匠人学院
JR ACADEMY

2026 实操版

终端里的 AI 结对程序员

CLAUDE CODE 中文上手手册

从装好到真正会用：多文件重构、Git 工作流自动化、Hooks 质量门控、MCP 接数据库和浏览器、Worktree 并行开发。8 章全是可复制的命令和真实 diff，照着敲就能跑。

- 5 分钟装好跑通
- 一句话改 10 个文件
- 多个 Claude 并行干活

手动半小时的活，它 2 分钟搞定

jiangren.com.au · 在线版随官网更新 /wiki/claude-code-guide

AI 来匠人 ✦ CLAUDE CODE ✦ JIANGREN.COM.AU ✦ 终端里的 AI 程序员 ✦ 学 AI 来

CONTENTS

目录（8 章）

📖 这本书怎么读

这本书和官网 wiki《Claude Code 完全指南》同源（jiangren.com.au/wiki/claude-code-guide，免费、不用注册，官网那份会持续更新）。还没装的，第 1 章 5 分钟搞定；已经在用但只会"问问题"的，第 2、3 章是它和 ChatGPT 拉开差距的地方；想让它自动守住代码质量，看第 4 章 Hooks；想让它查数据库、开浏览器、操作 GitHub，看第 5 章 MCP；最后第 8 章 Worktree 是进阶玩法 —— 同时开几个 Claude 各干各的活。每章的命令和配置都能直接复制。

💬 装的时候卡住了？扫码来问

Amelia 会拉你进群，群里每周还有 AI 工具实操分享。

CHAPTER 01

🚀 安装与第一个命令

安装

三种安装方式，选一个就行：

```
# macOS / Linux（推荐，自动更新）
curl -fsSL https://claude.ai/install.sh | bash

# Homebrew（不自动更新，需手动 brew upgrade）
brew install --cask claude-code

# Windows PowerShell（需要管理员权限打开终端）
irm https://claude.ai/install.ps1 | iex
```

系统要求：Node.js 18+（macOS/Linux 安装脚本会自动检测；Windows 需提前手动安装 Node）。

Windows 用户还需要先装 Git for Windows，否则启动时报 `requires git-bash` 错误。装完之后确认 `git` 在 PATH 里：

```
git --version   # 应输出 git version 2.x.x
claude --version  # 应输出 claude 1.x.x
```

Linux 上如果 `curl` 脚本因权限报错，可以加 `sudo`，或者用 npm 全局安装：

```
npm install -g @anthropic-ai/claude-code
```

登录

第一次启动会弹浏览器让你登录。支持这些账号：

账号类型	适用场景
Claude Pro / Max / Team	个人或团队日常开发
Claude Console（API）	按量付费，适合 CI/CD
Amazon Bedrock / Google Vertex / Azure Foundry	企业级云部署

登录后凭证缓存在本地，不用每次重新登录。想切换账号用 `/login`。

在无 GUI 的服务器（SSH 远程）上，启动时选 "print auth URL"，把 URL 复制到本地浏览器完成授权，授权码回填到终端即可。

第一个任务

进入项目目录启动：

```
cd my-project
claude
```

看懂陌生项目：第一次进陌生代码库，先让它给你定向：

```
> 这个项目是做什么的？用了哪些技术栈？主要目录结构是怎样的？
```

Claude Code 会自动读 `package.json`、`README`、源码目录，给出清晰的全局概览，比自己翻文件快得多。

代码分析：

```
> 帮我找一下所有没有错误处理的 async 函数，列出文件路径和行号
```

它会用 Grep 搜索代码、逐文件读取、分析控制流，汇报哪些函数缺少 try/catch 或 `.catch()`，同时说明为什么认为有风险。

修改代码：

```
> 把 src/utils/format.ts 里的 formatDate 函数改成支持传入 timezone 参数，默认 'Australia/Sydney'
```

改完直接写回文件，终端显示 diff。可以继续让它跑测试验证：

```
> 运行相关测试，看有没有因为这个改动挂掉
```

Git 工作流：

```
> 帮我写一个 commit message，描述刚才所有的改动
> 然后 commit，branch 名叫 feat/format-timezone
```

整个流程——读代码、改代码、跑测试、提交——全在终端里完成，不需要切窗口复制粘贴。

常用 Keybinding 速查

快捷键	作用
`Esc`	中断当前 AI 响应（不中断已在跑的 shell 命令）
`Ctrl+C`	强制退出（包括终止正在执行的命令）
`Ctrl+L`	清屏（保留上下文历史，不影响对话）
`↑ / ↓`	翻历史输入
`Shift+Enter`	多行输入（不提交，继续写下一行）
`/help`	查所有 slash 命令列表
`/clear`	清除当前会话上下文（重新开始）
`/compact`	压缩对话历史（上下文快满时用）
`/model`	切换模型（Opus 4 / Sonnet 4 / Haiku 4）
`/login`	切换账号或重新授权

VS Code 插件里额外多两个常用快捷键：`Cmd+Shift+P` → "Claude Code: Focus" 快速聚焦聊天窗口；`Cmd+I`（macOS）/ `Ctrl+I`（Windows/Linux）打开 inline edit 面板，可以直接针对选中代码下指令。

和 ChatGPT / Cursor 的区别

工具	工作方式	上下文范围	适合场景
ChatGPT	对话框粘贴代码	单次对话	问答、学习概念
Cursor	IDE 内 AI 辅助	当前文件 + 少量上下文	写新代码、补全
Claude Code	终端直接操作文件系统	整个项目	重构、多文件修改、调试、Git 操作

核心差异：Claude Code 直接在你的文件系统上操作。它能一次读几十个文件，理解跨文件依赖关系，执行终端命令验证结果——这是单文件 IDE 辅助做不到的。

多种使用方式

除了终端 CLI，Claude Code 还能在这些地方用：

```
# VS Code 扩展（支持 inline diff 和 @-mention 文件）
code --install-extension anthropic.claude-code

# 桌面应用（macOS / Windows，图形化界面）
# 从 https://claude.com/download 下载

# 网页版（不用安装任何东西）
# 直接打开 https://claude.ai/code
```

VS Code 里按 `Cmd+Shift+P` 搜索 "Claude Code" 就能打开侧边栏。JetBrains 系列也有插件，IntelliJ / PyCharm / WebStorm 均支持，安装方式和 VS Code 类似，在 Marketplace 里搜 "Claude Code" 即可。

CHAPTER 02

🛠️ 多文件编辑：Claude Code 的杀手锏

为什么多文件编辑重要

改一个接口名，可能涉及 controller、service、DTO、测试、前端调用 5+ 个文件。手动改容易漏，Claude Code 一次搞定。

实战：重命名一个 API

```
> 把 getUserList 改名为 fetchUsers，所有引用的地方都改掉
```

Claude Code 的执行步骤：

1. 用 Grep 搜索所有包含 `getUserList` 的文件
2. 分析每处引用的上下文（是函数定义、import、还是调用）
3. 逐个文件用 Edit 工具替换
4. 检查 import 路径、类型定义是否需要联动调整
5. 展示所有修改的 diff，等你确认

你可以按 `y` 逐个确认，也可以切换到 auto-accept 模式（`Shift+Tab` 两次）一次性应用。

实战：批量加错误处理

```
> 检查 src/services/ 下所有 fetch 调用，没有 try-catch 的都加上，
> 错误统一用 toast.error() 提示用户
```

这类跨文件的模式化修改是 Claude Code 的强项。它能识别每个 fetch 调用的上下文，根据已有的错误处理模式生成一致的代码。

手动做要半小时的事情，Claude Code 通常 2 分钟搞定。

实战：类型重构

TypeScript 项目中类型变更是最头疼的多文件修改之一：

```
> User 类型里的 name 字段拆成 firstName 和 lastName，
> 所有用到 user.name 的地方都改成 `${user.firstName} ${user.lastName}`
```

Claude Code 会处理：

- 类型定义文件（`types.ts`）
- API 响应解析（`services/user.ts`）
- 组件 props 传递（`components/UserCard.tsx`）
- 测试文件里的 mock 数据（`__tests__/user.test.ts`）

实战：React 组件改名（5 文件联动）

这是最典型的 React 多文件重构场景：把 `UserCard` 组件改名为 `MemberCard`。看起来简单，但涉及的文件比你想象的多：

1. `src/components/UserCard.tsx` — 组件文件本身（含 interface 名）
2. `src/components/index.ts` — barrel export（重命名后 export 路径变了）
3. `src/pages/Dashboard.tsx` — 引用处 1
4. `src/pages/Profile.tsx` — 引用处 2
5. `src/__tests__/UserCard.test.tsx` — 测试文件（文件名 + 内部引用都要改）

给 Claude Code 一句话：

```
> 把 UserCard 组件改名为 MemberCard，文件也一起重命名，所有 import 和测试都联动改
```

Claude Code 实际执行的每步 diff：

① 组件文件 → 重命名 + 内部 interface

```
// src/components/UserCard.tsx → src/components/MemberCard.tsx

-interface UserCardProps {
+interface MemberCardProps {
   user: User
   onEdit?: () => void
 }

-export function UserCard({ user, onEdit }: UserCardProps) {
+export function MemberCard({ user, onEdit }: MemberCardProps) {
   return (
-    <div data-testid="user-card" className={styles.userCard}>
+    <div data-testid="member-card" className={styles.memberCard}>
       ...
     </div>
   )
 }
```

② barrel export

```
// src/components/index.ts

-export { UserCard } from './UserCard'
+export { MemberCard } from './MemberCard'
 export { Avatar } from './Avatar'
 export { Badge } from './Badge'
```

③ 调用方（Dashboard + Profile）

```
// src/pages/Dashboard.tsx

-import { UserCard } from '@/components'
+import { MemberCard } from '@/components'

-<UserCard user={currentUser} onEdit={handleEdit} />
+<MemberCard user={currentUser} onEdit={handleEdit} />
```

④ 测试文件 → 文件名 + 内部引用 + testid

```
// src/__tests__/UserCard.test.tsx → src/__tests__/MemberCard.test.tsx

-import { UserCard } from '@/components'
+import { MemberCard } from '@/components'

 describe('MemberCard', () => {
   it('renders user info', () => {
-    render(<UserCard user={mockUser} />)
+    render(<MemberCard user={mockUser} />)

-    expect(screen.getByTestId('user-card')).toBeVisible()
+    expect(screen.getByTestId('member-card')).toBeVisible()
   })
 })
```

注意最后一行：`data-testid` 的值也被识别出来需要改。这不是简单的字符串 find-replace，Claude Code 理解了组件和测试之间的语义绑定关系，知道 `"user-card"` 这个 testid 属于这个组件，需要一起重命名。

改完之后在 terminal 跑：

```
npx tsc --noEmit && npx vitest run
```

5 个文件，0 TypeScript 编译错误，测试全绿。手动改大概要 10–15 分钟，而且几乎肯定会漏掉 testid 这种细节；Claude Code 大概 90 秒，不漏。

用 @ 快速引用文件

在提问时用 `@` 引用文件，Claude Code 直接读取内容，不用你手动打开：

```
> @src/types/user.ts 和 @src/services/api.ts 之间的类型定义有没有不一致的地方?
```

引用目录也行：

```
> @src/components 这个目录里有没有没用到的组件?
```

用 CLAUDE.md 控制编辑行为

在项目根目录放一个 `CLAUDE.md`，写上你团队的编码规范：

```
# CLAUDE.md
- 错误处理统一用 try-catch + toast.error()
- API 调用统一放 services/ 目录
- React 组件用 TypeScript + CSS Modules
- 测试用 Vitest + React Testing Library
- import 排序：第三方库 > 内部模块 > 相对路径
```

Claude Code 每次启动都会读这个文件，在做多文件编辑时自动遵守这些规范。团队成员把 `CLAUDE.md` 提交到 git 里，所有人共享同一套规则。

控制修改范围

有时候你只想改一部分文件，可以明确指定范围：

```
> 只改 src/api/ 目录下的文件，把所有 axios 调用换成 fetch
> 不要动测试文件
```

Claude Code 会严格遵守你给的范围限制。如果操作涉及范围外的文件（比如 import 需要调整），它会告诉你而不是直接改。

也可以反过来，排除特定目录：

```
> 全局把 moment.js 换成 dayjs，但 src/legacy/ 目录先别动
```

这种精细化控制在大型 monorepo 里特别有用——你可以逐模块推进重构，而不是一次改动几百个文件让 PR review 成噩梦。

CHAPTER 03

🌿 Git 工作流自动化

自动提交

改完代码直接说：

```
> 提交这些修改
```

Claude Code 会：

1. `git diff` 查看所有改动
2. 分析改动内容，生成语义化的 commit message
3. 只 stage 相关文件（自动跳过 `.env`、`node_modules` 等敏感文件）
4. 展示 commit 预览，等你确认后执行

生成的 commit message 格式规范，比如：

```
feat: add input validation to user registration form

- Add email format check with regex
- Add password strength requirements
- Show inline error messages on invalid input
```

创建 PR：从命令到上线

基本用法

```
> 创建一个 PR，目标分支 main，描述这次改了什么
```

Claude Code 会自动生成 PR title 和 description，包括：

- 改动摘要（从 diff 提炼）
- 影响范围
- 测试说明
- 关联的 issue（如果 commit message 里有 `#123` 引用）

接 GitHub MCP Server 之后

配置好 GitHub MCP Server（见第 5 章），PR 创建变成真正的一步操作：

```
> 我这个功能分支做完了，帮我推到 remote，然后对 main 开 PR，
  标题写「feat: 用户个人资料页改版」，reviewer 加上 @alice @bob
```

Claude Code 完成的步骤：

1. `git push -u origin feature/user-profile`
2. 调用 GitHub API 创建 PR，填写 title 和 body
3. 通过 API 添加 reviewer
4. 返回 PR 链接

整个过程不需要离开终端，也不用在浏览器里填表单。

用模板生成 PR Description

如果项目里有 `.github/pull_request_template.md`，直接告诉 Claude Code 按模板填：

```
> 按 PR 模板创建，测试这块写「单元测试 + 手动测试 iOS Safari 14」，
  风险评估写「涉及登录逻辑，建议先部署到 staging 验证」
```

它会把你说的内容映射到模板对应字段，其余字段从代码改动里自动提取。

PR 创建后的跟踪

有了 GitHub MCP，这些查询直接走 API，不用去浏览器刷页面：

```
> 我的 PR #234 CI 跑了多久？测试有没有过？
> PR #234 有什么 review 意见还没处理？
> PR #234 被 approve 了吗？帮我 merge 掉
```

分支管理

```
# 创建功能分支
> 从 main 创建一个新分支 feature/user-profile，然后切过去

# 查看分支状态
> 当前分支和 main 差了几个 commit？有没有冲突？

# 合并前检查
> 帮我 rebase 到最新的 main，如果有冲突就告诉我
```

实战场景：hotfix

线上出 bug 时，Claude Code 能帮你从排查到修复到提 PR 全流程搞定：

```
> 线上 /api/users 接口返回 500，这是错误日志：
> [粘贴日志]
> 帮我找到原因，修复，然后创建 hotfix PR
```

Claude Code 的执行流程：

1. 分析错误日志，定位到具体的代码文件和行号
2. 读取相关代码，理解上下文
3. 找到 root cause 并修复

4. 跑测试确认修复有效

5. 创建 hotfix 分支、提交、推送、创建 PR

从发现 bug 到 PR ready，通常 5 分钟内完成。

解决 Merge Conflict

快速解决

```
> 帮我解决当前的 merge conflict
```

Claude Code 会读取冲突文件，理解两边的改动意图，选择合理的合并策略。对于复杂冲突，它会解释每处冲突的取舍理由，让你做最终决定。

详细工作流

实际的 merge conflict 处理分三步走：

第一步：搞清楚冲突来源

```
> git status 看一下，哪些文件有冲突？每个冲突大概是什么类型的改动？
```

Claude Code 会逐一分析冲突标记（`<<<<<<<` / `=======` / `>>>>>>>`），判断是：

- 同一行被两边都改了（真冲突，需要人工决策）
- 两边改了不同函数（可以安全合并）
- 一边删文件、另一边修改了同一文件（最复杂，需要理解业务意图）

第二步：解决每个冲突

对于简单情况，让它直接解决：

```
> src/components/Header.tsx 那个冲突，用 main 分支的版本
> src/utils/api.ts 那里两边的修改都要保留，帮我合并
```

对于复杂冲突，让它解释后你来决定：

```
> UserService.java 有三处冲突，帮我逐一分析，告诉我每处用哪边更合理，
  但别直接改，等我确认
```

第三步：验证合并结果

```
> 冲突都解决了，帮我跑一下测试，看看有没有引入新问题
```

常见陷阱

package-lock.json / yarn.lock 冲突：这类文件手工合并必出错。正确做法：

```
> package-lock.json 有冲突，帮我选一个版本 checkout，然后重新 npm install 生成
```

Claude Code 会 checkout 某一边的版本，再跑 `npm install` 重新生成 lockfile，结果一定是正确的。

Rebase 中途冲突：rebase 过程中遇到冲突会停下来，可以这样接着处理：

```
# rebase 停在某个 commit，告诉 Claude Code
> rebase 暂停了，当前冲突是什么情况？帮我解决，然后 git rebase --continue
```

Git Worktree 配合 Claude Code

为什么需要 worktree

开发时常见场景：你在 `feature/checkout-v2` 上改了一半，突然要去修一个生产 bug。常规做法是 `git stash`，切分支，改完再切回来、`git stash pop`。如果两个任务都需要 Claude Code 介入，两个 session 会互相干扰，stash/unstash 也容易出错。

`git worktree` 让你在同一个仓库里有两个独立的工作目录，各自对应不同分支，互不干扰。

基本用法

```
# 给 hotfix 任务开一个独立工作目录
git worktree add ../my-project-hotfix hotfix/payment-bug

# 在新工作目录里启动 Claude Code
cd ../my-project-hotfix
claude
```

两个终端窗口，两个 Claude Code session，分别处理不同任务，git 状态完全隔离。

让 Claude Code 帮你管理 worktree

```
> 用 git worktree 给我创建一个新工作目录处理 hotfix，
  分支名 hotfix/fix-cart-total，目录放在 ../jr-shop-hotfix
```

Claude Code 执行：

```
git worktree add -b hotfix/fix-cart-total ../jr-shop-hotfix main
```

任务完成后清理：

```
> worktree 任务搞完了，帮我合并回 main 然后删掉那个 worktree
```

Claude Code 依次执行：

```
# 提交 worktree 里的改动
git -C ../jr-shop-hotfix add .
git -C ../jr-shop-hotfix commit -m "fix: correct cart total calculation"

# 回到主目录合并
git checkout main
git merge hotfix/fix-cart-total

# 清理
git worktree remove ../jr-shop-hotfix
git branch -d hotfix/fix-cart-total
```

多 agent 并行任务

Claude Code 的 Agent SDK 里，`isolation: "worktree"` 选项会自动为每个子 agent 创建独立 worktree，任务结束后自动清理。这是在 CI / 自动化流水线里跑多个并发 agent 的推荐方式——每个 agent 有自己的文件系统视图，不会踩脚：

```
// 两个 agent 同时工作，互不干扰
const [refactorResult, testResult] = await Promise.all([
  claude.run("重构 UserService", { isolation: "worktree" }),
  claude.run("补充单元测试", { isolation: "worktree" }),
]);
```

详见第 8 章 Agent SDK 的隔离模式。

代码审查

让 Claude Code 在提交前帮你检查代码质量：

```
> review 一下我这次改的所有文件，看有没有问题
```

它会检查：

- 逻辑错误和边界情况
- 类型安全问题
- 潜在的性能问题
- 不符合项目规范的写法

安全机制

Claude Code 对 Git 操作的默认行为很保守：

- 不会自动 push，修改都先到本地
- 不会 force push 或 `reset --hard`
- 不会跳过 pre-commit hook（`--no-verify`）
- 敏感文件（`.env`、credentials）不会被 stage

这些限制确保你不会因为 AI 的一次操作搞坏远程仓库。如果某个操作确实需要 force push，你需要明确告诉它。

💼 用 Claude Code 改完简历项目，想知道离 AI Engineer 岗位还差几步？Amelia 帮你按真实岗位池对一遍：

CHAPTER 04

🪝 Hooks 自动化：让 Claude Code 在对的时机干对的事

什么是 Hooks

Hooks 是 2026 年初发布的功能，让你在 Claude Code 执行特定动作**前后**自动触发自定义脚本。

Claude Code 改了文件？自动跑 lint。
Claude Code 要写某个敏感文件？先问你要不要允许。

配置在 `~/.claude/settings.json`（全局）或 `.claude/settings.json`（项目级）：

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "npx prettier --write $CLAUDE_TOOL_INPUT_FILE_PATH"
          }
        ]
      }
    ]
  }
}
```

两种 Hook 时机

时机	触发点	能否阻止操作	典型用途
PreToolUse	Claude **要执行**某操作之前	✅ 可以	安全门控、保护文件、强制审批
PostToolUse	Claude **执行完**某操作之后	❌ 不能（已完成）	格式化、跑测试、发通知

实战 1：改完代码自动格式化

每次 Claude Code 写文件，自动跑 Prettier：

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "npx prettier --write $CLAUDE_TOOL_INPUT_FILE_PATH 2>/dev/null || true"
          }
        ]
      }
    ]
  }
}
```

`$CLAUDE_TOOL_INPUT_FILE_PATH` 是 Claude Code 自动注入的环境变量，值是本次操作的文件路径。

实战 2：提交前自动跑测试

PreToolUse 拦截 Bash 命令，凡是含 `git commit` 的先跑测试：

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "bash ~/.claude/hooks/pre-commit-check.sh"
          }
        ]
      }
    ]
  }
}
```

`~/.claude/hooks/pre-commit-check.sh`：

```
#!/bin/bash
# 读取 Claude Code 传入的 JSON，判断是否是 git commit 命令
input=$(cat)
command=$(echo "$input" | python3 -c "import json,sys; d=json.load(sys.stdin);
print(d.get('command',''))")

if echo "$command" | grep -q "git commit"; then
  echo "检测到 git commit，先跑测试..."
  npm test
  if [ $? -ne 0 ]; then
    echo "测试失败，阻止提交" >&2
    exit 1  # 非零退出码 = 阻止 Claude Code 继续执行
  fi
fi
```

退出码非零 → PreToolUse Hook 阻止 Claude Code 执行后续操作。这是唯一能"拦截"的时机。

实战 3：保护 .env 文件不被改动

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "python3 -c \"\nimport json, sys\nd = json.load(sys.stdin)\npath =
d.get('file_path', '')\nif '.env' in path and '.env.' not in path:\n    print(f'拒绝修改敏感文件:
{path}', file=sys.stderr)\n    sys.exit(1)\n\""
          }
        ]
      }
    ]
  }
}
```

Hook 脚本收到的 JSON 格式

Claude Code 通过 stdin 传入上下文，你可以读取：

```
import json, sys

data = json.load(sys.stdin)
print(data.keys())
# 常见字段:
# - tool_name: "Write" / "Edit" / "Bash" / ...
# - file_path: 操作的文件路径
# - command: Bash 执行的命令
# - session_id: 当前会话 ID
```

常见坑

坑 1：PostToolUse 里用了退出码 1 试图阻止操作

→ 不行。PostToolUse 已经执行完，退出码只影响 Claude Code 是否报错，不能撤销操作。

坑 2：Hook 脚本没有可执行权限

```
chmod +x ~/.claude/hooks/your-hook.sh
```

坑 3：matcher 写错，Hook 没触发

matcher 是正则表达式，工具名包括：`Write`、`Edit`、`Bash`、`Read`、`Glob`、`Grep`、`Agent`。

用 `"matcher": "Write|Edit"` 匹配写操作，`"matcher": ".*"` 匹配所有工具。

推荐的项目级配置

在项目 `.claude/settings.json` 放这个，团队所有人共享：

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          { "type": "command", "command": "npx prettier --write $CLAUDE_TOOL_INPUT_FILE_PATH
2>/dev/null || true" },
          { "type": "command", "command": "npx eslint --fix $CLAUDE_TOOL_INPUT_FILE_PATH 2>/dev/null
|| true" }
        ]
      }
    ],
    "PreToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          { "type": "command", "command": "node .claude/hooks/protect-sensitive-files.js" }
        ]
      }
    ]
  }
}
```

一旦配置好，Claude Code 做任何修改都会自动走你团队的质量门控，不需要每次提醒它。

CHAPTER 05

🔌 MCP 服务器：给 Claude Code 装上超能力插件

MCP 是什么

Model Context Protocol（MCP）是 Anthropic 在 2024 年 11 月发布的开源标准，让 AI 能连接外部工具和数据源。

没有 MCP 之前：Claude Code 只能操作本地文件和运行终端命令。
有了 MCP 之后：Claude Code 能直接查数据库、操作 GitHub、搜索 Slack、控制浏览器。

```
Claude Code ←→ MCP Server ←→ 外部系统
                              (GitHub / PostgreSQL / Slack / 浏览器 / ...)
```

添加第一个 MCP 服务器

以官方的文件系统 MCP 为例（允许 Claude 访问指定目录）：

```
claude mcp add filesystem npx @modelcontextprotocol/server-filesystem /Users/yourname/projects
```

或者手动编辑配置文件 `.claude/mcp.json`：

```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": [
        "@modelcontextprotocol/server-filesystem",
        "/Users/yourname/projects"
      ]
    }
  }
}
```

启动 Claude Code 后直接说：

```
> 帮我看看 /Users/yourname/projects 下有哪些 package.json
```

高价值 MCP 服务器推荐

GitHub MCP — 直接操作仓库

```
{
  "mcpServers": {
    "github": {
      "command": "npx",
      "args": ["@modelcontextprotocol/server-github"],
      "env": {
        "GITHUB_PERSONAL_ACCESS_TOKEN": "ghp_xxxx"
      }
    }
  }
}
```

接入后可以说：

```
> 帮我看一下 anthropics/claude-code 最近 5 个 issue，有没有和 hooks 相关的
> 给 #123 issue 加个评论，说我在本地复现了这个 bug
> 帮我创建一个 PR，从 feature/xxx 到 main
```

PostgreSQL MCP — 自然语言查数据库

```
{
  "mcpServers": {
    "postgres": {
      "command": "npx",
      "args": ["@modelcontextprotocol/server-postgres",
               "postgresql://localhost/mydb"]
    }
  }
}
```

```
> 帮我查一下过去 7 天注册的用户数，按天分组
> users 表里有没有重复的 email?
```

Claude Code 自动生成 SQL、执行、解释结果，不用你写一行查询。

Playwright MCP — 控制浏览器

```
{
  "mcpServers": {
    "playwright": {
      "command": "npx",
      "args": ["@playwright/mcp@latest"]
    }
  }
}
```

```
> 打开我们的测试环境 http://localhost:3000，登录后截图看看首页是否正常
> 帮我测试一下注册流程，看看有没有报错
```

适合自动化 E2E 测试和 UI 调试。

项目级 vs 全局配置

配置文件	作用范围	适合场景
`.claude/mcp.json`	当前项目	项目专用工具（特定 DB、内部 API）
`~/.claude/mcp.json`	全局所有项目	常用工具（GitHub、Slack）

项目级配置可以提交到 git，让团队成员共享相同的工具集。

实战：连接内部 API

很多公司有自己的内部 API（查员工信息、调度系统等）。用 MCP 可以快速接入：

```
// 用 MCP SDK 写一个简单的内部 API server
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { StdioServerTransport } from "@modelcontextprotocol/sdk/server/stdio.js";

const server = new McpServer({ name: "internal-api", version: "1.0.0" });

server.tool("get_employee", { id: { type: "string" } }, async ({ id }) => {
  const res = await fetch(`https://hr.internal/api/employees/${id}`);
  const data = await res.json();
  return { content: [{ type: "text", text: JSON.stringify(data) }] };
});

const transport = new StdioServerTransport();
await server.connect(transport);
```

配置：

```
{
  "mcpServers": {
    "internal-api": {
      "command": "node",
      "args": ["./mcp-servers/internal-api.js"]
    }
  }
}
```

现在 Claude Code 可以直接说：

```
> 帮我查一下员工 ID 为 E12345 的人的信息
```

常见坑

坑 1：MCP 服务器启动失败，但 Claude Code 没有明显报错

检查方法：运行 `claude mcp list` 看服务器状态，或单独运行服务器命令看报错：

```
npx @modelcontextprotocol/server-github
```

坑 2：工具太多导致上下文膨胀

MCP 工具定义会占用 context window。Claude Code 采用"延迟加载"——只有 Claude 决定用某个工具时才加载其完整定义，所以多装几个 MCP 服务器影响不大。

坑 3：数据库 MCP 返回了敏感数据

用只读账号连接，或在 MCP server 层做权限过滤，不要把生产数据库的写权限给 Claude。

查找更多 MCP 服务器

社区已经有数千个 MCP 服务器：

- modelcontextprotocol/servers — 官方维护的参考实现
- mcp.so — 社区 MCP 索引
- 搜索 `npm search mcp-server-xxx`

主流平台基本都有现成的 MCP server：AWS、Slack、Notion、Linear、Jira、Figma……装上就能用，不需要自己写。

CHAPTER 06

🪲 调试技巧：让 Claude Code 帮你定位 Bug

给 Claude Code 错误信息

最有效的调试方式：把错误日志直接丢进去。

```
# 方式一：直接在对话里粘贴
> 我跑 npm run build 报了这个错：
> TypeError: Cannot read properties of undefined (reading 'map')
> at UserList (src/components/UserList.tsx:23:18)

# 方式二：管道输入（适合长日志）
npm run build 2>&1 | claude -p "分析这个构建错误的原因，给出修复方案"

# 方式三：从日志文件读取
> 帮我看一下 @logs/error.log 最近的报错，找到根因
```

Claude Code 拿到错误信息后会自动定位到源文件，分析调用链，找到 root cause。

实战：排查运行时错误

```
> 用户反馈说点击"提交订单"按钮没反应，帮我查一下 OrderForm 组件
```

Claude Code 的排查路径：

1. 读取 `OrderForm.tsx`，找到 submit handler
2. 追踪整个调用链：组件 → service → API endpoint
3. 检查每一层的错误处理是否有遗漏
4. 定位问题（比如 `await` 缺失导致 Promise 未处理）
5. 修复并验证

用 Extended Thinking 处理复杂 Bug

碰到逻辑复杂、涉及多个模块的 bug，开启 extended thinking 让 Claude 深度分析：

```
# 在 Claude Code 里按 Option+T（macOS）或 Alt+T 开启 thinking mode
# 或者启动时指定
claude --permission-mode plan
```

在 thinking mode 下，Claude 会先做完整的代码分析，画出依赖关系，然后再给出修复方案。按 `Ctrl+O` 能看到它的推理过程。

适合用 thinking mode 的场景：

- 竞态条件（race condition）
- 内存泄漏

- 跨服务的数据不一致
- 性能瓶颈定位

把 Claude Code 集成到测试流程

用管道把测试输出直接给 Claude 分析：

```
# 跑测试，失败的话自动分析原因
npm test 2>&1 | claude -p "分析失败的测试，找到 root cause 并修复"

# 只看特定测试文件的失败
npx vitest run src/services/auth.test.ts 2>&1 | claude -p "这个测试为什么失败了？"
```

更自动化的做法——写个脚本循环修复：

```
#!/bin/bash
MAX_ATTEMPTS=3
for i in $(seq 1 $MAX_ATTEMPTS); do
  npm test && echo "测试通过" && exit 0
  npm test 2>&1 | claude -p "测试失败了，读取相关源码，修复 bug。只改源码，不改测试。"
done
echo "修复 $MAX_ATTEMPTS 次仍然失败，需要人工介入"
```

调试 API 接口

后端接口出问题时，可以结合 MCP 工具一起调试：

```
> 帮我查一下 /api/orders 接口为什么返回 500
> 用 PostgreSQL MCP 看一下 orders 表最近 10 条记录的状态
> 再对比一下代码里的查询逻辑
```

如果接了数据库 MCP Server，Claude Code 能直接执行 SQL 验证数据，比手动切换终端、数据库客户端效率高得多。

常见调试模式

场景	推荐方式
构建报错	管道输入 build 输出，让 Claude 分析
运行时错误	粘贴错误日志 + 堆栈信息
测试失败	管道输入测试结果，自动定位修复
性能问题	用 thinking mode 分析代码路径
UI 问题	截图粘贴（`Ctrl+V`），让 Claude 看界面
数据问题	接 PostgreSQL MCP，直接查数据库

🪱 有个 bug 让 Claude Code 修不动？扫码进群贴出来

群里每周有人分享自己的调试 prompt，比一个人闷头试快。

CHAPTER 07

📋 CLAUDE.md：让 AI 理解你的项目规范

CLAUDE.md 是什么

`CLAUDE.md` 是放在项目根目录的 Markdown 文件，Claude Code 每次启动时自动读取。它就像给 AI 的 onboarding 文档——告诉它这个项目的技术栈、编码规范、常用命令、架构约束。

写了 `CLAUDE.md` 之后，你不用每次对话都重复"我们用 TypeScript"、"测试用 Vitest"、"错误处理用 toast"这些话。

一个实际的 CLAUDE.md 例子

```
# CLAUDE.md

## 项目概述
这是一个 Next.js 14 全栈应用，前端用 React + TypeScript，后端用 Route Handlers。

## 技术栈
- 框架：Next.js 14 (App Router)
- 语言：TypeScript (strict mode)
- 样式：Tailwind CSS
- 数据库：PostgreSQL + Prisma ORM
- 测试：Vitest + React Testing Library
- 包管理：pnpm

## 常用命令
- pnpm dev          # 启动开发服务器
- pnpm test         # 跑全部测试
- pnpm test:watch   # watch 模式
- pnpm lint         # ESLint 检查
- pnpm db:migrate   # 数据库迁移
- pnpm db:seed      # 填充测试数据

## 编码规范
- 组件文件用 PascalCase：UserProfile.tsx
- 工具函数用 camelCase：formatDate.ts
- API route 放 app/api/ 目录
- 共享类型定义放 types/ 目录
- 所有 API 调用必须有 try-catch + 用户友好的错误提示
- 不要用 any，必要时用 unknown 然后做类型守卫

## 架构约定
- 数据获取统一走 services/ 层，组件不直接调 fetch
- 全局状态用 Zustand，表单状态用 React Hook Form
- 环境变量统一在 env.ts 里用 zod 校验
```

配置层级

CLAUDE.md 支持三个层级，优先级从高到低：

层级	文件位置	适用场景
项目级	项目根目录 `CLAUDE.md`	项目特有的规范，提交到 git
用户级	`~/.claude/CLAUDE.md`	个人偏好（比如中文回复）
子目录级	`src/api/CLAUDE.md`	特定模块的约束

当你用 `@src/api/route.ts` 引用文件时，Claude Code 会自动加载该文件所在目录及父目录的 `CLAUDE.md`。

Auto Memory

Claude Code 有自动记忆功能——在工作中它会自己学习项目的构建命令、调试方法等知识，保存到 `~/.claude/memory.json`。

你也可以手动触发记忆：

```
> 记住：这个项目的 lint 命令是 pnpm lint --fix
> 记住：数据库连接字符串在 .env.local 里，不要用 .env
```

这些记忆在下次会话里自动生效，不需要重新写 CLAUDE.md。

实用模板片段

前端项目必备：

```
## 测试规范
- 每个组件都要有对应的 .test.tsx
- Mock 外部依赖，不 mock 内部模块
- 用 screen.getByRole 而不是 getByTestId
```

后端项目必备：

```
## API 规范
- RESTful 路由命名：GET /api/users，POST /api/users
- 返回格式：{ data: T, error?: string }
- 认证用 JWT，中间件在 middleware.ts
- 所有数据库操作包 transaction
```

Monorepo 项目：

```
## Monorepo 结构
- packages/ui：共享 UI 组件库
- packages/api：API client SDK
- apps/web：前端应用
- apps/server：后端服务
- 改了 packages/ 下的代码需要跑 pnpm build --filter=@scope/package
```

避免写什么

CLAUDE.md 不是文档站，写多了反而干扰 AI 判断。避免：

- 冗长的项目历史介绍
- 大段的 API 文档（这些应该放代码注释里）
- 频繁变化的内容（用 auto memory 代替）
- 显而易见的规则（比如"不要删除 node_modules"）

保持简洁。200–500 行以内最合适。

CHAPTER 08

🧵 并行开发：Worktree 让多个 Claude 同时干活

为什么需要并行开发

一个 Claude Code session 在改文件时会占用当前工作目录。如果你想同时修一个 bug 和做一个新功能，两个 session 操作同一份代码会冲突。

Git Worktree 解决这个问题：给每个 session 一份独立的代码副本，共享同一个 git 仓库历史。

快速上手

```
# 创建一个叫 feature-auth 的 worktree 并启动 Claude Code
claude --worktree feature-auth

# 同时开另一个终端，创建 bugfix worktree
claude --worktree bugfix-login

# 不想起名？自动生成一个随机名
claude --worktree
```

`--worktree`（简写 `-w`）做了三件事：

1. 在 `.claude/worktrees/feature-auth/` 创建一份代码副本
2. 自动创建并切换到 `worktree-feature-auth` 分支
3. 在这个隔离环境里启动 Claude Code

两个 session 各自改代码、跑测试，互不影响。

实战：同时修 Bug + 做功能

终端 1：

```
claude --worktree bugfix-user-api
> /api/users 接口在用户没有 avatar 时返回 null 导致前端崩溃
> 修复这个 bug 然后创建 PR
```

终端 2：

```
claude --worktree feature-dark-mode
> 给应用加一个 dark mode 功能
> 用 Tailwind CSS 的 dark: 前缀实现
> 在 header 加一个切换按钮
```

两个任务并行推进，完成后各自 push 到远程仓库，互不干扰。

Worktree 的清理

退出 session 时 Claude Code 自动处理清理：

场景	行为
没有任何改动	自动删除 worktree 和分支
有 commit 但没 push	问你要不要保留
已经 push 到远程	问你要不要保留本地 worktree

手动清理也很简单：

```
# 查看所有 worktree
git worktree list

# 删除指定 worktree
git worktree remove .claude/worktrees/feature-auth
```

建议在 `.gitignore` 里加上 `.claude/worktrees/`，防止 worktree 内容被主仓库追踪。

Subagent 也能用 Worktree

Claude Code 支持在一个 session 里启动多个 subagent，每个 subagent 各自用独立的 worktree：

```
> 帮我同时做这三件事（每件事用独立的 worktree）：
> 1. 给 auth 模块写单元测试
> 2. 把 utils/ 下的 CommonJS 改成 ES Module
> 3. 更新 README 的 API 文档
```

Claude Code 会给每个子任务分配独立的 worktree，三个 subagent 并行工作。主 agent 协调进度，合并结果。

也可以在自定义 subagent 配置里固定启用 worktree：

```
# .claude/agents/test-writer.md
---
description: 自动写测试的 agent
isolation: worktree
tools:
  - Read
  - Write
  - Edit
  - Bash
---
读取指定模块的源码，生成对应的单元测试文件。
```

处理 .env 文件

Worktree 是全新的 checkout，不会复制 `.env` 等 gitignore 文件。在项目根目录创建 `.worktreeinclude`：

```
.env
.env.local
config/secrets.json
```

Claude Code 创建 worktree 时会自动把这些文件拷贝过去。

最佳实践

- 给 worktree 起语义化名字（`bugfix-login`、`feature-dark-mode`），方便识别
- 完成任务后及时清理不需要的 worktree，避免磁盘空间浪费
- 长期任务用 `claude --resume` 恢复 session，不用重新创建 worktree
- 团队协作时，每人用自己的 worktree 前缀（`alice-feature-xxx`）

🎯 **能玩到 Worktree 并行开发，说明你已经不是入门玩家了。想把 Claude Code 用进真实工程项目、攒出能写上简历的作品？Rain 聊聊你的方向：**

匠人学院
JR ACADEMY

工具会用了，接下来用它做什么？

这本手册的在线版在 jiangren.com.au/wiki/claude-code-guide，随官网每周更新 —— 收藏网页比收藏 PDF 更不容易过期。匠人学院还有 26 本同款工具指南（N8N、Cursor、Prompt Engineering…）和 AI Engineer 方向的项目制课程，想知道 Claude Code 在真实工程里怎么用到极限，来聊聊。

扫码进社群，AI 工具实操答疑

把这本 PDF 转给还在复制粘贴代码进 ChatGPT 的朋友。

jiangren.com.au