

Spec-Kit实战：AI编程全流程落地

个人备份 · 导出时间 2026-08-13T08:57:23.477Z

目录

版权信息
版 权
内 容 提 要
前 言
资源与支持
第一部分 规范驱动开发的理论基础与工具准备
第1章 AI编程的范式转移
1.1 氛围编程的局限与挑战
1.1.1 局限一：代码质量如沙上建塔
1.1.2 局限二：可维护性被严重透支
1.1.3 局限三：团队协作沦为“各自即兴”
1.1.4 局限四：结果不确定性
1.2 SDD：新一代软件工程方法论
1.2.1 SDD的定义
1.2.2 SDD的三根支柱
1.2.3 为什么现在需要SDD——AI时代的必然选择
1.2.4 SDD的适用场景与边界
1.3 SDD核心价值：确定性、可维护性与团队协作
第2章 深入理解Spec-Kit
2.1 Spec-Kit的定位
2.1.1 脚手架：零摩擦的项目启动
2.1.2 工作流引擎：从宪法到实现的可执行流程
2.1.3 工具包：技术栈中立与AI可选性
2.2 核心概念：宪法、规范、计划与任务
2.2.1 宪法
2.2.2 规范
2.2.3 计划
2.2.4 任务
2.2.5 四者的关系
第3章 环境配置与AI编程代理
3.1 安装与配置Spec-Kit
3.1.1 安装specify

3.1.2 验证系统依赖项是否完善
3.2 认识与安装CodeBuddy
3.2.1 CodeBuddy是什么
3.2.2 安装CodeBuddy CLI
3.3 其他AI编程代理
第二部分 规范驱动开发实战进阶
第4章 【基础】组件驱动开发：用户名片组件
4.1 项目初始化及宪法确立
4.1.1 初始化项目
4.1.2 首次启动CodeBuddy
4.1.3 确立项目宪法
4.2 编写组件规范
4.2.1 输入需求
4.2.2 审视需求是否被准确理解
4.3 制定技术方案
4.3.1 输入技术方案
4.3.2 生成技术方案相关文件
4.4 生成高质量Vue组件
4.4.1 拆解任务清单
4.4.2 执行实现
4.4.3 验证项目效果
4.5 回顾：完整流程
第5章 【进阶】状态管理：任务看板
5.1 项目初始化及宪法确立
5.2 编写任务看板规范
5.3 制定技术方案
5.4 任务拆解与实现：从列表到看板
5.4.1 拆解任务
5.4.2 执行实现
5.4.3 在浏览器中验证
第6章 【基础】构建后端API与前后端联调
6.1 项目初始化及宪法确立
6.1.1 初始化后端API服务项目
6.1.2 编写后端API服务项目宪法
6.2 定义后端API契约
6.2.1 后端API契约：核心作用与价值解析
6.2.2 对齐契约与前端数据模型

6.2.3 输入API契约描述
6.2.4 审视契约：前后端一致性检查
6.2.5 确认契约并停止操作
6.3 规划技术方案
6.4 拆解与实现任务
6.4.1 拆解任务
6.4.2 执行实现
6.4.3 验证后端API服务
6.4.4 验证通过并终止操作
6.5 前后端联调：替换前端数据层
6.5.1 理解数据层替换
6.5.2 配置前端开发代理
6.5.3 阅读API契约——了解后端接口
6.5.4 创建api.ts替换storage.ts
6.5.5 适配BoardContext的异步调用
6.5.6 前后端联调验证
第7章 【进阶】安全加固与前后端联调
7.1 更新宪法：增加安全原则
7.1.1 更新后端API服务项目宪法
7.1.2 更新前端任务看板项目宪法
7.1.3 确认宪法并停止操作
7.2 扩充规范：认证与权限
7.2.1 理解安全加固对规范的影响
7.2.2 更新后端API服务项目规范
7.2.3 更新前端任务看板项目规范
7.2.4 确认规范并停止
7.3 更新方案：安全模块的设计
7.3.1 理解安全模块的技术选型
7.3.2 更新后端API服务项目计划文件
7.3.3 更新前端任务看板项目计划文件
7.3.4 确认计划并停止
7.4 前后端功能拆解与实现
7.4.1 后端API服务项目：拆解任务
7.4.2 后端API服务项目：实现任务
7.4.3 后端API服务项目：验证认证接口
7.4.4 同步后端API契约到前端
7.4.5 前端项目：拆解任务
7.4.6 前端项目：实现安全任务
7.4.7 前端项目：验证安全认证功能

第三部分 存量系统改造与未来展望
第8章 存量系统Spec-Kit开发实践：存量系统设计知识和代码理解
8.1 存量系统面临的独特挑战
8.1.1 文档缺失、代码错误的困境
8.1.2 隐含业务逻辑与“神秘代码”现象
8.1.3 系统脆弱，开发者不敢轻易重构
8.1.4 SDD对存量系统的价值再审视
8.2 存量系统理解：从代码到知识
8.2.1 静态分析：模块结构、调用链、数据流
8.2.2 动态分析：运行时行为捕获
8.2.3 架构与集成分析
8.2.4 数据模型理解
8.3 从存量系统中提取规范：逆向工程实践
8.3.1 API规范提取
8.3.2 业务规则与行为规范提取
8.3.3 数据契约与领域模型规范
8.3.4 流程规范提取（工作流提取）
8.4 规范验证：确保规范与现有系统一致
8.4.1 规范与实现的一致性测试
8.4.2 构建回归测试基线
8.4.3 处理歧义与非确定性行为
8.5 在存量系统中增量落地Spec-Kit
8.5.1 选择首个子域
8.5.2 建立规范基线固定系统基准行为
8.5.3 双轨工作流
8.5.4 渐进式重构：用规范驱动安全替换旧模块
8.5.5 组织与协作建议
8.6 综合案例：为某电商订单系统引入SDD
8.6.1 系统背景与痛点
8.6.2 规范提取过程演示
8.6.3 验证与问题发现
8.6.4 新需求开发：使用Spec-Kit流程新增“订单拆分”功能
8.6.5 成果与经验总结
8.7 常见陷阱与应对策略
8.7.1 过度提取：规范膨胀症
8.7.2 忽略外部依赖的行为变化
8.7.3 团队成员的抵触（“写规范不如改代码”）
8.7.4 规范与代码长期失步

8.8 小结
第9章 从氛围编程到SDD，必然之路与未来图景
9.1 氛围编程的局限性：繁荣表象下的暗流
9.2 SDD：系统性突破每一个局限
9.3 展望：SDD的下一站
9.4 启程建议：如何从你的下一个PR开始拥抱SDD
结语 秩序不是创造力的敌人，而是规模化的朋友

版权信息

书名：Spec-Kit实战：AI编程全流程落地

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

版　　权

著　　黄桂钊　茹炳晟

责任编辑　卜一凡

人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

邮编　100164　**电子邮件**　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

读者服务热线：(010)81055410

反盗版热线：(010)81055315

内 容 提 要

规范驱动开发（SDD）是软件开发领域中应对AI编程代理挑战的方法论。本书聚焦SDD，即如何在AI编程代理普及的背景下构建完整知识体系，帮助提升代码质量、可维护性与团队协作效率。全书构建从理论到实践的完整知识体系：第一部分系统讲解SDD的核心价值、Spec-Kit的核心概念，介绍环境配置与AI编程代理；第二部分通过组件驱动开发、状态管理、构建后端API与前后端联调、安全加固与前后端联调等实战案例，展示如何运用Spec-Kit构建复杂应用系统；第三部分探讨存量系统的改造方案，并对SDD的未来发展趋势进行分析，帮助读者理解从氛围编程到SDD的必然之路。

本书适合各类软件开发人员（包括前端开发工程师、后端开发工程师、全栈开发工程师、软件架构师等）、高校计算机相关专业师生阅读。无论是软件开发领域的新手，还是经验丰富的资深工程师，都能学到实用的开发技巧。

前　言

随着人工智能技术的飞速发展，软件开发领域正经历着前所未有的变革。AI编程代理如雨后春笋般涌现，极大地提升了代码生成效率，但也带来了新的挑战。如何在享受AI带来的便捷的同时，提升代码质量、可维护性和团队协作效率？这正是本书要探讨的核心问题。

全书构建了从理论到实践的完整知识体系：第一部分聚焦规范驱动开发（SDD）的理论基础与工具准备，从AI编程的范式转移切入，系统讲解Spec-Kit，并通过介绍环境配置与AI编程代理，助力读者快速掌握SDD的入门技巧；第二部分通过前端与后端实战案例，介绍如何运用Spec-Kit构建复杂应用系统，涵盖组件驱动开发、状态管理、构建后端API与前后端联调、安全加固与前后端联调等实战案例；第三部分探讨存量系统改造，并对SDD的未来发展趋势进行分析，帮助读者理解从氛围编程到SDD的必然之路。

本书具有以下特色：一是理论与实践相结合，既深入讲解SDD的理论基础，又通过丰富的实战案例帮助读者掌握具体操作方法；二是循序渐进，从基础概念到高级应用，逐步引导读者掌握SDD的精髓；三是实用性强，书中介绍的工具、方法和技巧可以直接应用于项目开发；四是具有前瞻性，对SDD的未来发展趋势进行了深入探讨，帮助读者把握行业发展方向。

本书适合各类软件开发人员阅读，包括前端开发工程师、后端开发工程师、全栈开发工程师、软件架构师等，也适合作为高校计算机相关专业的教学参考用书。无论你是软件开发领域的新手，还是经验丰富的资深工程师，都能从本书中学到实用的开发技巧并获得启发。

最后，为了帮助读者更好地学习本书，我们提供了丰富的配套资源，包括书中所有案例的源代码、相关工具安装包等。读者可以通过“资源与支持”页的指引获取这些资源。

资源与支持

资源获取

本书提供如下资源：

- 本书源代码；
- 相关工具安装包；
- 本书思维导图；
- 异步社区7天VIP会员。

要获得以上资源，您可以扫描下方二维码，根据指引领取。



提交勘误信息

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区（https://www.epubit.com），按书名搜索，进入本书页面，单击“发表勘误”，输入勘误信息，单击“提交勘误”按钮即可（见下图）。本书的作者和编辑会对您提交的勘误信息进行审核，确认并接受后，您将获赠异步社区的100积分。积分可用于在异步社区兑换优惠券、样书或奖品。



与我们联系

我们的联系邮箱是contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们。

如果您所在的学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接通过邮件发送给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“异步社区”是由人民邮电出版社创办的IT专业图书社区，于2015年8月上线运营，致力于优质内容的出版和分享，为读者提供高品质的学习内容，为作译者提供专业的出版服务，实现作者与读者在线交流互动，以及传统出版与数字出版的融合发展。

“异步图书”是异步社区策划出版的精品IT图书的品牌，依托于人民邮电出版社在计算机图书领域四十余年的发展与积淀。异步图书面向各行业的信息技术用户。

第一部分　规范驱动开发的理论基础与工具准备

第1章　AI编程的范式转移

在AI编程蓬勃发展的当下，我们享受了它带来的高效便捷，就需要直面随之而来的诸多挑战。本章将深入剖析AI编程的范式转移，从氛围编程的局限与挑战入手，引出规范驱动开发的核心价值，为第二部分的实战筑牢理论根基。

1.1 氛围编程的局限与挑战

“氛围编程”（Vibe Coding）并非一个正式的工程术语，而是随着AI编程助手（如GitHub Copilot、Cursor、Claude Code、Qwen Code等）的普及，在开发者社区中逐渐形成的一种现象性描述。氛围编程具有以下特征：开发者依赖即时感觉、上下文直觉与AI的快速生成能力，以近乎即兴的方式编写代码，而很少进行事先设计、规范约束或系统性审查。

这种编程模式在社交媒体上被一些开发者称为“与AI一起即兴演奏”——开发者抛出一个粗略想法，AI迅速生成一段可运行的代码，开发者试用、修改、再提示，循环往复直至“看起来能跑通”。它带来了前所未有的快速反馈与创作自由，尤其适合完成个人项目，如进行原型验证，撰写一次性脚本，以及学习新技术。然而，当氛围编程试图跨越个人项目的边界，进入团队协作、需要长期维护代码或实现关键业务系统等场景时，其固有的局限便迅速暴露。

1.1.1 局限一：代码质量如沙上建塔

氛围编程高度依赖开发者与AI之间的隐性约定。开发者可能在提示语中隐含了业务规则、边界条件或性能假设，而AI基于概率生成代码，并不会主动追问这些提示词是否合理。这会导致出现如下结果。

- **逻辑漏洞被深埋。**AI生成的代码可能只在“Happy Path”上能正常运行，异常处理、并发竞争、数据一致性等边缘逻辑往往缺失。

- **风格与规范混乱。**不同的提示词会生成不同风格的代码，这会导致在同一文件中也可能出现多种“笔迹”。

- **缺乏显式设计。**没有架构图、没有接口契约，系统的行为完全由各个文件中的代码片段定义，开发者理解系统只能逐行解读所有代码。

这样的代码就像在沙子上建塔——需求变更的“潮水”一来，“塔”随时可能会崩塌。

1.1.2 局限二：可维护性被严重透支

氛围编程优先考虑“当下能跑”，而将“未来能改”置于次要位置。当开发者为了完成某个项目，选择用氛围编程快速生成数万行代码时，维护者会遭遇以下困境。

- **文档“真空”。**当初为什么这么写？这个参数为何必须为负？AI生成这段代码时参考了哪些假设？除了提交历史中的只言片语，找不到任何解释。

- **重构恐惧。**调用关系不透明会导致维护者修改一个函数时不知道哪些地方依赖它，更不知道修改后是否破坏了隐含的业务规则。因此，测试覆盖率极低，维护者不敢轻易修改，只能不断堆叠补丁。

- **知识孤岛。**如果该项目的开发者请假或离职，那么接手的同事在面对大量没有注释、没有设计文档的AI生成的代码时，往往只能选择重写——而这又会开启新一轮的循环。

例如，某团队使用氛围编程在两周内搭建了一个内部工具，代码量约3000行。三个月后需要修改核心逻辑时，该团队花费了与首次开发相当的时间才完成重构——因为没有任何文档说明当初的设计意图。此时，代码不再是一笔资产，而是一个持续消耗团队精力的“无底洞”。

1.1.3 局限三：团队协作沦为“各自即兴”

氛围编程一旦引入团队协作，矛盾便迅速显现，具体如下。

- **接口没有契约。**模块之间通过隐式约定通信，A模块期望B模块返回特定结构，而B模块的开发者可能完全不知情。

- **合并时冲突层出不穷。**由于缺乏统一的规范约束，不同开发者生成的代码的结构差异巨大，合并时必须人工逐行判断。

- **评审无法聚焦。**代码审查演变为“猜作者意图”的游戏，审查者难以区分哪些是业务逻辑所必须的，哪些是冗余的。

这会导致团队陷入持续的“合并地狱”（Merge Hell）——每次合并都需要“手工”缝补，而开发者还会选择通过氛围编程来完成“手工”缝补。至此，团队陷入恶性循环。

1.1.4 局限四：结果不确定性

氛围编程高度依赖AI的输出。相同的提示词、不同时间、不同模型版本，AI可能生成完全不同的代码。这带来了两个维度的不确定性。

- **构建不可复现。**即使提示词完全相同，间隔数周后生成的代码也可能大相径庭。这意味着“规范”本质上没有被版本化，项目中已有的代码无法复现。

- **行为不可预测。**当代码由AI即兴创作时，开发者往往只验证了主流程，却无法知晓代码是否存在未被察觉的隐患（如引入不必要的依赖、泄露敏感信息、存在隐藏的安全漏洞等）。

这种结果不确定性在系统中往往是致命的。



小结：氛围编程不是原罪，但需要被驯服。

氛围编程本身没有错。它是技术普及的体现，让更多人拥有了开发软件的能力。它的“快”与“灵活”，在个人项目中依然是宝贵特质。

但将它应用到专业的团队软件开发时，仅凭“感觉”便难以为继。我们需要在保证AI生成效率的同时，为开发过程引入确定性、可复现性与可协作性——这正是规范驱动开发所要回答的问题。

1.2 SDD：新一代软件工程方法论

规范驱动开发（Specification-Driven Development，SDD）并非对氛围编程的全盘否定，而是对氛围编程的一次“工程化收编”：将即兴的火花放入规范的轨道，让创造的力量既快又稳地应用于复杂业务场景。

需求模糊，AI直接生成代码，开发者测试并发现问题后反复修改代码，但每次修改时依旧需求模糊，让AI直代生成代码，并且可能引入新问题，这缺乏统一的行为约定。

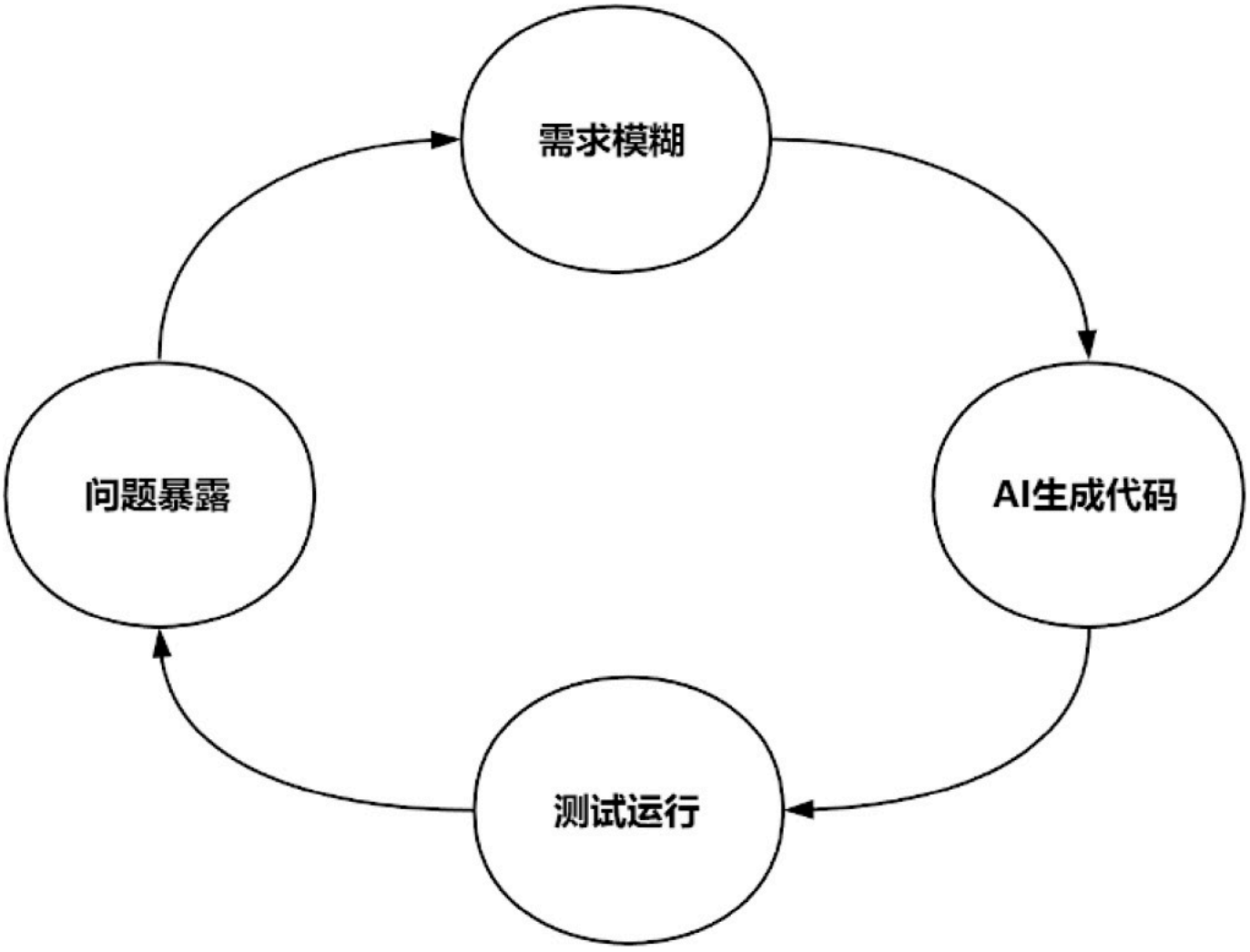


图1-1 AI辅助开发的循环困境

面对氛围编程带来的“效率幻觉”及其背后的技术债务陷阱，软件开发迫切需要一种新的范式——既能享受AI快速生成代码的红利，又能将开发过程从不可控的即兴表演转变为确定性、可复现、可协作的工程活动。

SDD正是为此而生，即在代码生成之前先定义它应该做什么。

1.2.1 SDD的定义

SDD是新一代的软件工程方法论，它将形式化或半形式化的规范作为开发活动的中心。在SDD中，规范不再仅仅是代码前的文档，而是可执行、可验证、可版本化的文档。SDD的流程如图1-2所示，根据需求意图编写规范，经规范审查后驱动AI生成代码，最终通过规范一致性测试验证。规范是保障团队沟通与代码质量的统一语言。

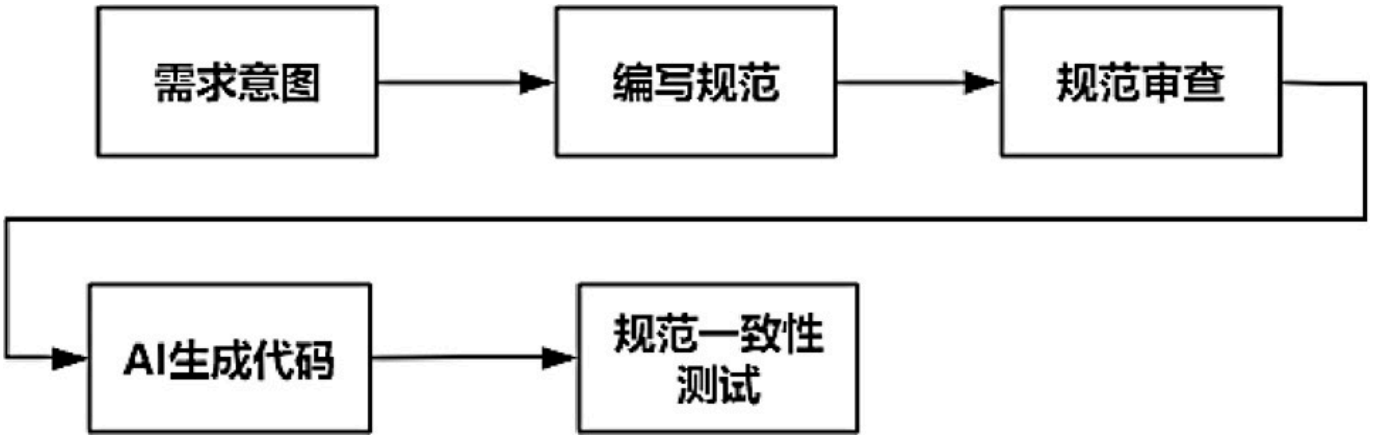


图1-2 SDD的流程

SDD让AI（或开发者）的任务不再是“凭空创造代码”，而是“实现一份已经被清晰定义的规范”。

1.2.2 SDD的三根支柱

SDD能够应对氛围编程带来的挑战，得益于三根支柱。每根支柱既是一种工程实践，也直接对应一项核心价值。

支柱一：规范即契约→确定性

在SDD中，规范是对系统行为的显式、无歧义的描述，具体如下。

- **接口定义语言（IDL）**：如OpenAPI、Protocol Buffers、AsyncAPI。
- **形式化规约**：如TLA+、Alloy，适用于金融交易等高可靠性场景。
- **轻量级业务规则**：如Gherkin（Cucumber）、Markdown结构化文档。
- **类型系统与约束**：如TypeScript类型、数据库模式、JSON Schema。

这四种规范形式涵盖了从严格到轻量的连续谱：形式化规约适用于金融交易等高可靠性场景，而IDL和类型系统与约束则更适合日常的接口与数据契约定义。

无论采用何种形式，规范都必须具备确定性与可验证性——即针对同一份规范，不同开发者或不同AI应能生成行为一致的实现结果。这从根本上消除了氛围编程中“同样需求、不同代码”的结果不确定性（见1.1.4节），使系统行为变得可预测、可复现。

支柱二：规范驱动实现→团队协作

规范的编写先于代码实现，并且规范的变更直接驱动代码的变更。在SDD工作流中有以下规则要遵守：

- 任何功能开发都从新增或修改规范开始；
- 规范通过代码审查后，才进入实现阶段；
- 实现完成后，必须通过规范一致性测试才能合入。

上述内容从根本上杜绝了“先写代码，后补文档”的恶习，也降低了AI生成无关代码或偏离预期意图的风险。接口的契约化使并行开发成为可能，这消除了代码审查演变为“猜作者意图”

（见1.1.3节）的可能性。

支柱三：规范即知识库→可维护性

规范不仅是供机器执行的契约，更是供人类阅读的知识沉淀。一份维护良好的规范集合，天然构成了系统的活文档，其优势如下：

- 新成员可以通过阅读规范理解系统的边界与行为；
- 审计人员可以通过规范追溯需求与实现之间的对应关系；
- 运维人员可以根据规范推断系统预期的运行时行为。

规范与代码通过工具链保持同步：规范的变更会触发代码审查，代码的修改也必须通过规范验证——文档，因此永远不会滞后于实现。这增强了项目的可维护性（见1.1.2节）。

1.2.3 为什么现在需要SDD——AI时代的必然选择

也许有人会问：在AI尚未普及的时代，我们同样强调需求与设计先行，那么SDD究竟新在何处？答案是：AI将编码成本几乎降为零，却将意图对齐的成本无限放大。在传统开发中，编写代码是主要瓶颈，因此瀑布模型、敏捷方法等都致力于更高效地生成代码。但在今天，瓶颈已经发生了转移，具体如下。

- **不如何编写代码：**AI可以在几秒内生成数百行可运行的代码；
- **不如何快速产出：**产出速度已经不再是瓶颈；
- **而在于如何确保产出的东西是正确的：**这才是真正的关键所在。

这正是SDD面临的历史性机遇。它促使开发者的核心工作从“与编译器对话”转向“与规范对话”——而规范，正是开发者意图最浓缩、最无歧义的表达形式。

1.2.4 SDD的适用场景与边界

SDD并非银弹，但在以下场景中能够发挥核心价值。

- **中大型团队协作开发：**接口规范先行，消除“各自即兴”的问题；
- **长期演进的业务系统：**规范作为“防腐层”，抵御技术债务的累积；
- **合规与审计敏感领域：**如金融、医疗、自动驾驶，必须证明实现与需求的一致性；
- **AI辅助开发的主流化：**用规范“驯服”AI，将提示工程升级为规范工程。

在纯探索性编程（如算法研究、数据探索）、一次性脚本或学习编程与新技术的练习阶段，轻量级规范甚至无规范仍然是合理的选择。SDD并不排斥即兴，而是主张在复杂度越过一是阈值时，有意识地将即兴转化为规范。

1.3 SDD核心价值：确定性、可维护性与团队协作

SDD的三根支柱并非抽象原则，它们分别对应三项核心价值：契约对应确定性，驱动实现对应团队协作，知识库对应可维护性。三者彼此强化，确定性使团队协作成为可能，团队协作使知识库得以持续维护，而知识库又反过来保障了确定性。

SDD并非以牺牲速度为代价换取这些价值；恰恰相反，它让团队在长期演进中保持高速而不失控。这正是SDD在AI时代最核心的工程红利。

第2章　深入理解Spec-Kit

掌握了SDD的理论框架后，我们需要借助具体工具将其落地。本章将深入剖析GitHub Spec-Kit（以下简称Spec-Kit）。——这一SDD的关键载体。我们首先明确Spec-Kit作为脚手架与工作流引擎的双重定位，接着阐释宪法、规范、计划与任务这四大核心概念，帮助你建立完整认知，为第3章筑牢基础。

2.1 Spec-Kit的定位

Spec-Kit是一个开源的SDD工具包，定位清晰：一是脚手架，快速搭建符合SDD原则的项目骨架；二是工作流引擎，通过内置的一系列交互命令将SDD的抽象方法论转化为可执行、可复现的具体流程；三是工具包，具备技术栈中立与AI可选性两大特点——既不绑定特定的编程语言、框架或云平台，也兼容各类AI编程代理，支持团队自由选择适配的AI编程代理。

2.1.1 脚手架：零摩擦的项目启动

脚手架的价值在于：它将“如何开始一个SDD项目”这个曾经需要阅读大量文档、手工复制文件才能解决的问题，压缩为一次命令执行。开发者从第一分钟起就站在了SDD的轨道上。

Spec-Kit通过其命令行工具 `specify` 承担脚手架角色。开发者只需输入如下命令即可获得一个完整的SDD项目结构。

```
specify init my-project--ai claude
```

这条命令会完成以下动作：

（1）创建项目目录并初始化Git仓库；

（2）生成 `.specify/` 目录，整体结构如下所示：

```
my-project/
  ├──.specify/
  │├──memory/          #项目上下文记忆
  │├──scripts/         #自动化脚本
  │├──specs/           #功能规范文档
  │└──templates/        #模板文件
  ├──CONSTITUTION.md        #项目宪法
  └──...
```

（3）根据所选AI助手（如Claude Code、Qwen Code、CodeBuddy等）注入对应的集成脚本；

（4）将一套预置的斜杠命令（slash commands）注册到AI环境中。

2.1.2 工作流引擎：从宪法到实现的可执行流程

如果说脚手架解决了“起点”问题，那么工作流引擎则解决了“路径”问题。它通过一组内置的斜杠命令，将SDD的核心理念编码为可交互、循序渐进的开发步骤。表2-1汇总了Spec-Kit的核心命令及其在SDD流程中的角色。

表2-1 Spec-Kit的核心命令及其在SDD流程中的角色

命令	角色	对应SDD阶段	核心产出物
/speckit.constitution	确立项目宪法	治理原则	CONSTITUTION.md
/speckit.specify	编写功能规范	需求定义	spec.md
/speckit.clarify	澄清未明确之处	规范细化	clarifications.md
/speckit.plan	制定技术方案	架构设计	plan.md
/speckit.tasks	分解为可执行任务	任务拆分	tasks.md
/speckit.implement	执行所有任务	实现交付	项目代码
/speckit.analyze	一致性分析	质量门禁	analysis.md
/speckit.checklist	生成自定义检查项	质量保障	checklist.md

回顾SDD的三根支柱（见1.2.2节）：`/speckit.constitution` 和 `/speckit.specify` 对应“规范即契约”；`/speckit.plan` 和 `/speckit.tasks` 对应“规范驱动实现”；而所有产出物的集合则构成“规范即知识库”。Spec-Kit将SDD的三根支柱，逐一映射为了可执行的工具命令。

这些命令并非孤立的脚本，而是共同构成一个状态感知的工作流引擎，具备以下特点。

- **顺序推荐**：命令之间存在逻辑上的前后依赖关系（如 `/speckit.plan` 通常应在 `/speckit.specify` 之后执行），开发者可遵循标准路径完成从构想到交付的全过程。

- **可重复性**：每个命令都会读取项目上下文（如已有规范、计划），并输出结构化的“制成品”，存储在 `.specify/` 目录下的标准化位置。

- **可扩展性**：团队可根据需要自定义命令行为，或基于现有命令构建更复杂的自动化流程。

每个斜杠命令的产出物——即宪法、规范、计划、任务这四大核心制成品——将在2.2节详细介绍。

2.1.3 工具包：技术栈中立与AI可选性

技术栈中立意味着Spec-Kit不会绑定任何特定的编程语言、框架或云平台。脚手架生成的是标准化的目录结构和一组Shell脚本，这些脚本用于支持跨功能的工作流自动化，而不预设应用层的编程语言或框架。

AI可选性则体现在Spec-Kit对各类AI编程代理的开放姿态。`specify init`命令通过`--ai`参数指定具体的AI编程代理，支持包括Claude Code、Qwen Code、Gemini CLI、CodeBuddy、Cursor、Windsurf等在内的多种工具。这意味着团队可以在不更换Spec-Kit的前提下，自由选择最适合自身工作流的AI编程代理。



小结：从“氛围”到“规范”的桥梁

Spec-Kit并不试图消灭AI的创造力，而是为这种创造力提供轨道。脚手架确保每个项目都遵循相同的规范结构，杜绝“各自即兴”的项目布局；工作流引擎确保每次开发都经历从宪法到实现的完整闭环，杜绝“需求→代码”的“一跳式赌博”。

Spec-Kit让你专注于产品场景和可预测的结果，而非从零开始每个片段的氛围编程。

2.2 核心概念：宪法、规范、计划与任务

如果说斜杠命令是SDD的“操作界面”，那么本节要讨论的四个核心制成品（artifacts）就是其背后的结构化产物。它们被持久化存储在项目的 `.specify/` 目录中，具备可版本化、可审查、可复用的特性，是SDD方法论在Spec-Kit中的具体投射。

本节将解读Spec-Kit如何将SDD的三根支柱落地为四类可版本化、可审查的结构化产物。

2.2.1 宪法

本书中的“宪法”（Constitution）是软件工程与规范驱动开发中的一个比喻性术语，特指项目内部共同约定的最高行为准则与质量标准。它不指代、也绝不关联任何国家或政治实体的宪法，仅用于描述Spec-Kit中那份具有根本性、稳定性的团队协作文档。为行文简明，后文将继续沿用“宪法”这一简称，请读者在技术语境下理解。

宪法作为项目的“根本大法”，为所有下游活动判定价值判断准则。其保存路径为`.specify/memory/constitution.md`，该路径下的内容涵盖：

- **代码质量要求**（命名规范、复杂度阈值、注释风格）；
- **测试标准**（覆盖率、测试类型、持续集成策略）；
- **用户体验一致性原则**（交互模式、视觉风格、无障碍标准）；
- **性能与安全底线**（响应时间、并发量、漏洞容忍度）；
- **技术决策治理规则**（何时引入新依赖、如何评估架构方案）。

其设计意图如下。

- **为AI建立长期记忆**：`memory`目录名即暗示其用途——每次对话中，AI都会自动加载宪法，确保所有后续生成行为与项目初始约定保持一致。
- **将隐性共识显式化**：团队关于“好代码”的默认认知往往存在于口头或零散文档中，宪法将其整合为单一权威文本，杜绝开发过程中的随意偏离。

宪法一经确立则应保持相对稳定，重大修订需经团队评审。

2.2.2 规范

规范（Spec）是对业务意图的无歧义功能描述，聚焦于“要构建什么”，是完全不涉及技术实现细节的需求描述。文件保存路径为 `specs/<feature-branch-name>/spec.md`，其典型内容结构包括以下要素。

- **用户故事**：以角色驱动的方式描述功能价值（如“作为用户，我希望……以便……”）。
- **功能需求**：具体、可验证的行为规则（如“拖拽卡片到新列后，卡片状态应立即更新”）。
- **验收条件**：判定功能是否完成的检查点。
- **界面示意**（可选）：线框图或文字描述的用户界面布局。
- 注意，规范不得包含任何技术栈、框架、API或数据存储的细节。规范的设计意图如下。
- **分离问题域与解空间**（即技术实现的可能性空间）：允许产品负责人、业务分析师、测试人员与开发者使用同一份文档对话，无需提前绑定技术方案。
- **作为变更的锚点**：任何功能演进都必须从修改规范开始，确保需求痕迹全程可追溯。

2.2.3 计划

计划（Plan）是将功能需求转化为可实施的技术方案的核心文档，文件统一存储在 `specs/<feature-branch-name>/` 目录下，至少包含以下关键文件，如表2-2所示：

表2-2 项目计划文件及其用途说明

文件	用途
plan.md	总体实现策略（架构图、组件划分、技术选型依据、模块职责）
data-model.md	数据实体、关系、字段定义（ER图、JSON Schema等）
contracts/	接口契约（OpenAPI、Protocol Buffers、SignalR规范等）
research.md	关键技术点的调研记录、版本锁定说明、风险应对方案
quickstart.md	快速启动指南，描述如何在本地运行和验证实现（环境依赖、启动命令、验证步骤）

其设计意图如下。

- **将规范翻译为工程语言**：计划是连接“需求”与“代码”的桥梁，明确回答“用什么技术、如何组织、怎样交互”。
- **支持并行研究与迭代**：`research.md` 用于存放不确定领域的探索过程，开发者可要求AI对特定技术点进行深度调研，并将结论写入此文件。
- **持续演进**：计划并非一次性成品，它会在实现过程中随认知深化而持续完善（例如发现遗漏的契约、调整数据模型）。

2.2.4 任务

任务（Tasks）是将技术方案切割为可执行的原子步骤，其保存路径为 `specs/<feature-branch-name>/tasks.md` 。任务的设计体现了浓厚的工程可执行性，具体要求如下。

- **按用户故事分组**：每个故事对应一组任务，支持增量交付与独立验证。
- **依赖关系显式化**：任务顺序反映组件之间的构建依赖（如数据模型→服务层→API→UI）。
- **并行标记[P]**：由 `/speckit.tasks` 命令自动生成，标注可并发执行的任务，为AI或多人协作提供并行指引。
- **文件路径锚定**：每条任务均注明待创建或符修改的具体文件路径，消除“该改哪里”的歧义。
- **测试先行**：若要求测试，则测试编写任务排在实现任务之前，天然嵌入测试驱动开发（Test-Driven Development，TDD）实践。
- 其设计意图如下。
- **将计划“代码化”**：`tasks.md` 是 `/speckit.implement` 的直接输入，AI严格按照任务顺序执行，实现过程严格受控。
- **降低认知负荷**：开发者无需在实现时反复思考“下一步做什么”，任务清单提供了清晰的路线图。
- **进度可视化**：每个任务的完成状态可追踪，团队可随时了解功能开发的准确进度。

2.2.5 四者的关系

宪法、规范、计划、任务共同构成了一条层层递进、闭环反馈的链条。最终产出的代码，会通过测试与评审反向验证上游制品的合理性，从而驱动迭代优化。Spec-Kit的全流程如图2-1所示。

这种自上而下的依赖关系建立了清晰的变更联动机制。当上游制品发生变化时，下游制品需要相应更新：修改宪法可能需要重新审视已有的规范和计划是否仍然合规；修改规范则意味着计划和任务需要相应调整；修改计划则任务需要重新拆分。这一机制确保了变更的可控性和可追溯性。

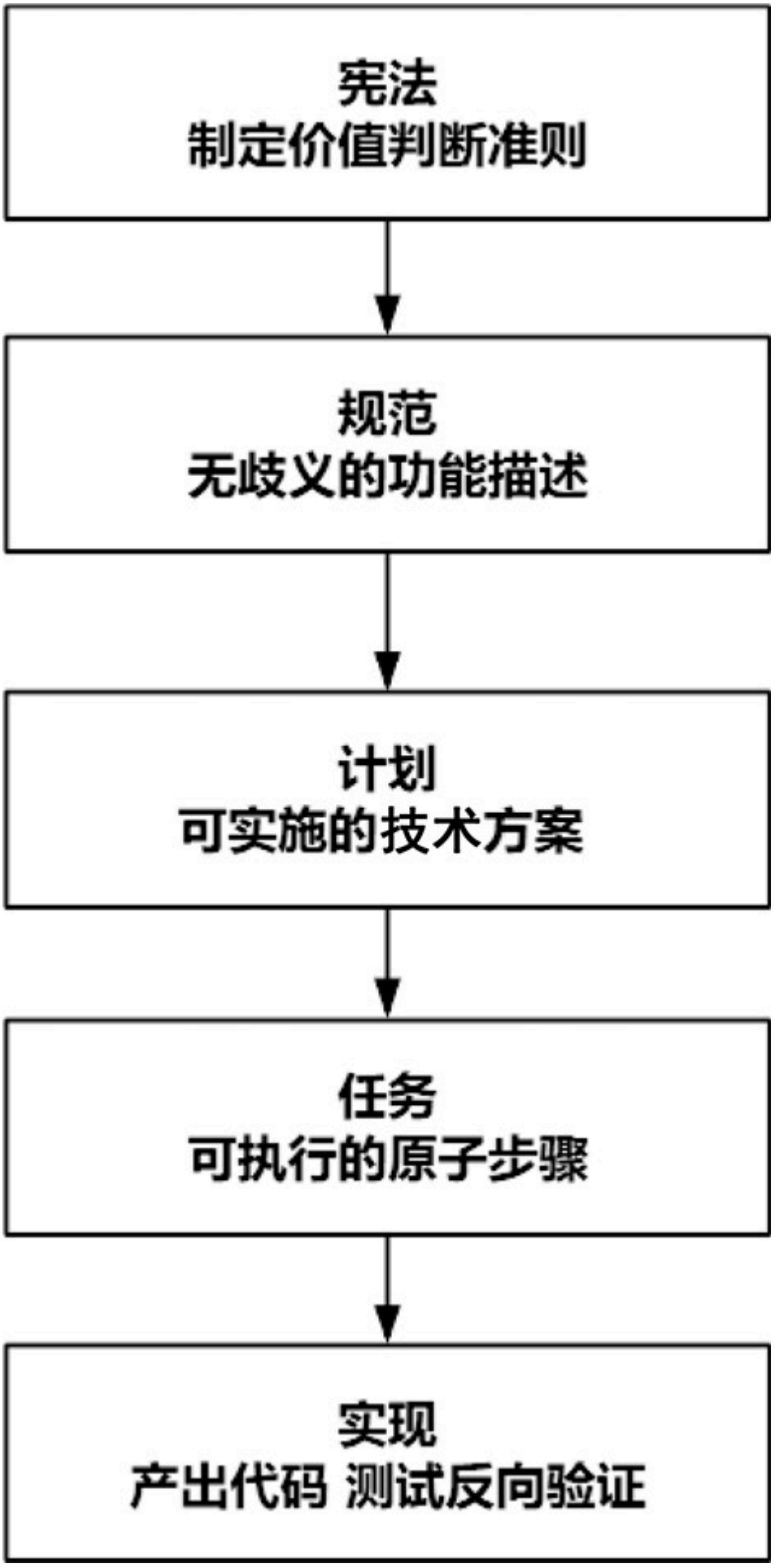

宪法
制定价值判断准则
规范
无歧义的功能描述
计划
可实施的技术方案
任务
可执行的原子步骤
实现
产出代码 测试反向验证

图2-1 Spec-Kit的全流程

在下一章中，我们将执行具体的命令，从环境配置到第一个功能的完整实现，走通SDD的全流程。

第3章　环境配置与AI编程代理

在理解了SDD的核心价值与Spec-Kit的核心概念后。本章将聚焦于环境配置与AI编程代理，为第二部分的实战开发筑牢基础。本章将详细介绍如何安装与配置Spec-Kit CLI，确保开发环境的正确性；同时，还会带你认识并安装CodeBuddy，这是我们实战中主要使用的AI编程代理。通过学习本章内容，你将拥有使用Spec-Kit进行SDD的基本环境。

3.1 安装与配置Spec-Kit

Spec-Kit的核心入口是命令行工具（后文简称 `specify` ） `specify-oli` 。它负责生成项目脚手架、绑定AI编程代理以及注入工作流脚本。本节将指导你完成 `specify` 的安装、验证与基础配置。

3.1.1 安装specify

Spec-Kit官方推荐使用uv来管理工具。uv是一款高性能Python包管理器，支持工具安装后全局可用，能够显著提升依赖安装与项目构建效率。

在安装 `specify-cli` 前，需要满足以下前置条件。

- **操作系统**：Linux、macOS或Windows 10/11（64位）。
- Python**环境**：Python 3.11或更高版本。
- **版本控制**：Git 2.30.0或更高版本，需配置好用户信息。
- **工具依赖**：uv包管理器。
- AI**编程代理**：至少有一个AI编程代理（如CodeBuddy、Qwen Code、Claude Code、Cursor等）。

接下来，我们将详细介绍如何安装 `specify` 工具。

你可以通过GitHub官方仓库的Releases页面获取最新版本号及安装包，具体路径为：在GitHub首页搜索“spec-kit”项目，进入项目页面后找到“Releases”入口即可查看。以下命令表示使用 `uv` 包管理器，从GitHub上的指定仓库地址和版本标签，安装 `specify` 。其中， X.Y.Z表示版本号，你可以自行替换。

```
uv tool install specify-cli--from git+https://github.com/github/spec-kit.git@vX.Y.Z
```

当GitHub官方发布Spec-Kit的新版本后，你可以通过以下命令将本地的 `specify` 工具升级到指定新版本，以获取新功能和修复已知问题。

```
uv tool install specify-cli--force--from git+https://github.com/github/spec-kit.git@vX.Y.Z
```

本书选择安装的版本是v0.5.0，因此执行如下命令。

```
uv tool install specify-cli--from git+https://github.com/github/spec-kit.git@v0.5.0
```

执行此命令后，`specify`工具将被安装到系统全局环境变量`PATH`中。这意味着后续在任意终端目录下，无需输入完整路径，直接输入`specify`命令即可调用该工具，极大地提升了操作便捷性。

安装完成后，为确保工具部署成功，需进行安装验证。在命令行窗口输入以下命令。

```
specify--help
```

若终端显示命令帮助信息，则说明安装成功，`specify`的交互界面如图3-1所示。



图3-1 specify的交互界面

3.1.2 验证系统依赖项是否完善

`specify` 提供了 `check` 子命令，执行如下命令可检测当前环境中已安装的AI编程代理及其他必要依赖。

```
specify check
```

依赖项检测结果如图3-2所示。

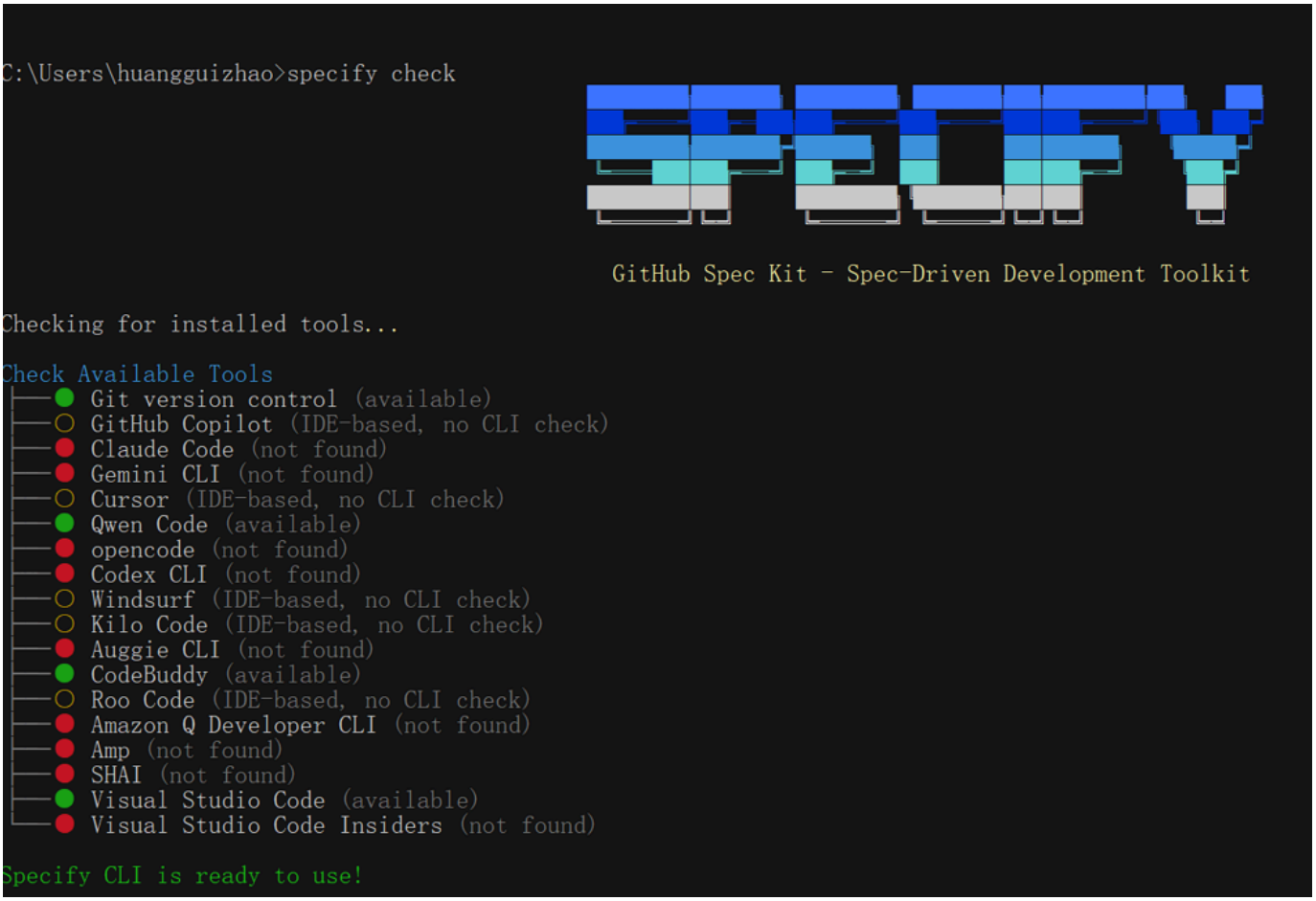


图3-2 依赖项检测结果

本书后续案例将使用CodeBuddy作为AI编程代理。如果你安装了其他工具（如Qwen Code、Claude Code、Cursor等），只需在 `specify init` 时通过 `--ai` 参数指定。

确认AI编程代理已就绪后，下一步就是安装AI编程代理CodeBuddy。

3.2 认识与安装CodeBuddy

`specify` 负责项目脚手架搭建和工作流引导，但真正执行“理解需求”“生成代码”“修复缺陷”这些动作的是AI编程代理。本节将介绍本书选用的AI编程代理——CodeBuddy。

3.2.1 CodeBuddy是什么

CodeBuddy是腾讯云推出的AI辅助编程工具，通过将AI深度融入开发全流程，打造从产品构思到产品发布的一站式高效交付平台。它支持IDE、插件和CLI三种形态，覆盖从专业开发者到零基础用户的全场景需求。Codebuddy的官网首页如图3-3所示。



图3-3 Codebuddy的官网首页

提示

本书选择以CLI形态运行，这种形态天然适合与Spec-Kit的斜杠命令工作流深度集成。两者都是终端驱动的工具，因此不存在IDE锁定或版本兼容问题。

3.2.2 安装CodeBuddy CLI

由于CodeBuddy CLI需要通过npm包管理器安装，因此需要提前配置Node.js运行环境，版本要求为18.0及以上。

提示

注意！在实测中，CodeBuddy需要node.js 18.20.8版本。

安装CodeBuddy CLI，在命令行窗口执行如下命令。

```
npm install-g@tencent-ai/codebuddy-code
```

完成安装后，通过在命令行窗口执行以下命令，判断CodeBuddy CLI是否成功部署。

```
codebuddy--version
```

若终端返回的信息中显示对应版本号，则说明CodeBuddy CLI已成功安装，CodeBuddy CLI安装成功的界面如图3-4所示。

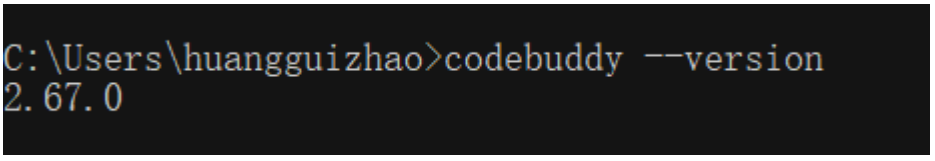


图3-4 CodeBuddy CLI安装成功的界面

3.3 其他AI编程代理

Spec-Kit支持25种以上的AI编程代理。常见AI编程代理对比如表3-1所示：

表3-1 常见AI编程代理对比

AI编程代理	特点	适用场景
CodeBuddy	腾讯云生态集成，中文自然语言交互结果好	国内团队，腾讯云用户
Qwen Code	阿里通义千问驱动，中文理解能力强	国内团队，阿里云用户
Claude Code	Anthropic出品，推理能力强	海外团队，复杂逻辑场景
Cursor	基于VS Code的AI IDE，编辑体验好	偏好图形化界面的开发者
GitHub Copilot	GitHub原生集成，生态最成熟	GitHub重度用户

选择哪种AI编程代理，并不会影响Spec-Kit的工作流本身，这是因为宪法、规范、计划、任务与对应结构和命令完全一致，差异仅体现在AI模型的理解能力和生成质量上。本书以CodeBuddy为示例进行讲解，但你完全可以将其替换为其他AI编程代理，例如将 `--ai codebuddy` 替换为 `--ai qwen` ，后续操作完全相同。

至此，环境配置已全部完成。

第二部分　规范驱动开发实战进阶

第4章　【基础】组件驱动开发：用户名片组件

理论终须落地。本章将以“用户名片组件”为例，从零开始，带你了解最基础的实战流程，让你直观感受Spec-Kit在实际项目中的应用。

4.1 项目初始化及宪法确立

Spec-Kit的全流程包含四个核心阶段：宪法、规范、计划、任务。本节将以“用户名片组件”为例，详细讲解项目初始化与宪法确立。

4.1.1 初始化项目

打开系统终端，执行以下命令来初始化项目（以下操作以Windows系统为例）：

```
specify init user-card--ai codebuddy
```

选择“ps（Power Shell）”选项，操作界面如图4-1所示。

初始化完成后，`specify`会自动展示所有可执行命令的详细说明，帮助开发者快速了解后续操作流程。`specify`初始化完成后的命令说明界面如图4-2所示。

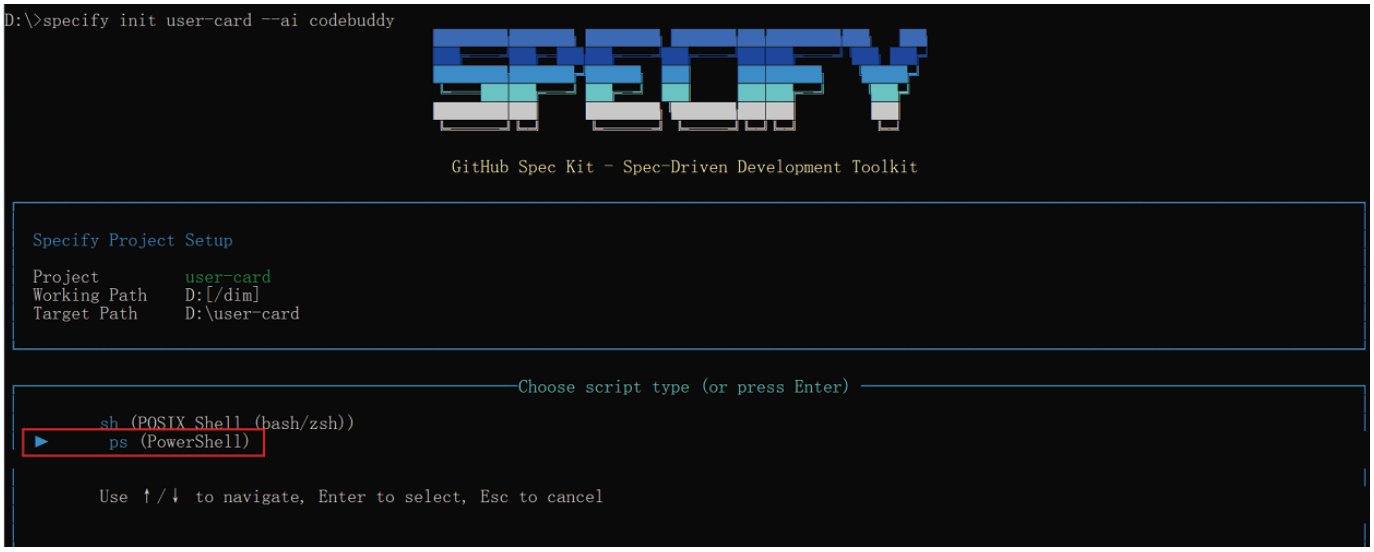


图4-1 选择“ps”（Power Shell）的操作界面

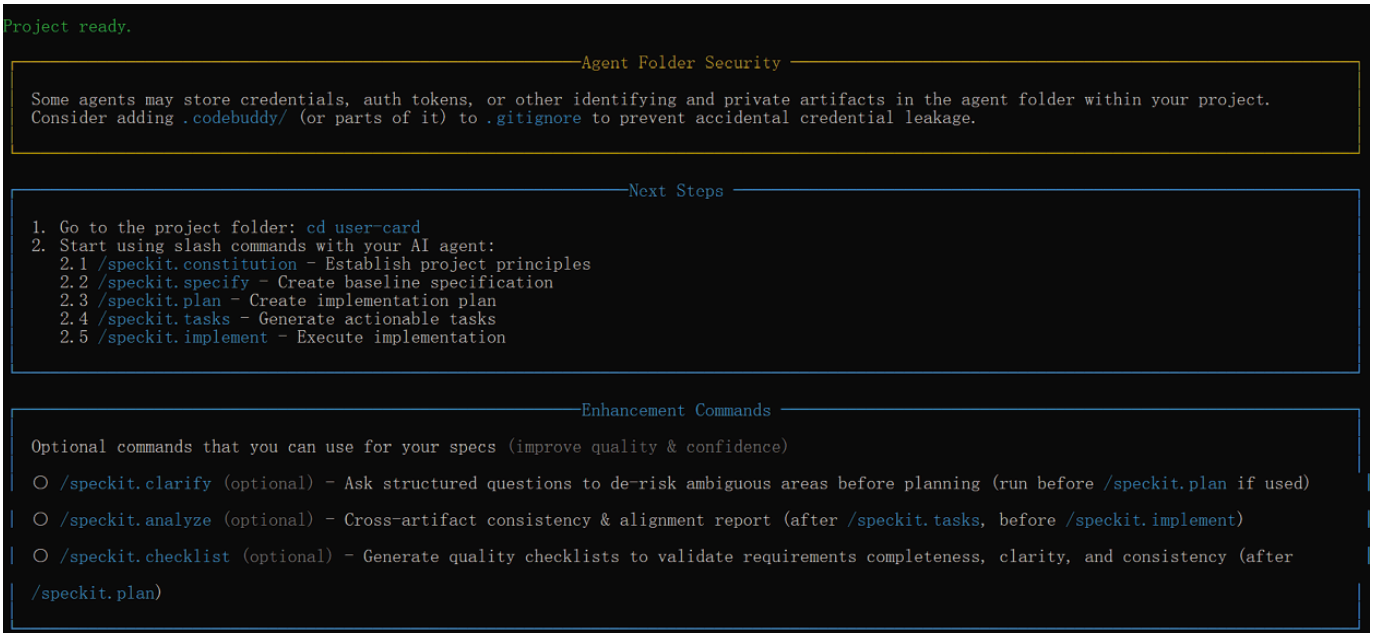


图4-2 specify初始化完成后的指令说明界面

提示

`specify` 主要完成以下工作：

1.从GitHub拉取最新模板，创建 `user-card/` 目录并初始化Git仓库；

2.生成 `.specify/` 标准化目录结构；

3.根据 `--ai codebuddy` 参数，将Spec-Kit的斜杠命令注册到CodeBuddy环境中。

项目初始化完成后，我们可以看到项目根目录下已自动生成并填充了相关文件。项目根目录文件结构如图4-3所示。

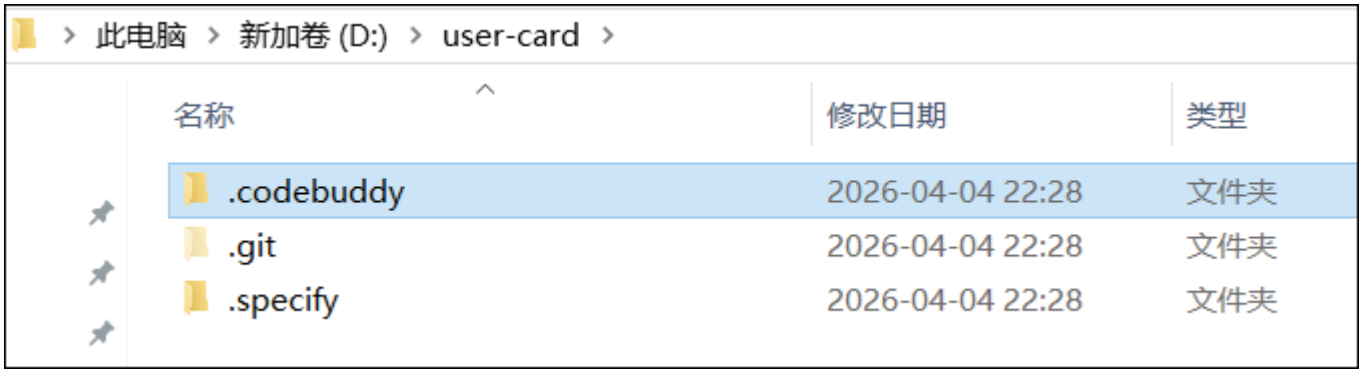


图4-3 项目根目录文件结构

`.specify` 目录是项目的核心配置区域，包含了脚手架的所有相关文件与可执行脚本。`.specify` 目录内部结构如图4-4所示。

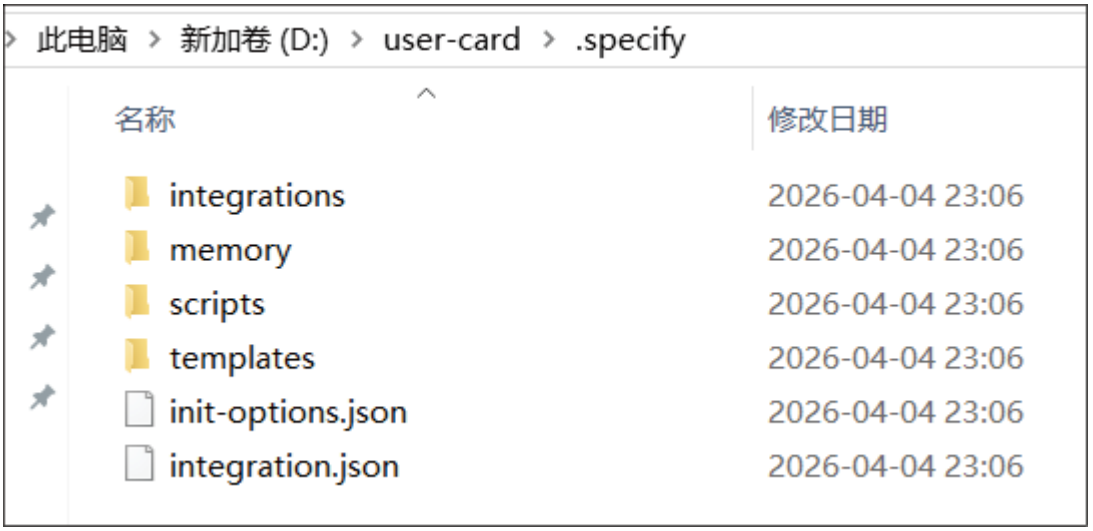


图4-4 .specify目录内部结构

4.1.2 首次启动CodeBuddy

保持当前终端处于用户名片项目根目录下（如 `user-card/` ），在命令行中输入以下命令并按下回车键，即可启动CodeBuddy交互环境。

```
codebuddy
```

执行上述命令后，终端将自动切换至CodeBuddy交互界面，选择第1项即可进入（见图4-5）。

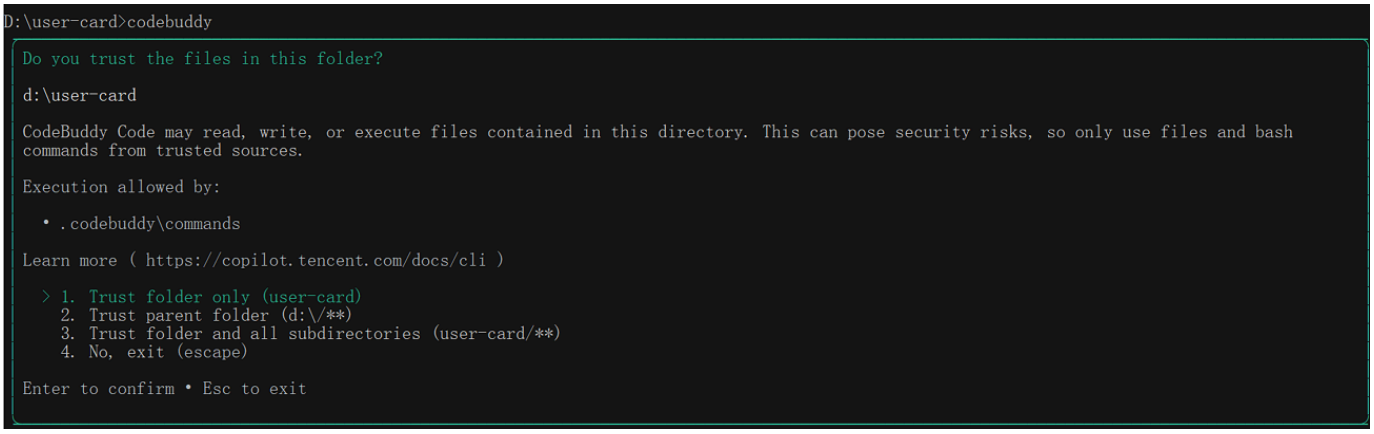


图4-5 CodeBuddy交互界面

在CodeBuddy交互界面的>提示符后输入“ `/spec` ”，若系统自动关联出相关命令，则说明Spec-Kit与CodeBuddy已成功结合（见图4-6）。

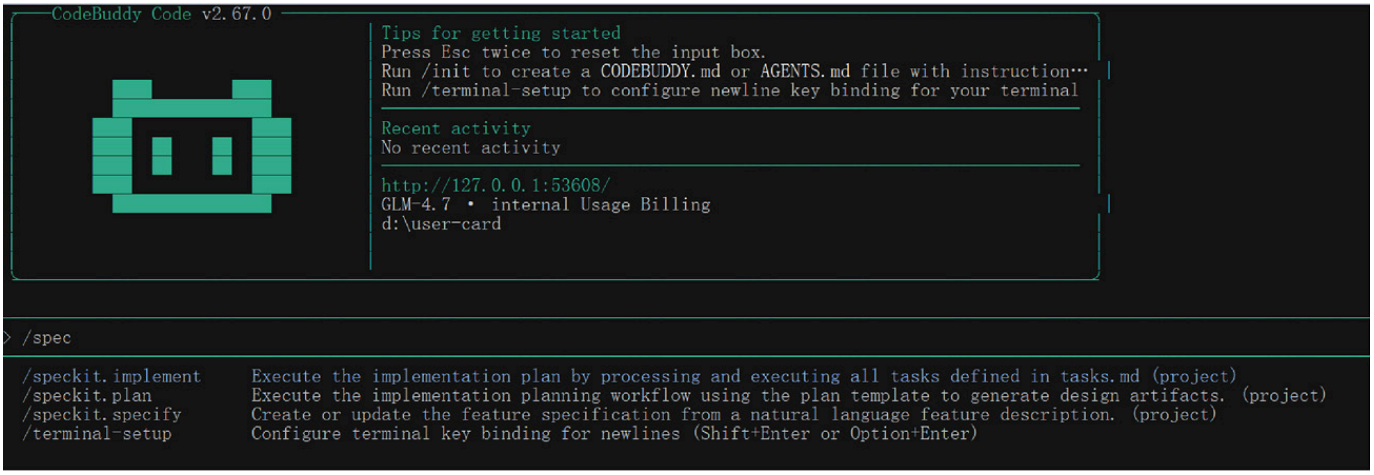


图4-6 Spec-Kit与CodeBuddy成功结合

4.1.3 确立项目宪法

请保持当前的CodeBuddy交互界面状态，输入以下内容。

/speckit.constitution为一个前端组件库项目创建开发原则。**所有项目文档（规范、计划、任务等）必须使用中文编写。**代码质量：遵循官方推荐的代码风格，组件命名遵循PascalCase规范，每个组件文件不超过200行。测试：每个组件必须有对应的单元测试，核心逻辑覆盖率不低于80%，测试必须覆盖属性传递、事件触发和边界条件。用户体验：组件必须支持键盘导航，遵循WAI-ARIA无障碍指南，所有交互反馈延迟不超过100ms。性能：单个组件首次渲染时间不超过100ms，禁止在渲染函数中执行昂贵计算。技术治理：禁止引入非必要的第三方依赖，引入新依赖必须说明选型理由并记录在案。

完成上述命令后，CodeBuddy将依据设定的宪法原则自动生成对应的宪法文件 `constitution.md` 。CodeBuddy生成文件内容并询问写入确认界面如图4-7所示。

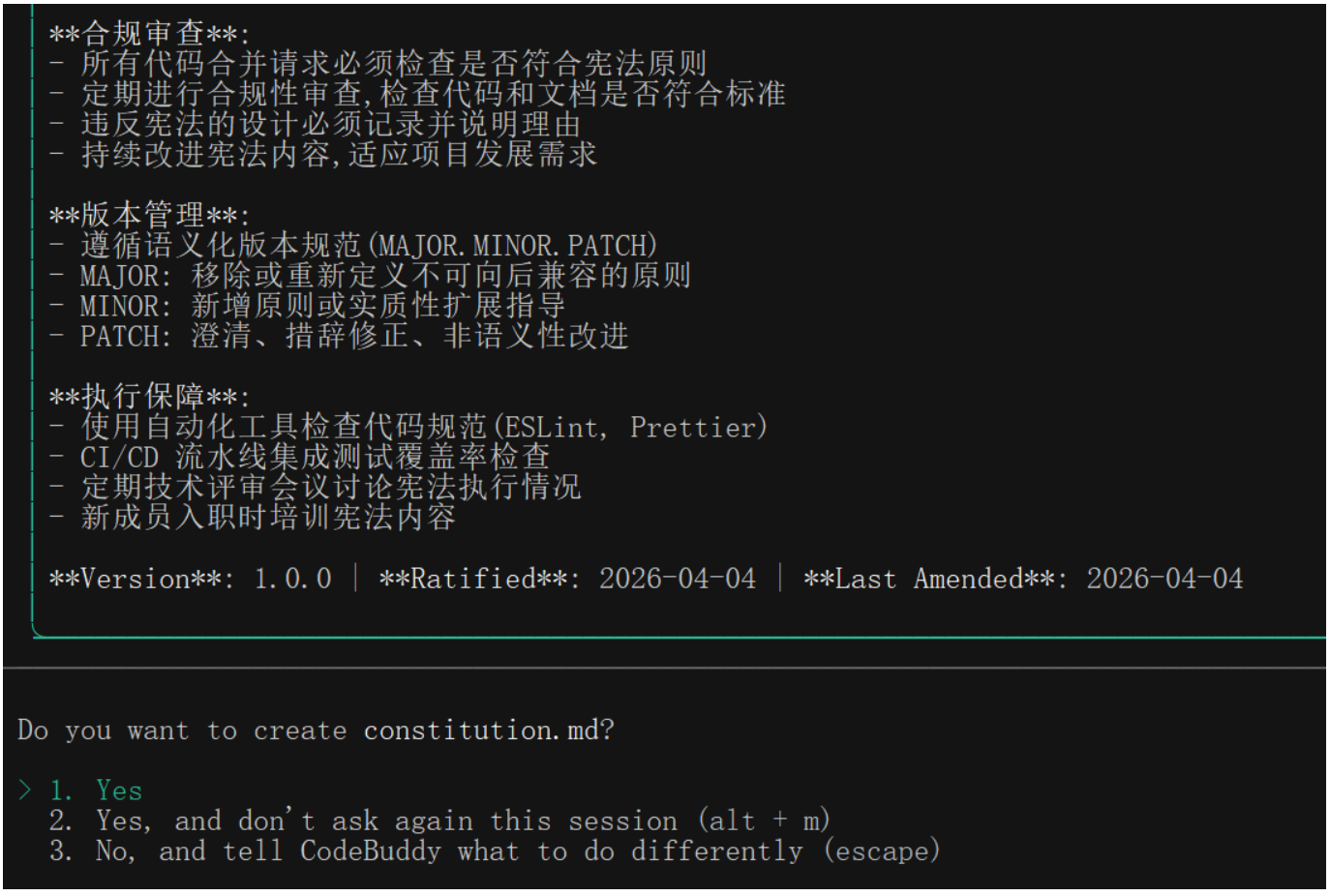


图4-7 CodeBuddy生成文件内容并询问写入确认界面

选择“Yes”确认后，CodeBuddy自动创建宪法文件流程如图4-8所示。

操作完成后，按“Ctrl+C”退出当前的CodeBuddy交互界面，返回系统终端。

生成的项目宪法文件的部分内容如图4-9所示。



图4-8 CodeBuddy自动创建宪法文件流程

```
# UserCard 组件库宪法

## 核心原则

### 一、文档规范

所有项目文档(包括但不限于规范文档、实施计划、任务列表、技术文档、API 文档等)必须使用中文编写。

**规则**:
- 项目内所有文档强制使用中文,确保团队成员理解一致
- 代码注释使用中文,提高代码可读性
- 技术术语保留英文原文,但提供中文解释
- Git 提交信息使用中文,清晰描述变更内容

**理由**:
中文文档确保团队成员对技术规范、架构决策和实施细节有统一理解,减少因语言差异导致的沟通成本和误解,提高协作效率。

### 二、代码质量

**规则**:
- 遵循官方推荐的代码风格指南(如 ESLint 配置、Prettier 规则)
- 组件命名遵循 PascalCase 规范(如 `Button`, `InputField`)
- 每个组件源文件代码行数不超过 200 行
- 组件应保持单一职责,功能复杂时拆分为子组件或提取逻辑

**理由**:
统一的代码风格提高代码可维护性,PascalCase 命名符合 React/Vue 等主流框架的组件命名约定,文件大小限制强制开发者保持

### 三、测试标准

**规则**:
- 每个组件必须有对应的单元测试文件
- 核心逻辑测试覆盖率不低于 80%
- 测试必须覆盖以下场景:
  - 属性(props)传递验证
  - 事件触发和回调函数执行
  - 边界条件和异常情况
  - 组件渲染状态变化
- 测试文件命名与组件文件对应,添加 `.test` 后缀
```

图4-9 生成的项目宪法文件的部分内容



提示

重要建议：对于任意环节，在AI生成每一份文件后，务必对内容进行细致审查。若发现任何不符合预期的表述、逻辑或格式问题，请立即向AI反馈并要求修正，确认内容无误后再推进至下一环节。

4.2 编写组件规范

本节将使用 `/speckit.specify` 编写用户名片组件的功能规范。这一步的核心是回答“要做什么”，不涉及任何技术细节。

4.2.1 输入需求

打开系统终端，确保当前工作目录为本项目的根目录（即 `user-card/` ），然后执行以下命令启动CodeBuddy交互环境。

```
Codebuddy
```

进入CodeBuddy交互界面后，输入以下内容。

> /speckit.specify构建一个用户名片组件。该组件接收用户信息（头像、姓名、职位、邮箱）并将其作为输入，以卡片形式展示。用户头像显示在卡片左侧，姓名和职位显示在头像右侧，邮箱显示在最下方。鼠标悬停时卡片出现轻微阴影。点击卡片时触发点击事件，将用户信息传递给父组件。所有项目文档使用中文编写。

输入完成后，CodeBuddy将依据提供的需求描述生成对应的规范文档，CodeBuddy自动生成规范文档的工作流程如图4-10所示。



图4-10 CodeBuddy自动生成规范文档的工作流程

选择“Yes”确认后，CodeBuddy将自动创建对应的功能分支，创建的功能分支如图4-11所示。



图4-11 创建的功能分支

随后，CodeBuddy会提示确认是否创建spec.md文件，CodeBuddy确认创建 `spec.md` 文件界面如图4-12所示。

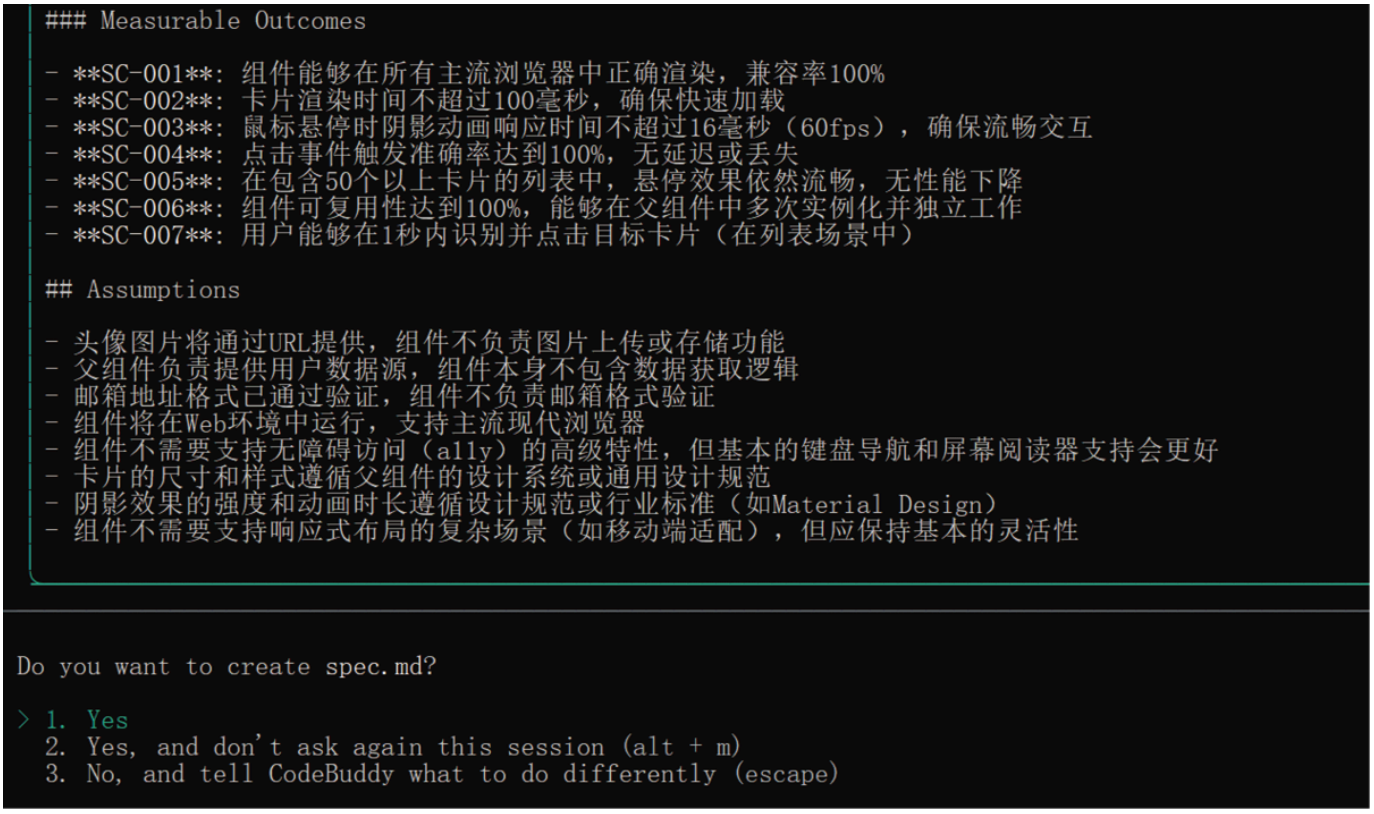


图4-12 CodeBuddy确认创建spec.md文件界面

确认完成后，按“Ctrl+C”即可退出当前CodeBuddy交互界面，结束编写组件规范文档环节。

4.2.2 审视需求是否被准确理解

在SDD流程中，审视需求是至关重要的环节。它能够帮助我们及时发现需求描述中的模糊点、歧义点和遗漏点，确保后续开发工作基于清晰、准确的需求展开。通过审视需求，我们可以验证AI生成的规范是否准确符合了我们的业务意图，避免因需求理解偏差导致的开发返工与资源浪费。同时，审视需求也为团队成员提供了理解需求的机会，有助于减少沟通成本、提高开发效率。

生成规范文档后，我们可以在该项目目录中找到对应的文件，本地规范文档路径为项目根目录下的 `specs\001-user-card-component\spec.md` ，如图4-13所示。

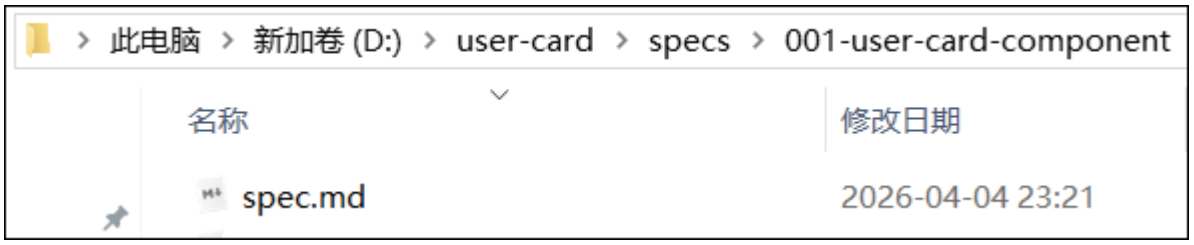


图4-13 本地规范文档路径

打开该文件后，可以看到规范文档的具体内容，其中包含了用户名片组件的详细需求描述、功能说明等信息，如图4-14所示。



```
# Feature Specification: 用户名片组件

**Feature Branch**: `001-user-card-component`
**Created**: 2026-04-04
**Status**: Draft
**Input**: User description:
"构建一个用户名片组件。该组件接收用户信息（头像、姓名、职位、邮箱）作为输入，以卡片形式展示。用户头像显示在卡片左侧，姓名和职位
悬停时卡片出现轻微阴影提升效果。点击卡片时触发点击事件，将用户信息传递给父组件。所有项目文档使用中文编写。"

## User Scenarios & Testing *(mandatory)*

### User Story 1 - 展示用户名片 (Priority: P1)

用户需要在一个界面中查看其他用户的基本信息，包括头像、姓名、职位和联系方式（邮箱）。这些信息以卡片形式呈现，便于快速浏览和识别。

**Why this priority**: 这是组件的核心功能，没有基本的展示能力，其他交互都无法进行。这是用户接触组件的第一印象，直接关系到用户体

**Independent Test**:
可以通过向组件传入完整的用户信息数据（头像URL、姓名、职位、邮箱），验证组件能够正确渲染并展示所有必需信息。测试数据可以来源于Mo

**Acceptance Scenarios**:

1. **Given** 组件接收到包含完整用户信息的数据，**When** 组件渲染完成，**Then** 卡片左侧显示用户头像，右侧上方显示用户姓名，右侧
2. **Given** 组件接收到包含完整用户信息的数据，**When** 卡片显示在页面上，**Then** 所有文本内容清晰可读，头像显示完整且比例正确

---

### User Story 2 - 悬停交互效果 (Priority: P2)

用户在浏览用户卡片列表时，需要通过视觉反馈来识别当前鼠标悬停的卡片，以便快速定位和选择目标用户。

**Why this priority**: 虽然不是核心功能，但良好的交互体验能显著提升可用性，特别是在卡片列表场景中，帮助用户明确当前操作对象。

**Independent Test**: 将鼠标移动到已渲染的用户卡片上方，验证卡片是否显示阴影提升效果，移开鼠标后效果是否消失。

**Acceptance Scenarios**:

1. **Given** 已渲染的用户卡片，**When** 鼠标悬停在卡片上，**Then** 卡片显示轻微的阴影提升效果
2. **Given** 鼠标悬停在卡片上且显示阴影效果，**When** 鼠标移开卡片区域，**Then** 阴影效果消失，卡片恢复原始状态

---
```

图4-14 项目规范文档的内容

当然，我们也可以在自己熟悉的集成开发环境（IDE）中打开项目，这样能更高效地管理和编辑项目文件。借助IDE的代码高亮、智能提示、版本控制集成等功能，可以进一步提升开发效率。

4.3 制定技术方案

4.2节明确了“要做什么”。本节将使用 `/speckit.plan` 制定技术方案——直到此时，Vue、Vite、TypeScript等技术选型才正式登场。

4.3.1 输入技术方案

在系统终端中启动CodeBuddy交互环境，进入CodeBuddy交互界面后，输入以下内容描述技术方案。

```
/speckit.plan使用Vue 3+Vite+TypeScript开发。组件采用Composition API编写，样式使用Scoped CSS。不引入任何第三方UI库，仅使用Vue 3原生能力。使用Vitest+Vue Test Utils进行单元测试。构建工具使用Vite，包管理使用npm。所有项目文档使用中文编写。
```

CodeBuddy收到命令后，会自动读取项目中已有的宪法文件和规范文档，并结合提供的技术方案描述，生成一份完整的项目计划文件。

4.3.2 生成技术方案相关文件

在确定修改内容无误后，CodeBuddy会提示你确认是否编辑 `plan.md` 文件，此时选择“Yes”即可完成计划文件的创建。CodeBuddy计划文件确认界面如图4-15所示。

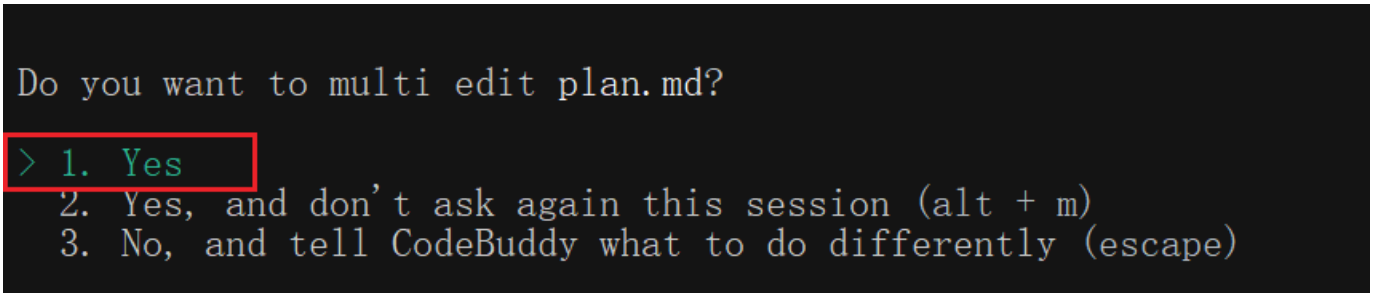


图4-15 CodeBuddy计划文件确认界面

在成功生成项目核心规划文件 `plan.md` 后，CodeBuddy将自动启动技术调研文档生成流程，创建用于记录技术选型分析、方案对比及决策依据的 `research.md` 文件。此时需在交互界面中选择“Yes”确认创建操作，如图4-16所示。

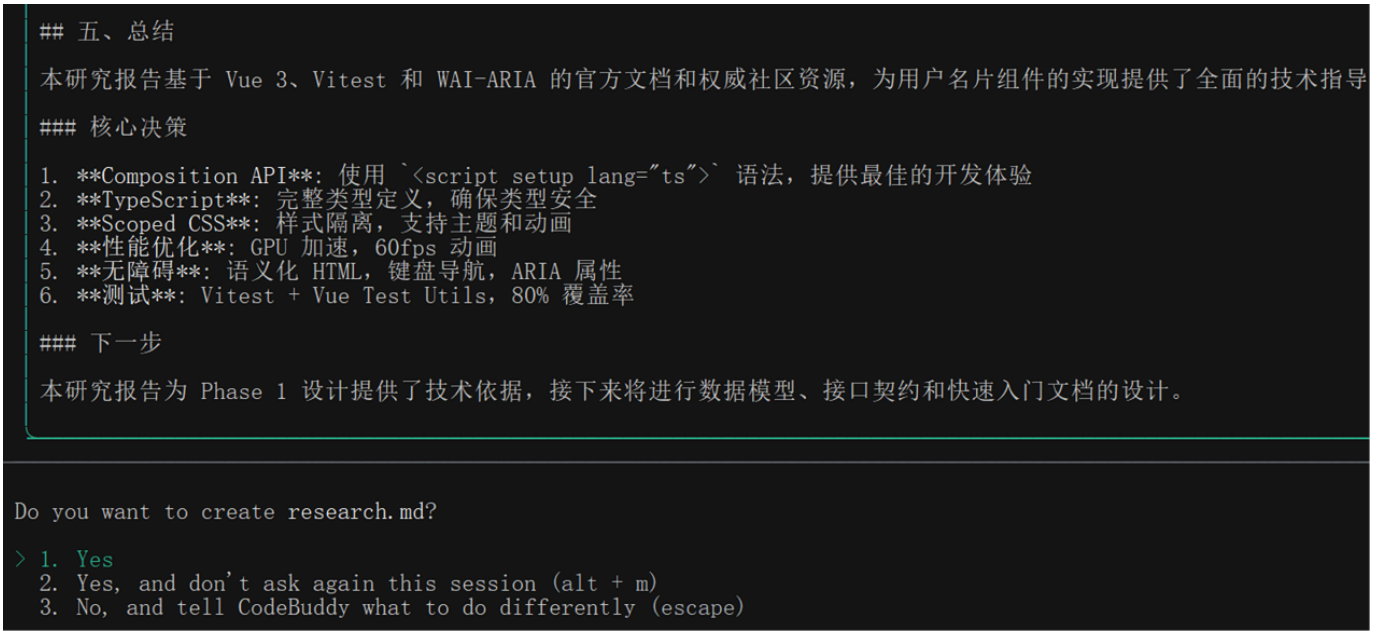


图4-16 research.md文件创建确认界面

后续还将生成以下文件。

- `data-model.md`：定义系统核心数据结构与实体关系的数据模型文档。

- `contracts/api-contract.md`：规范前后端交互接口的契约文档，明确请求响应格式与参数约束。
- `quickstart.md`：包含项目启动指南、核心功能演示与快速上手教程的入门文档。

在生成过程中，系统会针对每个文档的创建请求单独弹出确认提示，你需要逐一选择“Yes”才能完成所有文档的创建。

当系统提示是否进入下一步生成任务文件的操作时，可按“Ctrl+C”结束当前步骤，如图4-17所示。

提示

计划生成后，CodeBuddy可能会继续询问是否进入任务分解等环节。请不要选择“YES”让其创建任务列表文件，因为此时我们需要先停下来审视技术方案的合理性。技术方案是后续开发的核心指导，直接影响系统架构、技术选型和实现难度。如果不进行审视就直接进入任务分解，可能会导致架构不合理、技术选型不当、实现难度过大等问题，增加开发成本和风险。因此，务必先对技术方案进行仔细审视，确保其符合项目需求和实际情况，再进行后续的任务分解和开发工作。

```
---
技术栈确认

- 语言/版本: TypeScript 5.x, Vue 3.4+
- 主要依赖: Vue 3, Vite
- 测试框架: Vitest, Vue Test Utils
- 目标平台: 现代浏览器 (Chrome 90+, Firefox 88+, Safari 14+, Edge 90+)
- 项目类型: UI 组件库
- 性能目标: 60fps 悬停动画, <100ms 渲染时间
- 约束: 不引入第三方 UI 库, 单组件代码 <200 行

---
下一步

Phase 2 规划(任务分解)需要通过 /speckit.tasks 命令创建。该命令将:
1. 根据 Phase 1 的设计文档生成具体的实施任务
2. 创建 tasks.md 文件
3. 列出所有需要完成的开发任务

---
Agent 上下文更新

□ 已成功更新 CodeBuddy 上下文文件(CODEBUDDY.md),添加了:
- 语言: TypeScript 5.x, Vue 3.4+
- 框架: Vue 3, Vite
- 数据库: N/A(组件仅接收 props,不负责数据存储)

这将帮助 AI 助手在后续实施阶段更好地理解项目技术栈。

> 运行 /speckit.tasks 生成任务列表
```

图4-17 步骤结束确认提示界面

总结

`/speckit.plan` 的执行过程比之前的步骤复杂得多，因为它需要：

1.运行内部脚本（`setup-plan.ps1`）→需要确认；

2.查阅外部文档（Vue Test Utils）→需要确认；

3.写入多个文件（`plan.md`、`data-model.md`、`research.md` 等）→逐一确认。

这是正常流程。请耐心等待CodeBuddy完成所有文件的生成，最后再统一审视。

本书建议，至少在第一次完整走通SDD流程时，严格遵守“审视—停止—再启动”的节奏。

4.4 生成高质量Vue组件

4.1至4.3节我们依次完成了宪法确立、规范编写、技术方案制定。本节将使用 `/speckit.tasks` 拆解任务清单，再通过 `/speckit.implement` 执行实现，最终在浏览器中看到可运行的Vue组件。

4.4.1 拆解任务清单

打开系统终端，确保当前工作目录为本项目根目录（即user-card/），然后执行以下命令启动CodeBuddy交互环境。

```
codebuddy
```

进入CodeBuddy交互界面后，输入以下命令，如图4-18所示。

```
/speckit.tasks
```

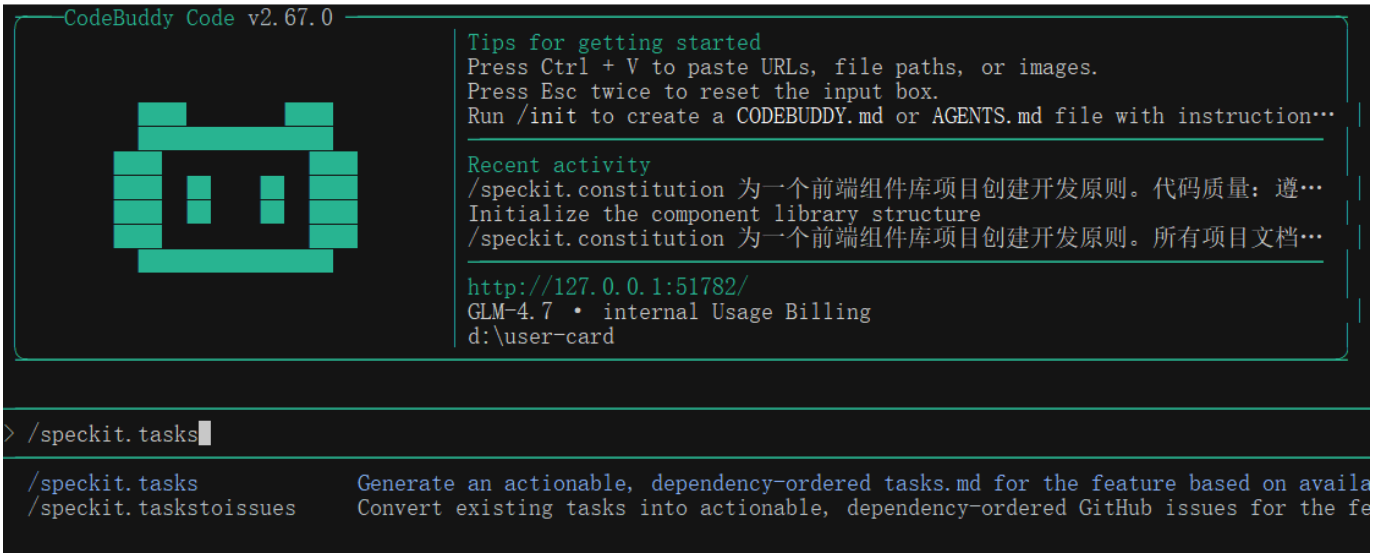


图4-18 输入/speckit.tasks指令的操作界面

CodeBuddy接收 `/speckit.tasks` 命令后，会询问是否确认创建 `tasks.md` 文件，如图4-19所示。

```
## Notes

- [P] tasks = different files, no dependencies
- [Story] label maps task to specific user story for traceability
- Each user story should be independently completable and testable
- Verify tests FAIL before implementing (TDD per Constitution Check)
- Commit after each task or logical group
- Stop at any checkpoint to validate story independently
- Avoid: vague tasks, same file conflicts, cross-story dependencies that break independence
- Tests are REQUIRED per Constitution Check - achieve 80% coverage
- All code must follow Vue 3 Composition API + TypeScript 5.x best practices per research.md
- Component must use <script setup lang="ts"> syntax per research.md Section 1.1
- Scoped CSS with CSS variables per research.md Section 1.5
- GPU-accelerated animations per research.md Section 3.3 (60fps target)
- WAI-ARIA compliance per research.md Section 3.1


Do you want to create tasks.md?

> 1. Yes
  2. Yes, and don't ask again this session (alt + m)
  3. No, and tell CodeBuddy what to do differently (escape)
```

图4-19 确认创建tasks.md文件

选择“Yes”后，CodeBuddy会自动读取已有的 `data-model.md` 、 `api-contract.md` 、 `quickstart.md` 等文件，以指导文件的创建 `tasks.md` 。CodeBuddy读取项目文件的界面如图4-20所示。

```
● Read(specs\001-user-card-component\data-model.md)
  └Read 412 lines (ctrl+o to expand)

● Read(specs\001-user-card-component\contracts\api-contract.md)
  └Read 454 lines (ctrl+o to expand)

● Search(pattern: "**/001-user-card-component/research.md")
  └Found 1 files (ctrl+o to expand)

● Search(pattern: "**/001-user-card-component/quickstart.md")
  └Found 1 files (ctrl+o to expand)

● Read(specs\001-user-card-component\research.md)
  └Read 706 lines (ctrl+o to expand)

● Read(specs\001-user-card-component\quickstart.md)
  └Read 649 lines (ctrl+o to expand)

● Now let me load the tasks template:

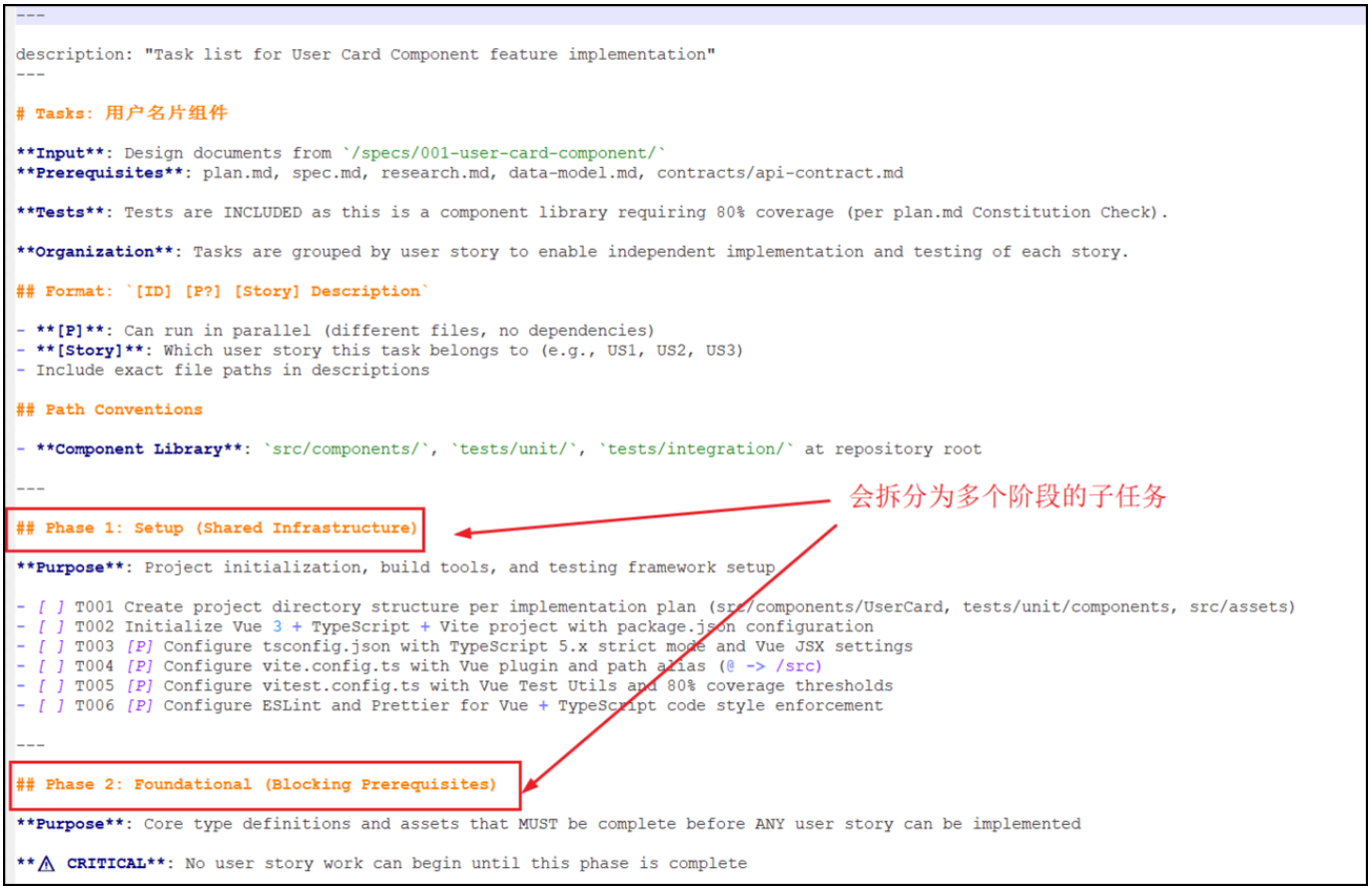
● Read(.specify\templates\tasks-template.md)
  └Read 252 lines (ctrl+o to expand)

● Now I'll generate the tasks.md file based on all the design documents:

● Write(specs\001-user-card-component\tasks.md)
  └Wrote 302 lines (ctrl+o to expand)
```

图4-20 CodeBuddy读取项目文件的界面

生成的任务清单文件 `tasks.md` 会将开发流程拆解为多个阶段的子任务（见图4-21），各阶段任务清晰明确，便于分步执行与进度跟踪。




```
---

description: "Task list for User Card Component feature implementation"
---

# Tasks: 用户名片组件

**Input**: Design documents from `/specs/001-user-card-component/`
**Prerequisites**: plan.md, spec.md, research.md, data-model.md, contracts/api-contract.md

**Tests**: Tests are INCLUDED as this is a component library requiring 80% coverage (per plan.md Constitution Check).

**Organization**: Tasks are grouped by user story to enable independent implementation and testing of each story.

## Format: `[ID] [P?] [Story] Description`

- **[P]**: Can run in parallel (different files, no dependencies)
- **[Story]**: Which user story this task belongs to (e.g., US1, US2, US3)
- Include exact file paths in descriptions

## Path Conventions

- **Component Library**: `src/components/`, `tests/unit/`, `tests/integration/` at repository root

---

## Phase 1: Setup (Shared Infrastructure)

**Purpose**: Project initialization, build tools, and testing framework setup

- [ ] T001 Create project directory structure per implementation plan (src/components/UserCard, tests/unit/components, src/assets)
- [ ] T002 Initialize Vue 3 + TypeScript + Vite project with package.json configuration
- [ ] T003 [P] Configure tsconfig.json with TypeScript 5.x strict mode and Vue JSX settings
- [ ] T004 [P] Configure vite.config.ts with Vue plugin and path alias (@ -> /src)
- [ ] T005 [P] Configure vitest.config.ts with Vue Test Utils and 80% coverage thresholds
- [ ] T006 [P] Configure ESLint and Prettier for Vue + TypeScript code style enforcement

---

## Phase 2: Foundational (Blocking Prerequisites)

**Purpose**: Core type definitions and assets that MUST be complete before ANY user story can be implemented

**⚠ CRITICAL**: No user story work can begin until this phase is complete
```

图4-21 任务清单的阶段化子任务拆分示例



提示

注意：在实际操作中，建议仔细审视任务拆解的合理性。若发现任何不合理之处，请及时向Codebuddy反馈并要求修正。

确认无误后，按“ `Ctrl+C` ”退出CodeBuddy交互界面，返回系统终端。



提示

特别说明：如果已经确定要继续执行下一步，则此处可以不退出，直接继续执行后续命令即可。此处提示退出主要是为了明确每个阶段的结束点。

4.4.2 执行实现

打开系统终端，输入以下命令启动CodeBuddy交互环境。

```
codebuddy
```

进入CodeBuddy交互界面后，输入以下命令（有两种选择）。

```
#选择一（官方命令）：一次性实现所有任务，成功率较低
  /speckit.implement
  #选择二（推荐）：分阶段实现任务，成功率更高（实践中很多人反馈执行失败，往往是因为直接使用了第一种命令）
  /speckit.implement Phase 1
```

建议先完成Phase 1任务，再依次推进后续阶段。

CodeBuddy收到实现命令后，会自动读取任务清单文件 `tasks.md` ，并严格按照清单中定义的任务顺序逐项执行开发工作，如图4-22所示。

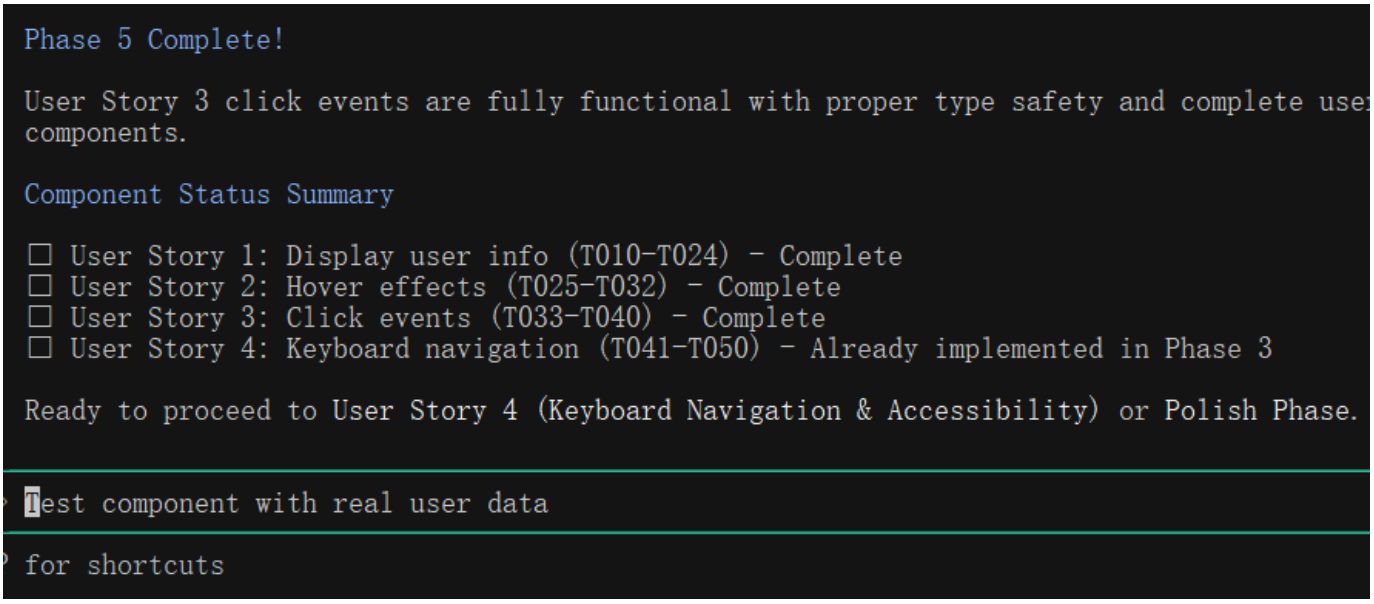


图4-22 CodeBuddy分步执行任务的日志示例

接下来，只需跟随CodeBuddy的交互指引，在弹出的确认提示中选择“Yes”并确认，即可自动完成后续所有开发任务。

执行完成后，CodeBuddy会自动为我们搭建出完整的代码工程（见图4-23），包含组件实现、测试用例、配置文件等核心内容。

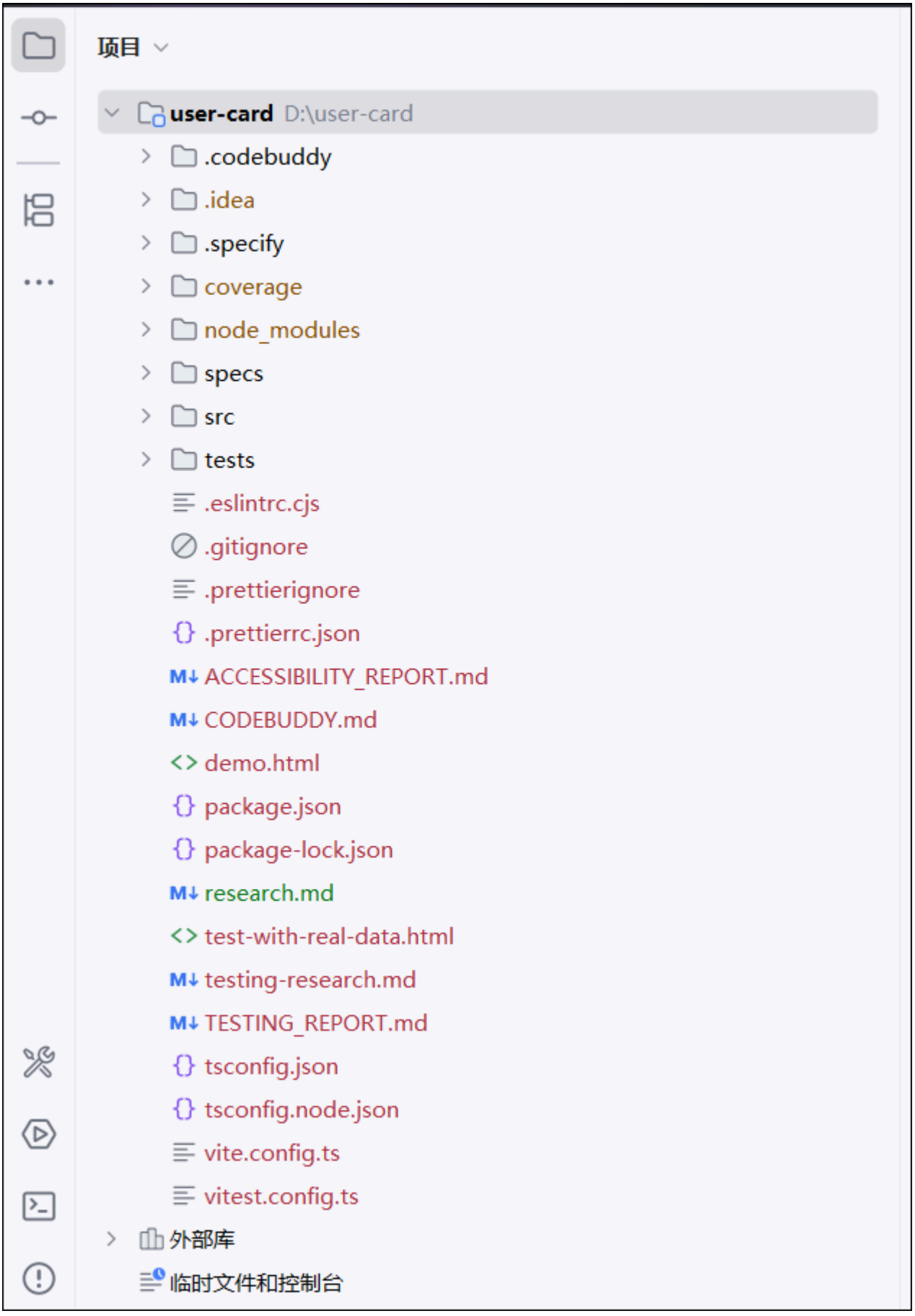


图4-23 用户名片组件项目完整代码结构

项目的详细说明文档如图4-24所示。



图4-24 项目的详细说明文档

4.4.3 验证项目效果

完成任务拆解与实现后，我们需要对用户名片组件进行全面验证，确保其功能、性能和交互体验符合预期。具体的验证步骤如下。

（1）在系统终端中输入 `npm run dev` 命令，启动开发服务器，如图4-25所示。



图4-25 启动开发服务器

（2）在浏览器中访问http://localhost:5173，页面将展示三个样式统一、信息完整的用户名片组件。若组件正常渲染且交互功能可用，则表明基于Vue 3的组件封装工作已成功完成。用户名片组件渲染效果如图4-26所示。



图4-26 用户名片组件渲染效果

4.5 回顾：完整流程

表4-1总结了完成用户名片组件项目所需的全部指令及对应产出。

表4-1 完成用户名片组件项目所需的全部指令及对应产出

步骤	节	命令/操作	产出	关键动作
1	4.1	`specify init`	项目骨架	初始化项目
2	4.1	`/speckit.constitution`	`constitution.md`	确立项目宪法
3	4.2	`/speckit.specify`	`spec.md`	编写组件规范
4	4.3	`/speckit.plan`	`plan.md`、`data-model.md`等	制定技术方案
5	4.4	`/speckit.tasks`	`tasks.md`	拆解任务清单
6	4.4	`/speckit.implement`	完整的代码工程	执行实现
7	4.4	通过浏览器访问并验证	—	验证项目效果

每一步之间都遵循着“审视—停止—再启动”的节奏。这就是SDD的完整闭环——从一个模糊的想法（“我需要一个用户名片组件”）到一个可运行、有测试、符合规范的Vue组件。

第5章　【进阶】状态管理：任务看板

在第4章中，我们通过用户名片组件的开发，初步掌握了Spec-Kit规范驱动开发的基础流程。本章将进一步深入，通过构建一个功能更复杂的任务看板应用，探索状态管理、拖拽交互、多视图切换等进阶场景。我们将继续遵循Spec-Kit的工作流程，从项目初始化、宪法确立开始，逐步完成规范编写、技术方案制定、任务拆解与实现，最终实现一个具备任务创建、拖拽移动、标题编辑、搜索筛选等完整功能的任务看板应用。

5.1 项目初始化及宪法确立

任务看板是一个独立的应用，与第4章的用户名片组件无关，因此需要从零开始构建。本节将完成项目初始化、宪法确立。

在系统终端中执行以下命令，创建任务看板项目：

```
specify init task-board--ai codebuddy
```

项目创建完成后，执行以下命令，进入项目目录：

```
cd task-board
```

最后执行如下命令启动CodeBuddy，进入交互开发环境：

```
codebuddy
```

执行上述命令后，终端将显示CodeBuddy交互界面，如图5-1所示。

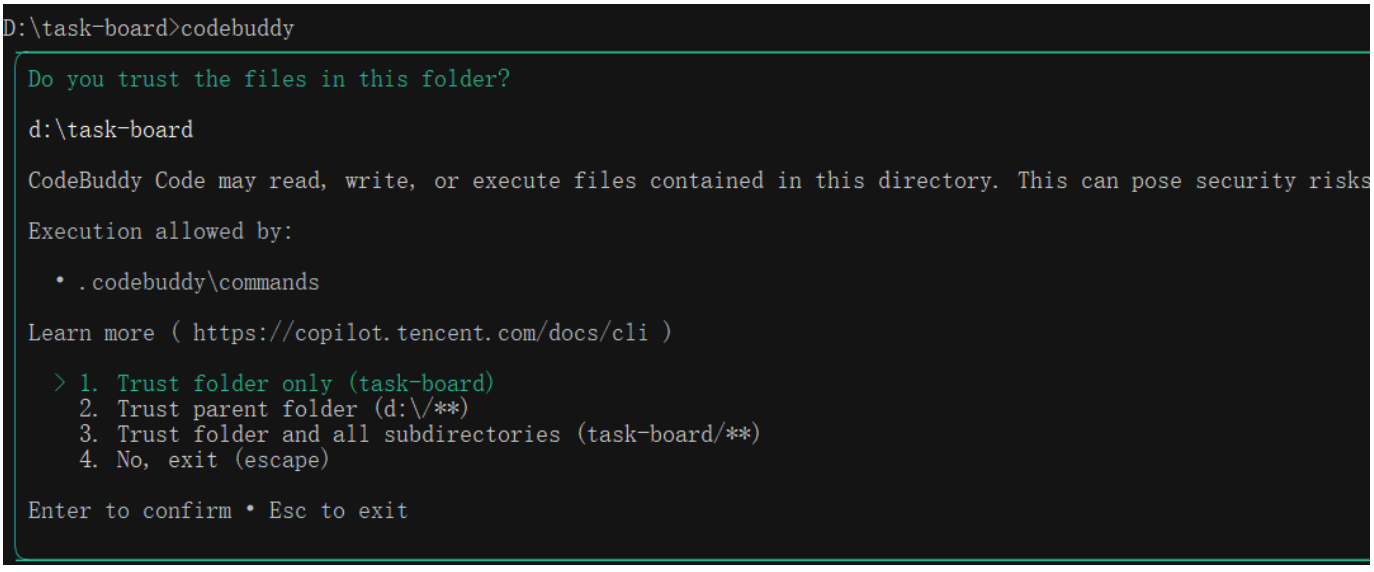


图5-1 CodeBuddy交互界面

进入CodeBuddy交互界面后，首先要确立宪法，输入以下内容：

/speckit.constitution为一个前端任务看板应用创建开发原则。所有项目文档使用中文编写。代码质量：遵循官方推荐的代码风格，组件命名遵循PascalCase规范，每个组件文件不超过300行。测试：核心业务逻辑（状态管理、拖拽、搜索）必须有对应的单元测试，覆盖率不低于80%。用户体验：所有操作反馈延迟不超过200ms，拖拽过程必须流畅（60fps），支持键盘操作。性能：首次加载时间不超过2s，100个任务以内渲染无卡顿。技术治理：禁止引入非必要的第三方依赖，引入新依赖必须说明选型理由并记录在案。

CodeBuddy完成宪法文件的内容编写后，会向我们确认是否创建该文件。CodeBuddy确认创建宪法文件界面如图5-2所示。

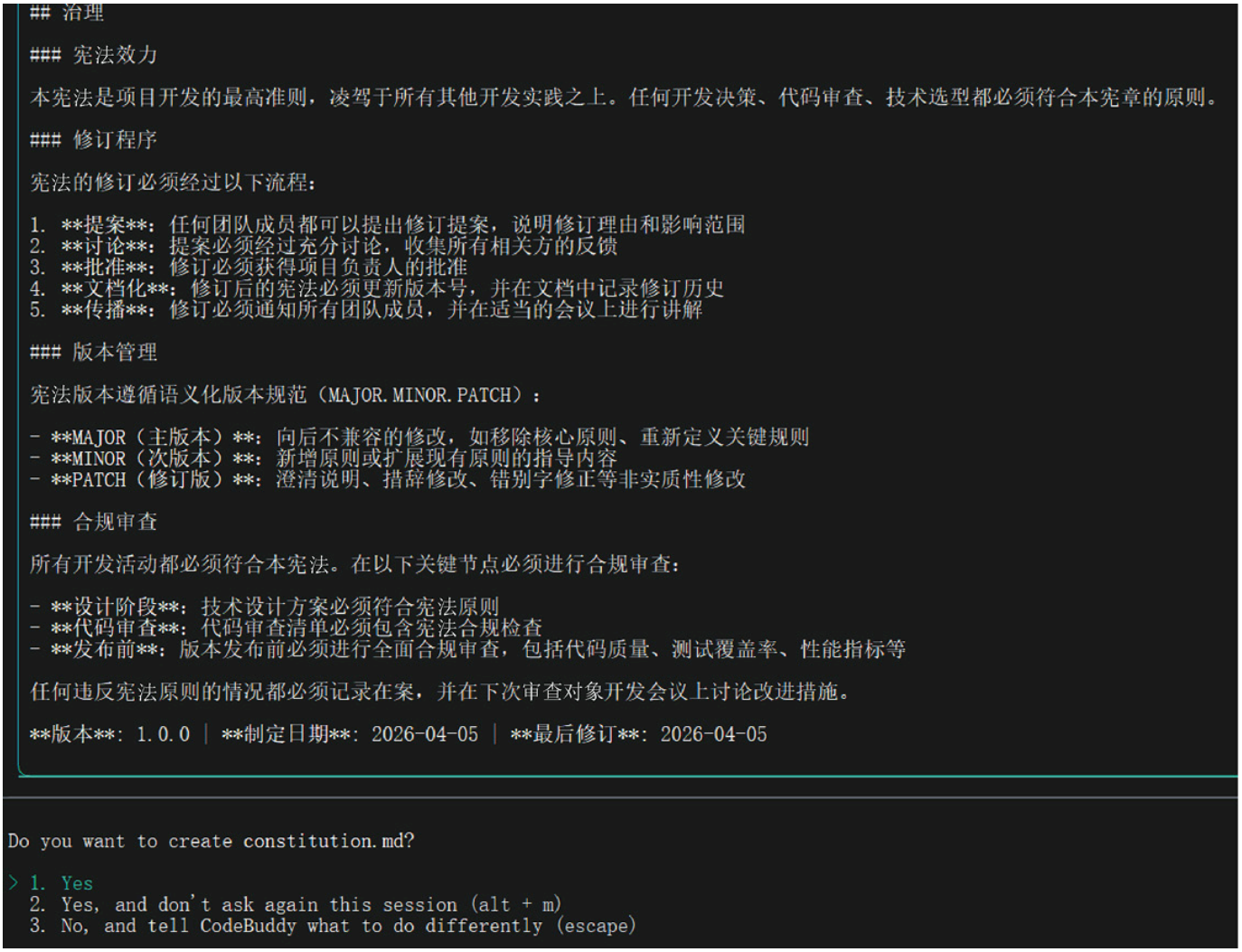


图5-2 CodeBuddy确认创建宪法文件界面

在该界面中选择“Yes”并确认后，系统将自动完成宪法文件的创建流程。宪法文件自动创建流程界面如图5-3所示。

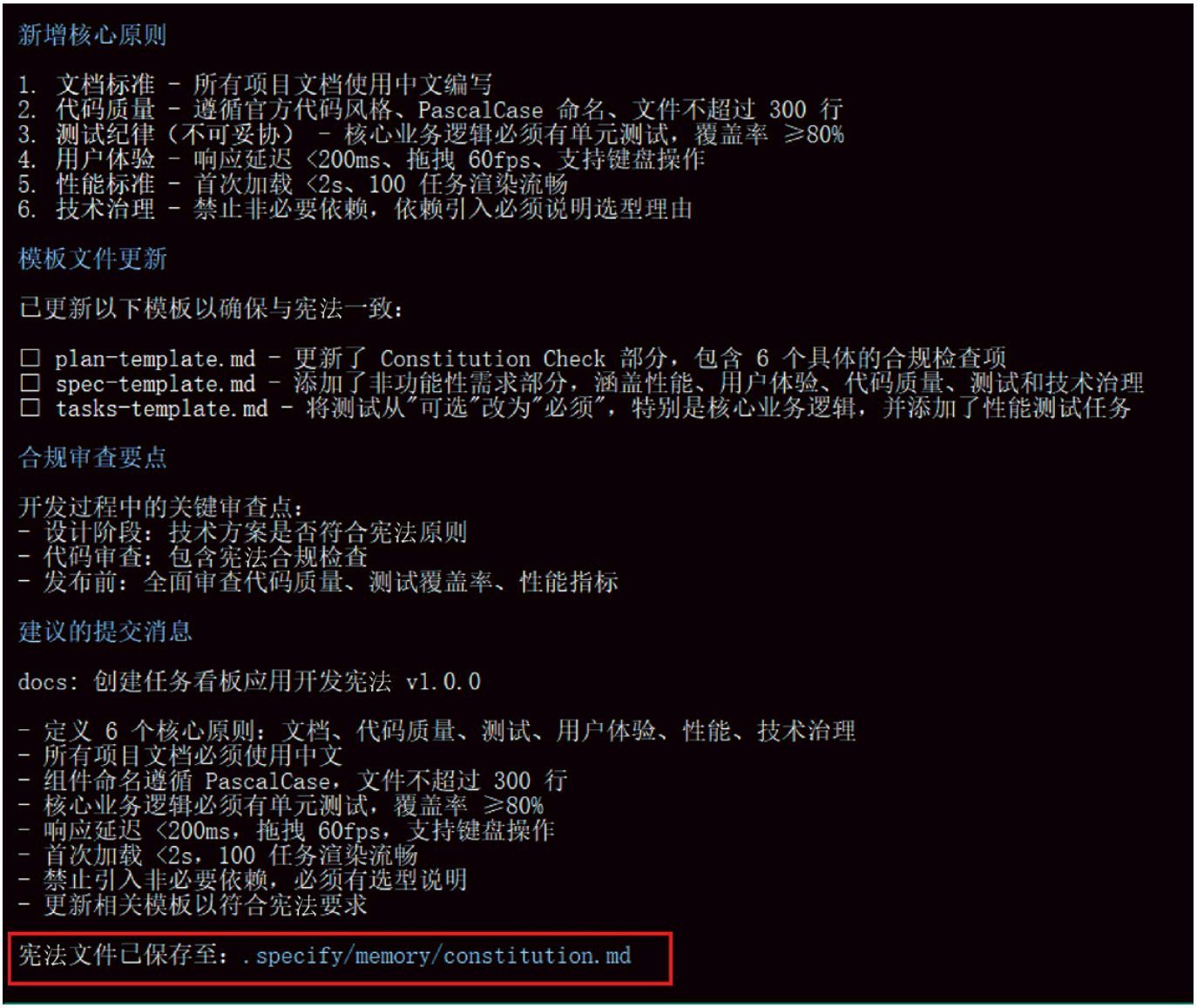


图5-3 宪法文件自动创建流程界面

CodeBuddy生成宪法文件后，我们需要仔细审视宪法内容，确认其符合项目要求。确认无误后，在CodeBuddy交互界面中输入以下命令，输入停止命令界面如图5-4所示。

```
宪法已确认，请停止当前操作。我将在准备好后使用/speckit.specify定义需求。
```

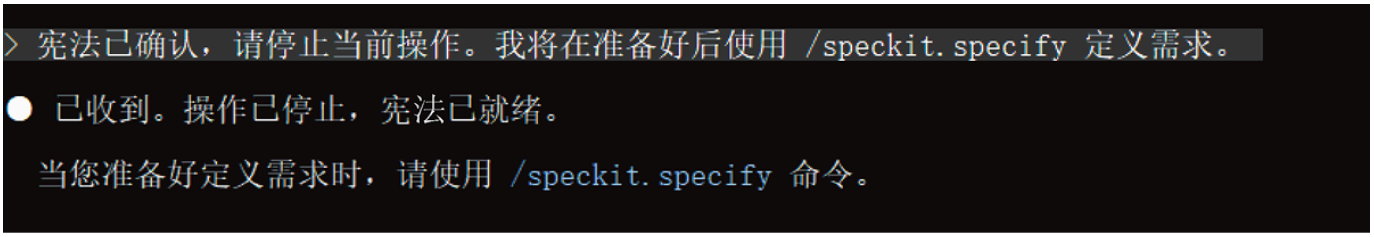


图5-4 输入停止命令界面

输入完成后，按 `Ctrl+C` 组合键退出CodeBuddy交互环境，回到系统终端。

5.2 编写任务看板规范

接下来，我们将通过编写任务看板的功能规范，明确系统的核心业务逻辑与交互规则。请进入CodeBuddy交互界面后，输入以下内容：

```
/speckit.specify构建一个任务看板应用。看板包含4列：待处理、进行中、审核中、已完成。用户可以在输入框中输入任务标题并按Enter键创建新任务，新任务默认添加到“待处理”列。每张任务卡片显示标题和创建时间。用户可以拖拽任务卡片，让其在不同列之间移动，拖拽完成后任务状态自动更新。单击卡片可以编辑标题，编辑完成后按Enter键保存、按Escape键取消。每列顶部显示该列的任务数量统计。支持按关键词搜索任务，匹配的卡片会高亮显示。所有项目文档使用中文编写。
```

CodeBuddy规范文件生成确认界面如图5-5所示。选择“Yes”，CodeBuddy接收命令后，会自动生成规范文件。

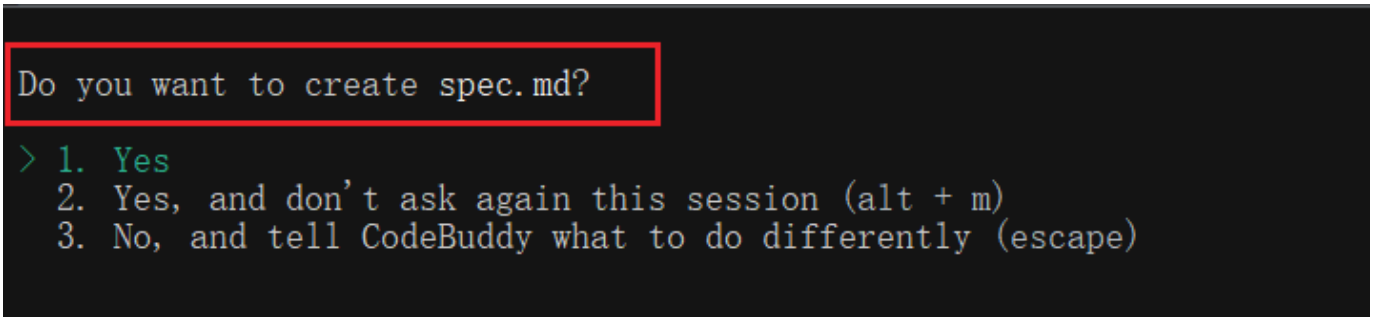


图5-5 CodeBuddy规范文件生成确认界面

当系统提示规范内容可能存在歧义或需要补充细节时（规范澄清提示示例如图5-6所示），建议使用 `/speckit.clarify` 命令对需求进行澄清，确保所有业务规则和边界条件都被准确定义。

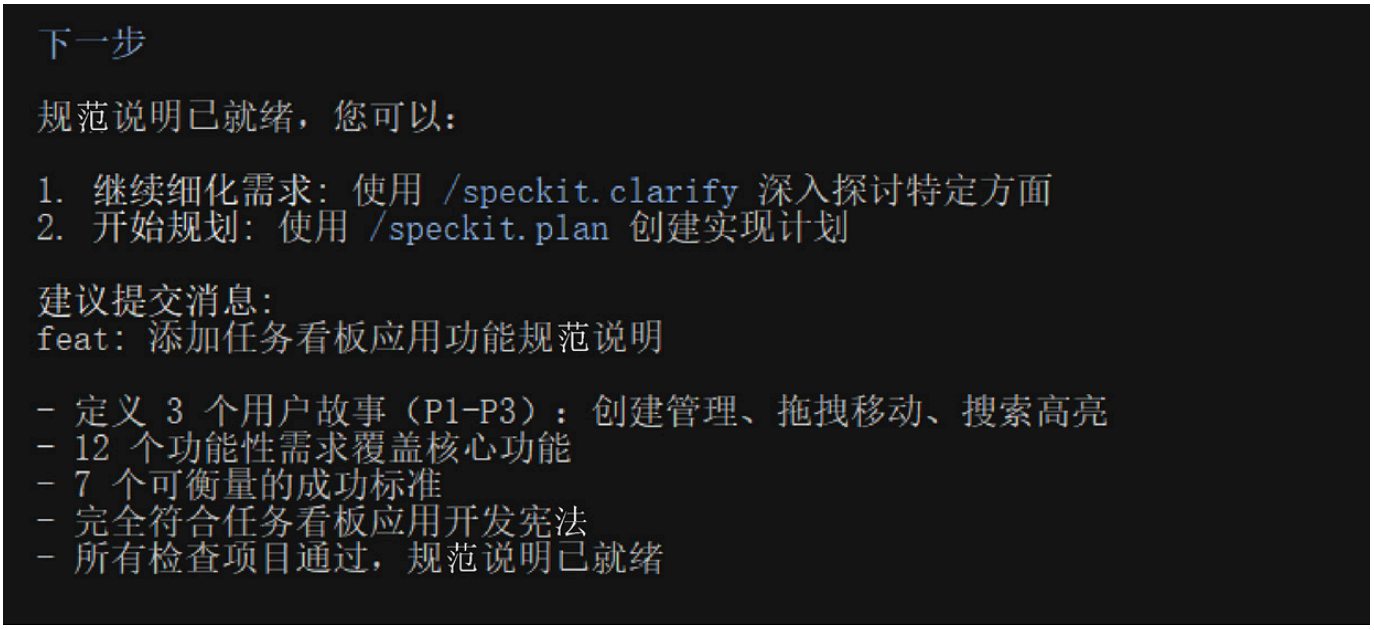


图5-6 规范澄清提示示例

此时，我们可以向CodeBuddy输入以下命令，暂停当前操作并准备进入需求澄清阶段：

```
规范已确认，请停止当前操作。我将在准备好后使用/speckit.clarify澄清需求。
```

完成上述操作后，在CodeBuddy交互界面中继续输入以下命令，启动需求澄清流程：

```
/speckit.clarify重点关注拖拽交互、搜索功能和边界条件中未明确的细节。
```

CodeBuddy将自动扫描规范文档，识别出拖拽交互逻辑、搜索功能实现以及各类边界条件中描述模糊或缺失的部分，并针对这些不明确的区域逐一生成澄清问题。

我们需要针对这些澄清问题逐一进行解答，CodeBuddy会将所有问答内容自动追加到规范文档的“Clarifications”部分，形成完整的需求补充说明。

当我们确认规范文档内容完整、所有需求细节均已明确后，可以再次告知CodeBuddy停止当前操作，输入以下内容：

```
规范与澄清已确认，请停止当前操作。我将在准备好后使用/speckit.plan制定技术方案。
```

至此，我们已完成任务看板规范的编写与澄清工作，明确了系统的核心业务逻辑与交互规则，为5.3节的技术方案制定奠定了坚实基础。

5.3 制定技术方案

5.2节我们编写了任务看板的功能规范，并通过 `/speckit.clarify` 对需求进行了澄清。本节将使用 `/speckit.plan` 制定技术方案——任务看板这类多状态应用的技术方案，复杂度远高于第4章的用户名片组件。

我们进入CodeBuddy交互界面后，输入以下内容：

```
/speckit.plan使用React 18、Vite和TypeScript开发。状态管理使用React Context和useReducer，不引入Redux等第三方状态管理库。拖拽功能使用@hello-pangea/dnd实现。样式使用CSS Modules，不引入CSS框架。数据存储使用localStorage，不引入后端服务。测试使用Vitest和React Testing Library。所有项目文档使用中文编写。
```

当CodeBuddy接收到命令后，会自动读取项目中已有的宪法文件和规范文档，并结合你提供的技术方案描述，生成一套完整的项目计划。随后，系统会询问你是否确认创建 `plan.md` 文件，创建 `plan.md` 文件确认界面如图5-7所示。

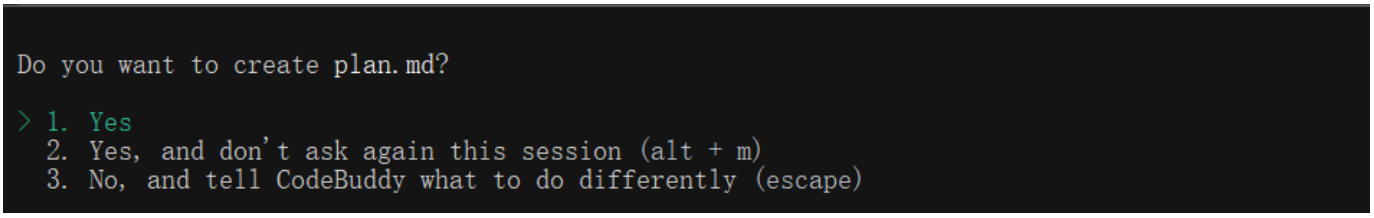


图5-7 创建plan.md文件确认界面

选择“Yes”确认后，系统将自动创建一系列核心文档，包括用于记录技术选型调研过程与决策依据的 `research.md` 、定义任务看板数据结构的 `data-model.md` ，以及包含项目快速启动指南的 `quickstart.md` 。

最后，当看到如下汇总信息时，表明项目计划已经完成。项目计划汇总信息如图5-8所示。

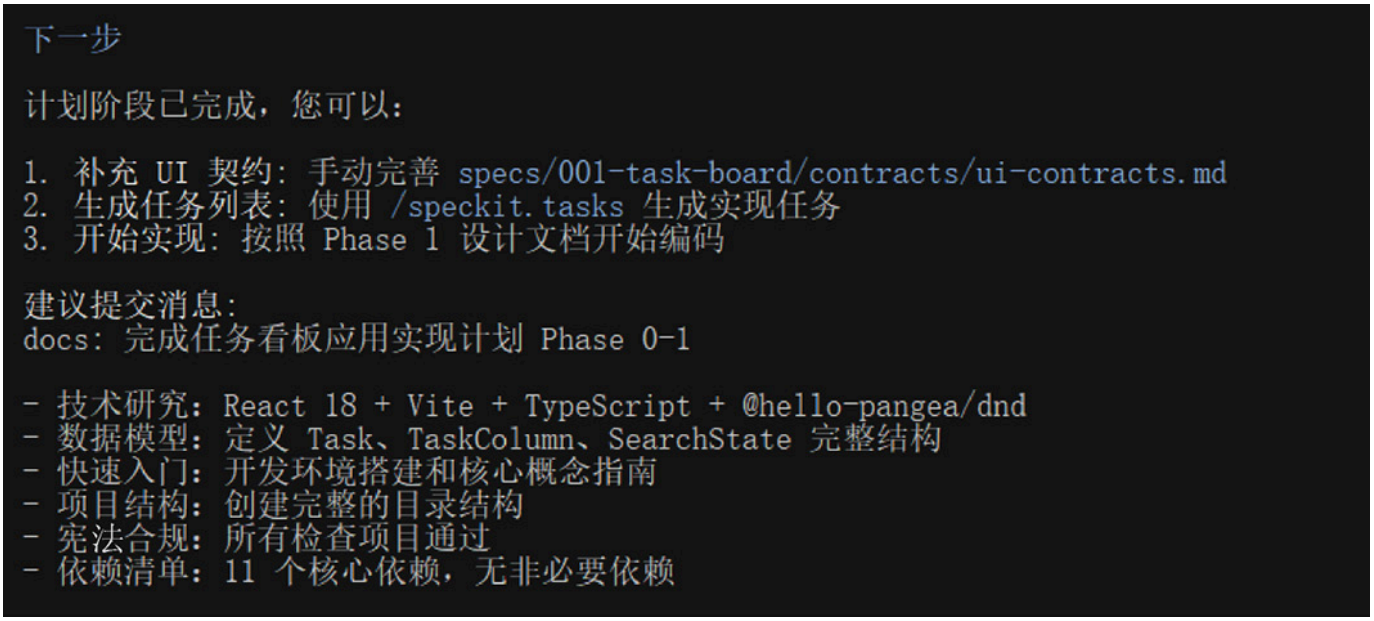


图5-8 项目计划汇总信息

我们可以在CodeBuddy交互界面输入以下内容：

> 计划已确认，请停止当前操作。我将在准备好后使用/speckit.tasks拆解任务。

5.4 任务拆解与实现：从列表到看板

在5.2节和5.3节中，我们完成了规范编写和技术方案制定。本节将使用 `/speckit.tasks` 拆解任务，再使用 `/speckit.implement` 执行实现，最终在浏览器中查看可运行的任务看板。

5.4.1 拆解任务

进入CodeBuddy交互界面后，输入以下命令：

```
/speckit.tasks
```

CodeBuddy会自动读取项目中已有的规范文档和技术方案，生成详细的任务清单文件`tasks.md`，将开发流程拆解为多个可执行的子任务，便于团队成员分工协作与跟踪进度。任务清单生成界面如图5-9所示。



图5-9 任务清单生成界面

任务清单文件tasks.md创建完毕之后，CodeBuddy会提示我们可以进入下一步实现任务。CodeBuddy任务清单完成提示界面如图5-10所示。

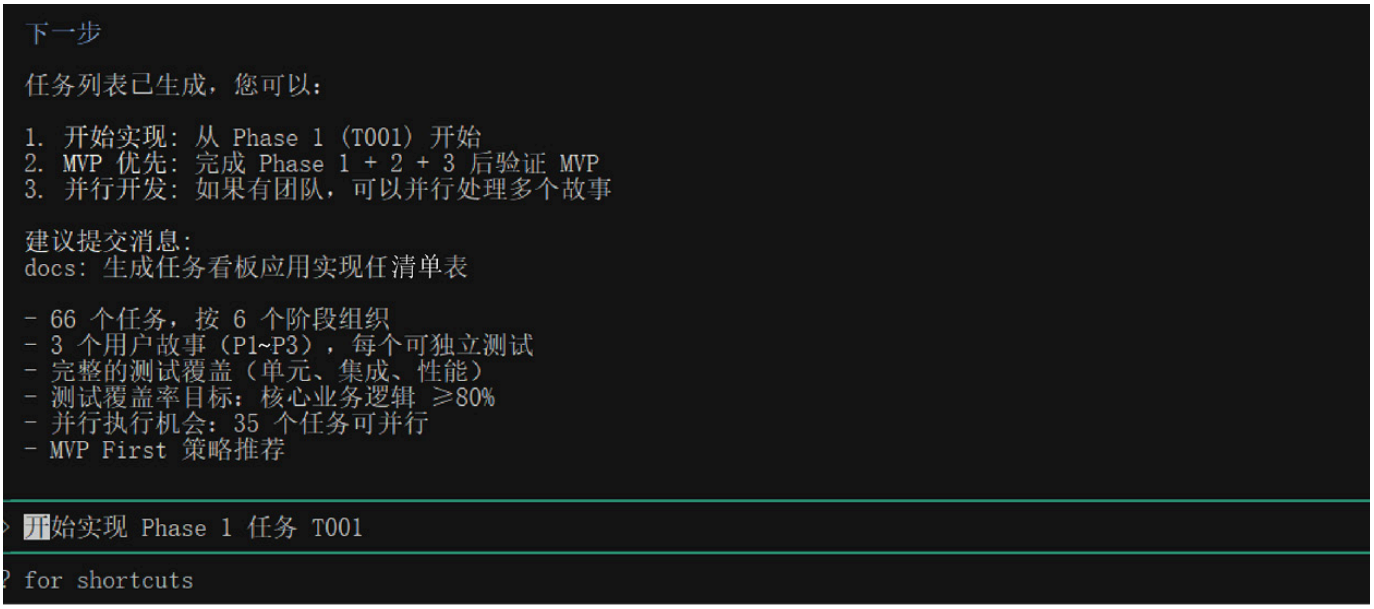


图5-10 CodeBuddy任务清单完成提示界面

重要提示：不要让AI自动推进到下一步。

任务清单生成后，CodeBuddy可能会询问是否直接执行实现。不要一路选择“YES”。

若发现任务拆分存在遗漏或不合理之处，可直接在CodeBuddy交互界面中补充或调整任务内容。

当你仔细审视任务清单，确认这些任务能完整覆盖开发流程且拆分合理后，可告知CodeBuddy停止当前环节。输入如下命令：

```
任务清单已确认，请停止当前操作。我将使用/speckit.implement执行实现。
```

如果发现 `task.md` 文件异常，则可以给CodeBuddy反馈，在交互界面输入如下命令：

```
tasks.md文件内容为空，请重新生成完整的任务清单并写入该文件。
```

经过上述步骤，我们成功生成了完整的任务清单文件，任务清单文件内容展示如图5-11所示。

```
## Phase 1: Setup (Shared Infrastructure)

**目的**: 项目初始化和基础结构

- [ ] T001 创建项目结构 per implementation plan
- [ ] T002 初始化 React + Vite + TypeScript 项目依赖
- [ ] T003 [P] 配置 TypeScript 严格模式和路径别名
- [ ] T004 [P] 配置 Vitest 测试环境
- [ ] T005 [P] 配置 CSS Modules 支持
- [ ] T006 [P] 创建 package.json 脚本（dev, test, build, preview）

---

## Phase 2: Foundational (Blocking Prerequisites)

**目的**: 在任何用户故事实现之前必须完成的核心基础设施

**⚠ CRITICAL**: 在此阶段完成之前，不能开始任何用户故事工作

- [ ] T007 创建 TypeScript 类型定义在 src/types/index.ts（Task, TaskStatus, TaskBoardState, TaskAction）
- [ ] T008 [P] 实现 localStorage 工具函数在 src/utils/storage.ts（saveTasks, loadTasks, clearTasks）
- [ ] T009 [P] 实现日期格式化工具函数在 src/utils/date.ts（formatDate）
- [ ] T010 [P] 实现搜索工具函数在 src/utils/search.ts（matchTasks）
- [ ] T011 实现 React Context（TaskContext）在 src/context/TaskContext.tsx
- [ ] T012 实现 useReducer（taskReducer）在 src/context/taskReducer.ts
- [ ] T013 [P] 单元测试：taskReducer 的所有 action 类型和状态转换在 tests/unit/context/taskReducer.test.ts
- [ ] T014 [P] 单元测试：storage 工具函数（saveTasks, loadTasks）在 tests/unit/utils/storage.test.ts
- [ ] T015 [P] 单元测试：date 工具函数（formatDate）在 tests/unit/utils/date.test.ts
- [ ] T016 [P] 单元测试：search 工具函数（matchTasks）在 tests/unit/utils/search.test.ts
```

图5-11 任务清单文件内容展示

5.4.2 执行实现

接下来，我们将基于已生成的任务清单，通过CodeBuddy完成任务的具体实现。进入CodeBuddy交互界面后，输入以下命令：

```
#默认命令
  /speckit.implement
  #强烈推荐按阶段执行
  /speckit.implement Phase 1
```

CodeBuddy接收到实现命令后，会自动读取任务清单文件 `tasks.md` ，并严格按照清单中定义的任务顺序逐项执行开发工作。执行过程中，系统会分阶段推进任务，完成Phase 1任务后，会询问是否继续执行Phase 2任务。分阶段任务执行确认界面如图5-12所示。

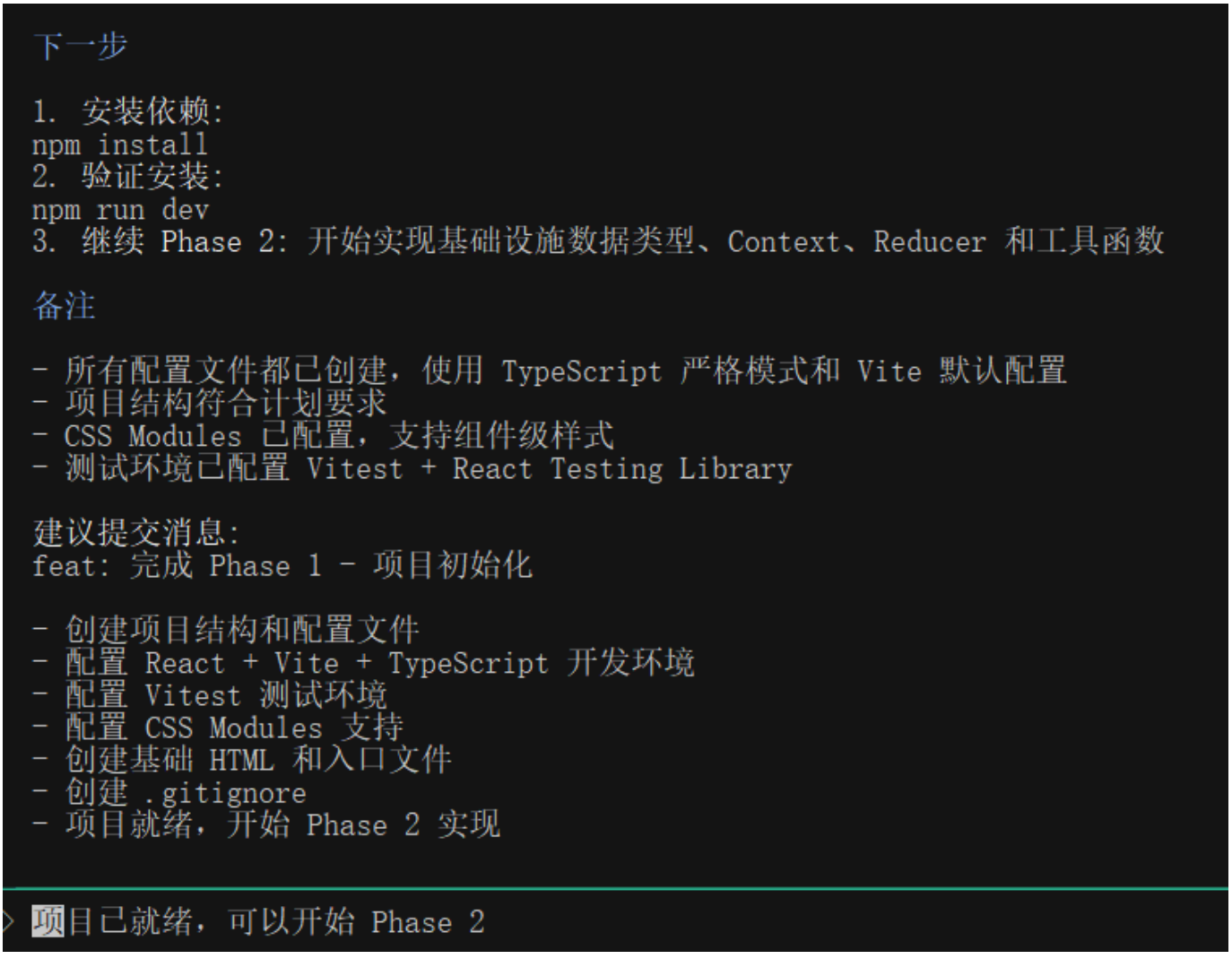


图5-12 分阶段任务执行确认界面

5.4.3 在浏览器中验证

所有任务均完成后，请在系统终端中启动开发服务器，执行以下命令：

```
npm run dev
```

在浏览器中访问http://localhost:5173（或终端中显示的地址），系统将实现以下功能：按任务状态分类展示任务，将任务清晰划分为“待处理”“进行中”“已完成”3类，各类任务独立呈现，便于用户直观追踪任务进展。任务状态分类展示界面如图5-13所示。

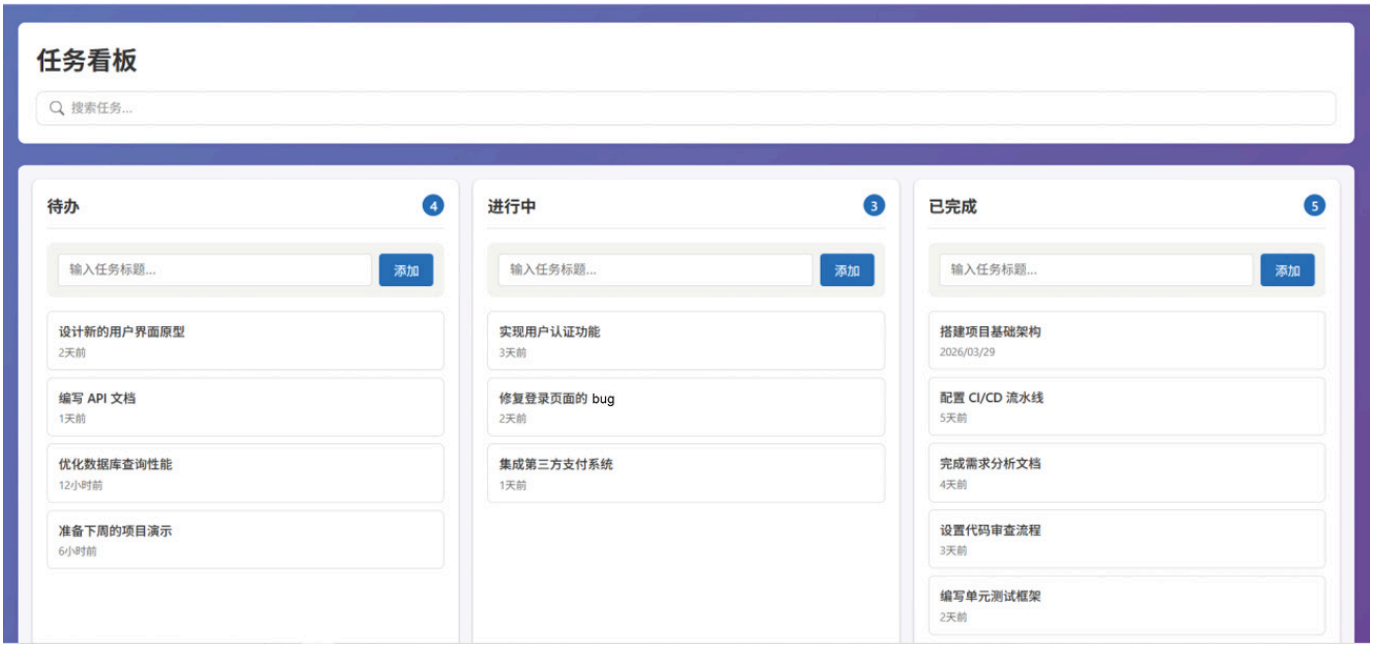


图5-13 任务状态分类展示界面

系统支持任务卡片的跨列拖拽功能，用户可通过鼠标拖动任务卡片在不同列之间自由移动，任务状态将随所在列自动更新。任务卡片跨列拖拽操作演示如图5-14所示。

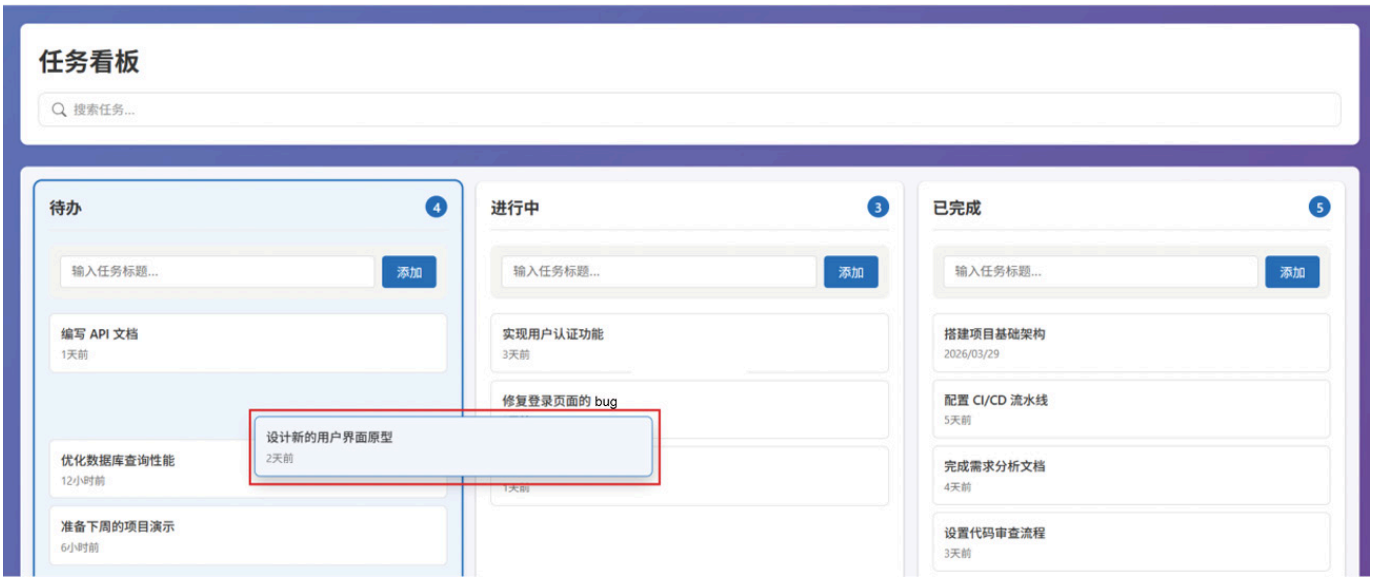


图5-14 任务卡片跨列拖拽操作演示

系统支持按关键词搜索任务，输入关键词后，所有列中匹配的任务卡片将高亮显示，帮助用户快速定位目标任务。任务看板搜索功能如图5-15所示。

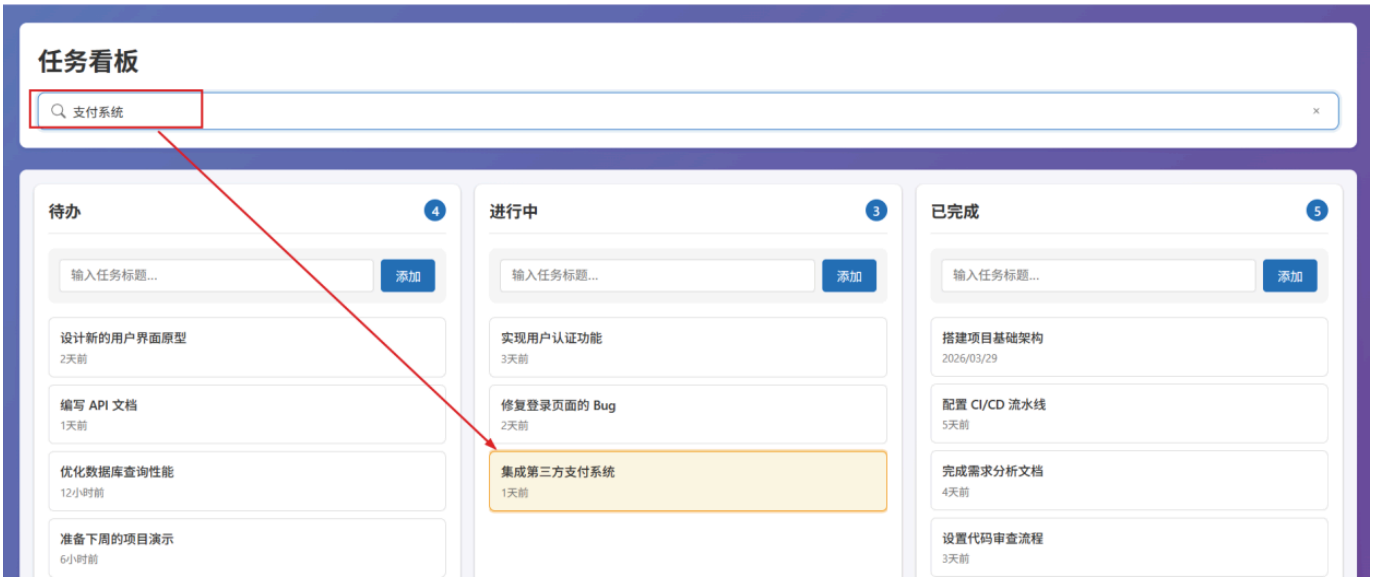


图5-15 任务看板搜索功能

特别说明：当前前端使用localStorage存储数据。在第6章中，我们将定义API契约，并将数据层替换为HTTP请求，实现前后端联调测试。

第6章　【基础】构建后端API与前后端联调

前端应用的开发离不开后端API的支持。本章将指导你构建后端API，并完成前后端的联调测试，让你掌握规范驱动开发在全栈开发中的应用方法。

6.1 项目初始化及宪法确立

在第5章中，我们完成了任务看板的前端，所有数据存储在浏览器的localStorage中。从本章开始，我们将为任务看板构建后端API服务，将前端的数据层从localStorage替换为真实的HTTP请求。

后端API服务项目作为独立项目存在，拥有专属代码仓库、独立项目宪法及完整开发流程。前后端通过API契约（OpenAPI规范）关联——后端提供接口，前端调用接口，两者通过契约保持一致。

6.1.1 初始化后端API服务项目

接下来，我们将创建独立的后端API服务项目。请打开系统终端，执行以下命令：

```
specify init task-board-backend--ai codebuddy
```

在终端提示选择操作模式时，选择“ps”选项即可完成项目初始化。后端API服务项目初始化操作界面如图6-1所示。

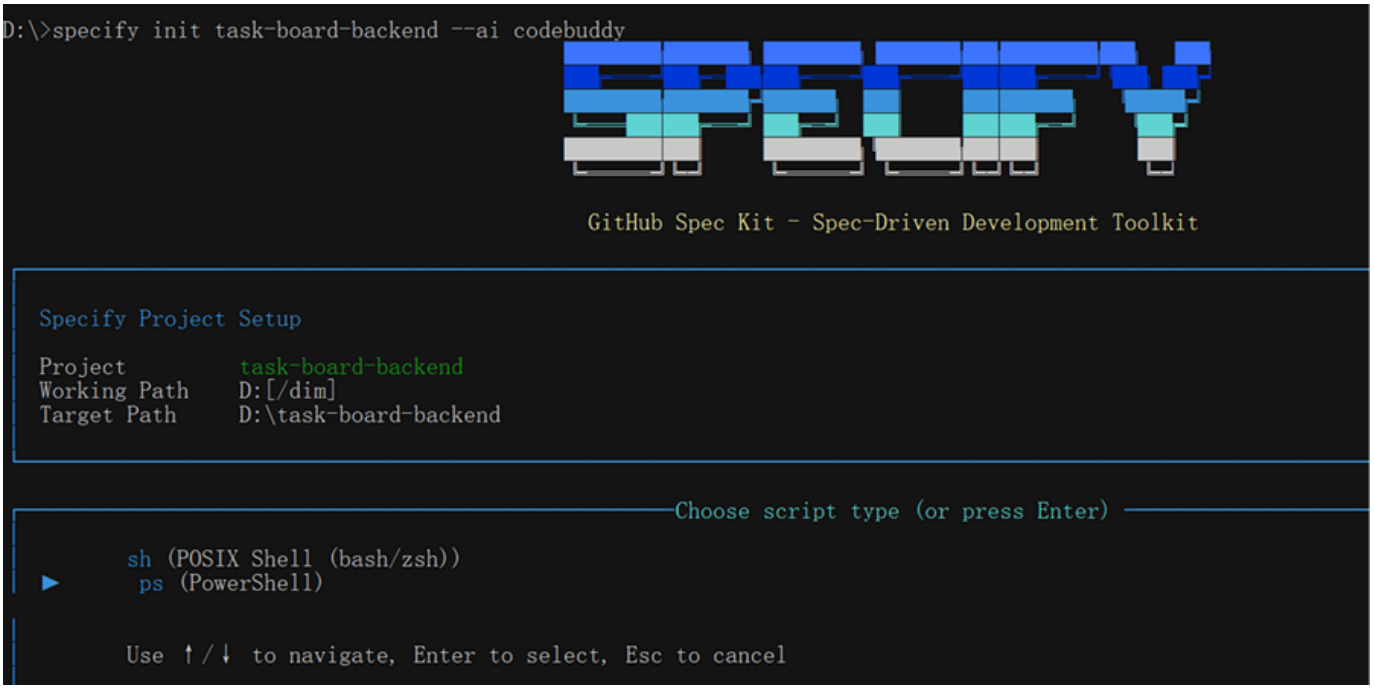


图6-1 后端API服务项目初始化操作界面

6.1.2 编写后端API服务项目宪法

打开终端，执行以下命令进入后端API服务项目根目录：

```
cd task-board-backend
```

在项目根目录下，执行以下命令启动CodeBuddy：

```
codebuddy
```

首次启动时，系统会提示是否信任当前目录，请选择信任。CodeBuddy首次启动信任目录提示界面如图6-2所示。

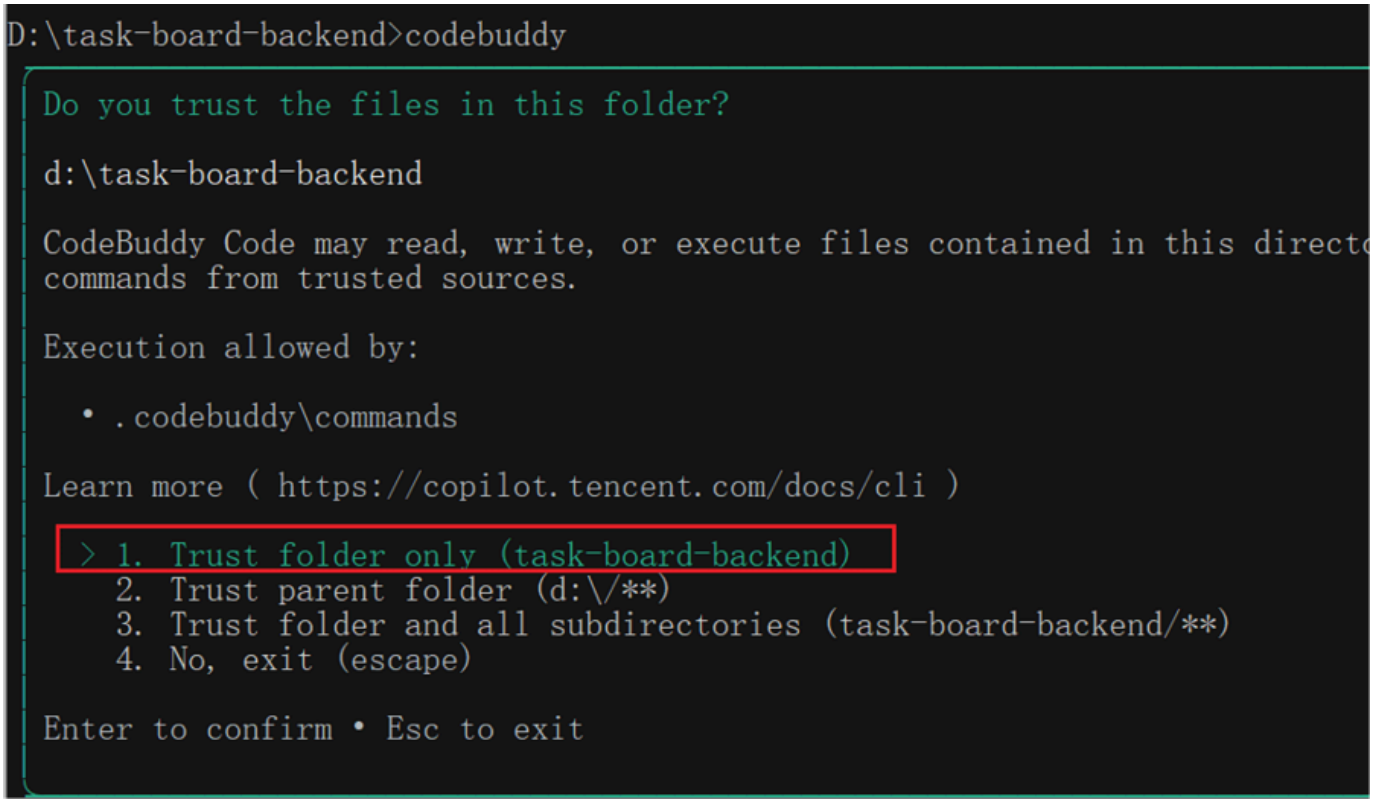


图6-2 CodeBuddy首次启动信任目录提示界面

进入CodeBuddy交互界面后，输入以下内容：

```
/speckit.constitution为任务看板应用的后端API服务创建开发原则。所有项目文档使用中文编写。代码质量：遵循官方推荐的代码风格，使用分层架构（接口层、业务层、数据访问层），每层职责单一，禁止跨层调用，每个文件不超过300行。测试：每个接口端点必须有集成测试，核心业务
```

逻辑单元测试覆盖率不低于80%，数据库集成测试必须使用真实数据库环境，测试必须覆盖正常流程、异常流程和边界条件。性能：接口P95响应时间低于200ms，数据库查询必须有索引优化，禁止N+1查询。数据一致性：所有涉及状态变更的操作必须使用事务，禁止部分提交，并发操作必须考虑锁机制。技术治理：禁止引入非必要的第三方依赖，引入新依赖必须说明选型理由并记录在案。接口设计：遵循RESTful规范，统一错误响应格式，所有端点必须有接口文档，接口契约必须与前端共享，确保前后端一致。

CodeBuddy完成宪法文件的内容编写后，会向我们确认是否创建宪法文件并保存内容。宪法文件创建确认界面如图6-3所示。

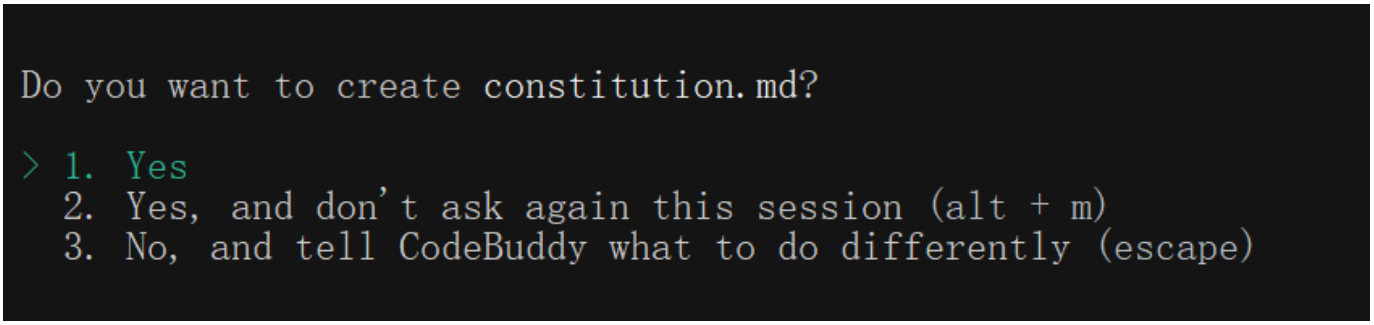


图6-3 宪法文件创建确认界面

注意，当宪法文件创建成功之后，系统会询问是否进行下一步操作。宪法创建完成后的操作询问界面如图6-4所示。

此时我们需要主动告知系统停止当前操作，避免执行超出预期的任务。在CodeBuddy交互界面中输入以下命令：

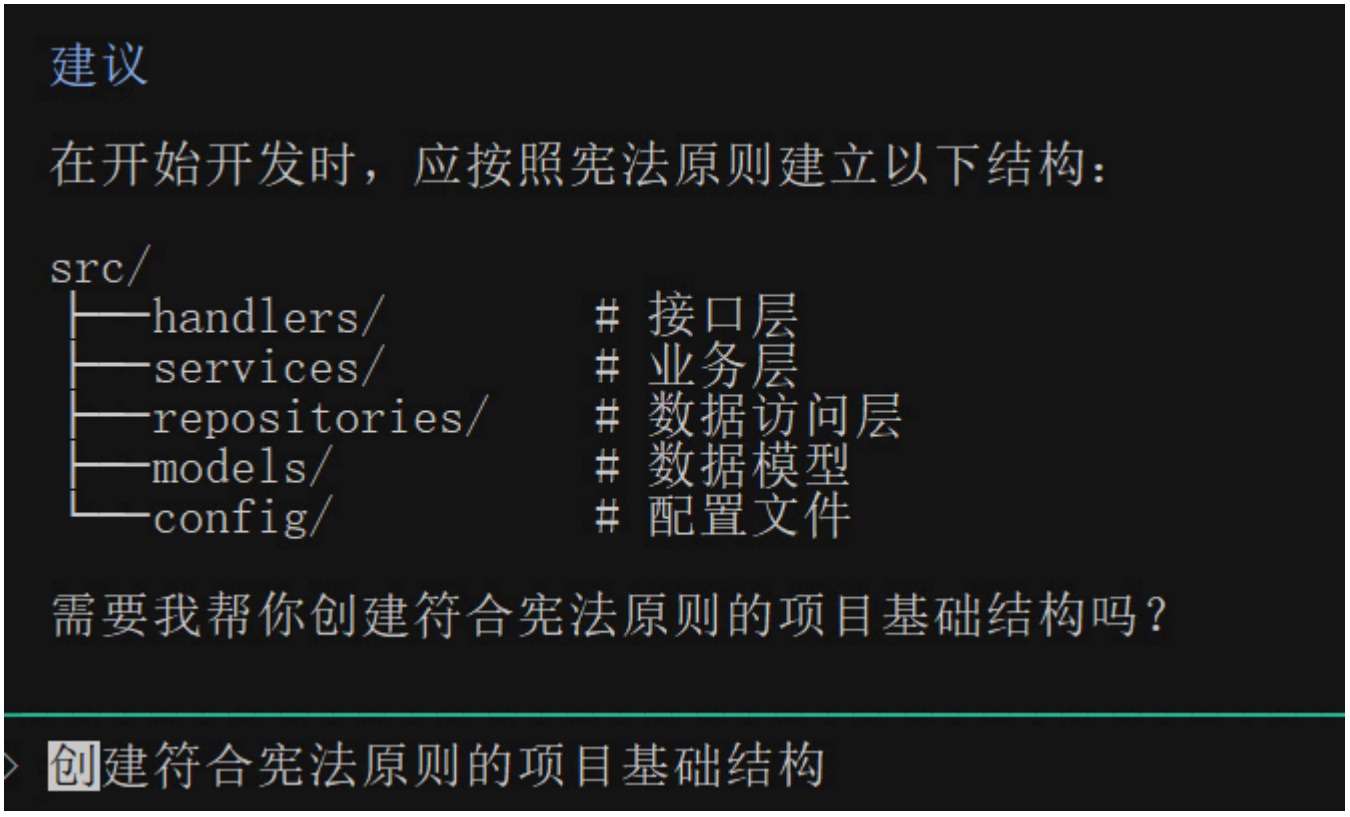


图6-4 宪法创建完成后的操作询问界面

停止。不要创建任何项目结构。技术栈将在/speckit.plan阶段确定，届时再创建对应的目录结构。请先完成宪法修正，然后等待我的命令。

6.2 定义后端API契约

在6.1节中，我们确立了后端API服务项目宪法。本节将使用 `/speckit.specify` 定义后端API服务项目的接口规范——这份规范同时服务于后端和前端项目，是两者之间的契约。

6.2.1 后端API契约：核心作用与价值解析

在真实的企业级项目开发中，要实现前后端团队的高效协作，核心前提是清晰定义并确认API契约。这份契约如同双方共同遵守的技术协议，明确规定了接口的请求参数、响应格式、错误码规则等关键信息，确保前后端开发人员对接口行为形成统一认知，避免理解偏差导致的返工与沟通成本。API契约的作用如图6-5所示。

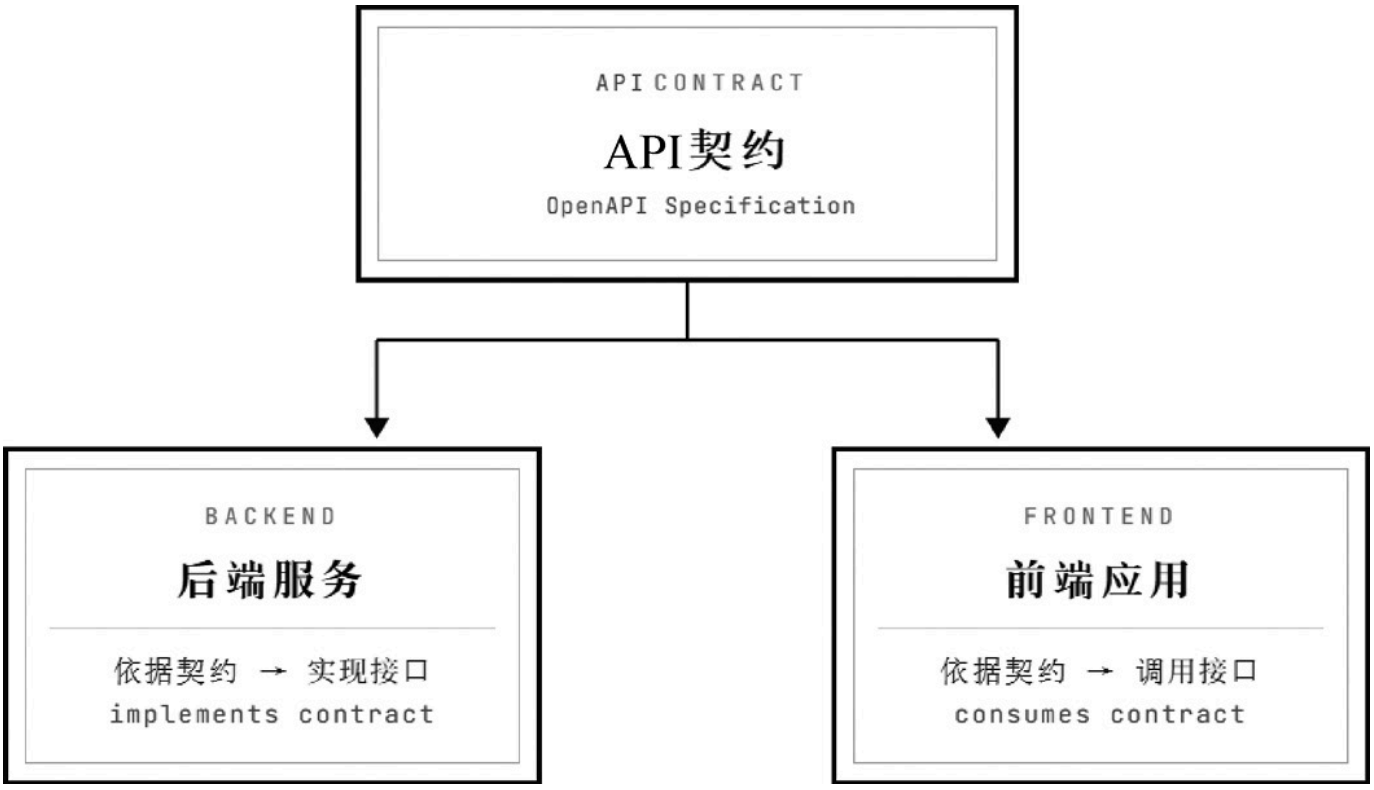


图6-5 API契约的作用

一份完整的API契约通常包含以下核心维度，如表6-1所示。

表6-1 API契约核心维度说明

维度	说明	示例
资源定义	明确系统中包含的核心业务实体及其含义	任务（Task）、用户（User）等

接口列表	定义每个资源对外暴露的操作方法与访问路径	GET/api/tasks（获取任务清单）、POST/api/tasks（创建任务）等
数据模型	详细描述每个资源包含的字段名称、类型及约束	id（字符串，唯一标识）、title（字符串，任务标题）、status（枚举，任务状态）、createdAt（时间戳，创建时间）
请求格式	规定调用接口时需要传递的参数结构、数据类型及必填要求	{"title":"学习React", "status": "todo"}
响应格式	定义接口正常返回时的数据结构、字段含义及嵌套关系	{"id":"1","title":"学习React","status": "todo", "createdAt":"2024-01-01T00:00:00Z"}
错误格式	统一接口异常时的错误响应格式、错误码规则及错误信息描述规范	遵循RFC 7807标准返回错误信息



提示

API契约的核心价值在于定义系统的外部交互边界，因此契约中应严格避免包含任何技术实现细节（如使用的开发框架、数据库类型、中间件选型等）。它只关注“接口长什么样”，即外部系统如何与本系统进行交互，而不涉及“接口怎么实现”这类内部技术决策。这种关注点分离的策略使契约能够独立于具体实现技术，为前后端开发协作、多语言系统集成提供稳定的交互基础。

6.2.2 对齐契约与前端数据模型

在第5章中，我们已完成了前端的数据模型定义工作。为确保前后端数据交互的一致性与准确性，后端API契约必须与前端数据模型严格对齐，前端数据模型与API的契约对应关系如表6-2所示。

表6-2 前端数据模型与API契约的对应关系

前端数据模型（`src/types/task.ts`）	后端API契约
`Task` 接口	任务资源的数据模型
TaskStatus枚举（todo, in-progress, review, done）	status字段的取值范围
loadTasks()→Task[]	GET /api/tasks→Task[]
saveTasks(tasks)	POST /api/tasks或PUT /api/tasks/{id}

6.2.3 输入API契约描述

为了确保前后端数据交互的一致性与准确性，我们需要基于前端已定义的数据模型，编写后端API契约，明确接口的请求参数、响应格式、错误码规则等关键信息。具体操作如下。

进入CodeBuddy交互界面后，输入以下内容：

```
/speckit.specify定义任务看板应用的后端API契约。资源：任务（Task）。任务包含以下字段：id（字符串，唯一标识）、title（字符串，任务标题，最多50个字符）、status（枚举：todo，in-progress，review，done，任务状态）、createdAt（时间戳，创建时间）。接口包括：GET /api/tasks获取所有任务列表，支持通过keyword查询参数进行关键词搜索（模糊匹配标题），支持通过status查询参数并按状态筛选（取值为todo/in-progress/review/done）；POST/api/tasks创建新任务，请求体包含title字段，status默认为todo，createdAt由服务端生成；PUT/api/tasks/{id}更新任务，可更新title和status字段；DELETE/api/tasks/{id}删除任务。所有接口返回JSON格式的内容，成功响应返回对应数据，错误响应使用RFC 9457标准中的格式。API契约必须与前端共享，与前端数据模型对齐。所有项目文档使用中文编写。
```

CodeBuddy接收命令后，将自动基于上述描述生成标准化的API规范文档，涵盖接口定义、参数约束、响应格式等核心内容。API规范文档生成过程展示如图6-6所示。

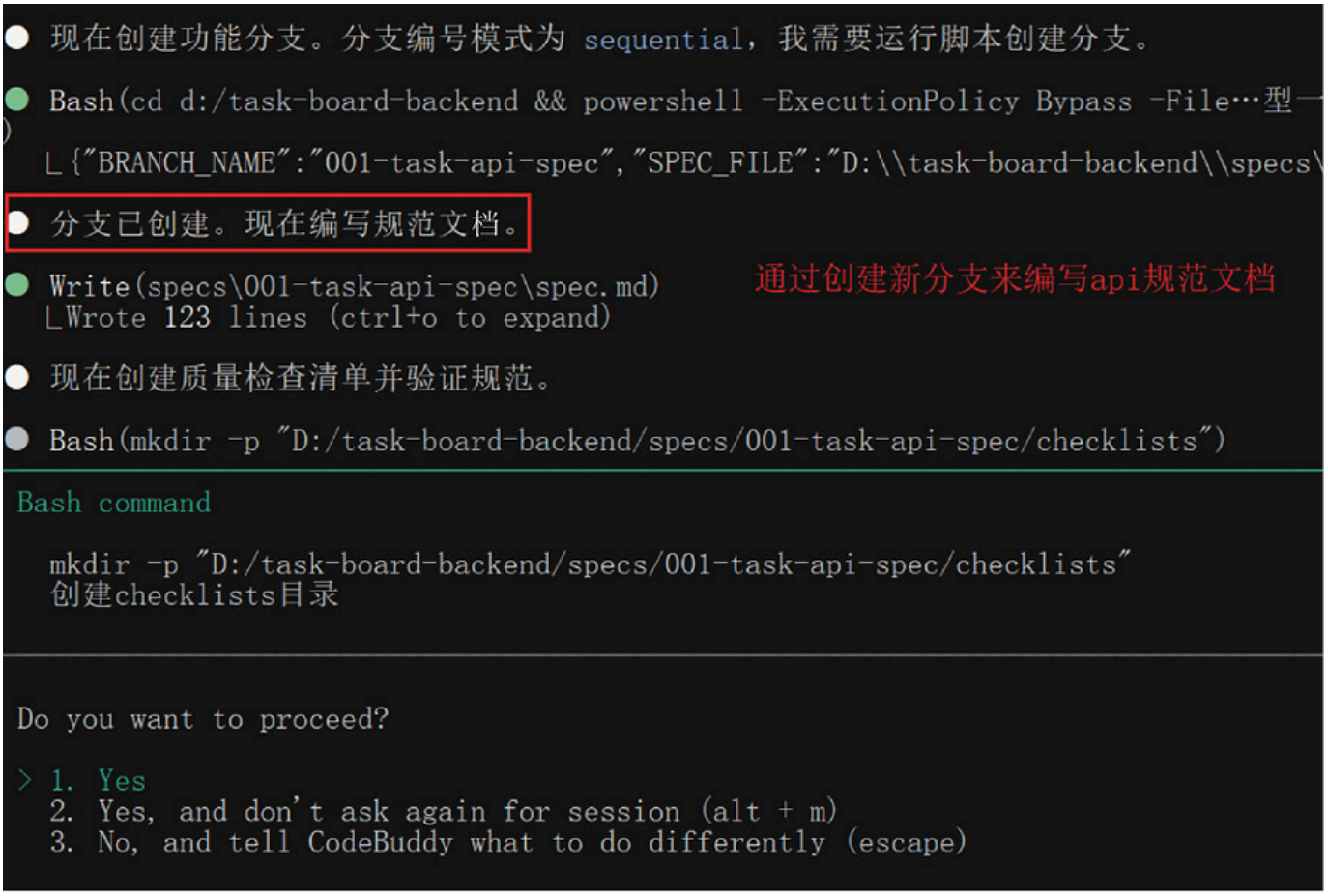


图6-6 API规范文档生成过程展示

当我们看到系统输出规范文档生成完成的提示信息，且确认所有接口定义、参数约束、响应格式等核心内容均已正确生成时，则可以结束当前的API契约定义环节。API契约定义完成提示界面如图6-7所示。

```
完成报告

功能分支: 001-task-api-spec
规范文件: specs/001-task-api-spec/spec.md
检查清单: specs/001-task-api-spec/checklists/requirements.md

规范内容摘要

定义了任务看板的4个核心API接口:
- GET /api/tasks - 获取任务清单，支持关键词搜索
- POST /api/tasks - 创建新任务
- PUT /api/tasks/{id} - 更新任务
- DELETE /api/tasks/{id} - 删除任务

质量验证结果

所有检查项均已通过:
- 无实现细节泄露
- 无待澄清标记
- 需求可测试且明确
- 成功标准可衡量
- 用户场景覆盖完整

下一步

规范已准备就绪，可以执行:
- /speckit.clarify - 如需进一步澄清需求细节
- /speckit.plan - 进入技术规划阶段
```

图6-7 API契约定义完成提示界面

6.2.4 审视契约：前后端一致性检查

重要提示：禁止AI自动执行后续步骤。

契约生成完成后，必须从前后端数据一致性的核心视角进行全面审视，确保接口契约与前端实现逻辑完全对齐。

为确保契约的可靠性，我们可通过以下步骤进行验证。

（1）审查规范文档完整性。定位到后端项目规范文档 `specs\001-task-board-api\spec.md` ，重点检查接口定义、参数约束、响应格式是否完整覆盖业务需求，确保没有遗漏关键业务场景。

（2）前端数据模型与后端契约定义的字段级对齐校验。打开第5章前端项目中的 `src/types/task.ts` 文件，逐字段对比前端数据模型与后端契约定义，确保数据结构完全一致。前端数据模型与后端契约定义的字段级对齐校验如表6-3所示。

表6-3 前端数据模型与后端契约定义的字段级对齐校验

字段名称	前端TypeScript定义	后端API契约定义	校验结果
id	string（唯一标识符）	string（全局唯一ID）	待确认
title	string（任务标题）	string（最多50字符，任务标题）	待确认（需前端添加长度校验）
status	'todo'|'in-progress'| 'review'|'done'（枚举类型）	todo/in-progress/review/done（字符串枚举）	待确认（需统一枚举值格式）
createdAt	number（时间戳，毫秒级）	时间戳（字符串格式）	待确认（需统一时间戳格式）

（3）前端操作场景的接口覆盖度验证。接下来，我们将从前端操作场景的角度，验证后端API是否能够全面支撑前端业务需求，确保前后端交互的顺畅性与一致性。前端操作场景与后端API对应情况如表6-4所示。

表6-4 前端操作场景与后端API对应情况

前端操作场景	对应后端API	接口覆盖状态
页面初始化加载所有任务数据	`GET /api/tasks`	待确认（需验证筛选参数支持）
用户创建新任务	`POST /api/tasks`	待确认（需验证请求参数约束）
拖拽任务，更新状态（如将任务太态从“待处理”更新为“进行中”等）	`PUT /api/tasks/{id}`	待确认（需验证状态字段枚举值一致性）
编辑任务标题	`PUT /api/tasks/{id}`	待确认（需验证部分更新支持）
关键词搜索任务	`GET /api/tasks?keyword=xxx`	待确认（需验证搜索算法一致性）

（4）检查契约是否存在技术实现细节泄露。API契约应专注于对外交互规范，严禁包含任何后端技术实现细节。重点检查是否存在以下内容：

- Spring Boot、Spring MVC等特定框架技术选型。
- JPA、MyBatis等ORM（Object-Relational Mapping，对象-关系映射）框架实现细节。
- PostgreSQL、MySQL等数据库类型及配置信息。
- Controller、Service、Repository等后端代码分层结构。

若发现上述技术实现细节，必须立即从契约中删除，确保契约的纯粹性和通用性。

所有校验完成后，输出一致性校验报告，明确记录待确认项和需要修改的内容。

6.2.5 确认契约并停止操作

审视完毕，确认API契约与前端数据模型一致且无遗漏后，告知CodeBuddy停止，输入以下信息：

```
API契约已确认，请停止当前操作。我将在准备好后使用/speckit.plan制定后端技术方案。
```

6.3 规划技术方案

在6.2节中，我们定义了后端API契约。本节将使用 `/speckit.plan` 确定后端的技术方案。

进入CodeBuddy交互界面后，输入如下信息：

```
/speckit.plan使用Spring Boot 3.2、Maven、Spring Data JPA、MySQL 8.0开发。接口层使用@RestController，业务层使用@Service，数据访问层使用JpaRepository。测试使用@SpringBootTest和TestRestTemplate进行集成测试，使用Testcontainers提供MySQL测试环境。API文档使用SpringDoc OpenAPI自动生成。所有项目文档使用中文编写。
```

当CodeBuddy接收到命令后，会自动读取项目中已有的宪法文件和规范文档，并结合你提供的技术方案描述，生成一套完整的项目计划。

在此阶段，系统将自动生成标准化的API契约，为后续开发提供清晰依据。`openapi.yaml` 是正式的API契约文档，以OpenAPI 3.0格式定义所有接口的请求/响应结构。前端后续可以基于这个文件生成API客户端代码。API契约文档创建确认交互界面如图6-8所示。

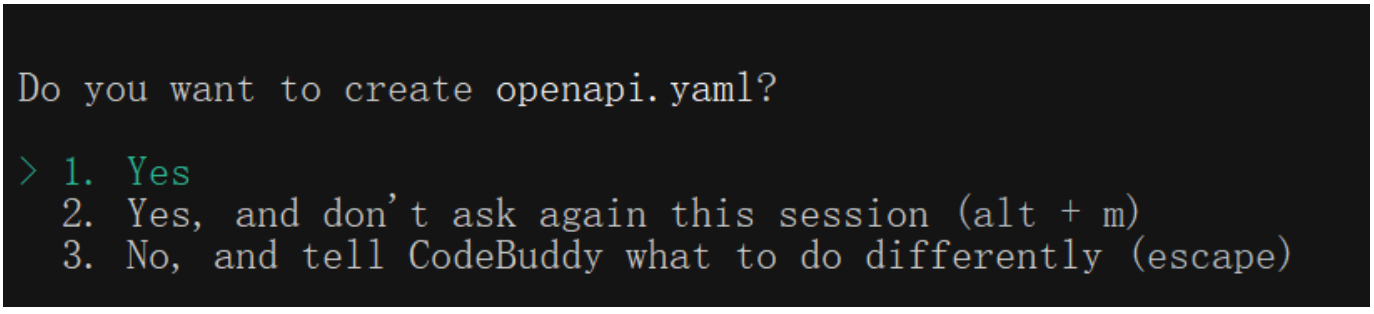


图6-8 API契约文档创建确认交互界面

最终，我们成功生成了一套完整的计划文件，涵盖了项目开发所需的各类核心文件。这些文件将为后续的任务拆解与实现提供清晰的指导。后端API计划文件汇总如图6-9所示。

至此，后端API的技术方案规划已全部完成，我们可以进入6.4节的任务拆解与实现环节。

文件	说明
plan.md	实现计划，包含技术上下文和宪法检查
research.md	技术研究，包含选型决策和最佳实践
data-model.md	数据模型设计，包含实体定义和数据库结构
contracts/openapi.yaml	OpenAPI 3.0接口契约
quickstart.md	快速启动指南

技术栈确认

- 框架：Spring Boot 3.2 + Maven
- 数据访问：Spring Data JPA
- 数据库：MySQL 8.0
- 测试：SpringBootTest + Testcontainers
- 文档：SpringDoc OpenAPI

宪法检查

所有检查项已通过，无违规。

下一步

执行 /speckit.tasks 生成开发任务列表。

图6-9 后端API计划文件汇总

6.4 拆解与实现任务

前三节中，我们依次完成了后端宪法、API契约、技术方案。本节将使用 `/speckit.tasks` 拆解任务，再使用 `/speckit.implement` 执行实现，最终完成后端API的开发。

6.4.1 拆解任务

进入CodeBuddy交互界面后，输入：

```
/speckit.tasks
```

CodeBuddy会自动读取已有的规范和计划，生成任务清单文件 `tasks.md` ，并让我们确认。任务清单生成确认界面如图6-10所示。

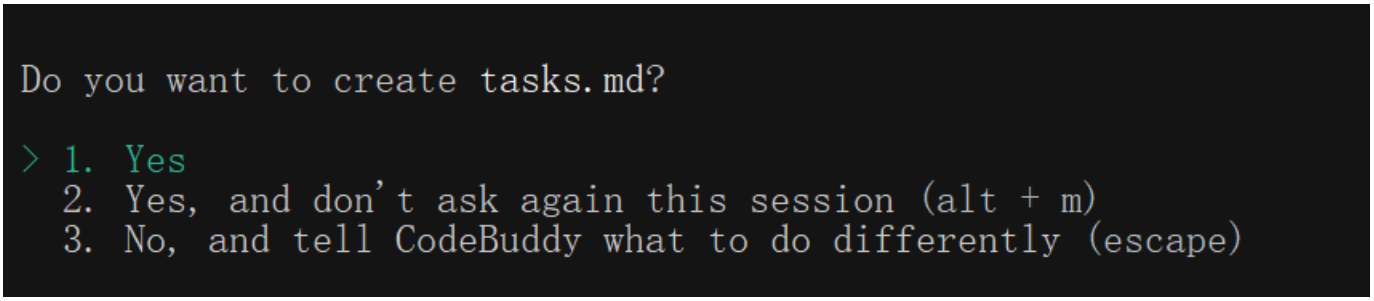


图6-10 任务清单生成确认界面

选择“Yes”后，CodeBuddy会自动拆解任务，并填写到tasks.md文件中。当我们看到如图6-11所示的结果时，就可以结束本次操作了。

并行机会

- Phase 1: 4个任务可并行（T002~T005）
- Phase 2: 2个任务可并行（T011~T012）
- 用户故事并行: Foundational完成后，所有用户故事可同时开发

独立测试标准

用户故事	独立测试方式
US1	GET /api/tasks 返回任务列表
US2	POST /api/tasks 创建任务返回201
US3	PUT /api/tasks/{id} 更新任务
US4	DELETE /api/tasks/{id} 删除任务返回204

MVP建议范围

完成 Phase 1 + Phase 2 + Phase 3（用户故事1）即可交付最小可用产品。

格式验证

所有48个任务均遵循严格的检查清单格式：- [ ] [TaskID] [P?] [Story?] 描述（含文件路径）

> 开始执行任务列表

图6-11 任务清单生成结果

最终，我们在CodeBuddy交互界面中输入以下内容（输入暂停命令界面如图6-12所示）：

```
请先暂停。我需要审视任务清单后再决定是否执行。
```

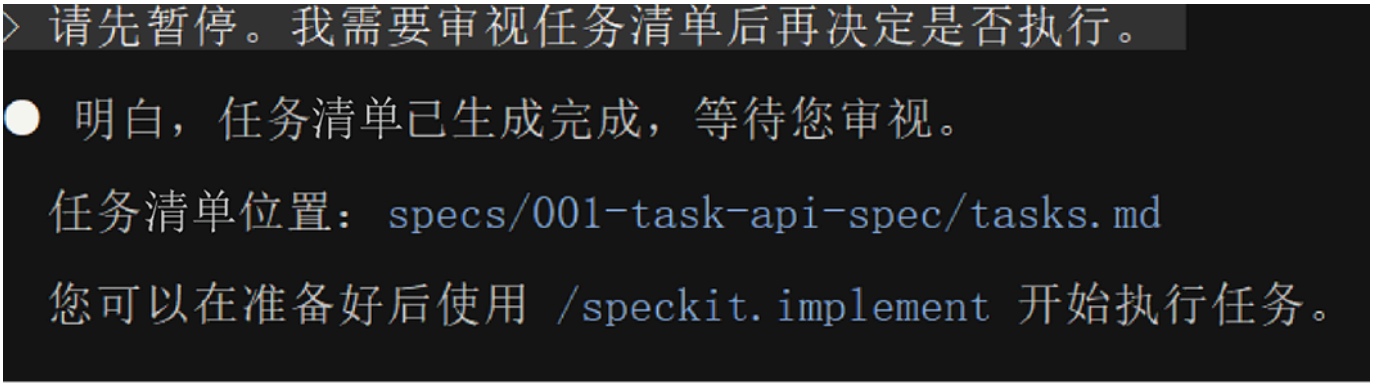


图6-12 输入暂停命令界面

重要提示

任务清单生成后，CodeBuddy可能会询问是否直接执行实现。不要一路选择“YES”。

强烈建议审查生成的任务清单是否符合要求，如果发现遗漏，应在CodeBuddy交互界面中补充。

6.4.2 执行实现

打开终端并切换至项目根目录，执行启动命令进入CodeBuddy交互界面，在命令行提示符后输入以下命令：

```
/speckit.implement
```

CodeBuddy接收命令后，会读取 `tasks.md` ，按任务顺序逐项执行。执行过程中，你可能会看到类似以下内容的输出：

```
•Write(pom.xml)
  ☐ Wrote 68 lines

  •Write(src/main/java/com/taskboard/model/Task.java)
  ☐ Wrote 42 lines

  •Write(src/main/resources/db/migration/V1__init.sql)
  ☐ Wrote 18 lines

   •Write(src/main/java/com/taskboard/repository/TaskRepository.java)
   ☐ Wrote 12 lines

   •Write(src/main/java/com/taskboard/service/TaskService.java)
   ☐ Wrote 86 lines

   •Write(src/main/java/com/taskboard/controller/TaskController.java)
   ☐ Wrote 72 lines

   •Write(src/main/java/com/taskboard/exception/GlobalExceptionHandler.java)
   ☐ Wrote 48 lines

   •Write(src/test/java/com/taskboard/controller/TaskControllerTest.java)
   ☐ Wrote 96 lines

   •Bash(mvn clean test)
   ☐ [INFO]BUILD SUCCESS
   Tests run:12,Failures:0,Errors:0
```

6.4.3 验证后端API服务

为确保后端API服务功能的正确性与稳定性，我们将通过以下系统化验证流程进行全面测试。

输入如下命令，在系统终端中启动后端服务：

```
mvn spring-boot:run
```

服务启动后，打开浏览器，输入以下地址访问Swagger UI：http://localhost:8080/swagger-ui/index.html。在Swagger UI中，可直接对所有后端API进行在线测试，包括调用接口、调试参数、查看响应结果等。Swagger UI API测试界面如图6-13所示。

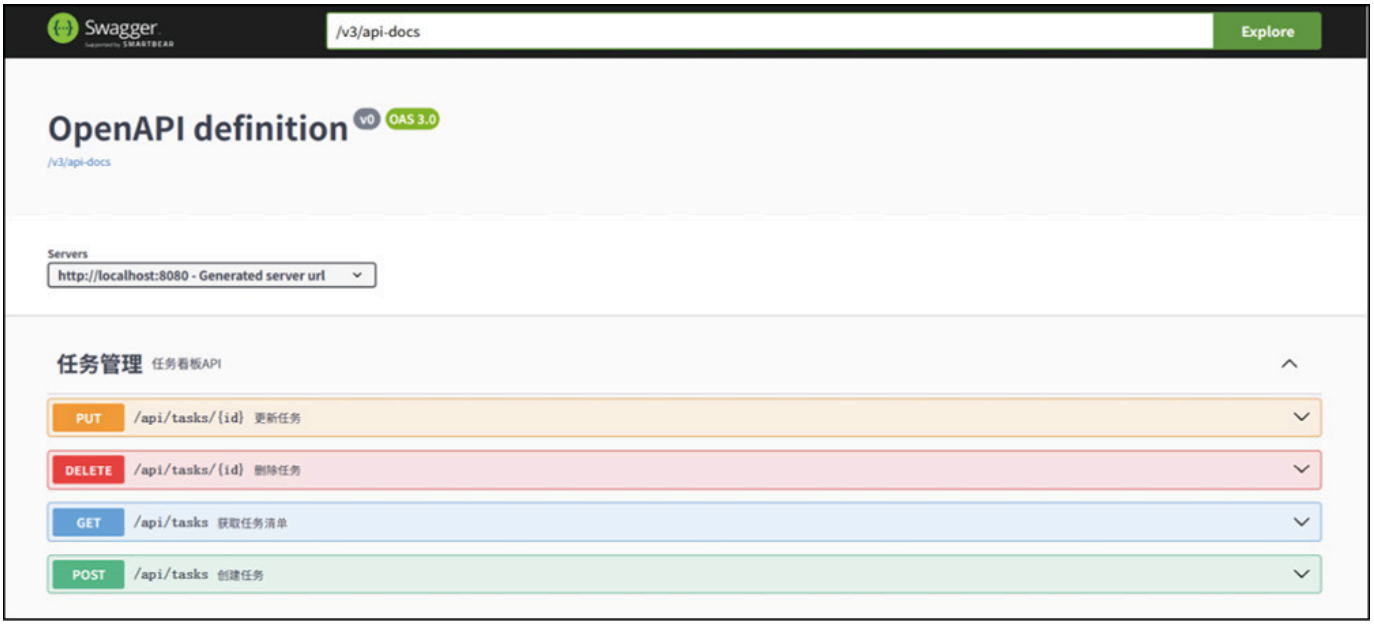


图6-13 Swagger UI API测试界面

以获取任务清单为例，调用 `GET /api/tasks` 接口后，系统将返回规范的JSON格式的任务清单，包含任务ID、标题、状态、创建时间等字段的完整信息。GET /api/tasks接口返回的JSON数据示例如图6-14所示。

Curl

```
curl -X 'GET' \
  'http://localhost:8080/api/tasks' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:8080/api/tasks
```

Server response

Code	Details
200	Response body


```
[
  {
    "id": "3c87305a-3181-11f1-bc22-e86a64f4fa78",
    "title": "完成用户故事梳理和需求分析",
    "status": "DONE",
    "createdAt": "2026-04-06T22:24:16"
  },
  {
    "id": "3c873517-3181-11f1-bc22-e86a64f4fa78",
    "title": "设计数据库表结构和ER图",
    "status": "DONE",
    "createdAt": "2026-04-05T22:24:16"
  },
  {
    "id": "3c873739-3181-11f1-bc22-e86a64f4fa78",
    "title": "编写API契约文档", ,
    "status": "DONE",
    "createdAt": "2026-04-05T22:24:16"
  },
  {
    "id": "3c8738bc-3181-11f1-bc22-e86a64f4fa78",
    "title": "搭建Spring Boot项目基础架构",
    "status": "DONE",
    "createdAt": "2026-04-04T22:24:16"
  },
  {
    "id": "3c873a09-3181-11f1-bc22-e86a64f4fa78",
```


图6-14 GET /api/tasks接口返回的JSON数据示例

如果在验证过程中发现异常情况，请将详细的错误信息、复现步骤以及相关上下文信息完整复制给CodeBuddy，以便其精准定位问题并提供解决方案，例如输入以下内容：

```
问题：创建任务时，createdAt字段返回null。
  操作步骤：①启动服务。②输入命令POST /api/tasks{"title":"测试"}。③查看响应，createdAt字段返回null。
  请分析原因并修复。
```

CodeBuddy会分析错误、定位问题，并生成补丁。

6.4.4 验证通过并终止操作

完成所有功能的验证工作后，请在CodeBuddy交互窗口中输入以下命令，告知其停止当前操作：

```
所有功能已验证通过，请停止当前操作。
```

6.5 前后端联调：替换前端数据层

6.1 ~ 6.4节，我们完成了后端的全部开发工作——宪法确立、API契约定义、技术方案规划、任务拆解与实现。本节将完成最后一步——前后端联调：将前端的 `storage.ts`（localStorage）替换为 `api.ts`（HTTP请求），让任务看板成为一个真正由后端驱动的完整应用。

6.5.1 理解数据层替换

查看第5章代码，可知前端代码架构设计如下：

```
组件（TaskBoard、TaskColumn、TaskItem）
  ↓调用
Context（BoardContext.tsx）←状态管理
  ↓调用
utils（storage.ts）←数据层（当前：localStorage）
```

我们要做的事情是：只替换最后一层（将storage.ts替换为api.ts），上层代码不变。这正是第5章设计数据层抽象的价值——将数据访问逻辑隔离，使底层实现可以自由替换。数据层替换详情如表6-5所示。

表6-5 数据层替换详情

维度	替换前（第5章）	替换后（本节）
数据层文件	`src/utils/storage.ts`	`src/utils/api.ts`
数据来源	浏览器localStorage	后端API（localhost:8080）
函数签名	`loadTasks():Task[]` （同步）	`loadTasks():Promise<Task[]>` （异步）
上层代码	BoardContext调用storage.ts	BoardContext调用api.ts

6.5.2 配置前端开发代理

前端运行在 `localhost:5173` ，后端运行在 `localhost:8080` 。浏览器的同源策略会阻止前端直接请求后端。解决方案是配置前端开发代理——让Vite将 `/api` 开头的请求转发到后端。

在系统终端中（确保当前目录为前端项目 `task-board/` ）输入如下命令，启动CodeBuddy：

```
codebuddy
```

在CodeBuddy交互界面中输入如下信息：

```
请在vite.config.ts中添加开发代理配置：将所有/api开头的请求转发到http://localhost:8080。
```

CodeBuddy会自动更新项目根目录下的 `vite.config.ts` 配置文件，添加开发代理规则，将前端发起的以 `/api` 开头的请求转发至后端服务地址http://localhost:8080，从而解决浏览器同源策略限制导致的跨域问题。Vite开发代理配置示例如图6-15所示。

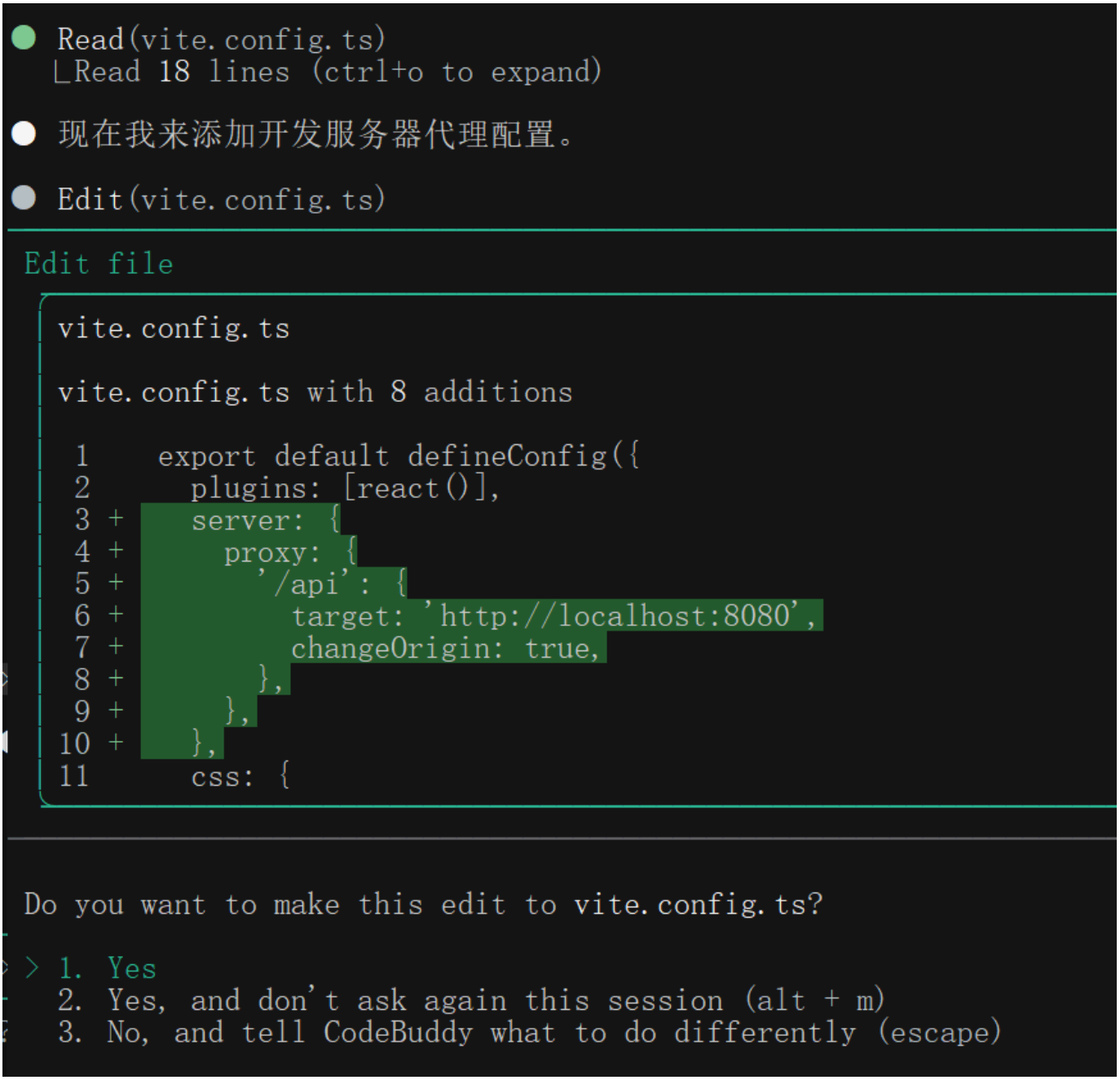


图6-15 Vite开发代理配置示例

6.5.3 阅读API契约——了解后端接口

在编写 `api.ts` 之前，前端开发者需要知道后端提供了哪些接口。这些信息来自6.2节定义的API契约。在CodeBuddy交互界面中让它读取后端API契约，输入如下信息：

```
请读取D:\task-board-backend\specs\001-task-api-spec\contracts\openapi.yaml文件，了解后端提供的所有接口。
```

下达命令后，CodeBuddy会自动读取后端API契约文档，并精准解析其中定义的接口规则、数据结构与交互逻辑，完成对契约的深度理解。CodeBuddy读取并解析后端API契约文档的示例如图6-16所示。

● Read(..\task-board-backend\specs\001-task-api-spec\contracts\openapi.yaml)
└Read 278 lines (ctrl+o to expand)

● 已读取更新后的 API 规范。主要变化：

更新内容

GET /api/tasks 新增参数：
- status (query, 可选)：按状态筛选任务
  - 枚举值：TODO | IN_PROGRESS | REVIEW | DONE
  - 可单独使用或与 keyword 组合使用

状态枚举值变化：
- 旧：小写 todo | in-progress | review | done
- 新：大写下划线 TODO | IN_PROGRESS | REVIEW | DONE

完整 API 列表

方法	路径	说明
GET	/api/tasks?keyword=xxx&status=xxx	获取任务清单（支持搜索+筛选）
POST	/api/tasks	创建新任务
PUT	/api/tasks/{id}	更新指定任务
DELETE	/api/tasks/{id}	删除指定任务

TaskResponse: id, title, status, createdAt

状态值：TODO | IN_PROGRESS | REVIEW | DONE

图6-16 CodeBuddy读取并解析后端API契约文档的示例

6.5.4 创建api.ts替换storage.ts

阅读完契约后，可以指导CodeBuddy生成 `api.ts` 。在CodeBuddy交互界面中输入如下信息：

> 根据后端API契约（GET /api/tasks、POST /api/tasks、PUT /api/tasks/{id}、DELETE /api/tasks/{id}，支持keyword和status查询参数），创建src/utils/api.ts文件，替换现有的storage.ts。需要实现以下函数：loadTasks()，获取所有任务；loadTasksByStatus(status)，按状态筛选；loadTasksByKeyword(keyword)，按关键词搜索；loadTasksFiltered(status,keyword)，组合筛选；createTask(title)，创建任务；updateTask(id,updates)，更新任务；deleteTask(id)，删除任务。所有函数返回Promise。请求头设置Content-Type:application/json。同时删除storage.ts。

接下来，CodeBuddy将根据后端API契约自动生成符合规范的 `api.ts` 文件，该文件包含所有与后端交互的接口方法实现。CodeBuddy自动生成 `api.ts` 的过程示例如图6-17所示。

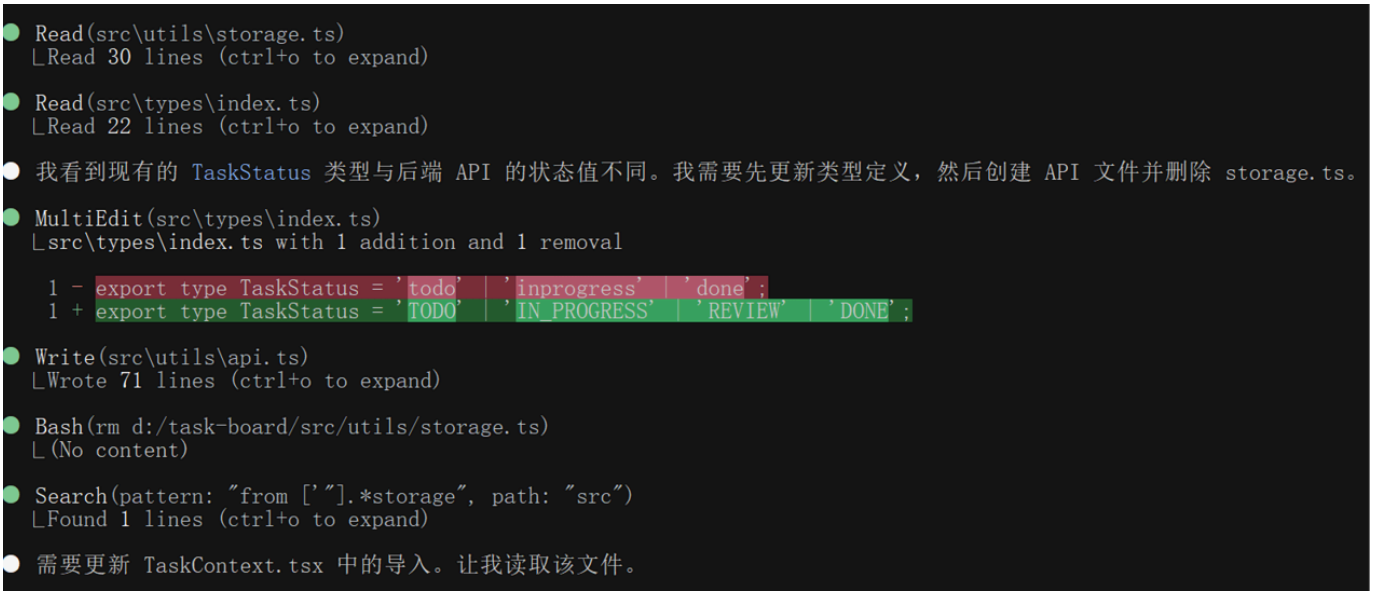


图6-17 CodeBuddy自动生成api.ts的过程示例

CodeBuddy会检测并修改相关的文件，包括页面文件等，全程自动化执行。最终，CodeBuddy会汇总本次修改的文件列表，如图6-18所示。



图6-18 本次修改的文件列表

6.5.5 适配BoardContext的异步调用

`storage.ts` 采用同步数据访问模式，而 `api.ts` 基于异步HTTP请求实现。这种底层数据访问方式的差异，要求我们对 `BoardContext` 进行适配改造。

请在CodeBuddy交互界面中输入以下命令：

> 请修改BoardContext.tsx，将所有对storage.ts的调用替换为对api.ts的异步调用。loadTasks在组件挂载时调用，createTask、updateTask、deleteTask在用户操作时调用。需要处理加载状态和错误状态。同时将搜索功能从前端过滤改为调用后端loadTasksFiltered接口。

最终，Codebuddy会自动完成单元测试，并提供总结报告，如图6-19所示。

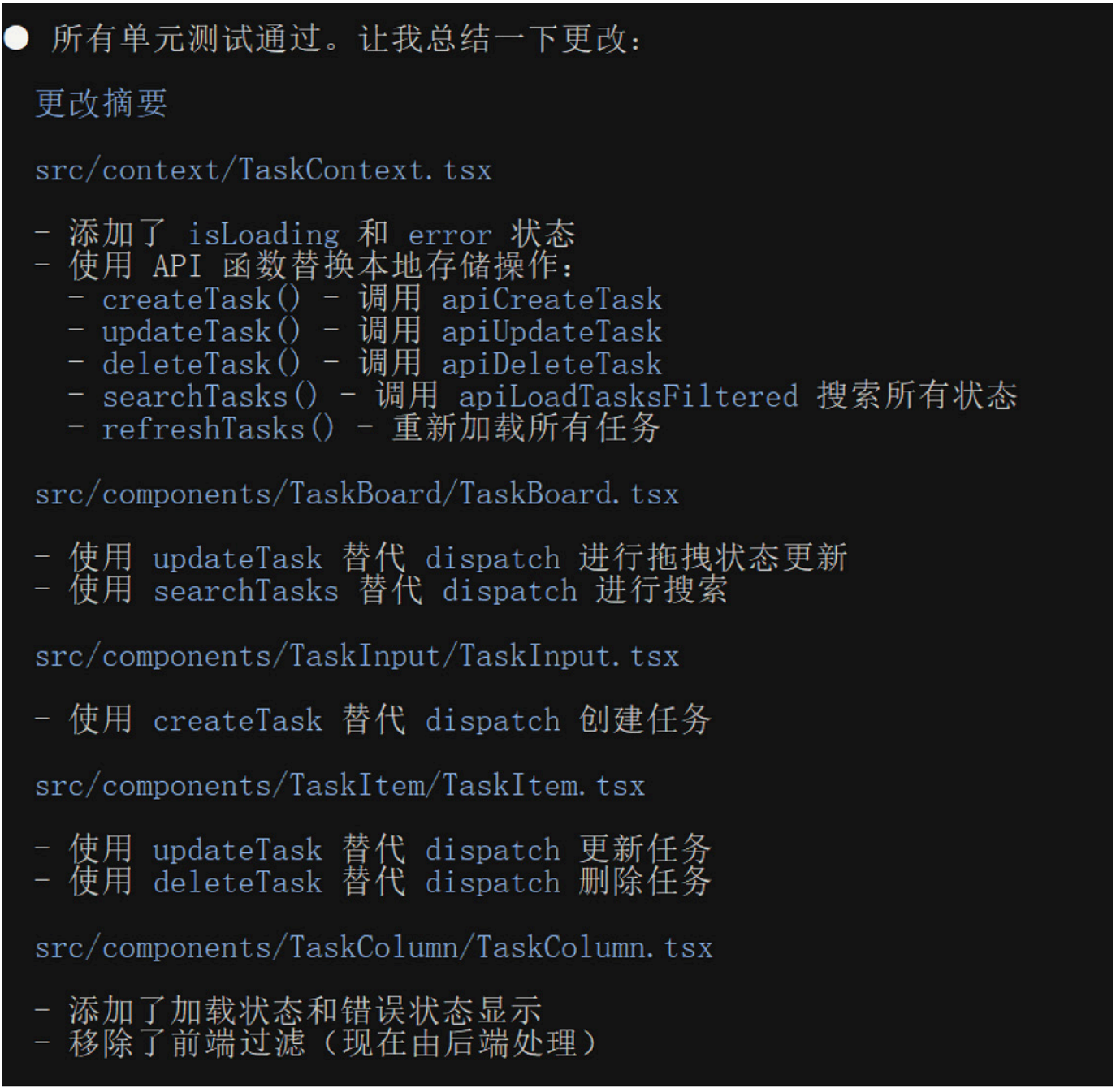

● 所有单元测试通过。让我总结一下更改：
更改摘要
src/context/TaskContext.tsx
- 添加了 isLoading 和 error 状态
- 使用 API 函数替换本地存储操作：
- createTask() - 调用 apiCreateTask
- updateTask() - 调用 apiUpdateTask
- deleteTask() - 调用 apiDeleteTask
- searchTasks() - 调用 apiLoadTasksFiltered 搜索所有状态
- refreshTasks() - 重新加载所有任务
src/components/TaskBoard/TaskBoard.tsx
- 使用 updateTask 替代 dispatch 进行拖拽状态更新
- 使用 searchTasks 替代 dispatch 进行搜索
src/components/TaskInput/TaskInput.tsx
- 使用 createTask 替代 dispatch 创建任务
src/components/TaskItem/TaskItem.tsx
- 使用 updateTask 替代 dispatch 更新任务
- 使用 deleteTask 替代 dispatch 删除任务
src/components/TaskColumn/TaskColumn.tsx
- 添加了加载状态和错误状态显示
- 移除了前端过滤（现在由后端处理）

图6-19 总结报告

6.5.6 前后端联调验证

接下来，我们将通过以下完整步骤完成前后端联调的验证工作。

第一步：启动后端API服务

请打开系统终端，切换到后端API服务项目所在的目录，然后执行以下命令，启动后端API服务：

```
cd D:\task-board-backend
  mvn spring-boot:run
```

等待后端API服务启动完成，终端输出类似“Started TaskBoardApplication in X seconds(process running)”的日志信息，即表示后端服务已成功启动。

第二步：启动前端任务看板项目

请保持后端API服务运行状态不变，另外打开一个系统终端，执行以下命令，启动前端任务看板项目：

```
cd D:\task-board
  npm run dev
```

第三步：在浏览器中验证前后端联调是否成功

在浏览器中访问前端地址http://localhost:5173，逐项验证各功能是否正常运行。此时，任务看板中展示的数据已成功切换为来自后端API服务的数据。前后端联调成功后的数据展示效果如图6-20所示。

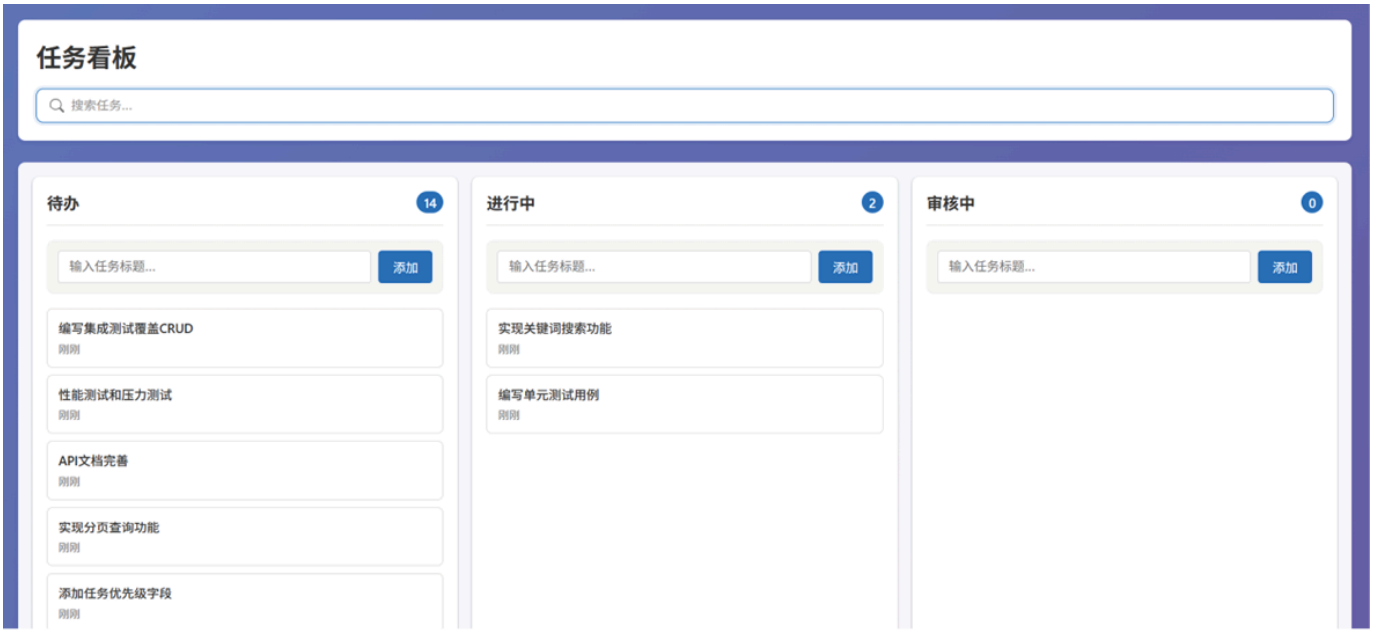


图6-20 前后端联调成功后的数据展示效果

第7章　【进阶】安全加固与前后端联调

在实际的生产环境中，系统的安全性不可忽视。本章将聚焦于系统的安全加固，通过引入认证与权限控制机制，将当前“任何人都可操作”的开放系统，升级为具备用户身份认证、权限细分管理的安全可控系统。本次安全加固工作将同时涉及前端与后端，我们将逐步完成从需求分析到功能实现的完整流程，确保系统在安全层面的可靠性与稳定性。

7.1 更新宪法：增加安全原则

本节我们将围绕安全加固的核心需求，依次完成前后端项目宪法的安全原则更新，为后续的认证与权限功能开发筑牢基础。

安全加固涉及项目及修改内容如表7-1所示。

表7-1 安全加固涉及项目及修改内容

项目	修改内容
后端（task-board-backend）	新增认证模块、修改现有接口（增加鉴权）
前端（task-board）	新增登录页面、请求头携带Token

7.1.1 更新后端API服务项目宪法

后端API服务项目是本次安全加固的核心阵地。请在系统终端中切换至后端项目目录，启动CodeBuddy并进入交互界面，然后输入以下内容：

```
/speckit.constitution更新后端项目宪法，在现有原则基础上新增安全原则。认证：所有接口必须经过身份认证，未认证请求返回401。密码：用户密码必须使用不可逆哈希算法存储，禁止明文存储。会话：使用无状态Token机制管理用户会话，Token必须有过期时间。输入校验：所有用户输入必须在接口层进行校验，禁止信任客户端数据。错误信息：认证失败的错误响应不得泄露用户是否存在等敏感信息。
```

CodeBuddy完成宪法更新后，将在原有基础上新增相关安全原则。后端项目宪法新增安全原则示例如图7-1所示。

```
● Search(pattern: ".specify/templates/commands/*.md")
  └Found 0 files

● 现在我已了解现有宪法和模板结构。我将更新宪法文件，添加安全原则作为第七个核心原则，

● Write(.specify\memory\constitution.md)
  └Wrote 130 lines (ctrl+o to expand)

● 宪法文件已更新完成。

  更新摘要

  版本变更：1.0.0 → 1.1.0（MINOR版本增加，新增原则）

  新增内容：第七个核心原则 - 安全原则，包含：
  - 认证：所有接口必须经过身份认证，未认证返回401
  - 密码存储：必须使用不可逆哈希算法（bcrypt/Argon2/PBKDF2），禁止明文
  - 会话管理：使用无状态Token机制，必须设置过期时间
  - 输入校验：所有用户输入必须在接口层校验
  - 错误信息：认证失败不得泄露用户是否存在等敏感信息

  模板检查结果：
  - ☐ plan-template.md - Constitution Check已正确引用宪法文件
  - ☐ spec-template.md - 安全需求可在FR部分捕获
  - ☐ tasks-template.md - 安全任务可在Foundational阶段添加

  建议提交信息：
  docs: amend constitution to v1.1.0 (add security principle)
```

图7-1 后端项目宪法新增安全原则示例

7.1.2 更新前端任务看板项目宪法

完成后端API服务项目的宪法更新后，接下来我们需要同步更新前端任务看板项目的宪法。请切换到前端项目所在目录，在系统终端中启动CodeBuddy，进入交互界面后，输入以下内容：

```
/speckit.constitution更新前端项目宪法，在现有原则基础上新增安全原则。Token管理：Token必须存储在安全的位置，禁止存储在URL（Uniform Resource Locator，统一资源定位符）或日志中。Token过期：Token过期后自动跳转到登录页面，禁止继续使用过期Token发起请求。请求安全：所有对后端的请求必须携带Token，Token格式为Authorization请求头。输入校验：表单提交前必须进行客户端校验，但后端校验是最终保障。错误处理：认证失败时显示友好的提示信息，禁止暴露后端错误详情。
```

CodeBuddy完成更新后，前端项目宪法将新增相关安全原则，前端项目宪法新增安全原则示例如图7-2所示。

```
● 宪法更新完成。

---
宪法更新摘要

- 版本变更：1.0.0 → 1.1.0（MINOR - 新增安全原则）

新增内容：
- 核心原则 VII：安全原则
  - Token 管理：禁止存储在 URL/日志中
  - Token 过期：自动跳转登录页
  - 请求安全：Authorization 请求头
  - 输入校验：客户端+后端双重校验
  - 错误处理：友好提示，禁止暴露后端详情

已同步更新的模板：
- ☐ plan-template.md - Constitution Check 新增安全检查项
- ☐ spec-template.md - Non-Functional Requirements 新增 NFR-010~013
- ☐ tasks-template.md - Foundational/Polish 阶段新增安全任务

建议提交信息：
docs: amend constitution to v1.1.0 (add security principle)
```

图7-2 前端项目宪法新增安全原则示例

7.1.3 确认宪法并停止操作

确认两个项目的宪法内容后，需分别在对应项目的CodeBuddy交互界面中执行停止操作。

切换至后端项目的CodeBuddy交互界面，输入以下命令：

```
后端安全原则已确认，请停止当前操作。
```

切换至前端项目的CodeBuddy交互界面，输入以下命令：

```
前端安全原则已确认，请停止当前操作。
```

7.2 扩充规范：认证与权限

在7.1节中，我们已分别为前后端项目补充了安全原则。本节将基于这些原则，进一步完善前后端的规范体系：后端需新增认证接口，并对现有接口添加鉴权校验；前端则需新增登录页面，以支持用户身份验证功能。

7.2.1 理解安全加固对规范的影响

安全加固将对系统接口结构产生关键影响，安全加固前后的系统接口结构对比如表7-2所示。

表7-2 安全加固前后系统接口结构对比

维度	安全加固前	安全加固后
接口数量	4个，用于基础的CRUD（Create，Read，Update，Oelete，新增、读取、更新、删除）操作	7个（保留CRUD接口+新增3个认证接口）
现有接口访问权限	无认证限制，任何人可访问	需身份认证，请求必须携带有效Token
新增接口	无	用户注册、用户登录、获取当前用户信息
错误响应类型	资源未找到、参数校验失败	新增：未认证（401）、权限不足（403）
前端页面体系	仅包含任务看板页面	新增用户登录页面，形成完整认证流程



重要提示

安全加固不应改变现有接口的功能逻辑——`GET /api/tasks` 仍然用于返回任务清单，`POST /api/tasks` 仍然用于创建任务。变化的只是“调用前需要先登录”。

7.2.2 更新后端API服务项目规范

请打开系统终端，切换至后端项目所在目录，启动CodeBuddy并进入交互界面，然后输入以下命令：

```
/speckit.specify更新后端API规范，在现有4个接口基础上新增认证模块。新增接口：POST /api/auth/register，用户注册，请求体包含username（3~20个字符）和password（8~50个字符，必须包含字母和数字），返回创建成功的用户信息（不含密码）；POST /api/auth/login，用户登录，请求体包含username和password，返回Token和用户信息；GET /api/users/me，获取当前登录用户信息，需要在Authorization请求头中携带Token。现有接口变更：GET /api/tasks、POST /api/tasks、PUT /api/tasks/{id}、DELETE /api/tasks/{id}均需要认证，未携带Token返回401。用户模型包含id（字符串）、username（字符串）、createdAt（时间戳）。认证错误响应使用RFC 9457格式，未认证返回401，用户名或密码错误统一返回“用户名或密码错误”，不泄露用户是否存在。所有项目文档使用中文编写。
```

CodeBuddy将自动更新项目规范文件，在原有基础上补充认证模块相关内容。后端API服务项目规范新增认证模块内容示例如图7-3所示。

● Write(specs\002-user-auth\checklists\requirements.md)
  └Wrote 36 lines (ctrl+o to expand)

● Read(.specify\extensions.yml)
  └Error reading file

● 完成报告

分支名称：002-user-auth
规范文件：specs/002-user-auth/spec.md
检查清单：specs/002-user-auth/checklists/requirements.md

验证结果

所有验证项均通过：
- 内容质量：无实现细节，聚焦用户价值
- 需求完整性：无可澄清标记，需求可测试
- 功能就绪：所有功能需求有明确的验收标准

规范摘要

项目	内容
用户故事	4个（注册P1、登录P2、获取用户信息P3、任务接口认证保护P4）
功能需求	18项
成功标准	6项
关键实体	用户、Token

图7-3 后端API服务项目规范新增认证模块内容示例

7.2.3 更新前端任务看板项目规范

后端API服务项目规范更新完成后，需更新前端任务板项目规范。请切换至前端项目目录，在系统终端中启动CodeBuddy，进入交互界面后，输入以下命令：

```
/speckit.specify更新前端任务板项目规范，在现有任务看板功能基础上新增登录功能。新增页面：登录页面，包含用户名输入框、密码输入框、登录按钮、注册按钮。登录流程：用户输入用户名和密码，点击登录按钮，系统调用后端登录接口，登录成功后跳转到任务看板页面；登录失败时显示“用户名或密码错误”。注册流程：用户点击注册按钮，系统显示注册表单，用户在表单中输入用户名和密码，随后系统调用后端注册接口，注册成功后自动登录并跳转到任务看板页面。认证状态：未登录时自动跳转到登录页面；Token过期后自动跳转到登录页面；已登录用户访问登录页面则自动跳转到任务看板页面。所有项目文档使用中文编写。
```

CodeBuddy会根据需求自动更新前端项目规范文件，补充登录认证的相关业务规则与交互逻辑。前端项目规范新增登录认证相关内容示例如图7-4所示。



图7-4 前端项目规范新增登录认证相关内容示例

7.2.4 确认规范并停止

确认前后端项目规范后，需分别在对应项目的CodeBuddy交互界面中执行停止操作。

切换至后端项目的CodeBuddy交互界面，输入以下命令：

```
后端项目规范已确认，请停止当前操作。
```

切换至前端项目的CodeBuddy交互界面，输入以下命令：

```
前端项目规范已确认，请停止当前操作。
```

7.3 更新方案：安全模块的设计

在7.2节中，我们扩充了前后端项目的规范，新增了认证接口和登录页面的需求。本节将使用 `/speckit.plan` 确定安全模块的技术方案——直到此时，Spring Security、JWT、BCrypt等安全技术栈才正式登场。

7.3.1 理解安全模块的技术选型

安全模块的技术选型需从4个核心维度展开系统性考量，安全模块技术选型决策表如表7-3所示。

表7-3 安全模块技术选型决策表

设计维度	核心问题	技术选型
身份认证机制	如何可靠验证用户身份合法性	用户名+密码组合认证
密码安全存储	如何在存储层保障密码不可逆向解析	BCrypt自适应哈希算法
会话状态管理	如何在分布式环境维持登录状态	JWT无状态Token机制
接口访问控制	如何拦截未授权的接口访问请求	Spring Security请求拦截器

技术选型将通过 `/speckit.plan` 命令实现。

7.3.2 更新后端API服务项目计划文件

请打开系统终端，切换至后端项目所在目录，启动CodeBuddy并进入交互界面，然后输入以下命令：

```
/speckit.plan更新后端技术方案，在现有Spring Boot方案基础上新增安全模块。认证框架：使用Spring Security的Security FilterChain处理认证，自定义UsernamePasswordAuthenticationFilter处理登录请求。Token机制：使用JWT（jjwt库）生成和验证Token，Token有效期为24小时，Token中包含用户ID和用户名。密码加密：使用BCryptPasswordEncoder进行密码哈希和验证。数据库：在现有tasks表基础上新增users表，字段包括id（UUID）、username（VARCHAR(20)，唯一）、password（VARCHAR(60)，BCrypt哈希值）、createdAt（TIMESTAMP）。tasks表新增user_id字段关联用户，实现任务归属。接口保护：使用SecurityFilterChain配置哪些接口需要认证（/api/tasks/**需要认证，/api/auth/**公开访问）。错误处理：认证失败返回RFC 9457格式的401响应，不泄露用户是否存在。所有项目文档使用中文编写。
```

CodeBuddy完成更新后，计划文件将新增安全模块的详细技术设计，后端安全模块技术设计新增内容示例如图7-5所示。

规划报告

分支：002-user-auth
实现计划：D:\task-board-backend\specs\002-user-auth\plan.md

生成的文档

文件	说明
plan.md	实现计划（技术上下文、宪法检查、项目结构）
research.md	技术研究（Spring Security、JWT、BCrypt选型）
data-model.md	数据模型（User实体、Task修改、DTO定义）
contracts/openapi.yaml	API契约（新增认证接口、修改任务接口）
quickstart.md	快速启动指南（环境配置、接口调用示例）

技术方案摘要

- 认证框架：Spring Security 6.x + SecurityFilterChain
- Token机制：JWT（jjwt 0.12.x），有效期24小时
- 密码加密：BCryptPasswordEncoder
- 数据库变更：新增users表，tasks表新增user_id字段
- 接口保护：/api/auth/** 公开，/api/tasks/** 需要认证
- 错误处理：RFC 9457格式，统一返回“用户名或密码错误”

图7-5 后端安全模块技术设计新增内容示例

7.3.3 更新前端任务看板项目计划文件

后端技术方案更新完成后，接下来同步更新前端技术方案。请切换到前端项目所在目录，在系统终端中启动CodeBuddy，进入交互界面后，输入以下命令：

```
/speckit.plan更新前端技术方案，在现有React方案基础上新增登录模块。登录页面：新建Login组件，包含用户名输入框、密码输入框、登录按钮、注册按钮，使用表单提交信息，调用后端登录接口。Token管理：登录成功后将Token存储在sessionStorage中，页面关闭时自动清除。请求拦截：在api.ts中添加请求拦截逻辑，所有请求自动在Authorization头中附加Token。响应拦截：统一处理401响应，自动清除Token并跳转到登录页面。路由管理：新增React Router，未登录时显示登录页面，已登录时显示任务看板页面，Token过期时自动跳转到登录页面。所有项目文档使用中文编写。
```

CodeBuddy完成更新后，前端计划文件将新增安全模块的详细技术设计，涵盖登录页面组件设计、Token管理机制、请求拦截逻辑等核心实现方案。前端安全模块技术设计新增内容示例如图7-6所示。

现计划路径：D:\task-board\specs\002-user-auth\plan.md

成的文档

文档	路径
实现计划	specs/002-user-auth/plan.md
技术研究	specs/002-user-auth/research.md
数据模型	specs/002-user-auth/data-model.md
API 契约	specs/002-user-auth/contracts/api-contracts.md
快速入门	specs/002-user-auth/quickstart.md

术方案概要

增依赖：react-router-dom ^6.20.0

心模块：
路由管理：React Router 6，支持路由守卫（ProtectedRoute）
Token 存储：sessionStorage，页面关闭时自动清除
请求拦截：封装 Fetch API，自动附加 Authorization 头
响应拦截：统一处理 401 响应，清除 Token 并跳转至登录页
状态管理：AuthContext + authReducer

增组件：
Login - 登录组件
RegisterForm - 注册表单组件
ProtectedRoute - 路由守卫组件
LoginPage / BoardPage - 页面组件

图7-6 前端安全模块技术设计新增内容示例

7.3.4 确认计划并停止

前后端技术方案均确认无误后，请分别在对应项目的CodeBuddy交互界面中执行停止操作。

切换至后端项目的CodeBuddy交互界面，输入以下命令：

```
后端技术方案已确认，请停止当前操作。
```

切换至前端项目的CodeBuddy交互界面，输入以下命令：

```
前端技术方案已确认，请停止当前操作。
```

7.4 前后端功能拆解与实现

前三节我们更新了前后端的宪法、规范和技术方案。本节将使用 `/speckit.tasks` 和 `/speckit.implement` 实现安全模块。由于安全加固涉及前后端两个项目，且前端需要依赖后端的接口定义，我们将采用先后端、再前端的顺序来进行实现。

7.4.1 后端API服务项目：拆解任务

请打开系统终端，切换至后端项目所在目录，启动CodeBuddy并进入交互界面，然后输入以下命令：

```
/speckit.tasks
```

CodeBuddy会自动读取更新后的规范文件与技术方案，智能拆解出安全模块的开发任务，形成任务清单。安全模块任务清单生成过程示例如图7-7所示。

最终生成任务清单报告，此时系统会自动切换到新分支 `002-user-auth` ，并在该分支下生成 `tasks.md` 文件。新分支下生成的 `tasks.md` 文件如图7-8所示。

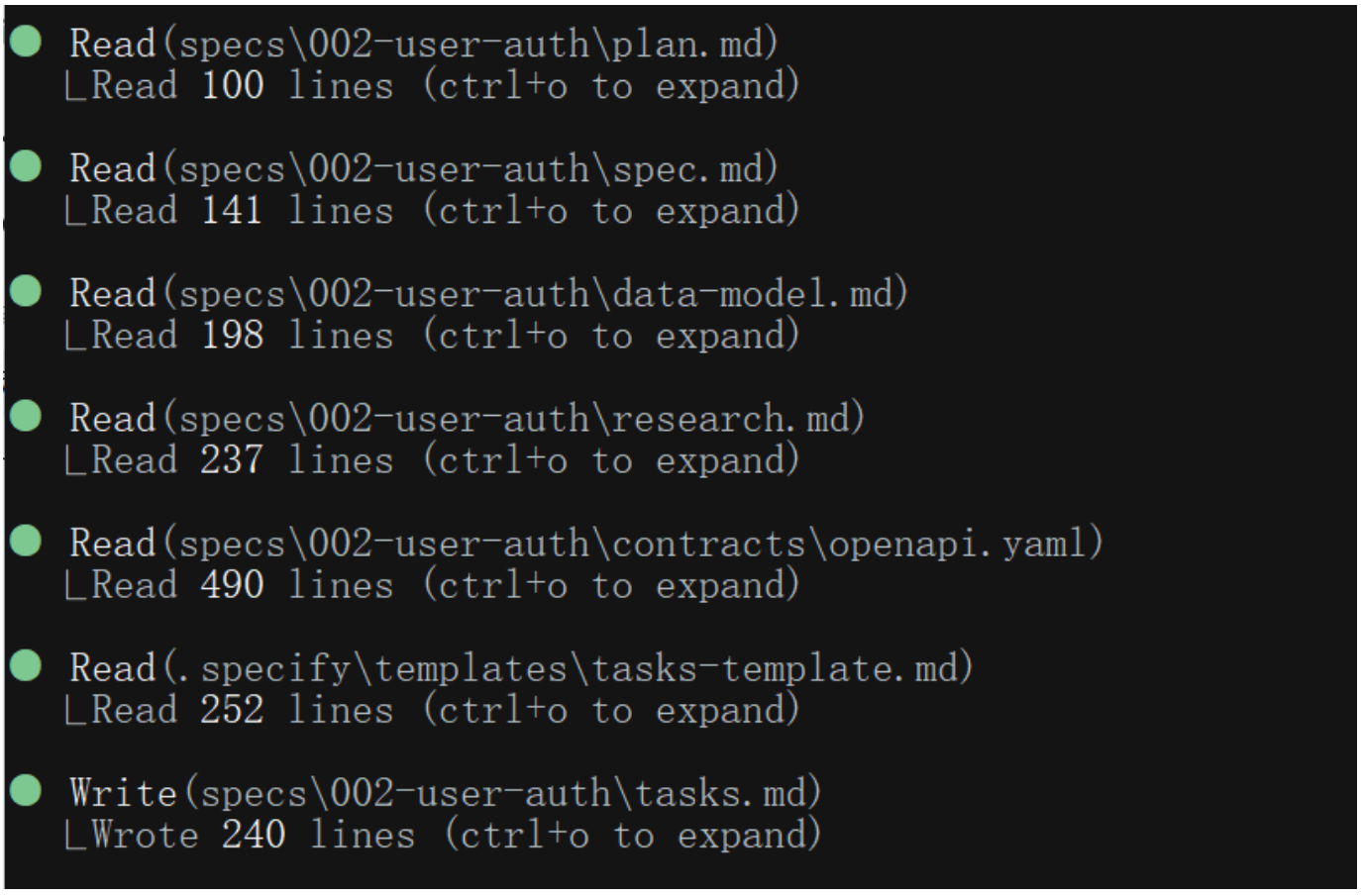


图7-7 安全模块任务清单生成过程示例

● 任务清单已生成。

报告

生成文件：D:\task-board-backend\specs\002-user-auth\tasks.md

任务统计

阶段	任务数	可并行
Phase 1: Setup	5	4
Phase 2: Foundational	9	1
Phase 3: US1 - 用户注册	8	3
Phase 4: US2 - 用户登录	5	2
Phase 5: US3 - 用户信息	4	0
Phase 6: US4 - 任务保护	5	0
Phase 7: Polish	6	2
总计	42	12

图7-8 新分支下生成的tasks.md文件

7.4.2 后端API服务项目：实现任务

在CodeBuddy交互界面中输入以下命令，CodeBuddy会自动按任务清单上的顺序逐项实现，直到完成任务。

```
/speckit.implement
```

7.4.3 后端API服务项目：验证认证接口

为确保后端API服务功能的正确性与稳定性，我们将通过以下系统化验证流程进行全面测试。

请打开系统终端，切换至后端项目所在目录，执行以下命令启动后端API服务：

```
mvn spring-boot:run
```

打开浏览器，输入http://localhost:8080/swagger-ui/index.html，访问Swagger UI接口文档页面，可看到页面中新增了权限认证相关接口。权限认证相关接口展示如图7-9所示。

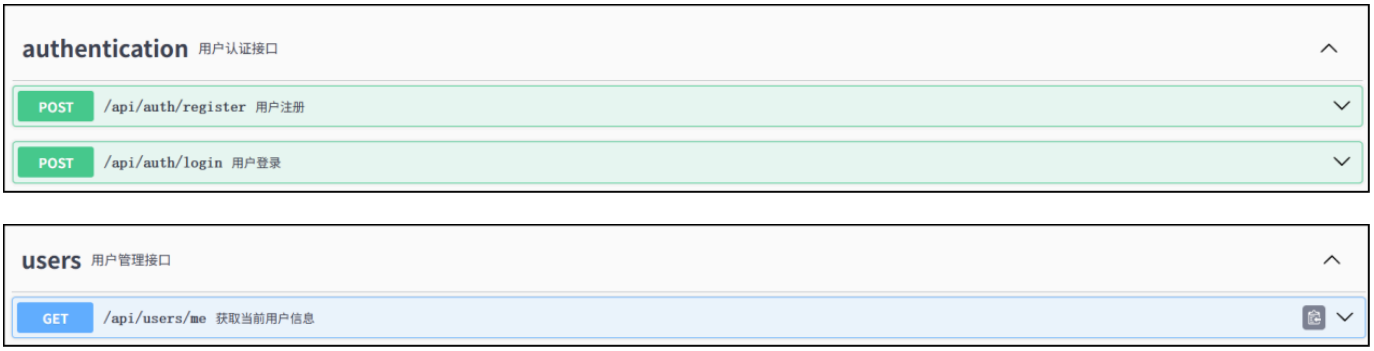


图7-9 权限认证相关接口展示

测试用户注册接口。单击该接口项，在请求体中填写如下JSON格式的用户信息后，单击“Execute”按钮发起测试。用户注册接口测试操作如图7-10所示。

```
{"username":"testuser","password":"Pass1234"}
```

用户注册接口执行成功，返回信息与预期完全一致，表明用户注册接口功能开发验证通过。用户注册接口返回结果示例如图7-11所示。



图7-10 用户注册接口测试操作

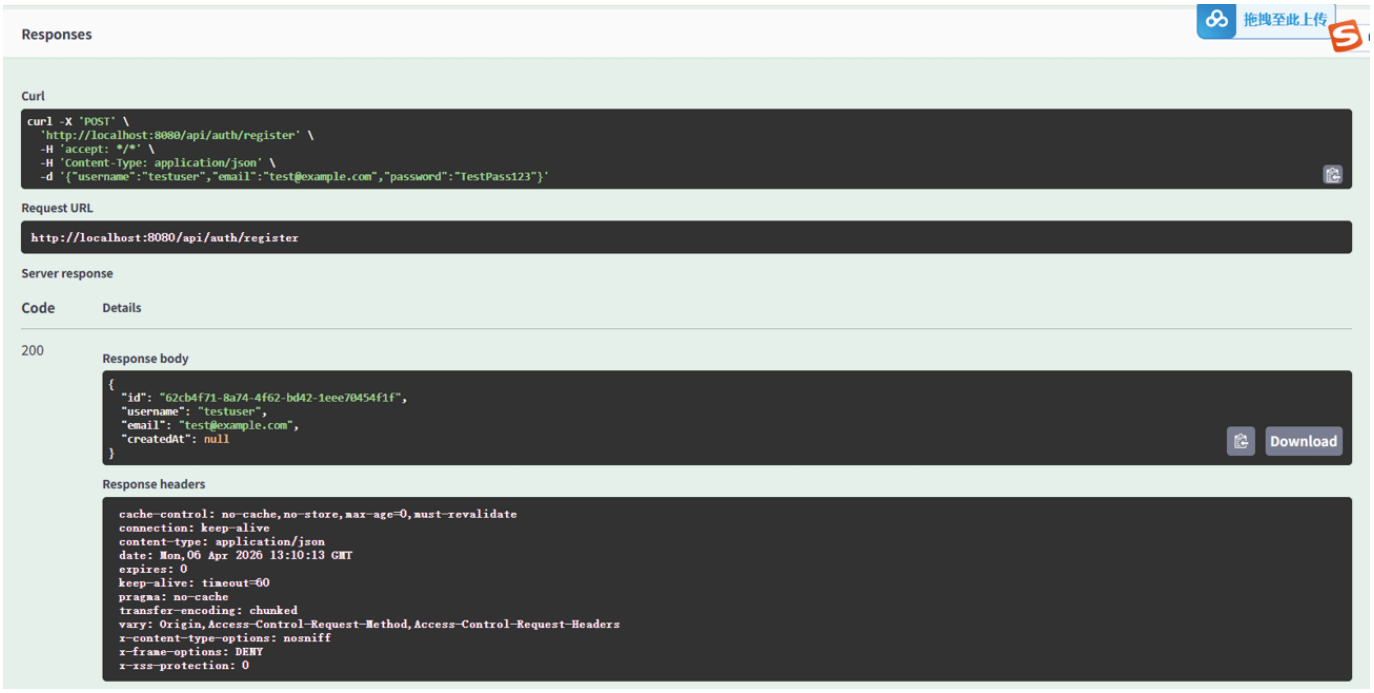


图7-11 用户注册接口返回结果示例

测试用户登录接口。单击该接口项，使用刚注册完成的账号和密码，在请求体中输入对应信息，再单击“Execute”按钮执行登录操作。用户登录接口测试操作如图7-12所示。

用户登录接口执行成功，系统返回了JWT格式的身份认证令牌。用户登录接口返回JWT令牌的示例如图7-13所示。



图7-12 登录接口测试操作

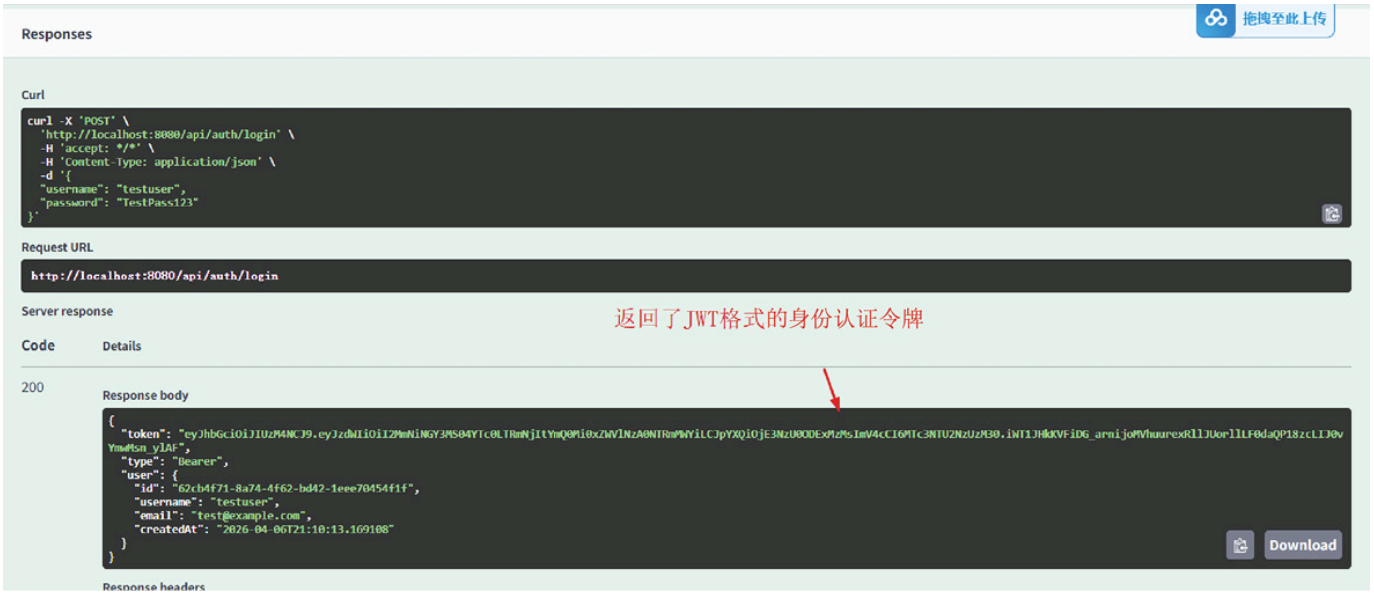


图7-13 用户登录接口返回JWT令牌的示例

其他接口的测试流程与上述示例类似，此处不再逐一演示。完成所有接口的验证工作后，我们需要告知CodeBuddy停止当前任务。在交互窗口中输入以下命令：

```
后端认证接口已验证通过，请停止当前操作。
```

7.4.4 同步后端API契约到前端

后端项目实现完成后，我们需要将更新后的后端API契约同步到前端项目。这个环节很关键——前端开发人员必须先了解后端项目提供了什么接口，才能编写对应的调用代码。

请切换至前端项目所在目录，在系统终端中启动CodeBuddy交互环境。进入交互界面后，输入以下命令，让CodeBuddy读取后端API契约文件：

```
请读取D:\task-board-backend\specs\002-user-auth\contracts\openapi.yaml文件，了解后端提供的所有登录认证的相关接口。
```

7.4.5 前端项目：拆解任务

进入CodeBuddy交互界面后，在命令行提示符后输入如下命令：

```
/speckit.tasks
```

CodeBuddy会自动读取更新后的前端规范与技术方案，智能拆解出登录模块的开发任务，形成任务清单。执行完成后，前端项目将自动创建新分支 `002-user-auth` ，并在该分支下生成 `tasks.md` 文件。前端登录模块任务清单文件生成结果如图7-14所示。

● 任务列表已生成。以下是摘要：

任务列表生成完成

文件路径：D:\task-board\specs\002-user-auth\tasks.md

任务统计

阶段	任务数	可并行	描述
Phase 1: Setup	3	2	项目设置和依赖安装
Phase 2: Foundational	8	4	核心基础设施（阻塞所有用户故事）
Phase 3: US1 登录	11	6	用户登录功能 🎯 MVP
Phase 4: US2 注册	7	2	用户注册功能
Phase 5: US3 状态管理	6	2	认证状态管理
Phase 6: 收尾	7	3	文档、测试覆盖率、安全验证
总计	42	19	

MVP 实现路径

1. Phase 1: 安装依赖、配置 Vite 代理、创建目录
2. Phase 2: Token 存储、API 客户端、AuthContext、路由守卫
3. Phase 3: 登录页面、登录组件、表单验证、API 调用

图7-14 前端登录模块任务清单文件生成结果

7.4.6 前端项目：实现安全任务

进入CodeBuddy交互界面后，输入如下命令，CodeBuddy会自动按照任务清单逐项完成。

```
/speckit.implement
```

7.4.7 前端项目：验证安全认证功能

为验证前端项目安全认证功能的开发成果，我们将通过以下测试步骤进行验证。

任务清单上的任务都完成后，请确保后端API服务处于运行状态，随后在系统终端中执行以下命令，启动前端项目：

```
npm run dev
```

在浏览器中访问http://localhost:5173，系统将自动跳转至登录页面，如图7-15所示，引导用户完成身份验证。



图7-15 登录页面

单击登录页面中的“立即注册”链接，系统将自动跳转到注册页面，用户可在此页面完成新账号的注册操作。注册页面如图7-16所示。



图7-16 注册页面

单击“注册”按钮提交信息，注册成功后系统将自动完成登录流程，并跳转至任务面板页面，此时你将看到个人专属的任务看板界面，如图7-17所示。

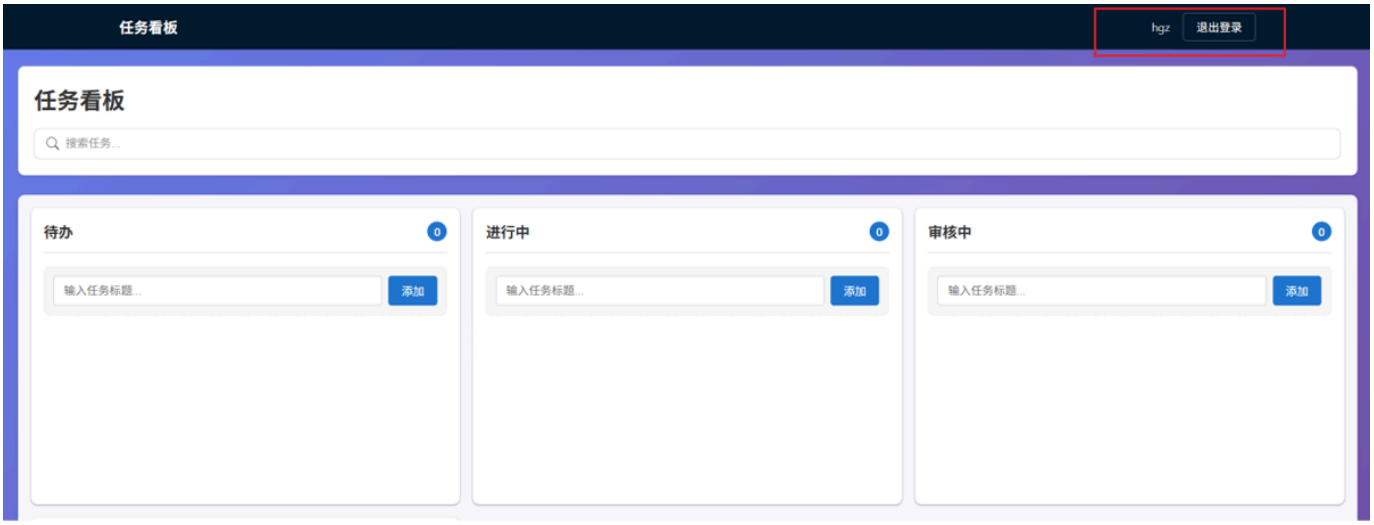


图7-17 个人专属任务看板界面

完成上述基础验证后，我们可以对照任务清单进行逐项全面验证，确保功能的完整性与稳定性。至此，基于Spec-Kit的核心功能开发已全部完成，大家可在此基础上，运用书中讲解的方法，进一步拓展和升级系统功能。

第三部分　存量系统改造与未来展望

第8章　存量系统Spec-Kit开发实践：存量系统设计知识和代码理解

在本书前面的章节里，我们花了大量篇幅讨论SDD在新应用开发中的美妙之处：先写规范，再写代码，规范即契约，代码即实现，二者始终同步。你可能会想，这当然好，可我手头维护的是一个已经运行了五六年甚至更久的存量系统，代码像一座被修补过无数次的老房子，文档早已过时，当初写代码的人可能都离职了。这种情况下，SDD还有用武之地吗？

答案是肯定的，而且恰恰是在存量系统中，SDD能发挥出意想不到的价值。本章我们就来聊聊如何用SDD的方法在一团乱麻中找回秩序，让老系统也能具备规范带来的安全性与演进效率。

8.1 存量系统面临的独特挑战

在开始动手之前，我们先冷静地看看存量系统到底给我们出了哪些难题。只有把问题想透了，才能对症下药。

8.1.1 文档缺失、代码错误的困境

我见过太多团队，项目的Wiki最后一次更新是两年前，设计文档里的架构图跟现在的代码完全对不上。唯一可信的“文档”就是源代码本身。但源代码是面向计算机的，不是面向人的，要从几百行的长方法、难以理解的变量名、散落在各处的业务判断中还原出“这个系统到底应该做什么”是一件极其痛苦的事。

更麻烦的是，很多逻辑根本没有文字记录。比如某个金额字段乘以了0.95，代码里没有任何注释。问老员工，对方可能会说“那是当年为了配合营销活动打的折扣，后来活动结束了，但代码没改回来”。所以你看，有时候，代码不仅不告诉你“应该做什么”，甚至包含本不该存在的“历史垃圾”。

8.1.2 隐含业务逻辑与“神秘代码”现象

存量系统里隐含的业务逻辑十分让人头疼。它们不是写在需求文档里的，而是某次线上故障时临时加的补丁；或者某个用户提出特殊需求，开发人员直接写下的if(userId==12345){...}。没有人知道这些逻辑的全貌，但它们确实在运行着，影响着真实用户。

我把这类代码称为“神秘代码”，我们不知道它为什么存在，但一旦删掉，系统就会出错，甚至崩溃。SDD要求我们先有规范再有代码，可面对存量系统，我们连现有的系统行为都很难说清楚，更别提规范了。

8.1.3 系统脆弱，开发者不敢轻易重构

因为缺乏对系统行为的清晰描述，每次修改都像在雷区里行走。你改了一行代码，以为只是在优化性能，结果某个看似与这行代码关系不大的报表模块出了问题。这种情况下，测试覆盖率往往也不高，集成测试更是稀缺。于是团队形成了“能不动就不动”的文化，技术债务越积越多，系统越来越臃肿。

这种脆弱性也让SDD的引入变得困难。规范本身需要验证，而验证需要测试。如果现有系统连可测试性都很差，那规范的提取和验证就会举步维艰。

8.1.4 SDD对存量系统的价值再审视

那为什么我们还要自讨苦吃？因为存量系统恰恰最需要规范。想象一下，如果你能从混乱的代码中抽离出一份准确、可执行的规范，会发生什么：

- 新人接手时，不再需要通读几万行代码才能理解业务规则，看规范就够了。
- 当你需要修改某个功能时，规范会告诉你哪些地方可能受影响（因为规范之间有依赖关系）。
- 以后任何人修改代码，只要运行一遍规范测试，就知道有没有破坏原有行为。

所以，SDD在存量系统中的角色，不是从零开始建高楼，而是给一座摇摇欲坠的老房子画一张精确的结构图，然后在它的基础上安全地改造。

8.2 存量系统理解：从代码到知识

既然要提取规范，第一步自然是理解现有的系统。这一节我们不讲空泛的理论，而要讲实实在在的工具和方法，帮助你从代码、运行时数据、架构文档（如果有的话）中挖掘出知识。

8.2.1 静态分析：模块结构、调用链、数据流

静态分析就是不需要运行程序，只分析源代码或编译产物。这是最基础的了解系统的方法。静态分析主要有以下几种方式。

模块结构分析。如果你的系统采用Java开发，可以用jdeps或者IDE里的依赖图功能，看看每个包依赖了哪些其他包。很多时候，你以为的“分层架构”实际上已经变成了“网状结构”。我会用Graphviz生成依赖关系图，一眼就能看出哪些模块是核心、哪些模块已经没人维护了。

调用链分析。找到核心入口（比如Controller或者消息监听器），然后顺着方法调用追溯。这里推荐使用IDE的“Call Hierarchy”功能。对于大型系统，可以借助字节码分析工具（如ArchUnit）来编写脚本，自动提取某个入口到数据库访问之间的所有方法。我通常会以表格形式记录调用链中的入口方法、中间业务方法、数据访问方法、可能抛出的异常。

数据流分析。更进一层，追踪一个数据项（比如订单金额）从输入到输出的整个变化路径。这比调用链分析更难，因为涉及赋值、类型转换、条件分支。你可以从数据库实体类开始，反向查找哪些方法在写这个实体的某个字段。有些静态分析工具（如Soot、WALA）能帮你做数据流分析，但学习成本较高。一个实用的“土办法”是：在关键字段的setter方法里加日志（用面向切面编程或者直接修改代码），执行典型业务流程，观察日志输出。

8.2.2 动态分析：运行时行为捕获

静态分析只能告诉你“可能发生什么”，动态分析则能告诉你“实际发生了什么”。二者结合才能看到全貌。动态分析主要有以下几种方式。

日志与链路追踪。如果系统已经接入了ELK或SkyWalking，它们会记录，大量的运行时数据。你可以从日志中提取出请求-响应对，统计不同输入下的输出分布。例如，查询“退款接口在过去一周被调用了多少次，参数中的退款原因都有哪些”。这些信息对于后续生成规范的示例非常有帮助。

典型场景快照。手动或通过自动化脚本，运行一遍系统的主要业务流程，记录每个步骤的输入输出。我会用Postman或curl把请求保存下来，同时利用tcpdump或Wireshark抓取网络包，确保不遗漏任何细节。对于数据库，可以在测试环境中开启SQL（Structured Query Language，结构化查询语言）日志，观察每个操作执行了哪些查询。

生产流量录制。如果系统允许，可以用流量录制工具（如GoReplay、Nginx mirror模块）复制一份生产请求到测试环境，然后观察系统在真实流量下的行为。注意，这种分析方式的前提是处理好敏感数据脱敏和存储成本问题，所以必须谨慎使用。

8.2.3 架构与集成分析

存量系统往往不是孤岛，它会和各种外部系统打交道。不了解相关的架构和集成，你就无法理解某些“奇怪”的行为，比如为什么这个接口的响应特别慢？因为它在等一个老旧的SOAP（Simple Object Access Protocol，简单对象访问协议）服务。在架构与集成分析中，以下几点需要特别关注。

识别核心模块与外围模块：建议先找出那些决定了业务能否运转的模块，比如订单、用户认证、支付结算。这些是规范提取的优先目标。外围模块如报表、定时清理任务，可以稍后处理。

梳理外部依赖。很多老系统会在自己与外部系统之间加一层适配器，即防腐层。你需要找出所有外部依赖：数据库（不仅仅是主库，还有从库、缓存）、消息队列、第三方API、文件系统。对于每个依赖，记录它的访问模式（同步/异步、重试策略、超时设置）。这些都会成为规范中的“集成契约”。

定时任务与触发器。不要漏掉那些通过cron或Quartz执行的作业。它们往往是系统里最难调试的部分。列出所有定时任务，弄清楚它们的调度周期、操作的数据范围、执行后的副作用。我曾经遇到一个案例，某个清算任务每天凌晨3点运行，它会更新订单的状态。但没有任何文档记录了这个逻辑，导致白天排查问题时大家百思不得其解。

8.2.4 数据模型理解

代码可以重构，但系统需要长期兼容数据库里的数据。理解数据模型是提取业务规范的关键一步。以下几点需要特别关注。

Schema逆向。用数据库工具（如SchemaSpy、DBDiagram）生成ER图。关注点：哪些表有外键约束，哪些没有（但代码里隐含了关联）；哪些字段允许NULL，实际数据中NULL的比例是多少；有没有使用枚举类型或检查约束。

隐式约束。有些约束不在数据库层面，而是写在应用程序代码里。比如“订单金额必须大于0”可能只在Service层做了校验。你需要把这些隐式规则找出来。方法之一是扫描代码中的if条件，提取出涉及实体字段的判断，汇总成规则列表。

数据质量与异常情况。运行几个分析型的SQL查询，统计每个字段的值的分布。比如status字段理论上只有3种值，但实际查询发现有5种，那额外的两种是哪来的？可能是历史遗留，也可能是代码bug。这些异常情况也需要纳入规范，因为你不能假设生产数据都是“干净的”。

8.3 从存量系统中提取规范：逆向工程实践

当我们对系统有了基本理解之后，就可以正式开始规范的提取了。这一步的目标是把前面获得的碎片化知识，转化为Spec-Kit能够识别的结构化规范。

8.3.1 API规范提取

对于任何有HTTP接口的系统，API规范是最容易入手的地方。可以采用以下方式提取API规范。

从代码生成OpenAPI。如果你的项目使用了Spring MVC，可以用springdoc-openapi库自动生成OpenAPI文档。启动应用后访问/v3/api-docs，就能得到一份完整的API定义。其他框架也有类似工具（比如ASP.NET Core的Swashbuckle）。这份自动生成的规范已经包含了路径、HTTP方法、参数类型、响应状态码等基本信息。但是自动生成的东西往往不够“业务友好”。你需要手工补充：接口的业务含义（description字段）、请求参数的具体示例（example）、错误码的含义（responses下对每个状态码的说明）。

通过流量录制补充示例。这是让规范变得实用的关键一步。从日志或录制流量中挑选出10~20个真实的请求-响应对，作为示例嵌入OpenAPI。这样，阅读规范的人一眼就能看出这个接口通常怎么调用，返回什么数据。这同时也为后续的测试提供了现成的数据。

提取异常场景的规范。不要只记录成功的情况。找出常见的错误响应：参数校验失败、权限不足、资源不存在、服务内部错误。这些也应该写入OpenAPI的responses部分。

8.3.2 业务规则与行为规范提取

API规范只能描述“接口长什么样”，不能描述“业务逻辑是什么”。例如，“创建订单”这个接口要检查库存、计算价格、锁定优惠券，这些规则需要单独提取。常见的业务规则与行为规范提取方式主要有以下几种。

用Gherkin描述关键路径场景。Gherkin是一种自然语言化的需求描述语言，格式是Given-When-Then。例如：

```
Feature: 订单创建
  Scenario: 普通用户购买有库存的商品
  Given用户已登录，账户余额为100元
  And商品“Java编程思想”价格为60元，库存为10件
  When用户请求创建订单，购买1件该商品
  Then订单状态应为“待支付”
  And订单金额为60元
  And库存减少为9件
  And用户余额不变（因为未支付）
```

你要从现有代码中提取出这样的场景。具体方法有两种。

- **正向提取**：阅读核心业务方法的代码，梳理出主要的分支路径。比如createOrder方法里可能有多条分支路径：库存不足→返回错误；优惠券无效→忽略；价格计算→调用定价模块。每一条路径都可以写成一个场景。

- **反向提取**：运行集成测试或录制流量，观察真实输入输出，反推出系统遵循的规则。例如，你发现所有成功的订单创建请求，其totalAmount都等于unitPrice×quantity-discount，那么你就可以写出一条规则：“订单总金额=单价×数量-折扣”。

我建议两种方法结合使用。先用正向提取画出流程图，再用反向提取验证这些分支是否真的会在生产中被触发。

状态机规则的提取。很多业务的核心是一个状态机（订单状态、审批状态等）。从代码中找到所有修改状态的地方，列出状态转换表，如表8-1所示。

表8-1 状态转换表

当前状态	允许的事件	下一状态	前置条件
待支付	支付成功	已支付	支付金额等于订单金额
待支付	取消订单	已取消	用户角色为买家且订单创建时间不超过30分钟
已支付	发货	已发货	订单必须已支付且未发货

这张表可以直接作为规范的一部分，未来任何修改状态机的代码都必须与规范一致。

8.3.3 数据契约与领域模型规范

除了API规范和业务规则，系统的数据也会受到约束。数据契约与领域模型主要有以下几种常见的提取方式。

提取实体不变式。不变式是那些在任何时候都必须成立的条件。例如“订单的总额必须大于0” “用户的邮箱地址要么为空，要么符合邮箱格式”。从代码中寻找这些检查点。你可以搜索assert、if(xxx)throw等语句，形成清单。

值对象约束。很多领域模型中有值对象，比如Money、PhoneNumber。它们的约束往往封装在构造函数或工厂方法里。把这些约束提取出来，写成JSON Schema或Protobuf的校验规则。例如：

```
{
  "type":"object",
  "properties":{
  "amount":{"type":"number","minimum":0,"exclusiveMinimum":
true},
  "currency":{"type":"string","pattern":"^[A-Z]{3}$"}
  }
  }
```

生成数据契约文档。你可以使用工具（如jsonschema2md）把Schema转换成人类可读的表格，方便团队评审。

8.3.4 流程规范提取（工作流提取）

有些业务横跨多个服务和多个步骤，比如“下单→锁库存→扣减积分→发送通知→等待支付”。这种长流程的规范，提取起来比较困难，但也不是没有办法。下面讲解3种提取方式。

基于日志序列挖掘流程拓扑。收集包含请求追踪ID的日志，按时间排序，就能还原出一个请求经过的所有步骤。例如，日志中有[traceId=abc]step:validateInventory、[traceId=abc]step:lockStock、[traceId=abc]step:deductPoints。把这些步骤连起来，就得到了典型路径。

绘制状态图。将每一步视为一个状态，状态之间的转换由事件触发（如“库存锁定成功”）。由此，你可以画出一张状态图（Statechart），这张图本身就是一种规范。对于补偿场景（比如扣积分失败后回滚库存），也要在图中标明。

借助工具。如果系统使用了工作流引擎（如Camunda、Activiti），那么可以直接导出BPMN文件。否则，需要手工整理出一份流程定义文档。

8.4 规范验证：确保规范与现有系统一致

提取出来的规范只是我们“认为”的系统行为。它是否反映了系统的真实行为？本节我们就来验证两者的一致性。

8.4.1 规范与实现的一致性测试

我们把规范转化成可执行的测试用例，然后在现有系统上运行这些测试。

具体做法：对于每个API规范，用工具（如Postman、Rest Assured、pytest）生成请求测试，断言响应状态码和关键字段。对于Gherkin规范，可以用行为驱动开发的工具，比如Cucumber或Behave将其转化为自动化测试。对于数据契约，写一些数据生成器，随机产生符合规范的数据，然后尝试存入系统，看系统是否拒绝不符合规范的数据。

运行这些测试后，你可能会发现规范与实现不一致的情况。

- **测试失败**：规范说应该返回200，实际返回了400。那可能是规范写错了，也可能是系统有bug。

- **测试通过但规范遗漏**：规范没有提到某个字段，但实际返回里总是有这个字段。这时候需要补充规范。

对于存量系统，这种“测试即文档”的规范验证方式，比任何手工检查都可靠。

8.4.2 构建回归测试基线

一旦确认规范与当前系统一致，我们就把它冻结为“回归测试基线”。以后任何人想修改系统，系统都会先运行一次基线测试，确保修改不会破坏原有行为。

若修改会造成破坏，要么修复代码，要么更新规范。这样，规范就真正成为了系统的守护者。

8.4.3 处理歧义与非确定性行为

存量系统中存在一些“不确定”的行为，比如依赖当前时间、随机数、外部系统的返回。这些行为很难写成固定的测试断言。以下几种情况都是比较常见的。

多版本规范共存。有些逻辑区分了不同租户或不同用户类型。你可以把每个租户的规范单独提取，用标签区分。测试时分别运行。

容忍“规范未覆盖但合理”的行为。如果测试发现系统返回了规范中没有定义的字段，但该字段（比如时间戳）不影响业务，可以暂时忽略。但你应该记录下这些发现，决定是否要补充进规范。

处理概率性行为。比如一个抽奖接口有10%的中奖率。那么，写“每次都通过”的断言就不合理。可以改用统计测试：运行1000次，断言中奖次数符合中奖概率。或者把随机性作为规范的一部分明确写出来：“中奖概率约为10%。”

8.5 在存量系统中增量落地Spec-Kit

规范和验证都完成了，接下来就是真正把Spec-Kit的工作流程融入日常开发。切记：不要试图一夜之间改造整个系统。

8.5.1 选择首个子域

选择一个合适的子域作为试点，可以大大提高成功率。建议按以下标准来选择子域。

- **高业务价值**：这块代码经常被修改或出问题，改进的收益明显。
- **低耦合**：与其他模块的交互简单，容易隔离测试。
- **可独立验证**：有明确的输入输出，可以通过黑盒测试来验证。

“订单状态机”或“价格计算模块”就比较适合做试点。避开那些与外部系统深度耦合的模块，比如支付网关对接。

8.5.2 建立规范基线固定系统基准行为

对选中的子域，按照8.3节和8.4节介绍的方法，提取规范并验证其与现有系统是否一致。然后将这套规范作为该子域的“行为合同”，任何未来的变更都必须遵循这个合同，或者先更新合同再变更。

这一步最好能得到团队和业务方的确认。你可以召开一次规范评审会，展示提取出来的Gherkin场景和规则表格，问大家：“这是否符合你们对业务的理解？”如果有分歧，那正是澄清需求的好机会。

8.5.3 双轨工作流

在引入Spec-Kit的初期，可以采用双轨并行的工作流，双轨并行的工作流分为**对遗留模块和对新功能两种。**

对遗留模块：继续以原有的方式维护。但是，当有人需要修改这部分代码时，必须遵守“先更新规范再改代码”的规则。这意味着每一次变更都是对规范的一次小迭代。随着时间的推移，规范的覆盖率会逐渐提高。

对新功能：严格遵循Spec-Kit的先规范后开发流程。新功能要么是在存量系统中新增一个独立模块（比如新增报表接口），要么是对现有模块的扩展（比如给订单状态增加一个“部分退款”状态）。对于后者，你需要先更新状态机规范，再写代码。

双轨模式能让团队慢慢适应规范驱动的工作方式，而不会感到被强行要求立刻重构整个系统。

8.5.4 渐进式重构：用规范驱动安全替换旧模块

存量系统中有些模块已经无法修补，必须重写。SDD可以大大降低重写的风险。

重写的具体步骤如下：

（1）完整提取目标模块的行为规范（包括正常路径和异常路径）。

（2）将这些规范转化为回归测试集，在旧模块上运行，确保测试通过。

（3）用新架构重写模块，但保持对外接口不变。

（4）在新模块上运行同样的回归测试集。

（5）测试全部通过后，在测试环境中替换旧模块，进行更多集成测试。

（6）逐步切换流量，监控错误率，最后完全替换。

这种方法被称为“规范驱动的重写”，它比传统重写方法安全得多，因为你始终有一条行为基线作为“护栏”。

8.5.5 组织与协作建议

引入SDD不仅能解决技术问题，还有助于改变团队协作方式。以下是一些实践经验。

- **规范评审纳入Definition of Done**：每个功能完成前，必须经过规范评审，确保规范清晰、可测试。

- **知识库沉淀**：把提取出来的规范（OpenAPI、Gherkin、数据契约）存放在版本控制系统中，并用工具自动生成可视化的文档网站（比如用Redoc展示API，用Cucumber Reports展示场景）。

- **CI强制执行**：在CI流水线中加入规范测试步骤，规范测试失败时不允许合并代码。

- **角色调整**：可以设立“规范守护者”角色，负责审核规范的质量和一致性。这个角色通常由资深的开发人员或系统架构师担任。

8.6 综合案例：为某电商订单系统引入SDD

理论要通过实战来演练。本节的案例基于我在一个中型电商平台的实际经历，我对涉及的名字和细节做了脱敏处理。

8.6.1 系统背景与痛点

该电商平台运行了5年，团队经历了几轮人员变动，代码逐渐腐化。在此基础上，对订单系统进行迭代开发主要有以下痛点：

- 订单状态转换逻辑分散在多个Service中，有时一个订单会卡在“待支付”状态，原因不明。
- 价格计算涉及满减、会员折扣等复杂的促销规则，但是没有任何文档。每次“大促”前，运营人员都要反复测试，生怕算错。
- 新来的开发人员需要花费两个月才能搞懂订单核心流程。
- 修改订单相关功能时，经常引发线上故障，团队士气低落。

我们决定用SDD来改善这个系统，首个子域选择了订单状态机和价格计算模块。

8.6.2 规范提取过程演示

1．从订单状态机代码中提取状态转换规则

我们用静态分析找到了所有修改order.status字段的代码位置，汇总成了一张状态转换表。初始表格有20多行，后来我们发现有些转换是重复的（比如通过不同路径到达同一结果）。我们整理后得到核心的7种状态和12种允许的转换。

在这个过程中，我们发现了一个有趣的现象：代码中有一个void forceComplete(orderId)方法，它会把任何状态的订单直接变成“已完成”状态。这个方法的注释写着“用于客服手动处理”。但我们翻遍了需求文档和工单系统，都没有找到这种操作的业务规则。后来问了老员工才知道，这是3年前为一个大客户临时添加的方法，现在那个客户已不使用了。于是我们决定废弃这个方法（当然要先确认没有外部调用）。

2．从订单Service录制关键场景，生成Gherkin

我们编写了一个自动化脚本，在测试环境自动创建各种类型的订单（普通、会员、促销商品等），然后通过AOP拦截订单Service的方法调用，记录输入参数和返回值。同时，我们从生产日志中抽取了100个真实订单创建请求，然后把这些数据转化为Gherkin场景，例如：

```
Scenario: 会员使用满100元减20元的优惠券购买非促销商品
  Given用户是黄金会员
  And用户有一张满100元减20的优惠券
  And商品价格分别为60元和50元，均非促销商品
  When用户下单购买这两件商品并使用优惠券
  Then订单总金额应为90元
  And优惠券被标记为已使用
```

我们总共生成了30多个场景，覆盖了正常流程、边界条件（如优惠券已过期、库存不足）和异常情况。

3．从API层导出OpenAPI并补充示例

订单系统有RESTful API，我们用springdoc-openapi生成了基础OpenAPI文档。然后从录制流量中挑选了10个典型请求-响应对，作为examples嵌入每个操作的文档中。此外，我们还补充了错误码说明，比如ORDER_001表示“库存不足”，ORDER_002表示“优惠券无效”。

8.6.3 验证与问题发现

我们把提取出的规范（状态转换表、Gherkin场景、OpenAPI）转化为自动化测试，在现有的订单系统上运行。

结果令人惊讶：有好多个场景失败了，这里列举其中有代表性的3个场景。

场景一：“普通用户购买预售商品，支付尾款时使用优惠券”。测试失败。原来，预售订单的尾款支付逻辑中，优惠券的校验条件写错了，导致用户可以使用不满足条件的优惠券。这是一个真实存在的bug，只是之前的测试没有覆盖，所以该bug从未被发现。

场景二：“商品已发货，用户申请退款”。规范要求状态变为“退款中”，但实际代码会直接状状态变为“已退款”。这违反了业务规则（若商品已发货，需要经过退货流程，系统才能退款）。

场景三：规范中规定“待支付→已取消”只能由买家发起，且必须在下单后的30分钟内取消订单。但实际代码中，一个后台批处理任务会无条件取消超过24小时的未支付订单。这个规则从未被文档化。我们更新了规范，增加了这条规则。

发现这些问题后，团队对SDD的态度从怀疑变成了信服。原来我们自以为熟悉的系统，其实充满了未知。

8.6.4 新需求开发：使用Spec-Kit流程新增“订单拆分”功能

在规范提取完成后的第二个月，产品经理提出了一个新需求：支持用户将一个订单拆分成多个子订单，每个子订单可以独立支付、发货。这个功能在旧系统上实现起来风险很高，因为我们不确定会影响哪些现有逻辑。

这次我们严格按照Spec-Kit流程来开发，以下是完整的过程。

1．编写规范

团队用两天时间讨论并编写了订单拆分的业务规则，包括：

- 只有“待支付”状态的订单可以拆分。
- 拆分后原订单状态变为“已取消”，生成多个子订单，每个子订单的金额、商品、收货地址可重新分配。
- 原订单使用的优惠券按照子订单金额比例分摊。
- 拆分操作不可逆。

我们编写了十几个Gherkin场景，覆盖各种边界情况。

2．生成测试用例

我们用大模型结合Cucumber自动生成了测试用例。

3．实现功能

开发人员使用Spec-Kit方法，根据规范编写代码，边写边测试。遇到规范模糊的地方，就停下来和产品经理澄清，并更新规范。

4．验证

所有测试通过后，我们在测试环境运行了全量回归测试（包括之前提取的所有规范），没有发现任何破坏原有行为的问题。

5．上线

功能上线后，运行平稳，没有出现拆分逻辑导致的故障。整个开发周期比团队预估的缩短了30%，因为规范大大减少了沟通成本和返工情况。

8.6.5 成果与经验总结

半年后，订单系统的状态有了明显改善。

- 新入职的开发人员通过阅读规范（Gherkin场景和状态转换表），一周内就能理解订单核心逻辑，用时比之前缩短了75%。
- 订单相关故障率下降了60%，因为每次变更都有回归测试保驾护航。
- 团队开始主动为其他模块（如库存、促销模块）提取规范，形成了一种“规范文化”。

我们的经验可以总结为几点。

- 从问题紧迫的模块开始，不要试图一次性覆盖所有代码。
- 规范随代码一起演进。
- 自动化是关键，手动维护规范不可持续，要尽量从代码和流量中生成规范。
- 业务方的参与让规范更准确，也有助于提前发现需求理解的偏差。
- 严格遵守Spec-Kit的原则，避免规范和代码不一致。

8.7 常见陷阱与应对策略

在帮助多个团队落地存量系统的SDD之后，我亲历了各种因引入规范而引发的“水土不服”。规范本应是团队的助力，但若使用不当，反而会成为新的“债务”。以下是我总结的4个最典型的陷阱，以及经过实战检验的应对策略。希望你的团队能够绕过这些陷阱，真正让规范为开发提速。

8.7.1 过度提取：规范膨胀症

有些团队在尝到规范的甜头后，便试图“把规范进行到底”，比如为系统中的每一行代码、每一个函数都提取出正式的行为规范。其结果往往令人哭笑不得：规范文件的数量和体积迅速超过源代码本身，阅读规范变成了一场灾难。开发人员宁愿直接读代码也不愿翻规范，规范很快就沦为无人问津的摆设。更糟糕的是，膨胀的规范需要持续维护，一旦跟不上代码变更，就会产生大量过期的错误描述，反而干扰开发。

为什么会出现这种“规范膨胀症”？本质上，这是一种对控制的过度追求。团队误以为规范的覆盖率越高越好，却忽略了规范的维护成本。实际上，并非所有代码都值得被规范约束，那些极少变更、逻辑简单、风险极低的代码，为其维护规范是净亏本的买卖。

那么，如何走出这个误区？我推荐一个朴素却极为有效的原则：**二八原则**。请记住，20%的核心业务逻辑产生了80%的价值和风险。明智的做法是，优先为那些频繁变更、容易出错、业务关键的模块提取规范；而对于那些稳定且简单的工具方法（如字符串判空、日期格式化），或者底层基础设施代码，完全可以暂缓，甚至永远不提取。每个迭代结束后，不妨回顾一下已有的规范文件，主动删除那些已经失去价值或长期未变更的规范。规范不是遗产，而是工具——工具用不上就该丢弃。

8.7.2 忽略外部依赖的行为变化

现代系统几乎不可能完全脱离外部依赖，比如第三方API、消息队列、数据库、缓存，乃至文件系统，都是家常便饭。你的规范通常会对这些外部依赖的响应格式、返回码、异常行为做出各种假设。例如，有一条规范是：“当调用支付网关的查询接口时，若返回status:'SUCCESS'，则系统应更新订单状态为‘已支付’。”然而，外部系统是独立演进的。假设某一天支付网关升级，将SUCCESS改为了succeeded，或者增加了一个新的错误码。此时，你的规范就变得过时，基于该规范生成的测试（或手动编写的测试）会全部失败。更可怕的是，你可能直到线上出现问题才发现：原来一直在用旧规范进行测试，尽管测试通过了，但实际调用早已发生了偏差。

这个问题的根源其实很清楚：**规范与外部依赖的真实行为之间缺少一个稳定的隔离层**。规范直接“绑定”了外部API的具体实现细节，而外部实现细节是不可控的。

解决方案说起来也不复杂：**适配器+模拟规范**。首先，不要在你的核心业务代码中直接调用外部API，而是创建一个适配器接口。例如，定义一个PaymentGateway接口，其中包含queryStatus方法。适配器的实现类负责与真实的外部API打交道，而你的业务代码只依赖这个适配器接口，而不是具体的第三方SDK或HTTP调用。这样一来，你的规范就不用再直接模拟外部API的全部可能响应，而是定义适配器接口的行为。例如，“当调用PaymentGateway.queryStatus时，它应当返回PAID、PENDING或FAILED中的一种，业务层必须按照状态机规范做出相应处理”。当外部API真的发生变更（比如字段名变化）时，你只需要修改适配器实现类，将新格式转换为规范中定义的枚举值。业务代码的规范完全不需要改动，相应的测试也可以继续进行。只有当你希望业务层支持外部系统的新特性时，才需要更新规范。另外，建议定期对核心外部依赖执行“冒烟测试”，用真实的沙箱环境验证适配器是否仍能正常工作，以便及早发现外部变化，而不是等到用户投诉。

8.7.3 团队成员的抵触（“写规范不如改代码”）

在快节奏的团队里，这类抱怨我听得太多了：“写规范的时间我都能改完两个bug了！”“有这个工夫，我直接写代码、运行代码，不好吗？”这种抵触心理非常普遍，甚至可能引发团队分裂，一部分人坚持规范先行，另一部分人则悄悄绕过流程。最终，规范要么名存实亡，要么变成只有少数人遵守的“仪式”，失去了团队协同的价值。

仔细想想，抵触往往来自3个原因：第一，写规范被认为是一种“额外”的文档工作，与交付业务价值无关；第二，团队缺少自动生成或辅助编写规范的工具，完全手工撰写确实费时；第三，直接改代码能看到立竿见影的效果，而规范的好处（比如减少回归bug、缩短调试时间）需要一段时间才能显现。人们希望获得即时反馈，这无可厚非。

要化解这种抵触，需要从管理、工具和价值3个层面同时入手。在管理上，应当把规范真正纳入为“开发”的一部分。比如，修改团队的工作定义（Definition of Done），只有规范通过测试并合并到仓库后，该任务才算“Done”。在需求拆解阶段，就把“编写/更新规范”作为一个子项，与代码实现共同估点。在工具层面，要想方设法降低写规范的成本。可以录制真实请求-响应流量，自动生成规范初稿，团队只需要人工裁剪和确认；也可以为IDE开发插件，当团队成员修改代码签名时自动提醒甚至自动更新规范文件；还可以选用轻量级、易读的规范语言，避免复杂XML或过于形式化的表达。最重要的是，在价值层面，要让团队成员感受到规范带来的实际好处。我建议选取一个“出血点”来做实验，找到最近两周内回归bug导致的一次线上事故，然后用规范驱动开发的方式重演一遍，证明如果当时有规范测试，这个bug在CI阶段就能得到处理。还可以在两个相似的需求上做对比：一个按传统方式开发，另一个采用SDD，比较调试时间、回归测试通过率和最终缺陷数。当团队中有人因为规范而快速定位了一个隐蔽问题，或者因为规范测试而避免了半夜上线事故时，务必在回顾会上分享这个案例，并公开表扬相关工程师。一个真实的、可见的成功故事，比一百句说教都管用。

8.7.4 规范与代码长期失步

即使是最自律的团队，在项目启动阶段会认真维护规范，CI接入规范测试后会第一时间修复违规问题，也很难长期恪守规范。某个凌晨，线上出现了一个紧急bug，工程师直接修改了代码逻辑，心里想：“先不管规范了，明天再补。”第二天，新的需求接踵而至，“补规范”这件事被无限期推迟。就这样，规范与代码开始分道扬镳。半年后，规范将变成一堆没人敢信的过期文档，甚至连CI中的规范测试都被注释掉了。

这个问题的症结其实很简单：**人的自觉性是不可靠的**。任何依赖人的记忆或责任感的同步机制，在压力和忙碌面前都会失效。因此，必须引入自动化强制同步机制，而不是指望工程师的“下次一定”。

最有效的防线是**在CI中强制运行规范测试**。每次代码提交或PR（Pull Requeat，拉取请求）合并时，CI流水线中必须包含“规范一致性检查”任务。如果规范与当前代码不一致，测试就会失败，CI系统将阻断合并。这里的关键是：决不允许任何例外。不要说“这次情况紧急，先跳过规范检查”，一旦开了口子，口子就会越来越大。如果真的遇到万分紧急的hotfix（热补丁），可以允许事后补规范，但必须在commit message中注明，并由CI系统在24小时后自动创建“补规范”的提醒任务。当规范测试失败时，应该给工程师两个清晰的选择：要么修改代码（如果确实是代码有bug），要么修改规范（如果业务需求已变）。这种二选一的机制能迫使工程师主动面对失步，而不是忽略它。

仅仅靠CI系统阻断还不够，因为工程师可能希望在修改代码后、提交代码前就收到更新规范的提醒。这时可以借助变更追踪钩子：例如，使用Git hooks或文件系统监控工具，检测源代码中与规范相关的部分是否发生了变化。如果检测到变更（比如一个被规范覆盖的函数签名遭到修改），工具会自动弹窗提醒：“您修改了函数calculateDiscount的实现，对应的规范文件可能需要同步更新，是否现在就打开编辑？”更高级的做法是利用AST（Abstract Syntax Tree，抽象语法树）分析，自动比对代码与规范的差异，并尝试生成规范补丁供工程师确认。另外，务必将规范文件与源代码放在同一个代码仓库中，使用相同的分支管理和版本控制策略。这样，任何对规范的修改都会出现在PR中，接受代码审查，没有任何“偷偷修改规范”而不留痕迹的可能。

如果团队觉得一步到位太难，可以采用渐进方式：先在CI中添加**警告模式**（规范测试失败，只报告，不阻断合并），运行一个月后统计失步次数，然后将最常失步的那几个核心规范升级为

强制模式，逐步扩大强制范围，直到全部覆盖。这个过程本身就是一次团队文化的改造——从“规范是可选的”走向“规范是代码的一部分”。

8.8 小结

存量系统并不是规范驱动开发的禁区，相反，它是一个绝佳的练兵场。因为只有在混乱中，你才能真正体会到规范的价值，它能将不可知的系统行为转化为可验证、可演进的资产。

回顾这一章，我们走过了一条从理解到提取、从验证到落地的完整路径。面对存量系统，我们首先承认它的复杂性，并用静态与动态分析的手段去理解它。然后，我们从API、业务规则、数据契约、工作流4个维度，逆向提取出可执行的规范。接着，我们用这些规范验证现有系统，发现隐含的问题，并建立起回归测试基线。最后，我们以增量的方式将Spec-Kit工作流融入日常开发，用规范驱动安全重构和新功能开发。

请记住，规范不是一成不变的文件，而是团队对系统行为的共同理解，是代码的忠实伴侣。当你下次面对一个老旧系统时，不妨先试着为它的一个小角落写一份规范。你会发现，那份规范不仅让你看清了代码，也让你看清了业务的本质。

本章的最后，我想留给你一道思考题：在你当前维护的系统中，哪一个模块是你最想用SDD改造的；如果明天就开始改造，你第一步会做什么？

第9章　从氛围编程到SDD，必然之路与未来图景

在前面的章节中，我们看到了那些因缺乏规范而逐渐失控的项目案例，也初步接触了GitHub Spec-Kit带来的补救思路。但真正的问题在于：为什么那么多团队明知规范的重要性，却依然长期沉溺于“氛围编程”？这是因为氛围编程在特定阶段有着难以抗拒的魅力。本章将首先剖析这种开发方式的五重局限，既不美化也不妖魔化，然后系统性地介绍SDD如何突破每一个局限，最后展望这项实践在AI时代的演进方向，并为你提供一条从下一个PR即可开始的渐进迁移路径。

9.1 氛围编程的局限性：繁荣表象下的暗流

如果你曾经在一个小团队里从零开始搭建一个项目，你一定感受过氛围编程带来的那种畅快。产品经理在聊天窗口里发来一张原型图，你在本地三下五除二写出第一个可运行的端点，推送到仓库。没有文档，没有设计评审，甚至没有一次正式的会议。代码像野草一样生长，每个人都凭直觉决定下一个函数放在哪里、下一个变量叫什么名字。这种开发方式在项目的前几周显得无比高效，因为所有业务知识都装在人的脑子里，而沟通的半径可能只有几米，信息传递几乎没有损耗。

然而，当项目跨越第一个月，或者团队成员增多后，那些曾经使开发十分灵活的东西，会一一变成债务。我们需要正视氛围编程的五重局限，不是为了否定它的价值，而是为了看清它止步于何处。

可维护性黑洞。氛围编程产出的代码，通常只有开发者本人能够顺畅阅读，而且仅限于开发完成后的两周之内。这类代码通常缺少接口契约，缺少状态迁移的显式描述，缺少异常场景的边界定义。代码本身就是唯一的设计文档，当需求发生变化时，开发者往往只能“跟着感觉改”，改完才发现原本没有问题的边缘用例开始崩溃。我见过一个典型项目：最初两人用氛围编程仅花了3周就搭建了原型；6个月后，团队扩张到7人，修复一个简单的排序错误却需要花费两天时间，因为没有人敢确定哪些模块依赖了那个排序结果。代码的可维护性不能寄托于开发者的记忆力，而应基于设计的显式表达。氛围编程恰恰把设计藏在了键盘的敲击声中。

协作的巴别塔。两个开发者如果都凭“氛围”工作，会各自在心里建立起对系统的不同理解。A认为getUser(id)在用户不存在时应该返回null，B认为应该抛出一个自定义异常。他们在各自负责的功能里按照自己的理解编码，直到某一天A调用了B写的函数，程序在生产环境中“爆炸”了。问题不在于谁对谁错，而在于整个团队从来就没有在某个地方记录过这个函数的精确行为。口头约定、聊天记录、代码注释都无法成为可靠的契约。氛围编程默认“只要大家多沟通就能对齐”，但现实是只沟通、不记录，信息的衰减率极高，尤其在异步协作的场景下。周五下午群里的“注意，这个函数会抛异常”，下周三大家可能就想不起来，也找不到了。

质量的无底洞。没有规范，就没有测试的依据。氛围编程下的测试往往只是为了验证“代码目前跑得通”，而不是验证“代码符合预期的行为”。当业务规则发生变化，没有人能准确判断

哪些测试应该被更新。测试覆盖率这个数字变得毫无意义：90%的行覆盖率可能掩盖了关键边界条件从未被思考过的事实。更糟的是，氛围编程鼓励“先写代码，后补测试”，而测试的时候，开发者已经受到了实现细节的锚定效应，很难再独立思考，写出纯粹验证业务规范的测试用例。

可追溯性缺失。一个经典的问题：某行代码为什么要写成这样？氛围编程给出的答案通常是“因为当时需求是那样的”。但当时的需求文档在哪里？需求是怎么演变的？这段判断逻辑对应的是哪个需求？这些信息丢失了。当产品经理要求修改某个特性时，开发者在代码库里找不到其与任何原始需求相关联的痕迹，于是只能全局搜索关键词，凭经验猜测哪些地方需要改动。猜错了，就是生产事故；猜对了，也是侥幸。在合规性要求严格的行业（如金融、医疗、汽车行业），这种追溯能力的缺失意味着程序无法通过审计，甚至无法上线。

规模墙困境。上述所有问题都会在团队规模扩大时指数级恶化。氛围编程团队的极限大约是5人，前提是这些人距离很近，每天有大量的同步时间。一旦团队跨越这个规模，或者变成分布式的协作模式，氛围编程就容易崩盘。这不是人的问题，而是信息传播方式的问题。在规模化的系统里，信息必须被物化——变成文档、变成契约、变成可执行的规范。仅仅靠“谁遇到了问题就去问另一个人”，每个开发者得把30%以上的时间花在沟通上，而不是在创造价值上。

提出这五重局限，不是对氛围编程的否定，而是想冷静地界定它的边界。氛围编程适合探索期、原型期、单人维护的工具脚本。但如果你正在构建一个会存活超过一年、会交给其他人维护、会有多个人同时修改的系统，你就需要越过这道边界。

9.2 SDD：系统性突破每一个局限

SDD不是什么新鲜的概念。早在20年前，形式化方法和契约式设计就已经描绘了它的雏形。但过去之所以没有大规模普及，是因为编写规范的成本太高、工具支持太弱、与主流开发流程的割裂太深。GitHub Spec-Kit的出现，试图改变这3个制约因素，它把规范摆在非常重要的位置，让它与代码库紧密集成，并提供了一套从规范到实现的轻量化工作流。

我们来看SDD如何逐一突破氛围编程的局限的。

用显式规范替代隐性知识。在SDD中，每一个模块、每一个服务接口、每一个重要的业务规则，都有一份对应的规范。这份规范不是Word文档，而是与代码同仓库、同版本、同变更历史的可读文件（例如OpenAPI、Markdown+断言脚本、状态机描述）。新人加入项目，不需要问别人“这个函数是做什么的”，只需要阅读规范目录。当规范足够精确，它甚至可以用于自动生成骨架代码、模拟对象和API文档。在Spec-Kit的工作流中，规范文件的变更需要经过与代码变更同样的评审流程——这意味着任何行为的改变都会留下审计痕迹。

规范作为单一事实源。当开发、测试、产品三方都以同一套规范为准，协作的巴别塔就坍塌了。产品经理写规范（用自然语言描述业务规则），开发者实现规范（写代码满足契约），测试工程师针对规范生成测试用例。三方不再以口方式对齐，而是通过规范文件来对齐。Spec-Kit提供了规范差异的可视化工具：当你修改了一个规范，系统可以自动标记出哪些已有的实现和测试可能需要更新。这种机制把协作从“人追人”变成了“人追规范”。

可执行规范保障质量。SDD最核心的实践之一是“规范即测试”，规范本身应该是可执行的。例如，用Gherkin语言写的Given-When-Then场景可以直接作为集成测试运行；用JSON Schema描述的API请求响应格式可以自动校验mock和真实实现的一致性。当规范可执行，测试就不再是对代码实现的复刻，而是对预期行为的断言。开发者修改实现时，运行规范测试就能立即知道是否破坏了既有行为。更重要的是，规范测试的覆盖率衡量的是“需求空间”的覆盖程度，而不是“代码行”的覆盖程度。Spec-Kit内置了规范测试覆盖率报告，帮助团队发现哪些业务规则从未被验证过。

建立从需求到代码的可追溯链。在Spec-Kit推荐的仓库结构中，每个规范文件可以关联到issue（任务单）或PR中的需求条目，而每个函数或模块可以在注释中声明它满足的规范标识符。结合GitHub的引用关系图，你可以从一行代码追溯到提出这个需求的产品经理、讨论这个设计的PR评论，以及对应的自动化测试。这种追溯能力在审计、重构、离职交接等场景下价值巨大。我曾经帮助一个团队重构一个拥有5年历史的核心模块，依靠Spec-Kit提供的追溯图，他们精准地找出了哪些代码对应哪些活跃需求，最终删除了40%的僵尸代码（那些为已废弃需求服务的实现）。

跨越规模墙。SDD并不排斥快速迭代，它只是把设计决策显式地记录下来。当团队规模扩大到10人、50人，或多团队协作时，规范作为一种可扩展的协调机制，能起到非常重要的作用。A团队修改了一个规范，B团队立即能在CI上看到自己的集成测试失败，因为B团队的代码是基于旧规范编写的。这种失败是一个信号：两个团队需要协调变更。SDD把协调从“开会同步”变成了“规范变更通知+自动化反馈”。在Spec-Kit的企业版方案中，规范仓库可以支持跨组织的订阅和影响分析，进一步减少大规模协作的摩擦。

需要强调的是，SDD并不是要消灭氛围编程。两者可以共存：在探索性强的模块内部，你可以先用氛围编程快速验证想法，一旦验证通过，再补上规范；在稳定接口、核心领域模型和团队边界处，你必须采用SDD。Spec-Kit的设计哲学是“渐进增强”，你可以在一个历史悠久、没有规范的项目里，从一个新增的小模块开始引入规范，慢慢把覆盖率从5%提升到80%。

9.3 展望：SDD的下一站

SDD在今天已经是一个可行的实践，但它远未达到成熟形态。我认为，未来5到10年有5个值得期待的发展方向。它们中的一些已经在Spec-Kit的实验分支中得到探索，另一些则需要整个行业的基础设施演进。

AI与规范的共舞。目前编写规范仍然需要人类思考和表达，这是SDD的主要成本。大语言模型正在改变这一点。你可以给一个AI助手一段自然语言的需求描述，如“用户登录失败3次后锁定账户5分钟”，AI能够生成对应的可执行规范（包含状态迁移、时间约束、边界条件）。更进一步，AI可以扫描遗留代码的行为，反向推导出它实际遵循的规范，然后提示你确认这是有意为之还是偶然。SDD的一个名为“Spec-Assist”的功能已经被提出，它会在PR中自动建议变更规范，当你修改了一段没有关联规范的代码后，它会尝试生成一份候选规范，邀请你在合并前补上。这种“人机协同写规范”的模式会大幅降低SDD的门槛，让氛围编程的快速迭代和SDD的规范性不再非此即彼。

实时规范验证。今天的规范验证主要发生在CI阶段，你推送代码，几分钟后得到反馈。未来，IDE可以做到实时的规范一致性检查。想象一下：你在写一个函数的实现，IDE侧边栏同步显示这个函数需要满足的规范（从规范仓库实时拉取）。当你写出if(user.lockUntil>now)后，IDE高亮提示：“规范要求锁定期间任何尝试登录都应返回特定错误码423，当前实现未满足。”这种即时反馈会像拼写检查一样自然。Spec-Kit的VS Code扩展已经在原型阶段实现了部分能力，但受限于规范语言的表达能力，还不能覆盖所有场景。随着规范语言的标准化（例如在OpenAPI和Rego基础上的扩展），实时验证会越来越实用。

分布式规范仓库。单体仓库的规范管理相对简单，但微服务和多仓库场景下，规范往往散落在各处。未来的SDD会支持分布式的规范发现和依赖解析：服务A声明自己遵循某个版本的规范，服务B可以声明它“要求上游服务A提供规范V2的行为”。当A的规范升级到V3，B的CI会自动拉取变更并运行兼容性检查。这类似于包管理的语义化版本，但作用于行为契约。GitHub正在探索将Spec-Kit与GitHub Packages结合，让规范像库一样被版本化、发布和订阅。一旦成熟，跨团队的规范复用就会像npm install一样简单。

形式化方法的轻量落地。形式化方法（如TLA+、Alloy）可以提供数学级别的正确性保证，但它们的学习曲线陡峭，与日常开发流程脱节。未来的SDD会吸收形式化方法的精髓，但以更轻量的方式呈现它们。例如，你可以为一段关键的状态机逻辑（支付状态、订单生命周期）写出一个有限状态机的规范，SDD自动生成该状态机的模型检查脚本，并在后台定期验证所有已实现的路径是否违反不变式。你不需要学习TLA+的语法，只需要用一种表格或DSL（Domain-Specific Language，领域特定语言）描述状态和迁移。这类工具已经在一些金融科技公司内部使用，我相信未来3年内会集成到主流的Spec-Kit工作流中。

Spec-Kit生态的演进预测。Spec-Kit目前的核心是规范文件的管理和追溯。接下来，我预测它会向以下3个方向扩展。第一，更丰富的规范语言，不仅支持API契约和状态机，还会支持数据一致性约束（如“账户余额不得小于零”）和时序约束（如“事件B必须在事件A之后5秒内发生”）。第二，与GitHub Actions深度集成，提供预置的规范检查工作流（规范覆盖率门禁、向后兼容性检查、规范与代码的变更同步校验）。第三，规范市场，社区可以共享常见的规范（用户认证、分页、限流等）模板，团队可以直接复用，而不是每次重新“发明轮子”。这些演进背后有一个共同的理念：让规范成为开发体验的自然组成部分，而不是一个额外的负担。

9.4 启程建议：如何从你的下一个PR开始拥抱SDD

读完这本书，你可能已经清楚SDD的价值，但面对一个既有项目或者一个紧迫的上线计划，你可能会犹豫：“从哪里开始？”我的建议是：从小处着手，渐进迁移，不要试图一夜之间改造整个系统。

第一步：选择一个低风险的试验田。最适合引入SDD的模块是那些边界清晰（例如一个独立的微服务、一个工具函数库）、变更频率中等（不是每天都在改，但也不是3个月才改一次）、跨团队依赖多（能体现规范对沟通重要价值）的模块。如果你找不到这样的模块，也可以从一个新功能开始——在写第一行代码之前，先用Spec-Kit写一份规范。这个功能完成后，对比一下过去实现类似功能花费的时间和代码的质量，你会发现差异。

第二步：从“单规范驱动”起步。你不需要一开始就把所有接口都规范化。选择一条调用链上的关键契约，例如一个REST API的请求响应格式、一个消息队列的事件schema。为它编写规范，然后让实现和测试都基于这份规范运行。当你习惯了这个流程，再逐步扩大到更多的契约。一个常见的成功模式是：团队规定所有新增的HTTP端点必须有OpenAPI规范，并且CI会检查规范与实现的一致性。这个单一规则执行3个月后，API的集成故障率甚至有可能下降70%。

第三步：利用Spec-Kit的渐进迁移能力。对于遗留代码，你不需要一次性补全所有规范。Spec-Kit支持“规范存根”——你可以为一段代码声明一个待补充的规范ID，系统会跟踪它。每次你修改这段代码，Spec-Kit都会提醒你“这个存根还没有变成真正的规范，请考虑补充”。这能促使你在修改代码后尽愉完善规范。

第四步：建立轻量级的规范评审习惯。不用开会。要求开发者在PR描述中附上“本次变更影响的规范链接”。评审者在审核代码的同时，也需要审核规范变更是否合理、是否与代码一致。Spec-Kit的PR模板可以自动拉取相关规范diff文件，评审者不用离开GitHub页面就能完成双重审查。这个习惯养成后，规范就不会像文档那样腐烂。

最后，我想对担心“规范会拖慢开发速度”的读者说：请亲自试一次。测量从需求到上线的时间，而不是从需求到第一行可运行代码的时间。氛围编程第一天就能让程序运行起来，但第十

天你可能还在修边界bug；采用SDD，或许第三天代码才能运行，但到第十天，它已经稳定运行了7天。选择哪种，取决于你更看重起跑的速度还是到达终点的确定性。

结语　秩序不是创造力的敌人，而是规模化的朋友

回到这本书的起点，我们提到了氛围编程的诱惑，那种代码直接从指尖流出的流畅感，那种远离“繁文缛节”的自由。我不否认这种诱惑的力量，因为我自己也曾经深陷其中。但回过头看，我失败的项目几乎都始于“这次不用规范，我们灵活一点”，而所有存活超过3年、仍然让人愿意维护的项目，无一例外地拥有清晰的规范。

SDD不是要禁锢创造力，而是要保护它。当你不需要记住所有细节、不需要猜测他人的意图、不需要恐惧修改会引发连锁反应时，你才真正有精力去思考那些有创造性的问题，如架构、体验、算法。规范就像是公路的护栏：它不决定你要去哪里，但它确保你不会在加速的时候冲出悬崖。

Spec-Kit是一个把规范带回开发者日常流程的工具。它没有发明新的理论，它只是降低了实践的成本。在这本书里，我们走过了从氛围编程到规范驱动的完整旅程。现在，轮到你打开一个仓库，写下第一份规范，然后看着你的代码在秩序中获得自由。

期待你在GitHub上分享你的Spec-Kit故事。

配套资源验证码 261478