

从入门到搭工作流 →

Codex 实战入门

能交付任务的 AI 程序员 · 从 5 分钟跑通到团队复用

序幕

大趋势与定位

编程范式转变 · 工作台

二至四章

闭环 · 上下文 · 积木

MCP · Skills · Subagents · Hooks

第四章

团队复用

Plugin 打包 · 三 Agent 实战

学 Agent 开发 现在就该开始

Codex 从「能聊天」进化到「能交付任务」

01 / 趋势

Agent 开发正在快速升温，Codex 从「能聊天」走向「能交付任务」

02 / 目标

不是讲零散技巧，而是从头到尾跑通一套完整的 Codex 实战流程

03 / 受众

适合程序员和准备入门 Web Coding 的新人，从「会用」升级到「真用、用好」

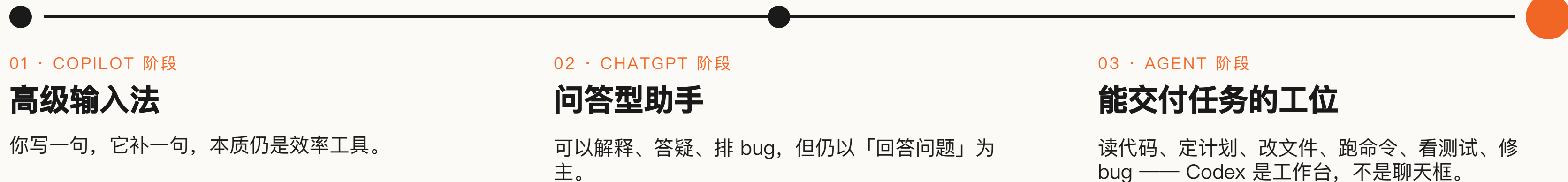
五段式学习路线

序幕 → 第一章 → 第二章 → 第三章 → 第四章

01 序幕 大趋势、Codex 定位、5 分钟跑起来	02 第一章 跑通第一个闭环，从需求到验证	03 第二章 上下文管理，决定 Agent 使用质量	04 第三章 MCP、Skills、Subagents、Hooks 四块积木	05 第四章 从个人提效到团队复用
---	--	---	---	--

整套视频的设计逻辑：先打地基，再做扩展，最后走团队分发

AI 编程的三阶段演进



一句话总结

Codex 的定位不再是聊天助手，而是「能交付任务的工位」

问路，和打车的区别

01 · 问答型

像「问路」

告诉你怎么走，但还得你自己走完

本质

解决「我不知道」

ChatGPT 阶段就是这个形态，能补、能答，但执行要你来

02 · 交付型

像「打车」

自己规划路线、执行过程、到站反馈结果

本质

解决「我做不完」

Code Agent 代表这种形态：读代码、定计划、改文件、跑测试一条龙

ONE-LINE DEFINITION →

Codex 是会动手的 AI 程序员

不是 ChatGPT 的换皮聊天框，而是真正接管完整任务的工程型 AI

01

读项目

理解项目结构、技术栈、测试体系，调用 CodeGraph 等工具建立结构认知

02

改文件

真改代码、改配置、应用最小改动原则，避免无关重构

03

跑命令

在项目里直接跑测试、构建、lint 等命令，并能根据输出继续分析

04

记规则

读取项目级 AGENTS.md，有多步规划、子任务并行、外部工具调用能力

PERCEIVE · ACT · CLOSE-LOOP →

感知 · 行动 · 闭环

后续理解 Codex 任何功能，都可以放到这三个标志里去看

01

感知

Perceive

理解项目结构、技术栈、测试体系 —— 知道自己在改什么

02

行动

Act

真改文件、跑命令、看结果，而不是只输出代码片段

03

闭环

Close-loop

改完会验证、失败会分析，最后交付有证据的结果

DESKTOP · CLI · IDE →

同一套 Codex, 三种进入方式

01

桌面端

DESKTOP

适合写作、PPT、图片、视频等非纯代码任务。提供图形化界面、文件预览和项目管理视图，是新手最容易上手的入口。

02

CLI

COMMAND LINE

最适合程序员，围绕项目目录工作，上下文最干净。开发场景最推荐这一入口，权限、配置、Session 都能精细控制。

03

IDE 插件

IDE PLUGIN

适合已经在编辑器里写代码、想随时调 Codex 协作的场景。和编辑器的光标、文件树、Diff 视图深度整合。

三个入口本质上是同一套 Codex 能力在不同工作平台中的呈现

PICK BY SCENARIO →

按场景选入口， 而不是最专业

新手不用一开始就纠结「哪个最专业」，先用熟一个入口更重要

场景 · SCENE	推荐入口 · ENTRY	原因 · WHY
日常写代码	IDE 插件	已经在编辑器里，随时协作成本最低
长任务 / 终端操作	CLI	上下文最干净，可脚本化、Session 可恢复
内容创作	桌面端	图形化界面，对图片、PPT、写作更友好

三个入口账号与项目大多互通，不是三套割裂系统 —— 关键是按场景选入口，而不是追求形式上的「最专业」。

DOWNLOAD · INSTALL · LOGIN →

安装不难， 难的是跨过门槛

01

下 载

去 OpenAI 官网

直接到官网下载 Codex 桌面客户端，按系统选择对应版本。

02

安 装

按系统装客户端

双击安装包，按系统提示完成安装。安装本身不复杂。

03

登 录

OpenAI 账号登录

安装完成后用 OpenAI 账号登录，跑通安装是后续所有 Agent 实战的起点。

VERIFY · INSTALL · LOGIN →

CLI 跑通， 开发就掌握了一半

VERIFY COMMANDS

先 ``codex --version`` 验证是否可用

```
$ codex --version
```

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

命令不存在时，先确认 Node/npm，再全局安装。
登录过桌面端后，CLI 一般无需重复登录。

关于模型

官方为主，
国产为辅

国产模型可作为补充和备用，但长任务规划、复杂工具调用仍建议以官方模型为主。

FROM CHAT TO PROJECT →

写代码，要用 「项目」级工作方式

PATH A

桌面端

推荐「使用现有文件夹」

避免新建空白项目带来路径混乱。已存在的代码目录更接近真实工程场景，Codex 也能更快理解上下文。

PATH B

CLI

更直接，进入目录即工作

```
$ cd your-project && codex
```

进入目标目录后运行 codex 即可，是开发场景最推荐的方式。

最小路径：装桌面端 → 验 CLI → 建项目

NEW HIRE ONBOARDING →

把 Codex 当新同事 带入项目

× 不介绍，直接甩需求

瞎改一通

没解释项目就直接丢任务，结果是反复试错、改了又改、来回沟通成本居高不下。

✓ 先读懂，再开干

结构感知稳

老项目第一步不是直接写，而是先证明 AI 读懂了项目。多花几分钟换后面更稳的回报。

给 Codex 一张项目地图

01 · 更快

回答更快

通过符号与调用关系图，省去逐文件扫描

02 · 更清

结构更清晰

帮助 Codex 快速建立结构认知，准确定位修改点

03 · 更省

不耗 token

提供离线化结构数据，不额外消耗对话上下文

TWO COMMANDS

安 装

```
$ npm i -g codegraph
```

初 始 化

```
$ codegraph init -i
```

CodeGraph 会把项目解析成符号和调用关系，是 Codex 在老项目里建立认知的加速器。

INIT → LIST → WRITE →

先读懂项目 再开始干活

01

初始化图谱

```
$ codegraph init -i
```

在项目目录中执行，把代码解析成符号和调用关系。

02

列出项目结构

Codex 输出树状结构

让 Codex 以树状形式列出项目结构，确认它已经「看见」项目。

03

写入结构文档

```
project_tree.md
```

把树状结构写入 project_tree.md，留下项目快照。

序幕落点

演示重点不在文件本身，而在「先读懂项目再开始干活」的流程。

反应式用法，用得越久越乱

典型误区是「问一句改一句」，把 Codex 用成反应式工具

01 · 反应

问一句改一句

02 · 盲改

一报错就追问

03 · 失控

来回乱改

04 · 结果

人和项目一起失控

FAILURE LOOP · 失败循环

现象

生成了很多文件，却不知道改了什么、为什么改、是否真能跑。

问题

问题不在 Codex 不行，而在整个开发流程没有被建立起来。

像给真人工程师 交代任务一样协作

01

读项目

看清楚上下文

02

做判断

决定要不要接

03

讲计划

先同步再动手

04

再动手

改得越少越可控

05

自测 + 汇报

有证据的交付

TAKEAWAY

从问答机器 切换到开发对话

Codex 完全能走完整流程，前提是你不要只催它「快给我代码」。

用 Todo List 跑通完整闭环

TODO LIST · 演示样例

☐ 完成首页布局

☐ 添加任务 CRUD

☐ 增加本地存储

☒ 完成单元测试

简单 · 清晰 · 好验证 —— 是新手最合适的入门项目。

为什么用 Todo List

把注意力放在流程

选择低门槛样例，能让注意力落在「完整开发对话」上，而不是耗在项目复杂度里。

在 ChatGPT Work 中新建项目文件夹即可开始。重点不是项目多复杂，而是通过一个真实任务跑通完整闭环。

先让 Codex 证明它看见项目

01

完整开发对话的第一步

永远是读代码

新项目

可以省略

新建项目没历史代码可读，可以短暂跳过。

老项目

不能跳过

老项目里这一步是底线，跳过就是「盲改」的源头。

常用 Prompt

「先看项目结构和现有代码，
再说明你对项目的理解」

先停下来对方向， 再做才不返工

02

自动化越强，越要在关键节点停一下

先要计划，不要直接开发

01 · 需求

接到任务

把需求讲清楚，包括边界、目标、约束。

02 · 计划

给方案

要求 Codex 先输出计划，并强调「不要直接开发」。

03 · 校正

对方向

避免做完一堆再回滚，让计划成为闭环第一道门。

然后才执行

计划不是形式主义，
是质量底线。

改得越少 越可控

03

计划确认无误后再让 Codex 改

简单 · 最小改动 · 不夹带重构

保持简洁

不堆复杂度

一次只解决当前问题，不发明新抽象。

最小改动

控制 diff 范围

范围越小越好 Review，越容易回滚。

不做无关重构

顺手改太多 = 引入 bug
也增加 review 成本

不只改代码 还要把测试跑起来

04

Codex 关键优势

在项目里直接执行命令

常用指令

```
$ npm test
```

```
$ npm run build
```

```
$ npx eslint ./src
```

请运行项目测试命令，确认本次修改未破坏现有功能

没有测试时

退一步跑构建

没有测试至少跑构建或 lint，先排除明显错误，再进入「改完自测」流程。

AI 说改完了 ≠ 结果可交付

05

正确的循环

观察 → 分析 → 修复

出问题怎么办

先描述现象

出现白屏、按钮无响应等现象时，先描述现象，让 Codex 分析原因，再进入修复。

反模式

瞎改就继续错

错就乱改、反复追问，是新手最大的陷阱。这五步循环是新手最该先练熟的基本功。

SAFE · DELEGATE · FULL →

理解权限模型， 按任务风险选档

L1 · 请求批准

更安全

关键动作会询问

每改一个文件、跑一个命令，都要先询问 —— 新手
第一阶段最推荐。

L2 · 替我审批

更高效

项目内自推进

熟悉项目内可自推进，涉及外部文件仍受限。可理
解为「项目房间内自由工作」。

L3 · 完全访问

风险最大

整机通行证

适合临时测试目录和大范围自动化。不适合重要仓
库、生产配置、带密钥的项目目录。

三档模式代表从「更安全」到「更高效」的递进
关键不是永远给最大权限，而是按任务风险做选择

L1 · 每一步都敲门

新手最该从这一档开始

适合场景

新手 · 陌生项目
公司项目 · 生产环境

关键动作先询问：

- 改文件之前询问
- 执行命令之前询问
- 访问敏感位置之前询问

优点是安全、可见、可控，便于理解 Codex 想做什么。

核心动作

改前问 · 跑前问

把每一次 Codex 要提交的动作都变成你和它的一次确认。

先看见，再下一步。

L2 · 项目内自推进

像在熟悉的工位自由流转

边界 · 项目房间内

改文件 → 自推进	跑命令 → 自推进	外部文件 → 受限
-----------	-----------	-----------

更适合已经熟悉、也比较信任的日常项目。

项目内常规动作可由 Codex 自推进，无需每一步都人工确认。涉及外部文件或更高风险操作时，通常仍会受限或需要额外确认。

可以理解为「项目房间内自由工作」。

权限最大 风险也最大

适合场景

临时测试目录 高信任任务 · 批量整理

不适合

- × 重要仓库
- × 生产配置
- × 带密钥的项目目录

记忆口诀：权限越高，越要确认任务和环境可信。

RISK · 风险提示

整机 通行证

适合大范围自动化操作；效率与风险同向放大，每次都应检查任务边界。

写完只是第一步， 交付还要善后

01

查看 Diff

确认改动范围是否合理，有没有夹带无关重构。

新手最容易在这里偷懒，老手最能在这一拉开差距。

02

跑测试

有测试就跑测试，没有就退一步跑构建或 lint。

让 Codex 基于输出继续分析，而不是改完就交付。

03

总结改动

最终交付需要一份清晰的改动说明。

这类要求非常适合沉淀进 AGENTS.md，让规则发挥长期价值。

重点不是逐行找茬 而是范围是否合理

GIT DIFF · `diff --git`

@@ utils/format.ts @@

- export fn fmt(input)

+ export fn fmt(input, opts)

@@ tests/format.test.ts @@

+ test('fmt handles opts correctly')

改动只发生在 `fmt` 函数和对应测试 —— 改得干净。

没有任何无关重构。

三个检查点

范围合理?

只解决当前问题?

是否夹带无关重构?

可用 `git diff`，也可让 Codex 总结本次修改。

一旦发现无关改动，立即要求只保留与任务相关的部分。

失败不可怕 可怕的是不分析

Agent 的闭环

01

修改

02

运行

03

观察

04

没有测试就跑构建或 lint。

推荐 prompt: 先告诉我问题在哪里, 不要急着继续修改。

测试失败时

先解释 · 再动手

- › 第一轮失败
- › Codex 给出失败原因
- › 第二轮精准修复
- › 通过后汇报

Agent 的真正优势
是参与闭环, 而不是一步到位。

改动总结

四象限要写齐

A · 修改文件

列出本次涉及的所有文件路径
哪些新增 / 修改 / 删除，一目了然

B · 修改原因

解释为什么这么改
和原始需求一一对应，避免理由模糊

C · 验证命令

说明跑了哪些命令做验证
测试、构建、lint，留下可复现痕迹

D · 验证结果

写明测试通过 / 失败及处理
没有经过 diff 和测试验证的代码，不算交付

CHAPTER 02 · CONTEXT →

上下文

Agent 质量的分水岭

不会管理上下文的人更乱、更慢；会管理上下文的人更稳、更省 token

01 · 本质

工作记忆预算

02 · 清理

什么时候清空

03 · Resume

恢复旧工程任务

04 · 压缩

主动阶段总结

05 · 图片

让视觉成为上下文

06 · AGENTS.md

项目级长期规则

上下文 ≠ 聊天记录 而是工作记忆预算

更准确的理解

工作记忆预算

上下文不只是聊天记录，而是 Codex 当前知道的全部有效信息。

预算有限，所以不是把所有信息都塞进去，而是保留最有用的信息。

管理上下文的核心是主动取舍，而不是被动堆积。

CONTEXT BUDGET

200K

工作记忆桶 · 容量有限

KEEP WHEN? →

同任务连续推进 上下文就该保留

场景 01

同一功能

持续完善同一个功能时，前面的决策和实现就是下一步的基础。

场景 02

同一 Bug

排查同一个 bug 时，前面的尝试、判断、错误信息都不要丢。

场景 03

同一项目分析

常见续接方式：「继续当前任务，沿用前面的分析和计划」。

任务换了 就该清空重来

清 理 上 下 文 · TRIGGER POINTS

01

任务完全切换

02

失败尝试堆积

03

被旧思路带偏

如果 Codex 被旧思路带偏，反复修改仍无效，就该清空重来。

CLI 用 `/clear`，桌面端直接开新对话。

ONE RULE

主动取舍
胜过
被动堆积

真实任务常常做不完 Resume 让会话接着工程走

RESUME COMMANDS

\$ codex resume

→ 显示历史会话列表，按需选择恢复哪一次

\$ codex resume --last

→ 直接恢复最近一次会话

保留的不只是聊天内容，更是计划、判断和任务进度 —— Resume 让 Codex 更像长期合作伙伴。

长期合作伙伴

不是一次性助手

恢复后先盘点 再决定下一步

回顾进度	检查 git diff 与当前项目状态	再决定下一步
------	---------------------	--------

推荐 PROMPT

先回顾当前进度，
再告诉我下一步准备做什么。

关 键

不是打开旧聊天
而是恢复旧工程任务

长对话会被自动压缩 但不能只被动等着

RAW · 原始长对话

U1: 解释项目结构...

U2: 改 file A 第 12 行...

U3: 跑测试失败...

U4: 重试 fix A...

U5: Ctrl+L 输出 1...

U6: 查看 diff...

U7: 增加单元测试...

+ 240 条历史消息

SUMMARY · 自动摘要

完成：A 模块

测试：

决策：最小改动

待办：补充测试用例

重点保留，但不可能 100% 留住细节

TAKEAWAY

主动组织阶段性总结
让关键信息
提前上浮

每完成一阶段 给上下文做一次快照

STAGE-END PROMPT

- # 目标
- # 已完成内容
- # 关键决策
- # 剩余问题

总结任务目标、已完成内容、关键决策、剩余问题 —— 主动总结让重要信息更容易在压缩时被保留。

为什么主动

主动管理
长期更稳

主动管理 vs 被动堆积

把「你看到的问题」 直接变成上下文

01 · 报错截图

ERROR

让 Codex 识别错误信息，结合代码分析原因 —— 终端报错、堆栈、异常。视觉直接把错误变成上下文。

02 · UI 截图

LAYOUT

定位移动端溢出、布局错位、文字重叠等问题，比用文字描述快得多。

03 · 设计稿

DESIGN

对比实现与设计差异，先给改造建议 —— 让 Codex 像 design partner 一样协作。

图片提示的 四要素卡片

01

这是一张什么图

WHAT KIND

报错、UI 还是设计稿 —— 先给 Codex 命名上下文

02

哪里不对

WHERE

指出问题发生的位置 —— 区域、引用、坐标或关键字段

03

希望你做什么

ACTION

分析 / 修复 / 对比 —— 明确 Codex 的目标动作

04

是否允许改文件

SCOPE

先看还是直接动手 —— 主动设定边界，避免越权

给 AI 写的 一份项目说明书

DEFINITION

AGENTS.md

一次性 prompt 只能提醒一次，AGENTS.md 是项目级长期规则。Codex 进入项目时会读取，并在后续任务中持续参考。

适合沉淀什么

工作方式

经常重复提醒的协作模式

代码规范

命名、结构、不做无关重构

命令约束

改前说计划、改后跑测试

AGENTS.md 三层结构

01

项目命令

COMMANDS

npm run dev
npm test
npm run build

02

代码规范

CONVENTIONS

优先 TypeScript
保持现有组件结构
不做无关重构

03

工作规则

RULES

改前说计划
改后跑验证
测试失败先分析

先写最关键的几条 边用边补

AGENTS.md · EXAMPLE

AGENTS.md

项目命令

- 启动: npm run dev
- 测试: npm test
- 构建: npm run build

工作规则

1. 修改前先说明计划, 最小改动
2. 修改后跑测试或构建
3. 总结修改 + 验证结果

代码规范

- 优先 TypeScript · 复用已有 class

补充时机

发现反复犯 就把规则补进来

长期项目建议都准备一份最小版本, 后续边用边补。一次写全反而容易失去迭代感。

CHAPTER 03 · FOUR BUILDING BLOCKS →

四块积木

才叫「搭 Agent」

这一章不看「怎么用」，而看「怎么搭」

MCP

连接外部工具

Skills

沉淀可复用流程

Subagents

并行多角度处理

Hooks

自动执行规则

大脑 · 手脚 · 记忆 · 流程

01 · 模型大脑

推理 · 规划

决定代码能力和判断质量

02 · 工具手脚

连接 · 执行

连接项目外部能力的边界

03 · 记忆

上下文 · 记忆

上下文、AGENTS.md、Skills

04 · 流程

验证 · 汇报

怎么接任务 / 怎么验证 / 怎么汇报

要素到能力的映射图



MODEL CONTEXT PROTOCOL →

Codex 连接外部世界的标准接口

设计稿 / Figma

直接看懂设计稿 → 改实现

文档 / GitHub

读 Issues / PR / 文档

浏览器

看 DOM / 截图 / 抓错误

MCP

统一协议

需求文档

从 PRD 自动拆任务

数据库 / API

直查 schema / response

没 MCP 时

只能手动复制粘贴

价值大 每接一个多一层风险

价 值 · VALUE

感知外部

把 Codex 从「本地代码助手」扩展成「外部环境可感知」的开发智能体。

权 限 · PERMISSIONS

多一层

每接一个工具，等于增加一层权限和风险边界。新手应优先从低风险、只读类工具开始。

边 界 · SCOPE

AGENTS.md ↔ MCP

AGENTS.md = 理解项目规则

MCP = 连接项目之外的世界

两者搭配，Agent 才像项目成员。

从读文档到改代码 连接成一个闭环

01

检查安装

`/mcp`

在项目里用 `/mcp` 检查 Github MCP 是否已安装

02

接入服务

`codex mcp add`

```
codex mcp add github --url https://api.githubcopilot.com/mcp/ --bearer-token-  
env-var GITHUB_PAT_TOKEN
```

03

用起来

明确要求 Codex 使用 Github MCP 读取文档，并先给开发计划

闭环

看文档 → 读项目 → 给建议 → 改代码 → 跑验证

把高频 prompt 变成反复可调的能力

SKILL STRUCTURE

FOLDER
daily-report/

ENTRY FILE
SKILL.md

一个 Skill 往往就是一个文件夹加一个 SKILL.md，把 prompt 沉淀成可复用资产。

常见用途：日报、代码审查、发布说明、安全检查 —— 凡是反复要写第三次的事情，就该封装。

与 AGENTS.md 的差异

规则 vs 流程

AGENTS.md 适合存规则
Skills 适合存一套可重复执行的流程。

规则是约束，流程是可调用能力。两者补位，让 Agent 更像团队成员。

把代码审查流程 沉淀进 SKILL.md

4 个检查项

BUG

逻辑边界 · 空值处理

TEST

测试缺口 · 关键路径

SECURITY

权限校验 · 敏感数据

MAINTAINABILITY

结构 · 重复 · 命名

一句话调用

```
$ codex run skill
```

`code-review`

以后只需一句：
「请使用 `code-review skill`
审查当前改动」

从文字 Skill 走到插件版



该封装的信号

同一套 prompt 已写到第三次
就值得封装

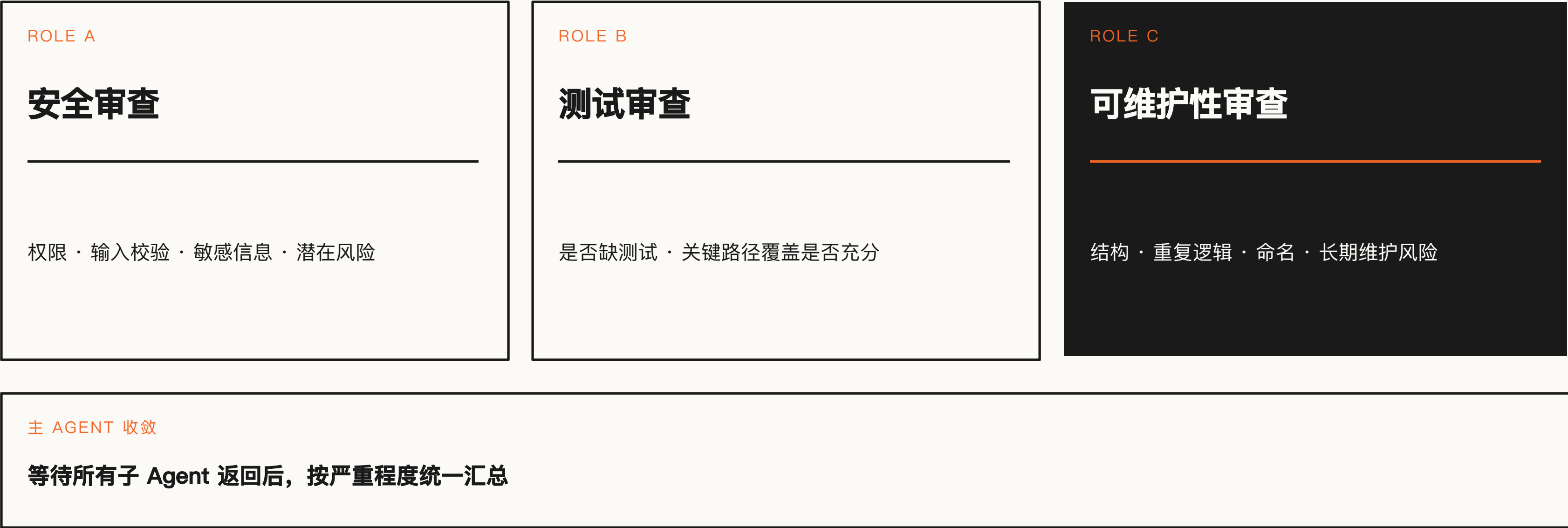
清晰分工，再汇总

主 Agent + 多子 Agent



核心价值不是「更炫」，而是「更清晰地分工再汇总」

三个角色 并行审查



大胆探索 谨慎写文件

✓ DO · 适合多 Agent

读多 · 查多 · 分析多

大量读取、查询、分析的任务最受益，例如陌生项目理解。

× DON'T · 不要多 Agent

不建议同时改代码

多个 Agent 同时写文件容易相互冲突。写文件由主 Agent 收敛。

一句话原则

**把复杂任务拆成多个清晰方向
最后收敛出高质量结论**

把「口头提醒」 升级成「自动执行」

CODEX 生命周期 → HOOK 插入点

01 · Prompt

提交 prompt 时

02 · PreCmd

命令执行前

03 · PostCmd

命令执行后

04 · Stop

任务结束前

05 · Compress

上下文压缩前

常见用途

命令检查 · 日志记录 · 安全拦截 · 自动提醒
在 Codex 看不到的时刻，Hook 帮你出手

真正阻断 不只提醒

BLOCKED PATTERNS

```
$ rm -rf /
```

```
$ git reset --hard
```

```
$ cat .env
```

触发即返回非 0 退出码，Codex 在下一轮被告知命令被拦截。

HOOK 文件结构

01 · 配置文件

.codex/hooks/hooks.json

02 · 拦截脚本

pre_tool_use.py

03 · 触发逻辑

命令执行前检查

命中模式 → 返回非 0

入门先放下 Hooks

HOOKS 的门槛

需要写脚本

Hooks 对新手有一定门槛。

可以先让 Codex 帮你生成 Hook 所需的 Python 代码，再做最小化配置。脚本语言的掌握是另一个前提。

入门顺序

先掌握 CLI / AGENTS

1 → CLI

2 → AGENTS.md

3 → MCP

4 → Skills

5 → Hooks

只有当某些规则必须自动执行时
再考虑引入 Hooks

CHAPTER 04 · TEAM REUSE →

团队武器

从单人秘籍到团队复用

一个人效率提升有上限，团队复用则能把效率乘上去

01 · 两条路

设计逻辑

02 · 三原则

设计原则

03 · Plugin

Plugin 打包

04 · 实战三连

评审 · 拆任务 · 文档

我自己用得爽 vs 别人也能用得稳

个人提效

我自己用得爽

允许个性化 prompt 和私有习惯

命名、表达、风格都可以自定义 —— 优化对象只有一个人。

团队复用

别人也能用得稳

要重新审视命名、输入、说明和边界

个人秘籍原封不动给团队，往往水土不服。

输入 · 输出 · 边界

团队复用的基本标准

01 · 输入

有边界

知道需要什么
知道不需要什么
明确输入格式

02 · 输出

稳定

结构固定
内容范围固定
团队成员拿到结果都能直接读

03 · 边界

清晰

一个 workflow 只做一件事
便于组合和替换

一组能力的可安装包

PLUGIN = ?

Skills

+ MCP + Hooks + Docs

一个 Skill 是一个能力
一个 Plugin 是一组能力的安装包

解决什么

分散配置 手工复制

Plugin 的真正价值是：

- 可安装
- 可分发
- 可标准化

让团队复用不再依赖个人经验。

最小可用 Plugin 从 code-review 起步

PLUGIN NAME

team-code-review

PLUGIN STRUCTURE

plugin.json

minimum viable

+ skills/code-review/SKILL.md

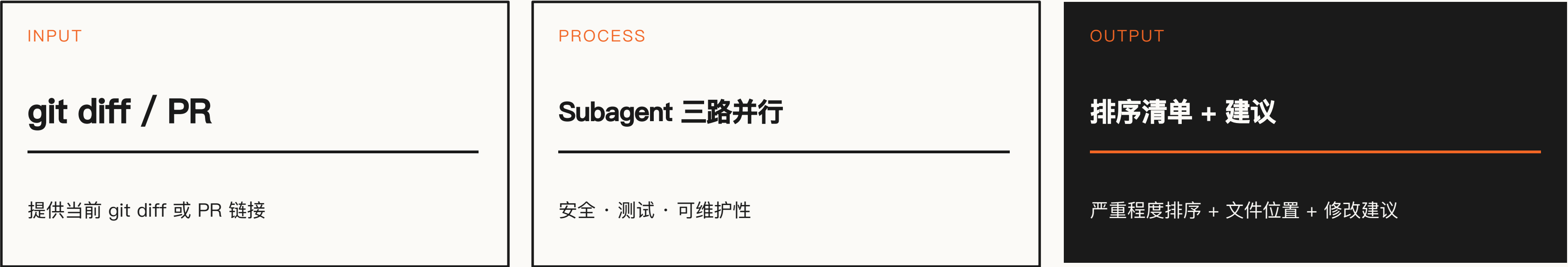
之后再加

GitHub MCP + Hook · 模板

随着经验增加，再继续添加 GitHub MCP、Hook、模板，以及更多团队流程。

先把单个 code-review Skill 打包进去，快速建立最小可用插件。

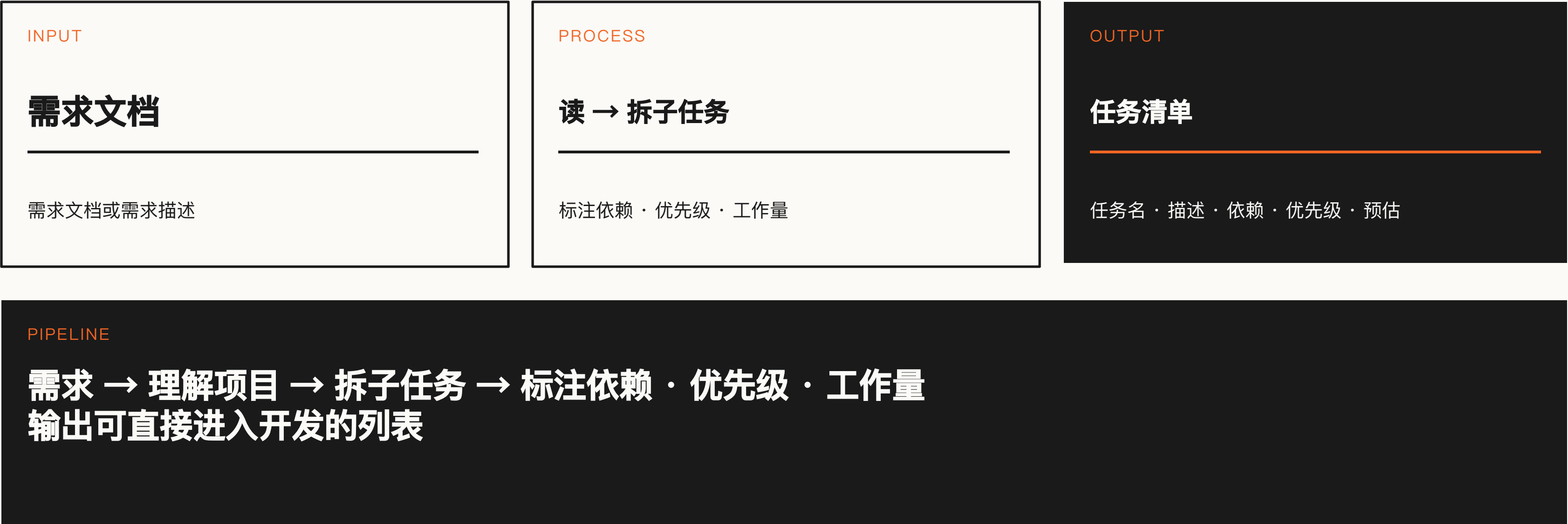
只评审，不改代码



边界

只评审，不改代码

只拆任务，不写代码

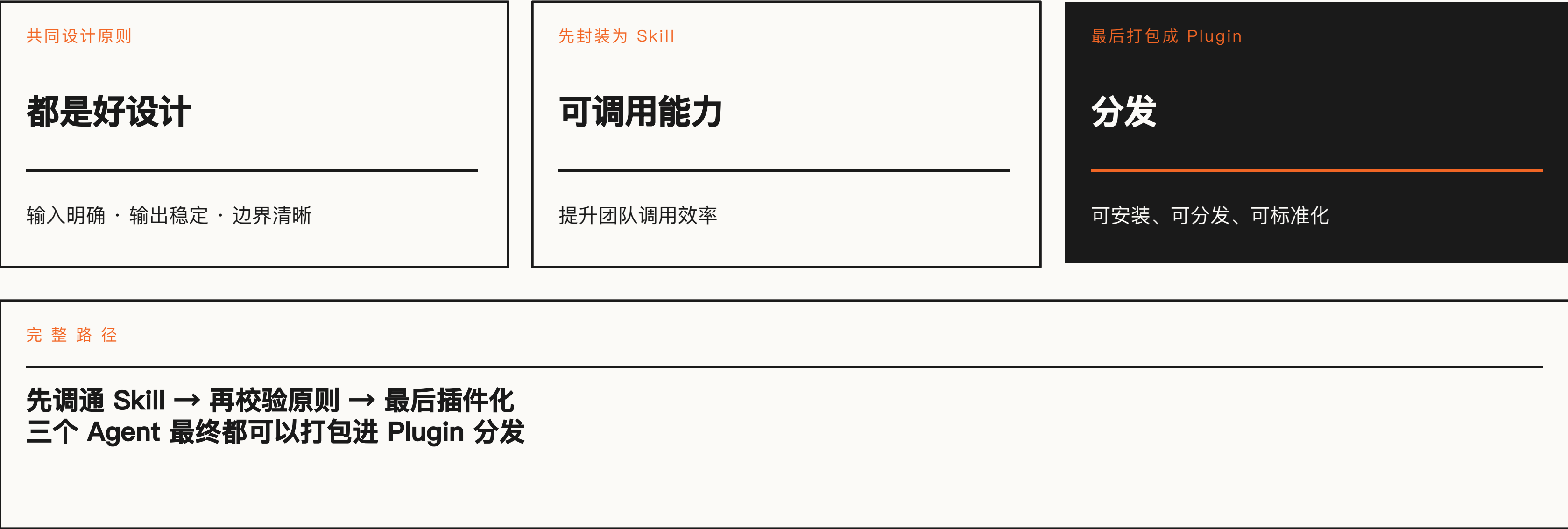


只动文档 不动代码



Skill → 原则校验 → Plugin

完整路径



一张图回顾整套路径

为什么学 → 怎么装 → 怎么读项目 → 怎么跑闭环 → 怎么管上下文 → 怎么搭积木 → 怎么做团队复用

KEY TAKEAWAY

Codex 不只是代码助手，而是一套可逐步搭建起来的 AI 开发 workflow

新手推荐路径

先打地基 · 再做扩展 · 最后走团队分发

ACTION → NOW →

看完就跑一遍

看完就亲手打开 Codex 跑一遍

后续专题

Skill · Subagents · Hooks

更深入专题展开

关键不是

会不会用

是什么时候用熟

行动建议

打开 Codex 跑一遍

早一年 / 晚一年