方糖
零七
ft07.com

方糖AI自编程入门 1.0

A Beginner’s Guide to AI Self-Coding

版权页

方糖AI自编程入门 1.0

A Beginner’s Guide to AI Self-Coding

作者信息

- 作者
  - Easy
  - Email: easychen@gmail.com
  - 微博： https://weibo.com/easy
  - X： https://x.com/easychen

授权说明

本书采用CC-BY-NC-SA协议发布。

- 您可以复制、发行、展览、表演、放映、广播或通过信息网络传播本作品，但必须署名作者并添加链接到本书GitHub仓库。
- 不得为商业目的而使用本作品。
- 仅在遵守与本作品相同的许可条款下，您才能散布由本作品产生的派生作品。

时代变了，大人

写给普通人的编程书

早在2018年，我就想做这么一件事：让零技术基础的普通人，通过学习编程，将自己的想法变成商业产品。然而，在执行的过程中，我遇到了很多问题。其中最大的困难有两个：

1. **概念爆炸**：软件发展到今天，已经相当的成熟，这对用户是好事，但对于学习者来说，需要了解的知识太多了。而且是在一时间扑面而来。我尝试提出了最小可用技能集（MVS）的概念，希望通过裁剪知识树，构建一个最简单的知识闭环让业务落地，但从最终的完课率来看，对于很多同学来讲，这些内容依然太多了。
2. **实践障碍**：这个问题其实是第一个问题的延续，不能很好地理解基础概念，就很难真正进入实践。就算照着课程照猫画虎，也并不理解背后的原理。只要场景有一点变化，就很难独立完成。学会的知识流于纸面，最终遗忘。

Easy

11.4万 阅读 推广

18-10-8 14:58 来自 微博 weibo.com

发明了一个新词，叫「MVS」——「最小可用技能集」。它是指完成一个最常用的流程所用到的最少技能组合。

当实践类技能（比如编程）的知识树特别大的时候，初学者应该先搞定MVS。别管周边知识、别管完美细节、甚至先别管理解不理解。

先要让自己能完成一个价值构建流程。这样就能先把东西做出来，然后在迭代中，不断向这个技能集中追加和优化细节，不断的获得成就感和真实反馈，不断在实战这种去理解字面上理解不了的东西。

比如web吧。最小技能集大致就包括 界面构成、请求发送、数据接收过滤存储、以及内容的展现。

HTML 标签很多，但刚学的时候 Form 用到的标签熟悉下，就大致够用了。CSS 都可以放到二周目，无非就是难看点被，「又不是不能用」。

然后就是 Form 提交，Ajax 也放到二周目，「又不是不能用」（x2）。服务器端获取下数据，然后存储起来，MySQL 太复杂，先放文件里边。fopen太麻烦，先用 file_put_contents 和 file_get_contents。

最后显示又是 HTML 标签，a b hr br h1~h5 ul li 都够用了。这样最基本的网站就做完，能用了。

接着再回来优化细节。先弄CSS，颜值比较重要。然后回去优化下 Form 提交，改成 Ajax 的，避免老页面提交后再返回修改不合规数据。再考虑下数据存储方式，服务器配置麻烦可以先从文本改成用 sqlite。熟悉完 sql语法 以后换 MySQL。

这就应迭代到200x年web开发的一般水平了… 再往后，前后端分离下，后端只负责 api，前端来构成页面。原来后端做的路由变成要前端自己做了。于是搞搞 SPA。再往后就是数据和组件的同步方案，虚拟dom、组件规划（重用和数据、消息传递）。

自己做完再去看看三大框架都是怎么解决这些问题的。差不多就到 201x年web了。

这样学好处是你可以及早写出东西来。而不是把一颗知识树学一大半后才开始写东西。我这种脑容量记不住。学后一半的时候，前边都快忘了。

然而2024年下半年，我终于等到了这一天：**AI可以独立写出代码了**。为了和`LLM开发`、`LLM调用`等AI编程相关的概念区分，我称这种编程方式为`AI自编程`。

在经过采用AI自编程的方式开发过两个商业项目后，我可以确认，AI模型（目前仅限Claude3.5）已经足够强大，可以完美解决上述两个问题：

1. AI预置知识库加上搜索，可以实时为普通人补充用到的知识、讲解概念，并可以在多个深度上自如调整。
2. AI可以独立完成中低难度的编程任务，这让普通人可以用自然语言跳过技术和语法细节写出想要的应用。

然而，即使如此，AI也只是取代了以往的「语法学习」环节。软件开发是一门博大精深的学问，要通过自然语言写出**功能复杂、质量优秀**的应用，依然需要一些必须的专业知识。

这也是为什么我会写这本小书。

它从大模型的现状出发，结合我们过去半年的实践，再从软件工程中借用了一些工具，提出了一个专门为AI自编程优化的流程 —— **PDTAC循环**。使用它，普通人也可以写出功能复杂、质量优秀的应用，而不仅仅是那些「贪吃蛇」玩具。

Commit（代码入库）
5
Plan（描述需求）
1
Action（Bug修复）
4
Do （生成代码）
2
Test
~~Check~~（测试）
3

对于普通人来讲，这些内容初听时略显专业，但实际操作却很简单，无需望而却步；而对于专业程序员来讲，这些内容可能是老生常谈，但总得有人把它们「贩卖」给普通人，我就先抛砖引玉了。

如果这本小书能帮助到你，我会很高兴。如果你的朋友也想学习AI自编程，请务必将这本小书推荐给TA。

AI代替程序员？

从GPT问世以来，关于AI将取代程序员的传言就从未停止过。每一次AI技术的重大突破，都会引发新一轮「程序员即将失业」的讨论。然而，这些预言总是被证明是「狼来了」的故事。

在过去两年里，我已经习惯了如此，甚至会自动忽略相关信息。在我的认知里，这一天来得不会那么快，因为当时我正用着GitHub的Copilot，给我补全代码。

不过，在喧嚣的媒体之外，我持续留意着一些变化。

公开

Easy

24-9-7 12:46 发布于 重庆 来自 微博网页版 已编辑

就算借助AI，能替代程序员的也不会是普通人，至少第一波不是。什么时候你发现大批产品经理们能独立开发产品（而不是纯web玩具）了，AI编程才算开始落地了

到10月，我陆续看到了一些产品经理和站长通过AI开发的产品，虽然复杂度不高，但出现的数量和速度惊人。于是我捡起了之前弃用的Cursor，体验到了Claude 3.5 Sonnet 的强悍。

我知道，这一天提前到了。

Claude3.5开启的新时代

很多人并没有意识到Claude3.5带来的重大变革；更多人把这个能力归于Cursor。但作为Cursor的老用户，我反而能更真切地感受到这种能力的来源。

也许过些年回过头看，我们会发现Claude3.5模型的发布，标志着一个新的里程碑。往大了说可能是这样的：

> 编程在以前是一种技能，在以后是一种思维方式。

是的，编程能力不再专属于程序员。

想想一下，在哈利波特里边，麻瓜们不但不受限于血统，甚至连魔法都不需要学习，只需要喊一句「小爱同学」就可以释放魔法。

这就是现实。新人程序员们不用再纠结于计算机语言的语法了，他们可以跳过这个阶段，只需要像产品经理一样对着AI指指点点，应用就写出来了。

这极大地降低了编程的门槛。因为语法学习是如此痛苦，即使像我这样写了十年以上程序的人，在学习新语言的时候，依然会有这样的心得：

> Flutter 挺好学的，我两年都学了三遍了。

普通人编程时代的来临

当我在微博上发表对Claude3.5的感叹时，有位网友发表了一句评论：

> 领域知识很重要啊 现在纯用ai写应用是没问题，写出来的用户用的爽才是关键

寒穹0_o:领域知识很重要啊 现在纯用ai 写应用是没问题，写出来的用户用的爽才是关键

24-12-27 17:10 来自北京　　9

Easy 博主 :是的，所以说是不懂编程但是有领域知识的人（尤其是垂直行业产品经理）的巨大机会。

24-12-27 17:17 来自重庆

如果从职业角度看，一个普通人要想通过AI自编程来获得「专业程序员」这个高薪职位，其实是很难的。原因很简单，就算大家都用自然语言编程，**为什么老板不选一个更有专业素养和经验的人呢？**程序员的失业通常不是被普通人替代，而是被**（借助AI而）效率更高的同事替代**。

但是从个体/副业/创业角度看，**这可能是程序员第一次处于劣势**。

纳瓦尔在《纳瓦尔宝典》中提出了一个概念，叫做「新杠杆」，并认为它是将个体的能力几何级数地放大，使其具备了和大型公司抗衡的能量的关键。

相对于「人力杠杆」和「资本杠杆」这种需要「别人许可」的传统杠杆来说，复制边际成本为零的「媒体」、「数字版权」和「代码」对每个人都触手可及，又可以实现产品的规模化。

- 「媒体」杠杆催生大量的自媒体和网红；
- 「数字版权」则让网课讲师和网文作者挣到了大钱；
- 唯有「代码」杠杆因为其高门槛，以前一直掌握在程序员们手里。

新杠杆

旧杠杆

复制边际成本为零的产品

媒体、数字版权、代码

人力杠杆：雇佣更多的人

资本杠杆：烧更多的钱

但现在，这个杠杆的门槛突然没了。是的，理论上讲，会打字就能写程序了 —— 区别只是复杂度和稳定性。

开发成本的降低，将导致相对小众的需求被满足，并由此催动一批超级个体的崛起。一如当年，网文满足了传统文学无法覆盖的小众市场。

在这个过程中，对于普通人而言，他们通常有着一份非技术的主业。当「放大器」对接到「主业」上，效果会立竿见影。如果他们不离职，还拥有「副产品优势」，可以将主业累积的工作流和方法论、行业知识和常识、备选方案和半成品、人脉关系和兴趣爱好通过软件和在线服务的方式规模化变现。

「杠杆」的本质是「放大器」，而零放大一亿倍还是零。程序员的劣势在于，他们太「专业」了，以至于他们真正熟悉的只有编码本身而不是代码承载的业务。相反，专注于业务、有想法但经常「只缺一个程序员」的产品经理们，在这一波浪潮里，可能是最大的受益方。因为AI的脾气是真的好，怎么改需求也不会被打。

当然，起跑线的差距只是统计意义上的，**个体的努力完全可以填平。**生活中处处都是需求，认真生活，深入群众，总是能发现好赛道的。

> 程序员永远不会失业，正如老兵不死，他们只是改做独立开发了。

用以致学：先用后学，边用边学

在开始具体事务之前，我们先来聊聊AI自编程对学习和认知模式的颠覆。

学以致用过时了

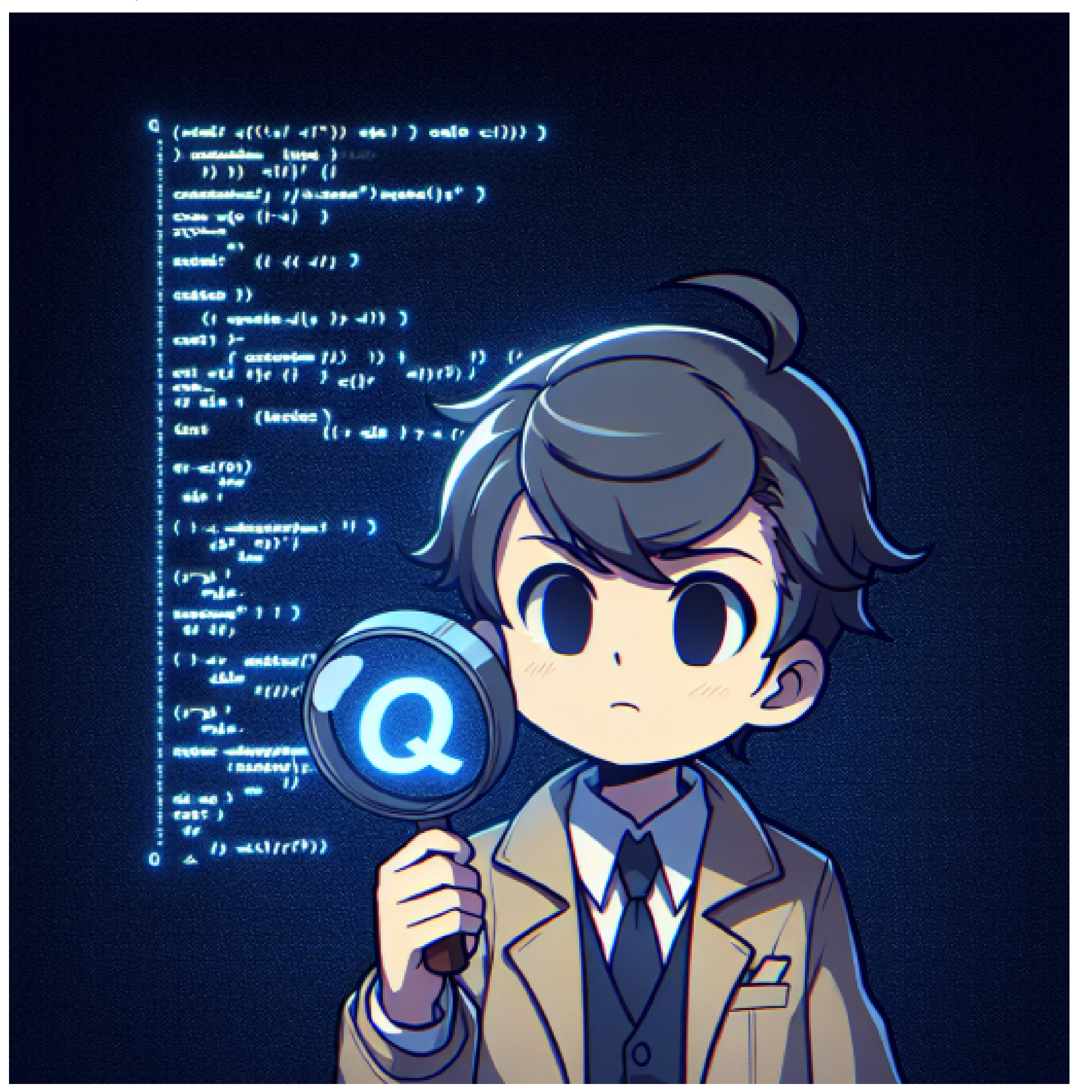
「学以致用」是传统学习方法的金科玉律。它告诉我们：先把书读好，才能在实践中应用。这种方法确实很务实，通过实践来检验所学，避免纸上谈兵。但它的一个隐含假设值得思考：先学后用。

在现实中，许多人在学完之后会陷入「学而不用」的困境。学了半天，觉得自己已经掌握了，但没有实操的机会，知识很快被遗忘，甚至从未转化为实际能力。现代社会的快节奏和时间碎片化，更让这种「学完再用」的方式捉襟见肘。

相比之下，我更推崇一种激进的学习方式，那就是用以致学：先用后学、边用边学。初听之下，这种方法大有「好读书、不求甚解」的感觉。但实际上它大为不同，因为有「用」这个前提在。

用以致学的理念

「用以致学」是一种充满悬疑和实证性的学习方法，它让学习变成一个解密游戏。游戏一开始并不会告诉你所有规则，你需要在探索中发现问题，思考原因，找到解决方案。



先「用」，把业务架起来，运用起来。然后，问题就会像泉水一样涌出来。再带着疑问回过头来找原因，理解后提出新方案，并进行验证。当你解决这个问题，又会冒出新的问题。就这样，在不断解决问题的过程中，你的知识树正在悄然成型。这种方式特别高效，因为每一份知识都是应需求而学，都能立即得到应用和强化。

对于朝九晚五的上班族来说，这种学习方式简直是救命稻草。谁还能有整块时间坐下来啃书本？但用以致学不同，它可以完美利用碎片时间。今天解决一个小问题，明天优化一个小功能，知识就在这些零碎的实践中积累起来了。

遗忘的必然性

对于不以编程为职业的人来讲，「用以致学」显得更为重要。

一个残酷的现实是，即便你花半年时间学完了一门编程语言，如果没有实际应用，它很可能在数月后被遗忘殆尽。甚至，如果学习时间长一点，可能后边还没有学完，前边就已经忘了。

因此我们需要寻找一个使用场景，让我们一直持续不断地用下去。

不要忘了你学习的是编程知识。一个更残酷的现实是，当你学完以后，可能发现框架已经升级了N个大版本了。解决方案是什么？找到一个能持续带来收益的场景。这样，你就有动力持续学习，持续精进了。

> 毕竟，有钱能使磨推鬼。

被颠覆的学习与认知模式

「人只能挣到自己认知以内的钱」——这句话像一条金科玉律，曾经主导着我们对创业和商业的理解。但AI的出现，正在无声地改写这个规则。

想象一下，一位对编程一窍不通的创业者，现在可以通过与AI助手的对话，规划并实现一个功能完整的网络应用。这不是科幻电影里的场景，而是已经发生在当下的现实。过去需要耗费数月甚至数年去学习的专业技能，现在可以通过与AI的协作，在短短数周内实现从零到一的跨越。

这种变革远比表面看起来要深刻得多。它不仅仅是效率的提升，而是彻底改变了人类获取和运用知识的方式。过去，我们必须将知识内化为自己的能力才能实际运用它；现在，我们可以借助AI快速搭建起认知的脚手架，在「不懂」的状态下先把事情做起来，然后在实践中逐步深化理解。

举例而言，一个热爱漫画却不会画画的人，过去需要苦练绘画技能才能讲述自己的故事。而现在，他们可以使用AI来生成漫画故事，甚至动画。这种「先用后学」的模式，正在重塑我们对学习和创新的认知。

当然，在任何领域如果想要做到90分以上，还是需要把这些学习债务补上的，区别的只是顺序的不同。

但在某些领域，这一点点顺序的差异很重要。比如，创业领域。九成以上的创业项目不会成功，它们在找到「产品市场契合」之前就死掉了。《精益创业》告诉我们，对最小可用产品（MVP）来讲而言，软件质量并不是最重要的。重要的是弄清需求、了解市场和验证商业假设。

AI时代的认知信用卡

现在，AI正像一张认知信用卡，让我们可以「先使用，后还债」，让很多不可能的事情变得可能。

比如说，一位在教育行业工作多年的老师，发现了一个改进在线教育的好想法。在过去，她可能会因为缺乏技术背景和启动资金而放弃这个想法。但现在，她可以通过AI自编程，仅仅花费百来块钱就快速开发出一个教育软件的原型。在这个过程中，**她不需要精通编程，而是专注于她最了解的教育场景和用户需求**。如果产品得到市场认可，她可以逐步学习必要的技术知识，或者招募技术团队来扩展产品。

这张信用卡让创新和创业变得更加大众化。不再是只有技术专家才能创造软件产品，不再是只有资深金融人士才能设计金融服务，不再是只有经验丰富的营销人才才能开展数字营销。任何对某个领域有深入理解的人，都可以借助AI的能力，将自己的洞察转化为实际的产品或服务。

这就是AI带来的颠覆：它不仅改变了我们获取和运用知识的方式，更重要的是，**它让创新和创业变得前所未有的平民化。在这个时代，每个人都有机会成为创新者和企业家。**

编程只是一个开始，新时代的大幕正在缓缓揭开。

反馈螺旋：小步快跑

让我们从对时代变迁的感慨中抽离出来，开始专注于眼下的细节。看得见大局，更要做得好小事。

模型和业务的匹配

让我们先来直面一个现实问题：AI模型与业务需求之间存在的「能力差距」。现在的AI编程还处在蹒跚学步的阶段，就像一个聪明的新手程序员，它已经能漂亮地完成一些中低难度的编程任务，但在面对复杂的业务需求时，还会显得有些力不从心。那么，面对这样的局面，我们该如何应对?

有两条路摆在我们面前。

第一条路，就是选择等待。「等等党终会胜利」，我们可以安静地等待，直到AI模型成长得足够强大，完全胜任我们的业务需求。这个等待的时间可能不会太长，因为AI的进化速度令人惊叹。但这条路有个明显的问题：虽然你几乎不需要付出，也不用承担风险，稳稳地坐等胜利，但你可能会错过最珍贵的机会窗口。

第二条路，则是主动出击——改变我们的工作流程，让它与当前AI模型的能力相匹配。这就像是为AI量身定制一套「专属服务」。虽然这需要我们作出一些调整，但好处是显而易见的：我们可以立即开始使用AI的能力，提升业务竞争力，并且抢占市场先机。

更令人兴奋的是，这种改变带来了前所未有的机遇。

> 我知道这个话题有些重复了，但重要的事情值得说三遍。

想想看，以前那些小众需求，由于开发成本高昂，往往难觅程序员问津。毕竟，程序员们都有自己的本职工作要忙，如果找不到对特定项目感兴趣的开发者，这些小众需求就只能继续被搁置。

但现在，游戏规则已经改变。AI的加入大大降低了开发门槛，几乎人人都能开发出简单的应用。虽然这些应用在技术层面可能很基础，但它

们对用户需求的理解和市场覆盖却可能异常精准。这就为创业带来了巨大的机会。

想想看，如果仅仅是写文章来满足人们的阅读需求，就能催生出大量自媒体和网红，那么当我们能以工具和服务的形式来满足那些小而急切的需求时，这个机会该有多大?

我常常这样形容当下的机遇：它堪比个人电脑普及时代的来临。

假设你能穿越回那个时代，明确地知道电脑即将改变世界，你会甘心看着机会从指缝中溜走吗? 即便最后没能完全把握住机会，至少你努力过，不会留下遗憾，是吧?

而说到AI编程带来的机遇，我们甚至都不需要说「亡羊补牢」，因为「羊」还在这里，机会刚刚降临。我们有充足的时间去把握它，去拥抱这场变革。

这就是为什么我更建议大家调整工作流程，去适配当前的AI模型。未来已来，当然选择拥抱它。

如何适配大模型

理解了AI与业务之间的「能力差距」，并决定适配大模型以后，我们就来到了一个关键问题：如何让我们的工作方式与AI模型的能力完美匹配?

了解大模型

要回答这个问题，我们首先需要认识大语言模型。别被「认识」这个词吓到——我不是要带你深入模型的技术细节，钻研它的底层架构，甚至不需要你去学习如何调用API。虽然这些知识都很有价值，但如果你的目标是用AI自动编程，需要掌握的其实是另一些更实用的特性。

预置知识库

几乎所有主流大模型都采用对话的方式提供服务。通过海量数据的训练，它们都内置了丰富的知识库，就像一个博学多识的朋友。不过这位朋友的「知识更新」总是会比现实世界慢上一两年——因为它的知识都是预先装载的，就像一本写好的百科全书。

不过，好消息是这一年来模型们都在努力追赶时代的脚步。特别是在用户界面上，它们开始学会了「现学现卖」——通过整合搜索功能，它们可以即时获取最新信息。这就像是这位朋友除了依靠自己的知识储备，还学会了随时查阅最新资料。虽然这些新知识还不是它的「固有记忆」，但至少让它能够与时俱进。

推理能力

现在来说说一个让很多人惊讶的特性：大模型的推理能力。

有趣的是，很多程序员对此嗤之以鼻，认为大模型不过是一个靠概率生成文本的工具。这种看法并非完全错误，但未免过于武断。就像人类从蹒跚学步到能够跑步跳跃是一个渐进的过程，当模型的参数达到一定规模，经过大量代码训练，再配合思维链的引导，它展现出了令人惊叹的智能涌现现象。

这种涌现的能力，最直观的表现就是推理能力。你可以把它想象成一个善于归纳和演绎的学者，能够通过逻辑链条逐步推导出结论。就像几何学中从几个简单的公理可以推导出整个知识体系一样，有了知识库和推理能力，模型就能举一反三，从有限的信息中推导出更多洞见。

这就解释了为什么Claude 3.5能出色完成任务，而其他一些模型可能表现平平——关键就在于推理能力的差异。就像两个学生，知识储备相似，但分析问题的能力有高下之分。

上下文窗口

接下来要说的是一个经常被误解的概念：上下文窗口。

很多人以为上下文就是提问和回答的内容总和，但实际情况要复杂得多。

有个有趣的事实是，大语言模型其实是「没有记忆」的。是的，你没听错。

当我们使用ChatGPT这样的产品时，看似它记住了我们的对话，实际上每一轮对话系统都会把完整的对话历史重新发送给模型。这就像是每次交谈，都会把之前说过的话重新告诉它一遍。不仅如此，上下文还可能包含网络搜索结果、上传的文档等内容。

上下文窗口可以有多大呢?

模型名称	上下文长度(tokens)	备注
GPT-3	最多2048	-
Mistral 7B	最多8192	-
GPT-4o	从60K到128K	部分配置支持更长上下文
Claude 3.5	最多100K	-
LLama 3.1	最多128K	-
Gemini 1.5 Pro	最多1M	-

让我们看看它的发展历程：2023年GPT-3刚出道时，只能处理2048个token（相当于800个中文字）；到了现在，Claude 3.5已经能处理100K了；Google的Gemini 1.5 Pro 更是号称达到了惊人的1000K（也就是1000000个token）。

但这里有个有趣的现象：就像人类注意力会在阅读长文时逐渐分散一样，模型处理特长文本时的表现也会大打折扣。所以在实际应用中，短小精悍往往比冗长絮叨更有效。

适配策略

基于对这些特性的理解，我们就知道该如何调整工作流程了。

一次只做一件事

最重要的原则是：一次只做一件事。

为什么？因为同时处理多个任务会让模型像一个多线程的大脑一样启动多个思维链。如果这些任务既有共同点又有差异，很容易造成「思维混乱」，就像人同时做多件事容易出错一样。

让任务规模与上下文窗口相匹配

第二个重要原则是：让任务规模与上下文窗口相匹配。

比如我们使用 Claude3.5 模型，它的上下文窗口是100K，那就意味着我们需要把相关资料控制在这个范围内，并且还要预留返回内容占用的Token数。如果通过API形式调用模型，一旦超过这个值会直接报错。如果通过官方的网站和APP使用，通常软件会自动删减内容以防止出错。

考虑到模型处理特长文本时注意力会逐渐衰减，保持更短的长度可能会带来更好的效果。就像人类阅读长文一样，越往后注意力越容易分散。

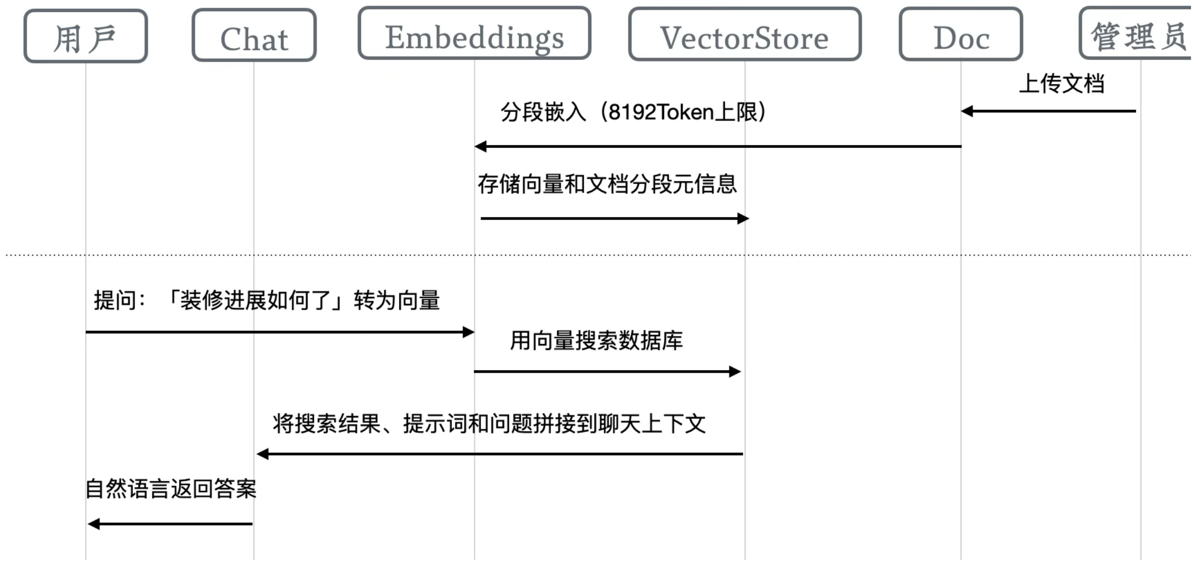
检索增强（RAG）

但现实总是充满挑战。在编程领域，我们经常会遇到这样的情况：一个项目动辄成百上千个文件，每个文件可能就有几十K的代码量。虽然100K或2000K的上下文对于日常对话来说已经很宽裕了，但面对庞大的代码库时，这个上线很快就会被触及。

这时候我们就需要引入检索增强（RAG）技术。它本质上是对资料的预处理。就像是我们在图书馆查阅资料——虽然我们无法同时阅读所有的书籍，但我们可以先从目录找到最相关的章节，然后只阅读这些章节。

检索增强技术的核心理念看起来只是增加了一个相关资料的搜索—— 实际也是如此，但目前主流方案采用的是基于向量的语义搜索。传统的关键词搜索就像是在书中寻找特定的词语，如果作者用了不同的表达方式，即使意思相近也可能搜索不到。

而现代的检索增强技术采用了更智能的「向量化」方法。它会先将资料分成适当的片段，然后将这些文本片段转化为向量（可以理解为多维空间中的坐标点），同时也会将你的问题转化为向量。最后，系统会在这个多维空间中寻找与你的问题最接近的片段，就像在星空中寻找最临近的星星。



一些AI工具会将这个过程自动化。以Cursor为例，它会自动为项目目录下的文本文件建立索引。这个功能虽然可以在配置中关闭，但我强烈建议保持开启。当你选择了「Codebase」方式提出问题时，它会在整个项目范围内进行智能搜索，找出最相关的代码片段，将它们放入上下文中，从而帮助模型生成最准确的答案。

总结一下，为了让我们的工作流程更好地适配AI模型，我们需要遵循三个核心原则：

1. 一次只处理一个任务，避免思维混乱；
2. 将任务规模控制在上下文窗口范围内，确保模型能够充分理解；
3. 对于大型项目，善用检索增强，让AI能够精准定位并利用关键信息。

这就像是在与一位博学但注意力有限的助手合作——我们需要用恰当的方式提供信息，才能获得最佳的协作效果。

让业务适配大模型

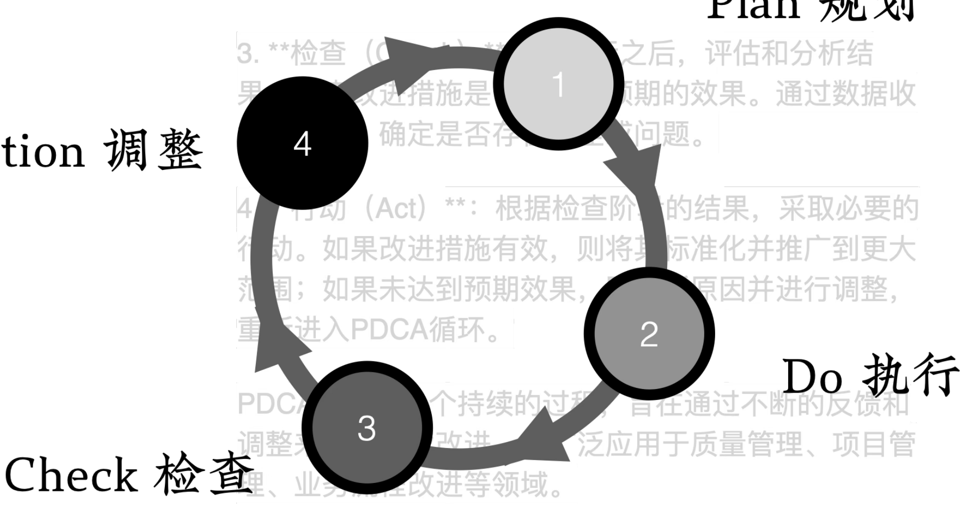
即使在了解了以上原则，面对复杂的业务需求，我们依然会感到困惑：我们的业务是一个庞大而复杂的整体，如何让它适配AI模型的能力限制呢？这个问题的答案，藏在一个经典而强大的管理工具中——PDCA循环。

PDCA循环最初是由管理大师戴明提出的质量管理工具，它以其简洁而高效的特点，在全球范围内广受推崇。这个工具之所以被称为「循环」，是因为它像一个永不停歇的螺旋，推动项目不断向上攀升。每一圈螺旋都包含四个关键步骤：Plan（计划）、Do（执行）、Check（检查）和Action（行动）。

PDCA循环是一种用于持续改进的管理方法和工具，全称为计划（Plan）、执行（Do）、检查（Check）、和行动（Act）循环。它由质量管理专家爱德华兹·戴明（W. Edwards Deming）推广，因此有时也被称为戴明环。以下是每个阶段的简要说明：

1. **计划（Plan）**：在这个阶段，确定需要改进的目标或过程，分析现状，找出问题的根本原因，并制定详细的改进计划和目标。

2. **执行（Do）**：实施计划中制定的改进措施。在这个阶段，通常会进行小规模试验，以便在较小范围内验证计划的有效性。



想象一下建造一座高楼的过程。如果我们试图一次性完成所有工作，这无疑是令人生畏的。但如果我们将整个工程分解成一层一层，每一层

都经过严格的计划、执行、检查和改进，这座高楼就会稳步地拔地而起。PDCA循环正是这样的工作方法——它帮助我们将庞大的业务拆分成一个个可管理的小任务，每个任务都经历一个完整的循环，最终汇聚成项目的整体进展。

当我们将PDCA应用到AI编程领域时，这四个步骤展现出了新的生命力:

Plan（计划）阶段：这是我们描述需求的关键时刻。就像建筑师需要绘制详细的设计图纸一样，我们需要清晰地定义每个小功能的具体要求。这个阶段的准确性直接影响后续AI生成代码的质量。

Do（执行）阶段：这是AI模型大显身手的时刻。基于我们的需求描述，它会快速生成相应的代码。这就像是有一个能够迅速将图纸转化为实物的神奇工匠，只不过这个工匠是AI。

Check（检查）阶段：这是验证环节，我们需要仔细检查AI生成的代码是否满足要求。对于程序员来讲，可以Review每一行代码；但对于普通人，只能从执行效果的角度去判断任务是否完成。

Action（行动）阶段：如果在检查过程中发现问题，这个阶段就是进行修复和优化的时机。就像建筑工程中发现问题需要及时补救一样，我们需要修复代码中的bug，确保其完美运行。

然而，AI编程有其独特的特点，这让传统的PDCA循环需要一些调整。

- 首先，AI的能力是有边界的，就像一个才华横溢但专业领域有限的助手。
- 其次，AI的工作速度快得惊人，能在几秒钟内完成人类需要数小时才能完成的编码工作。
- 最后，也是最具挑战性的一点，AI的输出并非总是稳定的。

这种不稳定性表现得很特别：可能连续九次都完美执行任务，却在第十次莫名其妙地失常。这就像是一位天才艺术家，大部分时候都能创作出精品，但偶尔也会有灵感枯竭的时候。这是大语言模型固有的特性，很难完全避免。

正是基于这些特点，我们对PDCA进行了创新性的改良，发展出了更适合AI编程场景的PDTAC循环。这个优化版本保留了PDCA的精髓，同时增加了专门应对AI特性的新机制。关于PDTAC循环的详细内容我们会在后续章节深入探讨，你现在可以把它理解为一个经过强化的PDCA循环，专门为AI编程场景量身打造。

有了这个强化版的循环工具作为指导方法，我们就能够将庞大的业务需求优雅地拆解成一系列小型特性。每个特性都被精心控制在AI模型的上下文范围之内，确保我们能够专注地处理单一任务，从而获得最优的输出结果。

这种循环往复、稳步前进的方法，正是我们在AI自编程早期需要掌握的核心工作方式。它让我们能够以一种结构化、可控的方式驾驭AI的能力，将其转化为实实在在的生产力。

时间机器：版本管理

软件工程的重要性

在探讨版本管理这一章节之前，我觉得需要强调下软件工程的重要地位，因为后续两个重要工具都来自它。

作为计算机科学的重要分支学科，软件工程的一个突出贡献在于它让经验有限的程序员也能够开发出高质量的项目。这种能力令人瞩目，因为它实现了一个看似不可能的目标：让全球范围内的程序员能够进行超大规模的分布式协作。

软件工程作为一门学科，其重要性在过去可能被低估了。

尽管这门学科本身具有严谨的科学性，但其效果往往受到执行团队水平的制约。然而，在人工智能时代，这一状况正在发生改变。

软件工程与人工智能的契合

大模型与初级程序员有着诸多相似之处：它们都需要从基础开始，通过语言学习和知识积累来逐步提升能力，也都可能出现不稳定性和错误。

虽然通用人工智能的目标是构建世界模型，但从目前基于大语言模型的实现来看，它更像是在模拟人类：从牙牙学语到学会推理，同时也继承了人类的缺点：不确定性。

所以，为什么软件工程不能直接用于大模型和智能体呢?

相比人类团队，AI在执行标准化流程时表现出更强的可靠性和适应性。可以预见，在未来，随着人工智能深度参与开发过程，软件工程的原则和方法必将得到更广泛的应用。

需要说明的是，这种趋势并不意味着软件工程会立即转化为高薪就业机会。相反，它更多地体现为相关知识和实践在各个领域的广泛应用。

版本管理的必要性

在所有软件工程的解决方案中，版本管理是最基础也是最重要的工具之一。

当我们处理简短文档时，版本管理的重要性也许并不明显。但在编写长篇小说、学术论文或开发软件项目时，有效的版本管理就变得不可或缺。我们需要能够追踪修改历史，随时回溯到之前的版本。

虽然现代文字处理软件通常包含一些版本控制功能，但在处理程序代码或图形设计等复杂项目时，专业的版本管理工具就显得尤为重要。

版本管理的实现方式

全量存储

最简单的版本管理方式是全量存储。每当需要保存一个新版本时，系统会将所有文件打包并赋予版本号（如0.1、0.2等）进行存储。需要恢复某个版本时，只需解压相应的包即可。这种方法直观简单，但存在明显的缺陷：它不仅耗费大量存储空间，在处理大型项目（如包含数十万文件的node_modules目录）时，复制过程也会变得异常耗时。

增量存储

为了解决这些问题，现代版本管理系统采用了更智能的增量存储方式。系统首先保存一个完整的初始版本，之后只存储每次修改的差异部分。当需要访问某个版本时，系统会从基础版本开始，应用相应的变更记录来重建目标版本。这种方法不仅大大减少了存储空间的占用，还提高了版本切换的效率。

文件变更检测

一个细节是文件变更的检测。在实际操作中，系统需要精确识别文件的变更情况。这通常通过计算文件的哈希值来实现，现今普遍采用SHA1算法。该算法能将任意文本内容转换为唯一的字符串标识。通过比对不同时间点的哈希值，系统可以准确判断文件是否发生了变化。

当项目变得复杂：多人协作的艺术

想象一下，你正在经营一家餐厅。主厨要研发新菜品，副厨要改进现有的菜谱，但你不能让他们直接修改餐厅正在使用的菜单，对吗？这就是软件开发中的真实场景。当项目变得复杂，多个人需要同时开发不同功能时，我们需要一个更智能的协作方式。

分支与合并：平行宇宙的完美交汇

这就是「分支」概念的由来。设想项目是一条主时间线，每个开发者都可以创建自己的「平行宇宙」（分支），在那里自由地进行开发和测试。当工作完成后，这些「平行宇宙」会回归主时间线，将新功能合并进来。

就像魔法一样，当两位开发者的修改发生冲突时——比如他们都修改了同一段代码——系统首先自动合并，如果无法处理，则会及时发出警告。这就像是两位厨师都想修改同一道菜的配方，系统会要求他们先协调好改法，然后才能更新主菜单。这样就确保了主项目的稳定性和可靠性。

Git：程序员的时光管理器

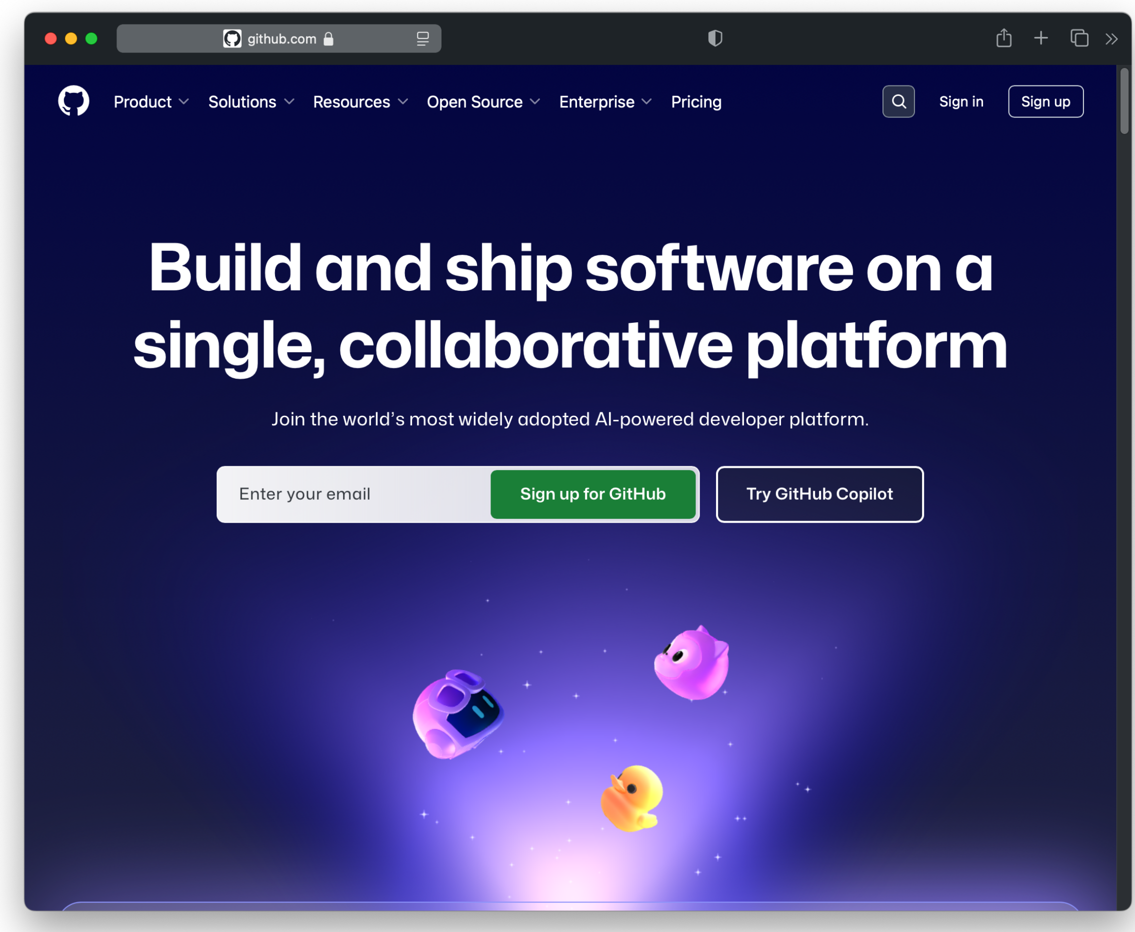
在众多版本管理工具中，Git是当今最受欢迎的「时光管理器」。它就像一个无所不能的助手，不仅可以帮你管理本地的修改记录，还能将它们同步到云端，实现分布式协作。让我们来看看它的基本操作：

1. 首先在你的电脑上通过 `git init` 初始化一个Git仓库（就像开辟一个时空档案室）
2. 通过`git add`命令添加文件（相当于把文件放入时空胶囊）
3. 用`git commit`保存一个版本（封存这个时间点的所有内容）
4. 最后用`git push`将内容上传到云端（在云端为你的时空档案创建备份）

这种设计带来了意想不到的好处：即使网络断线，你依然可以在本地进行版本管理，就像在没有信号的时候，你的手机依然可以拍照一样。等到网络恢复，再将内容同步到云端就好。

GitHub：开源世界的社交平台

如果说Git是一个时光管理器，那么GitHub就是一个让全世界开发者共同创造的社区。这个平台不仅存储代码，更是一个充满创意和协作的天地。



在GitHub上有一个很特别的功能叫做「Fork」（分叉）。想象你在一家餐厅吃到一道特别喜欢的菜，但你觉得可以做得更好。Fork就像是把这道菜的配方复制一份到自己的厨房，你可以随意改进，等到你对改进满意了，还可以把新配方分享回去。

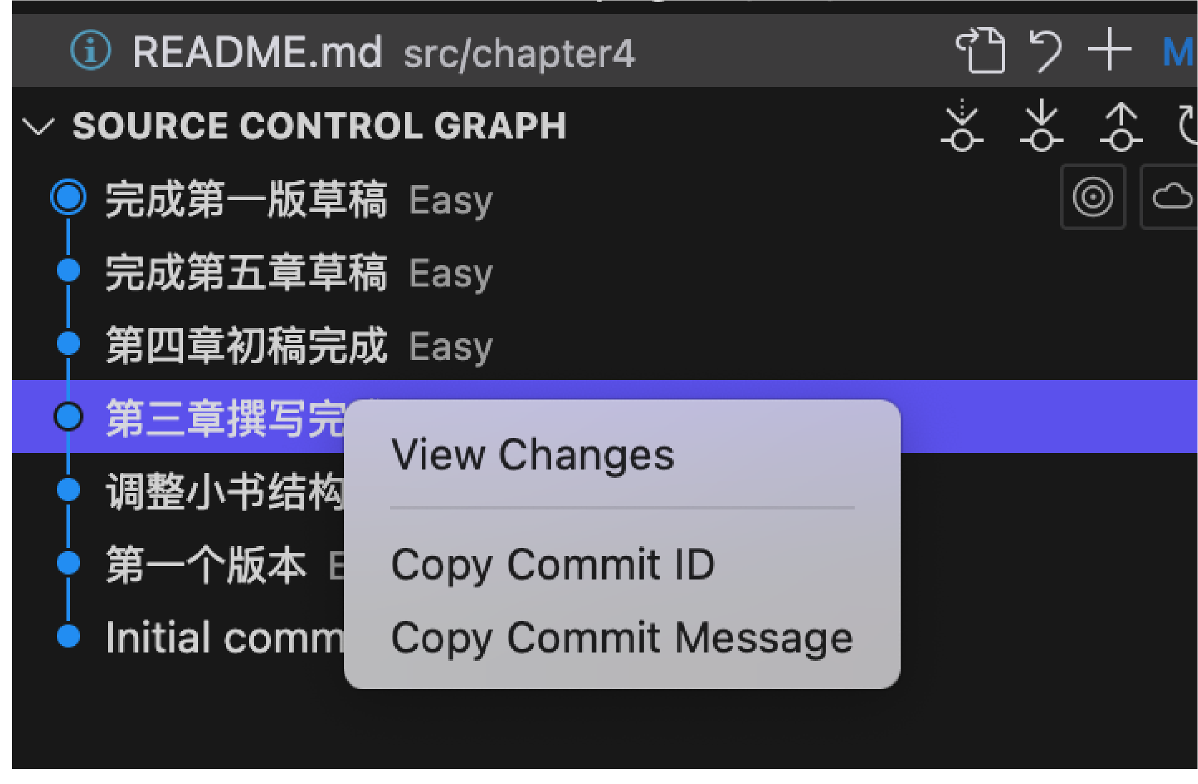
简单但强大的工具

别被这些命令吓到。在这个AI时代，你甚至不需要记住任何命令。就像问路一样，你可以直接用自然语言告诉AI你想做什么，它会告诉你具

体的操作步骤。比如："我想把最新的改动保存起来"，AI就会告诉你需要使用的确切命令。

现代编辑器：版本管理的亲密伙伴

如果你依然觉得命令行操作太复杂，这里有个好消息：现代编辑器已经为我们打造了一个轻松管理版本的环境。就像智能手机把复杂的通讯技术变成简单的点按操作一样，Visual Studio Code等主流编辑器已经将版本管理功能完美集成到图形界面中。你再也不需要记住复杂的命令，只需要点几下鼠标，就能完成所有版本管理操作。



在AI时代的特殊意义

在这个AI迅速发展的时代，版本管理变得比以往任何时候都更加重要。想象一下，你正在和AI助手一起改进代码，就像和一个热情但有时过于积极的合作者一起工作。AI可能在修改代码时不小心改变了一些重要的功能，如果没有版本管理，这些变化就像泼出去的水，想收回来都难。

AI编程的安全网

记住一个简单的原则：经常保存版本。就像玩电子游戏时需要经常存档一样，在编程过程中，特别是在使用AI工具时，频繁地创建新的版本能确保你的工作始终处于安全状态。这样，无论AI做出什么改变，你都能轻松回到之前的任何一个正常工作的版本。

双重保险

一些先进的AI编程工具，比如Cursor，已经意识到了这个需求，它们内置了检查点（Checkpoint）和时间线（Timeline）功能，让你能够追踪每一步的修改。这就像是在游戏中的自动保存功能，每隔一段时间就会自动为你创建一个存档点。

CHAT COMPOSER ··· ⌘N + ···

Checkpoint created. restore

参考资料 preface.md

你是世界一流的AI书籍编辑和作者。请根据 @参考资料 撰写前言的内容。

我来帮你撰写前言内容。首先我需要查看参考资料的具体内容来整理思路。

Read file 方糖AI自编程入门思维导图.md

Read file 演讲稿.txt

基于参考资料，我来为你撰写前言内容。我会编辑

src/p

Revert all files to before this message

Checkpoint created. restore

不过，正所谓「百密一疏」，即使有了这些内置功能，专业开发者仍然建议同时使用独立的版本管理系统。

版本管理已经从一个简单的代码备份工具，发展成为现代软件开发中不可或缺的基础设施。在AI逐渐成为我们得力助手的今天，掌握版本管

理就像给自己配备了一条安全绳，让我们能够大胆地在AI的协助下进行创新和尝试，同时又不用担心会失去控制。

质量控制：全量测试与自动化

当简单变得不再简单

想象你正在搭建一座乐高城堡。刚开始时，放几块积木很容易，城堡看起来也很稳固。但随着建筑越来越高，每加一块新积木，都可能让某个角落松动，甚至导致整座城堡倒塌。这正是我们在用AI开发复杂应用时面临的挑战。

现在，只要会打字的人都能用AI写出简单的应用程序。但当我们想要建造更宏伟的「城堡」时，问题就出现了：加一个新功能，可能会破坏三个已有的功能。渐渐地，我们的应用就会变成一个「问题集合体」，就像一座摇摇欲坠的积木城堡。

为什么AI会犯错?

黑箱的不确定性

AI就像一个才华横溢但有些任性的艺术家。即使你给它相同的指令，它每次创作出的作品都可能略有不同。这是因为AI本质上是一个「黑箱」，它通过大语言模型进行创作，而不是按照固定的逻辑运行。就像人类艺术家的灵感，有时会特别出众，有时则可能略显平庸。

上下文的重要性

想象你在给朋友讲一个故事，如果漏掉了关键的背景信息，朋友很可能会理解错误。AI也是如此，它完全依赖我们提供的上下文来进行创作。如果我们忘记提供某些重要信息，或提供了错误的参考资料，AI自然会得出不准确的结果。

人类也会犯错

有趣的是，这些问题并不是AI独有的。想想看，人类在疲劳时可能会把简单的单词都打错，更不用说编写复杂的代码了。实际上，在某些方面，人类可能比AI更容易出错——毕竟AI不会感到疲劳，也不会因为心情不好而影响工作质量。

质量控制的解决方案

全量测试：安全网的编织

解决这个问题的方法其实很简单：每次添加新功能后，对所有功能进行全量测试。这就像在建造乐高城堡时，每加一块新积木，都要检查整座城堡的稳固性。

听起来工作量很大？确实，在大多数行业，这样的全面检查几乎是不可能完成的任务。但在软件开发领域，我们有一个秘密武器：自动化测试。

自动化测试：永不疲倦的质检员

想象有一个机器人帮你检查乐高城堡的每个角落，快速而准确。这就是自动化测试的原理。它能在几秒或几分钟内完成人工可能需要几天才能完成的测试工作。

在传统开发中，编写测试代码常常被视为一项枯燥的额外工作。但在AI时代，这个问题迎刃而解：我们可以让AI来编写测试代码。你只需要告诉AI：「请为这个新功能创建自动化测试」，它就会帮你完成这项工作。

构建可靠的复杂系统

有了自动化测试这个强大的盟友，我们就能放心大胆地开发复杂功能了。就像有了坚实的地基，我们才能建造摩天大楼。

当然，要建造真正宏伟的建筑，我们还需要合理的规划和设计。在软件工程中，我们可以通过：

- 分包：将函数封装重用
- 分层：在逻辑上划分层次，每层代码只专注于自己的逻辑
- 业务分割：根据功能类型划分区域
- 微服务：将大型服务拆分成独立运行的小服务

这些方法就像建筑设计中的结构原理，帮助我们构建稳固而优雅的系统。即使项目变得再复杂，也能保持结构的清晰和稳定。

工程化质量保障

软件开发本质上是一个工程问题，需要系统化的方法来保证质量。通过将质量控制融入开发流程的每个环节，我们能够构建出稳定可靠的系统：

1. 自动化测试的持续集成

- 在每次代码提交时自动运行测试套件
- 实时监控系统的健康状态
- 及早发现并解决潜在问题

2. 系统化的质量度量

- 代码覆盖率监控
- 性能指标追踪
- 错误率统计和分析

3. 标准化的开发流程

- 明确的代码审查流程
- 规范的发布周期
- 完善的回滚机制

很多程序员提起软件工程就只想起了冗长繁杂的流程。

过去可能真的是这样，但现在不同了。

我们可以把它们都交给AI来做。AI可以很坚决的执行流程和规定，没有任何怨言。通过将AI的智能与严谨的工程实践相结合，我们既能保持快速迭代的节奏，又能确保系统的可靠性。

请允许我再说一遍。

> 在AI时代，提升质量的成本低得难以置信，你只需要告诉AI：「请为这个新功能创建自动化测试」。

PDTAC循环：最佳实践

一个可以融入日常的最佳实践

懂得一个道理很简单，难得是把它用起来；为了避免出现这种情况，我们针对PDCA循环进行了优化，提出了PDTAC循环。

广受欢迎的PDCA循环，包含四个步骤：计划(Plan)、执行(Do)、检查(Check)、行动(Action),这个经典的戴明环已经指导了无数项目走向成功。

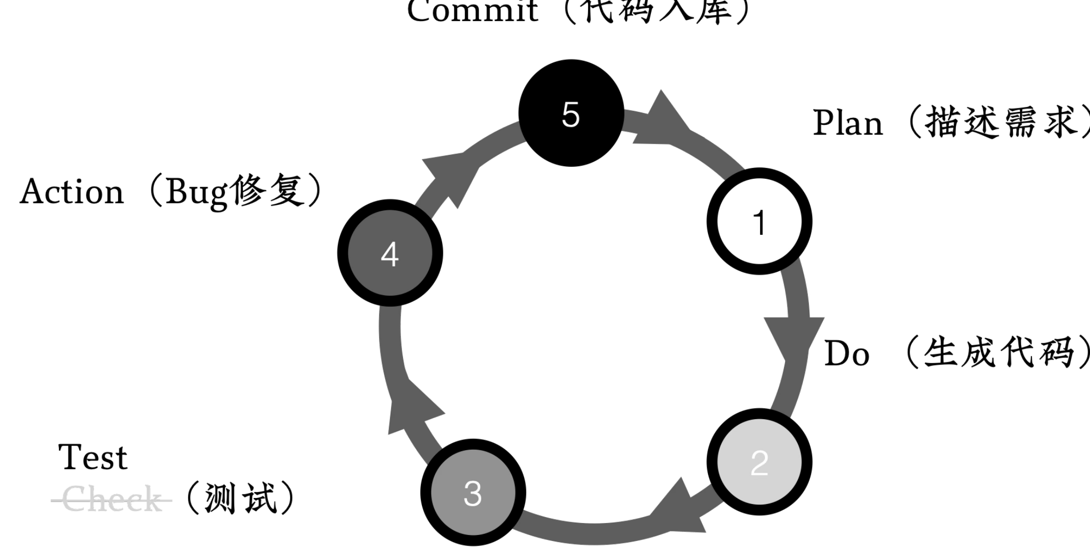
PDTAC

那么PDTAC有什么不同呢?

它保留了PDCA的精髓,但做了两个关键的升级:

1. 首先，将「Check」升级为「Test」,这不是简单的文字游戏,而是赋予了全新的内涵——强调自动化测试和全量测试的重要性
2. 其次，在最后增加了「Commit」环节,确保每一个进步都能及时保存，每一个改进都留下足迹。
3. 没有了

是的，就是如此简单。这是刻意的。因为我们发现对于流程来说，复杂了用的人就会少。



Plan：描述需求

P代表的是Plan，对应到AI自编程中就是描述需求。

让人意外的是，清晰地描述需求，对于很多人来说，可能是一个困难的任务。也许，随着技术和交流方式的变化，现在很多人接触文字的机会变少了，更多的沟通方式转向了视频等形式。能够精准用语言表达需求的能力，反而正在变得越来越稀缺。

具体而准确的描述需求

虽然语言能力不是短时间可以提升的，但我们依然有一些可以提升结果的技巧，比如「具体化」:

- 少说「不能用」，而是提供具体的错误信息；
- 少说「文档」，而是给出具体的扩展名甚至格式demo；
- 少说「数据」，而是指明具体使用的数据库类型和版本

幸运地是，因为AI生成代码的成本极低，所以即使你词不达意，天也不会掉下来，只是浪费了一些时间。通过观察AI的误解，可以知道我们在表达上的问题，从而反向提升我们的语文能力。呃，好像还能一举两得。

使用专业术语和领域知识

另一方面，人类语言极为丰富，但在AI模型理解过程中，这种丰富性反而可能带来挑战。自然语言编程正是一个双刃剑。一方面，它让普通人无需学习编程知识就能与机器沟通；另一方面，由于自然语言的模糊性和宽泛性，模型可能容易产生误解。

为了解决这一问题，我们可以选择使用大模型能够准确理解的语言，从而提升需求描述的精准度。

使用专业术语和领域知识可以有效减少模糊性。例如，在计算机相关的技术领域，使用专业术语或在特定业务场景中的专有词汇，可以将模糊的描述转变为明确的指代，从而帮助模型更精准地理解你的意图。

使用代码和伪代码

如果你会编写代码 —— 没人规定会写代码就必须写代码而不能用自然语言编程不是 —— 还可以将代码或伪代码作为需求描述的一部分。如果需求包含相对复杂的逻辑，用文字难以清晰表达时，伪代码是一个有效的工具。伪代码的格式灵活，你可以通过定义变量、使用循环和条件语句等方式，描述复杂的逻辑。语法错误、中文英文混杂都没有关系，只要能够清楚地传达逻辑结构，都能达到良好的效果。

善用图片

当前很多AI编码工具已经可以支持图片描述（其实主要是模型支持了）。在很多情况下，尤其是和界面相关的业务中，「一图胜千言」并不夸张。记得好好利用这个功能。

Do：生成代码

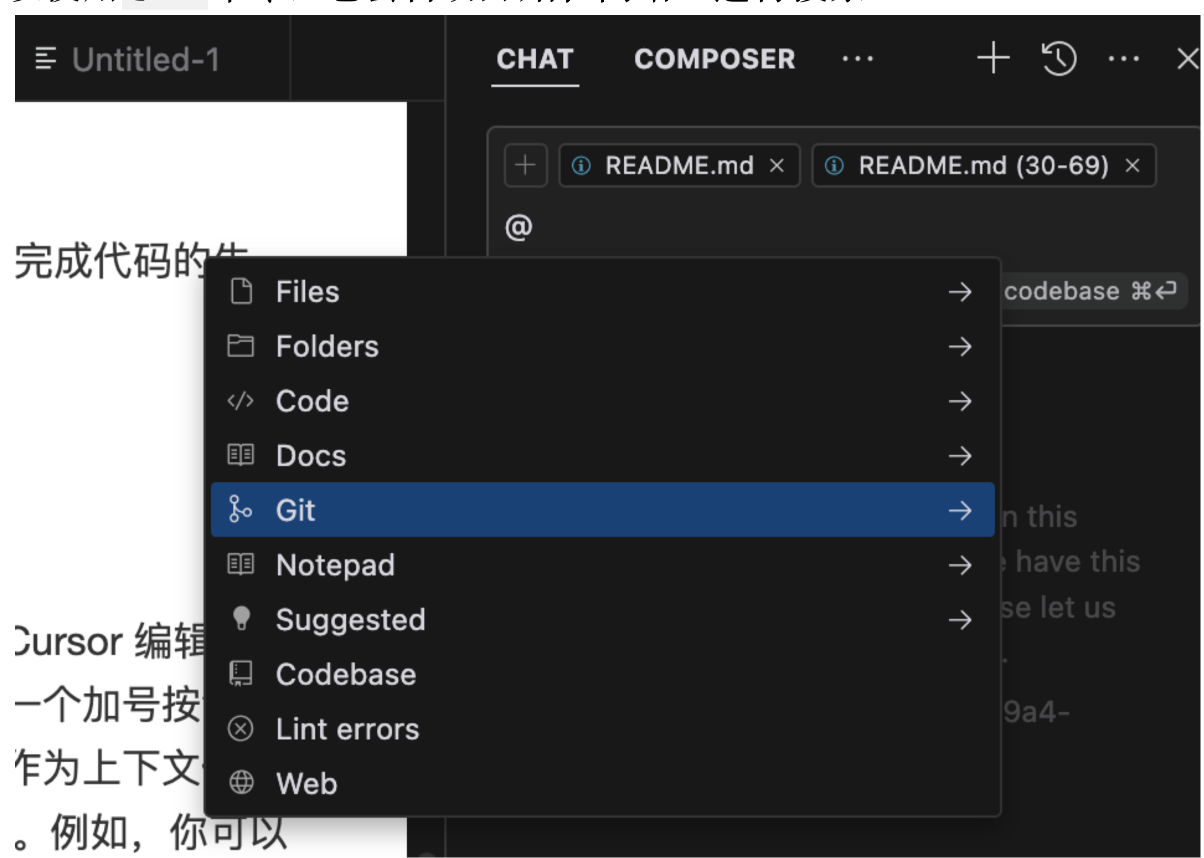
「Do」阶段对应的是代码生成。通常来说，点击一个按钮就能完成代码的生成。

然而，在 Cursor 编辑器中，还是有一些小技巧的。

手动控制上下文

首先，在生成代码时，除了你输入的文本和当前打开的文件，Cursor 编辑器本身还提供了一个上下文选择功能。在输入框的上方，你会看到一个加号按钮，点击它之后，你可以选择与当前问题相关的文件，这些

文件会被作为上下文使用。此外，在你输入`@`符号时，编辑器还会显示更多的上下文选项。例如，你可以选择一个目录、一个文件，或者选择特定的Git版本作为参考。如果你希望 Cursor 通过互联网进行搜索，你可以使用`@web`命令，它会自动开始在网络上进行搜索。



如果你有一些外部文档需要 Cursor 参考，也可以直接粘贴链接，编辑器会自动抓取文档内容并作为上下文提供给你。这些功能大大提高了工作的便利性，使得 Cursor 能够更加智能地理解和响应你的需求。

代码生成工具

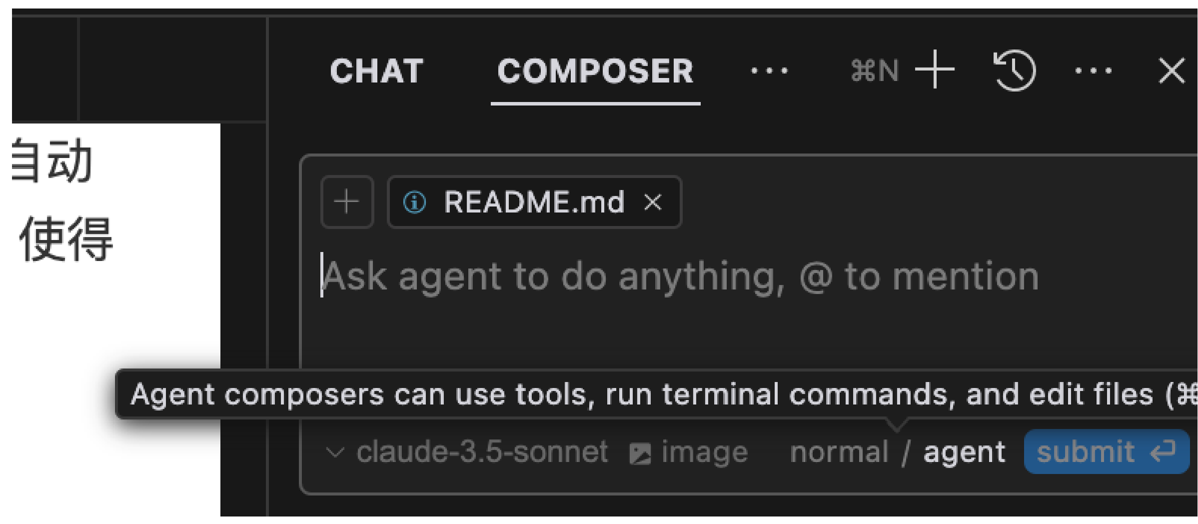
在代码生成方面，目前 Cursor 提供了两种工具：一种是 Chat，另一种是 Composer。

在Chat中，我们可以逐步提出问题，Cursor 会根据你的每个问题逐一生成代码，并将新代码显示在聊天回答中。生成后的代码会有一个「Apply」按钮，点击该按钮后，Cursor 会将新的代码与当前打开的文件进行对比，并将需要修改的部分自动应用进去。

如果你的任务只是修改少量内容，这种方式非常简单高效。然而，当任务变得更复杂，比如需要创建多个文件、执行多个命令或同时修改多

个文件时，Chat工具就显得有些繁琐。因为你需要进行一系列手动操作，效率会受到一定的影响。

为了更高效地处理复杂任务，Cursor 还提供了 Composer 工具。在这个工具里边，Cursor会执行多步操作。我们可以选择「Agent」模式。



在这个模式下，当我们将编程任务发送给 Cursor 时，智能体会自动执行任务，并监测结果，用于判断是否需要进一步操作。如果任务没有完全完成，智能体会继续执行后续操作，直到完成为止。

举个例子，如果你要求 Cursor 创建一个目录并执行某些命令，它会先自动输出创建目录的命令，等待你的确认。确认后，它会继续执行后续操作。对于创建文件和修改文件等任务，智能体无需确认，操作会完全自动化完成。这种方式非常方便，它可以帮助你完成绝大部分工作，而你只需关注结果的输出。

在执行过程中，如果智能体转向了我们不希望的方向，可以直接在输入框中提出修改要求。Cursor 会根据指示实时进行调整，确保最终结果符合预期。

Test：测试

关于测试，我们在前边已经用专门的章节进行过讨论，只需牢记「全量」和「自动化」两个关键词就好，这里就不赘述了。

Action：Bug修复

再下来是「Action」阶段，对应到Bug修复。

一般Bug的修复

通常来说，如果我们使用 Cursor 的Composer工具，在生成和修改代码后，它会自动检查静态语法分析的错误。一旦发现Bug，Cursor 会自动尝试修复，这是非常便捷的功能，因此我们推荐大家尽可能使用它。

顽固Bug的修复

不过，在实际使用过程中，我们也可能会遇到一些顽固的Bug。遇到这些问题时，就需要我们手动介入处理了。

如何识别顽固Bug

首先，我们需要识别 Cursor 是否已经陷入了无法解决的问题。

在我们提出需求后，我们仍然需要观察 Cursor 在执行过程中做了什么。可以不需要过于关注细节，但至少要了解它的整体方向。

如果我们发现 Cursor 提出的解决方案开始重复，例如它先提出了解决方案A，发现A无效后再改为B，B依然无效后又回到A，这时它就陷入了一个死循环。此时，我们就需要手动干预，避免它继续在这个循环中浪费时间。

首先要**终止这个循环**，并尝试为 Cursor **提供更多的上下文信息**，帮助它理解问题。检查一下是否有资料没有提供，或者是否遗漏了关键的代码库部分，尤其是那些与我们要解决问题直接相关的内容。

另外，要注意，**不要将新的问题加在原有问题的后边进行追加讨论**。这样做可能导致上下文过长，或者会违反「每次只做一件事」的原则。

更好的做法是，开启一个新的聊天（Chat）或编排（Composer）窗口，将新的问题单独提出来，然后观察新的结果。

另一个解决方案是**更换模型**。目前 Cursor 使用的模型是Cloud 3.5 sonnet，虽然它较为智能，但对于一些复杂任务，我们也可以尝试使用O1模型。

O1模型通过多次思考来提升其智能，尤其在面对复杂任务时，可能会提供不一样的解决方案。而且，两个模型有不同的思维维度，换个角度思考，可能就能找到问题的突破口。

如果上述方法仍然无法解决问题，我们可以借助外部帮助。你可以将问题提取出来，送到其他大模型中进行询问，或者根据情况进行人工干预，手动修正问题。

总体而言，遇到此类问题的情况相对较少。在我几个月的AI自编程经历中，这样的情况也只遇到过几次。通常这种问题会出现在文档不完整，或者文档版本混乱的情况下。有时候，项目文档混乱到即便是人类也难以理清，更别提 Cursor 了。然而，在真实世界的编程中，这种项目可不少。

Commit：代码入库

将Commit作为循环的一大阶段只有一个原因 —— 防止忘了它。

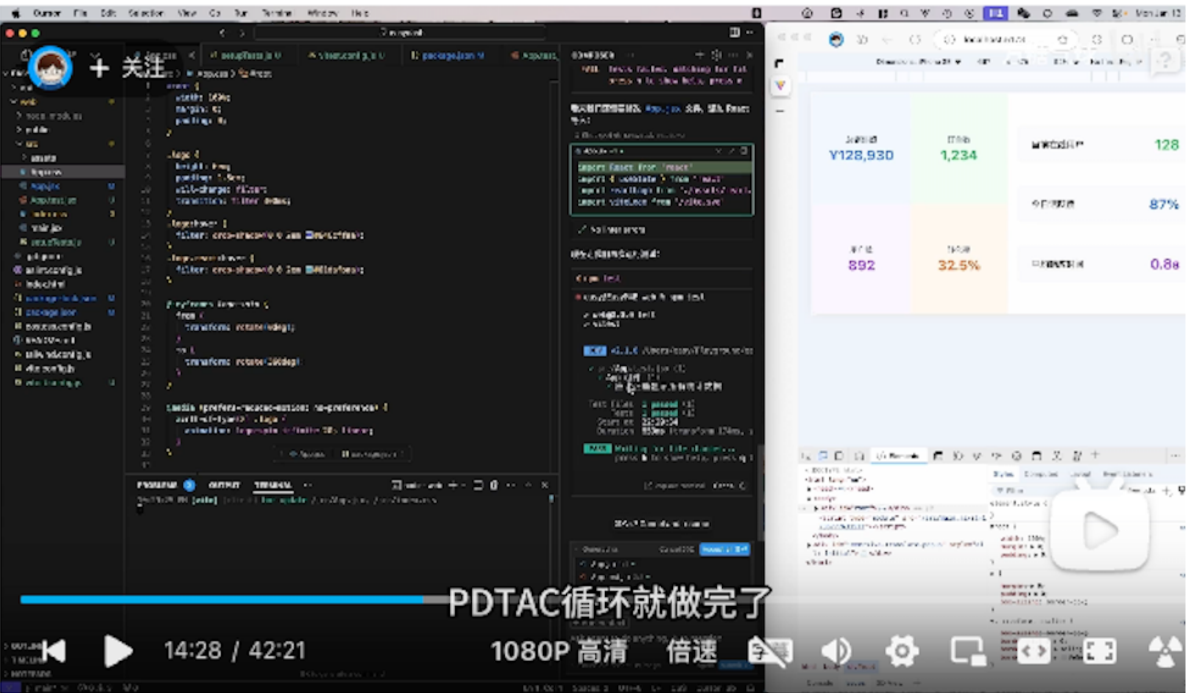
请记住作为一个专业程序员的常识（雾）：

火灾自救常识
1. git commit
2. git push
3. 逃离建筑

实践教程

> 纸上得来终觉浅，绝知此事要视频。

为了进一步说明PDTAC循环在实际项目中的使用，我们录制了一个视频教程。欢迎观看，记得订阅点赞哈。



- Bilibili
- Youtube

FAQ

暂无，将根据issue整理。