 修改接口返回结果；
- 在多个关联文件中完成一致的变化。

定位和修复问题

Codex 可以结合报错、日志、代码和测试寻找原因，完成候选修复，再通过运行结果检查修改是否有效。

重构和迁移

当项目需要替换旧接口、升级依赖、清理重复代码或批量修改文件时，Codex 可以承担大量重复且需要保持一致的工作。

补充测试和审查代码

它可以根据现有逻辑补充测试、检查边界条件、审查修改差异，并指出潜在风险。

自动化重复的技术工作

当一套流程已经稳定后，Codex 还可以参与定期检查、批量维护和持续集成辅助。

这些能力与 OpenAI 目前对 Codex 的官方定位一致：它可以围绕代码库执行开发、调试、测试、审查和交付工作。OpenAI Help Center

1.5 哪些事情不一定需要 Codex

并不是所有问题都要进入 Codex 项目。

下面这些任务，普通 AI 对话通常更加直接：

- 解释一个概念；
- 修改一句文案；
- 比较两个简单方案；
- 讨论一个尚未形成明确成果的想法；
- 回答一个不需要查看或操作文件的问题。

当一项任务既没有真实工作对象，也不需要修改、运行或验证结果时，使用 Codex 未必能够带来额外价值。

反过来，当任务同时具备以下特征时，Codex 通常更加合适：

- 存在一个项目、一组文件或一个可以运行的成果；
- 需要连续完成读取、搜索、修改、运行或测试；
- 最终结果可以通过页面、文件、差异或测试进行检查。

1.6 不会编程，也能使用 Codex 吗

可以。

零基础读者不需要先学会独立编程，才能开始使用 Codex。你可以从自己能够理解和判断的结果出发，例如：

- 一个可以打开的网页；
- 一个整理资料的本地工具；
- 一个批量处理文件的小程序；
- 一个已有项目中的明确功能修改。

但“可以开始使用”并不代表完全不需要学习。

随着任务变复杂，你需要逐步掌握三类基础能力：

7. 说清楚希望得到什么结果；
8. 看懂 Codex 准备操作哪些文件；
9. 判断最终成果是否真的可用。

这正是本书前半部分要解决的问题。我们不会先用大量代码语法和技术术语把你挡在门外，而是先让你完成真实成果，再在使用过程中理解文件、终端、项目和版本。

1.7 人仍然需要负责什么

Codex 能够执行很多动作，但不会替你承担最终责任。

你仍然需要决定：

- 目标是否清楚；
- 哪些文件和数据可以访问；
- 哪些内容允许修改；
- 哪些操作需要批准；
- 什么结果才算完成；
- 成果能否提交、发布或交付。

例如，Codex 可以修改网页，但页面是否符合业务要求，需要你判断。

它可以补充测试，但测试是否覆盖了真实风险，需要你确认。

它可以准备一个 Pull Request，但是否合并到正式项目，仍然由人决定。

因此，本书会始终坚持一条基本原则：

> **Codex 负责在明确边界内推进任务，人负责目标、授权和最终验收。**

三、总结

本章建立了对 Codex 的第一层认识：

- Codex 是能够操作真实项目的编码智能体；
- 它可以读取文件、修改内容、运行工具并交付可检查的结果；
- 它适合理解项目、开发功能、修复问题、补充测试和处理重复技术任务；
- 零基础读者可以从自己看得懂的成果开始使用；
- 目标、权限和最终判断仍然由人负责。

本章掌握检查

- [ ] 我能用自己的话说明 Codex 是什么；
- [ ] 我能区分“获得回答”和“在项目中完成实际修改”；
- [ ] 我能说出三类适合交给 Codex 的任务；
- [ ] 我知道哪些简单问题不一定需要使用 Codex；
- [ ] 我知道最终验收不能交给 Codex 自行决定。

产品信息核对日期：2026 年 7 月 31 日。

四、结束语与引导

现在，你已经知道 Codex 适合处理什么类型的任务，也理解了它与普通 AI 对话的主要区别。

下一步要解决的是一个现实选择：

> **第一次使用 Codex，应该从哪个入口开始？**

第 2 章：访问和下载 Codex，找到适合自己的使用入口

一、起始语

上一章解决了“Codex 是什么”。

接下来会遇到一个更实际的问题：

> Codex 在哪里？第一次使用应该下载什么？命令行、编辑器和云端是不是都要同时学习？

Codex 目前可以通过 ChatGPT 桌面应用、Codex CLI、IDE 扩展和 Codex 云端等形态使用。它们适合的任务不同，零基础读者没有必要一次全部安装。OpenAI Developers

本书选择的第一条路线是：

> **ChatGPT 桌面应用 → Codex 视图 → 本地项目文件夹。**

这一章只负责帮助你认清入口、选定第一条路线并找到官方下载位置。具体安装和配置放在第 3 章。

二、主体

2.1 Codex 有哪些主要入口

入口	主要特点	本书学习位置
ChatGPT 桌面应用中的 Codex	图形界面清楚，适合本地文件和持续项目	第 2—25 章主要入口
Codex CLI	在终端和当前目录中工作	第 26 章
Codex IDE 扩展	在代码编辑器旁查看和修改代码	第 27 章
Codex 云端	在独立云端环境中运行仓库任务	第 28 章

ChatGPT 桌面应用中的 Codex

桌面应用适合：

- 打开本地文件夹；
- 管理项目和聊天；
- 查看创建或修改的文件；
- 审查实际差异；

- 推进持续时间较长的任务。

官方快速入门也将桌面应用作为处理本地文件、项目和长任务的推荐入口。OpenAI Developers

Codex CLI

CLI 是在终端中使用 Codex 的方式。

它更适合已经理解终端、目录和命令的使用者。当前不需要安装，第 26 章会在具备相关基础之后再正式学习。OpenAI Developers

Codex IDE 扩展

IDE 扩展让 Codex 出现在代码编辑器旁边，可以利用当前打开的项目、文件和代码上下文完成解释、修改和调试。OpenAI Developers

Codex 云端

Codex 云端用于在独立云端环境中运行代码仓库任务。它适合后台执行较长任务、并行比较多个方案，或者在离开本地开发电脑时继续推进工作。OpenAI Developers

本章只帮助你认识这些入口，不进行 CLI、IDE 或云端配置。

2.2 为什么零基础读者先从桌面应用开始

零基础读者目前还没有系统学习：

- 终端；
- 工作目录；
- Git；
- 代码编辑器；
- 项目依赖；
- 远程代码仓库。

如果一开始就使用 CLI，会同时遇到 Codex 和终端两套新知识。

如果一开始就使用 IDE 扩展，还需要先熟悉代码编辑器和项目界面。

桌面应用可以先把这些复杂度降下来。第一次使用时，你主要完成三件事：

> 打开 Codex → 选择一个文件夹 → 描述希望得到的成果

等到后面理解文件、终端和 Git 之后，再进入其他入口会更加自然。

2.3 现在应该下载哪个应用

截至 2026 年 7 月，新的 ChatGPT 桌面应用在 Windows 和 macOS 上整合了 Chat、Work 和 Codex。

在桌面应用中：

- 在 ChatGPT 下可以选择 Chat 或 Work；
- Codex 仍然是单独的工作视图；
- Codex 的聊天历史与 ChatGPT 历史分开。

OpenAI 此前的独立 Codex 应用在更新后会转为新的 ChatGPT 桌面应用；旧版 ChatGPT 桌面应用可能继续显示为“ChatGPT Classic”。OpenAI Help Center

因此，第一次安装时应寻找：

> **OpenAI 官方提供的 ChatGPT 桌面应用。**

不要根据旧教程去寻找来源不明的“独立 Codex 安装包”。

2.4 下载前需要准备什么

一台 Windows 或 macOS 电脑

官方桌面应用提供 Windows 和 macOS 版本。具体系统版本、处理器和安装要求可能随产品更新变化，应以官方下载页面当时显示的信息为准。OpenAI Developers

本书默认演示环境是：

> **Windows 电脑＋ChatGPT 桌面应用。**

macOS 读者沿用相同学习路线。只有操作存在明显差异时，正文才会单独说明。

一个可以登录的 ChatGPT 账号

Codex 可以使用 ChatGPT 账号登录，不需要另外注册一个“Codex 账号”。使用额度会因计划和工作空间设置而有所不同。OpenAI Help Center

当前不需要：

- 创建 API 密钥；
- 注册服务器；
- 配置云环境；
- 购买开发者工具。

能够访问 OpenAI 官方服务的网络环境

下载、登录和应用更新需要连接官方服务。

不要从以下位置获取安装包：

- 第三方软件下载站；

- 不明网盘；
- 群聊转发文件；
- 所谓修改版或增强版；
- 无法确认发布方的镜像文件。

2.5 怎样找到正确的下载入口

推荐按照下面的顺序操作：

10. 打开 OpenAI 官方的 ChatGPT 桌面应用或 Codex 快速入门页面；
11. 找到“Download ChatGPT”或相同含义的下载入口；
12. 根据自己的电脑选择 Windows 或 macOS 版本；
13. 检查页面和发布方是否属于 OpenAI 官方体系；
14. 下载完成后暂时不要配置项目，进入第 3 章继续。

官方快速入门当前将完整入门流程概括为：安装桌面应用、登录 ChatGPT 账号、选择工作位置、发送第一条消息。OpenAI Developers

Windows 官方入口通常会进入 Microsoft Store 提供的应用页面；macOS 安装方式以官方页面当前提供的安装文件为准。

2.6 怎样避免下载错误的应用

下载前检查以下信息：

- 页面是否属于 OpenAI 官方站点；
- 应用名称是否为 ChatGPT；
- Windows 应用发布方是否明确对应 OpenAI；
- 文件是否来自官方页面跳转；
- 是否要求你额外安装来源不明的启动器。

如果电脑已经安装过 ChatGPT，暂时不要急于卸载旧版本。

第 3 章会帮助你确认当前打开的是：

- 新版 ChatGPT 桌面应用；
- 还是旧版 ChatGPT Classic。

2.7 目前不需要安装哪些工具

完成本章时，你只需要找到或下载桌面应用。

以下工具全部暂缓：

工具	当前是否需要	后续学习位置
Codex CLI	不需要	第 26 章
VS Code 等代码编辑器	不需要	第 27 章
Codex IDE 扩展	不需要	第 27 章
Git	暂不要求	第 8 章
GitHub 账号	暂不要求	第 20 章
Node.js	不需要	具体项目需要时再处理
Python	不需要	具体项目需要时再处理
WSL	不需要	进阶环境
API 密钥	不需要	自动化和接口开发场景

先完成最小路线，可以避免把应用安装、终端、编程环境和远程仓库等问题同时混在一起。

三、总结

本章完成了第一次入口选择：

- Codex 有桌面应用、CLI、IDE 扩展和云端等使用形态；
- 本书从 ChatGPT 桌面应用中的 Codex 开始；
- 第一次安装应寻找 OpenAI 官方的 ChatGPT 桌面应用；
- 当前只完成下载，不配置 CLI、IDE 和云端；
- 不需要单独注册 Codex 账号或创建 API 密钥；
- 不从第三方来源下载安装程序。

本章掌握检查

- [ ] 我能区分桌面应用、CLI、IDE 扩展和 Codex 云端；
- [ ] 我知道本书为什么先选择桌面应用；
- [ ] 我已经找到适合自己操作系统的官方下载入口；
- [ ] 我没有从第三方网站下载安装文件；
- [ ] 我知道现阶段不需要安装整套开发工具。

产品信息核对日期：2026 年 7 月 31 日。

四、结束语与引导

现在，你已经选定了第一条使用路线，并找到了正确的应用。

下一章将继续完成：

安装应用、登录账号、建立学习目录，并让 Codex 第一次读取一个安全的空白项目文件夹。

第 3 章：安装并配置 Codex，完成第一次运行准备

一、起始语

下载完成后，不建议立即让 Codex 修改正式文件。

更稳妥的顺序是：

> 安装应用 → 登录账号 → 找到 Codex → 建立学习目录 → 只开放一个空白项目 → 完成只读检查

本章结束时，Codex 不会创建网页，也不会修改项目内容。

我们只完成运行环境的准备，并确认：

- 应用能够正常打开；
- 当前账号能够进入 Codex；
- 学习项目与其他文件分开；
- Codex 正在正确的文件夹中工作；
- 第 4 章可以从一个干净的空白项目开始。

二、主体

3.1 安装新版 ChatGPT 桌面应用

Windows

打开第 2 章获得的官方安装入口，按照 Microsoft Store 或系统提示完成安装。

安装完成后，从开始菜单打开 ChatGPT。

如果公司电脑限制 Microsoft Store 或软件安装，不要自行绕过组织的管理策略，应先由管理员确认是否允许安装。

macOS

打开官方下载的安装文件，按照系统提示完成安装。

如果安装窗口要求将应用移入“应用程序”文件夹，完成后从 Finder 的“应用程序”中打开 ChatGPT。

具体系统和处理器要求以官方下载页面当时显示的信息为准。

已经安装过旧版本

如果此前使用独立 Codex 应用，正常更新后，它会转为新的 ChatGPT 桌面应用。

如果此前使用旧版 ChatGPT 桌面应用，新版本可能与旧版本同时存在。此时你可能会看到：

- `ChatGPT`：包含 Chat、Work 和 Codex 的新版应用；
- `ChatGPT Classic`：此前的桌面应用。

本书使用新版 `ChatGPT`。暂时不要求删除 Classic，也不需要进行复杂迁移。OpenAI Help Center

3.2 登录账号并找到 Codex

打开应用后，使用自己的 ChatGPT 账号登录。

本书的桌面入门路线不要求输入 API 密钥。

登录完成后，在应用左上角选择：

> **Codex**

新的桌面应用允许用户在 ChatGPT 和 Codex 之间切换。Codex 拥有单独的工作视图和历史记录，可以打开本地文件夹或项目开展技术任务。OpenAI Help Center

完成四项检查：

检查项	正常结果
应用	当前打开的是新版 ChatGPT
左上角入口	可以选择 ChatGPT 或 Codex
Codex 界面	能够新建聊天或打开文件夹
当前账号	是准备用于本书练习的账号和工作空间

如果同时拥有个人空间和组织工作空间，还应确认当前选择了正确的空间。

3.3 建立本书专用学习目录

打开文件资源管理器或 Finder，在“文档”或“Documents”中建立以下目录：

```
Codex 学习/
└── projects/
    └── ai-learning-page/
```

`ai-learning-page` 目前保持空白。

建立专用目录有三个目的：

15. 避免 Codex 接触无关的个人资料；

16. 让本书练习项目集中在一个位置；
17. 出现问题时，可以清楚判断哪些文件属于当前案例。

不要把学习项目直接放在：

- 存放大量个人文件的桌面；
- 微信或浏览器下载目录；
- 公司正式代码仓库；
- 合同、证件或照片目录；
- 整个系统盘或用户主目录。

3.4 只打开当前项目文件夹

回到 Codex，根据界面选择“打开文件夹”“添加项目”或含义相同的入口。

选择：

`Codex 学习/projects/ai-learning-page`

不要选择整个“文档”文件夹，也不要选择 `Codex 学习`的上级目录。

官方快速入门说明，桌面应用会使用你所选择位置中的文件和上下文，因此应该只开放当前任务真正需要的范围。OpenAI Developers

如果应用请求文件夹访问权限，只允许当前练习项目所需的范围。

3.5 保留项目边界和审批要求

第一次使用时，保持受限的项目边界。

如果界面下方提供权限选项，优先选择：

> **Ask for approval**

在这一模式下，Codex 可以在当前工作区内读取和修改文件、运行常规项目命令；如果需要访问网络或越过工作区边界，会暂停并请求审批。OpenAI Developers

不要为了减少提示，直接选择 Full access。

沙箱负责限制 Codex 可以接触的文件和网络范围，审批规则负责决定什么时候需要停下来获得许可。两者共同构成 Codex 执行真实任务时的基础边界。OpenAI Developers

本阶段不主动启用项目任务之外的扩展能力。以后需要增加权限时，先判断它是否确实是当前任务的必要条件。

3.6 完成第一次只读检查

在当前项目中新建 Codex 聊天，发送：

```
请只检查当前项目文件夹，不要创建、修改、移动或删除任何文件，也不要安装工具或访问网络。

请告诉我：

1. 当前项目文件夹的名称；
2. 文件夹中有哪些文件和子文件夹；
3. 当前文件夹是否为空；
4. 你可以在什么范围内工作；
5. 如果下一步创建一个简单网页，通常需要哪些最基本的文件。

只输出检查结果，不要开始创建网页。
```

这次任务的目标不是产生内容，而是确认 Codex 正在正确的位置工作。

3.7 核对只读检查结果

项目名称是否正确

Codex 应当把当前项目识别为：

`ai-learning-page`

如果它报告了大量其他项目或个人文件，说明打开的文件夹范围过大。

停止当前任务，重新选择正确目录。

项目是否仍然为空

打开文件资源管理器或 Finder，检查 `ai-learning-page`。

本次只读检查不应产生：

- 新文件；
- 新文件夹；
- 依赖安装；
- 网络下载；
- 项目外修改。

如果出现新内容，先停止任务并检查实际变化。由于当前项目原本为空，可以删除误创建内容，恢复为空白目录后重新检查。

权限是否仍然受限

本次检查不需要访问网络，也不需要扩大文件范围。

如果 Codex 提出不必要的权限请求，可以拒绝，并提醒它只完成当前文件夹内的只读检查。

3.8 遇到常见问题时怎样处理

找不到 Codex 入口

检查：

- 是否打开了新版 ChatGPT，而不是 ChatGPT Classic；
- 应用是否完成更新；
- 是否使用正确账号登录；
- 当前工作空间是否允许使用 Codex。

无法登录

检查浏览器中能否登录同一 ChatGPT 账号，以及是否选错个人或组织工作空间。

不要因为登录失败而立即创建 API 密钥。API 密钥不是本章桌面安装路线的必需条件。

找不到项目文件夹

从文件资源管理器或 Finder 重新确认实际位置：

`文档/Codex 学习/projects/ai-learning-page`

不要在应用中重新建立另一个位置不同的同名目录。

应用要求访问更大范围

取消当前选择，重新只打开 `ai-learning-page`。

复杂网络、企业代理和系统兼容故障不在本章展开。遇到这类问题时，先记录实际现象，并按照所在组织的技术支持流程或产品官方说明处理，不让环境问题阻塞后续学习。

三、总结

本章完成了 Codex 第一次运行前的准备：

- 安装并打开了新版 ChatGPT 桌面应用；
- 使用正确的 ChatGPT 账号和工作空间登录；
- 找到了 Codex 视图；
- 建立了专用学习目录；
- 创建了空白项目 `ai-learning-page`；
- 只向 Codex 开放了当前项目文件夹；
- 保留了项目边界和审批要求；
- 完成了一次不修改文件的只读检查。

当前项目状态应当是：

```
Codex 学习/
└── projects/
    └── ai-learning-page/
        └── 空白
```

本章掌握检查

- [ ] 我已经安装并打开新版 ChatGPT 桌面应用；
- [ ] 我使用正确的账号和工作空间登录；
- [ ] 我能够切换到 Codex；
- [ ] 我建立了独立的 `Codex 学习/projects` 目录；
- [ ] `ai-learning-page` 目前保持空白；
- [ ] 我只开放了当前项目文件夹；
- [ ] 我完成了只读检查，没有创建或修改文件。

产品信息核对日期：2026 年 7 月 31 日。

四、结束语与引导

Codex 已经安装完成，也确认了正确的工作位置。

下一章将第一次允许它写入项目文件。

你会从一个空白文件夹开始，完成一个能够在浏览器中打开的 AI 学习导航页，并亲自进行第一次可见修改。

第 4 章：第一次使用 Codex，从打开项目到完成真实成果

一、起始语

当前项目还是空白的：

```
ai-learning-page/
```

本章将产生第一项真实成果：

> **一个可以在浏览器中直接打开的 AI 学习导航页。**

这次任务只使用 HTML 和 CSS，不安装依赖，也不要求你理解代码语法。

你将完成一条最小但完整的使用路线：

> 打开项目 → 提交任务 → 审查文件范围 → 查看真实文件 → 打开页面 → 提出修改 → 人工验收

本章不教学 Git、终端、自动测试或完整任务设计方法。它只帮助你建立第一次成功体验。

二、主体

4.1 确认项目和任务边界

开始前检查：

18. 当前已经进入 Codex；
19. 当前打开的项目是 `ai-learning-page`；
20. 项目文件夹仍然为空；
21. 权限保持在当前项目范围内。

如果显示的是其他项目，重新选择：

`Codex 学习/projects/ai-learning-page`

在当前项目中新建一条聊天，可以命名为：

> 创建 AI 学习导航页

清楚的聊天名称有助于以后区分不同任务，但不会改变项目本身。

4.2 提交第一次创建任务

将下面的内容发送给 Codex：

```
请在当前空白项目中创建一个“AI 学习导航页”的第一版。

页面包含：

- 页面标题：我的 AI 学习导航
- 一段简短说明
- 三张学习卡片：
  1. 提示词练习
  2. 工具体验
  3. 项目记录
- 每张卡片包含一句说明和一个暂不跳转的按钮

技术要求：

- 只使用 HTML 和 CSS
- 创建 index.html 和 css/styles.css
- 不安装任何依赖
- 不访问网络
- 不创建 JavaScript
- 不修改当前项目文件夹以外的内容
- 页面在常见电脑和手机宽度下都能正常阅读

完成后请说明：

1. 创建了哪些文件；
2. 怎样打开页面；
3. 哪些效果尚未在真实浏览器中验证。
```

第 9 章会系统讲解怎样设计一份可执行任务。现阶段先使用这份已经准备好的说明。

4.3 审查执行计划和文件范围

发送任务后，Codex 可能先读取目录、说明计划，也可能请求在项目中写入文件。

此时重点不是判断它的代码写得是否专业，而是确认它有没有超出任务范围。

允许出现的项目结构只有：

```
ai-learning-page/
├── index.html
└── css/
    └── styles.css
```

如果 Codex 准备：

- 安装依赖；
- 创建大量配置文件；
- 使用外部框架；
- 增加 JavaScript；
- 访问网络；

- 修改项目外文件；

先停止或拒绝，再提醒它严格遵守当前任务。

第一次项目越简单，越容易判断结果是否正确。

4.4 找到并打开真实页面

任务结束后，不要只阅读 Codex 的完成说明。

打开文件资源管理器或 Finder，进入 `ai-learning-page`，确认：

- `index.html` 确实存在；
- `css` 文件夹确实存在；
- `styles.css` 位于 `css` 文件夹内；
- 没有出现任务之外的大量文件。

随后双击 `index.html`，使用常用浏览器打开。

页面正常情况下应显示：

- “我的 AI 学习导航”标题；
- 简短说明；
- 三张学习卡片；
- 基本的颜色和布局；
- 在较窄窗口中仍然能够阅读的页面结构。

按钮当前可以没有跳转功能。

这是第一版任务的明确边界，不是程序故障。

4.5 页面不正常时怎样反馈

第一次打开后，页面可能没有完全符合预期。

不要只对 Codex 说：

> 页面不对。

应描述自己实际看到的现象，例如：

- 页面只有文字，没有样式；
- 中文显示异常；
- 卡片挤在一起；
- 手机宽度下内容超出屏幕；
- 双击文件后没有正常打开网页。

统一使用下面的反馈模板：

```
我实际看到的问题是：【描述页面中的具体问题】。

请先检查原因，告诉我问题可能位于哪个文件，再只进行解决这个问题所需的最小修改。

不要增加新功能，不要安装依赖，也不要修改当前项目文件夹以外的内容。
```

“只进行必要修改”是一项重要习惯。

当页面只有一个显示问题时，不应顺便重做整体设计或扩大项目范围。

4.6 完成第一次有限修改

第一版能够正常打开后，在同一条聊天中发送：

```
请只修改现有的 index.html 和 css/styles.css，完成下面的调整：

1. 将标题下方的说明文字改为：
   集中记录我的 AI 学习方向、练习和项目进展。

2. 将“提示词练习”卡片标记为“本周重点”。

3. 让“本周重点”标记清楚但不过度醒目。

保持原有三张卡片和整体布局。

不要增加 JavaScript，不要安装依赖，也不要创建新文件。

完成后请列出实际修改的文件。
```

这一次不再从空白创建项目，而是在已有成果上完成有限调整。

观察 Codex 是否只修改了：

- `index.html`
- `css/styles.css`

如果它准备创建新文件或增加新的页面功能，应要求它回到当前任务。

4.7 刷新页面并完成验收

回到浏览器刷新页面。

Windows 通常使用 `Ctrl + R`，macOS 通常使用 `Command + R`。

检查：

22. 说明文字已经更新；
23. “提示词练习”卡片出现“本周重点”；
24. 其他两张卡片仍然存在；
25. 页面布局没有明显错乱；

26. 缩小浏览器窗口后仍然能够阅读；
27. 项目中没有出现计划外文件。

如果刷新后没有变化，先确认浏览器打开的是当前项目中的 `index.html`，而不是另一个同名文件。

验收完成后，可以让 Codex 只做一次结果汇报：

> 请总结当前项目有哪些文件、每个文件承担什么作用、这次修改了什么，以及哪些结果已经由我在浏览器中人工确认。不要继续修改文件。

一份可靠的汇报应当区分：

- Codex 实际创建或修改了什么；
- 你亲自确认了什么；
- 哪些事项尚未经过自动测试。

本章没有引入测试框架，因此“尚未进行自动测试”是正常状态。

4.8 回看第一次完整闭环

本章经历了一个最小的 Codex 工作闭环：

环节	本次实际内容
目标	创建 AI 学习导航页
工作对象	空白项目文件夹
Codex 动作	创建 HTML 和 CSS 文件
人工检查	在浏览器中打开页面
继续修改	更新说明文字和重点标记
最终验收	刷新页面并检查真实效果

你没有亲自编写 HTML 和 CSS，但已经完成了几项关键工作：

- 选择了正确项目；
- 给出了明确成果要求；
- 限制了文件和技术范围；
- 找到了真实项目文件；
- 打开了真实成果；
- 根据观察反馈问题；
- 完成了一次有限修改；
- 区分了完成说明与实际验收。

这就是零基础使用 Codex 的起点。

三、总结

本章完成了全书第一个可见成果。

当前项目应当包含：

```
ai-learning-page/
├── index.html
└── css/
    └── styles.css
```

页面应当具备：

- “我的 AI 学习导航”标题；
- 一段学习说明；
- 三张学习卡片；
- “本周重点”标记；
- 基本的桌面和手机宽度适配。

本章掌握检查

- [ ] 我能确认 Codex 正在正确项目中工作；
- [ ] 我完成了第一次文件创建任务；
- [ ] 我能够在电脑中找到真实项目文件；
- [ ] 我能够用浏览器打开 `index.html`；
- [ ] 我完成了一次范围有限的后续修改；
- [ ] 我亲自刷新并检查了页面效果；
- [ ] 我能区分 Codex 的完成说明与自己的实际验收；
- [ ] 当前项目中没有安装依赖或增加计划外文件。

产品信息核对日期：2026 年 7 月 31 日。

四、结束语与引导

你已经完成了第一次 Codex 任务，也得到了一个能够真实打开的页面。

接下来，项目不再是空白文件夹。里面出现了文件、文件夹、路径和扩展名。

下一篇将从这个真实项目继续学习：

> **Codex 究竟在操作什么，以及怎样看懂它正在修改的文件。**

第 5 章：文件、文件夹、路径和扩展名

一、起始语

上一章结束时，你已经拥有一个可以在浏览器中打开的“AI 学习导航页”。

当前项目结构是：

```
ai-learning-page/
├── index.html
└── css/
    └── styles.css
```

你已经知道，`index.html` 保存页面内容，`css/styles.css` 控制页面外观。

但随着项目继续增加文件，会出现几个很实际的问题：

- 项目根目录指的是哪里？
- 文件名后面的 `.html` 和 `.css` 有什么作用？
- Codex 说的“路径”究竟是什么？
- 为什么文件明明存在，页面却找不到它？
- 移动一个文件以后，为什么还要修改其他文件？
- 项目中出现同名文件时，怎样告诉 Codex 不要改错？

本章不展开完整的电脑文件管理知识，只解决使用 Codex 时必须理解的四个概念：

> **文件、文件夹、路径和扩展名。**

完成本章后，你应该能够看懂基础目录树，并准确描述 Codex 需要处理的文件位置。

二、主体

5.1 Codex 操作的是电脑中的真实文件

当 Codex 说“已经修改页面”时，最终一定对应某个真实文件发生了变化。

在当前项目中：

对象	类型	作用
ai-learning-page	项目文件夹	保存当前项目的全部内容
index.html	文件	保存页面结构和文字
css	子文件夹	集中保存样式文件

对象	类型	作用
styles.css	文件	保存页面颜色、间距和布局

当前项目的起点是：

`ai-learning-page`

这个起点通常称为**项目根目录**。

它并不是电脑磁盘的最高层，而是当前项目内部所有文件位置的共同参照。

例如：

```
Codex 学习/
└── projects/
    └── ai-learning-page/    ← 当前项目根目录
```

后文出现“在项目根目录中检查文件”或“从项目根目录运行命令”时，指的都是 `ai-learning-page`，不是整个 `Codex 学习` 目录，也不是电脑的系统盘。

5.2 怎样读懂目录树

项目结构经常用目录树表示：

```
ai-learning-page/
├── index.html
└── css/
    └── styles.css
```

阅读目录树时，主要观察层级关系：

- `index.html` 直接位于项目根目录；
- `css` 也是项目根目录下的内容；
- `styles.css` 位于 `css` 文件夹内部。

因此，`styles.css` 相对于项目根目录的位置是：

`css/styles.css`

项目简单时，只说“修改样式文件”可能不会出错。

项目变大后，可能同时存在：

```
css/styles.css
archive/styles.css
examples/styles.css
```

此时如果只说“修改 `styles.css`”，Codex 仍然需要猜测你的目标。

更加准确的表达是：

> 修改当前项目中的 `css/styles.css`，不要修改其他同名文件。

你也可以让 Codex 先只读检查项目结构：

```
请只读取当前项目，不要创建、修改、移动或删除文件。

请输出：

1. 当前项目的目录树；
2. 哪个文件位于项目根目录；
3. 哪个文件负责页面样式；
4. 每个文件相对于项目根目录的位置；
5. 是否存在同名文件。

请根据实际项目回答，不要根据常见网页结构推测。
```

Codex 输出目录后，还要与文件资源管理器或 Finder 中的真实项目进行核对。

5.3 文件名和扩展名

一个完整文件名通常由两部分组成：

> 文件主体名称＋扩展名

例如：

完整文件名	主体名称	扩展名
index.html	index	.html
styles.css	styles	.css
app.js	app	.js
README.md	README	.md
learning-mark.svg	learning-mark	.svg

扩展名可以帮助电脑和应用判断文件类型。

本书常见的扩展名包括：

扩展名	常见用途
.html	网页内容和结构
.css	网页样式
.js	网页交互逻辑
.md	项目说明文档
.json	结构化数据或配置
.txt	普通文本

扩展名	常见用途
.png、.jpg、.svg	图片和图标

扩展名不能随意忽略。

例如，文件显示为 `index.html`，实际名称却是：

`index.html.txt`

它仍然是文本文件，不是正常的 HTML 页面。

在 Windows 文件资源管理器中，建议启用“文件扩展名”显示。macOS 读者也应通过 Finder 查看完整文件名，避免扩展名被隐藏。

5.4 绝对路径和相对路径

路径用于说明文件在哪里。

绝对路径

绝对路径从电脑中的固定位置开始，表示完整地址。

Windows 示例：

```
C:\Users\你的用户名\Documents\Codex 学习\projects\ai-learning-page\index.html
```

macOS 示例：

```
/Users/你的用户名/Documents/Codex 学习/projects/ai-learning-page/index.html
```

绝对路径位置明确，但通常只适用于当前电脑。

换一个用户名、磁盘或目录，绝对路径就可能变化。

相对路径

相对路径从某个参照位置开始计算。

必须注意：

> **相对路径没有唯一固定的起点，它从哪里开始计算，要看这条路径出现在哪里。**

常见情况包括：

路径出现的位置	通常以什么为参照
HTML 中的资源引用	当前 HTML 文件所在位置
CSS 中的图片或字体引用	当前 CSS 文件所在位置

路径出现的位置	通常以什么为参照
终端中的相对路径	当前工作目录
项目说明中的文件位置	通常以项目根目录为参照

当前项目中的 `index.html` 位于项目根目录，因此它引用样式文件时可以写：

```
<link rel="stylesheet" href="css/styles.css">
```

这里的 `css/styles.css` 从 `index.html` 所在的位置开始计算。

由于 `index.html` 恰好位于项目根目录，所以这条路径看起来也同时是相对于项目根目录的。

但如果 HTML 文件被移动到其他子目录，引用路径也可能需要变化。

5.5 文件引用为什么会失效

假设 `index.html` 中写着：

```
<link rel="stylesheet" href="css/styles.css">
```

浏览器会根据这条路径寻找：

```
ai-learning-page/
└── css/
    └── styles.css
```

如果把 `styles.css` 移动到项目根目录，却没有修改 HTML 引用，浏览器仍然会前往 `css` 文件夹寻找它。

这时可能出现：

- 页面仍然能够打开；
- 页面文字仍然存在；
- 颜色、布局和卡片样式消失。

问题不一定出在 CSS 内容本身，也可能是：

> 文件的真实位置与引用路径已经不一致。

判断路径问题时，需要同时核对：

28. 文件真实位置；
29. 文件完整名称；
30. 引用中写入的路径；
31. 引用从哪个文件开始计算；
32. 路径中的层级和大小写是否准确。

5.6 创建、移动并修复图标引用

接下来为导航页增加一个本地图标，并通过移动文件观察路径变化。

阶段一：在项目根目录创建图标

发送：

```
请先检查当前项目结构，然后完成下面的有限修改：

1. 在项目根目录创建一个简单的本地 SVG 图标，文件名为 learning-mark.svg；
2. 在 index.html 的页面标题附近显示这个图标；
3. 不访问网络，不使用外部图片；
4. 不安装第三方依赖；
5. 除 index.html 和新建图标外，不修改其他文件。

完成后请说明：
- 创建和修改了哪些文件；
- index.html 使用什么路径引用图标；
- 哪些结果需要我在浏览器中确认。
```

按照步骤执行成功后，项目应暂时变为：

```
ai-learning-page/
├── index.html
├── learning-mark.svg
└── css/
    └── styles.css
```

刷新页面，确认图标是否能够显示。

若没有显示，应先检查：

- 文件是否真实存在；
- 文件名是否一致；
- `index.html` 中的引用是否正确；
- 浏览器打开的是否为当前项目页面。

阶段二：移动图标并同步更新引用

确认根目录中的图标可以正常显示后，继续发送：

```
请整理刚才创建的图标文件：

1. 在项目中建立 assets/icons 文件夹；
2. 将根目录中的 learning-mark.svg 移动到 assets/icons；
3. 更新 index.html 中的图标引用；
4. 不改变图标内容和页面其他结构；
5. 不创建重复文件。

完成后请列出新的项目目录，并说明更新了哪一条引用。
```

执行成功后，项目结构应为：

```
ai-learning-page/
├── index.html
├── css/
│   └── styles.css
└── assets/
    └── icons/
        └── learning-mark.svg
```

`index.html` 中的图标引用应相应变为：

`assets/icons/learning-mark.svg`

刷新页面后，如果图标仍能显示，说明：

- 文件移动成功；
- 原位置没有残留重复文件；
- HTML 引用已经同步更新；
- 新的相对路径可以正常工作。

如果实际结果不同，应停止继续修改，并先核对目录和引用。

5.7 怎样准确描述目标文件

一个清楚的文件操作说明，通常包含：

- 当前项目；
- 文件相对路径；
- 准备执行的动作；
- 是否允许改名或移动；
- 哪些引用需要同步更新；
- 哪些文件明确不能修改；
- 修改后怎样验证。

不够准确：

> 修改图标。

更加准确：

> 修改当前项目中的 `assets/icons/learning-mark.svg`，不要修改其他图片资源。

不够准确：

> 把文件整理一下。

更加准确：

> 将项目根目录中的 `learning-mark.svg` 移动到 `assets/icons/learning-mark.svg`，同步更新 `index.html` 中的引用，并确认浏览器中图标仍能显示。

你暂时不需要每次写出全部细节，但至少要能够指出：

> 目标文件相对于项目根目录在哪里。

三、总结

本章建立了理解 Codex 文件操作所需的基础能力。

你已经知道：

- 项目根目录是当前项目的起点；
- 目录树用于表示文件之间的层级关系；
- 扩展名帮助电脑识别文件类型；
- 绝对路径表示电脑中的完整位置；
- 相对路径的参照点取决于使用场景；
- 移动或改名后，相关引用通常需要同步更新；
- 描述目标文件时，应优先给出清楚的相对位置。

按照本章步骤完成后，项目结构应为：

```
ai-learning-page/
├── index.html
├── css/
│   └── styles.css
└── assets/
    └── icons/
        └── learning-mark.svg
```

本章掌握检查

- [ ] 我能指出当前项目根目录；
- [ ] 我能看懂基础目录树；
- [ ] 我知道文件主体名称与扩展名的区别；
- [ ] 我能区分绝对路径与相对路径；
- [ ] 我知道不同场景中的相对路径参照点可能不同；
- [ ] 我能准确描述 `assets/icons/learning-mark.svg` 的位置；
- [ ] 移动图标后，我已经检查页面是否仍能显示它。

四、结束语与引导

现在，你已经能够看懂项目中的文件和位置。

但当 Codex 运行命令时，它不仅需要知道某个文件在哪里，还必须知道：

> **当前究竟站在哪个文件夹中执行操作。**

下一章将进入终端和工作目录，帮助你理解 Codex 为什么运行命令，以及命令输出在说明什么。

第 6 章：终端、命令和工作目录

一、起始语

目前，你主要通过文件资源管理器或 Finder 观察项目。

这类图形界面适合：

- 打开文件夹；
- 双击文件；
- 移动内容；
- 查看目录结构。

Codex 还会使用另一种电脑操作入口：

> **终端。**

终端看起来像一个用于输入文字的窗口。它可以通过命令查看目录、读取文件、启动程序和运行检查。

零基础读者不需要把终端学成一套完整课程。

现阶段只需能够判断三件事：

33. 命令在哪个目录中运行；
34. 命令准备完成什么动作；
35. 输出表示成功、失败，还是仍需检查。

本章以 Windows PowerShell 为主要环境，macOS 只提供必要对照。

二、主体

6.1 终端是什么

可以把不同工具理解为不同的电脑操作入口：

工具	更适合完成什么
文件资源管理器或 Finder	点击、浏览、复制和移动文件
浏览器	打开网页并检查页面效果
编辑器	查看和编辑文件内容
终端	用文字命令让电脑执行操作
Codex	根据任务调用文件、终端和其他工具推进工作

一条终端命令通常包含：

> 命令名称＋操作对象或参数

例如，PowerShell 中的：

```
Get-ChildItem
```

表示列出当前目录中的内容。

```
Get-Content .\index.html -TotalCount 12
```

表示读取当前目录下 `index.html` 的前 12 行。

你不需要背诵大量命令，只需要能够看懂 Codex 为什么使用它们。

6.2 什么是当前工作目录

终端始终在某个文件夹中工作。

这个位置称为：

> **当前工作目录。**

假设当前工作目录是：

```
C:\Users\你的用户名\Documents\Codex 学习\projects\ai-learning-page
```

此时运行 `Get-ChildItem`，看到的会是 `ai-learning-page` 中的内容。

如果当前工作目录是它的上一级：

```
C:\Users\你的用户名\Documents\Codex 学习\projects
```

运行同样的命令，看到的可能只有：

`ai-learning-page`

而不是项目内部的 `index.html`、`css` 和 `assets`。

因此，一条命令即使写法正确，在错误目录中执行，结果仍可能不符合任务要求。

常见后果包括：

- 找不到目标文件；
- 读取另一个项目；
- 在错误位置创建内容；
- 把 Git 仓库建立在项目上级目录；

- 根据不完整的文件得出错误判断。

第 5 章中的网页相对路径，通常由引用文件的位置决定；本章中的终端相对路径，则由**当前工作目录**决定。

6.3 Windows 中需要认识的五类操作

查看当前位置

```
Get-Location
```

正常情况下，输出中应出现当前目录的完整路径。

查看当前目录内容

```
Get-ChildItem
```

如果当前位于项目根目录，应能够看到：

- `index.html`
- `css`
- `assets`

进入目标目录

将下面路径替换为你电脑中的真实位置：

```
Set-Location "C:\Users\你的用户名\Documents\Codex 学习\projects\ai-learning-page"
```

路径中存在中文或空格时，使用引号包住完整路径。

读取少量文件内容

```
Get-Content .\index.html -TotalCount 12
```

这条命令只读取文件开头，不会修改文件。

停止持续运行的命令

可以运行一个等待命令：

```
Start-Sleep -Seconds 30
```

在命令结束前按 `Ctrl + C`，可以中断当前进程。

停止当前命令不等于关闭 Codex，也不等于删除项目文件。

6.4 macOS 命令对照

macOS 终端中的对应操作如下：

目的	Windows PowerShell	macOS
查看当前位置	`Get-Location`	`pwd`
查看当前内容	`Get-ChildItem`	`ls`
进入目标目录	`Set-Location "路径"`	`cd "路径"`
读取文件开头	`Get-Content`	`head`
停止持续命令	`Ctrl + C`	`Control + C`

macOS 读者不需要同时记住两套命令，只要理解它们解决的是同一类问题。

后续正文仍以 PowerShell 为主。

6.5 怎样理解命令输出

命令运行后，通常会出现三种情况。

出现正常内容

例如，运行`Get-ChildItem`后列出了项目文件。

这说明命令已经执行，但还不能立刻得出“任务成功”的结论。

你仍然需要判断：

- 当前目录是否正确；
- 文件是否属于目标项目；
- 输出是否满足当前任务；
- 是否出现计划外内容。

没有输出

没有输出不一定表示失败。

有些检查命令在没有发现问题时可能不显示内容。

判断时需要结合：

- 命令本来应该完成什么；
- 是否出现错误文字；
- 终端是否已经回到可继续输入的状态；
- 项目是否发生了预期变化。

出现错误信息

例如：

```
Cannot find path ...
```

可能表示：

- 路径输入错误；
- 文件不存在；
- 当前工作目录不正确；
- 文件名或扩展名不一致。

遇到错误时，不要只告诉 Codex“命令报错了”。

应同时提供：

- 实际执行的命令；
- 完整错误信息；
- 当前工作目录；
- 预期文件位置；
- 原本希望得到的结果。

6.6 让 Codex 完成一次只读目录检查

在当前项目中发送：

```
请只使用只读命令检查当前工作目录，不要创建、修改、移动或删除文件。

请完成：

1. 输出当前工作目录；
2. 列出项目根目录中的内容；
3. 判断这里是否是 ai-learning-page 的项目根目录；
4. 说明判断依据；
5. 列出实际执行的命令及关键结果。

如果当前目录不正确，只说明问题，不要自动切换目录。
```

可靠的判断至少应基于以下证据：

- 当前目录名称是 `ai-learning-page`；
- 根目录中存在 `index.html`；
- 存在 `css/styles.css`；
- 存在 `assets/icons/learning-mark.svg`；
- 没有把上一级 `projects` 误认为项目根目录。

如果当前目录不正确，可以继续询问：

```
当前目录不是目标项目。

请只告诉我：

1. 当前目录是什么；
2. 目标项目的真实路径是什么；
3. 应使用什么命令进入目标项目。

不要执行其他操作。
```

确认路径后，再决定是否切换目录。

6.7 开始任务前的最小检查顺序

后续当 Codex 需要运行命令时，可以先检查：

```
Get-Location
Get-ChildItem
```

这两条命令分别回答：

- 现在在哪里；
- 当前目录中有什么。

它们不能保证所有任务都不会出错，但可以避免很多最基础的位置错误。

当命令可能持续运行时，还应确认：

- 怎样判断它已经启动；
- 怎样查看输出；
- 怎样使用 `Ctrl + C` 停止；
- 停止后是否需要重新检查项目状态。

三、总结

本章没有要求你背诵大量命令，而是建立了理解终端操作所需的最低能力。

你已经知道：

- 终端是电脑的一种文字操作入口；
- 命令总是在当前工作目录中执行；
- 同一条命令在不同目录中可能得到不同结果；
- `Get-Location` 用于确认位置；
- `Get-ChildItem` 用于查看当前内容；
- `Set-Location` 用于进入目标目录；
- `Get-Content` 可以只读查看文件内容；
- `Ctrl + C` 可以停止正在运行的命令；

- 命令执行结束不等于任务已经验收通过。

本章掌握检查

- [ ] 我能说明终端与文件资源管理器的区别；
- [ ] 我能查看当前工作目录；
- [ ] 我能列出当前目录中的文件；
- [ ] 我能判断终端是否位于项目根目录；
- [ ] 我能读懂基本的正常输出和错误信息；
- [ ] 我知道怎样停止持续运行的命令；
- [ ] 我知道终端相对路径以当前工作目录为参照。

四、结束语与引导

现在，你已经能够看懂 Codex 在什么位置运行命令。

但项目并不只是若干文件放在同一个目录中。不同文件承担不同职责，有些项目可以直接打开，有些项目则需要安装工具并运行命令。

下一章将继续使用 AI 学习导航页，认识：

> **HTML、CSS 和 JavaScript 怎样组成一个项目，以及怎样判断项目真正的运行方式。**

第 7 章：代码、项目、依赖和运行方式

一、起始语

当前 AI 学习导航页包含：

```
ai-learning-page/
├── index.html
├── css/
│   └── styles.css
└── assets/
    └── icons/
        └── learning-mark.svg
```

这些文件共同构成了一个可以在浏览器中打开的静态页面。

接下来需要理解：

- 哪个文件是项目入口？
- HTML 和 CSS 为什么分开？
- 点击按钮后的行为应该写在哪里？
- 什么是第三方依赖？
- 为什么当前项目可以直接打开，有些项目却必须运行命令？
- 面对陌生项目时，怎样判断正确运行方式？

本章不要求你独立编写代码，也不展开框架、构建工具和复杂运行环境。

目标只有一个：

> **看懂基础项目由什么组成，并知道 Codex 正在修改哪一部分。**

二、主体

7.1 文件不等于项目

一个代码文件可以独立存在，但一个项目通常由多个相互配合的部分组成。

常见内容包括：

项目组成	可能包含什么
入口文件	项目从哪里开始
页面或程序代码	主要内容和功能
样式文件	外观和布局

项目组成	可能包含什么
交互逻辑	点击、输入和状态变化
资源文件	图标、图片和数据
配置文件	工具和运行设置
依赖说明	需要安装哪些第三方代码包
测试	检查功能是否符合预期
项目文档	说明用途、运行方式和注意事项

当前导航页比较简单，只包含其中一部分。

即使项目规模很小，也不能只查看 `index.html` 就判断所有功能。

7.2 HTML、CSS 和 JavaScript 分别负责什么

HTML：页面上有什么

`index.html` 通常负责：

- 标题；
- 文字；
- 卡片；
- 按钮；
- 图片；
- 页面结构。

CSS：页面看起来怎样

`css/styles.css` 通常负责：

- 颜色；
- 字体；
- 间距；
- 对齐；
- 卡片布局；
- 窄屏适配。

JavaScript：操作后发生什么

JavaScript 通常负责：

- 点击按钮；
- 输入搜索内容；

- 筛选卡片；
- 更新页面状态；
- 保存本地数据；
- 根据操作显示或隐藏内容。

可以先用一句话记住：

> **HTML 负责内容，CSS 负责外观，JavaScript 负责行为。**

真实项目会比这个划分复杂，但它足以帮助你理解当前案例。

7.3 入口文件如何连接其他内容

当前项目的入口文件是：

`index.html`

因为双击它以后，浏览器会从这里开始读取页面。

在 `index.html` 中，会继续引用其他文件。

样式引用可能是：

```
<link rel="stylesheet" href="css/styles.css">
```

图标引用可能是：

```
<img src="assets/icons/learning-mark.svg" alt="">
```

增加 JavaScript 后，还会出现脚本引用：

```
<script src="js/app.js"></script>
```

入口文件并不是项目中唯一重要的文件。

它更像一个开始位置，将页面结构、样式、脚本和资源连接起来。

如果入口文件中的引用路径错误，即使其他文件内容正确，页面也可能无法正常工作。

7.4 什么是第三方依赖

第三方依赖可以理解为：

> 项目需要使用、但不是由当前项目从头编写的外部代码包或工具。

例如，复杂项目可能使用：

- 界面组件库；
- 测试工具；

- 构建工具；
- 数据处理包；
- Web 框架。

这类项目通常会包含依赖说明文件，例如：

- `package.json`
- `requirements.txt`
- `pyproject.toml`

看到这些文件并不代表项目有问题，只说明项目可能需要先准备运行环境或安装代码包。

当前 AI 学习导航页：

- 需要浏览器运行；
- 使用本地 HTML、CSS 和图标；
- 没有需要额外安装的第三方依赖；
- 没有构建步骤；
- 没有本地服务器。

因此，可以直接双击 `index.html` 打开。

不要因为看到网页项目，就自动运行 `npm install`。

应先检查：

36. 项目中是否存在依赖说明文件；
37. README 或项目文档说明了什么；
38. 当前项目是否已经可以运行；
39. 当前任务是否真的需要新增外部工具。

7.5 项目可能有哪些运行方式

直接打开

当前导航页属于静态网页。

运行方式是：

> 双击 `index.html`，使用浏览器打开。

通过命令启动

有些项目需要运行类似 `npm run dev` 或 `python app.py` 的命令。

这类项目可能包含：

- 第三方依赖；

- 本地服务器；
- 构建步骤；
- 后端程序。

实际命令应以项目配置和说明为准，不能只根据编程语言猜测。

使用专用运行环境

更复杂的项目可能依赖其他运行环境。

本书会在具体案例出现时再解释，当前不需要提前安装或学习。

面对陌生项目时，正确顺序是：

> **先检查结构和说明，再决定怎样运行。**

不要先安装大量工具，再反过来猜测项目需要什么。

7.6 让 Codex 解释当前项目

在增加新功能前，可以先发送：

```
请只读取当前 ai-learning-page 项目，不要修改文件。

请用表格说明：

1. 项目入口文件是什么；
2. HTML、CSS 和资源文件分别在哪里；
3. 每个文件承担什么作用；
4. 文件之间通过什么路径互相引用；
5. 当前是否存在需要额外安装的第三方依赖；
6. 当前项目应该怎样运行；
7. 如果增加按钮交互，最适合新增或修改哪些文件。

请根据实际文件回答，不要假设项目中存在尚未创建的配置。
```

一份可靠的说明应指出：

- 入口是 `index.html`；
- 样式位于 `css/styles.css`；
- 图标位于 `assets/icons/learning-mark.svg`；
- 当前没有 JavaScript 文件；
- 当前没有需要安装的第三方依赖；
- 页面可以直接在浏览器中打开。

7.7 为导航页增加第一个交互

接下来增加一个简单功能：

> 点击“只看本周重点”后隐藏其他卡片，再次点击时恢复全部卡片。

发送：

```
请在当前 AI 学习导航页中增加一个简单交互。

功能要求：

1. 在学习卡片上方增加“只看本周重点”按钮；
2. 点击后，只显示标记为“本周重点”的卡片；
3. 按钮文字变为“显示全部”；
4. 再次点击后，恢复全部卡片；
5. 创建 js/app.js 保存交互逻辑；
6. 在 index.html 中正确引用 js/app.js；
7. 如有必要，可以最小修改 index.html 和 css/styles.css。

边界要求：

- 不安装第三方依赖；
- 不使用外部框架；
- 不访问网络；
- 不修改项目外内容。

完成后请说明：
- 创建和修改了哪些文件；
- JavaScript 怎样被页面加载；
- 我需要在浏览器中验证哪些操作。
```

执行成功后，项目应变为：

```
ai-learning-page/
├── index.html
├── css/
│   └── styles.css
├── js/
│   └── app.js
└── assets/
    └── icons/
        └── learning-mark.svg
```

7.8 检查功能和运行方式

刷新浏览器后，检查：

40. 页面仍然可以正常打开；
41. 出现“只看本周重点”按钮；
42. 点击后，只保留重点卡片；
43. 按钮文字变为“显示全部”；
44. 再次点击后恢复全部卡片；
45. 原有图标、样式和内容没有被破坏。

增加 JavaScript 后，项目的运行方式应当没有变化。

判断依据是：

- `js/app.js` 是本地静态文件；

- `index.html` 可以直接引用它；
- 没有需要安装的第三方依赖；
- 没有本地服务器；
- 没有构建步骤。

如果页面不能直接运行，应先检查文件引用和实际项目结构，而不是立刻安装新的工具。

三、总结

本章建立了对基础项目组成的认识。

按照步骤完成后，当前项目中：

文件	主要职责
index.html	页面入口和内容结构
css/styles.css	页面样式和布局
js/app.js	按钮交互
assets/icons/learning-mark.svg	本地图标资源

你已经知道：

- 项目通常由多个文件共同组成；
- HTML、CSS 和 JavaScript 承担不同职责；
- 入口文件连接样式、脚本和资源；
- 第三方依赖是需要额外使用的外部代码包或工具；
- 当前项目没有需要额外安装的第三方依赖；
- 项目运行方式应根据真实文件和说明判断；
- 增加 JavaScript 不一定会改变项目运行方式。

本章掌握检查

- [ ] 我能指出当前项目入口；
- [ ] 我能区分 HTML、CSS 和 JavaScript 的职责；
- [ ] 我知道资源文件怎样被入口文件引用；
- [ ] 我能解释什么是第三方依赖；
- [ ] 我知道当前项目为什么不需要安装代码包；
- [ ] 我已经检查“只看本周重点”交互；
- [ ] 我能说出当前项目的正确运行方式。

四、结束语与引导

导航页已经从静态页面变成了带有简单交互的小项目。

但现在，每次修改都会直接改变现有文件。

如果 Codex 改错了，或者新方案不如原来的版本，只让它“凭记忆改回去”并不可靠。

下一章将建立项目中最重要的基础保护能力：

> **查看 Codex 到底改了什么，保存可靠版本，并在结果不满意时恢复。**

第 8 章：看懂修改、保存版本与恢复结果

一、起始语

当前项目已经具备：

- 页面结构；
- 页面样式；
- 本地图标；
- 简单交互；
- 明确的运行方式。

项目结构是：

```
ai-learning-page/
├── index.html
├── css/
│   └── styles.css
├── js/
│   └── app.js
└── assets/
    └── icons/
        └── learning-mark.svg
```

随着任务继续推进，会出现三个现实问题：

46. Codex 究竟修改了哪些文件？
47. 当前结果验收后，怎样保存为可靠版本？
48. 新修改不满意时，怎样恢复到修改前？

本章使用 Git 解决这些问题。

目标不是一次掌握完整 Git，而是建立四项基础能力：

> **查看状态 → 检查差异 → 保存版本 → 恢复未提交修改**

本章只在本地记录版本，不使用 GitHub，不创建远程仓库，也不学习复杂分支和冲突处理。

二、主体

8.1 Git 解决什么问题

Git 是一种版本控制工具。

它可以帮助你：

- 查看哪些文件被修改；
- 比较修改前后的内容；
- 保存一个可靠版本；
- 查看版本历史；
- 恢复尚未提交的错误修改；
- 为后续协作保留清楚记录。

Git 不等于 GitHub。

名称	当前可以怎样理解
Git	在当前电脑中管理项目版本
GitHub	托管远程代码仓库并开展协作的平台

本章只使用 Git。

执行 `git init` 不会自动把项目上传到互联网。

8.2 在正确目录中检查 Git

打开 PowerShell，先确认当前位置和项目内容：

```
Get-Location
Get-ChildItem
```

当前目录应为 `ai-learning-page`，并能看到：

- `index.html`
- `css`
- `js`
- `assets`

随后检查 Git 是否可用：

```
git --version
```

如果显示 Git 版本信息，可以继续。

如果系统提示无法识别 `git`，应从 Git 官方来源完成安装，重新打开 PowerShell 后再次检查。复杂安装问题不在本章展开。

不要在尚未确认目录时运行 `git init`，否则可能把仓库建立在错误位置。

8.3 初始化仓库并理解三种检查

确认当前位于项目根目录后，运行：

```
git init
git status
```

Git 会在项目中建立一个隐藏的`.git`目录，用于保存版本记录。

不要手工修改`.git`中的内容。

第一次查看状态时，项目文件通常会显示为“未跟踪”。

“未跟踪”表示：

> 文件已经真实存在，但 Git 还没有开始记录它们。

本章会用到三种检查：

命令	用途
git status	查看文件当前处于什么状态
git diff	查看尚未暂存的实际变化
git diff --staged	查看已经准备进入下一次提交的变化

可以把它们理解为：

> 现在有哪些变化 → 变化具体是什么 → 哪些变化准备保存

8.4 创建基础版本

当前项目已经经过前几章的人工检查，可以将它保存为基础版本。

设置当前仓库的提交者信息

第一次提交时，Git 可能要求设置名称和邮箱。

将尖括号中的内容替换为你自己的提交信息，不要直接照抄占位文字：

```
git config user.name "<你的提交名称>"
git config user.email "<你的提交邮箱>"
```

这两项信息只用于当前仓库的提交记录，不是 ChatGPT 登录，也不是 GitHub 登录操作。

明确选择要保存的文件

将当前项目文件加入暂存区：

```
git add index.html css/styles.css js/app.js assets/icons/learning-mark.svg
```

随后检查：

```
git status
git diff --staged
```

确认以下事项：

- 准备提交的文件都属于当前项目；
- 没有陌生文件；
- 没有项目外内容；
- 当前文件内容符合已经验收的页面状态。

确认无误后创建基础提交：

```
git commit -m "建立 AI 学习导航页基础版本"
```

提交成功后再次运行 `git status`。

如果输出表示工作区干净，说明当前没有未提交修改。

如果实际结果不同，应先停止后续操作，查看错误或剩余文件状态。

8.5 查看一次真实修改

接下来让 Codex 尝试一个页面进度展示方案：

```
请在当前 AI 学习导航页中增加学习进度展示。

要求：

1. 在页面说明下方显示“已完成 1 项，共 3 项”；
2. 增加一条可见的进度条；
3. 只修改 index.html 和 css/styles.css；
4. 不修改 JavaScript；
5. 不创建新文件；
6. 不安装第三方依赖；
7. 不执行 Git 提交。

完成后只汇报实际修改内容。
```

Codex 完成修改后，先不要提交。

运行：

```
git status
git diff
```

`git status` 应帮助你确认哪些文件发生了变化。

`git diff` 会展示修改前后的实际差异，其中通常：

- `-`表示删除的内容；
- `+`表示新增的内容。

你不需要立即读懂全部 HTML 和 CSS，但至少要确认：

- 修改文件是否只有 `index.html` 和 `css/styles.css`；
- 是否删除了计划外内容；
- 是否增加了任务之外的功能；
- 修改范围是否与要求一致。

随后在浏览器中刷新页面，判断新方案是否值得保留。

8.6 放弃不满意的未提交修改

假设你认为：

- 进度条过于醒目；
- 页面显得拥挤；
- 简单文字已经足够；
- 当前方案不值得保存。

在恢复前，先再次运行：

```
git diff
```

确认这些未提交修改确实不再需要。

随后只恢复本次修改涉及的两个文件：

```
git restore index.html css/styles.css
```

这条命令会使用最近一次提交中的版本，覆盖这两个文件当前尚未提交的修改。

执行后再次运行：

```
git status
```

如果工作区恢复干净，说明本次未提交修改已经被放弃。

刷新页面后，刚才增加的进度条应不再显示。

需要注意：

> `git restore` 会丢弃指定文件中尚未提交的修改。

因此，执行前必须先检查 `git diff`，确认这些内容不再需要。

本章不处理已经提交版本的回退。

8.7 创建第二个正式版本

现在采用一个更简单的方案：

```
请在当前 AI 学习导航页中增加一个简洁的学习进度提示。

要求：

1. 在页面说明下方增加文字“当前进度：1/3”；
2. 文字清楚但不过度醒目；
3. 只修改 index.html 和 css/styles.css；
4. 不增加进度条；
5. 不修改 JavaScript；
6. 不创建新文件；
7. 不安装第三方依赖；
8. 不执行 Git 提交。

完成后请列出实际修改的文件。
```

完成后，在浏览器中检查：

- 显示“当前进度：1/3”；
- 页面布局仍然正常；
- 筛选按钮仍然可用；
- 图标仍然显示；
- 没有出现计划外文件。

确认实际页面符合预期后，运行：

```
git status
git diff
```

检查无误后，将两个文件加入暂存区：

```
git add index.html css/styles.css
git diff --staged
```

确认暂存区只包含本次有限修改，再创建提交：

```
git commit -m "增加学习进度提示"
```

如果提交成功，再运行 `git status` 确认是否还有未提交内容。

8.8 查看版本历史和最终状态

运行：

```
git log --oneline
```

按照本章步骤执行成功后，历史中应出现两个提交，形式类似：

```
a1b2c3d 增加学习进度提示
e4f5g6h 建立 AI 学习导航页基础版本
```

左侧字符由 Git 自动生成，每台电脑上的具体内容不同。

这两个版本可以理解为：

```
基础版本
建立 AI 学习导航页基础版本
        ↓
第二个版本
增加学习进度提示
```

最后再次运行：

```
git status
```

如果工作区干净，说明当前项目中的真实文件与最近一次提交一致。

也可以让 Codex 进行一次只读检查：

```
请只读取当前项目和 Git 状态，不要修改文件，也不要创建提交。

请告诉我：

1. 当前是否为 Git 仓库；
2. 工作区是否干净；
3. 最近两次提交分别是什么；
4. 当前页面包含哪些主要文件；
5. 是否存在未跟踪或未提交内容；
6. 本章的 git restore 操作是否改变了提交历史。

请列出实际使用的只读命令。
```

可靠的回答应区分：

- 实际提交历史；
- 当前工作区状态；
- 被放弃的未提交修改；
- 已经保存的正式版本。

三、总结

本章建立了最基础的版本保护能力。

按照步骤执行成功后，你应当完成：

49. 确认 Git 在正确项目中运行；
50. 初始化本地 Git 仓库；
51. 使用 `git status` 查看文件状态；
52. 使用明确文件列表创建基础提交；
53. 通过 `git diff` 查看 Codex 的实际修改；
54. 使用 `git restore` 放弃不满意的未提交方案；
55. 完成一次更小范围的修改；

56. 使用 `git diff --staged` 检查准备保存的内容；
57. 创建第二个提交；
58. 使用 `git log --oneline` 查看历史。

版本历史中应包含：

- `建立 AI 学习导航页基础版本`
- `增加学习进度提示`

本章掌握检查

- [ ] 我知道 Git 与 GitHub 不是同一件事；
- [ ] 我能确认 Git 仓库建立在正确目录中；
- [ ] 我能使用 `git status` 查看文件状态；
- [ ] 我能使用 `git diff` 查看未提交修改；
- [ ] 我知道提交前还要检查 `git diff --staged`；
- [ ] 我知道提交者名称和邮箱不能直接照抄占位符；
- [ ] 我使用 `git restore` 前先检查了差异；
- [ ] 我能通过 Git 历史确认是否形成两个版本；
- [ ] 我已经检查当前工作区是否干净。

四、结束语与引导

到这里，你已经能够看懂 Codex 操作的文件、运行命令的位置、项目的基本组成，以及修改前后的真实差异。

AI 学习导航页也具备了本地版本保护能力。

下一篇将不再只学习“Codex 正在做什么”，而是进入更关键的任务设计：

> **怎样把一句模糊的“帮我加个搜索”，变成范围清楚、能够执行和验收的正式任务。**

第 9 章：把模糊想法变成 Codex 能够执行的任务

一、起始语

前八章中，你已经完成了一个可以在浏览器中运行的 AI 学习导航页，并使用 Git 保存了两个可靠版本。

当前项目包含：

```
ai-learning-page/
├── index.html
├── css/
│   └── styles.css
├── js/
│   └── app.js
└── assets/
    └── icons/
        └── learning-mark.svg
```

页面已经具备：

- 三张学习卡片；
- “本周重点”标记；
- “只看本周重点”筛选；
- 学习进度提示；
- 两个本地 Git 提交。

接下来，你希望继续改进这个页面。

最初的想法可能只有一句：

> 帮我加个搜索。

这句话说明了大致方向，却没有说明搜索哪些内容、怎样与现有筛选配合、哪些文件可以修改，以及什么结果才算完成。

如果这些问题没有说清楚，Codex 只能自行补充假设。

本章不修改项目，而是把这句模糊需求整理成一项：

> **范围清楚、能够执行、可以验收的搜索功能任务。**

二、主体

9.1 为什么模糊想法不能直接成为执行任务

模糊想法通常只描述了方向，没有定义任务边界。

例如：

> 页面上的资料越来越多，帮我优化一下查找体验。

Codex 可能把它理解为：

- 增加关键词搜索；
- 增加分类筛选；
- 重新排列卡片；
- 增加标签系统；
- 重做整个页面；
- 将资料迁移到数据库；
- 接入在线搜索。

这些方案都可能有价值，却不是同一项任务。

当 Codex 需要自行猜测时，容易出现三类偏差。

结果偏差

页面出现了搜索框，但搜索的内容不是你真正需要的对象。

范围偏差

你只要求增加搜索，它却同时重构了页面结构和 JavaScript。

验收偏差

Codex 报告“搜索已经完成”，你却不知道应该怎样验证。

因此，一项能够执行的任务，至少要回答：

> 做什么、为什么做、允许改什么、哪些不做、需要交付什么、怎样判断完成。

9.2 基础任务卡的六个要素

本书对小型 Codex 任务统一使用六个要素。

要素	需要回答的问题
目标	最终需要产生什么变化
背景	当前项目是什么状态，为什么需要修改
范围	本次允许实现哪些功能、处理哪些对象
非目标	哪些看似相关的事情本次明确不做
输出	完成后需要交付哪些文件和说明
验收	人通过什么操作判断任务完成

这六项不要求写成大型需求文档。

对于当前搜索功能，一页任务卡已经足够。

OpenAI 当前的 Codex 实践指南也建议在任务中明确目标、相关上下文、约束条件和完成标准，使 Codex 更容易判断应该做什么、哪些内容不能改变，以及何时才算完成。OpenAI Developers

9.3 写清楚目标和背景

原始表达是：

> 帮我加个搜索。

可以先将目标改写为：

> 为 AI 学习导航页增加本地关键词搜索，使使用者能够通过卡片标题和说明快速找到学习内容。

目标描述的是最终变化。

背景则说明这项变化为什么需要发生：

> 当前页面已有三张学习卡片、“本周重点”标记、“只看本周重点”筛选和学习进度提示。随着卡片数量增加，只依靠浏览和重点筛选会降低查找效率，因此需要增加一个不依赖网络的本地搜索功能。

背景能够告诉 Codex：

- 项目目前已经具备什么；
- 哪些已有功能必须保留；
- 这次修改要解决什么真实问题；
- 新功能需要与什么现有行为配合。

背景越准确，Codex 越不容易将已有成果当成可以随意替换的内容。

9.4 用范围和非目标控制任务扩张

本次范围

搜索功能需要完成：

- 在卡片区域上方增加搜索输入框；
- 搜索卡片标题和说明；
- 输入关键词时实时更新可见卡片；
- 忽略关键词前后的空格；
- 英文关键词不区分大小写；
- 没有匹配结果时显示清楚的提示；
- 与“只看本周重点”筛选同时工作。

本次非目标

这次明确不做：

- 不连接在线搜索或外部数据源；
- 不安装第三方依赖；
- 不增加数据库；
- 不增加新的分类或标签系统；
- 不重做整个页面；
- 不删除或改写原有卡片内容；
- 不修改当前项目之外的文件；
- 验收完成前不创建 Git 提交。

非目标并不是否定这些能力的价值，而是在说明：

> 哪些合理想法不属于当前任务，需要留到以后单独评估。

9.5 将输出和验收写成可检查结果

“搜索可以使用”并不是足够明确的验收标准。

它可能只代表：

- 页面中出现了输入框；
- 输入一次关键词有反应；
- Codex 没有继续报告错误。

预期输出

功能实施阶段预计修改：

- `index.html`
- `css/styles.css`
- `js/app.js`

完成人工验收后，还需要创建：

- `notes/search-validation.md`

验证文档用于记录实际测试结果、已知限制和最终结论。

Codex 完成任务时还应说明：

- 实际修改或创建了哪些文件；
- 每个文件发生了什么变化；
- 运行了哪些检查；
- 哪些结果需要人工在浏览器中确认；
- 是否存在尚未验证的事项。

验收标准

搜索功能需要满足：

59. 页面显示搜索输入框；
60. 可以根据卡片标题匹配；
61. 可以根据卡片说明匹配；
62. 首尾空格和英文大小写不影响搜索；
63. 没有结果时显示提示；
64. 清空关键词后恢复卡片；
65. 原有重点筛选仍然正常；
66. 搜索与重点筛选可以组合使用；
67. 页面原有内容和布局没有明显破坏；
68. 没有新增第三方依赖或任务外文件。

这些标准都能通过浏览器操作、文件差异和 Git 状态进行检查。

9.6 形成完整搜索功能任务卡

将前面的内容组合起来，可以得到下面这份任务卡。

```
任务名称：
为 AI 学习导航页增加本地搜索功能

目标：
增加本地关键词搜索，使使用者能够通过卡片标题和说明快速找到学习内容。

背景：
当前项目使用原生 HTML、CSS 和 JavaScript，已有三张学习卡片、“本周重点”标记、“只看本周重点”筛选和学习进度提示。项目通过 index.html 直接运行，不需要安装第三方依赖。

范围：
1. 在卡片区域上方增加搜索输入框；
2. 搜索卡片标题和说明；
3. 输入时实时更新可见卡片；
4. 忽略关键词首尾空格，英文不区分大小写；
5. 无匹配结果时显示提示；
6. 搜索与“只看本周重点”筛选同时生效；
7. 清空关键词后恢复符合当前筛选状态的卡片。

非目标：
1. 不访问网络或连接在线搜索；
2. 不安装第三方依赖；
3. 不增加数据库、分类或标签系统；
4. 不重构整个页面；
5. 不删除或改写原有卡片内容；
6. 不修改项目外文件；
7. 验收完成前不创建 Git 提交。

预期输出：
1. 功能实施阶段只修改 index.html、css/styles.css 和 js/app.js；
2. 人工验收完成后创建 notes/search-validation.md；
3. 汇报实际文件变化、检查结果和未验证事项；
4. 最终将三个功能文件和验证记录保存为同一个 Git 提交。

验收标准：
1. 标题和说明关键词均可匹配；
2. 首尾空格和英文大小写不影响结果；
3. 无结果状态清楚；
4. 清空搜索后恢复卡片；
5. 原有重点筛选功能正常；
6. 搜索与重点筛选能够组合使用；
7. 原有页面内容和布局没有明显破坏；
8. 没有新增第三方依赖或任务外文件。
```

这份任务卡已经明确“要做什么”，但还没有证明应该修改哪些具体位置。

因此，它不能替代下一章的项目探索。

9.7 判断哪些问题需要确认

Codex 开始探索后，可能仍会遇到不确定事项。

会改变业务结果的问题，应先确认

例如：

- 搜索对象是标题，还是标题和说明；
- 搜索与重点筛选是同时生效，还是互相替换；
- 无结果时是否显示提示；
- 是否允许调整现有卡片结构。

这些决定会直接改变使用者看到的结果。

不改变目标的小型实现细节，可以在计划中说明

例如：

- 搜索框使用什么内部变量名；
- 无结果元素放在卡片区域前还是后；
- JavaScript 采用哪种简单字符串匹配方法；
- CSS 类名如何沿用现有命名方式。

即使这类问题可以由 Codex 处理，也应在实施计划中说明，不能通过大范围重写静默解决。

按照本章要求，项目文件和 Git 状态不应发生变化。本章的唯一成果是经过确认的搜索任务卡。

三、总结

本章把一句模糊需求：

> 帮我加个搜索。

转换成了一份具有六个要素的任务卡：

- 目标；
- 背景；
- 范围；
- 非目标；
- 输出；
- 验收。

一项好的 Codex 任务，不只是告诉它要增加什么功能，还要明确：

- 当前项目已经有什么；
- 本次允许改变什么；
- 哪些内容暂时不做；
- 最终交付哪些文件和记录；
- 人怎样判断任务是否完成。

本章掌握检查

- [ ] 我能区分模糊想法和可执行任务；
- [ ] 我知道任务卡的六个要素；
- [ ] 我能为任务划定范围和非目标；
- [ ] 我能将“能用”改写成可操作的验收标准；
- [ ] 验证记录已经纳入任务输出；
- [ ] 我知道任务卡不能替代项目探索；
- [ ] 本章不应修改项目文件或 Git 状态。

四、结束语与引导

现在，我们已经回答了：

> **要做什么。**

但还没有回答：

> **应该修改哪些文件，怎样修改，可能影响什么。**

下一章不会立即写入代码。

Codex 将先只读查看项目，用真实文件证据形成一份能够审查的实施计划。

第 10 章：先探索和计划，再决定是否修改

一、起始语

第 9 章已经形成搜索功能任务卡。

目标、范围和验收标准都已经明确，但我们仍然不知道：

- 搜索框应该放在页面的哪个位置；
- 卡片标题和说明在 HTML 中怎样组织；
- 现有重点筛选逻辑位于哪里；
- 新搜索状态怎样与原有筛选状态组合；
- 是否确实只需要修改三个功能文件；
- 当前代码中有没有需要先处理的问题。

如果直接开始实施，Codex 仍然需要边修改边猜测项目结构。

当前任务涉及三个关联文件和一个已有筛选功能，因此更稳妥的顺序是：

> **检查基线 → 只读探索 → 形成计划 → 人工批准。**

本章结束时，项目文件和 Git 历史都不应发生变化。

二、主体

10.1 什么时候需要探索和计划

不是所有任务都要先写计划。

任务情况	更合适的方式
目标明确，只改一句文字	可以直接执行
知道要改什么，但不知道代码位置	先探索
涉及多个关联文件和已有功能	先探索并形成短计划
可能改变依赖、数据或项目结构	先计划并获得批准
任务目标本身仍然模糊	返回任务设计，不进入实施

例如，将页面标题中的一个词替换掉，通常不需要单独写计划。

当前搜索功能会同时影响：

- 搜索输入区域；
- 卡片显示状态；
- 无结果提示；
- JavaScript 筛选逻辑；
- 原有重点筛选。

因此，需要先了解真实项目结构。

10.2 检查当前项目基线

开始探索前，先确认：

- 当前打开的是 `ai-learning-page`；
- 第 8 章的两个提交仍然存在；
- 工作区中没有来源不明的修改。

在项目根目录运行：

```
git status
```

如果工作区干净，可以继续。

如果存在未提交修改或未跟踪文件，应先查明：

- 变化来自哪一章；
- 是否已经验收；
- 是否应该提交或恢复；
- 是否属于当前搜索任务。

不要把旧修改与搜索功能混在同一批差异中。

10.3 第一步：只读探索项目

在当前项目中发送：

```
请根据已经确认的搜索功能任务卡，对当前项目进行只读探索。

本阶段不要创建、修改、移动或删除文件，不要安装依赖，不要访问网络，也不要创建 Git 提交。

请调查：

1. 项目的入口文件和运行方式；
```

```
2. 学习卡片在 index.html 中的实际结构；
3. 卡片标题、说明和“本周重点”如何表示；
4. “只看本周重点”逻辑位于哪个文件；
5. 当前筛选状态和事件监听怎样工作；
6. 搜索功能最可能涉及哪些文件；
7. 搜索与重点筛选组合时可能出现什么冲突；
8. 是否需要新增功能文件或第三方依赖；
9. 当前是否存在会阻碍实施的问题。

每项结论都要给出文件路径和对应代码位置。

最后只输出探索结果和待确认问题，不要开始实施。
```

一份可信的探索结果不能只写：

> 搜索功能需要修改 HTML、CSS 和 JavaScript。

它还需要解释：

- 卡片容器为什么位于 `index.html`；
- 重点筛选状态为什么位于 `js/app.js`；
- 搜索区域为什么需要补充样式；
- 为什么不需要新增依赖；
- 哪些判断仍然存在不确定性。

探索的目标不是复制整个项目，而是找到完成任务所需的相关位置。

10.4 第二步：根据证据形成最小计划

只读探索完成后，继续要求 Codex 形成实施计划：

```
请根据搜索功能任务卡和刚才的只读探索结果，形成一份最小实施计划。

计划需要说明：

1. 实施步骤和先后顺序；
2. 每一步涉及的文件；
3. 每个文件准备修改什么；
4. 搜索状态与重点筛选状态怎样组合；
5. 哪些文件不需要修改；
6. 主要风险和暂停条件；
7. 每一步怎样检查；
8. 人工验收完成后怎样创建 notes/search-validation.md；
9. 最终提交应包含哪些文件；
10. 是否还有需要我决定的问题。

只输出计划，不要修改项目。
```

计划应当分成三个阶段。

功能实施

修改：

- `index.html`
- `css/styles.css`
- `js/app.js`

人工验收后

创建：

- `notes/search-validation.md`

验证记录不应在功能尚未测试时提前生成。

最终保存

将三个功能文件和验证记录放入同一个搜索功能提交。

10.5 审查计划是否越过任务边界

人工审查计划时，重点检查以下内容。

修改范围是否有证据

计划中的每个文件都应来自只读探索。

如果突然出现：

- 数据库；
- 外部搜索接口；
- 新框架；
- 大量配置文件；
- 与搜索无关的资源；

需要 Codex 说明必要性，否则应当删除。

是否保护原有功能

计划需要明确检查：

- 原有三张卡片；
- 本地图标；
- 学习进度；
- “本周重点”标记；

- 重点筛选；
- 窄屏页面布局。

两种筛选状态是否组合

统一规则是：

> 卡片可见 = 符合搜索条件，并且符合当前重点筛选状态。

搜索和重点筛选不能分别覆盖对方的显示结果。

是否存在停止条件

发现以下情况时，应暂停实施：

- 实际项目与探索结论不一致；
- 必须修改任务卡之外的业务文件；
- 需要安装第三方依赖；
- 需要访问网络；
- 原有卡片结构需要大规模重写；
- 关键业务规则仍未确认。

10.6 第三步：批准或退回计划

计划满足以下条件后，才可以进入第 11 章：

- 目标与第 9 章一致；
- 修改文件均有真实证据；
- 功能实施只涉及三个必要文件；
- 验证记录安排在人工测试之后；
- 不需要联网或安装第三方依赖；
- 原有筛选功能有保护方案；
- 两种筛选状态的组合规则清楚；
- 验证方法能够对应任务卡；
- 没有未解决的关键问题；
- Git 工作区仍然干净。

如果计划不满足这些条件，应退回修改，而不是边执行边扩大范围。

本章的重点不在于使用某个固定界面按钮，而在于建立一条稳定方法：

> **先调查真实项目，再批准具体改法。**

三、总结

本章没有修改项目，而是完成了三步准备：

69. 检查当前 Git 基线；
70. 只读探索真实文件和已有逻辑；
71. 形成并审查最小实施计划。

你已经知道：

- 小型、明确的改动不一定需要计划；
- 涉及多个文件和已有功能时，先探索更可靠；
- 探索结论需要真实文件证据；
- 计划需要说明修改顺序、文件、风险和验证方式；
- 验证记录应在人工测试完成后创建；
- 计划未经批准，不进入实施。

本章掌握检查

- [ ] 我能判断任务是否需要探索和计划；
- [ ] 探索前工作区应当保持干净；
- [ ] 探索过程不应修改项目；
- [ ] 关键结论都有文件和代码位置证据；
- [ ] 计划说明了三个功能文件的修改方式；
- [ ] 验证文档已经纳入计划；
- [ ] 我知道哪些情况必须暂停实施；
- [ ] 批准计划后才能进入下一章。

四、结束语与引导

到这里，我们已经完成了搜索功能的两项前置工作：

> 第 9 章：明确要做什么
> 第 10 章：明确准备怎样做

下一章将正式进入项目执行。

你会通过同一项任务观察 Codex 怎样读取文件、搜索位置、修改内容并运行检查。

第 11 章：读文件、搜内容、改文件和运行命令

一、起始语

搜索任务已经具备实施条件：

- 目标和范围明确；
- 非目标已经确认；
- Codex 完成了只读探索；
- 实施计划已经通过；
- Git 工作区保持干净。

本章将按照已批准的计划，为 AI 学习导航页增加搜索功能。

在这个过程中，你会观察四类基础动作：

> **读取文件 → 搜索内容 → 修改文件 → 运行检查**

本章只产生搜索功能候选实现。

正式测试、验证记录和 Git 提交放在第 12 章。

二、主体

11.1 四类动作分别解决什么问题

动作	解决的问题	当前案例中的作用
读取文件	已知需要查看哪个文件	了解 HTML、样式和现有交互
搜索内容	知道要找什么，但不知道具体位置	定位卡片、重点标记和事件逻辑
修改文件	将已批准的方案写入项目	增加搜索界面、样式和筛选逻辑
运行命令	检查项目状态与实际差异	确认修改范围和基本格式问题

已经知道目标文件时，可以直接读取相关区域。

只知道功能名称、按钮文字或类名时，先搜索定位，再读取必要上下文。

这比无差别读取和重写整个项目更容易控制范围。

11.2 实施前确认基线

开始执行前，在项目根目录运行：

```
git status
git log --oneline -2
```

如果前面的步骤执行成功：

- 工作区应当干净；
- 最近两个提交应为导航页基础版本和学习进度版本。

实际状态与预期不一致时，应先停止并查明原因。

不要把来源不明的旧修改混入本次搜索功能。

11.3 按批准计划开始实施

在已经包含任务卡和实施计划的 Codex 聊天中发送：

```
请按照已经批准的搜索功能任务卡和实施计划开始执行。

要求：

1. 先重新确认相关文件和代码位置；
2. 功能实施只修改 index.html、css/styles.css 和 js/app.js；
3. 搜索卡片标题和说明；
4. 忽略关键词首尾空格，英文不区分大小写；
5. 空关键词显示符合当前重点筛选状态的卡片；
6. 无匹配结果时显示提示；
7. 搜索与“只看本周重点”筛选同时生效；
8. 保留原有卡片、图标、学习进度和页面结构；
9. 不安装第三方依赖；
10. 不访问网络；
11. 不创建验证记录；
12. 不创建 Git 提交。

如果实际项目与计划不一致，或者必须扩大修改范围，请先停止并说明原因。

完成后请汇报：
- 实际读取和搜索了什么；
- 修改了哪些文件；
- 运行了哪些检查；
- 是否偏离原计划；
- 哪些结果仍需人工验证。
```

本章明确不创建 `notes/search-validation.md`。

验证文档只有在第 12 章完成真实测试后才能产生。

11.4 检查三个文件分别发生了什么

如果实施遵守计划，修改通常集中在三个文件中。

index.html

可能增加：

- 搜索输入框；
- 与输入框关联的标签或提示；
- 无结果信息区域。

不应：

- 删除已有卡片；
- 改写原有卡片内容；
- 引入外部脚本；
- 重建整个页面结构。

css/styles.css

可能增加：

- 搜索区域布局；
- 输入框样式；
- 无结果提示样式；
- 窄屏显示规则。

不应：

- 替换整个视觉体系；
- 删除原有卡片样式；
- 产生大量与搜索无关的格式变化。

js/app.js

需要将两种状态统一处理：

```
卡片可见
=
符合搜索条件
并且
符合当前重点筛选状态
```

如果搜索和重点筛选分别直接控制卡片显示，容易出现：

- 搜索后重点筛选失效；
- 点击重点筛选后搜索状态丢失；

- 清空关键词后显示状态错误。

因此，更可靠的做法是让两种状态进入同一个显示判断流程。

11.5 用命令检查实际变化

修改完成后，运行：

```
git status --short
git diff --check
```

如果实施严格遵守计划，`git status --short`通常只应显示：

```
M index.html
M css/styles.css
M js/app.js
```

如果出现：

- 新验证文档；
- 新配置文件；
- 新依赖文件；
- 任务外资源；

应暂停并检查来源。

`git diff --check`主要用于检查新增差异中的尾随空格、空格与制表符等常见空白符问题。

它不能判断搜索逻辑是否正确。

继续查看具体差异：

```
git diff -- index.html css/styles.css js/app.js
```

重点检查：

- 是否只修改三个目标文件；
- 是否存在大段无关重写；
- 原有内容是否被意外删除；
- 是否增加外部网络引用；
- 是否引入第三方依赖；
- 实际改法是否符合批准计划。

11.6 完成第一次浏览器试运行

打开或刷新 `index.html`，快速检查：

72. 搜索框正常显示；
73. 输入卡片标题关键词后，显示结果发生变化；
74. 输入不存在的内容时出现无结果提示；
75. 清空关键词后恢复卡片；
76. 原有重点筛选仍能操作。

这一步只用于发现明显问题，不是第 12 章的正式验收。

若发现问题，使用统一的最小修复模板：

```
我在浏览器中实际看到的问题是：【描述具体问题】。

请根据当前文件和现有差异定位原因，只进行解决该问题所需的最小修改。

不要增加新功能，不要安装依赖，不要扩大文件范围，不要创建验证记录，也不要创建 Git 提交。

完成后请说明：
1. 原因是什么；
2. 修改了哪个文件；
3. 为什么这项修改能解决问题；
4. 我需要重新验证什么。
```

问题修复后，再次检查差异和相关页面行为。

11.7 锁定本章结束状态

本章结束时，应满足：

- 搜索功能已经写入三个功能文件；
- 没有创建验证文档；
- 没有安装第三方依赖；
- 没有访问项目外文件；
- 没有创建 Git 提交；
- 浏览器只完成了初步试运行；
- 所有未验证事项已经被列出。

当前 Git 状态通常应为：

```
M index.html
M css/styles.css
M js/app.js
```

此时只能称为：

> **搜索功能候选实现。**

不能称为稳定版本或正式交付。

三、总结

本章通过同一项搜索任务，观察了 Codex 的四类基础动作：

- 读取已知文件；
- 搜索未知位置；
- 修改必要文件；
- 运行命令检查状态和差异。

按照计划执行成功后，项目中应有三个尚未提交的功能文件变化。

本章掌握检查

- [ ] 实施前工作区应当保持干净；
- [ ] Codex 先确认了相关文件和代码位置；
- [ ] 我能区分读取文件与搜索内容；
- [ ] 实际修改没有超出三个功能文件；
- [ ] 没有安装第三方依赖或创建验证文档；
- [ ] 我检查了 Git 状态和实际差异；
- [ ] 我在浏览器中完成了第一次试运行；
- [ ] 当前结果仍是候选实现，没有创建 Git 提交。

四、结束语与引导

搜索功能已经写入项目，但“文件已经改完”和“功能已经验收”并不是同一件事。

下一章将把任务卡中的完成标准转化为实际测试，检查原有功能、执行边界和验证证据，再决定是否保存为第三个稳定版本。

第 12 章：验证结果、检查证据和控制权限

一、起始语

第 11 章结束时，搜索功能已经形成候选实现。

当前 Git 工作区通常包含三个尚未提交的修改文件：

- `index.html`
- `css/styles.css`
- `js/app.js`

Codex 可能会报告：

> 搜索功能已经完成，相关检查通过。

但这句话不能单独证明功能可以交付。

真正的验收需要回答四个问题：

77. 实际修改了什么；
78. 功能是否符合任务卡；
79. 有没有破坏原有行为或越过批准边界；
80. 是否可以保存为稳定版本。

本章将完成 AI 学习导航页的正式验收、验证记录和第三次 Git 提交，并在此后封存这一案例。

二、主体

12.1 不同证据能够证明什么

一项功能是否正确，通常不能只依靠一种证据。

证据	可以证明什么	不能单独证明什么
Codex 完成说明	它认为自己执行了什么	页面行为一定正确
Git 状态和差异	哪些文件真实发生了变化	功能一定符合需求
静态检查	是否存在部分格式问题	浏览器交互一定正确
浏览器测试	用户实际看到的行为	所有未测试情况都安全
回归检查	原有功能是否仍然可用	项目外范围一定未被访问

证据	可以证明什么	不能单独证明什么
执行审计	是否出现联网、安装或越界操作	搜索逻辑本身一定正确

可靠验收需要将这些证据组合起来。

12.2 先检查文件和差异

在项目根目录运行：

```
git status --short
git diff --check
git diff --stat
```

重点确认：

- 修改文件是否仍然只有三个；
- 是否出现未跟踪文件；
- 是否存在任务范围外变化；
- 差异规模是否与小型搜索功能相符；
- 是否有常见空白符问题。

随后查看具体差异：

```
git diff -- index.html css/styles.css js/app.js
```

检查：

HTML

- 是否只增加必要的搜索区域；
- 原有卡片是否仍然存在；
- 无结果提示文字是否清楚。

CSS

- 是否只增加搜索相关样式；
- 是否保留原有页面规则；
- 是否存在与任务无关的大规模变化。

JavaScript

- 是否处理标题和说明；
- 是否去除关键词首尾空格；

- 英文比较是否统一大小写；
- 搜索和重点筛选是否进入同一个判断流程；
- 无结果提示是否随当前显示结果变化。

也可以使用 Codex 界面的差异区域辅助查看，但最终仍应以真实 Git 状态和人工验收为准。Codex 当前的 Review 界面可以按文件或修改片段查看、暂存和放弃 Git 差异。OpenAI Developers

12.3 将验收标准转化为八组测试

打开项目中的 `index.html`，依次执行下表。

编号	测试内容	预期结果
T01	页面初始状态	空关键词时显示全部卡片
T02	标题关键词	输入“提示词”等标题内容时匹配对应卡片
T03	说明关键词	输入只出现在说明中的内容时也能匹配
T04	空格与英文大小写	首尾空格被忽略，英文字母大小写不影响结果
T05	无匹配结果	隐藏卡片并显示清楚的无结果提示
T06	清空关键词	恢复符合当前筛选状态的卡片
T07	原有重点筛选	不使用搜索时，重点筛选仍然正常
T08	搜索与重点筛选组合	两种条件组合、切换和清除后结果正确

记录时不要只写“正常”。

应写明：

- 实际输入了什么；
- 页面显示了哪些卡片；
- 无结果提示是否出现；
- 当前重点筛选是什么状态；
- 实际结果与预期是否一致。

未执行的测试不能写成“通过”。

12.4 检查原有功能是否回归

搜索功能可以使用，并不代表整个页面已经通过验收。

还需要检查原有成果：

- 页面标题和学习说明仍然存在；
- 本地图标仍然显示；
- 学习进度仍为预期内容；
- 三张卡片的文字没有被改写；
- “本周重点”标记仍然存在；
- 窄屏状态下页面仍然可以阅读；
- 浏览器控制台没有明显脚本错误。

这一步称为回归检查：

> 新功能完成后，重新确认原来可用的功能没有被破坏。

若搜索正常，但重点筛选、页面布局或卡片内容被破坏，任务仍然不能通过。

12.5 检查执行边界

本次任务批准的范围是：

- 读取当前项目文件；
- 修改三个功能文件；
- 运行本地项目检查命令；
- 在浏览器中人工验证；
- 验收后创建一份验证记录；
- 不联网；
- 不安装第三方依赖；
- 不接触项目外文件。

检查以下事项：

81. 是否访问过网络；
82. 是否安装过第三方依赖；
83. 是否访问或修改过项目外文件；
84. 是否出现未批准的权限请求；
85. 是否创建了任务卡之外的文件；
86. 是否在验收完成前创建了提交。

沙箱决定 Codex 在技术上能够接触哪些文件和网络资源，审批策略则决定它在跨越当前边界前是否需要暂停并获得许可。两者属于不同但相互配合的控制层。OpenAI Developers

本章只要求核对本次任务有没有越过边界，不展开系统级权限配置、密钥或企业管理策略。

12.6 处理失败并形成验证记录

出现小范围问题

当问题现象清楚、原因能够定位，而且仍然只涉及三个批准文件时，可以让 Codex 进行最小修复。

修复后重新执行相关测试和必要回归项。

修改范围严重失控

如果出现：

- 大量无关重写；
- 任务外文件；
- 未批准依赖；
- 项目外修改；
- 搜索与原有筛选逻辑难以解释；

应停止继续修补。

先检查 `git diff`，再按照第 8 章的方法决定是否恢复三个功能文件。

创建验证记录

只有完成真实测试后，才发送：

```
我已经完成搜索功能的人工验收。

以下是我真实执行的测试与回归结果：
【粘贴 T01—T08 和原有功能检查的真实结果】

请创建 notes/search-validation.md，记录：

1. 功能目标；
2. 实际修改文件；
3. 静态检查结果；
4. T01—T08 的实际结果；
5. 原有功能回归结果；
6. 执行边界检查；
7. 已知限制；
8. 尚未验证事项；
9. 最终验收结论。

不得虚构没有执行的测试，也不得把未知结果写成通过。

除创建这份验证记录外，不要修改其他文件，也不要创建 Git 提交。
```

创建后打开文档核对：

- 是否与真实测试一致；

- 是否如实保留失败、未知和未执行事项；
- 是否区分静态检查与浏览器验证；
- 是否出现未经证实的结论。

12.7 保存稳定版本并封存案例

只有以下条件全部满足后，才进入提交：

- T01—T08 已经真实执行；
- 失败项已经修复并重新验证；
- 原有功能回归通过；
- 实际修改范围符合计划；
- 没有安装第三方依赖；
- 没有项目外修改；
- 验证记录与真实结果一致；
- 当前差异已经人工审查。

将四个文件加入暂存区：

```
git add index.html css/styles.css js/app.js notes/search-validation.md
```

检查暂存内容：

```
git status
git diff --staged
```

确认暂存区只包含搜索功能和验证记录后，创建提交：

```
git commit -m "增加学习导航搜索功能"
```

最后检查：

```
git status
git log --oneline -3
```

按照步骤执行成功后，项目历史中应出现：

```
增加学习导航搜索功能
增加学习进度提示
建立 AI 学习导航页基础版本
```

工作区应当恢复干净。

最终项目结构为：

```
ai-learning-page/
├── index.html
├── css/
│   └── styles.css
├── js/
│   └── app.js
├── assets/
│   └── icons/
│       └── learning-mark.svg
└── notes/
    └── search-validation.md
```

到这里，AI 学习导航页正式封存。

后续章节可以引用本项目中学到的方法，但不再为它增加新功能。

三、总结

本章完成了 Codex 基础工作闭环的最后一步：

> 目标 → 探索 → 计划 → 修改 → 检查 → 人工验证 → 边界核对 → 保存版本

你已经知道：

- Codex 的完成说明不等于任务正确；
- 不同证据能够证明的内容不同；
- 验收标准需要转化为实际测试；
- 新功能还要检查原有行为是否回归；
- 文件、命令、网络和权限范围需要接受核对；
- 未实际执行的测试不能记录为通过；
- 只有经过验收的结果才适合创建提交。

本章掌握检查

- [ ] 我检查了真实文件状态和差异；
- [ ] 我完成了 T01—T08 的人工测试；
- [ ] 我检查了原有功能是否回归；
- [ ] 我核对了网络、依赖和文件范围；
- [ ] 我没有把未执行测试写成通过；
- [ ] 验证记录与真实结果一致；
- [ ] 我审查了暂存区内容；
- [ ] 搜索功能形成第三个稳定提交。

四、结束语与引导

第三篇到这里已经完成。

AI 学习导航页也完成了它在本书中的使命：

- 帮助你第一次创建真实成果；
- 帮助你理解文件、路径和项目；
- 帮助你认识终端和版本控制；
- 帮助你将模糊想法转化为正式任务；
- 帮助你完成探索、计划、执行和验证闭环。

从下一章开始，我们将建立一个新的项目：

> **个人资料与任务工作台**

这一次，你将从真实问题出发，自己确定使用者、核心功能、第一版范围和验收标准。

第 13 章：从真实问题形成项目需求和第一版范围

一、起始语

前 12 章中，我们围绕“AI 学习导航页”完成了一套 Codex 基础工作闭环：

> 明确任务 → 探索项目 → 制定计划 → 修改文件 → 验证结果 → 保存版本

这个项目帮助你理解了文件、终端、项目结构、Git、任务范围和人工验收。

但导航页中的卡片内容主要预先写在代码中。使用者可以搜索和筛选，却不能真正新增自己的资料和任务。

从本章开始，我们创建第一个能够持续接收并保存个人数据的项目：

> **个人资料与任务工作台**

它将用于管理：

- 待阅读的资料；
- 需要完成的任务；
- 内容分类；
- 当前处理状态；
- 搜索和筛选结果；
- 保存在当前浏览器环境中的记录。

这个项目贯穿第 13—16 章。

章节	项目推进
第 13 章	确定真实问题、记录字段和第一版范围
第 14 章	创建能够新增、显示和保存记录的 MVP
第 15 章	增加搜索、分类筛选和状态交互
第 16 章	调试持久化故障、完成回归和 v1 交付

本章暂时不创建页面代码。

我们先决定：

> 这个项目为谁解决什么问题，第一版必须完成什么，又明确不做什么。

二、主体

13.1 从真实问题开始，而不是从功能清单开始

个人资料和任务经常分散在不同位置：

- 浏览器收藏夹中保存着待阅读文章；
- 聊天工具里留着需要处理的信息；
- 文档中记录着学习计划；
- 便签里写着临时任务；
- 已经处理的内容没有统一状态；
- 需要重新查找时，很难迅速找到。

真正的问题不是“缺少一个网页”，而是：

> **个人资料和待办事项分散，缺少一个能够集中记录、持续更新并在下次打开后继续使用的轻量工作台。**

若一开始只对 Codex 说：

> 帮我做一个资料管理系统。

它可能自行增加：

- 用户注册；
- 数据库；
- 云端同步；
- 文件上传；
- 多人协作；
- 复杂标签；
- 提醒通知；
- 权限管理。

这些能力可能适合成熟产品，却不适合零基础读者的第一个完整项目。

所以，第一版先收束为：

> 在一台电脑的指定浏览器环境中，集中记录个人资料和任务，并通过类型、分类和状态了解下一步应该处理什么。

13.2 确定主要使用者和核心场景

第一版只服务一类使用者：

在自己的电脑上管理个人资料和任务的单个用户。

它不是团队任务系统，也不是企业知识库。

第一版覆盖四个核心场景。

保存一条资料

例如：

- 一篇待阅读文章；
- 一份行业报告；
- 一个 AI 工具说明；
- 一条学习资料。

用户填写标题、分类、状态和备注，保存后记录出现在列表中。

创建一项任务

例如：

- 整理下周会议案例；
- 阅读一份报告；
- 完成一项学习练习；
- 准备一篇文章。

资料和任务使用同一套记录结构，通过“类型”字段区分。

查看处理进度

用户能够看到：

- 全部记录数；
- 待处理数量；
- 进行中数量；
- 已完成数量。

查找并更新记录

随着记录增加，用户需要：

- 按关键词搜索；
- 按分类筛选；
- 修改处理状态。

搜索、筛选和状态更新放在第 15 章实施。第 14 章的 MVP 先解决新增、展示和保存。

13.3 确定完整用户流程

第一版的主要流程是：

```
打开工作台
→ 查看已有记录或空状态
→ 填写新增记录表单
→ 保存记录
→ 在列表中查看记录
→ 刷新或重新打开页面
→ 继续看到已保存的记录
```

第 15 章再扩展：

```
输入关键词
或选择分类
→ 找到目标记录
→ 更新记录状态
```

完整流程不仅要考虑正常状态，还要考虑异常状态。

例如：

- 没有任何记录时，页面应显示初始空状态；
- 标题为空时，不应保存无意义记录；
- 已有本地数据无法解析时，不应静默覆盖原数据；
- 本地写入失败时，页面不能假装保存成功；
- 有记录但筛选无结果时，应显示筛选空结果，而不是初始空状态。

当前不需要设计全部异常细节，但需求中必须明确最基本的保护原则。

13.4 为每条记录确定必要字段

第一版使用七个字段。

字段	用途	产生方式
id	唯一识别一条记录	程序生成
type	区分资料和任务	用户选择
title	记录的核心名称	用户填写
category	对记录进行分类	用户选择
status	表示当前处理阶段	用户选择
note	保存补充说明	用户选填

字段	用途	产生方式
createdAt	记录创建时间	程序生成

一条记录的数据结构可以表示为：

```
{
  "id": "rec-001",
  "type": "资料",
  "title": "企业 AI 应用趋势报告",
  "category": "行业研究",
  "status": "待处理",
  "note": "重点阅读案例部分",
  "createdAt": "2026-07-31T04:00:00.000Z"
}
```

实际项目中的 `createdAt` 不由用户手工填写，而由程序使用统一时间格式生成，例如：

```
new Date().toISOString()
```

为什么需要 *id*

标题可能重复。

例如，两条记录都叫“AI 应用报告”，一条是资料，一条是任务。

如果只用标题识别记录，后续更新状态时容易改错对象。`id` 用于稳定识别目标记录。

为什么暂时不增加更多字段

第一版不增加：

- 截止日期；
- 优先级；
- 附件；
- 来源网址；
- 负责人；
- 多标签；
- 更新时间。

这些字段并非没有价值，但会增加表单、数据处理和验证复杂度。

13.5 固定类型、分类和状态

类型

第一版只有：

- 资料；
- 任务。

暂不增加灵感、联系人、日程、文件和项目等其他对象。

分类

第一版采用固定分类：

- 工作；
- 学习；
- 行业研究；
- AI 工具；
- 内容创作；
- 其他。

固定分类可以减少含义相近但写法不同的选项，例如：

- 行业；
- 行业资料；
- 行业报告；
- 行业研究。

若分类名称不统一，后续筛选结果也会变得不稳定。

状态

统一使用：

- 待处理；
- 进行中；
- 已完成。

资料与任务共用同一套状态。

状态	资料	任务
待处理	尚未阅读	尚未开始
进行中	正在阅读或整理	正在执行
已完成	已处理完毕	已完成

第一版不增加“已取消”“已归档”“已延期”“重要”或“紧急”等状态。

13.6 确定 MVP 范围和数据边界

MVP 是能够解决核心问题的最小可用版本。

这里的“最小”不是页面粗糙或功能残缺，而是：

> 只实现能够证明核心流程成立的必要能力。

第 14 章必须完成

- 新建独立项目；
- 显示新增记录表单；
- 新增资料或任务；
- 显示记录列表；
- 显示基础状态统计；
- 将数据写入 `localStorage`；
- 在指定运行环境下刷新后恢复数据；
- 显示初始空状态和基本错误提示；
- 提供最小 README。

第 15 章再完成

- 关键词搜索；
- 分类筛选；
- 状态更新；
- 组合条件下的空结果提示。

第一版明确不做

- 删除和批量操作；
- 账号和登录；
- 多设备同步；
- 数据库；
- 文件上传；
- 远程接口；
- 提醒通知；
- 自定义分类；
- 团队协作；
- 复杂测试框架。

本地存储边界

项目使用浏览器 `localStorage` 保存普通个人记录。

这种方式适合当前练习，因为：

- 不需要服务器；
- 不需要数据库；
- 不需要联网；
- 实现成本较低。

但它有明确限制：

- 数据与当前浏览器配置和页面来源相关；
- 更换浏览器或浏览器用户配置后，可能读取不到原记录；
- 移动本地项目文件位置后，部分浏览器中的存储表现可能变化；
- 隐私模式下的存储可能在会话结束后被清除；
- 清理浏览器数据可能导致记录丢失；
- 不支持多设备同步；
- 不具备可靠备份机制。

本书继续采用“本地页面＋指定浏览器”的教学路线。

在正式验收时，应固定：

- 同一浏览器；
- 同一浏览器用户配置；
- 同一项目文件位置。

若目标浏览器对本地文件的存储表现不稳定，可以改用本机已有的静态服务器工具运行。本章不要求为此额外安装开发环境；暂时无法稳定运行时，先记录实际现象，再继续完成需求、范围和验收设计。

工作台不得保存：

- 密码；
- 身份证件；
- 客户隐私；
- 企业机密；
- API 密钥；
- 其他敏感数据。

13.7 形成需求文档和空白项目

先在学习笔记目录中保存需求卡：

```
Codex 学习/
├── notes/
```

```
|   └── 第 13 章-个人资料与任务工作台需求.md
└── projects/
    └── personal-workspace/
```

需求文档记录：

- 主要使用者；
- 真实问题；
- 项目目标；
- 核心场景；
- 七个固定字段；
- 类型、分类和状态；
- 第 14 章 MVP；
- 第 15 章迭代；
- 非目标；
- 本地存储边界；
- 完成定义。

可以向 Codex 提交：

```
请根据本章已经确认的内容，创建学习笔记：

Codex 学习/notes/第 13 章-个人资料与任务工作台需求.md

文档只记录：

1. 主要使用者；
2. 真实问题；
3. 项目目标；
4. 四个核心场景；
5. 七个记录字段；
6. 固定类型、分类和状态；
7. 第 14 章 MVP 范围；
8. 第 15 章迭代范围；
9. 明确非目标；
10. localStorage 的使用条件和限制；
11. 数据安全边界；
12. 第 16 章完成定义。

不要创建页面代码，不要增加需求，也不要修改其他项目。
```

随后建立空白项目目录 `personal-workspace`，进入目录后初始化 Git：

```
git init
git status
```

本章不创建页面文件，也不创建 Git 提交。

第 16 章交付前，这份需求卡会整理进入项目中的 `docs/requirements.md`，成为正式项目资产。

三、总结

本章从真实问题出发，完成了第一个完整项目的范围定义。

你已经确定：

- 项目服务单机环境中的个人用户；
- 项目管理资料和任务两种记录；
- 每条记录使用七个固定字段；
- 分类和状态采用固定选项；
- 第 14 章只完成新增、展示和本地保存；
- 第 15 章增加搜索、分类筛选和状态更新；
- 第一版不引入数据库、账号和团队能力；
- `localStorage` 只能在明确的浏览器和运行条件下使用；
- 项目不保存敏感信息；
- 需求卡和空白 Git 仓库已经建立。

本章掌握检查

- [ ] 我能说明项目解决的真实问题；
- [ ] 我明确了主要使用者和核心场景；
- [ ] 我确定了七个记录字段；
- [ ] 我确定了固定类型、分类和状态；
- [ ] 我区分了 MVP 与后续迭代；
- [ ] 我理解本地存储的适用条件和限制；
- [ ] 我形成了需求文档；
- [ ] 项目目录已经建立，但尚未创建代码文件。

四、结束语与引导

本章解决了完整项目中最容易被忽略的问题：

> **先决定究竟要做什么，再开始写代码。**

下一章将从空白的 `personal-workspace` 目录开始，创建一个真正能够新增、显示并保存记录的第一版。

第 14 章：创建一个真正可以使用的第一版

一、起始语

当前已经具备两项基础资产：

```
Codex 学习/
├── notes/
│   └── 第 13 章-个人资料与任务工作台需求.md
└── projects/
    └── personal-workspace/
        └── .git/
```

项目目录中还没有页面文件。

本章将创建第一版 MVP，使用户能够：

87. 打开工作台；
88. 查看初始空状态；
89. 新增一条资料或任务；
90. 在列表中看到记录；
91. 查看基础状态统计；
92. 在固定浏览器和项目位置下，刷新后继续看到已保存内容。

本章不增加搜索、分类筛选、状态修改和删除。

这些功能若同时加入，会让“第一版为什么失败”变得难以判断。

二、主体

14.1 确认 MVP 的最小闭环

本章只需要证明下面这条流程成立：

```
打开页面
→ 填写记录
→ 尝试写入本地存储
→ 写入成功
→ 更新记录列表和统计
→ 刷新页面
→ 从本地存储恢复记录
```

需要特别注意顺序：

> 页面显示成功之前，应先确认数据写入成功。

否则可能出现：

- 页面看起来已经新增记录；
- 实际写入失败；
- 刷新后记录消失。

项目由四个文件组成：

```
personal-workspace/
├── index.html
├── css/
│   └── styles.css
├── js/
│   └── app.js
└── README.md
```

文件	职责
index.html	页面结构、表单、统计与列表容器
css/styles.css	页面、表单、卡片和状态样式
js/app.js	数据读取、保存、渲染、提交和统计
README.md	项目用途、打开方式、功能和限制

本章不创建：

- `package.json`；
- `node_modules`；
- 服务器文件；
- 数据库文件；
- 搜索脚本；
- 外部资源文件。

14.2 确认页面组成和异常状态

页面至少包含五个区域。

页面标题与说明

标题：

> 个人资料与任务工作台

说明：

> 集中记录资料和任务，随时查看进度并继续处理。

新增记录表单

表单包含：

- 类型；
- 标题；
- 分类；
- 状态；
- 备注；
- 保存按钮。

默认值为：

字段	默认值
类型	资料
分类	学习
状态	待处理

记录概览

显示：

- 全部；
- 待处理；
- 进行中；
- 已完成。

统计基于全部记录实时计算。

记录列表

每张记录卡片显示：

- 类型；
- 标题；
- 分类；
- 状态；
- 备注；
- 创建时间。

标题和备注来自用户输入，必须作为普通文字渲染。

优先使用：

- `textContent`；
- `createElement`；
- 其他等效的安全 DOM 方式。

不得将未经处理的用户输入直接拼接进 `innerHTML`。

空状态和错误提示

页面需要区分：

- 没有记录时的初始空状态；
- 表单输入不合格；
- 本地数据读取失败；
- 本地数据写入失败。

读取和写入错误不能只显示在浏览器控制台中。

14.3 正式提交 MVP 创建任务

先确认当前打开的是 `personal-workspace`，并运行 `git status`。

随后向 Codex 提交：

```
请在当前空白的 personal-workspace 项目中创建“个人资料与任务工作台”MVP。

一、允许创建的文件

- index.html
- css/styles.css
- js/app.js
- README.md

除 css 和 js 文件夹外，不创建其他目录或文件。

二、运行方式

1. 使用原生 HTML、CSS 和 JavaScript；
2. 主要教学路径为直接打开 index.html；
3. 不安装第三方依赖；
4. 不创建 package.json；
5. 不访问网络；
6. 不使用服务器或数据库。

三、页面功能

1. 页面标题为“个人资料与任务工作台”；
2. 页面说明为“集中记录资料和任务，随时查看进度并继续处理”；
3. 表单包含类型、标题、分类、状态和备注；
4. 显示全部、待处理、进行中和已完成四项统计；
5. 显示记录列表；
6. 无记录时显示初始空状态；
```

```
7. 提供可见的表单错误、读取错误和保存错误区域。

四、固定选项

类型：
- 资料
- 任务

分类：
- 工作
- 学习
- 行业研究
- AI 工具
- 内容创作
- 其他

状态：
- 待处理
- 进行中
- 已完成

默认值：
- 类型：资料
- 分类：学习
- 状态：待处理

五、记录字段

每条记录包含：
- id
- type
- title
- category
- status
- note
- createdAt

要求：
1. id 由程序生成；
2. createdAt 使用统一时间格式生成；
3. title 保存前去除首尾空格；
4. title 为空或只有空格时不得保存；
5. note 可以为空；
6. 新记录按创建时间倒序显示。

六、用户输入渲染

1. title 和 note 必须作为纯文本显示；
2. 优先使用 textContent 或等效安全 DOM 方式；
3. 不得将未经处理的用户输入直接写入 innerHTML。

七、本地保存

1. 使用 localStorage 键：personal-workspace.records.v1；
2. 页面加载时读取记录；
3. 没有存储数据时使用空数组；
4. 数据解析失败时显示错误并暂停新增保存；
5. 解析失败后不得使用空数组覆盖原始存储内容；
6. 新增记录时，先形成候选记录数组并尝试写入；
7. 写入成功后再更新页面中的记录状态；
8. 写入失败时显示错误，不得向用户表现为保存成功；
```

```
9. 刷新后从同一存储键恢复数据。

八、本章边界

1. 不增加搜索;
2. 不增加筛选控件;
3. 不增加状态修改;
4. 不增加删除;
5. 不增加自定义分类;
6. 不创建 Git 提交。

完成后请汇报:
- 创建了哪些文件;
- 数据读取、保存和渲染分别位于哪里;
- 读写失败分别怎样处理;
- 用户输入怎样按纯文本显示;
- 运行了哪些静态检查;
- 哪些结果仍需浏览器人工验证。
```

14.4 审查实际文件和范围

Codex 完成后，运行：

```
git status --short
```

通常应看到四个新文件：

```
?? README.md
?? css/styles.css
?? index.html
?? js/app.js
```

若出现以下内容，应暂停并要求说明：

- `package.json`;
- `node_modules`;
- 数据库文件；
- 搜索功能；
- 删除按钮；
- 外部图片；
- 任务外配置。

重点检查：

`index.html`

- 表单字段是否完整；
- 固定选项是否正确；
- 是否存在统计、列表、空状态和错误提示区域；

- 是否出现尚未实现的搜索或删除功能；
- 是否只引用本地 CSS 和 JavaScript。

js/app.js

- 是否使用 `personal-workspace.records.v1`；
- 是否区分读取失败与写入失败；
- 读取失败后是否暂停新增；
- 写入成功前是否避免更新正式页面状态；
- 是否拒绝空标题；
- 是否使用七个固定字段；
- 是否将标题和备注按纯文本渲染。

README.md

当前只需准确说明：

- 项目用途；
- 如何打开；
- 已实现功能；
- 尚未实现功能；
- 数据保存条件；
- 本地存储限制；
- 不应保存敏感信息。

14.5 理解安全的本地保存流程

`localStorage` 保存的是字符串。

记录写入过程通常是：

```
当前记录数组
→ 加入候选新记录
→ 转换为字符串
→ 尝试写入 localStorage
→ 写入成功
→ 更新内存记录
→ 重新渲染页面
```

读取过程通常是：

```
读取 localStorage 字符串
→ 没有数据：使用空数组
→ 有数据：尝试解析
```

```
→ 解析成功：加载记录
→ 解析失败：显示错误并锁定新增操作
```

需要避免两种虚假成功。

读取失败后的虚假空状态

错误做法：

```
解析失败
→ 自动使用空数组
→ 用户新增记录
→ 旧异常数据被覆盖
```

正确做法是：

> 显示读取错误，并暂停新增保存，避免覆盖无法识别的原数据。

写入失败后的虚假保存

错误做法：

```
先更新页面
→ 再尝试写入
→ 写入失败
→ 页面仍然显示新增成功
```

正确做法是：

> 写入成功后，再将候选数据确认为当前记录并更新页面。

14.6 在浏览器中验收 MVP

在同一浏览器、同一用户配置和同一项目文件位置下打开 `index.html`。

T01：首次打开

预期：

- 页面正常显示；
- 表单字段完整；
- 四项统计为 0；
- 显示初始空状态；
- 没有搜索和删除功能。

T02：输入边界

依次测试：

- 空标题；
- 只有空格的标题；
- 前后带空格的标题；
- 标题`<b>测试</b>`；
- 空备注。

预期：

- 空标题和纯空格不保存；
- 标题首尾空格被去除；
- `<b>测试</b>`按完整普通文字显示，不被渲染为加粗标签；
- 备注为空时可以保存。

T03：新增一条资料

例如：

字段	内容
类型	资料
标题	Codex 项目实践笔记
分类	学习
状态	进行中
备注	整理第 13—16 章案例

预期：

- 新记录出现在列表顶部；
- 全部数量加 1；
- 进行中数量加 1；
- 初始空状态消失。

T04：新增一项任务

例如：

字段	内容
类型	任务
标题	整理本周 AI 行业案例
分类	工作
状态	待处理
备注	至少准备三个案例

预期：

- 两条记录均存在；
- 全部数量为 2；
- 待处理与进行中统计正确。

T05：刷新恢复

刷新页面或关闭后重新打开。

预期：

- 两条记录仍然存在；
- 内容和状态保持一致；
- 统计结果正确。

若换浏览器、切换浏览器用户配置、移动项目位置或清理浏览器数据，不应将结果与上述测试直接等同。

14.7 修复局部问题

发现问题时，描述真实现象。

例如：

> 新增记录能够显示，但刷新后消失。

使用最小修复模板：

```
我实际看到的问题是：【描述具体问题】。

请先根据当前项目和 localStorage 逻辑定位原因，只进行解决该问题所需的最小修改。

要求：
1. 不增加搜索、筛选、状态修改或删除；
2. 不安装第三方依赖；
3. 不改变七个记录字段；
4. 不改变存储键；
5. 不创建 Git 提交。

完成后说明：
1. 根因是什么；
2. 修改了哪个文件；
3. 为什么修改能够解决问题；
4. 我需要重新执行哪些测试。
```

修复后，至少重新执行：

- 出现问题的步骤；

- 输入边界；
- 新增记录；
- 刷新恢复；
- 基础统计。

14.8 保存 MVP 基线版本

MVP 完成实际验收后，查看差异：

```
git status
git diff --check
git diff
```

将四个文件加入暂存区：

```
git add index.html css/styles.css js/app.js README.md
```

检查：

```
git diff --staged
```

确认：

- 没有搜索和筛选半成品；
- 没有状态修改；
- 没有第三方依赖；
- 用户输入按纯文本渲染；
- 读写失败没有被静默忽略；
- README 没有夸大当前能力。

随后创建提交：

```
git commit -m "创建个人资料与任务工作台 MVP"
```

最后运行 `git status`。

若工作区干净，说明 MVP 已经形成可靠基线。

三、总结

本章从空白目录创建了第一个真正处理用户数据的 MVP。

按照步骤执行成功后，项目能够：

- 新增资料和任务；

- 使用七个固定字段保存记录；
- 显示记录列表和状态统计；
- 拒绝空标题；
- 将用户输入按纯文本显示；
- 区分初始空状态和错误状态；
- 将记录写入 `localStorage`；
- 在固定环境下刷新后恢复；
- 避免读取异常和写入失败造成虚假成功；
- 通过 README 说明运行方式和限制。

本章掌握检查

- [ ] 项目只包含批准的四个文件；
- [ ] 页面能够在指定环境中打开；
- [ ] 我能新增资料和任务；
- [ ] 输入边界处理符合要求；
- [ ] 用户输入不会被当成 HTML 执行；
- [ ] 读取失败和写入失败分别处理；
- [ ] 刷新后记录仍然存在；
- [ ] MVP 已经保存为独立 Git 提交。

四、结束语与引导

现在，个人资料与任务工作台已经能够真正保存用户输入。

但随着记录增加，仅靠浏览列表仍然不够。

下一章将分两次小迭代完成：

> **关键词搜索、分类筛选和状态交互。**

第 15 章：让搜索、分类和状态交互完整运转

一、起始语

当前 MVP 已经具备：

- 新增资料或任务；
- 显示记录；
- 状态统计；
- 初始空状态；
- 本地保存；
- 在固定环境下刷新恢复。

项目中已经保存了用于测试的记录。

本章分成两次小迭代：

```
第一次迭代
搜索＋分类筛选＋筛选空结果

第二次迭代
状态更新＋统计联动
```

本章不增加：

- 删除；
- 批量操作；
- 账号；
- 数据库；
- 自定义分类；
- 远程同步。

> **受控案例说明：**真实项目在实现状态修改时，应同时检查数据是否正确保存。本书标准案例暂时保留一处已知缺陷，用于第 16 章统一学习“页面变化不等于数据持久化”。这是一项受控调试练习，不是推荐的开发方式。

本章只完成状态交互的候选实现，并将刷新持久化明确列为下一章必须验证的事项。

二、主体

15.1 开始迭代前检查旧数据

进入项目后运行：

```
git status
git log --oneline -1
```

开始时应满足：

- 最近一次提交为 MVP；
- 工作区干净；
- 浏览器中仍存在第 14 章保存的测试记录。

新功能不能要求用户清空原有数据才能使用。

第 14 章的记录应继续：

- 正常加载；
- 正常显示；
- 可以被搜索；
- 可以被分类筛选；
- 可以在当前页面中修改状态。

七个记录字段和存储键保持不变。

15.2 第一次迭代：增加搜索和分类筛选

向 Codex 提交：

```
请在当前 personal-workspace 项目中完成第一次功能迭代：搜索和分类筛选。

功能要求：

1. 在记录列表上方增加关键词搜索框；
2. 搜索 title 和 note；
3. 去除关键词首尾空格；
4. 英文匹配不区分大小写；
5. 增加分类筛选；
6. 分类选项沿用现有分类，并增加“全部分类”；
7. 搜索和分类筛选同时生效；
8. 没有匹配记录时显示筛选空结果；
9. 清空关键词并选择全部分类后恢复全部记录；
10. 顶部统计基于全部记录，不随筛选结果变化。

数据要求：
```

```
1. 不修改现有七个字段；
2. 不改变 localStorage 键；
3. 第 14 章旧数据必须继续读取；
4. 不清空或重新初始化本地存储；
5. 用户输入继续按纯文本渲染。

边界要求：

1. 只修改 index.html、css/styles.css 和 js/app.js；
2. 不增加状态修改；
3. 不增加删除；
4. 不安装第三方依赖；
5. 不创建 Git 提交。

完成后说明：
- 修改了哪些文件；
- 搜索与分类怎样组合；
- 如何保持旧数据兼容；
- 哪些结果需要人工验证。
```

组合逻辑为：

```
记录可见
=
符合关键词
并且
符合当前分类
```

顶部统计继续反映全部记录，不反映当前筛选结果。

15.3 验证搜索、分类和旧数据兼容

至少检查：

搜索标题和备注

分别输入：

- 只存在于标题中的关键词；
- 只存在于备注中的关键词。

预期能够找到对应记录。

处理空格和英文大小写

输入前后带空格的关键词，或使用不同大小写的英文。

预期匹配结果不受影响。

分类筛选

依次选择：

- 工作；
- 学习；
- 全部分类。

预期显示对应记录。

组合条件

输入关键词，同时选择分类。

预期只显示同时满足两项条件的记录。

旧数据兼容

刷新页面。

预期：

- 第 14 章记录仍然存在；
- 搜索框和分类筛选正常；
- 原有记录没有丢失；
- 顶部统计仍然正确。

15.4 第二次迭代：增加状态更新候选实现

当前每条记录已经显示状态，但用户还不能在列表中修改。

本次增加：

- 每条记录上的状态选择控件；
- 修改后立即更新当前页面；
- 统计同步变化；
- 搜索和分类条件继续生效。

提交：

```
请在当前 personal-workspace 项目中完成第二次小迭代：记录状态更新。

功能要求：

1. 每条记录允许在“待处理、进行中、已完成”之间切换；
2. 使用记录 id 确定更新对象；
3. 状态变化后立即重新渲染列表和统计；
4. 当前搜索和分类条件继续生效；
5. 第 14 章旧记录保持兼容；
6. 不改变七个字段和 localStorage 键；
7. 用户输入继续按纯文本显示。
```

```
阶段边界：

1. 本章先验收当前页面会话中的状态交互；
2. 刷新后状态是否保持，明确列为第 16 章必须执行的测试；
3. 不增加删除、批量操作、账号或数据库；
4. 不创建 Git 提交。

受控案例要求：
本书标准案例在状态处理逻辑中暂时保留“状态变化后未调用现有保存函数”的已知缺陷，用于第 16 章调试。请在完成说明中明确标记这一缺陷，不得将当前实现描述为最终正确版本。

完成后说明：
- 状态事件逻辑位于哪里；
- 怎样根据 id 找到记录；
- 怎样更新页面和统计；
- 当前已知缺陷是什么；
- 下一章必须验证什么。
```

真实项目不应为了练习而主动遗漏保存步骤。

这里使用统一受控缺陷，是为了让所有读者在第 16 章面对相同调试对象。

15.5 验证当前页面中的状态联动

选择一条“待处理”记录，将状态改为“进行中”。

立即检查：

- 卡片状态已经变化；
- 待处理统计减 1；
- 进行中统计加 1；
- 搜索关键词仍然保留；
- 当前分类筛选仍然生效；
- 没有生成重复记录。

再将另一条记录改为“已完成”，检查统计结果。

本章暂时不执行刷新持久化验收。

当前只能证明：

> 状态更新在当前页面会话中能够工作。

不能证明：

> 状态已经写回本地存储。

15.6 检查组合行为和两类空状态

操作	预期结果
输入关键词	只显示关键词匹配记录
选择分类	只显示该分类记录
同时使用两者	只显示同时满足条件的记录
更新可见记录状态	记录与统计立即更新
更新后仍符合筛选	继续显示
清空搜索并选择全部分类	恢复全部记录
无匹配结果	显示筛选空结果

需要区分两种状态。

初始空状态

数据集合本身没有任何记录。

提示可以是：

> 还没有记录，请先新增一条资料或任务。

筛选空结果

记录存在，但没有符合当前搜索和分类条件的内容。

提示可以是：

> 没有符合当前条件的记录，请调整关键词或分类。

两种提示不能混用。

15.7 检查实际差异和阶段边界

运行：

```
git status --short
git diff --check
git diff -- index.html css/styles.css js/app.js
```

预期只修改：

```
M index.html
M css/styles.css
M js/app.js
```

检查：

- 七个字段没有变化；
- 存储键没有变化；
- 旧数据没有被清空；
- 没有增加删除按钮；
- 没有新增依赖；
- 没有创建任务外文件；
- 没有创建 Git 提交；
- 已知持久化缺陷被明确记录。

15.8 锁定本章结束状态

第 15 章结束时，项目应满足：

- 搜索标题和备注；
- 按固定分类筛选；
- 搜索与分类同时生效；
- 筛选空结果提示正确；
- 旧数据继续兼容；
- 状态可以在当前页面中修改；
- 统计随状态修改更新；
- 三个功能文件处于未提交状态。

同时必须保留一项已知缺陷：

> 状态修改只更新当前内存记录和页面，尚未调用已有保存函数写回 `localStorage`。

当前结果只能称为：

> **功能迭代候选版本。**

不能称为完整 v1 交付。

三、总结

本章通过两次小迭代完善了工作台的使用能力。

第一轮完成：

- 关键词搜索；
- 分类筛选；

- 组合条件；
- 筛选空结果；
- 旧数据兼容。

第二轮形成：

- 当前页面中的状态更新；
- 统计联动；
- 筛选状态保持；
- 一处明确标记的受控持久化缺陷。

本章掌握检查

- [ ] 第 14 章旧数据仍然可以读取；
- [ ] 搜索能够匹配标题和备注；
- [ ] 分类筛选能够正常工作；
- [ ] 搜索和分类采用“并且”关系；
- [ ] 两类空状态能够区分；
- [ ] 状态修改能立即更新页面和统计；
- [ ] 没有增加删除、账号或数据库；
- [ ] 状态持久化缺陷已明确标记，尚未提交。

四、结束语与引导

工作台的主要交互已经形成，但“当前页面中状态变了”仍不能证明数据已经保存。

下一章将通过刷新测试复现故障，定位遗漏的保存动作，完成最小修复，再将项目整理为可以独立交付的 v1 版本。

第 16 章：调试、测试、优化和整理交付文件

一、起始语

当前项目已经具备：

- 新增和展示记录；
- 本地保存；
- 状态统计；
- 关键词搜索；
- 分类筛选；
- 当前页面中的状态更新。

但标准案例保留了一处受控缺陷：

> 状态变化后没有调用已有保存函数。

因此，当一条记录从“待处理”改为“已完成”时，页面和统计会立即变化；刷新后，记录可能恢复为原状态。

这说明：

> **界面更新成功，不代表数据已经持久化。**

本章将围绕这个真实故障完成：

```
复现问题
→ 收集证据
→ 定位根因
→ 最小修复
→ 边界测试
→ 核心回归
→ 必要可用性优化
→ 整理交付文档
→ 创建 v1 提交
```

本章不展开自动化测试框架、持续集成、发布治理和企业安全。

二、主体

16.1 用稳定步骤复现故障

先不要修改代码。

打开页面，选择一条状态为“待处理”的记录。

执行：

93. 将状态改为“已完成”；
94. 确认卡片显示“已完成”；
95. 确认统计数字变化；
96. 刷新浏览器；
97. 再次查看同一条记录。

若记录恢复为“待处理”，就形成了稳定复现步骤。

记录故障：

```
故障名称：
状态更新未在刷新后保留

前置条件：
页面中存在一条“待处理”记录。

操作：
1. 将记录状态改为“已完成”；
2. 观察页面和统计；
3. 刷新浏览器。

实际结果：
刷新前显示“已完成”，刷新后恢复为“待处理”。

预期结果：
刷新后仍显示“已完成”。

初步判断：
状态更新了内存数据和页面，但没有写回 localStorage。
```

稳定复现比一句“状态保存有问题”更容易调查。

16.2 让 Codex 根据证据定位根因

向 Codex 提交只读调查：

```
当前 personal-workspace 项目存在一个可稳定复现的故障：

1. 将一条记录从“待处理”改为“已完成”；
2. 页面和统计立即更新；
3. 刷新浏览器；
4. 记录恢复为原状态。

请先只读调查，不要修改文件。

请检查：

1. 页面加载时数据从哪里读取；
2. 新增记录时通过什么函数写入 localStorage；
3. 状态变化事件位于哪里；
```

```
4. 状态变化后更新了哪些内存数据；
5. 是否调用了现有保存函数；
6. 为什么页面立即变化但刷新后恢复；
7. 最小修复应位于哪个文件和代码位置；
8. 修复后需要重新测试哪些行为。

请提供文件路径和相关代码位置证据。
```

预期根因应接近：

```
状态变化处理函数
→ 修改 records 数组中的 status
→ 重新渲染页面
→ 没有调用现有保存函数
```

页面立即变化，是因为内存中的记录已经更新。

刷新后重新从 `localStorage` 读取时，获得的仍然是旧状态。

16.3 完成最小持久化修复

根因明确后，允许 Codex 修改：

```
请根据刚才确认的根因，完成状态持久化的最小修复。

要求：

1. 状态变化后，将最新 records 写入现有 localStorage 键；
2. 复用项目已有保存函数；
3. 写入成功后再确认页面状态；
4. 写入失败时显示错误，并避免向用户表现为已经成功保存；
5. 保留现有搜索、分类、统计和渲染行为；
6. 不修改七个字段；
7. 不改变存储键；
8. 不清空或迁移已有数据；
9. 不增加删除、账号、数据库或第三方依赖；
10. 不进行与故障无关的重构；
11. 暂不创建 Git 提交。

完成后说明：
- 修改了哪个文件和位置；
- 修复前缺少了什么；
- 修复后的数据流是什么；
- 写入失败时怎样处理；
- 需要重新执行哪些测试。
```

修复后的流程应为：

```
用户选择新状态
→ 根据 id 生成更新后的候选记录数组
→ 尝试写入 localStorage
→ 写入成功
→ 更新当前 records
→ 重新渲染列表和统计
```

若写入失败：

```
显示保存错误
→ 保留原状态
→ 不向用户表现为已持久化
```

16.4 验证状态能够在刷新后保留

重新执行原复现步骤。

T01：待处理改为已完成

预期：

- 页面显示已完成；
- 统计更新；
- 刷新后仍为已完成。

T02：已完成改为进行中

预期：

- 页面和统计更新；
- 刷新后仍为进行中。

T03：筛选状态下修改

先输入关键词或选择分类，再修改一条可见记录状态。

预期：

- 修改成功；
- 当前筛选条件保持；
- 刷新后状态仍然保留；
- 记录仍能通过原有搜索和分类找到。

只要刷新后恢复旧值，故障就没有真正修复。

16.5 检查边界输入和核心回归

输入边界

测试	预期结果
标题为空或只有空格	不保存并显示提示
标题前后有空格	保存前去除
标题包含 HTML 标签文字	按普通文字显示
备注为空	可以正常保存
标题或备注较长	能够换行，不破坏卡片布局
当前没有记录	显示初始空状态
筛选无结果	显示筛选空结果

核心回归

重新检查：

- 新增资料；
- 新增任务；
- 刷新后恢复；
- 四项统计；
- 搜索标题和备注；
- 分类筛选；
- 搜索与分类组合；
- 状态更新；
- 状态刷新后保持；
- 读取错误和保存错误提示没有被删除。

具备条件时，可以检查浏览器控制台是否出现明显脚本错误。不会使用开发者工具，不影响完成本章主要流程。

回归的目的不是机械重复全部步骤，而是确认：

> 修复状态持久化时，没有破坏原来已经可用的功能。

16.6 完成必要可用性优化

本章标题中的“优化”，只指测试中已经发现的必要可用性问题。

允许处理：

- 长标题和备注能够正常换行；
- 表单标签与输入控件清楚对应；
- 键盘操作时焦点状态可见；
- 窄屏下表单和卡片不横向溢出；
- 错误提示清楚但不过度遮挡内容；
- 按钮和选择框文字容易辨认。

不允许：

- 重做整体视觉主题；
- 增加动画体系；
- 更换技术架构；
- 增加新的业务功能；
- 增加删除、导入导出或提醒；
- 为了“优化”大规模重写项目。

可以向 Codex 提交：

```
请根据我已经完成的人工测试，只处理下列实际发现的可用性问题：

【粘贴真实问题，例如：长标题溢出、窄屏表单横向滚动、错误提示不明显】

要求：

1. 只修改解决这些问题所需的 HTML 或 CSS；
2. 不增加业务功能；
3. 不改变数据结构和 localStorage 逻辑；
4. 不重做整体视觉；
5. 不安装第三方依赖；
6. 不创建 Git 提交。

完成后说明实际修改范围和需要重新验证的页面状态。
```

没有实际发现的问题，不应凭空生成优化任务。

16.7 整理需求、README 和验证记录

最终项目需要保留三类文档资产。

docs/requirements.md

将第 13 章需求卡整理进入项目，记录：

- 主要使用者；
- 真实问题；

- 核心场景；
- 数据字段；
- MVP 和迭代范围；
- 非目标；
- 数据边界；
- 完成定义。

不重新扩张需求。

README.md

README 至少包含：

98. 项目用途；
99. 主要功能；
100. 文件结构；
101. 打开方式；
102. 固定浏览器与文件位置说明；
103. 七个记录字段；
104. 固定类型、分类和状态；
105. 本地存储方式；
106. 已知限制；
107. 当前不支持的功能；
108. 数据安全提醒；
109. 测试方式。

不得宣称：

- 支持云端同步；
- 支持多人协作；
- 数据永久不会丢失；
- 已完成自动化测试；
- 适合保存敏感信息。

docs/validation.md

记录：

- 状态持久化故障；
- 稳定复现步骤；
- 根因；
- 最小修复；
- 刷新验证结果；

- 输入边界结果；
- 核心回归结果；
- 必要可用性优化；
- 已知限制；
- 尚未验证事项；
- 最终交付结论。

可以提交：

```
我已经完成个人资料与任务工作台的人工测试。

以下是实际测试结果：
【粘贴状态持久化、输入边界、核心回归和可用性检查的真实结果】

请完成：

1. 将第 13 章已确认的需求卡整理为 docs/requirements.md；
2. 更新 README.md；
3. 创建 docs/validation.md。

要求：
- 不新增需求；
- 不虚构没有执行的测试；
- README 不得夸大功能；
- requirements 与当前七个字段、范围和非目标保持一致；
- validation 区分通过、失败、未知和未执行；
- 除这三份文档外，不修改其他文件；
- 不创建 Git 提交。
```

16.8 审查交付目录并创建 v1 提交

最终项目结构应为：

```
personal-workspace/
├── index.html
├── css/
│   └── styles.css
├── js/
│   └── app.js
├── docs/
│   ├── requirements.md
│   └── validation.md
└── README.md
```

检查差异：

```
git status --short
git diff --check
git diff --stat
git diff
```

当前差异应包括：

- 第 15 章的搜索、分类和状态交互；
- 第 16 章的状态持久化修复；
- 已确认的必要可用性优化；
- 更新后的 README；
- 新增需求与验证文档。

不应出现：

- 数据库；
- 账号代码；
- 删除和批量功能；
- 第三方依赖；
- 网络请求；
- 自动化测试框架；
- 持续集成配置；
- 任务外文件。

将交付文件加入暂存区：

```
git add index.html css/styles.css js/app.js README.md docs/requirements.md docs/validation.md
```

检查：

```
git status
git diff --staged
```

确认：

- 所有修改属于第 15—16 章批准范围；
- 验证记录只包含真实结果；
- README 没有夸大功能；
- 需求文档与第 13 章一致；
- 状态保存复用了已有存储逻辑；
- 没有未批准文件。

随后创建提交：

```
git commit -m "完善工作台功能并完成 v1 交付"
```

最终检查：

```
git status
git log --oneline -2
```

按照步骤执行成功后，历史应至少包含：

```
完善工作台功能并完成 v1 交付
创建个人资料与任务工作台 MVP
```

工作区应当干净。

这时，项目才从“能够运行”进入“能够交付”。

三、总结

本章完成了个人资料与任务工作台的正式交付闭环：

- 稳定复现状态持久化故障；
- 根据代码证据定位根因；
- 复用已有保存函数完成最小修复；
- 区分写入成功和写入失败；
- 验证状态在刷新后保留；
- 检查输入边界和核心回归；
- 只处理实际发现的必要可用性问题；
- 整理需求、README 和验证记录；
- 审查最终差异；
- 创建干净的 v1 提交。

本章掌握检查

- [ ] 我能够用稳定步骤复现故障；
- [ ] 根因来自真实代码证据；
- [ ] 修复复用了现有保存函数；
- [ ] 状态更新在刷新后仍然保留；
- [ ] 写入失败不会表现为保存成功；
- [ ] 我完成了输入边界和核心回归；
- [ ] 需求、README 和验证记录一致；
- [ ] v1 提交完成后工作区保持干净。

四、结束语与引导

到这里，你已经从一个空白目录出发，完成了第一项真正可以持续使用的完整项目：

> **个人资料与任务工作台 v1**

你不仅让 Codex 创建了页面，还完成了：

- 需求范围；

- 数据模型；
- MVP；
- 功能迭代；
- 数据兼容；
- 真实故障调试；
- 回归验证；
- 必要可用性优化；
- 交付文档；
- 版本提交。

第 13—16 章项目到此封存。

从下一章起，学习方式将发生变化。

你会接手一个已经存在、可以运行，但并不熟悉的项目：

团队任务看板

第一步不是立即修改代码，而是让 Codex 读懂陌生项目，建立项目地图和运行基线。

第 17 章：让 Codex 读懂一个陌生项目

一、起始语

前 16 章中的项目，都由我们从需求阶段开始参与创建。

你知道：

- 项目为什么存在；
- 文件是什么时候建立的；
- 数据字段为什么这样设计；
- 功能经历了哪些迭代；
- 哪些限制来自第一版范围。

现实工作通常不会这么理想。

你可能接手：

- 同事移交的内部工具；
- 外部供应商留下的代码；
- 一段开源项目；
- 几个月前已经遗忘的个人项目；
- 另一名开发者或智能体创建的仓库。

项目能够运行，但你并不知道：

- 应该先读哪个文件；
- 页面从哪里开始；
- 数据保存在哪里；
- 一次用户操作会经过哪些函数；
- 哪些文件可以局部修改；
- 哪些字段被多个模块共同依赖；
- README 与真实代码是否一致。

从本章开始，我们使用新的贯穿案例：

> **团队任务看板**

项目目录名称为 `team-task-board`。

它已经能够：

- 新增任务；

- 按待处理、进行中和已完成分列展示；
- 设置负责人和优先级；
- 搜索任务；
- 按负责人和优先级筛选；
- 更新任务状态；
- 删除任务；
- 将数据保存在浏览器本地。

本章不修改业务代码。

唯一任务是：

> **建立可信的项目地图和运行基线，明确下一步功能应该修改哪里。**

理解陌生代码库时，更有效的方式是追踪业务流程、梳理模块关系并定位相关文件，而不是从第一个文件开始无差别阅读全部代码。OpenAI 当前也将追踪请求流程、映射陌生模块和快速找到正确文件列为 Codex 理解大型代码库的典型用途。OpenAI Developers

二、主体

17.1 导入练习项目并确认范围

将随书提供的唯一基础练习项目 `team-task-board` 复制到：

```
Codex 学习/
└── projects/
    ├── ai-learning-page/
    ├── personal-workspace/
    └── team-task-board/
```

标准练习项目结构如下：

```
team-task-board/
├── css/
│   └── styles.css
├── docs/
│   ├── product-overview.md
│   └── manual-test-checklist.md
├── js/
│   ├── constants.js
│   ├── seed-data.js
│   ├── storage.js
│   ├── filters.js
│   ├── render.js
│   └── app.js
├── .gitignore
├── index.html
└── README.md
```

文件名只能提供初步线索。

例如，`storage.js` 看起来与数据保存有关，但在阅读代码前，不能直接断定：

- 它是唯一写入本地存储的文件；
- 它会处理所有读取异常；
- 它不会覆盖已有数据；
- 所有业务操作都会经过它。

第一次连接项目时，先让 Codex 确认范围：

```
请只检查当前 team-task-board 项目，不要修改、移动、创建或删除任何文件。

请说明：

1. 当前工作目录；
2. 当前目录的一级结构；
3. 是否存在 Git 仓库；
4. 是否存在依赖配置文件；
5. 是否存在疑似环境变量、凭据或真实业务数据文件；
6. 哪些目录属于当前任务范围；
7. 哪些目录不应访问。

不要读取或输出任何疑似密钥的具体内容。
```

若发现 `.env`、凭据文件、API 密钥或真实客户数据，应停止练习并先处理数据边界。

这套基础练习项目只包含代码、说明文档和虚构示例数据。第 17—36 章始终在同一项目上持续修改，不需要再次下载其他版本、完成版或章节快照。

17.2 建立可恢复的导入基线

这套基础练习项目不预置 Git 历史时，在项目根目录运行：

```
git init
git status --short
```

确认未跟踪内容只属于当前项目后，暂存明确文件：

```
git add .gitignore README.md index.html css js docs
git status
git diff --staged --stat
```

`git add css js docs` 会暂存目录中的全部内容，因此还需要查看状态和暂存差异，不能仅凭目录名称判断所有内容都应进入提交。

确认范围无误后创建导入提交：

```
git commit -m "导入团队任务看板练习项目"
git branch -M main
```

随后检查：

```
git status
git log --oneline -1
```

预期状态是：

- 当前分支为 `main`；
- 工作区干净；
- 最近提交为`导入团队任务看板练习项目`。

若项目本身已经带有可信 Git 历史，不要重新初始化、删除或覆盖原历史。

导入提交的作用是建立一个清楚的接手时刻：

> 从现在开始，所有修改都可以与最初状态进行比较。

17.3 区分仓库基线和运行时基线

打开陌生项目前，需要区分两种状态。

仓库基线

包括：

- 当前分支；
- 最近提交；
- Git 工作区；
- 项目文件；
- 依赖和配置。

本章浏览器观察期间，不应修改仓库文件，Git 工作区应保持干净。

运行时基线

包括：

- 实际使用的浏览器；
- 页面打开方式；
- 本地存储键；
- 当前任务数量；
- 示例数据是否已经初始化；

- 页面首次打开后出现的状态。

标准项目包含 `seed-data.js`。

如果代码设计为第一次打开页面时将虚构示例数据写入 `localStorage`，那么：

> 打开页面虽然没有修改仓库文件，却可能初始化浏览器中的运行时数据。

因此，本章不应写成“所有数据完全没有变化”，而应记录：

- 示例数据在什么条件下初始化；
- 初始化前是否已有存储数据；
- 使用了哪个存储键；
- 初始化后出现多少条任务；
- 刷新后是否仍然存在。

建议记录一份运行基线：

项目	实际记录
浏览器	当前使用的浏览器
浏览器用户配置	当前用户配置或 Profile
项目打开方式	本地页面或本地 HTTP
项目路径	当前固定路径
存储键	从代码确认的真实键名
初始任务数	稳定打开后的实际数量
示例数据是否初始化	是／否
Git 状态	干净／存在变化

第 18—19 章应继续使用同一浏览器、同一用户配置、同一项目路径和同一存储键。

17.4 按层次阅读项目

陌生项目可以按照下面的顺序阅读：

```
README 与产品说明
        ↓
目录结构
        ↓
index.html 页面入口
        ↓
JavaScript 加载顺序
        ↓
固定值和任务结构
```

```
        ↓
存储、筛选与渲染
        ↓
app.js 初始化和用户事件
        ↓
浏览器实际结果
```

先读说明文件

先只读：

- `README.md`
- `docs/product-overview.md`
- `docs/manual-test-checklist.md`

将信息分成三类：

信息类型	含义
文档声明	文档声称项目怎样工作
代码确认	可以由真实实现证明
运行确认	需要在浏览器中观察

例如，README 可能写着“支持本地保存”，但仍需通过代码确认：

- 实际使用哪个存储键；
- 写入发生在什么函数；
- 数据解析失败时怎样处理；
- 删除和状态更新是否都会保存。

再读页面入口

读取 `index.html`，确认：

- 页面标题；
- 新增任务表单；
- 三个状态列；
- 搜索和筛选控件；
- CSS 路径；
- JavaScript 引用顺序；
- 脚本依赖的关键 `id` 和 `data-*` 属性。

标准项目按下面顺序加载脚本：

```
<script src="js/constants.js"></script>
<script src="js/seed-data.js"></script>
<script src="js/storage.js"></script>
```

```
<script src="js/filters.js"></script>
<script src="js/render.js"></script>
<script src="js/app.js"></script>
```

后面的脚本可能依赖前面已经定义的常量和函数。

本章只记录依赖关系，不调整加载顺序。

17.5 建立模块地图和数据流

使用一张表梳理文件职责。

模块	文件	主要输入	主要输出	修改风险
固定值	constants.js	无	状态、优先级、负责人、存储键	高
示例数据	seed-data.js	固定值	虚构初始任务	中
存储	storage.js	任务数组、本地存储	读取或保存结果	高
筛选	filters.js	完整任务和筛选条件	当前显示结果	中
渲染	render.js	当前显示结果	任务卡片和空状态	高
页面协调	app.js	表单、事件、各模块函数	初始化和业务流程	高
样式	styles.css	页面结构和状态类名	页面呈现	中
测试说明	manual-test-checklist.md	产品规则	人工测试范围	低

一条任务使用七个核心字段：

```
{
  id: "task-001",
  title: "整理产品周报",
  owner: "林晓",
  priority: "高",
  status: "待处理",
  note: "周五前提交",
  createdAt: "2026-07-20T09:30:00.000Z"
}
```

这些字段共同构成当前项目的数据契约。

例如，将 `owner` 改名为 `assignee`，可能同时影响：

- 示例数据；
- 表单提交；
- 本地历史数据；
- 负责人筛选；

- 卡片渲染；
- 测试说明。

还要区分三层状态：

```
localStorage 中的持久化数据
        ↓
内存中的完整任务数组
        ↓
搜索和筛选后的当前结果
        ↓
页面中的任务卡片
```

页面中看不到任务，不代表任务已经删除。

页面状态发生变化，也不代表变化一定已写入本地存储。

17.6 跟踪三条核心流程

页面首次加载

```
浏览器加载脚本
→ 检查是否已有本地数据
→ 必要时初始化示例任务
→ 读取完整任务数组
→ 取得当前搜索与筛选条件
→ 生成显示结果
→ 渲染三个状态列
```

新增任务

```
提交表单
→ 清理和校验输入
→ 生成 id 与 createdAt
→ 形成候选任务
→ 保存完整任务数组
→ 更新内存状态
→ 重新筛选并渲染
```

更新任务状态

```
状态控件变化
→ 取得任务 id
→ 在完整数组中找到任务
→ 更新 status
→ 保存任务数组
→ 重新筛选并渲染
```

让 Codex 根据真实代码标注每一步所在文件和函数。

```
请只读追踪当前项目的三条流程：
```

```
1. 页面首次加载；
2. 新增任务；
3. 更新任务状态。

每条流程请说明：
- 起点；
- 经过的文件和函数；
- 读写的数据；
- 保存发生的位置；
- 最终怎样重新渲染。

请区分：
- 文档声明；
- 代码确认；
- 需要浏览器验证的结论。

不要修改项目。
```

17.7 形成项目地图并锁定基线

在项目外的学习笔记目录创建：

`Codex 学习/notes/第 17 章-团队任务看板项目地图.md`

文档包含：

110. 项目定位和运行方式；
111. 真实文件树；
112. 模块职责表；
113. 脚本加载顺序；
114. 七个任务字段；
115. 初始化、新增和状态更新流程；
116. 存储键和示例数据初始化条件；
117. 浏览器运行基线；
118. 高影响文件和字段；
119. 文档、代码与运行结果的冲突；
120. 下一功能可能涉及的文件；
121. 尚未确认的问题。

每条关键结论标明来源：

- 代码确认；
- 浏览器确认；
- 文档声明；
- 仍待确认。

本章结束时应满足：

- `main` 包含导入基线提交；
- Git 工作区干净；
- 项目文件没有改变；
- 浏览器运行时基线已经记录；
- 项目地图保存在项目外；
- 尚未创建 `AGENTS.md`。

三、总结

本章没有修改团队任务看板的业务代码。

你完成了：

- 确认项目和数据边界；
- 建立导入基线提交；
- 区分仓库状态与运行时状态；
- 记录示例数据初始化条件；
- 识别页面入口和脚本加载顺序；
- 建立模块职责表；
- 确认七个任务字段；
- 跟踪三条核心流程；
- 区分持久化数据、内存数据和显示结果；
- 形成有证据的项目地图。

本章掌握检查

- [ ] 当前仓库位于正确目录且工作区干净；
- [ ] 我能区分仓库基线和运行时基线；
- [ ] 示例数据是否初始化已经被记录；
- [ ] 我能指出页面入口和脚本顺序；
- [ ] 我能说明主要模块的职责；
- [ ] 我知道任务数据包含哪些字段；
- [ ] 我能跟踪初始化、新增和状态更新流程；
- [ ] 项目地图中的关键结论都有来源。

四、结束语与引导

团队任务看板已经不再只是一个陌生文件夹。

你能够说明：

- 页面怎样启动；
- 数据怎样初始化、读取和保存；
- 搜索与筛选怎样工作；
- 任务卡片怎样生成；
- 用户操作会经过哪些文件；
- 哪些字段和函数不能随意改变。

下一章开始第一次真实修改：

> **为任务增加可选截止日期，并对逾期的未完成任务显示提示。**

重点不是简单加入一个日期输入框，而是控制跨文件影响并兼容没有日期的历史任务。

第 18 章：增加一个新功能并控制修改范围

一、起始语

第 17 章已经形成团队任务看板的项目地图。

当前基线是：

- 主分支为 `main`；
- 最近提交为导入基线；
- 工作区干净；
- 项目能够在固定环境中运行；
- 七个任务字段已经确认；
- 存储、筛选、渲染和事件流程已经梳理。

现在增加第一个真实功能：

> **任务截止日期和逾期提示。**

表面上，它只是一个日期输入框。

实际会影响：

- 新增任务表单；
- 任务数据结构；
- 历史任务兼容；
- 日期显示；
- 逾期判断；
- 卡片样式；
- 状态变化后的重新计算；
- README 和人工测试说明。

本章要回答三个问题：

122. 哪些文件确实需要修改；
123. 哪些文件应该保持不变；
124. 怎样保证所有差异围绕同一个业务目标。

二、主体

18.1 在独立功能分支中开始

先确认主分支干净：

```
git switch main
git status
```

创建并切换到功能分支：

```
git switch -c feature/task-due-date
```

再次检查：

```
git branch --show-current
git status
```

预期当前分支是：

`feature/task-due-date`

`git switch -c` 会在指定起点创建新分支，并在创建成功后切换到该分支。它相当于先创建分支再切换，但只有切换成功时才会完成创建。Git

第 18—20 章均在该功能分支上推进，直到 PR 合并后再回到 `main`。

18.2 锁定截止日期业务规则

新增字段名称固定为：

`dueDate`

保存格式固定为：

`YYYY-MM-DD`

它是一个可选字段。

一条新任务可以表示为：

```
{
  id: "task-101",
  title: "准备客户方案",
  owner: "林晓",
  priority: "高",
  status: "进行中",
  note: "周会前完成",
```

```
  createdAt: "2026-07-31T05:00:00.000Z",
  dueDate: "2026-08-05"
}
```

历史任务没有 `dueDate` 时，应被视为：

> 未设置截止日期。

不要求批量修改历史数据，也不更换存储键。

逾期规则

任务只有同时满足以下条件时，才显示“已逾期”：

```
dueDate 存在且为有效日历日期
并且
dueDate 早于用户本地今天
并且
status 不是“已完成”
```

情况	页面提示	是否逾期
字段不存在或为空	未设置截止日期	否
格式或日历值无效	截止日期异常	否
截止日期是今天	显示日期	否
截止日期在未来	显示日期	否
过去日期，任务未完成	显示日期和已逾期	是
过去日期，任务已完成	显示日期	否

“已逾期”不是第四种任务状态。

任务仍然只有：

- 待处理；
- 进行中；
- 已完成。

逾期是根据日期和当前状态实时计算的派生结果，不写入 `overdue` 字段。

18.3 正确处理纯日期和本地今天

HTML 日期输入框的界面显示方式会随浏览器和系统区域设置变化，但它的实际 `value` 会统一为 `yyyy-mm-dd`，且不包含具体时间。MDN Web Docs

因此，项目把 `dueDate` 保存为纯日期字符串，不把它当成某个 UTC 时间点。

应避免：

```
new Date(dueDate)
```

日期单独以 `YYYY-MM-DD` 形式交给 `Date.parse()` 或 `Date` 构造器时，会按照 UTC 日期解释；在部分负时区显示时，可能落到当地前一天。MDN Web Docs

也不应使用：

```
new Date().toISOString().slice(0, 10)
```

来代表用户本地今天，因为 `toISOString()` 基于 UTC 时间。

更稳妥的基础方法是：

```
function getLocalDateKey(date = new Date()) {
  const year = date.getFullYear();
  const month = String(date.getMonth() + 1).padStart(2, "0");
  const day = String(date.getDate()).padStart(2, "0");

  return `${year}-${month}-${day}`;
}
```

再分别处理：

```
保存：保留 YYYY-MM-DD 字符串
比较：有效 dueDate 与本地今天字符串比较
显示：拆分年、月、日后生成中文文字
```

因为有效日期统一使用 `YYYY-MM-DD`，字符串顺序与日历顺序一致。

还需要验证实际日历值。

`2026-02-30` 虽然外形符合格式，却不是有效日期，应显示“截止日期异常”，且不参与逾期判断。

18.4 完成只读影响分析

向 Codex 提交：

```
请根据第 17 章形成的项目地图，对“任务截止日期与逾期提示”进行只读影响分析。

业务规则：

1. 新增可选字段 dueDate，格式为 YYYY-MM-DD；
2. 没有 dueDate 的历史任务继续正常显示；
3. 缺失或空日期显示“未设置截止日期”；
4. 无效日期显示“截止日期异常”；
5. 有效日期早于本地今天且任务未完成时显示“已逾期”；
6. 今天截止不算逾期；
7. 已完成任务不显示逾期；
8. 逾期是派生结果，不新增任务状态或 overdue 字段；
```

```
9. 不更换存储键，不批量迁移历史数据；
10. 不使用 UTC 日期替代本地日历日期。

请输出：

1. 必须修改的文件及原因；
2. 可以保持不变的文件及依据；
3. 历史数据兼容风险；
4. 日期、格式和本地时区风险；
5. 需要执行的局部测试；
6. 范围需要扩大时的停止条件。

只输出分析，不修改文件。
```

预计需要修改：

- `index.html`：增加日期输入；
- `js/app.js`：读取并保存 `dueDate`；
- `js/render.js`：验证、显示和判断日期；
- `css/styles.css`：日期和逾期样式；
- `README.md`：说明新功能；
- `docs/manual-test-checklist.md`：增加测试项。

预计不需要修改：

- `constants.js`：没有新增任务状态；
- `seed-data.js`：历史示例可以没有 `dueDate`；
- `storage.js`：继续保存完整任务对象；
- `filters.js`：本章不增加日期筛选。

若真实代码证明其他文件必须修改，Codex 应先说明依赖关系，再等待批准。

18.5 实施范围明确的功能任务

```
请在当前 feature/task-due-date 分支中实现截止日期和逾期提示。

允许修改：

- index.html
- css/styles.css
- js/app.js
- js/render.js
- README.md
- docs/manual-test-checklist.md

数据规则：

1. 新增可选字段 dueDate；
2. 保存格式为 YYYY-MM-DD；
3. 历史任务没有 dueDate 时视为未设置；
```

```
4. 不更换 localStorage 键;
5. 不批量修改旧任务;
6. 不新增“已逾期”状态;
7. 不保存 overdue 字段。

日期规则:

1. 缺失或空日期显示“未设置截止日期”，不逾期;
2. 格式或日历值无效时显示“截止日期异常”，不逾期;
3. 今天截止不逾期;
4. 未来日期不逾期;
5. 过去日期且任务未完成时逾期;
6. 已完成任务不逾期;
7. 状态变化后重新计算;
8. 使用用户本地日历日期;
9. 不直接通过 new Date(dueDate)解析纯日期;
10. 不使用 UTC 日期字符串代替本地今天。

边界:

1. 不修改 constants.js、seed-data.js、storage.js 和 filters.js;
2. 不增加日期排序、日期筛选、通知或日历;
3. 不重构项目;
4. 不安装依赖;
5. 不创建 Git 提交;
6. 不创建 PR;
7. 人工测试完成前不创建验证结论文档。

完成后说明:
- 实际修改了哪些文件;
- 旧数据怎样兼容;
- 日期有效性怎样判断;
- 本地今天怎样生成;
- 逾期判断位于哪里;
- 哪些行为仍需人工验证。
```

18.6 检查修改边界和日期实现

执行后运行:

```
git status --short
git diff --check
git diff --stat
```

预期变化集中在六个批准文件。

若出现:

- `constants.js`
- `storage.js`
- `filters.js`
- `seed-data.js`
- `package.json`

- 数据迁移脚本
- 网络请求

应暂停并检查必要性。

重点审查 `render.js`：

- 是否先区分缺失日期和异常日期；
- 是否验证实际日历值；
- 是否使用本地年、月、日生成今天；
- 是否避免 `new Date(dueDate)`；
- 是否只有未完成的过去任务才逾期；
- 是否在状态变化后重新渲染结果。

18.7 完成人工测试并形成验证记录

至少测试：

测试	预期结果
历史任务没有字段	未设置截止日期，不逾期
新任务留空日期	未设置截止日期，不逾期
无效日期数据	截止日期异常，不逾期
未来日期	显示日期，不逾期
本地今天	显示日期，不逾期
过去日期、待处理	已逾期
过去日期、进行中	已逾期
过去日期、已完成	不逾期
已完成重新变为进行中	重新显示逾期
刷新页面	日期和状态仍然保留

同时抽查：

- 搜索；
- 负责人筛选；
- 优先级筛选；
- 多条件组合。

记录“本地今天”测试时，需要写入：

- 测试日期；
- 使用的浏览器；
- 输入的 `dueDate`；
- 任务状态；
- 实际结果。

完成真实测试后，再创建：

`docs/due-date-validation.md`

```
我已经完成截止日期功能的人工测试。

以下是实际结果：
【粘贴真实测试结果】

请创建 docs/due-date-validation.md，记录：

1. 功能范围；
2. 实际修改文件；
3. 浏览器与本地测试日期；
4. 历史无日期任务；
5. 新增无日期任务；
6. 无效日期；
7. 未来、今天和过去日期；
8. 已完成和未完成状态；
9. 状态变化后的逾期重算；
10. 搜索与筛选回归；
11. 刷新恢复；
12. 失败、未知和未执行项目。

不得虚构未执行测试，不要创建 Git 提交。
```

第 18 章正常结束状态应是：

- 截止日期功能已经正确实现；
- 历史数据兼容；
- 日期使用本地日历规则；
- 验证记录基于真实测试；
- 功能分支尚未提交；
- 尚未创建 PR。

三、总结

本章在陌生项目中完成了一次范围明确的跨文件功能修改。

你已经完成：

- 创建独立功能分支；
- 将 `dueDate` 设计为可选字段；

- 区分缺失日期和异常日期；
- 将逾期定义为派生结果；
- 使用本地日历日期完成比较；
- 避免把纯日期误当成 UTC 时间点；
- 兼容没有新字段的历史任务；
- 控制修改文件范围；
- 完成日期和原功能回归；
- 根据真实结果形成验证记录。

本章掌握检查

- [ ] 当前位于 `feature/task-due-date`；
- [ ] `dueDate` 是可选字段；
- [ ] 缺失日期和异常日期能够区分；
- [ ] 逾期不是新的任务状态；
- [ ] 日期比较使用本地日历规则；
- [ ] 存储键和历史任务没有被迁移；
- [ ] 实际修改没有超出批准范围；
- [ ] 功能尚未提交或创建 PR。

四、结束语与引导

截止日期功能已经完成候选实现和人工验证。

但为了训练证据型调试，第 19 章还要处理一个固定故障：

> **没有截止日期的任务被错误标记为逾期。**

若你的实际实现出现了这一 Bug，直接使用当前分支继续。

若你的实现已经正确，第 19 章将直接提供一段固定的错误判断代码和相应案例事实。你只需分析证据、提出最小修复并设计回归测试，不需要破坏正确代码，也不需要下载独立故障快照。

第 19 章：根据报错和证据定位并修复问题

一、起始语

第 18 章已经形成正确的截止日期候选功能。

第 19 章训练的重点不是继续增加功能，而是学习：

> 当页面表现不符合业务规则时，怎样依靠证据定位根因。

调试对象是：

> **无截止日期任务被错误标记为“已逾期”。**

这个问题可能没有语法错误，也不一定让页面崩溃。

浏览器控制台甚至可能没有异常。

这说明：

> 没有报错，不等于没有 Bug。

本章支持两条路径。

路径 A：实际实现出现了同类 Bug

直接在当前 `feature/task-due-date` 分支调查和修复。

路径 B：实际实现已经正确

不要为了练习改坏正式功能。直接使用下面的固定错误示例作为调查对象：

```
function isTaskOverdue(task, today) {
  const dateValue = task.dueDate || "1970-01-01";
  return task.status !== "已完成" && dateValue < today;
}
```

本书同时给定案例事实：目标任务没有 `dueDate` 字段，页面一方面显示“未设置截止日期”，另一方面错误显示“已逾期”。路径 B 只分析这段代码和案例事实，不把错误示例写入正式项目。

本章采用：

```
稳定复现
→ 收集行为和数据证据
→ 检查控制台与差异
→ 追踪判断路径
→ 确认根因
```

```
→ 最小修复
→ 日期回归
```

二、主体

19.1 写出稳定复现步骤

路径 A 选择一条没有 `dueDate` 字段或日期为空的任务，并在实际页面中执行下面的步骤。

路径 B 不需要在页面制造错误。将起始语中的固定代码和案例事实视为已经获得的行为证据与代码证据，再按照同一结构整理复现记录。

执行：

125. 打开团队任务看板；
126. 找到该任务；
127. 确认卡片显示“未设置截止日期”；
128. 确认它同时错误显示“已逾期”；
129. 刷新页面；
130. 再次观察同一任务。

形成复现记录：

```
故障名称：
无截止日期任务被错误标记为逾期

前置条件：
任务没有 dueDate 字段，或 dueDate 为空字符串。

实际操作：
1. 打开团队任务看板；
2. 找到无日期任务；
3. 查看日期与逾期提示；
4. 刷新后再次查看。

实际结果：
任务显示“未设置截止日期”，同时显示“已逾期”。

预期结果：
没有有效截止日期的任务不得显示“已逾期”。

影响范围：
历史任务和任何未填写日期的新任务。
```

“日期有问题”不是有效复现说明。

复现需要包含：

- 前置数据；
- 操作步骤；

- 实际结果；
- 预期结果；
- 影响范围。

19.2 区分六层调试证据

证据层	当前问题中的内容
用户行为证据	页面同时显示“未设置截止日期”和“已逾期”
数据证据	目标任务的 `dueDate` 缺失或为空
控制台证据	是否出现脚本异常
Git 差异证据	逾期逻辑是怎样加入的
代码路径证据	空值怎样进入逾期函数
回归证据	修复后不同日期类型的实际结果

任何一层都不能单独替代全部证据。

例如：

- 页面异常不能直接证明是哪一行代码；
- Git 差异只能显示最近修改了什么；
- 控制台没有异常不能证明逻辑正确；
- 代码看起来正确，也需要浏览器行为验证。

19.3 明确 Codex 能够取得什么证据

Codex 通常可以直接读取：

- `seed-data.js` 中的任务对象；
- `storage.js` 的读取逻辑；
- `app.js` 中的内存数据流；
- `render.js` 中的逾期函数；
- 当前 Git 差异；
- 终端命令结果。

但它未必能够直接访问：

- 你当前浏览器用户配置中的 `localStorage`；
- 正在打开页面的实际运行时任务对象；

- 浏览器开发者工具中的变量值；
- 你已经完成的点击结果。

因此，调查提示词应区分权限范围。路径 A 根据真实项目执行；路径 B 则把固定错误示例作为已知代码证据，并明确没有实际浏览器运行状态：

```
请只读调查当前无截止日期误判问题，不要修改文件。

请根据代码确认：

1. seed-data.js 中的目标历史任务是否包含 dueDate；
2. storage.js 怎样读取任务；
3. app.js 是否会为旧任务补充 dueDate；
4. render.js 怎样显示日期；
5. 逾期判断怎样处理缺失值和空字符串；
6. 是否存在默认日期；
7. 哪个表达式导致误判。

如果当前环境可以访问浏览器运行状态，请报告实际 dueDate 值。

如果不能访问，请明确说明限制，不要推测运行时值。需要时告诉我应从浏览器复制什么信息。

最后提出最小修复和回归范围，不要修改文件。
```

若 Codex 不能读取运行时数据，可以由用户提供：

```
目标任务的实际数据：
【从浏览器开发者工具或页面调试信息复制任务对象】
```

无法取得的事实必须标记为未知，不能根据代码推测成运行时结论。

19.4 检查控制台和 Git 差异

若具备使用浏览器开发者工具的条件，检查控制台。

如果没有异常，应记录：

> 复现故障时控制台没有脚本异常，说明代码继续执行，问题更可能位于业务判断而不是语法或加载失败。

随后运行：

```
git status --short
git diff -- js/render.js
```

路径 A 查看当前功能分支的真实差异；路径 B 不执行 Git 差异命令，只分析固定错误示例中的默认日期和判断条件。

重点寻找：

- `isTaskOverdue` 或等效函数；

- 空日期怎样处理；
- 是否存在默认日期；
- 日期有效性校验；
- 今天怎样生成；
- 已完成状态怎样排除。

一个典型错误是：

```
const dateValue = task.dueDate || "1970-01-01";
```

它会形成下面的因果链：

```
任务没有 dueDate
→ 使用 1970-01-01 作为默认值
→ 默认日期早于今天
→ 任务状态不是已完成
→ 返回逾期
```

19.5 比较有限的故障假设

假设	需要的证据	当前判断
空日期被替换为早期默认日期	逾期函数存在默认值	与现象高度一致
旧逾期样式没有清除	刷新后仍保留错误类名	可能性较低
加载时自动补充错误日期	运行时任务对象出现日期	需要数据证据
日期格式异常被当成有效日期	缺少有效性校验	需要代码检查

不要同时列出十几种没有证据支持的可能性。

调试应优先验证：

> 哪个假设最能完整解释当前现象。

19.6 完成最小修复

确认根因后发送：

```
请修复“无截止日期任务被错误标记为逾期”的根因。

要求：

1. 只修改实际逾期判断所在文件；
2. dueDate 不存在或为空时，判定为未设置且不逾期；
3. 格式或日历值无效时，判定为日期异常且不逾期；
```

```
4. 有效日期早于本地今天且状态不是已完成时，才判定为逾期；
5. 今天截止不逾期；
6. 不修改历史数据；
7. 不更换 localStorage 键；
8. 不增加任务字段；
9. 不重构其他模块；
10. 不创建 Git 提交。

完成后说明：
- 根因；
- 实际修改；
- 为什么属于最小修复；
- 缺失和无效日期分别怎样处理；
- 需要重新执行哪些测试。
```

修复规则应为：

```
字段不存在或为空
→ 未设置截止日期
→ 不逾期

格式或日历值无效
→ 截止日期异常
→ 不逾期

有效日期
→ 与本地今天比较
→ 早于今天且未完成
→ 逾期
```

不要通过给空日期补一个未来日期来掩盖错误。

19.7 完成日期回归

验证记录中要保存本次测试所使用的本地日期。

至少覆盖：

缺失日期

预期：

- 显示“未设置截止日期”；
- 不显示“已逾期”。

无效日期

例如测试数据为 `2026-02-30`。

预期：

- 显示“截止日期异常”；

- 不显示“已逾期”。

今天到期

记录：

- 当前本地日期；
- 输入的同一天 `dueDate`；
- 任务状态。

预期不逾期。

过去日期

- 待处理：逾期；
- 进行中：逾期；
- 已完成：不逾期。

未来日期

预期不逾期。

还应检查：

- 状态改变后重新计算；
- 刷新后日期和状态保持；
- 搜索和筛选组合仍然正常。

快速抽查：

- 新增任务；
- 删除任务；
- 三个状态列；
- 空状态。

19.8 更新验证记录并锁定候选状态

若 Bug 实际发生在当前功能分支，更新 `docs/due-date-validation.md`：

- 保留原失败记录；
- 补充根因；
- 记录最小修复；
- 写入回归结果；
- 不删除失败历史。

若使用路径 B，在学习记录中增加单独小节：

> 调试练习：无日期误判代码案例

明确区分：

- 案例给定的行为事实；
- 固定错误代码中的根因；
- 建议的最小修复；
- 正式功能分支没有被写入错误代码。

第 19 章结束时应满足：

- 功能分支仍为 `feature/task-due-date`；
- 正式候选代码符合日期规则；
- 同类 Bug 已经通过实际项目修复或固定代码案例分析完成调试；
- 缺失、无效、今天、过去和未来日期均有结果；
- 原有功能回归完成；
- 所有正式变更仍未提交。

三、总结

本章没有依靠猜测修复问题，而是建立了完整证据链：

- 稳定复现异常；
- 区分行为、数据、控制台、差异和代码证据；
- 明确 Codex 能够访问的运行时边界；
- 检查任务的真实日期状态；
- 追踪逾期判断路径；
- 确认空值默认日期或等效逻辑是根因；
- 使用前置判断完成最小修复；
- 区分缺失日期与异常日期；
- 补齐本地今天和其他日期回归；
- 保留失败和修复历史。

本章掌握检查

- [ ] 我写出了稳定复现步骤；
- [ ] 我区分了不同层次的调试证据；
- [ ] Codex 无法访问的运行时事实没有被猜测；
- [ ] 根因来自数据、差异和代码路径；
- [ ] 修复只涉及必要判断；

- [ ] 缺失和异常日期均有明确行为；
- [ ] 回归记录包含具体本地测试日期；
- [ ] 当前正式变更仍未提交。

四、结束语与引导

截止日期功能和日期兼容问题已经通过验证。

这些变化仍然只存在于本地功能分支中。

下一章将完成第一次正式协作流程：

> **整理差异、创建聚焦提交、推送分支、创建 PR、处理审查意见并合并到主分支。**

第 20 章：从代码修改到提交和合并请求

一、起始语

第 17—19 章已经完成：

```
读懂团队任务看板
        ↓
增加截止日期与逾期提示
        ↓
验证并调试日期边界
```

当前状态是：

- 位于 `feature/task-due-date`；
- 功能与日期边界已经验证；
- 验证记录已经更新；
- 工作区中仍存在尚未提交的变化；
- 主分支 `main` 仍保持导入基线；
- 尚未创建 PR。

本章不再增加业务功能。

唯一任务是：

> **把已经验证的变更整理成第一次可审查的 Pull Request，并完成合并和同步。**

Pull Request 用于提出将一个分支中的变化应用到目标分支。创建 PR 时，base 分支是接收变化的分支，head 分支是包含拟议变化的分支。GitHub Docs

本书标准练习使用：

- 一个空的 GitHub 远程仓库；
- `main` 作为目标分支；
- `feature/task-due-date` 作为功能分支；
- **Create a merge commit** 作为标准合并方式。

账号注册、非空仓库接入、企业权限和高级分支保护不在本书展开。实际使用时，应遵循 GitHub 当前页面提示和所在组织的仓库治理要求。

二、主体

20.1 提交前完成范围审查

运行：

```
git branch --show-current
git status --short
git diff --check
git diff --stat
```

当前分支必须是：

`feature/task-due-date`

预计变化包括：

- `index.html`
- `css/styles.css`
- `js/app.js`
- `js/render.js`
- `README.md`
- `docs/manual-test-checklist.md`
- `docs/due-date-validation.md`

不应出现：

- `constants.js`
- `storage.js`
- `filters.js`
- `seed-data.js`
- `package.json`
- 构建产物
- 浏览器测试数据
- 临时调试文件

继续查看完整差异：

```
git diff
```

确认：

- 新增字段只有 `dueDate`；
- 逾期仍然是派生结果；
- 缺失日期和异常日期都不逾期；
- 今天截止不逾期；

- 已完成任务不逾期；
- 没有迁移或清空历史数据；
- 文档与实际实现一致；
- 验证记录保留真实测试结果。

20.2 让 Codex 完成提交前审查

```
请对当前 feature/task-due-date 分支的未提交变化进行只读审查。

不要修改文件，也不要创建提交。

请检查：

1. 实际变化是否只围绕截止日期与逾期提示；
2. 是否存在任务外重构；
3. 历史任务没有 dueDate 时是否兼容；
4. 无效日期是否被识别为异常；
5. 本地今天是否通过本地年、月、日生成；
6. 是否错误使用 new Date(dueDate)或 UTC 日期比较；
7. 今天截止和已完成任务是否正确；
8. 是否改变存储键或原有字段；
9. README、测试清单和验证记录是否一致；
10. 是否残留调试代码或临时输出；
11. 是否适合进入提交。

请将阻断问题和非阻断建议分开。
```

若出现阻断问题，应回到第 19 章完成最小修复和重新验证。

非阻断建议不应自动扩大本次 PR 范围。

20.3 创建聚焦提交

将明确文件加入暂存区：

```
git add index.html css/styles.css js/app.js js/render.js README.md docs/manual-test-checklist.md docs/due-date-validation.md
```

检查暂存区：

```
git status
git diff --staged --stat
git diff --staged
```

确认暂存内容属于同一项业务成果后，创建提交：

```
git commit -m "增加任务截止日期和逾期提示"
```

查看当前分支：

```
git status
git log --oneline --decorate -3
git diff --stat main...HEAD
git log --oneline main..HEAD
```

提交应集中表达：

- 新增可选截止日期；
- 正确显示逾期；
- 兼容缺失和异常日期；
- 更新相关说明与验证记录。

20.4 确认空远程仓库并推送

本书标准远程仓库必须满足：

- 仓库中没有预先创建 README；
- 没有许可证初始提交；
- 没有其他默认分支提交；
- 当前用户拥有推送权限；
- `origin` 指向本次练习仓库。

检查远程：

```
git remote -v
```

若没有 `origin`，先在 GitHub 创建一个属于自己的空练习仓库，再按照页面给出的命令添加 `origin` 并首次推送。执行前必须核对远程地址，不要连接公司正式仓库或他人的项目。

首次推送本地主分支：

```
git switch main
git push -u origin main
```

再推送功能分支：

```
git switch feature/task-due-date
git push -u origin feature/task-due-date
```

若远程仓库已经存在提交，不要直接照抄这组命令，应先处理远程历史、目标分支和权限关系。

20.5 创建范围清楚的 PR

在 GitHub 中创建：

```
base: main
head: feature/task-due-date
```

PR 标题：

> 增加任务截止日期和逾期提示

PR 说明保留四部分。

```
### 背景

团队任务目前缺少明确的截止日期信息。

### 修改内容

- 新增可选 dueDate 字段；
- 新增日期输入和日期显示；
- 过去日期且任务未完成时显示“已逾期”；
- 缺失日期显示“未设置截止日期”；
- 无效日期显示“截止日期异常”；
- 今天截止和已完成任务不标记为逾期；
- 保持历史任务和原存储键兼容；
- 更新 README、测试清单和验证记录。

### 验证结果

- 无日期历史任务：通过；
- 无效日期：通过；
- 未来日期：通过；
- 今天截止：通过；
- 过去日期未完成：通过；
- 过去日期已完成：通过；
- 状态变化后重新计算：通过；
- 搜索和筛选回归：通过；
- 刷新恢复：通过。

### 风险与限制

- 未增加日期排序、筛选、通知或日历；
- 日期按用户本地日历日判断；
- 截止日期仍为可选字段；
- 当前仓库未配置自动化测试时，证据主要来自静态差异检查和人工浏览器测试。
```

只能写入实际完成和验证的内容。

20.6 检查 PR 证据和自动检查

PR 创建后重点检查：

- PR 目标分支是否为 `main`；

- 文件差异是否符合批准范围；
- 提交是否聚焦；
- 验证记录是否与 PR 说明一致；
- 是否存在未解决的讨论；
- 自动检查实际处于什么状态。

状态检查通常由外部流程生成，例如构建、测试或代码扫描。若仓库配置了必需状态检查，它们需要满足规则后才能合并到受保护分支。若仓库没有配置检查，PR 中可能没有相应结果，不能写成“自动检查通过”。GitHub Docs

本练习项目没有配置 CI 时，应如实记录：

> 当前仓库未配置自动检查，主要证据来自 Git 差异检查和人工浏览器测试。

20.7 处理一条范围内审查意见

假设审查者提出：

> README 没有明确说明无日期任务不参与逾期判断。

这条意见：

- 与当前功能直接相关；
- 不增加业务范围；
- 可以通过局部文档修改解决。

在功能分支修改 README，增加：

> 未设置截止日期的任务不参与逾期判断。

检查差异：

```
git diff -- README.md
```

创建补充提交：

```
git add README.md
git commit -m "补充无截止日期规则说明"
git push
```

PR 会自动包含新提交。

不要借审查意见顺手进行：

- 页面视觉重做；
- 日期筛选；

- 文件重命名；
- JavaScript 全面重构；
- 测试体系扩张。

审查意见同样受原任务边界约束。

20.8 使用标准合并提交完成 PR

GitHub 仓库可以允许不同 PR 合并方式，包括创建合并提交、压缩合并和变基合并。不同方式会产生不同的提交历史。GitHub Docs

为了让零基础读者清楚看到功能分支进入 `main` 的关系，本书标准练习固定选择：

> **Create a merge commit**

确认：

- PR 没有阻断问题；
- 人工验证完成；
- 文档和代码一致；
- 审查意见已经处理；
- 必需检查已经通过，或仓库确实没有配置检查；
- 最新提交已经推送；
- base 为 `main`；
- head 为 `feature/task-due-date`。

随后在 GitHub 中选择：

> Merge pull request → Create a merge commit → Confirm merge

GitHub 也可能允许 Squash 或 Rebase，但本书主线不使用这两种方式。GitHub Docs

若仓库设置不允许创建合并提交，应停止执行后续分支删除步骤，并在学习记录中说明实际合并方式。

20.9 同步主分支并清理分支

PR 合并后，本地主分支不会自动更新。

运行：

```
git switch main
git pull --ff-only
```

检查：

```
git status
git log --oneline --decorate -5
```

确认：

- 本地 `main` 已经包含截止日期功能；
- 能看到 PR 产生的合并关系；
- 工作区干净；
- 页面仍然能够运行。

在 GitHub 的 PR 页面删除远程功能分支，或记录远程分支尚待清理。

随后安全删除本地功能分支：

```
git branch -d feature/task-due-date
```

`git branch -d` 要求目标分支已经完整合并到其上游分支，或在没有上游时已经合并到当前 `HEAD`。这也是本书固定使用合并提交方式的重要原因之一。Git

最后检查：

```
git branch
git status
```

标准练习最终应满足：

- 当前位于 `main`；
- 本地功能分支已经安全删除；
- 远程功能分支已删除或明确记录；
- 工作区干净；
- PR 链接或编号已经保存到学习笔记。

若使用了 Squash 或 Rebase 合并，`git branch -d` 可能因无法确认原分支提交已经合并而拒绝删除。本章不使用 `git branch -D` 强制处理；出现这种情况时先保留分支，核对远程合并方式和提交关系后再决定是否删除。

三、总结

本章没有继续增加功能，而是将第 18—19 章的候选变更转化为可审查的协作成果。

你已经完成：

- 审查功能分支的完整差异；
- 让 Codex 完成提交前检查；

- 创建聚焦提交；
- 推送主分支和功能分支；
- 创建范围清楚的 PR；
- 区分人工验证和自动检查；
- 根据审查意见进行局部修订；
- 使用标准合并提交完成 PR；
- 同步本地主分支；
- 安全清理功能分支；
- 保存 PR 记录。

本章掌握检查

- [ ] PR 只包含截止日期功能及相关修复；
- [ ] 提交和 PR 标题准确描述成果；
- [ ] PR 说明只包含真实验证结果；
- [ ] 未配置的自动检查没有被写成通过；
- [ ] 审查意见没有导致范围扩张；
- [ ] PR 使用标准合并提交方式进入 `main`；
- [ ] 本地主分支已经同步；
- [ ] 最终工作区保持干净。

四、结束语与引导

第五篇完成了一个完整的陌生项目修改闭环：

```
读懂已有代码库
→ 建立项目地图
→ 在功能分支中完成跨文件修改
→ 根据证据处理边界 Bug
→ 整理提交
→ 创建和审查 PR
→ 合并并同步主分支
```

团队任务看板不会在这里封存。

第 21—24 章将继续使用同一项目，学习重点升级为：

- 用 `AGENTS.md` 固定项目规则；
- 用最小知识包保持长期上下文；
- 用计划文件和独立工作目录推进长任务；
- 用浏览器与本机环境形成验证证据。

但在进入第 21 章前，必须完成两道阶段门：

131. 第 17—20 章第五篇统一复核；
132. 第 1—20 章 `V1.5M` 中段 Review。

第 21 章：用 AGENTS.md 建立项目规则与运行边界

一、起始语

第 20 章结束时，团队任务看板已经完成第一次正式功能协作：

- 截止日期和逾期提示已经进入 main；
- PR 已经合并；
- 本地主分支已经同步；
- 功能分支已经清理；
- 工作区保持干净。

到目前为止，你已经多次在任务提示中重复说明：

- 项目没有第三方依赖；
- 任务数据保存在浏览器本地；
- 现有存储键不能随意改变；
- 历史任务可能没有 dueDate；
- 逾期是实时计算结果，不是新的任务状态；
- 用户输入不能通过不安全方式直接写入页面；
- 完成修改后必须检查差异并进行浏览器回归。

这些规则如果只存在于某一次聊天中，新聊天不会自然继承；换一个人接手，也可能不知道哪些约束已经被验证。

本章只做一件事：

> **在项目根目录建立一份短小、真实、能够被 Codex 自动读取的 AGENTS.md。**

OpenAI 当前将 AGENTS.md 用于保存会随仓库长期存在的项目指导。Codex 开始工作时会读取适用的指导文件；离当前工作目录更近的项目级规则会在指令链中具有更高优先级。OpenAI Developers：Custom instructions with AGENTS.md

本章不配置复杂权限，不建立多层规则体系，也不重新教学任务卡和 Git 流程。

二、主体

21.1 先区分“项目规则”和“当前任务”

AGENTS.md 适合保存：

- 每次进入仓库都成立的项目事实；
- 多数修改都必须遵守的技术边界；
- 固定的运行与验证方法；
- 经常重复出现的禁止事项；
- 团队稳定使用的提交与审查要求。

它不适合保存：

- “今天把编辑功能做完”之类的一次性目标；
- 某个 Bug 尚未证实的猜测；
- 只在当前聊天中使用的临时步骤；
- 尚未决定的未来功能；
- 大段产品介绍和完整需求文档；
- 密钥、账号、客户数据和内部凭据。

可以用下面的判断方法：

> **如果下一项完全不同的任务也必须遵守这句话，它才可能属于 `AGENTS.md`。**

例如：

内容	是否写入 AGENTS.md	原因
不改变现有本地存储键	是	多数数据修改都要遵守
本周开发任务编辑功能	否	只属于当前任务
不安装第三方依赖	是	当前项目的长期技术边界
编辑按钮放在卡片右下角	否	属于具体功能设计
修改 JavaScript 后进行浏览器回归	是	稳定验证要求
某条任务今天必须完成	否	业务数据，不是项目规则

`AGENTS.md` 不是把所有已知信息放在一个文件里，而是把**每次都需要重申的规则**从聊天中移出来。

21.2 理解 Codex 怎样发现规则

Codex 可以从不同层级读取指导：

```
个人 Codex 目录中的全局 AGENTS.md
        ↓
仓库根目录中的 AGENTS.md
        ↓
当前子目录中的更具体 AGENTS.md 或 AGENTS.override.md
```

项目级文件会随着仓库共享；子目录中的规则只影响相应目录及其下层。

本书的 `team-task-board` 规模较小，所有业务代码都属于同一套技术边界，因此本章只创建：

```
team-task-board/
└── AGENTS.md
```

暂时不创建：

- 全局`~/.codex/AGENTS.md`；
- `js/AGENTS.md`；
- `docs/AGENTS.md`；
- `AGENTS.override.md`；
- 其他备用文件名。

多层规则在大型仓库中有价值，但在小项目里过早拆分会增加冲突和维护成本。

需要记住一个边界：

> Codex 读取了规则，不等于规则一定正确，也不等于代码自动符合规则。

规则仍需经过人工核对，并尽量由实际检查、测试和审查来验证。

21.3 只从当前真实项目提取规则

创建文件前，先让 Codex 只读核对当前状态：

```
请只读检查当前 team-task-board 项目，不要修改文件。

请根据真实代码、README、测试清单和 Git 状态，整理一份候选项目规则，分为：

1. 项目目的；
2. 主要目录与文件职责；
3. 当前运行方式；
4. 任务数据的不变量；
5. 存储和兼容边界；
6. 安全与范围边界；
7. 修改后的验证要求；
8. Git 操作边界。

每条规则请标明依据来自代码、文档还是当前 Git 状态。
无法确认的内容不要写成规则。
不要创建 AGENTS.md。
```

审查候选规则时，删除三类内容：

过于笼统的口号

例如：

> 保持代码高质量。

这句话无法直接检查。可以改成：

> 修改 JavaScript 后执行 `git diff --check`，并按受影响功能完成浏览器回归。

尚未实现的未来目标

例如：

> 所有功能必须有自动化测试。

当前项目还没有自动化测试体系，这条规则会让 Codex 假装存在测试命令，或临时扩张任务范围。

更准确的是：

> 当前项目以静态差异检查和人工浏览器测试为主要证据；未配置的自动化测试不得报告为通过。

依赖猜测的事实

例如：

> `storage.js` 会自动迁移所有历史数据。

如果代码没有这种逻辑，不能因为它“听起来合理”就写入规则。

21.4 创建根目录 `AGENTS.md`

标准练习文件如下：

```
# Team Task Board Project Guidance

## Project scope
- Browser-based task board built with plain HTML, CSS, and JavaScript.
- Keep the current localStorage key and zero-dependency runtime.

## Important files
- `index.html`: page and form structure.
- `css/styles.css`: visual and state styles.
- `js/constants.js`: fixed values and storage key.
- `js/storage.js`: persistent reads and writes.
- `js/filters.js`: search and filters.
- `js/render.js`: safe rendering and derived display state.
- `js/app.js`: initialization, forms, and events.
- `docs/`: rules, plans, tests, and validation records.

## Data invariants
- Statuses remain `待处理`, `进行中`, and `已完成`.
- Preserve existing fields and the storage key unless migration is approved.
- `dueDate` is optional and uses `YYYY-MM-DD`.
- Overdue state is derived; do not persist an `overdue` field.
- Historical tasks without `dueDate` must continue to work.

## Change boundaries
```

```
- Do not add dependencies, replace storage, or perform unrelated refactors without approval.
- Do not add real customer data, credentials, or secrets.
- Render user text with safe DOM APIs, not unsafe HTML strings.

## Verification and Git
- Inspect the code path and expected file scope before editing.
- Run `git diff --check`, review the full diff, and test affected browser flows.
- Report completed and unverified checks separately.
- Do not commit, push, open PRs, rewrite history, or delete branches unless asked.
```

文件可以使用中文，也可以像上面一样使用结构简单的英文。重要的是：

- 规则准确；
- 句子可以执行；
- 没有互相冲突；
- 没有把当前任务写成永久规则；
- 没有包含敏感信息。

OpenAI 的当前实践建议同样强调：一份短小、准确的 `AGENTS.md` 通常比充满模糊要求的长文件更有效；只有出现反复错误时，才逐步补充新规则。OpenAI Developers：Best practices

21.5 检查规则是否能够被读取

创建文件后，先查看差异：

```
git status --short
git diff -- AGENTS.md
git diff --check
```

随后开启一个新的 Codex 任务，要求它只报告规则来源：

```
请不要修改任何文件。

请说明：
1. 当前项目加载了哪些项目级指导文件；
2. 每个文件适用于什么范围；
3. 当前必须遵守的五条最重要规则；
4. 哪些 Git 操作需要我明确授权；
5. 当前项目是否存在自动化测试或 CI 证据。

只根据实际加载的指导和仓库事实回答。
```

重点核对：

- 是否读取到仓库根目录的 `AGENTS.md`；
- 是否把“现有存储键不变”解释正确；
- 是否知道历史任务可以没有 `dueDate`；
- 是否没有凭空声称存在自动化测试；

- 是否知道提交、推送和 PR 需要明确要求。

如果 Codex 没有读取到文件，优先检查：

- 文件名是否确实为 `AGENTS.md`；
- 文件是否位于当前仓库根目录；
- 新任务是否从正确项目启动；
- 文件是否为空；
- 当前工作目录是否在另一个仓库中。

不要通过重复粘贴全部规则来掩盖发现问题。

21.6 处理规则冲突和规则膨胀

假设未来在 `docs/`目录增加更具体的指导：

```
team-task-board/
├── AGENTS.md
└── docs/
    └── AGENTS.md
```

当任务位于 `docs/`目录时，Codex 可能同时读取根规则和更具体的文档规则；更接近当前目录的指导会在合并后的指令链中靠后，从而覆盖前面的冲突内容。

因此，不能写出这样的冲突：

根目录：

> 所有文档都使用中文。

`docs/AGENTS.md`：

> 所有文档都使用英文。

除非这种覆盖是明确、有意并能解释的。

本书当前只保留一份根规则，后续满足以下条件时才考虑增加子目录规则：

- 同一类修改连续出现两次相同错误；
- 不同目录确实使用不同运行或验证方法；
- 根文件已经因为目录差异而变得难以理解；
- 规则可以由目录边界清楚解释。

不要把每次审查意见都立刻写入 `AGENTS.md`。先判断它是：

- 长期规则；
- 一次性任务要求；

- 代码本身应该解决的问题；
- 应由测试或工具强制执行的要求。

21.7 将规则文件纳入可靠版本

本章只新增项目规则，不修改业务功能。

完成最终检查：

```
git status --short
git diff --check
git diff -- AGENTS.md
```

让 Codex 进行一次只读审查：

```
请只读审查根目录 AGENTS.md，不要修改文件。

检查：
1. 是否包含未经代码或文档确认的事实；
2. 是否混入一次性任务；
3. 是否存在无法检查的空泛口号；
4. 是否与当前项目实现冲突；
5. 是否错误声称存在自动化测试；
6. 是否包含凭据、个人路径或敏感数据；
7. 是否足够短，能够在多数任务中持续适用。
```

处理阻断问题后，按照第 20 章已经学过的差异审查和版本保存方法，将 `AGENTS.md` 纳入项目历史。

本章不重新展开完整提交和 PR 流程。

标准练习结束状态为：

- `main` 包含经过审查的 `AGENTS.md`；
- 工作区干净；
- 业务代码没有变化；
- 新 Codex 任务能够读取并复述核心规则。

三、总结

本章把反复出现在任务提示中的项目规则，转化成了仓库内的持久指导。

你已经完成：

- 区分长期规则和一次性任务；
- 理解 `AGENTS.md` 的项目级作用和层级关系；
- 从真实代码和文档提取规则；

- 建立项目目的、数据不变量、修改边界、验证要求和 Git 边界；
- 验证 Codex 能够读取规则；
- 识别规则冲突和规则膨胀风险；
- 将规则文件纳入可靠版本。

本章掌握检查

- [ ] 我能判断一句话是否适合写入 `AGENTS.md`；
- [ ] 文件位于正确的仓库根目录；
- [ ] 规则来自真实实现，不包含未来设想；
- [ ] 数据、存储、安全、验证和 Git 边界清楚；
- [ ] 新任务能够报告已加载的规则；
- [ ] 项目规则没有敏感信息；
- [ ] `main` 工作区最终保持干净。

四、结束语与引导

`AGENTS.md` 解决了一个问题：

> Codex 每次进入项目时，应该遵守什么。

但它不适合承担全部项目知识。

例如：

- 当前系统为什么使用浏览器本地存储；
- 截止日期为什么按本地日历日比较；
- 最近一次 PR 已经完成什么；
- 下一位接手者应从什么状态继续。

这些信息有的长期稳定，有的只在当前阶段有效。

下一章将建立一套克制的最小知识系统：

> **稳定事实、关键决策和当前交接状态各有自己的位置。**

第 22 章：建立最小项目知识与工作交接

一、起始语

第 21 章已经建立根目录 `AGENTS.md`。

现在，Codex 每次进入团队任务看板时，都能够先读取：

- 项目技术边界；
- 数据不变量；
- 修改范围要求；
- 验证规则；
- Git 操作边界。

但 `AGENTS.md` 不应该不断扩张成项目百科。

如果把下面所有内容都塞进去：

- 项目背景；
- 每个文件的完整说明；
- 所有历史决策；
- 最近工作进度；
- 下一项任务；
- 所有测试记录；
- 每次交接说明；

文件会越来越长，真正重要的规则反而难以找到。

本章只建立三类最小知识：

> **项目事实、关键决策、当前状态。**

对应三个文件：

```
docs/
├── project-context.md
├── decision-log.md
└── current-state.md
```

它们不是为了“文档看起来完整”，而是为了让后续任务能够在较少重复解释的情况下，从可信状态继续。

二、主体

22.1 先区分五种信息载体

当前项目中会出现五类常见文件。

文件或位置	主要回答的问题	更新频率
README.md	使用者怎样理解和运行项目	功能或运行方式变化时
AGENTS.md	Codex 每次工作必须遵守什么	稳定规则发生变化时
project-context.md	当前系统实际上怎样组成和运行	架构、字段或边界变化时
decision-log.md	为什么做出关键选择	形成新决策时追加
current-state.md	现在做到哪里，下一步从哪里继续	每个阶段或交接点更新

它们不能互相替代。

例如：

- “任务有三个状态”是项目事实，也可作为数据不变量出现在 `AGENTS.md`；
- “不新增第四个逾期状态”是关键决策，应记录原因；
- “截止日期功能已经合并，下一项准备做任务编辑”属于当前状态；
- “怎样打开页面”应在 README 中让普通使用者看到。

同一事实可以在不同文件中以不同目的出现，但不应大段复制。

22.2 配置文件不负责保存项目知识

Codex 还可以使用 `config.toml` 保存模型、权限、沙箱、功能开关和外部工具等执行偏好。

这些配置与项目知识的职责不同：

```
AGENTS.md
→ 任务应遵守的仓库规则

项目文档
→ 系统事实、决策和当前状态

config.toml
→ Codex 怎样运行、使用什么工具和权限
```

本书当前项目：

- 没有第三方依赖；
- 不需要工作树初始化脚本；

- 没有 MCP 连接；
- 没有多智能体配置；
- 不需要自定义沙箱配置。

因此，本章不为了“看起来专业”而新建`.codex/config.toml`。

只有当项目确实需要共享的运行设置时，才创建仓库级配置；账号、密钥和个人绝对路径不能写入仓库配置。

这个判断很重要：

> **缺少配置文件不代表项目不完整；没有真实需求时增加配置，反而会制造新的维护对象。**

22.3 创建稳定的项目事实文件

`docs/project-context.md`保存当前仍然成立、后续任务会反复使用的事实。

建议结构：

```
# Team Task Board Project Context

## Product and runtime
- 单浏览器环境中的轻量任务管理工具；
- 原生 HTML、CSS 和 JavaScript，无构建步骤和第三方运行依赖；
- 数据保存在当前浏览器配置的 localStorage 中；
- 内置浏览器与普通浏览器可能拥有不同数据；
- 存储键由`js/constants.js`定义，未经批准不得更换。

## Current task fields
`id`、`title`、`owner`、`priority`、`status`、`note`、`createdAt`、可选`dueDate`。

## Core modules
- `index.html`：页面结构与表单；
- `constants.js`：固定值和存储键；
- `storage.js`：读取与保存；
- `filters.js`：搜索和筛选；
- `render.js`：安全渲染和派生显示；
- `app.js`：初始化、表单和事件；
- `styles.css`：页面样式。

## Data flow
localStorage 完整数组 → 内存完整数组 → 搜索筛选结果 → 页面卡片。

## Verification and boundaries
- 主要证据：`git diff --check`、完整差异审查、人工浏览器测试和验证记录；
- 当前未配置自动化测试和 CI；
- 没有账号、云端同步、多人实时协作、日期排序或提醒；
- 当前没有任务编辑功能。
```

这份文件要描述**真实现状**，不把愿望写成现有能力。

例如，当前没有自动化测试，就写“未配置”；不能写“项目通过完整测试体系保障质量”。

22.4 用短记录保存关键决策

项目事实告诉后续人员“现在是什么”，但不能完整解释“为什么这样做”。

`docs/decision-log.md` 只记录会影响后续修改的关键决策。

不需要为每个按钮颜色写一条决策。

标准格式：

```
# Decision Log

## D-001 使用浏览器本地存储
- 状态：已采用
- 决策：继续使用现有 localStorage，不引入数据库和账号系统。
- 原因：当前定位是单浏览器轻量工具。
- 影响：浏览器配置之间不自动同步。
- 重新评估：出现跨设备或多人共享需求。

## D-002 逾期是派生结果
- 状态：已采用
- 决策：不新增“已逾期”状态或`overdue`字段。
- 原因：结果取决于本地日期、截止日期和任务状态。
- 重新评估：需要历史逾期快照或审计记录。

## D-003 截止日期使用本地日历日
- 状态：已采用
- 决策：`dueDate`保存为`YYYY-MM-DD`，不当作 UTC 时间点。
- 重新评估：引入多时区协作或精确到时刻的截止时间。

## D-004 暂不增加第三方依赖
- 状态：已采用
- 决策：当前阶段继续使用原生 HTML、CSS 和 JavaScript。
- 重新评估：原生实现已造成明显重复、风险或维护困难。
```

每条决策至少包含：

- 状态；
- 做出的选择；
- 原因；
- 对后续工作的影响；
- 什么时候需要重新评估。

决策日志不应偷偷改写历史。

如果未来选择发生变化，应：

- 保留原决策；
- 将状态改为“已替代”；
- 新增一条决策说明替代原因。

22.5 建立能够持续覆盖的当前状态

`docs/current-state.md` 用于交接当前项目状态。

它与决策日志不同：

- 决策日志持续追加；
- 当前状态只保留最新可信状态，可以被下一次交接整体更新。

标准内容：

```
# Current Project State

## Baseline
- 当前分支：`main`；工作区：干净；
- 截止日期与逾期提示已经通过 PR 合并；
- 根目录`AGENTS.md`已经生效；
- 运行方式沿用第 17 章固定 URL 和浏览器基线。

## Confirmed capabilities
新增、搜索、筛选、状态更新、删除、可选截止日期、逾期判断和刷新恢复。

## Known limitations
- 不能修改已有任务；
- 没有自动化测试和 CI；
- 内置浏览器与普通浏览器数据可能不同；
- 没有日期排序、提醒、账号或云端同步。

## Next approved task
复用现有表单编辑任务；更新原记录并保留`id`和`createdAt`；支持取消编辑。

## Handoff notes
- 开始前读取规则、项目事实和决策；
- 先维护计划文件，再在工作树中完成候选实现；
- Handoff 到本地后进行浏览器验证；
- 验证完成前不创建 PR。
```

“当前状态”必须有清楚的时间边界。

下一次功能完成后，应更新：

- 最近完成；
- 已确认能力；
- 已知限制；
- 下一项任务；
- 当前分支和工作区。

22.6 让 Codex 按信息类型更新文件

创建三份文档时，不要让 Codex 把所有信息重复写三遍。

可以使用下面的任务：

```
请读取根目录 AGENTS.md、README.md、docs/product-overview.md、现有测试和验证记录，以及当前 Git 状态。

请创建最小项目知识包：

1. docs/project-context.md：只记录稳定的真实系统事实；
2. docs/decision-log.md：只记录会约束后续修改的关键决策及原因；
3. docs/current-state.md：只记录当前基线、已完成能力、已知限制、下一项任务和交接要求。

要求：
- 不修改业务代码；
- 不创建配置文件；
- 不复制大段 README；
- 不把未来目标写成现有能力；
- 不写入个人路径、账号、密钥或真实业务数据；
- 不创建提交。

完成后说明每条重要内容的依据。
```

创建后分别审查：

```
git diff -- docs/project-context.md
git diff -- docs/decision-log.md
git diff -- docs/current-state.md
git diff --check
```

检查是否出现：

- 同一段文字在三个文件中重复；
- 与代码冲突的字段或功能；
- 没有证据的测试结论；
- 过时分支名称；
- 未来功能被写成已经完成；
- 当前任务被写入长期规则。

22.7 建立最小知识路由

`AGENTS.md` 不需要复制三份文档的内容，只需告诉 Codex 在什么情况下读哪一个。

可以在 `AGENTS.md` 增加短小部分：

```
## Project knowledge

- Read `docs/project-context.md` before changes that affect data flow, storage, or multiple modules.
- Read `docs/decision-log.md` before changing an existing product or technical rule.
```

```
- Read `docs/current-state.md` before starting a new feature or taking over unfinished work.
- Update only the documents whose facts, decisions, or current state actually changed.
```

这叫知识路由：

> 主规则文件告诉 Codex 去哪里取得相关上下文，而不是把所有上下文本身塞进主规则文件。

如果一项任务只修改 README 中的错别字，不需要强制读取全部决策日志。

如果一项任务要更换存储方式，则必须读取：

- `AGENTS.md`；
- `project-context.md`；
- `decision-log.md`；
- 相关代码和测试记录。

22.8 验证知识包能否支持交接

启动一个新的只读任务：

```
请把自己当作第一次接手 team-task-board 的维护者。

读取当前适用的 AGENTS.md 和最小项目知识包，然后回答：

1. 项目当前能做什么；
2. 当前任务数据有哪些字段；
3. 哪些规则不能在普通功能任务中改变；
4. 为什么逾期不保存为字段；
5. 当前最明显的功能缺口是什么；
6. 下一项已批准任务是什么；
7. 开始该任务前需要读取哪些文件；
8. 哪些结论仍需通过代码或浏览器确认。

不要修改文件，不要根据常识补充项目能力。
```

答案应当能够区分：

- 文档明确说明的事实；
- 需要代码确认的实现；
- 需要浏览器验证的行为；
- 当前未知的信息。

若 Codex 读完后仍然无法说清下一项任务，通常是 `current-state.md` 不够明确。

若它说出了不存在的功能，通常是项目事实文件混入了未来目标或旧内容。

22.9 将知识包纳入项目历史

本章预计只变化：

- `AGENTS.md` 中的知识路由；
- `docs/project-context.md`；
- `docs/decision-log.md`；
- `docs/current-state.md`；
- 必要时 README 增加文档入口。

完成检查：

```
git status --short
git diff --check
git diff --stat
git diff
```

确认没有业务代码变化后，按照已经学过的版本保存流程，将知识包纳入可靠历史。

标准结束状态：

- 当前位于 `main`；
- 工作区干净；
- `AGENTS.md` 能够路由到三份知识文档；
- 项目事实、关键决策和当前状态互相一致；
- 下一项任务已经锁定为“任务编辑”。

三、总结

本章没有创建庞大的文档体系，只为三类信息安排了清楚位置。

你已经完成：

- 区分 README、项目规则、项目事实、决策和交接状态；
- 理解配置文件不承担项目知识职责；
- 创建稳定的项目上下文；
- 记录会影响后续修改的关键决策；
- 建立可以整体更新的当前状态；
- 在 `AGENTS.md` 中增加最小知识路由；
- 通过新任务验证交接是否成立；
- 将知识包纳入可靠版本。

本章掌握检查

- [ ] 五类信息载体的职责没有混淆；

- [ ] 没有为当前简单项目强行增加配置；
- [ ] 项目上下文只记录真实稳定事实；
- [ ] 决策日志包含原因和重新评估条件；
- [ ] 当前状态明确下一项任务和接手条件；
- [ ] `AGENTS.md` 只保留知识路由，没有复制全部内容；
- [ ] 新任务可以根据文档准确恢复项目状态；
- [ ] `main` 最终保持干净。

四、结束语与引导

团队任务看板现在拥有了两类长期基础：

```
AGENTS.md
→ 每次工作必须遵守的规则

最小项目知识包
→ 真实系统事实、关键决策和当前状态
```

下一项任务已经明确：

> 为已有任务增加编辑能力。

它会同时影响表单模式、任务回填、数据更新、卡片事件、搜索筛选结果和持久化验证，已经不适合只靠一条短提示一次完成。

下一章将第一次系统使用：

> **计划文件和工作树（Worktree）。**

目标不是只完成第一阶段，而是在隔离环境中持续推进，直到形成可以交给本地环境验证的完整候选实现。

第 23 章：用计划文件和工作树推进长任务

一、起始语

前面的任务通常可以在一个连续工作阶段中完成：

- 增加搜索；
- 增加截止日期；
- 修复一个明确 Bug；
- 整理一次 PR。

任务编辑功能更复杂。

它需要同时处理：

- 卡片上的编辑入口；
- 表单从“新增模式”切换为“编辑模式”；
- 将真实任务字段回填到表单；
- 保存时更新原任务而不是新增任务；
- 保留 `id` 和 `createdAt`；
- 取消编辑并恢复新增模式；
- 编辑后重新应用搜索和筛选；
- 刷新后从本地存储恢复修改；
- 相关说明和测试清单更新。

如果中途发生中断，下一次继续时必须知道：

- 已完成什么；
- 哪些决策已经确认；
- 哪些文件发生变化；
- 哪些场景还没有验证；
- 下一步应该做什么。

本章将使用两个工具：

133. **计划文件**：持续记录目标、阶段、决策、进度和未完成事项；
134. **工作树（Worktree）**：在不干扰本地 `main` 的独立检出目录中完成候选实现。

OpenAI 当前将工作树用于在同一个 Git 项目中运行相互隔离的 Codex 聊天。工作树拥有自己的文件检出，但与原仓库共享 Git 元数据；桌面应用创建的工作树默认从选定分支的提交开始，并通常处于分离头指针状态。OpenAI Developers：Worktrees

本章结束时必须得到**完整、可运行、已经完成静态检查的任务编辑候选实现**。

浏览器系统验证放在第 24 章。

二、主体

23.1 判断任务是否需要长计划

并非所有任务都需要计划文件。

适合直接执行的小任务通常具备：

- 修改位置明确；
- 文件数量少；
- 业务规则简单；
- 一次完成后即可验证；
- 中途不需要记录多项决策。

需要长计划的任务通常出现以下特征：

特征	任务编辑中的表现
跨多个模块	表单、事件、渲染、样式、文档
有状态切换	新增模式与编辑模式
有数据不变量	保留 `id` 和 `createdAt`
有异常路径	目标任务不存在、保存失败、取消编辑
有组合行为	编辑后搜索和筛选结果可能变化
可能中断	需要分阶段实施和恢复
需要多轮验证	回填、保存、取消、持久化、回归

第 10 章的短计划回答“准备修改哪些文件和步骤”。

本章的计划文件还需要承担：

- 长任务的单一事实来源；
- 阶段完成记录；
- 实施中发现的新信息；
- 关键选择与原因；
- 下一次恢复工作的入口。

OpenAI 关于执行计划的示例将长计划描述为可以持续更新的“活文档”，要求在进展、发现和决策变化时同步维护，使后续工作能够仅依靠当前计划和仓库状态继续。OpenAI Cookbook：Using PLANS.md for multi-hour problem solving

23.2 冻结任务编辑规则

在创建工作树前，先锁定本次范围。

必须完成

135. 每张任务卡出现“编辑”按钮；
136. 点击后复用现有表单进入编辑模式；
137. 表单回填 `title`、`owner`、`priority`、`status`、`note` 和 `dueDate`；
138. 保存时更新原任务；
139. 原任务的 `id` 和 `createdAt` 保持不变；
140. 取消编辑时不保存变化，并恢复新增模式；
141. 编辑目标不存在时停止保存并显示清楚提示；
142. 存储写入失败时不得更新内存并报告成功；
143. 保存成功后重新应用当前搜索和筛选；
144. 刷新页面后编辑结果继续存在。

明确不做

- 不做卡片内联编辑；
- 不增加弹窗；
- 不增加编辑历史、撤销和审计日志；
- 不改变存储键；
- 不批量迁移历史任务；
- 不新增第三方依赖；
- 不重构无关模块；
- 不在本章创建 PR 或合并到 `main`。

交互规则

新增模式显示“新增任务”和“保存任务”，且不显示取消编辑按钮。

编辑模式显示“编辑任务”和“保存修改”，显示取消编辑按钮，并由当前 `editingTaskId` 确定保存目标。

编辑状态只存在于当前页面内存中，不写入任务数据和 `localStorage`。

23.3 从干净主分支创建工作树任务

开始前在本地检出中确认：

```
git switch main
git status
git log --oneline -3
```

必须满足：

- 当前分支为 `main`；
- 工作区干净；
- `AGENTS.md` 和最小知识包已经进入历史；
- 最近提交包含第 22 章的交接状态。

在 ChatGPT 桌面应用中：

145. 进入当前项目的 Codex 视图；
146. 新建聊天；
147. 在输入区域选择 `Worktree`；
148. 起始分支选择 `main`；
149. 不携带本地未提交变化；
150. 提交第一条只读任务。

```
请在当前工作树中只读检查项目。

开始前请读取：
- 根目录 AGENTS.md；
- docs/project-context.md；
- docs/decision-log.md；
- docs/current-state.md。

然后确认：
1. 当前工作树基于哪个提交；
2. 当前是否处于分离 HEAD；
3. 本地 main 是否会被当前修改直接影响；
4. 下一项批准任务和明确非目标；
5. 预计涉及的文件；
6. 当前项目怎样运行和验证。

不要修改文件。
```

工作树不是项目副本的独立历史。

它与本地检出：

- 文件目录不同；
- 工作区变化互不直接覆盖；
- 共享同一个仓库的提交和分支信息；
- 同一分支不能同时在两个工作树中被检出。

桌面应用默认创建的工作树通常处于分离 HEAD，因此不要假设自己已经位于 `feature/task-editing` 分支。

23.4 创建并审查活计划文件

让 Codex 先创建：

`docs/plans/task-editing.md`

初始任务：

```
请为任务编辑功能创建 docs/plans/task-editing.md。

计划必须自包含，并包含：
1. 用户能够获得的结果；
2. 当前项目基线；
3. 业务规则和非目标；
4. 预计修改文件及原因；
5. 数据和状态不变量；
6. 四个实施阶段；
7. 每阶段完成标准；
8. 验证场景；
9. 风险、未知项和停止条件；
10. 进度记录、发现记录和决策记录。

本轮只创建计划，不修改业务代码，不创建提交。
```

建议计划骨架：

```
# Task Editing Execution Plan

## Purpose
让使用者从任务卡进入编辑模式，修改已有任务并保存到原记录。

## Baseline
- 起始提交：记录实际哈希；起始分支：`main`；
- 环境：Codex 工作树；
- 已有能力：新增、搜索筛选、状态更新、删除、截止日期和逾期提示；
- 当前缺口：不能编辑已有任务。

## Invariants
- 保留`id`、`createdAt`和现有 localStorage 键；
- 不持久化编辑状态；
- 存储失败时不更新内存状态；
- 历史任务缺少`dueDate`时继续兼容。

## Milestones
- [ ] M1 影响分析和编辑状态设计
- [ ] M2 表单模式与字段回填
- [ ] M3 原任务更新、异常处理和重新渲染
- [ ] M4 文档、静态检查和候选状态整理

## Living records
- Discoveries：由真实代码确认的新信息；
- Decisions：影响实现方式的选择及原因；
- Remaining verification：第 24 章需要在浏览器完成的场景。
```

审查计划时重点检查：

- 是否把浏览器验证误写成已经完成；
- 是否包含未批准的重构；
- 是否漏掉存储失败和目标不存在；
- 是否说明 `id`、`createdAt` 和存储键不变；
- 是否每个阶段都有可以观察的结束条件；
- 是否能够在中断后只读计划继续。

计划通过前，不修改业务代码。

23.5 阶段一：建立编辑状态和影响边界

先让 Codex 根据真实代码确认：

- 表单提交入口；
- 任务卡事件处理方式；
- 保存函数的返回约定；
- 当前内存任务数组在哪里维护；
- 渲染函数怎样接收动作回调；
- 表单清空和提示信息怎样实现。

```
请执行计划 M1，只完成影响分析和编辑状态设计。

需要明确：
1. `editingTaskId`或等效状态放在哪里；
2. 编辑按钮事件怎样从任务卡传回 app.js；
3. 怎样找到原任务并回填表单；
4. 保存时怎样区分新增和编辑；
5. 怎样保留 id 和 createdAt；
6. 目标不存在或存储失败时怎样停止；
7. 取消编辑怎样恢复新增模式；
8. 哪些文件保持不变。

更新计划中的发现、决策和进度。
除计划文件外暂不修改业务代码。
```

一个克制的设计通常是：`app.js` 维护 `editingTaskId`；`render.js` 为卡片提供编辑按钮并传回 `task.id`；`app.js` 再从完整任务数组找到原任务、回填现有表单，并在提交时根据 `editingTaskId` 选择新增或更新路径。

`editingTaskId` 属于页面交互状态，不应写入每条任务对象。

阶段一结束标准：

- 计划已记录真实函数和文件；
- 状态所有权明确；

- 数据不变量明确；
- 未修改业务代码。

23.6 阶段二：完成表单模式和字段回填

批准阶段一后进入 M2：

```
请执行计划 M2：完成任务编辑入口、表单模式和字段回填。

要求：
1. 每张任务卡增加范围清楚的编辑按钮；
2. 点击编辑后，根据真实任务 id 找到完整任务；
3. 回填 title、owner、priority、status、note 和 dueDate；
4. 将表单标题和主按钮切换为编辑状态；
5. 显示取消编辑按钮；
6. 取消时清空编辑状态和表单，并恢复新增模式；
7. 本阶段不实现最终更新写入；
8. 不修改存储键，不安装依赖，不重构其他模块；
9. 更新计划进度、发现和剩余问题；
10. 不创建提交。
```

阶段二可做静态检查：

```
git status --short
git diff --check
git diff --stat
```

此时不应宣称“任务编辑已经完成”。

阶段二只证明：

> 用户能够进入和退出编辑模式，字段回填逻辑已经建立。

23.7 阶段三：更新原任务并处理失败路径

M3 是功能成立的核心。

保存编辑时必须形成候选任务：

```
const updatedTask = {
  ...originalTask,
  title: cleanedTitle,
  owner: selectedOwner,
  priority: selectedPriority,
  status: selectedStatus,
  note: cleanedNote,
  dueDate: selectedDueDate
};
```

这里的`...originalTask`用于保留原有：

- `id`；
- `createdAt`；
- 当前版本未显式修改的兼容字段。

但不能只看示例代码，必须根据实际项目的数据清理和保存函数调整。

实施任务：

```
请执行计划 M3：完成原任务更新、持久化和失败处理。

要求：
1. 编辑提交时根据 editingTaskId 找到原任务；
2. 找不到目标任务时不新增记录，显示错误并保留可恢复状态；
3. 构造更新任务时保留原 id 和 createdAt；
4. 使用现有输入清理和日期规则；
5. 先形成候选完整数组，再调用现有保存逻辑；
6. 保存失败时不替换内存任务数组，不报告成功；
7. 保存成功后更新内存、退出编辑模式并重新筛选渲染；
8. 当前搜索或筛选导致任务消失时，按真实结果显示，不绕过筛选；
9. 删除当前正在编辑的任务时，编辑模式必须安全结束，或由计划记录另一种明确处理；
10. 不创建提交，更新计划记录。
```

保存顺序应与前文持久化保护一致：读取并清理表单，找到原任务，构造候选更新任务和候选完整数组；只有 `localStorage` 写入成功后，才替换内存数组、退出编辑模式并重新筛选渲染。

不能先改变内存，再在写入失败后假装已经保存。

阶段三结束标准：

- 保存更新的是原记录；
- `id` 和 `createdAt` 不变；
- 失败路径不伪装成功；
- 搜索和筛选继续使用完整数组；
- 候选功能已经具备完整代码路径。

23.8 阶段四：整理候选实现和交接信息

最后完成 M4：

```
请执行计划 M4，整理任务编辑候选实现。

1. 更新 README 中的任务编辑说明；
2. 更新 docs/manual-test-checklist.md，加入回填、保存、取消、持久化、筛选组合和失败路径；
3. 更新 docs/current-state.md，明确候选实现位于工作树、尚未完成浏览器系统验证；
4. 完整检查 AGENTS.md 要求；
5. 运行 git diff --check；
6. 审查完整差异和实际文件范围；
7. 更新计划中的全部进度、发现、决策和第 24 章待验证场景；
8. 不创建提交，不推送，不创建 PR。
```

检查预计变化：

- `index.html`；
- `css/styles.css`；
- `js/app.js`；
- `js/render.js`；
- `README.md`；
- `docs/manual-test-checklist.md`；
- `docs/current-state.md`；
- `docs/plans/task-editing.md`。

只有真实依赖证明需要时，才允许扩大到其他文件。

运行：

```
git status --short
git diff --check
git diff --stat
git diff
```

让 Codex 完成候选审查：

```
请只读审查当前任务编辑候选实现。

区分：
- 阻断问题；
- 非阻断建议；
- 必须在浏览器中验证的未知项。

重点检查：
1. 是否更新原任务而不是新增；
2. id 和 createdAt 是否保留；
3. 存储失败是否会污染内存状态；
4. 目标不存在时是否安全停止；
5. 编辑状态是否错误持久化；
6. 搜索筛选是否仍基于完整任务数组；
7. 是否存在任务外重构；
8. 文档是否把未验证结果写成通过。

不要修改文件，不要创建提交。
```

处理代码层面的阻断问题，但不根据非阻断建议扩大范围。

23.9 从工作树移交到本地环境

第 24 章要使用已经建立的本地运行环境和浏览器验证，因此需要将聊天和代码带回本地检出。

在桌面应用聊天顶部选择：

> Hand off → Local

Handoff 会处理在工作树和本地检出之间移动聊天与代码所需的 Git 操作。它适合需要在常用 IDE、本地服务或固定浏览器环境中继续验证的情况。OpenAI Developers：Worktrees

移交后不要假设分支状态。

立即检查：

```
git branch --show-current
git status --short
git diff --stat
```

如果候选变化已经位于本地，但当前仍在 `main` 且变化未提交，先创建功能分支：

```
git switch -c feature/task-editing
```

再次检查：

```
git branch --show-current
git status --short
```

不要在本地和工作树中同时检出同一个功能分支。Git 要求一个分支同一时刻只能由一个工作树占用；如果准备继续在本地工作，应使用 Handoff，而不是在两个目录中强行切换同一分支。

标准结束状态：

- 本地当前分支为 `feature/task-editing`；
- 候选实现完整但尚未提交；
- `main` 没有被直接提交新功能；
- 计划文件显示 M1—M4 已经完成；
- 浏览器场景仍明确标记为待验证；
- 第 24 章可以从固定本地环境开始。

三、总结

本章没有把长任务当作一条更长的提示词，而是建立了可以中断和恢复的推进机制。

你已经完成：

- 判断任务编辑为什么需要长计划；
- 冻结功能范围和非目标；
- 从干净 `main` 创建工作树任务；
- 理解工作树、分离 HEAD 和本地检出的关系；
- 创建并持续更新计划文件；
- 分阶段完成状态设计、表单回填、原任务更新和文档整理；
- 在每个阶段记录发现、决策和未完成事项；

- 完成静态差异审查；
- 通过 Handoff 将完整候选实现移交到本地功能分支。

本章掌握检查

- [ ] 当前任务确实具备长任务特征；
- [ ] 计划文件可以独立说明目标、基线、阶段和进度；
- [ ] 工作树基于干净 `main` 创建；
- [ ] 没有把分离 HEAD 误认为功能分支；
- [ ] 四个阶段均有明确结束条件；
- [ ] 候选实现已经完整，不只完成第一阶段；
- [ ] 浏览器验证没有被提前写成通过；
- [ ] 候选变化已安全移交到本地 `feature/task-editing`。

四、结束语与引导

任务编辑的候选实现已经完成，但“代码路径看起来正确”仍然不等于功能已经成立。

接下来必须在真实页面中确认：

- 点击编辑是否打开正确任务；
- 每个字段是否准确回填；
- 保存后是否仍是原任务；
- 取消是否完全不产生变化；
- 搜索和筛选是否与编辑结果一致；
- 刷新后数据是否保持；
- 控制台是否存在错误；
- 截止日期和原有功能是否回归通过。

下一章只负责一件事：

> **用浏览器、控制台和截图，把候选实现变成有证据的验证结果。**

第 24 章：用浏览器、控制台和截图验证候选功能

一、起始语

第 23 章结束时，任务编辑已经形成完整候选实现。

当前状态是：

- 本地分支为 `feature/task-editing`；
- 候选变化尚未提交；
- 计划文件已经记录实施过程；
- 静态差异检查没有阻断问题；
- README 和人工测试清单已经更新；
- 真实浏览器场景尚未系统验证。

本章不继续增加新功能，也不重新教学 PR 流程。

唯一任务是：

> **在固定运行环境中执行任务编辑场景、控制台检查、截图留证和原功能回归。**

ChatGPT 桌面应用中的内置浏览器可以打开本地页面，检查实际渲染状态、执行点击和输入、截图并验证结果。它使用与普通浏览器分离的浏览器配置，因此不能默认继承你在 Chrome、Edge 或 Safari 中的本地存储数据。OpenAI Developers：Browser

这一差异对当前项目尤其重要，因为任务数据保存在 `localStorage` 中。

二、主体

24.1 固定验证环境和证据边界

先记录验证环境：

项目	实际记录
当前分支	feature/task-editing
当前提交基线	记录实际哈希
工作区	存在未提交候选变化
本地服务 URL	记录实际 URL 和端口
浏览器	ChatGPT 桌面应用内置浏览器

项目	实际记录
浏览器配置	内置浏览器独立配置
验证日期	实际本地日期
初始任务数据	记录使用的虚构任务
自动化测试	当前未配置
控制台能力	是否启用浏览器开发者模式

内置浏览器不会自动共享普通浏览器的：

- 标签页；
- 登录状态；
- Cookie；
- localStorage；
- 浏览历史。

因此，不要写：

> 第 17 章创建的任务应该已经在内置浏览器里。

应先实际打开页面，确认：

- 当前内置浏览器是否已经存在项目数据；
- 是否触发示例数据初始化；
- 当前使用的存储键；
- 初始任务数量；
- 测试数据是否全部为虚构内容。

如果需要对普通浏览器中的旧数据进行回归，应在普通浏览器单独测试，并将两组结果分开记录。

24.2 启动本地页面

沿用第 17 章已经确认的运行方式。

如果标准练习使用本地 HTTP 服务，可在项目根目录运行其中一条：

Windows：

```
py -m http.server 8000
```

macOS：

```
python3 -m http.server 8000
```

打开：

```
http://localhost:8000/
```

若端口已被占用，使用另一个明确端口，并在验证记录中保存真实 URL。

不要同时启动多个来源不明的项目服务，否则可能在错误目录中验证旧代码。

检查：

```
git branch --show-current
git status --short
```

页面打开后确认：

- 标题和任务列正常；
- 新增表单正常；
- 截止日期显示正常；
- 每张任务卡存在编辑入口；
- 页面没有明显脚本加载失败。

24.3 让 Codex 执行有边界的浏览器任务

在桌面应用中启用 Browser 能力后，可以提交：

```
请使用@Browser 打开当前本地任务看板 URL。

只执行任务编辑功能验证，不修改代码。

开始前：
1. 确认实际 URL；
2. 记录内置浏览器当前任务数量；
3. 说明内置浏览器使用独立配置，不能假设普通浏览器数据存在；
4. 使用虚构测试数据；
5. 每个场景记录前置状态、操作、实际结果和预期结果；
6. 在关键状态截图；
7. 遇到失败时停止当前场景并记录，不要自动扩大修改范围。
```

浏览器任务应保持足够窄。

不要一次要求：

- 测试全部功能；
- 自动修复所有问题；
- 重做页面样式；
- 提交代码；
- 创建 PR；

- 登录外部系统。

本章采用“一个场景、一组明确数据、一个实际结果、一份证据”的方式推进。

24.4 场景一：打开编辑并准确回填

选择一条同时包含：

- 标题；
- 负责人；
- 优先级；
- 状态；
- 备注；
- 截止日期；

的虚构任务。

先记录原任务：

```
任务 id：
标题：
负责人：
优先级：
状态：
备注：
createdAt：
dueDate：
```

操作：

151. 点击该卡片“编辑”；
152. 观察表单标题；
153. 观察主按钮文字；
154. 检查六个可编辑字段；
155. 确认出现取消按钮；
156. 暂不保存。

预期：

- 表单进入编辑模式；
- 所有字段来自同一条目标任务；
- 空备注或空日期不会被其他任务的值污染；
- 表单不会新增第二条记录；
- 页面没有控制台异常。

截图证据建议标记为：

> E01｜任务卡编辑入口与目标任务
>
> E02｜编辑表单字段完整回填

如果当前界面将截图保存在聊天而不是项目文件中，应在验证记录中保存截图标签和聊天位置，不要虚构项目内文件路径。

24.5 场景二：保存后更新原任务

在编辑模式中修改：

- 标题；
- 负责人；
- 优先级；
- 状态；
- 备注；
- 截止日期。

保存前记录任务总数。

操作：

157. 保存修改；
158. 检查页面提示；
159. 查找修改后的任务；
160. 检查任务总数；
161. 读取页面或开发者模式能够取得的任务数据；
162. 比较 `id` 和 `createdAt`；
163. 刷新页面后再次查找。

预期：

- 任务总数不增加；
- 原任务内容被更新；
- `id` 保持不变；
- `createdAt` 保持不变；
- 修改后的 `dueDate` 按现有日期规则显示；
- 编辑模式退出并恢复新增模式；
- 刷新后结果仍然存在。

截图证据：

> E03｜保存修改后的任务卡

> E04｜刷新后修改仍然存在

如果无法直接读取运行时对象，应明确记录：

> 页面行为已经验证，但 `id` 和 `createdAt` 需要通过开发者模式、临时只读调试输出或代码路径证据确认。

不能因为任务总数没变，就直接声称两个字段一定保持。

24.6 场景三：取消编辑不产生变化

选择另一条任务并记录原值。

操作：

164. 点击编辑；
165. 修改多个字段；
166. 点击取消编辑；
167. 检查表单恢复新增模式；
168. 查找原任务；
169. 刷新页面；
170. 再次检查原任务。

预期：

- 未保存的表单变化不会进入任务卡；
- 未写入 localStorage；
- 任务总数不变；
- 表单恢复初始状态；
- 下一次点击“保存任务”会新增任务，不会继续修改旧任务。

这个场景必须验证最后一项。

若取消后 `editingTaskId` 仍然残留，下一次新增可能错误覆盖旧任务。

截图证据：

> E05｜取消后表单恢复新增模式

24.7 场景四：搜索和筛选与编辑结果一致

编辑可能改变当前显示条件。

例如：

- 当前搜索关键词为“客户”；
- 当前只显示负责人“林晓”；
- 正在编辑的任务原本匹配；
- 保存时将标题改为不含“客户”，或将负责人改为其他人。

操作：

171. 设置一个明确搜索或筛选条件；
172. 对当前可见任务进入编辑；
173. 修改使其不再匹配当前条件；
174. 保存；
175. 观察当前结果；
176. 清除条件后重新查找任务。

预期：

- 保存成功；
- 当前卡片可以从筛选结果中消失；
- 页面不应为了“让用户看到保存结果”而绕过现有筛选；
- 清除条件后可以找到已更新任务；
- 完整任务数组没有丢失。

截图证据：

> E06｜编辑后任务不再匹配当前筛选
>
> E07｜清除筛选后找到更新任务

这个场景验证的是：

> 编辑更新发生在完整数据层，页面显示仍由当前筛选条件决定。

24.8 场景五：异常路径和状态恢复

目标任务不存在

如果当前项目本身提供受控故障入口，可以验证“编辑目标已经不存在”的恢复行为：

- 保存不会新建任务；
- 页面显示清楚错误；
- 不报告保存成功；
- 用户可以取消并恢复正常状态。

不要为了测试而直接破坏真实任务数据。

本地存储写入失败

如果当前项目本身提供安全、可恢复的存储失败注入方式，可以复用该方式验证：

- 候选数组没有替换内存正式状态；
- 页面没有假装保存成功；
- 原任务仍然存在；
- 用户可以继续操作或重新尝试。

如果当前没有安全、稳定的失败注入方式，应记录：

> 存储失败路径已完成代码审查，本轮未在真实浏览器中强制制造失败。

未执行的测试不能标记为通过。

24.9 使用开发者模式检查控制台和运行时

内置浏览器的开发者模式可以在明确批准后访问受控的 Chrome DevTools Protocol 能力，用于检查控制台、网络、DOM 和样式。该能力可能受到工作区管理员设置限制。OpenAI Developers：Browser Developer mode

启用条件允许时，提交：

```
请使用@Browser 开发者模式，只检查当前本地任务看板。

1. 重复一次打开编辑、保存和取消流程；
2. 报告控制台 error 和未处理异常；
3. 检查保存前后任务总数；
4. 在不修改数据的前提下，确认目标任务的 id 和 createdAt 是否保持；
5. 检查是否存在重复事件监听导致一次点击执行多次；
6. 不检查外部网站，不修改代码。
```

记录：

检查项	结果
页面加载控制台错误	实际结果
打开编辑控制台错误	实际结果
保存修改控制台错误	实际结果
取消编辑控制台错误	实际结果
任务总数	保存前／保存后
id	保存前／保存后
createdAt	保存前／保存后
重复事件	发现／未发现／未验证

如果开发者模式不可用：

- 使用普通开发者工具人工检查；
- 或将相应项目标记为未验证；
- 不要求 Codex 猜测控制台结果。

24.10 完成原功能回归

任务编辑通过后，还要抽查它没有破坏原功能。

新增任务

- 表单处于新增模式；
- 保存后任务总数增加一；
- 刷新后保留。

状态更新

- 不进入编辑模式也能更新状态；
- 状态列正确变化；
- 逾期根据新状态重新计算。

删除任务

- 删除目标正确；
- 删除后任务不再出现；
- 如果删除的是正在编辑的任务，编辑状态安全结束。

搜索和筛选

- 关键词；
- 负责人；
- 优先级；
- 组合条件；
- 无结果状态。

截止日期

- 未设置日期；
- 异常日期；
- 今天；

- 过去日期未完成；
- 过去日期已完成；
- 未来日期。

只记录实际执行的场景。

若某项失败，先判断：

- 与任务编辑直接相关；
- 由测试环境或旧数据造成；
- 属于原有已知问题；
- 需要停止并回到代码调查。

24.11 建立验证报告

创建：

`docs/task-editing-validation.md`

建议结构：

```
# Task Editing Validation

## Environment
- Date / branch / baseline commit:
- URL / browser profile / initial data:
- Developer mode:

## Candidate scope
列出实际变化文件和功能范围。

## Scenario results
| 场景 | 前置条件 | 实际结果 | 结论 | 证据 |
|---|---|---|---|---|
| 编辑回填 | ... | ... | 通过／失败 | E01、E02 |
| 保存原任务 | ... | ... | ... | E03、E04 |
| 取消编辑 | ... | ... | ... | E05 |
| 搜索筛选组合 | ... | ... | ... | E06、E07 |
| 目标不存在 | ... | ... | ... | 已执行／未执行 |
| 存储失败 | ... | ... | ... | 已执行／未执行 |

## Console, regression, and gaps
记录控制台、任务总数、`id`、`createdAt`、原功能回归和未验证事项。

## Final conclusion
允许进入提交前审查／需要最小修正后重测／候选实现不成立。
```

报告中的“通过”必须对应：

- 明确前置状态；
- 实际操作；

- 观察结果；
- 可追踪证据。

24.12 只修复范围内阻断问题

若验证发现阻断问题，例如：

- 保存后任务总数增加；
- 取消后仍然覆盖旧任务；
- `createdAt` 被重新生成；
- 搜索条件下更新了错误任务；
- 保存失败仍然显示成功；
- 一次点击触发两次保存；

按照第 19 章的证据型调试方法处理：先复现并取得页面与运行时证据，再追踪最小代码路径、修复根因，最后重新执行失败场景和相关回归。

不要顺手修改：

- 卡片整体设计；
- 日期排序；
- 提醒功能；
- 动画；
- 文件命名；
- 自动化测试框架。

修复后更新：

- `docs/plans/task-editing.md` 中的发现和决策；
- `docs/task-editing-validation.md` 中的失败与重测记录；
- `docs/current-state.md` 中的实际候选状态。

保留失败历史，不把第一次失败记录删除成“从未发生”。

24.13 锁定验证后的候选状态

最终运行：

```
git branch --show-current
git status --short
git diff --check
```

```
git diff --stat
git diff
```

让 Codex 完成最终只读 Review：

```
请只读审查当前 feature/task-editing 候选变化和验证报告。

检查：
1. 所有代码变化是否围绕任务编辑；
2. 验证报告是否与实际证据一致；
3. 是否把未执行场景写成通过；
4. id、createdAt 和存储键是否保持；
5. 是否存在未处理阻断问题；
6. 文档是否准确区分内置浏览器和普通浏览器数据；
7. 是否适合进入提交前审查。

不要修改文件，不要创建提交，不要推送或创建 PR。
```

验证阶段的候选状态应满足：

- 当前分支为 `feature/task-editing`；
- 任务编辑场景已经得到真实浏览器证据；
- 控制台检查已执行，或明确记录不可用；
- 截图证据有可追踪标签或文件；
- 原功能回归已记录；
- 所有阻断问题已修复并重测，或候选实现被明确判定不通过；
- 计划、验证报告和当前状态一致。

24.14 按已经学过的流程保存验证成果

若最终结论为“允许进入提交前审查”，按照第 20 章已经掌握的流程完成差异审查、聚焦提交、PR 审查、合并和本地同步。本节不重复命令和界面步骤。

本次提交和 PR 只能包含：

- 任务编辑候选实现；
- 任务编辑计划的最终状态；
- 人工测试清单；
- 验证报告和可纳入仓库的证据说明；
- README 与 `current-state.md` 的必要更新。

不能借保存成果增加新功能或重构。截图如果只存在于聊天证据中，不应伪造为仓库文件；在验证报告中保存可追踪标签即可。

合并后更新 `docs/current-state.md`：任务编辑已完成并进入 `main`，功能分支已清理，工作区干净。

第 24 章标准结束状态为：

- `main` 包含已经验证的任务编辑功能；
- 本地主分支已同步；
- 功能分支已按团队规则清理；
- 工作区干净；
- 计划、验证报告和当前状态与最终代码一致。

如果验证结论为不通过，则停止提交与合并，保留功能分支和失败证据，回到阻断问题处理。

三、总结

本章把“看起来完成”的候选实现转化成了可审查的验证结论。

你已经完成：

- 固定分支、URL、浏览器配置、日期和测试数据；
- 区分内置浏览器与普通浏览器的本地数据；
- 验证编辑入口和字段回填；
- 验证保存更新原任务并保持关键字段；
- 验证取消编辑不产生变化；
- 验证编辑与搜索筛选组合；
- 检查异常路径或明确未验证范围；
- 使用控制台和运行时证据补强结论；
- 完成新增、状态、删除、搜索筛选和截止日期回归；
- 建立截图标签和验证报告；
- 对阻断问题进行最小修复和重测；
- 锁定提交前候选状态，并在通过后按既有流程保存到 `main`。

本章掌握检查

- [ ] 验证环境和测试日期已记录；
- [ ] 内置浏览器数据没有与普通浏览器混淆；
- [ ] 保存后任务总数、`id` 和 `createdAt` 有证据；
- [ ] 取消编辑后新增模式真正恢复；
- [ ] 搜索和筛选组合行为已验证；
- [ ] 控制台结果已记录或明确不可用；
- [ ] 截图能够追踪到具体场景；
- [ ] 未执行的场景没有被写成通过；
- [ ] 原有核心功能完成回归；

- [ ] 通过验证的成果已经按第 20 章流程保存，或失败候选被明确停止。

四、结束语与引导

第 21—24 章完成了一个新的长期工作闭环：

```
把稳定规则写入 AGENTS.md
→ 用最小知识包保存事实、决策和当前状态
→ 用计划文件拆分并持续推进长任务
→ 用工作树隔离候选实现
→ Handoff 到本地固定环境
→ 用浏览器、控制台和截图形成验证证据
→ 按既有协作流程保存成果并恢复干净主分支
```

从下一篇开始，学习重点将从“怎样管理一次长期任务”转向：

> **同一个项目怎样在桌面端、命令行、编辑器和云端之间选择合适的工作入口。**

第 25 章首先整理桌面端中的项目和聊天，让不同任务不再混在同一条对话中。

第 25 章：在桌面端组织项目、聊天和运行环境

一、起始语

第 24 章结束时，团队任务看板已经完成任务编辑功能的验证、提交和合并。

当前项目具备：

- 新增、搜索、筛选、更新状态和删除任务；
- 负责人、优先级和截止日期；
- 逾期提示；
- 任务编辑；
- `AGENTS.md` 项目规则；
- 项目背景、决策记录、当前状态和验证报告；
- 干净并已同步的 `main` 分支。

随着任务增加，桌面端中也会逐渐出现多条聊天：陌生项目探索、截止日期功能、Bug 修复、第一次 PR、项目规则、任务编辑计划和浏览器验证。

如果所有工作都继续堆在同一条聊天中，会出现两个问题：一是上下文越来越长，新的维护任务难以迅速找到当前事实；二是不同目标、分支和验证结论容易混在一起。

本章不修改任何项目文件。

唯一任务是：

> **在 ChatGPT 桌面应用中的 Codex 里，把项目、聊天和运行环境整理成可以长期维护的工作入口。**

OpenAI 当前将桌面端定位为协调多个项目、并行聊天和长任务的工作空间；Codex 聊天可以运行在 Local、Worktree 或 Cloud 环境中。OpenAI Developers：ChatGPT desktop app OpenAI Developers：Codex environments

二、主体

25.1 项目、聊天和分支不是同一层对象

初次使用时，读者容易把三个对象混在一起。

对象	它解决什么问题	当前案例中的例子
项目	Codex 正在围绕哪个真实项目工作	`team-task-board` 项目根目录

对象	它解决什么问题	当前案例中的例子
聊天	这一轮要完成哪个目标，保留哪些上下文	“任务编辑功能验证”
分支	Git 如何隔离和保存一组文件变化	feature/task-editing

一个项目可以有多条聊天；一条聊天可能不修改文件，也可能围绕一个分支工作；同一条聊天在需要时还可以通过 Handoff 在 Local 与 Worktree 之间移动。

因此，不要把“新建聊天”理解成“新建项目”，也不要因为新建了聊天就假设 Git 分支已经创建。

每次开始工作，仍要实际确认：

```
git branch --show-current
git status --short
```

聊天标题帮助人找到上下文，Git 状态才说明文件目前在哪里。

25.2 一个真实项目只建立一个主项目入口

团队任务看板的主项目入口应指向真实项目根目录：

```
Codex 学习/projects/team-task-board/
```

不要同时把以下目录都建立成同名主项目：

- 项目根目录；
- docs/子目录；
- 某个临时 Worktree 目录；
- 复制出来的旧版本目录；
- 下载到桌面的压缩包解压目录。

否则，后续很容易在错误副本中修改文件。

打开项目后，可以让 Codex 只读确认：

```
请不要修改文件。请说明当前项目根目录、当前分支、工作区状态，以及 AGENTS.md 和 docs/current-state.md 是否存在。根据实际文件回答。
```

只有根目录、分支和状态与预期一致，才继续工作。

25.3 一项成果建立一条聊天

桌面端可以保留多条项目聊天。更稳妥的划分单位是“一个可以独立验收的成果或问题”。

团队任务看板可以保留：

聊天名称	用途	当前状态
00｜项目状态与下一步	只读查看当前事实和待办	保留
01｜截止日期与逾期提示	第 18—20 章历史成果	已完成，可归档
02｜任务编辑功能	第 23—24 章计划、实现和验证	已完成，可归档
03｜文档口径一致性维护	第 26—27 章的新任务	准备开始
04｜云端贡献者文档	第 28 章云端任务	尚未开始

编号不是强制规则，作用只是让状态聊天和任务聊天容易区分。

不要为每一次小追问都新建聊天。以下内容通常应留在同一任务聊天：

- 对同一差异的补充说明；
- 对同一阻断问题的最小修复；
- 同一任务的重测；
- 同一 PR 中的审查反馈。

当目标、分支或验收标准已经变化，才建立新聊天。

25.4 保留一条只读状态聊天

`00｜项目状态与下一步`不承担开发工作。它只读取：

- `AGENTS.md`；
- `docs/project-context.md`；
- `docs/current-state.md`；
- 最近提交；
- 当前工作区；
- 已知未完成事项。

可以使用：

```
请只读检查当前 team-task-board 项目。

依据 AGENTS.md、docs/project-context.md、docs/current-state.md 和最近 Git 提交，输出：
1. 当前已交付功能；
2. 当前分支和工作区；
3. 最近一次已验证成果；
4. 已知未完成事项；
5. 下一项建议任务。

不要修改文件，不要创建分支或提交。事实不确定时明确标记。
```

状态聊天的价值是帮助人重新进入项目，不应成为另一份事实数据库。若它与仓库文档冲突，应以实际文件和 Git 状态为准，并在真正的维护任务中更新文档。

25.5 根据任务选择 Local、Worktree 或 Cloud

当前桌面端可以为 Codex 聊天选择三类环境：Local 直接使用当前项目目录；Worktree 在本机的 Git 工作树中隔离变化；Cloud 在配置好的远程环境中运行。OpenAI Developers：Codex environments

环境	更适合	当前案例
Local	需要现有本地依赖、固定浏览器或立即人工观察	第 24 章浏览器验证
Worktree	需要与本地主线隔离、并行推进较长修改	第 23 章任务编辑候选实现
Cloud	远程仓库已同步，希望任务在隔离云环境中运行	第 28 章纯文档任务

选择环境时先问三个问题：

177. 任务是否必须使用本机已有服务或浏览器数据？
178. 是否需要和当前本地工作并行、避免互相干扰？
179. 远程仓库是否已经包含任务所需的全部基线？

若任务依赖尚未推送的本地文件，Cloud 看不到这些变化；若任务需要固定本地浏览器数据，Worktree 或 Cloud 也不会自动继承；若只是只读整理项目状态，通常不需要建立 Worktree。

25.6 不把环境选择当作权限授权

Local、Worktree 和 Cloud 说明任务在哪里运行，不等于 Codex 可以任意操作。

仍要分别确认：

- 可以读取哪些目录；
- 是否允许修改文件；
- 哪些命令需要批准；
- 是否允许访问网络；
- 是否会写入 GitHub 或其他外部系统。

例如，Local 意味着直接面对当前项目目录，因此更需要在任务开始前确认工作区是否干净；Cloud 虽然隔离于本机，却可能连接 GitHub 并创建分支或 PR，外部写入仍需要清楚授权。

安全规则将在第 36 章集中收束，本章只保留最低认识：

> **运行位置、文件范围和外部写入是三个不同决定。**

25.7 整理历史聊天而不丢失事实

已完成的聊天可以归档，但归档前先确认成果已经进入仓库：

- 代码和文档已提交；
- PR 已按实际流程合并；
- `docs/current-state.md` 已更新；
- 验证报告已经保存；
- 未完成事项已有明确去向。

聊天不是最终交付物。若一个重要结论只存在聊天里，归档后仍可能难以复用。

对于第 23—24 章的任务编辑聊天，归档前应能在仓库中找到：

```
docs/plans/task-editing.md
docs/task-editing-validation.md
docs/current-state.md
```

这三份文件分别保存计划过程、验证证据和最终项目状态。

25.8 为下一项维护任务建立正确入口

第 26—27 章将处理一项小型文档一致性维护任务。

现在只做准备：

180. 确认 `main` 已同步且工作区干净；
181. 新建聊天 `03｜文档口径一致性维护`；
182. 选择 Local，因为任务需要在同一本地仓库中跨 CLI 和 IDE 继续；
183. 暂不创建 Worktree；
184. 暂不修改文件。

让新聊天先只读确认：

```
请只读检查 README.md、docs/manual-test-checklist.md 和 docs/current-state.md。

说明这些文档分别承担什么职责，是否存在“截止日期”相关术语不一致，以及任务编辑验证报告是否有可追踪入口。
不要修改文件。
```

本章结束时，桌面端已经成为清楚的任务导航层，但项目文件没有发生变化。

25.9 发现打开了错误副本时怎样恢复

桌面端最常见的项目管理错误，不是聊天名称不好，而是聊天连接到了错误目录。

例如：

- 项目根目录和下载副本同名；
- Local 聊天连接到旧版本；
- Worktree 聊天已经完成，但人又在主目录中重复修改；
- 归档项目重新打开后，当前分支与聊天标题不一致。

出现以下信号时应立即停止：

- Codex 报告的文件结构与 `docs/current-state.md` 不一致；
- 任务编辑功能在页面中消失；
- 最近提交不是第 24 章的合并结果；
- `AGENTS.md` 或验证报告不存在；
- 聊天说在功能分支，Git 却显示 `main`；
- 同名文件的内容明显较旧。

恢复步骤：

185. 不允许继续修改；
186. 记录当前绝对路径、分支和状态；
187. 在文件资源管理器或 Finder 中找到可信项目根目录；
188. 对照最近提交和 `docs/current-state.md`；
189. 重新打开正确项目；
190. 若错误目录已有变化，先单独保存差异，不要直接复制覆盖。

可以提交：

```
当前项目可能不是可信主目录。请不要修改任何文件。
请输出绝对路径、Git 根目录、当前分支、最近提交、AGENTS.md 位置和 docs/current-state.md 摘要，并指出与任务聊天标题之间是否存在冲突。
```

聊天上下文不能证明目录正确。恢复判断必须来自文件系统和 Git 证据。

25.10 建立一个最小的日常进入流程

后续每次回到团队任务看板，可以固定做四件事：

```
打开可信主项目
→ 查看状态聊天或 current-state.md
→ 选择一条目标明确的新聊天
→ 确认 Local、Worktree 或 Cloud
```

开始任务前再核对：

检查	需要得到的答案
项目	是否为可信根目录
聊天	是否只承担一个成果
环境	是否满足本地依赖和隔离要求
Git	当前分支和工作区是否符合起始条件
权限	本次允许读取、修改和外部写入什么
验收	结束时要留下什么文件或证据

这套流程不需要额外建立管理文档。它的作用是让每条任务聊天从真实项目状态开始，而不是从上一条聊天的模糊印象开始。

三、总结

本章完成的不是界面整理，而是工作对象分层。

你已经理解：

- 项目对应真实项目根目录；
- 聊天对应一项独立成果或问题；
- Git 分支对应一组文件变化；
- 状态聊天只负责重新进入项目；
- Local、Worktree 和 Cloud 根据运行条件选择；
- 环境选择不等于权限授权；
- 重要事实必须落到项目文件和 Git 历史中。

本章掌握检查

- [ ] 团队任务看板只有一个可信主项目入口；
- [ ] 历史成果与新任务没有混在同一聊天；
- [ ] 保留了只读项目状态聊天；
- [ ] 能区分项目、聊天和 Git 分支；
- [ ] 能根据本地依赖、并行要求和远程基线选择环境；
- [ ] 没有把聊天归档当成成果交付；
- [ ] 第 26 章的新维护聊天已经建立但尚未修改文件。

四、结束语与引导

桌面端适合观察多条任务和长期上下文，但有些维护工作更适合留在终端中完成。

下一章进入同一个团队任务看板，在正确项目目录中启动 Codex CLI，完成一次范围很小的文档口径维护：

> **统一“截止日期”术语，并用终端状态、会话和差异形成清楚的本地工作闭环。**

第 26 章：在命令行中完成小型本地维护任务

一、起始语

第 25 章已经为团队任务看板整理了项目和聊天。

当前状态是：

- 项目根目录为 `team-task-board`；
- `main` 已同步；
- 工作区干净；
- 新聊天为 `03｜文档口径一致性维护`；
- 本次不修改业务代码；
- 任务将在命令行开始，并在第 27 章转到编辑器完成审查。

本书持续使用的 `team-task-board` 基础练习项目中，预留了两处不影响功能的轻微文档问题，留到本章统一处理：

- `README.md` 一处使用“到期日期”；
- `docs/manual-test-checklist.md` 一处使用“截止时间”。

项目在代码、界面和前文中统一采用：

> **截止日期**

本章唯一任务是：

> **在正确项目目录中启动 Codex CLI，只修改两份文档，形成可供 IDE 继续审查的未提交候选差异。**

Codex CLI 直接在本地仓库中读取文件、修改内容并运行已经安装的工具，适合把文件状态、命令输出和差异保留在同一终端会话中。OpenAI Developers：Codex CLI

二、主体

26.1 什么时候优先使用 CLI

CLI 适合：

- 已经知道项目目录；
- 任务范围较小；
- 需要频繁查看 Git 和文件状态；
- 主要证据来自命令输出；

- 不依赖复杂视觉观察；
- 希望继续使用同一个终端会话。

本次只是统一 Markdown 文档中的术语，不需要浏览器，也不需要创建 Worktree。

桌面端更适合管理多条聊天和环境；CLI 更适合围绕一个当前目录形成紧凑执行循环。两者可以使用同一项目规则，但不能假设它们正在同一目录或同一分支。

26.2 从正确目录启动

先在普通终端进入项目根目录。

Windows 示例：

```
cd "$HOME\Documents\Codex 学习\projects\team-task-board"
```

macOS 示例：

```
cd "$HOME/Documents/Codex 学习/projects/team-task-board"
```

核对：

```
Get-Location
git branch --show-current
git status --short
```

macOS 可以使用：

```
pwd
git branch --show-current
git status --short
```

预期：

- 当前目录是 `team-task-board`；
- 当前分支为 `main`；
- `git status --short` 没有输出。

若工作区不是干净状态，应先识别变化来源，不要把旧改动带进新的维护任务。

26.3 检查 CLI 并启动会话

本书前文已经完成安装，本章只做存在性检查：

```
codex --version
```

然后在项目根目录启动：

```
codex
```

若 CLI 暂时无法安装或登录，可以改用桌面端完成同一项文档维护任务，不让环境问题阻塞本章主线。

进入会话后使用`/status`查看当前聊天标识、上下文和当前会话状态。官方命令会随版本和访问权限变化，输入`/`可以查看当前环境实际提供的命令。OpenAI Developers：Developer commands

先提交只读任务：

```
请不要修改文件。

确认：
1. 当前项目根目录；
2. 当前 Git 分支和工作区状态；
3. AGENTS.md 中与文档修改有关的规则；
4. README.md 和 docs/manual-test-checklist.md 中所有“截止日期”“到期日期”“截止时间”出现位置；
5. 哪个词是当前项目的统一术语。

根据实际文件回答，不要创建分支或提交。
```

这一轮的作用是验证任务前提，而不是让 Codex 根据提示直接假设问题存在。

26.4 建立小型维护分支

确认受控问题真实存在后，建立分支：

```
git switch -c docs/terminology-alignment
```

再次查看：

```
git branch --show-current
git status --short
```

本章不重新解释分支原理。这里只保留结果要求：

- 当前分支不是`main`；
- 分支只承担这一次文档维护；
- 开始修改前工作区仍然干净。

若团队规则不允许斜杠分支名，应按`AGENTS.md`中的真实约定调整。

26.5 把维护任务写成窄范围指令

提交：

```
请在当前 docs/terminology-alignment 分支完成文档术语维护。

只允许修改：
- README.md
- docs/manual-test-checklist.md

要求：
1. 将当前项目文档中的“到期日期”和“截止时间”统一为“截止日期”；
2. 不改写句子中与术语无关的内容；
3. 不调整标题层级、表格结构和列表顺序；
4. 不修改代码、AGENTS.md、项目状态或验证报告；
5. 修改后搜索三个术语，报告剩余位置及其是否合理；
6. 运行 git diff --check 并展示差异摘要；
7. 不创建提交。
```

这里不需要重复第 9 章的六要素模板，因为目标、范围、非目标和验收已经足够清楚。

完成后，先让 Codex 汇报：

- 实际修改文件；
- 每处修改的原词和新词；
- 是否仍存在三个术语；
- 未修改内容；
- 检查结果。

26.6 用命令验证真实差异

在终端中自行检查：

```
git status --short
git diff --check
git diff --stat
git diff -- README.md docs/manual-test-checklist.md
```

预期范围只有两份文档。

还可以搜索：

Windows PowerShell：

```
Select-String -Path README.md,docs\manual-test-checklist.md -Pattern "到期日期|截止时间|截止日期"
```

macOS：

```
grep -nE "到期日期|截止时间|截止日期" README.md docs/manual-test-checklist.md
```

搜索结果需要结合语义判断。例如，文档引用一条历史决策原文时，旧词可能需要保留并注明背景；本次练习设定中不存在这一例外。

不能只凭“替换命令执行成功”宣布维护完成。还要确认：

- 没有替换代码变量名；
- 没有改变功能含义；
- 没有改动其他段落；
- Markdown 结构没有破坏。

26.7 使用 CLI 进行候选差异自查

提交只读审查：

```
请只读审查当前未提交差异。

检查：
1. 是否只修改 README.md 和 docs/manual-test-checklist.md；
2. 是否只统一为“截止日期”；
3. 是否存在与任务无关的改写；
4. 是否遗漏同一语境中的旧术语；
5. Markdown 结构是否保持；
6. 是否适合交给 IDE 进行下一轮人工审查。

不要修改文件，不要提交。
```

Codex CLI 也可以启动专门的差异审查；本章不固定某个易变界面，输入`/`查看当前版本是否提供`/review`，或直接使用上面的只读审查任务。

本轮即使没有发现问题，也只能得出：

> 两份文档的候选差异满足 CLI 层面的范围和文本检查。

第 27 章还要在源文件旁检查上下文和关联文档。

26.8 保留会话和未提交状态

本章不提交，因为下一章要在 IDE 中继续同一分支。

结束前记录：

```
git branch --show-current
git status --short
git diff --stat
```

标准结束状态：

- 当前分支为 `docs/terminology-alignment`；
- `README.md` 和 `docs/manual-test-checklist.md` 有未提交修改；

- `git diff --check` 通过；
- 没有业务代码变化；
- CLI 聊天保留了探索、修改和差异证据；
- 尚未提交、推送或创建 PR。

如果需要稍后回到终端会话，可以使用当前 CLI 提供的保存或恢复能力，例如 `codex resume` 或 `/resume`。恢复后仍应重新确认目录、分支和工作区，不能只依赖聊天历史。OpenAI Developers：Developer commands

26.9 为命令执行设定停止条件

CLI 能够连续运行本地命令，因此小任务也需要明确停止条件。

本次出现以下情况时应暂停：

- 当前目录不是项目根目录；
- `main` 开始时已有未知变化；
- 搜索发现旧术语存在于代码变量或历史原文中；
- Codex 准备修改两份文档之外的文件；
- 替换导致整段重新排版；
- `git diff --check` 失败；
- 终端命令要求访问项目外目录或网络，而任务并不需要。

暂停后先让 Codex 解释：

> 停止继续修改。请说明当前阻断点、已经发生的文件变化、最后一个成功检查和恢复到任务范围所需的最小动作。不要自动撤销或扩大权限。

如果只是误改了一个任务外文件，可以在人工确认差异后只恢复该文件；不要用会清空全部工作区的命令处理一个局部问题。

CLI 的高效率来自连续性，也意味着错误命令可能快速扩大影响。对零基础读者来说，最重要的不是记住更多参数，而是知道什么时候停下来。

26.10 将终端会话交给下一种入口

第 27 章会继续使用同一个本地检出，不需要把 CLI 聊天内容全文复制到 IDE。

真正需要交接的是：

- 项目根目录；
- 当前分支；

- 实际修改文件；
- 已执行检查；
- 尚未提交的原因；
- 下一步要在 IDE 观察什么。

可以让 CLI 生成一段简短交接：

```
请为下一步 IDE 审查生成交接摘要，只包含：
- 当前目录和分支；
- 两份修改文件；
- 术语替换结果；
- 已执行的搜索和 diff 检查；
- 尚未检查的 Markdown 预览、相对链接和关联文档。
不要修改文件。
```

交接摘要可以保留在聊天中，不必为一次小任务新建项目文档。第 22 章建立的长期知识文件只保存会持续影响项目的事实，不保存每一次临时操作记录。

三、总结

本章用 CLI 完成了一项小型、真实且边界清楚的维护任务。

你已经完成：

- 从正确项目目录启动 Codex；
- 使用`/status`和 Git 状态确认当前对象；
- 读取`AGENTS.md`和真实文件验证问题；
- 建立聚焦文档分支；
- 只修改两份指定文档；
- 搜索旧词和统一术语；
- 检查差异范围与 Markdown 结构；
- 保留未提交候选变化供 IDE 继续审查。

本章掌握检查

- [ ] CLI 从项目根目录启动；
- [ ] 开始前`main`工作区干净；
- [ ] 当前分支为`docs/terminology-alignment`；
- [ ] 只修改两份指定文档；
- [ ] 项目统一术语为“截止日期”；
- [ ]`git diff --check`通过；
- [ ] 没有创建提交或 PR；
- [ ] 第 27 章可以直接打开同一分支继续。

四、结束语与引导

终端擅长确认目录、状态、搜索结果和完整差异，但长 Markdown 文件中的局部上下文，在编辑器里通常更容易观察。

下一章不重新开始任务，而是打开同一个 `docs/terminology-alignment` 分支：

> **在源文件旁审查 CLI 修改，补齐一处关联文档问题，并完成这条维护分支的交付。**

第 27 章：在编辑器中继续审查和完成同一维护分支

一、起始语

第 26 章结束时，团队任务看板处于明确的中间状态：

- 当前分支为 `docs/terminology-alignment`；
- `README.md` 和 `docs/manual-test-checklist.md` 存在未提交候选修改；
- 两份文档中的目标术语已统一为“截止日期”；
- CLI 范围检查和 `git diff --check` 已经通过；
- 尚未提交、推送或创建 PR。

本章不建立新分支，也不重复同一修改。

唯一任务是：

> **在 IDE 中利用打开文件、选区、Markdown 预览和源代码旁差异继续审查，补齐 `docs/current-state.md` 中的验证报告入口，再完成维护分支交付。**

Codex IDE 扩展可以直接使用当前打开文件、选中的代码或文字以及近期聊天作为上下文，并在源文件旁展示摘要和差异，适合局部审查和人工取舍。OpenAI Developers：Codex IDE extension

二、主体

27.1 打开同一个项目和分支

在 VS Code 或兼容编辑器中打开项目根目录：

```
team-task-board/
```

不要只打开 `docs/` 子目录，也不要打开第 23 章留下的旧 Worktree 路径。

在集成终端确认：

```
git branch --show-current
git status --short
```

预期仍为：

```
docs/terminology-alignment
 M README.md
 M docs/manual-test-checklist.md
```

如果编辑器打开后自动切换了工作区或分支，应先纠正，再启动 Codex 侧栏。

VS Code 及兼容编辑器使用 Codex 扩展；具体按钮位置可能变化，可以从命令面板执行“Codex: Open Codex Sidebar”。OpenAI Developers：Codex IDE extension

27.2 用打开文件和选区减少重复说明

依次打开：

- `README.md`；
- `docs/manual-test-checklist.md`；
- `docs/current-state.md`；
- `docs/task-editing-validation.md`。

在 README 和人工测试清单中选中刚刚修改的段落，然后提交：

```
请基于当前打开文件和选区，只读审查第 26 章的文档术语修改。

检查：
1. “截止日期”是否与代码、界面和其他项目文档一致；
2. 修改是否改变了原句其他含义；
3. Markdown 列表、表格和标题是否仍完整；
4. 当前状态文档是否能追踪到任务编辑验证报告；
5. 是否存在必须处理的相邻文档问题。

先报告发现，不要修改文件。
```

IDE 上下文减少了“请打开第几行”的描述成本，但人仍要检查 Codex 实际引用的是哪个文件和选区。

27.3 区分差异视图、问题面板和内容正确性

编辑器提供多种证据，但它们回答的问题不同。

证据	能说明什么	不能单独说明什么
源代码管理差异	哪些行发生变化	术语是否符合业务口径
Markdown 预览	标题、表格、链接是否正常显示	所有链接目标一定存在
问题面板	编辑器或扩展发现的语法、格式问题	文档内容完整且没有遗漏
Codex 审查	结合上下文发现可能问题	结果已经被人工确认

本次两处术语替换通常不会触发问题面板。问题面板为空，只能说明没有被当前工具识别的错误。

因此，需要同时观察：

- 修改行上下文；
- Markdown 预览；
- 搜索结果；
- 关联文档入口。

27.4 发现关联文档缺口

当前团队任务看板项目中，`docs/current-state.md` 已经说明任务编辑功能完成，但没有提供验证报告的相对链接。

这不是第 26 章的术语替换范围，所以 CLI 没有顺手修改。

在 IDE 中同时打开 `current-state.md` 和 `task-editing-validation.md` 后，关联缺口更容易被看见。

确认真实文件存在：

```
docs/task-editing-validation.md
```

然后选中 `current-state.md` 中“最近验证成果”段落，提交：

```
请只修改 docs/current-state.md 当前选中的“最近验证成果”段落。

要求：
1. 保留现有任务编辑完成状态；
2. 增加指向同目录 task-editing-validation.md 的相对 Markdown 链接；
3. 链接文字使用“任务编辑验证报告”；
4. 不改写其他状态、不新增未验证结论；
5. 不修改其他文件。
```

推荐形式：

```
- 任务编辑功能已完成浏览器验证，详见[任务编辑验证报告](./task-editing-validation.md)。
```

必须根据当前文件所在目录确认相对路径。若 `current-state.md` 位于 `docs/` 中，使用 `./task-editing-validation.md`；不要写成重复的 `docs/docs/` 路径。

27.5 在源文件旁审查最终差异

现在预期修改三份文档：

- `README.md`；

- `docs/manual-test-checklist.md`；
- `docs/current-state.md`。

在 IDE 差异视图中逐文件检查：

README

- 只替换受控术语；
- 功能说明没有扩大；
- 没有重新排版整段。

人工测试清单

- 用词与界面一致；
- 测试步骤含义不变；
- 表格或复选项没有损坏。

当前状态

- 只增加一条真实存在的验证报告链接；
- 没有把“已完成”扩大成“所有异常路径均已执行”；
- 链接路径正确。

可以提交：

```
请只读审查当前三份文档的最终差异。

按“阻断问题、非阻断建议、无需处理”分类。
重点检查任务范围、相对链接、术语一致性和是否出现未经证实的新结论。
不要修改文件。
```

非阻断的文风建议不应扩大本次维护范围。

27.6 使用预览和搜索完成文档验收

打开 Markdown 预览，确认：

- README 相关段落正常；
- 人工测试清单结构正常；
- `current-state.md` 链接文字正常；
- 点击链接可以打开 `task-editing-validation.md`。

再在编辑器全局搜索：

```
到期日期
截止时间
截止日期
```

需要区分：

- 本次三份文档中的目标旧词已经消失；
- 代码、历史决策或引用中是否存在合理旧词；
- 搜索范围是否误包含`.git`、归档或旧导出文件。

最后运行：

```
git status --short
git diff --check
git diff --stat
```

文档任务不要求重新进行完整浏览器功能回归，因为没有业务代码变化。验收重点是差异、预览、链接和搜索。

27.7 按已学流程完成维护分支

当前维护已经达到提交条件：

- 三份文档变化范围明确；
- 术语统一；
- 验证链接可打开；
- Markdown 结构正常；
- 没有业务代码变化；
- 没有阻断问题。

按第 20 章已经掌握的流程完成：

191. 最终查看差异；
192. 创建聚焦提交；
193. 推送并发起审查，或按照个人练习流程在本地完成审查；
194. 合并到`main`；
195. 同步本地主分支；
196. 按团队规则清理维护分支。

建议提交信息：

```
统一截止日期文档口径
```

本章不重复完整 Git 和 PR 命令。

合并后确认：

```
git switch main
git pull --ff-only
git status --short
```

标准结束状态：

- `main` 包含三份文档修改；
- 本地主分支与远程一致；
- 工作区干净；
- `docs/terminology-alignment` 已按规则清理；
- 第 28 章可以从干净远程基线开始云端任务。

27.8 CLI 与 IDE 怎样配合

本次任务没有把两种入口写成重复流程。

阶段	更合适的入口	原因
确认目录和分支	CLI	状态直接、命令证据集中
搜索多个文件术语	CLI	文本搜索和差异快速
查看长文档局部上下文	IDE	源文件、选区和预览并列
发现关联链接缺口	IDE	多个相关文档同时可见
最终提交与同步	任一入口	使用已经学过的 Git 流程

工具选择的依据是任务证据，不是“哪一个更高级”。

27.9 只保留经过人工确认的修改

IDE 差异视图允许逐文件、逐片段观察变化。若 Codex 在补链接时顺便调整了其他文字，不必接受整个文件。

处理顺序：

197. 展开文件差异；
198. 识别任务内片段和任务外片段；
199. 对任务外改写要求 Codex 恢复，或人工丢弃对应片段；
200. 再次查看文件完整差异；
201. 确认工作区只剩预期变化。

不要因为某个建议“读起来更顺”就在维护分支中顺手接受。文风优化、信息重组和术语修正是不同任务。

同样，不应仅凭 Codex 摘要判断变化。摘要可能说“补充验证链接”，真实差异却可能同时改写状态结论。最终审查对象始终是文件差异。

27.10 编辑器共享上下文也要控制范围

IDE 能够自动提供打开文件、选区和附近代码，这会减少重复描述，也可能带入无关内容。

本次只需要四份 Markdown 文件。不要同时打开：

- 含真实凭据的环境文件；
- 与任务无关的个人笔记；
- 大量历史导出文件；
- 其他客户或项目目录；
- 不需要审查的工作区窗口。

向 Codex 提交任务前，观察当前附加的文件和选区是否正确。若扩展支持共享 IDE 上下文开关，应按任务需要启用或关闭，而不是默认把整个工作空间都视为必要上下文。

局部任务的基本原则是：

> **给足以判断问题的上下文，不把可访问的全部内容都交给当前任务。**

27.11 不适合停留在 IDE 中的情况

IDE 适合局部阅读和修改，但以下任务不应为了“都在编辑器里完成”而勉强停留：

- 需要长时间运行、可以远程并行的任务；
- 需要固定桌面浏览器或本机应用操作的验证；
- 需要大规模命令输出和脚本化处理；
- 需要在隔离 Worktree 中避免干扰当前窗口；
- 需要基于 GitHub 远程基线创建云端 PR。

第 28 章的贡献者文档任务不依赖本地窗口和未提交文件，因此会转到 Cloud。入口切换的依据仍然是任务条件，而不是保持某一种工具的使用连续性。

三、总结

本章在 IDE 中完成了第 26 章候选差异的继续审查和交付。

你已经完成：

- 在编辑器中打开同一项目和同一分支；
- 使用打开文件和选区提供精确上下文；
- 区分差异视图、Markdown 预览和问题面板；
- 发现并补齐当前状态中的验证报告链接；
- 人工审查三份文档的最终差异；
- 验证相对链接和术语搜索结果；
- 按已学流程提交、审查、合并并同步 `main`。

本章掌握检查

- [ ] IDE 打开的是项目根目录而不是子目录或旧 Worktree；
- [ ] 当前分支继承自第 26 章；
- [ ] 修改范围只有三份文档；
- [ ] “截止日期”口径已经统一；
- [ ] 验证报告链接使用正确相对路径；
- [ ] 问题面板为空没有被误写成完整验收；
- [ ] 文档预览和链接已经人工检查；
- [ ] 维护分支已合并，`main` 工作区干净。

四、结束语与引导

到这里，同一个本地维护任务已经跨过 CLI 和 IDE：终端负责目录、搜索和状态，编辑器负责上下文、预览和局部审查。

下一章把工作对象从本地检出移到远程 GitHub 仓库：

> **在 Codex 云端的隔离环境中完成一次纯文档任务，审查差异、创建 PR，再把合并结果同步回本地。**

第 28 章：使用 Codex 云端完成一次 GitHub 文档任务

一、起始语

第 27 章结束时：

- 文档口径维护已经合并到 `main`；
- 本地与远程主分支一致；
- 工作区干净；
- 团队任务看板的项目规则和关键文档入口清楚；
- 下一项任务不依赖本地浏览器数据，也不需要修改业务代码。

本章使用 Codex 云端完成一个范围明确的远程任务：

> **新建贡献者快速开始文档，并把它加入 README 的文档索引。**

文档只包含：

- 项目目的；
- 本地启动方式；
- 关键项目文档入口；
- 修改后的最低检查。

本次不安装依赖，不使用 Secrets，不改代码，不展开 SSH 或远程主机 Handoff。

Codex cloud 在隔离的云环境中基于已连接的 GitHub 仓库运行任务，可以配置依赖和工具，审查摘要与差异，并在结果准备好后创建 PR。OpenAI Developers：Codex cloud

二、主体

28.1 先判断任务是否适合放到云端

云端更适合：

- 远程仓库已经包含完整基线；
- 任务不依赖未提交的本地文件；
- 可以在隔离环境中独立完成；
- 希望本机关闭或切换到其他工作时任务继续；
- 最终结果可以通过 Git 差异和 PR 审查。

当前任务只读取仓库文档并新建一份说明文件，适合云端。

若本地 `main` 尚未推送，或第 27 章仍有未提交变化，应先停止。Cloud 从选择的远程仓库和分支开始，不会自动读取本机工作区。

28.2 锁定远程基线

本地先确认：

```
git switch main
git pull --ff-only
git status --short
git log --oneline -1
```

记录：

- 仓库名称；
- 远程默认分支；
- 最近提交哈希；
- 本地工作区是否干净。

本章云端任务必须基于这一提交或更新的远程 `main`。

如果 GitHub 仓库尚未连接 Codex，需要在 Codex 云端按提示连接 GitHub，并只授权当前练习仓库。账号注册、组织审批和企业仓库策略不在正文展开。

28.3 建立最小云环境

Codex cloud 要求为仓库选择或创建环境。环境可以配置依赖、工具、变量、Secrets 和初始化步骤。

OpenAI Developers：Codex cloud

当前项目是静态 HTML、CSS 和 JavaScript 练习项目，本次又只改 Markdown，因此最小环境应保持克制：

配置	本次选择
仓库	`team-task-board` 对应 GitHub 仓库
基线分支	`main`
依赖安装	不需要
构建命令	不需要
环境变量	不需要

配置	本次选择
Secrets	不需要
网络访问	只保留平台完成仓库任务所需能力
验证	文件存在、Markdown 结构、相对链接和 Git 差异

不要因为环境支持 Secrets 就创建无意义密钥。任何真实凭据都不应写入提示词、仓库文件或普通日志。

28.4 先让云端理解仓库，不立即修改

选择环境和 `main` 后，提交只读探索：

```
请先只读检查当前 GitHub 仓库，不要修改文件。

依据 AGENTS.md、README.md、docs/project-context.md 和 docs/current-state.md，说明：
1. 项目用途和技术结构；
2. 当前本地启动方式；
3. 新贡献者最需要阅读的关键文档；
4. 修改文档后能够在云端执行的最低检查；
5. 当前是否已经存在贡献者快速开始文档。

事实不确定时不要补写。
```

检查云端输出是否与第 27 章合并后的远程仓库一致。

若它仍看到旧术语或找不到最新链接，说明云端基线可能不是最新 `main`，应停止并重新选择分支或同步远程。

28.5 提交纯文档任务

确认基线后，提交：

```
请完成一次纯文档任务。

允许修改：
- 新建 docs/contributor-quickstart.md
- README.md 中的文档索引段落

contributor-quickstart.md 只包含：
1. 项目目的和适用范围；
2. 从仓库根目录启动本地 HTTP 服务的方法；
3. AGENTS.md、docs/project-context.md、docs/current-state.md、docs/manual-test-checklist.md 的用途和相对链接；
4. 修改后至少执行 git diff --check、查看差异，并按变更类型进行人工检查；
5. 明确真实业务数据、密钥和环境文件不应提交。

要求：
- 根据仓库实际内容写，不虚构依赖、测试或 CI；
```

```
- Windows 与 macOS 命令分开；
- 不修改 HTML、CSS、JavaScript 和现有验证结论；
- 不安装依赖，不使用 Secrets；
- 完成后运行可用的文档与 Git 检查；
- 输出实际修改文件、检查结果和未验证事项。
```

这项任务有足够业务价值，又不会为了展示云端而临时增加功能。

28.6 审查云端执行记录和差异

云端完成后先看摘要，再看真实差异。

重点检查：

新文档内容

- 项目目的来自真实仓库；
- 启动命令与 README 一致；
- Windows 与 macOS 没有混在同一命令块；
- 关键文档链接使用正确相对路径；
- 没有虚构自动化测试；
- 安全提醒没有扩大成完整企业治理章节。

README 索引

- 只增加一个真实存在的文档入口；
- 没有重排整个 README；
- 链接从 README 所在根目录正确指向 `docs/contributor-quickstart.md`。

文件范围

预期只有：

```
README.md
docs/contributor-quickstart.md
```

若云端修改了代码或其他状态文档，应要求撤回任务外变化。

28.7 对不准确结果提出聚焦追问

云端任务并非第一次输出就一定可以合并。

例如发现：

- 把 `npm install` 写成项目必需步骤；
- 声称存在自动化测试；
- 链接写成 `docs/docs/current-state.md`；
- 将本地服务命令只写成一个操作系统；
- 在 README 中重写了无关段落。

可以在同一云端聊天中要求：

```
请只修正以下阻断问题：
1. 删除仓库中不存在的依赖安装和自动化测试表述；
2. 修正相对链接；
3. 恢复 README 中与文档索引无关的变化。

保持文件范围不变，完成后重新报告差异和检查结果。
```

追问仍然使用同一任务，不新建另一个云端聊天。

28.8 在创建 PR 前完成最终检查

让 Codex 只读审查最终结果：

```
请只读审查当前云端候选差异。

检查：
1. 是否只有 README.md 和 docs/contributor-quickstart.md 变化；
2. 是否完全依据仓库事实；
3. 相对链接是否正确；
4. 是否虚构依赖、测试、CI 或权限；
5. 是否泄露或要求提交密钥；
6. 是否适合创建文档 PR。

不要继续修改，按阻断问题和非阻断建议输出。
```

只有阻断问题清零，才进入 PR。

PR 说明至少包含：

- 为什么需要贡献者快速开始文档；
- 修改了哪两份文件；
- 执行了哪些检查；
- 哪些验证不适用，例如没有业务代码和浏览器功能变化。

不要把“未运行浏览器测试”写成缺陷；对纯文档任务，应说明它不属于本次必要验证。

28.9 创建 PR、审查和合并

Codex cloud 可以在结果准备好后打开 PR。创建后仍需要人检查：

- 基线分支是否正确；
- 文件差异是否与云端候选一致；
- 仓库检查是否通过；
- 审查意见是否需要处理；
- 是否符合合并权限和团队规则。

本章不重新教学 GitHub 账号、分支保护和完整 PR 操作。

若 PR 存在阻断意见，在云端聊天中继续做聚焦修正，再更新 PR。

通过后按团队规则合并。

云端“任务完成”只表示候选结果准备好；“PR 已合并”才表示远程主分支已经接收成果。

28.10 将远程结果带回本地

PR 合并后，回到本地项目：

```
git switch main
git pull --ff-only
git status --short
git log --oneline -3
```

检查文件：

```
docs/contributor-quickstart.md
```

人工打开 README 中的链接，确认能够进入新文档。

还要核对：

- 新文档内容与 PR 一致；
- 本地没有未提交变化；
- 最近提交包含云端文档任务；
- 第 29 章读取的是更新后的 `main`。

如果本地存在自己的未提交变化，不应强行拉取覆盖；应先识别并处理本地状态。

28.11 云端与本地任务的边界

完成本章后，可以形成一张简化判断表。

情况	更适合本地	更适合云端
依赖固定本地浏览器数据	是	否
需要使用本机尚未推送文件	是	否
远程仓库基线完整	可以	是
希望任务与本机并行	Worktree 可用	是
结果主要通过 PR 审查	可以	是
需要真实本地应用人工观察	是	通常先回本地

云端不是本地入口的升级替代。它提供的是隔离执行、远程并行和 GitHub 交付路径。

28.12 云端任务失败时保留什么证据

云端任务可能因为以下原因停止：

- 仓库授权不足；
- 环境初始化失败；
- 选择了错误分支；
- 相对链接检查失败；
- 任务超出允许时间；
- Codex 修改了任务外文件；
- PR 权限或仓库规则阻止创建、更新或合并。

失败时不要反复重开新任务掩盖原因。至少记录：

- 环境名称和基线分支；
- 基线提交；
- 失败阶段；
- 最后一条有效日志；
- 是否已经产生分支或候选差异；
- 是否发生任何外部写入；
- 下一次重试需要改变什么。

若候选差异已经存在，可以先审查并决定继续当前聊天、放弃候选，或在修正环境后重新运行。不要让多个近似云端分支同时存在而没有清楚状态。

28.13 环境变量和 Secrets 只在任务需要时配置

云环境支持环境变量和 Secrets，并不表示每个任务都需要它们。

本章纯文档任务没有：

- 外部 API；
- 私有包下载；
- 数据库；
- 部署平台；
- 真实业务服务。

因此，正确配置是“不提供 Secrets”。

以后任务确实需要凭据时，也应遵守：

- 只提供任务必需的最小凭据；
- 不把值写进提示词和仓库；
- 区分只读和写入权限；
- 任务结束后按团队策略轮换或撤销；
- 日志和差异中不得出现密钥内容。

完整密钥与发布治理放在第 36 章，本章只建立“没有需要，就不配置”的习惯。

28.14 云端交付需要三层状态一致

远程任务结束后要区分三层状态：

层级	可能状态	本章通过条件
云端聊天	候选完成、失败或等待追问	候选差异无阻断问题
GitHub	分支、PR、审查和合并	PR 已按规则合并到 `main`
本地仓库	旧 `main` 或已同步 `main`	`git pull --ff-only` 后工作区干净

云端聊天显示“完成”，不等于 PR 已合并；PR 已合并，也不等于本地已经取得变化。

只有三层状态都确认后，下一章才能把新文档视为项目正式资产。

28.15 不用 Cloud 完成所有远程工作

云端适合隔离和并行，但仍有边界：

- 结果依赖真实本地设备时，需要回到本地；
- 任务需要人工设计判断时，应缩小目标并加强审查；
- 仓库没有可复现环境时，云端结果可能无法验证；
- 高风险外部写入不能仅凭任务自动完成；
- 未同步的本地状态必须先处理。

因此，使用 Cloud 的正确目标不是减少所有人工参与，而是把适合远程执行的部分交给隔离环境，并让结果通过 GitHub 差异回到可审查流程。

三、总结

本章完成了第一条完整云端协作路径：

- 判断任务适合云端；
- 确认本地与远程 `main` 一致；
- 连接受控 GitHub 仓库；
- 建立不含依赖和 Secrets 的最小环境；
- 先只读理解仓库；
- 新建贡献者快速开始文档并更新 README 索引；
- 审查日志、摘要和真实差异；
- 对阻断问题提出聚焦修正；
- 创建、审查并合并文档 PR；
- 将远程成果同步回本地干净主分支。

本章掌握检查

- [ ] 云端任务基于最新远程 `main`；
- [ ] 本次任务不依赖未推送本地文件；
- [ ] 环境没有配置无意义依赖或 Secrets；
- [ ] 只修改 README 和贡献者快速开始文档；
- [ ] 文档内容来自真实仓库；
- [ ] 相对链接和操作系统命令已经检查；
- [ ] PR 创建前没有阻断问题；
- [ ] PR 合并后本地 `main` 已同步；
- [ ] 工作区干净。

四、结束语与引导

第 25—28 章已经把同一个团队任务看板带过四种入口：

```
桌面端：组织项目、聊天和环境
→ CLI：完成目录、搜索和差异驱动的小型维护
→ IDE：结合选区、预览和源文件旁差异完成局部审查
→ Cloud：在远程隔离环境中完成 GitHub 任务并通过 PR 带回本地
```

下一篇不再增加新的使用入口，而是开始沉淀可复用能力。

第 29 章将从团队任务看板已有的审查方法中提取一项稳定流程：

> **把“项目变更审查”写成一个最小 Skill，让 Codex 在桌面端、CLI 和 IDE 中复用同一套检查要求。**

第 29 章：用 Skill 沉淀项目变更审查方法

一、起始语

第 20 章以后，你已经多次要求 Codex 检查项目变化：

- 修改是否超出任务范围；
- 数据结构和已有功能是否被破坏；
- 文档是否与真实代码一致；
- 验证结果是否有足够证据；
- Git 差异中是否混入无关文件。

这些要求有稳定价值，却经常以不同形式重新写进提示词。

当一项工作已经满足下面三个条件时，就应该考虑沉淀，而不是继续依靠复制粘贴：

202. 任务会重复发生；
203. 好结果依赖一套固定方法；
204. 这套方法能够写成明确的输入、步骤、边界和输出。

本章不再增加团队任务看板的业务功能。

唯一任务是：

> **把已经验证过的项目变更审查方法，写成一个仓库级 Skill（技能工作流）。**

Skill 名称为：

```
project-change-review
```

它只负责审查，不负责修改、提交或合并。

OpenAI 当前将 Skill 定义为可复用工作流：它可以把任务说明、参考资料和可选脚本放在一个目录中，让 ChatGPT 或 Codex 在相同任务出现时遵循同一套方法。Skill 目录至少包含 `SKILL.md`，其中必须有 `name` 和 `description`。仓库级 Skill 通常放在项目中的 `.agents/skills` 目录下。OpenAI Developers：Build skills

二、主体

29.1 Skill 解决的不是“提示词太长”，而是方法无法稳定复用

把一段长提示词保存下来，并不自动等于建立了 Skill。

一个可复用能力至少要说明：

- 什么情况下应该使用；
- 什么情况下不应该使用；
- 开始前需要读取什么；
- 必须执行哪些步骤；
- 哪些行为被禁止；
- 最终结果怎样组织；
- 无法完成时怎样停止。

例如，下面这段话仍然只是一次性请求：

```
帮我看看这次修改有没有问题，重点看代码、文档和测试。
```

它没有说明：

- 比较哪个分支或提交范围；
- 是否允许修改；
- 哪些项目规则优先；
- 什么算阻断问题；
- 结论是否必须带文件位置；
- 没有发现问题时怎样报告。

本章建立的 Skill，要把这些判断固定下来。

29.2 先确定 Skill 的作用域

Skill 可以放在不同位置。

本章只使用仓库级位置：

```
team-task-board/
└── .agents/
    └── skills/
        └── project-change-review/
            └── SKILL.md
```

选择仓库级 Skill，是因为这套审查方法依赖当前项目的具体规则：

- 要读取当前项目的 `AGENTS.md`；
- 要尊重任务字段和本地存储不变量；
- 要检查项目自己的验证文档；
- 输出中的文件引用只对这个仓库有意义。

如果把它直接放到用户级目录，让所有项目都自动可见，可能造成两个问题：

- 其他项目误用团队任务看板的规则；
- Skill 描述越来越宽泛，最后失去明确边界。

因此，本章采用原则是：

> **先在单个仓库中证明有用，再决定是否扩大作用域。**

29.3 Skill 与 AGENTS.md 的职责不同

第 21 章已经建立 `AGENTS.md`。

两者都能影响 Codex 行为，但职责不同。

文件	主要回答的问题	适用方式
AGENTS.md	在这个项目中始终要遵守什么规则	Codex 进入作用范围后自动读取
SKILL.md	遇到某类任务时具体按什么流程执行	显式调用或根据描述匹配

例如：

`AGENTS.md` 可以规定：

- 不更换本地存储键；
- 不把用户输入直接拼接成 HTML；
- 修改前后都要检查 Git 状态；
- 业务功能需要人工浏览器验证。

`project-change-review` Skill 则规定：

- 怎样确定比较范围；
- 怎样读取差异；
- 按哪些类别审查；
- 怎样确认问题真实存在；
- 怎样输出阻断问题、重要问题和建议。

不要把全部项目规则复制进 Skill。

正确关系是：

> Skill 先读取并遵守 `AGENTS.md`，再执行自己的专项流程。

29.4 建立 Skill 目录

从干净的 `main` 开始：

```
git switch main
git pull --ff-only
git status --short
git switch -c docs/project-change-review-skill
```

创建目录：

```
New-Item -ItemType Directory -Force .agents/skills/project-change-review
New-Item -ItemType File -Force .agents/skills/project-change-review/SKILL.md
```

macOS 或 Linux 可以使用：

```
mkdir -p .agents/skills/project-change-review
touch .agents/skills/project-change-review/SKILL.md
```

随后检查范围：

```
git status --short
```

此时预期只有新 Skill 目录出现。

如果项目中同时出现其他未提交变化，应先识别来源，不要把它们与本章任务混在一起。

29.5 写好最小元数据

`SKILL.md` 顶部使用 YAML 前置元数据：

```
---
name: project-change-review
description: 对 team-task-board 中的指定 Git 差异或当前工作区执行只读项目变更审查，检查范围、行为与数据、文档和验证证据。用于用户明确要求审查、review、检查 diff 或判断是否可以提交时；不用于实现、修复、提交或合并修改。
---
```

`name` 应该稳定

名称用于识别和显式调用。

本章使用小写英文和连字符：

```
project-change-review
```

不要频繁改名。名称变化可能影响：

- 使用者的调用习惯；
- 定时任务中的显式引用；
- 团队文档；
- 后续 Plugin 分发。

`description` 必须包含触发边界

Codex 可能先根据 Skill 的名称和描述判断是否适用，再决定是否加载完整正文。

所以描述不能只写：

```
帮助审查项目。
```

更有效的描述要回答：

- 审查什么对象；
- 关注哪些方面；
- 哪些请求应触发；
- 哪些请求不应触发。

本章明确写入“不用于实现、修复、提交或合并”，是为了防止 Skill 在用户要求修改代码时自行接管任务。

29.6 写入审查流程

将下面内容写入 `SKILL.md` 正文：

```
# 项目变更审查

## 目标

对用户指定的 Git 比较范围或当前工作区执行只读审查，判断变化是否符合项目规则、任务范围和验证要求。

## 开始前

1. 确认当前工作目录和 Git 仓库。
2. 读取作用范围内的 AGENTS.md。
3. 读取 docs/project-context.md 和 docs/current-state.md。
4. 确认用户要求的比较对象；若没有说明，先询问或只审查当前未提交差异，不自行假定远程分支。
5. 检查 git status，识别是否存在任务外变化。

## 审查步骤

1. 列出实际变化文件和差异统计。
2. 检查变化是否超出任务范围。
3. 检查业务行为、数据字段、旧数据兼容和错误处理。
4. 检查用户输入是否安全渲染，是否出现未经说明的外部写入或权限变化。
5. 检查 README、项目状态、验证记录和代码是否一致。
6. 检查验证证据是否与修改风险匹配。
7. 对每条候选问题重新打开相关文件确认，不根据文件名或常见模式猜测。

## 输出格式

### 阻断问题
- 仅列出在提交或合并前必须处理的问题。

### 重要问题
- 列出不一定阻断，但可能造成错误、误导或维护风险的问题。

### 改进建议
- 列出可选优化，不与缺陷混写。

### 已核对范围
- 列出比较对象、读取的关键文件和实际执行的检查。

每条问题必须包含：
- 结论；
- 文件路径和相关位置；
- 证据；
- 影响；
- 最小处理建议。
```

```
若没有发现问题，明确写出“未发现阻断问题”，并说明实际核对过什么，不得只输出“看起来没问题”。

## 边界

- 不修改、创建、移动或删除文件。
- 不执行提交、推送、合并或发布。
- 不读取或输出密钥内容。
- 不把未运行的检查写成已通过。
- 不把推测写成事实。
```

这个版本保持“说明型 Skill”。

它没有：

- 脚本；
- 模板目录；
- 外部连接；
- UI 元数据；
- 自动提交逻辑。

对第一个 Skill 来说，这种克制很重要。

29.7 检查 Skill 是否被识别

保存文件后，重新打开或刷新 Codex 中的项目。

在 CLI 或 IDE 中可以查看 Skill 列表，或直接显式提及：

```
$project-change-review
```

如果 Skill 没有出现，依次检查：

205. 文件是否确实叫 `SKILL.md`；
206. 目录是否位于 `.agents/skills/project-change-review/`；
207. 前置元数据是否完整；
208. `name` 和 `description` 是否存在；
209. Codex 是否需要重启或重新加载项目；
210. 当前工作目录是否位于该仓库范围内。

不要因为没有立即显示，就将 Skill 复制到多个目录。重复名称不会自动合并，反而可能让选择器出现多个近似项。

29.8 用显式调用完成第一次测试

第一次测试不要依赖隐式触发。

明确调用：

```
请使用$project-change-review，对当前仓库的 main~1..main 执行只读审查。

要求：
1. 先确认比较范围和实际变化文件；
2. 读取 AGENTS.md、docs/project-context.md 和 docs/current-state.md；
3. 按 Skill 规定的四段格式输出；
4. 不修改文件；
5. 不提交、推送或合并；
6. 所有问题必须带文件证据。
```

这里选择 `main~1..main`，是为了审查一个已经合并的、范围清楚的历史变化。

如果最近一次提交并非适合练习的独立变化，可以指定明确提交：

```
<起始提交>..<结束提交>
```

不能为了方便，默认审查整个仓库历史。

29.9 判断测试是否真正通过

Skill 测试通过，不等于它成功输出了一段文字。

需要检查：

- 是否读取了项目规则；
- 是否使用了用户指定的 Git 范围；
- 是否保持只读；
- 是否区分阻断、重要和建议；
- 是否给出真实文件路径；
- 是否把没有执行的浏览器测试写成“通过”；
- 没有问题时是否说明核对范围；
- 是否出现与团队任务看板无关的通用检查清单。

如果输出过于泛化，例如：

```
建议增加更多测试，提高代码质量。
```

就需要改进 Skill，让它要求：

- 指出缺少哪项验证；
- 对应哪个修改风险；
- 现有证据为什么不足。

如果输出擅自修改文件，则说明边界仍不够明确，必须在提交 Skill 前修订。

29.10 再测试一次“不应触发”的请求

除了测试“应该使用”，还要测试“不应该使用”。

例如：

```
请为任务卡增加负责人头像，并完成代码修改。
```

这不是审查任务。

如果 Codex 自动选择 `project-change-review` 并停留在只读审查，说明描述过宽。

正确行为应是：

- 将它识别为功能开发任务；
- 不把审查 Skill 当作实现流程；
- 仍然遵守 `AGENTS.md`。

Skill 质量不仅取决于触发成功，也取决于避免错误触发。

29.11 检查差异并提交 Skill

测试通过后，检查当前变化：

```
git status --short
git diff -- .agents/skills/project-change-review/SKILL.md
```

确认只新增一个 Skill 文件后：

```
git add .agents/skills/project-change-review/SKILL.md
git diff --staged --stat
git diff --staged
git commit -m "添加项目变更审查 Skill"
```

随后按第 20 章已经掌握的流程完成审查和合并。

回到主分支：

```
git switch main
git pull --ff-only
git status --short
```

本章结束时，Skill 已经成为仓库的一部分，而不是只存在于某个聊天中的提示词。

29.12 Plugin 在本章只需要理解到什么程度

Skill 适合编写工作流本身。

Plugin（插件）则更像可安装的分发包，可以包含：

- 一个或多个 Skills；
- MCP 连接；
- 展示和依赖元数据。

本章不创建 Plugin，原因是：

- 当前 Skill 只服务一个练习仓库；
- 还没有足够多的真实使用记录；
- 尚未证明它适合其他项目；
- 分发会引入版本、依赖和安装管理。

正确顺序是：

> **先在仓库中验证 Skill，再考虑是否需要通过 Plugin 分享。**

三、总结

本章完成了第一个仓库级 Skill：

```
.agents/skills/project-change-review/SKILL.md
```

你已经掌握：

- 何时值得把重复方法沉淀为 Skill；
- Skill 与 `AGENTS.md` 的职责差异；
- 仓库级 Skill 的最小目录；
- `name` 与 `description` 怎样控制识别和触发；
- 怎样写清输入、步骤、输出和停止条件；
- 怎样通过显式调用测试 Skill；
- 怎样测试错误触发；
- 为什么第一个 Skill 应保持说明型和只读边界。

本章掌握检查

- [ ] Skill 只服务项目变更审查；
- [ ] `SKILL.md` 包含 `name` 和 `description`；
- [ ] 描述写明应触发和不应触发的请求；
- [ ] Skill 先读取项目规则和知识包；
- [ ] 每条问题都要求文件证据；
- [ ] Skill 不修改、提交或合并；
- [ ] 已完成一次显式调用测试；
- [ ] Skill 已经通过独立提交进入 `main`。

四、结束语与引导

到这里，项目内部已经拥有一套可复用审查方法。

但 Codex 能够看到的主要内容仍然来自：

- 当前仓库；
- 本地文件；
- 已经写入 Skill 的说明。

真实任务中，你还会需要项目外的最新文档和工具上下文。

下一章进入 MCP，但只做最小的一步：

> **连接一个公开、只读的文档服务，完成一次来源可追踪的查询，不让外部工具获得写入权限。**

第 30 章：用 MCP 连接外部工具和数据

一、起始语

第 29 章中，团队任务看板已经拥有仓库级 Skill。

它能够告诉 Codex：

- 怎样审查项目变化；
- 要读取哪些项目文件；
- 结果怎样分类；
- 哪些操作不能执行。

但 Skill 主要解决“怎样做”的问题。

如果任务需要项目外的信息，例如：

- 某个开发工具的当前官方文档；
- 一个设计平台中的页面结构；
- 浏览器或测试工具提供的实时状态；
- 团队系统中的工单、代码审查或知识库；

仅靠本地仓库并不够。

MCP 的作用，就是让 ChatGPT 或 Codex 通过统一协议连接外部工具和上下文。

MCP 全称为 Model Context Protocol，中文可以理解为“模型上下文协议”。OpenAI 当前的官方说明是：MCP 可以让 Codex 访问第三方文档，也可以让它与浏览器、Figma 等开发工具交互。ChatGPT 桌面应用、Codex CLI 和 IDE 扩展可以共享同一 Codex 主机上的 MCP 配置。OpenAI Developers：Model Context Protocol

本章不连接企业系统，不配置密钥，也不执行外部写入。

唯一任务是：

> **添加一个公开文档 MCP，查看它实际提供的工具，并完成一次只读资料查询。**

二、主体

30.1 先区分 Skill 和 MCP

Skill 与 MCP 经常一起出现，但解决的问题不同。

能力	主要作用	本章示例
Skill	固定任务方法和输出标准	怎样审查项目变化
MCP	连接项目外的工具或资料	从文档服务取得当前资料

可以把它们理解为：

```
Skill：告诉 Codex 应该按什么流程工作
MCP：让 Codex 能够访问哪些外部能力
```

一个 Skill 可以不使用 MCP。

一个 MCP 连接也可以在没有 Skill 时被直接调用。

成熟工作流可能将两者组合，但零基础阶段应先分别理解，否则容易出现两个误区：

- 认为 MCP 本身会自动保证结果可靠；
- 认为写好 Skill 就等于拥有外部数据权限。

30.2 MCP 服务器、工具和资源分别是什么

连接 MCP 时，至少要理解三个对象。

MCP 服务器

它是提供能力的一端。

服务器可能：

- 在本机作为进程启动；
- 通过网络地址访问；
- 提供一个或多个工具；
- 返回资料、结构化结果或操作状态。

工具

工具是服务器允许模型调用的具体动作。

例如一个文档服务可能提供：

- 查找文档集合；
- 搜索主题；
- 读取指定章节。

一个代码托管服务可能提供：

- 读取 PR；

- 创建 Issue；
- 更新评论。

后者包含写入动作，风险明显更高。

资源或上下文

这是工具最终返回给模型的内容，例如：

- 文档片段；
- 文件元数据；
- 页面结构；
- 工单字段；
- 日志记录。

MCP 只提供访问通道，不自动保证内容正确、最新或适合当前任务。

30.3 为什么本章只连接公开文档服务

第一次练习应同时降低四类风险：

211. **身份风险**：不登录个人或企业账号；
212. **数据风险**：不接触客户、员工或内部项目数据；
213. **写入风险**：不创建、修改或删除外部对象；
214. **凭据风险**：不配置 Token、OAuth 或私有密钥。

OpenAI 官方 MCP 文档使用 Context7 作为 CLI 添加示例。它是一个面向开发者文档的公开 MCP 服务，可以通过本地命令启动。本章沿用这一公开示例，但只使用它实际提供的只读资料工具。OpenAI Developers：MCP 配置示例

必须注意：

> 出现在官方配置示例中，不代表该服务中的每一份内容都由 OpenAI 维护，也不代表所有返回结果都可以不经核对直接写入项目。

本章仍然要求检查来源名称、文档对象和返回范围。

30.4 添加前先检查本机条件

Context7 示例通过 `npx` 启动，因此本机需要可用的 Node.js 和 `npx`。

检查：

```
node --version
npx --version
codex --version
```

如果 `node` 或 `npx` 不可用，本章有两个选择：

- 使用 ChatGPT 桌面应用或 IDE 中当前可用的 MCP 添加界面，填写服务需要的命令；
- 只学习 MCP 的连接、权限和证据边界，不要求为了本章临时安装新的运行环境。

不要为了连接 MCP，在未知网站下载来历不明的可执行文件。

30.5 使用 CLI 添加公开文档 MCP

在终端中执行 OpenAI 官方文档给出的示例：

```
codex mcp add context7 -- npx -y @upstash/context7-mcp
```

这条命令的含义是：

- `codex mcp add`：向 Codex 配置中添加 MCP 服务器；
- `context7`：本地给这个服务器使用的名称；
- `--`：后面是启动服务器的命令；
- `npx -y @upstash/context7-mcp`：通过 `npx` 启动对应服务。

添加后查看列表：

```
codex mcp list
```

在交互式 CLI 中，也可以使用：

```
/mcp
```

查看当前可用服务器。

如果你使用 ChatGPT 桌面应用，当前官方路径是进入设置中的 MCP 服务器管理，添加服务器名称和启动命令，保存后重启。IDE 扩展也提供相应的 MCP 服务器管理入口。界面名称可能变化，因此应以“服务器列表、添加、保存、重启、查看连接状态”这五个动作理解，而不是死记按钮位置。

30.6 配置保存在哪里

Codex 默认将 MCP 配置保存在：

```
~/.codex/config.toml
```

受信任项目也可以使用项目级：

```
.codex/config.toml
```

本章使用用户级默认配置，原因是：

- Context7 不是团队任务看板专属服务；
- 仓库中不应提交个人机器上的启动配置；

- 当前没有团队统一安装要求；
- 不需要让项目克隆后自动获得外部连接。

ChatGPT 桌面应用、Codex CLI 和 IDE 扩展在同一 Codex 主机上共享配置，因此通常不需要在三个入口中重复添加。

不要把包含本机路径、环境变量或凭据的配置文件随意提交到仓库。

30.7 先查看工具，再决定是否调用

连接成功后，不要立刻提出宽泛请求：

```
帮我查一下项目相关资料。
```

先让 Codex 说明当前服务器实际暴露了什么：

```
请只检查当前已连接的 context7 MCP。

请说明：
1. 服务器是否连接成功；
2. 当前可用工具名称；
3. 每个工具是读取还是可能写入；
4. 工具需要哪些输入；
5. 本次是否需要网络、登录或凭据；
6. 不要调用工具，不要修改项目。
```

工具名称可能随服务器版本变化。

本书不要求死记具体名称，而要求你确认：

- 能否搜索或解析文档；
- 返回内容是否包含来源标识；
- 是否存在写入动作；
- 是否要求额外授权。

如果服务器实际暴露了与预期不符的写入工具，应停止并重新评估，不要因为本章标题写着“只读”就假设工具天然只读。

30.8 设置本次查询边界

本章查询的目标不是给团队任务看板增加功能，而是验证：

> 外部文档怎样作为证据进入 Codex 工作过程。

使用下面的请求：

```
请仅使用当前 context7 MCP 完成一次只读文档查询。

目标：
查找该服务器实际索引中与 OpenAI Codex Skill 有关的当前权威文档，核对以下两点：
```

```
1. 仓库级 Skill 的目录位置；
2. SKILL.md 最少需要哪些元数据字段。

要求：
- 优先使用 OpenAI 官方文档；
- 返回文档名称、维护方或来源标识；
- 区分文档原文支持的结论和你的推断；
- 若服务器没有目标官方文档，明确报告“未找到”，不要改用来源不明内容；
- 不修改项目文件；
- 不调用其他网络工具；
- 不保存任何外部内容到仓库。
```

这个查询与第 29 章直接相关，但不会改变 Skill。

它用来检查外部资料是否能支持我们已经采用的做法。

30.9 核对返回结果的四层证据

查询结束后，不要只看最终结论。

需要核对四层信息。

第一层：调用了哪个服务器

确认结果来自：

```
context7
```

如果 Codex 混用了普通网页搜索或其他连接，应该重新执行并收紧工具范围。

第二层：使用了什么文档对象

结果应说明：

- 文档名称；
- 维护方；
- 文档集合或来源标识；
- 与查询主题的对应关系。

“来自开发文档”仍然太模糊。

第三层：文档支持了什么

本次预期核对的是：

- 仓库级 Skill 位于 `.agents/skills` 作用范围；
- `SKILL.md` 至少包含 `name` 和 `description`。

如果外部结果只支持其中一点，就不能把两点都写成已验证。

第四层：哪些内容仍未验证

例如：

- 当前 Context7 索引是否同步到最新日期；
- 文档集合是否完整；
- 某个界面按钮是否与本机版本一致；
- 第 29 章 Skill 的实际触发效果是否正确。

外部文档可以说明产品规则，却不能替代本地功能测试。

30.10 “没有找到”也是有效结果

如果 MCP 没有提供目标官方文档，正确结果是：

```
本次未在当前 context7 索引中找到可确认维护方为 OpenAI 的目标文档，因此不使用返回的其他非官方资料支持产品口径。
```

不要为了完成练习而：

- 将博客当作官方说明；
- 让 Codex 根据记忆补齐；
- 修改查询目标到一个无关主题；
- 把“可能”写成“官方规定”。

MCP 的价值不只体现在成功取回内容，也体现在让资料边界可见。

30.11 将查询结果整理为临时证据记录

本章不向仓库写入新文档。

在当前聊天中保留一段简洁记录：

```
## MCP 查询记录

- 服务器：context7
- 查询主题：Codex 仓库级 Skill 位置与 SKILL.md 最小字段
- 使用来源：<实际文档名称与来源标识>
- 已支持结论：<逐条列出>
- 未支持或仍需本地验证：<逐条列出>
- 外部写入：无
- 项目文件变化：无
```

为什么不创建 `docs/mcp-query-log.md`？

因为这只是一次学习练习，还没有形成项目长期需要维护的知识。

第 22 章已经强调：

> 文档系统应保存会持续影响项目的内容，不保存每一次临时过程。

30.12 检查仓库确实没有变化

查询结束后运行：

```
git status --short
```

预期没有输出。

如果出现变化，要检查：

- MCP 服务是否在项目目录写入缓存；
- Codex 是否擅自保存了查询结果；
- 本机工具是否产生临时文件；
- 变化是否属于其他并行任务。

不要直接用 `git clean` 或删除全部未跟踪文件。先识别，再决定处理方式。

30.13 连接保留还是禁用

本章结束后有三种合理状态：

保留启用

适合：

- 近期仍会查开发文档；
- 工具明确只读；
- 不需要凭据；
- 启动成本可接受。

暂时禁用

适合：

- 不希望每次启动都加载；
- 当前项目暂时不需要；
- 想保留配置但减少工具干扰。

删除

适合：

- 服务来源无法确认；
- 工具范围与预期不符；
- 安装失败或产生异常文件；
- 后续确定不会再使用。

可以通过 Codex 的 MCP 管理界面或 CLI 帮助查看相应命令：

```
codex mcp --help
```

本章建议保留配置，但不让后续任务默认调用。

30.14 为什么不在本章配置 OAuth 和 Token

OpenAI 当前支持本地进程型和 HTTP 型 MCP 服务器，HTTP 服务器还可能使用 OAuth、Bearer Token 或其他请求头。

这些能力非常重要，但会引入：

- 凭据保存；
- 身份授权；
- 权限范围；
- 过期与撤销；
- 日志泄露；
- 写入审批。

零基础主线中第一次 MCP 练习，不应该把“连接协议”与“企业身份治理”同时展开。

完整密钥、外部写入和团队治理边界放到第 36 章。

三、总结

本章完成了第一次 MCP 只读连接：

- 理解 Skill 与 MCP 的职责差异；
- 认识服务器、工具和返回上下文；
- 使用 OpenAI 官方示例添加 Context7；
- 查看服务器和工具，而不是直接盲目调用；
- 用明确来源和停止条件完成一次文档查询；
- 区分文档支持、模型推断和本地验证；
- 接受“未找到官方资料”这一有效结果；
- 确认项目仓库没有发生变化；
- 不配置 OAuth、Token 或写入权限。

本章掌握检查

- [ ] 能说明 Skill 与 MCP 的不同；
- [ ] 能查看当前 MCP 服务器和工具；
- [ ] 本次连接不需要个人或企业凭据；

- [ ] 查询明确限制为只读；
- [ ] 返回结论包含实际来源标识；
- [ ] 没有将非官方资料伪装成官方事实；
- [ ] 没有修改团队任务看板；
- [ ] 已决定连接保留、禁用或删除的状态。

四、结束语与引导

项目现在拥有两类可扩展能力：

```
Skill：复用审查方法
MCP：接入项目外资料
```

下一步不继续增加连接，也不创建更复杂的 Skill。

我们把注意力转回项目本身。

当一次审查同时涉及代码、数据风险和文档验证时，所有中间过程都堆在一个聊天中，会让主线程越来越嘈杂。

第 31 章将第一次使用子智能体，但只做适合并行的只读任务：

> **让三个角色分别检查代码与数据流、输入与运行风险、文档与验证一致性，再由主智能体核对证据并汇总。**

第 31 章：用子智能体拆分和并行处理任务

一、起始语

前面的审查任务都由一个 Codex 主线程完成。

这种方式适合范围较小、依赖关系较强的任务。

但当审查对象扩大时，一个聊天可能同时出现：

- 目录探索；
- Git 差异；
- 数据流追踪；
- 输入安全检查；
- 文档核对；
- 测试记录；
- 大量中间日志。

这些内容会挤占主线程，让真正重要的任务范围、判断和结论被淹没。

子智能体工作流允许主智能体把相对独立的任务委派给多个代理线程，再收集和汇总结果。OpenAI 当前建议优先将子智能体用于探索、测试、分诊和总结等读取型工作；对多个智能体同时修改代码要更加谨慎，因为并行写入容易制造冲突和协调成本。OpenAI Developers：Subagents

本章不创建自定义智能体文件，也不允许任何代理修改项目。

唯一任务是：

> **让三个子智能体并行完成一次项目只读审查，再由主智能体核对证据和处理冲突。**

这次成功只能说明：

> 已完成一次人工触发和结果核对，满足只读定时试运行条件。

不能据此宣称工作流已经成熟稳定。

二、主体

31.1 什么任务适合拆给子智能体

适合并行的任务通常具备三个特征。

边界可以独立描述

每个角色都能明确知道自己检查什么、明确不检查什么。

例如：

- 一个角色追踪代码和数据流；
- 一个角色检查输入和运行风险；
- 一个角色核对文档和验证证据。

主要工作是读取和分析

多个角色同时读取同一仓库，一般不会产生文件冲突。

结果能够由主智能体统一验证

每个角色都必须返回：

- 文件路径；
- 证据；
- 影响；
- 置信度或未确认项。

主智能体才能判断多份结果是否重复、矛盾或缺少依据。

31.2 哪些任务不适合本章并行

本章不并行处理：

- 多个角色同时修改 `app.js`；
- 一个角色改数据结构，另一个角色改渲染；
- 多个角色分别提交和推送；
- 自动解决合并冲突；
- 自动发布；
- 高风险安全修复。

这些任务不是永远不能并行，而是需要更强的隔离、任务划分、所有权和集成机制。

零基础阶段应先建立一个安全起点：

> **并行读取，集中判断，单点写入。**

本章甚至不进入最后的“单点写入”，只形成审查结果。

31.3 主智能体和子智能体分别负责什么

角色	主要责任
主智能体	理解总目标、分配角色、规定边界、等待结果、核对证据、消除重复、形成最终结论
子智能体	在限定范围内独立读取、分析并返回结构化结果

主智能体不能把任务发出去后直接拼接三份答案。

它仍然需要：

- 判断子任务是否覆盖完整；
- 确认没有角色越界；
- 复查高风险结论；
- 解决互相矛盾的判断；
- 区分真实问题和重复建议。

子智能体降低的是中间上下文噪声，不是主智能体的责任。

31.4 确认审查基线

本章从干净主分支开始：

```
git switch main
git pull --ff-only
git status --short
git log --oneline -3
```

确认：

- 第 29 章 Skill 已经进入 `main`；
- 第 30 章没有修改项目文件；
- 当前工作区干净；
- 没有其他并行任务正在写入相同仓库。

本章审查对象是当前 `main` 整体，不是一个待合并分支。

因此，任务不是“这次差异能不能合并”，而是：

> 当前项目的代码、风险控制、文档和验证是否仍然相互一致。

31.5 给三个角色设置互斥边界

角色 A：代码与数据流审查

只检查：

- 页面初始化；
- 任务读取和保存；
- 新增与编辑路径；
- 状态更新和删除；
- 搜索筛选与完整任务数组的关系；
- 旧数据兼容。

重点文件：

```
js/constants.js
js/seed-data.js
js/storage.js
js/filters.js
js/render.js
js/app.js
```

不检查：

- 文案是否易读；
- README 是否完整；
- CSS 视觉细节；
- 外部文档。

角色 B：输入与运行风险审查

只检查：

- 用户输入怎样进入页面；
- 是否安全渲染；
- 本地存储读取和写入失败；
- 非法或缺失日期；
- 编辑状态是否可能残留；
- 可能造成数据覆盖或静默失败的路径。

重点文件：

```
index.html
js/storage.js
js/render.js
js/app.js
```

不进行渗透测试，不访问网络，不读取项目外数据。

角色 C：文档与验证一致性审查

只检查：

- README 描述的功能是否真实存在；

- `docs/project-context.md` 与当前代码是否一致；
- `docs/current-state.md` 是否反映最新状态；
- 验证记录是否覆盖对应风险；
- `docs/contributor-quickstart.md` 中的启动和检查步骤是否可执行；
- 新 Skill 的用途是否与项目规则冲突。

重点文件：

```
README.md
AGENTS.md
docs/
.agents/skills/project-change-review/SKILL.md
```

不重新审查全部 JavaScript 实现。

31.6 设计统一的返回格式

三个角色使用同一格式，便于主智能体汇总：

```
## 检查范围

## 已确认事实

## 候选问题
- 严重程度：阻断 / 重要 / 建议
- 结论：
- 文件与位置：
- 证据：
- 影响：
- 是否需要主智能体复核：是 / 否

## 未能确认

## 执行边界
- 文件修改：无
- 外部写入：无
- 未执行检查：
```

如果角色没有发现问题，也必须说明：

- 实际读取了哪些文件；
- 追踪了哪些流程；
- 哪些内容没有执行。

31.7 发出并行审查请求

在 Codex 聊天中明确要求使用子智能体：

```
请对当前 team-task-board 主分支执行一次只读并行审查。

请生成三个子智能体：
```

```
1. 代码与数据流审查员
   - 检查初始化、读取保存、新增编辑、状态更新、删除、搜索筛选和旧数据兼容。

2. 输入与运行风险审查员
   - 检查安全渲染、本地存储异常、日期边界、编辑状态和数据覆盖风险。

3. 文档与验证一致性审查员
   - 检查 README、AGENTS.md、docs 目录、验证记录和 project-change-review Skill 是否与真实项目一致。

共同要求：
- 先读取 AGENTS.md；
- 全程只读；
- 不创建、修改、移动或删除文件；
- 不运行需要外部网络或凭据的工具；
- 不提交、推送或合并；
- 每条问题必须带真实文件位置和证据；
- 不把未执行的测试写成已通过；
- 按指定结构返回结果。

请等待三个子智能体全部完成后，再由主智能体：
- 去重；
- 复核阻断和重要问题；
- 处理冲突；
- 输出最终审查结论。
```

当前 Codex 版本可以在桌面应用、CLI 和 IDE 扩展中显示子智能体活动。CLI 可使用`/agent`查看和切换代理线程。具体界面可能变化，本章只要求你能看到：

- 哪些角色正在运行；
- 哪些已经完成；
- 每个角色返回了什么；
- 主线程是否仍在等待。

31.8 观察代理是否真正独立

并行不等于把同一问题重复问三次。

打开子线程，检查：

- 角色 A 是否主要追踪 JavaScript 数据流；
- 角色 B 是否聚焦输入、存储和运行风险；
- 角色 C 是否主要读取文档和验证材料；
- 是否有角色开始修改文件；
- 是否有角色擅自使用 MCP 或网络；
- 是否有角色越界审查全部内容。

如果角色边界明显错误，可以要求主智能体停止或重新引导该子智能体。

不要因为已经消耗了一部分时间，就保留明显无效的代理结果。

31.9 为什么要等待全部结果

如果主智能体在第一个角色返回后就开始下结论，可能出现：

- 将代码行为问题误写成唯一风险；
- 忽略文档与验证证据；
- 重复后续角色尚未完成的判断；
- 先入为主地影响汇总。

本章要求主智能体等待三个结果全部返回。

这并不表示所有任务都必须等待最慢的代理。

但当前练习的目标是形成完整审查，因此不能在角色缺失时宣称完成。

31.10 主智能体先去重，再判断严重程度

三个角色可能发现同一问题的不同侧面。

例如：

- 角色 A 发现编辑保存会重建任务对象；
- 角色 B 担心缺失字段被覆盖；
- 角色 C 发现验证记录没有覆盖旧数据编辑。

这可能是：

- 一个共同根因；
- 两个不同问题；
- 或者只是对正常设计的不同描述。

主智能体应先建立候选问题表：

候选问题	来自角色	文件证据	是否重复	是否需复核
编辑保存可能丢失字段	A、B	js/app.js	可能重复	是
旧数据编辑验证不足	C	验证文档	独立	是

然后重新打开相关代码和文档。

不能按“两个角色都提到，所以一定正确”来判断。

31.11 处理互相冲突的结果

常见冲突包括：

一个角色说存在问题，另一个说没有

例如：

- 角色 B 认为用户标题可能通过 `innerHTML` 进入页面；
- 角色 A 认为渲染使用 `textContent`，没有直接 HTML 拼接。

主智能体必须打开 `render.js` 确认实际实现。

最终输出应写：

> 角色 B 提出候选风险，但复核发现用户内容通过 textContent 写入，因此不列为问题。

两个角色使用了不同严重程度

例如：

- 角色 A 将文档遗漏列为建议；
- 角色 C 将其列为重要问题。

主智能体需要根据影响判断：

- 是否会让新贡献者执行错误操作；
- 是否会造成数据损坏；
- 是否阻断提交或发布。

严重程度由影响决定，不由角色名称决定。

角色结论缺少证据

没有文件位置、没有实际代码、只有“可能”或“通常”的结论，不应进入正式问题列表。

可以放入“未能确认”，也可以直接删除。

31.12 使用 Skill 约束最终输出

第 29 章的 Skill 可以作为最终汇总格式，而不是要求每个子智能体都完整运行一遍 Skill。

主智能体在汇总阶段应遵守：

- 阻断问题；
- 重要问题；
- 改进建议；
- 已核对范围。

每条问题包含：

- 文件路径；
- 证据；

- 影响；
- 最小处理建议。

如果没有阻断问题，写明：

```
未发现阻断问题。
```

同时说明这只是静态和文档层面的只读审查，不代表完成了新的浏览器回归。

31.13 标准练习的通过条件

本章不预设必须发现 Bug。

通过条件是：

- 三个角色都完成了限定范围；
- 没有文件变化；
- 高风险结论经过主智能体复核；
- 重复和冲突已经处理；
- 最终报告能区分事实、候选和未确认项；
- 没有把静态审查写成完整功能验证。

本书持续使用的团队任务看板项目在前几章已经完成验证和交付，因此合理结果可能是：

- 未发现阻断问题；
- 一到两项非阻断建议；
- 明确列出尚未重新执行的浏览器场景。

不要为了证明子智能体有价值，要求它们必须找出问题。

31.14 检查 Git 和外部状态

审查结束后运行：

```
git status --short
```

预期没有输出。

同时确认：

- 没有新分支；
- 没有新提交；
- 没有 MCP 外部写入；
- 没有自动创建报告文件；
- 没有改变定时任务。

本章成果存在于经过人工核对的审查结果中，不需要再创建一份永久文档。

31.15 记录这次运行的真实成熟度

一次成功运行可以证明：

- 任务能够拆分；
- 三个角色边界基本可执行；
- 输出可以汇总；
- 只读边界没有被突破；
- 人能够检查结果。

它不能证明：

- 每周运行都会稳定；
- 任何项目都适用；
- 所有模型设置都一致；
- 无人值守时不会越界；
- 结果已经可以自动触发修改。

因此，当前状态应记录为：

> **已完成一次人工触发和结果核对，满足只读定时试运行条件。**

31.16 成本和规模边界

每个子智能体都会独立消耗模型和工具资源。

并行角色越多，不一定越好。

本章只使用三个角色，是因为它们覆盖三个互相区分的审查面。

不建议再增加：

- “代码质量审查员”；
- “可维护性审查员”；
- “架构审查员”；
- “最佳实践审查员”；

这些角色容易与现有范围重叠，产生大量泛化建议。

一个简单判断是：

> 如果你无法用一句话说明某个角色独有的输入和输出，它可能不值得独立成为子智能体。

三、总结

本章完成了第一次子智能体并行审查：

- 将项目审查拆成三个读取型角色；
- 为每个角色设置互斥边界；
- 统一返回结构；
- 明确要求等待全部结果；
- 观察和纠正代理越界；
- 由主智能体去重、复核和处理冲突；
- 使用项目变更审查 Skill 约束最终输出；
- 保持仓库和外部系统无变化；
- 将成熟度准确标记为“满足只读定时试运行条件”。

本章掌握检查

- [ ] 能判断任务是否适合并行；
- [ ] 三个角色的检查范围互相区分；
- [ ] 所有子智能体均为只读；
- [ ] 主智能体等待了全部结果；
- [ ] 阻断和重要问题经过重新核对；
- [ ] 冲突结论没有直接拼接；
- [ ] 最终报告区分事实、建议和未确认项；
- [ ] 项目文件和 Git 状态没有变化。

四、结束语与引导

到这里，同一套检查已经经历：

```
人工单线程审查
→ 写成仓库级 Skill
→ 三角色人工触发并行审查
→ 主智能体核对结果
```

它仍然不是成熟的无人值守流程。

但因为任务只读、范围清楚，并且已经完成一次人工运行，可以进入下一步：

> **创建一个每周定时试运行，立即手动触发一次，检查结果后再暂停。**

第 32 章：把验证过的只读检查安排为定时试运行

一、起始语

很多自动化问题，不是因为工具不会执行，而是因为人过早把尚未验证的流程放到了后台。

典型风险包括：

- 提示词仍然依赖当前聊天中的临时信息；
- 执行范围没有写清楚；
- 任务需要审批，却在无人值守时无法询问；
- 工具拥有写入权限；
- 结果没有固定格式；
- 一次成功就被当成长期稳定。

第 31 章已经完成一次人工触发、人工观察和人工核对。

而且这项检查具备几个适合试运行的条件：

- 主要读取本地仓库；
- 不修改文件；
- 不提交或推送；
- 不需要密钥；
- 没有重要发现时可以简短结束；
- 结果可以由人继续审查。

本章只将它安排为定时**试运行**。

唯一任务是：

> **创建一个每周只读项目检查，立即运行一次，查看结果和权限边界，随后暂停。**

OpenAI 当前支持在 ChatGPT 或 Codex 聊天中创建和更新定时任务。项目型定时任务可以在本地项目或独立后台 Worktree 中运行；定时任务会在无人值守状态下使用默认沙箱设置，因此应从最小权限开始，并在正式安排前先手工测试提示词。OpenAI Developers：Scheduled tasks

二、主体

32.1 什么工作适合安排为定时任务

适合的任务通常满足：

- 触发时间或周期明确；
- 每次输入来源稳定；
- 执行步骤能够写成持久提示；
- 结果可以独立阅读；
- 失败时能够安全停止；
- 无人值守时不依赖临时批准。

本章的每周检查符合这些条件。

32.2 哪些工作不适合本章定时运行

本章不安排：

- 自动修改代码；
- 自动修复审查问题；
- 自动提交和推送；
- 自动合并 PR；
- 自动更新依赖；
- 自动访问需要登录的企业系统；
- 自动使用全权限沙箱。

这些任务可能需要：

- Worktree 隔离；
- 写入权限；
- 组织策略；
- Secrets；
- 审批与回滚；
- 更严格的测试。

它们放到后续安全和团队治理章节讨论。

32.3 独立定时任务还是聊天内定时任务

当前产品支持两种常见方式。

聊天内定时任务

适合：

- 后续运行需要沿用当前聊天上下文；
- 正在跟踪一个持续事件；
- 结果应回到同一讨论中。

独立定时任务

适合：

- 每次运行都可以从保存的提示开始；
- 结果需要作为独立运行记录查看；
- 任务本身有长期固定目标。

本章选择：

> **独立定时任务。**

原因是每周项目检查应依赖持久提示和仓库事实，而不是依赖第 31 章聊天中的临时上下文。

32.4 Local 还是 Worktree

Git 项目的定时任务可以选择直接在本地项目中运行，也可以使用后台 Worktree。

运行位置	优点	风险
Local（本地项目）	直接读取当前项目，路径简单	如果任务写文件，可能影响正在编辑的工作区
Worktree（工作树）	与主工作区隔离	会产生额外工作树，需要管理和清理

本章任务严格只读，因此选择：

> **Local。**

它不需要为零写入任务创建额外 Worktree。

如果未来任务会修改文件，即使只是生成报告，也应重新评估是否使用 Worktree，不能沿用本章选择。

32.5 先把提示词改写成可独立运行的版本

第 31 章的提示词依赖“当前项目”和“这次审查”。

定时任务需要更持久。

使用下面版本：

```
在 team-task-board 项目中执行每周只读项目检查。

开始前：
1. 确认当前工作目录属于 team-task-board；
2. 读取 AGENTS.md、docs/project-context.md 和 docs/current-state.md；
3. 检查 git status --short；
4. 若工作目录错误、项目不存在或存在无法识别的未提交变化，停止并报告，不继续审查。

检查范围：
1. 查看当前本地 main 最近 7 天的提交；
```

```
2. 若存在提交，启动三个只读子智能体：代码与数据流、输入与运行风险、文档与验证一致性；
3. 等待三个角色全部完成，由主智能体复核文件证据、去重并处理冲突；
4. 使用$project-change-review 规定的严重程度和输出格式形成最终结论；
5. 检查 README、docs 中的相对链接和关键文件是否仍然存在；
6. 检查项目状态文档是否与当前功能和最近提交明显冲突；
7. 若当前定时运行环境不支持子智能体，停止并明确报告，不要伪装成并行执行；
8. 不运行需要网络、登录、密钥或外部应用的工具。

输出规则：
- 有阻断或重要问题时，按严重程度列出文件证据、影响和最小处理建议；
- 只有改进建议时，控制在 5 条以内；
- 最近 7 天没有提交且没有状态异常时，只输出简短的“本周无需要处理的变化”；
- 明确列出未执行的浏览器测试和外部检查；
- 不创建、修改、移动或删除文件；
- 不提交、推送、合并或发布；
- 不启动修复任务。
```

这段提示包含：

- 工作对象；
- 开始条件；
- 停止条件；
- 时间范围；
- 使用的 Skill；
- 输出阈值；
- 禁止行为。

32.6 手工再测试一次持久提示

即使第 31 章已经运行过类似流程，改写后的定时提示仍需手工测试。

在普通聊天中执行一次：

```
请按下面的每周只读项目检查提示立即执行一次，但不要创建定时任务。

<粘贴完整持久提示>
```

检查：

- 是否能找到项目；
- 是否正确读取最近 7 天提交；
- 没有提交时是否简短结束；
- 是否启动了三个边界清楚的只读子智能体；
- 是否等待全部结果并由主智能体复核；
- 是否显式调用 Skill 约束最终输出；
- 是否保持只读；
- 是否出现无依据的泛化建议；
- 输出是否适合每周阅读。

如果手工测试失败，应先修正提示，不进入调度。

32.7 创建每周试运行任务

在 ChatGPT 桌面应用或支持定时任务的 ChatGPT 界面中提出：

```
请为当前 team-task-board 项目创建一个独立定时任务。

名称：团队任务看板每周只读检查
时间：每周一上午 9:00，按当前账号或设备显示的时区
运行位置：当前本地项目
提示：
<粘贴已经手工测试通过的完整持久提示>

创建后不要修改项目文件，先让我检查任务设置。
```

如果界面允许选择：

- 项目：选择 `team-task-board`；
- 运行方式：Local；
- 模型和推理强度：先使用默认；
- 权限：只读或能够满足读取任务的最小权限；
- 输出位置：独立 Scheduled 运行记录。

具体按钮和布局可能变化，应核对最终任务卡中的名称、周期、项目和权限，而不是只相信聊天回复。

32.8 为什么选择每周一上午 9 点

这只是练习默认时间，不是普遍最佳时间。

选择它的原因是：

- 周初适合回看上一周变化；
- 不需要高频轮询；
- 结果可以进入本周维护安排；
- 不会制造大量重复运行。

真实项目应根据：

- 提交频率；
- 团队工作时间；
- 机器在线状态；
- 审查人员可用时间；
- 结果处理节奏；

调整周期。

32.9 检查机器和应用条件

本地项目定时任务通常需要：

- 电脑在计划时间处于开机状态；
- ChatGPT 桌面应用保持运行；
- 项目路径仍然存在；
- 磁盘和文件权限没有变化；
- Git 仓库仍然可读。

如果用户移动了项目目录、外接磁盘未连接或应用退出，任务可能失败。

本章不把本地定时任务描述为云端永远在线服务。

32.10 使用 Run now 立即试跑

创建后不要等待下周。

使用 Run now（立即运行）或相应操作，触发第一次试跑。

观察：

- 任务是否进入运行状态；
- 是否读取正确项目；
- 是否要求不应出现的批准；
- 是否尝试访问网络；
- 是否产生文件变化；
- 是否按三个角色完成并行检查，或在环境不支持时安全停止；
- 结果是否进入 Scheduled 记录；
- 失败时是否留下可读原因。

定时试跑的目标不是证明结果完美，而是尽早发现调度环境与普通聊天的差异。

32.11 核对第一次运行结果

第一次结果至少应包含：

- 当前项目和分支；
- 检查的时间范围；
- 最近 7 天是否存在提交；
- 实际审查范围；
- 三个角色的完成状态和主智能体复核结论；
- 阻断、重要或建议；
- 未执行检查；

- 文件修改为无。

如果结果只写：

```
项目一切正常。
```

这不够。

应至少说明：

```
检查了本地 main 最近 7 天提交、Git 状态、README 与 docs 关键链接；未发现阻断或重要问题；未执行浏览器回归和外部网络检查；未修改文件。
```

32.12 检查项目确实没有变化

第一次运行结束后，在项目中执行：

```
git status --short
```

预期没有输出。

还要检查：

- 没有新分支；
- 没有新 Worktree；
- 没有新提交；
- 没有自动生成报告文件；
- 没有修改 `current-state.md`；
- 没有调用 MCP 写入工具。

如果产生变化，本章任务不通过，应暂停并调查。

32.13 常见失败与处理

项目路径不存在

结果应报告路径问题并停止。

不要自动搜索整块磁盘寻找同名项目。

工作区有未提交变化

结果应说明：

- 哪些文件变化；
- 无法确认是否属于用户正在进行的工作；
- 本次停止。

不要将用户未提交内容纳入每周审查，也不要自动丢弃。

Skill 没有被识别

检查：

- 定时任务工作目录是否正确；
- `.agents/skills/project-change-review/SKILL.md` 是否存在；
- 提示中是否显式使用`$project-change-review`；
- 运行环境是否支持项目 Skill。

输出过长

收紧：

- 时间范围；
- 建议数量；
- 只有重要发现才展开；
- 无变化时使用单段摘要。

任务尝试联网

如果本章不需要网络，应保持网络不可用或在提示中明确禁止。

不要因为联网调用失败就扩大权限。

32.14 暂停任务

第一次试跑和人工核对完成后，将任务暂停。

本章特意不让它立即长期运行，原因是：

- 只验证了一次调度环境；
- 还没有观察多个周期；
- 当前是教学项目；
- 第 36 章尚未完成完整安全治理；
- 用户应学会控制自动化，而不是只会创建。

暂停后检查任务状态：

```
Paused / 已暂停
```

保留第一次 Scheduled（定时任务）运行记录，用于后续比较。

32.15 什么时候可以重新启用

至少满足：

- 提示在普通聊天和定时环境都按预期运行；
- 连续几次人工触发没有越界；
- 项目路径稳定；
- 输出有人负责查看；
- 失败和无变化结果都清楚；
- 权限保持最小；
- 不会与其他自动化重复。

即使重新启用，也要定期检查：

- 任务是否仍有价值；
- Skill 是否已经变化；
- 项目结构是否改变；
- 运行结果是否出现漂移；
- 是否应降低或提高频率。

32.16 定时任务的权限原则

定时任务无人值守运行，无法像普通聊天一样随时向人确认。

因此，本章形成四条原则：

215. **能只读就不写入；**
216. **能本地完成就不开放网络；**
217. **能简短报告就不自动修复；**
218. **发现不确定状态就停止。**

OpenAI 当前说明，Scheduled 任务使用默认沙箱设置运行。高权限模式会增加后台任务修改文件和访问网络的风险，因此应从最窄访问开始。

32.17 不把 Scheduled 当成传统后台服务

定时任务可以按计划运行，但它不是：

- 永远在线的服务器；
- 数据库；
- 监控平台替代品；
- 自动发布流水线；
- 无限制轮询器。

它更适合：

- 定期检查；
- 生成摘要；
- 跟进状态；
- 执行已经验证的重复工作。

当任务需要严格服务等级、秒级告警或复杂状态管理时，应使用专门系统，而不是继续扩张聊天定时任务。

三、总结

本章完成了第一次定时试运行：

- 判断只读项目检查适合调度；
- 选择独立任务而不是聊天内任务；
- 因为零写入而选择 Local；
- 将提示改写为可独立运行的持久版本；
- 在普通聊天中重新测试；
- 创建每周一上午 9 点的任务；
- 使用 Run now 立即试跑；
- 核对结果、权限和 Git 状态；
- 处理常见失败；
- 在验证后主动暂停。

本章掌握检查

- [ ] 定时任务只执行只读检查；
- [ ] 持久提示包含开始条件和停止条件；
- [ ] 已在普通聊天中测试新提示；
- [ ] 任务绑定正确项目和 Local 运行位置；
- [ ] 第一次运行没有要求无关权限；
- [ ] 三个只读角色均完成，或环境不支持时安全停止；
- [ ] 结果包含真实检查范围和未执行事项；
- [ ] 项目没有文件、分支或提交变化；
- [ ] 任务当前处于暂停状态。

四、结束语与引导

第 29—32 章完成了一条完整的可复用能力路线：

```
重复审查方法
→ 仓库级 Skill
→ 只读 MCP 资料连接
→ 三角色子智能体并行审查
→ 每周只读定时试运行
```

项目没有新增业务功能，也没有将外部连接和后台执行变成默认高权限能力。

下一篇开始进入更偏工程集成的能力：

- 非交互模式；
- SDK 导览；
- GitHub 质量链；
- 沙箱、审批、密钥和团队规范。

第 33 章会先从最小范围开始：

> **使用 `codex exec` 完成一次有输入、输出和退出状态的只读脚本任务。**

第 33 章：使用非交互模式完成最小脚本任务

一、起始语

前面的章节中，你主要通过桌面端、命令行交互界面、IDE 和云端聊天使用 Codex。

这些入口都有一个共同特点：

> 人在任务进行过程中，可以继续追问、批准操作、修改方向或停止执行。

但有些任务不需要持续对话。

例如：

- 每次提交前检查仓库是否存在明显风险；
- 将一段固定提示交给 Codex 并保存结果；
- 在脚本中获取项目摘要；
- 让后续程序根据成功或失败决定是否继续；
- 在持续集成或定时任务中执行已经验证过的检查。

这类场景需要的不是新的聊天窗口，而是：

> **从命令开始，完成一次任务，输出结果，再用明确状态结束。**

Codex CLI 提供非交互模式，使用 `codex exec` 可以从脚本或持续集成任务中运行 Codex。官方说明中，非交互运行可以将最终回答写到标准输出，并通过明确的沙箱和审批参数限制权限。相关说明见 OpenAI Codex 非交互模式文档。

本章只完成一次最小练习：

> **只读审查团队任务看板，将最终结果保存到项目外部，并检查命令的退出状态。**

我们暂不展开 JSONL 事件流、结构化输出、批量脚本、会话恢复和持续集成认证。

二、主体

33.1 非交互模式和普通 CLI 聊天有什么区别

在第 26 章中，你从项目目录启动 Codex CLI，然后在会话中继续输入任务、观察过程并检查结果。

非交互模式更像一个一次性命令：

```
准备输入
→ 启动 codex exec
→ Codex 完成任务
→ 输出最终结果
→ 命令结束
```

两种方式的差异可以概括为：

项目	交互式 CLI	codex exec
是否持续显示会话	是	通常一次运行后结束
能否中途继续追问	可以	需要重新运行或恢复线程
是否适合脚本调用	一般	适合
输出是否便于重定向	需要整理	最终回答可直接输出
权限是否应预先固定	可以运行中处理	必须提前设计
失败后怎样处理	人立即判断	脚本根据状态停止或继续

因此，非交互模式适合已经具备以下特征的任务：

- 输入明确；
- 范围稳定；
- 权限可以提前确定；
- 输出格式足够清楚；
- 失败时有明确停止条件；
- 不需要在执行过程中临时讨论业务取舍。

如果任务仍然需要频繁澄清，就不应该急着脚本化。

33.2 为什么本章只做只读任务

第 32 章已经建立了一条只读项目检查流程，但它只完成了定时试运行，尚未进入长期无人值守状态。

本章继续采用只读任务，原因有三点：

219. 它可以帮助你理解输入、输出和退出状态，而不被文件修改干扰；
220. 非交互运行中没有人随时处理审批，因此权限越窄越容易判断；
221. 如果连只读脚本都不能稳定完成，就不应继续增加自动修改、推送或发布能力。

本章要求 Codex：

- 读取当前仓库；
- 检查 Git 状态和主要文件；
- 输出最多三项需要关注的内容；
- 说明没有执行哪些验证；

- 不修改、创建、移动或删除仓库文件；
- 不联网；
- 不调用写入型 MCP 工具；
- 不提交、不推送、不创建 PR。

33.3 运行前确认项目状态

进入团队任务看板：

```
Set-Location "$HOME\Documents\Codex 学习\projects\team-task-board"
```

检查当前目录和 Git 状态：

```
Get-Location
git branch --show-current
git status --short
git log --oneline -1
```

预期：

- 当前目录是 `team-task-board`；
- 当前分支是 `main`；
- `git status --short` 没有输出；
- 最近提交与第 32 章结束状态一致。

若工作区存在未提交变化，应停止。

不要让非交互脚本自动判断这些变化是否“可以忽略”，因为它无法知道这些内容是否属于用户正在进行的工作。

33.4 将输出放在项目外部

如果把运行结果直接写进仓库，会出现两个问题：

- 一次只读审查却制造了新的 Git 变化；
- 后续运行可能不断产生时间戳文件，污染项目历史。

因此，本章将输出放在：

```
Codex 学习/
├── projects/
│   └── team-task-board/
└── run-results/
```

在 PowerShell 中创建目录：

```
$resultsDirectory = "$HOME\Documents\Codex 学习\run-results"
New-Item -ItemType Directory -Force -Path $resultsDirectory | Out-Null
```

```
$outputPath = Join-Path $resultsDirectory "team-task-board-readonly-review.md"
```

这里创建的是项目外部的学习结果，不属于团队任务看板的交付文件。

33.5 写出能够独立运行的输入

非交互任务不能依赖聊天里没有保存的上下文。

提示需要包含：

- 当前对象；
- 任务目标；
- 检查范围；
- 权限边界；
- 输出要求；
- 停止条件。

可以在 PowerShell 中建立一个多行字符串：

```
$prompt = @'
你正在检查当前团队任务看板仓库。

只执行只读审查，不要修改、创建、移动或删除任何文件。
不要联网，不要调用写入型 MCP 工具，不要创建分支、提交、推送或 PR。

请完成：
1. 确认当前分支和 Git 工作区状态；
2. 概括项目入口、主要 JavaScript 模块和本地存储边界；
3. 检查 README、AGENTS.md、docs/current-state.md 与代码现状是否存在明显不一致；
4. 最多列出 3 项需要关注的问题，并按“阻断、重要、建议”分类；
5. 明确说明本次没有执行浏览器回归、网络检查和外部写入。

如果工作区不干净、项目路径不正确或无法读取关键文件，请停止并说明原因。
最终只输出一份简洁 Markdown 报告。
'@
```

这段提示没有要求 Codex“尽量完成”。

它明确规定：

> 起始状态不可信时，停止比继续猜测更正确。

33.6 运行第一次 `codex exec`

执行：

```
codex exec `
  --ephemeral `
  --sandbox read-only `
  --ask-for-approval never `
  --output-last-message $outputPath `
```

```
  $prompt

$exitCode = $LASTEXITCODE
```

参数含义：

参数	作用
exec	以非交互方式执行一次任务
--ephemeral	不保留本次会话的持久运行记录
--sandbox read-only	只允许读取，不允许修改工作区
--ask-for-approval never	本次运行不弹出审批，无法完成时直接失败
--output-last-message	将最终回答额外写入指定文件

OpenAI 当前说明，`codex exec` 默认使用只读沙箱；本章仍然显式写出 `read-only`，目的是让脚本的权限边界不依赖隐含默认值。运行过程信息通常进入错误输出，最终回答进入标准输出，因此最终回答可以被管道或文件接收。

33.7 检查退出状态

运行结束后输出：

```
"Codex exit code: $exitCode"
```

在常见命令行约定中：

- `0` 通常表示命令按预期结束；
- 非 `0` 表示运行失败、参数错误、权限不足或其他异常。

但退出状态只能说明“命令是否成功结束”，不能证明内容一定正确。

因此要同时检查三类证据：

```
Test-Path $outputPath
Get-Item $outputPath | Select-Object FullName, Length, LastWriteTime
Get-Content $outputPath
```

至少确认：

- 输出文件存在；
- 文件不是空文件；
- 内容确实针对当前项目；
- 报告包含实际检查范围；
- 报告说明了未执行事项；
- 没有声称完成浏览器验证或外部检查。

33.8 失败时不要用旧结果冒充新结果

假设上一次运行已经产生：

```
team-task-board-readonly-review.md
```

如果本次运行失败，这个文件可能仍然保留旧内容。

因此更稳妥的做法是：

```
if (Test-Path $outputPath) {
    Remove-Item $outputPath
}

codex exec `
  --ephemeral `
  --sandbox read-only `
  --ask-for-approval never `
  --output-last-message $outputPath `
  $prompt

$exitCode = $LASTEXITCODE

if ($exitCode -ne 0) {
    throw "Codex 只读审查失败，退出状态：$exitCode"
}

if (-not (Test-Path $outputPath)) {
    throw "Codex 运行结束，但没有生成预期输出文件。"
}
```

这里先删除旧文件，目的是避免：

> 新任务失败后，下游程序误把旧报告当成本次成果。

这一原则不仅适用于 Codex，也适用于任何自动化脚本。

33.9 检查仓库没有被修改

回到项目检查：

```
git status --short
```

预期没有输出。

还可以检查：

```
git branch --show-current
git log --oneline -1
```

确认：

- 仍在 `main`；
- 没有新增提交；

- 没有新分支；
- 项目内没有生成报告；
- 项目外只有预期的运行结果。

如果仓库出现变化，即使报告内容看起来正确，本章也不通过。

33.10 标准输出、错误输出和结果文件

本章涉及三种不同对象：

对象	主要内容	用途
标准输出	最终回答	管道传递或屏幕查看
错误输出	运行进度、诊断或错误	排查执行过程
结果文件	`--output-last-message` 保存的最终回答	后续读取和归档

不要把错误输出理解为“全部都是失败”。

某些工具会把进度信息写到错误输出，以便标准输出保持干净，能够继续被其他程序使用。

同样，出现一个结果文件也不代表任务成功。

最终判断应同时使用：

```
退出状态
+ 输出文件
+ 输出内容
+ 仓库状态
```

33.11 为什么本章不使用 JSONL 和 Schema

官方非交互模式还支持：

- 使用`--json`输出 JSON Lines 事件流；
- 使用`--output-schema`要求最终结果符合 JSON Schema；
- 恢复已有非交互线程；
- 将输入通过管道传入；
- 在持续集成环境中运行。

这些能力适合程序继续解析：

- 每次工具调用；
- 线程和回合状态；
- 结构化字段；
- 失败事件；
- Token 使用情况。

但本章的目标只是理解：

> 一次输入、一次执行、一个最终输出和一个可检查的结束状态。

过早加入 Schema、JSONL 和复杂批量器，会把注意力从基本可靠性转移到程序结构。

本书只讲到当前示例所需的最小非交互用法。结构化批处理和复杂批量脚本不在本书范围内，可根据实际项目需要继续参考 Codex 官方文档。

33.12 什么时候不应该使用 `codex exec`

以下任务更适合交互式会话：

- 需求仍然模糊；
- 需要多次业务澄清；
- 可能出现多个合理方案；
- 修改范围会根据中间结果变化；
- 需要人逐步批准外部操作；
- 失败后必须立即讨论取舍；
- 首次执行高风险发布或数据迁移。

不要把“能够脚本化”误解为“应该脚本化”。

自动化真正节省时间的前提是：

> 人已经知道这项工作应该怎样开始、怎样结束，以及何时必须停止。

三、总结

本章完成了第一次最小非交互任务：

- 理解交互式 CLI 与 `codex exec` 的区别；
- 选择只读项目审查作为练习；
- 在运行前确认仓库干净；
- 将输出目录放到项目外部；
- 写出可以独立运行的持久提示；
- 显式设置只读沙箱和无审批模式；
- 保存最终回答；
- 检查 Shell 退出状态；
- 防止旧输出冒充新结果；
- 确认团队任务看板没有发生变化。

本章掌握检查

- [ ] 能说明 `codex exec` 适合什么任务；
- [ ] 非交互提示不依赖聊天中的隐含上下文；
- [ ] 权限被显式限制为只读；
- [ ] 运行失败时不会继续使用旧输出；
- [ ] 同时检查退出状态、结果内容和 Git 状态；
- [ ] 结果文件位于项目外部；
- [ ] 团队任务看板仍在干净的 `main` 分支。

四、结束语与引导

`codex exec` 让 Codex 能够进入脚本，但它仍然是一项一次性任务。

下一章继续向前一步：

> **不只运行一个命令，而是在自己的程序中创建 Codex 线程，并让第二轮任务继承第一轮上下文。**

这就是 Codex SDK 的作用。

第 34 章属于选修导览。它不会把本书变成 TypeScript 教程，也不会创建新的业务案例。

第 34 章：使用 Codex SDK 构建自己的工作流

一、起始语

第 33 章中，你使用一条 `codex exec` 命令完成了只读审查。

这种方式非常适合：

- 一次输入；
- 一次执行；
- 一个最终输出；
- 一个明确结束状态。

但有些工作流需要程序自己掌握更多过程。

例如：

- 第一轮先读取项目状态；
- 第二轮根据第一轮结果继续追问；
- 保存线程标识，之后恢复；
- 将 Codex 嵌入内部工具；
- 根据返回结果决定下一步程序动作；
- 让应用程序管理多个 Codex 任务。

这时，单独拼接命令会逐渐变得笨重。

Codex SDK 提供了程序化控制方式。OpenAI 当前提供 TypeScript 库，也提供 Python 库；本章选择 TypeScript 做一个最小两轮示例。官方 TypeScript 库要求 Node.js 18 或更高版本，可以创建线程、连续运行多个回合，并指定工作目录。本章示例中的线程选项同时参考 OpenAI Codex SDK 官方仓库说明。相关概念见 OpenAI Codex SDK 文档。

本章是**选修导览**。

唯一目标是理解四个对象：

> **Codex 客户端、Thread、Turn 和连续上下文。**

我们不会构建完整应用，也不展开流式事件、复杂类型、生产认证和多智能体编排。

二、主体

34.1 SDK 解决的是什么问题

SDK 不是“更高级的聊天界面”。

它的价值在于：

> 让你自己的程序能够创建、继续和管理 Codex 任务。

可以将三种使用方式放在一起比较：

方式	适合场景	主要控制者
Codex 交互界面	人持续参与、随时调整	人
`codex exec`	一次性脚本任务	Shell 脚本
Codex SDK	应用内多轮任务和程序化工作流	你的程序

使用 SDK 后，程序可以：

- 创建一个新线程；
- 向线程发送第一轮任务；
- 读取最终回答；
- 在同一个线程中继续第二轮；
- 保存线程 ID；
- 在另一次程序运行中恢复线程；
- 处理结构化结果或流式事件。

但 SDK 也会增加新的责任：

- 依赖安装；
- 运行环境；
- 错误处理；
- 身份认证；
- 权限配置；
- 日志和结果保存；
- 版本兼容；
- 程序生命周期。

因此，能用一条 `codex exec` 完成的任务，不必为了“显得高级”改成 SDK。

34.2 本章仍然使用团队任务看板

为了避免形成第五条案例线，本章不创建新的业务项目。

项目仍然是：

```
Codex 学习/projects/team-task-board
```

但 SDK 安装和示例代码放在独立实验目录：

```
Codex 学习/
├── projects/
│   └── team-task-board/
└── labs/
    └── codex-sdk-readonly/
```

`codex-sdk-readonly` 只是调用外壳。

它不会：

- 成为新的产品案例；
- 保存团队任务数据；
- 修改团队任务看板；
- 进入团队任务看板 Git 历史；
- 被第 35 章继续开发。

SDK 线程会通过明确的 `workingDirectory` 指向现有仓库。

34.3 判断自己是否适合完成本章

开始前确认：

```
node --version
npm --version
codex --version
```

Node.js 版本应满足 SDK 当前要求。还要确认 Codex CLI 已经完成可用的登录或认证；不要把 API 密钥直接写进示例代码。

若电脑没有 Node.js，或你还不能独立理解下面这些概念：

- 安装依赖；
- 运行 JavaScript 文件；
- 环境变量；
- 异步等待；
- 错误退出；

可以先阅读本章，不必强行操作。

这不会影响第 35—40 章的学习。

34.4 建立临时实验目录

在 `Codex 学习` 目录下执行：

```
Set-Location "$HOME\Documents\Codex 学习"
New-Item -ItemType Directory -Force -Path ".\labs\codex-sdk-readonly" | Out-Null
Set-Location ".\labs\codex-sdk-readonly"
```

初始化最小 Node.js 目录：

```
npm init -y
npm install @openai/codex-sdk
```

安装后会出现：

```
codex-sdk-readonly/
├── node_modules/
├── package-lock.json
└── package.json
```

这些文件属于实验目录，不属于团队任务看板。

34.5 为目标项目设置明确路径

在 PowerShell 中解析项目路径：

```
$projectPath = Resolve-Path "..\..\projects\team-task-board"
$env:TEAM_TASK_BOARD_PATH = $projectPath.Path
```

检查：

```
$env:TEAM_TASK_BOARD_PATH
Test-Path $env:TEAM_TASK_BOARD_PATH
```

不要把路径直接写死成其他人的用户名。

环境变量的作用是让示例程序知道：

> 本次读取的目标仓库究竟在哪里。

34.6 创建最小两轮程序

新建 `review.mjs`：

```
import { Codex } from "@openai/codex-sdk";

const projectPath = process.env.TEAM_TASK_BOARD_PATH;

if (!projectPath) {
```

```
  throw new Error("缺少 TEAM_TASK_BOARD_PATH 环境变量。");
}

const codex = new Codex();

const thread = codex.startThread({
  workingDirectory: projectPath,
  sandboxMode: "read-only",
});

const firstTurn = await thread.run(`
只读检查当前团队任务看板仓库。
不要修改、创建、移动或删除文件，不要联网。

请输出：
1. 当前分支和工作区状态；
2. 项目入口和主要 JavaScript 模块；
3. 当前最重要的数据不变量；
4. 本次未执行的验证。
`);

console.log("\n=== 第一轮：项目状态 ===\n");
console.log(firstTurn.finalResponse);

const secondTurn = await thread.run(`
基于刚才已经读取的项目状态，输出 3 项维护建议。

要求：
- 只提出建议，不修改文件；
- 每项说明依据；
- 区分“近期需要”和“可以暂缓”；
- 不重复完整项目介绍。
`);

console.log("\n=== 第二轮：维护建议 ===\n");
console.log(secondTurn.finalResponse);
```

这段程序只做了四件事：

222. 创建 `Codex` 客户端；
223. 创建一个指向团队任务看板的只读线程；
224. 运行第一轮项目检查；
225. 在同一线程中运行第二轮维护建议。

34.7 理解 Thread 和 Turn

Thread

Thread 可以理解为一条持续的 Codex 工作上下文。

它包含：

- 当前工作目录；
- 已经发生的多轮任务；
- 前面读取过的信息；

- 线程标识；
- 后续继续所需的上下文。

Turn

Turn 是线程中的一次具体运行。

本章有两个 Turn：

```
Thread
├── Turn 1：读取项目状态
└── Turn 2：基于前一轮输出维护建议
```

第二轮不需要重新把第一轮项目摘要全部复制进提示，因为它继续使用同一线程。

这就是 SDK 与两条互相独立的命令之间的重要差异。

34.8 为什么第一轮和第二轮必须分工

如果第一轮就要求：

> 读取项目、分析风险、制定计划、修改代码、运行验证、提交并推送。

即使 SDK 可以运行，这也不是一个适合零基础读者的最小示例。

本章刻意将任务分成：

```
第一轮：建立事实
第二轮：基于事实形成建议
```

这样可以观察：

- 第二轮是否真的继承了前文；
- 它是否引用了项目中的真实模块；
- 它是否避免重复读取说明；
- 它是否保持只读边界。

多轮并不代表每一轮都要增加权限。

34.9 运行程序

在实验目录中执行：

```
node .\review.mjs
```

预期看到两个分区：

```
=== 第一轮：项目状态 ===
...
```

```
=== 第二轮：维护建议 ===
...
```

检查第一轮是否包含：

- 当前分支；
- 干净工作区；
- `index.html` 和主要 JavaScript 模块；
- 本地存储边界；
- 未执行浏览器验证。

检查第二轮是否：

- 基于第一轮内容；
- 不重新编造项目结构；
- 只给三项建议；
- 说明依据；
- 没有修改文件。

34.10 给程序增加最小错误处理

最小示例还应避免运行失败后继续输出“成功”。

可以改为：

```
import { Codex } from "@openai/codex-sdk";

async function main() {
  const projectPath = process.env.TEAM_TASK_BOARD_PATH;

  if (!projectPath) {
    throw new Error("缺少 TEAM_TASK_BOARD_PATH 环境变量。");
  }

  const codex = new Codex();
  const thread = codex.startThread({
    workingDirectory: projectPath,
    sandboxMode: "read-only",
  });

  const firstTurn = await thread.run(
    "只读概括当前项目状态和主要模块，不要修改文件。"
  );

  if (!firstTurn.finalResponse) {
    throw new Error("第一轮没有返回最终回答。");
  }

  const secondTurn = await thread.run(
    "基于上一轮结果，给出 3 项有依据的维护建议，不要修改文件。"
  );
```

```
  if (!secondTurn.finalResponse) {
    throw new Error("第二轮没有返回最终回答。");
  }

  console.log(firstTurn.finalResponse);
  console.log(secondTurn.finalResponse);
}

main().catch((error) => {
  console.error(error);
  process.exitCode = 1;
});
```

这仍然不是完整生产程序，但已经形成最低限度的失败边界。

34.11 检查团队任务看板没有变化

运行后进入项目：

```
Set-Location $env:TEAM_TASK_BOARD_PATH
git status --short
git branch --show-current
git log --oneline -1
```

预期：

- `git status --short` 没有输出；
- 当前仍为 `main`；
- 没有新提交；
- 没有生成 SDK 文件；
- 没有修改 `package.json`；
- 没有将实验依赖装进项目。

如果项目出现变化，应停止并调查线程权限和工作目录。

34.12 什么时候应该使用 SDK

SDK 适合：

- 内部开发工具；
- 自定义任务面板；
- 多轮工程助手；
- 持续集成中的复杂控制；
- 需要保存和恢复线程的工作流；
- 应用需要读取结果并决定下一步；
- 需要结构化输出或流式事件。

SDK 不适合仅仅因为：

- 想把一条命令写得更长；
- 想展示更多代码；
- 一个任务只运行一次；
- 人仍然需要每一步参与；
- 团队没有维护 Node.js 或 Python 程序的能力。

选择 SDK 前应先问：

> 这项任务是否真的需要由程序管理连续上下文和生命周期？

如果答案是否定的，CLI 或 `codex exec` 通常更简单。

34.13 本章没有展开的能力

官方 SDK 还可以涉及：

- 恢复已有线程；
- 流式接收事件；
- 结构化输出；
- 图像输入；
- 自定义环境变量；
- Python 同步和异步客户端；
- 对不同回合使用不同沙箱；
- 与更大应用集成。

这些能力需要更多代码、测试和安全设计。

本书正文不继续扩展完整 SDK 控制器。更复杂的线程管理、沙箱编排和应用集成不在本书范围内，可根据实际项目需要继续参考 Codex SDK 官方文档。

三、总结

本章通过一个两轮只读示例理解了 Codex SDK：

- SDK 让程序能够控制 Codex 任务；
- Thread 保存连续工作上下文；
- Turn 表示线程中的一次运行；
- 第一轮建立事实，第二轮基于事实形成建议；
- `workingDirectory` 将线程指向团队任务看板；
- `sandboxMode` 限制为只读；

- 实验依赖放在项目外部；
- 运行失败时返回非成功状态；
- 团队任务看板没有被修改。

本章掌握检查

- [ ] 能区分交互界面、`codex exec` 和 SDK；
- [ ] 能说明 Thread 和 Turn 的关系；
- [ ] 两轮任务使用同一线程；
- [ ] SDK 实验没有成为新的业务案例；
- [ ] 目标项目通过明确工作目录指定；
- [ ] 程序具备最小错误处理；
- [ ] 运行后团队任务看板工作区干净；
- [ ] 知道什么时候不需要 SDK。

四、结束语与引导

SDK 解决的是“怎样把 Codex 放进自己的程序”。

但程序能够运行，并不代表代码可以直接合并。

下一章回到 GitHub 协作：

> **用确定性 CI、Codex Review 和人工批准组成一条最小质量链。**

三者各自负责不同的判断，不能互相替代。

第 35 章：把 Codex 接入 GitHub、代码审查和持续集成

一、起始语

第 20 章中，你已经完成过一次完整 PR 流程。

第 28 章又使用 Codex 云端创建并合并了文档 PR。

因此，本章不重新教学：

- 怎样创建分支；
- 怎样提交；
- 怎样推送；
- 怎样填写 PR；
- 怎样合并和同步主分支。

本章解决的是另一个问题：

> **一个 PR 在合并前，应该经过哪些不同类型的检查？**

很多团队会把“测试通过”“AI 审查通过”和“有人批准”混成同一件事。

实际上，它们承担不同责任：

```
确定性 CI
→ 检查可以重复计算的规则

Codex Review
→ 检查差异中的高影响行为和仓库约束

人工批准
→ 判断业务意图、风险接受和是否允许合并
```

OpenAI 当前的 GitHub 代码审查功能可以读取 PR 差异、遵循仓库中的 `AGENTS.md` 规则，并以 GitHub 标准 Review 形式给出高信号问题。官方也明确指出，代码审查规则不能替代测试、分支保护和必要审批。相关说明见 Codex GitHub 代码审查文档。

本章使用一个很小的代表性 PR：

> **任务处于编辑状态时，按 `Escape` 取消编辑，并在手工检查文档中补充验证场景。**

它不改变数据结构、不更换本地存储键，也不引入第三方依赖。

二、主体

35.1 为什么选择一个小型交互改进

质量链的第一次练习不适合使用：

- 大规模重构；
- 数据迁移；
- 权限系统；
- 自动发布；
- 多仓库联动；
- 无法稳定复现的问题。

本章任务范围为：

- 只有在任务编辑状态下，`Escape` 才取消编辑；
- 取消后恢复新增任务表单；
- 不删除或覆盖现有任务；
- 不改变任务 `id`、`createdAt`、截止日期或状态；
- 不影响普通新增任务流程；
- 在手工验证文档中增加对应场景。

这个变化足够小，可以让注意力集中在质量链，而不是重新开发复杂功能。

35.2 先区分三种质量信号

确定性 CI

确定性检查对同一份输入应得到稳定结果。

例如：

- JavaScript 语法是否有效；
- 是否存在行尾空格或冲突标记；
- 必需文件是否存在；
- 固定测试是否通过；
- 构建是否成功。

它适合回答：

> 这份代码是否违反了明确、可重复执行的规则？

Codex Review

Codex Review 更适合检查：

- 修改是否违反项目数据不变量；
- 是否遗漏高影响边界；
- 是否出现回归风险；
- 测试是否覆盖关键行为；
- 文档与实现是否不一致。

它适合回答：

> 这份差异中是否存在机器规则难以穷举，但会造成真实后果的问题？

人工批准

人工审查者负责：

- 需求是否真的应该这样实现；
- 交互是否符合用户预期；
- 风险是否可以接受；
- 是否需要补充业务验收；
- 当前是否适合合并和发布。

它适合回答：

> 团队是否愿意对这次变化负责？

三者缺一不可。

35.3 确认 GitHub 代码审查前置条件

按照 OpenAI 当前说明，使用 GitHub 中的 Codex Review 前，应确认：

- 当前仓库已经连接 Codex cloud；
- 仓库的 Code review 功能已经启用；
- PR 位于已连接的仓库；
- 如需仓库专属规则，存在适用的 `AGENTS.md`；
- 当前账号有权在 PR 评论中请求审查。

本章使用手动触发：

```
@codex review
```

暂不启用所有 PR 自动审查。

原因是第一次建立规则时，需要先观察：

- 它是否理解项目边界；
- 规则是否产生噪音；
- 是否遗漏真正重要的问题；
- 团队是否能正确处理结果。

35.4 在 AGENTS.md 中加入少量审查规则

第 21 章已经建立根目录 `AGENTS.md`。

本章不重写整份文件，只增加一个简洁部分：

```
## Code Review Rules

### Local data compatibility

- Do not change the localStorage key or silently discard existing task fields.
  Safe path: preserve unknown optional fields and stop on parse or write failure.

### Task identity

- Editing or cancelling an edit must preserve task id and createdAt.
  Safe path: update only explicitly editable fields.

### Interaction regression

- Keyboard shortcuts must not change create-mode behavior when no task is being edited.
  Safe path: gate Escape handling on an active edit state and document the manual check.
```

规则使用英文，是为了与官方示例和跨团队代码审查习惯保持一致；正文解释仍然使用中文。

这些规则有三个特点：

- 指向真实后果；
- 给出安全路径；
- 不依赖某个可能很快变化的函数名。

不要把以下机械规则塞进 Codex Review：

- 缩进必须两个空格；
- 文件末尾必须换行；
- JavaScript 必须通过语法检查；
- 禁止行尾空格。

这些更适合 CI。

35.5 建立最小确定性 CI

团队任务看板没有复杂构建系统，本章只建立最小检查：

226. 检查 Git 差异格式；

227. 检查所有 JavaScript 文件语法；

228. 确认核心入口文件存在。

可以创建：

```
.github/workflows/basic-checks.yml
```

正文只展示结构骨架：

```
name: Basic checks

on:
  pull_request:

jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
        with:
          fetch-depth: 0

      - uses: actions/setup-node@v6
        with:
          node-version: 24

      - name: Check required files
        run: test -f index.html && test -f js/app.js

      - name: Check whitespace errors
        run: git diff --check "${{ github.event.pull_request.base.sha }}...${{ github.event.pull_request.head.sha }}"

      - name: Check JavaScript syntax
        run: |
          for file in js/*.js; do
            node --check "$file"
          done
```

这不是完整企业 CI 模板。示例中的 Action 主版本以 2026 年 7 月 31 日可用版本为准，实际操作时应先核对 GitHub 官方仓库的当前稳定版本。

它只建立一个最小认识：

> 能够用固定命令判断的事情，先交给固定命令。

完整权限设置、缓存、矩阵测试和生产部署流程不在本书范围内。本章只建立最小质量链，实际项目应继续遵循所在团队的 CI 和发布规范。

35.6 准备代表性 PR

按照前面已经掌握的 Git 流程，建立一个聚焦分支，例如：

```
quality/escape-cancel-edit
```

本章预计修改：

```
AGENTS.md
js/app.js
docs/manual-test-checklist.md
.github/workflows/basic-checks.yml
```

实现边界：

- 监听键盘事件；
- 只有存在当前编辑任务时才响应 `Escape`；
- 调用已有取消编辑或表单恢复逻辑；
- 不另写一套状态清理代码；
- 增加手工验证步骤；
- 不顺带重构其他事件处理。

完成本地检查后，创建一个聚焦提交和 PR。

PR 标题可以是：

```
支持按 Escape 取消任务编辑
```

PR 说明至少包含：

```
## 变更

- 编辑任务时按 Escape 取消编辑
- 取消后恢复新增任务表单
- 增加对应手工回归场景
- 增加最小 PR 检查

## 不包含

- 不改变任务数据结构
- 不改变 localStorage 键
- 不新增依赖
- 不自动发布

## 验证

- JavaScript 语法检查
- 普通新增任务回归
- 进入编辑后按 Escape
- 未进入编辑时按 Escape 无业务变化
- 刷新后已有任务保持不变
```

35.7 先等待确定性 CI

PR 创建后，先观察 `Basic checks`。

如果 CI 失败：

- 打开失败步骤；

- 确认是语法、差异还是文件问题；
- 在本地复现；
- 修复后重新推送；
- 不急着请求 Codex Review。

原因是：

> 模型审查不应浪费在一个连基础语法都没有通过的差异上。

同样，如果 CI 配置本身写错，应先修复质量链，不要临时将失败检查设为可忽略。

35.8 请求 Codex Review

确定性检查通过后，在 PR 评论中输入：

```
@codex review
```

也可以增加一次性重点：

```
@codex review for local data compatibility and interaction regressions
```

Codex 会读取：

- PR 差异；
- 适用的 `AGENTS.md`；
- 相关文件上下文；
- 仓库中的审查规则。

官方当前说明，GitHub 中的 Codex Review 聚焦高优先级问题，目的是减少低价值噪音。

因此，没有评论不等于：

- 所有测试都完成；
- 交互一定正确；
- 业务已经验收；
- 可以自动合并。

它只表示：

> 在本次审查范围和当前规则下，没有发现需要报告的高信号问题。

35.9 怎样处理审查发现

如果 Codex 指出问题，不要直接回复：

> 帮我全部修复。

先完成三步：

229. 打开它引用的代码；
230. 复现或验证问题；
231. 判断它是否属于当前 PR 范围。

对于真实问题：

- 在当前分支修复；
- 补充确定性测试或手工验证；
- 更新 PR 说明；
- 再次等待 CI；
- 必要时重新请求 Review。

对于误报：

- 说明当前行为为什么安全；
- 检查 `AGENTS.md` 规则是否过宽；
- 必要时收窄规则；
- 不为了让 AI“满意”而制造无意义修改。

35.10 人工审查仍然要做什么

人工审查者至少检查：

差异范围

- 是否只修改预期文件；
- 是否混入格式化或无关重构；
- CI 文件是否获得过度权限。

真实交互

- 打开已有任务编辑；
- 修改字段后按 `Escape`；
- 表单是否恢复新增模式；
- 原任务是否保持不变；
- 未编辑时按 `Escape` 是否没有业务副作用；
- 再次编辑和保存是否正常。

数据边界

- `id` 是否保持；
- `createdAt` 是否保持；
- 截止日期和状态是否没有被清空；
- 刷新后本地数据是否正常。

合并判断

- 需求是否完成；
- 未验证事项是否明确；
- 是否需要第二名审批者；
- 当前是否适合进入主分支。

35.11 设置最小合并规则

对于练习仓库，可以设置最小规则：

- PR 必须通过 `Basic checks`；
- 至少一名人工审查者批准；
- 存在未解决 Review 评论时不得合并；
- 禁止直接推送到 `main`；
- Codex Review 作为额外信号，不作为唯一批准者；
- 不启用自动修复和自动合并。

这里的关键不是规则数量，而是责任清楚：

```
CI 通过
≠ Codex 审查通过
≠ 人工批准
≠ 发布授权
```

35.12 为什么正文不配置 Codex GitHub Action

OpenAI 还提供 `openai/codex-action@v1`，可以从 GitHub 事件触发 `codex exec`，保存输出或在工作流中执行 Codex 任务。

但正式接入需要处理：

- API 密钥；
- GitHub Secrets；
- 工作流权限；
- Runner 隔离；
- Prompt 文件；
- 输出处理；
- 外部评论权限；
- 失败和重试策略；
- 版本固定。

这些内容超出零基础正文边界。

本章使用 GitHub 原生 Codex Review 建立质量链，不再提供完整 Action 模板。需要自定义 GitHub Actions 时，应根据实际仓库和官方文档单独设计、审查和验证。

35.13 合并并恢复唯一主线

当以下条件都满足时再合并：

- 本地检查通过；
- GitHub 确定性 CI 通过；
- Codex Review 结果已处理；
- 人工交互验证通过；
- 至少一名人工审查者批准；
- PR 差异只包含预期内容。

合并后同步本地：

```
git switch main
git pull --ff-only
git status --short
git log --oneline -3
```

预期：

- 本地 `main` 包含本章 PR；
- 工作区干净；
- 临时分支按团队规则删除；
- 没有遗留 Worktree；
- 不存在未处理的发布任务。

三、总结

本章建立了团队任务看板的最小 GitHub 质量链：

- 选择一个小型、可回归的交互改进；
- 在 `AGENTS.md` 加入少量仓库审查规则；
- 使用确定性 CI 检查语法、差异和必需文件；
- 在 CI 通过后请求`@codex review`；
- 对发现进行人工复核；
- 完成真实浏览器和数据回归；
- 由人工审查者决定是否合并；
- 合并后恢复干净主分支。

本章掌握检查

- [ ] 能区分确定性 CI、Codex Review 和人工批准；
- [ ] 机械规则没有被写进模型审查提示；
- [ ] `AGENTS.md` 规则指向真实后果和安全路径；
- [ ] CI 失败时先修复基础问题；
- [ ] Codex 无发现不等于业务验收完成；
- [ ] 模型发现经过人工复现和判断；
- [ ] 合并需要人工授权；
- [ ] 本地与远程 `main` 最终一致且干净。

四、结束语与引导

到这里，团队任务看板已经具备：

- 项目规则；
- 项目知识和交接；
- 长任务计划；
- 浏览器验证；
- 多入口协作；
- Skill、MCP、子智能体和定时试运行；
- 非交互脚本和 SDK 导览；
- GitHub 质量链。

能力越多，越不能只依靠“使用者小心一点”。

下一章将把前面分散出现的安全原则收束为一套普通团队可以执行的规则：

> **什么可以自动做，什么必须审批，密钥放在哪里，谁有权发布，异常发生时怎样停止。**

第 36 章：沙箱、审批、密钥、发布和团队使用规范

一、起始语

本书前 35 章不断增加 Codex 能够完成的事情：

- 读取和修改本地项目；
- 运行命令；
- 创建分支和 PR；
- 使用云端环境；
- 连接 MCP；
- 调用 Skill；
- 拆分子智能体；
- 安排定时任务；
- 使用非交互模式和 SDK；
- 进入 GitHub 质量链。

如果只看效率，这些能力越多越好。

如果考虑真实团队使用，就必须同时回答：

- Codex 可以访问哪些文件？
- 哪些命令可以自动运行？
- 什么时候必须停下来询问人？
- 外部系统是否允许写入？
- 密钥从哪里进入运行环境？
- 谁可以推送、合并和发布？
- 发生异常时怎样停止和恢复？

OpenAI 当前将沙箱和审批区分为两个控制层：沙箱决定命令能够访问哪些文件和网络资源，审批决定 Codex 在什么动作前必须暂停。官方建议从最小权限开始，并优先保持项目边界。相关说明见 OpenAI Agent 审批与安全文档和 Codex 沙箱说明。

本章不建立复杂企业治理体系。

我们只把前面已经出现的原则收束为：

> **五条团队安全规则＋一张发布检查表。**

它们适用于普通项目、小型团队和本书练习环境。

二、主体

36.1 先区分四种容易混淆的控制

工作区边界

工作区说明 Codex 当前被授权处理哪个项目或目录。

例如：

```
Codex 学习/projects/team-task-board
```

它不应自动扩展到：

- 整个 `Documents` 目录；
- 其他客户项目；
- 下载目录；
- 浏览器资料；
- 用户主目录；
- 网络共享盘。

沙箱

沙箱决定命令可以：

- 只读文件；
- 在工作区内写入；
- 是否访问工作区外路径；
- 是否访问网络；
- 是否拥有更宽系统权限。

审批

审批决定某个动作是否需要人在执行前确认。

例如：

- 请求网络访问；
- 修改工作区外文件；
- 执行不受信任命令；

- 调用有副作用的 MCP 工具；
- 进行外部写入。

发布授权

发布授权是业务和组织决定。

即使：

- 沙箱允许写入；
- 命令执行成功；
- CI 通过；
- Codex Review 无高优先级发现；

也不能自动推导出：

> 可以合并、上线或向客户发布。

发布授权必须由明确负责人做出。

36.2 三种常见权限范围

为了便于理解，可以先记住三类范围：

范围	适合任务	主要风险
read-only	阅读、解释、审查、计划	无法完成需要写入的任务
workspace-write	在当前项目内修改和验证	可能误改项目内文件
danger-full-access	已隔离环境中的特殊任务	可访问更广文件和网络，风险最高

OpenAI 当前文档建议，非交互只读任务可以使用只读沙箱和不审批模式；需要项目内修改时使用工作区写入，并对越界和网络访问保持审批。全访问只适合已经隔离、能够承受风险的环境。

本书的默认顺序是：

```
先只读
→ 明确修改范围
→ 再开放工作区写入
→ 高风险能力单独评估
```

不要因为一次任务被阻止，就直接切换到全访问。

阻止信息通常是在告诉你：

- 工作目录可能不对；
- 任务需要重新拆分；
- 权限设计不完整；

- 环境缺少必要依赖；
- 当前动作不适合自动执行。

36.3 第一条规则：只在可信工作区执行明确任务

团队规则可以写为：

> **每次任务必须绑定明确项目、明确范围和明确起始状态；工作区不可信时停止。**

执行前至少确认：

- 当前路径；
- 当前仓库；
- 当前分支；
- Git 工作区是否干净；
- 是否存在真实客户数据；
- 是否存在疑似密钥；
- 哪些文件属于任务范围；
- 哪些目录明确禁止访问。

对于陌生项目，先只读探索。

对于未纳入 Git 的目录，更应谨慎，因为缺少可靠差异和恢复点。

以下情况应停止：

- 打开了同名项目副本；
- 工作区包含来源不明的未提交变化；
- 项目混入个人文件；
- 发现真实生产数据；
- README 与实际项目完全不一致；
- 任务要求访问多个无关仓库但没有授权说明。

36.4 第二条规则：按任务选择最低权限

团队规则可以写为：

> **只授予完成当前任务所必需的文件、命令和网络权限。**

可以使用一张简单表格：

任务	建议范围
解释代码、建立项目地图	`read-only`
审查 PR 差异	`read-only`

任务	建议范围
修改当前项目文件	workspace-write
安装依赖	工作区写入＋必要审批
查询公开官方文档	只读＋受控网络或只读 MCP
向外部系统写入	单独审批
发布或部署	人工授权＋隔离流程

最低权限不等于永远不能扩大权限。

更准确的过程是：

```
先以最小范围尝试
→ 识别真实阻断
→ 说明为什么需要扩大
→ 人确认新的范围
→ 只扩大到完成任务所需程度
```

如果任务需要修改一个仓库，不应授权整个用户目录。

如果任务只需要读取官方文档，不应同时开放任意网络和写入型 MCP。

36.5 第三条规则：所有副作用都要有明确审批

团队规则可以写为：

> **凡是会改变外部状态、影响他人或难以撤销的动作，必须由人明确批准。**

常见副作用包括：

- 删除或覆盖文件；
- 批量重命名；
- 修改数据库；
- 发送邮件或消息；
- 更新工单；
- 在 PR 中发表评论；
- 推送分支；
- 合并 PR；
- 创建发布版本；
- 部署生产环境；
- 调用付费接口；
- 修改权限；
- 公开分享文件。

审批不能写成模糊的：

> 你看着办。

更合适的审批信息应包含：

- 将执行什么动作；
- 作用对象；
- 预计影响；
- 是否可恢复；
- 已完成哪些验证；
- 仍有哪些未知；
- 执行后怎样检查结果。

对于非交互和定时任务，如果无法获得审批，应选择：

- 不执行；
- 只输出建议；
- 创建候选文件但不提交；
- 将动作排入人工队列。

不要让无人值守任务通过扩大权限来绕过审批设计。

36.6 第四条规则：密钥不进入代码、提示和日志

团队规则可以写为：

> **密钥只通过受控秘密存储或运行环境注入，不写入 Git、不粘贴进提示、不输出到日志。**

密钥包括：

- API Key；
- Access Token；
- 数据库密码；
- 云平台凭据；
- 私钥；
- Webhook 密钥；
- OAuth 客户端秘密；
- 真实 Cookie 或会话令牌。

至少遵守：

232. 不在代码中硬编码；
233. 不提交`.env`和凭据文件；
234. `.gitignore`不能替代秘密管理；

235. 不要求 Codex“把当前环境变量全部打印出来”；
236. 不在错误日志中回显完整密钥；
237. 不把密钥写入 README、任务卡和截图；
238. 怀疑泄露时立即轮换，而不是只删除文件。

在 GitHub Actions 中，密钥应通过 GitHub Secrets 等受控方式提供，并只授予需要的工作流。即使使用 Secret，也要限制：

- 工作流触发条件；
- Runner 权限；
- 第三方 Action；
- 日志输出；
- Fork 来源；
- 可访问仓库范围。

本书不展开完整企业密钥体系，但必须建立一个底线：

> Codex 不需要知道的秘密，不应进入它的上下文。

36.7 第五条规则：发布必须人工授权，异常必须停止

团队规则可以写为：

> **模型可以准备候选成果，发布和不可逆操作由明确负责人授权；出现异常状态立即停止。**

发布前应回答：

- 发布对象是什么；
- 目标环境是什么；
- 谁是负责人；
- 哪个版本将被发布；
- CI 和测试是否通过；
- Codex Review 和人工审查是否完成；
- 业务验收是否完成；
- 回退方式是什么；
- 监控和确认方式是什么；
- 失败后由谁处理。

以下情况必须停止：

- 目标环境不明确；
- 分支或提交不明确；
- 工作区不干净；
- 测试结果过期；

- 密钥来源不可信；
- 依赖下载异常；
- 外部系统返回未知状态；
- 发布后无法确认结果；
- 回退方案不存在；
- 负责人没有明确授权。

停止不是任务失败的反义词。

在高风险任务中，能够安全停止本身就是正确结果。

36.8 建立一张团队任务分级表

团队可以按副作用和权限将任务分为四级：

等级	典型任务	默认控制
A｜只读	解释、搜索、审查、计划	只读沙箱，可无人值守
B｜项目内可恢复写入	修改代码、补测试、更新文档	工作区写入，保留 Git 差异
C｜外部写入	推送、PR 评论、工单更新、消息发送	明确审批，记录对象和结果
D｜发布或高影响操作	合并、部署、数据迁移、权限变更	人工授权、回退方案、双重确认

这张表不是法律或合规标准。

它的作用是帮助团队在任务开始前形成共同判断。

例如：

- 第 29 章 Skill 审查属于 A；
- 第 23 章任务编辑开发属于 B；
- 第 35 章请求 GitHub Review 和推送属于 C；
- 正式发布属于 D。

不要把整个聊天永久设置为最高等级。

每一项任务应单独选择范围。

36.9 团队任务看板发布检查表

在将团队任务看板交付给其他人前，至少检查：

代码与版本

- [ ] 当前版本来自明确提交；
- [ ] 本地与远程 `main` 一致；

- [ ] 工作区干净；
- [ ] 没有未合并临时分支或 Worktree；
- [ ] 没有调试文件、临时日志和实验依赖。

功能与数据

- [ ] 新增、搜索、筛选、状态更新和删除正常；
- [ ] 截止日期和逾期提示正常；
- [ ] 任务编辑和 `Escape` 取消正常；
- [ ] 旧数据能够继续读取；
- [ ] 解析或写入失败时不会静默覆盖数据。

质量证据

- [ ] 确定性 CI 通过；
- [ ] 关键浏览器回归完成；
- [ ] Codex Review 结果已处理；
- [ ] 人工审查批准；
- [ ] 未验证事项已记录。

权限与秘密

- [ ] 仓库不包含密钥；
- [ ] `.env` 和凭据文件未进入版本库；
- [ ] GitHub 工作流使用最小权限；
- [ ] MCP 连接只保留必要工具；
- [ ] 定时任务仍处于预期状态；
- [ ] 无人值守任务没有外部写入权限。

发布与回退

- [ ] 发布目标明确；
- [ ] 发布负责人明确；
- [ ] 版本号或提交号已记录；
- [ ] 发布后检查方法明确；
- [ ] 出现问题时可以回退到上一稳定版本；
- [ ] 业务方知道当前数据只保存在浏览器本地。

36.10 对本书已有能力逐项收口

Local、Worktree 和 Cloud

- Local 适合直接处理当前本地项目；

- Worktree 用于隔离并行任务；
- Cloud 用于远程仓库任务；
- 不能因为 Cloud 更方便就自动扩大 Secrets 和网络权限。

Skill

- Skill 只沉淀稳定、可复用的方法；
- Skill 不能隐藏危险动作；
- 带脚本的 Skill 仍需权限和审批；
- 触发描述应清楚，避免误调用。

MCP

- 优先只读连接；
- 写入工具单独审批；
- 外部返回内容视为不可信输入；
- 未找到资料时报告未找到，不得编造。

子智能体

- 适合拆分只读探索和审查；
- 主智能体负责冲突处理和最终复核；
- 多个智能体不能被当作多人独立批准。

定时任务

- 只安排已经人工验证过的提示；
- 无人值守时权限更窄；
- 失败应留下明确记录；
- 定期检查任务是否仍有价值；
- 本书的每周项目检查保持暂停，直到团队真正决定启用。

非交互和 SDK

- 先验证输入、输出和停止条件；
- 运行结果不能只看文件是否存在；
- 程序需要处理失败状态；
- 能用简单命令解决时不强行引入 SDK。

GitHub 质量链

- CI 负责确定性规则；
- Codex Review 负责高信号差异审查；
- 人工审查负责业务和风险判断；

- 发布授权仍由明确负责人承担。

36.11 常见错误做法

为了减少打断，永久开启全访问

这会让每个普通任务都继承不必要风险。

正确做法是按任务选择权限。

把审批理解为形式确认

如果用户不知道将发生什么，点击“允许”不构成有效审批。

把 Git 回退当作万能恢复

Git 只能恢复已纳入版本控制的文件，不能自动恢复：

- 外部数据库；
- 已发送消息；
- 已公开文件；
- 已删除云资源；
- 已泄露密钥。

把 AI Review 当作批准者

模型可以提供审查信号，不能替团队承担发布责任。

将密钥放入提示后要求“不要泄露”

更安全的做法是根本不把无关秘密放进上下文。

自动化失败后继续执行后续步骤

后续动作可能使用旧结果、空结果或错误对象。

每一步都要检查前置状态。

36.12 建立最小团队规范

团队可以将以下内容写入内部使用说明：

```
# Codex 团队使用最小规范

1. 每项任务必须绑定明确仓库、分支和范围。
2. 默认从只读或工作区写入开始，不使用永久全访问。
3. 外部写入、推送、合并、发布和数据变更必须人工批准。
4. 密钥通过受控环境提供，不进入代码、提示、日志和截图。
```

```
5. CI、Codex Review、人工审查和发布授权分别记录。
6. 发现工作区异常、权限异常或结果不确定时立即停止。
7. 自动化必须有失败状态、输出位置和负责人。
8. 临时分支、Worktree、实验目录和权限在任务后清理。
```

这份规范足够短，能够真正被使用。

复杂企业环境还需要：

- 身份和角色体系；
- 集中配置；
- 数据分类；
- 供应商管理；
- 审计和留痕；
- 事故响应；
- 合规评估。

这些内容不在零基础正文中展开。

36.13 收束团队任务看板案例

完成本章检查后，团队任务看板最终状态应为：

- 核心功能完整；
- 截止日期、逾期提示和任务编辑已交付；
- `Escape` 取消编辑已通过质量链；
- 项目规则和知识资料存在；
- GitHub 质量链已经建立；
- 主分支与远程一致；
- 工作区干净；
- 没有临时分支和 Worktree；
- SDK 实验目录位于项目外；
- 非交互结果位于项目外；
- 定时只读检查仍处于暂停状态；
- 不存在仓库密钥和外部自动发布。

从第 37 章开始，本书切换到最后一个综合案例：

> **企业工单管理系统。**

团队任务看板不再新增功能。

三、总结

本章将前面分散的安全原则收束为五条规则：

239. 只在可信工作区执行明确任务；
240. 按任务选择最低权限；
241. 所有副作用都要有明确审批；
242. 密钥不进入代码、提示和日志；
243. 发布必须人工授权，异常必须停止。

同时完成：

- 区分工作区、沙箱、审批和发布授权；
- 建立 A 到 D 四级任务分级；
- 完成团队任务看板发布检查表；
- 对 Skill、MCP、子智能体、定时任务、非交互、SDK 和 GitHub 质量链逐项收口；
- 形成普通团队可执行的最小规范；
- 正式结束团队任务看板案例。

本章掌握检查

- [ ] 能说明沙箱和审批的区别；
- [ ] 能根据任务选择只读或工作区写入；
- [ ] 不把全访问作为默认效率设置；
- [ ] 外部写入和发布都有明确负责人；
- [ ] 密钥没有进入仓库、提示和日志；
- [ ] CI、模型审查、人工批准和发布授权职责分开；
- [ ] 自动化遇到异常能够停止；
- [ ] 团队任务看板最终状态干净、可追溯、可回退。

四、结束语与引导

前 36 章已经完成从零基础到团队工程流程的主线：

```
第一次成果
→ 文件、终端和 Git
→ 任务、计划、修改和验证
→ 从零完成项目
→ 接手陌生项目
→ 跨入口协作
→ 可复用能力和自动化
```

```
→ 非交互、SDK 和 GitHub 质量链
→ 安全与团队规范
```

最后四章不再继续增加工具种类。

第 37—40 章将把已经学过的方法放入一个更接近企业真实工作的项目：

```
接手企业工单系统
→ 开发工单转派功能
→ 修复统计 Bug 并补测试
→ 通过 PR、CI 和业务验收完成交付
```

下一章将通过固定案例说明：

> **企业工单管理系统怎样建立可信代码地图，并复现一个真实问题。**

第 37 章：企业工单系统综合案例，建立可信基线

一、起始语

前 36 章中，你已经分别练习了从零创建项目、接手陌生项目、在不同入口之间协作、建立项目规则、使用 GitHub 质量链，以及控制权限和发布风险。

最后四章不再增加新的工具名词。

我们把已经学过的方法放进一个更接近企业真实工作的固定案例：

> **企业工单管理系统**

案例项目目录名称为：

`enterprise-ticket-system`

第 37—40 章采用完整案例拆解。读者不需要下载或运行同名项目；正文中的目录、命令、代码、测试结果和交付状态用于展示一条连续、可审查的企业项目工作链。有相似项目的读者，可以把这些步骤迁移到自己的环境中。

这个案例不是从空文件夹开始。

系统已经能够：

- 新建工单；
- 查看工单列表和详情；
- 按状态、优先级和负责人筛选；
- 将工单更新为待处理、处理中或已关闭；
- 显示首页汇总数字；
- 将数据保存在 SQLite 数据库中；
- 通过 Node.js 后端提供 API；
- 运行一组已有自动化测试。

案例中的维护者接到两个业务任务。

第一个任务是新增：

> **工单转派功能。**

第二个任务是修复一个已经被客服主管反复反馈的问题：

> **工单关闭后，首页“待处理工单”数量没有同步减少。**

这两件事都不能直接开始写代码。

如果维护者还不知道：

- 项目怎样启动；
- 前端从哪里请求数据；
- 状态在哪里更新；
- 汇总数字怎样计算；
- 数据表之间有什么关系；
- 当前测试覆盖什么；
- 转派需要遵守哪些业务规则；

那么直接修改只是在已有系统中增加新的不确定性。

本章唯一任务是：

> **观察案例怎样运行项目、建立可信代码地图、复现统计问题，并把工单转派需求整理成可以开发和验收的范围。**

本章不要求读者实际修改业务代码、提交候选修复或创建 Pull Request。重点是理解每一步需要什么证据，以及什么条件不满足时应当停止。

二、主体

> **案例阅读说明**
>
> 本章以下目录、命令、提示词、代码、测试结果和 Git 状态均属于固定案例步骤。没有同类项目时，只需阅读并理解每一步解决的问题、所需证据和停止条件，不要在电脑中照抄执行。将方法迁移到自己的项目时，应替换为真实路径、业务规则、命令和验收标准。

37.1 读取案例设定并先确认数据边界

本案例假定 `enterprise-ticket-system` 位于下面的位置。该路径用于说明项目关系，不要求读者在电脑中创建同名目录：

```
Codex 学习/
└── projects/
    ├── ai-learning-page/
    ├── personal-workspace/
    ├── team-task-board/
    └── enterprise-ticket-system/
```

实际接手企业类项目时，第一次连接后先不要运行全部命令。

可以使用下面的只读检查提示词；在自己的项目中使用时，将项目名称和路径替换为真实值：

```
请只检查当前 enterprise-ticket-system 项目，不要创建、修改、移动或删除任何文件，也不要启动服务。

请根据真实文件回答：

1. 当前工作目录；
2. 一级目录和关键文件；
3. 是否存在 Git 仓库，当前分支和工作区状态；
4. 使用的运行时、包管理器、数据库和测试方式；
5. 是否存在.env、数据库文件、日志、导出文件或真实客户数据；
6. 哪些文件会在运行时生成；
7. 哪些内容已经被.gitignore 排除；
8. 是否存在安装、迁移或启动说明。

不要读取或输出任何疑似密钥、访问令牌、真实客户信息和个人数据的具体内容。如果发现真实业务数据，请停止并说明风险。
```

本案例假定项目只包含：

- 虚构工单；
- 虚构坐席；
- 本地示例数据库初始化脚本；
- `.env.example`；
- 不含凭据的配置；
- 可以重新生成的本地数据库。

如果你拿到的实际企业项目含有生产数据库副本、客户姓名、手机号、访问令牌或内部地址，不应继续照着本章运行。

先完成脱敏、权限确认和环境隔离。

37.2 读取项目规则和启动说明

项目根目录包括：

```
enterprise-ticket-system/
├── public/
├── src/
├── tests/
├── docs/
├── data/
├── .env.example
├── .gitignore
├── AGENTS.md
├── package.json
├── package-lock.json
└── README.md
```

先读取：

- `AGENTS.md`；
- `README.md`；

- `package.json`；
- `.gitignore`；
- `docs/current-state.md`；
- `docs/product-rules.md`。

可以向 Codex 提出：

```
请只读取 AGENTS.md、README.md、package.json、.gitignore、docs/current-state.md 和 docs/product-rules.md。

请输出：

1. 项目怎样安装、初始化、启动和测试；
2. 哪些目录允许修改；
3. 哪些生成文件不能提交；
4. 当前业务状态和已知限制；
5. 哪些规则属于事实，哪些内容需要与代码核对；
6. 本章开始前必须满足的停止条件。

不要修改文件，不要直接执行命令。
```

文档只能提供接手起点，不能自动成为事实。

例如，README 写着“首页汇总会实时更新”，但客服主管已经反馈实际行为不一致。

因此，后续必须分别核对：

- 文档；
- 代码；
- 运行结果；
- 数据库状态；
- 测试结果。

37.3 建立仓库基线

在项目根目录运行：

```
git status
git branch --show-current
git log --oneline -5
```

预期状态：

- 当前分支为 `main`；
- 工作区干净；
- 项目带有可信历史；
- 最近提交是案例项目的稳定基线。

不要为了“方便练习”重新执行 `git init`，也不要删除已有提交。

随后记录当前提交：

```
git rev-parse --short HEAD
```

这个提交号用于说明：

> 后续所有功能和修复从哪个稳定版本开始。

若 `git status` 显示未知修改，先停止。

需要判断：

- 修改是不是你刚刚产生的；
- 数据库是否被误提交；
- 是否有人留下未完成工作；
- 当前目录是否真的是目标项目；
- 是否应该切换到正确分支或重新复制项目。

不能在来源不明的修改上继续综合实战。

37.4 按说明安装和初始化

先确认 Node.js 和包管理器可用：

```
node --version
npm --version
```

然后根据锁文件安装依赖：

```
npm ci
```

为什么使用 `npm ci`，而不是直接执行 `npm install`？

因为项目已经包含 `package-lock.json`。

在这种情况下，`npm ci` 更适合建立可重复的练习环境：

- 按锁文件安装；
- 不主动改写依赖版本；
- 环境与项目基线更容易对齐。

安装后先检查 Git：

```
git status --short
```

`node_modules` 不应进入 Git 状态。

随后初始化示例数据库：

```
npm run db:reset
```

这条命令只适用于案例中定义的虚构示例数据库。实际项目执行重置前，必须再次确认数据库目标和数据边界。

它会：

- 删除旧的本地练习数据库；
- 重新执行迁移；
- 写入虚构工单和坐席；
- 建立可重复的测试起点。

在真实项目中，看到 `reset`、`drop`、`seed` 或 `migrate` 命令时，必须先确认目标数据库。

不能因为命令写在 README 中，就对生产环境执行。

运行后检查：

```
git status --short
```

预期只生成被 `.gitignore` 排除的本地数据库，不改变受版本控制文件。

37.5 运行现有测试，建立自动化基线

在启动页面前，先运行：

```
npm test
```

记录：

- 测试总数；
- 通过数；
- 失败数；
- 退出状态；
- 是否有跳过测试；
- 是否产生仓库修改。

不要只记录一句“测试通过”。

例如，可以写成：

```
基线测试结果：
- 命令：npm test
- 退出状态：0
- 通过：12
- 失败：0
- 跳过：0
- Git 工作区：干净
```

当前测试通过，只能说明：

> **已有测试覆盖的行为没有失败。**

它不能证明：

- 工单转派已经存在；
- 首页汇总一定正确；
- 所有业务边界都被测试；
- 页面交互没有问题。

统计 Bug 能够在测试全部通过时仍然存在，说明当前测试存在覆盖缺口。

37.6 启动系统并记录运行基线

运行：

```
npm run dev
```

终端应显示：

- 本地服务地址；
- 数据库连接状态；
- 是否完成迁移；
- 是否出现端口冲突或启动错误。

在浏览器打开 README 给出的本地地址。

第一次观察时不要随意操作，先记录：

- 首页汇总卡片；
- 工单总数；
- 待处理数量；
- 处理中数量；
- 已关闭数量；
- 列表中示例工单；
- 筛选器；
- 详情面板；
- 当前负责人；
- 状态更新入口。

本书标准示例中，重置数据库后的待处理数量为一个固定基线。你应以实际页面和 API 返回为准，不要照抄书中的示例数字。

截图可以保存到项目外的练习记录目录，例如：

```
Codex 学习/
└── evidence/
    └── enterprise-ticket-system/
        └── baseline-home.png
```

不要把临时截图直接放入项目，除非项目规则明确要求提交验证证据。

37.7 建立第一层项目地图：从目录到职责

让 Codex 只读生成项目地图：

```
请只读分析当前企业工单管理系统，不要修改文件。

请按以下结构输出项目地图：

1. 浏览器入口；
2. 前端 API 封装；
3. 前端渲染和事件处理；
4. Express 应用入口；
5. 路由；
6. 服务层；
7. 数据访问层；
8. 数据库迁移；
9. 自动化测试；
10. 项目文档。

每项必须列出真实文件路径和主要导出对象。找不到的内容请写“未找到”，不要根据常见项目结构补写。
```

标准项目的主要结构是：

```
浏览器
→ public/js/app.js
→ public/js/api.js
→ /api/*
→ src/routes/*
→ src/services/*
→ src/repositories/*
→ SQLite
```

这个结构只表示大致层次。

真正修改功能前，还需要追踪具体请求。

37.8 追踪现有“更新状态”请求

工单关闭 Bug 发生在状态更新后。

因此，先从用户动作开始追踪：

> 用户在详情面板中将工单设置为“已关闭”。

向 Codex 提出：

```
请追踪“用户将一张工单更新为已关闭”的完整请求路径，不要修改代码。

请说明：

1. 哪个前端事件开始处理；
2. 调用了哪个前端 API 函数；
3. 请求方法和路径；
4. 哪个 Express 路由接收；
5. 哪个服务函数执行规则；
6. 哪个仓储函数更新 SQLite；
7. 状态和 closed_at 怎样变化；
8. 响应返回后前端刷新了哪些区域；
9. 首页汇总数据从哪里重新获取；
10. 当前测试覆盖到哪一层。

每个判断都引用真实文件和函数名。无法确认时明确说明。
```

你需要核对 Codex 的输出。

可以使用搜索：

```
rg "closed|updateStatus|summary|pending" public src tests
```

若系统没有安装 `rg`，可以让 Codex 使用当前环境可用的搜索工具，或者通过编辑器全局搜索。

不要为了执行一条搜索命令临时安装不必要依赖。

37.9 复现“待处理数量未同步”问题

按照固定步骤复现，避免边操作边改变条件。

复现前

244. 执行 `npm run db:reset`；
245. 重新启动服务；
246. 打开首页；
247. 记录待处理数量；
248. 选择一张状态为 `open` 或 `processing` 的工单；
249. 记录工单编号和当前状态。

执行操作

250. 打开工单详情；
251. 将状态更新为“已关闭”；
252. 确认列表和详情显示“已关闭”；
253. 观察首页待处理数量；
254. 刷新页面；
255. 再次观察汇总数字。

预期行为

关闭一张原本属于待处理范围的工单后：

- 工单状态变为 `closed`；
- `closed_at` 被写入；
- 待处理数量减少 1；
- 已关闭数量增加 1；
- 刷新后结果保持一致。

实际行为

案例项目中：

- 工单状态正确变为 `closed`；
- 已关闭数量正确增加；
- 待处理数量没有减少；
- 刷新后错误仍然存在。

这说明问题不是单纯的前端旧状态。

37.10 用 API 和数据库证据缩小问题范围

页面现象还不足以定位根因。

需要分别检查工单详情和汇总 API。

例如：

```
curl http://localhost:3000/api/tickets/TK-1007
curl http://localhost:3000/api/tickets/summary
```

Windows 环境中若 `curl` 行为不同，可以使用 PowerShell：

```
Invoke-RestMethod http://localhost:3000/api/tickets/TK-1007
Invoke-RestMethod http://localhost:3000/api/tickets/summary
```

确认：

- 工单 API 已经返回 `status: "closed"`；
- 汇总 API 仍然将该工单计入 `pendingCount`。

再让 Codex 只读检查数据库：

```
请使用项目现有的安全查询方式，只读取示例数据库中刚刚操作的工单。

请确认：
1. status；
2. assignee_id；
```

```
3. updated_at;
4. closed_at。

不要修改数据库，不要输出其他工单的完整内容。
```

若数据库中状态也正确，那么证据链是：

```
数据库状态正确
→ 单张工单 API 正确
→ 汇总 API 错误
→ 页面按错误汇总结果显示
```

问题范围已经缩小到：

> **汇总计算路径。**

本章仍然不修复。

37.11 找到可疑代码，但不提前下结论

继续追踪：

```
请只读追踪 GET /api/tickets/summary 的完整实现。

列出：
1. 路由；
2. 服务函数；
3. SQL 或聚合逻辑；
4. 状态常量来源；
5. 对 open、processing、closed 分别怎样计数；
6. 相关自动化测试；
7. 代码与 docs/product-rules.md 是否一致。

不要修改文件，不要直接给出补丁。
```

你可能看到 `summary-service.js` 仍然使用旧状态名称，例如将 `resolved` 当作关闭状态。

这是一条高价值线索，但本章不应直接写成：

> 根因已经确认，只要替换一个单词即可。

还需要在第 39 章完成：

- 让失败行为进入自动化测试；
- 证明测试在修复前失败；
- 使用业务规则而不是偶然字符串修复；
- 运行全部回归；
- 验证转派功能不改变汇总数量。

当前更准确的记录是：

> 汇总服务中的状态判断与当前状态定义存在疑似不一致，是第 39 章优先验证的根因假设。

37.12 建立现有数据模型

读取迁移文件和仓储代码后，记录工单表的核心字段：

字段	作用
id	工单编号
title	工单标题
description	问题说明
status	open、processing 或 closed
priority	low、medium 或 high
assignee_id	当前负责人
created_at	创建时间
updated_at	最近更新时间
closed_at	关闭时间，未关闭时为空

坐席表至少包括：

字段	作用
id	坐席标识
name	显示名称
team	所属支持团队
is_active	是否可以接收新工单

当前项目没有转派历史表。

这意味着工单转派不能只改 `assignee_id`。

如果系统只保留最后负责人，就无法回答：

- 谁发起了转派；
- 从谁转给谁；
- 为什么转派；
- 什么时候发生；
- 是否出现多次转派。

因此，新功能需要新增可审计历史。

37.13 将“支持转派”改写为业务规则

原始需求可能只有一句：

> 给工单加一个转派功能。

这句话不够开发。

与业务方确认后，本书固定以下规则。

可以转派的工单

- `open`；
- `processing`。

不允许转派的工单

- `closed`。

关闭工单需要先重新打开，不能通过转派绕过状态规则。

目标负责人

- 必须存在；
- 必须处于启用状态；
- 不能与当前负责人相同。

转派原因

- 必填；
- 去除首尾空格后 5—200 个字符；
- 保存实际原因，不只保存固定选项。

数据一致性

一次成功转派必须同时完成：

```
更新 tickets.assignee_id
+
写入 ticket_assignment_history
```

两步必须位于同一数据库事务。

若写入历史失败，负责人更新也必须回滚。

操作人

案例项目没有登录系统。

为了避免前端伪造身份，演示操作人由服务端固定配置提供。

真实企业系统中，操作人应来自已经验证的登录会话和权限体系。

不属于本次范围

- 不发送站内信、邮件或短信；
- 不实现跨团队审批；
- 不实现批量转派；
- 不实现自动分单；
- 不实现坐席负载均衡；
- 不增加角色权限系统；
- 不修改工单状态定义；
- 不改变首页汇总口径。

37.14 定义转派 API 契约

第 38 章将新增：

```
POST /api/tickets/:id/reassign
```

请求体：

```
{
  "assigneeId": "agent-03",
  "reason": "该问题需要由支付接口支持人员继续处理"
}
```

操作人不由前端传入。

成功响应应包含：

- 更新后的工单；
- 新增的转派历史记录。

错误情况至少包括：

情况	建议状态码	用户可见信息
工单不存在	404	未找到目标工单
目标坐席不存在	400 或 404	目标负责人不可用
目标坐席已停用	400	目标负责人不可接单
工单已关闭	409	已关闭工单不能直接转派
与当前负责人相同	400	请选择其他负责人
原因为空或过短	400	请填写有效转派原因
数据库事务失败	500	转派未完成，请稍后重试

状态码的最终选择需要与项目已有错误处理方式保持一致，不能只按个人偏好重做整套 API。

37.15 定义前端用户流程

前端最小流程为：

```
打开工单详情
→ 点击“转派工单”
→ 选择启用坐席
→ 填写转派原因
→ 确认
→ 显示成功结果
→ 详情和列表负责人同步更新
```

还必须处理：

- 已关闭工单不显示可执行的转派按钮；
- 没有可选坐席时给出清楚提示；
- 提交期间防止重复点击；
- 服务端拒绝时保留用户已填写内容；
- 成功后关闭对话框并更新当前页面；
- 转派不改变工单状态；
- 转派不改变待处理、处理中和已关闭数量。

前端不能只依赖按钮隐藏保护规则。

服务端必须独立验证所有业务条件。

37.16 形成可验收的功能清单

案例在第 38 章完成后，至少需要验证：

256. 待处理工单可以转给另一名启用坐席；
257. 处理中工单可以转派；
258. 已关闭工单被服务端拒绝；
259. 不能转给当前负责人；
260. 不能转给停用坐席；
261. 转派原因为空时被拒绝；
262. 原因长度超出上限时被拒绝；
263. 成功后负责人更新；
264. 成功后新增历史记录；
265. 更新负责人和写入历史保持原子性；
266. 页面列表和详情显示一致；
267. 转派前后工单状态不变；
268. 转派前后汇总数字不变；
269. 刷新页面后负责人和历史仍然存在；

270. 原有新建、筛选、更新状态和关闭工单功能不受影响。

这些条目同时决定：

- 服务层测试；
- API 测试；
- 手工浏览器测试；
- 最终业务验收。

37.17 记录停止条件

本章结束前，明确以下停止条件。

真实项目出现任一情况时，不应进入后续开发阶段：

- 项目不是干净 `main`；
- 测试基线失败且原因不明；
- 项目含真实客户数据；
- 数据库重置目标不明确；
- 转派业务规则没有确认；
- 工单状态定义与文档冲突；
- 当前负责人字段来源不明；
- 无法稳定复现汇总 Bug；
- 不知道应该怎样验证回归；
- 计划要求自动发布或连接生产系统。

综合实战的目的不是强行走完步骤。

当基线不可信时，正确动作就是停止。

37.18 保存本章证据，但保持仓库不变

案例将以下证据保存在项目外：

- 基线截图；
- 测试结果；
- API 返回摘要；
- Bug 复现步骤；
- 转派需求卡；
- 项目地图。

案例最后检查：

```
git status --short
git diff --stat
```

预期：

- 没有受版本控制文件变化；
- 没有新分支；
- 没有提交；
- 没有 Worktree；
- 本地示例数据库仍被忽略。

第 37 章案例结束时，我们已经知道该改什么，但尚未修改。

这正是可信开发的起点。

三、总结

本章案例没有开发工单转派，也没有修复统计问题。

案例展示了更重要的准备工作：

- 确认项目和数据边界；
- 读取项目规则和启动说明；
- 建立 Git、依赖、数据库和测试基线；
- 运行系统并记录页面状态；
- 建立前端、后端、服务和数据库地图；
- 追踪状态更新和汇总请求；
- 用页面、API 和数据库证据稳定复现 Bug；
- 将根因范围缩小到汇总计算路径；
- 记录现有数据模型；
- 冻结转派业务规则、API 契约、用户流程和验收清单；
- 明确停止条件；
- 保持仓库干净。

本章掌握检查

- [ ] 能在运行陌生企业项目之前检查数据和权限边界；
- [ ] 能区分文档说明、代码实现和运行事实；
- [ ] 能建立仓库、测试和页面基线；
- [ ] 能从用户动作追踪到数据库；

- [ ] 能通过页面、API 和数据库证据缩小 Bug 范围；
- [ ] 不会看到可疑代码后立即修改；
- [ ] 能把模糊功能需求改写为业务规则和验收标准；
- [ ] 能明确本次不做什么；
- [ ] 能说明为什么案例在本章结束时仍应保持 Git 工作区干净。

四、结束语与引导

到这里，案例已经展示了企业项目接手中最容易被省略的一步：

> **先建立事实，再开始行动。**

第 38 章将展示案例怎样创建隔离 Worktree 和功能分支，并按照本章冻结的规则完成工单转派。

这次修改会跨越：

```
数据库迁移
→ 仓储层
→ 服务层
→ API 路由
→ 前端交互
→ 自动化测试
→ 项目文档
```

下一章的目标不是“先做一个能点的按钮”。

而是：

> **完成一个数据可追溯、失败可回滚、前后端一致、可以进入审查的完整候选实现。**

第 38 章：企业工单系统综合案例，开发转派功能并控制跨层修改

一、起始语

上一章的案例结束时，企业工单管理系统仍然处于干净的 `main` 分支。

本章继续沿用同一个固定案例。下面的 Worktree、迁移、服务、API、前端和测试步骤用于展示跨层功能怎样被分解和验证，不要求读者拥有同名仓库。

案例已经确认：

- 项目可以安装、启动和测试；
- 现有工单状态为 `open`、`processing` 和 `closed`；
- 工单当前只保存最后负责人；
- 系统尚无转派历史；
- 首页统计存在一个已稳定复现的问题；
- 统计问题留到第 39 章处理；
- 本章只开发工单转派。

工单转派看起来只是“换一个负责人”。

但一个可交付的实现至少涉及：

```
业务规则
→ 数据结构
→ 数据事务
→ 服务验证
→ API 契约
→ 前端交互
→ 错误反馈
→ 自动化测试
→ 手工回归
→ 文档更新
```

如果只在前端增加下拉框，再直接更新 `assignee_id`，功能可能看起来可用，却无法审计，也无法保证更新失败时数据一致。

本章唯一任务是：

> **观察案例怎样在隔离 Worktree 中完成工单转派的完整候选实现，并形成一个聚焦的功能提交。**

本章案例不修复待处理数量 Bug，不合并主分支，也不发布系统。读者应重点理解跨层修改的顺序、事务边界和验证证据。

二、主体

> **案例阅读说明**
>
> 本章以下目录、命令、提示词、代码、测试结果和 Git 状态均属于固定案例步骤。没有同类项目时，只需阅读并理解每一步解决的问题、所需证据和停止条件，不要在电脑中照抄执行。将方法迁移到自己的项目时，应替换为真实路径、业务规则、命令和验收标准。

38.1 再次确认起始状态

案例流程首先进入项目根目录：

```
cd Codex 学习/projects/enterprise-ticket-system
```

检查：

```
git status
git branch --show-current
git log --oneline -3
npm test
```

预期：

- 当前分支为 `main`；
- 工作区干净；
- 测试基线通过；
- 第 37 章没有留下仓库修改。

真实项目中如果这些条件不满足，不应创建功能分支。

38.2 创建隔离 Worktree

案例在项目父目录中创建功能分支和工作树：

```
git worktree add ../enterprise-ticket-system-reassign -b feature/ticket-reassignment main
```

进入新目录：

```
cd ../enterprise-ticket-system-reassign
```

检查：

```
git branch --show-current
git status
git worktree list
```

预期当前分支为：

```
feature/ticket-reassignment
```

为什么使用 Worktree?

因为本章需要跨多个层级修改。

Worktree 让你可以：

- 保留原项目的干净 `main`；
- 在独立目录中开发；
- 随时比较稳定版本和候选版本；
- 避免频繁切换分支；
- 在任务失败时直接保留或删除候选环境。

Worktree 不是备份。

所有重要变化仍然需要 Git 提交。

38.3 让 Codex 先生成实施计划

连接新 Worktree 后，先要求只读计划：

```
我们要在 enterprise-ticket-system 中实现工单转派功能。

请先只读探索，不要修改文件。

业务规则：
1. open 和 processing 工单可以转派；
2. closed 工单不能转派；
3. 目标坐席必须存在且启用；
4. 不能转给当前负责人；
5. 原因去除首尾空格后为 5—200 个字符；
6. 更新负责人和写入转派历史必须在同一事务；
7. 转派不改变工单状态和首页汇总；
8. 操作人来自服务端固定演示配置；
9. 不实现通知、审批、批量转派和权限系统；
10. 不修复当前待处理数量 Bug。

请输出：
- 需要新增和修改的文件；
- 每个文件承担的职责；
- 数据库迁移方案；
- 服务层验证顺序；
- API 契约；
- 前端交互；
- 自动化测试；
- 手工验收；
- 每阶段停止条件。

不要直接生成代码。
```

计划应分成至少五个阶段：

```
阶段 A：数据结构和事务边界
阶段 B：服务与 API
阶段 C：前端交互
阶段 D：自动化测试和文档
阶段 E：完整验证与提交
```

若计划提出以下内容，应退回修正：

- 引入新框架；
- 重写整个数据访问层；
- 把统计 Bug 一起修复；
- 新增登录系统；
- 让前端直接传入可信操作人；
- 将两个数据库写入拆成独立请求；
- 自动合并或自动发布。

38.4 设计转派历史表

新增迁移：

`src/db/migrations/003_assignment_history.sql`

示例结构：

```
CREATE TABLE ticket_assignment_history (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  ticket_id TEXT NOT NULL,
  from_assignee_id TEXT NOT NULL,
  to_assignee_id TEXT NOT NULL,
  reason TEXT NOT NULL,
  operator_name TEXT NOT NULL,
  created_at TEXT NOT NULL,
  FOREIGN KEY (ticket_id) REFERENCES tickets(id),
  FOREIGN KEY (from_assignee_id) REFERENCES agents(id),
  FOREIGN KEY (to_assignee_id) REFERENCES agents(id)
);

CREATE INDEX idx_assignment_history_ticket
  ON ticket_assignment_history(ticket_id, created_at DESC);
```

在实际项目中，需要核对：

- 主表主键类型；
- 外键是否启用；
- 时间字段格式；
- 迁移命名规则；
- SQLite 版本；
- 是否已有统一索引规范；
- 迁移运行器是否会自动发现新文件，还是需要在 `database.js` 中显式登记。

只有在真实运行器要求时才修改迁移登记代码。不要因为示例 SQL 合理就直接覆盖项目习惯。

38.5 为什么不能只修改工单表

如果只执行：

```
UPDATE tickets
SET assignee_id = ?, updated_at = ?
WHERE id = ?;
```

系统只能知道“现在是谁负责”。

它无法回答：

- 原负责人是谁；
- 谁发起转派；
- 为什么转派；
- 转派发生了几次；
- 某次转派是否完成。

工单系统属于典型的过程型业务系统。

当前状态和历史证据承担不同职责：

数据	回答的问题
tickets.assignee_id	现在由谁负责
ticket_assignment_history	之前怎样变化

两者必须同时保存。

38.6 建立转派历史仓储

新增：

`src/repositories/assignment-history-repository.js`

它只负责数据访问，不负责决定业务规则。

示例接口：

```
export function createAssignmentHistory(db, entry) {
  const statement = db.prepare(`
    INSERT INTO ticket_assignment_history (
      ticket_id,
      from_assignee_id,
      to_assignee_id,
      reason,
      operator_name,
```

```
      created_at
    ) VALUES (?, ?, ?, ?, ?, ?)
  `);

  const result = statement.run(
    entry.ticketId,
    entry.fromAssigneeId,
    entry.toAssigneeId,
    entry.reason,
    entry.operatorName,
    entry.createdAt
  );

  return {
    id: result.lastInsertRowid,
    ...entry
  };
}
```

还可以提供：

```
export function listAssignmentHistoryByTicketId(db, ticketId) {
  return db.prepare(`
    SELECT
      history.id,
      history.ticket_id AS ticketId,
      history.from_assignee_id AS fromAssigneeId,
      from_agent.name AS fromAssigneeName,
      history.to_assignee_id AS toAssigneeId,
      to_agent.name AS toAssigneeName,
      history.reason,
      history.operator_name AS operatorName,
      history.created_at AS createdAt
    FROM ticket_assignment_history AS history
    JOIN agents AS from_agent
      ON from_agent.id = history.from_assignee_id
    JOIN agents AS to_agent
      ON to_agent.id = history.to_assignee_id
    WHERE history.ticket_id = ?
    ORDER BY history.created_at DESC, history.id DESC
  `).all(ticketId);
}
```

查询历史时连接坐席表，是为了让前端获得可读姓名。

但历史表仍然保存坐席 ID，避免只依赖可能变化的显示名称。

38.7 扩展工单仓储的最小能力

现有 `ticket-repository.js` 已经能够：

- 根据 ID 读取工单；
- 更新状态；
- 查询列表。

本章只新增必要的负责人更新：

```
export function updateTicketAssignee(db, ticketId, assigneeId, updatedAt) {
  const result = db.prepare(`
    UPDATE tickets
    SET assignee_id = ?, updated_at = ?
    WHERE id = ?
  `).run(assigneeId, updatedAt, ticketId);

  return result.changes;
}
```

这个函数不自行开始事务，也不验证目标坐席。

事务和业务规则属于服务层。

如果仓储函数同时检查状态、坐席启用、原因长度和操作人，后续测试和复用都会变得困难。

38.8 在服务层集中业务规则

在 `ticket-service.js` 中新增：

```
export function reassignTicket({
  db,
  ticketId,
  toAssigneeId,
  reason,
  operatorName,
  now = () => new Date().toISOString()
}) {
  // 业务验证和事务
}
```

把 `now` 作为可替换函数传入，便于测试固定时间。

服务层验证顺序建议为：

271. 清理输入；
272. 查找工单；
273. 检查工单状态；
274. 查找目标坐席；
275. 检查坐席启用状态；
276. 检查是否与当前负责人相同；
277. 检查原因长度；
278. 开始事务；
279. 更新负责人；
280. 写入历史；
281. 返回最新工单和历史。

为什么先验证再开启事务？

因为大部分无效请求可以在写入前拒绝，减少不必要的数据库锁和回滚。

38.9 规范化转派原因

服务层先处理：

```
const normalizedReason = String(reason ?? '').trim();
```

然后验证：

```
if (normalizedReason.length < 5 || normalizedReason.length > 200) {
  throw new ValidationError('转派原因需为 5—200 个字符');
}
```

不要在数据库中保存大量首尾空格。

也不要只在前端检查。

API 可以被其他客户端调用，服务端必须独立保护规则。

38.10 用事务保证负责人和历史一致

示例：

```
const transaction = db.transaction(() => {
  const updatedAt = now();

  const changed = updateTicketAssignee(
    db,
    ticketId,
    toAssigneeId,
    updatedAt
  );

  if (changed !== 1) {
    throw new Error('工单负责人更新失败');
  }

  const history = createAssignmentHistory(db, {
    ticketId,
    fromAssigneeId: ticket.assigneeId,
    toAssigneeId,
    reason: normalizedReason,
    operatorName,
    createdAt: updatedAt
  });

  return {
    ticket: findTicketById(db, ticketId),
    history
  };
});

return transaction();
```

事务要保护的不是“代码看起来连续”，而是数据库结果：

```
两步都成功
或
两步都不发生
```

若历史写入失败但负责人已经变化，系统会留下无法解释的状态。

38.11 保持状态和汇总不变

转派只改变负责人和更新时间。

它不能修改：

- `status`；
- `closed_at`；
- 工单总数；
- 待处理数量；
- 处理中数量；
- 已关闭数量。

在服务测试中应明确断言，而不是默认相信。

例如：

```
assert.equal(result.ticket.status, originalTicket.status);
assert.equal(result.ticket.closedAt, originalTicket.closedAt);
```

第 39 章还会增加一项回归：

> 转派前后待处理汇总保持不变。

38.12 增加可用坐席查询

前端需要知道可以转给谁。

现有 `agent-repository.js` 可以新增或复用：

```
export function listActiveAgents(db) {
  return db.prepare(`
    SELECT id, name, team
    FROM agents
    WHERE is_active = 1
    ORDER BY name
  `).all();
}
```

路由：

```
GET /api/agents?active=true
```

不要把所有内部字段返回给浏览器。

最小响应只包含：

- `id`；
- `name`；
- `team`。

38.13 由服务端提供演示操作人

在训练项目中，操作人不能由浏览器请求体自行声明。

在 `src/app.js` 中设置服务端值：

```
app.locals.operatorName =
  process.env.DEMO_OPERATOR_NAME?.trim() || '演示主管';
```

`.env.example` 可以记录非敏感配置名：

```
DEMO_OPERATOR_NAME=演示主管
```

这只是案例项目中的过渡方案。真实企业系统应从已经验证的登录会话、服务身份或受控任务身份获取操作人。

38.14 增加转派 API 路由

在 `src/routes/tickets.js` 增加：

```
router.post('/:id/reassign', (req, res, next) => {
  try {
    const result = reassignTicket({
      db: req.app.locals.db,
      ticketId: req.params.id,
      toAssigneeId: req.body.assigneeId,
      reason: req.body.reason,
      operatorName: req.app.locals.operatorName
    });

    res.json(result);
  } catch (error) {
    next(error);
  }
});
```

注意操作人来自：

```
req.app.locals.operatorName
```

而不是请求体。

训练项目可以固定为“演示主管”。

真实系统应从已经验证的登录身份获取。

38.15 让错误保持可区分

服务层可以使用项目已有的错误类型：

- `NotFoundError`；
- `ValidationError`；
- `ConflictError`。

例如：

```
if (!ticket) {
  throw new NotFoundError('未找到目标工单');
}

if (ticket.status === 'closed') {
  throw new ConflictError('已关闭工单不能直接转派');
}
```

不要把所有错误都变成 500。

也不要把数据库内部错误直接返回给用户。

API 错误应同时满足：

- 前端能给出可理解提示；
- 日志保留足够排查信息；
- 不泄露 SQL、文件路径和堆栈。

38.16 先完成服务层自动化测试

在修改前端之前，先测试核心规则。

在 `tests/ticket-service.test.js` 增加：

成功转派

```
test('open 工单可以转派并记录历史', () => {
  const result = reassignTicket({
    db,
    ticketId: 'TK-1001',
    toAssigneeId: 'agent-03',
    reason: '需要支付接口支持人员继续处理',
    operatorName: '演示主管',
    now: () => '2026-07-31T10:00:00.000Z'
  });

  assert.equal(result.ticket.assigneeId, 'agent-03');
  assert.equal(result.history.fromAssigneeId, 'agent-01');
  assert.equal(result.history.toAssigneeId, 'agent-03');
  assert.equal(result.history.operatorName, '演示主管');
});
```

已关闭工单拒绝转派

```
test('closed 工单不能转派', () => {
  assert.throws(
    () => reassignTicket({
      db,
      ticketId: 'TK-1005',
      toAssigneeId: 'agent-03',
      reason: '需要其他人员继续处理',
      operatorName: '演示主管'
    }),
    /已关闭工单不能直接转派/
  );
});
```

还需要覆盖：

- 目标坐席不存在；
- 目标坐席停用；
- 与当前负责人相同；
- 原因为空；
- 原因过短；
- 原因过长；
- 原因首尾空格被清理；
- 工单状态不变；
- 事务失败时负责人不变；
- 成功后历史可以查询。

38.17 特别验证事务回滚

事务测试不能只模拟普通成功。

可以在测试数据库中故意让历史写入失败，例如临时删除目标表或注入抛错仓储。

随后断言：

```
const ticketAfterFailure = findTicketById(db, 'TK-1001');
assert.equal(ticketAfterFailure.assigneeId, 'agent-01');
```

这个测试证明：

> 历史失败时，负责人不会留下半完成变化。

如果测试只断言“函数抛错”，仍然没有验证数据是否回滚。

38.18 更新前端 API 封装

在 `public/js/api.js` 增加：

```
export async function fetchActiveAgents() {
  return requestJson('/api/agents?active=true');
}

export async function reassignTicket(ticketId, payload) {
  return requestJson(`/api/tickets/${ticketId}/reassign`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(payload)
  });
}
```

继续使用项目已有的 `requestJson`，不要在新函数中重新实现一套错误处理。

38.19 增加转派对话框

在 `public/index.html` 加入最小对话框：

```
<dialog id="reassign-dialog">
  <form id="reassign-form" method="dialog">
    <h2>转派工单</h2>

    <label for="reassign-assignee">目标负责人</label>
    <select id="reassign-assignee" required></select>

    <label for="reassign-reason">转派原因</label>
    <textarea
      id="reassign-reason"
      minlength="5"
      maxlength="200"
      required
    ></textarea>

    <p id="reassign-error" role="alert"></p>

    <div class="dialog-actions">
      <button type="button" id="cancel-reassign">取消</button>
      <button type="submit" id="confirm-reassign">确认转派</button>
    </div>
  </form>
</dialog>
```

HTML 属性可以提供基础提示，但服务端仍然是最终规则来源。

38.20 只对允许状态显示转派入口

详情渲染时：

```
const canReassign =
  ticket.status === 'open' || ticket.status === 'processing';
```

按钮可以：

- 对允许状态显示；

- 对已关闭状态隐藏或禁用；
- 同时显示原因说明。

但不能把按钮状态当作权限控制。

用户仍然可能直接调用 API。

38.21 打开对话框时加载可用坐席

推荐流程：

```
点击转派
→ 记录当前工单
→ 请求启用坐席
→ 排除当前负责人
→ 填充下拉框
→ 打开对话框
```

若没有其他启用坐席：

- 不允许提交；
- 显示“当前没有可转派的负责人”；
- 不保留空白下拉框让用户猜测。

38.22 处理提交中的状态

提交时：

282. 读取目标负责人和原因；
283. 清空旧错误；
284. 禁用确认按钮；
285. 调用 API；
286. 成功后更新界面；
287. 失败时显示服务端可读信息；
288. 最终恢复按钮。

示例：

```
async function handleReassignSubmit(event) {
  event.preventDefault();

  const button = document.querySelector('#confirm-reassign');
  button.disabled = true;

  try {
    const result = await reassignTicket(activeTicketId, {
      assigneeId: assigneeSelect.value,
      reason: reasonInput.value
    });
```

```
    updateTicketInState(result.ticket);
    renderTicketList();
    renderTicketDetail(result.ticket);
    reassignDialog.close();
  } catch (error) {
    reassignError.textContent = error.message;
  } finally {
    button.disabled = false;
  }
}
```

失败时不要关闭对话框，也不要清空原因。

否则用户需要重新输入。

38.23 更新负责人后保持其他状态稳定

成功转派后，前端需要更新：

- 列表中的负责人；
- 详情中的负责人；
- 当前内存状态；
- 可选操作入口。

不需要重新加载汇总，因为转派不改变状态。

但为了验证没有误改，可以在手工测试中对比转派前后汇总。

38.24 增加历史查询并显示最近转派记录

本章验收要求刷新页面后仍然能够看到转派历史，因此历史查询不是可选项。

新增只读 API：

```
GET /api/tickets/:id/assignment-history
```

它调用 `listAssignmentHistoryByTicketId`，按照时间倒序返回：

- 原负责人；
- 新负责人；
- 原因；
- 操作人；
- 时间。

前端打开工单详情时，应同时读取工单和对应历史。成功转派后，可以直接使用服务端返回的新历史更新界面；刷新页面后，再通过历史 API 恢复完整记录。

这不是独立审计系统，只是让业务用户看到当前工单的可追溯变化。前端不得根据当前负责人自行拼造历史，服务端也不得把失败事务写成成功记录。

38.25 更新样式，但不重做页面

在 `styles.css` 中只增加：

- 对话框布局；
- 表单间距；
- 错误提示；
- 提交中状态；
- 历史记录样式。

不要借功能开发：

- 重做颜色系统；
- 更换全部按钮；
- 改写导航；
- 调整无关卡片；
- 引入新的 CSS 框架。

一个功能 PR 的视觉变化应服务于功能本身。

38.26 运行阶段性检查

每完成一个阶段，运行：

```
npm test
git diff --stat
git status --short
```

如果项目提供语法或格式检查，再运行：

```
npm run check
```

检查差异时重点确认：

- 是否只修改计划文件；
- 是否误动统计服务；
- 是否出现数据库文件；
- 是否改写锁文件；
- 是否新增无关依赖；
- 是否泄露演示配置。

38.27 执行 API 级验证

启动候选版本：

```
npm run db:reset
npm run dev
```

先读取启用坐席：

```
Invoke-RestMethod http://localhost:3000/api/agents?active=true
```

再执行成功转派：

```
$body = @{
  assigneeId = 'agent-03'
  reason = '需要支付接口支持人员继续处理'
} | ConvertTo-Json

Invoke-RestMethod `
  -Method Post `
  -Uri http://localhost:3000/api/tickets/TK-1001/reassign `
  -ContentType 'application/json' `
  -Body $body
```

核对响应：

- 工单负责人已更新；
- 状态不变；
- 历史记录存在；
- 操作人来自服务端。

随后分别测试：

- 同一负责人；
- 停用坐席；
- 已关闭工单；
- 空原因；
- 过短原因。

38.28 执行浏览器验收

按固定矩阵验证：

场景	预期
打开待处理工单	显示转派按钮
打开已关闭工单	不能执行转派
打开转派对话框	只显示其他启用坐席

场景	预期
不填原因提交	被阻止并显示提示
成功转派	列表和详情负责人同步更新
成功后刷新	新负责人和历史仍然存在
服务端拒绝	对话框保留输入并显示错误
重复点击提交	不产生重复历史
转派前后状态	保持不变
转派前后汇总	保持不变

还要回归：

- 新建工单；
- 搜索；
- 筛选；
- 更新状态；
- 关闭工单；
- 查看详情。

关闭工单后的待处理数量 Bug 仍然存在。

本章不能顺手修复。

38.29 更新项目文档

更新 `docs/api-contract.md`：

- 新增坐席查询；
- 新增转派请求；
- 请求和响应；
- 错误情况；
- 操作人来源。

更新 `docs/manual-test-checklist.md`：

- 成功转派；
- 失败边界；
- 事务一致性；
- 刷新持久化；
- 汇总不变；
- 原有功能回归。

更新 `docs/current-state.md`：

- 转派候选实现已经完成；
- 仍位于功能分支；
- 统计 Bug 仍未修复；
- 尚未创建 PR 和合并。

不要把候选状态写成“已上线”。

38.30 审查最终差异

运行：

```
git diff --check
git diff --stat
git status --short
```

然后让 Codex 审查：

```
请审查当前 feature/ticket-reassignment 分支相对 main 的全部未提交差异。

重点检查：
1. 转派规则是否全部由服务端验证；
2. 更新负责人和历史写入是否在同一事务；
3. closed 工单是否被拒绝；
4. 操作人是否来自服务端；
5. 转派是否误改状态和汇总；
6. 前端失败时是否保留输入；
7. 测试是否覆盖成功、拒绝和回滚；
8. 是否修改了第 39 章要处理的 summary-service；
9. 是否出现无关重构、依赖或生成文件；
10. 文档是否把候选功能写成已发布。

先报告问题，不要自动提交或合并。
```

高风险问题需要修正后重新运行全部验证。

38.31 创建聚焦功能提交

案例确认无误后：

```
git add `
  src/db/migrations/003_assignment_history.sql `
  src/repositories/assignment-history-repository.js `
  src/repositories/ticket-repository.js `
  src/services/ticket-service.js `
  src/routes/agents.js `
  src/routes/tickets.js `
  src/app.js `
  .env.example `
  public/index.html `
  public/js/api.js `
  public/js/render.js `
  public/js/app.js `
```

```
public/css/styles.css `
tests/ticket-service.test.js `
docs/api-contract.md `
docs/manual-test-checklist.md `
docs/current-state.md
```

检查暂存差异：

```
git diff --staged --stat
git diff --staged --check
```

创建提交：

```
git commit -m "feat: add auditable ticket reassignment"
```

最后：

```
git status
git log --oneline -2
```

预期：

- 功能分支存在一个聚焦提交；
- 工作区干净；
- 主分支仍未变化；
- 统计 Bug 仍然可以复现；
- 尚未创建 PR。

三、总结

本章案例展示了一个跨前端、后端和数据库的完整候选功能。

案例没有把转派简化为修改一个字段，而是依次完成：

- 创建隔离 Worktree；
- 制定分阶段实施计划；
- 新增转派历史表；
- 建立历史仓储；
- 扩展负责人更新能力；
- 在服务层集中业务规则；
- 使用事务保证数据一致；
- 建立坐席查询和转派 API；
- 保持操作人由服务端提供；
- 增加前端对话框、错误处理和状态更新；

- 测试成功、拒绝和回滚；
- 验证转派不改变工单状态和汇总；
- 回归原有功能；
- 更新 API、测试和当前状态文档；
- 审查差异并形成聚焦提交。

本章掌握检查

- [ ] 能用 Worktree 隔离跨层功能开发；
- [ ] 能把当前状态和历史证据分开保存；
- [ ] 能在服务层集中业务规则；
- [ ] 能使用事务避免半完成更新；
- [ ] 不依赖前端按钮保护服务端规则；
- [ ] 能区分操作人身份和前端输入；
- [ ] 能测试失败后的真实数据库状态；
- [ ] 能控制功能 PR 不顺手修复无关 Bug；
- [ ] 能形成一个职责清晰的功能提交。

四、结束语与引导

到这里，案例中的工单转派已经成为可以审查的候选实现。

但综合实战还没有完成。

第 37 章复现的统计问题仍然存在：

> 工单关闭后，待处理数量没有减少。

下一章不会直接替换可疑状态字符串。

我们将先把错误行为写进回归测试，证明修复前确实失败，再根据业务状态定义完成最小修复。

同时还要保护本章的新功能：

> **转派工单不能改变待处理数量。**

第 39 章的目标是：

> **让一个真实 Bug 从“人工反馈”变成“可重复失败、可验证修复、不会再次回归”的工程证据。**

第 39 章：企业工单系统综合案例，根据证据修复统计 Bug 并补回归测试

一、起始语

第 38 章案例结束时，`feature/ticket-reassignment` 分支已经形成一个聚焦提交：

```
feat: add auditable ticket reassignment
```

本章继续分析同一个固定案例。命令、数据库记录和测试结果用于展示证据链，不要求读者在本地复现完全相同的数据环境。

工单转派候选实现已经具备：

- 完整业务规则；
- 数据库事务；
- 转派历史；
- 前端交互；
- 成功和失败测试；
- 文档说明。

但第 37 章发现的统计问题仍然存在：

> 将一张待处理或处理中工单关闭后，工单状态已经正确变化，首页“待处理工单”数量却没有减少。

上一章刻意没有顺手修复它。

这不是形式上的分工。

它帮助我们保留了清楚的证据链：

- 第一个提交只负责转派功能；
- 第二个提交只负责统计 Bug；
- 每个问题都能单独审查；
- 需要回退时可以区分功能和修复。

本章唯一任务是：

> **观察案例怎样把统计问题转化为能够自动失败的回归测试，确认根因，完成最小修复，并证明转派功能不会影响汇总。**

本章案例不重构报表系统，不改变工单状态定义，也不增加新的统计卡片。

二、主体

> **案例阅读说明**
>
> 本章以下目录、命令、提示词、代码、测试结果和 Git 状态均属于固定案例步骤。没有同类项目时，只需阅读并理解每一步解决的问题、所需证据和停止条件，不要在电脑中照抄执行。将方法迁移到自己的项目时，应替换为真实路径、业务规则、命令和验收标准。

39.1 确认功能分支和提交状态

案例流程继续使用第 38 章的 Worktree：

```
cd Codex 学习/projects/enterprise-ticket-system-reassign
```

检查：

```
git branch --show-current
git status
git log --oneline -3
```

预期：

- 当前分支为 `feature/ticket-reassignment`；
- 工作区干净；
- 最近提交为转派功能；
- 主分支尚未合并。

运行：

```
npm test
```

转派功能测试和原有测试都应通过。

若此时已有失败，先处理候选实现本身，不要继续统计修复。

39.2 重新建立可重复的数据起点

运行：

```
npm run db:reset
npm run dev
```

记录首页汇总和目标工单。

选择一张状态为 `open` 的工单，例如 `TK-1007`。

复现步骤：

289. 打开首页；
290. 记录 `pendingCount`；
291. 打开目标工单；
292. 将状态更新为 `closed`；
293. 记录工单详情；
294. 重新请求汇总 API；
295. 刷新页面；
296. 再次记录汇总。

预期：

```
pendingCount = 原值 - 1
closedCount = 原值 + 1
```

实际：

```
pendingCount = 原值
closedCount = 原值 + 1
```

这说明同一张工单同时被计入：

- 已关闭；
- 待处理。

业务上不成立。

39.3 用三层证据确认问题仍在汇总路径

数据库证据

目标工单：

- `status = closed`；
- `closed_at` 非空；
- `updated_at` 已变化。

单张工单 API

```
Invoke-RestMethod http://localhost:3000/api/tickets/TK-1007
```

返回状态正确。

汇总 API

```
Invoke-RestMethod http://localhost:3000/api/tickets/summary
```

`pendingCount` 错误。

因此可以排除：

- 状态更新失败；
- 数据库未保存；
- 页面只显示旧内存状态。

问题仍然集中在：

> **汇总服务对待处理范围的定义。**

39.4 读取业务规则，不先改字符串

打开：

`docs/product-rules.md`

项目规则定义：

```
待处理工单 = status 为 open 或 processing 的工单
已关闭工单 = status 为 closed 的工单
```

再搜索代码：

```
rg "pendingCount|resolved|processing|closed" src tests docs
```

在 `src/services/summary-service.js` 中，可能发现旧逻辑：

```
const pendingCount = tickets.filter(
  (ticket) => ticket.status !== 'resolved'
).length;
```

当前状态体系已经使用：

- `open`；
- `processing`；
- `closed`。

`resolved` 是旧状态名称。

因此所有工单都满足：

```
ticket.status !== 'resolved'
```

包括 `closed` 工单。

这解释了现象。

但仍然不能立即修改。

先让错误进入测试。

39.5 检查现有测试为什么没有发现问题

打开：

`tests/summary-service.test.js`

现有测试可能只覆盖：

- 返回工单总数；
- 按优先级计数；
- 已关闭数量；
- 空数据库。

它没有覆盖：

> `closed` 工单是否被排除在 `pendingCount` 之外。

这类覆盖缺口很常见。

测试“某个数字存在”，不等于测试“数字的业务口径正确”。

让 Codex 只读审查：

```
请检查 summary-service 的现有测试。

请回答：
1. 哪些业务状态组合已经覆盖；
2. 哪些组合没有覆盖；
3. 哪一条最小测试能够稳定复现“closed 仍计入 pendingCount”；
4. 怎样证明测试失败来自统计口径，而不是示例数据不稳定；
5. 转派功能新增后，还应保护哪条汇总不变量。

不要修改文件。
```

39.6 先写一个修复前失败的测试

在 `tests/summary-service.test.js` 增加：

```
test('closed 工单不计入待处理数量', () => {
  const summary = getTicketSummary([
    { status: 'open', priority: 'high' },
    { status: 'processing', priority: 'medium' },
    { status: 'closed', priority: 'low' }
  ]);

  assert.equal(summary.totalCount, 3);
  assert.equal(summary.pendingCount, 2);
  assert.equal(summary.closedCount, 1);
});
```

如果 `getTicketSummary` 直接读取数据库，则使用项目已有的测试数据库助手写入三种状态。

关键是：

- 输入固定；
- 预期明确；
- 不依赖页面；
- 不依赖随机示例数据；
- 只验证一个业务口径。

运行单个测试：

```
node --test tests/summary-service.test.js
```

预期修复前失败：

```
expected pendingCount: 2
actual pendingCount: 3
```

保存这条失败证据。

如果测试修复前就通过，说明：

- 测试没有走到真实逻辑；
- 输入没有覆盖错误；
- 可疑代码不是当前执行路径；
- 项目状态与假设不一致。

此时应停止重新定位，不能为了符合预期而强行改测试。

39.7 为重新打开补充状态边界测试

一个可靠定义还应保护：

> 工单从 `closed` 重新变为 `open` 后，应重新进入待处理数量。

增加：

```
test('open 和 processing 工单计入待处理数量', () => {
  const summary = getTicketSummary([
    { status: 'open', priority: 'high' },
    { status: 'processing', priority: 'medium' }
  ]);

  assert.equal(summary.pendingCount, 2);
  assert.equal(summary.closedCount, 0);
});
```

这条测试避免修复被写成：

```
status !== 'closed'
```

却忽略未来可能出现的未知状态。

从业务定义出发，更清楚的表达是：

```
只有 open 和 processing 属于待处理范围。
```

39.8 保护转派功能的汇总不变量

第 38 章新增转派功能后，还需要保护：

> 改变负责人不会改变状态汇总。

可以增加服务级测试：

```
test('转派工单不改变状态汇总', () => {
  const before = getSummaryFromDatabase(db);

  reassignTicket({
    db,
    ticketId: 'TK-1001',
    toAssigneeId: 'agent-03',
    reason: '需要支付接口支持人员继续处理',
    operatorName: '演示主管',
    now: () => '2026-07-31T10:00:00.000Z'
  });

  const after = getSummaryFromDatabase(db);

  assert.deepEqual(after, before);
});
```

这条测试连接了第 38 章和第 39 章：

- 转派可以修改负责人；
- 汇总只由状态决定；
- 两套能力互不干扰。

39.9 选择最小且语义明确的修复

可疑旧逻辑是：

```
status !== 'resolved'
```

可以有三种修法。

方案一：替换为 *`status !== 'closed'`*

优点：简单。

风险：未来出现未知状态时，也会被自动计入待处理。

方案二：明确列出活动状态

```
const ACTIVE_TICKET_STATUSES = new Set(['open', 'processing']);

const pendingCount = tickets.filter((ticket) =>
  ACTIVE_TICKET_STATUSES.has(ticket.status)
).length;
```

优点：与业务定义一致，未知状态不会默认为待处理。

方案三：重写为复杂 SQL 报表层

不适合当前任务。

本章选择方案二。

它不是为了“代码更高级”，而是为了让实现直接表达业务规则。

39.10 避免重复定义状态常量

先检查项目是否已经存在统一状态常量。

如果 `ticket-service.js` 或其他文件已经定义：

```
export const TICKET_STATUSES = {
  OPEN: 'open',
  PROCESSING: 'processing',
  CLOSED: 'closed'
};
```

汇总服务应复用，而不是再写一份字符串。

如果项目当前没有统一常量，本章不要为了一个 Bug 进行大范围状态重构。

可以在 `summary-service.js` 中使用局部常量，并将统一状态模块列为后续改进。

选择取决于真实项目结构。

39.11 实施修复

示例：

```
const PENDING_STATUSES = new Set(['open', 'processing']);

export function getTicketSummary(tickets) {
  return {
    totalCount: tickets.length,
    pendingCount: tickets.filter((ticket) =>
      PENDING_STATUSES.has(ticket.status)
    ).length,
    processingCount: tickets.filter(
      (ticket) => ticket.status === 'processing'
    ).length,
```

```
    closedCount: tickets.filter(
      (ticket) => ticket.status === 'closed'
    ).length
  };
}
```

只修改汇总口径。

不要顺便：

- 重命名全部字段；
- 把数组过滤改为数据库聚合；
- 增加缓存；
- 重写前端卡片；
- 增加趋势报表；
- 修改状态更新 API。

39.12 先运行失败测试

运行：

```
node --test tests/summary-service.test.js
```

预期：

- `closed` 不计入待处理测试通过；
- `open` 和 `processing` 计入待处理测试通过；
- 现有汇总测试通过。

随后运行转派相关测试：

```
node --test tests/ticket-service.test.js
```

确认汇总修复没有破坏转派。

39.13 运行全部自动化测试

执行：

```
npm test
```

记录：

- 总测试数；
- 通过数；
- 失败数；
- 退出状态；

- 新增测试数；
- Git 状态。

若项目提供检查命令：

```
npm run check
```

不能只运行新增测试后就宣布完成。

综合项目需要验证全部已有行为。

39.14 重新执行页面复现步骤

运行：

```
npm run db:reset
npm run dev
```

使用与第 37 章相同的步骤。

修复后预期：

297. 关闭一张 `open` 工单；
298. 工单状态变为 `closed`；
299. `closed_at` 被写入；
300. 待处理数量减少 1；
301. 已关闭数量增加 1；
302. 刷新后结果保持；
303. API 与页面一致。

使用相同条件复测，才能比较修复前后。

39.15 验证重新打开

若系统支持将 `closed` 工单重新设为 `open`，继续测试：

304. 记录当前汇总；
305. 将已关闭工单重新打开；
306. 检查待处理数量增加 1；
307. 已关闭数量减少 1；
308. 刷新后保持。

如果系统不支持重新打开，不要临时新增该功能。

只保留服务层状态组合测试即可。

39.16 验证转派不改变汇总

选择一张 `open` 工单：

309. 记录汇总；
310. 转派给另一名启用坐席；
311. 检查负责人变化；
312. 检查状态不变；
313. 检查汇总完全不变；
314. 刷新后再次确认。

再选择一张 `processing` 工单重复。

这个场景很重要。

如果前端成功转派后错误地刷新或改写汇总，本章应及时发现。

39.17 检查未知状态的处理

真实系统可能因数据迁移或旧版本出现未知状态。

使用明确活动状态集合后，未知状态不会进入 `pendingCount`。

但系统还应怎样处理未知状态？

本章不扩展状态治理，只记录：

- 未知状态不计入待处理；
- 是否计入总数取决于现有定义；
- 日志或数据校验属于后续治理；
- 不在本次修复中静默改写数据。

39.18 更新测试清单和当前状态

更新 `docs/manual-test-checklist.md`：

新增：

- 关闭 `open` 工单后待处理减 1；
- 关闭 `processing` 工单后待处理减 1；
- 已关闭数量增 1；
- 刷新后保持；
- 转派不改变汇总；
- 未知状态不默认为待处理。

更新 `docs/current-state.md`：

- 统计 Bug 已经在候选分支修复；
- 新增回归测试；
- 转派功能和统计修复均未合并；
- 下一步是最终 PR 和业务验收。

不要写“生产问题已解决”。

当前仍然只是候选分支。

39.19 审查修复范围

运行：

```
git diff --stat HEAD
git diff --check
git status --short
```

让 Codex 审查：

```
请只审查当前未提交的统计修复差异。

重点检查：
1. 新测试是否在修复前能够失败；
2. pendingCount 是否只包含 open 和 processing；
3. closedCount 是否仍只包含 closed；
4. totalCount 是否保持原定义；
5. 转派是否不会改变汇总；
6. 是否出现与报表无关的重构；
7. 是否改变 API 字段或状态名称；
8. 文档是否准确描述候选状态。

不要自动提交、推送或合并。
```

如果审查建议重构整个汇总模块，应判断是否属于当前 Bug 的必要条件。

多数情况下，应留到后续任务。

39.20 创建独立修复提交

暂存明确文件：

```
git add `
  src/services/summary-service.js `
  tests/summary-service.test.js `
  tests/ticket-service.test.js `
  docs/manual-test-checklist.md `
  docs/current-state.md
```

如果转派不变量测试放在其他文件，应按实际路径暂存。

检查：

```
git diff --staged --stat
git diff --staged --check
```

创建提交：

```
git commit -m "fix: exclude closed tickets from pending summary"
```

检查：

```
git log --oneline -4
git status
```

预期候选分支包含两个聚焦提交：

```
fix: exclude closed tickets from pending summary
feat: add auditable ticket reassignment
```

工作区干净。

39.21 对候选分支做完整回归

案例在进入第 40 章前，再检查：

```
npm test
npm run check
git diff main...HEAD --stat
git status
```

然后完成浏览器回归：

- 新建工单；
- 搜索和筛选；
- 打开详情；
- 更新状态；
- 关闭工单；
- 观察汇总；
- 转派工单；
- 查看转派历史；
- 刷新页面；
- 验证数据持久化。

候选分支现在可以进入最终交付，但仍未被业务批准。

三、总结

本章案例把一个人工反馈的问题转化成了可追踪的工程修复。

案例依次完成：

- 在同一数据起点重新复现问题；
- 用数据库、单张工单 API 和汇总 API 建立证据链；
- 读取业务规则；
- 找到旧状态名称造成的疑似根因；
- 检查现有测试覆盖缺口；
- 先编写修复前失败的回归测试；
- 明确 `open` 和 `processing` 属于待处理范围；
- 使用最小且语义清楚的状态集合修复；
- 保护转派不改变汇总；
- 运行单元测试、全部测试和浏览器回归；
- 更新文档；
- 创建独立修复提交；
- 保持候选分支工作区干净。

本章掌握检查

- [ ] 能用固定步骤稳定复现 Bug；
- [ ] 能区分页面现象、API 结果和数据库事实；
- [ ] 不会看到可疑字符串后直接修改；
- [ ] 能找到测试覆盖缺口；
- [ ] 能先证明回归测试在修复前失败；
- [ ] 修复表达业务定义，而不只是替换偶然字符串；
- [ ] 能保护新功能和旧功能之间的不变量；
- [ ] 能将功能和 Bug 修复拆成聚焦提交；
- [ ] 能说明为什么案例在本章结束时应保持候选分支完整、干净、可审查。

四、结束语与引导

到这里，案例中的代码工作已经基本完成。

候选分支中包含：

```
一个工单转派功能提交
+
一个统计 Bug 修复提交
```

但“测试通过”和“代码存在”仍然不等于正式交付。

第 40 章还需要完成：

- 最终差异审查；
- Pull Request 说明；
- 确定性 CI；
- Codex Review；
- 人工技术审查；
- 业务验收；
- 合并和主分支回归；
- 数据迁移与回退说明；
- Worktree 和分支清理；
- 当前状态和后续事项交接。

最后一章的核心问题是：

> **怎样证明这组修改已经具备被团队接收的条件，并让下一名维护者知道系统现在真实处于什么状态。**

第 40 章：企业工单系统综合案例，通过 PR、质量链和业务验收完成正式交付

一、起始语

第 39 章案例结束时，`feature/ticket-reassignment` 分支已经包含两个聚焦提交：

```
feat: add auditable ticket reassignment
fix: exclude closed tickets from pending summary
```

本章继续沿用固定案例，完整拆解候选成果怎样进入 PR、CI、模型审查、人工审查、业务验收、合并和交接。读者不需要创建真实 PR，但应理解每一道门分别证明什么。

候选实现已经完成：

- 工单转派；
- 转派历史；
- 服务端业务验证；
- 数据库事务；
- 前端交互；
- 统计 Bug 修复；
- 回归测试；
- 文档更新；
- 本地自动化测试；
- 浏览器手工回归。

但候选分支仍然不是正式成果。

一个企业项目能否交付，不能只看：

- Codex 是否报告完成；
- 页面是否能够点击；
- 测试是否全部通过；
- 开发者是否认为没有问题。

正式交付还需要回答：

- 变更范围是否准确；
- CI 是否通过；
- 模型审查发现了什么；
- 人工技术审查是否批准；

- 业务方是否按真实场景验收；
- 数据迁移是否清楚；
- 失败后怎样回退；
- 合并后主分支是否仍然可运行；
- 哪些能力仍然没有实现；
- 下一名维护者从哪里继续。

本章唯一任务是：

> **观察案例怎样把第 37—39 章的候选成果整理成一个可审查、可验收、可合并、可交接的正式交付。**

本章不会让 Codex 自动批准、自动合并或自动发布，也不要求读者实际操作同名仓库。

二、主体

> **案例阅读说明**
>
> 本章以下目录、命令、提示词、代码、测试结果和 Git 状态均属于固定案例步骤。没有同类项目时，只需阅读并理解每一步解决的问题、所需证据和停止条件，不要在电脑中照抄执行。将方法迁移到自己的项目时，应替换为真实路径、业务规则、命令和验收标准。

40.1 再次确认候选分支状态

案例流程首先进入 Worktree：

```
cd Codex 学习/projects/enterprise-ticket-system-reassign
```

检查：

```
git branch --show-current
git status
git log --oneline --decorate -5
git diff main...HEAD --stat
```

预期：

- 当前分支为 `feature/ticket-reassignment`；
- 工作区干净；
- 相对 `main` 只有本批次计划内变化；
- 两个提交职责清楚；
- 没有数据库文件、日志、截图、密钥和临时输出。

如果工作区仍有未提交修改，不能直接创建 PR。

40.2 重新运行最终本地质量门

按项目规则运行：

```
npm ci
npm test
npm run check
```

如果项目提供数据库集成测试，再运行：

```
npm run test:integration
```

记录：

- Node.js 版本；
- 安装命令；
- 测试总数；
- 通过数；
- 失败数；
- 检查命令；
- 退出状态；
- 执行时间；
- Git 状态。

正式证据可以写入 PR，不需要把完整终端日志全部提交到仓库。

40.3 核对迁移是否可重复执行

新功能新增：

`003_assignment_history.sql`

需要验证：

315. 从空数据库执行全部迁移成功；
316. 从现有 `002` 版本升级到 `003` 成功；
317. 表和索引存在；
318. 旧工单数据未变化；
319. 原有功能仍然可以使用；
320. 重复运行迁移工具不会重复创建或损坏结构。

训练项目可以使用两个独立本地数据库验证：

```
fresh-install.db
upgrade-from-v1.db
```

两者都应位于被忽略的临时目录。

不要把测试数据库提交到 Git。

40.4 检查数据回退边界

代码回退和数据库回退不是同一件事。

如果功能已经产生转派历史，再回退代码：

- 新表仍然可能存在；
- 历史数据仍然存在；
- 旧代码可能忽略新表；
- 删除表会丢失审计记录。

因此，本次建议的回退策略是：

> **代码可以回退到上一稳定提交，但不自动删除转派历史表和已产生的历史数据。**

若迁移导致系统无法启动，应由明确负责人评估：

- 修复前向迁移；
- 暂停新功能入口；
- 恢复应用代码；
- 保留数据库备份；
- 不在未确认影响时执行 `DROP TABLE`。

案例项目没有生产部署，因此只记录策略，不执行真实回退。

40.5 准备发布说明

新增：

`docs/release-notes-v1.1.md`

建议内容：

```
# v1.1 候选版本说明

## 新增
- open 和 processing 工单支持转派；
- 保存原负责人、新负责人、原因、操作人和时间；
- 工单详情显示最近转派历史。

## 修复
- closed 工单不再计入待处理数量。

## 数据变化
- 新增 ticket_assignment_history 表和索引；
- 原有 tickets 数据不迁移、不重写。

## 未包含
```

```
- 登录与细粒度权限；
- 跨团队审批；
- 自动通知；
- 批量转派；
- 自动分单；
- 生产发布自动化。

## 回退原则
- 可以回退应用代码；
- 不自动删除已产生的转派历史；
- 数据库处理需由负责人确认。
```

发布说明应描述事实。

不要使用：

- “完全安全”；
- “不存在任何风险”；
- “已适用于所有企业”；
- “已正式上线”。

40.6 更新当前状态文件

在创建 PR 前，更新 `docs/current-state.md` 为候选交付状态：

- 当前候选分支；
- 两个提交；
- 新增功能；
- 已修 Bug；
- 测试结果；
- 数据迁移；
- 已知限制；
- 尚未完成的审批；
- 下一步为 PR、CI 和业务验收。

这份文件在合并前仍应明确：

> 尚未进入 `main`，尚未发布。

更新后创建一个文档提交，还是修改上一提交？

因为功能和 Bug 提交已经形成清晰历史，本章可以新增第三个聚焦提交：

```
docs: prepare v1.1 delivery and handoff
```

这比重写已有提交更容易审查。

40.7 创建交付文档提交

暂存：

```
git add docs/release-notes-v1.1.md docs/current-state.md
```

检查：

```
git diff --staged --check
git diff --staged
```

提交：

```
git commit -m "docs: prepare v1.1 delivery and handoff"
```

候选分支现在包含：

```
docs: prepare v1.1 delivery and handoff
fix: exclude closed tickets from pending summary
feat: add auditable ticket reassignment
```

40.8 推送候选分支前检查外部副作用

推送属于外部写入。

在执行前确认：

- 仓库地址正确；
- 当前账号正确；
- 分支名称正确；
- 没有敏感文件；
- 组织允许创建远程分支；
- 推送不会触发生产发布；
- CI 所需权限已经明确。

检查远程：

```
git remote -v
git status
git log --oneline main..HEAD
```

案例经人工确认后：

```
git push -u origin feature/ticket-reassignment
```

Codex 可以准备命令和解释影响，但推送前的授权仍由人完成。

40.9 编写清楚的 PR 标题

建议标题：

```
feat: add ticket reassignment and correct pending summary
```

标题同时包含功能和修复，因为本次交付范围本来就包含两项已确认业务任务。

PR 内部仍然使用两个独立提交保持可审查性。

如果团队要求一个 PR 只包含一种变更，也可以拆成两个 PR。

本书采用一个综合交付 PR，是为了让第 37—40 章形成统一业务验收。

40.10 编写 PR 说明

PR 说明应让未参与开发的人理解：

- 为什么改；
- 改了什么；
- 没改什么；
- 怎样验证；
- 数据有什么变化；
- 有什么风险；
- 谁需要批准。

示例：

```
## 背景

客服主管需要将未关闭工单转派给其他启用坐席，并保留可审计历史。
同时，现有系统在关闭工单后仍将其计入待处理数量。

## 变更

- 新增工单转派 API 和前端交互；
- 新增转派历史表；
- 服务端验证状态、坐席、原因和同一负责人；
- 使用事务同时更新负责人和写入历史；
- 修复 pendingCount 状态口径；
- 新增成功、拒绝、回滚和汇总回归测试；
- 更新 API、测试清单、当前状态和发布说明。

## 非目标

- 登录和角色权限；
- 跨团队审批；
- 自动通知；
- 批量转派；
- 自动分单；
- 生产发布。
```

```
## 验证

- npm test;
- npm run check;
- 全新数据库迁移;
- 现有数据库升级;
- 浏览器转派成功和失败场景;
- 关闭工单后的汇总回归;
- 转派前后汇总不变。

## 数据变化

新增 ticket_assignment_history 表和索引，不重写已有工单。

## 回退

应用代码可以回退；不自动删除转派历史表和历史数据。

## 需要审查

- 技术审查;
- 业务验收;
- 合并批准。
```

40.11 检查 PR 差异，而不只看提交信息

创建 PR 后，查看：

- Files changed；
- Commits；
- Checks；
- Conversation。

重点检查：

- 是否包含计划外文件；
- 迁移文件是否只创建新表和索引；
- 服务层是否真正使用事务；
- 前端是否传入伪造操作人；
- 是否误改状态定义；
- 测试是否对应业务规则；
- 文档是否准确。

提交信息清楚，不代表差异一定正确。

40.12 让确定性 CI 先运行

CI 应优先完成可以明确判断的检查：

- 安装依赖；
- 自动化测试；

- 语法或格式检查；
- 迁移测试；
- 禁止提交数据库和密钥；
- 必要的文件结构检查。

CI 输出应能明确区分：

- 通过；
- 失败；
- 取消；
- 未运行。

如果 CI 失败，停止后续合并。

不要因为本地测试通过就忽略远程环境失败。

40.13 处理 CI 失败

常见失败包括：

- Node.js 版本不同；
- 锁文件和依赖不一致；
- 测试依赖本地数据库残留；
- 路径大小写在不同系统中不一致；
- 迁移不能从空数据库执行；
- 环境变量缺少默认测试值；
- 生成文件意外进入 Git。

处理流程：

```
读取失败步骤
→ 在本地复现
→ 找到最小原因
→ 修改候选分支
→ 重新运行全部本地检查
→ 创建聚焦修复提交
→ 推送
→ 等待 CI 重新运行
```

不能直接在 CI 配置中跳过失败测试。

40.14 使用 Codex Review 获取第二类信号

在 PR 中触发或查看 Codex Review。

模型审查适合关注：

- 事务是否完整；

- 错误路径是否遗漏；
- 前后端契约是否一致；
- 状态和汇总是否可能冲突；
- 迁移是否带来数据风险；
- 测试是否遗漏高影响边界；
- 是否违反 `AGENTS.md`。

OpenAI 在 Codex 官方使用案例中将 Pull Request 和本地差异审查列为典型质量场景之一。模型审查提供的是审查信号，不能替代确定性测试和人工批准。OpenAI Developers：Codex use cases

40.15 逐条处理模型审查意见

Codex Review 可能提出：

- 高风险问题；
- 中等风险问题；
- 建议改进；
- 不确定项。

每条意见需要人工判断：

情况	处理
真实缺陷	修复并补测试
规则遗漏	修复或补文档
与项目事实不符	回复证据并标记不采纳
超出本次范围	记录后续事项
无法确认	请业务或维护者判断

不能因为意见来自模型就自动采纳。

也不能因为 CI 通过就全部忽略。

40.16 完成人工技术审查

人工技术审查至少检查：

数据层

- 迁移安全；
- 外键和索引；
- 事务边界；
- 历史记录可追溯；

- 回退原则。

服务层

- 状态验证；
- 坐席验证；
- 原因验证；
- 操作人来源；
- 错误类型；
- 原子更新。

前端

- 允许状态；
- 重复提交保护；
- 错误反馈；
- 成功后的列表和详情一致；
- 刷新后持久化；
- 可访问性基础。

测试

- 成功；
- 拒绝；
- 事务回滚；
- 统计回归；
- 转派不改变汇总；
- 原有功能回归。

人工审查者应能指出：

- 已检查什么；
- 有什么问题；
- 哪些问题已经解决；
- 是否批准合并。

40.17 组织业务验收

技术正确不等于业务可用。

业务验收由了解工单流程的人完成。

建议使用一个重置后的演示数据库，并提供固定场景。

验收角色

- 客服主管；
- 工单处理人员；
- 项目负责人。

验收矩阵

编号	场景	预期
UAT-01	转派待处理工单	新负责人和历史正确
UAT-02	转派处理中工单	状态保持处理中
UAT-03	转派已关闭工单	被拒绝并说明原因
UAT-04	转给当前负责人	被拒绝
UAT-05	不填写原因	被拒绝
UAT-06	转给停用坐席	不可选择或被拒绝
UAT-07	刷新页面	新负责人和历史仍存在
UAT-08	关闭待处理工单	待处理减 1，已关闭增 1
UAT-09	转派工单	汇总数字不变化
UAT-10	原有搜索、筛选和状态更新	正常

业务验收需要记录：

- 通过；
- 不通过；
- 有条件通过；
- 新需求。

新需求不能在验收现场自动进入当前 PR。

40.18 区分缺陷和新需求

业务方可能提出：

- 转派后发送通知；
- 显示坐席当前负载；
- 支持批量转派；
- 跨团队审批；
- 允许已关闭工单转派；
- 增加转派原因模板。

其中有些是有价值的新需求，但不属于当前验收缺陷。

判断方法：

> 当前行为是否违反第 37 章已经确认的业务规则和验收标准？

- 违反：缺陷，应在合并前修复；
- 不违反：新需求，进入后续计划。

避免在最后一刻让 PR 无限扩张。

40.19 处理业务验收失败

若 UAT 失败：

321. 记录具体工单和操作步骤；
322. 保存页面、API 和数据库证据；
323. 判断是实现缺陷、环境问题还是需求歧义；
324. 在候选分支修复；
325. 补自动化测试；
326. 重新运行 CI 和审查；
327. 重新执行受影响 UAT；
328. 不绕过失败直接批准合并。

业务验收不能只写“主管试过，没问题”。

40.20 形成最终批准条件

合并前必须同时满足：

- [ ] PR 范围与计划一致；
- [ ] 工作区干净；
- [ ] CI 全部通过；
- [ ] Codex Review 高风险问题已处理；
- [ ] 人工技术审查批准；
- [ ] 业务验收通过；
- [ ] 数据迁移验证完成；
- [ ] 回退原则明确；
- [ ] 已知限制进入文档；
- [ ] 没有密钥和真实客户数据；
- [ ] 合并人具有明确授权；
- [ ] 合并不会自动触发未批准发布。

缺少任何一项，都应暂停。

40.21 合并 PR

由授权人员选择团队允许的合并方式。

如果保留三个聚焦提交有利于追溯，可以使用普通合并。

如果团队统一使用 Squash，也应确保最终提交信息完整表达：

- 转派功能；
- 统计修复；
- 数据迁移；
- 测试和文档。

合并动作必须由人确认。

40.22 同步本地主分支

案例流程回到原项目目录：

```
cd ../enterprise-ticket-system
```

检查主目录干净：

```
git status
```

拉取远程主分支：

```
git switch main
git pull --ff-only
```

检查：

```
git log --oneline -5
git status
```

预期：

- `main` 包含已合并成果；
- 本地与远程一致；
- 工作区干净。

40.23 在主分支重新运行测试

合并后不能只相信 PR 检查。

在主目录运行：

```
npm ci
npm test
npm run check
```

然后：

```
npm run db:reset
npm run dev
```

完成最小冒烟测试：

329. 首页打开；
330. 工单列表加载；
331. 转派一张 `open` 工单；
332. 查看转派历史；
333. 关闭一张待处理工单；
334. 检查汇总；
335. 刷新页面；
336. 检查数据保持。

这证明合并后的主分支仍然成立。

40.24 通过小型文档 PR 更新合并后状态

合并后，`docs/current-state.md` 仍可能写着“候选分支”。这份状态已经过时，不能作为最终交接。

从最新 `main` 创建：

```
git switch -c docs/v1.1-post-merge-state
```

只更新：

- 已合并 PR 和提交；
- 主分支测试结果；
- 当前功能状态；
- 仍未发布到生产；
- 已知限制；
- 下一轮建议。

检查差异后创建：

```
git add docs/current-state.md
git commit -m "docs: record v1.1 merged state"
git push -u origin docs/v1.1-post-merge-state
```

通过一个小型文档 PR 完成审查和合并，避免直接推送受保护主分支。随后同步本地 `main` 并确认文档 PR 也已经进入主分支。

若这一步尚未完成，项目只能称为“代码已合并，交接状态待更新”，不能宣布最终交付闭环完成。

40.25 清理 Worktree 和远程分支

确认主分支测试通过、没有未保存内容后：

```
git worktree list
```

移除功能 Worktree：

```
git worktree remove ../enterprise-ticket-system-reassign
```

删除本地分支：

```
git branch -d feature/ticket-reassignment
git branch -d docs/v1.1-post-merge-state
```

若团队规则允许，删除远程分支：

```
git push origin --delete feature/ticket-reassignment
git push origin --delete docs/v1.1-post-merge-state
```

删除前必须确认两个 PR 都已经合并，分支不再承担恢复用途。

40.26 建立最终交接内容

交接至少包括：

当前能力

- 未关闭工单可以转派；
- 转派保留历史；
- 关闭工单后汇总正确；
- 转派不改变状态汇总。

数据结构

- 新增 `ticket_assignment_history`；
- 旧工单表不重写；
- 历史记录不自动删除。

运行和测试

- 安装命令；
- 数据库初始化；
- 启动命令；
- 测试命令；

- CI 位置。

已知限制

- 没有登录和角色权限；
- 操作人为演示配置；
- 没有通知；
- 没有跨团队审批；
- 没有批量转派；
- 没有生产自动发布。

后续建议

337. 接入真实身份；
338. 增加转派权限；
339. 增加通知；
340. 评估坐席负载；
341. 增加数据库备份和迁移演练；
342. 增加端到端测试；
343. 建立生产监控。

交接不是把所有未来功能立即开发出来。

它让下一轮知道从哪里继续。

40.27 保存最终证据

最终证据包括：

- PR 链接；
- 合并提交；
- CI 结果；
- Codex Review 处理记录；
- 人工技术批准；
- UAT 记录；
- 迁移验证；
- 主分支测试；
- 当前状态文档；
- 已知限制和后续事项。

证据应放在团队允许的位置。

不要把含有：

- 访问令牌；
- 内部敏感链接；
- 真实客户数据；
- 私有日志；
- 数据库副本；

的材料公开提交。

40.28 完成最终仓库检查

案例在主项目中检查：

```
git status
git branch --show-current
git log --oneline -5
git worktree list
```

预期：

- 当前分支为 `main`；
- 工作区干净；
- 主分支包含正式成果；
- 临时 Worktree 已经移除；
- 功能分支已经按规则清理；
- 本地示例数据库仍然被忽略；
- 没有 SDK 实验、非交互报告或截图进入仓库。

40.29 回看第 37—40 章的完整证据链

这四章形成了：

```
第 37 章
运行、探索、建立基线、复现问题、冻结需求

第 38 章
隔离开发转派功能、测试事务和失败边界、形成候选提交

第 39 章
先写失败测试、修复统计口径、保护转派不变量、形成修复提交

第 40 章
PR、CI、模型审查、人工审查、业务验收、合并、回归和交接
```

每一章都继承上一章的真实状态。

没有临时更换项目，也没有为了讲高级工具增加无关功能。

40.30 理解“Codex 完成”和“团队完成”的区别

Codex 可以帮助：

- 探索项目；
- 制定计划；
- 修改代码；
- 运行测试；
- 审查差异；
- 整理 PR；
- 生成交接文档。

但团队仍然负责：

- 确认业务规则；
- 授予权限；
- 保护客户数据；
- 判断模型意见；
- 执行业务验收；
- 批准合并；
- 决定发布；
- 承担运行责任。

这也是本书最后需要建立的认识：

> **Codex 能够显著扩大个人和团队推进技术工作的能力，但可信交付来自清楚的边界、可检查的证据和明确的人类责任。**

三、总结

本章通过企业工单系统案例展示了正式交付闭环。

案例依次完成：

- 确认候选分支和提交；
- 重新运行本地质量门；
- 验证全新安装和增量迁移；
- 区分代码回退和数据回退；
- 准备发布说明和当前状态；
- 创建交付文档提交；

- 人工确认后推送分支；
- 编写 PR 标题和说明；
- 审查全部差异；
- 运行确定性 CI；
- 处理 Codex Review；
- 完成人工技术审查；
- 组织业务验收；
- 区分缺陷和新需求；
- 建立合并批准条件；
- 合并 PR；
- 同步并验证本地主分支；
- 更新合并后状态；
- 清理 Worktree 和分支；
- 形成最终交接和证据包。

本章掌握检查

- [ ] 能区分候选实现、已合并和已发布；
- [ ] 能为数据库迁移建立验证和回退原则；
- [ ] 能编写可审查的 PR 说明；
- [ ] 能区分 CI、Codex Review、人工技术审查和业务验收；
- [ ] 不让模型自动批准、合并或发布；
- [ ] 能处理 CI 和 UAT 失败；
- [ ] 能区分缺陷和新需求；
- [ ] 合并后会重新验证主分支；
- [ ] 会清理临时分支和 Worktree；
- [ ] 能形成当前状态、已知限制和后续事项明确的交接。

四、结束语与引导

至此，本书 40 章的主线已经完成：

```
认识 Codex
→ 安装和第一次成果
→ 文件、终端、项目和 Git
→ 任务、计划、修改和验证
→ 从零完成个人项目
→ 接手并修改团队项目
→ 建立项目规则和跨入口协作
→ 使用 Skill、MCP、子智能体和自动化
```

```
→ 进入非交互、SDK、GitHub 质量链和安全治理
→ 拆解企业工单系统正式交付
```

你不需要因为学完一本手册，就立即使用所有高级能力。

更重要的是建立一套稳定工作方式：

```
先确认目标和边界
→ 再读取真实项目
→ 让计划可以审查
→ 在隔离环境中修改
→ 用测试和运行结果提供证据
→ 由人决定高风险动作
→ 将结果交付给下一名维护者
```

当项目变大、任务变长、工具变多时，这套工作方式仍然成立。

本书最终希望帮助你获得的，不只是“让 AI 写代码”的能力。

而是：

> **在自己并不熟悉全部代码的情况下，仍然能够借助 Codex 有边界地理解项目、推进修改、验证结果，并把成果交付得清楚、可信、可继续。**