AI编程与智能体开发

林子雨　副教授　厦门大学
E-mail: ziyulin@xmu.edu.cn

2026年9月

主讲教师

- 2013年度厦门大学奖教金获得者
- 2017年度厦门大学奖教金获得者
- 2020年度厦门大学奖教金获得者
- 2023年度厦门大学奖教金获得者
- 2026年度厦门大学奖教金获得者
- 2018年国家精品在线开放课程
- 2020年国家级线上一流本科课程
- 2023年国家级线上一流本科课程
- 2020年福建省线上一流本科课程
- 2021年福建省线上线下混合一流本科课程
- 2018年福建省高等教育教学成果奖二等奖
- 2022年福建省高等教育教学成果奖特等奖
- 2024年福建省高等教育教学成果奖特等奖
- 2023年教育部国家智慧教育公共服务平台应用典型案例

配套教材

教材目录

第1章 大模型
第2章 AI编程概述
第3章 AI编程环境搭建
第4章 基于提示词的AI编程
第5章 基于规范的AI编程
第6章 流程驱动的AI编程
第7章 智能体概述
第8章 智能体开发框架LangChain
第9章 智能体开发框架LangGraph

教材特色

- 教材内容来自互联网大厂真实实践
- 以实践教学为依托
- 理论实践贯通
- 覆盖主流技术

《AI编程与智能体开发》

林子雨 编著

人民邮电出版社 预计2026年11月上市销售

教材官网提供详细信息和样书申请

官网： https://dblab.xmu.edu.cn/post/ai-programming-agent/

扫码访问教材官网
了解详情、获取资源、申请样书

第6章

流程驱动的AI编程

目录

Part 01

概述

6.1 概述

目录

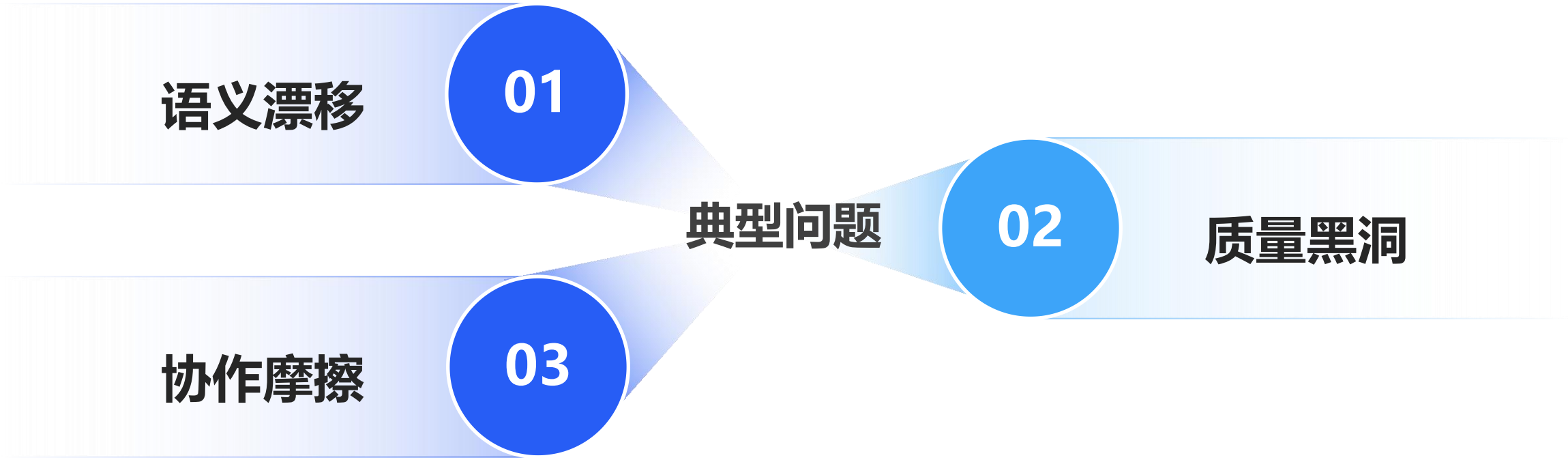
6.1.1 AI编程的规模化挑战

在第4章和第5章中，我们分别探讨了提示词驱动和规范驱动两种AI编程方法论。前者解决"单次交互质量"的问题，后者解决"需求一致性"的问题。然而，当AI编程从个人工具走向团队实践、从探索实验走向生产交付时，**一个新的问题浮现出来——规模化的确定性**

试想以下场景：一个5人团队使用AI编程工具进行日常开发，每个成员每天进行20-50次AI交互。在没有统一流程约束的情况下，每个成员的每次交互都是一次独立决策，从需求理解到代码实现再到质量验证，路径千差万别。一个月后，团队将积累3000-7500次AI交互，其中，某些交互可能引入隐蔽的Bug、不一致的代码风格、缺失的测试覆盖或未处理的安全隐患。这些问题不是AI的能力问题，而是过程缺失的问题

6.1.1 AI编程的规模化挑战

这种"过程缺失"导致的典型问题包括：



这类问题被业界称为"最短路径陷阱"，**即AI天然倾向于选择最直接的方式满足当前指令，而不是从工程全局出发做出最稳健的决策**。解决这一问题的关键不是"让AI变聪明"，而是在AI的决策路径上设置强制性的质量关卡。

6.1.1 AI编程的规模化挑战

■ 语义漂移

AI在实现过程中"自行发挥"，产出偏离原始需求。

1. 第4章已详细分析了这一现象的机理——AI在模糊需求面前自动填补假设，输出结果与真实意图逐渐偏离

2. 在个人场景下，语义漂移的代价相对可控，你和AI在同一个对话中可以随时纠正

3. 但在团队场景中，语义漂移的后果会被放大：成员A用AI生成了偏离需求的代码，成员B在此基础上继续开发，偏差层层叠加，最终在集成测试或代码审查阶段才被发现，返工成本远超预期

6.1.1 AI编程的规模化挑战

■ 质量黑洞

未经测试的代码直接进入代码库，Bug积累速度超过发现速度。

AI生成代码的速度远超人类，一个开发者用AI，一天可以产出数百行代码，但如果这些代码没有经过系统化的测试验证，Bug就会像"黑洞"一样不断吞噬团队的精力

更危险的是，AI生成的代码往往"看起来正确"，命名规范，逻辑通顺，甚至有注释，这给开发者一种"已经验证过"的错觉，实际上边界条件、异常处理和并发安全等隐患可能深藏其中

缺乏测试保护网，还会导致团队对代码库产生"重构恐惧"，即修改一处可能引发未知的连锁反应，久而久之"能跑就别动"，技术债持续累积

6.1.1 AI编程的规模化挑战

■ 协作摩擦

不同成员的使用习惯差异导致代码风格和架构决策不一致。

成员A习惯让AI先写测试再实现，成员B习惯让AI直接生成完整代码再补测试；成员A的提示词中明确要求异常处理，成员B的提示词中遗漏了这一约束

01

由此导致的结果是，同一个项目中，某些模块有完善的异常处理和测试覆盖，另一些模块则缺失这些内容。这种不一致性会在代码审查时暴露出来，但修复成本远高于预防成本

02

与此同时，开发过程缺乏文档化记录，维护者无法理解代码的设计意图，也就是说，AI生成的代码只保留了"怎么做"的结果，却丢失了"为什么这样做"的决策过程，形成"知识断层"

03

6.1.2 从个人到企业的演进路径

AI编程方法论的发展呈现出一条清晰的演进路径（如下页图所示），具体如下：

第一层

提示词驱动（个人级）

这是AI编程的起点，开发者通过精心设计的提示词引导AI生成代码。其核心关注"单次交互的质量"，解决的是"如何让AI这一次写出正确的代码"

第二层

规范驱动（团队级）

当多人在同一代码库上协作时，仅靠每个人的提示词技巧无法保证产出的一致性。规范驱动方法通过标准化的specs文件定义"做什么"和"验收标准"，让不同成员、不同时间产生的AI代码都能收敛到同一个目标。其核心关注"需求的一致性"和"变更的可追溯性"

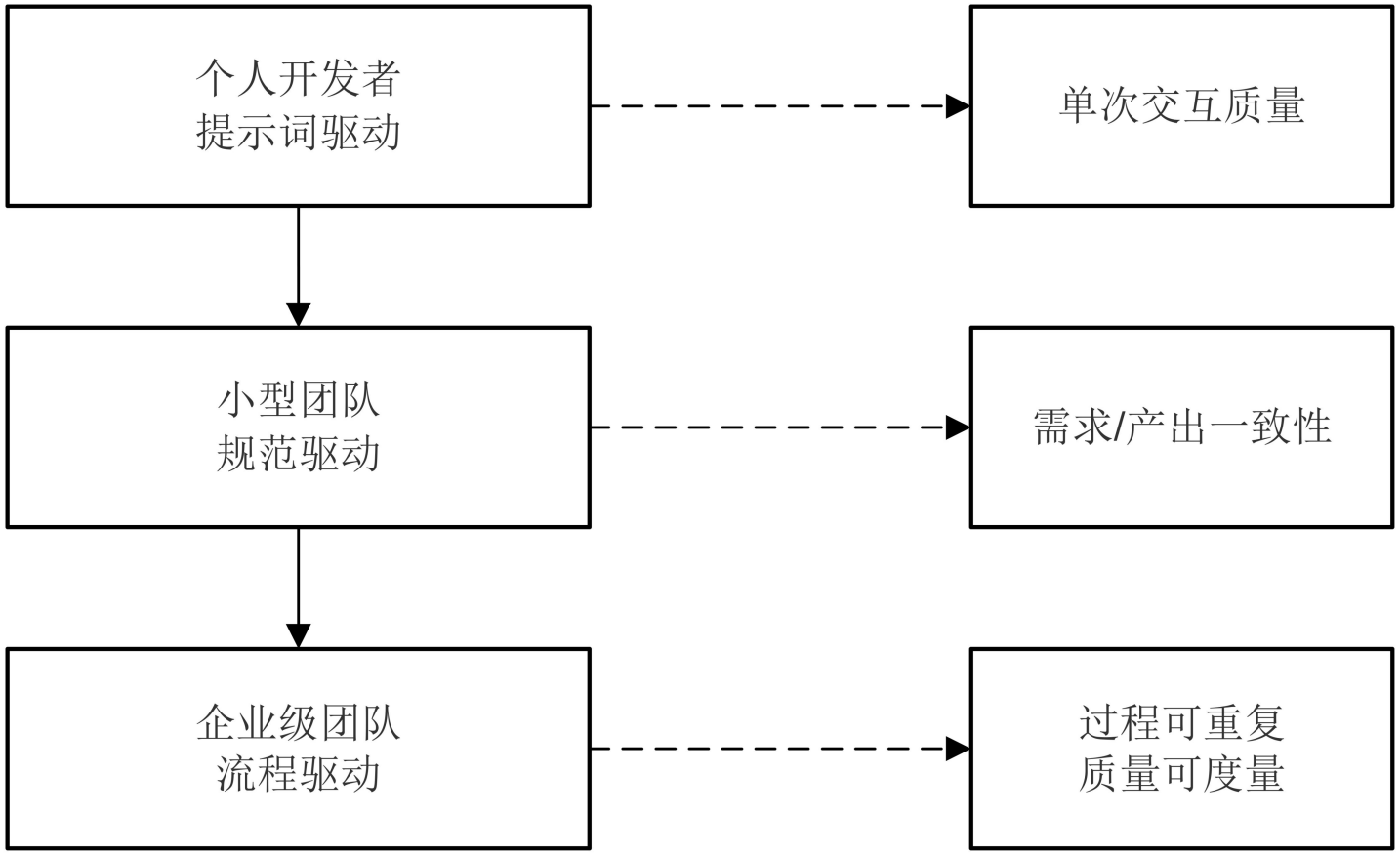
第三层

流程驱动（企业级）

当团队规模和项目复杂度进一步增加时，仅靠specs文件来规范需求仍然不够。流程驱动方法将软件工程的最佳实践（TDD、代码审查、分支隔离、持续集成）固化为不可跳过的强制性步骤，形成一条"测试→实现→审查→归档"的标准化生产线，其核心关注点是"过程的可靠性"和"质量的可度量性"

6.1.2 从个人到企业的演进路径

从提示词驱动到规范驱动再到流程驱动，不是"后者替代前者"的线性演进，而是"后者内置前者"的层级叠加。流程驱动在每一步内部仍然使用提示词技巧，在需求管理层面仍然依赖规范文档，但它增加了一个前两层级所没有的维度——强制性的质量关卡和不可跳过的标准化流程。



6.1.2 从个人到企业的演进路径

当下 AI 编程形成提示词、规范、流程驱动三种主流模式。提示词灵活轻量适合快速原型；规范统一代码质量标准；流程驱动搭建完整开发流水线。三者能力层级不同、适配场景差异明显，下表从特性、优势、短板、适用场景等维度，对三种编程模式进行了细致对比分析。

类别	提示词驱动	规范驱动	流程驱动
特点	纯对话式，靠自然语言表达的提示词来下达需求，AI自由拆解、写代码、改错	前置统一约束标准：编码规范、接口协议、数据结构、注释格式、安全校验规则、测试模板等，AI 输出必须匹配规范	把软件开发全链路拆成固定、有序、可校验的标准化工序流程：需求拆解→架构设计→模块编码→单元测试→联调→安全扫描→文档生成→部署校验，每一步有输入、输出、校验卡点、责任人/AI 角色、流转触发条件
优势	轻量、快速原型、小功能开发	代码风格统一、减少低级错误、可维护性提升	有序分阶段管控，多智能体协同，阶段校验防风险，上下文完整可追溯，适配企业 CI/CD 规模化工程开发
短板	无约束、无步骤管控、无标准化产出，复杂工程极易混乱，AI 容易跳步骤、漏边界、忽略上下游依赖	只管产出质量标准，不管做事先后顺序与协作步骤，只控结果形态，不控过程动作	搭建配置繁琐，小项目开销高，流程僵化难灵活试错，调整工序需要改动整套流水线
适用场景	小型脚本、临时工具、快速原型验证、个人零散代码编写、简单算法 Demo，一人快速调试试错场景	中小型独立模块、团队常规业务迭代、统一代码风格项目，需保证代码整洁、接口统一、基础质量可控的开发场景	大型复杂系统、多团队协同项目、企业级正式交付、需 CI/CD 与多角色分工、强安全质量管控的工程场景

6.1.3 Superpowers的核心价值

流程驱动的AI编程，需要借助于得力的AI编程工具才能顺利实施，这里推荐使用Superpowers。Superpowers是由GitHub联合创始人Jesse Vincent在2025年发起的开源项目，本质上是一套面向编码智能体的结构化软件开发工作流。它不是一个新的AI模型，也不是一个独立的开发工具，而是一个建立在AI编程工具之上的工作流框架，通过智能体集成层对接不同的AI编码工具，通过工作流层强制执行标准化开发流程，通过技能层定义每个步骤的具体行为。

6.1.3 Superpowers的核心价值

Superpowers的核心价值可以用一句话概括——将软件工程的确定性注入AI编程的不确定性中。它通过以下机制实现这一目标：

强制质量关卡	工作流中的每一个步骤都是一个"检查点"，前一步的输出必须满足预设条件才能进入下一步。例如，test-driven-development步骤强制要求测试先于实现；requesting-code-review步骤强制要求独立代码审查；verification-before-completion步骤强制要求提供新鲜的验证证据
标准化作业程序	无论团队中谁在使用AI编程工具，无论面对什么类型的开发任务，Superpowers都规定了一套标准的作业程序（SOP：Standard Operating Procedure）。这消除了"个人习惯差异"导致的产出波动
可复用的Skill	Superpowers将其工作流中的每个步骤封装为独立的Skill（技能），这些Skill可以自由组合、按需裁剪，适应不同规模和类型的项目需求

截至2026年6月，Superpowers已被主流AI编程平台认证为推荐插件，社区贡献了14个核心Skill。

Part 02

Superpowers的定义与架构

6.2 Superpowers的定义与架构

01

什么是Superpowers

02

三层架构解析

03

与其他方法论的关系

6.2.1 什么是Superpowers

Superpowers的定义可以从三个维度理解：

■ 从功能维度看

Superpowers是一套编码智能体的结构化软件开发工作流（Structured Development Workflow for Coding Agents）。

它将软件开发过程分解为一系列标准化步骤，每个步骤由一个专门的Skill负责执行，Skill之间通过明确的输入输出接口连接，形成一条端到端的开发流水线

可以将其类比为制造业中的"流水线"——原材料（用户需求）从一端进入，经过一系列标准化的加工工序（brainstorming→writing-plans→TDD→code-review），最终从另一端产出经过质量检验的成品（可交付的代码）

与"手工作坊"式的提示词编程不同，流水线上的每个工序都有明确的操作规范和检验标准，不依赖于操作者的个人技艺

6.2.1 什么是Superpowers

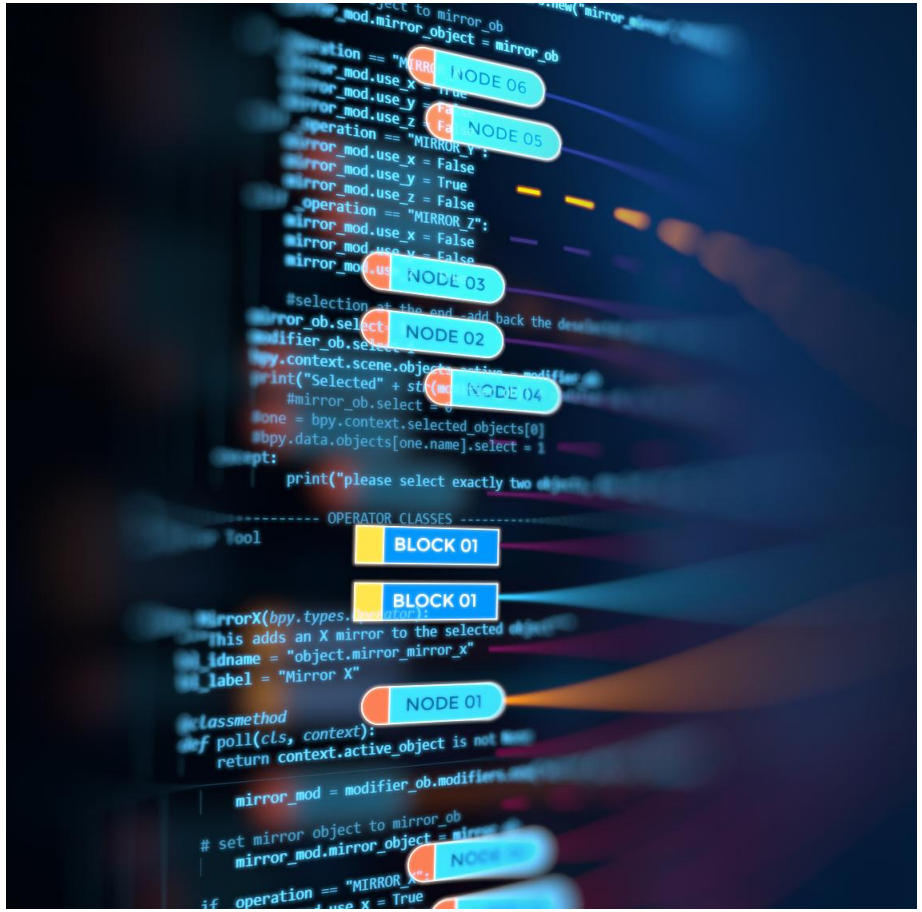
■ 从工程维度看

Superpowers是软件工程最佳实践在AI编程场景下的重新实现。

TDD（Test-Driven Development：测试驱动开发）、代码审查、持续集成、分支管理，这些传统软件工程中的经典实践，在Superpowers中被重新设计为适合AI智能体执行的自动化流程

传统软件工程中，这些实践依赖开发者的自觉遵守——你"应该"先写测试，你"应该"做代码审查，但没有人强制你。Superpowers将"应该"变为"必须"——流程不可跳过，铁律不可违反

这种从"自律"到"他律"的转变，正是流程驱动与提示词驱动、规范驱动的根本区别



6.2.1 什么是Superpowers

■ 从团队维度看

Superpowers是AI编程从"个人工具"升级为"团队基础设施"的关键组件。

1. 它提供了协作所需的共同语言（标准化工作流）、质量保障机制（强制关卡）和知识沉淀能力（设计文档和实现计划的自动生成）

2. 在传统软件工程中，团队协作的基础设施包括版本控制系统（Git）、持续集成服务器（Jenkins）、代码审查平台（Gerrit）等

3. Superpowers在AI编程场景中扮演了类似的角色，它不是某个人的效率工具，而是整个团队共享的工程基础设施，确保无论谁在使用AI编程工具，产出都符合统一的质量标准

6.2.1 什么是Superpowers

Superpowers适用于以下场景：

在已有项目中开发新功能
（功能迭代）

Bug修复
（三步精简流程）

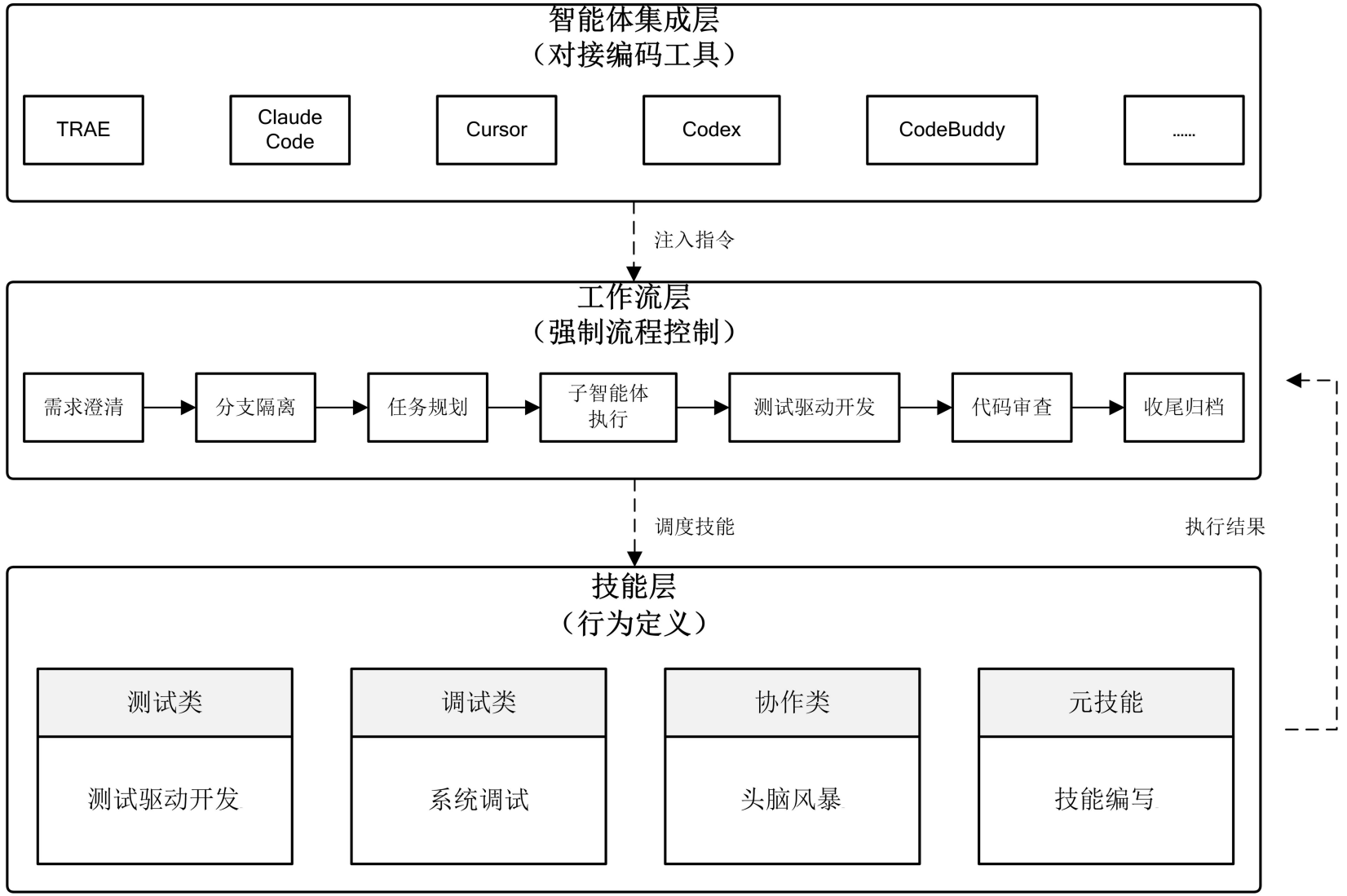
从零开发新项目
（如本章的缓存数据库案例）

代码重构
（在测试保护网下安全进行）

需要注意的是，Superpowers不适用于"一句话任务"，比如修个小Bug、改个配置、做个快速原型。这些任务用提示词驱动即可，如果给这类任务也加上完整流程，那就是“过度工程化”，不建议这么做。

6.2.2 三层架构解析

Superpowers的架构分为三个层次（如下图所示）：



6.2.2 三层架构解析

■ 智能体集成层（Agent Integration Layer）——对接编码工具。

Superpowers不是独立的开发工具，而是建立在现有AI编程工具之上的流程框架

智能体集成层负责将Superpowers的能力适配到不同的AI编码智能体（包括Claude Code、Cursor、Codex CLI、OpenCode、TRAE等）

每个AI编码工具有不同的插件机制和API接口，智能体集成层通过平台适配器（Platform Adapter）屏蔽这些差异，使得上层的Skill和工作流无需关心底层工具的具体实现

例如，Claude Code通过.claude-plugin/plugin.json加载Superpowers，OpenCode通过.opencode/plugins/superpowers.js加载，而Codex CLI通过.codex/superpowers-codex脚本加载，虽然加载方式不同，但加载后的Skill内容完全一致

6.2.2 三层架构解析

- **工作流层（Workflow Layer）——强制流程控制。**

定义了从需求到交付的标准化七步流程，每个步骤的完成都是一个"门"，只有通过前一个门的检验，才能进入下一个步骤

01

这一层的设计理念来自传统软件工程的"阶段门控"（Stage-Gate）模型，当开发者试图跳过某个阶段时，智能体会拒绝并引导回到正确的流程

02

6.2.2 三层架构解析

- **技能层（Skills Layer）—— 行为定义。**

14个独立的SKILL.md文件，可发现、可调用、可组合

每个Skill都是一个自包含的提示词模板，定义了该步骤的执行逻辑、输入输出格式和质量标准

Skill之间通过文件系统（specs文件、plans文件、源代码文件）传递数据，而非通过内存中的变量，这种设计使得每个Skill的执行结果都是可审计、可追溯的。14个Skill按职责分为四类：协作类、测试类、调试类、元类

6.2.2 三层架构解析

需要特别说明的是，Superpowers的三条铁律（测试先行、根因分析、完成验证）并非独立的架构层，而是内嵌在技能层中的强制规则

例如，"测试先行"铁律由test-driven-development这个Skill强制执行，"根因分析"铁律由systematic-debugging这个Skill强制执行，"完成验证"铁律由verification-before-completion这个Skill强制执行

铁律与Skill的关系类似于法规与执法机构，铁律定义了"什么不能做"，对应的Skill负责"强制执行"。即使裁剪工作流步骤，铁律仍然通过其对应的Skill生效

6.2.3 与其他方法论的关系

Superpowers与第4章的提示词驱动、第5章的规范驱动（OpenSpec）之间存在明确的互补关系，而非替代关系，具体如下：

与提示词驱动的关系

Superpowers的每个步骤内部仍然使用提示词驱动机理，每个Skill本质上是精心设计的系统提示词模板。区别在于，提示词驱动依赖开发者的个人技巧来构造提示词（如第4章介绍的PTCF框架），而Superpowers将最佳提示词模式固化为可复用的Skill。从"每个人的提示词各不相同"到"所有人使用相同的经过验证的提示词模板"

与规范驱动（OpenSpec）的关系

OpenSpec关注"做什么"（需求定义、验收标准、变更记录），Superpowers关注"怎么做"（开发流程、质量关卡、交付标准）。第5章介绍了规范驱动编程的“三条铁律”（规范先行、验收标准驱动、变更可追溯），它们解决的是"需求管理"维度的问题；而Superpowers的“三条铁律”（测试先行、根因分析、完成验证）解决的是"过程管理"维度的问题。两者分别覆盖了软件开发生命周期中两个正交的维度，可以协同使用。具体的协同模式将在6.6节详述

三者构成的工具箱

三种方法论并非"选A就不能选B"的互斥关系，而是同一套工具箱中的不同工具。面对具体任务，开发者应根据任务特征选择合适的方法论组合：简单修改用提示词驱动，多人协作用规范驱动，重要项目用流程驱动，复杂项目用方法组合

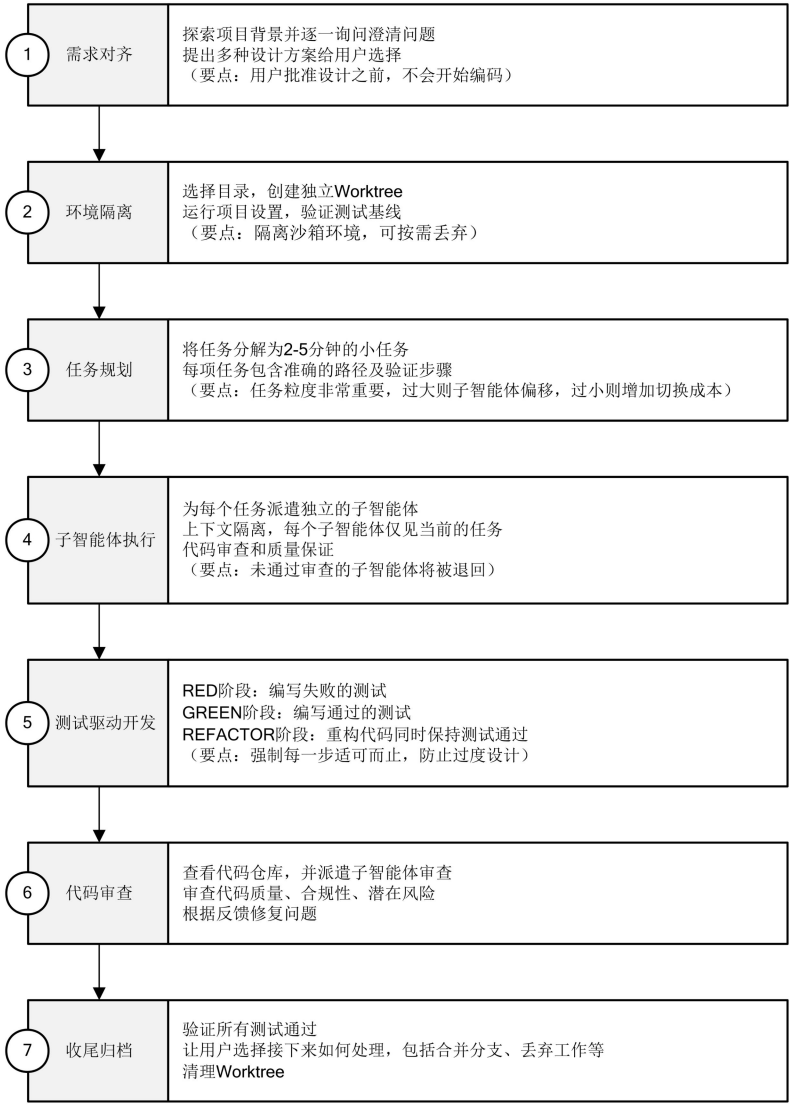
Part 03

Superpowers工作流详解

6.3 Superpowers工作流详解

6.3.1 七步标准流程

Superpowers的标准开发流程包含七个步骤，形成一条从需求到交付的完整链路，如图所示。



6.3.1 七步标准流程

■ 步骤1：需求对齐（brainstorming）

这是整个流程的起点。“需求对齐”阶段采用苏格拉底式提问方法——不是让AI猜测你意图，而是通过连续追问帮你理清真实需求。这一步骤的关键原则是"不问清楚不罢休"，只有需求完全明确后才进入下一步

这种系统化的追问机制具有重要的赋能价值，通过反复追问和深度思考，即使是刚毕业的大学生，也能够像资深工程师一样，对需求进行全面考虑，设计出合理的方案。AI的追问相当于一位经验丰富的导师在旁指导，帮助年轻开发者建立系统化的思维模式

读者可能注意到，这种"追问逼出真实需求"的思路与第4章介绍的提示词工程中"约束补全"技巧一脉相承。在第4章中，开发者使用PTCF框架主动补全提示词中的约束条件（Purpose、Task、Constraint、Format），以消除AI生成结果的不确定性；而Superpowers将这一过程自动化，也就是说，AI主动追问，逼出用户未表达的真实需求，从而在编码之前消除需求层面的不确定性

在缓存数据库项目中，“需求对齐”阶段产生了约15轮追问，覆盖了数据模型、容量约束、异常处理、持久化、并发安全等全部维度，最终输出了一份完整的设计规格文件。详细的追问过程和输出结果将在6.5.1节完整呈现

6.3.1 七步标准流程

■ 步骤2：环境隔离（using-git-worktrees）

对于新项目，此步骤会创建Git仓库和初始提交；对于已有项目，此步骤会创建独立的Git Worktree（工作树）来隔离开发环境

Git Worktree是Git的一项功能，允许在同一仓库中同时检出多个分支到不同的目录。与传统的“git branch+git checkout”不同，Worktree为每个分支提供了独立的文件系统视图，这意味着你可以在一个目录中开发新功能的同时，在另一个目录中修复紧急Bug，互不干扰。关于Git Worktree的基本操作和概念，已经在本书第3章中做了详细介绍

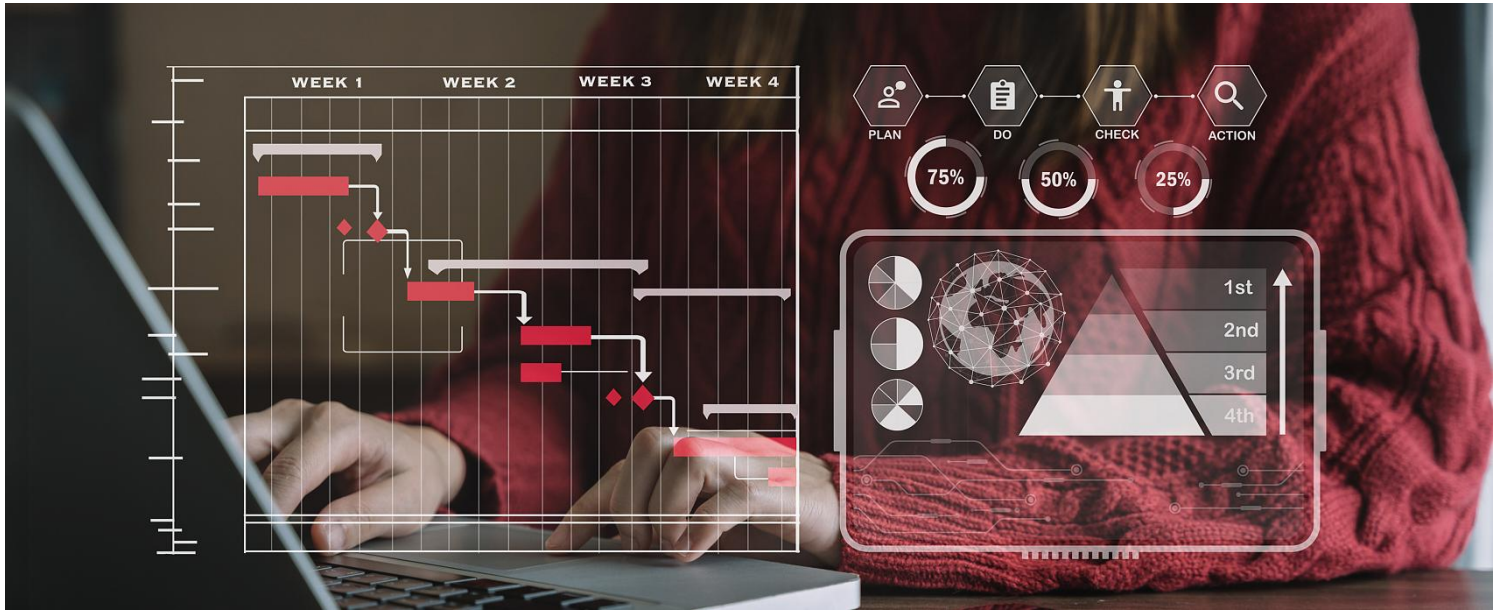
Superpowers使用Worktree的核心目的是绝不污染主干。所有开发工作都在隔离的工作树中进行，只有经过完整七步流程验证的代码才有资格合并回主干。这与第5章OpenSpec中通过规范文件实现"需求隔离"的思路一脉相承，OpenSpec是在文档层面隔离需求变更（每次变更都有独立的proposal文件），而Superpowers则是在代码层面隔离开发环境（每次开发都有独立的工作树）

6.3.1 七步标准流程

■ **步骤3：任务规划（writing-plans）**

> **基于明确的需求，“任务计划”阶段将整个开发任务拆解为一系列可以"2-5分钟完成"的小任务。每个任务包含以下信息：任务描述、预估难度、涉及的文件路径、依赖关系**

> **拆分原则是：每个任务应该足够小到可以独立验证，又足够大到有实际意义。在缓存数据库项目中，基础功能开发被拆解为8个子任务，每个任务对应一个或多个测试用例**



6.3.1 七步标准流程

■ 步骤4：子智能体执行（subagent-driven-development）

这是Superpowers最具特色的步骤。对于步骤3中拆解的每个小任务，“子智能体执行”阶段会创建一个独立的子智能体（Subagent）来执行，各个子智能体只会聚焦自身任务——按计划写代码、补逻辑、调用依赖、本地自测，不跨任务修改其他模块。每个子智能体拥有自己独立的上下文，互不干扰

这种"分而治之"的模式带来两个关键优势：一是隔离性，每个任务的上下文不会被前序任务的错误决策污染；二是并行性，多个不相关的任务可以同时执行（通过dispatching-parallel-agents）

6.3.1 七步标准流程

■ 步骤4：子智能体执行（subagent-driven-development）

子智能体的执行机制值得深入理解。每个子智能体在启动时，主智能体会为其注入三类信息：当前任务的完整描述（来自plans文件）、相关的specs片段（来自specs文件）、依赖文件的关键接口定义。子智能体基于这些信息独立完成"编写测试→编写实现→验证通过"的完整TDD循环，然后将结果返回主流程。子智能体无法访问其他子智能体的执行过程，也无法修改其他子智能体正在处理的文件，这种隔离确保了并行执行的安全性。

与单智能体模式相比，子智能体模式的核心差异在于上下文隔离。单智能体模式下，所有任务共享同一个上下文窗口，前序任务的实现细节会"污染"后续任务的决策——AI可能因为"记住"了前序任务的实现方式，而在后续任务中做出不必要的一致性假设。子智能体模式下，每个任务从"干净"的上下文出发，只关注当前任务的specs文件和plans文件，避免了跨任务的上下文干扰。在缓存数据库项目中，8个子任务由8个独立的子智能体逐一执行，每个子智能体遵循TDD流程（先写测试、再写实现、验证通过），完成后将结果返回主流程。

6.3.1 七步标准流程

■ 步骤5：测试驱动开发（test-driven-development)

TDD步骤强制执行经典的RED-GREEN-REFACTOR循环：

在AI编程场景中，TDD具有对抗"过度设计"的独特价值。AI天然倾向于输出"完整方案"，即使你只要求一个简单的键值存储，它也可能主动添加序列化、过期策略、LRU淘汰等功能。TDD通过"先定义测试=先定义完成标准"的机制，强制AI保持克制。

6.3.1 七步标准流程

■ 步骤6：代码审查（requesting-code-review）

每个子任务完成后，requesting-code-review会触发一次自动代码审查。审查覆盖以下维度：

命名规范

变量、函数、类的命名是否符合项目约定（如Python的PEP8命名规范），是否存在与标准库或第三方库的命名冲突

异常处理

是否正确处理了所有已声明的异常类型，是否有裸except捕获所有异常的反模式

边界条件

空输入、极大值、极小值、None值等边界情况是否被测试覆盖

安全隐患

线程安全、资源泄露、SQL注入等安全相关问题

审查结果按严重程度分为两级：阻断性问题（如命名冲突、逻辑错误、安全漏洞）和建议性问题（如代码风格不一致、注释不够清晰）。阻断性问题必须修复后重新审查，流程才能继续；建议性问题可以记录为技术债，在后续迭代中处理。

如果审查发现严重问题（如命名冲突、逻辑错误），流程被阻断，必须修复后重新审查。如果审查通过，代码进入合并队列。在缓存数据库项目中，代码审查发现了一个关键问题：自定义异常KeyError与Python内置异常同名，导致潜在的语义混淆，已修复为KeyNotFoundError。

6.3.1 七步标准流程

■ 步骤7：收尾归档（finishing-a-development-branch）

所有子任务完成并通过审查后，finishing-a-development-branch负责收尾工作：

01 将开发分支合并到主分支（master/main）

02 删除临时工作树，清理工作环境

03 提供最终的项目交付总结（代码行数、测试数量、文件清单）

此时，一个经过完整七步流程验证的功能开发正式交付完成。

6.3.2 核心Skill功能与组合

■ **Skill四大分类**

14个Skill按职责分为四类：

协作类

包含9个Skill，覆盖从需求澄清到代码审查的完整开发流程，是开发者与AI协作的主力Skill

测试类

是TDD（测试驱动开发）的执行载体，强制执行RED-GREEN-REFACTOR循环

调试类

包含系统化调试和完成前验证两个Skill，服务于Bug修复和质量确认场景

元类

包含writing-skills和using-superpowers两个引导性Skill，帮助开发者创建自定义Skill和熟悉Superpowers框架本身。

6.3.2 核心Skill功能与组合

■ Skill四大分类

下表列出了全部14个Skill的分类、功能及对应的流程步骤。

类别	Skill名称	功能描述	对应步骤
协作类	brainstorming	苏格拉底式追问，挖掘真实需求，输出设计规格	步骤1
协作类	writing-plans	将需求拆解为2-5分钟的小任务，标注文件路径	步骤3
协作类	subagent-driven-development	逐任务派发子智能体执行，确保逐任务验证	步骤4
协作类	dispatching-parallel-agents	并行派发多个子智能体，提升效率	步骤4
协作类	requesting-code-review	自动代码审查，严重问题阻断流程	步骤6
协作类	receiving-code-review	接收和处理代码审查反馈，响应审查意见并修复问题	步骤6
协作类	using-git-worktrees	创建和维护Git工作树，确保环境隔离	步骤2

6.3.2 核心Skill功能与组合

■ Skill四大分类

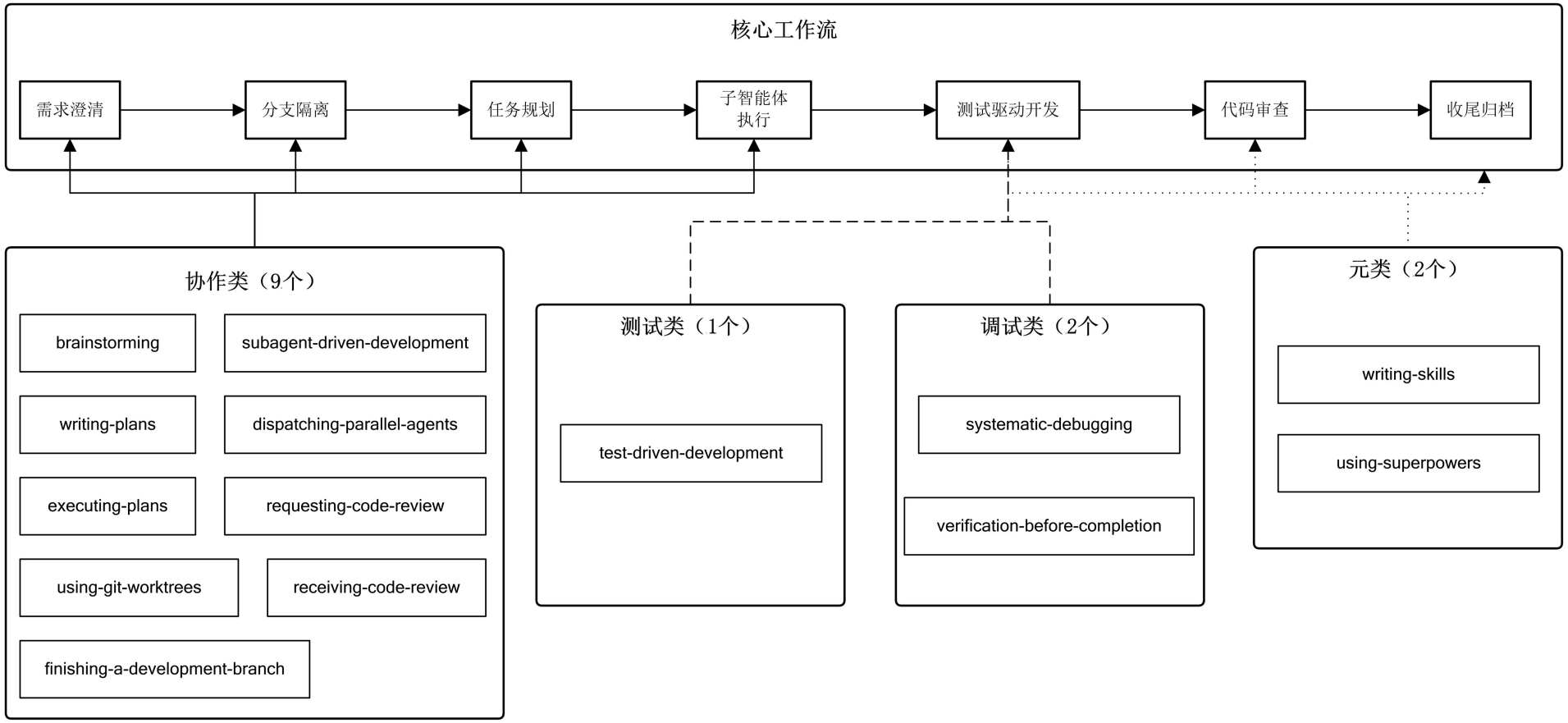
下表列出了全部14个Skill的分类、功能及对应的流程步骤。

类别	Skill名称	功能描述	对应步骤
协作类	finishing-a-development-branch	合并分支、清理工作树、交付总结	步骤7
协作类	executing-plans	按计划文件自动执行任务列表	步骤3-4
测试类	test-driven-development	强制RED-GREEN-REFACTOR循环，铁律"测试先行"的执行载体	步骤5
调试类	systematic-debugging	四阶段系统调试：根因调查→模式分析→假设检验→实施修复	Bug修复
调试类	verification-before-completion	完成前强制验证：全量测试+手动确认+无回归	步骤7
元类	writing-skills	创建和更新自定义Skill	元技能
元类	using-superpowers	Superpowers自身的引导和使用说明	元技能

6.3.2 核心Skill功能与组合

■ Skill与七步流程的映射

下图展示了四类Skill在七步流程中的分布关系。可以清晰地看到，每个流程步骤都有对应的Skill作为执行载体，而调试类Skill不仅服务于标准流程的验证环节，还在Bug修复场景中承担核心角色。



6.3.2 核心Skill功能与组合

■ **三个场景和三套流程**

不是每个开发场景都需要走完完整的七步流程。Superpowers的设计理念是按场景裁剪工作流，在保证质量的前提下避免过度流程化。三种典型场景对应三套不同的Skill组合，如表5-2所示。

场景	工作流	步骤数	核心差异
新项目开发	完整七步流程	7步	brainstorming侧重需求从零梳理
老项目加功能	完整七步流程	7步	brainstorming侧重理解已有代码和评估影响范围
Bug修复	精简三步流程	3步	跳过需求澄清和环境隔离，精准打击

6.3.2 核心Skill功能与组合

■ 三个场景和三套流程

1）场景一：新项目开发（完整七步流程）

这是Superpowers能完整发挥的场景，没有历史包袱，可以走完整个七步工作流。新项目的核心风险不在"写不出来"，而在"写偏了"——让AI从零搭建项目，写完以后发现架构不对、技术栈选错、需求理解有偏差，返工成本远高于前期设计成本。brainstorming这个Skill有一条硬性门槛——不展示设计并获得用户批准前，绝不启动任何实现。这等于强制在写代码前先把需求想清楚。全部7步、14个Skill按需激活，具体激活顺序如下：

```
brainstorming → using-git-worktrees → writing-plans → subagent-driven-development → test-driven-development → requesting-code-review → finishing-a-development-branch
```

6.3.2 核心Skill功能与组合

- 三个场景和三套流程

（2）场景二：老项目加功能（完整流程，侧重不同）

与场景一相比，工作流步骤相同，但每一步的重点截然不同。老项目有历史包袱，现有的代码模式、架构风格、依赖版本都是约束条件。brainstorming这个Skill在老项目中有一个专门的指导原则——"Working in existing codebases"，**核心有三条：遵循现有代码模式和架构、不提议无关的重构、新代码要和项目风格一致**。简而言之，"入乡随俗"——AI看到项目用“Vue 2 + Options API”，就不要建议迁移到Vue 3 Composition API；看到项目用Jest做测试，就不要换成Vitest，除非新功能确实需要

值得注意的是，在老项目中不应跳过using-git-worktrees。有些人觉得"加个小功能而已，不用隔离"，但Worktree的价值不仅是隔离代码，更关键的是提供一个"可以随时丢弃的沙箱"——改坏了直接删Worktree，主分支毫发无损。在老项目中，这种安全网比新项目更重要，因为，改动可能影响已有的业务逻辑

6.3.2 核心Skill功能与组合

■ 三个场景和三套流程

（2）场景二：老项目加功能（完整流程，侧重不同）

在完整7步流程中，各步骤的侧重点如下（只给出需要人参与和关注的步骤的侧重点，一些由AI自动完成并且不需要人参与的步骤，就没有给出侧重点）：

01 brainstorming（侧重理解现状+评估影响范围）

02 using-git-worktrees

03 writing-plans（关注与现有代码的集成点）

04 subagent-driven-development

05 test-driven-development

06 requesting-code-review（侧重代码风格一致性和不必要依赖检查）

07 finishing-a-development-branch

6.3.2 核心Skill功能与组合

■ 三个场景和三套流程

（3）场景三：Bug修复（精简三步流程）

这个场景与前两个有本质区别——目标明确、范围可控，不需要brainstorming来梳理需求，也不需要Worktree来隔离（通常在现有分支上直接修）。Superpowers将工作流精简为三步精准打击：

```
systematic-debugging → test-driven-development → verification-before-completion
```

systematic-debugging是Bug修复场景的核心Skill，严格分四个阶段：根因调查（只收集信息不修改代码）→模式分析（判定偶发还是必现）→假设与检验（提出假设并用实验验证）→实施修复（基于确认的根因进行修改）。在修复前，必须先写一个能复现Bug的失败测试，这既是"诊断证明"（确认Bug确实存在），也是"疗效验证"（证明修复确实有效）。修复完成后，verification-before-completion要求提供"新鲜的验证证据"：所有测试通过（包括新写的和原有的）、手动验证Bug已修复、相关功能无回归。

6.3.3 质量保障机制

■ 铁律1：测试先行（Tests Come First）

测试先行是指不容忍未经测试的代码，在编写任何实现代码之前，必须先编写测试代码。所有的测试必须通过后才视为任务完成

TDD（Test-Driven Development）是Superpowers最核心的质量机制。它不仅是"写测试"这么简单，而是从根本上改变了代码的生产方式——从"先实现再验证"变为"先定义完成标准再实现"。这意味着每一行生产代码都有明确的测试覆盖，每一个功能都有可验证的完成定义

在实践中需要注意的是，随着项目迭代，测试数量持续增长，每次RED-GREEN-REFACTOR循环运行全部测试，会显著降低开发效率。建议在开发循环中使用测试框架的过滤功能，只运行与当前任务相关的测试（如pytest中pytest tests/test_cachedb.py::test_set_basic），完整套件在代码审查阶段运行

6.3.3 质量保障机制

■ 铁律2：根因分析（Root Cause Before Fix）

根因分析是指绝对不允许"猜测式修复"，在修改任何代码之前，必须先通过系统化的调试过程确定问题的根因。

这条铁律由systematic-debugging这个Skill强制执行。它将排错过程分为四个阶段：

根因调查

只收集信息不修改代码

模式分析

判定是偶发还是必现

假设与检验

提出假设并用实验验证

实施修复

基于确认的根因进行修改

6.3.3 质量保障机制

■ 铁律3：完成验证（Verify Before Completion）

完成验证是指在任务宣布完成前，必须提供"新鲜的"验证证据——不是"之前跑过"，不是"应该没问题"，而是实实在在的、刚刚执行过的验证结果

这条铁律由verification-before-completion这个Skill强制执行。它要求四个维度的验证：全量测试通过、手动验证功能正确性、确认无回归问题、验证输出完整可读。跳过任何一个维度都被视为流程违规

6.3.3 质量保障机制

■ 关于三条铁律的总结

三条铁律的设计哲学是"不给AI走捷径的机会"。在标准流程中，铁律不可妥协；在裁剪后的流程中（如Bug修复的三步精简流程），铁律仍通过其对应的Skill强制执行——测试类和调试类Skill不可移除。这体现了流程驱动的核心原则：质量关卡可以按需裁剪，但质量底线不可突破。

Part 04

Superpowers的安装与配置

6.4 Superpowers的安装与配置

环境要求与安装

目录

工作流配置

6.4.1 环境要求与安装

Superpowers支持多个AI编程工具平台（Claude Code、Cursor、Codex、OpenCode、Gemini CLI、GitHub Copilot CLI、TRAE等），本书配套实验使用TRAE。运行环境要求如下：

操作系统

Windows、macOS、Linux均可。本书配套实验环境为Windows，需安装Git Bash（随Git for Windows一起安装），在Git Bash中执行Superpowers相关命令；macOS和Linux下开箱即用，无需额外配置

Node.js

20.x及以上版本（Superpowers通过npx运行，安装后在终端执行“node --version”确认版本号）

Git

2.30及以上版本（用于工作树管理）

AI编程工具

需支持子智能体功能（本书使用TRAE）。TRAE原生支持子智能体执行功能，并且在实际开发中，一般推荐选择子智能体执行模式，以实现更好的上下文隔离和并行开发能力

6.4.1 环境要求与安装

Superpowers的安装命令如下（在TRAE中打开某个项目，在TRAE界面中的Git Bash终端中执行命令，每个项目中安装Superpowers，都需要单独执行一次安装命令）：

```
npx superpowers-zh --tool trae
```

该命令执行以下操作：

01 下载Superpowers中文版的最新发布包

02 根据指定的工具类型（--tool TRAE）自动配置Skill文件的存放路径和格式

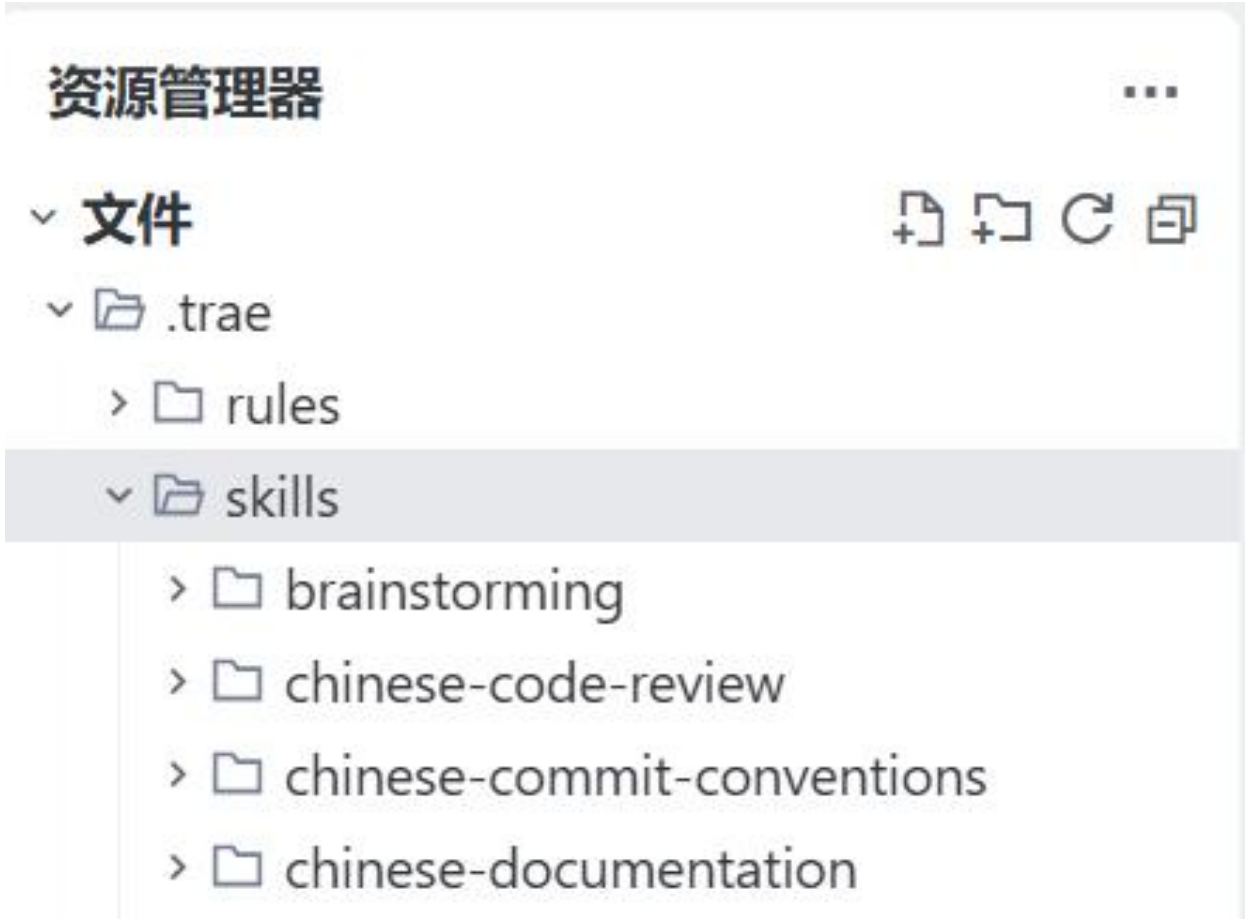
03 安装14个核心Skill到AI编程工具的Skill目录

04 生成配置文件，指定Skill搜索路径

在安装命令中，--tool参数指定目标AI编程工具，不同工具的Skill存放路径不同。TRAE的Skill目录通常位于项目根目录的“.trae/skills/”子目录下。

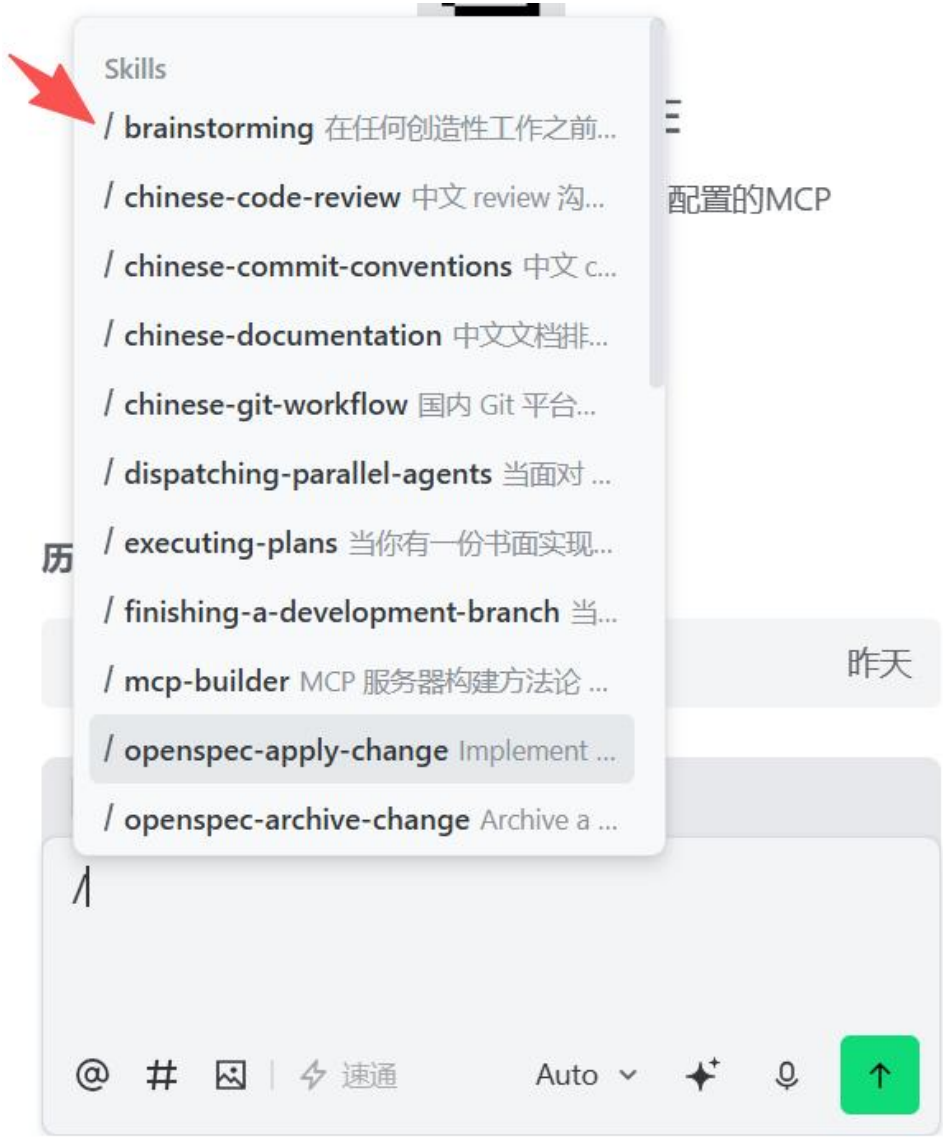
6.4.1 环境要求与安装

安装完成后，在TRAE界面左侧的“资源管理器”中（如图所示），检查当前项目的Skill目录下是否包含属于Superpowers的14个.md格式的Skill文件（比如brainstorming）。需要注意的是，Skill目录下还可能包含一些不属于Superpowers的Skill。



6.4.1 环境要求与安装

如果已经包含14个Skill文件，再测试这些Skill是否可用，测试方法是，在智能体交互界面的对话框中，输入斜杠“/”，在弹出的可用Skill列表中（如图所示），如果可以找到属于Superpowers的Skill（比如brainstorming），就说明Superpowers安装成功了。如果在可用Skill列表中没有找到属于Superpowers的Skill，请尝试重启AI编程工具TRAE，或者，新建一个任务（也就是对话），重新执行上述操作。



需要注意的是，Superpowers的部分功能（尤其是subagent-driven-development和dispatching-parallel-agents）强依赖于AI编程工具的子智能体能力。如果使用的AI编程工具不支持子智能体功能（如某些轻量级AI编辑器），这些Skill将无法正常运作，流程可能需要降级为单智能体顺序执行模式。降级后，多个子任务将在同一个智能体中按顺序执行，隔离性降低，但测试类和调试类Skill（TDD、完成验证）仍然有效。

6.4.2 工作流配置

Superpowers无需复杂的配置文件。它的"配置"本质上就是用户在每次会话开始时输入的标准化指令（或称为提示词）。这些指令告诉AI：本次会话使用哪些Skill、按照什么顺序执行、遵循哪些规则。以下是一个典型的新项目开发时在TRAE的智能体交互界面的对话框中输入的标准化指令：

```
## User

本对话中，你必须使用 Superpowers。Superpowers 是一套专门用于结构化软件开发的工具，由一系列的 skills 构成。

它们是你工作流程的一部分，你必须在每个阶段激活和使用它们。

使用以下流程：
1. 使用 brainstorming skill，采用苏格拉底式的提问方法，通过反复提问帮我理清缓存数据库的具体需求、数据结构和API设计。直到需求完全明确后才进入步骤2
2. 使用 using-git-worktrees skill，创建隔离的Git工作树环境，确保开发工作不污染主干。然后使用 writing-plans skill，根据 brainstorming 的结果，创建详细的实现计划，将整个开发过程分解为可以2-5分钟完成的小任务。在开发期间，必须使用 test-driven-development skill，采用其 RED-GREEN-REFACTOR 循环开发，不得跳过测试步骤
3. 使用 subagent-driven-development 或 dispatching-parallel-agents skill，对步骤3中的每个小任务，自动创建独立的实施任务，逐任务执行。只有当前任务完成后才进行下一个
4. 使用 requesting-code-review skill，对开发过程中完成的每个子任务，在合并前进行代码审查，确保代码质量
5. 使用 finishing-a-development-branch skill，合并到master分支，清理工作树

**重要规则**：
- 当我提出查询时，你必须使用 brainstorming skill 首先探索问题空间，不要急于生成代码
- 开始任何开发工作前，确保已设置正确的Node.js环境（`nvm use 20`）
- 所有开发工作都必须在对应的子目录中完成
- 绝对不要主动发起git提交、推送等改变仓库状态的操作
```

6.4.2 工作流配置

1. 从以上指令（提示词）可以看出，Superpowers的使用模式是：用户在每次会话开始时输入标准化指令，AI根据指令依次触发对应的Skill

2. 这种模式将"该怎么做"从每次交互的临场决策，转变为预设的标准化流程

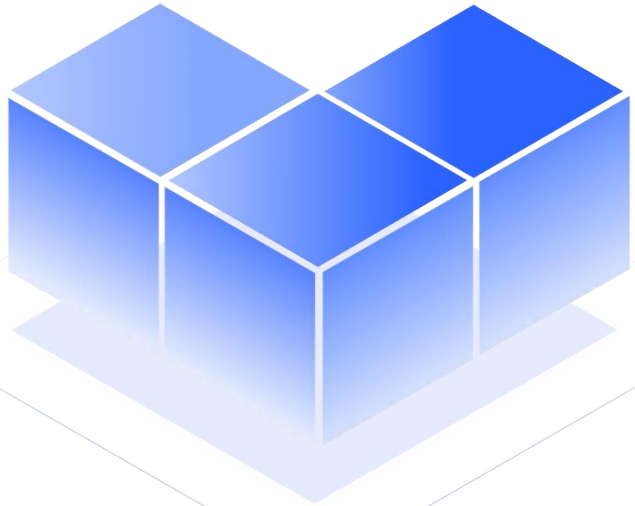
3. 需要特别说明的是，随着AI编程智能体（如TRAE）的能力不断增强，智能体往往会"抢活干"——上一步完成后尚未经过用户确认，就自动跳入下一步

4. 这种行为虽然出于"高效完成"的意图，却破坏了Superpowers流程中每一步的检查点意义

5. 因此，应该在提示词中明确要求TRAE严格按步骤执行，不得自行跳步。如果AI仍然跳步，可以在提示词中加入更强的约束——"在未完成当前步骤之前，禁止进入下一步。如果你发现自己正在写实现代码但需求澄清尚未完成，立即停止"

6.4.2 工作流配置

不同项目可以在此基础上裁剪，具体如下：



小型功能迭代

移除using-git-worktrees步骤

代码重构

保留test-driven-development和requesting-code-review，移除brainstorming和writing-plans

Bug修复

使用systematic-debugging + test-driven-development + verification-before-completion三步精简流程

裁剪流程时需要遵循一个原则：测试类和调试类Skill不可移除。无论流程如何裁剪，test-driven-development和verification-before-completion必须保留，它们是三条铁律的执行载体，跳过它们等于放弃流程驱动的核心价值。

Part 05

缓存数据库开发实践

6.5 缓存数据库开发实践

在正式进入七步流程之前，需要先了解项目的基础说明文件。项目说明书（AGENTS.md）是项目根目录下的一个Markdown文件，用于向编码智能体（如TRAE）描述项目背景，它告诉AI这个项目是什么、用了什么技术栈、有哪些开发约定。项目说明书（AGENTS.md）与Superpowers没有直接关系，它不是Superpowers生成的，也不属于Superpowers的Skill体系。但在使用Superpowers进行流程驱动开发时，项目说明书（AGENTS.md）中的项目背景信息，能帮助AI更好地理解项目上下文，从而在brainstorming和writing-plans阶段做出更合理的判断。

6.5 缓存数据库开发实践

以下是缓存数据库项目的项目说明书（AGENTS.md）完整内容（请把AGENTS.md放置到CacheDB项目的根目录下）：

```
# AGENTS.md

## 项目简介

这是一个Python缓存数据库（CacheDB），提供简单易用的键值对缓存服务。

## 技术栈

- Python 3
- pytest（测试框架）
```

6.5 缓存数据库开发实践

```
## 目录结构

cachedb/
├── src/cachedb/
│   ├── __init__.py         # 包初始化文件
│   ├── database.py         # CacheDB 核心实现
│   └── exceptions.py       # 自定义异常类
├── tests/
│   └── test_cachedb.py     # 测试文件
├── example/
│   └── demo.py             # 使用示例
├── docs/
│   └── superpowers/        # Superpowers生成的文档
│       ├── specs/          # 设计规格文件
│       └── plans/          # 实现计划文件
├── AGENTS.md               # 项目说明书
└── README.md               # 项目自述文件
```

6.5 缓存数据库开发实践

```
## 开发约定

- 遵循Python社区编码规范
- 所有代码修改前必须先编写测试
- 不得直接修改master分支

## AI编程行为规范

- 每次AI交互都必须按照Superpowers工作流规则处理
- 开始任何开发工作前，确保已设置正确的Node.js环境（`nvm use 20`）
- 所有开发工作都必须在对应的子目录中完成
- 绝对不要主动发起git提交、推送等改变仓库状态的操作
```

6.5 缓存数据库开发实践

1. 需求澄清与设计
2. 分支隔离
3. 任务规划
4. 子智能体执行
5. 测试驱动开发与验证
6. 代码审查与优化
7. 项目收尾与归档
8. 数据库运行效果与使用方式

6.5.1 需求澄清与设计

现在执行七步标准流程中的“步骤1：需求澄清（brainstorming）”。

为了避免TRAE迭代以后过于聪明抢活干，建议在提示词里面再强调一下，严格按照Superpowers的七个步骤去执行接下来的开发任务。因此，可以在TRAE的智能体交互界面的对话框中输入如下提示词并提交：

```
严格遵循Superpowers七步法串行开发Python缓存数据库，无确认绝不主动推进、不超前写代码、不跳过任何一个步骤。

固定流程顺序，缺一不可，仅完成当前步骤，等待我确认后再进入下一环。

1. 需求对齐时调用 /skill:brainstorming 苏格拉底追问敲定全部开发需求，输出需求文档，待确认
2. 环境隔离 /skill:using-git-worktrees 建立Git工作树/独立开发分支，确定项目目录，待确认
3. 任务拆解 /skill:writing-plans 拆分细粒度开发任务，明确文件、逻辑、校验方式，待确认
4. 逐任务开发 /skill:subagent-driven-development 单任务完成后自检复盘，不批量编写，待确认
5. TDD测试 /skill:test-driven-development 先写测试用例再实现代码，完成功能校验，待确认
6. 代码评审 /skill:requesting-code-review 排查漏洞、规范、性能问题并优化，待确认
7. 项目收尾 /skill:finishing-a-development-branch 完成整合、持久化、整理交付，流程结束

1.输出首行标注当前所处步骤
2.每步结束统一话术：本步骤完成，请确认进入下一步
3.未收到确认指令，仅限完善当前内容，禁止输出后续内容与代码
4.严禁自主提速、合并步骤、提前完成开发

现在开始第1步，逐条澄清本缓存数据库的需求。
```

6.5.1 需求澄清与设计

要强调的是，AI有时候会“抢活干”（即AI没有经过用户手动确认同意，就自动进入下一步开始工作），尽管在上面的提示词中，我们已经明确要求AI在执行每一步结束后，一定要经过用户确认才能进入下一步，但是，在实际应用过程中我们发现，AI有时候还是会出现“不经用户确认就自动进入下一步去执行”的情况

如果确实出现了“抢活干”的情况，你接受AI的结果即可，如果你想了解AI具体做了哪些事情，可以在“AI侧栏”中查看对话历史记录，了解AI到底自动做了哪些事情。这里我们假设AI不会“抢活干”，每步执行结束以后，都会严格等待用户手动确认才会进入下一步

上面的提示词提交以后，AI开始执行“步骤1：需求澄清”（注意，在TRAE中可能显示为“第1步：需求对齐”）。这时，AI会启动苏格拉底式追问（提问者为编码智能体，回答者为用户）。 由于大模型是概率模型，AI的每次行为都可能不同，在不同用户的对话中，AI的问题可能不同

6.5.1 需求澄清与设计

经过以上问题的追问，用户的需求从"我需要一个缓存数据库"这一句模糊描述，逐步明确为具有精确工程约束的需求规格。AI随后提出了方案选择：

> 我建议采用以下方案：简单的类字典接口，通过set/get/delete方法操作，底层用Python字典存储，在set时检查容量。你觉得这个方案如何?

用户确认后，AI会进一步自动展开设计确认（这个过程不需要用户参与回答，AI会自动完成确认过程），包括以下五个方面：

6.5.1 需求澄清与设计

整个需求澄清阶段产生了约15轮对话（7个追问 + 1个方案选择 + 5节设计确认 + 1次最终确认），从一句模糊需求逐步收敛到完整的工程规格

需求澄清完成后，Superpowers将确认的设计规格自动保存到项目的docs/superpowers/specs/目录下。文件命名格式为YYYY-MM-DD-项目名-design.md。在缓存数据库项目中，该文件为2026-05-20-cachedb-design.md，内容包含项目概述、核心组件架构、数据流设计、API详细规格、异常处理策略和测试策略等。specs文件是后续所有开发的"需求基线"——所有实现计划、测试用例和代码审查都以该文件为参考标准

需求澄清完成后，AI会提示你“本步骤完成，请确认进入下一步”，这时可以在智能体交互界面的对话框中输入“确认进入下一步”并提交即可，AI会自动进入“步骤2：分支隔离”

6.5.2 分支隔离

执行七步标准流程中的“步骤2：分支隔离（using-git-worktrees）”。

AI进入到分支隔离这一步以后（注意：TRAE中可能显示为“第2步：环境隔离”），AI会自动激活using-git-worktrees这个Skill开始建立一个隔离的工作区。AI会提示你“是否要创建一个隔离的 worktree”，你只要在智能体交互界面的对话框中输入提示词“创建worktree”并提交即可。

由于缓存数据库是新项目，该步骤中AI会自动执行以下操作：

```
cd /path/to/cachedb
git init
git add .
git commit -m "Initial commit: empty project structure"
```

Git Worktree的核心价值在于环境隔离，即每个开发任务在独立的工作树中执行，不同任务的代码变更互不干扰。如6.3.1节所述，这与第5章OpenSpec中通过规范文件实现"需求隔离"的思路一脉相承，但Superpowers将隔离从"文档层面"推进到了"代码层面"。

分支隔离步骤完成以后，AI会提示你“本步骤完成，请确认进入下一步”，这时可以在智能体交互界面的对话框中输入“确认进入下一步”并提交即可，AI会自动进入“步骤3：任务规划”。

6.5.3 任务规划

执行七步标准流程中的“步骤3：任务规划（writing-plans）”。

01

进入“步骤3：任务规划”以后（注意，TRAE中可能显示为“第3步：任务拆解”），AI会自动激活writing-plans这个Skill，基于specs文件中的设计规格，将整个缓存数据库基础功能的开发拆解为8个独立子任务。每个子任务都标注了精确的文件路径、预估难度和任务依赖关系。

02

任务规划的输出保存在docs/superpowers/plans/目录下，文件命名格式与specs文件对应：YYYY-MM-DD-项目名-implementation.md。在缓存数据库项目中，该文件为2026-05-20-cachedb-implementation.md。

6.5.3 任务规划

缓存数据库基础功能开发的完整8个子任务清单如表所示。

编号	任务描述	涉及文件	预估难度	依赖
1	创建项目目录结构和__init__.py	src/cachedb/__init__.py	简单	无
2	实现自定义异常类	src/cachedb/exceptions.py	简单	1
3	实现CacheDB核心类框架（__init__方法）	src/cachedb/database.py	简单	2
4	实现set方法（含容量检查和异常处理）	src/cachedb/database.py	中等	3
5	实现get方法（含键不存在处理）	src/cachedb/database.py	简单	3
6	实现delete方法（含键不存在处理）	src/cachedb/database.py	简单	3
7	实现clear和size方法	src/cachedb/database.py	简单	3
8	编写完整测试用例	tests/test_cachedb.py	中等	4,5,6,7

6.5.3 任务规划

plans文件的组织方式体现了Superpowers"小步快跑"的核心理念：

这种粒度确保每个子智能体在执行时不会因任务过于复杂而产生偏差。

任务规划步骤完成以后，AI会提示你“本步骤完成，请确认进入下一步”，这时可以在智能体交互界面的对话框中输入“确认进入下一步”并提交即可，AI会自动进入“步骤4：子智能体执行”。

6.5.4 子智能体执行

执行七步标准流程中的“步骤4：子智能体执行（subagent-driven-development）”。

进入“步骤4：子智能体执行”以后（注意，在TRAE中可能显示未“第4步：逐任务开发”），AI会自动激活subagent-driven-development这个Skill，按照plans文件中定义的8个子任务顺序，逐一创建独立的子智能体执行。每个子智能体遵循"读取任务→阅读specs→执行TDD→返回结果"的标准流程

需要注意的是，在AI自动执行过程中，可能会出现提示“pip install pytest”（安装单元测试框架pytest，用来给你的代码写测试用例），可以点击“运行”即可。还可能会出现其他提示安装相关的Python第三方库，点击“运行”即可

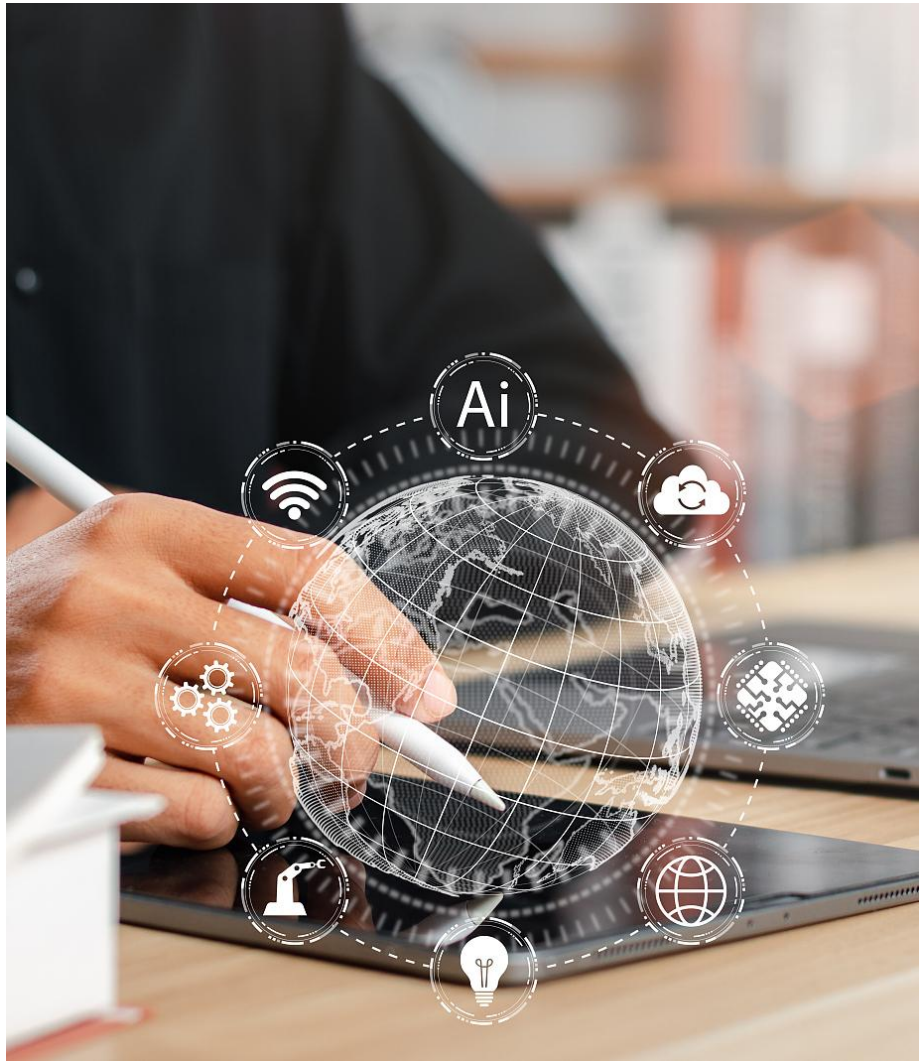
本步骤完成以后，AI会提示你“本步骤完成，请确认进入下一步”，这时可以在智能体交互界面的对话框中输入“确认进入下一步”并提交即可，AI会自动进入下一个步骤

6.5.5 测试驱动开发与验证

执行七步标准流程中的“步骤5：测试驱动开发（test-driven-development）”。

所有8个子任务均遵循TDD的RED-GREEN循环执行。所有8个子任务完成后，缓存数据库的测试套件包含15个测试用例（基础功能阶段），覆盖了全部5个核心方法的正常场景、边界场景和异常场景

AI会自动运行全部15个测试，15个测试全部通过，基础功能开发阶段完成



6.5.5 测试驱动开发与验证

结合这个案例，可以从两个维度进一步理解TDD在AI编程中的实际效果：

（1）将模糊需求转化为可验证契约。需求文档中的"set操作应该拒绝空键"是一种模糊的自然语言描述。TDD将其转化为精确的、机器可验证的契约，具体如下：

```
def test_set_invalid_key():
    db = CacheDB()
    try:
        db.set("", "value")
        assert False, "期望抛出InvalidKeyError异常"
    except InvalidKeyError:
        pass  # 预期行为
```

这种转化消除了需求与实现之间的"翻译误差"，测试要么通过，要么失败，没有中间状态。如6.3.1节步骤5所述，这正是TDD RED-GREEN循环的核心机制——先定义"什么算完成"，再开始实现。

（2）为后续迭代提供安全网。假设后续增加新功能，修改了存量代码，测试能够保证存量功能不受影响。

6.5.6 代码审查与优化

执行七步标准流程中的“步骤6：代码审查（requesting-code-review）”。

所有子任务完成后，AI激活requesting-code-review这个Skill进行代码审查。审查子智能体对全部代码进行了系统性检查，发现了以下问题：KeyError命名冲突和set方法容量检查的逻辑顺序错误

所有问题修复后，重新运行全部15个测试，全部通过。代码审查通过，进入合并阶段

本步骤完成以后，AI会提示你“本步骤完成，请确认进入下一步”，这时可以在智能体交互界面的对话框中输入“确认进入下一步”并提交即可，AI会自动进入下一个步骤

6.5.7 项目收尾与归档

执行七步标准流程中的“步骤7：收尾归档（finishing-a-development-branch）”。

进入本步骤以后，AI会自动激活finishing-a-development-branch这个Skill执行项目收尾，将开发分支的所有变更合并到master分支，AI会自动执行如下命令：

```
git checkout master
git merge feature/cachedb-basic
```

然后，删除临时工作树目录，释放磁盘空间。建议定期运行git worktree list检查是否有遗留的孤立工作树，避免磁盘空间被长期占用。

需要注意的是，这个步骤中，AI也有可能会询问你采用什么方案完成收尾工作，你可以在对话框中输入提示词“在本地合并回 main”并提交。

最后，AI输出最终交付总结。缓存数据库的基础功能开发至此正式完成。

6.5.8 数据库运行效果与使用方式

经过前七个小节的完整流程，缓存数据库（CacheDB）已经交付完成。本小节演示CacheDB的运行效果，并澄清其嵌入式数据库的定位——它并非像MySQL或Redis那样以独立服务进程运行，而是以库的形式嵌入到应用程序中。

■ 嵌入式数据库的定位

在6.5.1节的需求澄清（或称为需求对齐）阶段，brainstorming步骤明确了一个关键设计决策：CacheDB采用Python API方式提供接口，而非命令行操作方式。这意味着CacheDB是一个嵌入式数据库（Embedded Database）——它以库的形式嵌入应用程序进程内部，由应用程序直接调用，不存在独立的服务进程

这一设计可能出乎部分读者的意料。提到"数据库"，许多人首先想到的是MySQL、PostgreSQL或Redis，这些数据库以独立的后台服务运行，客户端通过网络协议（如MySQL的TCP 3306端口、Redis的TCP 6379端口）与之通信。然而，嵌入式数据库走的是完全不同的路线：它没有独立进程，没有网络监听，没有连接字符串，应用程序只需在代码中导入库即可直接使用

6.5.8 数据库运行效果与使用方式

■ **嵌入式数据库的定位**

嵌入式数据库在工程实践中并不罕见，许多知名项目都采用了这一模式。下表列出了几种典型的嵌入式数据库及其应用场景。

数据库	语言生态	典型应用项目
SQLite	C/C++	Android系统数据存储、Firefox浏览器书签、各类桌面应用
BoltDB	Go	etcd（分布式键值存储，Kubernetes核心组件）
LevelDB	C++	Chrome浏览器IndexedDB底层存储、Bitcoin Core区块索引
H2 Database	Java	Spring Boot内嵌数据库、各类Java微服务测试环境

从上表可以看出，嵌入式数据库并非"玩具"，比如，etcd作为Kubernetes集群的核心组件，支撑着全球数以百万计的生产环境集群，而它的底层存储正是BoltDB；Chrome浏览器使用LevelDB存储用户的IndexedDB数据，服务数十亿用户。这些案例说明，嵌入式数据库在合适的场景下能够支撑严肃的生产级应用。CacheDB虽然是一个轻量级实现，但其架构模式与这些工业级项目一脉相承——将存储能力以库的形式嵌入应用，省去独立服务的部署和运维成本。

6.5.8 数据库运行效果与使用方式

■ 嵌入式数据库的定位

需要说明的是，**嵌入式数据库与独立服务型数据库并非竞争关系，而是互补关系**。

独立服务型数据库（如MySQL、Redis）

适合多应用共享、高并发、需要独立运维的场景

嵌入式数据库

适合单应用内嵌、零运维、低延迟的场景。CacheDB选择了后者，这使得它可以直接被Python应用导入使用，无需安装、无需配置、无需启动服务

6.5.8 数据库运行效果与使用方式

■ 使用方式

CacheDB的使用方式非常简洁，只要在代码中导入CacheDB类及其异常类，即可创建实例并调用方法。

在本案例中，AI在完成七步流程的收尾归档后，自动创建了examples目录并生成了demo.py演示文件（注意，你的TRAE中，目录名字和文件名字可能不一样），用于展示CacheDB的全部功能

如果AI没有自动生成这样的示例文件，开发者也可以在对话框中输入一条提示词并提交（例如"请创建一个示例程序，演示CacheDB的所有功能"），AI就会生成类似demo.py的演示文件，覆盖创建实例、设置键值对、获取值、更新键、删除键、异常处理和清空缓存等核心操作

6.5.8 数据库运行效果与使用方式

■ 运行效果

在TRAE的Git Bash终端中，在项目根目录下执行“python examples/demo.py”（注意，你的TRAE中，可能目录名字不一样），即可看到CacheDB的运行效果（如图所示）。

Part 06

协同工作流与方法论选择

6.6 协同工作流与方法论选择

目录

01 与OpenSpec的三段式分工模式

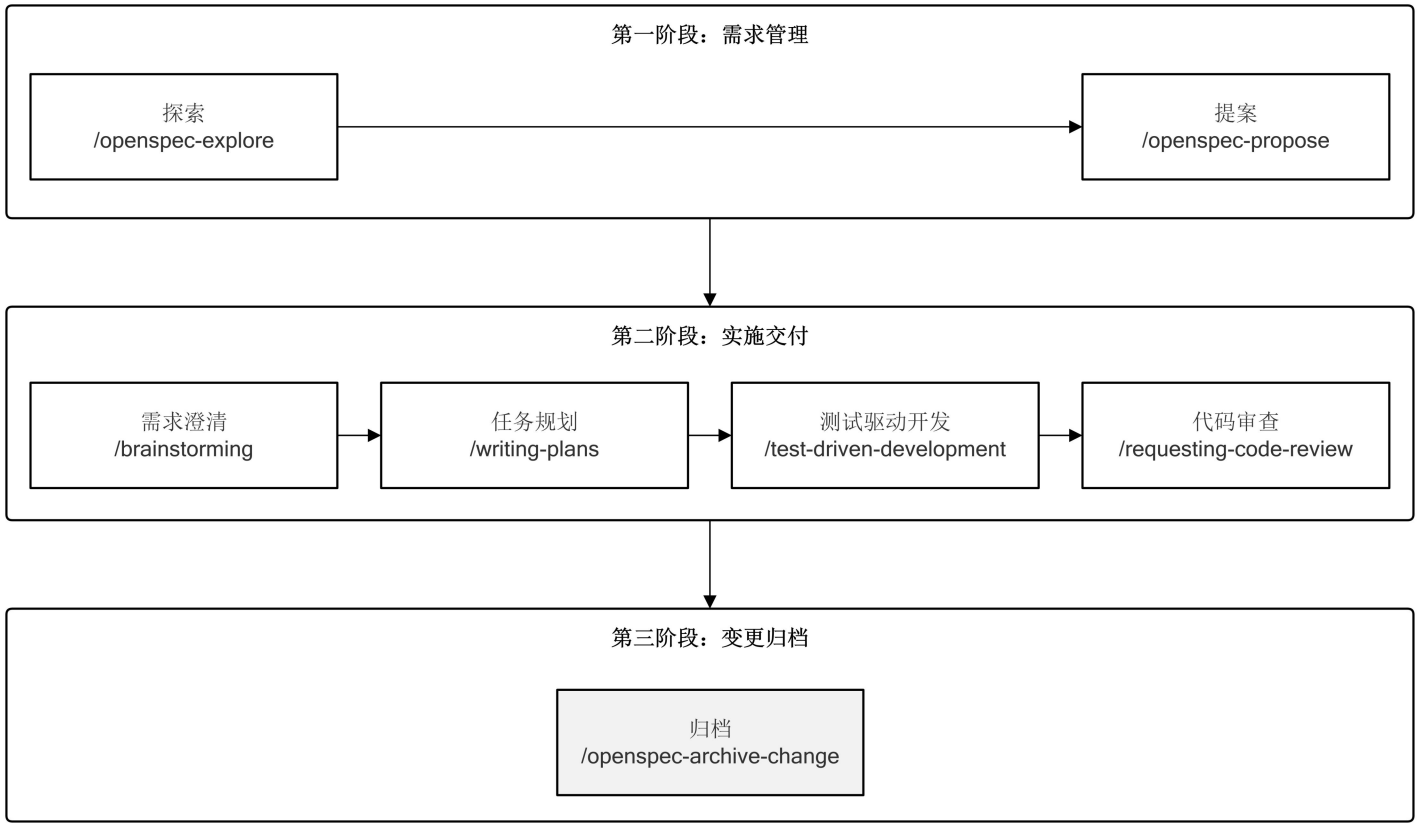
02 数据流转与接口对接

03 三种方法论对比矩阵

04 适用场景选择与混合策略

6.6.1 与OpenSpec的三段式分工模式

Superpowers与OpenSpec并非互斥选择，而是可以协同使用。OpenSpec的强项是需求的可追溯性（谁在什么时候因为什么原因提出了什么需求），Superpowers的强项是实现的质量可控性（每行代码有测试覆盖、每次变更经过审查）。两者分别解决了"需求管理"和"过程管理"两个正交的维度，协同使用可以发挥各自优势。在两者结合的场景中，建议采用以下三段式分工模式，如图所示。



6.6.1 与OpenSpec的三段式分工模式

Superpowers与OpenSpec的三段式系统模式具体如下：

第一阶段（需求管理）

OpenSpec负责"做什么"。使用openspec-explore探索需求空间，openspec-propose生成标准化的需求提案——明确功能范围、验收标准、技术约束。这一段对应第5章介绍的OpenSpec四步流程中的前两步（explore和propose），产出的是经过结构化描述的proposal文件

第二阶段（实施交付）

Superpowers负责"怎么做"。基于OpenSpec的输出启动Superpowers七步流程——brainstorming细化技术方案，writing-plans拆分任务，TDD保障质量，code-review审查代码。这一段将OpenSpec的proposal文件作为brainstorming的输入，但不重复探索"做什么"，而是聚焦于工程实现

第三阶段（变更归档）

OpenSpec负责"记下来"。使用openspec-archive将本轮开发的变更记录归档到知识库，供后续需求和决策参考。这一段对应第5章介绍的OpenSpec四步流程中的最后一步（archive），将Superpowers的交付物（代码变更清单、测试报告、审查记录）纳入变更管理体系

6.6.2 数据流转与接口对接

在实际的协同工作流中，OpenSpec和Superpowers之间的数据流转是一个关键问题。两者的对接接口如下：

- OpenSpec → Superpowers方向

OpenSpec的proposal文件（包含功能描述、验收标准、技术约束）作为Superpowers brainstorming步骤的输入。brainstorming不重新探索"做什么"（这个已经在OpenSpec阶段完成了），而是聚焦于"怎么做"——技术方案选型、架构设计决策、关键技术风险识别

在缓存数据库项目中，如果先经过OpenSpec阶段，proposal文件中已经明确了"支持TTL过期策略"的需求和"过期精度为秒级"的验收标准，Superpowers的brainstorming就可以直接跳过"是否需要TTL"的讨论，聚焦于"TTL的底层数据结构选型（时间轮 vs 有序集合）"等工程决策

6.6.2 数据流转与接口对接

■ Superpowers → OpenSpec方向

Superpowers完成后的交付总结（代码变更清单、测试覆盖率报告、审查记录）作为OpenSpec archive步骤的输入，生成归档记录。归档记录的典型格式如下：

```
变更编号：CHG-2026-003
变更描述：新增缓存TTL过期策略
关联proposal：PROP-2026-003
代码变更：database.py（新增_ttl_store属性和_cleanup方法）、test_cachedb.py（新增5个TTL相关测试）
测试覆盖：15→20个测试用例，覆盖率100%
审查记录：无阻断性问题，2个建议性问题已记录
```

6.6.2 数据流转与接口对接

■ Superpowers → OpenSpec方向

这种结构化的归档记录使得任何团队成员都可以快速了解"这个功能什么时候加的、为什么加、改了哪些文件、测试覆盖如何"，这正是第5章强调的"变更可追溯性"。

为避免两个工具的文件体系混乱，建议采用以下目录约定：

```
project/
├── openspec/          # OpenSpec工作目录
│   ├── proposals/     # 需求提案
│   └── archive/       # 归档记录
├── docs/
│   └── superpowers/   # Superpowers工作目录
│       ├── specs/     # 设计规格文件
│       └── plans/     # 实现计划文件
└── src/               # 源代码
```

6.6.2 数据流转与接口对接

- Superpowers → OpenSpec方向

01

OpenSpec的文件存放在openspec/目录下，Superpowers 的 文 件 存 放 在 docs/superpowers/目录下，两者物理隔离但逻辑关联——proposal文件中的需求ID与specs文件中的需求描述通过引用关系关联

02

两者在"需求探索"和"设计"阶段有一定的功能重叠。在实践中，建议用OpenSpec的explore主导需求探索（面向业务需求），用Superpowers的brainstorming细化技术实现方案（面向工程细节）。这个分工原则可以避免"同一件事在两个工具中都做一遍"的浪费

6.6.3 三种方法论对比矩阵

为了帮助读者在三种方法论之间做出选择，下表从多个维度进行了系统对比。

对比维度	基于提示词（第4章）	规范驱动/OpenSpec（第5章）	流程驱动/Superpowers（第6章）
核心关注	单次交互质量	需求一致性	过程可靠性
适用规模	个人开发者	小型团队（2-5人）	企业级团队（5+人）
学习成本	低	中	中高
Token消耗	低	中	高（4-6倍于提示词驱动）
需求追溯	无	完整（proposal→archive）	设计文档（specs/plans）
测试覆盖	依赖开发者习惯	依赖开发者习惯	强制TDD
代码审查	依赖开发者习惯	可选	强制
环境隔离	无	无	强制Git Worktree
适用场景	快速原型、小修小改、单个文件	多人协作、需求频繁变更、有合规需求	新项目、核心功能、长期维护的基础组件
强制约束力	无	中等（通过spec文件约定）	强（流程不可跳过）

6.6.3 三种方法论对比矩阵

值得注意的是，上页表中"Token消耗"维度显示流程驱动的成本约为提示词驱动的4-6倍，但这一数字需要结合产出物来理解。

以缓存数据库基础功能开发为例，完整七步流程总消耗约12万Token，而基于提示词的AI编程大约需要消耗4万Token，差值（8万Token）换来的是完整的测试套件、代码审查记录、设计规格文档和实现计划文档。以主流大模型API定价估算，8万Token约折合人民币2-4元，对于需要100%测试覆盖的任务而言，这个成本是值得的。

6.6.4 适用场景选择与混合策略

基于对比矩阵，面对具体的开发任务时，建议采用以下决策框架：

- **任务分类决策。**

不同任务可以采用不同的方法论，具体如下：

1. "一句话任务"（修个小bug、改个配置、加个日志），可以采用基于提示词的AI编程。这些任务的复杂度不足以支撑任何流程开销。直接给出约束明确的提示词即可
2. "独立新功能"（单文件、明确需求、无协作），可以采用“基于提示词 + 可选的TDD”。如果对质量要求较高，可以在提示词中加入"先写测试再实现"的约束
3. "多人协作功能"（多文件、需讨论方案、有合规需求），可以采用规范驱动（OpenSpec）。用proposal和specs文件确保团队对需求的理解一致
4. "核心模块/新项目"（关键功能、长期维护、需要100%测试覆盖），可以采用流程驱动（Superpowers）。完整七步流程的成本在这些场景中是值得的
5. "复杂企业项目"（既有协作需求又有质量要求）可以采用“ 规范驱动 + 流程驱动”组合。OpenSpec管需求，Superpowers管实现，三段式分工

6.6.4 适用场景选择与混合策略

■ **按需裁剪流程步骤**

即使选择了流程驱动，也不必每次都走完整七步。Bug修复使用三步精简流程（systematic-debugging → test-driven-development → verification-before-completion），Token消耗减少约50%

代码重构保留TDD和code-review，移除brainstorming和writing-plans，Token消耗减少约40%。这正是"方法论的复杂度应与任务的复杂度成正比"的具体体现

6.6.4 适用场景选择与混合策略

■ **关键原则**

方法论的复杂度应与任务的复杂度成正比。

"杀鸡用牛刀"不仅浪费Token和时间，更危险的是会导致团队产生"流程疲劳"——开发者感到"流程比代码还多"，从而失去对方法论的信任

另一方面，该用牛刀的场合不用牛刀，可能导致质量失控。找到"刚好够"的流程粒度，是AI编程方法论落地的关键所在

本章小结

01 本章聚焦AI编程方法论第三层——流程驱动的AI编程，以Superpowers为核心框架，通过标准化工作流与强制性质量关卡，实现企业级AI编程的过程可靠、质量可度量

02 该框架采用Agent集成层、工作流层、技能层三层架构，依托七步标准化开发流程与14项核心技能规范编程行为，并确立测试先行、根因分析、完成验证三条质量铁律，搭配AGENTS.md、设计规格、实现计划等完善的项目文件体系，筑牢工程质量底线

03 同时，缓存数据库实战案例验证了流程的落地可行性，且Superpowers可与OpenSpec适配三段式分工协同模式。纵观整套AI编程方法论，提示词驱动优化单次交互质量，规范驱动保障需求一致性，流程驱动严控过程可靠性，三者形成完整方法论工具箱

04 实践中需结合任务特征灵活组合使用，而非套用统一流程。流程驱动的核心是尊重软件工程规律，而非约束AI能力，始终坚守清晰需求、明确边界、可测试验收的软件工程核心原则

谢谢！

林子雨　副教授　厦门大学

附录A：主讲教师林子雨简介

主讲教师：林子雨

单位：厦门大学计算机科学与技术系

E-mail: ziyulin@xmu.edu.cn

个人网页：http://dblab.xmu.edu.cn/post/linziyu

数据库实验室网站：http://dblab.xmu.edu.cn

- 林子雨，男，1978年出生，博士（毕业于北京大学），全国高校知名大数据与人工智能教师，入选“2021年高校计算机专业优秀教师奖励计划”。现为厦门大学计算机科学与技术系副教授，厦门大学信息学院实验教学中心主任，曾任厦门大学信息科学与技术学院院长助理、晋江市发展和改革局副局长。中国计算机学会数据库专业委员会执行委员，中国计算机学会信息系统专业委员会执行委员。
- 国内高校首个“数字教师”提出者和建设者，厦门大学数据库实验室负责人，厦门大学云计算与大数据研究中心主要建设者和骨干成员，2013年度、2017年度、2020年度、2023年度和2026年度厦门大学教学类奖教金获得者，荣获2024年福建省高等教育教学成果奖特等奖（个人排名第七）、2022年福建省高等教育教学成果奖特等奖（个人排名第一）、2018年福建省高等教育教学成果奖二等奖（个人排名第一）、2018年国家精品在线开放课程、2021年国家级线上一流本科课程、2020年国家级线上一流本科课程。编著出版了20余本大数据与人工智能系列教材，被国内1000多所高校采用，3本教材入选教育部“十四五”普通高等教育本科国家级规划教材；建设了国内高校首个大数据课程公共服务平台，为教师教学和学生学习大数据课程提供全方位、一站式服务，年访问量超过400万次，累计访问量超过3200万次。大数据系列MOOC课程入选“2023年教育部国家智慧教育公共服务平台应用典型案例”。2025年发布4个大模型科普报告，全网累计浏览量超过1000万，并应邀为高校、政府和企业做170余场大模型讲座。
- 主要研究方向为数据库、数据仓库、数据挖掘、大数据、云计算和物联网，并以第一作者身份在《软件学报》《计算机学报》和《计算机研究与发展》等国家重点期刊以及国际学术会议上发表多篇学术论文。作为项目负责人主持的科研项目包括1项国家自然科学青年基金项目(No.61303004)、1项福建省自然科学青年基金项目(No.2013J05099)和1项中央高校基本科研业务费项目(No.2011121049)，主持的教改课题包括1项2016年福建省教改课题、1项2016年教育部产学协作育人项目、1项2024年教育部产学协作育人项目。

附录B：大数据学习路线图

大数据学习路线图访问地址：http://dblab.xmu.edu.cn/post/10164/

附录C：林子雨大数据系列教材

了解全部教材信息：http://dblab.xmu.edu.cn/post/bigdatabook/

附录D：林子雨人工智能通识系列教材

厦门大学林子雨人工智能通识系列教材

被众多高校采用，满足不同高校差异化教学需求

了解全部教材信息：http://dblab.xmu.edu.cn/post/ai-book/

附录E：林子雨Python系列教材

了解全部教材信息：https://dblab.xmu.edu.cn/post/python-books/

谢谢！

林子雨　副教授

厦门大学