Guido、Matz 與 Larry Wall：可讀性、幸福與多義性之間的三種人本語言設計

Guido, Matz, and Larry Wall: Three Human-Centered Language Designs Across Readability, Happiness, and Polysemy

v1.0 / 2026-07-30 / Neo.K / 公開版／第四部跨設計師正式比較研究

https://thisoneisneok.com/html/papers/pldst-024.html

英文名稱： Guido, Matz, and Larry Wall: Three Human-Centered Language Designs Across Readability, Happiness, and Polysemy

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-024

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／第四部跨設計師正式比較研究

SECTION

摘要

Guido van Rossum、Yukihiro “Matz” Matsumoto 與 Larry Wall 都曾把程式語言設計從「機器能否執行」推向「人如何使用、閱讀、感受與表達」。Python、Ruby 與 Perl 也經常被共同歸入動態語言、腳本語言、膠水語言或高生產力語言。然而，只用「三者都重視人」概括其設計，會掩蓋三套幾乎相反的人本模型。

Guido 的首要保護對象是程式的下一位讀者以及需要共同維護程式的團隊。Python 透過可見結構、相對稀疏的表面、明示慣例與「最好有一個明顯做法」壓縮等價表達空間，使不同作者的程式較容易進入共同閱讀協定。PEP 20 同時保留實用主義，但將可讀性、明示性、拒絕猜測與可解釋實作置於核心。[R1][R2][R3][R4]

Matz 的首要保護對象是正在編程的人及其持續性的認知流動。Ruby 不追求形式最小，而追求自然、和諧、可表達與「使用時感覺良好」。Matz 明確區分語言「能做什麼」與人「使用它時感覺如何」，並承認自己首先設計的是一門符合自身長期品味的語言，而不是對所有人都完美的語言。[R7][R8][R9][R10]

Wall 的首要保護對象則是背景、任務與表達習慣彼此不同的問題解決者。Perl 將自然語言的 Context、省略、冗餘、多種慣用法與高頻短形式帶入程式語言，拒絕假設所有真實問題只有一個最佳表示。其人本主義不是收斂，而是容納；不是消除方言，而是以文件、文化、工具與社群規範治理方言。[R12][R13][R14][R15][R16]

本文提出三種人本語言設計原型：

> 【Guido：Reader-Centered Convergent Humanism;
> Matz：Programmer-Centered Experiential Humanism;
> Wall：Pluralist Expressive Humanism】

三者對「自由」的理解也不同：

> 【Python：減少不必要選擇，使共同理解更自由;
> Ruby：移除表達摩擦，使程式設計過程更自由;
> Perl：保留多種慣用法，使異質問題求解更自由】

本文以十二個共同軸比較三人：

- 人本對象；
- 可讀性定義；
- 表達變異；
- 明示與隱式；
- 語法密度；
- 常見路徑；
- 物件與抽象；
- 錯誤與安全；
- 工具與慣例；

- 生態擴張；
- 治理；
- 長期複雜度配置。

核心結論是：

> 「以人為中心」不是一項單一價值，而是一場關於究竟保護作者、讀者、團隊、初學者、專家、領域社群或異質問題解決者的責任分配。

Python、Ruby 與 Perl 不是同一條人本語言路線上的優劣排序，而是三種不同的人類模型、三種不同的表達政治，以及三種不同的複雜度搬運方式。

關鍵詞： Guido van Rossum、Yukihiro Matsumoto、Matz、Larry Wall、Python、Ruby、Perl、人本語言設計、可讀性、程式設計者幸福、多義性、TIMTOWTDI、PLDST

SECTION

一、人本語言的最低定義

本文將人本語言設計定義為：

> 語言設計者不只評估機器可執行性與理論表達力，也把人的學習、閱讀、撰寫、記憶、協作、感受、創造與文化差異列為一級設計成本。

這一定義不表示：

- 效能不重要；
- 形式語義不重要；
- 實作可以不精確；
- 使用者喜歡什麼就全部加入；
- 語言設計等同介面美化。

人本設計仍須回答：

> Who benefits?
>
> Who pays?
>
> When is the cost paid?

SECTION

二、不能只問「是否友善」

「友善」至少包含不同方向：

- 對初學者友善；
- 對熟練作者友善；
- 對陌生讀者友善；
- 對大型團隊友善；
- 對工具友善；
- 對領域專家友善；
- 對歷史程式友善；
- 對不同文化與表達習慣友善。

一項設計可能同時提高其中兩項，卻降低另外三項。

SECTION

三、人本對象向量

定義：

> cal-H(L)=
> (
> H_(novice),
> H_(author),

其中每一項不是主觀滿意度，而是語言對該類使用者降低或增加的實際成本。

SECTION

四、人本價值向量

再定義：

> cal-V_(H)(L)=
> (
> R,
> F,
> E,
> C,
> P,
> L,
> T
>)

其中：

- R：Readability，可讀性；
- F：Flow，持續編程流；
- E：Expressiveness，表達力；
- C：Convergence，慣例收斂；
- P：Plurality，表達多元；
- L：Learnability，可學習性；
- T：Toolability，可工具化程度。

  Guido、Matz 與 Wall 都重視這些量，但排序與定義不同。

SECTION

五、複雜度責任向量

語言降低使用者表面成本時，複雜度通常被搬到其他位置。

> cal-B(L)=
> (
> B_(author),
> B_(reader),
> B_(runtime),
> B_(parser),
> B_(tool),
> B_(library),
> B_(governance)
>)

因此：

> HumanFriendlySurface
> not⇒
> TotalComplexitydownarrow

更常見的是：

> HumanFriendlySurface
> ⇒
> ComplexityRelocation

SECTION

六、本文所稱「設計者」

本文比較的是三位設計者在可直接歸因材料中反覆展現的決策風格，不把現代 Python、Ruby、Perl 的所有結果歸於個人。

需要區分：

創始設計者的原始問題
創始設計者的長期品味
核心實作者的工程選擇
社群形成的慣例
生態套件形成的事實語言
現代治理制度

PLDST 比較的是前兩者如何與後四者互動。

SECTION

七、都不是純理論語言起步

三者的原始動機都包含真實工作：

- Python 要成為 ABC 的後繼，同時吸引 Unix／C 使用者並接入現實系統；
- Ruby 要成為 Matz 自己願意使用的物件導向腳本語言；
- Perl 要整合文字處理、系統管理與多個 Unix 工具的工作流。[R1][R7][R8][R12]

  因此三者都不是先建立封閉公理，再尋找應用。

SECTION

八、都接受動態性

三者都使用動態機制降低前期宣告成本，但目的不同：

- Python：讓常見程式保持直接，同時維持可讀協定；
- Ruby：讓物件、Block 與 Metaprogramming 支撐自然表達；
- Perl：讓 Context、轉換、Regex 與多種慣用法快速映射異質資料。

  「動態語言」只是執行與型別機制分類，不能代替風格分類。

SECTION

九、都接受實用性超過純度

Guido 的 Python 格言明確保留「實用性勝過純度」；Matz 拒絕為形式完美犧牲使用感；Wall 則公開反對單一理論大敘事。[R3][R9][R13]

但三者不是同一種實用主義：

> Guido ：Practicality under readable convention;
> Matz ：Practicality under experiential harmony;
> Wall ：Practicality under expressive plurality

SECTION

十、都由強創始品味啟動

三者早期都不是匿名委員會設計：

- Python 長期由 Guido 作最終語言裁決；
- Ruby 長期由 Matz 調整整體語言感；
- Perl 早期核心與文化強烈受 Wall 影響。

  這使語言具有可辨識氣質，也產生權力集中問題。

SECTION

十一、都超出創始者

三種語言都形成：

- 核心實作者；
- Library／Package 生態；
- 教學傳統；
- Style guide；
- Framework；
- 替代實作；

- 正式或半正式治理。

因此現代語言結果不能反向全部歸因於創始者。

SECTION

十二、比較的真正問題

本文不問：

> Python、Ruby、Perl 哪一門最好？

而問：

> 當三位設計者都說自己在為人設計時，他們心中的「人」究竟是誰？

SECTION

十三、首要人本對象：下一位讀者

Python 的核心不是單純「容易寫」，而是程式在離開作者之後仍容易被辨認。

PEP 8 將 Guido 的一項核心洞見概括為：

> 程式被閱讀的次數遠多於被撰寫的次數。[R4]

因此 Python 的人本單位不是孤立作者，而是跨時間的作者—讀者鏈。

SECTION

十四、可讀性不是漂亮排版

Python 的可讀性包含：

- 視覺結構與語法結構一致；
- 名稱與 Namespace 清楚；
- 明示 self；

- Statement 與 Expression 保持可辨識分工；
- 錯誤不默默通過；
- 避免過密特殊符號；
- 常見任務形成共享慣用法。

  縮排只是其中最可見的一項。

SECTION

十五、明示性

PEP 20 的重要對偶為：

> Explicit > Implicit

這不是禁止所有隱式行為。Python 仍有：

- Duck typing；
- Protocol；
- Iterator；
- Context manager；
- Descriptor；
- Import machinery；
- Dynamic dispatch。

  真正限制的是：

> 不要讓讀者必須依靠過多隱藏狀態，才能知道局部程式正在做什麼。

SECTION

十六、一個明顯做法

Python 的「最好只有一個明顯做法」主要是慣例收斂原則，而非數學唯一性。

可形式化為：

> |W_(idiomatic)(T)|
> ≪
> |W_(possible)(T)|

其中：

- W_(possible)(T)：語言允許的所有做法；
- W_(idiomatic)(T)：社群預期且容易辨認的做法。

  Python 不是移除所有自由，而是縮小「正統常見路徑」集合。

SECTION

十七、稀疏表面

PEP 20 偏好 Sparse 而非 Dense。

這種稀疏性讓：

- 控制流容易掃描；
- 一行承載的語義較少；
- 陌生讀者可由結構推斷；
- 工具較容易建立一致格式。

  代價是某些熟練作者覺得冗長或缺乏壓縮力。

SECTION

十八、縮排作為共同協定

Python 以縮排表示 Block，使：

> VisualStructure
> =
> ParserStructure

這減少花括號風格分裂，卻也把空白從排版偏好提升為語法責任。[R5]

這是一種典型 Guido 風格：

> 讓原本由團隊 Style guide 管理的問題，進入語言本身。

SECTION

十九、常見路徑塑形

Guido 的人本設計常不是禁止替代方案，而是讓一條路：

- 最短；
- 最清楚；
- 最容易查到；
- 最符合 Standard Library；
- 最能被 Linter／Formatter 辨識。

因此：

> Humanism_(Guido)
> =
> ChoiceArchitecture

SECTION

二十、逃生口仍存在

Python 仍允許：

- Metaclass；
- Descriptor；
- Monkey patching；
- Dynamic import；
- Native extension；
- Introspection；
- Dunder protocol。

這說明 Python 不是封閉最小語言。

Guido 的典型策略是：

常見能力保持清楚
高階能力放入可辨識的特殊區域

SECTION

二十一、複雜度去向

Python 降低表面變異後，複雜度移至：

- Interpreter；
- Standard Library；
- Object protocol；
- Packaging；
- Tooling；
- PEP 與社群慣例；
- 版本遷移。

因此 Python 的「簡單」是經過制度支撐的簡單。

SECTION

二十二、團隊作為設計尺度

Python 特別適合將個人程式轉成團隊程式，因為：

- Style 收斂；
- 命名慣例清楚；
- Formatter 可統一表面；
- 常見模式容易被辨認；
- 錯誤訊息與 Traceback 可共享；
- 教學資料容易建立共同語彙。

這不是所有領域都最佳，但它明確保護協作。

SECTION

二十三、Guido 原型

本文將 Guido 的人本原型判定為：

> 【Reader-Centered Convergent Humanist】

其核心命題為：

> 使用者最大的自由，不一定來自最多語法選項，也可能來自能迅速理解別人程式的共同語言。

SECTION

二十四、首要人本對象：正在編程的人

Matz 明確區分：

- 語言能做什麼；

- 人使用語言時感覺如何。[R9]

  Ruby 的差異主要不在可計算性，而在：

- 摩擦；
- 節奏；
- 直覺；
- 創造感；
- 注意力是否被機械細節打斷。

SECTION

二十五、自然而非簡單

Ruby 官方資料引用 Matz：

natural, not simple
並指出 Ruby 可表面簡單、內部複雜，如同人體。[R7]

這表示 Ruby 接受：

> SurfaceNaturalnessuparrow
> ⇒
> InternalComplexityuparrow

只要內部複雜度能持續換取人類表達流暢。

SECTION

二十六、幸福成本函數

可將 Matz 的設計目標寫成：

> H_(programmer)
> =
> Flow
> +

「幸福」不是娛樂化，而是讓人把注意力留給問題、創造與抽象。

SECTION

二十七、和諧高於完全正交

Matz 警告形式正交性可能要求使用者在腦中組合太多機制。[R9]

完全正交的設計假設：

> Feature□
> ×
> Feature□

都應自由組合。

Ruby 更偏好：

> Harmony(Feature□,Feature□)

也就是功能在實際語言感中相處自然，即使不是最一般、最對稱的形式。

SECTION

二十八、一切皆物件的感受價值

Ruby 的一致物件觀不只是一項範式標籤。

它讓使用者可以用相近方式理解：

- 數字；
- 字串；
- Class；
- Module；
- Method call；

- nil；
- Metaprogramming。

這降低概念切換，增強「語言是一個可對話世界」的感受。

SECTION

二十九、Block 與 Iterator

Ruby 常把：

對集合做什麼

寫成由集合接收方法，再將行為以 Block 傳入。

這使操作接近：

> 讓這個物件，以這段行為處理自身元素。

它不一定比所有形式更短，卻符合 Ruby 的物件—訊息—Block 節奏。

SECTION

三十、適量隱式性

Ruby 接受：

- 省略 Parentheses；
- 最後一個 Expression 自動成為回傳值；
- Block 與 Method 協作；
- self 在常見呼叫中的省略；
- 開放類別；
- 動態方法；
- Symbol／String 慣例；

- DSL 式呼叫。

這些設計提高自然性，也增加讀者必須掌握的語境。

SECTION

三十一、Metaprogramming 的定位

Matz 指出動態新增方法等能力主要服務適應環境的 Library 與 Metaprogramming，而非鼓勵所有程式任意改寫自己。[R10]

因此 Ruby 的自由是：

語言提供強能力
社群以品味區分日常使用與高階使用

SECTION

三十二、自由與引導

Matz 並不是取消規則，而是相信：

- 語言應提供多種自然表達；
- 設計者應維持整體和諧；
- 社群可發展慣例；
- 不自然功能即使理論漂亮，也可被拒絕；
- 強大能力應由成熟使用者謹慎使用。

這是一種「調音式治理」。

SECTION

三十三、複雜度去向

Ruby 的人本表面把複雜度移至：

- Parser；

- Object model；
- Method lookup；
- Block／Closure；
- Metaprogramming；
- Runtime；
- Tooling；
- 效能最佳化；
- 團隊 Style。

  Ruby 的自然感由大量不可見機制支撐。

SECTION

三十四、設計者作為語言調音師

Guido 常像共同慣例的裁決者；Matz 更像語言感的長期調音者。

他不是只問：

> 此功能是否一致？

而是問：

> 此功能加入後，整門 Ruby 是否仍像 Ruby？

這種判斷難以完全形式化，也難由純投票取代。

SECTION

三十五、Matz 原型

本文將 Matz 的人本原型判定為：

> 【Programmer-Centered Experiential Humanist】

其核心命題為：

> 語言不只是一套正確規則，也是一種長時間停留其中的認知環境；設計者必須為人在其中的感受負責。

SECTION

三十六、首要人本對象：不同背景的人

Wall 不假設所有人都具有同一種自然思考方式。

Perl 面向：

- 系統管理者；
- 文字處理者；
- Unix 使用者；
- 報告生成者；
- Web 開發者；
- 生物資訊工作者；
- 一次性腳本作者；
- 長期模組維護者。

其人本主義從差異開始。

SECTION

三十七、問題先於語言純度

Perl 最初整合：

- Shell 程序能力；

- awk 資料處理；
- sed 轉換；
- grep 模式匹配；
- C 系統接入；
- 一般程式結構。

Wall 的起點是：

> 真實任務已經跨越多個工具，語言應把工作流帶回一個可操作空間。

SECTION

三十八、自然語言不是仿英文

Wall 的自然語言類比主要涉及：

- Context；
- 省略；
- 冗餘；
- 詞類線索；
- 常見形式較短；
- 同義與多種語氣；
- 由簡到繁的學習層次。

Perl 並不是要讓程式像普通英文，而是把人類語言的表達彈性帶入人工語言。

SECTION

三十九、Context 作為語義資源

Perl 的 Scalar、List、Boolean、Void 等 Context 使同一構造能依周圍需求產生不同結果。[R15]

這降低作者重複，卻提高讀者對周圍語境的依賴。

> LocalExpressionMeaning
> =
> Expression
> +
> Context

SECTION

四十、高頻操作壓縮

Wall 的表面常近似 Huffman coding：

> SyntaxLength(operation)
> ∝
> (1)/(Frequency(operation))

高頻文字處理、Regex、預設主題和管線操作可以很短。

這保護熟練作者的速度，但擴大新讀者的符號詞彙。

SECTION

四十一、TIMTOWTDI

“There’s more than one way to do it” 不表示任何寫法同樣好。

其較精確形式為：

> |W_(legitimate)(T)| > 1

因為：

- 問題不同；

- 背景不同；
- 程式生命週期不同；
- 一次性腳本與大型模組不同；
- 不同領域具有不同慣用語。

SECTION

四十二、作者自由與讀者成本

Perl 最清楚展示：

> AuthorChoiceuparrow
> ⇒
> ReaderVocabularyuparrow

多種寫法讓作者更容易貼近問題，卻要求讀者辨認更多方言。

這不是意外副作用，而是多元人本主義的結構性代價。

SECTION

四十三、Sigil、Regex 與預設主題

Perl 透過：

- Sigil 提示資料角色；
- Regex 作為第一級文字語言；
- $_ 省略明顯主題；
- Statement modifier；
- Contextual conversion；

  建立高密度表達。

這些機制對熟練者可形成清晰壓縮，對陌生者則可能像另一套文字系統。

SECTION

四十四、文化是語言第二層

Perl 依靠：

- perldoc；
- Style guide；
- strict；
- warnings；
- 書籍；
- 測試文化；
- CPAN 慣例；
- 社群格言；
- 幽默與演講；

  治理語言自由。

  官方 perlstyle 甚至明確提醒：既然語言提供多種方法，就應選擇最可讀的一種。[R16]

SECTION

四十五、CPAN 與表達分權

CPAN 讓領域社群自行建立：

- Module；
- Interface；
- Object system；

- Testing convention；
- DSL；
- Distribution practice。

這使語言核心不必預先理解所有人類需求。

但它也可能產生：

- 多套框架；
- 品質差異；
- 相依治理；
- 慣例分裂。

SECTION

四十六、平行重寫

當 Perl 5 歷史負擔過大時，Wall 沒有只要求永久相容，也發起 Perl 6 的社群式重設。

State of the Onion 2000 將 Perl 5 比作較由單一作曲者完成的作品，而 Perl 6 則改採社群即興與 RFC 式提案，同時保留 Wall 的藝術控制。[R17]

這顯示 Wall 的多元主義也會在更高層級重新組織。

SECTION

四十七、Wall 原型

本文將 Wall 的人本原型判定為：

> 【Pluralist Expressive Humanist】

其核心命題為：

> 真實世界與人類背景本來就不整齊；語言若為了形式純度消除差異，可能只是把複雜度退回使用者。

SECTION

四十八、Guido 保護讀者

Guido 的核心人本問題是：

> 一段程式離開原作者後，另一個人能否快速辨認其結構與意圖？

其保護對象偏向：

- 維護者；
- 團隊；
- 初學者；
- 跨專案讀者；
- 未來的自己。

SECTION

四十九、Matz 保護作者的持續感

Matz 的核心人本問題是：

> 人在編程時，是否被不必要的機械摩擦打斷？

其保護對象偏向：

- 創作者；
- 應用程式作者；
- DSL 設計者；
- 動態物件使用者；

- 重視語言感的人。

SECTION

五十、Wall 保護異質求解者

Wall 的核心人本問題是：

> 語言是否強迫背景不同的人，先放棄自己的問題慣用法，才能開始工作？

其保護對象偏向：

- 領域社群；
- Unix 工具使用者；
- 文字與資料工作者；
- 熟練壓縮型作者；
- 需要逐層深入能力的人。

SECTION

五十一、三種人類模型

> 【Guido ：Human as reader and collaborator;
> Matz ：Human as experiencing creator;
> Wall ：Human as culturally diverse problem solver】

這是三者最根本差異。

SECTION

五十二、Python：跨作者可預測性

Python 的可讀性主要是：

> Readability_(Python)
> =
> StructuralVisibility
> +
> ConventionConvergence
> +
> LowSymbolDensity
> +
> ExplicitIntent

它偏向公共可讀性。

SECTION

五十三、Ruby：語意接近作者意圖

Ruby 的可讀性主要是：

> Readability_(Ruby)
> =
> NaturalFlow
> +
> ObjectMessageRhythm
> +
> ExpressiveCloseness
> +
> AestheticCoherence

它偏向意圖與表面的接近。

SECTION

五十四、Perl：熟練語境中的資訊壓縮

Perl 的可讀性主要可能是：

> Readability_(Perl)
> =
> DomainFit
> +
> ContextualCompression
> +
> VisibleDataRole
> +
> IdiomRecognition

它高度依賴共享語境與專業詞彙。

SECTION

五十五、可讀性不是單一標尺

一段 Perl One-liner 對熟練文字處理者可能比展開的 Python 程式更直接。

一段 Python 程式對跨背景團隊可能比高度 DSL 化 Ruby 更容易接手。

一段 Ruby DSL 對領域使用者可能比一般函式呼叫更接近問題語言。

因此：

> Readability
> =
> f(Reader,Task,Context,Time)

SECTION

五十六、Python：收斂

Python 傾向：

> Variationdownarrow
> ⇒
> SharedRecognitionuparrow

它以語言、PEP、Style guide、Formatter 與社群慣例共同壓縮變異。

SECTION

五十七、Ruby：受品味約束的多樣性

Ruby 容許：

- 多種呼叫形式；
- 多種 Iterator；
- DSL；
- Metaprogramming；
- 開放類別。

  但 Matz 的長期品味與 Ruby 社群慣例會排除「不像 Ruby」的設計。

  因此：

> Variation_(Ruby)
> =
> Freedom
> ∩
> Harmony

SECTION

五十八、Perl：合法多義性

Perl 對多種寫法更寬容，並讓領域社群形成方言。

> Variation_(Perl)
> =
> ProblemDiversity
> +
> UserDiversity
> +
> HistoricalLayering

其治理主要依靠 Style、文件、工具與社群，而非全部由語法消除。

SECTION

五十九、三種自由

Python：從不必要差異中解放讀者
Ruby：從機械摩擦中解放作者
Perl：從單一表達模型中解放異質使用者

SECTION

六十、Python 的明示

Python 偏好：

- 明示 Block；
- 明示 self；
- 明示 Import；
- 明示 Exception；
- 明示 Namespace；
- 明示資源範圍。

  但以 Protocol 保留高階動態性。

SECTION

六十一、Ruby 的自然省略

Ruby 允許省略：

- Parentheses；
- return；
- 常見 self；
- 部分 Block 細節。

  其判準不是「越隱式越好」，而是：

> 省略是否符合整體語言感，並讓意圖更直接？

SECTION

六十二、Perl 的 Context 省略

Perl 更積極使用：

- 預設變數；
- Scalar／List context；
- 隱式轉換；
- Statement modifier；
- 特殊變數；
- Regex 狀態。

  這使熟練者可高度壓縮，也最依賴語境。

SECTION

六十三、隱式性的成本公式

> Benefit_(implicit)
> =
> RepetitionRemoved
> +
> FlowPreserved
> +
> CommonCaseCompressed

> Cost_(implicit)
> =
> HiddenDependency
> +
> ReaderInference
> +
> ToolAmbiguity
> +
> DebuggingDistance

三者的差異是對兩式權重不同。

SECTION

六十四、Python 的稀疏

Python 通常願意多寫幾個可辨識單詞，以換取：

- 較低符號負擔；
- 較穩定掃描；
- 容易教學；

- 容易格式化；
- 跨領域可讀。

SECTION

六十五、Ruby 的節奏

Ruby 的密度介於 Python 與 Perl 的典型風格之間。

它可以：

- 接近自然語句；
- 使用 Block；
- 省略部分標點；
- 建立 DSL；
- 保持物件訊息節奏。

密度服務的是流動感，而不是單純最短。

SECTION

六十六、Perl 的壓縮

Perl 願意用：

- Sigil；
- Regex；
- 預設主題；
- 特殊運算子；
- Context；

壓縮高頻操作。

其表面更像專業速記系統。

SECTION

六十七、最短不等於最人本

人本設計應區分：

> KeystrokeCost
> ≠
> CognitiveCost
> ≠
> MaintenanceCost

Python 常增加第一項以降低第三項。

Ruby 常降低第二項的摩擦感。

Perl 常降低熟練作者在高頻任務中的第一項與第二項。

SECTION

六十八、Python 的 Protocol 人本主義

Python 不要求所有抽象都由繼承完成，而以：

- Iterator protocol；
- Context manager；
- Callable；
- Descriptor；
- Dunder method；
- Duck typing；

  建立可組合行為。

但特殊能力通常以明顯命名區隔。

SECTION

六十九、Ruby 的物件世界

Ruby 將物件模型提升為語言感核心：

> Everything responds
> ⇒
> Programming as message composition

Block、Mixin、Open class 與 Metaprogramming 使語言可塑，但也提高 Runtime 與工具負擔。

SECTION

七十、Perl 的多模型整合

Perl 不要求所有問題進入單一物件模型。

它可同時使用：

- Procedural；
- Functional idiom；
- Package／Object；
- Regex DSL；
- Data transformation；
- Shell-like orchestration。

這符合 Wall 對多種局部真理的接受。

SECTION

七十一、抽象的三種人本判準

> Python ：抽象是否容易被共同辨認？;
> Ruby ：抽象是否自然地承載意圖？;
> Perl ：抽象是否適合此問題與此社群？

SECTION

七十二、Python：拒絕猜測

PEP 20 強調：

- 錯誤不應默默通過；
- 面對歧義不要猜測。[R3]

Python 仍是動態語言，但錯誤哲學偏向：

> 當系統不能合理確定意圖時，應讓問題顯現。

SECTION

七十三、Ruby：快速成形後測試

Matz 在訪談中將 Ruby 的動態性與快速測試連結，並指出 Interpreter 應保持穩健，而應用可靠性仍需測試與工程實踐。[R10]

其配置偏向：

> PrototypeFast
> +
> RunEarly
> +
> TestBehavior

而不是在語言表面強迫所有正確性證明。

SECTION

七十四、Perl：自由加警告文化

Perl 提供強大能力，也以：

- strict；
- warnings；
- Taint 歷史；
- Test culture；
- Module convention；
- Style guide；

  補充自由。

  其安全責任較多由作者、團隊與文化承擔。

SECTION

七十五、人本不等於無約束

三者都證明：

> HumanFreedom
> +
> NoFeedback
> =
> DeferredFailure

真正差異在於回饋放在哪一層：

- Python：較多放入語言規則與一致慣例；
- Ruby：較多放入 Runtime、測試與社群品味；

- Perl：較多放入 Warning、工具、文件與局部文化。

SECTION

七十六、Python：工具放大收斂

Python 的 Style、Formatter、Linter、Type checker、Test framework 與 Packaging 共同形成：

> Language
> +
> Convention
> +
> Tool
> =
> PracticalPython

工具不是外加裝飾，而是可讀性制度的一部分。

SECTION

七十七、Ruby：工具補足動態可塑性

Ruby 的 Formatter、Linter、Test、RBS／Type tools、Profiler、JIT 與 Framework 慣例，負責處理語言自由造成的部分成本。

Ruby 的實際人本體驗高度依賴成熟 Library 與 Framework。

SECTION

七十八、Perl：文件與 CPAN 作為第二核心

Perl 的實際語言由：

- Core；
- perldoc；
- CPAN；

- Module author；
- Style；
- Test；
- 社群文化；

  共同構成。

  對 Perl 而言，文件與文化不是補充，而是多義性得以運作的必要條件。

SECTION

七十九、生態不是設計者個人作品

Python 的科學計算、Ruby 的 Rails、Perl 的 CPAN 都不能全部歸於創始者。

較精確的因果鏈是：

> FounderStyle
> →
> Affordance
> →
> CommunitySelection
> →
> EcosystemAmplification

創始者提供可能空間，社群決定哪些可能成為歷史現實。

SECTION

八十、Guido：品味制度化後退出

Python 從 BDFL 逐步建立 PEP，再在 Guido 退任後轉為五人 Steering Council。PEP 13 賦予 Council 廣泛權力，同時要求盡量少直接使用，優先建立正常程序與共識。[R6]

這表示：

> FounderTaste
> →
> RecordedProcess
> →
> ElectiveGovernance

SECTION

八十一、Matz：長期調音中心

Ruby 的核心方向長期仍以 Matz 的語言感作重要最終參照，但：

- Interpreter；
- VM；
- JIT；
- Type tools；
- Libraries；
- Framework；
- 替代實作；

已是多團隊成果。

PLDST 將其判定為：

創始者品味中心
+
多團隊工程實作
+
社群信任治理
而不是「Matz 單人控制全部 Ruby」。

SECTION

八十二、Wall：文化權威、社群重寫與後創始制度

Perl 早期由 Wall 強烈塑形；Perl 6／Raku 開放 RFC 與多人實作；現代 Perl 又建立 Core Team 與 Steering Council 的正式治理。[R17][R18]

因此：

> WallCulturalAuthority
> ≠
> CurrentFormalAuthority

SECTION

八十三、治理風格與語言風格一致

三人的治理史與語言哲學具有同構性：

- Guido：由個人品味收斂，最後把收斂程序制度化；
- Matz：長期保留調音中心，以信任維持語言感；
- Wall：鼓勵多元提案與文化表達，再以綜合、分流和制度治理差異。

  這不是完全因果，但具有高度風格連續性。

SECTION

八十四、總體守恆

定義總人本成本：

> C_H
> =
> C_(write)
> +
> C_(read)
> +
> C_(learn)
> +
> C_(debug)
> +

沒有三者中的任何一門語言把 C_H 降為零。

SECTION

八十五、Python 的配置

Python 傾向：

> C_(write)uparrow_(slightly)
>
> C_(read)downarrow
>
> C_(coordinate)downarrow

同時：

> C_(tool)
> +
> C_(governance)
> +
> C_(library)
> uparrow

以維持公共一致性。

SECTION

八十六、Ruby 的配置

Ruby 傾向：

> C_(write-friction)downarrow
>
> C_(expression)downarrow

但：

> C_(runtime)
> +
> C_(tool)
> +
> C_(metaprogram-understanding)
> +
> C_(performance)
> uparrow

SECTION

八十七、Perl 的配置

Perl 傾向：

> C_(common-task-authoring)downarrow
>
> C_(domain-fit)downarrow

但：

> C_(reader-vocabulary)
> +
> C_(context-inference)
> +
> C_(style-coordination)
> +
> C_(historical-layering)
> uparrow

SECTION

八十八、作者—讀者不對稱

三者可放在一條作者—讀者負擔軸上，但不能簡化成單線排名。

Python：較主動限制作者表達變異，以降低陌生讀者成本
Ruby：讓作者意圖自然流出，以社群品味控制讀者成本
Perl：讓作者依任務選擇方言，以文化與專業共享承擔讀者成本

SECTION

八十九、專業化梯度

人本設計還需考慮專業化。

> C_(novice)
> ≠
> C_(expert)

Perl 的高密度可能對新手昂貴，對熟練文字處理者便宜。

Ruby 的 DSL 可能對 Framework 作者複雜，對領域使用者自然。

Python 的一致表面可能對團隊便宜，對追求極端語言可塑性的專家形成限制。

SECTION

九十、規模效應

令 n 為：

- 程式大小；
- 團隊人數；
- 維護年限；
- 套件數；
- 參與社群數。

則：

> C_H = C_H(n)

某設計在小腳本中非常人本，在大型長期系統中可能需要額外制度。

SECTION

九十一、迭代範例

Python：

```
for item in items:
  process(item)
```

Ruby：

```
items.each do |item|
 process(item)
end
```

Perl：

```
for my $item (@items) {
  process($item);
}
```

三者都清楚，不能由此簡單判定誰更人本。

真正差異在更大語境：

- Python 強調共同控制流形式；
- Ruby 強調集合接收行為 Block；
- Perl 保留資料角色 Sigil 與多種迴圈寫法。

SECTION

九十二、隱式主題範例

Perl 可使用預設主題壓縮：

```
while (<>) {
  chomp;
  print if /error/;
}
```

這對熟悉者接近文字管線，對陌生者則必須補出：

- 輸入來自何處；
- chomp 作用於誰；
- Regex 比對誰；
- print 輸出誰。

這正是作者壓縮與讀者推理的交換。

SECTION

九十三、Ruby DSL 範例

Ruby 可利用省略括號、Block 與方法呼叫建立：

```
route "/users" do
  authorize :admin
  render :index
end
```

這可能對領域使用者非常自然，但其可讀性依賴：

- 方法解析；
- Block scope；
- Framework convention；
- Runtime Metaprogramming。

SECTION

九十四、Python 慣例範例

Python 常把 Framework 能力約束為較明示的物件、Decorator、Context manager 或函式呼叫。

這通常增加可見結構，但不代表內部沒有 Reflection 或 Dynamic dispatch。

SECTION

九十五、範例不能取代歷史

幾行程式只能顯示表面。

PLDST 必須同時追蹤：

- 為何選擇此表面；
- 哪些替代方案被拒絕；
- 誰負責實作；
- 社群如何使用；
- 工具如何補足；
- 長期演化造成什麼代價。

SECTION

九十六、Python 並非真的只有一種做法

Python 仍有：

- Loop／Comprehension／Generator；
- Class／Closure／Callable object；
- Decorator／Context manager；
- 多種 Web、Packaging、Async 與 Type 工具。

因此「一個明顯做法」是核心慣例理想，不是生態事實的絕對描述。

SECTION

九十七、Ruby 並非只憑主觀感覺

Matz 的品味長期受到：

- 相容性；
- 實作可能；

- 效能；
- 社群使用；
- 工具；
- 真實程式；

  約束。

  「感覺」是設計證據的一類，不是唯一證據。

SECTION

九十八、Perl 並非以不可讀為目標

Perl 官方 Style guide 明確重視：

- 可讀；
- 可理解；
- 可維護；
- strict；
- warnings；
- 合理命名；
- 選擇較清楚寫法。[R16]

  TIMTOWTDI 不等於所有寫法同樣好。

SECTION

九十九、人本不等於心理人格診斷

本文不主張：

- Guido 天生控制；

- Matz 天生感性；
- Wall 天生混亂。

  PLDST 分析的是公開決策中的穩定模式，不推斷私人心理本質。

SECTION

一百、語言成功不能直接證明哲學正確

Python 的廣泛採用、Ruby 的 Framework 生產力、Perl 的歷史影響，都由多種因素共同造成：

- 時機；
- Library；
- 教育；
- Web；
- 組織採用；
- Package 生態；
- 既有系統；
- 社群；
- 替代語言狀態。

  成功是證據之一，不是唯一因果證明。

SECTION

一百零一、三者都有時間相位

Guido、Matz 與 Wall 的風格不是靜止標籤。

- Python 從個人 Interpreter 走向 PEP 與 Council；
- Ruby 從個人語言走向全球多團隊實作；

- Perl 從創始者核心走向 Porters、CPAN、Perl 6／Raku 與正式治理。

  比較必須標記時間。

SECTION

一百零二、十二軸矩陣

軸 Guido／Python Matz／Ruby Wall／Perl

首要人本對象 讀者、團隊、未來維護者 正在編程的人、創作者 異質問題解決者、領域社群

核心價值 可讀、明示、慣例收斂 幸福、自然、和諧 實用、多義、表達自由

變異態度 壓縮正統做法集合 容許多樣，但需像 Ruby 接受多種合法慣用法

明示／隱式 偏明示 自然省略 Contextual implicitness

表面密度 稀疏 節奏化、中度壓縮 高頻短碼、可高度密集

抽象模型 Protocol 與共享慣例 物件、Block、Metaprogramming 多模型與工具整合

常見路徑 一個明顯慣用路徑 一組和諧自然路徑 多個任務適配路徑

錯誤責任 顯性錯誤、拒絕猜測 快速執行、測試、Runtime strict、Warning、文化

複雜度去向 Interpreter、Library、工具、治理 Runtime、物件模型、工具、效能 Parser、Context、讀者詞彙、文化

治理原型 BDFL→PEP→Council 創始品味＋多團隊 創始文化→RFC／分流→Council

主要優勢 公共可讀與團隊協作 表達流與 DSL 創造力 異質任務整合與熟練壓縮

主要風險 慣例僵化、隱藏複雜度 Magic、工具與追蹤成本 方言、非局部理解、歷史負擔

SECTION

一百零三、三種風格公式

Guido：

$$J_G = \alpha R_{(reader)}$$

Matz：

> J_M
> =
> α F_(programmer)
> +
> β N_(natural)
> +
> γ E_(expression)
> -
> λ Friction
> -
> μ Disharmony

Wall：

> J_W
> =
> α Fit_(task)
> +
> β P_(plurality)
> +
> γ C_(context)
> -
> λ Imposition_(single-model)

這些是本文分析模型，不是三位設計者提出的量化公式。

SECTION

一百零四、風格距離

可定義：

> D(i,j)
> =
> Σ☒

但實際比較必須附上：

- 原始資料；
- 決策案例；
- 時期；
- 實作條件；
- 社群結果；
- 反例。

  數值不能替代歷史證據。

SECTION

一百零五、先定義「人」

任何宣稱 Human-centered 的新語言，都應先回答：

主要保護誰？
作者還是讀者？
個人還是團隊？
初學者還是專家？
一般程式設計者還是領域使用者？
短期原型還是二十年維護？

SECTION

一百零六、不要同時宣稱所有自由

新語言不能無代價地同時保證：

- 一種明顯做法；
- 完全自由表達；
- 極短語法；
- 陌生讀者立即理解；

- 強 Metaprogramming；
- 完美工具推斷；
- 永久相容。

  必須公開優先序。

SECTION

一百零七、建立表達變異預算

定義：

> Budget_(variation)
> =
> CoreAlternatives
> +
> SyntaxAliases
> +
> ImplicitContexts
> +
> MetaprogrammingEscape
> +
> LibraryDSL

若變異預算過高，需增加：

- Style；
- Linter；
- Typed interface；
- Documentation；
- Scope restriction；
- Migration rule。

SECTION

一百零八、建立人本負擔表

每項功能應記錄：

主要受益者：
主要受損者：
降低的撰寫成本：
增加的閱讀成本：
新增的 Runtime 負擔：
Tooling 是否可恢復資訊：
團隊是否需要額外慣例：
長期相容成本：

SECTION

一百零九、設計共同路徑與逃生口

三人的共同教訓是：

> 人本語言既需要日常路徑，也需要高階逃生口。

差異在於：

- Python 把逃生口標記得更特殊；
- Ruby 把逃生口融入物件與 Metaprogramming；
- Perl 讓多種路徑本身成為語言文化。

新語言應明示自己的選擇。

SECTION

一百一十、工具不能事後補救一切

若語言允許高隱式性、動態改寫與多種方言，Tooling 必須在設計早期介入。

否則：

> LanguageFreedom
> -
> ToolRecovery
> =
> MaintenanceDebt

SECTION

一百一十一、治理也是人本介面

使用者不只與語法互動，也與以下制度互動：

- 提案；
- 拒絕理由；
- 版本；
- 遷移；
- 相容承諾；
- 創始者權力；
- 社群申訴；
- 接班。

治理決定誰能改變語言，因此也是人本設計的一部分。

SECTION

一百一十二、三者不是同一條線

Guido、Matz 與 Wall 不能排列為：

最可讀 → 次可讀 → 最混亂
也不能排列為：

最理性 → 最感性 → 最自由

這些線性排序忽略了人本對象差異。

SECTION

一百一十三、三種人本憲法

Guido 的憲法：

> 語言應限制不必要的表達分歧，使程式成為可跨作者流通的公共文本。

Matz 的憲法：

> 語言應承擔足夠內部複雜度，讓程式設計者保持自然、創造與持續流動。

Wall 的憲法：

> 語言應承認問題與人類背景的異質性，提供多種可用慣用法，再以文化、文件與工具治理自由。

SECTION

一百一十四、三種複雜度政治

> 【Guido：把選擇複雜度移出日常表面;
> Matz：把表達摩擦移入 Runtime 與語言內部;
> Wall：把異質性保留在語言與文化之中】

SECTION

一百一十五、三種自由的不可合併性

Python 式自由：

> Freedom_(shared understanding)

Ruby 式自由：

> Freedom_(creative flow)

Perl 式自由：

> Freedom_(plural expression)

三者可以混合，但混合時必須支付交互成本。

SECTION

一百一十六、最終 PLDST 判定

> 【Guido van Rossum
> ：
> Reader-Centered Convergent Humanist;
> Yukihiro Matsumoto
> ：
> Programmer-Centered Experiential Humanist;
> Larry Wall
> ：
> Pluralist Expressive Humanist】

SECTION

一百一十七、本文最後命題

> 程式語言的「人本性」不在於是否讓人感到容易，而在於它是否誠實說明：它正在替哪一種人降低哪一種成本，又把代價交給誰。

Guido、Matz 與 Wall 的重要性，不只是創造 Python、Ruby 與 Perl。

他們共同證明：

> 【HumanCenteredLanguageDesign
> =
> HumanModel
> +
> BurdenAllocation
> +
> ExpressionPolitics
> +
> Governance】

只有把這四項同時展開，PLDST 才能超越人物傳記與語言功能比較，進入可重複使用的設計方法。

比較組：Guido van Rossum／Yukihiro Matsumoto／Larry Wall
主要語言：Python／Ruby／Perl
共同類型：動態、高階、腳本／一般用途、開源社群語言
共同命題：
把人類使用成本提升為語言設計的一級問題
Guido：
主要人本對象＝讀者與團隊
主要策略＝明示、稀疏、慣例收斂、共同路徑
複雜度去向＝Interpreter、Library、Tooling、PEP
主要風險＝慣例僵化、表面簡單掩蓋底層複雜
Matz：
主要人本對象＝正在編程的人
主要策略＝自然、和諧、Block、物件、Metaprogramming
複雜度去向＝Runtime、Object model、Tooling、效能工程
主要風險＝Magic、追蹤困難、團隊 Style 分裂
Wall：
主要人本對象＝異質問題解決者
主要策略＝Context、TIMTOWTDI、Regex、Sigil、文化
複雜度去向＝Parser、Reader vocabulary、文件、社群
主要風險＝方言、非局部理解、歷史層累積
核心比較：
Python＝收斂型人本
Ruby＝體驗型人本
Perl＝多元型人本
歸因信心：高

人物	問題	決策	主要受益者	複雜度去向	PLDST 標記
Guido	視覺與語法 Block 可能不一致	縮排成為語法	讀者、團隊	Parser、格式責任	G-R

Guido 等價做法過多 明顯慣用路徑 維護者、初學者 Style、治理 G-C

Guido 動態語言仍需系統接入 C Extension／Protocol 應用作者 Interpreter、邊界 G-P

Matz 語言能做事但使用感差 幸福與自然性優先 作者 Runtime M-H

Matz 完全正交增加腦內組合 和諧高於正交 一般使用者 設計者裁決 M-A

Matz 樣板打斷意圖 Block、省略、DSL 作者、領域使用者 Tooling、追蹤 M-F

Wall Unix 工具碎片化 Perl 整合工作流 系統與文字工作者 Interpreter W-U

Wall 不同問題需要不同慣用法 TIMTOWTDI 異質使用者 Reader、Style W-M

Wall 常見文字操作過長 Regex、Context、預設主題 熟練作者 Parser、讀者詞彙 W-C

Wall 歷史負擔難以局部修補 Perl 6／Raku 平行重設 未來社群 多實作、分流 W-R

[R1] Guido van Rossum, “Foreword for Programming Python,” Python.org, 1996.

— Python 起源、ABC 後繼、Unix／C 使用者、創始哲學。

[R2] Guido van Rossum, “Comparing Python to Other Languages,” Python.org, 1997.

— Python 可讀／可維護表面、與 Perl、Smalltalk、C++ 等比較；官方標記為歷史材料。

[R3] Tim Peters, “PEP 20 – The Zen of Python,” Python Enhancement Proposals.

— 明示、簡單、稀疏、可讀、實用、拒絕猜測與明顯做法。

[R4] Guido van Rossum, Barry Warsaw, Alyssa Coghlan, “PEP 8 – Style Guide for Python Code.”

— 程式閱讀高於撰寫、跨 Python 程式的一致可讀性。

[R5] Python Documentation, “Design and History FAQ.”

— 縮排、self、方法／函式等設計理由。

[R6] Python Enhancement Proposals, “PEP 13 – Python Language Governance.”

— Steering Council、權力、共識、選舉及後 BDFL 治理。

[R7] Ruby-lang.org, “About Ruby.”

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

[R8] Ruby-lang.org, “Official Ruby FAQ.”

— 1993 年起源、物件導向腳本目標、Perl／Python 背景及多作者文件。

[R9] Bill Venners, “The Philosophy of Ruby: A Conversation with Yukihiro Matsumoto, Part I,” Artima, 2003.

— 使用感、無完美語言、正交性風險、自由與引導、人本優先。

[R10] Bill Venners, “Dynamic Productivity with Ruby: A Conversation with Yukihiro Matsumoto, Part II,” Artima, 2003.

— Metaprogramming、Mixin、動態生產力、Runtime 與測試責任。

[R11] Ruby-lang.org, “Ruby From Other Languages,” including Python／Perl transition guides.

— Ruby、Python、Perl 的表面與物件慣例對照。

[R12] Perl official documentation, perlhist.

— Perl 發布歷史、多作者維護與長期共同體演化。

[R13] Larry Wall, “Perl, the First Postmodern Computer Language,” Perl.com, 1999.

— 多來源重組、使用者創造力、反單一理論與後現代實用主義。

[R14] Larry Wall, “The Culture of Perl,” Perl Conference keynote, 1997.

— Perl 文化、矛盾價值、創始者比喻與共同體。

[R15] Perl official documentation, perldata, perlsyn, perlop.

— Scalar／List context、自由格式、資料角色與實際語義。

[R16] Perl official documentation, perlstyle.

— 多種做法下的可讀、可理解、可維護、strict 與 warnings。

[R17] Larry Wall, “State of the Onion 2000,” Perl.com.

— Perl 6 社群設計、RFC、創始者藝術控制與實作分工。

[R18] Perl official documentation, perlgov and perlpolicy.

— Core Team、Steering Council、治理邊界及正式權力配置。

[R19] PLDST-017, Guido van Rossum：可讀性、實用主義與 BDFL 裁決.

— 本系列前置個案研究。

[R20] PLDST-018, Yukihiro Matsumoto：程式設計者幸福、語言自然性與社群信任.

— 本系列前置個案研究。

[R21] PLDST-019, Larry Wall：語言多義性、後現代實用主義與社群文化.

— 本系列前置個案研究。

資料查核日期： 2026-07-30。

[G-R] Guido: reader-centered readability
[G-C] Guido: convention convergence
[G-E] Guido: explicitness
[G-P] Guido: pragmatic escape hatch
[G-G] Guido: governance institutionalization
[M-H] Matz: programmer happiness
[M-N] Matz: naturalness
[M-A] Matz: harmony over orthogonality
[M-F] Matz: flow-preserving expression
[M-T] Matz: taste-centered governance
[W-U] Wall: Unix problem integration
[W-C] Wall: context-sensitive expression
[W-M] Wall: multiple legitimate idioms
[W-P] Wall: postmodern pragmatism
[W-K] Wall: culture as governance
[W-R] Wall: parallel rewrite
[C-H] comparative human model
[C-B] burden allocation
[C-V] variation politics
[C-R] reader-author asymmetry
[C-G] governance continuity

SECTION

E.1 PEP 20 的作者歸因

PEP 20 作者是 Tim Peters，文件表示其內容濃縮 BDFL 的 Python 設計原則。

因此本文採用：

格言文本作者：Tim Peters
被濃縮的設計風格：Guido／Python 傳統
不把 PEP 20 誤寫為 Guido 親自起草。

SECTION

E.2 Python 的「一種做法」不是絕對唯一

第二輪核對 PEP 20、PEP 8 與 Python 實際語言能力後，本文將其限定為：

- 核心常見任務的慣例收斂；
- 降低無必要的等價表面競爭；
- 不代表整個生態只有一個 Framework、Algorithm 或 Architecture。

SECTION

E.3 Matz 的「最小驚訝」邊界

Matz 在第一手訪談中指出，不同人具有不同驚訝來源，Ruby 無法滿足所有背景的既有直覺。

因此本文不把 Ruby 寫成：

符合每個人的直覺
而寫成：

在理解 Ruby 整體品味後，局部設計應維持自然與和諧

SECTION

E.4 Ruby 的內部複雜不是缺陷承認

Ruby 官方「表面簡單、內部複雜」不是說內部品質可以混亂，而是承認：

- Parser；
- Runtime；
- Object model；
- Metaprogramming；
- VM；

可承擔較多機制，以降低程式設計者表面摩擦。

SECTION

E.5 Wall 的多義性不是無確定語義

Perl 可以有：

- Context；
- 省略；
- 多種慣用法；
- 高密度符號。

  但每一段可執行程式仍需由 Parser 與 Runtime 決定精確行為。

  因此：

> LinguisticPolysemy
> ≠
> UndefinedSemantics

SECTION

E.6 TIMTOWTDI 與 Perl Style

第二輪核對 perlstyle：

- 官方文件承認個人格式偏好；
- 同時要求選擇可讀、可理解、可維護的寫法；
- 現代建議強調 strict 與 warnings。

  所以本文沒有把 TIMTOWTDI 解釋成反對 Coding standard。

SECTION

E.7 Perl 6 的社群設計仍有 Wall 裁決

State of the Onion 2000 同時包含：

- 社群即興；
- RFC；
- 多人實作；
- Wall 保留藝術控制。

  因此 Perl 6 既不是 Wall 單人設計，也不是所有 RFC 直接民主加入。

SECTION

E.8 現代治理分離

截至查核日：

- Python 正式治理由 PEP 13 Steering Council 承擔；
- Perl 正式治理由 Core Team 與 Steering Council 架構承擔；
- Ruby 的核心方向仍高度依賴 Matz 品味，但實作、生態與工具為多團隊成果。

  本文已分開：

> FounderInfluence
> ≠
> SoleCurrentControl

SECTION

E.9 「人本」不是道德優越排序

本文的三種 Humanism 是分析原型，不表示：

- Python 比 Ruby 更理性；
- Ruby 比 Perl 更幸福；
- Perl 比 Python 更尊重自由；
- 任一語言對所有人都更友善。

每種風格只在特定人類模型與條件下成立。

SECTION

E.10 PLDST 推論邊界

下列名稱是本文提出的比較原型，不是三位設計者自稱的正式學派：

Reader-Centered Convergent Humanist
Programmer-Centered Experiential Humanist
Pluralist Expressive Humanist
其用途是建立可重複比較的設計風格索引。

PLDST-024 比較的是三種人本語言設計：

讀者收斂
程式設計者體驗
異質表達多元
PLDST-025 將轉向另一組更基礎的問題：

> 當設計者把計算理解為函數、符號操作、資料轉換或極簡值模型時，語言的「簡單性」究竟指什麼？

下一篇預定為：

PLDST-025：Backus、McCarthy 與 Hickey——函數、符號與簡單性的不同道路。