控制責任配置論：使用者、編譯器、Runtime 與工具誰應承擔錯誤？

The Allocation of Control Responsibility: Who Should Bear Errors—the Programmer, Compiler, Runtime, or Tools?

v1.0 / 2026-07-30 / Neo.K / 公開版／方法論基礎論文

https://thisoneisneok.com/html/papers/pldst-003.html

英文名稱： The Allocation of Control Responsibility: Who Should Bear Errors—the Programmer, Compiler, Runtime, or Tools?

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-003

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／方法論基礎論文

SECTION

摘要

程式語言設計中的錯誤問題，經常被簡化成一場二選一爭論：錯誤應在編譯期阻止，還是交給運行期與程式設計者處理？這種二分法忽略了錯誤治理其實至少包含七種責任：預防、偵測、定位、圍堵、處理與恢復、升級，以及追責與制度修正。

Rust 的所有權與借用檢查把部分記憶體安全責任前移至語言規則、編譯器與使用者的程式建模；Kotlin 的可空型別讓編譯器在多數情況下阻止對可能為 null 的值進行不安全操作；Python 的 PEP 484 刻意把型別提示主要交給外部靜態分析工具，而不在運行期自動強制；Go 使用普通值表示可預期錯誤，要求呼叫端明確處理，同時保留 panic／recover 作為另一層異常展開機制；Erlang／OTP 透過隔離程序與 supervision tree，將部分故障恢復責任提升至監督架構；SPARK 則以契約、流程分析與形式證明，把一部分運行期錯誤轉化為

本文提出 控制責任配置論（Control-Responsibility Allocation Theory, CRAT-PL），將程式錯誤治理表示為：

> cal-R
> =
> (P,D,L,C,H,E,A)

其中：

- P：Prevention，預防；
- D：Detection，偵測；
- L：Localization，定位；
- C：Containment，圍堵；
- H：Handling／Recovery，處理與恢復；
- E：Escalation，升級；
- A：Accountability／Adaptation，追責與制度修正。

  核心命題為：

> 【Responsibility(a)
> ≤
> Authority(a)+Observability(a)+Competence(a)】

若一個主體被要求承擔錯誤，卻沒有足夠控制權、資訊或處理能力，系統就不是「給予自由」，而是在製造不可履行責任。

同時：

> 【最早處理
> ≠
> 永遠最好】

更精確的原則是：

> 錯誤應由最早具備足夠語義資訊、實際控制能力與可接受誤判成本的層級處理；若不能預防，就必須明確安排偵測、隔離、恢復、升級與追責。

關鍵詞： 程式語言設計、錯誤處理、責任配置、編譯器、Runtime、靜態分析、容錯、Design by Contract、所有權、監督樹、PLDST

SECTION

一、「誰應承擔錯誤」不是單一問題

當程式出現問題時，至少可能問：

- 誰應阻止它被寫出？
- 誰應在提交前發現？
- 誰應在編譯時拒絕？
- 誰應在運行時檢查？
- 誰應選擇恢復策略？
- 誰應向使用者解釋？
- 誰應保存事故證據？
- 誰應修改未來規則？

如果將這些責任混成一個「錯誤處理」欄位，就容易產生錯誤比較：

- 靜態型別語言被說成「處理所有錯誤」；
- 動態語言被說成「不處理錯誤」；
- Runtime 恢復被誤認為預防；
- Compiler warning 被誤認為保證；
- 測試工具發現問題被誤認為語言安全；

- 使用者可以捕捉例外，被誤認為使用者必然有能力正確恢復。

SECTION

二、Fault、Error 與 Failure

Avizienis、Laprie、Randell 與 Landwehr 的可靠性分類區分 fault、error 與 failure：

- Fault：可能導致錯誤狀態的原因；
- Error：系統內部狀態偏離正確狀態；
- Failure：對外提供的服務偏離正確服務。[R1]

本文採用這組區分，但加入可處理責任：

Fault source
→ invalid or dangerous state
→ observable service failure
→ containment / recovery / adaptation

例如：

忘記處理可能缺值
→ 取得 null 狀態
→ 解參考時崩潰
→ 捕捉、重啟、補償或修正型別模型

語言機制可能只處理其中一段，不應宣稱處理完整鏈條。

SECTION

三、錯誤並不都屬同一類

3.1 程式建模錯誤

- 型別不匹配；
- 非完整模式匹配；
- 不合法生命週期；
- 無效狀態；

- 違反不變量。

3.2 環境錯誤

- 檔案不存在；
- 權限不足；
- 網路超時；
- 磁碟滿；
- 外部服務失敗。

3.3 資源錯誤

- 記憶體不足；
- 配額耗盡；
- 過深遞迴；
- 執行超時。

3.4 並行與分散式錯誤

- 競態；
- 死鎖；
- 訊息遺失；
- 部分失敗；
- 節點分區；
- 重複執行。

3.5 安全與信任錯誤

- 未授權輸入；
- 能力越界；
- 依賴遭竄改；
- 不可信資料；
- 惡意程式。

3.6 規格錯誤

- 程式完全按照錯誤需求執行；
- 正確實作了不正確政策；
- 型別正確但業務含義錯誤。

不同錯誤需要不同層級承擔。編譯器通常無法判斷明天的網路是否中斷，也無法只靠型別知道商業政策是否正確。

SECTION

四、預防 Prevention

預防的目標是讓危險狀態無法形成，或使其形成成本提高。

例：

- 非可空型別；
- 所有權與借用；
- Capability；
- 代數資料型別；
- 私有建構子；
- 單位型別；

- 效果系統；
- 語法限制；
- 安全預設。

預防可以由語言設計、型別系統、編譯器、API 與組織政策共同完成。

SECTION

五、偵測 Detection

偵測回答：問題在何時被觀察？

編輯
靜態分析
編譯
連結
測試
部署
運行
監控
事故後

偵測越早通常越便宜，但前提是該層具有足夠資訊，且誤判成本可接受。

SECTION

六、定位 Localization

知道「出錯」不等於知道「哪裡錯」。定位包括：

- 精確來源位置；
- 呼叫鏈；
- 資料流；
- 型別推導路徑；
- 違反的契約；
- 借用衝突；

- 導致錯誤的遠端事件；
- 影響範圍。

語言安全若沒有良好診斷，可能只是把運行期事故換成難以理解的編譯器拒絕。

SECTION

七、圍堵 Containment

圍堵阻止錯誤擴散，例如：

- 程序隔離；
- Actor mailbox；
- Transaction；
- Region；
- Capability boundary；
- Sandbox；
- Memory safety；
- Typed channel；
- Circuit breaker。

圍堵不一定修復原因，但可以限制故障半徑。

SECTION

八、處理與恢復 Handling／Recovery

恢復可能包括：

- 回傳錯誤值；
- 重試；

- 使用替代資料；
- 回滾；
- 重啟；
- 切換副本；
- 降級服務；
- 補償交易；
- 通知人工。

不是所有呼叫端都具有正確恢復資訊。有時低層只知道「讀取失敗」，高層才知道應該重試、改用快取或終止工作。

SECTION

九、升級 Escalation

若當前層無法安全處理，必須：

- 傳回錯誤；
- 拋出例外；
- 取消任務；
- 讓程序終止；
- 通知 supervisor；
- 觸發治理；
- 呼叫人工。

不處理但明確升級，通常比吞掉錯誤更安全。

SECTION

十、追責與適應 Accountability／Adaptation

事故後需要：

- 日誌；
- Trace；
- Dump；
- 證書；
- 版本；
- 變更原因；
- 修復；
- 新測試；
- 新型別；
- 新政策；
- 新監控。

這一層負責讓同類問題未來更難再次發生。

SECTION

十一、語言設計者

語言設計者決定：

- 哪些狀態可表示；
- 哪些錯誤被視為語言問題；
- 哪些責任能交給編譯器；
- 是否提供恢復構造；
- 是否保留逃生口；

- 是否要求明確標註。

  設計者不直接處理每次事故，但決定其他主體可使用的控制工具。

SECTION

十二、程式設計者

負責：

- 領域規格；
- 選擇錯誤策略；
- 處理可預期環境失敗；
- 建立不變量；
- 選擇是否使用逃生口；
- 對未被語言自動處理的風險負責。

  但程式設計者不應被要求手動重複處理機器可可靠自動化的工作。

SECTION

十三、編譯器

編譯器適合負責：

- 可由程式文本與靜態語義判定的規則；
- 型別；
- 名稱；
- 所有權；
- 可證明的不變量；
- 資源與效果約束的一部分；

- 產生診斷與修正建議。

  編譯器不應把「無法證明」靜默改寫成「一定錯誤」，也不應以不可理解診斷把責任重新推回使用者。

SECTION

十四、Runtime

Runtime 擁有：

- 實際值；
- 資源狀態；
- 排程；
- 動態型別；
- 網路與 I/O 結果；
- 程序隔離；
- 回收與重啟機制。

  因此適合負責動態檢查、記憶體回收、stack unwinding、監控、隔離與調度。但自動恢復必須避免無限重試、重複副作用、隱藏資料損壞與將不一致狀態重新投入服務。

SECTION

十五、程式庫與框架

程式庫最理解領域邊界，可負責：

- 輸入驗證；
- API 契約；
- 錯誤分類；
- 重試語義；

- 資源清理；
- Idempotency；
- 業務補償。

SECTION

十六、工具

工具包括 IDE、Linter、Formatter、Static analyzer、Type checker、Debugger、Profiler、Model checker、Proof assistant 與 CI。

工具輸出必須區分：

```
error
warning
advice
proof
unknown
unavailable
```

SECTION

十七、Operator 與組織

負責：

- 部署；
- 監控；
- 容錯拓撲；
- 服務降級；
- 回滾；
- 事故應變；
- 權限；
- 備份；

- 政策；
- 責任分工。

語言無法單獨保證一個部署系統可靠，因為系統還包含配置、資料、網路、憑證與人員流程。

SECTION

十八、責任矩陣

令責任集合為：

> K={P,D,L,C,H,E,A}

令主體集合為：

> A=
> {
> designer,
> programmer,
> compiler,
> runtime,
> library,
> tool,
> operator,
> organization
> }

責任矩陣：

> M=[m_(ak)]

其中：

- m_(ak)=0：不負責；
- m_(ak)=1：輔助責任；

- m_(ak)=2：主要責任；
- m_(ak)=3：最終責任。

  同一責任可以有多個參與者，但必須有明確主要負責者。

SECTION

十九、控制權

> Auth(a)
> =
> (prevent,reject,modify,contain,recover,rollback,escalate)

若程式設計者被要求保證某個 Runtime 不暫停，卻無法控制 GC 策略，責任與控制權便不匹配。

SECTION

二十、可觀察性

> Obs(a)
> =
> (state,cause,history,scope,cost,outcome)

一個主體若看不到實際錯誤、原因、影響與先前操作，就難以正確處理。

SECTION

二十一、能力

能力不只指權限，也包括技術與語義知識：

> Comp(a)
> =
> (domain,mechanism,recovery,risk)

低層 Runtime 可能知道記憶體分配失敗，卻不知道商業交易能否重試；高層服務知道重試語義，卻不知道底層記憶體是否已損壞。

SECTION

二十二、平衡條件

> 【Resp(a,k)
> ≤
> Auth(a,k)+Obs(a,k)+Comp(a,k)】

若不成立，應採取至少一項：

- 降低責任；
- 增加控制權；
- 增加可觀察性；
- 增加工具與知識；
- 將問題升級至其他層。

SECTION

二十三、最早可知時間

對錯誤 e，定義：

> t_(know)(e,l)

表示層級 l 最早能具有足夠資訊判斷問題的時間。

例如：

- 未宣告名稱：編輯或編譯期可知；
- 檔案不存在：運行期才知；

- 交易可否重試：可能要到應用層才知；
- 分散式部分失敗：監控與協議層才知；
- 錯誤需求：可能事故後才知。

SECTION

二十四、最早可處理不等於最早應拒絕

靜態分析可能無法證明安全，但不代表程式一定錯誤。應區分：

definitely invalid
not proven safe
possibly unsafe
policy forbidden
unsupported

若把所有「無法證明」都當錯誤，會產生誤判與表達力損失。

SECTION

二十五、延遲成本

錯誤越晚發現，通常修復成本越高，但早期強制也可能產生大量標註、假陽性、原型速度下降與過早固定錯誤模型。

真正目標是：

```
min
(
DetectionCost
+
FalsePositiveCost
+
FailureCost
+
RecoveryCost
)
```

SECTION

二十六、編譯期預防型

代表機制包括 Rust ownership、Kotlin nullability、模式完整性、效果型別與 SPARK flow analysis。

配置：

預防：語言／編譯器
偵測：編譯器
定位：編譯器診斷
恢復：使用者修改程式
升級：拒絕建置

優勢：

- 錯誤在部署前被阻止；
- 故障半徑低；
- 不需依賴特定執行路徑觸發。

代價：

- 規則與編譯器複雜；
- 錯誤訊息壓力；
- 可能拒絕安全但無法證明的程式。

SECTION

二十七、顯式錯誤值型

Go 以 error 值表示異常狀態，讓普通控制流與語言的值、介面和組合能力參與錯誤處理。[R4]

配置：

偵測：函式／程式庫
傳播：顯式 return
處理：呼叫端
升級：逐層 return 或 panic

優勢：控制流可見，呼叫端知道哪些操作可能失敗。代價是重複檢查、忽略錯誤、包裝不足，以及低層呼叫端可能不知道正確恢復策略。

SECTION

二十八、例外與展開型

Java checked exceptions、一般例外系統與 Go panic／recover 都屬不同形式的展開機制。

配置：

偵測：拋出點
傳播：Runtime
定位：Stack trace
處理：動態 Handler

Java 的 checked exception 要求未捕捉的受檢例外出現在 throws 宣告中，將一部分傳播責任納入編譯器契約。[R8]

優勢是成功與錯誤路徑分離，可跨多層傳播。代價包括非局部控制流、過度捕捉、空 catch、介面傳染，以及例外型別不一定對應正確恢復語義。

SECTION

二十九、監督與重啟型

Erlang／OTP supervisor 負責啟動、停止、監控子程序，並依 child specification 在失敗後重啟。[R5]

配置：

預防：有限
偵測：Runtime／Supervisor
圍堵：Process isolation
恢復：Restart strategy
升級：Supervision hierarchy

這種設計接受部分錯誤無法事先列舉，將恢復邏輯集中於監督結構。代價是重啟不等於狀態正確，還必須處理 restart intensity、狀態重建與外部副作用。

「讓程序崩潰」不能脫離程序隔離與 supervision tree 被理解成忽略錯誤。

SECTION

三十、契約與證明型

Design by Contract 使用前置條件、後置條件與類別不變量分配 Client 與 Supplier 的義務。SPARK 進一步使用 flow analysis 與 proof obligations，使 GNATprove 能證明特定範圍內不存在運行期錯誤，或指出未能證明的檢查。[R6][R7]

配置：

規格：設計者／程式設計者
驗證：工具／Prover
證明義務：程式設計者
拒絕：Build／Certification process
證明成功是相對於程式、契約、假設與模型成立，不代表真實需求本身正確。

SECTION

三十一、工具輔助漸進型

Python PEP 484 提供型別標註的標準語法，主要供第三方工具進行靜態分析、重構與其他用途；標註本身預設不在 Runtime 執行型別檢查。[R3]

配置：

語言：保存 annotation
工具：選擇性檢查
Runtime：維持原動態語義
組織：決定是否強制
優勢是漸進採用、不破壞既有動態程式與無預設運行期檢查。代價是未執行 checker 時沒有保證，不同工具可能有差異，CI 與組織政策成為責任鏈的一部分。

SECTION

三十二、Null：把缺值狀態放進一般參考型別

Tony Hoare 在 2009 年回顧 null reference 時，將其稱為「billion-dollar mistake」，並把問題連結到錯誤、漏洞與系統崩潰。[R9]

傳統責任配置常是：

語言：允許 reference 為 null
程式設計者：記得檢查
Runtime：未檢查時拋出錯誤
工具：盡量分析
問題在於 null 可能在遠處產生、跨多層傳播，最後在與原因無關的位置失敗。

Kotlin 透過可空與非可空型別，把多數責任改為：

語言：區分 T 與 T?
編譯器：追蹤檢查與 smart cast
程式設計者：顯式處理 nullable branch
Runtime：處理逃生口與互操作邊界
Kotlin 官方仍列出可能產生 NPE 的情況，包括 !!、初始化不一致及 Java 互操作，說明靜態責任配置不是絕對封閉。[R2]

SECTION

三十三、Rust：安全責任前移，但保留 unsafe

Rust 所有權規則由編譯器檢查，以支援無垃圾回收器的記憶體安全。[R10]

責任被分配為：

- 語言：所有權、移動、借用規則；
- 編譯器：檢查生命週期與別名；
- 使用者：建立符合規則的資料結構；
- Library author：在必要處封裝 unsafe；
- Reviewer／Tool：審查安全邊界。

  unsafe 表示某些系統能力無法由安全子集完整表達，但逃生口必須縮小、封裝與可審查。責任不是消失，而是集中於少量高風險邊界。

SECTION

三十四、Go：錯誤值與 panic 的雙層模型

Go 一般以 error 值表示可預期異常。官方文件強調，錯誤作為值可以使用普通語言機制處理與組合。[R4]

另一方面，panic 會觸發 stack unwinding 並執行 deferred functions；recover 只能在適當的 deferred function 中攔截。[R11]

可以理解為：

- error：局部、預期、可由呼叫端處理的失敗；
- panic：當前一般控制流不適合繼續；
- recover：特定邊界的轉譯與圍堵。

  良好設計不是每層都 recover，而是在具有足夠語義的邊界，將 panic 轉成可理解錯誤或終止。

SECTION

三十五、Erlang／OTP：錯誤恢復不是每個 Worker 的責任

Supervisor 將重啟決策從每個 worker 集中到監督結構。[R5]

這降低 worker 的局部恢復複雜度，但系統設計者仍要處理：

- one-for-one；
- one-for-all；
- rest-for-one；
- restart intensity；
- child restart type；
- 狀態重建；
- 外部副作用。

  這是責任攤銷與制度化，而不是錯誤消失。

SECTION

三十六、SPARK：證明工具不能替代規格責任

GNATprove 可以證明一組明確的運行期檢查不會被違反，例如 overflow、range、index、division、discriminant 與 length check。[R6]

但工具只能證明相對於程式、契約、假設與模型的性質：

> Proof(Code|= Spec)
> not⇒
> Spec|= RealNeed

如果契約漏掉真正業務要求，證明成功也不代表系統目的正確。

SECTION

三十七、Python Typing：把部分保證交給工具與組織流程

PEP 484 明確讓型別提示服務於靜態分析，並不讓 Python Runtime 預設執行型別檢查。[R3]

所以：

只寫 annotation
≠
已執行 type checking
真正的組織保證需要：

- 選擇 checker；
- 設定規則；
- 在 CI 執行；
- 阻擋未通過變更；
- 管理 ignore；
- 處理第三方 stub。

  這是典型的工具—組織共同承擔模式。

SECTION

三十八、最早具備足夠語義資訊者處理

> l^*
> =
> argmin_l Cost(l)

subject to：

> Info(l)≥ threshold

> Authority(l)≥ required

> FalsePositive(l)≤ acceptable

SECTION

三十九、可自動化的機械責任不應重複推給使用者

若機器能可靠、低誤判地檢查名稱、型別、所有權、非完整模式、契約或 API misuse，就不應要求每位使用者只靠紀律記住。

SECTION

四十、恢復必須由知道業務語義的層決定

低層程式庫不應任意無限重試、吞掉錯誤、更換資料或重複交易。它應保存足夠原因與 context，讓更高層決策。

SECTION

四十一、無法處理就明確升級

若當前層無法恢復，應保留原因並 return、throw、panic、cancel、terminate 或通知 supervisor，而不是空 catch 或回傳無解釋預設值。

SECTION

四十二、預防、偵測與恢復不能互相冒充

- 型別檢查是預防／偵測，不是恢復；
- Supervisor 是圍堵／恢復，不是證明；
- 測試是偵測，不是保證未測路徑；
- Proof 是相對規格的證明，不是需求正確性；
- Retry 是恢復策略，不是根因修復。

SECTION

四十三、逃生口必須附帶責任升級

例如：

- Rust unsafe；
- Kotlin !!；
- Type checker ignore；
- unchecked cast；
- reflection；
- raw pointer；
- dynamic code execution。

若：

> StaticGuaranteedownarrow

則必須：

> Review+Test+Containment+Adduparrow

SECTION

四十四、錯誤處理本身也會出錯

恢復程式可能重複寫入、造成資料損壞、無限重試、隱藏故障或引發級聯。所以恢復策略本身也需要型別、測試、Idempotency、資源預算、Circuit breaker、監控與演練。

SECTION

四十五、預防主義設計者

傾向強型別、所有權、效果、不合法狀態排除與編譯期拒絕。保護生產環境與長期維護者，代價是初期摩擦、編譯器複雜與表達限制。

SECTION

四十六、顯式責任設計者

傾向錯誤值、明確控制流與少量隱式展開。保護成本透明與呼叫端自主，代價是重複與呼叫端負擔。

SECTION

四十七、運行期韌性設計者

傾向 Actor、Supervisor、隔離、重啟與動態監控。保護長期可用性與部分失敗，代價是狀態恢復、副作用一致性與運維複雜度。

SECTION

四十八、契約證明設計者

傾向前置條件、後置條件、不變量、Proof obligation 與 Certification。保護高保證領域和可審查證據，代價是規格與證明成本。

SECTION

四十九、工具漸進設計者

傾向 Optional typing、Lint、IDE、Static analysis 與漸進採用。保護既有生態與遷移彈性，代價是保證依賴工具與 CI，配置容易碎片化。

SECTION

五十、組織治理設計者

傾向 Coding policy、CI gate、Review、Incident process 與 Deprecation。保護多人協作與長期演化，代價是流程成本與決策延遲。

SECTION

五十一、責任卡

設計機制：
錯誤類型：
Fault source：
Error state：
External failure：
預防者：
偵測者：
定位者：
圍堵者：
恢復者：
升級路徑：
追責與修正者：
支付時間：
控制權：
可觀察性：
能力：
逃生口：
失敗外溢：
反例：
證據：
信心：

SECTION

五十二、責任失衡

無權責任

要求某主體保證結果，卻不給控制能力。

無知責任

要求低層恢復，但低層不知道業務語義。

無證責任

工具聲稱安全，卻未保存規則與證據。

無主責任

每層都假設其他層處理。

重複責任

每層都重試、包裝或記錄，造成放大。

SECTION

五十三、設計比較問題

比較兩位設計者時，應問：

- 他主要預防什麼？
- 他允許什麼延到 Runtime？
- 他信任使用者到什麼程度？
- 他願意增加多少編譯器負擔？

- 他如何安排恢復？
- 他是否提供逃生口？
- 逃生口由誰審查？
- 事故後如何回饋到語言或制度？

SECTION

五十四、輸入

SKILL 接受設計者、語言、錯誤處理機制、規格、RFC／PEP、官方教學、原始訪談與程式案例。

SECTION

五十五、分析管線

重新網路搜尋
→ 錯誤機制抽取
→ Fault/Error/Failure 分層
→ 七項責任標記
→ 主體矩陣
→ 時機分析
→ 權責平衡
→ 逃生口分析
→ 反例搜尋
→ 事實校對
→ 風格判定

SECTION

五十六、輸出 JSON 雛形

```
{
  "mechanism": "nullable types",
  "responsibilities": {
    "prevention": ["language", "compiler"],
    "detection": ["compiler", "runtime boundary"],
    "localization": ["compiler diagnostics"],
    "containment": ["type boundary"],
    "recovery": ["programmer"],
```

```
  "escalation": ["compile error", "runtime exception"],
  "adaptation": ["language and tooling"]
 },
 "balance": {
  "authority": "high",
  "observability": "medium",
  "competence_required": "medium"
 },
 "escape_hatches": ["!!", "interop"],
 "inference_confidence": "high"
}
```


SECTION

五十七、SKILL 禁止事項

不得：

- 將編譯期檢查寫成完整正確性；
- 將 Runtime 重啟寫成根因修復；
- 將 annotation 寫成 Runtime 保證；
- 將 warning 寫成 error；
- 將未證明寫成已證明錯誤；
- 將「讓它崩潰」脫離 Supervisor 與隔離語境；
- 將錯誤值寫成必然被正確處理；
- 將個人風格與後期制度混為一談。

SECTION

五十八、錯誤分類具有情境性

同一事件在不同系統中可能是可恢復錯誤、不變量破壞、安全事故或正常控制流。例如「找不到資料」可能是正常空結果，也可能是資料遺失。

SECTION

五十九、靜態與動態不是單一直線

現代語言通常同時具有靜態檢查、Runtime check、Tool warning、Contract、Test 與 Operator policy，不能只以「靜態／動態」描述全部責任。

SECTION

六十、責任不能完全自動量化

權限、可見性、能力與風險可結構化，但其權重依領域、團隊、錯誤代價、規模與時間而變。數值只適合作比較索引。

SECTION

六十一、原始資料可能重構歷史

設計者回顧、官方教學與後期制度文件可能反映不同時期。PLDST 必須記錄日期，避免把後期成熟原則直接回寫成創始動機。

程式語言設計中的錯誤責任，不能只回答：讓編譯器抓，還是讓程式設計者自己處理？

真正完整的問題是：

- 誰防止錯誤條件形成？
- 誰最早能可靠偵測？
- 誰能解釋原因？
- 誰能限制故障半徑？
- 誰知道如何恢復？
- 誰有權回滾或終止？
- 誰保存證據並修正制度？

本文將責任表示為：

> cal-R
> =
> (P,D,L,C,H,E,A)

並提出平衡條件：

> 【Resp(a,k)
> ≤
> Auth(a,k)+Obs(a,k)+Comp(a,k)】

成熟設計應建立一條責任鏈：

> 【能預防者預防
> →
> 能偵測者偵測
> →
> 能定位者解釋
> →
> 能圍堵者隔離
> →
> 懂業務者恢復
> →
> 無法處理者升級
> →
> 制度保存證據並修正】

設計師風格因此可以從責任分配中被辨識：有些設計者將責任前移至型別與編譯器；有些交給顯式呼叫端；有些接受不可預見錯誤，依靠 Runtime 隔離與監督恢復；有些要求契約與證明；有些把檢查放入可選工具與組織流程；有些保留危險逃生口，但要求邊界集中與審查。

PLDST 對每位設計者都應追問：

他保護誰免於承擔錯誤？他把哪些責任交給機器、哪些留給使用者、哪些移到 Runtime、工具或組織？當責任轉移後，相應的控制權、可觀察性與處理能力是否也一起被轉移？

這比單純問「語言安全嗎」更能揭示程式語言設計的深層人格。

代號	責任	核心問題
P	Prevention	如何防止危險狀態形成？
D	Detection	何時知道出錯？
L	Localization	能否找到原因與範圍？
C	Containment	如何阻止擴散？
H	Handling／Recovery	如何繼續、補償或回滾？
E	Escalation	無法處理時交給誰？
A	Accountability／Adaptation	如何保存證據並修正制度？

[R1] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing,” IEEE Transactions on Dependable and Secure Computing, 1(1), 2004, pp. 11–33.

— Fault、Error、Failure 與 dependability means 的系統分類。

[R2] JetBrains, “Null Safety,” Kotlin Documentation.

— Kotlin 可空／非可空型別、編譯期檢查與仍可能產生 NPE 的邊界。

[R3] Guido van Rossum et al., “PEP 484 – Type Hints,” Python Enhancement Proposals, 2014–2015.

— 型別提示服務靜態分析；預設不在 Runtime 執行型別檢查。

[R4] Andrew Gerrand, “Error handling and Go,” Go Blog, 2011; Rob Pike, “Errors are values,” Go Blog, 2015; Go Team, “Defer, Panic, and Recover,” 2010.

— Go error value、顯式處理、panic 與 recover。

[R5] Erlang/OTP Documentation, “Supervisor Behaviour — Supervision Principles.”

— Supervisor 啟動、停止、監控與重啟 child process。

[R6] AdaCore, SPARK User’s Guide, “Absence of Run-Time Errors” and “Applying SPARK in Practice.”

— GNATprove、運行期檢查、Flow analysis 與證明層級。

[R7] Bertrand Meyer, “Applying ‘Design by Contract’,” Computer, 25(10), 1992, pp. 40–51; Eiffel documentation, “Building Bug-Free O-O Software.”

— 前置條件、後置條件、不變量與 Client／Supplier 義務。

[R8] Oracle, “The Catch or Specify Requirement” and “Unchecked Exceptions — The Controversy,” Java Tutorials.

— Checked exceptions、throws 與 RuntimeException 的責任差異。該教學以 JDK 8 時期內容為主，本文只用於機制與歷史說明。

[R9] Tony Hoare, “Null References: The Billion Dollar Mistake,” QCon London, 2009.

— Null reference 的歷史回顧與後果評價。

[R10] Steve Klabnik and Carol Nichols, The Rust Programming Language, “Understanding Ownership,” official Rust documentation.

— 所有權規則、編譯器檢查與無 GC 記憶體安全。

[R11] Go Team, The Go Programming Language Specification, sections on defer, panic and recover.

— Stack unwinding、deferred calls 與 recover 的正式語義。

[F] 可確認史實或正式語義
[Q] 設計者／官方文件的明確表述
[D] 可辨識的設計決策
[I] 本文責任配置推論
[C] 反例、限制或逃生口
[U] 證據不足

本篇初稿完成後，已再次以原始論文、官方規格或官方文件核對下列項目：

- Fault／Error／Failure 的術語邊界

  本文採 Avizienis、Laprie、Randell 與 Landwehr 的 dependability taxonomy：fault 是可能的原因，error 是系統內部狀態，failure 是對外服務偏離。本文增加的七項責任模型是 PLDST 方法推論，不是該論文原有分類。

- Rust 所有權的責任配置

  Rust 官方書籍將 ownership 描述為由編譯器檢查的記憶體管理規則；違反規則時程式不會通過編譯。本文據此分析責任前移，但沒有把 ownership 寫成所有正確性或所有記憶體問題的完整證明。

- Kotlin Null Safety 的保證範圍

  Kotlin 官方文件確實將多數潛在 null 問題移至編譯期，但同時列出 !!、初始化不一致與 Java 互操作等可能產生 NPE 的情況。因此本文使用「多數情況」而非「完全消除 NPE」。

- Python Type Hints 不等於 Runtime Enforcement

  PEP 484 明確指出，annotation 雖可在 Runtime 取得，但預設不發生型別檢查；它假設存在可由使用者選擇執行的離線 checker。本文因此把保證責任配置到工具、CI 與組織流程，而不是 Python Runtime。

- Go 的雙層錯誤語義

  Go 官方文章使用 error value 作為一般錯誤處理機制；語言規格則規定 panic、defer 與 recover 的 stack unwinding 行為。本文把 error 與 panic 分成不同責任層，是方法論解讀，不代表官方將所有錯誤正式分為這兩類。

- Erlang Supervision 不等於忽略錯誤

  Erlang／OTP 官方文件的可確認內容是：supervisor 啟動、停止、監控 child process，並在需要時重啟。本文沒有把社群口號「let it crash」當成無條件設計規則，也明確保留狀態重建、restart intensity 與外部副作用責任。

- SPARK Proof 的範圍

  GNATprove 的 proof mode 可驗證運行期錯誤與 Pre／Post 等 assertion；官方文件也明確列出 AoRTE 的範圍與限制，例如 Storage_Error 不在一般 AoRTE 統計範圍內。本文因此只寫「特定範圍」與「相對於契約及假設」，未宣稱 SPARK 證明所有可能的真實系統失敗。

- Design by Contract 的歸因

  前置條件、後置條件與不變量的 Client／Supplier 義務，依 Bertrand Meyer 的正式文章與 Eiffel 文件。本文的七項責任映射則是新的 PLDST 分析，不是 Meyer 原文中的七層模型。

- Java Checked Exceptions 的資料年代

  Oracle Java Tutorials 的例外教學主要是 JDK 8 時期材料。本文只將其用於 checked／unchecked 與 catch-or-specify 機制的歷史說明，不以它代表 2026 年全部 Java 平台設計狀態。

- Tony Hoare 的 Null 回顧

  「billion-dollar mistake」來自 Hoare 2009 年 QCon London 的公開演講與活動摘要。本文把它作為歷史反省材料，未將所有 null 相關問題或後續語言設計全部歸因於 Hoare 個人。

- 公式的理論地位

  Resp≤ Auth+Obs+Comp 是責任配置的啟發式約束，不是可直接測量的物理定律。它的作用是檢查某主體是否被賦予沒有控制權、可觀察性或知識支撐的責任。