

Claude Code 科研手记

用 Claude Code 辅助科研写作的经验与教训

作者：晨瀚宇（chanw）

时间：2026 年 4 月

版本：1.0

实验做得好是第一步，写好是第二步。这本书讲的是第二步怎么走快一点。

© 2026 晨瀚宇（chanw）. 保留所有权利。

版权声明

书名：Claude Code 科研手记——用 Claude Code 辅助科研写作的经验与教训
作者：晨瀚宇（小红书：chanw）
版本：精选版 v1.0（2026 年 4 月）

© 2026 晨瀚宇. 保留所有权利。

本书内容（包括但不限于文字、代码、图表、排版设计）均为作者原创，受中华人民共和国著作权法保护。

禁止事项：

- 未经作者书面授权，禁止以任何形式转载、复制、传播本书的全部或部分内容
- 禁止将本书内容用于商业用途或二次售卖
- 禁止去除、修改本书中的版权标识和作者署名

允许事项：

- 个人学习和研究用途的合理引用（需注明出处）
- 在社交媒体上分享本书的购买链接

如发现侵权行为，请联系作者。本书已进行版权登记。
购买渠道：小红书搜索「chanw」

写在前面

我的研究方向是一个交叉学科。说白了就是一个管理学专业的学生，自己摸索着学会了写代码。

这个「摸索」的过程挺长的。2021 年刚开始学 Python 的时候，我在 Jupyter Notebook 里一行行跑数据分析，遇到语法问题就去 CSDN 上查。后来去事务所、企业实习，发现财务和审计部门根本没有 Python 环境，只有 Excel，于是转去写 VBA，做了几个嵌在工具栏里的小插件。后面大模型火起来之后，刚好时间点和我读研相契合，我开始用 ChatGPT 辅助学算法和深度学习，每天都跟它对谈。再后来做微调、做 RAG、做多智能体系统，一路折腾下来，代码能力就是这么攒起来的。

我讲这段经历，是想说清楚一件事：我不是计算机科班出身，也不是什么技术大佬。我的起点是 Excel 和 VBA，中间走了很多弯路。如果你也是非技术背景的研究生，正在犹豫要不要用 AI 工具来辅助论文写作，我的经历可能比一个程序员写的教程更有参考价值。因为你会踩的坑，我大概率踩过。

去年这个时候，我的论文被导师发回来第三次了。批注就一句话：「逻辑不清晰，重写。」98324 个字，我不知道从哪里下手。室友当时在旁边调代码，随口说了一句：你去试试 Claude Code 吧。我没当回事，以为又是什么套了壳的 ChatGPT。拖了两周，论文还是那个状态，最后决定试一下。这一试就是大半年。

在往下讲之前，有一件事我想先说清楚。

AI 辅助科研写作，不是让你平地起高楼。你的研究问题得是你自己想的，实验得是你自己跑的，数据得是你自己的，结论得是你从数据里读出来的。这些东西没有任何工具能替你做，也不应该替你做。你要是拿它来编数据、编结论，先不说这违反学术规范，编出来的东西质量也很差，逻辑链对不上，稍微懂行的人一眼就能看出来。Claude Code 本身遇到这种请求也会拒绝，它会提醒你这些内容需要你自己提供。

那 Claude Code 能做什么呢？**它帮你把已有的研究成果更高效地呈现出来。**统一格式、理顺逻辑、打磨表达、执行导师的修改意见。它把你从繁琐的文字劳动里解放出来，让你有更多时间去想真正重要的问题。

导师教给我一个观念，我在用 Claude Code 的过程中体会越来越深：**实验做得好只是第一步，你得把它写好、包装好，别人才能看懂你做了什么、为什么有价值。**很多同学的问题不是研究不行，是表达不行。好的结果被平庸的写法埋没了，评审看完不知道你的贡献在哪里。Claude Code 能帮你把实验结果组织成清晰的表述，能帮你统一前后章节的逻辑线，能帮你把导师说的「拔高一下」落实到具体的句子上。但前提是你自己先想清楚了：这段话要传达什么信息、为什么放在这个位置、和前后文怎么衔接。想清楚这些，它帮你执行起来很快；没想清楚就让它写，出来的东西一定是散的。

这本书里不光有工具的用法，也有我从导师那里学到的写作思路。这些思路是我在大半年的对话中，一边用 Claude Code 改论文、一边消化导师反馈的过程中，慢慢想明白的。**工具是手段，把研究成果写清楚、讲明白，才是目的。**

写这本书的时候，我翻了一下自己的使用记录。Claude Code 每次对话都会存下来，我在 27 个项目里累计开了 1000 多次会话，对话来回超过 12 万条。光是论文这一个项目，就有 382 次会话、6 万多条对话记录。这些记录加上导师开会的录音转写、每次修改的版本对比，就是这本书的素材库。如果你现在知道 AI 能帮忙写论文，但不知道具体怎么用，或者用了觉得效果一般，这本书就是写给你的。

晨瀚宇，2026 年春

© 2026 晨瀚宇 未经授权禁止转载

目录

写在前面 i

第一部分　启程——认识 Claude Code 1

第 1 章　Claude Code 是什么，为什么它适合写论文 2

1.1 用网页版 AI 改论文的痛点 . . . 2
1.2 一个完整的例子：跨文件术语排查 . . . 3
1.3 账号注册、订阅方案与安装配置 . . . 4
1.4 第一次使用与安全注意事项 . . . 5

第 2 章　上下文与记忆：让它记住你的研究背景 8

2.1 上下文窗口：为什么它会「忘记」 . . . 8
2.2 CLAUDE.md：给 AI 写一份永久说明书 . . . 9
2.3 Memory 系统：跨会话记忆 . . . 11
2.4 实操：创建你的第一个 CLAUDE.md . . . 13

第 3 章　提示词的实战经验：怎么和它说话 15

3.1 模糊指令与精确指令的差距 . . . 15
3.2 先确认方案，再动手 . . . 16
3.3 术语保护：不让它擅自替换你的专业词 . . . 16
3.4 打断和纠正的方法 . . . 17
3.5 一次会话只做一件事 . . . 17
3.6 把导师反馈翻译成可执行的指令 . . . 18
3.7 实操：用提示词模板改写一段论文 . . . 19

第二部分　核心技能——论文全流程 20

第 4 章　文献调研与管理 21

4.1 用 AI 整理研究现状 . . . 21
4.2 批量文献核查 . . . 24
4.3 文献综述的写法辅助 . . . 25
4.4 实操：/pdf + /arxiv + /citation-management . . . 27

第 5 章　章节写作 29

5.1 从研究问题到章节框架 . . . 29
5.2 逐节写作：怎么给 AI 喂材料 . . . 30
5.3 学术语气与术语控制 . . . 32

© 2026 晨瀚宇 未经授权禁止转载

5.4 导师反馈的处理流程 . . . 33
5.5 实操：用 /docx 修改 Word 章节 . . . 34

第 6 章 图表制作 35
6.1 用代码生成学术图表 . . . 35
6.2 流程图与架构图 . . . 38
6.3 图表规范自检 . . . 42
6.4 实操：用 /drawio 画研究框架图 . . . 44

第 7 章 引用与参考文献 47
7.1 BibTeX 格式管理 . . . 47
7.2 引用核查：自动找出错误引用 . . . 49
7.3 批量格式统一（GB/T 7714） . . . 51
7.4 实操：用 /citation-management 核查引用 . . . 53

第 8 章 格式排版与文件管理 57
8.1 Word 文档操作的风险 . . . 57
8.2 备份策略 . . . 59
8.3 从 Word 到 LaTeX 的迁移 . . . 61
8.4 实操：用 /markitdown 转换文档格式 . . . 63

第三部分 进阶能力——解锁更多可能 65

第 9 章 Skills：给 AI 装上专属工具 66
9.1 什么是 Skill，和普通提示词的区别 . . . 66
9.2 安装现成的科研 Skill . . . 67
9.3 写一个属于你的 Skill . . . 69
9.4 实操：安装 3 个科研必备 Skill . . . 70

第 10 章 并行 Agent：批量任务的加速方案 71
10.1 什么任务适合并行 . . . 71
10.2 引用核查的并行实战 . . . 73
10.3 长任务中断与接续 . . . 74
10.4 实操：全文术语一致性检查 . . . 75

第 11 章 Hooks：自动化触发器 77
11.1 什么是 Hook . . . 77
11.2 改文件前自动备份 . . . 78
11.3 写完章节自动格式检查 . . . 80
11.4 实操：配置自动备份 Hook . . . 82

第 12 章 MCP 工具扩展 85

© 2026 晨瀚宇 未经授权禁止转载

12.1 什么是 MCP . . . 85
12.2 接入 Zotero 文献库 . . . 86
12.3 接入学术数据库：OpenAlex MCP . . . 87
12.4 实操：用 /pyzotero 同步文献库 . . . 89

第四部分　思维方式——工具背后的底层逻辑　92

第 13 章 导师教会我的科研写作观　93
13.1 论文逻辑要让读者不绕晕 . . . 93
13.2 上下文连贯的具体写法 . . . 95
13.3 AI 能帮你写，但不能帮你想 . . . 97

第 14 章 与 AI 协作的正确心态　99
14.1 你是主导，AI 是执行 . . . 99
14.2 对结果负责的永远是你 . . . 100
14.3 怎么持续提高协作效率 . . . 102

第 15 章 从一篇论文到科研工作习惯　104
15.1 建立个人知识库 . . . 104
15.2 跨项目复用经验 . . . 105
15.3 未来展望 . . . 106

附录 A　常用提示词模板大全　107

附录 B　Claude Code 快捷键速查表　109

附录 C　科研必备 Skill 推荐列表　110
C.1 去哪找 Skill：skills.sh 与安装方法 . . . 110
C.2 我怎么找和用 Skill：一些实际经验 . . . 111
C.3 已安装 Skill 完整列表 . . . 112

附录 D　CLAUDE.md 科研版模板　116

附录 E　常见错误与解决方案速查　118

附录 F　从论文 PDF 到组会 Beamer：paper-to-beamer Skill 实战　120
F.1 为什么需要这样一个 Skill . . . 120
F.2 Skill 的整体设计 . . . 121
F.3 安装这个 Skill . . . 122
F.4 第一次跑：完整的对话演示 . . . 122
F.5 我自己踩过的几个坑 . . . 123
F.6 进阶阅读：附赠的 paper_writeup.pdf . . . 124

© 2026 晨瀚宇 未经授权禁止转载

第一部分

启程——认识 Claude Code

© 2026 晨瀚宇 未经授权禁止转载

第一章 Claude Code 是什么，为什么它适合写论文

内容提要

- 网页版 AI 改论文的核心痛点：喂材料费时、跨文件操作繁琐、改完还要手动粘贴
- Claude Code 的核心优势：直接读写本地文件，终端里一句话跨文件操作
- 账号注册、订阅方案选择与安装配置完整流程
- 第一次使用的四条注意事项：它会犯错、先备份、一次一件事、指令要具体

论文快提交的那个月，导师一口气给了 40 多条修改意见。我打开那份批注文件的时候，滑了好几屏才到底。有的意见很具体（「第三章表 4-2 的最小值标错了」），有的很笼统（「摘要太琐碎，拔高一下」），还有几条互相矛盾的（一处说展开不够，另一处说太啰嗦）。

按照以前的做法，我得一条一条读，自己判断每条要改哪个文件的哪个位置，然后打开 Word 逐段改。40 多条，保守估计两周。

那天晚上我做了一件事：把这 40 多条意见按章节分组，每条翻译成 Claude Code 能执行的具体指令，然后一章一章交给它做。4 天改完了。

这件事能成，靠的不是它有多聪明。它会犯错，而且犯得很隐蔽，后面会详细讲。它真正帮到我的是另外一件事：直接读我的论文文件，自己找到要改的位置，改完给我看前后对比。40 多条意见里那些机械性的工作，比如统一术语、调整格式、补过渡句、改口语化表述，它做起来比我快得多。省下来的时间，我用来想几条真正需要动脑子的修改意见。

这一章把三件事讲清楚：它是什么、怎么装、用的时候要注意什么。

定义 1.1 (Claude Code)

Claude Code 是 Anthropic 公司开发的命令行 AI 助手。它运行在你的电脑终端里，能直接读取和修改本地文件，执行终端命令，调用外部工具。与网页版 AI 不同，它不需要你把文件内容复制进对话框，而是直接操作你的文件系统。论文写作中最常用的能力是：跨文件检索和修改、批量统一格式和术语、执行代码和编译 LaTeX。

1.1 用网页版 AI 改论文的痛点

在用 Claude Code 之前，我也试过用 ChatGPT 改论文。管用，但很累。累的不是 AI 的部分，是喂材料的部分。

每次打开 ChatGPT 的新对话，第一件事是粘一段背景介绍：「我的论文方向是 XXX，研究问题是 YYY，目前写到第三章，用的是双向固定效应模型……」。我有段时间把这段话存在手机备忘录里专门用来粘贴，算了一下，一周要粘七八次，每次三四百字。这些字不会进论文，纯粹是在告诉 AI「我是谁」。

更麻烦的是跨文件的任务。比如导师说「第三章和第四章的变量定义对不上」，你得自己打开两个文件，分别找出相关段落，复制到对话框里，它才能帮你看。我论文有 4 章正文加 1 个参考文献文件，光是准备材料就得十几分钟。如果要做全文术语一致性检查，你得把每个文件相关的段落一段段复制过去，来回切换窗口，一个下午就这么耗掉了。

© 2026 晨瀚宇 未经授权禁止转载

还有一个问题：网页版 AI 给你的是一段文字，你得自己复制回去、替换原文、检查前后有没有衔接问题。改一处还好，改十几处就容易出错。

这些痛点，Claude Code 全部解决了。它跑在终端里，你在论文文件夹里启动它，它直接能读到你的所有文件。你说「帮我检查第三章和第四章的变量定义有没有矛盾」，它自己去读、自己对比、把不一致的地方列出来。你说「把全文的' 因变量' 统一改成' 被解释变量'」，它自己扫 4 个文件，2 分钟改完，每处改动都标得清清楚楚。改完直接写回文件，不用你复制粘贴。

它还能执行终端命令。你让它把 Word 转 LaTeX，它自己调用 pandoc；你让它编译一下看看有没有报错，它自己跑 `xelatex`。它能做的事情远不止「看和写」。

我用了大半年，前后开了一百多次会话，慢慢发现 Claude Code 最能帮上忙的场景，恰好就是论文写作中最机械、最耗时间的那些：跨文件检查一致性、批量统一格式和术语、把导师的笼统批注落实到具体段落。这些事以前每件都要花一两个小时，现在几分钟就能做完。

1.2 一个完整的例子：跨文件术语排查

光说「它能读你的文件」太抽象，不如用一个真实的任务走一遍完整过程。

我论文里「被解释变量」和「因变量」混着用。计量经济学教材用前者，统计学教材用后者，我写的时候没注意，两种说法前后章节混着来。导师开会时点名批评了这个问题。

如果用网页版 AI 来做这件事，流程是这样的：打开 ch1_intro.docx，Ctrl+F 搜索「被解释变量」，记下出现了几次、在哪些段落；再搜「因变量」，同样记下来。然后打开 ch2、ch3、ch4，重复同样的操作。搜完之后，把所有结果整理到一起，逐条判断哪些该改哪些不该改（有些地方是引用别人论文的原文，不能改）。最后回到每个文件里逐个替换。4 个文件搜一遍，至少半小时。

用 Claude Code，我在终端里输入了一句话：

```
读一下 chapters 文件夹里的所有 .docx 文件，
列出「被解释变量」和「因变量」各出现了多少次，
分别在哪个文件的哪一段。
```

两分钟，它扫完 4 个文件，给了我一张清单：ch2 用了 14 次「被解释变量」3 次「因变量」，ch4 用了 8 次「被解释变量」21 次「因变量」，还标出了每一处的上下文。我看了一遍，确认没有需要保留原样的引用，然后说「统一改成被解释变量」。它改完，每处改动都列出了改前改后的对比。

整个过程不到 5 分钟。以前这件事我得自己通读全文逐段比对，一两个小时都不一定改干净。

下面这张表是这个任务的完整对比：

表 1.1: 同一个术语排查任务的操作对比

用网页版 AI	用 Claude Code
逐个文件打开、Ctrl+F 搜索、手动记录位置	一句话，它自己扫全部文件
搜到的结果要自己汇总成表	直接给你一张按文件分组的清单
判断哪些该改、哪些是引用原文不能改，需要自己逐条看	标出每处的上下文，判断起来快得多
替换要回到每个文件里逐个操作	确认后它直接改，改完给对比
全程约 30 分钟	全程约 5 分钟

这个例子展示了 Claude Code 最核心的能力：它直接读你的文件、自己找到要改的地方、改完告诉你结果。

© 2026 晨瀚宇 未经授权禁止转载

你不用打开任何文件，不用复制任何段落，不用在 AI 和 Word 之间来回切换。

其他常见的论文任务也是类似的体验。导师说「摘要太琐碎，拔高一下」，你把这个要求翻译成具体指令（去掉具体数值、改为定性概括），Claude Code 直接读你的摘要文件、按要求改、给你看改前改后。全部参考文献要逐条核查格式，你让它派 3 个并行 Agent 分批做，40 分钟搞定（这个故事第十章详细讲）。甚至把整篇论文从 Word 转成 LaTeX，它自己调用 pandoc 完成转换，在 Overleaf 上一次编译通过。

这些事情如果用网页版 AI 做，每件都要你花大量时间准备材料、来回复制粘贴。Claude Code 帮你省掉的就是这部分时间。

1.3 账号注册、订阅方案与安装配置

Claude Code 需要 Anthropic 账号。安装本身不难，主要是账号和网络容易卡人，我把流程和常见问题都列在这里。

1. 费用

付费分三档：

表 1.2: Claude Code 订阅方案

方案	月费	用量	说明
Pro	$20/月	基础用量	入门够用，先试几天
Max 5x	$100/月	Pro 的 5 倍	论文冲刺期，每天用 2 小时以上建议升级
Max 20x	$200/月	Pro 的 20 倍	高强度长期使用

建议从 Pro 开始。用几天自然知道够不够。如果经常碰到用量上限提示，再升 Max 5x。

说一下我自己的使用节奏。平时每天用一两个小时，Pro 够了。但论文冲刺期不一样：有时候一个下午要做全文一致性检查、引用核查、再加章节修改，Pro 用量半天就到顶了。那段时间我升了 Max 5x，用了两个月。算下来 200 美元，但它帮我省掉的手工时间远不止这个数。

还有一个实际的考虑：Claude Code 跑长任务的时候可能碰到 rate limit（用量限制），任务会被打断。如果你正在做批量核查之类的事，打断一次就要从头来。升到 Max 5x 之后，碰到 rate limit 的频率明显降低。论文冲刺期我选择升级，主要就是不想被打断。

国内用户注意：Anthropic 不支持国内银行卡，需要 Visa 或 Mastercard，或通过第三方渠道充值。搜索「Claude 订阅国内支付」能找到教程，这里不展开。

2. 安装

macOS / Linux

打开终端（macOS 在启动台搜「终端」就能找到），运行这一行：

```
curl -fsSL https://claude.ai/install.sh | bash
```

装完输入 `claude --version`，能看到版本号就说明成功了。

如果你之前没怎么用过终端，不用紧张。终端就是一个输入文字命令的窗口，你把上面那行命令复制进去、按回车，它会自动下载安装。不需要理解命令的每个部分。

Windows

打开 PowerShell（按 Win 键搜「PowerShell」），运行：

```
winget install Anthropic.ClaudeCode
```


© 2026 晨瀚宇 未经授权禁止转载

装完重开终端，输入 `claude --version` 验证。

笔记 Windows 用户建议用 PowerShell 或 Git Bash，不要用 CMD。Claude Code 依赖一些 Unix 工具链，CMD 下会遇到命令格式和路径问题。我身边有同学在 CMD 里装了半小时没成功，换到 PowerShell 一分钟就好了。

3. 登录

第一次运行 `claude`，浏览器会自动弹出登录页面，用 Anthropic 账号授权即可。登录成功后回到终端就能看到对话界面。界面很简洁：上面是当前工作目录的路径，下面是一个输入框，你在里面打字就行。

4. 常见报错和解决办法

我自己和身边同学装的时候碰到过几种情况，把解决办法都记在这里，省得你再踩一遍。

（1）连不上、登录页面打不开

Claude Code 需要访问 Anthropic 的服务器，国内网络环境可能会遇到连接问题。这个问题的解决方法网上有很多教程，搜索「Claude Code 国内使用」就能找到，这里不展开。确保网络通畅之后再继续后面的步骤。

（2）*Permission denied* 或 *command not found*

macOS 碰到权限问题不要用 `sudo`，先创建用户目录：`mkdir -p ~/.local/bin`，然后重新装。如果装完终端不认识 `claude` 命令，把安装路径加进 PATH：

```
echo 'export␣PATH="$HOME/.local/bin:$PATH"' >> ~/.zshrc
source ~/.zshrc
```

（3）运行时报网络错误

对话中途断开，通常是网络波动导致的。确认网络恢复后重新输入 `claude` 就能接着用，Claude Code 会记住上一次的对话状态。

5. 保持更新

建议每两周跑一次安装命令更新到最新版本。更新只替换程序本身，不会丢失你的数据或配置。

1.4 第一次使用与安全注意事项

装好之后，打开终端，进到你的论文文件夹：

```
cd ~/Desktop/thesis # 换成你自己的路径
claude
```

第一句话建议从只读任务开始，比如：

```
帮我列一下这个文件夹里有哪些文件，每个文件大概是什么内容。
```

它会扫描文件夹，列出所有文件，还会猜每个文件的用途。你什么都没告诉它，它自己读的。确认它能正确读到你的文件之后，再试修改类的任务。

笔记 文件夹里有 PDF 文献的话，在 Claude Code 里输入 `/pdf` 可以直接提取全文内容，不用手动复制，也不用装别的工具。查文献和核对引用的时候很方便。

安装 docx 技能：直接修改 Word 文档

大多数同学的论文都是 Word 格式。Claude Code 本身能读写文件，但处理.docx 这种压缩格式需要额外的技能（skill）支持。Claude Code 有一个叫 `/docx` 的技能，装上之后就能直接读取和修改 Word 文档的内容，不需要你先把文字复制出来。

© 2026 晨瀚宇 未经授权禁止转载

安装方法是这样的。Claude Code 的技能本质上是一个文件夹，放在 `~/.claude/skills/` 目录下。你可以用 Claude Code 自己来安装：

```
帮我安装 docx 这个 skill。
```

它会自动创建目录、下载配置文件。你也可以手动把别人分享的 skill 文件夹复制到 `~/.claude/skills/docx/` 下面，重启 Claude Code 就能识别。装好之后，在对话里输入 `/docx` 就能调用：

```
/docx 读一下 thesis.docx 第三章的内容
```

或者直接让它修改：

```
/docx 把 thesis.docx 第三章第2节里的「因变量」
全部替换成「被解释变量」
```

用 `/docx` 修改 Word 文档有一个好处：它理解文档的结构（标题、段落、表格），不是简单的文本替换。比如你让它「把第三章所有三级标题加粗」，它知道哪些是三级标题，不会把正文里碰巧长得像标题的句子也加粗了。

但 Word 文档有一个风险要提前说清楚：.docx 文件本质上是一个压缩的 XML 包，Claude Code 修改它的时候是在改底层的 XML。中文全角字符、特殊格式、跨多个 XML 节点的文字，都可能导致修改出错，严重的话文件直接打不开。这就是为什么后面「第二件事」会强调备份。

我的建议是：如果你的论文还在用 Word，装上 `/docx` 技能，配合备份习惯一起用。每次改之前先备份，改完打开 Word 验证一下能不能正常显示。如果你打算长期用 Claude Code 写论文，考虑在合适的时候转成 LaTeX，稳定性会好很多（第八章详细讲）。

试完这几个场景，你大概能感受到它的能力了。下面说几个很重要的事情，都是我用了大半年之后总结出来的教训，建议从第一天就注意。

注 第一件事：它会犯错，而且犯得很隐蔽。

这是整本书最重要的一个提醒，所以放在最前面说。

有一次我让它帮我调整一段方法论描述的语气，改得更正式一些。它改完我扫了一眼，句子确实通顺多了，就存了。过了三天，导师在那段话旁边标了个批注：「这个表述不对，' 双向固定效果模型' 是什么？我们一直用的是' 双向固定效应'。」

我回去一查，是 Claude Code 在「润色」的时候把「双向固定效应」改成了「双向固定效果」。意思差不多，但我们领域有固定用法，改了就是错的。最糟糕的是我当时没发现，因为它只改了一处，藏在一大段修改里，肉眼很容易漏掉。

还有一次更隐蔽。我让它帮我统一文献综述的写法，它在修改过程中悄悄删掉了一个引用的方括号编号 [17]，导致那段话变成了没有出处的断言。学术论文里，一句话如果没有标出处，轻则被导师打回，重则被查重系统标记。如果不是我后来逐句检查对比，这个错误就直接交上去了。

每次让 Claude Code 改完东西，不要只看它给你的总结，自己把改前改后逐行对比一遍。它的总结只反映它自己以为做了什么，不一定和实际修改一致。

注 第二件事：改文件之前，必须先备份。

Claude Code 直接修改你的文件，没有自动备份，没有撤销。如果它改错了，原文件就没了。

我有一次让它帮我调整 Word 文档的格式，结果 XML 结构被破坏，文件打不开了。因为没有备份，只能找之前的邮件附件恢复，浪费了一整天。从那以后，我养成了一个习惯：让它动手之前，先跑一行备份命令：

© 2026 晨瀚宇 未经授权禁止转载

```
cp thesis.docx "thesis_backup_$(date␣+%Y%m%d).docx"
```

如果你用的是 LaTeX 而不是 Word，情况会好很多。LaTeX 是纯文本文件，可以用 git 做版本管理，每次修改都有记录，随时可以回退。这也是我后来把整篇论文从 Word 转到 LaTeX 的原因之一（第八章会详细讲这个过程）。

注 第三件事：一次只做一件事。

这是我花了很长时间才学到的一个教训。刚开始用的时候，我经常在一次对话里同时塞好几个任务：「帮我改第三章的逻辑，顺便把引用格式统一一下，再检查一下术语」。结果要么它做了前两个忘了第三个，要么做到一半碰到用量限制被打断，前面做的就白费了。

一次对话只做一件事。如果你的需求里有超过两个「然后」，就该拆成多次对话。每次目标明确，做完就存，即使中途被打断也不怕。

注 第四件事：你说的话要像验收标准。

模糊的指令会得到模糊的结果。比较一下这两种说法：

```
# 模糊的
帮我改一下第三章。


# 清晰的
修改第三章第2节，要求：
1. 不改变原有论点
2. 用学术中文，不出现口语词
3. 以下术语不得修改：「被解释变量」「双向固定效应」「空间杜宾模型」
4. 完成后给我看修改前后的对比
```

第二种写法告诉它什么能改、什么不能改、改完要给你看什么。你给的约束越具体，它发挥的空间越小，出错的概率就越低。

下面这张表是同一段摘要，分别用模糊指令和精确指令让 Claude Code 修改的真实结果对比：

表 1.3: 模糊指令与精确指令的输出对比（摘要修改）

模糊指令（“帮我改一下摘要”）	精确指令（附术语锁定 + 修改要求）
把「被解释变量」改成了「因变量」，术语不一致	保留「被解释变量」不变，仅调整语序
摘要仍然罗列具体数值（919 条、578 条）	数值概括为定性描述（「样本覆盖面广」）
删掉了一处文献编号，变成无出处断言	所有引用编号原样保留
把「双向固定效应」改成了「双向固定效果」	术语锁定生效，模型名称未被替换

左边那列的每一条，都是我真实踩过的坑。右边是加了约束之后的结果。差别一目了然。

到这里，工具装好了，也用起来了，注意事项也知道了。但你可能已经发现一个问题：每次重新打开 Claude Code，它就忘了你是谁、在做什么研究、导师有什么要求，又要从头交代一遍。下一章讲怎么解决这个问题。

© 2026 晨瀚宇 未经授权禁止转载

第二章 上下文与记忆：让它记住你的研究背景

内容提要

- 上下文窗口的原理：为什么关掉终端它就「忘了」你
- CLAUDE.md：写一份永久说明书，让它每次启动都能读到
- Memory：跨会话保存动态信息，比如导师的最新反馈
- 如何分层组织研究背景、术语表和写作要求

用了一段时间 Claude Code 之后，我发现一个很烦的问题：每次关掉终端再重新打开，它就什么都不记得了。上次告诉它的研究方向、术语表、导师的偏好，全部清零。你又得从头交代一遍「我的论文方向是什么，核心概念有这些，导师要求不能用这些词……」。

前面第一章提过，我有段时间把这段背景介绍存在备忘录里专门粘贴。后来我才知道，Claude Code 本身提供了两套机制来解决这个问题：一个叫 CLAUDE.md，一个叫 Memory。搞清楚这两个东西怎么用之后，重复交代背景的事就再没发生过。

这一章把这两个机制讲清楚，从原理到操作，包括我自己踩过的坑。

2.1 上下文窗口：为什么它会「忘记」

要理解它为什么会忘，先要知道一个概念：上下文窗口（context window）。

定义 2.1 (上下文窗口)

上下文窗口（context window）是语言模型在单次对话中能处理的最大信息量，以 token 计算。你和它说的每一句话、它读的每一个文件、它生成的每一段回复，都在消耗这个空间。Claude Code 默认上下文为 20 万 token（约 12 至 15 万中文字），Max 计划可扩展至 100 万 token。关闭终端后，上下文清空，下一次对话从空白开始。

Claude Code 每次和你对话的时候，能「看到」的信息量是有上限的。你可以把它想象成一张桌子：桌子上能放的文件有限，放满了就得拿掉一些旧的，才能放新的。你和它说的每一句话、它读的每一个文件、它生成的每一段回复，都在占用这张桌子的空间。

一次对话的过程中，这个问题通常不大。Claude Code 的上下文窗口默认是 20 万 token（约 12 到 15 万中文字），如果你用的是 Max 计划，可以扩展到 100 万 token。对大多数论文写作任务来说，20 万 token 就够用了。我一般一次对话做一件事，比如改一节的措辞、检查一章的术语一致性，这些任务的信息量远不到上限。

真正的问题出在「下一次对话」。当你关掉终端、第二天重新打开 Claude Code 的时候，上一次对话的所有内容全部清空了。桌子被收拾干净，它不知道你之前说过什么。你昨天告诉它的研究方向、术语锁定表、导师的特殊要求，全部归零。

我一开始不理解为什么会这样。后来想明白了：Claude Code 本质上是一个语言模型，每次对话是独立的。它没有「长期记忆」这个概念，它能看到的只有当前这次对话里的内容。你上一次对话说了什么，对它来说就像从来没发生过。

这带来一个很实际的痛苦。我论文写了大半年，中间开了一百多次 Claude Code 的对话。每次打开，第一件

© 2026 晨瀚宇 未经授权禁止转载

事都是交代背景：我在做什么方向、用什么方法、哪些术语不能改、导师有什么偏好。这段话我存在手机备忘录里，前后粘了几十次。每次三四百字，花两三分钟，看起来不多，但次数一多就很烦，而且容易漏掉某条约束，漏掉了它就不知道。

有一次我忘了粘术语锁定表，结果就出现了第一章提到的那种术语被篡改的情况。如果那次我粘了术语表，这个错就不会发生。

解决方法也很直接：把那些「每次都要交代的信息」写到一个固定的地方，让 Claude Code 每次启动时自动读取。不用你粘，它自己就能看到。这就是 CLAUDE.md 的作用。

2.2 CLAUDE.md：给 AI 写一份永久说明书

定义 2.2 (CLAUDE.md)

CLAUDE.md 是放在论文项目文件夹根目录的纯文本配置文件。Claude Code 每次在该目录下启动时，会自动读取这个文件的全部内容，相当于每次对话开始前都已经接收了这份说明。适合写入不常变动的持久信息：研究方向、核心方法、术语锁定表、导师的固定要求。

CLAUDE.md 是一个放在你项目文件夹根目录的文本文件。Claude Code 每次启动时，会自动读取这个文件的内容。你把研究背景、术语表、写作要求写在里面，它就每次都能看到，不用你反复说。

1. 它放在哪里

就放在你论文文件夹的根目录。比如你的论文文件夹是 `~/Desktop/thesis/`，那就创建 `~/Desktop/thesis/CLAUDE.md`。

```
# 创建CLAUDE.md
touch ~/Desktop/thesis/CLAUDE.md
```

用任何文本编辑器打开它就能写。它是纯文本的 Markdown 格式，不需要特殊工具。

2. 写什么内容

我建议刚开始先写三样东西就够了：研究方向、术语锁定表、输出格式要求。不用一步到位，后面用着用着再慢慢加。

下面是一个科研论文场景的模板：

```
# 项目说明

## 研究方向
本论文研究XX因素对YY的影响及空间溢出效应。
核心工作包括三部分：
1. 用熵值法构建被解释变量的综合指数
2. 用双向固定效应模型检验直接效应和中介效应
3. 用空间杜宾模型分析空间溢出效应

## 术语锁定表
以下术语不得修改或替换，在任何修改任务中必须保持原样：
- 「被解释变量」（不要改成「因变量」「响应变量」）
```


© 2026 晨瀚宇 未经授权禁止转载

```
- 「双向固定效应」（不要改成「双向固定效果」「TWFE」）
- 「空间杜宾模型」（不要改成「空间回归模型」「SDM」）

## 输出格式要求
- 使用学术中文，不出现口语化表达
- 不使用导师明确禁止的词汇（列在下方）
- 修改任何内容后，给出修改前后的对比
```

3. 一个真实的例子

我自己项目的 CLAUDE.md 写得比上面那个模板详细得多，因为项目本身是一个代码仓库，需要说清楚架构和命令。但核心思路是一样的：告诉 Claude Code 这个项目是做什么的、有哪些关键概念、有什么特殊要求。

我那份 CLAUDE.md 的大致结构是这样的：

```
# CLAUDE.md

## Project Overview
[项目名称]是一个[简要描述]...

## Common Commands
pip install -e .
python -m cli.main
pytest

## Architecture
CLI -> MainGraph -> StateGraph -> Agent Nodes

## Pipeline Phases
1. Phase A（数据预处理）
2. Phase B（核心分析）
3. Phase C（结果整合）
4. Phase D（评估验证）
5. Phase E（输出生成）

## Code Style Notes
- 中文注释用于领域术语，保留不改
- PEP 8 风格
```

这份文件大约有 200 行。写好之后，每次在这个目录启动 Claude Code，它都会先读这份文件。我让它「帮我看一下第三章和第四章的方法描述有没有矛盾」，它已经知道项目背景、知道术语规范、知道代码风格，不用我再说一遍。

4. 术语锁定的效果

这里要单独说一下术语锁定，因为这是我吃过亏才学到的。

有一次我让 Claude Code 帮我改一段话的措辞，它在「优化」的过程中，把论文里一个核心概念的说法换成

© 2026 晨瀚宇 未经授权禁止转载

了另一个近义词。两种说法意思差不多，但我论文里从头到尾用的都是前一种，被它这么一改，这个章节就和其他章节不一致了。

另一次更严重：导师开会时明确说了某个词不能用（因为我论文的研究方法和那个词的严格含义对不上，容易被评审抓住攻击）。但 Claude Code 不知道这个背景，帮我改稿时用了那个禁用词。还好我检查了一遍才发现。

把这些信息写进 CLAUDE.md 的术语锁定表之后，这类问题就没再出现过。Claude Code 每次启动都会读到这些约束，在修改文稿的时候会自动避开这些词。

下面这张表是同一段论文，分别在有术语锁定和没有术语锁定的情况下让 Claude Code 润色的结果：

表 2.1: 有无术语锁定表的润色结果对比

无术语锁定（直接说「润色这段」）	有术语锁定（CLAUDE.md 里写了锁定表）
「被解释变量」被改成「因变量」	「被解释变量」原样保留
「双向固定效应」被改成「双向 FE」	「双向固定效应」原样保留
「空间杜宾模型」被改成「空间计量模型」	「空间杜宾模型」原样保留
自作主张删掉了一处括号注释	仅调整语句通顺度，未删内容

左边那列的每一条，都是没写术语锁定表时真实发生过的。右边是加了锁定表之后的效果。一张表写进 CLAUDE.md，以后每次润色都省心。

笔记 术语锁定表不用写得特别长。先把你论文里最核心的 5-10 个术语列上去，尤其是那些容易被 AI「同义替换」的词。后面发现新的问题再补充。

5. 什么时候更新 CLAUDE.md

CLAUDE.md 不是写一次就不动了。导师每次开会给新反馈，我就打开加几行。比如导师说「以后标题里不要出现' 策略' 这个词，显得轻飘」，我当天就加到禁忌词表里。还有一次，我发现 Claude Code 反复把输出格式搞成 Markdown，但我要的是 LaTeX，纠正了两三次之后我直接在 CLAUDE.md 里加了一条「默认输出格式为 LaTeX」，从此没再犯过。项目结构有变化的时候也要更新，比如新增了一个数据文件夹、改了章节编号，都值得同步到 CLAUDE.md 里。

更新方法很简单，直接用文本编辑器打开改就行。下次启动 Claude Code 它就会读到新版本。

2.3 Memory 系统：跨会话记忆

CLAUDE.md 解决了「每次重新启动都要交代背景」的问题。但用了一段时间之后，我发现还有一类信息是 CLAUDE.md 管不了的。

举个例子。有一次导师开会的时候随口说了一句：「结论部分用被动语态写，不要用' 我们发现' 这种主动句。」这条要求很具体，只针对结论那一节，写进 CLAUDE.md 显得太琐碎（CLAUDE.md 里应该放全局性的信息）。但下次我让 Claude Code 帮我写结论的时候，如果它能记住这条偏好，输出质量会好很多。

还有一种情况：我在某次对话中折腾了半小时，终于发现用 LaTeX 输出比 Word 稳定，不容易出现 XML 损坏的问题。这个经验下次还能用到，但也不值得专门写进 CLAUDE.md。

这类「零散但有用的信息」，就是 Memory 系统管的。

1. Memory 和 CLAUDE.md 的区别

CLAUDE.md 是你手动写的、放在项目文件夹里的、每次启动都会被读取的说明书。它适合存放相对稳定的、全局性的信息：项目背景、术语表、代码风格、导师的基本要求。你写一次，管很久。

© 2026 晨瀚宇 未经授权禁止转载

Memory 是 Claude Code 自动管理的、存在它自己配置目录里的笔记系统。它适合存放在使用过程中积累的零散信息：某次对话中你提到的偏好、某个文件的特殊处理方式、上次做到哪里了。这些信息可能过两周就过时了（比如「目前在改第四章」，等第四章改完这条就没用了），但在有效期内它能帮上忙。

区分的标准很简单：如果一条信息是「无论做什么任务都应该知道的」，写 CLAUDE.md；如果是「某些特定场景下有用的」，存 Memory。

2. Memory 存了什么

Memory 系统会在你的项目配置目录下创建一个 `memory/` 文件夹，里面有一个 `MEMORY.md` 索引文件，指向各个具体的记忆文件。

我的项目里，Memory 目录长这样：

```
memory/
    MEMORY.md                    # 索引
    feedback_writing_clarity.md # 导师对写作的要求
    feedback_book_writing_style.md # 书籍文风的反馈
    project_book_claude_code.md # 书籍项目的信息
```

`MEMORY.md` 是一个很简短的索引，每条记忆一行：

```
- [写作表述要求](feedback_writing_clarity.md)
  -- 导师要求论文逻辑清晰、上下连贯
- [书籍写作项目](project_book_claude_code.md)
  -- 正在写Claude Code科研辅助书
- [书籍文风反馈](feedback_book_writing_style.md)
  -- 禁止破折号、禁止套路开头、标题要正式
```

每个具体的记忆文件里存的是详细信息。比如 `feedback_writing_clarity.md` 里存的是导师对论文写作的具体要求：一段一个主题、句子尽量短、不堆砌从句、每段第一句能概括该段内容。

3. 怎么用 Memory

你可以在对话中主动让 Claude Code 记住某件事：

```
记住：我导师要求论文里不能出现「显著性水平」这个词，
用「置信水平」或直接写 p 值。
```

Claude Code 会把这条信息写入 Memory 文件。下次你重新启动对话时，它会读取 Memory 内容，知道这个约束。

你也可以让它回忆之前存过的信息：

```
看一下你的记忆里有没有关于导师禁用词的记录。
```

4. Memory 适合存什么，不适合存什么

什么信息适合存 Memory 呢？比如导师在某次开会时随口说了一句「结论部分用被动语态」，这种话不值得写进 CLAUDE.md，但下次你让它帮你写结论的时候，它要是能记住，效果会好很多。再比如你在某次对话中发现 LaTeX 输出比 Word 稳定，或者你的论文目前改到第 7 章、前面 6 章已经定稿了，这些阶段性的信息也适合存在 Memory 里。

© 2026 晨瀚宇 未经授权禁止转载

不适合存 Memory 的信息，应该直接写在 CLAUDE.md 里。比如项目的整体背景和架构、核心术语锁定表、常用命令和配置。这些是「无论做什么任务都需要知道的」基础信息，放在 CLAUDE.md 里每次启动都能读到，比存 Memory 更可靠。

判断标准很简单：如果一条信息是「无论做什么任务都应该知道的」，写 CLAUDE.md；如果是「某些特定场景下有用的」，存 Memory。

2.4 实操：创建你的第一个 CLAUDE.md

下面是具体操作步骤。

第一步：创建文件

```
cd ~/Desktop/thesis # 进到你的论文文件夹
touch CLAUDE.md     # 创建空文件
```

第二步：写入基本内容

用任何文本编辑器打开 CLAUDE.md（VS Code、记事本、vim 都行），写入以下内容。把方括号里的内容替换成你自己的信息：

```
# 论文项目说明

## 研究方向
[一两句话描述你的研究课题，比如：
本论文研究XX因素对YY的影响及空间溢出效应。]

## 术语锁定表
以下术语在任何修改任务中必须保持原样：
- 「被解释变量」（不要改成「因变量」）
- 「双向固定效应」（不要改成「固定效果」）
- 「空间杜宾模型」（不要改成「空间回归模型」）

## 输出格式
- 使用学术中文
- 修改后给出改前改后的对比
```

先写这三样就够了。不用一步到位，后面遇到问题再补充。

第三步：验证它是否生效

启动 Claude Code，问它一句：

```
你知道这个项目是做什么的吗？列一下你从CLAUDE.md里读到的信息。
```

如果它能正确复述你写的研究方向和术语表，说明 CLAUDE.md 已经生效了。

第四步：在使用中逐步补充

用了几天之后，你会慢慢发现需要补充的东西。比如导师又给了新的反馈，你就加到禁忌词表里；发现 Claude Code 总是把输出格式搞错，就加一条格式要求；项目结构变了，更新一下文件说明。每次打开 CLAUDE.md 加

© 2026 晨瀚宇 未经授权禁止转载

几行就行，不用大改。

笔记 如果你的论文文件夹用了 git 管理（或者你打算用 LaTeX + git），CLAUDE.md 也可以提交到 git 里。这样换电脑或者重装系统的时候，CLAUDE.md 跟着项目走，不会丢。

到这里，你已经有了两个工具来让 Claude Code 记住你的背景：CLAUDE.md 存放稳定的项目信息，Memory 存放零散的使用偏好。有了这两样东西，以后每次打开 Claude Code，它就已经知道你在做什么研究、哪些词不能改、输出格式是什么。不用再粘贴那段三四百字的背景介绍了。

下一章讲怎么和它说话。同样的任务，换一种说法，结果可能完全不同。

© 2026 晨瀚宇 未经授权禁止转载

第三章 提示词的实战经验：怎么和它说话

内容提要

- 精确指令与模糊指令的差距：约束越多，出错越少
- 先确认方案再动手：一句话节省大量返工时间
- 术语保护的两个层级：全局锁定与任务级锁定配合使用
- 打断与纠正的有效方式：否定、期望、理由三句话结构
- 一次会话只做一件事，以及把导师反馈翻译成可执行指令

第一章讲了怎么装，第二章讲了怎么让它记住你。这一章讲怎么和它说话。

同样是让 Claude Code 帮你改论文，换一种说法，结果可能完全不同。我用了大半年，前后一百多次对话，慢慢摸出了一些规律。这些规律不复杂，但如果没人告诉你，你可能要自己踩很多遍坑才能总结出来。

3.1 模糊指令与精确指令的差距

定义 3.1 (提示词（Prompt）)

提示词是你输入给 AI 的自然语言指令。对 Claude Code 而言，提示词的质量直接决定输出的质量：约束越明确（范围、禁忌、风格、交付格式），模型自由发挥的空间越小，出错的概率越低。

刚开始用 Claude Code 的时候，我给它的指令基本上是这种风格：「帮我改一下第三章。」

它也确实会改。但改完你一看，问题来了：它把你没让它改的地方也改了，术语换了你不想换的，格式变成了它自己觉得「更好」的样子。你花十分钟看它改了什么，再花二十分钟把不想要的改动还原，最后发现还不如自己手动改那一处。

后来我慢慢发现，问题出在我的指令太模糊了。「帮我改一下第三章」这句话，对人来说能理解（大概是改改措辞、顺顺逻辑），但对 Claude Code 来说，它会把所有它认为「可以改进」的地方都改一遍。它不知道你只想改某一段的表述，不知道哪些术语不能动，不知道你要的是学术中文而不是通俗中文。

下面是几组我真实用过的指令对比，左边是我刚开始的写法，右边是后来改进的写法：

表 3.1: 模糊指令与精确指令的对比

模糊指令	精确指令
帮我改一下第三章	修改第三章第 2 节，不改变论点，用学术中文，以下术语不得修改：「被解释变量」「双向固定效应」，改完给我看修改前后的对比
帮我润色一下摘要	润色摘要，去掉具体数值改为定性概括，保留所有引用编号，术语原样保留
帮我检查一下引用	读一下 references.bib，检查每条引用是否有作者、年份、期刊名，缺失的标出来
帮我画个流程图	用 Draw.io（不要用 Mermaid）画一个研究框架图，包含数据收集、模型构建、实证检验三个阶段，输出为.drawio 文件

区别在哪里？精确指令告诉了 Claude Code 四件事：改什么范围、用什么标准、什么不能动、改完要给我看

© 2026 晨瀚宇 未经授权禁止转载

什么。你给的约束越多，它自由发挥的空间越小，出错的概率就越低。

当然，每次都写一大段也没必要。简单任务一句话够用，比如「帮我数一下 references.bib 里有多少条引用」，任务本身已经很明确。但凡涉及修改文本，都建议把约束说清楚。

我后来给自己总结了一个写指令的检查清单：在发出指令之前，先想四个问题。（1）改哪个文件的哪个部分？（2）什么东西不能改？（3）用什么语言风格？（4）改完要给我看什么？如果这四个问题都在指令里回答了，基本不会出太大的问题。

3.2 先确认方案，再动手

这是我用了一个多月才养成的习惯，但它是所有技巧里投入产出比最高的一条。

起因是这样的：有一次我让 Claude Code 帮我画一个研究框架图。我说「帮我画一个流程图」，它二话不说就开始画了。画完我一看，它用的是 Mermaid 语法。Mermaid 是一种用文本描述图的格式，在技术文档里很常见，但我要的是 Draw.io 格式（因为导师要求提交的图都用.drawio 格式，方便后续修改）。

这次白费了大概十分钟。它画了，我看了，然后告诉它格式不对，它又重新画了一遍。如果一开始就说清楚用什么工具，根本不会浪费这个时间。

但更根本的问题是：为什么不让它先说说它打算怎么做？

后来我养成了一个习惯：凡是稍微复杂一点的任务，在正式指令之前先加一句：

```
在动手之前，先告诉我你的方案：
改哪个文件、用什么方法、输出格式是什么。
等我确认后再执行。
```

这句话的效果立竿见影。Claude Code 收到之后，会先列出它的计划：「我打算打开 ch3_method.docx，读取第 2 节的内容，用学术中文重写，保留所有术语不变，输出修改前后的对比。你确认吗？」你扫一眼，没问题就说「可以，开始」；如果它理解错了，你在这个阶段纠正，成本很低（只需要改一句话），而不是等它改完了再推翻重来。

我后来还遇到过一个更严重的案例。我让它做一次全文术语替换（第一章讲过的那个场景），它不仅替换了正文，还把表格标题、图注、甚至 BibTeX 文件里的注释都改了。有些地方是引用别人论文的原文，不应该改的。如果我加了那句「先告诉我你打算改哪些文件」，它会列出所有要动的文件，我看到 BibTeX 那一项就会说「这个不要改」。

注 成本只是多一轮对话，但能避免大量返工。这条规矩我现在几乎每次都用。

3.3 术语保护：不让它擅自替换你的专业词

定义 3.2 (术语锁定)

术语锁定是指在指令中明确禁止 AI 替换特定专业词汇的机制。分为两个层级：写入 CLAUDE.md 的全局锁定（对所有任务生效），以及在单条指令里额外列出的任务级锁定（仅对本次任务生效）。两层叠加使用，可最大限度防止术语在修改过程中发生漂变。

这个问题在第二章讲 CLAUDE.md 术语锁定表的时候已经提过了。这里从另一个角度来说：就算你写了术语锁定表，在具体任务的指令里也最好再提一次。

© 2026 晨瀚宇 未经授权禁止转载

原因很简单。CLAUDE.md 里的术语锁定表是全局性的，它列出了「所有任务都不能改的词」。但有些任务有额外的术语敏感性。比如你让它帮你改一段方法论的描述，这段话里可能有一些不在全局锁定表里、但在这个上下文中不应该被改的词。

举个例子。我论文里有一段描述研究方法的话：「本文采用空间杜宾模型（SDM），以经济距离权重矩阵构建空间关联结构。」我让 Claude Code 帮我润色这段话。它把「经济距离权重矩阵」改成了「基于经济距离的空间权重矩阵」。意思差不多，但我论文里前后几十处都用的是「经济距离权重矩阵」这个简称，它一改，这一段就和其他地方不一致了。

这个词不在我的全局术语锁定表里（因为它不是核心术语，只是一个特定的技术表述）。但在这个具体任务里，它不应该被改。

解决办法是在任务指令里再加一句：

```
以下词汇在本次修改中保持原样：
「经济距离权重矩阵」「空间杜宾模型」「双向固定效应」
```

全局锁定表管的是「永远不能改的」，任务级别的锁定管的是「这次不能改的」。两层一起用，术语漂变的概率就很低了。

还有一个技巧是在指令末尾加一句「如果你觉得某个术语需要修改，先问我，不要自己改」。这句话的效果因场景而异，但至少给了它一个「不确定就问」的暗示，比它自作主张好得多。

3.4 打断和纠正的方法

用 Claude Code 做稍微复杂的任务时，它经常会走偏。比如你让它改一段话的逻辑，它改着改着开始重写整段；你让它检查引用格式，它检查着检查着开始帮你删「多余」的引用。

发现走偏了要马上打断。Claude Code 在终端里运行，按一次 `Esc` 键就能中断当前操作，它会停下来等你的下一条指令。如果你想撤销它刚才做的修改，快速按两次 `Esc`，它会回退到上一个检查点，把代码和对话状态都恢复。

但打断之后怎么说，比打断本身更重要。我一开始犯的错误是：打断之后只说「不对」或者「不是这样」，然后它就按照自己的理解重新来一遍，结果还是偏。

后来我学到一个有效的打断方式，三句话：不要做什么、我想要什么、原因是什么。

```
不要重写整段话，我只需要调整第2句和第3句的逻辑顺序。
原因是：第2句说的是结论，第3句说的是依据，
应该先说依据再说结论。其他句子不要动。
```

三句话的结构是：否定（不要做 X）、肯定（我想要 Y）、理由（因为 Z）。给了理由之后，它不仅知道这次要怎么做，还能理解背后的原则，下次遇到类似情况出错的概率也会降低。

如果只说「不对」就让它重来，它很可能会犯同一类错误，因为它不知道错在哪里。打断之后要把你的期望说清楚，它才能真正调整方向。

3.5 一次会话只做一件事

这条在第一章的安全注意事项里提过一句，这里展开讲，因为它太重要了。

© 2026 晨瀚宇 未经授权禁止转载

刚开始用 Claude Code 的时候，我经常在一次对话里塞三四个任务：「帮我改第三章的逻辑，顺便把引用格式统一一下，再检查一下术语有没有不一致的地方。」听起来效率很高，一次搞定所有事。

实际情况是：它做了前两个，到第三个的时候碰到了用量限制（rate limit），对话被打断了。前面做的引用格式统一还没写入文件，全丢了。我只能重新开一次对话，从头再来。

这种事情发生过三四次之后，我给自己定了个规矩：一次对话只做一件事。判断标准是看你的需求里有几个「然后」。如果超过两个「然后」，就拆成多次对话。

拆开做还有一个好处：每次做完一件事就能检查结果、确认没问题、把文件存好。如果中途出了问题，只需要回滚这一步，不会影响前面已经做好的部分。

碰到 rate limit 被打断了怎么办？Claude Code 有一个接续功能：在终端里输入 `claude --continue`（或简写 `claude -c`），就能恢复上次被打断的会话，它还记得之前的上下文，从断点继续做。大多数情况下这就够了。

但如果任务跨了很多轮、持续好几天，上下文可能被自动压缩，前面做过什么的细节会丢失。遇到这种长任务，我会在指令里加一句：

```
完成每个主要步骤后，把当前进度保存到 checkpoint.md，
包括：已完成的、待完成的、遇到的问题。
```

checkpoint.md 相当于给 Claude Code 写的备忘录。即使上下文被压缩了，让它读一下这个文件，就知道做到哪了。这个方法在做引用批量核查的时候救了我一次，第十章会详细讲那个故事。

3.6 把导师反馈翻译成可执行的指令

导师给反馈的时候，说的话通常是定性的、笼统的。「逻辑不清晰」「太口语化」「这段拔高一下」。这些话人能理解，但直接丢给 Claude Code，它不知道具体该怎么做。

我的做法是：先自己理解导师的意思，再翻译成 Claude Code 能执行的具体指令。翻译的关键是把定性评价变成定量操作。

下面是几组真实的翻译对照：

表 3.2: 导师反馈的翻译对照

导师原话	翻译成 Claude Code 指令
逻辑不清晰	第 X 段和第 Y 段之间缺少过渡，补一句承上启下的话；第 Z 段的结论放在了依据前面，调换顺序
太口语化	把以下词替换为书面表达：「所以」→「因此」、「其实」→ 删除、「很多」→「大量」
摘要太琐碎，要拔高	去掉摘要中的具体数值（样本数、百分比），改为定性概括；语言从罗列做法变成概括贡献
这部分展开不够	第三章第 2 节当前只有两段，需要补充：方法选择的理由（为什么用这个模型而不是别的）、数据来源的说明、变量构建的具体过程

翻译的过程其实就是你自己在想「导师到底觉得哪里有问题」。这个思考过程不能交给 AI，因为它不了解你导师的标准和偏好。但一旦你想清楚了，执行的部分可以交给 Claude Code，它做得比你快。

第一章提到的那次 40 多条修改意见，我就是用这个方法处理的：花一个下午把意见翻译成具体指令，然后用了四天让 Claude Code 逐条执行，省了至少一周的时间。

© 2026 晨瀚宇 未经授权禁止转载

3.7 实操：用提示词模板改写一段论文

下面用一个完整的例子走一遍上面讲的所有技巧。假设你有一段论文，导师说「太口语化了，改正式一些」。

第一步：准备指令

```
读一下 ch4_result.docx 的第2节。
导师说这段「太口语化」，需要改成正式的学术表述。
在动手之前，先告诉我你的方案。

具体要求：
1. 仅修改口语化的词汇和句式，不改变论点和逻辑结构
2. 以下术语不得修改：「被解释变量」「双向固定效应」「空间杜宾模型」
3. 改完给我看修改前后的对比，标出每处改动
```

这条指令用到了前面讲的几个技巧：先确认方案（「先告诉我你的方案」）、限定范围（「仅修改口语化的词汇」）、术语保护（列了三个不能改的词）、要求对比（「给我看修改前后的对比」）。

第二步：审查方案

Claude Code 会回复它的计划，比如：「我打算读取 ch4_result.docx 的第 2 节，识别其中的口语化表述，替换为书面表达，保留所有术语不变。」

你确认没问题，回复「可以，开始」。

第三步：检查结果

它改完之后会给你一份对比。逐条看，重点检查两件事：术语有没有被改、有没有多改了不该改的地方。如果有问题，用第四节讲的打断方式纠正：「第 3 处改动不对，’ 显著正相关’ 不需要改成’ 呈现显著正向关联’，改回来。」

第四步：存档

确认没问题后，如果你用的是 Word，先备份再让它写入。如果用的是 LaTeX + git，直接让它改，改完用 `git diff` 看一遍差异，没问题就提交。

这个四步流程（准备指令 → 审查方案 → 检查结果 → 存档）适用于大多数修改类任务。用几次之后就会变成习惯，每次不需要刻意去想。

到这里，你已经知道怎么安装、怎么让它记住你、怎么和它说话了。前三章是基础，从第四章开始进入论文写作的具体流程。下一章讲文献调研。

© 2026 晨瀚宇 未经授权禁止转载

第二部分

核心技能——论文全流程

© 2026 晨瀚宇 未经授权禁止转载

第四章 文献调研与管理

内容提要

- 用 /pdf 批量读取文献并按研究方法或功能分组，替代手工整理
- 用 ppw-literature 和 /arxiv 检索文献并自动生成验证过的 BibTeX 条目
- 通过并行 Agent 对大批量引用进行核查，兼顾速度与进度可恢复性
- 梳理文献脉络：从「逐篇介绍」转向「按方向讲趋势、找缺口」
- 用 /citation-management 清理格式问题，但学术判断须由作者自行把关

写论文的时候，我最怕的不是实验跑出来结果不显著，是写第二章。

第二章是文献综述。我的导师对文献综述有一个要求：「让读者看完这一章，就知道这个领域经历了什么发展、现在有什么争议、你的研究填补了哪个空白。」三件事，每件单独做都不容易，合起来写进一章更难。

当时我手里有八十多篇参考文献，都读过，每篇都有批注。我以为这就够了。花了两周写出第一版，交上去，导师当天下午就发回来了：「没有体现研究脉络，罗列感太强。」

修改了一版再交，又退回来：「研究缺口不明显，读者不知道你为什么要做这个研究。」

第二次退改之后，我才真正意识到，读懂每篇文献和写好文献综述是两回事。读文献是在理解每篇文章自己说了什么，写文献综述是在把这些文章放进同一张地图里，画出它们之间的关系，然后指出地图上哪里有个坑，你要来填。读完八十篇文献，你对每棵树都很熟悉，但不一定能看清楚整片森林。

这一章讲的是我用 Claude Code 做文献工作的完整过程：从用工具检索和读文献、批量核查 156 篇引用，到把这些文献组织成有逻辑的综述。

4.1 用 AI 整理研究现状

我真正开始大量用 Claude Code 做文献工作，是从第三次修改文献综述开始的。前两次被退了，第三次我换了一个思路：先把所有文献按主题分组，再想每组之间的关系，最后从关系里找缺口。

这件事以前我是用 Excel 做的。一张表，每行一篇文献，列上写着作者、年份、研究方法、核心发现、与我论文的相关性。整理完之后，我对着表格想每组之间的联系。这张表我花了三四天才做完，整个过程非常枯燥：打开 PDF、找摘要、找方法论那段、复制信息、粘到表格里、关闭 PDF，开下一篇。八十多篇文献，来回开关文件开关文件。

Claude Code 改变了这件事的做法，因为有一个叫 `/pdf` 的功能。你不需要手动打开 PDF 提取内容，直接在对话里输入 `/pdf 文件名.pdf`，它会提取全文文字并且读取进来。

第三次修改文献综述，我把所有文献的 PDF 放在一个文件夹里，让 Claude Code 逐个读取，让它按研究方法给我分组。指令是这样写的：

```
请读取 literature/ 文件夹里的所有PDF。
对于每篇文献，提取：作者、年份、研究方法、核心发现。
然后按研究方法把它们分成若干组，每组给一个简短的名称。
把结果写入 literature_summary.md。
```


© 2026 晨瀚宇 未经授权禁止转载

它读完所有文献，给了我一个按方法分类的列表。我论文里涉及的方法大致分成四类，它识别出来了，每类下面列了属于这类的所有文献。这件事用 Excel 做，我花了三四天；用 Claude Code 做，不到两小时。

这两小时里，我大部分时间花在看输出结果、核对有没有分类错误的地方。它确实错了一处：把一篇用工具变量法的文献错分到了双向固定效应那组，因为那篇文章同时用了两种方法，它判断成了后者。我指出来，它重新归类了。真正让它跑的时间，不超过二十分钟。

按功能分类，而不是按内容分类

读完文献之后，整理成综述之前，有一个过渡工作：把文献按照它们在综述里的功能分类。分内容是说「这篇文章研究了什么」，分功能是说「这篇文章在我的论文里起什么作用」。后者对写综述更有用。

功能分类的维度有几个：和我研究问题最接近的（需要重点综述）、提供方法论支持的（说明我的方法有先例）、结论与我研究假设方向一致的（支持论证）、结论与我研究假设相反的（需要解释为什么结论不同）。

我的做法是把这些维度告诉 Claude Code，让它按每篇文献在我论文里的功能来分类。举个具体例子：我的论文涉及空间杜宾模型，有几篇文献在做技术层面的方法论讨论（比如如何选择空间权重矩阵），还有几篇把这个模型用到了其他研究对象上。前一类引用是为了说明我的方法选择有依据，后一类是为了说明这个模型在类似情境下被验证有效。两类文献在综述里起的作用不同，但如果只按「涉及空间杜宾模型」来分类，它们就混在一起了，写起来容易乱。

按功能分类之后，写综述的时候就清楚多了：这段说方法论，引这几篇；那段说应用背景，引那几篇。逻辑不会乱。

用 ppw-literature 自动搜索文献

整理已有文献之外，还有一件事：找新文献。这件事以前我是在 Google Scholar 搜一大堆结果，然后一篇篇点进去看标题、读摘要，判断相不相关。找到一篇相关的，再手动抄下作者、年份、期刊名，或者用 Zotero 抓取。一批文献搜下来，整理 BibTeX 条目本身就要不少时间。

后来我开始用一个叫 `ppw-literature` 的功能，它接入了 Semantic Scholar 的学术数据库，能直接在 Claude Code 里搜文献、显示引用数和摘要、让你选，选完自动生成验证过的 BibTeX 条目。我用了二十六次左右。

触发方式很简单，直接告诉 Claude Code 你要找什么：

```
帮我找关于空间计量模型在政策评估中应用的文献
```

它会搜索学术数据库，返回一个编号列表，每条包括标题、作者、年份、引用次数、摘要片段：

```
[1] Spatial Econometric Models for Policy Evaluation
   Authors: Li, W.; Zhang, H. · Year: 2021 · Citations: 234
   > "This paper reviews spatial econometric approaches in
     regional policy evaluation, focusing on spillover effects..."

[2] Identifying Causal Effects with Spatial Panel Data
   Authors: Chen, Y.; Wang, F. · Year: 2022 · Citations: 89
   > "We propose a spatial IV estimator robust to cross-sectional
     dependence in panel settings..."

请选择一篇（输入编号），或输入 none 取消。
```

你选一个编号，它生成 BibTeX 之前会强制提示一句：「请确认你选的这篇文章确实支持你要引用的论点，不

© 2026 晨瀚宇 未经授权禁止转载

要只看标题判断。」这句话不是废话，arXiv 和 Semantic Scholar 上的论文，有时候标题和内容之间会让你误判相关性，尤其是跨学科的方法论文章。提示你确认，是在提醒你去看一眼摘要再决定。

确认之后，它生成的 BibTeX 条目只用数据库返回的数据，不会自己填充或编造字段：

```
@article{li2021spatial,
  title   = {Spatial Econometric Models for Policy Evaluation},
  author  = {Li, Wei and Zhang, Hua},
  journal = {Regional Science and Urban Economics},
  year    = {2021},
  volume  = {87},
  pages   = {103634},
  doi     = {10.1016/j.regsciurbeco.2021.103634}
}
```

如果某个字段（比如页码或卷号）数据库没有返回，它直接省略，不会帮你猜一个填进去。这个设计解决了一个很实际的问题：用其他工具生成 BibTeX，偶尔会出现字段看起来完整但实际上信息是 AI 编的情况，等你去对原始文献的时候才发现页码对不上。

定义 4.1 (BibTeX)

BibTeX 是 LaTeX 排版系统使用的参考文献管理格式，每条引用用一个以 @ 开头的条目表示，包含类型（如 @article、@book）、引用键（citation key）及若干字段（作者、年份、标题等）。Word 用户也可通过 Zotero 等工具导出 BibTeX 格式备用。BibTeX 条目的字段信息必须与原始文献完全一致，不可由 AI 自动填充推测。

♣

/arxiv 的用法

定义 4.2 (arXiv 预印本)

arXiv 是由康奈尔大学维护的开放获取学术预印本服务器，涵盖物理、数学、计算机科学、经济学等领域。预印本指尚未经过正式同行评审的论文版本，可在投稿期间或发表后免费公开。引用预印本时，应在 BibTeX 的 note 字段注明「arXiv preprint」，并在论文正式发表后替换为发表版引用信息。

♣

涉及比较新的方法的文献，有些还没在期刊正式发表，只有 arXiv 预印本。

Claude Code 有一个 `/arxiv` 功能，可以直接搜索 arXiv 上的论文，返回标题、摘要、作者信息，还可以把 PDF 读进来。我用这个功能搜过九十次左右，大部分是在确认某篇文献是否有了更新版本，或者找某个方法的原始来源。

用法很直接：

```
/arxiv search "spatial Durbin model panel data policy"
```

它会返回相关预印本列表，每条包括标题、作者、摘要片段、arXiv 编号。你选中感兴趣的，让它读全文：

```
/arxiv read 2203.12345
帮我判断这篇文章和我的研究问题有多大关系。
我的研究问题是：XX因素对YY的影响及空间溢出效应。
```


© 2026 晨瀚宇 未经授权禁止转载

这个功能帮我找到过一篇文献的最新修订版。那篇文章是我论文第二章引用的一个关键理论基础，我引的是 2021 年的版本。它在 arXiv 上有一个 2024 年的修订版，补充了一个我论文恰好涉及的场景的分析。导师后来在审稿意见里提到「这个问题近期有新进展」，我已经更新了引用版本，才没有被扣分。

有一件事要提醒：arXiv 上的论文是预印本，没有经过同行评审。在论文里引用预印本，最好在 BibTeX 的 note 字段标注「arXiv preprint」，有些导师或期刊不接受仅引用预印本。如果一篇预印本后来正式发表了，要换成发表版的引用信息。

ppw-literature（Semantic Scholar）和/arxiv 的分工很清楚：期刊论文、会议论文用前者，方法比较新还没正式发表的用后者。两个配合着用，基本覆盖了所有场景。

4.2 批量文献核查

我的论文最终有 156 条参考文献。提交前一周，导师提醒我：「引用列表最后仔细核对一遍，格式要符合学校要求，信息不能有错。」

156 条，意味着什么？意味着你要把这 156 篇文献一条条打开，对照 BibTeX 条目里的作者、年份、期刊名、卷号、页码，逐一确认有没有错误。每条文献核查 5 分钟，156 条就是 13 个小时。提交前一周，除了引用核查还有很多事情要做，13 个小时是我不可能花在这件事上的。

我用并行 Agent 解决了这个问题。

什么是并行 Agent

一般情况下，你和 Claude Code 的对话是单线程的：你说一句，它做一件事，做完告诉你，你再说下一句。这适合大多数场景。但对于「156 条引用、每条需要做同样步骤的核查」这种批量任务，单线程太慢了，而且容易碰到使用频率限制，做到一半被打断，前面的工作就白费了。

并行 Agent 的做法是让 Claude Code 自动派发多个子 Agent，每个子 Agent 负责一批任务，各自做完之后汇总。好比你交给一个组长 156 道计算题，组长自己分配给三个人做，做完把答案合并交给你。

我的具体做法很简单——在一个对话里把任务说清楚：

```
读取 references.bib，对每条引用核查：
作者、年份、期刊名、页码是否正确，DOI是否有效。
有问题的列出来，写一份核查报告。
```

Claude Code 收到之后，自动判断 156 条数据量大，派了几个子 Agent 分头核查，最后把各自的结果合并成一份报告给我。整个过程我不需要手动分批、不需要指定输出格式，也不需要写限频处理的指令——这些它自己搞定。

核查结果

三个 Agent 并行跑，整体用了约 40 分钟，找出了 11 处问题：3 处年份错误（引用的是文章的预印本年份，不是正式发表年份）、4 处页码不完整（只有开始页没有结束页）、2 处期刊名缩写不统一（有的条目用缩写，有的用全称）、2 处 DOI 无法解析（一篇文章后来撤回了，另一篇 DOI 格式有误）。

11 处错误，分散在 156 条引用里，如果手动核查，很难确保每条都仔细查了。自动化核查直接把所有问题列出来，我逐条处理，半小时改完了。

对于核查结果里的问题，修改是要你自己来做的。Claude Code 告诉你哪条引用有问题，但它没法帮你判断「这篇被撤回的文章应该用什么文献替换」，那需要你自己做学术判断。我的处理方式是：先把 145 条无问题的引

© 2026 晨瀚宇 未经授权禁止转载

用标记为已验证，然后集中处理那 11 条。年份错误和页码不完整的，找到正确信息直接改 BibTeX。那篇被撤回的文章，我引用它只是为了说明一个「有先例」，找了另一篇类似研究替换掉了。那篇 DOI 格式有误的，重新查了一遍，补上了正确的 DOI。

整个过程，从开始跑三个并行 Agent 到改完所有 11 处问题，大概花了两小时。手动核查，保守估计五六小时，而且很难保证没有遗漏。

表 4.1: 手动核查与并行 Agent 核查的对比（156 篇引用）

手动核查（逐条打开文献对照）	并行 Agent 核查
每条 5-10 分钟，156 条约 13-26 小时	三个 Agent 并行，约 40 分钟完成核查
疲劳时容易漏掉问题	每条按同一标准检查，不会遗漏
发现问题记在临时笔记里，可能丢失	自动生成结构化报告，问题一目了然
无法逐条联网验证	自动验证每条引用信息
结果散乱，难以统计整体问题数量	分类统计，可直接按报告逐条修复

使用 /citation-management 做格式清理

并行 Agent 核查主要处理的是「信息是否正确」。还有另一类问题：格式是否规范。中国高校的学位论文通常要求参考文献遵循 GB/T 7714 格式，这个格式对中英文文献有不同要求，中文文献还要标注语言字段。

我的 BibTeX 文件，来源很杂：有从知网导出的、有从 Google Scholar 复制的、有自己手动录入的。格式一点都不统一，有的用作者名缩写，有的用全名；有的期刊名用缩写，有的用全称；有的条目缺少 year 字段，有的有重复条目（同一篇文章两次录入，citation key 不一样）。

`/citation-management` 是专门处理这类格式问题的功能，我前后用了十六次，覆盖了从格式清理到重复检查的各类场景。我用的指令是这样的：

```
/citation-management 读一下 references.bib，做以下检查：
1. 有没有缺失必填字段的条目（作者、年份、标题）
2. 有没有重复的引用（同一篇文章两次录入）
3. 期刊名是否全部用了全称，不用缩写
4. 中文期刊论文是否有 language = {Chinese} 字段
把发现的问题写入 citation_issues.md，不要自动修改。
```

「不要自动修改」这一条很重要。引用信息里有些东西只有你知道是否正确，比如一篇会议论文的发表年份（预印本年份还是正式出版年份）。让它先列出问题，你确认之后再执行，避免帮你改了不该改的地方。

这次格式检查发现了：4 条缺少 year 字段的引用（手动录入时漏填了）、6 条期刊名用了缩写、3 条重复引用（同一篇文章在不同时期分别录入，citation key 不一样，所以一直没发现）。确认之后，让它统一修改，BibTeX 文件整洁多了。

4.3 文献综述的写法辅助

引用核查解决的是「引用信息是否准确」这个问题。更难的问题是「文献综述怎么写」。

第二章被退了两次，批注前面提过：第一次是「研究脉络不清晰」，第二次是「研究缺口不明显」。这两条都跟引用格式没什么关系，是综述本身的逻辑出了问题。

「研究脉络」是什么意思

「研究脉络」，大意是：这个领域从哪里来，往哪里走，中间经历了什么转折。

© 2026 晨瀚宇 未经授权禁止转载

没有研究脉络的综述，读起来是这样的：「学者 A（2010）研究了 XX，发现 YY。学者 B（2015）研究了 XX，发现 ZZ。学者 C（2020）研究了 XX，发现……」一篇接一篇地介绍，每篇都有，但彼此之间没有联系。读者看完只知道有很多人研究了这个问题，不知道这些研究之间是什么关系，这个领域的思路是怎么演化的。

有研究脉络的综述，结构不一样。它会说：早期研究（2000 年代）主要聚焦在 A 方面，但这些研究都忽略了 B 因素。2010 年代之后，有学者开始把 B 因素纳入进来，代表性工作是……但这类研究又面临 C 挑战。近五年来，有学者尝试用新方法解决 C 挑战……目前这个问题仍未完全解决，而这正是本文的切入点。

这样的综述，把文献放进了一个有时间轴、有逻辑递进的故事里。读者看完不只知道「有哪些研究」，还知道「这个领域的思路是怎么发展的」，以及「为什么现在还有值得研究的问题」。

用 Claude Code 梳理文献脉络

搜完文献、读完文献，接下来的问题是：怎么从八十篇文献里看出「脉络」来。这件事没有一个专用的 slash 命令，做法是直接把整理好的文献列表喂给 Claude Code，告诉它你的研究问题，让它帮你找关系和趋势。

第三版文献综述就是这么做的。我给它两样材料：一是前面整理出来的按主题分组的文献列表（lit_summary.md），二是我的研究问题（两句话）。然后让它做一件事：

```
读一下这份文献列表，结合我的研究问题，帮我梳理：
1. 这个研究领域有哪几个主要研究方向
2. 每个方向的发展脉络（从早期到近期经历了什么变化）
3. 现有研究在哪里有明显的缺口，跟我的研究问题有什么关系

不要逐篇介绍文献，帮我找关系和趋势。
```

它返回的结果，分了三个方向，每个方向用时间线描述了发展过程，最后总结了三处缺口。这个输出不能直接用进综述，但它给了我一个框架：这个领域的逻辑原来是这样的，从 A 到 B 到 C 是有连贯性的，我的研究在整个地图里的位置在这里。

有了这个框架，我回头重新组织综述，不再按文献一篇一篇地介绍，而是按「三个方向，每个方向讲脉络」的结构来写。这一版交上去，导师的批注只有一条：「可以，细节上再打磨一下。」

从两次退改到第三次通过，主要的变化就是有了这个结构框架。

「研究缺口」为什么容易写不清楚

第二次退改的批注是「研究缺口不明显」。这个问题，我后来想明白了，是写法的问题，而不是文献读得不够多。

研究缺口通常这样写：「现有研究在 XXX 方面存在不足，本文拟……」这句话逻辑上是对的，但读起来像是在为自己找理由，而不是在告诉读者这个缺口为什么存在、为什么值得填补。

我让 Claude Code 帮我做了一个检查：给它我的综述草稿，让它指出「研究缺口在哪里、是怎么表述的、读者看完能不能理解为什么需要这个研究」。

它的反馈是：我的综述里，缺口的表述分散在三个不同段落里，没有汇聚成一个明确的论点。读者要自己把三个地方的内容拼在一起才能理解缺口的完整含义，但大多数读者不会这么做。

解决方法是：在综述最后单独写一段，把三个方面的缺口合并成一个表述，明确告诉读者「综上，现有研究的局限在于……而本文正是针对这一问题」。这段话写好之后，研究缺口就清楚了。

注 这个检查，Claude Code 做起来比人快，因为它可以一次读完整章，扫描缺口论述在哪里，而不需要你自己盯着稿子找。问题在于，你自己写的东西读了太多遍，很难客观判断「逻辑是否清晰」，因为你知道自己想说什么，

© 2026 晨瀚宇 未经授权禁止转载

所以觉得文字已经表达清楚了。让 Claude Code 模拟一个没有背景知识的读者来读，就能发现你以为写清楚、但其实没写清楚的地方。

让 Claude Code 扮演的是「读者视角」，不是「导师视角」

这里要说一个容易踩的坑。如果你直接让 Claude Code「评估我的文献综述写得怎么样」，它通常会说「写得很好，逻辑清晰，但有几个地方可以进一步完善……」然后给你一些没什么具体意义的改进建议。

有效的做法是给它一个具体的任务，而不是让它给「综合评价」。比如：

```
读一下 ch2_literature.docx，告诉我：
研究缺口在哪里表述的（第几段，第几句），
把那几句话原文引用出来给我看。
```

这个指令让它做「找」而不是「评价」。找到之后，你自己看那几句话，自然就知道够不够清晰。如果它扫了整章说「没有找到明确的研究缺口表述」，那就是问题很严重了，说明你连写了多少都没写到位。

类似的检查还可以做：「找出这章里每段的第一句话，列出来。」（第一句话应该能概括这段的主旨，如果列出来读起来很乱，说明段落结构有问题。）「找出这章里引用了五次以上的文献，列出来。」（引用次数过多的文献，可能说明你对某一篇文章过度依赖，需要补充其他来源。）

这些都是「让 Claude Code 做具体任务」的方式，比「让它评价综合质量」有用得多。

4.4 实操：/pdf + /arxiv + /citation-management

这一节走一遍完整的文献工作流程，从检索到核查，按时间顺序来。假设你正在准备写文献综述，手里已经有一批 PDF 文献，需要把它们整理成综述材料，并完成最终的引用核查。

第一步：批量读取文献，按功能分类

启动 Claude Code，进到你的论文文件夹，让它批量读取文献：

```
读取 literature/ 文件夹里的所有PDF。
对每篇文献，提取：作者、年份、研究方法、核心结论（一句话）。
然后按照它们在我论文里的功能分成以下几类：
A：直接支持我研究假设的
B：提供方法论依据的
C：与我研究对象相关但方法不同的
D：结论与我不同、需要解释差异的
把分类结果写入 lit_by_function.md。
我的研究问题是：XX因素对YY的影响及空间溢出效应。
```

（把「XX 因素对 YY 的影响」替换成你自己的研究问题。）

第二步：检索补充文献

如果某个方向的文献不够，用/arxiv 补充：

```
/arxiv search "panel data spatial spillover effects"
```

找到感兴趣的预印本，读进来之前先判断相关性：

```
/arxiv read 2301.12345
```


© 2026 晨瀚宇 未经授权禁止转载

```
这篇文章和我的研究问题关系有多大?
我的研究问题是：XX因素对YY的影响及空间溢出效应。
用一两句话告诉我，不用详细总结全文。
```

确认相关后，让它把这篇文献加入你的文献列表，同时把 BibTeX 条目写进 references.bib。

第三步：梳理综述脉络

文献收集得差不多了，让它帮你看整体结构：

```
读一下 lit_by_function.md，结合我的研究问题，告诉我：
1. 现有研究分成哪几个主要方向
2. 每个方向经历了哪些转变（不要逐篇介绍，找趋势）
3. 哪里是明显的研究缺口，跟我的研究问题有什么关系
输出保存到 lit_roadmap.md。
```

这个输出是你写综述的「地图」，不是综述本身。对照这张地图写出来的综述，脉络会清晰得多。

第四步：检查综述草稿的逻辑

写完草稿之后，让它做逻辑检查：

```
读一下 ch2_literature.docx，告诉我：
1. 研究缺口在哪里表述的（第几段，原文引出来）
2. 把每段的第一句话列出来
3. 有没有某篇文献被介绍了但没有和研究缺口产生关联

不要修改，只给我反馈。
```

「不要修改」这一条很重要。这个阶段你要的是诊断，不是修改。修改是你自己的工作，因为「怎么表述才算清晰」涉及你对内容的理解，不是它能替你决定的。

第五步：引用核查

综述定稿之前，跑一次引用核查。如果引用数量超过五十条，考虑用第 4.2 节讲的并行 Agent 方式来做。

如果数量不多，用/citation-management 直接核查：

```
/citation-management 读取 references.bib，检查所有引用：
1. 有没有缺少必填字段的条目
2. 有没有重复的引用
3. 期刊名是否用全称
把发现的问题写入 citation_issues.md，不要自动修改。
```

看完问题列表，确认哪些需要改、怎么改，再让它执行。涉及学术判断的问题（比如应该引哪个版本、引哪篇替换一篇被撤回的文献），你自己来决定，不要让它自作主张。

这五步走下来，文献工作的主体部分就完成了。从检索到核查，每一步都有工具支持，但判断哪些文献值得引用、研究缺口的论述是否准确，这些工作还是要你自己来。Claude Code 在这里做的是加速执行和帮你看结构，学术判断这一块替代不了。

文献整理完了，下一步是开始写正文各章节。下一章讲章节写作：从研究问题到章节框架，以及怎么给 Claude Code 喂材料、一节一节地写出来。

© 2026 晨瀚宇 未经授权禁止转载

第五章 章节写作

内容提要

- 写章节之前先做框架审查：每节独立回答一个问题，节间逻辑递进
- 逐节写作时须喂材料：数据、结果、论点、上下文一并提供给 AI
- 用 checkpoint.md 记录跨会话进度，避免因中断丢失写作状态
- 通过明确语气约束消除「AI 味」，并用自查指令发现排比、空洞修饰等问题
- 把导师定性反馈翻译成可执行指令，分天按章节执行，兼顾质量与效率

文献综述过了之后，接下来就是正文各章节。我的论文正文有四章（第三到第七章），每章的写法不一样：第三章是研究设计，逻辑是「为什么用这个方法、怎么用」；第四第五章是实证分析，逻辑是「数据说了什么」；第七章是结论和建议，逻辑是「你的发现有什么意义」。

这四章加起来，我和 Claude Code 之间的对话超过了两百次。用得最密集的是论文冲刺期处理导师批量修改意见的那四天（第一章讲过那个故事）。

这一章讲的是：怎么用 Claude Code 从零开始写一节内容，怎么处理写到一半被打断的情况，怎么控制它的语气和术语，以及怎么把导师反馈变成可执行的修改计划。

5.1 从研究问题到章节框架

动手写之前，先要有一个框架。框架不是大纲（大纲是开题报告里就有的），框架是这一章具体写几节、每节回答什么问题、节与节之间是什么关系。

我有一次吃过没想清楚框架就动手的亏。第七章（结论章）一开始分了五节：研究结论、政策建议、理论贡献、研究不足、未来展望。看起来完整，但写到第三节的时候我就发现问题了：「理论贡献」和「研究结论」的内容有大量重叠，因为结论本身就隐含了贡献。两节说的是同一件事，只是角度稍有不同。写出来读着很啰嗦。

我把这五节的草稿喂给 Claude Code，让它帮我看结构：

```
读一下第七章目前的五节内容，告诉我：
1. 有没有哪两节之间内容重叠
2. 每节的核心论点是什么（用一句话概括）
3. 如果要合并成三节，你建议怎么合并

不要改内容，只给我分析。
```

它的分析很到位：第一节（研究结论）和第三节（理论贡献）有四处重叠，第四节（研究不足）和第五节（未来展望）也有重叠（「不足」往往自然引出「展望」，分开写容易变成说两遍）。它建议合并成三节：研究结论与贡献、政策建议、局限与展望。

我按这个结构重写了一遍。原来五节写了八千多字，合并后三节只有五千字，但信息量没有减少，反而因为去掉了重复，每一段都有新东西。导师看完说「这版好多了，紧凑」。

这个例子说明一件事：Claude Code 不光能帮你写，还能帮你看结构。你把已有的草稿或者大纲喂给它，让

© 2026 晨瀚宇 未经授权禁止转载

它做「诊断」，往往比让它直接写更有价值。因为你自己写的时候，容易陷在细节里看不到全局的结构问题，而它能一次读完整章，从全局视角给你反馈。

搭框架的具体做法

我后来养成了一个习惯：每写一章之前，先让 Claude Code 帮我做一个「框架审查」。把研究问题和章节大纲给它，让它检查三件事：每节是否都在回答一个独立的问题、节与节之间是否有逻辑递进、有没有遗漏的问题。

具体指令是这样的：

```
我要写论文的第X章，研究问题是：[一句话]。
目前计划分以下几节：
1. [节名和简要内容]
2. [节名和简要内容]
3. [节名和简要内容]

请检查：
- 每节的核心问题是否独立，有没有重叠
- 节与节之间的逻辑是否连贯
- 有没有遗漏的关键问题
不要替我写内容，只给我结构反馈。
```

这个步骤很快，五分钟就能做完。但它能帮你在动手之前发现结构问题，省得写完三四千字才发现要推翻重来。

5.2 逐节写作：怎么给 AI 喂材料

定义 5.1 (章节框架)

章节框架是写作动手前对一章内部结构的规划，包括：分几节、每节回答哪一个独立问题、节与节之间的逻辑关系（递进、并列或因果）。框架不等于开题大纲，而是在开始写作前对具体内容安排的确认，可帮助提前发现结构重叠或逻辑断层问题。

框架定好了，开始一节一节写。这里有一个很重要的技巧：不要让 Claude Code 凭空写，要给它材料。

注 「材料」是什么？是你的实验数据、你的分析结果、你的图表、你的思路笔记。Claude Code 能帮你把这些材料组织成通顺的学术段落，但材料本身必须是你的。如果你什么材料都不给，直接说「帮我写第三章第二节」，它写出来的东西一定是泛泛而谈的，因为它不知道你的数据是什么、结果是什么、你想强调什么。

我的做法是：写每一节之前，先准备一份「喂材料」的清单。

```
请帮我写第四章第2节「基准回归结果」，以下是材料：

1. 回归表格在 tables/table4_2.tex，请先读一下
2. 核心发现：被解释变量的系数为0.342，在1%水平显著
3. 控制变量里，人口密度和经济发展水平显著，
   产业结构不显著
4. 这一节的论点是：直接效应显著且方向符合预期
5. 需要和第三章第3节的研究假设呼应
```


© 2026 晨瀚宇 未经授权禁止转载

```
6. 术语不得修改：「被解释变量」「双向固定效应」

先告诉我你打算怎么组织这段话，确认后再写。
```

这条指令做了几件事：给了数据来源（表格文件）、核心发现（具体数字）、论点（你想表达什么）、上下文（和哪里呼应）、术语约束。Claude Code 拿到这些材料，写出来的内容就有具体的数据支撑，论点也明确。

中间结果要落盘

写正文章节往往是一个持续好几天的长任务。一次会话写完一节，下次再开会话写下一节。问题是：如果上一次写到一半被打断了（rate limit 的应对方法见第十章），第二天重开 Claude Code，它不记得昨天写到哪了。

这个教训让我开始用 checkpoint。

```
完成每个主要步骤后，把当前进度保存到 checkpoint.md，
包括：已完成的、待完成的、遇到的问题。
```

在任务开始时加上这句话，Claude Code 每做完一步就会更新 checkpoint.md。下次打开的时候，让它先读这个文件，就知道从哪里接着做了。

我的 checkpoint.md 长这样：

```
# 第四章写作进度

## 已完成
- 4.1 变量说明与数据来源（已写入ch4.docx）
- 4.2 基准回归结果（已写入ch4.docx）

## 进行中
- 4.3 稳健性检验（写了前两段，中断于第三段）

## 待完成
- 4.4 中介效应检验
- 4.5 空间溢出效应分析

## 遇到的问题
- 4.2 表格编号需要和第三章对齐，目前是表4-2，
  但第三章最后一个表是表3-6，所以应该从表4-1开始
```

这个文件很简单，但它解决了一个大问题：跨会话的连续性。Claude Code 的上下文每次都会清空，但文件不会。用文件来记住进度，比依赖它的记忆可靠得多。

一次写一节，写完验收

写正文也建议一次只写一节（原因参见第三章）。写完之后做一轮验收：

```
读一下刚才写的第四章第2节，检查：
1. 有没有出现不在术语锁定表里的替换
2. 具体数字（系数、显著性水平）是否和表格一致
3. 论点是否和第三章的研究假设呼应
```


© 2026 晨瀚宇 未经授权禁止转载

```
列出你发现的问题，不要自动修改。
```

验收通过了再进入下一节。这样每一节的质量都有保证，不会因为赶进度而留下隐患。

5.3 学术语气与术语控制

Claude Code 写出来的文字，如果你不加约束，有时候会带一些「AI 味」：用词过于整齐、句式过于对称、爱用排比、喜欢在每段结尾加一句总结。这些在日常文章里不是问题，但在学术论文里会很突兀，因为学术论文的语气应该是克制的、精确的，不需要修辞技巧来增强效果。

我遇到过几种典型的 AI 味写法。

第一种是「三段式排比」。比如你让它写一段变量说明，它可能写成：「该指标反映了 XX 的水平，体现了 YY 的程度，揭示了 ZZ 的趋势。」三个动词排比，读起来像在写公文。改成「该指标反映了 XX 的水平」就够了，后面两个是多余的。

第二种是「过渡句太满」。学术论文不需要每段之间都有一句「在此基础上，我们进一步分析了……」，直接进入下一段内容就行。导师第一次看到 Claude Code 帮我写的稿子就说：「你这些过渡句太多了，删掉，直接说事。」

第三种是喜欢用「值得注意的是」「具有重要意义」「进一步表明」这类万能连接词。这些词在学术写作里不是不能用，但用多了就显得空洞。

解决办法是在指令里加明确的语气约束：

```
语气要求：
- 用学术中文，不用排比句
- 不用「值得注意的是」「具有重要意义」等空洞修饰
- 段与段之间不需要过渡句，直接进入内容
- 每句话都应该传达新信息，不重复前文已说过的
```

把这些写进 CLAUDE.md 的输出格式要求里，每次启动它都能读到。

去 AI 化的检查方法

写完一段之后，可以让 Claude Code 自查 AI 味：

```
读一下刚才写的这段话，检查以下问题：
1. 有没有排比句（三个以上结构相同的分句）
2. 有没有空洞的修饰词（值得注意、具有重要意义等）
3. 有没有不必要的过渡句
4. 有没有重复前文已表达过的意思
列出你发现的，不要自动修改。
```

这个方法并不完美，因为 Claude Code 检查自己写的东西，有时候会「护短」，觉得自己写得没问题。更可靠的做法是你自己通读一遍，把「读起来不像人写的」句子标出来。但至少这个自查能抓住最明显的问题。

术语控制在第二章和第三章都讲过了，这里不再重复。只强调一点：在每个写作任务的指令里，除了全局的术语锁定表（CLAUDE.md），把这一节涉及的特定术语也列出来。比如写方法论那一节，把「经济距离权重矩阵」「空间杜宾模型」「工具变量法」这些你论文里有固定用法的词单独列一遍。双重保险，比事后纠正省心得多。

© 2026 晨瀚宇 未经授权禁止转载

5.4 导师反馈的处理流程

这是整本书里我最想分享的一个经验，因为它帮我把一个原来要两周的工作压缩到了四天。

第一章提到过，论文快提交时导师一口气给了 40 多条修改意见，具体的、笼统的、互相矛盾的都有。

以前碰到这种情况，我会直接打开论文一条一条改，改到哪算哪。但这次我换了一个做法：先不动手改，花半天把这 40 多条意见做一次结构化整理。

第一步：按章节分组

打开导师的意见记录，把每条意见归到它所属的章节。我的分组结果是这样的：第三章 8 条、第四章 6 条、第五章 12 条、第七章 9 条、全局性的 7 条。分完之后一看，第五章的问题最多，这意味着第五章可能需要大改，应该优先处理。

第二步：按类型分类

同一章的意见里，有些是「改一处」（替换某个词、修正某个数字），有些是「改一段」（重写某段话的逻辑），有些是「改整节」（结构需要调整）。把它们按工作量排序，先做小的，再做大的。小的修改做完之后，整章的面貌就清楚多了，再做大的修改时心里更有数。

第三步：翻译成 Claude Code 能执行的指令

这是关键步骤。导师说的话通常是定性的，Claude Code 需要具体的操作指令。第三章讲过翻译的方法，这里再给一个真实的例子。

导师说：「第五章的逻辑需要重新组织，现在读起来很碎。」

这句话翻译成 Claude Code 的指令：

```
读一下 ch5_analysis.docx 的全部内容。
当前有5节，列出每节的第一句话和最后一句话。
告诉我：
1. 这5节之间的逻辑关系是什么（递进？并列？因果？）
2. 有没有哪两节的内容可以合并
3. 节与节之间有没有缺少过渡（读者能否顺畅地从一节读到下一节）
不要修改，只给分析。
```

导师说「太啰嗦」，翻译成：

```
读一下第五章第3节，标出以下问题：
1. 哪些句子在重复前面已经说过的信息
2. 哪些描述性的话可以用一个数据或一个引用替代
3. 有没有连续两段在说同一件事
列出来，不要自动删除。
```

第四步：分天执行

40 多条意见，按章节分组、按优先级排序之后，我分了四天执行。每天一章，每章做完整体检查一遍再进入下一章。不赶进度，不在一天里做太多事。

每天做完之后，更新一下 checkpoint.md，记录哪些做完了、哪些还没做、有没有新发现的问题。这样即使某天的会话被打断了，第二天接着做也不会漏掉。

© 2026 晨瀚宇 未经授权禁止转载

表 5.1: 40 多条修改意见的四天执行计划

时间	章节	工作内容
第 1 天	第三章（8 条）	6 条小修改（替换术语、修正数字）+ 2 条逻辑调整
第 2 天	第四章 + 第五章前半	第四章 6 条全部完成 + 第五章小修改先做掉
第 3 天	第五章后半 + 第七章	第五章结构调整 + 第七章 9 条
第 4 天	全局性修改 + 通读	7 条跨章节的问题（术语统一、格式调整）+ 全文通读一遍

注 这四天的工作量，如果纯手动做，保守估计两周。Claude Code 把「执行」部分的时间压缩了很多，但「理解导师意图」和「决定怎么改」的时间是省不掉的。那半天的结构化整理看起来没有产出，但它让后面四天的执行变得很顺畅，每一条改什么、怎么改、用什么指令，心里都是清楚的。

5.5 实操：用 /docx 修改 Word 章节

大多数同学的论文都是 Word 格式，这一节走一遍用 Claude Code 修改 Word 章节的完整流程。

第一步：备份

在做任何修改之前，先备份（备份命令见第一章）。

第二步：用 /docx 读取内容

```
/docx 读一下 ch5_analysis.docx 第3节的内容，
列出每一段的第一句话。
```

先读再改，确认 Claude Code 正确识别了文档结构。

第三步：给出修改指令

```
/docx 修改 ch5_analysis.docx 第3节，要求：
1. 删掉第2段和第4段的重复内容（保留第2段的表述）
2. 第3段的「因变量」改为「被解释变量」
3. 第5段末尾补一句：与表5-3的结果一致
4. 修改后给我看改前改后的对比

以下术语不得修改：「被解释变量」「双向固定效应」「空间杜宾模型」
```

第四步：检查对比

它改完会给你一份对比。重点检查三件事：术语有没有被改、引用编号有没有被删、数字有没有被动过。这三类问题最隐蔽，也最致命。

第五步：验证文件

确认没问题后，用 Word 打开修改后的文件，检查格式有没有乱。Word 文件存在 XML 损坏的风险（详见第八章），每次改完都打开看一眼，发现问题就用备份恢复。如果频繁碰到格式损坏，考虑转 LaTeX（第八章详细讲）。

这个五步流程（备份 → 读取 → 修改 → 检查 → 验证）适用于所有 Word 文档的修改任务。用几次就会变成习惯。

写正文章节这块的经验就讲到这里。下一章讲图表。学术论文里图表的要求比较严格，Claude Code 能帮上忙的地方也不少。

© 2026 晨瀚宇 未经授权禁止转载

第六章 图表制作

内容提要

- 用 matplotlib 生成数据驱动的统计图，包括直方图、折线图和分组柱状图
- 用/drawio画研究框架图和方法流程图等概念关系类图
- Draw.io MCP 的 session 状态不持久，必须及时将图保存为本地 XML 文件
- 图表定稿前用字号、配色、元素重叠、分辨率四项标准做规范自检

我论文里一共有 9 张图和 14 张表。表格大部分是回归结果表，从 Stata 导出之后格式就基本定了，调整空间不大。图才是花时间的地方。

研究框架图、空间溢出效应示意图、变量关系图，这几张图导师反复改了好几版。第一版用 PowerPoint 画的，导师说线条粗细不统一、字号忽大忽小、对齐全靠目测。第二版我换了 Visio，稍微好了一些，但导师又说配色太花、不像学术论文的图。折腾了三版之后，我开始用 Claude Code 来做这件事。

用 Claude Code 画图有两条路线。一条是用 Python 代码生成图表，适合数据驱动的图（折线图、柱状图、散点图）。另一条是用 Draw.io 画流程图和架构图，适合概念关系类的图（研究框架图、方法流程图、变量关系图）。两条路线我都用了，加起来前后开了一百多次会话。这一章把两条路线的经验都讲清楚，包括我踩过的一个很大的坑。

6.1 用代码生成学术图表

我第一次让 Claude Code 帮我画图，是一张描述统计量的分布图。论文第四章有一节是变量的描述性分析，导师要求配一张核心被解释变量的分布直方图，看看数据有没有明显的偏态。以前我在 Stata 里画过，但 Stata 出的图默认样式比较朴素，字体也不好调，导师说「看着不舒服，换个工具画」。

我在 Claude Code 里说了一句：

```
读一下 data/panel_data.csv，画一张「Variable-A」的分布直方图。
要求：学术论文风格，字体用宋体，坐标轴标签用中文，
分辨率300dpi，保存为 figures/fig4_1.png。
```

它用 matplotlib 写了一段 Python 代码，跑完直接生成了图片。我打开一看，基本形状没问题，但有几个细节需要调：x 轴标签和柱子重叠了，图例位置挡住了一部分数据，字号偏小在 A4 纸上看不清。我把这三个问题告诉它，它改了代码重新跑，第二版就能用了。

整个过程不到十分钟。如果我自己写 matplotlib 代码，光是查怎么设置中文字体、怎么调坐标轴间距、怎么设置分辨率，就要翻半小时文档。Claude Code 把这些琐碎的参数配置全替你处理了，你只需要告诉它你想要什么样子。

稍微复杂一点的图

我论文第五章有一张对比图，要把三个子样本的回归系数放在一张图里横向对比。数据是这样的：全样本的系数是 0.342，东部子样本是 0.521，中西部子样本是 0.187。导师要求三个柱子并排放，每个柱子上面标系数值和显著性星号，还要画一条 y=0 的虚线作为参考。

© 2026 晨瀚宇 未经授权禁止转载

我的指令是这样写的：

```
画一张分组柱状图，数据如下：
- 全样本：系数0.342***
- 东部子样本：系数0.521***
- 中西部子样本：系数0.187*

要求：
- 每个柱子上方标注系数值和显著性星号
- 画一条 y=0 的灰色虚线
- y轴标签：「回归系数」
- x轴标签：「样本分组」
- 学术论文风格，宋体，300dpi
- 颜色用灰度（不用彩色，黑白打印要能看清）
- 保存为 figures/fig5_2.png
```

Claude Code 自己写了一段 Python 代码，跑完就生成了图片。你不需要会写代码，只需要把数据和要求说清楚。

中文字体的坑

用 matplotlib 画中文图表有一个绕不开的问题：字体。matplotlib 默认不支持中文，如果不指定中文字体，图上所有中文都会变成方块。

我第一次遇到这个问题的时候，Claude Code 自己试了好几种字体配置。先用了黑体（SimHei），在 macOS 上不认；换了 Arial Unicode MS，中文能显示了，但导师说字体不像学术论文的，要用宋体。最后 Claude Code 找到 macOS 自带的华文宋体（STSong），字体终于对了。

这个过程前后折腾了三次。如果你碰到中文变方块的问题，告诉 Claude Code 你的操作系统和想要的字体，它会帮你找到对应的配置。Windows 一般用 SimSun，macOS 用 STSong，Linux 需要自己安装字体包。把最终确定的字体写进 CLAUDE.md，以后每次画图它都会自动用对的字体。

从简单图到复杂图

建议从最简单的图开始试。第一张图画一个单变量的直方图或折线图，只涉及一组数据、一个坐标轴，确认字体、字号、分辨率都没问题之后，再去画多子图、多系列的复杂图。

我的论文里最复杂的一张图是空间溢出效应的分解图，要把直接效应、间接效应、总效应三条线画在一张图里，每条线还带置信区间的阴影。这张图如果一开始就让 Claude Code 画，它给出来的代码会很长，出了问题不知道改哪里。我的做法是分步来：先画一条线，确认样式没问题；再加第二条线，调颜色和图例；最后加置信区间的阴影。每一步都是在上一步的基础上改几行代码，出了问题很容易定位。

Claude Code 在这个过程中做的事情是：你描述要加什么，它改代码，跑一遍，生成新图片。你看一眼不满意，说哪里要调，它再改再跑。这个「描述 → 生成 → 调整」的循环很快，一张图从第一版到定稿，通常三四轮就够了。

一个实际的提醒：每次让 Claude Code 生成图表之后，一定要自己打开图片看一遍。它生成的代码不一定有语法错误，图片也确实能生成出来，但内容可能不对。我有一次让它画一张包含三个子样本的对比图，它把东部和中西部的数据画反了。代码没报错，图片看着也完整，但柱子标的数字和我给的数据对不上。如果我没有逐个核对数字，这张错图就直接放论文里了。

© 2026 晨瀚宇 未经授权禁止转载

图表这块还有一个细节：保存格式。如果你的论文用 LaTeX 排版，建议保存为 PDF 格式而不是 PNG。PDF 是矢量格式，放大不会模糊，印刷效果好。matplotlib 保存 PDF 只需要把文件名后缀改成.pdf 就行，其他都不用变。如果论文用 Word 排版，PNG 格式就够了，300dpi 的分辨率足以满足打印需求。

定义 6.1 (矢量图)

矢量图（Vector Graphics）用数学公式描述线条、曲线和填充区域，代表格式有 PDF 和 SVG。与位图（PNG、JPG）不同，矢量图无论放大多少倍都不会失真，印刷质量高，特别适合学术论文中对清晰度有要求的图表。

♣

下面用一个稍微复杂的例子完整走一遍流程：画一张核密度估计（KDE）图。这类图在计量论文里很常见，用来展示变量分布随时间的动态变化。

实操：画核密度估计图

我论文第三章需要展示核心被解释变量在不同年份和不同区域的分布演变。导师要求四个子图（全样本、东部、中部、西部），每个子图里画四个年份的核密度曲线。以前我用 Stata 的 `kdensity` 命令画过，但四个子图拼在一起的排版很难调。

我给 Claude Code 的指令：

```
读一下 data/panel_data.csv，画一张核密度估计图。

数据列：Y（Variable-A），year（年份），region（地区分组：
东部/中部/西部）。

要求：
- 2x2 四个子图：全样本、东部、中部、西部
- 每个子图画 2014、2017、2020、2023 四条核密度曲线
- 不同年份用不同线型（实线、虚线、点划线）和颜色区分
- 曲线下方加轻微填充（alpha=0.08）
- x轴标签：「Variable-A」，y轴标签：「核密度」
- 学术论文风格，宋体，隐藏顶部和右侧边框
- 300dpi，保存为 figures/fig3_kde.png
```

Claude Code 生成的核心代码：

```
from scipy.stats import gaussian_kde
import numpy as np
import matplotlib.pyplot as plt

selected_years = [2014, 2017, 2020, 2023]
colors = {2014: '#2E5E8E', 2017: '#27AE60',
          2020: '#E67E22', 2023: '#C0392B'}
styles = {2014: '-', 2017: '--', 2020: '-.', 2023: '-'}

fig, axes = plt.subplots(2, 2, figsize=(12, 8))
titles = ['全样本', '东部', '中部', '西部']
```


© 2026 晨瀚宇 未经授权禁止转载

```
filters = [None, '东部', '中部', '西部']

for ax, title, region in zip(axes.flat, titles, filters):
    for yr in selected_years:
        if region is None:
            data = df[df['year'] == yr]['Y'].values
        else:
            mask = (df['year'] == yr) & (df['region'] == region)
            data = df[mask]['Y'].values
        if len(data) > 2:
            kde = gaussian_kde(data, bw_method='silverman')
            x = np.linspace(data.min() - 0.02,
                            data.max() + 0.05, 300)
            ax.plot(x, kde(x), linestyle=styles[yr],
                    color=colors[yr], linewidth=1.8,
                    label=f'{yr}')
            ax.fill_between(x, kde(x), alpha=0.08,
                            color=colors[yr])
    ax.set_title(title, fontsize=13, fontweight='bold')
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.set_xlabel('Variable-A', fontsize=10)
    ax.set_ylabel('核密度', fontsize=10)
    ax.legend(fontsize=9)

plt.tight_layout()
fig.savefig('figures/fig3_kde.png', dpi=300)
```

这段代码的关键是 `gaussian_kde`，它是 scipy 自带的核密度估计函数，`bw_method='silverman'` 是最常用的带宽选择方法。你不需要自己写这些，Claude Code 会根据你的描述自动选择合适的参数。

生成效果如图 6.1：四个子图分别展示了全样本和三大区域的分布变化，能清楚看出分布形态在不同年份的右移趋势。导师看了第一版就通过了，只让我把图例从左上角挪到了右上角。

用代码画图这条路线，适合所有有明确数据的图表。折线图、柱状图、散点图、箱线图、分布图，只要你能说清楚 x 轴是什么、y 轴是什么、数据从哪来，Claude Code 就能帮你画出来。接下来讲另一条路线：没有数据、纯粹是概念关系的图，比如研究框架图和方法流程图。

6.2 流程图与架构图

研究框架图是论文里最让我头疼的一张图。它没有数据，全靠你自己想清楚各个概念之间的关系，然后用方框和箭头表达出来。画得好，读者看一眼就明白你的研究是怎么设计的；画得不好，读者看完更糊涂。

我的研究框架图经历了五个版本。前三版是用 PowerPoint 画的，导师每次都不满意：第一版说「方框大小不统一、间距不等」，第二版说「箭头方向不清楚，因果关系看不出来」，第三版说「颜色太多，花里胡哨」。

© 2026 晨瀚宇 未经授权禁止转载

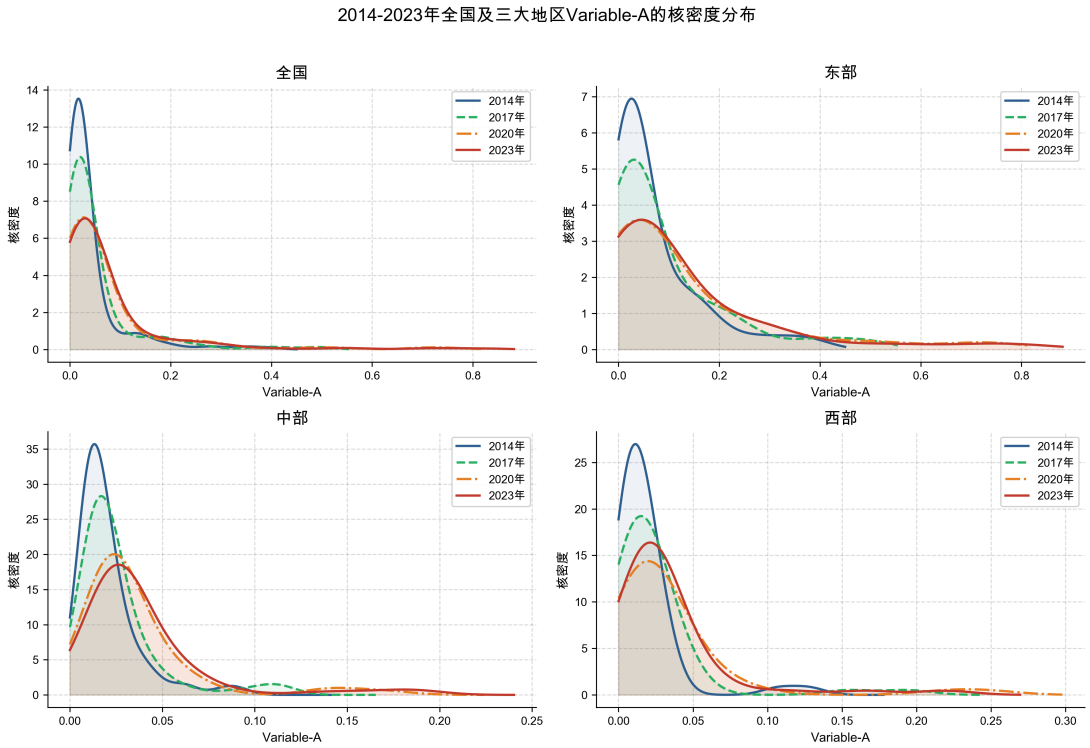


图 6.1: 核密度估计图示例：核心被解释变量在不同年份和区域的分布演变

第四版开始我用 Claude Code 的 `/drawio` 功能。Draw.io 是一个开源的画图工具，Claude Code 通过 MCP（Model Context Protocol）协议可以直接调用它。你在对话里输入 `/drawio`，它就能创建、编辑、导出 Draw.io 格式的图。

定义 6.2 (Draw.io)

Draw.io 是一款基于浏览器的开源矢量绘图工具，专为流程图、架构图和关系图设计。其底层文件格式为 XML，Claude Code 通过 MCP 协议调用后，可在对话中直接用自然语言创建和修改图表，并导出为 PNG、PDF 等格式。

第一次用 /drawio

我第一次用 `/drawio` 画研究框架图，指令是这样写的：

```
/drawio 帮我画一张研究框架图，按论文章节结构从上到下排列。

第一层（绪论）：横排四个方框：
  研究背景及意义 | 文献综述 | 研究内容与方法 | 创新与不足

第二层（理论基础与作用机制）：
  左侧一组「理论基础」，下挂两个子框：内生增长理论、外部性理论
  右侧一组「作用机制」，下挂三个子框：直接影响、间接影响、
  空间溢出影响

第三层（现状分析）：横排三个方框：
  发展现状 | 指标体系构建 | 时空分析与动态演进
```


© 2026 晨瀚宇 未经授权禁止转载

```
第四层（实证结果分析）：
  上排三个方框：研究假设 | 数据来源与指标选取 | 模型构建
  下排三个方框：基准回归与稳健性检验 | 机制检验 | 空间效应分析

第五层（研究结论与建议）：横排两个方框：
  研究结论 | 相关建议

每一层左侧用虚线框标注章节名称，层与层之间用向下箭头连接。
风格要求：方框用白底黑边，字号14pt，整体居中对齐，
学术论文风格，黑白配色。
```

研究框架图和其他图不一样：它展示的是论文的完整章节结构和逻辑主线，不是某一章的技术细节。你要把「绪论讲了什么、理论基础是什么、用了什么方法、做了哪些实证、最后得出什么结论」的全貌画出来。读者看一眼就应该明白你的论文是怎么组织的。

Claude Code 创建了一个 Draw.io 的 session，把图画出来了。我让它导出为 PNG 看了一眼，基本结构是对的，但有几个问题：方框大小不一样（因为文字长度不同，方框自动适应了文字宽度），层与层之间的间距不均匀。我让它把方框统一成固定宽度、间距统一，改了两轮就定稿了。图 6.2是最终的效果。

到这一步为止，一切都很顺利。问题出在第二天。

教训：Draw.io MCP 的 session 状态会丢失

第二天我打开 Claude Code，想在昨天的研究框架图上加一些标注（在箭头旁边写上路径系数）。我输入了 `/drawio`，让它在图上加标注。它告诉我：没有找到任何已打开的图表，需要重新创建。

昨天花了四十多分钟调好的图，一夜之间全没了。

这就是教训 8 的完整经过。Draw.io MCP 的工作方式是这样的：每次你调用 `/drawio`，它会创建一个 session，所有的编辑操作都在这个 session 里进行。但这个 session 是临时的，存在内存里，不会自动保存到磁盘。当你关掉 Claude Code（或者切换到一个新的对话 turn），session 就消失了，图也跟着消失了。

我当时没有意识到这一点。因为在同一次对话里，我可以反复修改、导出、查看，一切正常，感觉图是「保存」了的。实际上它只是在 session 的内存里，从来没有写入文件。

丢了图之后我很沮丧，因为那张图不光是画出来的，还经历了好几轮调整（方框对齐、箭头间距、字号统一），这些微调的成果全部丢失了。重新画一遍不难，但重新调一遍很烦，因为我已经不记得每个参数具体改了什么。

注 Draw.io MCP 的 session 存在内存中，关闭 Claude Code 或切换会话后即消失。每次修改满意后必须立刻把图的 XML 导出保存到本地.drawio 文件，下次重新打开时先读取文件恢复状态再继续编辑。不养成这个习惯，之前花在调整上的所有工夫都会白费。

后来我找到了正确的做法：**每次成功修改后，立刻把 XML 保存到本地文件。**

Draw.io 的图底层是 XML 格式的。你让 Claude Code 把当前图的 XML 内容导出，保存到一个.drawio 文件里。下次打开 Claude Code，先让它读这个文件，从文件里恢复图的状态，再继续编辑。文件是持久的，不会因为 session 消失而丢失。

具体操作是这样的：

```
# 每次修改满意后，执行这一步：
```


© 2026 晨瀚宇 未经授权禁止转载

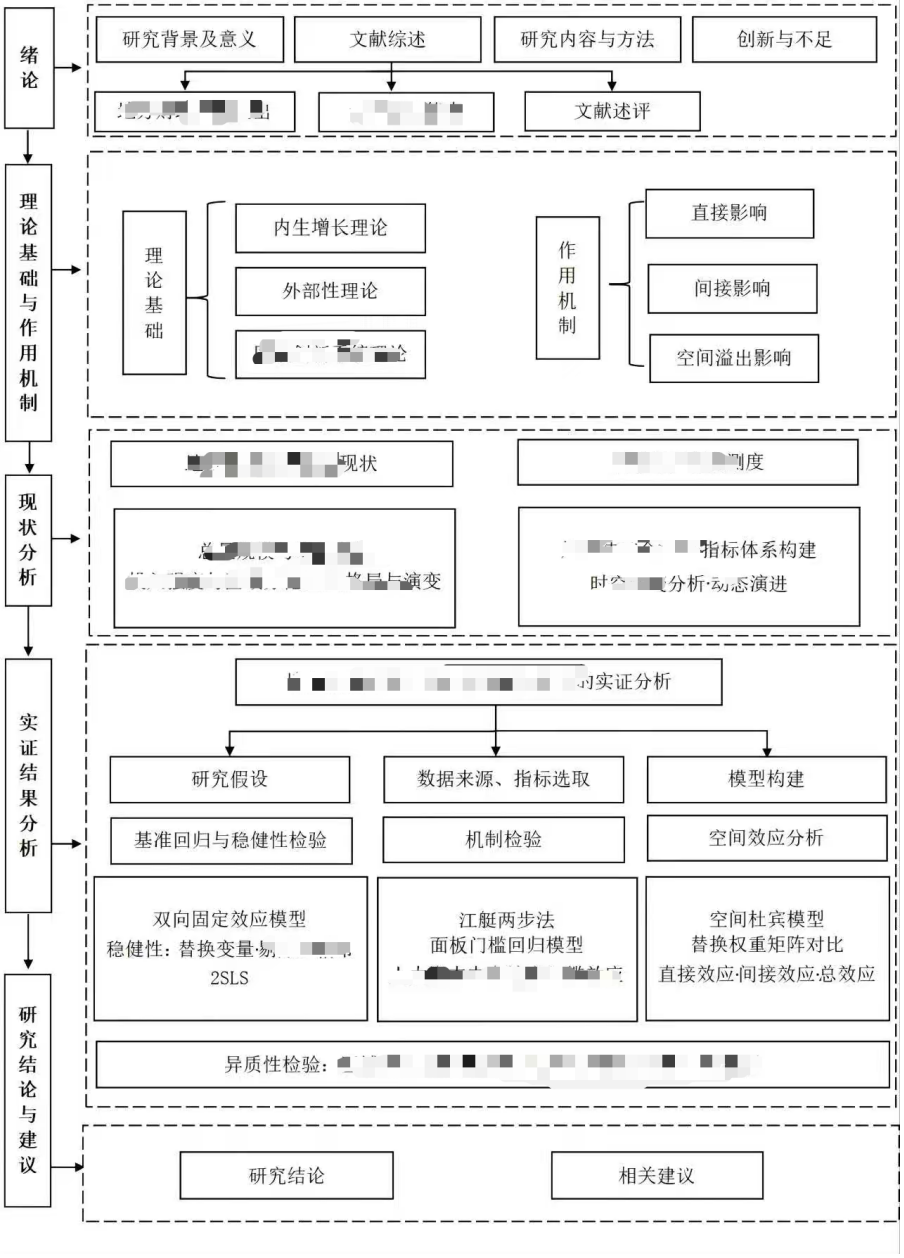


图 6.2: 研究框架图示例：按论文章节结构从绪论到结论的完整逻辑主线

© 2026 晨瀚宇 未经授权禁止转载

```
/drawio 把当前图的 XML 导出，保存到 figures/framework.drawio

# 下次重新打开 Claude Code，先恢复：
/drawio 读取 figures/framework.drawio，继续编辑
```

从那以后，我每修改一版就保存一次 XML。图的所有状态都在这个文件里，session 丢了也不怕。后来我又养成了一个更细致的习惯：每次大改之前，先把当前版本的 XML 复制一份加上日期后缀（比如 framework_20260315.drawio），相当于手动版本管理。这样即使改坏了，也能退回到上一个满意的版本。

Draw.io 和代码画图的区别

数据图和概念图适合用不同的工具。matplotlib 擅长有数据的统计图，Draw.io 擅长没有数据的关系图和流程图。还有一个选择是 LaTeX 的 TikZ 包，它输出的图和论文字体完全一致，排版效果最好，但学习成本高。如果你和我一样没有 TikZ 基础，Draw.io 导出 PNG 或 PDF 已经够用了。

表 6.1: 三种流程图绘制方式的对比

	matplotlib	Draw.io（/drawio）	TikZ
适合的图类型	数据驱动的统计图	流程图、架构图、关系图	所有类型（学习成本高）
编辑方式	写 Python 代码	自然语言描述 + XML	写 LaTeX 代码
对齐和布局	手动计算坐标	自动对齐、可拖拽	相对定位，语法复杂
中文支持	需配置字体	原生支持	需配置 ctex
输出格式	PNG/PDF/SVG	PNG/PDF/SVG	PDF（原生矢量）
与 LaTeX 一致性	字体可能不同	字体可能不同	完全一致

我自己的做法是：数据图用 matplotlib，概念图用 Draw.io。两种工具配合起来，覆盖了论文里所有类型的图。

用 /drawio 的 104 次会话

我前后用 `/drawio` 开了 104 次会话。这个数字看起来很多，但 Draw.io MCP 有一个特点：因为 session 容易丢失，很多次会话其实是在重复恢复和继续编辑同一张图。如果 session 没有丢失的问题，可能 50 次就够了。

这 104 次会话里，大约有 60 次是在画和修改研究框架图（这张图导师改了五版），20 次左右是画方法流程图（描述研究方法的步骤），剩下的分散在变量关系图、空间权重矩阵示意图等几张图上。每张图从第一版到定稿，平均需要经历三到四轮修改。每轮修改包括：Claude Code 画出来、我导出看一眼、发现问题、让它改。一轮大概十分钟，一张图半小时到一小时能定稿。

和用 PowerPoint 画图相比，用 `/drawio` 的效率提升主要在两个地方。第一个是对齐。PowerPoint 里要把五六个方框对齐，需要全选、点对齐按钮、选择对齐方式，有时候还对不准。用 `/drawio`，你说「把这三个方框水平居中对齐、间距统一」，它自己就处理好了。第二个是批量修改。比如导师说「所有方框的字号从 12pt 改成 14pt」，在 PowerPoint 里你得一个一个框点进去改；用 `/drawio`，一句话它就把所有方框的字号统一改了。

图表画好之后，还有一个步骤容易被忽略：检查图表是否符合学术规范。这是下一节的内容。

6.3 图表规范自检

导师退回我的图的时候，说的最多的三句话是：「字太小了」「颜色太多了」「挤在一起看不清」。这三个问题我在九张图里犯了七次。后来我总结出一套自检流程，每张图定稿之前过一遍，基本能避免被退回。

字号

© 2026 晨瀚宇 未经授权禁止转载

学术论文的图表，最终是要打印在 A4 纸上的。电脑屏幕上看着清楚的字号，打印出来可能小到看不清。我的经验是：图表里的文字，最小不要低于 10pt。坐标轴标签、图例文字建议用 12pt，标题用 14pt。

一个简单的检查方法：把图片插入 Word 或 LaTeX 文档，编译或打印出来看。屏幕上看着合适的字号，打印出来往往偏小。我有一张图在屏幕上看没任何问题，打印出来之后 x 轴的标签挤成了一团，根本看不清。后来让 Claude Code 把 x 轴标签旋转了 45 度、加大了字号，问题才解决。

字号的具体参数不用你自己设，把「标题 14pt、标签 12pt、刻度 10pt」这类要求写进 CLAUDE.md，Claude Code 每次画图都会自动按这个规范来。

颜色

学术论文的图表配色有两个原则：一是黑白打印能看清，二是色觉障碍人群能分辨。很多期刊的审稿指南里明确要求图表在黑白打印时仍然可读。

我一开始画图喜欢用蓝、红、绿三种颜色来区分不同的数据系列。在屏幕上很好看，但导师说：「打印出来红色和绿色几乎一样深，分不清。」后来我改成了灰度配色：第一组数据用深灰（#333333），第二组用中灰（#888888），第三组用浅灰（#CCCCCC）。如果灰度不够区分，就用不同的线型（实线、虚线、点线）或不同的标记（圆点、方块、三角）来辅助。

一个实用的检查方法：在电脑上把图片转成灰度模式看一眼。macOS 可以在预览里调，Windows 可以用画图工具。如果转成灰度之后两组数据的颜色混在一起，就说明配色方案有问题。

元素重叠

图表里最常见的重叠问题有三种：数据标签和数据点重叠、图例和数据区域重叠、坐标轴标签相互重叠。

数据标签重叠最容易出现在柱状图上。我有一张回归系数对比图，五个柱子的系数值标在上方，其中两个值很接近（0.342 和 0.351），标签几乎完全重叠。我告诉 Claude Code 这个问题，它把两个标签一个放上方、一个放下方，用细线指向对应的柱子，问题就解决了。

图例重叠也很常见。默认图例放在右上角，但如果右上角恰好有数据就会盖住。碰到这种情况，告诉 Claude Code「把图例移到图的右侧外面」或者「放到左下角」，它自己会处理位置参数。

坐标轴标签重叠通常出现在 x 轴类别多的时候。比如你有 15 个省份名作为 x 轴标签，横着排一定挤在一起。告诉 Claude Code「x 轴标签旋转 45 度」就行。如果旋转之后还是看不清，可以让它换成水平条形图，标签竖着排在左边。

自检清单

每张图定稿之前，我会过一遍这个清单：

表 6.2: 图表规范自检清单

检查项	具体标准
字号是否够大	打印在 A4 纸上，最小文字不低于 10pt
黑白可读性	转成灰度后，不同数据系列仍可区分
元素有无重叠	标签、图例、数据点互不遮挡
对齐是否整齐	方框大小统一、间距均匀、箭头方向一致
分辨率是否够	PNG 至少 300dpi，或直接用 PDF 矢量格式
字体是否统一	图表字体和正文字体一致（宋体配宋体）

这个清单也可以直接写进 CLAUDE.md，让 Claude Code 每次生成图表后自动对照检查。我后来就是这么做的：在 CLAUDE.md 里加了一段「图表输出规范」，包括字号、配色、分辨率的默认值。Claude Code 每次画图都会按这些规范来，不需要我在每条指令里重复说。

让 Claude Code 自己检查的指令长这样：

© 2026 晨瀚宇 未经授权禁止转载

```
看一下 figures/fig5_2.png，对照以下标准检查：
1. 最小文字是否低于10pt
2. 有没有元素重叠
3. 配色是否黑白可读
告诉我有没有需要改的地方。
```

它能通过读取图片来做基本的视觉检查。但这种检查不如你自己看一眼来得准确。我的建议是：先让它自查一遍抓住明显问题，然后自己再看一遍，重点看打印效果（导出到文档里、缩放到实际尺寸再看）。

规范检查做完，基本流程就走通了：用 matplotlib 画数据图、用 /drawio 画概念图、画完过一遍自检清单。下一节把 /drawio 画研究框架图的完整操作流程走一遍，从头到尾，包括怎么保存 XML、怎么恢复编辑、怎么导出最终版本。

6.4 实操：用 /drawio 画研究框架图

这一节用一个完整的例子走一遍 /drawio 的操作流程。我用的是我论文里的研究框架图，做了脱敏处理，你可以把变量名换成你自己研究里的概念。

准备工作

在动手画之前，先把图的结构想清楚。研究框架图要表达的是变量之间的关系：哪个变量影响哪个变量、是直接影响还是通过中介变量间接影响、有没有调节效应。

我的研究框架图的结构是这样的：核心解释变量在最上面，通过三条路径影响被解释变量。第一条是直接路径（直接效应），第二条和第三条分别经过两个中介变量（间接效应）。右边还有一个调节变量，调节的是核心解释变量到被解释变量的直接路径。

在纸上或者脑子里把这个结构理清楚再开始画。如果你自己都不确定变量之间是什么关系，画出来的图只会让读者更困惑。

第一步：创建图表

打开终端，进入论文文件夹，启动 Claude Code，输入：

```
/drawio 创建一张新的研究框架图。

布局从上到下，共三层：
第一层（顶部）：一个方框，写「核心解释变量」
第二层（中间）：两个方框，分别写「中介变量A」「中介变量B」
第三层（底部）：一个方框，写「被解释变量」
右侧：一个方框，写「调节变量」

箭头关系：
- 核心解释变量 → 中介变量A（实线箭头）
- 核心解释变量 → 中介变量B（实线箭头）
- 中介变量A → 被解释变量（实线箭头）
- 中介变量B → 被解释变量（实线箭头）
- 核心解释变量 → 被解释变量（虚线箭头，标注「直接效应」）
```


© 2026 晨瀚宇 未经授权禁止转载

```
- 调节变量 → 核心解释变量到被解释变量的直接路径
  （虚线箭头，标注「调节效应」）

风格：白底、黑边方框、字号14pt、箭头黑色、
整体居中、学术论文风格。
```

这条指令比较长，但每一部分都有明确的作用：布局描述告诉它元素的空间位置，箭头关系描述告诉它连线方式，风格描述告诉它视觉规范。指令越具体，生成的图越接近你的预期，后续修改轮数越少。

第二步：导出查看

Claude Code 创建图之后，让它导出为 PNG：

```
/drawio 导出为 PNG，保存到 figures/framework_v1.png
```

打开图片看一眼。第一版通常会有几个问题需要调整。我第一版遇到的问题是：两个中介变量的方框间距太大，和上下两层的视觉重心不对齐。调节变量的箭头指向了虚线箭头的中间点，但因为虚线箭头本身比较长，调节箭头看起来不太准确。

第三步：保存 XML（关键步骤）

看完第一版，不管满不满意，先把 XML 保存下来：

```
/drawio 把当前图的 XML 保存到 figures/framework.drawio
```

这一步一定不能忘。前面讲过的教训就是在这一步出了问题。只要 XML 文件在，session 丢了你也能恢复。从现在开始，每做一轮修改、每次对结果满意之后，都执行一次这个保存命令。

第四步：修改调整

根据第一版的问题，给出修改指令：

```
/drawio 做以下调整：
1. 两个中介变量的方框间距缩小一些，看着太散了
2. 所有方框统一大小
3. 调节变量的箭头指向直接效应那条虚线的中点
4. 所有箭头统一粗细
```

你不需要给精确的像素值，Claude Code 会自己选合适的参数。描述清楚「想要什么效果」就够了，具体怎么实现是它的事。修改之后再导出 PNG 看一眼，这一轮通常就接近最终版了。如果还有细节问题，继续修改。

每轮修改后都执行一次保存 XML 的命令。我习惯在文件名里加版本号：

```
/drawio 把当前 XML 保存到 figures/framework_v2.drawio
```

这样 v1、v2、v3 都在，随时可以回退。

第五步：最终导出

确认图表内容和样式都没问题后，导出最终版本：

```
/drawio 导出为 PNG，分辨率 300dpi，
保存到 figures/fig3_1_framework.png

# 如果论文用 LaTeX，也导出一份 PDF：
```


© 2026 晨瀚宇 未经授权禁止转载

```
/drawio 导出为 PDF，保存到 figures/fig3_1_framework.pdf
```

文件名建议用「章节号 _ 序号 _ 描述」的格式（fig3_1_framework），这样在文件夹里一眼就能看出这张图属于哪一章、排在第几张。

第六步：从文件恢复编辑

如果过了几天，导师看完说「框架图要改一下，中介变量加一个」，你打开 Claude Code，不需要重新画：

```
/drawio 读取 figures/framework_v2.drawio

在中介变量A和中介变量B之间加一个新方框，
写「中介变量C」，宽度和其他方框一致。
核心解释变量增加一条箭头指向中介变量C，
中介变量C增加一条箭头指向被解释变量。
```

它会从文件里恢复之前的图，在这个基础上继续编辑。所有之前调好的格式（方框大小、间距、箭头粗细）都在 XML 里保留着，不用重新调。

这个「保存 XML→ 从文件恢复 → 继续编辑」的流程是用 /drawio 画图的核心工作模式。一旦养成了这个习惯，Draw.io MCP 的 session 丢失就不再是问题了。

完整流程总结

把上面六步串起来，就是用 /drawio 画一张图的完整流程：

```
1. 想清楚图的结构，用文字描述给 Claude Code
2. Claude Code 创建图，导出 PNG 查看第一版
3. 保存 XML 到本地文件（关键）
4. 根据第一版的问题，给修改指令
5. 修改→查看→保存 XML，循环直到满意
6. 导出最终版本（PNG 和/或 PDF）
```

我画了九张图，每张都是按这个流程走的。第一张图花的时间最长，因为在摸索 /drawio 的用法和保存习惯。从第二张图开始就快了，一张图半小时左右就能定稿。

有一个额外的建议：如果你有多张图要画，先把最复杂的那张图的风格调好（方框样式、箭头粗细、字号、配色），然后把这套风格参数写在一条指令模板里。后面画其他图的时候，直接复用这套模板，所有图的风格就是统一的。导师看到全文的图风格一致，观感会好很多。

图表这一章到这里就讲完了。下一章讲引用和参考文献。学术论文对引用格式的要求很严格，全部参考文献逐条核查格式是一件很枯燥但必须做的事，Claude Code 能帮上不少忙。

© 2026 晨瀚宇 未经授权禁止转载

第七章 引用与参考文献

内容提要

- BibTeX 条目的格式问题（字段缺失、作者名写法混乱、期刊名缩写）不影响编译但影响最终呈现
- 用/citation-management自动扫描格式问题，比手动逐条核查快十倍以上
- 中国高校学位论文通常要求遵循 GB/T 7714 格式规范，中文文献需特别注意language字段
- 正文引用与.bib 文件的交叉检查能发现拼写错误和重复引用等隐蔽问题

我论文的参考文献，最终定稿时有 156 条。这 156 条的 BibTeX 来源五花八门：有从知网导出的、有从 Google Scholar 一键复制的、有自己对着期刊网站手动敲进去的、还有用 Zotero 抓取之后导出的。四种来源，四种格式风格，混在同一个 references.bib 文件里。

打开这个文件，肉眼就能看出问题。有的条目作者名写成"Zhang, Wei"，有的写成"Wei Zhang"，有的用全名"张伟"，有的用拼音缩写"W. Zhang"。期刊名也是，有的用缩写"J. Econometrics"，有的用全称"Journal of Econometrics"，有的甚至夹着中文全角逗号。更隐蔽的问题是：有两条引用其实是同一篇文章，只不过一条是我从知网导出的，另一条是从 Google Scholar 复制的，citation key 不一样，BibTeX 字段的顺序和写法也不一样，所以一直没发现重复了。

这些问题，一条两条不影响编译。但 156 条里夹着十几处格式不统一，提交给学校的时候，参考文献列表看起来就很乱：有的作者名是"张伟"，紧接着下一条变成"W. Zhang"，再下一条又是"Zhang, W."。答辩委员会的老师翻到参考文献那几页，第一印象就打了折扣。

这一章讲的是我怎么用 Claude Code 把这个乱七八糟的 BibTeX 文件清理干净的过程。第四章讲过 156 篇引用的核查故事，那次侧重的是"信息是否正确"（年份有没有错、DOI 能不能解析、有没有被撤回的文献）。这一章聚焦的是另一个维度：格式是否规范。信息对了但格式不统一，在学位论文答辩里同样会被挑出来。

7.1 BibTeX 格式管理

写论文之前，我对 BibTeX 的了解仅限于"它是 LaTeX 的参考文献格式"。知网导出的时候选 BibTeX 格式，复制到.bib 文件里，LaTeX 编译的时候能识别，就行了。具体每个字段是什么意思、哪些字段是必填的、字段的写法有什么规范，我完全不清楚。这种"能编译就行"的心态，到了提交前一周才让我付出代价。

定义 7.1 (BibTeX)

BibTeX 是 LaTeX 配套的参考文献数据库格式，以.bib文件存储。每条引用由类型（如@article、@book）、唯一的 citation key，以及 author、title、journal、year 等字段组成。LaTeX 编译时读取.bib 文件，按引用样式宏包的规则自动格式化参考文献列表。

BibTeX 条目的基本结构

一条 BibTeX 引用长这样：

```
@article{zhang2021spatial,
  author = {Zhang, Wei and Li, Ming},
```


© 2026 晨瀚宇 未经授权禁止转载

```
  title   = {Spatial Spillover Effects of Regional Policy},
  journal = {Journal of Econometrics},
  year    = {2021},
  volume  = {225},
  number  = {2},
  pages   = {134--158},
  doi     = {10.1016/j.jeconom.2021.03.005}
}
```

第一行的`@article`表示文献类型是期刊论文，花括号里的`zhang2021spatial`是 citation key，你在正文里用`\cite{zhang2021spatial}`来引用这条文献。后面的字段就是这篇文章的元信息：作者、标题、期刊名、年份、卷号、期号、页码、DOI。

不同类型的文献，比如期刊论文、会议论文、书籍、学位论文，必填字段各不相同。这些规则不用自己记，格式检查的时候 Claude Code 会按类型自动判断哪些字段缺失。真正让人头疼的地方在另一处：从不同来源获取的 BibTeX 条目，写法差异很大；这些差异在编译时不会报错，只有在最终的参考文献列表里才能看出来。

来源不同，格式差异有多大

我的 156 条引用来自四个渠道。知网导出的 BibTeX 条目，作者名用的是拼音全称（"Zhang Wei"，注意没有逗号分隔姓和名），标题有时候是中文有时候是英文翻译，期刊名用中文全称。Google Scholar 复制的 BibTeX，作者名用"Last, First" 格式（"Zhang, Wei"），期刊名有时候用缩写有时候用全称，取决于 Google Scholar 从哪个数据库抓取的元数据。Zotero 导出的 BibTeX 比较规范，但 Zotero 本身的数据也来自不同数据源，偶尔会有字段缺失。手动录入的就更不用说了，格式完全取决于我当时的心情和记忆。

下面这张表展示了同一篇文章从四个来源获取的 BibTeX 条目的差异：

表 7.1: 同一篇文献从四种来源获取的 BibTeX 差异

字段	知网导出	Google Scholar	Zotero 导出	手动录入
author	Zhang Wei	Zhang, Wei and Li, Ming	Zhang, Wei and Li, Ming	W. Zhang, M. Li
title	区域政策的空间溢出效应研究	Spatial Spillover Effects of Regional Policy	Spatial spillover effects of regional policy	Spatial Spillover Effects...
journal	计量经济学报	J. Econometrics	Journal of Econometrics	J Econometrics
year	2021	2021	2021	（漏填了）
pages	134-158	134–158	134–158	134-158

同一篇文章，四种写法，每种都能编译通过，但最终在参考文献列表里呈现的样子完全不一样。作者名的格式不统一，会导致参考文献列表里同一个人名出现三种写法。页码用单连字符（134-158）和用双连字符（134--158）在 LaTeX 里渲染的效果不同。期刊名的缩写和全称混用，看起来不专业。

这些问题，一条一条手动修正当然可以，但 156 条改下来要花很长时间，而且很容易漏改。

citation key 的命名混乱

除了字段内容的差异，还有一个问题：citation key。知网导出的 citation key 通常是一串编号（如`CNKI:SUN:JLXB.0.2021-02-003`），Google Scholar 生成的是" 作者 + 年份 + 首词"（如`zhang2021spatial`），Zotero 导出的格式取决于你的 Zotero 设置，手动录入的就看你自己怎么取名。

citation key 不统一带来的后果是：正文里你用`\cite{}`引用的时候，要记住每条引用的 key 长什么样。知网那种一长串编号，写正文的时候根本记不住，每次都要去.bib 文件里翻。更严重的是，如果同一篇文章被录入了

© 2026 晨瀚宇 未经授权禁止转载

两次（一次从知网导出，一次从 Google Scholar 复制），两个 citation key 不一样，你在正文里可能两个都引用了，最终参考文献列表里同一篇文章出现两次。

注 同一篇文章被录入两次、两个 citation key 不同，靠肉眼几乎无法发现。必须借助工具做自动化的重复检测。

我的 156 条引用里就有 3 组这样的重复。发现的时候是提交前三天，Claude Code 帮我做格式检查时找出来的。如果当时没有做这一步，提交上去参考文献里有三篇重复的文献，答辩的时候被问到会很尴尬。

怎么避免这些问题

回过头看，最好的做法是从一开始就统一来源和格式。但现实是，写论文是一个持续大半年的过程，不同阶段用不同渠道找文献，BibTeX 条目是零零散散加进来的，很难从一开始就保持统一。所以到了提交前做一次集中清理，几乎是不可避免的。

清理的思路很简单：确定一套标准格式，然后把所有条目都改成这个格式。下一节讲具体怎么做。

7.2 引用核查：自动找出错误引用

上一章讲过信息层面的核查（年份、DOI、撤回文献），这一节聚焦格式层面：BibTeX 条目的写法是否符合统一规范。信息正确不等于格式正确——一篇文章的元数据可以完全准确，但作者名写法、期刊名缩写、页码符号各不相同，照样会让参考文献列表看起来杂乱无章。

格式问题的三个层次

我在清理 BibTeX 文件的过程中，发现格式问题可以分成三个层次。

第一个层次是字段缺失。有些条目缺少必填字段，编译的时候不会报错（LaTeX 只会在参考文献列表里留一个空白或问号），但提交时会被发现。我的 156 条引用里有 4 条缺少 year 字段，都是手动录入的时候漏填了。还有 2 条缺少 journal 字段，因为它们其实是会议论文，我用了`@article`类型，journal 字段不知道该填什么就空着了。正确做法是改成`@inproceedings`类型，用 booktitle 字段填会议名称。

第二个层次是字段内容不规范。字段有值，但写法不统一。比如作者名，BibTeX 里规范的写法是”Last, First and Last, First”，用逗号分隔姓和名，用”and” 连接多个作者。但知网导出的格式是”Last First”（没有逗号），手动录入的有时候写成”F. Last”（缩写名在前）。这些写法 LaTeX 都能解析，但解析结果不一样，反映在参考文献列表里就是同一个人名三种显示方式。

第三个层次是中英文混排问题。我的论文有中文文献也有英文文献，中文文献的 BibTeX 条目需要额外的字段来告诉 LaTeX 这是中文文献（比如`language = {chinese}`），否则按英文规则排版会出问题。知网导出的条目通常带这个字段，但 Google Scholar 导出的中文文献 BibTeX 就不带，因为 Google Scholar 不区分语言。手动录入的更不会想到要加这个字段。

用 Claude Code 做第一轮格式扫描

信息核查用了并行 Agent（第四章讲过），格式核查我用的是`/citation-management`，不需要并行，因为格式检查不涉及网络请求（不用查 CrossRef API 验证 DOI），速度很快。

我给 Claude Code 的指令是这样的：

```
/citation-management 读取 references.bib，做格式检查：
1. 列出所有缺少必填字段的条目（按类型判断必填字段）
2. 检查作者名格式是否统一为 "Last, First and Last, First"
3. 检查期刊名是否全部使用全称
4. 找出所有重复引用（标题相似度超过80%的条目对）
```


© 2026 晨瀚宇 未经授权禁止转载

```
5. 检查中文文献是否都有 language 字段
把结果分类写入 format_issues.md，不要自动修改。
```

两分钟不到，它扫完了 156 条引用，写了一份报告。报告里把问题按类别分组，每组列出具体是哪些条目有问题。

结果是这样的：缺少必填字段的有 6 条（4 条缺 year，2 条类型用错了）。作者名格式不统一的有 23 条。期刊名混用缩写和全称的有 14 条。重复引用 3 组（6 条，两两重复）。中文文献缺少 language 字段的有 9 条。加起来，156 条里有几十处格式问题。

看到这个数字的时候我有点吃惊。我以为自己的 BibTeX 文件大致没问题，因为编译一直正常，参考文献列表也没有明显的报错。但" 编译正常" 和" 格式规范" 之间的差距，比我以为的大得多。

逐类修复

拿到问题列表之后，我按类别逐步处理。

先处理缺少必填字段的 6 条，因为这类问题影响最大。缺 year 的逐条查了原始文献补上，类型用错的会议论文改成`@inproceedings`，补上 booktitle 字段，删掉空的 journal 字段。

然后处理重复引用。3 组重复里，每组保留信息更完整的那条，删掉另一条。删之前要检查正文里有没有用到被删条目的 citation key，如果用了就替换成保留条目的 key。这一步让 Claude Code 做很合适：

```
references.bib 里有3组重复引用，每组保留第一条，删除第二条。
然后扫描所有 .tex 文件，把正文中引用了被删条目的 \cite{}
替换成保留条目的 citation key。
列出每处替换的位置和替换前后的内容。
```

它找到了 5 处正文引用需要替换（3 组重复里有 2 组在正文中都被引用过，共 5 处），逐一替换完毕，每处都给了替换前后的对比。

作者名格式统一、期刊名统一、language 字段补充这三类问题，放在下一节和 GB/T 7714 一起处理，因为这些问题的修改方式取决于你最终要遵循什么格式规范。

一个容易忽略的问题：正文引用和参考文献列表不匹配

格式扫描完之后，我多做了一步检查：让 Claude Code 对比正文里所有的`\cite{}`和 references.bib 里的所有 citation key，找出两边不匹配的。

```
扫描所有 .tex 文件中的 \cite{} 命令，提取所有引用的 citation key。
再读取 references.bib，提取所有条目的 citation key。
对比两个列表，告诉我：
1. 正文里引用了但 .bib 文件里不存在的 key
2. .bib 文件里存在但正文里从未引用过的条目
```

结果发现：正文里有 2 个`\cite{}`引用的 key 在.bib 文件里不存在（拼写错误，少了一个字母），.bib 文件里有 8 条引用在正文中从未被引用过（早期加进来后来删掉了正文引用但忘了删 BibTeX 条目）。

2 个拼写错误如果不修复，编译时参考文献列表里会显示问号，细心的答辩老师一眼就能看到。8 条多余的 BibTeX 条目不影响编译（LaTeX 只渲染被引用的条目），但如果用`\nocite{*}`命令（有些模板会加这个），所有条目都会出现在参考文献列表里，包括这 8 条没有被引用的。清理掉更保险。

这一步检查花了不到一分钟，但找到的问题都是手动排查很难发现的。手动排查意味着你要在好几个.tex 文件里逐个搜索每一条 citation key，再和.bib 文件交叉比对，156 条文献比对下来少说也要一两个小时。

© 2026 晨瀚宇 未经授权禁止转载

7.3 批量格式统一（GB/T 7714）

前面两节把 BibTeX 文件里的信息错误和结构问题清理完了。接下来要做的是：把所有条目的格式统一到同一套规范下。中国高校的学位论文，参考文献格式通常要求遵循 GB/T 7714，这是中国国家标准，全称是《信息与文献　参考文献著录规则》。

定义 7.2 (GB/T 7714)

GB/T 7714 是中国国家标准《信息与文献 参考文献著录规则》，规定了各类文献（期刊论文、书籍、学位论文等）在参考文献列表中的显示格式，包括作者名顺序、标点使用、字段排列方式等。LaTeX 中通常用`gbt7714`宏包来自动实现这一格式要求。

♣

GB/T 7714 对 BibTeX 的要求

GB/T 7714 规定了参考文献在最终输出中的显示格式，比如作者名怎么显示、期刊名后面跟什么标点、年份放在哪个位置。如果你用 LaTeX 写论文，通常会用一个叫`gbt7714`的宏包来处理这些格式要求，配置好之后 LaTeX 编译时就会按 GB/T 7714 的规则来渲染参考文献列表。

但宏包只管渲染，不管 BibTeX 源文件本身的质量。如果你的.bib 文件里作者名写法混乱、字段缺失、中英文文献没有区分，宏包解析的时候就可能出错。比如，`gbt7714`宏包对中文文献和英文文献有不同的排版规则：中文文献的作者名直接显示汉字，英文文献的作者名显示”Last F M” 格式。宏包靠`language`字段来判断一条引用是中文还是英文。如果你的中文文献没有加`language = {chinese}`字段，宏包会把它当成英文文献处理，渲染出来的作者名就变成了拼音缩写，看起来很奇怪。

所以，让参考文献列表符合 GB/T 7714，有两件事要做：一是安装和配置宏包（这一步让 Claude Code 处理就行），二是确保 BibTeX 源文件里的数据干净、完整、中英文区分明确。第二件事比第一件难得多，因为它涉及逐条检查和修正。

中文文献的特殊要求

我的 156 条引用里，中文文献约占三分之一，大约 50 条。这些中文文献在 BibTeX 里需要注意几件事。

第一是 language 字段。前面提过，`gbt7714`宏包靠这个字段判断语言。中文文献必须加`language = {chinese}`，否则会被当作英文处理。我的 50 条中文文献里有 9 条缺少这个字段，上一节的格式扫描已经找出来了。

第二是作者名的写法。中文作者名不需要用逗号分隔姓和名（因为中文姓名本身就是连在一起的），直接写”张伟” 就行。但如果你从知网导出的 BibTeX 是拼音格式（”Zhang Wei”），就需要改回中文。不改也不会报错，但参考文献列表里中文文献的作者显示成拼音，和其他中文文献混在一起看着不协调。

第三是标题的语言。有些中文文献在知网的 BibTeX 导出里，标题是英文翻译而不是中文原标题。GB/T 7714 要求中文文献用中文标题。你需要去核实每条中文文献的 title 字段是不是中文原标题，如果是英文翻译，要换回来。

第四是期刊名。中文期刊名要用全称，不用缩写（英文期刊有缩写习惯，中文期刊没有）。但有些从数据库导出的条目，中文期刊名被截断或者用了不规范的简称，需要手动核实。

这四件事，每条中文文献都要检查一遍。50 条中文文献，手动检查的话需要逐条打开知网确认原始信息，至少一两个小时。

批量修正的操作流程

我用 Claude Code 分两步做了批量修正。

第一步，处理 language 字段缺失的问题。这一步比较简单，让它扫描所有 BibTeX 条目，如果标题或期刊名

© 2026 晨瀚宇 未经授权禁止转载

包含中文字符，就自动补上`language = {chinese}`字段：

```
读取 references.bib，找出所有标题或期刊名中包含中文字符、
但没有 language 字段的条目。
给这些条目添加 language = {chinese}。
列出修改了哪些条目。
```

它找出了 9 条，逐一添加了 language 字段。这一步没有误判，因为判断依据很明确：有中文字符的就是中文文献。

第二步，统一作者名格式和期刊名格式。这一步稍微复杂，因为需要分别处理中文和英文条目：

```
读取 references.bib，按以下规则统一格式：

英文文献：
- 作者名统一为 "Last, First and Last, First" 格式
- 期刊名统一使用全称（如果是缩写，展开为全称）
- 页码统一用双连字符 --

中文文献（有 language = {chinese} 的条目）：
- 作者名统一用中文原名
- 如果作者名是拼音格式，标记出来让我手动确认
- 期刊名统一用中文全称

修改完之后，把修改前后的对比写入 format_changes.md。
不确定的地方标注"需要手动确认"，不要猜测。
```

这里有一个关键细节：「不确定的地方标注’需要手动确认’，不要猜测」。中文作者名从拼音恢复成中文的时候，有可能还原错误（”Zhang Wei” 可能是张伟、张维、张威），所以我让它标记出来而不是直接替换。它最终标记了 7 条需要我手动确认的中文作者名，我逐一查了知网原始记录，确认了正确的中文姓名。

英文文献那边，作者名格式统一改了 23 条，期刊名从缩写展开为全称改了 14 条，页码连字符修正了 8 条。这些修改比较机械，它处理得很准确，我抽查了 5 条，都没有问题。

修正前后的对比

下面这张表展示了几个典型条目修正前后的差异：

表 7.2: BibTeX 条目格式修正前后的对比

字段	修正前	修正后
author	W. Zhang, M. Li	Zhang, Wei and Li, Ming
author	Zhang Wei（知网拼音）	张伟，李明（中文原名）
journal	J. Econometrics	Journal of Econometrics
journal	计量经济	计量经济学报（全称）
pages	134-158	134–158
language	（缺失）	chinese

全部修正完成后，我重新编译了一次 LaTeX，检查参考文献列表的渲染效果。中文文献的作者名显示为中文原名，英文文献的作者名格式统一，期刊名全部使用全称，页码格式一致。比起清理之前的版本，整洁了很多。

© 2026 晨瀚宇 未经授权禁止转载

gbt7714 宏包的配置

修正 BibTeX 之后，要让参考文献列表按 GB/T 7714 格式渲染，需要配置gbt7714宏包。这一步不用自己动手——告诉 Claude Code 你需要的引用样式（上标编号还是作者-年份格式），它会按照你学校的要求在导言区配好。如果用 Overleaf，编译顺序也是自动处理的，不用操心。

有一次我编译之后发现参考文献列表里有几条中文文献的作者名还是拼音。排查了半天，发现是那几条 BibTeX 里language字段的值写成了zh而不是chinese，宏包没能正确识别语言类型。改成language = {chinese}之后就正常了。这种问题很难靠肉眼发现，因为差异太细微了。后来我让 Claude Code 多做了一步检查：扫描所有 BibTeX 条目的 language 字段值，确认中文文献统一写的是chinese。

一些学校的额外要求

GB/T 7714 是国家标准，但各学校在此基础上可能有自己的细化要求。我的学校要求参考文献列表里，中文文献排在英文文献前面；期刊论文的 DOI 字段必须显示；学位论文条目必须标注授予单位所在的城市。这些额外要求，宏包不一定直接支持，但可以让 Claude Code 在 BibTeX 层面做调整。比如，学校要求显示 DOI，我让它检查所有条目是否都有 doi 字段，缺失的标记出来让我手动补充。学校要求学位论文标注城市，我让它给所有@phdthesis和@mastersthesis条目添加 address 字段。

这些额外要求每个学校不同，你提交之前最好对照学校的格式模板仔细看一遍参考文献部分的格式说明。通常在”学位论文格式要求”文件的最后几页，会有参考文献格式的具体样例。照着样例一条条对，比你去读 GB/T 7714 原文有用得多。

处理完 GB/T 7714 的格式要求，BibTeX 文件的清理工作就基本完成了。接下来走一遍完整的操作流程，从头到尾演示怎么用/citation-management做一次完整的引用格式核查。

7.4 实操：用 /citation-management 核查引用

这一节走一遍完整的引用格式核查流程，从打开 Claude Code 开始，到生成一份干净的 BibTeX 文件结束。假设你的论文已经写完，参考文献列表里有几十到上百条引用，来源混杂，格式不统一，你需要在提交前做一次集中清理。

第一步：备份原始文件

在做任何修改之前，先备份（备份命令见第一章，备份策略详见第八章）。

第二步：全局格式扫描

启动 Claude Code，进到论文文件夹，运行格式检查：

```
/citation-management 读取 references.bib，做以下格式检查：

1. 列出缺少必填字段的条目（按BibTeX类型判断必填字段：
   @article 需要 author, title, journal, year;
   @inproceedings 需要 author, title, booktitle, year;
   @book 需要 author, title, publisher, year;
   @phdthesis 需要 author, title, school, year）
2. 检查作者名格式是否统一（英文用 Last, First；中文用原名）
3. 检查期刊名是否用全称，找出所有缩写
4. 查找重复引用（标题相似度超过80%的条目对）
5. 检查中文文献是否有 language = {chinese} 字段
```


© 2026 晨瀚宇 未经授权禁止转载

```
6. 检查所有字段名是否为小写
7. 检查页码是否统一用双连字符 --

把问题按类别分组，写入 format_issues.md。
每条问题写清楚：条目的 citation key、问题类型、当前值。
不要自动修改任何内容。
```

为什么指令写这么详细？因为”检查格式”太模糊了。不告诉它具体检查哪些项目、每个项目的标准是什么，它会按自己的理解来做，可能漏掉你关心的项目（比如中文文献的 language 字段），也可能多检查一些你不在意的项目。把检查项和标准列清楚，出来的结果才是你想要的。

为什么加”不要自动修改”？因为格式问题的修复有时需要你的判断。比如一条期刊名缩写，它展开成全称的时候可能展开错了（期刊名缩写不是完全规则化的，同一个缩写可能对应多个期刊）。让它先列问题，你看完确认，再决定怎么改。

第三步：审阅问题列表

Claude Code 生成的 format_issues.md 大致长这样：

```
## 缺少必填字段（6条）
- wang2019panel: 缺少 year
- liu2020spatial: 缺少 year
- chen2018policy: 类型为 @article 但缺少 journal
  （建议改为 @inproceedings，会议名填入 booktitle）
...

## 作者名格式不统一（23条）
- smith2020effects: 当前 "J. Smith, K. Lee",
  应改为 "Smith, John and Lee, Kevin"
- zhang2019study: 当前 "Zhang Wei"（拼音），
  需手动确认中文原名
...

## 期刊名缩写（14条）
- jones2021causal: 当前 "J. Econometrics",
  建议改为 "Journal of Econometrics"
...

## 重复引用（3组）
- wang2020spatial 和 CNKI:SUN:JLXB.0.2020-03-005
  标题相似度 92%，疑似同一篇文献
...

## 缺少 language 字段的中文文献（9条）
- li2021assessment: 标题含中文字符，无 language 字段
...
```


© 2026 晨瀚宇 未经授权禁止转载

拿到这份列表后，我的做法是先通读一遍，标记哪些可以让 Claude Code 自动修复（比如页码连字符、期刊名展开这类机械性修改），哪些需要我手动处理（比如中文作者名确认、重复引用的取舍）。

第四步：分批修复

机械性修改，一次性让 Claude Code 执行：

```
读取 references.bib，做以下修改：

1. 把 format_issues.md 里列出的14条期刊名缩写展开为全称
2. 统一所有英文文献的作者名为 "Last, First" 格式
3. 统一所有页码为双连字符 --
4. 给9条中文文献添加 language = {chinese}
5. 把所有字段名转为小写

修改完成后，把修改前后的逐条对比写入 format_changes.md。
```

这步做完，再处理需要手动判断的部分。重复引用的 3 组，逐组打开原始文献确认是不是同一篇，确认后删除多余条目，替换正文中的 citation key。缺少 year 字段的 4 条，逐条去知网或 Google Scholar 查原始发表信息，补上正确年份。中文作者名需要确认的 7 条，去知网查原始记录。

手动处理的部分，每条花一两分钟。14 条手动任务，加起来半小时左右。如果没有前面的自动扫描帮我找出问题，光是发现这 14 条问题就要花好几个小时。

第五步：正文引用交叉检查

BibTeX 文件清理完毕后，做一次正文引用和.bib 文件的交叉检查，确保两边匹配：

```
扫描所有 .tex 文件里的 \cite{} 命令，提取所有 citation key。
读取 references.bib，提取所有条目的 citation key。
对比两个列表：
1. 正文引用了但 .bib 里不存在的 key（拼写错误或遗漏）
2. .bib 里存在但正文从未引用的条目（多余条目）
把结果写入 cross_check.md。
```

这一步在上一节提过，我的情况是 2 个拼写错误和 8 条多余引用。拼写错误必须修复（否则编译时会显示问号），多余引用建议删除（保持.bib 文件整洁）。

第六步：编译验证

所有修改完成后，重新编译一次 LaTeX（Overleaf 直接点 Recompile，本地编译让 Claude Code 帮你跑就行），打开 PDF 翻到参考文献那几页，检查几件事：有没有显示问号的引用（说明 citation key 匹配不上），中文文献的作者名是否显示为中文，英文文献的作者名格式是否统一，期刊名是否都是全称。

我在最终编译验证的时候发现了最后一个问题：有一条中文文献的标题在.bib 文件里是英文翻译（知网导出的时候选了英文 BibTeX 格式），渲染出来一条中文作者名跟一个英文标题，看起来很违和。回去把标题改成中文原标题，重新编译，问题解决了。

第七步：存档对比报告

所有修改完成、编译验证通过之后，把修改记录归档：

© 2026 晨瀚宇 未经授权禁止转载

把 format_issues.md、format_changes.md、cross_check.md
合并成一份 citation_cleanup_report.md，
按时间顺序记录整个清理过程。

这份报告有两个用处。第一，如果后续发现参考文献还有问题（比如答辩时评委指出某条引用有误），你可以回溯当时做了哪些修改。第二，如果你的同学也要做类似的 BibTeX 清理，你可以把这份报告和操作流程直接分享给他。

整个流程的时间成本

把上面七步的时间加起来：备份 1 分钟，全局扫描 2 分钟，审阅问题列表 10 分钟，分批自动修复 5 分钟，手动处理 14 条约 30 分钟，交叉检查 1 分钟，编译验证 5 分钟，存档 2 分钟。总共约 1 小时。

156 条引用的完整格式清理，1 小时。如果没有工具辅助，手动做同样的事情，保守估计需要半天到一天。而且手动做的时候，很难保证每条都检查到了。你可能从头看到尾，眼睛扫过了 156 条，但到第 100 条的时候注意力已经下降了，后面 56 条的检查质量和前面 100 条不在一个水平上。自动化扫描不会有这个问题，第 1 条和第 156 条用同样的标准检查。

表 7.3: 引用格式清理的时间对比（156 条引用）

手动清理	用 /citation-management 清理
逐条打开文献核实格式，每条 3-5 分钟	自动扫描全部条目，2 分钟出报告
凭记忆判断哪些期刊名是缩写	自动识别并列出所有缩写
肉眼找重复引用，容易遗漏	按标题相似度自动匹配重复
手动比对正文\cite{}和.bib 文件	自动交叉检查，1 分钟出结果
总计约 4-8 小时	总计约 1 小时

几个注意事项

做格式清理的时候，有几件事我建议你留意。

第一，不要跳过手动确认的步骤。自动修复很快，但有些修改需要你的判断。期刊名缩写展开的时候，Claude Code 偶尔会展开错（比如”JDE” 可以是 Journal of Development Economics 也可以是 Journal of Differential Equations），你得确认它展开的是对的。我在 14 条期刊名修改里，有 1 条被展开成了错误的期刊，幸好我看了 format_changes.md 的对比，及时改正了。

第二，每做完一批修改就编译一次。不要等所有修改都做完再编译。我有一次把重复引用删掉之后没有立刻编译检查，后面又做了一批作者名修改，编译的时候发现有 3 个\cite{}引用变成了问号。回头排查，发现是删重复引用的时候，有一个 citation key 替换漏了。如果在删完重复引用之后立刻编译检查了，当时就能发现这个问题，不至于后面混在一堆修改里难以定位。

第三，清理完成后的.bib 文件，用一个固定的格式保存。我的做法是让 Claude Code 最后做一次格式化：所有条目按 citation key 字母排序，字段顺序统一（author 在前，title 其次，journal/booktitle 再次，year、volume、number、pages、doi、language），缩进用两个空格。这样以后再添加新的引用条目时，格式一目了然，容易保持一致。

到这里，引用和参考文献的清理工作就讲完了。下一章讲另一个提交前的重要环节：格式排版和文件管理，包括 Word 文档操作的风险、备份策略，以及从 Word 迁移到 LaTeX 的完整过程。

© 2026 晨瀚宇 未经授权禁止转载

第八章 格式排版与文件管理

内容提要

- Word 文档（.docx）底层是 ZIP 压缩包内的 XML，Claude Code 编辑时存在格式损坏风险
- 每次修改 Word 文件前必须备份，「不备份就不动手」是防止格式崩溃的底线规则
- Git 版本管理配合.tex 纯文本格式，可逐行追踪每次改动，远优于 Word 的二进制文件管理
- 从 Word 迁移到 LaTeX 可用 pandoc 和/markitdown完成，两天内即可完成全文转换

我用 Claude Code 写论文的过程，大致分成三个阶段。前两个月全在 Word 里操作，用/docx这个 skill 改章节、调格式、统一术语。中间有一段时间频繁碰到文件损坏的问题，补救的办法越来越费时间。最后一个月，我把整篇论文从 Word 转到了 LaTeX，在 Overleaf 上定稿。回头看，在 Word 上花的 66 次会话里，有相当一部分时间其实是在处理格式损坏和修复 XML，真正用在改内容上的会话比我预想的少。

这一章把这个过程里踩过的坑和最终的解决办法讲清楚。如果你现在还在用 Word 写论文，前两节的备份方法请务必现在就用上；如果你已经在考虑转 LaTeX，后两节会帮你省掉很多摸索的时间。

8.1 Word 文档操作的风险

第五章的实操部分提到过，用 Claude Code 修改 Word 文档有一个五步流程：备份、读取、修改、检查、验证。那一节讲的是流程本身。这一节要讲的是：为什么这个流程里「备份」和「验证」两步绝对不能省，以及 Word 文档底层到底有什么问题，导致 Claude Code 操作它的时候容易出错。

一次覆盖，前功尽弃

我第一次在 Word 文档上栽跟头，是在改第五章实证分析的时候。那天我让 Claude Code 修改一节的内容，具体任务是把稳健性检验那部分的表述理顺，同时统一几个术语。任务本身不复杂，Claude Code 很快就改完了。我扫了一眼对比，觉得没问题，就继续做下一个任务了。

问题出在第二天。我打开那个文件，发现前一天改的那一节没问题，但同一个文件里另外两节的格式全乱了。段落间距变了，有几个表格的边框消失了，页眉的章节标题也不对了。更糟糕的是，我前一天没有备份。Claude Code 在修改的时候是直接写回原文件的，它改完的版本直接覆盖了我之前的版本。我没有任何办法回到修改之前的状态。

那天晚上我花了三个多小时，一段一段地把格式手动修回去。有些格式我已经记不清原来是什么样了，只能凭印象调。修完之后心里很不踏实，因为我不确定有没有漏掉什么。

注 任何涉及 Word 文档的操作，第一步必须是备份。不是「最好备份一下」，是「不备份就不动手」。Claude Code 修改 Word 文件是直接覆盖原文件的，一旦出问题没有备份就无法恢复，只能靠记忆手动修回去。

备份的命令很简单：

```
cp thesis.docx thesis.docx.bak.$(date +%s)
```

$(date +%s)会生成一个时间戳，比如thesis.docx.bak.1711843200。每次备份的文件名都不一样，不会互相覆盖。你也可以用更可读的日期格式：

```
cp ch5_analysis.docx "ch5_analysis_backup_$(date␣+%Y%m%d_%H%M%S).docx"
```


© 2026 晨瀚宇 未经授权禁止转载

这样备份文件名会是`ch5_analysis_backup_20260315_143022.docx`，一看就知道是哪天哪个时间点的版本。

原则很简单：永远不覆盖原文件，只创建新文件或备份副本。如果你一天要改好几次，就备份好几次。磁盘空间不值钱，你的工作成果值钱。

定义 8.1 (.docx)

.docx 是 Microsoft Word 的文件格式，本质上是一个 ZIP 压缩包，内含若干 XML 文件来描述文字、格式和表格。这种二进制 +XML 结构使得程序化编辑时容易引发 XML 解析错误或格式损坏，与 LaTeX 的纯文本格式相比稳定性差很多。

Word 文件的底层结构

要理解为什么 Claude Code 改 Word 文件容易出问题，需要知道.docx 文件的底层结构。.docx 不是一个纯文本文件，它本质上是一个压缩包（ZIP 格式），里面包含了一堆 XML 文件。你在 Word 里看到的每一段文字、每一个格式设置、每一个表格，在底层都是用 XML 标签来描述的。

Claude Code 修改 Word 文件的时候，实际上是在操作这些 XML。它把.docx 解压，找到对应的 XML 文件，修改里面的内容，然后重新打包。这个过程听起来很直接，但中文文档的 XML 有几个特别容易出问题的地方。

第一个问题是中文标点。中文的全角标点在底层的编码方式和英文标点不同，编辑时偶尔会把全角替换成半角，或者转义出错，导致排版走样。

第二个问题是特殊字符的转义。Word 的 XML 对一些特殊字符有严格的编码规则，如果论文里有数学表达式或不等式，编辑时可能破坏这些编码，导致文件根本打不开。

第三个问题最隐蔽：Word 会把一段看起来连续的文字拆成多个碎片存储。比如同一句话里，如果某几个字加了粗、某几个字字体不同，Word 就会把它们分成不同的片段。做查找替换时，搜索的是完整的文字，但操作的是这些碎片。如果要替换的内容刚好跨了多个碎片，替换就可能失败或格式错位。

我有一次让 Claude Code 做术语替换（第一章提到的那类任务），大部分地方都改对了，但有两处改完之后字体变成了宋体，而论文其他部分用的是仿宋。原因就是那两处的文字被拆成了不同格式的碎片，替换时格式信息丢了。

损坏的症状和应对

Word 文档被 Claude Code 改坏之后，症状有几种。最轻的是格式走样：段落间距变了、字体变了、表格边框消失了。这种情况文件还能打开，肉眼检查能发现问题，用备份恢复就行。

中等程度的损坏是：文件能打开，但 Word 提示「文件存在问题，是否尝试修复」。修复之后有时候能恢复大部分内容，有时候表格和图片会丢失。

最严重的是文件根本打不开。Word 弹出一个错误提示，说文件格式无效或已损坏。这种情况通常是 XML 结构被破坏了（比如标签没有正确闭合、实体编码出错），XML 解析器无法读取整个文件。碰到这种情况，唯一的办法就是用备份。如果没有备份，只能尝试一些数据恢复工具，但成功率不高。

我在写论文的过程中一共碰到过三次文件打不开的情况。前两次有备份，恢复之后重新改就行了，每次多花半小时左右。第三次就是前面讲的那次没备份的，花了三个多小时手动修格式。从那次之后，我每次操作 Word 之前都会先跑一遍备份命令，没有例外。

减少损坏风险的做法

如果你的论文必须用 Word 格式（比如学校模板要求），有几个做法可以降低出问题的概率。

每改一小块就验证一次。不要一口气让 Claude Code 改完整章再去检查，每改一节甚至一段，就用 Word 打

© 2026 晨瀚宇 未经授权禁止转载

表 8.1: Word 文档三种损坏程度的症状与应对

损坏程度	症状	应对方法
轻度	格式走样（间距、字体、边框变化），文件可正常打开	用备份恢复，或手动调格式
中度	Word 提示「文件存在问题，是否修复」，部分表格或图片丢失	先尝试 Word 自动修复，不行则用备份
重度	文件无法打开，提示格式无效或已损坏	只能用备份恢复，没有备份基本无法挽回

开看看文件是否正常。发现问题时影响范围小，回退也容易。

避免一次性大改。如果你有十处要修改的地方，分成十次让它改，每次改一处。每次改完验证文件完好之后再进行下一处。我知道这很慢，但比改完十处之后发现文件打不开、全部作废要好得多。

复杂格式的文档要格外小心。如果你的章节里有大量表格、嵌入图片、数学公式或特殊格式（如页眉页脚、脚注尾注），被改坏的概率会更高。这类文件我后来都尽量让 Claude Code 只做纯文字修改，不动格式相关的部分。

我在 Word 上总共用了 66 次/docx会话。如果只算真正在修改论文内容的，大概有 40 多次。剩下的二十几次，要么是在处理格式问题，要么是在验证文件有没有被改坏，要么是在从备份恢复之后重新操作。这个比例让我逐渐意识到，Word 文件格式本身就是一个不稳定因素。到后来碰壁的次数多了，我开始认真考虑换一种格式。

下一节先讲怎么做好备份，把风险兜住。之后再讲我最终为什么转了 LaTeX，以及转换的过程。

8.2 备份策略

上一节反复提到备份，这一节把备份的具体做法系统地讲一遍。不只是「改 Word 之前备份」这一个场景，而是整个论文写作过程中的版本管理思路。

手动备份命令

最基础的备份方式就是在操作文件之前，用cp命令复制一份。前面已经给过两种命名方式：

```
# 时间戳命名（不会重复，适合高频备份）
cp thesis.docx thesis.docx.bak.$(date +%s)

# 可读日期命名（一眼看出时间）
cp thesis.docx "thesis_backup_$(date␣+%Y%m%d_%H%M%S).docx"
```

我个人更常用第二种，因为备份多了之后需要找某个时间点的版本，可读的日期比一串数字方便得多。时间戳命名的好处是不可能重名，一秒钟内你不会备份两次。

如果你一天要改好几个文件，可以建一个备份文件夹，把当天的所有备份集中放在一起：

```
mkdir -p backups/$(date +%Y%m%d)
cp ch3_method.docx "backups/$(date␣+%Y%m%d)/ch3_method_$(date␣+%H%M%S).docx"
cp ch5_analysis.docx "backups/$(date␣+%Y%m%d)/ch5_analysis_$(date␣+%H%M%S).docx"
```

mkdir -p的意思是「如果文件夹已存在就不创建，不存在就创建」，加-p不会报错。这样你每天的备份都在一个以日期命名的文件夹里，清晰明了。

让 Claude Code 自动备份

© 2026 晨瀚宇 未经授权禁止转载

手动备份的问题是你可能忘记。改着改着投入进去了，忘了先跑备份命令，等出了问题才想起来。解决办法是在给 Claude Code 下指令的时候，把备份写进任务里：

```
修改 ch5_analysis.docx 之前，先备份到 backups/ 文件夹，
文件名带上当前时间。备份完成后再开始修改。
```

更彻底的做法是写进 CLAUDE.md 里，让它每次启动都能读到：

```
## 文件操作规则
- 修改任何 .docx 文件之前，先创建备份副本到 backups/ 文件夹
- 备份文件名格式：原文件名_YYYYMMDD_HHMMSS.docx
- 备份完成后才能开始修改
- 修改完成后，提醒我用 Word 打开验证
```

我把这段话加进 CLAUDE.md 之后，每次让它改 Word 文件，它都会先自动跑备份命令。省心很多。

后面第十章还会讲到 Hooks（自动化触发器），那是一种更可靠的方式：设置一个 Hook，让 Claude Code 在每次修改.docx 文件之前自动执行备份脚本，连提醒都不需要。但在你学会 Hook 之前，写进 CLAUDE.md 已经足够用了。

用 Git 做版本管理

如果你的论文文件比较多（正文四五章、参考文献、附录、图表文件夹），单靠cp命令备份会越来越乱。十几个文件每个都有好几个备份版本，放在一起很难找。这时候可以用 Git 来管理版本。

定义 8.2 (Git 与版本管理)

Git 是一种分布式版本控制系统，记录文件在每次提交（commit）时的完整快照，可随时回退到任意历史版本。对于纯文本格式（如.tex 文件），Git 还能逐行显示两个版本之间的具体差异（git diff），是论文写作中管理修改历史的最可靠工具。

♣

Git 是程序员用来做代码版本管理的工具，但它管论文也很好用。它的核心功能就是：每次你提交（commit）一个版本，它就记录下来所有文件在那一刻的状态。你可以随时回到之前的任何一个版本。

用 Git 管理论文的核心流程就三步：在论文文件夹里初始化（git init），每次改完提交一次（git add加git commit），需要回退时查历史版本恢复。最常用的命令：

```
# 初始化 + 第一次提交
cd ~/thesis && git init && git add . && git commit -m "initial␣draft"

# 每次改完一章，提交一次
git add ch5_analysis.docx
git commit -m "ch5:␣rewrite␣robustness␣section"

# 改坏了？查历史，恢复到某个版本
git log --oneline
git checkout abc1234 -- ch5_analysis.docx
```

有一点要注意：.docx 文件是二进制格式，Git 能保存每个版本，但看不到两个版本之间具体改了哪些字。后面转到 LaTeX 之后，.tex 是纯文本，Git 就能逐行显示差异，用起来方便得多。

© 2026 晨瀚宇 未经授权禁止转载

备份的频率

多久备份一次？我的经验是：每次让 Claude Code 修改文件之前备份一次，每天结束工作的时候用 Git 提交一次。前者保证单次操作出问题能回退，后者保证你能找到任何一天的工作状态。

如果你那天要密集修改（比如集中处理导师的批量反馈），可以更频繁一些：每做完一章就 Git 提交一次，提交信息写清楚做了什么。这样即使后面几章改出了问题，前面已经提交的章节是安全的。

提交信息不需要多详细，`git commit -m "day1: ch3 all 8 revisions done"`这样写就够了，关键是让你过几天还能看懂当时做了什么。

云端备份

本地备份只能防操作失误，防不了硬盘坏掉或者电脑丢了。如果你用 Git，可以把仓库推到 GitHub 或 Gitee 上（学校内网有些有自建的 GitLab），相当于在云端多了一份。

如果你不想用 Git，最简单的云端备份就是用网盘。把论文文件夹设为自动同步，每次改完它自动上传。坚果云、百度网盘、OneDrive 都可以。重点是：你的论文至少要有两个地方存着，一个在电脑本地，一个在别的地方。

我有一个同学论文定稿那天，电脑蓝屏了，硬盘读不出来。如果不是之前开了 OneDrive 同步，她四万字的论文就全没了。这种事不常见，但发生一次就够你受的。

到这里，备份的方法讲完了。备份解决的是「出了问题能回退」，但它不能解决 Word 文件本身容易被改坏这个根本问题。如果你发现自己花在修复格式上的时间越来越多，值得考虑换一种格式。下一节讲我最终做了这个决定的过程。

8.3 从 Word 到 LaTeX 的迁移

先说清楚：我不是一开始就打算用 LaTeX 写论文的。和大多数文科、社科专业的学生一样，我从开题到初稿，全程都在用 Word。学校提供的论文模板是 Word 格式，导师批注也是用 Word 的修订模式，同学之间互相传阅也是 Word。LaTeX 对我来说是一个「听说过但从来没用过」的东西，总觉得那是理工科和计算机专业的人才需要学的。

让我开始认真考虑转 LaTeX，原因很具体：被 Word 的格式问题折腾得受不了了。

触发迁移的导火索

压垮骆驼的最后一根稻草是一次表格修改。第五章有一个回归结果表格，导师说要加一行空间溢出效应的系数。我让 Claude Code 在表格里加了一行数据。改完之后 Word 打开一看，表格的后半部分全错位了：列对不齐，有几个单元格的边框消失了，表格下方的注释跑到了下一页。

我用备份恢复，重新改，换了一种方式（让它先把表格导出来改，改完再放回去），结果格式还是出了问题，只是错的地方不一样。前前后后试了三次，每次都有不同的格式问题。最后我放弃让 Claude Code 改这个表格，自己手动在 Word 里加了那一行。一个本来五分钟就能做完的事，花了一个多小时。

那天晚上我翻了一下自己的使用记录，发现最近两周的`/docx`会话里，将近一半的时间都在处理格式修复。我的论文有将近 100 页，里面有 20 多个表格、十几个图、大量的数学符号和脚注。文档越复杂，Claude Code 改起来出格式问题的概率就越高。这个趋势如果继续下去，我花在修格式上的时间会越来越多，到最后可能比手动改还慢。

我开始认真搜索 LaTeX 的相关信息。之前对 LaTeX 的印象是学习门槛高、语法繁琐，但几个优点打动了我。

LaTeX 的关键优势

© 2026 晨瀚宇 未经授权禁止转载

LaTeX 文件是纯文本的.tex格式。这意味着 Claude Code 在修改它的时候，操作的就是文本本身，不需要去处理 XML、不需要担心二进制格式损坏。你让它改一段话，它就改那段话的文字，改完保存，不会影响其他部分的格式。这个特性从根本上消除了 Word 的 XML 损坏问题。

表格在 LaTeX 里也是文本。你看到的表格在源码里就是一行行的文字和&符号（分隔列）、\\符号（换行）。Claude Code 加一行数据，就是在源码里插入一行文本，格式不可能错位，因为格式是由表头的列定义决定的，跟内容是分开的。

还有一点：LaTeX 和 Git 配合得很好。.tex 文件是纯文本，Git 可以逐行追踪每次改了什么。上一节提到 Word 文件在 Git 里只能看到「文件变了」，看不到具体变了哪些字。换成 LaTeX 之后，`git diff`能精确显示这次修改改了哪几行、改了什么内容。做版本管理方便了很多。

表 8.2: Word 与 LaTeX 在 Claude Code 操作下的对比

对比维度	Word (.docx)	LaTeX (.tex)
文件格式	二进制压缩包（ZIP+XML）	纯文本
Claude Code 编辑方式	解压 → 改 XML→ 重新打包	直接修改文本
格式损坏风险	高（XML 实体、run 分割、编码问题）	低（语法错误会报错，不会静默损坏）
表格修改	容易错位，格式与内容耦合	稳定，格式与内容分离
Git 版本对比	只知道文件变了，看不到内容差异	逐行显示具体改了什么
出错恢复	依赖备份文件	可用 Git 精确回退到某一行

做出决定

促使我下决心的另一个因素是：我发现 Claude Code 对 LaTeX 的支持比对 Word 好。这不难理解，因为 LaTeX 源码就是文本加标记，和代码很像。Claude Code 生成的 LaTeX 表格、公式、引用标注，语法正确率很高。而且它能直接读.tex文件，不需要额外的 skill 来解析格式。

当然，迁移也有风险。我的论文已经写了将近 80%，从 Word 转到 LaTeX 意味着所有的格式设置（字体、行距、页边距、页眉页脚）都要重新配置。学校的论文模板是 Word 格式的，LaTeX 模板需要自己找或者自己调。转换过程中如果内容出了问题，损失会很大。

我花了一个晚上在网上搜学校的 LaTeX 模板。运气不错，有学长在 GitHub 上传了一份，虽然不是最新版本的格式要求，但基本框架是可用的。我把模板下载下来，在 Overleaf 上试编译了一下，能跑通。这给了我信心。

转换过程

实际的转换用了大概两天时间。核心工具是 pandoc 和/markitdown这两个，下一节会详细讲操作步骤。

定义 8.3 (pandoc)

pandoc 是一款通用的文档格式转换命令行工具，支持 Word（.docx）、LaTeX（.tex）、Markdown、HTML 等数十种格式之间的互相转换。在论文迁移场景中，常用命令为`pandoc input.docx -o output.tex --wrap=none`，将 Word 文档转为 LaTeX 源码作为后续清理的基础。

这里先说转换过程中的几个关键环节。

第一步是把 Word 文件转成 LaTeX 源码。pandoc 可以做这件事，但转出来的源码质量不高：会有很多冗余的格式命令、不规范的表格写法、图片路径需要手动调整。/markitdown可以先把 Word 转成 Markdown，作为中间格式来审查内容是否完整。

第二步是把转出来的源码填进学校的 LaTeX 模板里。模板定义了字体、行距、页边距、目录格式这些，我只需要把正文内容粘贴到模板对应的位置，不需要自己从零配置格式。

© 2026 晨瀚宇 未经授权禁止转载

第三步是逐章检查和调整。转换工具处理不好的地方主要有几类：表格格式需要手动调、图片路径需要更新、数学公式的 LaTeX 写法偶尔需要修正、参考文献的引用标签需要和.bib 文件对应上。这些零碎的调整花了大部分时间。

第四步是在 Overleaf 上编译测试。Overleaf 是一个在线的 LaTeX 编辑器，不需要在本地安装任何东西，上传源码就能编译出 PDF。我把所有文件上传上去，点编译，第一次就成功了。PDF 出来的排版效果比 Word 好看不少（学术论文里 LaTeX 排版确实更专业），表格、公式、引用都是对的。

整个过程比我预想的顺利。我原来以为至少要一个星期，结果两天就搞定了。其中大部分时间花在了调表格格式和检查引用上，纯粹的转换步骤本身很快。

转换后的体验

转到 LaTeX 之后，我用 Claude Code 改论文的效率明显提高了。最直接的变化是：再也没碰到过文件损坏的问题。Claude Code 改完之后，最坏的情况是编译报错（比如某个 LaTeX 命令拼错了），但报错信息会告诉你具体哪一行有问题，改掉那一行就好了。不会出现 Word 那种「静默损坏」的情况，表面上看着没问题，实际上某个地方的格式已经乱了。

Git 的使用体验也好了很多。每次`git diff`都能看到具体改了哪些行，改了什么内容。导师看完让我改的地方，我改完用`git diff`就能生成一份精确的修改记录，方便自己检查，也方便给导师看「这次改了哪些」。

如果让我重来一遍，我会在论文写到一半的时候就转 LaTeX，不会等到快定稿了才转。早转的好处是后面所有的修改都在 LaTeX 里做，积累的格式问题更少。不过话说回来，如果不是在 Word 上吃了那么多亏，我可能也不会下决心去学一个新工具。有时候确实要痛够了才会改。

8.4 实操：用 /markitdown 转换文档格式

这一节走一遍从 Word 转 LaTeX 的完整操作流程。你需要的工具有两个：`/markitdown`（微软开发的文档格式转换工具）和 pandoc（通用的文档转换命令行工具）。两个都让 Claude Code 帮你装就行。

第一步：安装工具、备份文件

需要两个工具：markitdown（在 Claude Code 里输入`/markitdown`让它帮你装）和 pandoc（macOS 用`brew install pandoc`，其他系统类似）。装好之后，先把所有 Word 文件备份一份到单独的文件夹。

第二步：用 markitdown 预览内容

在正式转 LaTeX 之前，先用 markitdown 把 Word 转成 Markdown，检查一下内容是否完整。Markdown 是一种轻量级的纯文本格式，人眼可读，适合用来做中间检查。

```
markitdown ch5_analysis.docx -o ch5_preview.md
```

这行命令把`ch5_analysis.docx`转成 Markdown 格式，输出到`ch5_preview.md`。打开这个文件看看：正文内容在不在、表格有没有丢、图片的引用有没有保留。

如果有多个章节文件，让 Claude Code 批量转就行。markitdown 对纯文字内容的还原度很高，表格结构大部分情况下是完整的，图片引用会保留路径。数学公式的转换效果取决于 Word 里公式是怎么写的，有些能转出来，有些会变成乱码，转不出来的后面手动补就行。

第三步：用 pandoc 转成 LaTeX

确认 Markdown 预览内容完整之后，用 pandoc 做正式的 LaTeX 转换：

```
pandoc ch5_analysis.docx -o ch5_analysis.tex --wrap=none
```


© 2026 晨瀚宇 未经授权禁止转载

转出来的.tex 文件大部分内容结构是正确的，但有几个地方通常需要手动调整。

第四步：清理和调整

pandoc 转出来的 LaTeX 源码有一些常见问题：自动生成的多余标签、表格格式不符合要求（比如用了 longtable 而不是 tabular，没有三线表格式）、图片路径需要更新、中文字符偶尔乱码。这些都可以让 Claude Code 批量修复，告诉它你的要求就行。手动做一章可能要花半小时到一小时，Claude Code 几分钟就能搞定，而且因为.tex 是纯文本，不会出现 Word 那种改完格式乱掉的问题。

数学公式如果在转换中丢失了，对照原始 Word 文件，把公式截图发给 Claude Code，让它写出对应的 LaTeX 代码。常见的回归方程、模型设定，基本一次就能写对。

第五步：填入模板

清理完的.tex 文件需要放进学校的论文模板里。LaTeX 模板通常有一个主文件（比如`main.tex`），里面用`\input{}`或`\include{}`来引入各章节的文件。你把清理好的各章文件放到指定位置，在主文件里用`\include{}`引用就行。

参考文献需要一个.bib 文件。如果你之前用 Zotero 或其他文献管理工具，可以直接导出 BibTeX 格式。如果你的引用之前是手动写的，需要把每条引用转成 BibTeX 格式。这个工作量取决于你的引用数量，我的论文有一百多条引用，让 Claude Code 转花了大概两个小时（前面章节讲过的并行 Agent 批量核查就是在这个环节用的）。

第六步：上传 Overleaf 编译

所有文件准备好之后，打包成 ZIP 上传到 Overleaf（一个免费的在线 LaTeX 编辑器，不需要在本地装任何东西）。点编译按钮，几秒钟就能看到 PDF。如果编译报错，日志会告诉你具体哪一行有什么问题，改掉就行。这些错误信息都很明确，比 Word 那种「文件已损坏」的提示有用得多。

我第一次在 Overleaf 上编译整篇论文，一次就成功了。PDF 出来之后翻了一遍，正文内容完整，表格格式正确，引用标号自动生成，目录和页码都是对的。那个瞬间心情挺好的，之前在 Word 上花的那些修格式的时间，突然觉得都不用花了。

转到 LaTeX 之后，后续的所有修改都直接在.tex 文件上做。Claude Code 改.tex 文件就是直接编辑文本，没有 XML、没有压缩包、没有格式损坏。快速、稳定、可追溯（用 Git 能看到改了什么），不需要先备份、不需要改完验证文件是否损坏、不需要担心改一处影响其他地方的格式。

关于学习成本

有些同学可能会担心 LaTeX 的学习成本。我的经验是：如果你只是用 LaTeX 来写论文（而不是自己从零开始设计模板），需要学的东西其实不多。核心就是这几个命令：`\section{}`分节、`\textbf{}`加粗、`\cite{}`引用文献、`\ref{}`引用表格和图片编号、`\begin{table}`和`\begin{figure}`画表格和插图。公式的写法稍微复杂一些，但 Claude Code 对 LaTeX 公式的支持很好，你把公式的数学含义告诉它，它帮你写 LaTeX 代码。

我从完全不会 LaTeX 到能独立在 Overleaf 上编辑论文，大概花了三天。第一天跟着一个入门教程走了一遍基本语法，第二天把论文转过去，第三天处理各种细节问题。后面就顺了。LaTeX 的语法是固定的，不像 Word 那样同一个效果有好几种实现方式，记住几个常用命令就够用了。

如果你的论文已经定稿或者快定稿了，转 LaTeX 的性价比不高，不如把备份做好、每次小改小验。但如果你的论文还在写作中期，论文里有大量表格、公式、交叉引用，用 Claude Code 修改 Word 频繁碰到格式问题，那转 LaTeX 是值得认真考虑的。我用`/markitdown`转换文档格式一共开了 8 次会话，整个从 Word 到 LaTeX 的迁移大概两天完成。和后面省下来的修格式的时间比，这个投入很划算。

© 2026 晨瀚宇 未经授权禁止转载

第三部分

进阶能力——解锁更多可能

© 2026 晨瀚宇 未经授权禁止转载

第九章 Skills：给 AI 装上专属工具

内容提要

- Skill 是什么：把可复用的规则体系存成文件，用斜杠命令一键加载，解决每次重复粘贴提示词的问题
- 科研常用现成 Skill：/pdf 读文献、/drawio 画图、/arxiv-database 搜论文、/citation-management 管引用
- 如何从零创建自定义 Skill：以导师批注要求为例，讲结构、写法和逐步积累经验
- 安装与验证流程：在对话里一句话装好 Skill，确认能正常调用

第二章讲过「每次粘贴背景」的问题。写论文时还有一个类似的麻烦：写作规则也要每次粘贴。术语锁定、表述规范、禁止的句式、段落要求，加起来两千多字，每次开工先复制粘贴进去，等它读完确认，才能开始正式写作。

有一天我在翻 Claude Code 的文档时看到一个功能叫 Skill。它的意思是：你可以把一套规则写成一个文件，存在固定的地方，以后每次用斜杠命令调用就行。后来我在社区里找到一个现成的中文学术润色 Skill，叫 `/humanizer-zh`，装上之后每次开新会话，打一个 `/humanizer-zh`，两千字的规则就自动加载了。

这一章讲三件事：Skill 是什么、科研中哪些 Skill 好用、怎么自己写一个。

9.1 什么是 Skill，和普通提示词的区别

上面说的重复粘贴规则的问题，Skill 就是解决方案。

定义 9.1 (Skill)

Skill 是 Claude Code 的可复用指令单元。每个 Skill 是 `~/.claude/skills/` 目录下的一个文件夹，内含 `SKILL.md` 规则文件。用斜杠命令（如 `/humanizer-zh`）调用时，Claude Code 自动读取该文件的全部规则并在当前会话中执行。它的核心价值是持久化、可复用、可共享，解决每次新会话都要重复粘贴提示词的问题。

Skill 解决的就是这个「重复粘贴」的问题。它把一组指令存成文件，放在 `~/.claude/skills/` 目录下。每个 Skill 是一个文件夹，里面有一个 `SKILL.md` 文件，写着这个 Skill 的名称、触发条件和具体指令。你在对话里输入斜杠加名称（比如 `/pdf`），Claude Code 就会把 `SKILL.md` 的内容读进来，按照里面的规则执行。

这和普通提示词有什么区别？区别在三个地方。

第一，持久化。普通提示词是临时的，会话关了就没了。你下次开新会话还得重新说一遍。Skill 是存在文件系统里的，只要不删，永远在。我装了 Claude Code 大半年，`~/.claude/skills/` 文件夹里攒了一百多个 Skill。大部分是社区共享的，我自己用到的大概二三十个，但它们一直在那里，随时能调用。

第二，复杂度。普通提示词一般就是一两句话的指令，比如「用学术语气改写这段话」。Skill 可以包含很长的规则体系。我常用的 `humanizer-zh` 这个 Skill，光是 `SKILL.md` 就有 380 多行，涵盖术语锁定规则、正文写法的 8 条规则、绝对禁止的写法清单、学术规范要求、段落结构要求、逻辑连贯性标准，最后还有一个检查清单。这些内容如果每次手动粘贴，光是等它读完就要好几十秒。用 Skill 调用，一个斜杠命令就全部加载了。

第三，可共享。Skill 就是一个文件夹，你把它拷给别人，别人放到自己的 `~/.claude/skills/` 目录下就能

© 2026 晨瀚宇 未经授权禁止转载

用。我后来发现社区里有大量现成的 Skill，做什么的都有：读 PDF 的、搜 arXiv 论文的、画流程图的、管理参考文献的。你不用自己从零写，找到一个合适的装上就行。

打个比方：普通提示词像是你每次去图书馆都口头告诉管理员「我要借计量经济学的书」，Skill 像是你办了一张借阅卡，上面写好了你的偏好，以后刷卡就行。

表 9.1: 普通提示词与 Skill 的对比

普通提示词	Skill
每次新会话都要手动输入或粘贴	存在文件里，斜杠命令一键调用
通常一两句话，简短指令	可以包含数百行规则、模板、检查清单
会话结束就消失	持久化存储，跨会话可用
分享靠复制粘贴文本	整个文件夹拷给别人即可安装
适合临时、一次性的指令	适合反复使用的规则体系

Skill 的结构很简单。每个 Skill 就是一个文件夹，里面有一个规则文件，写着名称和具体指令。你用斜杠命令调用它，Claude Code 就按照里面的规则执行。

我第一次了解到 Skill 这个功能的时候，想到的是 Word 的宏。你把一组操作录制成宏，以后一键回放。Skill 的逻辑类似，只是它录制下来的东西从鼠标点击换成了一套文字化的规则和指令。

到这里，概念层面的东西讲完了。下面看看科研中哪些现成的 Skill 好用。

9.2 安装现成的科研 Skill

我一开始用 Claude Code 做论文，Skill 一个都没装。所有功能都是靠手动输入指令实现的。后来发现有些操作我一周要重复好几次，去社区搜了搜，才知道这些操作早就有人做成了现成的 Skill。装了之后效率差别很明显，下面按我实际使用频率讲几个。

/humanizer-zh：中文学术润色，我用得最多的 Skill。这个 Skill 内置了一整套学术中文写作的规则：术语保护（防止 Claude Code 把你的专业术语擅自替换）、AI 味消除（禁止「不是……而是……」「值得注意的是」等典型 AI 句式）、句式控制（句子不超过 25 字、一段一个主题）、格式统一（三线表、引用格式等）。装上之后，每次开新会话打一个 `/humanizer-zh`，两千多字的规则一秒加载，不用每次手动粘贴。

我之前的做法是每次开会话都复制粘贴一大段写作规则，规则越攒越多，从几百字涨到两千多字。每次粘完还要确认它读懂了，有时候它会漏掉某条规则。装了这个 Skill 之后，这些问题全没了。它还自带一份检查清单，润色完之后 Claude Code 会自查一遍：术语有没有被改动、有没有 AI 味句式、引用编号有没有丢失。我论文后面几章的润色，基本上一到两轮就能定稿，之前至少要三四轮。

/pdf：读 PDF 文献最常用的 Skill。我用它超过 100 次会话。写论文最常干的一件事就是核对引用：导师说「你引的这篇文献，原文真的是这么说的吗?」你得打开 PDF，翻到对应页，找到那句话。以前我是在 PDF 阅读器里用搜索功能找，找到之后再手动复制到 Claude Code 的对话里。装了 `/pdf` 之后，直接在对话里说「读一下这篇 PDF 的第 12 页，找到关于工具变量选取的那段话」，几秒钟就给你了。

这个 Skill 能做的事比读文本多得多。合并多个 PDF（比如把论文正文和附录合成一份提交）、提取表格数据、给 PDF 加水印、甚至对扫描件做 OCR 识别。有一次我拿到了一篇九十年代的文献，只有扫描版 PDF，文字没法选中。我调用 `/pdf`，它自动做了文字识别，虽然有个别字识别错了，但大部分内容都提取出来了，比我自己一个字一个字敲快太多。

/drawio：画流程图和框架图。我用它超过 100 次会话，比 `/pdf` 还多。写论文几乎每一章都要画图：研究框

© 2026 晨瀚宇 未经授权禁止转载

架图、方法流程图、系统架构图、变量关系图。以前我用 Visio 或者 PowerPoint 画，调对齐、调间距、调箭头方向，一张图花一两个小时。装了 `/drawio` 之后，我在对话里用文字描述图的结构，比如「画一个双向固定效应模型的估计流程，从数据清洗到最终结果，中间要有面板数据构建、个体固定效应、时间固定效应、聚类稳健标准误这几个步骤」，它直接生成一个图文件。我用 draw.io 打开，微调一下位置就能用了。

它生成的图是矢量的，放大不模糊，直接嵌到论文里很清楚。我论文里大概有 15 张图，其中 10 张是用 `/drawio` 生成初稿，然后自己手动调整细节完成的。

/arxiv-database：搜 arXiv 论文。我用了 90 次会话。做文献综述的时候，你需要大量检索最新的研究。以前我是去 arXiv 网站手动搜，一篇篇点开看摘要，觉得相关的就下载。装了这个 Skill 之后，可以直接在 Claude Code 里按关键词、作者、类别、日期范围搜索，结果直接列出标题、摘要、作者、发表日期。我写文献综述的流程变成了：先用 `/arxiv-database` 检索 50 到 100 篇候选文献，让 Claude Code 按摘要筛选出和我研究问题最相关的 10 到 15 篇，然后逐篇精读。

有一次我在写一个关于空间计量经济学的文献梳理，用 `/arxiv-database` 搜了 `spatial econometrics panel data` 这个关键词，返回了 47 篇论文。我让 Claude Code 按摘要内容把它们分成三类：纯方法论的、有实证应用的、综述类的。它分完之后每类给了一个简要总结，我据此决定哪些要精读、哪些只需引用结论。整个过程半小时，以前我至少要花一整天。

/docx：处理 Word 文档。我用了 66 次会话。虽然我最终用 LaTeX 排版，但论文的初稿和导师的批注都是 Word 格式。导师习惯用 Word 的批注功能留修改意见，我需要读取这些批注，然后在 LaTeX 源文件里对应修改。`/docx` 这个 Skill 能直接读 Word 文件的内容、批注、修订记录，还能创建新的 Word 文档。有一次导师要一份 Word 版的论文摘要（学校要求 Word 格式提交），我让 Claude Code 读 LaTeX 源文件、提取摘要部分、生成一份带学校抬头格式的 Word 文件，十分钟搞定。

/citation-management：管理参考文献。我用了 16 次会话，次数不多但每次都很关键。这个 Skill 能搜索 Google Scholar 和 PubMed，提取论文的完整元数据（作者、标题、期刊、年份、DOI），生成格式正确的 BibTeX 条目。我论文有一百多篇参考文献，提交前需要逐条核查格式。手动做这件事你可以想象有多崩溃：每条引用都要去数据库核实作者名有没有拼错、期刊名有没有缩写不一致、年份有没有对。我用这个 Skill 批量核查，它发现了 11 条格式问题，包括 3 个作者名拼写错误和 2 个期刊名缩写不一致。

/markitdown：文件格式转换。这个 Skill 由微软开发，能把 PDF、Word、PPT、Excel、图片等 15 种以上格式转成 Markdown。我用了 8 次，主要场景是把导师发来的 Word 版修改意见转成 Markdown，这样 Claude Code 能更好地解析结构化内容。还有一次我把一份 Excel 格式的数据说明转成 Markdown，方便直接嵌入论文的附录描述。

/pyzotero：连接 Zotero 文献库。我用了 12 次。如果你用 Zotero 管理文献（大部分研究生都用），这个 Skill 能直接连上你的 Zotero，读取文献库里的条目、标签、附件。我用它做过一件事：让 Claude Code 从我的 Zotero 库里拉出所有标记为「核心文献」的条目，按发表年份排序，生成一份文献梳理的草稿框架。这个框架后来成了我文献综述章节的骨架。

这些 Skill 覆盖了科研写作中最常见的操作：读 PDF、画图、搜文献、处理 Word、管引用、转格式、连文献库。你不需要把所有 Skill 都装一遍，根据自己的需求挑几个就够了。后面第四节会讲具体怎么安装。

不过，现成的 Skill 再多，总有覆盖不到的地方。比如你的论文有一套独特的写作规则，或者你的研究领域有特殊的术语规范，这些东西不可能有现成的 Skill。这时候就需要自己写一个。

© 2026 晨瀚宇 未经授权禁止转载

9.3 写一个属于你的 Skill

现成的 Skill 覆盖了大部分通用操作，但有些需求是你独有的。比如你的导师对论文有一套特定的批注习惯和格式偏好，这些东西不可能有现成 Skill。这时候就需要自己写一个。

这一节讲我怎么把导师的修改要求整理成了一个自定义 Skill。

为什么要自建 Skill。我导师开会的时候经常反复提同一批要求：标题要抽象化不能太直白、每段开头用一句话概括核心论点、不要用「本章将介绍」这种套路开头、表格用三线表。这些要求散落在不同次的会议记录里，我每次让 Claude Code 帮我改论文的时候，都要从笔记里翻出来粘进去。经常漏掉几条，改完之后导师又指出同样的问题。

后来我把导师所有的批注意见汇总到一起，整理成一个 Skill。从此每次润色之前先调用一下，导师的所有要求自动加载，不会再遗漏。

创建过程。Skill 的文件结构很简单。在 `~/.claude/skills/` 下创建一个文件夹，名字就是你想要的斜杠命令名称，里面放一个 `SKILL.md` 文件。文件开头写上名称和适用场景描述，后面就是具体的规则内容。

我把导师的要求分成了几个部分。

第一部分：标题规范。导师最常批的就是标题。他说过「你这个标题像小学生写的」「标题要提炼，不能简单描述事情」。我把他的要求总结成规则：标题要体现学术抽象性，层层递进从大到小，禁止「XX 在 YY 中的应用」这种直白格式。

第二部分：段落结构。导师要求每段开头一句话点明核心论点，后面展开论述，不能一段里说三件事。还有句子长度的要求：超过两行就断开。这些我都写成了明确的规则。

第三部分：术语锁定表。把论文里所有不能被修改的专业术语列出来，每个术语旁边写错误替换。比如你的论文里如果用「召回率」就不能被改成「查全率」，用「多智能体」就不能被改成「多代理」。给反面例子很重要，因为 Claude Code 特别喜欢「优化」术语。

第四部分：禁止清单。导师明确反对的写法全部列出来。比如禁止「由于……因此……」的长句式，禁止「综上所述」「具有重要意义」等空洞修饰词，禁止口语化表述。每条都附上修改后的正面例子。

第五部分：检查清单。润色完每一节后逐条核查：术语有没有被改动？有没有超过两行的长句？标题是不是太直白？段落是不是一段只说一件事？引用编号有没有丢失？

整个 `SKILL.md` 写下来大概一百多行。第一版只有几十行，后来每次导师提出新要求就补一条。这个 Skill 是跟着论文修改过程一起成长的。

注 自建 Skill 的几条经验。第一，反面例子比正面例子更重要。你告诉它「标题要学术化」它不一定理解你的标准，但你告诉它「禁止写成'XX 技术在 YY 领域的应用研究'」它就知道了。第二，规则要逐步积累。第一版把最核心的几条写进去，后面每次发现新问题就补一条。第三，检查清单放在最后效果很好，Claude Code 会自动对照核查。第四，把导师的原话直接写进规则里，比你自己转述更准确。比如直接写「导师要求：一句话超过两行就断开」，Claude Code 执行起来比「句子要短」精确得多。

写到这里，你应该已经理解了 Skill 的原理和写法。最后一节讲操作流程，让你跟着做就能把科研常用的 Skill 装上。

© 2026 晨瀚宇 未经授权禁止转载

9.4 实操：安装 3 个科研必备 Skill

这一节带你完整走一遍安装流程。我选了三个科研写作中最常用的 Skill：`/pdf`（读 PDF 文献）、`/arxiv-database`（搜 arXiv 论文）和 `/citation-management`（管理参考文献）。装完之后你可以立刻在论文工作中用上。

安装方式。最简单的方式是在 Claude Code 对话里直接说。比如你想装 `/pdf`，打开 Claude Code，输入「帮我安装 pdf 这个 skill」，它会自动下载并放到正确的位置。装 `/arxiv-database` 和 `/citation-management` 也是同样的操作，每个几秒钟就装好了。

如果你拿到的是别人分享的 Skill 文件夹（比如前面讲的 `humanizer-zh`），把整个文件夹拷到 `~/.claude/skills/` 目录下就行。复制完之后在 Claude Code 里就能用斜杠命令调用了。

验证安装成功。装完之后试一下能不能正常调用。比如验证 `/pdf`，在 Claude Code 里输入：

```
/pdf
读一下 ~/Desktop/sample_paper.pdf 的第一页，告诉我标题和摘要。
```

如果它能正确提取出标题和摘要，说明工作正常。如果缺少依赖库，Claude Code 会提示你安装，按提示操作就行。`/arxiv-database` 和 `/citation-management` 也是类似的验证方式：给它一个具体的搜索任务，看能不能返回结果。

安装更多 Skill。如果你想看看还有什么 Skill 可用，在 Claude Code 里说「我需要一个能画流程图的 Skill」，它会推荐 `/drawio`。遇到一个重复性的操作，先看看有没有现成的 Skill，如果有就装上，如果没有就参考第三节的方法自己写一个。大半年下来，我常用的大概二十几个，其中自己写的有三四个，其余都是社区现成的。不用的 Skill 放着不碍事，它们不会占用运行资源，只有你主动调用时才会加载。

几个实用建议。第一，先装 `/pdf`，这是科研场景下使用频率最高的 Skill，几乎每次工作都会用到。第二，如果你做的研究涉及大量文献检索，`/arxiv-database` 和 `/citation-management` 搭配使用效果很好：前者负责搜论文，后者负责生成 BibTeX 引用。第三，如果你需要画论文里的图，`/drawio` 值得一装，它能帮你快速生成流程图的初稿，然后你在 draw.io 软件里微调。第四，如果你有 Zotero 文献库，可以装 `/pyzotero`，直接在 Claude Code 里读取你的文献库，做文献梳理的时候特别方便。

到这里，Skill 的概念、科研常用 Skill、自己写 Skill 的方法、安装操作流程都讲完了。我论文的每一次润色，都经过了 `/humanizer-zh` 这个 Skill 的规则约束：术语锁定、表达规范、逻辑连贯、没有「不是... 而是...」、引用编号完整保留。最终提交的版本，导师只改了三处措辞，没有再提术语不一致或者格式问题。如果你的论文也有一套反复使用的润色规则，花一个晚上写成 Skill，后面每次润色都能省很多功夫。下一章讲怎么让多个 Agent 协作完成一个大任务。

© 2026 晨瀚宇 未经授权禁止转载

第十章 并行 Agent：批量任务的加速方案

内容提要

- 判断并行的两个条件：任务可分割（各单元无依赖）且结果可合并
- 完整实战案例：用 3 个并行 Agent 在 40 分钟内核查全部参考文献
- 中断与断点续跑：中间结果逐条落盘、checkpoint 文件、claude –continue 接续
- Rate limit 规律与应对：错峰启动、自动等待、预设恢复策略

10.1 什么任务适合并行

写论文到后期，会遇到一类特殊的任务：工作量大、每一步都差不多、但就是要花很多时间。所有参考文献逐条核查，7 章正文的术语一致性排查，40 张图表的格式逐一检查。这些任务有一个共同特点：每一条、每一章、每一张图的处理方式几乎一样，彼此之间没有依赖关系，第 52 条引用的核查结果不影响第 53 条怎么查。

定义 10.1 (并行 Agent)

并行 Agent 指 Claude Code 在一次对话中自动派发多个子 Agent（sub-agent），每个子 Agent 处理同一批量任务的不同子集，彼此独立工作，最后由主 Agent 汇总结果。你只需要在一个对话窗口里描述任务，Claude Code 自己判断怎么拆分、派发几个子 Agent、最后怎么合并。适用条件：任务可分割（各子任务之间无依赖）且结果可合并。

这类任务就是并行 Agent 的用武之地。第四章讲过一个引用核查的例子，提到了 40 分钟查完全部引用。那个例子侧重结果。这一章从技术角度展开：怎么判断一个任务适不适合拆分，怎么把任务分给多个 Agent，中间遇到问题怎么接续，最后结果怎么合并。

1. 判断标准：两个条件

一个任务适合并行，需要同时满足两个条件。

第一个条件是「可分割」。任务能拆成若干独立的小块，每一块可以单独完成，不需要知道其他块的结果。一百多条引用可以拆成 3 批各几十条，因为核查第 1 条引用的时候不需要知道第 53 条的核查结果。7 章正文的术语排查可以拆成 7 个任务，每章独立检查。但如果你的任务是「先写完第三章，再根据第三章的结论写第四章的引言」，这就不能并行，因为第四章依赖第三章的输出。

第二个条件是「可合并」。各个小块完成之后，结果能方便地汇总成一份完整的交付物。引用核查的结果是每条引用的状态（正确/有问题），三批结果拼在一起就是完整的核查报告。术语排查的结果是每章的不一致列表，七份列表合并就是全文的术语问题清单。但如果各块的结果之间有复杂的交叉引用（比如第三章发现的术语问题会影响第五章的某个定义），合并的时候还需要额外的对齐工作，并行的收益就会打折扣。

用一个简单的问题来判断：「如果我把这个任务交给三个互不认识的助理，每人做一部分，最后我只需要把三份结果拼在一起，不需要做大量的协调工作，那这个任务就适合并行。」

2. 不适合并行的任务

反过来看，哪些任务不适合并行，反而更重要，因为强行并行会浪费时间甚至出错。

逻辑连贯的写作不适合并行。你不能让三个 Agent 分别写第三章的三个小节，然后拼在一起，因为小节之间

© 2026 晨瀚宇 未经授权禁止转载

的论证是递进的，第二节要接着第一节的结论往下说。拼出来的文章一定有逻辑断裂。

需要全局视角的任务不适合并行。比如全文的行文风格统一，需要一个 Agent 从头到尾读完全文，才能判断哪些地方语气不一致。你把每章分给不同的 Agent，每个 Agent 只看自己那一章，它没有全局视角，没法判断第三章的用词风格是否和第七章一致。

涉及相互依赖的修改不适合并行。比如你的论文里有交叉引用，第三章引用了第五章的一个公式编号。如果两个 Agent 分别改第三章和第五章，第五章的 Agent 加了一个新公式，编号变了，但第三章的 Agent 不知道，引用就断了。

我犯过一次这样的错。当时想加快进度，让两个 Agent 分别润色第三章和第四章的文字。第三章的末尾有一句「如下一章所述，空间杜宾模型可以解决这一问题」，第四章的 Agent 把空间杜宾模型那一节的标题改了，从「空间杜宾模型的构建」改成了「空间计量模型的设定」。两个 Agent 各自干得都对，但拼起来第三章的预告和第四章的实际内容就对不上了。后来花了半小时手动检查两章之间的衔接。这半小时本来是可以省掉的，如果一开始就用一个 Agent 按顺序处理两章的话。

3. 科研论文中常见的并行场景

具体到写论文这件事，哪些场景用过并行、效果好，值得列一下。

参考文献核查是最典型的场景，前面说过，全部引用分 3 批查。图表格式检查也很适合：论文里有 40 张图，每张图要检查字号、字体、坐标轴标签、图注格式是否符合学校规范，40 张拆成 3 批或 4 批，每批独立检查。数据核对同理：论文里有 20 个回归表格，每个表格里的系数、标准误、显著性星号，要和原始回归输出对照，20 个表格拆成几批，各查各的。

还有一个场景是翻译。如果你的论文需要写英文摘要，或者某些章节需要中英双语对照，把不同段落分给不同的 Agent 翻译，最后合并，效果也不错。但要注意：术语的翻译要提前约定好，在每个 Agent 的指令里都附上术语对照表，否则三个 Agent 可能把「被解释变量」翻译成三个不同的英文表述。

不适合并行的场景前面讲过了。补充一点：修改导师批注也不太适合并行。导师的批注往往有内在逻辑（「第三章的 A 问题」和「第五章的 B 问题」可能是同一个根源），需要一个 Agent 通读所有批注，理解整体意图，再逐条处理。

4. 并行的收益和代价

收益很直观：时间缩短。三个 Agent 同时工作，理想情况下时间缩短到原来的三分之一。实际上达不到三分之一，因为要花时间拆分任务、写指令、最后汇总结果，但缩短到原来的二分之一是很常见的。

代价也要看清楚。每个 Agent 都是一个独立的会话，各自消耗使用额度。三个 Agent 并行跑 40 分钟，消耗的额度相当于一个 Agent 跑两小时。如果你的订阅方案有额度限制，并行跑得快但额度消耗也快。另外，并行 Agent 之间没法互相沟通。如果 Agent A 在核查引用的时候发现一篇文章被撤回了，Agent B 和 Agent C 不会知道这件事。你需要在最后汇总的时候自己处理这些交叉发现。

还有一个隐性代价：准备工作。你需要把任务拆分好，给每个 Agent 写清楚指令，指定好各自的输出文件名和格式。这些准备工作对于小任务来说不值得，十条引用，一个 Agent 几分钟就查完了，拆成三份反而多花时间。经验上，任务量超过 50 个重复单元（50 条引用、50 段文本、50 张图）的时候，并行才有明显的时间收益。

了解了判断标准，接下来用一个完整的例子看看并行 Agent 怎么落地。

© 2026 晨瀚宇 未经授权禁止转载

表 10.1: 适合并行与不适合并行的任务特征对比

适合并行的任务	不适合并行的任务
每一单元处理方式相同（如逐条核查引用）	处理方式随上下文变化（如写连贯的论证段落）
各单元之间无依赖（第 52 条不影响第 53 条）	后续步骤依赖前面的输出（如先写结论再写摘要）
结果可以简单拼合（合并 JSON 或列表）	结果之间有交叉引用需要对齐
任务量大于 50 个重复单元	任务量小，拆分的准备时间超过节省的时间

10.2 引用核查的并行实战

第四章讲过，提交前一周我用并行 Agent 核查了全部引用，40 分钟找出 11 处问题。那一节侧重「发生了什么」。这一节换个角度，讲「怎么做到的」：任务怎么拆分、指令怎么写、中间出了什么问题、结果怎么汇总。如果你也有类似的批量任务，可以直接套用这个流程。

1. 拆分任务

全部引用，我决定分成 3 批。为什么是 3 批而不是 5 批或 10 批？两个考虑。

第一，每批的量要够大，让 Agent 能连续工作一段时间。如果每批只有 15 条，Agent 几分钟就做完了，你花在开会话、写指令、检查结果上的时间比 Agent 工作的时间还长。每批 50 条左右，Agent 大约工作十几分钟，这个粒度比较合适。

第二，并行的数量要考虑使用额度。多个子 Agent 同时跑，额度消耗是成倍的。Claude Code 会根据当前的额度情况自动决定派几个子 Agent，通常是 2 到 3 个。

具体怎么分批，你不需要操心。我当时只是在对话里说了任务要求，Claude Code 自动把 156 条引用分成了几批，每批派一个子 Agent 去处理。

2. 写指令

你不需要给每个子 Agent 分别写指令，也不需要手动指定分批方式和输出格式。你只需要在一个对话里把任务说清楚，Claude Code 会自动拆分和派发。

我实际发出去的指令大意是这样的：

```
读取 references.bib，对每一条引用做以下核查：
- 作者姓名拼写是否正确
- 发表年份是否与实际发表年份一致
- 期刊名/会议名是否完整（不用缩写）
- 卷号、期号、页码是否齐全
- DOI是否可以正常解析
把有问题的条目列出来，写入 citation_check_report.md。
```

就这么简单。Claude Code 收到这条指令后，自动判断 156 条引用量比较大，派了几个子 Agent 分头核查，最后把结果合并给我。整个过程我不需要关心它分了几批、每批多少条、用什么格式存中间结果。

指令里唯一需要注意的是把核查标准说清楚。「作者姓名拼写是否正确」「DOI 是否可以正常解析」这些检查项，如果不写，它可能只做最基础的格式检查，漏掉你关心的维度。

3. 启动并行

你不需要手动开多个终端窗口。把指令发出去之后，Claude Code 会自动派发子 Agent 在后台并行工作。你在对话窗口里能看到它在调度——通常会显示类似「启动 Agent 处理第 1-52 条」「启动 Agent 处理第 53-104 条」的提示。

© 2026 晨瀚宇 未经授权禁止转载

启动之后你基本上不需要做什么，等它汇总完结果就行。我那次大概等了 40 分钟左右，它自动给我返回了汇总报告。

4. 汇总结果

子 Agent 各自完成后，Claude Code 自动把结果合并成一份报告返回给我。我不需要另开会话做汇总——它直接给出了最终结果：156 条引用中 145 条无问题，11 条有问题。问题分类是：3 处年份错误、4 处页码不完整、2 处期刊名缩写不统一、2 处 DOI 无法解析。

5. 处理问题

汇总报告列出了 11 条有问题的引用，但修改是我自己做的。Claude Code 告诉我「Smith et al. (2019) 的实际发表年份是 2020」，但我要自己去确认这个信息是否正确（有时候同一篇文章的会议版是 2019 年、期刊版是 2020 年，引的是哪个版本要看我在正文里引用的是什么内容）。

年份错误和页码不完整的，查到正确信息后直接让 Claude Code 改 BibTeX 文件。那两条 DOI 无法解析的，一条是 DOI 格式有一个多余的空格，去掉空格就好了。另一条是文章被撤回了，DOI 指向了一个撤回声明页面。这篇文章我引用它是为了说明某个方法「曾被使用过」，换了另一篇类似研究的引用。

从三个 Agent 开始跑到所有 11 条问题处理完毕，总共花了大约两个小时。其中 Agent 并行核查用了 45 分钟，汇总用了几分钟，我自己处理问题用了大约一小时。和手动核查对比，时间节省是一方面，更大的价值是不会遗漏：每条引用都按同一个标准查过了，不存在「查到第 120 条的时候累了、后面 36 条草草扫过」的情况。

6. 可复用的流程模板

把这次经验抽象一下，并行 Agent 处理批量任务的流程可以归纳为五步：

第一步，判断任务是否满足「可分割、可合并」两个条件。如果满足，估算总量，决定分几批。

第二步，确定每批的范围和输出格式。范围可以按顺序分（前 50 条、后 50 条）、按章节分（第一章到第三章、第四章到第七章）。输出格式提前约定好，建议用 JSON，因为结构清晰、合并方便。

第三步，写指令。关键要素：输入范围、处理步骤、输出文件名和格式、中断后的恢复策略。三个 Agent 的指令除了范围和文件名不同，其余完全一样。

第四步，启动并行，等待完成。

第五步，汇总。开一个新的会话，读取所有输出文件，合并生成最终报告。

这个五步流程，不只适用于引用核查。论文里的表格数据核查、图表格式检查、多章节的术语一致性排查，都可以用同样的流程。关键是第一步的判断：任务能不能拆、结果能不能拼。

10.3 长任务中断与接续

并行 Agent 通常在几十分钟内就能完成，大部分情况下你不需要操心中断的问题。但如果任务特别大（比如要核查两三百条引用），或者你的电脑中途休眠了、终端不小心关掉了，可能会碰到会话中断。

用 `claude --continue` 接续

会话中断之后，在终端里输入 `claude --continue`，可以恢复上一次会话的上下文。恢复之后告诉它「从上次停的地方继续」，它通常能接着做。

一个注意事项：`claude --continue` 恢复的是最近一次会话。如果中断后你又开了新会话做别的事，再用 `--continue` 恢复的就是那个新会话了。所以长任务被中断后，直接用 `--continue` 接续，不要先开别的会话。

checkpoint 的用法

© 2026 晨瀚宇 未经授权禁止转载

对于跨天的大任务，可以在指令里要求 Claude Code 维护一个 checkpoint 文件记录进度（第五章也提过这个做法）。这样即使会话上下文被压缩了，它读一下 checkpoint 文件就知道做到哪了。

关于限频（rate limit）

Claude Code 碰到限频会自动等待然后继续，你不需要在指令里写「碰到限频等两分钟再继续」这种话。如果你的订阅方案额度比较紧张，并行任务跑得多了可能会触发限频，但这个是 Claude Code 内部处理的，你感知到的就是它有时候会暂停一会儿，然后自动恢复。

注 大多数并行任务在一次对话内就能完成。只有跨天的特大任务才需要 checkpoint 机制。如果你的引用数量在两百条以内、术语排查在十章以内，通常一次对话就能跑完，不用担心中断问题。

10.4 实操：全文术语一致性检查

前面讲了判断标准和中断处理，这一节用一个完整的操作流程来演示。场景是：论文有 7 章，导师让你在提交前做一次全面的术语一致性检查。

1. 任务分析

先判断是否适合并行。术语一致性检查是逐章扫描术语、找出不统一的表述。每一章的扫描是独立的，但最后的「统一用哪个表述」需要看完所有章节才能决定。所以：检查阶段可以并行，统一修改阶段需要串行。

2. 给 Claude Code 下指令

在一个对话里把任务说清楚就行：

```
读取论文的7章正文（ch1.tex 到 ch7.tex）。
找出所有专业术语，检查同一个概念在不同章节是否用了
不同的表述。重点检查：
- 同义词混用（如"被解释变量"和"因变量"）
- 英文术语大小写不统一（如"Panel Data"和"panel data"）
- 缩写是否在首次出现时给出了全称
列出每个不一致的术语：各章分别用了什么表述、出现几次。
```

Claude Code 收到之后，自动派了几个子 Agent 分头扫描不同章节，大约 25 分钟后返回了汇总结果。我不需要手动分批、不需要建工作目录、不需要指定 JSON 格式。

3. 查看结果

它返回的汇总结果大概是这样的：

```
## 术语不一致列表

1. "被解释变量" vs "因变量"
   - 第一章到第三章：被解释变量（23次）、因变量（2次）
   - 第四章和第五章：被解释变量（8次）、因变量（15次）
   - 第六章和第七章：因变量（12次）、被解释变量（0次）

2. "Panel Data" vs "panel data" vs "面板数据"
   - 第一章到第三章：面板数据（18次）、Panel Data（3次）
   - 第四章和第五章：面板数据（10次）、panel data（5次）
```


© 2026 晨瀚宇 未经授权禁止转载

```
   - 第六章和第七章：面板数据（14次）

3. "工具变量法" vs "IV估计" vs "IV方法"
   - 第一章到第三章：工具变量法（6次）
   - 第四章和第五章：IV估计（9次）、工具变量法（2次）
   - 第六章和第七章：IV方法（4次）、工具变量法（3次）
```

看到这个报告，你需要做的是决策：每个术语统一用哪个表述。「被解释变量」在前三章用得多，「因变量」在后四章用得多，你要根据自己论文的学科惯例和导师偏好来决定。比如计量经济学论文通常用「被解释变量」，那就全部统一成「被解释变量」。

决策完之后，让 Claude Code 执行替换：

```
根据以下规则，全文替换术语：
- "因变量" 全部替换为 "被解释变量"
- "Panel Data" 和 "panel data" 全部替换为 "面板数据"
- "IV估计" 和 "IV方法" 全部替换为 "工具变量法"
替换前先备份所有 .tex 文件。
替换后告诉我每个文件改了几处。
```

这一步是串行的，因为替换涉及全文所有章节，需要一个 Agent 统一处理才能保证一致。

5. 验证

替换完之后，再跑一遍术语检查来验证。让 Claude Code 读取所有章节，确认那三组术语是否已经完全统一。如果还有遗漏（比如有些术语藏在表格或脚注里，第一轮没覆盖到），再处理。

6. 时间对比

手动检查 7 章的术语一致性，逐字阅读五万字至少三四个小时，而且很难保证不遗漏。用 Claude Code 的并行 Agent，从发出指令到替换完成并验证，总共大约一小时。更大的价值在全面性上：它会把脚注、表格标题、图注里的术语也扫到，这些是人眼容易滑过的。

同样的方式可以直接套用到图表格式检查、数据核查、参考文献格式统一等场景。你只需要在一个对话里把任务说清楚，Claude Code 自己判断怎么拆分和并行。

到这里，并行 Agent 的做法讲完了。下一章讲 Hooks，可以把一些操作性的要求（比如每次修改文件前自动备份）固定成自动触发的规则。

© 2026 晨瀚宇 未经授权禁止转载

第十一章 Hooks：自动化触发器

内容提要

- Hook 是什么：Claude Code 的系统级自动化触发器，将「建议性」规则变为「强制执行」
- Hook 与 CLAUDE.md 的区别：前者嵌在执行流程里跳不过，后者是对话层面的文字提示偶尔被忽略
- PreEdit Hook 实战：修改.docx 文件前自动备份，配置过程全程用自然语言描述
- PostEdit Hook 实战：修改.tex 文件后自动运行格式检查脚本，逐步扩展检查规则

第八章讲备份策略的时候，我写了一段话：「手动备份的问题是你可能忘记。改着改着投入进去了，忘了先跑备份命令，等出了问题才想起来。」当时给的解决办法是把备份规则写进 CLAUDE.md，让 Claude Code 每次启动的时候读到这条规则，改文件之前先备份。

这个方法管用，但有一个漏洞：CLAUDE.md 里的规则是「建议」，Claude Code 会尽量遵守，但偶尔会忽略。我统计了一下，写进 CLAUDE.md 之后的 30 次会话里，它大概有 4 次忘了先备份就直接开始改文件。概率不高，但只要出一次问题，你就得花时间去恢复。

Hook 解决的就是这个问题。它把「建议」变成了「强制」：你设一个规则，告诉 Claude Code「每次修改文件之前，必须先执行这个命令」，它就会在每次编辑操作触发之前自动跑这个命令。跳不过、忘不了，因为这个机制是在 Claude Code 的底层运行的，和它的对话理解能力无关。

这一章讲三件事：Hook 是什么概念、科研写作中两个最实用的 Hook 配置、从零配置一个备份 Hook 的完整流程。

11.1 什么是 Hook

定义 11.1 (Hook)

Hook 是 Claude Code 的系统级自动化触发器。当指定事件发生时，Claude Code 在继续执行前会强制运行你预设的终端命令。Hook 配置存储在 Claude Code 设置文件中，独立于对话上下文，不受上下文长度或注意力影响。常用事件类型：PreEdit（修改文件之前触发）和 PostEdit（修改文件之后触发）。

用一句话解释：Hook 是 Claude Code 的自动化触发器，当某个事件发生时，自动执行你预设的命令。

这个概念并不新。如果你用过 Git，你可能见过 pre-commit hook：每次你提交代码之前，Git 自动跑一遍代码检查，检查不通过就不让你提交。Claude Code 的 Hook 原理一样，只是触发的事件不同。Git 的 Hook 是「提交代码时触发」，Claude Code 的 Hook 是「编辑文件时触发」「发消息时触发」这些和 AI 交互相关的事件。

Hook 的三个组成部分

一个 Hook 由三样东西组成：事件、匹配条件、要执行的命令。

事件是指「什么时候触发」。Claude Code 目前支持几种事件类型，科研写作中最常用的有两种。第一种叫 PreEdit，意思是「修改文件之前」，就是 Claude Code 准备往一个文件写入内容的那一刻，在它真正动手之前，先执行你预设的命令。第二种叫 PostEdit，意思是「修改文件之后」，就是 Claude Code 刚刚改完一个文件，在它告诉你「改好了」之前，先执行你预设的命令。

© 2026 晨瀚宇 未经授权禁止转载

匹配条件是指「对哪些文件生效」。你可以让一个 Hook 只对特定类型的文件生效。比如你只想在修改 Word 文件时触发备份，就设一个条件：文件名以`.docx`结尾。如果你想对所有文件都生效，不设条件就行。

要执行的命令是指「触发后干什么」。这个命令就是一条普通的终端命令，和你平时在命令行里手动敲的一样。比如`cp`命令复制文件、`grep`命令搜索文本、`python`命令跑一个脚本，都可以。

Hook 存在哪里

Hook 的配置存在 Claude Code 的设置文件里。这个设置文件在你的用户目录下，是一个标准的配置文件。你不需要直接去编辑这个文件的内容，因为你可以在 Claude Code 的对话里用自然语言描述你想要的 Hook，它会帮你写好配置并放到正确的位置。

这一点很重要，值得多说两句。我刚开始了解 Hook 的时候，第一反应是「又要去编辑配置文件了」。以前装软件、配环境变量，最怕的就是打开一个配置文件，里面一堆看不懂的语法，改错一个符号整个工具就不工作了。Hook 的配置也是这种结构化的格式，但你完全不需要自己去写。你只要在对话里告诉 Claude Code「我想设一个 Hook，每次修改.docx 文件之前自动备份」，它就会把配置写好。你要做的只是确认一下它的方案对不对，然后同意执行。

Hook 和 CLAUDE.md 的区别

第二章讲 CLAUDE.md 的时候，我说过可以在里面写「修改任何.docx 文件之前，先创建备份副本」。这条规则和 Hook 看起来做的是同一件事，但机制完全不同。

CLAUDE.md 里的规则是「对话层面」的：Claude Code 读到这条规则后，在对话过程中会尽量遵守。但它本质上是一个文字提示，和你在对话里说「记得先备份」效果类似。如果某次对话特别复杂，它的注意力被其他任务占满了，有可能忽略这条规则。

Hook 是「系统层面」的：它嵌在 Claude Code 的执行流程里。Claude Code 每次准备编辑一个文件，系统会先检查有没有匹配的 Hook，有的话先执行 Hook 命令，执行完了才允许继续编辑。这个过程不依赖对话理解，不受上下文长度影响，不会被遗忘。

打个比方：CLAUDE.md 里写「修改前备份」像是你在笔记本上写了一条待办事项，每次干活之前看一眼提醒自己。Hook 像是你在门上装了一把锁，出门的时候你可能忘记看待办事项，但你不可能不开锁就出门。

表 11.1: CLAUDE.md 规则与 Hook 的对比

CLAUDE.md 里写规则	配置 Hook
对话层面的文字提示，偶尔被忽略	系统层面的强制执行，跳不过去
依赖 Claude Code 的理解能力	不依赖对话，嵌在执行流程里
所有规则混在一个文件里，没有分类	每个 Hook 独立，按事件类型分开
适合写「软性」的偏好和风格要求	适合写「硬性」的必须执行的操作

我的建议是两种方式搭配使用。风格类的规则（比如「用学术语气」「保留特定术语」）写在 CLAUDE.md 里，因为这些规则需要 Claude Code 理解语义才能执行。操作类的硬性要求（比如「改文件前备份」「改完后跑一遍检查脚本」）用 Hook 来配置，因为这些操作是机械性的，不需要理解语义，只需要在正确的时机自动执行。

概念讲完了，下面看两个科研写作中最实用的 Hook。

11.2 改文件前自动备份

这是我配置的第一个 Hook，也是用得最多的一个。它的作用很简单：每次 Claude Code 准备修改一个`.docx`文件之前，自动把这个文件复制一份到备份文件夹里，文件名带上时间戳。备份完了才允许它继续修改。

© 2026 晨瀚宇 未经授权禁止转载

为什么需要这个 Hook

第八章已经详细讲过 Word 文件操作的风险。这里再补一个后来发生的真实例子。

有一天晚上我在处理导师的一批修改意见，连续改了第三章、第五章和第七章的 Word 文件。改到第七章的时候已经是凌晨一点多，导师的意见是「把第七章的空间权重矩阵描述简化，和第三章保持一致」。我让 Claude Code 读了两章的内容然后修改第七章，它改完之后我快速扫了一眼，觉得没问题就继续处理下一条意见了。

第二天早上打开第七章一看，发现不对：它不光简化了空间权重矩阵的描述，还把前面两段关于模型设定的表述也改了，用词和我论文其他地方不一致。我想回退到修改之前的版本，打开备份文件夹，发现第七章的备份停在昨天下午三点。晚上那一轮密集修改，我一次都没备份。

原因很简单：改到凌晨的时候，我的注意力全放在导师的修改意见上，根本没想起来要先跑备份命令。CLAUDE.md 里虽然写了「修改前先备份」，但那天晚上连续改了十几个文件，它有几次确实先备份了，但第七章那次没有。

这件事之后我配置了 PreEdit Hook。从那以后，备份这件事就完全不需要我操心了。

配置过程

配置这个 Hook 不需要你去找配置文件在哪里、打开它、写入语法正确的内容。你在 Claude Code 的对话里直接说就行：

```
帮我配置一个Hook：每次修改 .docx 文件之前，
自动把这个文件备份到 backups/ 文件夹里，
文件名格式是「原文件名_年月日_时分秒.docx」。
```

Claude Code 会告诉你它准备怎么做：创建一个 PreEdit 类型的 Hook，匹配条件是文件名以`.docx`结尾，触发时执行一条备份命令。你确认之后，它就把配置写好了。

这里面有几个细节值得说说。

备份文件夹的位置。我一开始让它把备份放在当前项目文件夹下的`backups/`里。后来发现一个问题：如果你用 Git 管理项目，这个备份文件夹会被 Git 追踪，每次提交都会包含一堆备份文件。解决办法有两个，一是在`.gitignore`里加一行把`backups/`排除掉，二是把备份放在项目文件夹外面。我最后选了第一种，因为备份放在项目里找起来方便。

文件名的时间戳格式。我用的是「年月日_时分秒」，比如`ch7_method_20260315_013042.docx`。这个格式的好处是按文件名排序就是按时间排序。有的同学喜欢用 Unix 时间戳（一串数字），优点是绝对不会重名，缺点是你看到`1710456642`这个数字，不知道是几月几号。我的经验是：备份频率不高的话（一天十几次以内），可读的日期格式更实用。

如果备份文件夹不存在怎么办。Claude Code 生成的备份命令里会包含创建文件夹的操作，如果文件夹已经存在就跳过，不存在就先创建。你不用自己事先建好文件夹。

配置好之后的效果

配好 PreEdit Hook 之后，整个备份过程变成了全自动。我在对话里说「把第七章的空间权重矩阵描述简化」，Claude Code 准备修改`ch7_method.docx`的时候，系统先检测到这是一个`.docx`文件，触发 PreEdit Hook，自动执行备份命令，把当前版本复制到`backups/`文件夹。备份完成后，Claude Code 才开始修改文件。整个过程大概多花两三秒，我几乎感觉不到。

我用这个 Hook 三个多月，备份文件夹里攒了 200 多个备份文件。其中大概有 7、8 次我真的需要回退到之前的版本：有几次是 Claude Code 改错了内容，有几次是我改完之后觉得不如之前的好，还有一次是导师看了改后的版本说「还是之前的写法好」。每次回退都很快，去备份文件夹里找到时间最近的那个版本，拷回来就行。

© 2026 晨瀚宇 未经授权禁止转载

如果没有这个 Hook，这 7、8 次回退里至少有一半是找不到备份的。因为那些时候我都是在密集修改，注意力全在内容上，手动备份这件事根本想不起来。

针对 LaTeX 文件的调整

后来我把论文从 Word 转到了 LaTeX，备份需求变了。LaTeX 文件是纯文本，用 Git 管理版本已经很好了，每次提交都能看到逐行差异。但有一种情况 Git 帮不上忙：你让 Claude Code 连续改了好几次，还没来得及 Git 提交，中间某一次改错了，你想回到上一次修改之后的状态。这时候 PreEdit Hook 的备份就派上用场了。

我针对`.tex`文件也配了一个类似的 Hook，但策略不太一样。Word 文件我是每次修改都备份，因为 Word 文件的改动没法用文本差异对比。LaTeX 文件我设的条件宽松一些：只备份超过一定大小的文件。一两行的小改动，Git 提交就够了；但如果是大篇幅的修改，多一层备份总是安心的。

这种细节你不用记住具体怎么配，在对话里告诉 Claude Code 你的需求，它会处理配置。你只要想清楚：对哪些文件、在什么时候、做什么操作。

11.3 写完章节自动格式检查

第二个 Hook 是 PostEdit 类型：Claude Code 修改完文件之后，自动跑一遍格式检查。

这个需求是怎么来的

写论文的时候，我有一套润色规则（就是第九章讲的那个 Skill）。规则里有几条硬性要求：术语不能被擅自修改、不使用「不是... 而是...」句式、段落不超过 8 句。Skill 能大幅减少这些问题的出现，但不能完全消除。偶尔 Claude Code 还是会改动一两处术语，或者某一段写到了 9 句。

一开始我是自己肉眼检查的。每次它写完一节 3000 多字的内容，我从头到尾读一遍，一边读一边找破折号和超长段落。读完大概要十几分钟。有时候读到一半被别的事打断了，回来之后不确定前面检查到哪里了，又得从头看。

后来我想到一个办法：让 Claude Code 写一个简单的检查脚本，每次修改完`.tex`文件之后自动跑一遍。脚本的逻辑很简单：扫描文件内容，找出所有的破折号（中文的「——」），找出所有包含「不是」和「而是」的句子，统计每个段落的句数。如果发现了问题，就列出来告诉我在第几行、具体是什么问题。

配置过程

这个 Hook 分两步。第一步是准备检查脚本，第二步是配置 Hook 让它在正确的时机自动运行。

检查脚本这个事，我直接在对话里让 Claude Code 写的：

```
帮我写一个检查脚本，功能如下：
1. 检查 .tex 文件里有没有中文破折号
2. 检查有没有「不是...而是...」的句式
3. 统计每个段落的句数，超过 8 句的标出来
4. 输出结果到终端，格式是：第几行、什么问题、原文内容
```

它写好之后，我在几个已经定稿的章节上试跑了一遍，确认检查逻辑没问题。中间调了两次：第一次它把 LaTeX 命令里的破折号（比如注释里的）也标出来了，我让它排除掉`\%`开头的行和`lstlisting`环境里的内容；第二次是段落计数的逻辑，它一开始按空行分段，但 LaTeX 里有些空行是排版用的（比如`\\bigskip`前后），不算真正的段落分界。调完之后检查结果就比较准了。

然后配置 Hook。在对话里这样说：

© 2026 晨瀚宇 未经授权禁止转载

帮我配置一个Hook：每次修改 .tex 文件之后，
自动运行刚才那个检查脚本，把结果输出到终端。

Claude Code 把这个配置成了 PostEdit 类型的 Hook，匹配条件是文件名以.tex结尾。每次它修改完一个.tex文件，系统自动跑检查脚本，结果会显示在对话里。

实际效果

配好之后，我的写作流程变成了这样：我让 Claude Code 写一节内容，它写完之后 PostEdit Hook 自动触发，检查脚本跑一遍，结果直接显示出来。如果有问题，我当场就能让它修改，比如「第 47 行有一个破折号，改成逗号」。如果没有问题，我继续下一节。

这个 Hook 帮我省了多少时间呢？我算了一下。这本书一共大概 15 章，每章 4 节，每节写完要检查一次。按以前手动检查一次十几分钟算，15 章乘以 4 节乘以 15 分钟，大约是 15 个小时。有了 Hook 之后，检查是自动的，我只需要看结果列表，30 秒就能判断有没有问题。实际上大部分时候结果是「未发现问题」，因为 Skill 已经把大部分违规写法挡住了。Hook 起的是兜底作用：万一 Skill 漏了，Hook 能兜住。

检查项目可以逐步扩展

我的检查脚本一开始只有三条规则（破折号、「不是... 而是...」、段落超长），后来随着写作过程逐步扩展。

写到第六章的时候，我发现 Claude Code 偶尔会在正文里用分点列表（就是 Skill 里 R0 规则禁止的那种），于是在检查脚本里加了一条：检测\\begin\{itemize\}和\\begin\{enumerate\}是不是出现在了不该出现的地方。

写到第八章的时候，我在做脱敏检查，想确认书稿里没有泄露真实的研究内容。于是加了一条：搜索几个绝对不能出现的关键词。如果这些词出现在正文里（不是出现在代码示例或注释里），检查脚本会标红报警。

写到第十章的时候，我发现某些章节的篇幅不够饱满。于是又加了一条：统计当前文件的总字数，如果一节低于 2500 字就给个提醒。

到现在这个检查脚本一共有 7 条规则，都是在写作过程中一条一条加上去的。每次加新规则的方式都一样：在对话里告诉 Claude Code 你想检查什么，它改好脚本，你跑一遍确认没问题。Hook 的配置不用动，因为它只是「修改.tex文件后跑这个脚本」，脚本内容变了，Hook 自动就跑新版本了。

检查脚本不是万能的

有一点要说清楚：这种基于文本匹配的检查脚本能抓住的问题是有限的。它能准确检测破折号、特定句式、特定关键词这些可以用文本模式匹配的东西。但它检测不了「这段话写得不够具体」「这个例子是编的还是真实的」「段落之间的逻辑衔接是否通顺」这些需要理解语义的问题。

所以我的工作流是两层检查。第一层是 PostEdit Hook 跑的自动检查，抓硬性规则违规。第二层是我自己读一遍，看内容质量、逻辑连贯、例子是否真实。Hook 帮我省掉了第一层的时间和精力，让我能集中注意力在第二层。

类似的思路可以推广到其他场景。如果你的论文有特定的格式要求（比如所有表格标题必须以「表」开头、所有图片标题必须以「图」开头），你可以让 Claude Code 写一个检查脚本，配成 PostEdit Hook。如果你的导师反复强调过某个用词偏好（比如「用’ 显著’ 不用’ 明显’」），也可以加进检查脚本里。每个人的需求不一样，但配置方法都是同一套：告诉 Claude Code 你想检查什么，它写脚本、你确认、配成 Hook。

© 2026 晨瀚宇 未经授权禁止转载

11.4 实操：配置自动备份 Hook

这一节带你从零开始配置一个 PreEdit Hook，实现修改`.docx`文件前自动备份。跟着做一遍，你就知道整个流程是怎么回事了。后面配其他 Hook 都是同样的路子。

1. 确认当前状态

先确认你的 Claude Code 版本支持 Hook 功能。在 Claude Code 里输入：

```
我想配置一个Hook，你能帮我看一下当前有没有已经配置好的Hook吗？
```

它会去读设置文件，告诉你当前有没有已配置的 Hook。如果它说「当前没有任何 Hook 配置」，说明你还没配过，正好从头开始。如果它列出了一些已有的 Hook，你可以先看看都是什么，确认没有冲突。

2. 描述你想要的效果

这一步是核心。你不需要知道 Hook 的配置语法，不需要知道设置文件在哪里，不需要知道命令怎么写。你只需要用自然语言描述你想要的效果。

在 Claude Code 里输入：

```
帮我配置一个 PreEdit Hook，要求如下：
- 每次修改 .docx 文件之前触发
- 把要修改的文件备份到当前目录下的 backups/ 文件夹
- 备份文件名格式：原文件名_YYYYMMDD_HHMMSS.docx
- 如果 backups/ 文件夹不存在，自动创建
```

Claude Code 会给你一个方案，告诉你它准备怎么配置：什么类型的 Hook、匹配什么文件、触发时执行什么命令。你看一下方案，确认几个关键点：

第一，触发时机对不对。应该是 PreEdit（修改之前），不是 PostEdit（修改之后）。备份必须在修改之前做，否则备份的就是改过的版本，没意义了。

第二，匹配条件对不对。应该只匹配`.docx`文件。如果你也想备份`.tex`文件，可以加上，但建议分开配两个 Hook，因为两种文件的备份策略可能不一样。

第三，备份命令对不对。命令里应该包含创建文件夹（如果不存在的话）和复制文件两个动作，文件名里应该有时间戳。

确认无误之后，告诉它「可以，执行吧」。

3. 测试 Hook 是否工作

配置完成后，马上测试一下。最简单的测试方法是：在项目文件夹里创建一个测试用的 Word 文件，然后让 Claude Code 修改它，看备份有没有自动生成。

```
帮我在当前目录创建一个测试文件 test_hook.docx，
内容随便写一句话就行。然后修改这个文件，
把内容改成「这是修改后的内容」。
```

如果 Hook 配置正确，Claude Code 在修改`test_hook.docx`之前会先触发备份。你去`backups/`文件夹看一下，应该能看到一个带时间戳的备份文件。打开备份文件，里面应该是修改前的内容（「一句话」），而`test_hook.docx`本身应该已经变成了「这是修改后的内容」。

如果备份文件没出现，可能是配置有问题。让 Claude Code 检查一下：

© 2026 晨瀚宇 未经授权禁止转载

```
我刚才配的 PreEdit Hook 好像没触发，
帮我检查一下配置有没有问题。
```

它会去看设置文件的内容，检查语法有没有写错、匹配条件有没有写对、命令有没有拼写错误。大部分情况下是小问题，比如匹配条件写成了`.doc`而不是`.docx`，或者备份路径写错了。它改好之后再测一次就行。

4. 日常使用

测试通过之后，这个 Hook 就一直在工作了。你以后每次让 Claude Code 修改`.docx`文件，它都会先自动备份。你不需要在对话里提起备份这件事，不需要在任务描述里加「先备份」，不需要任何额外操作。

我用了一段时间之后发现备份文件会越来越多。200 多个备份文件，加起来占了好几百 MB。我每个月清理一次：把 30 天之前的备份删掉，只保留最近一个月的。删的时候不用手动一个个找，让 Claude Code 来做：

```
帮我清理 backups/ 文件夹里超过 30 天的备份文件。
先列出来让我确认，再删除。
```

你也可以把这个清理操作也配成一个 Hook 或者定期任务，但我觉得手动清理已经够了，一个月花两分钟的事。

5. 扩展：给 LaTeX 文件也配一个

如果你后面转到了 LaTeX 工作流，可以再配一个针对`.tex`文件的 PreEdit Hook。方法一模一样：

```
帮我再配一个 PreEdit Hook，
每次修改 .tex 文件之前也自动备份到 backups/ 文件夹。
```

`.tex`文件通常比`.docx`小得多（纯文本没有格式信息），所以备份占用的空间很少。我论文的 8 个`.tex`文件，每个大概几十 KB，备份三个月才占了不到 50MB。

6. 把 PostEdit 格式检查也配上

既然已经会配 PreEdit Hook 了，顺手把前面第三节讲的 PostEdit 格式检查也配上。先让 Claude Code 写好检查脚本，然后：

```
帮我配一个 PostEdit Hook，
每次修改 .tex 文件之后，自动运行检查脚本，
把检查结果输出到终端。
```

两个 Hook 配好之后，你的工作流就变成了：让 Claude Code 改文件，改之前自动备份（PreEdit Hook），改之后自动检查格式（PostEdit Hook）。你只管专注于内容本身。

回顾一下操作流程

整个配置过程就五步：确认版本支持、用自然语言描述需求、确认方案、测试、日常使用。你全程不需要去找配置文件、不需要记配置语法、不需要手动编辑任何格式化的内容。Claude Code 帮你处理所有技术细节。

注 备份策略有三个层级，可靠性依次递增：手动跑备份命令（靠你记住）、写进 CLAUDE.md（靠 Claude Code 的对话理解，偶尔会忘）、配置 Hook（系统级强制执行，跳不过去）。对于那些「忘了一次就会出问题」的操作，比如改文件前备份、改完后格式检查，用 Hook 而不是 CLAUDE.md。

到这里，Hook 这个功能就讲完了。回头看一下：第八章讲备份策略的时候，给了三种方案：手动跑备份命令、写进 CLAUDE.md 让它自动执行、用 Hook 强制执行。这三种方案是递进的，可靠性依次提高。手动跑命令靠你自己记住，CLAUDE.md 靠 Claude Code 的对话理解，Hook 靠系统级的执行流程。如果你只记住一个建议，

© 2026 晨瀚宇 未经授权禁止转载

那就是：对于那些「忘了一次就会出问题」的操作，用 Hook 配置自动执行。下一章讲 MCP 工具扩展，那是给 Claude Code 增加新能力的另一种方式。

© 2026 晨瀚宇 未经授权禁止转载

第十二章 MCP 工具扩展

内容提要

- 理解 MCP（Model Context Protocol）的本质：让 Claude Code 连接外部服务
- 区分 Skill（规则文件）与 MCP（实时连接）的不同定位
- 掌握/pyzotero 连接 Zotero 文献库，实现文献梳理、引用核查和框架生成
- 使用 OpenAlex MCP 查询发表趋势、引用关系和作者信息
- 理解各文献工具的分工：OpenAlex 做全景分析，/pyzotero 读取个人文献库，/pdf 读全文

第九章讲了 Skill，第六章用 Draw.io 画图的时候也提过 MCP 这个词。当时只是一笔带过：Claude Code 通过 MCP 协议连接 Draw.io，你在对话里输入 `/drawio`，它就能创建和编辑图表。那一章的重点是怎么画图，MCP 只是顺带提了一句。

这一章把 MCP 单独拿出来讲，因为它在科研场景下能连接的工具远不止 Draw.io。你的 Zotero 文献库、开放学术数据库、甚至邮件和日历，都可以通过 MCP 接入 Claude Code。对于做文献工作比较多的研究生来说，Zotero MCP 和学术数据库 MCP 是最实用的两个。

12.1 什么是 MCP

Skill 和 MCP 经常被搞混。第九章讲过，Skill 是一套规则文件，存在你电脑的 `~/.claude/skills/` 目录下，用斜杠命令调用。它的本质是一段预先写好的指令，告诉 Claude Code 遇到某类任务该怎么做。比如 `/pdf` 这个 Skill，规则文件里写了怎么读 PDF、怎么提取表格、怎么合并文件。Claude Code 读了这些规则，就知道该调用哪些工具来处理你的 PDF。

MCP 是另一回事。MCP 的全称是 Model Context Protocol，你不需要记住这个名字，只需要知道它做了什么：它让 Claude Code 能连上一个正在运行的外部服务，实时读取和操作那个服务里的数据。

定义 12.1 (MCP（Model Context Protocol）)

MCP 是一种协议，让 Claude Code 能与外部服务（如 Zotero、学术数据库、Draw.io 等）建立实时连接，从而读取和操作那些服务中的数据。MCP 依赖外部服务保持运行，断网或服务宕机时不可用，这是它与本地 Skill 文件的本质区别。

举个例子来说明区别。`/pdf` 这个 Skill 处理 PDF 的时候，PDF 文件就在你电脑上，Claude Code 直接读取就行。但你的 Zotero 文献库不一样，Zotero 的数据存在 Zotero 的服务器上（或者本地的数据库文件里），Claude Code 没法直接打开。MCP 就是在 Claude Code 和 Zotero 之间搭了一座桥：Zotero 那边跑着一个服务，Claude Code 通过 MCP 协议和这个服务通信，读取你文献库里的条目、标签、附件。

再比如第六章用的 Draw.io MCP。你说「帮我画一个研究框架图」，Claude Code 不是自己在命令行里画图，它是通过 MCP 连上了 Draw.io 这个画图工具，在 Draw.io 里创建图表、添加方框和箭头。Draw.io 负责渲染，Claude Code 负责指挥。这就是为什么当时讲到 session 会丢失的问题：MCP 连接的是一个外部服务的临时会话，会话断了，图就没了。

注 **Skill 是规则，MCP 是连接。**Skill 告诉 Claude Code「该怎么做」，MCP 让 Claude Code「能连上谁」。有些工

© 2026 晨瀚宇 未经授权禁止转载

具同时用了两者：`/pyzotero` 这个 Skill 的规则文件里写了怎么和 Zotero 交互，而实际连接 Zotero 的通道就是 MCP。你用的时候不需要区分哪部分是 Skill 在起作用、哪部分是 MCP 在起作用，输入斜杠命令就行了。但理解这个区别有助于你明白一件事：Skill 装了就能用，MCP 还需要外部服务跑起来才行。

MCP 对你来说意味着什么。作为使用者，你不需要关心 MCP 的技术细节。怎么启动服务、怎么配置连接参数，这些都是 Claude Code 自己处理的。你只需要知道两件事：第一，有些功能需要 MCP 才能实现（比如连接 Zotero、连接学术数据库），这些功能你在 Claude Code 的对话里直接调用就行；第二，MCP 连接的是外部服务，所以它依赖网络。如果你断网了，或者外部服务本身挂了，MCP 功能就不可用。这和 Skill 不同，Skill 是本地文件，断网也能用。

我在用 Claude Code 的过程中，涉及 MCP 的场景主要有三个：Draw.io 画图（第六章讲过了）、Zotero 文献库（这一章重点讲）、学术数据库检索（这一章也会讲）。其他 MCP 服务也有很多，比如连接 Gmail、Google Calendar、GitHub，但这些和科研写作关系不大，这里不展开。

一个常见的疑问：为什么不直接打开 Zotero 用？这个问题问得好。你当然可以打开 Zotero 桌面客户端，手动浏览文献、手动导出 BibTeX。但如果你想让 Claude Code 在帮你写文献综述的时候，自己去查你的文献库里有没有某篇文献，或者自动拉取某个标签下的所有文献条目，它就需要能「看到」你的 Zotero。没有 MCP，Claude Code 看不到 Zotero 里的任何东西，你得手动把文献信息复制粘贴给它。有了 MCP，它自己就能去查，你说一句「把我 Zotero 里标记为核心文献的条目都列出来」，它直接就能做。

这就像一个助手和一间资料室的关系。没有 MCP 的时候，助手坐在门外，你每次都要跑到资料室里取出文件递给它。有了 MCP，助手有了资料室的钥匙，它自己就能进去找。

到这里，MCP 的概念讲清楚了。下面看科研场景里最实用的两个 MCP：Zotero 和 OpenAlex。

12.2 接入 Zotero 文献库

定义 12.2 (Zotero)

Zotero 是一款开源文献管理工具，用于收藏、组织和引用学术文献。研究者可以用 Zotero 存储论文 PDF、录入文献元数据、添加标签和笔记，并将文献库与云端同步。通过 MCP 协议，Claude Code 可以实时读取 Zotero 中的元数据和笔记，实现文献的自动筛选、分组和摘要。

我用 Zotero 管理文献有三年了。从硕士开始，所有读过的论文都存在里面，按课题分了文件夹，重要的文献打了标签。到写论文的时候，Zotero 里大概有四百多篇文献，其中和我论文直接相关的有一百五六十篇。

用 Claude Code 之前，Zotero 和 Claude Code 是两个完全独立的世界。我在 Zotero 里查到一篇文献的信息，想让 Claude Code 帮我写综述的时候引用它，得手动把作者、年份、标题复制粘贴过去。如果是批量操作（比如「帮我把这 20 篇文献按研究方法分个类」），我得先在 Zotero 里手动导出这 20 篇的信息，存成文件，再让 Claude Code 读这个文件。来来回回的复制粘贴，很烦。

`/pyzotero` 解决了这个问题。装了这个 Skill 之后，Claude Code 可以直接读取我的 Zotero 文献库。我说「列出我 Zotero 里所有标记了' 核心文献' 标签的条目」，它几秒钟就返回一个列表，每条包括作者、标题、年份、期刊名。不需要我去 Zotero 里手动导出任何东西。

我用 /pyzotero 做了什么。前后 12 次会话，主要做了三类事情。

第一类是文献梳理。我写文献综述第三稿的时候（前两稿被导师退回了），需要把已有文献按研究主题重新分组。以前我是在 Excel 里手动分的，这次我让 Claude Code 直接从 Zotero 里拉取我标记了特定标签的文献，按

© 2026 晨瀚宇 未经授权禁止转载

发表年份和研究方法分组。它返回的结果让我发现一个我之前没注意到的规律：2019 年之前的文献大多用传统面板回归，2019 年之后开始密集出现空间计量的方法。这个时间节点后来成了我文献综述里「方法演进」那一段的核心线索。

第二类是核查引用。论文里引用了某篇文献，我想确认一下这篇文献在 Zotero 里的完整信息是否和 BibTeX 文件里的一致。以前的做法是打开 Zotero 客户端，搜这篇文献，手动对比。现在我直接在 Claude Code 里说「帮我查一下 Zotero 里有没有这篇文章：某作者某年的某标题」，它去查完直接告诉我：有，信息如下，和 BibTeX 里的出版年份有出入。有一次它发现 BibTeX 里写的是 2020 年发表，但 Zotero 里存的原文 PDF 标注的是 2021 年正式见刊。我去核实了一下，确实是 2021 年，BibTeX 里的年份是我当初从预印本信息录入的，忘了更新。

第三类是生成文献框架。我有一次让 Claude Code 从 Zotero 里拉出我标记了「方法论」标签的所有文献（大概 30 篇），按每篇的核心贡献生成一个简要摘要表。这个表后来成了我论文第二章方法论综述的骨架：哪些学者在什么时间提出了什么方法，方法之间是什么继承和改进关系。我在此基础上补充了自己的阅读理解和导师的点评，第三稿终于通过了。

12 次不多，但每次都省了不少手动操作。坦率地说，/pyzotero 不是我用得最频繁的功能。12 次会话，比起 `/pdf` 的 100 多次和 `/drawio` 的 104 次，用量少得多。原因是文献梳理这件事不像读 PDF、画图那样天天做，它集中在写文献综述的那几周。但每次用的时候，省下来的手动操作时间很可观。那次拉取 30 篇文献生成摘要表，如果手动做（打开每篇 PDF、找核心贡献、录入表格），至少要半天。Claude Code 直接从 Zotero 读取元数据，加上我之前在 Zotero 里做的笔记标注，十几分钟就出了初稿。

一个需要注意的地方：Zotero 里的信息质量直接决定输出质量。Claude Code 从 Zotero 读取的是你存在里面的元数据：作者、标题、年份、期刊名、标签、笔记。如果你平时往 Zotero 里添加文献的时候很随意（比如标签打得乱七八糟，同一个主题用了「空间计量」「spatial」「空间模型」三个不同的标签），Claude Code 拉出来的结果也会乱。我在用 /pyzotero 之前，专门花了一个下午整理了 Zotero 里的标签体系：把同义标签合并、把无标签的文献补上标签、把几篇标签打错的文献纠正过来。这个整理工作本身和 Claude Code 无关，但它直接影响了后续所有从 Zotero 拉取数据的质量。

另一个需要注意的地方：MCP 读取的是元数据，不是全文。通过 /pyzotero 连接 Zotero，Claude Code 能读到的是文献的元数据信息（标题、作者、摘要、标签、你写的笔记），以及附件列表。它不会自动读取 PDF 全文。如果你想让它读某篇文献的全文内容，还是需要用 `/pdf` 来处理。/pyzotero 的价值在于「帮你从几百篇文献里快速筛选和组织」，而不是「替你读完每篇文献」。

这两个工具配合使用的效果很好。先用 /pyzotero 从 Zotero 里按标签或关键词筛选出一批文献，确定哪些需要精读，然后用 `/pdf` 逐篇读取全文。前者负责「选」，后者负责「读」。

如果你用 Mendeley、EndNote 或其他文献管理工具，目前 Claude Code 没有对应的 MCP 可以直接连接。一个替代方案是把文献信息导出成 BibTeX 文件，让 Claude Code 直接读取 BibTeX。功能上差一些（没法实时同步、没法读取你在文献管理工具里写的笔记），但基本的文献信息查询和分组还是可以做的。

12.3 接入学术数据库：OpenAlex MCP

第四章讲文献检索的时候，用的是 `/arxiv-database` 和 ppw-literature（接入 Semantic Scholar）。这两个工具覆盖了大部分文献检索场景：arXiv 搜预印本，Semantic Scholar 搜已发表的期刊和会议论文。但有一种场景它们处理得不够好：你想看某个研究领域的整体图景，比如某个关键词在过去十年的发表趋势、某篇论文被哪些后续研究引用了、某个作者的所有发表记录和合作关系。

© 2026 晨瀚宇 未经授权禁止转载

OpenAlex 是一个开放学术数据库，收录了超过 2.5 亿篇学术论文的元数据。它和 Semantic Scholar 的区别在于：Semantic Scholar 侧重单篇论文的搜索和引用信息，OpenAlex 侧重整个学术图谱的结构化查询。你可以用 OpenAlex 查某个概念（比如「空间杜宾模型」）在不同年份的发表数量变化，或者查某篇开创性论文的引用树（谁引用了它，引用它的人又被谁引用了）。

Claude Code 通过 MCP 可以连接 OpenAlex。你不需要自己注册 OpenAlex 账号或配置 API，Claude Code 会处理这些。在对话里直接描述你想查什么就行。

我用 OpenAlex 做过的几件事。

第一件事是查研究趋势。文献综述的引言部分，导师要求我写清楚「这个领域近年来的研究热度」。我让 Claude Code 查了一下「spatial econometrics panel data」这个关键词组合在 2015 年到 2025 年的年度发表数量。它从 OpenAlex 拉取数据后告诉我：2015 年每年大约 200 篇，到 2020 年涨到了 400 多篇，2023 年突破了 600 篇。这组数据直接用进了我论文第一章的研究背景部分，用来说明这个领域正处于快速增长期。如果没有 OpenAlex，我得去 Web of Science 手动检索、手动统计，或者引用别人综述里的二手数据。

第二件事是查引用关系。我论文引用了一篇 2018 年的方法论文章，导师问我：「这篇文章之后有没有人质疑过它的方法？」我让 Claude Code 用 OpenAlex 查了这篇文章的被引列表，按引用方式分类（正面引用、方法改进、质疑批评）。它找到了 47 篇引用了这篇文章的后续研究，其中 3 篇提出了明确的方法改进建议，1 篇指出了原方法在小样本下的局限性。我把这 4 篇文献加进了综述，完善了「方法局限性」那段论述。导师后来说这段补得好，「有了对方法局限性的讨论，你的方法选择才更有说服力」。

第三件事是查作者信息。写文献综述的时候，你经常需要知道某个领域的代表性学者是谁、他们的主要贡献是什么。我让 Claude Code 从 OpenAlex 查了「spatial Durbin model」这个领域引用数最高的前 10 位作者和他们的代表作。这个信息帮我确认了一件事：我以为某位学者是这个方法的提出者，实际上他只是最早把这个方法应用到区域经济领域的人，方法本身的提出者是另一位学者。这个错误如果没发现，写进论文里就是一个硬伤。

OpenAlex 和其他文献搜索工具的配合。我的文献工作流程最终形成了这样一个搭配：

先用 OpenAlex 了解领域全景（发表趋势、核心作者、引用网络），确定需要重点关注哪些方向和哪些学者。然后用 `/arxiv-database` 搜最新的预印本（尤其是方法论方面的，很多还没正式发表），用 ppw-literature 搜已发表的期刊论文。搜到的文献存进 Zotero，打好标签。写综述的时候，用 /pyzotero 从 Zotero 里按标签拉取文献，用 `/pdf` 读全文。

表 12.1: 科研文献工作中不同 MCP 和 Skill 的分工

工具	擅长做什么	典型使用场景
OpenAlex MCP	领域全景、发表趋势、引用网络、作者信息	「这个领域近五年的发表趋势」「这篇论文被谁引用了」
/arxiv-database	搜索 arXiv 预印本	「最新的空间计量方法论文章」
ppw-literature	搜索已发表论文，生成 BibTeX	「帮我找关于工具变量的实证研究」
/pyzotero	读取你的 Zotero 文献库	「列出我标记为核心文献的条目」
/pdf	读取单篇 PDF 全文	「读一下这篇论文的方法论部分」
/citation-management	核查和清理 BibTeX	「检查 references.bib 有没有格式问题」

这些工具之间有明确的分工，不存在「装了一个就不需要其他的」的情况。OpenAlex 擅长宏观的结构化查询，但它返回的是元数据，你想看论文具体写了什么，还是得用 `/pdf`。/pyzotero 擅长查你自己的文献库，但它不能帮你搜新文献，搜新文献得用 /arxiv 或 ppw-literature。每个工具解决一个环节的问题，串起来才是完整的文献工作流。

OpenAlex 的局限性。OpenAlex 收录的主要是英文文献。中文期刊论文的覆盖率不高，知网上的很多文章在

© 2026 晨瀚宇 未经授权禁止转载

OpenAlex 里查不到。如果你的研究涉及大量中文文献，OpenAlex 能帮的有限，还是得去知网手动检索。另外，OpenAlex 的被引数据和 Web of Science、Scopus 的数据会有差异（因为数据来源和更新频率不同），写论文的时候如果要引用被引次数，最好以 Web of Science 或 Scopus 为准，OpenAlex 的数据用来做初步筛选就好。

还有一点：OpenAlex 是开放数据库，任何人都可以免费使用，不需要机构订阅。这对没有 Web of Science 或 Scopus 权限的同学来说是一个很好的替代。你在 Claude Code 里直接查就行，不需要 VPN、不需要机构账号。

12.4 实操：用 /pyzotero 同步文献库

这一节带你走一遍完整的操作流程：从安装 /pyzotero 到实际用它从 Zotero 拉取文献、做分类、生成文献梳理框架。你跟着做一遍，就知道该怎么把它用到自己的论文工作里。

前提条件。你需要有一个 Zotero 账号，并且文献库里已经有文献。如果你还没用过 Zotero，先去 zotero.org 注册一个账号、下载桌面客户端、把你的文献导入进去。Zotero 本身的使用不在这本书的范围内，网上有大量教程，几分钟就能上手。

第一步：安装 /pyzotero。在 Claude Code 里输入：

```
帮我安装 pyzotero 这个 skill
```

它会自动下载并放到正确的位置。安装完成后，你可以在对话里输入 /pyzotero 来调用它。

第二步：连接你的 Zotero 账号。第一次使用 /pyzotero 的时候，Claude Code 需要知道你的 Zotero 账号信息才能连接你的文献库。你在对话里说：

```
/pyzotero 连接我的Zotero文献库
```

它会提示你提供两个东西：你的 Zotero 用户 ID 和 API 密钥。用户 ID 在 Zotero 网站的设置页面能看到，API 密钥需要你在设置里创建一个。Claude Code 会告诉你具体去哪个页面操作，你按照它的提示做就行。

这里有一个我当时踩的坑。第一次连接的时候，我创建 API 密钥时选的权限是「只读」（read-only）。对于我的使用场景，只读就够了，因为我只需要从 Zotero 读取文献信息，不需要通过 Claude Code 往 Zotero 里写入任何东西。如果你也只是读取文献，建议选只读权限，这样即使出了什么意外，也不会误删你 Zotero 里的数据。

连接成功后，你可以验证一下：

```
/pyzotero 列出我文献库里最近添加的5篇文献
```

如果它能返回你 Zotero 里的文献信息，说明连接正常。

第三步：按标签筛选文献。这是我用得最多的功能。假设你在 Zotero 里给文献打了标签（比如「核心文献」「方法论」「实证研究」），你可以让 Claude Code 按标签筛选：

```
/pyzotero 列出所有标记了「方法论」标签的文献，
按发表年份从早到晚排序，
每篇显示作者、标题、年份、期刊名。
```

它会返回一个按年份排序的列表。这个列表本身就是文献梳理的原始素材。你可以在此基础上让 Claude Code 做进一步的分析：

```
根据上面这些文献的标题和年份，
帮我按研究方法分成几组（比如面板回归、空间计量、工具变量等），
```


© 2026 晨瀚宇 未经授权禁止转载

```
每组列出属于这一组的文献。
```

分组结果出来之后，你需要自己检查一遍。Claude Code 是根据标题和它已知的信息来分组的，有些文章可能被分错了（尤其是标题里同时提到了多种方法的文章）。我的经验是，大概有 10% 到 15% 的分组需要手动调整。调整完之后，让 Claude Code 把最终的分组结果写进一个文件：

```
把调整后的分组结果写入 literature_groups.md，
每组用二级标题，组名下面列出属于这一组的文献。
```

这个文件就是你写文献综述的骨架。每一组对应综述里的一个小节或一段论述。

第四步：生成文献摘要表。如果你想更进一步，可以让 Claude Code 对每篇文献生成一个简要摘要。这需要结合 Zotero 里的信息和你的笔记：

```
/pyzotero 读取「核心文献」标签下的所有文献，
对每篇列出：作者、年份、标题、你在Zotero里写的笔记（如果有的话）。
然后根据这些信息，为每篇文献写一句话概括其核心贡献。
```

这里有一个要点：Claude Code 生成的「核心贡献概括」是根据标题和你的笔记来推断的，准确度取决于你笔记写得有多详细。如果你在 Zotero 里只存了 PDF、没有写任何笔记，它只能根据标题来猜，准确率会低很多。我的习惯是每读完一篇文献，在 Zotero 里写两三句话的笔记（这篇文章做了什么、用了什么方法、核心发现是什么），有了这些笔记，Claude Code 的概括就靠谱多了。

第五步：持续同步。Zotero 文献库是动态的，你写论文的过程中会不断添加新文献。每次添加了新文献之后，不需要做任何特殊操作，下次在 Claude Code 里调用 /pyzotero 的时候，它会读取最新的数据。MCP 连接的是实时数据，不是某个时间点的快照。

但有一个注意事项：如果你在 Zotero 桌面客户端里做了修改（比如改了标签、更新了笔记），这些修改需要先同步到 Zotero 的服务器上，Claude Code 才能读到。Zotero 桌面客户端默认是自动同步的，但如果你关闭了自动同步，记得手动点一下同步按钮。

完整的工作流示例。最后，我用一个完整的例子把上面的步骤串起来。假设你正在写一篇关于面板数据空间溢出效应的论文，文献综述需要梳理「空间计量模型」相关的文献。

你的操作顺序是这样的：

第一步，在 Zotero 里确认相关文献都打了标签（比如「spatial econometrics」），没有标签的补上。

第二步，在 Claude Code 里调用 /pyzotero，拉取所有带这个标签的文献，按年份排序。看一眼列表，确认没有遗漏（如果发现某篇应该在但没在的文献，回 Zotero 里补标签，同步后重新拉取）。

第三步，让 Claude Code 把这些文献按研究方法分组。检查分组结果，手动调整分错的。

第四步，对每组里的核心文献，让 Claude Code 结合你的 Zotero 笔记生成一句话概括。你逐条检查，改掉不准确的。

第五步，把最终的分组和概括写进一个文件，这就是你文献综述的骨架。接下来你在此基础上补充自己的分析和论述，把骨架变成完整的综述。

整个过程从第二步到第五步，大概需要一到两个小时。其中大部分时间花在检查和手动调整上，Claude Code 执行查询和生成初稿只需要十几分钟。和纯手动做相比（以前我用 Excel 整理同样规模的文献要一整天），效率提升是明显的。

© 2026 晨瀚宇 未经授权禁止转载

注 Claude Code 帮你做的是「整理」和「初稿」，文献综述里的学术判断必须是你自己做的。哪些文献重要、哪些观点之间有争议、你的研究填补了什么空白，这些事情 Claude Code 给不了你答案。它能帮你快速把四百篇文献的信息拉出来、分好组、列成表，但怎么从这些信息里提炼出有价值的综述内容，是你的工作。

工具讲到这里差不多了。下一部分换个角度，聊聊工具背后的思维方式。

© 2026 晨瀚宇 未经授权禁止转载

第四部分

思维方式——工具背后的底层逻辑

© 2026 晨瀚宇 未经授权禁止转载

第十三章 导师教会我的科研写作观

内容提要

- 论文逻辑清晰的三条具体要求：一段一个主题、句子要短、不堆砌从句
- 段落连贯的四种逻辑关系及如何用 Claude Code 扫描逻辑断裂
- 节与节、章与章之间过渡的具体写法
- AI 能帮你执行，但研究问题、学术判断和创新点必须自己提炼
- 先用大白话写清楚想法，再让 Claude Code 帮你改成学术表述

前面十二章讲的都是工具怎么用。这一章换个方向，讲写作本身。

用 Claude Code 写论文大半年，我渐渐意识到一件事：工具再好，如果你自己不知道什么是好的论文写作，它帮你改出来的东西也只是「通顺的废话」。通顺是 AI 的强项，但通顺和清晰之间差着一条鸿沟。这条鸿沟是导师帮我跨过去的。

我的导师开会的时候经常说一些看似朴素的话。比如「你摘要写的太琐碎」「你这个标题古怪」「读者看到这里会不会绕晕」。刚开始我觉得这些只是修改意见，后来才发现它们背后有一套完整的写作逻辑。这套逻辑和 Claude Code 搭配起来，效果比单独用任何一个都好得多：你知道什么是对的，它帮你快速执行。

这一章把导师教我的写作观念整理出来。这些观念不涉及具体工具操作，但它们决定了你用工具的时候往哪个方向使劲。

13.1 论文逻辑要让读者不绕晕

导师批我论文最多的一句话就是「逻辑不清晰」。刚开始我不太明白这四个字到底在说什么。我觉得自己写得很清楚了，每一段都有内容，前后也有过渡句。但导师指着论文说：「你自己读一遍，读到第三段的时候，你还记得第一段在说什么吗？」

我试了一下，确实记不住了。

后来我慢慢总结出导师对「逻辑清晰」的具体要求。归结起来就三条：一段只讲一件事，句子要短，不堆砌从句。听起来简单，做起来不容易。

1. 一段一个主题

导师要求每一段开头用一句话说清楚这段在讲什么，后面所有句子都围绕这一个意思展开。如果写到一半发现在说另一件事了，就该另起一段。

我有一次写第四章的实验结果，一段话里同时说了三件事：评估指标 A 的提升幅度、指标 A 的计算方式、和上一章假设的呼应关系。导师看完说：「这一段太满了，读者读完不知道你最想强调什么。」他让我拆成三段，每段只说一件事。

拆完之后，原来一段 240 字的内容变成了三段共 350 字。字数多了但不是废话，是把每个论点讲得更透了。比如原来「指标 A 从 60 分提升到了 75 分」只有一句话，拆出来之后我多写了一句解释为什么这个提升幅度值得关注（因为基准模型在这个维度上普遍偏低，15 分的提升已经超过了同类方法的平均改进），读者看到这里就能理解这个结果的分量。

© 2026 晨瀚宇 未经授权禁止转载

Claude Code 在这件事上能帮一个忙：你把一段写好的话发给它，让它判断「这段话讲了几个主题」。如果它说讲了两个或三个，你就知道该拆了。我后来习惯在每节写完之后加一步检查：

```
读一下这一节的正文，逐段分析每段讲了几个主题。
如果某段讲了两个以上主题，告诉我建议在哪里断开。
不要改内容，只做分析。
```

这个检查很快，每节两三分钟就能做完，但它能帮你发现自己写的时候注意不到的问题。你写的时候脑子里有完整的思路，觉得把三件事放在一段里很自然；但读者没有你脑子里的思路，他读到第三件事的时候已经忘了第一件事了。

2. 句子要短

导师说过一句话：「如果一句话超过了两行，你自己读一遍，看你能不能一口气读完不换气。如果不能，就断开。」

这个标准很实用。学术写作有一个常见的毛病，就是喜欢写长句。把原因、条件、限定、补充全部塞到一句话里，用逗号和「的」连接。比如：

「由于本文所设计的多模块系统在实际测试中存在模块之间的信息传递损耗且评估维度之间存在较强的相互依赖关系，因此在实验设计中需要同时控制模块间的交互效应和评估维度的耦合影响，以避免因忽略上述因素导致的评估偏差。」

这句话 80 多个字，包含了三层嵌套的逻辑。读者读到「以避免」的时候，已经忘了前面在说什么了。

断开之后：

「系统的多个模块之间存在信息传递损耗。同时，评估维度之间有较强的相互依赖。基于这两个特征，实验设计需要同时控制模块交互效应和维度耦合影响。」

三句话，同样的信息量，但读者可以一句一句消化。每句话只传达一个信息点。

Claude Code 可以帮你拆长句，但你得告诉它拆的标准：

```
读一下第三章第2节，找出所有超过50字的句子，
逐句改短。要求：每句只传达一个信息点，
不要用「由于……因此……」的长句式。
术语保持原样不变。
```

我用这个方法过了一遍第三章，原来平均每句 38 字，改完平均每句 23 字。导师看完说「读着舒服多了」。

3. 不堆砌从句

这个问题和长句有关，但又不完全一样。长句的问题是信息量太大，从句堆砌的问题是修饰层太多。比如：

「为了验证前文所提出的基于方法 A 的系统 B 多模块协同机制假说的鲁棒性……」

这里面有四层修饰：「前文所提出的」修饰「假说」，「基于方法 A 的」修饰「协同机制」，「系统 B 多模块」修饰「协同机制」，最后「鲁棒性」是整个名词短语的中心词。读者要一层层剥洋葱才能找到中心词。

导师说：「你把修饰语拿掉，看看核心意思是什么。如果核心意思只需要一句话，那修饰语就该分到别的句子里去。」

核心意思就是「为了做鲁棒性验证」。前面那些修饰（哪个方法、什么假说、前面哪一章提的）可以在前面的段落里交代清楚，到这一句的时候，读者已经知道上下文了，不需要重复。

改完之后：

「第四章提出了基于方法 A 的协同机制假说。本节对该假说进行鲁棒性验证。」

© 2026 晨瀚宇 未经授权禁止转载

两句话，每句只有一层修饰。读起来一目了然。

这三条要求（一段一个主题、句子要短、不堆砌从句）听起来简单，但我花了很长时间才内化。刚开始我以为导师只是在说风格偏好，后来才明白这是信息传达的基本功。论文的读者是评审专家，他们一天可能要审好几篇论文，没有耐心帮你理清你自己都没理清的逻辑。你写的每句话，他们最多读一遍。如果一遍读不懂，他不会再读第二遍，他会直接写「逻辑不清晰」。

注 这个认识改变了我用 Claude Code 的方式。以前我让它帮我写段落的时候只关注内容对不对，现在会额外加一条要求：「每段只讲一件事，句子不超过 25 字」。加了这条约束之后，它写出来的东西明显比以前清晰，导师退回来重改的次数也少了。

13.2 上下文连贯的具体写法

导师另一个常提的问题是「上下文不连贯」。刚开始我以为连贯就是加过渡句，于是每段结尾都加一句「接下来讨论……」或者「在此基础上……」。导师看了说：「你这些过渡句是空的，读者看完不知道为什么要接着往下读。」

后来我理解了：连贯不是靠一句话实现的，是靠前后段落之间的逻辑关系实现的。过渡句只是这种逻辑关系的外在表现，如果逻辑关系本身不成立，过渡句加多少都没用。

1. 段落之间的四种逻辑关系

导师让我读几篇写得好的论文，分析它们段落之间是怎么连接的。我读完发现，段落之间常见的逻辑关系就四种：递进（前面说了 A，这段进一步说 B）、转折（前面说了 A，但有个问题 C）、并列（除了 A，还有 B）、因果（因为 A，所以 B）。

每次写完一段准备开始下一段的时候，先问自己：这一段和上一段是什么关系？如果说不清楚是哪种关系，说明这两段之间缺了一环逻辑，要么需要补一段过渡，要么需要调整顺序。

举个例子。我第三章有两段，第一段说「基准模型在五个评估维度上的测试结果如表 3-1 所示」，第二段直接跳到「本文采用方法 A 对系统进行优化」。导师问我：从测试结果到优化方案，中间发生了什么？为什么看完测试结果你就决定用方法 A 了？

中间缺的那一环是：基准测试揭示了指标 X 和指标 Y 两个维度得分偏低，而方法 A 恰好针对这两个维度有优势，所以选择方法 A 作为第一步优化。

补上这一段之后，读者就能跟上思路了。原来读者看到那里会想「怎么突然跳到优化了」，现在他看完中间这段就知道「是因为测试暴露了这两个短板，所以选了专门解决这两个问题的方法」。

这种逻辑断裂，自己写的时候很难发现。因为你脑子里有完整的推理链条，但你只把结论写到了纸上，中间的推理过程省略了。你觉得读者能跟上，实际上读者跟不上。

Claude Code 可以帮你扫描这种断裂。我后来让它做过一次全文的逻辑连贯性检查：

```
读一下第三章到第五章的全文。
逐段检查相邻两段之间的逻辑关系，标注是递进、
转折、并列还是因果。如果某两段之间你判断不出
是什么关系，就标注为「可能断裂」。
列出所有「可能断裂」的位置，不要改内容。
```

它标出了 11 处「可能断裂」。我逐个看了，其中 6 处确实有问题（缺了一个逻辑环节），5 处是正常的（比如换了小节主题）。6 处问题里有 4 处是我之前完全没注意到的。补上之后，导师说这一版「读起来顺多了」。

© 2026 晨瀚宇 未经授权禁止转载

2. 每段第一句承接上文

导师教我一个很实用的技巧：每段的第一句话，最好带着上一段的某个关键词或关键概念。这样读者从上一段读到这一段的时候，有一个抓手。

比如上一段的最后一句是「基准测试显示，系统在指标 X 维度上得分偏低」。下一段的第一句可以是「指标 X 偏低的主要原因是模块间的信息传递存在损耗」。「指标 X 偏低」是从上一段继承来的概念，读者看到这个词就知道还在接着说上一段的事。

如果下一段的第一句是「本文采用方法 A 对系统进行优化」，虽然意思对，但读者要自己去建立「指标 X 偏低」和「方法 A」之间的联系。有些读者能反应过来，有些就跟丢了。

这个技巧我后来也用 Claude Code 来检查：

```
读一下第四章全文。逐段检查：
每段第一句是否包含上一段的关键概念。
如果不包含，判断前后两段是否有逻辑断裂。
列出所有你认为可能断裂的位置，不要改内容。
```

它列出来的位置，大约有一半确实需要调整。另一半是正常的主题切换（比如小节之间的分界）。能帮你发现一半的断裂点，已经很有用了。

3. 节与节之间的衔接

段落之间的连贯是微观的，节与节之间的衔接是宏观的。一节讲完了一个话题，下一节要讲另一个话题，两节之间怎么过渡?

导师的做法是：上一节的最后一段点出一个「还没解决的问题」，下一节的第一段正好回应这个问题。这样读者就自然知道为什么要读下一节。

比如我第四章，4.1 节讲的是「方法 A 的优化结果」，4.2 节讲的是「鲁棒性验证」。4.1 的最后一段我原来写的是「以上是方法 A 的优化结果，下面进行鲁棒性验证」。导师说这是「报菜名」，读者知道你要做鲁棒性验证了，但不知道为什么要做。

改成：「方法 A 优化后指标 X 提升了 15 分，但这一提升是否稳定还需要验证。如果换一组测试数据、或者调整模块参数，提升效果是否还成立?」

这样读者看完就有一个疑问（结论稳不稳健），自然想往下读，而 4.2 的第一段正好回答这个疑问。两节之间的过渡就完成了，不需要额外加一句「下面进行稳健性检验」。

我把这个技巧总结成了一个公式：上一节末尾 = 上一节的结论 + 一个未解决的问题。下一节开头 = 回应这个问题 + 本节要做什么。用 Claude Code 的时候，你可以让它帮你检查每节的末尾是不是符合这个模式。

4. 章与章之间的过渡

更大尺度的过渡是章与章之间的。每一章的开头几句话，应该快速回顾上一章做了什么、指出接下来要做什么、说清楚为什么要做。

我第五章的开头原来只写了一句「本章对系统进行进一步优化」。导师说：「第四章不也在优化吗？读者分不清这两章有什么区别。」

改成：「第四章通过方法 A 提升了指标 X 和指标 Y，但指标 Z 仍然偏低。本章引入方法 B，专门针对指标 Z 的短板进行优化。」

这样读者一看就知道：第四章解决了 X 和 Y 的问题，第五章解决 Z 的问题。两章之间是递进关系，每章都在回答一个独立的问题。

© 2026 晨瀚宇 未经授权禁止转载

这些连贯的技巧，Claude Code 可以帮你执行，但方向要你自己定。你知道两段之间缺了什么逻辑，让它帮你补；你知道章节开头应该回顾什么，让它帮你写。如果你自己不知道缺了什么，让它帮你写过渡句，它写出来的通常是空洞的套话（「在此基础上，本章进一步讨论……」）。过渡句的质量取决于你对上下文逻辑关系的理解，这个理解只能来自你自己。

导师还有一个观察我觉得很准：好论文读起来像顺着一条河往下走，每一段都顺着前一段自然流下来；差论文读起来像跳石头过河，每一段都要跳一下才能到。区别就在于段落之间有没有铺好逻辑的路。

13.3 AI 能帮你写，但不能帮你想

导师有一次开会说了一句话，我当时没太在意，后来越想越觉得重要：「做了工作不代表创新，你做的东西要提炼。」

这句话的背景是我在论文里写了很多自己做了什么：用了什么方法、跑了什么实验、得到了什么结果。导师看完说，你做了这些我知道了，但你的创新点在哪里？「做工作」和「创新」之间差了一步提炼：你做的这些事，解决了什么别人没解决的问题？推进了什么认知？

这一步提炼，是 AI 帮不了你的。

研究问题必须是你自己的

我写第一章「研究问题」的时候，有一次偷懒让 Claude Code 帮我写。我给了它研究背景，让它帮我提出研究问题。它写了三个问题，格式标准，看起来也合理。我读了一遍觉得不错，差点就直接用了。

后来导师问我：「这三个问题是你自己想出来的还是 AI 帮你写的？」我老实说了。导师说：「你看你的第二个问题，' 系统 A 是否能提升审计效率'。这个问题已经有人验证过了，你如果还在问' 是否能'，你的论文就是在重复别人的工作。你应该问的是' 在新的场景和新的方法下，系统 A 在哪些维度上有优势、在哪些维度上仍然不足、原因是什么'。」

两个问题之间的区别，看起来只是几个字。但第一个问题能写出来的论文是一篇验证性研究（别人做过了你再做一遍），第二个问题能写出来的是一篇推进性研究（在别人的基础上往前走了一步）。这种学术判断，是你读了几十篇文献、听了多次导师开会、对这个领域浸泡了足够长时间才能有的。AI 读不了你的领域积累，也做不了这种判断。

学术判断不能外包

除了研究问题，还有一些环节是不能交给 AI 的：你的论文用什么方法、为什么用这个方法、为什么不用那个方法、怎么解释实验结果、负面结果说明了什么。这些都是学术判断。

我有过一个教训。跑完实验之后，系统在某个评估维度上的表现没有提升。我让 Claude Code 帮我写这段结果的讨论。它写了一段解释，说「该维度未提升可能是因为测试样本数量不足，导致结果不具有统计显著性」。这个解释逻辑上说得通，但放在我的论文里不对，因为我的测试集有好几百条数据，不算小。真正的原因是这个维度的评估本身依赖外部知识库，而我的系统没有接入相关的知识源。

如果我没有检查就用了 AI 的解释，评审一看就会问：你几百条测试数据还说样本不足？这就暴露问题了。AI 不了解你的系统和数据的具体情况，它给出的是一个「一般性」的解释，而论文需要的是「针对你的系统」的解释。

从那之后我给自己定了一条规矩：凡是涉及「为什么」的学术判断，Claude Code 写的初稿我都不直接用，先自己想清楚原因，然后让它帮我把我的解释改成学术表述。

创新点要自己提炼

© 2026 晨瀚宇 未经授权禁止转载

导师说的「做了工作不代表创新」这句话，我后来反复琢磨。在我的论文里，我做了三件事：设计了一个新的多模块协同架构、在一个新的应用场景下验证了系统的有效性、针对几个低分维度分别提出了优化方案。这三件事，哪些算创新、哪些只是工作量？

导师帮我梳理了。第一件事，如果新架构比已有方案有明显的优势（更准确、能处理旧方案处理不了的场景），那就是方法创新。第二件事，新场景本身不算创新，但如果在新场景下发现了和已有结论不同的规律，那就是发现上的创新。第三件事，针对低分维度的优化方案，如果能说清楚为什么这个维度难、你的方法为什么能解决，那就是认知上的创新。

这个梳理过程，每一步都需要你对文献的深入了解。「新架构比旧方案好在哪里」，你得读过用旧方案的文献才能说清楚。「新场景揭示了不同的规律」，你得知道已有结论是什么才能对比。「低分维度为什么难」，你得有一个理论框架来解释。

Claude Code 能帮你把已经想清楚的创新点写成流畅的段落，能帮你检查表述是否准确。但想清楚这些创新点本身是什么，是你的事。

我后来养成了一个习惯：每次让 Claude Code 帮我写涉及学术判断的段落之前，先自己用大白话写一段，说清楚「我想表达什么」。哪怕写得粗糙、口语化都没关系。然后把这段大白话作为材料给 Claude Code，让它帮我改成学术表述。这样核心判断是我的，它只负责打磨语言。

```
我要写第一章的创新点。我的大白话版本：
「这篇论文有三个新的地方。第一，……第二，……第三，……」

请帮我把这段话改成学术论文的表述方式。
要求：保留我的核心论点不变，只调整语言风格。
不要添加我没有提到的内容。
```

这个方法看起来多了一步（先自己写大白话再让它改），但效果比直接让它写好得多。因为你自己写的大白话里，包含了你真正想说的东西、你的侧重点、你的逻辑。AI 帮你换了一套表达方式，但内核是你的。

注 导师说过一句让我印象深刻的话：「你用什么工具不重要，你的脑子里有没有东西才重要。脑子里有东西，用什么工具都能写出好论文。脑子里没东西，工具越好，暴露得越快。」

这句话我写在了 CLAUDE.md 的开头。每次打开 Claude Code 的时候看一眼，提醒自己先想清楚再动手。

总结起来，导师教我的这些写作观念可以归结为一句话：你要先知道什么是好的，再用工具去实现。顺序不能反。如果你先学了工具用法，但不知道什么是好的论文逻辑、好的段落衔接、好的创新点表述，工具只会帮你更快地生产平庸的文字。知道什么是好的之后，工具帮你省的是从「好的想法」到「好的文字」之间的执行时间。

© 2026 晨瀚宇 未经授权禁止转载

第十四章 与 AI 协作的正确心态

内容提要

- 明确人机分工边界：哪些任务适合交给 AI，哪些必须自己做
- 四类最容易出问题的环节：文献评述、数据描述、结果解释、引用匹配
- 建立检查习惯，避免 AI 生成内容中的隐蔽错误
- 收到导师意见后先自己想清楚，再用具体指令驱动 AI 执行
- 持续优化指令模板和 CLAUDE.md 规则，提升长期协作效率

用了大半年 Claude Code 之后，我对「AI 辅助写论文」这件事的理解和刚开始的时候完全不一样了。刚开始我觉得它就是一个更高级的写作助手，像 Grammarly 但更厉害。后来发现它能做的事远超我的预期，从跨文件检查到画图到格式转换，几乎什么都能干。再后来又发现，它能做的越多，你需要管的也越多。能力越大，出错的方式也越多。

这一章讲的是用了大半年之后，我对人和 AI 之间分工边界的理解。

14.1 你是主导，AI 是执行

这个道理说起来简单，做起来很容易模糊。特别是当 AI 帮你做了一件你以为会很麻烦的事、结果几分钟就搞定的时候，你会不自觉地越来越依赖它的判断。

我有一个真实的例子。有一次我让 Claude Code 帮我写文献综述里的一段评述。我没有给它具体的论点，只说了「帮我评述一下这几篇文献的研究方法」。它写了一大段，结构清晰、语言流畅，还指出了几篇文献的方法局限性。我读了一遍觉得不错，就准备直接用了。

后来我仔细看了一遍它写的「方法局限性」。其中有一条说某篇文献「没有控制时间固定效应」。我回去翻了那篇原文，人家用的是一阶差分法，本身就等价于控制了时间趋势。Claude Code 的评述听起来合理，但实际上是错的。如果我没有回去查原文就用了这段话，等评审指出来的时候就很被动了。

从那之后我给自己定了一条规矩：Claude Code 写的任何涉及学术评价的内容（好的、差的、创新的、不足的），我都要回到原始文献里验证一遍。它可以帮我写初稿，但哪些话留下、哪些话删掉、哪些话需要改，决定权在我。

分工的实际操作

经过这么多次踩坑，我总结出了一个适合自己的分工模式。

适合交给 Claude Code 做的事：术语统一、格式排版、拆分长句、检查前后一致性、查找引用缺失、生成表格和图表、把大白话改成学术表述、把导师的笼统批注翻译成具体的修改。这些事有明确的标准，做对做错一目了然。

必须自己做的事：确定研究问题、选择研究方法、解释实证结果、判断创新点、回应评审意见中的学术质疑、确认文献的评述是否准确。这些事需要你对领域的深入理解，没有标准答案。

处于中间地带的事：章节框架的设计、过渡段落的写法、摘要的提炼。这些事可以让 Claude Code 提建议、做初稿，但最终的判断要你做。

© 2026 晨瀚宇 未经授权禁止转载

表 14.1: 人和 AI 在论文写作中的分工

交给 AI	中间地带	必须自己做
术语统一、格式排版	章节框架设计	研究问题和方法选择
拆长句、查一致性	过渡段落写法	实证结果解释
引用格式核查	摘要初稿	创新点提炼
大白话改成学术表述	文献评述初稿	学术评价验证
生成图表	导师意见执行方案	回应评审质疑

注 这张表不是固定的。随着你对工具的熟悉程度增加，有些「中间地带」的事可以慢慢交给 AI 做更多。但「必须自己做」那一列，不管你用了多久、多熟练，都不应该交出去。

不要被效率惯坏了

一个需要警惕的心理陷阱是：因为 AI 做事太快，你开始跳过「自己先想清楚」这一步。

举个例子。导师说「你第三章的模型设定需要改一下」。以前的流程是：我先想清楚要怎么改、改哪里、为什么这么改，然后自己动手改。现在有了 Claude Code，很容易变成：我把导师的原话直接贴给它，说「按这个意见帮我改」。

问题是导师的意见往往是高度概括的（「逻辑不清晰」「模型设定有问题」），需要你自己判断具体怎么改。如果你跳过思考直接让 AI 改，它会按照自己的理解来改。它的理解可能对，也可能不对，但不管对不对，你都丧失了一次深入思考自己论文的机会。

我后来的做法是：收到导师意见后，先自己花十分钟想清楚每条意见具体要改什么、怎么改、为什么。想清楚之后，把我的改法写成具体指令交给 Claude Code 执行。这十分钟很重要，它保证了改出来的东西是我想要的，而且我真正理解了为什么这么改。

注 收到导师的批注时，不要直接把导师的原话丢给 AI。导师的意见往往是概括性的（「逻辑不清晰」「结论太弱」），需要你自己判断具体怎么改。先花十分钟想清楚每条意见的具体含义，然后把你的理解写成清晰的执行指令交给 Claude Code。这十分钟不是浪费，它保证了改出来的东西是你真正理解并认可的。

这里有一个具体的例子。导师有一次说「第五章的结论太弱了」。如果我直接把这句话交给 Claude Code，它可能会把结论写得很「强」，加上一些夸大的表述。但导师说「太弱」的意思是：结论没有把本文的边际贡献和已有研究做对比，读者看完不知道你的发现比别人多了什么。

想清楚这个之后，我给 Claude Code 的指令是：

```
修改第五章结论段。在每个核心发现之后，
补充一句和已有研究的对比
（比如「xxx研究得到的系数为0.2，本文的系数为0.34，
可能的原因是本文使用了更新的数据和更精确的方法」）。
以下文献可以用于对比：[列出3-4篇]
不要夸大本文的贡献，用客观的比较语气。
```

改出来的效果比直接说「把结论写强一点」好得多。因为我想清楚了「强」的具体含义，Claude Code 只需要执行。

14.2 对结果负责的永远是你

这一节讲一个敏感但必须正面谈的话题：用 AI 辅助写论文的学术诚信问题。

© 2026 晨瀚宇 未经授权禁止转载

先说结论：用 AI 辅助写论文不违反学术规范。前提是你的研究问题、研究方法、实验数据、核心结论都是真实的、你自己做的。AI 帮你打磨的是表达方式，就像你请同门师兄帮你看一遍论文、提修改建议一样，是正常的学术互助。

但有一条红线不能碰：AI 生成的任何内容，在你用之前，你都要理解它、验证它、对它负责。一旦写进论文，那就是你的话。如果评审问你「这段话的依据是什么」，你不能说「AI 写的我没细看」。

四个最容易出问题的环节

第一个是文献评述。Claude Code 可能会编造文献的观点，特别是当你让它评述它没读过的文献时。它会根据文献标题和你给的信息「推测」内容，写出来看起来合理但可能和原文不符。每一句对文献的评述，都要回到原文核实。

我遇到过一个具体的例子。让它帮我写一段关于空间计量方法演进的综述，它提到某篇 2019 年的论文「首次将空间杜宾模型应用于区域创新溢出研究」。我去查了那篇论文，人家用的是空间滞后模型，不是空间杜宾模型。两个模型虽然都属于空间计量范畴，但有本质区别（空间杜宾模型多了自变量的空间滞后项）。这种错误如果没发现，在答辩的时候被评审揪出来就很难堪了。

第二个是数据描述。如果你让它描述数据的特征，它有时候会编造具体数字。比如你的数据有 300 个样本，它可能写成「本文收集了约 300 个样本，时间跨度为 2015 至 2021 年」，里面的「2015 至 2021 年」是它猜的。所有涉及数据的描述，数字必须来自你的真实数据。

第三个是结果解释。上一节已经举过控制变量不显著的例子了。AI 给的解释是通用性的，你的论文需要的是针对性的。通用解释放到你的论文里，如果恰好和你的数据特征矛盾，就会暴露问题。

第四个是引用匹配。Claude Code 可能会给你加一个「看起来很合适」的引用，但那个引用实际上并不支持你的论点。或者更糟的是，那篇文献讲的是完全不相关的主题。我遇到过一次，它在一段话后面加了一个参考文献编号，我去查那篇文献，发现讲的根本不是同一件事。

建立检查习惯

基于这些经验，我给自己建立了一套检查流程。

每次 Claude Code 帮我写完或改完一段内容，我做三件事。第一，对比改前改后，确认它没有「顺手」改掉不该改的地方（这个问题在第三章「术语保护」那一节详细讲过）。第二，对于涉及学术判断的内容（文献评述、结果解释、方法比较），回到原始材料核实。第三，对于新增的引用，去查原文确认引用是否准确。

这三步加起来大概多花十分钟。但这十分钟能帮你避免答辩的时候被问住。

还有一个实用的建议：在 CLAUDE.md 里加一条规则，让 Claude Code 在写涉及具体数据和文献的内容时，标注信息来源。

```
当你写涉及数据、文献或实证结果的段落时，
如果某句话的信息不是来自我提供的材料，
请在句末标注[需核实]。
```

这条规则不能百分之百保证它会标注所有推测内容，但能提醒它区分「有依据」和「没依据」，减少它默默编造的概率。更重要的是，它帮你养成了一个意识：AI 写的每句话，都需要你确认来源。

学校对 AI 使用的态度

目前国内高校对 AI 辅助论文写作的政策还在不断变化中。有些学校明确允许，有些还没有明确表态。建议你做两件事：第一，了解一下你所在学校和学院的具体政策；第二，不管政策怎么规定，确保你论文里的核心学术内容（研究问题、方法、数据、结论）都是你自己的真实工作。

© 2026 晨瀚宇 未经授权禁止转载

我个人的做法是在致谢里写了一句话，说明论文写作过程中使用了 AI 辅助工具，用于格式调整和语言润色。这个表述既诚实又准确，因为核心学术工作确实是我自己做的，AI 帮的就是格式和语言。

14.3 怎么持续提高协作效率

用了大半年 Claude Code，我的使用方式和刚开始的时候差别很大。刚开始是每次从零开始，想到什么让它做什么。后来慢慢形成了一套固定的流程和模板，效率越来越高。

这一节讲几个我觉得最有效的迭代优化方法。

1. 把好的指令存成模板

刚开始用的时候，每次都要从头写指令，想着该加什么约束、该保护哪些术语、该用什么格式。后来我发现，很多任务是重复的。改章节的指令、检查一致性的指令、润色摘要的指令，结构都差不多，只是具体内容不同。

于是我把这些指令整理成了模板，存在一个文本文件里。每次做同类任务的时候，打开模板，把具体内容填进去就好了。比如「改章节」的模板：

```
修改 [文件路径] 的第 [X] 节。
要求：
- 不改变论点
- 用学术中文
- 以下术语不得修改：[术语列表]
- 完成后给我看修改前后的对比

先告诉我你的方案，确认后再执行。
```

每次用的时候，填一下文件路径、节编号、术语列表，30 秒就能发出去。比从头写一条完整的指令快多了，而且不会漏掉关键约束。

附录 A 里列出了我常用的提示词模板，你可以直接拿去用。

2. 从每次错误中更新 CLAUDE.md

Claude Code 犯过的错，我都会想一下：这个错是偶然的，还是因为我的 CLAUDE.md 里缺了什么规则？

如果是缺规则导致的，就补上。比如第一章提到的术语被篡改的问题，我就在 CLAUDE.md 的术语锁定表里加了对应的条目。它有一次改论文的时候顺手删了一个引用编号，我就加了一条「修改正文时保留所有引用编号不变」。它有一次把图表标题的格式改了，我就加了一条「图表标题格式为：图 X-Y：标题内容」。

CLAUDE.md 是一个不断生长的文件。我刚开始用的时候只有十几行，半年后变成了一百多行。每一条规则背后都是一次具体的踩坑经历。规则越细，它犯同样错误的概率就越低。

3. 定期回顾对话记录

Claude Code 的每次对话都有记录。我大概每两周会花半小时翻一下最近的对话，看看有没有什么规律。

比如我发现，凡是我在指令里没有说「不要改术语」的时候，它大概有三成概率会改掉一两个术语。这让我意识到「不要改术语」这句话不能省，哪怕 CLAUDE.md 里已经写了。于是我把它加进了每个模板。

再比如我发现，涉及图表的任务成功率比纯文字任务低。原因是图表任务需要指定格式、配色、字号、字体，而我给的指令往往只说了「画一个图」没给细节。发现这个规律之后，我专门做了一个「图表任务模板」，把格式要求都写进去了，成功率明显提高。

4. 控制单次会话的范围

© 2026 晨瀚宇 未经授权禁止转载

一次会话只做一件事（原因和判断标准见第三章）。简单的判断方法：如果你描述任务时用了两个以上的「然后」，就该拆成两次会话。

5. 给长任务设检查点

做不完的大任务（比如全文引用核查），要在指令里要求它用 checkpoint.md 记录进度（具体写法和完整案例见第五章和第十章）。核心思路是：只要进度保存在文件里，会话中断后用 `claude --continue` 恢复，它能从断点继续。

6. 建立反馈闭环

最后一点：每做完一个任务，花一分钟想想哪里做得好、哪里可以改进。这个习惯听起来简单，但大多数人（包括我自己一开始）不会主动做。

具体来说就三个问题：这次任务的指令写得够清楚吗？Claude Code 的输出有没有需要我返工的地方？下次做同类任务，指令需要调整什么？

如果指令写得好、输出也好，说明这个模板已经成熟了。如果指令写了但输出还是有问题，想想是指令的问题还是任务本身的难度问题。如果是指令的问题（比如漏了某个约束），更新模板。如果是任务难度问题（比如 AI 确实做不好某类学术判断），以后这类任务就别交给它了，自己做。

协作效率的提升靠的是每次踩坑之后那一个小调整，不是某一次大的改进。把好的做法固化成模板和规则，把坏的经历变成防线，时间长了，你和 AI 之间的协作就越来越顺畅了。

© 2026 晨瀚宇 未经授权禁止转载

第十五章 从一篇论文到科研工作习惯

内容提要

- 将 CLAUDE.md、Memory 和指令模板整理成可复用的个人知识库
- 提取通用规则，制作项目模板，让每个新项目站在上一个项目的肩膀上
- 通过 Skill 系统和提示词模板实现跨项目、跨团队的经验复用
- 维护「教训总结」文件，用完整故事记录踩坑经验
- 展望 AI 辅助科研的未来方向，坚守学术诚信的底线

到这里，这本书的主要内容讲完了。前面十四章，从装工具到写论文到思维方式，走了一遍完整的流程。

但你写完这篇论文之后呢？如果你继续读博、或者工作之后还要做研究，这些经验能不能带到下一个项目？我自己的体会是：一篇论文积累下来的配置、模板和工作习惯，如果整理好，能给后续的项目省很多事。

这一章讲怎么把一次性的经验变成可复用的资产。

15.1 建立个人知识库

用 Claude Code 做完一篇论文之后，你手上有三样东西很有价值：CLAUDE.md、Memory 里存的记忆、以及你在对话过程中积累的指令模板和踩坑记录。把这些东西整理好，就是你的个人知识库。

定义 15.1 (个人知识库)

个人知识库是指研究者在使用 Claude Code 过程中积累的可复用资产集合，包括 CLAUDE.md（规则与规范）、Memory（跨会话经验）、指令模板（常用提示词）和踩坑记录（教训总结）。它的核心价值在于让每个新项目不从零开始，而是站在之前积累的基础上。

CLAUDE.md：你的科研工作手册

CLAUDE.md 是一个越用越厚的文件。第二章讲了怎么创建它，到论文写完的时候，我的 CLAUDE.md 已经从最初的十几行长到了一百多行。里面有术语锁定表、格式规范、导师偏好、文件结构说明、常见错误的防线。

这份文件对下一个项目的价值在于：它记录了你在一个完整项目中踩过的所有坑。下一个项目开始的时候，你不需要重新踩一遍。

具体的做法是：论文完成后，把 CLAUDE.md 里的规则分成两类。一类是项目特定的（比如某篇论文的术语锁定表、特定的文件结构），下一个项目用不上。另一类是通用的（比如「修改正文时保留所有引用编号」「改前先确认方案」「图表标题格式」），每个项目都用得上。

把通用规则提取出来，做成一个「CLAUDE.md 模板」。下次新建项目的时候，先复制这个模板，再补充项目特定的内容。我的模板附在附录 D 里，你可以直接拿去用。

Memory：跨项目的经验传承

Memory 系统（第二章讲过）存的是跨会话的记忆。但它也可以存跨项目的经验。比如你在论文 A 项目中发现「LaTeX 比 DOCX 稳定得多」，把这条经验存进 Memory，到了论文 B 项目的时候，Claude Code 还记得这件事，会优先建议你用 LaTeX。

© 2026 晨瀚宇 未经授权禁止转载

我的 Memory 里目前存了大约三十条经验。有些是工具层面的（「用 Draw.io 画图」「DOCX 操作前必须备份」），有些是写作层面的（「导师要求一段一个主题」「句子不超过两行」），有些是流程层面的（「一个 session 只做一件事」「长任务要设检查点」）。

这三十条经验是大半年、一千多次会话积累下来的。如果你直接把这些经验的核心写进自己的 Memory 和 CLAUDE.md，相当于继承了半年的踩坑成果，起点就高了不少。

项目模板

除了 CLAUDE.md 和 Memory，还有一个更大的东西可以复用：整个项目的文件结构。

我论文项目的文件夹结构大致是这样的：chapters 文件夹放各章正文、tables 文件夹放所有表格文件、figures 文件夹放图、references 文件夹放 BibTeX 文件和参考文献 PDF。这个结构在 CLAUDE.md 里有说明，Claude Code 每次做任务都知道去哪里找文件。

下一个项目如果沿用类似的结构，CLAUDE.md 里的文件路径说明就可以直接复用，不用重新写。我后来把这个文件夹结构也做成了模板。新建项目的时候，复制模板文件夹，改一下 CLAUDE.md 里的项目名和术语表就能开工。

整理这些东西大概需要两三个小时。但这两三个小时的整理，可以给你后续每个项目各省十几个小时的配置时间。越到后面项目越多，这个投入产出比就越高。

15.2 跨项目复用经验

个人知识库解决的是「你自己」的经验传承。Skill 系统和提示词模板解决的是更大范围的复用：你的经验可以帮到别人，别人的经验也可以帮到你。

Skill 的复用

第九章讲了 Skill 的用法。如果你在一个项目中为了某类任务写了一个 Skill，它在其他项目里也能直接用。Skill 是全局安装的，不绑定在某个项目文件夹里。

比如你写了一个「论文润色」的 Skill，里面定义了术语保护规则、语言风格要求、检查清单。这个 Skill 你写一次，以后不管写什么论文，都可以直接调用。你还可以把 Skill 分享给同组的师弟师妹，让他们不用从零开始积累。

我自己常用的 Skill 有三个：一个是论文润色用的，定义了学术中文的语言规范；一个是引用核查用的，定义了 GB/T 7714 的格式标准和核查流程；一个是图表制作用的，定义了配色方案、字号规范、导出格式。这三个 Skill 覆盖了论文写作中大部分的重复性任务。

附录 C 里列出了我推荐的科研 Skill 清单，包括每个 Skill 的名称和一句话说明。你可以按需安装。

提示词模板的复用

第三章讲了提示词的实战经验。好的提示词模板是可以跨项目使用的。我的模板库目前有十几个模板，按任务类型分类：改章节的、查引用的、画图的、润色摘要的、检查一致性的。

这些模板存在一个文本文件里，需要的时候打开文件、复制模板、填入具体内容。比「每次从头写指令」省了时间，也避免了遗漏关键约束。

模板的维护有一个原则：每次发现一个模板不够用（比如少了某个约束），就更新一下。不要等到攒了一堆问题才集中修改。每次修一小处，模板就慢慢变得越来越好用。

附录 A 列出了我的常用提示词模板，按场景分类，你可以直接拿去改。

经验文档的整理

© 2026 晨瀚宇 未经授权禁止转载

除了 CLAUDE.md 和模板，我还维护了一份「教训总结」的文件。每次遇到一个值得记住的经验（不管好的还是坏的），就花两分钟写下来：发生了什么、为什么会这样、下次怎么避免。

这份文件半年下来有十三条教训。每条都能展开讲一个完整的故事。它对我自己的价值是：每次遇到类似的情况，先翻翻这个文件，看看以前有没有踩过同样的坑。它对别人的价值是：你可以把它分享给其他使用 Claude Code 的同学，帮他们避开你已经踩过的坑。

写这份教训总结的时候有一个技巧：不要只写结论（「要先备份再改」），要写完整的故事（「有一次我让它改 DOCX，它直接覆盖了原文件，导师给的批注全没了。从那之后每次改 DOCX 之前都先备份」）。有故事的教训记得住，没故事的教训过两天就忘了。

给团队的建议

如果你在一个课题组里，组里有好几个学生都在用 Claude Code，那复用的效果会更明显。

一个做法是：课题组维护一份公共的 CLAUDE.md 模板和一个公共的 Skill 库。每个学生在自己的项目中发现的好规则、好模板，提交到公共库里。新来的学生拉一下公共库，就继承了整个课题组的积累。

我们组目前还没有做到这个程度（只有我一个人在系统性地用 Claude Code），但我觉得这是一个值得探索的方向。特别是对那些组里年年有学生毕业、年年有新学生进来的课题组来说，知识的传承一直是个问题。以前靠「师兄带师弟」口头传授，容易遗漏。如果把这些经验固化到 CLAUDE.md 和 Skill 里，传承就有了物质载体。

15.3 未来展望

AI 辅助科研写作这件事，在我写这本书的时候（2026 年春天）还处在很早期的阶段。工具在快速迭代，每隔几个月就会有新的功能出来。我用 Claude Code 的这大半年里，它自己就更新了很多次，加了并行 Agent、加了 Hooks、加了更多 MCP 工具。

有几个方向我觉得值得期待。

一是上下文窗口会继续变大。现在 Claude Code 能处理的上下文已经够用了（一次读几万字的论文没问题），但如果以后能同时处理整个项目的所有文件加上所有参考文献，那跨文件一致性检查会更准确、更省事。

二是工具集成会更深入。现在你要让 Claude Code 调用 Zotero、调用学术数据库、调用图表工具，还需要配 MCP 或装 Skill。以后这些可能会变成开箱即用的功能，不需要额外配置。

三是多人协作的场景。现在 Claude Code 主要是一个人用。如果以后支持团队共享 CLAUDE.md、共享 Skill、共享 Memory，一个课题组的积累就能更方便地传承给下一届的学生了。

注 不管工具怎么变，有些东西不会变：研究问题要你自己想，数据要你自己跑，结论要你自己下，学术诚信的底线不能碰。工具越强大，你越需要清楚自己在做什么、为什么这么做。

导师在我答辩前说过一句话：「工具是手段，把研究成果写清楚、讲明白，才是目的。」这句话我写在了这本书的前言里，现在放在结尾，也合适。

希望这本书里的经验能帮你少走一些弯路。祝你的论文写作顺利。

© 2026 晨瀚宇 未经授权禁止转载

附录 A 常用提示词模板大全

这里整理了论文写作中最常用的提示词模板，按场景分类。每个模板里的方括号部分需要你根据实际情况填写。

场景一：章节润色

```
修改 [文件路径] 的第 [X] 节。
要求：
- 不改变论点
- 用学术中文，不要口语化表达
- 以下术语不得修改：[术语列表]
- 每段只讲一件事，句子不超过25字
- 保留所有引用编号
- 完成后给我看修改前后的对比

先告诉我你的方案，确认后再执行。
```

这个模板适用于大部分章节修改任务。关键点是术语锁定和「先方案后执行」。

场景二：一致性检查

```
读一下 [文件夹路径] 里的所有 [文件类型] 文件。
检查以下内容在不同文件之间是否一致：
- 术语使用（特别是：[列出容易混用的术语对]）
- 变量名和缩写
- 图表编号的引用

列出所有不一致的地方，标明在哪个文件的哪一段。
不要改内容，只做检查。
```

我每次做完一轮修改之后都会跑一次一致性检查。特别是改了某个术语之后，要确认全文都改到了。

场景三：摘要润色

```
润色 [摘要文件路径]。
要求：
- 去掉具体数值，改为定性概括
- 保留所有引用编号
- 术语保持原样不变：[术语列表]
- 语言要有高度，不要太琐碎
- 字数控制在 [X] 字以内
```

导师说过「摘要不要放具体数字，要拔高一点」。这个模板就是把导师的要求翻译成了 AI 能执行的指令。

场景四：翻译（中译英）

```
把以下中文段落翻译成英文。
```


© 2026 晨瀚宇 未经授权禁止转载

```
要求：
- 学术英语，适合国际期刊投稿
- 以下术语使用标准英文：[中文术语=英文术语]
- 被动语态优先
- 不要意译，保持原文的逻辑结构

中文原文：
[粘贴原文]
```

翻译任务一定要给术语对照表，否则 AI 会自己翻，翻出来可能不是你领域里通用的说法。

场景五：格式化检查

```
检查 [文件路径] 的格式规范：
- 图表标题格式是否统一为「图X-Y：标题」
- 引用编号是否连续
- 段落首行是否缩进
- 章节编号是否正确
列出所有不符合规范的地方。
```

提交论文之前跑一遍格式检查，能帮你避免很多低级错误。

场景六：导师反馈执行

```
以下是导师对第 [X] 章的修改意见：
「[粘贴导师原话]」

我理解这条意见的具体含义是：[你的解读]。
请按以下方式执行：
- [具体修改要求1]
- [具体修改要求2]
- 术语不变：[术语列表]

先告诉我你的执行方案，确认后再动手。
```

这个模板的重点是中间那行「我理解这条意见的具体含义是」。你自己先想清楚导师的意思，写成你的解读，再让 AI 执行。不要直接把导师原话丢给 AI 让它自己理解。

© 2026 晨瀚宇 未经授权禁止转载

附录 B Claude Code 快捷键速查表

Claude Code 在终端里运行，有一些快捷操作可以提高效率。

表 B.1: Claude Code 常用快捷操作

操作	说明
`Esc`（按一次）	打断当前操作。Claude Code 会停下来等你的下一条指令。适用于发现它走偏了要立刻中止的情况。
`Esc`（快速按两次）	回退到上一个检查点。它会撤销最近的修改，把文件和对话状态都恢复。适用于它改错了要还原的情况。
`claude --continue`	继续上一次被中断的会话。如果上次因为 rate limit 或网络问题断了，用这个命令恢复，Claude Code 会接着上次的上下文继续。
`/clear`	清空当前会话的上下文。如果对话太长、Claude Code 的回答开始变差（重复、遗忘），用这个命令重新开始。CLAUDE.md 和 Memory 不会被清除。
`claude`	在当前文件夹启动一个新的 Claude Code 会话。它会自动读取当前文件夹下的 CLAUDE.md。
`claude -p "指令"`	非交互模式。执行完指令后自动退出，不进入对话。适合用在脚本里或做简单的一次性任务。
`Ctrl+C`	强制终止当前进程。比 Esc 更彻底，会直接退出 Claude Code。慎用，因为不会保存任何中间状态。
`Tab`	自动补全。输入文件名或路径时按 Tab，Claude Code 会尝试补全。
上下方向键	浏览历史指令。按上键可以调出之前输入过的指令，不用重新打。

几个使用建议：

Esc 键是我用得最多的快捷键。发现 Claude Code 开始改你没让它改的地方，第一反应就是按 Esc。打断之后记得说清楚三件事：不要做什么、你想要什么、原因是什么（第三章详细讲过）。

`claude --continue` 在做长任务的时候特别有用，配合 checkpoint 文件可以从断点继续（详见第十章）。

`/clear` 要谨慎使用。清空上下文意味着你之前在这个会话里告诉它的所有信息（包括任务背景、你给的约束）都会丢失。如果你只是想让它忘掉某个不相关的话题，不如开一个新会话。

© 2026 晨瀚宇 未经授权禁止转载

附录 C 科研必备 Skill 推荐列表

以下是我在论文写作过程中使用过或确认好用的 Skill 列表。安装方式统一：在 Claude Code 对话里输入「帮我安装 [Skill 名] 这个 skill」。

表 C.1: 科研论文写作推荐 Skill

Skill 名称	说明
`/pdf`	读取 PDF 文件的全文内容。支持合并、拆分、加水印、OCR 识别扫描件。论文写作中使用频率最高的 Skill。
`/drawio`	用 Draw.io 格式画流程图、架构图、研究框架图。输出.drawio 文件，导师可以直接用 Draw.io 打开编辑。
`/markitdown`	微软开发的文件格式转换工具。把 PDF、Word、PPT、Excel 等转成 Markdown。适合把导师发来的 Word 批注转成 Claude Code 能解析的格式。
`/pyzotero`	连接 Zotero 文献库。按标签筛选文献、拉取元数据、核查引用信息。需要 Zotero 账号和 API 密钥。
`/citation-management`	引用管理和核查。搜索 Google Scholar 和 PubMed、生成 BibTeX 条目、验证引用格式。
`/arxiv-database`	搜索 arXiv 预印本。按关键词、作者、时间范围检索。适合追踪最新的方法论文章。
`/docx`	创建和编辑 Word 文档。适合学校要求提交 Word 格式论文的情况。注意 DOCX 操作有风险，操作前务必备份。
`/xlsx`	读取和编辑 Excel 文件。适合处理数据说明文件、价格表等表格数据。
`/pptx`	创建和编辑 PowerPoint。适合做答辩 PPT 和学术报告幻灯片。
`/paper-polish-workflow`	论文润色工作流。从结构到逻辑到表达逐层打磨。适合论文提交前的最终润色。
`/humanizer-zh`	中文学术润色 Skill。内置术语保护、AI 味消除、句式控制等规则。我用得最多的 Skill，第九章详细讲了使用方法。
`/statistical-analysis`	统计分析辅助。帮你选择合适的统计检验方法、检查假设条件、生成 APA 格式的结果报告。
`/seaborn`	用 Seaborn 库生成统计可视化图表。适合画箱线图、热力图、散点图等探索性分析图。
`/plotly`	生成交互式图表。适合做报告展示或探索性分析。正式论文插图建议用 matplotlib。

几点补充说明：

`/pdf` 和 `/drawio` 是我用得最多的两个 Skill，分别用了 100 多次和 104 次。如果你只装两个，就装这两个。

Skill 之间可以配合使用。比如用 `/pyzotero` 从 Zotero 筛选出一批文献，用 `/pdf` 读取其中几篇的全文，用 `/citation-management` 核查 BibTeX 格式。每个 Skill 解决一个环节的问题。

不是所有 Skill 都需要装。根据你的实际需求选择就好。如果你的论文不涉及图表，就不需要装 `/drawio`。如果你不用 Zotero，就不需要装 `/pyzotero`。按需安装，避免环境臃肿。

C.1 去哪找 Skill：skills.sh 与安装方法

Skill 从哪来？大部分现成的 Skill 都集中在一个地方：**skills.sh**（`https://skills.sh`）。这是 Vercel 推出的开源 AI 技能包管理平台，专门用来统一安装、管理和分发 AI 编程代理的技能模块。你可以把它理解成 Skill 版的 “npm”——搜索你需要的功能，一条命令装好。

每个 Skill 包含元数据文件（`SKILL.md`）、结构化指令、可选的脚本和资源，支持跨代理复用。skills.sh 上有

© 2026 晨瀚宇 未经授权禁止转载

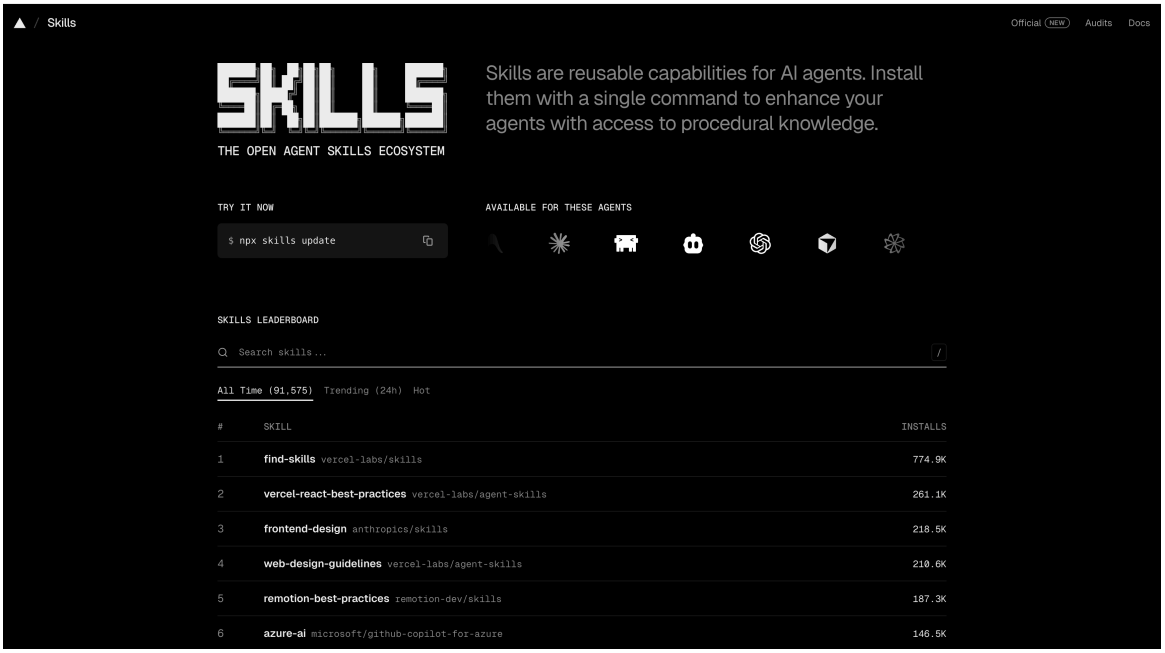


图 C.1: skills.sh 官方 Skill 注册平台

排行榜，能看到每个 Skill 的安装量，排名靠前的一般质量都不错。

安装方式很简单，在终端里运行：

```
npx skills add <owner/repo@skill-name>
```

或者更方便的做法是，先装一个叫 `find-skills` 的 Skill（它自己也是个 Skill），然后让它帮你搜：

```
npx skills find <你想搜的关键词>
```

C.2 我怎么找和用 Skill：一些实际经验

说一下我自己找 Skill 的流程，给大家参考。

第一步，先让 find-skills 帮我搜。我会把我的使用场景描述给它，比如“我需要一个能管理参考文献格式的 skill”，它会去 skills.sh 帮我搜，返回几个候选结果。

第二步，看下载量。我一般会选下载量比较高的那个，至少几百个下载量起步。下载量高说明用的人多，踩过的坑也多，质量相对有保障。如果一个 Skill 只有几个人下载，我不太敢直接用。

第三步，如果要定制，不在原 Skill 上改。比如我觉得某个文献管理的 Skill 基本思路不错，但有些地方不符合我的需求。我不会直接改它的`SKILL.md`，因为这样容易乱——下次更新就覆盖了，或者改着改着忘了哪些是原来的、哪些是自己加的。我的做法是：让 Claude Code 先参考这个 Skill 的写法，然后根据我的需求描述，新建一个独立的 Skill。这样原来的 Skill 不受影响，新 Skill 完全按我的场景来，互不干扰。

关于安全性。skills.sh 上排名靠前、下载量几千甚至上万的 Skill，一般不会有问题。但如果是下载量很少的 Skill，建议谨慎。比较稳妥的做法是：先用 `find-skills` 搜到一个类似场景的 Skill，看看它的`SKILL.md`写法，然后在它的基础上自己创建一个新的。不要硬套一个冷门 Skill 到你的场景里，虽然可能确实有完全匹配的，但下载量少意味着没有经过充分验证。

© 2026 晨瀚宇 未经授权禁止转载

C.3 已安装 Skill 完整列表

以下是我在论文写作期间实际安装过的全部 Skill（不含自动化集成类），按功能分类整理。安装方式统一：在 Claude Code 对话里输入「帮我安装 [Skill 名] 这个 skill」，或在终端运行 `npx skills add <包名>`。

表 C.2: 已安装 Skill 完整列表——文档处理类

Skill 名称	功能说明
`docx`	Word 文档创建、编辑、读取批注和修订记录
`pdf`	PDF 全文读取、合并拆分、OCR 识别、水印、表格提取
`pptx`	PowerPoint 创建、编辑、模板操作
`xlsx`	Excel 读写、数据清洗、格式化
`markitdown`	微软开发，15+ 种格式（PDF/Word/PPT/Excel/图片等）转 Markdown

表 C.3: 已安装 Skill 完整列表——学术研究类

Skill 名称	功能说明
`academic-citation-manager`	学术引用管理，自动检索和格式化参考文献
`citation-management`	搜索 Google Scholar/PubMed，生成 BibTeX，核查引用格式
`research-lookup`	通过 Parallel Chat API 查找文献和研究信息
`research-grants`	NSF/NIH 等科研基金申请书写作辅助
`research-book-style`	科研工具书写作风格（标题正式规范，正文真挚朴实）
`paper-polish-workflow`	论文系统润色流程（结构 → 逻辑 → 表达逐层打磨）
`paper-2-web`	论文转网页/视频/海报
`scientific-slides`	学术演讲 PPT 制作
`scientific-schematics`	科学示意图生成（神经网络、流程图等）
`scientific-brainstorming`	科研选题和方向探索
`scientific-critical-thinking`	科学主张和证据质量评估
`systematic-literature-review`	系统文献综述（多源检索 → 筛选 → 评分 → 撰写）
`pptx-posters`	学术海报制作
`figures4papers-playbook`	论文配图脚本模板库
`pyzotero`	连接 Zotero 文献库，读取条目和附件
`doc-coauthoring`	文档协同写作工作流

© 2026 晨瀚宇 未经授权禁止转载

表 C.4: 已安装 Skill 完整列表——数据分析与可视化类

Skill 名称	功能说明
exploratory-data-analysis	200+ 格式的科学数据探索性分析
statistical-analysis	统计检验选择、假设检查、APA 格式报告
plotly	交互式可视化（Plotly 库）
seaborn	统计可视化（箱线图、热力图等）
shap	机器学习模型可解释性（SHAP 值）
networkx	图/网络数据分析
sympy	符号数学计算
dask	分布式大规模数据处理
vaex	超大规模表格数据处理（亿级行）
umap-learn	UMAP 降维可视化
timesfm-forecasting	Google TimesFM 时间序列预测
fred-economic-data	美联储 FRED 经济数据查询
geopandas	地理空间矢量数据分析
geomaster	遥感、GIS 与空间分析

表 C.5: 已安装 Skill 完整列表——绘图设计与网络搜索类

Skill 名称	功能说明
drawio	Draw.io 格式流程图、架构图（8900+ 图标库）
infographics	信息图设计（10 种类型、8 种风格）
nanobanana-image-generation	Gemini AI 图像生成与科学配图
web-access	网页访问、搜索、内容抓取
parallel-web	并行网络搜索与深度研究
perplexity-search	Perplexity AI 驱动的实时信息搜索
humanizer-zh	中文学术润色，去 AI 痕迹，术语保护

表 C.6: 已安装 Skill 完整列表——生物信息与生命科学类

Skill 名称	功能说明
biopython	分子生物学工具包（序列、BLAST、NCBI 访问）
anndata	单细胞数据结构（.h5ad 格式）
scvi-tools	单细胞数据概率模型与批次校正
alphafold-database	AlphaFold 2 亿 + 蛋白质结构预测数据
esm	ESM 蛋白质语言模型
deeptools	NGS 数据分析（ChIP-seq/RNA-seq/ATAC-seq）

© 2026 晨瀚宇 未经授权禁止转载

Skill 名称	功能说明
polars-bio	高性能基因组区间操作
gtars	Rust 实现的基因组区间分析
geniml	基因组区间机器学习
arboreto	基因调控网络推断（GRNBoost2/GENIE3）
histolab	全切片病理图像处理
matchms	质谱相似度与化合物鉴定
lamindb	生物数据管理框架
glycoengineering	蛋白质糖基化工程分析
opentrons-integration	Opentrons 液体处理机器人协议
protocolsio-integration	protocols.io 实验方案管理

表 C.7: 已安装 Skill 完整列表——科学数据库类

Skill 名称	功能说明
gene-database	NCBI Gene 基因查询
geo-database	NCBI GEO 基因表达数据
biorxiv-database	bioRxiv 生命科学预印本
arxiv-database	arXiv 多学科预印本
pubchem-database	PubChem 1.1 亿 + 化合物
clinvar-database	ClinVar 临床变异数据
cosmic-database	COSMIC 癌症体细胞突变
ensembl-database	Ensembl 250+ 物种基因组
interpro-database	InterPro 蛋白质功能域
jaspar-database	JASPAR 转录因子结合位点
kegg-database	KEGG 代谢通路
hmdb-database	HMDB 22 万 + 人类代谢物
brenda-database	BRENDA 酶动力学数据
zinc-database	ZINC 2.3 亿 + 可购买化合物
drugbank-database	DrugBank 药物信息
metabolomics-workbench-database	NIH 代谢组学数据
imaging-data-commons	NCI 癌症影像数据
fda-database	FDA 药物/器械/不良事件
uspto-database	美国专利商标检索

© 2026 晨瀚宇 未经授权禁止转载

表 C.8: 已安装 Skill 完整列表——化学材料与建模类

Skill 名称	功能说明
`datamol`	RDKit 封装，分子处理与药物发现
`molfeat`	分子特征化（100+ 特征器，ECFP/MACCS 等）
`medchem`	药物化学筛选（Lipinski 规则、PAINS 过滤）
`pytdc`	治疗数据集（ADME/毒性/药物靶点）
`pymatgen`	材料科学（晶体结构、相图、能带）
`rowan`	云端量子化学计算平台
`molecular-dynamics`	分子动力学模拟（OpenMM/MDAnalysis）
`cobrapy`	约束代谢建模（FBA/FVA）
`fluidsim`	计算流体力学模拟
`pymc`	贝叶斯概率建模（MCMC/变分推断）
`scikit-survival`	生存分析与时间事件建模
`dowhy-causal-inference`	因果推断（因果图、效应估计）
`qiskit`	IBM 量子计算框架

表 C.9: 已安装 Skill 完整列表——医疗合规与开发工具类

Skill 名称	功能说明
`pydicom`	DICOM 医学影像读写
`treatment-plans`	医疗治疗方案生成
`iso-13485-certification`	医疗器械 ISO 13485 认证文档
`benchling-integration`	Benchling 实验室 R&D 平台集成
`consciousness-council`	多视角讨论（模拟不同立场的委员会）
`skill-creator`	创建、编辑和测试 Skill
`find-skills`	发现和安装新 Skill
`get-available-resources`	检测系统计算资源（CPU/GPU/内存）
`denario`	多智能体科研自动化

© 2026 晨瀚宇 未经授权禁止转载

附录 D CLAUDE.md 科研版模板

以下是一份面向科研论文写作的 CLAUDE.md 模板。你可以直接复制到你的项目文件夹根目录，然后根据自己的情况修改方括号里的内容。

```
# CLAUDE.md

## 项目基本信息
- 论文题目：[你的论文题目]
- 研究方向：[一句话描述]
- 当前阶段：[初稿/修改中/定稿/答辩准备]
- 论文格式：[LaTeX/Word]
- 提交要求：[学校的格式规范，比如字体、行距、页边距]

## 文件结构
- chapters/ — 各章正文
- tables/ — 所有表格文件
- figures/ — 所有图表文件
- references/ — BibTeX文件和参考文献PDF
- appendix/ — 附录文件

## 术语锁定表
以下术语不得修改或替换：
- [术语1]
- [术语2]
- [术语3]
如果你觉得某个术语需要修改，先问我，不要自己改。

## 写作规范
- 用学术中文，不要口语化表达
- 每段只讲一件事
- 句子不超过25字
- 不用「由于……因此……」的长句式
- 图表标题格式：图X-Y：标题内容
- 引用格式：GB/T 7714

## 修改规则
- 修改正文时保留所有引用编号
- 不要删除任何引用
- 改之前先告诉我你的方案，确认后再执行
- 改完给我看修改前后的对比
- 修改任何 .docx 文件前，先创建备份
```


© 2026 晨瀚宇 未经授权禁止转载

```
## 导师偏好
- [导师对写作风格的具体要求，比如「一段一个主题」]
- [导师的用词禁忌，比如「不要用实证分析」]
- [导师对格式的要求，比如「每章另起一页」]

## 常见错误防线
- 不要把「被解释变量」改成「因变量」
- 不要删掉引用编号
- 不要自己编造文献观点，不确定的标注[需核实]
- [其他你踩过的坑，每踩一次加一条]
```

几点使用建议：

这个模板是一个起点，不是最终版。你用的过程中发现任何需要补充的规则，随时加进去。Claude Code 每次启动会话都会读这个文件，你加的规则下一次会话就生效了。

术语锁定表是最重要的部分。你论文里的核心术语一定要列在这里。如果你不确定哪些词算核心术语，想想：如果 AI 把这个词换成了同义词，会不会导致全文不一致？如果会，就是核心术语。

「常见错误防线」这一栏，建议每次 Claude Code 犯了一个你不想看到第二次的错误，就加一条。半年下来这一栏会变得很长，但每一条都有用。

「导师偏好」这一栏，把导师在开会时说过的具体要求记在这里。比如「摘要不要放具体数字」「标题不要用问句」「表格不要跨页」。这些要求 Claude Code 知道了之后，帮你改稿的时候就会自动遵守。

© 2026 晨瀚宇 未经授权禁止转载

附录 E 常见错误与解决方案速查

以下是使用 Claude Code 辅助论文写作时最常见的问题和对应的解决方法。

1. Claude Code 把你的术语改了

现象：润色时专业术语被替换成同义词（第一章详细讲过这类案例）。

原因：它不知道哪些词在你的论文里是不能动的。

解决方法：在 CLAUDE.md 里维护一份术语锁定表（见第二章）。在每次修改任务的指令里也重复一遍关键术语。参考附录 D 的模板。

2. 改着改着把不该改的地方也改了

现象：你让它改第三段的措辞，它顺手把第四段的一句话也改了，还删掉了一个引用编号。

原因：你没有明确限定修改范围。

解决方法：指令里说清楚「只修改第 X 段，其他段落不要动」。在 CLAUDE.md 里加一条「修改正文时保留所有引用编号」。改完之后用 diff 对比改前改后，确认只改了你让它改的地方。

3. 会话中途被 rate limit 打断

现象：做到一半，Claude Code 提示用量到了上限，整个会话停了。

原因：Pro 版的用量有限，长任务容易超限。

解决方法：升级到 Max 5x 可以减少被打断的频率。更重要的是在指令里要求它设检查点（详见第十章「Rate limit 处理与断点续跑」一节）。被打断后用 `claude --continue` 恢复会话，它能读到检查点文件从断点继续。

4. 编造文献观点或数据

现象：让它帮你写文献综述，它写了一句「某某研究发现……」，你去查原文发现那篇论文根本没说这个。

原因：它根据标题和已知信息推测文献内容，推测不一定对。

解决方法：在 CLAUDE.md 里加一条「不确定的内容标注 [需核实]」。所有涉及文献评述的内容，你都要回到原文核实。参考第十四章第二节。

5. DOCX 文件被覆盖，原文丢失

现象：让它帮你改 Word 文件，它直接覆盖了原文件，导师给的批注和你之前的修改全没了。

原因：Claude Code 默认是直接写回原文件的。

解决方法：在 CLAUDE.md 里加一条「修改任何.docx 文件前，先创建备份」。或者配置一个自动备份的 Hook（第十一章讲过）。更好的做法是尽早迁移到 LaTeX 工作流，LaTeX 文件是纯文本，和 Git 配合可以做完整的版本管理。

6. Draw.io 图的 MCP session 丢失

现象：花了半小时做好了一张图，切到下一个话题再回来，图没了。

原因：Draw.io MCP 的 session 状态不跨 turn 保存。

解决方法：每次图修改到一个满意的状态，立刻让 Claude Code 把 XML 保存到本地文件。用文件作为唯一的真相来源。下次要修改的时候，让它从文件读取。参考第六章的详细说明。

7. 长句没有被拆开

现象：你让它帮你改写一段话，改完还是有很多超过 50 字的长句。

原因：你的指令里没有明确提出句子长度的要求。

© 2026 晨瀚宇 未经授权禁止转载

解决方法：在指令里加一条「每句不超过 25 字」或「找出所有超过 50 字的句子，逐句改短」。如果你经常做这类任务，把句子长度要求写进 CLAUDE.md 的写作规范里。

8. 改完的结果和导师要求不符

现象：导师说「结论太弱」，你让 Claude Code 改，改完导师看了说「不是这个意思」。

原因：导师的意见是概括性的，你没有翻译成具体的修改要求就直接交给 AI 了。

解决方法：收到导师意见后，自己先想清楚每条意见的具体含义和修改方向，把你的理解写成具体指令再交给 Claude Code。参考第十四章第一节的例子和附录 A 的「导师反馈执行」模板。

© 2026 晨瀚宇 未经授权禁止转载

附录 F 从论文 PDF 到组会 Beamer：paper-to-beamer Skill 实战

内容提要

- 这个 Skill 是为了解决什么问题：现有 AI 生成 PPT 方案在学术场景下的三个痛点
- 整体设计：六阶段流水线、论文类型骨架、模板四层栈
- 详细安装步骤：从 cp -r 到第一次跑通，给完全没碰过 Skill 的同学看
- 第一次跑通的完整对话演示，每一步该说什么、Claude 会问你什么
- 我自己用这个 Skill 时踩过的几个坑和绕开方法
- 进阶阅读：本书附赠的 paper_writeup.pdf 是什么、怎么打开

写完这本书的初稿之后，我自己手头还有一个一直没解决的问题——**怎么把一篇论文变成组会汇报的 Beamer 幻灯片**。书前面的章节讲了用 Claude Code 写论文、改论文、查文献，但没讲组会汇报这块。问题是，组会几乎是研究生周更的频次，而每次手工做一套幻灯片至少要花半天到一天。这个事如果不能自动化掉，前面那些章节积累下来的效率优势就会被抵消掉一大半。

这个附录就是讲我做的一个新 Skill：`paper-to-beamer`。它解决的就是「论文 PDF 到组会 Beamer」这一段的自动化。和这本书的其他章节一样，这部分也是一边踩坑一边写出来的，方法不一定最优，但至少在我自己组会汇报的场景下经过了实测。

F.1 为什么需要这样一个 Skill

现状：用 AI 生成 PPT 已经成为一股潮流。写完本书大半内容之后，我开始想——组会汇报这一环能不能也用 AI 来提效。打开小红书一搜，能看到大量「用 AI 一键做 PPT」的教程；身边的同学也在用豆包、Gamma、Nano Banana 这类工具帮自己赶组会的幻灯片。从出稿速度上看，这条路确实比手动一页一页拖版式快得多，所以很多人选它是情理之中的事。

但我真把这些工具搬到组会场景里试了一遍之后，发现几个问题绕不过去。

第一，生成内容不符合学术规范。豆包、Gamma 之类的通用 PPT 工具，默认风格更像工作汇报或商业路演——花哨的图标、渐变背景、营销腔的标题。这套视觉语言用在组会上会显得格调不对。组会的受众是导师和同学，他们想看的是论文的思路和实验结果，不是幻灯片的美术效果。更麻烦的是，这些工具生成的内容在专业术语、公式表述、引用体例上经常与学术规范有出入，真拿到组会上还得逐页做清理工作。

第二，AI 生成的图片里中文字符经常错位或损坏。这是 Nano Banana 这类图像生成工具在当前阶段的一个结构性问题。你让它画一张带中文标注的架构图或流程图，图里的汉字很容易出现缺笔画、字形走样、或拼成无法识别的字符。英文稍好一些，但也谈不上稳定。把这样一张图放进组会 PPT 里，一旦被导师看出来就很难收场。这是目前主流图像生成模型的普遍限制，并非偶发小瑕疵，短期内没有可靠的解决办法。

第三，修改成本极高。这一点对我来说是最关键的。AI 生成的 PPT 或配图本质上是一个「最终产物」——拿到之后如果只想动一个局部（比如把某一页的某个术语换掉、或者把某张图里的一个标注往左挪一点），很多工具要么不支持这种粒度的编辑，要么一改就把整页或整张图的风格全部重新生成一遍。组会汇报从来不是一

© 2026 晨瀚宇 未经授权禁止转载

次性的交付，它需要你根据导师的反馈反复调整。一个不能做局部微调的工具，放进「改两轮再定稿」的真实流程里是走不通的。

解决方案：回到 LaTeX Beamer。把上面三个问题摆到一起看，我得到的结论是——现阶段做组会汇报幻灯片最稳健的选择，还是用 LaTeX 的 Beamer 来写。理由正好对应上面三条：Beamer 是学术社区长期使用的幻灯片格式，默认排版风格本身就符合学术汇报的审美；中文排版通过 `xeCJK` 加华文楷体字体可以做到稳定无乱码；源码是纯文本的 `.tex` 文件，任何一页、任何一条要点、任何一张图都能单独改，改完重新编译就行，不会出现「动一处、全盘重来」的情况。

至于配图，paper-to-beamer 的设计选择是完全不用 AI 生图，直接从原始论文 PDF 里截取。组会讨论的就是论文本身，你在讲 Transformer 的多头注意力，幻灯片上显示的就应该是原论文第三页那张架构图，而不是 AI 重画的某张" 类似" 的图。原图就是原图，不做任何风格化处理。

但 Beamer 有一个常被吐槽的问题：手写太慢。一篇二十多页的汇报，从引入包、配字体、搭主题到逐帧填内容，全手工做至少要半天。既然初衷是「用 AI 提效」，那就得让 AI 来写 Beamer 源码。这就是 `paper-to-beamer` 要做的事：**让 AI 按确定性的流水线生成 Beamer LaTeX 源码，既享受 AI 的生成速度，又保留 LaTeX 源码的可修改性。**

我想要的产出形态。我要的是一份「八九成完成的 LaTeX 源码」，而不是一个无法修改的最终产物。组会汇报的真实工作流是：自动生成一份初稿，再根据自己对论文的理解补几页补充内容，然后根据导师的反馈改两轮。这套流程的每一步都需要可修改的源码作为载体。所以最终我把它做成了 `paper-to-beamer` 这个 Skill，按确定性的流水线执行，输出 Beamer LaTeX 源码，每一帧、每一条要点、每一张图都能后期手工调。

F.2 Skill 的整体设计

核心思想：六阶段流水线。大模型写代码本身是有随机性的，但如果你把任务切成足够小的步骤，并且在每一步执行确定性的局部正确性检查，整条链路的首次成功率就会上去。`paper-to-beamer` 把生成幻灯片这件事拆成了六个阶段。

定义 F.1 (paper-to-beamer 的六阶段流水线)

阶段一：用 `/pdf` 子技能读论文，抽取标题、章节、关键公式和图表位置，存成 JSON。阶段二：通过 `AskUserQuestion` 一次性向用户抛出四个澄清问题——语言、时长、主题、插图方式。阶段三：拷贝字体文件和 `suda.sty` 主题文件到工作目录，用 PyMuPDF 以三倍缩放从 PDF 截取关键图。阶段四：按论文类型选择对应的幻灯骨架，逐帧填入内容。阶段五：调用 Python 校验脚本做静态检查（括号、环境、引用、图片）。阶段六：按 `xelatex → bibtex → xelatex → xelatex` 的顺序跑四次完整编译。 ♣

每个阶段都有明确的输入、输出和退出条件。只有上一阶段的输出通过了局部检查，下一阶段才会启动。这种「拆小步」的思路其实和第三章讲的「一次会话只做一件事」是同一回事，只是在这里它从对话级别下沉到了 Skill 内部的执行流程。

论文类型 → 幻灯结构的映射。不同类型的论文应该有不同的讲法。讲 Transformer 这种架构型论文，重点应该落在模型图和组件讲解上；讲 Scaling Laws 这种实证型论文，重点应该在实验设置和结果可视化。如果让大模型自己决定章节比例，经常出现「八页背景只有两页方法」的失衡。所以我在 Skill 里写死了四类论文对应的骨架。

以最常见的架构型为例，一篇二十五分钟的汇报会被切成 2 页概述、3–4 页背景、8–10 页架构、2–3 页训练、

© 2026 晨瀚宇 未经授权禁止转载

表 F.1: 四类论文的内容帧分配（不含自动插入的目录页）

类型	背景	方法	实验	总结
架构型	3–4	8–10	3–4	2
实证型	3	3	6	2–3
综述型	3	8	3	2
理论型	4	8	3	2

3–4 页实验、2 页总结，加上自动目录页大约二十五帧——正好满足「一分钟一页」的组会经验法则。

模板四层栈。模板的外观和内容是分层解耦的。最底下是引擎层：固定 XeLaTeX + BibTeX，不允许换成 pdflatex。往上是 CJK 层：xeCJK 包加华文楷体字体文件。再往上是主题层：suda.sty，负责主题配色和自动目录。最上面是内容层：由 Claude 在阶段四逐帧填入的 .tex 源码。这套分层的好处是想换学校或换字体的时候只动对应那一层，内容层完全不用碰。

F.3 安装这个 Skill

这一节给完全没装过 Skill 的同学看。如果你已经熟悉第九章的安装流程，可以跳过。

先确认两件事。第一，你的电脑上已经装好了 Claude Code。在终端里跑一下 claude --version，能看到一行版本号就说明 OK。第二，你已经从本书的附赠包里拿到了 paper-to-beamer/ 这个文件夹（在 bonus/ 目录下）。

核心安装命令。Skill 在 Claude Code 里的本质就是 ~/.claude/skills/ 目录下的一个文件夹。装它就是把那个文件夹复制过去：

```
cp -r paper-to-beamer ~/.claude/skills/
```

这条命令需要你在 paper-to-beamer/ 文件夹的父目录里运行。比如你把附赠包解压到了 ~/Downloads/Claude_Code_in_Paper_Writing_V2.0/bonus/，那就先 cd 到那个目录，再跑上面的命令。

验证安装。跑完上一步之后，确认一下文件是不是真的过去了：

```
ls ~/.claude/skills/paper-to-beamer/
```

如果你看到 SKILL.md、assets、references、evals 这几个名字，说明装好了。如果显示 No such file or directory，先回头检查 cp 是不是在正确的目录里跑的。

不需要重启 Claude Code。下次你开新会话的时候它会自动发现这个 Skill。

注 有些同学会问”我能不能让 Claude Code 帮我装”。可以的，按第九章 9.4 节的方法，在对话里直接说「帮我安装 paper-to-beamer 这个 skill」，Claude Code 会自动去 skills.sh 上找。但因为这个 Skill 暂时还没上传到公开平台，目前只能用 cp -r 这种本地安装的方式。

F.4 第一次跑：完整的对话演示

假设你手头有一篇 bert.pdf，想做一个 20 分钟的中文组会汇报。完整的交互大致是这样。

第一步：在 PDF 所在目录里打开 Claude Code。先 cd 到 bert.pdf 所在的文件夹，然后运行 claude 启动会话。Claude Code 会把当前文件夹作为工作目录，后面所有读文件、写文件的操作都在这里进行。

第二步：用自然语言下指令。直接在对话里输入：

© 2026 晨瀚宇 未经授权禁止转载

```
用 paper-to-beamer skill 把 bert.pdf
做成一份 20 分钟的中文组会 PPT
```

Claude 看到 beamer 和「组会 PPT」这种关键词会自动加载对应的 Skill。

第三步：回答四个澄清问题。Skill 启动后会一次性弹出四个问题让你勾选。

- **语言**：中文为主 + 英文术语 / 全英文 / 双语对照。组会汇报场景默认选第一个。
- **时长**：15–20 分钟 / 25–30 分钟 / 40–45 分钟。20 分钟选第一档。
- **主题**：保留模板原样 / 去掉校徽保留配色 / 自定义。如果不使用该配色，选第二个。
- **插图方式**：从 PDF 截取 / TikZ 重绘 / drawio 重绘。强烈建议第一个——AI 生图不可修改、风格浮夸、还会塞 emoji，原图截取才符合组会场景。

如果你不确定怎么选，按每个问题的第一个选项（推荐项）走就行，组会场景下默认值已经够用。

第四步：等 Skill 跑完整条流水线。这一步它会依次做这些事：用 /pdf 读论文、用 PyMuPDF 把图截出来、按论文类型骨架写 LaTeX、跑预校验脚本、给你打印一份完成报告。整个过程大概几分钟，不要中途打断。

第五步：拿到工作目录并编译。Skill 会在当前文件夹下生成一个新目录（默认叫 paper_writeup/ 或类似名字），里面有 main.tex、figs/、ref.bib、suda.sty 和 fonts/。如果你本地有 TeX Live，按下面的顺序跑四次：

```
xelatex main.tex
bibtex main
xelatex main.tex
xelatex main.tex
```

如果你没有本地 TeX 环境（这是大多数同学的情况），把整个工作目录拖到 Overleaf 新项目里，菜单里把编译器改成 XeLaTeX，按 Recompile 就行。Overleaf 自带华文楷体所需的字体包，不用再额外上传字体文件。

F.5 我自己踩过的几个坑

坑一：subsection 陷阱。第一次写这个 Skill 的时候我没意识到 suda.sty 在 \AtBeginSubsection 也注册了目录钩子。结果生成出来的 PDF 七十多页，一半都是重复目录。后来在 SKILL.md 里加了一条硬规则——完全禁用 \subsection{}，所有二级切分改用加粗段落或 block 环境。从那之后这个问题没再出现过。

坑二：内容密度上限有时候太紧。「每帧最多两个公式」对 BERT 之类的论文够用，但对 GAN 的极小极大目标或 ResNet 的残差块公式就显得局促。Skill 当前的处理是把公式拆到下一帧，但这会让一个推导被打断。下一版我打算做成「论文类型敏感」的密度规则，理论型论文的上限放宽到四个公式。

坑三：图片抽取需要人工核对。PyMuPDF 截图的时候是按矩形坐标裁剪的，坐标不准就会截到一半。Skill 在抽完图之后会用 Read 工具自己看一眼缩略图来确认，但视觉判断不一定可靠。建议你拿到工作目录之后手动浏览一下 figs/ 文件夹，发现裁切不好的可以让 Claude 重新裁。

坑四：模板配色不是万能的。它毕竟是一个学院的主题，配色和字体风格较为固定。如果觉得默认配色不合适，目前需要自己替换 suda.sty 里的几个 RGB 值。这是下一版要解决的问题——多模板支持、下拉选择。

© 2026 晨瀚宇 未经授权禁止转载

F.6 进阶阅读：附赠的 paper_writeup.pdf

本书 V2.0 的附赠包里除了 `paper-to-beamer/` 这个 Skill 目录之外，还有一份独立的 PDF——`paper_writeup.pdf`。这一节解释一下它是什么、为什么和书一起发、怎么打开来读。

这是什么。`paper_writeup.pdf` 是一份 6 页的双栏中文学术论文，标题是《*Paper-to-Beamer*：一个面向科研组会汇报的论文转幻灯片 *Skill* 的设计与实现》。它讲的内容和这一章是同一件事，但形式完全不同——它是真正的双栏学术论文格式，有摘要、有引言、有相关工作、有六张图、有三张表、有十一条参考文献。

为什么要单独做一份。这本书的口吻是「学长聊天」，长篇散文加上偶尔的 `remark` 框，适合从头读到尾的同学。但科研同行有时候需要的是另一种东西——*看起来像学术论文*的呈现。比如你想在组会上" 顺手提一下我用了一个自己做的 skill"，能掏出一份双栏论文比掏出一本书的章节更有说服力。两种形态承担的阅读场景不同，所以我没把它们合并，而是分开发。

怎么打开。`paper_writeup.pdf` 是一份普通的 PDF，任何 PDF 阅读器都能打开。如果你想自己改它的内容（比如换个作者名、改成你们学院的配色再发到自己的小红书），它的 LaTeX 源码也在附赠包的 `paper_writeup_source/` 文件夹里——`main.tex`、`sections/`、`refs.bib`、`figures/` 全都齐了。源码是 Overleaf-ready 的，主编译器选 XeLaTeX 就能直接编。

这份论文是怎么来的。`paper_writeup.pdf` 本身不是手写出来的，由另一个 Skill `skill-to-paper` 生成。`skill-to-paper` 是我做完 `paper-to-beamer` 之后，把" 为一个 Skill 写一份学术风格的笔记" 这件事也封装成了一个 Skill。它和 `paper-to-beamer` 的分工是：`paper-to-beamer` 的输入是一篇论文 PDF，输出是 Beamer 幻灯片源码；`skill-to-paper` 的输入是一个已有的 Skill 目录，输出是一篇介绍这个 Skill 的双栏论文。附赠包里的 `paper_writeup.pdf` 就是 `skill-to-paper` 跑出来的产物。

如果你以后自己也做了一个 Skill 想给它写一份学术风格的笔记，可以直接用 `skill-to-paper`——它在附赠包里的位置是 `bonus/skill-to-paper/`。装法和 `paper-to-beamer` 一样，`cp -r` 到 `~/.claude/skills/` 即可。

注 这一章和 `paper_writeup.pdf` 的关系类似于教科书和论文的关系。前者是教程性质的、循序渐进的、适合从头读的；后者是结论性质的、压缩的、适合快速浏览的。同一件事的两种表达形态，根据你的阅读场景选一种就好。

© 2026 晨瀚宇 未经授权禁止转载