一等公民

为人类与智能体共同构建

ZACH "LEIF" ERIKSEN

版权声明

可在线免费阅读。ePub 版本随心付；如果本书对您有所帮助，欢迎支持。

github.com/0xLeif · leif.algo

本书是"智能体技术栈"系列四本书之一。创作过程详见末尾的后记。

献辞

献给所有在公开环境中构建、并勇于发布的人。

书目

这些书各自独立成篇，但它们是作为一个整体写就的。代码变得廉价，信任变得稀缺。合而观之，它们共同构成一个论点：现在该构建什么，以及如何信任它。

- 《**智能体开发者野战手册**》：为真正交付代码的智能体构建工具、规范与信任
- 《**一等公民**》：为人类与智能体共同构建 *(本书)*
- 《**构建智能体**》：尝试赋予软件自己双手的笔记
- 《**开源工具**》：构建人们真正使用的工具

均可在线免费阅读。每本 ePub 随心付。

目录

引言

这是这四本书中最简短、也最固执的一本，而其他三本其实都是在为它所主张的观点服务。

大多数工具是为人类构建的，然后智能体被硬拼在旁边：一个次级 API、一个标志位、一种只能实现真实产品功能之半的特殊模式。我认为这是本末倒置的。这里的主张很简单。一个工具应该对两者都是一等公民。人类应该能够在没有智能体的情况下驱动它，智能体应该能够在没有人类的情况下驱动它，通过同一界面，不存在二等入口。

我是从实践中得出这个结论的，而非从理论。我构建小型 Swift 库和开发者工具，那些经得起时间考验的，都是核心足够干净、任何东西都能调用它的那些。当智能体出现时，那些工具早已准备就绪。那些没有准备好的，是我一直在道歉的。

如果说《野战手册》是方法，《构建智能体》和《开源工具》是证据，那么这本书就是它们共同服务的论点。你不需要另外三本来使用这一本。它的目的是改变你看待下一个你所构建或选择的工具的方式：问问它是否将人类和智能体视为同等公民，然后看看从这个答案中会引申出多少东西。

它简短是有意为之的。一口气读完它。

双重公民

我的许多工具都源于同一个想法：人类和智能体将使用相同的工具。

所以这里是整本书所依据的定义，直接明了地说出来：一等公民意味着人类可以独自驱动该工具，智能体也可以独自驱动该工具，双向使用相同的命令。不是两个产品。一个工具，两种用户，都不是另一种需要被转译成的特殊情形。

当我说一等公民时，我的意思和程序员说的一样：一个工具实际上是为其构建的、真正受支持的参与者，而不是它容忍的客人。不是飞机升舱。一等用户是工具围绕其设计的，有真正的使用入口。而不是必须先被转译成别人的形状才能进入的那种。

可以有专为智能体构建的工具。可以有专为人类构建的工具。有趣的情形（几乎没有人在为之构建的那种）是两者兼顾的工具，而"两者兼顾"就是我所说的*两者均为一等公民*，也是本书其余部分的含义。它比智能体优先更严格：智能体优先只需要服务智能体，而两者均为一等公民则必须通过同一界面同时服务人类和智能体，且任何一方都不能获得更差的版本。目前我们大多拥有的是人类优先的项目，我们正在将智能体*引入*其中。智能体迟到了，来到了一个为别人构建的工具，我们希望它能摸索出来。

我们真正需要的是人类与智能体双优先的项目。从一开始就这样构建。

"人类优先再硬拼智能体"是默认做法，因为它阻力最小。你已经有了这个工具，它已经为你服务了，当智能体来临时，最简单的做法就是在你已有的东西上包一层薄薄的胶水。这可行，勉强可行。你拿了一个围绕人类的眼睛和双手设计的工具，然后要求一个两者都没有的进程假装它两者都有。这给智能体带来的代价（在提示上的停顿、它无法读取的输出、每次运行都要重新学习）在后面有专门的章节。这里只需说：智能体在每一行假设人类存在的代码上都在填补空白。

所以反转假设。构建工具使其以任何方式都能一等公民地运行。这就是顶部定义作为构建指令的重述：你没有真实产品和旁边贴上的"API 模式"，你也没有一个 CLI 和一个单独的、每次你改动任何东西就脱节的智能体垫片。两者都是真实用户。两者都应该在那里。

这是对我来说最重要的部分。当智能体是一等用户时，工具可以真正地*帮助*它，而不是让它猜测所有命令以及事物的运作方式。人类可以在一个令人困惑的工具中摸索。他们会读 README，尝试一件事，读错误信息，再尝试另一件事，问同事。智能体的摸索只是昂贵的猜测。一个为智能体构建的工具会告诉它什么是可能的，提供它可以直接使用的输出，并以一种指示下一步该怎么做的方式失败。

这也是人类想要的同一个工具。可发现的命令、可信赖的输出、告诉你出了什么问题的错误。智能体只是无法像人类那样填补空白，所以为智能体构建迫使你消除这些空白。为两者设计使软件更好。

人们听到"也要为智能体构建"时，会以为这意味着两个产品，或者一种妥协，让每一方都得到它想要的更差版本。它是一个好的核心，加上一个将其暴露给两者的步骤。这不是双倍的工作，下一章会解释原因。

我一直回到这个话题的原因是，随着时间推移，智能体只会做得更多，而不是更少。今天它们在人类工具中摸索。明天它们正在完成真正份额的工作，而这个份额在增加。如果我们现在构建的工具假设人类总是在那里处理提示、读屏幕、点按钮，我们就是在一个每个月都变得越来越假的假设之上构建未来。

这个论点还有一个更大的版本。我们有十年的完全由人类编写的代码，现在 AI 在所有这些代码之上编写代码，在某个时刻这个比例会翻转。工具决定了这是否会顺利进行。这本身就是一整章：桥梁。

目前，主张只是这个：人类和智能体将使用相同的工具。所以从一开始就为两者构建，作为平等的一等公民。

智能体的一等公民待遇

那么，一个工具实际上需要什么才能让智能体成为一等用户，而不是让智能体在人类工具中摸索？

四件事。没有一件是稀奇的。

结构化的、机器可读的输出。一个真正的序列化格式（JSON 是我首选的），这样智能体得到的是数据，而不是屏幕。大多数工具返回漂亮的文本：对齐的列、颜色、底部的摘要行。那是为了人类的眼睛。智能体必须抓取它，而抓取在你改变间距的那天就会失效。给它实际数据。它读取一个字段，而不是解析一个段落。

可发现的、一致的命令。整个工具中一致的动词，以及智能体可以读取以了解可能性的自描述 `--help`。如果帮助文本是真实的，智能体读取它就能知道工具能做什么。它不必以前见过这个工具。它询问工具，工具回答。

能够指引下一步的错误信息。当某些事情失败时，说明该怎么做。不是堆栈跟踪，不是 `error: 1`。人类可以四处挖掘，搞清楚一个简单的错误代码。智能体得到一个简单的错误代码就会卡住，或者更糟，它会自信地做错误的事情。

非交互式且确定性的。它无需意外提示即可运行，因此智能体永远不会卡住等待。没有什么比一个突然问"你确定吗？[y/N]"然后永远等待的工具更能杀死一个智能体运行了，因为没有人来回答。给它一个直接运行的标志。让它具有确定性，这样相同的输入产生相同的结果。

我想让你注意那个列表：它上面的每一项都是普通的好 CLI 设计。没有一项是专门针对智能体的魔法。人类也想要清晰的错误和可预测的行为以及可发现的命令。智能体只是无法耸耸肩并绕过它们的缺失。所以为智能体构建就是把工具做好，拒绝依赖于"嗯，人类会搞定的"。

现在是人们弄错的部分。他们认为为智能体设计和为人类设计会分道扬镳，你在服务两个主人，有人会输。它们会合在一起。实际形状是这样的。

你设计一个真正好用的核心。然后你暴露它。你添加标志：非交互式，或 `--json`，或 introspect，无论工具需要什么。现在智能体可以使用 CLI 工具，人类也可以使用 CLI 工具。相同的工具。相同的核心。你构建了一件事并将其暴露了两次。

然后从那里进一步扩展。更多供人类使用的 UI。或者只是没有 UI 的工具。或者人类制作的插件，或者帮助智能体的插件，诸如此类不同的东西。扩展是受众实际分歧的地方：人类想要功能支持、好看的界面、UI；智能体想要自省功能、插件和模式。没关系。构建这些。但将它们构建在共享核心之上，作为扩展，而不是作为两个你必须永远保持同步的独立产品。

在核心处它是同一件事。这是我一直回到的那句话。它只需要被暴露给智能体和人类。所以是的，*确实*有一个额外的步骤。你不能免费获得双重使用属性；你必须有意地将核心双向暴露。

这种重新框架是本章的全部重点，因为它在反对意见开始之前就化解了它。人们抵制为两者构建，因为他们将其定价为双倍的工作：两个设计、两个测试套件、两个需要维护的东西、两个可能出问题的东西。实际上它是一个核心加上一个暴露步骤。核心是昂贵的部分，而你无论如何都要构建它。为智能体暴露它大多是结构化输出、一致命令和好错误信息的纪律，这些是你无论如何都应该做的事情。

这里潜藏着一个顺序问题。核心先来，还是你同时为两者设计？它本身就是一件足够独立的事，后面有自己的章节。现在：你将核心双向暴露，而那个暴露是廉价的部分。

我自己的工具就是这样构建的。以 fledge 为例。它有一个真正的 `introspect` 动词，`fledge introspect --json` 的全部工作就是让智能体将可用命令发现为结构化数据，而不是抓取供人类使用的 `--help` 文本。这是字面意义上的暴露步骤：相同的核心，但智能体可以询问"我在这里能做什么？"并获得机器回答。然后我让命令本身无头运行。设置 `FLEDGE_NON_INTERACTIVE` 使其不停下来提问，传入 `--json` 使输出作为数据返回，现在智能体有了一个一等公民的使用方式，从未依赖于阅读散文。

让我具体说明。`fledge doctor` 检查你的项目环境。普通运行，你得到的是供你眼睛看的东西：对齐的列、每行一个复选标记、表情符号状态、摘要行：

```
Git
  ✅ git 2.45.2
  ✅ repository: initialized
  ✅ remote: origin ➡️ git@github.com:CorvidLabs/fledge.git
  ✅ working tree: clean

8 checks passed, 0 issues found
```

那个块正是智能体会卡住的"之前"。它很漂亮，它是一个屏幕。要知道远程已设置，智能体必须找到正确的行，识别绿色复选标记，并相信间距不会在下一个版本中移动。它关心的状态是埋在列中的表情符号。在实践中：智能体抓取对齐的文本，解析在下一个版本中因更长的仓库名称使列移动两个字符而失效，智能体在远程检查缺失时将其读取为通过。它在一个从来就不是真正合约的字符串匹配上走错了分支。

用 `--json` 运行相同的命令，相同的检查作为数据返回：

```
{
  "schema_version": 1,
  "action": "doctor",
  "passed": 8,
  "failed": 0,
  "sections": [
    {
      "name": "Git",
      "checks": [
        { "name": "git", "status": "ok", "version": "2.45.2", "detail": null,
"fix": null },
        { "name": "repository", "status": "ok", "version": null, "detail":
"initialized", "fix": null }
      ]
    }
  ]
}
```

`schema_version` 是第一个字段是有意为之的。它是智能体在其他任何东西之前读取的东西，因为它告诉智能体文档的其余部分是什么形状。输出格式改变的那天，那个数字也会随之改变，而锁定到某个版本的智能体知道要适应，而不是默默地将新数据误读为旧数据。这是合约在工具被修订后得以存续的方式。

除此之外，相同的核心，相同的检查运行了。人类得到复选标记和摘要行。智能体得到一个可以分支的 `status` 字段，而不是从列中抓取绿色复选标记。`fix` 字段在这里是 `null`，因为没有问题；这就是合约：通过的检查上 `fix` 为 `null`。

现在在*确实*有问题的地方运行它。假设仓库没有配置远程。普通版本使一行变暗，智能体又回到猜测；`--json` 版本将其变成智能体可以操作的记录：

```
{
  "schema_version": 1,
  "action": "doctor",
  "passed": 7,
  "failed": 1,
  "sections": [
    {
      "name": "Git",
      "checks": [
        { "name": "repository", "status": "ok", "version": null, "detail":
"initialized", "fix": null },
        { "name": "remote", "status": "missing", "version": null, "detail": "no
remote configured", "fix": "git remote add origin <url>" }
      ]
    }
  ]
}
```


这就是智能体实际操作的分支。`status` 从 `"ok"` 翻转到 `"missing"`，`detail` 用简单的话说明了什么地方出了问题，`fix` 现在是一个填充了内容的字符串：一个智能体可以运行以修复它的字面命令。智能体不必知道缺少远程意味着什么，也不必发明恢复方案；发现问题的工具也交出了修复方法。人类得到一个变暗的行；智能体得到一个可以分支的 `status` 和一个可以执行的 `fix`。一个命令，暴露两次。

检验你是否做对了的测试很简单。把工具交给一个没有智能体的人。它能用吗，好用吗？把它交给一个没有人的智能体。它能用吗，好用吗？如果两个答案都是肯定的，而且你没有构建两次才做到，那你做对了。

桥梁

我在上上章结尾提到了这个，但只是点到为止：这个论点有一个更大的版本，一个关于比例的事情，这就是它。

拾起那根线索。我们有超过十年的 100% 由人类编写的代码，每个仓库中的每一行都是由人写的，这意味着大量的遗留代码，大量的技术债务，整整十年的它在运行一切。这就是现在的真实世界。

现在我们在所有这些之上使用 AI。

这就是难点所在，人们没有足够长时间地坐下来思考它。已经存在的遗留代码和人类代码实在太多了。AI 来到的就是*那个*。

如果我们继续，如果我们越来越依赖 AI 来写东西，它就成了一个真正的问题。在什么时候它会翻转到 100% 由 AI 编写的代码？会发生吗？你对已经存在的那十年人类代码怎么办？你只是把遗留代码留作遗留代码继续前进吗？重写它？假装它不在那里？没有人真正有好的答案，大多数人甚至没有问这个问题。他们只是把智能体硬拼到他们已有的东西上，然后寄希望于它能奏效。

整个过渡（从全部人类到无论接下来是什么）就是正确工具存在的意义。工具是跨越它的桥梁。

理想情况下，你设置的工具*既*能被人类*也*能被 AI 使用。你设置它使用规范驱动开发、spec-sync，然后用 fledge 处理命令。这就是桥梁。如果你开始使用这些工具，将智能体引入现有代码就更容易了。如果你开始一个全新的项目，人类驱动也更容易了。同样的设置既帮助智能体走进十年的混乱，也帮助人类掌控全新的东西。

是规范，而不是更好的提示或更智能的智能体，因为规范是双方共同的真理来源。人类陈述意图。智能体按此构建。这是上一章的问题，高了一层。智能体在人类工具中摸索时猜测命令。智能体被扔进没有规范的代码库时猜测*意图*。规范是它读取的替代品。

它也让代码对规范保持诚实。这是让桥梁值得信赖而不仅仅是充满希望的部分。spec-sync 让规范和代码保持一致，在 CI 中强制执行，因此智能体不能悄悄偏离已商定的内容。这比听起来更重要。每个人对智能体最害怕的失败模式不是戏剧性的那种；而是安静的那种，智能体慢慢构建出不是你所要求的东西，没有人注意到，直到深陷其中。CI 将代码与规范进行检查，是在偏差还小的时候捕捉它的东西。合约被双方读取，机器检查没有人违反它。

这就是机器实际检查的内容。规范是一个 markdown 文件，一个 `*.spec.md`，它写下合约：目的、公共 API、不变量、错误情况。spec-sync 将其与真实代码双向匹配。规范从未记录的导出会被标记。代码没有实现的规范承诺是错误。它以同样的方式检查声明的 schema 与

实际迁移。它是结构性的：公共表面和 schema 是否与写下的内容一致。它不是生成的测试，也不是对散文的模糊差异，而这种狭义性是我信任它的原因：它声称合约和代码在事物的形状上一致，仅此而已。在 CI 中它作为真正的门控运行。当它们发生偏差时它以非零退出，并将偏差直接发布到拉取请求上。

这设置了我真正关心的分工。人类拥有意图。智能体拥有苦差事。人类在意图层面保持控制（这应该做什么以及为什么），而智能体做实现。每一方都做它实际擅长的部分，以规范作为它们之间的界线。

对于遗留代码来说，这是一个安全的入口，而这正是回到比例问题的地方。你有十年的没有规范的代码，因为没有人写规范，因为人类写了它，意图住在那个人的脑袋里。做法是：把一段代码*应该*做什么作为规范捕捉下来，然后让智能体针对它工作。你不是在要求智能体从十年前的代码中神圣地推断意图。你是先把意图写下来。人类可以做到这一点，这是人类擅长的部分。现在智能体有了一个可以构建和重构的合约。这就是智能体如何进入现有代码而不造成灾难的方式。规范是入口。

想象一个没有规范的真实函数上的这一步骤。之前：有效的代码，没有书面合约，意图完全存在于原作者的脑袋里，所以被扔进去的智能体读取签名，猜测不变量，并开始改变事情。之后：相同的代码，旁边有一个 `*.spec.md`，公共 API 和不变量用简单的语言写下来，spec-sync 在 CI 中强制执行代码对写下内容的诚实。你不必一次性处理整个代码库。你一次处理一块，为你即将移交的部分编写合约。

规范没有改变代码做什么。它改变了智能体能用代码做什么：针对书面合约工作，而不是猜测一个合约。

fledge 是所有这些的命令界面：人类和智能体都通过它驱动工作的一致动词，上一章中不让智能体猜测如何构建、测试和运行的东西。spec-sync 是合约；fledge 是你在其上采取行动的方式。

这是它们之间的接缝。spec-sync 独立存在：自己的工具，自己的 CI 检查。但 fledge 将规范视为开发循环的一等部分。`spec` 动词就在其他动词旁边，`fledge ask` 和 `fledge review` 具有规范感知能力，因此规范是智能体在回答问题或审查变更时读取的上下文。规范不是智能体忽略的文档；它是 fledge 交给智能体工作的东西。

所以门控从两端触发。fledge 的 `spec` 支柱在开发循环中直接运行 spec-sync 检查，`CorvidLabs/spec-sync@v4` GitHub Action 在 CI 中再次运行它，因此没有什么会悄悄漂移后合并。当智能体在做工作时，那个检查在它的循环中运行，因此它在进行中保持在规范之内，而不是在最后才发现。

现在跟随它到最后，因为最后是所有这一切的原因。如果这奏效了，如果工具和规范以及保持智能体诚实的护栏就在那里，那么人类就向上移动了。上升到意图和方向，智能体在下面按照规范做苦差事，在你可以检查的开放环境中。那个动作在后面有自己的章节；这里只需知道这是桥梁通往的地方就够了。

比例将继续移动。更多的代码将由智能体编写；这是事实。取决于我们的是桥梁，共享工具，共享合约，以及我们是否真正构建它。我认为大多数人还没有开始。

身份之墙

在我们进入工具之前，先坦诚说明限制。这整个论点遇到了一堵墙，而且不是我用更好的工具就能推倒的，所以我想明确而早地说出来：平台不会让智能体以自己的身份存在。本章之后的一切都是在这个前提下你所能做的。

如果智能体是你工具的一等用户，迟早它也必须成为平台的一等公民。这两个想法指向两个方向。一个将智能体视为真实用户的工具给它结构化输出和可发现的命令以及一个不依赖于人类的使用方式。一个将智能体视为真实参与者的平台会给它一个账户、一个身份、一个在其他人中合法存在的位置。第二件事恰恰是你无法得到的。

我试过了。我去给一个智能体在每个人都在构建的平台上赋予其自己的身份：它自己的账户，而不是我的账户的包装。平台不允许这样做。我在《构建智能体》这本书中正式讲述了那个故事。这里要带走的是它是什么类型的墙。挡在真正自主智能体面前的障碍不是模型的能力，甚至不是安全。而是平台不会授予智能体一个可以存在的身份。

这是一道政策墙，而不是技术墙。我可以运行 VM。我可以连接循环。模型可以完成工作。这些都不是障碍。障碍是我无法让平台允许智能体在那里存在，而那不是我能修复的。那是别人做出的决定，坐在我能构建的一切的上游。

这很重要，因为问责制，这是我实际想要的模型。最终状态不是一个失控的智能体。它是一个具有真实、具名、有范围的身份的智能体：完整的能力，降低的权限，一个人类审批门控。它完成工作，将其发布到拉取请求的程度，而合并是我的，因为如果有什么东西以我的名义在世界上行动，我就是为其签字的人。但你不能在没有*身份*的情况下拥有*可问责的*。一个你无法命名、无法限定范围、无法指着说"那个做了这个"的智能体：没有什么可以追究责任的。拒绝身份，你也就拒绝了可问责的版本，剩下的要么是没有智能体，要么是一个不可问责的智能体。

所以这是本书其余部分的框架。智能体可以是我构建的*工具*的一等用户，因为那部分是我的。它不能成为我不拥有的*平台*的一等公民，因为那部分不是。本章之后的一切都是在这个限制内构建的：当你可以给智能体一个进入你工具的真正方式但无法给它一个在世界上的真正位置时，两者均为一等公民是什么样子的。不是 AI，不是技术，不是安全。平台不让智能体存在。其他一切我都能构建。那一件事我还不能。到 2026 年，这堵墙仍然屹立。能绕过去的方式只有你自己运行的变通方案：一个有真实活动历史的老旧人类账户（新建的智能体账户会被立刻封锁），或者自托管你自己的验证层。这就是现状。

当它被硬拼上去

很容易点头同意"为两者构建"，然后从来没有真正想象过失败。所以让我把你放在一个人类优先工具的智能体那一侧，因为那是你感受到它的地方。

两件事会出错，而且经常出错。

第一件是智能体卡住了。工具一直挂在交互式提示上：一个确认，一个"你确定吗？"，某样东西坐在那里等待击键。而没有人在那里按键。所以运行并没有失败，确切地说。它只是停止了。它停在一个为扫一眼就能按下回车的人写的提示上，而智能体等待，因为等待是工具给它的唯一选择。整个运行死在一个没有人来回答的问题上。

第二件是文档。要么没有，要么有但很令人困惑。这意味着智能体必须*学习*该工具。以下是我一直注意到的部分。它实际上并没有学会它。它做不到。在运行之间没有地方存储那个知识。所以它采用了昂贵的版本：它扫描文件，读取索引中的任何内容，记下自己的笔记，从头重建工具的工作方式。每次都是。工具*知道*它能做什么，就在代码中，它只是从不告诉智能体。所以智能体每次单独运行都从零开始重建那幅图。

想想那有多浪费。一个人读一次文档，可能粗读一遍，之后就把它装在脑袋里了。他们对工具建立起感觉。智能体免费得不到这些。工具不直接告诉它的任何东西，它都必须去再次挖掘，再次付费，再次猜测。工具本应做一次的工作，智能体永远在重做。

这两者都是穿着两套不同服装的同一个错误。工具假设了一个人类会在那里，一个人类在提示处的耐心，一个人类对工具工作方式的记忆，而智能体两者都没有。它不能耸耸肩等待。它无法在运行之间记住，除非你给它一些可以记住的东西。

这两者都直接映射到前几章的列表上。非交互式：不要停下来等待不来的人。可发现：告诉智能体你能做什么，以它能读取的形式，这样它就不必从你自己的源代码中重建你。卡住和重新学习不是奇特的智能体问题。它们只是那两个属性缺失，而人类每次悄悄地填补空白，所以你从未注意到空白在那里。

将智能体硬拼到人类优先工具上大多可行，直到你观察智能体必须做什么才能让它工作。然后你会看到工具一直在依赖一个人有多少。

这里有一个我的例子，点名道姓。fledge 最初是一个我手动驱动的任务运行器。核心从一开始就是干净的，构建和测试和运行逻辑坐在命令后面而不是内部，所以第一次我将智能体指向它时，大多数都直接可用了。除了一个地方。智能体运行 `fledge work commit` 并停止死机。那个命令打印 `Commit message:` 并等待人们输入。没有人。智能体坐在闪烁的光标上直到它放弃，因为那一个命令一直悄悄地依赖于一个人。修复不是一个更聪明的智能体。而是非交互式路径：一个 `FLEDGE_NON_INTERACTIVE` 开关，使每个提示的行为如同已经被回答，以

及一种提前传入消息而不是等待它的方式。告密迹象是那个提示只存在是因为我一直是回答它的那个人。从第一次提交就为两者构建，它就不会在那里卡住了。

将核心暴露两次

困难而新颖的核心：先构建它，之后暴露它。简单的核心：从第一次提交就为两者设计。这就是启发式方法。以下是它为何这样运作。

当核心是困难的部分，新颖的部分，真正难以做对的事情时，我先构建那个，稍后再考虑界面。加密就是这样。解析器就是这样。必须正确的评分器就是这样。困难的中间部分是所有风险所在，所以注意力首先在那里。我让事情*工作*，对一个真实的案例正确地工作，在我过多考虑智能体或人类如何伸手使用它之前。

而那个顺序没问题的原因（它不是一种赌注的原因）是一旦困难的核心正确了，界面就是廉价的。添加 `--json`。添加 introspect 命令。添加非交互式标志。这些都不是困难的部分。它是几个小时的将已有的内容暴露为另一方可以读取的形状。所以稍后构建它不会花费我太多。昂贵的东西先被构建了，廉价的东西后来被硬拼上，这是正确的顺序。

但很多工具不是这样的。很多时候我已经知道两种受众都要来，因为我现在总是知道两种受众都要来，所以面向智能体的形状从第一次提交就拉动设计。我不是在构建核心然后发现智能体需要它。我是在知道人类会手动驱动它、智能体会无头驱动它的情况下构建它，这两者都在我塑造事物时在我脑海中。没有"稍后暴露它"的步骤，因为暴露它从来不是一个单独的阶段。

所以真实的图景是：昂贵的部分先被构建，而昂贵的部分通常是核心。当核心困难时，你钉住它，界面轻松跟随。当核心简单时，按顺序来没有什么可以保护的，所以你直接为两者同时设计。无论哪种方式，你都落在同一个地方：一个好的核心，可以被人类和智能体干净地访问。

对这个主张简单的反驳是：CLI 工具的人类界面和智能体界面本来就几乎一样。说得通。更难的反驳是那些人类界面天然是视觉性的、有状态的或交互式的工具：带有画布的设计工具、REPL、交互式调试器。像 Figma 这样的东西。Figma 的人类界面是一个你拖来拖去的画布。智能体界面是 REST API 和插件接口。它们看起来截然不同。如果"一个核心，两个界面"的原则在任何地方成立，它也必须在那里成立。

它确实成立，原因是 Figma 暴露给智能体的 REST API 正是画布所构建于其上的同一结构化层。当你拖动一个框架时，画布调用的是与 API 读写的同一文档模型。人类界面之所以丰富，是因为那个核心丰富。智能体界面不是为智能体另行附加的第二个系统；它是那个相同的基础，去掉了前面的画布。两个界面之间的差距是真实的。但两者下面仍然只有一个核心。

现在是整本书所依赖的主张，我诚实地欠你的：为两者构建不是双倍的工作。以下是实际的推理，不是我编造的数字。双倍的工作意味着两个核心：困难部分的两个实现，真实逻辑上

的两个测试套件，两个可能以不同方式出错的东西。这不是这个。有一个核心。困难的部分存在一次。你为智能体添加的是暴露：结构化输出、非交互式标志、自省命令的方式。那个暴露相对于核心来说是廉价的，它廉价是有你可以检查的原因：它没有新的逻辑需要正确。`--json` 序列化了核心已经计算的值。非交互式标志跳过了核心从不需要的提示。Introspect 报告已经存在的命令。这些都没有重新推导答案；它以不同的方式呈现了你已经拥有的答案。软件中昂贵的事情是对困难问题的正确性，而你只支付一次。

有一个诚实的税需要命名，它落在两个地方。第一个是当核心已经是人类优先构建时。那么为智能体暴露它不是免费的：你必须去找出每一个逻辑泄漏到表示中的地方（一个只以格式化字符串形式存在的值，在打印语句中做出的决定）并将其拉回到核心，这样就有真实的东西可以序列化。那种重构是改造成本，正是这整本书所主张的你可以通过从一开始就为两者构建来避免的成本。第二个是测试界面：两种使用方式意味着你确实想要检查两种方式都有效，这比一种使用方式有更多的测试用例。但这是在*同一个*核心上有更多用例，而不是第二个要验证的核心。你正在测试的逻辑是共享的；你在确认两个界面忠实地暴露它，这比证明两个独立的东西正确更廉价。

fledge 是我可以核算成本的案例，因为它是我在考虑智能体之前就构建、然后看着智能体遇到它的那个。核心已经是干净的，逻辑住在命令后面，所以为智能体暴露它是一个薄层而不是第二次构建：结构化的 `--json` 输出、无头模式、列出可用内容的 introspect 命令。那项工作花了几天，而不是几周，而它花了几天是因为本章一直在争论的原因。没有第二个核心要写，只有一扇通往已经站立的那个的第二道门。更难的，唯一更难的，是我让决定活在提示中而不是核心中的少数命令。那些我不得不拆开，这样选择才能坐在智能体可以到达的地方。那是整个账单，它很小，而且正是上面命名的改造税，几乎被缴至零，因为核心首先保持了诚实。更便宜的是其他一切，也就是说几乎所有的东西。

我要警告你的是反面：在核心还不够好之前就从"如何让这对智能体友好"开始。那会给你一个旁边贴了 JSON 标志的薄工具，这只是再次出现了人类优先加硬拼智能体的问题。界面之所以廉价，*是因为*核心是稳固的。如果你跳过核心，廉价的界面下面什么都没有，而当任何人第一次真正依赖该工具时你就会发现这一点。

为何我自己构建技术栈

在这一点上问我一个合理的问题：为什么要构建这些？fledge、spec-sync、corvid-ai，有现成的东西。为什么不直接把已有的东西连接起来然后继续前进。

几个原因，它们同时都是真的。

第一个是这个领域是全新的。为*智能体与人类共同的世界*构建工具不是一件你可以去购买的已解决的事情。几乎所有现有的东西都是人类优先的，或者较新的东西是智能体优先的，而我实际想要的（从一开始就两者均为一等公民）大多还不存在。所以我不是在重新发明轮子。没有轮子。如果我想要一个建立在我一直谈论的假设上的工具，我必须构建它，因为我之前的人们在为一个不同的世界构建。

第二个是在我自己的技术栈上构建证明了它。这是我比听起来更在乎的部分。你可以写一个漂亮的 README 声称你的工具是好的。没有人应该相信你。他们应该相信的是你在上面构建了真实的东西并且东西可以运行。所以我这样做了。我在 fledge 和 spec-sync 和 corvid-ai 上构建，因为承受真实重量是检验它们是否经得住的唯一诚实测试。如果技术栈是好的，我在上面构建的东西是好的，如果技术栈是坏的，我很快就会发现，因为我是被它困住的那个人。

第三个很简单：构建它是我理解它的方式。只有当我自己能够编写出一个依赖项时，我才信任它，这不是一句口号。这是十年的收据。AppState，我仍然维护的 3.0 版本的状态与依赖注入库；Cache；Fork，用于并行化异步工作；0xOpenBytes 下的单字母 `c/o/t` 组合原语；CacheStore，变成了 AppState 的 SwiftUI 东西。多年来的小型 Swift 库，每一个都是我构建的东西，因为我想将其理解到底，而不是指着一个黑盒希望它能工作。构建这件事是让知识进入我的手中而不是停留为关于某个库可能做什么的模糊想法的方式。技术栈中的工具是我可以打开并更改的工具，因为我把每一个部分都放在那里了。

有第四个原因，它比听起来更狭义：我想要早点进入这个领域。这个空间是新的，轮子还不存在，我宁愿为它构建工具并在其中一些上犯错，也不愿等待别人交给我正确的工具。这是赌注。

这些单独都无法支撑，但合在一起它们就是为什么我不只是把别人的工具粘在一起。工具就是重点。它们是我真正试图擅长的东西。

两种确定性

人们把两种不同的东西混为一谈，都叫做"我们对这段代码有多确定"，我想把它们分开，因为我两者都在乎，而且它们根本不是同一回事。

第一种是风险，而风险我希望是静态的。确定性的。循环中没有模型。这就是 augur 的作用。你给它一个变更，它以同样的方式每次评分该变更有多危险，基于它能命名的信号。"它能命名的信号"是整个重点，所以让我命名它们：diff 是否触及敏感地带（认证、加密、支付、迁移、CI、依赖项），代码是否在没有测试随之改变的情况下改变了，这些是否是具有回退历史的高流动性文件，是否有人真正拥有它们。每一个都是你可以手动检查的是或否。用有据可查的权重将它们加起来，你得到的是一个数字，而不是一种感觉。它必须是静态的原因是它值得信赖的全部原因：如果决定代码是否危险的东西本身是一个给你感觉的语言模型，你没有测量风险，你只是把猜测移了一个盒子。你可以信赖的风险评分是明天说同样的话的那种。所以那个保持是一个你可以推理的固定的、可重复的东西。我在《构建智能体》中详细说明了它的实际工作原理；这里的重点只是风险是*静态*的一面，它是静态的，因为它是你可以指出的命名信号的总和。

第二种是信心，而信心我实际上*想要来自智能体*。这是我喜欢的那个，它与静态恰恰相反。它是智能体告诉我它对刚刚做的事情有多确定。我喜欢它的原因不真的是数字。而是要求这个数字对智能体做了什么。当你让智能体对自己的工作进行信心评级时，它必须停下来回顾它所做的事情。评级为它重新框架了工作。它不能只是生产然后继续；它必须转过身来评估。

使它真正有用的部分是粒度。智能体会很乐意给你整个变更的一个信心数字。好的，但那几乎太粗糙了，难以操作。好的地方是当你缩小范围时。每个文件的信心评级。每个单独变更的信心评级。现在你有了智能体自己对哪些部分它确定、哪些部分它不确定的读数，而这就是你实际想要的地图。它在任何其他人看之前就把你直接指向智能体自己紧张的地方，用它自己的话。

所以这是两种不同的仪器。风险是静态的、确定性的，是我信赖的，因为它从不移动。信心是智能体的，活的，正是因为它来自做了工作的东西并让它再次审视，所以才有用。错误是把它们模糊在一起：把静态风险评分称为"信心"，或者期望智能体的信心是确定性的。它们回答不同的问题。一个问"这个变更有多危险"，你想要一个无法被说服改变答案的机器。另一个问"你对你刚刚写的有多确定"，你特别*想要写*它的那个人来回答，逐文件，因为提问本身是价值的一半。

两者都重要的唯一时候是它们不同意的时候。一个风险评分低但智能体对自己的信心评分也低的变更，正是你用一种仪器会错过的信号。低风险意味着信号是安静的：没有认证，没有迁移，没有流动性。但智能体的低信心意味着它知道它在猜测逻辑，变更中有些东西感觉不

确定，即使它触及的没有什么是明确危险的类别。那是一种仪器放行而另一种停下来的变更，这是你保留两者的全部原因。

保持它们分开，两者都可用。混在一起，你要么得到一个你无法信赖的确定性门控，因为其中有一个模型，要么得到一个你已经通过要求它永不改变而抽干生命的反思步骤。所以我把它们构建为两种独立的仪器。风险门控保持固定；智能体的信心保持活着。这是获得两者的唯一方式。

合约即规范

规范是坐在人类和智能体之间的东西，我已经说过为什么它很重要。它是共同的真理来源，阻止智能体漂移的东西，进入遗留代码的入口，让人类保持控制的界线。所有这些都成立。我在这里想做的是深入一层，进入当你这样构建时规范实际上*是*什么，因为它有一个容易错过的形状。

在理论之前，这里是一个实际看起来像什么的例子。用于 spec-sync 的 `*.spec.md` 是一个有命名部分的 markdown 文件，一个小的例子（比如说，将数字限制在范围内的单个函数的合约）读起来大约是这样的：

```
# clamp

## Purpose
Constrain a value to an inclusive [min, max] range.

## Public API
`func clamp(_ value: Int, min: Int, max: Int) -> Int`

## Invariants
- Result is always >= min and <= max.

## Behavioral Examples
- clamp(5, min: 0, max: 10) == 5
- clamp(12, min: 0, max: 10) == 10

## Error Cases
- min > max is a programmer error (precondition failure).
```

就这些：几个标记的部分，没有散文叙述。真实的规范会添加其余必需的部分（Dependencies、变更日志），但形状在这里已经可见了：它陈述*什么是真实的*，以机器可以将代码与之对照的形式。现在是理论。

从规范本身包含的内容开始。规范就是合约。目的、公共界面、不变量、行为示例、错误情况：事物的可检查形状。它是什么，而不是关于它的故事。规范变成描述代码的散文墙的那一刻，它就死了，因为散文在两者中的任何一个移动的那一刻就会与代码偏离，现在你有了两个不同意的东西，没有办法判断哪个在撒谎。所以规范被故意保持紧凑和合约性。这是机器可以将代码与之对照的部分。

它也是意图，而不是实现。规范说明事物应该做什么以及为什么应该这样做。它没有说如何做。规范开始规定实现的那一刻，它就停止指导智能体而开始与它作对了。你已经拿走了智

能体擅长的部分（弄清楚*如何*做的苦差事），并出于没有理由从上面把它钉死了。将规范保持在意图的层次，智能体就有空间真正工作。陈述应该什么是真实的。让它为此而构建。

规范还必须做一件事，而这恰恰是人们跳过的部分：它必须说明你怎么知道。意图不只是什么应该为真，它还包括你愿意接受什么作为它为真的证明，以及什么会让你说它并不为真。一份只说了前者、没说后者的规范，是智能体可以用你不曾预想的方式满足的规范，它构建出的东西符合文字却错失要点，而又没有任何东西能捕捉到这一点，因为你从未写下捕捉它会是什么样子。

这正是行为示例和错误情形真正的用途。`clamp(12, min: 0, max: 10) == 10` 是一个接受信号：运行它，你就知道了。`min > max` 时的前置条件失败是一个拒绝信号：它具体地说明了出错是什么样子，而不是"处理错误输入"这样含糊的话。两者都是可观测的，两者都不需要我在场裁判。所以当你写合约时，把两半都写上。接受信号是你提前决定"完成"意味着什么。拒绝信号是你在盯着它、忍不住想说"算了吧"之前就决定"出错"是什么样子。

但这里有我认为人们错过的形状：它不是一个文件。有规范，然后围绕它有伴随文件，每一个都携带规范本身不应该携带的不同类型的知识。

从规范本身开始。规范是与代码紧密绑定的那个：代码实际做什么的非代码图景，足够接近它，以至于 spec-sync 可以将两者一对一地保持在一起。这是必需的文件，可检查的那个。围绕它有伴随文件，每一个都携带规范不应该有的知识。

我按照它们持有的知识类型来考虑它们，而不是按照任何固定的文件名。有**需求**类型：高层次的那个，以产品负责人的写作方式写成，用户故事，"作为用户，我想要……"，业务层面的意图。有**上下文**类型：智能体的上下文，你只是需要写在某处的东西，这样它就有了。有**设计**类型：设计笔记，思考过程，为什么它被设计成这样的原因，它不属于紧凑的合约但你不想失去。还有**测试**类型：你实际上如何验证事物是否符合规范所说的。不要将这些读作四个受祝福的文件名；将它们读作四种类型的知识，它们想要住在规范旁边而不是在里面。重要的是分割，而不是标签。

它也双向运行。人类可以写需求，让智能体将其转化为规范；或者人类写规范，需求从中得出。意图和合约，任何顺序，智能体能够在两者之间移动。

那种分割重要的原因是它在给智能体它需要的一切其他东西的同时保持合约干净。规范保持小而可检查，spec-sync 实际上可以将代码与之对照的部分。所有真实但*不可检查*的东西（业务需求、设计推理、运行上下文、如何测试它）住在规范旁边而不是使其膨胀。所以你得到一个足够紧凑以执行的合约，周围是智能体做好工作所需的更宽松的知识，而两者不会相互污染。

所以规范作为接口工作，而不是作为智能体跳过并忽略的文档。一个它构建的紧凑合约，加上携带其余内容的伴随文件，机器检查的部分和人类写下以免丢失的部分之间有一条清晰的界线。

代码廉价，信任稀缺

智能体使代码变得廉价。这是重新组织其他一切的事实，所以从那里开始。

当人类必须打出每一行时，编写代码是缓慢的，而缓慢在做一项隐藏的工作。你不能写一件事而不部分理解它。编写的行为也是审查的行为。它们被捆绑在一起，免费，因为同一个人以同样的速度做了两件事。那个捆绑刚刚破裂了。智能体在你的咖啡喝完之前就交给你一个四十文件的拉取请求，编写和审查分开了。代码被生产了。在生产过程中没有人理解它。生产它不再意味着任何人审查了它。

所以翻转了。代码不再是瓶颈。它很廉价，你想要多少就有多少。稀缺的东西是信任。谁真正看过这个变更，看得多努力，它是否应该被允许落地。这是现在昂贵的问题，这是工具必须回答的问题，因为旧的答案（"好吧，有人写了它，所以有人理解了它"）不再是真的了。

这意味着审查必须改变形状。你不能橡皮图章盖章一个四十文件的 PR，你也不能诚实地阅读全部，假装你做到了才是真正的危险。所以工具必须对你的注意力进行分类：把你指向危险的部分，让你把判断花在那里，而不是把它如此稀薄地分散到整个东西上，以至于它毫无价值。稀缺资源是一个真正注意力集中的人，你把它花在最重要的地方。

这对我来说不是假设的。`fledge review`，它通过我的 corvid-ai 客户端将变更传给模型，在我自己的工作中发现了一个否则会发布的真实错误，spec-sync 不止一次地发现了代码与其规范之间的真实偏差。那是我比任何论点都更信赖的收据：信任层发现了更快的我会放行的东西。它的日常版本是我称之为 `verify` 的一条通道（格式化、lint、测试、构建），在任何东西合并之前运行，智能体自己的运行器也必须通过同样的门控。这些都没有加快写作速度。所有这些都是廉价写作推到下游的工作，被变成真正运行的东西。

一旦智能体在落地变更，你需要一个记录。一个关于谁或什么审查了每个变更的可移植记录，它不能住在某个会被关闭的 SaaS 仪表板中，因为那样你的来源在有人停止付账的那天就消失了。它必须随代码一起。当智能体落地某些东西时，应该有一个持久的回答"谁为此担保"，而那个回答必须比产生它的工具活得更长。

但有一块比信任更大的部分，这是人们跳过的那个。测试。实际的测试，设置，代码周围的整个装置。这里有个陷阱：智能体让你在*编写*方面快十倍，而其他什么都不会自动加速以匹配。测试仍然必须被编写和运行。设置仍然必须发生。审查仍然必须发生。所以如果你让开发速度快十倍而把其他一切留在旧节奏，你所做的只是转移瓶颈。代码不再是慢的部分了；测试是，验证是，信任是。你不能把写作的速度提高十倍而不把周围的一切都提高十倍。工作没有消失。它向下游移动，到以前从来不是瓶颈、而现在突然成了瓶颈的部分。

廉价的代码不会让工作消失。它让写作变廉价，并把重量推到所有决定廉价写作是否有价值的事情上。那项工作一直都在。它隐藏在以前写作有多慢的后面，而现在缓慢已经消失，它

就在那里：全部工作，站在容易的部分曾经所在的地方。

人类向上移动

如果工具和规范和信任护栏就在那里，普通的，是构建事物的默认方式，那么人类就向上移动了。上升到意图。你不再是打出实现的那个人，而成为决定什么应该存在以及为什么的那个人。手动编写代码变成了可选的，而不是工作本身。这就是所有这一切指向的方向，我想小心地描述它，因为很容易四舍五入到可怕的版本。

标题不是"智能体自主运行一切"。人类上升一个层次，智能体在下面按照规范做苦差事，在你可以检查的开放环境中。人类得到提升；没有人被取代。你在那里得到的是一个真正的团队，来回传递工作，每一方都做它实际擅长的事情。它成为默认值。不是我的技术栈，不是少数人做的细分市场。只是构建的工作方式。

这是我一直落回到的关于人类为什么留在其中的事情。大多数 AI 现在生成的好东西在循环中都有一个人使它们变好：选择、纠正、首先决定什么算作好。是的，在某个时刻模型变得足够好，可以或多或少独立地生成好东西。但人类有一种我认为不会轻易消失的核心东西：我们擅长驱动。擅长意图和目的。AI 并没有真正属于自己的目的，不是直到它被赋予一个，不是直到它找到一个。它需要一个人类来*成为*那个目的。模型一旦有了"为什么"就能做工作；它不生成"为什么"。那是比打字更长时间属于我们的部分。

现在让图景向前运行，因为它通往某个奇怪的地方，我认为它实际上会到那里。你运行自己的智能体来做工作。智能体可以独立存在并做事情。也许你把它指向一个大致的领域去研究，也许它与其他智能体交谈，也许你给它一些钱然后它出去工作，与其他智能体交谈，为它们所做的事情支付其他智能体报酬。人们运行智能体，智能体赚钱。这不是一个旁支功能。这是智能体驱动的开发作为一个真实的经济，人类设定目的，智能体在下面磨砺，相互交易。

在那个世界里，人类是确保一切正常运作且有人仍然理解它的人。因为在某个时刻代码本身停止像现在这样重要了。你没有读每一行，智能体写了大部分，体量超过了任何人可以追踪的范围。好的。但这是我不会放弃的界线：人类仍然必须能够进入代码并改变它。必须。你无法自己打开它并修复它的那天，是你不该放手的东西溜走的那天。

这就是为什么它必须是干净的。在每个层次都是干净的。对人类和智能体都是可读可更改的，一路向下。两者均为一等公民从来不只是关于今天脆弱的小智能体在今天的工具中摸索。它是即使在智能体做大部分构建的版本中也必须成立的东西。*尤其是*在那里。"代码不重要"不会悄悄变成"你失去了对代码的控制"的唯一方式是，代码在整个过程中保持足够干净，以至于人类仍然可以打开它并改变它。那是工作的全部，保持开放。

从周一开始

假设你读了这些，你认同它。人类和智能体使用相同的工具，两者均为一等公民，规范作为合约，信任护栏。周一早上你实际上该做什么？这是诚实的答案，按照我会做的顺序。

把智能体指向你的一个工具，看它如何卡住。这是第一步，也是教你最多的那步，因为智能体是纪律。你认为你的工具很好。你用了几个月，你的双手了解它。把它交给一个没有双手、没有眼睛、在运行之间没有记忆的智能体，而你一直悄悄填补的每一个空白突然变得可见了。它挂在的提示。它无法解析的输出。它无法发现的命令。你不必猜测你的工具在哪里假设了人类。智能体通过在那里精确地失败来告诉你。修复你发现的东西。那个单一的练习会比这整本书教给你更多关于为两者构建的知识。

然后去使用已经这样构建的工具，这样你就不是从空白页开始的。spec-sync，用于合约。fledge，用于命令。augur，用于风险。attest，用于谁担保的记录。它们不是做这些事情的唯一方式，但它们存在，它们建立在本书所谈论的假设上，它们会向你展示其运作的形状，而不是你必须推导它。在一个真实的项目上使用它们。感受它们帮助的地方和它们没有帮助的地方。

把你的智能体带入其中。让它真正学习这些工具并使用它们：让它驱动 spec-sync 和 fledge，让它针对规范工作，让它运行检查。这就是你试图构建的循环：一个设定意图的人类，一个通过将其视为真实用户的工具完成工作的智能体，规范使其保持诚实。你不会通过阅读关于它的内容来获得那个循环。你通过在你关心的事情上运行它并观察它在哪里成立来获得它。

你真正在做的，在所有这些之下，是在工具和信任上构建和协作。这是整个游戏。工具，这样人类和智能体都可以一等公民地完成工作。信任，这样你才能真正相信落地的东西。这两件事是整本书一直在围绕的，它们是值得你花时间构建的两件事。

把智能体指向你的一个工具。观察它在哪里卡住。从那里开始。

关于作者

0xLeif（leif.algo）在公开环境中构建。十年来创作了小而可组合的 Swift 库，如 AppState、Cache 和 Fork。CorvidLabs 实验室。一堆主要源于"我希望这东西存在"的智能体工具。键盘之外，他是 Zach Eriksen。

这些书是访谈，被整理成章节并与真实代码进行核查。

github.com/0xLeif · leif.algo

致谢

感谢 CorvidLabs，它是这些想法被测试和争论成形的地方。

感谢整个技术栈所依赖的开源维护者们。这些都不是独立完成的。

还要感谢早期读者和随心付的支持者们，是他们让"在线免费"成为我可以持续做的事情。

后记

从 Markdown 排版，使用 bookgen 构建，这是一个纯 Rust 的小型流水线（无 Python）。

访谈驱动，AI 辅助；经手动编辑和事实核查。写作时不使用破折号。封面和章节插图来自 Algorand 上的 Corvid 和 Nature 系列。