程式語言也是歷史妥協體嗎？從理想語言到生態相容性

v0.1 / 2026-08-01 / Neo.K

https://thisoneisneok.com/html/papers/ois-09.html

系列：《表觀完好系統：從軟體屎山、補償性完整到動態架構治理》

篇次：09 / 12

作者：Neo.K

協作整理：Aletheia / GPT

版本：v0.1

日期：2026-08-01

SECTION

摘要

前八篇已將「表觀完好」從單一程式碼品質問題推進到補償性完好、有效架構、生存架構、歷史殘留、補償凝固、複雜度轉移與軟體結構動力學。這些研究共同顯示：長期運行的軟體不是一次完成的純粹設計物，而是在變更、相容性、使用者依賴、歷史決策與治理能力共同作用下形成的演化結構。

本文進一步提出一個更大的問題：

> 程式語言本身是否也會經歷同樣的演化壓力？

答案需要非常小心。本文不主張「成熟程式語言都是缺陷品」，也不把所有 backward compatibility 視為負擔。相反地，本文將成熟程式語言視為一類特殊的「長壽公共基礎設施」：其語法、語義、ABI、標準函式庫、工具、compiler behavior 與生態慣例，會逐步成為大量既有程式的依賴表面。當語言成功之後，設計者便不再只面對「什麼語言設計最好」這個問題，而必須同時處理：

- 新能力；
- 舊程式；
- source compatibility；
- binary compatibility；
- ecosystem interoperability；
- migration cost；
- implementation diversity；
- toolchain stability；
- user expectations。

C++ 的現行語言演化原則明確把與舊 C++ 的 source／link compatibility 視為預設要求；Java Language Specification 專章定義 binary compatibility，並指出 widely distributed classes/interfaces 的 accessible members 及其行為形成對使用者的 contract；Python PEP 387 明確要求不相容變更具有高 benefit-to-breakage ratio，並在 2025 年更新為偏好 5 年 deprecation period；ECMAScript 2026/2027 規格 Annex B 更直接保留一組具有「undesirable characteristics」、若沒有 legacy usage 原本會被移除的瀏覽器語言特性。Rust 則提供一個不同治理策略：透過 editions 讓 backward-incompatible language changes 由專案主動 opt-in，並要求不同 editions 的 crates 仍能互通，再用 cargo fix --edition 降低 migration cost。

本文因此提出「語言歷史妥協體（Historically Constrained Language System）」概念：成熟語言不是純粹由當代設計原則決定，而是：

> 【L(t)
> =
> L_(core)
> +
> H(t)
> +
> K(t)
> +
> E(t)
> +
> M(t)】

其中 H 是歷史沉積，K 是相容性約束，E 是生態依賴，M 是遷移治理機制。語言成功越大，其「相容性表面」通常越大，因此任何語言改善都會受到更高的現實破壞成本約束。

本文進一步提出「語言兼容壓力」「語言凝固」「語言歷史殘留」「語義雙軌」「遷移通道」與「演化自由度」等概念，並指出成熟語言真正的品質不能只由「今天從零設計是否漂亮」判斷，而應同時考察它能否在不破壞龐大生態的情況下持續演化。

關鍵詞：程式語言演化、backward compatibility、C++、Java、Python、ECMAScript、Rust editions、語言設計、歷史沉積、生態相容性、演化自由度

SECTION

1. 從「大部分程式語言是不是缺陷品？」重新開始

本系列最初有一個帶有歪臉笑意味、但非常值得認真處理的疑問：

> 如果連一套經過明確設計的 MSSP 在實作後都能找出大量問題，那些長期堆疊、歷史複雜的軟體，以及承載它們的程式語言本身，會不會其實也像「很多不完美元件組成的表觀完好產品」？
> 對 application layer，前八篇已經證明這個問題有研究價值。
>
> 但到了 programming language layer，不能直接把結論搬過來。
>
> 因為程式語言與一般應用程式不同。
>
> 一個成熟語言的「使用者」不是幾萬個人在操作 UI，而可能是：

> 10□~10□

級別的 source files、libraries、binaries、tools、build scripts 與外部系統。

所以當語言設計者想改一個看似很小的規則時，真正的問題不是：

> 這個新規則比較漂亮嗎？

而是：

> 有多少既有世界已經建立在舊規則上？

SECTION

2. 語言不是只有 grammar

將程式語言簡化成：

> L=(Syntax,Semantics)

對教科書足夠，

但對成熟語言演化不夠。

一個實際語言生態至少包含：

> cal-L
> =
> (
> S_y,
> S_m,
> T,
> A,
> B,
> R,
> P,
> E,
> G
>)

其中：

- S_y：Syntax；
- S_m：Semantics；
- T：Type / static rules；
- A：ABI / calling conventions / binary assumptions；

- B：Standard library behavior；
- R：Runtime；
- P：Compiler / toolchain behavior；
- E：Ecosystem；
- G：Governance / evolution process。

  因此改「語言」可能意味著改：

- parser；
- type checker；
- optimizer；
- binary interface；
- standard library；
- package ecosystem；
- build system；
- lint；
- IDE；
- existing code assumptions。

  這使成熟語言天然具有：

SECTION

Large Compatibility Surface

SECTION

3. 成功使語言從「設計產品」變成「公共基礎設施」

新語言早期：

> $N_{users} \approx 0$

所以：

> $C_{breakage} \approx 0$

設計者可以：

- 改 keyword；
- 改 syntax；
- 改 ownership rule；
- 改 type inference；
- 改 standard library；
- 改 error handling；

而不需要擔心數十億行程式。

但成功後：

> $N_{users} \uparrow$

> $N_{libraries} \uparrow$

> $N_{binaries} \uparrow$

> $N_{tools} \uparrow$

則：

> $C_{breakage} \uparrow$

因此語言的優化問題從：

> max Q_(design)

轉成：

> max
> (
> Q_(design)
> -
> C_(migration)
> -
> C_(breakage)
>)

這是成熟語言與新語言最根本的差異之一。

SECTION

4. C++：相容性不是副作用，而是明確的演化原則

C++ 是最典型的案例。

2024 年更新的 C++ Language Evolution Principles 明確重申兩個原則：

- Retain link compatibility with C and previous C++；
- No gratuitous incompatibilities with C or previous C++。

  該文件甚至把：

  > 與舊 C++ 100% seamless friction-free 的 link compatibility

  視為預設要求，ABI break 必須被明確討論與批准。

  同時：

> 與舊 C++ 的 source compatibility

也被設為強烈預設。

因此 C++ 今天的一部分複雜度不能簡單解釋成：

> 以前的人設計不好。

更準確是：

> 很多舊能力一旦被龐大生態使用，就具有很高的移除成本。

SECTION

5. C++ 的「太複雜」其實與「不能亂刪」同時成立

ISO C++ 方向文件曾非常直接地描述這個矛盾：

> C++ 很複雜，甚至太複雜；但 significant facilities 很難移除，因為使用者依賴它們。
> 這形成：

> Desire for Simplicity
> ↔
> Compatibility Cost

如果：

> F_{old}

很複雜，

但：

> $N_{dependents}(F_{old})$
> $\gg 0$

則：

> Remove(F_(old))

可能比：

> Add(F_(new))

更昂貴。

於是語言容易形成：

old mechanism
+
new safer mechanism
+
bridge
+
deprecation
這就是：

SECTION

Language Layering

而不是單純的「設計不統一」。

SECTION

6. C++ 的語言凝固

第六篇提出：

> C⊠→ A_(t+1)

即 workaround 可以凝固為 architecture。

程式語言也存在相似現象。

一個早期 feature：

> F☒

如果被大量程式、compiler、library、ABI 使用，

則：

> Dependents(F☒)uparrow

它的：

> W_L(F☒)

即 Language Architectural Weight 上升。

最後：

> F☒

即使不符合今天最佳設計，

也很難刪除。

本文稱為：

SECTION

Language Solidification

形式上：

> 【Usage
> +
> Dependency
> +
> Compatibility
> →
> Language Solidification】

SECTION

7. Java：公開行為本身就是 contract

Java Language Specification Chapter 13 對 binary compatibility 的處理非常值得注意。

規格指出：

對廣泛散布的 packages、classes 與 interfaces，

重新編譯所有既有 consumer 往往不實際。

因此 Java 明確規範哪些變更：

> Δ L

可以保持既有 binaries 繼續 linkage。

更重要的是：

> accessible members、constructors，以及其存在與行為，都應被視為對使用者的 contract。
> 這等於直接承認：

> Published Behavior
> →
> Compatibility Obligation

所以成熟語言不能只問：

> 新版本 compiler 能不能編譯新 code？

還必須問：

> 舊 binary 還能不能活？

SECTION

8. Source Compatibility 與 Binary Compatibility 不是同一件事

Java 規格甚至明確區分：

> K_s
> =
> Source Compatibility

與：

> K_b
> =
> Binary Compatibility

一組 pre-existing binaries 可以保持正常，

但同一批 source code：

> recompile

後卻可能不再成立。

因此成熟語言的兼容問題不是單一：

backwards compatible = yes/no

而是一個向量：

> K_L
> =
> [
> K_s,
> K_b,
> K_a,
> K_r,
> K_e
>]

其中：

- K_s：source compatibility；

- K_b：binary compatibility；
- K_a：API compatibility；
- K_r：runtime/behavior compatibility；
- K_e：ecosystem compatibility。

SECTION

9. Python：破壞可以做，但必須有足夠理由與遷移時間

Python PEP 387 是另一種語言治理方式。

其基本原則非常清楚：

> incompatible change 必須有足夠大的 benefit-to-breakage ratio，而且 affected code 應容易修正。

Python 並沒有宣稱：

> 永遠不破壞。

而是把破壞成本變成治理問題。

其 public API 包括：

- syntax；
- behavior；
- C API；
- function/class/module；
- return value；
- side effects；
- raised exceptions。

也就是：

> Compatibility Surface

遠比 function name 大。

SECTION

10. Python 的 deprecation period 就是一條時間化遷移通道

PEP 387 在 2025 年更新為：

> preferred 5-year deprecation period before removal。

這是一個非常值得抽象化的設計。

當舊能力：

> F_(old)

需要退出，

不是：

> F_(old)
> →∅

而是：

> F_(old)
> →
> Warn
> →
> Migration Window
> →
> Removal

本文稱：

SECTION

Migration Corridor

即遷移通道。

它的作用是把：

> $C_{(breakage)}$

從瞬間衝擊，

轉成：

> $C_{(migration)}(t)$

可隨時間分散。

SECTION

11. Deprecation 本質上是在買「演化自由度」

如果語言永遠不能破壞：

> $\Delta L_{(breaking)}=0$

則：

> $F_{(evolution)}$

演化自由度會逐步下降。

但如果語言可以隨意 break：

> $C_{(ecosystem)}$

會快速上升。

因此成熟語言需要：

> Deprecation
> +
> Tooling
> +
> Migration

作為中間機制。

本文定義：

> F_e
> =
> Evolutionary Freedom

則：

> F_e
> =
> f(
> M,
> T,
> K,
> E
>)

其中：

- M：migration mechanism；
- T：tooling；
- K：compatibility policy；
- E：ecosystem structure。

SECTION

12. JavaScript：歷史殘留可以被寫進標準

ECMAScript 是這篇最直接的案例。

ECMAScript 2026／2027 規格 Annex B 專門定義：

SECTION

Additional ECMAScript Features for Web Browsers

其中明確說明：

> 這些 legacy features 與 browser-host characteristics 具有一個或多個 undesirable characteristics；如果沒有 legacy usage，它們原本會被移除。
>
> 但 web browser：

> must support

這些 feature。

這幾乎是「歷史殘留」在語言規格中的正式案例。

SECTION

13. JavaScript Annex B：不是理想設計，而是現實契約

假設某 feature：

> F

今天從零設計：

> Q_(design)(F)<0

即不會被接受。

但因為：

> $N_{\text{(legacy usage)}}(F) \gg 0$

所以：

> $C_{\text{(removal)}}(F)$
> $\gg$
> $B_{\text{(cleanup)}}(F)$

於是：

> $F \in L$

繼續成立。

這就是：

SECTION

Ecosystem-Locked Residue

即：

> 被生態依賴鎖住的歷史殘留。

SECTION

14. Hyrum’s Law 在語言層的極端版本

對 web language 而言：

> $N_{\text{(consumers)}}$

大到幾乎無法完整枚舉。

所以任何：

- parsing behavior；
- coercion；
- error handling；
- property enumeration；
- timing；
- legacy syntax；

  都可能有人依賴。

  因此：

> Observable Language Behavior
> →
> Possible Contract

這可以視為 Hyrum’s Law 在語言層的放大版。

所以 JavaScript 的歷史怪異行為不能只用：

> 「當初設計錯了。」

解釋。

更完整是：

> 當初行為進入全球 web 後，後來的設計者失去了自由修改它的能力。

SECTION

15. Rust：把 incompatible evolution 放進 edition

Rust 提供一個非常有趣的對照。

Rust 1.0 後把：

SECTION

stability without stagnation

當成核心原則之一。

穩定 feature 會持續受到支援。

但語言仍然有時需要：

> $\Delta L_{(incompatible)}>0$

例如新增：

async
await
作為 keyword。

Rust 的解法不是讓所有舊 code 同一天失效，

而是：

SECTION

Edition

SECTION

16. Edition 是一種「時間分區語義」

每個 crate 選擇：

> e
> ∈
> {
> 2015,

因此同一個 compiler 可以對不同 crate 採用不同 edition-level syntax／rule。

這等於：

> L
> =
> bigcup⊠ L^((e⊠))

但要求：

> 不同 edition 的 crate 必須 seamless interoperate。

所以：

> Semantic Evolution

被部分隔離在 source interpretation layer，

而：

> Ecosystem Interoperability

保持。

這是非常有價值的語言治理設計。

SECTION

17. Rust 的真正創新不是「可以 break」，而是「把 break 變成 migration」

Rust Edition Guide 的 transition 流程包括：

cargo update
cargo fix --edition
edit Cargo.toml
cargo build / test
cargo fmt

甚至明確表示：

> 如果升級 edition 很困難，Rust 將其視為 bug。

這是一個重要思想：

> Breaking Change
> +
> Automated Migration
> ≠
> Unmanaged Breakage

也就是：

> 演化自由度可以靠 migration tooling 買回來。

SECTION

18. 五種語言演化策略

由以上案例，可以抽象出五種成熟語言策略。

SECTION

18.1 Preserve

舊行為永久保留。

典型：

> $F_{old} \in L$

SECTION

18.2 Deprecate

先警告，再移除。

> F
> →
> D(F)
> →
> ∅

SECTION

18.3 Layer

保留舊機制，再加新機制。

> L_(t+1)
> =
> L⊠+F_(new)

SECTION

18.4 Isolate

將 legacy behavior 放在特殊區域。

ECMAScript Annex B 就有這種味道。

SECTION

18.5 Version / Edition

讓不同語義世代共存，

再靠 interoperability 維持單一生態。

Rust edition 是代表案例。

SECTION

19. 語言歷史妥協體

因此本文定義：

SECTION

語言歷史妥協體

SECTION

Historically Constrained Language System

指：

> 一個成熟程式語言的現行形態，不只由當代語言設計原則決定，也受到既有程式、生態依賴、相容政策、ABI、runtime、歷史 feature 與遷移能力共同約束。

形式上：

> 【L(t)
> =
> L_(core)
> +
> H(t)
> +
> K(t)
> +
> E(t)
> +
> M(t)】

其中：

- L_(core)：核心語言設計；
- H(t)：Historical Residue；
- K(t)：Compatibility Constraints；
- E(t)：Ecosystem Dependencies；
- M(t)：Migration Mechanisms。

SECTION

20. 這不是說成熟語言等於缺陷品

這裡必須明確限制。

如果說：

> JavaScript 有 legacy behavior。

不能推出：

> JavaScript 是缺陷品。

如果說：

> C++ 很多舊 feature 不能移除。

不能推出：

> C++ 的設計全部失敗。

因為：

> Historical Constraint
> ≠
> Defect

有些歷史行為可能：

- 原本就是合理設計；
- 曾經是最佳方案；
- 現在仍有巨大價值；
- 只是後來出現更好的替代方案。

所以：

> 【Old
> nRightarrow
> Wrong】

SECTION

21. 語言成功本身會降低設計自由度

一個新語言：

> F_e≈1

設計者幾乎可以任意改。

成熟語言：

> N_(dependents)uparrow

則：

> F_edownarrow

若沒有 migration tooling。

因此可以概念化：

> F_e(t)
> =
> (M(t)+T(t))/(K(t)+E(t)+H(t))

其中：

- numerator：migration／tooling；
- denominator：compatibility、ecosystem、history。

這不是數值量表，

而是表示：

> 工具與遷移能力提高語言的演化自由度；生態依賴與歷史沉積降低演化自由度。

SECTION

22. 語言的「補償性完好」

第二篇提出 Compensated Integrity。

成熟語言同樣可能有：

- compiler warning；
- transpiler；
- polyfill；
- compatibility mode；
- adapter；
- runtime fallback；
- lint；
- code fix；
- migration tool。

這些都在補償：

> Old World
> ↔
> New World

所以：

> L_(effective)

並不只靠 grammar 維持。

它也靠：

> C_L

即語言層補償機制。

SECTION

23. 語言的「補償凝固」

一些 compatibility workaround 一旦進入：

- compiler；
- runtime；
- standard；
- browser；
- library；

  並被大量依賴，

  就可能：

> C_L
> →
> L_(effective)

這與第六篇：

> C⊠→ A_(t+1)

是同一結構。

只是作用對象從 application architecture 換成 language ecosystem。

SECTION

24. 語言的「複雜度轉移」

第七篇提出：

> Repair
> nRightarrow
> Total Complexity Reduction

語言演化也一樣。

例如 Rust edition：

> C_(global breakage)downarrow

但增加：

> C_(edition machinery)
> +
> C_(migration tooling)

Python deprecation：

> C_(sudden breakage)downarrow

但增加：

> C_(deprecation management)

Java binary compatibility：

> C_(consumer rebuild)downarrow

但增加：

> C_(compatibility rules)

這些通常是好的複雜度轉移。

因為複雜度被移到更可治理的位置。

SECTION

25. 語言設計的真正目標不再只是「簡潔」

對成熟語言，優化函數應該至少包含：

> Q_L
> =
> f(
> S,
> E,
> K,
> M,
> T,
> P
>)

其中：

- S：simplicity；
- E：expressiveness；
- K：compatibility；
- M：migration；
- T：toolability；
- P：performance / predictability。

所以：

> minSyntax Complexity

不是充分目標。

一個非常漂亮但每三年 break 整個 ecosystem 的語言，

未必比：

> 稍微複雜，但能持續演化二十年

更成功。

SECTION

26. 「歷史妥協」其實是成功的副產品之一

如果一個語言沒有使用者：

> $H \approx 0$

很容易。

沒有 legacy。

沒有 compatibility。

沒有 ABI。

沒有 ecosystem breakage。

但這不是因為語言特別成功，

而可能只是：

> 沒有人依賴它。

所以：

> Huparrow

有時恰恰是：

> Success Historyuparrow

的副產品。

問題不是：

> 有沒有歷史。

而是：

> 歷史是否被治理。

SECTION

27. 語言也需要 Structural Metabolism

第八篇提出：

SECTION

Structural Metabolism

對語言同樣適用。

語言需要：

proposal
→ experiment
→ standardize
→ observe
→ deprecate
→ migrate
→ retire / retain
C++ 的 Technical Specification／experimental 機制、

Python 的 PEP／deprecation、

TC39 的 proposal process、

Rust edition／cargo fix，

都可以視為不同形式的：

SECTION

Language Metabolism

目標不是讓語言永遠不變，

而是：

> 讓語言有能力消化自己的歷史。

SECTION

28. 新語言為什麼常看起來比較乾淨？

這也解釋一個常見現象。

新語言：

> Hdownarrow

> Kdownarrow

> E_(legacy)downarrow

所以設計者能：

- 使用新研究；
- 重新定義 defaults；

- 刪掉舊假設；
- 不承擔歷史 ABI；
- 不保留舊 syntax。

因此：

> Q_(apparent simplicity)uparrow

但如果它成功活二十年：

> H,K,Euparrow

它也會開始面臨同一問題。

所以：

> 新語言的乾淨，一部分來自更好的設計；另一部分則來自它還沒有歷史。

SECTION

29. 比較語言時必須控制「年齡與成功」變量

因此拿：

30 年成熟語言

和：

3 年新語言

比較「語言乾淨程度」，

並不完全公平。

因為：

> T_(history)

以及：

> N_(ecosystem)

差異巨大。

更合理的比較是：

> 這個語言在自己的歷史與生態規模下，如何治理演化？

而不是：

> 哪個 grammar 看起來最漂亮？

SECTION

30. FPL／新語言設計的一個重要教訓

這對 FPL 以及任何新語言都非常重要。

早期最容易犯的錯是：

> 為了未來兼容，把所有可能性都提前塞進 1.0。

這會造成：

> C_(premature)uparrow

另一個極端則是：

> 1.0 隨便定，之後想改就改。

如果 FPL 真正被大量專案、Agent、IDE 使用，

那麼：

> K_(FPL)uparrow

之後破壞成本會快速上升。

所以真正需要從第一天設計的是：

SECTION

Evolution Mechanism

而不是預測所有未來 feature。

SECTION

31. FPL 應該從 Rust／Python 學什麼？

至少可以抽象出：

SECTION

31.1 Versioned Semantics

> FPL^((v))

允許明確語義版本。

SECTION

31.2 Automated Migration

fpl migrate
能自動轉換常見 incompatible change。

SECTION

31.3 Deprecation Metadata

```
deprecated:
 since: 2.1
 removal_target: 4.0
 replacement: authority.v2
```

SECTION

31.4 Cross-Version Interoperability

不同版本 IR 應盡可能共存。

SECTION

31.5 Compatibility Impact Analysis

每次語言規格改動前：

> Impact(Δ L)

必須分析 existing projects。

這比「我們今天把語法設計完美」更重要。

SECTION

32. Dynamic MSSP 也可以治理語言本身

這裡會出現一個有趣的自指。

MSSP/FPL 原本用來治理：

> Software Architecture

但 FPL 自己也是：

> Evolving Software Language

所以它也需要：

- declared semantics；
- effective usage；

- compatibility；
- historical residue；
- migration；
- deprecation；
- tooling evidence。

也就是：

> FPL 最後也需要用自己的思想治理自己。

這不是矛盾。

而是成熟語言不可避免的 reflexive governance。

SECTION

33. 命題 9：語言歷史妥協命題

本文提出系列第九命題。

SECTION

語言歷史妥協命題

對成熟且具有大規模生態的程式語言 L：

> 【L(t)
> ≠
> L_(ideal)(t)】

其現行形態由：

> 【L(t)
> =
> L_(core)
> +

共同形成。

因此：

> 【Language Success
> →
> Compatibility Pressure】

而：

> 【Compatibility Pressure
> →
> Reduced Unmanaged Evolution Freedom】

除非：

> M(t)+T(t)

即 migration mechanisms 與 tooling 同步增強。

SECTION

34. 強版本：成功語言必須治理自己的歷史

因此成熟語言真正的能力，不只是：

> 能不能新增 feature。

而是：

> 能不能在不毀掉已存在世界的情況下改變自己。

這可以稱為：

SECTION

Language Evolution Capacity

令：

> E_L
> =
> Language Evolution Capacity

則其核心不是：

> Feature Velocity

而是：

> E_L
> =
> f(
> Compatibility,
> Migration,
> Tooling,
> Governance,
> Ecosystem Feedback
>)

SECTION

35. 結論：成熟程式語言不是「缺陷品」，而是被歷史約束的活語言

回到最初問題：

> 大部分程式語言會不會也是很多缺陷組成、外表功能完好的東西？

本文認為：

> 這個說法太強，也不夠準確。

成熟語言更合理的描述是：

SECTION

歷史約束的演化結構

SECTION

Historically Constrained Evolving Structure

它確實可能包含：

- 過去今天不會再選的 feature；
- legacy semantics；
- compatibility mode；
- deprecation；
- redundant mechanisms；
- historical syntax。

但這些東西的存在，不一定是因為語言設計者無能。

很多時候：

> Removal Cost
> >
> Design Purity Benefit

所以它們被保留。

因此：

> 【Successful Language
> ≠
> Perfect Language】

但同時：

> 【Successful Language
> ≠
> Defective Language】

更接近：

> 【Successful Language
> =
> Core Design
> +
> Accumulated History
> +
> Compatibility
> +
> Migration Governance】

這也是為什麼 C++、Java、Python、JavaScript 會有非常不同、卻都合理的歷史妥協方式。

Rust editions 則提供另一條值得注意的路：

> 不要假裝歷史不存在；把歷史分區，再提供機器可輔助的遷移通道。

因此對未來 FPL、AI-native language 與任何新程式語言而言，最重要的問題可能不只是：

> 「我們現在的語言設計好不好？」

而是：

> 「如果它真的成功活二十年，我們準備讓它怎麼老？」

下一篇將正式進入 AI-native software development：

SECTION

〈AI 寫得越快，屎山會長得越快嗎？AI 原生開發的新技術債〉

也就是：

> 當 code generation cost 急劇下降，但 verification、architecture understanding 與 governance throughput 沒有同比下降時，前八篇描述的補償、凝固、侵蝕與複雜度流，是否會被 AI 大幅加速？

SECTION

參考文獻

- ISO C++ / WG21. SD-10: Language Evolution (EWG) Principles. Standard C++. https://isocpp.org/std/standing-documents/sd-10-language-evolution-principles
- Sutter, H. et al. (Re)affirm Design Principles for Future C++ Evolution, P3466R1. ISO C++ WG21.
- ISO C++ Direction Group. Directions for ISO C++, P0939R0.
- Oracle. The Java Language Specification, Java SE 26 Edition, Chapter 13: Binary Compatibility. https://docs.oracle.com/en/java/javase/26/docs/specs/jls/jls-13.html
- Python Software Foundation. PEP 387 – Backwards Compatibility Policy. https://peps.python.org/pep-0387/
- Ecma International / TC39. ECMAScript 2026 Language Specification, Annex B: Additional ECMAScript Features for Web Browsers. https://tc39.es/ecma262/2026/multipage/additional-ecmascript-features-for-web-browsers.html
- Rust Project. The Rust Edition Guide — What Are Editions?. https://doc.rust-lang.org/edition-guide/editions/
- Rust Project. Transitioning an Existing Project to a New Edition. https://doc.rust-lang.org/stable/edition-guide/editions/transitioning-an-existing-project-to-a-new-edition.html

- Wright, H. Hyrum’ s Law. https://www.hyrumslaw.com/
- Stroustrup, B. The C++ Programming Language, Second Edition — Preface. https://stroustrup.com/2nd_pref.html
- Neo.K / EveMissLab.《表觀完好系統》系列第 1–8 篇，以及 MSSP / FPL 既有架構與程式語言設計研究。