
前言

「規則比較類似指導方針，而不是硬性規定。」

——Hector Barbosa

在百花齊放的現代程式語言領域中，Rust 顯得與眾不同。Rust 結合了編譯型語言的速度、無資源回收機制（non-garbage-collected）語言的效率，以及泛函（functional）語言的型態安全，並提供一種獨特的方式來解決記憶體安全問題。因此，Rust 經常被評為「最受喜愛的程式語言」（<https://oreil.ly/KKcb6>）。

Rust 型態系統的強大與一致性意味著，如果一個 Rust 程式可以成功編譯，它就應該可以正常運作，以前，這種現象只會在學術性更高、普及性較低的語言中出現，如 Haskell。一旦 Rust 程式能夠成功編譯，它也會是安全的。

然而，這種安全性（包括型態安全和記憶體安全）是有代價的。儘管 Rust 的基本文件有很高的品質，但它依然有著「入門門檻偏高」的評價。新手往往需要經歷一系列的「啟蒙儀式」，例如與借用檢查器（borrow checker）奮戰、重新設計資料結構，以及理解生命週期。雖然成功地編譯出來的 Rust 程式應該第一次就可以正確運行，但是成功編譯它的過程並不輕鬆，即使 Rust 編譯器提供了非常有用的錯誤診斷工具。

本書適用對象

本書試圖幫助程式設計師從他們的困境中脫困，即使他們已經具備使用 C++ 等編譯型語言的經驗。因此，與其他的《高效的〈某種語言〉》書籍一樣，本書應該是 Rust 初學者的第二本書，適合在他們已經從其他教材（例如《*The Rust Programming Language*》（Steve Klabnik 和 Carol Nichols 著，No Starch Press 出版）或《*Programming Rust*》（Jim Blandy 等著，O'Reilly 出版））學會基本知識之後閱讀。

然而，由於 Rust 的安全性，本書介紹的招式（Items）所探討的角度與 Scott Meyers 原創的《Effective C++》稍微不同。C++ 語言充滿各種「陷阱」（footguns），因此《Effective C++》主要圍繞一系列避免這些陷阱的建議，這些建議是基於「開發 C++ 軟體」的實際經驗。重要的是，這些建議是指導方針而非硬性規則，因為指導方針有例外。我們將提供指導方針背後的理由，讓讀者自行判斷對他們所面臨的情況而言，這些規範是否依然適用。

本書保留了「提供建議及其理由」的整體風格。然而，由於 Rust 幾乎沒有「陷阱」，書中的招式集中於 Rust 引入的概念。許多招式的標題有「瞭解…」和「熟悉…」等字眼，它們的目的是幫助讀者寫出流暢且符合 Rust 習慣的程式碼。

Rust 的安全性也導致本書沒有標題裡有「絕對不要（Never）…」的招式。如果有事情「絕對不要做」，編譯器通常會阻止你那樣做。

Rust 版本

本書基於 2018 版的 Rust 和穩定的（stable）工具鏈。Rust 的回溯相容性承諾（<https://oreil.ly/husN4>）意味著任何後續的 Rust 版本，也括 2021 版，仍然支援針對 2018 版而設計的程式碼，即使後來的版本引入破壞性變更（breaking change）。目前 Rust 已經足夠穩定了，所以 2018 版與 2021 版的差異很小；本書的程式碼不需要修改即可與 2021 版相容（但招式 19 有一個例外情況，說明 Rust 的後續版本允許之前不支援的新行為）。

本書的招式未涵蓋 Rust 的 async 功能的任何層面（<https://oreil.ly/a9r1B>），因為它涉及更進階的概念，以及相對不穩定的工具鏈支援——光是同步 Rust 就有很多要談的了。以後或許還會出一本《Effective Async Rust》…

書中的程式片段和錯誤訊息使用的是 rustc 1.70。這些程式片段在後續版本中應該不需要更改，但錯誤訊息可能會隨著編譯器的版本而有所不同。本書中的錯誤訊息可能經過編輯，以配合書本版面的寬度限制，除此之外皆與編譯器的輸出相同。

本書多次參考並比較其他靜態型態語言，例如 Java、Go 和 C++，以幫助具有這些語言背景的讀者快速上手（C++ 應該是最接近的對應語言，尤其是在涉及 C++11 的移動語義（move semantic）時）。

本書結構

本書包含六章：

第 1 章，「型態」

圍繞 Rust 的核心型態系統之建議

第 2 章，「*trait*」

關於 Rust *trait* 的建議

第 3 章，「概念」

Rust 的主要設計概念

第 4 章，「依賴項目」

關於使用 Rust 套件生態系統的建議

第 5 章，「工具」

除了 Rust 編譯器之外，改善碼庫的建議

第 6 章：「在標準 Rust 之外」

當你需要在 Rust 的標準、安全的環境之外工作時的建議

儘管「概念」一章比「型態」和「*trait*」等章更基本，但本書故意將它放在後面，為從頭開始依序閱讀到最後的讀者建立一些信心。

本書編排慣例

本書使用下列的編排方式：

斜體字 (*Italic*)

代表新術語、URL、email 地址、檔名，與副檔名。中文以楷體表示。

定寬字 (`Constant width`)

在長程式中，或是在文章中，代表變數、函式名稱、資料庫、資料型態、環境變數、表達式、關鍵字…等程式元素。

無法編譯

```
// 無法編譯的範例程式
```

不理想的行為

```
// 產生不理想行為的範例程式
```

致謝

感謝以下的貴人協助完成本書：

- 感謝所有技術校閱，他們對本書的所有層面提供了專業且詳盡的回饋：Pietro Albinì、Jess Males、Mike Capp，特別是 Carol Nichols。
- 感謝 O'Reilly 的編輯們：Jeff Bleiel、Brian Guerin 與 Katie Tozer。
- 感謝 Tiziano Santoro 教我許多 Rust 知識。
- 感謝 Danny Elfandbaum 提供了重要的技術協助來處理本書的 AsciiDoc 格式化。
- 感謝本書原始網路版的認真讀者，特別是：
 - Julian Rosse，發現了網路版本中的數十個拼寫和其他錯誤。
 - Martin Disch，指出了幾個招式可能需要改進和不準確之處。
 - Chris Fleetwood、Sergey Kaunov、Clifford Matthews、Remo Senekowitsch、Kirill Zaborsky，以及一位匿名 Proton Mail 用戶，他們指出了文本中的錯誤。
- 感謝我的家人忍受了我在許多週末把心思放在寫作上。

型態

本書的第 1 章提供關於 Rust 型態系統的建議。Rust 的型態系統比其他主流語言的型態系統更具表達力，較類似 OCaml 和 Haskell 之類的「學術性」語言的型態系統。

Rust 型態系統的核心之一是它的 `enum` 型態，其表達能力遠超過其他語言中的列舉（`enumeration`）型態，且支援代數資料型態。

本章的招式將介紹這個語言提供的基本型態，以及如何使用它們來組成能夠精確地表達程式語義的資料結構。這種「將行為植入型態系統」的概念有助於減少專門用來檢查程式碼與處理錯誤的程式碼，因為無效的狀態在編譯階段就會被工具鏈拒絕，而不是在執行階段才被程式拒絕。

本章也將介紹 Rust 標準程式庫的常見資料結構：`Option`、`Result`、`Error` 和 `Iterator`。熟悉這些標準工具可幫助你寫出符合 Rust 慣例、且既高效又精簡的程式碼——尤其是它們支援 Rust 的問號運算子，提供一種既低調又型態安全的錯誤處理手段。

要注意的是，涉及 Rust *trait* 的招式會在下一章提供，但由於 *trait* 是為了描述型態的行為而設計的，所以本章的招式難免與 *trait* 有一定程度的重疊。

招式 1：使用型態系統來表達你的資料結構

到底是誰將他們稱為程式設計師（*programmer*），
而不是型態寫手（*type writer*）的？

—— @thingskatedid（<https://oreil.ly/hHj5c>）

本招式將快速介紹 Rust 的型態系統，從編譯器提供的基本型態開始，延伸至「將數值組成資料結構」的各種方式。

Rust 的 `enum` 型態扮演重要的角色。雖然 `enum` 的基本版本類似其他語言的 `enum` 型態，但它的變體可以結合資料欄位，提供更大的彈性與表達能力。

基本型態

熟悉其他靜態定型程式語言（例如 C++、Go 或 Java）的人應該對 Rust 型態系統的基本概念瞭若指掌。Rust 提供一組具有特定大小的整數型態，包括 `signed`（`i8`、`i16`、`i32`、`i64`、`i128`）和 `unsigned`（`u8`、`u16`、`u32`、`u64`、`u128`）整數型態。

此外，還有「與目標系統的指標大小相符」的 `signed`（`isize`）和 `unsigned`（`usize`）整數型態。然而，在 Rust 中，你不會經常在指標與整數之間轉換，因此大小是否相符不太重要。不過，標準集合使用 `usize` 來回傳它們的大小（以 `.len()`），所以在集合的檢索操作中，`usize` 型態相當常見——這完全不會有容量方面的問題，因為在記憶體內，集合項目數量不可能超過系統的記憶體位址數量。

從整數型態也可以看出 Rust 是比 C++ 更嚴謹的語言。在 Rust 中，將較大的整數型態（例如 `i32`）存入較小的整數型態（例如 `i16`）會造成編譯錯誤：

無法編譯

```
let x: i32 = 42;
let y: i16 = x;
```

```
error[E0308]: mismatched types
--> src/main.rs:18:18
   |
18 |     let y: i16 = x;
   |               --- ^ expected `i16`, found `i32`
   |               |
   |               expected due to this
```

```

|
help: you can convert an `i32` to an `i16` and panic if the converted value
      doesn't fit
|
18 |     let y: i16 = x.try_into().unwrap();
|                               ++++++

```

Rust 不會默默地容忍程式設計師做出危險的操作，這一點令人安心。儘管這個範例使用那些值來做型態轉換沒有任何問題，但編譯器必須考慮可能導致轉換無法正常執行的情況：

無法編譯

```

let x: i32 = 66_000;
let y: i16 = x; // 這個值會是什麼？

```

從錯誤訊息也可以看到，雖然 Rust 有比較嚴格的規則，但它也提供有用的編譯器訊息，來指示如何遵守那些規則。訊息中的解決方案帶來一個問題：在轉換過程中，如果需要調整數值的話，該怎麼處理？稍後會討論錯誤處理（招式 4）以及 **panic!**（招式 18）。

Rust 也不允許一些看似「安全」的操作，例如將較小整數型態值存入較大的整數型態：

無法編譯

```

let x = 42i32; // 帶有型態後綴的整數常值
let y: i64 = x;

```

```

error[E0308]: mismatched types
--> src/main.rs:36:18
|
36 |     let y: i64 = x;
|         ^ expected `i64`, found `i32`
|
|         expected due to this
|
help: you can convert an `i32` to an `i64`
|
36 |     let y: i64 = x.into();
|                   ++++++

```

編譯器建議的解決方案不會引發錯誤處理方面的問題，但轉換仍然必須明確地進行。稍後會詳細討論型態轉換（招式 5）。

Rust 的參考

參考是 Rust 中最普遍的類指標型態，其型態寫成 `&T`，其中 `T` 是某種型態。雖然它的底層是指標值，但編譯器會確保關於其用法的各種規則都被遵守：它必須始終指向一個有效且正確對齊的 `T` 型態實例，該實例的生命週期（招式 14）必須超出使用它的範圍，並且必須符合借用檢查規則（招式 15）。Rust 的參考一詞暗示了這些額外的限制，因此在 Rust 中，比較不會單純使用指標這個術語。

C++ 的參考型態也有 Rust 的「參考必須指向有效且正確對齊的項目」這一條限制，但 C++ 沒有生命週期的概念，因此會有「懸空參考（dangling references）」的隱患：¹⁹

不理想的行為

```
// C++
const int& dangle() {
    int x = 32; // 位於堆疊，稍後會被覆蓋
    return x; // 回傳指向堆疊變數的參考！
}
```

Rust 的借用和生命週期檢查機制意味著等效的程式碼無法成功編譯：

無法編譯

```
fn dangle() -> &'static i64 {
    let x: i64 = 32; // 位於堆疊
    &x
}
```

```
error[E0515]: cannot return reference to local variable `x`
--> src/main.rs:477:5
   |
477 |     &x
   |     ^^ returns a reference to data owned by the current function
```

Rust 的參考 `&T` 可以用來對底層的項目進行唯讀操作（大致相當於 C++ 的 `const T&`）。若要修改底層項目，你就要使用可變參考 `&mut T`，它同樣受制於招式 15 所討論的借用檢查規則。這種命名模式反映了 Rust 和 C++ 之間的思維差異：

¹⁹ 但現代編譯器會發出警告。

- Rust 的預設做法是唯讀，可寫的型態必須特別標註（使用 `mut`）。
- C++ 的預設做法是可寫的，唯讀型態必須特別標註（使用 `const`）。

編譯器會將使用參考的 Rust 程式碼轉換成使用簡單指標的機器碼，在 64-bit 平台上，這些指標的大小為 8 bytes（本招式皆假設如此）。例如，一對區域變數以及指向它們的參考：

```
pub struct Point {
    pub x: u32,
    pub y: u32,
}

let pt = Point { x: 1, y: 2 };
let x = 0u64;
let ref_x = &x;
let ref_pt = &pt;
```

在堆疊（stack）上的最終布局可能像圖 1-2 這樣。

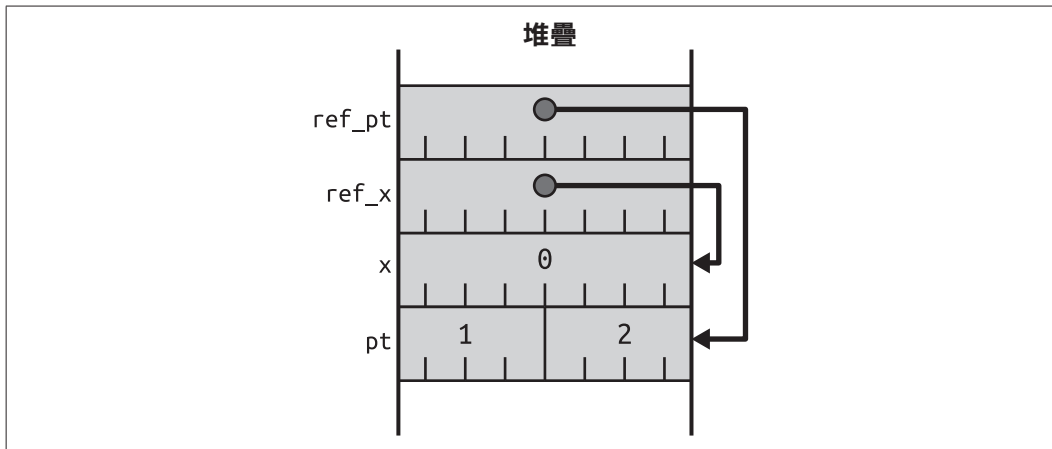


圖 1-2 堆疊布局，包含指向區域變數的指標

Rust 的參考可以指向堆疊或 heap 裡的項目。在預設情況下，Rust 會將項目放到堆疊裡，但使用 `Box<T>` 指標型態（大致相當於 C++ 的 `std::unique_ptr<T>`）會將項目強制放到 heap，這意味著它配置的項目可能超出當下程式碼區塊的作用域繼續存在。在底層，`Box<T>` 也是一個簡單的 8-byte 指標值：

```
let box_pt = Box::new(Point { x: 10, y: 20 });
```

如圖 1-3 所示。

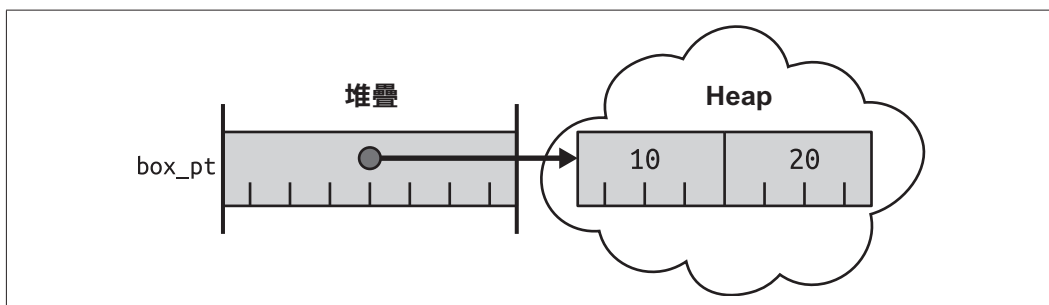


圖 1-3 堆疊 Box 指標指向 heap 中的 struct

指標 trait

接受 `&Point` 之類的參考引數的方法也可以接收 `&Box<Point>`：

```
fn show(pt: &Point) {  
    println!("{}", pt.x, pt.y);  
}  
show(ref_pt);  
show(&box_pt);  
  
(1, 2)  
(10, 20)
```

這是因為 `Box<T>` 實作了 `Deref trait`，且 `Target = T`。為某種型態實作這種 trait，意味著你可以用 trait 的 `deref()` 方法來建立一個指向 `Target` 型態的參考。Rust 還有一個對應的 `DerefMut trait`，它可產生一個指向 `Target` 型態的可變參考。

`Deref/DerefMut trait` 比較特別，因為 Rust 編譯器在處理實作了這些 trait 的型態時有特定的行為。當編譯器遇到解參考表達式（例如 `*x`）時，它會尋找並使用其中一個 trait 的實作，具體取決於解參考是否需要可變存取（mutable access）。這種 *Deref* 強制轉型讓各種智慧指標型態的行為就像普通參考一般，這是在 Rust 中，允許隱性型態轉換的少數幾個機制之一（詳見招式 5）。

作為技術面補充資訊，`Deref trait` 的目標型態不能是泛型的（`Deref<Target>`），這件事情值得探究。如果可以是泛型的，那麼某種型態 `ConfusedPtr` 就可以同時實作 `Deref<TypeA>` 和 `Deref<TypeB>`，導致編譯器無法為 `*x` 之類的運算式推斷出唯一的型態。因此，Rust 將目標型態寫成名為 `Target` 的相關型態。

概念

本書的前兩章介紹了 Rust 的型態和 `trait`，說明 Rust 程式的一些概念所使用的詞彙，那些概念將是本章討論的主題。

借用檢查器（`borrow checker`）與生命週期檢查（`lifetime checks`）是讓 Rust 如此獨特的元素，然而，它們也往往是 Rust 新手的絆腳石，因此，本章的前兩個招式將重點探討它們。

本章的其他招式則涵蓋一些相對容易理解、但與其他程式語言的寫法仍有一些差異的概念，具體包括：

- 關於 Rust 的 `unsafe` 模式及如何避免使用它的建議（招式 16）
- 關於撰寫 Rust 多執行緒程式的好消息與壞消息（招式 17）
- 關於避免執行期中止的建議（招式 18）
- 關於 Rust 如何處理反射（`reflection`）的資訊（招式 19）
- 關於在「最佳化」與「易維護性」之間取得平衡的建議（招式 20）

最好可以讓你的程式碼與這些概念的影響保持一致。雖然在 Rust 中確實可以重現 C/C++ 的行為（在某種程度上），但是要這樣做的話，何必使用 Rust ？

招式 14：瞭解生命週期

本招式介紹 Rust 的生命週期，生命週期以更精確的方式描述以前的編譯型語言（例如 C 和 C++）就具備的概念（即使理論上不一定如此，但在實務上確實存在）。生命週期是招式 15 介紹的借用檢查器的必要輸入。這些功能一起構成 Rust 的記憶體安全保證的核心。

堆疊簡介

生命週期與堆疊有根本關係，因此必須快速地介紹或回顧一下相關的概念。

當程式執行時，它使用的記憶體會被分成不同的區塊，有時也稱為區段（*segment*）。其中有些區塊的大小是固定的，例如用來存放程式碼或全域資料的區塊，但有兩個區塊會隨著程式的執行過程而改變大小，它們是 *heap* 與堆疊。為了實現大小的改變，這兩個區塊通常被配置在程式虛擬記憶體空間的相對兩端，一個向下增長，另一個向上增長（至少在你的程式耗盡記憶體並崩潰之前），如圖 3-1 所示。

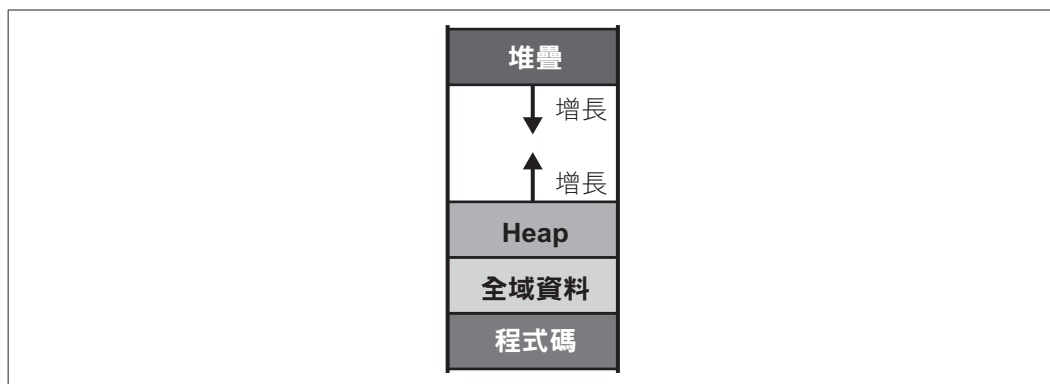


圖 3-1 程式記憶體布局，包含向上增長的 heap 與向下增長的堆疊

在這兩個動態改變大小的區塊裡面，堆疊保存與目前執行中的函式有關的狀態。這些狀態可能包括以下元素：

- 被傳給函式的參數
- 在函式中使用的區域變數
- 在函式中計算出來的暫時值
- 在呼叫函式的程式碼中，函式的返回位址

當函式 `f()` 被呼叫時，Rust 會將一個新的堆疊框（*stack frame*）加入堆疊中，其位置在呼叫端函式的堆疊框尾端之後，CPU 通常會更新一個暫存器（堆疊指標）以指向新的堆疊框。

當內部函式 `f()` `return` 時，堆疊指標會被設回去該函式被呼叫之前的位置，也就是呼叫端的堆疊框的位置，該框仍保持完整且未被修改。

如果呼叫端後來又呼叫另一個函式 `g()`，這個過程會再次發生，這意味著函式 `g()` 的堆疊框會重複使用 `f()` 用過的記憶體區域（如圖 3-2 所示）：

```
fn caller() -> u64 {  
    let x = 42u64;  
    let y = 19u64;  
    f(x) + g(y)  
}  
  
fn f(f_param: u64) -> u64 {  
    let two = 2u64;  
    f_param + two  
}  
  
fn g(g_param: u64) -> u64 {  
    let arr = [2u64, 3u64];  
    g_param + arr[1]  
}
```

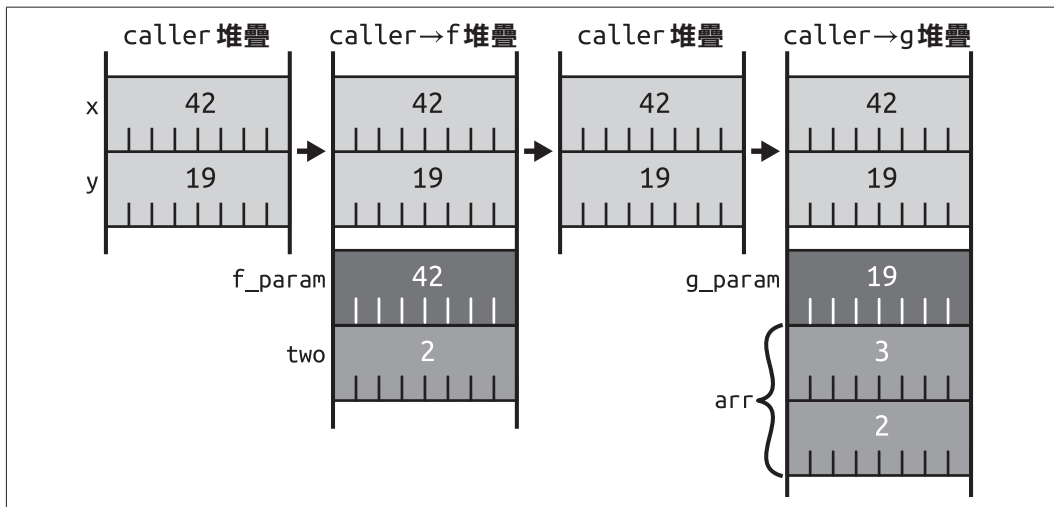


圖 3-2 當函式被呼叫，以及從函式 `return` 時，堆疊的演變

當然，以上的說明大幅簡化了實際的情況，把資料放入堆疊，以及從堆疊取出資料需要時間，因此實際的處理程序會做許多優化。然而，這些簡化的概念已經足以幫助你理解本招式的主題了。

生命週期的演變

上一節解釋了參數與區域變數如何被儲存在堆疊，並指出這些值只是暫時存在的。

歷史上，這種機制是一把兩面刃：如果你持有一個指向這些暫時存在的堆疊值的指標會怎樣？

從 C 開始，回傳一個指向區域變數的指標是完全 OK 的（但現代編譯器會發出警告）：

不理想的行為

```
/* C 程式碼 */
struct File {
    int fd;
};

struct File* open_bugged() {
    struct File f = { open("README.md", O_RDONLY) };
    return &f; /* 回傳堆疊物件的位址！ */
}
```

如果你運氣不好，呼叫端程式碼立刻使用回傳值，也許你會僥倖沒事：

不理想的行為

```
struct File* f = open_bugged();
printf("in caller: file at %p has fd=%d\n", f, f->fd);
```

```
in caller: file at 0x7ff7bc019408 has fd=3
```

說你運氣不好，是因為它僅僅看似正確。一旦程式呼叫其他函式，堆疊區域將被重複使用，原本保存物件的記憶體會被覆寫：

不理想的行為

```
investigate_file(f);
```

```
/* C 程式碼 */
void investigate_file(struct File* f) {
    long array[4] = {1, 2, 3, 4}; // 將資料放入堆疊
    printf("in function: file at %p has fd=%d\n", f, f->fd);
}
```

```
in function: file at 0x7ff7bc019408 has fd=1592262883
```

在這個例子中，遺失物件的內容會造成額外的負面影響：被開啟的檔案的檔案描述符會遺失，導致程式流失本來被保存在資料結構裡的資源。

後來，C++ 藉著引入解構式來解決這種與資源失聯的問題，實現了 RAII（招式 11）。現在，堆疊裡的物件有自行清理資源的能力：如果那些物件持有某種類型的資源，解構式可以清理資源，而 C++ 編譯器可保證在清理堆疊框時，堆疊裡的物件的解構式一定會被呼叫：

```
// C++ 程式碼
File::~File() {
    std::cout << "~File(): close fd " << fd << "\n";
    close(fd);
    fd = -1;
}
```

現在，呼叫端得到一個指向已被銷毀且資源已被回收的物件的（無效）指標：

不理想的行為

```
File* f = open_bugged();
printf("in caller: file at %p has fd=%d\n", f, f->fd);
```

```
~File(): close fd 3
in caller: file at 0x7ff7b6a7c438 has fd=-1
```

然而，C++ 並未解決懸空指標的問題，你仍然可能持有一個指向已銷毀物件（解構式已經被呼叫）的指標：

```
// C++ 程式碼
void investigate_file(File* f) {
    long array[4] = {1, 2, 3, 4}; // 將資料放入堆疊
    std::cout << "in function: file at " << f << " has fd=" << f->fd << "\n";
}
in function: file at 0x7ff7b6a7c438 has fd=-183042004
```

C/C++ 程式設計師必須自行注意這種情況，避免將一個指向已消失物件的指標解參考。然而，如果你是攻擊者，並且發現這樣的懸空指標，你可能會眉開眼笑地解參考該指標，開始進行攻擊。

接下來是 Rust 時代，它吸引人的核心特點之一，在於它徹底解決了懸空指標的問題，從而瞬間解決了大量的安全性問題。¹

1 例如，Chromium 專案估計 70% 的安全性 bug 是由記憶體安全問題引起的（<https://oreil.ly/GJkt0>）。