"without.boats" Async Rust 系列深度解析与展望

I. 引言

A. "without.boats" Async Rust 系列的重要性

"without.boats" 博客，由 Saoirse Shipwreckt 撰写，已成为 Rust 异步编程领域一个极具影响力的声音。该作者作为 Rust 异步功能开发的关键贡献者之一 [1]，其博文不仅提供了技术细节的阐释，更深入探讨了设计原理、历史背景、当前争议以及 Rust 异步编程的未来愿景。本报告分析的系列文章（部分由 [1] 所代表）是理解 Rust 最强大也最具争议特性之一——异步编程——其细微之处和复杂性的关键资源。这些文章共同构成了一幅关于 Rust 异步生态系统演进的详尽图景。

这些博文超越了单纯的技术文档，它们是作者积极参与 Rust 语言设计过程的一种体现，尽管是以公开的、个人化的视角。作者将博客作为平台，影响正在进行的讨论，并倡导特定的技术方向，例如在 AsyncIterator 设计上坚持 "poll next" [2]，以及提出详细的未来发展路线图 [3]。这种参与方式表明，这些文章的受众不仅包括普通的 Rust 用户，也包括 Rust 核心开发者和决策者。

B. 作者的声音及其对 Async Rust 的贡献

作者直接参与了 async/await 语法及 Pin API 的设计工作 [1]，这为其观点赋予了显著的权威性。其坦诚且时常带有批判性的写作风格，挑战了传统思维，并积极推动异步生态系统的改进。例如，作者在某些文章中对 Rust 项目内部沟通和决策过程表达了不满 [3]，并在技术辩论中（如 AsyncIterator 的设计）展现出坚定且论证充分的立场 [2]。

贯穿这些文章的一个核心议题，是在 Rust 的 foundational goals（如零成本抽象、内存安全、C ABI 兼容性）与实现人体工程学且功能强大的异步编程之间的内在张力。作者的论述常常深入探讨了在这种张力下所做的权衡与取舍。

C. 本分析报告的范围与目标

本报告旨在综合并分析 "without.boats" 博客中关于异步 Rust 的一系列重要文章的核心论点和技术细节，这些文章的内容主要通过所提供的研究摘要得以呈现。本报告的目标是提供一个结构化的、专家级的解读，阐明复杂的技术主题，识别贯穿始终的核心思想，并进行批判性评论。这份报告可视为对原始（假设已翻译）文章的分析性伴侣，帮助读者更深入地理解 Rust 异步编程的演进、挑战与未来。

II. 基本原则：Async Rust 的“为何”与“如何”

A. 解构“为何选择 Async Rust？”[1] 与“让 Future 保持其本质”[4]

历史背景：从绿色线程到无栈协程

Rust 的异步编程并非一蹴而就，其最初曾尝试过绿色线程（green threads）模型。然而，由于一系列难以克服的挑战，该模型最终被放弃。主要问题包括：

1. 栈管理：绿色线程的栈大小管理是个难题。最初采用的“分段栈”（segmented stacks）因在热循环中可能频繁分配和释放栈段导致性能波动而被废弃。后续考虑的“栈复制”（stack copying）方案，虽然能实现栈的动态增长，但在 Rust 这种没有垃圾回收（GC）且允许指针指向栈外部的语言中，跟踪和更新所有指向栈的指针变得不可行 [1]。最终，Rust 的绿色线程不得不采用类似操作系统线程的大栈，从而失去了其主要优势之一。
2. **FFI** 成本：绿色线程与使用C ABI的外部库交互时，线程栈切换的成本过高。Go 语言接受了这种成本，但 C# 则因此中止了绿色线程的实验 [1]。对于致力于嵌入式场景和与C代码高效互操作的 Rust 而言，这是一个严重缺陷。
3. **“两种 Rust”**模型的失败：为了解决上述问题，Rust 曾尝试让绿色线程运行时可选，允许代码在原生线程（使用阻塞IO）或绿色线程（使用非阻塞IO）上编译。然而，这种试图让所有代码兼容两种模型的做法未能良好运作，导致抽象并非“零成本”，并限制了 API 的设计 [1]。

这些因素共同导致 Rust 在 2014 年末移除了绿色线程，为后续基于 Future 和 async/await 的无栈协程模型铺平了道路。这一决策深刻地影响了 Rust 异步编程的形态，使其与 Go 等语言区分开来。

迭代器 **(Iterator)** 的类比

Rust 中 Future 的设计深受其成功的外部迭代器（external iterators）模型的影响。在 2013 年，Rust 从内部迭代器（基于回调）转向外部迭代器，后者将迭代状态封装在一个结构体中，通过 next 方法驱动 [1]。这种设计与 Rust 的所有权和借用系统完美结合，并通过单态化（monomorphization）实现了高效优化。

当面临设计异步抽象的挑战时，Aaron Turon 和 Alex Crichton 最初尝试了其他语言中常见的基于“续体传递风格”（Continuation Passing Style, CPS）的 Future。这种 Future 接受一个回调（续体），并在完成时调用它。然而，他们发现 CPS 风格在 Rust 中会导致过多的堆分配，尤其是在实现如 join（并发执行多个 Future）这样的组合子时，因为续体需要在多个子 Future 间共享所有权 [1]。

借鉴 C 程序员手动构建状态机处理非阻塞 IO 的方式，并受到外部迭代器模型的启发，他们转向了一种“基于就绪状态”（readiness-based）的方法。Future Trait 定义了一个 poll 方法，由外部执行器调用。这种控制反转避免了存储回调，使得 Future 可以被编译成单一的状态机，类似于迭代器组合子被编译成单一迭代器状态机一样 [1]。这种从续体到轮询的转变，与迭代器的转变如出一辙，再次证明了 Rust 处理带有生命周期的结构体和无栈协程的能力。

“Future 不是廉价线程”之辩 **(Eriksen vs. Nystrom)**

在探讨异步模型的哲学基础时，"without.boats" 的作者明确支持 Marius Eriksen 在其文章 *"Futures aren't ersatz threads"* 中的观点，而非 Bob Nystrom 在 *"What color is your function?"* 中提出的担忧 [4]。

Eriksen 认为，Future 提供了一种与线程根本不同的并发模型。其优势包括：

1. 类型区分：执行异步操作的函数（返回 Future）与"纯"函数在类型上有所区别，这有助于理解函数是否执行 IO。
2. 组合性：Future 作为工作的直接表示，可以按顺序或并发组合。阻塞函数调用若不启动新线程，则只能顺序组合。
3. 表达力：Future 的并发组合能力使得并发代码能更直接地表达逻辑，可以将特定并发模式抽象出来 [4]。

Nystrom 则将异步函数（RED 函数）和同步函数（BLUE 函数）的区分（即"函数着色问题"）视为一大痛点，因为 RED 函数通常只能被其他 RED 函数调用，且调用语法可能更繁琐。尽管 async/await 语法在一定程度上缓解了调用 RED 函数的痛苦，但 Nystrom 认为世界依然被分裂为两部分 [4]。

作者采纳 Eriksen 的立场，珍视 Future 提供的额外"赋能"（affordances），认为这些赋能是 Future 模型的关键优势，而非仅仅是调用函数的一个障碍 [4]。这一哲学立场是理解 "without.boats" 系列中许多后续论证和设计偏好的基础。它解释了为何作者积极倡导那些能够充分发挥 Rust 异步模型独特能力（如具有 poll_next 的 AsyncIterator、精细的内部任务并发原语）的特性，并对可能稀释这些独特性的想法（如 maybe(async) [4]）持批判态度。

B. Rust 的无栈协程模型与轮询机制

Rust 的异步实现核心是无栈协程。一个 async fn 或 async 块会被编译器转换成一个状态机，这个状态机实现了 Future Trait [1]。

- **Future Trait**: 该 Trait 的核心是 poll 方法，其签名为 fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output;。poll 方法返回一个 Poll 枚举，它有两个变体：Poll::Ready(value) 表示 Future 已完成并产生一个值 value；Poll::Pending 表示 Future 尚未准备好，需要稍后再次轮询。Context 参数则用于传递唤醒机制（Waker），当 Future 准备好继续执行时，它会通过 Waker 通知执行器 [1]。
- 执行器的角色: Future 本身不做任何事情，它们是被动的。需要一个异步运行时（executor）来驱动它们。执行器负责调用 Future 的 poll 方法。如果返回 Poll::Pending，执行器会暂停该 Future，转而执行其他任务。当被暂停的 Future 通过 Waker 发出通知后，执行器会再次将其加入轮询队列 [4]。一个"任务"（task）通常指一个已被调度到执行器上执行的顶层 Future。
- **async/await** 语法: async/await 是建立在 Future Trait 之上的语法糖。async 关键字用于声明一个函数或块返回一个实现了 Future 的匿名类型。在 async 上下文中，.

await 操作符可以用于等待另一个 Future 完成。当遇到 .await 时，如果其后的 Future 返回 Poll::Pending，当前的 async fn 会暂停执行并让出控制权，其状态被保存在编译器生成的状态机中。这使得开发者可以用看似同步的、命令式的代码风格编写异步逻辑，而编译器负责处理复杂的状态转换和暂停点管理[1]。这极大地改善了以往手动链接 Future 组合子所带来的复杂性和糟糕的用户体验。

Rust 的这种设计，即编译器将高级的声明式构造（如 async fn）编译为优化的、单一对象的状态机，是一种强大的、反复出现的模式。它不仅应用于 Future，也应用于 Iterator 和 Entry API[1]。这体现了 Rust 在提供零成本抽象方面的一种核心设计哲学，特别适合资源受限的环境和性能关键型应用。

C. “函数着色问题”与 "without.boats" 的视角

Nystrom 提出的“函数着色问题”[4] 是异步编程中一个广为人知的痛点。它指的是异步函数（RED 函数）和同步函数（BLUE 函数）之间的不兼容性：通常情况下，异步函数不能被同步函数直接调用（需要通过特定的运行时或适配器，如 block_on），并且异步函数往往会“传染”其调用者，迫使它们也变成异步的。

然而，"without.boats" 的作者在此基础上提出了一个更微妙、特定于 Rust 的问题，称之为“函数不着色问题”（The function *non*-coloring problem）[4]。这里，作者引入了第三种颜色——GREEN 函数，代表那些会阻塞当前线程的同步函数。

GREEN 函数的问题在于：

1. 难以区分：从函数签名和调用方式上看，GREEN 函数与普通的 BLUE 函数（不阻塞的纯计算或快速操作）完全一样。开发者可能无从判断一个函数是否会阻塞。
2. 在异步上下文中的危险性：在 RED 函数（即 async 函数）内部调用 GREEN 函数（阻塞函数）可能会导致严重问题。尤其是在单线程执行器上，或者当一个任务在持有锁（如标准库的 Mutex）的情况下调用阻塞操作然后 .await 时，很容易造成死锁或性能急剧下降。因为标准库的 Mutex 是不可重入的，如果一个任务持有锁并阻塞，同一线程上的其他任务尝试获取该锁就会死锁[4]。

这个问题在 Rust 中尤为突出，因为 Rust 旨在与现有的阻塞代码生态系统无缝集成，并且语言本身不通过类型系统来区分阻塞调用和非阻塞调用。作者曾提议通过属性标记阻塞函数以进行 lint 检查，但未被采纳[4]。这种“不着色”的阻塞函数给异步 Rust 用户带来了隐蔽的风险。

Rust 对零成本 FFI 和避免垃圾回收的承诺[1]，直接导致了其放弃绿色线程并采纳无栈协程。这反过来又催生了 Future 的轮询模型和 async/await 语法，进而引入了“函数着色问题”[4]。而“函数不着色问题”（即在异步代码中误用阻塞调用）则是进一步的后果，因为 Rust 力求同时支持同步和异步两种范式。这揭示了一条深刻的因果链：基础的系统级目标决定了高级并发抽象的形态及其伴随的挑战。这些“问题”并非孤立的设计缺陷，而是源于 Rust 作

为一门系统语言的定位所固有的权衡。

III. 核心抽象的运用与固有挑战

A. Pin 抽象：目标、问题与改进建议 [1]

Pin 是 Rust 异步编程中一个核心但 notoriously 复杂的概念。它的引入是为了解决自引用结构体（self-referential structs）的内存安全问题，这对于 async fn 至关重要。在 async fn 中，局部变量等状态可能需要在 .await 点之间被引用。编译器会将这样的 async fn 转换成一个包含这些状态和引用的状态机（即 Future）。如果这个 Future 在轮询开始后被移动到内存中的其他位置，那么其内部的自引用指针就会失效，导致未定义行为。Pin<P> 通过类型系统确保一旦一个值被“钉住”（pinned），它在内存中的位置就不会再改变（除非该类型是 Unpin 的）[1]。

尽管 Pin 的设计初衷是供底层库作者使用，以构建安全的异步原语，但它却时常“泄露”到高层应用代码中，给用户带来困惑和挫败感 [5]。主要体现在以下几个方面：

1. 在循环中选择 **Future**: 当在循环中使用 select! 宏等待多个 Future 时，如果这些 Future 在每次循环迭代中被重新创建，并且它们不是 Unpin 的，那么通常需要将它们 Pin 住。这是因为 select! 需要通过引用来轮询这些 Future，如果它们可能被移动，则其内部的自引用（若有）会失效。当前的变通方法通常是将这些异步函数调用提升到循环外部，使用 Box::pin 将其固定在堆上，并可能使用 fuse() 来确保它们在完成后不会被再次轮询 [5]。作者提出的更理想的解决方案是引入一个新的基于 AsyncIterator 的 merge! 宏，它可以轮询所有迭代器并产生项，而无需取消和重建 Future，从而避免手动固定。
2. 调用 **Stream::next**: 当用户尝试在一个未实现 Unpin Trait 的 Stream（AsyncIterator 的前身）上调用 next 方法时，常常会遇到与 Pin 相关的问题。这是因为 Stream::next 方法通常以 &mut self 接收 self，而底层的 poll_next 方法则要求 Pin<&mut Self>。一个固定的引用（Pin<&mut Self>）只有在类型实现 Unpin 时才能安全地转换为可变引用（&mut Self）。目前的变通方法是显式地将 Stream 固定住（例如，通过 Box::pin），因为一个被固定引用的流（Pin<&mut S>）通常会同时实现 Unpin 和 Stream。作者认为，即将到来的 AsyncIterator 应该通过引入 for await 循环语法来解决这个问题，该语法会在内部处理流的固定，减少用户显式调用 next 并处理 Pin 的需要 [5]。
3. 等待间接 **Future**: 当等待通过指针（如 Box<dyn Future> trait 对象，或递归异步函数调用中为避免无限大状态而使用的 Box）访问的 Future 时，也会遇到 Pin 的问题。作者特别指出了 Pin 设计中与 Box<T> 相关的一个“错误”：即决定让 Box<T> 始终实现 Unpin，即使其拥有的类型 T 没有实现 Unpin。这个看似武断的决定意味着 Box<T: Future> 不能直接实现 Future Trait。因为如果它实现了，人们就可以通过 Box 来轮询 Future，然后（由于 Box 是 Unpin 的）将 Future 从 Box 中移出，这会导致健全性漏洞。因此，只有 Pin<Box<T: Future>> 才实现 Future，迫使用户必须固定盒装的 Future 才能 .await 它们。作者认为，如果当初没有为 Box 实现 Unpin（当 T 不是 Unpin 时），

那么 Box<T: Future> 本可以直接实现 Future，从而简化等待盒装 Future 的过程。这被作者视为异步/等待设计中最大的失误之一，并且很可能是无法逆转的 [5]。

作者总结道，除了盒装 Future 的问题外，大多数 Pin 在高层代码中出现的情况，都可以通过对 AsyncIterator 更好的支持（特别是 merge! 宏和 for await 循环）来解决 [5]。这表明，对核心异步迭代抽象的改进，对于提升整个异步生态的用户体验至关重要。

下表总结了 Pin 的主要问题及其建议的解决方案：

表 **1**：**Pin** 相关问题及其建议解决方案总结

问题领域	具体表现	当前变通方法	建议解决方案	相关文献
循环中选择 Future	select! 在循环中重构 Future 时，若 Future 非 Unpin，则需要固定	将 Future 提升出循环，使用 Box::pin 固定，并可能 fuse	基于 AsyncIterator 的 merge! 宏，避免重构和取消 Future	[5]
调用 Stream::next	在非 Unpin 的 Stream 上调用 next 时，因 poll_next 需 Pin 而遇到问题	显式固定 Stream (如 Box::pin(stream))	AsyncIterator 的 for await 循环语法，内部处理固定	[5]
等待间接 Future	如 Box<dyn Future>。由于 Box<T> 总是 Unpin，Box<T: Future> 不实现 Future	必须使用 Pin<Box<T: Future>> 来 .await	(设计失误，被认为是难以逆转的)。理想情况下 Box<T:!Unpin> 不应 Unpin。	[5]

B. FuturesUnordered: 管理动态并发及其陷阱 [6]

在 Rust 异步编程中，管理一组动态数量的并发操作是一个常见的需求。FuturesUnordered 是 futures 库提供的一个关键工具，用于处理这类场景。要理解 FuturesUnordered 的角色，首先需要区分两种并发模式：

- 多任务并发 **(Multi-task concurrency)**: 这种模式下，每个并发操作都作为一个独立的任务（task）被派生（spawn）到执行器上。例如，使用 tokio::spawn 或 JoinSet。任务之间通常通过同步原语（如通道、互斥锁）进行通信。这种模式的一个主要限制是，被派生的任务通常必须是 'static 的，这意味着它们不能借用其父任务的非静态数据，这往往需要使用 Arc 和 Mutex 等共享所有权机制，增加了复杂性 [4]。

- 任务内并发 **(Intra-task concurrency)**: 在这种模式下，多个异步操作在同一个任务内部并发执行。这通常通过 join!（等待所有 Future 完成）或 select!（等待第一个 Future 完成）等宏来实现。这些 Future 可以直接借用其所在任务的外部状态，因为它们的状态机被嵌入到父 Future 中。然而，这种方式通常只能处理静态数量的并发 Future [4]。

FuturesUnordered 则提供了一种混合方案。它允许用户将动态数量的 Future 推入一个集合中进行并发轮询，这些 Future 在 FuturesUnordered 本身被轮询时得到执行，而不是被单独派生为独立的任务。重要的是，这些 Future 不需要是 'static 的，它们可以借用周围的状态，只要 FuturesUnordered 实例的生命周期不超过被借用的数据。这使得 FuturesUnordered 结合了多任务并发的动态性与任务内并发的借用能力 [6]。

尽管功能强大，FuturesUnordered 也因其相关的复杂性和潜在的错误而成为异步 Rust 用户困扰的来源。文章重点讨论了两种主要的 FuturesUnordered 使用模式及其相关问题：

1. **“缓冲流”模式 (The “buffered stream” pattern)**: 在这种模式下，工作被表示为一个 Future 流，然后使用像 StreamExt 提供的 buffered 或 buffer_unordered 这样的适配器进行缓冲。当缓冲空间可用时，可以推入更多的工作；当结果就绪时，进行处理。这种模式通过限制缓冲 Future 的数量来实现反压（backpressure），从而对生成 Future 的上游流施加压力。然而，缓冲流的顺序执行特性——只有当 Buffered 流有空间时才会轮询底层流，并且只有当处理循环准备好接收结果时才会轮询 Buffered 流——可能导致并发度低于预期，甚至在特定情况下引发死锁 [6]。
2. **“作用域任务”模式 (The “scoped task” pattern)**: 这种模式下，工作被视为通过句柄“派生”到 FuturesUnordered 上的任务，Niko Matsakis 的 moro 库是这种模式的一个例子。用户可以从任何已派生的任务中派生更多的任务。结果通过等待 JoinHandle 或通过同步原语传输数据来处理。moro 中的这种实现本身不提供反压机制，允许用户派生无限数量的任务（尽管可以通过异步的 spawn 来实现反压）。作用域任务方法默认不引入顺序执行点；任务独立进行，顺序由用户通过并发原语施加 [6]。

作者建议，为了降低死锁风险，应优先考虑“作用域任务”模式，遵循“缓冲数据，而非代码”（Buffer data, not code）的原则。因为在作用域任务模式下，共享资源（如通道）更为明确，从而使得循环依赖更容易被发现 [6]。

这些抽象，如 Pin 和 FuturesUnordered，虽然强大，但也引入了它们自身的复杂性和潜在的误用风险（如死锁、非直观行为）。作者的工作常常涉及剖析这些成本，并倡导采用能够减轻这些成本的模式或新的抽象（如 merge! 宏、偏好“作用域任务”模式）。这反映了对 Rust “零成本抽象”理念的一种务实态度——虽然运行时成本可能为零，但认知成本或（如果误用）安全成本并非为零，需要仔细管理。

C. 理解并缓解 **Async Rust** 中的死锁 [6]

死锁是并发系统中一个经典的问题，在异步 Rust 中也可能发生。文章详细探讨了死锁的产生条件以及与 FuturesUnordered 相关的特定场景 [6]。

死锁的发生通常需要满足四个被称为科夫曼条件 **(Coffman conditions)** 的必要条件：

1. 互斥 **(Mutual exclusion)**: 任务对所需资源拥有独占控制权。
2. 占有并等待 **(Wait for / Hold and wait)**: 任务在持有已分配资源的同时，等待获取额外的资源。
3. 不可抢占 **(No preemption)**: 资源不能被强制从持有它的任务中移除，必须由任务使用完毕后自愿释放。
4. 循环等待 **(Circular wait)**: 存在一个任务链，其中每个任务都持有下一个任务所请求的资源，形成循环。

文章通过一个简单的例子演示了异步 Rust 中的死锁：一个异步生成器（async generator）在持有一个 Mutex 锁的情况下 yield 一个值，而消费这个值的 for await 循环尝试获取同一个 Mutex 锁来处理该值。这导致了循环等待。在这个单锁场景中，第二个共享资源被识别为“代码执行本身的控制权”，因为在单个任务中，一次只有一个部分可以执行。这种问题并非异步 Rust独有，而是任何命令式过程语言中都固有的。防止这种死锁需要抢占对其中一个资源的独占访问，要么是 Mutex（例如，使用可重入锁，但这可能引入微妙的逻辑错误），要么是计算控制权（这在具有显式顺序执行的命令式语言中不可能）[6]。

将这种对死锁的理解应用于 FuturesUnordered：

- 缓冲流模式下的死锁: 如果一个被缓冲的 Future 获取了一个锁，而外部的 for 循环（或其异步处理体）尝试获取同一个锁，就会发生死锁。即使 poll_progress（后文将讨论）解决了某个顺序执行点问题，反压机制引入的顺序执行点（即只有当缓冲区有空间时才轮询上游流）仍然是一个潜在的死锁源，如果上游流或被缓冲的 Future 获取了锁。
- 作用域任务模式下的死锁: 文章给出了一个例子，两个子任务通过一个长度为0的有界通道进行通信。一个任务在持有锁的情况下发送数据，另一个任务接收数据并尝试获取同一个锁。这同样导致了循环等待，此时通道空间取代了任务控制权成为第二个资源。

文章从中得出两个重要结论：

1. **“缓冲数据，而非代码” (Buffer data, not code)**: 当锁和通道等同步原语使共享资源变得明确时，循环依赖更容易被发现。这表明应优先选择作用域任务模式而非缓冲流模式，以减少并发单元之间因隐式共享资源而导致的死锁风险。
2. 死锁最终源于对缓冲单元（缓冲流中的并发 Future 或通道中的对象）数量的限制，而这些限制对于实现反压是必要的。如果一个循环依赖在两个方向上都是有界的，并且队列都满了，就可能发生死锁。设置任意大的队列大小可能会掩盖潜在的死锁，使其在生产环境中变得不太可能发生，但并非不可能。

一个关键的观察是，死锁只有在使用同步原语（如锁、通道）时才可能发生。FuturesUnordered 本身，如果不与这些原语结合使用，是不会导致循环等待的。这是因为 Rust 的所有权和借用系统会阻止“占有并等待”条件的形成——代码等待的唯一“资源”是当前任务的控制权，没有多个资源就不会有循环等待，否则会导致借用检查器错误 [6]。这强调了使用 FuturesUnordered 的一个主要动机：避免共享所有权和同步原语，从而避免潜在的死锁。Rust 类型系统能够实现这一点，被认为是其重要优势。

对“缓冲数据，而非代码”的建议以及对“作用域任务”模式的偏好 [6]，与一个更广泛的主题相呼应，即让资源依赖和潜在的竞争点变得明确。隐式的共享或顺序执行（如缓冲流模式中的反压机制）可能会隐藏死锁的可能性。这与 Rust 在内存管理和错误处理等领域通常所体现的显式性是一致的。

IV. 异步迭代的演进与细微差异

异步迭代是处理一系列异步事件的核心机制。在 Rust 中，这主要通过 AsyncIterator Trait（及其前身 Stream Trait）来实现。其设计和稳定性对于整个异步生态系统的健康至关重要。

A. AsyncIterator 与 Stream 的核心地位 [2]

AsyncIterator 用于表示一个可以异步产生一系列值的序列。它不仅是简单迭代任务的基础，其稳定和恰当的设计还被认为是解决异步生态系统中其他问题的关键，例如前文讨论的某些 Pin 相关的人体工程学问题 [5]。作者指出，AsyncIterator 在 async/await MVP 发布四年后仍未稳定，这对异步 Rust 造成了伤害，因为基于异步迭代的 API 被限制在不稳定特性或第三方库中，使得用户在处理重复性异步事件（一种非常常见的模式）时感到困惑且缺乏良好支持 [3]。作者认为，Rust 项目能为用户做的最好的事情就是稳定 AsyncIterator，以便生态系统可以在其上构建。

Pin 问题 [5]、FuturesUnordered 的复杂性 [6] 以及对 AsyncIterator 的需求 [2] 之间存在着深刻的内在联系。例如，一个设计良好并带有 merge! 和 for await 功能的 AsyncIterator 被提议作为解决许多 Pin 相关人体工程学问题的主要方案。这表明，改进一个核心抽象可以对其他抽象的可用性产生连锁的积极效应。作者推动 AsyncIterator 稳定的动机，也源于这些更广泛的生态系统效益。

B. poll_next vs. "Async Next" 设计之争：深度分析 [2]

关于 AsyncIterator Trait 的具体设计，存在一个长期的争论，主要集中在核心的 next 方法应该如何定义。两种主要的设计方案是：

- **Poll Next (当前提案):** 这是作者强烈支持的设计。它在 AsyncIterator Trait 中定义一个 poll_next 方法，其签名类似于 Future::poll：fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Option<Self::Item>>;。在这种设计下，AsyncIterator 本身

是一个单一的状态机，在整个循环迭代过程中被固定 (pinned) 和激活 [2]。
- **Async Next (替代方案)**: 该方案提议使用一个 async fn next(&mut self) -> Option<Item>; 作为 Trait 方法。在这种设计下，存在两个状态机：一个是为每次 next() 调用创建的短生命周期的 Future（在单次迭代中被固定），另一个是更长生命周期的底层 Iterator（在整个循环中是非固定的）[2]。

作者认为追求 "async next" 设计是一个错误，并详细论证了 "poll next" 的优越性 [2]：

- 性能: "Async next" 引入了额外的间接层（第二个短生命周期状态机），LLVM 编译器可能并不总能优化掉这种间接性。这对于动态分发尤其成问题，因为每次迭代都可能需要为 next() 返回的 Future 进行动态分配。相比之下，"poll next" 设计由于只有一个状态机，可以“免费”地支持动态分发，无需额外的分配 [2]。
- 固定 **(Pinning)**: 在 "async next" 设计中，只有短生命周期的 Future 被固定，而长生命周期的迭代器本身则不是。这对于那些使用侵入式链表进行同步的异步原语（例如 tokio 中的某些原语）是有问题的，因为这些原语通常要求其状态机被固定以在通知队列中保持其位置。如果只有短生命周期的 Future 能加入通知队列，那么丢弃该 Future（例如，由于与另一个 Future 竞争失败）将导致其失去位置，可能导致某些接收者饿死。"Poll next" 设计则直接固定了 AsyncIterator 本身，避免了这个问题 [2]。
- 取消 **(Cancellation)**: "Async next" 中 next() 方法返回的 Future 可以具有有意义的取消行为（即取消它会对后续操作产生影响），这种行为可能对用户来说并不明显，类似于在循环中使用 select! 时遇到的问题——未完成的 Future 在每次迭代中被取消和重建。而在 "poll next" 设计中，取消对 poll_next 方法的调用（即丢弃 poll_next 返回的 Pending 状态）对迭代器本身的内部状态没有影响，下次调用 poll_next 时会从之前的状态继续 [2]。

关于人体工程学，"async next" 的支持者认为它避免了用户直接处理 Pin 和 Poll，因此更易用。然而，作者反驳说，真正解决 AsyncIterator 手动实现复杂性的方案是提供异步生成器（async generators）语法。异步生成器允许用户以类似 async fn 的方式编写迭代逻辑，编译器负责生成底层的状态机。这种语法与 "poll next" 设计完全兼容，并且可以为那些希望获得 async fn next 体验的用户提供简单的转换 [2]。

作者进一步强调，AsyncIterator 是 Iterator 和 Future 的乘积 **(product)**，它同时体现了迭代和异步的特性，而不是简单地将 Iterator 异步化。试图修改其中一个以产生另一个是武断的 [2]。

这场 "poll_next" 与 "async next" 的辩论 [2] 是 Rust 异步设计中一个更广泛的哲学张力的缩影：

- **"Poll next" + 异步生成器**：倾向于在 Trait 层面保持显式性（poll_next 的机制是可见的），但为常见情况提供人体工程学的语法糖（异步生成器）。这维持了底层库作者所需的控制力，同时为应用开发者提供了易用性。它符合 Rust 通常的策略，即提供强大的

底层原语，并在其上构建便捷的抽象。

- **"Async next"**：试图在 Trait 层面实现更内在的简洁性，但这可能以牺牲性能、带来固定复杂性以及更复杂的底层机制（两个状态机，可能涉及“关键字泛型”或“效应系统”）为代价。作者对 "poll next" 的强烈偏好[2]反映了一种信念，即核心 Trait 应该暴露底层的状态机机制以获得最大的灵活性和性能，而人体工程学问题应通过语法（生成器）来解决。

下表系统地比较了 poll_next 和 "async next" 设计在多个关键维度上的差异：

表 2：**AsyncIterator** 的 **poll_next** 与 **"async next"** 设计比较

特性	**poll_next** 设计	**async next** 设计	作者评估/偏好 (基于)
状态机模型	单个 AsyncIterator 状态机，在其生命周期内被固定	两个状态机：为每次 next() 调用创建的短生命周期 Future (固定)，以及长生命周期的 Iterator (非固定)	poll_next 更简单，状态机单一，与 Future 模型一致。
性能 (动态分发)	对动态分发友好，通常无额外分配	每次迭代可能需要为 next() 返回的 Future 进行动态分配，对动态分发不利	poll_next 在动态分发场景下性能更优，更符合 Rust 的零成本抽象原则。
固定 (Pinning) 行为	AsyncIterator 本身被固定，与使用侵入式数据结构（如链表）的异步原语兼容良好	仅短生命周期的 Future 被固定，长生命周期的迭代器未固定，可能与某些现有库的实现冲突	poll_next 的固定行为更稳健，能更好地支持需要固定状态的底层库。
取消 (Cancellation) 语义	取消对 poll_next 的轮询（丢弃 Pending）对迭代器内部状态无副作用	next() 返回的 Future 的取消可能具有副作用，这种行为可能不明显，增加了复杂性	poll_next 的取消语义更简单、更可预测，减少了用户意外引入错误的风险。
人体工程学 (Trait 层面)	在 Trait 定义和实现时需要直接处理 Pin 和 Poll	在 Trait 定义层面隐藏了 Pin 和 Poll 的复杂性	"async next" 在 Trait 层面看似更简单，但这种简单性可能掩盖了底层的复杂性。
人体工程学 (配合生成	异步生成器可以极大地简化 AsyncIterator 的	异步生成器同样可以用	配合异步生成器，两种设计都能提供良好的人

器的用户层面)	实现，用户无需直接操作 poll_next	于简化实现	体工程学。关键在于底层 Trait 的稳健性。

AsyncIterator 稳定性的延迟 [3] 可能对异步生态系统产生了显著且不易察觉的连锁反应。除了缺乏异步迭代功能本身，它还阻碍了其他问题的解决方案（如 Pin 的人体工程学问题，依据 [5] 的分析），并可能导致更复杂的变通方法或次优的库设计。这表明，基础性但悬而未决的特性会成为更广泛生态系统发展的瓶颈。

C. 应对迭代挑战：**poll_progress** 的提出 [7]

即使 AsyncIterator 的核心设计得以确定，仍然存在一些与特定用例相关的挑战。其中一个被称为“芭芭拉与缓冲流的战斗”（Barbara battles buffered streams）的问题，涉及 futures 库中的缓冲适配器（如 Buffered 和 BufferUnordered）与 for await 循环的交互方式，特别是当循环体本身是异步的（即包含 .await 表达式）[7]。

问题在于，当 for await 循环的循环体因为 .await 而暂停时，缓冲适配器中的子 Future（即被缓冲的那些 Future）将不会取得进展。这是因为这些子 Future 通常只在调用 AsyncIterator 的 poll_next 方法时才被轮询。如果循环体长时间挂起，即使缓冲中的某些子 Future 已经准备好完成，它们也无法执行。这种行为对用户来说是不直观的，并且也是不必要的，因为 for await 循环拥有异步迭代器的所有权，可以防止迭代器失效 [7]。

Tyler Mandry 曾建议通过回归到内部迭代（internal iteration）来解决这个问题，即让流暴露 for_each 这样的方法而不是 poll_next。其想法是，这允许缓冲流适配器在执行传递给 for_each 的闭包的同时，并发地执行其缓冲逻辑。然而，作者认为内部迭代有其固有的重大缺点，这些缺点在 Rust 迭代和并发历史中已有记载，包括：

1. 无法在循环内部使用普通的控制流，如 return 或 ?，因为循环处理是作为闭包编写的，而这些操作符对闭包不透明，这将需要复杂的组合子。
2. 难以定义从迭代器返回引用的健全 API，因为不能保证循环在被指示中断时实际中断。
3. 可能无法定义零成本的“交错”操作符（如 zip）来并发运行两个子异步迭代器 [7]。鉴于 Rust 为 Iterator 和 Future 选择了无栈状态机，作者得出结论，内部迭代对于 Rust 来说不是一个可行的解决方案。

作为替代方案，文章提出了 poll_progress [7]。这涉及向 AsyncIterator Trait 添加一个额外的方法：

Rust

```
trait AsyncIterator {
    type Item;
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<Self::Item>>;
    fn poll_progress(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<()>; // 新增方法
}
```

poll_progress 方法允许 AsyncIterator 在不调用 poll_next 的情况下取得进展。实现者应该使 poll_progress 在取得进一步进展的唯一方法是调用 poll_next 时返回 Ready(())。对于大多数异步迭代器，这通常总是成立的。然而，对于像 Buffered 和 BufferUnordered 这样的缓冲流，poll_progress 会继续轮询子 Future，直到达到最大缓冲数量。

为了利用 poll_progress，for await 循环的脱糖（desugaring）过程将被修改为：当循环体处于 Pending 状态时，在异步迭代器上轮询 poll_progress。这有效地使迭代器与循环体并发运行。对于像 Buffered 这样的缓冲流，这意味着所有子 Future 将在循环执行的每个点被轮询，从而解决了“与缓冲流的战斗”问题。如果最大数量的子任务在循环体完成之前完成，poll_progress 将返回 Ready(())，表明在通过 poll_next 从队列中取出一个任务之前，不能再轮询更多的子任务。这种机制确保了反压得以维持 [7]。

“芭芭拉与缓冲流的战斗”问题 [7] 突显了看似直观的抽象（如并发缓冲区）在与其他异步特性（如异步循环体）结合时，可能会变得“泄漏”。缓冲项不取得进展是一种非显而易见行为。poll_progress 是通过使并发更明确并可由 for await 循环控制，来修补这种泄漏的一种尝试。这表明需要仔细考虑不同异步组件如何交互。

V. Async 系统中的并发模型与性能考量

异步 Rust 的设计不仅关乎语言特性，也深刻影响着并发程序的架构方式和性能表现。核心的争论和考量点包括执行器模型、任务内并发与多任务并发的选择，以及共享状态对性能的影响。

A. 工作窃取 (Work-Stealing) vs. 每核一线程 (Thread-Per-Core)：批判性审视 [8]

在 Rust 社区中，关于异步运行时（如 Tokio、async-std、smol）默认采用带有工作窃取机制的多线程执行器存在一些争议。部分用户对此表示强烈不满，认为这是一种“过早优化”，并给代码带来了不必要的 Send + 'static 约束，从而“扼杀了编写 Rust 的乐趣” [8]。

这些用户倡导的替代架构是所谓的“每核一线程”。然而，“每核一线程”这个名称本身就存在歧义，因为被批评的多线程执行器实际上也是在每核创建一个操作系统线程，并在其上调度大量任务。Pekka Enberg 对“每核一线程”架构的定义更为清晰，它包含三个主要思想：

1. 并发应在用户空间处理，避免昂贵的内核线程。
2. I/O 应该是异步的，以防止阻塞每核线程。
3. 数据在 CPU 核心之间进行分区，以消除同步成本和 CPU 缓存之间的数据移动 [8]。Enberg 指出，前两点对于高吞吐量系统至关重要，而异步 Rust 已经满足了这两点。争议的核心在于第三点，即数据分区，这涉及到工作窃取和最小化共享状态这两个相互制约的优化方向。

工作窃取 **(Work-Stealing)**:

- 目的: 通过确保每个线程保持活跃来改善尾部延迟 (tail latency)。在实际系统中，任务所需的工作量往往不同，即使进行了初始工作分配，也可能导致某些线程过载而其他线程空闲。工作窃取允许空闲线程从过载线程“窃取”工作，从而减少尾部延迟并提高 CPU 利用率 [8]。
- 缺点: 工作窃取意味着一个任务可能在一个线程上暂停，在另一个线程上恢复。这就要求任何跨越 await 点（即屈服点）使用的状态都必须是线程安全的，在 Rust 中这通常表现为 Future 需要满足 Send Trait 约束。这给开发者带来了一定的挑战。此外，在线程间移动状态会产生同步成本和缓存未命中，这与“无共享”（share-nothing）架构的原则相悖，后者强调每个 CPU 对其操作状态拥有独占访问权 [8]。

无共享架构 **(Share-Nothing Architecture)**:

- 目标: 通过将数据保留在单个 CPU 核心的快速缓存中，而不是在多个核心共享的较慢缓存中，来改善尾部延迟 [8]。
- 实现: Enberg 的论文通过一个键值存储的例子展示了无共享架构的性能优势。该键值存储使用哈希函数对键空间进行分区，并使用 SO_REUSEPORT 对传入的 TCP 连接进行分区。请求随后通过消息传递通道从连接管理线程路由到相关的键空间管理线程。这与 memcached 的线程共享分区键空间（每个分区由互斥锁保护）形成对比。Enberg 的论文表明，使用通道而非互斥锁可以带来更低的尾部延迟，这很可能是因为频繁访问的分区保留在单个核心的缓存中，从而减少了缓存未命中。
- 作者观点: 作者承认，精心设计的、避免 CPU 缓存间数据移动的系统会产生更好的性能，尤其对于像键值存储这样易于分区的状态。然而，对于需要跨多个分区进行事务性或原子性突变的更复杂应用，实现无共享架构要困难得多。作者质疑 Enberg 的架构是否比 memcached 的架构更容易实现，并暗示通过高级内核特性和仔细的架构设计来避免数据移动可能比简单地用互斥锁包装数据更难。作者还怀疑那些抱怨 Send 约束的开发者是否真的在进行如此复杂的工程设计。如果已经在使用共享状态，那么工作窃取很可能在高负载下提高 CPU 利用率 [8]。

这场关于每核一线程与工作窃取的辩论 [8] 突显了理想化的“无共享”架构（通常在特定条件下基准测试更好）与现实世界应用的实用性之间的张力，后者可能具有复杂的状态依赖或不均匀的工作负载。工作窃取虽然可能给代码带来 Send 约束，但在非理想场景下是最大化利用率的务实解决方案。流行的运行时中默认的工作窃取模型，对于通用的异步编程而

言，是一种合理的折衷，而纯粹的无共享则是专门的优化。

B. 任务内并发 **(Intra-Task) vs.** 多任务并发 **(Multi-Task)**：选择正确的方法 [4]

异步 Rust 提供了不同粒度的并发方式，理解它们的区别和适用场景对于编写高效且易于维护的代码至关重要。

- 多任务并发 **(Multi-task concurrency)**:
  - 定义: 指将不同的操作派生为独立的、并发执行的任务（通常由执行器管理）。这类似于传统的多线程编程模型。
  - 特征: 任务之间通常需要通过同步原语（如异步 Mutex、异步通道）进行通信和协调。被派生的任务（例如通过 tokio::spawn）通常需要满足 'static 生命周期约束，因为它们可能比创建它们的上下文活得更久，这意味着它们不能直接借用栈上的数据，往往需要 Arc 来共享数据 [4]。
  - 适用场景: 适用于表示逻辑上独立的、可能长时间运行的操作单元，或者当 'static 约束可以接受时。
- 任务内并发 **(Intra-task concurrency)**:
  - 定义: 指在单个任务的上下文中，并发地执行多个异步操作。
  - 特征: 通过像 join!（等待所有提供的 Future 完成）和 select!（等待第一个完成的 Future）这样的组合子宏来实现。这些并发操作的状态直接嵌入到父 Future（即调用 join! 或 select! 的 async 块或函数所代表的 Future）的状态机中，从而避免了为每个并发操作进行单独的堆分配或任务创建开销。它们可以直接借用父任务作用域内的数据。然而，任务内并发通常只能处理在编译时数量固定的并发操作。此外，它不实现真正的并行性（parallelism），因为所有操作最终都在同一个任务的上下文中被轮询；因此，它不适合计算密集型工作。取消父任务必然会导致所有子操作被取消 [4]。
  - 作者强调: 任务内并发是异步 Rust 的一个独特且强大的特性，它得益于基于就绪状态的 Future 设计，从而实现了极高的效率。例如，一个网络服务器的单个连接处理任务，可以在内部使用 select! 来并发处理该套接字上的入站和出站读写，以及来自其他任务的消息。这种架构在高级别上类似于 Actor 模型，但由于任务内并发，它可以编译成每个套接字一个高效的状态机，其运行时表示非常接近于用 C 语言手写的异步服务器 [4]。

作者对任务内并发的强调 [4] 不仅仅关乎性能，更在于它提供了一种在质量上不同的方式来构建并发逻辑，这是 Rust 的 Future 模型所独有的能力。这种将多个并发操作的状态直接嵌入父 Future 而无需单独分配或任务开销的能力，是基于就CSS状态轮询设计的直接结果，也是区分 Rust 异步与其他并发模型（如 Actor 模型或基于线程的并发）的一个关键“赋能”。

选择哪种并发模型取决于具体的场景和需求。开发者需要认识到各自的限制，并根据操作

的独立性、生命周期、数据共享需求以及是否需要真正的并行性来做出决策。

C. 无共享架构与缓存一致性考量 [8]

无共享架构的核心目标是通过将数据和处理尽可能地限制在单个 CPU 核心内，来最大限度地减少核心间的通信和缓存同步开销，从而提升性能，特别是尾部延迟。当数据被多个核心共享和修改时，需要通过缓存一致性协议（如 MESI）来确保所有核心看到的是数据的最新版本。这个过程会引入延迟，因为一个核心修改数据后，其他核心的相应缓存行可能需要被置为无效或更新，这涉及到总线通信和潜在的内存访问。

无共享架构试图通过以下方式缓解这些问题：

- 数据分区: 将整体数据集划分为多个独立的分区，每个分区主要由一个特定的 CPU 核心负责。
- 请求路由: 将传入的请求根据其涉及的数据导向到负责该数据分区的核心。
- 消息传递: 核心间如果需要通信，则通过显式的消息传递机制，而不是直接共享内存。

这种设计可以显著减少缓存争用和伪共享（false sharing）等问题，因为每个核心主要操作其本地缓存中的数据。Enberg 的键值存储例子 [8] 就体现了这一点，通过将键空间分区并使用 SO_REUSEPORT 将连接绑定到特定核心，实现了请求的本地化处理。

然而，实现真正的无共享架构并非没有代价。其主要的权衡在于：

- 潜在性能收益: 对于那些数据可以清晰分区且核心间交互较少的应用（如某些类型的键值存储、无状态服务），性能收益可以非常显著，尤其是在高并发和低延迟要求的场景下。
- 设计复杂度增加: 设计有效的数据分区策略、请求路由逻辑以及高效的消息传递机制，本身就是一项复杂的工程任务。对于那些数据依赖关系复杂、需要频繁跨分区事务或原子操作的应用，无共享架构的实现难度会急剧上升，甚至可能不适用。

作者暗示，对于许多应用程序而言，实现一个严格的无共享设计的复杂性可能超过其带来的好处，特别是如果应用中存在难以避免的共享状态，或者工作负载本身不均衡。在这种情况下，带有工作窃取的执行器模型，尽管引入了 Send 约束和一定的同步开销，但可能通过更有效地利用所有核心来提供更好的整体性能和资源利用率。关于 Send 约束的抱怨，也常常成为一个焦点，它迫使开发者显式地处理线程安全，这本身是 Rust 的一个核心原则，即便它增加了复杂性。Send 要求的讨论不仅仅是关于一个 Trait 约束，它反映了对并发架构的深层假设以及开发者管理线程安全的意愿。

VI. Async Rust 的发展轨迹：路线图分析

异步 Rust 自 2019 年 MVP (Minimum Viable Product) 发布以来，虽然取得了显著的行业应用，但在语言特性扩展方面进展相对缓慢。这在一定程度上导致了社区中对其复杂性和用户体验的一些负面评价。"without.boats" 的作者在 "A four year plan for async Rust" [3]

一文中，详细阐述了其对异步 Rust 未来发展的设想，旨在改善用户体验并补全缺失的功能。该计划将所需特性按预期时间表分为近期、中期和长期三个阶段。

这份“四年计划”[3] 在很大程度上阐明了因将 async/await 作为 MVP 发布而产生的“技术债务”。许多“近期”特性，如 AsyncIterator、异步生成器和协程方法/闭包，可以说是“完整”异步系统本应包含的部分。在交付这些特性方面的缓慢进展，加剧了用户的挫败感，并导致了异步 Rust 在复杂性方面的声誉。

A. 近期特性 **(**预计未来 **18** 个月至 **2** 年**)** [3]

这些特性被认为改动相对较小，依赖于已有的抽象能力，且语法层面也与现有特性一脉相承，因此有望在短期内实现。

1. **AsyncIterator** 和异步生成器 **(Async Generators)**:
   - 核心作用: AsyncIterator 对于处理重复性的异步事件至关重要。作者强烈主张尽快稳定 AsyncIterator Trait（采用 poll_next 设计），认为其缺失已对生态造成损害。
   - 异步生成器: 语法如 async gen fn sum_pairs(...) yields i32 {... }，它们会被编译成实现了 AsyncIterator 的状态机。这将极大简化 AsyncIterator 的手动实现。
   - **for await** 循环: 这是消费 AsyncIterator 的基本语法，允许在异步上下文中遍历异步序列，并在迭代器返回 Pending 时让出控制权。
2. 协程方法 **(Coroutine Methods)** 与返回类型标记法 **(Return Type Notation - RTN)**:
   - 背景: 异步 Trait 方法（async fn in traits）即将稳定。生成器和异步生成器也应能自然地在 Trait 中使用。
   - **RTN** 的需求: 在 Trait 中添加协程方法（如 async fn）会隐式地为该 Trait 添加一个匿名的关联类型（即该方法的返回类型，通常是 impl Future）。有时用户需要对这个匿名关联类型添加额外的约束（例如，Send），尤其是在将该方法返回的 Future 派生到工作窃取执行器的任务中或跨线程移动时。RTN 提供了一种语法来声明这种约束，例如 where F: Foo + Send, F::foo(): Send。
   - **“**能修复吗？**”**原则: 作者认为 RTN 对于遵循“能修复吗？”（Can you fix it?）原则至关重要。即使用户依赖的库中的异步方法最初没有声明某个约束（如 Send），用户也应该能够通过 RTN 在自己的代码中添加这个约束，而无需 fork 该库。
3. 协程闭包 **(Coroutine Closures)**:
   - 建模方式: 作者倾向于将异步闭包（以及生成器闭包）建模为返回 impl Trait（例如 F: Fn() -> impl Future）的闭包，而不是为每种协程引入新的 AsyncFn、AsyncFnMut、AsyncFnOnce Trait 层级。后者会导致 Trait 数量激增（例如，普通、异步、生成器、异步生成器四种，每种三种 Fn 变体，共 12 个 Trait）。前者只需调整现有的 Fn Trait。
   - 所需改动: 这需要调整 Fn Trait，使其 Output 关联类型能够捕获输入的生命周期，并且 -> impl Trait 的脱糖方式可能需要修改（可能涉及 Rust 版本更迭）。

- 语法糖: 建议为协程闭包的 Trait 约束添加类似函数声明的 async 或 gen 语法糖，例如 where F: async FnOnce() -> T 等价于 where F: FnOnce() -> impl Future<Output = T>。

B. 中期特性 **(**难度较大，预计 **2-3** 年**)** [3]

这些特性涉及更复杂的设计问题和实现工作。

1. 对象安全的协程方法 **(Object-Safe Coroutine Methods)**:
   - 问题: 尽管异步 Trait 方法即将稳定，但它们最初不会是对象安全的（即不能用于 dyn Trait）。主要障碍在于，每个协程方法都意味着一个匿名的关联 Future 类型，其大小和布局在不同实现中各不相同。为了对 Trait 对象进行类型擦除，也需要对这个返回的 Future 类型进行某种形式的类型擦除，通常是将其转换为另一个 Trait 对象，如 Pin<Box<dyn Future<Output = ()>>>。
   - 语法和分配: 这种转换可能导致非常冗长的类型签名。此外，关于 Future 状态机在堆上分配的显式性也存在争议。一些人主张使用新的包装类型（如 Boxed<dyn Foo>）来明确指示堆分配，而作者倾向于为堆分配的 Trait 对象（如 Box<dyn Foo>）提供一个合理的默认行为，即也使用相同的分配器为 Future 状态机分配内存。
   - 自定义**"**胶水代码**"**: 无论默认行为如何，都应允许用户提供自定义的"胶水代码"来处理 Future 的分配和转换，例如使用栈分配（如 alloca）。作者认为 dyn* 特性并非此功能的先决条件。
2. 异步析构函数 **(Async Destructors)**:
   - 需求: 有时，一个类型的析构逻辑（Drop Trait）可能需要执行 I/O 操作或以其他方式阻塞当前线程。理想情况下，应支持非阻塞的析构函数，它们可以 yield 控制权，以便其他任务可以并发运行。
   - 挑战:
     1. 尽力而为 **(Best-effort execution)**: 如果一个带有异步析构函数的类型在非异步上下文中被丢弃，那么其异步析构逻辑将无法运行。可能的解决方案包括引入 let async 绑定来限制这类类型不能被移动到非异步上下文，或者仅仅接受异步析构作为对同步析构的一种优化。
     2. 状态存储: 如果异步析构函数需要自身的状态（例如，一个 Vec 在析构时需要知道哪些元素已经被异步地 drop 了），那么这些状态存储在哪里是一个问题，这与 Trait 对象中 Future 状态的存储问题类似。一种选择是禁止异步析构函数拥有状态，采用轮询方法，但这对于数据结构等场景可能不可接受。
     3. 与展开 **(Unwinding)** 的交互: 如果在栈展开过程中遇到一个异步析构函数，并且它返回 Pending，那么会发生什么？需要某种异步版本的 catch_unwind，以便挂起的调用可以跳转到那里，让其他任务运行。
   - 作者对异步析构函数的实用性与实现难度之间的权衡持矛盾态度。

C. 长期特性 **(**可能涉及重大语言变更，预计 **4** 年以上，如 **2027** 版**)** [3]

这些特性旨在解决 Rust 类型系统层面的一些深层限制，可能需要跨版本进行重大更改。

1. 处理不可移动 **(Immovable) /** 不可忘记 **(Unforgettable) /** 不可丢弃 **(Undroppable)** 类型:
    - 不可移动类型: 对于实现自引用协程（特别是生成器）和侵入式数据结构至关重要。当前的 Pin API 是一个笨拙的变通方法，且与 Iterator 和 Drop 等现有 Trait 的向后兼容性存在问题（这些 Trait 最初设计时未考虑不可移动性）。
    - **Move Trait** 设想: 如果 Rust 引入一个 Move Trait（默认为所有类型实现，但某些类型如自引用生成器可以选择不实现它），那么 Pin 就可以被弃用。一个对 T:!Move 类型的引用将具有与当前 Pin<&T> 类似的语义。
    - 不可忘记**/**不可丢弃类型 **(**线性类型**)**: “作用域任务三难困境” (scoped task trilemma) 为那些不能被“忘记”（即其析构函数必须运行）的类型提供了强有力的论据。无栈协程无法使用基于析构函数的并发借用技巧，除非有这类类型的支持。
    - 作者立场: 作者认为这些是重大的改变，但可能对于更好地集成现有特性（如协程、迭代器）是合理的。项目组应尽早做出决策，并为生成器等问题找到临时解决方案（例如，要求生成器在使用为迭代器前必须被固定）。

D. 对提议的语言演进的批判性评估

作者提出的路线图 [3] 雄心勃勃，触及了许多已知的痛点。近中期特性大多旨在完善 MVP 的不足，提升用户体验。而长期提议（如 Move Trait）则代表了对 Rust 类型系统规则的根本性调整，其成本和收益都需要仔细权衡。作者的务实精神体现在区分了近期可实现的目标和长期更具探索性的方向。

路线图中的各项提议 [3] 并非孤立，而是高度相互依赖。例如，符合人体工程学的 AsyncIterator 定义依赖于异步生成器的实现。对象安全的协程方法则取决于 RTN 以及关联 Future 类型处理方式的最终方案。更长期的改变，如 Move Trait，将简化 Pin 相关的问题以及自引用生成器的实现。这意味着进展并非总是线性的；一个领域的突破可能会为其他领域扫清障碍，而一个停滞点也可能导致连锁延迟。这种依赖关系网突显了像 Rust 这样的语言演进的复杂性，也解释了为何进展有时感觉缓慢——许多部分需要协同一致。

“能修复吗？”原则，在 RTN [3] 中被明确提及，是许多提议的人体工程学改进背后一个强大的潜在动机。它关乎赋予用户能力，使其能够绕过依赖项中的限制，而无需诉诸于 fork。这一原则可能也延伸到其他领域，例如，希望在对象安全的方法中对分配有更多控制，或者在同步/异步代码之间有更好的互操作性。它反映了一种以用户为中心的语言设计方法，专注于为开发者扫清障碍。

下表概述了 "Async Rust 四年计划" 中的关键特性：

表 **3**: **"Async Rust** 四年计划**"** 关键特性概览

特性类别	特性名称	核心问题	解决方案/机制	主要影响/挑战 (基于)
近期 (Near)	AsyncIterator 与异步生成器	处理重复异步事件；AsyncIterator 长期未稳定，影响生态；手动实现 AsyncIterator 复杂	尽快稳定 AsyncIterator (采用 poll_next 设计)；引入 async gen fn 语法，编译到 AsyncIterator；稳定 for await 循环	对生态系统至关重要；gen 关键字已预留；自引用生成器的固定问题需要解决。
近期 (Near)	协程方法 (含 RTN)	Trait 中的异步方法返回的 Future 类型难以添加额外约束（如 Send）	稳定异步 Trait 方法；引入返回类型标记法 (RTN) 语法，如 F::foo(): Send	遵循“能修复吗？”原则，避免用户 fork 库；RTN 的确切语法尚在讨论中。
近期 (Near)	协程闭包	如何以符合人体工程学且不造成 Trait 激增的方式表示异步/生成器闭包	将协程闭包建模为返回 impl Future/Iterator 的普通闭包，而非引入新的 AsyncFn 等 Trait 系列	避免 Trait 数量爆炸；Fn Trait 的 Output 关联类型需能捕获输入生命周期；-> impl Trait 脱糖可能需调整，或涉及版本更迭。
中期 (Medium)	对象安全的协程方法	异步 Trait 方法难以实现对象安全 (dyn Trait)，因其返回匿名的 Future 类型	返回的 Future 类型需被类型擦除 (如 Pin<Box<dyn Future>>)；讨论了分配的显式性 (如 Boxed<dyn Foo>) 和自定义“胶水代码”	类型签名可能非常冗长；堆分配的默认行为和显式性存在争议；dyn* 作者认为非必需。
中期 (Medium)	异步析构函数	Drop Trait 的析构逻辑中可能需要执行非阻塞 I/O	引入 async Drop；面临执行时机（非异步上下文无法运行）、状态存储、与栈展开交互等难题	设计上非常棘手；作者对其最终实用性与实现难度之间的平衡持观望态度。
长期 (Long)	不可移动/不可忘	Pin API 笨重；自	考虑引入 Move	对 Rust 类型系统

	记/不可丢弃类型	引用协程(如生成器)难以完美支持;作用域任务三难困境等问题	Trait (默认实现，特定类型可选退出以表不可移动);探索线性类型(处理不可忘记/丢弃)	是根本性改变，成本巨大;但可能从根本上解决 Pin 问题和改进自引用类型及资源管理的表达力。作者认为值得认真考虑。

VII. 综合:贯穿始终的主题与批判性视角

"without.boats" 系列博文不仅深入探讨了异步 Rust 的技术细节，更展现了作者在语言设计和生态发展方面的一贯哲学与批判性思考。综合分析这些文章，可以提炼出几个核心主题和视角。

A. 系列中一以贯之的论点与哲学

1. 实用主义优于纯粹理论 **(Pragmatism over Purity)**: 尽管 Rust 语言本身具有坚实的理论基础，但作者在许多论证中展现出对实用主义的倾向，即优先选择那些能够为用户带来实际价值的解决方案，即便它们并非理论上的"完美"形态。例如，主张尽快稳定采用 poll_next 设计的 AsyncIterator [2]，而不是无限期等待一个可能更"理想"但尚无定论的 "async next" 方案。同样，对于异步析构函数，也考虑到其可能是"尽力而为"的执行方式 [3]。
2. 拥抱 **Rust** 的独特性 **(Embrace Rust's Uniqueness)**: 一个反复出现的主题是，应当充分利用 Rust 异步模型(无栈协程、Future、Pin)的独特优势和强大能力，而不是试图将其强行塞入源自其他语言的范式中。这在 "Let futures be futures" [4] 一文中体现得尤为明显。这种哲学也体现在对任务内并发价值的强调上，认为这是 Rust 异步区别于其他模型的一个重要赋能。
3. 用户体验驱动 **(User Experience as a Key Driver)**: 许多论点和提议的出发点都是为了改善开发者的体验，减少样板代码，并使复杂的特性更容易被理解和使用。例如，对异步生成器、返回类型标记法 (RTN) 的倡导 [3]，以及致力于解决 Pin 在高层代码中暴露的问题 [5]，都反映了对提升用户体验的重视。
4. 基础抽象的重要性 **(Importance of Foundational Abstractions)**: 作者认为，正确设计和稳定核心的基础抽象(如 AsyncIterator)对于整个异步生态系统的健康发展至关重要 [2]。这些基础抽象的质量直接影响上层库和应用代码的简洁性、健壮性和性能。

B. 对作者提出的挑战与解决方案的批判性评估

作者的分析无疑具有深度，所识别出的许多问题也确实是异步 Rust 社区面临的真实挑战。例如，关于 Box<T> 总是 Unpin 的决定被认为是"可能无法逆转的" [5]，异步析构函数也被描述为"非常棘手" [3]，这些都反映了某些问题的根深蒂固。

在评估作者提出的解决方案时，也需要考虑其是否完全解决了复杂性，或者是否引入了新

的权衡。例如，虽然 poll_progress [7] 解决了缓冲流在特定场景下的进度停滞问题，但它也给 AsyncIterator Trait 增加了一个额外的方法，略微增加了 Trait 本身的复杂性。对于长期特性如 Move Trait，虽然可能从根本上解决 Pin 的问题，但其对现有代码和整个类型系统的冲击将是巨大的，需要极其审慎的评估。

一个反复出现的解决问题的模式是通过引入新的抽象或语法来处理现有的复杂性或人体工程学问题（例如，用 merge! 解决 select!/Pin 的问题，用 for await 解决 Stream::next/Pin 的问题，用异步生成器简化 AsyncIterator 的定义，用 RTN 处理 Trait 方法的边界，用 Move Trait 作为 Pin 的更根本性修复）。虽然这些旨在简化特定用例，但它们也共同增加了语言表面的整体规模和复杂性。这反映了一种权衡：以增加另一个需要学习/维护的语言特性为代价，解决一个特定的痛点。作者似乎认为，这些新增内容因其解决的问题而是合理的。

C. "without.boats" 所描绘的 Async Rust 演进蓝图

这些文章共同描绘了一幅异步 Rust 作为语言中一个强大但仍在不断成熟的部分的图景。从最初的 MVP 版本开始，取得了明显的进展，但也清晰地认识到仍有大量工作需要完成 [3]。作者既强调了已取得的技术成就（例如 Future 模型本身的设计，它如何从迭代器模型演化而来），也坦诚地指出了持续存在的挣扎（例如，项目决策过程的效率和透明度问题 [3]）。

"without.boats" 系列，当作为一个整体来阅读时，不仅仅是在讲述异步 Rust 的故事，更是在讲述一个复杂的、社区驱动的编程语言演进过程中所面临的普遍挑战。MVP 的局限性 [3]、创新与稳定性之间的张力（体现在对长期特性的讨论中 [3]）、达成共识的困难（如 AsyncIterator 设计的辩论 [2]，以及 await 语法早期的争议 [3]），以及个体贡献者（如作者本人）的深远影响，都是这个宏大叙事的一部分。这些文章揭示了语言发展的曲折、人性化和技术性过程，其中最初的设计选择会产生持久的影响，而路线修正本身也充满复杂性。

此外，虽然这些文章主要聚焦于语言特性和 Trait（如 Future、Pin、AsyncIterator），但异步运行时/执行器（如 Tokio）的行为和设计，是一个至关重要的、虽未被时刻强调却始终存在的背景因素，它影响着许多被讨论的问题和解决方案。例如，工作窃取执行器导致了 Send 约束的必要性 [8]；Tokio 中同步原语的侵入式链表实现与 AsyncIterator 对 Pin 的需求相互作用 [2]。语言特性和运行时是共同演进并相互制约的。

VIII. 对开发者与 Rust 社区的建议

基于对 "without.boats" 系列文章的深入分析，可以为正在使用或关注异步 Rust 的开发者以及整个 Rust 社区提供一些有价值的建议。

A. 对使用 Async Rust 开发者的可行性洞察

1. 深入理解核心抽象的“为何”与“如何”：开发者不应仅仅满足于知道如何使用 async/await，更应努力理解其背后的设计原理，如无栈协程、轮询模型、Pin 的作用

等。这种理解有助于在实际项目中做出更明智的架构决策，例如，何时选择任务内并发而非多任务并发，如何正确处理 Future 的生命周期和数据共享。

2. 警惕 **Pin** 的陷阱并采用推荐模式: 认识到 Pin 可能在循环中的 select!、非 Unpin 流的 next 调用以及盒装 Future 等场景中意外出现 [5]。在可能的情况下，优先使用作者推荐的更高级抽象（如未来可能出现的 merge! 宏、for await 循环）来避免直接操作 Pin。
3. 谨慎使用 **FuturesUnordered**: 理解“缓冲流”模式和“作用域任务”模式的差异及其潜在的死锁风险 [6]。优先考虑“作用域任务”模式，并遵循“缓冲数据，而非代码”的原则，使共享资源和依赖关系更明确。
4. 主动规避异步上下文中的阻塞调用: 认识到“函数不着色问题”的危害 [4]。避免在异步任务中（尤其是在持有锁的情况下跨越 .await 点时）调用可能阻塞线程的同步函数。
5. 关注异步迭代的进展: AsyncIterator 及其相关特性（如异步生成器、poll_progress）对于提升异步代码的简洁性和健壮性至关重要 [2]。开发者应关注这些特性的稳定和推广，并准备好在它们可用时采用。
6. 持续学习和适应: 异步 Rust 仍然是一个快速发展的领域。新的特性、库和最佳实践会不断涌现。开发者需要保持学习的心态，关注社区讨论和官方文档的更新。鉴于异步 Rust 的复杂性和演进特性，高质量的教育材料至关重要，它们不仅要解释如何使用特性，还要解释为何如此设计以及常见的陷阱。

B. 对 **Rust** 项目和生态系统发展的战略性考量

1. 优先稳定和交付基础性特性: Rust 项目应高度重视并优先完成 "Async Rust 四年计划" [3] 中提出的近期特性，特别是 AsyncIterator（及其 poll_next 设计）和异步生成器。这些特性具有广泛的积极影响，能够解锁许多人体工程学的改进，并为生态系统的进一步发展奠定坚实基础。
2. 改善设计过程的沟通与透明度: 呼应作者在 [3] 中表达的担忧，Rust 项目应努力提升其设计和决策过程的沟通效率与对社区的透明度。清晰的路线图、公开的讨论记录以及对社区反馈的积极回应，有助于重建信任并汇聚社区的智慧。
3. 审慎评估根本性的语言变更: 对于像 Move Trait 这样的长期、根本性语言变更提议 [3]，项目组需要进行极其审慎和全面的评估。这需要在解决深层问题、追求技术理想与维护语言稳定性、降低生态系统迁移成本之间取得平衡。任何此类变更都应有充分的论证、广泛的社区讨论以及清晰的过渡计划。
4. 平衡专家需求与初学者友好性: 在异步设计中，Rust 项目面临着一个持续的张力：既要满足专家用户对细粒度控制和极致性能的需求（他们可能愿意接受像 Pin 或 poll_next 这样的复杂性），也要让异步编程对更广泛的开发者群体易于上手（他们可能偏好更简单、更抽象的接口）。作者的许多提议试图通过提供强大的底层原语和符合人体工程学的高层语法（如 poll_next + 异步生成器）来弥合这一差距。保持这种平衡，确保抽象层次分明且过渡顺畅，对于异步 Rust 的持续推广至关重要。
5. 促进高质量运行时和库的发展: 虽然语言特性是基础，但异步 Rust 的可用性在很大程度上也依赖于高质量的运行时（如 Tokio, async-std）和生态库。Rust 项目应继续鼓励

和支持这些组件的健康发展，确保它们与语言特性良好协同，并共同提升用户体验。

IX. 结论

A. "without.boats" Async 系列核心学习回顾

通过对 "without.boats" 博客中关于异步 Rust 的一系列关键文章的深度分析，我们可以清晰地看到作者 Saoirse Shipwreckt 对 Rust 异步编程的深刻洞察、设计哲学以及未来展望。这些文章揭示了异步 Rust 的设计并非凭空产生，而是特定历史决策、语言核心目标（如内存安全、零成本抽象、C ABI 兼容性）以及对性能与人体工程学不懈追求的复杂产物。

核心学习点包括：

- 演进之路: Rust 从最初的绿色线程尝试，到借鉴外部迭代器模型，最终确立了基于 Future Trait 和轮询机制的无栈协程模型，并辅以 async/await 语法糖 [1]。
- 核心抽象与挑战: Pin 的引入是为了解决自引用状态机的内存安全，但其复杂性也给用户带来了挑战 [5]。FuturesUnordered 提供了动态并发管理能力，但也伴随着死锁等潜在风险 [6]。
- 异步迭代的关键性: AsyncIterator 的设计（特别是 poll_next vs "async next" 之争 [2]）及其稳定性，被视为整个异步生态健康发展的基石，并能缓解其他抽象（如 Pin）带来的问题。poll_progress [7] 等提议则致力于解决具体场景下的交互难题。
- 并发模型与性能: 对工作窃取与每核一线程架构的讨论 [8]，以及对任务内并发独特价值的强调 [4]，反映了在不同并发策略和性能目标间的权衡。
- 未来蓝图: 作者提出的“四年计划” [3] 为异步 Rust 的进一步发展规划了清晰的路径，从近期的功能完善到长期的语言级变革，旨在持续提升用户体验和系统能力。

这些讨论共同强调了一个核心观点：异步 Rust 的设计选择是在一系列约束条件下，为了实现高性能、高安全性和良好表达力而做出的务实权衡。

B. 这些讨论的持久影响与现实意义

"without.boats" 系列文章对于任何希望深入理解异步 Rust 的人来说，都是极其宝贵的资源。它们不仅记录了异步 Rust 的当前状态和演进历程，更通过作者富有洞察力的分析和前瞻性的思考，积极地塑造着关于其未来的对话。这些文章因此成为 Rust 社区成员不可或缺的参考读物。

该系列也突显了异步 Rust 并非一个已完成的“产品”，而是一个仍在积极发展、辩论和完善中的“生命系统”。作者的“四年计划”[3] 本身就是这一点的明证。这意味着开发者和整个社区都必须抱持持续学习和适应的心态，以跟上其演进的步伐。

更深层次来看，这些讨论也揭示了技术设计背后的人文因素：关键个体的巨大影响、社区共识的达成之难、对现有流程可能产生的挫败感，以及对技术卓越的不懈热情。作者在回顾

await 语法争议时的个人反思[3]，提醒我们技术进步与这些人文因素是密不可分的。

追求一个高性能、安全且符合人体工程学的异步模型，是 Rust 语言发展过程中的一段持续旅程。Saoirse Shipwreckt 通过 "without.boats" 博客，为这段旅程提供了至关重要的视角、深刻的分析和富有远见的指引，其影响无疑将是持久而深远的。

引用的著作

1. Why async Rust? - Without boats, dreams dry up, 访问时间为 五月 27, 2025, https://without.boats/blog/why-async-rust/
2. poll_next - Without boats, dreams dry up, 访问时间为 五月 27, 2025, https://without.boats/blog/poll-next/
3. A four year plan for async Rust - Without boats, dreams dry up, 访问时间为 五月 27, 2025, https://without.boats/blog/a-four-year-plan/
4. Let futures be futures - Without boats, dreams dry up, 访问时间为 五月 27, 2025, https://without.boats/blog/let-futures-be-futures/
5. Three problems of pinning - Without boats, dreams dry up, 访问时间为 五月 27, 2025, https://without.boats/blog/three-problems-of-pinning/
6. FuturesUnordered and the order of futures, 访问时间为 五月 27, 2025, https://without.boats/blog/futures-unordered/
7. poll_progress - Without boats, dreams dry up, 访问时间为 五月 27, 2025, https://without.boats/blog/poll-progress/
8. Thread-per-core - Without boats, dreams dry up, 访问时间为 五月 27, 2025, https://without.boats/blog/thread-per-core/
9. async - Without boats, dreams dry up, 访问时间为 五月 27, 2025, https://without.boats/tags/async/