安全約束與表達自由：語言應禁止多少錯誤，又應允許多少逃生？

Safety Constraints and Expressive Freedom: How Much Error Should a Language Forbid, and How Much Escape Should It Permit?

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-009.html

英文名稱： Safety Constraints and Expressive Freedom: How Much Error Should a Language Forbid, and How Much Escape Should It Permit?

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-009

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

程式語言安全經常被描述成一條單軸：

自由、低階、可做任何事 ←——————————→ 安全、受限、只能做被允許的事
這個模型同時誤解安全與自由。

安全不只是「編譯器拒絕更多程式」，而是對特定錯誤類別建立可說明的保證；自由也不只是「任何位元都能被任意操作」，而包括表達新抽象、連接外部系統、控制資源、實作 Runtime，以及在現有分析能力之外建立新機制的能力。

Rust 將 Safe Rust 與 Unsafe Rust 以 unsafe 邊界分開：unsafe fn、unsafe trait 等建立額外安全契約，unsafe {}、unsafe impl 等表示程式設計者聲稱已履行契約；unsafe 並不宣告任意行為正確，而是把原本由編譯器承擔的部分證明義務轉交給人。SPARK 透過 Ada 子集、Flow analysis、契約與 GNATprove 排除或證明多類運行期錯誤，同時對 Unchecked_Conversion、外部呼叫與不支援構造建立明確限制。Haskell 以 IO 區隔純運算與效果，但 unsafePerformIO 允許突破此邊界；官方文件因此要求極強的使用前提，並警告副作用順序可能不確定。TypeScript 為 JavaScript 生態提供漸進型別安全，卻刻意保留 any、Type assertion 與部分不健全相容規則；unknown 則作為較安全的逃生入口，要求先 Narrow 才能操作。C# 以可驗證安全程式作為主要模式，透過 unsafe Context 開放 Pointer、Function pointer 與手動 Memory 操作；Go 的 unsafe Package 則明確提供繞過型別安全的操作，並警告相關程式可能不可攜，且不受 Go 1 相容承諾完整保護。[R1][R2][R3][R4][R5][R6]

這些語言證明：成熟的安全設計通常不是消滅所有危險能力，而是建立一個 安全包絡（Safety Envelope），並控制危險能力如何穿越包絡。

本文提出 安全—自由配置模型（Safety–Freedom Allocation Model, SFAM）：

> cal-S
> =
> (
> G,
> H,
> O,
> C,
> A,
> F,
> R
>)

其中：

- G：Guarantees，安全子語言提供的保證；
- H：Hatches，逃生口與不安全操作；
- O：Obligations，轉移給呼叫者或實作者的證明義務；
- C：Containment，危險能力的局部化與封裝；

- A：Auditability，標記、文件、工具與審計能力；
- F：Fallback，失敗、未知與無法證明時的處理；
- R：Recovery，事故後的圍堵、恢復與制度修正。

  本文將安全拆成八類：

- 記憶體安全；
- 型別與表示安全；
- 初始化與資料有效性；
- 控制流完整性；
- 並行與資料競爭安全；
- 效果與純度邊界；
- 資源與終止安全；
- 外部互操作與供應鏈信任。

  並提出七種逃生形式：

- 安全但低階的受控 API；
- 顯式局部不安全區塊；
- 呼叫者承擔契約的不安全函式；
- 未檢查轉換與型別斷言；
- 外部函式與 ABI 邊界；
- 可選關閉的檢查與漸進降級；
- 全域或非局部不健全機制。

  本文核心命題為：

> 【存在逃生口
> not⇒
> 語言沒有安全保證】

真正關鍵是：

> 【逃生口是否局部、顯式、可審計、可封裝，
>
> 以及安全呼叫者是否仍能依賴穩定保證。】

相反地：

> 【禁止危險語法
> not⇒
> 系統安全】

因為規格錯誤、外部服務、資源耗盡、供應鏈、邏輯漏洞與錯誤的安全抽象，仍可能存在。

關鍵詞： 程式語言安全、表達自由、逃生口、Unsafe Rust、SPARK、unsafePerformIO、TypeScript any、C# unsafe、Go unsafe、PLDST

SECTION

一、記憶體安全 G_m

涵蓋：

- Use-after-free；
- Double free；
- Dangling pointer；
- Out-of-bounds；
- Invalid alias；
- Data race 導致的未定義行為；

- 錯誤資料表示；
- 任意 Memory corruption。

一種語言可以高度記憶體安全，卻仍允許：

- 無限迴圈；
- 業務邏輯錯誤；
- SQL injection；
- 過量配置；
- 外部服務誤用。

SECTION

二、型別與表示安全 G□

涵蓋：

- 值是否符合宣告型別；
- Pattern 是否完整；
- Dynamic cast 是否檢查；
- 表示轉換是否有效；
- 外部資料是否驗證；
- ABI 是否一致。

未檢查轉換往往不是只「改變型別名稱」，而是要求程式設計者證明底層表示真的相容。

SECTION

三、初始化與資料有效性 G□

涵蓋：

- 變數是否初始化；
- Scalar 是否具有有效表示；
- 物件不變量是否成立；
- Partial initialization；
- 外部讀入資料；
- null／Optional；
- Invalid enum value。

SPARK 對資料有效性與 Unchecked_Conversion 的處理顯示：即使語言有強型別，外部表示與未檢查轉換仍需要額外證明或保守標記。[R2]

SECTION

四、控制流完整性 G_c

涵蓋：

- 跳轉與 Return address；
- Exception；
- Panic；
- Cancellation；
- 非局部跳轉；
- Foreign callback；
- Stack unwinding；
- Control-flow integrity。

記憶體安全通常有助於控制流安全，但不等同完整的控制流與業務流程正確性。

SECTION

五、並行與資料競爭安全 G_d

涵蓋：

- Data race；
- Deadlock；
- Atomicity；
- Message ordering；
- Cancellation；
- Shared mutable state；
- Scheduler interaction。

  Rust 的型別與所有權能排除多類不安全共享，但仍不能自動消除：

- Deadlock；
- Starvation；
- 邏輯 Race；
- 分散式一致性錯誤。

SECTION

六、效果與純度安全 G_e

涵蓋：

- I/O 是否可見；
- 函式是否純；
- 是否讀寫全域狀態；

- 是否丟出例外；
- 是否阻塞；
- 是否使用時間、亂數與環境；
- 是否可重放。

  Haskell IO 把效果放入型別，但 unsafePerformIO 能把 IO a 映射為表面純值 a。因此其安全前提不是 Memory safety，而是參照透明性、效果順序與最佳化穩定性。[R3]

SECTION

七、資源與終止安全 G_r

涵蓋：

- 記憶體上限；
- Stack；
- CPU；
- 配額；
- 終止；
- 即時 Deadline；
- 檔案描述符；
- Goroutine／Thread leak；
- 外部請求。

  多數「Safe」語言並不保證終止或資源有界。

SECTION

八、外部互操作與供應鏈安全 G_x

涵蓋：

- FFI；
- Native Library；
- ABI；
- Unsafe package；
- 反序列化；
- 編譯器插件；
- 巨集；
- Dependency；
- Runtime；
- Build script。

  安全子語言通常建立在更大的 Trusted Computing Base 上。若 Runtime、Compiler、FFI 或 Native dependency 不可靠，語言內部保證可能被破壞。

SECTION

九、安全向量

$$
\begin{aligned}
&G \\
&= \\
&(\\
&G_m, \\
&G\square, \\
&G\square, \\
&G_c, \\
&G_d, \\
&G_e, \\
&G_r, \\
&G_x
\end{aligned}
$$

PLDST 不使用一個「安全分數」抹去不同保證。

SECTION

十、表達自由

能否自然表達：

- 新資料模型；
- 高階抽象；
- DSL；
- 並行；
- 資源；
- 外部系統；
- Runtime；
- 編譯器；
- Kernel；
- 硬體。

SECTION

十一、實作自由

語言實作者能否：

- 建立垃圾回收器；
- 實作同步原語；
- 操作 Pointer；
- 連接 C；

- 使用 SIMD；
- 控制布局；
- 建立自訂配置器；
- 寫作業系統與 Driver。

安全語言若完全不能實作自己的安全基礎，就只能永遠依賴外部不安全語言。

SECTION

十二、遷移自由

漸進型別語言保留：

- 舊程式逐步加入型別；
- 無法建模的外部 Library；
- 動態資料；
- JavaScript 生態；
- Prototype。

TypeScript 的 any 具有重大風險，但也是從既有 JavaScript 世界向靜態分析過渡的重要橋樑。[R4]

SECTION

十三、研究自由

新資料結構、編譯技術與資源模型可能暫時超出現有型別系統。

若完全禁止：

- Raw memory；
- 自訂表示；
- 未驗證原語；

- Runtime Hook；

語言可能失去自我擴張能力。

SECTION

十四、逃生自由與任意破壞不同

成熟逃生口通常不是：

關掉所有檢查，任意執行。

而是：

你可以執行一組額外操作，
但必須承擔明確且可審查的安全契約。

SECTION

十五、安全但低階的受控 API

有些能力看似低階，仍可由安全 API 提供：

- Safe slice；
- Checked pointer wrapper；
- Span；
- Arena；
- Pinned buffer；
- Managed handle；
- Atomic；
- Capability object。

其目標是：

> 將一次性不安全實作封裝成可重複使用的安全介面。

SECTION

十六、顯式局部不安全區塊

代表：

```
unsafe {
  // limited operations
}
```

理想性質：

- 範圍小；
- 操作可列舉；
- 周圍仍受普通語言規則約束；
- Reviewer 能集中審查；
- 能以 Lint 或 Policy 統計；
- 可由 Safe wrapper 隔離。

Rust Reference 將 unsafe 描述為建立或履行額外安全義務，而不是取消所有語言規則。[R1]

SECTION

十七、呼叫者承擔契約的不安全函式

unsafe fn 的含義是：

> 呼叫者必須保證額外前置條件。

例如：

- Pointer 有效；
- 對齊正確；

- 範圍未重疊；
- 生命週期足夠；
- 外部資源狀態成立。

  如果條件無法在型別中完整表達，就必須透過 Safety documentation 保留。

SECTION

十八、未檢查轉換與型別斷言

包括：

- Unchecked_Conversion；
- Type assertion；
- transmute；
- Reinterpret cast；
- any；
- Double assertion；
- Raw representation view。

  這些機制可能：

- 僅告訴 Checker「相信我」；
- 真正重新解釋位元；
- 插入 Runtime check；
- 完全跳過檢查。

  PLDST 必須區分，不能都叫 Cast。

SECTION

十九、外部函式與 ABI 邊界

FFI 的安全需要證明：

- ABI；
- Calling convention；
- Pointer；
- Ownership；
- Thread；
- Error；
- Callback；
- Layout；
- Lifetime；
- Unwind。

  Rust 2024 要求 extern Block 明示 unsafe，正是要讓定義外部項目本身具有安全義務這件事更可見。[R7]

SECTION

二十、可選關閉的檢查與漸進降級

例如：

- TypeScript strictNullChecks；
- any；
- Compiler warning disable；
- Proof Level；

- Runtime assertion profile；
- Unchecked build；
- Feature gate。

其價值是：

- 遷移；
- 性能；
- 相容；
- 分階段證明。

風險是：

- 不同模組保證不同；
- 使用者誤以為全系統已安全；
- 弱保證可沿 API 傳播。

SECTION

二十一、全域或非局部不健全機制

例如：

- 全域關閉型別檢查；
- 無邊界修改 Compiler；
- unsafePerformIO 污染純介面；
- 任意 Native plugin；
- 未標記 ABI；
- 全域 mutable state。

這類逃生口最難審計，應具有最高准入門檻。

SECTION

二十二、安全子語言

令 Safe fragment 為：

$$
L_s
\subseteq
L
$$

在特定 Trusted Computing Base 與前提下：

$$
\text{Program} \in L_s
\Rightarrow
\text{Guarantees}(G)
$$

這些保證必須明確列出，不能只說「Safe」。

SECTION

二十三、不安全擴展

令逃生能力集合為：

$$
H=\{h□,h□,\ldots,h□\}
$$

每個 h□ 都應有：

$$
h□
=
(
\text{Operation},
\text{Precondition},
\text{Postcondition},
\text{Invariant},
\text{Scope},
$$

SECTION

二十四、安全封裝

若不安全實作 u 對外暴露 Safe API a：

> u
> xrightarrowproof obligation
> a

則需要證明：

> ∀ x∈ ValidInputs(a):
> Execution(a,x)
> not⇒
> Violation(G)

這可能依靠：

- 人工推理；
- 測試；
- Miri／Sanitizer；
- Proof；
- Code review；
- Representation invariant；
- Fuzzing；
- Platform guarantee。

SECTION

二十五、安全包絡不變量

> SafeCaller
> +
> SafeInputs
> ⇒
> SafeGuarantees

即使內部使用逃生口，安全呼叫者也不應被迫承擔未公開的額外義務。

這是 Safe abstraction 的核心。

SECTION

二十六、錯誤封裝

若 Safe API 實際要求：

- 特殊呼叫順序；
- 隱藏有效期；
- 未文件化對齊；
- 不可見 Thread 限制；
- 外部全域狀態；

它只是「語法上 Safe」，而非真正 Sound。

SECTION

二十七、語言證明

由型別與語義保證：

- Safe references；
- Borrow rules；
- Definite initialization；

- Exhaustiveness；
- Pure type；
- Capability；
- Effect。

SECTION

二十八、工具證明

由：

- Static analyzer；
- Model checker；
- SMT；
- GNATprove；
- Miri；
- Sanitizer；
- Linter；

提供。

工具必須區分：

proved
checked on tested path
warning
unknown
unsupported
tool unavailable

SECTION

二十九、人工證明

不安全區塊的正確性常依靠：

- Safety comment；
- Representation invariant；
- Code review；
- API contract；
- 外部標準；
- 對平台的假設。

人工證明不是形式證明，但仍應結構化。

SECTION

三十、環境假設

例如：

- C 函式遵守 ABI；
- OS 不違反 Memory mapping；
- 外部 Pointer 仍有效；
- Compiler 正確；
- Native Library 無漏洞；
- Hardware 符合 Memory model。

安全保證永遠相對於 Trusted assumptions。

SECTION

三十一、局部契約式安全

代表：

- Rust。

  風格：

- Safe 子語言提供強保證；
- unsafe 只開放特定額外操作；
- 不安全義務局部標記；
- Safe wrapper 可重新封裝；
- 可使用 #![forbid(unsafe_code)] 禁止整個 Crate 使用不安全程式碼。[R1]

  優勢：

- 低階系統能力與 Safe API 共存；
- 危險區域可搜尋與審查；
- 程式庫可集中 Trusted code；
- 一般使用者維持 Safe 體驗。

  風險：

- Soundness 依賴不安全實作者；
- Safety contract 可能文件不足；
- Unsafe code 仍受複雜 Alias、Provenance 與 FFI 規則限制；
- Safe wrapper 的一個漏洞可能影響大量呼叫者。

SECTION

三十二、受限子集與證明式安全

代表：

- SPARK／Ada。

風格：

- 排除或限制難以分析的 Ada 功能；
- 使用 Flow analysis、Contract 與 Proof；
- 依保證等級逐步提高證據；
- 對 Unchecked conversion、外部程式與不支援構造保守處理。[R2]

優勢：

- 高保證與認證領域；
- 保證範圍可分級；
- Proof obligation 可追蹤；
- 設計與實作責任清楚。

風險：

- 規格與 Proof 成本高；
- 未分析外部程式形成 TCB；
- 某些低階構造只能保守建模；
- Proof 仍相對於 Contract 與工具假設。

SECTION

三十三、純度邊界與極端逃生

代表：

- Haskell IO／unsafePerformIO。

風格：

- 一般效果透過 IO 類型顯式化；
- 純程式依賴參照透明；

- 少數底層機制可以突破邊界；
- 逃生口被明確標記為 Unsafe 且伴隨嚴格警告。[R3]

優勢：

- 純度與效果分離；
- 大部分程式保持可推理；
- Runtime／Library 可實作特殊抽象。

風險：

- 一旦錯誤封裝，表面純函式可能具有隱藏效果；
- Compiler 可重排、共享或消除表達式；
- 副作用順序可能不確定；
- 錯誤可能跨越遠離逃生點的位置表現。

SECTION

三十四、漸進與選擇性健全

代表：

- TypeScript。

風格：

- 與既有 JavaScript 相容；
- 允許逐步增加型別；
- 提供 Strict flags；
- 保留 any、Assertion 與結構相容中的有意不健全；
- 提供 unknown 作為需要 Narrow 的安全頂型別。[R4]

優勢：

- 遷移現有生態；
- 可逐步提高安全；
- 動態資料與第三方 Library 容易接入；
- 工具價值高。

風險：

- any 會沿資料流傳播並關閉檢查；
- Type assertion 不執行 Runtime 驗證；
- 不同專案 Strictness 不同；
- 型別正確不代表 JavaScript Runtime 資料符合宣告；
- 結構型別的刻意不健全需要使用者理解。

SECTION

三十五、安全預設與顯式 Unverifiable 區域

代表：

- C#／.NET。

風格：

- 一般程式以 Managed memory、Reference safety 與 Runtime checks 為基礎；
- 在既有穩定模型中，Pointer、Function pointer 與部分 Memory 操作需進入 unsafe Context；
- 專案需允許編譯 Unsafe code；
- 官方建議優先尋找 Safe API，只有必要時才採用不安全替代。[R5]

截至本文日期，C# 15／.NET 11 正在預覽一套更新的 Memory safety model，嘗試讓 unsafe 更直接標記實際未受管理的 Memory access，並把部分安全審計義務傳遞給呼叫者。本文不將 Preview 行為當成現行穩定 C# 的既定規格。

優勢：

- 主流應用維持 Managed safe model；
- Native interop 與高性能路徑仍可實作；
- Unsafe 區域可被 Compiler 與 Review 辨識。

風險：

- Unverifiable 不等於一定危險，但也不再由 .NET 驗證；
- Pointer 可繞過 Bounds 與 GC 安全；
- Raw IL、Interop 與 Runtime helper 擴大 TCB；
- 性能理由可能導致過早使用 Unsafe。

SECTION

三十六、Package 級型別安全逃生

代表：

- Go unsafe。

風格：

- 核心語言與一般 Standard library 維持簡單型別安全模型；
- 低階表示操作集中於特殊 Package；
- Import 直接暴露使用；
- 官方明確標示不可攜與相容性風險。[R6]

優勢：

- 語法核心不增加完整 Pointer arithmetic 子語言；
- 低階 Runtime、Syscall、序列化與最佳化仍有出口；
- Package import 可被靜態搜尋。

風險：

- unsafe.Pointer 與 uintptr 轉換具有 GC／Pointer liveness 限制；
- 程式可依賴實作布局；
- Go 1 Compatibility 不完整保護 Unsafe 程式；
- Package 級逃生本身不保證危險被局部封裝成 Safe API。

SECTION

三十七、Rust：unsafe 是義務標記，不是安全關閉按鈕

Rust Reference 指出，unsafe 一方面可標記額外安全條件的存在，另一方面表示程式設計者聲稱已履行條件。[R1]

Unsafe Rust 額外允許的能力包括：

- Dereference raw pointer；
- Access union field；
- Access／modify mutable static；
- Call unsafe function；
- Implement unsafe trait；
- 部分 FFI 與 unsafe attribute。

但 unsafe 不代表：

- 可以違反語法；
- 可以忽略型別；
- Undefined behavior 變成合法；
- Safe code 保證自動失效；
- Compiler 替你證明契約。

Rust Reference 對 Undefined behavior 的說明明確指出：Unsafe code 中發生 UB 仍是錯誤，只是避免它的責任落在程式設計者。[R8]

SECTION

三十八、SPARK：排除、分析與證明的分層

SPARK 不只依賴一個「Safe」關鍵字，而是透過：

- Ada Feature restriction；
- Flow analysis；
- Contract；
- Data validity；
- Proof Level；
- GNATprove；
- Review；

  建立分層保證。

  Silver Level 可證明多類運行期錯誤不存在；Gold Level 進一步處理更完整的資料與控制流完整性目標。[R9] [R10]

  但：

- Unchecked_Conversion 可能導致 GNATprove 對結果資訊不足；
- 某些構造不支援；
- Storage error、Compiler、Target、外部呼叫仍需要額外考量；
- Proof 成功不代表需求規格正確。

SECTION

三十九、Haskell：unsafePerformIO 對最佳化模型提出契約

unsafePerformIO :: IO a -> a 看似只是移除 IO，實際卻要求：

- 計算應在語義上可視為純；
- 結果不應依賴執行環境；
- 副作用不能依賴順序；
- 需要考慮 Inlining、Common subexpression elimination 與 Sharing；
- 多執行緒下還有重複或競態風險。[R3]

它適合實作少數底層抽象，不適合作為逃避 IO 型別的一般工具。

SECTION

四十、TypeScript：any 與 unknown 代表兩種自由

any 表示：

- 可以存入任何值；
- 幾乎可以執行任何操作；
- 型別檢查在相關資料流中被大幅削弱。

unknown 表示：

- 可以接收任何值；
- 但使用前必須 Narrow、檢查或 Assertion。

因此：

> any
> =
> 自由操作
> +
> 放棄檢查

> unknown
> =
> 自由接收
> +
> 延後證明

TypeScript 官方將 unknown 稱為 any 的 type-safe counterpart，並建議不需要 any 時避免使用它。[R4]

SECTION

四十一、C#：Unsafe 不是「危險」的同義詞

Microsoft 文件明確說明：Unsafe code 不一定危險，它是 .NET 工具無法驗證安全性的程式碼。[R5]

這個區分重要：

- 某段 Pointer 程式可能人工證明正確；
- 某段 Safe C# 也可能有邏輯或安全漏洞；
- unsafe 表示驗證責任轉移，而不是事故已發生。

  但官方最佳實務同樣強調：

- Unsafe 可繞過安全檢查；
- 可能造成 Memory corruption；
- 應先尋找或提出 Safe API；
- 需謹慎且限於必要場景。[R11]

SECTION

四十二、Go：unsafe 連相容性也一起放棄部分保證

Go Specification 與 unsafe Package 文件指出：

- 使用 unsafe 的 Package 必須人工審查型別安全；

- 可能不可攜；
- 不受 Go 1 Compatibility guidelines 完整保護。[R6]

  這表示逃生口不只降低型別保證，也可能降低：

- 平台穩定；
- 版本穩定；
- GC 行為穩定；
- 表示穩定。

SECTION

四十三、局部性 C_l

逃生口是否可以限制在：

- 一個表達式；
- 一個 Block；
- 一個 Function；
- 一個 Module；
- 一個 Package；
- 整個 Program。

  越局部越容易審計。

SECTION

四十四、可搜尋性 A_s

能否使用：

- Keyword；

- Import；
- Attribute；
- Compiler flag；
- Lint；
- Manifest；

找出全部逃生點。

SECTION

四十五、契約完整性 O_c

每個逃生點是否記錄：

- Safety preconditions；
- Representation invariant；
- Ownership；
- Lifetime；
- Thread；
- Alignment；
- Failure；
- Platform；
- Evidence。

SECTION

四十六、封裝性 C_e

危險能力是否可以：

- 隱藏於 Private module；
- 暴露 Safe API；
- 禁止一般呼叫者進入；
- 減少重複人工證明。

SECTION

四十七、傳染性 T

令逃生能力的傳播範圍為：

> T(h)
> =
> 需要知道或承擔該義務的呼叫者集合

理想 Safe abstraction：

> T(h)≈ InternalReviewers

而非：

> T(h)=AllUsers

SECTION

四十八、可撤銷性 R_v

能否：

- 用 Safe API 取代；
- 透過 Lint 禁用；

- 在高保證 Profile 中禁止；
- 逐步遷移；
- 將 TCB 縮小。

SECTION

四十九、禁止主義

假設只要禁止足夠多操作就安全。

問題：

- 使用者可能轉向 FFI；
- 形成外部程式碼生成；
- 把危險移到 Runtime；
- 無法實作必要系統能力；
- 規格與邏輯錯誤仍存在。

SECTION

五十、逃生口正常化

當 unsafe、any、Assertion 或 Unchecked conversion 成為日常做法時：

- 安全子語言縮小；
- Review 疲勞；
- 人工證明重複；
- 保證無法推理；
- 新手無法識別風險。

SECTION

五十一、安全標記儀式化

只寫：

```
unsafe
// SAFETY: trust me
```

不代表契約成立。

安全註解必須回答：

- 為何 Pointer 有效？
- 誰維持 Invariant？
- 哪些操作可能失效？
- 跨 Thread 是否安全？
- 何時釋放？
- 平台假設是什麼？

SECTION

五十二、安全封裝洩漏

Safe API 內部使用不安全程式碼，卻讓 Safe caller 能以合法輸入觸發 UB。

這是 Soundness bug，不能歸咎於呼叫者。

SECTION

五十三、漸進安全停滯

TypeScript／Gradual typing 專案可能永久停留在：

- any；
- 關閉 Strict flags；

- 大量 Assertion；
- 未驗證外部資料。

漸進路徑需要：

- 指標；
- CI；
- Boundary validation；
- Debt budget；
- 遷移策略。

SECTION

五十四、形式證明過度宣稱

Proof 只能證明：

> Model+Assumptions+Spec
> ⇒
> Property

不能自動證明：

- Spec 符合現實；
- Compiler 無 Bug；
- Hardware 無故障；
- 外部依賴可信；
- 安全政策合理。

SECTION

五十五、性能作為無限豁免

以「性能」為由使用逃生口，卻沒有：

- Benchmark；
- Profile；
- Safe 替代比較；
- Regression；
- 平台矩陣；
- 審計。

這是低品質工程，不是機器現實主義。

SECTION

五十六、先建立安全表達，再提供逃生

安全 API 若能表達大多數需求，逃生口才有可能保持稀缺。

SECTION

五十七、逃生口應縮小證明表面

好的逃生機制不是讓更多程式變得不安全，而是讓少量底層程式承擔證明，供大量 Safe code 使用。

> UnsafeCoredownarrow
>
> SafeSurfaceuparrow

SECTION

五十八、危險操作應與證明義務同時出現

語法或文件必須把：

能力
+

前置條件
+
責任主體
綁在一起。

SECTION

五十九、未知應保持未知

無法證明時，工具不應輸出「安全」。

應輸出：

- Unknown；
- Unsupported；
- Requires review；
- Runtime check；
- Unsafe boundary。

SECTION

六十、外部輸入應先驗證再降權

外部資料進入型別系統時：

> bytes
> →
> unknown
> →
> validated
> →
> typed

而不是直接 Assertion 成可信型別。

SECTION

六十一、逃生口應有 Profile

可建立：

General profile
No-unsafe profile
Verified profile
Embedded profile
FFI profile
Migration profile
讓不同風險領域使用不同能力集合。

SECTION

六十二、安全保證必須列出 TCB

至少列出：

- Compiler；
- Runtime；
- Unsafe modules；
- Native libraries；
- Build scripts；
- Generated code；
- Proof tools；
- Platform assumptions。

SECTION

六十三、事故應反饋到安全抽象

若某類 Unsafe bug 重複出現，應考慮：

- 新 Safe API；
- 新型別；
- 新 Lint；
- 新 Proof；
- 新工具；
- 禁止舊模式。

SECTION

六十四、安全保證指紋

Memory
Type
Initialization
Control
Concurrency
Effect
Resource
Interop

SECTION

六十五、逃生口指紋

Local block
Unsafe function
Unchecked conversion
Dynamic top type
FFI
Compiler flag
Unsafe package
Global effect escape

SECTION

六十六、證明義務指紋

Compiler-proved
Runtime-checked
Tool-proved
Caller-proved
Reviewer-audited
Environment-assumed
Unspecified

SECTION

六十七、設計師比較問題

- 他首先保護哪種安全？
- 哪些錯誤被語言禁止？
- 哪些只被警告？
- 哪些可在 Runtime 檢查？
- 哪些能力必須逃生？
- 逃生是 Expression、Module 還是全域？
- 義務交給誰？
- 是否能封裝成 Safe API？
- 是否允許禁用逃生口？
- 如何處理外部系統與既有生態？
- 安全失敗時是拒絕、降級、Unknown 還是繼續？
- 安全主張是否列出 TCB？

SECTION

六十八、不能只給「安全／自由」分數

一位設計者可能：

- 高記憶體安全；
- 中型別健全；
- 高 FFI 自由；
- 低隱式逃生；
- 高形式證明；
- 低資源保證。

必須保留多維描述。

SECTION

六十九、輸入

designer
language
version
safety_feature
escape_hatch
official_reference
compiler_options
tooling
governance_documents

SECTION

七十、分析管線

重新網路搜尋
→ 安全保證向量
→ 逃生操作抽取
→ 證明義務
→ 局部性與傳染性
→ Safe wrapper 檢查
→ TCB 抽取
→ Unknown／unsupported 處理
→ 事故與反例搜尋
→ 第二輪校對
→ 風格報告

SECTION

七十一、JSON 雛形

```
{
 "mechanism": "unsafe block",
 "guarantees_relaxed": [
  "compiler verification of selected memory-safety preconditions"
 ],
 "still_enforced": [
  "syntax",
  "ordinary type checking",
  "lifetime rules not specifically escaped"
 ],
 "obligation_holder": "unsafe code author",
 "scope": "lexical block",
 "auditability": {
  "searchable_keyword": true,
  "safety_documentation": "required by policy"
 },
 "safe_encapsulation": {
  "possible": true,
  "caller_obligation": "none if wrapper is sound"
 },
 "failure": "undefined behavior if contract is violated"
}
```

SECTION

七十二、SKILL 禁止事項

不得：

- 把 Memory safety 寫成完整系統安全；
- 把 unsafe 寫成關閉所有規則；
- 把 Safe wrapper 寫成自動 Sound；
- 把 SPARK Proof 寫成需求正確性；
- 把 unsafePerformIO 當一般效能技巧；

- 把 Type assertion 寫成 Runtime validation；
- 把 unknown 與 any 寫成相同；
- 把 C# Unverifiable 直接等同已發生 Memory corruption；
- 把 Go unsafe 視為受完整 Go 1 相容保證；
- 忽略 FFI、Compiler、Runtime 與 Native Library TCB；
- 將 Preview 安全模型寫成 Stable 規格；
- 不標示版本與 Profile。

SECTION

七十三、安全性依賴威脅模型

嵌入式、瀏覽器、金融、遊戲、Kernel 與教學環境的必要保證不同。

SECTION

七十四、安全子集可能過小

若 Safe fragment 無法有效完成工作，使用者會大量逃生，反而降低整體安全。

SECTION

七十五、證明成本具有規模效應

Proof、Audit 與 Safe wrapper 在 Library 層可攤銷；若每個應用都重做，成本可能不可接受。

SECTION

七十六、工具與規格會演化

Rust Unsafe 規則、C# Memory safety model、SPARK 支援範圍、TypeScript Strict flags 都可能改變。每篇個案需重新搜尋當時版本。

SECTION

七十七、安全文化也很重要

即使語言提供良好標記，若社群：

- 習慣忽略警告；
- 不寫 Safety contract；
- 缺少 Review；
- 以性能為由濫用；
- 不維護 TCB；

  形式邊界仍會失效。

  安全與自由不是一條只能選一端的直線。

  真正成熟的設計問題是：

- 哪些錯誤可以由語言可靠禁止？
- 哪些需要 Runtime 檢查？
- 哪些需要工具與 Proof？
- 哪些只能由領域專家承擔？
- 哪些低階能力必須存在，語言才能實作自己的 Runtime、Interop 與高性能抽象？
- 如何讓少量危險程式碼支撐大量安全程式，而不是讓危險義務傳染整個系統？

  本文提出：

> cal-S
> =
> (
> G,
> H,
> O,

其成熟判準為：

> 【保證明確
> +
> 逃生稀缺
> +
> 義務可見
> +
> 危險局部
> +
> 封裝可靠
> +
> 證據可審
> +
> 失敗誠實】

因此：

> 【EscapeHatch
> not⇒
> NoSafety】

前提是：

> 【UnsafeImplementation
> +
> ProvenInvariant
> →
> SoundSafeInterface】

同時：

> 【SafeSyntax
> not⇒
> SafeSystem】

因為安全仍受：

- 規格；
- TCB；
- 外部系統；
- 資源；
- 供應鏈；
- 組織；

  限制。

  PLDST 對設計師最關鍵的問題，不再是：

> 他偏好安全，還是偏好自由？

而是：

> 他願意由語言排除哪些風險、哪些風險交給 Runtime 或工具、哪些能力必須保留逃生；當他把證明責任交還給人時，是否同時提供局部邊界、清楚契約、審計工具與重新封裝成安全抽象的路徑？

最終原則為：

> 【讓安全成為預設
> ∧
> 讓低階能力仍可實作
> ∧
> 讓每一次逃生都帶著可見的證明義務】

語言：
版本／Profile：
安全保證：
不保證事項：
逃生口：
允許操作：
證明義務：
責任主體：
作用範圍：

是否傳染：
能否封裝：
Safe caller 是否增加義務：
審計工具：
Runtime 檢查：
Proof：
外部假設：
TCB：
相容性影響：
失敗模式：
替代 Safe API：
證據：
信心：
需求：
為何 Safe fragment 不足：
現有 Safe 替代：
性能／互操作證據：
最小逃生範圍：
Safety contract：
Representation invariant：
平台假設：
測試：
Fuzz：
Sanitizer／Miri／Proof：
Review owner：
Safe wrapper：
下游義務：
撤回計畫：

[R1] Rust Project, The Rust Reference, “The unsafe keyword,” “Unsafety,” “Behavior considered undefined,” and The Rustonomicon, “How Safe and Unsafe Interact.”

— unsafe 建立／履行義務、Safe／Unsafe 邊界、Unsafe operations 與 UB 責任。

[R2] AdaCore, SPARK User’s Guide, “Language Restrictions,” “Specification Features,” “Applying SPARK in Practice,” and GNATprove limitations.

— SPARK 子集、Unchecked conversion、資料有效性、Flow analysis、Proof 與不支援範圍。

[R3] GHC／Haskell Base Libraries, System.IO.Unsafe and GHC.IO.Unsafe.

— unsafePerformIO 的純度前提、副作用順序、最佳化與多執行緒注意事項。

[R4] TypeScript Documentation, “Type Compatibility,” “Everyday Types,” strictNullChecks, TypeScript 3.0 unknown, and Declaration File Do’s and Don’ts.

— 有意不健全、any、unknown、Null safety 與遷移邊界。

[R5] Microsoft Learn, “Unsafe code, pointer types, and function pointers,” and “Unsafe code best practices.”

— C# 可驗證安全模式、Unsafe Context、Pointer 與風險治理。

[R6] Go Project, The Go Programming Language Specification and Package unsafe documentation.

— 繞過型別安全、人工審查、不可攜與相容性限制。

[R7] Rust Edition Guide, “Unsafe extern blocks,” Rust 2024.

— External definition 的 Safety obligation 與 unsafe extern 顯式化。

[R8] Rust Reference, “Behavior considered undefined.”

— Unsafe code 中 UB 仍不合法，責任轉移不等於行為合法化。

[R9] AdaCore, SPARK Silver Level documentation.

— 證明多類運行期錯誤不存在的保證範圍。

[R10] AdaCore, SPARK Gold Level documentation.

— Program integrity、資料與控制流安全邊界。

[R11] Microsoft Learn, “Unsafe code best practices.”

— Unsafe 可繞過 Memory safety，應優先尋找 Safe API 並限制必要用途。

[G-M] Memory guarantee
[G-T] Type guarantee
[G-I] Initialization guarantee
[G-C] Control guarantee
[G-D] Data-race／concurrency guarantee
[G-E] Effect guarantee
[G-R] Resource guarantee
[G-X] Interop guarantee
[H-B] Unsafe block
[H-F] Unsafe function／caller contract
[H-C] Unchecked conversion
[H-A] Any／assertion
[H-X] FFI
[H-P] Unsafe package
[H-G] Global escape
[O-C] Compiler obligation

[O-R] Runtime check
[O-T] Tool proof
[O-H] Human proof
[O-E] Environment assumption

SECTION

E.1 Rust unsafe 的兩種義務

已重新核對 Rust Reference、Rustonomicon 與 Rust 2024 Edition Guide：

- unsafe fn、unsafe trait、unsafe static 等可用來宣告額外安全條件；
- unsafe {}、unsafe impl 等表示程式設計者聲稱已履行其他位置所定義的安全條件；
- unsafe 只開放特定操作，不會取消語法與整個型別系統；
- Unsafe code 造成 Undefined behavior 仍然是錯誤，只是避免 UB 的證明責任從 Compiler 移給程式設計者；
- #![forbid(unsafe_code)] 可以用於禁止 Crate 中的 Unsafe code；
- Rust 2024 要求 extern Block 明示 unsafe，是為了讓外部宣告本身的 Safety obligation 更清楚。

  本文因此沒有把 unsafe 描述成全域「關閉安全模式」。

SECTION

E.2 SPARK 保證等級

已重新核對 2026 年 SPARK User’s Guide：

Stone：有效 SPARK
Bronze：初始化與正確資料流
Silver：Absence of Run-Time Errors
Gold：關鍵完整性屬性
Platinum：完整功能需求證明

- Silver 的核心目標是證明不發生指定範圍內的意外運行期例外；
- Gold 不是「所有程式完全正確」，而是對關鍵完整性屬性提供證明；
- Platinum 成本很高，官方也說實際較少適用；

- Unchecked_Conversion 的結果可能無法被 GNATprove 精確掌握；
- 不支援構造會被報錯或需額外假設。

本文已避免把 SPARK、Ada 與特定 Proof Level 混成同一種絕對安全聲明。

SECTION

E.3 Haskell unsafePerformIO

已重新核對最新 GHC Base Library 與 Safe Haskell 文件：

- 若 unsafePerformIO 包裝的運算具有副作用，相對於主 I/O 路徑與其他 unsafePerformIO 呼叫的副作用順序可能不確定；
- Compiler Inlining、Common-subexpression elimination 與 Sharing 會影響執行次數及時機；
- unsafeDupablePerformIO 甚至可能在多處或多執行緒執行，某次執行也可能中途被中斷；
- Safe Haskell 的 Safe dialect 會排除 unsafePerformIO 等可破壞型別或抽象邊界的機制；
- Safe Haskell 提供的是嚴格型別安全基礎，不直接等於完整 Untrusted-code sandbox。

本文因此將它定位為效果與純度的極端逃生口，而不是一般 I/O 便利函式。

SECTION

E.4 TypeScript any、unknown 與不健全性

已重新核對 TypeScript Handbook、Type Compatibility、strictNullChecks 與 unknown 發行文件：

- TypeScript 官方明確承認部分不健全行為是為了 JavaScript 生態與常見用法而有意保留；
- any 會對相關值大幅停用型別檢查；
- unknown 可以接收任何值，但使用前需 Narrow 或 Assertion；
- Type assertion 主要向 Type checker 提供信任，不會自動驗證 Runtime 資料；
- strictNullChecks=false 時，null 與 undefined 的風險會被型別系統大幅忽略；

- 官方建議在不知道值型別而仍需檢查時，優先考慮 unknown 而非 any。

  本文沒有將 TypeScript 寫成完全 Sound，也沒有把漸進型別描述成毫無安全價值。

SECTION

E.5 C# Stable 與 Preview 邊界

已重新核對 2026 年 Microsoft Learn：

- 現行既有模型以 unsafe Context 允許 Pointer、Function pointer、未受管理 Memory 等不可驗證操作；
- Unsafe code 不一定已經危險，而是安全性無法由 .NET 工具完整驗證；
- 官方最佳實務警告它可繞過 Memory safety 並造成 Memory corruption；
- C# 15／.NET 11 正在 Preview 更新後的 Memory safety model；
- Preview 模型嘗試將 unsafe 更靠近實際 Memory access，並使 Caller safety obligation 更清楚。

  本篇以既有穩定模型為正式案例，只把新版模型當成帶有日期的演化訊號。

SECTION

E.6 Go unsafe

已重新核對 Go Specification、Package unsafe 與 Go 1.4 Release Notes：

- unsafe 是 Compiler 知道的特殊 Built-in package；
- 它提供會違反一般 Type system 的低階能力；
- 使用 unsafe 的 Package 必須人工審查 Type safety；
- 可能不可攜；
- Go 官方已明確說明 Unsafe code 不保證持續符合 Go 1 Compatibility promise。

  本文因此不將 unsafe 只描述成語法便利，而視為同時放棄部分型別、表示、平台與演化保證的邊界。

SECTION

E.7 「安全語言」的用語限制

本文中的「安全子語言」只表示：

> 在列明的 Trusted Computing Base、威脅模型與語言前提下，對特定錯誤類別提供保證。
> 它不表示：

- 無邏輯錯誤；
- 無漏洞；
- 無資源耗盡；
- 必然終止；
- 外部依賴可信；
- 規格符合真實需求；
- Compiler、Runtime 與 Hardware 不會失效。

SECTION

E.8 逃生口不是自動缺陷

第二輪校對維持本文核心區分：

有逃生口
≠
安全設計失敗
逃生口非局部、無契約、不可審計、無法封裝
→
安全設計失敗風險上升
評價重點是安全包絡是否仍能成立，而不是語言是否完全沒有低階能力。