Graydon Hoare 與 Rust 共同體：安全系統語言如何從個人原型轉為制度工程

Graydon Hoare and the Rust Community: How a Safe Systems Language Became Institutional Engineering

v1.0 / 2026-07-30 / Neo.K / 公開版／第三部設計師與共同體正式個案研究

https://thisoneisneok.com/html/papers/pldst-022.html

英文名稱： Graydon Hoare and the Rust Community: How a Safe Systems Language Became Institutional Engineering

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-022

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／第三部設計師與共同體正式個案研究

SECTION

摘要

Rust 經常被描述為 Graydon Hoare 創造的一門安全系統程式語言，其 Ownership、Borrow checker 和無 Garbage collector 的 Memory safety 解決了 C／C++ 長期問題。這種敘述抓住 Rust 的創始起點，卻不能準確解釋今日 Rust。

Hoare 在 2006 年以個人專案開始 Rust，早期 Compiler 以 OCaml 實作，語言包含後來被移除的 Typestate、Object、Green thread、Garbage-collected pointer 及多種 Pointer／Channel 機制。Mozilla 於 2009 年開始投資，團隊和 Servo 使 Rust 接受瀏覽器引擎級系統壓力；2012–2015 年間，Ownership、Borrowing、Trait、Lifetime、Cargo、Error model、Compiler infrastructure 及 Library 被大量重新設計。Hoare 於 2013 年離開 Rust 的日常開發，Rust 1.0 於 2015 年發布時已是多人重構後、與初始原型顯著不

Hoare 在 Rust 穩定十週年的回顧中直接反對把 Rust 歷史寫成單一靈感故事。他將 Rust 描述為大量使用者、語言設計者、Compiler 實作者、教育者、機構、企業和志願者共同建立的技術基礎設施；他也承認 2006–2009 年的「Baby Rust」幾乎沒有型別檢查、產生很慢的程式，只能在極少平台運作。到 1.0 時，它已被徹底重寫，幾乎難以辨認為同一語言。[R1]

因此，本篇故意不使用單純的「Graydon Hoare 個人風格個案」格式，而把研究單位分成兩層：

- Graydon Hoare 的原始問題與早期方向；
- Rust 共同體將方向轉成可穩定依賴之制度的方式。

Rust 的原始問題是：

> 如何建立一門能撰寫可靠基礎設施、保留 C／C++ 等級控制和效能，又能在編譯期阻止常見 Memory safety 與 Concurrency 錯誤的語言？

其解法不是單一 Borrow checker，而是完整責任鏈：

> 【Ownership
> +
> Borrowing
> +
> Lifetimes
> +
> Algebraic data types
> +
> Traits
> +
> Unsafe boundary
> +
> Compiler diagnostics
> +
> Cargo ecosystem】

Ownership 將每個 Value 的管理責任指派給 Owner；Borrowing 允許在不移交 Ownership 的情況下暫時存取；Mutable reference 的排他性降低 Alias＋Mutation 交互；Send／Sync 將 Thread transfer 和共享能力放入型別關係；unsafe 則保留 Raw pointer、FFI、Allocator、Kernel 及 Runtime 必須使用的低階操作，但把證明義務明確集中到受審查邊界。[R3][R4][R5]

然而，Rust 的真正歷史創新不只在技術模型。它還建立：

- RFC；
- Feature gate；
- Nightly／Beta／Stable Channel；
- Crater 生態回歸測試；
- 六週 Release；
- Edition；
- Cargo；
- Teams；
- Code of Conduct；
- Foundation；
- Leadership Council；
- 穩定而不停滯。

  Rust 1.0 後，已穩定功能原則上持續支援；需要小幅不相容表面變化時，以 Opt-in Edition 隔離，同一依賴圖內不同 Edition 的 Crate 仍須互操作；重大變更則以 RFC、社群討論、團隊共識、實作及穩定化程序逐步進入。[R6][R7][R8]

  本文將 Rust 歷史分成七個相位：

- 個人原型期：Hoare 以舊語言研究思想尋找可靠系統語言；
- Mozilla 投資期：小型原型轉成團隊和 Servo 測試平台；
- Pre-1.0 激進重寫期：大量特徵移除、Ownership 收斂及 Compiler 自舉；
- 1.0 穩定契約期：Stable channel、Compatibility、Cargo 和生態承諾；
- RFC／Team 聯邦治理期：設計權由創始者轉成公開制度；
- Edition 與長期演化期：局部 Opt-in 修正而不分裂生態；

- Foundation／Leadership Council 基礎設施期：企業資助、法律財務、規格、工具及維護接班。

本文核心判斷為：

> 【Graydon Hoare 創造了問題框架與第一原型，
>
> Rust 共同體創造了今日可被依賴的 Rust。】

Rust 的深層風格不是單一人物品味，而是一種制度化責任配置：

> 【將記憶體和並發證明義務提前給型別系統，
>
> 將不可避免低階風險集中給 Unsafe 邊界，
>
> 將語言演化風險交給公開治理和生態測試。】

這種配置同樣有代價。Borrow checker 會拒絕部分實際安全、但 Compiler 無法證明的程式；Ownership、Lifetime、Trait、Async、Pin、Unsafe invariant 和 Compiler error 形成高學習門檻；RFC、Team、Review、Crater 和 Release infrastructure 需要大量持續勞動。Hoare 自己在十週年回顧中提醒，許多貢獻者會因疲憊或其他需求離開，他本人也只持續約七年；可靠基礎設施的最大問題之一是維護接班，而不是功能創始。

因此，本篇最終判定不是「Graydon Hoare 是 Rust 所有設計的作者」，而是：

> Hoare 是安全基礎設施問題的原型設計者；Rust 共同體則將原型重寫為一套技術、社會及治理共同保證的公共系統。

關鍵詞： Graydon Hoare、Rust、Ownership、Borrowing、Memory safety、Unsafe、RFC、Edition、Cargo、共同體治理、PLDST

SECTION

一、為何標題包含共同體

Rust 1.0 前已經歷大幅重寫；Hoare 於 2013 年離開日常開發；Ownership、Borrow checker、Trait、Cargo、Edition、Async、MIR、NLL、Diagnostics 及現代 Governance 涉及大量不同作者。

若仍把全部內容寫成「Graydon Hoare 的設計風格」，就會重犯 PLDST-004 所批判的創始人歸因偏誤。

SECTION

二、Hoare 可直接歸因的部分

- 2006 年開始私人專案；
- 初始 Compiler；
- 原始可靠系統語言問題；
- 安全、並發、效能及舊研究語言的綜合方向；
- Mozilla 內部推廣；
- 早期團隊建立；
- Rust 名稱及早期語言文化；
- 在 Pre-1.0 階段的多項設計參與。

SECTION

三、共同體可直接歸因的部分

包括但不限於：

- Patrick Walton；
- Niko Matsakis；
- Brian Anderson；
- Dave Herman；
- Aaron Turon；
- Huon Wilson；
- Manish Goregaokar；

- Steve Klabnik；
- Carol Nichols；
- Alex Crichton；
- Yehuda Katz；
- Ashley Williams；
- Mara Bos；
- Language／Compiler／Library／Cargo／Infra／Dev-tools Teams；
- Mozilla、Samsung、AWS、Google、Microsoft、Meta 等投入者；
- Rust Foundation；
- 數千名志願者和企業使用者。

  具體功能仍需逐案歸因。

SECTION

四、Hoare 自己的歸因修正

2025 年回顧中，Hoare明確說：

- 從壞電梯或一個 Idea 講起會錯過更大的故事；
- Rust 是 Stakeholder 共同建立的 Infrastructure；
- 其早期 Bootstrap Compiler 接近無資助個人專案的極限；
- Mozilla 投資後才真正形成 Team；
- 1.0 時語言已被多人徹底重建；
- Rust 的長期成功來自巨大、持續、非單人投入。[R1]

  這是本篇多主體寫法最重要的一手依據。

SECTION

五、問題不是單純取代 C++

Hoare 的目標可概括為：

- Reliable；
- Efficient；
- Concurrent；
- Memory-safe；
- Systems-level；
- 可控制 Layout；
- 不依重型 GC；
- 可建造 Infrastructure。

SECTION

六、從舊研究語言取材

早期 Rust 受到多種較舊語言及研究影響，包括：

- CLU；
- BETA；
- Mesa；
- NIL；
- Erlang；
- Newsqueak；
- Napier；

- Hermes；
- Sather；
- Alef；
- Limbo；
- Cyclone；
- ML／OCaml 系列。

Hoare 曾把方向描述為「過去的技術來拯救未來」。這不是逐項直接複製，而是拒絕「系統語言只能沿 C 家族累加」的歷史宿命。

SECTION

七、早期 Rust 與今日差異

Baby Rust 曾具有：

- OCaml Compiler；
- 很弱或幾乎沒有型別檢查；
- Typestate；
- Object syntax；
- Green thread；
- GC Pointer；
- 多種 Pointer；
- 不成熟 Runtime；
- 極慢 Bootstrap。

因此：

> Rust_(2008)
> ≠
> Rust_(1.0)
> ≠
> Rust_(2026)

SECTION

八、個人原型的真正價值

原型證明：

- 安全系統語言值得投入；
- 多項研究概念可以被放入一個可實作方向；
- Mozilla 可能用它建造 Browser infrastructure；
- 問題足夠重要，值得建立團隊。

  原型不需成為最終規格才算成功。

SECTION

九、Mozilla 投資

Mozilla 於 2009 年開始投資 Rust，使個人專案變成團隊；Hoare 說投資立即使團隊規模增為數倍，並在後續年份持續擴張。[R1]

SECTION

十、Servo 作為壓力測試

Servo 從 2012 年起作為 Rust 的關鍵使用者，要求語言處理：

- Layout；
- DOM；

- Graphics；
- Network；
- Parallelism；
- FFI；
- Browser security；
- 大型 Build；
- Cross-platform。

  語言不再只由小型 Demo 驗證。

SECTION

十一、使用者與設計者重疊

Rust／Servo 團隊可以：

> LanguageChange
> →
> ServoExperiment
> →
> CompilerFeedback
> →
> LanguageRevision

在短週期內共同演化。

SECTION

十二、企業投資的雙重作用

優勢：

- Full-time engineer；

- Infrastructure；
- Testbed；
- CI；
- Compiler；
- Tool。

風險：

- 公司目標偏向；
- 資助集中；
- 組織變動；
- 專案自主性；
- 維護者失業或離開。

2020 年後 Foundation 的形成部分回應資助多元及法律基礎問題。

SECTION

十三、為何可快速移除

在 1.0 前：

- 沒有永久 Stable 契約；
- 使用者預期變化；
- Feature gate；
- Nightly；
- 團隊仍找尋核心；
- Servo 可驗證。

因此可以移除：

- Typestate；
- GC；
- Object syntax；
- 特殊 Channel／Pointer；
- Pure annotation；
- 其他未證明功能。

SECTION

十四、刪除也是設計成果

Rust 的成熟不只是新增 Ownership，而是：

> DesignProgress
> =
> Addition
> +
> Removal
> +
> Unification

能夠刪除早期功能，避免所有實驗永久化石化。

SECTION

十五、Ownership 的收斂

Rust 最終把多個問題統一到 Ownership：

- Allocation／Drop；
- Move；

- Borrow；
- Alias；
- Mutation；
- Thread transfer；
- Resource lifetime。

但 Ownership 並非 Hoare 一人瞬間設計完整；型別研究者、Compiler 工程師和使用者持續改進其形式及人體工學。

SECTION

十六、Compiler 自舉與 LLVM

Rust 由 OCaml Bootstrap 逐步轉成 Rust Compiler，並使用 LLVM 作後端。

這帶來：

- Dogfooding；
- 多平台；
- Optimization；
- 巨大 Compiler codebase；
- 對 LLVM Infrastructure 的依賴。

SECTION

十七、Owner

Rust Book 將 Ownership 定義為管理程式記憶體的一組規則，由 Compiler 檢查，且 Ownership 本身不增加 Runtime 成本。[R3]

基本規則可概括為：

- 每個 Value 有一個 Owner；

- 同一時間只有一個 Owner；
- Owner 離開 Scope 時，Value 被 Drop。

SECTION

十八、Move

Move 使 Ownership 顯式轉移。

它阻止：

- Double free；
- 使用已轉移資源；
- 多個未協調 Owner。

SECTION

十九、Borrow

Reference 允許暫時使用而不取得 Ownership。

Borrowing 的核心分離是：

> Use
> ≠
> Own

SECTION

二十、共享與可變排他

一般規則：

多個 Immutable references
或
一個 Mutable reference

它限制危險組合：

> Alias
> +
> Mutation
> +
> Concurrency

SECTION

二十一、Lifetime

Lifetime 表達 Reference 不得活得比被參照資料更久。

Compiler 會推導大部分 Lifetime；複雜介面需要 Annotation。

SECTION

二十二、Drop／RAII

資源隨 Owner Scope 結束釋放：

- Memory；
- File；
- Lock；
- Socket；
- Transaction guard。

  Rust 繼承並型別化 Resource acquisition／destruction 傳統。

SECTION

二十三、早期視為兩個問題

Rust Book 回顧，團隊最初把 Memory safety 和 Concurrency 視為不同挑戰；後來發現 Ownership／Type system 可共同處理兩者。[R4]

SECTION

二十四、Send

Send 表示 Ownership 可以安全移到另一 Thread。

SECTION

二十五、Sync

Sync 表示對該型別的 Reference 可以安全跨 Thread 共享。

SECTION

二十六、Library concurrency

Thread、Channel、Mutex、Atomic 等多數機制在 Library，而 Send／Sync 及 Ownership 保證提供語言級邊界。

這使新的 Concurrency abstraction 仍可由 Crate 建立。

SECTION

二十七、Fearless 不是沒有錯誤

Rust 可阻止：

- Data race；
- Dangling reference；
- 未同步共享；
- 許多 Lifetime bug。

不能自動阻止：

- Deadlock；
- Livelock；
- Race condition 的所有高階形式；
- Distributed ordering；
- Logic error；
- Unsafe code 錯誤。

SECTION

二十八、為何必須有 Unsafe

系統需要：

- Raw pointer；
- FFI；
- Kernel；
- Allocator；
- Intrinsic；
- SIMD；
- Device；
- Runtime primitive。

若語言不能實作自身基礎，就不能作一般系統語言。

SECTION

二十九、Unsafe 不是關閉全部檢查

unsafe 只開放特定操作：

- Dereference raw pointer；
- Call unsafe function；
- Access mutable static 等。

  一般型別、Borrow 及語法規則仍存在。

SECTION

三十、證明責任轉移

Safe Rust 使用者依賴：

> PublicSafeAPI
> ⇒
> InvariantsHeld

Unsafe 作者必須證明：

- Pointer valid；
- Alias rule；
- Initialization；
- Lifetime；
- Thread safety；
- Layout；
- FFI contract。

SECTION

三十一、局部化

理想模式：

小型 Unsafe core
+
安全抽象 API
+
大量 Safe caller
這使低階風險攤銷。

SECTION

三十二、制度仍在補規格

Hoare 的十週年文章指出，Rust 1.0 當時甚至沒有充分說明 Unsafe 的正確性邊界；RustBelt、Miri、Formal model、Reference 和後來 Specification 工作才持續補足。[R1]

安全語言並非 1.0 即完成所有形式基礎。

SECTION

三十三、語言安全不足以被採用

系統語言還需要：

- Package；
- Dependency；
- Build；
- Test；
- Documentation；
- Publish；
- Version；
- Cross compile。

SECTION

三十四、Cargo

Cargo 逐步成為：

- Standard build tool；
- Package manager；
- Test runner；
- Documentation integration；
- crates.io client；
- Workspace manager。

  它降低「每個專案自造 Build system」的碎片化。

SECTION

三十五、Cargo 不是 Hoare 單人設計

Cargo 由後期 Rust 團隊和社群建立，並在 1.0 前後快速成熟。

Hoare 2025 年回顧指出，1.0 時 Cargo 仍只有約六個月歷史，遠未達今日成熟程度。[R1]

SECTION

三十六、整體安全供應鏈

Cargo 提高可用性，也建立新責任：

- Dependency trust；
- SemVer；
- Registry；

- Build script；
- Proc macro；
- Supply-chain security。

  語言安全不自動等於 Package 安全。

SECTION

三十七、1.0 的制度含義

Rust 1.0 不表示語言已完成全部功能，而表示：

- Stable API 承諾；
- Release channel；
- Backward compatibility；
- 可用於 Production；
- 演化程序更嚴格。

SECTION

三十八、Stability without stagnation

原則：

Stable 功能長期支持
+
新功能持續加入
不能用頻繁破壞換取進步。

SECTION

三十九、Feature gate

新功能先在 Nightly／Unstable：

- 收集實作經驗；
- 修改；
- 可能刪除；
- 不形成 Stable 契約。

  只有經 Stabilization 才進入一般使用者基線。

SECTION

四十、Channel

- Nightly；
- Beta；
- Stable。

  它將實驗、預覽和公共承諾分開。

SECTION

四十一、RFC 的起因

RFC 0002 說明，早期自由加入功能適合快速發展，但成熟平台需要更有紀律、一致和受控的路徑。[R6]

SECTION

四十二、RFC 要求

重大改變需描述：

- Motivation；
- Detailed design；
- Drawbacks；

- Alternatives；
- Unresolved questions；
- Compatibility；
- Implementation path。

SECTION

四十三、不是直接多數票

官方 Governance 表示，重大決策從 RFC 開始，任何人可參與討論，目標是建立對 Tradeoff 的共同理解；最終由負責團隊依共識程序推進。[R7]

SECTION

四十四、Team 聯邦

現代 Rust 具有：

- Leadership Council；
- Language；
- Compiler；
- Library；
- Dev tools；
- Infrastructure；
- Moderation；
- 其他專門 Team。

不同專業擁有不同決策權，避免單一 Core team 處理所有細節。

SECTION

四十五、聯邦代價

- 邊界重疊；
- Cross-team 協調；
- RFC 延遲；
- Review 負荷；
- 權責難懂；
- 志願者 Burnout。

  制度複雜度是技術成熟的成本。

SECTION

四十六、為何需要 Edition

穩定承諾阻止直接：

- 新增會破壞舊 Identifier 的 Keyword；
- 改變某些語法；
- 更正部分預設。

  Edition 允許 Crate Opt-in。

SECTION

四十七、不分裂生態

最重要規則：

> Crate_(edition A)
> ↔
> Crate_(edition B)

必須無縫互操作。[R8]

因此 Edition 不是 Python 2／3 式整個生態 Fork。

SECTION

四十八、遷移工具

Cargo／Lint 可自動修改大量表面語法。

但工具不保證所有 Macro、Generated code 或語義情況完全自動。

SECTION

四十九、Skin-deep 限制

因跨 Edition 互操作要求，Edition 通常只能處理表面及局部語義，不能重建完全不相容的核心型別世界。

SECTION

五十、Mozilla 退出後的風險

2020 年 Mozilla 組織變動顯示：

- 單一公司資助不穩定；
- Maintainer employment 會突然改變；
- Trademark、Registry、CI、法律及財務需要獨立機構。

SECTION

五十一、Rust Foundation

Foundation 提供：

- Legal；
- Trademark；

- Infrastructure；
- Grant；
- Funding；
- Staff；
- crates.io 支援；
- 多公司參與。

它不直接取代 Rust Project 的全部技術治理。

SECTION

五十二、Project 與 Foundation 分離

理想分工：

Rust Project：語言、Compiler、Library、Technical governance
Rust Foundation：法律、財務、基礎設施及資助
實際仍需持續協調權力和責任。

SECTION

五十三、Leadership Council

現行官方治理由 Leadership Council 協調整體成功，成員來自 Top-level teams；它取代過去部分 Core team 結構。[R7]

這再次證明 Rust Governance 仍在演化，而非 2015 即固定。

SECTION

五十四、拒絕程式只是第一步

Borrow checker 若只說「錯」，使用者無法建立正確心智模型。

Rust 因此投入：

- Error code；
- Source span；
- Borrow path；
- Help；
- Suggestion；
- Explain；
- rust-analyzer。

SECTION

五十五、Compiler 是教學者

因語言把更多錯誤提前，Compiler 必須承擔：

> Detection
> +
> Localization
> +
> Explanation
> +
> RepairGuidance

否則安全成本全轉成挫折。

SECTION

五十六、診斷仍有極限

Trait、Lifetime、Async 和 Generic error 仍可能：

- 很長；
- 非局部；

- 涉及推導；
- 難以理解。

好的錯誤訊息是長期工程，不是一次設計完成。

SECTION

五十七、個人原型期

問題：基礎設施語言難以安全使用
策略：綜合舊研究語言概念建立 Rust

SECTION

五十八、Mozilla／Servo 期

問題：原型不足以建造真實 Browser
策略：Full-time team、LLVM、Servo dogfooding

SECTION

五十九、Pre-1.0 重寫期

問題：早期功能過多且模型未收斂
策略：刪除、統一、Ownership、Compiler 自舉

SECTION

六十、1.0 契約期

問題：使用者無法依賴快速變動語言
策略：Stable channel、Compatibility、Cargo

SECTION

六十一、RFC／Team 期

問題：單一創始者／自由流程不能治理成熟平台
策略：RFC、Team、Consensus、Feature gate

SECTION

六十二、Edition 期

問題：穩定承諾阻礙局部修正
策略：Opt-in、跨 Edition 互操作、Cargo migration

SECTION

六十三、Foundation／Infrastructure 期

問題：企業資助、法律及維護接班不可依單一公司
策略：Foundation、Leadership Council、多元投入

SECTION

六十四、Graydon 原始問題 framing

> 如何把功能語言和研究型別系統中的安全能力，帶入真正需要控制 Layout、效能及並發的基礎設施程式？

SECTION

六十五、共同體後期 framing

> 如何讓語言的安全、工具、相容和治理保證，在數十年及數千名貢獻者中仍可持續？

SECTION

六十六、價值優先序

V_(Rust)
≈
(
Reliability,
Performance,
MemorySafety,
ConcurrencySafety,
Control,

SECTION

六十七、核心—擴張偏好

核心：

- Ownership；
- Borrow；
- Trait；
- Enum／Pattern；
- Unsafe boundary。

擴張：

- Cargo；
- Crate；
- Macro；
- Async runtime；
- Tool；
- Edition；
- Library。

SECTION

六十八、顯式—推導偏好

明示：

- Ownership transfer；
- Mutation；

- Unsafe；
- Trait bound；
- Error type。

  推導：

- Lifetime 多數情況；
- Generic；
- Borrow region；
- Auto trait；
- Type inference。

SECTION

六十九、效率—可讀性偏好

不使用 GC 作一般基礎，保留：

- Stack／Heap；
- Layout；
- Allocation；
- Zero-cost abstraction；
- Static dispatch。

  同時投入高階 Iterator、Pattern、Trait 和 Tooling。

SECTION

七十、安全—自由偏好

Safe Rust 提供強保證；Unsafe Rust 保留系統底層能力。

真正判準不是有無 Unsafe，而是：

- Unsafe 是否局部；
- Invariant 是否可說明；
- Safe API 是否可靠；
- Tool／Formal model 是否能審查。

SECTION

七十一、相容性偏好

Pre-1.0：可激進破壞。

Post-1.0：Stable without stagnation。

Edition：局部 Opt-in 修正。

生態：Crater／Regression test。

SECTION

七十二、治理偏好

由：

個人原型
轉向：

企業團隊
再轉向：

RFC＋聯邦 Team＋Foundation＋全球共同體

SECTION

七十三、Borrow checker 會拒絕安全程式

Compiler 只能接受能被其分析證明的程式。

使用者有時需：

- 重構；
- Clone；
- Index；
- Interior mutability；
- Unsafe；
- 不同資料結構。

SECTION

七十四、Memory safety 不等於完整安全

Rust 不能自動保證：

- Logic；
- Authentication；
- Cryptographic correctness；
- Deadlock；
- Resource exhaustion；
- Supply chain；
- Unsafe FFI；
- Hardware failure。

SECTION

七十五、Unsafe 生態是 Trusted Computing Base

Standard Library、Allocator、OS binding、Crate 中的 Unsafe 共同構成 TCB。

Safe caller 依賴它們正確。

SECTION

七十六、治理本身會耗盡人

RFC、Review、Release、Moderation、Documentation 和 Infra 需要長期人力。

Hoare 明確提醒貢獻者 Burnout 和接班是未來核心問題。[R1]

SECTION

七十七、Rust 不是 Hoare 原型的線性實現

今日 Rust 的許多最核心形式和工具在 Hoare 離開後形成。

不能用今日結果回寫創始者在 2006 年已完整預見。

SECTION

七十八、共同體也不是單一人格

Rust 社群包含：

- 企業；
- 志願者；
- Embedded；
- Web；
- Academic；
- Safety-critical；
- OS；
- Tool。

「Community decided」需要追問是哪個 Team、RFC、投票、Consensus 或 Foundation 行動。

SECTION

七十九、制度成熟不表示沒有治理危機

Rust 歷史曾經歷 Team 衝突、Leadership 重組、Moderation 問題和資助轉變。

本文分析制度能力，不將其理想化成無摩擦共同體。

時期	問題	決策	複雜度去向	主體
2006–09	安全基礎設施語言缺口	Rust 私人原型	Compiler／設計者	Hoare
2009+	原型缺乏資源與真實測試	Mozilla 投資	團隊／公司	Mozilla＋早期團隊
2012+	語言需驗證瀏覽器規模	Servo	使用者／Compiler	Rust／Servo 團隊
2012–15	核心模型過多	刪除 GC、Typestate 等，收斂 Ownership	Type system	多位設計者
2014	自由變更不適合成熟平台	RFC 0002	公開程序	Core community
2015	使用者需要穩定	Rust 1.0	Compatibility／Release infra	Rust Team
2015+	新功能與舊碼衝突	Channel、Crater、Stabilization	Tool／Review	多 Team
2018+	需要局部不相容修正	Edition	Cargo／Migration	Edition WG
2021+	資助及法律不可依單一公司	Rust Foundation	Institution	多公司／Project
當代	Core team 結構需擴展	Leadership Council	聯邦治理	Top-level Teams

SECTION

八十、Graydon Hoare 原型

- 安全基礎設施問題發現者；
- 研究語言綜合型原型設計者；
- 願意讓原型被重寫的創始者；
- 多主體歷史的自覺修正者。

SECTION

八十一、Rust 共同體原型

- 型別化責任配置共同體；
- 穩定而不停滯的制度設計者；
- RFC／Team 聯邦治理共同體；
- 生態回歸測試和工具驅動演化者；
- 基礎設施型語言維護制度。

SECTION

八十二、不適合的簡單標籤

不應只稱：

Graydon Hoare 單人發明 Borrow checker
Mozilla 單獨創造 Rust
無 GC 的 C++ 替代品
絕對安全語言
社群民主直接投票語言
較精確的描述是：

> 一個由個人原型提出安全系統語言方向，再由研究者、企業、志願者、工具作者和治理制度反覆刪除、重寫、穩定及維護的公共基礎設施。

SECTION

八十三、最重要的技術連續性

從原型到今日：

> 【可靠性不能只依靠程式設計者記住規則】

應把可檢查責任放入語言和 Compiler。

SECTION

八十四、最重要的技術不連續性

今日 Ownership／Trait／Cargo／Edition Rust 與早期 Typestate／Object／GC Rust 差異巨大。

成功來自方向延續，不是功能保留。

SECTION

八十五、最重要的制度連續性

Rust 不斷把私人或小組責任轉成公共基礎：

- Compiler；
- RFC；
- Test；
- Cargo；
- Book；
- Team；
- Foundation；
- Spec。

SECTION

八十六、最重要的創始者修正

Hoare 的後期歷史觀把「我創造了一個 Idea」改寫為：

> 很多人共同投資、維護和接班，才使一個早期 Idea 成為可靠 Infrastructure。

Graydon Hoare 的歷史功勞既不能被共同體敘事抹除，也不能被創始者神話無限放大。

他做出的關鍵貢獻是：

- 辨識基礎設施語言的可靠性缺口；
- 拒絕把安全與系統效能視為必然對立；
- 從舊研究語言吸收被主流遺忘的機制；
- 建立足以吸引 Mozilla 投資和團隊加入的原型；
- 接受原型被大幅改造；
- 在後期主動把成功歸於廣泛共同體和制度。

Rust 共同體的關鍵貢獻是：

- 把 Ownership、Borrowing 和 Trait 收斂成可實用模型；
- 以 Servo 和真實系統驗證；
- 建立 Cargo 和開發者體驗；
- 在 1.0 建立相容契約；
- 以 RFC 和 Team 分配設計權；
- 以 Edition 修正局部歷史而不分裂生態；
- 以 Unsafe 邊界保留底層能力；
- 以 Crater、CI 和 Release pipeline 檢驗變更；
- 以 Foundation 和 Leadership Council 支持長期接班；
- 以教育、錯誤訊息和工具把安全成本轉成可學習介面。

本文的 PLDST 雙層判定為：

> 【Hoare: Safe-Infrastructure Prototype Designer】

以及：

> 【Rust Community: Institutional Reliability Engineering Collective】

Rust 的核心優勢是：

- Memory safety 與多數 Data-race 防止提前至 Compiler；
- 不依一般 GC 仍保留系統效能；
- Unsafe 提供受限逃生；
- Cargo、Diagnostics 和 Documentation 改善採用；
- RFC／Edition／Release 建立可依賴演化；
- 多主體制度降低創始者永久依賴。

  其核心代價是：

- Ownership 學習門檻；
- Compiler 拒絕部分安全程式；
- Trait／Lifetime／Async 交互複雜；
- Unsafe TCB 仍需高專業審計；
- 工具和治理基礎設施成本巨大；
- Contributor burnout、資助及接班是長期風險。

  最終原則為：

> 【可靠性必須同時被編碼進語言、工具與制度】

只建立 Ownership 不足；只建立友善 Compiler 不足；只建立 RFC 也不足。Rust 的真正成果是三層共同成立：

> 【TechnicalSafety
> +
> EvolutionSafety
> +
> InstitutionalContinuity】

這也使 Rust 成為 PLDST 中最重要的反創始者偏誤案例之一：

> 一門語言可以由一個人開始，但當它真正成為基礎設施後，最值得分析的設計者已不再只是個人，而是能讓技術在創始者離開後仍持續修正、穩定和接班的共同體制度。

初始設計者：Graydon Hoare
共同體：Rust Project／Mozilla／Rust Foundation／全球貢獻者
主要語言／制度：Rust、Cargo、RFC、Edition、Teams、Foundation
核心時期：2006–至今
主要問題：基礎設施效率、安全和並發可靠性分裂
主要策略：Ownership、Borrow、Trait、Unsafe boundary、Compiler evidence
複雜度去向：Type system、Compiler、Tool、Governance、Institution
責任去向：安全規則交給 Compiler，低階證明交給 Unsafe，演化交給制度
主要保護對象：基礎設施開發者、使用者和長期生態
主要限制：學習、Unsafe TCB、制度成本、Burnout、規格持續補完
歸因信心：高

[R1] Graydon Hoare, “10 Years of Stable Rust: An Infrastructure Story,” Rust Foundation, 2025.

— 多主體基礎設施敘事、Baby Rust、Mozilla 投資、1.0 前重寫、Cargo／Edition／RFC／貢獻者和接班。

[R2] Graydon Hoare, Rust historical talks, prehistory archive and interviews.

— 2006 個人原型、早期語言影響、OCaml Compiler 和初期目標；回顧性材料需與 Repository 交叉校對。

[R3] Steve Klabnik, Carol Nichols, Chris Krycho and Rust Community, The Rust Programming Language, Ownership chapters.

— Ownership、Move、Borrow、Lifetime 和無一般 GC 的 Memory management。

[R4] The Rust Programming Language, Concurrency chapters.

— Ownership 與 Concurrency 的統一、Send／Sync、Thread、Channel 和 Shared state。

[R5] Rust Reference and Rustonomicon, Unsafe Rust.

— Unsafe operation、Safety invariant、Raw pointer、FFI 和 Safe abstraction 邊界。

[R6] Rust RFC 0002, “RFC Process,” 2014.

— 從早期自由演化轉向一致、受控及共識式重大功能流程。

[R7] Rust official Governance page and Rust Forge.

— Leadership Council、Top-level Teams、RFC 及現代聯邦治理。

[R8] Rust Edition Guide and Rust 2021 Edition plan.

— Stability without stagnation、Opt-in Edition、跨 Edition Crate 互操作及自動遷移。

[R9] Rust 1.0 release announcement and official release history.

— 2015 年 5 月穩定發布、Channel 及穩定承諾。

[R10] Rust Foundation official history and governance materials.

— 2021 後的法律、財務、基礎設施和企業多元支持。

[R11] Rust RFC repository, Team repositories, Crater and Release infrastructure documents.

— 生態回歸、Stabilization、Release train 和多主體演化。

[R12] RustBelt, Miri and Formal Specification materials.

— Unsafe 語義、形式安全與規格持續補完；屬共同體及研究機構成果，不歸於 Hoare 個人。

[T-P] Personal prototype phase
[T-M] Mozilla／Servo phase
[T-R] Pre-1.0 rewrite phase
[T-S] Stable-contract phase
[T-F] RFC／federated-team phase
[T-E] Edition phase
[T-I] Foundation／infrastructure phase
[S-O] Ownership responsibility
[S-B] Borrowed access
[S-U] Unsafe boundary
[S-C] Compiler-enforced safety
[S-R] RFC governance
[S-E] Edition compatibility
[S-I] Institutional continuity

附錄 D　第二輪史實、技術與治理校對紀錄
D.1 個人原型與 Mozilla 投資
第二輪重新核對 Graydon Hoare 2025 年 Rust 穩定十週年回顧：
- Hoare 於 2006 年開始私人原型；

- Mozilla 於 2009 年開始投資；
- 投資使原本的單人專案真正形成 Team；
- Servo 自 2012 年起成為重要壓力測試；
- Hoare 把自己的初始 Compiler 描述為數萬行、接近無資助個人專案可負擔的上限；
- 2006–2009 的 Baby Rust 幾乎沒有成熟 Type checker、效能很差且平台範圍極窄；
- 到 2015 年 1.0 時已被多人徹底重寫。

本文因此沒有使用「2006 年 Rust 已具有今日 Ownership／Borrow checker」的回寫敘事。

D.2 Hoare 離開與 1.0 的時間邊界

歷史資料支持：

- Hoare 約於 2013 年離開 Rust 的日常核心開發；
- RFC Process 於 2014 年建立；
- Rust 1.0 於 2015 年 5 月 15 日發布；
- Cargo、Borrow checker 人體工學、Compiler infrastructure、Library 及大量語言細節在 Hoare 離開前後由多人持續完成。

本文將 Hoare 定位為 Initial author／Prototype designer，而不稱其為 1.0 全部機制的唯一設計者。

D.3 Ownership 的保證範圍

第二輪直接核對 Rust Book：

- Ownership 是由 Compiler 檢查的 Memory-management rule set；
- 一般 Ownership 機制本身不增加 Runtime overhead；
- Borrowing 分離使用權和 Ownership；
- `&mut` 的排他性限制 Alias＋Mutation；
- Lifetime 用於確保 Reference 不超過被參照資料的有效期。

這些規則不表示每個資源永遠只有一個 Conceptual stakeholder；`Rc`、`Arc`、Interior mutability、Mutex 等型別可建立受控共享。

D.4 Concurrency 保證的精確邊界

Rust Book 說明：

- Rust 團隊後來發現 Ownership 和 Type system 可同時支援 Memory safety 及許多 Concurrency safety；
- `Send` 表示型別值可安全跨 Thread 轉移；
- `Sync` 表示 Reference 可安全跨 Thread 共享；
- Message passing、Shared state、Mutex 等方案都可以使用；
- Data-race freedom 不等於不存在 Deadlock、Livelock、Starvation 或所有高階 Race condition。

本文因此使用「多數 Data race 在 Safe Rust 中被型別規則阻止」，而不寫成所有並發錯誤被消除。

D.5 Unsafe 的兩種證明義務

第二輪直接核對 2026 年 Rust Reference：

- `unsafe fn`、`unsafe trait`、`unsafe static` 等可**建立額外安全條件**；
- `unsafe {}`、`unsafe impl`、`unsafe extern` 等可表示程式設計者聲稱已**履行安全條件**；
- Unsafe block 只開放特定受限操作，不會取消全部型別、Ownership 或語法規則；
- Undefined behavior 在 Unsafe code 中仍是錯誤；
- 安全包裝必須確保 Safe caller 無法觸發 UB；
- Rust Reference 仍提醒 Unsafe 的完整形式語義模型持續發展。

本文因此把 Unsafe 定位為 Proof-obligation boundary，而不是「關閉安全模式」。

D.6 Rust 2024 的 Unsafe 明示

現行 Reference 規定：

- 2024 Edition 的 External block 必須明示 `unsafe extern`；
- 某些影響 Symbol／Section 的 Attribute 需以 `#[unsafe(...)]` 標記；
- 此演化使 Safety obligation 更靠近實際宣告和外部邊界。

這是 Rust 共同體在 Stable／Edition 框架內持續改善 Unsafe 可審計性的例子，不是 Hoare 初始原型的直接功能。

D.7 RFC 0002 的制度目的

第二輪直接核對 RFC 0002：

- 2014 年建立 RFC Process；
- 目標是為重大 Language／Standard Library 變更提供一致、受控的進入路徑；
- 早期自由加入功能適合探索，但成熟平台需要更強自律；
- RFC 合併表示設計成為 Active、可實作及繼續驗證，不保證自動穩定；
- 最終仍需實作、Tracking issue、Feature gate、Review 和 Stabilization。

本文沒有把 RFC 寫成直接多數投票或提案合併即永久規格。

D.8 Edition 的相容邊界

第二輪核對 Edition Guide：

- Edition 是每個 Crate 的 Opt-in 選擇；
- 不同 Edition Crate 必須無縫互操作；
- 遷移通常由 Cargo／Lint 大量自動化；
- 為維持跨 Edition 互操作，變更一般較「Skin-deep」；
- 自動遷移不保證覆蓋所有 Macro、Generated code 或語義情況。

本文因此沒有把 Edition 描述成 Python 2／3 式生態分裂，也沒有宣稱它能承載任何核心不相容重設。

D.9 2025 十週年數據的時間邊界

Hoare 的 2025 年文章提供當時快照，包括：

- 1.0 後數十萬次變更；
- 約 6,700 名 Repository Contributor；
- 數千份 RFC；
- 多個 Edition；
- 對龐大 Public crate 生態進行 Release regression testing。

這些數字會繼續變動。

本文只用它們證明 Rust 已成為大規模公共基礎設施，不把數字當成 2026 年永久固定值。

D.10 現行 Rust 版本與治理

截至 2026 年 7 月 30 日：

- Rust 官方網站列出的穩定版本為 1.97.1；
- 官方 Governance 頁列出 Leadership Council、Compiler、Dev tools、Infrastructure、Language、Library、Moderation 等 Top-level teams；
- Rust Project 由多 Team 和多貢獻者開發；
- Rust Foundation 提供法律、財務、商標、基礎設施和資助支持，但不等同所有技術決策機關。

版本資訊是時間敏感快照，不作長期人物風格核心依據。

D.11 規格的持續補完

第二輪核對官方 Learn／Reference 及 Hoare 十週年回顧：
- Rust Book 是教學文本；
- Rust Reference 比 Book 詳細，但官方仍提示它不完全等同最終形式規格；
- Unsafe 的正確性邊界由 Reference、Rustonomicon、Miri、RustBelt、Formal specification 等多條工作持續補完；
- 1.0 的 Production stability 不表示完整形式語義於 2015 年即全部完成。
本文因此把 Rust 的可靠性理解為持續演進的技術—制度組合，而不是一次性證明完畢。

D.12 Foundation 與 Project 的權責
Rust Foundation：
- 擁有並保護 Rust／Cargo 商標；
- 支援 Crates.io、整合測試等基礎設施；
- 提供資助和法律組織。
Rust Project：
- 透過 Leadership Council 和各技術 Team 處理語言、Compiler、Library、Tool 及政策。
兩者可能互相依賴，但不應被寫成單一中央法人完全控制語言。

D.13 創始者歸因的最終校準
本文的歸因結論與 Hoare 自己 2025 年的修正一致：

Hoare：

問題框架、初始原型、早期方向與團隊形成

Mozilla／Servo：

資助、專職工程、真實壓力測試

Rust 共同體：

Ownership 收斂、Compiler、Cargo、RFC、1.0、Edition、

Diagnostics、Foundation 與長期演化

這是概括層次；每項具體功能仍需在個別研究中查核作者、Reviewer、實作者及決策 Team。

D.14 PLDST 雙層原型邊界
下列名稱是本文分析原型：

Hoare：安全基礎設施原型設計者

Rust 共同體：制度化可靠性工程共同體

它們不是官方頭銜。
「Institutional reliability」表示 Memory safety、Compatibility、Release、Governance、Tooling 和 Funding 共同支持可靠性，不表示 Rust Project 的所有制度已無爭議或已達最終形態。