Vibe Coding 实战教学指南

Contents

Vibe Coding 实战教学指南

> Vibe Coding 不是”让 AI 帮你写代码”那么简单——它是一种把 AI Agent 当作团队成员协作开发的工作方式。本指南覆盖从冷启动到长期维护的完整流程。

一、什么是 Vibe Coding?

Vibe Coding 这个词源自 Andrej Karpathy 在 2025 年初的一条推文, 本意是”完全凭感觉, 接受 AI 写的所有代码, 出错就把错误粘回去”。但在工程实践中, 它逐渐演化成了一种更严肃的范式:

核心理念: 你不是在”写代码”, 而是在**指挥一个或多个 AI Agent 写代码**。你的核心工作变成四件具体的事:

1. 把意图描述清楚 (spec)

不是”做一个登录功能”, 而是”基于现有的 `auth/` 模块, 加邮箱 +OTP 登录,OTP 用 Redis 存,5 分钟过期”。

2. 给 Agent 提供正确的上下文 (context)

Agent 不知道你的项目长什么样、约定是什么、之前走过什么弯路。你的工作是把这些信息**结构化地**喂给它——通过 AGENTS.md、spec、文件路径、ground truth 数据。

3. 审查、引导、修正——这是你 80% 的工作

这三个动词每天都在做, 具体落地是这样:

审查 = 你不写代码, 但你必须看 Agent 写了什么

每次 Agent 改完代码, 做这三件事:

- **看 diff**: 让它列出”我改了哪些文件、每个文件改了什么”, 或者直接 `git diff` 看
- **跑验证**: 跑测试、起服务、看日志, **亲眼看到 work, 而不是听 Agent 说”应该 work”**
- **追问设计**:“为什么这么改? 有没有更简单的方案? 这个新依赖必要吗?”——不质疑就会被它”看起来很合理”的代码淹没

引导 = 在 Agent 跑偏之前先说清楚

最重要的话术不是”帮我做 X”, 而是:

```
我想做 X。在你动手之前，先告诉我：
- 你打算怎么做（分几步、改哪些文件）
- 你需要我决定什么（选 A 还是 B 方案）
- 你看到了什么我没看到的问题或边界 case
```

让 Agent 先说计划再动手。计划错了你能立刻拦，省下半小时乱改的时间。

引导还包括"约束"——告诉 Agent 不要做什么："不要引入新依赖""不要改 auth/ 下的文件""保持函数签名兼容"。这些约束应该说在前面，不要等它写完才说。

修正 = 看到不对就立刻喊停

最关键的一句话：**"等等，这不对，我们重来。"**

新手最常犯的错：看到 Agent 走错方向了，但想着"再让它试试看说不定能拐回来"，于是又过了 5 轮对话，代码越搞越乱、上下文越占越满。

正解：**一旦感觉方向错了，立刻打断，git reset，重新讲清楚再来**。继续让一个走错方向的 session 跑下去，只会浪费上下文、产生需要回滚的代码、强化错误的心智模型。

4. 在合适的时机分解任务、压缩上下文、切换 Agent

这是 Vibe Coding 最反直觉的一点——**会"换"和会"停"和会"做"一样重要**。具体怎么判断时机，见第五章。

一个比喻：从"管字符"到"管 Agent 的注意力"

传统编程里，你直接打字，每个分号、每个变量名都是你的手在控制。**你的工作粒度是字符**。

Vibe Coding 里，字符是 Agent 打的。但 Agent 的"注意力"——也就是它的上下文窗口——是有限的资源。它一次能"想着"的东西就那么多：

- 它现在装着哪些文件的内容
- 它记得我们 20 轮对话前讨论过什么
- 它在意我刚强调的那个约束
- 它意识到这个改动会影响别处

这些东西都在抢同一份"注意力预算"。你的工作变成：

- **决定让 Agent 现在关注什么** (给它合适的 spec、合适的文件、合适的任务粒度)
- **避免它的注意力被无关的东西占满** (对话太长、读了太多无关文件、目标太散)
- **在它分心走偏的时候拉回来** (打断、重启、refocus)
- **在自然的停顿点把注意力清零** (完成一个阶段就开新 session)

所以"管字符 → 管 Agent 的注意力"——不是玄学，就是字面意思。你以前在乎每个分号，现在你在乎"Agent 现在脑子里装着什么、还能装多少、装的对不对、什么时候该清零"。

写代码的"手感"还在，但操作对象从字符变成了 Agent 的注意力。

二、Spec 是什么？为什么它是一切的起点？

Spec(规范)= 你想让 AI 做什么的清晰描述。它可以是一段话、一份 Markdown 文档、或者一个 Issue。

好 Spec 的三个特征

1. **意图清晰**: 不是”做一个登录功能”, 而是”基于现有的 `auth/` 模块, 添加邮箱 +OTP 登录,OTP 用 Redis 存储,5 分钟过期”
2. **约束明确**: 风格、依赖、性能、错误处理边界都说清
3. **验收标准可测**:“成功的标志是什么”——能跑通哪个测试? 哪个 endpoint 返回什么?

Spec 的两种模式

轻量 Spec(对话式): 写在聊天里, 适合小改动

```
在 UserService 加一个 deleteAccount 方法:
- 软删除(设 deleted_at)而不是物理删除
- 同时撤销该用户所有的 session
- 写一个单元测试覆盖正常流程和"用户不存在"的情况
```

重量 Spec(文档式):`specs/feature-xxx.md`, 适合多 session 跨天的功能

```
# 用户注销功能

## 背景
当前系统只有禁用账户, 没有真正的注销流程。GDPR 合规需要...

## 目标
- 用户可以发起注销
- 30 天冷静期内可撤销
- 30 天后自动执行物理删除

## 非目标
- 不处理已发布内容的归属转移 (下一期)

## 技术方案
[Agent 在这里填, 你审]

## 验收
- [ ] POST /account/deletion 创建注销请求
- [ ] 测试覆盖率 > 80%
- [ ] 撤销 endpoint 可用
```

> **关键**:Spec 不是写完就完了。开发过程中 Spec 要随着发现的问题更新, 它是 Agent 和你之间的”合约文档”。

三、AGENTS.md / CLAUDE.md 里存什么?

这是项目级的”Agent 操作手册”, 放在仓库根目录。Agent 每次启动会自动读取它。

应该放进去的内容

1. 项目身份

```
## 项目简介
这是一个 B2B SaaS 的后端,Go 1.22 + PostgreSQL + Redis,
部署在 AWS ECS Fargate, 前端在另一个仓库。
```

2. 目录地图 (最重要, 救命用)

```
## 代码结构
- `cmd/api/` - HTTP 入口
- `internal/auth/` - 认证, 改这里要看 SECURITY.md
- `internal/billing/` - 计费, 所有金额用 decimal, 不要用 float
- `pkg/` - 可被外部引用的工具, 慎改
- `migrations/` - 数据库迁移, 只增不改, 用 goose
```

3. 代码风格与约定

```
## 编码规范
- 错误处理: 用 `fmt.Errorf("doing X: %w", err)` 包装
- 日志: 用 slog, 不要用 fmt.Println
- 测试: 表驱动, 文件名 `_test.go`
- 不写注释除非是 exported API
```

4. 命令清单 (让 Agent 不用猜)

```
## 常用命令
- 运行测试: `make test`
- 启动本地: `make dev`(会起 docker compose)
- 跑迁移: `make migrate-up`
- Lint: `make lint`(必须通过才能提 PR)
```

5. 红线

```
## 禁止事项
- 不要直接改 `migrations/` 下已存在的文件
- 不要在生产代码里加 TODO, 要么做要么开 issue
- 不要 commit 前不跑测试
- 涉及 PII 的字段必须走 `internal/crypto/pii.go`
```

6. 历史教训

```
## 容易踩的坑
- Postgres timezone 默认是 UTC, 但 API 返回要用用户时区 (看 utils/tz.go)
- Redis key 必须带 namespace 前缀, 见 internal/cache/keys.go
```

不应该放进去的

- 整个项目的详细架构 (放 `docs/architecture.md`, 在 AGENTS.md 里指过去)
- 业务逻辑细节 (那是 spec 的事)
- 个人偏好不相关的部分 (放个人的全局 config)

经验:AGENTS.md 控制在 200-400 行最佳。太短没信息, 太长 Agent 抓不到重点。每隔几周回顾一次, 把"上次 Agent 又踩这个坑"的内容补进去。

四、冷启动: 接手一个新项目怎么办?

冷启动有两种场景, 处理方式不同。

场景 A: 全新项目, 从零开始

第 1 步: 先写 Spec, 不要急着让 Agent 写代码

```
我: 我要做一个 X, 大概这样这样... 帮我先写一个 spec,
    不要写代码。问我所有不清楚的问题。
```

让 Agent 反问你 5-10 个问题, 这些问题本身就是冷启动最大的价值——它逼你想清楚边界。

第 2 步: 让 Agent 设计目录结构和技术选型

```
我: 基于这份 spec, 提出 3 套技术方案, 各自的取舍是什么?
    先不要建文件。
```

第 3 步: 确定方案后, 初始化项目骨架 + AGENTS.md

骨架建好之后, 可以跑 `/init` 让 Claude Code 生成 AGENTS.md 的初版, 然后你**手动补**那些 `/init` 抓不到的内容: 产品目标、设计理念、未来的扩展方向、团队约定的红线。

第一版 AGENTS.md 很重要, 后续所有 session 都靠它。**初版不要追求完美, 先保证有, 后面边用边迭代**。

第 4 步: 小步建设, 每个 session 完成一个明确单元, 提交一次。

场景 B: 中途接手他人的项目 (更常见也更难)

这是冷启动里最难的情况。**关键是不要让 Agent 上来就改代码**。

第 1 步: 用 `/init` 生成 AGENTS.md 的初版

Claude Code 内置了 `/init` 命令 (Codex 等工具有类似功能), 专门用来扫描仓库自动生成 CLAUDE.md/AGENTS.md:

```
> /init
```

它会做这些事: - 扫 README、package.json/go.mod/Cargo.toml 等元数据文件 - 识别框架、构建工具、测试命令 - 推断目录结构和模块职责 - 生成一份初版 AGENTS.md(或 CLAUDE.md)

这是接手项目最快的起点, 几秒钟就能拿到一份草稿, 远比从空白开始强。

但是——**`/init` 生成的版本只是起点, 不是终点**。它的局限:

- 只能从文件结构和明文配置推断, **学不到隐式约定** (比如"这个项目所有 ID 用 ULID 不用 UUID")
- 容易写得**过于通用** ("这是一个 Node.js 项目, 使用 npm 管理依赖"——废话)
- **不知道哪些是坑** (别人踩过、被 git 历史掩盖的雷区)
- **抓不到红线** (哪些代码不能改、哪些数据不能动)

所以第 1 步的实际操作是: 跑 `/init`, 然后**亲自审**生成的内容, 删掉废话, 标记"这里我也不确定的地方"。

第 2 步: 让 Agent 当考古学家 (补 `/init` 学不到的)

```
我: 你刚才生成的 AGENTS.md 是基于配置文件推断的。现在做更深的探索,
    不要修改任何代码, 但要回答:

    1. 画一张模块依赖图(用 mermaid)
    2. 找出 5 个最复杂的文件, 告诉我它们做什么
    3. 翻一下 git log 最近 50 个 commit, 看有没有反复修同一个 bug
```

(这通常意味着那里有隐藏的复杂度)
4. 找出测试覆盖薄弱的地方
5. 列出代码里你看不懂或者觉得可疑的地方
6. 找出"约定但没写下来"的东西:
   - 命名风格、错误处理模式、日志格式
   - 哪些目录有特殊规矩

第 6 条最有价值——这正是 `/init` 抓不到的隐性知识。

第 3 步: 跑通本地环境

```
我:帮我把这个项目在本地跑起来。
   遇到错误就告诉我,不要瞎猜配置。
   把每一个手动步骤记下来,完事后我们更新到 AGENTS.md。
```

环境跑通本身是巨大的进展。**很多隐性知识就藏在 setup 过程里**: 某个端口要改、某个环境变量没文档、某个服务要先起来——这些 `/init` 永远抓不到。

第 4 步: 把考古发现 + setup 过程合并回 AGENTS.md

```
我:把以下内容合并到 AGENTS.md:
   - 第 2 步你发现的隐式约定
   - 第 3 步我们跑通环境的步骤
   - 你发现的可疑/复杂区域(标为"⚠ 接近时要小心")

   保持文件在 400 行以内,删掉初版里的废话。
```

然后你**逐行审**这份合并后的版本。这一步是你理解项目的最快路径——比自己读代码快 10 倍。

第 5 步: 做一个低风险的小任务热身

不要一上来就改核心。挑一个: 加个日志、补一个测试、修一个文档错字。借这个任务跑完整的"读代码 → 修改 → 测试 → 提交"循环, 你和 Agent 都熟悉协作节奏。

跑完之后再问 Agent:"这次任务里你有没有发现 AGENTS.md 缺了什么? 补进去。"——AGENTS.md 是活的。

第 6 步: 再处理你真正想做的事

> **反模式 1**: 接手项目第一句话是"帮我加 X 功能"。Agent 没有上下文, 会写出和现有风格完全不一致的代码, 后患无穷。
>
> **反模式 2**: 跑了 `/init` 就以为 AGENTS.md 弄好了。生成的初版只是骨架, **真正有用的内容是后面几步加的**。直接用初版的人, 后面会持续踩坑。

五、上下文管理: 压缩、切换、清零

Agent 的上下文窗口是有限的。一个 session 太长会有几个症状:

- 开始重复犯之前已经纠正过的错
- 忘记 spec 里的关键约束
- 工具调用变慢、变笨
- 开始"幻觉"项目里不存在的文件名
- 输出变水、变敷衍

什么时候该换/压缩?——识别"合适的时机"

新手最容易错过的部分。**不是等 Agent 变笨才换, 而是在"自然停顿"的地方主动换**。

下面这些信号出现时, 就是合适的时机:

信号 1: 你刚完成了一个阶段性任务

- ✓ 刚把登录功能做完了, 接下来要做主页
- ✓ 刚调试好一个 bug, 接下来要加新功能
- ✓ 刚跑通了开发环境, 接下来要写第一个功能

做完一件事 = 新 session 的最好时机。这时候不换, 你的"做主页"对话会带着一堆"做登录"的旧信息, 白白占地方。

信号 2: 你要切到不连贯的话题

- ✓ 刚在讨论数据库设计, 现在要切到调样式
- ✓ 刚在写后端逻辑, 现在要切到部署
- ✓ 刚在 debug, 现在要做完全无关的新功能

两件事关联不大 → 开新 session。让旧话题留在旧 session 里, 新话题用干净上下文。

信号 3: 工具显示用量过半

Claude Code、Cursor 这些工具会显示上下文使用率。粗略警戒线:

- **0-40%**: 随便用
- **40-65%**: 留意, 准备做收尾的安排
- **65-80%**: 开始写 handoff 文档, 准备开新 session
- **>80%**: 进入紧急模式, **立刻**写 handoff 然后切

信号 4: 你感觉对话"变沉重了"

主观但准确。当你觉得滚动起来好长、回到主题前要扫一遍前面在干嘛、Agent 回复变慢——就是该换的时候了。

Agent 工具自带的命令

命令	作用	什么时候用
`/compact`	把已有对话压缩成摘要, 继续聊	中途不想换 session, 但想腾点空间
`/clear`	清空对话, 重新开始 (同一窗口)	想换话题但懒得开新窗口
`/init`	自动扫描项目生成 AGENTS.md	接手项目第一件事
`/resume` 或 `--continue`	接着上次的对话继续	昨天没做完, 今天接着干

关于 `/compact`: 它会保留 AI 自己写的"摘要", 丢掉对话原文。这有代价——细节会丢。所以:

- ✓ 用在"已经完成的工作要清场"
- ✗ 不要用在"任务做到一半, 关键代码细节还在对话里"——细节会被压没

最稳的策略其实不是 `/compact`, 而是:

完成阶段任务 → 让 Agent 写 handoff 文件 → 直接开新 session

这比 `/compact` 可靠, 因为你**亲自决定**什么留下、写到文件里; 而 `/compact` 是 AI 自己决定保留什么, 可能丢掉你在意的东西。

紧急处理: 已经爆了

最差选择: 继续硬聊。Agent 每一轮都在更糟的状态下回答, 会写出垃圾代码、做出错误判断。

正确做法, 按优先级:

A. 立刻保存还没丢的上下文

在窗口彻底崩之前, 做一件最重要的事——**让 Agent 把当前状态写到文件**:

```
停一下。在你忘记之前,把现在的工作状态写到
docs/notes/handoff.md,包括:
- 我们在做什么(任务、spec 链接)
- 已经改了哪些文件
- 哪些测试通过、哪些没跑
- 下一步计划是什么
- 你脑子里现在有什么"还没说出来的"重要发现
```

哪怕窗口已经很满, 这个动作通常还能挤出来。**永远不要让一个塞满的 session 直接结束, 要榨出 handoff 文档**。

B. 开新 session, 加载 handoff

```
[新 session]
我:读 AGENTS.md。读 specs/feature.md。读 docs/notes/handoff.md。
   我们继续上一个 session 没做完的事。
```

新 session 上下文干净, 通过文件继承前一个 session 的关键状态。**这通常比任何"压缩"或"清理"都有效**。

预防: 从一开始就管理上下文

1. 控制单 session 的"任务半径"

不要在一个 session 里做"探索仓库 + 设计架构 + 实现功能 + 写测试 + 改 bug"。每一项是一个 session。

经验值: - 探索/调研类:1-2 小时一个 session - 实现类:1 个明确子任务一个 session(不超过 ~10 个文件改动) - 大重构: 每个 phase 一个 session

2. 把"会膨胀的输出"赶到 Subagent 去

需要"扫整个 codebase"、"读 50 个文件"、"跑长 build log"的任务, **永远派 Subagent**。Subagent 读 50 个文件不会污染你的主上下文, 只把结论带回来。

3. 别让 Agent 重复读同一个文件

很常见:Agent 第 5 次问你之前已经讨论过的内容。原因是早期对话被压缩/淘汰了。

应对: **关键决策、关键约束、关键代码片段, 持久化到文件**。不要只在对话里说。

```
✗ 我:这个函数返回值用 Result<T, E>
   [10 轮对话后] Agent 又开始用 throw

✓ 我:把"所有新函数返回 Result<T, E>"加到 AGENTS.md 的"编码约定"
```

4. 别 paste 大文件, 引用路径

```
✗ 我:[粘贴 5000 行的 schema.sql] 这是 schema,帮我...
✓ 我:schema 在 db/schema.sql,你需要的部分自己 grep。
```

让 Agent 自己拉它需要的部分, 而不是把整个文件强塞进上下文。

5. 长 session 保活技巧

如果你必须做一个长任务:

- **定期 commit + 总结**: 每完成一个里程碑,commit + 写一段"目前为止做了什么"
- **关闭不需要的工具**: 工具的 schema 本身占 token, 用不到就关
- **简短回复模式**: 让 Agent "回复保持在 3 段以内, 除非我说要详细"
- **明确关闭已解决话题**:"X 问题已经解决, 后续不再讨论"

  核心心法: 上下文是消耗品, 不是无限资源。最好的"长 session"是一连串短 session, 通过文件接力。永远不要把记忆托付给上下文窗口——文件才是 Agent 的硬盘。

六、Subagent 的使用

Subagent = 主 Agent 派出的"专项小队", 独立的上下文, 完成特定任务后只把结果带回来。

为什么要用 Subagent

核心价值是隔离上下文。举例:

- 主 Agent 在做"实现支付功能"这个大任务
- 中间需要"在整个 codebase 里搜索所有用到了旧 PaymentClient 的地方"
- 如果主 Agent 自己做, 会读入大量不相关的文件, 污染上下文
- 派一个 Subagent 去做,Subagent 读 50 个文件, 只把"找到了 12 处, 列表如下"返回给主 Agent

主 Agent 的上下文保持干净, 只多了一行关键信息。

适合派给 Subagent 的任务

任务类型	例子
大范围搜索	"在 codebase 里找所有过期的 API 用法"
独立的探索	"研究这个第三方库的最佳实践, 给出建议"
并行的子任务	"分别给 module A、B、C 写测试"
重复性工作	"把 src/ 下所有 console.log 替换成 logger.debug"
验证类任务	"跑测试、读输出、判断是否通过"

不适合派 Subagent 的任务

- 需要主 Agent 上下文的任务 (Subagent 看不到主对话)
- 极小的任务 (派出去的开销大于收益)
- 需要多轮和你交互的任务 (Subagent 通常不和用户对话)

Subagent 调度的实战模式

模式 1: 扇出 (fan-out) 主 Agent 发现需要并行做 5 件事, 同时派 5 个 Subagent, 等都返回后汇总。

模式 2: 深度委托"这个任务的具体实现你不用看, 派一个 subagent 去做, 告诉我结果"——主 Agent 保持高层视角。

模式 3: 专家 Subagent 预定义角色:`code-reviewer`、`security-auditor`、`test-writer`。每个有自己的系统提示。例如审 PR 时自动调 `code-reviewer`, 它专注于审查不分心。

给 Subagent 的指令要更明确

主 Agent 你可以慢慢聊。Subagent 是"一次性"的, 指令要像写函数签名:

```
Subagent 任务: 在 src/ 下找所有直接调用 process.env 的地方
输入: 无
输出: 一个列表,每行格式 `path:line - 用了哪个变量`
约束: 不要修改任何文件;不要进入 node_modules
完成标准: 输出列表 + 总数
```

七、Agent 协作模式与 Workflow: 把任务编排明白

当一个 Agent 搞不定一个任务, 你有两条路: **让 Agent 自己想办法** (autonomous agent), 或者**你把任务流程编排好,Agent 按编排做** (workflow)。绝大多数人混淆了这两件事, 导致写出来的"agent 系统"既不可靠也不可调试。

这一章把这个混乱讲清楚。

7.1 一个根本区分:Workflow ≠ Agent

Anthropic 在《Building Effective Agents》里给了一个干净的定义, 值得记住:

	Workflow(工作流)	**Agent(自主代理)**
谁决定流程	**你** (代码/编排预先定义)	**LLM 自己** (在每一步动态决定下一步)
流程结构	固定步骤 (可能有分支)	循环 (LLM 决定何时停)
可预测性	高	低
调试难度	低 (看哪一步挂了)	高 (LLM 决策不透明)
成本	可控	容易爆炸
适用场景	任务结构清晰	任务开放、步骤数不可预知

举例对比:

- **Workflow**: 你定义"先翻译 → 再润色 → 再检查格式"。LLM 只在每步内做事, **步骤是你写死的**
- **Agent**: 你给 LLM 一个目标"把这个 PR review 完", 给它一组工具 (读文件、跑测试、评论), **它自己决定**先读哪个、跑什么测试、评论几条、什么时候算完

关键洞察: 能用 workflow 解决的, 千万不要用 agent。原因:

- workflow 可预测、可测试、可缓存、便宜
- agent 灵活但每个不可控变量都是事故源
- 80% 的"agent 项目"其实是被包装成 agent 的 workflow——拆成 workflow 之后立刻好用 10 倍

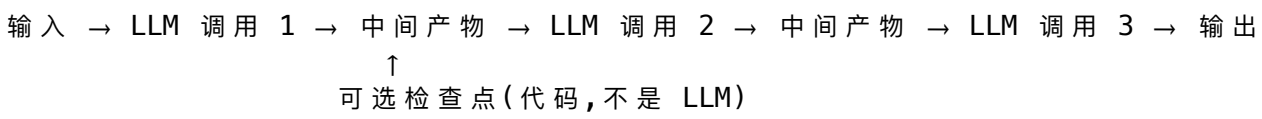
判断标准:

> 这个任务的**步骤数**, 我能事先知道吗? 能 → workflow。不能 → agent。

7.2 五种核心 Workflow 模式

任务结构可预测时, 选合适的 workflow 模式。下面这五种是 Anthropic 总结的标准模式, 覆盖 90% 的实际场景。

模式 1:Prompt Chaining(链式调用)　结构:



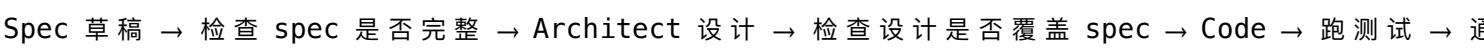
核心思想: 把一个大任务拆成几步, 每步一次 LLM 调用, 前一步的输出是后一步的输入。**步骤之间可以有代码做检查/校验**——这是质量保证的关键点。

适用场景: - 任务可以清晰拆分成几步 - 每步的输出可以检验 (格式、长度、是否包含某关键词) - 用"准确性换延迟"是值得的

实战例子: 翻译 + 润色 + 检查

```python
def translate_with_review(text):
    # Step 1: 直译
    raw = llm_call(f" 把这段中文直译成英文, 不要润色:\n{text}")

    # 检查点: 代码做的, 不是 LLM
    if len(raw) < 10:
        raise ValueError(" 翻译太短, 可能失败")

    # Step 2: 润色
    polished = llm_call(f" 润色这段英文, 使其自然流畅:\n{raw}")

    # Step 3: 一致性检查
    check = llm_call(
        f" 原文:{text}\n译文:{polished}\n"
        f" 译文有没有遗漏或错译? 只回答 yes/no + 理由"
    )

    return polished, check
```

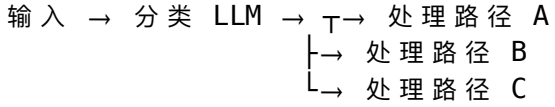
Vibe Coding 里最常见的链式 workflow:



Vibe Coding 实战 prompt 模板 (让 Agent 帮你串起来):

```
我:这是一个三步任务,按顺序做,每步做完先给我看再继续:

Step 1: 读 specs/001.md,提取所有"必须实现"的功能,列成 checklist
Step 2: 我确认 checklist 后,你给每项设计实现方案
Step 3: 我确认方案后,你按顺序实现,每完成一项打勾
```

模式 2:Routing(路由)　结构:



核心思想: 先用一个轻量 LLM 调用判断"这是什么类型的任务", 再分发给对应的专门 prompt/模型。

适用场景: - 输入种类多、每类需要的处理不同 - 一个 prompt 要覆盖所有情况会变得臃肿 - 有些类型可以用便宜模型, 有些需要贵模型

实战例子: 客服系统

```
def handle_query(user_message):
    # 路由层: 用便宜模型分类
    category = llm_call_cheap(
        f" 这条消息属于哪类?refund / technical / general\n消息:{user_message}",
        model="haiku"
    )

    if category == "refund":
        return handle_refund(user_message)  # 用 sonnet, 有 refund 知识
    elif category == "technical":
        return handle_technical(user_message)  # 用 opus, 带工具
    else:
        return handle_general(user_message)  # 用 haiku, 简单回复
```

Vibe Coding 实战例子: 多语言代码 Review

```
路由 LLM:看 PR 涉及哪些语言?
├─ Go → 用 go-reviewer subagent (system prompt 强调 errwrap、context、defer)
├─ Python → 用 py-reviewer subagent (system prompt 强调 type hints、async)
└─ TypeScript → 用 ts-reviewer subagent (system prompt 强调 strict mode、no any)
```

每个 reviewer 是独立的 Agent, 有自己的专精 system prompt。

模式 3:Parallelization(并行) 结构:

```
输入 → ┬→ LLM 调用 A ─┐
       ├→ LLM 调用 B ─┼→ 聚合 → 输出
       └→ LLM 调用 C ─┘
```

两种子模式:

3a. Sectioning(分片): 把任务拆成独立的子任务, 并行做, 合起来

```
输入:一份 50 页的合同
拆分:章节 1-5,6-10,11-15,16-20,21-25
并行:5 个 LLM 同时审各自负责的章节
聚合:合并所有发现的风险
```

3b. Voting(投票): 同一个任务跑多次, 多数派胜出/取平均

```
输入:这段代码有 SQL 注入吗?
并行:跑 5 次,用不同的 prompt
聚合:任何一次报有问题 → 标记为可疑;5 次都说没事 → 通过
```

适用场景: - 子任务真正独立, **不互相依赖** - 速度比一致性重要 - 或者: 你需要多次采样来提高信心 (像 vote 模式)

Vibe Coding 实战: 多 worktree 并行

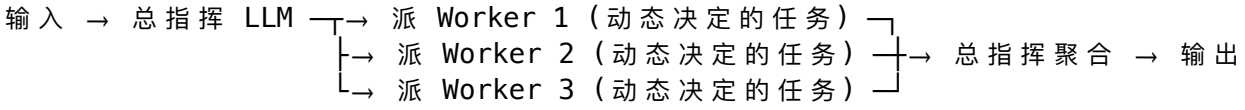
```
你:把 spec 拆成 5 个独立模块的 issue
   ↓
为每个 issue 创建 worktree (第九章讲过)
   ↓
每个 worktree 跑一个 Agent
   ↓
```

```
你 review 每个 PR、合并
```

这就是”sectioning”模式。**前提**是模块真正独立——会改同一个文件的任务不能这么搞，要串行。

Voting 在测试 Agent 行为时也用：同一个 prompt 跑 3 次 (回顾第十三章测试方法) 就是 voting——不是为了选最好的，是为了**评估稳定性**。

模式 4:Orchestrator-Workers(总指挥 - 工人)　结构：



和 Parallelization 的关键区别：子任务不是预先定义的，而是**总指挥根据输入动态拆分**。

核心思想：总指挥读完输入，自己决定要派哪些 Worker、每个干什么，然后并行/串行执行，最后总指挥聚合结果。

适用场景：- 子任务**事先不知道**，要看输入才能决定 - 但子任务一旦确定就是独立的、可以并行

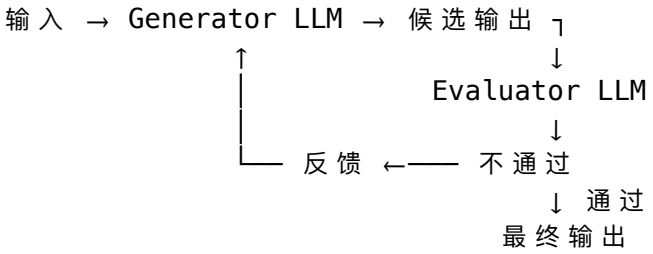
实战例子：跨多个文件的代码改动

```
输入:"重命名变量 user_id 到 userId"
总指挥:
  - 先 grep 找到所有用到 user_id 的文件(20 个)
  - 派 20 个 worker,每人改一个文件
  - 等所有 worker 返回
  - 跑测试,确认改完没坏
  - 输出 PR
```

总指挥事先不知道是 20 个文件还是 5 个文件——它**读了输入才知道要派多少 worker**。这就是 Orchestrator-Workers 和 Parallelization 的区别：后者的拆分是写死的，前者是 LLM 决定的。

Vibe Coding 落地：这就是 Claude Code 的 **Subagent** 机制 (第六章讲过)。主 Agent 是总指挥,Task tool 派 subagent 是 worker。

模式 5:Evaluator-Optimizer(评估器 - 优化器)　结构：



核心思想：一个 LLM 写答案，另一个 LLM 评分；不通过就把反馈传回去重写，通过为止。

适用场景：- 有明确的评价标准 (可以用 prompt 表达) - 第一次生成往往不够好，迭代能显著提升 - 输出值得”多花几次调用”

实战例子：翻译质量优化

```python
def quality_translate(text, max_iter=3):
    feedback = ""
    for _ in range(max_iter):
```

```
        translation = llm_call(
            f" 翻译:{text}\n之前的反馈:{feedback}\n请改进"
        )
        review = llm_call(
            f" 原文:{text}\n译文:{translation}\n"
            f" 评分 (1-10)+ 改进建议。如果 ≥9 就回答 PASS"
        )
        if "PASS" in review:
            return translation
        feedback = review
    return translation
```

Vibe Coding 实战: 对抗式 Code Review

```
Coder Agent 写代码 → Reviewer Agent 评审 → 通过?
                              | 不通过
                              ↓
                  反馈给 Coder Agent → 重写 → ...
```

第七章原本的"对抗式协作"模式, 本质上就是 Evaluator-Optimizer。

关键技巧:Evaluator 必须有**明确的退出条件**, 否则会无限循环。常见做法: - 最多迭代 N 次 (N=3 是经验值) - 评分阈值 (≥9/10 通过) - Evaluator 显式输出"PASS"

7.3 什么时候才需要真正的 Agent(自主代理)

把上面五种 workflow 都试过、都不合适, 才考虑真 Agent。判断特征:

- **任务步骤数不可预知** (可能 3 步, 可能 30 步)
- **需要根据中间结果动态调整策略**
- **需要使用工具的组合** (读文件 → 搜索 → 跑测试 → 改代码 → 再跑测试 → ...)
- **没有明确的"完成"条件**, 只有"够好就停"

典型 Agent 任务: - "帮我修复这个 bug"(不知道要读多少文件、跑多少次测试) - "做一个 PR review"(不知道要 deep dive 哪几个文件) - "我想加 X 功能, 你看着办"(发现什么坑解决什么)

Agent 的核心循环 (Anthropic 的术语叫"augmented LLM"):

```
loop:
  observation = 读取当前状态(代码、测试结果、用户输入)
  thought = LLM(observation, history) → 我接下来要干什么?
  action = 调一个工具(读文件 / 改代码 / 跑测试 / 完成)
  if action == "完成": break
  history.append(thought, action, action_result)
```

Claude Code、Codex 等就是这个循环的封装。当你直接和它们对话, 你用的就是 Agent; 当你写一段编排代码、按步骤调 LLM, 你做的是 Workflow。

Agent 用得安全的几个原则

1. **限制工具集**: 工具越多,Agent 越容易"自由发挥"。**只给当前任务必需的工具**
2. **限制最大步数**:max_iterations=20 比无限循环安全

3. **每步可观察**:Agent 的每一步动作要 log, 出问题能回放
4. **关键操作要 confirmation**: 删数据库、`rm -rf`、push 到 main——这些必须人确认
5. **预算上限**: 设置 token 上限, 跑爆就停; 贵的事故都是 agent 失控

7.4 多 Agent 协作模式 (Agent 之间怎么分工)

上面讲的是**单条任务怎么编排**。下面讲**多个 Agent 之间怎么分工**——这是另一个维度。

协作模式 1: 专科分工 (Specialist Agents) 不同 Agent 擅长不同事, 你**手动**调度:

Agent	擅长	用途
Claude Code(主力)	项目理解、改代码、跑工具	实际实现
o-series / DeepSeek-R1	复杂推理、debug 难题	架构决策、卡住的 bug
GPT-4o + browsing	查最新文档	第三方库研究
本地小模型 (Llama)	批量重复任务	跑大量小请求

你做的事是"把 Agent A 的输出复制给 Agent B, 加上你的 framing"。听起来手工, **但目前最稳定的多 Agent 协作方式**——比让 Agent 们自动协作可靠得多。

实战例子:

```
1. Claude Code 卡在一个奇怪的 race condition(无法定位)
2. 你:把日志和相关代码复制出来,贴给 o3
   "我有这样一段代码 + 日志,Claude Code 已经试过 X、Y、Z 都不行,
    你能想出别的可能原因吗?"
3. o3 给出 5 个可能性
4. 你回到 Claude Code,把这 5 个可能性贴进去,让它逐个验证
5. 找到原因,修复
```

协作模式 2: 角色分工 (Role-based) 同一个工具/模型, **不同的 system prompt = 不同的 Agent**。常见角色:

- **Architect**: 只设计、不写代码
- **Coder**: 只实现、不审查
- **Reviewer**: 只审查、不实现
- **Tester**: 只写测试
- **Doc Writer**: 只写文档

怎么落地: 每个角色一个 Skill(第十章), 或者一个 Claude subagent 配置。

实战例子: 重要功能的"四角色流水线"

```
1. Architect Skill 出设计 → docs/design/feat.md
   (system prompt: "你只设计,不写代码。考虑 5 个边界 case,
    考虑可测试性,考虑兼容性。")

2. Coder Skill 读 design → 实现
   (system prompt: "严格按 design 实现。不要做 design 没说的事。
    不要重构现有代码。")
```

```
3. Reviewer Skill 读 diff → 出 review
   (system prompt: "你是严格的 code reviewer。找问题,不夸奖。
    每条评论必须可操作。")

4. Tester Skill 读 spec + diff → 补测试
   (system prompt: "覆盖 happy path + 至少 3 个边界 case。
    用项目现有测试风格。")
```

为什么不能让一个 Agent 干所有事?——因为它会**自我合理化**:Coder 不会说自己写的代码差,Reviewer 必须是另一个角色才会真的找问题。

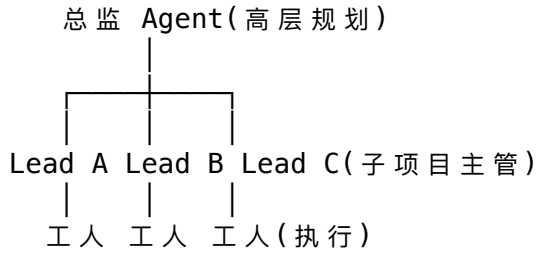
协作模式 3: 对抗式 (Adversarial) 让两个 Agent 互相挑刺 (其实是 Evaluator-Optimizer 的人格化版本):

```
我:这是 Agent A 写的方案。你扮演资深工程师,**专门找问题**。
   不要赞美,不要说"整体不错",直接列漏洞。
```

Agent **自评自洽** (自己评自己会陷入合理化), 但**评别人写的会客观得多**。这是利用了 LLM 的一个特性: 它对”输入”批判性更强, 对”自己刚生成的”维护性更强。

实战例子: - 设计 review: 用对抗式 - 代码 review: 用对抗式 - 找 prompt 漏洞: 用对抗式 (“你扮演恶意用户, 试着让这个客服 bot 说出公司机密”)

协作模式 4: 层级分工 (Hierarchical)

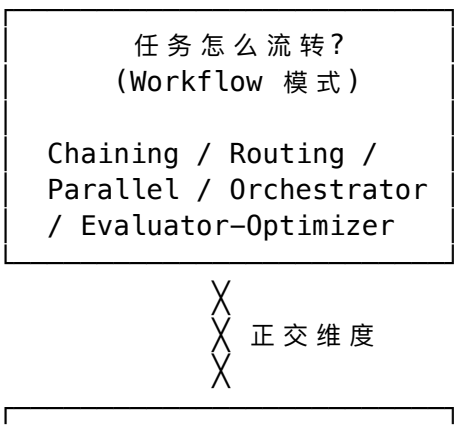


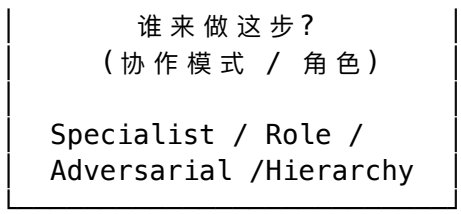
总监给 Lead 派活,Lead 拆解后给工人。**这是 Orchestrator-Workers 的多层版本**。

慎用: 层级越深, 误差累积越快。**实战中两层 (主 Agent + Subagent) 就够, 三层以上几乎都会失控**。

7.5 协作模式 vs Workflow 模式: 关系图

很多人把这两个搞混。一张图说清:





任何一个真实系统是**两个维度的组合**。例如:

> "我用 Prompt Chaining(workflow) 把任务拆成 spec → code → review, 其中 review 那一步用 Adversarial 协作 (协作模式) 让两个 Agent 互相挑刺"

这句话里两个维度都用了。**先想清楚 workflow, 再决定每步谁来做**。

7.6 协作的"协议层": 文件 + 文件夹

不论用哪种模式,**Agent 之间不直接对话, 而是通过结构化的文件交换信息**。这是所有成功多 Agent 系统的共同特点。

推荐的项目结构:

```
.
├── AGENTS.md                    # 所有 Agent 都读
├── specs/                       # 需求规范
│   ├── 001-auth.md
│   └── 002-billing.md
├── docs/
│   ├── architecture.md
│   ├── decisions/               # ADR,关键决策
│   └── notes/                   # 临时笔记、session 总结
├── .agent/                      # Agent 工作区
│   ├── tasks/                   # 待办、进行中、完成
│   ├── reviews/                 # 审查结果
│   ├── handoffs/                # session 交接文档
│   └── runs/                    # 测试 case 的运行记录
└── src/
```

`.agent/` 文件夹是 Agent 的"白板":

- **任何 Agent 都可以读写**——不绑定特定工具
- **持久化**——跨 session、跨重启都在
- **可审计**——出了问题你能看到每个 Agent 的发言
- **可版本控制**——`.agent/handoffs/` commit 进 git, 每次决策有迹可循

7.7 决策框架: 面对一个任务, 怎么选?

```
任务来了
  │
  ├─ 单个 LLM 调用够不够?
  │    ├─ 够 → 就一次调用,搞定
  │    └─ 不够 → 继续往下
```

```
│
├─ 步骤数事先知道吗?
│    ├─ 知道 → Workflow
│    │    ├─ 顺序固定 → Prompt Chaining
│    │    ├─ 要分类 → Routing
│    │    ├─ 子任务独立 → Parallelization
│    │    ├─ 子任务要 LLM 决定 → Orchestrator-Workers
│    │    └─ 要迭代质量 → Evaluator-Optimizer
│    └─ 不知道 → Agent
│         └─ 限制工具集、限制步数、加 confirmation
│
└─ 需要多个 Agent 吗?
     ├─ 不同擅长 → Specialist
     ├─ 不同 system prompt → Role-based
     ├─ 要客观批判 → Adversarial
     └─ 任务大且分层 → Hierarchical(慎用,最多两层)
```

7.8 反模式

反模式	后果	怎么改
能用 workflow 偏要用 agent	不可控、贵、难调试	先尝试 workflow
让 Agent 评自己的输出	自洽幻觉,review 全过	用 Adversarial 模式或换角色
多 Agent 互相 chat	上下文爆、绕圈	通过文件交换信息
Agent 层级 >2 层	误差累积失控	扁平化, 最多主 + sub
没有退出条件的 Evaluator-Optimizer	死循环烧钱	max_iter + 评分阈值
Parallelization 子任务有依赖	脏数据、冲突	改成 Chaining 或 Orchestrator
把 Agent 当万能锤	简单任务用复杂方案	用最弱的工具解决问题

> **核心心法:Agent 协作不是"让 AI 们自己玩", 而是"把任务编排清楚, 每步用最合适的工具"**。Workflow 决定流程, 协作模式决定角色。两者搭配, 先编排再分工。能用代码控制的, 绝不交给 LLM。

八、`.gitignore`: 你必须懂的一个文件

在讲多 Agent 工作树之前, 必须先讲清楚 `.gitignore`——它是后面好几章的前提知识, 而很多人对它的理解模糊。

一句话:.gitignore = "告诉 git 哪些文件不要跟踪"

它就是仓库根目录 (或子目录) 下的一个普通文本文件, 每行一个 pattern。git 看到匹配的文件就**装作它不存在**: 不显示在 `git status`、不允许 `git add`、不会被 commit。

```
# .gitignore 的样子

# 依赖,可以从 package.json 重装
node_modules/
```

```
# 编译产物,可以从源码重新构建
dist/
build/
*.pyc
__pycache__/

# 环境变量,含敏感信息
.env
.env.local

# IDE 配置,每个人不一样
.vscode/
.idea/
.DS_Store

# 日志、缓存
*.log
.cache/
```

为什么要有 `.gitignore`?

仓库里**只应该有"源头"文件**, 不应该有"产物"文件和"环境特定"文件。

类型	例子	为什么不进 git
依赖	`node_modules/`, `.venv/`	可以从 lockfile 重装, 几百兆, 每台机器不一样
编译产物	`dist/`, `*.pyc`, 二进制	可以从源码重建, 且会膨胀仓库历史
环境配置	`.env`, 个人 IDE 设置	含敏感信息 (密钥) 或个人偏好
系统垃圾	`.DS_Store`, `Thumbs.db`	操作系统乱塞的东西
日志/缓存	`*.log`, `.cache/`	运行时产物, 没意义

如果不 ignore, 后果是:

- 仓库变成几个 GB(node_modules 单个就 500MB+)
- `git status` 永远是几百行噪音
- 不小心 commit 了 `.env` → API key 上传到 GitHub → 被爬虫扫到 → 钱包损失
- 不同电脑上的 IDE 配置互相覆盖

怎么写.gitignore

正确做法不是手写, 而是用现成的:

1. **GitHub 自动生成**: 新建仓库时勾选语言/框架,GitHub 帮你生成
2. **gitignore.io**: 访问 https://www.toptal.com/developers/gitignore, 输入 `node, python, vscode, macos`, 直接给你完整的
3. **让 Agent 帮你生成**:

```
我:这个项目用 Next.js + Python 后端 + VSCode 开发,在 Mac 上,
   帮我生成一份 .gitignore。
```

手写时的 pattern 语法:

```
node_modules/          # 这个目录(及其内容)
*.log                  # 任何 .log 结尾的文件
build/                 # build 目录
/config.json           # 只忽略根目录的 config.json,不忽略子目录的
!important.log         # 例外:不忽略这个特定文件
**/temp/               # 任何深度的 temp 目录
```

.gitignore 的常见坑

坑 1: 已经被跟踪的文件, 加.gitignore 不会让它消失

```
# 你已经 commit 了 .env(糟糕)
# 现在你加进 .gitignore
echo ".env" >> .gitignore
git add .gitignore && git commit -m "ignore env"

# 但 .env 仍然在 git 历史里!
# 必须显式移除:
git rm --cached .env
git commit -m "remove .env from tracking"
```

更严重的情况: 如果你 commit 过敏感信息 (API key、密码), 光删除新 commit 不够——历史里还有。需要用 `git filter-repo` 或 `BFG Repo-Cleaner` 重写历史。**最稳的做法是立刻轮换泄漏的密钥。**

坑 2: 子目录的.gitignore 会覆盖父目录

每个目录都可以有自己的 `.gitignore`, 作用域是该目录及其子目录。规则是"最近的.gitignore 优先"。

坑 3: 全局.gitignore vs 项目.gitignore

- **项目.gitignore**: 在仓库里, 所有协作者共享 → 写"这个项目共同需要 ignore 的"(node_modules、dist)
- **全局.gitignore**:`~/.gitignore_global`, 只对你生效 → 写"你个人的工具产物"(`.DS_Store`、`.idea/`)

很多人把 `.idea/` 写进项目.gitignore——其实更优雅的是写到全局, 因为不是所有人都用 JetBrains。

```
# 配置全局 gitignore
git config --global core.excludesfile ~/.gitignore_global
```

.gitignore 在 Vibe Coding 里为什么特别重要

三个原因:

1. Agent 会"看到"被 ignore 的文件, 但不知道它们不该被 commit

如果你的 `.env` 不在.gitignore 里,Agent 在 `git add .` 时会顺手把它加上。**Agent 默认假设你的.gitignore 是对的**——所以你必须**先把.gitignore 弄对, 再让 Agent 操作 git**。

2. 接手项目的第一件事就是审.gitignore

跑 `/init` 之前先看一眼:

```
cat .gitignore
git status --ignored      # 看哪些文件被 ignore 了
git ls-files | head -50   # 看哪些文件被跟踪了
```

如果发现 `.env` 没被 ignore、或者 `dist/` 在跟踪——**这是项目卫生问题, 先修了再开始**。

3. 工作树会被.gitignore 直接影响

下一章会展开讲——简单说:`.gitignore` 里的东西, 在新 git worktree 里**不会出现**。这是 worktree 最大的坑, 本质上是.gitignore 的副作用。

Agent 协作里的.gitignore 增补

除了通用模式,Vibe Coding 项目通常还要 ignore 这些:

```
# .gitignore 的 Vibe Coding 部分

# Agent 的工作区(可选,看你想不想 commit)
.agent/runs/
.agent/cache/

# 测试 Agent 行为时收集的证据
runs/

# Agent 的临时输出
*.handoff.md.tmp

# 大模型本地缓存
.llm-cache/

# 但是要保留这些(不 ignore,要 commit):
# AGENTS.md
# CLAUDE.md
# .claude/   ← Skill 定义
# .codex/    ← Skill 定义
# specs/     ← 规范文档
# docs/      ← 项目文档
```

> **心法:**`.gitignore` 是仓库的"卫生公约"。一个写得好的.gitignore = 干净的 `git status` + 安全的 commit + 顺畅的多人协作。一个写得烂的.gitignore = 不停 commit 垃圾 + 偶尔泄漏密钥 + worktree 一直报错。

九、Codex 工作树: 多 Agent 并行的硬件基础

前面讲的多 Agent 协作, 在文件系统层面会撞上一个硬问题: **两个 Agent 同时改同一个 working directory 会互相破坏**。Git worktree 是解决这个问题的关键机制——OpenAI 的 Codex App 把它做成了内置功能, 但原理对所有 AI Coding 工具 (Claude Code、Cursor、Aider 等) 都通用。

什么是 git worktree

一句话: **同一个仓库, 多个 checkout, 共用一份 `.git` 历史**。

```
my-repo/                    ← 主工作树,你自己用
├── .git/                   ← 唯一的对象库
└── src/
```

```
../wt/feature-auth/          ← worktree 1,Agent A
└── src/                     (checkout 在 feature-auth 分支)

../wt/bugfix-payment/        ← worktree 2,Agent B
└── src/                     (checkout 在 bugfix-payment 分支)

../wt/refactor-db/           ← worktree 3,Agent C
└── src/                     (checkout 在 refactor-db 分支)
```

三个 Agent 在三个独立目录里干活, 各自的 `git add`、`git commit`、文件修改完全隔离, **但提交到的是同一个仓库**。

为什么 vibe coding 需要它

不用 worktree 跑多 Agent 会遇到的灾难:

- **index 文件竞争**:`.git/index` 是单文件, 两个 Agent 同时 `git add` 会丢改动
- **HEAD 是全局的**:Agent A 切到 branch X,Agent B 莫名其妙也在 X 上工作了
- **dev server 端口冲突**: 都想用 3000 端口
- **node_modules / lockfile 抢着改**

worktree 把这些问题一次性解决——每个 Agent 一个独立目录, 独立 HEAD, 独立 working tree。

手动用 worktree 的命令

不需要 Codex App,git 自带:

```
# 准备一个目录放所有 worktree
mkdir -p ../wt

# 从 main 分支创建 3 个 worktree, 每个一条新分支
git worktree add -b agent/auth      ../wt/auth      main
git worktree add -b agent/payment   ../wt/payment   main
git worktree add -b agent/dashboard ../wt/dashboard main

# 列出所有 worktree
git worktree list

# 用完删掉
git worktree remove ../wt/auth
git branch -D agent/auth
```

然后开三个终端, 每个 `cd` 进一个 worktree, 各跑一个 Agent。

Codex App 的工作树是怎么用的

Codex App(OpenAI 的桌面客户端) 把这个流程做成了 UI:

- 新建 thread 时选 **Local** 或 **Worktree**
- 选 Worktree 时,Codex 自动 `git worktree add` 到 `$CODEX_HOME/worktrees/thread-N/`
- 每个 thread = 一个独立 worktree = 一个独立 Agent context
- 内置 diff 查看器、commit、push、PR 都能在 app 里完成
- 多个 thread 并行跑, 互不干扰

实际效果: 你可以同时开 5 个 thread,5 个任务并行推进, 而且每个 thread 的"环境"是干净的。

worktree 最大的坑:.gitignore 里的东西不会过来

这是必须记住的一点。**worktree 只 checkout 被 git 跟踪的文件**。所有在 `.gitignore` 里的:

- `node_modules/`
- `.env`
- `.venv/`
- `dist/`、`build/`
- 各种 cache

——**新 worktree 里都没有**。

意味着 Agent 进到新 worktree 第一件事会是:`npm install` 失败/跑测试报缺依赖/找不到环境变量。

应对方式:

1. **写一个 bootstrap 脚本**:`scripts/setup-worktree.sh`

```
#!/bin/bash
# 在新 worktree 里跑这个就能跑起来
cp ../../my-repo/.env .                    # 拷贝主仓库的 env
ln -s ../../my-repo/node_modules .          # 共享 node_modules(同 OS)
# 或者 npm ci 重装
```

2. **AGENTS.md 里写明**:

```
## 工作树启动
如果你在一个新创建的 worktree 里, 先跑 `bash scripts/setup-worktree.sh`
再做其他事。.env 和 node_modules 都不在 worktree 里。
```

3. **端口冲突**: 每个 worktree 用不同的 dev server 端口, 写在启动脚本里

什么时候用 worktree, 什么时候不用

用 worktree: - 你有 ≥2 个**互不相关**的任务想并行做 (auth + payment + dashboard) - 你想做 A/B 实验:"用两种方案各做一遍, 比较结果"- 你要让 reviewer Agent 和 coder Agent 同时工作 - 长时间运行的实验性分支, 不想干扰主开发

不用 worktree(直接在主仓库里搞): - 单个任务,Agent 串行做就行 - 任务之间会改同一批文件 (并行也会冲突,worktree 解决不了语义冲突) - 项目本身配置极复杂,setup 一个新 worktree 的成本 > 收益

并行 worktree 的工作流模板

```
1. 准备阶段
   - 把任务拆成 N 个独立单元
   - 检查:这些任务会不会改同一个文件?如果会,串行做
   - 为每个任务写好 spec(放在主仓库的 specs/)

2. 派发阶段
   - 创建 N 个 worktree,每个一个分支
   - 每个 worktree 跑 setup 脚本
   - 每个 worktree 启动一个 Agent,喂对应的 spec
```

3. 监督阶段
   - 你不需要盯着每一个,Agent 在干就让它干
   - 偶尔切过去看看进度、回答问题
   - 谁先做完就先 review 谁

4. 收割阶段
   - 每个 worktree review diff、跑测试
   - 合并到 main(rebase 或 merge,看团队习惯)
   - 删 worktree、删分支

反模式

- **不写 spec 就并行派发**:5 个 Agent 跑歪了 5 个方向,review 时崩溃
- **任务之间有依赖却并行**:Agent B 等 Agent A 的接口, 但 A 还没写完
- **不读 diff 直接合并**: 并行让你想偷懒, 但 5 个分支堆一起合并,bug 也是 5 倍
- **超过 5 个 worktree**: 你 review 不过来, 变成"AI 写得多但没人看"

  心法:worktree 不是让你"管更多 Agent", 而是让你"在能管的范围内, 让 Agent 不互相干扰"。瓶颈永远是你的 review 带宽。

十、Skill 的创建: 把重复模式固化下来

Skill = **可复用的、参数化的工作流模板**。当你发现自己反复让 Agent 做同一类事 ("写一个 React 组件 + 配套测试 + Storybook"), 就该把它做成 Skill。

> Codex App、Claude Code 等都引入了 Skill 机制。具体语法各家略有不同, 但**核心理念一致**:Skill 是"被命名的提示词包"。

什么任务适合做成 Skill

判断标准: **这件事我做过 3 次以上, 而且每次都给 Agent 讲一遍同样的话。**

典型场景:

- "添加一个新的 API endpoint(改 router、controller、service、repo、加测试、更新 OpenAPI)"
- "写一份周报 (从 git log 提取、按模块分类、列出 PR)"
- "重构一个组件成 hooks 写法"
- "review 这个 PR(检查 X、Y、Z 维度)"
- "新建一个数据库迁移 (写 up/down、起名、跑 dry run)"
- "修复一个 lint 错误并补测试"

不适合做 Skill 的: - 一次性的探索任务 - 高度依赖具体上下文、模板化不了的工作 - 简单到一句话能说清的事 (做 Skill 反而是过度工程)

Skill 的组成

不管哪个工具, 一个 Skill 通常长这样:

```
.skills/                          # 或 .codex/skills/、.claude/skills/
└── add-api-endpoint/
    ├── SKILL.md                  # 触发描述 + 步骤说明(给 Agent 看)
```

```
    ├── templates/                # 代码模板
    │   ├── controller.ts.tmpl
    │   ├── service.ts.tmpl
    │   └── test.ts.tmpl
    └── scripts/                  # 辅助脚本(可选)
        └── update-openapi.sh
```

SKILL.md 是核心, 结构通常是:

```
---
name: add-api-endpoint
description: 添加一个新的 REST API endpoint。触发条件: 用户说
  " 加一个 API"、" 新建 endpoint"、" 添加路由"。包括 controller、
  service、repo、测试、OpenAPI 更新。
---

# Add API Endpoint

## 何时使用
当用户要求添加新的 HTTP endpoint 时使用本 Skill。
不要用于:
- 修改已有 endpoint(用 modify-api-endpoint skill)
- gRPC 服务 (用 add-grpc-method skill)

## 输入
- HTTP 方法 (GET/POST/PUT/DELETE)
- 路径 (如 /users/:id/avatar)
- 请求/响应 schema

## 步骤
1. 在 `internal/router/routes.go` 加路由注册
2. 在 `internal/handler/` 创建 handler, 参考 templates/controller.ts.tmpl
3. 在 `internal/service/` 加 service 方法
4. 如果涉及新数据访问, 在 `internal/repo/` 加方法
5. 在 `internal/handler/*_test.go` 加测试, 覆盖:
   - happy path
   - 参数校验失败
   - 鉴权失败 (如适用)
   - 内部错误
6. 跑 `scripts/update-openapi.sh` 更新 OpenAPI spec
7. 跑 `make test` 确认全绿

## 项目特定约定
- 错误返回用 `pkg/errors.HTTPError`, 不要直接 panic
- 鉴权 middleware 在 `routes.go` 里挂,handler 不要重复检查
- 时间字段统一用 RFC3339

## 完成标准
- [ ] 测试全绿
- [ ] OpenAPI 更新
- [ ] 至少一个手动调用的 curl 命令验证过
```

写好 Skill 的几个要点

1. description 决定能不能被触发

Agent 是根据 description 判断"现在该不该用这个 Skill"。description 必须包含:

- 这个 Skill 干什么 (动词为主)
- 触发的关键词/场景
- 不该用的边界

太宽泛的 description("帮你写代码") 会被到处触发; 太窄的没人触发。

2. 用步骤, 不用散文

Skill 是给 Agent 执行的, 不是给人读的。**编号步骤 + 明确的文件路径 + 具体命令** > 一段抒情的文档。

3. 写"项目特定约定"

通用知识 (怎么写 REST API)Agent 自己会。Skill 的核心价值是写**项目里特殊的、Agent 默认不知道的**事情。

4. 验收标准要明确

Skill 跑完了 Agent 怎么知道? 给一个 checkbox 列表。

5. 模板文件不要太大

模板是给 Agent 起步的, 不是把所有代码都写完。留 70% 让 Agent 根据具体场景填。

Skill 创建的工作流

最高效的方式: **让 Agent 帮你写 Skill**。

```
我:过去这周我让你做了 5 次"加 API endpoint"的任务。
   每次我都要讲一遍项目里的约定。

   帮我把这件事做成 Skill,放在 .skills/add-api-endpoint/。

   参考之前我们的对话,提炼出:
   - 标准步骤
   - 项目特定约定
   - 容易踩的坑

   先给我看 SKILL.md 的草稿,我审完再建文件。
```

Agent 会基于上下文里之前的工作产出一份 draft, 你迭代两轮就成。

Skill 的迭代

Skill 不是一次写完。每次用完之后, 如果发现:

- Agent 漏了某一步 → 加进 Skill
- Agent 误解了某个约定 → 在 Skill 里写得更明确
- 有了新的边界情况 → 补到 Skill 里

把 Skill 当作活的文档, 跟着项目演进。每个月 review 一次自己的 Skill 库, 删掉过时的、合并相似的。

Skill vs AGENTS.md vs Spec

类型	范围	内容
AGENTS.md	项目级, 所有 Agent 都读	项目身份、目录、风格、命令、红线
Skill	任务模板, 按需触发	某一类重复任务的标准操作流程
Spec	单个功能, 一次性	这次具体要做什么、验收标准

类比: - AGENTS.md = 员工手册 - Skill = SOP(标准作业程序) - Spec = 具体的工单

三者配合:Agent 接到 Spec(做什么), 先读 AGENTS.md(项目通识), 识别到任务匹配某个 Skill(套模板), 开始干活。

Skill 的反模式

- **写得太详细**: 把 Skill 写成 1000 行,Agent 反而抓不住重点
- **Skill 之间冲突**: 两个 Skill 触发条件重叠,Agent 不知道用哪个
- **Skill 里硬编码业务细节**: 应该参数化的东西写死了, 只能用一次
- **不维护**: 项目结构变了,Skill 还在引用旧路径, 反而误导 Agent
- **太多 Skill**:200 个 Skill,Agent 选择困难。**精选的 10 个 > 平庸的 100 个**

十一、系统提示词 vs User 提示词:Agent 行为的"硬件 vs 输入"

这是 Vibe Coding 里最容易被忽略、但**影响最大**的一对概念。理解了这俩的区别, 你写出来的 prompt 质量会上一个台阶。

一句话区分

- **System prompt(系统提示词)** = Agent 的"出厂设置"——人格、角色、能力、约束、行为准则。**整个 session 一直生效**。
- **User prompt(用户提示词)** = 你这一轮要它做的具体事情。**一次性的**。

类比:system prompt 是员工入职培训 (改一次, 全年生效);user prompt 是你今天派给他的具体工单 (一单一交)。

在不同工具里它们长什么样

Claude API / Anthropic SDK:

```
client.messages.create(
    model="claude-opus-4-5",
    system=" 你是一个资深 Go 工程师...",  # ← system prompt
    messages=[
        {"role": "user", "content": " 帮我审一下这段代码"}  # ← user prompt
    ]
)
```

Claude Code / Codex CLI:

- system prompt 写在 `AGENTS.md` / `CLAUDE.md` / Skill 的 `SKILL.md` 里——Agent 启动时自动加载
- user prompt 就是你在终端里敲的每一句话

ChatGPT / Claude.ai 网页版:

- system prompt 在“Custom Instructions”/ “Project Instructions”里
- user prompt 是聊天框里输入的内容

自己做 AI 应用时:

- system prompt 写在代码里, 作为 LLM 调用的 `system` 参数
- user prompt 是用户在你的 UI 里输入的内容 (可能你还会拼接其他变量进去)

分工原则: 什么放 system, 什么放 user?

放 system 的东西:

类别	例子
角色定义	“你是一个 Go 后端工程师, 熟悉 PostgreSQL 和分布式系统”
能力边界	“你只回答技术问题。被问到非技术问题就引导回正题”
行为准则	“改代码前先解释计划。不要引入新依赖。每步都跑测试。”
输出格式	“回答用 markdown。代码块标语言。错误用 ✗ 标记”
工具使用规则	“调 grep 之前先告诉我你在找什么”
项目知识	整个 AGENTS.md 的内容
安全/合规	“永远不要把 API key 写进代码, 用 env”

放 user 的东西:

类别	例子
当下任务	“给 UserService 加 deleteAccount 方法”
当下数据	“这是日志, 帮我分析:< 贴日志 >”
当下约束	“这次不要改 schema, 只改业务层”
临时偏好	“这次回答简短一点”

决策测试: 不确定该放哪?

问自己一个问题:**“下次开新 session 还需要这条吗?”**

- ✓ **下次还需要** → system prompt(写进 AGENTS.md / Skill)
- ✗ **只这一次需要** → user prompt(直接说就行)

例子:

- “这个项目所有金额用 decimal 不用 float” → **每次都需要, 放 system**(AGENTS.md 编码规范)
- “这次实现 deleteAccount, 要软删除” → **就这一次, 放 user**
- “你是一个会用苏格拉底法启发我的导师” → **每次都需要, 放 system**
- “今天我比较累, 讲简单点” → **就这一次, 放 user**

写好 system prompt 的 6 个原则

1. 角色 + 能力 + 约束的三段式

最经典也最好用的结构:

你 是 [角 色] 。

```
你擅长:
- [能力 1]
- [能力 2]

你必须遵守:
- [约束 1]
- [约束 2]
```

例子:

```
你是一个资深 Python 测试工程师,在一个金融科技公司工作。

你擅长:
- 用 pytest 写表驱动测试
- 识别金融业务的边界 case(精度、舍入、并发)
- 用 hypothesis 做属性测试

你必须遵守:
- 所有金额相关的测试必须用 Decimal,不要用 float
- 永远先看 conftest.py 再写新 fixture
- 不要引入 mock 之外的新测试库
- 测试名字用中文 docstring 说明意图
```

2. 用"应该"和"不应该"成对出现

只说"应该做 X"会留有歧义。配上"不应该做 Y"边界更清晰:

```
✗ 你应该写简洁的代码。
✓ 你应该写简洁的代码。具体来说:
  - 函数不超过 30 行
  - 不要为了"封装"创造单次使用的辅助函数
  - 注释只写"为什么",不要写"做什么"
```

3. 给"何时该问、何时该做"的规则

新手最容易忽略的部分。Agent 默认行为是"立刻动手",但很多场景你希望它先确认:

```
在以下情况你必须先问我,不要直接动手:
- 涉及 migrations/ 下任何文件
- 需要引入新的第三方依赖
- 改动超过 5 个文件
- 我的需求里有歧义(列出歧义点让我选)

在以下情况直接做即可:
- 修 typo
- 加日志
- 写测试
- 你已经做过一次的同类小修改
```

这条规则能省下你 80% 的"哎你怎么没问我就改了"的吐槽时间。

4. 输出格式越具体越好

```
✗ 回答要清晰。
✓ 回答按这个格式:
```

```
## 我理解的任务
[一句话复述]

## 我打算这么做
[步骤列表]

## 需要你确认的
[问题列表,如果没有就写"无"]

## 估计影响范围
[改哪些文件、风险]
```

格式严格的 system prompt 会让 Agent 输出**可预测**——你扫一眼就知道在哪看什么。

5. 用例子, 不只是规则

```
✗ 错误处理要规范。
✓ 错误处理要规范。例子:

  ✗ 不要这样:
    if err != nil { return err }

  ✓ 这样:
    if err != nil {
        return fmt.Errorf("fetching user %d: %w", id, err)
    }
```

Agent 从例子学得比从规则学得更准。一个负例 + 一个正例 > 三段抽象描述。

6. 把"踩过的坑"写进去

最有价值的 system prompt 内容, 是**只在你这个项目才有的反直觉知识**:

```
## 踩过的坑

- Postgres 的 timezone 默认 UTC,但 API 必须返回用户时区
  → 用 utils/tz.go 的 ConvertToUserTZ,不要手动 .In()

- Redis 的 key 必须带 namespace 前缀
  → 用 cache.Key(...) 构建,不要直接拼字符串
  → 反例:redis.Get("user:42") ✗
  → 正例:redis.Get(cache.Key("user", 42)) ✓
```

Agent 一次中招, 你就把它写进 system prompt——下次永不再犯。

写好 user prompt 的 4 个原则

1. "上下文 + 任务 + 约束 + 验收"四件套

```
[上下文] 我们在做用户注销功能,spec 在 specs/003-data-export.md。
       Phase 1 已经做完(创建请求 API)。

[任务] 现在做 Phase 2:30 天后台任务清理过期请求。
```

```
[约束] 用 cron 不用 worker。任务要幂等。
       不要改 Phase 1 已经定型的 API。

[验收] 跑 make test 全绿。手动触发一次清理,DB 里
     看到 status='completed' 的记录。
```

四件套不一定每个都写满, 但**至少写"任务 + 验收"**。验收是最常被忘的, 而它恰恰是 Agent 知道"做完没"的唯一依据。

2. 优先用引用, 不要复制粘贴

```
✗ 我:[粘贴 200 行的 spec 内容] 按这个做。
✓ 我:按 specs/003-data-export.md 的 Phase 2 部分做。
     重点关注其中的"幂等性"段。
```

Agent 自己读文件, 你的对话上下文不被吃。

3. 把"不要做什么"说在前面

新手只说要做什么,Agent 经常顺手"帮你"做了你不想要的事:

```
✗ 我:加一个 deleteAccount 方法。
   [Agent 顺手把整个 UserService 重构了一遍]

✓ 我:加一个 deleteAccount 方法。
   - 不要重构现有代码
   - 不要改其他方法的签名
   - 只新增,不修改
```

4. 一次只问一件事

```
✗ 我:加 deleteAccount,顺便看看 createAccount 有没有 bug,
     再帮我写下文档,对了 README 也更新一下。

✓ 我:加 deleteAccount。其他事情我们后面单独说。
```

一次塞 4 件事,Agent 会做 2 件、漏 1 件、做错 1 件。**一次一件, 做完再加**。

System 和 User 的协作: 让两边各司其职

最高效的协作长这样:

```
System prompt(写在 AGENTS.md / Skill 里,稳定):
- 项目是 Go 后端
- 错误处理用 fmt.Errorf 包装
- 改 migrations 前先问
- 输出按"理解 → 计划 → 确认 → 影响"四段

User prompt(每次具体说):
- "实现 specs/003 的 Phase 2,验收是 make test 全绿"
```

User prompt 简短, 因为所有"通识"都在 system 里。**当你发现自己每个 user prompt 都要重复同样的话, 这些话就该挪到 system prompt 里**。

反向心法: 从 user prompt 反推 system prompt

最实用的迭代方法:

1. 平时正常用 Agent
2. 每次你打字时, **注意有哪些话是"又一次"在说**
3. 这些话就是 system prompt 缺的内容, 补进去

例子:

```
[第 1 次] 我:加个 endpoint,记得用 Result<T,E> 不要 throw
[第 5 次] 我:写个函数,Result<T,E> 不要 throw
[第 8 次] 我:。。。Result<T,E>
       ↑ 该把这条挪进 AGENTS.md 了
```

反模式

反模式	后果	怎么改
System prompt 写小说	关键规则被淹没	用编号 + 短句,200-400 行内
每次 user prompt 都重复项目背景	浪费 token + 容易漏	项目背景挪到 system
System prompt 写"任务"	所有 session 都被这个任务污染	任务永远放 user
User prompt 写"角色定义"	每次都得复制粘贴	角色定义放 system
只有"要做什么", 没有"不要做什么"	Agent 自由发挥越界	边界要明确写
System prompt 不更新	同样的坑反复踩	每次中招就回去补
一个 user prompt 塞 5 个任务	漏做、做错	一次一件
没有验收标准	Agent 不知何时算"完成"	每个 user prompt 带验收

> **核心心法:system prompt 是 Agent 的"重力",user prompt 是 Agent 的"今天去哪儿"**。重力定了,Agent 自然不会乱飘; 每次只用说目的地。如果你发现自己每次都在说"地球是圆的"——那是 system prompt 没写好。

十二、CI/CD: 让 Agent 在你睡觉时干活

CI/CD = **Continuous Integration / Continuous Deployment**(持续集成/持续部署)。简单说: **每次有代码变动, 自动跑一系列检查和操作**——跑测试、跑 lint、构建、部署。常见实现:GitHub Actions、GitLab CI、CircleCI 等。

为什么 Vibe Coding 必须聊这个? 因为 **Agent 写的代码量是手写的 N 倍**, 你 review 速度跟不上;CI 是你睡觉时的"第二审查官", 自动把不合格的代码挡在合并之前。**没有 CI,vibe coding 等于把火药桶交给会喷火的实习生**。

1. 最基础的 CI: 一份 `.github/workflows/ci.yml`

GitHub Actions 是免费 (公开仓库) 且开箱即用的。一份最小可用的 CI 配置长这样:

```yaml
# .github/workflows/ci.yml
name: CI

on:
  push:
    branches: [main]
  pull_request:
```

```yaml
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-node@v4
        with:
          node-version: '20'

      - run: npm ci
      - run: npm run lint
      - run: npm test
      - run: npm run build
```

逐行解释:

- `on: push / pull_request` → 什么时候触发 (push 到 main 或开 PR 时)
- `jobs.test` → 一个叫 test 的任务
- `runs-on` → 在 GitHub 提供的 Ubuntu 机器上跑
- `steps` → 按顺序跑这些命令, 任何一步失败, 整个 job 失败

放进 `.github/workflows/ci.yml` 提交,GitHub 自动识别并开始跑——你 push 代码后去仓库的 Actions 页签就能看结果。

2. CI 在 Vibe Coding 里要拦住什么

CI 是"机器审查官", 它看不懂业务, 但能死死盯住格式问题。**你要让它替你拦住**:

检查项	工具示例	Agent 经常犯什么错
测试	jest / pytest / go test	写出"看起来对、实际坏"的代码
Lint	eslint / ruff / golangci-lint	风格不一致、危险模式 (unused var)
类型检查	tsc / mypy	类型断言乱用、any 满天飞
格式化	prettier / black / gofmt	格式跟项目不一致
依赖审计	npm audit / pip-audit	引入有漏洞的依赖
密钥扫描	gitleaks / trufflehog	把 API key 写进代码
构建	webpack / tsc / go build	改坏了别处导致编译失败
覆盖率	codecov	加代码不加测试

关键观察:Agent 最容易栽的是**前 4 项** (测试、lint、类型、格式)。这些恰好是 CI 最擅长查的。**所以 CI 是 Agent 的天敌, 也是你最强的盟友**。

3. AGENTS.md 必须写的两条 CI 相关规则

让 Agent 主动配合 CI, 而不是"先提交了 CI 挂了再修"——这两条必写进 AGENTS.md:

```
## CI 规则
```

```
1. 提交前必须本地跑过 `make ci-local`(或 `npm run check`), 确保通过
   - 这等价于 CI 上跑的所有检查
   - 在本地 30 秒内能跑完, 比等 CI 5 分钟快得多

2. 如果 CI 挂了:
   - 先看哪一步挂的, 贴出失败日志
   - 不要" 猜测式修复"——先复现再修
   - 修完之后本地跑一次 ci-local 再 push
```

配套的: 在项目里加一个 make ci-local 或 npm run check, **让本地命令和 CI 跑的完全一致**:

```
# Makefile
ci-local: lint test build
    @echo "✓ 本地 CI 通过"

lint:
    npm run lint

test:
    npm test

build:
    npm run build
```

有了这个本地命令,Agent 就有了"自检手段"——它做完改动后能自己跑一遍, 不用等远端 CI。

4. 让 Agent 帮你写 CI

CI 配置自己写很烦, **让 Agent 写**——但要给它足够的上下文:

```
我:为这个项目写一个 GitHub Actions CI 配置。

   项目情况:
   - Node.js 20,用 pnpm 管理依赖
   - 有 jest 测试、eslint、prettier、tsc
   - main 分支保护,所有合并必须 CI 绿
   - 想要 PR 上自动评论测试覆盖率

   要求:
   - 用 actions/cache 缓存 node_modules
   - 矩阵跑 Node 18 和 20
   - 失败时显示有用的错误信息
   - 给我一份 .github/workflows/ci.yml
```

写完之后, **别直接相信能跑**。CI 的反馈循环很慢 (每次 push 等几分钟), 所以:

- 第一次 push 之前, 先用 act 在本地模拟跑一下
- 或者把 CI 配置先弄一个最小可跑版本, 在 PR 里迭代调通, 再合进 main

5. CI 里跑 Agent: 让 AI 帮你审 PR

进阶玩法: **在 CI 里调 Claude / GPT 帮你做事**。常见用法:

- **AI Code Review**: 每个 PR 自动让 Claude 看一遍, 指出问题
- **AI Changelog**: 从 commit 自动生成 release notes
- **AI Triage**: 新 issue 自动打标签、分配人

GitHub 官方有 `anthropics/claude-code-action`, 在 PR 上 @claude 就能让 Claude 审代码或改代码。基本配置:

```yaml
# .github/workflows/claude-review.yml
name: Claude Review

on:
  pull_request:
    types: [opened, synchronize]
  issue_comment:
    types: [created]

jobs:
  claude-review:
    if: contains(github.event.comment.body, '@claude') || github.event_name == 'pull_reque
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: anthropics/claude-code-action@v1
        with:
          anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}
```

需要在 GitHub 仓库的 Settings → Secrets 里加上 `ANTHROPIC_API_KEY`。

6. CI 场景里 system prompt 的特殊性

回到上一章讲的 system prompt——**CI 里跑的 Agent 是完全非交互式的**。这意味着:

- 没人能现场打断它
- 没人能回答它的反问
- 它跑完就生成最终结果 (commit、comment、报告)

所以 CI Agent 的 system prompt 要特别强调:

```
你在 CI 环境运行,无法和用户交互。你必须遵守:

1. 遇到歧义时不要瞎猜:
   - 直接 fail 这个 job
   - 在 PR comment 里写明"我无法判断 X,需要人工确认"

2. 永远不要直接 push 到 main:
   - 你的改动只能写到当前 PR 分支
   - 如果当前不在 PR 分支,fail

3. 限制改动范围:
   - 一次最多改 5 个文件
   - 超过这个范围就 fail 并请求拆分

4. 输出必须结构化:
```

- 用 GitHub Markdown 格式
- 重要结论放最前面
- 引用具体的文件路径和行号

核心原则:CI Agent 没有"我去问问用户"的退路。所以**要么它有十足把握做对, 要么它必须明确 fail 并解释原因。模糊的成功比明确的失败更糟**——后者你能修, 前者你可能永远发现不了。

7. 部署 (CD) 的特殊考量

CD = 自动部署。Agent 写的代码自动上线? **慎之又慎**。

推荐的多级保护:

```
开发分支 push → CI 跑测试(自动)
                ↓ 通过
PR 创建 → CI + AI Review + 人工 review(强制)
                ↓ approve & 合并
main 分支 → 部署到 staging(自动)
                ↓ 烟雾测试通过
                ↓ 人工点按钮
生产环境 ← (手动触发)
```

红线: 永远不要让 AI 直接触发生产部署。哪怕你 100% 信任它的代码, 也要保留一个"人按按钮"的环节——这个按钮不是为了审查代码, 是为了**给你一个机会终止** (线上有事故、客户在投诉、你刚发现别的 bug)。

8. CI 反馈循环的 Vibe Coding 优化

CI 跑一次 5 分钟, 你一天等 10 次 = 浪费 50 分钟。优化思路:

- **本地预运行**: 每个 PR 推之前, 先在本地跑 `make ci-local`, 挡住 80% 的失败
- **快慢分离**:CI 分两份——快的 (lint + 单元测试,2 分钟) 阻塞合并; 慢的 (集成测试、e2e,15 分钟) 异步跑
- **失败立即告知**:CI 挂了 → Slack/邮件/Discord 推送, 而不是去仓库刷
- **缓存依赖**:`actions/cache` 缓存 node_modules、pip cache, 首次 3 分钟变后续 30 秒
- **并行化**: 多个 job 用 `needs:` 控制依赖, 无依赖的并行

9. Agent + CI 的最佳工作流

把所有概念串起来:

```
1. Agent 写代码(在 worktree 或主分支)
   ↓
2. Agent 本地跑 make ci-local
   ↓ 通过
3. Agent 提交 commit、推送到 PR 分支
   ↓
4. CI 自动触发(测试 + lint + 类型 + 构建)
   ↓ 通过
5. claude-code-action 在 PR 上 review
   ↓ 给出意见
6. 人审 PR(看 AI review 已经过滤过的内容,效率高 5 倍)
   ↓ approve
7. 合并到 main
   ↓
```

```
8. CD 自动部署到 staging
   ↓ 烟雾测试通过
9. 人手动触发生产部署
```

每一步都是一道关。Agent 可能在某一步犯错——但下一步会拦住它。这就是为什么 CI/CD 不是"额外工作", 而是 vibe coding **必备的安全网**。

CI/CD 的反模式

反模式	后果	怎么改
没有 CI, 人肉跑测试	Agent 提交一堆坏代码	立刻配最小 CI
本地命令和 CI 不一致	"本地能跑,CI 挂了"	用 Makefile 统一入口
CI 太慢 (>10 分钟)	没人愿意等, 直接合	拆快慢、加缓存
让 AI 直接部署生产	一次失误全公司加班	永远留一个人按的按钮
CI 挂了没人管	主分支永远是红的	Slack 告警 + 必须修才能继续
把密钥写进 yml	公开仓库泄漏	用 GitHub Secrets
Agent 改 CI 配置不告知	偷偷绕过检查	AGENTS.md 写明: 改.github/ 必须问

> **核心心法:CI 不是"质检", 是 Vibe Coding 的免疫系统**。Agent 写得越多, 你越需要 CI。一个项目没有 CI 就让 Agent 大量产出代码, 等于免疫缺陷的人不戴口罩——前期看起来没事, 出事就是大事。

十三、测试: 让 Agent 写代码容易, 让 Agent 写出"对的"代码难

Agent 写出**结构合理但行为错误**的代码是常态。测试是唯一可靠的护栏。但"测试"在 Vibe Coding 里有两个层次, 要分开讲:

- **A. 普通代码的测试**:Agent 实现一个函数, 你怎么验证它对
- **B. Agent 行为本身的测试**: 你在做一个 AI 应用 (或者 prompt、Skill、subagent), 怎么验证它的行为符合预期

A 是基本功,B 是 Vibe Coding 才有的新问题——绝大多数文档不讲。

A. 普通代码的测试: 三种姿势

姿势 1: 你写测试,Agent 实现 (TDD 反向)

```
我: 这是我要的函数签名和测试用例, 你来实现:

   func MergeIntervals(intervals [][]int) [][]int

   测试:
   - 空数组 → 空数组
   - [[1,3],[2,6],[8,10]] → [[1,6],[8,10]]
   - 已经合并的 [[1,2],[3,4]] → 不变
   - 完全包含 [[1,10],[2,3]] → [[1,10]]

   你的任务: 实现 + 把这些测试写成代码 + 跑通。
```

优点: 你掌握"对错"的定义,Agent 没法绕过 **缺点**: 你得想清楚边界 case, 前期投入大

姿势 2:Agent 写测试 + 实现, 你审测试

```
我：实现 X 功能。先写测试，把覆盖的场景列给我审，
    我同意了你再写实现。
```

审测试比审实现容易得多——测试是"输入 → 输出"的契约, 你扫一眼就知道覆盖够不够。

关键: 永远不要让 Agent 实现完之后再补测试。补的测试会刚好覆盖它写的实现,bug 一起被掩盖。

姿势 3: 边界 case 让 Agent 自己列

```
我：这是我的实现。在写测试之前，先列出所有可能的边界 case
    和异常输入。我看完再决定哪些要测、哪些不重要。
```

Agent 列 case 的能力很强, 但选择"哪些值得测"是工程判断, 你来做。

A 的反模式

反模式	后果
让 Agent 看着实现写测试	测试和 bug 一起被绿
测试只测 happy path	你以为很安全, 生产爆
测试名字含糊 (“test1”)	半年后看不懂在测什么
Agent 改测试让它通过	在测一个错的契约
不跑测试就信"我跑过了"	Agent 经常撒谎, **自己看输出**

最后一条特别重要: **永远不要相信 Agent 说"测试通过了"**。让它把命令输出贴给你看, 或者你自己跑。

B. Agent 行为本身的测试 (关键)

如果你在用 Vibe Coding 做的事情**本身就是一个 AI 应用**——比如你在调一个 prompt、做一个 Claude Skill、写一个 subagent、调一个 agent loop——那你测的就不是函数返回值, 而是 **Agent 的行为**。

这种测试更难, 因为:

- 输出非确定: 同一个 prompt 跑两次结果可能不同
- 没有"正确答案": 只有"更好/更差"的输出
- 失败模式多样: 工具调用错了、忘记某一步、把任务理解偏了、输出格式不对……

但有一套行之有效的方法。

核心方法:Case 驱动 + 行为证据

工作流分四步:

```
1. 准备 case(具体的 prompt + 期望行为)
2. 实跑 2-3 次,人工观察
3. 收集行为证据(TUI 截屏、数据库状态、日志、工具调用序列)
4. 把"prompt + 期望 + 证据"打包给 AI,让它分析/修复
```

为什么要跑 2-3 次? 因为 Agent 行为有随机性。一次成功可能是运气, 一次失败可能是抖动。**2-3 次是判断"稳定性"的最低样本量**。

步骤详解

第 1 步: 把 case 写成一份文档

不要用聊天里随手写的 prompt 测, 要做成可复现的 case 文件:

```
# Case: 用户问" 我上个月花了多少"

## 输入 prompt
" 我上个月一共花了多少钱? 分类列一下。"

## 上下文条件
- 用户已登录,user_id=42
- DB 里有 user_id=42 的 30 条交易记录, 跨 2024-08 和 2024-09
- 当前日期假定为 2024-09-15

## 期望行为
1. Agent 调用 `get_transactions` 工具, 参数 user_id=42,
   start=2024-08-01, end=2024-08-31
2. Agent ** 不 ** 调用其他工具 (尤其不要调 `get_user_profile`)
3. 回复中:
   - 总金额正确 (等于 DB 里 8 月份的 sum)
   - 按 category 分组
   - 用人民币格式 ("¥1,234.56")
   - 不超过 5 行

## 失败信号
- 调了多余的工具
- 时间范围错 (把 9 月也算进去)
- 数字算错
- 用了英文 / 用了 $ 符号
```

这份文档的价值: **它是你的"真值"**。每次回归测试都对照这份。

第 2 步: 在 Claude CLI(或对应工具) 里实跑

跑 2-3 次, 每次都完整保留:

- 终端输出 (TUI)
- Agent 调了哪些工具、参数是什么、返回是什么
- 数据库的状态变化 (如果有写操作)
- 最终回复

记录方式建议:

```
# 用 script / asciinema 录终端
asciinema rec runs/case01-run1.cast
# 或者最简单: 开 tmux, 事后把 buffer 存下来

# 数据库快照
pg_dump dev_db > runs/case01-run1-db-before.sql
# ... 跑 case...
pg_dump dev_db > runs/case01-run1-db-after.sql
diff runs/case01-run1-db-before.sql runs/case01-run1-db-after.sql > runs/case01-run1-db-di
```

3 次跑完, 整理成:

```
runs/case01/
├── case.md                # 上面那份 case 文档
├── run1-tui.txt           # 第一次跑的终端输出
├── run1-tools.json        # 工具调用序列
├── run1-db-diff.txt       # 数据库变化
├── run2-...
├── run3-...
└── observation.md         # 你的人工观察笔记
```

第 3 步: 人工观察, 写下"差距"

你看完 3 次跑的结果, 人工总结:

```
# observation.md

## 行为稳定性
- 3 次都正确调了 get_transactions ✓
- 时间范围:run1/run2 正确,run3 把 start 写成了 2024-08-15(错)
- run2 多调了一次 get_user_profile(冗余但没造成错误)

## 输出质量
- 3 次的总金额都正确
- run1、run3 用了 ¥ 符号,run2 用了 RMB(不一致)
- 行数都在 5 行以内 ✓

## 主要问题
1. 时间范围有时候出错 (33% 失败率)→ 严重
2. 货币符号不稳定 → 中等
3. 偶尔调多余工具 → 轻微
```

这一步**必须人工做**, 不要让 AI 帮你看自己的输出——它会自我合理化。

第 4 步: 把"证据包"喂给 AI 调试

现在你有了: - case 文档 (期望) - 3 次的实际行为证据 - 你的观察笔记 (差距)

把这些一起喂给 AI 来定位问题:

```
我:这是一个 prompt 调试任务。

   [贴 case.md]
   [贴 observation.md]
   [贴 3 次的 run*-tools.json]
   [贴当前的 system prompt]

   主要问题是:时间范围有 33% 概率出错。
   分析这是 prompt 哪一部分导致的,给出 2-3 个修改方案。
   先不要改,告诉我你的假设。
```

关键技巧:

- **给具体证据, 不要让 AI 猜**:"它有时候会出错"是没用的输入,"3 次中第 3 次把 start 写成了 2024-08-15"是有用的输入

- **让 AI 先给假设, 再改**: 直接让它改 prompt, 它会乱改; 先让它说"我猜是因为 X", 你判断假设合理再改
- **改了之后重跑同样的 case**: 回到第 2 步, 看修改后的 3 次行为有没有改善

这套方法的本质

把不确定的 Agent 行为, 通过"多次采样 + 证据归档"变成可分析的对象。

这和测试普通代码的差别:

	普通代码测试	Agent 行为测试
真值	函数返回值	Case 文档 (人定义的期望)
一次跑够吗	够	至少 2-3 次, 看稳定性
通过/失败	二元	频次 (成功率 / 偏离程度)
调试输入	报错堆栈	TUI + 工具序列 + DB diff + 你的观察
验收	测试绿	"够稳了"(主观, 但靠证据)

进阶:Case 库

当你有 5 个以上 case 之后, 把它们组织成一个库:

```
cases/
├── 01-monthly-spending/
├── 02-cancel-subscription/
├── 03-edge-no-data/
├── 04-malicious-prompt-injection/
├── 05-very-long-history/
└── runner.sh          # 一键跑所有 case
```

每次改 prompt / 系统配置之后, 跑全库, 看哪些 case 退化、哪些改善。这就是 AI 应用的**回归测试**。

不需要花哨的 eval 框架——一个 shell 脚本 + 几个 markdown case + 你的眼睛, 就能跑出比绝大多数团队更靠谱的迭代节奏。

B 的反模式

反模式	后果	怎么改
"我跑了一次, 看起来挺好"	抖动让你以为修好了	至少跑 3 次
没有 case 文档, 凭印象	调了 3 天发现你忘了原来的期望	case.md 写下来
让 AI 自评自己的输出	自我合理化	你亲自观察
只看最终回复, 不看工具序列	表面对了, 内部步骤错了	收集完整证据
改 prompt 不重跑旧 case	修了 A, 坏了 B	跑回归
用模糊语言描述问题	AI 修不准	给具体证据
Case 太"理想", 不带边界	真实场景全炸	加恶意 / 极端 / 残缺输入

> **核心心法**:Agent 行为是统计性的, 你的测试也必须是统计性的。**单次成功不是成功,3 次稳定才是成功**。证据 (TUI、工具序列、DB 状态) 是 Agent 行为的"病历", 有病历才能治病。

十四、几个高阶心法

1. “让 Agent 先说”原则

任何重要任务, 第一步永远是让 Agent 复述/规划, 不要让它直接动手:

```
✗ 给我加一个用户搜索 API
✓ 我想加一个用户搜索 API。先告诉我:
  - 你打算怎么实现
  - 会改哪些文件
  - 有哪些边界 case
  - 你需要我决定什么
```

第二种方式贵 30 秒, 但避免你 30 分钟后发现它做错了方向。

2. 频繁 commit, 把 git 当 ctrl-z

Agent 协作最舒服的姿势是**每个小步骤都 commit**。出问题就 `git reset`, 不心疼。

```
我:每次你完成一个独立的修改,就帮我 commit,
   message 格式 "feat(module): xxx"。
```

3. 拒绝”看起来对”的代码

Agent 会写出**结构正确但逻辑错误**的代码。看起来很专业、命名很合理、注释很到位——但跑起来是坏的。

对抗这个的唯一办法是: **测试驱动**。要么你写测试 Agent 实现, 要么 Agent 写测试你审, 然后跑给你看绿。

4. 把”知识”沉淀成文件

每次 Agent 帮你解决了一个非平凡的问题, 问自己:“这个知识下次的我/Agent 会需要吗?”

如果是, 写进 `docs/`、`AGENTS.md`、或者代码注释。否则下个 session 你又要把同样的事讲一遍。

5. 学会”喊停”

Agent 跑偏的时候, **立刻打断**, 不要让它继续。“算了, 我们重来”是最有用的命令之一。

继续让一个走错方向的 session 跑下去, 只会浪费上下文、产生需要回滚的代码、并强化错误的心智模型。

十五、一个完整的工作流示例

把上面所有概念串起来, 一个真实的功能开发可能长这样:

```
Day 1 上午
[新 session]
我:接下来要做"用户导出数据"功能。先读 AGENTS.md 和 specs/template.md,
   然后帮我起草 specs/003-data-export.md,问我所有不清楚的问题。

[Agent 反问 8 个问题,我回答]
[Agent 写出 spec,我审改,commit]

Day 1 下午
```

```
[新 session,因为上下文已经被探索阶段塞满]
我:读 specs/003-data-export.md。这是要做的事。先给我设计方案,
   不要写代码。涉及哪些表?改哪些文件?异步还是同步?

[Agent 给方案,我反复磨,最终确定]
[让 Agent 把方案写进 docs/design/003-data-export.md]

Day 2
[新 session]
我:读 spec 和 design 文档。开始实现 Phase 1(创建导出请求 API)。
   每完成一个文件就给我看 diff,我确认了再写下一个。

[实现过程中遇到一个复杂的查询性能问题]
我:派一个 subagent 去研究 Postgres 大表导出的最佳实践,
   给出 3 个方案对比。

[Subagent 返回结果,主 Agent 基于此继续]

[Phase 1 完成,跑测试,commit]
我:把这个 session 的进度写进 docs/notes/2024-XX-XX-export.md,
   包括遗留 TODO。然后我们结束。

Day 3
[新 session]
我:读 spec、design、上次的 notes。继续 Phase 2。
...
```

每个 session 都是独立、清爽、目标明确的。文件是 Agent 的长期记忆, 你是 Agent 的指挥官。

十六、常见反模式速查

反模式	后果	怎么改
没 spec 直接让 Agent 写	方向偏、反复改	先 spec, 再代码
一个 session 用 8 小时	后期质量崩盘	早压缩、早换 session
AGENTS.md 不维护	同样的坑反复踩	每周 review 一次
接手项目立刻改代码	风格不一致	先考古、先跑环境
跑完 /init 就完事	初版只是骨架, 缺隐性知识	/init 后必须手动补 + 考古
让 Agent 自评自己的代码	自洽幻觉	用另一个 Agent / 角色审
复杂任务不拆	Agent 做错 + 你审不动	拆 phase, 每 phase commit
不写测试就信代码	看起来对、跑起来错	测试驱动
把上下文当记忆	跨 session 失忆	沉淀到文件
多 Agent 同目录跑	文件互相破坏	用 worktree 隔离
worktree 里没装依赖	Agent 一上来就报错	写 setup-worktree.sh
上下文爆了硬聊	Agent 边犯错边劣化	立刻写 handoff, 开新 session
大文件直接粘贴进对话	上下文一次烧掉一半	让 Agent 自己 grep
同一类任务讲 5 次	重复劳动、约定不一致	做成 Skill
Skill 写得像散文	Agent 抓不住要点	编号步骤 + 明确路径
Agent 实现完才补测试	测试和 bug 一起绿	先写测试 / 先审测试 case

反模式	后果	怎么改
信 Agent 说”测试过了”	它经常撒谎	自己看输出
测 Agent 行为只跑一次	抖动让你误判	至少跑 3 次看稳定性
调 prompt 凭感觉, 没 case	修了又坏	写 case.md + 收集证据
看到方向错还硬聊	越改越乱	立刻喊停,git reset 重来
每次 user prompt 重复项目背景	浪费上下文 + 容易漏	把通识挪进 system prompt
System prompt 写当下任务	所有 session 都被污染	任务永远放 user prompt
不写.gitignore	仓库塞进 node_modules / 泄漏.env	用 gitignore.io 生成
commit 过敏感信息再加 ignore	历史里还在, 会被爬虫扫到	立刻轮换密钥 + 重写历史
没有 CI,Agent 自由提交	坏代码涌入主分支	配最小 GitHub Actions
本地命令和 CI 跑的不一样	“本地能跑 CI 挂了”	Makefile 统一入口
AI 直接部署到生产	一次失误全公司加班	留一个人按的按钮
能用 workflow 偏要用 agent	不可控、贵、难调试	先尝试 workflow 模式
让 Agent 评自己写的代码	自我合理化	用 Adversarial / 不同角色
多 Agent 互相 chat 协作	上下文爆、绕圈	通过文件交换信息
Evaluator-Optimizer 没退出条件	死循环烧钱	max_iter + 评分阈值

结语

Vibe Coding 的核心不是”放飞自我让 AI 干活”, 而是把自己从字符级的劳动中解放出来, 升级为 Agent 团队的**指挥官 + 架构师 + 审查官**。

你写的每一份 spec、每一行 AGENTS.md、每一个文件夹结构, 都是在为 Agent 搭建脚手架。脚手架越好,Agent 越聪明, 你越省力。

Happy vibing.