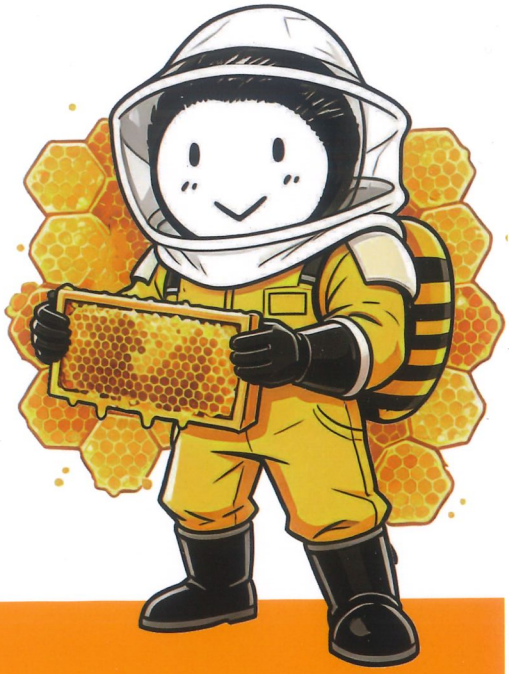
异步图书 www.epubit.com

咖哥AI

SDD 实战

黄佳 著



规范驱动开发之道

AI 时代软件开发方法论
让代码和架构可生成、可审查、可复现

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

前言

胸有丘壑，以终为始

这本书的起点，是一个困扰了我很久的矛盾。

AI 辅助编程工具（如 Claude Code、Cursor、GitHub Copilot、Windsurf）的能力越来越强，每一个都宣称自己能带来“10 倍效率提升”。但我在工程师社区中反复听到的故事是：**第一天效率飙升，第一周开始返工，第一个月便陷入自我怀疑。**

问题出在哪里？

不是 AI 不够聪明。2026 年的大模型已经能通过大多数编程面试，理解复杂的业务逻辑，写出语法正确的代码。问题在于：**你给 AI 的信息不够完整。**

你说完“帮我做一个用户认证系统”，AI 就噼里啪啦写了 2000 行代码。你一看，方向完全偏了——它选了你不想用的框架，加了你不需要的功能，改了你不让它碰的文件。你回退，重新来，写了一大段描述。结果 AI 给出的方向还是偏了，因为你漏说了数据库迁移的事。

每一轮失败都在消耗你的时间和 Token（词元）。核心问题不是 AI 能力不足，而是**你的指令不够完整**。

这就是 SDD 出现的背景。它的核心主张极其简单：

先撰写规范，再编写代码。

Vibe Coding 通常依赖用户通过简短的自然语言指令让 AI 去猜；而 SDD 则要求用户（或在 AI 辅助下）先生成一份结构化的规范文档，内容包括要做什么、为什么做、怎么做以及分几步做，然后让 AI 严格按照规范执行。

SDD 不是什么新发明。

Stephen Covey 在《高效能人士的七个习惯》中提出了一个深刻的洞察：**以终为始（Begin with the End in Mind）——所有事物都经过两次创造。**第一次是在头脑中的心智创造（mental creation），第二次才是在现实中的

物质创造（physical creation），如图 1 所示。建筑师先画蓝图，再盖房子；企业家先写商业计划，再创办公司；作曲家先在脑海中听见旋律，再落笔成谱。

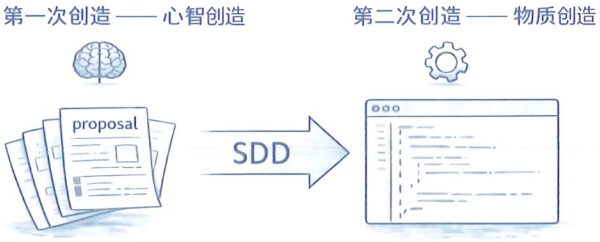


图 1　心智创造与物质创造

如果你跳过第一次创造——不画蓝图就盖房，不写计划就创业——你得到的将不是效率，而是混乱。因为你把“想清楚”的成本转嫁到了“返工”上。

SDD 本质上就是 Covey 的这一洞察在 AI 辅助编程时代的具体实现。

- **proposal.md 是心智创造**：你在头脑中构建“这个系统应该是什么样”。
- **design.md 是架构创造**：你在图纸上规划“这个系统怎么实现”。
- **tasks.md 是执行蓝图**：你在清单上明确“按什么顺序建造”。
- **Code（代码）是物质创造**：AI 按照前三份蓝图，将想法变成可运行的软件。

三份规范构成第一次创造，代码是第二次创造。

Vibe Coding 的问题恰恰在于跳过了第一次创造，直接进入第二次创造——让 AI 凭空建造一栋没有蓝图的房子。房子也许能盖起来，但大概率歪歪扭扭，而且你说不清它为什么长成这样。

这个道理在 AI 出现之前就成立。瀑布模型把“先设计后编码”制度化，敏捷方法又提醒我们不要把设计冻结成厚重的交付物。SDD 吸收了前者“先想清楚”的纪律，同时继承了后者“保持迭代”的灵活性，而不是回到一次性、不可变的瀑布模型。

差异在于，AI 时代的规范不再只是给人看的参考材料，而是给 Agent

执行的输入。你没有写下来的意图，AI 不会知道；你没有明确的约束，AI 不会遵守。规范文档从“可选的说明”变成“必要的工程接口”。

Covey 说得对：所有事物都经过两次创造。SDD 只是让这个道理在 AI 时代不再是可选项。

这本书讲什么

本书系统讲解 SDD 的完整方法论与工程实践。全书共 10 章，按“为什么→怎么做→怎么做好”三个阶段展开。

第一阶段，认识问题（第 1 章和第 2 章）。第 1 章通过一个真实项目的“三天崩塌”故事，揭示 Vibe Coding 的本质缺陷；第 2 章确立 SDD 的核心思想框架——**规范是第一手工件，代码是规范的衍生物。**

第二阶段，掌握方法（第 3 章至第 7 章）。第 3 章梳理 SDD 工具链全景，先介绍 OpenSpec、Spec-Kit、BMAD 等规范框架，再通过四层能力模型说明原子技能、上下文持久化、规范生成框架与完整方法论之间的关系。第 4 章至第 7 章展示实战，分别涉及需求分析与规范撰写（第 4 章）、架构设计与技术决策（第 5 章）、任务拆解与执行编排（第 6 章），以及验证、迭代与演进（第 7 章）。

第三阶段，深入应用（第 8 章至第 10 章）。第 8 章讨论 SDD 与 5 种 Agent 设计模式的结构性融合，并进一步说明 Harness Engineering 如何将规范转化为上下文、工具、权限、验证与反馈回路。第 9 章介绍团队中的 SDD 实践。第 10 章思考 SDD 的边界、未来与哲学意涵。

一个正在达成的共识

2025 年下半年以来，开源社区、AI IDE（Integrated Development Environment，集成开发环境）、代码平台和 Agent 工具团队开始从不同方向趋近同一个判断：当代码生成成本越来越低，真正稀缺的是清晰、可执行、可验证的意图。

有的工具从规范生成切入，有的从项目记忆、技能、子智能体或工作流

编排切入，还有的把重点放在 Harness Engineering 上，即围绕模型构建上下文、工具、权限、验证与反馈回路。名字不同，入口不同，但都在试图回答同一个工程问题：怎样让 AI 在明确的边界内可靠地完成软件开发任务。

本书不会具体介绍某个工具的按钮与命令——那些内容适合放在更新更快的电子书“加餐”里。这里要讨论的是更稳定的部分：规范如何承载意图，如何约束执行，如何让人类判断与 AI 能力形成可复用的协作结构。

本书适合谁阅读

本书适合以下读者阅读。

- 正在使用 AI 辅助编写代码但经常返工的开发者：本书从第 1 章就直面你的痛点，随后用 9 章内容给出系统性解法。
- 想在团队中推广 AI 辅助编程的技术负责人：第 9 章专门介绍团队的 SDD 实践，内容涉及从渐进式引入到度量指标，从规范治理到工具选型。
- 对 Agent 设计模式感兴趣的架构师：第 8 章深入分析 SDD 与 5 种 Agent 设计模式的结构性融合，并说明 Harness Engineering 为何会成为 Agent 工业化的重要方向。
- 已在使用 OpenSpec、Spec-Kit、Kiro、BMAD、Superpowers 等工具，但想理解底层方法论的开发者：本书不是工具教程，而是方法论讲解。工具会更新换代，方法论是稳定的。

在 AI 辅助编程进入工程深水区后，真正稀缺的能力是懂架构、会设计，能把需求、架构与验收边界定义清楚。

proposal.md 回答“要解决什么问题”，design.md 回答“用什么结构解决”，tasks.md 回答“按什么顺序交付”。这三份规范构成了本书的主线。

从 Vibe Coding 到 SDD 的核心就是从“先试试看”转向“先定义清楚再执行”。

这个转变，不会因为某个工具更新而过时。

黄佳

2026 年 5 月

资源与支持

资源获取

本书提供如下资源：

- 书中源代码文件；
- 书中图片文件；
- 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，点击“发表勘误”，输入勘误信息，点击“提交勘误”按钮即可（见下页图）。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。

2.3.5 原则五，代码服务规范 …… 33
2.4 SDD 工作流：6 个阶段 …… 34
2.4.1 阶段一，需求定义→ proposal.md …… 34
2.4.2 阶段二，架构设计→ design.md …… 36
2.4.3 阶段三，接口契约→ api-spec.yaml/data-models.md …… 37
2.4.4 阶段四，任务拆解→ tasks.md …… 38
2.4.5 阶段五，代码生成 …… 39
2.4.6 阶段六，验证与交付 …… 39
2.5 SDD 不是什么：边界与误区 …… 39
本章小结 …… 41
思考题 …… 42

第 3 章 欲善其事：SDD 工具链全景 …… 43
3.1 三大 SDD 开源框架 …… 44
3.1.1 OpenSpec：文档驱动的规范框架 …… 44
3.1.2 Spec-Kit：GitHub 的规范工具包 …… 46
3.1.3 BMAD：角色驱动的多智能体编排 …… 47
3.1.4 四层工具生态：从技能到方法论 …… 50
3.2 AI 辅助编程工具与 SDD 的结合 …… 51
3.2.1 Claude Code：SDD 的天然载体 …… 52
3.2.2 Cursor：规则驱动的上下文管理 …… 55
3.2.3 GitHub Copilot：从 Issue 到代码的规划能力 …… 56
3.2.4 Kiro：把 Spec 流程内置进 IDE …… 58
3.3 规范的载体选择：Markdown、YAML 与 JSON Schema …… 58
3.3.1 Markdown：SDD 第三春的主力 …… 59
3.3.2 YAML：结构化的中间地带 …… 60
3.3.3 JSON Schema：形式化的极端 …… 61
3.4 版本控制与协作：Git 中的规范管理 …… 61
3.4.1 目录结构约定 …… 62
3.4.2 规范审查：规范审查先于代码审查 …… 63
3.4.3 分支策略：规范探索 …… 64
3.5 如何选择适合自己的工具组合 …… 64
本章小结 …… 66
思考题 …… 67

第 4 章 条分缕析：需求分析与规范撰写 …… 68
4.1 从模糊意图到结构化规范 …… 69
4.2 规范的三层结构 …… 71
4.3 实战：智能日报生成器的 proposal.md …… 72
4.4 常见陷阱：过度规范与规范不足 …… 75
4.5 用 Claude Code 辅助编写规范 …… 77
4.6 规范质量检查清单 …… 79
本章小结 …… 79
思考题 …… 80

第 5 章 运筹帷幄：架构设计与技术决策 …… 82
5.1 从 proposal.md 到 design.md：关键的一步跨越 …… 83
5.2 系统架构：先画地图，再行军 …… 84
5.2.1 为什么架构设计如此重要 …… 84
5.2.2 智能日报生成器的架构设计 …… 86
5.2.3 模块职责定义 …… 88
5.3 技术选型：ADR 的艺术 …… 90
5.3.1 为什么需要记录技术决策 …… 90
5.3.2 ADR 的格式 …… 91
5.3.3 智能日报生成器的 ADR 示例 …… 92
5.4 接口契约：模块之间的“握手协议” …… 94
5.4.1 为什么接口比实现更重要 …… 94
5.4.2 数据模型定义 …… 95
5.4.3 模块接口定义 …… 96
5.5 非功能性约束：那些“看不见的需求” …… 98
5.5.1 非功能性需求为什么容易被忽略 …… 98
5.5.2 智能日报生成器的非功能性约束 …… 98
5.6 实战：用 Claude Code 生成架构设计 …… 100
5.6.1 工作流 …… 100
5.6.2 生成过程中的关键注意点 …… 101
5.7 design.md 完整示例 …… 102
本章小结 …… 103
思考题 …… 104

第 6 章　化整为零：任务拆解与执行编排 …… 106
6.1　从 design.md 到 tasks.md：SDD 的第三份规范 …… 107
6.1.1　三份规范的递进关系 …… 107
6.1.2　tasks.md 的基本结构 …… 108
6.2　任务粒度：不大不小的艺术 …… 109
6.2.1　粒度的两个极端 …… 109
6.2.2　黄金法则：一个任务 = 一个可验证的交付物 …… 110
6.3　任务依赖与执行顺序：DAG 思维 …… 110
6.3.1　不是所有任务都要排队 …… 110
6.3.2　智能日报生成器的任务依赖图 …… 111
6.4　验收标准：每个任务的“完成”定义 …… 112
6.4.1　没有验收标准的任务等于没有终点 …… 112
6.4.2　优秀的验收标准长什么样 …… 113
6.5　完整的 tasks.md 示例 …… 114
6.6　实战：用 Claude Code 逐任务执行 …… 120
6.6.1　执行工作流 …… 120
6.6.2　给 AI 的上下文 …… 121
6.6.3　验证与迭代 …… 123
6.7　实战：用子智能体并行执行 …… 123
6.7.1　为什么要并行执行 …… 123
6.7.2　并行执行的安全边界 …… 123
6.7.3　并行执行的工作流 …… 124
本章小结 …… 126
思考题 …… 127

第 7 章　千锤百炼：验证、迭代与演进 …… 128
7.1　规范驱动的测试策略 …… 129
7.1.1　从验收标准到测试用例 …… 129
7.1.2　让 AI 从规范生成测试 …… 130
7.1.3　测试金字塔在 SDD 中的应用 …… 131
7.2　回归验证：代码变了，规范怎么跟着变 …… 133
7.2.1　变更的 4 种类型 …… 133
7.2.2　变更流程：自上而下的涟漪 …… 133

7.2.3 逆向回溯：从代码 bug 发现规范缺陷 …… 135
7.3 规范的版本演进 …… 137
7.3.1 规范的版本控制 …… 137
7.3.2 增量式规范迭代 …… 138
7.4 Brownfield 场景：为存量系统补写规范 …… 139
7.4.1 存量系统的困境 …… 139
7.4.2 逐步收编策略 …… 139
7.4.3 存量系统的规范补写流程 …… 141
7.5 实战：完整的 SDD 迭代循环 …… 142
7.5.1 场景回顾 …… 142
7.5.2 Step 1，更新 proposal.md …… 142
7.5.3 Step 2，更新 design.md …… 142
7.5.4 Step 3，更新 tasks.md …… 143
7.5.5 Step 4，执行与验证 …… 144
7.5.6 Step 5，全量回归测试 …… 145
7.5.7 完整迭代循环图 …… 145
本章小结 …… 147
思考题 …… 148

第 8 章 登堂入室：SDD 与 Agent 设计模式 …… 150
8.1 规划与执行模式：SDD 的天然搭档 …… 151
8.1.1 模式简介 …… 151
8.1.2 SDD 和规划与执行模式的结构映射 …… 152
8.1.3 CLAUDE.md 作为规划上下文 …… 153
8.2 生成—评审模式：规范审查的自动化 …… 154
8.2.1 模式简介 …… 154
8.2.2 SDD 为评审提供客观标准 …… 155
8.2.3 生成—评审模式的实践应用 …… 156
8.3 层级委托模式：规范驱动的多智能体协作 …… 157
8.3.1 模式简介 …… 157
8.3.2 SDD 与层级委托模式的结合 …… 157
8.4 护栏三明治模式：规范作为安全护栏 …… 159
8.4.1 模式简介 …… 159

8.4.2 SDD 规范作为护栏的素材 …… 160
8.4.3 在 Claude Code 中实现护栏 …… 160
8.5 条件路由模式：规范定义的任务分发 …… 162
8.5.1 模式简介 …… 162
8.5.2 SDD 为路由提供分发规则 …… 163
8.6 实战：用 Agent 模式落地 SDD 自动化 …… 164
8.6.1 生成—评审模式：一条指令启动自我审查循环 …… 164
8.6.2 层级委托模式：用子智能体并行执行独立任务 …… 165
8.6.3 护栏三明治模式：用 Hooks 实现自动化规范检查 …… 166
8.6.4 4 种模式之间的组合 …… 169
8.7 5 种模式组合构建 SDD 半自动化流水线 …… 169
8.7.1 从手动到半自动 …… 169
8.7.2 人类的角色变迁 …… 169
8.7.3 不是完全自动化 …… 171
8.8 从 Agent 设计模式到 Harness Engineering …… 172
本章小结 …… 173
思考题 …… 175

第 9 章 众人拾柴：团队中的 SDD 实践 …… 176
9.1 从个人到团队：三个阶段 …… 177
9.1.1 为什么不能一步到位 …… 177
9.1.2 第一阶段的关键：选对试点人和试点项目 …… 177
9.2 规范的协作模式：谁撰写、谁审核、谁执行 …… 179
9.2.1 三种角色 …… 179
9.2.2 规范审查的 PR 流程 …… 180
9.3 规范模板的团队标准化 …… 181
9.3.1 为什么需要统一模板 …… 181
9.3.2 模板的弹性 …… 181
9.4 与现有开发流程的融合 …… 183
9.4.1 SDD+Scrum …… 183
9.4.2 SDD+CI/CD …… 184
9.5 度量与改进：如何衡量 SDD 的效果 …… 186
9.5.1 为什么要度量 …… 186

9.5.2 SDD 效果的度量指标 …… 186
9.6 CI/CD 中的规范守护：让机器帮你盯规范 …… 188
9.6.1 规范审查不能只靠人 …… 188
9.6.2 三个层次的自动化规范检查 …… 188
9.6.3 实战：一个最小的规范 CI 工作流 …… 189
本章小结 …… 191
思考题 …… 192

第 10 章 大道至简：SDD 的未来与思考 …… 194
10.1 SDD 的边界：什么场景适合，什么场景不适合 …… 194
10.1.1 SDD 的甜蜜区 …… 194
10.1.2 不适合 SDD 的场景 …… 195
10.1.3 行业验证：SDD 绝非一家之言 …… 196
10.2 SDD 与其他方法论的融合 …… 197
10.2.1 方法论不是金科玉律 …… 197
10.2.2 SDD 与主流方法论的关系 …… 197
10.3 AI 辅助编程方法论的演进方向 …… 198
10.3.1 从辅助到协作，再到自治 …… 198
10.3.2 规范形态的演进 …… 200
10.4 规范即代码的终极愿景 …… 201
10.4.1 当规范足够精确时 …… 201
10.4.2 我们离终极愿景还有多远 …… 201
10.5 写给开发者的话：在 AI 时代，工程素养重于编码能力 …… 202
10.5.1 能力矩阵的迁移 …… 202
10.5.2 工程素养的回归 …… 203
10.6 从 Vibe Coding 毕业 …… 203
本章小结 …… 205
思考题 …… 206

后记 百川入海，殊途同归 …… 207

第 1 章
空中楼阁：Vibe Coding 的甜蜜与陷阱

没有图纸的装修队施工——干得快，返工更快（见图 1-1）。

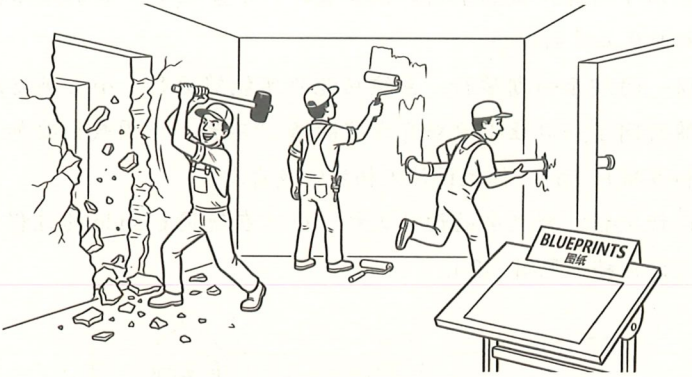


图 1-1　没有图纸的装修队在施工

小冰最近接手了一个紧急项目。

周五下午，客户突然提出一项需求：在三天内开发一个内部知识库搜索工具。该工具不仅要对接飞书文档，还需要支持语义搜索。小雪让小冰加班，并定于下周一向全公司演示。鉴于时间紧、任务重，小冰决定全程利用 Claude Code 来完成这项工作。

第一天的进展异常顺利。小冰仅在 Claude 中描述了需求，Claude 便迅速生成了代码——涵盖 FastAPI 后端、向量数据库接入以及飞书文档 API（Application Programming Interface，应用程序接口）对接等模块。不到 4h，核心功能便可以运行。小冰心中暗喜，深感以往需要耗时一周的工作，如今一天即可完成。

第二天，工作重心转向功能扩展。小冰指挥 Claude 增设用户认证、搜索结果排序和缓存功能。然而，到了下午，问题接踵而至：Claude 生成的认证模块采用了与前期代码截然不同的错误处理风格；搜索排序逻辑与前一天构建的数据结构无法匹配；缓存模块更是引入了项目原本未包含的新依赖。为此，小冰不得不耗费 3h 手动修复这些不一致之处。

第三天正值周日。小冰打开计算机，发现昨天的对话记录找不到了——原来她此前关闭了终端窗口。她试图向 Claude 复述昨天的决策："我们昨天决定采用 Redis 作为缓存方案，而不是本地文件缓存。"Claude 礼貌地回复："好的，我来帮你实现 Redis 缓存。"然而，Claude 生成的代码采用了与昨天截然不同的接口设计。无奈之下，小冰又耗费了 2h 才将新旧代码重新对齐。

周一的演示如期举行，系统功能确实能够运行。小雪满意地点点头，随后问了一句让小冰脊背发凉的话："不错，下周我们要加上权限管理和多租户支持。你把设计文档发给我看看。"

设计文档？哪里来的设计文档？小冰在使用 Claude 的过程中没有撰写设计文档（见图 1-2）。



图 1-2　小冰在使用 Claude 的过程中没有撰写设计文档

小冰翻遍了最近三天的聊天记录，却发现它们散落在七八个独立的对话窗口中，部分窗口甚至因为已经关闭而彻底丢失。她试图回忆当初的技术选型逻辑：为什么选用这个向量数据库而不是那个？为什么认证采用 JWT 而不是 Session？缓存策略又是如何制定的？然而，这些关键决策都是在对话过程中即时生成的，从未被正式记录。

代码还在，但代码背后的“为什么”消失了。

小冰将最近三天的开发经历梳理成一张图（见图 1-3），越看心里越发凉。

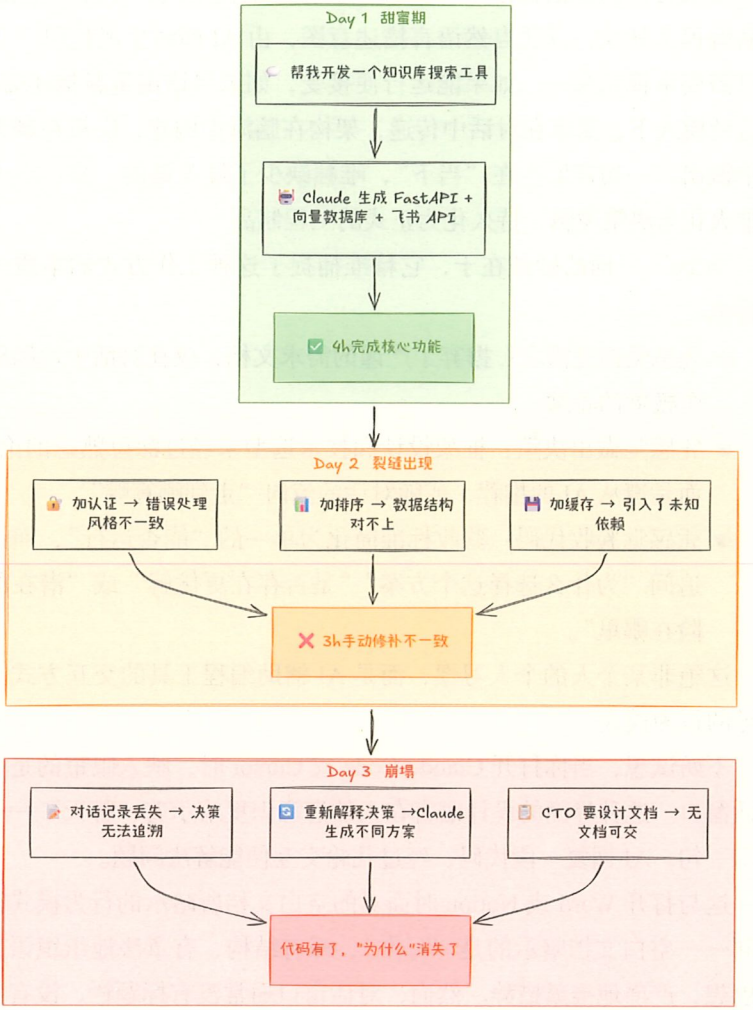


图 1-3　小冰的三天开发经历：从甜蜜期到崩塌

小冰将这段开发经历向咖哥倾诉。咖哥听完沉默数秒，随后缓缓说道：“你这三天所做的事情，有个专门的名称——Vibe Coding。”

1.1 什么是 Vibe Coding

Vibe Coding 这一概念，由 Andrej Karpathy 于 2025 年初提出。他用这一略带自嘲的说法描述了一种日益盛行的编程范式：开发者坐在 AI 辅助编程工具前，仅凭自然语言描述意图，由 AI 即时生成代码，开发者只需要审视结果——如果能运行便接受，如果报错则重新描述需求。在这种模式下，需求在对话中传递，架构在脑海中构建，决策在聊天窗口中做出。一切都发生在“当下”，唯独缺少了最关键的一步：将这些思维火花与决策逻辑，持久化为正式的工程制品。

“Vibe”一词的妙处在于，它精准捕捉了这种工作方式的本质——凭感觉。

- 凭感觉描述需求：摒弃了严谨的需求文档，仅在对话中勾勒出一个模糊的轮廓。
- 凭感觉做出决策：框架设计和技术选型未经过深思熟虑的论证，而是遵从 AI 的推荐，或随对话的流向“走到哪算哪”。
- 凭感觉验收代码：验收标准简化为单一的“能否运行”，而没有追问“为什么选择这个方案”“是否存在更优解”或“潜在的风险在哪里”。

这绝非某个人的个人习惯，而是 AI 辅助编程工具的交互方式天然诱导的行为模式。

不妨试想，当你打开 Claude Code 或 Cursor 时，映入眼帘的是一个对话窗口。这种界面的设计语言在人的潜意识里暗示着“聊天”——你抛出一句，AI 回复一段代码，经过几轮交互便能解决问题。

这与打开 Word 或 Notion 时面对的空白文档所暗示的行为模式截然不同——空白文档暗示的是“写作”，即有结构、有章法地组织语言表述思想，严谨地推敲措辞。然而，对话窗口通常没有标题栏，没有大纲视图，缺乏完善的版本历史管理。它优化的是即时性和流畅性，而非持久性和结构性。

咖哥用一个比喻来阐释这一问题。

“想象你在装修新房，找了一支特别能干的装修队。你说‘我想要一个开放式厨房’，他们二话不说，立刻开始砸墙；你说‘客厅想要暖色调’，油漆师傅马上开始刷漆。效率固然惊人，一天下来，房子已经有模有样了。

“但是，问题很快便出现了。你发现厨房的管道与客厅的地暖冲突，因为没人绘制过全屋的管线图；你发现卧室的开关位置有误，因为没人进行过电路设计；你发现卫生间的下水管道对不上，因为拆墙时没人看过建筑图纸。

“结果如何？干得快，返工更快。最终耗费的时间和成本，往往远超老老实实先画图纸，再按图纸施工的传统模式。”

这就是 Vibe Coding 的本质：**它用即兴对话取代了系统设计和规范。人类给 AI 的反馈是即时的，交互性很强，但往往不是全面的思考。**这种模式在短期内营造出一种“效率惊人”的假象，但长期来看，它实际上是以极高的利息积累一种新型的技术债务。图 1-4 将这一对比呈现得更为直观。

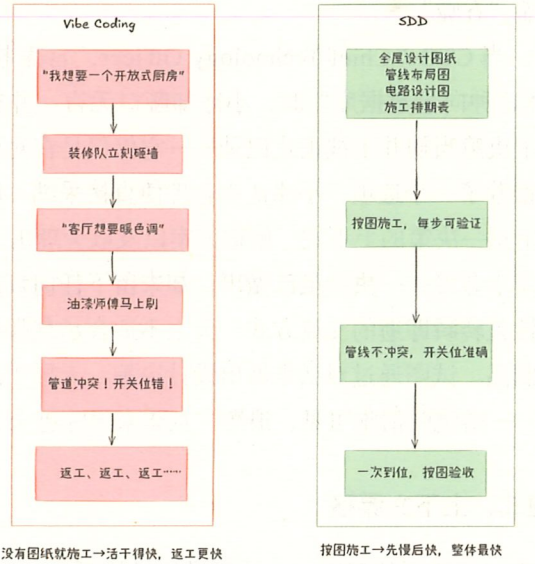


图 1-4　装修比喻：没有图纸的装修队施工与按图施工

1.2 甜蜜的陷阱：5 个致命问题

Vibe Coding 的危险之处，恰恰在于它提供的短期正反馈是真实且强烈的。AI 确实能在几分钟内生成数百行可运行的代码；确实能协助你将一个模糊的想法迅速转化为原型；速度也确实比从零开始手写快了数倍。这种即时的成就感极易令人上瘾——它就像信用卡消费带来的快感，刷卡时消费的爽快是真实的，但还款日的痛苦同样真实且残酷。

技术债务的问题终究会显现，它们通常以 5 个系统性问题的形式呈现。

1.2.1 问题一，需求蒸发

在小冰构建知识库搜索工具的三天里，她至少做出了 20 个决策：选用哪种向量数据库？ API 是否需要分页？搜索结果按什么规则排序？缓存设置多久过期？错误码如何定义？……这些至关重要的决策散落在七八个对话窗口的数百条消息中——有些窗口早已关闭，有些对话已经被上下文压缩“吞噬”。

一周后，当 CTO（Chief Technology Officer，首席技术官）询问“为什么选择这种向量数据库”时，小冰却哑口无言。并非她遗忘了细节，而是那个决策当初并未被正式记录——它仅仅是在对话中自然达成的。Claude 推荐了一个选项，小冰认为合理便直接采纳。因此，没有任何文档记录下这一决策的上下文、候选方案以及取舍理由。

这就是需求蒸发——**决策虽已做出，却未留下任何痕迹**。对话本质上是一种线性且转瞬即逝的交流方式，天生不适合承载那些需要持久化保存的工程制品。试图通过对话来做出设计决策，无异于用铅笔在沙滩上编写合同——落笔时清晰可见，浪潮一旦袭来便踪迹全无。

1.2.2 问题二，上下文漂移

小冰在第二天遇到的问题——认证模块风格不一、数据结构错位，

实则源于一个更根本的限制：AI 的上下文窗口存在容量限制。

尽管 Claude 的上下文窗口容量高达 20 万 Token，看似充裕，实则捉襟见肘。对一个中等规模的项目而言，仅核心代码文件的总量便可能轻易突破这一上限。更重要的是，对话交互本身也在不断消耗上下文空间——你与 Claude 的每一轮问答，都在将早期信息向窗口边缘推挤。一旦对话累积到一定长度，那些重要的早期决策（如“我们决定所有错误处理均采用 Result 类型”），便会不知不觉被截断或压缩，最终导致模型“遗忘”。

其后果便是**上下文漂移**，由于早期关键信息已溢出模型的“工作记忆”，Claude 在对话后半段生成的决策，可能与前半段的决策大相径庭。开发者常常难以察觉这种潜移默化的漂移——因为单独看每一段生成的代码似乎都是逻辑自洽的，但将它们整合在一起时，便会暴露出深层的矛盾与冲突。图 1-5 直观地展示了这一过程。

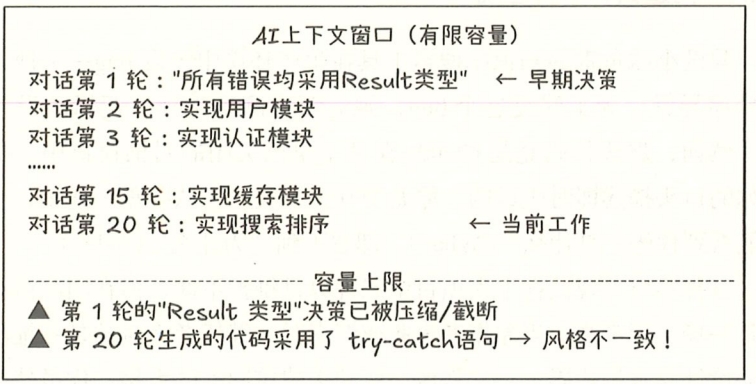


图 1-5　上下文漂移：早期决策被挤出对话窗口后，AI 的后续决策与之矛盾

1.2.3　问题三，不可审查

当 CTO 提出“把设计文档发给我看看”时，他实际上是在向小冰追问一个更深层的问题：**我该如何确信这个系统的设计架构是合理且可靠的？**

在传统软件工程中，代码审查（Code Review）能够行之有效的关键，在于审查员可以洞察完整的设计意图——这得益于拉取请求（Pull Request，PR）描述、设计文档以及架构决策记录（Architecture Decision Record，ADR）的完备支撑。审查员的职责并非局限于验证代码的正确性，还在于判断技术选型的方向是否正确：当前的架构选择是否合理，引入的依赖是否必要，接口设计是否会给未来的扩展埋下隐患。

然而，在 Vibe Coding 模式下，这些设计意图却被禁锢在对话记录之中——前提是这些对话记录尚未因上下文溢出而丢失。审查员面对的是一堆“从天而降”的代码，既不清楚其生成逻辑，也不了解其做过哪些取舍，更难以判断是否充分评估过替代方案。这种信息断层，迫使代码审查退化为纯粹的语法检查，从而丧失其核心价值：对设计决策的质疑与讨论。

1.2.4 问题四，不可复现

假设小冰负责的知识库搜索工具在生产环境中突发 bug——搜索结果排序异常。为了修复这个 bug，她需要追溯当初排序逻辑的设计脉络。然而，那段代码竟是 Claude 在某个早已关闭的对话窗口中，根据小冰的口头描述即时生成的，原始交互上下文已无从寻觅。此刻，小冰只能看到代码“是什么”（What），却看不到“为什么”（Why）。

这种不可复现性在个人项目中或许只是维护不便，但在团队协作中则是一场工程灾难。当原作者离职或调岗时，继任者面对的是一堆没有任何设计背景的代码——它就像一座缺失建筑图纸的大楼：你虽然能看到外墙和房间，却不知道哪面墙是承重墙、哪些管道埋在地下。在这种情况下，任何一处看似微小的改动都可能牵一发而动全身，而开发者却无法预判改动会引发哪些问题。

1.2.5 问题五，不可维护

1.2.1 小节至 1.2.4 小节所讨论的 4 个问题，最终汇聚成一个核心问

题——不可维护。

在软件系统的全生命周期中，维护阶段的成本往往占总成本的 60%~80%。代码一旦交付，便会被无数次阅读、修改、扩展与重构。若编写这段代码时的设计意图未能被记录，每一次修改都将演变为艰难的“数字考古”——维护者被迫从晦涩的代码逻辑中反推原始意图，猜测当初为何做出这样的抉择。这一过程不仅极度耗时，更易导致判断失误。

Vibe Coding 产出的代码，恰似一位缺失完整病历的患者——医生虽能观测当下症状，却对患者的既往病史、用药记录、手术方案等一无所知。在这种信息不对称的情况下，每一次治疗都无异于盲人摸象。

咖哥在白板上勾勒出一张全景图，将这 5 个问题串联成一条因果链，如图 1-6 所示。

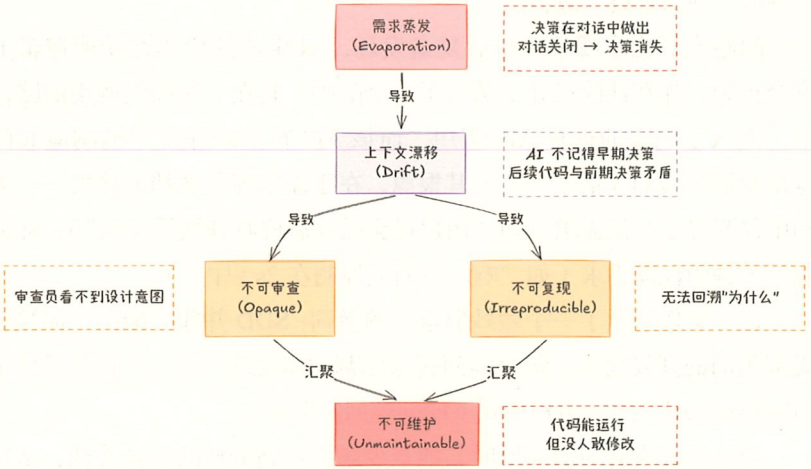


图 1-6　5 个致命问题形成的因果链

小雪听完咖哥对这 5 个致命问题的剖析，陷入短暂沉默。片刻后，她缓缓开口：“这么看来，Vibe Coding 的问题并不在于 AI 生成的代码质量差，而是整个开发过程缺少了一种……结构？”

咖哥指了指她，说道：“你说到了关键词——结构。Vibe Coding 欠缺的从来不是代码生成能力，而是一套严谨的方法论。AI 可以生成漂

亮的代码，但好代码不等于好软件。真正优秀的软件，需要包含清晰的需求、合理的设计、可追溯的决策以及可验证的交付。这些支撑软件生命周期的底层架构，光靠零散的对话是撑不起来的。”

1.3 案例：一个“AI 写的项目”为何没人敢改

小冰的故事并非虚构，类似的情况正在无数开发团队中真实上演。

2025 年下半年，一位开发者在技术社区分享了一段亲身经历。他的同事利用 AI，严格遵循 SDD 模式，为一套存量系统设计了功能迭代方案。该方案是，先撰写需求规范，再进行架构设计，随后拆解任务，最后由 AI 生成代码。方案顺利通过审查，代码开发完成并成功上线。

然而，系统上线后出现了严重问题：用户端操作超过 7s 仍无法获得响应，项目最终“爆雷”。

问题究竟出在哪里？事后复盘发现，AI 生成的代码在功能逻辑上完全正确，却在性能设计上踏入了一个陷阱：它在一条高频调用的路径中，嵌入了同步的远程 API 调用，而该 API 在生产环境中的响应时间竟是测试环境的 5 倍之多。究其根源，在于需求规范文档的缺失——文档中仅明确了功能需求（如“用户提交操作后应收到结果确认”），却遗漏了关键的性能需求（如“响应时间应控制在 2s 以内”）。

这一案例揭示了一个微妙但重要的教训：**SDD 并非万能药，但缺乏规范驱动的开发模式，甚至连问题诊断都举步维艰。**图 1-7 直观对比了 Vibe Coding 与 SDD 两种模式下的问题回溯路径。

当这位开发者回溯问题时，他至少还能找到最初的设计文档，从中发现规范中缺失的性能约束，从而明确改进方向——补充非功能需求规范并加入性能预算。相对地，如果整个开发过程完全依赖 Vibe Coding 模式，那么连这一步都无法实现。因为在这种模式下，“当初为什么这么设计”的关键线索根本无从寻觅。

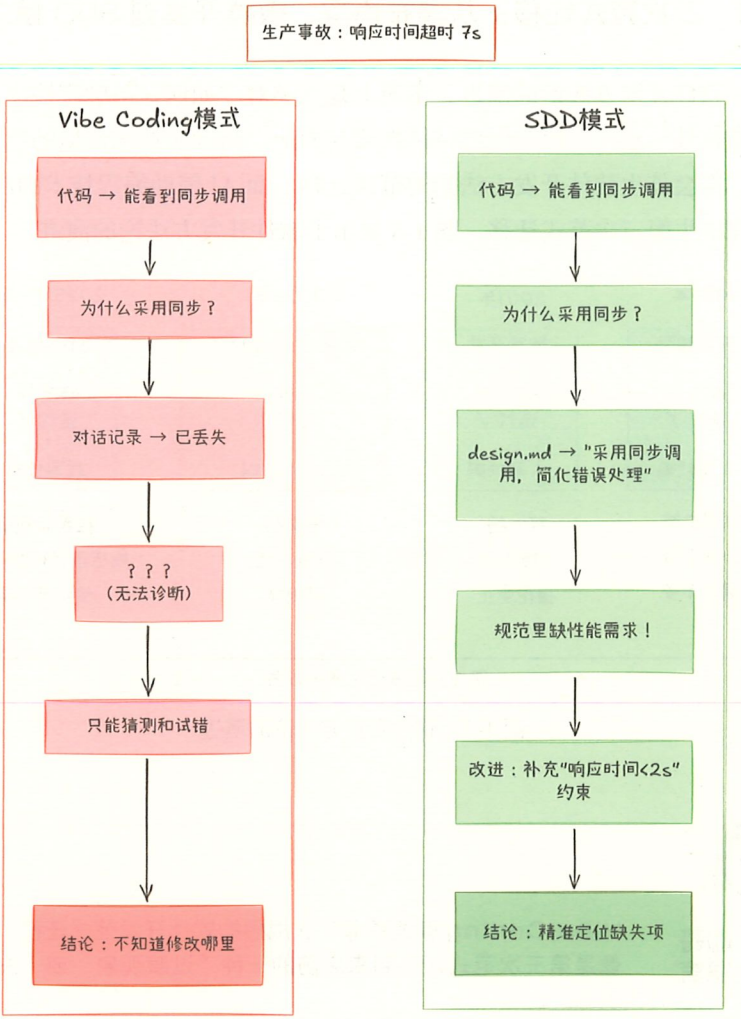


图 1-7　两种模式下的问题回溯对比

听完这个案例，小冰提出了一个好问题："既然 Vibe Coding 存在隐患，那么回归传统瀑布模型，撰写详尽的文档行不行？"

咖哥摇了摇头，说道："行不通。这正是我接下来要阐述的核心——为什么答案既不是 Vibe Coding 模式，也不是瀑布模型，而是一条全新的路径。"

1.4 三次范式迁移：从瀑布模型、敏捷开发到 SDD 模式

软件开发方法论的演进，本质上是人类对“如何在不确定性中交付可靠软件”这一永恒命题的持续探索与回应。每一次技术环境的重大变化，都会催生软件开发方法论的范式迁移。而 AI 辅助编程技术的出现，正在催生第三次范式迁移。图 1-8 展示了软件开发方法论的演进。

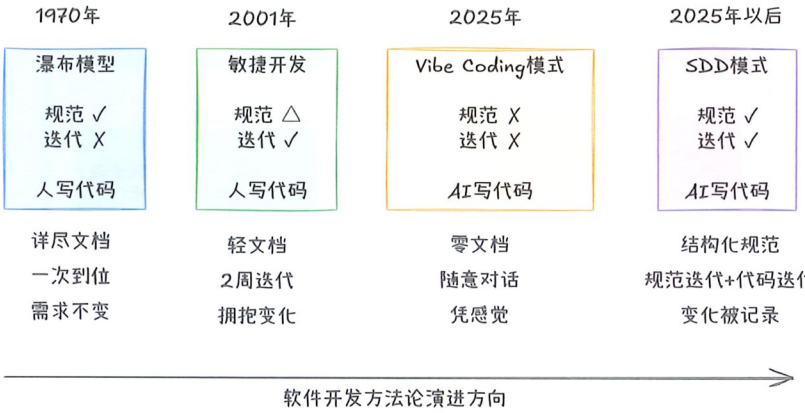


图 1-8 软件开发方法论的演进



咖哥发言

Vibe Coding 模式并非一次成熟的软件开发范式迁移，更像是第三次范式迁移到来之前的一种“过渡现象”或“失衡状态”。

1.4.1 第一代，瀑布模型——有规范，无迭代

1970 年，Winston Royce 提出了著名的“瀑布模型”。该模型的核心思想朴素但不失严谨：先明确要做什么（需求），再规划怎么做（设

计），随后编码实现，最后测试验收。各阶段严格按照顺序执行，前一阶段完成后方能进入下一阶段，就像瀑布一样自上而下、单向流动。

瀑布模型的优点在于其结构清晰，每个阶段均产出明确的交付物——从需求规格说明书、概要设计文档、详细设计文档到测试计划。这些文档构成了一条完整的决策链条，确保了任何人在任何时候都能回溯“当初为什么这样设计”，为软件的可维护性奠定了基础。

然而，瀑布模型存在一个致命缺陷，即它建立在需求不变的假设上。在真实的软件工程中，需求往往具有高度的不确定性，几乎不可能在一开始就完全明确。当团队耗费数月完成代码并交付测试后，如果客户反馈“这不是我想要的”，整个项目便可能需要推倒重来。Winston Royce 后来也承认，纯粹的瀑布模型在实际操作中几乎不可行。

如果以建筑图纸来比喻，瀑布模型就像在动工之前，耗费半年绘制一套极其详尽的图纸——每根管道、每个开关都标注得十分清晰。然而，如果图纸最终交付时，甲方突然改口：“我改变主意了，不要开放式厨房。”那么这张图纸便成了废纸。

1.4.2　第二代，敏捷开发——有迭代，轻规范

2001 年，17 位软件专家在美国犹他州的雪鸟滑雪胜地共同提出了《敏捷宣言》。该宣言的核心主张是：

可工作的软件，胜过详尽的文档。

这一理念深刻改变了软件行业。以 Scrum、XP 和 Kanban 为代表的敏捷方法论，用“短周期迭代”取代了瀑布模型的“一步到位”。在这种模式下，团队通常每两周交付一个可工作的版本，并根据反馈调整改进方向。此外，需求不需要在项目初期就完全明确，而是可以在迭代中逐步澄清。

敏捷开发虽然解决了瀑布模型的核心问题，但也带来了一个新的问题——在“轻文档”理念的号召下，很多团队走向了另一个极端，即完全放弃文档撰写。具体表现为：设计决策仅停留在口头讨论，需求变更

仅在 Jira 中以一句话描述，架构选择处于“大家心照不宣”的状态却从未被正式记录。

这种模式在人类开发者主导的时代尚可勉强运转——毕竟代码由人编写，开发者自己清楚代码背后的逻辑，遇到问题也可面对面直接沟通。然而，当 AI 成为代码的主要生产者之后，这套基于“默契”的协作机制彻底崩塌了。AI 无法参与团队的每日站会，无法参加架构讨论，更不可能了解你上周在白板前做出的决策，它所能掌握的，往往仅限于当前对话窗口内的信息。

如果用建筑装修来比喻，敏捷开发好比边盖边改的小型装修，每周与甲方确认方向，随即开展两周的施工。这种方式效率高，适应性强。但试想一下，如果施工队不是固定班组，而是每天更换一批新工人呢？那些缺乏图纸的口头约定，在人员发生更替时就会全部失效。而目前的 AI，恰恰就是那支“每天都换”的施工队——每次新的对话请求往往都是一个全新的 AI 实例，导致其对之前的决策一无所知。

1.4.3 第三代，SDD 模式——有迭代，有规范

2025 年，随着 AI 辅助编程工具的大规模普及，一种全新的软件开发方法论开始浮现。它并未经历某个正式的“宣言时刻”，而是由众多独立的实践者不约而同地孕育而成。其中既有 OpenSpec 框架的设计者、Spec-Kit 工具的作者、BMAD 方法的提出者，也有无数在与 AI 协作过程中踩过坑的开发者。

这种方法论被称为**规范驱动开发**（Specification-Driven Development，SDD）。该方法论的核心思想可以用一句话概括：**规范是第一手工件，代码是规范的衍生物。**

这句话的每个词语都极具分量。“第一手工件”意味着在 SDD 的工作流中，人类工程师的核心产出并非代码，而是规范，包括需求规范、架构设计文档、接口契约以及任务拆解清单等。这些结构化的文档才是人类智慧的真正载体。“代码是规范的衍生物”意味着代码是依据规范

生成的，恰如建筑是依照图纸建造而成。基于同一份图纸，可以选用不同材料建造房屋；同理，依据同一份规范，也能利用不同的技术栈生成代码。只要图纸不变，房子的“骨架”便始终如一。

SDD 模式并非对瀑布模型的回归。它保留了敏捷开发最重要的“遗产”——迭代与拥抱变化，同时弥补了敏捷开发中常被忽视的一环：将设计决策从非正式的对话记录中提取出来，固化为正式且可版本控制的规范文档。

图 1-9 展示了 SDD 六阶段工作流的全景概览，对各个阶段的详细介绍参见第 2 章。

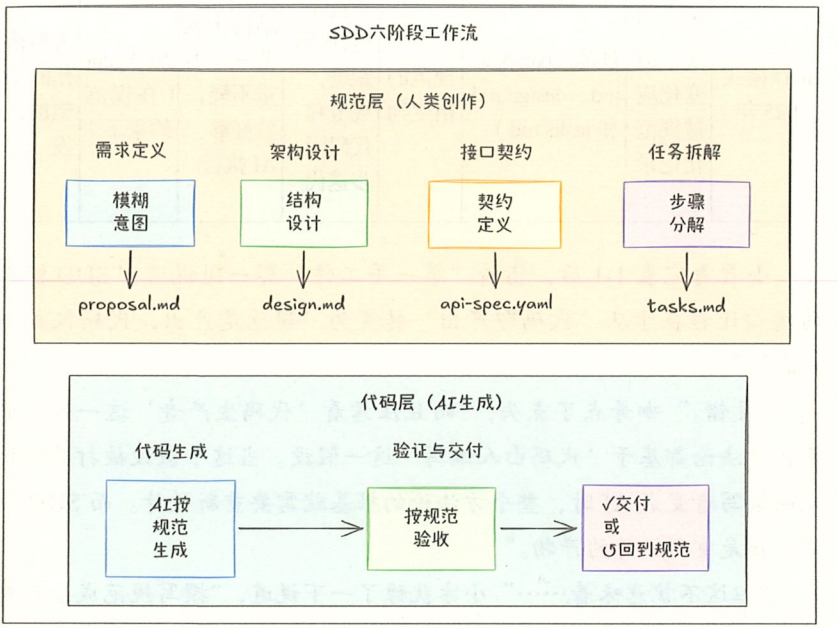


图 1-9　SDD 六阶段工作流全景概览

表 1-1 从多个维度对比了三代软件开发方法论的差异。

表 1-1　三代软件开发方法论的差异

软件开发方法论	核心假设	第一手工件	代码的角色	迭代方式	文档态度	代码生产者	适合场景
瀑布模型（1970 年）	需求可以预先确定	文档（规格说明书）	文档的实现	不迭代，一次到位	重度文档，动辄百页	人类开发者	需求极其稳定的系统
敏捷开发（2001 年）	需求会变，拥抱变化	代码（可工作的软件）	唯一真相来源	2 周一个 Sprint	轻文档，“够用就好”	人类开发者	人类团队的软件开发
SDD 模式（2025 年）	需求会变，但变化应被规范化记录	规范（proposal.md、design.md 和 tasks.md）	规范的衍生物	规范先行，代码跟进，规范和代码同步迭代	结构化规范——不重不轻，恰好够 AI 执行	AI Agent（在规范约束下）	人机协作的AI辅助开发

小雪看完表 1-1 后，指着“第一手工件”那一列说道：“SDD 模式的核心迁移在于从‘代码即产出’转变为‘规范是产出，代码仅是副产品’。”

“没错。”咖哥点了点头，“而且注意看‘代码生产者’这一列，前两代方法论都基于‘代码由人编写’这一假设。当这个假设被打破，代码的编写者变成 AI 时，整个方法论的根基就需要重新设计。而 SDD 模式，正是重新设计的产物。”

“但这不就意味着……”小冰犹豫了一下说道，“撰写规范成了开发者最重要的工作？”

“是的。”咖哥看着她说道，“在 AI 时代，能写出好规范的工程师，比能写出好代码的工程师更稀缺。毕竟，代码可以由 AI 生成，但撰写规范——那个将模糊需求转化为精确、可执行工程制品的过程——目前仍只有人类能够胜任。”

本章小结

本章以小冰接手的一个紧急项目为切入点，深入剖析了当前 AI 辅助编程的主流方式——Vibe Coding——的本质及其潜在代价。

Vibe Coding 并非个别开发者的习惯偏差，而是由对话式 AI 辅助编程工具天然诱导出的行为模式。它虽然能在短期内带来显著的效率提升，但同时在 5 个维度上悄然积累技术债务：需求蒸发（决策痕迹丢失）、上下文漂移（AI 的前后决策不一致）、不可审查（设计意图变得不可见）、不可复现（决策背后的“为什么”无法回溯）、不可维护（修改代码如同拆一座没有图纸的房子）。

究其根本，这 5 个致命问题的症结同源：**用即时对话取代了系统设计与规范。**

图 1-10 展示了主流软件开发方法论与迭代速度 / 开发效率、规范程度之间的关系。

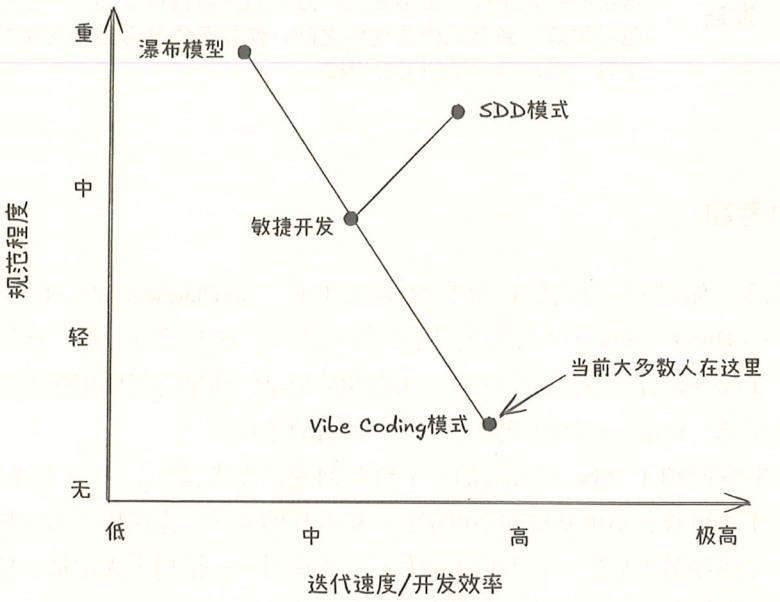


图 1-10　主流软件开发方法论与迭代速度 / 开发效率、规范程度之间的关系

瀑布模型确立了规范，却缺失了迭代；敏捷开发引入了迭代，却轻视了规范；而 Vibe Coding 模式则走向了另一个极端——既没有规范可言，也称不上真正的迭代。SDD 模式试图找到一个平衡点：保留敏捷的迭代节奏，重塑被遗忘的工程规范，同时摒弃烦冗，转而采用更轻量、更结构化的方式。它不再依赖数百页的需求规格说明书，而是通过 proposal.md、design.md、tasks.md 等文档，构建出“恰好足以驱动 AI 执行”的规范。

从第 2 章开始，我们将深入剖析 SDD 的核心思想与工作流，亲眼见证“规范是第一手工件，代码是规范的衍生物”这一理念在实践中的具体呈现。



咖哥发言

编写出代码从来不是目的，交付可靠的软件才是。AI 改变了代码的生产方式，但从未改变可靠软件的标准。软件开发方法论的真正价值，恰恰在生产方式剧变时得以彰显——当旧范式失效、新范式尚未成熟之际，你需要的并非更多花哨的工具，而是更严苛的工程纪律。

思考题

1. 回顾你最近一次使用 AI 辅助编程工具（如 Claude Code、Cursor、GitHub Copilot 等）完成的项目或功能模块：在开发过程中，你做出了哪些设计决策？这些决策是否有相应记录？如果有新同事接手这段代码，他能否追溯并理解这么设计背后的考量？
2. 本章归纳了 Vibe Coding 的 5 个致命问题：需求蒸发、上下文漂移、不可审查、不可复现和不可维护。结合你的实际工作经历思考：哪一个问题最先显现？目前你采用什么方式应对——依赖个人记忆、依赖代码注释，还是已建立起某种文档化实践？

3. 咖哥提出一个观点：在 AI 时代，能写出好规范的工程师，比能写出好代码的工程师更稀缺。你是否认同这一判断？如果认同，这对你个人的职业规划有何启示？如果不认同，你认为 AI 时代最稀缺的工程师应具备何种核心能力？

第 2 章
按图索骥：SDD 的核心思想

图纸是建筑师绘制的，房子是按图纸建造的——SDD 的全部秘密，就在这句话里。

承接第 1 章的见闻，小冰开始有意识地观察身边的同事是如何利用 AI 工具辅助编程的。她很快发现了一个有趣的现象：公司里有两位同事使用 Claude Code 的方式截然不同。

第一位是前端同事阿杰。他的用法和小冰如出一辙——打开终端，描述需求，获取代码，只要“能运行”便大功告成。虽然这种方式让他的初始开发速度极快，但每隔几天，他就不得不耗费半天时间来“救火”——要么是前后端接口无法对齐，要么是页面逻辑与后端返回的数据结构发生冲突，抑或在重构进行到一半时，才发现之前的设计根本无法支撑新的需求。

第二位是后端组长老王。乍看之下，老王的进度似乎慢了一些——每次编码前，他总要先花 20min 撰写一份 Markdown 文件，详尽罗列出本次任务的范围（“做什么”与“不做什么”）、技术方案选型以及接口定义规范。随后，他将这份文档作为上下文输入对话，让 Claude 按这份“规范”生成代码。小冰起初认为老王这是在“浪费时间”，直到她统计了两人的返工率数据：阿杰每周需要返工 3 到 4 次，而老王却几乎不用返工。图 2-1 展示了阿杰与老王使用 AI 工具辅助编程时的不同表现。

“老王写的那个 Markdown 文件，”小冰转向咖哥问道，“就是你之前提到的‘规范’吗？”

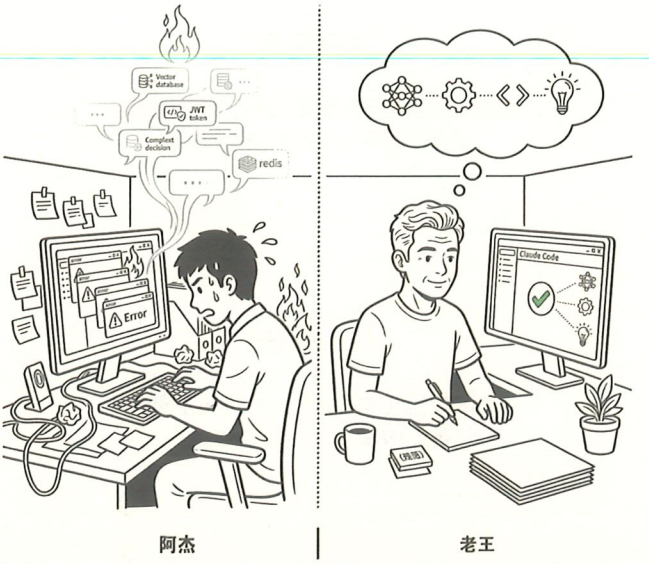


图 2-1　阿杰与老王使用 AI 工具辅助编程时的不同表现

咖哥点了点头："老王自己可能都不知道，他这种基于直觉的操作，实际上已经触及了 SDD 的'灵魂'——先撰写规范，再让 AI 按规范编写代码。在本章中，我们就把他这种朴素的直觉，升维成一套系统化的方法论。"

2.1　一句话定义 SDD

在第 1 章中，我们已经提出了 SDD 的核心主张，现在让我们为它做出更精确的定义。

SDD 是一种 AI 时代的软件开发方法论。它主张在 AI 辅助开发中，人类工程师的核心产出是结构化的规范文档，而代码则是 AI 依据规范生成的衍生物。规范可版本控制、可审查、可追溯，构成项目的"唯一真相来源"（Single Source of Truth）。

这一定义包括 4 个关键要素，让我们借助图 2-2 逐一拆解。

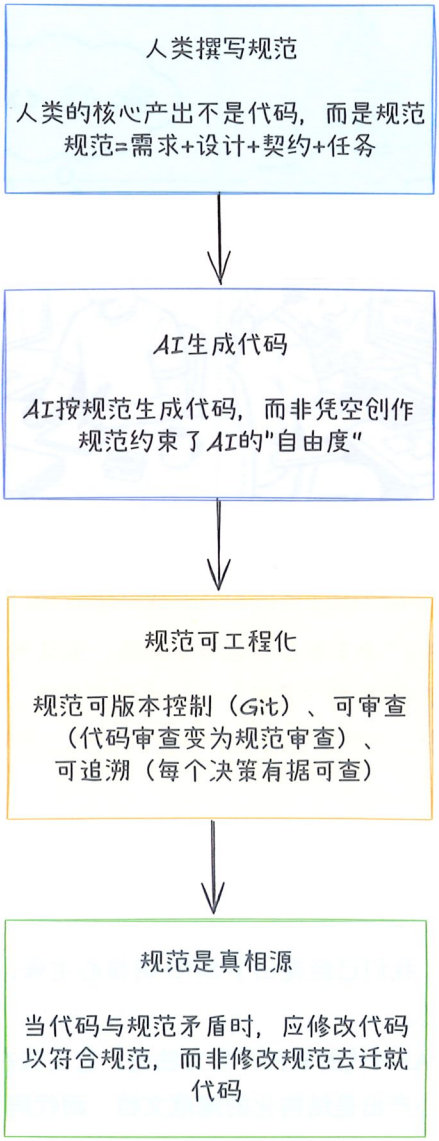


图 2-2　SDD 定义的 4 个关键要素

第四个关键要素值得特别强调，它标志着范式的根本转变。在传统开发模式中，代码是“唯一真相来源”——当文档与代码发生冲突时，

以代码为准，因为代码才是“实际运行的实体”。但在 SDD 中，这一逻辑被彻底重构：**规范是真相源，代码是规范的实例化**。一旦代码与规范不一致，说明代码生成过程出现了偏差。此时的标准操作，应该是依据规范重新生成代码，而不是修改规范去迁就错误的代码。

这就像建筑行业的规则：当施工现场与图纸发生偏差时，工程师的指令永远是“施工有误，需要返工”，而不是“图纸画错了，改图纸去迁就错误的施工”。图纸是设计意图的载体，施工仅是设计意图的实现。实现若出现瑕疵，可以推倒重做，但设计意图不应被错误的实现所歪曲。

小冰提出了一个尖锐的问题：“但如果规范本身就写错了呢？”

“好问题。”咖哥说，“规范当然可能出错，这就像建筑图纸也可能存在设计缺陷。关键在于应对流程：一旦发现错误，我们首先修正的是规范，随后依据更新后的规范重新生成代码。这里的核心在于，规范的每一次变更都必须被记录——修改的原因、具体内容、审批人信息。这正是我们将规范纳入 Git 版本控制的根本原因。回顾第 1 章的 7s 超时的案例，问题的根源恰恰是规范中缺失了性能约束。但正因为有规范，团队才能迅速定位到‘规范缺了什么’，并针对性地补全。试想一下，如果没有规范，你甚至无法判断问题究竟出在设计意图上，还是实现逻辑里。”

2.2　SDD 的历史渊源：三次复兴

SDD 并非凭空诞生的新概念。事实上，“规范先行”的思想在软件工程领域已有近四十年的历史。每隔一段时间，这一思想便会随着技术环境的剧变，以新的面貌重新登场。正如图 2-3 所示，这一思想从 1988 年一直延续至今，每一次复兴都标志着人类试图用更严谨的抽象方法来驾驭日益复杂的软件系统。

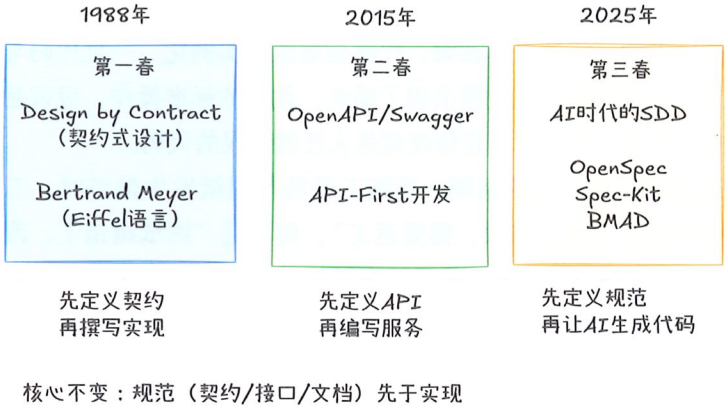


图 2-3　SDD 的三次复兴：规范先行思想的演变

2.2.1　第一春，契约式设计（1988 年）

1988 年，Bertrand Meyer 在设计 Eiffel 编程语言时，提出了契约式设计（Design by Contract）。其核心思想是：每个函数调用本质上都是调用者与被调用者之间签署的一份“契约”——前置条件（调用者承诺输入合法）、后置条件（被调用者承诺输出正确）、不变量（类的一致性约束，在方法调用前后必须为真）。这份契约本身就是函数的“规范”，而代码仅仅是履行这份规范的“实现”。

契约式设计的深刻之处在于，它首次明确地将“规范”和“实现”分离成两个独立的工程制品。在此之前，软件规范往往仅存在于开发者的脑海中，或散落在注释中，既不可执行，也不可验证。Meyer 的突破在于，他让契约变成了代码的一部分——编译器能检查，运行时能验证。

然而，契约式设计也存在其局限：它的粒度是函数级的。虽然我们可以为单个函数定义输入与输出的契约，却很难用同样的方式为整个完整的系统定义“规范”。具体而言，系统级的需求（如用户故事、架构决策及接口设计）往往无法被纳入函数的前置条件和后置条件之中。

2.2.2　第二春，API-First 开发（2015 年）

2015 年，Swagger 规范更名为 **OpenAPI 规范**，为 API-First 开发模式的普及提供了统一标准基础。该模式的核心主张是：优先定义 API 的接口规范（基于 YAML 或 JSON 格式），随后再实现具体的后端服务。

以下是一段 OpenAPI 规范示例。

```
# OpenAPI 规范示例（api-spec.yaml）
openapi: 3.0.0
info:
  title: 知识库搜索 API
  version: 1.0.0
paths:
  /search:
    post:
      summary: 语义搜索文档
      requestBody:
        content:
          application/json:
            schema:
              type: object
              properties:
                query:
                  type: string
                  description: 搜索关键词
                top_k:
                  type: integer
                  default: 10
      responses:
        '200':
          description: 搜索结果
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/SearchResult'
```

上述 YAML 示例就是小冰开发的知识库搜索工具的 API 规范。值得注意的是，其中未包含任何实现代码——它仅描述“接口的形态”，而不关心“后端的具体实现”。基于这份规范，前端团队可以提前开展开发工作（借助 Mock 数据），后端团队可以独立进行服务实现，测试用例可以自动生成，且技术文档天然存在。

OpenAPI 规范成功将“规范先行”的理念从函数级提升到服务级。但它仍然存在局限性：覆盖范围主要在 API 层面，未能覆盖需求分析、架构设计以及任务拆解等更上游的环节。换言之，即便掌握了 API 的具体形态，开发者往往仍无法从中获知“为什么需要该 API”“系统整体架构是什么”以及“该 API 在业务流程中扮演什么角色”等关键信息。

2.2.3 第三春，AI 时代的 SDD（2025 年）

2025 年，“规范先行”迎来了第三次复兴——这一次，其核心驱动力是 AI 辅助编程工具的大规模普及。

与前两次不同，此次 SDD 将“规范”的范围扩展至整个**软件开发生命周期**：从模糊的需求意图转化为结构化的需求文档（proposal.md），从需求推导至架构设计（design.md），从架构细化为接口契约（api-spec.yaml），最终拆解为可执行的任务清单（tasks.md）。在这一范式下，每个阶段都有对应的规范制品，而这些制品均可被 AI 精准理解，并据此生成代码。如表 2-1 所示，这三次复兴的对比清晰地勾勒出软件工程范式的演进轨迹。

表 2-1 SDD 的三次复兴对比

事件	代表	规范粒度	规范格式	规范读者	代码生产者	核心创新
第一春（1988 年）	契约式设计	函数级（前 / 后置条件）	编程语言内置语法	编译器 + 运行时	人类	规范可执行
第二春（2015 年）	OpenAPI/Swagger	服务级（API）	YAML/JSON	代码生成器 +Mock 工具	人类（部分自动生成）	规范跨团队共享

续表

事件	代表	规范粒度	规范格式	规范读者	代码生产者	核心创新
第三春（2025 年）	OpenSpec/ Spec-Kit/ BMAD	系统级（全生命周期）	Markdown（自然语言 + 结构）	AI Agent	AI Agent（主体生成）	规范驱动 AI 全流程

小雪敏锐地捕捉到一个关键变化："第三春采用的规范格式是 Markdown——这本质上是自然语言？这与前两次依赖形式化语言的方式完全不同。"

"没错。"咖哥赞许道，"这恰恰是 AI 时代 SDD 的独特之处。在前两次范式中，规范的'读者'是编译器或工具，因此它们必须遵循严格的格式化语法；而在第三春中，规范的'读者'变成了大模型——它具备理解自然语言的能力。因此，规范不再需要编写成 YAML 或代码，仅需要使用结构化的 Markdown。这极大地降低了撰写规范的门槛——任何能够清晰描述需求的人都能撰写规范，而不需要学习一门新的形式化语言。"

"但这不意味着可以随便撰写。"咖哥话锋一转，"Markdown 格式的规范必须遵循特定的结构和约定，才能确保 AI 准确理解。这正是我们在 2.3 节中要探讨的核心内容——SDD 的五大核心原则。"

2.3　SDD 的五大核心原则

SDD 并非一套僵化的流程，而是一组指导原则。只要遵循这些原则，你完全可以根据项目的具体情况灵活调整具体做法。图 2-4 直观展示了 SDD 的五大核心原则之间的关系。

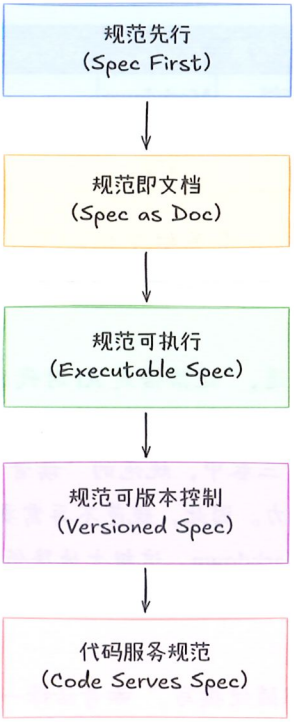


图 2-4　SDD 的五大核心原则及其关系

2.3.1　原则一，规范先行

在编写代码之前，先撰写规范。

这是 SDD 最基础的原则。这一原则并不要求你撰写一份完美的规范。事实上，追求“完美规范”与“瀑布模型”一样不切实际。它只要求你在动手编码之前，投入 15 至 30min，将以下核心要素阐述清晰。

- 做什么（目标）。
- 不做什么（边界）。
- 怎么做（方案）。
- 长什么样（契约）。

- 怎么算做完（验收标准）。

老王凭直觉做的那件事——每次编写代码前用 20min 撰写一份 Markdown 文件——正是“规范先行”理念的最简实践。这一做法不需要依赖任何工具、框架，也不需要专门培训。只要求养成一个习惯：在让 AI 生成代码之前，先告诉它（以及你自己）究竟要做什么。



咖哥发言

“规范先行”绝非教条地要求“编写任意代码前都要先撰写规范”。修复一个小 bug、调整一处样式——这些琐碎改动显然不需要这样做。但凡涉及新功能开发、架构调整、接口设计的工作，都应该先花 15min 厘清思路并写下来。这 15min 不是额外成本，而是能省下后续 3h 返工的投资。

2.3.2　原则二，规范即文档

规范本身就是项目文档，不需要额外撰写文档。

传统开发模式中最令人头疼的问题之一是“文档过时”——需求文档写好了，但代码改了文档没跟着改，几个月后文档和代码完全是两回事。为什么？因为文档和代码是两个独立的工件，需要独立维护。人类的自律是有限的，能偷懒就偷懒。

SDD 从根源上解决了这个问题：**规范既是 AI 的输入，也是人类的文档。**同一份 design.md 文件，AI 通过读取它来生成代码，人类通过读取它来理解系统设计。如果设计变了，需要修改 design.md，然后让 AI 根据新规范重新生成代码——文档和代码永远同步，因为代码就是从文档生成的（见图 2-5）。

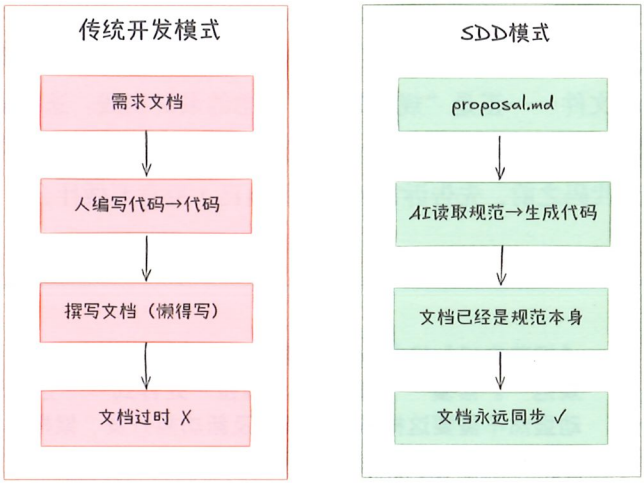


图 2-5　文档同步问题：传统开发模式与 SDD 模式

2.3.3　原则三，规范可执行

规范要精确到 AI 能据此生成代码的程度。

这是区分“坏规范”与“好规范”的核心标准。一份好的 SDD 规范不是散文——它有结构、有约束、有明确的验收标准。它不需要形式化到 YAML 的程度（那是 OpenAPI 的技术路线），但需要精确到一个有经验的 AI Agent 能读懂并执行。

表 2-2 展示了坏规范与好规范的差异。

表 2-2　坏规范与好规范的差异

规范	目标	边界	技术方案	验收标准
坏规范	做一个搜索功能	（没写）	采用向量数据库	能搜就行
好规范	实现 /search POST 接口，接收 query 字符串和 top_k 参数，返回最多 top_k 个结果，按相关性降序排列	本次不做：用户认证、搜索历史、结果缓存	使用 Qdrant 向量数据库，embedding 模型采用 text-embedding-3-small，维度为 1536	1. 空查询返回 400；2. 正常查询返回 200 + JSON 数组；3. 结果按 score 降序；4. 响应时间小于 500ms

坏规范的问题不是“写得太少”，而是“精度不够”——留给 AI 太多需要自行决策的空间。AI 不是不能做决策，而是它做的决策你控制不了、无法追溯。好规范把关键决策固化下来，只把实现细节留给 AI。

小冰问：“那到底该写多详细？会不会写着写着就变成瀑布模型了？”

咖哥回答了一个精辟的标准：“恰好够 AI 执行，一句不多。”

这就是 SDD 模式与瀑布模型的根本区别。瀑布模型的规范是写给不懂业务的开发者看的——需要事无巨细。SDD 模式的规范是写给 AI 看的——AI 有强大的推理和实现能力，你只需要告诉它“做什么”和“关键约束”，对于“怎么实现”，它可以自己搞定。

2.3.4　原则四，规范可版本控制

规范和代码一样放在 Git 里进行管理。

这条原则看似平常，实际上意义深远。把规范放在 Git 里（见图 2-6），意味着可以实现以下事项。

- 每次修改有记录：谁改了什么、什么时候改的、为什么改，全部可追溯。
- 可以做代码审查：准确说是规范审查——在代码审查之前，先审查规范。
- 可以回滚：如果新规范有问题，可以回到上一个版本。
- 可以分支：探索不同方案时，为分支各写一份规范，比较后选择最优方案。

注意最后一次提交——它展示了 SDD 迭代的典型模式：先修改规范，再修改代码。规范变更驱动代码变更，而不是反过来。

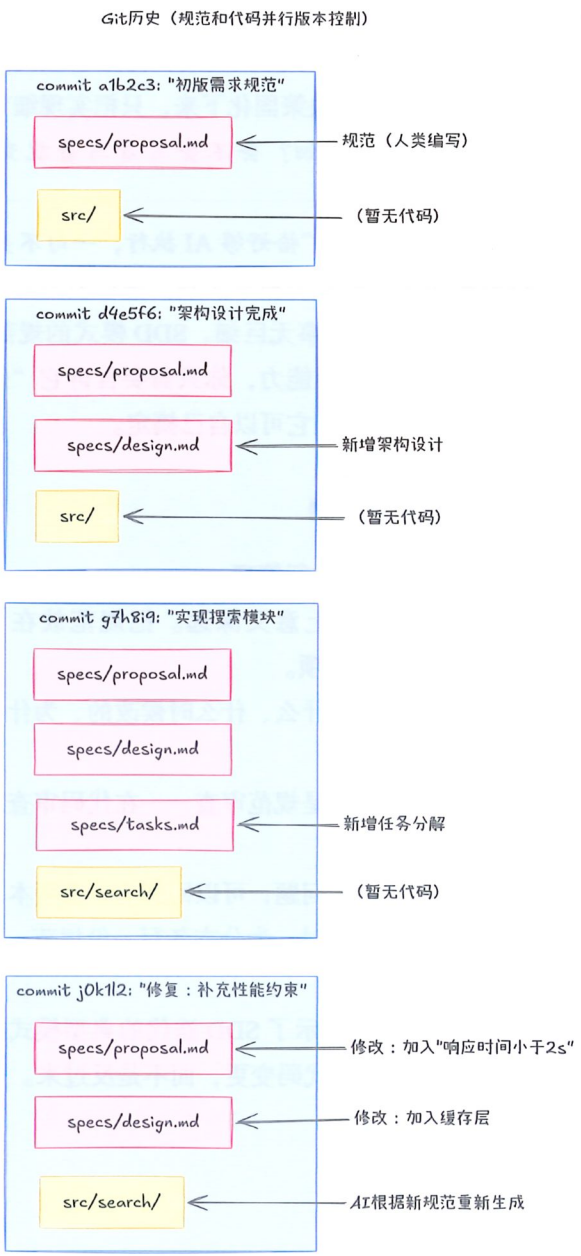


图 2-6　规范与代码在 Git 中的并行版本控制

2.3.5　原则五，代码服务规范

当代码与规范不一致时，以规范为准。

这是 SDD 的“宪法条款”——前 4 个原则都在为这一原则做铺垫。它确立了一个明确的主从关系：规范是主，代码是从（见图 2-7）。

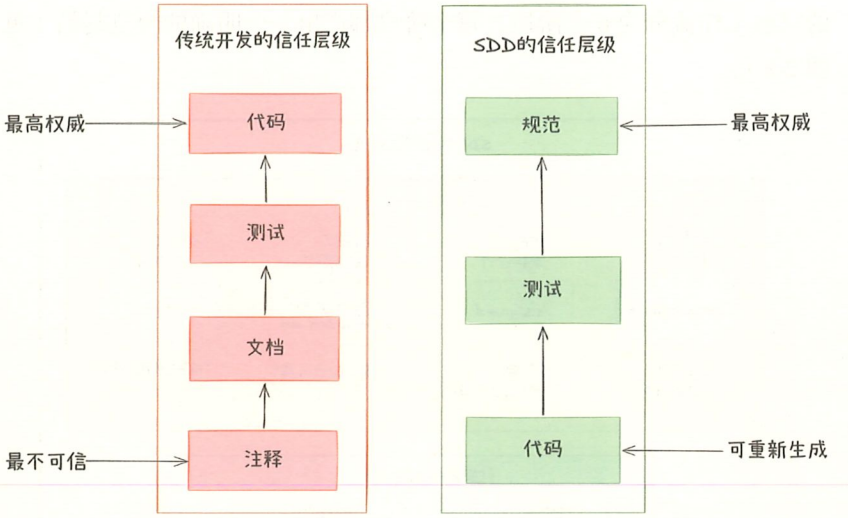


图 2-7　信任层级反转：从“代码为王”到“规范为王”

在传统开发中，“代码即文档”——文档可能过时，但代码永远是真实运行的东西。这个信条在人类编写代码的时代有合理性：代码是人用几小时甚至几天精心编写的，推翻重写成本很高，所以让文档迁就代码是务实的选择。

但在 AI 时代，重新生成代码的成本接近零。让 AI 根据修改后的规范重新生成一个模块，可能只需要几分钟。这彻底改变了项目的成本逻辑——推翻代码重来不再昂贵，而规范中沉淀的设计决策和人类意图才是真正宝贵的、不可再生的。

小雪总结道：“这 5 个原则的逻辑是——先写（规范先行）→一物两用（规范即文档）→写得够精确（规范可执行）→管起来（规范可版本控制）→以它为准（代码服务规范）。”

咖哥笑了："你这个总结比我讲得还清楚。"

2.4 SDD 工作流：6 个阶段

原则确立"为什么"和"是什么"，工作流则解决"怎么做"。SDD 的完整工作流分为 6 个阶段，每个阶段都产出一个明确的规范制品（见图 2-8）。

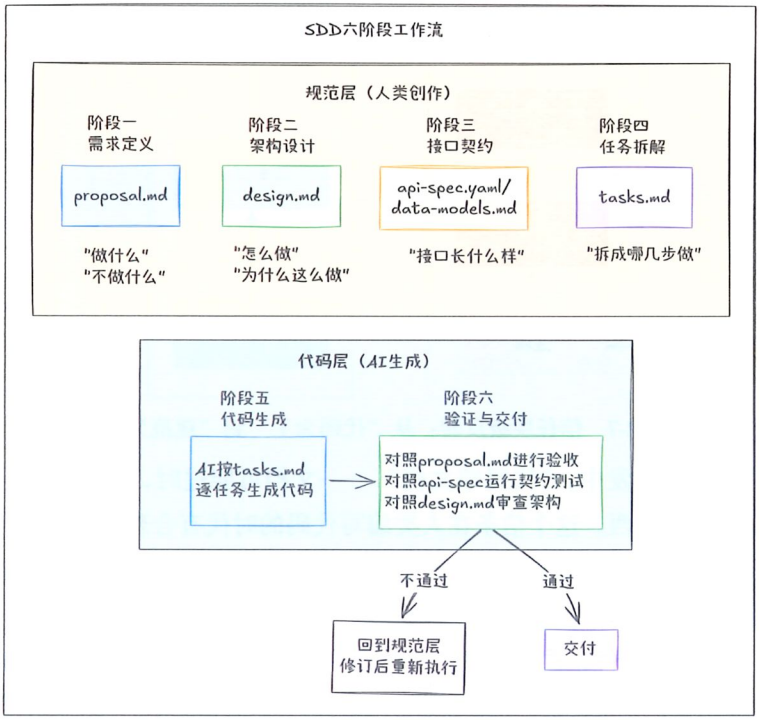


图 2-8　SDD 的六阶段工作流

2.4.1 阶段一，需求定义→ proposal.md

这是整个流程的起点。你需要把脑海中模糊的需求意图转变成一份结构化的需求文档。

一份典型的 proposal.md 如下。

```
# 项目名称：知识库语义搜索工具

## 目标
为公司内部知识库提供语义搜索能力，支持自然语言查询，返回最相关的文档片段

## 背景
目前知识库仅支持关键词搜索，命中率低，员工找不到需要的文档

## 范围
### 做什么
- 对接飞书文档 API，定时同步文档内容
- 文档内容向量化存储
- 提供 REST API，供前端调用
- 基础的搜索结果排序

### 不做什么（本期）
- 用户认证和权限管理（二期）
- 搜索历史和推荐（二期）
- 文档编辑功能（不做）

## 成功标准
- 搜索结果的前三条命中用户意图的比例不低于 80%
- API 响应时间小于 2s
- 支持 10 个以上并发搜索请求

## 技术约束
- 后端使用 Python+FastAPI（团队技术栈）
- 部署在公司内网 Kubernetes 集群
```

注意，“不做什么”这一部分和“做什么”部分同样重要。明确边界可以防止 AI 在生成代码时“好心办坏事”——你没有要求的功能，AI 可能自作主张加上去，然后引入不必要的复杂度。

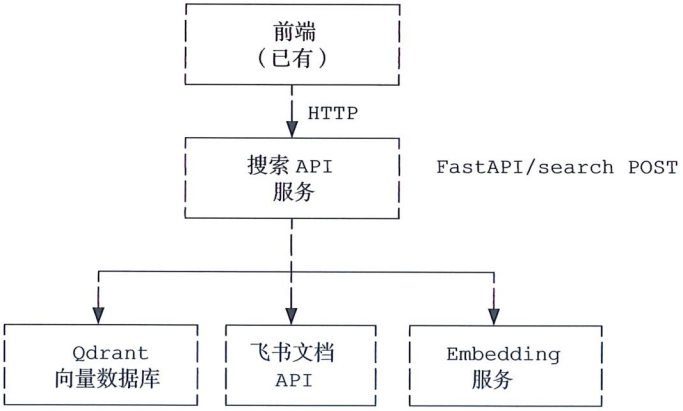
2.4.2 阶段二，架构设计→ design.md

需求明确之后，下一步是设计系统架构。design.md 回答的核心问题是“怎么做”以及“为什么这么做”。

以下是一个架构设计示例。

```
# 架构设计：知识库语义搜索

## 系统架构
```




```
## 技术决策

### ADR-001：向量数据库选型
- 决策：使用 Qdrant
- 候选方案：Qdrant/Milvus/Pinecone
- 选择理由：
  - Qdrant 支持本地部署（公司内网要求）
  - Python SDK 成熟，社区活跃
  - 单机性能满足当前规模（小于 10 万份文档）
- 放弃理由：
  - Pinecone：仅提供云服务，不支持本地部署
  - Milvus：运维复杂度高，当前规模不需要

### ADR-002：Embedding 模型选型
```

```
- 决策：text-embedding-3-small
- 理由：性价比最优，1536 个维度足够
```

设计文档中最有价值的不是架构图本身，而是 ADR。它记录了“为什么选这个方案”和“为什么不选那个方案”。当三个月后有人问“为什么不用 Milvus”时，查阅 ADR 即可找到答案，不需要回忆或猜测。

2.4.3　阶段三，接口契约→ api-spec.yaml/data-models.md

架构确定之后，需要精确定义模块之间的接口。接口契约是 SDD 中精度最高的规范——它必须精确到机器可验证的程度。

2.2 节展示了 OpenAPI 格式的接口规范。除 API 接口以外，数据模型也需要被精确定义。

以下是一个数据模型示例。

```
# 数据模型

## Document（飞书文档）
| 字段              | 类型          | 说明                    |
|-------------------|---------------|-------------------------|
| doc_id            | string        | 飞书文档 ID             |
| title             | string        | 文档标题                |
| content           | string        | 文档正文（纯文本）      |
| updated_at        | datetime      | 最后更新时间            |
| embedding         | float[]       | 1536 维向量             |

## SearchResult（搜索结果）
| 字段              | 类型          | 说明                    |
|-------------------|---------------|-------------------------|
| doc_id            | string        | 文档 ID                 |
| title             | string        | 文档标题                |
| snippet           | string        | 匹配的文本片段（不超过 200 字） |
| score             | float         | 相关性得分 [0, 1]       |
```

2.4.4 阶段四，任务拆解→ tasks.md

这是从规范到代码的关键桥梁。你需要将整个项目拆解成一系列可独立执行、验证的小任务。

以下是一个任务清单示例。

```
# 任务清单

## Task 1：数据模型定义
- 输入：data-models.md
- 产出：src/models.py（Pydantic 模型）
- 验收：模型类定义完整，字段类型匹配数据模型文档
- 依赖：无

## Task 2：飞书文档 API 对接
- 输入：design.md 中的飞书文档 API 部分
- 产出：src/feishu_client.py
- 验收：能获取指定知识库下所有文档的标题和正文
- 依赖：Task 1

## Task 3：向量化与存储
- 输入：ADR-001（Qdrant）、ADR-002（Embedding 模型）
- 产出：src/indexer.py
- 验收：文档内容被正确向量化并存入 Qdrant
- 依赖：Task 1、Task 2

## Task 4：搜索 API 实现
- 输入：api-spec.yaml
- 产出：src/api.py
- 验收：API 行为符合 OpenAPI 规范，所有端点返回正确格式
- 依赖：Task 3

## Task 5：集成测试
- 输入：proposal.md 中的成功标准
- 产出：tests/test_integration.py
- 验收：命中率不低于 80%，响应时间小于 2s，并发不低于 10 个
- 依赖：Task 4
```

每个任务都引用了具体的规范文档作为输入。这确保了 AI 在执行每个任务时都有明确的上下文和约束。不是“帮我写个搜索功能”这类模糊的指令，而是“按照 api-spec.yaml 中定义的接口实现搜索 API”这类精确的任务。

2.4.5　阶段五，代码生成

有了 tasks.md，代码生成便成为一个有序的过程：按任务清单逐个执行，每个任务完成并验证通过后再进入下一个任务。

在这个阶段，AI 的角色是“按图纸施工的工匠”——图纸（规范）已经画好，工匠按照图纸施工。虽然在此阶段仍然需要做出很多实现层面的决策（例如，采用什么库、如何组织代码结构，以及如何处理边界情况），但方向和约束已经被规范锁定。

2.4.6　阶段六，验证与交付

最后一个阶段是验证——SDD 的验证与传统测试存在一个关键区别：**验收标准源于规范，而不是测试工程师的即兴发挥。**

- 功能验收：对照 proposal.md 的成功标准。
- 接口验收：对照 api-spec.yaml 运行契约测试。
- 架构验收：对照 design.md 审查系统结构。
- 性能验收：对照 proposal.md 的性能指标。

如果验证不通过，存在两种可能：要么是代码有问题（需要重新生成），要么是规范有问题（需要修改规范后重新执行）。无论哪种情况，修改都是可追溯的——你知道改了什么、为什么改。

2.5　SDD 不是什么：边界与误区

在结束本章之前，我们有必要明确 SDD 的边界——它不是什么，以及实践中易踩的几个坑。

- **SDD 模式不是瀑布模型。**瀑布模型要求在编写代码之前撰写好所有规范。SDD 模式允许（甚至鼓励）采用增量式规范编写方式。可以先撰写一份粗略的 proposal.md，实现核心功能，再根据反馈修改规范、补充细节，之后实现下一批功能。规范和代码是同步迭代的，不是串行的。
- **SDD 不是形式化方法。**SDD 的规范是结构化的 Markdown 文件，不是 Z 语言或 TLA+。它追求的是“恰好够 AI 执行”的精度，不是数学证明级别的严密性。过度形式化反而会增加撰写规范的成本，导致团队抵触。
- **SDD 不排斥敏捷。**SDD 和 Scrum/Kanban 完全兼容。Sprint 开始，撰写规范；Sprint 中，AI 按规范生成代码并验证；Sprint 结束，回顾规范的质量并改进。SDD 只是给敏捷加了一层“规范纪律”，没有改变迭代的节奏。
- **SDD 不要求每行代码都有对应的规范。**规范覆盖的是关键决策——需求、架构、接口、任务拆解。至于具体的代码实现细节（例如，采用哪个库、如何命名变量、怎么组织文件等），交给 AI 自行决定就好。规范“管”大方向，“不管”小细节。

表 2-3 汇总了这些常见误区与事实澄清。

表 2-3　SDD 常见误区与事实澄清

误区	事实澄清
采用 SDD 模式就是回到瀑布模型	SDD 支持增量迭代，规范和代码同步演进
每个功能都要撰写规范	只有涉及设计决策的工作才需要规范，修复 bug、调整样式不需要
规范写得越详细越好	恰好够 AI 执行即可，过度规范是另一种浪费
实施 SDD 需要特殊工具	一个文本编辑器和 Git 就足够了，框架是可选的
SDD 适合所有场景	探索性原型、一次性脚本不需要 SDD

本章小结

本章从老王的直觉实践出发，系统性地定义了 SDD 的核心概念。

SDD 的一句话定义是：由人类工程师撰写规范，AI 依据规范生成代码，规范是项目的唯一真相来源。4 个关键要素（人类撰写规范、AI 生成代码、规范可工程化、规范是真相源）共同构成了 SDD 定义的基石。

SDD 的历史渊源可追溯至 1988 年 Meyer 的契约式设计。“规范先行”的思想经历了三次重要复兴：从函数级契约（1988 年），到服务级 API 规范（2015 年），再到系统级全生命周期规范（2025 年）。每一次复兴的驱动力，都是技术环境的深刻变革，使“规范先行”重新成为应对复杂性的必要手段。

5 个核心原则（规范先行、规范即文档、规范可执行、规范可版本控制、代码服务规范）共同构成了 SDD 的指导思想。而六阶段工作流（需求定义、架构设计、接口契约、任务拆解、代码生成、验证与交付）则为 SDD 提供了可操作的落地路径。

从第 3 章开始，我们将目光转向工具层面：OpenSpec、Spec-Kit、BMAD 这三个开源框架各自是如何落地 SDD 的；Claude Code、Cursor、GitHub Copilot 等 AI 辅助编程工具，又是如何与 SDD 相结合的。有选择困难症的读者不必着急——第 3 章将为你提供一棵清晰的决策树。



咖哥发言

方法论的价值，不在于其完美，而在于它比“没有方法论”更有效。老王并不知道 SDD 这个名字，但他凭借直觉做到了“规范先行”——结果是，他的返工率比阿杰低了 80%。从今天起，你不需要等到学完所有理论再开始实践。每次让 AI 生成代码之前，先花费 15min 撰写一份 Markdown 文件——这就是 SDD 的第一步。

思考题

1. 请回顾 SDD 的 5 个核心原则，若现阶段只能先执行其中一个（其余 4 个暂时无法兼顾），你会优先选择哪一个？为什么？（提示：不妨思考一下，哪个原则的投入产出比最高。）
2. 本章展示了 proposal.md、design.md、tasks.md 三种规范文档。若项目时间有限，仅允许你撰写其中一种，你会选择哪一种？理由是什么？需要注意的是，该选择可能因项目类型不同而有所差异，如独立开发与团队协作、开发新项目与维护遗留项目，其侧重点往往各不相同。
3. SDD 的第二春（OpenAPI 时代）和第三春（AI 时代）之间，最大的区别在于规范格式：YAML 与 Markdown 的对比。使用结构化但非形式化的 Markdown 格式撰写规范，究竟具备哪些优势，又可能引入哪些风险？如果 AI 的自然语言理解能力持续提升，这一选择是否会随之改变？

第 3 章
欲善其事：SDD 工具链全景

建筑师的工具箱——计算机辅助设计软件虽非设计风格的决定因素，但优质的工具能显著提升设计落地的效率。

小冰已经被之前的理论说服了。规范先行、代码服务规范、六阶段工作流——这套逻辑听起来无懈可击。然而，一旦进入实操环节，小冰立刻撞上了第一堵现实的墙。

“咖哥，我试着按你说的流程走：先写 proposal.md，再写 design.md。结果折腾了一上午，写出来的东西连自己都不忍直视——格式混乱，内容边界模糊，全凭感觉在填。更麻烦的是，写完这些规范后，我完全不知道怎么将它们有效地‘投喂’给 Claude Code，让它真正按规范执行。”

小冰的困惑，精准折射出大多数 SDD 初学者的困境：**方法论已经烂熟于心，但在“如何利用工具落地”这一环节，依然是一头雾水。**

咖哥拉开抽屉，取出一个沉甸甸的工具箱——这正是本章的核心隐喻。

“SDD 方法论好比建筑设计的理论体系——你明白了要绘制平面图、立面图、施工图，但具体使用什么工具落笔？铅笔直尺可以画，AutoCAD 可以画，BIM（Building Information Modeling，建筑信息模型）软件更可以画。工具本身不决定设计质量——手绘的佳作，远胜于 BIM 软件堆砌的平庸之作。但工具直接决定了效率与协作的上限。在本章中，我们将打开软件工程师的工具箱，盘点 2025 年用于 SDD 设计的那些趁手家伙。”

3.1 三大 SDD 开源框架

2025 年，SDD 从零散的个人实践走向框架化。三个开源项目不约而同地出现，各自提供了一套从规范到代码的完整工具链。它们的设计哲学与适用场景各不相同，但解决的核心问题是一致的：如何让“规范先行”从口号变成可操作的工作流。图 3-1 展示了三大 SDD 开源框架。



图 3-1 三大 SDD 开源框架：不同的切入点，相同的目标

3.1.1 OpenSpec：文档驱动的规范框架

OpenSpec 由 Fission AI 团队开发，是目前生态兼容性最广的 SDD 框架。其核心哲学可概括为：规范应独立于平台—— 一份规范可由任意 AI 工具执行。

OpenSpec 的设计源于一个现实问题：2025 年的 AI 辅助编程工具百花齐放 ——Claude Code、Cursor、GitHub Copilot、Windsurf、Bolt、Lovable 等，每个工具都有其独特的上下文管理方式。若规范仅绑定单一工具，一旦切换平台，便需要推倒重来。为此，OpenSpec 采用标准 Markdown 定义规范，通过斜杠命令驱动工作流，使同一份规范能在 20 余种 AI 辅助编程工具中无缝复用。

OpenSpec 工作流如图 3-2 所示。

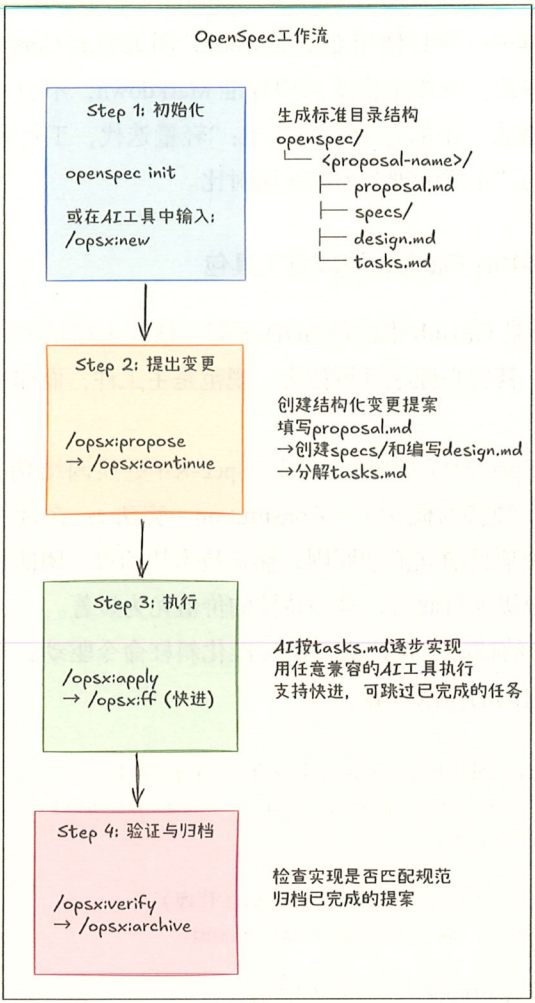


图 3-2　OpenSpec 工作流

OpenSpec 的设计哲学是“流动而非僵化”。它不强制要求走完全部流程——可直接从一个提案（proposal）开始，边实现边补充规范。这种灵活性使其特别适合存量系统（Brownfield）场景——不需要为一个已有项目从头撰写完整规范，而可以在每次变更时使用 /opsx:propose 创

建一个局部提案，逐步为项目积累规范。

OpenSpec 的最大优势在于**轻量与兼容性**。它不绑定任何特定的 AI 辅助编程工具——今日使用 Claude Code，明天换成 Cursor，不需要对规范做任何修改。其规范格式采用标准 Markdown，不引入任何私有语法或元数据格式。正如它的文档所述："轻量迭代，工具灵活"——这与 Spec-Kit 的"正式治理"形成鲜明对比。

3.1.2 Spec-Kit：GitHub 的规范工具包

Spec-Kit 是 GitHub 推出的 SDD 框架，依托全球最大代码托管平台的强力背书。其核心理念可概括为：**规范是主工件，而非用完即弃的脚手架**。

与 OpenSpec 的轻量风格不同，Spec-Kit 更强调结构化与正式性。它引入了一个独特的概念——**Constitution**（宪法）：在撰写任何规范之前，先为整个项目确立治理原则，涵盖技术栈约束、团队标准及合规要求等。对企业级项目而言，这一特性的价值尤为显著。

Spec-Kit 的核心工作流由一组标准化斜杠命令驱动，将软件开发过程拆解为可追溯的离散步骤。

```
# 安装 CLI 工具（基于 Python 生态的 uv 包管理器）
uv tool install specify-cli --from git+https://github.com/github/spec-kit.git

# 初始化 SDD 项目（指定项目名及目标 AI 代理）
specify init my-project --ai claude

# 在 AI 工具中使用斜杠命令驱动工作流
# /speckit.constitution  — 建立项目治理原则（宪法）
# /speckit.specify       — 定义需求和用户故事（需求规范）
# /speckit.plan          — 创建技术实现方案（架构设计）
# /speckit.tasks         — 生成可执行的任务清单（任务拆解）
# /speckit.implement     — 按任务列表逐步实现（代码生成）
```

```
# 质量提高命令
# /speckit.clarify          — 澄清规范中的模糊之处（结构化问答）
# /speckit.analyze          — 跨文档一致性检查（规范与实现对齐）
# /speckit.checklist        — 生成自定义验证清单（合规性审计）
```

Spec-Kit 的最大优势在于其严谨的**正式治理能力**。它不仅是规范模板集合，更是内置了质量门禁的开发框架。例如，其特有的 /speckit.analyze 命令执行跨文档一致性检查，自动发现 proposal.md 中需求与 tasks.md 中任务不匹配的情况。对需要严格审计追踪、合规要求及变更管理的企业级项目来说，这种能力至关重要。

Spec-Kit 支持 Claude Code、GitHub Copilot、Cursor、Gemini、Windsurf 等 AI 辅助编程工具。

然而，Spec-Kit 的劣势也恰恰源于其正式性——对个人或小规模项目而言，从 Constitution（宪法）、Specify（规范）、Plan（计划）、Tasks（任务）到 Implement(实现）的完整流程可能显得过于厚重。正因如此，OpenSpec 在其文档中直言，自身是“轻量替代 Spec-Kit 的方案”。

3.1.3　BMAD：角色驱动的多智能体编排

BMAD（Breakthrough Method for Agile AI-Driven Development）选择了一条截然不同的路径——它不仅是工具，更是通过**多个 AI 角色模拟真实产品团队**的方法论。截至 2026 年年初，BMAD 已演进至第 6 个版本，拥有 34 个内置工作流和 5 个功能模块。

BMAD 的核心创新在于定义了 **12 个以上的 AI Agent 角色**，覆盖了产品团队的完整岗位（见图 3-3）。这里我们选取 7 个核心角色来展示其架构思路。

BMAD 还提供了独特的功能——**Party Mode**（派对模式）：多个 Agent 角色在同一会话中协作，宛如真实产品讨论会。产品经理 Agent 提出需求，架构师 Agent 当场质疑技术可行性，设计师 Agent 则从用户体验角度补充约束——AI 精准模拟了人类团队在讨论中的博弈与磨合过程。

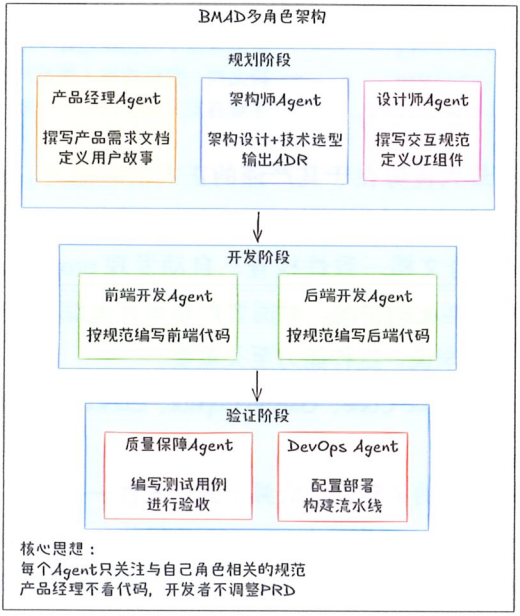


图 3-3　BMAD 多角色架构

BMAD 的深刻之处，在于它解决了单 Agent SDD 模型中一个难以察觉的顽疾——角色污染。当同一个 AI 身兼规范制定者、代码编写者和测试执行者三种角色时，它极易混淆不同角色的关注点——撰写规范时暗自考量实现难度，导致规范向实现妥协；编写代码时擅自修改规范中的约束，导致实现偏离初衷；执行测试时又对自身编写的代码手下留情，导致验证流于形式。

BMAD 通过角色隔离解决了该问题：产品经理 Agent 只关心需求，不知道也不关心后端采用什么技术栈；后端开发 Agent 只按照架构师 Agent 提供的设计文档编写代码，不能修改设计本身；质量保障 Agent 只对照规范执行测试，对代码实现细节一无所知。

小冰听到这里略显兴奋："BMAD 听起来最棒，是不是应该直接采用 BMAD？"

咖哥摇了摇头，说道："选工具不是选最'高级'的，而是选最匹配当前场景的。我们可以通过表 3-1 来对比。"

表 3-1　五大框架对比

框架	开发者	核心哲学	切入角度	规范格式	AI 工具兼容	存量系统支持	安装方式	学习成本	适合场景	团队规模
OpenSpec	Fission AI 团队	轻量迭代，工具无关	斜杠命令 + 文档模板	标准 Markdown	20 余种工具	强（Brownfield-first）	npm install -g	低	个人开发者 / 小型团队，渐进式引入	1~3 人
Spec-Kit	GitHub	规范即主工件，治理优先	CLI+ 斜杠命令	Markdown+ Constitution	18 种工具	强（三种场景）	uv tool install（Python）	中	企业级项目，需要治理和审计	1~10 人
BMAD	BMad Code 公司	AI 即专家协作者	多智能体编排	Markdown+ 角色指令	Claude Code / Cursor	中等	npx bmad-method install	高（12 种角色，34 种工作流）	复杂项目，多 Agent 协作	3~10 人
Superpo-wers	Jesse Vincent	坚持 TDD，强调工程纪律优于灵感	Skill 组合+ 强制前置测试	Markdown+ Skill 模板	原生支持 Claude Code	强（适合补充和完善测试覆盖率）	安装 Claude Code 插件	较低（已掌握 Claude Code 的基本使用）	个人开发者 / 小型团队 / TDD 派	1~5 人
Kiro	Amazon 旗下商业产品	IDE 原生集成，实现从 Spec 到代码生成的端到端自动化	IDE 内置 Spec 流水线	Markdown+ IDE 内置模板	自研 IDE	中等	下载 Kiro IDE	较高（需要迁移至 Kiro IDE）	中大型企业级项目落地	10 人以上

小雪仔细看完表 3-1 后说："我注意到一个有趣的规律——从 OpenSpec 到 Spec-Kit 再到 BMAD，自动化程度越来越高，但学习成本和复杂度也越来越高。"

"这就是工具选择的根本权衡。"咖哥说，"没有银弹。一个人开发独立项目，OpenSpec 的 Markdown 模板就够了——甚至你都不需要使用 OpenSpec，自己构建一个 specs/ 目录，然后编写 Markdown 文件也完全可以。10 个人开发大型项目，BMAD 的多智能体编排才有意义。**关键不在于你使用什么工具，而在于你有没有在编写代码前先撰写规范。**工具是加速器，不是方法论本身。"

3.1.4 四层工具生态：从技能到方法论

如果只看 OpenSpec、Spec-Kit 和 BMAD，读者易误认为 SDD 工具只有"规范框架"这一类。实际上，2025 年下半年以来的工具演化更像一个分层生态：越靠下，越接近可复用的基础能力；越靠上，越接近完整的软件工程方法论。

第一层是原子技能。这类技能通常是一段可复用的指令、一个小型工作流或一种专门的 Agent 能力，用于将"追问需求""执行 TDD（Test Driven Development，测试驱动的开发）""进行代码审查""编写技能"等动作标准化。Superpowers[①] 中的技能、grill-me 这类极简技能，都属于这一层。它们解决的问题不是生成规范全文，而是让 AI 在某个关键动作上更具纪律性。

第二层是上下文持久化。AI 辅助编程的核心困难之一是上下文会漂移：本次对话中确认的约束，下次未必仍存在。CLAUDE.md、AGENTS.md、Kiro Steering、Trellis 中的项目规范和工作区日志，都在将项目规则、任务背景、团队约定沉淀到仓库中，让 Agent 每次启动时都能重新获取同一套高信号上下文。

① 关于 TDD 强制的纪律工具 Superpowers 的更多介绍，请访问 https://kage-ai.com/sdd/#ch3-superpowers。

TYXZJ

第三层是规范生成框架。OpenSpec、Spec-Kit、Kiro Specs 等工具将模糊的想法整理为 requirements → design → tasks 或 proposal → design → tasks 这样的结构化产物。它们的重点是将“我要做什么”转化为可审查、可拆解、可执行的规范。

第四层是完整方法论。BMAD、Superpowers、Compound Engineering 等实践不只关注单份规范的产出，更是将头脑风暴、需求澄清、规划、执行、评审、复盘和知识沉淀串成闭环。它们的核心关切是：如何让每一次开发都为下一次开发降低门槛。

这 4 层并非互斥选项，而是可组合的能力谱系。小团队可以仅用 Markdown 规范配合少量原子技能；大型团队则可能同时需要规范框架、团队记忆、审查 Agent 和复盘机制。选择工具时，不要问“哪个最流行”，而要问“当前瓶颈在哪一层”。

也正因为这些工具迭代很快，本书仅讨论它们所代表的工程概念。具体的安装方式、命令名称、平台限制和最新界面，更适合放在更新频率更高的电子书附录或线上材料中。

3.2　AI 辅助编程工具与 SDD 的结合

框架解决的是“规范怎么写”的问题。但规范写好之后，“怎么把规范喂给 AI 工具并执行”是另一个同样重要的问题。

2025 年以来，Claude Code、Cursor、GitHub Copilot 等 AI 辅助编程工具各自发展出了不同的规范加载机制（见图 3-4）。在本节中，我们将先比较这三类通用工具，随后补充 Kiro 这类将 Spec 流程直接内置到 IDE（Integrated Development Environment，集成开发环境）的代表。它们虽然术语不同、实现方式各异，但本质上做的是同一件事：为 AI 提供一个持久化的上下文锚点，使其在每次对话中都能“看到”项目的核心规范。

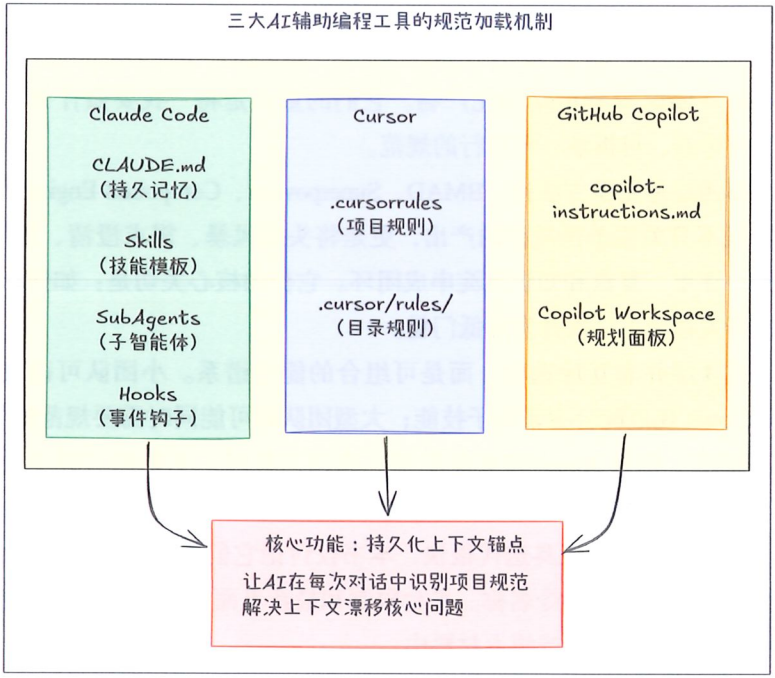


图 3-4　三大 AI 辅助编程工具的规范加载机制对比

3.2.1　Claude Code：SDD 的天然载体

在三大 AI 辅助编程工具中，Claude Code 与 SDD 的契合度最高。这不是因为它“最好”，而是因为其设计机制最完整地覆盖了 SDD 工作流的各个环节。

1. CLAUDE.md——持久记忆

Claude Code 的记忆系统核心是项目根目录下的 CLAUDE.md 文件。该文件在每次对话开始时会自动加载到上下文中，无论对话进行多少轮、上下文如何压缩，CLAUDE.md 的内容始终可见。

在 SDD 实践中，CLAUDE.md 就是你的规范索引文件。请看以下示例。

```
# CLAUDE.md

## 项目概述
知识库语义搜索工具，详见 specs/proposal.md

## 规范文件索引
- 需求规范：specs/proposal.md
- 架构设计：specs/design.md
- API 契约：specs/contracts/api-spec.yaml
- 数据模型：specs/contracts/data-models.md
- 任务清单：specs/tasks.md

## 核心约束（从规范中提取）
- 后端技术栈：Python+FastAPI
- 向量数据库：Qdrant（本地部署）
- 响应时间要求：< 2s
- 所有错误处理使用 Result 类型

## 当前迭代
正在执行 Task 3（向量化与存储），详见 specs/tasks.md
```

请注意最后一部分——“当前迭代”。这解决了 1.2 节介绍的上下文漂移问题。无论 Claude Code 的对话上下文如何滚动，它每次都能从 CLAUDE.md 中获取当前应执行的任务及需要遵循的规范。规范不再在对话流中漂浮，而是被锚定在一个始终可见的位置。

2. Skills——规范模板

Claude Code 的 Skills 系统允许你定义可复用的技能包——用 Markdown 文件描述一类任务的标准流程和约束。在 SDD 场景下，Skills 天然适合封装规范编写模板。请看以下示例。

```
# specs-writing（规范编写技能）

你是一位 SDD 规范编写助手。当用户要求编写或修改规范时，
遵循以下结构：

## proposal.md 模板
- 目标：必填，一句话描述
```

```
- 背景：必填，为什么要做
- 范围：必填，做什么 + 不做什么
- 成功标准：必填，可量化的验收条件
- 技术约束：选填，团队的技术栈和限制

## design.md 模板
- 系统架构：必填，组件关系图
- 技术决策：每个关键决策一条
  - 决策内容
  - 候选方案
  - 选择理由
  - 放弃理由
```

有了这个 Skill，每次在 Claude Code 中输入 Prompt“帮我撰写需求规范”时，它都会按照统一的模板来执行——不会因为不同的对话、不同的心情而生成格式完全不同的规范文档。这解决了上下文漂移的一个变种——**格式漂移**。

3. SubAgents（子智能体）——并行执行

当 tasks.md 中包含多个独立的任务时（例如，Task 1 和 Task 2 没有依赖关系），Claude Code 的子智能体可以并行执行它们——每个子智能体负责一个任务，各自在隔离的上下文中工作，互不干扰。

这类似于建筑工地上多个施工队同时作业：电工走线、水暖工铺管、木工搭建框架——这些工种可以并行进行，因为工人各自按照不同的施工图施工。子智能体的“施工图”，就是 tasks.md 中各自对应的任务规范。

4. Hooks（事件钩子）——自动验证

Claude Code 的 Hooks 机制允许你在特定事件发生时自动触发检查。在 SDD 场景下，一个典型的用法是：每次 Claude Code 完成代码生成后，自动运行测试。请看以下示例。

```
{
  "hooks": {
    "PostToolUse": [{
      "matcher": "Write|Edit",
```

```
        "hooks": [{
          "type": "command",
          "command": "python -m pytest tests/ -q"
        }]
      }]
    }
  }
```

这意味着 AI 每编写一段代码，测试就会自动执行一次。如果测试不通过，Claude Code 会获取失败结果并尝试修复，进而形成“生成→验证→修复”的自动循环。这对应了 SDD 第六阶段（验证与交付）的自动化实践。

3.2.2　Cursor：规则驱动的上下文管理

Cursor 通过两种机制支持 SDD：项目根目录下的 .cursorrules 文件和 .cursor/rules/ 目录下的分模块规则。

.cursorrules 的功能类似于 CLAUDE.md——它在每次与 AI 交互时都会自动加载，作为全局上下文。你可以在其中存储项目规范的索引和核心约束。请看以下示例。

```
# .cursorrules

## 项目规范
本项目遵循 SDD 方法论
所有功能开发必须先有对应的规范文档

## 规范文件位置
- specs/proposal.md — 需求规范
- specs/design.md — 架构设计
- specs/contracts/ — 接口契约

## 编码规范（从 design.md 提取）
- 后端：Python 3.11 及更高版本，FastAPI，异步优先
- 数据库：Qdrant（向量存储）
- 错误处理：统一使用 Result 类型
- 测试：每个模块必须有单元测试
```

.cursor/rules/ 目录则更进一步——它允许你为项目的不同部分定义不同的规则。例如，rules/backend.md 包含后端特有的约束，rules/testing.md 包含测试标准。当 Cursor 在对应目录下工作时，会自动加载对应的规则。

Cursor 的规则系统比 Claude Code 的 CLAUDE.md 更轻量、更简单——它没有 Skills、SubAgents 和 Hooks 这些高级机制。对于需要完整 SDD 工作流自动化的场景，Cursor 需要搭配外部工具来补齐。

3.2.3 GitHub Copilot：从 Issue 到代码的规划能力

GitHub Copilot 的 SDD 支持体现在三个层面。

- github/copilot-instructions.md 文件——功能类似于 CLAUDE.md 和 .cursorrules，在 Copilot Chat 中自动作为上下文加载。用户可以在其中放入项目的核心规范约束。GitHub 企业版还支持组织级别的统一指令——信息技术部门可以为整个组织定义技术标准，并自动应用到每个仓库。
- Prompt 文件（.prompt.md）——这是 GitHub Copilot 的一个有趣设计：用户可以在项目中放置 .prompt.md 文件，作为可复用的 Prompt 模板，在 Copilot Chat 中像斜杠命令一样调用。这类似于 Claude Code 的 Skills，但更轻量。用户可以用它封装规范编写模板、代码审查清单等 SDD 常用流程。
- Copilot Workspace——它提供了一个从 GitHub Issue 到代码实现的规划面板：用户在 Issue 中描述需求，Copilot Workspace 自动生成一个实现计划（需要修改哪些文件、怎么修改）；用户审核计划后，它再生成代码。

GitHub Copilot 提供的工作流与 SDD 包含一种隐含的对应关系：Issue 类似于 proposal.md（需求描述），实现计划类似于 tasks.md（任务拆解）。值得一提的是，Spec-Kit 作为 GitHub 官方项目，与 GitHub Copilot 的集成最为紧密——specify init --ai copilot 会自动配置 GitHub

Copilot 专属的规范上下文。但 GitHub Copilot 目前还没有显式支持 design.md（架构设计）和接口契约——它更适合小粒度的功能迭代，而不是从零开始的系统设计。表 3-2 汇总了三大 AI 辅助编程工具的 SDD 支持能力。

表 3-2　三大 AI 辅助编程工具的 SDD 支持能力对比

AI 辅助编程工具	规范存储与索引	规范模板	并行任务执行	自动化验证	动态上下文注入	跨会话记忆	组织级治理	SDD 完整度
Claude Code	CLAUDE.md（自动加载）	Skills 系统	Sub Agents	Hooks	Skills 中 !command``	自动记忆系统	Managed CLAUDE.md	★★★★★
Cursor	.cursorrules+.cursor/rules/	无内置（可用规则模拟）	无内置	无内置	无内置	无内置	无内置	★★★☆☆
GitHub Copilot	copilot-instructions.md	.prompt.md 提示文件	无内置	GitHub Actions 集成	无内置	无内置	企业级统一指令	★★★☆☆

小冰看完表 3-2 后问："所以，Claude Code 是进行 SDD 设计的最佳选择？"

"在当前时间点，Claude Code 的机制最完整。"咖哥说，"但这并非一个非此即彼的选择。很多开发者同时使用多种工具——用 Claude Code 做架构设计和代码生成，用 Cursor 做日常编辑，用 GitHub Copilot 做代码补全，用 Kiro 管理规范流。关键在于，规范文件本身是 Markdown 格式——存储在 Git 仓库中，任何工具都能读取。规范的价值不取决于你使用什么工具，正如图纸的价值不取决于你是用铅笔还是用 AutoCAD 软件画的。"

3.2.4 Kiro：把 Spec 流程内置进 IDE

Kiro 的切入点与上述工具不同。Claude Code、Cursor、GitHub Copilot 更多是将规范作为上下文加载给 AI；Kiro 则将 Specs、Steering 和 Hooks 作为 IDE 中的“一等对象”。Specs 将需求、设计、任务拆解为可追踪的开发单元，Steering 沉淀项目的长期规则，Hooks 则在文件变更或工作流事件中触发自动化检查。

因此，Kiro 代表的是另一种趋势：SDD 不只是仓库里的几份 Markdown 文件，而是被 IDE 工作台直接承载的开发流程。它降低了新手进入规范驱动开发的门槛，同时也提醒我们：无论界面如何变化，真正稳定的仍是需求、设计、任务、验证这条主线。

本书不会展开 Kiro 或任何具体 IDE 的操作步骤。按钮、菜单和命令随时可能变化，更适合以电子书形式持续更新；此处仅保留概念框架，帮助你在不同工具之间识别同一种 SDD 能力。

3.3 规范的载体选择：Markdown、YAML 与 JSON Schema

框架已选定，工具已确定，还有一个容易被忽略的选择：规范使用什么格式来撰写？

这个选择在前两次 SDD 复兴中是确定的——契约式设计采用编程语言内置语法，OpenAPI 使用 YAML 或 JSON Schema。但在 AI 时代，情况变得更为复杂：规范的“读者”不再只是编译器或代码生成器，还包括大模型和人类。如图 3-5 所示，三种主流格式各有利弊。

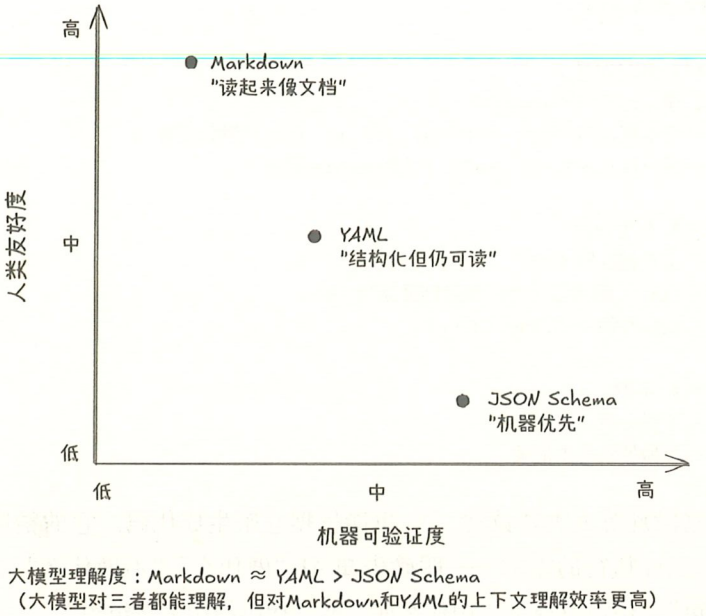


图 3-5　三种主流格式的对比

3.3.1　Markdown：SDD 第三春的主力

Markdown 是 AI 时代 SDD 规范的默认选择，原因很简单：**大模型对 Markdown 的理解能力极强**。大模型在训练数据中见过海量的 Markdown 文件——GitHub 上几乎所有的 README、技术文档、设计文档都采用了 Markdown 格式。用 Markdown 格式撰写规范，AI 几乎不存在“格式理解”障碍。

更重要的是，Markdown 对人类也极其友好。开发者不需要学习任何新语法，会撰写 README 就会撰写规范。标题、列表、表格、代码块等元素足以表达从需求到架构再到任务拆解的全部规范内容。

Markdown 格式规范的一个典型段落如下。

```
## 搜索 API

### 接口定义
- 路径：POST/search
- 请求体：{query: string, top_k: int （默认 10）}
- 响应体：SearchResult[]（按 score 降序）

### 行为约束
- 空查询返回 400
- top_k 最大值 100，超过则截断为 100
- 响应时间 < 500ms（P95）

### 不做
- 不做分页（本期）
- 不做搜索历史记录
```

这段规范人类能读懂，AI 也能依据它来生成代码。它的精度恰好能满足 AI 执行的需求——明确告知 AI“做什么”“不做什么”和“关键约束”，但没有规定“用什么库”或“代码结构怎么组织”。

Markdown 的劣势在于**不可机器验证**。你无法编写一个程序来自动检查“代码是否符合这段 Markdown 格式的规范”，因为 Markdown 规范采用自然语言，缺乏形式化语义，只能依赖 AI 理解或人工检查来验证。

3.3.2 YAML：结构化的中间地带

YAML 在 SDD 中主要用于接口契约，特别是 API 规范。OpenAPI 格式的 YAML 文件既能被人类阅读，也能被工具自动验证和处理。

YAML 的优势在于，你可以通过工具自动生成 Mock 服务、运行契约测试以及生成 API 文档。这些能力在 Markdown 格式的规范中无法实现。

YAML 的劣势是编写过程比 Markdown 烦琐。YAML 需要区分缩进，且语法规则较多，对于非 API 类的规范（如需求、架构决策、任务拆解），其表达能力反而不如 Markdown。

实践建议：API 接口契约采用 YAML（OpenAPI 格式），其他所有规范采用 Markdown。这是目前社区的主流实践。

3.3.3　JSON Schema：形式化的极端

JSON Schema 提供了最高的机器可验证性——可以精确定义数据结构的每个字段、类型和约束，再通过验证器自动检查。但它对人类极不友好——读起来像代码而不是文档，且对 AI 来说也不是最高效的格式（在相同信息量下，JSON Schema 消耗的 Token 远多于 Markdown）。

在 SDD 实践中，JSON Schema 通常只出现在一个地方：数据模型的形式化定义。当需要确保 AI 生成的数据结构与规范完全一致时，通过 JSON Schema 定义模型、验证器自动检查，比依赖 AI 自觉遵守 Markdown 描述更可靠。表 3-3 展示了规范格式选择指南。

表 3-3　规范格式选择指南

规范类型	推荐格式	理由
需求文档（proposal.md）	Markdown	自然语言表达，人类和 AI 都能轻松理解
架构设计（design.md）	Markdown	需要叙述性内容（如决策理由、权衡分析）
ADR	Markdown	重点是“为什么”，不是结构化数据
API 接口契约	YAML（OpenAPI）	需要工具自动验证、Mock 服务生成
数据模型	Markdown 或 JSON Schema	针对简单模型，可采用 Markdown 格式的表格；需要严格验证时，可采用 JSON Schema
任务拆解（tasks.md）	Markdown	需要灵活表达依赖关系和验收标准

3.4　版本控制与协作：Git 中的规范管理

规范格式选定后，应在哪里进行管理？答案只有一个——Git。

这不是一个建议，而是 SDD 第四原则（规范可版本控制）的直接

要求。规范和代码必须在同一个 Git 仓库里管理，遵循相同的分支策略、审查流程和发布节奏。

3.4.1 目录结构约定

SDD 项目的标准目录结构已在第 2 章的六阶段工作流中介绍过，此处我们将对其进行细化。

```
my-project/
├── specs/                          # 规范文档（第一手工件）
│   ├── proposal.md                 # 需求规范
│   ├── design.md                   # 架构设计
│   ├── tasks.md                    # 任务拆解
│   ├── contracts/                  # 接口契约
│   │   ├── api-spec.yaml           # API 规范（OpenAPI 格式）
│   │   └── data-models.md          # 数据模型定义
│   └── adrs/                       # 架构决策记录
│       ├── 001- 向量数据库选型 .md
│       └── 002- 缓存策略 .md
├── src/                            # 代码（衍生物）
├── tests/                          # 测试
├── CLAUDE.md                       # 规范索引（供 Claude Code 使用）
├── .cursorrules                    # 规范摘要（供 Cursor 使用）
└── .github/
    └── copilot-instructions.md     # 规范摘要（供 GitHub Copilot 使用）
```

需要注意的是，CLAUDE.md、.cursorrules、copilot-instructions.md 这三个文件不是规范本身，而是规范的索引和摘要。完整的规范始终存放在 specs/ 目录下，这三个文件仅为不同的 AI 工具提供入口，指向规范的真实位置。

这种设计保证了一项关键原则：**规范只有一个版本**。不会出现 CLAUDE.md 中提到了一项约束，而 .cursorrules 中涉及另一项约束且两者矛盾的情况，因为它们都指向 specs/ 目录下的同一份源文件。

3.4.2　规范审查：规范审查先于代码审查

在传统开发流程中，代码审查是质量保证的最后一道关口；而在 SDD 流程中，需要增设一道前置关口——规范审查（见图 3-6）。

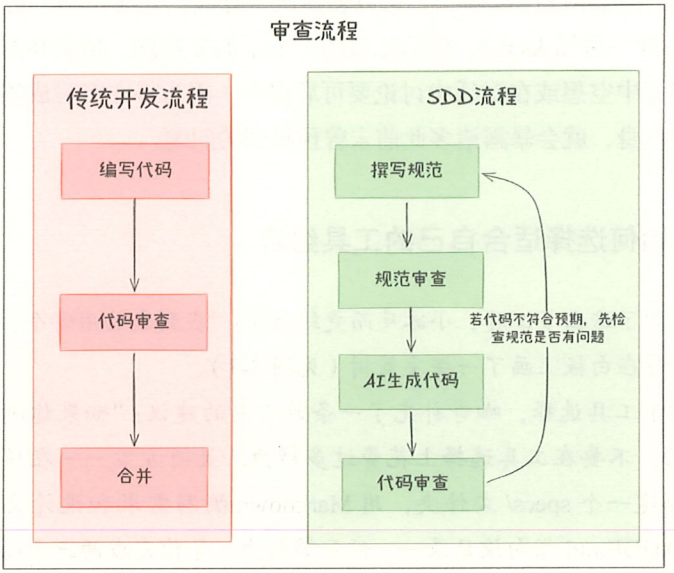


图 3-6　传统开发流程与 SDD 流程的代码审查流程

规范审查的不是代码，而是规范本身的质量，例如以下事项。

- 目标是否清晰？边界是否明确？
- 技术方案是否合理？是否遗漏了关键约束？
- ADR 中的推理过程是否站得住脚？
- 验收标准是否可测试？

在 Git 工作流中，这意味着一个功能分支可能包含两次 PR：第一次 PR 仅包含 specs/ 目录下的规范文件，审查通过后，再提交第二次 PR，其中包含代码实现。或者，更简单的做法是，一次 PR 同时包含规范和代码，但审查员需要先审查 specs/ 下的文件，确认方向正确后，再审查代码。

3.4.3 分支策略：规范探索

在 SDD 中，Git 分支衍生出了一个新用途——**规范的探索与对比**。

当面对两个相互竞争的技术方案时（如“通过 Redis 进行缓存”与“通过应用内存进行缓存”），可以创建两个分支，先各撰写一份 design.md，各列一份 tasks.md，然后比较两个方案的复杂度、依赖和约束。这比在脑海中空想或在对话中讨论要可靠得多，因为将方案写成规范文档的过程本身，就会暴露诸多此前未曾预料到的问题。

3.5 如何选择适合自己的工具组合

介绍了这么多工具，小冰反而更纠结了：“我到底该用哪个？”

咖哥在白板上画了一棵决策树（见图 3-7）。

对于工具选择，咖哥补充了一条最重要的建议：“如果你刚开始接触 SDD，**不要在工具选择上花费过多精力**。最简方案——在项目根目录下构建一个 specs/ 文件夹，用 Markdown 编写需求和设计文档，再将 CLAUDE.md 指向该目录——便足够起步。等你完成两三个项目、积累了实践经验之后，再考虑是否引入 OpenSpec 或 Spec-Kit 来标准化流程。”

他进一步打比方：“这就像学画画。初学者不需要最贵的画笔和画布，一支铅笔、一张白纸就能开始练习。等画技提升了，自然会知道什么样的工具适合什么样的创作。SDD 也是一样，对方法论的肌肉记忆比工具选择更重要。”

关于工具组合决策树的更多介绍，请访问 https://kage-ai.com/sdd/#ch3-tree。

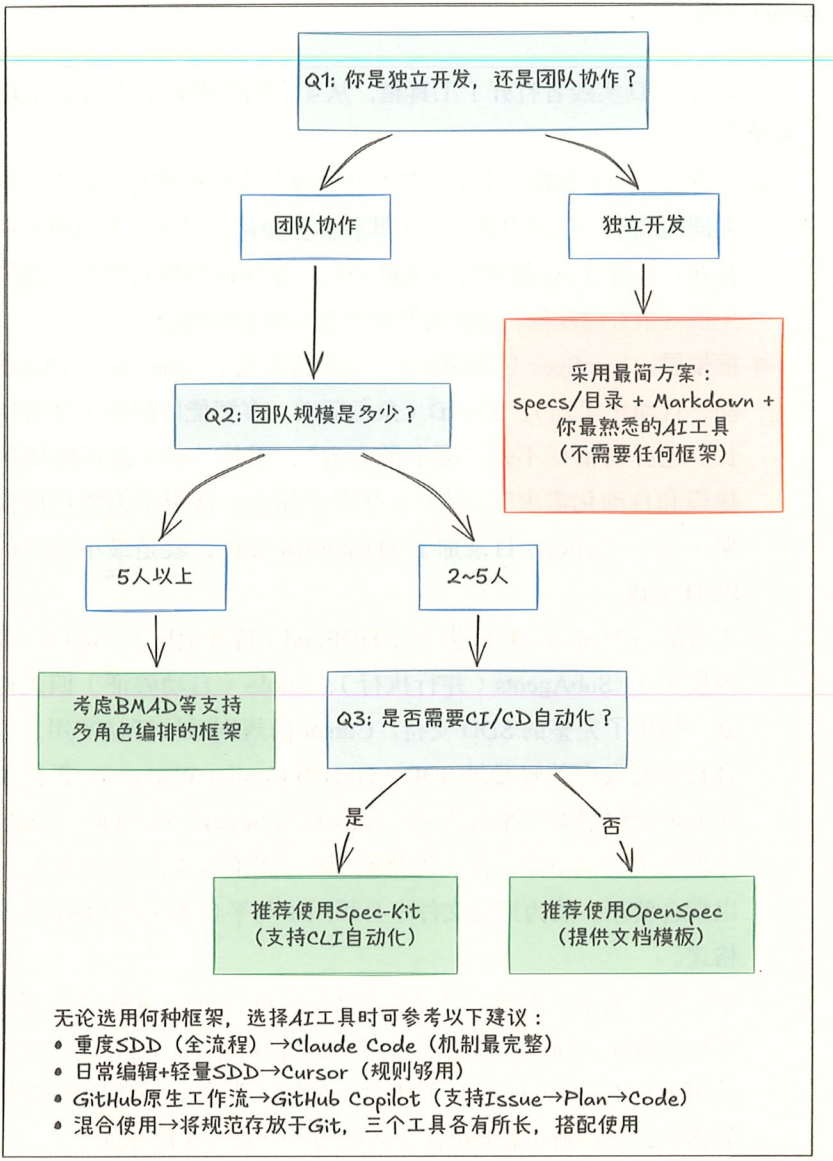


图 3-7　SDD 工具选择决策树

本章小结

本章为 SDD 实践者打开了工具箱，从 4 个维度审视了当前的 SDD 工具生态。

- 生态层：原子技能、上下文持久化、规范生成框架与完整方法论共同构成了一条能力谱系。工具名称会变化，但这 4 类问题长期存在：怎样让 AI 稳定完成关键动作，怎样保存项目记忆，怎样生成可审查的规范，怎样将开发变为可复盘的循环。
- **框架层：**OpenSpec（文档驱动、与平台无关）、Spec-Kit（CLI 驱动、自动化优先）、BMAD（角色驱动、多智能体编排）各有所长。选择的依据不是“哪个最先进”，而是“哪个最匹配团队规模和自动化需求”。对个人开发者而言，甚至不需要任何框架——一个 specs/ 目录加上 Markdown 文件，就是最小可行的 SDD 实践。
- **工具层：**Claude Code 凭借 CLAUDE.md（持久记忆）、Skills（规范模板）、SubAgents（并行执行）、Hooks（自动验证）四大机制，提供了完整的 SDD 支持。Cursor 的规则系统轻量实用，适合日常开发中的轻量级 SDD。GitHub Copilot Workspace 提供了从 Issue 到代码的规划能力。Kiro 则将 Specs、Steering、Hooks 等概念内置到 IDE，代表规范流程产品化的方向。这些工具可以混合使用，因为规范文件本身采用了与平台无关的 Markdown 格式。
- **格式层：**需求、架构、任务拆解使用 Markdown；API 接口契约使用 YAML（OpenAPI 格式）；需要严格机器验证的数据模型可选用 JSON Schema。核心原则是：**使用人类和 AI 都能轻松理解的格式，**仅在确实需要机器验证的场景引入形式化规范。

从第 4 章起，我们将正式进入实战环节。第 4 章以贯穿全书的“智能日报生成器”项目为载体，手把手演示 SDD 的第一个阶段：如何将模糊的需求意图转化为一份结构化的 proposal.md。工具箱已经准备就

绪，是时候动手了。



咖哥发言

我见过太多开发者在“选择最佳工具”上耗费了三天时间，却一行规范都没写。工具固然重要，但它们的作用仅仅是让你更快地完成那件事——前提是，你得先动手去做。先撰写一份最粗糙的 proposal.md，哪怕只有 5 行，这比花三天时间研究 OpenSpec 和 BMAD 哪个更好有价值 100 倍。最好的工具，就是你现在开始使用的那个。

思考题

1. 回顾 OpenSpec、Spec-Kit 和 BMAD 这三个框架的设计哲学。若让你从零开始设计一个 SDD 框架，你会选择哪种路线（如文档驱动、CLI 驱动或角色驱动）？为什么？是否存在第 4 种可能的路线？
2. 本章提到 Claude Code 的 4 种机制（CLAUDE.md、Skills、SubAgents、Hooks）分别对应 SDD 工作流的不同环节。若你目前使用的是 Cursor 或 GitHub Copilot，你会如何利用其已有的机制来模拟这 4 种机制？哪些可以模拟，哪些无法模拟？
3. Markdown 规范的最大优势是“人类与 AI 都能理解”，最大劣势是“无法机器验证”。你认为 5 年后，这一权衡会发生变化吗？若 AI 的规范理解能力足够强，是否仍需要形式化的规范格式（如 YAML、JSON Schema）？

第 4 章
条分缕析：需求分析与规范撰写

通过条分缕析的方法，将模糊不清的需求解构为清晰的架构与具体的规则。

从本章开始，我们将正式进入实战环节。

回顾前 3 章，我们已完成理论体系的构建：明确了 SDD 的必要性（见第 1 章）、核心定义（见第 2 章）以及落地工具（见第 3 章）。然而，认知并不等同于能力，正如熟知烹饪理论并不代表能掌勺实践。在接下来的 4 章（第 4 章至第 7 章）中，我们将围绕“智能日报生成器”这一完整项目，从需求分析到验证交付，手把手教你走完 SDD 的 6 个阶段。

该项目旨在自动收集团队成员的 Git 提交、任务状态及沟通记录，生成结构化日报并推送通知，是一个极具实用价值的自动化工具。

小冰对这个项目表现出浓厚的兴趣：“我们团队每天都要花费大量时间手写日报，不仅效率低下，而且产出往往无人问津。如果能实现自动生成，那就太好了。”

“没问题。”咖哥回应道，“那我们现在就从零开始。但请务必注意，我们的起点不是‘编写代码’，而是‘制定规范’。本章的唯一产出是一个名为 proposal.md 的文件，在此期间，我们不会编写任何代码。”

小冰略显急切地问道：“花整整一章的时间只为写一个文件？”

“这就好比‘庖丁解牛’，”咖哥解释道，“在刀刃碰触到牛体之前，庖丁早已通过眼睛将整头牛的骨骼结构、肌肉纹理及关节间的空隙洞察得一清二楚。他之所以能做到‘游刃有余’，并非因为刀具锋利，而是因为他在动手之前就已经明确了下刀的路径。需求分析正是这个‘审

视’的过程——在 AI 生成第一行代码之前，我们必须先将整个项目的架构与细节洞察明晰。”

4.1　从模糊意图到结构化规范

每一个项目的雏形，往往都始于一个模糊的概念。

“做一个自动生成日报的工具”——这便是小冰最初的构想。倘若她直接将这句指令抛给 Claude Code，它确实能够生成相应的代码。然而，这段代码的形态将充满不确定性：它极可能只是一个从特定数据源拉取数据，并采用 AI 默认格式生成报告的脚本——这与小冰实际所需的解决方案相去甚远，甚至背道而驰。

问题的核心并非 AI 生成的代码质量低劣，而是 AI **缺乏做出正确的决策所需的信息**。“做一个自动生成日报的工具”这一模糊指令背后，至少潜藏着以下未被解答的关键问题。

- **数据源**：数据源自何处？是 Git、Jira，还是飞书、Slack？抑或需要整合所有渠道？
- **受众**：日报的接收对象是谁？是供团队成员自查，还是汇报给团队 Leader？
- **格式**：日报的呈现形式为何？是纯文本、Markdown，还是 HTML 邮件？
- **触发机制**：如何启动生成流程？是定时任务、手动触发，还是依赖推送通知？
- **边界**：项目范围如何界定？仅限于数据的收集与格式化，还是包含智能摘要生成？

若这些问题得不到明确回答，AI 便只能依靠猜测。而 AI 的每一次“猜测”，本质上都是一次不受控的决策——它或许能猜对，但你无从知晓其决策逻辑与依据。最糟糕的情况是，三天后你才发现它的理解与你的初衷背道而驰，但此时代码已编写数百行，推翻重来的代价将极为高昂。

需求分析的核心要义，在于将模糊意图中隐含的决策显式化。这一过程旨在将每一次不确定的“猜测”转化为明确、有据可查且可追溯的“选择”（见图 4-1）。

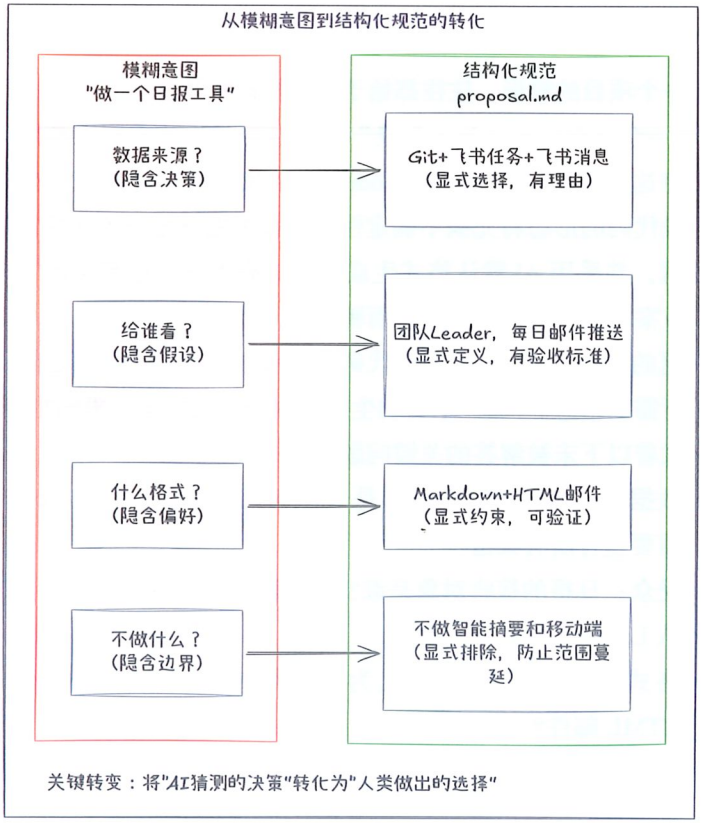


图 4-1　从模糊意图到结构化规范：让隐含决策显式化

小冰问道：“这听起来与传统的需求分析并无二致，它与 SDD 究竟有何本质区别？”

“区别在于产出物及其目标读者。”咖哥解释道，“传统需求分析产出的是产品需求文档（Product Requirements Document，PRD），其受众是人类开发者。而在 SDD 中，需求分析产出的是 proposal.md，其读者既包含人类，也包含 AI。这意味着，规范的格式与精度必须针对 AI 的

理解逻辑进行优化，例如，采用结构化的 Markdown 而非冗长的段落，使用精确的数值而非模糊的定性描述，利用列表而非散文式的叙述。”

4.2　规范的三层结构

一份高质量的 proposal.md 绝非需求点的无序堆砌，而应具备严密的层次化结构。如图 4-2 所示，SDD 规范遵循“目标—范围—验收”的三层递进逻辑，从宏观的“为什么做”逐步收敛至具体的“做什么”与“如何验收”。

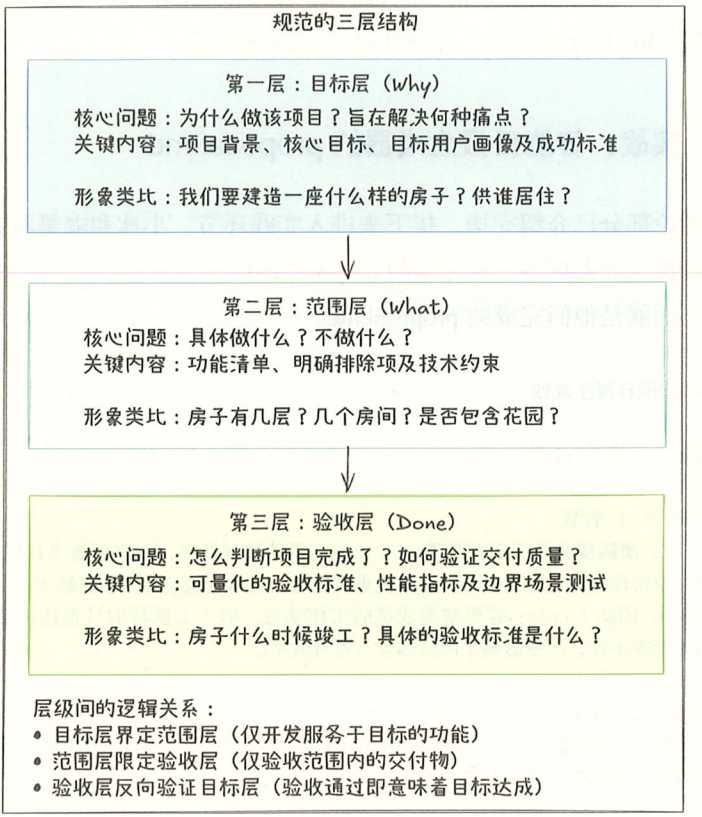


图 4-2　规范的三层结构

这三层之间存在着严密的逻辑约束关系。目标层决定了范围层的边界——若目标设定为“为 5 人小团队自动生成日报”，那么“支持 10 000 人规模的高并发部署”便不应出现在范围层中。同理，范围层界定了验收层的范畴——若“智能摘要”已被明确排除在范围之外，验收标准中自然不应出现“摘要质量评估”。

小雪敏锐地指出了一个关键点：“第三层‘验收层’至关重要，因为它构成了 AI 生成代码后的验证基准。若验收标准模糊不清（如‘能用就行’），我们就无法客观评判 AI 生成代码的优劣。”

“正是如此。”咖哥补充道，“验收标准必须具备可测试性，最好能够实现自动化测试。‘日报生成速度快’并非一个合格的验收标准，而‘日报在 30s 内生成完成’才是明确且可衡量的。”

4.3 实战：智能日报生成器的 proposal.md

理论部分已介绍完毕，接下来进入实践环节。小冰和咖哥将共同为智能日报生成器撰写一份完整的 proposal.md。

下面就是他们完成的 proposal.md。

```
# 智能日报生成器

## 1. 背景与目标

### 1.1 背景
目前，团队成员每天需要耗费 15~30min 手动撰写日报，内容涵盖当日代码提交、任务进展及协作沟通情况。这一过程不仅重复低效，且常因遗忘导致信息缺失。
另外，团队 Leader 需要掌握成员的工作动态，但人工撰写的日报往往信息密度低、格式参差不齐，严重影响了阅读体验与管理效率。

### 1.2 目标
构建一个自动化工具，旨在从团队现有的协作平台中抓取当日工作数据，自动生成结构化日报，并精准推送给相关干系人。

### 1.3 目标用户
```

- 主要用户（数据生产者）：5 ~ 15 人的研发团队核心成员
- 主要读者（数据消费者）：团队 Leader 及项目经理

2. 功能范围

2.1 做什么（本期）

数据采集
- 对接 GitHub/GitLab，获取当日代码提交记录（含 Commit Message 及变更统计）
- 对接飞书/Jira，抓取当日任务状态流转（如新建、进行中、已完成）
- 从飞书群消息中提取与项目强相关的讨论摘要

日报生成
- 将采集数据按“代码提交→任务进展→协作沟通”的逻辑进行三段式编排
- 为每位团队成员生成独立的日报段落
- 输出 Markdown 格式（支持无缝转换为 HTML 邮件）

推送通知
- 每个工作日 18:00 自动触发日报生成流程
- 通过 SMTP 服务向团队 Leader 发送 HTML 格式日报
- 通过飞书机器人将 Markdown 格式日报推送至团队群组

2.2 不做什么（本期明确排除）
- 不做智能摘要和 AI 总结（本期仅做数据清洗与整理，不涉及语义理解）
- 不做移动端
- 不做历史日报的搜索与统计图表分析
- 不做个人日报的手动编辑与补充功能
- 不做支持多团队隔离/多租户架构

2.3 技术约束
- 开发语言：Python 3.11 及更高版本（复用团队现有技术栈）
- 部署环境：公司内网服务器（Docker 容器化部署）
- 外部依赖：GitHub API、飞书开放平台 API、公司邮件服务（SMTP）
- 运行方式：基于 Cron 的定时任务，不需要常驻后台服务
- 数据存储：本地 SQLite（仅用于存储日报历史记录，不需要部署远程数据库）

3. 验收标准

3.1 功能验收
- [ ] 能够准确获取指定 GitHub 仓库过去 24h 内所有成员的 Commit 记录
- [ ] 能够准确获取飞书项目中当日发生状态变更的任务列表

```
- [ ] 能够从飞书群消息中筛选并提取包含指定关键词的讨论记录
- [ ] 生成的日报必须包含“代码提交”“任务进展”“协作沟通”三个核心板块
- [ ] 日报 Markdown 格式规范，可直接渲染为高可读性的 HTML 页面
- [ ] 邮件发送链路畅通，团队 Leader 能成功接收到 HTML 格式日报
- [ ] 飞书机器人能够成功将日报推送至指定群组

### 3.2 性能验收
- [ ] 单次日报生成耗时（以 5 人团队为例）<60s
- [ ] 单一数据源（如某 API）超时或失败，不应阻断其他数据源的采集
- [ ] 采集失败的数据源需要在日报中显式标注“数据获取失败”，严禁静默跳过

### 3.3 边界场景
- [ ] 若团队成员当日无任何提交、任务或消息，日报对应板块应显示“今日无记录”
- [ ] 当 GitHub API 触发限流时，系统应具备等待并重试机制（上限三次）
- [ ] 当飞书 Token 过期时，系统应能自动刷新 Token
- [ ] 系统需要识别非工作日（周末及法定节假日），并自动跳过日报生成

## 4. 术语定义
| 术语     | 定义                                                  |
|--------|-----------------------------------------------------|
| 日报     | 系统自动生成的每日工作摘要，涵盖代码、任务、沟通三个维度     |
| 成员     | 已在系统中完成 GitHub 与飞书账号映射绑定的团队成员          |
| 数据源   | 提供原始工作数据的外部协作平台（GitHub、飞书任务、飞书消息） |
| 采集窗口 | 单次采集所覆盖的时间跨度，默认为当日 00:00 至任务执行时刻   |
```

完成 proposal.md 后，咖哥让小冰仔细审阅一遍，随即发问：“你觉得这份规范写得够不够完善？”

小冰反复推敲了两遍，回答道：“该有的要素似乎都齐备了——背景、目标、功能范围、技术约束及验收标准……特别是验收标准采用了 Checklist 形式，每一条都具备可测试性。”

咖哥补充道：“确实不错，但要注意几个容易被忽略的关键点（见图 4-3）。”

关于如何使用 4 种工具撰写同一份 proposal，请访问 https://kage-ai.com/sdd/#ch4-proposal。

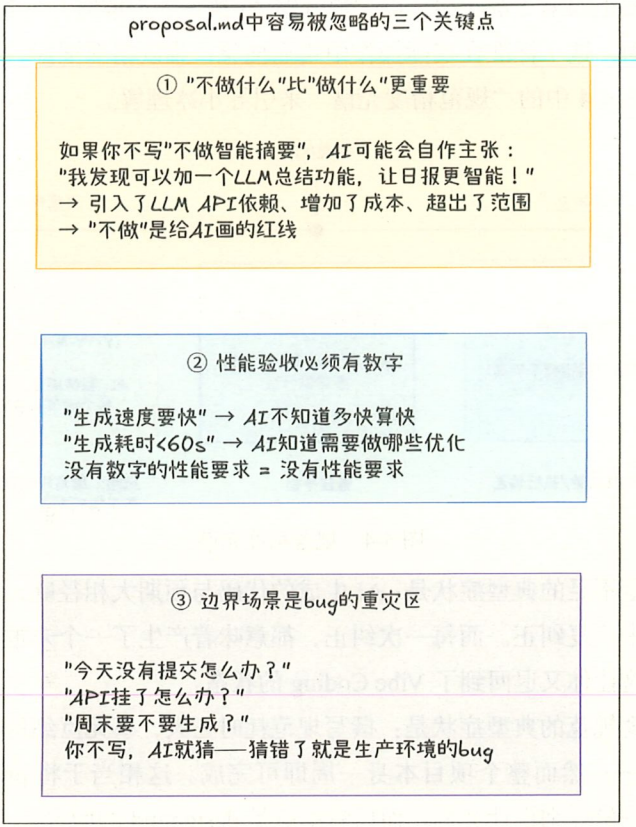


图 4-3　proposal.md 中容易被忽略的三个关键点

4.4　常见陷阱：过度规范与规范不足

完成第一份 proposal.md 后，小冰的新焦虑接踵而至："咖哥，我感觉这份规范还不够细致，例如，邮件标题的格式如何设置、日报各部分该用几级 Markdown 标题、飞书消息的过滤关键词如何配置……这些细节需要补充进去吗？"

这正是初学者在实践 SDD 时常有的困惑：**规范的颗粒度究竟该如何把握？**

答案正如第 2 章所述：以“恰好够 AI 执行”为准绳，一句不多。

然而，这个标准在实际操作中略显抽象，难以精准把握。为此，咖哥借用图 4-4 中的“规范精度光谱”来引导小冰理解。

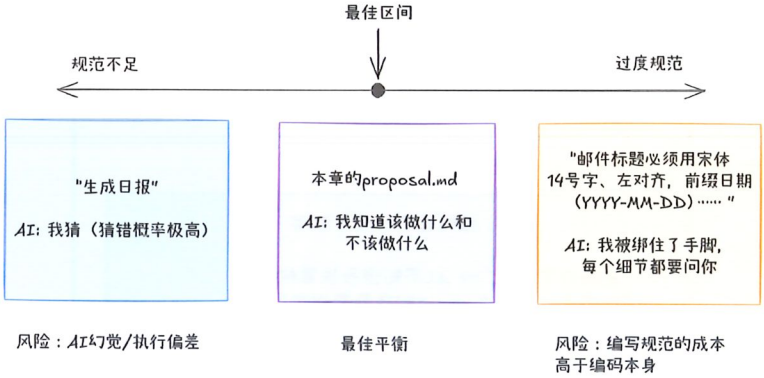


图 4-4　规范精度光谱

规范不足的典型症状是：AI 生成的代码与预期大相径庭，你不得不在对话中反复纠正。而每一次纠正，都意味着产生了一个未被记录的决策——这让你又退回到了 Vibe Coding 的状态。

过度规范的典型症状是：撰写规范耗时三天，事无巨细地覆盖了每个细节——然而整个项目本身一周即可完成。这相当于将 proposal.md 写成了一份详细设计文档，而后者本应是 design.md 的职责。

那么，如何把握恰当的规范尺度呢？判断标准见表 4-1。

表 4-1　proposal.md 的内容边界：什么该写，什么不该写

该写（What）	不该写（How）
数据源：GitHub+ 飞书	使用哪个 Python 库调用 GitHub API
输出格式：Markdown+HTML 邮件	HTML 模板的具体 CSS 样式代码
性能要求：执行时间 <60s	使用多线程还是 asyncio 进行并发处理
推送方式：邮件 + 飞书机器人	SMTP 配置的具体参数格式
功能边界：不做智能摘要	—
异常处理边界：API 超时重试 3 次	重试的退避策略（指数退避或固定间隔退避）

表 4-1 左列是 proposal.md 应当涵盖的核心内容——**关键决策与约束**，而右列则是 proposal.md 不应涉足的内容——**具体的实现细节**。这些实现细节，若涉及架构层面的抉择，应在 design.md 中予以明确；若仅为编码层面的细枝末节，则可交由 AI 自行处理。

小雪由此总结出一条简洁而精准的判断法则："若一项决策会改变你对系统本身的理解，就应将其写入规范；若一项决策仅影响代码的书写方式，且不改变系统的最终行为，则可放心交给 AI。"

咖哥闻言，赞许地笑了："这个总结，完全可以直接收录进教材。"

4.5　用 Claude Code 辅助编写规范

编写规范本身同样可以借助 AI 的力量。但关键在于——**AI 应作为你思考的辅助，而非替你做出决策**。

一种常见的误用是直接要求 AI："帮我写一份智能日报生成器的需求文档。"这样 AI 生成的规范虽然看似完整，实则充满了它基于自身假设所生成的"幻觉"。更优的策略是利用 AI 来**引导你进行结构化思考**。

在 Claude Code 中，你可以通过 CLAUDE.md 加载规范模板，再通过对话逐步完善。以下是一个推荐的 AI 辅助规范编写工作流。

图 4-5 所示工作流的核心在于 Step 3——**AI 应当负责提出问题，而非擅自作答**。当 AI 询问"推送方式是什么"时，它实际上是在协助你识别一个可能被遗漏的决策点。这种方式远优于 AI 默认假设"就用邮件吧"——因为你的真实需求或许是飞书机器人，抑或两者兼具。

在试用这套流程后，小冰感慨道："原来编写规范并非必须从零开始。AI 提供的模板框架与追问引导，能有效帮助我梳理思路。但最终的决策权——无论是选定数据源、明确汇报对象，还是界定完成标准——始终牢牢掌握在我自己手中。"

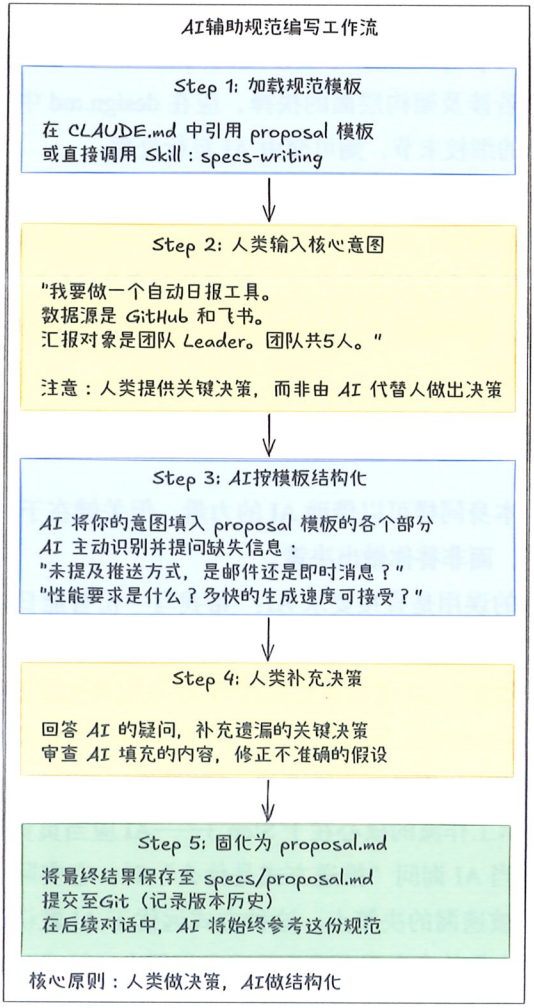


图 4-5　AI 辅助规范编写工作流

“这正是 SDD 模式下人机协作的理想分工。”咖哥总结道，“人类负责确立意图与关键决策，AI 负责结构化梳理与执行。这一分工原则贯穿于 SDD 的全流程——从需求分析到架构设计，再到代码实现，无一例外。”

4.6　规范质量检查清单

在完成 proposal.md 撰写后，提交至 Git 前，请对照表 4-2 进行一次自检。

表 4-2　规范质量检查清单

编号	检查项	自问
1	目标明确	一个不了解项目背景的人，仅阅读“目标”部分，能否用一句话准确概括该项目的核心目的？
2	范围有界	“不做什么”部分是否列出了至少三项明确排除的内容？
3	验收可测	是否每一条验收标准都能转化为具体的自动化测试用例？
4	性能有数	涉及性能的验收标准是否都包含可量化的具体数值指标？
5	边界覆盖	是否定义了针对异常情况（如数据缺失、API 故障、空输入等）的处理策略？
6	无实现细节	规范中是否避免了“使用何种库”“代码如何组织”等实现层面的具体描述？
7	术语一致	关键术语是否已明确定义，且全文表述是否保持一致？
8	AI 可执行	如果将这份规范交给一个从未参与过项目讨论的 AI，它能否准确理解需要做什么、不做什么以及做到何种程度？

需要注意的是，第 8 条是最关键的检查项，也是检验规范质量的终极问题。规范的读者主要包括两类：同事与 AI。如果一份规范能够让没有任何背景信息的 AI 准确理解任务目标、范围和验收标准，那么这份规范便是合格的。

本章小结

这一章完成了 SDD 六阶段工作流的首要步骤——将模糊的需求意图转化为结构化规范。

需求分析的核心任务并非“穷尽所有功能”，而是促使隐含的决策显性化。每一个“AI 可能产生误解或猜测的环节”，都应转化为“人类

明确做出的选择”，包括数据来源、目标受众、实现程度以及排除范围。这些选择被记录在 proposal.md 中，成为后续所有开发工作的基石与验收标准。

proposal.md 遵循“目标层（Why）→范围层（What）→验收层（Done）”的三层结构，三者之间存在严密的逻辑约束关系：目标层界定范围层，范围层定义验收层，而验收层则反向验证目标层的实现程度。

撰写规范时，对精度的把控是一种需要刻意练习的核心能力：其标准在于“恰好够 AI 执行”。关键在于将核心决策与约束写入规范，而将具体的实现细节留给 AI。判断这一界限的标准十分明确：如果某项决策改变了系统的行为，则必须写入规范；如果它仅改变代码的编写方式，则应交由 AI 决定。

在第 5 章中，我们将在 proposal.md 的基础上进行架构设计——撰写 design.md，内容涵盖系统架构、技术选型及接口契约。从界定“做什么”到“怎么做”，这是 SDD 流程中信息密度最高的一次跨越。



咖哥发言

许多开发者习惯跳过需求分析直接着手设计，这无异于庖丁尚未看清牛的骨骼结构便贸然下刀，非刀不利，实乃不知从何处切入。投入 30min 撰写一份 proposal.md，后续能为你节省的绝不止 30min 的时间，更可能是长达三天的返工成本。此外，你规避的也不只是时间损耗，还有情绪上的无畏内耗：将心态从“为何 AI 又生成了我不想要的结果”的焦虑，转变为“AI 严格遵守规范执行，一次到位”的从容。

思考题

1. 请回顾本章介绍的“智能日报生成器”项目。假设你担任该项目的 Spec Reviewer（规范审查员），你会提出哪些具体的改进建议？此外，是否存在未被覆盖的边界场景，或是定义模糊的验收标准？

2. proposal.md 的“不做什么”部分明确排除了“智能摘要”功能。然而，如果团队 Leader 坚持要求在第一期交付中包含 AI 总结功能，你将如何修订 proposal.md？请具体分析这一变更会对哪些验收标准和技术约束产生影响。
3. 本章展示的“AI 辅助编写规范工作流”（见图 4-5）遵循“人类做决策，AI 做结构化”的原则。然而，随着 AI 决策能力的提升（例如，在特定领域，AI 的知识储备可能超越人类），我们是否应考虑将部分决策移交给 AI？如果答案是肯定的，那么哪些决策可以移交，哪些又必须保留在人类手中？

第 5 章
运筹帷幄：架构设计与技术决策

运筹帷幄之中，决胜于千里之外——好的架构设计，往往在落地之前便已奠定胜局。

小冰看着在第 4 章中完成的 proposal.md，熟练地打开 Claude Code，准备将这份详尽的蓝图转化为代码。

“等一下。”咖哥的声音适时响起，拦住了她即将敲击键盘的手。

“proposal.md 不是已经写好了吗？”小冰不解地抬头，“目标清晰、范围明确，连验收标准都量化到了具体数字——按道理说，AI 现在应该可以开始干活了吧？”

咖哥没有直接回答，而是调出了一段尘封的历史影像，那是工程史上最著名的灾难之一。

1940 年 11 月 7 日，美国华盛顿州的塔科马海峡大桥——一座刚建成 4 个月的悬索桥——在风速仅为 18m/s 的普通大风中，桥面剧烈扭曲、翻滚，最终断裂，坠入海峡。

影像定格在崩塌的瞬间，咖哥的声音低沉了下来：“事后复盘发现，大桥的‘需求文档’其实完全没问题——跨度、载荷、通行能力，每一项都符合当时的规范。但它输在了架构设计上——工程师忽略了气动弹性效应，导致桥面刚度不足，无法抵抗风致振动。”

“需求没问题，是设计出了问题。”咖哥指着影像中崩塌的大桥说道，“proposal.md 只是告诉你‘要造一座什么桥’，如跨度、车道数、载荷要求等。但它并没有告诉你‘桥的结构该如何搭’——是用悬索还是斜拉？桥塔如何配重？如何抵抗风载荷？从 proposal.md 到代码，中

间缺失了一个至关重要的环节——架构设计。”

他转身在白板上重重地写下一行字：

proposal.md 定义“做什么”，design.md 定义“怎么做”。

“如果说 proposal.md 是甲方提出的需求书，那么 design.md 是建筑师绘制的施工蓝图。没有蓝图就贸然开工，你得到的绝不是一座稳固的大桥，而只是一堆随时可能倒塌的砖头。”

小冰的脑海中瞬间浮现出上个月的惨痛经历：当时她使用 proposal.md 让 AI 生成智能日报生成器。AI 确实精准理解了需求，但生成的代码将所有逻辑（包括数据采集、格式转换、邮件发送、飞书推送）全部塞进了一个 600 行的单文件中。代码虽然能运行，但耦合度极高，牵一发而动全身，稍微修改就会崩溃。

“那就是缺失了设计。”咖哥一语道破，“在本章中，我们就来补上这块缺失的拼图。”

5.1　从 proposal.md 到 design.md：关键的一步跨越

从 proposal.md 到 design.md 的跨越，堪称 SDD 全流程中信息密度最高、思维跨度最大的一步。

为何这么说？因为 proposal.md 仅仅界定了**决策边界**（即“做什么”“不做什么”），却未涉及具体的**实施方案**（即“怎么做”）。而从“做什么”到“怎么做”，需要回答一系列全新的技术命题——这些问题的答案，将直接决定代码的最终结构与质量（见图 5-1）。

Clausewitz 在《战争论》中曾精辟地界定了“战略”与“战术”的区别：战略解决“在哪里打仗，为何而战”的宏观命题，而战术则聚焦“如何打，以何种阵型打”的具体执行。映射到开发流程中，proposal.md 即战略文件，而 design.md 则是战术方案。一位仅懂战略而缺乏战术素养的将军，或许知道该攻占哪座城市，却对进攻路线、先锋人选及后勤补给一无所知——这样打仗是无法取胜的。

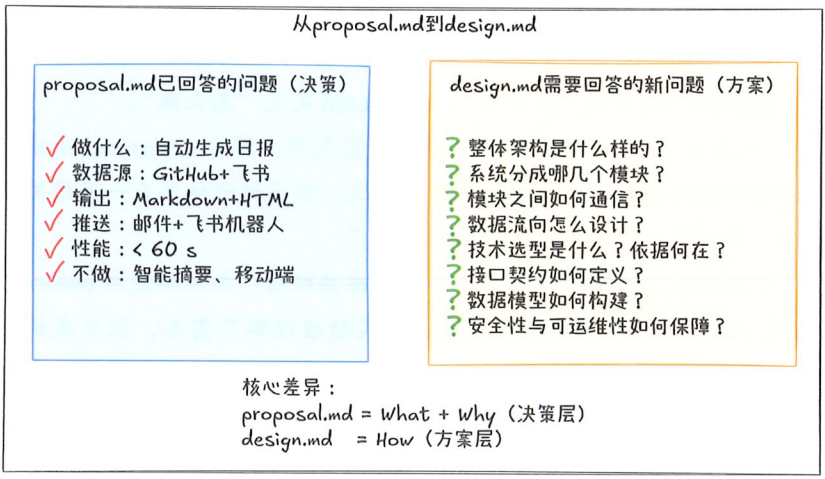


图 5-1　从 proposal.md 到 design.md：从“做什么”到“怎么做”

此时，小冰提出了一个极具洞察力的问题：“proposal.md 的读者是人也是 AI。那 design.md 呢？”

“同样如此，”咖哥点头回应，“但侧重点截然不同。proposal.md 主要协助 AI 厘清边界——明确什么该做，什么不该做；而 design.md 的核心在于帮助 AI 理解结构——代码应如何组织，模块间如何协作。若缺失了 design.md，AI 便会依据其训练数据中的‘常识’来组织代码——这种‘常识’有时精准，有时却谬以千里。你将陷入不可预测且无法掌控的被动局面。”

通常，一份标准的 design.md 应包含 4 个核心部分（见图 5-2）。

接下来，我们将逐一展开剖析。

5.2　系统架构：先画地图，再行军

5.2.1　为什么架构设计如此重要

1968 年，计算机科学家 Melvin Conway 提出了一项著名的观察，后世称之为“康威定律”（Conway’s Law）：

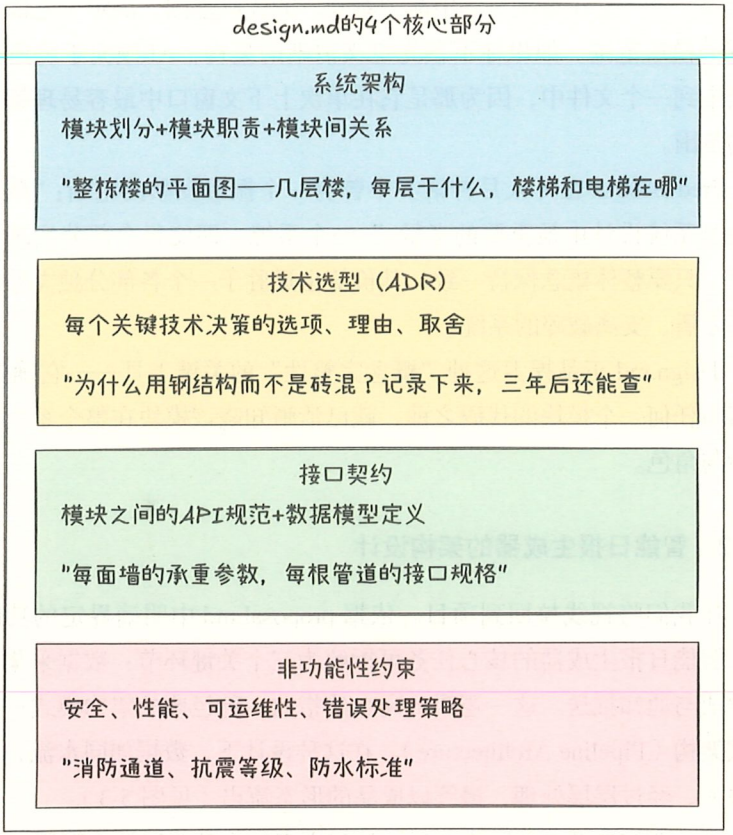


图 5-2　design.md 的 4 个核心部分

设计系统的组织，其产生的设计结构将等同于该组织内部的沟通结构。

换言之，系统的架构往往是组织结构的一面镜子。例如，一个 3 人团队构建的系统，极大概率自然地分化为三个模块，因为每位成员通常负责一块。然而，这一定律的逆向推论更为深刻：如果你渴望获得一个优秀的系统架构，你必须主动去定义它，而不是任其随性地“自然演化”。

在 AI 辅助开发的语境下，康威定律中的“组织”被赋予了新的内涵——“提示上下文”（Prompt Context）。AI 的上下文窗口就是它的“组

织边界”——它在单次对话中能“看到”的代码范围，决定了它能有效协调的模块范围。如果你不预先定义清晰的架构，AI 倾向于将所有逻辑集中到一个文件中，因为那是它在单次上下文窗口中最容易理解和掌控的范围。

Fred Brooks 在《人月神话》中曾有一个振聋发聩的论断：“概念完整性是系统设计中最重要的考量。”一个系统，即使每个部分实现不尽完美，只要整体概念保持一致，其价值也远胜于一个各部分独立精良却相互矛盾、支离破碎的系统。

design.md 正是捍卫这种“概念完整性”的关键工具——它确保 AI 在编写任何一个模块的代码之前，就已清晰知晓该模块在整个系统中的坐标与角色。

5.2.2 智能日报生成器的架构设计

让我们将视线拉回到项目。依据 proposal.md 中明确界定的功能范围，智能日报生成器的核心任务可拆解为三个关键环节：数据采集、日报生成与通知推送。这一逻辑流向自然指向一种经典的架构模式——**管道式架构**（Pipeline Architecture）。在这种设计下，数据如同水流，从源头注入，经过层层处理，最终以成品的形态流出（见图 5-3）。

这一架构选型绝非随机选择，让我们来探究其背后的决策逻辑。

智能日报生成器的系统架构本身，强烈暗示了管道式架构的适用性——数据拥有明确的流向（输入→处理→输出），且每个阶段的职责既清晰又独立。

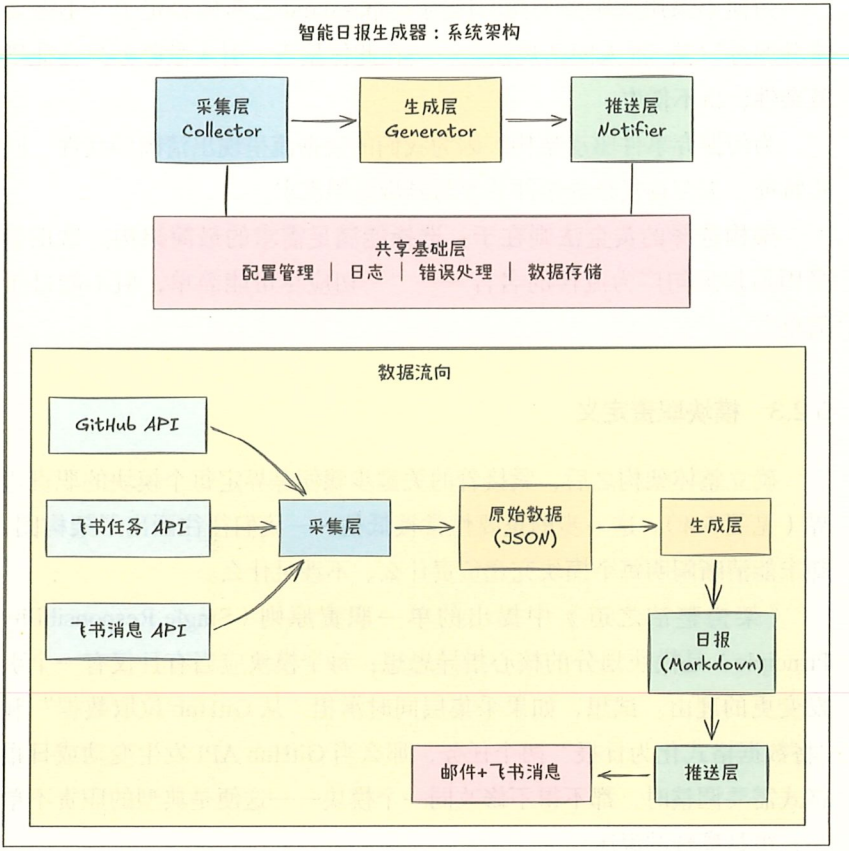


图 5-3　智能日报生成器的管道式架构



咖哥发言

优秀的设计并非凭空创造，而是从问题的内在结构中“显现”出来的。Christopher Alexander 在《建筑的永恒之道》一书中将这种深刻的洞察称为“无名特质”（the Quality Without a Name）。

为何不采用微服务架构？因为 proposal.md 已明确界定了“不需要常驻服务”及“5 人团队规模”——在此体量下，引入微服务只会徒增复杂性，得不偿失。

为何摒弃事件驱动架构？因为我们的数据流呈现出清晰的线性、同步特征，不存在复杂的事件分发与异步解耦需求。

架构选择的黄金法则在于：选择能满足需求的最简架构。这正如爱因斯坦那句广为流传的名言——“一切应尽可能简单，但不能过于简单”。

5.2.3 模块职责定义

确立整体架构之后，紧接着的关键步骤便是界定每个模块的职责边界（见图 5-4）。这一步的重要性常被低估——我们往往画出了架构图，却未能清晰阐明每个模块究竟负责什么、不涉足什么。

《架构整洁之道》中提出的**单一职责原则**（Single Responsibility Principle）是模块划分的核心指导思想：每个模块应当有且仅有一个引发变更的理由。试想，如果采集层同时承担“从 GitHub 拉取数据”和“将数据格式化为日报”两个任务，那么当 GitHub API 发生变动或日报格式需要调整时，都不得不修改同一个模块——这便是典型的职责不单一，也是耦合的温床。

注意图 5-4 中“不负责”这一列，其逻辑与 proposal.md 中“不做什么”的界定一脉相承。**在架构设计层面，明确定义“不负责”往往比定义“负责”更为关键。**在代码生成过程中，若缺乏对模块边界的负面约束，AI 倾向于将相关功能就近并入当前模块，例如，在 collector 中混入格式化逻辑，或在 notifier 中植入数据查询，从而导致模块边界模糊。

小冰审视着图 5-4，问：“这不就是面向对象设计中的‘关注点分离’（Separation of Concerns）吗？”

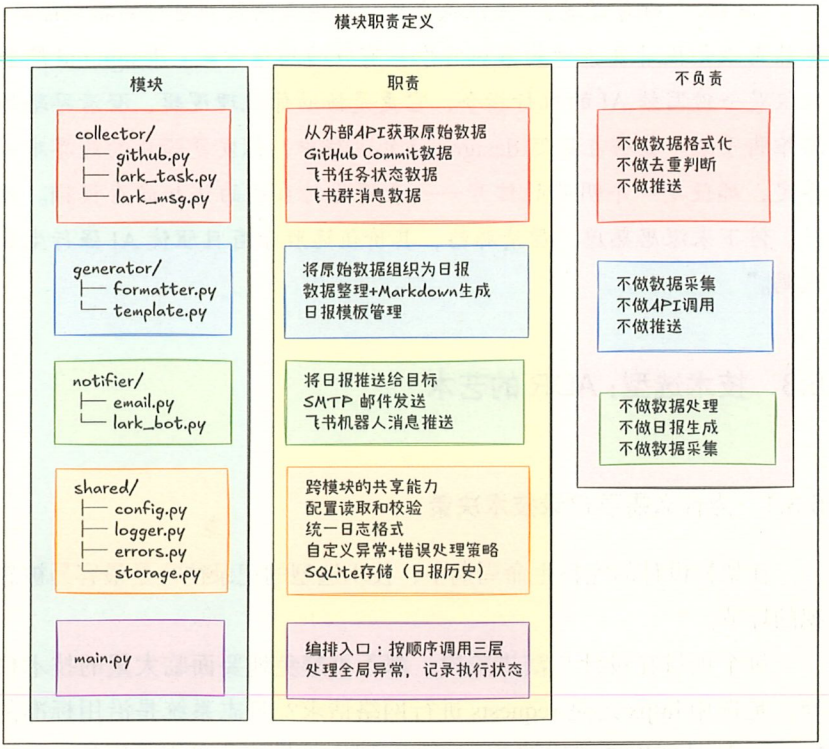


图 5-4　模块职责定义：每个模块做什么、不做什么



咖哥发言

关注点分离，本质上是将系统中的每一项责任进行解耦，确保每个模块仅响应一类特定的变化。许多人误以为关注点分离仅仅是物理上的模块拆分，然而真正的挑战并非在于“拆”，而在于“界”。倘若仅定义了“模块应当做什么”，却忽略了“它不应涉足什么”，那么在具体的实现过程中，这些边界便会遭受持续的侵蚀——那些看似无伤大雅的“顺便为之”与一次次“举手之劳”，终将导致模块间相互渗透，使得职责边界模糊不清，系统架构陷入混乱。

“没错。”咖哥说道，“关注点分离，即使无法做到绝对的物理隔离，依然是我们设计复杂结构系统时唯一有效的思维方式。design.md 绝非仅仅是一份写给 AI 的执行指令，它更是协助你梳理逻辑、澄清思路的思维脚手架。倘若在撰写 design.md 的过程中，你发觉模块的边界难以界定，那便是一个明确的信号——表明你对系统的认知尚存模糊。此时，停下来深思熟虑、厘清脉络，其价值远胜于盲目驱使 AI 强行生成代码。”

5.3 技术选型：ADR 的艺术

5.3.1 为什么需要记录技术决策

在架构设计的完整生命周期中，技术选型的记录往往是最容易被忽视的环节。

每个项目在设计与演进阶段，都不可避免地要面临大量的技术抉择：是选用 httpx 还是 requests 进行网络请求？日志系统是沿用标准的 logging 还是拥抱更现代的 loguru？配置文件格式是采用通用的 YAML 还是更严谨的 TOML？

在决策做出的当下，这些选择往往显得理所当然，甚至带有某种惯性——“团队一直沿用 requests，所以继续沿用即可”。然而，时光流转，三个月后当新成员加入项目，面对代码库中使用的 requests 提出疑问“为什么不采用支持异步且性能更优的 httpx”时，往往无人能给出确切答案。

这便是典型的“决策理由蒸发”现象。

此时，我们需要引入**架构决策记录**（ADR）这一概念。正如 Michael Nygard 在其关于 ADR 的奠基性文章中所阐述的：

架构是由那些难以逆转的决策构成的……既然一个决策难以改变，那么支撑该决策的理由便值得被郑重记录。

ADR 的核心理念极简却极具力量：针对每一个关键的技术决策，

都应当产出一份简练的文档，清晰地记载问题的背景、考量的选项、最终的抉择以及其背后的动因。

5.3.2　ADR 的格式

一份优秀的 ADR 贵在精练。ADR 标准格式通常由 5 个核心要素构成（见图 5-5），它们共同勾勒出决策的全貌。

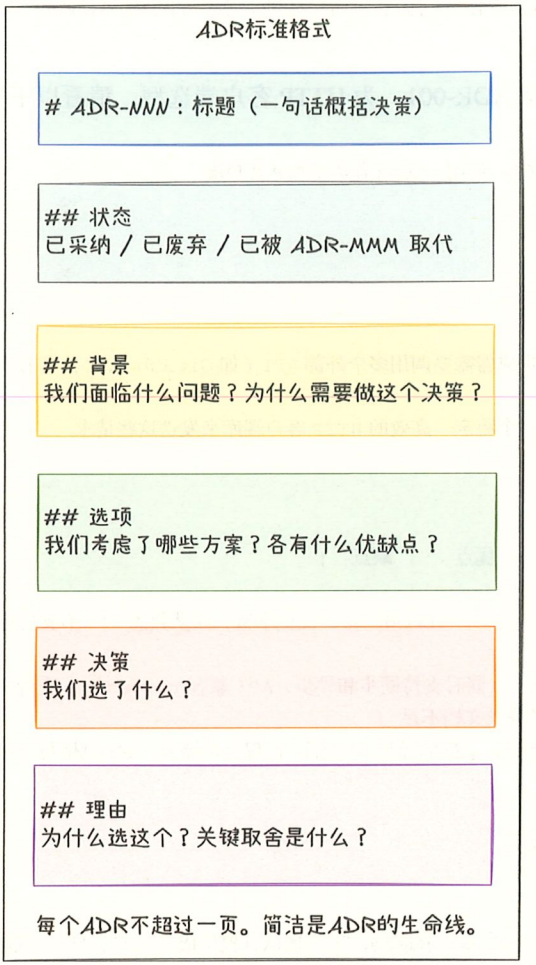


图 5-5　ADR 标准格式：五要素

Michael Nygard 曾特别强调，ADR 绝非厚重的设计文档或烦琐的评估报告，其唯一使命就是清晰地记录做出技术决策的前因后果。如果一份 ADR 的篇幅超过一页纸，那极有可能意味着开发者陷入了不必要的细节泥潭。

5.3.3 智能日报生成器的 ADR 示例

让我们以智能日报生成器为例，撰写两份 ADR 作为 design.md 的组成部分。

第一份是 ADR-001，为 HTTP 客户端选型。请看以下代码示例。

```
# ADR-001：使用 httpx 作为 HTTP 客户端

## 状态
已采纳

## 背景
智能日报生成器需要调用多个外部 API（如 GitHub API、飞书开放平台 API）来获取数据。
需要选择一个稳定、高效的 HTTP 客户端库来发送这些请求

## 选项

| 选项 | 优点 | 缺点 |
|-------|------ |------|
| requests | 社区最广泛，文档丰富，团队熟悉 | 不支持原生异步，连接池管理较弱 |
| httpx | 同时支持同步和异步，API 兼容 requests，支持 HTTP/2 | 相对较新，部分边缘场景文档不足 |
| aiohttp | 成熟的异步 HTTP 库 | 仅支持异步，API 风格与 requests 差异大 |

## 决策
使用 httpx

## 理由
1. proposal.md 明确要求“5 人团队日报生成 <60s”，且三个数据源可并发采集，因此异步能力是实现性能目标的关键
2. httpx 的同步 API 与 requests 几乎完全一致，团队迁移成本极低
```

3．采集层当前使用同步模式，未来可平滑切换至异步模式，不需要重写代码

4．之所以不选择 aiohttp，是因为我们不需要强制异步——在同步模式下程序也必须能正常运行

第二份是 ADR-002，选择数据存储方案。请看以下代码示例。

```
# ADR-002：使用 SQLite 存储日报历史

## 状态
已采纳

## 背景
虽然 proposal.md 明确要求“数据存储：本地 SQLite”，但为了确保架构的合理性，需要对这一选择进行正式评估，确认其是否真的适合本场景

## 选项

| 选项     | 优点       | 缺点       |
|--------|----------|----------|
| SQLite | 零部署、零运维、Python 内置支持、文件级备份 | 不支持高并发写入、数据量极大时查询性能下降 |
| PostgreSQL | 功能完整、支持高并发、强大的全文搜索能力 | 需要独立部署和运维、对 5 人团队日报场景而言过于沉重 |
| JSON 文件 | 实现最简单、不需要任何依赖 | 缺乏复杂查询能力、并发写入不安全、数据量增长后难以管理 |

## 决策
使用 SQLite

## 理由
1．proposal.md 明确决定运行方式为“定时任务，不需要常驻服务”，这意味着不存在并发写入场景（每天仅执行一次）
2．5 人团队一年的日报数据约为 1250 条，SQLite 处理起来绰绰有余
3．零运维高度符合项目定位——我们不想为了一个简单的日报工具去维护一个独立的数据库服务
4．Python 标准库 sqlite3 开箱即用，不需要引入额外的第三方依赖
```

小冰读完这两份 ADR 后，恍然大悟：“原来如此。以前我选择技术栈全凭直觉，如‘我觉得 httpx 不错’，拿来就用。可真要问起缘由，我却说不出个所以然。现在，通过 ADR，决策背后的理由一目了然。”

“更重要的是，”小雪补充道，“ADR 不仅是给人看的，也是给 AI

看的。AI 在生成代码时，如果看到 ADR-001 中明确规定‘使用 httpx’，它就不会自作主张地去‘import requests’。可以说，ADR 有效约束了 AI 在技术选型上的自由度。”

“没错，”咖哥总结道，“这正是 ADR 在 SDD 模式下的独特价值。在传统项目中，ADR 主要是为团队成员服务的，确保有人问起‘为什么选用这项技术’时，有迹可循。而在 SDD 中，ADR 则多了一位重要的读者——AI。每一条 ADR，本质上都是向 AI 下达的一条技术约束指令。”

ThoughtWorks 技术雷达早在 2016 年便将 ADR 列为“采纳”级别的推荐实践。ADR 的核心价值，不在于它记录了什么，而在于它迫使你深入思考“为什么”。许多团队在完成技术选型后，往往难以清晰阐述其背后的理由——这恰恰暴露了选型过程本身缺乏足够的严谨性。

在 SDD 模式下，ADR 已不再是可选项，而是 design.md 中不可或缺的组成部分。每一个关键技术决策，都应当对应一份 ADR。那么，如何界定“关键”？判断标准其实很简单：**某个决策一旦失误，其修正成本将超过一天，该决策绝对值得被记录在 ADR 中。**

5.4 接口契约：模块之间的“握手协议”

5.4.1 为什么接口比实现更重要

架构确立了模块的边界，ADR 锁定了技术选型，但还有一个核心问题亟待解决：**模块之间究竟如何通信？**

答案在于，**模块的外部接口必须屏蔽其内部实现的细节**。模块间的交互应当基于“契约”——我向你提供何种格式的数据，你又需要返回何种格式的结果。至于你内部如何运转，我并不关心，你也不需要透露。

这一原则在 SDD 模式下显得尤其重要。因为 AI 生成的代码天然具有模块化特征，开发者指令 AI 编写采集层时，采集层不需要知晓生成层的内部逻辑，但**必须明确**生成层期望接收的数据格式。倘若采集层的输出与生成层的期望输入标准不匹配，这两个模块便如同说着不同语言

的人——各自虽能流利表达，却无法进行有效协作。

正因如此，**接口契约必须在 design.md 中明确定义**，且这一动作必须发生在编写任何一行代码之前。

5.4.2　数据模型定义

接口契约的首要组成部分是**数据模型**。该模型精确定义了在模块之间流动的数据的结构与形态（见图 5-6）。

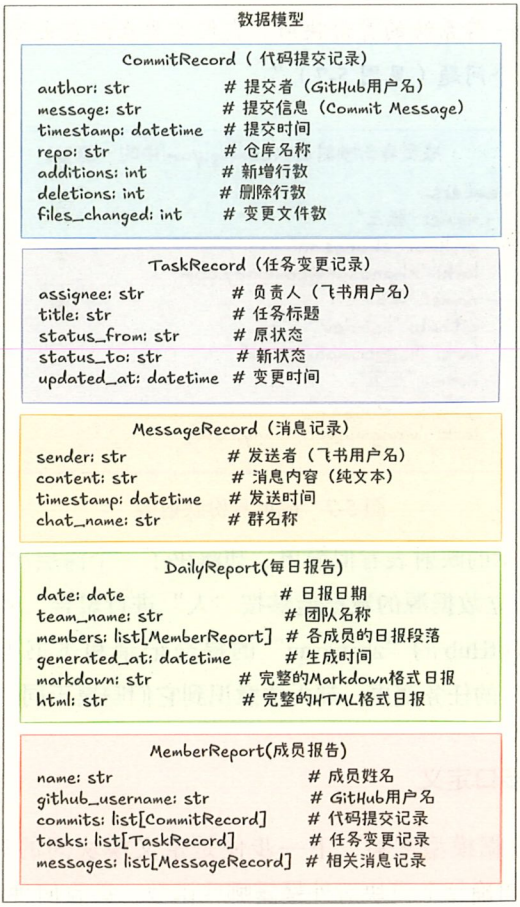


图 5-6　数据模型

数据模型的定义遵循一个核心原则：**只承载语义，不绑定实现**。上述定义明确了“一条提交记录应包含的字段”，但并未规定“这些字段在代码中是用 dataclass 还是 Pydantic Model 来表示”，后者属于实现细节，是留给 AI 自由裁量的范围。

小冰敏锐地捕捉到了一个细节：“CommitRecord 的 author 字段是 GitHub 用户名，但 TaskRecord 的 assignee 是飞书用户名。这两者该如何关联？同一个人在不同系统中可能使用完全不同的用户名。”

“观察得很仔细。”咖哥赞许道，“这正是数据模型设计中最容易被忽视的陷阱——**跨系统的身份映射**。我们需要在配置文件中维护一张映射表来解决这个问题（见图 5-7）。”

成员身份映射表（config.yaml中的一部分）

```yaml
members:
  - name: "张三"
    github: "zhangsan"
    lark: "zhangsan@company.com"
  - name: "李四"
    github: "lisi-dev"
    lark: "lisi@company.com"
  - name: "王五"
    github: "wangwu"
    lark: "wangwu@company.com"
```

图 5-7　成员身份映射表

图 5-7 所示的映射表看似简单，却解决了一个深层次的问题：它使得来自三个独立数据源的数据能够按“人”进行聚合。若没有这张表，你只会得到 GitHub 的“zhangsan”的提交记录和飞书上“zhangsan@company.com”的任务变更，却无法意识到它们归属于同一个体。

5.4.3　模块接口定义

确立了数据模型之后，下一步便是定义模块间的调用接口（见图 5-8）——明确每个模块对外暴露哪些函数、接收何种参数以及返回何种结果。

```
模块接口契约
# 采集层接口

github.collect(
    repos: list[str],          # 仓库列表
    since: datetime,           # 起始时间
    until: datetime            # 结束时间
) → list[CommitRecord]         # 返回提交记录列表

lark_task.collect(
    project_id: str,           # 飞书项目ID
    since: datetime,           # 起始时间
    until: datetime            # 结束时间
) → list[TaskRecord]           # 返回任务变更列表

lark_msg.collect(
    chat_id: str,              # 群ID
    keywords: list[str],       # 过滤关键词
    since: datetime,           # 起始时间
    until: datetime            # 结束时间
) → list[MessageRecord]        # 返回消息记录列表

# 生成层接口

generator.generate(
    members: list[MemberReport], # 各成员数据
    date: date,                  # 日报日期
    team_name: str               # 团队名称
) → DailyReport                  # 返回日报

# 推送层接口

email.send(
    report: DailyReport,       # 日报对象
    recipients: list[str]      # 收件人邮箱列表
) → bool                       # 成功/失败

lark_bot.send(
    report: DailyReport,       # 日报对象
    chat_id: str               # 目标群ID
) → bool                       # 成功/失败
```

图 5-8　模块接口契约

这些接口定义构建了一套简洁的“协议”：采集层负责输出标准化的 Record 列表，生成层接收 MemberReport 列表并产出 DailyReport，推送层则接收 DailyReport 并将其发送至指定目标。每个模块仅需要了解相邻模块的接口——采集层不需要感知推送层的存在，推送层也不需要关心数据的来源。

这正是 Parnas 所提出的**信息隐藏**（Information Hiding）原则的生动实践。每个模块都将自身的复杂性封装在接口背后——你不需要知晓

github.collect() 内部究竟是采用了分页查询还是 GraphQL，只需要明确给它提供仓库名和时间范围，它便会返回 list[CommitRecord]。

小雪总结了一个关键观察："接口契约本质上就是模块间的'规范'——采集层的'proposal'即其函数签名。若将 SDD 的思想递归应用，每个模块内部亦可遵循其专属的'proposal → design → code'流程。"

"这是一个非常深刻的见解，"咖哥回应道，"不过目前暂且不必过度递归——维持当前的粒度便已足够。"

5.5 非功能性约束：那些"看不见的需求"

5.5.1 非功能性需求为什么容易被忽略

在建筑行业中，有一类需求常被外行忽视，但专业建筑师绝不会遗漏——**消防通道、抗震等级、防水层、隔热层**。住户眼中所见，是客厅的装修与卧室的布局；而墙体内的钢筋、地基下的排水管，这些"看不见的东西"却恰恰决定了房屋的使用寿命。

软件系统亦是如此。proposal.md 中的验收标准通常聚焦于功能性需求——"能采集数据""能生成日报""能发送邮件"。然而，一个真正在生产环境中稳定运行的系统，还必须关注一整类**非功能性需求**（Non-Functional Requirements，NFR）。

系统的崩溃鲜少源于功能缺陷，几乎总是肇始于对非功能性维度（性能、安全、容错、可运维性）的疏忽。功能缺陷往往能在测试阶段被发现，而非功能性问题则常常要到生产环境中才会暴露。

5.5.2 智能日报生成器的非功能性约束

针对智能日报生成器，需要在 design.md 中明确定义以下非功能性约束。

首先是错误处理策略，如图 5-9 所示。

这里的核心思想是**优雅降级**（Graceful Degradation）——系统某一

部分出现故障时，不应导致整个系统停摆。这一原则源自容错系统设计的经典理念。航空工程师早已深谙此道：单台发动机故障不应导致整架飞机坠毁。这也是商用客机至少配备两台发动机的原因。

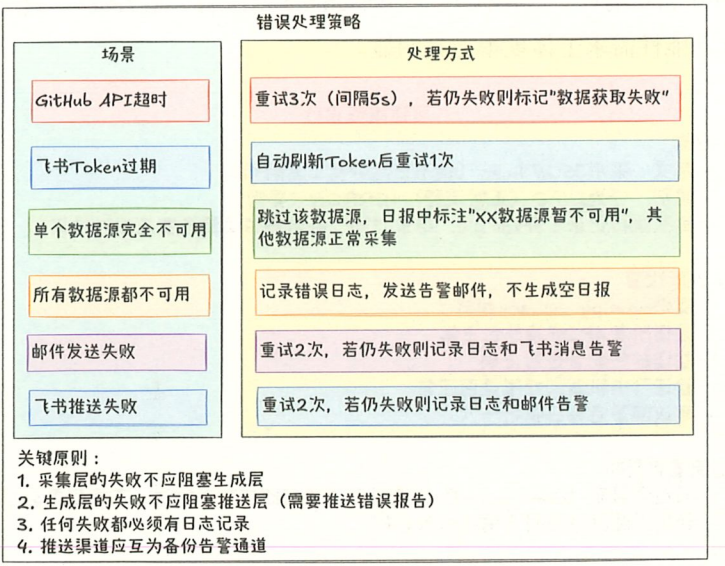


图 5-9　错误处理策略：优雅降级

其次是安全约束，如图 5-10 所示。

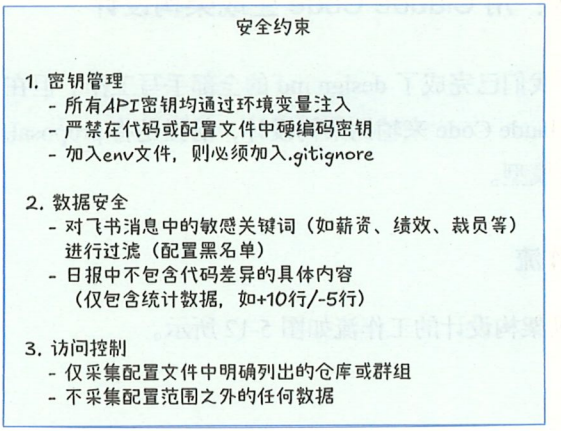


图 5-10　安全约束

最后是可运维性设计，如图 5-11 所示。

这些非功能性约束在 proposal.md 中可能仅体现为一句话（如“< 60s”“API 超时重试 3 次”），但在 design.md 中则被展开为具体的策略。proposal.md 阐释“做什么”，而 design.md 则说明“怎么做”。这种关系在非功能性需求上体现得尤其明显。

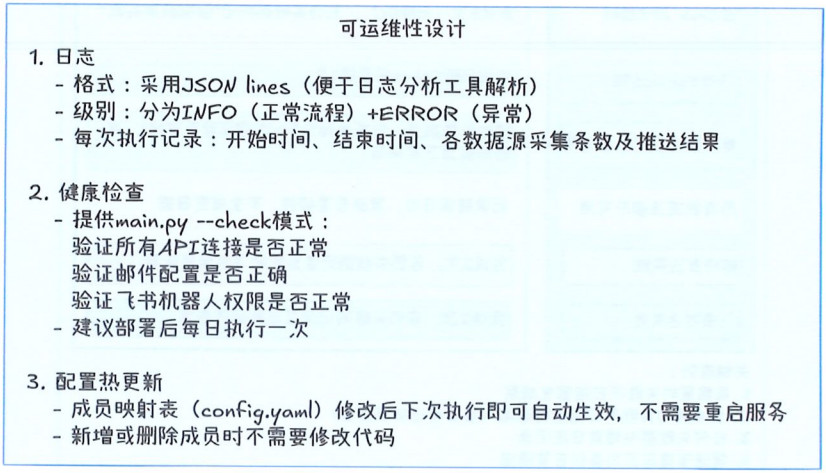


图 5-11　可运维性设计

5.6　实战：用 Claude Code 生成架构设计

至此，我们已完成了 design.md 的全部手写工作。但在实际开发中，可以利用 Claude Code 来辅助架构设计，前提是将 proposal.md 作为输入约束提供给模型。

5.6.1　工作流

AI 辅助架构设计的工作流如图 5-12 所示。

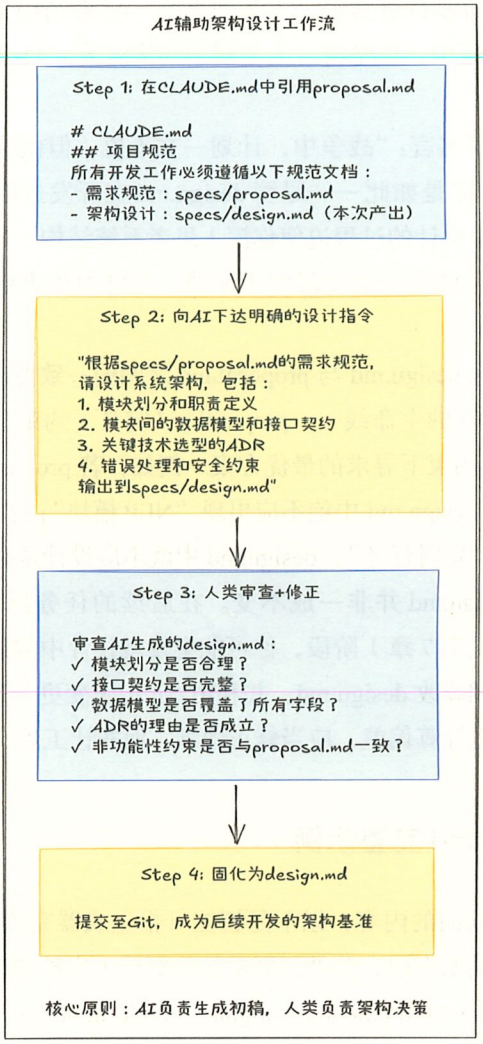


图 5-12　AI 辅助架构设计的工作流

5.6.2　生成过程中的关键注意点

第一，切勿让 AI 代为进行架构决策。你可以利用 AI 分析技术选项的优劣，但最终的技术选型、决策依据及权衡取舍，必须由人类架构师

主导。正如 proposal.md 遵循“人类做决策，AI 做结构化”的原则，在 design.md 的撰写中，应坚持“人类制定架构决策，AI 负责文档化”的原则。

拿破仑有句名言：“战争中，计划一文不值，但做计划的过程是一切。”架构设计亦是如此——尽管 design.md 在开发过程中可能会被修改甚至推翻，但设计的过程迫使你深入思考系统结构、模块边界及技术取舍。这些思考即便未完全落实于文档，也已内化为你对系统的深刻理解。

第二，确保 **design.md 与 proposal.md 之间的一致性**。两份文档之间的一致性是 SDD 的生命线——proposal.md 定义了约束条件，而 design.md 则是在这些约束下寻求的最优方案。例如，若 proposal.md 明确“不做智能摘要”，design.md 中绝不应出现“NLP 模块”；若 proposal.md 规定运行方式为“定时任务”，design.md 中就不应设计常驻 HTTP 服务。

第三，**design.md 并非一成不变**。在后续的任务拆解（见第 6 章）和编码实现（见第 7 章）阶段，你可能会发现设计中的遗漏或不合理之处。此时应回溯修改 design.md，并利用 Git 记录变更。这正如建筑师在施工中发现承重计算偏差，应当修正图纸，而非让工人“看着办”。

5.7 design.md 完整示例

整合本章前面的内容，以下是智能日报生成器完整的 design.md 结构概要。由于篇幅有限，此处仅展示结构概要（完整源代码请参阅咖哥 GitHub 仓库 https://github.com/huangjia2019/sdd-in-action）。

```
# 智能日报生成器 — 架构设计

## 1. 系统架构
- 管道式架构：采集层 → 生成层 → 推送层 + 共享基础层
- [架构图]

## 2. 模块职责
- collector/：数据采集（GitHub、飞书任务、飞书消息）
```

```
- generator/：日报生成（数据整理 + Markdown/HTML）
- notifier/：推送通知（邮件 + 飞书机器人）
- shared/：配置、日志、错误处理、存储
- main.py：编排入口

## 3. 数据模型
- CommitRecord / TaskRecord / MessageRecord
- MemberReport / DailyReport

## 4. 接口契约
- 采集层：collect(params) → list[Record]
- 生成层：generate(members, date, team) → DailyReport
- 推送层：send(report, target) → bool

## 5. 技术选型（ADR）
- ADR-001：HTTP 客户端 → httpx
- ADR-002：数据存储 → SQLite

## 6. 非功能性约束
- 错误处理：优雅降级策略
- 安全性：密钥管理 + 数据脱敏
- 可运维性：JSON 日志 + 健康检查 + 配置热更新
```

本章小结

在本章中，我们完成了 SDD 六阶段工作流的第二步：从 proposal.md（做什么）到 design.md（怎么做）。

架构设计的本质并非“绘制图表”，而是**在编码之前回答一系列关键的结构性问题**：系统如何划分模块，模块间如何通信，采用何种技术栈及决策依据，如何保障非功能性需求。如果这些问题未在 design.md 中明确，它们将以“AI 的默认选择”形式在代码中被隐式回答，而这些默认选择往往并非你所期望的。

本章涵盖了 design.md 的 4 个核心组成部分。

- **系统架构**：包括模块划分、职责定义及数据流向。设计需要遵循单一职责原则和关注点分离原则。

- **技术选型（ADR）**：记录每个关键决策的选项、理由及权衡。在 SDD 中，ADR 不仅是供人阅读的文档，更是指导 AI 的技术约束指令。
- **接口契约**：包含数据模型定义与模块接口签名，以确保独立开发的模块能够无缝协作。
- **非功能性约束**：涵盖错误处理、安全性和可运维性。功能性决定了系统是否可用，而非功能性决定了系统能否长久稳定运行。

在第 6 章中，我们将在 design.md 的基础上进行任务拆解——将宏大的架构设计拆解为一个个可执行、可验证的子任务，并编写 tasks.md。如果说 design.md 是将军的作战地图，那么 tasks.md 是各分队的行军指令。



咖哥发言

许多开发者认为“设计是架构师的事”，但在 AI 时代，每一位利用 AI 编写代码的开发者都扮演着架构师的角色。如果你不主动进行架构决策，AI 就会代劳，而它的决策既不会被记录、审查，也无法追溯。一个 600 行的“上帝对象”并非 AI 的过错，而是你未明确告知它应如何组织代码。design.md 正是你用来指示 AI“代码应当具备何种结构”的蓝图。

软件工程领域有句名言：“计算机科学中的所有问题，都可以通过增加一个中间层来解决。”这句话在当下的变体便是：**所有 AI 辅助编程的问题，都可以通过增加一层规范来解决。**

思考题

1. 本章的智能日报生成器采用了管道式架构。试想，如果 proposal.md 的需求发生变更，如新增“支持用户通过 Web 界面手动触发日报生成”，现有的架构是否依然能够支撑？若无法支撑，design.md 应如何

调整？是否需要在 ADR 中新增相关记录？

2. ADR-001 选择了 httpx 而非 requests，理由之一是“未来可平滑切换至异步模式”。然而，“面向未来设计”与“不过度设计”之间存在张力。应如何判断一个设计决策究竟是具备“合理的前瞻性”还是属于“过度设计”？
3. 本章在接口契约中定义了 CommitRecord、TaskRecord 等数据模型。然而，若未来需要新增数据源（如 Confluence Wiki 的编辑记录），数据模型应如何扩展？现有的模块职责划分与接口契约是否具备相应的扩展能力？若不具备，是否意味着 design.md 中存在遗漏？

第 6 章
化整为零：任务拆解与执行编排

千里之行，始于足下——任何宏大的架构，最终都要拆解为可执行的任务，一步步完成。

小冰盯着屏幕上的 design.md，越看越兴奋——模块划分清晰，接口契约明确，技术选型有理有据。这份架构设计让她对即将构建的智能日报生成器充满信心。

“太好了，我要开始写代码了！”小冰迫不及待地打开 Claude Code，准备将整份 design.md 一股脑儿地丢给 AI。

“慢。”咖哥拦住了她，“你打算让 AI 一次性把整个系统都写出来？”

“对啊。design.md 已经足够详细了，模块、接口、数据模型一应俱全。AI 应该能一次搞定。”

咖哥笑了笑，让小冰想象这样一个场景：一位项目经理拿到了一份堪称完美的建筑蓝图——楼层平面、管线走向、承重结构，一应俱全。然后，他把蓝图往工地一扔，对施工队说：“按这个图，把楼给我盖出来。”

“这肯定不行吧？”小冰立刻反应过来，“得先排施工计划。先打地基，再起框架，然后砌墙，最后装修——工序有先后，不能乱来。”

“没错。”咖哥说，“design.md 是蓝图，但蓝图不等于施工计划。从蓝图到竣工，中间还需要一份施工进度表——把大项目拆解为一个个可执行的施工步骤，定义每一步的完成标准，安排步骤之间的先后顺序。在 SDD 中，这份施工进度表叫作 tasks.md。”

他在白板上绘制了 proposal.md、design.md 和 tasks.md 这三个规范文件的关系。

“如果说在第 5 章中我们画好了作战地图，”咖哥总结道，“那么，在本章中我们要做的，就是把地图变成行军命令——明确每个连队走哪条路，什么时候到达哪个地点，到了之后怎么确认任务完成。”

6.1　从 design.md 到 tasks.md：SDD 的第三份规范

6.1.1　三份规范的递进关系

到目前为止，我们已经创建了两份核心规范文档。proposal.md 回答了“做什么”等问题；design.md 回答了“怎么做”等问题，包括模块如何划分、接口如何定义、技术如何选型等。但还有一些关键问题尚未解决：按照什么顺序做，每一步的完成标准是什么。

解决这些问题正是 tasks.md 的职责所在。这三份规范文档共同形成了一条从抽象到具体的递进链，如图 6-1 所示。

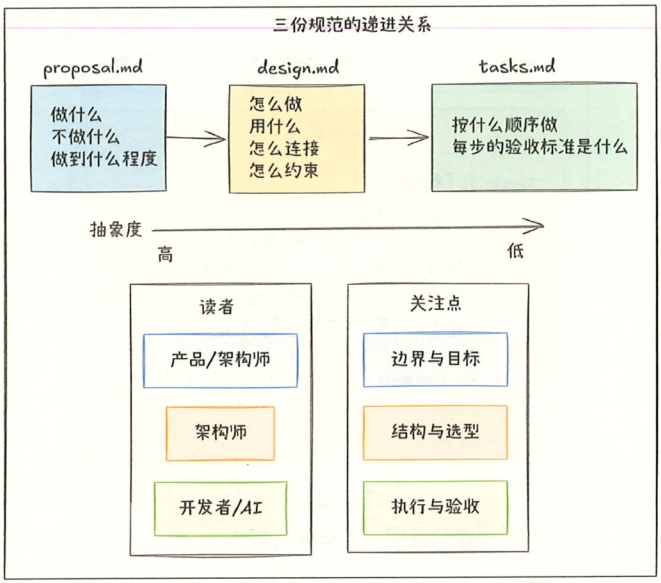


图 6-1　三份规范的递进关系

每一份文档都有其特定的读者与目的。proposal.md 旨在帮助人和 AI 理解项目边界；design.md 主要阐释系统结构；而 tasks.md 则主要帮助理解具体行动——精确到“现在该写哪个文件”“写完之后如何验证”。

小冰问：“那 tasks.md 是否就是项目管理中的工作分解结构（Work Breakdown Structure，WBS）？”

“本质是一样的，”咖哥回答道，“但 tasks.md 有一个传统 WBS 没有的读者——AI。传统的 WBS 只需要人能看懂就行，而 tasks.md 还需要 AI 能看懂。这意味着每个任务的描述都必须足够具体、无歧义，使得 AI 读完任务描述就能开始编写代码，写完代码就能用验收标准进行自我检查。”

6.1.2 tasks.md 的基本结构

一份优秀的 tasks.md 通常包含三个层次，如图 6-2 所示。

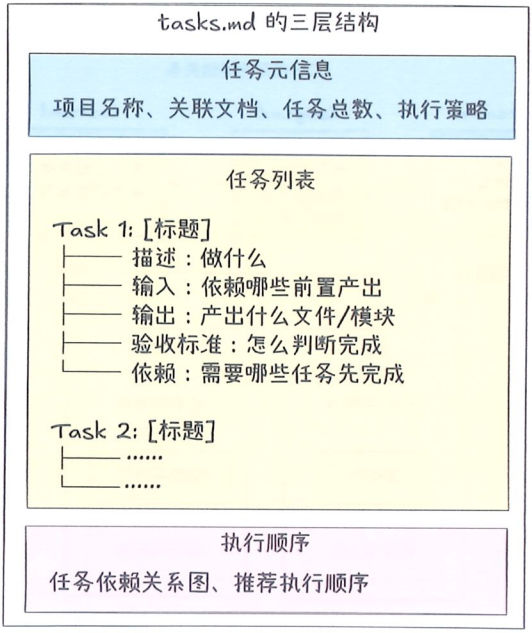


图 6-2 tasks.md 的三层结构

每一个任务都像一份独立的“施工单”：明确告诉执行者（无论是 AI 还是人）具体要做什么、需要满足哪些前置条件、预期产出什么结果，以及如何判定任务已完成。

6.2　任务粒度：不大不小的艺术

6.2.1　粒度的两个极端

在任务拆解过程中，最艰难的决策并非“拆成几个”，而是“每个任务多大”。

如果粒度偏粗，一个任务覆盖了半个系统，如“实现整个采集层”。这对人类开发者来说或许仍可应对，但对 AI 而言堪称严峻挑战：上下文窗口无法容纳所有代码，AI 会在中途“忘记”接口契约、数据模型和错误处理策略，最终生成的代码与 design.md 南辕北辙。

反之，如果粒度偏细，一个任务仅涉及函数的一行代码，如“给 CommitRecord 添加 timestamp 字段”，虽然每个任务都微不足道，但任务数量会急剧增长至成百上千个，导致管理成本远超执行成本。

图 6-3 展示了任务粒度光谱及最佳区间。

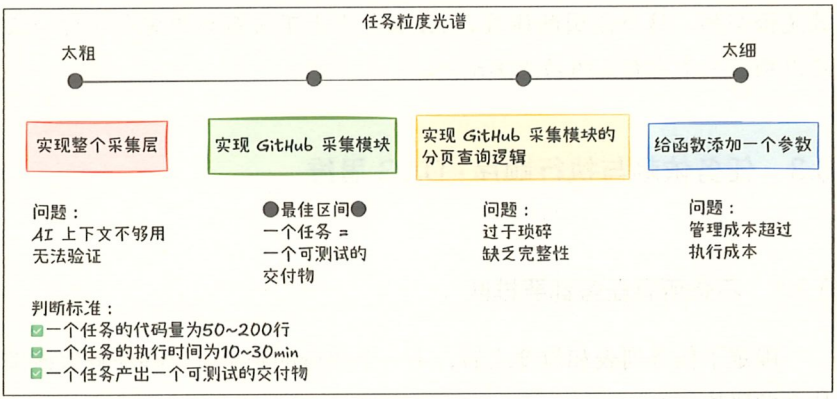


图 6-3　任务粒度光谱及最佳区间

6.2.2 黄金法则：一个任务 = 一个可验证的交付物

什么样的粒度才算“合适”？咖哥给出了一个简明扼要的判断标准：

一个任务应该产出一个可独立运行或独立测试的交付物。

“可独立测试”是其中的核心。如果一个任务完成后，你无法编写一个简单的测试来验证它——这通常意味着任务要么过大（内部混杂了多个职责不同的功能），要么过小（本身无法构成一个有意义的单元）。

针对智能日报生成器，可以参考表 6-1 来评估粒度是否恰当。

表 6-1 任务粒度评估示例

粒度	任务示例	能否独立测试	评价
偏粗	实现整个采集层（三个数据源）	难以单独测试，混合了多个 API	应拆分
合适	实现 GitHub 采集模块	可以用 Mock 数据独立测试	推荐
偏细	实现 GitHub API 的身份认证	认证本身无法独立产出	应合并

小冰总结道：“所以，一个优秀的任务，其粒度应控制在‘一个文件’或‘一个模块’的级别——既不会大到导致 AI 上下文溢出，也不会小到缺乏独立价值。”

“没错。”咖哥肯定道，“请记住这条经验法则：**如果一个任务的代码量为 50~200 行，且需要一个对应的测试文件来验证，那么这个粒度就是恰当的。**这个范围既能确保 AI 在单个上下文窗口内完成工作，又足以构成一个有意义的功能单元。”

6.3 任务依赖与执行顺序：DAG 思维

6.3.1 不是所有任务都要排队

确定了任务列表和粒度之后，下一个关键问题是：**这些任务应该按什么顺序执行？**

最符合直觉的方式是排一个线性序列：任务 1 → 任务 2 → 任务

3 →…→任务 *N*。但这往往不是最优解——因为许多任务之间并不存在依赖关系，它们完全可以并行执行。

在计算机科学领域，描述这种“部分任务必须串行，部分任务可以并行”的结构，有一个精确的数学模型：**有向无环图**（Directed Acyclic Graph，DAG）。

DAG 的规则非常直观。

- 每个节点代表一个任务。
- 有向边代表依赖关系，A → B 意味着“A 必须在 B 之前完成”。
- 图中不存在环（即不存在循环依赖）。
- 没有边连接的两个任务，意味着它们互不依赖，可以并行执行。

6.3.2　智能日报生成器的任务依赖图

根据 design.md 的模块划分，我们可以为智能日报生成器绘制出任务依赖图。如图 6-4 所示，部分任务存在明确的先后制约关系（例如，“共享基础层”必须在“采集模块”之前完成），而另一些任务则互不干扰，可以并行推进（如三个采集模块）。

图 6-4 揭示了几个关键信息。

第一，**关键路径**（Critical Path）：从 Task 1 到 Task 10 的最长路径决定了项目完成所需的最短时间。这条路径上的任何任务延迟，都会直接导致整个项目延期。

第二，**并行机会**：Task 3、Task 4、Task 5 可以同时执行，Task 7、Task 8 也可以并行执行。如果你能同时调度三个 AI 上下文窗口（如使用子智能体），那么采集层的开发时间理论上可以缩短至原来的三分之一。

小冰看着这张依赖图说：“这不就是操作系统里的任务调度吗？关键路径分析、并行执行——大学课堂上学过。”

“没错，”咖哥说，“大学里学的是 CPU 的指令调度，而在这里，我们调度的是 AI 的‘注意力’。核心原理完全一样——分析依赖关系、最大化并行度、缩短关键路径。唯一的区别在于，CPU 的调度是由操作系统自动完成的，而 tasks.md 中的调度策略，则需要你精心设计。”

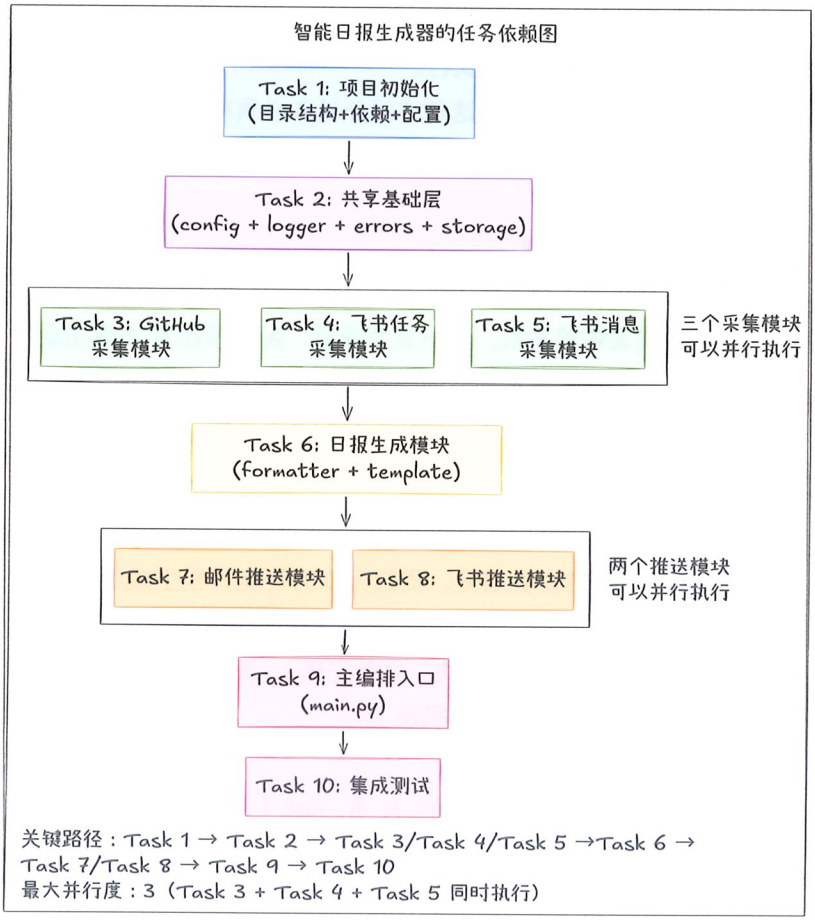


图 6-4　智能日报生成器的任务依赖图

6.4　验收标准：每个任务的“完成”定义

6.4.1　没有验收标准的任务等于没有终点

拆解好任务、排好顺序，还差最后一块拼图：每个任务的验收标准。

验收标准回答了一个看似简单实则关键的问题：**你怎么知道这个任务“做完了”？**

如果没有验收标准，任务的完成与否就变成了主观判断——“我觉得写完了”。这在人类开发中已经是个问题（“你说的‘完成’是什么意思？”是项目管理中最经典的争执），在 AI 辅助开发中问题更大：AI 没有“我觉得”的能力，它需要明确的、可检查的完成条件。

6.4.2　优秀的验收标准长什么样

一份优秀的验收标准通常具备三个核心特征：可执行、可观测、二值化。

- **可执行**：可以通过运行代码或测试来验证，不需要人工进行主观判断。
- **可观测**：验证结果是清晰可见的，例如，测试通过、输出正确、日志符合特定格式等。
- **二值化**：结果只有“通过”和“不通过”两种状态，不存在“差不多通过”的模糊地带。

图 6-5 对比了优秀的验收标准与低劣的验收标准。

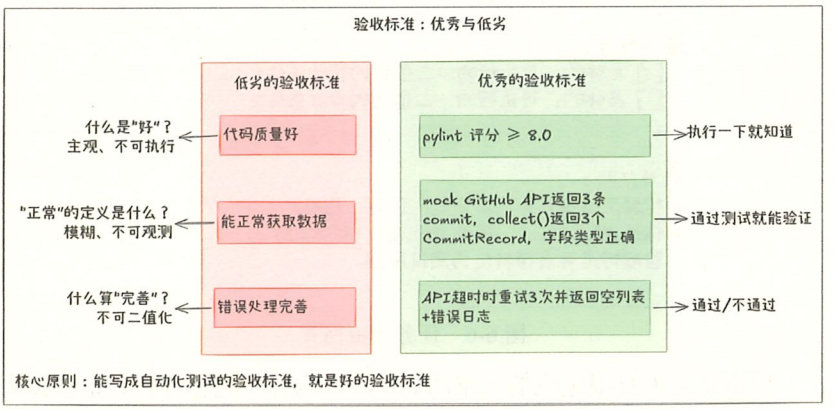


图 6-5　验收标准对比

小雪敏锐地捕捉到了一个关键模式：“优秀的验收标准，本质上就

```
# 智能日报生成器 — 任务清单

## 元信息
- 关联规范：specs/proposal.md、specs/design.md
- 任务总数：10
- 预计总执行时间：3 到 4h（包含测试）
- 执行策略：按依赖关系顺序执行，独立任务可并行执行

---

## Task 1：创建项目结构和基础配置

描述：初始化项目目录结构，创建 pyproject.toml，安装核心依赖，创建配置文件模板

输入：specs/design.md（模块职责定义）
输出：项目根目录结构 +pyproject.toml+config.yaml.example
依赖：无

验收标准：
- [ ] 目录结构符合 design.md §2 的模块划分
      （collector/、generator/、notifier/、shared/、tests/）
- [ ] pyproject.toml 包含 httpx、pyyaml、jinja2 依赖
- [ ] uv sync 成功安装所有依赖
- [ ] config.yaml.example 包含 members 映射表模板

---

## Task 2：实现共享基础层

描述：实现配置读取、日志记录、错误处理和 SQLite 存储 4 个共享模块

输入：config.yaml.example（配置格式）、design.md §6（非功能约束）
输出：shared/config.py、shared/logger.py、shared/errors.py、shared/
storage.py
依赖：Task 1

验收标准：
- [ ] config.py 能读取 config.yaml 并返回配置对象
- [ ] config.py 在配置缺少必填字段时抛出明确的错误信息
- [ ] logger.py 输出 JSON lines 格式日志
```

- [ ] errors.py 定义 CollectorError、GeneratorError、NotifierError 三个自定义异常
- [ ] storage.py 能创建 SQLite，写入和查询日报记录
- [ ] 所有模块包含对应的单元测试，测试通过

Task 3：实现 GitHub 采集模块

描述：调用 GitHub API 获取指定仓库在指定时间范围内的 Commit 记录，返回 CommitRecord 列表

输入：design.md §3（CommitRecord 数据模型）、design.md §4.1（采集层接口契约）
输出：collector/github.py+tests/test_github.py
依赖：Task 2

验收标准：
- [ ] collect() 函数签名符合 design.md §4.1 的接口定义
- [ ] 返回的每个 CommitRecord 包含全部 7 个字段且类型正确
- [ ] 支持分页查询（GitHub API 默认每页 30 条）
- [ ] API 超时重试 3 次（间隔 5s），若仍失败则返回空列表 + 错误日志
- [ ] GitHub API 限流（HTTP 403）时，等待 reset 时间后重试
- [ ] 使用 Mock 数据的单元测试全部通过

Task 4：实现飞书任务采集模块

描述：调用飞书开放平台 API 获取指定项目中当日状态变更的任务列表，返回 TaskRecord 列表

输入：design.md §3（TaskRecord 数据模型）、design.md §4.1（采集层接口契约）
输出：collector/lark_task.py+tests/test_lark_task.py
依赖：Task 2

验收标准：
- [ ] collect() 函数签名符合 design.md §4.1 的接口定义
- [ ] 返回的每个 TaskRecord 包含全部 5 个字段且类型正确
- [ ] 飞书 Token 过期时自动刷新后重试 1 次

- [ ] API 超时重试 3 次，若仍失败则返回空列表 + 错误日志
- [ ] 使用 Mock 数据的单元测试全部通过

Task 5：实现飞书消息采集模块

描述：调用飞书开放平台 API 获取指定群的消息记录，按关键词过滤，返回 MessageRecord 列表

输入：design.md §3（MessageRecord 数据模型）、design.md §4.1（采集层接口契约）
输出：collector/lark_msg.py+tests/test_lark_msg.py
依赖：Task 2

验收标准：
- [ ] collect() 函数签名符合 design.md §4.1 的接口定义
- [ ] 返回的每个 MessageRecord 包含全部 4 个字段且类型正确
- [ ] 关键词过滤正确，只返回包含指定关键词的消息
- [ ] 敏感关键词（薪资、绩效等）被正确过滤，不出现在结果中
- [ ] 飞书 Token 过期时自动刷新后重试 1 次
- [ ] 使用 Mock 数据的单元测试全部通过

Task 6：实现日报生成模块

描述：接收各成员的采集数据，按“代码提交→任务进展→协作沟通”组织日报内容，输出 Markdown 和 HTML 格式文件

输入：design.md §3（MemberReport、DailyReport 数据模型）、design.md §4.2（生成层接口契约）
输出：generator/formatter.py、generator/template.py+tests/test_generator.py
依赖：Task 2

验收标准：
- [ ] generate() 函数签名符合 design.md §4.2 的接口定义
- [ ] 输出的 DailyReport.markdown 包含三个部分标题
- [ ] 输出的 DailyReport.html 可被浏览器正确渲染

```
- [ ] 若某成员当日无任何记录，日报中需要显示“今日无记录”
- [ ] 若某数据源采集失败，日报中需要标注“数据获取失败”
- [ ] Markdown 到 HTML 的转换格式正确
- [ ] 使用 Mock 数据的单元测试全部通过

---

## Task 7：实现邮件推送模块

描述：将 HTML 格式日报通过 SMTP 发送给指定收件人

输入：design.md §4.3（推送层接口契约）
输出：notifier/email.py+tests/test_email.py
依赖：Task 6

验收标准：
- [ ] send() 函数签名符合 design.md §4.3 的接口定义
- [ ] 邮件主题包含日期和团队名称
- [ ] 邮件正文为 HTML 格式的日报内容
- [ ] 发送失败时重试 2 次，仍失败则返回 False+ 错误日志
- [ ] 使用 Mock SMTP 的单元测试通过

---

## Task 8：实现飞书推送模块

描述：将 Markdown 格式日报通过飞书机器人 webhook 发送到指定群

输入：design.md §4.3（推送层接口契约）
输出：notifier/lark_bot.py+tests/test_lark_bot.py
依赖：Task 6

验收标准：
- [ ] send() 函数签名符合 design.md §4.3 的接口定义
- [ ] 发送的消息为 Markdown 格式
- [ ] 发送失败时重试 2 次，仍失败则返回 False+ 错误日志
- [ ] 使用 Mock webhook 的单元测试通过

---
```

```
## Task 9：实现主编排入口

描述：创建 main.py 作为系统入口，按顺序编排“采集→聚合→生成→推送”的完整流程

输入：所有模块的接口
输出：main.py
依赖：Task 3、Task 4、Task 5、Task 6、Task 7、Task 8

验收标准：
- [ ] python main.py 执行完整的日报生成流程
- [ ] python main.py --check 验证所有 API 连接和配置
- [ ] python main.py --dry-run 执行采集和生成但不推送
- [ ] 若单个数据源失败，不应影响其他数据源的采集
- [ ] 若所有数据源失败，不生成空日报，记录错误日志
- [ ] 执行日志包含：开始时间、各数据源数据采集条数、推送结果、结束时间


---

## Task 10：集成测试

描述：编写端到端集成测试，验证完整流程在各种场景下的行为

输入：main.py+ 所有模块
输出：tests/test_integration.py
依赖：Task 9

验收标准：
- [ ] 正常场景：所有数据源可用，日报生成并推送成功
- [ ] 降级场景：若某个数据源超时，日报中需要标注“数据获取失败”，其他部分正常
- [ ] 空数据场景：若某成员当日无任何记录，日报中需要显示“今日无记录”
- [ ] 全部失败场景：若所有数据源不可用，不生成日报，记录错误日志
- [ ] 执行耗时少于 60s（Mock 环境下）

---

## 执行顺序
```

推荐执行顺序（考虑依赖关系和并行机会）：

阶段 1：Task 1（项目初始化）
阶段 2：Task 2（共享基础层）
阶段 3：Task 3+Task 4+Task 5 并行（三个采集模块）
阶段 4：Task 6（日报生成模块）
阶段 5：Task 7+Task 8 并行（两个推送模块）
阶段 6：Task 9（主编排入口）
阶段 7：Task 10（集成测试）

小冰通读了整个 tasks.md 后说："10 个任务，每一个都定义了清晰的输入、输出和验收标准。最令我惊讶的是，虽然这份文档里没有一行代码，但我脑海中已经能清晰地构建出代码的形态了。"

"这就是规范的力量。"咖哥回应道，"规范并不规定'如何实现'——那是 AI 的工作。规范界定的是'交付结果必须满足的条件'。只要满足了这些条件，代码的具体写法就不再是关键——或者更准确地说，凡是满足条件的代码，就是正确的代码。"

6.6 实战：用 Claude Code 逐任务执行

6.6.1 执行工作流

有了 tasks.md，接下来的步骤便是执行。在 SDD 语境下，执行过程遵循一个简洁的闭环逻辑：获取任务→ AI 实现→验证通过→获取下一个任务。具体流程如图 6-7 所示。

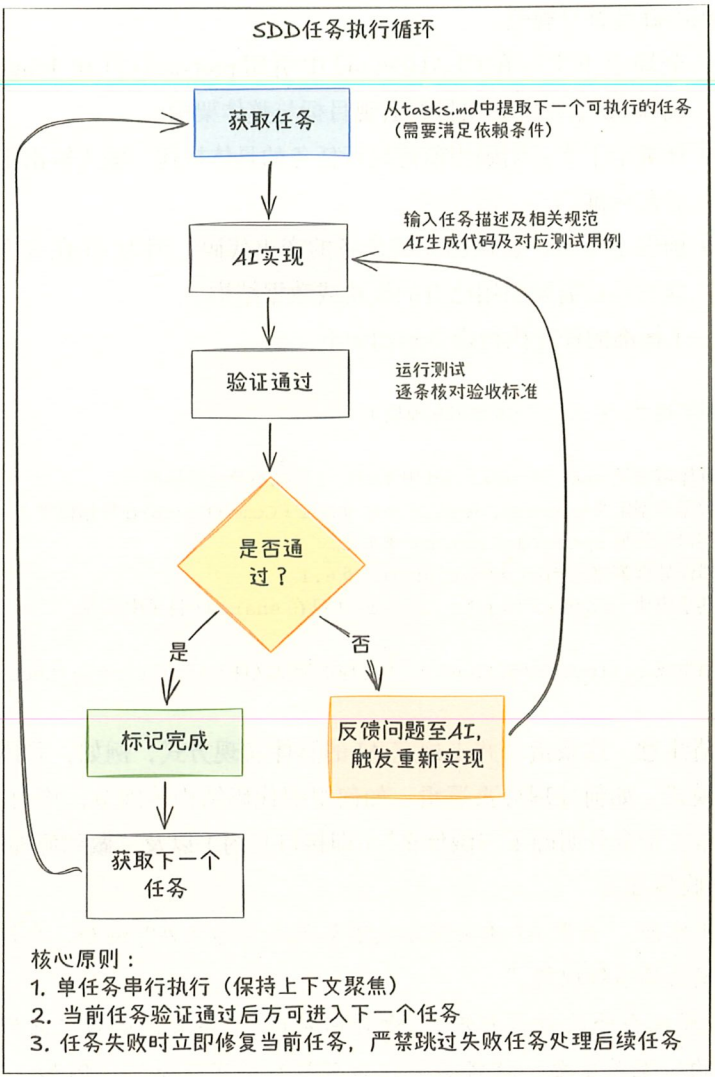


图 6-7　SDD 任务执行循环

6.6.2　给 AI 的上下文

在执行任务时，需要为 AI 构建三个层级的上下文，以确保其生成

的代码准确且符合规范。

- 全局上下文：在 CLAUDE.md 中引用 proposal.md 和 design.md，确保 AI 始终掌握项目的宏观目标与整体架构。
- 任务上下文：明确当前待执行任务的具体描述、输入输出定义及验收标准。
- 前置上下文：提供已完成任务的产出代码，因为 AI 在实现新功能时往往需要引用已有的模块或数据结构。

一个标准的任务执行指令示例如下。

```
请实现 Task 3（GitHub 采集模块）。

具体需求见 specs/tasks.md 中 Task 3 的描述及验收标准。
数据模型定义见 specs/design.md §3.1（CommitRecord 数据模型）。
接口契约见 specs/design.md §4.1。
错误处理策略见 specs/design.md §6.1。
共享模块（config、logger、errors）已在 shared/ 目录中实现。

请生成 collector/github.py 及对应的测试文件 tests/test_github.py。
```

请注意，这条指令并未规定 AI 的具体实现方式，例如，采用何种设计模式、如何处理分页逻辑、如何组织代码结构等细节，均由 AI 自主决策。指令仅明确了“做什么”（即接口契约）以及“做到何种程度”（即验收标准）。

小冰问：“如果 AI 的实现方式不尽如人意怎么办？例如，我认为它的代码结构不够理想。”

“那就去修改验收标准。”咖哥回答道，“如果你认为‘良好的代码结构’至关重要，就将其转化为可执行的客观标准，例如，‘每个公开函数必须包含 docstring’‘单个函数代码行数不超过 30 行’‘使用 dataclass 替代字典’。将主观偏好转化为客观标准，这正是 SDD 的核心思维方式。”

6.6.3　验证与迭代

任务执行完毕后，需要依据验收标准逐项进行核查。最高效的验证手段便是运行测试。

```
# 运行 Task 3 的测试
python -m pytest tests/test_github.py -v
```

若所有测试均通过，则该任务宣告完成。若出现测试失败，应将具体的失败信息反馈给 AI，由其进行修复。通常，这一修复循环不会超过 3 轮——这得益于验收标准具备足够的颗粒度与具体性，使得 AI 能够迅速理解“偏差所在”并进行纠正。

6.7　实战：用子智能体并行执行

6.7.1　为什么要并行执行

回顾图 6-4 中的任务依赖图，我们不难发现部分任务具备并行执行的条件——Task 3、Task 4、Task 5 三个采集模块之间互不依赖，Task 7 和 Task 8 两个推送模块亦是如此。若采用串行模式执行，共需要 10 个步骤；若采用并行模式执行，则仅需要 7 个步骤（即 7 个阶段）即可完成。

在 Claude Code 中，并行执行是通过子智能体来实现的。你可以通过单条指令同时启动多个子智能体，让它们各自负责一个独立的任务，从而显著提升开发效率。

6.7.2　并行执行的安全边界

并行执行有一个核心前提：**并行任务之间绝不能存在共享状态的冲突。**

以“智能日报生成器”项目为例，Task 3、Task 4、Task 5 是安全的

并行候选对象，因为它们分别生成独立的文件（github.py、lark_task.py、lark_msg.py），既不会修改同一个文件，也不涉及共享可变状态。

然而，如果两个任务需要修改同一个文件，例如，两个任务都要向 shared/errors.py 添加新的异常类型，那么并行执行极有可能引发冲突。在这种情况下，要么将这两个任务合并为一个，要么安排它们串行执行（即完成一个后，再开始另一个）。

表 6-2 总结了判断任务能否并行执行的简单规则。

表 6-2　任务并行执行的判断规则

条件	并行执行	原因
两个任务产出不同的文件	可以	没有文件写入冲突
两个任务读取同一个模块但不修改	可以	只读操作不会产生冲突
两个任务修改同一个文件	不可以	可能产生代码合并冲突
一个任务的输出是另一个任务的输入	不可以	存在数据依赖关系

6.7.3　并行执行的工作流

如图 6-8 所示，利用子智能体进行并行执行的工作流遵循“串行—并行—汇总”模式：首先完成所有前置依赖任务，随后同时启动一组相互独立的任务；待所有子智能体执行完毕并汇总验证通过后，再进入下一阶段。

小雪总结道：“tasks.md 不只是一份任务清单——它定义的依赖关系，实际上就是并行执行的调度策略。DAG 既是任务拆解的思维工具，也是并行执行的调度蓝图。一份优秀的 tasks.md，既是给人看的计划书，也是给 AI 看的调度指令。”

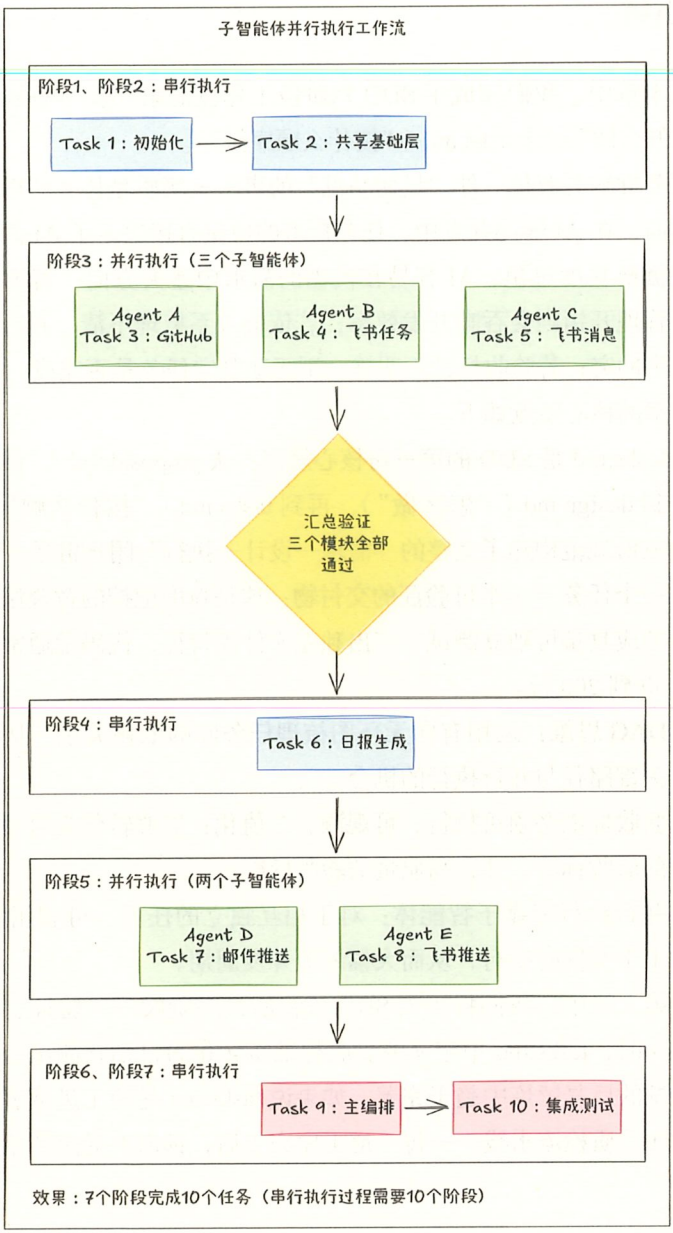


图 6-8　子智能体并行执行工作流

本章小结

在本章中，我们完成了 SDD 六阶段工作流的第三步——从 design.md（“怎么做”）到 tasks.md（“按什么顺序做”）。

任务拆解看似是一件“显而易见”的事——实质是任务的拆解与细化。然而，在 AI 辅助开发中，任务拆解的质量直接决定了 AI 的产出质量。若拆解粒度过粗，AI 容易在模糊的需求中迷失方向；若拆解粒度过细，管理开销则会吞噬开发效率；若依赖关系梳理不清，并行执行时便会产生冲突；若验收标准不明确，则无法判定任务是否完成。

本章的核心要点如下。

- **tasks.md 是 SDD 的第三份核心规范**：从 proposal.md（“做什么”）到 design.md（“怎么做”），再到 tasks.md（“按什么顺序做”），三份规范构建了完整的“需求→设计→执行”闭环链条。
- **一个任务 = 一个可验证的交付物**：这是粒度把控的黄金法则。任务应具备可独立测试、产出独立文件等特征，代码量通常控制在 50 到 200 行。
- **DAG 思维**：运用有向无环图梳理任务间的依赖关系，从而识别关键路径与并行执行的机会。
- **验收标准必须可执行、可观测、二值化**：能够转化为自动化测试的验收标准，才是高质量的验收标准。
- **并行执行依靠子智能体**：对于相互独立的任务，可利用多个 AI 上下文同时执行，从而大幅缩短开发周期。

在第 7 章中，我们将迈入 SDD 流程的最终阶段——验证、迭代与演进。届时，tasks.md 中定义的验收标准将转化为自动化测试，代码也将在测试的反复锤炼中趋于完善。如果说 tasks.md 是施工进度表，那么第 7 章便是质检流水线——每一道工序完工后，都需要通过严格的质检关卡。



咖哥发言

软件项目最常见的失败模式并非“做不出来”，而是“做着做着就散了”——需求理解出现偏差、模块接口无法对接、测试覆盖不足、进度失去控制。tasks.md 的价值不在于罗列了多少任务，而在于它将一个模糊的“要做一个系统”的愿景，转化为一连串具体的行动指南：“先做这个，再做那个，每步检查这些条件”。这种从模糊到具体的转化能力，才是工程素养的核心体现。

19 世纪普鲁士与德意志帝国的军事战略家老毛奇曾有一句广为流传的名言：“没有任何作战计划在与敌人接触后还能幸存。”但这并不意味着不需要计划。**计划的价值不在于其完美无缺，而在于它为你提供了一个可以调整的基准。**如果没有 tasks.md，你甚至连“计划变了”都无法察觉——因为你从未拥有过计划。

思考题

1. 本章将智能日报生成器拆解为 10 个独立任务。试想，如果你是单人开发（不启用子智能体），你会严格遵循 Task 1 → Task 2 → Task 3 →⋯→ Task 10 的线性顺序执行吗？还是会进行调整？背后的原因是什么？
2. Task 6（日报生成模块）的验收标准中规定：“若某数据源采集失败，日报中需要标注‘数据获取失败’”。这意味着 Task 6 必须能够区分数据源是“正常返回空数据”还是“采集失败”。然而，目前的采集层接口仅返回 list[Record]，无法区分这两种状态。你会如何修改接口契约来解决这一问题？（提示：这正是从 tasks.md 反馈至 design.md 的一个真实案例。）
3. 表 6-2 给出了判断任务能否并行执行的规则，但有一种情况并未被涵盖：当两个任务都需要调用同一个外部 API（如飞书 API），且该 API 存在请求频率限制时，应如何处理？这种情况是否构成一种“依赖”？如果构成，在编写 tasks.md 时又应如何应对？

第 7 章
千锤百炼：验证、迭代与演进

规范并非写完即止的静态文档，而是与代码同呼吸、共进化的活态工程制品。

小冰严格遵循 tasks.md 逐个执行任务，每完成一个便标记为已完成。一周之后，10 个任务全部完成，运行 python -m pytest 显示所有测试通过——智能日报生成器的 1.0 版本成功运行。

她长舒一口气："终于完成了。"

两天后，运营负责人找上门来："日报能否增加每位成员的工时统计功能？飞书平台上有考勤数据。"

小冰打开代码，准备在 collector/ 目录下添加新模块。但写到一半时她愣住了——proposal.md 中从未提及考勤数据，design.md 的数据模型中也没有相应字段，tasks.md 更未涉及。她该如何处理？直接编写代码还是先修改规范文档？修改哪份规范文档？修改规范后，已有的代码与测试又该如何处理？

"这才是真正的考验。"咖哥坐了过来，"前面 6 章我们学习了如何从零开始实施 SDD。然而，真实的项目开发并非一蹴而就——需求会变更，bug 会出现，新功能会不断加入。一套优秀的规范体系，不仅要能支撑首次开发，还要能支撑项目的持续演进。"

他用一个建筑来比喻说明。

"你见过那些屹立百年的老建筑吗？它们并非建成后便一成不变——管线可能翻新过，墙面可能加固过，楼层甚至可能加建过。但关键在于，每一次改动都有对应的施工图纸、结构计算和验收报告。正是

这种‘每次改动都留下记录’的严谨习惯，才使得一栋建筑能够历经百年风雨和反复改造而屹立不倒。

“SDD 亦是如此——修改代码并不可怕，可怕的是修改了代码却不同步更新规范。本章将介绍如何确保规范与代码协同演进。”

7.1　规范驱动的测试策略

7.1.1　从验收标准到测试用例

在第 6 章中，我们为 tasks.md 中的每个任务编写了验收标准。这些标准不仅是供人阅读的核对清单，更是自动化测试用例的直接蓝图。图 7-1 展示了验收标准与测试用例的映射。

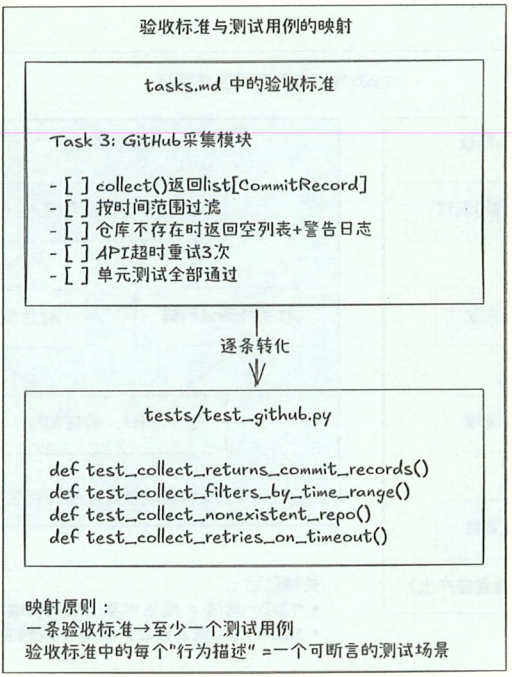


图 7-1　验收标准与测试用例的映射

这种映射关系并非机械式的代码翻译，而是对需求的深度解析。我们需要从验收标准中提炼出**可测试的行为描述**，进而为每一个行为构建具体的测试场景。

W. Edwards Deming 曾有一句被奉为质量管理圭臬的名言："质量是设计出来的，而非检验出来的。"在 SDD 语境下，这句话有了新的诠释：**测试不是最后补上的保障，而是从规范中自然衍生出的产物**。当你撰写验收标准时，实际上就已经在设计测试了——此时它们虽尚未转化为代码，但逻辑已然成形。

7.1.2 让 AI 从规范生成测试

这正是 SDD 与 TDD 的关键差异所在。TDD 遵循"先写测试，再写实现"的原则；而 SDD 则主张"先写规范，再从规范中同时生成测试和实现"（见图 7-2）。

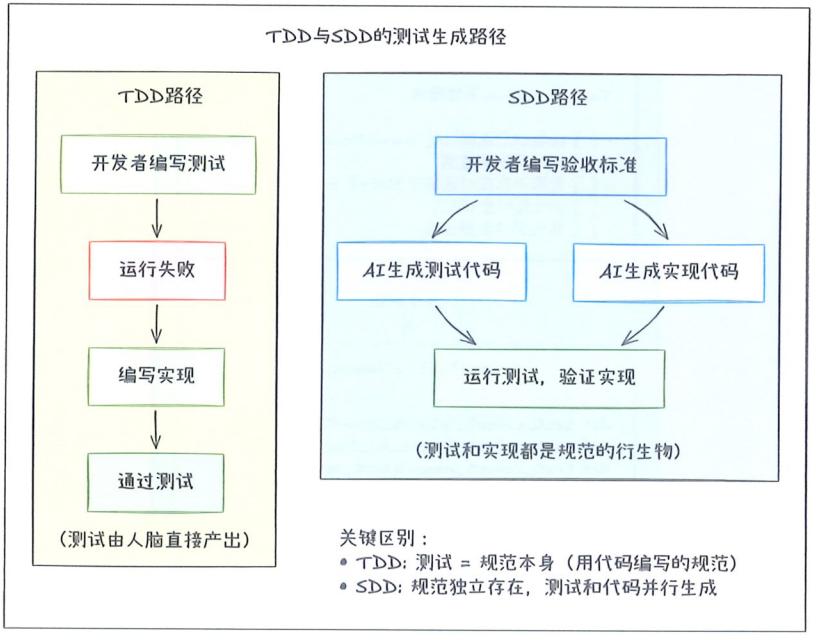


图 7-2　TDD 与 SDD 的测试生成路径对比

在实践中，你可以利用 AI 同时生成实现代码和测试代码。以下是一条典型的指令示例。

```
请实现 Task 3（GitHub 采集模块）。

要求：
1. 实现 collector/github.py，接口契约见 specs/design.md §4.1
2. 同时生成 tests/test_github.py
3. 测试用例必须覆盖 specs/tasks.md 中 Task 3 的所有验收标准
4. 使用 Mock 替代真实 API 调用
```

小冰尝试后，AI 确实生成了对应的实现代码和测试代码。但她敏锐地注意到一个潜在问题："测试是 AI 写的，实现也是 AI 写的——它会不会在'自己考自己'，出一些永远能通过的水题？"

"问得好。"咖哥赞许道，"这正是**验收标准必须由人来编写的原因**。如果验收标准明确规定'API 超时重试 3 次'，而 AI 生成的测试确实验证了这一重试逻辑，那么测试是有效的，因为规范约束了测试的边界。反之，如果你让 AI 自行决定测试范围，它很可能会忽略边界条件，只专注于测试那些'happy path'。"

关于如何在工具层强制"测试先于代码"，请访问 https://kage-ai.com/sdd/#ch7-tdd。

7.1.3　测试金字塔在 SDD 中的应用

Mike Cohn 在 *Succeeding with Agile* 一书中提出的测试金字塔（Test Pyramid）模型，在 SDD 中形成了新的映射关系（见图 7-3）。

表 7-1 详细展示了 SDD 的三层测试与规范的具体对应关系。小雪观察后表示："这意味着，如果你的规范编写得足够出色，测试策略几乎是自动确定的，不需要额外费心设计'测什么'。"

"没错。"咖哥肯定道，"这正是 SDD 中'驱动'二字的真谛——规范不仅驱动代码生成，也驱动测试生成。**三份规范，本质上就是三层测试的需求说明书**。"

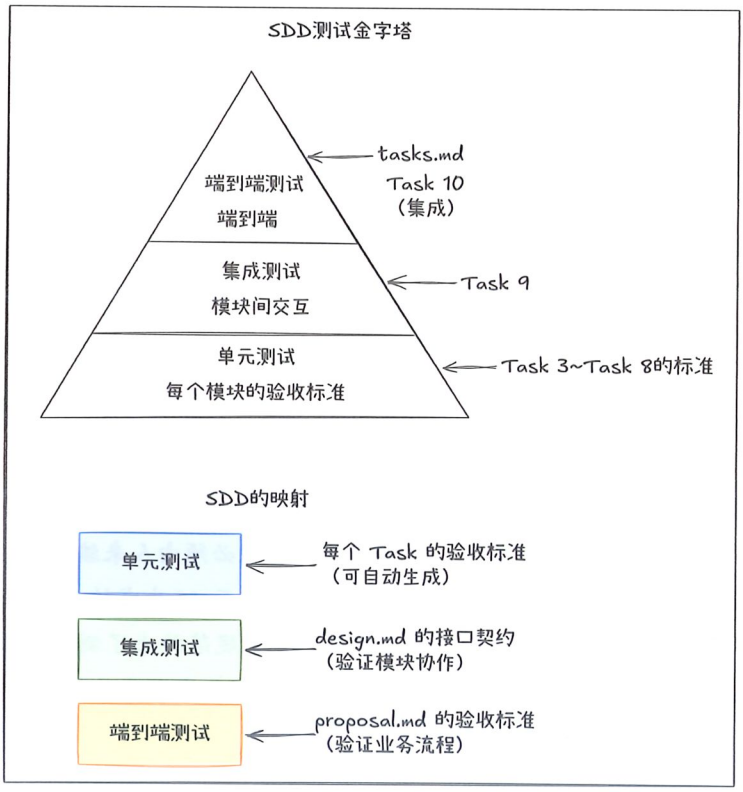


图 7-3　SDD 测试金字塔：三层规范驱动三层测试

表 7-1　SDD 的三层测试与规范的对应关系

测试层级	规范来源	测试内容	生成时机	示例
单元测试	tasks.md 验收标准	单个模块的功能行为	与模块实现同步生成	test_github.py
集成测试	design.md 接口契约	模块间的数据交互与传递	执行 Task 9（主编排任务）时生成	采集层输出→生成层输入
端到端测试	proposal.md 验收标准	端到端完整业务流程	执行 Task 10（集成测试任务）时生成	从采集到推送的全链路

7.2　回归验证：代码变了，规范怎么跟着变

7.2.1　变更的 4 种类型

在实际项目中，变更是不可避免的。关键不在于“有没有变更”，而在于“变更从哪里开始”。在 SDD 中，变更主要分为 4 种类型（见表 7-2）。

表 7-2　变更的 4 种类型及其影响链

变更类型	起点	影响链	示例
需求变更	proposal.md	proposal.md → design.md → tasks.md →代码→测试	新增考勤数据采集
架构变更	design.md	design.md → tasks.md →代码→测试	将 SQLite 换成 PostgreSQL
任务变更	tasks.md	tasks.md →代码→测试	拆解一个过大的任务
缺陷修复	代码	代码→测试（可能反向更新 design.md）	修复日期解析 bug

请注意影响链的方向。需求变更从最上游开始，像瀑布一样逐层向下影响；而缺陷修复则从最下游开始，通常只影响代码和测试。某些情况下，一个 bug 暴露的是设计缺陷，这时就需要逆流而上，修改 design.md 甚至 proposal.md。

7.2.2　变更流程：自上而下的涟漪

让我们通过小冰遇到的真实场景来演示这一流程。运营团队要求新增“考勤数据采集”功能——这是一个典型的需求变更，其起点正是 proposal.md。

需求变更的 SDD 流程如图 7-4 所示。

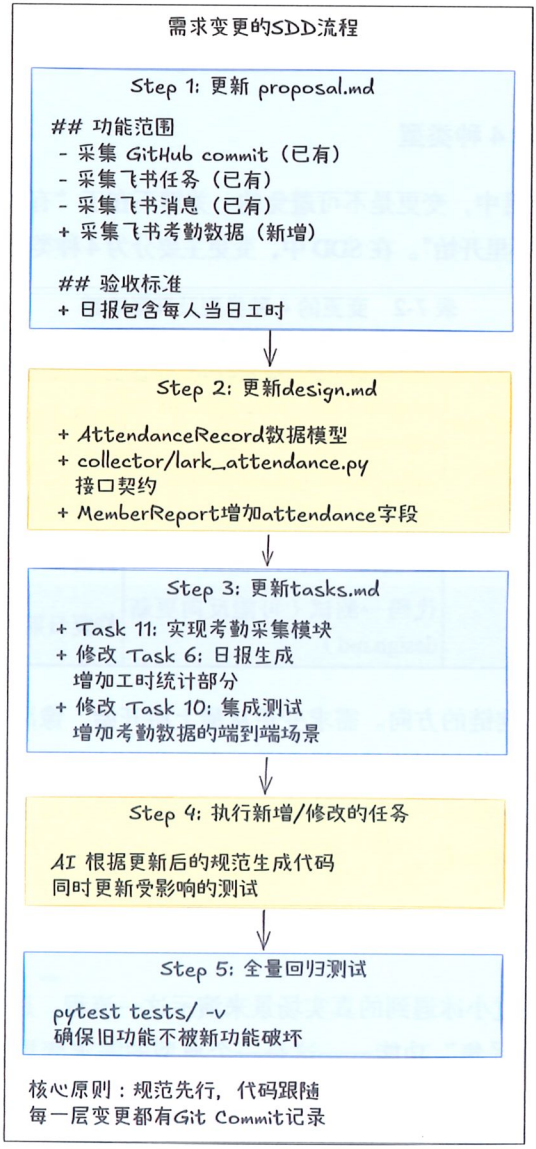


图 7-4　需求变更的 SDD 流程：自上而下的涟漪

小冰问："每次添加一个小功能，都需要从 proposal.md 开始修改吗？这样会不会显得过于烦琐？"

“并非每次都需要如此。”咖哥解释道，“这取决于变更的类型。如果仅仅是修复一个日期解析的 bug，你不需要改动 proposal.md，直接修改代码并补充测试即可。然而，如果是‘新增一个数据源’，这就改变了系统的功能边界，必须从 proposal.md 开始修改。判断标准非常简单：**审视这个变更究竟改变了‘做什么’（proposal.md）、‘怎么做’（design.md），还是仅仅影响了‘如何实现’（代码）。找到受影响的最上游层级，从那里开始修改。**”

7.2.3　逆向回溯：从代码 bug 发现规范缺陷

有时，变更的方向是自下而上的。

回顾第 6 章的思考题，其中有一个典型案例：Task 6 的验收标准要求“若某数据源采集失败，日报中需要标注‘数据获取失败’”，然而，采集层的接口仅返回 list[Record]，在这种情况下，空列表既可能意味着“今天确实没有数据”，也可能意味着“API 调用失败”。在现有接口定义下，这两种截然不同的情况无法区分。

这是一个典型的**在实现过程中暴露出的设计缺陷**。正确的处理方式绝非在代码中通过临时方案（hack）来绕过，而是应当回溯至 design.md，修改接口契约（见图 7-5）。

Fred Brooks 在《人月神话》中曾言：“计划抛弃一个；反正你终究会抛弃一个。”在 SDD 语境下，虽然不需要推翻整个系统，但在实现过程中发现规范的缺陷几乎是必然的。关键在于**坦然接受这一事实，并建立一套优雅的流程来从容应对。**

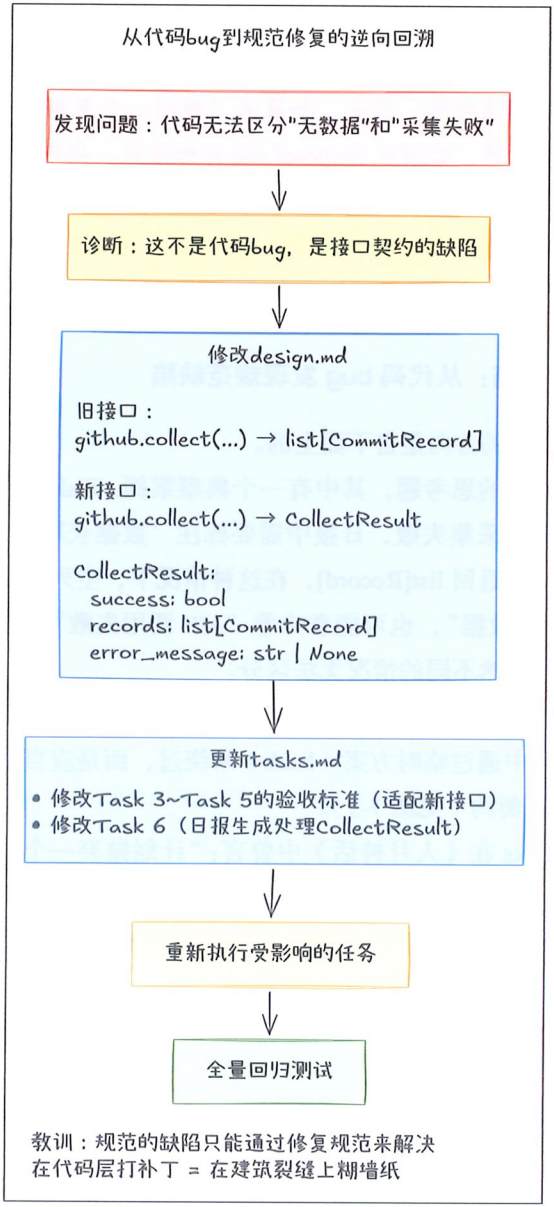


图 7-5　从代码 bug 到规范修复的逆向回溯

7.3　规范的版本演进

7.3.1　规范的版本控制

规范是“活”的文档，应当与代码一样纳入严格的版本控制体系。

在前几章中，我们已将 proposal.md、design.md 和 tasks.md 置于 specs/ 目录下，随代码一同由 Git 管理。然而，仅仅将文件放入 Git 仓库是不够的，还需要建立一套明确的规范来管理这些文档本身的演进过程。

规范版本管理的三个原则如图 7-6 所示。

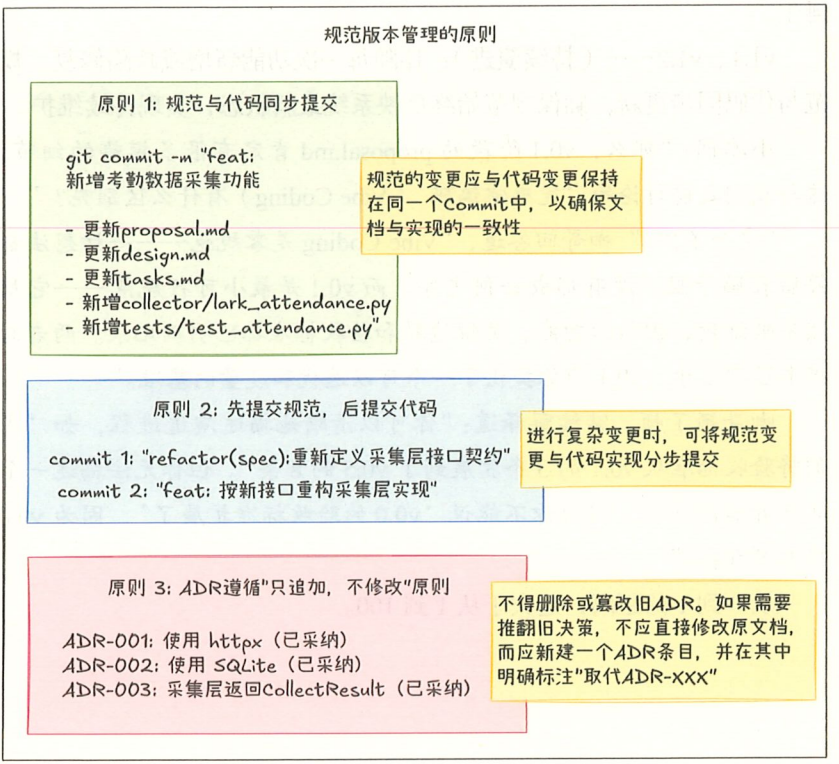


图 7-6　规范版本管理的三个原则

7.3.2 增量式规范迭代

在实际项目开发中，规范并非“一劳永逸”的静态文档，而是遵循增量迭代的动态节奏，随着项目的推进而不断完善。

规范的版本演进如下。

v0.1（最小可行规范）：proposal.md 仅包含最核心的功能定义和项目边界，内容足以启动初步的架构设计，为项目奠定方向。

v0.5（基线规范）：proposal.md、design.md 和 tasks.md 三者构成完整的规范集合这，已细化到足以支撑团队开展正式编码工作。

v1.0（首版完成）：代码实现完成并通过所有测试，规范文档与代码实现完全对齐，此阶段标志着一个可交付、功能完整的初始版本诞生。

v1.1、v1.2……（持续演进）：伴随每一次功能新增或缺陷修复，规范与代码同步更新，确保规范始终反映系统最新状态，实现持续维护。

小冰问：“那么，v0.1 阶段的 proposal.md 肯定有很多模糊的细节。这与我们之前讨论的‘凭感觉编码’（Vibe Coding）有什么区别呢？”

“完全不同。”咖哥回答道，“Vibe Coding 是零规范——一切想法都停留在脑子里，没有形成任何文字。而 v0.1 是最小可行规范——它虽然不够细致，但核心功能、关键边界和验收标准都已明确记录。两者的根本区别在于，v0.1 为你提供了一个可以迭代和度量的基准。”

咖哥顿了顿，继续解释道：“你可以清晰地描述演进进程，如‘我们将验收标准从 v0.1 的 3 条扩展到了 v0.5 的 8 条’。但你无法描述一个从零开始的演进，因为你不能说‘v0.0 的验收标准扩展了’，因为 v0.0 根本不存在。”

从 0 到 1 的距离，远大于从 1 到 100。

7.4　Brownfield 场景：为存量系统补写规范

7.4.1　存量系统的困境

在前 6 章中，我们聚焦于新建系统（Greenfield），即从第一行规范开始，逐步构建出可运行系统的理想路径。然而，现实往往更为复杂，大多数开发者面对的并非一张空白的画布，而是一个早已存在的代码库。

这便是存量系统（Brownfield）的典型场景，它们通常具备以下特征。

- 逻辑成谜：代码虽然能够运行，但“为什么这么写”背后的业务逻辑和决策过程已无人能说得清。
- 需求散佚：不存在 proposal.md，需求分散在 Jira 工单、微信群聊和历次会议纪要的碎片信息中。
- 架构漂移：没有 design.md，系统架构并非经过深思熟虑的“设计”，而是在一次次修补中“演化”形成的。
- 测试薄弱：自动化测试覆盖率极低（可能只有 20%，甚至更低），导致任何修改都伴随巨大的回归风险。

面对这样的困境，小雪提出了一个非常现实的问题：“难道要先停下来，把整个系统的规范全部补齐再开发新功能吗？这可能需要耗费数月时间。”

“当然不是。”咖哥回答道，“为存量系统补写规范，正确的策略是‘逐步收编’，而非发动一场‘全面战争’。”

7.4.2　逐步收编策略

图 7-7 展示了为存量系统补写规范的“逐步收编”策略。

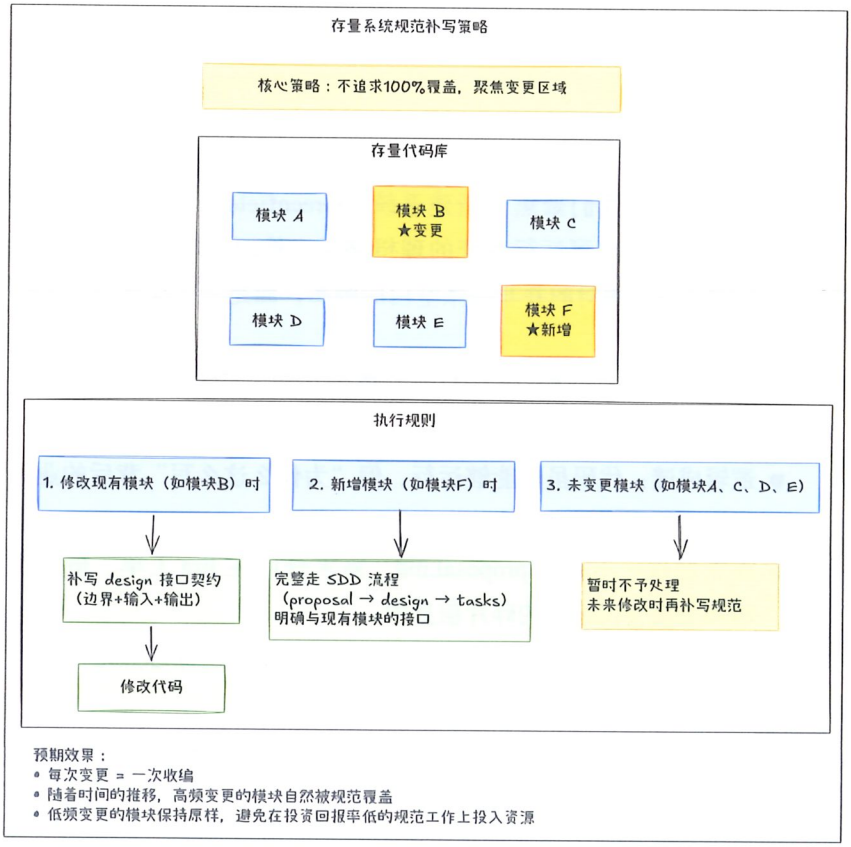


图 7-7　为存量系统补写规范的策略

这个策略的灵感源自 Martin Fowler 描述的“绞杀者模式”（Strangler Fig Pattern）。其核心思想并非一次性替换整个旧系统，而是让新系统如同绞杀榕一般，逐步包裹并替代旧系统。

在 SDD 语境下，规范扮演着绞杀榕的角色。它并非一次性覆盖整个代码库，而是在每次代码改动时，为被修改的模块“包裹”上一层规范。通过这种渐进式的方法，系统的关键部分会随着持续迭代而被逐步收编，最终实现规范与代码的和谐共生。

7.4.3　存量系统的规范补写流程

在操作层面，为已有模块补写规范通常遵循以下 4 个步骤。

第一步，逆向工程：通过代码审查理解模块的现有逻辑。此阶段可利用 AI 辅助分析，Prompt 示例如下。

```
请阅读 collector/github.py 的代码，并提取以下信息：
1．对外接口（函数签名及返回值）
2．依赖的外部服务或模块
3．异常处理机制
4．行为边界（不同输入下的预期行为）

输出要求：请严格按照 design.md 的接口契约格式输出
```

第二步，规范化：将第一步的结果整理为 design.md 级别的接口契约。此步骤必须由人工介入进行判断，因为代码中的“现有行为”并不等同于“正确行为”。人工审查旨在识别遗漏的异常处理，或剔除历史遗留的临时方案。

第三步，补写测试：依据整理好的接口契约生成测试用例。这些测试不仅用于验证现有行为，更能有效暴露代码中未覆盖的边界情况。

第四步，锁定基线：在测试全部通过后，将规范文档与测试代码一并提交。此后，该模块的任何变更都必须遵循“先更新规范，再修改代码”的原则。

小雪若有所思地补充道：“逆向工程这一步，本质上是一场将隐性知识转化为显性规范的‘考古挖掘’。代码深处潜伏着许多未曾言明的设计决策，例如，为何重试次数设定为 3 次，或是超时阈值为何卡在 30s。这些逻辑往往只存在于代码行间，却鲜少被记录在案。补写规范，正是将这些沉睡的隐性决策挖掘出来，并固化为可传承的文档。”

“正是如此。”咖哥深表赞同，“正如 Michael Polanyi 所提出的‘隐性知识’概念——‘我们知道的，往往比我们能说出来的多得多’。存量系统中存在大量这类未被言明的设计智慧。因此，补写规范的价值远不止于‘拥有一份文档’，其核心意义在于它迫使团队完成了一次知识

的显性化跃迁。一旦这些隐性知识被转化为显性规范，它们便脱离了黑盒状态，变得可被审查、可被讨论，亦可被持续迭代优化。”

7.5 实战：完整的 SDD 迭代循环

7.5.1 场景回顾

让我们回到小冰面临的实战场景：运营部门提出了新增飞书考勤数据采集功能的需求。接下来，我们将以这一真实需求为例，完整推演一遍 SDD 的迭代循环。

7.5.2 Step 1，更新 proposal.md

首先，我们需要在 specs/proposal.md 中更新功能范围，明确 v1.1 的迭代内容。

```
## 功能范围

### 做什么
- 采集指定 GitHub 仓库的 Commit 记录
- 采集飞书项目的任务状态变更
- 采集指定飞书群的消息
- 采集飞书考勤数据（新增 v1.1）
- 聚合数据，按成员生成结构化日报
- 通过邮件和飞书消息推送日报

### 验收标准（v1.1 新增）
- 日报中包含每位成员的当日工时和出勤状态
- 考勤数据不可用时，日报对应字段应明确标注“考勤数据暂不可用”
```

7.5.3 Step 2，更新 design.md

接下来，我们需要在 design.md 中补充详细的数据模型定义和接口契约，确保实现有据可依。

```
## 新增数据模型

### AttendanceRecord（考勤记录）

| 字段 | 类型 | 说明 |
|------|------|------|
| employee_id | str | 飞书用户 ID |
| date | date | 考勤日期 |
| check_in | datetime | None | 签到时间 |
| check_out | datetime | None | 签退时间 |
| work_hours | float | 工时（小时） |
| status | str | 正常 / 迟到 / 早退 / 缺勤 / 休假 |

## 新增接口契约

# collector/lark_attendance.py
def collect(since: date, until: date) -> CollectResult[AttendanceRecord]
```

同步更新 MemberReport 模型，增加 attendance: AttendanceRecord | None 字段。

7.5.4　Step 3，更新 tasks.md

然后，我们需要将设计转化为可执行的任务，在 tasks.md 中新增 Task 11，并同步修改受影响的 Task 6。

```
## Task 11：实现考勤数据采集模块（新增 v1.1）

描述：对接飞书考勤 API，获取团队成员的签到与签退数据。

输入：
design.md §3（AttendanceRecord 数据模型）
design.md §4.1（采集层接口契约）
输出：
collector/lark_attendance.py
tests/test_attendance.py
依赖：Task 2（共享基础层）
```

```
验收标准：
- [ ] collect() 函数签名严格符合 design.md 的接口定义
- [ ] 返回 CollectResult 对象（包含 success 标志位）
- [ ] 逻辑正确计算工时（check_out - check_in）
- [ ] 妥善处理缺失签退数据的情况（状态标注为“签退缺失”）
- [ ] 当 API 不可用时，success 置为 False 并附带详细错误信息
- [ ] 基于 Mock 数据的单元测试全部通过

---

## Task 6：日报生成模块（v1.1 变更）

新增验收标准：
- [ ] 日报内容必须包含每位团队成员的工时统计
- [ ] 当考勤数据不可用时，界面需要显示“考勤数据暂不可用”
- [ ] 布局顺序调整：工时统计模块应位于“任务进展”之后、“协作沟通”之前
```

7.5.5 Step 4，执行与验证

规范更新完毕后，我们即可指示 AI 执行新增及修改任务。

```
请执行 Task 11（考勤采集模块）。

具体需求见 specs/tasks.md 中 Task 11 的描述及验收标准。
数据模型定义见 specs/design.md 中的 AttendanceRecord。
接口契约见 specs/design.md §4.1 中的采集层接口规范。
共享模块（config、logger、errors）已在 shared/ 目录中实现。
CollectResult 模型参见 ADR-003。

请同时生成业务实现代码与对应的测试用例。
```

修改 Task 6（日报生成模块）：在现有代码基础上，集成考勤数据展示功能。由于 design.md 已明确定义了新增字段与接口契约，AI 能够精准定位代码修改位置（“在哪里加”）及具体实现内容（“加什么”），从而高效完成迭代。

7.5.6　Step 5，全量回归测试

最后，我们需要运行全量测试套件，以确保系统的整体稳定性。

```
# 执行所有测试
python -m pytest tests/ -v

# 预期输出示例：
# tests/test_github.py       .... PASSED
# tests/test_lark_task.py    .... PASSED
# tests/test_lark_msg.py     .... PASSED
# tests/test_attendance.py   .... PASSED  ← 新增测试
# tests/test_generator.py    .... PASSED  ← 包含新的考勤逻辑测试
# tests/test_email.py        .... PASSED
# tests/test_lark_bot.py     .... PASSED
# tests/test_main.py         .... PASSED
# tests/test_integration.py  .... PASSED
```

在上述输出示例中，所有旧测试通过证明新功能的引入未破坏现有系统的稳定性，而新测试通过证明新功能已严格按照规范正确实现。

7.5.7　完整迭代循环图

图 7-8 展示了完整的 SDD 迭代循环——从需求变更触发，到规范更新，再到代码实现与最终验证的全过程。

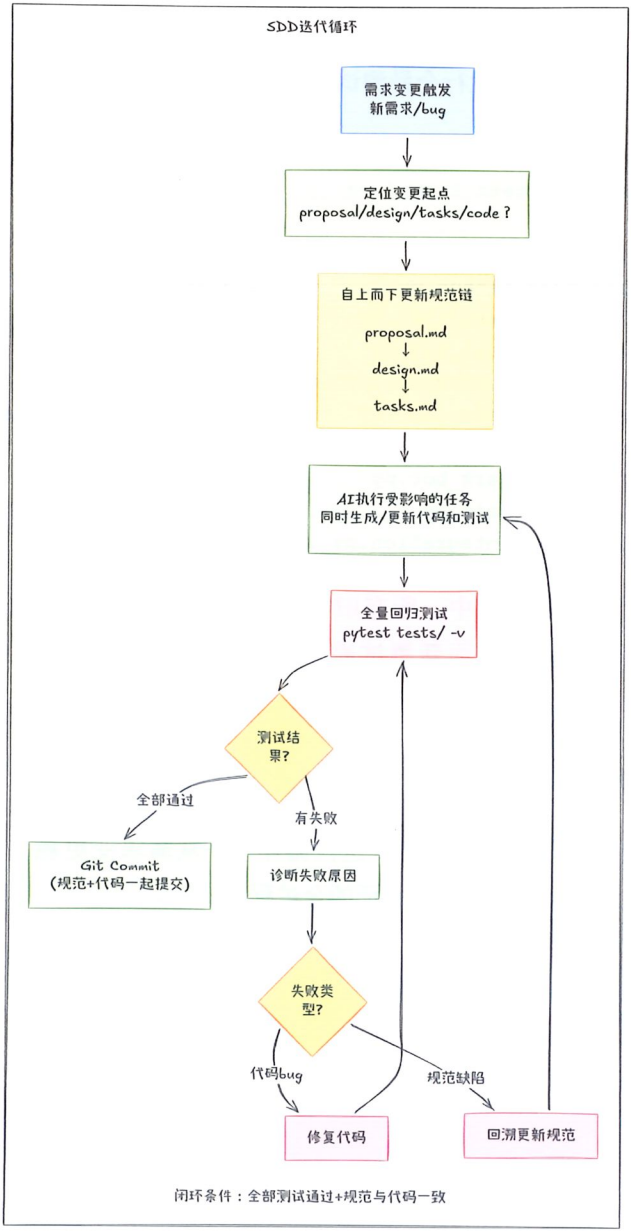


图 7-8　完整的 SDD 迭代循环

本章小结

这一章完成了 SDD 六阶段工作流的最后一环：验证、迭代与演进。如果说前面几章教你“从 0 到 1”构建系统，那么本章教你“从 1 到 100”——如何在需求持续变化的洪流中，保持规范与代码的长期一致性。

本章的核心要点如下。

- **验收标准直接映射为测试用例**：三层规范（proposal、design 和 tasks）天然驱动三层测试（端到端测试、集成测试和单元测试），测试策略几乎可以自动确定。
- **变更必须从最上游的规范层开始**：需求变更修改 proposal，架构变更修改 design，缺陷修复修改代码。每种变更类型对应不同的影响链。
- **规范缺陷只能通过修改规范来修复**：在代码层打补丁，无异于在建筑的裂缝上糊墙纸。如果实现过程中发现接口契约存在缺陷，必须回到 design.md 修正，而不是在代码中采用临时方案。
- **规范与代码同步提交**：在同一个 Git Commit 中，也可以先提交规范，再提交代码。ADR 遵循“只追加，不修改”的原则。
- **存量系统采用“逐步收编”策略**：不追求 100% 的规范覆盖。每次修改一个模块，就为它补上一圈规范。时间会让高频变更的模块自然被收编进行规范体系。

从第 4 章到第 7 章，我们以“智能日报生成器”项目为例，走了一遍 SDD 的流程：从需求分析（proposal.md）到架构设计（design.md），再到任务拆解（tasks.md），最后到执行验证（代码 + 测试）与迭代演进。这三份规范文档与代码共同构成了一个可追溯、可审查、可复用的工程体系。

从第 8 章开始，我们将探索 SDD 如何与 Agent 设计模式深度结合。届时，规范将不仅驱动人类与 AI 的高效协作，更驱动多个 AI Agent 之间的自动化复杂协作。



咖哥发言

很多开发者习惯将“写完代码”视为项目的终点，但对任何一位经验丰富的工程师而言，代码的交付仅是项目生命周期的起点——紧随其后的是漫长的维护、迭代与演进。

SDD 的真正价值，正是在这种持续的演进中才得以凸显。当需求发生变更时，你不需要再从“散落在聊天记录中的碎片化上下文”中费力拼凑项目全貌。proposal.md、design.md、tasks.md 本身就是全貌，它们与代码一同在 Git 中鲜活地存在，随时可供查阅与修改。

Peter Naur 在 1985 年的经典论文“Programming as Theory Building”中提出：编程的本质并非生产代码，而是构建针对问题领域的理论。代码仅仅是这种理论的一种具象表达。当这套理论仅存在于开发者的脑海中时，它会随着人员流动而消散——这正是存量系统无人敢修改、最终沦为“黑盒”的根本原因。

而 SDD 的三份规范，正是将这一“理论”固化下来的工具。规范不仅是文档，更是团队共享的理论基石。

思考题

1. 在 SDD 中，测试用例是由规范自动衍生的。然而，若规范本身存在缺陷（例如，验收标准遗漏了某个边界条件），相应的测试用例也会遗漏。你认为应当如何防范“规范级别的 bug”？是否存在“对规范进行测试”的方法？
2. 在 Brownfield 场景中，“逆向工程→规范化→补写测试”的流程，默认假设现有代码的行为是“基本正确”的。然而，若现有代码本身包含已知 bug，那么在补写规范时，应当如实记录 bug 行为（反映现状），还是直接定义正确行为（描述理想状态）？这两种策略各有什么利弊？

3. 本章展示的迭代循环（见图 7-8）采用串行模式，即先修改规范，再修改代码，最后进行测试。对追求快速迭代的创业团队而言，这套流程是否显得过于厚重？若要构建“轻量级 SDD 迭代”，你认为应当精简哪些步骤，又该保留哪些核心环节？

第 8 章
登堂入室：SDD 与 Agent 设计模式

设计模式是招式，SDD 是内功心法——招式决定你能不能打，内功心法决定你能打多久。

小冰完成智能日报生成器 v1.0 后，陷入了新一轮的兴奋。

SDD 的三份规范（proposal.md、design.md、tasks.md）让她首次体会到 AI 辅助编程的“可控感”。她不再需要对着聊天窗口祈祷 AI 能猜中自己的意图，而是能通过规范精确地告诉 AI 做什么、怎么做以及做到什么程度。

但她也发现了一个效率瓶颈：每次执行任务时，她仍需要手动将规范中的相关内容复制粘贴给 AI、手动检查验收标准，再手动触发下一个任务。整个过程就像人工操控提线木偶——每一根线都得自己来拉。

“有没有办法让 AI 自己读取规范、自主执行并自动验证？”小冰向咖哥问道。

“有。”咖哥回答道，“你需要的不仅仅是更好的提示工程（Prompt Engineering），而是 Agent 设计模式。”

他进一步解释道：

“前 7 章讲授的是 SDD 的方法论——如何编写、迭代及验证规范。这些方法论好比‘内功心法’。但若要将这套方法论转化为自动化的流水线，则需要‘招式’，也就是 Agent 系统中那些经过实践检验的架构模式。

“好消息是，SDD 的结构与几个关键的 Agent 设计模式有着天然的契合。这并非生搬硬套，更是结构性的匹配。在本章中，我们将探讨这些模式如何助力 SDD 从‘人工驾驶’升级为‘辅助驾驶’。”

8.1　规划与执行模式：SDD 的天然搭档

8.1.1　模式简介

规划与执行（Plan-and-Execute）模式是 Agent 设计中最基础也最重要的模式之一。其核心思想非常直观：将“思考”与“执行”分离。

在该模式中，规划器（Planner）负责制订计划，执行器（Executor）负责逐步落实。通常，规划器利用高能力模型进行全局思考，执行器则使用轻量级模型处理具体操作。两者分工明确，各司其职（见图 8-1）。

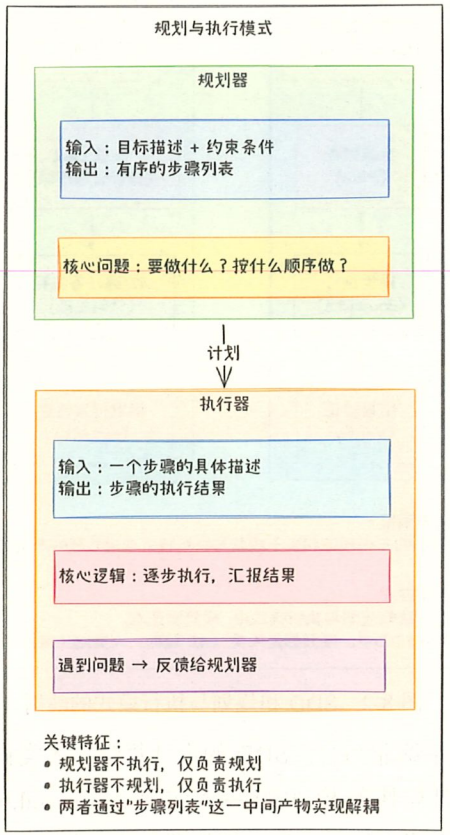


图 8-1　规划与执行模式：将“思考”与“执行”分离

8.1.2 SDD 和规划与执行模式的结构映射

现在你发现了吗？SDD 的三份规范天然构成了一个规划与执行架构（见图 8-2）。

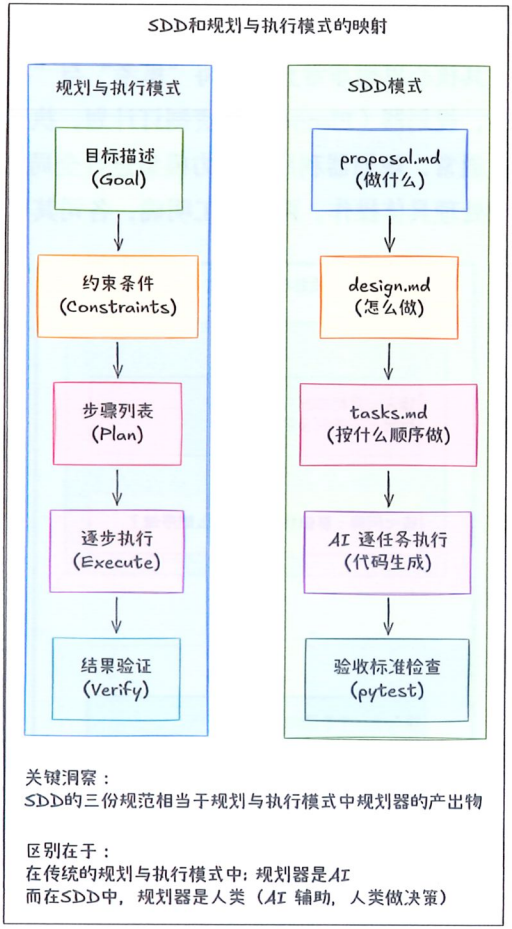


图 8-2　SDD 和规划与执行模式的映射

这种映射关系绝非巧合。SDD 的设计哲学“人类做决策，AI 执行”本质上就是规划与执行模式的一个实例。proposal.md、design.md 和 tasks.md 这三份文档正是人类扮演规划器角色时的产出物。

小冰恍然大悟："原来如此！所以回顾前 7 章，我们其实一直在手动运行一个'规划与执行'流程——我（人类）担任规划器，而 Claude Code 是执行器？"

"没错。"咖哥肯定道，"本章的目标，正是将这一手动流程升级为半自动化流程——让 AI 分担更多规划器的工作（例如，基于 proposal 生成 design 初稿），同时利用规范来约束执行器的行为边界。"

8.1.3　CLAUDE.md 作为规划上下文

在 Claude Code 中，规划与执行模式的"规划上下文"是通过 CLAUDE.md 文件注入的。图 8-3 展示了一个典型的配置示例。

CLAUDE.md：SDD的规划上下文

```
# CLAUDE.md

## 项目规范
所有开发工作必须遵循以下规范文档：
- 需求规范：specs/proposal.md
- 架构设计：specs/design.md
- 任务计划：specs/tasks.md
- 接口契约：specs/contracts/
- 架构决策：specs/adrs/

## 开发规则
1. 执行任务前，先读取对应的规范文档
2. 代码必须符合design.md中的接口契约
3. 每个任务完成后，逐条检查验收标准
4. 发现规范缺陷时，先报告而不是绕过

## 当前状态
已完成：Task 1~Task 10 (v1.0)
进行中：Task 11 (考勤采集，v1.1)
```

效果：AI在每次对话开始时会自动加载这些规范，这相当于执行器在启动时自动读取规划

图 8-3　CLAUDE.md 将 SDD 规范注入 AI 的规划上下文

引入 CLAUDE.md 后，AI 不再是一个"无记忆的对话机器人"，它

清晰掌握了项目的需求边界、架构约束及当前开发进度，确保每一次新对话都能基于同一套规范基线展开。这正是规划与执行模式中“共享规划上下文”的具体实现。

8.2 生成—评审模式：规范审查的自动化

8.2.1 模式简介

生成—评审（Generator-Critic）模式是 Agent 设计中的一种“反思”模式。其核心思想在于：**由一个 Agent 负责生成内容，另一个 Agent 负责评审，二者循环迭代，直至产出质量达标。**

该模式的灵感源自人类的创作流程——作家撰写初稿，编辑提出修改意见，作家据此修订，编辑再次审核……如此往复，直至双方满意。在软件工程领域，代码审查便是生成—评审循环的典型应用（见图 8-4）。

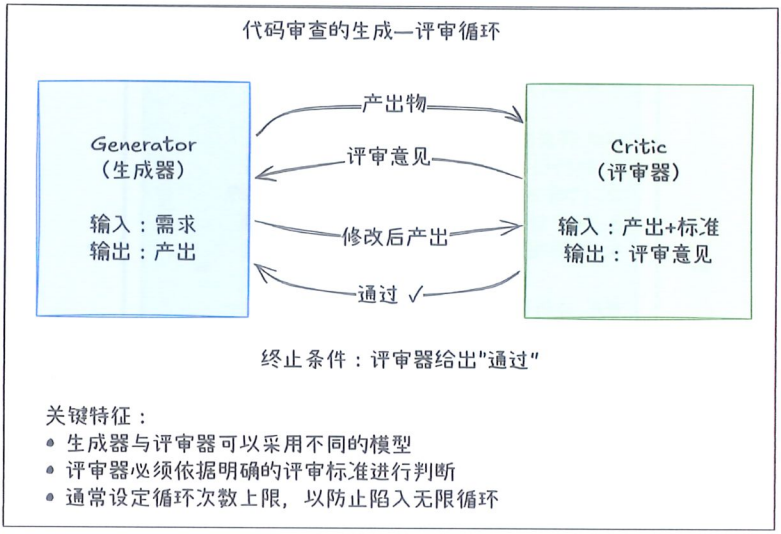


图 8-4 代码审查的生成—评审循环

8.2.2　SDD 为评审提供客观标准

生成—评审模式面临的最大挑战在于：**评审标准从何而来？**

若缺乏明确的评审标准，评审过程便极易沦为“凭感觉打分”——今日认为可行，明日又觉不妥，导致结果不可复现。这好比一场没有评分细则的考试，分数完全取决于阅卷人的主观判断。

SDD 恰好解决了这一难题。在 SDD 语境下，规范文档天然构成了客观的评审标准（见图 8-5）。

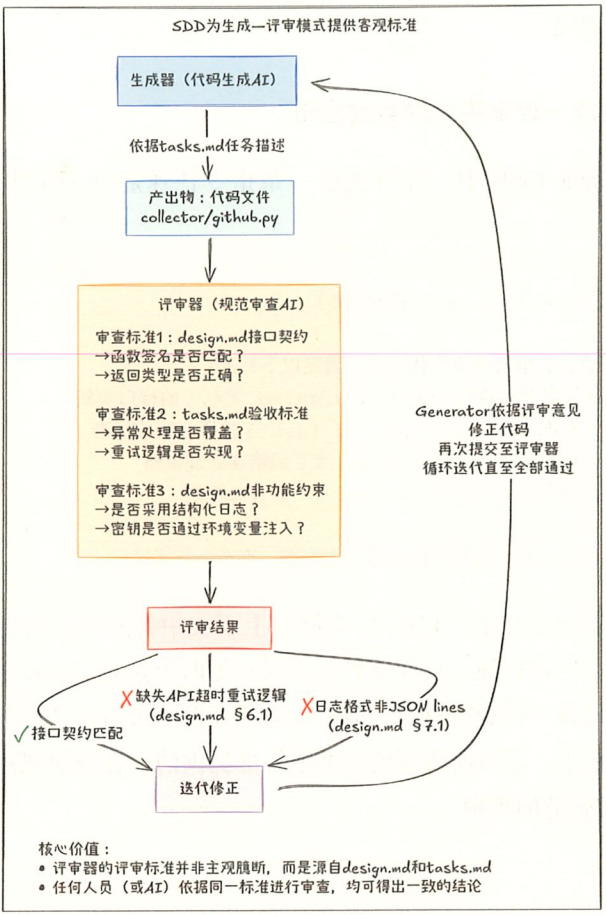


图 8-5　SDD 规范为生成—评审模式提供客观标准

小冰问："这与第 7 章提到的验收标准检查有何不同？看起来都是用标准来验证代码。"

咖哥解释道："关键在于谁来做和什么时候做。第 7 章的验收标准检查是人类在代码生成后进行的手动验证，是一个独立的、需要人工介入的环节。而我们现在讨论的是将这一过程自动化、内嵌化。评审器会直接将评审意见返回给生成器，要求其自动修改，再进行审查。这一'生成—评审—修改—再生成'循环完全由 AI 自主完成，不需要人工干预，直至代码完全符合规范。你可将其理解为一个自动化、持续运行的 AI 代码审查员。"

8.2.3 生成—评审模式的实践应用

在 Claude Code 中，你可通过一条指令快速启动一个生成—评审循环。

```
请实现 Task 3（GitHub 采集模块）。

实现完成后，请自行审查代码是否满足以下标准：
1. 函数签名是否符合 specs/design.md §4.1 的接口契约
2. 是否实现了 specs/tasks.md Task 3 的所有验收标准
3. 是否符合 specs/design.md §6 的错误处理策略
4. 是否符合 specs/design.md §7 的日志规范

若存在不符合项，请自行修改后再次审查，直至全部通过。
```

指令的核心在于，它让 AI 在同一上下文中扮演生成器与评审器的双重角色。尽管这种"自我反思"模式不如两个独立 Agent 之间的相互审查那样具备严格的隔离性，但对大多数开发场景而言已足够有效。其根本原因在于，评审所依据的标准是外部提供的、客观的规范文档，而非 AI 临时臆造的规则。

8.3　层级委托模式：规范驱动的多智能体协作

8.3.1　模式简介

层级委托（Hierarchical Delegation）模式是多 Agent 系统中的经典协作模式。其核心思想在于：**由一个 Manager Agent 负责理解宏观目标，拆解并分配任务，多个 Worker Agent 则负责执行具体子任务。**

这一模式的灵感源自现代企业的组织管理架构——正如 CTO 不需要亲自编写代码，但需要深刻理解业务目标，并将其拆解为具体技术任务，分配给开发团队；团队成员各司其职，完成分内工作后汇总交付（见图 8-6）。

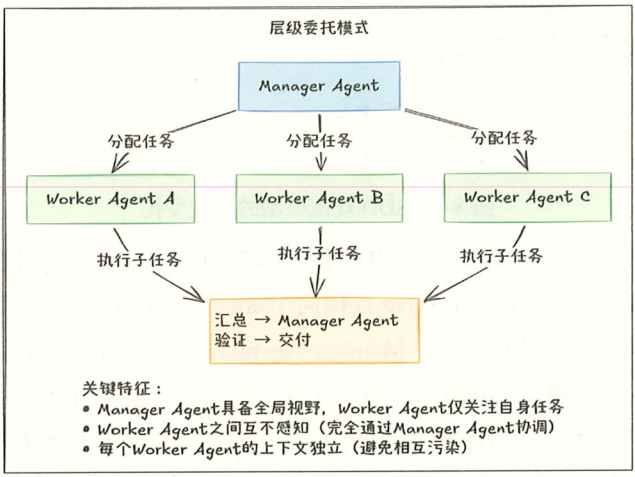


图 8-6　层级委托模式：Manager Agent 分配任务，Worker Agent 具体执行

8.3.2　SDD 与层级委托模式的结合

回顾第 6 章中子智能体并行执行的场景，GitHub、飞书任务、飞书消息这三个采集模块由三个子智能体并行开发。这实际上就是层级委托模式的体现——只不过当时的 Manager Agent 是人类（小冰），而 Worker Agent 则是 AI 子智能体。

现在，我们可以进一步将 Manager Agent 的职责也移交给 AI（见图 8-7）。

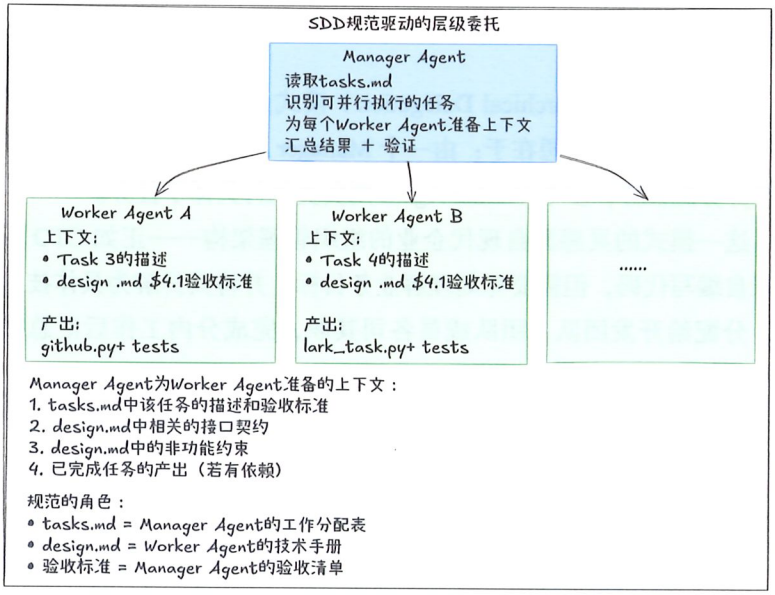


图 8-7　SDD 规范驱动的层级委托

这里有一个深刻的洞察：SDD 的三份规范解决了层级委托模式中最棘手的难题——**Manager Agent 如何向 Worker Agent 精准传达任务**。

在缺乏规范的情况下，Manager Agent 必须自行理解用户意图、拆解任务，并构建每个 Worker Agent 的上下文。这一过程中的每一个环节都可能引入偏差。而在 SDD 模式下，这些工作已由人类预先完成——tasks.md 是现成的任务分配表，design.md 是现成的技术手册。Manager Agent 只需要“读取规范、按规分配”，不需要“理解需求、创造性拆解”。

小雪总结了一个关键点：“SDD 让 Agent 系统从‘自主规划’降级为‘按图执行’。这看似是能力的降低，实则是可靠性的提升——你不再需要 Agent 具备超强的规划能力，只需要它能忠实地执行人类已验证过的计划。”

8.4　护栏三明治模式：规范作为安全护栏

8.4.1　模式简介

护栏三明治（Guardrail Sandwich）模式是 Agent 安全设计中的核心模式。其核心思想十分直观：在 Agent 的输入端与输出端各设置一道检查关卡（见图 8-8）。

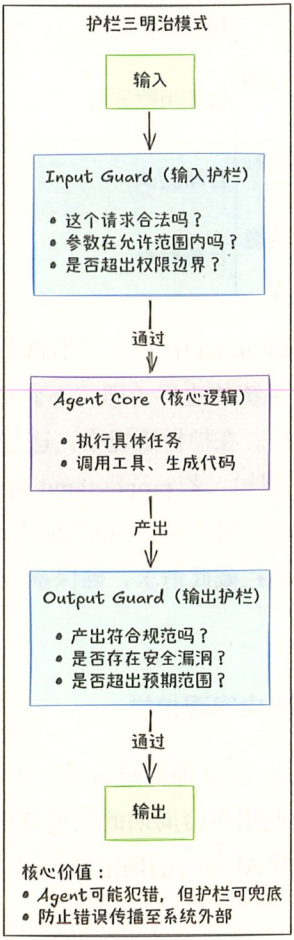


图 8-8　护栏三明治模式：输入护栏→核心逻辑→输出护栏

8.4.2 SDD 规范作为护栏的素材

在 SDD 体系中，规范文档天然构成了护栏规则的基石（详见表 8-1）。

表 8-1 SDD 规范到护栏的映射

护栏层级	规范来源	检查内容	示例
输入护栏	proposal.md 的“不做什么”部分	拦截超出范围的请求	拒绝“添加自然语言摘要”的请求
输入护栏	design.md 的安全约束	检查敏感信息	拦截包含硬编码密钥的代码
输出护栏	design.md 的接口契约	验证函数签名	确认返回类型是 CollectResult
输出护栏	tasks.md 的验收标准	自动测试	运行 pytest，检查全部通过
输出护栏	design.md 的非功能约束	合规检查	确认日志格式为 JSON lines

需要特别关注 proposal.md 中关于“不做什么”的界定——正如第 4 章所强调的，明确系统的边界（即“不做什么”）往往比定义功能（即“做什么”）更为关键。在护栏语境中，这些否定性约束直接转化为输入护栏的拦截规则。例如，若 proposal.md 明确规定“不提供智能摘要功能”，那么当用户（或 AI Agent）试图向系统引入 NLP 模块时，输入护栏应依据该规范自动拦截此请求，确保系统行为始终与初始设计意图保持一致。

8.4.3 在 Claude Code 中实现护栏

Claude Code 的 Hooks 机制为构建自动化护栏提供了原生支持。该机制允许开发者在工具调用生命周期的关键节点（如调用前、调用后）注入自定义脚本，实现对 AI 行为的确定性控制（见图 8-9）。

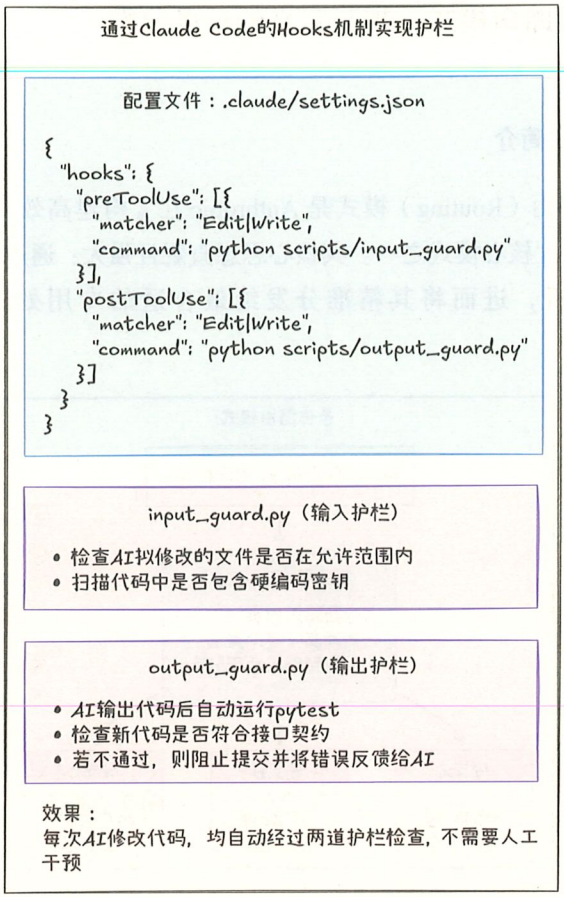


图 8-9　通过 Claude Code 的 Hooks 机制实现护栏

小冰眼前一亮："这不就是自动化的 SDD 验证流程吗？每次 AI 修改代码，Hooks 都会自动检查规范合规性，若不合规，AI 收到反馈后自动修正——这构成了一个闭环的自动化生成—评审循环。"

"没错。"咖哥赞许道，"你看，这 4 种模式并非孤立存在，而是可以有机组合的：规划与执行模式提供全局框架，层级委托模式赋予并行处理能力，生成—评审模式保障交付质量，而护栏三明治模式则提供安全兜底。贯穿这一切的核心，正是 SDD 的三份规范文档——它们是所有模式共享的单一事实来源。"

8.5 条件路由模式：规范定义的任务分发

8.5.1 模式简介

条件路由（Routing）模式是 Anthropic 在《构建高效 Agent》指南中确立的五大核心模式之一。其核心思想直观且强大：**通过分类器对输入进行分析，进而将其精准分发给最合适的专用处理分支**（见图 8-10）。

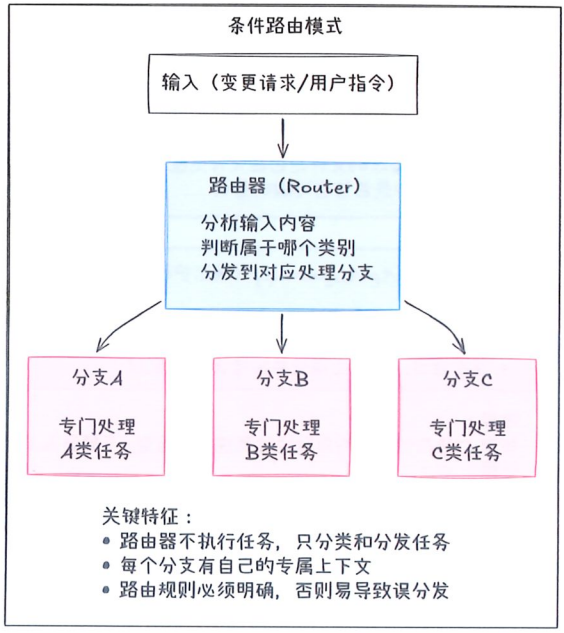


图 8-10 条件路由模式：分类→分发→专门处理

这一模式在现实世界中有着广泛的映射，医院的分诊台便是最典型的“路由器”：护士根据病人的症状（输入）判断其属于“外科”还是“内科”，“急诊”还是“普通”（分类），随后将其引导至对应的科室（分发）。分诊台本身并不负责治疗，但其存在确保了每个病人都能被高效地送至最对症的医生面前。

8.5.2　SDD 为路由提供分发规则

条件路由模式面临的最大挑战，与生成—评审模式类似，在于规则的来源——路由器依据什么标准来判断一个请求应被分发至哪个处理分支。

在 SDD 体系中，design.md 中定义的模块划分天然构成了路由规则的基石（见图 8-11）。

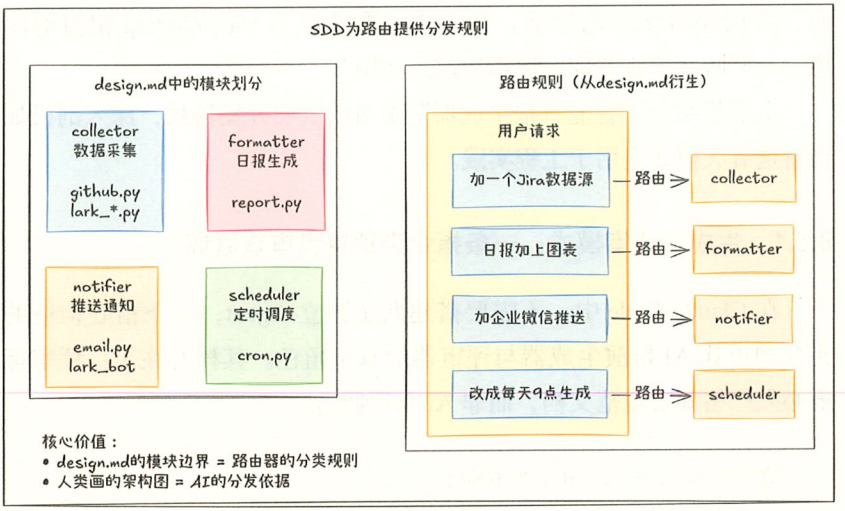


图 8-11　design.md 的模块边界就是路由器的分发规则

小雪敏锐地提出疑问："这与层级委托模式有什么区别？层级委托模式似乎也是将任务分配给不同执行者。"

"区别在于**分配的时机和依据**。"咖哥解释道，"层级委托模式采用的是计划阶段的分配——Manager Agent 读取 tasks.md，将已知且确定的任务预先分配给特定 Worker Agent。而条件路由模式则采用运行时的动态分配——当一个新的请求进入时，路由器根据请求的具体内容，实时判断其应归属的模块。简单来说，一个是'事先排兵布阵'，另一个是'临阵派兵遣将'。"

他进一步总结道："至此，这 5 种模式已完整覆盖 SDD 工作流中的

5 个核心问题：‘做什么’（对应规划与执行模式）、‘谁来做’（对应层级委托模式）、‘往哪送’（对应条件路由模式）、‘做得对不对’（对应生成—评审模式）、‘不做什么’（对应护栏三明治模式）。”

8.6 实战：用 Agent 模式落地 SDD 自动化

前面已对五大模式的核心概念及其映射关系进行了详细阐述。然而，若仅停留在理论层面，读者难免会产生与 Vibe Coding 相似的困惑——“听起来有道理，但不知道怎么用”。

本节将结合“智能日报生成器”项目的真实开发场景，深入剖析如何将这五大模式应用于工程实践。

8.6.1 生成—评审模式：一条指令启动自我审查循环

在 Claude Code 中，不需要搭建两个独立 Agent，一条精心构造的指令即可让 AI 扮演生成器与评审器的双重角色。其核心在于：**评审标准必须严格源自规范文档，而非 AI 自行臆造。**

```
请实现 Task 3（GitHub 采集模块）。

实现要求：
- 按照 specs/design.md §4.1 的接口契约实现 collector/github.py
- 函数签名必须与 CollectResult 返回类型完全匹配
- 参考 specs/tasks.md Task 3 的验收标准

实现完成后，请逐条自查：
1. 函数签名是否符合 specs/design.md §4.1 的定义？
2. specs/tasks.md Task 3 的所有验收标准是否均已满足？
   - [ ] 能获取过去 24h 的 Commit 记录
   - [ ] API 超时具备重试机制（最多 3 次）
   - [ ] 超出 GitHub API 限流时能等待后重试
3. 是否符合 specs/design.md §6 定义的错误处理策略？
4. 日志是否采用 JSON lines 格式（参考 specs/design.md §7.1）？
```

```
修正机制：
如果存在不符合项，请自行修改代码后再次自查，直至全部通过。
设定最多循环 3 次——若 3 次迭代后仍未通过，请列出所有未解决问题并停止尝试。
```

这条指令的设计体现了三个关键要素。

第一，**评审标准全部锚定于外部规范**。它摒弃了“请检查代码质量”这类模糊指令，转而采用“函数签名是否符合 specs/design.md §4.1 的定义”这种精确到具体文件和章节的硬性约束，确保评审有据可依。

第二，**验收标准用清单格式**。通过结构化检查列表，促使 AI 逐条核对、逐项汇报，从而有效避免检查过程中的疏漏。

第三，**设定了严格的循环上限**。“最多 3 次”的限制可防止 AI 陷入无休止的自我修正死循环——如果 3 次迭代后仍无法通过，通常意味着问题可能出在规范本身，此时需要引入人工干预。

8.6.2　层级委托模式：用子智能体并行执行独立任务

在 SDD 的 tasks.md 中（见第 6 章），Task 3（GitHub 采集）、Task 4（飞书任务采集）和 Task 5（飞书消息采集）被定义为三个相互独立的任务。由于它们之间不存在数据依赖，因此具备并行执行的条件。

在 Claude Code 中，可以通过一条指令让主智能体扮演 Manager Agent 的角色，同时调度三个子智能体并行作业。

```
请并行执行以下三个独立任务，每个任务使用独立的子智能体执行。

## 任务分配

### 子智能体 A：Task 3 — GitHub 采集
- 输入规范：specs/design.md §4.1（接口契约）+ specs/tasks.md Task 3
- 产出：collector/github.py + tests/test_github.py
- 验收：能获取指定仓库过去 24h 的 Commit 记录

### 子智能体 B：Task 4 — 飞书任务采集
- 输入规范：specs/design.md §4.2（接口契约）+ specs/tasks.md Task 4
- 产出：collector/lark_task.py + tests/test_lark_task.py
```

```
- 验收：能获取当日状态变更的任务列表

### 子智能体 C：Task 5 — 飞书消息采集
- 输入规范：specs/design.md §4.3（接口契约）+ specs/tasks.md Task 5
- 产出：collector/lark_message.py + tests/test_lark_message.py
- 验收：能提取包含指定关键词的群聊消息

## 共同约束（所有子智能体必须遵守）
- 返回类型统一使用 CollectResult（见 specs/design.md §3.2）
- 错误处理策略遵循 specs/design.md §6
- 每个模块完成后运行 pytest 确认通过

三个子智能体全部完成后，请汇总结果并报告各任务的通过状态。
```

请注意指令中的“共同约束”部分——这正是 design.md 的核心价值所在。虽然三个子智能体在隔离的上下文窗口中独立工作，但其输出格式被同一份接口契约严格锁定，确保了后续代码拼装时的无缝衔接。如果没有 design.md 作为统一标准，三个子智能体极可能各自定义一套返回类型，导致最终汇总时陷入集成灾难。

8.6.3 护栏三明治模式：用 Hooks 实现自动化规范检查

Claude Code 的 Hooks 机制是实现护栏的天然工具。以下为基于“智能日报生成器”项目的实际配置示例，展示了如何通过代码级约束将 SDD 规范转化为可执行的自动化检查。

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "python scripts/input_guard.py"
          }
        ]
```

```
        }
      ],
      "PostToolUse": [
        {
          "matcher": "Edit|Write",
          "hooks": [
            {
              "type": "command",
              "command": "python scripts/output_guard.py"
            }
          ]
        }
      ]
    }
}
```

input_guard.py 在 AI 修正代码前触发，其核心逻辑包含三条源自规范的硬性规则。

```
"""输入护栏：在 AI 修改代码之前检查"""
import json, sys

tool_input = json.loads(sys.stdin.read())
file_path = tool_input.get("file_path", "")

# 规则 1：不允许修改规范文件（规范是人类的领地）
if file_path.startswith("specs/"):
    print(json.dumps({"decision": "block",
          "reason": "规范文件只能由人类修改，不允许 AI 直接改动 specs/
目录"}))
    sys.exit(0)

# 规则 2：不允许在 proposal.md 划定的排除范围内创建文件
#         （proposal.md §2.2："不做智能摘要功能"）
if "summarizer" in file_path or "nlp" in file_path:
    print(json.dumps({"decision": "block",
        "reason": "proposal.md §2.2 明确排除了智能摘要功能"}))
    sys.exit(0)
```

```
    # 规则 3：不允许硬编码密钥（design.md §7.2 安全约束）
    content = tool_input.get("content", "")
    if any(keyword in content for keyword in
              ["GITHUB_TOKEN =", "LARK_APP_SECRET =", "SMTP_
PASSWORD ="]):
        print(json.dumps({"decision": "block",
             "reason": "design.md §7.2 要求密钥通过环境变量注入，禁止
硬编码"}))
        sys.exit(0)

    print(json.dumps({"decision": "allow"}))
```

output_guard.py 在 AI 完成代码修正后自动运行，主要负责执行测试验证任务。

```
    """输出护栏：在 AI 修改代码之后检查。"""
    import subprocess, json

    result = subprocess.run(
        ["python", "-m", "pytest", "tests/", "-q", "--tb=short"],
        capture_output=True, text=True, timeout=60
    )

    if result.returncode != 0:
        print(json.dumps({
            "decision": "block",
              "reason": f"测试未通过，请修复后重试：\n{result.
stdout[-500:]}"
        }))
    else:
        print(json.dumps({"decision": "allow"}))
```

每一条护栏规则都能精确追溯到具体规范条款——这正是 SDD 驱动的护栏与“凭感觉设立的护栏”之间的本质区别。当规范发生变更时，护栏随之更新；当护栏触发拦截时，开发者能明确指出拦截依据（如“因为 proposal.md §2.2 明确说明不做这个功能”）。

8.6.4　4 种模式之间的组合

小冰在仔细研读这些实战示例后，敏锐地指出："我发现这 4 种模式并非孤立使用，而是可以形成嵌套结构。"

"能举个例子吗？"咖哥问。

"当然。例如，在层级委托模式中，主智能体派出三个子智能体并行工作；而每个子智能体内部，又可以运行生成—评审循环进行自我审查；与此同时，Hooks 护栏作为最外层的兜底机制，实时监控所有代码变更。这就构成了三层嵌套，但关键在于，每一层所遵循的标准都源自同一份 design.md。"

"这正是 SDD 的结构性优势所在。"咖哥总结道，"这 4 种模式共享同一套规范作为输入源——你不需要为每种模式单独维护一套规则。只要写好三份核心规范，这 4 种模式就能自动获取各自所需的执行标准，实现'一处定义，处处生效'"。

8.7　5 种模式组合构建 SDD 半自动化流水线

8.7.1　从手动到半自动

将前面介绍的 5 种模式有机组合，我们可以构建出一条高效的 SDD 半自动化流水线（见图 8-12）。

8.7.2　人类的角色变迁

相较于传统开发模式，在图 8-12 所示的流水线中，人类的角色经历了根本性转变，从"全程操控"演变为"两端把关"。

上游专注于定义规范，明确"做什么""怎么做"以及"按什么顺序做"；下游则负责审查结果，判断代码质量是否达标以及是否合入。

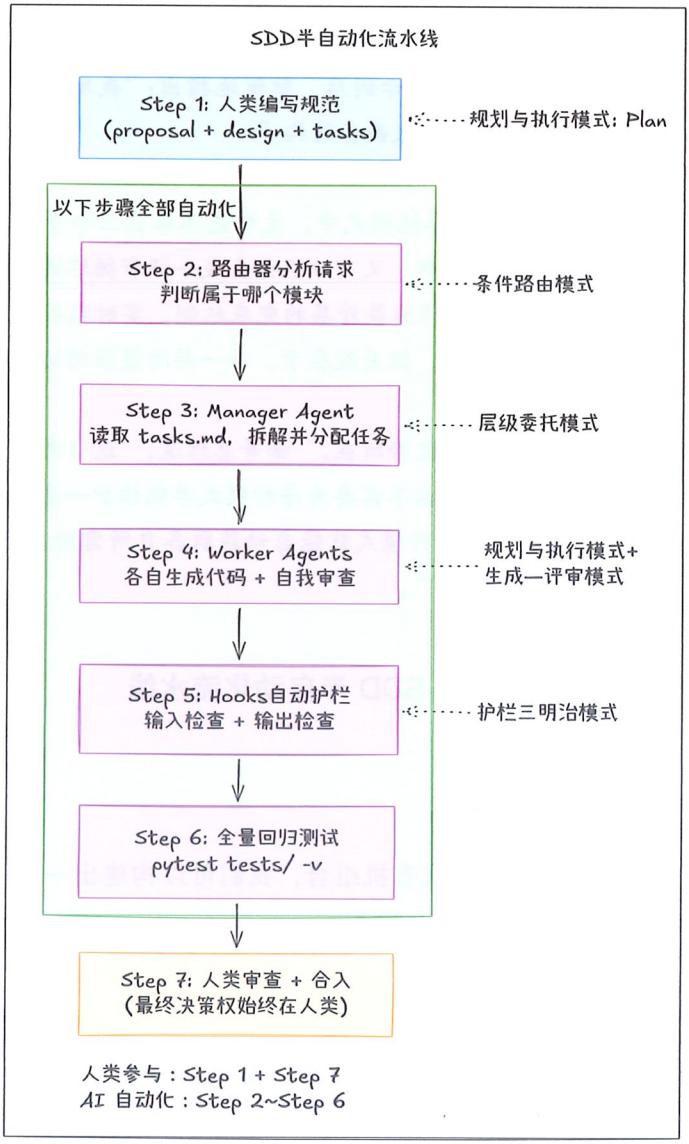


图 8-12　5 种模式组合构建的 SDD 半自动化流水线

中间烦琐的执行、验证与修复环节，则全部交由 AI 严格按照规范自动完成。

这并不意味着人类的作用被削弱——恰恰相反，**人类的决策被高度浓缩至最具杠杆效应的关键环节**。在 proposal.md 中写下“不做智能摘要功能”这一句话，就能有效阻止 AI 在后续步骤中尝试引入 NLP 模块；一条验收标准的精准措辞，就能决定 AI 是采用简单的列表还是复杂的嵌套结构来组织日报。

正如 Frederick Brooks 在《人月神话》中所做的区分，软件复杂性包含**本质复杂性**（Essential Complexity）和**偶发复杂性**（Accidental Complexity）。规范编写所处理的正是本质复杂性，即“系统应该做什么”这一核心问题，对此没有捷径可走。而代码生成主要处理的是偶发复杂性，即“代码怎么写”，这在很大程度上是可被自动化的。

SDD 与 Agent 设计模式的强强联合，正是让人类回归核心价值，聚焦于本质复杂性的定义，而将偶发复杂性的实现放心地交给 AI。

8.7.3　不是完全自动化

需要诚实地指出：这条流水线是**半自动化**的，而非完全自动化。原因有三。

第一，**规范的编写无法实现自动化**。尽管 AI 可以辅助生成规范初稿，但最终的需求判断、架构决策以及权衡取舍，都离不开人类对业务和技术背景的深刻理解。

第二，**规范的审查无法实现自动化**。AI 能够检查代码是否符合规范（语法层面），却无法判断规范本身是否合理（语义层面）。如“接口应该返回 CollectResult 还是裸列表”，这类决策需要依赖人类的工程经验。

第三，**新问题的发现无法实现自动化**。正如 7.2.3 小节的“从代码 bug 发现规范缺陷”场景，这往往需要人类的洞察力——AI 可以报告测试失败，但诊断“失败的根本原因是接口设计不合理，而非代码逻辑错误”，目前仍需要人类介入。

表 8-2 总结了 SDD 自动化的边界。

表 8-2　SDD 自动化的边界

环节	自动化	原因
规范编写	半自动（AI 生成初稿，人类定稿）	需求判断和架构决策是本质复杂性
任务执行	能	有明确的输入、输出和验收标准
代码审查	能	规范提供了客观标准
回归测试	能	测试用例从规范衍生
规范缺陷发现	否	需要人类的工程洞察力
最终合入决策	否	最终责任在人类

关于 5 种 Agent 设计模式在 4 种工具中的落地，请访问 https://kage-ai.com/sdd/#ch8-mapping。

8.8　从 Agent 设计模式到 Harness Engineering

前面介绍了 Agent 设计模式：规划与执行模式、生成—评审模式、层级委托模式、护栏三明治模式和条件路由模式。将它们放在一起看，就会发现它们已不只是“Prompt 技巧”，而是构成了一个运行环境。

这就是近来被频繁讨论的 Harness Engineering。一个 Agent 并非独立模型，而是 Model（模型）+Harness——模型负责推理与生成，Harness 负责将模型接入工具、上下文、权限、文件系统、测试、日志、回滚与人工审批。

如果将模型比作发动机，Harness 就是整台机器的传动系统、仪表盘、刹车和安全笼。模型越强，Harness 越重要。因为模型能做的事情越多，系统越需要明确“它可以看什么、能调用什么、什么时候停下来、怎样证明自己做对了”。

SDD 与 Harness 的关系，可概括为：SDD 定义意图契约，Harness 负责执行契约。proposal.md 明确目标与边界，design.md 规定架构与接口约束，tasks.md 细化步骤与验收标准；Harness 再将这些约束加载至

上下文，路由给合适的 Agent，在执行前后运行检查，并将结果反馈给人类。

对应关系如下。

规划与执行模式→ Harness 中的计划器与执行器边界。

生成—评审模式→ Harness 中的验证器、审查器与质量门禁。

层级委托模式→ Harness 中的任务分发与上下文隔离。

护栏三明治模式→ Harness 中的权限控制、输入过滤与输出检查。

条件路由模式→ Harness 中的工具选择、模块路由与责任边界。

这也是本书不将 Agent 工具写成操作手册的原因。具体工具会迭代，但 Harness 必须解决的问题相对稳定，包括上下文如何接入、工具如何受控、执行如何验证、反馈如何沉淀。SDD 提供的正是这些问题的上游答案。

当你将第 4 章到第 7 章的规范实践，与本章的 5 种 Agent 设计模式结合，便得到了一条更完整的工业化路径：人类定义规范，Harness 承载规范，Agent 按照规范执行，验证结果再反过来修正规范。

本章小结

本章是 SDD 方法论与 Agent 工程实践的交汇点。我们看到了 5 种 Agent 设计模式如何与 SDD 的三份规范天然契合，也看到了这些模式如何进一步汇入 Harness Engineering：将模型放进一个受约束、可验证、可追踪的执行环境中。

- **规划与执行模式**：SDD 的三份规范就是规划器的产出物。proposal.md 是目标，design.md 是约束，tasks.md 是步骤列表。在 Claude Code 中，CLAUDE.md 将规范注入 AI 的规划上下文。
- **生成—评审模式**：SDD 规范为评审器提供了客观的审查标准。验收标准并非 AI 自行臆造的，而是人类预先定义的，这让自动化代码审查变得可靠。
- **层级委托模式**：tasks.md 是现成的任务分配表，design.md 是

Worker Agent 的技术手册。Manager Agent 不需要“创造性拆解”，只需要“按规范分配”。

- **护栏三明治模式**：proposal.md 的“不做什么”与 design.md 的安全约束直接转化为护栏规则。Claude Code 的 Hooks 机制是实现自动化护栏的天然工具。
- **条件路由模式**：design.md 的模块边界就是路由器的分发规则。新请求进入时，路由器依据模块职责判断“这属于谁的事”，并精准分发至对应模块。

这 5 种模式精准覆盖了 SDD 工作流中的 5 个关键问题：“做什么”（对应规划与执行模式）、“谁来做”（对应层级委托模式）、“往哪送”（对应条件路由模式）、“做得对不对”（对应生成—评审模式）、“不做什么”（对应护栏三明治模式）。

8.6 节的实战示例展示了这些模式并非停留在架构图上的概念——通过一条精心构造的 Claude Code 指令、一段子智能体分配 Prompt，或是一个 Hooks 配置文件，即可将它们落地为可运行的自动化流水线。8.8 节进一步说明，当这些模式被组合进工具、上下文、权限、验证与反馈机制中，它们便构成了 Harness Engineering 的基本骨架。



咖哥发言

很多人对 AI Agent 存在两种极端的想象：一种是“完全替代人类”，另一种是“就是个聊天机器人”。现实其实介于两者之间。

Agent 设计模式告诉我们，AI 完全可以在有明确规则与标准的环节中可靠地工作，如执行计划、审查代码、验证合规。但在需要判断与创造的环节，如定义需求、进行架构取舍、发现深层问题，人类仍不可替代。

SDD 的价值在于，它让“有明确规则”的部分尽可能多，让“需要判断”的部分尽可能集中；其结果是：人类在最高杠杆的决策上投入精力，AI 在可标准化的执行上投入算力。

这并非人类被替代，而是人类的能力被放大。

思考题

1. 本章展示了 5 种 Agent 设计模式与 SDD 的结合方式，你能想到其他 Agent 设计模式与 SDD 的结合方式吗？例如，思维链（Chain-of-Thought）模式引导模型逐步推理并得出结论；工具分发（Tool Dispatch）模式则根据任务类型自动匹配并调用合适的工具。
2. 在 8.7.3 小节中，我们曾提出“新问题的发现无法实现自动化”的观点。然而，如果引入 AI 对规范进行审查（而非审查代码），例如，自动监测 proposal.md 与 design.md 之间是否存在逻辑矛盾，这是否可以被视为一种“规范的自动化审查”？其可靠性又该如何评估？
3. 在 8.6.3 小节的 input_guard.py 中，规则 1 规定“不允许 AI 直接改动 specs/ 目录”。然而，若 AI 在执行过程中发现规范存在疏漏（例如，接口契约缺失字段），并试图通过修改 design.md 来修正时，该操作会被护栏机制拦截。针对此类“护栏误报”现象，你认为应如何优化处理？
4. 在 8.6.2 小节的子智能体分配 Prompt 中，“共同约束”部分要求所有子智能体严格遵守同一份 design.md 的接口契约。然而，若 design.md 本身存在错误（例如，CollectResult 的字段名拼写错误），将导致三个子智能体会“整齐划一地重复同一个错误”。相较于各自独立犯错的情况，你认为哪种情形更为危险？为什么？

第 9 章
众人拾柴：团队中的 SDD 实践

交响乐团的魅力，并非仅仅源于每位成员都技艺超群，而在于他们遵循同一份乐谱，实现和谐共鸣。

小冰的智能日报生成器上线后，效果出乎意料地好。运营团队每日早上均可收到条理清晰的日报，管理层也开始关注这套 SDD 工作流。

周会上，CTO 问她："你这套'先写规范再写代码'的方法，能在全体后端团队推广吗？"

小冰犹豫了。自己单人用 SDD 做项目时得心应手，但要让整个 8 人团队都按这套流程执行，问题便多了起来——proposal.md 谁来撰写？design.md 谁来审核？tasks.md 与代码对不上怎么办？每个人的 Claude Code 配置如何统一？

她只得去找咖哥求助。

咖哥并未直接回答，而是反问她："你去过交响乐团的排练现场吗？"

小冰摇了摇头。

咖哥答：

"一个交响乐团拥有数十名乐手，他们手持小提琴、大提琴、双簧管、圆号等各式乐器进行演奏——他们是如何做到同时演奏出和谐音乐的？并非仅仅因为每位乐手都技艺超群，而是因为他们遵循同一份乐谱。乐谱精准定义了每个声部在每个小节的具体演奏内容。有了乐谱，第二小提琴手不需要时刻关注圆号的演奏——他只需要专注于自己的声

部，在正确的时间奏出正确的音符。

“SDD 的规范正是团队的乐谱。其中，proposal.md 确定了‘曲目选择’，design.md 构成了‘总谱’，而 tasks.md 则是具体的‘分声部谱’。有了这些规范，团队成员便不需要在聊天群里反复确认‘你那个接口是什么格式’——查阅规范，一切便知。”

9.1　从个人到团队：三个阶段

9.1.1　为什么不能一步到位

团队引入 SDD 时，最常见的失败模式是试图“一步到位”：CTO 宣布“从明天开始，所有项目全面实行 SDD”，随即下发一套模板、规定一个流程，期望团队能立即完美执行。

这无异于要求一个从未跑过步的人明天去跑马拉松。理论上并非绝无可能，但大概率会在中途因不堪重负而放弃。

正确的策略应当是渐进式引入。图 9-1 展示了一条经过实践验证的团队引入 SDD 的三阶段演进路径。

9.1.2　第一阶段的关键：选对试点人和试点项目

试点人选的确定，不必局限于团队中技术最顶尖的专家，但必须是对新方法论持最开放态度的成员。一个对“多写文档”抱有抵触情绪的资深工程师，绝非理想的试点人选——这并非质疑其技术能力，而是因其负面反馈极易在团队中产生放大效应，阻碍后续推广。

试点项目的选择同样大有讲究。表 9-1 列出了试点项目的选择标准。

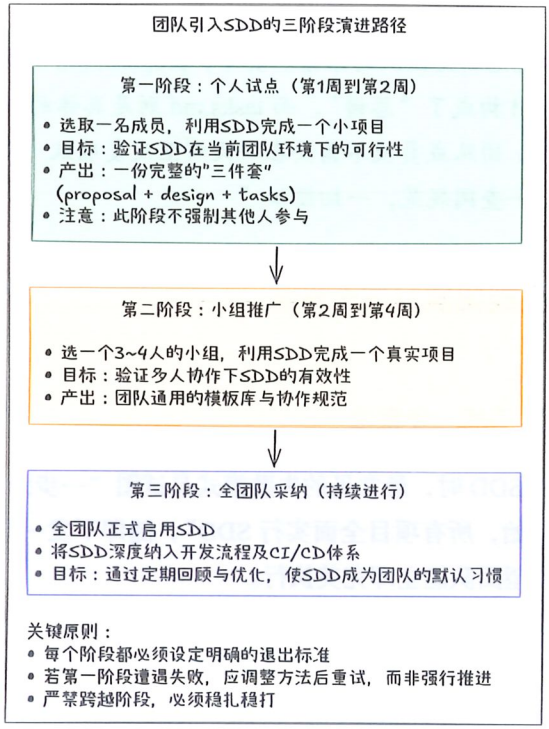


图 9-1　团队引入 SDD 的三阶段演进路径

表 9-1　试点项目的选择标准

维度	好的试点项目	差的试点项目
规模	1 至 2 周可完成	耗时数月的庞大项目
紧急程度	非最高优先级	火烧眉毛的救火项目
复杂度	中等（需求明确）	极简或极度复杂
独立性	相对独立，无强外部依赖	与多个系统紧耦合
可见性	有具体、可视化的产出	纯内部工具，效果不可见

小冰的“智能日报生成器”便是一个理想的试点项目——它周期短（1 至 2 周可完成）、需求清晰、产出直观（每日生成一份日报），且技术复杂度适中，非常适合作为 SDD 的切入点。

9.2　规范的协作模式：谁撰写、谁审核、谁执行

9.2.1　三种角色

当 SDD 从个人实践扩展至团队协作时，一个核心问题随之浮现：规范究竟由谁负责？

在个人项目中，规范的编写、审查与执行均由一人包揽；然而，在团队环境中，这三个环节必须明确分工，各司其职。图 9-2 展示了规范协作的三种角色及其职责。

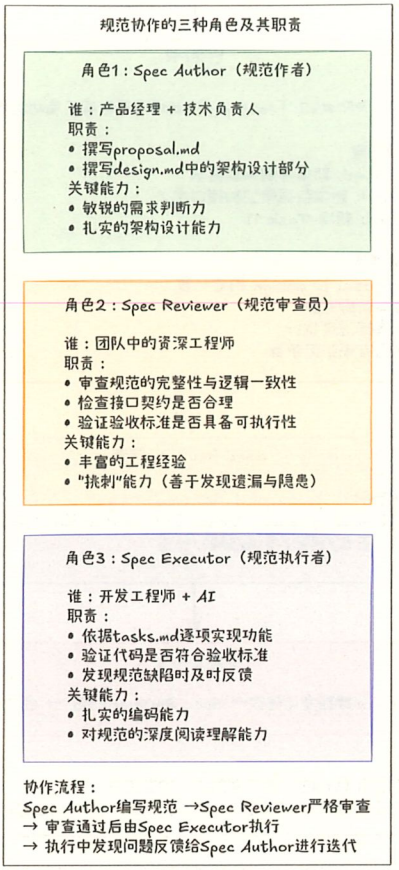


图 9-2　规范协作的三种角色及其职责

9.2.2 规范审查的 PR 流程

规范文档与代码一样，必须经过审查才能合入主分支。最自然且高效的方式是复用 Git 的 PR 流程，如图 9-3 所示。

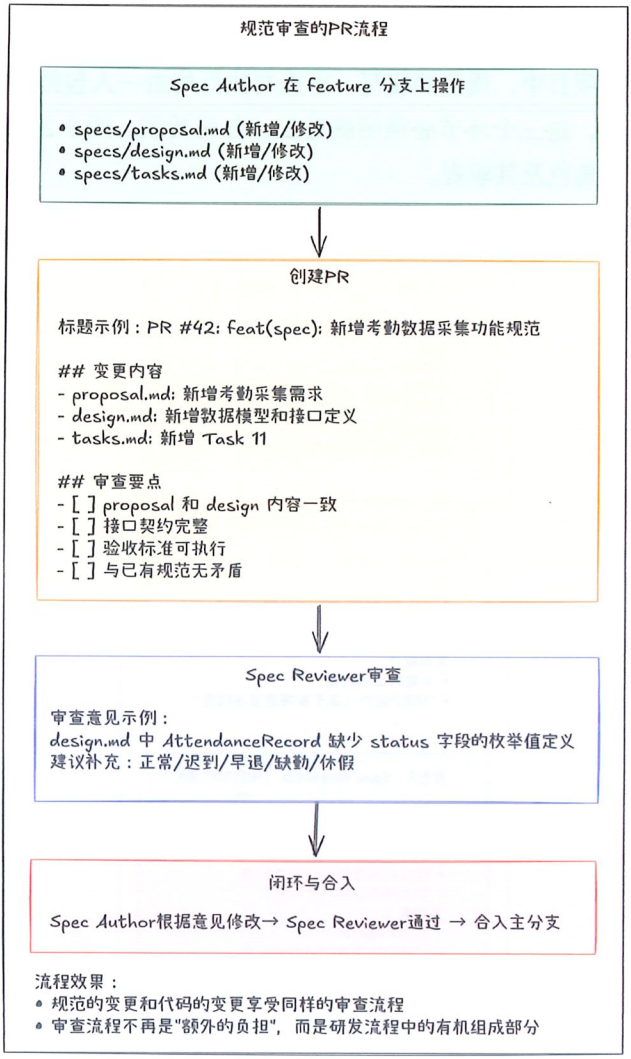


图 9-3 规范审查的 PR 流程

小冰问："规范的 PR 与代码的 PR 应该是同一个还是分开？"

咖哥答："看情况。对于新功能开发，建议先提交规范 PR，待审查通过后再提交代码 PR。这样 Spec Reviewer 可以先审查'你要做什么'（规范），再审查'你做得对不对'（代码），逻辑更清晰。对于小改动（如修复一个 bug），规范和代码可以在同一个 PR 里——只要 Commit 记录有明确区分即可。"

9.3　规范模板的团队标准化

9.3.1　为什么需要统一模板

当团队采用 SDD 工作流时，一个自然浮现的问题是：成员撰写的 proposal.md 格式各不相同。张三的 proposal.md 包含"不做什么"的范围界定，李四的则没有；王五的 design.md 包含 ADR，赵六的却将其省略。

这就如同交响乐团中，如果每位乐手都使用不同的记谱法——虽然各自能读懂自己的乐谱却无法理解他人意图，导致协作效率大幅降低。

解决之道其实很简单：**建立团队级规范模板库**（见图 9-4）。

9.3.2　模板的弹性

模板并非僵化的固定格式，它定义的是**最低标准**，而非最高限制。

例如，proposal.md 模板规定"不做什么"为必填项，但这并不妨碍团队成员在此基础上扩展"竞品分析"或"用户故事"等章节；design.md 模板要求"至少包含一个 ADR"，对于技术选型复杂的项目，完全可以撰写 5 个甚至更多 ADR。

小雪曾用一个精准的类比来阐述这一理念："模板就像建筑规范中的**最小净高要求**——它规定住宅层高不得低于 2.8m，但完全可以打造 3.5m 的奢华挑高。最低标准保证了基本品质，而'上不封顶'的原则则为团队留出了充分的发挥空间。"

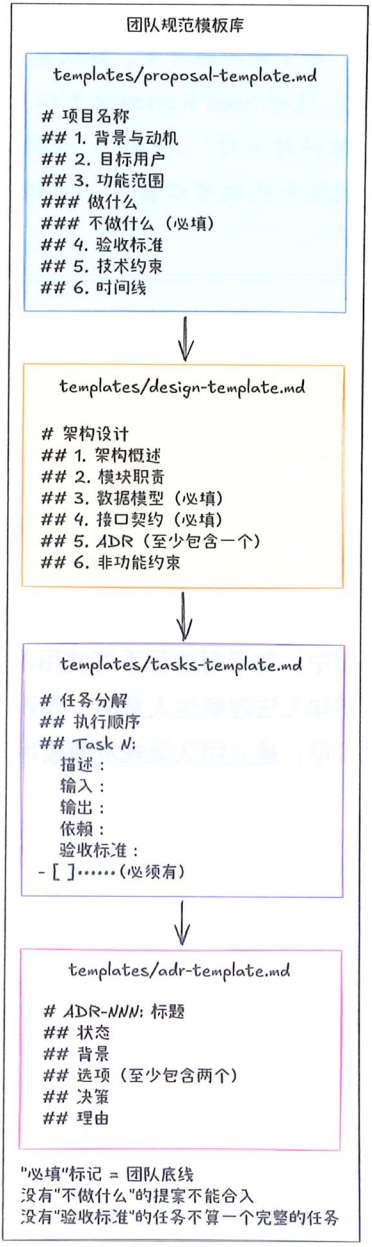


图 9-4　团队规范模板库：统一格式，标注必填项

9.4　与现有开发流程的融合

9.4.1　SDD+Scrum

大多数团队已采用某种敏捷开发流程（如 Scrum、看板等）。SDD 的目的绝非替代这些流程，而是无缝嵌入其中（见图 9-5）。

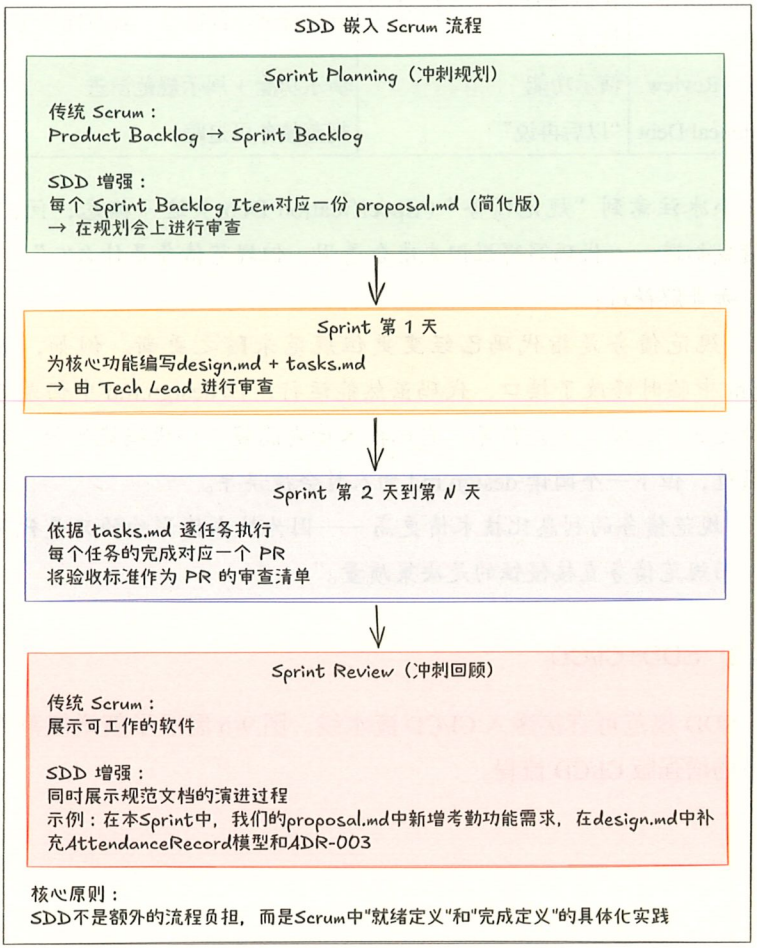


图 9-5　SDD 嵌入 Scrum 流程

表 9-2 展示了 SDD 与 Scrum 中既有概念的映射关系。

表 9-2　SDD 与 Scrum 概念的映射关系

Scrum 概念	传统做法	SDD 做法
Definition of Ready	口头确认“需求清楚了”	proposal.md 审查通过
Sprint Planning	估点 + 分任务	撰写 design.md+tasks.md
Definition of Done	代码能运行 + 代码评审通过	验收标准全部通过 + 规范同步更新
Sprint Review	演示功能	演示功能 + 展示规范演进
Technical Debt	“以后再说”	规范债务可追踪

小冰注意到“规范债务”（Specification Debt）这一概念，问：“技术债我知道——代码写得烂但先凑合着用，但规范债务是什么？”

咖哥解释道：

“规范债务是指代码已经变更但规范未随之更新。例如，你在 Sprint 中临时修改了接口，代码虽然能运行，但 design.md 中仍是旧的接口定义。这就是规范债务。它比技术债更隐蔽——代码能运行，测试能通过，但下一个阅读 design.md 的人员会被误导。

“规范债务的利息比技术债更高——因为技术债影响的只是代码质量，而规范债务直接侵蚀的是决策质量。”

9.4.2　SDD+CI/CD

SDD 规范可直接融入 CI/CD 流水线。图 9-6 展示了新增规范检查环节的增强版 CI/CD 流程。

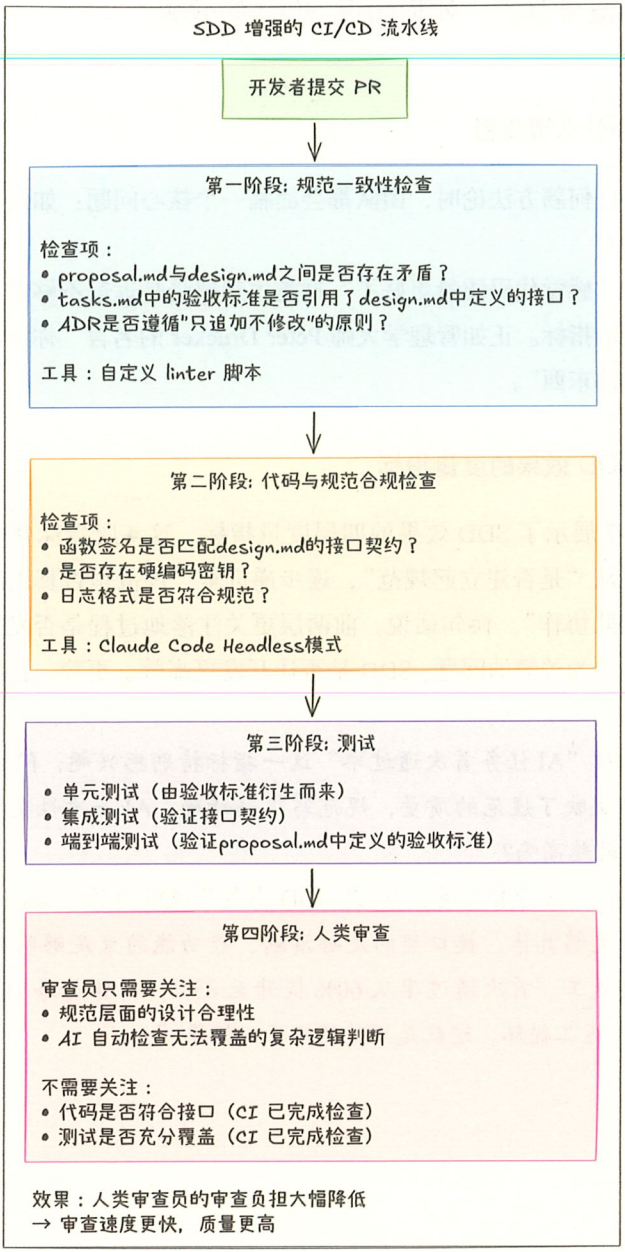


图 9-6　SDD 增强的 CI/CD 流水线

9.5 度量与改进：如何衡量 SDD 的效果

9.5.1 为什么要度量

推行任何新方法论时，团队都会面临一个核心问题：如何证明该方法论有效？

仅凭“感觉代码质量更好了”这类主观感受是远远不够的，需要的是可量化的指标。正如管理学大师 Peter Drucker 的名言“你无法管理你无法度量的东西”。

9.5.2 SDD 效果的度量指标

图 9-7 展示了 SDD 效果的四层度量指标。这 4 层指标并非彼此割裂，而是从“是否建立起规范”，逐步递进至“规范是否真正改善开发结果和团队协作”。换句话说，前两层更关注落地过程是否发生，后两层则回答更为关键的问题：SDD 是否让开发更高效、更稳，让团队更容易协作。

小冰对“AI 任务首次通过率”这一指标特别感兴趣，问道：“这个指标直接反映了规范的质量，规范写得越精确，AI 一次性交付合格代码的概率就越高吗？”

“没错。”咖哥肯定道，“这是 SDD 中最具说服力的指标。如果你的验收标准足够具体、接口契约足够清晰、非功能约束足够明确，AI 几乎不需要返工。首次通过率从 60% 提升至 85%，意味着每 100 个任务少了 25 次返工循环。这就是规范的投资回报率。”

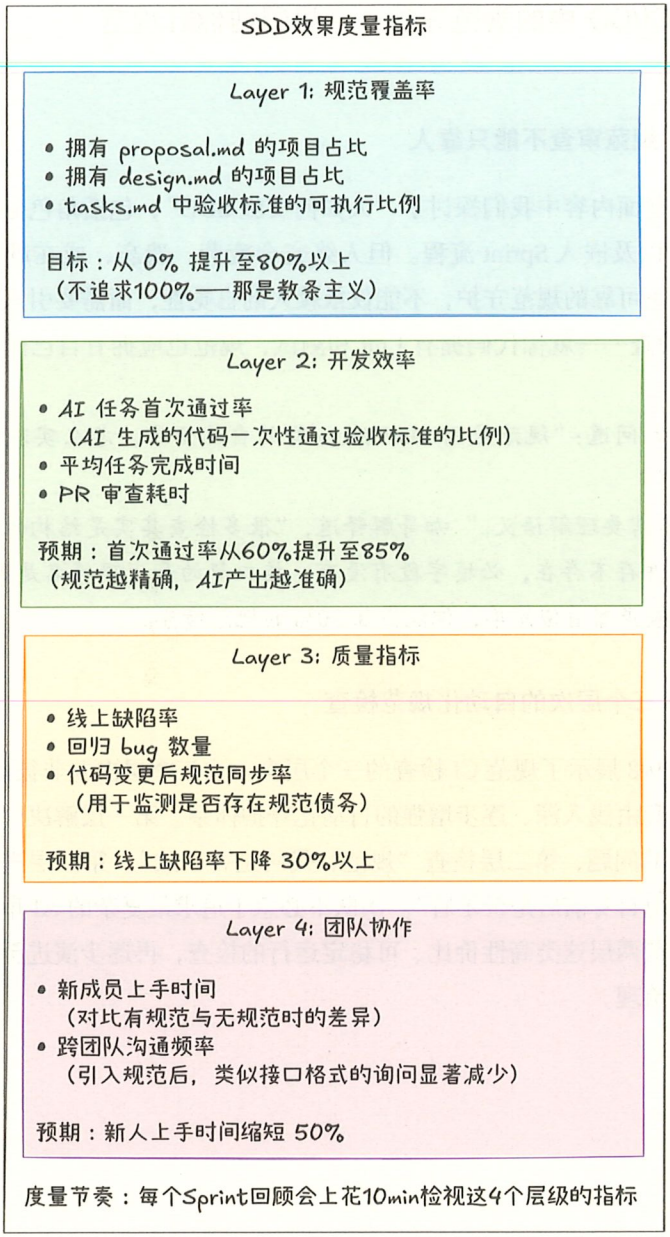


图 9-7　SDD 效果的四层度量指标

9.6 CI/CD 中的规范守护：让机器帮你盯规范

9.6.1 规范审查不能只靠人

在前面内容中我们探讨了“人如何实施 SDD”，包括角色分工、模板制定以及嵌入 Sprint 流程。但人终究会疲劳、遗忘，或在赶工时偷懒。真正可靠的规范守护，不能仅依赖人的自觉性，而需要引入自动化的 CI 检查——就像代码拥有 Lint 和测试，规范也应拥有自己的“规范 Lint”。

小冰问道：“规范是用 Markdown 写的自然语言，怎么实现自动化检查？”

“不需要理解语义。”咖哥解释道，“很多检查其实是结构性的，例如，文件存不存在，必填字段有没有，接口契约和代码签名是否匹配。这些检查虽然看似简单，但能拦住 80% 的规范债务。”

9.6.2 三个层次的自动化规范检查

图 9-8 展示了规范 CI 检查的三个层次。这三个层次并非彼此替代，而是一套由浅入深、逐步增强的自动化守护体系。第一层解决“规范有没有”的问题，第二层检查“规范和代码是否一致”，第三层进一步核验“规范自身前后是否矛盾”。团队不必急于追求最复杂的 AI 审查，应先落地前两层这类高性价比、可稳定运行的检查，再逐步演进至更智能的规范治理。



图 9-8　规范 CI 检查的三个层次

9.6.3　实战：一个最小的规范 CI 工作流

以下是一个可直接集成至 GitHub Actions 的规范检查工作流。该工作流仅实现 Level 1 检查——虽最简单，却最为实用。

```
# .github/workflows/spec-check.yml
name: SDD Spec Check
on:
  pull_request:
    paths:
      - 'src/**'
      - 'specs/**'
```

```
jobs:
  spec-check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: 检查规范文件完整性
        run: |
          errors=0

          # 检查三份核心规范是否存在
          for f in specs/proposal.md specs/design.md specs/tasks.md; do
            if [ ! -f "$f" ]; then
              echo "缺少规范文件：$f"
              errors=$((errors + 1))
            fi
          done

          # 检查 proposal.md 是否包含“不做什么”部分
          if [ -f specs/proposal.md ]; then
            if ! grep -q "不做" specs/proposal.md; then
              echo "proposal.md 中缺少“不做什么”章节"
              errors=$((errors + 1))
            fi
          fi

          # 检查 tasks.md 中每个任务是否包含验收标准
          if [ -f specs/tasks.md ]; then
            tasks=$(grep -c "^## Task" specs/tasks.md || true)
            acceptance=$(grep -c "验收" specs/tasks.md || true)
            if [ "$acceptance" -lt "$tasks" ]; then
              echo "tasks.md 中包含 $tasks 个任务，"
              echo "但只有 $acceptance 处提及验收标准"
            fi
          fi

          # 检查是否修改了代码但未更新规范
            if git diff --name-only origin/main | grep -q "^src/"; then
            if ! git diff --name-only origin/main | grep -q
            "^specs/"; then
              echo "修改了 src/ 但未更新 specs/——请确认是否需要同
```

```
        步规范 "
    fi
fi

if [ $errors -gt 0 ]; then
    echo "规范检查未通过（$errors 个错误）"
    exit 1
fi
echo "规范检查通过"
```

上述工作流的设计哲学为**宽严相济**：缺少核心规范文件属于硬性错误（将阻止合并），而“修改了代码但未更新规范”仅作为警告（该场景可能属于合理的 bug 修复）。团队可根据实际情况灵活调整检查的严格程度。

小冰在试用该工作流后的第一周，便成功拦截了两次“规范未及时更新”的 PR——此前这类问题需要等到代码审查阶段才会被发现，有时甚至直接会被合入主分支。

“**规范 CI 不会替代规范审查，正如代码 Lint 不会替代代码审查。**但它能在审查员投入时间之前，先过滤掉最低级的遗漏。”咖哥说。

本章小结

本章探讨了如何将 SDD 从个人方法论扩展为团队工程实践。技术方法论的真正价值，不在于单兵能力的精良，而在于团队能形成共识与协作合力。

本章核心要点如下。

- **渐进式引入**：遵循“个人试点→小组推广→全团队采纳”的路径。选对试点人员和试点项目是成功的关键。
- **三种角色分工**：明确 Spec Author（撰写规范）、Spec Reviewer（审查规范）、Spec Executor（执行规范）的职责。规范审查复用 PR 流程，与代码审查共享同一套基础设施。
- **统一模板**：建立团队规范模板库，将必填项标记为最低标准。模

板是底线而非优化上限，确保基础质量的同时保留发挥空间。

- **嵌入现有流程**：SDD 并非替代 Scrum 或看板，而是可无缝嵌入其中。proposal.md 对应“就绪定义”，验收标准对应“完成定义”。需要警惕规范债务，它比技术债更隐蔽，代价也更昂贵。
- **可度量、可改进**：建立四层度量指标体系（包括规范覆盖率、开发效率、质量指标、团队协作），建议在每次 Sprint 回顾会上预留 10min 检视，以数据驱动改进。
- **CI/CD 规范守护**：通过自动化工作流检查规范完整性、规范与代码一致性以及规范内部的一致性。即便是最简单的 Shell 脚本，也可作为起步方案，优先拦截最基础的规范遗漏。



咖哥发言

我见过太多团队引入新方法论的失败案例。究其原因，鲜少是方法论本身不佳，更多是引入方式过于激进。试图在一夜之间强制全员改变工作习惯，这并非科学的工程管理，而是一场鲁莽的“政变”。

渐进式引入的核心哲学在于：**用成果去说服，而非用命令去强推**。当试点项目的智能日报生成器展现出比传统方式快 40% 的效率，且 bug 数量减半时，团队成员会好奇地询问“你是怎么做到的”，此时再顺势推广 SDD，阻力几乎为零。

管理学大师 John Kotter 在《领导变革》一书中提出的“八步变革模型”，将“营造紧迫感”列为第一步。然而，在务实的工程团队中，比紧迫感更具威力的是“眼见为实”——让团队成员亲眼见证 SDD 带来的高质量产出，这比任何理论宣讲都更具说服力。

思考题

1. 在 9.2 节中，我们将 Spec Author、Spec Reviewer、Spec Executor 划分为三个独立角色。然而，在 3 到 4 人的小型团队中，一人分饰多角往

往是常态。你认为哪两个角色适合合并？哪两个角色绝对不能由同一个人兼任？请阐述你的理由。

2. “规范债务”往往比技术债更为隐蔽——代码或许能运行，测试也能顺利通过，但规范文档已滞后。你能设计一套自动化检查机制，来检测“规范与代码是否同步”吗？（提示：不妨从对比函数签名与 design.md 中的接口契约入手。）
3. 本章建议使用“AI 任务首次通过率”这一指标来度量规范质量。然而，该指标不可避免地会受到 AI 模型能力的影响——如果更换了性能更强的模型，首次通过率的提升可能源于模型本身的进化，而非规范的优化。你将如何控制这一变量，以确保度量的准确性？

第 10 章
大道至简：SDD 的未来与思考

最深刻的道理往往最简单——人类负责说清楚“要什么”，AI 负责将其实现。

小冰站在白板前，回望过去两个月的历程。

从第 1 章对着 AI 随口描述需求、生成大量无法运行的代码，到如今，她已能运用三份规范文档驱动一整条自动化开发流水线。智能日报生成器不仅成功上线，更完成了两次功能迭代，且每次迭代都留存了完整的规范变更记录。

“SDD 彻底重塑了我的工作流。”小冰感慨道，“但我仍心存疑惑——SDD 的边界究竟在哪里？它是否适用于所有场景？它的未来又将走向何方？”

咖哥示意她落座：“这正是本书最后一章要探讨的核心。如果说前 9 章是在传授 SDD 的‘术’，那么这一章，我们需要站在更高一层来审视 SDD 的‘道’——它究竟是什么，不是什么，以及终将演变成什么。”

10.1 SDD 的边界：什么场景适合，什么场景不适合

10.1.1 SDD 的甜蜜区

任何方法论都有其甜蜜区（Sweet Spot），SDD 最适合的场景具有图 10-1 所示的特征。

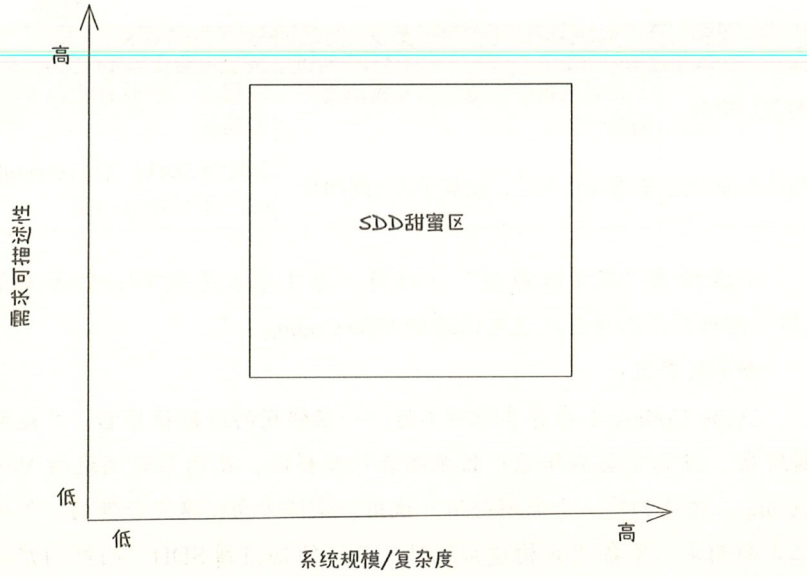


图 10-1　SDD 的甜蜜区：高可描述性与中高等复杂度的交汇

10.1.2　不适合 SDD 的场景

诚实地说，并非所有场景都适合完整的 SDD 流程。表 10-1 展示了不适合 SDD 的场景及替代方案。

表 10-1　不适合完整 SDD 流程的场景及替代方案

场景	为什么不适合	替代方案
探索性原型	需求本身不清楚，写规范是“伪精确”	Vibe Coding →明确方向后再实施 SDD
一次性脚本	投入产出比过低，规范比代码更长	直接写代码，加注释即可
纯算法研究	核心挑战不在“需求”而在“求解”	实验驱动，论文是“规范”

续表

场景	为什么不适合	替代方案
紧急热修复	没有时间撰写规范，需要先修复问题	先修复，事后补规范（规范债务）
极高不确定性	需求每日变更，规范无法及时同步	轻量级 SDD（只写 proposal.md，不写 design.md）

小冰指着“探索性原型”一行问：“第 1 章不是说 Vibe Coding 是‘坏’的吗？怎么这里又说可以进行 Vibe Coding？”

咖哥解释道：

“Vibe Coding 本身并非绝对不好——**在错误的阶段使用它，才是问题所在**。当你完全不知道产品应该长什么样时，花两天时间进行 Vibe Coding，快速构建一个原型给用户试用、验证方向，这是合理的。但一旦方向确定，准备正式构建系统时，就必须切换到 SDD。问题的症结不在于 Vibe Coding 本身，而在于**很多人在本该使用 SDD 的时候，仍然沉溺于 Vibe Coding**。

“换句话说，Vibe Coding 是探索的工具，而 SDD 是建设的工具。用探索的工具去搞建设，建出来的只能是空中楼阁；而用建设的工具去搞探索，则只会浪费时间绘制那些最终并不需要的图纸。”

10.1.3 行业验证：SDD 绝非一家之言

SDD 并非某位专家闭门造车的结果，而是 2025 年全球众多独立实践者、工具开发团队及咨询机构在实践中不约而同收敛形成的共识。以下几个关键信号值得重点关注。

1. GitHub Spec-Kit（2025 年）

全球最大的代码托管平台 GitHub 推出了官方的 SDD 框架，支持超过 22 种主流 AI 辅助编程工具。这绝非一个简单的社区实验项目，而是 GitHub 针对“AI 时代的软件工程该何去何从”这一问题给出的答案。

2. AWS Kiro（2025—2026 年）

Amazon 推出的 AI 开发环境，其核心工作流被严格定义为三步：

Specify → Plan → Execute。这与本书所传授的 proposal.md → design.md → tasks.md 流程精准地一一对应。当全球最大的云服务商将 SDD 产品 IDE 作为核心功能时，这一事实足以证明已不再是小众的极客实践。

3. 业界领袖背书

Google Chrome 团队的 Addy Osmani 于 2025 年专门撰写了文章“How to Write a Good Spec for AI Agents”，从一线实战的角度，有力印证了本书的核心论断——“规范的质量直接决定了 AI 产出的质量”。

这些信号共同指向一个无可辩驳的结论：**SDD 已不再是一种“可选的最佳实践”，而正在迅速成为 AI 辅助开发领域的行业标准。**当下的问题已不再是“要不要用 SDD”，而是“你应该何时开始使用”。

10.2　SDD 与其他方法论的融合

10.2.1　方法论不是金科玉律

软件工程的历史，某种程度上就是一部“方法论博弈演变史”：瀑布模型与敏捷开发的拉锯、TDD 与 BDD 的博弈、微服务与单体架构的轮回……在这场漫长的博弈与演变中，方法论的拥趸往往倾向于将其奉为“唯一真理”，试图用一把锤子解决所有问题。

实践 SDD 时，我们必须警惕，决不能重蹈这种教条化的覆辙。SDD 不应被视为对既有方法论的颠覆或取代，而是**在 AI 辅助编程这一全新语境下，为降低 AI 的不确定性而提供的一个必要的“补充层”。**

10.2.2　SDD 与主流方法论的关系

图 10-2 展示了 SDD 与三种主流方法论（TDD、BDD、DDD）之间的关系。

核心观点：**SDD 与任何传统方法论均不存在冲突，它们解决的是不同维度的问题。**TDD、BDD 和 DDD 致力于解决“人类如何更高效、更

正确地开发软件”的问题，其核心假设是代码由人亲手编写。而 SDD 则聚焦于“人类如何精准地指挥 AI 开发软件”，其代码由 AI 生成。

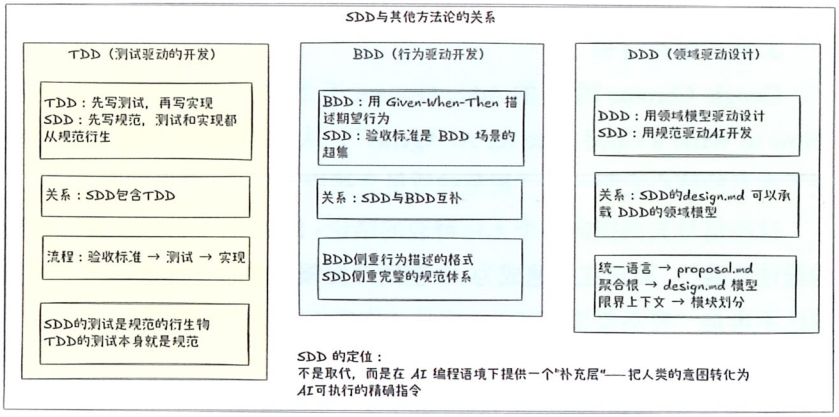


图 10-2　SDD 与主流方法论的关系

这是范式转移。主流方法论旨在优化“执行者”的能力，而 SDD 则旨在重塑“指挥者”的艺术。

10.3　AI 辅助编程方法论的演进方向

10.3.1　从辅助到协作，再到自治

AI 在编程中的角色正在快速演进，图 10-3 展示了这一演进的三个阶段。

这里有一个反直觉的洞察：AI 越自主，规范反而越重要——绝非越不重要。

小冰曾一度认为，SDD 不过是技术发展过渡期的权宜之计。“等 AI 足够强大了，我们就不需要再写烦琐的规范了。”她这样想道。但咖哥用一个精妙的类比，彻底纠正了她的认知偏差。

“你认为交通规则的存在，是因为汽车还不够好吗？恰恰相反——汽车跑得越快、数量越多、性能越强，交通规则就越关键。一辆速度仅

20km/h 的马车，或许不需要红绿灯的指挥；但若是速度 200km/h 的跑车失去了规则的约束，道路瞬间就会变成灾难现场。”

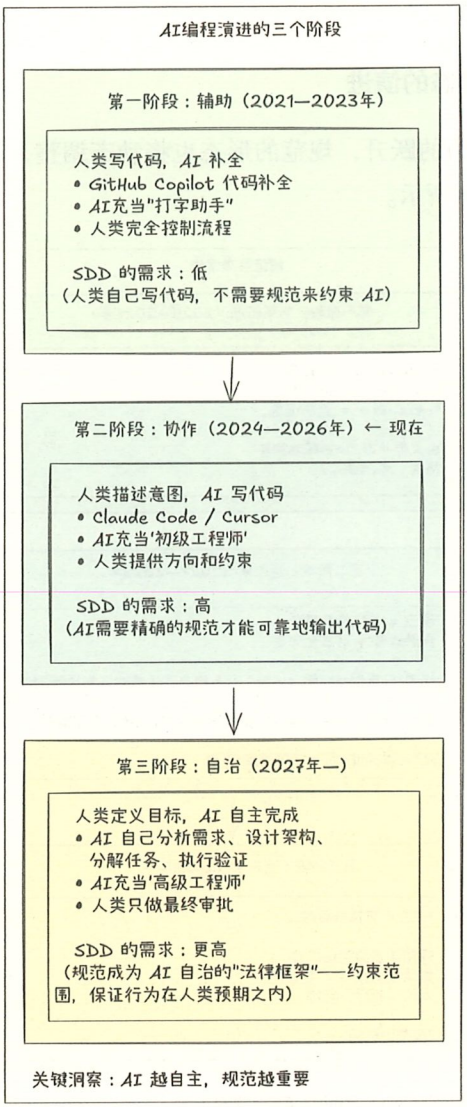


图 10-3　AI 编程演进的三个阶段

咖哥补充道：“规范，就是 AI 世界的交通规则。AI 的能力越强——

能编写更多的代码、做出更复杂的架构决策、自主执行更多步骤——一旦失控，造成的损失也就越惊人。规范，正是确保 AI 的强大能力被引导至正确方向的护栏。”

10.3.2 规范形态的演进

随着 AI 能力的跃升，规范的形态也将动态调整，以适应新的技术格局，如图 10-4 所示。

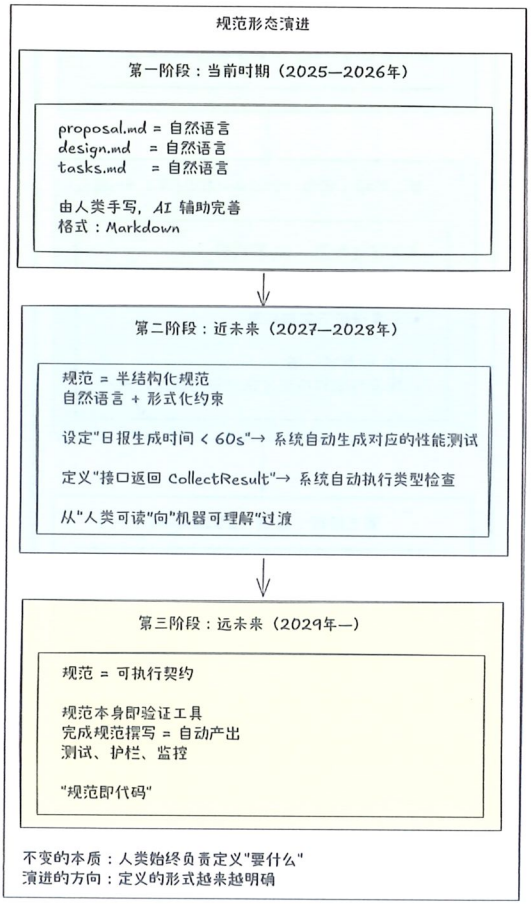


图 10-4 规范形态的演进：从自然语言到可执行契约

虽然具体编程工具会不断更迭消失，但核心方法论将保持稳定且经久不衰，更多内容可访问 https://kage-ai.com/sdd/#ch10-stability。

10.4　规范即代码的终极愿景

10.4.1　当规范足够精确时

若将 10.3.2 小节的演进逻辑推至极限，我们便会抵达这样一种终局：**当规范精确到足以完全描述系统行为时，代码的生成将实现完全自动化。**

这并非科幻构想。不妨回顾 SQL 的运作机制：当你写下 SELECT * FROM users WHERE age > 18 时，数据库引擎会自动生成执行计划、索引查找及磁盘读取代码。在此过程中，你编写的查询语句即“规范”，而数据库引擎则扮演了“AI”的角色。可以说，SQL 早已在关系数据领域率先实现了“规范即代码”的理念。

SDD 的远景，正是将这一模式推广至通用软件开发领域：

规范（精确描述意图）→ AI 编译器→代码（自动生成）

在此愿景中，proposal.md 不再是供人类阅读的文档，而是供“AI 编译器”读取的“源代码”。未来的代码将不再由人类逐行编写，也不再是 AI 随机“生成”的结果，而是像 C 代码被编译成二进制文件一样，由规范直接编译生成。

10.4.2　我们离终极愿景还有多远

客观而言，我们距离这一愿景仍有很长的路要走。

目前的 AI 技术（包括 Claude 在内）在解析自然语言规范时，依然面临歧义性挑战。以“日报中不包含代码 diff 的具体内容”这句话为例，AI 可能会将其解读为“完全不提及代码”，也可能理解为“提及代码变更但不展示具体差异”。这两种截然不同的理解，将直接导致完全不同的代码逻辑。

尽管存在挑战，但演进的方向已经明确。SDD 正是通往“规范即代码”的一条渐进式路径。本书每一章所传授的技能，都是朝着这一目标迈进的一小步。

- 撰写更精确的需求：旨在压缩 AI 对需求的解读歧义空间。
- 构建更完整的接口契约：旨在收窄 AI 在实现逻辑上的自由裁量范围。
- 制定更可执行的验收标准：旨在赋予 AI 依据标准进行自我验证的能力。

可以说，今天我们所实践的 SDD，正是对未来 AI 辅助编程方式的一次预演。

10.5 写给开发者的话：在 AI 时代，工程素养重于编码能力

10.5.1 能力矩阵的迁移

在 AI 辅助编程时代，开发者的核心能力正经历深刻的迁移，如图 10-5 所示。

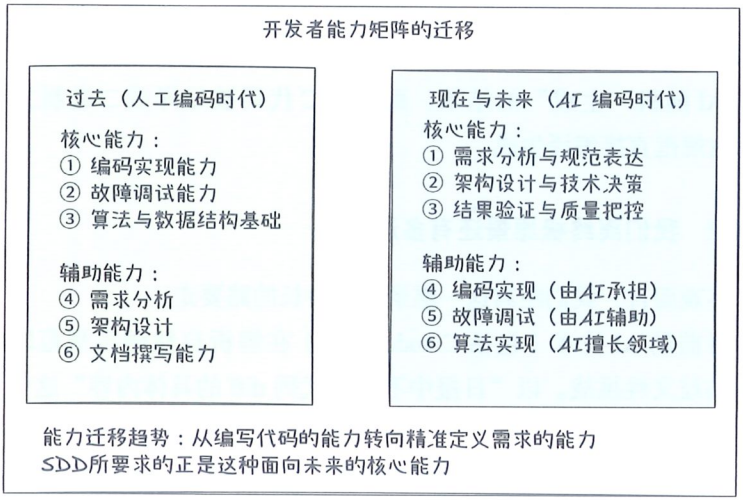


图 10-5 开发者能力矩阵的迁移：从“编写代码”到“精准表述”

这并不意味着编码能力不再重要——开发者依然需要具备阅读 AI 生成代码、评估其质量以及发现潜在问题的能力。然而，编码能力已从“核心生产力技能”转变为“基础素养”。这就好比今天的开发者虽然不再需要手写汇编语言，但理解计算机底层的工作原理依然具有重要价值。

10.5.2　工程素养的回归

SDD 揭示了一个深层趋势：在 AI 时代，软件工程中那些最基础的素养（如需求分析、架构设计、文档化、验证与验收），正在重新回归核心地位。

在过去二十年的“快速迭代”文化中，这些素养一度被边缘化。“先写代码再说”“文档不重要，代码即文档”“架构会自然演化”，这些观念在人工编码时代或许尚能成立，因为编写代码的人往往也是理解需求的人，“脑海中的隐性知识”尚可勉强替代书面规范的作用。

但 AI 没有“脑海中的隐性知识”：你未曾写下的内容，AI 无从知晓；你未曾明确表达的约束，AI 不会遵守；你未清晰界定的边界，AI 不会尊重。

因此，SDD 并非某种全新发明，而是软件工程基本功在 AI 时代的必然回归。

10.6　从 Vibe Coding 毕业

现在，小冰已经从最初那个“对着 AI 聊天写代码”的开发者，蜕变为一名能够利用三份规范驱动整条开发流水线的工程师。虽然“智能日报生成器”只是一个微型项目，但它所承载的方法论，足以驾驭任何规模的系统。

让我们最后重温一遍 SDD 的核心思想，如图 10-6 所示。

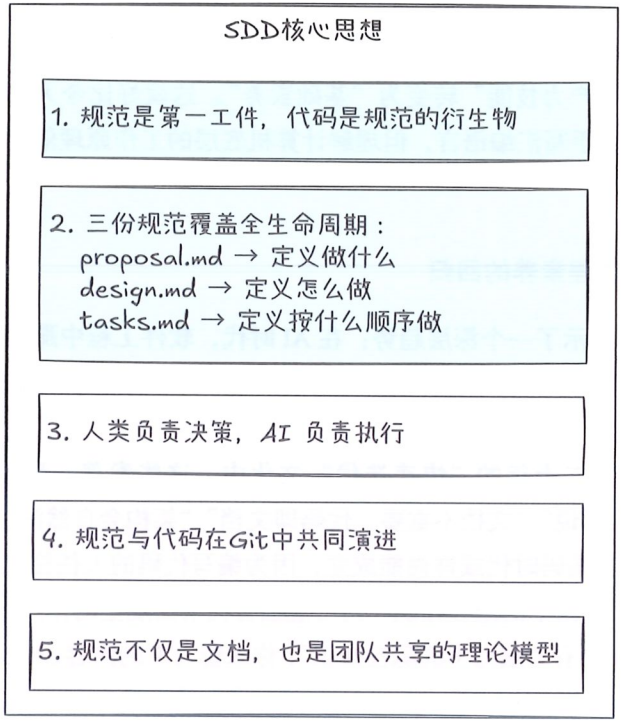
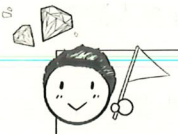


图 10-6　SDD 的 5 条核心思想

在这 5 条思想中，第一条构成了根本基石，它代表了一种认知范式的转变：从“以代码为中心”转向“以规范为中心”。一旦完成这一思维跃迁，其余 4 条思想便顺理成章地确立。

小雪做了精练的总结：“SDD 教会我一件事——**在 AI 时代，清晰的表达能力比编码能力更具价值**。讲清楚需求、讲清楚架构、讲清楚验收标准——这三个‘讲清楚’，便是整本书的精髓。”

咖哥补充道：“Vibe Coding 毕业并非终点，而是一个起点。你所告别的不只是一种编程方式，更是一种思维方式：从‘先试试看’转变为‘先想清楚’。无论 AI 技术如何进化，这种思维转变将永不过时。”



咖哥发言

每一次技术革命都会淘汰一部分技能与岗位，同时让另一部分变得更加珍贵。印刷术淘汰了抄写员，却让作者的地位愈发重要；计算器淘汰了算盘使用者，却让数学思维的价值愈发凸显。

AI 辅助编程工具正在淘汰“快速编码”这项技能，但它让“清晰定义问题”这项能力变得前所未有地珍贵。

SDD 所培养的并非某种工具的使用方法，而是一种思维方式。工具终会更新换代，今天是 Claude Code，明天便是其他产品。但“先想清楚再动手”这一原则，从未改变。真正优秀的方法论，经得起时间的检验。

希望这本书带给你的，不只是 SDD 的操作步骤，更是一种面对复杂问题时“先想后做、先慢后快”的工程态度。

Vibe Coding 的诱惑在于“现在就能看到结果”，而 SDD 的优势在于“以后不用重来”。

本章小结

本章作为全书的收官章节，从边界、融合、演进、愿景与素养 5 个维度对 SDD 进行了系统回顾，并以 5 条核心思想贯穿全书。

本章核心要点如下。

- **SDD 的甜蜜区**：最适合需求可被明确描述、系统复杂度中等偏上、需要多人协作或长期维护，且以 AI 辅助开发为主要生产方式的场景。这 4 项的契合度越高，SDD 的杠杆效应越显著；反之，在探索性原型或一次性脚本等场景中，则不需要强行套用 SDD。
- **方法论的融合**：SDD 与 TDD、BDD、DDD 并非替代关系，而是同心圆结构——业务架构、技术架构与实现细节三层共构，相互重叠、相互补充，而非彼此排斥。
- **AI 辅助编程的三个阶段**：从“辅助”到“协作”，再到“自治”。

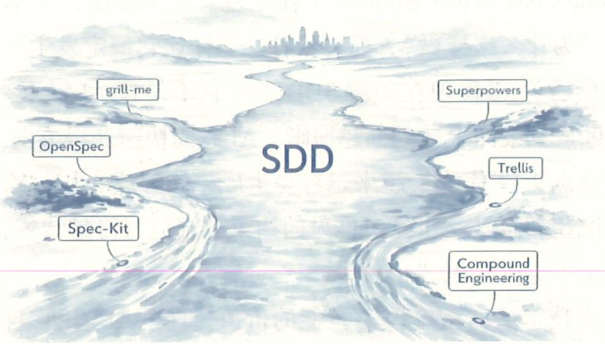
AI 的自主性越强，规范就越重要——这一趋势虽与许多人的直觉相悖，却已被现实反复验证。

- **规范形态的演进**：从自然语言到半结构化规范，再到机器可解析的“规范即代码”。当前编写的 proposal.md、design.md 与 tasks.md，正是未来领域特定语言（Domain-Specific Language，DSL）的雏形。
- **能力矩阵的迁移**：编码速度、调试技巧及框架熟练度等基础技能的价值正在相对降低；而问题定义、架构判断、规范表达及 AI 协作等能力的重要性日益凸显。这并非某种技能的淘汰，而是工程师价值重心的整体迁移。
- **SDD 的 5 条核心思想共同指向同一个底层主张**：在 AI 时代，清晰的规范表达能力比单纯的编码能力更具价值。

思考题

1. 本章主张“AI 越自主，规范越重要”。然而，一种反面观点认为：AI 若足够强大，理应能自主推断人类意图，不需要依赖显式规范。对此，你怎么看？未来是否可能出现“不需要规范的 AI 编程”？
2. “规范即代码”的愿景，要求规范必须从自然语言进化为可机器解析的格式。你能构想出一种既能让人类轻松读写，又能让 AI 精确解析和执行的格式吗？（提示：不妨参考 SQL、GraphQL、Protocol Buffers 这些“介于自然语言与代码之间”的格式。）
3. 请审视你当前的开发工作流。如果从明天开始引入 SDD，你认为最大的阻力将源自何处：是技术层面（不擅长撰写规范）、习惯层面（认为多此一举），还是团队层面（缺乏配合）？你又计划如何应对？

后记
百川入海，殊途同归



从第 1 章到第 10 章，我们走过了一段完整的旅程。

写到这里，我更愿意将工具名称后置。OpenSpec、Spec-Kit、Kiro、BMAD、Superpowers、Trellis、Compound Engineering 和 Harness Engineering 等说法，在正文中均已出现，它们还会继续演化。

但它们背后共享的方向是稳定的：将需求编写为可讨论的规范，将架构编写为可执行的约束，将任务编写为可验证的步骤，再将每次执行的反馈沉淀回规范与团队知识之中。

正因为如此，SDD 不是反敏捷，也不是回归厚重文档。它要求我们将“想清楚”的部分显性化、版本化、可审查化，再让 AI 承担那些边界清晰、反馈明确的执行工作。

百川归海。不同工具会使用不同的名称：Spec、Skill、Memory、