# Yukihiro Matsumoto：程式設計者幸福、語言自然性與社群信任

Yukihiro Matsumoto: Programmer Happiness, Linguistic Naturalness, and Community Trust

v1.0 / 2026-07-30 / Neo.K / 公開版／第三部設計師個案正式研究

https://thisoneisneok.com/html/papers/pldst-018.html

英文名稱： Yukihiro Matsumoto: Programmer Happiness, Linguistic Naturalness, and Community Trust

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-018

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／第三部設計師個案正式研究

SECTION

## 摘要

Yukihiro “Matz” Matsumoto 經常被描述為 Ruby 的創造者，以及「為了讓程式設計者幸福」而設計語言的人。這個描述捕捉了 Ruby 最著名的價值主張，卻也容易產生四種誤讀：

- 把「幸福」理解成只追求語法漂亮或短；
- 把「自然」理解成模仿英文句子；
- 把「最小驚訝原則」理解成滿足每個人的既有直覺；
- 把 Matz 的最終裁決理解成沒有實作、社群與相容性約束的任意個人品味。

Ruby 的起點是 Matz 在 1993 年尋找一門真正物件導向、又適合日常腳本的語言。官方 FAQ 記錄，他熟悉 Perl 與 Python，卻不滿意前者的語言感及後者當時在他眼中的物件模型，希望建立容易使用的物件導向腳本語言。[R1] Ruby 最終綜合：

- Perl 的文字處理與實用腳本能力；
- Smalltalk 的一致物件模型與訊息式方法；
- Lisp 的函數、Block 和元程式傳統；
- Eiffel 的物件與設計觀念；
- Ada 等語言的清楚結構；
- Unix 系統呼叫及現實工作能力。[R2][R3]

  Matz 的設計重點不是形式最小，而是人使用語言時的感受。2003 年第一手訪談中，他明確把語言評價分成「能做什麼」與「使用時感覺如何」，並說 Ruby 主要強調後者；他也說，自己不是要為每個人設計完美語言，而是先做一門自己樂於使用的語言。[R3]

  這種人本設計常被濃縮成「Principle of Least Surprise」。然而，Matz 自己提醒，該說法常被誤解：不同背景的人會對不同設計感到驚訝，Ruby 無法滿足所有人的既有預期；他真正最小化的是自己在充分理解 Ruby 之後的挫折與不自然感。[R3]

  本文將 Matz 的設計生涯分成六個相位：

- 個人工具尋找期：Perl、Python、Smalltalk、Lisp 等語言的不滿與綜合；
- Ruby 核心形成期：一切皆物件、Block、Iterator、Mixin、開放類別與可讀腳本；
- 日本社群孵化期：Ruby 0.x／1.x、日文討論、小型核心與 Matz 最終裁決；
- 全球化與 Rails 放大期：Ruby 由個人喜好語言成為 Web 生產力平台；
- 相容與性能再平衡期：Ruby 1.9／2.x、Keyword argument、Encoding、VM、JIT；
- Ruby 3／4 與後期制度期：Performance、Concurrency、Typing、RBS、Ractor、JIT／ZJIT，以及仍由 Matz 保留最終語言方向的多團隊治理。

  本文核心判斷為：

> 【Matz 所謂「幸福」不是即時愉悅，
>
> 而是讓程式設計者把注意力留給問題、創造與表達。】

其深層配置可表示為：

> 【自然表達
> +
> 高彈性物件模型
> +
> 多種可讀慣用法
> +
> 實用腳本能力
> +
> 集中品味與社群信任】

Ruby 的官方介紹把其設計描述為「Careful balance」，並引用 Matz 的說法：Ruby 追求 Natural 而非 Simple；它表面簡單，內部像人體一樣複雜。[R2] 這是理解 Ruby 的關鍵。Matz 不把完全正交性視為最高美德；他認為正交特徵任意組合可能讓使用者必須在腦中模擬 Compiler，因而偏好「和諧」——功能可以不是理論上最一般，但要在共同語言感中相處自然。[R3]

這種風格的力量與代價也來自同一來源。Ruby 可以像可執行的偽程式碼，Block、Iterator、Mixin、Duck typing 和 Metaprogramming 能快速建立 DSL 與 Framework；但多種寫法、隱式轉換、開放類別和運行期元程式會增加大型系統的工具、安全、追蹤和性能負擔。Matz 的品味能維持整體氣質，卻也讓治理高度依賴社群對個人裁決的信任。

因此，Matz 不應只被分類為「讓程式設計變快樂的人」。更精確的判定是：

> 他把程式設計者的認知感受提升為語言設計的一級成本，並以長期個人品味協調彈性、實用、自然性、相容性與性能之間的衝突。

關鍵詞： Yukihiro Matsumoto、Matz、Ruby、程式設計者幸福、自然性、最小驚訝、和諧、物件導向、BDFL、社群信任、PLDST

SECTION

## 一、本文研究範圍

本文主要分析：

- 1993 年 Ruby 問題形成；
- Ruby 0.x／1.x 的核心；
- Matz 的語言哲學訪談；
- Ruby 1.8／1.9 的重大演化；
- Rails 對 Ruby 全球化的放大；
- Ruby 2.x／3.x 的性能、並行與型別工具；
- Ruby Core 的提案和 Matz 最終裁決模式。

本文不把下列成果全部歸於 Matz：

- CRuby 所有實作；
- YARV；
- RubyGems；
- Bundler；
- Rails；
- RSpec；
- JRuby；
- TruffleRuby；
- YJIT；
- RBS 所有設計；
- Ruby 社群文化的全部形成。

SECTION

## 二、Ruby 的創始權重

Matz 對下列事項具有高度直接權重：

- 原始問題設定；
- Ruby 名稱和第一實作；
- 物件模型與 Block 核心方向；
- 語法和語言感；
- 長期接受、拒絕和修改功能的最終權威；
- Ruby 3 的宏觀方向；
- 對幸福、自然和和諧的明確哲學。

  因此稱其為 Ruby 創造者和主要設計者合理。

SECTION

## 三、Ruby 是多團隊成果

Ruby 長期演化涉及：

- Keiju Ishitsuka 等早期日本 Rubyist；
- Ruby Core committers；
- Koichi Sasada：YARV／VM；
- Nobuyoshi Nakada；
- Yusuke Endoh；
- Shugo Maeda；
- Akira Tanaka；
- Tanaka Akira；
- RubyGems、Bundler、Rails、RSpec 團隊；

- JRuby、TruffleRuby；
- Shopify YJIT／ZJIT 團隊；
- RBS、TypeProf 等工具作者。

Ruby 官方首頁也明確把豐富生態歸功於社群，並引用 Matz 表示自己「讓程式設計者幸福」的願望是由社群以他單獨無法完成的方式實現。[R4]

SECTION

## 四、Ruby 的誕生問題

官方 FAQ 記錄 Ruby「出生」於 1993 年 2 月 24 日的討論；Matz 想要的是：

- 物件導向；
- 容易使用；
- 適合腳本；
- 不像當時 Perl 那樣令他不滿；
- 比他當時理解的 Python 更一致地物件導向。[R1]

SECTION

## 五、不是從單一語言繼承

官方 Ruby 介紹將影響列為：

- Perl；
- Smalltalk；
- Eiffel；
- Ada；
- Lisp。[R2]

較完整分析可分成：

Perl：文字、正規表示式、Unix 實用性
Smalltalk：物件、方法、動態性
Lisp：Block、高階函數、元程式
Eiffel：物件設計與一致性
Ada：結構與可讀性
Ruby 是綜合，不是其中任何一門的表面翻版。

SECTION

## 六、個人品味作為原型測試

Matz 在 Artima 訪談中說，他首先嘗試做一門對自己而言接近完美的語言，而不是假設可同時滿足所有人。[R3]

這種方法的優點：

- 具有一致感；
- 快速原型；
- 不被委員會平均化；
- 設計者自己長期使用。

風險：

- 個人背景可能不代表其他領域；
- 「自然」容易變成不可反駁直覺；
- 社群規模擴大後需要更多證據。

SECTION

## 七、幸福不是娛樂化

Matz 所說的幸福包括：

- 快速完成工作；

- 專注問題本身；
- 程式接近偽程式碼；
- 減少機械操作；
- 使用語言時有節奏與美感；
- 保留創造感。[R3]

  可表示為：

> H_(programmer)
> =
> Focus
> +
> Flow
> +
> Expressiveness
> +
> Feedback
> -
> Boilerplate
> -
> Friction
> -
> Confusion

SECTION

## 八、幸福不是所有人的即時偏好總和

若每位使用者都要求語言符合原語言直覺：

- Python 使用者；
- Perl 使用者；

- Java 使用者；
- C++ 使用者；

  會得到互相矛盾的要求。

  因此 Matz 的「最小驚訝」不是民主平均，而是：

> 熟悉 Ruby 整體品味後，局部行為應與這套品味相容。

SECTION

## 九、自然而非簡單

Ruby 官方頁面引用：

trying to make Ruby natural, not simple
以及：

simple in appearance, complex inside
這表示：

- 使用者表面自然；
- Interpreter 和語義可以複雜；
- 不追求最少語法規則；
- 允許多個方便入口；
- 機器承擔更多轉譯工作。[R2]

SECTION

## 十、自然性的社會來源

「自然」來自：

- 日常程式設計經驗；
- Unix 腳本；

- 物件訊息；
- 英文式方法名；
- 數學運算；
- Ruby 社群慣例；
- Matz 個人語言感。

  它不是普遍心理定律。

SECTION

## 十一、正交性的吸引力

完全正交意味：

- 每項功能獨立；
- 任意組合；
- 少例外；
- Compiler 規則一般。

  這是許多極簡設計的理想。

SECTION

## 十二、Matz 的批判

Matz 在 2003 年訪談中說，Consistency 和 Orthogonality 是設計工具，不是最高目的；兩個各自合理的功能任意組合，可能造成普通讀者必須模擬 Compiler，形成複雜爆炸。[R3]

因此他偏好：

> Harmony
> >
> MaximalOrthogonality

SECTION

## 十三、和諧的含義

和諧要求：

- 功能組合符合共同語言感；
- 常見情況流暢；
- 不需要記住大量交互；
- 可以拒絕理論上合理但整體不自然的功能；
- 局部例外可能比完全一般更好讀。

SECTION

## 十四、和諧的風險

因為和諧難以形式化，可能造成：

- 裁決依賴 Matz；
- 新貢獻者難預測；
- 相似提案獲得不同結果；
- 文件只能事後解釋；
- 社群把品味人格化。

SECTION

## 十五、一切皆物件

Ruby 將：

- 數字；

- 字串；
- 類別；
- Module；
- 方法結果；
- nil；

  納入一致物件世界。

  這提高：

- 方法組合；
- 反射；
- 元程式；
- API 一致感。

SECTION

## 十六、Block 與 Iterator

Ruby 的 Block 讓：

```
3.times do
  ...
end
```

能把控制、資源和集合遍歷交給方法。

其作用包括：

- 高階函數；
- Iterator；
- Resource scope；
- DSL；

- Callback；
- Internal iteration。

  它使程式更接近「描述要做什麼」。

SECTION

## 十七、Mixin

Ruby 以 Module 支援 Mixin，避免只用單一繼承階層承擔重用。

優勢：

- 行為組合；
- Duck typing；
- Concern；
- 開放擴展。

風險：

- 方法來源不透明；
- Include order；
- Monkey patch；
- 名稱衝突；
- 大型系統追蹤。

SECTION

## 十八、開放類別與元程式

Ruby 允許：

- 重新開啟 Class；

- Define method；
- Method missing；
- Reflection；
- DSL；
- 運行期生成。

這使 Rails、RSpec 等 Framework 得以建立高度自然的領域語言。

同時增加：

- 靜態工具困難；
- 行為非局部；
- API 衝突；
- 安全審計；
- 性能最佳化難度。

SECTION

## 十九、多種寫法

Ruby 常允許：

- 有無括號；
- 多種 String；
- Block syntax；
- Modifier form；
- Symbol／String；
- Enumerable 慣用法；

- Explicit／Implicit receiver。

  這可使程式依情境自然，也可能降低跨團隊一致性。

  因此 Ruby 不是 Python 式「一個明顯做法」設計。

SECTION

## 二十、早期社群

Ruby 先在日本形成：

- 郵件列表；
- 使用者；
- 書籍；
- Core contributors；
- 實際腳本；
- Matz 最終決策。

  小型社群讓：

> Proposal
> →
> Discussion
> →
> MatzDecision
> →
> Implementation

循環較短。

SECTION

## 二十一、語言設計權

Ruby Issue Tracker 的設計流程討論中，曾有核心開發者直接表述：

Ruby is matz'
Matz 對名為 Ruby 的語言保有最終覆寫權；社群可以說服他，或建立 Fork。[R5]

這是參與者的治理陳述，不是中立憲法，但準確顯示 Ruby 長期制度的中心性。

SECTION

## 二十二、BDFL 與社群信任

Matz 的權威來自：

- 創始設計；
- 長期一致使用；
- 技術參與；
- 對社群友善；
- 接受說服；
- 願意調整過渡；
- 象徵性人格。

  這種模式依賴：

> Authority
> +
> Benevolence
> +
> TasteConsistency
> +
> CommunityTrust

SECTION

## 二十三、信任不等於程序透明

Ruby 有：

- Issue tracker；
- Developer meeting；
- Mailing list；
- Commit；
- Release note。

但外部參與者仍可能難以知道：

- 哪項提案接近接受；
- Matz 如何衡量和諧；
- 何時需要原型；
- 最終拒絕理由；
- 語言方向優先序。

SECTION

## 二十四、Rails 不是 Ruby 本身

Ruby on Rails 由 David Heinemeier Hansson 等社群建立。

Rails 使用 Ruby 的：

- Block；
- Metaprogramming；
- Open class；
- Symbol；
- DSL；

- Convention；
- Reflection。

其成功放大 Ruby，但不能歸為 Matz 的直接 Framework 設計。

SECTION

## 二十五、Rails 驗證了自然 DSL

Rails 展示 Ruby 可讓：

```
has_many :comments
validates :name, presence: true
```

看起來接近領域陳述。

這是 Ruby 自然性與元程式能力的強證據。

SECTION

## 二十六、Rails 也放大 Ruby 的代價

- 啟動；
- Magic；
- Dynamic method；
- Tooling；
- Dependency；
- Performance；
- Upgrade compatibility。

語言能力在大型 Framework 中會形成有效語言。

SECTION

## 二十七、人本優先不等於不在乎性能

Matz 的優先序通常是：

- Programmer experience；
- Language consistency／harmony；
- Compatibility；
- Implementation feasibility；
- Performance。

  但性能低到妨礙使用時，也會傷害幸福。

SECTION

## 二十八、Ruby 3x3

Ruby 3 以：

- Performance；
- Concurrency；
- Typing；

  為三大方向。官方 Ruby 3.0 Release 說明，特定 Optcarrot Benchmark 達到相對 Ruby 2.0 的三倍性能，同時明確警告並非所有環境和 Benchmark 都有三倍。[R6]

  這是證據導向而非普遍速度口號。

SECTION

## 二十九、Typing without changing Ruby into a static language

Ruby 3 推進：

- RBS；
- TypeProf；

- Static analysis。

  目標不是移除動態 Ruby，而是：

- 改善 Library 契約；
- 工具；
- 大型程式；
- 漸進分析。

  這延續：

> DynamicFreedom
> +
> OptionalEvidence

SECTION

## 三十、Concurrency

Ruby 3 引入：

- Ractor；
- Fiber Scheduler。

  Ractor 在 3.0 被標為 Experimental，試圖以不共享一般物件的 Actor-like 模型降低 Thread-safety 問題。[R6]

  本文不把它寫成已解決 Ruby 所有並行問題。

SECTION

## 三十一、Ruby 會破壞相容

Ruby Issue Tracker 的治理論述明確主張：

- Ruby 必須朝理想 Ruby 改變；

- 即使破壞相容，也可經過過渡期修改。[R5]

  因此 Matz 並非絕對相容主義者。

SECTION

## 三十二、Keyword argument 分離案例

Ruby 2.7／3.0 的 Positional Hash 與 Keyword argument 分離造成大量 Library 遷移成本。

基於 Rails Core 等回饋，Matz 曾決定延後完整分離，顯示：

- 語言一致性重要；
- 真實生態痛苦也重要；
- 過渡節奏可調整。[R7]

SECTION

## 三十三、自然不只面向新手

一個新規則可能對新手更清楚，卻破壞：

- 舊 Library；
- DSL；
- 慣用法；
- 生態心理模型。

  Matz 的自然性是歷史化的，而非只看空白紙。

SECTION

## 三十四、Ruby 4：版本號改變而核心方向延續

Ruby 4.0 於 2025 年 12 月正式發布；截至 2026 年 7 月，官方當前穩定版本為 Ruby 4.0.6。Ruby 4 引入 Ruby Box、ZJIT 等實作與隔離方向，但版本號並不表示 Ruby 已捨棄 Ruby 3 的人本、性能、並行及型別工具路線。

Ractor 在 Ruby 4.0 和 4.1 開發版中仍會發出 Experimental 警告，不能把 Ruby 3 的並行實驗描述成已穩定完成的普遍解答。

SECTION

## 三十五、Matz 仍是最終方向中心

截至當代 Ruby：

- Matz 仍可接受或拒絕語言功能；
- Developer meetings 和 Issue tracker 提供討論；
- Core team 實作和維護；
- Release manager 管理版本；
- 不同公司投入 VM、JIT、Tool 和 Library。

  這是：

> CentralTaste
> +
> DistributedEngineering

SECTION

## 三十六、實作權與語言權分離

例如：

- YARV；
- MJIT；
- YJIT；
- ZJIT；
- RBS；

- Ractor；

都有專門作者和公司投入。

Matz 可設定方向，但成果取決於實作者。

SECTION

## 三十七、Ruby 生態塑造語言

Rails、RSpec、Bundler、RubyGems 和 Shopify 性能工作會反向影響：

- Keyword；
- Block；
- VM；
- GC；
- JIT；
- Pattern matching；
- Tooling。

Ruby 不是只從 Matz 向下傳播。

SECTION

## 三十八、個人工具期

問題：現有腳本語言不符合自己的物件與語言感
策略：綜合 Perl、Smalltalk、Lisp 等

SECTION

## 三十九、Ruby 核心期

問題：如何使腳本自然、物件一致且可表達
策略：Block、Iterator、Mixin、Open object model

SECTION

## 四十、日本社群期

問題：新語言需快速形成共同風格
策略：小社群討論＋Matz 最終裁決

SECTION

## 四十一、全球化期

問題：Ruby 進入大型 Web 生態
結果：Rails 放大生產力與元程式代價

SECTION

## 四十二、再平衡期

問題：性能、相容、Encoding 和 VM 壓力
策略：YARV、1.9／2.x 演化、過渡期

SECTION

## 四十三、Ruby 3／4 期

問題：動態幸福需支撐大型、並行、性能與隔離需求
策略：3x3、Ractor、RBS、JIT／YJIT／ZJIT、Ruby Box

SECTION

## 四十四、問題 framing

Matz 的核心問題是：

> 如何讓程式設計者以接近自己思考和領域表達的方式工作，而不是持續服從機器或語言的機械要求？

SECTION

## 四十五、價值優先序

V_(Matz)
≈

SECTION

## 四十六、核心—擴張偏好

偏好：

- 動態核心；
- Object consistency；
- Block；
- Mixin；
- Library／DSL；
- 元程式；
- 多種自然寫法。

  不追求極小正交核心。

SECTION

## 四十七、顯式—推導偏好

Ruby 偏好：

- 省略括號；
- Duck typing；
- Contextual block；
- Implicit receiver；
- Dynamic dispatch。

  但也透過命名、慣例和工具保持可讀。

SECTION

## 四十八、效率—可讀性偏好

表面優先人類表達；Implementation 承擔：

- GC；
- Dynamic dispatch；
- JIT；
- Inline cache；
- Type speculation。

  後期性能改革不改變人本優先，而是降低其成本。

SECTION

## 四十九、安全—自由偏好

Ruby 提供 Memory-managed runtime，但保留：

- Monkey patch；
- Eval；
- Native extension；
- Metaprogramming；
- Dynamic typing。

  安全更多依賴工具、Library 和組織。

SECTION

## 五十、相容性偏好

偏好保護社群慣用法，但若舊設計阻礙理想 Ruby，可經過過渡期破壞。

SECTION

## 五十一、治理偏好

核心模式：

> CommunityPersuasion
> →
> MatzJudgment
> →
> DistributedImplementation

這比 Python 後 BDFL 更集中，比純私人語言更公開。

SECTION

## 五十二、幸福無法直接測量

「讓程式設計者幸福」是價值方向，不是可直接從語法計算的客觀指標。

需要觀察：

- 學習；
- 維護；
- 除錯；
- 團隊；
- 性能；
- 安全；
- 長期升級。

SECTION

## 五十三、多種自然寫法可能傷害閱讀者

作者的流暢不一定等於團隊的可預測。

Ruby Style Guide、Formatter 和團隊慣例因此很重要。

SECTION

## 五十四、元程式能力可能把成本延後

DSL 看起來自然，但錯誤可能出現在：

- Method missing；
- Runtime generation；
- Framework boot；
- Hidden callback；
- Library interaction。

SECTION

## 五十五、Matz 的品味不等於 Ruby 全部現況

實作、標準 Library、Rails 和工具包含大量其他作者決策。

SECTION

## 五十六、Ruby 3 性能目標有 Benchmark 邊界

官方明確說明三倍結果限於特定 Benchmark 和環境。[R6]

不能寫成 Ruby 3 所有程式皆比 Ruby 2 快三倍。

SECTION

## 五十七、最小驚訝不是正式規格

它不能用來證明某功能必須接受。

Matz 甚至說，更準確是最小化自己的驚訝；真正標準是 Ruby 整體和諧。

| 時期 | 問題 | 決策 | 複雜度去向 | 風格 |
| --- | --- | --- | --- | --- |
| 1993 | 現有腳本語言不自然 | Ruby | Interpreter | 個人品味綜合 |
| 1990s | 物件與腳本分裂 | 一切皆物件、Block、Mixin | Runtime | 自然表達 |
| 1995+ | 新語言需共同方向 | Matz 最終裁決 | 創始者 | 信任治理 |
| 2000s | Web DSL 與生產力 | Rails 生態放大 | Framework | 延展性 |
| 1.9／2.x | Encoding、VM、語義修正 | YARV、重大演化 | VM／遷移 | 再平衡 |
| 2.7／3.0 | Keyword 一致與相容衝突 | 延後、警告、過渡 | 生態 | 信任調節 |
| Ruby 3 | 性能、並行、型別 | 3x3、Ractor、RBS | VM／Tool | 人本可持續 |

SECTION

## 五十八、主要原型

Yukihiro Matsumoto 同時屬於：

- 程式設計者感受導向設計者；
- 語言自然性調音者；
- 和諧高於正交的功能裁決者；
- 動態物件與 DSL 建築師；
- 信任型創始者治理者。

SECTION

## 五十九、不適合的簡單標籤

不應只稱：

幸福語言發明者
最小驚訝設計者
語法糖設計者
Rails 發明者
Ruby 獨裁者
較精確的描述是：

> 把程式設計者的認知感受當成核心設計成本，以個人長期品味協調動態自由、語言和諧、實用生態與相容壓力的設計者。

SECTION

## 六十、最重要的連續性

1993 至 Ruby 3：

> 【機器應為人支付更多轉譯成本】

SECTION

## 六十一、最重要的修正

由：

幸福主要來自自然動態表達
擴張為：

幸福也需要性能、並行、工具、型別證據和可升級生態

SECTION

## 六十二、最重要的制度條件

Ruby 的集中品味能長期存在，是因為：

- Matz 穩定參與；
- 社群信任；
- 公開討論；
- 實作者可說服；
- Fork 始終存在；
- 裁決通常被視為仁慈而非支配。

這些條件不是自動永久成立。

Yukihiro Matsumoto 的核心貢獻，不是證明語言只要看起來像英文就會讓人幸福。

他建立的是一種更深的設計立場：

- 程式設計是一種人類創造活動；
- 語言的價值不只由可計算集合決定；
- 表達時的節奏、挫折、驚訝和專注也是真實工程成本；
- 正交和一致只是工具，不能凌駕人類理解；
- 自然表面可以由複雜實作支撐；
- 動態自由需要社群慣例及長期品味；
- 性能和型別工具最終也服務於可持續幸福。

本文對 Matz 的 PLDST 判定為：

> 【Human-Feeling Language Designer
> →
> Harmonious Dynamic-Language Architect
> →
> Trust-Centered Language Steward】

其核心優勢是：

- 將使用者感受提升為設計的一級目標；
- Block、Mixin 和物件模型具有高度表達力；
- Ruby 可建立自然 DSL；
- 多種慣用法支持不同情境；
- 社群文化與語言氣質相互增強；
- 後期願意投入性能、並行和型別工具。

其核心代價是：

- 自然性具有主觀性；
- 多種寫法增加團隊差異；
- 元程式和開放類別降低靜態可見性；
- 性能及工具需由複雜 VM 補足；
- 相容修正可能造成生態疼痛；
- 語言方向高度依賴對 Matz 的信任。

最終原則為：

> 【幸福不是最少字元
>
> ∧
> 自然不是模仿口語
>
> ∧
> 最小驚訝不是迎合所有人】

Matz 真正提出的是：

> 語言應形成一套可學習、可預測、有節奏的整體品味，使熟悉它的人能把注意力放在問題與創造上；而語言治理者的責任，是在每一次功能、相容與性能衝突中維持這套品味值得社群繼續信任。

人物：Yukihiro Matsumoto

主要語言／制度：Ruby、CRuby language design、Matz-centered governance

核心時期：1993–至今

主要問題：現有腳本語言不能同時提供物件一致性、實用與愉悅

主要策略：自然語法、Block、Mixin、Open class、Metaprogramming

複雜度去向：Interpreter、VM、Tool、Framework、社群慣例

責任去向：機器承擔轉譯，Matz 承擔整體品味裁決

主要保護對象：日常程式設計者、DSL 作者、Web／腳本使用者

主要限制：主觀性、元程式、性能、相容、集中治理

歸因信心：高

[R1] Ruby official FAQ, “What is Ruby?” and origin history.

— 1993 起源、Perl／Python 評價、物件導向腳本需求及名稱歷史。

[R2] Ruby official website, “About Ruby.”

— 影響來源、Careful balance、Natural not simple、表面簡單而內部複雜。

[R3] Bill Venners, “The Philosophy of Ruby: A Conversation with Yukihiro Matsumoto,” Artima, 2003.

— 使用感受、幸福、和諧／正交、最小我的驚訝及人本設計。

[R4] Ruby official homepage and community materials.

— 生態由社群建立、Matz 對社群貢獻的公開評價。

[R5] Ruby Issue Tracker, Feature #7549, “A Ruby Design Process.”

— Matz 最終語言權、說服／Fork、理想 Ruby 與經過過渡期的相容破壞；其中部分表述來自核心參與者，需標示為治理實踐敘述而非正式憲法。

[R6] Ruby official release notes, “Ruby 3.0.0 Released.”

— Performance／Concurrency／Typing、Ruby 3x3 Benchmark 邊界、Ractor、RBS 及 TypeProf。

[R7] Ruby Issue Tracker, Feature #16891 and Ruby keyword-argument transition discussions.

— 生態回饋、相容疼痛、Matz 延後完整分離及過渡政策。

[R8] Yukihiro Matsumoto interviews and talks, including 2021 Evrone interview.

— Developer instinct、Ruby 3、Pattern matching、Ractor 及後期方向；使用時標記為訪談觀點。

[R9] Ruby Core repository and official contributor history.

— CRuby 多作者實作、Core team 及語言／實作分離。

[T-I] Individual-language search phase
[T-C] Ruby core formation phase
[T-J] Japanese community incubation phase
[T-G] Global／Rails expansion phase
[T-B] Compatibility／performance rebalance phase
[T-3] Ruby 3 sustainability phase
[S-H] Programmer happiness
[S-N] Naturalness
[S-A] Harmony over orthogonality
[S-D] Dynamic object model
[S-M] Metaprogramming／DSL
[S-T] Trust-centered governance
---
# 附錄 D　第二輪史實與歸因校對紀錄
## D.1 Ruby 起源日期與原始需求
第二輪重新核對 Ruby 官方 FAQ：
- Ruby 的命名與最初討論日期被記為 1993 年 2 月 24 日；
- Matz 當時討論的是 Object-oriented scripting language；
- 他熟悉 Perl 4 和 Python；
- 對 Perl 的感受及對當時 Python 物件模型的評價，是 Matz 的歷史主觀判斷，不應被改寫為對兩門語言的普遍客觀結論；
- 第一個公開 Ruby 版本於 1995 年發布。
本文因此把 1993 視為問題及語言形成起點，把 1995 視為公共發布起點。
---
## D.2 影響來源與「自然、不簡單」
第二輪核對 Ruby 官方 About 頁：
- 官方列出 Perl、Smalltalk、Eiffel、Ada 和 Lisp 等影響；
- Ruby 被描述為 Careful balance；
- Matz 的公開說法是追求 Natural，而非 Simple；
- Ruby 表面簡單，但內部可以非常複雜。
本文因此不將 Ruby 分類成極小核心語言，而是分類成「以實作與語義複雜度換取表面自然」的設計。
---
## D.3 幸福、和諧與最小驚訝
第二輪重新核對 2003 年 Artima 第一手訪談：
- Matz 明確區分語言「能做什麼」與「使用時如何感受」；
- 他把 Ruby 的主要差異放在人使用時的感覺；
- Programmer happiness 的目標包括減少工作阻力，讓人專注問題；
- 他說 Consistency 和 Orthogonality 是設計工具，而非最高目標；
- 完全正交的功能組合可能造成認知複雜爆炸；
- 他明確指出 Principle of Least Surprise 常被誤解，接近「least my surprise」。
因此本文沒有把最小驚訝寫成對所有初學者背景的普遍承諾。
---
## D.4 Ruby 的設計權

Feature #7549 中「Ruby is matz'」等論述來自 Ruby 核心參與者對當時實際治理的描述，內容包括：
- Matz 對名為 Ruby 的設計和實作方向保有最終覆寫權；
- 社群可以說服 Matz 或 Fork；
- Ruby 可以為理想方向經過過渡期破壞相容。
它不是經社群投票通過的正式憲法，也不表示 Matz 會忽略：
- Developer meeting；
- 實作可行性；
- Core team；
- 生態回饋；
- 相容成本。
本文因此把 Ruby 治理描述為「信任型創始者最終裁決」，而非無程序私人所有。
---
## D.5 Keyword argument 遷移
第二輪核對 Feature #16891 與相關 Ruby 2.7／3.0 討論：
- Positional Hash 與 Keyword argument 分離造成真實 Rails／Library 遷移壓力；
- 基於生態回饋，Matz 決定延後完整分離的一部分；
- 提案者仍主張逐步警告和最終語義一致；
- Issue 最終狀態為 Rejected，表示特定恢復方案沒有被整體採用，而非所有相容考量被忽略。
本文把此案例用作「語言一致性與生態疼痛之間調節」，不把它寫成永久撤回 Keyword 分離。
---
## D.6 Ruby 3x3 的證據邊界
Ruby 3.0 官方 Release Note 明確記錄：
- 目標為 Performance、Concurrency、Typing；
- Optcarrot 特定單執行緒 Benchmark 達到相對 Ruby 2.0 的三倍；
- 官方同時警告不同環境或 Benchmark 未必三倍；
- Ractor 在 Ruby 3.0 被標示為 Experimental；
- RBS 和 TypeProf 是 Static-analysis 工具，不把 Ruby Runtime 改成強制靜態型別。
本文因此沒有把 Ruby 3 寫成所有工作負載三倍或完整靜態型別語言。
---
## D.7 2026 年 Ruby 4 現況
截至 2026 年 7 月 30 日，Ruby 官方資料顯示：
- Ruby 4.0.0 於 2025 年 12 月 25 日發布；
- 當前穩定版本為 Ruby 4.0.6，於 2026 年 7 月 14 日發布；
- Ruby 4.0 引入 Ruby Box、ZJIT 及多項改進；
- Ruby 4.0 採定期維護版本發布；
- Ractor 在 Ruby 4.0.5 及 Ruby 4.1 開發版中仍輸出 Experimental 警告，API 仍可能變更。
因此本文已把最後相位由「Ruby 3」更新成「Ruby 3／4 與後期制度」，並把 Ractor 定位為持續實驗，不宣稱已穩定完成。
---
## D.8 Ruby 4 並非語言哲學斷裂
Ruby 版本號不採嚴格 Semantic Versioning 意義；Major number 改變不自動等於全面破壞相容或治理模式改變。
本文將 Ruby 4 視為：
- Ruby 3 性能、並行、型別工具方向的延續；
- VM、JIT、隔離和實作多團隊化的加深；
- 不是 Matz 人本哲學被另一套機器優先哲學取代。
---

## D.9 生態歸因

Rails、RSpec、RubyGems、Bundler、YARV、YJIT、ZJIT、RBS 和其他工具都有獨立作者與團隊。

Matz 的可直接歸因主要是：

- 語言核心方向；
- 最終功能品味；
- 宏觀目標；
- 對社群及實作者的協調。

本文沒有把 Ruby 生態的全部生產力或性能工作歸入 Matz 個人。

---

## D.10 PLDST 推論邊界

下列名稱是本文分析原型，不是 Matz 自稱的正式學派：

程式設計者感受導向設計者

語言自然性調音者

和諧高於正交的功能裁決者

信任型創始者治理者

「幸福成本函數」是本文的分析模型，不是 Matz 提出的可量化心理學公式。