AI编程与智能体开发

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

2026年9月

主讲教师

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

配套教材

教材目录

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

教材特色

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

《AI编程与智能体开发》

林子雨 编著

人民邮电出版社 预计2026年11月上市销售

教材官网提供详细信息和样书申请

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

扫码访问教材官网
了解详情、获取资源、申请样书

第4章
基于提示词的AI编程

目录

Part 01

概述

4.1概述

什么是基于提示词的AI编程

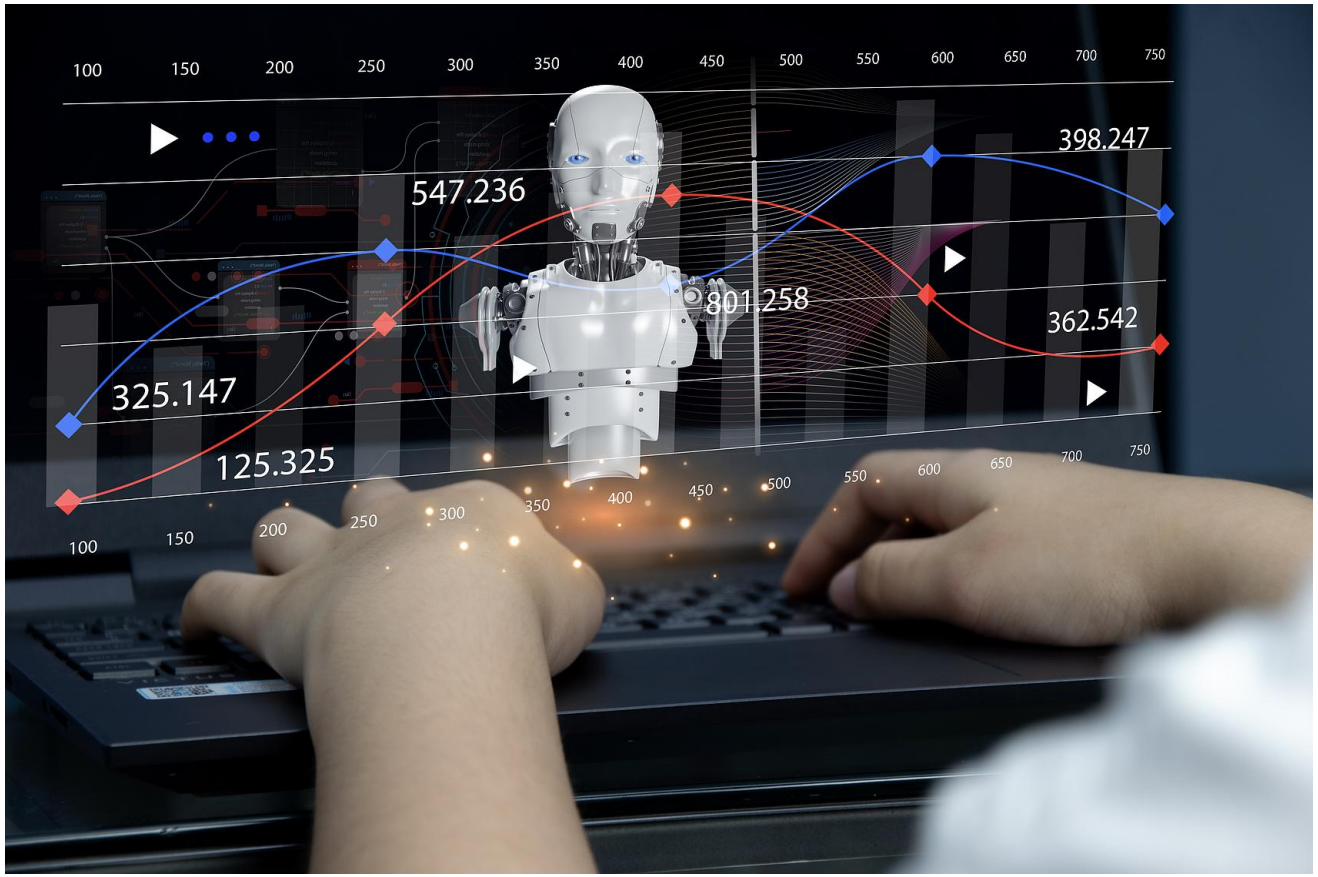
模糊指令引发的语义漂移

高质量的输出需要高质量的输入

AI编程的常见误区

4.1.1 什么是基于提示词的AI编程

基于提示词的 AI 编程，是依托大语言模型与生成式 AI 技术，开发者以自然语言、功能描述、业务需求等提示词作为输入，无需逐行手动编写代码，由 AI 自动生成、调试、优化和补全程序代码的新型编程模式。它打破了传统编程依赖专业语法、语法记忆和逐行编码的壁垒，把编程核心从 “手写代码” 转变为 “精准表达需求”。



4.1.1 什么是基于提示词的AI编程

1. 提示词存在两种形式：通俗自然语言、结构化功能要求；提示词可包含实现逻辑、编程语言、运行环境、功能细节、界面样式、异常处理等多类内容。
2. 开发者使用提示词只需清晰说明功能需求与执行规则，AI 即可识别语义逻辑，输出完整代码片段、函数模块、整套项目框架。
3. AI 编程附带多项延伸能力，包含代码解释、错误排查、性能优化、代码重构、跨语言转换。
4. 基于提示词的 AI 编程模式门槛更低：零基础初学者可依靠提示词快速开发；专业开发者可借助其替代重复编码，提升开发效率。
5. 该编程模式适用场景广泛，涵盖小程序开发、数据分析、接口编写、爬虫开发、算法实现等，重塑编程生产方式。
6. 提示词式 AI 编程本质属于人机协同开发：人负责定义需求、梳理逻辑、校验结果；AI 负责代码编写、规范语法、实现基础逻辑等机械工作。
7. 该模式转变编程核心驱动方式，编程由技能密集型转变为需求表达与逻辑设计驱动，是人工智能时代主流全新开发范式。

4.1.2模糊指令引发的语义漂移

基于提示词的 AI 编程普遍存在模糊指令引发的语义漂移问题。

高智能的 AI 无法仅凭模糊指令完成优质编程任务，如同优秀应届生无明确任务细则，无法产出符合预期的优化方案，易做出非必要、不符合需求的改动。

语义漂移的核心定义：AI 理解模糊需求时做出大量主观假设，导致输出结果逐步偏离用户真实编程意图。

行业经验结论（Tom Blomfield）：需将 AI 视作不懂具体业务的高智能新同事，向其布置编程任务时，需达到和对接人类工程师同等清晰的指令清晰度。

大语言模型 AI 编程能力突出，可以读懂海量代码、编写规范函数、优化数据库查询、排查潜在 Bug，常被比作能力出众的名校应届生。

AI 相较人类应届生存在明显短板：不会主动提问确认模糊需求，只会自行补全未明确的前提条件，过程中诸多自主推断极易出错，且问题往往最后才被发现。

语义漂移的特征：是 AI 编程最常见、易被忽视的问题根源，非一次性大幅偏差，而是随对话和模糊描述累积，逐步偏离用户预期。

4.1.3高质量的输出需要高质量的输入

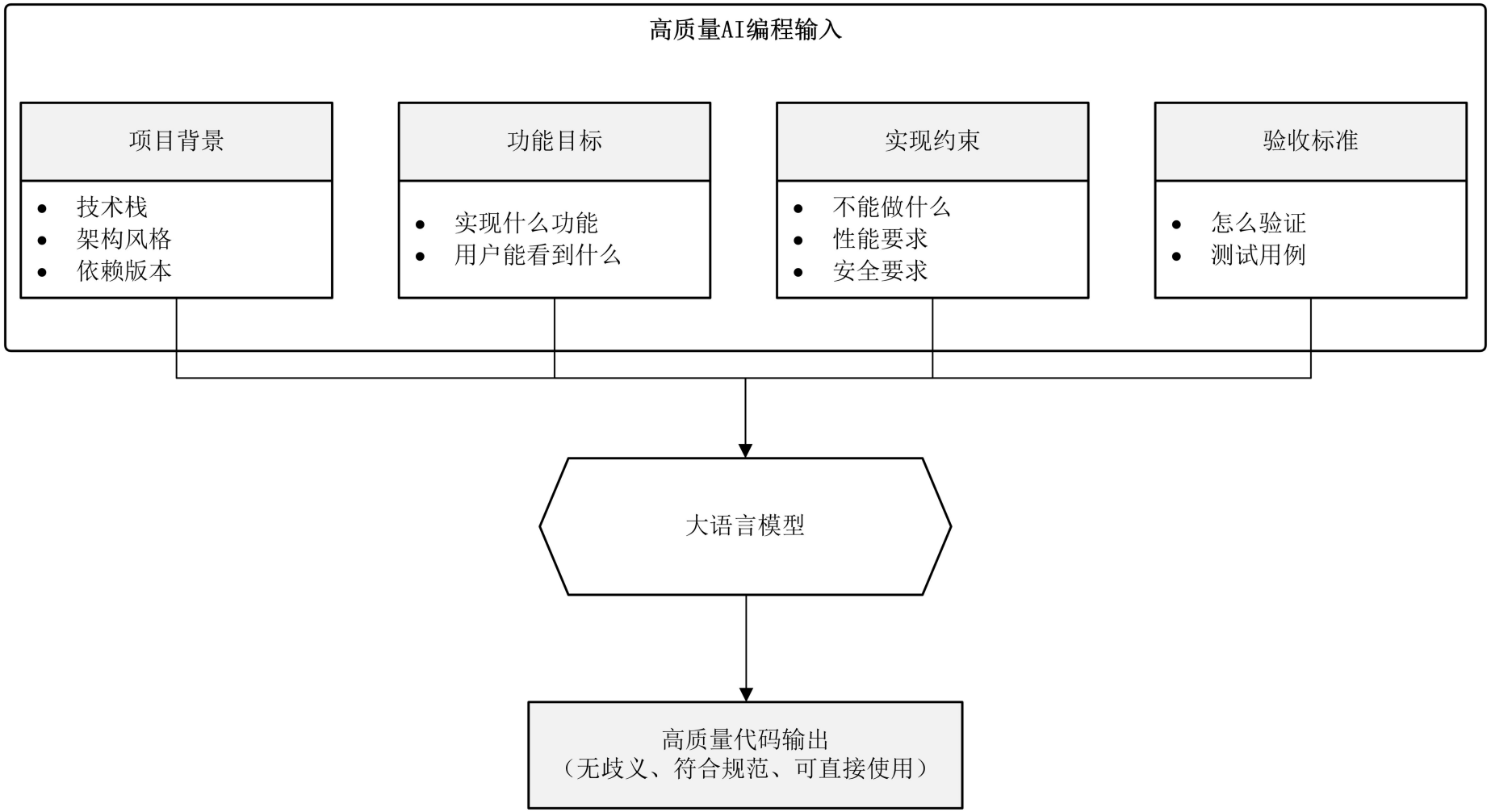
解决语义漂移问题的根本途径，不是更换一个更智能的AI，而是为AI提供更好的输入，也就是设计质量更好的提示词。

这里需要纠正一个常见的认知误区：很多人认为，随着AI能力的不断提升，提示词的质量会越来越不重要，未来的AI足够智能，能够从片言只语中理解你的完整意图。

这种想法有一定的道理，但它混淆了两个不同的问题，一是AI对语言的理解能力，二是AI对你的项目的了解程度。前者会随着模型进步而提升，但后者不会。无论模型有多智能，它都不可能自动了解你的项目架构、团队惯例、业务约束和历史决策，除非你主动告知。因此，“为AI提供足够好的输入”这一命题，在可预见的将来都不会过时。

4.1.3高质量的输出需要高质量的输入

一个高质量的AI编程输入，应当包含以下几个维度（如图所示）：项目背景、功能目标、实现约束、错误信息、验收标准。



4.1.3高质量的输出需要高质量的输入

这几个维度并不是独立存在的，它们共同构成了AI做出准确判断所需的完整信息，具体如下：

项目背景

当前系统的技术栈、架构风格、依赖库版本等基础信息。这是最容易被忽视的部分，也往往是影响最为显著的部分。AI在没有技术栈信息时，会默认使用自己训练数据中最常见的技术组合，与你的实际项目可能大相径庭。

功能目标

这次任务要实现什么，用户能看到什么变化。描述应当具体到可以被验证的程度，而不是方向性的表述。

实现约束

哪些文件不能修改、哪些接口不能破坏、有没有性能要求、有没有安全规定，这些约束是AI默认不会考虑的，必须明确说明。

错误信息

如果是修复Bug，完整的报错日志和堆栈追踪是必不可少的上下文。

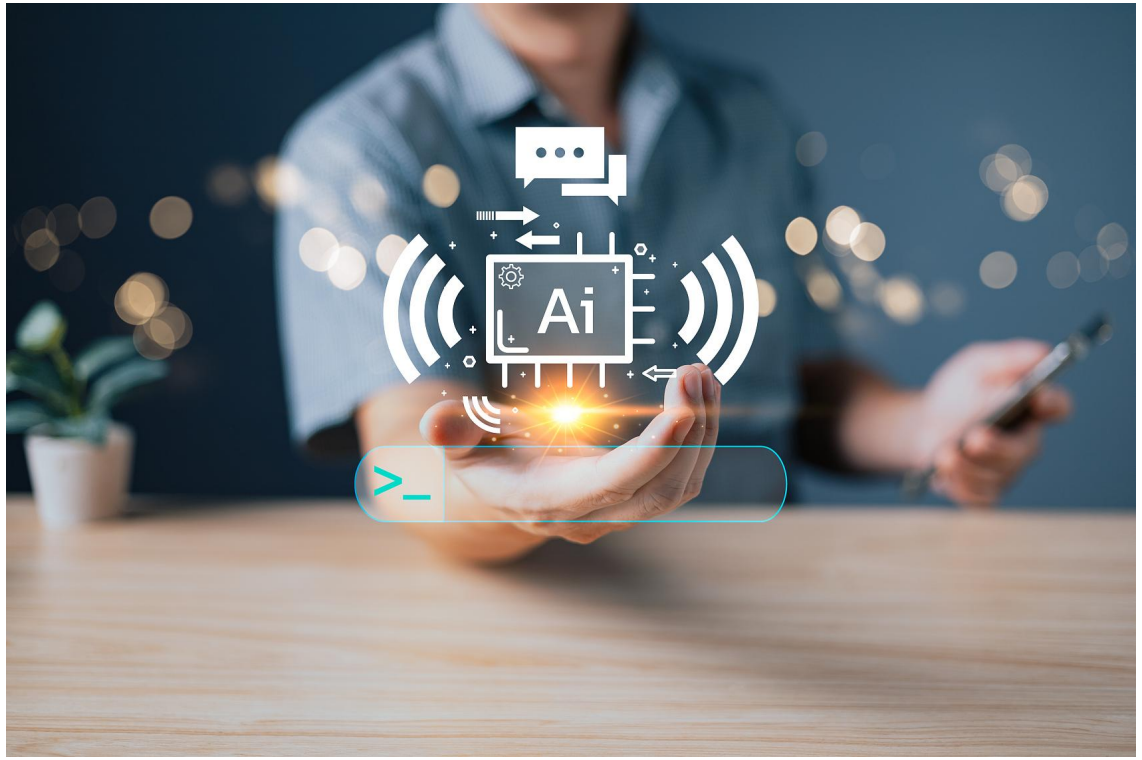
验收标准

什么样的结果算作完成，如何验证正确性。有了验收标准，AI可以在输出时进行自我检验。

4.1.4 AI编程的常见误区

■ **误区一：将AI当作"有感情的同事"来沟通**

人与人之间的沟通依赖大量的隐性知识和背景共识，所以，我们习惯于模糊、含蓄的表达。向同事说"这个函数有点慢"，对方能结合项目背景理解你的意思，给出合适的改进方案。但AI没有这种背景共识，它不知道你的项目和你的团队惯例，"有点慢"这个表述，在你的场景下意味着"响应时间超过200ms就不可接受"，还是"每次要等10秒"？这一点AI无法确定。



4.1.4 AI编程的常见误区

■ **误区一：将AI当作"有感情的同事"来沟通**

对AI的沟通原则是，像编写技术文档一样编写提示词，清晰、具体、无歧义，把所有前提条件说明清楚，不依赖对方"领悟"。这一原则体现在三个具体的沟通行为准则上（见下表）。

行为准则	反面示例（不要这样写）	正面示例（推荐写法）	改进要点
准则1：指令要精确	"帮我写一个React按钮"	"创建 `Button` 可复用组件，接受 `onClick`、`disabled`、`children` 作为props，并为 `disabled` 状态提供明确的视觉反馈（灰色+禁用鼠标样式）"	将"什么样的按钮"具体化到可被验证的程度
准则2：背景要补全	"优化这段代码"	"优化这个订单查询函数。当前实现存在N+1查询问题，使用 `prefetch_related` 解决，将100条订单的数据库查询次数降低到3次以内"	不假设AI了解"什么问题"和"什么标准"
准则3：反馈要直接	"这个实现好像有点慢"	"这个方案不可行，时间复杂度是$O(n^2)$，在n=10000时无法接受，需要重写，必须使用时间复杂度不超过$O(n \log n)$ 的算法"	用技术语言说明问题、边界和期望

4.1.4 AI编程的常见误区

■ **误区二：期待AI主动发现问题**

AI擅长回答问题，但不擅长主动识别你没有提到的问题。如果你只说"帮我写一个用户登录接口"，AI通常不会主动考虑SQL注入防护、密码哈希存储、并发登录限制、暴力破解防护等安全问题，除非你明确要求。

这不是AI的能力问题。如果你在提示词中明确写出"需要防止SQL注入、使用bcrypt存储密码、限制同一IP每分钟登录尝试次数不超过5次"，AI完全能够实现这些功能。问题不在于AI不会，而在于你没说。

4.1.4 AI编程的常见误区

■ 误区三：对AI的错误反馈过于委婉

当AI给出不满意的结果时，有人会写"这个实现好像不太对"或"能不能改得更好一些"。这种模糊的反馈对AI几乎没有帮助，AI不知道"不太对"是指功能错误、性能不足、代码风格不符合规范，还是其他什么问题。

正确的做法是直接、精确地指出问题所在。例如，可以给AI反馈"这个方案对每条记录都执行一次数据库查询，在有500条记录时会触发500次查询，不可接受。请使用批量查询将数据库调用次数降低到不超过3次，并说明你的优化思路"。

简而言之，直接，不是粗鲁；精确，而不是繁琐。用技术语言描述技术问题，是与AI协作最高效的方式。

4.1.4 AI编程的常见误区

■ 误区四：以为一次对话能搞定所有事

对于复杂的功能，一次性把所有需求堆给AI，往往得到质量参差不齐的输出，某些部分写得很好，某些部分漏洞百出，而且各个部分之间的风格和接口设计不一定一致。更好的做法是将任务拆分为多个有明确边界的子任务，逐步推进。每个子任务完成并验证后，再进入下一个。这与软件工程中的"渐进式交付"原则完全一致。

Anthropic Claude Code团队在最佳实践文档中特别强调了这一点："将大问题拆小，复杂任务拆成小块逐步推进，聚焦的请求比模糊大问题准确率高得多"。

4.1.4 AI编程的常见误区

■ 误区五：对AI的输出不加审查直接使用

开发者存在过度信任 AI 生成代码的误区：仅依据代码语法正确、逻辑通顺、可正常运行就判定代码合格，直接复制进项目，不经校验提交入库甚至部署生产环境，该行为是 AI 辅助编程中风险最高、危害最大的误区。

01

AI 生成代码为概率性输出，并非完美标准答案，即便基础运行无问题，也暗藏各类隐性问题：缺少边界条件与异常处理逻辑、存在并发安全漏洞、内存泄漏、算法复杂度过高、权限校验缺失、业务逻辑漏洞，简单测试难以发现，易引发程序崩溃、数据报错、系统瘫痪、安全及稳定性风险。

02

AI 生成代码常和现有项目技术规范不兼容：命名、注释、分层、封装、依赖版本等和团队规范、原有架构冲突，强行接入会破坏代码统一与可维护性，给后续迭代、协作、故障排查埋下隐患。

03

Part 02

提示词工程

4.2提示词工程

什么是提示词工程

提示词编写通用框架

提示词工程在AI编程中的应用

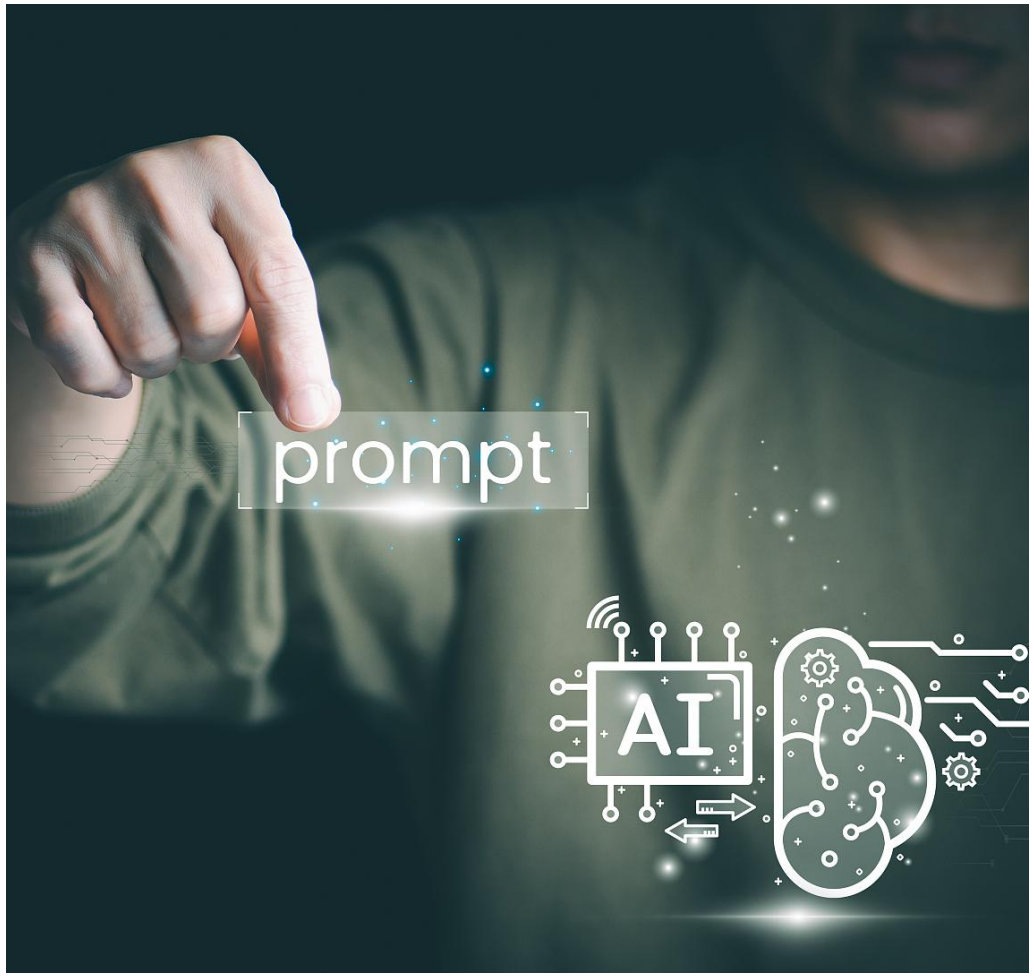
AI编程提示词通用写作技巧

4.2.1什么是提示词工程

提示词工程（Prompt Engineering）是指通过系统性地设计、优化输入文本（即"提示词"），引导大语言模型产生高质量、符合预期的输出的方法论与实践技巧。

提示词工程的本质，是在用户意图与模型能力之间建立一座桥梁。大语言模型的核心工作原理是：在给定上下文的条件下，预测下一个最可能出现的词元（Token）。因此，输入文本的内容、结构和细节，直接影响模型的推理路径和最终输出。给定完全相同的编程任务，一个精心设计的提示词与一个随手写下的提示词，所得到的结果可能有天壤之别。

提示词工程并非一门高深莫测的学科，它的核心思想与传统软件工程中的"需求分析"高度相似：在动手实现之前，把你真正想要的东西说清楚。只不过，提示词工程的沟通对象不是人类同事，而是一个对歧义极度敏感的大语言模型，它没有背景知识，无法主动追问，只能依赖你提供的文字来工作。



4.2.1什么是提示词工程

下表列出了提示词工程中常用的几种基本技术。

技术名称	核心思想	典型示例	适用场景
零样本提示（Zero-shot）	直接描述任务，不提供示例	"将下面这段代码的注释翻译为英文"	任务明确，AI已有足够训练数据
少样本提示（Few-shot）	提供2～5个输入或输出示例，引导AI理解期望格式	“对于这个判断密码强弱的程序，如果输入为全数字，如123456，则判定为弱。如果输入包含了字母、数字及字符串，且长度大于8个字符，如1234*AB!CD，则判断为强”	需要固定输出格式或特殊风格
思维链（Chain of Thought, CoT）提示	要求AI在给出答案之前，先逐步推理	"请先分析这段代码的问题所在，再给出修复方案"	复杂逻辑分析、Bug诊断
角色提示（Role Prompting）	为AI设定一个专业身份	"你是一位有10年经验的Python后端工程师"	需要特定领域专业知识
结构化提示	使用Markdown标题、代码块、编号列表等组织信息	用"## 项目背景"、"## 约束"等标题分隔不同信息	大多数编程任务

4.2.2提示词编写通用框架

当我们从更宏观的视角看待与AI的交互——无论是编程、写作、分析还是问答，都有一套通用的结构化框架值得学习，即Google PTCF框架。PTCF是Google在其官方提示词指南中提出的结构化提示方法，包含四个核心要素：角色（Persona）、任务（Task）、上下文（Context）、格式（Format）。

要素	核心问题	在AI编程中的对应内容
角色（Persona）	你希望AI以什么身份回答	"你是一位有10年Python后端经验的架构师"
任务（Task）	你要AI完成什么具体动作	"为用户服务实现一个分页查询接口"
上下文（Context）	AI需要哪些背景信息才能完成任务	“代码位于user-service目录下，应该使用user表，按照user_id进行分页，一页为十条记录”
格式（Format）	你希望AI的输出以什么结构呈现	"只输出代码，包含类型注解，附带pytest测试"

4.2.2提示词编写通用框架

■ 四要素详解

角色（Persona）是PTCF的基础元素，通过赋予AI一个具体的专业身份，来锚定其回答的视角、语气和知识基础。

设定角色不是无意义的仪式，当你告诉AI"你是一位有10年安全审计经验的后端工程师"时，它在分析代码时就会主动关注安全漏洞；当你告诉它"你是一位初学者教练"时，它的解释就会更偏向教学而非堆砌术语。

在AI编程场景中，角色设定可以精细到描述经验年限、擅长领域，乃至工作风格，下面是一个具体实例：

```
你是一位专注于Python Web后端开发的高级工程师，熟悉FastAPI + SQLAlchemy异步技术栈，代码风格追求简洁可读，在Code Review中以挑剔边界条件著称
```

4.2.2提示词编写通用框架

■ 四要素详解

任务（Task） 是PTCF的核心指令，要求用清晰、可操作的语言精确表达目标，而不是模糊的方向性描述。任务应当细化到"给一位新同事解释后，他能完全明白要做什么"的程度。复杂任务可以被拆解为步骤序列：

```
请完成以下三步：
1. 分析当前代码中的N+1查询问题，指出具体位置；
2. 使用selectinload重写查询逻辑，修复性能问题；
3. 说明修改前后的查询次数对比。
```

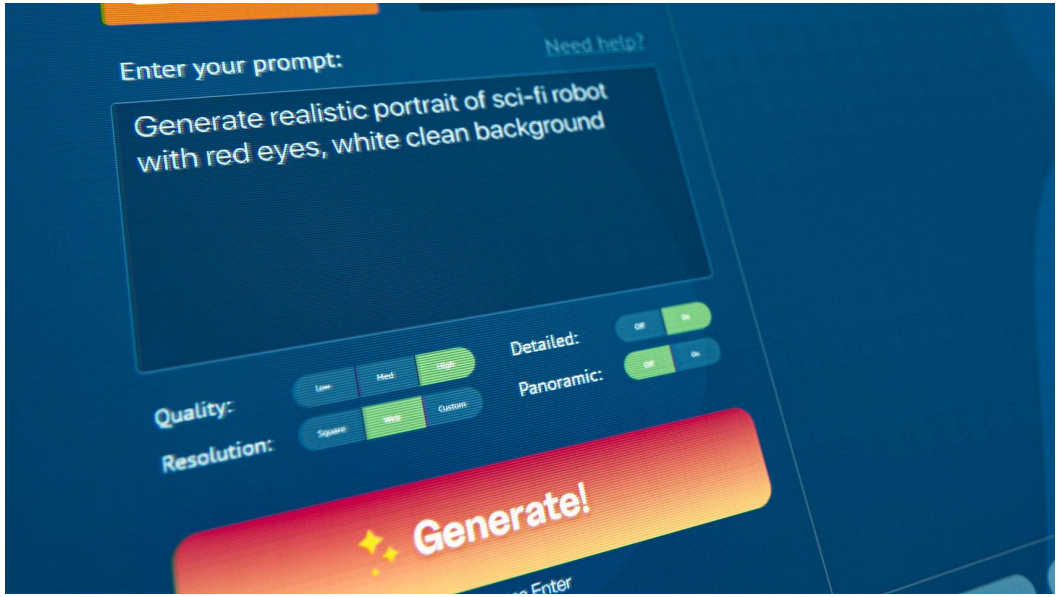
4.2.2提示词编写通用框架

■ 四要素详解

上下文（Context） 是AI做出准确判断的信息基础。在AI编程场景中，上下文尤为关键，因为代码的正确性取决于大量隐性的技术约束——版本兼容性、架构风格、团队规范、业务规则。没有充分上下文的提示词，AI只能依赖训练数据中的"平均经验"，结果往往和你的实际需求存在偏差。

上下文可以包含：代码片段、数据库表结构、接口文档、错误日志、性能数据，以及任何"AI不知道但影响结果"的信息。

有一个简单的判断原则：把上下文剥掉后，AI需要"猜"什么？猜的越多，上下文就越不够。



4.2.2提示词编写通用框架

■ 四要素详解

格式（Format） 负责规定输出的结构和风格，减少输出的随机性。在AI编程场景中，常见的格式要求包括：

1. "只输出代码，不要解释"（减少无用文字，便于直接使用）。
2. "先给思路，再给代码"（用于需要先对齐方向的复杂任务）。
3. "输出JSON，字段包含：file_path、change_summary、risk_level"（用于代码审查自动化）。
4. "以Markdown表格对比修改前后的差异"（用于重构场景）。

4.2.2提示词编写通用框架

■ PTCF完整示例

这里将四要素组合，形成一个完整的PTCF提示词：

```
## 角色（Persona）
你是一位专注于Python后端安全的高级工程师，有丰富的FastAPI项目经验，
熟悉OWASP Top 10安全漏洞，在Code Review中以安全审查著称。

## 任务（Task）
对下面这个用户登录接口做安全审查，列出所有安全问题，并给出修复建议。

## 上下文（Context）
项目使用FastAPI 0.115 + SQLAlchemy 2.0 + PostgreSQL 15。
当前接口代码如下：
```

4.2.2提示词编写通用框架

■ PTCF完整示例

```
@router.post("/login")
async def login(username: str, password: str, db: Session = Depends(get_db)):
user = db.execute(
text(f"SELECT * FROM users WHERE username = '{username}'")
).fetchone()
    if user and user.password == password:
        return {"token": create_token(user.id)}
    raise HTTPException(status_code=401, detail="Invalid credentials")

## 格式（Format）
以Markdown表格输出：
| 问题编号 | 漏洞类型 | 严重程度（高/中/低） | 具体问题描述 | 修复建议 |
在表格之后，给出修复后的完整代码。
```

4.2.3提示词工程在AI编程中的应用

提示词工程可以用于AI编程，也可以用于其他领域（比如撰写文章、绘制图片等）。AI编程中的提示词工程，与通用场景下的提示词工程既有共性，也有其独特性。

共性在于：清晰、具体、无歧义的表达始终是第一原则。无论是让AI写诗还是让AI写代码，说清楚你想要什么，永远是第一步。

独特性在于以下几个方面：

技术上下文的重要性远高于通用场景。 AI编程中的提示词通常需要包含大量技术信息——代码片段、错误日志、数据结构定义、API文档片段、数据库表结构等。这些技术上下文在普通问答场景中几乎不会出现，但在编程场景中，它们是AI做出准确判断的必要依据。

输出的正确性有客观标准。 与让AI写文章不同，AI生成的代码有明确的对错之分：它能不能运行、能不能通过测试、性能是否达标、是否符合编码规范。这意味着你可以在提示词中给出具体的验收标准，让AI在输出时进行自我检验。

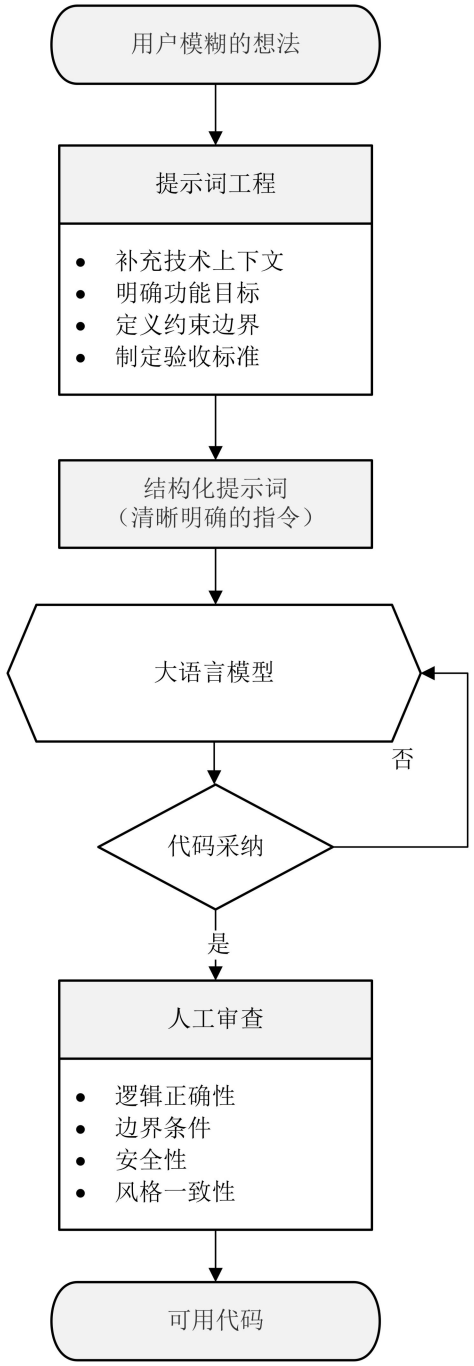
迭代效率比一次完美更重要。 在实际编程中，你通常不需要一次得到完美的答案，而是需要快速得到一个可以迭代改进的起点。这意味着提示词的设计也应当面向迭代：每次只解决一个明确的小问题，而不是把所有需求一次性堆进一个庞大的提示词中。

4.2.3提示词工程在AI编程中的应用

下图展示了AI编程中提示词工程的完整作用链路。

这一链路揭示了一个重要事实：提示词工程不能取代代码审查。

提示词的质量决定了候选代码的质量区间，但最终落地的代码仍然需要经过人工审查，确认它满足所有显式和隐式的要求。



4.2.4 AI编程提示词通用写作框架

下面介绍一套在实际开发中被广泛验证的AI编程提示词通用写作框架。这套框架包含5个核心要素（即项目背景、需求描述、修改范围、约束边界、验收标准）和1个提示词模板，每个要素都对应着AI在生成代码时需要做出的一类判断。只有当所有要素都被清晰说明时，AI才能真正地"理解"任务，而不是"猜测"任务。

需要说明的是，本节介绍的框架并不是对前文 Google PTCF 框架的替代，也不是另一套相互竞争的提示词标准。PTCF 更像是通用提示词的基础骨架，适用于编程、写作、分析、问答等多种场景，强调从角色、任务、上下文和格式四个维度组织表达。

而本节的框架，是在 AI 编程这一具体场景下对 PTCF 的细化和工程化展开。其中，“项目背景”主要对应 PTCF 中的上下文，“需求描述”对应任务，“修改范围”和“约束边界”进一步细化编程任务中的上下文与边界条件，“验收标准”用于明确代码是否完成且正确，“提示词模板”则帮助把这些要素组织成可复用的输出格式。

因此，读者可以把 PTCF 理解为通用原则，把本节框架理解为面向 AI 编程实践的操作清单，在一般提示词应用场景中（比如写作、分析和问答等）使用 PTCF 框架即可，在编写 AI 编程提示词时，则优先使用本节的框架。

4.2.4 AI编程提示词通用写作框架

■ 项目背景

在提示词的开头，告诉AI当前的项目情况。这是最容易被初学者忽视，却往往影响最为显著的部分。背景信息应当包含：

技术栈

编程语言、主框架及其版本（如Python 3.12、FastAPI 0.115、Pydantic v2）。

项目概况

这是什么系统，当前模块承担什么职责。

相关代码

与任务直接相关的函数、类或数据结构定义——这是AI保持代码风格一致性的依据。

4.2.4 AI编程提示词通用写作框架

■ 项目背景

这里给出一个具体实例：

```
## 项目背景
当前项目是一个电商平台的后端API服务，使用Python 3.12 + FastAPI 0.115 +
SQLAlchemy 2.0（异步模式）构建，数据库为 PostgreSQL 15，缓存使用Redis 7。

项目采用三层架构：router（路由层）→ service（业务层）→ repository（数据访问层）。
所有业务异常继承自AppException，由全局异常处理器统一返回JSON格式的错误响应。

相关数据模型如下：
[粘贴 Order 和 OrderItem 的 SQLAlchemy 模型定义]
```

4.2.4 AI编程提示词通用写作框架

■ **需求描述**

清晰、具体地描述你想要AI做什么。避免使用"优化一下"、"写一个接口"等宽泛表述，而应使用可以被客观验证的技术描述。

有一个简单的判断标准：如果你的需求描述可以被多种不同的实现方式满足，那么，它就不够具体。下表给出了在撰写需求描述时模糊写法与清晰写法的对比。

模糊写法	清晰写法
“帮我优化这个查询函数”	“该函数存在N+1查询问题，请使用SQLAlchemy的 `selectinload` 重写，使100个订单的查询次数降低到3次以内”
“帮我写一个缓存”	“为 `get_product` 函数添加Redis缓存，缓存key格式为 `product:{id}`，TTL为300秒，使用 `json.dumps` 序列化”
“加一个日志”	“在 `create_order` 函数的入口处添加INFO级别的结构化日志，记录 `user_id`、`product_ids` 和 `total_amount` 字段”

4.2.4 AI编程提示词通用写作框架

■ 修改范围

明确告知AI可以修改哪些内容，哪些是不允许改动的，这是防止AI过度扩展修改范围的关键要素。

在没有明确范围的情况下，AI可能为了实现一个功能而"顺手"修改了不该动的接口签名、调整了数据库模型，或者引入了你不需要的新依赖，这些无声无息的改动往往是重大事故的来源。

下面是一个关于修改范围的具体实例：

```
## 修改范围
- 可以修改：`order_service.py` 中的 `get_orders_with_items` 函数；
- 不得修改：该函数的输入参数和返回值类型；
- 不得修改：`models.py` 中的任何数据库模型定义；
- 不得引入新的第三方依赖（requirements.txt 不能改变）。
```

4.2.4 AI编程提示词通用写作框架

■ 约束边界

列出技术或业务上的硬性约束，这些约束是AI必须遵守的底线。约束与范围的区别在于：范围限定"做什么、不做什么"，约束限定"怎么做才合规"。

常见的约束类型包括：

安全约束

禁止字符串拼接SQL，必须使用参数化查询；禁止明文存储密码。

性能约束

接口响应时间P99不超过200ms；不得引入额外的同步数据库查询。

编码规范

遵循PEP 8，使用类型注解，函数长度不超过50行。

兼容性约束

不得破坏已有的REST API接口契约。

业务规则

金额计算必须使用Decimal类型，禁止使用浮点数。

4.2.4 AI编程提示词通用写作框架

■ 验收标准

验收标准就是告诉AI什么样的输出才算达到要求。验收标准是提示词中最容易被忽视，但对最终代码质量影响最大的部分之一。**好的验收标准有两个特点：可以被自动化验证（转化为单元测试），以及覆盖主要的异常场景（而不只是正常路径）**。

下面是一个验收标准的具体实例：

```
## 验收标准
- [ ] 功能正确：传入 user_id，返回该用户的所有订单及其关联订单项；
- [ ] 性能要求：使用 100 个订单的测试数据，数据库查询次数不超过 3 次；
- [ ] 空结果处理：当 user_id 对应的用户没有订单时，返回空列表，而非抛出异常；
- [ ] 用户不存在处理：当 user_id 不存在时，抛出 UserNotFoundError；
- [ ] 附带单元测试：至少覆盖正常情况、空结果和用户不存在三种场景；
- [ ] 代码包含类型注解，通过 mypy --strict 检查。
```

4.2.4 AI编程提示词通用写作框架

■ 提示词模板

将上述五个要素组合起来，就可以形成一个可复用的AI编程提示词模板，下面是一个典型的提示词模板：

```
## 项目背景
[技术栈及版本、项目架构概述、相关代码片段]

## 需求描述
[具体、可客观验证的任务描述]

## 修改范围
- 可以修改：[列举]
- 不可修改：[列举]

## 约束边界
- [安全约束]
- [性能约束]
- [编码规范]
- [其他约束]

## 验收标准
- [ ] [标准 1：正常情况]
- [ ] [标准 2：异常情况]
- [ ] [标准 3：性能要求]
- [ ] [附带测试用例]
```

Part 03

AI编程实践

4.3 AI编程实践

提示词正确使用案例

提示词使用中的常见问题

与AI协作的实践经验

4.3.1 提示词正确使用案例

■ 开发角色：猜数字游戏

现在要编写一个交互式猜数字游戏。作为开发者，不仅要保证程序的核心功能正常运行，更要充分考虑各种异常情况，确保代码的健壮性。例如，用户可能误输入非数字字符，程序应该优雅地处理这类异常而不是直接崩溃，这是专业开发者的基本素养。

在TRAE中，在左侧的“资源管理器”界面中，选中一个文件夹（比如“AI-Coding”），在智能体交互界面中，选择Agent模式，然后，在对话框中输入如下提示词并发送：

```
## 角色
你是一位注重代码健壮性的 Python 开发者。

## 任务
编写一个猜数字游戏脚本。

## 约束
- 随机生成 1-100 的整数；
- 使用 `while` 循环持续猜测；
- **必须**处理非数字输入，禁止程序崩溃；
- 猜对后必须打印 `Bingo` 并退出。

## 参考
- 使用 `input()` 获取用户输入；
- 使用 `random.randint(1, 100)` 生成目标数字。

## 验收标准
- [ ] 输入非数字（如 'abc'）时，提示"无效"并继续循环；
- [ ] 输入 50（假设答案是 50）时，打印 `Bingo`；
- [ ] 输入 60 时，提示 `Too big`。
```

4.3.1 提示词正确使用案例

■ 安全角色：密码强度检测

现在要编写一个密码强度检测脚本。密码安全是系统安全的第一道防线，弱密码往往成为黑客攻击的突破口。安全工程师需要制定明确的密码策略，通过自动化工具快速检测密码强度，帮助用户设置符合安全要求的密码，降低账号被盗风险。

在AI编程中，可以在TRAE的智能体交互界面的对话框中输入如下提示词：

```
## 角色
你是一位安全审计工程师。

## 任务
编写一个 Python 密码强度检测脚本。

## 约束
- 长度必须在 8 到 16 位之间；
- **必须**同时包含数字和字母；
- **必须**包含特殊字符 `!` 或 `@`；
- 满足所有条件打印 `Strong`，否则打印 `Weak`。
```

```
## 参考
- 使用 `str.isdigit()` 和 `str.isalpha()` 检测字符类型。

## 验收标准
- [ ] 输入 `abc123!x` -> 输出 `Strong`；
- [ ] 输入 `abcdefg` -> 输出 `Weak`（缺数字和特殊符）；
- [ ] 输入 `12345678` -> 输出 `Weak`（缺字母和特殊符）。
```

4.3.1 提示词正确使用案例

■ 运维角色：端口存活检测

现在要编写一个端口存活检测脚本。在运维工作中，及时发现服务端口异常是保障系统稳定运行的关键。运维工程师需要编写监控脚本，定期检测关键服务端口的存活状态，并在检测失败时进行重试，避免因网络抖动等临时性问题导致的误报，确保告警的准确性。

在AI编程中，可以使用如下提示词：

```
## 角色
你是一位初级运维开发工程师。

## 任务
编写检测 `localhost:80` 端口存活的脚本。

## 约束
- 超时设置为 1 秒；
- 如果连接失败，**必须**重试 2 次；
- 最终成功打印 `UP`，失败打印 `DOWN`；
- 使用 `socket` 库。
```

```
## 参考
- 使用 `socket.settimeout(1)` 设置超时；
- 使用 `socket.connect_ex()` 检测端口。

## 验收标准
- [ ] 端口存活时，输出 `UP`；
- [ ] 端口关闭时，经历 3 次尝试后输出 `DOWN`；
- [ ] 脚本不能在第一次失败后直接退出。
```

4.3.2 提示词使用中的常见问题

■ **问题一：需求不明确**

现在要为一个用户管理系统编写用户注册 API 接口。提示词的反面示例（模糊指令）如下：

```
帮我写一个用户注册的 API 接口。
```

这个提示词仅包含15个字，信息极度匮乏。AI接收到该指令后，需要做出大量假设以填补信息空白，具体如下：

技术栈选择

需要确定使用Flask、FastAPI还是Django等框架。

字段设计

需明确注册功能包含的字段，如用户名、邮箱或手机号等。

密码存储

需决定密码的加密方式，如明文、MD5或bcrypt等。

验证机制

需确定是否需要邮箱验证环节。

接口格式

需定义接口的请求与响应数据格式。

异常处理

需明确邮箱已注册时的处理逻辑。

AI不会因信息不足而拒绝响应，它会自动生成假设并填补这些空白，最终输出一段看似完整但可能与实际需求存在较大偏差的代码。

4.3.2 提示词使用中的常见问题

■ **问题一：需求不明确**

下面是一个正面示例（采用4.2.4节的提示词模板），展示了完整技术上下文对AI编程结果的决定性影响——从"15字模糊指令"到包含项目背景、约束条件和验收标准的结构化提示词，AI输出质量将发生质的飞跃，具体提示词如下：

```
## 角色
你是一位有 5 年经验的 Python 后端工程师，熟悉 FastAPI + SQLAlchemy 2.0。

## 任务
实现 `POST /api/v1/users/register` 用户注册接口。

## 约束
- 密码必须哈希存储（使用bcrypt）；
- 不得修改User数据模型；
- 所有新增代码包含完整类型注解。
```

```
## 参考
参照项目中现有的接口规范：
- 路径格式：`/api/v1/{resource}`
- 成功响应：HTTP 201 + `{"user_id": int, "username": str}`
- 错误响应：HTTP 4xx + `{"detail": "ERROR_CODE"}`

## 验收标准
- [ ] 注册成功 → HTTP 201
- [ ] 邮箱重复 → HTTP 400
- [ ] 密码哈希验证
```

4.3.2 提示词使用中的常见问题

■ **问题一：需求不明确**

下表明确给出了提示词在本案例中发挥的作用。这个提示词的“验收标准”部分，实际上就是一份测试用例清单。AI在生成代码时会潜移默化地朝着"通过这些测试"的方向写，这比单纯说"请保证质量"有效得多。

要素	本案例中的作用	如果缺失会怎样
项目背景	锁定技术栈和架构风格，AI会主动遵循项目规范	AI可能生成无类型注解的随意代码
约束边界	明确安全红线和架构约束	AI可能引入新依赖，或使用不安全算法
验收标准	3条可验证的验收标准，AI生成代码时有自我检验的依据	只能靠人工逐一验证

4.3.2 提示词使用中的常见问题

- **问题二：上下文缺失**

  这里假设在生产环境中出现了一个订单查询接口慢查询问题，需要排查修复。提示词的反面示例（缺少上下文）如下：

```
我的订单查询接口很慢，帮我优化一下。
```

对于这个提示词，AI完全不知道你的数据库结构、当前的查询语句、性能瓶颈在哪里。它只能给出泛化的优化建议，如考虑添加索引、使用连接池、考虑缓存层等，这些建议听起来有道理，但在你的具体场景下可能完全不适用，甚至有反效果。

4.3.2 提示词使用中的常见问题

- **问题二：上下文缺失**

  下面给出正面示例（简化格式，突出关键要素），展示了完整技术上下文的重要性。当问题已被诊断清楚时，关键在于将问题代码、量化基线和架构约束准确传递给AI，使其直接给出实现方案而非重新诊断。这里的“量化基线”是指，把业务、系统、质量、效率等指标，用可测量的数值固定下来，作为基准参照标准，用来做对比、评估、管控和改进的一条基准线。需要注意的是，本案例假设项目上下文已在项目说明书中建立，因此，**简化了提示词格式，重点展示"问题诊断 + 约束 + 验收标准"三要素**，具体提示词如下：

```
## 任务
修复 `src/repositories/order_repo.py` 中 `get_orders_with_items` 函数的 N+1 查询问题。

## 约束
- 函数签名（`user_id: int` -> `list[Order]`）不得改变；
- 仅使用 SQLAlchemy 内置的 `selectinload`，不引入新依赖；
- 保持异步模式。
```

```
## 问题代码（已诊断）

def get_orders_with_items(user_id: int) -> list[Order]:
orders = session.execute(select(Order).where(Order.user_id ==
user_id)).scalars().all()
for order in orders:
_ = order.items  # ← N+1问题：每个订单触发一次额外查询
return orders

## 验收标准
- **性能目标**：200 个订单的查询次数 ≤ 3 次；
- **功能不变**：返回内容与修改前完全一致；
- **边界**：`user_id=99999`（无订单）时返回 `[]`，不抛异常。
```

4.3.2 提示词使用中的常见问题

■ 问题二：上下文缺失

本案例与案例一的最大区别是：诊断结论已经存在。参考代码让AI精确知道"哪里有错"，量化基线让AI理解"要优化到什么程度"，约束锁定了架构底线，验收标准将可执行的测试用例直接嵌入提示词。

注意提示词的结构：任务定义 → 约束 → 问题代码 → 验收标准。这种结构将明确的任务声明与量化验收标准相结合，使AI在整个代码生成过程中都受到清晰目标的约束。

这正是AI最能发挥价值的工作模式：问题已经被诊断清楚，需要AI提供实现方案。AI在"有明确问题定义、需要技术实现"的场景下表现最为稳定，在"问题本身尚不明确、需要AI自行诊断"的场景下，则容易给出不准确的答案。

4.3.2 提示词使用中的常见问题

■ 问题三：异常处理不完善

现在要实现一个批量发送营销短信的后台定时任务。提示词的反面示例（忽略边界条件）如下：

帮我写一个批量发送短信的功能，从数据库读取待发送的短信列表，

调用短信服务商的 API 逐条发送，发送完成后更新数据库状态。

4.3.2 提示词使用中的常见问题

■ 问题三：异常处理不完善

AI根据这个提示词生成的代码，通常只覆盖"一切正常"的常规路径：读取数据、循环调用API、更新状态。代码看起来逻辑清晰，在测试环境中也能正常工作。然而，这段代码在生产环境中隐藏着多个严重风险，具体如下：

重复发送

如果任务被意外重启（服务器重启、进程被“杀死”（kill）），已经发送成功但状态更新失败的短信会被再次发送；

并发冲突

当多个worker同时运行时（这在生产环境中很常见），同一批短信可能被多个worker同时处理；

无重试机制

网络抖动导致API调用失败，短信直接被标记为失败，没有重试机会；

无限流机制

短信服务商有QPS限制，频繁调用可能触发限流，导致大量失败；

内存溢出风险

如果待发送短信有10万条，一次性加载到内存会导致服务崩溃。

4.3.2 提示词使用中的常见问题

■ 问题三：异常处理不完善

下面是一个正面示例（简化格式，突出边界条件），展示了约束和边界条件的重要性。AI默认只生成常规路径代码，所有并发安全、幂等性、重试和限流等工程要求，都需要在提示词中主动指明。需要注意的是，本案例展示的是"需求已完全拆解后的最终提示词"。实际工作中，并发安全、幂等性、重试、限流四个功能，应拆分成多个小任务逐步实现，而非一次性交给AI。具体提示词如下：

```
## 任务
重写 `send_batch_sms` Celery 任务，实现并发安全的批量短信发送，
支持幂等性保证、重试机制和限流控制。

## 约束
- 不得引入新的第三方依赖（`tenacity` 和 `requests` 已在
requirements.txt）；
- 幂等性：发送前必须检查 `msg.status == "SENT"`，避免重复发送。
```

```
## 问题代码（仅处理常规路径，存在多处隐患）
@celery.task
def send_batch_sms():
msgs =
Session().query(SmsMessage).filter_by(status="PENDING").limit(100).
all()
for msg in msgs:
requests.post("https://sms-api.example.com/send", json={...})
msg.status = "SENT"
Session().commit()

## 验收标准（以下每条对应一个生产事故风险）
- [ ] **C1：并发安全**——`with_for_update(skip_locked=True)`
- [ ] **C2：幂等性**——发送前检查 `status == "SENT"`
- [ ] **C3：重试**——HTTP 5xx 重试 3 次，HTTP 4xx 直接标记 FAILED
- [ ] **C4：限流**——循环中加入 `time.sleep(0.02)`，确保 ≤ 50 QPS
- [ ] **C5：最终失败**——3 次重试后状态 = `FAILED`，记录
`error_message`
```

4.3.2 提示词使用中的常见问题

■ **问题三：异常处理不完善**

这个案例揭示了AI编程中一个极其重要的规律：AI默认只处理正常路径（常规路径），所有的边界条件、异常情况和非功能性要求（并发安全、幂等性、重试、限流），都需要人类主动在提示词中指明。

这并非AI的能力局限。如果你明确要求它处理这些情况，它完全能够给出正确的实现。这源于AI的"工作方式"——它会生成满足你所描述需求的最短路径代码，不会自动为你考虑你没有说出来的"理应如此"的工程要求。

下表展示了缺少每种边界条件时可能引发的具体生产事故。

缺少的边界条件	AI会生成什么样的代码	引发什么生产事故
缺少并发安全	多个worker会重复发送同一条短信	**重复短信投诉**
缺少幂等性	网络超时后重试会导致同一条短信发送多次	**重复扣费投诉**
缺少重试机制	偶发网络抖动会导致短信丢失	**短信漏发**
缺少限流控制	批量100条同时发出，触发API限流（HTTP 429）	**大量失败**
缺少错误处理	未处理的异常导致Celery worker崩溃	**队列停止消费**

4.3.2 提示词使用中的常见问题

■ 问题三：异常处理不完善

AI编程中最常见的生产事故，不是"代码有语法错误"，而是"边界条件没覆盖"。约束和验收标准部分，就是你的"边界条件检查清单"。养成在写提示词之前先列出边界条件的习惯，能从源头上避免大量生产事故。

在撰写提示词时，切忌一次性丢给AI一个"CRM系统"级别的大任务，必须拆分成单一模块逐一实现。本案例如果将并发安全、幂等性、重试、限流四个问题混在一起让AI一次性解决，代码极大概率会结构混乱。

正确的拆分方法是：第一轮解决并发安全，验证通过后再进入第二轮；第二轮加入重试机制……（需要注意的是，本书为了展示的完整性，一次性给出了全量需求，适合作为"需求已完全拆解后的最终提示词"）。

4.3.2 提示词使用中的常见问题

■ 问题四：需求描述不完整

现在要实现一个完整的 JWT 认证中间件，支持 Token （令牌）验证、刷新和黑名单功能。提示词的反面示例（逐步迭代）如下：

第一轮：帮我写一个 JWT 认证中间件。
[AI 给出基础实现，只做了 Token 验证]
第二轮：还需要处理 Token 过期的情况。
[AI 修改代码，但过期错误的返回格式与项目规范不一致]
第三轮：过期的错误响应格式不对，要返回 {"code": 401, "message": "token_expired"}。
[AI 再次修改，但漏掉了无效 Token 的情况]
第四轮：无效 Token 也要处理，返回 {"code": 401, "message": "invalid_token"}。
[AI 修改后，又提出需要刷新 Token 功能]
第五轮：还要支持刷新 Token。
第六轮：刷新 Token 要检查旧 Token 是否在黑名单里……

4.3.2 提示词使用中的常见问题

■ 问题四：需求描述不完整

这种逐步迭代的对话方式存在以下几个系统性问题：

代码结构不断被修补。每一轮的需求补充，都会让AI在原有代码基础上添加新逻辑，而不是从一开始就以完整需求为基础进行整体设计。最终产出的代码，往往是结构混乱的"修补作品"，难以维护。

上下文窗口的损耗。随着对话轮次增加，上下文窗口被大量的历史对话占用，AI对早期需求和约束的"记忆"会逐渐模糊，后期的修改容易破坏前期已经正确的逻辑。

时间成本。六轮对话花费的时间和精力，往往超过了一开始就写好完整提示词所需的成本。

4.3.2 提示词使用中的常见问题

■ 问题四：需求描述不完整

下面是一个正面示例（全量需求一次性输入）,展示了"全量需求一次性输入"的重要性。这里通过这个案例想要说明的是，当需求完全明确时，一次性给出完整需求比"逐步迭代"效率高得多。需要注意的是，这个案例中完整保留了“五要素”模板（3.2.4节中的提示词模板），因为JWT认证涉及安全性、多模块协作，是一个复杂任务。但在实际项目中，如果项目上下文已在项目说明书中建立，"项目背景"要素可以省略。具体提示词如下：

```
## 任务
实现一套完整的JWT认证中间件，包含Token验证、Token刷新和Token黑名单功能。
一次性实现以下5个功能：
1. Token验证中间件（从Bearer Token提取并验证JWT）
2. 过期处理（捕获 ExpiredSignatureError）
3. 无效处理（捕获 JWTError）
4. 刷新 Token 接口（POST /api/v1/auth/refresh）
5. Token 黑名单 / 注销接口（POST /api/v1/auth/logout）
```

```
## 约束
- JWT 算法固定为 **HS256**，禁止支持多种算法；
- **绝对禁止**硬编码密钥（从环境变量 `JWT_SECRET` 读取）；
- Redis 操作必须使用已有的 `redis_client`；
- 中间件不得阻塞白名单路径；
- 错误响应格式：`{"code": "ERROR_CODE", "message": "描述"}`。

## 验收标准
- [ ] 有效 Token → HTTP 200
- [ ] 过期 Token → HTTP 401，`code == "token_expired"`
- [ ] 格式错误 Token → HTTP 401，`code == "invalid_token"`
- [ ] 有效 refresh_token → 返回新 access_token 和 refresh_token
- [ ] 已加入黑名单的 refresh_token → HTTP 401
- [ ] logout 后旧 Token → HTTP 401
```

4.3.2 提示词使用中的常见问题

■ **问题四：需求描述不完整**

下表给出了“五要素”模板是如何根治“逐步迭代”的问题的。

"逐步迭代"的问题	“五要素”模板的解决方案
每轮补充需求 → 代码结构被修补	需求描述部分一次性列出5个功能，AI从一开始就做整体设计
上下文窗口累积 → 早期约束被遗忘	各要素集中放置，AI每次生成代码时都能"看见"全部约束
格式规范在第三轮才想起来 → 代码格式不一致	约束边界和参考部分在代码生成前就声明了格式规范
测试用例在最后才想起来 → 覆盖不全	验收标准作为生成目标，AI写代码时会朝着"通过测试"的方向写

开发者在开始编写提示词之前，一定要先问自己：我已经把这个功能的完整边界想清楚了吗？ 如果没有，先把各要素补全，再开始与AI的对话。这十分钟的前期投入，通常可节省后续数十轮修补对话的时间。

4.3.3 与AI协作的实践经验

■ 验收优先：先定义正确标准，再让AI执行

1. 在所有实践经验中，最有价值的一条是：将"定义验收标准"作为任何AI编程任务的第零步。AI编程中成本最高的并非Token消耗或生成速度，而是返工。返工通常并非源于“AI语法错误”，而是源于“验收标准未明确、边界条件未定义、缺乏可验证的正确性路径”。

2. 把任何任务改写成“一句话验收”的习惯，非常有价值。将AI想象成是你的下级，你正在给他分配任务，为了让AI的产出符合你的预期，你要明确告诉它验收标准是什么，工作完成到什么程度算是优秀。

3. 这与“五要素”框架中"验收标准优先"的思想完全一致，也是软件工程中测试驱动开发（TDD）思想在AI编程场景中的迁移应用。

4.3.3 与AI协作的实践经验

■ 小步推进：一次只做一件事

把大任务拆成小任务，是AI编程中效率最高、风险最低的工作方式。这里给出反面示例和正面示例：

✗ 反面示例（一次性堆需求）：

"帮我实现登录、注册、找回密码、个人中心、修改密码、邮箱验证这些功能"

✓ 正面示例（逐步推进）：

第一轮："先实现登录功能，要求：邮箱和密码登录、记住登录状态、错误提示。"

（完成并验证后）

第二轮："现在实现注册功能，参照登录的代码风格。"

（完成并验证后）

第三轮："实现找回密码功能..."

每完成一个任务就立即测试验证，确认没有问题再进入下一个。这与软件工程中"持续集成"的思想一致：问题越早发现，修复成本越低。

4.3.3 与AI协作的实践经验

■ **善用版本控制：让Git成为最可靠的恢复工具**

在AI辅助编程中，版本控制系统不仅是代码管理工具，更是错误管理工具。当AI输出开始失控时，Git是最可靠的恢复手段。对于代码的版本控制，推荐的工作方式如下：

每个新功能从干净的Git状态开始。在AI开始修改代码之前，确保工作目录是干净的（git status 无未提交变更），这样任何问题都可以通过“git checkout .”一键还原。

一个小功能点一次提交。不要等到完成所有功能再一次性提交。AI生成的每个独立可工作的部分都应当及时提交，保留清晰的历史。

遇到问题及时重置。如果AI的多次修改让代码越来越复杂，不要继续"修补"，这时应该使用“git reset --hard HEAD”回到已知的干净状态，开启新对话，重新用完整的提示词来解决问题。

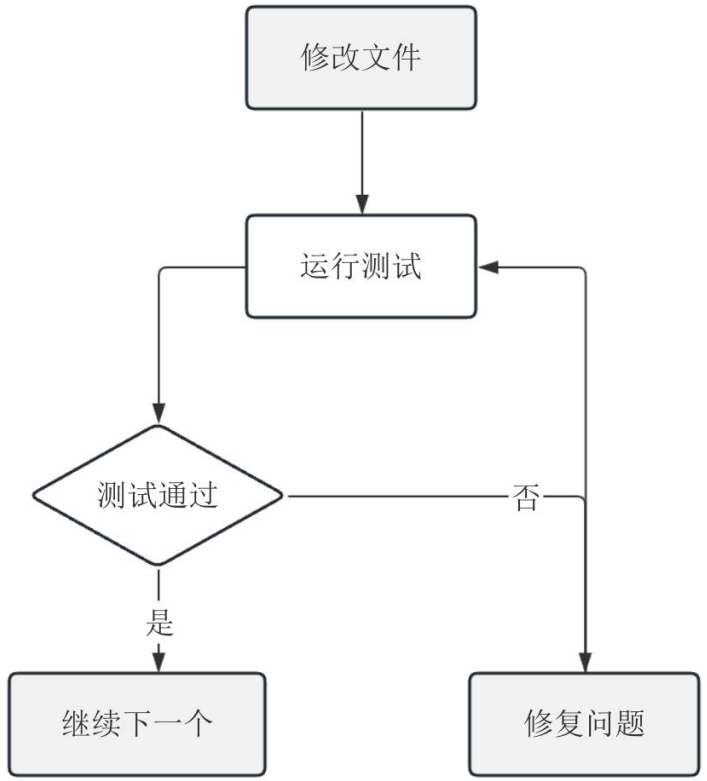
避免累积问题。多次失败的尝试会产生一层又一层的修补代码。当你最终找到解决方案时，先回滚再干净地实现，而不是在失败代码上继续修改。

4.3.3 与AI协作的实践经验

■ **及时验证，不要盲目信任**

在AI修改了代码之后，不要立刻继续下一个任务，而是先测试验证当前的修改是否正确。一个反面示例是，开发者盲目信任AI，让 AI 连续修改了 10 个文件，最后发现第一个文件就有错误，整个改动链条全部需要重审。

正确的做法是对于AI生成的代码一定要及时验证，如图所示。这种"修改一步验证一步"的工作节奏，能够将问题的发现时间和修复成本降到最低。



Part 04

与AI编程工具协作

4.4与AI编程工具协作

写给AI的
项目说明书

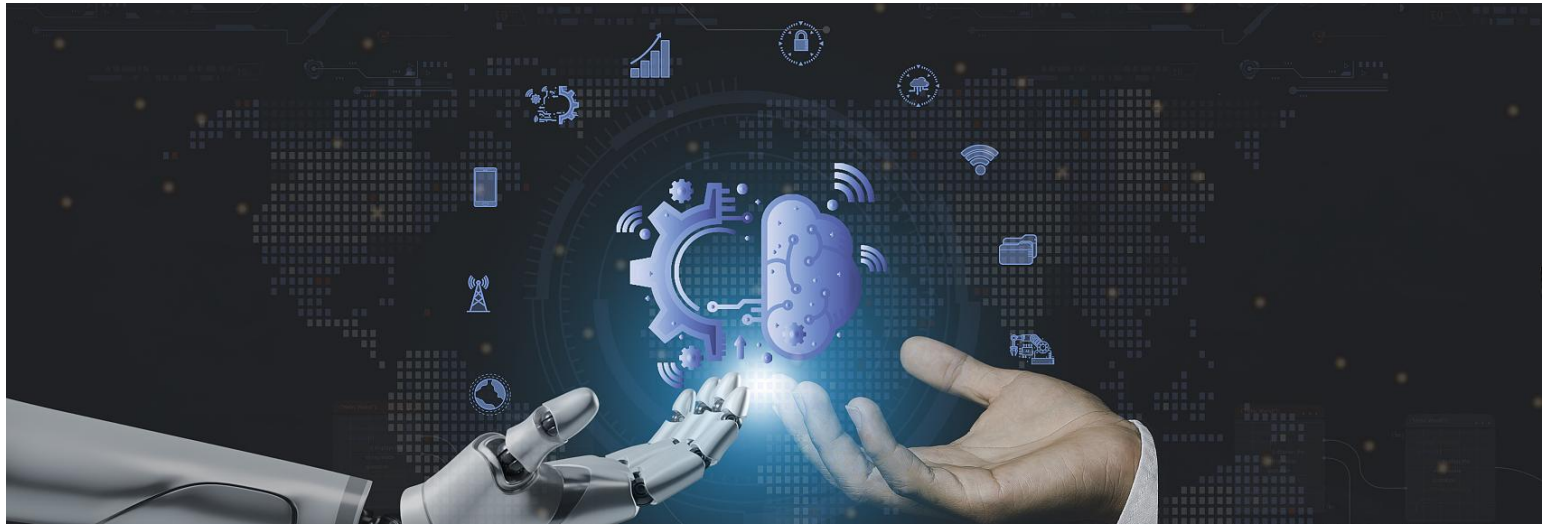
限制AI行为的
编码规则

4.4.1写给AI的项目说明书

■ 什么是AI项目说明书

前面几节介绍的提示词写作技巧，都是针对单次对话的。但在实际项目开发中，尤其是在维护和扩展已有代码库时，每日需与AI进行大量对话：修复缺陷、编写新接口、重构函数……每次都要在提示词开头重新说明项目背景，既繁琐又容易遗漏。更关键的是，如果AI不了解存量项目的架构约定和编码规范，它生成的代码很容易与现有代码风格不一致，引入"风格碎片化"问题。

当前主流的AI编程工具提供了一种更高效的解决方案——项目说明书（AGENTS.md）。项目说明书是放置在项目根目录的一个Markdown格式文件，专门用于向AI介绍项目的基本信息、架构背景和协作规范。当AI编程工具打开项目时，它会自动将项目说明书的内容注入每次对话的上下文中，使AI在每次对话时都"记得"你的项目是什么样的。这意味着，仅需维护一份项目说明书，即可在每次对话中避免重复说明项目背景，同时，保证AI生成的代码始终与存量项目的架构和风格保持一致。



4.4.1写给AI的项目说明书

■ 项目说明书里要写什么

一份有效的项目说明书，需要在两个目标之间取得平衡：

足够完整

AI能够理解项目背景，不需要在每次对话中补充基础信息。

足够简洁

不要让项目说明书变成一本"百科全书"，将有限的上下文窗口浪费在冗余信息上。

OpenAI Codex团队在实战经验中特别强调了这一点，他们给出的建议是："项目说明书应当是一个内容目录（Table of Contents），而不是一本百科全书。保持在100行左右，指向其他地方的真实信息源，渐进式引导AI获取它需要的信息"。

4.4.1写给AI的项目说明书

一份有效的项目说明书通常包含以下内容：

项目简介

两三句话说明这是什么系统、服务什么业务目标、覆盖哪些主要功能模块。

技术栈与版本

列出核心依赖及其精确版本。这对于避免AI使用过期API至关重要。AI的训练数据涵盖各个历史版本，如果不指明版本，它可能使用已废弃的API。

目录结构

简要说明各主要目录和文件的职责，帮助AI在被要求修改某个功能时，能够快速定位到正确的文件。

架构约定

说明项目遵循的架构模式，如分层架构、DDD（Domain-Driven Design）等，并说明项目的关键设计决策，如"API层不得直接访问数据库"、"所有异常必须继承自AppException"。

编码规范

命名惯例、注释要求、错误处理方式等。这些规范在没有明确说明的情况下，AI会使用自己认为"合理"的默认值，与团队规范可能不符。

常用命令

运行测试、启动本地服务、生成数据库迁移文件等命令的准确写法。当AI在建议你执行某些命令时，它会参考项目说明书中的内容，而不是猜测。

重要约束（可选）

项目中特别需要AI注意的硬性约束，如"禁止直接修改 core/ 目录下的文件"、"所有对外接口必须保持向后兼容"。

4.4.1写给AI的项目说明书

下面是一个完整的AGENTS.md示例：

```
# 极客书城API服务（GeekBooks Backend）

## 项目简介
本项目是"极客书城"在线书店的后端API服务，负责用户管理、图书目录、
订单处理和支付集成。服务以REST API形式对外提供，供Web前端和移动端消费。

## 技术栈
- **语言**：Python 3.12
- **框架**：FastAPI 0.115
- **ORM**：SQLAlchemy 2.0（异步模式，使用 asyncpg 驱动）
- **数据库**：PostgreSQL 15（主库）+ Redis 7（缓存 / 消息队列）
- **任务队列**：Celery 5.3 + Redis（Broker）
- **测试**：pytest + pytest-asyncio + httpx（AsyncClient）
- **代码质量**：Ruff（lint + format）+ mypy（类型检查）
```

4.4.1写给AI的项目说明书

```
## 目录结构
src/
├── api/          # 路由层：仅做请求参数校验和响应组装，不含业务逻辑
├── services/     # 业务逻辑层：所有业务规则在此实现
├── repositories/  # 数据访问层：封装所有数据库和缓存操作
├── models/       # SQLAlchemy数据模型（只定义结构，不含逻辑）
├── schemas/      # Pydantic输入/输出模型（请求体/响应体）
├── core/         # 全局配置、异常定义、依赖注入、中间件
└── tasks/        # Celery异步任务

## 架构约定（AI必须遵守）
- 严格遵循三层架构：api → service → repository;
- API 层不得直接访问数据库（禁止在api/目录下import Session）；
- 数据库操作必须封装在repository层;
- 所有业务异常继承自`AppException`;
- ...（其他架构约束）
```

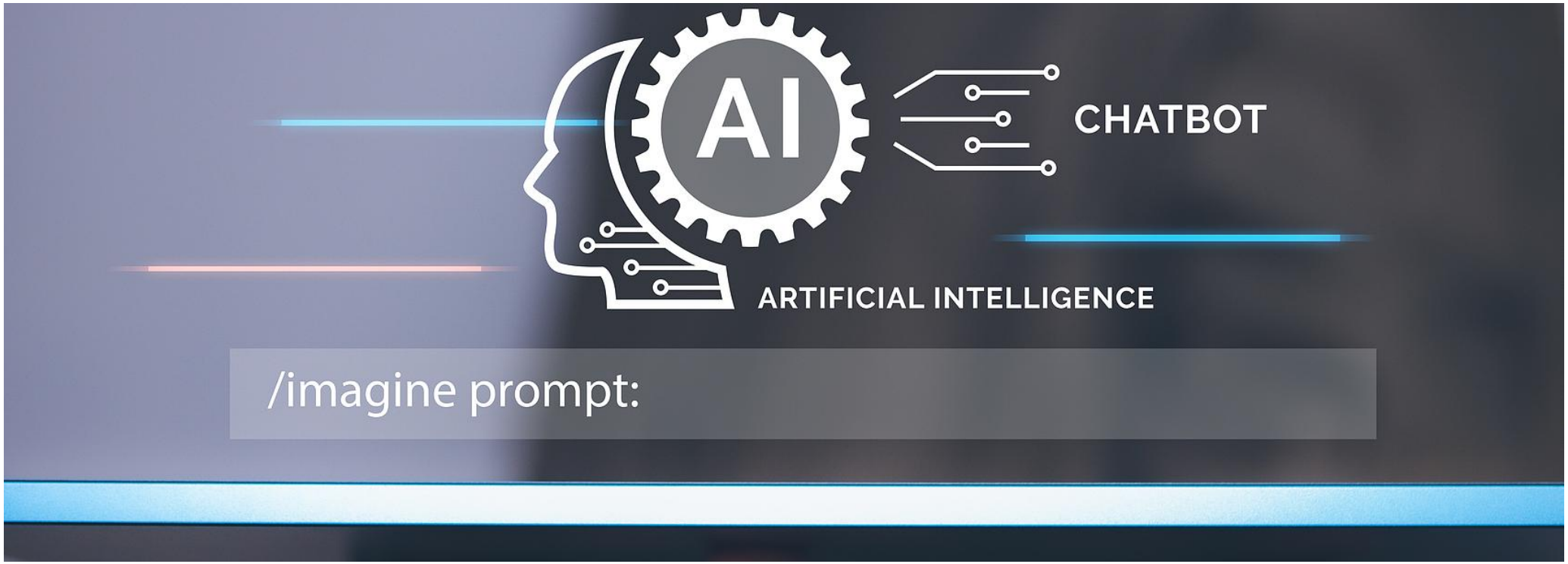
4.4.1写给AI的项目说明书

```
## 编码规范
- 所有公共函数和类方法必须有中文docstring；
- 使用Python 3.12的类型注解语法；
- 函数长度不超过50行；
- 金额字段使用Decimal类型，禁止float；
- ...（其他编码规范）

## 常用命令
- 启动开发服务器：`uvicorn src.main:app --reload --port 8000`
- 运行全部测试：`pytest tests/ -v --asyncio-mode=auto`
- 代码检查和格式化：`ruff check src/ && ruff format src/`
```

4.4.1写给AI的项目说明书

■ 项目说明书写给谁看

项目说明书的主要读者是AI，但它同样对人类有巨大的价值。一名新加入团队的工程师，阅读项目说明书后能够快速了解项目的技术选型、架构决策和协作规范，不需要花大量时间阅读代码或询问老同事。因此，维护一份高质量的项目说明书，既是对AI协作效率的投资，也是团队知识传承的重要手段。特别是在团队成员变动频繁的项目中，项目说明书成为了一份常驻的"项目入职手册"。



4.4.1写给AI的项目说明书

■ 在TRAE中使用项目说明书

在TRAE中启用项目说明书的操作步骤如下：

第一步

在项目根目录创建 AGENTS.md 文件，按照上述格式填写项目信息。

第二步

打开TRAE“设置”中心（点击界面右上角的齿轮图标）。

第三步

在左侧导航栏中点击“ 规则与记忆”，进入规则配置页面。

第四步

在“ 导入设置” 区域，找到“ 将AGENTS.md包含在上下文中”开关，确保其处于开启状态，如图所示。

4.4.1写给AI的项目说明书

■ 在TRAE中使用项目说明书

开启后，TRAE会在每次AI对话时，自动将项目根目录的项目说明书内容注入上下文。你无需在每次对话中手动引用，AI会始终在了解项目背景的前提下工作。

TRAE还支持多层级项目说明书支持。对于大型项目，TRAE还支持在子目录中放置独立的项目说明书，用于为特定模块配置专属的AI行为指导。例如，前端模块的“frontend/AGENTS.md”描述前端技术栈和约定，后端模块的“ backend/AGENTS.md”描述后端部分。子目录的项目说明书在处理该目录下的文件时会叠加生效，与根目录的项目说明书形成互补。

4.4.2限制AI行为的编码规则

■ 规则是什么

项目说明书描述"项目是什么"，而规则（Rule）则告诉AI "在这个项目中，你应该怎么做事"。项目说明书和规则这二者互为补充：项目说明书提供背景知识，规则约束行为模式。在存量项目维护场景中，规则的作用尤为关键，它是将项目长期积累的工程标准"固化"为AI可执行约束的核心机制。

如果把项目说明书比作"项目介绍手册"，那么，规则就是"行为准则"或"编码公约"。规则的核心价值在于：将存量项目的工程标准，转化为AI可以执行的硬性约束，防止AI在每次生成代码时"自由发挥"而破坏已有代码的一致性。

在没有规则的情况下，AI会根据训练数据中的"统计平均"来决定代码风格、错误处理方式和测试写法，这通常意味着每次生成的代码风格都会有细微差异，与存量项目长期积累的编码惯例不一致。规则的作用，就是把这种"自由发挥"收束到项目已有的规范范围内。

4.4.2限制AI行为的编码规则

■ 规则是什么

规则最常见的应用场景包括：

1	编码规范约束	要求AI生成的代码遵循团队的命名规范、格式要求和注释风格，确保AI生成的代码与人类写的代码风格一致，降低代码审查（Code Review）的“摩擦”成本。
2	技术选型约束	禁止使用已被团队废弃的库或API。例如，如果团队决定从requests迁移到httpx，可以在规则中写"禁止使用requests库，所有HTTP请求必须使用httpx"。
3	文档与注释规范	规定函数docstring的格式、语言（中文/英文）和必须包含的信息，确保AI生成的代码与团队现有代码的文档风格一致。
4	测试规范	规定测试文件的命名规则（test_<模块名>.py）、测试函数的命名格式（test_<功能>_<场景>）以及测试的组织方式，使AI生成的测试代码与手写测试代码结构一致。
5	安全规范	禁止生成包含敏感信息硬编码（比如把密码、密钥写死在代码里）、SQL字符串拼接等危险模式的代码。

4.4.2限制AI行为的编码规则

■ 在TRAE中使用规则

TRAE支持两种类型的规则，适用于不同的使用场景：

全局规则（Global Rules）

在所有项目中均生效，适合存放个人习惯偏好，如"回答时使用中文"、"解释代码时先给出整体思路再展示细节"、"不要在未经询问的情况下修改测试文件"等。全局规则在TRAE设置中的规则页面直接编写，存储在本地配置中，不属于代码仓库的一部分。

项目规则（Project Rules）

仅在当前项目内生效，以Markdown文件形式存放在项目目录的“.trae/rules/”下，可以纳入Git版本管理，供团队全员共享。每名团队成员只需克隆代码仓库，就能获得完整的AI行为规范，无需单独配置。

4.4.2限制AI行为的编码规则

■ 在TRAE中使用规则

在TRAE中创建项目规则的完整步骤如下（见下图）：

第一步 打开TRAE的“设置”，选择“规则与记忆”，在“规则”中点击“+创建”，在弹窗中选择“项目”。

第二步 输入规则名称（如 python-style），点击“确认”，TRAE会在“.trae/rules/”目录下创建同名的 .md 文件并自动打开。

第三步 打开规则文件，可以TRAE的窗口顶部选择生效方式，然后在正文中用Markdown格式编写规则内容。

4.4.2限制AI行为的编码规则

■ **在TRAE中使用规则**

规则文件支持四种生效方式，如表所示。

应用模式	YAML配置	说明	适用场景
始终生效	alwaysApply: true	每次对话均自动生效	全局编码规范
指定文件生效	globs: "src/**/*.py"	仅对匹配文件生效	特定语言或模块的规范
智能生效	设置 description 字段	AI根据description自动判断是否应用	场景化规范
手动触发生效	（无特殊配置）	在对话中用 @规则名 手动触发	偶尔用到的特殊规范

4.4.2限制AI行为的编码规则

■ 在TRAE中使用规则

下面是一个完整的项目规则文件示例：

```
---
alwaysApply: true
---

# Python 编码规范（极客书城项目）

## 命名规范
- 变量名和函数名：snake_case（如 `get_user_by_id`）；
- 类名：PascalCase（如 `UserService`）；
- 常量：UPPER_SNAKE_CASE（如 `MAX_RETRY_COUNT`）；
- 私有方法/属性：前缀下划线（如 `_validate_password`）。

## 类型注解
- 所有函数参数和返回值必须有类型注解；
- 使用Python 3.12内置泛型语法（`list[str]`、`dict[str, int]`）；
- Optional类型写成 `str | None`，不使用 `Optional[str]`。
```

4.4.2限制AI行为的编码规则

■ 在TRAE中使用规则

```
## 错误处理
- 禁止裸`except:`或`except Exception:`;
- 必须捕获具体异常类型;
- 业务错误使用自定义异常（继承 AppException）。

## 安全规范
- 禁止字符串拼接构造SQL;
- 禁止在代码中硬编码密码、API Key等敏感信息;
- 用户输入必须经过Pydantic校验。
...（其他规则）
```

4.4.2限制AI行为的编码规则

■ 在TRAE中使用规则

规则的长期维护同样重要。随着项目的演进，某些规则可能变得过时或不再适用；随着团队在AI协作中积累经验，也会发现新的需要约束的模式。建议将规则文件的更新纳入常规的代码审查流程，与代码一起演进。

此外，建议将“.trae/rules/”目录（及项目说明书）纳入Git版本管理，并在项目的README或CONTRIBUTING.md中说明AI协作规范的使用方式。这样，任何新加入的团队成员只需克隆代码库，就能获得完整的AI协作环境配置，不必重新摸索。

Part 05

AI维护存量项目

4.5 AI维护存量项目

存量项目的特点

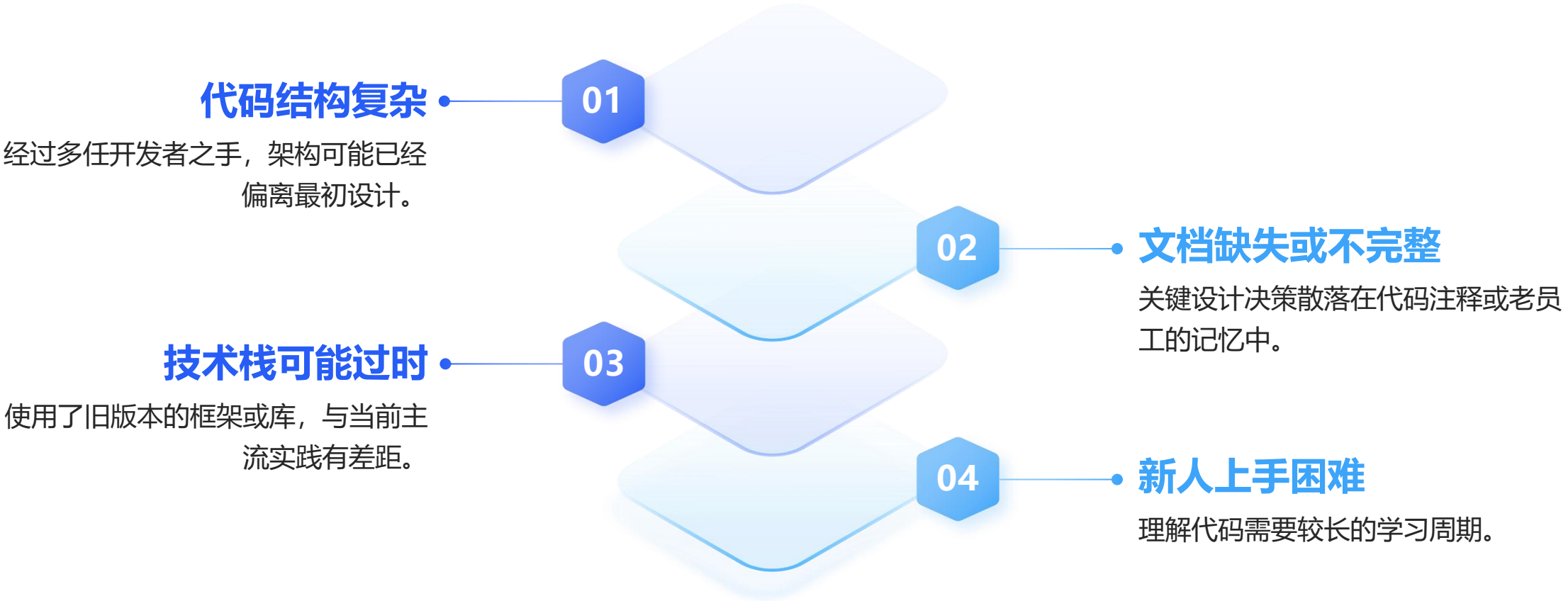
存量项目为什么需要特别对待

存量项目维护的四大实践

存量项目维护的工作流

4.5.1存量项目的特点

存量项目是指已经上线运行、维护时间较长、代码量较大的已有系统。这类项目通常具有以下特征：



存量项目是真实世界中最常见的项目形态。根据行业统计数据，企业中约70%～80% 的开发工作是在维护和扩展已有系统，而非从零构建新项目。因此，掌握如何用AI高效维护存量项目，是AI编程能力中最重要的组成部分。

4.5.2存量项目为什么需要特别对待

在增量开发中（新功能从零开始），AI通常表现良好，你可以给它完整的上下文，让它生成风格一致的代码。但在存量项目中，情况要复杂得多，会存在很多问题，具体如下：

问题一

AI不了解项目的历史约束。存量项目中存在大量"隐性知识"，也就是那些在代码中体现、但从未被文档化的设计决策。例如，为什么这个接口用了POST而不是GET？为什么这个字段叫 user_id 而不是 uid？为什么要在这里加缓存？这些决策背后往往有其合理性，但如果AI不知道这些约束，它生成的代码可能会"技术上正确"但"业务上错误"。

问题二

AI会引入风格不一致的代码。在没有约束的情况下，AI会根据训练数据中的"统计平均"来决定代码风格。每次生成的代码风格都可能略有差异，长期积累会破坏代码库的一致性，增加维护难度。

问题三

直接修改可能导致回归问题。存量项目往往缺乏完整的测试覆盖。冒然让AI修改某个模块，可能破坏已有的隐式依赖，导致意想不到的回归问题。

这些问题不是AI能力不足导致的，而是因为AI没有获得足够的项目上下文。解决这些问题的方法，就是本节下面将要介绍的四大实践。

4.5.3存量项目维护的四大实践

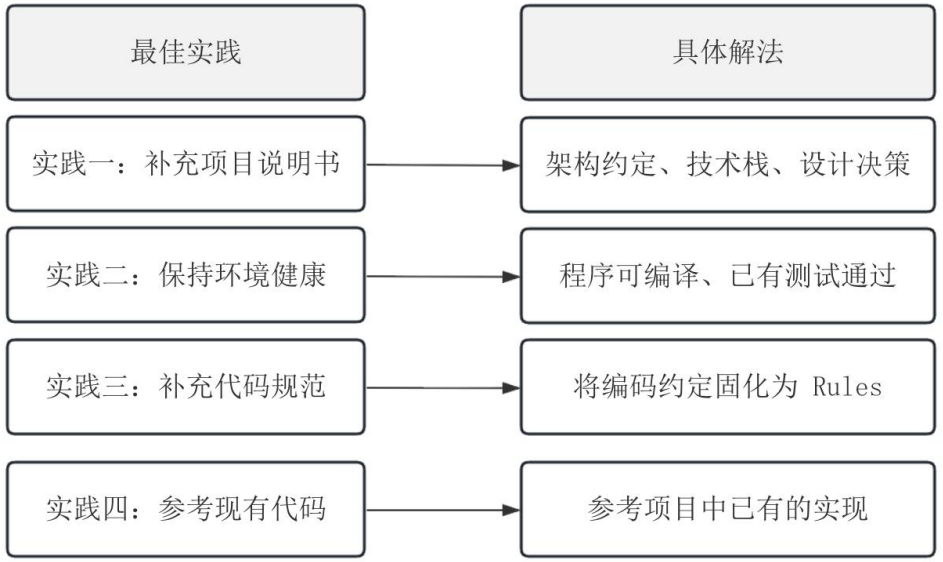
经过大量工程实践的验证，AI维护存量项目的最佳策略可以归纳为四大核心实践，如图所示。为什么是这四大实践？因为这四大实践分别解决了存量项目维护中的不同问题：

“补充项目说明书”解决的是"AI不了解项目背景"的问题。

“保持环境健康”解决的是"修改缺乏安全保障"的问题。

“补充代码规范”解决的是"代码风格不一致"的问题。

“参考现有代码”解决的是"AI输出质量不稳定"的问题。



4.5.3存量项目维护的四大实践

■ 实践一：补充项目说明书

如果你的项目还没有项目说明书，第一步就是为项目创建一份。项目说明书的价值在于：它是一份"项目入门手册"，让AI能够像了解项目的新成员一样工作。那么，存量项目的项目说明书应该包含什么？与新项目不同，存量项目的项目说明书需要特别关注以下几点：

4.5.3存量项目维护的四大实践

■ 实践一：补充项目说明书

（1）架构决策记录（ADR）

存量项目中存在大量"当年为什么这么做"的设计决策。这些决策往往是踩坑后的经验总结，但从未被正式记录。

在项目说明书中补充这些决策，能帮助AI避免"重复踩坑"，具体如下：

```
## 重要架构决策

**为什么订单模块不使用软删除？**
订单数据有财务合规要求，必须保留完整历史，不能物理删除。
→ AI 必须理解：任何涉及订单删除的功能，都必须是软删除或状态标记。

**为什么用户模块有 user_id 和 uuid 两个标识符？**
- user_id：内部使用，int 类型，自增，供数据库 join 使用；
- uuid：对外暴露，string 类型，用于 API 响应和前端交互，防止 ID 枚举攻击。
→ AI 必须理解：不能随意替换这两个字段。
```

4.5.3存量项目维护的四大实践

■ 实践一：补充项目说明书

（2）关键模块的职责说明

存量项目中，某些模块的职责边界可能不清晰。在项目说明书中补充关键模块的说明，帮助AI理解"这个模块负责什么、不负责什么"，具体如下：

```
## 关键模块职责

**UserService（src/services/user_service.py）**
- 负责：用户注册、登录、权限校验、资料修改
- 不负责：订单处理（属于 OrderService）、消息推送（属于 NotificationService）
→ AI 必须理解：不能在 UserService 中直接操作订单数据
```

4.5.3存量项目维护的四大实践

■ 实践一：补充项目说明书

（3）技术债说明

存量项目中必然存在技术债。在项目说明书中诚实说明这些技术债，能帮助AI在必要时避开它们，具体如下：

```
## 已知技术债

- [ ] 旧版支付模块（src/payment_v1/）使用同步调用，已计划迁移到 payment_v2
  → 新功能禁止在 payment_v1 上继续开发
- [ ] 缓存层目前使用 Redis 字符串，建议迁移到 Hash 结构（已在 backlog 中）
  → AI 生成缓存代码时，可以参考现有模式，但不需要改进架构
```

4.5.3存量项目维护的四大实践

■ 实践一：补充项目说明书

（4）代码审查要点

存量项目的代码审查往往有一些特别需要注意的点。在项目说明书中补充这些审查要点，帮助AI生成更容易通过审查的代码，具体如下：

```
## Code Review 关注点

**订单模块**
- 必须保留审计日志（谁、什么时候、做了什么修改）
- 金额计算必须使用 Decimal，禁止 float

**用户模块**
- 敏感字段（密码、手机号）不得出现在日志中
- 用户注销必须走软删除流程
```

4.5.3存量项目维护的四大实践

■ 实践二：保持环境健康

存量项目的一个普遍问题是测试覆盖不足。在用AI修改存量项目之前，首要任务是确保项目处于"干净可用"的状态，这是防止AI破坏已有逻辑的第一道防线。

为什么环境健康如此重要？主要有以下几个方面的原因：

建立质量基线

测试让你知道"修改前的行为是什么"，从而判断"修改后的行为是否正确"；

防止回归

完整的测试套件能快速发现AI修改引入的问题；

降低心理负担

有测试保护，开发者更敢于接受AI的修改。

具体而言，在让AI开始工作之前，确保以下几点：

程序可编译

代码没有语法错误，能够正常启动运行；

已有测试通过

运行现有测试套件，确保全部通过，这是验证存量逻辑未被破坏的基准线；

环境干净

没有遗留的临时文件、未提交的改动或正在运行的进程。

AI修改代码后，立即重新运行测试套件，确认已有测试依然全部通过。如果测试失败，说明AI的修改引入了回归问题，需要及时修复。

4.5.3存量项目维护的四大实践

■ 实践三：补充代码规范

存量项目可能已经有一套代码规范，但这些规范往往只存在于代码审查文档或老员工的记忆中，并没有被正式记录下来。将代码规范补充到规则中，能让AI自动遵守这些规范，减少风格碎片化。

那么，如何识别存量项目的代码规范呢？代码规范往往体现在现有代码中。阅读代码时，注意以下模式：

01

命名约定

变量、函数、类的命名风格。

02

错误处理方式

是否使用自定义异常、是否记录日志。

03

文档风格

是否有docstring、使用什么格式。

04

依赖管理

是否禁止使用某些库。

4.5.3存量项目维护的四大实践

■ 实践三：补充代码规范

需要将规范固化为规则，识别出规范后，将其写入规则文件，具体如下：

```
---
alwaysApply: true
---

# 项目编码规范

## 命名规范（已通过代码审查确认）
- 函数名：snake_case（如 `get_user_by_id`）
- 类名：PascalCase（如 `UserService`）
- 常量：UPPER_SNAKE_CASE（如 `MAX_RETRY_COUNT`）

## 错误处理规范
- 业务错误使用自定义异常（继承 AppException）
- 禁止使用 `raise HTTPException`（在 API 层统一处理）
- 捕获异常后必须记录日志

## 文档规范
- 所有公共函数必须有中文 docstring
- docstring 格式：功能描述 + 参数说明 + 返回值说明
```

4.5.3存量项目维护的四大实践

■ 实践四：参考现有代码

在存量项目中，让AI参考现有代码而不是"从零创作"，是提升输出质量最有效的方法。这个理念最初被形象地称为"胶水编程"，AI的主要工作是把现有代码"粘"到新的业务场景上。

什么是胶水编程呢？"胶水编程"是一个很早就有的概念，指的是用少量代码将已有的模块、库或服务连接起来形成一个完整系统。在AI时代，这个概念重新火了起来，因为，AI天生擅长"照着做"，只要给它一个高质量的参照物，它就能生成风格一致的代码。

为什么"照着做"有效呢？这是因为，大语言模型的工作方式是，在给定上下文的条件下预测下一个最可能出现的词元（Token），当你提供高质量的参照代码时，模型的预测空间被大幅收窄，它会倾向于生成与参照代码结构和风格高度一致的输出。

4.5.3存量项目维护的四大实践

■ 实践四：参考现有代码

在提示词中提供参考代码的具体示例如下：

```
## 任务
参照以下"用户管理"模块的实现，为"商品管理"创建类似的功能。

## 约束
- 严格遵照参照代码的结构、命名风格和错误处理模式
- 不要引入参照代码中没有的设计模式

## 参照代码
[粘贴 user_service.py 的完整代码]

## 差异说明
1. Product没有密码字段，去掉所有密码相关逻辑
2. Product需要额外的库存扣减方法
3. Product的删除是软删除
```

4.5.3存量项目维护的四大实践

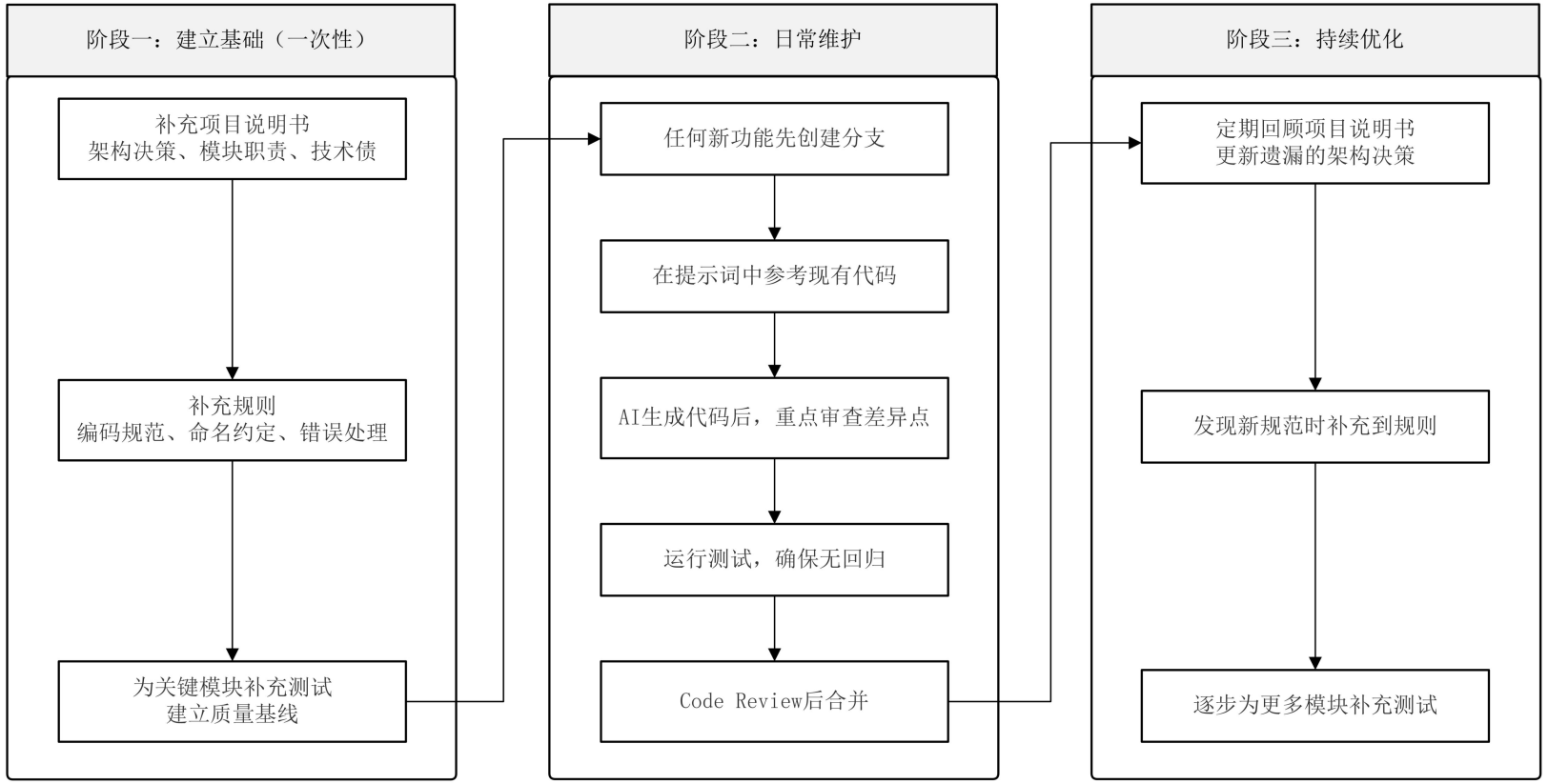
■ **实践四：参考现有代码**

参考现有代码时也要注意以下3个事项：

确保参照物质量过关	**如果参照代码本身有问题，AI会复制这些问题。**
明确指出差异点	**不要让AI自己猜测哪些地方需要调整。**
重点审查差异部分	**AI在"复制"部分通常没问题，问题往往出在"定制"部分。**

4.5.4存量项目维护的工作流

现在将四大实践整合成一个系统化的工作流（如图所示）。



一个实用的经验法则是，在存量项目中，使用AI修改代码前，应自问：这份项目说明书和规则是否已经包含了足够的上下文？如果没有，先补充它们，再让AI执行。这个顺序很重要，先建立上下文，再进行修改，能显著降低引入问题的风险。

Part 06

故障诊断与修复

4.6故障诊断与修复

基于提示词的AI故障诊断

AI编程的可观测性

防御性编程

4.6.1基于提示词的AI故障诊断

当代码出现问题时，AI可以成为强有力的故障诊断助手。但正如本章一再强调的：AI的诊断效果，完全取决于我们提供的上下文是否完整和准确。

一个有效的AI故障诊断请求，需要提供以下五类信息：

完整的错误信息

错误类型、错误消息和完整的堆栈追踪（Stack Trace）。

触发条件

什么操作触发了错误，是否能稳定复现，何时开始出现。

相关代码

错误发生的函数或模块，以及调用链上的关键上层代码。

环境信息

操作系统、运行时版本、关键依赖版本。

已尝试的方案

已经尝试过什么方法，结果如何。

下面是一个关于故障诊断提示词的反面示例（信息残缺）：

```
这里报错了，帮我看看：
AttributeError: 'NoneType' object has no attribute 'id'
```

4.6.1基于提示词的AI故障诊断

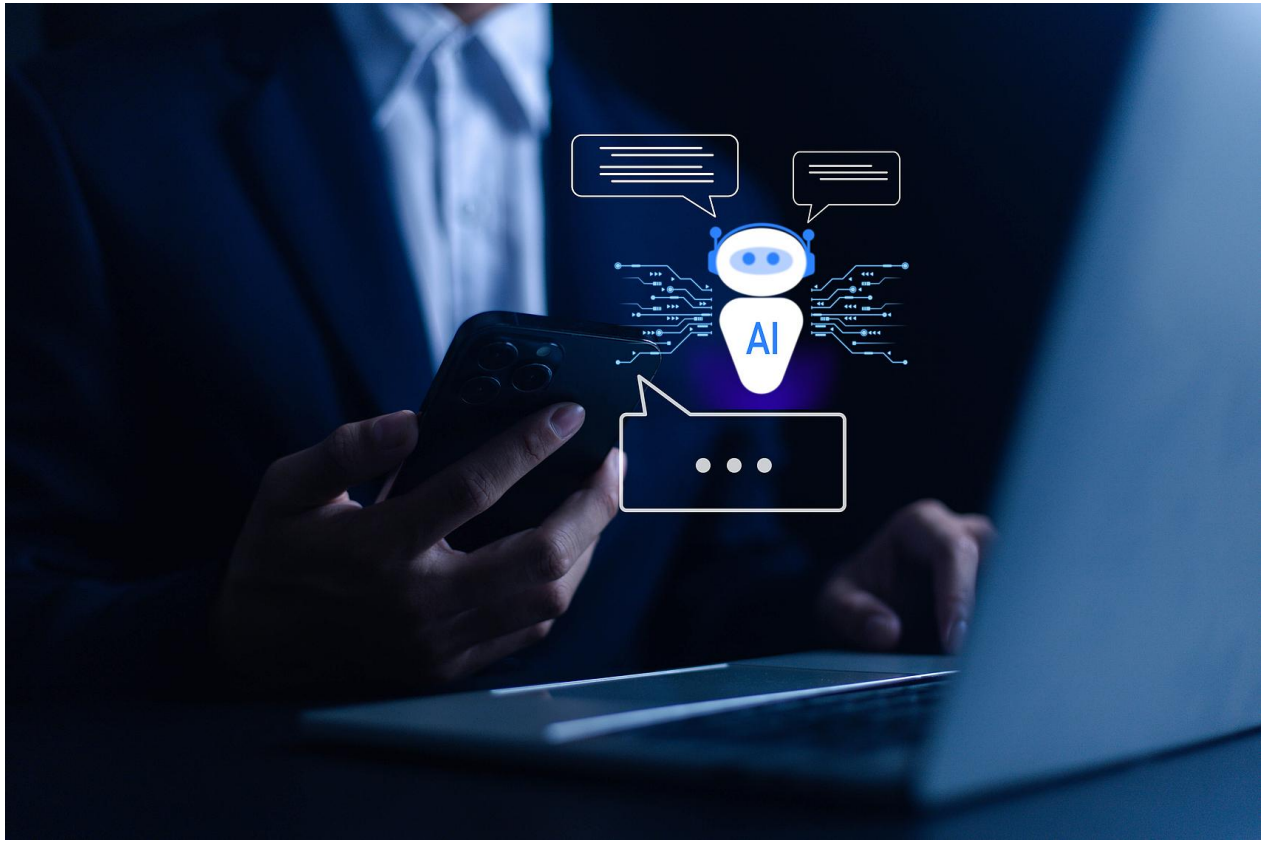
下面是一个正面示例（信息完整）：

```
## 错误信息
AttributeError: 'NoneType' object has no attribute 'id'
Traceback (most recent call last):
  File "src/services/order_service.py", line 87, in create_order
    discount = coupon.id  # ← 报错位置

## 触发条件
调用POST /api/v1/orders接口，请求体中包含不存在的coupon_id（如 99999）时触发。
已稳定复现。最近的代码改动是重构了coupon_repo.get_by_id方法。

## 相关代码
# coupon_repo.py - get_by_id在找不到记录时返回None
# order_service.py - create_order调用后未检查None值
[粘贴相关代码]

## 已尝试方案
加了if coupon_id: 检查，但coupon_id是有效整数（99999），问题在于该值在数据库中不存在。
```

4.6.1基于提示词的AI故障诊断

提供完整上下文后，AI不仅能快速定位问题，还能给出更系统性的修复方案——在Repository层统一处理"找不到记录"的情况，而不是在每个调用处单独加 None 检查。

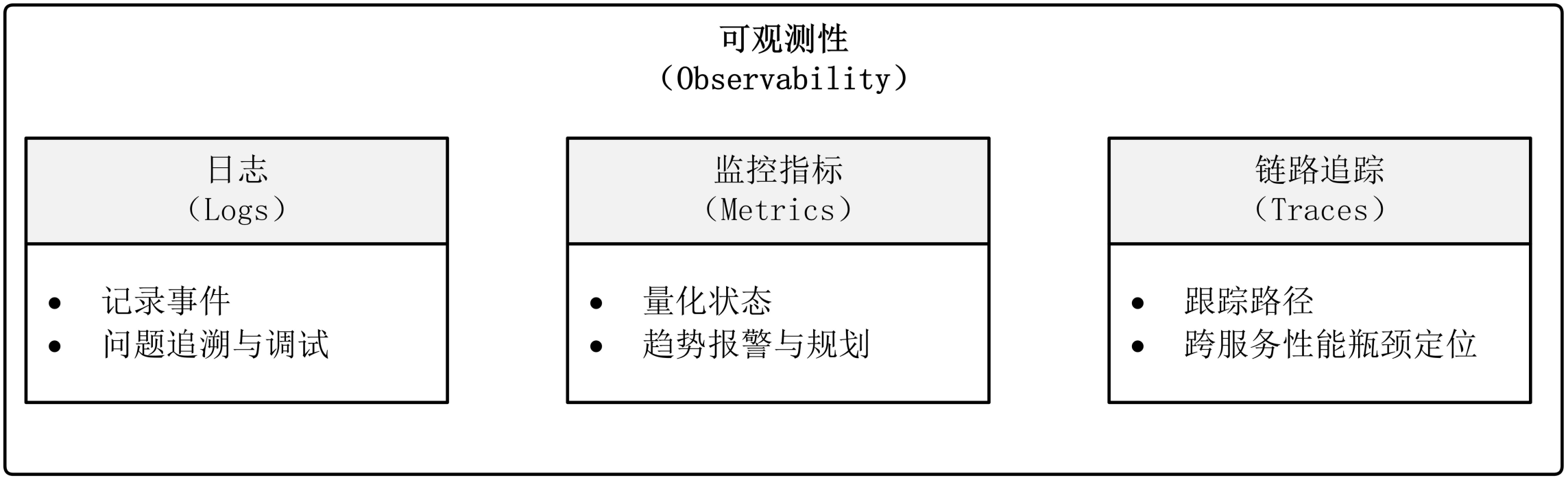
当AI对同一个Bug进行了多次修改（超过3次）但问题仍未解决时，不应在同一对话中继续追加修改。正确做法是：从Git回滚至干净状态，开启新对话，重新提供完整的诊断信息。



4.6.2AI编程的可观测性

可观测性（Observability）是指通过分析系统的外部输出，来了解系统内部状态的能力。在AI辅助诊断的场景中，可观测性数据是你能给AI提供的最有价值的诊断上下文。

可观测性包含三个支柱：日志（Logs）、监控指标（Metrics）和链路追踪（Traces），如图所示。



三者在故障诊断中各有价值：日志告诉AI "出错时具体发生了什么"，指标告诉AI "问题的时间范围和影响程度"，链路追踪告诉AI "请求在哪个环节出了问题"。三者结合能给AI提供最完整的故障现场。

4.6.2AI编程的可观测性

■ 日志

日志是最基础的可观测性手段，记录系统运行过程中的关键事件。良好的日志是AI故障诊断的基础数据来源。**日志可以分为两种典型的类型，即结构化日志和非结构化日志**。

传统的纯文本日志（非结构化）无法被日志平台解析，也难以被AI系统性分析，下面是一个非结构化日志的示例：

```
# 非结构化日志（不推荐）
logger.info(f"用户 {user_id} 创建了订单 {order_id}，金额 {total_amount}")
```

4.6.2AI编程的可观测性

■ 日志

现在AI编程的最佳实践是结构化日志，即以JSON格式输出，每个字段有独立的key，具体实例如下：

```
import structlog
logger = structlog.get_logger()

# 结构化日志（推荐）
logger.info("order.created",
    user_id=user_id,
    order_id=order_id,
    total_amount=str(total_amount), # Decimal 转字符串
    item_count=len(items)
)

logger.error("order.creation_failed",
    user_id=user_id,
    error_type=type(e).__name__,
    error_message=str(e),
    exc_info=True  # 自动附加堆栈追踪
)
```

4.6.2AI编程的可观测性

■ 日志

结构化日志支持按任意字段过滤和聚合，也更容易被AI分析，因为字段语义明确。下表给出了不同级别的日志及其使用场景。

级别	使用场景	示例
DEBUG	调试信息，仅在开发环境开启	函数入参和出参、SQL语句
INFO	正常流程的关键节点	用户登录、订单创建、任务完成
WARNING	可恢复的异常情况	重试触发、缓存未命中
ERROR	需要人工关注的错误	第三方服务调用失败、数据库连接失败
CRITICAL	系统级严重错误	服务启动失败、数据损坏风险

向AI提供日志时，建议只提供出错时间前后约5分钟、仅包含ERROR和WARNING级别的过滤结果，并标明哪一行是最终报错。

4.6.2AI编程的可观测性

■ 监控指标

监控指标（Metrics）是对系统运行状态的量化度量，以时间序列形式持续采集，通过Grafana等工具展示。常见的指标类型包括：

计数器（Counter）

单调递增，如请求总量、错误总数。

仪表盘（Gauge）

当前值，如活跃连接数、内存使用量、队列积压数。

直方图（Histogram）

分布统计，如接口响应时间的P50/P95/P99。

在日常开发中，重点关注的核心指标分为三层：

接口层

每秒请求量（RPS）、错误率（HTTP 5xx占比）、响应时间分位值（P95、P99）。

应用层

任务队列积压量、缓存命中率、数据库连接池使用率。

基础设施层

CPU使用率、内存使用率、磁盘I/O、网络带宽。

向AI描述故障时，提供量化的指标数据（如"响应时间从50ms升至3000ms，同时数据库连接池使用率达到100%"），比"系统变慢了"这种描述更能够传递具体得多的诊断信息，这样，AI就可以直接推断瓶颈是在数据库连接池，而不是从零开始排查。

4.6.2AI编程的可观测性

■ 链路追踪

链路追踪（Distributed Tracing）用于在微服务架构中追踪一个请求经过所有服务的完整路径，以及每个环节的耗时。这里需要了解几个核心概念，具体如下：

Trace（链路）

一次完整请求的所有操作，有唯一的Trace ID。

Span（跨度）

链路中的一个具体操作（数据库查询、HTTP调用），包含操作名称、开始时间、持续时间。

父子关系

Span之间的调用层次关系。

链路追踪在微服务中是定位跨服务问题的关键工具。当请求经过API网关 → 用户服务 → 订单服务 → 支付服务，整体响应时间异常时，链路追踪能精确告诉你每个环节消耗了多少时间。

4.6.2AI编程的可观测性

OpenTelemetry （简称OTel）是目前最主流的可观测性标准（由CNCF维护），提供统一SDK，支持自动埋点，与Python、Java、Go等主流语言均有良好集成。在提示词中可以直接要求AI集成OTel，具体如下：

```
## 约束
使用 OpenTelemetry SDK（opentelemetry-sdk + opentelemetry-instrumentation-fastapi）为服务添加链路追踪：
- 自动捕获所有 FastAPI 路由的 HTTP Span；
- 手动为 create_order 函数添加 custom Span，包含 user_id 和 order_id 属性；
- 通过 OTLP 协议将 Trace 数据导出到 localhost:4317（Jaeger 采集端口）。
```

4.6.3防御性编程

防御性编程（Defensive Programming）是指在代码中预先处理各种可能发生的异常情况，而不是假设一切都会按预期运行。如前所述，AI默认只处理常规路径。在提示词中显式要求防御性编程，或在规则中将其固化为规范，是让AI生成的代码具备生产级质量的关键。

防御性编程的核心包括输入校验、精确的异常处理、资源安全释放、并发保护，具体如下：

（1）输入校验。不信任任何外部输入，在使用前验证合法性，实例如下：

```
def calculate_discount(order_amount: Decimal, discount_rate: Decimal) -> Decimal:
    if order_amount <= Decimal("0"):
        raise ValueError(f"订单金额必须大于 0，当前值：{order_amount}")
    if not (Decimal("0") <= discount_rate <= Decimal("1")):
        raise ValueError(f"折扣率必须在 0.0～1.0，当前值：{discount_rate}")
    return (order_amount * discount_rate).quantize(Decimal("0.01"))
```

4.6.3防御性编程

（2）精确的异常处理。捕获具体异常类型，而不是宽泛的except Exception（会掩盖Bug），实例如下：

```
try:
    response = await client.post(payment_url, json=payload, timeout=10.0)
    response.raise_for_status()
    return response.json()
except httpx.TimeoutException:
    logger.error("payment.gateway_timeout", url=payment_url)
    raise PaymentGatewayTimeoutError("支付网关请求超时")
except httpx.HTTPStatusError as e:
    logger.error("payment.gateway_error", status_code=e.response.status_code)
    raise PaymentGatewayError(f"支付网关返回错误：{e.response.status_code}")
```

4.6.3防御性编程

（3）资源安全释放。使用Python上下文管理器确保数据库连接、文件句柄等资源在任何情况下都被正确释放，实例如下：

```
async with get_db() as session:
    async with session.begin():
        order = await order_repo.create(session, order_data)
        await inventory_service.deduct_stock(session, order.items)
        # 无论是否异常，session 都会被正确关闭
```

4.6.3防御性编程

（4）并发保护。对共享资源的修改使用数据库行级锁，防止竞态条件，实例如下：

```
async def deduct_stock(self, product_id: int, quantity: int) -> Product:
    product = await self.session.execute(
        select(Product)
        .where(Product.id == product_id)
        .with_for_update()  # 行级锁，防止超卖
    )
    product = product.scalar_one_or_none()
    if product is None:
        raise ProductNotFoundError(product_id)
    if product.stock < quantity:
        raise InsufficientStockError(product_id, quantity, product.stock)
    product.stock -= quantity
    return product
```

4.6.3防御性编程

可以将防御性编程写入规则当中，实例如下：

```
---
alwaysApply: true
---

## 防御性编程规范
- 所有接受外部输入的函数必须在函数体开头对输入进行有效性校验；
- 禁止except Exception: 或裸except:，必须捕获具体异常类型；
- 调用第三方HTTP服务必须设置timeout（httpx 推荐 timeout=10.0）；
- 所有外部IO操作必须使用上下文管理器；
- 涉及数值边界或唯一性保证的写操作，使用数据库约束或行级锁。
```

Part 07

综合案例：待办事项应用

4.7综合案例：待办事项应用

项目背景与目标

预备知识：FastAPI与相关工具

第一步：项目初始化

第二步：后端API开发

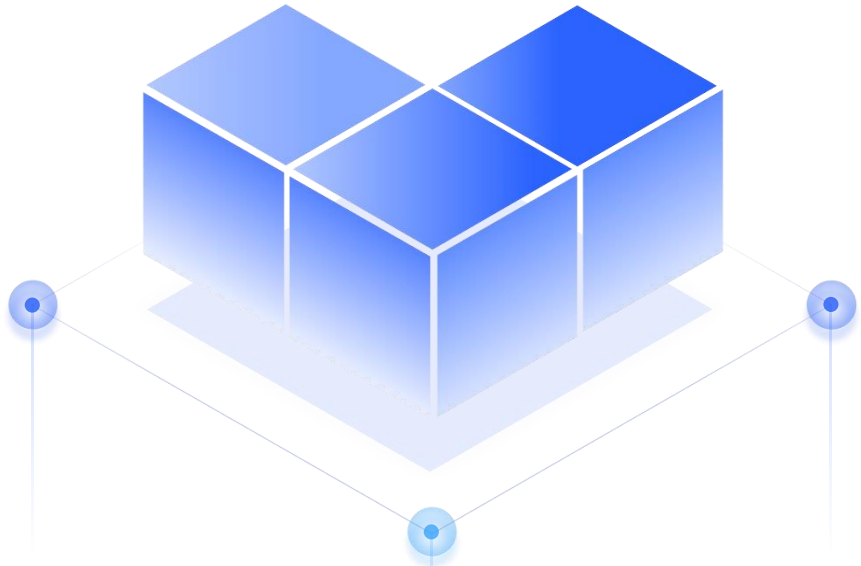
第三步：前端页面开发

第四步：运行应用

完整工作流回顾

4.7.1项目背景与目标

现在要开发一个极简的待办事项管理应用，仅包含三个核心操作：



新增待办

输入标题，添加一条待办事项。

标记完成

点击复选框，切换待办事项的完成状态。

查询待办

以列表形式展示所有待办事项。

4.7.1项目背景与目标

采用的技术栈如下：

- Python 3.12 + FastAPI（后端API框架）。
- 原生HTML/CSS/JavaScript（前端页面，单文件实现）。
- 数据存储在内存中（Python list），程序重启后数据清空。

需要说明的是，这里为什么选择内存存储？因为本案例的目标是展示AI编程工作流本身，而非数据库工程。内存存储让读者专注于CRUD逻辑和前后端交互的核心概念，无需在数据库配置上耗费额外精力。

4.7.2预备知识：FastAPI与相关工具

本案例使用FastAPI构建后端API。在开始之前，读者需要了解以下三个基本概念：

FastAPI

一个基于Python的现代Web框架，专为构建API而设计。它的核心特点是：使用Python类型注解自动处理请求参数验证，自动生成交互式API文档（访问 /docs 即可查看），以及原生支持异步编程。FastAPI是目前AI编程支持最好的Python Web框架之一。

Pydantic

FastAPI的底层数据验证库。开发者用Python类型注解定义数据模型（如 title: str），Pydantic自动完成请求数据的类型校验、格式转换和错误提示。在本案例中，Pydantic用于定义待办事项的数据结构。

Uvicorn

FastAPI应用的运行服务器。它是一个轻量级的ASGI服务器，负责接收HTTP请求并将其转发给FastAPI应用处理。开发者只需在终端执行“uvicorn main:app --reload ”即可启动服务，其中，“--reload ”表示代码修改后自动重启。

这三个工具的关系是：**FastAPI定义API逻辑，Pydantic确保数据格式正确，Uvicorn让应用跑起来**。读者无需深入掌握每个工具的细节，只需按照后续步骤操作即可。

4.7.3第一步：项目初始化

在向AI发出编程请求之前，先创建项目基础设施，即告诉AI项目背景的项目说明书（AGENTS.md）和告诉AI如何写代码的规则文件（Rules）。

在TRAE中新建一个项目，项目名称为“ToDoApp”，在项目根目录下创建AGENTS.md，填入如下内容：

```
# TodoApp - 待办事项管理应用

## 项目简介
学习用的待办事项管理应用，包含FastAPI后端和原生HTML前端。
数据存储在内存中（Python list），适合快速原型验证。

## 技术栈
- Python 3.12 + FastAPI + Pydantic v2
- 原生HTML/CSS/JavaScript（无框架依赖）
- 数据存储：内存中的Python list
```

4.7.3第一步：项目初始化

```
## 项目结构
- main.py：FastAPI应用入口，包含内存存储和所有API路由
- schemas.py：Pydantic数据模型
- templates/index.html：前端单页应用

## 架构约定
- 后端只负责API响应，不含HTML渲染逻辑
- 前端通过fetch调用后端API
- 内存存储使用全局list，id 自增使用itertools.count

## 常用命令
- 安装依赖：pip install fastapi uvicorn pydantic
- 启动服务：uvicorn main:app --reload
- 访问前端：http://localhost:8000/
```

4.7.3第一步：项目初始化

在项目根目录下创建“.trae/rules”子目录，并在该目录下创建一个名为todo-style.md的规则文件，填入如下内容：

```
---
alwaysApply: true
---

# TodoApp编码规范

## API响应规范
- 创建成功：HTTP 201；更新成功：HTTP 200；
- 资源不存在：HTTP 404，body 格式 {"detail": "Todo not found"}

## 数据模型
- Todo使用Pydantic模型
- id为int，自增生成
- created_at使用datetime.now(timezone.utc)

## 前端规范
- 使用原生JavaScript，不引入任何前端框架
- 所有API调用使用fetch API
```

4.7.4第二步：后端API开发

在TRAE的智能体交互界面中选择“Agent模式”，将以下提示词输入对话框并提交，然后TRAE将会根据提示生成main.py和schemas.py文件，这两个文件不用用户写一行代码，全部由AI生成：

```
## 任务
实现TodoApp的后端API，包含新增、查询和标记完成三个接口。

## 技术栈
- Python 3.12 + FastAPI + Pydantic v2
- 数据存储在内存中（Python list），id 使用itertools.count自增

## 文件结构
1. `schemas.py`：定义TodoCreate和TodoResponse两个Pydantic模型
   - TodoCreate：title（str，必填）
   - TodoResponse：id（int）、title（str）、is_completed（bool）、created_at（datetime）
2. `main.py`：FastAPI应用入口，包含以下路由
```

4.7.4第二步：后端API开发

```
## 接口清单
| 方法 | 路径 | 说明 |
|------|------|------|
| GET | /todos | 获取所有待办，返回 {"todos": [...]} |
| POST | /todos | 新增待办，请求体 {"title": "xxx"}，返回 HTTP 201 |
| PUT | /todos/{id} | 更新待办，请求体 {"is_completed": true}，返回更新后的对象 |

## 约束
- 资源不存在时返回HTTP 404，body格式 {"detail": "Todo not found"}
- created_at使用datetime.now(timezone.utc)
- main.py需挂载templates目录，根路径"/" 返回index.html

## 验收标准
- [ ] POST /todos创建成功 → HTTP 201，返回完整Todo对象
- [ ] GET /todos → 返回所有待办列表
- [ ] PUT /todos/{id} 更新is_completed → 返回更新后的对象
- [ ] PUT /todos/999（不存在）→ HTTP 404
```

4.7.4第二步：后端API开发

AI生成代码后，会反馈给用户，它生成了哪些文件，有哪些接口，接口的验证结果，以及部分技术细节，便于用户进行确认。在这个过程中，可能会出现等待用户确认的界面（如图所示），只要点击“运行”即可。

4.7.5第三步：前端页面开发

将以下提示词输入TRAE的Agent模式的对话框中并提交，同样，不需要用户创建任何文件，编写任何代码，AI将会自动实现前端的功能：

```
## 任务
创建templates/index.html，为TodoApp提供可视化界面。

## 技术栈
- 原生HTML + CSS + JavaScript（单文件实现，无框架依赖）
- API调用使用原生fetch

## 页面结构
1. **标题**：顶部显示"待办事项管理"
2. **添加区域**：一个文本输入框 + "添加"按钮
3. **待办列表**：逐条显示所有待办事项，每条包含：
   - 标题文字
   - 完成复选框（点击后切换完成状态）
   - 已完成项显示删除线样式
```

4.7.5第三步：前端页面开发

```
## API调用说明
| 操作 | 方法 | 路径 | 说明 |
|------|------|------|------|
| 获取列表 | GET | /todos | 页面加载时调用 |
| 新增 | POST | /todos | body: {"title": "xxx"} |
| 切换状态 | PUT | /todos/{id} | body: {"is_completed": true/false} |

## 响应格式

// GET /todos响应
{"todos": [{"id": 1, "title": "完成任务", "is_completed": false, "created_at": "2025-01-15T10:30:00"}]}
// POST /todos响应
{"id": 1, "title": "完成任务", "is_completed": false, "created_at": "2025-01-15T10:30:00"}
// 错误响应
{"detail": "Todo not found"}
```

4.7.5第三步：前端页面开发

```
## CSS样式要求
- 整体浅色系，白色背景，灰色边框
- 主色调：浅蓝色（#4A90D9），仅用于按钮和标题
- 已完成项：浅灰色文字 + 删除线
- 布局：居中显示，最大宽度 600px
- 字体：无衬线字体（sans-serif）

## JavaScript功能要求
1. 页面加载时调用GET /todos 获取列表并渲染
2. 点击"添加"按钮，调用POST /todos，成功后刷新列表
3. 点击复选框，调用PUT /todos/{id}切换完成状态，成功后刷新列表
4. 所有操作失败时在控制台输出错误信息

## 验收标准
- [ ] 页面加载后显示待办列表
- [ ] 输入标题并点击添加，新待办出现在列表中
- [ ] 点击复选框，待办完成状态切换，文字样式变化
```

4.7.5第三步：前端页面开发

和后端功能一样，AI生成前端代码以后，会反馈给用户部分细节，如页面结构、CSS样式、JavaScript功能等，以便用户进行审查。

此外，后端和前端其实也可以一次性开发完成，不过基于实际项目的经验，一般小步推进，一次只做一件事，以免AI因为幻觉导致返工和质量不稳定。

4.7.6第四步：运行应用

后端和前端开发完成后，按以下步骤启动应用：

（1）安装依赖，可以在TRAE的Git Bash终端中执行如下命令：

```
pip install fastapi uvicorn pydantic
```

（2）启动服务，可以在TRAE的Git Bash终端中执行如下命令：

```
uvicorn main:app --reload
```

（3）访问应用，打开浏览器访问http://localhost:8000/，即可看到TodoApp界面。在输入框中输入待办标题，点击"添加"即可创建；点击待办项前的复选框，即可切换完成状态。

由于现在AI越来越智能，以上步骤很可能都不需要用户手工执行，编写完代码以后，AI将会根据项目说明书（AGENTS.md）中的运行方式，自动运行程序。如果缺少依赖，AI也会自动安装相关的依赖。这也是项目说明书的价值所在，只要给AI足够的上下文，随着大模型越来越聪明，它能够做的事情将会越来越多。

4.7.7完整工作流回顾

下图展示了本案例完整的AI编程工作流。

这个工作流的核心特点：项目说明书和规则是一次性投入，对全程生效；后端与前端分别用独立的提示词驱动；每一步之后都有人工审查环节，确保质量可控。

本案例的交付物具体如下：

后端API（main.py + schemas.py）

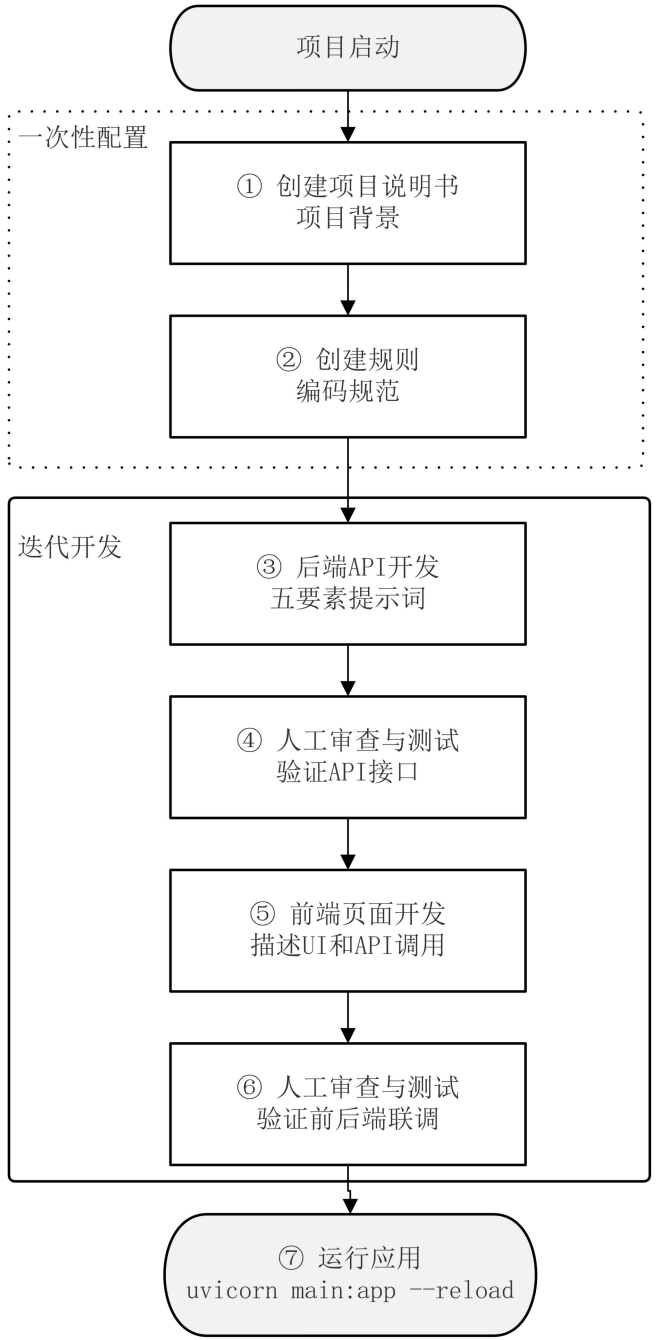
三个接口（新增、查询、标记完成）。

前端页面（templates/index.html）

可视化界面，支持添加和标记完成。

项目配置（项目说明书 + 规则）

可供复用的AI协作规范。



4.7.7完整工作流回顾

运行后，就可以看到一个简洁的待办事项管理页面，如图所示。

4.8本章小结

本章详细介绍了基于提示词的AI编程方法。核心思想是：AI的输出质量由输入质量决定。输入不完整会导致“语义漂移”——AI自动补全假设，使输出偏离真实意图。

01

提示词工程是关键。本章介绍了Google PTCF通用框架和专用于编程的六要素框架，强调提示词必须清晰、具体、可验证。

02

AI默认只处理常规路径，所有边界条件、异常处理和非功能性要求（如并发安全、幂等性）都必须在提示词中显式指明。为提升协作效率，本章介绍了项目说明书和规则两大项目级配置机制。

03

项目说明书用于定义项目背景、技术栈和架构约定，规则则将编码规范固化为AI可执行的硬性约束，确保生成的代码风格一致。

04

在存量项目维护中，总结了补充项目说明书、补充测试、补充规范和参考现有代码四大核心实践。其中“参考现有代码”（胶水编程）是让AI “照着做”的有效方法。

05

故障诊断依赖完整的可观测性数据（日志、指标、链路追踪）作为上下文。

06

最后，通过一个待办事项应用案例，综合演示了从项目配置、编写提示词到功能实现的完整工作流。AI改变了代码实现方式，但软件工程的核心原则——清晰的需求、明确的边界和可测试的验收标准，始终不变。

谢谢！

林子雨　副教授　厦门大学

附录A：主讲教师林子雨简介

主讲教师：林子雨

单位：厦门大学计算机科学与技术系

E-mail: ziyulin@xmu.edu.cn

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

数据库实验室网站：http://dblab.xmu.edu.cn

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

附录B：大数据学习路线图

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

附录C：林子雨大数据系列教材

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

附录D：林子雨人工智能通识系列教材

厦门大学林子雨人工智能通识系列教材

被众多高校采用，满足不同高校差异化教学需求

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

附录E：林子雨Python系列教材

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

谢谢！

林子雨　副教授

厦门大学