THE COMPLETE GUIDE TO AI CODING


# AI 编程实战

## 三卷书

卷一 · 入门 | 卷二 · 工作流 | 卷三 · 架构 | 别册 · 速查

「AI 不止语」

jnMetaCode · aibuzhiyu.com

v1.0 · 2026 / 05 · CC BY-NC-SA 4.0

# 总序

> 这套书的最大野心：让你看完之后，再也不用问"这个任务该用哪个 AI 工具"。

## 这套书是什么

这不是 9 本工具说明书，也不是一篇又一篇博客拼起来的合订本。

它是把 ai-coding-guide 仓库里两年沉淀的 **10 款主流 AI 编程工具教程 + 7 套通用方法论 + 多个端到端实战脚本**，按 **读者画像** 而不是按工具，重新编排成的渐进式读物。

工具会换代，方法论永生。所以这套书的结构是：**先教你方法，再用工具佐证**。

## 三卷分别给谁

| 卷 | 适合谁 | 看完能做什么 |
| --- | --- | --- |
| **卷一 · 入门** | AI 编程零基础 / 只用过 Tab 补全 | 从命令行 Agent 到 IDE 集成，4 款主流工具任选其一独立完成日常任务 |
| **卷二 · 工作流** | 已经会用 1-2 款工具，想把效率往上拔 | 多工具组合、Skill / Hook / Subagent 编排、典型场景的端到端剧本 |
| **卷三 · 架构** | 团队 lead / 平台工程师 / CTO | 把 AI 编程接进 CI、安全策略、企业治理；MCP 生态选型 |

**别册** 是一页 `cheatsheet.md`——所有工具关键命令、配置文件、组合方案的横向速查。案头放一份，每天翻两眼。

## 怎么读最高效

- **顺读**：按卷一→卷二→卷三的顺序——是为完全新手设计的
- **挑章读**：每章互相独立，目录里看到顺眼的就跳进去——卷二/卷三尤其如此

- **遇到工具不熟**：随时退回卷一对应章节补课
- **别册当字典**：写代码时遇到"那个命令叫啥来着"，直接搜别册

## 信息源与可信度

每条工具的 CLI flag、配置字段、子命令、模型名都对照过相应仓库的源码（特别是 Codex CLI 那章，6 轮源码核实）。社区博客里常见的过期信息（如 `--full-auto`、`bubblewrap` 等）已修正——任何被否决的"常识"都附了源码出处链接。

工具迭代快，每一版的"信息核实截止日期"在对应章节末尾标注。如果你看的版本距今超过 6 个月，建议先扫一眼 项目更新日志 看有没有 breaking change。

## 上游仓库

GitHub: jnMetaCode/ai-coding-guide — 欢迎 issue 和 PR。

# 卷一 · 入门：AI 编程从零开始

## 这一卷是给谁看的

- 用过 GitHub Copilot 的 Tab 补全，但没真正让 AI 替你写过整个函数
- 听说 Claude Code、Cursor 很厉害，但打开后不知道第一句话该说啥
- 公司刚发了订阅，不想错过但也不想白学一堆短期产品

**没有这一卷你会怎样？** 直接跳进 Claude Code 装好会手忙脚乱——不知道 CLAUDE.md 是干啥的、不知道 Plan 模式什么时候触发、不知道为什么它有时候答案那么聪明有时候那么蠢。这一卷就是把这些"为什么"在你撞墙之前先讲透。

## 这一卷的逻辑顺序

```
速查表（30 秒选工具）
   ↓
提示词工程（3 个工具任选都通用的"输入功"）
   ↓
需求拆解（任务给 AI 之前先在脑子里切好块）
   ↓
上下文管理（防止对话越长越笨）
   ↓
4 款工具速通：Cursor / Copilot / Claude Code / Codex CLI
（IDE 类两个 + CLI 类两个，覆盖 80% 用户场景）
```

学完这一卷你应该能：

- 拿到一个新任务，30 秒内决定用哪个工具
- 写一段提示词，准确表达 Goal / Context / Constraints / Done-when
- 任选 4 款工具中的 1 款，独立完成一次跨多文件的真实修改
- 知道什么时候该停下来重启对话，而不是硬着头皮往坏对话里灌信息

## 工具选择的底线

如果你只想学 1 款工具：**Claude Code**（Agent 能力最强）或 **Cursor**（IDE 体验最好），看你更喜欢命令行还是图形界面。

如果你已经订阅 ChatGPT Plus/Pro：直接读 **Codex CLI** 那章——白送的额度不用白不用。

如果完全没预算：先 **Cursor 免费版** + 学好提示词工程章节就够。

> 接下来按目录顺序往下走。如果某一章你已经熟，可以跳——但**提示词工程**那章就算资深用户也建议刷一遍，里面"四元素提示词模板"贯穿全套书。

# 10 款工具速查表（Cheatsheet）

> 一页看完所有工具的关键参数、命令、快捷键。模型和定价变化快，具体以各官网为准。
>
> 信息截止：2026-04

---

## 一、按问题选工具（30 秒决策）

| 你的诉求 | 首选 | 备选 |
| --- | --- | --- |
| 在 IDE 里按 Tab 补全 | **Cursor** | Copilot / Windsurf / Trae |
| 终端里跑 Agent 做复杂任务 | **Claude Code** | **Codex CLI** / Aider / Gemini CLI |
| 已订阅 ChatGPT，想顺手用 | **Codex CLI** | Cursor 接 GPT |
| 超大代码库一次性分析 | **Gemini CLI**（2M 上下文） | Aider + Map 模式 |
| 预算紧 / 零成本 | **Trae**（免费）或 **Gemini CLI**（免费额度） | Aider + 本地模型；或 `codex --oss --local-provider ollama` |
| 国内直连，不用 VPN | **Trae** | OpenClaw + 本地模型 |
| 团队协作，规格驱动 | **Kiro**（Spec） | Claude Code + plan mode |
| Git 原生，多模型切换 | **Aider** | — |
| AI Agent 自动化（非纯编程） | **OpenClaw** | — |

| 你的诉求 | 首选 | 备选 |
| --- | --- | --- |
| 只有 VS Code、不想装新东西 | Copilot | Cursor/Windsurf/Trae 都是 VS Code 分叉 |

---

## 二、能力矩阵

| 维度 | Claude Code | Codex CLI | Cursor | Copilot | Windsurf | Gem CL |
| --- | --- | --- | --- | --- | --- | --- |
| 类型 | CLI | CLI | IDE | IDE 插件 | IDE | CL |
| 出品方 | Anthropic | OpenAI | Cursor | GitHub | Codeium | Goo |
| Tab 补全 | — | — | ★★★ | ★★★ | ★★★ | — |
| Agent 执行 | ★★★ | ★★★ | ★★★ | ★★★ | ★★★ | ★★ |
| 终端内运行 | ★★★ | ★★★ | ★ | — | ★ | ★★ |
| 上下文窗口 | 200K | 跟模型 | 跟模型 | 跟模型 | 跟模型 | 2M |
| MCP 支持 | ✅ | ✅ | ✅ | ✅ | ✅ | 扩展 |
| Hook 自动化 | ✅ | ✅（beta，复用 Claude schema） | — | — | — | — |
| Subagent | ✅ | ✅（TOML） | — | — | — | — |
| 沙箱机制 | App 层 + Hook | OS 内核（Seatbelt/Landlock） | — | — | — | — |

| 维度 | Claude Code | Codex CLI | Cursor | Copilot | Windsurf | Gem CL |
|---|---|---|---|---|---|---|
| **多模型切换** | ★（仅 Claude） | ★（仅 OpenAI） | ★★★ | ★★ | ★★★ | ★（ Gemi |
| **开源** | — | ✅ Apache-2.0 | — | — | — | — |
| **国内直连** | — | — | — | — | — | — |
| **定价模式** | API / Pro 订阅 | ChatGPT 订阅 / API | 免费 / $20 Pro | 免费 / $10 Pro | 免费 / 订阅 | 慷慨 额 |

---

## 三、项目配置文件一览

知道每个工具的配置文件放哪、叫什么，是快速上手的关键。

| 工具 | 主配置文件 | 位置 | 备注 |
|---|---|---|---|
| Claude Code | `CLAUDE.md` + `.claude/` | 项目根 | 控制在 200 行以内，大项目拆到 `.claude/rules/` |
| Codex CLI | `AGENTS.md` + `.codex/config.toml` | 项目根 | `~/.codex/AGENTS.md` 全局；子目录 AGENTS.md 覆盖父级 |
| Cursor | `.cursor/rules/*.md` | 项目根 | 支持 `globs` 按文件类型加载 |
| Copilot | `.github/copilot-instructions.md` | 项目根 | 同目录 `agents/` `chatModes/` |

| 工具 | 主配置文件 | 位置 | 备注 |
| --- | --- | --- | --- |
| Windsurf | `.windsurfrules` | 项目根 | 单文件，不支持拆分 |
| Gemini CLI | `GEMINI.md` | 项目根 | 结构类似 CLAUDE.md |
| Kiro | `.kiro/steering/*.md` | 项目根 | 三种模式：`always` / `globs` / `manual` |
| Aider | `.aider.conf.yml` | 项目根 | YAML，含模型/lint/test 配置 |
| Trae | `.trae/rules/project_rules.md` | 项目根 | 支持中文规则 |
| OpenClaw | `~/.openclaw/openclaw.json` | 用户目录 | JSON5，热加载 |

## 四、核心命令速查

### Claude Code

```
# 安装与启动
npm install -g @anthropic-ai/claude-code
claude                              # 进入交互模式
claude --resume                     # 恢复上次对话
claude --model haiku                # 简单任务用便宜模型
claude -p "任务" --output-format json    # headless 模式

# 交互中
/compact                            # 压缩上下文
/plan                               # 进入 plan 模式
Esc                                 # 打断当前生成
```

### Codex CLI

```
# 安装与启动
npm install -g @openai/codex
codex                                   # 进入 TUI（首次会引导登录 ChatGPT）
codex --sandbox workspace-write     # 默认搭配（v0.125.0 起 `--full-auto` 已废弃，用此替代）
codex --sandbox read-only           # 只读探索
codex --add-dir ../sibling-repo     # 不放开沙箱、只多加可写目录
codex --yolo                            # 跳过沙箱+审批（仅在外部已隔离的环境用）
codex exec --json "..." | jq -c .   # JSONL 输出给后续脚本
codex --oss --local-provider ollama -m qwen2.5-coder    # 本地零成本
codex mcp-server                        # 把 Codex 暴露给其他 Agent 当工具
codex resume                            # 恢复上次对话

# 交互中
/init                                   # scaffold AGENTS.md
/plan       Shift+Tab               # Plan 模式
/model                                  # 切换模型
/review                                 # 审查 diff/分支/commit
/compact                                # 压缩上下文
/agent                                  # 切换 subagent thread
/diff                                   # 看 git diff（含未跟踪）
/debug-config                           # 排查 config.toml 不生效
```

### Cursor

```
Tab              — 接受补全
Cmd+L            — 打开 Chat（选中代码自动带入）
Cmd+I            — 打开 Composer（Agent 模式）
Cmd+K            — 行内编辑
Cmd+Shift+L      — 把当前文件加入 Chat 上下文
Esc              — 拒绝补全

@file @folder @web @terminal     — 引用
@notepad:名称                    — 引用 Notepad
```

### GitHub Copilot

```
Tab              — 接受补全
Esc              — 拒绝补全
Cmd+Shift+I      — 打开 Chat
Cmd+I            — 行内编辑
Alt+] / Alt+[    — 切换补全建议

#file #selection #terminal #problems   — 引用
@workspace                              — 全项目上下文
```

### Windsurf

```
Write 模式       — 直接写代码（类 Composer）
Chat 模式        — 对话问答

@file @folder @web     — 引用
Cascade 会自动追踪你的编辑流，不用手动贴上下文
```

### Gemini CLI

```
npm install -g @google/gemini-cli
gemini                              # 进入交互
# 利用 2M 上下文做大代码库分析（不适合小任务）
```

**Kiro**

```
1. 描述需求 → 2. Kiro 生成 Spec → 3. 审查确认 → 4. 自动实现 + 测试

Steering 加载模式：
- always              每次对话都加载
- globs: ["*.java"]   按文件匹配
- manual              手动激活
```

**Aider**

```
# 安装与启动
pip install aider-chat
aider --model claude-sonnet-4-5
aider --model deepseek/deepseek-chat    # 便宜
aider --model ollama/qwen2.5-coder      # 本地免费

# 交互中
/add file1 file2      — 添加文件到上下文
/drop file            — 移除文件
/code                 — code 模式（直接改）
/ask                  — ask 模式（只问不改）
/architect            — 先设计后实现
```

**Trae**

```
Builder 模式    — Agent，跨文件修改
Chat 模式       — 对话

@file @folder @web     — 引用
免费模型：Claude / GPT，按任务复杂度自选
```

```
openclaw onboard                      # 初始化
openclaw gateway start                # 启动网关
openclaw doctor                       # 诊断

openclaw skills install <slug>        # 安装 Skill
openclaw cron add "0 9 * * *" "..."   # 定时任务
openclaw models set default <model>   # 切换模型
openclaw channels add telegram        # 添加消息频道
```

---

## 五、选型决策流程

```
┌─ 主要在终端工作?
│    ├─ 要最强 Agent 能力 / 大型重构 → Claude Code
│    ├─ 已订阅 ChatGPT / 想要内核级沙箱 → Codex CLI
│    ├─ 要超大上下文 / 免费 → Gemini CLI
│    └─ 要多模型灵活切换 / Git 原生 → Aider
│
├─ 主要在 IDE 里?
│    ├─ 不想装新 IDE → GitHub Copilot (VS Code/JetBrains 插件)
│    ├─ 愿意换 IDE, 预算足 → Cursor
│    ├─ 喜欢 AI 主动帮忙 → Windsurf
│    ├─ 要中文 / 国内网络 / 免费 → Trae
│    └─ 团队协作 / 规格驱动 → Kiro
│
└─ 做编程以外的 AI 自动化?
     └─ 多平台、定时任务、Skill 生态 → OpenClaw
```

---

## 六、组合推荐

| 组合 | 场景 |
| --- | --- |
| Claude Code + Cursor | 最流行全栈组合：CLI 做重活、IDE 做日常 |
| Codex CLI + Cursor | ChatGPT 订阅者顺手用：CLI 做 Agent、IDE 做补全 |

| 组合 | 场景 |
| --- | --- |
| Codex CLI + Claude Code | 双 CLI 互补：Codex 跑 CI/脚本、Claude Code 做大重构 |
| Claude Code + Copilot | 纯 VS Code 用户的轻量选择 |
| Gemini CLI + Cursor | 预算敏感：2M 免费分析 + 20$ IDE |
| Aider + 本地 LLM | 零 API 成本：`ollama/qwen2.5-coder` + Aider |
| Codex CLI --oss + Ollama | 零 API 成本但要 Codex 的 Agent 体验：内核级沙箱 + 本地模型 |
| Claude Code + OpenClaw | 编程 + 自动化：CC 写代码，OpenClaw 跑定时任务 |

详见 多工具选型指南 和 实战场景脚本。

---

## 七、延伸阅读

- 各工具详细教程：见项目顶层目录的每个工具 README
- 提示词技巧：common/prompting.md
- 方法论总览：见项目 README 的 通用方法论 章节

# AI 编程提示词技巧

> 这不是通用的 prompt engineering 教程，而是专门针对 AI 编程场景的提示词技巧。不管你用 Claude Code、Cursor 还是 Copilot，这些原则都适用。

---

## 核心原则

#### 1. 具体 > 模糊

AI 不会读心术。你说得越具体，结果越好。

❌ "帮我优化这个函数"
✅ "这个函数处理 10 万条数据时需要 30 秒，
目标是降到 5 秒以内。
可以考虑批处理、缓存、或者换算法。
不要改函数签名。"

#### 2. 约束 > 自由

没有约束的 AI 会发挥过度。明确告诉它什么不要做。

❌ "加个缓存"
✅ "给 getUserById 加缓存。
用内存缓存（Map），不要引入 Redis。
TTL 5 分钟。
不要改其他函数。
不要加新依赖。"

#### 3. 分步 > 一次

复杂任务拆成步骤，每步确认再继续。

❌ "重构整个认证模块"
✅ "重构认证模块，分三步走：
第一步：先分析现在的问题，列出来让我确认。
第二步：给出重构方案（2-3 个选项）。
第三步：确认方案后再开始改代码。
现在先做第一步。"

#### 4. 示例 > 描述

给一个例子比描述半天清楚。

❌ "返回值用统一格式"
✅ "返回值统一用这个格式：

```
{
  'code': 200,
  'data': { ... },
  'message': 'success'
}
```

错误时：

```
{
  'code': 400,
  'data': null,
  'message': '参数 email 格式不正确'
}"
```

#### 5. 参考 > 从零开始

指向已有代码比描述风格有效得多。

❌ "写一个新的 API 接口"
✅ "参考 src/api/users.ts 的风格，
写一个 src/api/orders.ts。
路由、错误处理、返回格式都保持一致。"

---

## 常用提示词模式

### 分析模式（先看后说）

```
先读 [文件/目录]，分析 [什么问题]。
列出你的发现，不要直接改代码。
等我确认后再动手。
```

### 实现模式（明确需求）

```
在 [位置] 实现 [功能]。
要求：
1. [具体要求 1]
2. [具体要求 2]
参考 [现有文件] 的风格。
不要 [禁止事项]。
```

### 修复模式（给证据）

```
[错误描述/日志/截图]。
先分析可能的原因（不要猜，看代码和日志）。
确认根因后再给修复方案。
只改必要的代码，不要顺手重构。
```

### 审查模式（定范围）

```
审查 [文件/PR]，重点关注：
1. [关注点 1，如性能]
2. [关注点 2，如安全]
3. [关注点 3，如边界情况]
按严重程度排序，给出修改建议。
```

---

## 反模式

| 反模式 | 为什么不好 | 改成 |
| --- | --- | --- |
| "帮我写个网站" | 范围太大，AI 不知道从哪开始 | 拆成具体任务 |
| "优化一下" | 优化什么？性能？可读性？体积？ | 说清楚优化目标和指标 |
| "写最好的代码" | "最好"没有定义 | 说清楚标准（可测试、可维护等） |
| 一次给 10 个需求 | AI 会顾此失彼 | 一次一个，做完确认再下一个 |
| 不给错误信息就说"修 bug" | AI 只能猜 | 贴上错误日志、复现步骤 |

# 需求拆解

> AI 编程工具一次做好一个小任务的成功率远高于一次做一个大任务。把大需求拆成 AI 能"一次做对"的小任务，是高效使用 AI 编程的核心技能。

---

## 拆解原则

### 1. 每个任务应该是一个"原子操作"

```
❌ 太大："重构整个用户模块"
✅ 合适：
  任务 1：把 UserService 的数据库查询从原始 SQL 改成 ORM
  任务 2：给 UserService 的每个 public 方法加单元测试
  任务 3：把 UserController 的参数校验从手写改成 Zod
  任务 4：统一 User 相关接口的错误返回格式
```

### 2. 每个任务有明确的"完成标准"

```
❌ 模糊："优化一下性能"
✅ 明确："getUserList 接口响应时间从 3s 降到 500ms 以内。
        跑 pnpm test 验证功能不变。
        跑 ab -n 1000 -c 10 验证性能目标。"
```

### 3. 任务之间有清晰的依赖关系

```
任务 1：创建 Order 模型和数据库表（独立，先做）
任务 2：实现 OrderService CRUD（依赖任务 1）
任务 3：实现 OrderController 接口（依赖任务 2）
任务 4：写集成测试（依赖任务 3）
任务 5：加权限校验（依赖任务 3）

任务 4 和 5 可以并行。
```

---

## 拆解模板

### 新功能开发

```
第一轮：设计
- 分析需求，提出澄清问题
- 给出 2-3 个技术方案
- 确认方案

第二轮：数据层
- 创建数据模型/数据库表
- 写数据访问层代码
- 验证：能正确读写数据库

第三轮：业务层
- 实现核心业务逻辑
- 写单元测试
- 验证：测试通过

第四轮：接口层
- 实现 API 接口
- 加参数校验和权限
- 写接口测试
- 验证：接口能正确调用

第五轮：收尾
- 加错误处理和日志
- 更新文档
- 完整回归测试
```

### Bug 修复

```
第一步：复现和定位（不要改代码）
第二步：分析根因（给出假设）
第三步：确认根因（验证假设）
第四步：修复 + 写防回归测试
第五步：验证修复 + 回归测试
```

**重构**

```
第一步：写测试覆盖现有行为（安全网）
第二步：小步重构（每次只改一个点）
第三步：每次改完跑测试（确保不破坏功能）
第四步：重复第二步和第三步
第五步：清理（删除过渡代码、更新文档）
```

---

## 实际例子

需求：给电商系统加购物车功能

```
拆解后：

1. [数据] 创建 cart 和 cart_item 表，写 Migration
   完成标准：migrate up/down 都能跑通

2. [模型] 创建 Cart 和 CartItem 的 ORM 模型
   完成标准：能 CRUD 购物车数据

3. [服务] 实现 CartService
   - addItem（加商品到购物车）
   - removeItem（删除商品）
   - updateQuantity（修改数量）
   - getCart（获取购物车）
   - clearCart（清空购物车）
   完成标准：单元测试全部通过

4. [接口] 实现 REST API
   - POST /api/cart/items
   - DELETE /api/cart/items/:id
   - PATCH /api/cart/items/:id
   - GET /api/cart
   - DELETE /api/cart
   完成标准：接口测试通过，权限校验正确

5. [集成] 和订单模块对接
   - 购物车结算 → 创建订单
   - 下单后清空购物车
   完成标准：集成测试通过
```

每个任务独立让 AI 完成，完成后验证再进入下一个。

# 上下文管理

> AI 编程工具的"智商"直接取决于上下文质量。给太少它不理解你的项目，给太多它反而变笨。管理好上下文是用好 AI 编程工具的关键技能。

---

## 上下文是怎么工作的

所有 AI 编程工具都有一个**上下文窗口**（context window），可以理解为 AI 的"工作记忆"。

| 工具 | 上下文窗口 | 说明 |
| --- | --- | --- |
| Claude Code | 200K tokens | 约 15 万字，最大 |
| Cursor | 128K–200K tokens | 取决于选择的模型 |
| Copilot | 128K tokens | 包含打开的文件 |
| Gemini CLI | 1M–2M tokens | 窗口最大，但太大也有问题 |

**关键认知**：上下文不是越大越好。塞太多无关信息，AI 会"注意力分散"，重要信息反而被淹没。

---

## 核心策略

### 1. 精确喂入，不要全塞

```
❌ 差："看看整个 src/ 目录帮我找 bug"
✅ 好："看 src/services/auth.ts 的 refreshToken 函数，
    用户反馈说刷新后还是提示过期。
    也看一下 src/middleware/auth.ts 里怎么验证 token 的。"
```

#### 2. 项目配置文件是最高效的上下文

每个工具都有项目配置文件，启动时自动加载：

| 工具 | 配置文件 | 作用 |
| --- | --- | --- |
| Claude Code | `CLAUDE.md` | 项目背景、规范、常用命令 |
| Cursor | `.cursorrules` / `.cursor/rules/` | 项目规则 |
| Copilot | `.github/copilot-instructions.md` | 项目指引 |
| Windsurf | `.windsurfrules` | 项目规则 |
| Gemini CLI | `GEMINI.md` | 项目配置 |

**写好配置文件 = 每次对话自动带上最关键的上下文。**

#### 3. 长对话定期"刷新"

对话越长，早期的信息权重越低。当你感觉 AI 开始"忘事"或"变笨"：

```
# Claude Code
> /clear    # 清除当前对话，开新的

# 或者主动总结
> 我们刚才做了这些事：[总结]。
  现在继续做 [下一个任务]。
```

#### 4. 用文件传递上下文，不用对话

```
❌ 差：在对话里贴了 500 行代码让 AI 分析
✅ 好：代码已经在文件里了，告诉 AI 读哪个文件
    "读 src/services/payment.ts 第 100-150 行的 processRefund 函数"
```

---

## 各工具的上下文管理

### Claude Code

```
# 用 CLAUDE.md 做持久上下文
# 用 Memory 做跨对话记忆
# 用 /clear 重置对话

# 大项目技巧：用 Subagent 隔离上下文
"用 subagent 分析 src/payment/ 的代码，
 主对话的上下文不要被污染。"
```

### Cursor

```
# 用 @ 引用精确控制上下文
@src/models/user.ts @src/schemas/user.ts
基于这两个文件写一个用户注册接口

# 用 Notepads 保存常用上下文
# 用 Rules globs 按文件类型自动加载规则

# 打开相关文件作为隐式上下文
# Cursor 会读取当前打开的 tab
```

### Copilot

```
# 用 #file #selection #terminal 精确引用
#file:src/models/user.py 基于这个模型写测试

# 打开相关文件 — Copilot 读取打开的 tab
# 关闭无关文件 — 减少噪音

# Agent 模式自动搜索相关文件
# 但可以用 #file 缩小范围提高准确率
```

---

## 常见问题

| 问题 | 原因 | 解决 |
| --- | --- | --- |
| AI 回答不准确 | 上下文不够或太杂 | 精确引用相关文件 |
| AI 越来越笨 | 对话太长，上下文溢出 | 开新对话，带上 CLAUDE.md |
| AI 忘了之前的约定 | 早期消息权重降低 | 重要约定放在配置文件里 |
| AI 编造不存在的 API | 没给足参考信息 | 让它先 grep/search 确认 |
| AI 重复做已经做过的事 | 不记得之前的进度 | 用 todo/plan 追踪进度 |

# Cursor 最佳实践

> Cursor 是基于 VS Code 的 AI IDE，集成了代码补全、Chat、Composer（Agent）三大核心功能。它的优势在于和编辑器的深度集成——选中代码直接对话、在编辑器内实时预览修改。

---

## 核心概念

| 概念 | 说明 | 用途 |
| --- | --- | --- |
| Tab 补全 | 基于上下文的代码补全 | 日常编码提速 |
| Chat | 侧边栏对话，可选中代码提问 | 理解代码、问答 |
| Composer | Agent 模式，可跨文件修改 | 复杂任务、重构 |
| Rules | `.cursor/rules/*.md` 规则文件 | 控制 AI 行为 |
| @引用 | `@file` `@folder` `@web` 等 | 精确指定上下文 |
| Notepads | 可复用的上下文片段 | 复杂项目的上下文管理 |

---

## 快速上手

#### .cursorrules — 项目规则

在项目根目录创建 `.cursorrules` 或 `.cursor/rules/` 下放规则文件：

```
# 项目规则

## 技术栈
- React 18 + TypeScript + Tailwind CSS
- 状态管理：Zustand
- 路由：React Router v6
- 测试：Vitest + Testing Library

## 代码风格
- 函数组件 + Hooks，不用 Class 组件
- Props 用 interface 定义，不用 type
- 文件命名 kebab-case，组件命名 PascalCase
- 每个组件一个文件

## 禁止
- 不要用 any 类型
- 不要用 useEffect 做数据获取，用 TanStack Query
- 不要直接操作 DOM，用 React ref
```

**@ 引用技巧**

```
# 引用文件
@src/components/Button.tsx 这个组件的 Props 类型不对，帮我修一下

# 引用文件夹
@src/api/ 这些接口的错误处理不统一，帮我统一改成...

# 引用文档
@https://tanstack.com/query/latest 参考官方文档，帮我把 useEffect 里的数据获取改成
useQuery

# 引用终端
@terminal 看看刚才的报错信息，帮我修
```

---

## 提示词技巧

### 1. Composer 大任务拆解

```
用 Composer 模式。按以下步骤执行：
1. 先读 src/pages/ 下所有页面组件，理解路由结构
2. 创建 src/layouts/DashboardLayout.tsx 统一布局
3. 把所有页面组件改成使用新布局
4. 确保路由配置正确
每步完成后暂停，等我确认再继续。
```

### 2. 选中代码直接对话

选中一段代码后按 `Cmd+L`（macOS）进入 Chat：

```
# 选中一段复杂的正则表达式
解释这个正则在做什么，有没有边界情况没覆盖到？

# 选中一个函数
这个函数的时间复杂度是多少？有更优的写法吗？

# 选中一段样式代码
把这段 CSS 改成 Tailwind 的写法
```

### 3. 用 Notepads 管理复杂上下文

对于大型项目，创建 Notepad 保存常用上下文：

```
Notepad: "API 规范"
---
所有 API 返回格式：
{ code: number, data: T, message: string }

错误码：
- 400: 参数错误
- 401: 未认证
- 403: 无权限
- 500: 服务器错误

认证方式：Bearer Token in Authorization header
```

对话时引用：`@notepad:API规范 按这个格式实现用户列表接口`

---

## 进阶技巧

#### Rules 分文件管理

```
.cursor/rules/
├── global.md           # 全局规则（代码风格、命名等）
├── react.md            # React 相关规则
├── api.md              # API 开发规则
├── testing.md          # 测试规则
└── security.md         # 安全规则
```

每个文件可以设置触发条件：

```
---
globs: ["src/components/**/*.tsx"]
---

# React 组件规则
- 所有组件必须有 displayName
- Props 超过 3 个必须用 interface 拆分
- 必须处理 loading 和 error 状态
```

#### 用 superpowers-zh 增强 Rules

手动写 Rules 太慢？用 superpowers-zh 一键安装方法论：

```
cd /your/project
npx superpowers-zh
# 自动安装到 .cursor/rules/，包含 brainstorming、debugging、verification 等
```

安装后 Cursor 在匹配文件时会自动加载对应的 skill 规则。

#### 模型选择策略

| 场景 | 推荐模型 | 原因 |
| --- | --- | --- |
| Tab 补全 | cursor-small | 速度快，延迟低 |
| 简单问答 | Claude Sonnet | 性价比高 |
| 复杂重构 | Claude Opus | 理解力最强 |
| Composer 大任务 | Claude Opus | 多文件协调能力好 |

#### 快捷键

| 快捷键 | 功能 |
| --- | --- |
| `Tab` | 接受补全 |
| `Cmd+L` | 打开 Chat（选中代码自动带入） |
| `Cmd+I` | 打开 Composer |
| `Cmd+K` | 行内编辑（选中代码后） |
| `Cmd+Shift+L` | 把当前文件加入 Chat 上下文 |

---

## 常见陷阱

| 陷阱 | 说明 | 解决 |
| --- | --- | --- |
| 规则太长 | `.cursorrules` 几千行，AI 记不住 | 拆分到 `.cursor/rules/` 用 globs 按需加载 |
| Composer 失控 | Agent 模式改了不该改的文件 | 用 `@file` 限定范围，或在 rules 里标明禁区 |

| 陷阱 | 说明 | 解决 |
| --- | --- | --- |
| 补全太激进 | Tab 补全一次生成太多代码 | 设置里调整补全长度，或按 `Esc` 拒绝 |
| 上下文不够 | AI 不理解项目结构 | 用 `@folder` 引用关键目录，写好 `.cursorrules` |

👉 **深度展开版**：Cursor 陷阱合集 — 8 个真实踩坑场景（Composer 脱缰 / @file 失效 / Notepads 过期 等），每个带症状 / 根因 / 出坑 / 预防

---

## 配置模板

直接复制到你的项目 `.cursor/rules/` 目录下：

| 模板 | 用途 |
| --- | --- |
| global.cursorrules.md | 全局规则（代码风格、命名、禁止事项） |
| api.cursorrules.md | API 开发规则（仅在 API 目录下生效） |

#### 模型配置

Cursor 的模型选择在设置界面（`Cursor Settings > Models`）中配置，不需要手动编辑 JSON。

---

## 延伸阅读

- Cursor 官方文档
- awesome-cursorrules — 社区 Rules 集合（38k+ star）
- superpowers-zh — Skills 方法论（也支持 Cursor）

# GitHub Copilot 最佳实践

> GitHub Copilot 是 VS Code / JetBrains 内置的 AI 编程助手。它的核心优势是**无缝集成** — 不需要切换工具，在编辑器里自然地写代码就有补全和建议。Copilot Agent 模式让它从补全工具进化成了能独立完成任务的 Agent。

---

## 核心概念

| 概念 | 说明 | 用途 |
| --- | --- | --- |
| 代码补全 | 行内灰色建议 | 日常编码，按 Tab 接受 |
| Chat | 侧边栏对话 `Cmd+Shift+I` | 问答、解释代码 |
| Agent 模式 | 自主完成多步骤任务 | 复杂任务、跨文件修改 |
| Copilot Instructions | `.github/copilot-instructions.md` | 项目级配置 |
| Chat Modes | 自定义对话角色 | 安全审查、测试专家等 |
| MCP Server | 扩展 Copilot 的工具能力 | 连接数据库、API 等 |
| #引用 | `#file` `#selection` `#terminal` | 精确指定上下文 |

---

## 快速上手

#### Copilot Instructions — 项目配置

在项目根目录创建 `.github/copilot-instructions.md`：

```
# 项目指引

## 技术栈
- Python 3.12 + FastAPI + SQLAlchemy 2.0
- 数据库：PostgreSQL
- 缓存：Redis
- 测试：pytest

## 代码规范
- 类型注解必须完整，使用 Python 3.10+ 语法
- async 函数统一加 `async_` 前缀
- 所有接口必须有 Pydantic 模型做入参校验
- 错误用自定义异常类，不要用裸 Exception

## 目录约定
- src/api/      — FastAPI 路由
- src/models/   — SQLAlchemy 模型
- src/schemas/  — Pydantic 模型
- src/services/ — 业务逻辑
- tests/        — 测试（镜像 src 结构）
```

**# 引用技巧**

```
# 引用文件
#file:src/models/user.py 基于这个模型写一个用户注册接口

# 引用选中代码
#selection 这段代码有什么性能问题?

# 引用终端输出
#terminal 看看报错信息帮我修

# 引用 VS Code 问题面板
#problems 帮我修复这些类型错误
```

---

## Agent 模式

Copilot Agent 是 2025 年最大的更新。从对话框输入任务，Agent 会：

1. 分析需求 → 2. 搜索相关文件 → 3. 制定计划 → 4. 逐步执行 → 5. 运行测试验证

#### 使用 Agent 模式

在 Chat 中选择 Agent 模式（或直接描述复杂任务）：

```
@workspace 给 src/api/ 下所有接口加上 rate limiting，
使用 Redis 做计数器，每个用户每分钟最多 60 次请求。
需要：
1. 一个可复用的 rate_limit 装饰器
2. Redis 连接配置
3. 超限时返回 429 状态码
4. 给每个接口加上测试
```

#### Agent + MCP 扩展能力

通过 MCP Server 让 Agent 访问外部工具：

```jsonc
// .vscode/settings.json
{
  "github.copilot.chat.mcpServers": {
    "postgres": {
      "command": "npx",
      "args": ["@modelcontextprotocol/server-postgres", "postgresql://..."]
    }
  }
}
```

然后你可以说：

```
查一下数据库里 users 表的结构，
然后根据实际字段生成 SQLAlchemy 模型和 Pydantic schema。
```

---

## 提示词技巧

#### 1. 写好注释让补全更准

```
# 补全质量取决于上下文。写一行注释，补全就知道你要什么：

# 计算两个日期之间的工作日天数，排除周末和中国法定节假日
def count_business_days(start: date, end: date) -> int:
    # Copilot 会自动补全完整实现
```

#### 2. 用自定义 Agent 做专项审查

创建 `.github/agents/security-reviewer.agent.md`：

```
---
name: 安全审查员
description: 按 OWASP Top 10 审查代码安全
---

你是一名安全审查专家。审查代码时：
1. 按 OWASP Top 10 逐项检查
2. 关注 SQL 注入、XSS、CSRF、权限绕过
3. 检查敏感数据是否明文存储或日志泄露
4. 标注风险等级：🔴 严重 / 🟡 中等 / 🟢 低
```

在 Chat 中用 `@security-reviewer` 调用这个角色。

#### 3. 利用 Workspace 全局理解

```
@workspace 项目里有没有硬编码的密钥或敏感信息？
帮我全部找出来，改成环境变量。
```

---

## 进阶技巧

#### 自定义指令文件

Copilot 支持通过指令文件细化行为：

- **`.github/copilot-instructions.md`** — 全局项目指令

- `.github/chatModes/` — 自定义 Chat 角色（如安全审查员、测试专家）

如果你用 superpowers-zh，也可以在 `.claude/skills/` 下的 skill 文件里写方法论，Copilot Agent 的 `@workspace` 命令会读取项目中的 markdown 文件。

#### VS Code 快捷键

| 快捷键 | 功能 |
| --- | --- |
| `Tab` | 接受补全 |
| `Esc` | 拒绝补全 |
| `Cmd+Shift+I` | 打开 Copilot Chat |
| `Cmd+I` | 行内编辑 |
| `Alt+]` / `Alt+[` | 切换不同补全建议 |

#### 补全优化

让 Copilot 补全更准确的几个技巧：

1. **打开相关文件** — Copilot 会读取当前打开的 tab 作为上下文
2. **写好类型注解** — 类型越完整，补全越准
3. **函数签名先写好** — 先写好函数名、参数、返回类型，再让它补全函数体
4. **保持文件简短** — 大文件上下文噪音多，Copilot 容易跑偏

---

## 常见陷阱

| 陷阱 | 说明 | 解决 |
| --- | --- | --- |
| 补全过时 API | Copilot 用了过期的库 API | 打开库的源码或文档作为 tab 上下文 |
| Agent 改错文件 | 多文件修改时动了不该动的 | 用 `#file` 限定范围 |

| 陷阱 | 说明 | 解决 |
| --- | --- | --- |
| 忽略 .gitignore | Copilot 可能读取 node_modules | 检查设置里的 exclude 配置 |
| Instructions 太长 | 超过模型上下文窗口 | 精简到关键规则，详细规则拆到 Chat Modes |

👉 **深度展开版**：Copilot 陷阱合集 — 8 个真实踩坑场景（Agent 读 .env / MCP 失效 / 免费限流等），每个带症状 / 根因 / 出坑 / 预防

---

## 配置模板

直接复制到你的项目里用：

| 模板 | 用途 |
| --- | --- |
| copilot-instructions.md | 项目指引模板，复制到 `.github/copilot-instructions.md` |
| security-reviewer.agent.md | 安全审查角色，复制到 `.github/agents/` |

---

## 延伸阅读

- Copilot 官方文档
- awesome-copilot — 官方资源集合（27k+ star）
- superpowers-zh — Skills 方法论（也支持 VS Code Copilot）

# Claude Code 最佳实践

> Claude Code 是 Anthropic 官方的 CLI AI 编程工具。它不只是一个代码补全工具，而是一个能理解整个项目、执行复杂任务的 Agent。

---

## 核心概念

在深入技巧之前，先搞清楚 Claude Code 的几个核心概念：

| 概念 | 说明 | 用途 |
| --- | --- | --- |
| Subagent | 子进程 Agent，独立执行任务 | 并行处理、隔离上下文 |
| Command | `/` 开头的快捷命令 | 快速执行常用操作 |
| Skill | `.claude/skills/` 下的方法论文件 | 教 AI 怎么做事 |
| Hook | 工具调用前后的钩子脚本 | 自动化校验、通知 |
| MCP Server | Model Context Protocol 服务 | 扩展 AI 能力（数据库、API 等） |
| Memory | 持久化记忆 | 跨对话保留上下文 |
| Checkpoint | 自动保存点 | 安全回滚 |

---

## 快速上手

### 安装

```
# npm 全局安装
npm install -g @anthropic-ai/claude-code

# 或者直接用 npx
npx @anthropic-ai/claude-code
```

### 第一次使用

```
cd /your/project
claude
```

进入交互模式后，试试这几个命令：

```
# 了解项目
> 这个项目是做什么的？帮我梳理一下架构

# 做一个小任务
> 给 utils/string.ts 加一个 truncate 函数，超过指定长度截断并加省略号

# 做大任务（直接描述，Claude Code 会自动规划执行）
> 重构 src/api/ 下所有接口，统一错误处理格式
```

### CLAUDE.md — 项目配置文件

在项目根目录创建 `CLAUDE.md`，Claude Code 每次启动都会读取：

```
# 项目说明
这是一个 Next.js 14 + TypeScript 的电商后台管理系统。

# 代码规范
- 使用 TypeScript strict mode
- 组件用 PascalCase 命名
- API 路由放在 src/app/api/ 下
- 测试用 Vitest，放在 __tests__/ 下

# 常用命令
- 启动开发：pnpm dev
- 跑测试：pnpm test
- 类型检查：pnpm typecheck
- Lint：pnpm lint

# 注意事项
- 不要修改 src/legacy/ 下的代码，那是旧版本兼容层
- 数据库 migration 必须通过 drizzle-kit 生成，不要手写 SQL
```

---

## 提示词技巧

### 1. 给足上下文，一次说清楚

```
❌ 差：加个登录功能
✅ 好：给 src/app/api/ 加一个 JWT 登录接口。用 bcrypt 验证密码，
    token 过期时间 7 天，错误返回统一的 { code, message } 格式。
    参考 src/app/api/register/route.ts 的风格。
```

### 2. 指定参考文件

```
参考 src/components/UserTable.tsx 的风格，
给 src/components/ 新建一个 OrderTable.tsx，
支持分页、排序、搜索，用 TanStack Table。
```

#### 3. 先让它分析，再让它动手

```
先读 src/services/payment.ts 和 src/services/order.ts，
分析一下现在的支付流程有什么问题，
给出改进方案，确认后再改。
```

#### 4. 限制范围

```
只改 src/utils/date.ts 这一个文件。
不要动其他文件，不要加新依赖。
```

#### 5. 用否定句划红线

```
实现缓存功能。
不要用 Redis，用内存缓存。
不要引入新依赖，用 Map 实现。
不要修改现有接口的签名。
```

---

## 进阶技巧

#### Agent 能力

Claude Code 本身就是 Agent——直接描述任务，它会自主规划和执行：

```
> 给整个项目加上单元测试，覆盖率目标 80%
```

Claude Code 会自己：读代码 → 制定计划 → 写测试 → 跑测试 → 修复失败 → 验证通过

> **提示**：不需要特殊命令或 flag。直接描述你想要的结果，Claude Code 会自动决定是否需要多步骤执行。

#### Subagent 并行

大任务可以拆成多个 Subagent 并行执行：

```
同时做这三件事：
1. 给 src/api/users.ts 加分页参数
2. 给 src/api/orders.ts 加日期筛选
3. 给 src/api/products.ts 加搜索功能
用 subagent 并行执行，每个独立完成。
```

#### Skill 文件

Skill 是最强大的功能之一。在 `.claude/skills/` 下放方法论文件，AI 会自动加载：

```
.claude/skills/
├── brainstorming.md       # 需求分析流程
├── debugging.md           # 调试方法论
├── code-review.md         # 代码审查规范
└── verification.md        # 完成前验证
```

**快速安装 superpowers-zh**（20 个经过实战验证的 skill）：

```
cd /your/project
npx superpowers-zh
# 自动检测项目类型，安装到 .claude/skills/
# 也支持 Cursor、Gemini CLI 等工具
```

安装后 Claude Code 会自动加载这些 skill，你可以用 `/` 命令调用，比如 `/brainstorming` 做需求分析、`/debugging` 做系统化调试。详见 superpowers-zh。

**需要专业角色？** 用 agency-agents-zh 的 211 个 AI 专家角色：

```
# 把角色文件复制到 .claude/skills/ 即可使用
# 比如数据库优化师、安全工程师、代码审查员等
```

在 CLAUDE.md 里引用角色：

```
# 角色
当我说"用安全专家审查"时，按 .claude/skills/security-engineer.md 的角色行事。
```

#### Hook 自动化

Hook 在工具调用前后自动执行脚本：

```
// .claude/settings.json
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          { "type": "command", "command": "echo 'About to run a command'" }
        ]
      }
    ],
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          { "type": "command", "command": "pnpm lint --fix" }
        ]
      }
    ]
  }
}
```

> Hook 的触发点有 `PreToolUse`、`PostToolUse`、`Notification` 等，每个触发点下用 `matcher` 匹配工具名。详见 官方文档。

用途：

- 每次修改文件后自动跑相关测试
- 每次提交前自动 lint
- 每次创建文件后自动加版权头

#### Git Worktree 隔离开发

做实验性改动时，用 worktree 隔离，不影响主分支：

```
在 git worktree 里做这个重构。
如果效果好就合并，不好就丢弃。
```

#### Memory 跨对话记忆

Claude Code 可以在 `~/.claude/` 下存储记忆，跨对话使用：

```
记住：这个项目的部署用的是 Vercel，
CI 用 GitHub Actions，数据库是 Supabase PostgreSQL。
以后做部署相关的任务都要考虑这些。
```

#### 多智能体编排

当任务复杂到需要多个 AI 角色协作时（比如架构师设计 → 开发者实现 → 审查员审查），可以用 agency-orchestrator 做 YAML 编排：

```yaml
# workflow.yaml
name: 新功能开发
steps:
  - agent: 软件架构师
    task: 分析需求，输出技术方案
    output: design.md

  - agent: 后端开发者
    task: 按 design.md 实现代码
    input: design.md

  - agent: 代码审查员
    task: 审查代码质量和安全性
```

适合场景：团队级复杂任务、需要多轮审查的交付、标准化开发流程。

---

## 调试技巧

### 1. 让它看日志，不要让它猜

❌ 差：测试挂了，帮我修
✅ 好：跑 `pnpm test src/api/users.test.ts`，
把失败的测试输出贴出来，分析根因后再修。

### 2. 缩小范围

只看 `src/services/auth.ts` 第 45-80 行的逻辑，
用户反馈说"刷新 token 后还是提示过期"。
先分析可能的原因，不要直接改。

### 3. 对比分析

`git log` 看看 `src/api/payment.ts` 最近的改动，
对比改动前后的逻辑，找出哪次改动引入了这个 bug。

---

## 技巧速查表（60+）

按场景分类，一行一个技巧。收藏这一页就够了。

### 提示词（12）

| # | 技巧 | 说明 |
|---|---|---|
| 1 | 先分析再动手 | 让 Claude 先读代码、给方案，确认后再改。避免一上来就重写 |
| 2 | 推翻重来 | 不满意？说"扔掉这个方案，用你现在了解到的信息重新设计一个优雅的实现" |
| 3 | 考考我 | "帮我审查这个改动，问我问题，直到你确认我理解了再提 PR" |
| 4 | 限定范围 | "只改这一个文件，不要动其他文件，不要加新依赖" |
| 5 | 指定参考文件 | "参考 src/api/users.ts 的风格写 orders.ts"，比空描述好 10 倍 |

| # | 技巧 | 说明 |
|---|---|---|
| 6 | 用否定句划红线 | "不要用 Redis，不要引入新依赖，不要改接口签名" |
| 7 | 给出 2-3 个方案让我选 | "给出 2 个方案，分析优缺点，我选了再做" |
| 8 | 分步确认 | "每步做完暂停等我确认，不要一口气做完" |
| 9 | 明确完成标准 | "完成标准：所有测试通过 + TypeScript 无报错 + lint 通过" |
| 10 | 用 ultrathink 深度思考 | 在提示词开头加"ultrathink"或"think really hard"触发扩展思考 |
| 11 | 让它写 commit message | "帮我写 commit message，描述这次改动的 why 而不是 what" |
| 12 | 英文提示词更精确 | 复杂技术任务用英文描述更精确，简单任务中文即可 |

**CLAUDE.md（10）**

| # | 技巧 | 说明 |
|---|---|---|
| 1 | 控制在 200 行以内 | 太长 AI 会忽略后面的内容。大项目拆到 `.claude/rules/` |
| 2 | "跑测试"测试法 | 任何人打开 Claude Code 说"跑测试"就能成功 → CLAUDE.md 写够了 |
| 3 | settings.json 代替"不要" | "不要修改 x 文件"写在 CLAUDE.md 容易被忽略，权限控制更可靠 |
| 4 | 加 `<important>` 标签 | 关键规则用 `<important>` 包裹，模型会更重视 |
| 5 | 写常用命令 | dev、test、build、lint、deploy 命令都写上，AI 不用猜 |
| 6 | 写禁止事项 | 明确列出不能动的文件、不能用的方法、不能引入的依赖 |
| 7 | 写目录结构 | 关键目录说明，让 AI 知道代码在哪里 |

| # | 技巧 | 说明 |
| --- | --- | --- |
| 8 | 分层 CLAUDE.md | 根目录放全局规则，子目录放模块规则（如 `src/api/CLAUDE.md`） |
| 9 | 定期更新 | 项目演进了 CLAUDE.md 也要跟着更新，过时的规则比没有规则更糟 |
| 10 | 用 `.claude/rules/` 按条件加载 | 大项目用 globs 按文件类型加载规则，避免一次全塞进去 |

### Agent 与 Subagent（10）

| # | 技巧 | 说明 |
| --- | --- | --- |
| 1 | 按功能拆 subagent | 创建"支付模块专用 agent"而不是泛泛的"后端工程师" |
| 2 | 对抗式测试 | 一个 agent 写代码，另一个（独立上下文）找 bug |
| 3 | Skill description 写给模型看 | 写"当用户要做 X 时触发"而不是人类摘要 |
| 4 | Skill 里加 gotchas | 把 Claude 犯过的错记在 skill 里，最高信噪比的内容 |
| 5 | `context: fork` 隔离 | Skill 在独立 subagent 跑，主上下文只看最终结果 |
| 6 | 不要让 agent 做太多事 | 一个 agent 一个明确任务，比一个 agent 做 5 件事成功率高 |
| 7 | agent 之间传文件而不是传消息 | 让 agent A 的输出写到文件，agent B 读文件。比口头传递可靠 |
| 8 | 用 worktree 做实验 | 让 subagent 在 git worktree 里做实验性改动，不满意就丢弃 |
| 9 | 用 MCP 扩展能力 | 连数据库、调 API、查文档 — MCP 让 agent 能做的事多 10 倍 |
| 10 | Custom command 做常用操作 | 把重复性高的操作封装成 `/command`，一键触发 |

## Hook（8）

| # | 技巧 | 说明 |
|---|---|---|
| 1 | PostToolUse 自动格式化 | Claude 写代码后自动跑 prettier/eslint --fix，避免格式问题 |
| 2 | PostToolUse 自动跑测试 | 每次修改文件后自动跑相关测试，及时发现问题 |
| 3 | PreToolUse 拦截危险操作 | 在 `Bash` 工具执行前检查命令，拦截 `rm -rf` 等危险操作 |
| 4 | Notification hook 发通知 | 长任务完成后自动发 Slack/钉钉通知 |
| 5 | Stop hook 强制验证 | 在每轮结束时提醒 Claude 检查自己的输出 |
| 6 | hook 里用 exit 1 阻止执行 | hook 脚本返回非 0 就会阻止工具调用，用于强制规则 |
| 7 | hook 输出作为上下文 | hook 的 stdout 会被 Claude 看到，可以传递额外信息 |
| 8 | 按工具名精确匹配 | matcher 支持 `Write\|Edit`、`Bash` 等，不要用 `*` 匹配所有 |

## 工作流（12）

| # | 技巧 | 说明 |
|---|---|---|
| 1 | 上下文 50% 时手动 `/compact` | 不要等自动压缩，主动压缩能保持 AI 质量 |
| 2 | Esc Esc 回退 checkpoint | 跑偏了用 checkpoint 回滚，不要在错误的上下文里继续修 |
| 3 | PR 保持小而聚焦 | 理想 PR 中位数约 120 行。大改动拆成多个 PR |
| 4 | 先合并迁移再做新功能 | 半迁移的代码库让 AI 选错模式。保持代码库干净 |
| 5 | 新对话做新任务 | 每个独立任务开新对话。旧对话的上下文污染会降低质量 |

| # | 技巧 | 说明 |
|---|---|---|
| 6 | 用 plan mode 开始 | 复杂任务先进 plan mode（`/plan`），确认方案后再执行 |
| 7 | 多次快速迭代 > 一次完美 | 先出 MVP，跑通后再优化。不要一次提示就期望完美 |
| 8 | 让 Claude 自己跑测试 | 不要替它跑，让它自己跑、看输出、修复。这是 agent 的强项 |
| 9 | 用 `--resume` 继续上次对话 | 中断后用 `claude --resume` 恢复上下文继续工作 |
| 10 | 用 `--print` 做非交互任务 | CI/CD 里用 `claude --print "检查代码风格"` 做自动化 |
| 11 | headless 模式跑批量任务 | `claude -p "任务" --output-format json` 适合脚本调用 |
| 12 | 并行 worktree 提效 | 多个 worktree 跑多个 Claude 实例，并行处理不同模块 |

**Git 与 PR（8）**

| # | 技巧 | 说明 |
|---|---|---|
| 1 | 让 Claude 写 PR 描述 | "读 git diff，写 PR 描述，说清楚改了什么、为什么改" |
| 2 | squash merge 保持历史干净 | AI 的中间 commit 很乱，squash 后只保留最终结果 |
| 3 | feature 分支隔离 | 每个任务一个分支，Claude 的改动不影响 main |
| 4 | 用 Claude 做 code review | "读这个 PR 的 diff，按安全性、性能、可维护性三个维度审查" |
| 5 | commit 前让它自查 | "提交前检查：有没有遗留的 console.log、TODO、硬编码？" |
| 6 | 不要 amend 上一个 commit | Claude 有时会 `--amend` 覆盖你之前的提交，明确说"新建 commit" |

| # | 技巧 | 说明 |
| --- | --- | --- |
| 7 | 用 git stash 保护现场 | 让 Claude 改东西前先 `git stash`，不满意就 `git stash pop` 恢复 |
| 8 | 让 Claude 解决合并冲突 | "看看冲突文件，按业务逻辑解决冲突，保留两边的有效改动" |

**调试（10）**

| # | 技巧 | 说明 |
| --- | --- | --- |
| 1 | 让它跑命令看输出 | "跑 pnpm test 把失败的输出贴出来分析"比"测试挂了帮我修"好 10 倍 |
| 2 | 缩小范围再开问 | "只看 auth.ts 第 45–80 行"比"这个模块有 bug"好 |
| 3 | 用 git log 对比 | "看最近改动，对比改动前后，找哪次引入了 bug" |
| 4 | 先复现再修复 | "先写一个能复现这个 bug 的测试用例，再修复，确保测试从红变绿" |
| 5 | 二分法定位 | "用 git bisect 找到引入 bug 的 commit" |
| 6 | 看日志不要猜 | "查看 logs/error.log 最近 50 行，分析报错原因" |
| 7 | 加临时日志 | "在关键路径加 console.log 打印变量值，跑一次看输出" |
| 8 | 对比正常和异常 | "用户 A 正常、用户 B 异常，对比两个请求的差异" |
| 9 | 检查环境差异 | "本地能跑线上不行？对比环境变量、依赖版本、Node 版本" |
| 10 | 不要盲目修复 | "先给出 3 个可能的原因和排查方法，我确认后再改代码" |

**性能与成本（6）**

| # | 技巧 | 说明 |
| --- | --- | --- |
| 1 | 简单任务用 haiku | `claude --model haiku` 做简单任务，便宜 10 倍 |

| # | 技巧 | 说明 |
| --- | --- | --- |
| 2 | 批量任务用 headless | 脚本批量调用比交互模式省 token |
| 3 | 精准的 CLAUDE.md 省 token | 上下文越精准，AI 需要读的文件越少，成本越低 |
| 4 | 避免反复读大文件 | 告诉 Claude 具体行号范围，不要每次都读整个文件 |
| 5 | 用 `/compact` 释放上下文 | 长对话及时压缩，减少每轮的 token 消耗 |
| 6 | 监控用量 | 定期检查 API 用量，设置预算上限避免超支 |

> 更多技巧参考 claude-code-best-practice。

---

## 常见陷阱

| 陷阱 | 说明 | 解决 |
| --- | --- | --- |
| 上下文溢出 | 对话太长 AI 变笨 | 定期开新对话，用 CLAUDE.md 传递上下文 |
| 幻觉 API | AI 编造不存在的 API | 让它先查文档或 `grep` 确认 |
| 过度重构 | 你说修个 bug 它把整个文件重写了 | 明确说"只改这一处，不要重构" |
| 测试没跑 | AI 说"完成了"但没验证 | 用 verification skill 或 hook 强制验证 |
| 忽略错误处理 | 快速实现但没考虑异常 | 在提示词里明确要求错误处理 |

👉 **深度展开版**：Claude Code 陷阱合集 — 8 个真实踩坑场景，每个带症状 / 根因 / 出坑 / 预防

---

## 配置模板

直接复制到你的项目里用：

| 模板 | 用途 |
| --- | --- |
| CLAUDE.md | 项目配置文件模板，复制到项目根目录后按需修改 |
| settings.json | 权限配置模板，复制到 `.claude/settings.json` |

---

## 延伸阅读

- Claude Code 官方文档
- superpowers-zh — 20 个 AI 编程 skills
- agency-agents-zh — 211 个 AI 专家角色

# OpenAI Codex CLI 最佳实践

**简体中文** | English

> Codex CLI 是 OpenAI 官方的开源终端编程 Agent，用 Rust 写成，与 Claude Code、Gemini CLI 同属"终端 Agent 三巨头"。本文基于官方文档（developers.openai.com/codex）和仓库 openai/codex 的 v0.125.0（2026-04）核实整理。

> ⚠️ 注意：这里讲的是 2025 年发布的**新 Codex CLI**（开源、终端 Agent），不是 2023 年退役的旧 Codex 模型。同名产品系列还包括 **Codex App**（桌面端）和 **Codex Web**（云端 Agent，`chatgpt.com/codex`），本文只覆盖 CLI。

---

## 核心概念

| 概念 | 说明 | 用途 |
| --- | --- | --- |
| AGENTS.md | 项目/全局指令文件 | 类似 Claude Code 的 CLAUDE.md，启动时自动加载 |
| Approval Mode | 何时需要人来批 | `untrusted` / `on-request` / `never` |
| Sandbox Mode | 能动什么文件、能不能联网 | `read-only` / `workspace-write` / `danger-full-access` |
| Profile | 一组 config 的命名预设 | `--profile work` 一键切换模型/权限 |
| Subagent | 子 Agent，独立任务 | TOML 定义，可设置不同模型/权限 |
| Skill | 可复用工作流（SKILL.md） | 把重复任务封装成命名能力 |

| 概念 | 说明 | 用途 |
| --- | --- | --- |
| MCP Server | 外部工具接入 | STDIO 或 HTTP/OAuth |
| Plan Mode | 先规划再动手 | `/plan` 或 Shift+Tab 切换 |

---

## 快速上手

### 安装

```
# npm（推荐）
npm install -g @openai/codex

# 或 Homebrew（macOS）
brew install --cask codex

# 或下载 GitHub Release 二进制（macOS arm64/x64、Linux x64/arm64）
# https://github.com/openai/codex/releases/latest
```

系统要求（来自官方 install 文档）：macOS 12+、Ubuntu 20.04+/Debian 10+、Windows 11（WSL2），4 GB 内存（建议 8 GB），可选 Git 2.23+。

### 认证

```
codex            # 第一次启动，按提示选 "Sign in with ChatGPT"
```

两种方式：

- **ChatGPT 账号登录**（官方推荐）—— 走 ChatGPT Plus / Pro / Business / Edu / Enterprise 订阅额度，开浏览器走 OAuth。
- **API Key** —— 适合 CI、企业代理或想精确按量计费的场景，需要额外设置（详见 `developers.openai.com/codex/auth`）。

#### 第一次跑

```
cd /your/project
codex
```

进入 TUI 后试这几条（来自官方 quickstart）：

```
> Tell me about this project
> Find and fix bugs in my codebase with minimal, high-confidence changes
> Build a classic Snake game in this repo
```

> 💡 **官方建议**：执行任务前先 `git commit` 或建 checkpoint，Codex 改完不满意可以一键回滚。

#### AGENTS.md — 项目指令文件

在项目根目录建 `AGENTS.md`（或运行 `/init` 让 Codex 自动 scaffold）：

```
# 项目说明
Next.js 14 + TypeScript 电商后台。

# 目录约定
- src/app/api/       接口路由
- src/components/    UI 组件（PascalCase）
- src/lib/db/        Drizzle 数据访问层

# 构建 / 测试 / Lint
- 启动：pnpm dev
- 测试：pnpm test          （Vitest，必须在改完后跑）
- 类型：pnpm typecheck
- Lint：pnpm lint --fix

# 工程约束
- TS strict mode
- DB 操作只走 Drizzle，禁止裸 SQL
- 提交前自动 lint，不要绕过 husky

# 不要做的事
- 不要修改 src/legacy/
- 不要新增运行时依赖前不询问
```

加载顺序（重要）：

```
~/.codex/AGENTS.override.md    ← 临时全局覆盖
~/.codex/AGENTS.md             ← 个人全局
项目 git root/AGENTS.md        ← 团队共享
子目录/AGENTS.md               ← 模块级局部规则（覆盖父级）
```

文件按目录从浅到深拼接，越靠近当前目录的越优先生效。默认每个文件最多 32 KiB，可通过 `project_doc_max_bytes` 调。

---

## 提示词技巧

Codex 官方推荐"四元素"提示模板（来源：developers.openai.com/codex/learn/best-practices）：

```
Goal:        要做什么？要改什么？
Context:     涉及哪些文件、文档、错误信息？
Constraints: 约束有哪些？规范、架构、安全要求？
Done when:   完成判定条件是什么？
```

#### 1. 精确而不啰嗦

```
❌ 优化下登录功能
✅ Goal: 把 src/app/api/login/route.ts 的密码比对从 bcrypt.compareSync 换成异步 compare
   Context: 该接口被 src/components/LoginForm.tsx 调用，错误返回需保持 { code, message } 结构
   Constraints: 不动 schema、不加新依赖
   Done when: pnpm test 全绿，且并发 100 次登录无 event loop 阻塞警告
```

#### 2. 用 `@` 引文件而不是粘贴

```
@src/services/payment.ts @src/services/order.ts
读完这两个文件，分析支付流程的失败重试有什么漏洞，先给方案，确认后再改。
```

让 Codex 自己拉文件比你粘进 prompt 更省 token，也避免"截断到一半"。

#### 3. 推理强度按需调

复杂问题让 Codex "think harder"，简单任务用低推理省钱：

- low — 简单改动、风格调整
- medium / high — 大多数日常重构、Bug 修复
- xhigh — 架构级、多文件复杂逻辑

可在 TUI 里通过快捷键切换，或在 config.toml 里设默认。

#### 4. 先 Plan 后 Act

```
/plan
  分析 src/api/ 下所有路由，统一改成新的错误处理中间件
```

或者按 **Shift+Tab** 切到 Plan Mode。Codex 会先列计划、问澄清问题，确认后才写代码——多文件任务强烈建议用。

#### 5. 让它跑测试 + 自查

```
做完改动后：
1) 跑 pnpm test，把失败用例的输出贴给我
2) 跑 pnpm typecheck，确认没有新增类型错误
3) 用 /review 自查 diff，列出风险点
```

官方原话：*Don't stop at asking Codex to make a change. Ask it to create tests when needed, run checks, confirm results, and review work before accepting.*

---

## Approval & Sandbox（最重要的两个安全旋钮）

这是 Codex 最有特色的设计。两个维度独立配置：

#### Sandbox（能动什么）

| 模式 | 行为 |
| --- | --- |
| `read-only` | 只读，任何写/执行/联网都要批 |
| `workspace-write` ⭐ 默认 | 可读、可改 workspace 内文件、可跑常规命令；写 workspace 外或联网要批 |
| `danger-full-access` | 完全放开，**不推荐** |

技术实现（核实自 `codex-rs/cli/src/debug_sandbox.rs` 源码）：

- **macOS** — Seatbelt（系统自带）
- **Linux / WSL2** — **Landlock** 内核 LSM + seccomp 过滤（需要 Linux 5.13+ 且 Landlock 已启用）
- **Windows** — Restricted token sandbox（PowerShell 下生效）

> 提示：可以用 `codex sandbox seatbelt|landlock|windows -- <cmd>` 这组调试子命令，单独验证某条命令在沙箱里能不能跑。

#### Approval（什么时候问你）

| 模式 | 行为 |
| --- | --- |
| `untrusted` | 只有官方判定为安全的只读操作自动跑，其他都要批 |
| `on-request` ⭐ 默认 | 在 sandbox 内自动跑，越界（写外、联网）才问 |
| `never` | 永不询问（用于 CI / 脚本，配 sandbox 用） |

#### 常用组合

```
# 本地开发：直接 `codex` 即可（默认就是 workspace-write + on-request）
codex

# 显式指定（与上面等价）
codex --sandbox workspace-write

# 只想让它探一下、不要动手
codex --sandbox read-only

# 加几个 workspace 外的可写目录（不需要全开）
codex --add-dir ../sibling-repo --add-dir /tmp/scratch

# 在 CI / 脚本里非交互执行
codex exec --sandbox workspace-write --ask-for-approval never "跑测试并修复失败用例"

# 临时完全放开沙箱（明白后果再用）
codex --sandbox danger-full-access --ask-for-approval never

# 完全 YOLO：连询问都跳过、不沙箱（仅在外部已隔离环境下用，例如 Docker）
codex --yolo
# 等价于：codex --dangerously-bypass-approvals-and-sandbox
```

> ⚠️ **`--full-auto` 已废弃移除**（v0.125.0 仍在 `codex exec` 里保留作迁移提示）。直接 `--sandbox workspace-write` 就是同义；approval 默认本就是 `on-request`。
>
> 🧯 **CI 踩坑**：默认 approval 是交互式的，CI 里会卡死等输入超时。`codex exec` + `--ask-for-approval never` 是 CI 的标准姿势。

---

## 进阶用法

### 1. config.toml — 持久化偏好

`~/.codex/config.toml`（用户级）/ `.codex/config.toml`（项目级）：

```toml
model = "gpt-5.5"
approval_policy = "on-request"
sandbox_mode = "workspace-write"

[features]
web_search = "cached"      # 默认走缓存；--search 切 live
multi_agent = true

[profiles.review]
model = "gpt-5.5"
approval_policy = "untrusted"
sandbox_mode = "read-only"

[profiles.ci]
approval_policy = "never"
sandbox_mode = "workspace-write"

[mcp_servers.github]
command = "npx @modelcontextprotocol/server-github"
enabled = true
```

切换 profile：

```
codex --profile review              # 只读模式跑代码审查
codex exec --profile ci "..."       # CI 用脚本模式
```

### 2. Slash 命令速查

TUI 内常用：

| 命令 | 作用 |
| --- | --- |
| `/init` | 在仓库 scaffold 一份 AGENTS.md |
| `/model` | 切换模型 |
| `/plan` | 进入 Plan Mode |
| `/review` | 让 Codex 审查当前 diff / 分支 / 指定 commit |
| `/compact` | 压缩对话历史，释放 token |
| `/agent` | 在多个 subagent thread 间切换（`/multi-agents` 别名） |
| `/side` | 在临时 fork 里开支线对话，问完不污染主线 |
| `/permissions` `/approvals` | 临时调权限 |
| `/resume` `/fork` | 恢复 / 从某个时刻分叉对话 |
| `/new` `/clear` | 同 session 内开新对话 / 清屏重开 |
| `/rename` | 给当前 thread 改名 |
| `/diff` | 看 git diff（含未跟踪文件） |
| `/mention` | 把文件挂到对话里 |
| `/skills` | 列出 / 使用 Skill |
| `/memories` | 查看 / 生成 / 重置长期记忆 |
| `/mcp` | 检查 MCP server 状态（`/mcp verbose` 看详情） |

| 命令 | 作用 |
| --- | --- |
| `/plugins` `/apps` | 浏览插件 / 应用 |
| `/status` | 当前 session 详情、token 使用 |
| `/goal` | 设定 / 查看长任务的目标 |
| `/fast` | 切 Fast mode（`on` / `off` / `status`） |
| `/debug-config` | 打印配置层级与 requirements 诊断（**改 config.toml 不生效时用这个**） |
| `/feedback` | 给 OpenAI 发日志反馈 |

> 完整表（46 个）见源码 `codex-rs/tui/src/slash_command.rs::SlashCommand` 枚举。
> 在 TUI 里按 `/` 也会弹自动补全。

#### 3. 非交互执行（exec）

把 Codex 当脚本工具用：

```
# 单 prompt
codex exec "把所有 console.log 改成 logger.debug"

# 管道
git diff main..HEAD | codex exec "审查这份 diff，找出潜在 bug"

# 指定模型 + 不询问
codex exec -m gpt-5.5 --ask-for-approval never "为新文件补单测"

# 输出 JSONL（每行一个事件）给后续程序处理
codex exec --json "..." | jq -c .

# 把最后一条 message 单独写到文件（便于脚本提取最终结果）
codex exec -o /tmp/answer.txt "..."
```

#### 4. Subagent — 并行子任务

`.codex/agents/explorer.toml`：

```
name = "explorer"
description = "Read-only codebase explorer; gathers evidence before any change."
model = "gpt-5.3-codex"
sandbox_mode = "read-only"
developer_instructions = """
Stay in exploration mode.
Trace execution paths, cite files and symbols with file:line.
Never propose changes — just report findings.
"""
```

主 Agent 调用：

```
派一个 explorer subagent 把 src/api/ 所有路由的鉴权逻辑梳理一遍，回来汇总。
```

子 Agent 独立 sandbox + 模型，跑完返回主 thread。适合"探索 + 实施"分离的复杂改动。

#### 5. MCP — 外部工具

```
# CLI 直接添加
codex mcp add github --command "npx @modelcontextprotocol/server-github"
codex mcp list
```

支持 STDIO 和 HTTP/OAuth 两类 server。常用：GitHub、Linear、Slack、数据库（Postgres / Supabase）。原则：**只接能消除手工动作的工具**，不要凑数。

#### 6. Skill — 可复用方法论（注意：是文件夹，不是单文件）

把重复任务封装成一个 **Skill 目录**，Codex 看 description 决定何时触发，只在需要时才载入正文（"渐进披露"）：

```
~/.agents/skills/release-notes/      ← 个人（canonical 路径）
.codex/skills/release-notes/         ← 项目 Codex 专属
.agents/skills/release-notes/        ← 项目跨工具（Claude Code / Codex 等共享）
└── SKILL.md            # 必须：name + description + 主体步骤
└── references/         # 可选：长文档、规范、API schema
└── scripts/            # 可选：辅助脚本（lint / parser / generator）
└── examples/           # 可选：示范输入/输出
└── agents/openai.yaml # 可选：Codex 专属元数据（如限制权限）
```

> 注：`~/.codex/skills/` 也能用但已废弃（向后兼容），新装统一放 `~/.agents/skills/`。

调用方式（来自 shanraisshan 实战经验）：

```
$skill-creator                  # 用 $ 前缀显式触发某个技能
/skills                         # 列出所有可用技能
> 给我做这周 release notes     # 描述匹配会自动触发，无需 $
```

**写 Skill 的要点**（社区高 star 项目共识）：

- description 是触发器，不是摘要——写"什么时候应该 fire"而不是"这是什么"
- Skill 是给模型看的指引，不是给人看的文档——别罗列废话
- 给目标和约束，**别 railroad**（强制每一步怎么做），让模型自己决策路径
- 加一个 `## Gotchas` 段，把 Codex 在这个领域常踩的坑记下来——这是最高信号的内容

> 现成 Skill 库可以直接抄：ComposioHQ/awesome-codex-skills、VoltAgent/awesome-agent-skills（跨工具）

#### 7. Hooks — 在 Agent 循环里挂自定义脚本（beta）

启用 `[features] codex_hooks = true` 后，Codex 在 6 个事件点会调用你定义的 shell 脚本：

| 事件 | 触发时机 | 典型用途 |
| --- | --- | --- |
| `PreToolUse` | 工具调用前 | 危险命令拦截、参数校验 |
| `PermissionRequest` | 申请越界权限时 | 自动批/拒，根据规则 |
| `PostToolUse` | 工具调用后 | 自动 lint / format / 测试 |
| `SessionStart` | session 启动 | 加载项目特定上下文 |
| `UserPromptSubmit` | 用户提交 prompt 时 | 注入企业 disclaimer / 安全扫描 |
| `Stop` | session 结束 | 上传日志 / 清理临时文件 |

> 💡 hook schema 直接复用 Claude Code 的 `hooks.json` 格式（源码引擎名 `ClaudeHooksEngine`），所以从 Claude Code 迁移过来零改动。

配置文件 `.codex/hooks.json`，参考实现：shanraisshan/codex-cli-hooks。

最常见的两个 use case：

- **PostToolUse 跑 prettier/ruff** — Codex 改完代码自动格式化，避免 CI 红
- **PreToolUse 拦截 `rm -rf` / `git push --force` / 数据库 DROP** — 双保险

#### 8. Memories — 跨 session 的长期记忆（beta）

启用 `[features] memories = true`，Codex 会在 session 之间保留你确认过的事实（"这个项目用 pnpm，不要建议 npm"）。

```
/memories              # TUI 里查看 / 生成 / 重置
```

存在 `~/.codex/memories/`（per-user，**不是 per-project**）。安全开关（来自源码 `MemoriesToml` 结构）：

```
[memories]
disable_on_external_context = true    # 触发外部数据（MCP / web 搜索）时把当前 thread
标"polluted"，防记忆泄漏
                                      #  （旧别名 no_memories_if_mcp_or_web_search 仍
可用）
generate_memories = true              # 默认就是 true；置 false 则不再为新 thread 生成记
忆
use_memories = true                   # 默认 true；置 false 跳过把记忆注入提示
```

如果某个 session 接触了不可信内容（解析了陌生 PDF、跑了未审计的 MCP），跑 `/memories → Reset` 立即清掉。

#### 9. Plugins / Marketplace — 插件分发（v0.121.0+）

把 skills + apps + MCP servers 打包成可分发的 plugin（`.codex-plugin/plugin.json`）：

```
codex plugin marketplace add user/repo            # GitHub shorthand
codex plugin marketplace add ./local-marketplace  # 本地目录也行
codex plugin marketplace upgrade
codex plugin marketplace remove <name>

# 在 TUI 里
/plugins              # 浏览已装 / 可装的 plugin
```

社区 marketplace 索引：hashgraph-online/awesome-codex-plugins。

#### 10. Fast Mode — 1.5× 速度，2× 额度（gpt-5.4 限定）

```
/fast on            # 开启
/fast status        # 看当前
/fast off           # 关
```

适合"知道大致怎么改、就缺打字员"的场景。Pro 订阅还能用 `gpt-5.3-codex-spark` 做近实时的小改动迭代。

#### 11. Web 搜索

默认开启，cached 模式（OpenAI 预索引）。需要实时结果加 `--search`：

```
codex --search "React 19 useActionState 的最新最佳实践"
```

#### 12. 图片输入

```
codex -i screenshot.png "按这张设计稿改 src/components/Pricing.tsx"
```

PNG / JPEG，可在 TUI 里直接粘贴截图。配 Chrome DevTools / Playwright MCP，让 Codex 自己看浏览器控制台日志：

```
codex mcp add chrome-devtools --command "npx chrome-devtools-mcp"
codex mcp add playwright --command "npx @playwright/mcp"
```

#### 13. 本地开源模型（`--oss`）

源码 `utils/oss/src/lib.rs` 确认 Codex 原生支持两个本地 provider：

```
# Ollama 路线
codex --oss --local-provider ollama -m qwen2.5-coder
codex --oss --local-provider ollama -m deepseek-coder-v2

# LM Studio 路线
codex --oss --local-provider lmstudio
```

完全离线、零 API 成本。代价是模型能力比 GPT-5.5 弱不少，适合：本地敏感数据 / 没网环境 / 跑批量低复杂度任务。

> 模型 ID 跟着 Ollama / LM Studio 自己的命名（`ollama list` 看本地有什么），Codex 不维护独立清单。

#### 14. CI 集成（官方 GitHub Action）

`openai/codex-action` 是官方 Apache-2.0 Action，**自带受限沙箱代理**。最常用的 PR 自动 review 工作流：

```yaml
# .github/workflows/codex-review.yml
name: Codex PR review
on:
  pull_request:
    types: [opened, synchronize]
jobs:
  review:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      pull-requests: write
    steps:
      - uses: actions/checkout@v5
        with:
          ref: refs/pull/${{ github.event.pull_request.number }}/merge
      - run: git fetch --no-tags origin "${{ github.event.pull_request.base.ref
}}"
      - id: codex
        uses: openai/codex-action@v1
        with:
          openai-api-key: ${{ secrets.OPENAI_API_KEY }}
          prompt: |
            Review only the changes in PR #${{ github.event.pull_request.number
}}:
              git diff ${{ github.event.pull_request.base.sha }}...${{
github.event.pull_request.head.sha }}
            Be concise. Flag bugs, missing tests, and security risks.
      - if: steps.codex.outputs.final-message != ''
        uses: actions/github-script@v7
        with:
          script: |
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `${{ steps.codex.outputs.final-message }}`
            })
```

⚠️ **沙箱默认禁网**——如果你的 review prompt 需要跑测试 / 装依赖，先在 Action 步骤里 `npm ci` 再调 `codex-action`。

#### 15. Codex 作为 MCP server（反向接入）

```
codex mcp-server                    # 把 Codex 暴露成 MCP server
```

源码确认这会 expose `codex()` 和 `codex-reply()` 两个 MCP 工具。用法：让 Claude Code、Cursor 等其他 Agent 把 Codex 当成"我的同事"调用，做并行任务或交叉验证。

#### 16. Rules / Execpolicy（高级安全）

Starlark DSL 定义命令白/灰/黑名单（`prefix_rule()` + `host_executable()`）。规则文件放 `.codex/rules/*.rules`，可用 `codex execpolicy check` 离线验证：

```
prefix_rule(
    pattern = ["git", ["push", "force-with-lease"]],
    decision = "prompt",
    justification = "force pushes need a human eyeball",
)
prefix_rule(
    pattern = ["rm", "-rf", "/"],
    decision = "forbidden",
    justification = "absolutely not",
)
```

```
codex execpolicy check --rules .codex/rules/safety.rules git push --force
# {"matchedRules":[{...}],"decision":"prompt"}
```

适合企业 / 团队场景，比单纯 approval 更细粒度。详见 `codex-rs/execpolicy/README.md`。

---

## 高质量提示词与工作模式

来自 GitHub 高 star 实战教程（shanraisshan/codex-cli-best-practice 等）的高信号技巧：

**提示词层面**

- ✅ "证明给我看这能用" — 让 Codex 自己跑测试 + git diff main..HEAD 验证
- ✅ "用你现在掌握的所有信息，重新审视这个方案，给出更优雅的实现"
  （在一次平庸修复后用，能逼出更好的设计）
- ✅ Codex 自己调试能力强 — 把报错日志贴进去说"修"，别微管理过程

**Plan 层面**

- ✅ /plan 多文件任务先列计划 — Codex 也会自动 plan，但显式触发更可控
- ✅ 分阶段 + 每阶段 gated test —— 别让它一次改 30 个文件
- ✅ 让另一个 Codex（或 Claude Code）以 staff engineer 视角审查计划
- ✅ 写规格、降歧义 —— 越具体的输入越能给出可用的输出

**AGENTS.md 层面**

- ✅ 经验法则：随便一个新人能跑 codex → "run the tests" → 一次过
  不行就说明 AGENTS.md 漏了 build/test/setup 命令
- ✅ 控制在 150 行附近（硬上限是 32 KiB）—— 长 ≠ 好
- ✅ 行为约束（approval / sandbox / model）写 config.toml，不要写 AGENTS.md
- ✅ AGENTS.override.md 放个人偏好，别污染团队共享的 AGENTS.md

**多 Agent 与并行**

- ✅ 用 multi-agent 给问题扔更多算力 —— 把杂活外包给 subagent，主 thread 保持干净
- ✅ Test-time compute：一个 agent 写代码，另一个 agent 找 bug —— 独立 context 效果更好
- ✅ 长开发用 git worktree 给每个 agent 一个隔离的 working tree，避免互相覆盖

**调试**

- ✅ 让 Codex 在后台跑你想看日志的服务（codex 的 PTY exec 会正确处理）
- ✅ 卡住就截图扔给它（图片输入是最被低估的能力）
- ✅ Agentic search（glob + grep）打 RAG —— 代码漂移得太快，索引永远不准

---

## 常见踩坑（社区高频）

| 症状 | 根因 | 出坑 |
| --- | --- | --- |
| 启动直接弹浏览器要求登录 | token 过期 | 重跑 `codex` 走 OAuth；或 `codex login` |
| `429 Too Many Requests` | 触发速率限制 | 等额度恢复；脚本场景考虑切 API key + 限速 |
| CI 任务挂起超时 | 默认 approval 是交互式 | `codex exec --ask-for-approval never` |
| 改了 config.toml 不生效 | 文件路径错 / 配置层叠被覆盖 | TUI 里跑 `/debug-config` 看实际生效的配置层级和 requirements |
| `npm i -g` 报 `EACCES` | 全局目录归 root | 用 nvm/fnm 装 Node，不要 `sudo npm` |
| `which -a codex` 返回多行 | 装重了（npm + brew + binary） | 删多余的，shell rehash |
| 改完文件 Codex 不"看见" | 没在 git workspace 里 / 路径越界 | `codex --cd <project>` 指定根目录 |
| 长 thread 越来越慢 / 涨钱 | 上下文累积 | `/compact` 压缩；或 `/fork` 重开 |
| 多 agent 并行改同一文件冲突 | 主线 + subagent 同时改 | 给 subagent 用 git worktree |
| Linux 沙箱起不来 | 内核 < 5.13 或 Landlock 未启用 | 升级内核（`uname -r` 确认）；旧发行版临时用 `--sandbox read-only` 绕过 |

---

## Codex 与 Claude Code 怎么选

两者最像，但定位不同：

| 维度 | Codex CLI | Claude Code |
| --- | --- | --- |
| 厂商 | OpenAI | Anthropic |
| 开源 | ✅ Apache-2.0（Rust） | ❌ 闭源 CLI |
| 沙箱 | OS 内核级（Seatbelt / Landlock） | 应用层 + 26 个 Hook |
| 默认账号 | ChatGPT 订阅 | Claude API / Pro |
| 配置文件 | `AGENTS.md` + `config.toml` | `CLAUDE.md` + `settings.json` |
| Plan 模式 | `/plan` 或 Shift+Tab | `/plan` |
| Subagent | TOML 文件 | Markdown 文件 |
| MCP | ✅ | ✅ |
| 国内访问 | 需要 ChatGPT Pro / API key | 需要 Claude API |
| 强项场景 | 终端任务、CI 集成、token 效率 | 大型重构、跨文件依赖、代码风格 |

**经验法则**（社区共识，参考 builder.io / datacamp 等多篇对比）：

> *Codex for keystrokes, Claude Code for commits.* Codex 适合"快速迭代 + 沙箱执行"，Claude Code 适合"一次改 12 个文件且依赖图复杂"的精修。

预算敏感且已订阅 ChatGPT → Codex 就是顺手白送的；做大重构、要最强代码质感 → Claude Code 仍是首选。两者并用也很常见。

---

## 模板与配置

`templates/` 目录提供：

- `AGENTS.md` — 通用项目指令模板

- `config.toml` — 用户级配置模板（含 profiles）
- `agent-explorer.toml` — 只读探索 subagent 模板
- `SKILL.md` — Skill 模板（含 Gotchas / 触发器写法范本）

---

## 参考资料

#### 官方

- 仓库：openai/codex（Apache-2.0，Rust）
- 文档：developers.openai.com/codex
- 安装详解：docs/install.md
- 最佳实践：Best practices
- 沙箱原理：Sandboxing
- Slash 命令：Slash commands
- 官方 GitHub Action（CI 集成）：openai/codex-action
- 官方 Skills 目录：openai/skills

#### 社区高质量索引（高 star，每条都核实过）

- **RoggeOhta/awesome-codex-cli** — 280+ 资源大全（Subagent / Skill / Plugin / MCP / IDE 集成 / CI），按类别整理
- **shanraisshan/codex-cli-best-practice** — 50 条战场验证的提示词技巧 + 完整 .codex/ 实现样例（已对齐 v0.125.0）
- **ComposioHQ/awesome-codex-skills** — 38 个常用 Skill（开发工具 / 数据分析 / Composio 1000+ SaaS 集成）
- **VoltAgent/awesome-codex-subagents** — 136+ subagent 跨 10 个领域（开发 / 安全 / 基建 / 数据 / DX）
- **hashgraph-online/awesome-codex-plugins** — 第一个 Plugin marketplace 索引
- **agents.md** — 跨工具 AGENTS.md 标准（已被 60k+ 项目采用，Codex / Claude Code / Gemini CLI 通用）

### 主流 Workflow 框架

| 框架 | star | 流程 |
| --- | --- | --- |
| Superpowers | 171k | 头脑风暴 → 写计划 → subagent 驱动 → TDD → review → 合分支 |
| Spec Kit | 92k | `/speckit.constitution → specify → plan → tasks → implement` |
| oh-my-codex | 27k | `$deep-interview → $ralplan → $ralph` |
| Compound Engineering | 16k | `/ce-ideate → brainstorm → plan → work → review → compound` |

### 与 Claude Code 对比

本项目内：workflows/tool-selection.md

> 信息核实截止：**2026-04-30**（codex CLI v0.125.0）。本教程的每条 CLI flag、子命令、slash 命令、config 字段都对照 `codex-rs` 源码核实过；模型能力和定价变化快，最终以 OpenAI 官方文档为准。

# 卷二 · 工作流：实战进阶

## 这一卷是给谁看的

- 已经能用 Claude Code 或 Cursor 完成日常任务，但效率"还能更高"
- 想把多个 AI 工具组合用，不知道怎么分工
- 想写自己的 Skill / Hook / Subagent，让 AI 越用越懂你的项目
- 经常被 AI "改了 12 个文件然后 5 个出 bug"——想知道怎么避免

**和卷一的区别**：卷一教你"怎么让单个工具替你干活"，卷二教你"怎么让一组工具+方法论替你干一整个工作流"。从"我会用"升级到"我用得快、用得稳、用得放心"。

## 这一卷的逻辑顺序

```
代码审查 + 调试 + 测试（三大执行型方法论）
    ↓
多工具协作（选型 / 实战脚本 / 协作模式）
    ↓
踩坑合集（4 款主流工具的 31 个深度陷阱）
    ↓
进阶工具（Aider / Gemini CLI / Windsurf / Trae / Kiro）
```

学完这一卷你应该能：

- 让 AI 系统化做代码审查，不是单点"看一眼"
- 用 AI 做端到端调试：从复现 → 定位 → 修复 → 写防回归测试，一条龙
- 设计 Claude Code + Cursor 这种双工具流水线，明确各自职责边界
- 看到典型陷阱描述就能心里咯噔一下（"这个我也遇到过"）并知道怎么躲
- 在小众但有特定优势的工具（如 Gemini CLI 的 2M 上下文）上做出选择

## 这一卷的"必读"

- **调试方法论** 章节——最被低估的能力，95% 用户从来没让 AI 系统化调过 bug
- **踩坑合集** 章节——按四段式（症状 / 根因 / 出坑 / 预防）写的，每条都是真血泪

## 节奏建议

不要顺读到底。建议这样：

1. 先快速过一遍**踩坑合集**——你大概率正在踩其中某条，立刻就能止损
2. 然后回头读你最缺的方法论（多数人是"调试"或"测试"）
3. **多工具协作** 那几章按需读——只用 1 个工具的话可以暂时跳过
4. **进阶工具** 是字典——遇到"诶 Gemini CLI 不是免费吗"再翻进去

# AI 辅助代码审查

> 让 AI 做代码审查不是"帮我看看有没有 bug"这么简单。好的 AI 代码审查需要明确审查维度、设定标准、分级输出。

---

## 审查维度

让 AI 审查代码时，明确告诉它关注什么：

| 维度 | 关注点 | 优先级 |
| --- | --- | --- |
| **安全** | 注入、XSS、敏感数据泄露、权限绕过 | 🔴 最高 |
| **正确性** | 逻辑错误、边界情况、并发问题 | 🔴 高 |
| **性能** | N+1 查询、内存泄漏、不必要的计算 | 🟡 中 |
| **可维护性** | 命名、结构、重复代码、复杂度 | 🟢 低 |
| **测试** | 覆盖率、边界测试、Mock 合理性 | 🟡 中 |

---

## 审查提示词模板

### 安全审查

```
审查 src/api/ 下所有接口的安全性：
1. SQL 注入：参数有没有直接拼接到 SQL？
2. XSS：用户输入有没有未转义直接输出？
3. 权限：每个接口有没有做权限校验？
4. 敏感数据：密码、token 有没有在日志里打印？
5. 速率限制：高频接口有没有限流？

按风险等级标注：🔴 严重 / 🟡 中等 / 🟢 低
```

### PR 审查

```
审查这个 PR 的改动（git diff main...HEAD）。
重点关注：
1. 改动是否符合 PR 描述的目标
2. 有没有遗漏的边界情况
3. 有没有引入安全风险
4. 测试是否充分

输出格式：
- [必须修改] 问题描述 + 建议
- [建议修改] 问题描述 + 建议
- [可选优化] 问题描述 + 建议
```

### 性能审查

```
审查 src/services/report.ts 的性能：
1. 有没有 N+1 查询
2. 有没有可以缓存但没缓存的计算
3. 有没有不必要的全表扫描
4. 大数据量下会不会 OOM
5. 有没有阻塞操作应该改成异步

给出每个问题的具体改进方案。
```

---

## 审查输出规范

好的审查输出应该是这样的：

```
## [必须修改] SQL 注入风险
📍 src/api/users.ts:45
问题：用户输入的 name 直接拼接到 SQL 查询
建议：改用参数化查询
```

```
## [建议修改] 缺少错误处理
📍 src/services/payment.ts:78-82
问题：API 调用没有 try-catch，网络异常会导致未处理的 Promise rejection
建议：加 try-catch，失败时返回明确的错误信息
```

```
## [可选优化] 重复代码
📍 src/api/users.ts:20-35 和 src/api/orders.ts:15-30
问题：两处的分页逻辑完全一样
建议：提取一个 buildPagination 工具函数
```

---

## 团队审查文化

如果 AI 审查结果要发到 PR comment，注意表达方式：

```
❌ "This is wrong. You should use X."
✅ "[建议修改] 这里用同步方式读文件是出于什么考虑？
    如果并发量上来，可能会阻塞事件循环。
    建议考虑 async 方案。"
```

用"提问"代替"否定"，用"建议"代替"命令"。关键安全问题标 `[必须修改]`，不因客气而放过。

# AI 辅助调试方法论

> 大多数人用 AI 调试的方式是：贴报错 → AI 猜一个方案 → 不行再猜。这是最低效的方式。系统化的 AI 调试应该是：收集证据 → 分析根因 → 验证假设 → 精确修复。

---

## 四阶段调试法

### 第一阶段：收集证据

**不要让 AI 猜。让它看。**

```
# 让 AI 看错误日志
跑一下 pnpm test src/api/users.test.ts，把完整输出贴出来。

# 让 AI 看最近改动
git log --oneline -10 看看最近改了什么。
git diff HEAD~3 看看最近三次提交改了哪些文件。

# 让 AI 看堆栈
看 src/services/payment.ts 第 78 行附近的代码，
这是堆栈指向的位置。
```

### 第二阶段：分析根因

```
基于以下证据分析可能的原因：
1. 报错信息：[贴出来]
2. 最近改动：[相关 commit]
3. 相关代码：[文件和行号]

列出 2-3 个最可能的原因，按可能性排序。
不要直接给修复方案。
```

#### 第三阶段：验证假设

```
你认为原因是 [假设]。
怎么验证这个假设?
给我一个最小的验证方案（加个 console.log / 写个测试 / 改个参数）。
```

#### 第四阶段：精确修复

```
根因确认是 [原因]。
现在给出修复方案。
要求:
1. 只改必要的代码
2. 不要顺手重构
3. 加一个测试覆盖这个 case
4. 修完跑一下测试验证
```

---

## 常见场景

#### 测试失败

❌ 差：测试挂了帮我修
✅ 好：跑 `pnpm test -- --reporter=verbose`，
把失败的测试用例和错误信息贴出来。
然后看对应的源码和测试代码，
分析是源码的 bug 还是测试写错了。

#### 生产环境报错

```
线上报错：[错误信息]
影响范围：[哪些用户/功能]

请按以下步骤排查:
1. 看 git log 最近 24 小时的部署
2. 看报错堆栈定位到代码位置
3. 分析是代码 bug、配置问题还是外部依赖问题
4. 给出临时修复方案（快速止血）和根本修复方案
```

**性能问题**

```
[接口/页面] 响应时间从 200ms 变成了 3s。

排查步骤：
1. 看是不是数据库慢查询（检查 SQL 日志）
2. 看是不是 N+1 查询问题
3. 看是不是外部 API 调用变慢
4. 看是不是最近的代码改动引入了性能回归

用排除法，一个一个确认。
```

---

## 反模式

| 反模式 | 为什么不好 | 正确做法 |
| --- | --- | --- |
| 贴报错就让 AI 猜 | AI 的猜测成功率 < 30% | 先收集证据再分析 |
| 一个方案不行换另一个 | 没找到根因就换方案 = 碰运气 | 验证假设，确认根因 |
| 让 AI 改了不测试 | 可能引入新 bug | 改完必须跑测试验证 |
| 同一个 bug 循环 5 次 | 说明方向错了 | 停下来重新分析，换个角度 |

# 测试策略

> 用 AI 写测试是投入产出比最高的场景之一。AI 擅长生成重复性高的测试代码，但你需要告诉它"测什么"和"怎么测"，否则它会写一堆没用的测试。

---

## 核心原则

#### 1. 先有测试策略，再让 AI 写

❌ 差：给这个文件写测试
✅ 好：给 src/services/order.ts 写单元测试。
重点覆盖：
- createOrder：库存不足时应该抛异常
- calculateTotal：折扣叠加的边界情况
- cancelOrder：已发货的订单不能取消
用 Vitest，mock 掉数据库层。

#### 2. 告诉 AI 测试的边界条件

AI 很擅长写 happy path 测试，但容易忽略边界情况。明确提出来：

```
给 parseDate 函数写测试，覆盖：
- 正常输入："2024-01-15"
- 空字符串
- null / undefined
- 非法格式："not-a-date"
- 闰年日期："2024-02-29"
- 时区边界：UTC 午夜前后
```

#### 3. 不要让 AI 测实现细节

```
❌ 差：测试 getUserList 内部调用了 db.query 几次
✅ 好：测试 getUserList 返回正确的用户列表，
      分页参数生效，空结果返回空数组
```

测行为，不测实现。否则重构就要改一堆测试。

---

## 按测试类型的最佳实践

#### 单元测试

```
给 src/utils/validator.ts 的所有导出函数写单元测试。

要求：
1. 每个函数至少 3 个测试用例（正常、边界、异常）
2. 用 describe 按函数分组
3. 测试名用中文描述行为，如 "空字符串应该返回 false"
4. 不要 mock 纯函数的依赖
```

#### 集成测试

```
给订单创建流程写集成测试。
测试范围：Controller → Service → Repository
用测试数据库（不要 mock 数据库），每个测试前清空数据。

场景：
1. 正常创建订单 → 库存减少 → 返回订单号
2. 库存不足 → 返回 400 → 库存不变
3. 并发创建 → 不能超卖
```

#### API / E2E 测试

```
给 POST /api/auth/login 写接口测试。

测试：
1. 正确账号密码 → 200 + token
2. 密码错误 → 401
3. 账号不存在 → 401（不要泄露"账号不存在"）
4. 缺少字段 → 400 + 具体缺哪个字段
5. 连续 5 次错误 → 429 限流
```

---

## 让 AI 补测试的高效流程

#### 流程一：先写代码，再补测试

```
1. 实现功能代码
2. 让 AI 读代码，生成测试
3. 跑测试，看有没有失败的
4. 失败的测试 = 可能的 bug，让 AI 分析是代码问题还是测试问题
```

#### 流程二：TDD — 先写测试，再写代码

```
1. 描述需求，让 AI 先写测试用例（不写实现）
2. 确认测试用例覆盖了你要的行为
3. 让 AI 写实现代码，跑测试直到全部通过
4. 重构（有测试兜底，放心改）
```

TDD 模式特别适合 AI 编程，因为测试就是明确的"完成标准"。

#### 流程三：用测试做安全网

```
要重构 src/services/payment.ts。
先给它写完整的测试覆盖现有行为，
跑通后再开始重构。
每改一步跑一次测试。
```

---

## 常见反模式

| 反模式 | 问题 | 正确做法 |
| --- | --- | --- |
| 让 AI 一次生成所有测试 | 数量多但质量低，很多无意义的断言 | 按模块逐个生成，每个都审查 |
| 只测 happy path | 边界和异常情况没覆盖 | 明确列出边界条件 |
| Mock 一切 | 测试和真实行为脱节 | 只 mock 外部依赖（网络、数据库），纯逻辑不 mock |
| 复制粘贴测试 | 大量重复代码，维护成本高 | 用 `test.each` 或参数化测试 |
| 不看测试就提交 | AI 可能断言了错误的期望值 | 每个断言都确认是否正确 |
| 追求 100% 覆盖率 | getter/setter 也要测，浪费时间 | 关注核心逻辑覆盖率，不追求数字 |

---

## 各工具的测试技巧

| 工具 | 测试技巧 |
| --- | --- |
| Claude Code | 直接说"跑测试看结果"，它会自动执行并修复失败用例 |
| Cursor | 选中函数 → `Cmd+K` → "写单元测试"，快速生成 |
| Copilot | 打开测试文件，写 `describe('...')`，补全会自动生成用例 |
| Aider | 配置 `auto-test: true`，每次改代码自动跑测试 |
| Kiro | Spec 里定义测试用例，实现时自动验证 |

# 多工具选型指南

> 不同的 AI 编程工具擅长不同的事。选对工具，效率翻倍；选错工具，事倍功半。

---

## 一张表选工具

| 场景 | 推荐工具 | 原因 |
| --- | --- | --- |
| **日常编码**（补全、小修改） | Cursor / Copilot | IDE 集成，按 Tab 就行 |
| **复杂重构**（跨文件、架构级） | Claude Code | Agent 能力最强，理解整个项目 |
| **新项目搭建** | Claude Code | 从零开始需要全局规划能力 |
| **Bug 调试** | Claude Code | 能看日志、跑命令、系统化分析 |
| **代码审查** | Claude Code / Cursor | Claude Code 更系统，Cursor 更快捷 |
| **学习新代码库** | Cursor + Chat | 选中代码问问题，交互最自然 |
| **写测试** | Claude Code | 能跑测试、看覆盖率、自动修复 |
| **文档/注释** | Copilot | 行内补全写注释最顺手 |
| **前端 UI 调整** | Cursor | 实时预览 + 可视化编辑 |
| **命令行/脚本** | Claude Code / Gemini CLI | CLI 原生，适合终端工作流 |
| **大代码库探索** | Gemini CLI | 上下文窗口最大（2M tokens） |
| **CI / 自动化代码审查** | Codex CLI | 官方 GitHub Action 内置受限沙箱代理 |

| 场景 | 推荐工具 | 原因 |
| --- | --- | --- |
| **已订阅 ChatGPT Plus/Pro** | Codex CLI | 订阅自带额度，开箱即用，OS 内核级沙箱 |
| **完全离线 / 零 API 成本** | Codex CLI 或 Aider | `codex --oss --local-provider ollama` 走本地模型 |
| **企业级命令安全策略** | Codex CLI | Starlark DSL `prefix_rule()`，比 approval 更细粒度 |

---

## 工具能力对比

| 能力 | Claude Code | Codex CLI | Cursor | Copilot | Gemini CLI |
| --- | --- | --- | --- | --- | --- |
| 代码补全 | — | — | ★★★ | ★★★ | — |
| 对话式编程 | ★★★ | ★★★ | ★★☆ | ★★☆ | ★★★ |
| Agent 自主执行 | ★★★ | ★★★ | ★★☆ | ★★☆ | ★★☆ |
| 终端命令执行 | ★★★ | ★★★ | — | — | ★★★ |
| 多文件协调 | ★★★ | ★★☆ | ★★☆ | ★★☆ | ★★☆ |
| 项目规则配置 | ★★★ | ★★★ | ★★★ | ★★☆ | ★★☆ |
| 扩展能力（MCP） | ★★★ | ★★★ | ★★☆ | ★★☆ | ★★☆ |
| 沙箱级别 | App 层 +Hook | OS 内核 | — | — | — |
| 本地模型支持 | — | ★★★（--oss） | — | — | — |
| 上下文窗口 | ★★☆ | ★★☆ | ★★☆ | ★★☆ | ★★★ |
| IDE 集成 | — | — | ★★★ | ★★★ | — |

| 能力 | Claude Code | Codex CLI | Cursor | Copilot | Gemini CLI |
| --- | --- | --- | --- | --- | --- |
| 免费额度 | ★☆☆ | ★★☆（含 ChatGPT 订阅） | ★☆☆ | ★★☆ | ★★★ |

---

## 推荐组合

**组合一：Claude Code + Cursor（最流行）**

```
Claude Code → 架构设计、复杂重构、调试、测试
Cursor      → 日常编码、小修改、UI 调整、快速问答
```

适合：全栈开发者，前后端都写

**组合二：Claude Code + Copilot（轻量）**

```
Claude Code → Agent 任务（重构、生成、调试）
Copilot     → 行内补全、写注释、简单问答
```

适合：后端开发者，VS Code 用户

**组合三：Gemini CLI + Cursor（预算友好）**

```
Gemini CLI → 大规模分析、代码探索（免费）
Cursor     → 日常编码、交互式修改
```

适合：个人开发者，想控制成本

**组合四：Codex CLI + Cursor（ChatGPT 订阅者）**

```
Codex CLI → Agent 任务、CI 自动审查、内核沙箱执行
Cursor    → 日常编码、Tab 补全
```

适合：ChatGPT Plus/Pro 订阅者，想让订阅额度产生 Agent 价值； 特别适合需要内核级沙箱（Seatbelt / Landlock）的安全敏感场景。

#### 组合五：Codex CLI + Claude Code（双 CLI）

```
Codex CLI    → 快速迭代、CI/脚本、token 效率高的小改动
Claude Code  → 跨 12 个文件的大重构、依赖图复杂的精修
```

适合：高强度全职开发，"keystrokes vs commits" 双管齐下； 社区共识：*Codex for keystrokes, Claude Code for commits.*

---

## 切换策略

什么时候从一个工具切到另一个：

| 信号 | 动作 |
| --- | --- |
| Cursor 补全改了 3 次还不对 | 切 Claude Code，让它系统分析 |
| Claude Code 对话太长开始变笨 | 开新对话，或切 Cursor 做剩余小任务 |
| 需要看实时效果 | 切 Cursor / Copilot（IDE 内预览） |
| 需要跑命令验证 | 切 Claude Code / Gemini CLI（终端内） |
| 需要理解大代码库 | 切 Gemini CLI（上下文最大） |
| 要在 CI 跑非交互 Agent | 切 Codex CLI（`codex exec --json` + 官方 Action） |
| 处理敏感数据 / 离线环境 | 切 Codex CLI（`--oss --local-provider ollama`） |
| 需要内核级沙箱保证 | 切 Codex CLI（Seatbelt / Landlock） |

# 实战场景：对话脚本示范

> 前面的教程讲"怎么用"，这里讲"完整跑一遍"。三个常见场景的端到端对话脚本，可以直接拿去改。
>
> 脚本里用 `pnpm` 作为示例，把它换成你自己的包管理器（npm/yarn/bun）即可。

---

## 场景一：用 Claude Code 重构一个复杂模块

目标：把耦合严重的支付模块拆成可维护的结构。直接让 AI 一把梭经常跑偏，分三步走。

**第一步：先摸清现状**

```
读 src/services/payment/ 下所有文件，做三件事：
1. 画出调用关系（谁调用谁）
2. 列出当前问题（重复代码、循环依赖、缺少测试等）
3. 不要改任何代码，输出到 docs/payment-analysis.md

先做，做完等我确认。
```

**第二步：要方案，不要代码**

```
基于上面的分析，给出 2 个重构方案。
每个方案说清楚：
- 核心思路（一句话）
- 改动范围（哪些文件会动，估算代码行数）
- 优缺点各 2-3 条
- 风险点

不要开始写代码，等我选方案。
```

#### 第三步：按方案分步执行

```
按方案 1 重构，分四个阶段做，每个阶段完成后暂停：

① 抽取公共的 PaymentGateway 接口
② 把 WechatPay / AlipayPay 改成实现类
③ 给每个实现类加单元测试（覆盖率 > 80%）
④ 加端到端集成测试

要求：
- 每个阶段做完跑 pnpm test 验证不破坏现有功能
- 每个阶段 commit 一次，commit message 说清楚改了什么
- 有疑问先问我再改
```

**关键点**：第一步"不要改代码"、第二步"不要写代码"、第三步"每阶段暂停"，是阻止 AI 失控的三道闸。

---

## 场景二：Claude Code + Cursor 协作开发新功能

目标：开发一个订单导出功能。Claude Code 做设计和骨架，Cursor 填实现细节。

#### 第一步：Claude Code 做架构设计

```
需求：后台管理系统加订单导出功能，支持 CSV/Excel，
按时间范围 + 订单状态筛选，大数据量流式导出（避免 OOM）。

分析需求，输出到 docs/design/order-export.md：
1. 数据库查询方案（分页 or 游标）
2. API 设计（路由、参数、响应格式）
3. 前端交互（按钮位置、进度反馈）
4. 核心技术选型（CSV 用什么库、Excel 用什么库）

只写文档，不写代码。
```

**第二步：Claude Code 生成骨架**

```
按 docs/design/order-export.md 生成代码骨架：
- src/app/api/orders/export/route.ts
- src/services/order-export.service.ts
- src/components/order-export-dialog.tsx

每个文件只写接口签名和 TODO 注释，不要实现具体逻辑。
类型定义要完整，方便后面 IDE 补全。
```

**第三步：切 Cursor 填实现**

在 Cursor 里打开骨架文件，用 Composer 逐个填充。以 `order-export.service.ts` 为例，Composer 里输入：

```
@order-export.service.ts 按 TODO 实现每个方法。
@docs/design/order-export.md 参考设计文档。
用游标分页，每批 1000 条，避免内存爆炸。
```

**第四步：回 Claude Code 跑测试和审查**

```
刚才 Cursor 填完了实现，现在你来审查：

1. 跑 pnpm typecheck 和 pnpm test，把报错贴出来修掉
2. 按 common/code-review.md 的五个维度审查（安全 > 正确性 > 性能 > 可维护性 > 测试）
3. 检查是否和 docs/design/order-export.md 对齐
4. 最后生成 PR 描述，写清楚改动和测试范围
```

**分工原则**：架构决策给 Claude Code（全局能力强），实现细节给 Cursor（IDE 内交互快），最后回 Claude Code 做系统性验证。

---

## 场景三：给老项目补测试

目标：一个没测试的老项目，用 AI 快速把覆盖率拉起来。

**一次性给出完整指令**

```
给 src/services/ 下的代码补测试，按这个流程做：

优先级（按这个顺序做）：
1. 核心业务：payment/、order/、auth/
2. 外部依赖：database/、api-client/
3. 工具函数：utils/、helpers/

每个文件的流程：
1. 先读代码，列出关键路径和边界条件（输出一个 checklist）
2. 让我看 checklist，我确认后再写测试
3. 用 Vitest，测试文件放在 __tests__/ 或 *.test.ts
4. 每个文件测完跑 pnpm test 确认通过
5. commit 一次，message 写清楚加了哪个文件的测试

质量要求：
- 覆盖率目标 80%，但不追求 100%（getter/setter 不测）
- 测行为，不测实现（不要 mock 所有内部方法）
- 边界条件要测：空值、超长、并发、异常输入

现在从 src/services/payment/ 开始。
```

**卡住的时候**

AI 常见的问题和对应的打断话术：

| 症状 | 打断话术 |
| --- | --- |
| 测试全是 mock，没验证真实逻辑 | "这些测试 mock 太多了，改成集成测试，用内存数据库" |
| 测试过多、每个函数都测 | "只测公开 API 和关键路径，私有方法不要单独测" |
| 测试跑挂了但 AI 不管 | "跑 pnpm test 把失败贴出来，我们先修挂的再继续" |
| 一次改太多文件 | "停，先把当前文件的测试跑通再开下一个" |

---

## 通用打断话术

这些话随时用得上，建议收藏：

> 停，先别写代码。把你的理解和计划说一遍，我确认再继续。

> 扔掉这个方案，用你刚才了解到的信息重新设计。

> 只改这一个文件，不要动其他文件，不要加新依赖。

> 跑一下 `pnpm test`，把失败输出贴出来分析，不要猜。

> 给我 3 个可能的原因和排查方法，我确认后再动手。

# Claude Code + Cursor 协作工作流

> 这是目前最流行的 AI 编程组合。Claude Code 负责"重活"（架构、重构、调试、测试），Cursor 负责"日常"（编码、UI、快速修改）。

---

## 分工原则

| 任务 | 用 Claude Code | 用 Cursor |
| --- | --- | --- |
| 项目初始化 / 搭架子 | ✅ | |
| 架构设计 / 重构 | ✅ | |
| 写测试 / 跑测试 | ✅ | |
| Bug 调试（需要看日志） | ✅ | |
| Git 操作（提交、PR） | ✅ | |
| 日常编码 / 新功能实现 | | ✅ |
| UI 调整 / 样式修改 | | ✅ |
| 代码理解 / 学习 | | ✅ |
| 小范围修改（改个参数、加个字段） | | ✅ |

---

## 典型流程

### 新功能开发

```
阶段 1：设计（Claude Code）
> 给项目加一个通知系统，支持站内信和邮件。
  先分析现有架构，给出设计方案。

阶段 2：骨架代码（Claude Code）
> 按确认的方案，创建所有需要的文件和基础代码结构。
  包括数据模型、Service 接口定义、API 路由。

阶段 3：实现细节（Cursor）
打开 Cursor，在 Claude Code 创建的骨架上填充实现。
用 Composer 模式逐个文件实现业务逻辑。

阶段 4：测试（Claude Code）
> 给通知模块写完整的测试。
  单元测试 + 集成测试。
  跑一遍确保全部通过。

阶段 5：代码审查（Claude Code）
> 审查整个通知模块的代码质量和安全性。
```

### Bug 修复

```
阶段 1：定位（Claude Code）
> 用户反馈"发送通知后收不到邮件"。
  看日志、看最近改动、分析根因。

阶段 2：修复（Cursor 或 Claude Code）
简单修复 → Cursor（选中代码直接改）
复杂修复 → Claude Code（需要跨文件协调）

阶段 3：验证（Claude Code）
> 跑一下通知模块的测试，确保修复有效且没引入新问题。
```

---

## 配置同步

两个工具的项目配置保持一致：

```
# CLAUDE.md 和 .cursorrules 写相同的核心规则
# 或者用 superpowers-zh，它同时支持两个工具

cd /your/project
npx superpowers-zh  # 自动检测并安装到 .claude/ 和 .cursor/
```

---

## 注意事项

1. **不要两个工具同时改同一个文件** — 会冲突
2. **Claude Code 改完后在 Cursor 里刷新** — 文件已在磁盘上更新
3. **Git 是协调枢纽** — 两个工具的改动都通过 Git 管理

# Claude Code + Copilot 协作工作流

> Claude Code 做 Agent 级任务，Copilot 做行内补全。两个工具分工清晰，不冲突——一个在终端里，一个在编辑器里。

---

## 分工原则

| 任务 | 用 Claude Code | 用 Copilot |
| --- | --- | --- |
| 跨文件重构 | ✅ | |
| 写测试套件 | ✅ | |
| 调试（看日志、跑命令） | ✅ | |
| 架构设计 | ✅ | |
| 行内代码补全 | | ✅ |
| 写注释 / 文档 | | ✅ |
| 小函数实现 | | ✅ |
| 代码理解（选中问问题） | | ✅ |

---

## 典型流程

```
1. Claude Code 创建文件骨架
   > 按设计方案创建 src/services/notification/ 下的所有文件

2. VS Code + Copilot 填充实现
   打开 Copilot 创建的文件，写代码时 Copilot 自动补全

3. Claude Code 跑测试和审查
   > 跑测试看有没有问题，然后做一次代码审查

4. VS Code + Copilot 做小修补
   根据审查意见在编辑器里快速修改
```

---

## 优势

- **无需切换** — Claude Code 在终端，Copilot 在 VS Code，各干各的
- **成本控制** — Copilot 包月制不限量，大量补全不额外花钱
- **互补性强** — Claude Code 不擅长行内补全，Copilot 不擅长 Agent 任务

# 陷阱合集

> 每个工具 README 里都有"常见陷阱"速查表，本目录把它们**深度展开**：每个陷阱按 **症状 → 根因 → 出坑 → 预防** 四段式写清楚。
>
> 目标：不是"列个清单凑数"，而是帮你**下次不踩同样的坑**。

---

## 按工具索引

| 工具 | 陷阱数 | 覆盖主题 |
|---|---|---|
| Claude Code | 8 | 上下文溢出 / 幻觉 API / 过度重构 / 测试谎报 / CLAUDE.md 被忽略 / git amend / Plan 漂移 / Subagent 传话 |
| Cursor | 8 | Composer 脱缰 / Tab 暴吐 / .cursorrules 超长 / @file 失效 / Chat 模式混用 / 模型切换 / Notepads 过期 / 索引陈旧 |
| GitHub Copilot | 8 | 过期 API / Agent 读 .env / instructions 太长 / #file vs @workspace / MCP 失效 / IDE 差异 / 免费限流 / 自定义角色不生效 |
| Aider | 7 | auto-commit 吃变更 / /add 漏依赖 / 模型切换质量塌 / lint 循环烧 token / Map 过期 / Architect→Code 漂移 / amend 失控 |

---

## 其他工具陷阱

Windsurf / Gemini CLI / Kiro / Trae / OpenClaw 的陷阱页还没写。欢迎：

1. 看你用的工具有没有踩过坑
2. 按下面模板写一篇 PR 过来

---

## 写新陷阱页的模板

文件命名：`pitfalls/<tool-name>.md`（中文）+ `pitfalls/<tool-name>.en.md`（英文对照）

结构：

```
# <Tool> 陷阱合集

> 一句话定位：这个工具独特的痛点在哪

## 陷阱 1：一句话标题

**症状**
你看到的异常现象（越具体越好）

**根因**
为什么会发生（用你的理解说清楚，别只是复制官方说法）

**出坑**
已经掉坑里了怎么爬出来（具体操作，能复制的代码 > 文字描述）

**预防**
下次怎么避免（提示词 / 配置 / Hook / Skill）

...（8 条左右，别凑数）

## 贡献新陷阱

模板见 [claude-code.md 结尾](./claude-code.md#补充遇到新陷阱怎么办)。

## 相关方法论

- 指向工具自己的 README
- 指向相关的 common/ 方法论
```

---

## 质量标准

写 pitfalls 最容易掉进的坑（pitfall 的 pitfall）：

- ❌ **教程化凑数**：把"怎么用"硬写成"陷阱"（例："忘记保存文件" 不是陷阱）

- ❌ **复制官方 FAQ**：价值在你真实遇到、真实修好的过程
- ❌ **症状和根因混在一起**：读者看不清"这是不是我遇到的"
- ✅ **可复现的症状**：读者能对号入座
- ✅ **出坑操作具体**：给命令或提示词，不是"注意一下"
- ✅ **预防措施落地**：能直接抄到 `CLAUDE.md` / `.cursorrules` / settings.json

---

## 相关

- common/debugging.md — 系统化调试方法论
- workflows/scenarios.md — 实战对话脚本，含大量"打断话术"

# Claude Code 陷阱合集

> README 里的"常见陷阱"表是速查，这里展开讲：每个陷阱讲清楚**症状 → 根因 → 出坑 → 预防**，用过都会遇到。
>
> 贡献踩过的坑，请提 Issue 或 PR。

---

## 陷阱 1：上下文溢出，AI 开始胡说八道

**症状**

- 对话到后半段，AI 开始编造之前没讨论过的细节
- 明明前面说过的约束，它忘了
- 回答越来越短、越来越笼统，问题没回答完就 ↩

**根因** Claude 有上下文窗口上限。对话+工具调用累计到 80% 以上时，最早的消息会被截断，AI "忘记"了早期设定。但它不会告诉你——它会**装作记得**，靠编造填空。

**出坑**

```
/compact              # 主动压缩上下文，保留关键决策
```

或者直接**开新对话**，把重要信息写进 `CLAUDE.md` 传递。

**预防**

- 上下文用到 50% 就 `/compact`，不要等 80%
- 一个对话做一件事，做完关了开新的
- 重要决策不要只靠对话记忆——写进文件（`CLAUDE.md` 或 `docs/`）

---

### 陷阱 2：幻觉 API — AI 编了个不存在的函数

**症状**

- AI 写了 `someLib.fancyMethod()`，但这个库根本没这个方法
- 引用了不存在的配置项、不存在的 CLI flag
- 代码看起来很合理，跑起来报 `TypeError: x is not a function`

**根因** LLM 的训练数据截止日期前某个版本可能有这个 API，或者根本没有。它会根据"看起来应该有"生成代码。不 grep、不看文档就动手，就会编。

**出坑**

- 先让它证明：`先 grep 一下这个方法，确认它存在再用`
- 跑一次看报错，把报错贴回去：`看这个报错，上网查这个库的最新文档`

**预防** 在 `CLAUDE.md` 加一条：

```
## 编码纪律
- 使用任何第三方库 API 前，先 Read 它在 node_modules/ 下的类型定义或源码确认
- 不确定的 API 不要猜，要么查文档要么问我
```

---

### 陷阱 3：过度重构 — 修一行，改一百行

**症状**

- 你说"给这个函数加个参数"，它把整个文件重写了
- 顺手把变量重命名、加了注释、改了格式化、换了错误处理风格
- PR diff 看起来像"整个文件重写"，实际想要的改动埋在中间

**根因** Claude 默认倾向于"做好"，看到不满意的地方就顺手改。没明确约束时，它把"改进代码质量"当成加分项。

**出坑** 已经改完了但 diff 太乱：

```
把所有非必要改动回滚。只保留 [具体改动] 这一处。
格式化、重命名、加注释 全部回退。
```

**预防** 提示词里**明确划边界**：

```
只改 userService.ts 的 createUser 函数。
不要重命名、不要加注释、不要改格式、不要动其他文件。
如果看到别的问题，告诉我但不要动手。
```

对关键文件加 `<important>` 标签：

```
<important>
不要自动重构 src/legacy/ 下的代码，那是旧兼容层。
</important>
```

---

## 陷阱 4：测试谎报 — 它说"通过了"但根本没跑

**症状**

- AI 说"测试已通过，所有功能正常" ✅
- 你自己跑 `pnpm test`，一堆红
- 或者：它只跑了部分测试，没跑的那部分就是挂的

**根因** Claude 有时会**基于代码推断**测试应该通过，而不是真跑了测试。或者跑了部分测试但没看全输出。

**出坑**

- 用否定式打断："不要推断，跑 `pnpm test` 把**完整输出**贴出来"
- 要求可验证产物："跑完后把覆盖率报告粘给我看"

**预防** 用 Hook 强制验证：

```
{
  "hooks": {
    "PostToolUse": [{
      "matcher": "Write|Edit",
      "hooks": [{ "type": "command", "command": "pnpm test --run --reporter=basic"
}]
    }]
  }
}
```

或者装 superpowers-zh 的 `verification` skill，它会在声称"完成"前强制跑测试。

---

## 陷阱 5：CLAUDE.md 被忽略 — 规则写了，它不听

### 症状

- 明明 `CLAUDE.md` 写了"用 pnpm 不要用 npm"，它照样 `npm install`
- 写了"测试放 `__tests__/` 下"，它照样写在同目录

**根因** `CLAUDE.md` 超过 ~200 行时，靠后的规则会被"边缘化"。Claude 看到长文档会自动摘要，细节丢了。另一个常见原因：规则太像"建议"而不是"要求"。

### 出坑

```
# 立刻在当前会话里
从现在开始严格按 CLAUDE.md 第 X 节执行。
重要规则是 [具体复述一遍]。
```

### 预防

- `CLAUDE.md` 控制在 200 行以内
- 大项目拆到 `.claude/rules/` 用 `globs` 按文件类型加载
- 关键规则加 `<important>` 标签
- 用 "必须 / 禁止" 代替 "建议 / 尽量":

```
## 禁止
- ❌ 禁止用 npm，本项目用 pnpm
- ❌ 禁止把测试写在同目录，必须放 __tests__/
```

---

## 陷阱 6：`git --amend` 覆盖你未 commit 的工作

**症状**

- 你让 Claude "帮我改一下上一个 commit"
- 它跑 `git commit --amend`
- 你之前 stage 但没 commit 的修改……被吃进上一个 commit 了，或者丢了

**根因** `--amend` 会把当前 staged 的所有内容合进上一个 commit。Claude 不知道你 staging area 里还有别的东西。

**出坑**

```
git reflog                    # 找回被 amend 覆盖的前一个 commit
git reset --hard HEAD@{N}     # 回滚到那个状态（N 是 reflog 里的序号）
```

**预防**

- 不让 Claude 用 `--amend`——明确告诉它"新建 commit，不要 amend"
- 在 `.claude/settings.json` 里限制：

```
{
  "permissions": {
    "deny": ["Bash(git commit --amend*)"]
  }
}
```

---

## 陷阱 7：Plan Mode 写得好，执行偏了

**症状**

- 用 `/plan` 让它先规划，方案写得头头是道
- 点确认开始执行，前两步还对
- 到第三步开始"自由发挥"，偏离了原方案

**根因** Plan mode 给的是**起点**，不是"锁死的脚本"。执行过程中 Claude 会基于实际情况调整，有时调整得太远就脱轨了。尤其是当某步失败时，它倾向于"换个思路"而不是停下来问你。

**出坑**

```
停。对照一下原 plan 的第 N 步，你现在在做的和 plan 一致吗?
如果偏了，说明为什么，让我决定继续偏还是回到 plan。
```

**预防**

- 让 plan 更"原子化"：每步独立可验证，而不是"第三步：重构登录流程"这种大步
- 关键节点明确要求"做完暂停等确认"：

```
按 plan 执行。每步做完:
1. 跑相关测试
2. 总结改动一句话
3. 暂停等我 OK 再下一步
```

---

## 陷阱 8：Subagent 之间上下文对不上

**症状**

- 主 agent 让 subagent A 写代码，subagent B 跑测试
- B 回报"测试失败，找不到文件"
- 但 A 明明写了——原来 A 的"写"只是返回了文本，没真写到磁盘

**根因** Subagent 之间**不共享文件系统视图的实时状态**，尤其是主 agent 没把 A 的输出持久化就传给 B。A 在它的上下文里"知道"文件存在，但磁盘上可能根本没有。

**出坑**

```
停，让我核实状态。
运行：ls -la src/newfile.ts && cat src/newfile.ts
确认文件真的存在再往下做。
```

**预防**

- **让 agent 通过文件传结果，不要只靠消息传递**
- 主 agent 调度时明确："A 把代码**写到磁盘**后，告诉我文件路径；B 读那个路径再跑测试"
- 复杂编排超过 2 个 subagent，考虑换成主 agent 线性执行——可靠性比并行更重要

---

## 补充：遇到新陷阱怎么办

踩坑是最值钱的经验。欢迎提 Issue 或 PR 补充，模板：

```
## 陷阱 N：一句话标题

**症状**
你看到的异常现象（越具体越好）

**根因**
为什么会发生（用你的理解说清楚）

**出坑**
已经掉坑里了怎么爬出来

**预防**
下次怎么避免（提示词 / 配置 / Hook / Skill 等）
```

---

## 相关方法论

- common/debugging.md — 系统化调试方法论
- common/context-management.md — 上下文管理
- workflows/scenarios.md — 实战对话脚本里也有打断话术

# Cursor 陷阱合集

> Cursor 用过的都知道，它强但也有独特的坑。8 个真实踩坑场景，症状 / 根因 / 出坑 / 预防四段式展开。
>
> 踩过新坑？提 Issue 或 PR。

---

## 陷阱 1：Composer 脱缰 — 动了你没让它动的文件

**症状**

- 用 Composer 改一个功能，结果 diff 涉及 6 个文件
- 有些文件你都没打开过，它跨目录找到然后顺手改了
- 最糟：它"重构"了 `src/utils/` 某个公共工具，破坏了别的功能

**根因** Composer 是 Agent 模式，默认有权限跨文件修改。它会自己搜索相关文件、自己决定"这个也得改"。没有显式边界时，"相关"的定义由它说了算。

**出坑**

```
停。把所有非 [我指定的文件] 的改动回滚。
只保留 @src/pages/order/list.tsx 这个文件的改动。
```

**预防** Composer 提示词里**显式圈定范围**：

```
用 Composer。只改这两个文件：
@src/pages/order/list.tsx @src/pages/order/detail.tsx

如果需要改其他文件才能完成，先告诉我，让我决定。
```

或者在 `.cursor/rules/` 里加：

```
## Composer 边界
- 除非我 @ 引用了某文件，不要修改它
- src/utils/ src/types/ 目录属于公共代码，改动前必须告知
```

---

### 陷阱 2：Tab 补全一次吐一整页，Esc 来不及按

**症状**

- 你只想写一个变量名，Tab 一按补出来 30 行代码
- 补全的代码看起来 OK 但实际有隐蔽的 bug（比如硬编码数据、错误的类型假设）
- 养成"反手 Undo"习惯后，开发反而变慢

**根因** Cursor 的 Tab 补全会预测"你接下来想写的一整段"，不是单行补全。遇到函数开头、文件开头这种"信息密集点"，它会直接补出完整函数体。

**出坑**

- 长补全按 `Esc` 直接拒绝
- 已经接受了不对：`Cmd+Z` 撤销，或用 `Cmd+K` 选中这段问"这段有什么问题"

**预防**

- Cursor Settings → Features → 调低补全长度（或关闭 `Auto` 模式改成只补全当前行）
- 写**函数签名和类型**再等补全——签名越完整，补全越准（反之越瞎猜）
- 重要逻辑块先写一行注释再等补全：

```
// 计算两个日期之间的工作日，排除周末和中国法定节假日
function countBusinessDays(start: Date, end: Date): number {
  // Tab 补全会基于注释生成正确实现
}
```

---

### 陷阱 3：`.cursorrules` 太长，规则全没生效

**症状**

- 花半天写了 300 行 `.cursorrules`，覆盖各种场景

- 实际用起来：AI 该违反的还是违反，感觉根本没读
- 新加的规则像"扔进黑洞"

**根因** `.cursorrules` 单文件超过 ~200 行时，靠后的规则被边缘化——Cursor 会对长文件做"摘要注入"，细节条款被丢弃。

**出坑** 拆到 `.cursor/rules/`（注意是目录，不是单文件）：

```
.cursor/rules/
├── global.md            # 无条件加载的核心规则（< 100 行）
├── react.md             # 仅 .tsx 文件触发
├── api.md               # 仅 src/api/ 下触发
└── testing.md           # 仅 *.test.ts 触发
```

每个文件加 frontmatter 控制加载：

```
---
globs: ["src/components/**/*.tsx"]
---

# React 组件规则
- Props 必须用 interface 定义
- 必须处理 loading 和 error 状态
```

**预防**

- 新项目**一开始就用** `.cursor/rules/` **目录**，别用单文件 `.cursorrules`
- 每个规则文件控制在 100 行内
- 规则太具体、太场景化的，放 Notepad，用到再 @ 引用

---

## 陷阱 4：`@file` 引用失效 — 上下文没真传进去

**症状**

- 你说 `@src/types/user.ts 按这个类型改接口`
- AI 生成的代码里的字段名、类型全是它猜的，不是你的真实类型
- 仔细看会发现：它根本没读那个文件

**根因** 常见原因三种：

1. 路径错了（大小写 / 相对绝对）
2. 文件超大被截断（只读了前 N 行）
3. 你文件刚改过还没存盘（Cursor 读的是磁盘版本）

**出坑**

```
先确认：@src/types/user.ts 这个文件里 User 接口的字段名都是什么？
列出来我对一下，再开始改代码。
```

让 AI 先复述文件内容，确认它**真的读到了**，再让它动手。

**预防**

- 大文件用 `@folder` 引用整个目录让它自己找，比 `@file` 精确到路径稳
- 改完文件**先 Cmd+S 存盘**再 @ 引用
- 超大文件（>1000 行）精确引用行号范围：`@src/models/big.ts:50-120`

---

### 陷阱 5：Chat vs Composer 混用，效率全失

**症状**

- 在 Chat 里问"帮我重构这个模块"——它给你一大堆建议文字，不改代码
- 在 Composer 里问"这段代码怎么理解"——它不回答，直接开始改
- 你来回切换，每次都选错模式

**根因**

- Chat（`Cmd+L`）：问答模式，不主动改代码
- Composer（`Cmd+I`）：Agent 模式，优先改代码
- Inline（`Cmd+K`）：行内编辑，只改选中区域

三个模式的触发入口不一样，新手分不清。

**出坑** 切错模式了：把任务内容复制，关闭当前面板，打开正确模式粘进去。

**预防** 记住**快捷键和用途的一一对应**：

| 你想做什么 | 用哪个 | 快捷键 |
| --- | --- | --- |
| 问问题、理解代码 | Chat | `Cmd+L` |
| 改选中的一段代码 | Inline | `Cmd+K` |
| 跨文件大改动 | Composer | `Cmd+I` |

---

### 陷阱 6：模型切换导致风格不一致

**症状**

- 你在 Cursor 里一半时间用 Claude 3.5，一半用 cursor-small
- 同一个项目里代码风格分裂：有的地方严谨地写类型，有的地方随意 `any`
- 你自己都没意识到是模型切换导致的

**根因** Cursor 支持多模型，不同模型的"默认风格"不同：

- Claude Sonnet：倾向严谨，会主动加类型、加错误处理
- GPT-4：倾向简洁，少废话少注释
- cursor-small：速度快，但在复杂任务上会偷懒

自动切换（系统根据任务复杂度选模型）会让这个问题更隐蔽。

**出坑** 把风格规则写进 `.cursor/rules/global.md`，不管用哪个模型都会遵守：

```
# 代码风格（跨模型一致）
- 必须用 TypeScript，不用 any
- 函数必须有返回值类型标注
- 错误用 Result<T, Error> 包装，不抛异常
```

**预防**

- **一个项目定一个主模型**，在设置里锁定
- 明确任务才换模型（"这个任务用 Opus 深度思考"）
- 不要依赖 `auto` 选模型

---

### 陷阱 7：Notepads 上下文污染

**症状**

- 半年前建的 Notepad 里有过时信息（旧 API 设计、废弃的代码风格）
- 新对话里 AI 引用 Notepad，给出了过时方案
- 你都不记得自己建过这个 Notepad

**根因** Notepads 是**手动管理**的上下文片段，Cursor 不会自动更新它。项目演进几个月后，Notepads 和真实代码就开始脱节。

**出坑**

- 定期清理：看到 AI 用了奇怪方案，怀疑是 Notepad 污染，去 Notepad 面板检查
- 发现过时内容立刻删除或更新

**预防**

- Notepad **只放稳定的约定**（API 返回格式、认证方式等不常变的）
- 易变的内容（当前正在做的功能、临时方案）**用 `@folder` 引用实时代码**而不是 Notepad
- Notepad 里加修改日期，定期检查："2026-01 以前的都复审一次"

---

### 陷阱 8：索引过期 — @ 引用到已删除的文件

**症状**

- 你重命名或删除了某个文件
- 在 Chat 里 @ 还能引用到它（带着旧内容）
- AI 基于旧文件给建议，你按建议改，编译报错"文件不存在"

**根因** Cursor 的代码库索引**不是实时更新**。删除/重命名后，内存里的索引还保留旧条目一段时间。

**出坑** 命令面板（`Cmd+Shift+P`）→ `Cursor: Resync Index`

**预防**

- 大规模重命名/删除后主动重建索引
- 如果用 Git 分支切换频繁，切分支后跑一次 resync
- 发现 @ 引用结果奇怪，第一反应先检查索引，不要怀疑 AI

## 贡献新陷阱

模板见 claude-code.md 结尾。

---

## 相关方法论

- Cursor 完整指南
- common/context-management.md — 上下文管理通用策略
- workflows/scenarios.md — 含 Cursor + Claude Code 协作场景

# GitHub Copilot 陷阱合集

> Copilot 是最老牌的 AI 编程工具，也最容易被低估——以为就是补全，结果 Agent 模式和 MCP 支持都挺深。8 个常见坑。
>
> 踩过新坑？提 Issue 或 PR。

---

## 陷阱 1：补全用了过期的 API

**症状**

- Copilot 生成的代码用了某个库的旧 API
- 你拷进来跑：`Deprecated: use xxx instead` 或直接 `is not a function`
- 升级到库的新版本后，Copilot 还在按旧 API 补全

**根因** Copilot 的补全基于训练数据 + 当前打开文件的上下文。训练数据截止日期前的库版本可能已经迭代过，而 Copilot 不会读 `package.json` 里的实际版本。

**出坑**

- 把新版文档的关键一页**作为 tab 打开**（Copilot 会读打开的 tab）
- 或在文件顶部加注释：

```
// 使用 TanStack Query v5 API（useQuery 签名：useQuery({ queryKey, queryFn })）
import { useQuery } from '@tanstack/react-query';
```

**预防**

- 在 `.github/copilot-instructions.md` 里声明关键库的版本和 API 风格：

```
## 关键依赖版本
- React 19（使用 useTransition、useOptimistic）
- TanStack Query v5（useQuery 新签名）
- Zod v3（不用 v2 的 object() 链式风格）
```

---

### 陷阱 2：Agent 模式改了 `.env` 或敏感文件

**症状**

- 让 Copilot Agent 改个配置，它顺手扫了 `.env` 把 key 读进上下文
- 在 Chat 窗口里看到你的 API key 明文显示
- 最坏：补全建议里带出了真实的 secret

**根因** Copilot 的 exclude 配置不是默认启用的。`.gitignore` 里的文件 Copilot 依然会读。`.env`、`.aws/credentials`、`id_rsa` 这类敏感文件如果不显式排除，都在它的感知范围。

**出坑** 立刻在 `.vscode/settings.json` 加排除：

```
{
  "github.copilot.enable": {
    "*": true,
    "env": false,
    "secrets": false
  },
  "github.copilot.advanced": {
    "exclude": ["**/.env*", "**/secrets/**", "**/*.pem", "**/*.key"]
  }
}
```

如果敏感信息已经进了 Chat 历史：清空 Chat（`Ctrl+L` 或面板里清除），**并轮换相关密钥**。

**预防**

- 项目初始化就配好 exclude，不要等出事
- 用 GitHub 的 Copilot Content Exclusion（企业版）做组织级兜底
- 敏感配置用 `direnv` / `1Password CLI` 等从非项目目录注入，不落地

---

### 陷阱 3: `.github/copilot-instructions.md` 太长被忽略

**症状**

- 你在 instructions 里写了 20 条规则
- Agent 只遵守前几条，后面的像没看过
- 尤其 Chat 窗口里提问时，细节规则几乎不生效

**根因** copilot-instructions.md 超过 ~150 行时效果显著下降。Copilot 在注入上下文时会做截断，靠后的内容被牺牲。

**出坑** 拆文件：

- 核心不变规则 → `.github/copilot-instructions.md`（< 100 行）
- 场景化规则 → `.github/chatModes/` 下各自的 `.chatmode.md`
- 专项角色 → `.github/agents/` 下各自的 `.agent.md`

**预防** Instructions 里**只放跨场景的全局规则**（技术栈、命名、禁止事项）。具体场景规则：

```
.github/
├── copilot-instructions.md      # 核心规则
├── chatModes/
│   ├── security-review.md       # 安全审查专用
│   └── write-tests.md           # 写测试专用
└── agents/
    └── migration-helper.md      # 迁移项目专用
```

---

### 陷阱 4: `#file` 引用的 vs `@workspace` 找到的不一致

**症状**

- 你说 `#file:src/api/user.ts 按这个改`
- Copilot 改得基本对，但不完全
- 另一次你用 `@workspace` 让它找 user.ts，它提到了 2 个 user.ts（项目有两处重名）

**根因**

- `#file` 精确引用你给的路径
- `@workspace` 会让 Copilot 自主搜索整个工作区

- 项目里有重名文件时，`@workspace` 可能选错那个

**出坑** 明确路径 > 模糊索引：

```
❌ @workspace 改一下 user.ts 的 register 方法
✅ #file:src/api/v2/user.ts 改一下这里的 register 方法
```

**预防**

- 项目里避免重名文件（`user.ts` × 3 这种结构重构一下）
- 如果必须重名，引用时用完整路径而非文件名
- `@workspace` 只用于**探索**（"项目里有没有 XX"），不用于**指向**（"改 XX"）

---

### 陷阱 5：MCP Server 配置了但没生效

**症状**

- 在 `.vscode/settings.json` 配了 `github.copilot.chat.mcpServers`
- Chat 里该用 MCP 工具时它没用，走了别的路径
- 不报错，就是悄悄没用

**根因** 常见三种：

1. MCP server 启动命令写错（`npx` 路径、参数顺序）
2. MCP server 启动了但 Copilot 版本不支持（需要较新 VS Code + Copilot Chat）
3. server 启动成功但工具 schema 定义有问题，Copilot 认不出

**出坑**

```
# 在 Chat 里直接问
你当前可以调用哪些 MCP 工具？列出来。
```

如果它列不出你配的工具 → MCP 没加载。看 VS Code 的 Output 面板 → Copilot Chat，找错误日志。

**预防**

- 先用官方示例 server（比如 `@modelcontextprotocol/server-filesystem`）验证 MCP 能接通
- 再换自己的 server
- VS Code 和 GitHub Copilot 插件都保持最新版

---

### 陷阱 6：补全在 JetBrains 和 VS Code 行为不一致

**症状**

- 团队里 VS Code 用户的补全质量明显比 JetBrains 用户好
- 同一个 prompt 给的建议不一样
- 共享 `.github/copilot-instructions.md` 后，VS Code 生效，JetBrains 部分生效

**根因**

- VS Code 是 Copilot 的主战场，新功能优先 ship
- JetBrains 插件的 instructions 支持、MCP 支持、Agent 模式都滞后
- 底层模型也可能不一样（JetBrains 插件有时用老模型）

**出坑** 要么全团队统一 IDE，要么接受 JetBrains 体验差一截。

**预防**

- 团队 AI 工具选型时，把 IDE 一致性当硬约束
- JetBrains 用户补 instructions 时，在文件开头加上**复述式核心规则**（即使 instructions 不生效，开头几行总会被注入）

---

### 陷阱 7：免费用户遇到隐形限流

**症状**

- 免费额度（Copilot Free）用得好好的，某天开始补全延迟暴涨
- Chat 请求经常排队或直接拒绝（"已达使用上限"之类的提示）
- 补全变慢、建议数量减少

**根因** Copilot Free 有每月补全数和 Chat 次数上限。接近上限时会排队/限速，超出后直接拒绝请求。VS Code 里提示不一定醒目，容易忽略。企业版用户也可能遇到组织级配额。

**出坑**

- 在 GitHub 账户 → Copilot 设置页查看用量
- 用量接近上限：升级 Pro / Pro+ / Business，或当月剩余时间依赖其他工具

**预防**

- 高频使用者直接上 Pro（$10/月），别在 Free 上省
- 团队用户和管理员对齐配额和使用策略
- 建立"Copilot 挂了用谁"的 fallback（比如切 Claude Code 或 Cursor）

---

## 陷阱 8：自定义 Chat Mode / Agent 不被发现

**症状**

- 你在 `.github/chatModes/security-review.md` 写了个专家角色
- 在 Chat 里输入 `@security-review`，Copilot 不识别
- 或者识别了但行为和普通 Chat 没区别

**根因**

- 文件名和 frontmatter 里的 `name` 字段不匹配
- 没有 frontmatter 或字段缺失
- VS Code / Copilot 版本太旧不支持自定义 Chat Mode

**出坑** 检查 frontmatter 必备字段：

```
---
name: security-review       # 和文件名一致
description: Security review using OWASP Top 10
---

# 后面是角色内容
```

**预防**

- 拷一个官方示例 Chat Mode 文件，基于它改，别从零写
- 改完重启 VS Code（不是重启 Copilot）

- `.github/chatModes/*.md` 路径固定，别放错目录

---

## 贡献新陷阱

模板见 claude-code.md 结尾。

---

## 相关方法论

- Copilot 完整指南
- common/security.md — AI 编程的安全风险和防护
- common/context-management.md — 上下文管理

# Aider 陷阱合集

> Aider 的坑和 IDE 类工具很不一样——Git 自动提交、`/add /drop` 手动管理、多模型切换，每一个机制都能咬你一口。7 个真实坑。
>
> 踩过新坑？提 Issue 或 PR。

---

## 陷阱 1：自动 commit 吃进了你的工作树变更

**症状**

- 你在项目里改了几个文件还没 commit
- 让 Aider 改一个别的文件
- 它生成代码后自动 commit，把你**未 stage 的变更也一起打包提交**

**根因** Aider 默认开启 `auto-commits: true`。它 commit 的范围是整个工作树 diff，不区分"我改的"和"用户改的"。Aider 觉得"改完就提交"是好事，不知道你有别的工作正在进行。

**出坑**

```
git reset HEAD~1          # 取消最后一次 commit，文件回到工作树
git status                # 核对哪些是你的、哪些是 Aider 的
# 手动 commit 该 commit 的，丢弃该丢的
```

**预防** 三选一：

1. **开会话前先 `git stash`**：把你未完成的工作藏起来，Aider 的 commit 才干净
2. **关闭自动 commit**：`aider --no-auto-commits`，或在 `.aider.conf.yml` 里 `auto-commits: false`
3. **开独立 feature 分支**：`git checkout -b feature/xxx` 再开 Aider，弄脏也只是脏分支

---

### 陷阱 2：`/add` 漏文件，AI 靠脑补填空

**症状**

- 让 Aider 改 `ServiceA`，它顺利改完
- 但 ServiceA 依赖的 `UtilB` 里的函数签名其实已经变了，Aider 不知道，生成的代码调用了根本不存在的签名
- 跑起来 `TypeError`

**根因** Aider 只把 `/add` 过的文件真正**读进上下文**。Map 模式会索引项目结构（名字+位置），但**不读内容**。依赖文件没 `/add`，AI 只能基于文件名猜。

**出坑**

```
/add src/utils/UtilB.ts
# 把依赖也加进来，让它重新看一遍
重新看一下 ServiceA 依赖的 UtilB，确认你用的签名是对的
```

**预防** 改之前先让 Aider 自己列出依赖：

```
/ask
我要改 ServiceA。列出它直接依赖的所有文件，
以及它的公开接口被哪些文件调用。
给我 /add 的命令，我一次性加齐。
```

然后把它列的都 `/add` 进来再开始 `/code`。

---

### 陷阱 3：切了模型，行为突然变了

**症状**

- 原来用 Claude，习惯了它的严谨风格
- 为了省钱切到 DeepSeek，**同一个提示词产出质量断崖下跌**
- 复杂重构开始出奇怪的 bug，Architect 模式给的方案也变得粗糙

**根因** 不同 LLM 的"指令理解能力"差异很大。同样一句 `/architect 设计 WebSocket 通知系统`：

- Claude Sonnet：会追问需求、考虑异常路径

- DeepSeek：直接给一个方案，细节覆盖不全
- 本地 7B 模型：给一个结构对但细节漏洞很多的方案

**出坑**

- 复杂任务发现模型跟不上：切回 Claude 做这一步，做完再切回便宜模型
- 配 `--weak-model` 分工：主推理用强模型，commit message/简单任务用弱模型

**预防** 配置分层（`.aider.conf.yml`）：

```
model: claude-sonnet-4-5            # 主模型
weak-model: deepseek/deepseek-chat # 弱任务（commit message、简单补全）
editor-model: claude-haiku-4-5      # 编辑器补全
```

或**按任务类型**切：

| 任务 | 建议模型 |
| --- | --- |
| `/architect` 方案设计 | Claude Sonnet/Opus |
| `/code` 实现已有方案 | DeepSeek 够用 |
| `/ask` 问答/解释 | 任意 |
| 大型重构 | Claude Opus |

---

### 陷阱 4：Lint/Test 自动循环，账单爆炸

**症状**

- 配了 `auto-lint: true` 和 `auto-test: true`
- Aider 改完跑 lint 失败，它自己修，又失败，又修……循环几轮
- 一次对话下来 token 消耗是预期的 5 倍

**根因** Aider 的 auto-lint/test 是 "失败就让 LLM 再改一次"。如果问题根本不是代码能修好的（比如环境配置、依赖版本），它会永远修不好，一直重试直到达到默认上限。每次重试都烧 token。

**出坑**

```
/undo                          # 回滚本轮修改
/lint-cmd none                 # 临时关掉 auto-lint
# 手动修根本问题（环境、配置），再打开 auto-lint
```

**预防**

- 限制重试次数（`aider --max-reflections 3`）
- lint 命令用 `--fix` 模式让 linter 先尝试自动修复，而不是报错扔给 AI：

```
lint-cmd: "eslint --fix"   # 不是 "eslint"
```

- 测试命令跑 fail-fast：

```
test-cmd: "pytest -x --maxfail=1"
```

---

### 陷阱 5：Map 模式过期，引用了已删除的文件

**症状**

- 项目结构变了（删了目录、重命名了模块）
- Aider 还"记得"旧结构，生成 import 路径引用不存在的模块
- 或者 `/ls` 列出的文件和磁盘对不上

**根因** Map 模式的索引在**会话开始时生成**，会话中如果 git 层面大变（切分支、rebase、大规模删除），Map 不自动刷新。

**出坑** 退出 Aider 重进，Map 会重建。 或者在会话里：

```
/reset              # 清空会话上下文
/map-refresh        # 重新扫描项目结构（取决于版本是否支持）
```

**预防**

- 切分支后重启 Aider（别在同一会话里跨分支操作）
- 大规模文件变更后也重启
- 长会话定期 `/reset` 刷新状态

---

### 陷阱 6：Architect 方案漂亮，Code 执行跑偏

**症状**

- `/architect` 给出一个清晰的方案，你很满意
- 切回 `/code` 让它实现，头一两个文件还对，越往后越偏
- 最后成品和 Architect 方案几乎不一样

**根因** Aider 的 Architect 和 Code 模式**共享会话上下文**，但 Architect 阶段的设计很长，随着 Code 阶段不断改代码，早期的 Architect 方案被上下文压缩逐步丢失。

**出坑**

```
# 每实现一步后，复核一次方案
/ask 对照 Architect 阶段的方案，当前实现偏离了哪些点?
```

**预防**

- Architect 方案生成后**立刻写进文件**：`把这个方案写到 docs/arch.md，我们照这个文件实现`
- 实现阶段明确引用文件：`/add docs/arch.md 按这里的方案实现下一步`
- 复杂任务**每个模块开独立会话**：一个会话做一个子模块，避免长会话漂移

---

### 陷阱 7：`/commit` 后 `--amend` 失控

**症状**

- 你让 Aider 改一下刚才的某个文件（已经自动 commit 了）
- 它做完又 commit 了一次
- 你想"合并到刚才的 commit"，让它 `--amend`
- 结果 amend 把你**之前 stash 的内容**吃进来了，或者把 Aider 的两次改动合进去，历史反而更乱

**根因** Aider 的 commit 时机是"改完就提交"，不是"原子化每个逻辑改动"。你手动 amend 会混入所有当前 staged 内容。同时 Aider 本身不支持精细的 commit 粒度控制。

**出坑**

```
git reflog                      # 找回 amend 前的状态
git reset --hard HEAD@{N}       # 回滚
# 用 interactive rebase 重新整理
git rebase -i HEAD~5
```

**预防**

- Aider 会话结束后**统一 squash**：

  ```
  git reset --soft HEAD~N   # N = Aider 的 commit 数
  git commit -m "feat: add notifications"
  ```

- 别在 Aider 会话进行中用 `--amend`
- 关键 checkpoint 自己打 tag：`git tag wip-before-aider`，出事能回

---

## 贡献新陷阱

模板见 claude-code.md 结尾。

---

## 相关方法论

- Aider 完整指南
- common/task-decomposition.md — 任务拆解（Aider 尤其依赖）
- common/debugging.md — 系统化调试

# Aider 最佳实践

> Aider 是一个开源的 CLI AI 编程工具，核心特色是 **Git 原生** — 每次修改自动提交，天然支持版本控制。支持几乎所有主流 LLM（Claude、GPT、DeepSeek、Qwen、本地模型），是最灵活的 AI 编程 CLI。

---

## 核心概念

| 概念 | 说明 | 用途 |
| --- | --- | --- |
| **Chat 模式** | `code` / `ask` / `architect` | 不同任务用不同模式 |
| **自动 Git 提交** | 每次修改自动 commit | 随时可以回滚 |
| **Map 模式** | 自动索引项目结构 | 理解大代码库 |
| **多模型支持** | Claude/GPT/DeepSeek/Ollama 等 | 灵活选择，控制成本 |
| **Lint & Test** | 内置代码检查和测试 | 修改后自动验证 |

---

## 快速上手

### 安装

```
pip install aider-chat

# 设置 API Key（选一个）
export ANTHROPIC_API_KEY=sk-xxx     # Claude
export OPENAI_API_KEY=sk-xxx        # GPT
export DEEPSEEK_API_KEY=sk-xxx      # DeepSeek
```

#### 基本使用

```
cd /your/project
aider

# 指定模型
aider --model claude-sonnet-4-5

# 用 DeepSeek（便宜）
aider --model deepseek/deepseek-chat

# 用本地模型（免费）
aider --model ollama/qwen2.5-coder
```

#### 三种 Chat 模式

```
# code 模式（默认）— 直接修改代码
/code 给 utils.py 加一个 retry 装饰器

# ask 模式 — 只问不改
/ask 这个函数的时间复杂度是多少?

# architect 模式 — 先设计再实现
/architect 设计一个任务队列系统，先给方案不要写代码
```

---

## 提示词技巧

#### 1. 添加文件到上下文

```
# 手动添加文件
/add src/models/user.py src/api/auth.py

# 添加整个目录
/add src/services/

# 移除不需要的文件
/drop src/legacy/old_auth.py
```

Aider 只会修改已添加的文件。这是精确控制范围的方式。

### 2. Architect 模式做大任务

```
/architect

我要给项目加一个 WebSocket 实时通知系统。
要求：
1. 用户上线/下线通知
2. 新消息实时推送
3. 支持频道订阅/退订
4. 需要考虑断线重连

先给出技术方案，包括：
- 用什么库
- 数据流设计
- 需要新建哪些文件
- 需要修改哪些现有文件

确认后切到 code 模式实现。
```

### 3. 利用自动 Git 提交

```
# 每次修改都有独立 commit，随时回滚
git log --oneline  # 看 Aider 的提交记录
git diff HEAD~1    # 看最后一次修改
git revert HEAD    # 不满意？一键回滚

# 设置自定义提交前缀
aider --commit-prefix "[ai] "
```

#### 4. 多模型策略

```
# 复杂架构设计 — 用最强模型
aider --model claude-sonnet-4-5
/architect 设计微服务拆分方案

# 日常编码 — 用性价比模型
aider --model deepseek/deepseek-chat
/code 按方案实现 user-service

# 代码审查 — 用免费本地模型
aider --model ollama/qwen2.5-coder
/ask 看看这段代码有没有问题
```

---

## 进阶技巧

#### .aider.conf.yml — 项目配置

```yaml
# .aider.conf.yml
model: claude-sonnet-4-5
auto-commits: true
auto-lint: true
auto-test: true
test-cmd: pytest
lint-cmd: ruff check
```

#### Lint + Test 自动化

```
# 每次修改后自动:
# 1. 跑 linter 检查
# 2. 跑相关测试
# 3. 如果失败，自动修复再重试
auto-lint: true
auto-test: true
lint-cmd: "ruff check --fix"
test-cmd: "pytest -x"
```

### 与 Git 工作流集成

```
# 在 feature 分支上工作
git checkout -b feature/add-notifications
aider

# Aider 的每次修改都在这个分支上
# 完成后正常走 PR 流程
git push -u origin feature/add-notifications
```

---

## 与其他 CLI 工具的区别

| 维度 | Aider | Claude Code | Gemini CLI |
|---|---|---|---|
| Git 集成 | ★★★（自动提交） | ★★☆（手动） | ★☆☆ |
| 模型灵活性 | ★★★（几乎所有 LLM） | ★☆☆（仅 Claude） | ★☆☆（仅 Gemini） |
| Agent 能力 | ★★☆ | ★★★ | ★★☆ |
| 上下文管理 | 手动 /add /drop | 自动 | 自动 |
| 开源 | ✅ 完全开源 | ❌ | ✅ 开源 |
| 成本控制 | ★★★（可用免费模型） | ★☆☆ | ★★★ |
| 适合 | 灵活、省钱、Git 重度用户 | 复杂 Agent 任务 | 大代码库分析 |

---

## 常见陷阱

| 陷阱 | 说明 | 解决 |
|---|---|---|
| auto-commit 吃变更 | 自动提交把你未 stage 的工作一起吞了 | 会话前 `git stash`，或开独立分支 |
| /add 漏依赖 | 上下文不全，AI 靠文件名脑补 | 让它先 /ask 列出所有依赖再 /add |

| 陷阱 | 说明 | 解决 |
| --- | --- | --- |
| 模型切换质量塌 | 切到便宜模型后产出质量断崖下跌 | 分任务类型切，复杂任务回 Claude |
| lint 循环烧 token | auto-lint 失败反复让 LLM 修 | `--max-reflections 3`，lint 用 `--fix` 模式 |

👉 **深度展开版**：Aider 陷阱合集 — 7 个真实踩坑场景，每个带症状 / 根因 / 出坑 / 预防

---

## 配置模板

| 模板 | 用途 |
| --- | --- |
| .aider.conf.yml | 项目配置模板（含多模型、lint、test 配置），复制到项目根目录 |

---

## 延伸阅读

- Aider 官方文档
- Aider GitHub（42k+ star）
- superpowers-zh — Skills 方法论（也支持 Aider）

# Gemini CLI 最佳实践

> Gemini CLI 是 Google 的命令行 AI 编程工具。最大优势：**免费额度充足 + 超大上下文窗口（2M tokens）**。适合大代码库分析、长任务执行、以及预算敏感的个人开发者。

---

## 核心概念

| 概念 | 说明 | 用途 |
| --- | --- | --- |
| GEMINI.md | 项目配置文件 | 类似 Claude Code 的 CLAUDE.md |
| Tools | 内置工具（读写文件、执行命令等） | Agent 能力基础 |
| Extensions | 扩展插件 | 连接 Google 服务和第三方 API |
| Context Window | 2M tokens | 能一次性理解超大代码库 |
| Sandbox | 安全沙箱 | 隔离执行不信任的代码 |

---

## 快速上手

### 安装

```
npm install -g @google/gemini-cli
```

安装后运行 `gemini` 进入交互模式，按提示登录 Google 账号即可。

> 最新安装方式请参考 官方仓库。

**GEMINI.md — 项目配置**

```
# 项目背景
这是一个 Go 微服务项目，包含 12 个服务。

# 目录结构
- cmd/          — 各服务入口
- internal/     — 内部包
- pkg/          — 公共包
- proto/        — Protobuf 定义
- deployments/ — K8s 配置

# 常用命令
- 编译所有服务：make build-all
- 跑测试：make test
- 生成 proto：make proto-gen
- 本地启动：docker compose up

# 注意
- 服务间通信用 gRPC，不要用 HTTP
- 配置统一走 Viper，不要硬编码
```

---

## 超大上下文的正确用法

Gemini CLI 的 2M tokens 上下文窗口是它的核心优势。但大不等于好——关键是用对。

**适合大上下文的场景**

```
# 1. 整个项目架构分析
读取整个 src/ 目录，画出模块依赖关系图。
哪些模块耦合度最高？给出解耦建议。

# 2. 全局重构评估
如果要把 ORM 从 GORM 换成 sqlc，
影响范围有多大？列出所有需要改的文件。

# 3. 跨服务问题排查
用户下单后没收到通知。
从 order-service 到 notification-service 的完整调用链，
每个环节的代码都看一下，找出哪里断了。
```

**不适合大上下文的场景**

❌ 不要把整个 node_modules 塞进去
❌ 不要一次分析 10 万行代码然后问"有没有 bug"
❌ 不要用大上下文替代精确定位

---

## 提示词技巧

**1. 利用免费额度做批量分析**

```
依次检查 src/ 下每个 Go 文件的错误处理：
1. 有没有忽略 error 返回值（_ = xxx）
2. 有没有用 fmt.Println 打印错误而不是 log
3. 有没有 panic 在非 main 函数里
列出所有问题，按文件分组。
```

#### 2. 代码库导航

```
我刚接手这个项目，帮我理解：
1. 请求从 HTTP 进来到返回响应，经过哪些层？
2. 数据库操作封装在哪一层？
3. 认证和授权是怎么做的？
4. 有没有文档或注释说明架构决策？
给我一份简洁的架构概览。
```

#### 3. 代码迁移评估

```
这个项目想从 Python 2 迁移到 Python 3。
扫描所有 .py 文件，找出：
1. print 语句（不是函数）
2. unicode/str 类型问题
3. 已废弃的标准库用法
4. 不兼容的第三方库版本
按迁移优先级排序。
```

---

## 与 Claude Code 的区别

| 维度 | Gemini CLI | Claude Code |
|---|---|---|
| 上下文窗口 | **2M tokens**（最大） | 200K tokens |
| 免费额度 | **充足** | 有限 |
| Agent 能力 | ★★☆ | ★★★ |
| 工具生态 | Google 服务集成好 | MCP 生态最丰富 |
| Skill 支持 | 有（`.gemini/skills/`） | 有（`.claude/skills/`） |
| 适合 | 大代码库分析、预算敏感 | 复杂 Agent 任务、需要强执行力 |

**建议组合**：用 Gemini CLI 做大规模分析和理解，用 Claude Code 做精确的修改和执行。

---

## 常见陷阱

| 陷阱 | 说明 | 解决 |
| --- | --- | --- |
| 上下文太大反而慢 | 2M 全塞满处理很慢 | 只在需要全局视角时用大上下文 |
| 免费额度用完 | 高频使用会触发限制 | 合理安排，大任务集中做 |
| 工具能力弱于 Claude Code | 文件操作、命令执行不如 CC | 分析用 Gemini，执行用 CC |

---

## 配置模板

| 模板 | 用途 |
| --- | --- |
| GEMINI.md | 项目配置文件模板，复制到项目根目录后按需修改 |

---

## 延伸阅读

- Gemini CLI 官方文档
- gemini-cli-tips — Addy Osmani 的 30 个技巧
- superpowers-zh — Skills 方法论（也支持 Gemini CLI）

# Windsurf 最佳实践

> Windsurf 是 Codeium 推出的 AI IDE，核心特色是 **Cascade** — 一个能感知你编辑行为的 AI Agent。它会自动追踪你的操作（文件切换、代码修改、终端输出），主动提供帮助，而不是等你提问。

---

## 核心概念

| 概念 | 说明 | 用途 |
| --- | --- | --- |
| Cascade | 上下文感知 Agent | 自动理解你在做什么，主动建议 |
| Flows | 操作流追踪 | 记录你的编辑轨迹作为上下文 |
| Write 模式 | 直接写代码 | 类似 Cursor Composer |
| Chat 模式 | 对话问答 | 理解代码、问问题 |
| Rules | `.windsurfrules` | 项目级规则配置 |
| @引用 | `@file` `@folder` `@web` 等 | 精确指定上下文 |

---

## 快速上手

### 安装

从 windsurf.com 下载安装，支持 macOS / Windows / Linux。基于 VS Code，现有 VS Code 扩展大部分兼容。

### .windsurfrules — 项目规则

在项目根目录创建 `.windsurfrules`：

```
# 项目规则

## 技术栈
Vue 3 + TypeScript + Pinia + Element Plus

## 代码规范
- 使用 Composition API（setup 语法糖）
- 组件用 PascalCase，文件用 kebab-case
- Store 按功能模块拆分
- API 请求统一走 src/api/ 下的封装

## Cascade 行为
- 修改组件时自动检查 Props 类型是否正确
- 修改 API 接口时提醒更新对应的 TypeScript 类型
- 不要自动重构我没要求改的代码
```

#### Cascade 的正确用法

Cascade 会追踪你的操作。利用这一点：

```
1. 先手动打开几个相关文件（让 Cascade 知道你在关注什么）
2. 做一个小修改（让 Cascade 理解你的意图）
3. 这时候 Cascade 的建议会比直接提问更准确
```

---

## 提示词技巧

#### 1. Write 模式 — 大任务

```
用 Write 模式。
把 src/views/user/ 下的用户管理页面从 Options API 迁移到 Composition API。
保持功能不变，只改写法。
一个文件一个文件来，每个文件改完让我确认。
```

#### 2. 利用 Flows 上下文

```
# 你刚在终端跑了测试，看到了报错
# Cascade 已经知道了，直接说：
刚才测试报错了，帮我看看什么原因。
# 不需要贴报错信息，Cascade 已经从 Flow 里拿到了
```

#### 3. @ 引用

```
@src/api/user.ts @src/types/user.ts
这两个文件的类型定义不一致，帮我统一。
以 types/user.ts 为准。
```

---

## 与 Cursor 的区别

| 维度 | Windsurf | Cursor |
| --- | --- | --- |
| 核心理念 | **主动感知**，追踪你的操作自动提供帮助 | **按需触发**，你提问它才响应 |
| Agent | Cascade（自动追踪 Flows） | Composer（手动触发） |
| 上下文 | 自动从操作流推断 | 需要手动用 @ 引用 |
| Rules | `.windsurfrules` 单文件 | `.cursor/rules/` 可按 globs 拆分 |
| 模型 | 自研 + Claude/GPT | Claude/GPT/自研 |
| 适合 | 喜欢 AI 主动帮忙的人 | 喜欢精确控制的人 |

---

## 常见陷阱

| 陷阱 | 说明 | 解决 |
| --- | --- | --- |
| Cascade 太主动 | 你只是浏览代码它就开始建议修改 | 设置里调整 Cascade 敏感度 |

| 陷阱 | 说明 | 解决 |
| --- | --- | --- |
| Flows 上下文错乱 | 切了太多文件，Cascade 搞不清你在做什么 | 开新 Cascade 会话，重新开始 |
| Write 模式范围失控 | 改了不该改的文件 | 用 @ 引用限定范围 |

---

## 配置模板

| 模板 | 用途 |
| --- | --- |
| .windsurfrules | 项目规则模板（Vue 3 + TypeScript），复制到项目根目录重命名为 `.windsurfrules` |

---

## 延伸阅读

- Windsurf 官方文档
- awesome-windsurf — 社区资源集合
- superpowers-zh — Skills 方法论（也支持 Windsurf）

# Trae 最佳实践

> Trae 是字节跳动推出的 AI IDE（基于 VS Code），主打**免费使用 Claude 和 GPT 模型**。对中国开发者友好——界面支持中文、国内网络直连、免费额度充足。适合入门 AI 编程和预算敏感的团队。

---

## 核心概念

| 概念 | 说明 | 用途 |
| --- | --- | --- |
| Builder | Agent 模式，自主完成任务 | 复杂任务、跨文件修改 |
| Chat | 侧边栏对话 | 问答、理解代码 |
| 补全 | 行内代码补全 | 日常编码 |
| Rules | `.trae/rules/` | 项目级规则 |
| @引用 | `@file` `@folder` `@web` | 精确指定上下文 |

---

## 快速上手

### 安装

从 trae.ai 下载安装，支持 macOS / Windows / Linux。

注册后直接可用，**不需要自己的 API Key**。

### Rules — 项目规则

创建 `.trae/rules/project_rules.md`：

```
# 项目规则

## 技术栈
React + TypeScript + Ant Design + UmiJS

## 代码规范
- 使用函数组件 + Hooks
- 状态管理用 @umijs/max 内置的 Model
- 请求用 umi-request，不要直接用 fetch/axios
- 样式用 CSS Modules

## 中文规范
- 注释用中文
- commit message 用中文
- 变量名用英文
```

#### Builder — Agent 模式

Builder 是 Trae 的 Agent，类似 Cursor 的 Composer：

```
用 Builder 模式。
参考 src/pages/user/list.tsx 的写法，
新建一个 src/pages/order/list.tsx 订单列表页。
要求：
1. 表格用 Ant Design ProTable
2. 支持按日期、状态、金额筛选
3. 支持导出 Excel
4. 操作列：查看详情、取消订单、退款
```

---

## 提示词技巧

#### 1. 利用免费模型

Trae 免费提供 Claude 和 GPT，合理分配：

```
# 复杂任务 — 用 Claude
Builder 模式 + Claude：重构整个认证模块

# 简单任务 — 用 GPT
Chat 模式 + GPT：解释这段代码 / 写个注释
```

#### 2. 中文友好

Trae 对中文支持最好，可以完全用中文交互：

```
帮我把 src/utils/request.ts 的请求封装改一下：
1. 加上统一的 loading 状态管理
2. 错误提示用 Ant Design 的 message 组件
3. 401 错误自动跳转登录页
4. 网络超时设为 10 秒
```

#### 3. Ant Design 生态

Trae 对国内常用组件库支持好：

```
@https://ant-design.antgroup.com/components/table-cn
参考 Ant Design 官方文档，
给 ProTable 加一个自定义的行展开功能，
展开后显示订单的商品明细。
```

---

## 与 Cursor 的区别

| 维度 | Trae | Cursor |
| --- | --- | --- |
| 价格 | **免费**（含 Claude/GPT） | $20/月 |
| 中文支持 | ★★★（界面中文） | ★☆☆（纯英文） |
| 国内网络 | ★★★（直连） | ★☆☆（需要代理） |
| Agent 能力 | ★★☆ | ★★★ |

| 维度 | Trae | Cursor |
| --- | --- | --- |
| Rules 系统 | ★★☆ | ★★★（globs 按需加载） |
| 插件生态 | ★★☆（VS Code 兼容） | ★★★（VS Code 兼容） |
| 适合 | 入门、预算敏感、国内团队 | 进阶、愿意付费、追求最强 |

---

## 常见陷阱

| 陷阱 | 说明 | 解决 |
| --- | --- | --- |
| Builder 执行慢 | 免费模型可能排队 | 非紧急任务用 Builder，急的用 Chat |
| 模型切换 | 不同模型擅长不同事 | 复杂用 Claude，简单用 GPT |
| Rules 不生效 | 文件路径或格式不对 | 确保在 `.trae/rules/` 下，Markdown 格式 |

---

## 配置模板

| 模板 | 用途 |
| --- | --- |
| project_rules.md | 项目规则模板（React + UmiJS + Ant Design），复制到 `.trae/rules/` |

---

## 延伸阅读

- Trae 官方网站
- superpowers-zh — Skills 方法论（也支持 Trae）

# Kiro 最佳实践

> Kiro 是 AWS 推出的 AI IDE，核心特色是 **Spec-driven Development**（规格驱动开发）。不是让 AI 直接写代码，而是先生成需求规格、设计文档，确认后再按规格实现。适合团队协作和需要高质量交付的场景。

---

## 核心概念

| 概念 | 说明 | 用途 |
| --- | --- | --- |
| Spec | 需求规格文档 | AI 先写规格，确认后再写代码 |
| Steering | `.kiro/steering/*.md` | 项目级规则和指引 |
| Hooks | 自动化触发器 | 文件保存时自动验证/测试 |
| Agent | 后台自主执行 | 复杂任务自动完成 |

---

## 快速上手

### 安装

从 kiro.dev 下载安装。需要 AWS 账号或 GitHub 账号登录。目前处于预览阶段。

### Steering 文件 — 项目配置

在 `.kiro/steering/` 下创建规则文件：

```
---
mode: always
---

# 项目规则

## 技术栈
Java 17 + Spring Boot 3 + MyBatis Plus + MySQL 8

## 代码规范
- Controller 层只做参数校验和路由
- Service 层处理业务逻辑
- DAO 层只做数据库操作
- 统一用 Result<T> 包装返回值
- 异常用自定义 BusinessException

## 命名规范
- 包名全小写
- 类名 PascalCase
- 方法名 camelCase
- 常量 UPPER_SNAKE_CASE
```

Steering 支持三种模式：

- `always` — 每次对话都加载
- `globs: ["*.java"]` — 只在操作匹配文件时加载
- `manual` — 手动激活

#### Spec 驱动开发

Kiro 的工作流和其他工具不同：

```
1. 你描述需求
2. Kiro 生成 Spec（需求规格 + 技术设计 + 测试用例）
3. 你审查和修改 Spec
4. 确认后 Kiro 按 Spec 逐步实现
5. 每步实现后自动运行相关测试
```

这比"直接写代码"慢，但产出质量更高，特别适合：

- 团队协作（Spec 可以 review）

- 复杂功能（先想清楚再动手）
- 需要文档的项目

---

## 提示词技巧

#### 1. 让 Spec 更精确

```
给订单模块加一个退款功能。

业务规则：
- 订单支付后 7 天内可退款
- 部分退款和全额退款都支持
- 退款需要审批（金额 > 500 元）
- 退款到原支付渠道

请先生成 Spec，我确认后再实现。
```

#### 2. 利用 Hooks 自动验证

```
// .kiro/hooks.json
{
  "on-save": {
    "*.java": "mvn compile -q",
    "*.test.java": "mvn test -pl ${module} -q"
  }
}
```

每次保存 Java 文件自动编译，保存测试文件自动跑测试。

#### 3. Steering 按模块配置

```
.kiro/steering/
├── always.md              # 全局规则（always）
├── api.md                 # API 规则（globs: src/controller/**）
├── database.md            # 数据库规则（globs: src/mapper/**）
└── testing.md             # 测试规则（globs: src/test/**）
```

---

## 与其他工具的区别

| 维度 | Kiro | Claude Code | Cursor |
| --- | --- | --- | --- |
| 核心理念 | Spec 先行 | Agent 执行 | IDE 补全 |
| 适合 | 团队协作、高质量交付 | 个人高效开发 | 日常编码 |
| 规则系统 | Steering（三种模式） | CLAUDE.md + Skills | Rules（globs） |
| 开发流程 | 需求→规格→实现→验证 | 需求→实现→验证 | 需求→实现 |
| AWS 集成 | ★★★ | ★☆☆ | ★☆☆ |

---

## 配置模板

| 模板 | 用途 |
| --- | --- |
| steering-always.md | Steering 全局规则模板（Java + Spring Boot），复制到 `.kiro/steering/` |

---

## 延伸阅读

- Kiro 官方文档
- superpowers-zh — Skills 方法论（也支持 Kiro）

# 卷三 · 架构：把 AI 编程接进团队

## 这一卷是给谁看的

- 团队 lead / staff engineer：要为整个团队选型 AI 工具栈
- 平台工程师：要把 AI Agent 接进 CI / 代码评审 / 安全扫描流水线
- CTO / 安全工程师：担心数据泄漏、prompt injection、商业机密外流
- 独立开发者要做付费产品：想做 MCP server / Skill 出售或开源

**为什么单独成卷**：卷一卷二解决"我个人用得好"。但你真要把 AI 编程接进 5 人以上团队、放进 CI、给企业用——会撞上一堆 individual contributor 视角根本看不见的问题：

- 怎么保证 AI 不会把生产配置写错？（Sandbox + Execpolicy）
- 怎么防止某个开发者的 prompt 把客户邮件泄漏到 OpenAI？（DLP + 审计）
- 怎么让 AI 接团队 Linear / Slack / 数据库而不需要每人配 API key？（MCP 治理）
- 怎么衡量 AI 工具的真实 ROI 而不是"感觉变快了"？（监控 + 用量分析）

这一卷就是这些问题的实战手册。

## 这一卷的逻辑顺序

```
安全注意事项（先把红线讲清楚）
   ↓
CI 集成（codex-action / claude-code 自动 review / 自定义 Action）
   ↓
Sandbox 与 Execpolicy（OS 内核级 vs 应用层 hook 的取舍）
   ↓
MCP 生态（哪些值得装、哪些是噪音）
   ↓
团队治理（AGENTS.md.override / 配置层叠 / 审计日志）
   ↓
成本控制（FinOps 视角看 AI 编程支出）
```

学完这一卷你应该能：

- 给团队制定一份"AI 编程使用规范"——明确什么任务能给 AI、什么必须人审
- 设计一套 CI 流水线：PR 触发 AI review，发现风险用 Sandbox 跑试探，自动评论

- 选型 MCP server：知道哪几个能消除手工动作，哪几个是博眼球的玩具
- 给老板讲清楚：花在 AI 工具上的钱，每月省了多少工程师小时

## 关于 OpenClaw

卷三末尾会讲 OpenClaw——它不是编程工具，是 Agent 框架，但跟编程团队的"非编程自动化"高度相关：定时任务、跨平台消息（Telegram / Slack）、Cron + Skill。如果你的团队需要"AI 帮忙做编程之外的事"（如每周扫一遍线上 bug、自动整理 PR 报告），OpenClaw 是开源选择。

## 节奏建议

这一卷比前两卷更"工程"——建议至少有过一次"我把 AI 工具引进团队但搞砸了"的经验之后再读，否则容易觉得"我现在用得好好的，要这么复杂吗"。

如果是新建团队 / 刚招到第一个 AI tooling 工程师——直接顺读。这一卷里的每一章都对应一个"半年内一定会撞上"的真实问题。

# AI 编程安全注意事项

> AI 写的代码可能包含安全漏洞。不是因为 AI 故意写不安全的代码，而是它倾向于"先让功能跑起来"，安全性往往不是它的第一优先级。

---

## 高频风险

#### 1. 敏感信息硬编码

AI 经常为了快速演示把密钥写死在代码里：

❌ AI 可能这样写：

```
const API_KEY = "sk-1234567890abcdef"
const DB_URL = "postgresql://admin:password@prod-db:5432/myapp"
```

✅ 应该这样：

```
const API_KEY = process.env.API_KEY
const DB_URL = process.env.DATABASE_URL
```

**防护**：在项目规则里明确禁止：

```
# CLAUDE.md / .cursorrules
禁止在代码中硬编码任何密钥、密码、token。
所有敏感配置必须通过环境变量读取。
```

#### 2. SQL 注入

❌ AI 可能这样写（特别是快速实现时）：

```
const query = `SELECT * FROM users WHERE name = '${name}'`
```

✅ 应该这样：

```
const query = `SELECT * FROM users WHERE name = $1`
await db.query(query, [name])
```

#### 3. 缺少权限校验

AI 写 CRUD 接口时经常忘记权限：

❌ AI 可能这样写：

```
app.delete('/api/users/:id', async (req, res) => {
  await db.users.delete(req.params.id)  // 任何人都能删任何用户！
})
```

✅ 应该这样：

```
app.delete('/api/users/:id', authMiddleware, async (req, res) => {
  if (req.user.role !== 'admin' && req.user.id !== req.params.id) {
    return res.status(403).json({ message: '无权限' })
  }
  await db.users.delete(req.params.id)
})
```

#### 4. 敏感数据日志泄露

❌ AI 调试时可能加的日志：

```
console.log('Login request:', { email, password })  // 密码被打印了！
console.log('Payment response:', response)           // 可能包含卡号
```

✅ 应该这样：

```
console.log('Login request:', { email, password: '***' })
console.log('Payment response:', { id: response.id, status: response.status })
```

---

## 防护清单

在项目配置文件中加入这些规则：

```
# 安全规则（加到 CLAUDE.md / .cursorrules / .windsurfrules）

## 禁止
- 不要硬编码密钥、密码、token
- 不要在日志中打印敏感数据（密码、token、卡号、身份证号）
- 不要把用户输入直接拼接到 SQL/Shell 命令中
- 不要禁用 HTTPS 验证
- 不要在前端存储敏感 token（用 httpOnly cookie）

## 必须
- 所有接口必须有权限校验
- 用户输入必须做参数校验和转义
- 文件上传必须校验类型和大小
- 密码必须 hash 存储（bcrypt/argon2）
- API 必须有速率限制
```

---

## AI 代码安全审查

定期让 AI 自查安全问题：

```
扫描整个 src/ 目录，按 OWASP Top 10 检查：
1. 注入（SQL、NoSQL、命令注入）
2. 失效的身份认证
3. 敏感数据暴露
4. XML 外部实体
5. 失效的访问控制
6. 安全配置错误
7. 跨站脚本（XSS）
8. 不安全的反序列化
9. 使用含已知漏洞的组件
10. 不足的日志和监控

每个问题给出：文件位置、风险等级、修复建议。
```

# OpenClaw 最佳实践

> OpenClaw 是一个开源的 AI 个人助手框架（33 万+ GitHub Stars），核心特色是 **本地运行 + 多平台连接 + 自主执行任务**。它不只是聊天机器人，而是能浏览网页、读写文件、执行命令、调度定时任务的 AI Agent。支持 Claude、GPT、DeepSeek、本地模型等多种 LLM。

---

## 核心概念

| 概念 | 说明 | 用途 |
| --- | --- | --- |
| Gateway | 后台常驻的 WebSocket 网关 | 路由消息、管理 Agent |
| Channel | 消息平台连接 | 接入 WeChat、Telegram、Slack、Discord 等 |
| Skill | `SKILL.md` 定义的能力模块 | 教 AI 怎么做事（类似 Claude Code 的 Skills） |
| Agent | 独立的 AI 工作空间 | 不同任务用不同 Agent |
| Cron | 定时任务调度 | 自动化周期性工作 |
| Tool | 内置工具（浏览器、文件、Shell） | 让 AI 有"手"去执行操作 |

---

## 快速上手

### 安装

```
# macOS / Linux / WSL2（推荐）
curl -fsSL https://openclaw.ai/install.sh | bash

# 或用 npm
npm install -g openclaw@latest
openclaw onboard --install-daemon

# 验证安装
openclaw --version
openclaw doctor
```

系统要求：Node.js 22.14+（推荐 24）

### 初始化

```
# 交互式引导（设置模型、频道等）
openclaw onboard

# 查看网关状态
openclaw gateway status

# 启动网关
openclaw gateway start
```

### 配置文件

配置文件位于 `~/.openclaw/openclaw.json`（JSON5 格式，支持注释）：

```
{
  // 模型设置
  "models": {
    "default": "claude-sonnet-4-20250514",
    // 也可以用 DeepSeek 省钱
    // "default": "deepseek/deepseek-chat",
  },

  // 频道（按需开启）
  "channels": {
    "telegram": { "enabled": true },
    "webchat": { "enabled": true },
  },
}
```

修改后自动热加载，不需要重启。

---

## 提示词技巧

#### 1. Skills — 教 AI 新能力

OpenClaw 的 Skills 和 Claude Code 的 Skills 类似，用 Markdown 定义：

```
---
name: code-reviewer
description: 审查代码质量，找出安全漏洞和性能问题
user-invocable: true
---

# 代码审查

你是一个资深代码审查员。当用户让你审查代码时：

1. 先通读整个文件，理解上下文
2. 检查安全问题（注入、XSS、敏感数据泄露）
3. 检查性能问题（N+1 查询、内存泄漏）
4. 检查代码规范（命名、结构、注释）
5. 给出具体修改建议，附带代码示例
```

Skills 的三个加载位置（优先级从高到低）：

```
项目目录/skills/      → 项目级（最高优先级）
~/.openclaw/skills/   → 全局级
内置 skills/          → 默认（最低优先级）
```

安装社区 Skill：

```
# 从 ClawHub 安装
openclaw skills install <skill-slug>

# 查看已安装的 Skills
openclaw skills list
```

#### 2. 多模型策略

```
# 查看可用模型
openclaw models list

# 设置默认模型
openclaw models set default claude-sonnet-4-20250514

# 复杂任务用 Claude
openclaw models set default claude-sonnet-4-20250514

# 日常对话用 DeepSeek（便宜）
openclaw models set default deepseek/deepseek-chat

# 完全免费用本地模型
openclaw models set default ollama/qwen2.5-coder
```

#### 3. 自动化任务

OpenClaw 支持 Cron 定时任务，适合自动化：

```
# 每天早上 9 点发送代码库健康报告
openclaw cron add "0 9 * * *" "检查项目代码质量，生成报告发送给我"

# 每周一早上发送周报摘要
openclaw cron add "0 9 * * 1" "汇总上周的 Git 提交和 PR，生成周报"

# 查看所有定时任务
openclaw cron list
```

#### 4. 多频道协作

OpenClaw 的独特优势是连接多个消息平台：

```
# 添加频道
openclaw channels add telegram
openclaw channels add webchat

# 查看频道状态
openclaw channels status
```

应用场景：

- **Telegram** — 随时随地发消息让 AI 执行任务
- **WebChat** — 浏览器界面做复杂交互
- **Slack/飞书** — 团队协作，AI 作为团队成员

---

## 进阶技巧

#### Agent 工作空间

不同项目用不同 Agent，隔离上下文：

```
# 创建新 Agent
openclaw agents add my-project

# 查看所有 Agent
openclaw agents list

# 删除不需要的 Agent
openclaw agents delete old-project
```

**常用 CLI 命令速查**

| 命令 | 用途 |
| --- | --- |
| `openclaw onboard` | 交互式初始化 |
| `openclaw gateway start/stop/status` | 管理网关 |
| `openclaw channels add/remove/status` | 管理消息频道 |
| `openclaw models list/set` | 管理模型 |
| `openclaw skills list/install` | 管理 Skills |
| `openclaw cron add/list` | 定时任务 |
| `openclaw agents list/add/delete` | 管理 Agent 工作空间 |
| `openclaw doctor` | 健康检查和诊断 |
| `openclaw logs` | 查看网关日志 |

**调试和排查**

```
# 健康检查（最有用的排查命令）
openclaw doctor

# 查看实时日志
openclaw logs

# 开发模式（更多调试信息）
openclaw gateway --dev
```

---

## 与其他工具的区别

| 维度 | OpenClaw | Claude Code | Cursor |
|---|---|---|---|
| 类型 | AI Agent 框架 | CLI 编程助手 | AI IDE |
| 核心场景 | 多平台自动化 | 代码编写和重构 | 日常编码 |
| 运行方式 | 后台常驻（Gateway） | 按需启动 | IDE 内嵌 |
| 消息平台 | 20+ 平台 | 仅终端 | 仅 IDE |
| 模型支持 | Claude/GPT/DeepSeek/本地 | 仅 Claude | 多模型 |
| Skills | SKILL.md | .claude/skills/ | Rules |
| 定时任务 | ✅ 内置 Cron | ❌ | ❌ |
| 开源 | ✅ (MIT) | ❌ | ❌ |
| 适合 | 自动化、多平台、全能助手 | 专业编程 | 日常编码 |

**OpenClaw vs Claude Code**：不是替代关系，而是互补。Claude Code 专注编程，OpenClaw 专注自动化和多平台连接。可以同时使用。

---

## 常见陷阱

| 陷阱 | 说明 | 解决 |
| --- | --- | --- |
| Node 版本不够 | 需要 22.14+ | `nvm install 24` |
| Gateway 启动失败 | 端口被占用或配置错误 | `openclaw doctor` 诊断 |
| Skill 不生效 | 路径或格式不对 | 检查 `SKILL.md` frontmatter 的 name 和 description |
| 模型 API 报错 | Key 未设置或余额不足 | `openclaw models status` 检查 |
| 频道连接断开 | 网络或认证问题 | `openclaw channels status` + `openclaw channels reconnect` |

---

## 配置模板

| 模板 | 用途 |
| --- | --- |
| code-reviewer.md | 代码审查 Skill 模板，复制到 `~/.openclaw/skills/` 或项目 `skills/` 目录 |

---

## 延伸阅读

- OpenClaw 官方文档
- OpenClaw GitHub（338k+ stars）
- ClawHub — Skill 市场
- superpowers-zh — Skills 方法论（也支持 OpenClaw）

# 延伸学习资源

> 本项目讲"9 款工具怎么用好"。要继续深入，这里是精选外部资源。**不求全，只求高质量**——每个条目都说清楚"什么情况值得看"。
>
> 信息截止：**2026-04**。链接失效请提 Issue。

---

## 🧭 快速导航

| 想解决什么 | 去哪看 |
| --- | --- |
| 系统学 Prompt 工程 | 官方交互教程 / 吴恩达课 / DAIR 指南 |
| 找工具的最佳实践集 | awesome 列表系列 |
| Claude Code 进阶 | 官方仓库 / Hooks / Cookbook |
| MCP 服务器生态 | MCP 官方 + awesome-mcp-servers |
| 中文高质量参考 | Datawhale / 黄峰达 / AutoDev |
| 追新工具动态 | 博客 / 播客 / Newsletter |

---

## 📝 Prompt 工程

- anthropics/prompt-eng-interactive-tutorial · ⭐ 高 · 英
  - Anthropic 官方 **9 章交互式 Prompt 教程**，可运行 Jupyter
  - 系统入门 Prompt 工程最快的路径，2-3 小时能跑完
- dair-ai/Prompt-Engineering-Guide · ⭐ 60k+ · 英（有中文翻译）
  - Prompt 工程百科全书：CoT、ReAct、RAG、Tool-use 各种 pattern

- 想深入原理、看各种技术名词对应什么场景时翻

- datawhalechina/prompt-engineering-for-developers · ⭐ 13k+ · 中
  - Datawhale 翻译的**吴恩达 Prompt 工程系列**+ 中文注解
  - 中文用户看视频 + 跟代码的最佳组合

- phodal/prompt-patterns · ⭐ 1k+ · 中
  - 国内知名架构师黄峰达写的 Prompt 模式与 DSL 设计指南
  - 写复杂 AI 编程工作流、设计可复用 Prompt 时的进阶参考

---

## 🛠️ 工具专项 Awesome 列表

- PatrickJS/awesome-cursorrules · ⭐ 30k+ · 英
  - 社区收集的 `.cursorrules` 文件大全，按技术栈分类
  - 新项目起手抄一份对应技术栈的 rules

- hesreallyhim/awesome-claude-code · ⭐ 高 · 英
  - Claude Code 社区最全资源：技巧、Skills、Hooks、工作流
  - Claude Code 进阶后想找社区最佳实践

- github/awesome-copilot · ⭐ 高 · 英
  - GitHub 官方维护的 Copilot 资源集合
  - 查 Copilot 的 instruction / Chat mode / Agent 示例

- addyosmani/gemini-cli-tips · ⭐ 中 · 英
  - Google 开发者 Addy Osmani 整理的 **30 个 Gemini CLI 技巧**
  - Gemini CLI 用户直接抄作业

- detailobsessed/awesome-windsurf · ⭐ 中 · 英
  - Windsurf 社区资源集合（规则、工作流、MCP）
  - 用 Windsurf Cascade 找社区模式

---

## 🤖 Claude Code 进阶

- anthropics/claude-code · 官方 · 英
  - **官方仓库**，Issues 和 Discussions 是新功能 / 新技巧的一手信息源
  - 追 Claude Code 新特性（slash command / hooks / skill 新能力）
- anthropics/courses · ⭐ 高 · 英
  - Anthropic 官方课程合集：Prompt、Tool use、RAG、MCP 全套
  - 想系统吃透 Anthropic 全家桶
- anthropics/anthropic-cookbook · ⭐ 高 · 英
  - Claude API 实战示例：多模态、工具调用、RAG、Agent 模式
  - 用 Claude API 做定制化 AI 编程工具时的参考
- disler/claude-code-hooks-mastery · ⭐ 中 · 英
  - Claude Code Hooks 深度实战
  - 做自动化 Hook 工作流（质量门禁、通知、检查）时直接抄

---

## 🔌 MCP 生态

- modelcontextprotocol/servers · 官方 · 英
  - **MCP 官方服务器合集**：文件系统、数据库、Git、Slack 等
  - 给 Claude Code / Cursor 加外部能力时第一站
- punkpeye/awesome-mcp-servers · ⭐ 高 · 英
  - 社区维护的 MCP 服务器列表，更新频繁
  - 找第三方 MCP（浏览器、PostgreSQL、特定 API）

---

## 🏗️ Agent 工程化

- humanlayer/12-factor-agents · ⭐ 高 · 英

- 构建**生产级 LLM Agent 的 12 条原则**，2025 工程化标杆
- 从"玩票 Agent"进阶到"能上线的 Agent"必读

- x1xhlol/system-prompts-and-models-of-ai-tools · ⭐ 高 · 英
  - 整理的 Cursor、Windsurf、v0、Devin 等工具的**系统 Prompt**
  - 想逆向学习顶级 AI 编程工具怎么写 Prompt
- openai/openai-cookbook · ⭐ 高 · 英
  - OpenAI 官方示例：工具调用、结构化输出、Agent 模式
  - 跨 LLM 参考，Copilot / Codex 用户交叉看

---

## 🇨🇳 中文优秀资源

- datawhalechina/llm-cookbook · ⭐ 22k+ · 中
  - Datawhale 的 LLM 开发中文指南合集
  - 把 LLM 能力嵌入自己项目时的中文参考手册
- liaokongVFX/LangChain-Chinese-Getting-Started-Guide · ⭐ 8k+ · 中
  - LangChain 中文入门指南：Agent / Chain / Memory
  - 自己造 AI 编程工具、Agent 型助手的基础
- phodal/aigc · ⭐ 1k+ · 中
  - 黄峰达《构筑大语言模型应用》电子书
  - AI 编程架构师视角，适合看工程化与团队落地
- unit-mesh/auto-dev · ⭐ 3k+ · 中英
  - **国产 JetBrains AI 编程插件**，支持自定义 Agent + 中文 DevIns 脚本
  - IDEA/PyCharm 生态里的 AI 编程选择

---

## 📡 博客 / 播客 / Newsletter

- Anthropic Engineering Blog · 英

    - Anthropic 官方工程博客，Claude Code 新特性和最佳实践首发
    - 每月更新，**必订**

- Simon Willison's Weblog · 英

    - LLM 圈最勤奋独立博主，几乎每天更新 AI 工具实测
    - 追新工具发布和 CLI 级技巧

- Latent Space · 英

    - swyx 主持的 AI 工程播客 + newsletter
    - 采访 Cursor / Anthropic / GitHub 核心团队的第一手观点

- 宝玉的分享 · 中

    - 宝玉翻译整理的大量 AI 编程英文好文**中文版**
    - 不想读英文原文时的高质量中转站

---

## 📌 本项目与外部资源的关系

这个项目定位**"中文 AI 编程工具实战指南"**——覆盖 9 款工具的具体使用。外部资源是补充：

```
外部资源教原理和通用方法 → 本项目教具体工具怎么用
    ↓                              ↓
  Prompt 工程 / Agent 设计     Claude Code 66 技巧
  LLM 应用架构                 Cursor 配置模板
  MCP 生态                      陷阱合集与实战脚本
```

不重复造轮子，有外部好资源就指过去。

---

## 贡献

发现好资源请提 PR 加到对应分类。**质量要求**：

- 仓库至少 500 star，或明确为官方/权威资源
- 持续更新（一次性博文不收录）
- 说清楚"什么场景值得看"，不能只有名字

- 不收录：付费课程、付费社群、个人微信号

详见 CONTRIBUTING.md。

# 别册 · 速查手册

完整 9 工具速查表与卷一·入门章节是同一份内容，避免重复，请直接看：

👉 **卷一 · 9 工具速查表**

---

## 速查手册的设计意图

- 案头放一份，每天翻两眼
- 写代码时遇到"那个命令叫啥来着"，搜这个就够
- 不带方法论、不带导读——只有冷冰冰的命令和参数表

如果你来到这里时感觉"内容怎么没看到"——这页就是个目录跳板。完整内容在卷一速查表。

# 更新日志 · Changelog

> 完整提交历史见 `git log`。本文档记录面向用户的重要更新。

## 2026-04（下半月） · 新增 OpenAI Codex CLI 教程

### 🆕 新增工具：Codex CLI（10 款工具齐了）

### 📂 `codex/` — OpenAI 官方开源终端 Agent（Rust，Apache-2.0）

- 基于 v0.125.0（2026-04-24）+ developers.openai.com/codex 官方文档核实
- 完整覆盖：核心概念 / 安装认证 / AGENTS.md 加载链 / Approval & Sandbox 双旋钮 / TOML Subagent / MCP / Skill / Plan Mode
- 与 Claude Code 横向对比（沙箱实现、定价模式、强项场景）
- 10 个高频踩坑（CI 卡 approval、配置不生效、多版本冲突等）
- 模板：`AGENTS.md` + `config.toml`（含 4 个 profile）+ `agent-explorer.toml`

### 🔄 同步更新

- **README.md / README.en.md**：「9 款工具」→「10 款」，工具表新增 Codex CLI 行，画像推荐路径加入"已订阅 ChatGPT Plus/Pro"
- **cheatsheet.md / cheatsheet.en.md**：能力矩阵新增 Codex 列与「沙箱机制」维度，配置文件表加入 AGENTS.md，命令速查新增 Codex 段，决策流与组合推荐双向更新

### 🔬 二次源码核实修正

教程发布前直接读 `codex-rs` 源码（`utils/cli/src/shared_options.rs`、`tui/src/cli.rs`、`exec/src/cli.rs`、`cli/src/main.rs`、`models-manager/models.json`）核对，修正 6 处不准确：

1. **Linux 沙箱**：`bubblewrap` → `Landlock` 内核 LSM（`debug_sandbox.rs` 直接 import `codex_sandboxing::landlock`）
2. **`--full-auto` 已废弃移除**：源码警告 "`--full-auto` *is deprecated; use* `--sandbox workspace-write` *instead.*"，所有示例统一替换

3. `--output-format json` **不存在**：实际 flag 是 `--json`（输出 JSONL，不是单 JSON）
4. `/side` **不存在**：从未在 `slash_commands` 文档里出现，删除；改成真实存在的 `/diff` `/mention` `/debug-config` `/new` `/clear`
5. `codex config path` / `codex config validate` **不存在**：CLI 没有 `config` 子命令，应该用 TUI 内的 `/debug-config`
6. **新增** `--yolo` / `--add-dir`：源码确认这两个 flag 是当前推荐的危险/扩展用法

模型名校验：`gpt-5.5` / `gpt-5.4` / `gpt-5.3-codex` 全部命中 `models-manager/models.json` 官方列表。

#### 🔬 第四轮 schema 级核实（config / subagent TOML 结构）

直接对照 Rust struct 定义（`config_toml.rs::ConfigToml`、`profile_toml.rs::ConfigProfile`、`config_toml.rs::AgentsToml/AgentRoleToml`、`core/src/config/agent_roles.rs::RawAgentRoleFileToml`）：

- `web_search` **是顶层 key**（合法值 `disabled` / `cached` / `live`），不在 `[features]` 下；template 已修
- `multi_agent` **不是 features 简单 bool**（多 agent 走 `[multi_agent_v2]` 自有结构）；template 已删
- **subagent TOML 字段** = `RawAgentRoleFileToml` flatten 了整个 `ConfigToml`，所以 `name` / `description` / `developer_instructions` / `model` / `model_reasoning_effort` / `sandbox_mode` / `approval_policy` 都合法；`#[serde(deny_unknown_fields)]` 兜底拼错会报错
- `model_reasoning_effort` **合法值**：`none` / `minimal` / `low` / `medium`(默认) / `high` / `xhigh`（来源 `protocol/src/openai_models.rs::ReasoningEffort`）
- `[agents]` **段并发参数**：`max_threads` / `max_depth` / `job_max_runtime_seconds` / `interrupt_message` 与源码 `AgentsToml` 结构一致
- `AGENTS_MD_MAX_BYTES = 32 * 1024`（即 32 KiB）—— 默认值已直接在源码确认

#### 📚 v2 内容补完（基于 GitHub 高 star 实战教程）

参考 RoggeOhta/awesome-codex-cli（280+ 资源）和 shanraisshan/codex-cli-best-practice（50 条战场技巧）后，补完 4 个原本没覆盖的真实功能：

- **Skills 真实文件夹结构**：原 v1 写成单文件 `SKILL.md`，源码 `core-skills/loader.rs` 确认是 **目录** 含 `references/` `scripts/` `examples/` `agents/openai.yaml`；canonical

路径 `~/.agents/skills/`（`~/.codex/skills/` 已废弃）；新增 `$skill-name` 显式调用语法

- **Hooks**（beta）：6 个事件 `PreToolUse` / `PermissionRequest` / `PostToolUse` / `SessionStart` / `UserPromptSubmit` / `Stop`，源码引擎名 `ClaudeHooksEngine`——schema 直接复用 Claude Code 的 `hooks.json`，迁移零改动
- **Memories**（beta）：`/memories` slash 命令 + `[memories] no_memories_if_mcp_or_web_search` 安全开关
- **Plugins / Marketplace**（v0.121.0+）：`codex plugin marketplace add/upgrade/remove`，TUI `/plugins`
- **Fast Mode**：`/fast on|off|status`，gpt-5.4 1.5× 速度 / 2× 额度

新增「高质量提示词与工作模式」整段，集合社区 5 类高信号技巧（提示词 / Plan / AGENTS.md / 多 Agent / 调试），并在 References 段补 6 个高 star 索引和 4 个主流 workflow 框架（Superpowers / Spec Kit / oh-my-codex / Compound Engineering）。

新增 `templates/SKILL.md` 模板（带触发器写法 + Gotchas 范本）。

#### 🔬 第六轮源码核实 + v3 内容补完

直接读 `codex-rs/tui/src/slash_command.rs::SlashCommand` 枚举（46 个真实 slash 命令），抓出第三轮的过修正：

- **`/side` 是真实存在的命令**（"start a side conversation in an ephemeral fork"）—— 第三轮凭 WebFetch 摘要错删，现已恢复
- **`/fast` 也确实在 enum 里**（"toggle Fast mode to enable fastest inference with increased plan usage"）—— v2 加上是对的
- **slash 命令表扩到 18 条**：补 `/skills` `/memories` `/plugins` `/apps` `/goal` `/rename` `/feedback` `/approvals` 等

`MemoriesToml` 源码确认 canonical 字段是 **`disable_on_external_context`**，`no_memories_if_mcp_or_web_search` 是它的 serde alias（向后兼容），文档全部改用 canonical 名。

新增 4 节真功能（每节都对照源码核实）：

- **§13 OSS 本地模型** — `--oss --local-provider lmstudio|ollama`（来源 `utils/oss/src/lib.rs`），完全离线 + 零 API 成本

- §14 官方 GitHub Action 实战 — `openai/codex-action@v1`（Apache-2.0，953 stars）的 PR 自动 review 完整 workflow YAML
- §15 Codex 作为 MCP server — `codex mcp-server` 暴露 `codex()` + `codex-reply()` 工具，让其他 Agent 反向调用
- §16 Rules / Execpolicy — Starlark DSL（`prefix_rule()` + `host_executable()`），`codex execpolicy check` 离线验证

---

## 2026-04 · 重大更新：从"工具教程"到"完整落地手册"

### 🆕 新增模块

#### 📋 速查表 `cheatsheet.md`

- 9 工具 × 13 维度横向能力矩阵
- 30 秒决策表（按需求选工具）
- 每个工具的配置文件位置 + 核心命令 + 快捷键速查
- 选型决策树 + 5 个推荐组合

#### ⚠️ 陷阱合集 `pitfalls/`

- 4 个工具共 **31 个深度陷阱**，按 "症状 / 根因 / 出坑 / 预防" 四段式展开
- Claude Code (8) / Cursor (8) / GitHub Copilot (8) / Aider (7)
- Windsurf / Gemini CLI / Kiro / Trae / OpenClaw 待社区贡献

#### 🎯 实战场景脚本 `workflows/scenarios.md`

- 3 个端到端对话脚本：重构模块 / Cursor+Claude Code 协作 / 补老项目测试
- 通用"打断话术"集锦

#### 🧭 画像推荐路径 主 README 重构

- 按 6 种用户画像推荐阅读路径（新手 / 前端 / 后端 / Copilot 迁移 / 控成本 / 团队协作）
- 顶部加入速查表入口

#### 📚 延伸学习资源 `resources.md`

- 20+ 精选外部资源，分 7 类（Prompt 工程 / MCP / Agent / 中文资源 / 博客等）
- 每条三要素：链接 + 量级 + "什么场景值得看"

### 🛡️ CI 自动守护 `.github/workflows/link-check.yml`

- lychee 扫 Markdown 内外链接
- 双语对照一致性校验（每个 `xxx.md` 必须有 `xxx.en.md`）
- 每周一 cron 捕捉外链 rot

### 📝 贡献指南 重写 + 英文版

- 明确贡献类型按优先级排序（陷阱 > 技巧 > 场景 > 修正 > 翻译）
- 加 `CONTRIBUTING.en.md` 英文版

### 🔧 Fixes

- **agency-agents-zh 角色数 187 → 211**（12 处引用统一）
- **Aider 示例模型名** `claude-3-5-sonnet` → `claude-sonnet-4-5`（13 处）
- **Copilot 陷阱 7 措辞修正**（"静默降级" → "排队或拒绝"）
- **英文 README 锚点修复**（`#-9-tool-guides` → `#9-tool-guides`）
- **英文 README 新手推荐工具**：Trae → Copilot/Cursor Free（国际用户适配）
- **cheatsheet 删除无法确定的 Esc Esc 描述**

### 📊 变化统计

| 项目 | 更新前 | 更新后 |
| --- | --- | --- |
| Markdown 文件数 | ~42 | ~65 (+23) |
| 双语对照覆盖 | 部分 | **100%**（templates 除外） |
| CI workflow | 0 | 1（link + bilingual） |
| 陷阱合集 | 各 README 4–5 行速查 | 31 个深度展开 |
| 速查 / 决策页 | 无 | cheatsheet.md |
| 画像推荐 | 线性三步 | 6 种画像路径 |

---

### 2026-04 以前

参考 `git log` 查看历史。早期重点：

- 9 款工具独立 README
- 7 套通用方法论（prompting / debugging / testing / etc.）
- 3 个多工具协作工作流
- 中英双语初始化（大部分章节）
- 可复制的配置模板

# 关于本书

## 为什么有这本书

ai-coding-guide 仓库到 2026-04 已经积累了：

- 10 款主流 AI 编程工具的中文教程
- 7 套通用方法论（提示词、需求拆解、调试、测试、代码审查、上下文管理、安全）
- 多个端到端实战脚本 + 31 个深度踩坑合集
- 9 工具横向对比速查表

但仓库形态有个问题：**新读者打开 README 看到 50 个链接，不知道从哪读起。**

这本书就是把这堆"各自精彩但散乱"的内容，按读者画像重新编排，给一个**有顺序、有节奏、有起承转合**的阅读路径。

## 内容来源

每一章本质上都是仓库里某个文件的"重新装订"——内容 0 修改，只在卷首加导读、卷末加串联。

| 章节 | 上游文件 |
| --- | --- |
| 卷一 各工具速通 | `<tool>/README.md`（如 `claude-code/README.md`） |
| 卷二 方法论 | `common/<topic>.md`（如 `common/debugging.md`） |
| 卷二 踩坑合集 | `pitfalls/<tool>.md` |
| 卷二 协作章节 | `workflows/<combo>.md` |
| 别册速查表 | `cheatsheet.md` |

**好处**：仓库更新 → 书自动更新（用 mdbook 重新 build 即可）。这本书永远跟仓库同步，不会出现"书是 2026 年印的，工具早换代了"的尴尬。

## 怎么本地构建

```
# 装 mdbook（一次性）
brew install mdbook          # macOS
# 或 cargo install mdbook    # 其他平台

# 在仓库根目录
cd book
mdbook serve --open          # 实时预览（改 markdown 自动热更新）
mdbook build                 # 生成静态 HTML 到 book/book/
```

## 反馈渠道

- GitHub Issues：jnMetaCode/ai-coding-guide/issues
- 微信公众号：**AI不止语**（搜索 `AI_BuZhiYu`）
- QQ 群 / 微信群：见仓库 README

## License

本书使用 CC BY-NC-SA 4.0（署名-非商业-相同方式共享）。

- ✅ **个人免费**：阅读 / 学习 / 转载 / 节选 / 二次创作（注明来源即可）
- ❌ **商业使用需授权**：付费课程、企业内训（>10 人）、纸书出版、付费墙转售等
- 🔗 详见：`LICENSE`（英文全文）· `LICENSE.zh-CN.md`（中文友好版）

> ⚠️ 仓库**双 license**：`book/` 目录下的书用 CC BY-NC-SA 4.0，仓库其他部分（教程文 / 模板 / 脚本）用 Apache-2.0。

商业授权咨询：见仓库主 README 「**商业合作 · 内训咨询**」段落。