

如何使用 github copilot 完成 50% 的日常工作

Nov 25, 2023

LLM

chinese

“原文来自于 x 推文，所以保留了原始分段的风格”

0. 一些基础信息

1. github copilot 是 gpt3 针对代码场景优化而来的 Codex 模型，其基础性能不如 gpt4，但在代码场景效果更好

2. copilot 不是银弹，并不是一秒解决 50% 的工作，而是将 50% 的工作时间替换成了 10% 的 prompt/chat 时间
 3. 认清 copilot 的定位，其是一个副驾驶的角色，自己的思维方式要从“如何去做这件事” => “如何激发 copilot 去做这件事”
 4. 尝试 ai-native 的开发方式，从自己编码 + ai copilot 到自己编写 prompt、copilot 编码，然后自己去进行修改
 5. copilot 已经非常强，但还是一个发布并不久的工具，深度使用需要思考如何更贴近它的思维和使用方式，也会遇到很多 bug
 6. 再强调一下，copilot 不是银弹，不是你告诉他需求他就能输出完美方案的 bot，你只是把编码时间换成了 更少 prompt 时间
 7. 不要编程这件事妄自菲薄，不要高看也不要低看，一个学习过所有开源代码的 llm 编程能力是很强的。但依旧需要人类去“激活”和引导，且人类也有其独特的优势
-

1. 基本使用思路


1. 把自己的视野拉高，让 copilot 去做更低维度的事情
 2. copilot 是极度廉价劳动力，是可以让他去帮你试错、可以多开浪费他的思考来节约自己的思考时间
 3. 问 copilot 的问题，自己需要至少有鉴别基础质量的能力，从而能够对他的输出取其精华。在不擅长的领域完全信赖会导致非常严重的问题
 4. 不要懒得写长的 prompt，从 llm 的原理来说，你给的 context 越多，他越容易召回你想要的知识，并给你需要的答案，把他看作一个知识丰富的人类助手，用给人类讲话的耐心去写 prompt。你会发现这事并不会花你太多时间
-

2. 变量命名

这是非常基础但是很多人浪费了很多时间的点。你可以把你想要的这个 变量/类 想要承担的任务和一些想法给到 copilot chat，然后他输出你需要的命名

并且，copilot 的劳动力极度廉价，灵活应用“给我十个，再给我十个，再给我十个”人类想出十个合适答案的能力不如 llm，但很擅长从十个答案中选出合适的一个

在 react 中，我有一个状态量用来表示一个组件是否被激活，应该命名为什么？给我十个候选答案并解释

 GitHub Copilot

Used 1 reference >

1. **isActive**: 这是一个简单直观的命名，表示组件是否“活跃”或“激活”。
2. **isEnabled**: 如果“激活”的含义是组件是否被启用或可用，那么这个命名可能更合适。
3. **isActivated**: 这个命名更强调了“激活”的动作已经发生。
4. **isOpened**: 如果“激活”的含义是组件是否被打开，那么这个命名可能更合适。
5. **isVisible**: 如果“激活”的含义是组件是否可见，那么这个命名可能更合适。
6. **isRunning**: 如果“激活”的含义是组件是否正在运行，那么这个命名可能更合适。
7. **isLive**: 如果“激活”的含义是组件是否处于实时状态，那么这个命名可能更合适。
8. **isEngaged**: 如果“激活”的含义是组件是否被用户或其他组件所使用，那么这个命名可能更合适。
9. **isOn**: 如果“激活”的含义是组件是否处于开启状态，那么这个命名可能更合适。
10. **isInUse**: 如果“激活”的含义是组件是否正在被使用，那么这个命名可能更合适。

以上命名的选择应根据你的具体需求和组件的功能来决定。

3. 代码速读，代码精读，加注释解析，寻找修改项

接收其他人项目、读开源项目等情况，找到需要读的文件，全选，然后打开 copilot chat（它会读取你选中的代码），使用内置的 /explain 命令，这个会内置一些 prompt 让输出质量更好

我常用的几句话是

“从架构设计角度，分析这段代码的设计思路，并讲解这种思路的优劣”

“分析 xxx 函数的详细逻辑，以及在整个文件中起到的作用”

“给 xxx 函数每一行加上注释，以详细解析该函数”

“我现在需要通过修改这个文件以实现 xxx 功能，如何修改？”

“我现在需要用 ts 重写这段 python 代码，详细解析这段 python 代码的设计逻辑，并分析如何在 ts 中实现”

“解析这段代码中可能有哪些风险”

“在这段代码中，run 和 test 方法有什么区别”

copilot 的劳动力极度廉价！

所以在我修一个大系统的 bug 时，我会对多个可能的文件问类似于“我的需求是 xxx，能通过修改这个文件实现么？”，直到找到我需要修改的地方和方案。

llm 读懂代码逻辑的速度极快，可以快速给你一个 80 分的答案，你再判断是否有必要精读。然后再使用 copilot 辅助精读。

4. 代码改写，用 xx 库实现整体逻辑

在要用 b 库改写使用 a 库实现的逻辑时，copilot 做的非常快，因为你 a 库写的逻辑就是最完美的 prompt，在实现完往往只需要通读一边确认答案即可。

这里涉及到对 context 的应用，而因为 codex 的数据库更新并不及时，可能并不了解 b 库。那一个常用的小技巧：

“这是 b 库这个函数的文档，帮我改写这部分用 a 库写的逻辑”

“这是 b 库的官方实例，我想用 b 实现 xx 功能，帮我实现”

这种 few shot 的 prompt 技巧，可以极大程度提高输出质量。不只是在这种场景，很多场景可以应用

5. ai-native 的开发方式

copilot 依旧是个初期产品，但随着发展一定会越来越强大。所以我们应该尝试使用 ai-native 的开发模式，学着更深入的使用 copilot.

我常用的技巧

“我需要一个 ts 类，他的使用方式和调用方式是：<伪代码>，帮我实现一个最基础的版本”

这个其实替代了之前 模板插件 的功能，帮你更快的搭起一个 class 的基础框架，然后自己填充细节。（不会只有我每次都忘记一些 class 的语法还需要每次搜索文档 🐶）

全选所有类代码，然后 “我给这个类添加一个 xxx 函数，帮我参考现有代码，进行实现”

往往质量够用，甚至可以直接使用

“在这个 class 内，我想记录一个逐步产生的 xxx 数据，应该用什么结构比较符合 ts 的编程模式，帮我设计解释你的思路”

“这是我设计的 class/架构/数据结构，目的是 xxx，从优点和缺点各提五点理由，并详细解释原因”

大模型的劳动力极度廉价！

所以先让 copilot 替你思考，很多时候他给的架构非常优秀。即使给的质量比较差，一个错误的答案对你的思考也是有益的。更何况廉价的劳动力，你可以引导他生成非常多，也可以质疑他的架构，并提出你看到的问题，多次沟通直到生成有意义的架构或者理清自己的思路。

ai-native 不是让 ai 设计架构，而是与 ai 多次讨论，让自己的思路更加清晰。

有时候我们知道这个架构有点问题，但不知道怎么改，ai 会给你思路。有时候我们不知道这个架构有什么问题，ai 可以帮你找到问题。

总是，大模型的劳动力极度廉价，用他大量的思考来节约自己的思考

6. 报错解析

这是我高强度使用的一个点，首先代码报错信息是给人类读的，但又不是人类可读的 🐼，且人类很难有 llm 那样无限的上下文和知识

除了非常基础的报错信息，先复制给 copilot chat，使用内置的 /explain 命令，让他分析报错。如果是 vsc 用户，现在已经有一键操作了

再强调一遍，llm 不是银弹，他的答案有偏差，一定注意引导。并且，你问的问题一定是你能够判断基础对错的问题。

常用的几句话

“解释这个报错，并分析可能的原因和修改方式”

“我认为这不是报错的根源，根据你的知识，给出三种可能的出错根源”

尝试一次，你就会发现，与其自己花时间去思考和分析报错，不让先让 llm 给你一个 80 分的答案，在大多数时间他的答案已经可以帮你解决问题了

7. 解释 review message

无论是作为一个 junior sde 还是一个开源新人，外加人类语言表达的局限性。很多 review message 并没有那么明确，与其自己想半天，不如先让 llm 分析下。

复制对应的 diff 和你认为合适的上下文，附上 review message

“这是我的前辈对我的 pr 的 comments，帮我分析意思，并提出合适的解决方案”

llm 的知识库对此做出的解析，以及对 review 黑话/缩写 的分析，往往结果还不错

8. 提高代码质量，设计优化

llm 读过的代码太多了，常用的几句话

“这个 class 的设计有没有考虑到 xxx 的问题”

“解析这个 class 是否有安全风险”

“...，在哪些场景场景在可能会有泄露风险”

“这个 class 如何针对 xxx 做优化”

注意，一般直接问可能并不能拿到高质量的回答，需要人类做方向性的引导，比如提示在什么问题、什么方面等 prompt，可以帮助 llm 沿着具体思路思考

并且要灵活使用 “给我 5 个 xx，并详细解释原因”

9. 灵活使用 cmd+i

最新的 copilot 支持了直接在代码上唤起 chat，你可以选中一段代码，然后 cmd + i，输出你的 prompt，比如 “使用 promise.all 改写” “添加类型注释”

这个很多人没注意到这个功能，结合前面提到的 prompt 技巧很好用。

但目前 diff 功能有些 bug，在部分时候会删改不需要的代码，注意灵活应对。

我一般是把需要代码复制出来，然后 ctrl z 掉他所有更改，然后再粘贴进去。

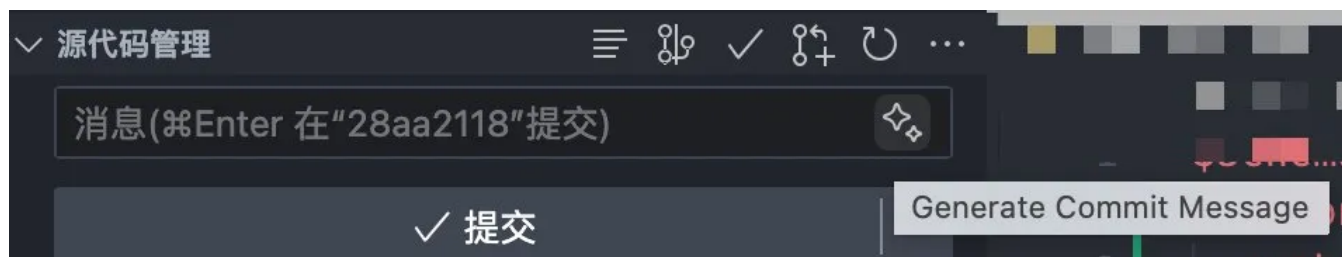
因为这个功能没有上下文，但也有多次对话的能力。适合比较小的需求点，大的最好是用 copilot chat

10. 写 commit message

这个已经在最新的 vsc 中集成，根据你本次的 diff 生成 commit message。

这个思路非常好，但实测其风格不太符合我日常的风格，我相信这个未来会有风格选型，或者以你之前的 commit message 作为上下文进行生成。

目前我推荐在这个 generate 的基础上自己修改，或通过 chat 的方式生成



11. 基础脚手架、基础 poc

这也是 ai-native 的一部分，也是我最近用起来比较顺手的

“我要写一个 nodejs 库，帮我写一个基础的 rollup 配置、tsconfig 和 package.json 的配置” “帮我用 react 写一个基础的 xxx 组件”

前者是，很多时候没有好用的现成配置，用 llm 就很方便。后者是有一个迅速能看到的基础代码，会帮助你思考和工作。

12. 中间插入一些唠叨

vsc 设置成你最熟悉的自然语言！

虽然未来（或者已经）会有给 copilot chat 单独设置自然语言的功能，但我建议直接把 vsc 设置成你最熟悉的自然语言。然后方便的速读

llm 不是人类，不用字斟句酌，有合适的关键词即可。如果不知道怎么表达，就用最暴力的表达方式即可

写 prompt 的时间和 写 code 的时间，这两个随着深入使用，你会逐步找到自己舒服的状态。当你做一个事情的时候，你会知道是使用 llm 还是直接写 code 会更快/质量更高。

一般理想的是 人类冷启动/llm 冷启动 => 人类编写细节/llm 编写细节 => 人类 polish / llm polish

熟练后，在每个阶段都可以非常快速的判断出，这个时候是人类做还是 llm 做，还是一起做

13. llm as doc/search

再强调，一定要问 llm 自己能够判断基础对错的问题！

这里的工具就不限于的 copilot chat 了，我一般也会混着 new bing （有联网能力）使用。

比如

“ts 中，interface 和 type 的区别”

“ts decorators 是否 stable？现在进入 stage 几了？”（new bing）

几个非常好用的 magic word：“举例详细说明”、“详细对比这两个的优缺点”、“举出实际场景对比这两个区别”、“使用 xxx 函数，写一个简单 demo，介绍其优势”

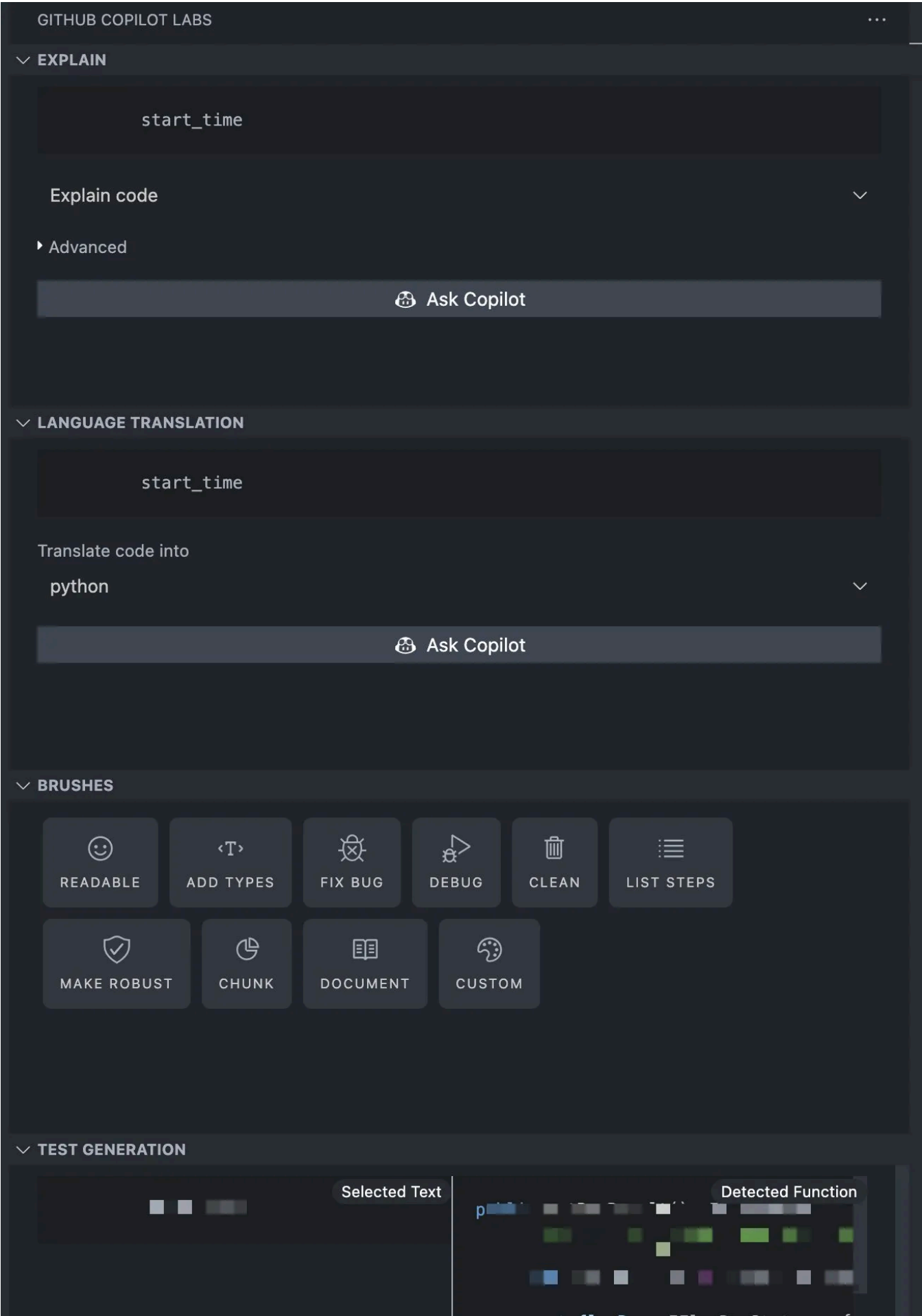
作完调研后，“用 xx 实现我的 xx 需求”，从调研到实现，几分钟 几轮对话，就结束了

一定要有基础的技术视野和知识去判断其输出的质量。我遇到过好几次，llm 硬着脖子非要用 moment 去处理 ts 中的时间，直接被我喷回去，然后乖乖用 dayjs 了 🐼

14. 碎碎念

我因为开源项目，一直可以免费用 copilot，算是非常老的用户了。之前比较流行的写注释然后让 copilot 补全代码的模式一直不太会用，会让我觉得很怪。但 copilot chat 确实是 game changer，几天内就替代了我 50% 的工作。我相信下一次飞跃就是 copilot 带联网功能的时候，到时候会进一步挤压人类的编码空间，亦或是说，人类可以更从容的做更有创造性的工作。

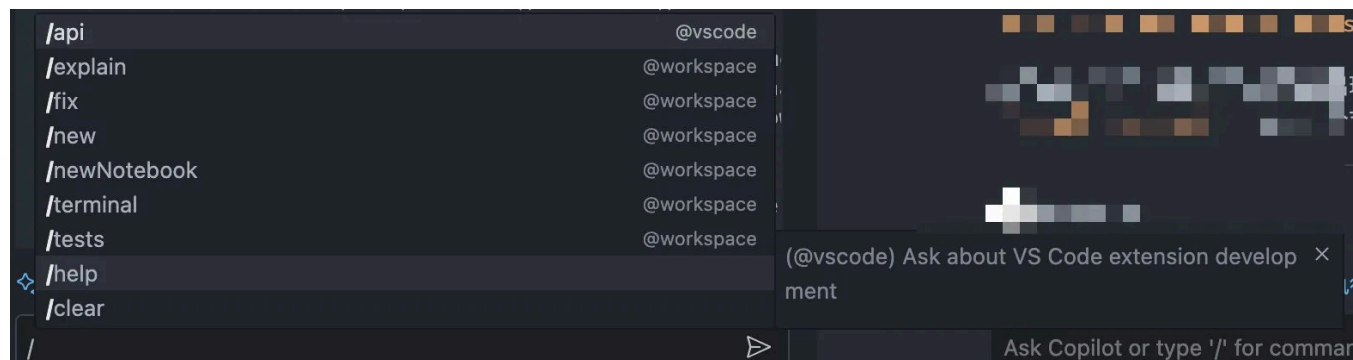
顺嘴提一句 copilot labs，可能很多人都不知道有这个东西。在 chat 出来之前玩玩还可以，在 chat 面前一文不值的 🐼



15. vsc plugin 开发

这也是我看很多人没提到的点，我日常工作有 vsc plugin 的开发工作，copilot chat 已经内置了 plugin 相关的文档，你可以直接用自然语言提问你的问题和需要开发的功能在 vsc 中如何实现。

也可以通过 /help 命令，看看 chat 内置的一些功能，这些功能往往伴随着内置的 prompt 和数据库，对特定任务有增强



/help



GitHub Copilot

You can ask me general programming questions, or chat with the following agents which have specialized expertise and can perform actions:

- **@workspace** - Ask a question about the files in your current workspace
 - **/explain** - Explain how the selected code works
 - **/tests** - Generate unit tests for the selected code
 - **/fix** - Propose a fix for the problems in the selected code
 - **/new** - Scaffold code for a new workspace
 - **/newNotebook** - Create a new Jupyter Notebook
 - **/terminal** - Ask how to do something in the terminal
- **@vscode** - Ask questions about VS Code
 - **/api** - Ask about VS Code extension development

To have a great conversation, ask me questions as if I was a real programmer:

- **Show me the code** you want to talk about by having the files open and selecting the most important lines.
- **Make refinements** by asking me follow-up questions, adding clarifications, providing errors, etc.
- **Review my suggested code** and tell me about issues or improvements, so I can iterate on it.

You can also ask me questions about your editor selection by **starting an inline chat session**.

Learn more about **GitHub Copilot** in **Visual Studio Code**.

