Google亲自下场力挺：“Go才是AI辅助软件工程理想语言”，却引全网热议

CSDN 2026年8月13日 19:00 江苏

编译 | 苏宓

出品 | CSDN（ID：CSDNnews）

编程语言之争，在技术圈已经吵了三十多年。如今，随着 AI 编程工具越来越强，这场争论也逐渐从“哪种编程语言最强”，演变成了一个新问题：什么样的编程语言，才最适合 AI 写代码？

这一次，Google 亲自下场给出了自己的答案：**Go。**

8 月 11 日，Google 产品经理 Cameron Balahan 和 Google Cloud 首席布道师 Richard Seroter 联合发布了一篇长文，标题相当直接——《**Why Go is an Ideal Language for AI-Assisted Software Engineering**》（为什么 Go 是 AI 辅助软件工程的理想语言）。

其二人的核心观点也很明确：AI 正在改变软件开发的分工。过去，开发者最在意的是“写代码有多快”。现在，AI 几秒钟就能生成几百行代码，真正拖慢开发流程的反而变成了阅读、审查、验证和维护这些代码。因此，一门适合 AI 编程的语言，不应该只追求“让代码更容易写”，还应该让生成出来的代码更容易看懂、更容易验证，也更不容易在长期维护中失控。

而在 Google 看来，Go 恰好具备这些特点。

这也引出一个更有意思的问题——**当 AI 开始替程序员写代码后，我们究竟应该重新评价一门编程语言的什么？**

01

Google 为什么认为 Go 是 AI 辅助软件工程的理想语言？

在这篇文章中，Google 给出了五个原因解释了自己的判断。

一、Go 是为软件工程而生，而非单纯面向编程

二十多年前，Rob Pike、Robert Griesemer 与 Ken Thompson 在 Google 创造 Go 语言，初衷正是围绕团队协作开发进行考量。当其他编程语言不断快速增加新特性，试图扩展表达程序逻辑的方式时，Go 关注的是一个更大的目标：让语言设计服务于软件工程。

软件工程和编程并不是一回事。编程只是编写代码、运行代码解决单一问题；软件工程，则是多人协作设计、搭建一套能够长期迭代演进的稳定系统。编程只是软件工程的其中一环。

让语言设计服务于软件工程，意味着需要的不只是一门编程语言，而是一套贯穿整个软件开发生命周期的完整平台和工具链。这要求语言足够简单且有明确的规范，让整个团队可以用统一的方式组织、格式化和测试代码；也要求有强大的兼容性保障，**确保今天写下的代码不仅十年后依然能够运行，而且十年后依然是一份好的代码。**

它还需要一个强大的生态系统，以及一套能够随着团队规模扩大而扩展的全球依赖管理体系。同时，安全性也必须被纳入其中，在整个工具链中提供合理、可靠的安全机制和工具。

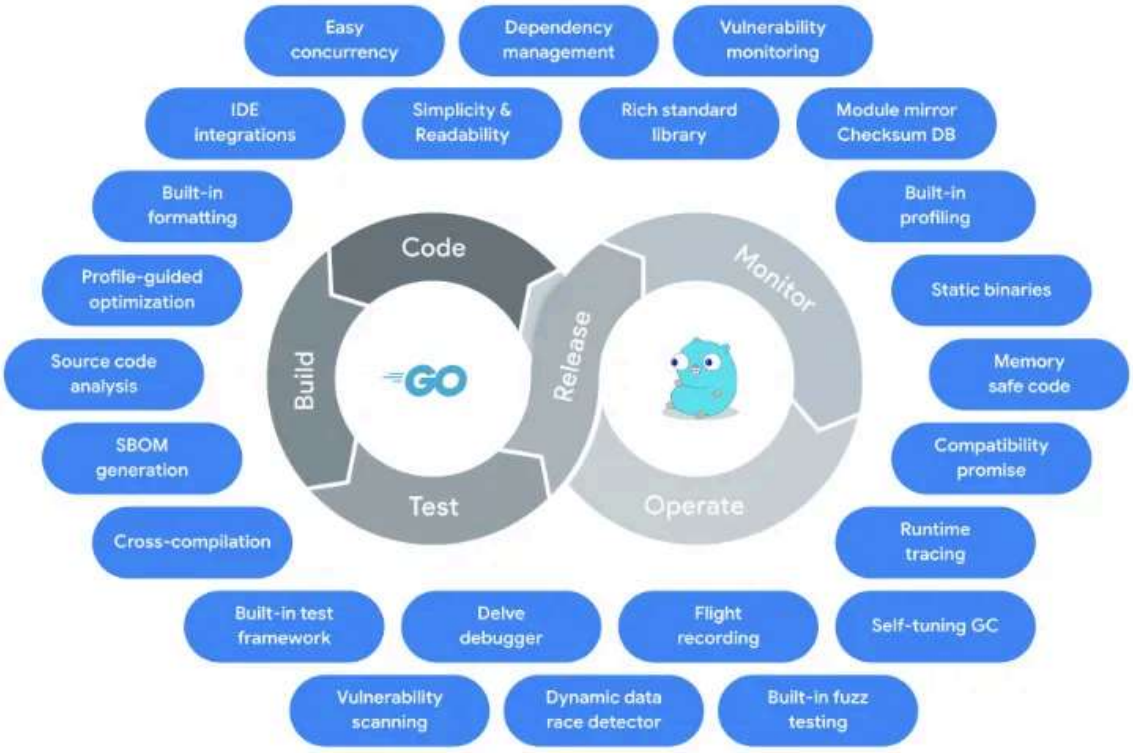
以上要素共同构成长期大规模团队协作的基础，支撑项目在原始开发者离职多年后依旧可以持续维护。如今 AI 成为开发团队的一员，这套底层基础变得前所未有的重要。

二、Go 不只是一门语言，更是一个平台

Go 最与众不同的一点，就是它不只是一门编程语言，更是一整套平台。

从诞生之初，Go 就自带了一套完整、可靠的端到端工具链，覆盖软件开发生命周期中的各个环节。开箱即用的 Go 平台提供了内置格式化工具、测试框架、依赖管理以及先进的安全工具，这些能力都可以直接通过标准工具链使用。

再加上功能全面的标准库，开发者无需依赖复杂的外部框架，就能完成大量工作。这些能力共同构成了一套其他语言很难比拟的统一基础。



这套工具链最初为人类开发者打造，但 AI 和人类开发者的诉求高度相似。如果让 AI 智能体持续迭代重构代码却缺少校验手段，输出质量会快速下滑，和人类徒手重构遇到的困境如出一辙：首轮生成正确率或许能达到 95%，反复迭代后错误持续累积，占用大量上下文窗口，不仅准确率下降，还会拉高 Token 开销。

依托 Go 一体化平台工具链，大模型能够更快、更低成本、更可靠地处理 Go 代码，产出质量更高、更安全、逻辑更严谨的程序。

这套集成工具链还带来一个容易被忽视的优势：整个生态高度统一。绝大多数 Go 开发者使用同一套核心工具，社区可以同步跟进语言重大升级，运行时、IDE、开源包生态无缝适配。再加上标准库进一步抹平项目之间的差异，催生大量易懂、可预期的通用代码范式，人类和 AI 都能快速读懂。这种统一特性，既方便大型团队维护巨型代码库，也为大模型提供规范、干净的训练数据。

三、Go 更容易阅读，降低 AI 代码审查成本

Go 另一大标志性设计，是可读性优先于编写便捷性。

Rob 等人很早就意识到：**开发者阅读已有代码耗费的时间，远远多于编写新代码。**在纯人工开发时代，这套理念催生了崇尚简洁、摒弃“语法黑魔法”的社区文化。Go 开发者常常提到，你很难分辨一段代码出自团队里哪位成员 —— 所有代码风格高度趋同。

进入 AI 开发时代，“可读性优先”的设计理念进一步放大优势。过去很多开发者偏爱简洁语法、隐式类型、各类技巧快速搭建原型。但 AI 智能体生成代码，以及后续人工复核的流程，恰恰需要相反的特性：可预测性、显式声明、严谨结构。

软件研发流程的瓶颈，从代码生成，彻底转向代码校验。**如果一门语言存在十几种写法实现相同逻辑，AI 生成的代码风格会杂乱碎片化。**对于负责 Review 的人来说，验证这样的代码会变成一场令人疲惫的“猜作者到底想干什么”。

Go 通过近乎苛刻的一致性解决了这个问题。借助内置的 gofmt 工具，Go 强制使用统一的代码格式。同时，Go 的语言设计也有意限制复杂抽象的使用。无论代码是由资深工程师、初级贡献者还是 LLM 编写，最终看起来都大致一样。

当语法完全可预测时，人类开发者就能更快发现 AI 幻觉出来的 API 调用、逻辑错误或安全漏洞。而且，这种标准化还延伸到了整个 Go 开源生态。模型因此能够在更加标准化的数据上进行训练，从而用更少的尝试生成正确、符合 Go 惯用风格的代码。

归根结底，适合人类阅读的语言，天然更容易被 AI 理解。随着 AI 持续拉高代码产出规模，Go 对可读性的坚持，可以保证系统持续扩张的同时，我们依然能够读懂、校验、安全维护代码。

四、Go 具备可靠性，自动拦截 AI 常见缺陷

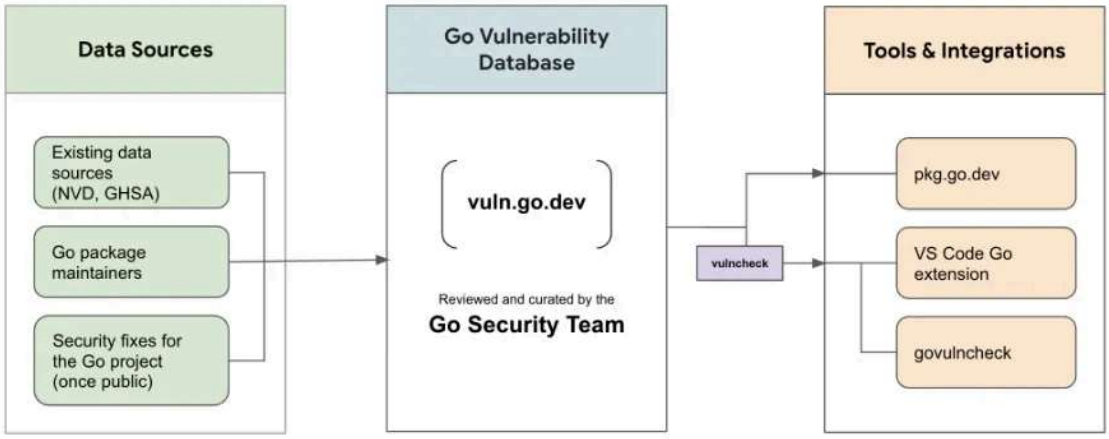
可读性与开发效率只是一部分。即便代码再好读、开发再高效，如果程序脆弱、存在安全隐患、高负载下行为不可预测，依旧无法投入生产环境。

静态类型系统是 Go 第一道防线，充当 AI 生成代码的自动安全屏障。大模型经常难以跨文件把握结构边界与类型一致性，容易虚构属性、埋下难以察觉的隐性 Bug。在 Python 这类动态语言中，这类错误可以绕过基础语法检查，等到生产环境特定负载下才触发崩溃。

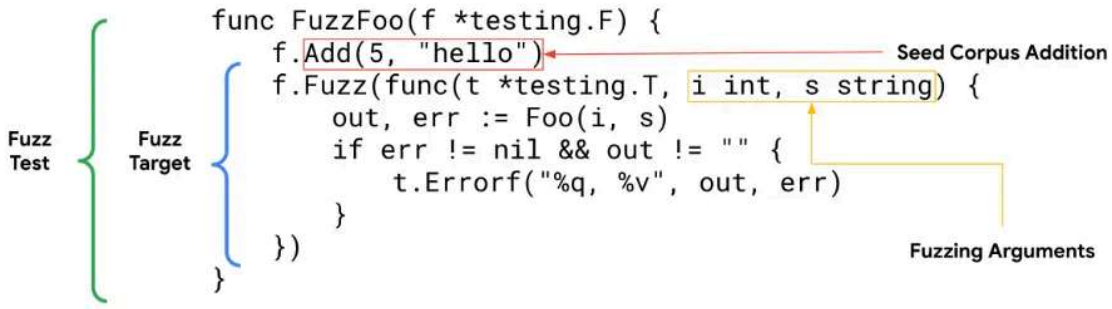
而**在 Go 中，编译器会直接拦截这类问题。如果 AI 调用不存在的方法、传入错误类型参数、变量未初始化，代码直接编译失败。**

同时 **Go 编译速度远超 Java、C#、Rust 等主流编译型生产语言**，AI 智能体可以快速迭代修正类型、语法错误，形成高效自校正循环，代码在交给人类审核前就完成基础纠错。

除编译器之外，Go“开箱即用”的理念，解决了 AI 代码一大典型安全隐患：软件供应链风险。大模型实现功能时，常会参考训练数据引入老旧、无人维护甚至带有恶意的第三方依赖。Go 强大的标准库会引导 AI 优先选用官方维护、经过优化的内置包，减少引入外部依赖，大幅缩小供应链攻击面，保持代码精简易维护。



当确实需要引入外部依赖时，Go 平台保障依赖完整性。所有模块校验和、缓存副本记录在校验数据库与模块镜像，杜绝中间人攻击，防止依赖包消失或者被暗中篡改。配套漏洞数据库与 govulncheck 工具，可以精准扫描依赖漏洞，只标记代码实际调用到的风险函数，为人类和 AI 提供精准、低干扰的修复指引。



最后，内置测试框架与原生模糊测试工具，提供标准化持续验证环境。无需拼凑各类外部测试组件，AI 和开发者可以直接使用原生工具编写测试用例。借助模糊测试持续挖掘边界条件缺陷，AI 能够不断加固自身生成的逻辑，最终形成一套高可靠研发流程，代码上线前完成充分健壮性验证。

五、Go 易于维护

可读的代码能够帮助你把软件送进生产环境，可靠的代码能够让它今天继续稳定运行，但**真正衡量一个软件系统的标准，是它在上线第二天以及更长时间之后，是否依然容易维护。**

代码库是一个不断变化的系统。它会自然老化，积累技术债务，同时还必须持续适应不断变化的需求。

在人类开发者是软件唯一作者的时代，这种维护负担是运营成本中一个相对可预测的部分。但如今，自治 AI Agent 可以随意生成数百个 Pull Request，甚至一次性重构整个服务，代码库演进的速度以及架构逐渐偏离原始设计的风险都在大幅增加。

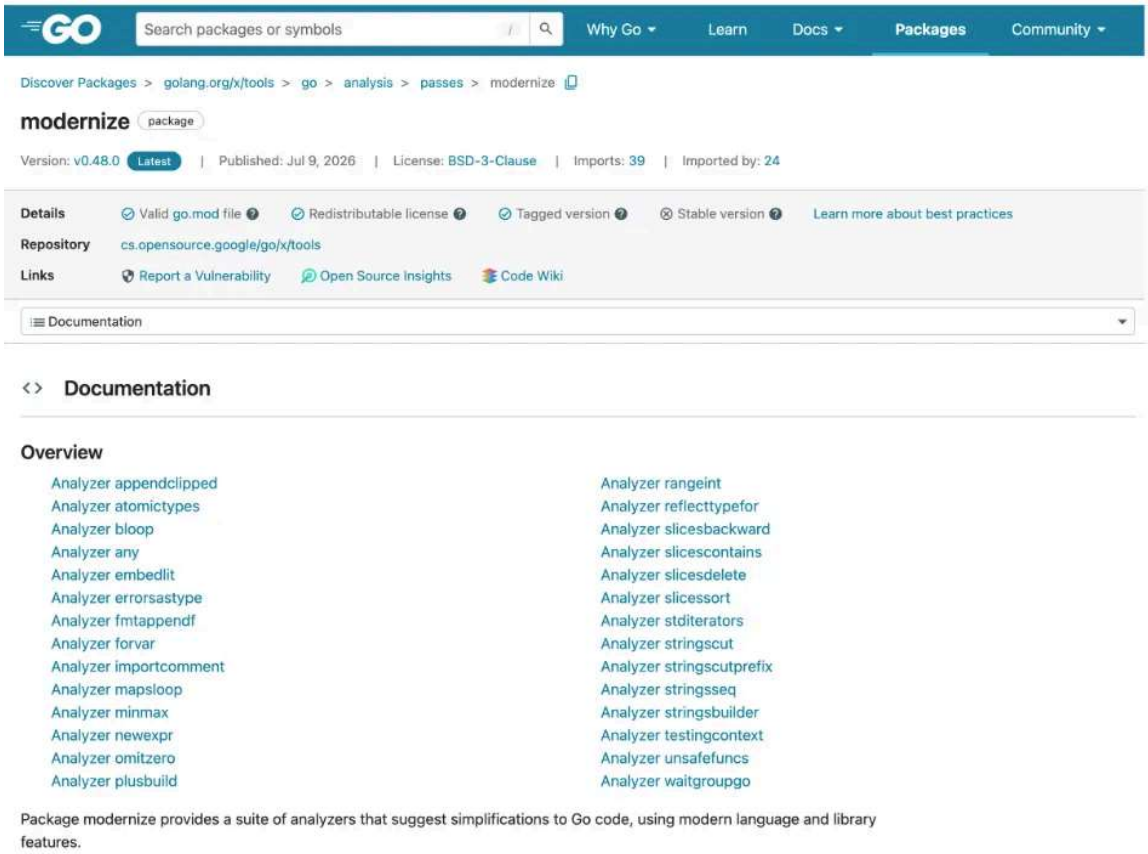
Go 应对这种变化的核心能力之一，就是它著名的兼容性承诺。对于 Go 来说，兼容性并不只是为了方便开发者，而是一项关键的安全和运维要求。

得益于这项兼容性承诺，15 年前为 Go 1.0 编写的代码，不需要任何修改，就可以使用最新的 Go 工具链进行编译和运行。而且，由于 Go 承诺永远不会破坏向后兼容性——不会有 Go 2.0！——Go 代码也不会因为语言本身的升级而失效。

相反，随着 Go 编译器和运行时不断改进，你的代码也会随之变得更好，而且不需要修改任何代码：升级、重新编译，就能直接获得这些改进。

这种长期的稳定性，再结合 Go 出色的运行环境可移植性，会变得更加有价值。Go 可以直接编译成一个独立的静态二进制文件，不依赖任何系统环境。

随着自治 AI Agent 越来越多地承担系统管理员的工作——启动微服务、执行脚本，以及通过命令行与各种环境交互——这种自包含的设计变得前所未有地重要。同时，由于 Go 编译器支持跨操作系统和系统架构进行交叉编译，这些 AI Agent 可以根据需要轻松构建面向各种目标平台的二进制文件，而无需搭建复杂的构建系统。



为了应对架构逐渐偏离的问题，Go 提供了一系列内置、确定性的工具，用于大规模重构和现代化代码库以及整个 Go 生态。其中包括 Go 官方语言服务器 gopls，以及重新构建的 go fix。后者如今加入了 Modernizers 的概念。

Modernizers 可以确定性地将旧的代码写法更新为最新的惯用写法和语言特性，从而保持代码的一致性。在大规模场景下，这些工具推动的不只是你自己的代码，还有整个 Go 生态向前演进，让库、开源项目以及其他第三方代码库都能够保持统一。

而且，由于这些工具经过标准化，并直接集成在 Go 平台中，AI Agent 可以利用它们安全地重构代码包、管理依赖以及清理技术债务，同时避免破坏现有代码库。

最后，Go 还通过内置的可观测性和性能调优工具，让这种可维护性能够一直延伸到生产环境。Go 运行时开箱即用地提供性能分析和执行追踪能力，让开发者能够深入了解应用在高负载下的实际运行情况。Go 编译器还原生支持基于性能剖析的优化，可以利用真实生产环境中的性能数据进行编译，生成针对实际使用情况高度优化的二进制文件。

当这些能力与由 AI 编排的部署流水线结合起来，就可以形成一个高度自动化的闭环优化过程：**生产环境的数据可以自动反馈给编译器，再重新构建并优化整个系统。**

// 02

HN 网友的热议两极分化

当然，此文一发布，也有不少网友觉得这里多少有一点"王婆卖瓜"的意味。毕竟 Go 本身就是 Google 一手推动起来的语言，如今由 Google 的产品经理和 Google Cloud 高管出来告诉大家"Go 是 AI 时代的理想语言"，难免会让人产生疑问：这是技术判断，还是给自家语言做推广？

随着争议愈演愈烈，这篇文章很快冲上 Hacker News 热榜，评论区直接变成了一场新一轮的编程语言大战。

有 Netflix Go 团队负责人分享一线实践，直接印证 Google 博客提出的观点：

在 Netflix，我负责 Go 语言技术团队。我们发现，越来越多的用户反馈，他们的 AI Agent 写出来的 Go 代码，比用其他编程语言写出的代码质量更高；与此同时，也有越来越多的项目开始倾向于选择 Go，而不是其他语言。

我还想补充两点：

- Go 在如何写出高质量 Go 代码方面，有非常丰富的资料，包括 Effective Go 和 Google Go Style Guide 这样的宝藏资源。补充一下：不好意思，我刚才忘了说，我们也会把这些资料提供给 AI Agent，它们会利用这些资料写出更好的 Go 代码。
- 对一个语言团队来说，Go 简直是梦想。go fix 工具、AST/SSA 包、go.mod 文件易于读取和修改（包括 go mod edit 等工具），以及其他各种偏"平台级"的特性，都让大规模修改 Go 代码变得比其他语言容易得多。

但另一边，质疑声同样此起彼伏。有网友分享道：

我们在评测中发现了一个相当稳定的现象：Go 和 Python 一样，都是模型表现最差的语言之一，至于原因，目前还不清楚。我们的代码评测通常会关注模型在代码中体现出的预判能力和规划能力，而这些代码会在交互式环境或游戏中实际运行。

从 2026 年 2 月我们开始使用不同编程语言评测模型以来，这一趋势就一直存在。如果一定要说有什么变化，那就是在前沿模型中，这种差距反而越来越明显。甚至 Google 自家的模型，在构思有创意的解决方案时，也更偏爱 Kotlin、C# 和 Rust（但能不能成功编译又是另一回事）。相关数据可以参考 GertLabs Rankings（https://gertlabs.com/rankings）。

话虽如此，模型确实很喜欢推荐 Go，而 Go 本身也有很多优势，尤其是在开发面向公众的网站时。因此，我们大多数面向公众的 API Handler 都是用 Go 写的，同时也会把一些最重要的二进制程序交给 Rust 来实现。毕竟，模型在某些语言上表现得稍微更好，并不足以成为我们选择这些语言的理由——不使用它们的理由实在太多了。

也有开发者做了一个假设：

假设你现在需要根据一组明确的需求写一个程序，同时手里还有一根“魔法棒”：它可以让你瞬间、免费地用任何编程语言生成一个高质量的程序实现。

我的“大胆观点”是：**如果真有这样的魔法棒，你可能不会选择 Go，而很可能会选择 Rust。**

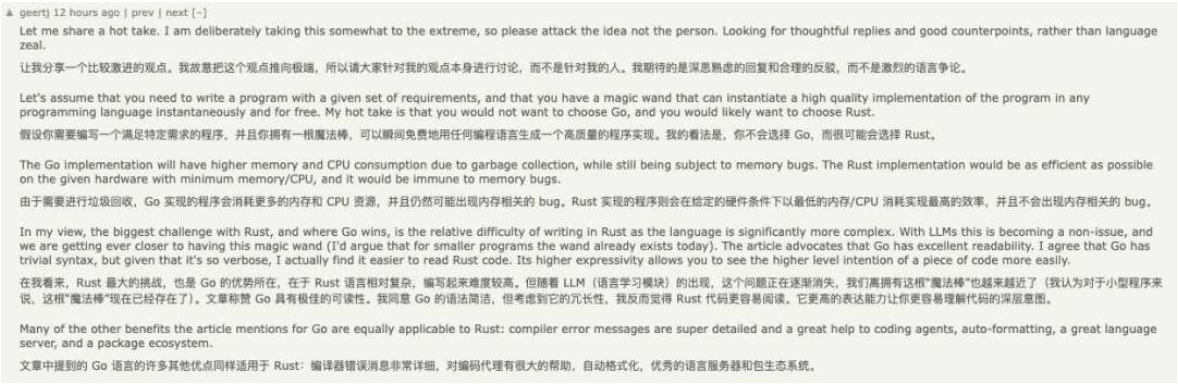
Go 版本由于需要垃圾回收，因此会消耗更多内存和 CPU，同时依然无法完全避免内存相关 Bug。而 Rust 则可以在给定硬件条件下，以尽可能低的内存和 CPU 消耗实现最高效的运行，并且能够从根本上避免内存安全问题。

在我看来，Rust 最大的问题，也是 Go 占优势的地方，在于用 Rust 写代码相对更难，因为这门语言本身要复杂得多。

但随着 LLM 的出现，这一点正在逐渐变得不再重要，我们也越来越接近拥有那根“魔法棒”——甚至可以说，对于规模较小的程序来说，这根魔法棒今天已经存在了。

Google 的文章强调 Go 具有出色的可读性，这一点我同意。Go 的语法确实非常简单，但由于 Go 的代码往往比较冗长，我反而觉得 Rust 代码更容易阅读。Rust 更强的表达能力，可以让你更容易从一段代码中看出它更高层次的意图。

文章提到的 Go 的其他许多优势，同样也适用于 Rust：编译器错误信息非常详细，对 Coding Agent 有很大帮助；自动格式化；优秀的语言服务器；以及完善的包生态。


▲ geertj 12 hours ago | prev | next [–]

Let me share a hot take. I am deliberately taking this somewhat to the extreme, so please attack the idea not the person. Looking for thoughtful replies and good counterpoints, rather than language zeal.

让我分享一个比较激进的观点。我故意把这个观点推向极端，所以请大家针对我的观点本身进行讨论，而不是针对我的人。我期待的是深思熟虑的回复和合理的反驳，而不是激烈的语言争论。

Let's assume that you need to write a program with a given set of requirements, and that you have a magic wand that can instantiate a high quality implementation of the program in any programming language instantaneously and for free. My hot take is that you would not want to choose Go, and you would likely want to choose Rust.

假设你需要编写一个满足特定需求的程序，并且你拥有一根魔法棒，可以瞬间免费地用任何编程语言生成一个高质量的程序实现。我的看法是，你不会选择 Go，而很可能会选择 Rust。

The Go implementation will have higher memory and CPU consumption due to garbage collection, while still being subject to memory bugs. The Rust implementation would be as efficient as possible on the given hardware with minimum memory/CPU, and it would be immune to memory bugs.

由于需要进行垃圾回收，Go 实现的程序会消耗更多的内存和 CPU 资源，并且仍然可能出现内存相关的 bug。Rust 实现的程序则会在给定的硬件条件下以最低的内存/CPU 消耗实现最高的效率，并且不会出现内存相关的 bug。

In my view, the biggest challenge with Rust, and where Go wins, is the relative difficulty of writing in Rust as the language is significantly more complex. With LLMs this is becoming a non-issue, and we are getting ever closer to having this magic wand (I'd argue that for smaller programs the wand already exists today). The article advocates that Go has excellent readability. I agree that Go has trivial syntax, but given that it's so verbose, I actually find it easier to read Rust code. Its higher expressivity allows you to see the higher level intention of a piece of code more easily.

在我看来，Rust 最大的挑战，也是 Go 的优势所在，在于 Rust 语言相对复杂，编写起来难度较高。但随着 LLM（语言学习模块）的出现，这个问题正在逐渐消失，我们离拥有这根“魔法棒”也越来越近了（我认为对于小型程序来说，这根“魔法棒”现在已经存在了）。文章称赞 Go 具有极佳的可读性。我同意 Go 的语法简洁，但考虑到它的冗长性，我反而觉得 Rust 代码更容易阅读。它更高的表达能力让你更容易理解代码的深层意图。

Many of the other benefits the article mentions for Go are equally applicable to Rust: compiler error messages are super detailed and a great help to coding agents, auto-formatting, a great language server, and a package ecosystem.

文章中提到的 Go 语言的许多其他优点同样适用于 Rust：编译器错误消息非常详细，对编码代理有很大的帮助，自动格式化，优秀的语言服务器和包生态系统。

此外，更有评论直截了当地指出：现在几乎每隔一段时间，就会有人宣布某门语言是“最适合 LLM 的语言”——Rust、Zig、Lisp、Erlang、TypeScript、Python……

事实上，在此之前，开发者评判一门编程语言，更多看性能、开发效率、生态和就业机会。生成式 AI 普及之后，一些新的标准正在进入这场比较：AI 生成代码的准确性、代码的可读性和歧义程度、依赖复杂度，以及对自动化测试、重构和长期维护的适配能力。

Google 这篇博客的意义，也许不在于证明“Go 就是 AI 时代最好的编程语言”，而是把一个过去很少被单独讨论的问题摆到了台面上：当写代码的人逐渐变成了 AI，我们是不是也该换一套标准来重新审视编程语言？

至于 Go 能不能成为这场变化中的赢家，现在下结论还为时尚早。毕竟，AI 写代码的能力还在快速变化，而不同语言在真实开发场景中的表现，也远比一篇博客复杂得多。

那么问题来了：**如果代码越来越多由 AI 来写，你会选择什么语言？**

来源： https://developers.googleblog.com/why-go-is-an-ideal-language-for-ai-assisted-software-engineering/

https://news.ycombinator.com/item?id=49261133

“写代码从来都不是难点”？25年开发者怒写3000字反驳：这句话是对所有程序员的一种侮辱！

每周 120 万个 App 诞生，每位开发者都是智能体管理者 —— Google 在上海画出了一张全栈 AI 出海路线图

openJiuwen协同昇腾打造智能体「算力亲和」技术，首 token 时延砍半，推理存储占用下降25%

11.20-21 日，2026 奇点智能大会·北京站。

OpenAI 资深研究科学家 Łukasz Kaiser 确认进行主题分享。

他是 Transformer 八子中唯一还在科研前线的人，GPT-4/5、o1、o3、ChatGPT 的核心开发者。

2021 年，他在这个大会上分享了三个方向：多模态、更大更好的 Transformer、模型即服务。五年后，多模态爆发，ChatGPT 席卷全球，前沿观点全部成了现实。

去年，他讲的是推理模型的进化：从"记忆"到"策略"再到"研究器"——模型开始学会自己思考。

欢迎大家扫码领取 Łukasz Kaiser 历年在奇点智能大会上的分享 PPT 和视频

if like:

click("分享🔗", "点赞👍", "在看🌸")