AI Coding 课程讲稿（精简版 · Markdown）

> 主题：从定义、方法论到工具与实战 —— 一套可落地的 AI 辅助编程实践路径
> 适用：公司内部培训 / 社区分享 / 工作坊

第 0 章 · 导入：这门课要解决什么问题?

> 核心：为什么现在要认真学 “AI 辅助编程”？听完课程可以带走什么?

大家好，今天这门课我们聚焦一个很具体的问题——**AI Coding（AI 辅助编程）**，也就是：

> 在真实的软件工程环境里，人类工程师和大模型怎么高效协作，既提效，又不放大技术债和风险?

很多人已经在用 ChatGPT、Claude、Cursor、Copilot 等工具，但大多是“随缘使用”：

- 有人离不开 AI，什么都丢给模型；
- 有人只当它是智能补全；
- 有人试过几次觉得“不靠谱”，干脆不用。

今天这门课不讨论“AI 会不会取代程序员”这种宏大命题，而是回答一个务实问题：

> **在我们现有的项目和团队环境里，如何系统性地使用 AI 来写代码?**

课程结束，希望你能带走两样东西：

1. 一套清晰的 **AI Coding 认知框架**（定义、边界、局限）。

2. 一套可以直接照着实践的 **实战流程 + Prompt 模板**，可以在自己的项目里小规模落地。

第 1 章 · 什么是 AI Coding：定义与边界

1.1 定义（课程采用的工作定义）

AI Coding（AI 辅助编程） 指的是：

- 使用以大模型为代表的 AI 工具，
- 在需求理解、架构设计、代码编写、重构、测试、文档等多个环节中，
- 将 AI 视作一个 **协作开发者（pair programmer）**，
- 通过多轮协作和工程流程，把 AI 产出整合进我们的软件系统，
- 最终由 **人类工程师对架构、关键实现和质量负责**。

可以记一句简化版：

> **不把 AI 当“搜索引擎加强版”，而是当一个真实的搭档，并纳入工程流程。**

1.2 AI Coding 不等于什么?

为了统一预期，需要先澄清一些误解：

1. **不等于甩锅**
   不是“把活全丢给 AI、不看结果”，那叫甩锅，不叫协作。
2. **不等于偶尔问两句**
   偶尔在 ChatGPT 问法条、查 API，更像“增强搜索”，还算不上系统性的 AI Coding。
3. **不等于自动替代程序员**
   当前现实反而是：
   - 会用 AI 的工程师，被成倍放大生产力；
   - 不会用 AI 的工程师，可能慢慢感觉“写不过 AI”。
4. **不是银弹**
   - AI 不会自动理解你的业务上下文、组织约束、历史坑；

- 不会自动帮你处理好安全、性能、可维护性这些工程问题。

> 可以把 AI 看作一个很强的“放大器”：
> 你的方法好、工程体系稳，它帮你放大优势；
> 方法乱、工程薄弱，它也会放大混乱和风险。

第 2 章 · AI Coding 的发展历程（站在时间轴上看现在所处的位置）

2.1 从传统工具到多 Agent 协作

从工具形态看，AI Coding 大致经历了几个阶段：

1. **阶段 0：传统智能开发工具**
   - IDE 自动补全、重构工具、静态分析、安全扫描等。
   - 没有大模型，但已经在 “辅助编程”。
2. **阶段 1：基于 ML 的代码补全**
   - 例如早期 TabNine，基于语言模型做“下一 token 预测”。
   - 局限在单文件、本地上下文。
3. **阶段 2：大模型代码生成 + IDE 插件**
   - 代表：GitHub Copilot、早期 ChatGPT + 复制粘贴。
   - 能够根据自然语言描述生成函数、类、测试、文档。
4. **阶段 3：项目级理解 + 多轮协作**
   - 工具可以读取整个仓库，多文件、项目级理解。
   - 代表：Cursor、Claude Code 等 AI IDE / Agent 工具。
5. **阶段 4：流程化 & 多 Agent（正在形成中）**
   - AI 进入 Issue、PR、CI/CD、代码库治理等环节；
   - MCP 等协议让 AI 可以安全调用文件系统、数据库、内部系统；
   - 一个任务可以由多个 Agent 分工完成。

> Claude Code、Cursor 等工具，已经站在第 3 → 4 阶段交界：
> 既能读项目、写代码，又开始能“连工具、跑流程、协调多个子 Agent”。

2.2 模型能力演进的关键点

- **上下文范围变大**：
  从几百行到几万 token，模型开始有能力“读懂”一个中小型服务。
- **从“写一段代码”到“理解/改造一个模块”**：
  可以帮你分析遗留系统、找耦合点、提出重构方案。
- **环境感知**：
  通过 MCP 等机制调用命令行、查看日志、访问 API，变成“在环境中工作”的 Agent，而不是只能看静态文本。

> 对开发者的影响是：
> **工作重点从“亲手写每一行代码”，转向“设计、拆解任务、评审和把关”。**

第 3 章 · 从哪些维度理解 AI Coding?

可以用一个简单的“四象限视角”来理解：

1. **工具维度**：AI 以什么形态出现？（IDE 插件 / AI IDE / CLI / CI 插件…）
2. **模型维度**：用什么模型？上下文多大？能不能调用外部工具？
3. **方法论维度**：是“想到什么就问什么”，还是有系统工作流（如 SDD）？
4. **工程维度**：AI 产出的东西如何安全地进入主干分支？

如果只看其中一个维度，很容易走向极端：

- 只看模型 → 感觉“无所不能”；
- 只看工程风险 → 感觉“完全不能用”。

我们真正希望的是：

> **工具、模型、方法论、工程四个维度配齐，**
> **才能让 AI Coding 稳定地产生业务价值。**

3.1 工具维度：AI 在开发流程的什么位置出现?

常见形态：

- **IDE / 编辑器插件**
  - 各类 Copilot，行级补全/片段生成。
  - 优点：零门槛，工作流变化小。
  - 适合：日常写码、局部问题。
- **专用 AI IDE（如 Cursor）**
  - 深度集成项目视图、Agent 面板、内置浏览器等。
  - 适合：前端、多文件协作开发。
- **命令行 / CLI 工具（如 Claude Code）**
  - 通过命令操作仓库、运行脚本、生成 patch。
  - 适合：后端、DevOps、自动化改造任务。
- **流程工具（Issue / PR / CI 集成）**
  - AI 帮你写 PR 描述、做初步 review、生成变更摘要。

> 问自己：在团队中，希望 AI 只出现在编辑器里，还是渗透到整个工程流程?

3.2 方法论维度：从 “vibe coding” 到 SDD

- **vibe coding 模式**：
  - 想到什么就丢给 AI；
  - 出 bug 再让 AI 修；
  - 过程不可复现、团队难协作。
- **Spec-Driven Development（SDD）模式**：
  - 先把需求写成结构化 Spec / 设计文档；
  - 以 Spec 为“契约”，AI 按 Spec 输出实现、测试和文档；

- Spec 本身可版本化和 review，是协作基准。

> 在课程后面的 Todo API 实战里，我们会走一遍 **SDD 工作流**：
> 从自然语言需求 → Spec → 项目结构 → 实现 → 测试 → 文档 → 复盘。

3.3 工程维度：下限由工程体系决定

- 如果只引入 AI，不配套工程约束，很容易出现：
  - 代码质量不稳定、风格不统一；
  - 技术债堆积更快（AI 写得太快，人来不及重构）；
  - 安全、合规问题更隐蔽。
- 需要配套：
  - 代码规范、目录约定；
  - 测试覆盖策略；
  - Review & 安全审计；
  - 写给 AI 的规则文件：`AGENTS.md`、`.cursor/rules`、`CLAUDE.md` 等。

> 简单一句：
> **AI 的上限由工具和模型决定，下限由团队的工程实践兜底。**

第 4 章 · 限制与问题 & 解决方案

本章聚焦四个字：**“哪里会翻车？”**

4.1 模型局限

- **幻觉与错误**：
  安全、边界条件、复杂业务规则上尤为明显。
- **过度自信**：
  即使“不确定”，也会给出一本正经的答案。
- **上下文选择性注意**：

有大窗口 ≠ 真看完，它会重点关注自己“觉得相关”的部分。

4.2 工程风险

- 代码质量、风格、结构不统一；
- 技术债放大：快写、慢重构；
- 安全问题：依赖选择随意、容易产生经典漏洞模式。

4.3 人的误区

- 过度依赖：什么都交给 AI，自身技能退化；
- 期待值失衡：以为“一键生成完美系统”，现实是多轮协作；
- 团队缺乏共识：每个人用法不同，项目变成“风格拼盘”。

4.4 解决思路：把 AI 当“勤奋实习生”

可以建立几个兜底策略：

1. **测试前置**
    - 先让 AI 帮你列测试场景，再写测试代码，最后是实现。
2. **刻意要求边界条件**
    - Prompt 中明确要求：边界情况列表 + 对应的测试或分支逻辑。
3. **多视角检查**
    - 关键代码用多模型 + 静态分析工具交叉核对（例如：一个模型写实现，另一个模型做审查）。
4. **结构与规范前置**
    - 人先决定模块划分、命名约定、目录结构；
    - AI 在既定结构内填充实现。
5. **安全敏感区域仍由经验丰富的工程师主导**
    - AI 可参与重构和 review 草稿，但不能“放养”。

第 5 章 · 工具实战：Claude Code & Cursor

> 本章给一个现实问题的答案：
> **“工具这么多，我该重点关注谁？怎么组合使用？”**

5.1 Claude Code vs Cursor：定位

- **Claude Code**
  - 终端 / CLI 优先工作流；
  - 能读项目、改文件、跑命令；
  - 深度集成 MCP、Skills、Subagents 等能力；
  - 更像一个“AI 程序员 + 自动化工作流平台”。
- **Cursor**
  - 基于 VS Code 的 AI IDE；
  - 提供智能补全、项目级 Agent、内置浏览器、Bugbot 等；
  - 对前端、本地开发体验极好。

> 总结：
> **Cursor = 今天最好用之一的 AI IDE**
> **Claude Code = 面向未来的多 Agent + 工具编排平台**

5.2 Claude Code 的典型场景

1. **项目级理解 & 导航**
   - 总结 `src/` 下各模块职责；
   - 分析某个复杂模块的边界和依赖；
   - 追踪一个函数的全局调用链。
2. **功能开发 / 修 bug / 重构**
   - 根据任务描述生成 patch（包含说明 + 修改点）；
   - 把“屎山函数”拆分重构；
   - 从 callback 风格迁移到 async/await 等。

3. **测试 / 文档 / 脚本**
    - 从现有代码生成单测、集成测试；
    - 自动生成 README / API 文档草稿；
    - 生成数据迁移脚本、运维脚本。
4. **代码阅读与学习**
    - 帮新人制定“阅读路线图”；
    - 用自然语言解释复杂业务逻辑。

5.3 Claude Code 的高级能力：MCP / Skills / Subagents

- **MCP（Model Context Protocol）**
    - 给 AI 插上标准化“USB-C 接口”；
    - 可安全接入文件系统、数据库、HTTP API、内部系统。
- **Skills（技能包）**
    - 把某类任务的工作流固化下来，如：写单测、生成 Release Note；
    - 多处复用：Claude Code、Claude 网页、Agent SDK 等。
- **Subagents（子代理）**
    - 同一任务中开多个“Claude 小号”并行协作：
        - 一个写实现、一个写测试、一个写文档、一个做审查。

> 一整套组合起来，更像一个可编排的 **AI 开发平台**，而不是单一工具。

5.4 Cursor 的几个关键点（简述）

- `.cursor/rules` / `AGENTS.md`：
    - 写给 AI / 新人的项目规范与背景说明。
- 内置浏览器 / WebDev：
    - 对前端开发体验极佳。
- Bugbot / CLI 集成：
    - 参与 PR review 和 CI 自动检查。

第 6 章 · 端到端案例：Todo API（SDD 工作流示例）

> 用一个简单但完整的 Todo API 服务，把前面讲的定义、方法论、工具和工程实践串起来。

6.1 案例设定

- 功能：
  - 创建 / 查询 / 更新 / 删除 Todo；
  - 按状态过滤，简单分页。
- 技术栈（可按团队替换）：
  - 示例使用 Node.js + TypeScript + Express + Jest。

> 重点不是写一个“完美 Todo 服务”，而是演示：
> **需求 → Spec → 结构 → 实现 → 测试 → 文档 → 复盘** 的 AI 协作流程。

6.2 阶段一：需求整理 & Spec（SDD 核心）

1. 用自然语言描述业务需求；
2. 让 AI 把需求整理为结构化 Spec（数据模型、接口、状态码、边界情况）；
3. 人工审核和修改 Spec，形成“可版本化的契约”。

6.3 阶段二：项目结构 & 技术栈决策

- 人：决定技术栈与整体目录结构（例如 `routes/`，`services/`，`repositories/` 等）；
- AI：生成初始化命令、`tsconfig` / `jest.config`、基础目录和入口文件。

> 工程骨架、关键约定由人拍板，AI 负责“落地执行”。

6.4 阶段三：实现接口（以 `POST /todos` 为例）

- 通过 Claude Code 下达任务：
  - 按 Spec 实现 `POST /todos`；
  - 限定只修改特定目录；
  - 禁止新增多余依赖；
  - 输出 patch + 设计说明。
- 人关注的点：
  - 是否遵守 Spec；
  - 是否符合项目约定；
  - 有无明显设计漏洞。

6.5 阶段四：测试优先 & 迭代修复

1. 让 AI 列出测试场景（正常、缺字段、非法状态等）；
2. 根据场景生成 Jest 测试文件；
3. 本地跑测试，接受失败输出；
4. 把失败日志和代码丢回 Claude Code，让其分析并给出最小修复 patch。

> 通过这个过程强调：
> **AI 写代码可以，但最终对错由测试和工程体系说话。**

6.6 阶段五：文档 & 脚本

- 让 AI 根据当前代码和 Spec 生成 README：
  - 项目简介、启动步骤、API 文档、示例请求。
- 生成常用脚本：
  - 启动脚本、简单 Dockerfile 等。

> 文档和脚本对团队协作非常重要，而 AI 在这部分的性价比极高。

6.7 阶段六：复盘——人和 AI 的职责边界

- **人类工程师负责**：
  - 需求澄清、Spec 最终版；
  - 技术栈、项目结构、规范；
  - 关键设计决策与最终 Review；
  - 测试策略与安全要求。
- **AI 主要负责**：
  - 整理需求 → Spec 草稿；
  - 生成项目骨架、实现代码、测试与文档草稿；
  - 分析测试失败并提出修复方案；
  - 在整个过程中充当“非常勤奋的 Pair Programmer”。

> 核心结论：
> **AI Coding 的关键不在于“AI 写了多少代码”，**
> **而在于你是否拥有一条可重复、可演进的协作流程。**

第 7 章 · 未来趋势（简要展望）

7.1 工具与平台

- 本地 / 私有模型会在安全敏感场景下普及；
- IDE 中的 AI 能力会从“插件”变成“默认标配”；
- MCP 之类协议成熟后，会出现大量可直接复用的连接器（MCP servers）。

7.2 开发模式与角色变化

- Spec-Driven Development、Agent 协作、工作流编排会更加常见；
- 工程师角色从“写每一行代码”，转向：
  - 设计系统、拆解任务、管理 Agent 与工具。

> 因此：
> **架构能力、抽象能力、沟通协作能力的相对重要性会持续上升。**

第 8 章 · 行动建议与资源

8.1 个人层面

- 选一个熟悉技术栈的“小项目”，照着 Todo API 流程完整走一遍；
- 刻意记录：
  - 哪些场景 AI 好用；
  - 哪些场景 AI 易踩坑。

8.2 小组层面

- 选一个风险较低的内部项目 / 工具，尝试用 AI 完成一次迭代；
- 把过程中用到的 Prompt、规范、踩坑记录下来，沉淀为组内的 **AI Coding 使用手册**。

8.3 团队层面

- 逐步引入：
  - `AGENTS.md` / `.cursor/rules` / `CLAUDE.md` 等规则文件；
  - 统一的 Skills 和 MCP 接入规范。
- 把“写给 AI 的说明书”和“写给新人看的 README”统一起来。

> 目标不是“大家都在用 AI”，
> 而是**团队拥有一套可复用、可演化的 AI Coding 能力**。

完