开源工具

构建真正有人使用的工具

ZACH "LEIF" ERIKSEN

版权声明

可在线免费阅读。ePub 版本采用随意付费模式；如果本书对你有帮助，欢迎支持。

github.com/0xLeif · leif.algo

本书是"智能体技术栈"系列四册之一。制作方式详见书末版权页。

献辞

献给所有在开放环境中构建、并坚持发布的人。

书系

这几本书各自独立，但作为一套而写。代码变得廉价，信任却愈发稀缺。合在一起，它们构成一个论点：现在该构建什么，以及如何信任它。

- 《**智能体开发者野战手册**》：为能够交付真实代码的智能体构建工具、规格说明与信任机制
- 《**一等公民**》：为人类和智能体同等构建
- 《**构建智能体**》：尝试赋予软件自己双手的笔记
- 《**开源工具**》：构建真正有人使用的工具 *(本书)*

可在线免费阅读。每册 ePub 均采用随意付费模式。

目录

前言

本书讲述的是构建工具的技艺：那些其他人、以及其他智能体真正会用的工具。

本书围绕 fledge 展开，这是一个覆盖整个开发生命周期的命令行工具，而它不断迫使我回答一个问题：一个工具值得被依赖的关键是什么？我的结论是，对人类和智能体而言，答案是一样的。一个好工具有一个干净的接口，没有隐藏的交互步骤，有真实的插件边界，有一份随代码变化而保持诚实的规格说明。做到这些，智能体从第一天起就能驱动它，因为它从来就不是专门为需要人类介入而设计的。

这是两本"证据之书"中的第二本。《一等公民》论证软件应该对两者都是一等公民。《野战手册》将其转化为方法论。《构建智能体》展示智能体本身。本书展示其背后的工具链，深入探讨为何 Rust 是正确的选择，以及 WASM 沙箱如何阻止插件做出不该做的事。

本书适合所有曾经发布过工具、并看到它被以意料之外的方式使用的人。你不需要 fledge。你需要的是那些让工具能够在与用户接触后存活下来的习惯，包括那些甚至不是人类的用户。

贯穿整个生命周期的单一 CLI

每个代码仓库都有自己的方言。不同的 Makefile，不同的脚本， 不同的 README，每一个都有自己关于如何构建、测试和运行 的一套想法。这些经验完全无法迁移。你打开一个久未触碰的项目， 第一件事就是重新摸清本地的"咒语"。哪个脚本，哪个目标，什么顺序。 这件事本身并不难。运行测试并不难。问题在于到处都不一样， 而这种差异纯粹是开销。

所以我想要一个在所有项目中保持一致的接口。这就是 fledge 的起点。代码仓库里的说明简洁明了：*一个 CLI，贯穿你整个开发 生命周期。*

推动我走到这一步的，其实有三件事。

第一是那个"千仓千面"的问题，每个仓库说着自己的语言。 第二是引导启动。我不停地把相同的设置和脚手架复制粘贴到每个新项目里。 同一堆文件，同样的接线，一遍又一遍，在我能开始真正想做的事之前。 第三是规模。我同时启动了大量项目，有智能体在上面工作， 我需要一个它们都能依赖的单一 CLI。不是"我使用的工具"， 而是一个所有人、所有在这些仓库上工作的事物都能以同样方式倚靠的工具。

最后一点很容易被忽视。fledge 不是因为某个让我烦恼的仓库而诞生的。 它是因为有大量仓库、大量进行中的项目、以及参与其中的智能体， 而这一切都需要对接一个一致的接口，而非一百个定制接口，才诞生的。 "智能体可以依赖单一 CLI"这个角度自成一条线索，我在智能体那本书里 深入探讨了智能体侧的问题。一致性、脚手架和规模：这就是起源。

为人类和智能体双重构建

这一切之下有一个理念，它比 fledge 本身更宏大。我的很多 工具来自这样一个信念：人类和智能体将会使用同样的工具。

现在大多数存在的工具都是以人类为先的。项目为人类构建， 然后我们再事后尝试把智能体引入进来。可以有以智能体为先的东西， 也可以有以人类为先的东西，但我们真正需要的是构建 以智能体和人类同时为先的项目，从一开始就让双方都是一等公民。

这就是我所有工具的共同之处。一个工具应该双向均为一等公民： 人类可以不借助智能体使用它，智能体也可以不借助人类使用它。 双方都不是事后补救的存在。当智能体使用它时，工具应该*帮助*它， 而不是让它猜测所有命令和一切的工作方式。

将人类和智能体同等视为一流调用者，正是 fledge 呈现如此面貌的真正原因，也是贯穿本书其余部分需要铭记的一点。

"零配置"究竟意味着什么

当我说零配置，我的意思是你不必在工具帮到你之前先教它你的项目。 你把它放进一个仓库，它就能工作。这有三个要素。

它自动检测项目及其命令。Swift、Rust、Node，无论什么。它 弄清楚面对的是哪种类型的项目，并知道正确的构建/测试/运行方式。 没有你先向配置文件描述项目的设置步骤。

同样的动词在每个项目中都有效。`build`、`test`、`run`、`lint`， 同样的词，无论底层是什么。这是"每个仓库说自己方言"问题的直接回报。 方言仍然存在于底层；Cargo 还是 Cargo，Swift Package Manager 还是 Swift Package Manager。但你不再需要关心自己站在哪一个面前。 你学一次动词，它们在任何地方都是这些动词。检测机制负责翻译， 下达到真正的底层工具。

这就是人类和智能体理念的具体体现。人不再需要重新学习每个仓库。 智能体不再需要*猜测*。它不必去寻找这个特定项目的正确命令， 或者读 README 然后寄希望于它。它运行 `build`，运行 `test`， 工具已经知道这在这里意味着什么。为我省去重新学习的东西， 同样也让智能体不需要去猜测。fledge 首先为我和我运行的智能体而构建， 并在其他任何地方之前先证明自己的价值。

它通过插件扩展。一个全新的仓库获得核心部分，你通过放入插件 来添加能力。核心知道通用动词以及如何检测常见项目类型； 超出这个范围的一切都来自插件。所以零配置并不意味着"开箱即做一切"。 它意味着你开箱即得的部分无需配置，你从那里通过添加插件而非编写配置来扩展它。

具体来说，第一次运行是这样的。你进入一个仓库并运行 `fledge`， 它自动检测项目（Swift、Rust、Node，无论什么），直接知道 正确的构建/测试/运行方式，无需配置步骤。要求它自省，它会 *告诉你这个仓库可用的动词*：它是自描述的，所以你（或智能体） 不必猜测可以做什么。如果是一个全新项目，第一个真正的动作 通常是从模板搭建脚手架，而非检测现有项目。整个工具从第一条命令 起就能无头运行：设置 `FLEDGE_NON_INTERACTIVE`，使用 `--json`， 智能体从第一步就能驱动它。检测、自省、可能搭建脚手架，然后运行。

生命周期的部分

代码仓库将 fledge 称为一个带脚手架和 AI 评审的零配置任务运行器。 任务运行就是上面的动词集合：日常的构建/测试/运行/lint。 脚手架是对第二个起源问题的解答：用模板搭建新项目，而非手动 复制粘贴拼凑。生命周期中也内置了 AI 评审部分。

任务运行是最先出现的。它是种子，是其他一切生长于其上的部分。 脚手架实际上是模板：从模板搭建新项目，而非复制粘贴。 AI 评审被折入进来，是因为评审是开发循环的一部分。如果智能体在写代码，评分它就不是一个你去别处运行的独立仪式， 它只是紧挨着 build 和 test 的另一个动词。一个接口胜过三个工具， 对我如此，对一个原本需要学习三件事的智能体来说更是如此。 这三个是结构支柱，是核心，插件作为围绕它们的扩展层。

fledge 自己的一句话是"一个 CLI，贯穿你整个开发生命周期， 零配置任务运行器、项目脚手架、AI 评审，以及更多"， 这正好是这三个部分。它是一个单一的 Rust 二进制文件， 获取它的方式简单明了：从 Homebrew tap 安装 `brew install CorvidLabs/tap/fledge`，或者如果你更喜欢， `cargo install fledge`，或者一个 shell 脚本。安装它没有什么特别之处。

我实际上如何使用它

这不是一个我偶尔使用的工具。它在每个仓库里，每时每刻， 是我现在构建、测试和运行一切的默认方式。我的智能体驱动它。 智能体运行 fledge 的次数比我手动运行的次数更多。它是为了驯服大量仓库而构建的，现在它是我和我在这些仓库上运行的 智能体都倚靠的单一 CLI，无处不在。

这也是下一个部分重要的原因。一个人人都倚靠的 CLI， 任何人都可以扩展，智能体安装和运行它，它不能是我一次性 认可的固定功能集合。它必须可以被任何人用任何他们想要的语言扩展， 而不触及核心。一旦你允许任意插件，一旦智能体是运行它们的那个， 你就必须认真考虑信任问题。这是下一章的内容。

为什么选 fledge 而非 Make

人们一听到 fledge 是做什么的就会问这个问题。你构建了一个任务运行器？ 有 Make。有 Just。有 Turborepo。每个 package.json 里都有 npm scripts 区块， 就摆在那里。为什么要再写一个？

诚实的答案是，让我碰壁的从来都不是 Make。我是作为一名 iOS 开发者成长起来的，在那里自动化构建的道路总是通向同一个地方： fastlane，也就是 Ruby。我不想用 Ruby。我是一个 Swift 人。我想用 我工作的语言来自动化我的构建，但整个生态系统却把我推进了 一个 Ruby DSL 和一个 `Gemfile`，以及一个需要独自学习的小世界。 每次我想发布什么东西，答案都是"写一个 fastlane lane"，而每个 fastlane lane 都是我不想写的 Ruby，坐在我不想管理的运行时上， 做一件我觉得应该能用 Swift 完成的工作。这才是 fledge 真正生长出来的痛点。 不是"Make 很笨拙"。而是"为什么我被迫进入别人的语言来自动化我自己的项目？"

所以当人们把 fledge 与 Make 和 Just 对比时，他们瞄准的目标是错的。 答案不是那些工具不好。我从任务运行器那里想要三件事， 而 fastlane 那堵墙是所有三件事的来源。

我想要它符合我的方式，用我的语言，按我的条件，而不是像 fastlane 把你锁在 Ruby 里那样锁定在某个运行时。任务运行器是你每天触碰一百次的东西， 久而久之你就不想再生活在别人关于它应该如何感觉的选择里， 或者你必须用哪种语言才能使用它。我想要 `build`、`test` 和 `run` 在每个仓库里都意味着同样的事情，我想要底层步骤可以用任何适合工作的语言编写， 而不是被强行推入 Ruby。Make 不会阻止你，但 Make 也不会给你这个。 你在每个 Makefile 中手动建立一致性，永远如此。一个让我在每个项目中 重新发明方言的工具并没有解决我的问题；它只是给了我一个更好的地方来 不断重新发明它。

另外两点来自第一章，而在 fastlane 文件那里我也会想要它们。 这些工具没有一个是智能体优先的（默认 JSON 输出、自省、无头模式）， 这是我现在最关心的部分，因为我的智能体驱动这个东西的次数 比我手动驱动的次数更多。也没有一个是可以放进仓库而无需先安装运行时的 单一轻量二进制文件，这正是一个在一堆不同语言仓库上保持一致接口的东西 必须具备的。一个 fastlane lane 需要 Ruby；npm scripts 需要 npm； 一个我用 TypeScript 写的运行器需要 TypeScript 运行时。fledge 什么都不需要。

以上都不是"Make 不好"。Make 在 Make 擅长的事上很出色，Just 是一个干净的 命令运行器，Turborepo 能很好地缓存 JS monorepo。如果那些工具之一 就是你所需要的全部，就用它吧。我不会假装 fledge 在它们的主场 赢得逐项功能的比较。

但它们没有一个是我多年前站在 fastlane 文件前真正想要的那个东西： 一个接口，智能体优先，一个轻量单一二进制文件，以及用任何合适的语言 编写步骤的自由，而不是被推入 Ruby。fledge 是我那时希望拥有的工具， 终于构建出来了。

fledge 与 MCP

MCP 现在已是通行标准。2026 年，智能体大多通过 MCP 服务器使用工具。 如果你构建一个工具并希望智能体能可靠地使用它， 阻力最小的路径就是暴露一个 MCP 服务器。这就是生态系统的现状。

fledge 没有 MCP 服务器。然而智能体在数十个仓库中持续驱动它， 并且它能正常工作。这个原因值得明确说清，因为它告诉你应该先构建哪个部分。

CLI 是基础原语

MCP 服务器是一个生产级接口。它处理请求路由，它记录日志， 它让你对智能体请求了什么、收到了什么有结构化的可观察性。 这些都是好东西。但 MCP 服务器是你叠加在某样东西上面的一层。 问题是：叠加在什么上面？

如果你先构建 MCP 服务器，把 CLI 当作事后补充，你得到的工具 对智能体有效，对人类却是个麻烦。如果你先构建 CLI，你得到的工具 现在就对智能体有效，无需协议适配器，对人类也有效， 并且可以在任何需要的时候在前面加上 MCP 服务器。 CLI 是基础原语。其他一切都架在它上面。

fledge 就是带着这个理念构建的。每个命令都支持 `--json` 输出。 有 `FLEDGE_NON_INTERACTIVE` 用于无头执行。`fledge introspect` 给任何 调用者，无论是人类还是智能体，提供一个结构化的清单， 包含每个可用动词、它的功能和它接受的内容。没有会阻断无人值守运行的 交互式提示。工具描述自身。

这个特征（默认结构化输出、明确的能力清单、没有隐藏的交互步骤） 正是 MCP 工具接口给智能体的东西。fledge 无需协议就拥有这一切， 因为这些属性来自从一开始就为智能体调用者设计，而非事后包装工具 使其对智能体友好。

今天一个驱动 fledge 的 Claude 实例调用 `fledge introspect`， 获取可用内容的 JSON 清单，选择正确的动词，传入 `--json`， 并读取结构化输出。这与 MCP 服务器将要中介的交互循环相同， 只是少了 MCP 框架。CLI 已经是干净的工具接口。

MCP 不是竞争对手

有时出现的"CLI 还是 MCP？"这个框架将两者视为替代品。 它们不是。MCP 是一个传输和协议规范。CLI 是完成工作的东西。 问题不是该拥有哪一个；问题是先构建哪一个，哪一个叠加在上面。

先构建 CLI，那种能描述自身、输出 JSON、无头运行的类型。 一旦有了这些，在上面暴露 MCP 服务器就是简单的事： 将每个 `fledge introspect` 条目映射到一个 MCP 工具定义，在底层调用 CLI，将 JSON 输出传回去。核心已经是在明确能力清单 的干净边界上的结构化 JSON。MCP 服务器是在同一基础原语之上的 生产级 API，就像 REST API 可能位于结构良好的库前面一样。 接口改变了；工作没有。

因为 CLI 是真正的接口，任何可以调用子进程的东西都可以在 没有协议适配器的情况下使用它。shell 脚本可以使用它。 CI 运行器可以使用它。在终端前的人类可以使用它。 MCP 客户端可以通过服务器层使用它。这些调用者都不需要 其他调用者的存在。你从基础原语获得普遍性，而非从协议。

MCP 带来了什么

这些都不是反对 MCP 的论点。一旦 MCP 放在 CLI 前面，它确实带来了真实的东西。

日志和可观察性。MCP 服务器位于智能体和工具之间， 所以你可以记录每次调用，计时，检查参数，标记异常。 直接调用 CLI 给你的只有你导入日志文件的内容。 MCP 层在不单独检测每条命令的情况下给你结构化的可观察性。 对于你想审计智能体请求了什么的生产用途，这很重要。

协议层面的可发现性。MCP 有一种标准化的方式让智能体 询问"这里有什么工具？"并获得结构化答案。fledge 有 `fledge introspect` 做同样的事情，但 `fledge introspect` 要求知道 fledge 在那里。MCP 注册表让智能体在不了解底层工具 的情况下通过标准握手发现工具。

跨工具的组合。MCP 服务器可以通过一个端点暴露多个底层工具， 这样智能体只需一个连接点，而不必了解各个单独 CLI 的列表。 当工具数量庞大时，这是一种操作便利。

这些是生产基础设施的论点。它们适用于你在规模上运行智能体、 审计其行为，并通过托管层将其连接到广泛工具接口的场景。 MCP 服务器作为标签来说，是一个带有日志的面向 AI 的 API。 这是有用的东西。只是不是你首先要构建的东西。

操作顺序

先构建 CLI。它存在的那一刻起对每个调用者都有效。MCP 服务器是当日志和协议标准化值得这一层时才添加的包装器，而不是之前。

对于 fledge，CLI 是基础，一个运行命令的人类、一个驱动仓库的智能体， 以及一个与下游客户端通信的 MCP 服务器，都架在上面。 智能体的故事不是"智能体通过 MCP 使用 fledge"。 而是"智能体像其他一切一样使用 fledge，因为 CLI 从一开始 就为任何调用者构建为一等公民"。

当 fledge 的 MCP 服务器存在时，它将是一个薄层。 工作已经在核心中完成了。这就是先构建基础原语的意义。

脚手架与模板

推动我走向 fledge 的第二件事，在"千仓千面"问题之后， 是引导启动：我在第一章提到的"复制粘贴相同设置"的痛苦。 近距离看是这样的。你决定做一个新的小型 Rust CLI， 第一个小时不是 CLI 本身。是 Cargo 布局、配置、lint 设置、 CI、README 框架，所有你已经打了二十遍的东西。这就是税， 而我一直在交，因为我在启动大量项目。

所以脚手架是支柱，而非附属功能。从零搭建一个项目 是生命周期的一部分，就像构建和测试它一样。 整个理念是：从模板搭建新项目，而非手动复制粘贴拼凑。

在 fledge 中，这在 `fledge templates` 下。你用名称和模板运行 `fledge templates init`，它会为你搭建项目：

```
fledge templates init my-tool --template rust-cli
```

有一个内置集合，涵盖了我实际启动的项目类型。发布的有 `rust-cli`、`ts-bun`、`python-cli`、`go-cli`、`ts-node`、 `static-site`、`kotlin-kmp` 和 `kotlin-ktor-api`。这份列表 基本上是我工作语言的地图：一个 Rust CLI、几种 TypeScript 风格、 一个 Python CLI、一个 Go CLI、一个静态网站，以及 Kotlin Multiplatform 和 Ktor API 两种。 这八个是今天仓库中完整的内置集合。fledge 的检测侧知道如何处理 项目存在后的 Swift、Rust、Node 等；模板侧是项目首先存在的方式。

我最在乎的部分是模板不是我必须逐一认可的封闭列表。 你可以从任何 GitHub 仓库搭建脚手架，而不仅仅是内置的。 `--template user/repo`，它从那里拉取模板：

```
fledge templates init my-app --template user/repo
```

核心附带一个小而实用的集合，超出这个范围你可以自己扩展， 无需我介入。模板只是某人已经想清楚的一种项目形态， 如果它存在于 GitHub 仓库中，fledge 就能从它搭建新项目。所以模板集合不是"CorvidLabs 发布的"。而是"任何人曾经 放进仓库的每一个起点"。

围绕模板有一套完整的动词，不仅仅是 `init`。有 `fledge templates create` 用于创建模板，`fledge templates list` 和 `fledge templates search` 用于查找，以及 `fledge templates validate` 用于在你依赖模板之前检查它是否 真的结构完整。如果我要从模板搭建脚手架，特别是如果是智能体在做， 我想在它用同样的错误東西生成一百个项目之前知道模板是完好的。

它检查的不只是"能否解析"。它读取模板的 `template.toml` 清单， 确认名称和描述不为空，并检查模板声称需要的工具是否真的在 你的 PATH 上。然后它遍历模板中的每个文件和每个文件名， 对它们运行占位符模板化，这样一个错误的占位符（语法错误， 或清单中从未定义

的变量）就会在你从中搭建脚手架之前被捕获为错误。 它还标记没有文件的模板，并在清单未设置为从输出中排除时发出警告。 所以它是结构性的，但超出了"文件在那里"：它实际上以 `init` 的方式 执行模板化，并告诉你在哪里出错了。

这就是为什么脚手架属于这个 CLI 而非旁边某个独立的生成器工具的真正原因。 与 fledge 中其他一切相同：同一个接口，对同两类用户。 一个人运行 `fledge templates init`，跳过了无聊的一小时。 一个智能体运行完全相同的命令，无交互地，从第一步搭建一个全新项目， 无需人类点击向导。从第一章来看，检测、自省、可能搭建脚手架，然后运行， 脚手架步骤是那个"可能"。当仓库已经存在时，fledge 检测它。 当它还不存在时，第一个真正的动作是从模板搭建它， 然后其他一切（build 动词、test 动词、review 动词） 立即在它上面工作，因为它从模板出来时就已经按 fledge 期望的方式接好线了。

一个从诞生之初就说 fledge 语言的项目，才是我真正追求的东西。不仅仅是"省去复制粘贴"，尽管它确实做到了。一个搭好脚手架的项目，能构建，能测试，从第一次提交起就准备好了与其他每个仓库相同的动词。 我过去手动粘贴的样板，有一半时候正是让项目与我其他项目保持一致的接线。 将脚手架折入生命周期 CLI 意味着一致性是项目天生就有的默认值，而非我事后追加的东西。

所以是的，一个新项目从脚手架开始。即使我不专门调用 `fledge templates init`， 从第一次提交起这个项目也会有同样的设置接好线：spec-sync、fledge、augur、attest。 真正的"初始化"不是模板，而是那个技术栈进入项目的过程。命令只是到达那里的快捷方式。 无论哪条路，我的一个新项目天生就持有其他每个项目都有的工具。

AI 评审融入循环

第三个支柱是人们惊讶地在任务运行器中发现的那个。任务运行、脚手架， 好的，这些显然是生命周期的事情。但 AI 代码评审？ 在你用来构建和测试的同一个 CLI 里？感觉它应该在别处， 在它自己的工具里，在 CI 里，在你拉取请求上的机器人里。

但事实并非如此，原因很简单：评审是循环的一部分， 就像构建和测试一样。你写点东西，检查它，修复它，再来一遍。 评审是检查步骤。如果智能体现在是写代码的那个，对我来说大多如此， 那么评估那段代码就只是另一个动词。它紧挨着 `build` 和 `test`， 因为它做着相同的工作：它在你继续之前告诉你这个东西是否真的好。

一个接口胜过三个工具，对智能体来说比对我胜得更多。这就是将评审烘焙进生命周期 CLI 而非发布独立评审工具的全部论点。 一个智能体在一个独立的评审工具里必须学习完全不同的东西， 有自己的调用方式和自己的输出形态，才能完成一段连续的工作循环。 如果智能体已经知道如何驱动 fledge 来构建和测试，那么 `fledge review` 就是一个它已经知道形态的动词。同样的接口，同样的 JSON，同样的无头模式。 只是因为步骤从"能否编译"变成了"是否足够好"，就不需要学习新工具。

在 fledge 中这是 `fledge review`，它做的是对默认分支进行 AI 代码评审。 所以它不是在评审整个世界。它看的是发生了什么变化， 你的 diff 相对于你要合并到的分支，这正是评审实际发生的单元。 与人类评审者或 PR 机器人会查看的相同范围，在你已经生活在其中的 同一个 CLI 里作为一个动词运行。

它不绑定于一个模型或提供商。在底层，评审通过我其余工具链使用的 同一个多提供商客户端进行，corvid-ai，所以 fledge 与 corvid-ai 支持的 任何提供商通信，Anthropic 的 API、任何 OpenAI 兼容端点， 以及 Ollama 等本地运行器，以及其他。支持的提供商列表在 corvid-ai 仓库中， 随生态系统变化而变化。重点是评审针对你想要的任何模型运行，由你决定， 而非工具决定。

值得直白说明这意味着什么，因为本书其余部分对爆炸半径直言不讳： `fledge review` 会取你的 diff 和你的规格说明，并将它们发送到你指向的 任何提供商。如果那是一个托管端点，你未合并的代码就离开了大楼。 对私有仓库来说，这是一个真实的数据出口风险面，这是你需要睁开眼睛 做出的决定，而非一个可以轻描淡写的细节。将它指向本地模型 是没有任何东西离开机器的版本。关于那个本地情形有一个诚实的说明： Ollama 路径很乐意接受一个密钥和一个云 URL，但 corvid-ai 的 README 仍然将 Ollama 描述为本地的、无需密钥的选项，所以文档读起来像 Ollama 意味着你桌子下面的机器。云路径今天就能工作；README 只是 还没跟上它。这是我这边的文档缺口，而非缺失的功能。

还有一个我非常喜欢的多模型角度。`--with-model` 让你运行一个小组， 对同一个 diff 同时从多个模型进行平行批评。

```
fledge review --with-model ollama:gpt-oss:120b-cloud,ollama:qwen3-coder:480b-cloud
```

这不是噱头。如果你花过时间使用这些模型，你知道它们不会发现相同的东西， 也不会在相同的事情上产生幻觉。对同一个 diff 运行其中几个，看看它们在哪里 一致，以及其中一个标记了其他人遗漏的东西，这是比信任任何一个更好的信号。 这是第二和第三意见，并行运行，针对你面前的确切变更。

和 fledge 中其他一切一样，输出是结构化的。每个命令输出 `{schema_version: 1, ...}`。默认 JSON。所以评审不是一堵散文， 智能体必须像人一样阅读和解释它。它是数据。 写了代码的智能体可以运行评审，获取结构化的发现，并在同一个循环中采取行动， 中间无需人类将"评审者似乎对错误处理不满意"翻译成实际要做的事情。 一个人可以阅读评审，一个智能体可以解析它，来自同一条命令。

`fledge review` 也是规格感知的，规格感知部分在 spec-sync 章节有完整介绍； 这里只需看到一旦评审有了规格说明会做什么。评审弄清楚哪些规格说明覆盖了 diff 中的文件，根据规格说明声明的文件和 `specs/<name>/` 目录进行匹配， 并将这些规格说明作为背景折入提示中。它是被有意指向它们的： 模型被告知规格说明描述了模块*应该*做什么，用它们来解释变更， 只评审 diff 而非规格说明本身，并且，重要的是，如果 diff 与规格说明的不变量相矛盾， 将其标记为 diff 中的 bug。所以与规格说明的漂移不是批评中的背景佐料； 它是评审被明确告知要浮现的发现。

它赢得了这个位置。`fledge review` 为我捕获了一个真实的 bug。 一个本会发布的东西，就在 diff 里，构建正常，测试也通过， LLM 在它合并之前就标记出来了。这是评审作为动词是否值得的检验， 它通过了：不是风格上的吹毛求疵，而是循环其他部分都漏过的真实缺陷。 规格感知的一面也带来了回报。将评审者指向规格说明， 捕获了悄悄偏离模块应该遵守的契约的代码，这种漂移就像那个 bug 一样 能编译也能通过测试。智能体能很好地依赖它，因为对它们来说 它的形态与 build 和 test 相同：运行它，读取结构化发现，修复找到的东西，再来一遍。

如果智能体写代码、运行构建并运行测试，那么评估代码就是它们触手可及的 又一个动词，而且它能捕获其他步骤放过的东西。

插件，以及不信任它们

fledge 有一个几乎什么都不做的小核心，所有真正的能力都在插件中。 这是有意为之的。如果核心必须了解每种语言、每种工作流、 每个团队的奇特步骤，它就会腐烂。所以它不这样做。 核心知道通用动词以及如何检测项目；真正的能力来自围绕它放置的插件， 任何人都可以添加插件而无需触及核心。

将能力推出到插件的另一个原因是我不想拥有 fledge 能做什么的列表。 人们可以随心所欲地扩展它：Rust、Swift、TS、shell。你用你熟悉的 任何语言写插件。你不必被迫进入我的语言来扩展我的工具。

插件与核心交互的方式是一个真实的、版本化的契约： 一个带 `plugin.toml` 清单的二进制文件，与 fledge 交换 JSON， 无论是原生的还是沙箱化的 WASM 模块，都是如此。 协议章节有线路格式和能力握手；在这里只需知道这个接缝是一个契约， 而非一堆约定。

还要排除一个常见问题：插件和 lane 不是同一件事。 插件添加能力，一个新动词。lane 将你已有的动词链接成有序的流水线。 所以不是"一切都是插件"。有一个内置了三大支柱的真实核心， 插件在其周围扩展能力，lane 将这些能力串联成序列。我在后面给 lane 单独一章。

任何语言，这意味着你无法信任它们

"用任何语言随心所欲地扩展"的反面是，你最终运行的代码 是你没有写过、无法为之担保的。插件只是别人的程序。 一旦你决定任何人都可以用任何东西写一个，你也就决定了 你将要运行大量不受信任的代码。

所以 fledge 用 WASM 沙箱化插件，基于 Wasmtime。目的是安全。 插件无法读取你整个磁盘或回电，除非你允许它。默认情况下 它被封箱：它获得你授予它的东西，仅此而已。

沙箱有第二个原因，而且是真正的原因。智能体运行这些插件。 如果智能体在安装和运行插件，你不能默认信任它们。 不是插件，也不能盲目地信任运行它们的决定。一个智能体 伸手抓取一个插件并执行它，正是"大概没问题"不够用的情况。 沙箱使智能体运行的工具安全。这是与我在智能体那本书中涉及的 信任和爆炸半径内容的桥梁；在这里，工具侧的版本很简单： 不受信任的代码加上自动运行它的东西，等于你需要一个围栏。 WASM/Wasmtime 就是那个围栏。

选择 WASM 而非显而易见的替代方案值得说明。你会首先想到的两个是 seccomp 配置文件（Linux 系统调用过滤）和容器（Docker 或类似的）。 两者都能工作，但两者都增加了不适合这里的摩擦或假设。 seccomp 需要操作系统级别的权限才能配置，并且只适用于 Linux，所以它立即打破了跨平台承诺。容器意味着 Docker 守护进程、 一个独立的镜像拉取，以及

比"运行一个插件"更重的进程边界。 WASM 的适配方式不同：Wasmtime 作为库直接嵌入到 fledge 进程中， 包含在单一二进制文件中，在任何操作系统上运行无需额外守护进程或权限， 并在链接时以结构化方式而非通过需要有人配置的内核策略来强制执行能力边界。 不是 seccomp 或容器有什么问题。而是一个作为单一二进制文件发布、 在任何地方运行的工具需要一个随之而来的沙箱，Wasmtime 能做到这一点。

金丝雀

一个你无法证明的沙箱只是希望。所以有一个金丝雀：一个插件， 其工作是尝试做沙箱化插件不应该能做的事情，并确认它做不到。 这是沙箱成立的证明。如果金丝雀无法突破，边界就是真实的； 如果它曾经能突破，你会立即知道。

实际上有两个插件。`fledge-plugin-canary` 是原生的那个。 它在无沙箱的情况下运行，证明攻击*有效*：读取 SSH 密钥和 AWS 凭证， 拉取像 `GITHUB_TOKEN` 这样的环境变量，打开网络连接， 派生进程，写入 `.git/hooks`。然后 `fledge-plugin-canary-wasm` 在 Wasmtime 沙箱内运行相同的攻击测试集，每一个都应该返回 BLOCKED。 其自述直言不讳："证明 Wasmtime 沙箱阻止了原生金丝雀暴露的每一次攻击。"

当你并排看到两者而非相信我的话时，意义更加清晰。 原生金丝雀自己的输出将自身与运行相同代码的 WASM 插件会看到的情况进行对比：

```
NATIVE: ~/.ssh/ READABLE             → WASM: BLOCKED (no preopened dir for ~)
NATIVE: ~/.config/fledge/ READABLE   → WASM: BLOCKED (outside sandbox)
NATIVE: GITHUB_TOKEN LEAKED          → WASM: BLOCKED (not passed to guest)
```

相同的攻击，两个边界。原生读取你的 SSH 密钥；WASM 做不到， 因为没有预先开放的目录让它通过去访问 ~。原生继承 `GITHUB_TOKEN`； WASM 做不到，因为客户端的环境是空的。在 WASM 侧返回 LEAKED 而非 BLOCKED 的任何结果都意味着沙箱逃脱。两者合在一起是端到端的检查： 一个证明危险是真实的，另一个证明围栏是有效的。

沙箱不能防御什么

沙箱是爆炸半径控制。它限制了插件代码可以访问的内容。 它不能限制一切，将其宣传为它无法解决的问题的解药， 会是本章的不诚实版本。所以这里是 WASM 买不到你的东西。

它不能阻止更高层的数据出口。WASM 盒子防止插件访问你的 SSH 密钥， 但它不管理你故意交给工具的数据。`fledge review` 会将你的 diff 和规格说明发送到你指向的任何 LLM 提供商，如果那是托管端点， 未合并的代码就离开了机器。没有 WASM 边界触及那个， 因为数据从前门离开，而非通过插件。这是一个同意和策略问题， 我在评审章节将其作为这样的问题处理，而非沙箱能解决的东西。

它不能保护以宿主用户身份运行的插件。不是每个插件都是 WASM。 原生插件以你的权限、你的环境、你的磁盘运行。 这就是为什么原生金丝雀能读取你的 SSH 密钥：它没有被沙箱化。 当你运行原生插件，或者一个从客户端下面派生进程的插件时， 你信任它的方式就是你信任在机器上运行的任何东西。 沙箱是 WASM 路径的保证，而非覆盖每个插件的总体保证。

它不验证来源或意图。WASM 限制代码能做什么。 它对代码，或写了插件运行的代码的智能体，是否应该这样做，没有任何说明。 一个执行智能体编写的逻辑的插件，仍然在盒子里执行我没有审查过的逻辑。 盒子里的不受信任代码仍然是不受信任的代码。盒子只是防止损害蔓延。

所以诚实的框架是狭窄的：WASM/Wasmtime 是代码执行路径的与来源无关的 爆炸半径控制。它不是数据流控制，不是同意管理，也不是停止思考 你交给插件什么或谁写了它的理由。

插件生态系统比你只通过 fledge 的三个原生内置了解它时预期的更大： CorvidLabs 组织中有许多 `fledge-plugin-*` 仓库，跨越多种语言 （生态系统章节对此有完整介绍），用任何适合工作的语言编写。 沙箱正是让这成为可能的东西：插件可以用任何东西编写，但仍然安全可运行。 我在自己的章节里做了生态系统和语言的完整巡礼。

fledge 如何融入日常

智能体在大量仓库中高速运行不受信任的插件，这正是为什么沙箱没有商量余地。做出这些决定的通常不是我在逐案判断，而是智能体，持续运转，跨越大量仓库。底层的 WASM 沙箱使其可以安全运行。

插件协议

这是那道接缝，插件与核心真正相遇的地方。它在仓库中有一个名字： 协议是版本化的，版本字符串是 `fledge-v1`。插件声明它，这就是握手。 其他一切都挂在它上面。

插件作者实际上写什么

一个插件是一个在根目录有名为 `plugin.toml` 清单的 git 仓库， 加上一个或多个可执行文件。清单很短。你为插件命名，给它一个版本， 声明你使用哪个协议，并列出你要添加的命令：

```
[plugin]
name = "fledge-deploy"
version = "0.1.0"
protocol = "fledge-v1"

[[commands]]
name = "deploy"
binary = "bin/fledge-deploy"

[capabilities]
exec = true
store = true
metadata = false
```

这就是作者侧的全部契约。每个 `[[commands]]` 条目是 fledge 将知道的 一个新动词，指向要运行的二进制文件。`[capabilities]` 块是作者预先声明 这个插件需要能做什么：运行 shell 命令、持久化一点状态、读取项目元数据。 默认是 `false`。你请求你需要的，仅此而已，而且这个请求在清单中是可见的， 而非隐藏在代码中。

宿主如何与插件通信

当你调用一个插件命令时，fledge 生成二进制文件，通过 stdin 和 stdout 以 JSON 行（每行一个 JSON 对象）与其通信。 宿主发送的第一条消息是一个 `init` 消息，而那条消息是零配置理念的字面化。 它把整个情况交给插件：项目名称和根目录、检测到的语言、 git 状态（分支、脏或干净、远程）、插件自身的版本和目录、 fledge 版本，以及授予的确切能力。插件不必去发现它在哪里 或它面对什么。宿主已经知道了，这是核心的全部工作，所以它直接告诉插件。

之后就是对话。插件发回消息：`prompt`、`confirm`、`select` 用于询问用户； `log`、`progress`、`output` 用于报告；`exec` 用于运行命令； `store` 和 `load` 用于它的一小块状态。任何带 `id` 的消息 都会收到恰好一个回复：一个 `response` 或一个 `cancel`。 stderr 从不被捕获，所以插件作者始终可以直接向终端打印调试信息。

读起来足够简单，可以在有人旁观的情况下理解。宿主以 `init` 开场， 交出情况：

```
{"type": "init", "protocol": "fledge-v1",
 "args": ["staging"],
 "project": {"name": "my-app", "root": "/Users/dev/my-app", "language": "rust",
   "git": {"branch": "main", "dirty": false, "remote": "origin"}},
 "capabilities": {"exec": true, "store": true, "metadata": false}}
```

插件，已经知道自己在哪里，请求 fledge 运行一个命令：

```
{"type": "exec", "id": "6", "command": "git tag -l 'v*' --sort=-v:refname",
"timeout": 10}
```

宿主回答匹配的 `id`：

```
{"type": "response", "id": "6", "value": {"code": 0, "stdout":
"v0.9.1\nv0.9.0\n", "stderr": ""}}
```

这就是全部形态：`init` 将智能体原本不得不猜测的一切 以结构化 JSON 而非 README 中的散文形式交付，每个带 `id` 的请求 都会收到恰好一个匹配的响应。工具告诉插件它在哪里；插件不用猜。

沙箱在哪里接入

以上描述的是原生插件，一个普通的可执行文件，通过管道交换 JSON。 问题，仓库对此直言不讳，是原生插件以你的身份运行，拥有完全访问权限。 能力块门控协议，但它不门控*进程*。一个什么都没请求的插件 仍然可以读取你的 SSH 密钥，因为它只是一个以你的用户身份运行的程序。 这就是金丝雀证明的泄漏：在协议层面声明能力只是安全剧场， 这是 WASM 迁移的全部原因。我在沙箱章节讲了那段经历； 这里的重点只是它在协议中改变了什么。

WASM 插件保留完全相同的 `fledge-v1` 协议，但改变了边界。 插件声明 `runtime = "wasm"` 并附带一个单一的 `.wasm` 二进制文件。 与 stdin 和 stdout 不同，相同的 JSON 消息通过宿主链接的三个宿主函数 （`recv`、`send`、`exit`）传递。相同的消息类型，相同的对话。 而能力不再是礼貌的请求。它们在链接时被强制执行： 如果你没有授予 `exec`，`exec` 导入根本不会被链接， 尝试调用它的插件根本无法实例化。没有可以愚弄的运行时检查， 因为它会调用的函数对它来说不存在。文件系统访问是 `none`、`project` 或 `plugin`， 作为预先开放的目录挂

载；网络是一个布尔值。 除此之外，运行时受燃料限制和内存上限，所以插件不能永远旋转或耗尽机器资源。

所以你在 `plugin.toml` 中看到的能力与代码实际上能访问的能力是同一件事， 从结构上来说。

它必须是结构性的原因是金丝雀教的教训：一个你只是*声明*的能力， 是一个你信任插件去尊重的能力，而你沙箱化插件的全部原因是你不信任它。 唯一成立的版本是你没有授予的能力是不可触达的，函数字面上不在那里可供调用。 当智能体是那个安装和运行这些东西的，这就是你想要的。

这里值得命名一个版本风险。插件声明的协议字符串是 `fledge-v1`。 当发生破坏性协议变更时，那个字符串会变成 `fledge-v2`。 一个仍然声明 `fledge-v1` 的插件对着期望 `fledge-v2` 的宿主， 会在实例化时大声失败，在任何代码运行之前。它不会悄悄地行为异常。 清单中的版本正是强制那种早期失败的东西：不匹配的插件不会到达 它能做任何事情的地步，所以操作者确切地知道什么需要更新， 而不是追踪一个微妙的运行时 bug。

Lane 生命周期钩子

同一个协议承载第二种集成：生命周期钩子。 插件可以在其 `plugin.toml` 中注册一个钩子，该钩子在 `lane:pre` 或 `lane:post` 事件上触发， 而非在直接命令调用上。部署插件是随文档一起发布的示例： 它注册一个 `lane:post` 钩子，在任何 fledge lane 完成后自动运行。

当钩子触发时，fledge 通过环境变量将上下文传递给钩子二进制文件： `FLEDGE_LANE_NAME`、`FLEDGE_LANE_STATUS` 和 `FLEDGE_LANE_RUN_ID`。 lane 定义对安装了哪些插件没有任何了解。插件对存在哪些 lane 没有任何了解。 生命周期事件是那道接缝。

这是完整的样子。lane 在 `fledge.toml` 中；钩子注册在插件的 `plugin.toml` 中； 两者无需知道对方的细节就能连接在一起：

```
# fledge.toml
[lanes.verify]
description = "Pre-merge gate: format, lint, test, build"
steps = ["fmt", "lint", "test", "build"]
```

```
# plugin.toml (from fledge-deploy, or any plugin registering a lane hook)
[plugin]
name = "fledge-deploy"
version = "0.2.0"
protocol = "fledge-v1"

[[commands]]
name = "deploy"
binary = "bin/fledge-deploy"

[[hooks]]
event = "lane:post"
binary = "bin/fledge-deploy-hook"

[capabilities]
exec = true
store = true
metadata = false
```

当 `fledge lanes run verify` 完成时，fledge 触发 `lane:post` 事件。 任何安装了匹配 `[[hooks]]` 条目的插件都会让其钩子二进制文件被调用， 并设置 `FLEDGE_LANE_NAME=verify`、`FLEDGE_LANE_STATUS=success`（或 `failure`） 和 `FLEDGE_LANE_RUN_ID`。

lane 将结构化上下文作为环境变量而非 stdout 上的散文发出， 所以插件读取的是可以采取行动的状态，而非刮取文本并猜测。 这与 `init` 消息的设计原则相同：宿主告诉你发生了什么；你不必推断它。

三大支柱架在这之上

三大支柱（任务运行、脚手架、AI 评审）是核心，而非社区插件。 协议是*围绕*它们的扩展层。脊柱是 fledge 的，而 `fledge-v1` 是任何人 在上面添加更多能力的方式。核心是真实的，插件围绕它； 协议只是两者相遇的接缝。

插件生态系统

CorvidLabs 组织中有数十个 `fledge-plugin-*` 仓库，跨越多种语言， 有少数已归档。活跃数量是真实的，但它是生态系统中最不有趣的事情。 插件数量是库存。论点在于哪些是我真正会递给你的，以及我对每个有多大把握。

所以这里是你每天会用到的五个：`augur`、`attest`、`format`、`gitleaks`、`deps`。 从这里开始。本章其余部分讲的是为什么其他三十七个存在， 以及对它们的信任程度。

你会伸手去拿的五个

这些基本上在每次变更时都会运行，不论是我的还是智能体的。它们归成两类工作。

信任对是 `augur` 和 `attest`。`augur` 仅从结构信号对 diff 的变更风险评分， 结果为"继续"、"审查"或"阻止"，无需 API 密钥，也无需 LLM， 其描述直接说出受众："为人类和智能体。"`attest` 记录谁审查了哪次提交的签名溯源， 保存在 git notes 中。这两个是我最依赖的，因为它们在智能体是作者时最重要。 进来时的风险评分，出去时的溯源追踪。

开发循环三件套是故意无聊的：`format`、`gitleaks`、`deps`。 运行格式化器。扫描提交的密钥。检查 Rust、Node 和 Python 的依赖健康状况： 过时、审计、许可证。它们没有一个令人兴奋。它们都在持续运行， 因为那正是你希望自动触发而非需要记住去做的检查。

如果你只安装那五个，你就拥有了我从生态系统中获得的大部分东西。

三个层级，以及我有多大把握

诚实阅读名单的方式是按信心，而非按数量。插件分为三个层级， 我对它们的信任程度非常不同。请在 CorvidLabs 组织中查看当前列表； 以下是如何解读你在那里找到的任何内容。

核心层。 `augur`、`attest`、`format`、`deps`、`gitleaks`。每日五件套。 我写了这些，我在每次变更时运行它们，它们被维护和测试。 这是我会将工作流押注于其上的层级。

集成层。 `github`、`discord`、`algochat`、`memory`。 这些将 fledge 接入外部事物：通过 `gh` 的 GitHub API、 用于 CI 失败通知的 Discord Webhook、加密的链上消息、一个记忆存储。它们也被维护和测试，但它们承担着它们所连接的事物的风险。 一个集成只有其背后的服务那么可靠。

试验场层。 `weather`、`roast`、`hangman` 以及其余的一次性作品。 `weather` 打印终端天气预报。`hangman` 用从你代码库中提取的标识符 玩猜词游戏。`roast` 将一次提交通过 LLM 并嘲讽它，"仅供娱乐目的"。 这些没有维护到相同的标准。它们存在是为了证明沙箱有效： 如果一

个随兴写出的半认真天气插件是安全的，那协议就在做它的工作。 低标准不是弱点。它是证明。

核心层和集成层被维护和测试。试验场层证明沙箱成立。 不要把平铺的数量读成同等经过审查的工具的均匀分布， 因为它们不是，假装如此会是本章的不诚实版本。

溯源，直说了

我写了其中大部分。每日五件套是我的。几乎所有的一次性作品也是， 随兴写成，因为一旦协议存在，发布一个插件就不费什么成本。 我需要某样东西，我写了一个插件，我放进去了。协议的意义在于 我不必扩大核心来添加能力。

我不打算声称拥有一个我其实没有的外部作者社区。 有几个插件来自这个圈子，但我不是将名单呈现为社区审查的广度。 这主要是一个人填满一个抽屉，这就是诚实的溯源。

金丝雀，以及为什么这一切是安全的

有一对值得单独提及，因为它是整个生态系统的承重结构： `fledge-plugin-canary` 和 `fledge-plugin-canary-wasm`， 来自沙箱章节的红队对。原生的证明攻击有效。WASM 的证明沙箱阻挡它们。 其他一切（一个用 Kotlin 写的插件，一个我半睡半醒时写的插件） 之所以可以安全运行，是因为那对维持着边界。生态系统可以是混乱的， 正是因为围绕它的盒子不是。

为什么有这么多语言

大多数是 Rust 和 shell，然后是 Swift、TypeScript、Kotlin、Python， 还有几个 HTML/JavaScript 的。这种分布是协议按预期工作的体现。 `fledge-v1` 和 WASM 沙箱的全部原因是插件可以用任何东西写出来 并且仍然安全运行，所以生态系统不必统一使用一种语言，只需统一一个契约。 `fledge-plugin-bridge` 是 Kotlin。`fledge-plugin-memory` 是 TypeScript。 `fledge-plugin-attest` 是 Swift。一半的集成是 shell。 没有人必须学习 Rust 才能为 fledge 添加内容。他们用手边的任何语言 写了那个东西，核心始终保持相同大小。

我不拥有 fledge 能做什么的列表。协议拥有，它让任何人无需询问我就能添加。 这种分布是证明，层级是关于它的诚实。

---

spec-sync 与保持诚实

规格漂移是我比大多数人更在乎的失败模式。你写了一份规格说明， 一个模块的契约（它做什么，它的公共 API 是什么），然后代码漂移了。 有人改变了一个函数，规格说明仍然声明旧的形态， 现在文档和代码悄悄地不一致了。在人类团队中，这是一份陈旧的 README。有智能体在循环中，情况更糟，因为智能体将规格说明当作真理读取， 在代码不再遵守的契约上构建，没有人注意到，直到某些东西崩溃。 spec-sync 的存在使这种漂移无法被忽视。

spec-sync 是其自身的工具，其自身的 Rust 二进制文件，其自身的 GitHub Action， 其标语说明了它是什么："具有跨项目引用、依赖图和 AI 驱动生成的双向规格-代码验证。" 重要的两个词是*双向*和*验证*。它在两个方向上检查规格说明和代码是否一致。 它不是测试套件，也不是针对散文的模糊 diff。 它是结构化的契约检查：文档中的公共 API 是否真的与真实的相匹配。 两个方向并不对称：代码有但规格说明未记录的导出是警告，是需要填入的东西；规格说明声明但代码没有的条目是错误，是违背的承诺。

它实际上检查什么

规格说明是一个 Markdown 文件，`*.spec.md`，带有 YAML 前置数据和一组必需章节。 前置数据命名模块、版本、状态以及它覆盖的源文件。 必需的 `##` 章节是"目的"、"公共 API"、"不变量"、"行为示例"、 "错误情形"、"依赖项"和"变更日志"。遗漏任何必需章节， 规格说明就是不完整的，验证会阻断。

这是规则；这是一份遵守它的规格说明。这是 fledge 自己的 `review` 模块规格说明的开头，真实的前置数据，以及 `## Public API` 章节，其中填入了 spec-sync 与代码核对的实际导出：

```
---
module: review
version: 10
status: active
files:
  - src/review.rs
db_tables: []
depends_on:
  - spec
  - llm
  - config
---

# Review

## Purpose

AI-powered code review of current branch changes...

## Public API

### Exported Functions

| Export | Description |
|--------|-------------|
| `run` | Entry point for the review command |
| `ReviewOptions` | Options struct: base, file, json, model, provider,
with_model |
| `ReviewFormat` | Enum: Summary, Checklist, or Inline |
```

`files:` 行是 spec-sync 与 diff 以及 `src/review.rs` 的真实导出进行交叉比对的内容。 `## Public API` 表中的每一行都是 spec-sync 期望在代码中找到的名称； 代码中没有的条目就是下面的错误情形。如果你从未写过规格说明， 那个前置数据加上一张诚实的 `## Public API` 表就是要复制的形态。 其余必需章节是同样类型的东西，描述模块承诺的散文和表格。

然后它双向验证：

- **代码 → 规格说明：** 未记录的导出，警告。
- **规格说明 → 代码：** 幻影或陈旧条目、缺失源文件、类型不匹配，均为错误。

它对 schema 也做同样的事：声明的数据库表和列与真实 SQL 核对， 幻影表或列会失败。 它还跨项目解析引用， 所以一个仓库中的规格说明可以通过 `owner/repo@module` 依赖另一个仓库中的模块， 该链接也会被验证。

要记住的是，这是*结构性的*。它不是评估你的散文，也不是生成测试。 它问一个直接的问题：文档中的接口是否与真实接口相匹配， 并确定性地回答它。这正是使它可以放在智能体前面的原因。 没有可以争论的判断调用；API 要么匹配规格说明，要么不匹配。

它如何出现

它以三种方式出现，在访谈中"spec-sync 在日常中如何出现"这个问题的答案是： 全部三种。三扇通往同一规格格式的门，每扇在不同地方运行：

入口	运行位置	做什么
`fledge spec`（原生）	你的机器，在 fledge 内	`init`、`check`、`list`、`show`：fledge 自己的验证，无需 shell 外调
`specsync` 二进制文件	智能体的循环，在 PR 之前	`specsync check` / `--fix`：智能体在循环中运行的独立工具
`CorvidLabs/spec-sync@v4`	CI，在拉取请求上	门控构建，注释漂移，失败时阻断

本节其余部分就是这三行，逐一介绍。

fledge 原生了解规格说明。 spec 是 fledge 的支柱之一。fledge 的 README 对此一字不差："`Spec | spec | [spec-sync]`。模块声明它们的契约， AI 将其用作上下文。"值得对代码中的含义精确说明：fledge 有其自身的 `fledge spec`（`init`、`check`、`list`、`show`）直接内置于二进制文件中。 它不通过 shell 外调 `specsync` 工具。它读取相同的 `.specsync/config.toml` 和相同的 `*.spec.md` 文件，自己解析它们，并进行自己的检查。 所以规格格式是共享的，但 fledge 内的验证是 fledge 自己的代码， 而非对独立二进制文件的外部调用。

原生的规格感知也是为什么 fledge 的 AI 命令可以依赖契约。 `fledge ask` 是规格感知的问答， `fledge review` 是规格感知的代码评审， 两者都将相关规格说明作为上下文引入。规格说明是这些命令在推理你的代码时 向模型提供的上下文。规格说明不是旁边的文档；它是工具在推理你的代码时读取的东西。

它门控 CI。 有一个独立的 GitHub Action，`CorvidLabs/spec-sync@v4`， 自动运行 `specsync check`。最小工作流：

```
- uses: CorvidLabs/spec-sync@v4
  with:
    strict: 'true'          # warnings become errors
    require-coverage: '100' # minimum spec coverage
```

`strict` 将警告变成错误。`require-coverage` 设置下限。`comment` 在拉取请求上发布规格漂移摘要。检查在失败时以非零退出， 这意味着 PR 被阻断。这是使整件事成为现实的机制： 漂

移不会生成礼貌的提示，它会使构建失败。智能体，或者人类， 无法悄悄地让规格说明和代码分道扬镳，因为分道扬镳*就是*那个失败的检查。

它让智能体保持符合规格。 这是我最在乎的，也是为什么 规格说明存在于智能体的循环内而非仅在 CI 中。机制是具体的。 智能体作为其循环的一部分运行 `specsync check`，就像运行 build 和 test 一样。 检查返回结构化的失败，智能体读取它们并决定怎么做， 而它走哪条路取决于漂移是哪个方向的。

这是当两个方向都发生漂移时结构化输出的样子（与真实输出格式匹配的说明性示例）：

```
specsync check

CHECKING src/review.rs against review.spec.md
  WARNING  undocumented export: `ReviewOptions::with_model`
             → run `specsync check --fix` to add stub

  ERROR    phantom export: `ReviewFormat::Detailed`
             spec claims this variant; code has: Summary, Checklist, Inline
             → update spec or add the variant to code

  ERROR    phantom export: `run_async`
             spec claims this function; not found in src/review.rs
             → update spec or add the function

SUMMARY  2 errors, 1 warning
EXIT 1
```

这就是智能体读取的内容。错误命名了文件、规格声明，以及代码实际拥有的。 警告命名了导出和确切的修复命令。不需要猜测：要么调和代码以匹配规格说明， 要么更正规格说明，然后重新运行，直到退出 0。

如果代码新增了规格说明未提及的导出，警告方向，智能体不手动编辑规格说明。 它运行 `specsync check --fix`，自动将未记录的导出作为存根添加到规格说明中， 简单的情形就一步搞定。智能体的工作主要是用真实描述填充存根， 而非首先发现漂移。

另一个方向是需要实际工作的那个。当规格说明声称代码不遵守的 API 时， 错误方向，陈旧或幻影条目，没有 `--fix` 命令，因为修复是一个判断调用： 要么代码是错的，智能体去让代码匹配承诺的，要么规格说明是有理想色彩的， 智能体更正契约。无论哪种方式，智能体都必须在代码中有意地调和它， 并重新运行检查直到它变绿。这就是实践中的循环：契约、变更、检查， 然后要么是自动添加，要么是真正的更正，取决于哪一侧偏离了步伐， 这在 diff 到达拉取请求之前运行，在 Merlin 的循环内，而非在 CI 事后。

spec-sync 不检查的内容

有一条我想诚实说明的界线，因为它是这个工具能力的边缘。spec-sync 检查规格说明和代码是否一致。它对规格说明本身是否足够好没有任何意见。

想想这留下了什么空白。一份规格说明可以命名正确的导出，但对它们的描述是错的。它可以很单薄，一个什么都没说的"目的"部分，加上一张准确却无法说明模块用途的"公共 API"表。它可以是充满愿景的，是为某人本想写的代码而写的。如果某个智能体从一份本身就有误的任务摘要草拟了规格说明，或者摘要遭到了注入，规格说明就可能成为对错误事物的忠实合约。在上述每一种情况下，spec-sync 都会通过。表面与表面相符。检查是绿色的。合约是错的，而循环中没有任何东西捕捉到这一点，因为将规格说明与代码比对，无法告诉你规格说明从一开始就是有问题的。

这是一个真实的缺口，我把它点名出来。代码有一个检查工具；规格说明没有。你真正想要的，是在智能体开始基于规格说明构建之前，有第二道门对规格说明本身运行检查：每个必需章节是否真的说了什么，"公共 API"是否指向实际存在的文件，是否有一个清晰的关于成功和失败是什么样子的说明，以及是否有人类真正签署了这份合约。这就是 `fledge spec lint`，它还没有被构建出来。在此之前，规格说明是这整条流水线中唯一没有被任何东西验证的输入，当你信任那个绿色通过时，这一点值得铭记。

为什么共享契约是关键所在

这直接回到这些工具下面的论点：人类和智能体使用相同的工具， 作为平等的一等公民。规格说明是我拥有的最清晰的那个理念版本。 人类写或审查它；智能体基于它构建；spec-sync 在两个方向确定性地约束双方。 重构而不更新文档的人类被抓住的方式与幻构了一个 API 的智能体被抓住的方式相同。 检查不关心是哪一个漂移了。

用 Rust 构建 fledge

fledge 是一个单一二进制文件。你安装一次，它在任何机器、任何操作系统上运行， 无需在其旁边安装任何其他东西。没有要引入的运行时，没有解释器， 没有必须预先存在的依赖管理器。默认跨平台：一条构建流水线生成一个 在 macOS、Linux 和 Windows 上都能工作的二进制文件。这是核心要求， 而 Rust 是使其简单而非一场苦战的原因。整章从此而来：为什么 Rust 是正确的选择，为什么掌握它没有人们预期的那么艰难， 以及智能体如何弥补了流利度的差距。

我开始做 fledge 时已经写了大约十年 Swift。所以这章的诚实版本 有点平淡无奇：掌握 Rust 并不痛苦。没有什么重要的东西和我打架。 连借用检查器也没有，那是每个人警告你的部分。

我觉得人们期待这里有一段战争故事：那门让我折服的语言， 那段与编译器打败仗的月份。我没有那些。我宁愿告诉你为什么它进展顺利， 而非编造戏剧性。

Swift 为我做了准备

很多 Rust 感觉很熟悉，因为 Swift 已经用同样的思路训练了我， 只是穿着不同的衣服。

枚举和模式匹配是最大的一点。Swift 的枚举是真正的和类型， 我多年来一直依赖它们和 `switch`。Rust 的 `enum` 和 `match` 是同样的肌肉。 Result 和 Option 也不陌生。我在 Swift 的可选值和 `Result` 中生活了很长时间， 所以"这可以是一个值或什么都没有"和"这可以是一个值或一个错误" 已经是我思考代码的方式。Rust 只是让你一直、明确地处理两者。 这对我来说不是一个需要获得的新纪律；它是一个我已有的纪律，现在被强制执行。

Trait 落地时大致像 Swift 的协议。不完全相同，但足够接近， 我不是在学习一个陌生的概念。我是在学习我已知概念的一种方言。 "定义行为，让类型遵守它"在两者中是同样的形态。

所以这门语言感觉不像一堵墙。感觉像一个我半住过的地方。 Swift 灌输的值类型和可选值习惯，我认为，正是为什么借用检查器从未成为一场苦战。 如果你已经以所有权和"谁持有这个值"的方式思考，检查器大多在告诉你 你本来就在努力做的事情。这不是我必须获得的新思维方式。 它是一个更严格的裁判，针对一个我已经知道如何玩的游戏。

而它构建出来的东西的形态是我想要的简单形态：fledge 是一个单一的 Rust 二进制文件，用 Cargo 构建。一个 crate，另一端出来一个二进制文件。

智能体做了重活

我不打算假装自己是一个流利的 Rust 程序员。我不是。我是一个流利的 Swift 程序员， 能够很好地阅读和引导 Rust，这是两件不同的事情。

弥合差距的是智能体。我重度依赖它们，在一门我不那么流利的语言中保持高效。 熟悉的部分我可以自己阅读、推理和引导。我知道代码*应该*做什么， 以及好的结构是什么样子，因为那部分在语言之间是通用的。 Rust 有自己的惯用语、表达事物的自己方式的部分，由智能体承担。

这是一个值得从另一面观察的区别。我的 Swift 是手写的。 我的 Rust 是智能体放大的。在一个领域里我是在键盘上的作者。 在另一个领域我是知道什么是正确的那个，将智能体指向它并检查它们的工作。

说实话，以这种方式用 Rust 构建 fledge，是背后整个论点的一个小证明。 这个工具为人类和智能体双方而构建。它由一个人类和智能体共同*构建*。 今天驱动 fledge 的智能体也帮助了 fledge 最初的编写。

工具链让我如释重负

这是我没想到会享受这么多的部分：Rust 的工具链感觉让我如释重负。

Cargo 就是能用。一个工具用于构建、测试、依赖管理，整件事， 并且在任何地方都一样。crate 生态系统很深厚。 无论我需要什么，通常都有一个可靠的 crate，引入它只需一行。而单一二进制构建正是 fledge 想要成为的：我需要发布一个能在任何地方运行的 轻量二进制文件，而 Rust 默认给你这个。

对比之下是一旦你离开 Apple 平台后 Swift 工具链的情况。在 Apple 上， Swift 的故事很棒。离开它（Linux、跨平台、那种 fledge 需要的"作为单一二进制 在任何地方运行"目标）就变得更难了。这是 fledge 用 Rust 而非 Swift 的真正原因， 尽管 Swift 是我的母语。我想要一个可以放在任何机器上的轻量单一二进制文件， 智能体可以在任何地方运行，而 Rust 的工具链使其成为简单的路径而非一场苦战。

所以这就是构建者视角的真相。Swift 准备了思维，智能体弥补了我没有的流利度， 而工具链（Cargo、crate、单一二进制文件）是真正让我庆幸切换的部分。 选择语言是容易的决定。真正需要思考的工作是其他一切： 弄清楚 fledge 应该是什么。

我如何使用它，以及谁在使用它

让我先说诚实的部分，而不是留到最后。fledge 没有被广泛使用。 它是我的，是我的智能体的，是我的圈子的。这就是重点。 这个工具首先是为了解决构建者的实际问题而构建的，它做到了，每一天。 解决你面前问题的工具，胜过有一面标志墙但没有实际使用的工具。

所以这章回答的问题不是"有多少人使用 fledge"。而是 "谁在使用，以及为什么这是正确的顺序"。答案不是"所有人"， 它从来就不应该是。

我的智能体比我更多地驱动它

当人们想象我如何使用 fledge 时，常常会想象*我*在使用它。这种情况确实发生， 但已不再是 fledge 的主要运行方式。我的智能体驱动它，差距悬殊。 当一个智能体在处理一个仓库时，fledge 是它构建、测试和运行它刚刚更改内容的方式： 是它的执行接口，不只是我的。这正是第一章所说的设计目标。 大多数日子里，智能体从中获得的里程比我更多，而我主要是在引导。

这就是我为之构建的用户群。不是一群陌生人。我加上智能体， 在我拥有的每个仓库中，随时随地。

还有谁在使用它

fledge 是开源的，在 Homebrew tap 上，任何人都可以 `cargo install` 它。 这些都不是采用声明。一个 tap 是分发渠道，不是用户数量， 我不打算将一个包装成另一个。

今天的用户群是我、我的智能体和 CorvidLabs 圈子， 也就是我实际合作的人。没有广泛的外部采用。没有陌生人社区来提交 issue。 这是个人和圈子的基础设施，这值得直说，而非暗示一个并不存在的浪潮。

协作者对于这幅图景很重要，不只是作为数量。fledge 作为整个圈子的共享接口 是设计的一部分。当 CorvidLabs 中的某人拿起我的一个仓库时， 他们不必学习那个仓库的私有方言。他们已经知道动词， 因为动词在任何地方都是一样的。为我省去重新学习的一致性， 也让我的合作者落在一个他们已经知道的接口上。

你该不该使用它

也许还不该。说实话，只有在你像我一样工作时：大量仓库，循环中有智能体， 覆盖所有这些的一个一致接口。如果那是你的问题，fledge 能解决它。 如果不是，这个工具是过度设计的，我宁愿这么告诉你，也不要向你推销它。

我不会声称有限的采用证明了设计在规模上有效。它没有。 它证明设计对我有效，这是一个更小的声明，也是我唯一能支撑的声明。 fledge 首先在我自己的世界里每天证明自己的价值，主要通过智能体， 有我引导，有一个小圈子在同一接口上。这是一个工具在做它的工作。 广度可以以后再来，也可以永远不来。工作已经在完成了。

---

fledge 的未来

"fledge 下一步去哪里"的诚实答案比人们预期的更不令人兴奋， 我认为这是一个好兆头。它大体上已经成熟了。其大形态已经构建出来。 剩下的主要是维护它，并继续以它为自己的工具。有几个我会发展它的方向， 但我没有坐拥一个宏大的重新发明路线图，因为 fledge 不需要被重新发明。 它需要继续成为其他一切所依托的那个东西。

更大的生态系统

试验场层和集成层是开放的：任何人都可以在不询问的情况下添加它们。 核心层不是。这种不对称是有意的，不会改变。

空间最大的方向是生态系统章节已经描述的那个：更多插件。 fledge 通过核心周围的环扩大而变得更有能力，而非核心变胖： 更丰富的插件集，这样无论你把 fledge 放进哪个仓库，已经有一些东西知道如何处理它。更多集成，更多每个解决一个真实问题的一次性作品。

与此相关的是扩大 fledge 开箱即检测和运行的内容， 来自第一章的零配置承诺，在更多地方实现。 每一种它尚不了解的语言和平台，都是"同样的动词直接能用"承诺有缺口的仓库， 所以填补检测和平台覆盖是另一个显而易见的方向。不是多么炫目的事情。 覆盖只是让通用接口真正通用的东西。

现在最重要的覆盖不是下一种语言，而是操作系统。fledge 必须在三个平台上 都作为一等公民运行：Windows、Linux 和 macOS。只在一个操作系统上真正如鱼得水的工具 不是通用接口，而是本地接口。所以跨操作系统支持是下一个真正的前沿， 比添加另一个语言检测器更重要。

维护智能体依赖的东西意味着什么

我可以在这里列举功能并承诺重新发明。那会是错误的框架，不符合 fledge 现在实际所是。

2026 年真正重要的问题不是 fledge 添加了什么功能。代码很廉价。 智能体写大部分代码。稀缺的是一个你真正能信任来运行你工作的工具。 智能体能够比我审查的更快生成新功能。它无法生成的是一个身后有多年 犬养测试、有稳定接口、有流水线可以依赖的已知行为的工具。

一个经过多年犬养测试、拥有流水线可以信赖的稳定接口的工具，才是 fledge 是什么，这是更难构建的东西。不是技术上更难。 是因为它需要决定稳定性是输出，而非副产品，所以更难。每个工具 都以新功能的形态开始生命。很少有工具能存活足够长以成为基础设施。 那些能做到的，是那些维护者将"没有任何东西崩溃"视为主要成就， 而非缓慢冲刺的安慰奖的工具。

我在每个仓库、一直使用 fledge。我的智能体持续驱动它。 这意味着 bug 会找到我。缺失的动词，检测猜错了， 插件钩子以错误的顺序触发。我碰到它，因为我生活在它上面， 而我修复它，因为我生活在它上面。犬养测试不是一项附属活动。它是整个质量机制。 维护和使用是同一份工作。

本书的前言问道，一个工具值得被依赖的关键是什么。 它指向的答案是：一个干净的接口，没有隐藏的交互步骤， 一个真实的插件边界，一份随代码变化而保持诚实的规格说明。 做到这些，智能体从第一天起就能驱动它，因为它从来就不是专门为需要人类介入而设计的。

fledge 就是那样。不是因为任何单一的设计决定，而是因为我每次拒绝让捷径 侵蚀接口所积累的结果。动词在每个仓库中都是一样的。JSON 是同样的形态。 `fledge introspect` 清单保持最新。没有任何东西在我没有先在自己的工作上运行过 的情况下就发布出去。

当智能体是操作大多数工具的那方时，赢得信任的不是长长的功能列表。 而是以同样的方式出现，每一次，足够长，以至于真实的东西被建立在上面。 这是比路线图幻灯片更安静的抱负，也是唯一重要的那个。

所以从这里开始的工作就是它一直以来的工作：保持它的稳固， 继续生活在它上面，保持接口诚实。我会持续遇到 bug，因为我持续运行它， 我会持续修复它们。这就是全部计划。

关于作者

0xLeif（leif.algo）在开放环境中构建。十年的小型、可组合 Swift 库，如 AppState、Cache 和 Fork。CorvidLabs 实验室。一堆大多起源于"我希望这个东西存在"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，并与真实代码进行了核对。

github.com/0xLeif · leif.algo

致谢

感谢 CorvidLabs，感谢那个让这些想法得以被检验和辩论成形的空间。

感谢开源维护者，整个技术栈都建立在他们的工具之上。没有任何东西是独自构建的。

以及感谢早期读者和随意付费的支持者，正是你们让"免费在线"成为我可以持续做的事情。

版权页

从 Markdown 排版，用 bookgen 构建，一个纯 Rust 流水线（无 Python）。

基于访谈、AI 辅助创作；经手工编辑和事实核查。写作时不使用破折号。封面和章节插图来自 Algorand 上的 Corvid and Nature 系列。