第 1 章

生成式 AI 驱动的软件开发新纪元

1.1 软件开发的效率革命

1.1.1 人工驱动开发模式的痛点与挑战

如果我们将 2023 年以前没有引入生成式 AI 的开发模式称为“人工驱动开发模式”，而将 2023 年开始引入生成式 AI 的开发模式称为“人机协作开发模式”，那么人工驱动开发模式最显著的痛点在于大量重复性劳动占据了开发者的宝贵时间。

开发者经常需要编写大量样板代码，如数据模型定义、CRUD 操作、API 接口实现等，这些工作虽然必要但缺乏创造性。开发项目初期的环境搭建、依赖配置、脚手架（scaffold）创建同样需要重复投入，而每当接触新的技术栈或框架时，开发者又必须从零开始学习和配置。这种高重复性的工作模式不仅消耗了大量人力资源，也让开发者难以将精力集中在真正有价值的业务逻辑设计和创新解决方案上，严重制约了整体开发效率的提升。

获取知识和解决问题是人工驱动开发模式的另一大挑战，开发者在遇到技术问题时往往需要在海量信息中进行低效率的人工搜索。技术文档分散在官方手册、技术博客、问答社区等多个渠道，找到相关信息后还需要结合具体业务场景进行适配和调整。当系统出现缺陷或性能问题时，问题定位和排查高度依赖个人经验，复杂的错误信息、堆栈跟踪和日志分析对新手开发者来说往往是巨大的障碍。这种知识获取方式不仅效率低下，还容易因为信息不准确或过时而导致开发者走弯路，增加了问题解决的时间成本。

在质量保证和团队协作方面，人工驱动开发模式严重依赖人工经验和主观判断，导致质量标准难以统一。代码审查的覆盖度和深度受限于审查者的知识背景和时间投入，潜在的安全漏洞、性能问题和设计缺陷往往难以在早期发现。团队

内部的知识传递效率低下，经验丰富的开发者的最佳实践难以快速传播给团队其他成员，新人培训成本居高不下。项目进度评估、技术选型决策、架构设计等关键环节缺乏客观的数据支撑，主要依靠团队负责人的个人判断，这种模式在面对复杂项目和快速变化的业务需求时显得力不从心，容易产生决策偏差和执行不一致的问题。

这些挑战叠加了全球范围内持续扩大的开发人才缺口，使得软件开发需求与供给之间的矛盾扩大。据 IDC 预测，全球全职开发者的短缺将从 2021 年的 140 万人增长到 2025 年的 400 万人。尽管存在合格人才的缺口，但企业仍然必须实现其增长和收入目标，这意味着对能够提升效率和产出的工具和流程的需求增加，于是生成式 AI 的大爆发为软件开发行业带来了解决困境的可能性。

1.1.2 人机协作开发时代来临

自 ChatGPT 诞生以来，软件开发行业正经历一场前所未有的变革。这场变革不仅是工具或技术的更迭，而且从根本上重新定义了软件的构建方式。IDC 等多家机构的最新研究一致指向一个重要趋势：AI 正在重塑软件开发的工作模式，并突破了人工驱动开发模式对效率的限制。

这场变革的速度远超行业预期。我们正处于软件开发方法论演进的关键节点，仅在 ChatGPT 出现的短短两年内，AI 辅助开发工具就从实验性的产品发展为多数主流集成开发环境的标准配置。这种转变不只影响个体开发者，还更深刻地改变着团队协作方式、组织结构和整个产品开发生命周期。我们正在见证软件工程领域几十年来又一个具有颠覆性的变革。

就以 JetBrains 2024 年开发者生态系统报告为例，人机协作开发时代的来临已经成为一个不可逆转的趋势。这一转变不再是概念层面的展望，而是正在发生的现实。

关于“使用 AI 工具进行编码和其他开发相关活动有什么收益”这一问题的调查数据，揭示了人机协作开发模式正面临和应对人工驱动开发模式的痛点与挑战。调查显示，67%的开发者表示减少了搜索信息的时间，58%的开发者体验到更快的编码和开发速度，57%的开发者在重复性任务完成和生产力提升方面获得显著改善，45%的开发者在学习新技术、框架、语言等方面获得加速。这些数据反映出开发工作流程正在发生深刻的结构性转变。传统的人工驱动开发模式要求开发者花费大量时间在搜索引擎、技术文档和开发者论坛中寻找解决方案，然后理解、消化并应用到具体场景中，而 AI 工具的介入使得这一过程从“搜索—理解—应用”转变为“询问—获得—验证”的高效模式。

AI 工具的广泛采用标志着时代转折点的出现。调查显示，69%的开发者已经尝试使用 ChatGPT 进行编程和其他开发相关活动，其中 49%的开发者正在定期使

用。作为第二受欢迎的 AI 工具，GitHub Copilot 也有 40%的开发者尝试使用，26%的开发者定期使用。这一数据清晰地表明，AI 工具已经从实验性质的尝试转向了日常开发工作流程的核心组成部分。

企业态度的根本性转变进一步推动了这一进程。调查显示，近 80%的公司要么允许在不同程度上使用第三方 AI 工具，要么没有制定相关政策，只有 11%的组织完全禁止使用云端的第三方 AI 工具。这一现象表明，“对 AI 的恐惧正在逐渐消退”，企业开始拥抱这种新的开发模式。

此外，AI 集成正在重塑软件功能边界。调查显示，18%的开发者正在构建与 AI 的集成功能，这意味着 AI 不仅在改变开发方式，更在重新定义软件产品本身的能力范围。从工具辅助到产品内核，AI 正在全方位渗透软件开发生态系统。这些数据共同描绘出一幅清晰的图景：人机协作开发时代已经到来。开发者正在经历从单纯依赖人工技能到与 AI 工具深度协作的根本性转变。这种转变不仅体现在工作效率的提升上，更体现在开发思维模式和工作流程的重构上。传统的“人工驱动开发时代”正在让位于一个全新的“人机协作开发时代”，这一时代的特征是人类创造性思维与 AI 辅助能力的有机融合。

1.1.3　生成式 AI 如何重塑软件开发流程

生成式 AI 对软件开发流程的重塑是一个全方位的变革过程，涉及从需求分析、架构设计到部署运维的整个软件生命周期。这一变革不仅改变了开发者的工作方式，也重新定义了团队协作模式和技能要求。下面将简要介绍生成式 AI 在软件开发应用中比较广泛的几个领域。

1. 代码生成

代码生成是生成式 AI 重塑软件开发最直接、最显著的领域。AI 已从简单的语法补全发展为能够理解开发者意图和开发上下文的智能助手，可以基于自然语言描述生成完整功能代码并解释算法原理。GitHub Copilot 等工具提升了开发者完成任务的速度，同时提升了代码质量，让开发者能专注于更具创造性的工作领域。AI 代码生成正在打破传统编程技能壁垒，使非专业开发者也能通过自然语言参与应用开发，但 AI 生成的代码仍需要人类的验证和优化，我们需要将 AI 视为智能助手而非替代者。

2. 代码审查

AI 通过将代码审查过程自动化和智能化，显著提升了代码审查的效率和质量。相比人类审查者，AI 系统能持续保持专注和严谨。这种“左移”的质量控制带来巨大经济效益——IBM 研究显示生产阶段修复 Bug 的成本是开发阶段的 15～100 倍。AI 代码审查工具正从被动检测转向主动指导，不仅能指出问题，还能提供修

复建议，甚至预测潜在的性能瓶颈和安全风险，使代码审查从质量控制工具变成开发者的学习成长伙伴。

3. 低代码/无代码开发的加速

生成式 AI 的引入显著降低了低代码/无代码平台的使用门槛，缩小了业务需求与实现技能之间的鸿沟。AI 能够理解业务人员使用自然语言描述的需求，将其自动分解为更具体的描述并转化为可执行的代码，使原本需要数月完成的项目在数周甚至数天内交付。需要注意的是，AI 加持的低代码/无代码平台更适合快速原型开发、业务流程自动化和一般应用开发，而对于企业级复杂应用，传统编程的深度仍然必要。这种“公民开发者”的崛起正在缓解开发人才短缺压力，为组织带来更丰富的创新思路。

4. 智能测试

生成式 AI 通过自动化和智能化测试用例生成以及预测性质量分析，提升了测试覆盖率和测试效率。AI 可以分析代码变更并自动生成单元测试、集成测试和回归测试用例；AI 还能发现人类容易忽略的特殊场景，生成边界条件测试；AI 可以从历史数据中学习模式，预测哪些代码模块最可能出现缺陷。智能测试工具如 Testim、Mabl 已能通过自然语言生成端到端测试场景，并在 UI 变更时自动调整测试脚本。

5. 文档生成与知识管理

生成式 AI 通过辅助文档生成来帮助开发者更好地维护文档与代码的一致性。AI 可以通过分析代码结构、代码注释和变更历史，辅助生成或更新 API 文档和代码说明。智能知识管理工具如 Notion AI、Confluence Intelligence 等将团队的历史项目和决策记录转化为可搜索的知识库，当开发者遇到问题时能快速定位相关解决方案。AI 还可以协助项目交接，通过分析项目代码库和提交历史，生成项目概览和关键模块说明，降低人员流动对项目连续性的影响。

1.1.4 人机协作开发模式下的效率革命

多项权威报告和调查数据显示，人机协作开发模式正在引发一场前所未有的效率革命，这场革命的核心特征是通过 AI 工具与人类开发者的深度协作，实现软件开发全流程的效率跃升。JetBrains 2024 年开发者生态系统报告表明，企业正在积极使用 AI 工具提高效率，传统的开发模式正在被高效的人机协作模式快速替代。Stack Overflow 2024 年开发者调查进一步证实了这一趋势，76%的受访者正在使用或计划在开发过程中使用 AI 工具，其中 81%的开发者相信 AI 工具将提升生产力，这表明人机协作开发带来的效率提升已经深入人心。

这场效率革命最直观的体现是开发速度的大幅跃升。GitHub 研究表明：使用 GitHub Copilot 的开发者任务完成速度提升了 55%，这意味着原本需要一天完成的开发任务现在可能只需要半天就能完成。麦肯锡的研究报告进一步证实了这场效率革命的普遍性，指出生成式 AI 已经为软件工程带来了 20%～45%的生产力增长，其原因主要是生成式 AI 的引入减少了在特定活动上花费的时间，例如生成初始代码、代码修正和重构、根本原因分析以及生成新的系统设计。这些数据清楚地表明，生成式 AI 正在重新定义软件开发的时间成本和产出效率。

效率革命的另一层价值在于开发者工作质量和满意度的同步提升。GitHub 与埃森哲的研究数据显示，90%的企业开发者表示使用 GitHub Copilot 后工作满意度得到提升，编码时的挫败感减少，并且能够专注于更有价值的工作；95%的企业开发者表示在 GitHub Copilot 的帮助下，他们更享受编码的过程。这种现象表明，效率革命不仅仅是速度的提升，更是工作内容的质的改变——从重复性的代码编写转向创造性的问题解决和架构设计。

从产业层面看，这场效率革命的影响将是颠覆性的。麦肯锡研究显示，未来 AI 在企业中广泛应用，这一应用得益于 AI 带来的生产力提升，有望创造出 4.4 万亿美元的经济效益。AI 有能力从根本上改变软件产品的开发过程，提高开发速度和最终产出的质量。中国 2024 年政府工作报告首次提出“人工智能 +”概念，多重利好因素推动大模型快速发展，如麦肯锡报告所言，AI 像 19 世纪工业革命中的蒸汽机一样具有变革性，AI 已成为引领新一代产业变革的核心动力。这些趋势表明，人机协作开发模式带来的效率革命不仅体现在重塑个体开发者的工作方式上，更体现在重构整个软件产业的竞争格局上。那些能够有效利用人机协作模式的组织将获得显著的竞争优势，而固守传统开发模式的企业可能面临被淘汰的风险。这场效率革命正在成为决定软件开发行业未来十年发展轨迹的关键力量。

1.2　生成式 AI 概述

生成式 AI 已经成为一个热门研究与应用方向。它不仅为文本、图像、音频以及代码等内容的自动生成带来了全新的可能性，也在一定程度上重塑了开发者的工作流程和软件交付模式。敏捷开发一直强调快速迭代、紧密反馈以及自适应的开发方式，而生成式 AI 通过自动化和智能化能力，将进一步放大敏捷的核心价值，使团队能够在高压、高变动的环境下依然保持高效率和高质量。

在后续的章节中，我们将见到各种生成式 AI 在 Scrum 活动中的深度集成案例，包括自动化代码生成、测试用例编写、回顾分析、部署脚本优化等。为了让读者对这些应用场景有系统、清晰的理解，有必要在本章中详细介绍生成式 AI 背后的技术原理和演化脉络。

我们将以“生成式 AI 是什么、有什么技术基础、主要应用在哪些领域、面临哪些挑战、未来又会如何发展”为主线展开介绍，也会穿插对生成式 AI 与传统工具相比的差异化优势的分析。更重要的是，我们会结合本书后面章节中用到的技术手段——提示词工程（prompt engineering）、微调（fine tuning）、检索增强生成（retrieval-augmented generation，RAG）、智能体（agent）等，在此给出一些初步的伪代码甚至部分示例代码，让读者明确在后续章节中反复出现的 API 或函数，其依托的是何种核心组件。对于自研插件的探索也将在后续章节中提及。本章力求深度与广度兼具，帮助读者在阅读过程中不仅学习到概念和原理，更能掌握如何将这些技术在实际工作中落地。

1. 生成式 AI 的定义以及与传统 AI 的区别

生成式 AI 通常指的是利用深度学习、概率模型或进化算法等方法，直接根据输入内容生成新的输出内容的人工智能技术。与传统的判别式模型（discriminative model）主要聚焦于分类、回归等判别任务不同，生成式 AI 的目标是“创造”。在自然语言处理领域，生成式 AI 常常表现为机器翻译、对话系统、文本生成；在计算机视觉中则是图像生成、图像风格迁移等。近年来，随着深度学习模型（特别是大型语言模型）的出现，生成式 AI 在文本和代码生成方面取得了突破性进展，也因此在软件开发和敏捷流程中展现出前所未有的潜力。

一个直观的例子是：当开发者想要实现某个函数或脚本，却缺少灵感或范例时，传统做法是搜索网络、翻阅文档或者尝试自己编写样板代码。现在，借助生成式 AI，开发者只需在编辑器中描述需求或提供输入输出示例，AI 就可以自动生成初步的函数实现或测试脚本，大大加速了迭代开发速度。

2. 生成式 AI 发展的关键时刻和里程碑

在深度学习成为主流之前，统计模型如高斯混合模型（GMM）和隐马尔可夫模型（HMM）被广泛应用于序列生成和数据模拟中。这些模型在许多传统应用中取得了不错的效果，但在处理复杂和高维数据时遇到了困难。

21 世纪初，深度神经网络的复兴带来了新的可能性。卷积神经网络（CNN）在图像处理上的巨大成功催生了人们对其他神经网络结构的探索，这促进了生成式模型的发展。

2014 年，伊恩·古德费勒等人提出了生成对抗网络（GAN）的概念。这一新颖的框架引发了学术界和工业界的广泛关注，为高质量数据生成开启了新的篇章。其对抗训练策略为模型生成更真实的数据提供了可能。

变分自编码器（VAE）在数据生成和表示学习方面提供了另一种视角。通过引入隐变量和推断技术，VAE 可以学习到数据的连续潜在结构，为各种应用提供了有力工具。

2017 年，瓦斯瓦尼等人提出了 Transformer 架构，该架构很快成为自然语言处理中的标准工具。不久之后，基于 Transformer 的模型如 GPT、BERT 等纷纷涌现，其中 GPT 系列模型在文本生成领域表现尤为出色。

随着技术的进步，生成式 AI 开始与其他领域如强化学习等融合，为仿真、游戏、机器人等多个应用领域带来了创新。

生成式 AI 的发展历程反映了技术、应用和理论的不断进步和交叉融合。从最初的统计模型到现代的深度学习框架，生成式 AI 已经走过了漫长而多彩的历程，未来仍然充满了无限的可能性。

3. 生成式 AI 的基本原理

生成式 AI 的基本原理涉及学习和模仿数据的潜在分布或结构。这一过程通常需要大量的数据来训练模型，使其能够捕捉到数据内部的复杂模式和规律。通过这种学习，生成式模型能够理解数据的本质特征，并在此基础上创造出新的数据实例。

生成式 AI 首先通过分析大量的样本数据来学习数据的分布特性。这一步骤是创造性生成的基础，确保生成的内容既新颖又具有一定的真实性和合理性。

学习完成后，模型能够基于内化的数据模式自动生成新的数据实例。这些生成的数据可以是已知类别的新成员，如生成一张符合特定风格的图像，也可以是完全原创的作品，如创作一首诗歌。

生成式 AI 的这种能力源自其复杂的算法结构，最典型的包括生成对抗网络和变分自编码器。这些算法通过内部的优化机制来不断改进生成结果的质量，使得最终产出既多样化又高质量。

本书所探讨的是如何让生成式 AI 与 Scrum 敏捷流程深度融合，将“机器创造”与“人类洞察”完美结合，从而形成更高级别的自动化与智能化开发模式。我们的关注点不仅停留在“AI 可以生成什么”，更在意“Scrum 团队如何吸纳这些自动生成的内容并赋予它们有价值的应用”。

要说明生成式 AI 区别于传统自动化工具，需要先了解生成式 AI 的核心工作原理及其核心结构。在后面的章节中，我们将详细阐述基础模型、训练方法以及在应用中常见的微调、提示词设计、检索增强生成以及智能体等技术。

1.2.1　与传统工具的差异化优势

虽然在软件开发界，自动化与工具化并不算新概念，但生成式 AI 与以往的自动化工具相比，有着显著的优势。

传统自动化工具通常基于固化的规则或脚本来执行重复性工作。它们往往需要大量的人工配置或逻辑编写来覆盖各种边界场景，缺乏足够的灵活性，一旦需

求变化或场景超出设计范围，就必须手动更新工具逻辑。对比之下，生成式 AI 利用大规模数据训练的模型本身具备更大的泛化能力，可以在面对陌生需求或新环境时，通过语言理解或模式识别快速给出相对合理的生成结果。

传统工具往往只解决某一局部领域的问题，如自动生成 DAO 代码、自动化测试脚本，但很难实现跨领域的信息联动。例如，它们无法根据用户故事直接生成完整的 UI 原型、后端接口骨架与测试脚本的综合解决方案。生成式 AI 则有机会通过大模型所具备的多模态或多任务学习能力，在同一个对话或命令下给出各个环节的自动化产出，让 Scrum 团队一次性完成多个角度的建模或原型工作。

在传统自动化工具里，若需要进行复杂的上下文理解，如项目历史、需求逻辑、团队约定等，往往力不从心。而生成式 AI 可以在提示词里包含大量上下文信息，使得模型输出时考虑项目领域知识、团队代码风格甚至历史缺陷模式等要素，从而给出更符合团队约定的代码或文档。也就是说，生成式 AI 对环境信息与上下文有更强的学习和适应能力，能够在复杂的敏捷流程中提供更细致的支持。

在后续章节（如第 2 章 Scrum 集成、第 4 章规划与准备、第 5 章执行与监控、第 6 章回顾与改进）中，我们会反复看到生成式 AI 与 Scrum 场景完美契合的实例。有赖于上述差异化优势，Scrum 团队往往只需要提供必要的提示词或配置，就能在需求分析、代码生成、测试、部署、回顾等多个环节获得强大的自动化辅助，而不会被烦琐的规则和脚本束缚。

1.2.2 核心架构与工作原理

从本质上看，生成式 AI 的核心可以归纳为“基于学习到的分布或知识，在给定条件下进行采样或推断，从而输出一个新的样本（文本、图像、语音、代码等）”。以主流的文本生成模型（如 ChatGPT）为例，其背后依赖的是深度神经网络对海量文本数据分布进行参数化学习，使得模型能够捕捉语言内部的统计规律和语义关联。一旦给定新的提示词或上下文，模型就能通过内部的注意力机制或概率采样机制，逐字或逐词元（token）地“生成”新的文本。

在本书各章出现的很多伪代码函数——例如自动生成用户故事、测试用例、发布脚本、会议纪要等——基本遵循这样一个流程：

（1）准备好上下文数据（如项目需求文本、代码片段、会议记录），并对其做一定的清洗或预处理；

（2）将上下文数据和指令一并打包为提示词或查询，输入给一个或多个大语言模型（large language model，LLM）；

（3）接收模型返回的文本生成结果，对其进行进一步解析或结构化处理，融入 Scrum 的执行流程中；

（4）若生成结果存在错误或不满足需求，通过再次编辑提示词或微调模型来

迭代改进。

这一循环结构在本书的各章都扮演极其重要的角色。为了让读者在后面阅读伪代码或实现细节时感到熟悉和连续，本章会在稍后的章节中给出一些示例函数，让读者看到“提示词构建→模型调用→结果后处理→再度调用”这一通用范式。

1.3　生成式 AI 核心技术和模型

上一节从整体上阐述了生成式 AI 的核心思路及其与传统工具的区别。接下来，让我们更具体地聚焦在几项对 Scrum 团队极其重要的技术要点上：大型语言模型的结构与调用方式，提示词工程的实践操作，微调的数据准备与操作示例，检索增强生成的具体实现方式，以及智能体技术在多步骤自动化场景中的用法。我们将以 OpenAI 等常见平台的 API 为例，为读者展示“如何实际操作”以及在 Scrum 项目中“如何融合”。

1.3.1　大型语言模型的实操：以 OpenAI API 为例

大型语言模型（LLM）是生成式 AI 的核心引擎，它基于庞大的预训练数据，能够对任意文本输入进行理解并给出相应的生成响应。OpenAI 的 API 是市面上最具代表性的大语言模型调用方式之一，许多团队通过简单的 HTTP 请求就能让 Scrum 中的各个环节与大语言模型衔接。

在 Scrum 场景中，开发者或许需要在冲刺规划阶段生成用户故事，或在冲刺执行阶段自动产出测试脚本。一般可以使用 OpenAI 提供的 chat/completions 或 completions 端点来完成这些自动化操作。下面是一个最简化的 Python 伪代码示例，展示如何调用 OpenAI 的 GPT-4 模型，生成简短的技术文档段落：

```
    import openai

    def generateTechDoc(content_description: str) -> str:
        openai.api_key = "YOUR_OPENAI_API_KEY"
        prompt_text = f"请基于以下描述编写一段技术文档，要求简明扼要:\n"
                      f"{content_description}"
        response = openai.ChatCompletion.create(
            model="gpt-4",
            messages=[
                {"role": "system", "content": "你是一个专业的技术写作 AI。
"},
                {"role": "user", "content": prompt_text}
            ],
            temperature=0.7
        )
        generated_text = response["choices"][0]["message"]["content"]
```

```
        return generated_text

    if __name__ == "__main__":
        doc = generateTechDoc("本系统实现了一个分布式缓存，支持多节点一
致性...")
        print("AI 生成的文档:", doc)
```

这里我们用 openai.ChatCompletion.create 方法调用了 GPT-4 模型，并设定了一个 system 角色内容以提供全局指令，user 角色内容则包含实际的描述文本。生成后，即可得到一段较为连贯的技术文档段落。当然，若要产出更长或更精细的文档，需要在提示词中加入更多上下文信息。

1.3.2 提示词工程的实用策略

在上面的小示例中，提示词只有两三句话。然而在 Scrum 复杂场景中，提示词往往需要更精细化的设计才能产出高质量结果，这也是提示词工程的核心。想让团队在后续迭代中坚持复用同一套提示词逻辑，需要遵循以下几点：

（1）明确开发者的意图和角色定位。开发者可以将模型设定为“Scrum 领导人 AI”或“资深后端工程师 AI”，让它的回答更符合某种专业视角；

（2）给出尽量详细的输入格式示例；

（3）如果需要输出某种结构（如 JSON、Markdown 表格），要在提示词里明确指定；

（4）反复在实践中迭代提示词，直到生成结果符合预期。

以下是一个 Scrum 团队常用的示例函数，用来自动生成用户故事的伪代码：

```
    import openai

    def  generateUserStories(feature_description:  str,  example_
format: str) -> str:
        system_role = "你是一个资深的 Scrum 产品负责人 AI，善于编写高质量用
户故事。"
        user_prompt = f"""\
    请为以下功能描述生成完整的用户故事，遵循此格式:
    {example_format}
    功能描述: {feature_description}
    在故事中注明验收标准。\
    """
        response = openai.ChatCompletion.create(
            model="gpt-4",
            messages=[
                {"role": "system", "content": system_role},
                {"role": "user", "content": user_prompt}
```

```
            ],
            temperature=0.4
        )
        return response["choices"][0]["message"]["content"]
```

你可以事先定义 example_format 为一个示例文本，类似“作为某种角色，我希望……以便……验收标准包括……”这样就能让 AI 输出更加符合 Scrum 团队要求的故事模板。随着团队需求变化，也可以在提示词中动态嵌入更多上下文信息。

1.3.3　微调：数据准备与操作示例

在使用 OpenAI 或其他平台时，如果需要在特定领域或风格方面得到更一致的表现，可以采用微调的方式对模型进行二次训练。对于 Scrum 团队来说，若拥有大量历史需求文档、代码片段、测试脚本等，这些都可以整理成训练数据，让 AI 学会内部最佳实践或公司特有的领域语言，从而在后续调用时更加贴合实际需求。

在 OpenAI 平台上进行微调，目前主要针对 GPT-3.5 等可微调的基础模型。虽然 GPT-4 暂时还不支持公开微调，但原理类似，后续开放后也可以沿用。其大致步骤包括：

（1）准备训练数据：通常为 JSON 格式，每行包含 prompt 和 completion 字段；

（2）上传数据文件，创建微调作业；

（3）等待微调完成并验证结果；

（4）使用微调后的模型名称进行调用。

下面的示例展示了如何准备一个数据集并调用 OpenAI 的微调 API：

```
    # 1. 准备训练数据
    # 每行 JSON 示例：
    # {"prompt": "功能描述：用户可以在系统中提交多张图片。",
        "completion": "作为系统用户，我希望能够一次性上传多张图片...###"}
    # 注意：OpenAI 指定当一条训练数据完成时要用某种终止符，如'###'或
'<|END|>'分割

    def createTrainingData():
        data = [
            {
                "prompt": "功能描述：用户可以在系统中提交多张图片。",
                "completion": "作为一名用户，我希望能够一次性上传多张图
片...###"
            },
            {
                "prompt": "功能描述：需要对每张图片进行自动化测试，识别可能
```

```
的格式错误。",
                "completion": "作为测试工程师，我需要系统自动检查图片格式，\
                               如果遇到不兼容格式则及时发出提醒...###"
            },
            # 这里可以加更多历史的 User Story 或需求描述
        ]
        return data

    def saveAsJsonl(data, filename="training_data.jsonl"):
        with open(filename, "w", encoding="utf-8") as f:
            for item in data:
                f.write(json.dumps(item, ensure_ascii=False))
                f.write("\n")

    # 2. 上传并创建微调作业（以 OpenAI CLI 为例）
    # 你可以在命令行中执行:
    # openai api fine_tunes.create -t training_data.jsonl -m "davinci"
    # 也可以使用 OpenAI Python API:
    import openai, json

    def fineTuneModel(datafile_path, base_model="davinci"):
        openai.api_key = "YOUR_OPENAI_API_KEY"
        # 假设已经在 OpenAI 文档里创建上传了文件并拿到 file_id
        # 也可以用 openai.File.create(...) 先把 datafile_path 上传
        response = openai.FineTune.create(
            training_file="file-xxx", # 上传后的文件 ID
            model=base_model
        )
        return response

    # 3. 等待微调作业完成后，你会得到一个 fine_tuned_model 名称
    # 4. 使用微调后模型调用时，将 model 参数换成这个名称即可

    def callFineTunedModel(prompt_text, fine_tuned_model_name):
        completion = openai.Completion.create(
            model=fine_tuned_model_name,
            prompt=prompt_text,
            max_tokens=300,
            temperature=0.5
        )
        return completion.choices[0].text

    if __name__ == "__main__":
        data = createTrainingData()
        saveAsJsonl(data)
        # 接下来就按照 OpenAI 官方文档上传并执行 fine-tune
```

```
        # 然后用 callFineTunedModel 做测试
```

在实际 Scrum 开发中，若有上百条甚至上千条历史需求描述或测试脚本，按照这种方式将其构建为 prompt-completion 对，就可以让模型“学到”开发团队的特定写作风格、技术术语、业务逻辑。微调完成后，调用时更大概率产出符合团队要求的结果。

1.3.4　检索增强生成：示例与落地

当团队拥有大量文档、代码库或知识库，需要 AI 在回答或生成内容时利用这些资料，就要用到检索增强生成技术。简单说，RAG 流程是首先把全部资料向量化，然后在每次生成之前做检索，最后把检索到的文档片段拼进提示词。

下述示例在 Python 中展示了用 faiss 或 langchain 进行检索增强生成的大致结构：

```
    import openai, faiss
    from langchain.embeddings.openai import OpenAIEmbeddings

    class RAGSystem:
        def __init__(self, index_path):
            self.index = faiss.read_index(index_path)
            self.embeddings = OpenAIEmbeddings(openai_api_key="YOUR_
KEY")

        def searchRelevantDocs(self, query, topK=3):
            query_vector = self.embeddings.embed(query)
            _, idxs = self.index.search(query_vector, topK)
            # idxs 是最相似文档的 ID
            relevant_docs = fetchDocsByIds(idxs)
            return relevant_docs

        def generateWithRAG(self, query):
            docs = self.searchRelevantDocs(query)
            context_text = "\n".join([d["content"] for d in docs])
            prompt = f"基于以下参考资料回答问题:\n{context_text}\n 问
题:{query}\n 回答:"
            response = openai.Completion.create(
                model="text-davinci-003",
                prompt=prompt,
                max_tokens=200
            )
            return response.choices[0].text.strip()

    # 在 Scrum 实际中
```

```
    if __name__ == "__main__":
        ragEngine = RAGSystem("my_faiss_index.index")
        answer = ragEngine.generateWithRAG("如何在系统里实现多用户会话
管理？")
        print("AI 结合内部文档回答:", answer)
```

通过这种方式，大语言模型并不需要在自身权重里就包含所有信息，而是在每次调用前动态检索相关资料，从而获得更准确且上下文相关的回答。Scrum 团队在规划、执行、回顾等阶段，如需引用大量项目文档、API 规范、旧版本记录时，这个模式能极大减少“AI 胡编”的情况。

1.3.5 智能体与多步骤自动执行

若 Scrum 团队想在某些活动中实现真正的“自动执行”，例如自动生成测试用例并运行测试，再根据结果修复缺陷，就需要智能体机制让 AI 在多步骤对话与操作里进行“自我调用”。下面用一个极其简单的范式演示如何构造一个“自动测试智能体”：

```
    class AutoTesterAgent:
        def __init__(self, llm):
            self.llm = llm
            self.context = []

        def plan(self, task_desc):
            prompt = f"你是一个自动测试 Agent. 需求: {task_desc}.\n 请给
出测试计划."
            return self.llm.run(prompt)

        def executeTest(self, code_snippet):
            # 假设可以用某种 sandbox 或 CI 系统来执行测试
            result = runTestInCI(code_snippet)
            return result

        def loop(self, task_desc):
            plan_response = self.plan(task_desc)
            self.context.append(plan_response)
            # 生成脚本
            test_code = self.llm.run(f"基于计划: {plan_response}, \"
                                     "生成可执行的 pytest 脚本.")
            test_result = self.executeTest(test_code)
            if test_result["status"] == "FAIL":
                fix_suggestion = self.llm.run(f" 测 试 失 败 log:
{test_result['logs']}, 请给出修复建议.")
                # 可能自动修复代码
                # ...
```

```
        else:
            print("测试通过")
```

这是一个相对简化的智能体循环：AI 先根据用户描述生成测试计划并编写脚本，然后执行测试，若失败则再次调用 AI 解析日志并尝试修复。虽然仅为伪代码，但已展示了如何让 AI 多次调用自身或外部工具，逐步完成一个复杂任务。Scrum 团队可以把这样的智能体接入 CI/CD 流水线或每日站会的自动任务当中。需要注意的是，智能体目前还存在不确定性与潜在错误，需要谨慎地结合人工审核。

1.3.6　本书所使用的技术与伪代码对应说明

在阅读本书后面章节时，读者将会频繁看到例如：

```
    function generateCodeSnippet(promptData):
        # ...
        codeResult = aiModel.generateCode(somePrompt)
        # ...
        return codeResult
```

或是：

```
    function refineTestCases(originalCases, feedbackLogs):
        retrievalResults = vectorIndex.search(feedbackLogs)
        refinedPrompt = "以下是用户反馈... " + retrievalResults + ",
请改进测试用例..."
        improvedCases = testModel.generate(refinedPrompt)
        return improvedCases
```

读者会反复见到各类伪代码函数，如 generateUserStory、autoCreateTest、analyzeLogsAndFix、ragSearchDocs、agentLoop 等。它们背后的实现基本脱胎于接下来介绍的五大要素：

（1）大语言模型调用：大部分生成文本或代码的函数，底层都是调用 OpenAI 等供应商的 ChatCompletion 或 Completion API；

（2）提示词设计：每个函数中都少不了“系统角色”或“用户角色”提示词的精细设计，让模型明白具体任务需求；

（3）微调：在大公司或有复杂领域知识需求时，需要在本地或 OpenAI 平台上进行微调，使模型更理解项目内部的风格、业务逻辑；

（4）检索增强生成：若要引用项目文档、旧版本需求或大量参考资料，需要在调用模型前进行向量检索，拼接检索到的上下文；

（5）智能体：当需要在 Scrum 流程中执行一系列自动化操作时，比如测试与修复循环，或是自动部署脚本修改，就可能使用智能体范式。

读者在后续看到这些伪代码时，可以将之与本节列举的几大技术要点对照理

解，并在自己的实际项目中用 OpenAI 或其他厂商提供的 API 进行复现。这些函数本质上依赖以上介绍的核心方法，如提示词工程、检索增强生成、微调后的模型等。有些示例中可能还使用智能体的多步对话与操作。通过本章对关键技术架构的概述，读者应能清晰理解后续伪代码背后的逻辑，也能在需要时针对这些方法进行改进或替换。在实际工程中，团队可以使用 OpenAI、DeepSeek 等云端 API，也可以部署自己的模型（如 ChatGLM、LLaMA 等）来完成对接，只需保持调用方式和提示词思路的一致性即可。

1.4 生成式 AI 的挑战与限制

尽管生成式 AI 为各行各业带来了无数机遇，但它也面临很多潜在的挑战和限制。Scrum 团队在落地这些技术时，需要有清醒的认识与应对措施。本节先列出若干关键点，后续章节在具体实践中还会进一步提及。

（1）大型语言模型凭借“下一个词元预测”原理来生成文本，极易出现貌似正确却实则荒谬的“幻觉”式输出。它可能编造一个 API，或在总结资料时杜撰引用来源。对于敏捷开发流程，这种错误若未及时发现，会导致缺陷更迭、误导决策或浪费时间。Scrum 团队需要在 AI 生成后进行必要的审查与验证（如代码审查、测试执行等），也可以借助检索增强生成等技术让 AI 基于真实资料，降低幻觉概率。

（2）许多 Scrum 项目有商业机密或用户隐私数据，若将之传递至云端 API 进行生成，存在泄露风险。一些大型模型的服务方也会使用输入数据来继续训练模型。因此，企业内部对敏感信息的保护在采用生成式 AI 时显得尤为重要。可能需要部署私有化模型，或者采用加密与脱敏技术来保护数据。

（3）大语言模型都有一个上下文窗口限制，一般是数千到上万词元不等。超出窗口大小就无法一次性处理，这对 Scrum 团队的大型文档或大量历史数据而言是个瓶颈。解决方案包括分段调用、向量化检索或使用特定长上下文模型等。仍需在实践中精心设计提示词与数据切分流程。

（4）大模型往往并非一次性封装后永久可用。新版本的模型会不断推出，其推理速度、费用、API 格式都可能发生变化。Scrum 团队需要适应这些变化，动态管理对 AI 服务的依赖，也要关注训练费用、API 调用成本等因素。若对 AI 功能过度依赖而不做应急预案，当模型出现故障或涨价时可能直接影响团队的敏捷节奏。

（5）生成式 AI 可能输出不适宜内容，如歧视性文本、侵权内容等，这与道德及法律法规相关。Scrum 团队若将 AI 生成的内容直接面向用户，需要做好审计与过滤，避免产生负面影响或法律风险。对于内部使用，必须确保模型训练或微调

时未违反版权许可或隐私政策。

1.5　生成式 AI 未来趋势与发展方向

生成式 AI 作为一项方兴未艾的技术，还在迅速迭代与进化中。以下从几个视角对其未来可能走向作简要分析，以便 Scrum 团队对后续的规划和应用预留更多想象与策略空间。

从纯文本生成扩展到跨模态协作，如图文混合、音频与视频编辑、对话与 AR/VR 场景融合等，正成为业界热点。Scrum 团队中的 UI/UX 设计、音视频团队，乃至产品培训环节都将从中获益。以后或许可以在一个提示词中，既让 AI 做语音播报脚本，又让 AI 自动生成与该脚本匹配的 UI 动画或视频素材，无须多个单点工具的切换。

虽然通用模型如 GPT-4 功能强大，但在某些高度专业化场景，行业或企业仍倾向于训练或微调自己的专用模型。医学、法律、金融等对准确度和合规性要求极高的领域尤为如此。Scrum 团队中若有大量的内部历史项目或特定行业知识，也可能倾向于在通用模型之上建立一个内部的“微调大语言模型”，以便获得更贴合需求的生成效果。

随着模型体量的爆炸式增长，部署和推理的成本也不断上升。有很多研究致力于模型压缩与蒸馏（distillation），以便在有限资源的设备上也能运行强大的生成式 AI。对 Scrum 团队来说，这意味着在 CI/CD 流水线或本地 IDE 中就可以部署某些小型但实用的生成模型，而不再过度依赖外部云服务。

未来，智能体不仅在文本与信息处理上具备优势，还可能与硬件自动化或更丰富的 API 结合，形成“自动作业机器人”。它们在 Scrum 中也许能自动发现工作项，执行并验证，然后再做汇报与回顾。人类团队成员更关注产品创新和高级决策，而将大量重复琐事交给智能体去执行。在这种高度自动化的 Scrum 中，生成式 AI 与多步骤智能体的协同会产生巨大效率提升。

大规模生成式 AI 在社会层面也会引发关于隐私、版权、虚假信息等方面的争议，各国政府和组织也在陆续推出规范和监管政策。对 Scrum 团队而言，需要在使用 AI 时保持对相关法规的关注和遵守，也要建立内部审查制度，以免 AI 输出内容与法律或道德相冲突。同时，行业也会出现更多开源或多方协作平台，让企业在合理范围内共享与合力构建通用模型。

在本书后面的部分，我们会介绍一个“自研插件”思路，即 Scrum 团队若具备一定工程能力，可以在现有大模型 API 基础上开发自定义的插件或扩展，让 AI 更好地与特定的项目管理工具、代码库、测试框架结合，从而实现如自动分类需求、自动重构代码、自动执行部署回滚等功能。这一自研插件的实现尚处在原型

阶段，会在后续章节进一步展开，欢迎读者持续关注。

在本章中，我们首先阐述了生成式 AI 的基本概念及其与传统自动化工具的差异，然后深入解析了它的核心技术与模型，包括大型语言模型、提示词工程、微调、检索增强生成、智能体等关键要素。我们也示范了若干伪代码或简易代码，以帮助读者理解后续章节将要出现的 API 和函数背后所依托的技术基石。

正是这些基础使生成式 AI 在 Scrum 及敏捷开发领域具备了产生革命性价值的可能性。在后续章节里，我们会持续运用这些技术来展现如何在规划、执行、回顾、审查与用户反馈环节中把生成式 AI 用到极致，助力 Scrum 团队在面对复杂需求、短迭代、高质量要求时依然能够稳健前进。

在此也要再次强调，生成式 AI 尽管潜力惊人，但其并非“银弹”，依然存在诸多挑战和限制。Scrum 团队要理性对待这些问题，尤其在谬误产生、数据安全、合规与伦理等方面必须保持警惕。唯有在人机协同的模式下，让 AI 成为赋能而非替代的角色，才能真正达成我们所期望的基于生成式 AI 前沿的敏捷探索与实践。

希望本章能为读者在后续的阅读和实战中提供一幅清晰的技术全景图。无论你是初次接触生成式 AI 的新手，还是已有一定经验的专家，都可以在这一章找到足够的背景知识与思路，进而更好地投入到后面的 Scrum 实践篇章中。