機器效率與人類可讀性：成本模型應寫在語言表面，還是藏在編譯器之後？

Machine Efficiency and Human Readability: Should Cost Models Appear in Source Code or Hide Behind the Compiler?

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-008.html

英文名稱： Machine Efficiency and Human Readability: Should Cost Models Appear in Source Code or Hide Behind the Compiler?

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-008

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

程式語言設計經常把「接近機器」與「容易閱讀」描繪成一條單軸：

機器透明、效能可控 ←——————————→ 高階抽象、人類可讀

這種圖像過度簡化。C 與 C++ 讓資料布局、指標、配置與硬體操作較容易出現在程式表面，但編譯器仍依抽象機器與 as-if 原則進行轉換，表面敘述並不是實際指令序列的逐字記錄；C++ 的零額外成本原則約束特定抽象的運行時間與空間開銷，並不保證編譯時間、診斷或程式碼體積為零。Rust 以所有權、借用、迭代器與型態狀態等高階構造，試圖讓安全及可讀抽象編譯成接近手寫低階程式的結果，但其代價部分前移至型別規則、編譯器與建置時間。Go 以簡單、可讀的表面和垃圾回收降低大型團隊協作負擔，卻仍要求有性能需求的程式設計者理解配置、逃逸、資料布局與 GC 壓力。Java／HotSpot 透過直譯、分層編譯、Profile 與 Escape Analysis，允許簡潔的物件程式在 Runtime 中被重新最佳化，但暖機、去最佳化、配置移除與峰值性

這些案例顯示：成本不是只有「明示」或「隱藏」兩種狀態。它可以被放在：

- 語法；
- 型別；
- API；
- 命名慣例；
- 編譯器診斷；
- 最佳化契約；
- Profiler；
- Runtime 指標；
- 組織基準測試。

  本文提出 成本可見性與效率配置模型（Cost Visibility and Efficiency Allocation Model, CVEAM），將一項設計表示為：

> cal-P
> =
> (
> K,
> V,
> O,
> E,
> S,
> R
>)

其中：

- K：Cost vector，成本向量；
- V：Visibility，成本可見位置；
- O：Optimization dependence，對最佳化器的依賴；
- E：Evidence，效能證據；
- S：Stability，成本穩定性；
- R：Responsibility，性能責任配置。

本文將效率拆成九個維度：

- 執行時間；
- 記憶體使用；
- 配置與回收；
- 資料區域性；
- 間接層與動態派發；
- 安全檢查；
- 排程與同步；
- 編譯、暖機與專門化；
- 尾延遲與性能變異。

核心命題是：

> 【高階抽象
> not⇒
> 高運行成本】

同時：

> 【來源碼看似低階
> not⇒
> 成本完全可預測】

成熟設計不應讓所有機器細節污染每一行程式，也不應要求使用者相信不可觀察的「編譯器會最佳化」。更合理的原則是：

> 讓一般程式以意圖清楚的形式書寫；讓跨邊界、量級改變、不可攤銷、可能造成尾延遲或資源失控的成本保持可辨識；讓編譯器自由消除偶發成本，但以文件、診斷、反組譯、配置分析、Profiler 與基準測試提供可驗證證據。

關鍵詞： 程式語言設計、成本模型、零額外成本、可讀性、垃圾回收、JIT、惰性求值、型別穩定、效能可觀察性、PLDST

SECTION

一、執行時間 K

包括：

- 指令數；
- 分支；
- 向量化；
- 演算法複雜度；
- 呼叫與派發；
- Cache miss；
- I/O 等待。

平均速度不能取代：

- P95；
- P99；

- Worst case；
- 暖機後與冷啟動；
- 不同資料分布。

SECTION

二、記憶體 K_m

包括：

- 常駐集合；
- 峰值；
- Stack／Heap；
- Metadata；
- Code cache；
- JIT code；
- Fragmentation；
- Working set。

時間相近的兩個實作，可能具有完全不同的記憶體與 Cache 行為。

SECTION

三、配置與回收 K_a

配置成本包括：

- 物件建立；
- Reference counting；
- GC 掃描；

- Free；
- Arena；
- 生命週期；
- Finalization；
- 暫時值。

  一段可讀的鏈式 API 可能：

- 被融合為零配置；
- 產生多個暫時物件；
- 依編譯器版本不同而改變；
- 只有在熱路徑才被消除。

  因此「抽象語法」本身不能決定配置成本。

SECTION

四、資料區域性 K_l

資料布局影響：

- Cache；
- Prefetch；
- SIMD；
- False sharing；
- Traversal；
- Pointer chasing。

  物件導向表面可能掩蓋分散配置；資料導向 API 也可能以較高表面複雜度換取連續布局。

SECTION

五、間接層 K□

包括：

- Virtual call；
- Trait object；
- Interface；
- Closure；
- Function pointer；
- Reflection；
- Dynamic dispatch；
- Boxing。

  間接層有時被 devirtualize 或 inline，有時則保留。

  語言應區分：

  語義上必須動態
  可能被最佳化成靜態
  語義上已是靜態

SECTION

六、安全檢查 K_b

包括：

- Bounds check；
- Null check；
- Overflow check；
- Type check；

- Borrow check；
- Capability check；
- Array store check。

  檢查可能：

- 在編譯期消除；
- 在熱路徑合併；
- 始終保留；
- 在不同 Build profile 中改變。

  安全檢查具有成本，但事故、漏洞與人工證明同樣有成本。

SECTION

七、排程與同步 K_s

包括：

- Thread；
- Goroutine；
- Actor；
- Future；
- Task；
- Lock；
- Channel；
- Work stealing；
- Context switch。

go f()、async 或 Actor send 的表面短小，不代表：

- 建立零成本；
- 排程立即；
- 不配置；
- 不需背壓；
- 不可能形成佇列。

SECTION

八、編譯、暖機與專門化 K_w

包括：

- Ahead-of-time compile；
- Template／Monomorphization；
- JIT；
- Profile collection；
- Code cache；
- Deoptimization；
- First-call latency；
- Precompilation。

若只比較穩態 throughput，會忽略 CLI、Serverless、短任務與互動式環境。

SECTION

九、尾延遲與變異 K_v

Runtime 可能有：

- GC pause；
- JIT compilation；
- OS scheduling；
- Cache contention；
- Deoptimization；
- Lock convoy；
- Background compilation。

  平均值相近，尾延遲可能完全不同。

SECTION

十、成本向量

> K
> =
> (
> K□,
> K_m,
> K_a,
> K_l,
> K□,
> K_b,
> K_s,
> K_w,
> K_v
>)

PLDST 不用單一「效率分數」取代此向量。

SECTION

十一、意圖可讀性

讀者能否快速知道程式想做什麼？

高階集合操作、Iterator、Query、Pattern matching 可能比顯式 Index 與 Pointer arithmetic 更清楚。

SECTION

十二、成本可讀性

讀者能否知道：

- 是否配置；
- 是否複製；
- 是否阻塞；
- 是否遠端；
- 是否動態派發；
- 是否保留資料；
- 是否可能重試；
- 是否有 GC 壓力？

  意圖可讀性與成本可讀性可能互相衝突。

SECTION

十三、局部可讀性

單行程式是否容易理解。

鏈式 API、Operator overloading、Implicit conversion 可能提高局部流暢度，卻降低實際成本可見性。

SECTION

十四、系統可讀性

大型系統能否推理：

- 資源所有權；
- 服務邊界；
- 故障；
- Backpressure；
- Memory lifetime；
- Parallelism；
- Deployment。

局部簡潔可能增加全域不透明。

SECTION

十五、診斷可讀性

當性能不符預期時，使用者能否知道：

- 哪個抽象未被消除；
- 哪裡配置；
- 哪裡逃逸；
- 哪個 Call 未 Inline；
- 哪裡形成 Thunk；
- 哪個型別不穩定；
- 哪次 GC 造成停頓？

這決定「隱藏成本」是否可治理。

SECTION

十六、語法表面

可用語法明示：

- new／allocation；
- move；
- copy；
- unsafe；
- await；
- spawn；
- volatile；
- strict；
- lazy；
- mut；
- throws。

  優勢：

- Code review 可見；
- 不依賴工具；
- 邊界清楚。

  代價：

- 樣板；

- 使用者可能機械標註；
- 編譯器最佳化後實際成本不同；
- 語法只顯示類別，不顯示量級。

SECTION

十七、型別與效果

成本資訊可以進入：

- Ownership；
- Borrow；
- Linear type；
- Effect；
- Async type；
- Region；
- Sized／Unsized；
- Value／Reference；
- Capability；
- Strictness。

型別能保證某些資源關係，但不一定保證實際時間。

例如 Rust 的所有權能讓複製與移動具有較明確語義，但編譯器仍可能消除實際 Memory move。

SECTION

十八、API 與命名

命名可暗示成本：

clone
copy
collect
to_list
materialize
blocking
spawn
remote
unchecked
lazy
eager

API 命名是重要成本契約。

但名稱若沒有：

- 文件；
- 複雜度；
- 配置行為；
- Ownership；
- Failure；

  仍不充分。

SECTION

十九、編譯器診斷

Compiler 可提供：

- Escape analysis；
- Allocation report；
- Inlining report；
- Vectorization report；
- Bounds-check report；

- Monomorphization size；
- Specialization；
- Missed optimization；
- Code size。

這讓語言表面保持清楚，同時提供機器層證據。

SECTION

二十、Profiler 與 Runtime 指標

Profiler 可觀察：

- CPU；
- Allocation；
- Heap；
- GC；
- Lock；
- Scheduler；
- Trace；
- Cache；
- JIT；
- Tail latency。

其限制是：

- 只觀察已執行路徑；
- 結果依負載與環境；

- 可能有量測干擾；
- 無法單獨證明最壞情況。

SECTION

二十一、規格與性能契約

語言或 Library 可承諾：

- 複雜度上界；
- Amortized complexity；
- 不配置；
- 不阻塞；
- 無 GC；
- Constant time；
- Stable layout；
- Zero-overhead；
- Real-time bound。

承諾必須區分：

語言保證
標準程式庫保證
特定實作保證
最佳化機會
經驗性結果

SECTION

二十二、機器透明型

代表：

- C；

- 部分系統語言設計。

  偏好：

- 資料表示接近硬體；
- 指標、陣列、配置與呼叫可見；
- Runtime 小；
- 編譯器以 as-if 規則最佳化。

  優勢：

- 低層控制；
- ABI 與系統介面直接；
- 成本類別較容易定位；
- 適合嵌入、Kernel、Driver。

  風險：

- 表面操作不等於實際機器操作；
- Undefined behavior 可能放大最佳化影響；
- 手寫低階程式未必優於最佳化器；
- 安全責任高度落在使用者。

  C 標準的抽象機器方法本身就表示：規格要求可觀察結果如同某種抽象執行，而不要求實作真的逐步使用該機制。[R1]

SECTION

二十三、零額外成本抽象型

代表：

- C++；

- Rust。

  偏好：

- 以泛型、Iterator、RAII、Ownership、Trait 等抽象表達意圖；
- 不使用的功能不支付 Runtime 成本；
- 使用的抽象力求不比精心手寫低階程式更差。

  優勢：

- 高階意圖與低階效率可兼得；
- Library author 可建立領域抽象；
- 安全與資源管理能前移到型別及編譯期。

  風險：

- 「零額外成本」只約束特定比較基線；
- 編譯時間、Code size、Monomorphization 與診斷可能上升；
- 是否消除抽象常依賴最佳化；
- Debug build 與 Release build 差異可能很大。

  Stroustrup 將零額外成本概括為「不用的不付費，使用的難以手寫得更好」；Rust 官方則以 Iterator 與 typestate 等案例說明高階構造可編譯成接近直接實作的程式碼。[R2][R3]

SECTION

二十四、可讀組織工程型

代表：

- Go。

  偏好：

- 語言表面一致；

- 建置快速；
- 依賴明確；
- Goroutine、Interface 與 GC 提供簡潔模型；
- 效率接近系統需求，但不暴露所有低層控制。

優勢：

- 團隊可讀性；
- 工具統一；
- 輕量並行表面；
- 常見服務開發成本低。

風險：

- 配置、逃逸與 GC 壓力不完全顯示在語法；
- Goroutine 建立容易，生命週期與背壓仍需設計；
- Interface 可能帶來間接派發與配置；
- 性能敏感程式仍需 Profile 與資料布局知識。

Go 的官方設計回顧同時強調來源文字清楚表達意圖，以及懂得資料表示與配置的程式設計者仍可降低 Collector 壓力。[R4]

SECTION

二十五、自適應 Runtime 型

代表：

- Java／HotSpot；
- 部分 JVM 語言。

偏好：

- 語言與 Bytecode 保持抽象；
- Runtime 依實際 Profile 對熱點編譯；
- 使用 Escape Analysis、Inlining、Devirtualization 等消除部分表面成本。

優勢：

- 可依真實工作負載最佳化；
- 動態載入與大型平台互操作；
- 物件抽象不必固定等於 Heap 配置；
- 冷程式碼不必全部重度編譯。

風險：

- 暖機；
- Code cache；
- Deoptimization；
- GC；
- 不同 VM 與旗標；
- Benchmark 容易失真。

Oracle 的 HotSpot 文件指出，VM 只編譯性能關鍵區域；Escape Analysis 可分析物件使用範圍，並支援消除部分配置或鎖定成本。[R5]

SECTION

二十六、語義抽象—嚴格度逃生型

代表：

- Haskell／GHC。

偏好：

- 非嚴格求值；
- 純函數組合；
- 由 Runtime 與 Compiler 決定何時求值；
- 在性能敏感區使用 seq、BangPatterns、StrictData 與 Profiling。

優勢：

- 組合性；
- 無限資料結構；
- 控制求值需求；
- 共享；
- 高階重構。

風險：

- Thunk；
- Space leak；
- 求值時機不直觀；
- Full laziness 增加共享也可能提高 Memory residency；
- 高性能程式需理解嚴格度。

Haskell 2010 明確將函式應用定義為 non-strict；GHC 文件則指出，數值內圈中消除 thunk 可能帶來巨大收益，且 Full laziness 增加共享時也可能提高記憶體駐留。[R6][R7]

SECTION

二十七、推導專門化型

代表：

- Julia。

偏好：

- 動態表面；
- 多重派發；
- Compiler 依實際型別專門化；
- 使用者以 type-stable function 與 function barrier 協助最佳化。

優勢：

- 高階數值程式可接近低階性能；
- 泛型程式碼自然；
- Interactive workflow；
- 同一函式服務多型別。

風險：

- Type instability 形成性能懸崖；
- 首次執行編譯延遲；
- 全域變數與抽象 Container 影響最佳化；
- 小型臨時陣列造成配置；
- 表面相同程式可因型別不同產生巨大成本差。

Julia 官方建議通常不必宣告回傳型別，而應撰寫讓編譯器能推導回傳型別的 type-stable function；效能指引也把非預期配置視為 type instability 或暫時陣列問題的警訊。[R8]

SECTION

二十八、C／C++：表面接近硬體仍有抽象機器

C 語言常被說成「可攜式組合語言」，但標準規格描述的是抽象機器與可觀察行為，不是固定指令序列。

這意味：

- x + y 不等於保證某一條 CPU 指令；
- 來源順序不等於全部執行順序；
- 未定義行為會影響可允許最佳化；
- volatile 只具有特定語義，不是完整同步工具。

  成本透明是相對的，不是逐指令同一。

SECTION

二十九、C++：高階抽象依賴可消除性

RAII、Generic algorithm、Range 與容器可以使程式：

- 更清楚；
- 更安全；
- 更容易替換。

  但性能結果取決於：

- Inlining；
- Specialization；
- Layout；
- Allocation；
- Iterator category；
- Exception model；
- Build mode。

  零額外成本不是「每個抽象在每個編譯器都永遠消失」，而是語言與 Library 的設計方向及比較原則。

SECTION

三十、Rust：Ownership 同時是安全與成本語言

Rust 區分：

- Move；
- Borrow；
- clone；
- Copy；
- Owned／Borrowed iterator；
- Static／Dynamic dispatch。

這讓部分成本在 API 與型別中可見。

但：

- Move 通常不代表逐 byte 深拷貝；
- clone 的實際成本由型別實作決定；
- Iterator 是否完全融合依程式與最佳化；
- Bounds check 是否消除需要證據。

因此 Rust 提高的是成本類別可辨識性，不是精確 Cycle 可見性。

SECTION

三十一、Go：簡單 Goroutine 不等於無成本並行

go f() 很短，但系統仍需要：

- Stack；
- Scheduler metadata；

- Capture；
- Channel；
- Cancellation；
- Backpressure；
- Leak prevention。

Effective Go 說明 Goroutine 使 Thread 建立與管理的複雜性被隱藏，但其 HTTP 範例也提醒：每個請求建立 Goroutine 時，若並行限制設計不當，仍會建立大量等待工作的 Goroutine。[R9]

因此，並行表面可讀性必須配合生命週期與負載工具。

SECTION

三十二、Java：來源中的 new 不等於必然 Heap 配置

HotSpot Escape Analysis 可以判斷物件是否逃離方法或 Thread，並支援：

- Scalar replacement；
- 消除可被 Scalar replacement 的物件配置；
- Lock elimination。

本文不將這些實作最佳化簡化為「Java 物件一般會被配置在 Stack」；Oracle 文件的正式重點是分析 Escape scope，並消除可被純量替代的物件配置與相關鎖。

但使用者不應將某次 JIT 結果寫成語言保證。

正確層級是：

Java semantics：建立物件語義
HotSpot implementation：可能消除物理配置
Profiler／JIT log：特定執行證據

SECTION

三十三、Haskell：優雅組合可能隱藏求值圖

一個 Pipeline 可以具有高度意圖可讀性，但性能取決於：

- 是否融合；
- 是否共享；
- 是否形成 thunk；
- 是否保留不再需要的參考；
- 是否因 laziness 延後錯誤；
- 是否使用嚴格資料欄位。

  所以 Haskell 的性能閱讀單位不只是來源控制流，也包括需求圖與 Heap profile。

SECTION

三十四、Julia：同一表面因型別穩定性而分岔

Julia 中兩個看似相似的函式，若一個回傳型別穩定、另一個依資料分支回傳不同型別，Compiler 可能產生完全不同的程式碼。

這使性能成本部分位於：

- Type inference；
- Concrete type；
- Function boundary；
- Specialization；
- Allocation profile。

  Julia 的設計不是把機器成本完全藏起來，而是讓使用者透過 @code_warntype、Allocation measurement 與 Profile 觀察推導結果。

SECTION

三十五、L0：完全隱藏

使用者看不到：

- 成本類別；
- 量級；
- 工具；
- 文件；
- 失敗原因。

這是最低治理能力。

SECTION

三十六、L1：文件化

成本只存在於：

- API 文件；
- 效能指南；
- 實作說明。

適合非核心與可變最佳化，但容易被忽略。

SECTION

三十七、L2：命名與 API 可見

例如：

clone
collect
blocking
spawn
materialize
unchecked
能讓 Code review 發現高成本操作。

SECTION

三十八、L3：型別與語法可見

例如：

- Owned／Borrowed；
- Async；
- Effect；
- Strict；
- Dynamic dispatch；
- Unsafe。

適合跨邊界與安全敏感成本。

SECTION

三十九、L4：Compiler 可解釋

Compiler 可顯示：

- 配置；
- 逃逸；
- 未 Inline；
- Bounds check；
- Specialization；
- Vectorization。

適合依實作與最佳化條件改變的成本。

SECTION

四十、L5：Runtime 可觀察

Profiler、Trace 與 Metrics 顯示實際負載下成本。

適合：

- JIT；
- GC；
- Scheduler；
- Tail latency；
- Cache；
- Contention。

SECTION

四十一、L6：規格化保證

例如：

- 不配置；
- O(1)；
- Amortized；
- No GC；
- Stable layout；
- Real-time bound；
- Constant-time security。

此層最強，也最限制實作與演化。

SECTION

四十二、語言設計者

負責：

- 哪些成本可被表達；
- 哪些成本可被推導；
- 哪些操作有隱藏 Runtime；
- 是否提供低階逃生口；
- 是否允許量級意外改變。

SECTION

四十三、編譯器與 Runtime

負責：

- 遵守語義；
- 實作最佳化；
- 不虛假承諾；
- 提供診斷與觀察；
- 記錄去最佳化與配置；
- 保持結果可驗證。

SECTION

四十四、Library author

負責：

- 複雜度文件；
- 配置與 Ownership；
- Blocking；
- Iterator laziness；
- Error；
- Big-O；
- Cancellation；
- Allocation behavior。

SECTION

四十五、程式設計者

負責：

- 選擇合適演算法；
- Profile；
- 建立代表負載；
- 區分冷啟動與穩態；
- 不把單次 Benchmark 當普遍定律；
- 在需要時使用明確低階控制。

SECTION

四十六、組織

負責：

- 性能預算；

- Benchmark infrastructure；
- Regression；
- SLO；
- Capacity；
- Production trace；
- Hardware matrix。

SECTION

四十七、平衡條件

> Resp_(perf)(a)
> ≤
> Control(a)+Observability(a)+Evidence(a)

如果要求使用者保證低延遲，卻不提供 GC、Scheduler 與配置可觀察性，責任配置失衡。

SECTION

四十八、來源碼決定論

錯誤觀念：

> 看起來低階，所以一定快。

忽略：

- 演算法；
- Cache；
- Undefined behavior；
- Compiler；

- I/O；
- Parallelism；
- Data layout。

SECTION

四十九、最佳化器信仰

錯誤觀念：

> 編譯器一定會消除。

正確態度：

可能最佳化
→ 檢查 Compiler report／assembly／profile
→ 不把它寫成語言保證

SECTION

五十、抽象恐懼

錯誤觀念：

> 高階 Iterator、Generic 或 Object 一定慢。

事實上抽象可能：

- Inline；
- Fuse；
- Specialize；
- Scalar replace；
- Devirtualize。

需要證據，而不是語法印象。

SECTION

五十一、性能懸崖

小改動造成量級改變：

- Type instability；
- 失去 Inline；
- 介面逃逸；
- Box；
- Dynamic dispatch；
- Thunk；
- GC promotion；
- Monomorphization explosion。

  語言應提供可解釋工具。

SECTION

五十二、Debug／Release 分裂

若 Debug build 與 Release build 的性能模型差異過大：

- 使用者可能誤診；
- 測試不代表部署；
- 安全檢查行為可能不同；
- Benchmark 失真。

  文件與工具必須明示 Build profile。

SECTION

五十三、平均值遮蔽尾延遲

GC 與 JIT 系統尤其不能只看平均 throughput。

至少分開：

- Cold start；
- Warm throughput；
- P99；
- Peak memory；
- Pause；
- Compile time。

SECTION

五十四、微基準過度推論

微基準可能：

- 被 Dead-code eliminate；
- 不代表 Cache；
- 不代表真實資料；
- 不含 I/O；
- 不含 GC；
- 不含並行；
- 對 JIT 暖機敏感。

SECTION

五十五、意圖優先，但成本類別不可消失

一般程式應以清楚意圖書寫；配置、阻塞、遠端、複製、生成大量工作等高影響成本應可辨識。

SECTION

五十六、穩定成本寫入 API，易變成本交給工具

若成本是語義與長期契約的一部分，應在：

- 型別；
- 名稱；
- 規格；

  中明示。

  若成本依最佳化器與硬體而變，應由：

- Report；
- Profiler；
- Benchmark；

  呈現。

SECTION

五十七、量級改變必須顯眼

從：

- O(1) 到 O(n)；
- Lazy 到 materialized；

- Borrow 到 clone；
- Local 到 remote；
- Stack-like 到 Heap；
- Static 到 dynamic dispatch；

  應有明確邊界。

SECTION

五十八、最佳化是證據，不是語義

> OptimizationObserved
> ≠
> LanguageGuaranteed

除非規格明確承諾。

SECTION

五十九、抽象應可降解

高階抽象至少應提供：

- 展開；
- IR；
- Assembly；
- Allocation report；
- Profile；
- Cost documentation。

SECTION

六十、安全與效率不可只看 Runtime

靜態檢查增加編譯成本，卻可能降低：

- Runtime check；
- 事故；
- 防禦程式；
- 測試狀態；
- 安全修復。

  應做全生命週期比較。

SECTION

六十一、可讀性必須包含性能除錯

只有正常路徑好讀、性能異常時完全無法解釋，不是完整可讀設計。

SECTION

六十二、成本暴露指紋

Allocation visibility
Copy／move visibility
Dispatch visibility
Evaluation-order visibility
Blocking visibility
Concurrency visibility
GC visibility
Warm-up visibility
Tail-latency visibility

SECTION

六十三、最佳化依賴指紋

Inlining dependence
Specialization dependence
Escape-analysis dependence
Devirtualization dependence
Fusion dependence
Strictness-analysis dependence
JIT-profile dependence

SECTION

六十四、證據指紋

Complexity documentation
Compiler reports
Assembly／IR inspection
Allocation profiler
CPU profiler
Heap profiler
Runtime trace
Benchmark guidance
Production metrics

SECTION

六十五、設計師比較問題

- 他把哪些機器成本放在語法上？
- 哪些放在型別或 API？
- 哪些交給最佳化器？
- 哪些交給 Runtime？
- 他接受多大的暖機與性能變異？
- 他要求使用者理解資料布局到什麼程度？
- 安全檢查如何付費？
- 抽象未被消除時如何診斷？
- 是否承諾零額外成本，承諾範圍是什麼？

- 他優先保護平均開發效率、峰值性能還是最壞情況？

SECTION

六十六、輸入

designer
language
version
feature
source_example
compiler
runtime
build_profile
hardware
official_performance_docs

SECTION

六十七、分析管線

重新網路搜尋
→ 語義成本抽取
→ 成本向量
→ 可見性階梯
→ 最佳化依賴
→ 規格保證／實作行為分離
→ 工具與證據檢查
→ 冷啟動／穩態／尾延遲分離
→ 反例與性能懸崖
→ 第二輪事實校對
→ 風格報告

SECTION

六十八、JSON 雛形

```
{
 "mechanism": "iterator pipeline",
 "cost_vector": {
  "runtime_time": "implementation-dependent",
  "allocation": "may be eliminated",
  "dispatch": "usually static in generic form",
  "compile_time": "increased"
```

```
 },
 "visibility": {
  "source": "high-level intent",
  "type": "ownership and item type",
  "compiler_report": "recommended",
  "profile": "required for workload claim"
 },
 "guarantee_level": "design principle, not universal per-program proof",
 "performance_cliffs": [
  "dynamic dispatch",
  "lost inlining",
  "unexpected collection"
 ]
}
```

SECTION

六十九、SKILL 禁止事項

不得：

- 以來源碼高低階直接判定速度；
- 把零額外成本寫成所有成本為零；
- 把 JIT 最佳化寫成 Java 語言保證；
- 把 Rust Move 寫成深拷貝；
- 把 Go Goroutine 寫成零成本 Thread；
- 把 Haskell Laziness 寫成必然慢或必然省計算；
- 把 Julia 動態語法寫成無法靜態專門化；
- 忽略 Debug／Release；
- 忽略暖機與尾延遲；
- 用單次微基準評價整個語言；
- 混淆規格、實作與特定版本。

SECTION

七十、效能依賴工作負載

同一抽象在：

- 短任務；
- 長服務；
- 嵌入式；
- HPC；
- Web；
- Serverless；

  可能有完全不同結果。

SECTION

七十一、硬體改變設計優勢

Cache、SIMD、NUMA、GPU、記憶體與核心數量會改變最佳配置。

設計師風格分析需要標示年代。

SECTION

七十二、最佳化器會演化

今日未被消除的成本，未來可能消除；反之，曾經有效的技巧可能在新版失效。

每篇個案必須重新核對版本。

SECTION

七十三、可讀性具有社群性

熟悉 Iterator、Ownership、Monad 或 Multiple dispatch 的使用者，會有不同可讀判斷。

SECTION

七十四、保證與經驗不能混同

PLDST 必須明確標記：

[G] language guarantee
[L] library guarantee
[I] implementation behavior
[O] optimization opportunity
[M] measured result
[A] analytical inference
機器效率與人類可讀性不是固定的零和關係。

高階抽象可以：

- 消除重複；
- 提供更好的最佳化邊界；
- 讓 Library 專家集中處理低階細節；
- 透過型別與編譯器移除 Runtime 成本。

低階表面也可能：

- 破壞區域性；
- 產生不必要配置；
- 阻礙最佳化；
- 隱藏 Undefined behavior；
- 讓每位使用者重複處理安全與資源問題。

本文提出：

> K
> =
> (
> K⊠,

並將成本可見性分成：

> 【完全隱藏
> →
> 文件
> →
> API 命名
> →
> 型別／語法
> →
> Compiler 解釋
> →
> Runtime 觀察
> →
> 規格保證】

成熟設計不要求所有成本都在來源碼逐字顯示，而要求：

- 跨邊界與量級改變的成本可辨識；
- 安全與資源關係有穩定契約；
- 依最佳化器而變的成本可被檢查；
- Runtime 行為可被 Profile 與 Trace；
- 性能宣稱區分語言保證、實作機會與量測結果；
- 高階抽象在失效時具有可解釋降解路徑。

因此，PLDST 不再只將設計師分成「重效率」或「重可讀」。更精確的問題是：

> 他把哪些成本視為程式語義的一部分，哪些交給型別，哪些允許編譯器消除，哪些接受 Runtime 自適應；當抽象未能被消除時，他是否提供足夠證據，讓使用者知道成本真正發生在哪裡？

最終原則為：

> 【讓來源碼表達意圖
> ∧
> 讓高影響成本保持可辨識

語言：
版本：
編譯器／Runtime：
機制：
意圖可讀性：
成本可讀性：
執行時間：
記憶體：
配置／回收：
資料區域性：
派發：
安全檢查：
排程：
編譯／暖機：
尾延遲：
來源可見：
型別可見：
API 可見：
Compiler 報告：
Runtime 指標：
規格保證：
最佳化機會：
性能懸崖：
證據：
信心：

[R1] ISO/IEC JTC1/SC22/WG14, Rationale for International Standard—Programming Languages—C and ISO C working drafts.

— C 抽象機器、as-if 式可觀察語義、硬體映射與實作自由。

[R2] Bjarne Stroustrup, “Abstraction and the C++ Machine Model,” 2004／2005; “Foundations of C++,” 2012.

— C++ 機器模型、抽象與零額外成本原則。

[R3] Rust Project, The Rust Programming Language, “Performance in Loops vs. Iterators”; The Embedded Rust Book, “Zero Cost Abstractions.”

— Iterator、Closure、Typestate 與 Zero Sized Type 的最佳化案例。

[R4] Rob Pike, “Go at Google: Language Design in the Service of Software Engineering,” 2012; Go FAQ and Effective Go.

— Go 的效率、可讀性、垃圾回收、資料表示與組織工程。

[R5] Oracle, Java HotSpot Virtual Machine Performance Enhancements and Java Virtual Machine Technology Overview.

— 熱點編譯、Escape Analysis、Code cache 與 Runtime 最佳化。

[R6] Simon Marlow (ed.), Haskell 2010 Language Report.

— Haskell 非嚴格語義與 seq。

[R7] GHC User’s Guide, “Optimisation” and “Bang Patterns and Strict Haskell.”

— Strictness、Thunk、Full laziness、共享與記憶體駐留。

[R8] Julia Documentation, “Performance Tips,” “Functions,” and “Profiling.”

— Type stability、回傳型別推導、配置、Function barrier 與性能工具。

[R9] Go Project, Effective Go, Goroutine and concurrency examples.

— Goroutine 隱藏 Thread 管理複雜度，以及未限制建立所形成的資源問題。

[G] Language guarantee
[L] Library guarantee
[I] Implementation behavior
[O] Optimization opportunity
[M] Measured result
[A] Analytical inference
[V-S] Syntax-visible cost
[V-T] Type-visible cost
[V-A] API-visible cost
[V-C] Compiler-visible cost
[V-R] Runtime-visible cost
[V-G] Guaranteed cost

SECTION

D.1 C 與抽象機器

已重新核對 C 標準 Rationale 與 WG14 Working Draft：

- C 規格以 Abstract machine 描述程式行為；
- 實作只需在規定的可觀察點符合抽象語義，不必逐步執行來源碼所暗示的機器操作；
- 部分算術、表示與轉換確實容許依目標硬體效率決定；

- 本文因此只將 C 稱為「較高機器成本可見性」，沒有把它寫成來源碼與硬體指令的一一對應。

SECTION

D.2 C++ 零額外成本

已重新核對 Stroustrup 的〈Abstraction and the C++ Machine Model〉與〈Foundations of C++〉：

- Zero-overhead principle 的典型表述是「不用的不付費；使用的難以手寫得更好」；
- 其主要比較維度是 Runtime time 與 space，以及高階抽象相對於對應手寫低階方案的成本；
- 它不是編譯時間、錯誤診斷、標準複雜度、Monomorphization 或 Binary size 全部為零的保證；
- 本文已明確將其定位為設計原則與比較基線，而非每個程式、每個 Compiler 的個別證明。

SECTION

D.3 Rust Zero-Cost Abstraction

已重新核對 Rust Book 與 Embedded Rust Book：

- Rust 官方將 Iterator 稱為 zero-cost abstraction，並用特定 Benchmark 與編譯結果說明其可接近手寫迴圈；
- Embedded Rust 的 typestate 案例使用 Zero Sized Types，說明特定狀態標記可在 Runtime 無資料表示；
- 官方用語仍是「strives for」以及特定案例的結果；
- 本文沒有把它推廣成所有 Iterator chain、Trait、Closure 或安全抽象都必然在所有 Build profile 中完全消失。

SECTION

D.4 Go 的可讀性與 GC 成本

已重新核對〈Go at Google〉、Go FAQ 與 Effective Go：

- Go 的設計同時追求可讀、簡潔、可靠與高效率；
- 官方材料明確指出，懂得資料表示的程式設計者可減少配置與 Collector 壓力；
- Goroutine 隱藏了 Thread 建立與管理的許多複雜度，但不等於無配置、無排程或無生命週期成本；
- Effective Go 的並行 HTTP 範例確實警告：為每個請求直接建立 Goroutine 時，即使同時執行數受限，也可能累積大量等待中的 Goroutine。

SECTION

D.5 Java／HotSpot Escape Analysis

已重新核對 Oracle HotSpot Performance Enhancements：

- Escape Analysis 判斷新物件的使用範圍；
- Server Compiler 可消除可被 Scalar replacement 的物件配置與相關鎖；
- Oracle 文件沒有將這項最佳化表述為 Java 語言的一般 Stack allocation 保證；
- 本文因此將語言語義、HotSpot 實作最佳化與特定執行證據分開。

SECTION

D.6 Haskell 與 GHC

已重新核對 Haskell 2010 Report 與 GHC 9.14 User’s Guide：

- Haskell 函式應用預設為 Non-strict；
- seq、BangPatterns、Strict 與 StrictData 提供不同層級的嚴格度控制；
- GHC 明確指出，在高性能數值內圈中消除 Thunk 可能帶來巨大收益；
- Full laziness 增加 Sharing，也可能提高 Memory residency；
- GHC 並未完整實作所有理論上的 Full laziness transformation，官方也警告不能依賴其一致發生。

  本文因此沒有將 Laziness 單向描述為慢、快、省記憶體或浪費記憶體。

SECTION

D.7 Julia Type Stability

已重新核對 Julia 1.x 官方 Performance Tips、Functions、Types 與 Profiling 文件：

- Julia 通常不建議僅為性能而宣告函式回傳型別，而建議撰寫 Compiler 可推導回傳型別的 Type-stable function；
- Unexpected allocation 常是 Type instability 或小型暫時陣列等問題的警訊；
- @code_warntype、Allocation tracking 與 Profile 是官方建議工具；
- 這些是編譯器與程式撰寫的性能模型，不是所有動態 Julia 程式都自動獲得相同性能的語言保證。

SECTION

D.8 性能數字與版本

本文沒有固定寫入可能隨版本、硬體與 Benchmark 改變的絕對性能排行。

所有案例只用於說明：

語義保證
實作機會
診斷工具
量測證據
四者如何分離。

後續人物個案若涉及實際 Benchmark，仍須針對當次版本、Compiler、Build profile、硬體與資料集重新搜尋與測量。