# 理解大模型，用好ai coding

## 一、AI 是怎么学会"思考"的

故事要从 1950 年说起，大名鼎鼎的计算机科学之父 ——Alan Turing，提出了著名的图灵测试。也就是说，如果一个人隔着墙和机器人聊天，如果不能分辨对方是人还是机器，这个机器就算是具有智能了。

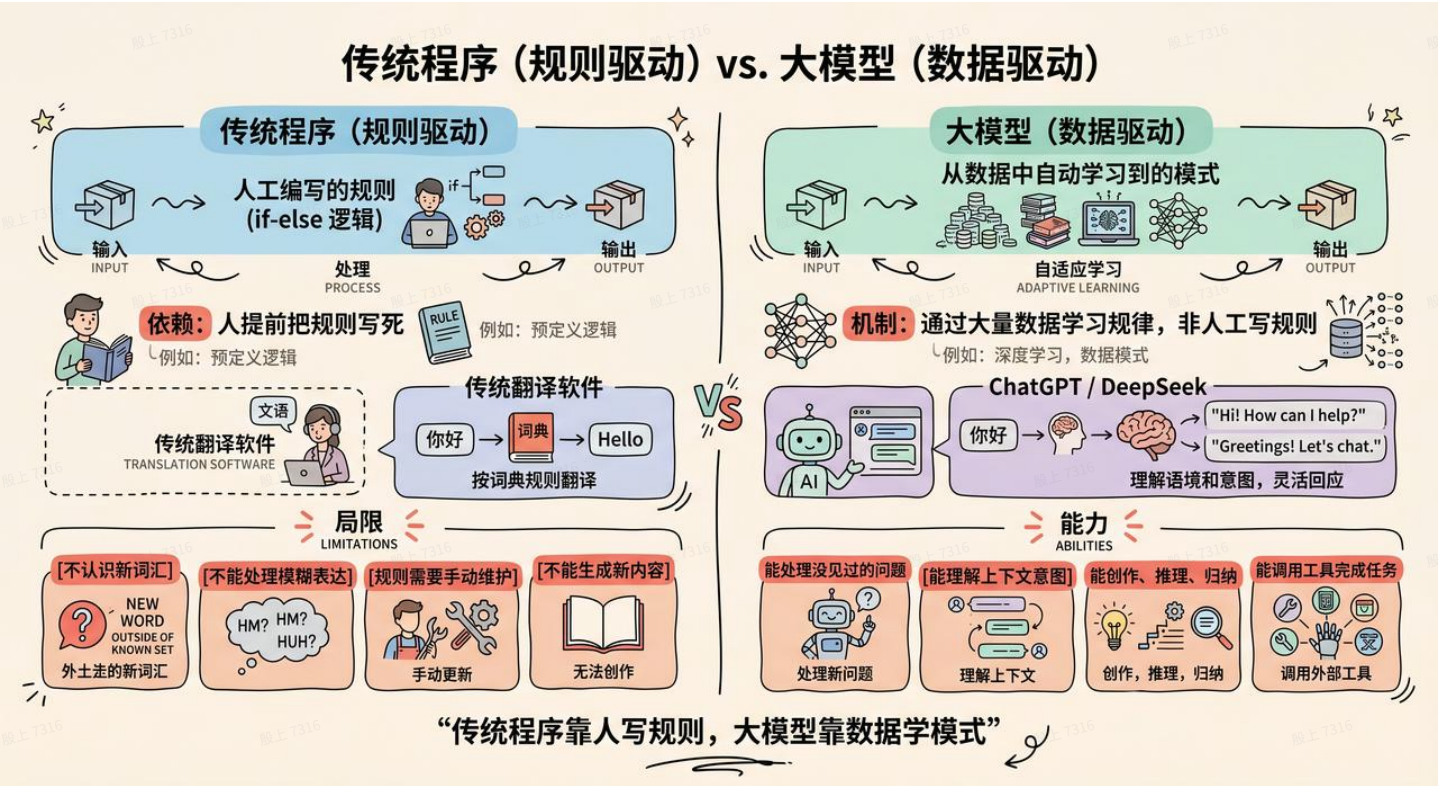
最开始研发的是 —— 专家系统，这种基于规则驱动的程序，本质上是规则的穷举。程序员将人类的知识编写成无数条 if else 规则。

```
代码块
1    如果  看到毛茸茸的、有尖耳朵的、有长尾巴的动物，那么  它是猫。
```

这种方式的局限性显而易见。如果一只猫因为意外失去了尾巴，或者它是一只没有毛的猫呢？

世界太过于复杂，人力资源有限，不可能写入所有规则。同时世界也在不断变化，规则很难及时更新

我们无法将人类总结的规则交给机器，那么为什么不让机器针对数据自己找规律呢？




于是，AI 进一步发展到了**机器学习**阶段：我们不再手工编写每一条规则，而是提供大量数据和对应答案，让机器自己从中“学出”中间的规律。

在这一阶段，主要有三种主流学习方式：

- **监督学习（Supervised Learning）**：给机器提供带标签的数据。例如，给它看 10000 张猫的图片，并标注为“猫”；再给它看 10000 张狗的图片，并标注为“非猫”。机器通过梯度下降等方法不断试错、调整内部参数，最终自己总结出“猫”的统计特征。
- **无监督学习（Unsupervised Learning）**：不给数据打标签，而是让机器自己发现其中的结构和规律。比如，把一批新闻交给它，它可以自动将体育、财经、娱乐等内容聚类区分开来。
- **强化学习（Reinforcement Learning）**：不直接告诉机器正确答案，而是通过奖励和惩罚引导它不断优化行为。就像训练小狗，做对了给予奖励，做错了受到惩罚。自动驾驶，以及当年击败柯洁的 AlphaGo，背后都能看到强化学习的影子。

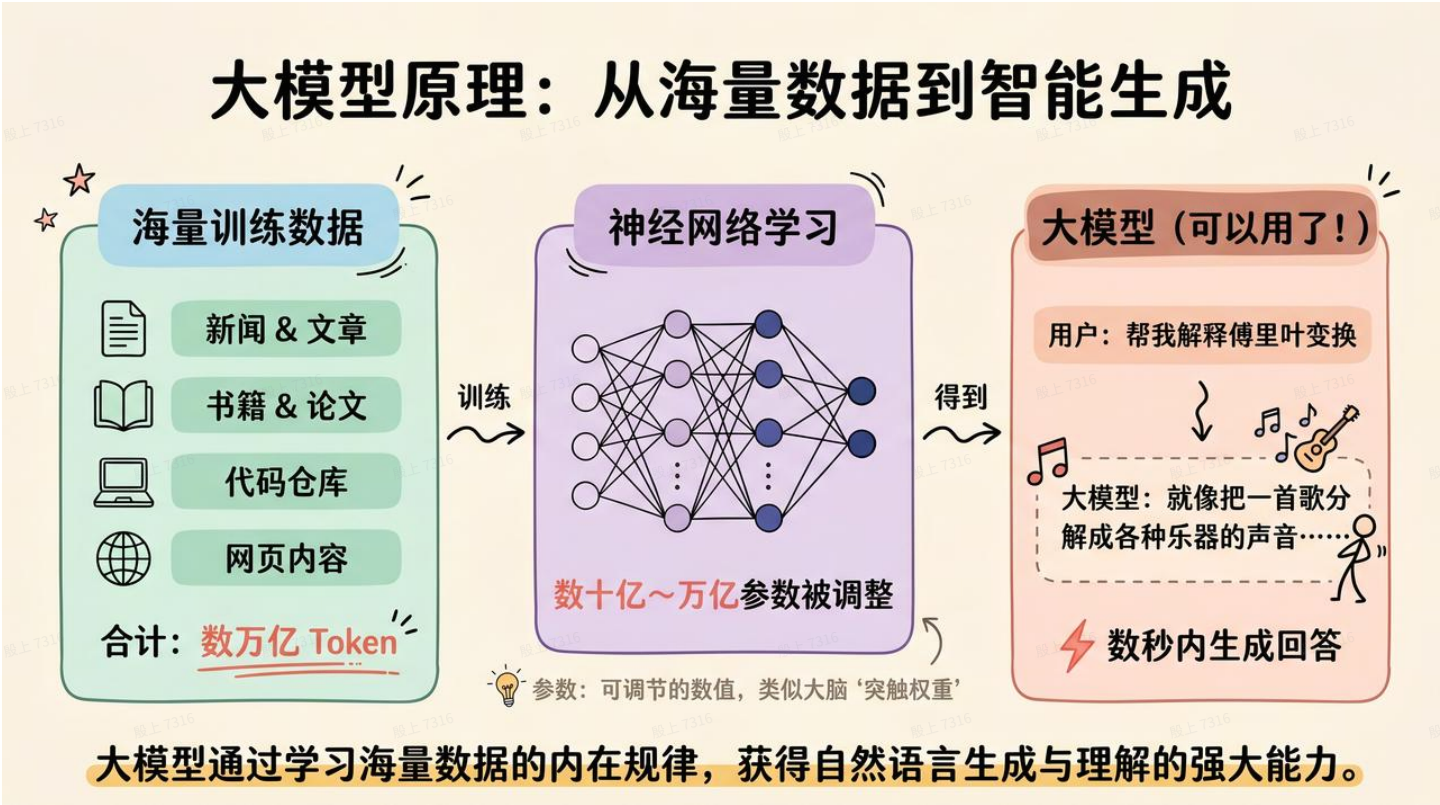
数据从输入层进入，经过中间一层层**隐藏层（Hidden Layers）**的过滤、提取与组合，最终在输出层得到结果。

在这个阶段，机器已经不再依赖人工手动提取特征；只要神经网络的层数足够深、数据量足够大，它就能够自动从海量数据中学习并提取有效特征。

也正是在这一时期，**深度学习**在自然语言处理和计算机视觉领域大放异彩。
**不过，即便如此，这一阶段的 AI 本质上仍然更擅长“选择题”和“判断题”——也就是在已有选项或目标中做出最优判断。**

真正的下一次跃迁，来自那篇大名鼎鼎的论文 **《Attention Is All You Need》**。2017 年，Google 提出了 **Transformer** 架构。

其中最关键的突破，就是**注意力机制（Attention Mechanism）**。它克服了 RNN 和 LSTM 必须按从左到右、逐词处理信息的局限，使模型能够同时看到句子中的所有词语，并快速捕捉它们之间的关系。




比如这句话：

代码块

```
1    我今天买了两个 苹果 ，洗干净后就吃了。
```

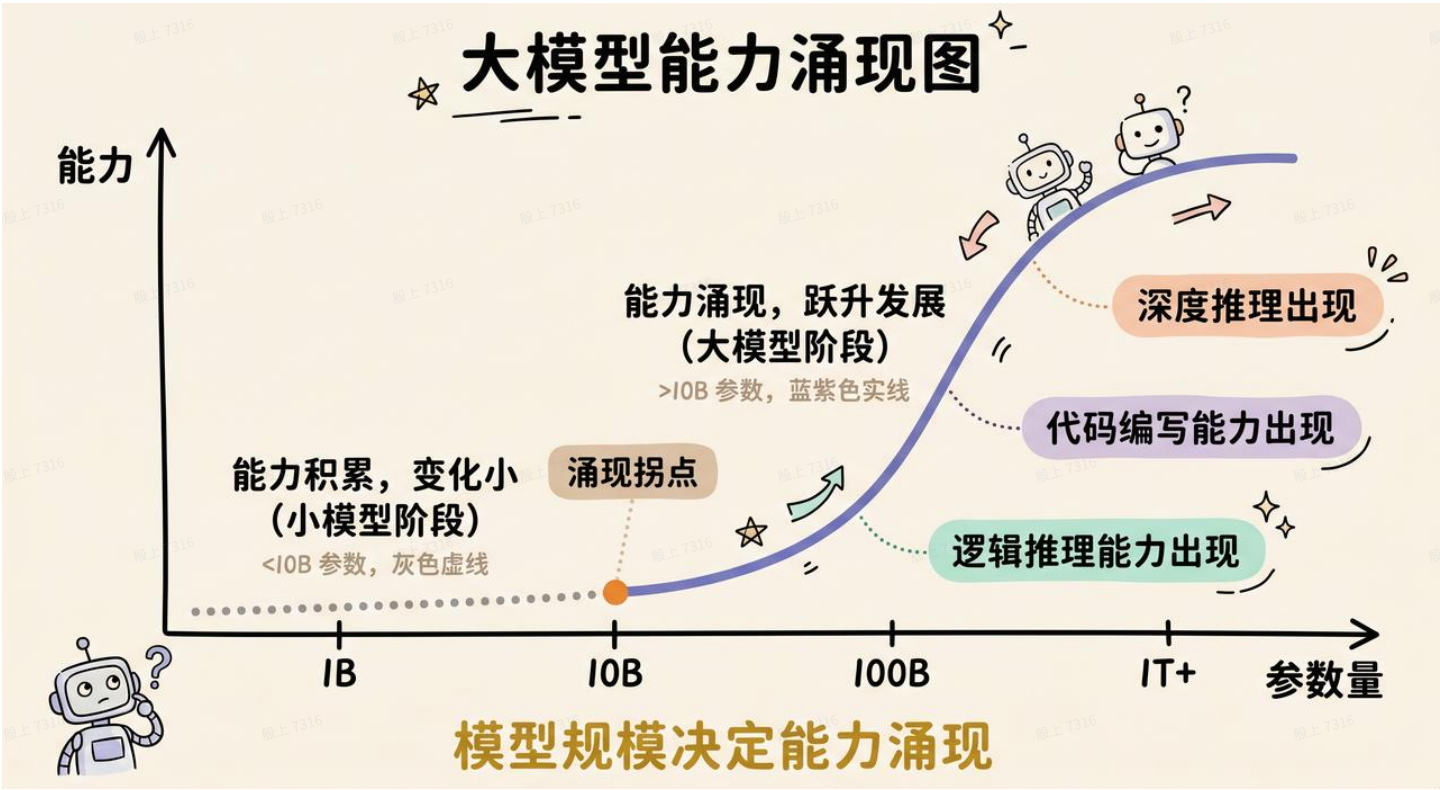
"苹果"这个词，既可以指水果，也可以指苹果公司。
如果是传统模型，可能只能根据词典做机械匹配，无法准确判断具体含义。
但在注意力机制下，模型会同时关注“苹果”周围的上下文词汇，比如“两个”“洗干净”“吃了”，并对这些与语义强相关的信息赋予更高权重，因此能够迅速理解，这里的“苹果”指的是**水果**，而不是公司。

与此同时，Transformer 架构还有一个关键优势：**天然适合并行计算**。
这意味着，随着 GPU 规模不断扩展，模型训练效率和能力也会同步跃升，最终让算力的“量变”逐步引发模型能力的“质变”。

当数据规模、模型参数和训练算力都达到一定程度后，**基础模型（Foundation Model）开始出现。这类模型不仅学会了人类语言的语法和表达方式，还逐渐展现出一些此前并未被显式设计进去的能力，也就是所谓的涌现能力（Emergent Abilities）**，例如逻辑推理、代码生成、复杂表达，甚至对情绪语气的理解与模拟。

到了这一步，AI 的能力边界也发生了变化：

它不再只是做“选择题”，而是开始进入“填空题”和“作文题”的阶段——不仅能判断哪个答案更对，还能主动生成答案、组织语言，甚至完成较复杂的内容创作。




## 二、大模型底层的特性

# 1. 核心思想：预测下一个词

大模型的训练目标，出乎意料地简单：

> 给定前面所有的词，通过注意力机制预测下一个词最可能是什么。

就这一句话，撑起了整个大模型的世界。

来看一个例子：

```
代码块
1    输入：小明在操场___，因为今天天气很好
2    模型预测：踢球（42%）、跑步（18%）、.....
```

模型的阅读理解机制，当我们让模型完成句子填空：“小明在操场___，因为今天天气很好”

模型要做 3 件事，这就是自注意力机制的工作流程：

**1. 词嵌入（Token Embedding）：**把每个词转换成计算机能看懂的 “数字向量”，比如 “小明”→[0.2, 0.5, -0.1]，“操场”→[0.3, 0.1, 0.7]。

**2. 注意力权重计算：**模型计算 “每个词和其他词的关联程度”，用权重值表示。比如：

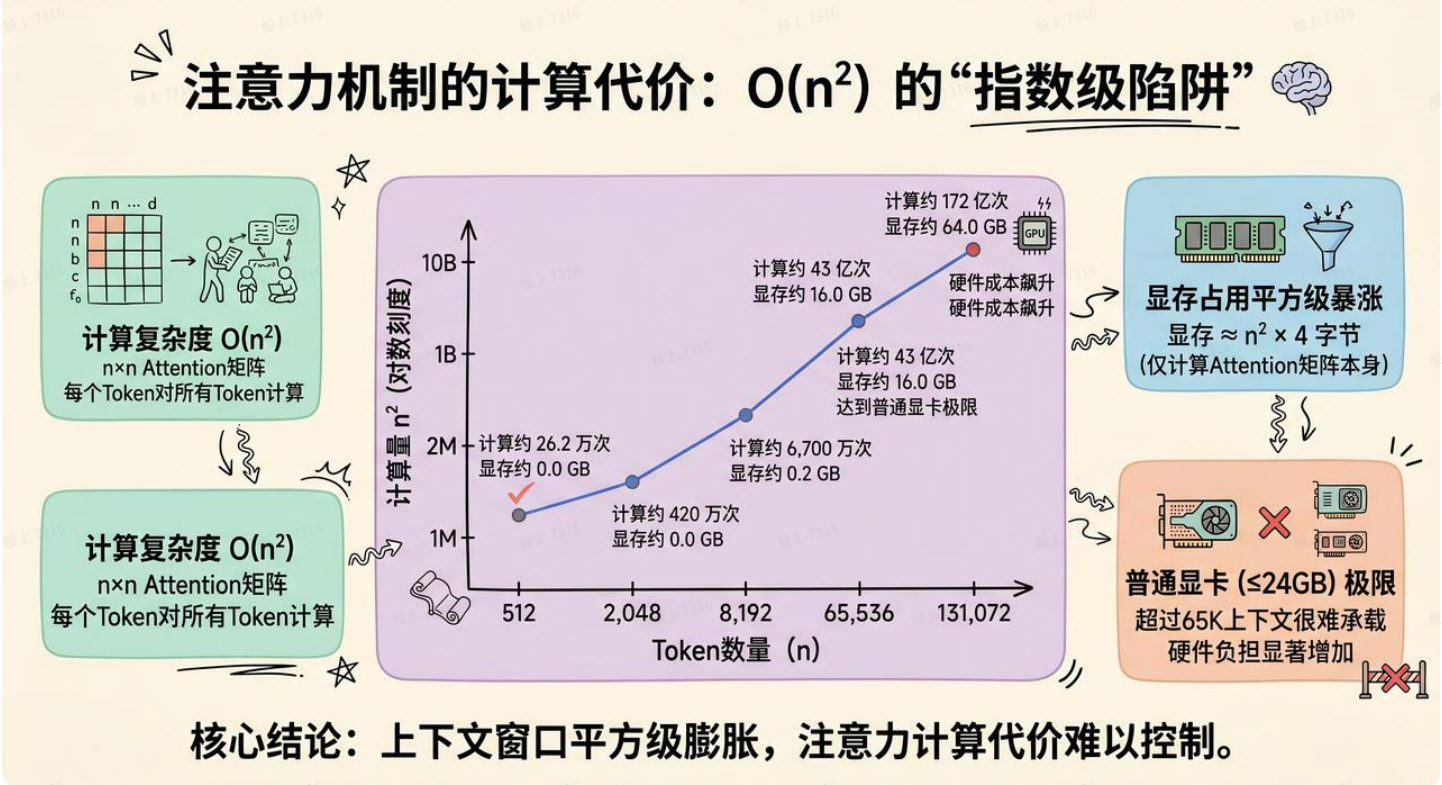
- “___” 和 “操场” 的权重是 0.9（因为操场是活动地点）；
- “___” 和 “天气很好” 的权重是 0.8（因为天气好适合户外活动）；
- “___” 和 “小明” 的权重是 0.7（因为小明是动作的执行者）。

**3. 加权求和：**把高权重词的向量 “加起来”，得到 “___” 的最终向量，然后模型根据这个向量预测出 “跑步” “踢球” 这类词。

**总结：**自注意力机制让模型“知道该关注哪些词”，从而理解文本的逻辑关系。同时我们可以看出，大模型本质上是基于统计的参数化概率模型，擅长相关性，不天然具备严格的因果理解。

# 2. 上下文窗口

## 2.1 为什么上下文窗口不能无限扩大




上下文窗口的本质是注意力计算的范围枷锁，为什么模型不能无限制扩大上下文窗口？核心是注意力计算的复杂度陷阱。

- 我们先明确一个公式：自注意力的计算复杂度 = $O(n^2)$
- 其中 n 是输入文本的 token 数量，这个公式的意思是：计算量会随着 token 数量的增加呈平方级爆炸增长。

我们用具体数字对比，直观感受这个计算量：

| token 数量 n | 计算量 $n^2$ | 所需显存（假设每个元素 4 字节） | 普通显卡能否支撑 |
|---|---|---|---|
| 512 | 262,144 | 262144 × 4 = 1,048,576 字节 ≈ 1MB | 轻松支撑 |
| 2048 | 4,194,304 | ≈ 16MB | 轻松支撑 |
| 8192 | 67,108,864 | ≈ 262MB | 勉强支撑 |
| 65536 | 4,294,967,296 | ≈ 16.4GB | 只有高端显卡能支撑 |
| 131072 | 17,179,869,184 | ≈ 65.5GB | 普通显卡直接 “罢工” |

这就是数学边界的核心枷锁：哪怕你想把窗口从 8192 扩容到 131072，计算量会从 6700 万暴涨到 170 亿，显存需求从 262MB 涨到 65.5GB，普通硬件根本扛不住。

## 2.2 上下文窗口越大，放入越多内容就越好吗？

窗口变大，不等于模型就能稳定理解和利用全部内容。

随着上下文不断变长，模型往往会出现“约束遗忘”和“智能下降”：前面写过的规则更容易失效，回答也更容易偏题、变浅、抓不住重点。

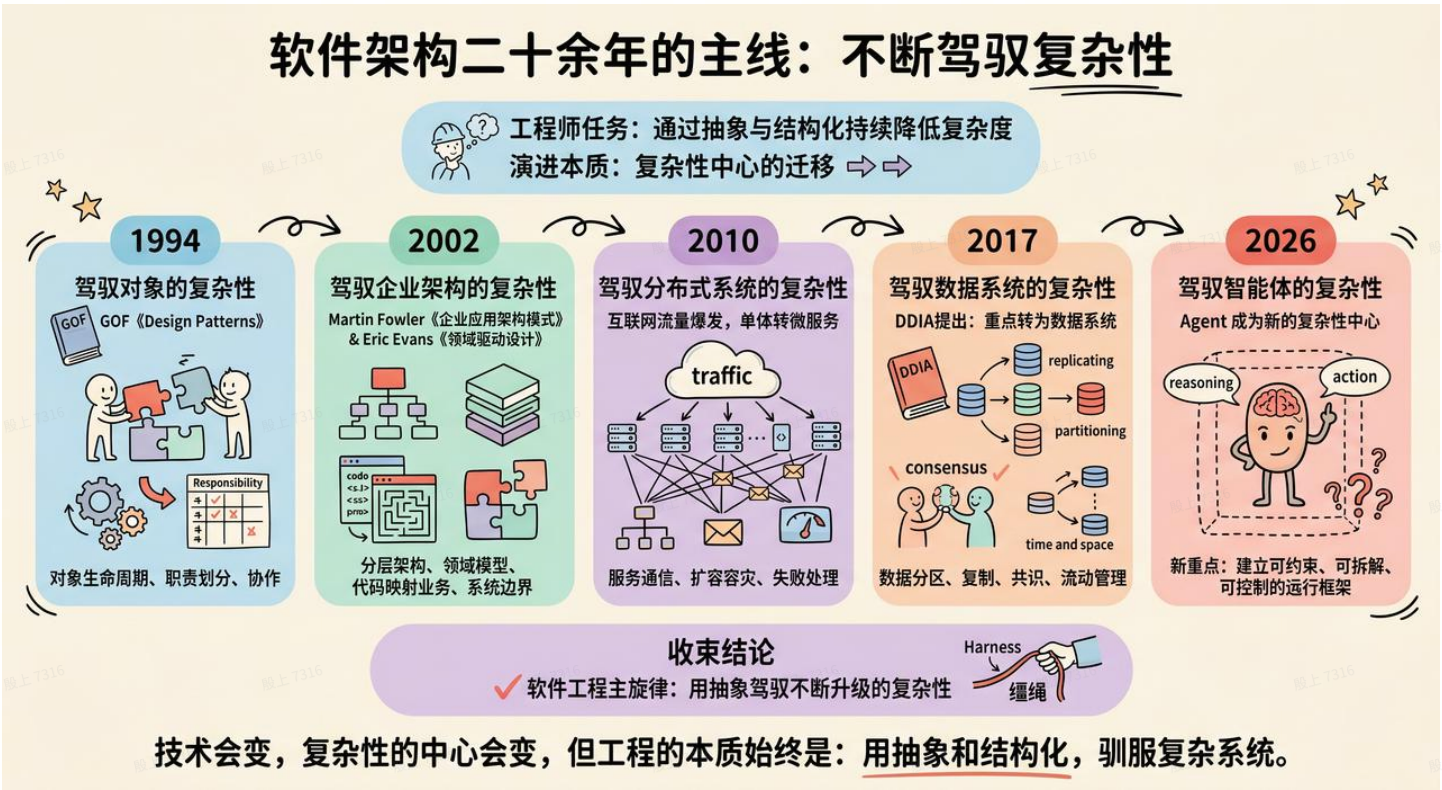
原因在于，大模型并不是像数据库一样“精确存储并调用前文”，而是在生成每个 token 时，**动态地从整段上下文中分配注意力**。当内容越来越多时，会出现几个典型问题：

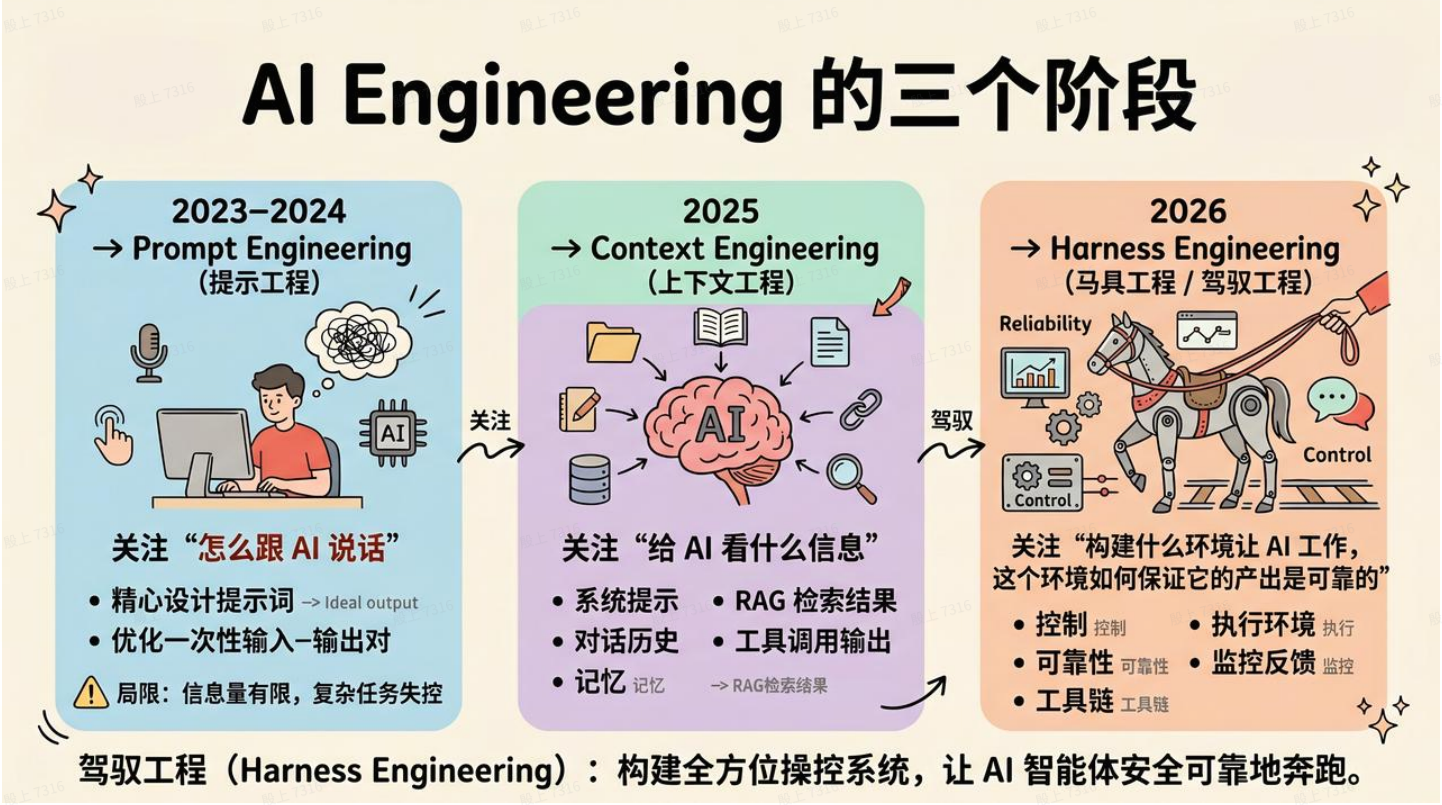
1. **注意力被稀释**：前面写入的规则虽然还在窗口里，但在大量新内容冲刷下，权重会不断下降。
2. **模型更偏向最近信息**：离当前问题更近的内容，通常更容易在注意力竞争中占优。
3. **中间区域最容易被忽略**：在长上下文中，开头的内容往往还能作为初始任务背景保留一定权重，结尾的内容因为离当前生成位置更近也更容易被关注，反而是中间那一大段信息最容易“掉”——看似放进去了，实际利用率却不高。
4. **信噪比下降**：上下文越长，无关信息、重复信息、弱相关信息越多，模型筛选重点和维持推理一致性的难度越大。

因此，长上下文带来的并不只是“能装更多内容”，还会带来一个副作用：
**信息越多，模型越容易失焦；约束越长，模型越容易遗忘；而且中间区域的信息最容易在注意力竞争中被淹没。**

# 三、大模型应用的演进

## 1. 历史演变






## 2. 什么是Harness

你可以把一个 Agent 系统想象成一台电脑。

模型是 CPU，提供原始算力。上下文窗口是内存，临时存点东西，关机就没了。Agent 是跑在上面的应用程序。

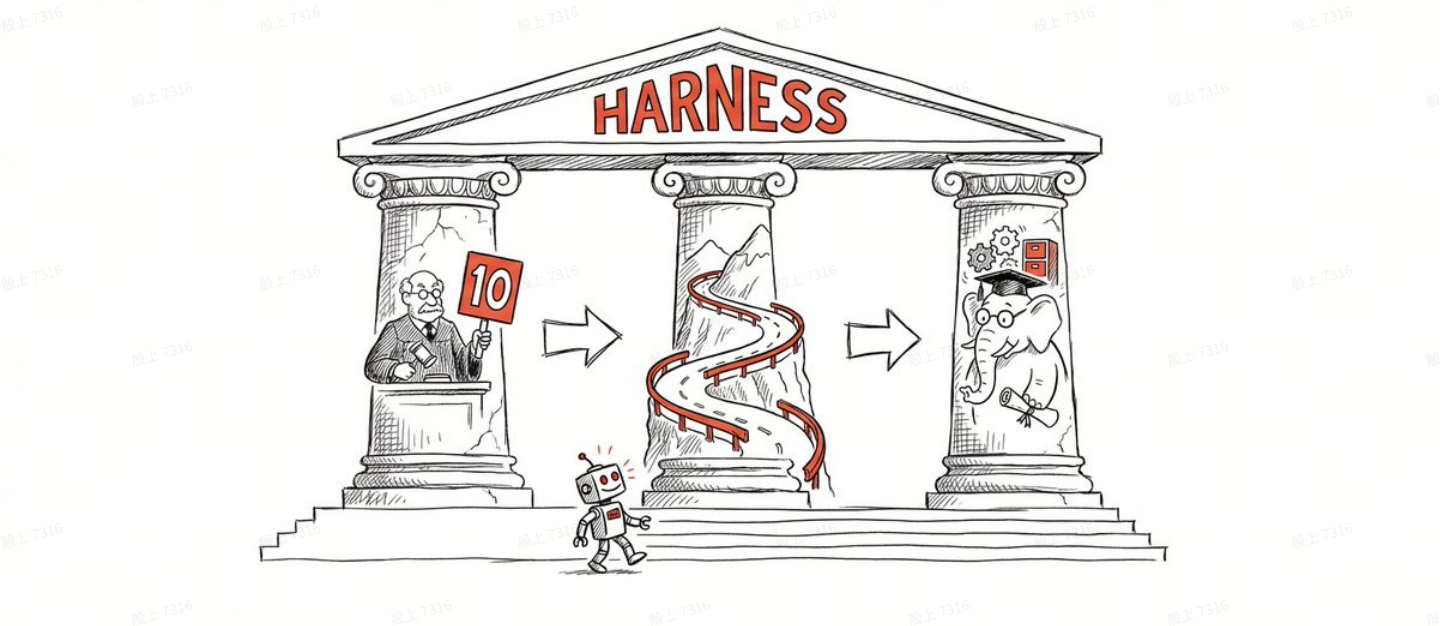
那操作系统呢？

**Harness 就是操作系统。**

没有操作系统，CPU 再猛也只是一块芯片。你总不能对着芯片敲键盘吧。

同样的道理，没有 Harness，模型再聪明也只是一个聊天框。你让它跑一个小时的复杂任务，它中间忘了上下文怎么办？它写了一堆垃圾代码谁来拦？它犯了错自己都不知道怎么办？

这些问题，都不是"换一个更聪明的模型"能解决的。




好，Harness 到底包含什么？核心就三件事：**评估闭环、架构约束、记忆治理**

**1. 架构约束**

先把规则立住，让 Agent 清楚什么能做、什么不能做。这里包括架构边界、代码规范、权限范围，以及 CI、Linter 等硬性约束。关键不只是“写在文档里”，而是要把这些规则显式化、工具化、自动执行，避免 Agent 越界发挥。

**2. 执行闭环**

Harness 不是让 Agent “写完就算”，而是让它在一个可控流程里持续运行：任务拆解、上下文注入、工具调用、测试校验、审查反馈、错误修正、PR 与合并流程，以及必要的人类介入点。核心目标是把 Agent 从“单次回答”变成“在反馈中不断修正的执行者”。这也是所谓的“人类掌舵，Agent 执行”。

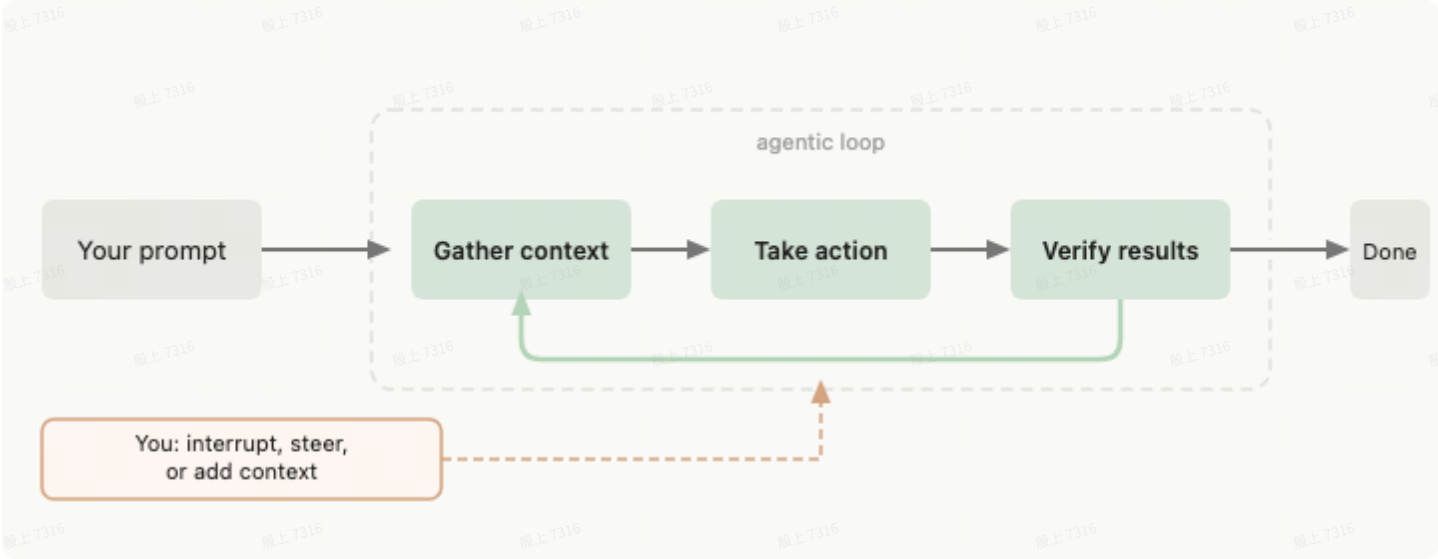
**3. 记忆治理**

每次 Agent 出错，都不应只停留在当次修复，而应把经验沉淀为长期资产：文档、规则、测试用例、检查器、操作规范。这样，同类问题以后会越来越少，系统会随着使用不断变得更稳、更可靠。

**所以，Harness Engineering 并不是去优化模型本身，而是在模型之外设计一套可运行、可约束、可反馈、可沉淀的工程环境，让 Agent 能稳定、可靠地完成工作。**

# 四、claude code

## 1. 底层的运行逻辑




Claude Code 的核心不是"回答"，而是一个反复循环的代理过程：

代码块

```
收集上下文 → 采取行动 → 验证结果 → [完成 or 回到收集]
     ↑                      ↓
  CLAUDE.md          Hooks / 权限 / 沙箱
  Skills             Tools / MCP
  Memory
```

## 2. 上下文工程：最重要的系统约束

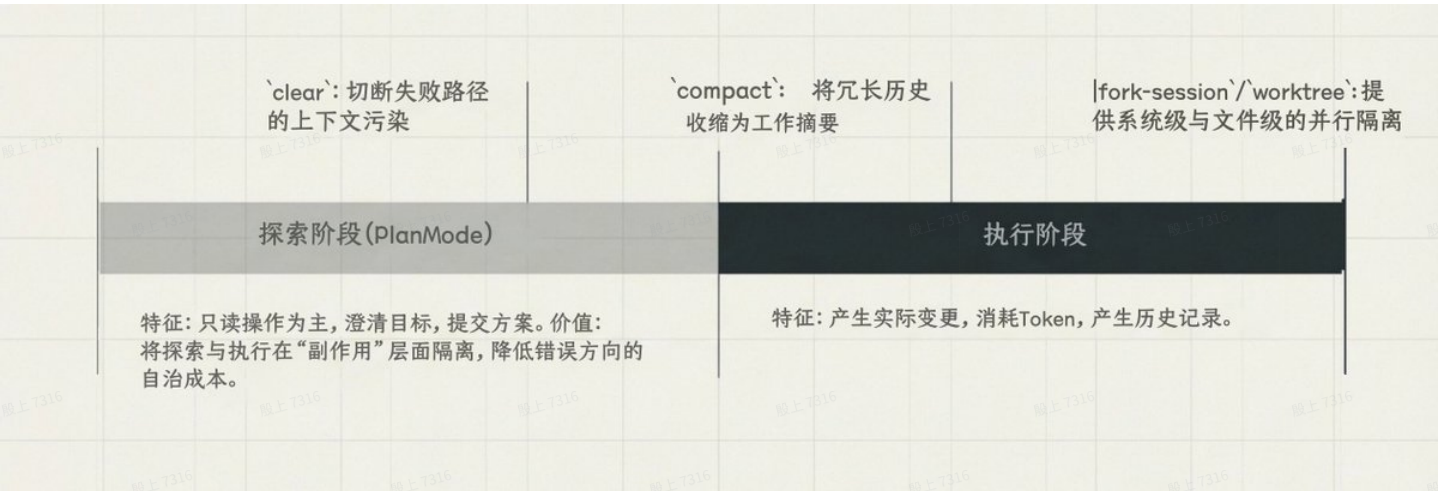
Claude Code 的 200K 上下文并非全部可用：

代码块

```
200K 总上下文
├── 固定开销 (~15-20K)
│   ├── 系统指令: ~2K
│   ├── 所有启用的 Skill 描述符: ~1-5K
│   ├── MCP Server 工具定义: ~10-20K  ← 最大隐形杀手
│   └── LSP 状态: ~2-5K
│
├── 半固定 (~5-10K)
│   ├── CLAUDE.md: ~2-5K
│   └── Memory: ~1-2K
│
└── 动态可用 (~160-180K)
    ├── 对话历史
    ├── 文件内容
    └── 工具调用结果
```


一个典型 MCP Server（如 GitHub）包含 20-30 个工具定义，每个约 200 tokens，合计 **4,000-6,000 tokens**。接 5 个 Server，光这部分固定开销就到了 **25,000 tokens（12.5%）**。

## 3. Plan Mode 的工程价值




Plan Mode 的核心是把探索和执行拆开，探索阶段不动文件，确认方案后再执行：

- 探索阶段以只读操作为主
- Claude 可以先澄清目标和边界，再提交具体方案
- 执行成本在计划确认之后才发生

## 4. Subagents：派一个独立的 Claude 去干一件具体的事

Subagent 就是从主对话派出去的一个独立 Claude 实例，有自己的上下文窗口，只用你指定的工具，干完汇报结果。我用下来觉得它最大的价值不是"并行"，而是隔离，扫代码库、跑测试、做审查这类会产生大量输出的事，塞进主线程很快就把有效上下文挤没了，交给 Subagent 做，主线程只拿一个摘要，干净很多。

Claude Code 内置了三个：**Explore**（只读扫库，默认跑 Haiku 省成本）、**Plan**（规划调研）、**General-purpose**（通用），也可以自定义。

**配置时要显式约束**

- tools / disallowedTools：限定能用什么工具，别给和主线程一样宽的权限

- model：探索任务用 Haiku/Sonnet，重要审查用 Opus
- maxTurns：防止跑飞
- isolation: worktree：需要动文件时隔离文件系统

另一个实用细节：长时间运行的 bash 命令可以按 Ctrl+B 移到后台，Claude 之后会用 BashOutput 工具查看结果，不会阻塞主线程继续工作。subagent 同理，直接告诉它「在后台跑」就行。

**几个常见反模式**

- 子代理权限和主线程一样宽，隔离没有意义
- 输出格式不固定，主线程拿到没法用
- 子任务之间强依赖，频繁要共享中间状态，这种情况用 Subagent 不合适

## 5. Skills 设计：不是模板库，是用的时候才加载的工作流

Skill 官方描述是"按需加载的知识与工作流"，描述符常驻上下文，完整内容按需加载，用起来和"保存的 Prompt"差别挺大的。

**一个好 Skill 应该满足什么？**

- 描述要让模型知道"何时该用我"，而不是"我是干什么的"，这两个差很多
- 有完整步骤、输入、输出和停止条件，别写了个开头没有结尾
- 正文只放导航和核心约束，大资料拆到 supporting files 里
- 有副作用的 Skill 要显式设置 disable-model-invocation: true，不然 Claude 会自己决定要不要跑

**Skill 怎么做到按需加载**

Claude Code 团队在内部设计中反复强调 **"渐进式披露"**，意思不是让模型一次性看到所有信息，而是先获得索引和导航，再按需拉取细节：

- SKILL.md 负责定义任务语义、边界和执行骨架
- supporting files 负责提供领域细节
- 脚本负责确定性收集上下文或证据

一个比较稳定的结构长这样：

代码块

```
.claude/skills/
└── incident-triage/
    ├── SKILL.md
    ├── runbook.md
    ├── examples.md
    └── scripts/
        └── collect-context.sh
```

**类型一：检查清单型（质量门禁）**

发布前跑一遍，确保不漏项：

代码块

```
---
name: release-check
description: 在发布前使用，用于验证构建、版本号和冒烟测试。
---
## 发布前检查（以下项目必须全部通过）
- [ ] `cargo build --release` 通过
- [ ] `cargo clippy -- -D warnings` 无警告
- [ ] `Cargo.toml` 中的版本号已更新
- [ ] `CHANGELOG` 已更新
- [ ] 在干净环境下执行 `kaku doctor` 通过

## 输出
逐项给出 Pass / Fail。只要有任一项 Fail，都必须先修复后再发布。
```

**类型二：工作流型（标准化操作）**

配置迁移高风险，显式调用 + 内置回滚步骤： 这种执行频率较低，直接把disable-model-invocation设置为true

代码块

```
---
name: config-migration
description: 迁移配置 schema。仅在被明确要求时运行。
disable-model-invocation: true
---
## 步骤
1. 备份：`cp ~/.config/kaku/config.toml ~/.config/kaku/config.toml.bak`
2. 试运行：`kaku config migrate --dry-run`
3. 应用：确认输出无误后，去掉 `--dry-run`
4. 验证：`kaku doctor` 全部通过

## 回滚
`cp ~/.config/kaku/config.toml.bak ~/.config/kaku/config.toml`
```

**类型三：领域专家型（封装决策框架）**

运行时出问题时让 Claude 按固定路径收集证据，不要瞎猜：

代码块

```
---
name: runtime-diagnosis
description: 当 kaku 在运行时崩溃、卡死或出现异常行为时使用。
---
## 证据收集
1. 运行 `kaku doctor` 并完整保存输出结果
2. 获取 `~/.local/share/kaku/logs/` 中最后 50 行日志
3. 插件状态：`kaku --list-plugins`

## 决策矩阵
| 症状 | 首先检查 |
|---|---|
| 启动时崩溃 | doctor 输出 → Lua 语法错误 |
| 渲染异常 | GPU 后端 / 终端能力 |
| 配置未生效 | 配置路径 + schema 版本 |

## 输出格式
根因 / 影响范围 / 修复步骤 / 验证命令
```

描述符写短点，每个 Skill 都在偷你的上下文空间，每个启用的 Skill，描述符常驻上下文，优化前后差距很大：

代码块

```
# 低效（~45 tokens）
description: |
  这个 skill 用于帮助你审查 Rust 项目的代码变更。
  它会检查一些常见问题，例如 unsafe 代码、错误处理等……
  当你希望在合并前确保代码质量时使用它。

# 高效（~9 tokens）
description: 用于 PR 审查，重点关注正确性。
```

还有一个很重要的 disable-auto-invoke 使用策略：

- 高频（>1 次/会话）→ 保持 auto-invoke，优化描述符
- 低频（<1 次/会话）→ disable-auto-invoke，手动触发，描述符完全脱离上下文
- 极低频（<1 次/月）→ 移除 Skill，改为编写单独md文档，需要使用时通过@调用

**Skills 反模式**

- 描述过短：description: help with backend（任何后端工作都能触发，哈哈）
- 正文过长：几百行工作手册全塞进 SKILL.md 正文
- 一个 Skill 覆盖 review、deploy、debug、docs、incident 五件事

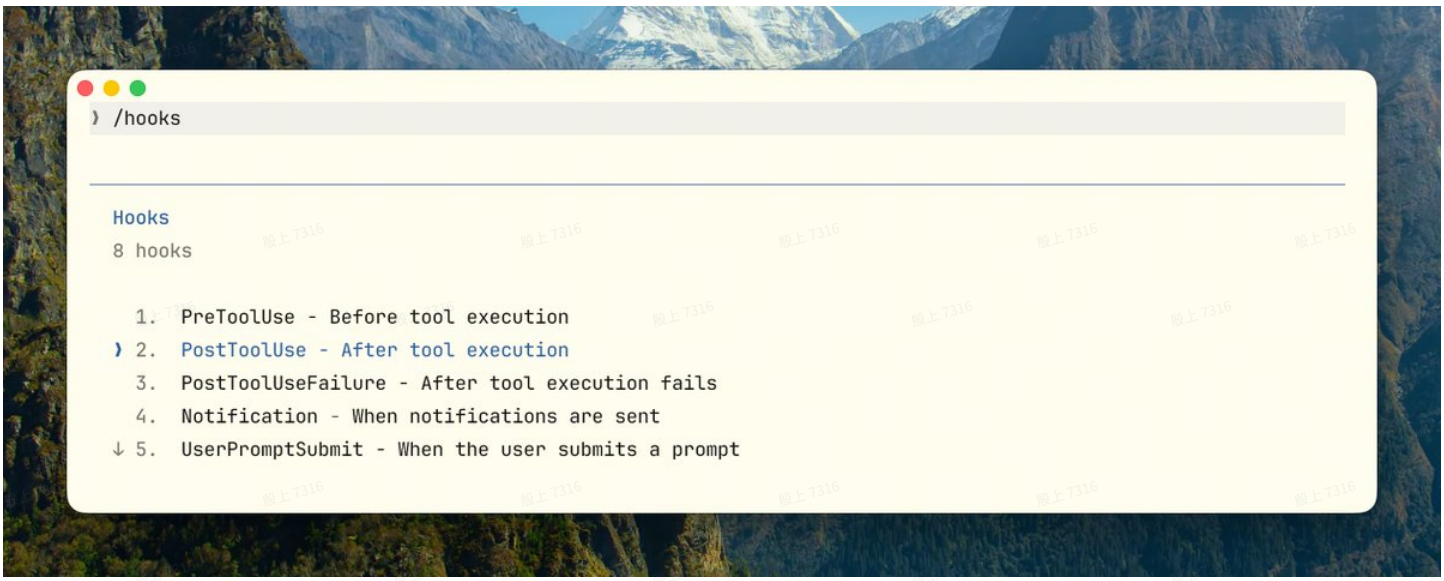
- 有副作用的 Skill 允许模型自动调用

## 6. Hooks：在 Claude 执行操作前后，强制插入你自己的逻辑

Hooks 很容易被当成"自动运行的脚本"，但我自己用下来，觉得它更像是把一些不能交给 Claude 临场发挥的事情，重新收回到确定性的流程里。

比如格式化要不要跑、保护文件能不能改、任务完成后要不要通知，这些事真不要指望 Claude 每次都自己记得。

当前支持的 Hook 点




**适合 vs 不适合放到 Hooks 的**

**适合**：阻断修改受保护文件、Edit 后自动格式化/lint/轻量校验、SessionStart 后注入动态上下文（Git 分支、环境变量）、任务完成后推送通知。

**不适合**：需要读大量上下文的复杂语义判断、长时间运行的业务流程、需要多步推理和权衡的决策，这些该在 Skill 或 Subagent 里

## 7. 上下文压缩

默认压缩算法按"可重新读取"判断，早期的 Tool Output 和文件内容会被优先删掉，顺带把**架构决策和约束理由**也一起扔了。两小时后再改，可能根本不记得两小时前定了什么，莫名其妙的 Bug 就是这么来的。

- Continue，继续在同一个会话里发下一条消息
- /rewind（esc esc），跳回之前某条消息，从那里重试。出现错误比较大的方向性错误，优先使用 rewind，与其纠错，不如回退
- /clear，开始新会话，通常带上你刚提炼出来的简报

- Compact，总结当前会话并在总结之上继续。指明需要保留的方向，提前使用，不要等工具自动压缩
- Subagents，把下一段工作委托给有独立干净上下文的代理，只把结果拉回来