AI Coding 入门：从编程进化到 AI 大脑

课程逐字稿（讲师讲解词）

第一章

共 52 页幻灯片

第 1 页

【幻灯片标题】课程封面

【逐字稿】

同学们好，欢迎来到《AI Coding 入门：从编程进化到 AI 大脑》课程。我是本课程的讲师，非常高兴能和大家一起开启这段 AI 编程的探索之旅。在正式开始之前，我想先问大家一个问题：你们有没有想过，为什么今天我们不需要深入学习编程语法，也能通过自然语言让 AI 帮我们写出完整的程序？这背后到底发生了什么？这门课程将带你从编程语言的历史演变出发，一步步理解 AI 如何改变了我们写代码的方式，最终帮助你建立起完整的 AI 编程知识体系。本课程共分为四大板块，我们将从编程世界的三次革命讲起，再到 Vibe Coding 和 SDD 两种开发模式，最后深入大模型的底层原理。让我们正式开始吧！

第 2 页

【幻灯片标题】前言

【逐字稿】

本章作为全课程的开篇，主要讲述编程三次技术变革的脉络。我们会从最早的汇编语言讲起，一路走到今天的自然语言编程时代。在这个过程中，我们会重点详解两种新兴的开发模式——「Vibe Coding」和「SDD 规格驱动开发」，分析它们各自的优势与适用场景。同时，我们还会探究大模型的演进历程，理解 LLM 是如何落地赋能 AI 编程的。这一章的目标是帮大家建立起对 AI 编程的全局认知，后续章节会在此基础上深入每个技术细节。所以请大家认真听，跟着我的节奏走，一步步搭建知识框架。

第 3 页

【幻灯片标题】课程目标

【逐字稿】

我们来看本章的学习目标。学完这堂课之后，大家将能够做到以下四点：第一，梳理编程三次迭代变革的脉络，理清编程范式的演进历史，知道每一次变革解决了什么问题、又带来了什么新挑战。第二，了解 Vibe Coding 这种编程思路的核心原理，客观辨识它的优势与潜在隐患，不会盲目追捧也不会一味排斥。第三，了解 SDD 规格驱动开发的理念，知道它和 Vibe Coding 之间的对比关系与互补价值。第四，熟知大模型的演进历程，理解 LLM 落地赋能 AI 编程的底层逻辑，明白 AI 为什么能听懂人话并生成代码。这四个目标层层递进，从历史到现状，从方法到原理，帮助大家形成完整的知识闭环。

第 4 页

【幻灯片标题】目录

【逐字稿】

我们来看本章的目录。本章分为四个部分：第一部分是「编程世界的三次革命：迈入自然语言时代」，我们从历史维度理解编程语言如何一步步逼近人类语言。第二部分是「Vibe Coding：用直觉写代码的自由与风险」，讲解当下最火的 AI 编程方式。第三部分是「SDD：先写说明书，再让 AI 动工」，介绍一种更规范的 AI 开发方法论。第四部分是「LLM：从大模型的发展到现在的 AI 编程的引进过程」，深入技术底座，理解 AI 编程的原理。这四部分从历史到实践，从方法到底层，形成完整的知识链条。我们逐一展开。

第 5 页

【幻灯片标题】从机器思维到人类语言：编程的三次蜕变

【逐字稿】

让我们进入第一部分——编程的三次蜕变。这张幻灯片展示了从机器思维到人类语言的三个关键阶段。第一阶段是汇编语言时代，大约从 1947 年到 1950 年代，人类用汇编符号替代了二进制代码，但汇编语言仍然和硬件强绑定，换一台机器就得重写。第二阶段是高级语言时代，从 1957 年起，Fortran、C、Python 等语言出现，程序员终于脱离了硬件细节，可以专注于逻辑本身，代码也具备了可移植性。第三阶段就是我们现在所处的自然语言编程时代，从 2023 年至今，对话即代码，人机协作成为主流，开发者只需要定义需求。大家注意这条主线——抽象层级的不断跃升。每一次跃升，都让更多人能够参与到编程中来，一个「全民皆可创造」的时代正在到来。

第 6 页

【幻灯片标题】第一次革命：汇编语言的出现

【逐字稿】

我们来深入看第一次革命——汇编语言的出现。提到汇编语言，就不得不提到一位重要的女性科学家——凯瑟琳·布斯（Kathleen Booth），她是汇编语言概念的提出者之一。在那个年代，程序员必须直接用 0 和 1 编写机器码，比如要把数值 5 放入寄存器，需要写 10110000 这样的二进制序列；要让两个寄存器相加，需要写 00010100。这极其反人类，容易出错且难以阅读。汇编语言的出现改变了这一切——同样的操作，现在可以写成 MOV A, #5 和 ADD A, B。从「01 代码」到「英语指令」，这是人类首次成功「教会」机器理解抽象符号。虽然汇编语言仍然和硬件紧密绑定，不同 CPU 的汇编指令各不相同，

但这一步的意义是巨大的——它证明了机器指令可以用人类可读的符号来表示，为后续高级语言的诞生奠定了基础。

第 7 页

【幻灯片标题】第二次革命：高级语言与 IDE 的出现

【逐字稿】

接下来看第二次革命——高级语言与 IDE 的出现。高级编程语言最大的突破在于：它接近人类自然语言，独立于硬件，抽象度高且可移植性强。比如你用 Python 写的一段排序算法，可以在 Windows、Linux、Mac 上直接运行，不需要修改。不同的语言有不同的 IDE 生态：Python 有 PyCharm、VS Code、Spyder；Java 有 IntelliJ IDEA、Eclipse、NetBeans；C++有 Visual Studio、CLion；Go 有 GoLand 等。这里要特别讲一下 IDE——集成开发环境。IDE 是一站式开发工具，它把编辑器、编译器、调试器、项目管理工具集成在一起。在 IDE 里，你可以写代码、自动补全、一键编译、断点调试、管理依赖，所有操作在一个界面内完成。IDE 的出现极大提升了开发效率，让程序员从繁琐的工具切换中解放出来，专注于业务逻辑本身。这为第三次革命——自然语言编程——做好了工具层面的准备。

第 8 页

【幻灯片标题】第三次革命：自然语言编程

【逐字稿】

现在我们来到最激动人心的第三次革命——自然语言编程。这一次革命的核心理念是：开发者只需要输出自然语言，描述业务意图，AI 智能体就能执行全流程——从设计到

编码到实现。大家注意这张图上的角色变化：在汇编时代，人是硬件操控者；在高级语言时代，人是代码生产者；而在自然语言编程时代，人变成了系统价值架构师。LLM 在这里扮演的角色是「新型智能编译器」——以前的编译器把高级语言翻译成机器码，现在的 LLM 把自然语言翻译成可运行的代码。开发者不再需要手动编写每一行代码，不需要记忆语法细节，不需要查 API 文档，只需要用自然语言向 AI 智能体描述意图。这意味着编程的门槛被极大地降低了，非计算机专业的人也能通过 AI 来构建软件。当然，这并不意味着程序员会失业——恰恰相反，程序员的价值从「写代码」上升到了「定义需求、设计架构、审核结果」，对人的综合能力要求更高了。

第 9 页

【幻灯片标题】三代编程语言的对比

【逐字稿】

让我们通过这张对比表来深入理解三代编程语言的差异。从七个维度来对比：编程语言层面，从汇编到高级语言再到自然语言，越来越接近人类语言。核心活动层面，从操控硬件到编写代码再到描述意图，越来越抽象。核心技能层面，从理解硬件到掌握语法到表达需求，越来越偏向思维。抽象层级层面，从最低到中等到最高。人机关系层面，从人适应机器到人机器协作到机器适应人。效率瓶颈层面，从硬件限制到编码速度到需求表达精度。最关键的是工程师定位的变化——从「硬件操控者」到「代码生产者」再到「系统价值架构师」。大家想想，这个变化意味着什么？意味着未来的工程师最重要的能力不再是写代码的速度，而是理解业务、定义需求、把控全局的能力。代码生产这件事，正在被 AI 逐步接管。

第 10 页

【幻灯片标题】AI 编程工具的三代演进路径

【逐字稿】

讲完了编程语言的演进，我们来看 AI 编程工具自身的三代演进路径。第一代出现在 2023 年，是插件式代码补全，代表工具如 GitHub Copilot 的早期版本。这时的 AI 本质上是一个「超级输入法」——你写一半，它帮你补全另一半，理解粒度仅限于行或片段级别。第二代出现在 2024 到 2025 年，是 AI 原生 IDE 和对话式编程，代表工具如 Cursor、Windsurf。这时的 AI 升级为「副驾驶」——你可以用自然语言对话，让它生成整个文件或模块，理解粒度提升到了文件和模块级别。第三代从 2025 年至今，是自主智能体，代表工具如 CodeArts、Claude Code 等。这时的 AI 已经进化为「工程代理」——你给它一个目标，它就能自主规划、编码、测试、修复，理解粒度扩展到了全项目和业务级别。大家注意理解粒度的变化：从行/片段到文件/模块再到全项目/业务，AI 的理解范围在不断扩大，自主能力在不断增强。

第 11 页

【幻灯片标题】AI 编程工具：第一代&第二代

【逐字稿】

我们来展开看第一代和第二代 AI 编程工具。第一代是代码补全助手，出现在 2023 年以前。它的工作方式是作为 IDE 的附属插件，你在写代码时按 Tab 键，AI 帮你补全当前行或下一段代码。它的局限性很明显——仅限单文件内理解，无法跨文件感知项目上下文，生成的代码往往是局部的、碎片化的。第二代是 AI-人协同模式，出现在 2024 到 2025 年。代表特征是拥有独立的开发环境，支持自然语言对话，能感知全局项目上下文。工作流程

变成了：人提需求→AI 实现→人工审核优化。你可以说「帮我实现一个登录功能」，AI 就能生成完整的前后端代码。但注意，这时的 AI 仍然是「副驾驶」角色——它帮你写，但你来审核和决策。从第一代到第二代，最大的进步是从被动补全变成了主动生成，从单文件理解变成了全局项目感知。

第 12 页

【幻灯片标题】AI 编程工具：第三代：自主智能体代理

【逐字稿】

现在来看第三代——自主智能体代理，这是当前最前沿的形态。从 2025 年至今，AI 编程工具的定位发生了质变——从「副驾驶」升级为可以独立交付的「工程师」。第三代的核心特征有三个：第一，目标驱动——你只需要给出目标，比如「开发一个校园美食推荐应用」，AI 就能自主分解任务、规划步骤。第二，全链路自主——从需求分析到架构设计到编码实现到测试验证，AI 全程自主执行，不需要你逐步指导。第三，多智能体协作——复杂的任务可以由多个专业化的 Agent 分工协作，比如一个负责前端，一个负责后端，一个负责测试。工作流程变成了：人定目标→AI 自主执行→人验收。大家注意，人的角色进一步后退——从「审核每一行代码」变成了「验收最终交付物」。这并不意味着人可以完全不管，而是人的精力从执行层面转移到了目标定义和结果验收层面，这对人的系统思维和判断能力提出了更高要求。

第 13 页

【幻灯片标题】软件开发的演进

【逐字稿】

让我们从更高维度来看软件开发的演进历程。整体分为三个阶段：第一阶段是传统人工开发阶段，2024 年以前，纯人工主导，开发者从需求到编码到测试全程手动完成。第二阶段是 AI 辅助开发阶段，2024 到 2025 年，人为主导，AI 做辅助——人写主要逻辑，AI 帮忙补全、优化、生成测试用例。第三阶段是 AI 自主智能开发阶段，2025 年至今，目标驱动，AI 全链路自主执行——人只需要给出目标和约束，AI 自主完成开发。这个演进带来的最根本变化是：从传统的 SDLC（软件开发生命周期）向 ADLC（智能体开发生命周期）转变。接下来两张幻灯片我们会分别详细讲解 SDLC 和 ADLC 的具体流程，帮助大家理解这个转变的深层含义。

第 14 页

【幻灯片标题】传统 SDLC：软件开发生命周期

【逐字稿】

我们先来看传统的 SDLC——软件开发生命周期。这是一套由人类主导的、确定性的、阶段式的软件开发标准流程。标准流程分为六步：需求分析→系统设计→开发编码→集成测试→部署上线→运维维护。每一步都有明确的输入和输出，前一步的输出是后一步的输入。SDLC 有几个核心特征：第一，线性推进——按顺序一步步走，不能跳步。第二，阶段不可逆——进入测试阶段后，如果发现设计问题，需要走变更流程回到设计阶段，代价很大。第三，文档为王——每个阶段都要产出详细文档，文档是团队协作的契约。第四，人全程主导——从需求到运维，每个环节都需要人深度参与。SDLC 在过去几十年被证明是可靠的、成熟的，但它也有明显的局限：流程重、周期长、响应变化慢。在 AI 时代，我们需要一种更敏捷、更迭代的开发范式，这就是 ADLC。

第 15 页

【幻灯片标题】ADLC：智能体开发生命周期

【逐字稿】

现在来看 ADLC——智能体开发生命周期。这是面向 AI 智能体和大模型驱动系统的新型开发范式。核心流程分为五步加一个闭环：目标锚定→智能体设计→能力构建→仿真评估→动态治理，再加上持续反馈闭环。我们逐一解释：目标锚定，就是明确定义系统要解决什么问题、达成什么目标。智能体设计，就是设计 AI Agent 的角色、能力边界、协作方式。能力构建，就是为 Agent 配备工具、知识库、MCP 服务等。仿真评估，就是在沙盒环境中测试 Agent 的表现。动态治理，就是在运行中持续监控和调整。最关键的是那个持续反馈闭环——系统不是一次开发完就结束，而是在运行中不断收集反馈、持续进化。ADLC 的核心理念是：人定目标、AI 全链路执行、人做决策与风控、系统持续进化。对比 SDLC 的线性推进，ADLC 是闭环迭代的；对比 SDLC 的确定性，ADLC 是概率性的——因为 AI 的输出不是 100%确定的。这种范式特别适合需求快速变化、需要持续进化的智能系统。

第 16 页

【幻灯片标题】SDLC vs ADLC：核心对比

【逐字稿】

让我们通过核心对比来加深理解。从四个维度来看 SDLC 和 ADLC 的区别：第一，核心驱动——SDLC 是人主导的代码驱动，每一行代码都是人写的；ADLC 是目标驱动的 AI 智能体主导，人只定义目标。第二，行为特性——SDLC 是确定性的，同样的输入必然得到同样的输出；ADLC 是概率性的，AI 可能给出不同的实现方案。第三，流程模式——

SDLC 是线性瀑布式，一步步推进；ADLC 是闭环迭代式，持续反馈持续优化。第四，适用系统——SDLC 适用于需求稳定的传统软件，比如银行核心系统；ADLC 适用于动态复杂的 AI 智能体系统，比如智能客服、个性化推荐。大家不要理解为 ADLC 要取代 SDLC——两者是互补的。传统业务系统仍然适合 SDLC，而 AI 驱动的智能系统则需要 ADLC。在实际项目中，往往是两者混合使用——核心逻辑用 SDLC 保证稳定性，智能模块用 ADLC 保证灵活性。

第 17 页

【幻灯片标题】研发新范式

【逐字稿】

我们来看研发新范式的全景图。从编程语言的角度，经历了三个阶段：机器码和汇编时代追求极致性能，高级语言时代追求效率平衡，自然语言编程 AIGC 时代追求极低门槛。每个阶段都有自己的适用场景，不是越新越好。未来的趋势是什么？是「底层硬核+上层柔性」的深度分层协作。什么意思呢？底层——操作系统、编译器、数据库内核等——仍然需要硬核的 C/C++和汇编来保证性能和可靠性。上层——业务应用、前端页面、数据分析等——可以用自然语言编程来快速交付。回到 SDLC 和 ADLC 的选择：SDLC 适用于传统软件开发，需求明确且稳定；ADLC 适用于目标驱动、快速开发、持续进化的场景。在实际工作中，大家要根据项目特点选择合适的范式，而不是盲目追新。好，第一部分编程三次革命就讲到这里，接下来我们进入第二部分——Vibe Coding。

第 18 页

【幻灯片标题】目录

【逐字稿】

我们进入第二部分——Vibe Coding。在讲具体概念之前，先回顾一下前面的内容：我们了解了编程语言的三次革命，知道了 AI 编程工具的三代演进。现在我们来深入一种具体的 AI 编程方式——Vibe Coding。这部分我们会讲清楚四个问题：什么是 Vibe Coding？它能做什么、不能做什么？它有什么风险？以及如何高效使用它？通过这部分的学习，大家将对 Vibe Coding 有一个全面、客观的认识，既知道它的威力，也了解它的局限。

第 19 页

【幻灯片标题】了解什么是 Vibe Coding

【逐字稿】

我们来正式定义 Vibe Coding。Vibe Coding，中文叫「氛围编程」，是一种 AI 原生的开发范式。它的核心理念是：开发者通过自然语言描述功能意图或「氛围」，由 AI 大模型自动完成代码生成。注意这里的「氛围」二字——它强调的不是精确的技术规格，而是一种模糊的、直觉性的意图表达。工作流程很简洁：意图描述→AI 引擎→产出可运行的应用程序。你可以对 AI 说「帮我做一个看起来很酷的太空主题登录页面」，AI 就能生成完整的 HTML、CSS 和 JavaScript 代码。如果效果不满意，通过对话迭代调整——「颜色再深一点」「加个动画效果」「按钮改成圆角」。Vibe Coding 的精髓在于多轮交互快速调整，通过不断对话来逼近你想要的效果。它特别适合快速验证创意、做原型开发。但注意，Vibe Coding 产出的代码往往缺乏严谨的架构设计，适合做 Demo 但不适合直接用于生产环境。

第 20 页

【幻灯片标题】Vibe Coding 的诞生

【逐字稿】

Vibe Coding 这个概念是怎么诞生的呢？2025 年 2 月，由 Andrej Karpathy 在 X（原 Twitter）上发文正式提出。Karpathy 是 AI 领域的顶级大牛——他是 OpenAI 的联合创始人，也曾担任特斯拉的 AI 总监。他在推文中描述了一种全新的编程体验：你完全凭感觉、凭氛围跟 AI 对话，AI 帮你生成所有代码，你甚至不需要理解 AI 生成的每一行。这个概念一经提出就迅速走红——2 月提出概念，3 月 Merriam-Webster 词典就将其收录，4 月各大媒体集中报道，5 月各类工具平台纷纷跟进支持。为什么 Vibe Coding 能引起这么大的反响？因为它代表了一种根本性的转变——编程不再是少数人的专业技能，而变成了人人可用的创造工具。当然，Karpathy 自己也强调，Vibe Coding 适合快速原型和创意验证，不适合严肃的工程项目。这一点我们在后面的风险部分会详细讲。

第 21 页

【幻灯片标题】Vibe Coding 有什么用?

【逐字稿】

Vibe Coding 到底有什么用？核心价值就是 AI 驱动的意图式编程。用一个公式来表达：开发者的意图（Vibe）+ AI 的生成能力 = 极速落地的代码。具体来说有三大产出价值：第一，极速提效——原来需要一天才能写完的页面，现在几分钟就能生成，效率提升数倍甚至数十倍。第二，降低门槛——不懂编程的产品经理、设计师、运营人员，也能通过 Vibe Coding 快速做出可运行的 Demo 来验证想法。第三，全民创作——编程不再是程序员的专利，任何有创意的人都可以用 AI 把自己的想法变成产品。这三点合在一起，意味着 Vibe Coding 正在 democratizing software development——让软件开发民主化。但大家要清醒地认

识到，Vibe Coding 产出的代码在质量、架构、安全性方面往往存在隐患，它适合「从 0 到 1」的创意验证，但不适合「从 1 到 100」的工程迭代。

第 22 页

【幻灯片标题】Vibe Coding 的适用场景

【逐字稿】

我们来看 Vibe Coding 的适用和不适用场景。适用场景包括五类：第一，快速原型和创意验证——比如你想验证一个产品想法，用 Vibe Coding 半小时就能做出可交互的原型。第二，个人开发和学习练手——学新框架时，让 AI 帮你生成示例代码，比看文档快得多。第三，自动化脚本和小工具——比如写个批量重命名文件的脚本，直接跟 AI 说就行。第四，前端页面和可视化——HTML/CSS/JS 是 Vibe Coding 的强项，因为前端效果直观可感知。第五，应急开发和功能补全——项目紧急需要加个小功能，用 Vibe Coding 快速搞定。不适用场景也有三类：第一，高合规高安全类项目——比如医疗、金融系统，代码必须经过严格审计，不能容忍 AI 随意生成。第二，多人长期协作工程——Vibe Coding 产出的代码缺乏统一架构，多人协作会混乱。第三，大型核心业务系统——这类系统需要严谨的架构设计和严格的测试，Vibe Coding 无法保证。大家在使用 Vibe Coding 时，一定要先判断场景是否合适，不要拿着锤子看什么都是钉子。

第 23 页

【幻灯片标题】Vibe Coding 案例

【逐字稿】

我们来看一个具体的 Vibe Coding 案例。场景描述是这样的：我想快速开发一个西安三日游的前端网页，包含景点介绍、特色美食、推荐路线、酒店推荐等模块。用传统方式开发，你需要设计页面布局、写 HTML 结构、调 CSS 样式、写 JS 交互、准备数据，至少要半天时间。用 Vibe Coding 怎么做？你只需要对 AI 说：「帮我做一个西安三日游的前端网页，包含景点、美食、路线、酒店四个板块，风格要有西安古城的韵味，配色用暖色调，要有轮播图和卡片布局。」AI 就能一次性生成完整的页面代码，包含界面和数据。如果效果不满意，继续对话调整：「景点卡片加上评分星级」「路线板块改成时间轴样式」。几轮对话下来，一个完整的旅游网页就做好了。这就是 Vibe Coding 的威力——仅口头描述需求，AI 一次性完成界面和数据。但请注意，这个网页的代码质量、可维护性、性能优化都没有经过严格考量，适合做展示 Demo，如果要上线生产环境，还需要进一步打磨。

第 24 页

【幻灯片标题】Vibe Coding 的风险与局限

【逐字稿】

Vibe Coding 虽然好用，但也有明显的风险和局限，我们必须客观面对。第一，代码质量不可控——AI 生成的代码可能看起来能跑，但内部可能存在隐性问题，比如内存泄漏、性能瓶颈、不规范的写法。第二，架构混乱——Vibe Coding 是逐轮对话生成的，每一轮可能引入新的不一致，长期积累下来代码难以维护。第三，需求理解偏差——你用自然语言描述的需求，AI 的理解可能和你不同，生成的代码可能南辕北辙。第四，安全与合规风险——AI 可能生成包含安全漏洞的代码，比如 SQL 注入、XSS 攻击，也可能使用有版权争议的开源组件。这四个风险不是理论上的，而是实际项目中经常遇到的。所以大家在使用 Vibe Coding 时，一定要有风险意识——快速做原型可以，但不要直接把 Vibe Coding 产

出的代码推到生产环境。这就是为什么我们后面要讲 SDD——规格驱动开发，它就是来解决 Vibe Coding 这些问题的。

第 25 页

【幻灯片标题】如何高效使用 Vibe Coding?

【逐字稿】

那么如何高效使用 Vibe Coding 呢？我给大家一个四步方法论。第一步：先用 Vibe 探索创意——不要一上来就写详细规格，先凭直觉快速生成一个原型，看看方向对不对。这一步的目标是快速验证想法，不追求质量。第二步：梳理需求，提炼规格——原型确认方向后，回过头来梳理真正的需求，把模糊的「氛围」转化为清晰的「规格」。第三步：切换 SDD 规范开发——用提炼好的规格驱动 AI 编码，而不是继续凭感觉对话。这一步保证了代码的质量和可维护性。第四步：人工审核加测试验收——无论 AI 多强大，最终都需要人来审核代码逻辑、测试功能正确性。这四步方法论的核心思想是：Vibe Coding 和 SDD 不是对立的，而是互补的。Vibe Coding 负责「从 0 到 1」的快速探索，SDD 负责「从 1 到 100」的规范交付。先用 Vibe Coding 跑通想法，再用 SDD 规范落地，这才是 AI 编程的正确姿势。

第 26 页

【幻灯片标题】目录

【逐字稿】

我们进入第三部分——SDD 规格驱动开发。前面我们讲了 Vibe Coding 的威力和风险，现在来讲如何用 SDD 来解决 Vibe Coding 的问题。这部分同样会讲清楚四个问题：SDD 是

什么？它怎么来的？它有什么用？以及怎么用？通过这部分的学习，大家将掌握一种更规范、更可靠的 AI 编程方法论，知道在什么场景下应该用 SDD 而不是 Vibe Coding。

第 27 页

【幻灯片标题】SDD 是什么?

【逐字稿】

SDD 是什么？SDD 全称 Spec-Driven Development，规格驱动开发，是一种先写规范、后写代码的 AI 时代软件工程方法论。核心理念是四句话：规范先行、代码后至、规范是契约、代码是产物。什么意思呢？在 SDD 中，你首先花时间写一份详细的规格说明书（Spec），定义清楚系统要做什么、约束条件是什么、验收标准是什么。然后把这份 Spec 发给 AI，让 AI 严格按照 Spec 来生成代码。最后，用 Spec 来验收 AI 生成的代码是否合格。SDD 解决的核心问题有四个：第一，需求模糊——先写 Spec 逼着你把需求想清楚。第二，AI 局部正确全局错误——Spec 提供了全局约束，AI 不会跑偏。第三，协作低效——Spec 是团队共识，减少沟通成本。第四，质量滞后——Spec 定义了验收标准，质量可以前置检验。对比 Vibe Coding 的「凭感觉」，SDD 的「凭规格」更适合严肃的工程项目。

第 28 页

【幻灯片标题】SDD 怎么来的?

【逐字稿】

SDD 是怎么来的？它不是凭空出现的，而是软件工程方法论一路演进的结果。1980 年代，形式化规范出现，用数学方法严格定义系统行为。2000 年后，TDD（测试驱动开发）和 BDD（行为驱动开发）流行，强调先写测试再写代码。2023 到 2024 年，Vibe Coding

泛滥导致 AI 代码失控——代码质量差、架构混乱、无法维护。2025 年，SDD 作为新范式被提出，将传统的规格驱动思想与 AI 编程结合。我们来看三种驱动方式的对比：TDD 是测试驱动——先写测试用例，再写代码让测试通过。BDD 是行为驱动——先定义业务行为，再写代码实现行为。SDD 是规格驱动——先写完整规格，再让 AI 按规格生成代码。三者的共同点是「先想清楚再动手」，区别在于驱动物的粒度和形式不同。SDD 继承了 TDD 和 BDD 的思想，但更适配 AI 时代——因为 AI 可以帮你把 Spec 转化为代码，人只需要专注于定义 Spec。

第 29 页

【幻灯片标题】SDD 有什么用?

【逐字稿】

SDD 有什么用？我们来看五大核心价值。第一，约束 AI 编码，解决 Vibe Coding 乱码问题——有了 Spec 作为约束，AI 不能随意发挥，必须按规矩来。第二，统一协作标准，减少团队沟通内耗——Spec 是团队共识的载体，所有人围绕同一份 Spec 工作，不需要反复开会确认。第三，质量左移，提前规避 bug——传统方式是写完代码再测，发现 bug 再改；SDD 是在 Spec 阶段就把验收标准定义好，问题在早期就能发现。第四，项目易维护，方便迭代升级——Spec 本身就是最好的项目文档，新人来了看 Spec 就能理解项目。第五，提升落地效率，放大 AI 产能——有了清晰的 Spec，AI 的生成质量和成功率都大幅提升，减少返工。这五点合在一起，说明 SDD 不是增加负担，而是通过前期投入来减少后期的返工和维护成本。前期写 Spec 多花的时间，后期会数倍地省回来。

第 30 页

【幻灯片标题】SDD 怎么用

【逐字稿】

SDD 具体怎么用呢？流程很简洁，分四步。第一步，先写 Spec——把需求想清楚，写成结构化的规格文档。Spec 包含功能描述、约束条件、验收标准等，只写「做什么」不写「怎么写」。第二步，把 Spec 发给 AI——让 AI 阅读并理解 Spec，按照 Spec 的约束来生成代码。第三步，AI 按 Spec 写代码——AI 生成完整的代码实现，确保每一行代码都符合 Spec 的定义。第四步，最后按 Spec 验收——不是凭感觉看代码好不好，而是严格对照 Spec 的验收标准来检验。这个流程的核心原则有三条：先 Spec 后代码——顺序不能反；Spec 只写「做什么」不写「怎么写」——把实现细节交给 AI；验收只看 Spec 不看感觉——客观验收而非主观判断。这个方法论看起来简单，但真正执行起来需要 discipline——因为人很容易跳过写 Spec 直接开始编码，尤其是觉得需求很简单的时候。但恰恰是这种偷懒，导致了后期的返工和质量问题。

第 31 页

【幻灯片标题】企业和个人为什么需要 SDD

【逐字稿】

我们来看企业和个人为什么都需要 SDD。对于个人开发者来说，SDD 有三个好处：约束 AI 乱生成——有了 Spec，AI 不会天马行空；提升效率——Spec 让 AI 一次生成就能达标，减少多轮对话调整；控制代码、减少返工——Spec 定义了边界，不会做多了再做少了。个人用 SDD 的核心目的是「不返工」——一次做对，省时省力。对于企业团队来说，SDD 有三个好处：管控 AI 开发风险——Spec 是审核依据，管理者可以审查 Spec 是否合理；控风险控成本控质量——Spec 统一了标准，不同人用 AI 生成的代码风格一致；方便交接

和维护——Spec 就是文档，新人来了能快速上手。企业用 SDD 的核心目的是「不出事」——规模化开发中，没有规范必然出事。总结一下：个人用 SDD 为了效率，企业用 SDD 为了安全。但无论个人还是企业，SDD 的本质都是「先想清楚再动手」，这在任何工程领域都是不变的真理。

第 32 页

【幻灯片标题】SDD 与 Vibe Coding 的对比

【逐字稿】

让我们通过对比来深入理解 SDD 和 Vibe Coding 的区别。从四个维度来看：第一，开发顺序——Vibe Coding 是需求→prompt→直接生成代码，跳过了规格定义；SDD 是需求→编写 Spec→AI 编码→按 Spec 验收，多了规格和验收两个环节。第二，AI 约束力——Vibe Coding 对 AI 无约束，自由发挥，好处是灵活，坏处是容易跑偏；SDD 用 Spec 做契约，严格遵照，好处是可控，坏处是前期需要花时间写 Spec。第三，项目规模——Vibe Coding 适合小型 Demo，快速验证想法；SDD 适合整体项目和中大型企业项目，需要可维护可交接。第四，维护交接——Vibe Coding 无文档，代码就是全部，新人接手很痛苦；SDD 的 Spec 本身就是项目文档，交接时给 Spec 加代码即可。大家不要把两者看作对立的——它们是不同场景下的最优选择。快速验证用 Vibe Coding，规范交付用 SDD，这才是正确的认知。

第 33 页

【幻灯片标题】SDD + Vibe Coding：开发新范式

【逐字稿】

我们来看 SDD 和 Vibe Coding 如何搭配使用。用一个比喻：Vibe Coding 是「油门+提速引擎」——快速生成代码、提升开发效率；SDD 是「刹车+施工图纸」——定规范、控边界、防跑偏。只有油门没有刹车，开车会翻车；只有刹车没有油门，车走不动。二者搭配使用，才能「提速不翻车，开发又快又稳」。具体怎么搭配？在实际项目中，推荐的工作流是：先用 Vibe Coding 快速生成原型，验证想法是否可行。原型确认后，梳理需求写成 Spec。然后用 SDD 方式让 AI 按 Spec 重新生成或优化代码。最后按 Spec 验收，确保质量达标。这样既利用了 Vibe Coding 的快速探索能力，又利用了 SDD 的规范约束力。好，第三部分 SDD 就讲到这里，接下来我们进入第四部分——LLM 大模型。

第 34 页

【幻灯片标题】目录

【逐字稿】

我们进入第四部分——LLM 大模型。前面三部分我们讲了编程革命、Vibe Coding 和 SDD，都是从方法论层面讲的。现在我们下沉到技术底座，理解 AI 编程背后的核心引擎——大型语言模型 LLM。这部分会讲清楚 LLM 是什么、它的发展历程、它的底层原理，以及它如何赋能 AI 编程。理解了 LLM，你才能真正理解 AI 编程的能力边界和未来方向。

第 35 页

【幻灯片标题】实现 AI 编程：完整六步流程

【逐字稿】

我们先来看 AI 编程的完整六步流程，这把前面讲的所有内容串起来了。第一步，明确需求——想清楚要做什么，这是起点。第二步，挑选适配大模型——根据任务特点选择

合适的 LLM，代码生成选代码模型，通用对话选通用模型。第三步，构建知识库——给 AI 喂专业知识，让它了解你的业务领域。第四步，上下文检索——让 AI 在对话中能引用之前的上下文和知识库内容。第五步，生成代码并验证——AI 生成代码后，人工审核和测试。第六步，部署上线与迭代——代码通过验证后部署，根据反馈持续优化。这六步构成了 AI 编程的完整链路，每一步都有技术细节，后续课程会逐一展开。这一页是全章的纲领，建议大家拍照保存，作为 AI 编程的 checklist。

第 36 页

【幻灯片标题】AI 编程核心底座——大型语言模型 LLM

【逐字稿】

我们来认识 AI 编程的核心底座——大型语言模型 LLM。LLM 全称 Large Language Model，大型语言模型。它的本质是：依靠海量文本数据训练的 AI 大模型，能听懂人话的 AI 人工智能大脑。大家注意「能听懂人话」这五个字——这是 LLM 最根本的能力突破。以前的 AI，你需要用规则、用特征工程来定义输入；现在的 LLM，你直接用自然语言跟它对话就行。LLM 的能力范围很广：对话问答——你问它答；写代码——给它需求它生成代码；总结文案——给它长文它提炼要点；逻辑推理——给它前提它推导结论；翻译——给它一种语言它翻译成另一种；生成内容——给它主题它创作文章。在 AI 编程场景中，LLM 是核心引擎——你描述需求，LLM 理解并生成代码。所以理解 LLM 的能力和局限，就是理解 AI 编程的能力和局限。

第 37 页

【幻灯片标题】从 LLM（大语言模型）发展的历程

【逐字稿】

我们来看 LLM 的发展历程。整体脉络分为四个阶段：第一阶段，Transformer 奠基——2017 年 Google 提出 Transformer 架构，成为所有现代 LLM 的底层骨架。第二阶段，通用大模型规模化——GPT 系列、LLaMA 系列等通用大模型出现，参数规模从亿级到千亿级。第三阶段，代码专用大模型诞生——针对代码生成场景训练的模型出现，如 CodeLlama、StarCoder 等，在编程任务上表现更优。第四阶段，国内技术引进、本土化自研、规模化落地——国内厂商如百度文心、阿里通义、华为盘古等推出自研大模型，并在各行业规模化落地。这个发展历程告诉我们：LLM 从通用走向专用，从研究走向落地。在 AI 编程领域，代码专用模型的表现已经超过很多人类程序员的基础水平，而且还在快速进化。

第 38 页

【幻灯片标题】LLM 的骨架、血液、大脑

【逐字稿】

我们来深入 LLM 的底层原理——用三个比喻来理解：骨架、血液和大脑。Transformer 做骨架——它是物理载体，决定模型的上限，就像人的骨架决定身高上限。Token 分词做血液——它是数据输入流转输出的介质，就像血液把氧气和营养输送到全身。注意力机制做大脑——它负责筛选重点、关联上下文、理解逻辑，就像人的大脑负责思考。三者合一，构成了 LLM 的底层原理。这个比喻很重要，因为后续课程会分别深入讲解这三个部分。大家先建立这个整体认知：骨架提供结构，血液提供数据，大脑提供智能，缺一不可。接下来三张幻灯片我们逐一展开。

第 39 页

【幻灯片标题】骨架：Transformer 架构

【逐字稿】

先看骨架——Transformer 架构。Transformer 是当前所有主流 LLM 的底层骨架，不管是 GPT、LLaMA 还是盘古，都建立在 Transformer 之上。Transformer 规定了三件事：模块拼接方式——编码器和解码器怎么组织；数据流转路径——信息从输入到输出怎么流动；计算单元排布——注意力头、前馈网络怎么排布。没有 Transformer 骨架，血液无处流淌，大脑无处附着——就像没有骨架，血液和大脑都没有容器。Transformer 之所以能成为统一骨架，是因为它解决了前代架构的核心问题——并行计算和长距离依赖。RNN 只能串行处理，Transformer 可以并行；RNN 难以捕捉长距离依赖，Transformer 通过自注意力机制直接关联任意距离的 token。这两个优势让 Transformer 成为 LLM 时代的基石。

第 40 页

【幻灯片标题】血液：Token 分词

【逐字稿】

再看血液——Token 分词。分词是什么？就是把连续的文本切分成模型最小处理单元，再映射成数字 ID。比如「我爱编程」可能被切分成「我」「爱」「编」「程」四个 token，每个 token 映射成一个数字。模型内部处理的不是文字，而是这些数字 ID。Token 分词有三大特性：第一，信息载体——每个 token 承载着语义信息，是模型理解语言的基本单位。第二，全局流通——token 在整个模型中流转，从输入到每一层到输出，都是 token 在流动。第三，有限容量——模型的上下文长度是有限的，比如 4K、32K、128K 个 token，超出就丢失。理解 token 对 AI 编程很重要——因为你的 prompt 和 AI 生成的代码都是按 token 计

费的，token 越多越贵越慢。所以高效使用 AI 编程的一个技巧就是精简 prompt，用最少的 token 表达最清晰的意图。

第 41 页

【幻灯片标题】大脑：注意力机制

【逐字稿】

最后看大脑——注意力机制。注意力机制负责思考，具体做四件事：判断关系——token 之间的关联程度；抓取重点——哪些信息更重要；理解语义——词句的真实含义；连贯生成——基于上下文生成合理的下一个 token。举个直观的例子：「下雨天记得带」→注意力机制推理出下一个词应该是「伞」。它是怎么做到的？因为在训练数据中，「下雨天」和「伞」经常一起出现，注意力机制学到了这个关联。没有注意力机制，模型只是机械复读字符——给它「下雨天记得带」，它可能输出「饭」「水」「书」等任何词，因为它不理解语义。有了注意力，它能理解「下雨天」这个上下文，从而推断出「伞」是最合理的下一个词。在代码生成场景中，注意力机制让模型能理解变量之间的引用关系、函数调用链、代码逻辑结构，从而生成连贯正确的代码。

第 42 页

【幻灯片标题】构建知识库——给 AI 喂知识

【逐字稿】

有了 LLM 这个大脑，但它还缺少专业知识——LLM 的训练数据是通用的，不了解你的具体业务。这就需要构建知识库，给 AI 喂知识。举个场景：你让 AI 帮你写一个华为云 ECS 部署脚本，但 AI 不知道你们公司的 ECS 配置规范、安全组策略、网络架构。如果你

直接问，AI 可能给出一个通用但不匹配你们规范的脚本。解决方案是构建知识库——把你们的配置规范、部署模板、最佳实践整理成文档，导入知识库。这样 AI 在生成代码时，就能参考这些专业知识，生成符合你们公司规范的部署脚本。知识库让 AI 从「通用智能」变成「领域专家」。在 AI 编程中，知识库通常包含：代码规范、架构文档、API 文档、历史代码、业务逻辑等。构建好的知识库是 AI 编程质量的倍增器——同样的 LLM，有无知识库的代码生成质量天差地别。

第 43 页

【幻灯片标题】AI 通过上下文检索对应知识

【逐字稿】

有了知识库，还需要上下文检索能力来使用它。上下文检索是什么？就是 AI 在对话中能自动从知识库中检索相关信息，加入到当前上下文中。这个能力带来四个好处：第一，不用重复复述需求——你之前说过的，AI 记得，不需要每次都说一遍。第二，实现多轮连贯对话——AI 能理解你说的「把它改成蓝色」中的「它」指的是之前讨论的元素。第三，长文档统一解读——你可以上传一份 50 页的设计文档，AI 能整体理解并在后续对话中引用。第四，缓解 AI 幻觉和前后逻辑矛盾——有了上下文，AI 的生成更准确、更一致。在 AI 编程中，上下文检索特别重要——因为代码项目有大量文件间的依赖关系，AI 需要理解全局上下文才能生成正确的局部代码。这也是为什么 AI 编程工具都在强调「项目感知」「全局上下文」能力。

第 44 页

【幻灯片标题】AI 编程生成代码到部署上线

【逐字稿】

我们来看 AI 编程从生成代码到部署上线的完整流程。分为三个阶段：准备阶段——描述想法→选择大模型→构建知识库→上下文检索，把开发环境和 AI 能力准备好。AI 编码阶段——代码生成→静态检查→动态验证→问题修复，AI 生成代码后需要多维度验证。静态检查看语法和风格，动态验证跑测试用例看功能，发现问题后让 AI 修复。部署上线阶段——部署→上线校验→多维度分析→迭代优化，代码验证通过后部署到生产环境，上线后持续监控，根据数据反馈迭代优化。这个流程的每个环节都有 AI 参与，但人的审核和决策贯穿始终。AI 不是替代人，而是增强人——AI 做执行，人做判断。这个流程也是后续实操课程的主线，大家在这里先建立整体认知。

第 45 页

【幻灯片标题】当下 AI 编程的核心功能与技术优势

【逐字稿】

我们来看当下 AI 编程的核心功能与技术优势，分三个层面。对个人开发者：低门槛上手——非专业程序员也能用 AI 编程；提效革新——编码效率提升数倍；跨界学习——前端工程师也能用 AI 写后端；提质避错——AI 帮忙发现和修复 bug。对企业组织：压降研发开支——同样的人力产出更多代码；缩短交付周期——从月级到周级甚至天级交付；标准化管控——通过 Spec 和 Rule 统一代码风格；优化人力配置——人聚焦架构和需求，AI 聚焦编码。对整个行业：优化人才结构——从码农到架构师；重塑生产模式——从手工到 AI 驱动；加速产业数智化——更多企业能快速构建软件能力；繁荣软件生态——更多应用被快速创造。这三个层面从微观到宏观，展现了 AI 编程的全面影响。但大家也要清醒——这些优势是 potential，要真正实现还需要方法论、工具链和人才能力的配合。

第 46 页

【幻灯片标题】现存痛点与技术瓶颈

【逐字稿】

我们也要客观看到 AI 编程现存的痛点与技术瓶颈，分四个层面。底层技术瓶颈：模型认知有限——复杂逻辑可能出错；优质数据不足——某些领域训练数据缺乏；高昂算力成本——大模型推理成本高；黑箱与多模态缺陷——模型决策不可解释，多模态能力不完善。产品使用痛点：上下文有限——长项目可能超出上下文窗口；输出不稳定——同样的 prompt 可能得到不同结果；部署困难——AI 生成的代码部署到生产环境有挑战。产业落地痛点：难适配企业系统——企业现有系统复杂，AI 生成的代码难以集成；投入高回报不明——AI 工具成本不低，ROI 不确定。安全合规痛点：模型被攻破——prompt 注入等攻击手段；隐私泄露——代码中可能包含敏感信息；版权不清——AI 生成的代码版权归属不明。了解这些痛点很重要——因为只有知道边界，才能正确使用。AI 编程不是万能的，它有明确的能力边界，在这个边界内它是强大的，超出边界它可能帮倒忙。

第 47 页

【幻灯片标题】发展趋势

【逐字稿】

最后来看发展趋势，给大家一个面向未来的视角。技术突破方向：算力优化——推理成本持续下降；模型轻量化——小模型也能做好任务；数据突破——更多高质量训练数据；自主智能体——Agent 能力持续增强。产品发展方向：本地普及——AI 编程工具更易获取；交互自然——从打字到语音到多模态；实时与长效——即时响应和长期记忆；场景

深耕——针对垂直领域优化。产业规模方向：价值量化——AI 编程的 ROI 可度量；人才协同——人与 AI 的最佳协作模式；无缝集成——AI 编程工具融入现有研发流程。安全合规方向：监管完善——AI 编程的法规标准建立；可信 AI——模型输出可解释可审计；版权明晰——生成代码的版权规则明确；治理标准——企业级 AI 编程治理框架。这些趋势告诉我们，AI 编程还在快速演进中，今天的能力只是起点。建议大家保持学习，持续关注技术进展，在这个浪潮中找到自己的定位。

第 48 页

【幻灯片标题】本章小结

【逐字稿】

我们来做一个本章小结，回顾这堂课的四个核心知识点。第一，我们梳理了编程三次变革的脉络——从汇编语言到高级语言到自然语言编程，抽象层级不断跃升，AI 编程工具也经历了从代码补全到对话编程到自主智能体的三代演进。第二，我们讲解了 Vibe Coding 的基础概念——它是一种凭直觉、凭氛围的 AI 编程方式，优势是快速灵活，风险是质量不可控，适合原型验证不适合工程交付。第三，我们讲解了 SDD 规格驱动开发的基础概念——先写 Spec 再让 AI 编码，和 Vibe Coding 是互补关系，Vibe Coding 负责快速探索，SDD 负责规范交付。第四，我们讲解了大模型的发展历程和底层原理——Transformer 做骨架、Token 做血液、注意力机制做大脑，三者构成 LLM 的底层原理。这四个知识点从历史到方法到原理，构成了 AI 编程的完整知识框架。后续课程会在这个框架下深入每个技术细节。

第 49 页

【幻灯片标题】思考题

【逐字稿】

我们来看一道思考题，检验大家对 Vibe Coding 和 SDD 的理解。题目是：关于 Vibe Coding 与 SDD 规格驱动开发，说法正确的是？我们逐一分析。选项 A 说 SDD 依托灵感随意开发，Vibe Coding 提前制定开发规范——这说反了，SDD 是先写规范的，Vibe Coding 才是凭灵感。选项 B 说 Vibe Coding 侧重灵感快速做原型，SDD 通过规范约束 AI 编码——这是正确的，准确描述了两者的定位。选项 C 说二者都需要提前编写详细规格文档再编写代码——不对，Vibe Coding 不需要写规格，直接凭感觉对话。选项 D 说 Vibe Coding 适合正式工程落地，SDD 用于快速验证创意——也说反了，Vibe Coding 适合快速验证，SDD 适合工程落地。所以正确答案是 B。这道题的核心是让大家分清两种模式的定位：Vibe Coding=快速+灵活+原型，SDD=规范+可控+工程。

第 50 页

【幻灯片标题】缩略语

【逐字稿】

我们来看本章涉及的缩略语。第一个是 SDD，全称 Spec-Driven Development，中文叫规格驱动开发——先写规格再写代码的开发方法论。第二个是 LLM，全称 Large Language Model，中文叫大语言模型——AI 编程的核心底座。这两个概念是本章最重要的两个术语，后续课程会反复使用。建议大家记住全称和含义，在阅读技术文档时不会混淆。

第 51 页

【幻灯片标题】学习推荐

【逐字稿】

给大家一些学习推荐资源。华为云官网是 https://www.huaweicloud.com/，可以了解华为云的 AI 编程产品和服务。华为云学堂是 https://edu.huaweicloud.com/，有丰富的 AI 和云计算课程。建议大家课后访问这些平台，动手实践本课程讲的内容。AI 编程是一门实践性很强的技能，光听不练是学不会的。最好的学习方式是：听完每堂课后，用 AI 编程工具做一个对应的小练习，把理论知识转化为实操经验。后续课程也会有实操环节，带大家真正动手用 AI 编程。好，本章内容就到这里，感谢大家的聆听！
