Rich Hickey：價值、身分與簡單性的分離

Rich Hickey: Separating Values, Identity, and Simplicity

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-021.html

英文名稱： Rich Hickey: Separating Values, Identity, and Simplicity

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-021

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

Rich Hickey 常被描述為 Clojure 的創造者、不可變資料與函數式程式設計的倡議者，以及〈Simple Made Easy〉的講者。這些標籤都正確，卻容易把他的設計思想壓縮成幾句口號：

- 不可變就是好；
- Lisp 很簡單；
- 狀態很危險；
- 簡單勝過容易。

若只停在這一層，便無法解釋 Clojure 為何同時具有：

- Dynamic language；
- Lisp macro；
- Java interoperability；
- Persistent data structure；
- Atom、Ref、Agent、Var 等多種可變參照；
- Software transactional memory；
- Protocol、Multimethod；
- Transducer；
- Spec；
- REPL 驅動開發。

Hickey 並沒有主張真實世界沒有變化，也沒有要求所有程式完全無狀態。他的核心工作是把傳統命令式語言中被混成單一「變數／物件」概念的多個維度拆開：

- Value：某個不可改變的資訊；
- Identity：跨時間被認為是同一事物的名稱或身分；
- State：某個 Identity 在特定時間所關聯的 Value；
- Reference：程式如何在協調規則下取得或更新 State；
- Time：新舊值的順序，而不是把舊值抹除成不存在。

Clojure 官方對 Identity and State 的說明明確主張：值是不可變的；變化應理解為 Identity 在不同時間關聯不同值；Clojure 以不同參照型別處理同步、協調和可觀察性，而不是讓任意物件在任何位置被無規則改寫。[R1][R2]

Hickey 對「簡單」的定義同樣不是「初學容易」或「程式碼短」。〈Simple Made Easy〉把 Simple 解釋為沒有被纏結、交織成多個不可獨立理解的部分；Easy 則表示接近既有能力、熟悉工具或立即可取得。熟悉的技術可以很容易，卻在結構上高度複雜；不熟悉的值、不可變資料與函數組合起初可能不容易，卻能形成更簡單的系統。[R3][R4]

本文將 Hickey 的設計生涯分為六個相位：

- 既有 Lisp／Java 整合問題形成期：Jfli、Common Lisp 與 JVM 接近性的限制；
- Clojure 核心建造期：兩年自資研究、JVM Lisp、Persistent values 與 Java interop；
- Identity／State 模型期：Ref、Atom、Agent、Var 與 STM；
- 簡單性理論明文化期：Simple／Easy、Complect、Incidental complexity；
- 值導向系統與資料庫期：Datomic、Database as value、Information／History；
- 解除情境綁定期：Reducer、Transducer、Spec 及 Clojure 長期克制演化。

本文核心判斷為：

> 【Hickey 的主要設計方法不是單純移除功能，
>
> 而是把原本纏結的概念重新分離，】

使每個機制只承擔一項較清楚的責任：

> 【Value
> ≠
> Identity
> ≠
> State
> ≠
> Reference
> ≠
> Time】

其深層風格可以表示為：

> 【不可變值
> +
> 明確參照語義
> +
> 函數組合

這種設計並非沒有代價。Persistent data structure、Macro、Dynamic typing、JVM interop、Lazy sequence、STM 及多種 Reference type 會把複雜度移到 Runtime、Compiler、Library、學習和性能模型。Hickey 的「簡單」也可能被社群誤用成對其他語言或需求的道德化評斷；而 Clojure 的集中式設計和保守演化雖維持一致性，也可能使貢獻者感到決策速度慢、入口不透明或個人權重過高。

因此，更精確的 PLDST 判定是：

> Hickey 把程式設計中的時間、值、身分、變化和資料處理重新建模，以結構分離換取可推理性，再把必要的變化集中到具有明確協調語義的參照與系統邊界。

關鍵詞： Rich Hickey、Clojure、簡單性、值、身分、狀態、不可變資料、Persistent data structure、STM、Transducer、PLDST

SECTION

一、本文研究範圍

本文主要分析：

- Clojure 的形成；
- Rich Hickey 的官方 Rationale；
- Value、Identity 與 State 模型；
- Persistent collection；
- Ref、Atom、Agent、Var；
- Simple Made Easy；
- Value of Values；
- Datomic 的值導向思想；
- Reducer／Transducer；
- Spec；
- Clojure 的治理與長期演化。

本文不把以下成果全部歸於 Hickey：

- Lisp；
- Persistent data structure 的所有理論；
- Software transactional memory；
- JVM；
- HAMT；
- Datomic 的全部實作；
- ClojureScript；
- ClojureCLR；
- 所有 Contrib Library；
- 當代 Clojure Core 的全部工作。

SECTION

二、Hickey 的創始權重

Hickey 對下列事項具有高度直接權重：

- Clojure 原始問題設定；
- 第一 Compiler；
- JVM Lisp 核心；
- Persistent data 與 Reference model 的組合；
- 語言 Rationale；
- 多項核心 Library／語義；
- 長期功能裁決；

- Clojure 的整體設計品味。

  官方歷史與治理資料將 Clojure 明確描述為 Rich Hickey 創造的語言。[R5][R6]

SECTION

三、Clojure 很快成為團隊及社群成果

重要共同主體包括：

- Alex Miller；
- Stuart Halloway；
- David Nolen；
- Michael Fogus；
- Timothy Baldridge；
- Cognitect／Nubank 團隊；
- ClojureScript、ClojureCLR 團隊；
- Library、Tool、Editor、Documentation 作者；
- 企業使用者和社群。

  現行官方開發頁明確寫道：Clojure 由 Hickey 創造，現由 Nubank 支持的 Core team 開發，並重視有節制、深思熟慮及向後相容的演化。[R6]

  因此：

原始語言與哲學：Hickey 極高
核心實作和長期裁決：Hickey 高＋Core team
ClojureScript／CLR：獨立團隊
工具、生態、教育：多社群
當代演化：集中式 Core team

SECTION

四、為何仍需另一個 Lisp

Hickey 已具有 Common Lisp、C++、Java 與大型系統經驗。

問題並不是 Lisp 缺乏表達力，而是：

- Common Lisp 與主流 JVM 生態連接不自然；
- Java 有巨大 Library 和部署基礎；
- Java 的可變物件與並發模型使大型系統推理困難；
- 既有 Lisp 實作與宿主平台之間存在阻力；
- 使用者不願放棄現有產業平台。

SECTION

五、Jfli 作為前置實驗

Clojure 官方治理歷史記錄，Hickey 先建立 Jfli，在 LispWorks 的 Common Lisp 中嵌入 JVM；該方案不足以滿足目標後，他以約兩年 Sabbatical 建立全新的 JVM Lisp。[R5]

這個過程顯示：

> InteropLayer
> not⇒
> IntegratedLanguageModel

單純提供 FFI 不足以形成自然平台語言。

SECTION

六、第二個深層風格：新語言不必新建整個世界

Clojure 選擇：

- JVM Bytecode；
- Java Library；
- Java Deployment；

- Java Class；
- Java Thread；
- Existing operations。

  同時重建：

- 資料；
- 狀態；
- 函數；
- Sequence；
- 語法；
- 宏；
- 並發責任。

  因此：

> Clojure
> =
> NewSemanticModel
> +
> ExistingIndustrialRuntime

SECTION

七、為何是 Lisp

Lisp 提供：

- Code as data；
- Macro；

- REPL；
- 小型語法；
- Expression orientation；
- 函數式傳統；
- 可擴展語言。

  但 Hickey 沒有只複製 Common Lisp。

SECTION

八、宿主互操作是一級功能

Clojure 可以：

- 建立 Java Object；
- 呼叫 Method；
- 實作 Interface；
- 使用 Class；
- 捕捉 Exception；
- 發出 JVM Bytecode。

  它不是隔離的學術語言，而是可進入現有企業系統的宿主語言。

SECTION

九、動態但編譯

Clojure 是動態語言，但一般編譯至 JVM Bytecode。

官方首頁強調：它是 Compiled language，同時所有語言能力在 Runtime 中仍可使用。[R7]

因此：

Dynamic
≠
Only interpreted

SECTION

十、互動開發

Clojure 的官方入門將它描述為動態開發環境：程式在運行時持續成長，開發者可載入資料、增加功能、修正問題和測試，而不必每次重啟整個世界。[R8]

這延續 Lisp 活系統傳統，但不採 Smalltalk Image 作為唯一持久邊界。

SECTION

十一、值

Value 是：

- 不可變；
- 可比較；
- 可傳遞；
- 不受位置改寫；
- 表示某項資訊。

  若兩個值相等，它們的資訊相同，而不是依賴「同一記憶體位置」。

SECTION

十二、身分

Identity 是跨時間指稱同一概念實體的方式，例如：

- 某帳戶；
- 某玩家；
- 某程序；

- 某訂單；
- 某資料庫。

Identity 本身不是某一時刻所有欄位的集合。

SECTION

十三、狀態

State 是：

> State(identity,t)=Value⊠

變化不是把舊值「變成」另一個值，而是：

> Identity
> xrightarrowtime
> Value⊠,Value⊠,Value⊠,…

SECTION

十四、參照

Reference 是程式取得 Identity 當前狀態及提交新狀態的機制。

不同 Reference type 應對不同協調需求，而不是用單一可變物件覆蓋所有情況。

SECTION

十五、時間

若新值取代舊值但舊資訊完全消失，系統難以：

- 比較；
- Audit；

- Replay；
- Debug；
- Snapshot；
- Reason about causality。

  值導向設計將時間變成可建模維度。

SECTION

十六、不可變不等於每次完整複製

Persistent collection 透過結構共享，使新版本重用舊版本的大部分節點。

理想成本為：

> UpdateCost
> ≪
> CopyWholeCollection

SECTION

十七、結構共享的責任配置

使用者獲得：

- 不可變語義；
- 舊值仍可使用；
- Snapshot；
- 無防禦性複製；
- Thread sharing。

  Runtime／Library 承擔：

- Tree structure；
- Path copy；
- Hash trie；
- Memory sharing；
- Allocation／GC。

SECTION

十八、值語義改變 API

若資料不可變：

- Function 可直接接收 Map；
- 不必以 Getter 保護欄位；
- 不必防止呼叫者偷偷修改；
- Data 可以跨模組直接流動。

官方 FAQ 因此說，Clojure 對 Immutable data 通常不強調傳統 Encapsulation；直接存取資料具有實用價值。[R9]

SECTION

十九、代價

- Allocation；
- GC；
- Cache behavior；
- Transient／Mutable optimization 的額外知識；
- 與 Java mutable collection 的邊界；

- 某些算法不自然。

  不可變是預設，不等於所有內部實作都沒有 Mutation。

SECTION

二十、Atom

Atom 用於：

- 同步；
- 獨立；
- 單一 Identity 的狀態。

  swap! 透過 Compare-and-set 反覆應用純函數。

  官方 Reference 說明 Atom 的更新不產生 Race condition。[R10]

SECTION

二十一、Ref 與 STM

Ref 用於：

- 多個 Identity；
- 同步；
- 協調式更新；
- Transaction。

  STM 將讀取和寫入放入一致 Transaction，必要時重試。

SECTION

二十二、Agent

Agent 用於：

- 非同步；
- 獨立狀態；
- 由 Action Queue 序列更新。

  它把「提交變更」與「等待完成」分離。

SECTION

二十三、Var

Var 支援：

- Namespace Root binding；
- Dynamic thread-local binding；
- 程式定義和環境動態性。

  它不是一般共享 Mutable field 的替代品。

SECTION

二十四、參照型別不是便利 API 分類

四種機制編碼的是不同時間及協調語義：

參照 同步 協調 典型用途

Atom 是 獨立 Cache、Counter、單一狀態

Ref 是 多參照協調 Transaction

Agent 否 獨立序列 非同步更新

Var 依綁定 名稱／執行環境 定義與動態 Context

這正是「解除纏結」的語言化。

SECTION

二十五、Easy

Easy 可指：

- 熟悉；
- 接近；
- 已安裝；
- API 一行；
- 現成 Library；
- 不需學習。

  它描述使用者與事物的相對距離。

SECTION

二十六、Simple

Simple 指：

- 一個角色；
- 一個概念；
- 不與其他責任纏結；
- 可以獨立理解；
- 組合後仍保留邊界。

  它描述事物的客觀結構程度，而非使用者熟悉度。[R3][R4]

SECTION

二十七、Complect

Complect 表示把本可分離的維度纏在一起。

典型例子：

- State 與 Identity；
- Value 與 Place；
- What 與 How；
- Policy 與 Mechanism；
- Operation 與 Traversal；
- Data 與 Behavior；
- Logic 與 Timing。

SECTION

二十八、熟悉複雜性

OOP Mutable object 對許多開發者很 Easy：

```
obj.setX(...)
obj.getX()
```

但它可能同時纏結：

- Identity；
- State；
- Time；
- Method；
- Lock；
- Alias；
- Ownership。

SECTION

二十九、不熟悉簡單性

不可變 Value 和純函數最初可能較難學，但：

- Input 不變；
- Output 是新值；
- Function 沒有隱藏改寫；
- 可局部推理；
- 可重試；
- 可並行。

SECTION

三十、簡單性不是數量極小

一個系統可以有多個簡單元件。

一個只有少數 API 的 Framework 也可能把多個責任纏在同一機制。

所以：

> SmallFeatureCount
> not⇒
> Simplicity

SECTION

三十一、Database as value

Hickey 的 Datomic 思想把 Database 視為某一時間點的值，而不是只能透過可變 Server position 觀察的地方。

這允許：

- Snapshot query；
- Historical query；
- Consistent read；
- Function 以 Database value 為 Input；
- 新舊狀態比較。

SECTION

三十二、Information 與 Place 分離

傳統 Place-oriented model：

某地址現在放什麼

Value-oriented model：

某項資訊是什麼
在何時為真
由哪個 Identity 指稱

SECTION

三十三、歷史不是備份副產品

若新 Fact 只增加而非破壞舊 Fact：

- Audit；
- Time travel；
- Debug；
- Causal reasoning；
- Reproducibility；

成為系統原生能力。

SECTION

三十四、Datomic 不是 Clojure 語言功能

它是獨立產品及系統，有自己的團隊、商業模型和實作。

本文只把「Database as value」視為 Hickey 對值／時間模型的跨系統延伸。

SECTION

三十五、Sequence operation 的纏結

傳統 map、filter 可能同時綁定：

- Transformation；
- Input collection；
- Output collection；
- Laziness；
- Traversal；
- Execution context。

SECTION

三十六、Transducer

Transducer 將 Transformation 從來源和目的地分離，成為 Reducing function transformation。

因此同一轉換可用於：

- Sequence；
- Collection；
- Channel；

- Stream；
- 自訂 Reduce context。

  官方發布把 Transducer 定義為可組合、可在多種情境重用的 Algorithmic transformation。[R11]

SECTION

三十七、設計模式

> Operation
> -
> Context
> =
> ReusableEssence

Hickey 的方法不是加入更多 Adapter，而是找出被情境纏住的核心轉換並解耦。

SECTION

三十八、代價

Transducer 概念對初學者並不 Easy：

- Arity；
- Reducing function；
- Completion；
- State；
- Eduction；
- Transduce。

  它是典型「先付學習成本，換取後續結構簡單」。

SECTION

三十九、為何不是傳統封閉型別

Clojure 的 Map 常包含：

- 可選欄位；
- 部分資料；
- 多來源資料；
- 外部資料；
- 逐步建立資料；
- 同 Key 跨資料集共享語義。

  官方 Spec Rationale 明確指出，動態組合、合併和建構 Map 是 Clojure 的能力來源；同一 Key 應在不同集合中保持相同語義。[R12]

SECTION

四十、Spec 的作用

Spec 可提供：

- Predicate；
- Validation；
- Documentation；
- Generator；
- Instrument；
- Destructuring；
- Open data contract。

它不要求所有值被封裝進名義型別階層。

SECTION

四十一、Spec 不是完整靜態型別替代

它通常在 Runtime 或測試／工具階段提供證據。

不能保證：

- 所有路徑；
- 所有呼叫；
- 編譯前完整證明；
- 外部資料自動可信。

SECTION

四十二、集中而克制

Clojure 的核心演化特色包括：

- Rich Hickey 高權重；
- 小型 Core team；
- Design page；
- Patch／Ticket 歷史；
- 對核心新增保守；
- 高度向後相容；
- 鼓勵 Library 實驗。

SECTION

四十三、優勢

- 語言氣質穩定；
- 核心不快速膨脹；
- 舊程式可長期運行；
- 大功能需經多年思考；
- 避免短期流行。

SECTION

四十四、風險

- 提案入口可能不透明；
- 決策慢；
- 社群難以知道接受條件；
- 創始者品味權重過高；
- Core 與 Library 邊界可能讓使用者等待；
- 少數維護者負荷集中。

SECTION

四十五、與 BDFL 的差異

Clojure 有集中創始者影響，但並不主要使用 Python 式 PEP／公開最終裁決文化，也不是 Ruby 式 Matz 人格社群。

其制度更接近：

> MeasuredCoreStewardship
> +
> LibraryExperimentation
> +
> CompatibilityBias

SECTION

四十六、Jfli／問題形成期

問題：Common Lisp 與 JVM 整合不自然
策略：先做 Bridge，再判定需要新語言

SECTION

四十七、Clojure 核心期

問題：Java 平台缺乏值導向 Lisp
策略：JVM Lisp＋Persistent data＋Interop

SECTION

四十八、Identity／State 期

問題：共享可變物件纏結時間與協調
策略：Ref／Atom／Agent／Var 分工

SECTION

四十九、簡單性明文化期

問題：業界把容易、熟悉和簡單混為一談
策略：Simple／Easy／Complect 分析

SECTION

五十、值導向系統期

問題：Database／Object 抹除歷史
策略：Database as value、時間與 Fact

SECTION

五十一、解除情境期

問題：Transformation 與來源／執行方式綁定
策略：Reducer、Transducer、Spec

SECTION

五十二、問題 framing

Hickey 的核心問題是：

> 哪些原本可以獨立理解的概念，被主流語言或 Framework 纏在同一機制中，迫使所有使用者共同承擔交互複雜度？

SECTION

五十三、價值優先序

```
V_(Hickey)
≈
(
Simplicity,
Values,
Reasonability,
Immutability,
ExplicitStateSemantics,
Composition,
PracticalPlatformUse,
Stability
)
```

SECTION

五十四、核心—擴張偏好

偏好：

- Lisp 小語法；
- General data；

- Persistent collection；
- Function；
- Macro；
- Protocol；
- Library 擴張；
- 少量但深層的核心機制。

SECTION

五十五、顯式—推導偏好

偏好：

- State mechanism 明示；
- Mutation boundary 明示；
- Data 直接可見；
- Function transformation 組合。

同時保留：

- Dynamic typing；
- Macro；
- Runtime Protocol；
- Java Reflection fallback。

SECTION

五十六、效率—可讀性偏好

願意讓 Runtime／Persistent structure 承擔成本，以換取：

- 可推理；
- Snapshot；
- 並發共享；
- General data flow。

  但仍使用 JVM、Transient、Primitive hint 和 Compiler 最佳化處理熱點。

SECTION

五十七、安全—自由偏好

Clojure 提高：

- Memory safety；
- Immutable default；
- Controlled references；
- Transaction。

  仍保留：

- Java mutable object；
- Interop；
- Reflection；
- Macro；
- Dynamic execution。

  安全包絡不覆蓋所有宿主操作。

SECTION

五十八、相容性偏好

Clojure 長期高度克制，官方開發說明明確強調向後相容。

這維持信任，也可能延緩核心修正。

SECTION

五十九、治理偏好

- 創始者和小團隊深思；
- 功能先在 Library／實驗中成熟；
- 不以多數票加入語言；
- 重視長期方向和概念一致。

SECTION

六十、簡單性並非完全客觀可量測

「Complect」判斷常依賴：

- 抽象邊界；
- 使用情境；
- 觀察層級；
- 團隊知識；
- 性能需求。

不同設計者可能對同一機制是否纏結有不同判斷。

SECTION

六十一、不可變資料不消除狀態問題

真實系統仍需：

- I/O；
- Clock；
- Network；
- Database；
- Resource；
- External process；
- Human input。

  Clojure 是重新配置，不是移除所有 Effect。

SECTION

六十二、多參照型別增加學習成本

使用者需先判斷：

- 同步或非同步；
- 獨立或協調；
- Thread-local 或共享；
- Transaction 是否必要。

  分離責任提高結構清楚，也增加入口決策。

SECTION

六十三、Persistent data 有實際成本

特定數值、圖形、HPC、低延遲場景可能需要：

- Mutable array；
- In-place update；
- Native buffer；
- Specialized layout。

  Clojure 可透過 Java interop 使用它們，但會跨越主要價值模型。

SECTION

六十四、Macro 可重新纏結

Lisp Macro 可以建立：

- 隱藏控制；
- 新語法；
- 非局部效果；
- 工具困難。

  「Code as data」不自動產生簡單系統。

SECTION

六十五、集中治理不可只以一致性正當化

高一致性可能來自少數人承擔巨大審查成本；若缺少清楚參與和接班制度，也會形成可持續性問題。

時期　問題　決策　複雜度去向　風格

2000s 初　Lisp／JVM 邊界不自然　Jfli 實驗　FFI　問題驗證

2005–07　需要 JVM 原生 Lisp　Clojure　Compiler／Runtime　平台槓桿

2007+　共享狀態纏結　Ref、Atom、Agent、Var　參照語義　狀態分離

2011　Easy 被誤當 Simple　Simple Made Easy　設計理論　結構批判

2012　Mutable place 抹除資訊　Value of Values／Datomic　歷史資料　時間建模

2012–14　Collection operation 綁定執行情境　Reducer／Transducer　Higher-order abstraction　解纏結

2016+　動態開放資料缺契約工具　Spec　Runtime／Test tool　開放資料規格

長期　核心膨脹和相容風險　克制演化　Core team／Library　穩定治理

SECTION

六十六、主要原型

Rich Hickey 同時屬於：

- 概念解纏結設計者；
- 值導向系統建築師；
- 身分—狀態分離理論家；
- 宿主平台實用主義者；
- 克制型核心治理者。

SECTION

六十七、不適合的簡單標籤

不應只稱：

不可變資料倡議者
函數式純粹主義者
Lisp 極簡派
反物件導向者
STM 發明者

較精確的描述是：

> 反覆辨識被語言纏結的概念，將資料、時間、身分、變化及處理情境重新分離，再以宿主平台和少量專門機制使該模型可實際部署的設計者。

SECTION

六十八、最重要的連續性

Clojure、Datomic、Transducer 和 Spec 的共同方向是：

> 【從被綁定的情境中提取可重用的值與轉換】

SECTION

六十九、最重要的責任轉移

由：

每個可變物件自行隱藏並改寫狀態
轉為：

值保持不可變，參照機制明確承擔時間與協調

SECTION

七十、最重要的治理矛盾

Hickey 要求系統分離責任，但 Clojure 的語言裁決本身相對集中。

這不必然矛盾，卻是後續制度分析不可忽略的反例。

Rich Hickey 的設計思想不能縮成「不用變數」或「不可變資料比較好」。

他的核心貢獻是建立一套可跨語言、資料庫和系統設計使用的分離方法：

- 值是資訊，不會被改寫；
- 身分是在時間中持續的指稱；
- 狀態是身分在某一時刻的值；

- 變化是新值的建立和身分關聯更新；
- 參照應明確表達同步、協調及可觀察語義；
- 簡單是沒有纏結，不是立即熟悉；
- 容易可以服務採用，但不能替代簡單；
- 抽象應解除 Operation 與 Context 的綁定；
- 宿主平台可以被利用，而不必接受其全部語義；
- 語言核心的每一項新增都應承擔長期結構成本。

本文對 Hickey 的 PLDST 判定為：

> 【Conceptual Decomplecting Designer
> →
> Value-Oriented Systems Architect
> →
> Conservative Semantic Steward】

其核心優勢是：

- 對 Value、Identity、State 和 Time 提供清楚模型；
- Persistent data 降低 Alias 和共享變化；
- Reference type 把並發協調分類；
- Lisp 與 JVM 結合具有實用性；
- Transducer 展示從情境提取一般轉換；
- 長期演化保持高度一致和相容。

其核心代價是：

- 初始學習不 Easy；
- 多種 State mechanism 需要精確選擇；

- Persistent structure 和 JVM 具有成本；
- Dynamic typing 和 Macro 仍可能產生非局部複雜；
- 集中式設計降低社群預測性；
- 「Simple」容易被文化化成不可量測的優越評語。

最終原則為：

> 【不要把變化藏在值中
>
> ∧
> 不要把時間藏在位置中
>
> ∧
> 不要把熟悉誤認成簡單】

Hickey 的歷史提出的不是「所有人都應使用 Clojure」，而是一項更普遍的設計責任：

> 當系統難以理解時，先不要增加抽象層；應先檢查哪些本可分離的概念，被語言、物件、框架或流程纏成了同一件事。

人物：Rich Hickey
主要語言／系統：Clojure、Datomic、Transducer、Spec
核心時期：2000s–至今
主要問題：值、身分、狀態、時間和處理情境被纏結
主要策略：不可變值、Persistent data、專門參照、函數組合
複雜度去向：Runtime、Compiler、Library、學習與 Core governance
責任去向：值保持穩定，參照明確承擔變化與協調
主要保護對象：大型並發系統的開發者與長期維護者
主要限制：學習、性能邊界、動態工具、集中治理
歸因信心：高

[R1] Rich Hickey, “Values and Change: Clojure’s approach to Identity and State,” Clojure official site.

— Value、Identity、State、Time 及參照模型。

[R2] Clojure Reference, Atoms／Refs／Agents／Vars.

— 不同參照的同步、協調及更新語義。

[R3] Rich Hickey, “Simple Made Easy,” Strange Loop, 2011.

— Simple／Easy、Complect、熟悉性與結構複雜度。

[R4] Rich Hickey, “Simplicity Matters” slides and related transcripts.

— 理解、可靠性、交織機制及設計價值。

[R5] Stuart Halloway, “Clojure Governance and How It Got That Way,” Clojure official news, 2012.

— Jfli、兩年 Sabbatical、Clojure 起源及治理背景。

[R6] Clojure official Development page.

— Rich Hickey 創始、Nubank 支持 Core team、克制演化與向後相容。

[R7] Clojure official homepage and Rationale.

— JVM、Dynamic compiled language、Persistent data、STM、Pragmatic language design。

[R8] Clojure official Getting Started guide.

— Dynamic development、REPL 及運行中持續建立程式。

[R9] Clojure official FAQ.

— Immutable data、直接資料存取與 Encapsulation 觀念。

[R10] Clojure Reference, “Atoms.”

— 同步獨立狀態、swap!、CAS 和 Race-free update。

[R11] Rich Hickey, “Transducers are Coming,” Clojure official news, 2014.

— 可組合及跨情境的 Algorithmic transformation。

[R12] Rich Hickey, “clojure.spec – Rationale and Overview.”

— 開放 Map、可選／部分資料、Key 語義和 Dynamic composition。

[R13] Rich Hickey, “The Value of Values,” 2012.

— Value、Place-oriented programming、Information 和時間。

[R14] Rich Hickey, “A History of Clojure,” HOPL IV, 2020.

— Clojure 的原始目標、設計、取捨、團隊和後期歷史。

[T-J] Jfli／problem-formation phase
[T-C] Clojure core phase
[T-I] Identity／state phase
[T-S] Simplicity-theory phase
[T-V] Value-oriented systems phase
[T-D] Decontextualization phase
[S-D] Decomplecting
[S-V] Values
[S-I] Identity／state separation
[S-P] Persistent data
[S-H] Host-platform pragmatism
[S-C] Conservative evolution

附錄 D　第二輪史實、語義與治理校對紀錄
D.1 Clojure 的形成時間與個人投入
第二輪重新核對〈A History of Clojure〉及官方治理歷史：
- Clojure 的初始設計開始於 2005 年；
- 首次公開發布於 2007 年；
- Hickey 在 Jfli 不足以滿足 JVM／Lisp 整合目標後，以約兩年自資 Sabbatical 建立新的 Compiler 和語言；
- 「兩年」是形成核心實作的歷史描述，不表示 Clojure 此後不再有多人重寫及長期演化。
本文因此區分：
2005–2007：Hickey 主導的原型與公開發布

2007 後：Core team、社群、企業和多平台長期建設

D.2 Clojure 的四種參照
第二輪直接核對官方 Reference：
- Atom：Shared、Synchronous、Independent state；
- Ref：Shared、Synchronous、Coordinated state，通常由 Transaction 更新；
- Agent：Shared、Asynchronous、Independent state；
- Var：Global name／Root binding，並可選擇 Thread-local Dynamic binding。
這些機制不是四種相同 Mutable cell 的 API 包裝，而是不同的時間、同步和協調模型。
本文已避免把 Var 簡化成一般 Shared mutable global；官方也將任意修改 Var root 視為通常不佳風格。

D.3 Atom 的 Race-free 邊界
官方 Atoms 文件說明：
- `swap!` 讀取當前值、應用函數並使用 Compare-and-set 提交；
- CAS 失敗時，函數可能重新執行；
- Atom State change 不會產生資料競爭；
- `swap!` 函數因此應避免不可安全重試的外部副作用。
本文使用「Race-free update」指提交機制，不表示所有放入 `swap!` 的函數都自動具有任意 Effect safety。

D.4 Agent 的精確含義
官方 Agents 文件明確區分：
- Ref：多位置、同步協調；

- Agent：單位置、非同步、獨立更新；
- Agent Action 是 Function，其回傳值成為新 State；
- Agent 是 Reactive，不是具有 Blocking receive loop 的 Autonomous Actor；
- 觀察 Agent State 不需發送 Message。

本文因此沒有將 Agent 直接等同 Erlang Actor。

D.5 Value、Identity 與 State

第二輪核對官方 Values and Change Essay：

- Clojure 不否認真實世界變化；
- Value 被視為不可改變資訊；
- Identity 在時間中可以關聯到不同 State value；
- Reference model 使變化協調方式明示；
- Persistent collection 並不保存所有歷史版本的全域索引，只使舊值在仍被引用時保持可用。

本文因此沒有把 Persistent data structure 誤寫成自動事件資料庫或完整 Audit log。

D.6 Simple／Easy 的來源邊界

〈Simple Made Easy〉及其官方／保存投影片與逐字稿支持：

- Simple 指沒有被交織或纏結；
- Easy 與接近、熟悉、立即可用有關；
- 熟悉事物可以複雜；
- 不熟悉事物可以簡單；
- 簡單性被 Hickey 視為可靠及可理解系統的重要條件。

「客觀結構程度」是本文對 Hickey 定義的分析轉述，不表示存在一個被普遍接受、可精確量化的 Simplicity meter。

D.7 Clojure 與 JVM

第二輪核對官方首頁、Rationale 及 Java Interop：

- Clojure 是 JVM 上的 Dynamic、Compiled、General-purpose Lisp；
- 語言功能在 Runtime 仍可使用；
- Java Object、Collection、Method、Field 及 Exception 可直接互操作；
- 只有 Java Field、Vars、Refs、Agents 等特定位置可透過相應機制改變，Local binding 並非一般可賦值變數；
- Java Mutable object 仍可越過 Clojure 的主要不可變資料模型。

本文因此把「宿主平台槓桿」和「宿主風險邊界」同時保留。

D.8 Transducer 的歸因與概念

第二輪核對 Rich Hickey 2014 年官方公告：

- Transducer 被定義為可組合、可在多種情境重用的 Algorithmic transformation；
- 它們與 Input／Output source 解耦，可用於 Core collection 及 `core.async` 等 Context；
- 「Operation - Context = Reusable essence」是本文的形式化概括，不是官方數學公式；
- Reducer、Transducer 及相關實作仍有 Core team 和社群工作，不能全部歸為 Hickey 單人程式。

D.9 Spec 的開放資料前提

官方 Spec Rationale 明確指出：

- Clojure 經常合併、組合及逐步建立 Map；
- 資料可能 Optional、Partial 或來自不可靠外部來源；
- 同一 Qualified key 應跨資料集合保持語義；

- Spec 嘗試在不封閉資料的情況下提供 Predicate、Documentation、Generation 及 Instrumentation。
本文沒有把 Spec 描述成靜態型別系統，亦沒有宣稱它在所有 Runtime boundary 自動驗證資料。

D.10 現行治理與版本邊界
截至 2026 年 7 月：
- 官方 Development 頁仍把 Clojure 描述為由 Rich Hickey 創造、由 Nubank 支持的 Core team 開發；
- 官方明確強調 Measured、Thoughtful 和 Backward-compatible evolution；
- 官方 REPL 文件目前示例版本為 Clojure 1.12.0。
版本號是時間敏感資訊，只記於校對附錄；人物風格主要依賴跨版本的長期克制原則，而不是單一當前版本。

D.11 集中治理的判斷邊界
官方資料支持：
- Hickey 長期具有高設計權重；
- Core team 相對集中；
- 語言演化保守。
本文對「提案入口可能不透明、決策慢」的描述是基於治理結構的風險分析，不等於聲稱每位貢獻者都遭遇相同經驗，也不否認官方 Ask Clojure、JIRA、Design pages、Mailing lists 和社群討論。

D.12 PLDST 原型邊界
下列名稱是本文分析原型，不是 Hickey 自稱的正式學派：

概念解纏結設計者

值導向系統建築師

身分—狀態分離理論家

克制型核心治理者

其中「Conceptual Decomplecting」直接承接 Hickey 的 Complect 詞彙；其他名稱是跨 Clojure、Datomic、Transducer 和治理決策形成的分析綜合。