設計一致性、相容性與社群演化：語言何時應堅持原則，何時應接受歷史？

Design Coherence, Compatibility, and Community Evolution: When Should a Language Defend Its Principles, and When Should It Accept History?

v1.0 / 2026-07-30 / Neo.K / 公開版／第二部核心風格原型正式論文

https://thisoneisneok.com/html/papers/pldst-010.html

英文名稱： Design Coherence, Compatibility, and Community Evolution: When Should a Language Defend Its Principles, and When Should It Accept History?

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-010

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／第二部核心風格原型正式論文

SECTION

摘要

程式語言進入長期使用後，設計者與共同體會持續面對一個無法永久迴避的衝突：

> 應該為了更一致、更安全、更清楚的設計修正歷史，還是應該保護既有程式、工具、組織知識與使用者信任？

若永遠優先原則，語言可能頻繁破壞程式、分裂生態並耗盡使用者；若永遠優先相容，早期偶然、錯誤預設與不理想介面可能被永久保留，形成規格膨脹、教學負擔與設計化石。

Go 將 Go 1 相容承諾視為最重要的設計選擇之一，並在 Go 1.21 以語言版本、GODEBUG、中央文件與 Runtime metrics 進一步保存「規格允許改變、但真實程式仍可能依賴」的舊行為。Python 的 PEP 387 將語法、行為、C API、例外及公共介面納入相容政策，要求不相容變更具有高收益／破壞比、經過至少兩年的棄用期，並以 Soft Deprecation 保留「不再鼓勵，但也不預定移除」的 API。Rust 以「穩定而不停滯」為核心承諾：一旦功能進入 Stable，原則上持續支援；真正需要不相容表面變更時，透過 opt-in Edition、跨 Edition 互操作與 cargo fix --edition 將破壞限制在可選遷移中。C++ 由 WG21 在廣泛硬體、產業與既有程式碼約束下演化，其正式原則要求與舊 C++ 的相容應預設可用，但相容也使語言長期保存多代構造。ECMAScript 的 Annex B 更直接把部分具有「不理想特性」的 Web Legacy 行為列入規範：若沒有大量既有網頁依賴，本可刪除，但瀏覽器仍必須支援。Swift 則同時發展 Source compatibility、ABI stability、Module stability、Library evolution、Upcoming Feature Flags、Code migration 與社群相容測試套件，展示相容不是單一布林值，而是一組跨編譯器、來源、模組與二進位邊界的工程制度。[R1][R2][R3][R4][R5][R6]

本文提出 一致性—相容性—演化模型（Coherence–Compatibility–Evolution Model, CCEM）：

> cal-E
> =
> (
> Q,
> C,
> B,
> M,
> G,
> L,
> J
>)

其中：

- Q：Coherence，一致性與設計品質；
- C：Compatibility，相容性向量；
- B：Breakage，破壞範圍與代價；
- M：Migration，遷移機制；
- G：Governance，決策與例外治理；

- L：Legacy，歷史負擔與化石；
- J：Legitimacy，社群正當性與信任。

本文區分四種一致性：

- 概念一致性；
- 表面一致性；
- 行為一致性；
- 制度一致性。

並區分七種相容性：

- 來源相容；
- 二進位／ABI 相容；
- 行為相容；
- 資料與序列化相容；
- 工具與建置相容；
- 生態與依賴相容；
- 組織知識相容。

核心命題為：

> 【向後相容
> not⇒
> 舊設計仍被認為良好】

以及：

> 【修正不一致
> not⇒
> 破壞就是正當的】

成熟語言不應把相容性當成絕對禁令，也不應把「設計更漂亮」視為破壞使用者的充分理由。更合理的判準是：

> 只有當不一致造成明確且持續的安全、正確性、可理解性或演化阻塞，而新設計具有足夠收益、可限制破壞範圍、可提供工具化遷移、可經社群驗證並保存長期信任時，破壞才可能正當。

關鍵詞： 程式語言設計、向後相容、設計一致性、語言演化、棄用、Edition、ABI、Legacy、治理、PLDST

SECTION

一、概念一致性 Q_c

概念一致性指：

- 相似問題由相似機制解決；
- 核心概念能覆蓋多個場景；
- 不需要大量例外；
- 使用者能從已知規則推導新情況；
- 安全與資源模型沒有互相矛盾的逃生慣例。

例如：

所有資源都遵循一致所有權
所有錯誤都使用一致傳播模型
所有集合都遵循相似迭代抽象
概念一致不要求語法完全相同，而要求深層規則可組合。

SECTION

二、表面一致性 Q_s

表面一致性包括：

- 命名；
- 關鍵字；

- 宣告順序；
- API 風格；
- 錯誤訊息；
- 格式；
- 可見性；
- 泛型語法。

  它能降低記憶負擔，但過度追求表面對稱，可能迫使不同概念使用相同語法而增加歧義。

SECTION

三、行為一致性 Q_b

行為一致性回答：

- 相似操作是否具有相似副作用？
- 同名 API 是否具有相近錯誤與複雜度？
- 不同平台是否遵守相同語義？
- 新 Compiler 是否保留既有程式可觀察結果？
- Debug／Release 是否維持可接受差異？

  這是相容性政策經常真正保護的層級。

SECTION

四、制度一致性 Q_g

制度一致性指：

- 相似提案使用相似審查標準；
- 例外具有可說明理由；

- Stable、Experimental、Deprecated 等狀態清楚；
- 不同公司與人物不因地位而獲得完全不同准入門檻；
- 已公布承諾不被任意撤回。

  語言技術一致，但治理任意，同樣會失去社群信任。

SECTION

五、一致性向量

> Q
> =
> (
> Q_c,
> Q_s,
> Q_b,
> Q_g
>)

一個語言可以：

- 概念一致但表面特殊；
- 表面整齊但行為例外多；
- 技術一致但治理不透明；
- 行為穩定但核心概念已化石化。

SECTION

六、來源相容 C_s

舊來源碼能否由新工具鏈：

- 編譯；

- 型別檢查；
- 連結；
- 執行；

  而不需要修改。

  來源相容仍需區分：

  完全無修改
  只需警告
  可由工具自動修改
  需人工遷移
  僅特定語言模式

SECTION

七、二進位與 ABI 相容 C_a

舊 Binary 能否與新：

- Runtime；
- 標準程式庫；
- Framework；
- Dynamic library；
- Compiler-produced module；

  共同運行，而不重新編譯。

  ABI 相容牽涉：

- Calling convention；
- Name mangling；
- Layout；
- Vtable；

- Symbol；
- Exception；
- Metadata；
- Inlining boundary。

  Swift 將 ABI stability、Module stability 與 Library evolution 明確區分，正是因為它們並非同一保證。[R6]

SECTION

八、行為相容 C_b

舊程式即使仍能編譯，也可能因：

- Bug fix；
- 安全預設；
- 排序；
- Protocol；
- 例外；
- Parser；
- GC；
- Timing；

  改變結果。

  因此：

> SourceCompatible
> not⇒
> BehaviorCompatible

Go 1.21 的 GODEBUG 機制即處理「規格允許改，但實際程式可能依賴舊行為」的灰區。[R1]

SECTION

九、資料相容 C_d

包括：

- Serialized format；
- Database Schema；
- Wire protocol；
- Hash；
- Enum；
- Timestamp；
- Numeric representation。

  語言與 Library 改變可能破壞保存多年或跨版本交換的資料，即使來源和 ABI 完全不變。

SECTION

十、工具與建置相容 C□

包括：

- Build system；
- Package manifest；
- Linter；
- Formatter；
- IDE；
- Macro；
- Generated code；

- Compiler plugin；
- CI flag；
- Test discovery。

工具往往是有效語言的一部分。只保證 Parser 接受舊語法，不足以保證專案可建置。

SECTION

十一、生態與依賴相容 C_e

包括：

- 舊 Library；
- 依賴版本；
- Feature；
- Package resolver；
- Plugin；
- 多 Compiler；
- 多平台。

Rust Edition 強調不同 Edition crate 可互操作，避免語言修正將 Package 生態切成不相容島嶼。[R3]

SECTION

十二、組織知識相容 C_o

包括：

- 教材；
- Code review 規範；
- 人才訓練；

- Debug 經驗；
- Runbook；
- 安全審計；
- 內部 Framework；
- 招聘與維護能力。

  語言破壞不只重寫程式，也可能使多年組織知識折舊。

SECTION

十三、相容向量

> C
> =
> (
> C_s,
> C_a,
> C_b,
> C_d,
> C□,
> C_e,
> C_o
>)

「此變更向後相容」若未指明維度，就是不完整聲明。

SECTION

十四、活躍設計 Active Design

仍被認為合理，且推薦新程式使用。

SECTION

十五、相容性化石 Compatibility Fossil

被保留是因為既有程式依賴，而非現代設計仍偏好。

ECMAScript Annex B 的說明極為直接：相關 Web Legacy 功能具有不理想特性，若沒有大量既有網頁使用，本應從規格刪除；但瀏覽器仍須支援。[R5]

SECTION

十六、軟性棄用 Soft Deprecation

Python PEP 387 將 Soft Deprecation 定義為：

- 不再建議新程式使用；
- 仍保持文件與測試；
- 不再積極擴展；
- 不自動意味預定移除。[R2]

它適合：

- 移除收益不足；
- 既有使用仍安全；
- 替代方案較佳；
- 不值得強迫全生態遷移。

SECTION

十七、硬性棄用 Hard Deprecation

包含：

- 警告；

- 替代路徑；
- 最低等待時間；
- 最終移除目標；
- 遷移文件。

硬性棄用是破壞流程，不只是文件標籤。

SECTION

十八、事故性穩定 Accidental Stabilization

某項行為可能因：

- 實作長期存在；
- 使用者依賴；
- 測試固定；
- 文件模糊；
- 工具觀察；

形成事實契約，即使從未被正式設計。

成熟共同體需決定：

- 正式承認；
- 限時遷移；
- 以版本旗標保存；
- 判定為未定義並允許修正。

SECTION

十九、實驗性負擔 Experimental Debt

Preview、Nightly、Provisional、Feature flag 的價值，是在功能永久承諾前取得真實證據。

若實驗狀態：

- 無期限；
- 大量依賴；
- 無退出政策；
- 被生產環境當 Stable；

  就會形成影子相容負擔。

SECTION

二十、直接修改成本 B_m

包括：

- 修改來源；
- 重建；
- 重新測試；
- 部署；
- 資料轉換。

SECTION

二十一、語義風險 B_s

自動遷移後程式可能：

- 編譯成功但行為改變；
- 對錯誤的假設失效；
- 效能變化；

- 安全邊界變化；
- 產生新的歧義。

SECTION

二十二、生態協調成本 B_e

多層依賴需要：

- 先更新底層；
- 等待 Release；
- 支援新舊雙版本；
- 處理 Diamond dependency；
- 更新 CI 與文件。

SECTION

二十三、信任成本 B□

若使用者認為：

- 承諾不可靠；
- 升級風險未知；
- 每次版本皆需全面重測；
- 例外由權力任意決定；

他們可能：

- 停留舊版本；
- Fork；
- 避免採用新功能；

- 離開生態。

SECTION

二十四、破壞向量

> B
> =
> (
> B_m,
> B_s,
> B_e,
> B□
>)

設計美學收益若不足以支付這些成本，就不構成正當破壞。

SECTION

二十五、安全必要性

若舊行為：

- 產生可利用漏洞；
- 無法合理修補；
- 破壞資料；
- 違反外部安全標準；
- 使安全預設無法建立；

破壞相容較可能正當。

但仍需：

- 風險證據；

- 暫時相容開關；
- 版本溝通；
- 遷移工具；
- 緊急例外程序。

SECTION

二十六、語義錯誤

若規格或實作行為：

- 自相矛盾；
- 無法跨實作一致；
- 造成不可預測結果；
- 阻止後續功能；
- 使基本推理失效；

可以考慮修正。

「我更喜歡另一種語法」通常不足。

SECTION

二十七、規模阻塞

某些早期設計在小型系統可接受，卻在大型生態造成：

- 全域命名衝突；
- Build 不可擴展；
- ABI 無法演化；
- Library 無法新增 Enum case；

- 安全工具無法判斷；
- 模組邊界僵化。

此時修正可能是生態持續成長的前提。

SECTION

二十八、收益—破壞比

Python PEP 387 使用「Benefit to breakage ratio」作為基本政策語言。[R2]

可表示為：

> J(Δ)
> =
> (Safety+
> Correctness+
> Coherence+
> FutureCapacity)/(Migration+
> SemanticRisk+
> EcosystemCost+
> TrustCost)

只有 J(Δ) 足夠高，且沒有成本更低的替代方案，才進入破壞流程。

SECTION

二十九、警告

好的警告應：

- 在實際移除前足夠早；
- 指出替代；
- 定位使用點；

- 可在 CI 升格；
- 不產生無法處理的噪音。

SECTION

三十、自動修正

例如：

- cargo fix --edition；
- Swift Migrator／Fix-it；
- Formatter rewrite；
- Codemod；
- Compiler suggestion。

自動修正應區分：

機械等價
可能改變行為
需人工確認
無法自動修正

SECTION

三十一、版本化行為

Go 1.21 讓 GODEBUG 預設值依主套件 go.mod 中的 Go 版本選擇，使新 Toolchain 能保留舊版本行為。[R1]

此模式可表示：

```
Behavior
=
f(
ToolchainVersion,
DeclaredLanguageVersion,
CompatibilityFlags
)
```

優勢：

- 新 Compiler 可運行舊行為；
- 升級 Toolchain 與升級語義分離；
- 可用 metrics 觀察舊行為依賴。

風險：

- 行為矩陣擴大；
- 測試需涵蓋版本；
- 相容開關可能長期存在。

SECTION

三十二、Edition

Rust Edition 將部分不相容表面變更：

- 置於 opt-in 邊界；
- 允許同一依賴圖內跨 Edition 互操作；
- 使用 Lint 和 cargo fix --edition 協助遷移；
- 保留 Stable feature 支援。[R3]

它不是 Fork，也不是傳統完全不相容 major version。

SECTION

三十三、Upcoming Feature Flags

Swift 允許專案逐項啟用預定進入下一重大語言模式的 Source-breaking change，並要求提案描述 Source compatibility 影響。[R6]

這使：

- 語言新行為可提前測試；
- 遷移可按 Target 分段；
- 團隊可在正式切換前準備；
- 破壞不必在單一版本瞬間完成。

SECTION

三十四、相容測試套件

Swift Source Compatibility Test Suite 將真實開源專案納入持續整合，用來在 Compiler 變更合併前發現來源相容回歸。[R6]

這是一個重要原則：

> 相容性不能只依規格推論，也需要以真實生態驗證。

SECTION

三十五、承諾優先型

代表：

- Go。

特徵：

- 長期 Compatibility promise；
- 變更高度保守；
- 新功能以相容方式加入；
- 使用版本化行為與 GODEBUG 處理灰區；
- 升級應盡可能「無聊」。

優勢：

- 生態信任；
- 升級容易；
- 企業長期採用；
- 工具鏈統一。

風險：

- 舊預設長期保存；
- 行為版本矩陣；
- 難以徹底修正核心設計；
- 新功能需繞過歷史限制。

SECTION

三十六、棄用治理型

代表：

- Python。

特徵：

- 將 Public API 與 Private／Provisional 分開；
- 不相容變更需高收益；
- 至少兩年棄用；
- Steering Council 可在極端情況例外；
- Soft Deprecation 不承諾移除。

優勢：

- 兼顧演化與使用者；

- 政策透明；
- 可區分「不推薦」與「預定刪除」；
- 例外有治理主體。

風險：

- 警告長期累積；
- 標準程式庫維護負擔；
- 動態行為可能難以完整警告；
- 大型生態遷移速度不一致。

SECTION

三十七、穩定而不停止型

代表：

- Rust。

特徵：

- Stable feature 長期支援；
- Nightly／Feature gate 實驗；
- Edition 管理部分不相容表面；
- 跨 Edition 互操作；
- 工具化遷移。

優勢：

- 穩定核心與快速創新共存；
- 破壞由使用者 opt-in；

- 生態不因 Edition 分裂；
- 實驗未成熟前不形成承諾。

風險：

- Stable accidental behavior 難修正；
- Edition 與 Feature gate 制度複雜；
- Migration Lint 無法處理所有語義；
- Standard Library API 仍受 SemVer 與生態限制。

SECTION

三十八、歷史疊加型

代表：

- C++。

特徵：

- 強調與舊 C++ 的持續相容；
- 國際標準與多實作；
- 以新增一般機制現代化；
- 舊構造多數仍保留；
- Library 與 Core 分組治理。

優勢：

- 巨大既有程式與產業投資持續可用；
- 長期平台；
- 多領域與多供應商；

- 漸進採用現代特徵。

  風險：

- 特徵疊加；
- 教學需要區分多代風格；
- 更安全預設難以取代歷史預設；
- ABI 與 Library 相容可能阻礙修正；
- 一致性常需以 Guideline 而非移除達成。

  WG21 的 Language Evolution 原則將與較舊 C++ 的順暢相容列為預設要求；這是一種制度性保守，而非單一人物偏好。[R4]

SECTION

三十九、Web 現實保存型

代表：

- ECMAScript。

  特徵：

- 全球 Web 本身就是不可集中遷移的程式庫；
- 瀏覽器必須執行大量無法更新的舊頁面；
- Legacy 行為以 Annex B 隔離；
- 新功能透過 TC39 Stage、實作與 Test262 逐步成熟；
- 對新程式明確不推薦 Legacy。

  優勢：

- Web 長期可用；
- 多瀏覽器互操作；

- 新舊頁面共存；
- Legacy 與 Core 在規格中有邊界。

風險：

- 無法真正刪除廣泛使用的錯誤行為；
- Parser 與語義需保存特殊規則；
- 安全與一致性改進受 Web 相容限制；
- Annex B 仍是瀏覽器實作成本。

SECTION

四十、分層穩定與遷移型

代表：

- Swift。

特徵：

- Source compatibility；
- Language compatibility mode；
- ABI stability；
- Module stability；
- Library evolution；
- Upcoming Feature Flags；
- Migrator；
- 生態相容測試。

優勢：

- 可精確區分不同相容承諾；
- Binary framework 可獨立演化；
- 破壞性功能可提前 opt-in；
- 真實專案測試回歸。

風險：

- 多層模式複雜；
- Library evolution 可能改變性能與可用設計；
- @frozen 等選擇會用未來演化自由換取當前效率；
- Source、ABI 與 Module 保證容易被使用者混淆。

SECTION

四十一、不安全預設不能修正

若相容政策使語言無法：

- 改成安全預設；
- 禁止危險轉換；
- 改善錯誤處理；
- 收緊 Null；
- 關閉隱式行為；

歷史便直接阻礙安全。

SECTION

四十二、所有舊行為都成為教學負擔

新手不只需學現代推薦方式，也要理解：

- 舊語法；
- 舊 API；
- 相容模式；
- Legacy 架構；
- 舊錯誤行為。

即使「可以不使用」，閱讀舊程式仍需掌握。

SECTION

四十三、規格與實作矩陣爆炸

每個新功能都要與：

- 舊版本；
- 舊 ABI；
- 舊 Compiler；
- 舊 Library；
- Compatibility flag；
- Legacy mode；

交互測試。

SECTION

四十四、設計一致性只能靠社群規範

語言無法移除不理想構造時，常依賴：

- Linter；
- Style guide；

- Safe subset；
- Modern profile；
- Framework；
- Code review。

  這可以有效，但也意味語言規格與實際推薦語言分裂。

SECTION

四十五、只為美學

僅因：

- 語法更漂亮；
- 關鍵字更一致；
- 個人偏好；
- 重新命名；

  而要求大規模遷移，通常正當性不足。

SECTION

四十六、沒有真實資料

若未測量：

- 使用率；
- 受影響專案；
- 依賴圖；
- 生產行為；
- 自動修正率；

- 性能影響；

  就無法合理估計破壞。

SECTION

四十七、把成本推給無權使用者

語言共同體決定修正，卻由：

- Library maintainer；
- 企業舊系統；
- 無資源開源作者；
- 下游使用者；

  支付全部遷移成本，會削弱正當性。

SECTION

四十八、沒有雙軌與回退

若新版本：

- 沒有 Compatibility mode；
- 沒有警告期；
- 沒有遷移工具；
- 沒有回退；
- 沒有跨版本互操作；

  破壞半徑會放大。

SECTION

四十九、把反對者寫成落後

使用者可能反對破壞，不是因為不理解新設計，而是因為：

- 資料無法重建；
- 驗證成本高；
- 法規要求；
- 裝置無法升級；
- 上游已停止維護；
- 大量下游依賴。

成熟治理需要理解真實成本。

SECTION

五十、程序正當性

包括：

- 公開提案；
- 替代方案；
- 相容分析；
- 實作經驗；
- 反對意見；
- 決策記錄；
- 例外理由。

SECTION

五十一、代表正當性

誰參與決策？

- 語言團隊；
- Compiler；
- Library；
- 工具；
- 企業；
- 獨立使用者；
- 教育；
- 安全；
- 不同平台。

「社群決定」不能只代表最能參加會議的人。

SECTION

五十二、結果正當性

即使程序完整，結果仍需：

- 解決真問題；
- 不產生更大傷害；
- 有可用替代；
- 具有工程證據；
- 能被實際採用。

SECTION

五十三、補償正當性

若共同體要求使用者遷移，也應提供：

- 工具；
- 文件；
- 支援；
- 時間；
- 相容層；
- 測試；
- 資源。

SECTION

五十四、信任方程

可啟發式表示：

> Trust_(t+1)
> =
> Trust⊠
> +
> Transparency
> +
> Predictability
> +
> Support
> -
> Surprise
> -
> ArbitraryException
> -
> UnpaidBreakage

它不是精確社會科學量表，而是制度檢查框架。

SECTION

五十五、堅持原則

較適合堅持原則的情況：

- 功能尚未 Stable；
- 使用量低；
- 可安全撤回；
- 問題涉及安全不變量；
- 現有設計阻塞核心模型；
- 可由工具完整遷移；
- 破壞被限制在 opt-in Profile。

SECTION

五十六、接受歷史

較適合接受歷史的情況：

- 全球大量不可更新程式依賴；
- 行為雖不理想但仍可安全支援；
- 修正收益有限；
- 遷移無法自動；
- 資料或 ABI 已長期保存；
- 相容層成本可控；
- 可將 Legacy 明確隔離。

SECTION

五十七、雙層處理

常見最佳方案不是二選一，而是：

舊行為保留
+
新程式預設新行為
+
明確版本／Edition／Profile
+
遷移工具
+
跨層互操作

SECTION

五十八、刪除判準

一項歷史功能可考慮移除，需回答：

是否有明確傷害？
是否有可靠替代？
使用率多高？
能否自動遷移？
資料與 ABI 是否受影響？
是否已充分警告？
生態是否已有時間反應？
是否有例外與回退？
決策者是否承擔支援？

SECTION

五十九、一致性指紋

Conceptual coherence
Surface uniformity
Behavioral consistency
Governance consistency
Tolerance for exceptions
Tolerance for profiles

SECTION

六十、相容性指紋

Source compatibility
ABI compatibility
Behavior compatibility
Data compatibility
Tool compatibility
Ecosystem compatibility
Organizational compatibility

SECTION

六十一、演化指紋

Deprecation period
Soft deprecation
Feature gates
Edition／language mode
Migration tools
Compatibility flags
Cross-version interop
Legacy isolation
Removal willingness
Exception authority

SECTION

六十二、設計師比較問題

- 他認為何種不一致不可接受？
- 他願意為一致性破壞多少舊程式？
- 他保護哪一種相容性？
- 哪些相容性可犧牲？
- 是否區分 Stable、Experimental 與 Private？
- 是否提供自動遷移？

- 是否接受版本化語義？
- 是否允許 Legacy profile？
- 誰能批准例外？
- 使用者如何參與或反對？
- 舊功能保留是認同還是化石？
- 破壞後由誰支付維護與支援？

SECTION

六十三、不能只給「保守／激進」分數

一個共同體可能：

- 對 Source 極保守；
- 對 Private API 激進；
- 對 ABI 保守；
- 對工具格式彈性；
- 對安全漏洞快速破壞；
- 對 Legacy 行為永久隔離。

必須保留多維向量。

SECTION

六十四、輸入

```
designer_or_governance_body
language
version_range
feature_or_change
compatibility_policy
proposal
```

migration_tools
ecosystem_evidence

SECTION

六十五、分析管線

重新網路搜尋
→ 一致性目標抽取
→ 相容性向量
→ Public／Private／Experimental 邊界
→ 破壞範圍
→ Legacy 分類
→ 遷移與回退
→ 治理與例外權
→ 生態證據
→ 正當性分析
→ 第二輪事實校對
→ 風格報告

SECTION

六十六、JSON 雛形


```
{
  "change": "edition-gated syntax change",
  "coherence_goal": "remove an ambiguous legacy form",
  "compatibility": {
    "source": "breaking only after opt-in",
    "binary": "not split by edition",
    "behavior": "version-dependent",
    "ecosystem": "cross-edition interoperation"
  },
  "migration": {
    "warning": true,
    "automatic_fix": true,
    "manual_review": "sometimes required",
    "rollback": "retain old edition"
  },
  "governance": {
    "proposal_required": true,
    "stable_feature_removed": false
  },
  "legacy_status": "supported but not preferred"
}
```

SECTION

六十七、SKILL 禁止事項

不得：

- 把 Source 相容寫成行為相容；
- 把 ABI stability 寫成 Library 可任意改變；
- 把 Soft Deprecation 寫成已排定移除；
- 把 Edition 寫成生態 Fork；
- 把 GODEBUG 寫成所有變更都能永久回退；
- 把 Annex B Legacy 功能寫成現代推薦設計；
- 把 WG21 的相容偏好歸因於單一設計者；
- 把 Swift Source compatibility、ABI stability、Module stability 混成同一概念；
- 只用編譯成功判定相容；
- 忽略資料、工具與組織知識；
- 把美學一致當成足夠破壞理由；
- 忽略受影響者的遷移成本。

SECTION

六十八、相容性無法完整測量

真實生態包含：

- 私有程式；
- 未發布依賴；
- 動態行為；

- 生成程式；
- 未文件化使用；
- 平台特例。

任何相容測試都只是樣本。

SECTION

六十九、Bug fix 與行為破壞邊界模糊

使用者可能依賴 Bug；修正後：

- 規格更正確；
- 真實程式仍壞掉。

需區分：

規格相容
實作相容
生態相容

SECTION

七十、安全例外可能被濫用

治理者可以用「安全」合理化任意破壞。需要：

- 公開威脅模型；
- 風險證據；
- 替代方案；
- 最小破壞；
- 後續審查。

SECTION

七十一、相容政策也會演化

Go、Python、Rust、C++、ECMAScript 與 Swift 的政策及工具都可能改變。後續個案必須重新查核當時版本。

SECTION

七十二、社群不是單一利益主體

新使用者、舊系統、Compiler、Library、企業、安全團隊與教育者對相容有不同需求。不存在無成本共識。

SECTION

七十三、Go

已重新核對 Go 1 Compatibility 與 Go 1.21 相容文章：

- Go Team 明確將相容性視為 Go 1 最重要的設計決定之一；
- Go 1 的主要承諾是 Source-level compatibility；不同 Release 間不保證已編譯 Package 的 Binary compatibility，通常需要重新編譯；
- Go 1.21 對「規格允許但仍可能破壞」的改變建立 GODEBUG 退出機制；
- 新增的相容 GODEBUG 一般至少維持兩年／四個 Go Release，但某些可更久；
- GODEBUG 預設可由主套件 go.mod 的語言版本控制；
- Runtime metrics 可觀察非預設相容行為；
- 並非所有改變都必然能提供 GODEBUG。

  本文沒有把 Go 相容承諾寫成零例外的永久行為凍結。

SECTION

七十四、Python

已重新核對當前 Active 的 PEP 387：

- 政策涵蓋語法、行為、C API、公共名稱與型別、回傳值、副作用與例外等；

- Private、明示 Private 與 Provisional API 不具有相同保證；
- 不相容變更需有足夠高的 Benefit／breakage ratio；
- 一般棄用期至少兩年；
- Steering Council 可在危險、嚴重錯誤或無合理依賴的極端情況縮短；
- Soft Deprecation 不預定移除，仍保存文件與測試。

本文沒有把所有底線開頭名稱簡化成絕對可任意破壞，仍保留 PEP 對特殊名稱與文件邊界的限定。

SECTION

七十五、Rust

已重新核對 Edition Guide、RFC 3085 與 Migration Guide：

- Stable feature 原則上持續支援；
- 不相容表面變更可進入新 Edition；
- Edition 是 opt-in；
- 不同 Edition crate 可互操作；
- 遷移可使用 cargo fix --edition；
- 自動修正不能處理所有 Macro、Generated code、Doctest 與語義問題；
- Edition 不等於所有新功能都只在新 Edition 可用，多數向後相容功能可供所有 Edition 使用。

本文沒有把 Edition 描述成傳統 major-version 生態分裂。

SECTION

七十六、C++

已重新核對 WG21 官方資料、SD-10、SD-8 與 Stroustrup 的 HOPL 回顧：

- WG21 是 C++ 的國際標準化工作組；

- 現代演化由 Papers、Subgroups、Polls、實作者與國家會員共同構成；
- Language Evolution 原則將與舊 C++ 的順暢相容視為預設要求；
- Standard Library compatibility 文件同時區分使用者可依賴的介面與不可依賴的實作細節；
- Stroustrup 的歷史回顧明確將 C++ 描述為重視向後相容的活語言。

  本文沒有聲稱 C++ 永不破壞，也沒有將現代相容政策只歸於 Stroustrup。

SECTION

七十七、ECMAScript

已重新核對 2026-07-17 更新的 ECMAScript 2027 Draft 與 Annex B：

- tc39.es/ecma262 包含最近年度快照與已達 Stage 4 的完成提案；
- 當前 TC39 Process 使用 Stage 0、1、2、2.7、3、4；Stage 4 需要相容實作、Test262 驗收與實際發行經驗等條件；
- Annex B 對 Web Browser host 是 Normative，非瀏覽器 Host 可選；
- Annex B 明確說明其中 Legacy 功能具有不理想特性，若無既有使用本可移除；
- 新 ECMAScript 程式不應依賴這些功能；
- 瀏覽器因大量既有網頁而必須支援。

  本文把它分類為相容性化石與 Legacy 隔離，而非核心設計推薦。

SECTION

七十八、Swift

已重新核對 Swift 官方 Source Compatibility Test Suite、Upcoming Feature Flags 與 Library Evolution 文件：

- Source Compatibility Test Suite 以真實開源專案測試 Compiler 回歸；
- Upcoming Feature Flags 自 Swift 5.8 起允許逐項、逐 Target 採用下一重大模式的 Source-breaking feature；

- 多數 Swift 變更仍維持 Source compatibility，因此不需要 Upcoming flag；
- Swift 5.0 在 Apple 平台引入 Stable ABI；
- Swift 5.1 增加 Module stability 與 Library evolution，主要服務可與 Client 分開編譯、發行及更新的 Binary framework；
- Library evolution 只應在 Framework 需與 Client 分開發行時啟用；
- Library evolution 會改變性能特性與 Enum exhaustiveness 等設計邊界；
- Resilient change 要求新 Framework 對舊 Client 保持 Source 與 Binary compatibility。

  本文沒有把所有 Swift Package 都寫成具有 Binary evolution 保證。

  設計一致性、向後相容與社群演化並不是三個可以分開處理的議題。

  一致性決定語言是否仍然可理解；相容性決定使用者是否能相信升級；治理決定誰有權要求所有人支付改變成本。

  本文提出：

> cal-E
> =
> (
> Q,
> C,
> B,
> M,
> G,
> L,
> J
>)

並將語言歷史分成：

Active design
Compatibility fossil
Soft deprecation
Hard deprecation
Accidental stabilization

Experimental debt

成熟演化不等於永遠不破壞，也不等於每隔幾年重設語言。它需要：

> 【明確問題
> +
> 足夠收益
> +
> 最小破壞
> +
> 分層相容
> +
> 工具遷移
> +
> 跨版本互操作
> +
> 公開治理
> +
> 社群支持】

因此：

> 【LegacyPreserved
> not⇒
> LegacyEndorsed】

同時：

> 【CleanerDesign
> not⇒
> LegitimateBreakage】

PLDST 對設計師與治理共同體最關鍵的問題，不再只是：

> 他保守還是激進？

而是：

> 他認為哪些原則值得讓使用者支付遷移成本？他保護哪一種相容性？他如何區分活躍設計與歷史化石？當破壞不可避免時，是否提供版本邊界、工具、回退、證據與足夠正當性？當歷史不可刪除時，是否能將 Legacy 隔離，使新程式不必永遠重複舊錯誤？

最終原則為：

> 【保護使用者信任
> ∧
> 保留修正歷史的能力
> ∧
> 不把相容變成停滯，也不把進步變成暴力】

語言：
版本／時期：
變更：
一致性目標：
現有行為：
Legacy 類型：
來源相容：
ABI 相容：
行為相容：
資料相容：
工具相容：
生態相容：
組織相容：
受影響範圍：
安全收益：
未來演化收益：
遷移方式：
自動修正：
人工修正：
版本模式：
回退：
棄用期：
例外批准者：
真實生態測試：
信任風險：
證據：

信心：
問題是否具體：
現有傷害：
是否安全問題：
替代方案：
不破壞方案：
受影響使用率：
Benefit／breakage ratio：
是否可工具化：
資料／ABI 影響：
跨版本互操作：
警告期：
支持期限：
回退機制：
例外與救濟：
社群參與：
決策紀錄：

[R1] Go Project, “Go 1 and the Future of Go Programs,” “Go 1 Compatibility,” and Russ Cox, “Backward Compatibility, Go 1.21, and Go 2,” 2023.

— Go 1 相容承諾、GODEBUG、語言版本、Metrics 與 Go 2 的相容演化。

[R2] Python Enhancement Proposals, “PEP 387 – Backwards Compatibility Policy” and “PEP 5 – Guidelines for Language Evolution.”

— Public API、收益／破壞比、棄用期、Soft Deprecation 與 Steering Council 例外。

[R3] Rust Project, The Rust Edition Guide, RFC 2052, RFC 3085, RFC 1105, and Release Channels RFC 507.

— Stability without stagnation、Edition、跨 Edition 互操作、遷移工具與 Stable API 生命週期。

[R4] ISO/IEC JTC1/SC22/WG21, “SD-10: Language Evolution Principles,” “SD-8: Standard Library Compatibility,” WG21 official committee pages; Bjarne Stroustrup, “Evolving a Language in and for the Real World: C++ 1991–2006.”

— C++ 相容預設、Library 介面界線、國際標準治理與歷史演化。

[R5] Ecma TC39, ECMAScript Language Specification, Annex B, and “The TC39 Process.”

— Web Legacy Compatibility、Stage 4、實作與年度規格。

[R6] Swift Project, “Swift Source Compatibility Test Suite Now Available,” “Using Upcoming Feature Flags,” and “Library Evolution in Swift.”

— Source compatibility、Upcoming feature、ABI、Module stability、Library evolution 與生態回歸測試。

[Q-C] Conceptual coherence
[Q-S] Surface coherence
[Q-B] Behavioral coherence
[Q-G] Governance coherence
[C-S] Source compatibility
[C-A] ABI compatibility
[C-B] Behavioral compatibility
[C-D] Data compatibility
[C-T] Tool compatibility
[C-E] Ecosystem compatibility
[C-O] Organizational compatibility
[L-A] Active design
[L-F] Compatibility fossil
[L-S] Soft deprecation
[L-H] Hard deprecation
[L-X] Accidental stabilization
[L-E] Experimental debt
[M-W] Warning
[M-F] Automatic fix
[M-V] Versioned behavior
[M-E] Edition／mode
[M-R] Rollback
[M-I] Cross-version interoperation