



# 深度研究：使用 Cursor 编辑器和 Claude Opus 4.5 开发复杂项目最佳实践

## 高效使用 Cursor + Opus 进行复杂系统开发

**完善项目前期规划与上下文提供：**在开发“韬睿量化系统插件”等复杂项目时，先利用强大的模型（如 Claude 4 with search）进行充分的架构规划非常重要<sup>1</sup>。将项目拆解成模块，明确每个模块的功能与接口，并生成一份全面的项目设计说明（例如 README 规划文档）。这份 README 将作为项目的“单一事实源”（Single Source of Truth），记录所有架构决策、文件结构和接口设计<sup>2</sup>。之后，将该 README 导入 Cursor 的对话中，让 Opus 理解整个项目蓝图和约定，再开始具体编码<sup>3</sup>。通过这种方式，AI 助手在生成代码前就 **全面了解项目现状和目标**，避免凭空臆测模块或偏离设计初衷。

**利用单一事实源保持长期一致性：**始终以 README 或设计文档为准绳，每个新功能或修改都应追溯到这份“黄金文档”<sup>4</sup>。在编码过程中，可以反复让 Cursor 对照 README，确认新文件位置正确、遵循了既定架构<sup>5</sup>。例如，添加新模块后，询问 AI “该文件的位置是否符合架构设计？”以防止模块放错目录或出现架构漂移<sup>5</sup>。同时，将重要的全局约定（命名规范、关键接口定义等）写入规则文件或项目文档中，让模型**每次对话都加载这些统一规则**<sup>6</sup>。Cursor 支持编写 .mdc 规则文件，为不同场景提供详细指南，以确保模型严格按照您的规范行事<sup>6</sup>。通过维护**单一可靠的信息源**并在每次对话中提供或引用它，可以在不依赖过往聊天记录的情况下，让 Opus 始终遵循既定的模块命名、接口设计和调用链规范。

**避免上下文遗失和重复生成：**由于 Cursor 当前对话历史无法在会话间永久保存<sup>7</sup>，建议将关键讨论内容手动保存到文档（如更新 README 或注释）以备后续引用。当重新启动 Cursor 或长时间未互动后，应主动提供项目概要（如文件列表和模块职责）以重建上下文，从而避免 AI 遗忘先前决定而重复生成已有模块<sup>8</sup>。Cursor Claude 4 提供了“**内存文件**”等增强记忆功能，会自动记录长期项目信息<sup>9</sup>。充分利用这些特性，或定期让 AI 总结当前项目状态写入备忘文档，能帮助模型在跨会话时保持**会话连续性和项目感知**<sup>10</sup>。总之，每次新对话都明确告知模型当前项目结构和已完成部分，确保其在**无历史记录依赖**下仍能持续一致地扩展代码。

**模块化任务输入与接口约定：**为防止 Opus 误生成不需要的模块，向其提供清晰的任务描述和相关代码片段十分关键<sup>11</sup>。在每个任务开始前，指出**应修改或扩展的现有模块**，以及**禁止新建重复模块**。如果要新增组件，也应在提示中说明其与现有系统的集成关系，从而让模型理解新代码应如何衔接已有调用链。您可以利用 Cursor 的 chat 模式（Ctrl+L）先与模型讨论实现思路，将涉及的代码片段、接口定义一并提供，然后再切换到 agent 模式执行修改<sup>12</sup>。确保**聊天上下文和代理执行共享信息**（Cursor 有相应的共享上下文设置<sup>13</sup>），这样AI在应用改动时不会偏离之前讨论的方案。通过细粒度地**划分任务**、逐步集成，每一步都让 AI 明确“基于现有代码 X 进行改造”而非凭空造轮子，可显著提高复杂系统开发的一致性和准确性。

## 控制 Token 消耗与调用次数的策略

**上下文缓存与重用：**重复的上下文分析会浪费大量 token。例如实现一个功能通常需要**分析代码→编写代码→验证代码**三个步骤，每一步AI都可能重新解析整个项目，导致同一上下文被分析三次<sup>14</sup> <sup>15</sup>。针对这种情况，可采用“**上下文缓存**”策略：将首次耗费大量 token 的分析结果保存下来，供后续步骤复用，从而避免重复解析<sup>16</sup>。实

测显示，使用缓存后，后续步骤可减少约30%的上下文Token，每个功能总体Token消耗降低约20%<sup>17</sup><sup>18</sup>。实现方法可以是将分析所得的文件依赖、架构要点、测试计划等数据序列化保存，当AI执行实现或验证步骤时，将缓存内容注入提示，告知它“请使用缓存上下文，勿重复分析已知部分”<sup>19</sup>。通过这种方式，像新功能的接口依赖、已有模块信息都能一次分析，多次使用，大幅减少Token浪费。

**精简上下文与结构化提示：** 避免每次调用都附带整个代码库作为上下文。应根据任务相关性**挑选必要的文件片段**提供给AI<sup>20</sup>。Cursor 社区有开发者反映，一次小改动如果让Cursor自动选择过多文件，可能导致十几倍于文件大小的Token消耗<sup>21</sup>。因此要谨慎控制上下文范围：如编辑某模块函数，只提供该文件和直接相关的接口定义，而非整个项目。对于大型文件，可提前让AI**生成摘要**或提取关键数据结构，以较短的描述代替原文件，提高上下文利用率。还可以利用 Cursor 提供的“快速编辑”模式或精简模型，降低对长上下文的依赖。**结构化Prompt**也是降低重复沟通的利器——准备一套标准提示模板（Snippets），涵盖角色设定、任务要求和上下文要点<sup>22</sup>。例如，代替随意提问，用模板明确指出“**功能X所需的架构检查、依赖分析、验证项**”，并引用缓存的上下文ID<sup>23</sup><sup>24</sup>。标准化的提示不仅减少遗漏（避免来回追问），还可重复复用，从而**减少无效调用**。实际示例对比显示，结构化提示比自由提问得到的答案更完整有序，同时充分利用了缓存信息<sup>23</sup><sup>24</sup>。

**上下文分块与渐进提供：** 面对超长的上下文，最佳做法是**分而治之**。将长对话或长文档拆解为逻辑板块，每块单独提供并与AI讨论，然后将精炼的结果用于后续步骤。这种“短上下文块+缓存”的方式一方面确保每次提示简明、聚焦，另一方面通过缓存串联起整个长流程。比如在一个100K Token上下文的任务中，可先让AI分析概要，再分段深入各模块细节，各段分析结果写入缓存供整合使用。这样既避免单次调用超出模型上下文窗口，也降低了一次性调用的Token上限风险。Cursor也在完善**上下文裁剪和提示优化机制**：有帖子指出Cursor在Apply代码时通过特殊模型避免重复生成现有代码，从而减少输出Token<sup>25</sup>。作为用户，也应留意**模型的回应长度**，适当引导其简洁回答或分步输出，防止因为输出冗长而浪费Token。

**按需调用与成本控制：** 在Cursor中启用**Auto模式**或使用Opus等高级模型需注意按需使用，以免超额成本。Auto模式会根据任务自动切换模型，但有用户反馈它往往倾向于更便宜的旧模型，导致回答质量下降<sup>26</sup>。为保持质量，可手动选择可靠模型用于关键步骤，或确保提供足够上下文和规则帮助较廉价模型也能正确发挥<sup>27</sup>。Opus 4.5 拥有超强长时推理能力，但其每百万Token成本较高<sup>28</sup>。建议将Opus用于复杂架构改造、长时间调试等高难度环节<sup>29</sup>，平时的简单生成或日常编码改用成本低很多的Sonnet 4等<sup>30</sup>。此外，善用**Thinking Mode**（深度思考模式）的开关：遇到疑难再启用，平常关闭以节省请求配额<sup>31</sup>。监控Cursor仪表板或日志中的Token使用情况，一旦发现某次请求消耗异常巨大，应暂停并分析原因（是否引入了不必要的大段上下文或循环调用）<sup>32</sup>。设定月度预算提醒，在接近配额或费用上限时切换到手动模式，避免因连续自动调用导致“烧币”过多而触发溢出风险。总之，通过**按需选模型、控制调用频率和上下文大小**，可以在不牺牲开发质量的前提下，将Token消耗降至最低<sup>33</sup>。

## 避免常见问题：策略与应对

**模块重复生成与架构漂移：** 在大型项目中，若AI没有清晰的全局视图，可能会重复创建已有功能模块或者偏离原有架构设计。这通常表现为代码冗余（出现两个类似的类/函数）或层次错乱（逻辑放在错误的层级）。为避免此问题，务必**强化AI对现有模块的认知**：在提示中列出当前模块清单或架构图，强调“不得重复实现已存在的模块X”以及系统各部分的职责边界。Cursor Agent 的最佳实践是“**拓展而非复制**”：要求AI尽量在已有代码基础上扩展功能，而非另起炉灶<sup>34</sup>。如果怀疑出现了重复模块，立即对照README核查，每个新文件都应能在README的架构树中找到对应位置<sup>35</sup>；若找不到，就可能是冗余的，应让AI修改为使用已有模块。利用**规则文件**可以提前防范这类问题，例如设置准则“**禁止启动平行的重复实现，所有新特性应集成到现有代码路径**”<sup>36</sup>。同时，坚持DRY原则，在抽象层集中共享逻辑，也能减少AI编出重复代码的可能<sup>37</sup>。

**接口不一致与调用链断裂：** 复杂系统中接口定义众多，AI有时会擅自更改函数签名或使用错误的参数，导致模块间对接失灵。为此，**应集中管理接口规范**：把关键数据结构和API接口定义收录在设计文档或Cursor的全局上下文中，每次生成相关代码时都提供这些定义作为参考。从社区经验看，**让AI自行校对接口**是有帮助的步骤：生成完模块后，询问“请检查新代码是否与现有接口定义一致”，督促模型对比新旧接口并修正不匹配之处。此外，在引入新模块前，可要求AI**给出模块与系统其它部分交互的清单**（函数调用链或序列图），确保它没遗漏调用点。测试驱动开发也是防止接口不符的有效手段：为主要接口预先编写单元测试或集成测试，如果AI的实现未通过测试，说明接口契约未兑现，需进一步调整<sup>35</sup>。通过**文档约束+生成后校验+测试验证**三管齐下，可以最大限度保证接口设计的一致性。

**代码未集成导致功能断链：** Agent 有时会生成某段代码却**没有将其接入整体流程**，比如写了新函数但没有在任何地方调用，或创建了插件类但未注册。这种功能断链的问题需要在生成后认真审查和补救。对策首先是在**提示中要求全流程考虑**：例如在让AI创建新功能时，加一句“完成实现后，将其整合进现有工作流并演示调用”。社区有高手建议在Auto模式生成代码前，**让Agent为每个待写的文件先生成一个计划**，说明该文件如何被使用<sup>36</sup>——这样AI在动手编写时就清楚代码的用途，减少漏调情况。若发现AI遗漏集成，可进一步提示：“代码已生成，下一步请将新功能接入应用：例如在主程序初始化时调用它”，督促AI补齐调用关系。实践中，**一定要测试新功能的实际运行路径**：比如如果是后端接口，启动服务实际发一次请求看是否触发新逻辑；如果是插件，看主系统是否正确加载。必要时，人为地将生成的代码插入调用链，然后反馈AI遇到的问题，让其修正。**持续的集成验证**可以避免静态代码看似完整却无法工作的情况。

**扩展插件与主系统对接问题：** 针对桌面端主程序与Web扩展件之间的对接失败，通常原因是两端的接口契约不一致或通讯流程出错。为降低风险，需提前**明确主系统与插件的交互协议**，并让AI严格按照协议实现扩展件。例如，如果主系统通过IPC或API调用插件，那么在让AI编写插件代码时，应提供主系统的相关接口定义和调用示例作为参考，使其对接时方法名、数据格式都完全匹配。开发过程中，可要求AI模拟一次主程序与插件的交互流程，输出预期的调用顺序和数据流，以检查理解是否正确。如有可能，**将主系统和插件放在同一上下文**让AI同时考虑（例如提供主系统代码片段和插件模版一起让其修改），保证AI在写插件时直接调用主系统的实际函数。完成插件开发后，一定进行**综合测试**：启动主应用加载插件，看是否报错或行为异常。如果有问题，再让AI分析日志或错误信息，并根据主-插件协议调整实现。通过**契约驱动开发和严格测试**，可以显著减少插件集成失败的情况。

**会话中断与状态丢失：** 当Cursor窗口重启或项目重新载入时，先前的对话状态可能丢失，AI不再“记得”哪些模块已完成或设计过哪些方案<sup>37</sup>。为防止因此导致的模块失联或决策遗忘，开发者需要**主动保存和恢复上下文**。具体做法包括：定期将AI对话记录或关键结论复制到项目文档（如维护一个CHANGELOG或设计笔记）；重启后，首先向AI重述项目概况和进展，让其载入相关文件摘要。Cursor 提供导出聊天记录为 Markdown 的功能，或可借助 SpecStory扩展自动保存每次对话<sup>37</sup>。善用这些工具，在意外中断后快速恢复AI “记忆”。如果某模块在中断后AI不再关注（“失联”），可以通过提示提醒它：“项目已有模块A/B，请勿重复，实现新功能时应调用它们”。**主动回灌关键信息**是跨会话一致性的保障。另一方面，**尽量减少一次对话跨度**也是策略之一：将任务划分为多次较短的对话，每次完成后都落实到代码文件中（这种情况下即使历史丢失也无碍，因为代码已落地成事实）。总之，准备好**断点续传**的方案，把AI能忘的全部记录下来，下次再输入，让复杂项目的开发不会因一次重启而推倒重来。

**Diff 应用失败与版本回滚：** Cursor 的 Agent 模式通常通过 diff 来应用代码更改，但在实践中有时出现**应用不完全或格式错乱**的问题<sup>38</sup>。遇到这种情况，不要贸然“Accept All”合并所有改动<sup>39</sup>；应先检查每个diff块的正确性。如果AI修改中途停滞，贸然接受可能引入不完整代码<sup>39</sup>。正确做法是利用Cursor的**“Restore checkpoint”**功能或手工撤销更改<sup>40</sup>。为防范代码丢失和无法回滚，**养成使用Git的习惯**：在让AI进行大量改动前先提交当前版本，并在每轮大改后再次提交<sup>41</sup>。这样一旦AI改动出错，可以轻松还原到之前的稳定状态。社区开发者形象地比喻：Git提交就是AI编程的“存档点”，当AI乱改代码时，它是你最后的保险<sup>41</sup>。此外，Cursor本身在快速迭代，**代码应用可靠性**是官方关注的重点之一<sup>40</sup>。及时升级Cursor版本，可获得对diff应用问题的最新修复。对于“自动保存缺失”的忧虑，手动保存文件仍是最稳妥的方法——确认代理改动已正确写入文件后再进行下一步，避免因为Cursor界面卡

顿或Bug导致更改未写入磁盘。归根结底，通过**小步提交、勤备份、多验证**，可以最大限度避免AI改动引入的严重问题，并在需要时从容回退代码。

## 社区反馈与实践经验汇总

**Auto模式可靠性争议：** Cursor 社区中，不少用户对Auto模式的表现提出质疑。一些开发者发现，当高级请求用完转入Auto模式后，AI往往给出错误百出的回应，无法正确修复Bug或添加功能，频繁出现荒谬的错误<sup>42</sup>。有人直言“Auto模式没什么用处”，认为它调用的是**能力较弱的模型**，导致回答不仅过时而且伴随幻觉<sup>26</sup>。也有声音指出，Cursor的Auto模式倾向于使用**成本更低的模型**以节省费用，但上下文和准确性也随之降低<sup>43</sup>。因此资深用户更倾向**手动选择模型**，在需要高可靠性时宁可消耗高级请求，也避免Auto模式乱给建议。当然，也有人提出Auto模式下只要提供足够详细的规则和Prompt，即使旧版模型也能有不错表现<sup>27</sup>。总体来说，社区对Auto模式褒贬不一，但共识是：**了解Auto模式的取舍**（低成本 vs. 低智商），重要任务上谨慎使用。

**Claude Opus 4.5 的强大与代价：** Cursor 第一时间集成了Anthropic的Claude 4模型，开发者对其在编程任务中的表现给予高度评价<sup>44</sup>。Claude Opus 4.5 被誉为当前“**世界上最强的编码模型**”，在多文件理解和持续推理上有卓越表现<sup>45</sup>。许多用户分享了Opus处理复杂重构、长时间代码审查的成功案例，认为它对大型代码库的理解力远超以往模型<sup>29</sup>。然而代之而来的是**高昂的调用成本**和较慢的响应。社区建议对此采取**模型混用策略**：即“重活交给Opus，日常交给Sonnet”等较便宜模型<sup>30</sup>。例如一位用户在项目初期用Claude 4详细规划架构，得到高质量的设计方案，但在实现阶段改用3.5或Sonnet完成大部分编码，然后在关键环节再召回Opus审核把关。这样既享受了Opus的智慧，又将Token花费控制在合理范围内<sup>11</sup>。开发者还提醒**关注定价机制**：Cursor提供订阅和按用计费两种模式，Pro套餐含一定量Sonnet请求，而Max模式开放Opus按量计费<sup>46</sup>。根据实际需要选择计费模式，善用每日请求额度，可以降低总体费用。换言之，社区的共识是充分利用Claude 4.5的强项，但**不要滥用**——把它当成狙击手而非机关枪。

**Prompt 编写与规则工程：** 许多使用Cursor的工程师分享了他们在**Prompt Engineering**方面的心得。一大要点是：“**LLM本身不懂你的项目，智能来自于你的指引**”<sup>6</sup>。资深用户强调编写详尽的CursorRules规则文件，越细致越好，用规则把Agent的行为“圈养”起来<sup>47</sup><sup>6</sup>。比如针对单元测试、分层架构、代码风格等，各自制定成文的规范，让AI在生成代码时有据可依<sup>48</sup>。有些开发者甚至采用**自反式迭代**：让AI自己检查规则找漏洞，不断完善，最终规则文件可能长达数百行<sup>49</sup><sup>6</sup>。事实证明，花时间制定规则可以大幅减少AI出错的概率，令其产出的代码更符合预期。另一方面，也有人分享了高效Prompt的套路，例如“**就像带新人编程**”那样给予充分背景和逐步指导。Krzysztof Zabłocki 等开发者建议教会AI遵循你的编码风格<sup>50</sup>、采用团队惯用模式等，以免生成的代码风格割裂。还有人提出**编写提示时应少用否定词**，多用正面引导并解释背后原因<sup>51</sup>——模型据此更能理解规范的重要性而非机械遵从。总结来看，社区达成的共识是：**与其埋怨AI乱写，不如优化自己的提示**。精心设计的Prompt（无论通过规则文件、README还是即时指令实现）可以显著提升Cursor助手的工作质量。

**“Vibe Coding”实践与教训：** 一些开发者利用Cursor实现了惊人的开发效率提升，但他们也分享了走过的弯路和避坑技巧。例如一位用户自称通过“vibe coding”让编码速度提高了10倍，但也警告说：“AI助手能在几分钟内毁掉你数周的成果”<sup>52</sup>。为此他总结了一份**生存指南**，列举了避免灾难的关键措施<sup>53</sup>。其中包括：**绝不跳过架构规划**，一定要在AI写码前敲定整体方案；**使用Git作安全网**，频繁提交以便随时“踩刹车”回滚<sup>41</sup>；在启用自动模式前，先让Agent列出详细的文件实现计划，以防代码“霰弹式”乱飞<sup>36</sup>；快速甄别AI生成代码中的**危险信号**（如出现“dummy”、“mock”字样就要提高警惕，继续迭代直到去除臆造内容）<sup>54</sup>；充分发挥LLM**善于生成文字**的优势，让它为每个模块产出清晰的docstring和注释，确保可维护性<sup>55</sup>。此外，他还建议利用Cursor的**文档索引功能**，将相关框架/库文档添加到环境中，便于AI随时查询<sup>56</sup>；创建新文件后要**核对其是否按README指定的位置存放**，防止架构走样<sup>5</sup>；遇到错误要定位来源而非泛泛要求修复，以免AI南辕北辙<sup>57</sup>。这些一线经验印证了一个结论：

**AI加速开发需要人来掌舵。**只要策略得当、监督到位，Cursor 确实能极大提升生产力；反之忽视管理，AI也可能带来“10倍速踩坑”。社区整体氛围鼓励新人借鉴这些实战技巧，少走弯路，安全高效地拥抱AI编程助理带来的变革。

**持续改进与未来展望：** Cursor 官方和社区都在不断摸索改进之道。从官方路线图来看，未来会有 **更智能的自主 Agent流程** 和 **更完善的长程记忆** <sup>58</sup>。社区用户也积极建言，例如希望更透明的Token用量提示、更高效的上下文处理等 <sup>59</sup> <sup>60</sup>。最近的InfoQ 报道指出，Cursor在平衡**API成本与用户体验**上正面临挑战，一些曾经的激进策略导致用户口碑受损 <sup>61</sup>。这也提醒我们，利用AI编程虽然前景诱人，但需要理性看待其成熟度。目前来看，**AI不会取代架构师的角色** <sup>62</sup>——它可以提出方案但仍会出错，开发者不敢完全放手不管。这一观点也得到知乎等国内技术社区的认可：Cursor 可以做规划，但还达不到让人完全放心的程度 <sup>63</sup>。因此短期内，“**人机协同**”仍是主旋律：人提供智慧与监督，机器提供速度与知识。在这个过程中，社区实践沉淀下来的各种**最佳实践**和**避坑指南**将是宝贵财富，帮助更多开发者高效且稳健地使用Cursor和Claude Opus等模型开发复杂系统。我们有理由相信，随着工具和方法论的进化，AI辅佐下的软件开发将变得更加**高效、可靠且可控** <sup>64</sup>。

---

1 2 3 4 5 35 36 41 52 53 54 55 56 57 Vibe Coding with Cursor: A Survival Guide for 10x Speed Without 10x Disasters! | by Ahmed Abulkhair | Jun, 2025 | Medium

<https://medium.com/@aabulkhair/vibe-coding-with-cursor-a-survival-guide-for-10x-speed-without-10x-disasters-50e25edaf3d9>

6 33 47 48 49 51 AI Rules and other best practices - Discussions - Cursor - Community Forum

<https://forum.cursor.com/t/ai-rules-and-other-best-practices/132291>

7 8 39 Cursor AI 编程助手使用指南：15个实用避坑技巧 | 智能代码生成器教程

<https://www.axtonliu.ai/blog/cursor-ai-programming-assistant-guide>

9 10 11 28 29 30 38 40 44 45 46 58 64 Cursor Claude Opus 4: Complete Guide to the World's Best Coding AI in 2025 – LaoZhang-AI

<https://blog.laozhang.ai/ai/cursor-claude-opus-4-complete-guide-2025/>

12 13 34 How To Make Cursor Run FASTER As Your Codebase Grows (MUST use Cursor Agent) - Discussions - Cursor - Community Forum

<https://forum.cursor.com/t/how-to-make-cursor-run-faster-as-your-codebase-grows-must-use-cursor-agent/60966>

14 15 16 17 18 19 21 22 23 24 32 How to Reduce 30–40% of AI Token Costs with Context Cache, Snippets, and Structured Prompts | by Raphael Vitor | Nov, 2025 | Medium

<https://medium.com/@sun.rafael/how-to-reduce-30-40-of-ai-token-costs-with-context-cache-snippets-and-structured-prompts-01fe6bbebb37>

20 31 37 59 60 Why is a simple edit eating 100,000+ tokens? Let's talk about this - Discussions - Cursor - Community Forum

<https://forum.cursor.com/t/why-is-a-simple-edit-eating-100-000-tokens-lets-talk-about-this/120025>

25 How Cursor minimises output tokens with 'Apply' model - LinkedIn

[https://www.linkedin.com/posts/ishandutta0098\\_do-you-know-how-cursor-minimises-the-number-activity-7368869114599124993-sMNg](https://www.linkedin.com/posts/ishandutta0098_do-you-know-how-cursor-minimises-the-number-activity-7368869114599124993-sMNg)

26 27 42 43 Cursor's Auto mode is useless. It renders Cursor past the premium requests useless. : r/ ChatGPTCoding

[https://www.reddit.com/r/ChatGPTCoding/comments/1lyi4rw/cursors\\_auto\\_mode\\_is\\_useless\\_it\\_renderer/](https://www.reddit.com/r/ChatGPTCoding/comments/1lyi4rw/cursors_auto_mode_is_useless_it_renderer/)

50 Stop Getting Average Code from Your LLM | Krzysztof Zabłocki

<https://merowing.info/posts/stop-getting-average-code-from-your-lm/>

<sup>61</sup> 用户集体大逃亡！Cursor“自杀式策略”致口碑崩塌：“补贴”换来的王座  
<https://www.infoq.cn/article/06ov3meaqskngp6gm9od>

<sup>62</sup> <sup>63</sup> Cursor IDE的实际使用体验是怎样？ - 知乎专栏  
<https://zhuanlan.zhihu.com/p/719624868>