從零開始學 Rust

Andy Shiue

生成日期：2026 年 8 月 3 日

從零開始學 Rust

你好！這篇教學的主要目的是讓完全沒寫過程式的初學者理解 Rust 這個程式語言中的各種概念。雖然網路上已經有很多 Rust 的教學了，但以前的教學似乎都是寫給已經會至少另一個程式語言的學習者，因此我希望這份教學能填補這樣的空白。Rust 這個語言有著很鮮明的特色，Rust 的功能強大，用 Rust 寫的程式執行效率也高，並且使用 Rust 的時候，還更容易在寫程式初期提早發現錯誤。大家都聽過的 C++ 雖然同樣也是功能強大且執行效率高，但在安全性上頭卻做了不少犧牲。因為 Rust 如上所述的特性，Rust 也在這個以 AI 寫程式的時代佔有舉足輕重的地位。

這份教學的思路也因此比較符合目前 AI 時代的開發流程，教學中並不會提供各種「作業」，認為你一定要能寫出特定規格的算法。取而代之的是，這份教學會讓你有辦法讀懂 Rust 的程式大概在做什麼，也希望能讓你理解一個真正軟體的架構是怎麼被設計出來的。甚至我會說，如果你懶得使用電腦的話，單單閱讀本教學而不實際撰寫程式來學習 Rust 也是一種可行的方式。

儘管如此，我還是建議學習者一章一章閱讀本教學，如果你有靜態語言的基礎，你或許可以跳過第 1 章，但我更推薦的方法還是簡單花幾分鐘掃過第 1 章之後再往後繼續閱讀。如果是完全的初學者就更不用說了。當然，要是你不怕漏掉什麼內容的話，你也可以直接跳到有興趣的部分看，或者是使用搜尋功能提前閱讀教學後面才給的解釋，這些都是可行的做法。啊對了，文中有一章「附錄一」，雖然叫作附錄但也建議讀者能全部觀看。

如果讀者在閱讀的過程中遇到看不懂的段落、想要練習題，或寫的程式跑不起來，而想問 AI 怎麼辦，我提供了一個壓縮檔給 AI 閱讀，讓 AI 能更精準回應你的需求。使用的時候只要把整份壓縮檔上傳給 AI，和他說「請先閱讀壓縮檔裡的 `GUIDE.md` 再回答我」，並且告訴 AI 你目前讀到第幾章第幾集再提出你的需求就好了。壓縮檔的下載連結如下：

> `rust-book-src.zip`：https://andyshiue.github.io/learning-rust-from-zero/zh-TW/rust-book-src.zip

強烈建議要用好一點的 AI 模型讀壓縮檔！免費的 AI 甚至可能完全懶得讀壓縮檔內的內容。

有點遲鈍的是，我後來才意識到很多讀者可能會直接把整部教學丟給 AI 閱讀而非親自閱讀。如果你還認真讀前言讀到了這邊，又或者是你的 AI 讀到了這部分的內容，我對這樣讀者的想法是：

1. 我的確曾經想過有人會用這樣的閱讀方式，但並沒有想到有那麼多人會採取。的確那也是閱讀我教學的其中一個方法。
2. 儘管如此，教學本文的存在是有意義的，否則我就不會撰寫。只在遇到不懂的程式碼時問 AI 可能會造成讀者只有能力閱讀程式，卻無法寫出程式，但我也承認這些人也是我本來就設定好的客群。
3. 教學本文的意義在於你能一步步建立對 Rust 各個功能的全面理解。
4. 就算你使用 AI 幫助閱讀，那也不代表需要的時候你不能回來閱讀教學本文。
5. 其實你並不一定要去 GitHub 對我的專案進行 `git clone`；上面的 `.zip` 檔已經包含了本教學的全部內容（你可以手動解壓縮看看）。之所以會建議使用壓縮檔而非直接閱讀網頁是因為壓縮檔中的內容是純文字，對於 AI 會更友善。
6. 我後來才發現，現代「完全的初學者」可能並沒有那麼多，因此會想跳過第 1 章。上面有說這或許可行，但如果你發現自己好像沒搞懂最基礎的概念，第 1 章的本文都非常短，閱讀一下真的不需要花多少時間。

7. 請特別注意非同步一章的前半。那部分的精髓就在於練習讀懂程式碼，就算你用 AI 閱讀也一樣。在此特別推薦一書：Asynchronous Programming in Rust。本教學的非同步一章是參考該書後自己重寫出來的。

另外本教學也有 PDF 版供讀者下載：

> PDF 版：https://andyshiue.github.io/learning-rust-from-zero/zh-TW/book.pdf

但我可能不會長期維護 PDF 版，因此還是比較建議閱讀有互動功能的網頁版教學。如果你現在就正在閱讀 PDF 版的話，以下是網頁版的網址：

> 網頁版：https://andyshiue.github.io/learning-rust-from-zero/zh-TW/

最後提一下上面說的互動功能，不然怕沒人知道：你可以直接在網頁版教學裡面跑程式。文內程式原始碼的右上角有幾個按鈕，按了就知道會發生什麼事了。大概先這樣吧……

本教學除大綱外，初稿由 AI 完成，並經人類修改：

- 模型：Claude 4.5 ~ 5 / GPT-5.5 ~ 5.6
- 馬具：OpenClaw / Claude Code / ChatGPT (Codex)

目錄

從零開始學 Rust i

第一部 1

1 基礎 2

1.1 安裝 Rust . . . 2
1.2 第一個程式 . . . 3
1.3 變數與輸出 . . . 4
1.4 註解 . . . 5
1.5 算術運算子 . . . 7
1.6 運算子優先順序 . . . 8
1.7 比較運算子 . . . 9
1.8 `if` . . . 10
1.9 作用域 . . . 11
1.10 `else` . . . 12
1.11 `else if` . . . 13
1.12 邏輯運算子 . . . 14
1.13 `let mut` . . . 15
1.14 複合賦值運算子 . . . 16
1.15 `stdin` . . . 18
1.16 `parse` . . . 19
1.17 綜合練習 . . . 20
1.18 `loop` + `break` . . . 21
1.19 `while` . . . 23
1.20 `for` + range . . . 24
1.21 巢狀迴圈 . . . 25
1.22 `continue` . . . 27
1.23 型別（基礎） . . . 28
1.24 型別（數字詳解） . . . 29
1.25 `char` . . . 31
1.26 跳脫字元 . . . 33
1.27 `if` 當表達式 . . . 35

2 函數、陣列與切片 37

2.1 `const` . . . 37
2.2 shadowing . . . 38
2.3 底線變數 . . . 40

2.4 tuple . . . 42
2.5 {:?} Debug 格式 . . . 44
2.6 簡單函數 . . . 45
2.7 函數參數 . . . 47
2.8 函數回傳值 . . . 48
2.9 early return . . . 50
2.10 遞迴 . . . 53
2.11 陣列基礎 . . . 54
2.12 陣列走訪 . . . 56
2.13 切片 &[T] . . . 58
2.14 切片作為參數 . . . 60
2.15 字串切片 &str . . . 62

3 Struct、Enum 與 Pattern Matching 65

3.1 struct（named fields） . . . 65
3.2 tuple struct 與 unit struct . . . 66
3.3 enum（C-style） . . . 68
3.4 match C-style enum . . . 69
3.5 match 當表達式 . . . 70
3.6 block 表達式 . . . 72
3.7 enum 攜帶 tuple variant . . . 73
3.8 enum 攜帶 struct variant . . . 75
3.9 match 解構 tuple variant . . . 76
3.10 match 解構 struct variant . . . 77
3.11 field shorthand . . . 79
3.12 tuple pattern . . . 80
3.13 slice pattern . . . 82
3.14 巢狀 pattern matching . . . 83
3.15 _ wildcard . . . 85
3.16 .. 忽略多個值 . . . 86
3.17 range pattern . . . 89
3.18 多個值 | . . . 91
3.19 @ 綁定 . . . 93
3.20 match guard . . . 94
3.21 let 解構 tuple . . . 97
3.22 let 解構 struct . . . 99
3.23 for 迴圈解構 . . . 100
3.24 函數參數解構 . . . 102
3.25 if let . . . 104
3.26 while let . . . 106
3.27 let else . . . 108
3.28 associated function . . . 110
3.29 method . . . 111

3.30 大寫 Self . . . 114

4 所有權與借用 117

4.1 所有權（鑰匙圈比喻） . . . 117
4.2 trait 簡介 . . . 118
4.3 move 與 Clone . . . 120
4.4 Copy . . . 122
4.5 借用 & . . . 125
4.6 可變借用 &mut . . . 129
4.7 借用規則 . . . 131
4.8 self vs &self vs &mut self . . . 134
4.9 stack 與 heap . . . 137
4.10 String . . . 139
4.11 String vs &str . . . 141
4.12 Vec 基礎 . . . 143
4.13 Vec 與所有權 . . . 145

5 泛型、Trait Bound 與生命週期 149

5.1 泛型函數 . . . 149
5.2 泛型 struct . . . 150
5.3 泛型 enum . . . 152
5.4 turbofish 語法 . . . 153
5.5 placeholder type _ . . . 155
5.6 型別別名 . . . 156
5.7 泛型 impl . . . 157
5.8 Option<T> . . . 159
5.9 Option 常用方法 . . . 161
5.10 Result<T, E> . . . 163
5.11 ? 運算子 . . . 165
5.12 多個方法的 trait 與預設實作 . . . 167
5.13 trait bound . . . 169
5.14 use 基礎 . . . 171
5.15 Display trait . . . 172
5.16 多個 trait bound 與 where . . . 174
5.17 impl Trait 語法 . . . 176
5.18 多參數 trait . . . 178
5.19 From<T> / Into<T> . . . 180
5.20 Drop . . . 182
5.21 Box<T> . . . 185
5.22 Rc<T> . . . 187
5.23 Deref . . . 190
5.24 Cell<T> . . . 195
5.25 RefCell<T> . . . 197
5.26 生命週期基礎 . . . 199

5.27 生命週期省略規則 . . . 202
5.28 型別上的生命週期 . . . 204
5.29 lifetime bound . . . 207
5.30 supertrait . . . 209
5.31 常見的 `derive` `trait` . . . 211
5.32 associated type . . . 214
5.33 `Cow<'a, B>` . . . 218

6 閉包與迭代器 221

6.1 函數指標 . . . 221
6.2 閉包用法展示 . . . 223
6.3 手動實作閉包 . . . 226
6.4 閉包種類的推斷 . . . 231
6.5 `FnOnce` / `FnMut` / `Fn` . . . 232
6.6 `move` 閉包 . . . 235
6.7 `Option` / `Result` 的閉包方法 . . . 238
6.8 `Iterator` `trait` . . . 242
6.9 `for` 迴圈的真面目 . . . 245
6.10 `into_iter` / `iter_mut` / `iter` . . . 248
6.11 收集 . . . 251
6.12 聚合 . . . 252
6.13 組合與截取 . . . 255
6.14 轉換與過濾 . . . 258
6.15 惰性求值 . . . 260

7 Cargo、Crate 與 Mod 系統 265

7.1 Cargo 與 crates.io . . . 265
7.2 `mod` . . . 267
7.3 檔案 `mod` . . . 269
7.4 `pub` 可見性 . . . 272
7.5 `use` . . . 277
7.6 `cargo test` . . . 281
7.7 `pub use` . . . 284
7.8 orphan rule . . . 287
7.9 文件註解 . . . 289
7.10 `cargo publish` . . . 293

8 附錄一 298

8.1 數字字面值格式 . . . 298
8.2 && 和 || 的短路行為 . . . 300
8.3 `break` 回傳值 . . . 301
8.4 多行字串 & raw string literal . . . 303
8.5 格式化字串進階 . . . 305
8.6 `struct` / `enum` 放在 `fn` 裡面 . . . 308
8.7 `struct` update syntax . . . 310

8.8 ref pattern 與 match ergonomics . . . 312
8.9 panic! / todo! / unimplemented! / unreachable! . . . 314
8.10 let chains . . . 317
8.11 Rc 迴圈與 Weak . . . 319
8.12 fully qualified syntax . . . 323
8.13 DST 簡介 . . . 326
8.14 extern crate . . . 330

第二部 333

9 多執行緒 334

9.1 指標 . . . 334
9.2 thread::spawn . . . 336
9.3 Send / Sync . . . 339
9.4 RefCell 在多執行緒 . . . 341
9.5 Rc 在多執行緒 . . . 343
9.6 Arc<T> . . . 344
9.7 atomic 型別 . . . 346
9.8 Mutex<T> . . . 350
9.9 RwLock<T> . . . 352
9.10 poisoning . . . 355
9.11 mpsc . . . 358
9.12 死鎖 . . . 360
9.13 thread::scope 簡介 . . . 362

10 進階語言功能 365

10.1 dyn Trait 基礎 . . . 365
10.2 dyn compatibility . . . 370
10.3 const fn . . . 373
10.4 associated const . . . 375
10.5 const generics . . . 377
10.6 預設參數 . . . 380
10.7 運算子重載 . . . 381
10.8 型別轉換 as . . . 384
10.9 enum discriminant . . . 386
10.10 attribute 總覽 . . . 388
10.11 cfg! 巨集 . . . 391
10.12 macro_rules! . . . 393
10.13 proc macro . . . 396
10.14 unsafe . . . 398
10.15 static 變數 . . . 401
10.16 LazyLock . . . 403
10.17 extern blocks . . . 404
10.18 union . . . 406
10.19 never type ! . . . 408

11 進階標準庫 410
11.1 AsRef<T> / AsMut<T> . . . 410
11.2 Ordering 與排序 . . . 411
11.3 HashMap<K, V> . . . 414
11.4 HashSet<T> . . . 417
11.5 其他集合簡介 . . . 420
11.6 std::env / std::process . . . 422
11.7 std::path . . . 424
11.8 字串方法 . . . 426
11.9 I/O：stdin、檔案讀寫 . . . 428
11.10 Error trait . . . 431
11.11 thiserror / anyhow 簡介 . . . 434
11.12 catch_unwind . . . 436
12 非同步 439
12.1 第一個 async 程式 . . . 439
12.2 為什麼需要 async . . . 441
12.3 async fn 回傳的是 Future . . . 443
12.4 async 是一種作用 . . . 444
12.5 async block . . . 447
12.6 Future trait 與最陽春的 executor . . . 449
12.7 手寫一個 Delay Future . . . 452
12.8 用.await 等待 Delay . . . 454
12.9 手寫 join . . . 456
12.10 用 Thread 與 Waker 喚醒 executor . . . 460
12.11 spawn 與 ready queue . . . 464
12.12 spawn 與 JoinHandle . . . 468
12.13 mio . . . 474
12.14 手寫 reactor . . . 477
12.15 async fn 背後的狀態機 . . . 484
12.16 自我參照的 Future . . . 488
12.17 為什麼 poll 需要 Pin . . . 490
12.18 Unpin . . . 493
12.19 pin! . . . 496
12.20 async 遞迴 . . . 497
12.21 回到 Tokio . . . 499
12.22 spawn_blocking . . . 502
12.23 join! . . . 503
12.24 Semaphore 與 backpressure . . . 505
12.25 mpsc . . . 507
12.26 oneshot、watch 與 broadcast . . . 508
12.27 AsyncRead 與 AsyncWrite . . . 510
12.28 select! . . . 512

12.29 async 的 Mutex、RwLock 與 Notify 516
12.30 Stream 518
12.31 JoinSet 與 FuturesUnordered 520
12.32 graceful shutdown 523
12.33 測試 async 程式 527
12.34 Tokio 以外的 runtime 528

第一部

第 1 章

基礎

在這一章中，本教學會帶你理解大部分程式語言都有的基本程式控制流程。舉例來說，如果我們想要叫程式依據目前的天氣溫度進行決策，例如溫度超過 30 度時打開冷氣，那麼我們會需要進行條件的判斷：現在的溫度大於某數字嗎？如果答案是肯定的，則開啟冷氣；如果答案是否定的，則做其他事情。儘管本章還不會教你要怎麼真正把程式連接到冷氣上，但你至少會學到如何在語言內撰寫條件判斷。又或者，你可能會想控制 LED 燈過一秒就閃爍一次，在這種狀況下，我們需要的是叫電腦不斷進行重複的動作：燈亮及燈滅。同樣地，本章不會教你把程式連接到 LED 燈上，甚至也不會教你如何等待一秒，但你會學到如何叫電腦不斷進行重複的操作。

1.1 安裝 Rust

1.1.1 本集目標

把 Rust 裝到你的電腦上，確認它可以用。

1.1.2 正文

哈囉！歡迎來到 Rust 教學系列！

今天是第一集，我們什麼程式都還不寫，先把工具裝好就好。就像你要煮菜，總得先有鍋子對吧？

1.1.2.1 安裝 rustup

Rust 有一個官方的安裝工具叫 **rustup**，它會幫你把所有需要的東西一次裝好。

打開你的瀏覽器，去這個網址：

```
https://rustup.rs
```

- **Windows 使用者**：下載 `rustup-init.exe`，雙擊執行，然後一路按 enter 選預設就好。
- **Mac / Linux 使用者**：打開終端機（Terminal），貼上這行指令：

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

它會問你要不要用預設設定，直接按 enter 就好。

1.1.2.2 確認安裝成功

裝完之後，**重新打開**一個終端機（這很重要，舊的終端機可能還抓不到），然後輸入：

```
rustc --version
```

如果你看到類似這樣的東西：

```
rustc 1.XX.X (xxxxxxx 20XX-XX-XX)
```

恭喜你！Rust 已經裝好了！

`rustc` 就是 Rust 的編譯器，它負責把你寫的程式碼變成電腦能執行的東西。至於什麼是編譯器，之後我們慢慢聊，現在你只要知道「裝好了、能用了」就夠了。

1.1.3 重點整理

- 用 **rustup** 安裝 Rust，它會幫你把所有需要的工具一次裝好。
- 安裝完要重新開終端機，再用 `rustc --version` 確認安裝成功。
- `rustc` 是 Rust 的編譯器，負責把程式碼轉成電腦能執行的東西。

1.2 第一個程式

1.2.1 本集目標

用 Cargo 建立專案，跑出你人生中第一個 Rust 程式。

1.2.2 正文

上一集我們裝好了 Rust，今天就來寫第一個程式吧！

1.2.2.1 用 Cargo 建立專案

Rust 有一個超好用的工具叫 **Cargo**，它負責 Rust 專案的建置與相依管理。你可以把它想成一個管家，幫你整理程式碼、編譯、執行，全部包辦。

打開終端機，輸入：

```
cargo new hello
```

這會幫你建立一個叫 `hello` 的資料夾，裡面已經幫你準備好了基本的檔案結構。

1.2.2.2 用 VS Code 打開

接著用 VS Code（或你喜歡的編輯器）打開這個 `hello` 資料夾。你會看到兩個重要的東西：

1. **Cargo.toml** — 這是專案的設定檔，記錄你的專案叫什麼名字、用什麼版本之類的。現在不用管它，知道有這個檔案就好。
2. **src/main.rs** — 這就是你的程式碼！打開來看看：

```rust
fn main() {
    println!("Hello, world!");
}
```

這就是 Rust 自動幫你生成的第一個程式。`fn main()` 是程式的入口點，所有程式都從這裡開始跑。`println!` 是印東西到螢幕上的指令。我們在第 1 章裡面暫時都只會在 `fn main()` 後面接的大括號裡面寫程式。

1.2.2.3 什麼是編譯？

在跑程式之前，先來了解一個重要概念。

我們寫的 `.rs` 檔案是給人看的程式碼，電腦其實看不懂。所以需要一個翻譯的過程，把我們寫的程式碼變成電腦能直接執行的檔案——這個翻譯的過程就叫做**編譯**（compile）。

負責做這件事的工具叫做**編譯器**（compiler），Rust 的編譯器就是上一集安裝的 `rustc`。

好消息是，你不需要自己去呼叫 `rustc`，等一下用的 `cargo run` 會自動幫你編譯再執行，一步搞定。

1.2.2.4　執行看看

回到終端機，先進入 `hello` 資料夾：

```
cd hello
```

然後輸入：

```
cargo run
```

你應該會看到螢幕上印出：

```
Hello, world!
```

太棒了！你的第一個 Rust 程式成功跑起來了！

1.2.2.5　改一下再跑

現在回到你喜歡的文字編輯器，把 `println!` 裡面的文字改成：

```
fn main() {
    println!("Hello, Rust!");
}
```

存檔，再回到終端機跑一次 `cargo run`：

```
Hello, Rust!
```

看到了嗎？你改了什麼，它就印什麼。程式設計就是這麼回事——你告訴電腦要做什麼，它就照做。

1.2.3　重點整理

- **Cargo** 負責 Rust 專案的建置與相依管理，用 `cargo new` 建立新專案。
- 專案裡 `Cargo.toml` 是設定檔，`src/main.rs` 是主程式碼。
- **編譯**是把人看得懂的程式碼翻譯成電腦能跑的檔案。
- 用 `cargo run` 一鍵完成編譯和執行。
- `fn main()` 是程式的入口點，`println!` 用來印東西到螢幕上。

1.3　變數與輸出

1.3.1　本集目標

學會用 `let` 建立變數，再用 `println!` 把它印出來。

1.3.2　正文

上一集我們成功跑出了 “Hello, Rust!”，但那個文字是寫死在程式裡的。如果我們想要更靈活一點呢？這時候就需要**變數**了。

1.3.2.1　什麼是變數？

變數讓我們替一個值取名字，之後可以透過這個名字使用該值。

來看看怎麼用：

```rust
fn main() {
    let x = 5;
    println!("{}", x);
}
```

這裡的 `let x = 5;` 會把名稱 x 綁定到值 5。

然後 `println!("{}", x);` 裡面的 `{}` 就是一個佔位符，意思是「這個位置，請幫我填入 x 的值」。

1.3.2.2 文字變數

變數名稱不只能綁定數字，也能綁定文字：

```rust
fn main() {
    let name = "Rust";
    println!("Hello, {}!", name);
}
```

看到了嗎？`{}` 的位置被 name 的值 `"Rust"` 取代了。

你也可以試著把 `"Rust"` 改成你自己的名字，看看會印出什麼！

1.3.2.3 let 不一定要馬上賦值

let 宣告變數的時候，不一定要馬上給值。你可以先宣告，之後再賦值：

```rust
fn main() {
    let x;
    x = 5;
    println!("{}", x);
}
```

這樣完全合法，但使用前一定要賦值剛好一次，沒有賦值就使用會發生編譯錯誤。

1.3.3 重點整理

- let 用來建立變數。
- "雙引號" 包起來的是文字。
- println!("{}", 變數) 可以把變數的值印出來。
- {} 是佔位符，會被後面的值取代。
- let 宣告不一定要馬上賦值，但使用前一定要剛好賦值一次。

1.4 註解

1.4.1 本集目標

學會在程式碼裡寫筆記（註解），讓自己和別人看得懂你在幹嘛。

1.4.2 正文

寫程式的時候，有時候你會想在旁邊做個筆記，提醒自己「這段在幹嘛」。這就是**註解**的用途。

註解不會被電腦執行，它純粹是寫給人看的。

1.4.2.1 單行註解

用 // 開頭，後面的內容整行都是註解：

```
fn main() {
    // 這是一個註解，電腦會忽略這行
    let x = 5; // 也可以寫在程式碼後面
    println!("{}", x);
}
```

跑起來還是只會印出 5，那兩行註解完全不會影響程式。

1.4.2.2 多行註解

如果你要寫很長的筆記，可以用 /* */ 把它包起來：

```
fn main() {
    /*
        這是多行註解
        可以寫好幾行
        電腦通通會忽略
    */
    let x = 10;
    println!("{}", x);
}
```

1.4.2.3 什麼時候要寫註解？

- 當這段程式碼的邏輯不太明顯的時候。
- 當你怕自己過幾天回來看會忘記的時候。
- 當你想暫時讓某行程式碼不要執行（把它「註解掉」）。

```
fn main() {
    let x = 5;
    // println!("{}", x); // 暫時不印，但不想刪掉
    println!("程式結束");
}
```

這樣 println!("{}", x); 就不會被執行了，但你隨時可以把 // 拿掉讓它復活。

1.4.2.4 小提醒

不用每一行都寫註解喔！好的程式碼本身就應該夠清楚。註解是用在「不明顯」的地方，不是每行都要解釋。

1.4.3 重點整理

- // 是單行註解，/* */ 是多行註解。
- 註解是寫給人看的，電腦完全忽略。
- 可以用註解暫時「關掉」某行程式碼，不用刪掉它。
- 好的程式碼本身就該夠清楚，註解用在不明顯的地方就好。

1.5 算術運算子

1.5.1 本集目標

學會在 Rust 裡做加減乘除和取餘數。

1.5.2 正文

今天來學數學！別怕，就是加減乘除而已。

1.5.2.1 基本四則運算

先建立兩個變數：

```
fn main() {
    let a = 10;
    let b = 3;

    println!("{} + {} = {}", a, b, a + b); // 13
    println!("{} - {} = {}", a, b, a - b); // 7
    println!("{} * {} = {}", a, b, a * b); // 30
    println!("{} / {} = {}", a, b, a / b); // 3
    println!("{} % {} = {}", a, b, a % b); // 1
}
```

1.5.2.2 等等，10 / 3 怎麼是 3？

好問題！因為 a 和 b 都是整數，所以 Rust 做的是**整數除法**，小數點後面直接砍掉。10 除以 3 等於 3.333...，砍掉小數就是 3。

1.5.2.3 % 是什麼？

% 叫做**取餘數**（模數運算）。10 除以 3 等於 3 餘 1，所以 `10 % 3` 就是 1。

你可以想成：「10 裡面有幾個 3？有 3 個，然後剩下 1。」那個剩下的就是餘數。

1.5.2.4 多個 {} 的用法

你有注意到嗎？我們在 `println!` 裡面放了三個 `{}`：

```
fn main() {
    let a = 10;
    let b = 3;
    println!("{} + {} = {}", a, b, a + b);
}
```

Rust 會按照順序把值填進去：

- 第一個 `{}` → a 的值（10）。
- 第二個 `{}` → b 的值（3）。
- 第三個 `{}` → `a + b` 的值（13）。

幾個 `{}` 就對應後面幾個值，順序要對上。

1.5.3 重點整理

- 五個算術運算子：`+`（加）、`-`（減）、`*`（乘）、`/`（除）、`%`（取餘數）。

- 整數除法會直接捨去小數部分（`10 / 3` 是 3 不是 3.333）。
- `%` 取餘數：`10 % 3` 就是 10 除以 3 剩下的 1。
- `println!` 裡可以放多個 `{}`，按順序對應後面的值。

1.6 運算子優先順序

1.6.1 本集目標

了解 Rust 的運算順序——先乘除後加減，以及怎麼用括號改變順序。

1.6.2 正文

上一集我們學了加減乘除，但如果把它們混在一起呢？電腦會先算哪個？

1.6.2.1 先乘除，後加減

```rust
fn main() {
    println!("{}", 2 + 3 * 4);
}
```

你覺得答案是多少？

如果你覺得是 20（先算 2 + 3 = 5，再乘 4），那就錯了！

答案是 **14**。因為 Rust 跟數學一樣，**先乘除，後加減**。所以它先算 `3 * 4 = 12`，再算 `2 + 12 = 14`。

1.6.2.2 用括號改變順序

如果你真的想先算加法呢？加括號就對了：

```rust
fn main() {
    println!("{}", (2 + 3) * 4);
}
```

這次答案就是 **20** 了。括號裡面的會先算，`2 + 3 = 5`，然後 `5 * 4 = 20`。

1.6.2.3 小訣竅

不確定順序的時候，加括號就對了。括號不只是改順序，有時也讓程式碼更好讀。就算順序本來就對，加個括號讓意圖更明確也沒什麼不好。

```rust
fn main() {
    // 這兩行結果一樣，但第二行更清楚
    println!("{}", 2 + 3 * 4);
    println!("{}", 2 + (3 * 4));
}
```

1.6.3 重點整理

- Rust 的運算優先順序跟數學一樣：先乘除後加減。
- 用括號 `()` 可以強制改變運算順序。
- 運算的先後順序不清楚時，可以用括號讓意圖更明確。

1.7 比較運算子

1.7.1 本集目標

學會用比較運算子來比大小、判斷相不相等。

1.7.2 正文

到目前為止我們都在做數學運算，但程式設計裡還有另一種很重要的運算——**比較**。

比較的結果不是數字，而是 `true`（對）或 `false`（錯）。

1.7.2.1 == 等於

```
fn main() {
    println!("{}", 5 == 5);
}
```

5 等於 5 嗎？對，所以是 `true`。

注意喔，是**兩個等號** `==`，不是一個。一個等號 `=` 是拿來給變數賦值的（`let x = 5;`），兩個等號 `==` 才是拿來比較的。

1.7.2.2 != 不等於

```
fn main() {
    println!("{}", 5 != 3);
}
```

5 不等於 3 嗎？對。

1.7.2.3 < 小於

```
fn main() {
    println!("{}", 3 < 5);
}
```

3 小於 5。

1.7.2.4 > 大於

```
fn main() {
    println!("{}", 10 > 7);
}
```

10 大於 7。

1.7.2.5 <= 小於等於

```
fn main() {
    println!("{}", 5 <= 5);
}
```

5 小於或等於 5 嗎？等於的話也算。

1.7.2.6 >= 大於等於

```
fn main() {
    println!("{}", 8 >= 10);
}
```

8 大於或等於 10 嗎？不是。

1.7.2.7 一覽表

運算子	意思	範例	結果
==	等於	5 == 5	true
!=	不等於	5 != 3	true
<	小於	3 < 5	true
>	大於	10 > 7	true
<=	小於等於	5 <= 5	true
>=	大於等於	8 >= 10	false

1.7.3 重點整理

- 六個比較運算子：==、!=、<、>、<=、>=。
- 比較的結果是 true（對）或 false（錯）。
- ==（兩個等號）是比較，=（一個等號）是賦值，別搞混。

1.8 if

1.8.1 本集目標

用 if 讓程式根據條件決定要不要做某件事。

1.8.2 正文

到目前為止，我們的程式都是從頭到尾一行一行執行的。但真正的程式需要會「判斷」——如果怎樣，就做什麼事。

這就是 if 的用途。

1.8.2.1 基本用法

```
fn main() {
    let x = 7;
    if x > 3 {
        println!("大於 3");
    }
}
```

邏輯很簡單：x 是 7，7 大於 3 嗎？對，所以就執行大括號 {} 裡面的程式碼。

1.8.2.2 條件不成立的話呢？

把 x 改成 1 試試看：

```rust
fn main() {
    let x = 1;
    if x > 3 {
        println!("大於 3");
    }
}
```

跑起來……什麼都沒有。因為 1 不大於 3，條件是 `false`，所以大括號裡的程式碼就被跳過了。

1.8.3 重點整理

- `if` 後面接條件，條件為 `true` 就執行大括號裡的程式碼。
- 條件為 `false` 就整段跳過不執行。
- Rust 的 `if` 條件不需要加小括號。

1.9 作用域

1.9.1 本集目標

了解大括號 `{}` 創造的「範圍」，以及為什麼變數出了大括號就不能用了。

1.9.2 正文

這集來聊一個很重要的概念——**作用域**（scope）。

1.9.2.1 什麼是作用域？

你可以把大括號 `{}` 想成一個房間。在房間裡面建立的東西，出了房間就看不到了。

來看這個例子：

```rust
fn main() {
    {
        let y = 10;
        println!("{}", y);
    }
}
```

到這裡都沒問題。

1.9.2.2 出了大括號會怎樣？

現在試著在大括號外面用 y：

```rust
fn main() {
    {
        let y = 10;
        println!("{}", y);
    }
    println!("{}", y); // 這行會出錯！
}
```

你會得到一個編譯錯誤，Rust 在跟你說：「我找不到 y 這個東西。」

為什麼？因為 y 是在那對大括號裡面建立的，一出了大括號，y 就不見了。就像你在一個房間裡放了一張椅子，關上門之後，走廊上是看不到那張椅子的。

1.9.2.3 為什麼要有作用域？

這其實是一件好事。它讓你的變數不會在不該出現的地方亂跑。想像一下如果每個變數在程式的任何地方都能用，那程式一大起來就會超級混亂。作用域幫你把東西整理得有條有理。

1.9.2.4 不只是獨立的大括號

上一集教的 `if` 也有大括號對吧？其實 `if` 的大括號也是一個作用域，外面看不到裡面的變數。只要看到 `{}`，裡面常常就是一個作用域。這是 Rust 裡面一個很一致的規則。

1.9.3 重點整理

- 大括號 `{}` 裡面常常是一個作用域（scope）。
- 在作用域裡建立的變數，出了 `{}` 就消失、不能再用。
- `if` 會形成自己的作用域。

1.10 `else`

1.10.1 本集目標

用 `else` 讓程式在條件不成立時，做另一件事。

1.10.2 正文

上次學 `if` 的時候，如果條件不成立，程式就什麼都不做。但很多時候我們想說：「如果這樣就做 A，否則就做 B。」這就是 `else` 的用途。

1.10.2.1 基本用法

```
fn main() {
    let x = 2;
    if x > 5 {
        println!("大");
    } else {
        println!("小");
    }
}
```

x 是 2，2 大於 5 嗎？不是，所以跳過 `if` 的大括號，執行 `else` 的大括號，印出「小」。

1.10.2.2 換個值試試

把 x 改成 8：

```
fn main() {
    let x = 8;
    if x > 5 {
        println!("大");
    } else {
        println!("小");
    }
}
```

這次印出「大」，因為 8 大於 5，條件成立，走 `if` 那邊。

1.10.2.3 白話文

你可以把 `if...else...` 想成：

> **如果**條件成立，就做這個；**否則**，就做那個。

一定會走其中一邊，不會兩邊都走，也不會兩邊都不走。

1.10.3 重點整理

- `else` 接在 `if` 後面，處理條件不成立時要做的事。
- `if...else...` 是二選一：一定會走其中一邊，不會兩邊都走或都不走。

1.11 else if

1.11.1 本集目標

用 `else if` 處理多個條件分支——不只二選一，還能三選一、四選一。

1.11.2 正文

上一集學的 `if...else...` 只能處理「二選一」。但如果有更多情況呢？比如成績分等第：A、B、C、F……這時候就需要 `else if`。

1.11.2.1 範例：成績等第

```
fn main() {
    let score = 85;

    if score >= 90 {
        println!("A");
    } else if score >= 80 {
        println!("B");
    } else if score >= 70 {
        println!("C");
    } else {
        println!("F");
    }
}
```

1.11.2.2 它是怎麼判斷的？

Rust 會**從上到下**，一個一個條件去看：

1. `score >= 90`？85 >= 90？不是，跳過。
2. `score >= 80`？85 >= 80？是！印 `"B"`，然後結束。
3. 後面的都不看了。

這很重要：**一旦某個條件成立，後面的都會被跳過**。

1.11.2.3 試試其他分數

- `score = 95` → 印 `"A"`。
- `score = 73` → 印 `"C"`。
- `score = 50` → 印 `"F"`（前面全部不成立，走到 `else`）。

1.11.2.4 結構

```
if 條件1 {
    ...
} else if 條件2 {
    ...
} else if 條件3 {
    ...
} else {
    ...（以上都不成立時）
}
```

你可以放任意多個 `else if`，最後的 `else` 是選擇性的（但通常建議加上去，以防漏掉什麼情況）。

1.11.3 重點整理

- `else if` 可以處理多個條件分支，不只二選一。
- Rust 從上到下檢查條件，第一個成立的就執行，後面全部跳過。
- 最後的 `else` 是選擇性的，用來處理「以上條件都不成立」的情況。

1.12 邏輯運算子

1.12.1 本集目標

學會用 `&&`（而且）、`||`（或者）、`!`（不是）來組合多個條件。

1.12.2 正文

上幾集我們學了 `if`，但條件都很簡單——只有一個。現實中常常需要同時考慮好幾個條件，比如「年滿 18 歲**而且**是學生」。這就需要邏輯運算子。

1.12.2.1 `&&` —— 而且（AND）

兩個條件**都要成立**，結果才是 `true`：

```
fn main() {
    let age = 24;
    let is_student = true;

    if age >= 18 && is_student {
        println!("是個成年學生");
    }
}
```

因為 24 >= 18 是 `true`，`is_student` 也是 `true`，兩個都成立，所以整體是 `true`。

如果把 `age` 改成 15，15 >= 18 是 `false`，不管 `is_student` 是不是 `true`，整體就是 `false`，就不會印了。

1.12.2.2 `||` —— 或者（OR）

只要**其中一個**條件成立，結果就是 `true`：

```
fn main() {
    let is_weekend = false;
```

```
    let is_holiday = true;

    if is_weekend || is_holiday {
        println!("今天放假！");
    }
}
```

雖然 is_weekend 是 false，但 is_holiday 是 true，只要有一個是 true 就夠了。

1.12.2.3 ！—— 不是（NOT）

把 true 變成 false，把 false 變成 true：

```
fn main() {
    let raining = false;

    if !raining {
        println!("出門走走吧！");
    }
}
```

raining 是 false，加上！之後就變成 true，所以條件成立。

你可以讀成：「如果**沒有**在下雨，就出門走走。」

1.12.3 重點整理

- &&（而且）：兩邊都為 true 才是 true。
- ||（或者）：只要一邊為 true 就是 true。
- ！（不是）：把 true 變 false，false 變 true。
- 用邏輯運算子組合多個條件，寫出更精確的判斷。

1.13 let mut

1.13.1 本集目標

了解 Rust 的變數預設不可變，要用 mut 才能改值。

1.13.2 正文

今天來聊 Rust 一個很有特色的設計——**變數預設不可變**。

1.13.2.1 先看看會怎樣

```
fn main() {
    let x = 5;
    x = 10;
    println!("{}", x);
}
```

你覺得會印出 10 嗎？不會。你會得到一個編譯錯誤。Rust 在跟你說：「x 是不可變的，你不能再給它新的值。」

1.13.2.2 等等，為什麼不行？

在很多程式語言裡，變數就是可以隨便改的。但 Rust 的態度是：**如果你不打算改它，就別讓它可以被改**。

為什麼？因為如果你知道一個值不會變，你在讀程式的時候就不用擔心它被偷改了。這在大型程式裡很重要。

1.13.2.3 要改的話，加 `mut`

如果你確實需要改變值，加上 `mut`（mutable 的縮寫，意思是「可變的」）：

```
fn main() {
    let mut x = 5;
    println!("x 原本是 {}", x);
    x = 10;
    println!("x 現在是 {}", x);
}
```

這次就沒問題了！因為你用 `let mut` 告訴 Rust：「這個變數我之後會改。」

1.13.2.4 小整理

```
    let x = 5;     // 不可變，之後不能改
    let mut x = 5; // 可變，之後可以改
```

Rust 不是不讓你改變數，它只是要你**明確說出來**。這是 Rust 的一個設計哲學：**有意識地做出選擇**。

1.13.3 重點整理

- Rust 的變數預設是**不可變**的，不能重新賦值。
- 要讓變數可以改值，宣告時加上 `mut`：`let mut x = 5;`。
- 修改可變變數的值：直接用 `x = 新值;`（不需要再寫 `let`）。
- 這是 Rust 的設計哲學：要你明確選擇，而不是默默允許修改。

1.14 複合賦值運算子

1.14.1 本集目標

學會用 `+=`、`-=` 等簡寫方式來更新變數的值。

1.14.2 正文

上一集學了 `let mut` 讓變數可以改值。今天來學一個偷懶的寫法。

1.14.2.1 `x = x + 5` 是什麼意思？

先來看一個很重要的觀念。假設你有一個變數 x 是 10，你想讓它加 5：

```
fn main() {
    let mut x = 10;
    x = x + 5;
    println!("{}", x); // 15
}
```

這裡 x 同時出現在 = 的左邊和右邊。這不是數學上的「x 等於 x + 5」（那在數學上根本不成立對

吧？），而是程式語言的意思：**先算右邊的 x + 5（也就是 10 + 5 = 15），然後把結果存回左邊的 x**。所以 x 的值就從 10 變成了 15。

1.14.2.2 += 簡寫

上面那行其實有個更簡短的寫法：

```
fn main() {
    let mut x = 10;
    x += 5;
    println!("{}", x); // 15
}
```

x += 5 的意思就是 x = x + 5，只是比較簡潔。

1.14.2.3 其他複合賦值運算子

減法、乘法、除法、取餘數都有對應的簡寫：

```
fn main() {
    let mut a = 20;

    a -= 3;
    println!("20 - 3 = {}", a); // 17

    a *= 2;
    println!("17 * 2 = {}", a); // 34

    a /= 4;
    println!("34 / 4 = {}", a); // 8（整數除法）

    a %= 3;
    println!("8 % 3 = {}", a);  // 2
}
```

1.14.2.4 一覽表

簡寫	等同於
x += 5	x = x + 5
x -= 5	x = x - 5
x *= 5	x = x * 5
x /= 5	x = x / 5
x %= 5	x = x % 5

1.14.2.5 小提醒

要用這些運算子，變數一定要是 let mut 宣告的，因為你正在**改變**它的值。

1.14.3 重點整理

- 複合賦值運算子：+=、-=、*=、/=、%=。
- x += 5 就是 x = x + 5 的簡寫——先算右邊，再存回左邊。
- 使用前提：變數必須用 let mut 宣告。

1.15 stdin

1.15.1 本集目標

讓程式讀取使用者的鍵盤輸入——先照抄，不用完全理解每一行。

1.15.2 正文

到目前為止，我們程式裡的值都是寫死的。但如果想讓使用者自己輸入呢？比如讓使用者輸入名字，然後程式跟他打招呼？

1.15.2.1 先照抄這段程式碼

```
fn main() {
    println!("請輸入你的名字：");

    let mut input = String::new();
    std::io::stdin().read_line(&mut input).expect("讀取失敗");

    println!("你好，{}！", input.trim());
}
```

跑起來的效果：

```
請輸入你的名字：
Andy
你好，Andy！
```

1.15.2.2 這段在幹嘛？

我知道這段看起來有點嚇人，但別擔心，我們先把它當成一個**黑盒子**——你只要知道它能讀取使用者輸入就好。

大概的意思是：

1. `let mut input = String::new();` → 建立一個空的文字變數，準備接收輸入。
2. `std::io::stdin().read_line(&mut input).expect("讀取失敗");` → 從鍵盤讀一行文字，存到 `input` 裡。
3. `input.trim()` → 把多餘的空白和換行符號去掉。

至於 `String::new()`、`&mut`、`.expect()` 這些是什麼意思？以後會慢慢教，現在先照抄就好。

1.15.2.3 為什麼先不解釋？

因為要解釋這段程式碼，需要先理解好幾個還沒學的觀念。與其硬塞一堆看不懂的解釋，不如先學會用，之後自然就懂了。

就像小時候你學騎腳踏車，不用先學力學和陀螺效應——先騎就對了。

1.15.2.4 重要的事

每次要讀使用者輸入，就把這三行拿去用：

```
    let mut input = String::new();
    std::io::stdin().read_line(&mut input).expect("讀取失敗");
    let name = input.trim(); // 去掉尾巴的換行
```

1.15.3 重點整理

- 讀取使用者輸入的三行固定寫法：`String::new()` → `stdin().read_line(&mut input)` → `.trim()`。
- 這段先當成黑盒子照抄即可，背後的觀念之後會慢慢學。
- `.trim()` 用來去掉輸入尾巴的換行符號。

1.16 parse

1.16.1 本集目標

學會把使用者輸入的文字轉換成數字。

1.16.2 正文

和上一集一樣，這集的語法先照抄就好，不用完全理解每一行在做什麼。之後學到更多概念會回來解釋。

上一集我們學了怎麼讀取使用者的輸入，但讀進來的東西是**文字**。如果使用者輸入了 42，對 Rust 來說那是一串文字，不是數字 42。

你不能拿文字去做加減乘除，所以我們需要把它「轉換」成數字。

1.16.2.1 文字轉數字

```rust
fn main() {
    println!("請輸入一個數字：");

    let mut input = String::new();
    std::io::stdin().read_line(&mut input).expect("讀取失敗");

    let num = input.trim().parse::<i32>().expect("不是數字");

    println!("你輸入的數字是 {}", num);
}
```

跑起來：

```
請輸入一個數字：
42
你輸入的數字是 42
```

1.16.2.2 關鍵在這行

```rust
    let num = input.trim().parse::<i32>().expect("不是數字");
```

拆開來看：

1. `input.trim()` → 去掉頭尾的空白和換行。
2. `.parse::<i32>()` → 把文字**解析**成整數（i32 就是一種整數型別）。
3. `.expect("不是數字")` → 如果轉換失敗（比如使用者輸入了 “abc”），就印這個錯誤訊息然後結束程式。

1.16.2.3 完整的互動範例

```rust
fn main() {
    println!("請輸入一個數字：");

    let mut input = String::new();
    std::io::stdin().read_line(&mut input).expect("讀取失敗");

    let num = input.trim().parse::<i32>().expect("不是數字");

    println!("{} 乘以 2 等於 {}", num, num * 2);
}
```

```
請輸入一個數字：
7
7 乘以 2 等於 14
```

現在你可以讀取數字並拿來運算了！

1.16.3 重點整理

- 使用者輸入的東西是文字，要用 `.parse::<i32>()` 轉成整數才能做運算。
- `.expect("錯誤訊息")` 在轉換失敗時會印出訊息並結束程式。
- 完整流程：`input.trim().parse::<i32>().expect("不是數字")`。

1.17 綜合練習

1.17.1 本集目標

把前面學的東西組合起來，做一個「輸入分數 → 判斷等第」的小程式。

1.17.2 正文

恭喜你撐到現在！今天我們要把前面學的東西全部串起來，做一個真正有用的小程式。

1.17.2.1 目標

讓使用者輸入分數，程式自動判斷等第並印出來。

1.17.2.2 完整程式碼

```rust
fn main() {
    println!("請輸入你的分數：");

    // 讀取使用者輸入
    let mut input = String::new();
    std::io::stdin().read_line(&mut input).expect("讀取失敗");

    // 把文字轉成數字
    let score = input.trim().parse::<i32>().expect("不是數字");

    // 判斷等第
    if score >= 90 {
        println!("你的成績是 A");
```

```
    } else if score >= 80 {
        println!("你的成績是 B");
    } else if score >= 70 {
        println!("你的成績是 C");
    } else {
        println!("你的成績是 F");
    }
}
```

1.17.2.3 跑跑看

```
請輸入你的分數：
85
你的成績是 B
```

```
請輸入你的分數：
92
你的成績是 A
```

```
請輸入你的分數：
45
你的成績是 F
```

1.17.2.4 回顧一下用了哪些技巧

1. `println!` → 印提示訊息（第 2 集）
2. `let mut` + `String::new()` → 準備接收輸入（第 15 集）
3. `stdin().read_line(&mut input)` → 讀取鍵盤輸入（第 15 集）
4. `.trim().parse::<i32>()` → 文字轉數字（第 16 集）
5. `if` / `else if` / `else` → 條件判斷（第 8、10、11 集）

看到了嗎？把各種功能組合起來就能做出一個有互動性的小程式。這就是程式設計的魅力——把小塊的知識拼在一起，就能做出有用的東西。

1.17.2.5 挑戰

如果你想多練習，試試看：

- 加一個 D 等第（60 ~ 69 分）。
- 如果分數超過 100 或小於 0，印出「分數不正確」。

1.17.3 重點整理

- 結合前面學的 `stdin`、`parse`、`if` 和 `else if`，就能做出有互動性的程式。
- 程式設計的魅力：把小塊知識拼在一起，就能做出有用的東西。
- 除了學新語法，多練習組合應用也很重要。

1.18 loop + break

1.18.1 本集目標

用 `loop` 做出一個無限迴圈，再用 `break` 在適當的時機跳出來。

1.18.2 正文

到目前為止，我們的程式都是跑一次就結束。但如果有些事情需要重複做呢？比如倒數計時：5、4、3、2、1、發射！

這就需要**迴圈**。

1.18.2.1 `loop` —— 無限迴圈

`loop` 就是一直跑、一直跑，永遠不停：

```
fn main() {
    loop {
        println!("停不下來啊啊啊");
    }
}
```

如果你真的跑了這段程式，它會一直印一直印……你得用 `Ctrl + C` 強制停止它。

所以我們需要一個「出口」。

1.18.2.2 `break` —— 跳出迴圈

```
fn main() {
    let mut count = 5;
    loop {
        if count == 0 {
            println!("發射！");
            break;
        }
        println!("{}", count);
        count -= 1;
    }
}
```

1.18.2.3 它是怎麼運作的？

1. `count` 從 5 開始。
2. 進入 `loop`，先檢查 `count == 0` 嗎？不是，印出 5，然後 `count -= 1`，count 變成 4。
3. 再回到 `loop` 的開頭，檢查 `count == 0` 嗎？不是，印出 4，count 變成 3。
4. ……一直重複……
5. 當 `count` 變成 0，`if count == 0` 成立，印出「發射 !」，然後 `break` 跳出迴圈。
6. 程式結束。

1.18.3 重點整理

- `loop` 會無限重複執行大括號裡的程式碼。
- `break` 用來跳出迴圈，讓程式繼續往下跑。
- 沒有 `break` 的 `loop` 就是無限迴圈，記得要有出口。

1.19 while

1.19.1 本集目標

用 while 迴圈改寫倒數計時，對比 loop + break 的寫法。

1.19.2 正文

上一集我們用 loop + break 做了倒數計時。今天來學另一種迴圈——while，它讓同樣的邏輯寫起來更乾淨。

1.19.2.1 用 while 改寫倒數計時

```
fn main() {
    let mut count = 5;
    while count > 0 {
        println!("{}", count);
        count -= 1;
    }
    println!("發射！");
}
```

結果一模一樣！

1.19.2.2 跟 loop + break 比一下

上一集的寫法：

```
fn main() {
    let mut count = 5;
    loop {
        if count == 0 {
            println!("發射！");
            break;
        }
        println!("{}", count);
        count -= 1;
    }
}
```

while 的寫法：

```
fn main() {
    let mut count = 5;
    while count > 0 {
        println!("{}", count);
        count -= 1;
    }
    println!("發射！");
}
```

有看出差別嗎？while 把「條件判斷」和「迴圈」合在一起了。不需要自己寫 if 和 break，只要告訴 while：「一旦這個條件成立，就繼續跑。」

1.19.2.3 while 的白話文

當條件成立的時候，就一直做大括號裡的事。

`while count > 0` → 只要 `count` 大於 0，就繼續跑。`count` 一旦變成 0，條件不成立了，就自動停下來。

1.19.2.4 什麼時候用 `loop`，什麼時候用 `while`？

- **`while`**：你在迴圈開始前就知道要不要繼續（先檢查條件，再決定要不要跑）。
- **`loop` + `break`**：你需要在迴圈中間某個地方才能決定要不要停下來。

兩種都可以達成目的，只是 `while` 在很多情況下寫起來更簡潔。

1.19.3 重點整理

- `while` 迴圈：只要條件成立就一直跑，條件不成立就自動停。
- 比 `loop` + `break` 更簡潔，適合在迴圈開始前就知道要不要繼續的情況。
- `loop` + `break` 適合在迴圈中間才能決定要不要停的情況。

1.20 `for` + range

1.20.1 本集目標

用 `for` 迴圈搭配範圍（range）來重複執行，不用自己處理計數器。

1.20.2 正文

前兩集我們學了 `loop` 和 `while`，都需要自己手動管理計數器（`count -= 1` 之類的）。今天來學一個更簡單的寫法——`for` 迴圈。

1.20.2.1 `for` + 範圍

```
fn main() {
    for i in 0..5 {
        println!("{}", i);
    }
}
```

1.20.2.2 `0..5` 是什麼？

`0..5` 叫做**範圍**（range），意思是「從 0 開始，到 5 之前」。注意，**不包含 5**！

所以 `0..5` 就是 0、1、2、3、4 這五個數字。

你可以把 `for i in 0..5` 讀成：「讓 `i` 依序從 0 跑到 4，每次做大括號裡的事。」

注意這裡的 `i` 不需要寫 `let`——`for` 會自動幫你宣告它。而且 `i` 在 `for` 後面的大括號 `{}` 裡面都可以使用。

1.20.2.3 要包含結尾呢？用 `0..=5`

```
fn main() {
    for i in 0..=5 {
        println!("{}", i);
    }
}
```

`0..=5` 多了一個 `=`，表示「包含 5」。

1.20.2.4 比較一下

語法	意思	產生的數字
`0..5`	0 到 4	0, 1, 2, 3, 4
`0..=5`	0 到 5	0, 1, 2, 3, 4, 5
`1..4`	1 到 3	1, 2, 3
`1..=4`	1 到 4	1, 2, 3, 4

1.20.2.5 `while` 和 `for` 也能用 `break`

之前在 `loop` 裡面學過 `break`，其實 `while` 和 `for` 也能用。要注意的是，`break` 只會跳出**迴圈**，不會跳出 `if` 之類的控制結構。所以下面這段程式碼裡，`break` 是跳出 `for` 迴圈，不是跳出 `if`：

```
fn main() {
    for i in 0..10 {
        if i == 5 {
            println!("找到 5 了，不找了！");
            break;
        }
        println!("{}", i);
    }
}
```

跑起來只會印 0~4，遇到 5 就 `break` 跳出迴圈了。

1.20.3 重點整理

- `for i in 0..5` 讓 `i` 從 0 跑到 4（不含尾），`0..=5` 則包含 5。
- 相比 `while`，`for` 不用手動遞增計數器和檢查條件，更簡潔也更不容易出錯。
- `break` 在 `loop`、`while`、`for` 裡面都可以使用。

1.21 巢狀迴圈

1.21.1 本集目標

把迴圈放進迴圈裡——用巢狀迴圈印出九九乘法表。

1.21.2 正文

上一集學了 `for` 迴圈，今天來玩個進階的——把一個迴圈**放進另一個迴圈裡面**。

1.21.2.1 什麼是巢狀迴圈？

「巢狀」就是「一層包一層」的意思，像俄羅斯套娃一樣。外面的迴圈跑一次，裡面的迴圈就會從頭到尾完整跑完一次。

1.21.2.2 九九乘法表

來挑戰一下，用巢狀迴圈印出九九乘法表：

```
fn main() {
    for i in 1..=9 {
        for j in 1..=9 {
```

```
            print!("{} x {} = {}   ", i, j, i * j);
        }
        println!(); // 換行
    }
}
```

1.21.2.3 它是怎麼運作的？

1. 外面的迴圈 `i` 從 1 跑到 9。
2. 當 `i = 1` 時，裡面的迴圈 `j` 從 1 跑到 9 → 印出 1×1, 1×2, ... 1×9。
3. 裡面的迴圈跑完後，`println!()` 換行。
4. 外面的迴圈走到 `i = 2`，裡面的迴圈又從 1 跑到 9 → 印出 2×1, 2×2, ... 2×9。
5. 以此類推.....

1.21.2.4 `print!` vs `println!`

這裡用了一個新東西：`print!`。它跟 `println!` 很像，差別在於 `print!` 印完**不會換行**，而 `println!` 印完會換行。

1.21.2.5 視覺化

外迴圈跑一次 = 一行：

```
i=1 → [j=1, j=2, j=3, ... j=9] → 換行
i=2 → [j=1, j=2, j=3, ... j=9] → 換行
...
i=9 → [j=1, j=2, j=3, ... j=9] → 換行
```

1.21.2.6 `break` 只跳出最內層

在巢狀迴圈裡用 `break`，它只會跳出**最裡面**那一層迴圈，外面那層還會繼續跑：

```
fn main() {
    for i in 1..=3 {
        for j in 1..=3 {
            if j == 2 {
                break; // 只跳出內層迴圈
            }
            println!("i={}, j={}", i, j);
        }
    }
}
```

每次 `j` 到 2 就 `break` 了，但外層的 `i` 還是繼續跑 1、2、3。

1.21.2.7 loop label：跳出指定層

如果你想直接跳出外面那層呢？可以用 **loop label**：

```
fn main() {
    'outer: for i in 1..=3 {
        for j in 1..=3 {
            if j == 2 {
                break 'outer; // 跳出外層迴圈
            }
            println!("i={}, j={}", i, j);
        }
```

```
    }
    println!("結束！");
}
```

'outer: 是一個**標籤**（label），放在迴圈前面。break 'outer 就是說「跳出標記為 'outer 的那個迴圈」。注意標籤名稱前面要加 '（單引號）。

1.21.3 重點整理

- 巢狀迴圈就是迴圈裡面再放迴圈，外層每跑一次，內層就會從頭到尾完整跑完一次。
- print! 和 println! 的差別：print! 印完不換行。
- break 在巢狀迴圈裡只會跳出最內層的迴圈。
- 用 loop label（'outer: + break 'outer）可以跳出指定的外層迴圈。

1.22 continue

1.22.1 本集目標

用 continue 跳過迴圈中的某些迭代。

1.22.2 正文

之前學了 break 可以跳出迴圈。今天來學 continue——它不是跳出迴圈，而是**跳過這一次，直接進入下一次迴圈**。

1.22.2.1 只印奇數

```
fn main() {
    for i in 0..10 {
        if i % 2 == 0 {
            continue;
        }
        println!("{}", i);
    }
}
```

1.22.2.2 它是怎麼運作的？

迴圈 i 從 0 跑到 9：

- i = 0 → 0 % 2 == 0 嗎？是（偶數），continue！跳過，不印。
- i = 1 → 1 % 2 == 0 嗎？不是（奇數），繼續往下跑，印出 1。
- i = 2 → 偶數，continue，跳過。
- i = 3 → 奇數，印出 3。
- ……以此類推。

1.22.2.3 break vs continue

- **break**：整個迴圈結束，不再跑了。
- **continue**：這一次跳過，但迴圈繼續跑下一次。

和 break 一樣，continue 也是只作用在**迴圈**上，不會跳過 if 之類的控制結構。上面的程式碼裡，continue 是跳過 for 迴圈的這一次，不是跳過 if。

1.22.2.4 另一個例子

跳過 5 不印：

```
fn main() {
    for i in 1..=10 {
        if i == 5 {
            continue;
        }
        println!("{}", i);
    }
}
```

5 被跳過了，其他都正常印出來。

1.22.2.5 continue + loop label

上一集學過 break 'outer 可以跳出指定層迴圈，continue 也可以搭配 label：

```
fn main() {
    'outer: for i in 1..=3 {
        for j in 1..=3 {
            if j == 2 {
                continue 'outer; // 跳過外層迴圈的這一次
            }
            println!("i={}, j={}", i, j);
        }
    }
}
```

每次 j 到 2，就 continue 'outer 直接跳到外層迴圈的下一次迭代，所以 j=2 和 j=3 都不會印。

1.22.3 重點整理

- continue 跳過這一次，直接進入下一次迴圈。
- break 是「整個迴圈不跑了」，continue 是「這次跳過，跑下一次」。
- 搭配 if 可以有選擇性地跳過特定情況。
- continue 'outer 可以搭配 loop label 跳過外層迴圈的這一次。

1.23 型別（基礎）

1.23.1 本集目標

認識 Rust 的基本型別。

1.23.2 正文

到現在為止，我們寫 let x = 5; 的時候都沒有特別說 x 是什麼「型別」。今天來正式認識一下型別是什麼。

1.23.2.1 什麼是型別？

型別就是在告訴 Rust：「這個變數裡面放的是什麼東西。」

是整數？小數？文字？還是 true / false？不同的型別代表不同的資料。

1.23.2.2 手動標註型別

你可以在變數名稱後面加上： 型別 來指定：

```
fn main() {
    let x: i32 = 5;
    let negative: i32 = -10;
    let y: f64 = 3.14;
    let z: bool = true;

    println!("x = {}", x);
    println!("negative = {}", negative);
    println!("y = {}", y);
    println!("z = {}", z);
}
```

- `i32` → 整數（integer，32 位元）。
- `f64` → 浮點數（floating-point number，64 位元），用來表示可以有小數部分的數值，例如 `3.14` 或 `0.5`。
- `bool` → 布林值，只有 `true` 和 `false`。

1.23.2.3 那之前為什麼不用標？

因為 Rust 很聰明！它會看你給的值，自動**推斷**型別：

```
    let x = 5;    // Rust 自動判斷：這是 i32
    let y = 3.14; // Rust 自動判斷：這是 f64
    let z = true; // Rust 自動判斷：這是 bool
```

這叫做**型別推斷**（type inference）。大部分的時候 Rust 都能自己搞定，你不用特別標。

1.23.3 重點整理

- 三個基本型別：`i32`（整數）、`f64`（浮點數）、`bool`（布林值）。
- Rust 有**型別推斷**，大部分時候不用手動標註型別。
- 需要時可以用 `let x: i32 = 5;` 手動指定型別。

1.24 型別（數字詳解）

1.24.1 本集目標

認識 Rust 所有的數字型別，以及數字後綴的用法。

1.24.2 正文

上一集我們簡單認識了 `i32` 和 `f64`。今天來把 Rust 所有的數字型別都看一遍。

1.24.2.1 整數型別

Rust 的整數型別分成**有號**（可以是負數）和**無號**（只能是正數和零）：

有號	無號	位元數	範圍（有號）
i8	u8	8	-128 ~ 127
i16	u16	16	-32,768 ~ 32,767

有號	無號	位元數	範圍（有號）
`i32`	`u32`	32	約 ±21 億
`i64`	`u64`	64	超級大
`i128`	`u128`	128	天文數字
`isize`	`usize`	看系統	64 位元系統 = 64 位元

- `i` = integer（整數），`u` = unsigned（無號）。
- 數字代表用幾個位元來存——位元越多，能存的數字越大。
- `isize` 和 `usize` 的大小取決於你的系統是 32 位元還是 64 位元（現在幾乎都是 64 位元）。

日常用 `i32` 就夠了。不確定的時候，用 `i32`。

1.24.2.2 浮點數型別

浮點數用來表示可以有小數部分的數值，在 Rust 裡有兩種：

型別	精確度
`f32`	單精度（約 7 位有效位數）
`f64`	雙精度（約 15 位有效位數）

日常用 `f64` 就夠了。 Rust 預設的浮點數就是 `f64`。

1.24.2.3 浮點數運算

第 5 集教算術運算的時候，我們都用整數。浮點數一樣可以用 `+ - * / %`，但有一個重要的差別——**浮點數除法可以得到小數部分**：

```
fn main() {
    let a = 10.0;
    let b = 3.0;
    println!("{}", a / b); // 3.3333333333333335
    println!("{}", a % b); // 1
}
```

還記得第 5 集 `10 / 3` 的結果是 `3`（整數除法直接截斷）嗎？浮點數不會截斷，`10.0 / 3.0` 會得到 `3.3333...`。

另外注意 `a % b` 印出來的是 `1` 而不是 `1.0`。別誤會——它的型別還是 `f64`，只是 `{}` 在印浮點數時，小數部分是零就不會多印一個 `.0`。

不過浮點數有一個經典的坑——**精確度問題**：

```
fn main() {
    println!("{}", 0.1 + 0.2); // 0.30000000000000004
}
```

`0.1 + 0.2` 不是 `0.3`！這不是 Rust 的 bug，而是幾乎所有程式語言都有的浮點數精確度限制。電腦用二進位存小數，有些十進位小數沒辦法精確表示。知道有這件事就好，不用太擔心。

1.24.2.4 Rust 怎麼推斷數字型別？

當你寫 `let x = 5;`，Rust 預設把它當成 `i32`。當你寫 `let y = 3.14;`，Rust 預設把它當成 `f64`。

但 Rust 不只看數字本身，它也會根據**你怎麼使用這個變數**來推斷型別。有時候根據上下文，Rust 會

推斷出 i32 以外的整數型別。這個之後遇到的時候會更清楚。

不過基本上來說，整數預設就是 i32、浮點數預設就是 f64。

1.24.2.5 數字後綴（literal suffix）

如果你想要指定型別，除了 `let x: i64 = 5;` 之外，還有一個更簡潔的寫法——直接在數字後面加型別名稱：

```
fn main() {
    let a = 5i32;      // i32
    let b = 5u8;       // u8
    let c = 3.14f64;   // f64
    let d = 2.0f32;    // f32
    let e = 100000i64; // i64

    println!("{} {} {} {} {}", a, b, c, d, e);
}
```

5i32 意思就是「5 這個數字，型別是 i32」。數字和型別之間不用空格，直接接在一起。

1.24.2.6 小提醒

不同型別的數字**不能直接混著算**：

```
fn main() {
    let a: i32 = 5;
    let b: i64 = 10;
    println!("{}", a + b); // ✕ 編譯錯誤！i32 和 i64 不能直接相加
}
```

這是 Rust 的安全設計：Rust 一般不自動幫你轉換型別。

1.24.3 重點整理

- 整數分有號（i8-i128）和無號（u8-u128），日常用 i32 就夠了。
- isize 和 usize 的大小取決於系統（64 位元上是 64 位元）。
- 浮點數有 f32 和 f64，日常用 f64（Rust 預設）。
- 數字後綴（如 5i32、3.14f64）可以直接指定型別。
- 浮點數除法可以得到小數部分，但有精確度問題（0.1 + 0.2 ≠ 0.3）。
- Rust 一般不自動幫你轉換型別。

1.25 char

1.25.1 本集目標

認識 char 型別——用來存放「一個字元」的型別。

1.25.2 正文

之前我們用過字串（string，用雙引號 " 包起來的文字），今天來認識一個更小的單位——**字元**（char）。

1.25.2.1 char 是什麼？

char 就是**一個字元**。注意，是「一個」，不是一串。

```
fn main() {
    let c = 'A';
    let c2 = '你';
    let c3 = '🦀';

    println!("{}", c);
    println!("{}", c2);
    println!("{}", c3);
}
```

1.25.2.2 單引號 vs 雙引號

這很重要：

- **單引號** `'` → char，只能放**一個**字元。
- **雙引號** `"` → 字串，可以放很多字元。

```
fn main() {
    let c = 'A';     // char，一個字元
    let s = "Hello"; // 字串，五個字元
}
```

如果你用單引號放超過一個字元，Rust 會報錯：

```
    let c = 'AB'; // ✕ 錯誤！char 只能放一個字元
```

1.25.2.3 Unicode

Rust 的 char 支援 **Unicode**，所以不只是英文字母，中文、日文、甚至 emoji 都可以：

```
fn main() {
    let letter = 'R';
    let chinese = '美';
    let japanese = 'の';
    let emoji = '😊';

    println!("{} {} {} {}", letter, chinese, japanese, emoji);
}
```

每一個都是合法的 char。

1.25.2.4 型別標註

如果你想明確標註型別：

```
fn main() {
    let c: char = 'Z';
    println!("{}", c);
}
```

不過通常不用特別標，Rust 看到單引號就知道是 char。

1.25.3 重點整理

- char 是「一個字元」的型別，用**單引號**包起來：`'A'`、`'你'`、`'🦀'`。

- 支援 Unicode，中文、日文、emoji 都是合法的 char。
- 單引號 ' = char（一個字元），雙引號 " = 字串（一串字元），別搞混。

1.26 跳脫字元

1.26.1 本集目標

學會用反斜線 \ 在字串裡插入換行、tab 等特殊字元。

1.26.2 正文

有時候你想在字串裡面放一些「特殊」的東西，比如換行、tab、或者雙引號本身。這時候就需要**跳脫字元**（escape character）。

1.26.2.1 \n —— 換行

```
fn main() {
    println!("第一行\n第二行");
}
```

\n 就是告訴 Rust：「這裡換一行。」它不會真的印出 \n 這兩個字，而是產生一個換行的效果。

1.26.2.2 \t —— tab

```
fn main() {
    println!("名字\t分數");
    println!("小明\t85");
    println!("小華\t92");
}
```

\t 會插入一個 tab 空間。

1.26.2.3 \\ —— 反斜線本身

如果你想印出反斜線 \ 本身呢？因為 \ 已經被拿來當跳脫字元的開頭了，所以要用兩個反斜線：

```
fn main() {
    println!("檔案路徑：C:\\Users\\Andy");
}
```

1.26.2.4 \" —— 雙引號

字串是用 " 包起來的，那如果字串裡面要有 " 呢？

```
fn main() {
    println!("他說：\"你好！\"");
}
```

\" 告訴 Rust：「這個雙引號是字串內容，不是字串的結尾。」

1.26.2.5 在 char 裡使用

跳脫字元在 char 裡面也能用：

```
fn main() {
    let newline: char = '\n';
```

```
    let tab: char = '\t';
    let backslash: char = '\\';

    print!("A{}B{}C{}", newline, tab, backslash);
}
```

1.26.2.6 \' —— 單引號

在 char 裡面，如果你想表示單引號本身，就要跳脫：

```
fn main() {
    let quote: char = '\'';
    println!("{}", quote);
}
```

因為 char 是用 ' 包起來的，所以裡面要放 ' 就得用 \'。

1.26.2.7 不需要跳脫的情況

在字串（""）裡面，單引號不需要跳脫，可以直接用：

```
fn main() {
    println!("It's a test"); // ' 在字串裡不用跳脫
}
```

同樣地，在 char（''）裡面，雙引號也不需要跳脫：

```
fn main() {
    let c: char = '"'; // " 在 char 裡不用跳脫
    println!("{}", c);
}
```

簡單來說：**包在外面的那個符號才需要跳脫，另一個不用。**

1.26.2.8 一覽表

跳脫字元	效果
\n	換行
\t	tab
\\	反斜線 \
\"	雙引號 "
\'	單引號 '

1.26.3 重點整理

- 跳脫字元用 \ 開頭，代表特殊字元：\n（換行）、\t（tab）、\\（反斜線）。
- \" 在字串裡表示雙引號本身，\' 在 char 裡表示單引號本身。
- 跳脫字元在字串和 char 裡都能使用。
- 規則：包在外面的那個符號才需要跳脫，另一個不用。

1.27 if 當表達式

1.27.1 本集目標

學會把 if 當成一個「表達式」，直接用它來給變數賦值。

1.27.2 正文

這是第 1 章的最後一集！今天要介紹 Rust 一個很酷的特性——if 不只是判斷用的，它還可以**回傳值**。

1.27.2.1 先看一般的寫法

假設你要根據條件給變數不同的值，你可能會這樣寫：

```
fn main() {
    let condition = true;
    let x;

    if condition {
        x = 1;
    } else {
        x = 2;
    }

    println!("{}", x);
}
```

這樣沒問題，但 Rust 有一個更簡潔的寫法。

1.27.2.2 if 當表達式

```
fn main() {
    let condition = true;
    let x = if condition { 1 } else { 2 };

    println!("{}", x);
}
```

跑起來印出 1。

看到了嗎？if condition { 1 } else { 2 } 整個放在 let x = 的右邊，直接把結果賦值給 x。

如果 condition 是 true，x 就是 1；如果是 false，x 就是 2。

1.27.2.3 注意：大括號裡面不加分號

```
fn main() {
    let condition = true;
    let x = if condition { 1 } else { 2 };
    //                       ^          ^
    //                   沒有分號      沒有分號
}
```

這些值（1 和 2）後面**沒有分號**。在 Rust 裡，不加分號的值就是「回傳值」。這是 Rust 的表達式語法，之後學函數的時候會更詳細地講。

1.27.2.4 兩邊型別要一致！

```
fn main() {
    let condition = true;
    let x = if condition { 1 } else { "hello" }; // ✕ 錯誤！
}
```

這會報錯，因為 1 是整數，"hello" 是字串。Rust 不允許 x 有時候是數字、有時候是字串——它需要一個確定的型別。

兩邊的大括號裡，值的型別**必須相同**：

```
    // ✓ 兩邊都是整數
    let x = if condition { 1 } else { 2 };

    // ✓ 兩邊都是字串
    let msg = if condition { "好" } else { "壞" };

    // ✕ 一邊整數一邊字串
    let bad = if condition { 1 } else { "hello" };
```

1.27.2.5 這有什麼好處？

1. x 不需要是 mut。
2. 程式碼更簡潔。
3. Rust 的設計哲學：很多東西都可以是「表達式」，都能回傳值。

1.27.3 重點整理

- if 在 Rust 裡是**表達式**，可以直接回傳值：let x = if condition { 1 } else { 2 };。
- 大括號裡作為回傳值的部分**不加分號**。
- if 和 else 兩邊的型別必須一致。

恭喜你完成了第 1 章！🎉 你已經學會了 Rust 的基本語法，包括變數、運算、條件判斷、迴圈、型別等等。下一章我們會開始學更多 Rust 的特色功能！

第 2 章

函數、陣列與切片

在本章中，我們會學習如何將程式進行最基本的模組化。除此之外，你也會學到如何操作複數個值，將它們儲存在寫死的程式碼中並觀看其中部分的值。

2.1 const

2.1.1 本集目標

用 const 宣告一個永遠不會變的常數，並了解它和 let 的差別。

2.1.2 正文

第 1 章我們學了 let 來宣告變數，今天來認識它的好朋友——const。

const 就是「常數」，意思是：**這個值從頭到尾都不會變，而且在編譯的時候就已經決定好了。**

來看語法：

```
fn main() {
    const MAX_SCORE: i32 = 100;
    println!("最高分是：{}", MAX_SCORE);
}
```

看起來跟 let 很像對吧？但有幾個重要的差別：

2.1.2.1 差別一：const 一定要標型別

```
fn main() {
    const MAX_SCORE: i32 = 100; // ✅ 一定要寫 : i32
    let max_score = 100;        // ✅ let 可以省略，編譯器會自己推
}
```

用 const 的時候，你不能偷懶不寫型別，編譯器會跟你抱怨。

2.1.2.2 差別二：命名慣例是全大寫加底線

```
fn main() {
    const MAX_SCORE: i32 = 100;     // ✅ 全大寫，用底線分隔
    const PI_VALUE: f64 = 3.14159; // ✅ 這樣
    const maxScore: i32 = 100;     // ⚠ 可以編譯，但編譯器會警告你
}
```

這是 Rust 社群的慣例：常數用 SCREAMING_SNAKE_CASE（全大寫蛇形命名）。不遵守的話程式還是能跑，但編譯器會碎碎念。

2.1.2.3 差別三：const 不能用 mut

```
    const mut MAX: i32 = 100; // ✕ 不存在這種東西
    let mut x = 5;            // ✓ 這個可以
```

常數就是常數，不能變就是不能變，沒有「可變的常數」這種矛盾的東西。

2.1.2.4 差別四：const 可以放在 fn 外面

```
const MAX_PLAYERS: i32 = 10;

fn main() {
    println!("最多 {} 位玩家", MAX_PLAYERS);
}
```

const 可以宣告在程式最外層，let 不行。

2.1.2.5 什麼時候用 const？

當你有一個值是**固定不變**的，而且你在寫程式的時候就知道它是多少，就用 const。比如：

```
const TAX_RATE: f64 = 0.05;
const MAX_RETRY: i32 = 3;
```

2.1.3 重點整理

- const 宣告編譯時期常數，值永遠不會變。
- 一定要標型別（不能省略）。
- 命名慣例是全大寫加底線，像 MAX_SCORE。
- 不能加 mut。
- 可以放在 fn 外面，讓程式的其他部分也能用。

2.2 shadowing

2.2.1 本集目標

用 let 重新宣告同名變數（shadowing），以及它和 mut 的關鍵差別。

2.2.2 正文

Rust 有一個很有趣的功能叫做 **shadowing**（遮蔽）。簡單說就是：你可以用 let 再次宣告一個同名的變數，新的會「蓋掉」舊的。

```
fn main() {
    let x = 5;
    let x = x + 1;
    println!("x = {}", x);
}
```

第二行的 let x = x + 1; 其實是在說：「我要建立一個**全新的** x，它的值是舊的 x 加 1。」舊的 x 就被蓋掉了，從此以後 x 就是 6。

你甚至可以連續 shadow 好幾次：

```
fn main() {
    let x = 1;
    let x = x + 1; // x = 2
    let x = x * 3; // x = 6
    println!("x = {}", x);
}
```

2.2.2.1 shadowing vs `mut`：最大的差別

「等等，這跟 `mut` 有什麼不一樣？不都是改值嗎？」

最大的差別是：**shadowing 可以換型別，`mut` 不行。**

```
fn main() {
    // shadowing：可以從數字變成字串
    let x = 5;
    let x = "hello";
    println!("x = {}", x);
}
```

這完全合法！因為第二個 `let x` 是一個**全新的變數**，只是剛好同名而已。

但如果用 `mut` 試試看：

```
fn main() {
    let mut x = 5;
    x = "hello"; // ✗ 編譯錯誤！不能把字串塞進 i32
}
```

`mut` 只是讓你改「值」，型別還是鎖死的。但 shadowing 是建立一個全新的變數，所以型別可以完全不同。

2.2.2.2 實際用途

shadowing 最常見的用途是「轉換型別但保留名字」：

```
fn main() {
    let input = "42"; // 這是字串
    let input = input.trim().parse::<i32>().expect("不是數字"); // 現在是 i32，名字相同
    println!("input + 1 = {}", input + 1);
}
```

如果沒有 shadowing，你就得取兩個不同的名字，像 `input_str` 和 `input_num`，有點囉嗦。

2.2.2.3 shadowing 和作用域

還記得第 1 章第 9 集學的作用域嗎？Shadowing 在大括號 `{}` 裡面也能用，而且出了大括號，遮蔽就會結束，舊的變數會「回來」：

```
fn main() {
    let x = 1;
    {
        let x = 2; // 在這個區塊這行之後，x 被遮蔽為 2
        println!("區塊內 x = {}", x); // 2
    }
    println!("區塊外 x = {}", x);     // 1
}
```

大括號裡的 `let x = 2` 建立了一個新的 x，遮蔽了外面的 x。但這個遮蔽只在大括號裡面有效——一出大括號，新的 x 就消失了，原本的 x（值為 1）又可以使用了。

這跟 `mut` 完全不同。如果用 `mut` 在區塊裡改值，出了區塊值就真的變了：

```
fn main() {
    let mut x = 1;
    {
        x = 2; // 直接改值，不是 shadowing
    }
    println!("x = {}", x); // 2
}
```

所以再強調一次：shadowing 是**建立新變數**，`mut` 是**改舊變數的值**。在作用域裡這個差別特別明顯。

2.2.3 重點整理

- 用 `let` 重新宣告同名變數叫做 shadowing。
- 新的變數會蓋掉舊的。
- 和 `mut` 最大的差別：**shadowing 可以換型別**。
- 實際上每次 `let` 都是建立一個全新的變數，只是名字一樣。
- 在大括號裡 shadow 的變數，出了大括號就消失，原本的變數會「回來」。

2.3 底線變數

2.3.1 本集目標

用底線 _ 開頭的變數名來告訴編譯器「我知道這個沒用到，別唸我」。

2.3.2 正文

Rust 的編譯器很貼心（有時候有點煩），如果你宣告了一個變數但沒有使用它，它會給你一個警告：

```
fn main() {
    let x = 5;
    // 沒有用到 x
}
```

程式還是能跑，但那個黃色的警告看了就不舒服。怎麼消除呢？

2.3.2.1 方法一：加底線前綴

在變數名前面加一個底線 _：

```
fn main() {
    let _x = 5;
    // 沒有用到 _x，但編譯器不會警告了
}
```

這樣編譯器就懂了：「喔，你是故意不用的，好吧。」

注意，_x 還是一個正常的變數，你想用的話還是可以用：

```
fn main() {
    let _x = 5;
    println!("{}", _x);  // 還是可以用
```

```
}
```

2.3.2.2 方法二：單獨的底線 _

如果你連名字都不想取，就直接用一個底線：

```
fn main() {
    let _ = 42;
}
```

這代表「我完全不在乎這個值」。沒有名字，你之後也沒辦法用它。

2.3.2.3 _x vs _ 的差別

- _x：有名字，值會被保留，之後還能用。
- _：沒名字，值馬上就丟掉了。

大部分情況下用哪個都行。

2.3.2.4 for 迴圈也能用底線

上一章學了 `for i in 0..5`，迴圈變數 `i` 會依序是 0, 1, 2, 3, 4。但如果你只是想重複做某件事五次，根本不在乎現在是第幾次呢？這時候就可以用 _：

```
fn main() {
    for _ in 0..5 {
        println!("重複五次！");
    }
}
```

`for _ in 0..5` 的意思是「跑五次，但我不需要知道目前是第幾次」。

2.3.2.5 實際用途：猜數字

來看一個稍微完整一點的例子——讓玩家挑戰在五次以內猜中一個數字：

```
fn main() {
    let secret = 67;
    let mut success = false;

    println!("猜一個 1~100 的數字：");

    for _ in 0..5 {
        let mut input = String::new();
        std::io::stdin().read_line(&mut input).expect("讀取失敗");
        let guess = input.trim().parse::<i32>().expect("不是數字");

        if guess == secret {
            success = true;
            break;
        }

        println!("沒猜中.....");
    }

    if success {
        println!("恭喜你在五次內猜中了！");
    } else {
```

```
        println!("五次都沒猜中......");
    }
}
```

猜中的話：

```
猜一個 1~100 的數字：
50
沒猜中......
70
沒猜中......
67
恭喜你在五次內猜中了！
```

五次都沒猜中的話：

```
猜一個 1~100 的數字：
50
沒猜中......
75
沒猜中......
60
沒猜中......
80
沒猜中......
90
沒猜中......
五次都沒猜中......
```

這裡 `for _ in 0..5` 代表「最多猜五次」，我們不需要知道現在是第幾次，只需要讓迴圈跑五次就好。猜對了就把 `success` 設成 `true` 然後 `break` 跳出迴圈。

2.3.3 重點整理

- 變數沒用到時，Rust 編譯器會警告你。
- 在變數名前面加 `_`（像 `_x`）可以消除警告。
- 單獨的 `_` 代表「我完全不在乎這個值」。
- `_x` 還能用，`_` 不能用。
- 不需要迴圈變數的話，寫 `for _ in 0..5` 可以單純重複五次。

2.4 tuple

2.4.1 本集目標

用 tuple 把多個不同型別的值組合成一個，並學會怎麼取出裡面的值。

2.4.2 正文

到目前為止，我們一個變數只能存一個值。但如果我想把「一個整數、一個小數、一個布林值」綁在一起呢？這就是 **tuple**（元組）的用途。

2.4.2.1 建立 tuple

```
fn main() {
    let t = (1, 3.14, true);
    println!("{}", t.0); // 1
    println!("{}", t.1); // 3.14
    println!("{}", t.2); // true
}
```

用小括號 `()` 把值包起來，用逗號隔開，就是一個 tuple 了。

要取裡面的值，用**點加索引**：`t.0`、`t.1`、`t.2`。注意索引從 0 開始喔！

2.4.2.2 unit type — 空的 tuple

Rust 有一個特殊的 tuple，裡面什麼都沒有：

```
fn main() {
    let _u: () = ();
}
```

這個 `()` 叫做 **unit type**（單元型別）。它是「只有一個值的型別」。它的值也是寫成 `()`。

2.4.2.3 標型別

如果你想明確寫出 tuple 的型別：

```
fn main() {
    let t: (i32, f64, bool) = (1, 3.14, true);
    println!("{} {} {}", t.0, t.1, t.2);
}
```

每個位置的型別都要對應上。

2.4.2.4 單元素 tuple — 別忘了逗號！

如果你想建立一個只有一個元素的 tuple，要記得加逗號：

```
fn main() {
    let not_a_tuple = (5);      // 這只是數字 5，加了括號而已
    let a_tuple = (5,);         // 這才是 tuple！注意逗號

    println!("{}", a_tuple.0); // 5
}
```

`(5)` 只是一個被括號包住的數字，不是 tuple。`(5,)` 才是。那個逗號很重要！

型別也是一樣的寫法：

```
fn main() {
    let t: (i32,) = (5,);
    println!("{}", t.0);
}
```

`(i32)` 只是 `i32` 加了括號，`(i32,)` 才是單元素 tuple 的型別。

2.4.2.5 修改 tuple 裡面的值

如果 tuple 用 `let mut` 宣告，就可以修改裡面的值：

```rust
fn main() {
    let mut t = (1, 2, 3, 4);
    println!("修改前：{}", t.1); // 2
    t.1 = 99;
    println!("修改後：{}", t.1); // 99
}
```

跟其他 mut 變數一樣——沒有 mut 就不能改。

2.4.3 重點整理

- tuple 用 () 把不同型別的值打包在一起。
- 用 t.0、t.1、t.2……取值（索引從 0 開始）。
- () 是 unit type，代表「沒有有意義的值」。
- 單元素 tuple 要加逗號：(5,) 才是 tuple，(5) 只是數字。
- let mut 的 tuple 可以用 t.0 = 新值 修改裡面的值。

2.5 {:?} Debug 格式

2.5.1 本集目標

用 {:?} 印出 tuple 等「沒辦法用 {} 印」的東西。

2.5.2 正文

到目前為止，我們都用 {} 來印東西：

```rust
fn main() {
    let x = 42;
    println!("{}", x); // ✅ 42
}
```

數字、bool 這些基本型別用 {} 都沒問題。但如果你試著用 {} 印一個 tuple：

```rust
fn main() {
    let t = (1, 2, 3);
    println!("{}", t); // ❌ 編譯錯誤！
}
```

編譯器會吐出一堆錯誤訊息，簡單來說：「這個型別沒有實作 Display，我不知道要怎麼用『好看的方式』印出來。」

2.5.2.1 解決方法：用 {:?}

```rust
fn main() {
    let t = (1, 2, 3);
    println!("{:?}", t); // ✅ (1, 2, 3)
}
```

{:?} 叫做 **Debug 格式**。它不是給使用者看的「漂亮格式」，而是給開發者看的「偵錯格式」。

2.5.2.2 Display {} vs Debug {:?}

對數字、bool 這些簡單型別來說，{} 和 {:?} 印出來的結果差不多。那 {:?} 的重點在哪？

它能印出 {} 印不了的東西——像 tuple。 tuple 只有 Debug 格式，沒有 Display 格式。

2.5.2.3 美化版：{:#?}

如果資料很複雜（比如 tuple 套 tuple），可以用 {:#?} 印出「美化過的 Debug 格式」：

```
fn main() {
    let data = ((1, 2), (3, 4), (5, 6));
    println!("{:#?}", data);
}
```

2.5.2.4 小技巧：dbg! 巨集

Rust 還有一個很方便的偵錯工具 dbg!：

```
fn main() {
    let x = 5;
    dbg!(x);
    dbg!(x + 1);
}
```

它會印出檔名、第幾行第幾個字元，還有值，超方便：

```
[src/main.rs:3:5] x = 5
[src/main.rs:4:5] x + 1 = 6
```

2.5.3 重點整理

- {} 是 Display 格式，給使用者看的，但不是所有型別都支援。
- {:?} 是 Debug 格式，給開發者看的，tuple 等複合型別都能用。
- {:#?} 是美化版的 Debug 格式，複雜資料用這個更清楚。
- dbg! 是快速偵錯的好幫手，會印出檔名和位置（第幾行第幾個字元）。

2.6 簡單函數

2.6.1 本集目標

用 fn 定義自己的函數，並在 main 裡呼叫它。

2.6.2 正文

到目前為止，我們幾乎所有的程式碼都寫在 main 裡面。但如果程式越來越大，全部擠在一起就很亂。這時候我們可以把一段程式碼「包裝」成一個**函數**（function），想用的時候呼叫它就好。

2.6.2.1 定義一個函數

```
fn greet() {
    println!("你好！歡迎來到 Rust 的世界！");
}

fn main() {
    greet();
}
```

拆解一下語法：

- `fn` → 告訴 Rust「我要定義一個函數」。
- `greet` → 函數的名字。
- `()` → 參數列表（目前是空的，下一集會學）。
- `{ ... }` → 函數要做的事。

然後在 `main` 裡面寫 `greet();` 就是呼叫它。

2.6.2.2　函數可以呼叫好幾次

```
fn greet() {
    println!("哈囉！");
}

fn main() {
    greet();
    greet();
    greet();
}
```

這就是函數的好處——寫一次，用很多次。

2.6.2.3　函數之間可以互相呼叫

不只是 `main` 裡面可以呼叫函數——函數之間也可以互相呼叫。`main` 只是程式的**進入點**（程式開始執行的地方），但裡面呼叫的函數也能再呼叫其他函數：

```
fn say_name() {
    println!("我是 Rust！");
}

fn greet() {
    say_name();
}

fn main() {
    greet(); // main 呼叫 greet，greet 再呼叫 say_name
}
```

2.6.2.4　函數定義的位置：上面或下面都行

在某些語言裡，函數必須在使用之前先定義。但 Rust 不用！

```
fn main() {
    greet(); // ✓ 先呼叫
}

fn greet() { // 後定義
    println!("你好！");
}
```

這樣也完全沒問題。Rust 編譯器會先掃過整個檔案，所以不管你把函數放在 `main` 上面還是下面，都找得到。

2.6.2.5　函數命名慣例

Rust 的函數名用**蛇形命名法**（snake_case）：全小寫，單字之間用底線 _ 隔開。

```rust
fn say_hello() { // ✅ 蛇形命名
    println!("Hello!");
}

fn sayHello() {  // ⚠ 可以跑，但編譯器會警告
    println!("Hello!");
}
```

2.6.3 重點整理

- 用 `fn 名字() { ... }` 定義函數。
- 用 `名字();` 呼叫函數。
- `main` 是程式的進入點，但函數之間也可以互相呼叫。
- 函數定義放在 `main` 上面或下面都可以。
- 命名慣例是 snake_case（全小寫加底線）。

2.7 函數參數

2.7.1 本集目標

幫函數加上參數，讓它能接收外部傳進來的資料。

2.7.2 正文

上一集的 `greet` 函數每次都只能印一樣的東西，有點無聊。如果我們想讓函數更靈活——比如「你告訴我兩個數字，我幫你加起來」——就需要**參數**（parameter）。

2.7.2.1 加上參數

```rust
fn add(a: i32, b: i32) {
    println!("{} + {} = {}", a, b, a + b);
}

fn main() {
    add(3, 4);
    add(10, 20);
}
```

語法拆解：

- `a: i32` → 第一個參數叫 a，型別是 `i32`。
- `b: i32` → 第二個參數叫 b，型別也是 `i32`。
- 參數之間用逗號隔開。

呼叫的時候，`add(3, 4)` 就是把 3 傳給 a、4 傳給 b。

2.7.2.2 參數一定要標型別

在 Rust 裡，函數的參數**一定要標型別**，不能偷懶：

```rust
fn add_v1(a, b) {           // ✗ 編譯錯誤！沒標型別
    println!("{}", a + b);
}
```

```rust
fn add_v2(a: i32, b: i32) { // ✅ 一定要標
    println!("{}", a + b);
}
```

「可是 `let x = 5;` 不是可以不標嗎？」

沒錯，`let` 可以讓編譯器自己推斷。但函數參數一定要標型別——函數的定義必須清清楚楚，Rust 會依照這些型別檢查函數裡的程式碼。

2.7.2.3 多個參數、不同型別

參數可以有不同的型別：

```rust
fn describe(x: i32, is_positive: bool) {
    println!("{} 是正數嗎？{}", x, is_positive);
}

fn main() {
    describe(5, true);
    describe(-3, false);
}
```

2.7.2.4 一個參數也行

```rust
fn double(x: i32) {
    println!("{} 的兩倍是 {}", x, x * 2);
}

fn main() {
    double(5);
    double(100);
}
```

2.7.3 重點整理

- 函數參數寫在小括號裡：`fn 名字(參數: 型別)`。
- 多個參數用逗號隔開。
- **參數一定要標型別**，這是 Rust 的硬性規定。
- 呼叫時傳入對應的值就好。

2.8 函數回傳值

2.8.1 本集目標

讓函數回傳一個值，並學會 Rust 獨特的「不加分號就是回傳值」的寫法。

2.8.2 正文

上一集的函數只是把結果印出來。但很多時候我們想要的是：「你算完之後把答案**交回來**，我自己決定要怎麼用。」

2.8.2.1 基本語法

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}

fn main() {
    let result = add(3, 4);
    println!("3 + 4 = {}", result);
}
```

重點來了：

1. `-> i32` 寫在參數後面，告訴 Rust「這個函數會回傳一個 `i32`」。
2. 函數最後一行 `a + b` **沒有加分號** → 這就是回傳值。

2.8.2.2 不加分號 = 回傳值

這是 Rust 最獨特的設計之一。函數最後一行如果**不加分號**，它的值就會自動被當成回傳值：

```
fn double(x: i32) -> i32 {
    x * 2 // ✅ 沒有分號，這就是回傳值
}
```

2.8.2.3 加了分號會怎樣？

如果你不小心加了分號：

```
fn double(x: i32) -> i32 {
    x * 2; // ❌ 加了分號
}
```

編譯器會報錯。為什麼？因為加了分號之後，`x * 2` 的計算結果會被丟掉，而函數最後沒有留下任何有意義的值。在這種狀況下，實際回傳的是 `()`（unit type，還記得第 4 集嗎？）。但你答應了要回傳 `i32`，型別不符，編譯器就會抱怨。

2.8.2.4 沒寫回傳值的函數

回頭看本章第 6 集的 `greet` 函數，它沒有寫 `->` 回傳值：

```
fn greet() {
    println!("你好！");
}
```

在 Rust 裡，所有函數都有回傳值。沒寫 `->` 的話，就等同於寫 `-> ()`：

```
fn greet() -> () {
    println!("你好！");
}
```

只是 `-> ()` 通常省略不寫。`println!("你好！");` 最後有分號，計算結果被丟掉，函數回傳 `()`——剛好符合宣告。

2.8.2.5 接住回傳值

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

```rust
fn main() {
    let result = add(3, 4);
    println!("結果：{}", result);

    // 也可以直接用在表達式裡
    println!("再加 10：{}", add(3, 4) + 10);
}
```

2.8.2.6　用 tuple 回傳多個值

函數只能回傳「一個」值，但如果你想回傳多個呢？把它們裝在 tuple 裡就好：

```rust
fn swap(a: i32, b: i32) -> (i32, i32) {
    (b, a)
}

fn main() {
    let result = swap(1, 2);
    println!("第一個：{}，第二個：{}", result.0, result.1);
}
```

`-> (i32, i32)` 代表回傳一個包含兩個 `i32` 的 tuple。呼叫之後用 `.0`、`.1` 取出裡面的值。

再來一個實用的例子：

```rust
fn min_max(a: i32, b: i32) -> (i32, i32) {
    if a < b {
        (a, b)
    } else {
        (b, a)
    }
}

fn main() {
    let result = min_max(7, 3);
    println!("最小：{}，最大：{}", result.0, result.1);
}
```

2.8.3　重點整理

- 用 `->` 型別 宣告函數的回傳型別。
- 函數最後一行**不加分號**，就是回傳值（這是 Rust 的慣用寫法）。
- 加了分號就變成普通語句，回傳的會是 `()`。
- 沒寫回傳值的函數，其實回傳的是 `()`。
- 想回傳多個值？用 tuple 包起來：`-> (i32, i32)`，用 `.0`、`.1` 取值。

2.9　early `return`

2.9.1　本集目標

用 `return` 關鍵字在函數中途就把值回傳出去，不用等到最後一行。

2.9.2 正文

上一集我們學到：函數最後一行不加分號就是回傳值。但有時候你想在函數**中間**就回傳——遇到某個條件就提前結束。這時候就要用 `return` 關鍵字。

2.9.2.1 基本範例：絕對值

```
fn abs(x: i32) -> i32 {
    if x >= 0 {
        return x; // 如果 x 是正數或零，直接回傳
    }
    -x            // 走到這裡代表 x 是負數，回傳 -x
}

fn main() {
    println!("abs(5) = {}", abs(5));
    println!("abs(-3) = {}", abs(-3));
    println!("abs(0) = {}", abs(0));
}
```

注意看：

- `return x;` → 用 `return` 關鍵字。
- 最後一行 `-x` → 不加分號，這是「自然回傳」。

2.9.2.2 `return` vs 不加分號

兩種回傳方式的比較：

```
// 方式一：用 return（通常用在「提前離開」）
fn abs_v1(x: i32) -> i32 {
    if x >= 0 {
        return x;
    }
    -x
}

// 方式二：純粹用表達式（整個 if-else 就是回傳值）
fn abs_v2(x: i32) -> i32 {
    if x >= 0 {
        x
    } else {
        -x
    }
}
```

兩種都對！Rust 社群的慣例是：

- **有「我想要提前離開」的意思時用 `return`**（方式一）。
- **其他時候都用表達式**（方式二）。

2.9.2.3 實用場景：提前擋掉不合法的輸入

```
fn divide(a: f64, b: f64) -> f64 {
    if b == 0.0 {
        println!("錯誤：不能除以零！");
```

```
        return 0.0; // 提前離開
    }

    // 這邊可能又做了很多其他事情......

    a / b
}

fn main() {
    println!("{}", divide(10.0, 3.0));
    println!("{}", divide(10.0, 0.0));
}
```

這種「先檢查、不對就提前走人」的寫法叫做 **guard clause**（守衛子句），在實務中非常常見。

2.9.2.4　回傳 () 的 return

如果函數回傳 ()（例如沒有寫 -> 型別），return 後面不用寫值：

```
fn check_age(age: i32) {
    if age < 0 {
        println!("年齡不能是負數！");
        return; // 等同於 return ();
    }
    println!("你的年齡是 {}", age);
}

fn main() {
    check_age(25);
    check_age(-3);
}
```

return; 是 return (); 的簡寫——因為回傳的是 ()（unit type），省略不寫更簡潔。

2.9.2.5　不要到處用 return

雖然每個回傳值都寫 return 也能跑，但在 Rust 裡這不是好習慣：

```
// 不太 Rust 的寫法
fn add_v1(a: i32, b: i32) -> i32 {
    return a + b; // 可以跑，但沒必要
}

// Rust 慣用寫法
fn add_v2(a: i32, b: i32) -> i32 {
    a + b         // 最後一行直接當回傳值
}
```

return 留給「提前離開」的場景就好。

2.9.3　重點整理

- return 值; 可以在函數中途提前回傳。
- 最後一行不加分號的自然回傳是 Rust 的慣用寫法。
- return 最常用在 guard clause：先檢查條件，不對就提前走人。
- 回傳 () 的函數裡，return; 是 return (); 的簡寫。

- 不要每個回傳值都寫 `return`，只在需要提前離開時才用。

2.10 遞迴

2.10.1 本集目標

讓函數呼叫自己來解決問題，這個技巧叫做「遞迴」。

2.10.2 正文

你有沒有想過：函數可以在自己裡面呼叫自己嗎？

答案是**可以的**，而且這個技巧叫做**遞迴**（recursion）。聽起來很玄，但其實概念很簡單。

2.10.2.1 經典範例：階乘

「5 的階乘」寫成 `5!`，意思是 `5 × 4 × 3 × 2 × 1 = 120`。

用遞迴的思路想：

- `5! = 5 × 4!`
- `4! = 4 × 3!`
- `3! = 3 × 2!`
- `2! = 2 × 1!`
- `1! = 1`（到這裡停下來）

看到了嗎？每一步都是「自己乘以比自己小一號的階乘」，最後到 1 就停。

```rust
fn factorial(n: u32) -> u32 {
    if n <= 1 {
        1
    } else {
        n * factorial(n - 1)
    }
}

fn main() {
    println!("5! = {}", factorial(5));
    println!("3! = {}", factorial(3));
    println!("1! = {}", factorial(1));
}
```

2.10.2.2 遞迴的兩個關鍵

每個遞迴函數都需要兩樣東西：

1. base case（基底情況）：什麼時候停下來

```rust
if n <= 1 {
    1 // 停！不再呼叫自己
}
```

2. recursive case（遞迴情況）：怎麼把問題縮小

```rust
n * factorial(n - 1) // 把問題縮小：n 變成 n - 1
```

如果忘記寫 base case，函數就會無限呼叫自己，最後程式就炸了。

2.10.2.3　追蹤執行過程

讓我們追蹤 `factorial(5)` 的執行過程：

```
factorial(5)
= 5 * factorial(4)
= 5 * (4 * factorial(3))
= 5 * (4 * (3 * factorial(2)))
= 5 * (4 * (3 * (2 * factorial(1))))
= 5 * (4 * (3 * (2 * 1)))
= 5 * (4 * (3 * 2))
= 5 * (4 * 6)
= 5 * 24
= 120
```

就像俄羅斯套娃一樣，一層一層展開，到底之後再一層一層收回來。

2.10.2.4　另一個例子：倒數

```rust
fn countdown(n: u32) {
    if n == 0 {
        println!("發射！🚀");
        return;
    }
    println!("{}...", n);
    countdown(n - 1);
}

fn main() {
    countdown(5);
}
```

2.10.2.5　遞迴 vs 迴圈

其實上面的例子都可以用迴圈寫。那什麼時候用遞迴？什麼時候用迴圈？

- **簡單的重複** → 迴圈比較直覺。
- **問題本身就是遞迴結構** → 遞迴比較自然。

現在先知道遞迴怎麼寫就好，之後遇到適合的場景自然會用到。

2.10.3　重點整理

- 遞迴就是函數呼叫自己。
- 一定要有 **base case**（停止條件），不然會無限遞迴。
- 每次呼叫都要讓問題**變小**，往 base case 靠近。

2.11　陣列基礎

2.11.1　本集目標

用陣列（array）把多個相同型別的值排成一列，並學會怎麼存取和建立。

2.11.2 正文

之前學了 tuple 可以把不同型別的值打包在一起。今天來認識另一個好朋友——**陣列**（array）。陣列是「把一堆**相同型別**的值排成一列」。

2.11.2.1 建立陣列

```
fn main() {
    let arr = [1, 2, 3, 4, 5];
    println!("{:?}", arr);
}
```

這裡用了 `{:?}`（Debug 格式）來印陣列——還記得本章第 5 集嗎？陣列和 tuple 一樣，只有 Debug 格式，不能用 `{}`。

用中括號 `[]` 包起來，逗號隔開。注意：陣列裡的值**必須是同一個型別**。

```
    let arr = [1, "hello", 3.14]; // ✗ 不行！型別不同
```

想混不同型別？用上幾集學的 tuple。

2.11.2.2 用索引取值

```
fn main() {
    let arr = [1, 2, 3, 4, 5];
    println!("第一個：{}", arr[0]);
    println!("第三個：{}", arr[2]);
    println!("最後一個：{}", arr[4]);
}
```

重點：**索引從 0 開始**！所以 5 個元素的索引是 0、1、2、3、4。

2.11.2.3 越界會 panic

如果你存取一個不存在的索引：

```
fn main() {
    let arr = [1, 2, 3, 4, 5];
    println!("{}", arr[10]); // 💥 index out of bounds!
}
```

程式會直接**崩潰**（panic），印出錯誤。Rust 不會讓你偷偷讀到不該讀的記憶體。比起默默給你一個垃圾值，直接崩潰反而更安全——至少你馬上知道哪裡出錯了。

2.11.2.4 陣列的型別

陣列的型別寫法是 `[元素型別; 長度]`：

```
fn main() {
    let arr: [i32; 5] = [1, 2, 3, 4, 5];
    println!("{:?}", arr);
}
```

`[i32; 5]` 代表「一個放 5 個 `i32` 的陣列」。注意，**長度也是型別的一部分**——`[i32; 3]` 和 `[i32; 5]` 是不同的型別！

大部分時候 Rust 可以自動推斷，不用手動標。但知道怎麼寫型別在之後會很有用。

2.11.2.5 快速建立：重複語法

如果你想建立一個「5 個 0」的陣列：

```
fn main() {
    let zeros = [0; 5];
    println!("{:?}", zeros);
}
```

`[0; 5]` 的意思是「值 0，重複 5 次」。分號前面是值，後面是個數。

再來幾個例子：

```
fn main() {
    let ones = [1; 10];    // 10 個 1
    let flags = [true; 3]; // 3 個 true
    println!("{:?}", ones);
    println!("{:?}", flags);
}
```

2.11.3 重點整理

- 陣列用 `[值1, 值2, ...]` 建立，所有元素必須同型別。
- 索引從 0 開始，用 `arr[0]` 取值。
- 存取超出範圍的索引會 panic（程式崩潰）。
- 陣列型別寫法是 `[型別; 長度]`，如 `[i32; 5]`（長度也是型別的一部分）。
- `[值; 個數]` 可以快速建立重複的陣列，如 `[0; 5]`。
- 用 `{:?}` 印整個陣列。

2.12 陣列走訪

2.12.1 本集目標

用 `for` 迴圈走過陣列裡的每一個元素。

2.12.2 正文

上一集我們學了怎麼用 `arr[0]`、`arr[1]` 一個一個取值。但如果陣列有 100 個元素，總不能寫 100 行吧？這時候就要用 `for` 迴圈來**走訪**（iterate）整個陣列。

2.12.2.1 基本語法

```
fn main() {
    let arr = [1, 2, 3, 4, 5];

    for x in arr {
        println!("{}", x);
    }
}
```

`for x in arr` 的意思是：「把 `arr` 裡的元素一個一個拿出來，每次放進 `x`，然後執行大括號裡的程式碼。」

2.12.2.2 幫元素做運算

```
fn main() {
    let scores = [80, 95, 72, 88, 100];

    for score in scores {
        if score >= 90 {
            println!("{} 分 → 優秀！", score);
        } else {
            println!("{} 分 → 加油！", score);
        }
    }
}
```

2.12.2.3 加總所有元素

```
fn main() {
    let arr = [1, 2, 3, 4, 5];
    let mut total = 0;

    for x in arr {
        total += x;
    }

    println!("總和：{}", total);
}
```

先用 `let mut total = 0;` 建立一個可變的累加器，每次迴圈把值加上去。

2.12.2.4 `for in` range vs `for in` 陣列

第 1 章學的 `for i in 0..5` 是走訪一個**數字範圍**。這集的 `for x in arr` 是走訪一個**陣列**。語法一樣，只是 in 後面放的東西不同：

```
fn main() {
    let arr = [10, 20, 30];

    // 走訪數字範圍：i 依序是 0, 1, 2
    for i in 0..3 {
        println!("索引 {}：{}", i, arr[i]);
    }

    // 走訪陣列：x 依序是 10, 20, 30
    for x in arr {
        println!("值：{}", x);
    }
}
```

走訪陣列時用 `for x in arr` 比用索引更簡潔、更安全、也可能更快速——不用擔心索引越界。需要同時拿到索引和值的時候，之後會學到更好的方式。

2.12.3 重點整理

- `for x in arr { ... }` 走訪陣列的每個元素。
- 可以在迴圈裡對每個元素做運算、判斷、累加。

- `for x in arr`（走訪陣列）和 `for i in 0..n`（走訪範圍）語法一樣，差在 `in` 後面的東西。
- 走訪陣列時用 `for x in arr` 比用索引更簡潔、更安全、也可能更快速。

2.13 切片 &[T]

2.13.1 本集目標

用切片（slice）取出陣列的一部分，像透過窗戶看裡面的東西。

2.13.2 正文

有時候你不需要整個陣列，只想看其中一段。比如一個有 5 個元素的陣列，你只想看第 2 到第 4 個。這時候就可以用**切片**（slice）。

2.13.2.1 基本語法

```
fn main() {
    let arr = [1, 2, 3, 4, 5];
    let slice = &arr[1..4];
    println!("{:?}", slice);
}
```

和陣列、tuple 一樣，切片也只能用 `{:?}` 來印，不能用 `{}`。

`&arr[1..4]` 的意思是：「從索引 1 開始，到索引 4 **之前**為止。」

- 索引 1 → 值 2（包含）。
- 索引 2 → 值 3（包含）。
- 索引 3 → 值 4（包含）。
- 索引 4 → 值 5（**不包含**）。

所以結果是 `[2, 3, 4]`。

2.13.2.2 範圍的寫法

```
fn main() {
    let arr = [1, 2, 3, 4, 5];

    let a = &arr[0..3];  // [1, 2, 3]       從 0 到 3（不含 3）
    let b = &arr[0..=2]; // [1, 2, 3]       從 0 到 2（包含 2）
    let c = &arr[2..];   // [3, 4, 5]       從 2 到最後
    let d = &arr[..3];   // [1, 2, 3]       從頭到 3（不含 3）
    let e = &arr[..];    // [1, 2, 3, 4, 5] 整個陣列

    println!("{:?}", a);
    println!("{:?}", b);
    println!("{:?}", c);
    println!("{:?}", d);
    println!("{:?}", e);
}
```

- `1..4` → 從 1 到 4（不含 4）。
- `1..=3` → 從 1 到 3（**包含** 3），還記得第 1 章第 20 集的 `..=` 嗎？一樣的用法。
- `2..` → 從 2 到結尾。

- `..3` → 從開頭到 3（不含 3）。
- `..` → 整個。

2.13.2.3 切片是「視窗」，不是「複製」

這裡有個重要的觀念：切片**不是**把資料複製一份出來，而是「指向原本陣列的某一段」。就像透過窗戶看房間裡的東西——東西還是在房間裡，你只是從窗戶看進去。

```
fn main() {
    let arr = [10, 20, 30, 40, 50];
    let slice = &arr[1..4];

    println!("陣列：{:?}", arr);
    println!("切片：{:?}", slice);
}
```

2.13.2.4 那個 & 是什麼？

你可能注意到切片前面有個 &。這個符號代表「借用」（borrow），是 Rust 最重要的概念之一。但現在不用深入理解——先記住「切片要加 &」就好，之後我們會詳細解釋。

現在你只需要知道：寫切片的時候前面要加 &。

2.13.2.5 切片的型別

還記得陣列的型別是 `[i32; 5]`（型別包含長度）嗎？切片的型別是 `&[i32]`——**沒有長度**：

```
fn main() {
    let arr: [i32; 5] = [1, 2, 3, 4, 5];
    let slice: &[i32] = &arr[1..4];
    println!("{:?}", slice);
}
```

`&[i32]` 代表「一段 i32 的切片」，不管長度是多少。這是切片和陣列最大的不同——陣列的長度是型別的一部分（`[i32; 3]` 和 `[i32; 5]` 是不同型別），但切片不管長度，`&[i32]` 可以指向任意長度的連續區段。

2.13.2.6 走訪切片

切片也可以用 `for` 走訪：

```
fn main() {
    let arr = [1, 2, 3, 4, 5];
    let slice = &arr[1..4];

    for x in slice {
        println!("{}", x);
    }
}
```

2.13.2.7 複合型別

學完切片之後，讓我們整理一下：到目前為止，我們學過兩類型別。

基本型別（primitive types）： `i32`、`f64`、`bool`、`char` 等等，每個值就是一個單獨的東西。

複合型別（compound types）：把其他型別組合在一起的型別。我們已經學了三種：

- **tuple**：`(i32, f64, bool)` — 可以裝不同型別。
- **陣列**：`[i32; 5]` — 同一種型別，固定長度。
- **切片**：`&[i32]` — 同一種型別，不限長度。

複合型別裡面的型別不一定要是基本型別，也能是一層複合型別套另一層複合型別：

```
fn main() {
    // 陣列裡面裝 tuple
    let pairs: [(i32, bool); 3] = [(1, true), (2, false), (3, true)];
    println!("{:?}", pairs);

    // tuple 裡面裝陣列
    let t: ([i32; 3], [i32; 3]) = ([1, 2, 3], [4, 5, 6]);
    println!("{:?}", t);

    // 陣列裡面裝陣列
    let grid: [[i32; 2]; 3] = [[1, 2], [3, 4], [5, 6]];
    println!("{:?}", grid);

    // 切片也是複合型別
    let arr: [i32; 5] = [10, 20, 30, 40, 50];
    let slice: &[i32] = &arr[1..4];
    println!("{:?}", slice);

    // tuple 裡面裝切片
    let pair: (&[i32], &[i32]) = (&arr[..2], &arr[3..]);
    println!("{:?}", pair);
}
```

2.13.3　重點整理

- 切片用 `&arr[start..end]` 或 `&arr[start..=end]` 取出陣列的一部分。
- `start..end` 是「包含 start、不包含 end」。
- `start..=end` 是「包含 start 也包含 end」。
- 切片是陣列的「視窗」，不是複製。
- 切片型別是 `&[i32]`（不含長度），陣列型別是 `[i32; 5]`（含長度）。
- 前面的 `&` 代表借用，之後會詳細解釋。
- 切片也可以用 `for` 走訪。
- tuple、陣列、切片都是**複合型別**——它們裡面可以裝其他型別，包含複合型別。

2.14　切片作為參數

2.14.1　本集目標

用切片 `&[i32]` 當函數參數，這樣不管陣列多長都能傳進去。

2.14.2　正文

上一集學了切片，這集來看一個超實用的應用：**把切片當作函數的參數**。

2.14.2.1　先看問題

假設你想寫一個函數來計算陣列的總和。如果用陣列當參數：

```
fn sum(nums: [i32; 5]) -> i32 {
    let mut total = 0;
    for x in nums {
        total += x;
    }
    total
}

fn main() {
    let a = [1, 2, 3, 4, 5];
    println!("{}", sum(a)); // ✅ 可以

    let b = [1, 2, 3];
    println!("{}", sum(b)); // ❌ 不行！b 有 3 個元素，但函數要 5 個
}
```

問題出在 `[i32; 5]`——你把長度寫死成 5 了。3 個元素的陣列就傳不進去。

2.14.2.2 解決方案：用切片

```
fn sum(nums: &[i32]) -> i32 {
    let mut total = 0;
    for x in nums {
        total += x;
    }
    total
}

fn main() {
    let a = [1, 2, 3, 4, 5];
    let b = [10, 20, 30];
    let c = [7];

    println!("a 的總和：{}", sum(&a)); // 15
    println!("b 的總和：{}", sum(&b)); // 60
    println!("c 的總和：{}", sum(&c)); // 7
}
```

把參數型別從 `[i32; 5]` 改成 `&[i32]`，就能接受**任何長度**的切片了！

呼叫的時候要加 `&`：`sum(&a)` 表示「把 a 的切片傳進去」。

2.14.2.3 也可以傳切片的一部分

因為參數是 `&[i32]`，你不只能傳整個陣列，也能傳一段切片：

```
fn sum(nums: &[i32]) -> i32 {
    let mut total = 0;
    for x in nums {
        total += x;
    }
    total
}

fn main() {
    let arr = [1, 2, 3, 4, 5];
```

```
    println!("全部：{}", sum(&arr));        // 15
    println!("前三個：{}", sum(&arr[..3])); // 6
    println!("後三個：{}", sum(&arr[2..])); // 12
}
```

這就是切片的威力——一個函數，各種用法。

2.14.2.4 為什麼切片比固定長度陣列好？

固定長度 [i32; 5]	切片 &[i32]
只能接受剛好 5 個元素	任何長度都行
換長度要重寫函數	一個函數通吃

在實務中，幾乎所有接受陣列的函數都用切片當參數。

2.14.3 重點整理

- 函數參數用 &[i32] 而不是 [i32; 5]，就能接受任意長度。
- 呼叫時傳 &arr 或 &arr[1..4] 都行。
- 切片參數讓函數更靈活、更通用。
- 這是 Rust 實務中最常見的寫法。

2.15 字串切片 &str

2.15.1 本集目標

認識 &str 這個型別，原來我們一直在用的字串就是切片！

2.15.2 正文

前幾集我們學了陣列的切片 &[i32]。今天來認識另一種切片——**字串切片**。其實我們之前寫的 "hello" 就是字串切片。

2.15.2.1 字串的真面目

```
fn main() {
    let s = "hello";
    println!("{}", s);
}
```

這樣的程式碼你已經看了無數次了。但 s 的型別是什麼？

答案是：**&str**（字串切片）。

```
fn main() {
    let s: &str = "hello"; // 明確標出型別
    println!("{}", s);
}
```

&str 唸作「string slice」。它就像陣列切片 &[i32] 一樣，是「指向一段資料的視窗」。

2.15.2.2 和陣列切片的對比

陣列切片	字串切片
&[i32]	&str
指向一段 i32 資料	指向一段文字資料
let s = &arr[1..4];	let s = "hello";

概念完全一樣！只是一個是數字的切片，一個是文字的切片。

2.15.2.3 字串切片也可以取子字串

```
fn main() {
    let s = "hello world";
    let hello = &s[0..5];
    let world = &s[6..11];
    println!("{}", hello); // hello
    println!("{}", world); // world
}
```

&s[0..5] 就是取 s 的前 5 個 bytes（注意是 bytes，不是字元）。

和陣列切片一樣，也可以用 ..= 來包含結尾：

```
fn main() {
    let s = "hello world";
    let hello = &s[0..=4]; // 包含索引 4，等同於 &s[0..5]
    println!("{}", hello); // hello
}
```

2.15.2.4 ⚠ 中文字串切片要小心！

英文字母一個字佔 1 個 byte，但中文字通常佔 **3 個 bytes**。如果你切的位置剛好在一個中文字的「中間」，程式會直接崩潰：

```
fn main() {
    let s = "你好";
    let first = &s[0..3]; // ✅ "你"（剛好 3 個 bytes）
    println!("{}", first);
}
```

但如果你試著切 &s[0..1]：

```
fn main() {
    let s = "你好";
    let oops = &s[0..1];  // ✗ 程式崩潰！
    println!("{}", oops);
}
```

因為「你」佔了 3 個 bytes（索引 0、1、2），你切到索引 1 是這個字的「中間」，Rust 不允許這樣做。

簡單來說：對英文字串做切片很安全，但對中文字串做切片時，要確保切的位置剛好在字元的邊界上。如果不確定，先不要對中文字串用 &s[start..end]。

2.15.2.5 函數參數用 &str

現在你知道字串是 &str 了，就可以把它當函數參數：

```rust
fn greet(name: &str) {
    println!("嗨，{}！", name);
}

fn main() {
    greet("Andy");
    greet("小明");
}
```

"Andy" 本身就是 &str 型別，所以直接傳進去就行。

2.15.3 重點整理

- "hello" 的型別是 &str，就是字串切片。
- &str 和陣列切片 &[i32] 的概念一樣——都是「指向一段資料的視窗」。
- 函數參數寫 &str 就能接受字串。
- 也可以用 &s[start..end] 取子字串，但要小心：索引是 byte 位置，不是字元位置，切在中文等多 byte 字元的中間會 panic。

恭喜你完成了第 2 章！🎉 這一章我們學到了更多組織程式的方法——函數、陣列、切片，還有各種讓程式碼更清晰的技巧。下一章我們將開始自訂型別，用 struct 和 enum 來描述你自己的資料！

第 3 章

Struct、Enum 與 Pattern Matching

本章會教你如何寫出自己的型別，並對原有的型別和自己創造的型別進行拆解分析。雖然這章教的知識可能沒有辦法讓你寫出更複雜的演算法，卻會讓你在進行真正軟體工程的路上踏出第一步。你創造的型別儘管是由其他已存在的簡單型別所組成，卻可能被賦予邏輯上更複雜的意義。當上一個型別又接著成為下一個型別的一部分，我們便得以管理更複雜的邏輯，更精準地為現實或人們的想像建模。

3.1 `struct`（named fields）

3.1.1 本集目標

學會用 `struct` 把多個相關的值組合在一起，形成一個自訂型別。

3.1.2 概念說明

到目前為止，我們用過的型別都是 Rust 內建的：`i32`、`f64`、`bool`、`char`，還有 tuple、陣列和切片。但實際寫程式的時候，你會需要**自己定義新的型別**。

`struct` 就是 Rust 讓你定義新型別的方式之一。定義一個 `struct`，就是在告訴 Rust：「我要一個新的型別，它裡面包含這些欄位。」

比如說，一個「點」有 x 座標和 y 座標。我們可以用 tuple `(i32, i32)` 來表示，但 tuple 只能用 `.0`、`.1` 取值，看不出哪個是 x、哪個是 y。用 `struct` 就能幫每個欄位取名字。

定義 `struct` 的語法是：

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {}
```

`struct` 定義一般放在 `fn main()` 外面，這樣其他函數也能用到。放在上面或下面都可以（和函數一樣，不受定義順序限制）。

建立一個 `struct` 的值時，要用 `型別名 { 欄位名: 值 }` 的寫法。取值的時候用 `.欄位名`。如果要修改 `struct` 的欄位，變數必須加 `mut`。

3.1.3 範例程式碼

```
struct Point {
    x: i32,
    y: i32,
}
```

```rust
fn main() {
    let p = Point { x: 3, y: 7 };
    println!("x 座標是 {}", p.x);
    println!("y 座標是 {}", p.y);

    // Point 是一個型別，就像 i32 一樣，可以用在型別標注上
    let p2: Point = Point { x: 100, y: 200 };
    println!("p2 的座標是 ({}, {})", p2.x, p2.y);

    // 也可以用 mut 讓 struct 的值可以修改
    let mut q = Point { x: 0, y: 0 };
    q.x = 10;
    q.y = 20;
    println!("q 的座標是 ({}, {})", q.x, q.y);
}
```

3.1.4 補充：trailing comma（結尾逗號）

注意 `struct` 定義裡，最後一個欄位後面也有逗號：

```rust
struct Point {
    x: i32,
    y: i32, // ← 這個逗號可加可不加
}

fn main() {}
```

Rust 允許在 `struct` 定義、`struct` 建立、函數呼叫等地方的最後一個項目後面加逗號。這叫做 **trailing comma**（結尾逗號）。加了不會錯，而且好處是之後新增欄位時，不用回去幫上一行補逗號，git diff 也比較乾淨。

Rust 社群慣例是**加上 trailing comma**。

3.1.5 重點整理

- `struct` 讓你定義一個有名字欄位的自訂型別。
- `struct` 定義一般放在 `fn main()` 外面，上面或下面都可以（和函數一樣）。
- 建立 `struct` 值的語法：`Point { x: 1, y: 2 }`。
- 用 `.欄位名` 取得欄位的值，例如 `p.x`。
- 如果要修改 `struct` 的欄位，變數必須加 `mut`。
- 最後一個欄位後面的逗號（trailing comma）可加可不加，慣例是加。

3.2 tuple `struct` 與 unit `struct`

3.2.1 本集目標

學會用 tuple `struct` 定義沒有欄位名的 `struct`，以及完全沒有欄位的 unit `struct`。

3.2.2 概念說明

上一集學的 `struct` 每個欄位都有名字。但有時候，欄位的意義已經很明顯了，不需要特別取名。這時候可以用 **tuple `struct`**——它長得像 tuple 和 `struct` 的混合體。

```
struct Point(i32, i32);

fn main() {}
```

建立值的時候用 `Point(3, 7)`——注意，這裡的 `Point` 既是**型別的名字**，也是**建立值時使用的名字**。取值用 `.0`、`.1`，就像 tuple 一樣。

上一集的 named-field `struct` 也是同樣的道理：`Point` 既是型別名，也是建立值時寫 `Point { x: 1, y: 2 }` 用的名字。

另外還有一種更極端的情況：`struct` 完全沒有欄位，叫做 **unit `struct`**。它通常用來當作一個「標記」，表示某種身份或角色，但本身不帶任何資料。

```
struct Marker;

fn main() {}
```

3.2.3 範例程式碼

```
// tuple struct：欄位沒有名字，用位置存取
struct Point(i32, i32);

// 另一個 tuple struct 的例子
struct Color(i32, i32, i32);

// unit struct：完全沒有欄位
struct Marker;

fn main() {
    let p: Point = Point(3, 7);
    println!("x = {}, y = {}", p.0, p.1);

    let red: Color = Color(255, 0, 0);
    println!("R={}, G={}, B={}", red.0, red.1, red.2);

    // unit struct 建立時不需要括號或大括號
    let _m: Marker = Marker;
    println!("Marker 被建立了！（它不帶任何資料）");
}
```

3.2.4 重點整理

- **tuple `struct`**：`struct Point(i32, i32);`，用 `.0`、`.1` 取值。
- **unit `struct`**：`struct Marker;`，沒有任何欄位。
- tuple `struct` 適合欄位意義明顯、不需要命名的情況。
- unit `struct` 適合當標記用，本身不攜帶資料。
- 即使兩個 tuple `struct` 的欄位型別完全一樣，它們也是不同的型別（例如 `Point(i32, i32)` 和 `Size(i32, i32)` 不能互換）。

3.3 enum（C-style）

3.3.1 本集目標

學會用 enum 定義一組固定的選項，讓變數只能是其中一個值。

3.3.2 概念說明

上一集我們學了 struct——用來定義「把多個值組合在一起」的新型別。這一集要學另一種定義新型別的方式：enum。

有時候我們想表達「這個東西只能是幾個選項之一」。比如說，一個交通燈只能是紅、黃、綠其中一種。

enum（enumeration，列舉）就是用來定義這種「多選一」的型別。和 struct 一樣，定義一個 enum 就是在告訴 Rust：「我要一個新的型別，它的值只能是這幾個選項之一。」最簡單的 enum 長這樣：

```
enum Color {
    Red,
    Green,
    Blue,
}

fn main() {}
```

每一個選項叫做一個 **variant**（變體）。建立 enum 值的時候，要用 型別名::變體名 的寫法：

```
enum Color {
    Red,
    Green,
    Blue,
}

fn main() {
    let c = Color::Red;
}
```

注意中間是兩個冒號 ::，這在 Rust 裡叫做「路徑運算子」，表示「Color 底下的 Red」。

這種最基本的 enum——每個 variant 都不帶任何額外資料——有時候稱為 C-style enum，因為 C 語言的 enum 就是這樣。

3.3.3 範例程式碼

```
enum Direction {
    Up,
    Down,
    Left,
    Right,
}

fn main() {
    let _dir = Direction::Up;

    // 目前我們還不會用 enum 做太多事
```

```
    // 下一集會學 match，就能根據 enum 的值做不同的事情
    // 這邊先展示怎麼建立不同的 enum 值

    let _d1 = Direction::Down;
    let _d2 = Direction::Left;
    let _d3 = Direction::Right;

    println!("方向已經設定好了！");
    println!("（下一集，match 會讓我們根據方向行動）");
}
```

3.3.4 重點整理

- enum 和 struct 一樣，都是定義新型別的方式。
- struct：把多個值組合在一起；enum：從多個選項中選一個。
- 用 :: 來指定是哪一個 variant，例如 Direction::Up。
- C-style enum 的每個 variant 都不帶額外資料。
- 和 struct 一樣，enum 定義一般放在 fn main() 外面，上面或下面都可以。
- 目前還無法直接印出 enum 的值（下一集學 match 就可以了）。

3.4 match C-style enum

3.4.1 本集目標

學會用 match 來根據 enum 的值執行不同的程式碼，並理解「窮舉」的概念。

3.4.2 概念說明

上一集我們定義了 enum，但沒辦法根據它的值做不同的事。現在來學 match——Rust 最強大的**模式匹配**（pattern matching）工具。

match 的基本語法是：

```
match 變數 {
    模式1 => 做某件事,
    模式2 => 做另一件事,
    模式3 => 做第三件事,
}
```

每一行叫做一個「分支」（arm）。Rust 會從上到下檢查，找到第一個符合的模式就執行對應的程式碼。

最重要的規則：match 必須窮舉所有可能的值。如果你的 enum 有三個 variant，你就必須處理全部三個。少寫一個，編譯器就會報錯。這是 Rust 幫你抓 bug 的方式——確保你不會忘記處理某種情況。

和 struct 還有 enum 一樣，match 的最後一個分支後面也可以加 trailing comma（結尾逗號）。Rust 社群慣例是加上的。

3.4.3 範例程式碼

```
enum Color {
    Red,
    Green,
    Blue,
}

fn main() {
    let c = Color::Green;

    match c {
        Color::Red => println!("紅色"),
        Color::Green => println!("綠色"),
        Color::Blue => println!("藍色"),
    }

    // 再來一個例子
    let light = Color::Red;

    match light {
        Color::Red => println!("停下來！"),
        Color::Green => println!("可以走了！"),
        Color::Blue => println!("這個交通燈有點奇怪..."),
    }
}
```

3.4.4 重點整理

- match 會根據值比對不同的模式，執行對應的分支。
- 每個分支用 => 分隔模式和要執行的程式碼。
- **match 必須窮舉所有 variant**——少一個就編譯失敗。
- 分支從上到下比對，第一個符合的就會執行。
- match 是 Rust 處理 enum 最基本的方式。

3.5 match 當表達式

3.5.1 本集目標

學會把 match 當作表達式使用，讓它回傳一個值。

3.5.2 概念說明

還記得第 1 章學過 if 可以當表達式嗎？

```
fn main() {
    let condition = true;
    let x = if condition { 1 } else { 2 };
}
```

match 也可以！你可以把整個 match 放在 let 的右邊，讓每個分支回傳一個值：

```
enum Color {
    Red,
    Green,
    Blue,
}

fn main() {
    let c = Color::Red;
    let msg = match c {
        Color::Red => "紅色",
        Color::Green => "綠色",
        Color::Blue => "藍色",
    };
}
```

注意最後面有一個分號 ;，因為整個 `let msg = match ... { ... };` 是一個 let 陳述式。

每個分支回傳的值的型別必須一致。如果第一個分支回傳 &str，其他分支也都要回傳 &str。

3.5.3 範例程式碼

```
enum Season {
    Spring,
    Summer,
    Autumn,
    Winter,
}

fn main() {
    // match 當表達式，回傳 &str
    let season = Season::Autumn;
    let name = match season {
        Season::Spring => "春天",
        Season::Summer => "夏天",
        Season::Autumn => "秋天",
        Season::Winter => "冬天",
    };
    println!("現在是{}", name);

    // 另一個例子：match 回傳 i32
    let weather = Season::Summer;
    let temp = match weather {
        Season::Spring => 22,
        Season::Summer => 35,
        Season::Autumn => 18,
        Season::Winter => 8,
    };
    println!("大約 {} 度", temp);
}
```

3.5.4 重點整理

- match 可以當表達式，整個 match 會回傳一個值。
- 用法：`let x = match ... { ... };`（最後別忘了分號）。

- 所有分支的回傳值型別必須一致。
- 這和 `if` 表達式的概念相同——Rust 裡很多東西都可以是表達式。

3.6 block 表達式

3.6.1 本集目標

學會用大括號 {} 建立 block 表達式，在裡面執行多行程式碼後回傳一個值。

3.6.2 概念說明

在 Rust 裡，一對大括號 {} 不只是作用域，它本身也是一個**表達式**，可以回傳值。規則很簡單：block 裡面最後一行如果不加分號，那一行的值就是整個 block 的回傳值。

```
fn main() {
    let x = {
        let y = 5;
        y + 1 // 沒有分號 → 這就是 block 的回傳值
    };
    // x 現在是 6
}
```

這個概念在 `match` 裡特別有用。之前的 `match` 分支都只有一行，但如果你想在某個分支裡做多件事，就可以用 block：

```
    match c {
        Color::Red => {
            println!("是紅色！");
            "red"
        }
        // ...
    }
```

block 裡可以宣告變數、做計算，最後一行不加分號就是回傳值。

注意：如果 `match` 的分支用了 block {}，後面的逗號可以省略。因為 } 本身就是明確的結束標記，Rust 不需要逗號來分隔。但如果分支只有一行（沒有用 block），後面的逗號就不能省——唯一的例外是整個 `match` 的最後一個分支，它是結尾，後面沒有東西需要分隔，所以逗號可加可不加。

```
    match s {
        Season::Summer => {
            println!("好熱啊！");
            "炎熱的夏天"
        } // ← 沒有逗號，OK
        Season::Autumn => "涼爽的秋天", // ← 一行的分支，要逗號
        // ...
    }
```

3.6.3 範例程式碼

```
enum Season {
    Spring,
    Summer,
    Autumn,
```

```
    Winter,
}

fn main() {
    // block 表達式的基本用法
    let result = {
        let a = 10;
        let b = 20;
        a + b // 最後一行不加分號 → 回傳值
    };
    println!("result = {}", result);

    // 在 match 分支裡使用 block
    let s = Season::Summer;

    let description = match s {
        Season::Spring => {
            let temp = 22;
            println!("春暖花開");
            if temp > 20 {
                "溫暖的春天"
            } else {
                "還有點涼的春天"
            }
        }
        Season::Summer => {
            println!("好熱啊！");
            "炎熱的夏天"
        }
        Season::Autumn => "涼爽的秋天",
        Season::Winter => "寒冷的冬天",
    };
    println!("{}", description);
}
```

3.6.4 重點整理

- {} block 本身是一個表達式，最後一行不加分號就是回傳值。
- `let x = { ... };` 可以在 block 裡做多行計算後把結果賦值給 x。
- match 分支可以用 `=> { ... }` 來執行多行程式碼，而且 block 後面不用加逗號。
- block 裡宣告的變數只在 block 內有效（作用域）。
- block 表達式在 Rust 裡非常常見，是很重要的基礎概念。

3.7 enum 攜帶 tuple variant

3.7.1 本集目標

學會讓 enum 的 variant 攜帶額外的資料，像 tuple 一樣。

3.7.2 概念說明

之前學的 C-style enum，每個 variant 就只是一個名字，不帶任何資料。但很多時候，不同的選項需要攜帶不同的資料。

比如說，「形狀」可以是圓形或長方形。圓形需要一個半徑，長方形需要寬和高——它們需要的資料不一樣。在 Rust 裡，你可以讓每個 variant 攜帶資料：

```
enum Shape {
    Circle(f64),          // 攜帶一個 f64（半徑）
    Rectangle(i32, i32), // 攜帶兩個 i32（寬、高）
}
```

這種寫法像是在 variant 名字後面加上 tuple 的欄位，所以叫做 **tuple variant**。

建立值的方式就像呼叫函數一樣，把資料放在括號裡：

```
    let s = Shape::Circle(3.14);
    let r = Shape::Rectangle(10, 20);
```

注意：現在我們知道怎麼建立帶資料的 enum 了，但要「取出」裡面的資料，需要用 match——這個我們第 9 集會學。

3.7.3 範例程式碼

```
enum Shape {
    Circle(f64),
    Rectangle(i32, i32),
}

enum Message {
    Quit,            // 不帶資料（就像 C-style）
    Echo(i32),       // 帶一個 i32
    Move(i32, i32), // 帶兩個 i32
}

fn main() {
    let s1 = Shape::Circle(5.0);
    let s2 = Shape::Rectangle(10, 20);

    let m1 = Message::Quit;
    let m2 = Message::Echo(42);
    let m3 = Message::Move(3, 7);

    // 目前先建立值就好
    // 第 9 集會學怎麼用 match 取出裡面的資料
    println!("形狀和訊息都建立好了！");

    // 同一個 enum 裡，不同 variant 可以帶不同數量、不同型別的資料
    // 甚至有些 variant 不帶資料也完全沒問題（像 Message::Quit）
}
```

3.7.4 重點整理

- enum variant 可以攜帶資料：`Circle(f64)` 表示 Circle 帶一個 `f64`。

- 建立帶資料的 variant：`Shape::Circle(5.0)`。
- 同一個 enum 裡，不同 variant 可以帶不同的資料。
- 有些 variant 可以不帶資料，有些帶一個，有些帶多個——很靈活。
- 要取出 variant 裡的資料，需要用 match（第 9 集會學）。

3.8 enum 攜帶 struct variant

3.8.1 本集目標

學會讓 enum 的 variant 用類似 struct 的方式攜帶有名字的欄位。

3.8.2 概念說明

上一集學了 tuple variant，欄位沒有名字，用位置來區分。但如果一個 variant 攜帶的資料比較多，沒有名字就很容易搞混。

Rust 允許你用類似 named-field struct 的寫法，讓 variant 的每個欄位都有名字：

```
enum Shape {
    Circle { radius: f64 },
    Rectangle { width: i32, height: i32 },
}
```

建立值的時候就像建立 struct 一樣：

```
    let s = Shape::Circle { radius: 5.0 };
    let r = Shape::Rectangle { width: 10, height: 20 };
```

同一個 enum 裡，有些 variant 可以用 tuple 形式，有些可以用 struct 形式，甚至有些不帶資料，完全可以混搭。

3.8.3 範例程式碼

```
enum Shape {
    Circle { radius: f64 },
    Rectangle { width: i32, height: i32 },
    Dot, // 不帶資料的 variant 也可以混在一起
}

fn main() {
    let s1 = Shape::Circle { radius: 5.0 };
    let s2 = Shape::Rectangle { width: 10, height: 20 };
    let s3 = Shape::Dot;

    // 目前還不能直接取出裡面的欄位
    // 第 10 集會學怎麼用 match 取出 struct variant 的資料
    println!("三種形狀都建立好了！");

    // 一個更生活化的例子
    let event = Event::Click { x: 100, y: 200 };
    println!("事件已建立！");
}
```

```
enum Event {
    Click { x: i32, y: i32 },
    KeyPress(char), // tuple 形式也可以混搭
    Quit,           // 不帶資料也行
}
```

3.8.4 重點整理

- variant 可以用 struct 形式攜帶有名字的欄位：`Circle { radius: f64 }`。
- 建立值：`Shape::Circle { radius: 5.0 }`。
- 同一個 enum 可以混搭：有的用 tuple 形式、有的用 struct 形式、有的不帶資料。
- struct 形式的好處是欄位有名字，程式碼有時候更容易讀懂。
- 取出欄位資料需要用 match（第 10 集會學）。

3.9 match 解構 tuple variant

3.9.1 本集目標

學會用 match 解構 enum tuple variant，取出裡面攜帶的資料。

3.9.2 概念說明

第 7 集我們學了怎麼建立帶資料的 enum variant，但一直沒辦法取出裡面的資料。現在終於可以了！

在 match 的模式裡，你可以用變數名來「接住」variant 裡的資料：

```
    match s {
        Shape::Circle(r) => println!("半徑是 {}", r),
        Shape::Rectangle(w, h) => println!("寬 {}，高 {}", w, h),
    }
```

`Shape::Circle(r)` 裡的 `r` 不是固定的名字——你可以取任何名字。它的意思是「如果 s 是 Circle，就把裡面的那個 f64 值取出來，叫做 r」。

這個動作叫做**解構**（destructuring）——把一個複合的東西拆開，取出裡面的各個部分。match 不只是比對「是哪個 variant」，還能同時把裡面的資料解構出來給你用。

3.9.3 範例程式碼

```
enum Shape {
    Circle(f64),
    Rectangle(i32, i32),
}

fn main() {
    let s = Shape::Circle(5.0);

    match s {
        Shape::Circle(r) => {
            println!("這是一個圓形");
            println!("半徑是 {}", r);
            let area = r * r * 3.14159;
```

```
            println!("面積大約是 {}", area);
        }
        Shape::Rectangle(w, h) => {
            println!("這是一個長方形");
            println!("寬 {}，高 {}", w, h);
            let area = w * h;
            println!("面積是 {}", area);
        }
    }

    // 再一個例子
    let action = Action::Move(3, -2);

    match action {
        Action::Stop => println!("停止不動"),
        Action::Move(dx, dy) => {
            println!("往 x 方向移動 {}，往 y 方向移動 {}", dx, dy);
        }
    }
}

enum Action {
    Stop,
    Move(i32, i32),
}
```

3.9.4 重點整理

- **解構**（destructuring）：把複合的東西拆開，取出裡面的各個部分。
- 在 `match` 模式裡，可以在括號中使用變數名解構 tuple variant。
- `Shape::Circle(r)` → 把 `Circle` 裡的值取出來叫做 `r`。
- `Shape::Rectangle(w, h)` → 把 `Rectangle` 裡的兩個值分別叫做 `w` 和 `h`。
- 變數名可以自己取。
- `match` 依然要窮舉所有 variant。

3.10 `match` 解構 `struct` variant

3.10.1 本集目標

學會用 `match` 解構 `enum` `struct` variant，取出裡面的有名字欄位。

3.10.2 概念說明

第 9 集學了怎麼解構 tuple variant（用位置），現在來學著解構 `struct` variant（用欄位名）。

語法是在模式裡用 `欄位名: 變數名` 的寫法：

```
    match s {
        Shape::Circle { radius: r } => println!("半徑 {}", r),
        Shape::Rectangle { width: w, height: h } => println!("{}x{}", w, h),
    }
```

`radius: r` 的意思是「把 `radius` 這個欄位的值取出來，叫做 `r`」。冒號左邊是欄位名，右邊是你自

己取的變數名。

這和建立 struct variant 的語法很像，只是方向相反：建立是「把值放進去」，match 是「把值拿出來」。

3.10.3　範例程式碼

```
enum Shape {
    Circle { radius: f64 },
    Rectangle { width: i32, height: i32 },
}

fn main() {
    let s = Shape::Rectangle { width: 10, height: 5 };

    match s {
        Shape::Circle { radius: r } => {
            println!("這是圓形，半徑 = {}", r);
            let area = r * r * 3.14159;
            println!("面積大約 {}", area);
        }
        Shape::Rectangle { width: w, height: h } => {
            println!("這是長方形");
            println!("寬 = {}，高 = {}", w, h);
            let area = w * h;
            println!("面積 = {}", area);
            let perimeter = 2 * (w + h);
            println!("周長 = {}", perimeter);
        }
    }
}
```

3.10.4　一般的 struct 也能用同樣的方式

不只 enum 的 struct variant，一般的 named-field struct 也能用同樣的方式解構：

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 3, y: 0 };

    match p {
        Point { x: 0, y: 0 } => println!("原點"),
        Point { x: a, y: 0 } => println!("在 x 軸上，x = {}", a),
        Point { x: 0, y: b } => println!("在 y 軸上，y = {}", b),
        Point { x: a, y: b } => println!("在 ({}, {})", a, b),
    }
}
```

語法完全一樣——型別名 { 欄位名: 變數名 }。

注意上面的模式混用了**固定的值**和**變數**：Point { x: 0, y: b } 裡面 x: 0 是固定值（只在 x 等於 0

的時候才符合），`y: b` 是變數（把 y 的值取出來叫 b）。這個技巧在 `match` 裡很常用。`match` 會從上到下依序比對每個模式。一旦比對成功，就執行右手邊的程式碼，執行完後直接離開整個 `match`——不會繼續往下比對。

3.10.5 重點整理

- 在 `match` 裡用 `欄位名: 變數名` 來解構 `struct` variant。
- `Shape::Circle { radius: r }` → 把 `radius` 欄位取出來叫做 `r`。
- 冒號左邊是欄位名（必須和定義一樣），右邊是你自己取的變數名。
- 一般的 named-field `struct` 也能用同樣的方式在 `match` 裡解構。
- 模式裡可以混用固定值和變數：`Point { x: 0, y: b }` 表示「x 必須是 0，y 取出來叫 b」。
- `match` 從上到下比對，一旦成功就執行該分支的程式碼然後離開 `match`。
- 所有欄位都要寫出來（目前是這樣，之後會學怎麼忽略）。

3.11 field shorthand

3.11.1 本集目標

學會用 field shorthand 簡化 `struct` 的建立和模式匹配。

3.11.2 概念說明

上一集在 match 裡寫了 `radius: r`，意思是把 `radius` 欄位取出來叫做 `r`。但如果你想讓變數名就叫做 `radius` 呢？按照之前的寫法要寫 `radius: radius`——欄位名和變數名重複了，有點囉嗦。

Rust 提供了一個簡寫：如果變數名和欄位名一樣，可以只寫一次：

```
    // 完整寫法
    Shape::Circle { radius: radius };
    // 簡寫（field shorthand）
    Shape::Circle { radius };
```

這個簡寫不只在 `match` 裡可以用，**建立 `struct` 的時候也可以用**：

```
    let x = 3;
    let y = 7;
    // 完整寫法
    let p = Point { x: x, y: y };
    // 簡寫
    let p = Point { x, y };
```

只要變數名和欄位名一樣，就能省略 `: 變數名` 的部分。

3.11.3 範例程式碼

```
struct Point {
    x: i32,
    y: i32,
}

enum Shape {
    Circle { radius: f64 },
```

```
    Rectangle { width: i32, height: i32 },
}

fn main() {
    // 建立 struct 時使用 field shorthand
    let x = 5;
    let y = 10;
    let p = Point { x, y }; // 等同於 Point { x: x, y: y }
    println!("點的座標：({}, {})", p.x, p.y);

    // 建立 enum struct variant 時也可以用
    let radius = 3.5;
    let s = Shape::Circle { radius }; // 等同於 Shape::Circle { radius: radius }

    // match 裡也可以用 field shorthand
    match s {
        Shape::Circle { radius } => {
            println!("圓形，半徑 = {}", radius);
        }
        Shape::Rectangle { width, height } => {
            println!("長方形 {}x{}", width, height);
        }
    }
}
```

3.11.4　重點整理

- 當變數名和欄位名相同時，可以只寫一次：`Point { x, y }` 等同於 `Point { x: x, y: y }`。
- 這個簡寫叫做 **field shorthand**。
- 建立 `struct` / `enum` variant 時可以用。
- `match` 模式裡也可以用。
- 這是一個很常見的寫法，實際寫 Rust 時常常用簡寫。

3.12　tuple pattern

3.12.1　本集目標

學會在 `match` 裡解構一般的 tuple 與 tuple `struct`。

3.12.2　概念說明

第 9 集學了怎麼在 `match` 裡解構 `enum` variant。其實不只 `enum`，我們也能用 `match` 解構一般的 tuple！

```
fn main() {
    let point = (3, 7);

    match point {
        (0, 0) => println!("原點"),
        (x, 0) => println!("在 x 軸上，x = {}", x),
        (0, y) => println!("在 y 軸上，y = {}", y),
        (x, y) => println!("在 ({}, {})", x, y),
```

```
    }
}
```

`match` 會從上到下比對：

- `(0, 0)` → 兩個值都是 0 才會符合。
- `(x, 0)` → 第二個值是 0，第一個值隨便（取出來叫 x）。
- `(0, y)` → 第一個值是 0，第二個值隨便。
- `(x, y)` → 什麼都會符合（最後一個分支當「預設」）。

跟第 10 集學的一樣，模式裡可以混用「固定的值」和「變數」。固定的值用來比對，變數用來接住資料。

3.12.3 範例程式碼

```
fn main() {
    let point = (2, 0);

    match point {
        (0, 0) => println!("原點"),
        (x, 0) => println!("在 x 軸上，x = {}", x),
        (0, y) => println!("在 y 軸上，y = {}", y),
        (x, y) => println!("一般的點 ({}, {})", x, y),
    }

    // 用 match 搭配 tuple 做簡單的分類
    let score = (85, 90);

    match score {
        (100, 100) => println!("雙滿分！"),
        (a, b) => {
            println!("國文 {}，數學 {}", a, b);
            let total = a + b;
            println!("總分 {}", total);
        }
    }
}
```

3.12.4 tuple struct 也能用同樣的方式

還記得第 2 集學的 tuple struct 嗎？它的模式匹配方式和一般 tuple 一模一樣：

```
struct Point(i32, i32);

fn main() {
    let p = Point(3, 0);

    match p {
        Point(0, 0) => println!("原點"),
        Point(x, 0) => println!("在 x 軸上，x = {}", x),
        Point(0, y) => println!("在 y 軸上，y = {}", y),
        Point(x, y) => println!("在 ({}, {})", x, y),
    }
}
```

唯一的差別是模式前面要加上型別名稱 `Point(...)`，而普通 tuple 直接寫 `(...)`。

3.12.5　重點整理

- 一般的 tuple 也可以拿來 `match`。
- tuple `struct` 也能用同樣的方式進行模式匹配，只是前面要加型別名稱：`Point(x, y)`。

3.13　slice pattern

3.13.1　本集目標

學會用 slice pattern 解構陣列和切片。

3.13.2　概念說明

3.13.2.1　對陣列進行模式匹配

上一集我們學了如何解構 tuple，其實我們也可以對陣列和切片進行模式匹配！就是用 `[a, b, c]` 這種 slice pattern 來比對陣列的每個元素：

```
fn main() {
    let rgb = [255, 128, 0];

    match rgb {
        [255, 0, 0] => println!("純紅色"),
        [0, 255, 0] => println!("純綠色"),
        [0, 0, 255] => println!("純藍色"),
        [r, g, b] => println!("自訂顏色：R={}, G={}, B={}", r, g, b),
    }
}
```

跟前面幾集很像，你可以在模式裡混用「固定的值」和「變數」。固定的值用來比對，變數用來接住資料。

3.13.2.2　切片也能用

不只固定長度的陣列，切片（`&[T]`）也能用 slice pattern。差別在於切片的長度在編譯時期是未知的，所以你可以用不同長度的模式來匹配。下面最後那個 `_` 代表「其他所有情況」，第 15 集會再詳細講：

```
fn describe(numbers: &[i32]) {
    match numbers {
        [] => println!("空的"),
        [x] => println!("只有一個元素：{}", x),
        [x, y] => println!("兩個元素：{} 和 {}", x, y),
        [x, y, z] => println!("三個元素：{}, {}, {}", x, y, z),
        _ => println!("超過三個元素"),
    }
}
```

固定長度的陣列永遠是固定的長度，像 `[i32; 3]` 的分支永遠不會匹配到 `[]` 或 `[x]`。切片才需要考慮不同長度的情況。

3.13.3 範例程式碼

```
fn describe(data: &[i32]) {
    match data {
        [] => println!("空的切片"),
        [only] => println!("只有一個元素：{}", only),
        [first, second] => println!("兩個元素：{} 和 {}", first, second),
        _ => println!("有很多元素，第一個是 {}", data[0]),
    }
}

fn main() {
    // 固定長度陣列的 slice pattern
    let rgb = [255, 128, 0];

    match rgb {
        [255, 0, 0] => println!("純紅色"),
        [0, 255, 0] => println!("純綠色"),
        [0, 0, 255] => println!("純藍色"),
        [r, g, b] => println!("自訂顏色：R={}, G={}, B={}", r, g, b),
    }

    println!("---");

    // 切片的 slice pattern——可以匹配不同長度
    describe(&[]);
    describe(&[42]);
    describe(&[1, 2]);
    describe(&[10, 20, 30, 40, 50]);
}
```

3.13.4 重點整理

- 遇到陣列的時候，可以在 `match` 時用 `[a, b, c]` 這種 slice pattern，跟 tuple pattern 類似。
- 切片 `&[T]` 長度不固定，可以用不同長度的模式來匹配（`[]`、`[x]`、`[x, y]`……）

3.14 巢狀 pattern matching

3.14.1 本集目標

學會在 `match` 裡面再解構更深層的結構——巢狀的模式匹配。

3.14.2 概念說明

到目前為止，我們的 `match` 都只解構一層。但如果資料結構是巢狀的呢？比如一個 tuple 裡面包著 `enum`，或是一個 `enum` 裡面包著另一個 `struct`？

Rust 的 pattern matching 可以一次解構好幾層，就像剝洋蔥一樣，一層一層往裡面拿。

比如說，你有一個 tuple `(i32, Shape)`，你可以在 `match` 裡同時解構 tuple 和裡面的 `Shape`：

```
enum Shape {
    Circle(f64),
    Rectangle(i32, i32),
```

```
}

fn main() {
    let data = (666, Shape::Circle(42.0));
    match data {
        (id, Shape::Circle(r)) => println!("#{} 是圓形，半徑 {}", id, r),
        (id, Shape::Rectangle(w, h)) => println!("#{} 是長方形 {}x{}", id, w, h),
    }
}
```

一個模式裡，外層解構 tuple 取出 id 和 Shape，內層再解構 Shape 取出裡面的資料。全部在一行完成！

3.14.3　範例程式碼

```
enum Shape {
    Circle(f64),
    Rectangle(i32, i32),
}

struct Point {
    x: i32,
    y: i32,
}

fn main() {
    // 範例一：tuple 裡面包 enum
    let data = (1, Shape::Circle(5.0));

    match data {
        (id, Shape::Circle(r)) => {
            println!("形狀 #{} 是圓形，半徑 {}", id, r);
        }
        (id, Shape::Rectangle(w, h)) => {
            println!("形狀 #{} 是長方形 {}x{}", id, w, h);
        }
    }

    // 範例二：tuple 裡面包 struct
    let item = ("原點", Point { x: 0, y: 0 });

    match item {
        (name, Point { x, y }) => {
            println!("{}：座標 ({}, {})", name, x, y);
        }
    }
}
```

3.14.4　重點整理

- Rust 的 pattern matching 可以解構多層巢狀結構。
- 可以在一個模式裡同時解構 tuple + enum、tuple + struct 等。
- 巢狀解構讓你不需要寫多個 match，一次就能把所有資料取出來。

- 寫法就是把模式一層一層嵌進去，和資料的結構對應。

3.15 _ wildcard

3.15.1 本集目標

學會用 _ 來忽略不關心的值，以及在 match 裡建立預設分支。

3.15.2 概念說明

有時候在 match 裡，我們只關心某幾種情況，其他的都想「忽略」。Rust 提供了 _（底線）作為 **wildcard**（萬用字元），它可以匹配任何值，但不會把值綁定到變數上。

最常見的用法有兩種：

1. 預設分支：_ => ...

放在 match 的最後面，表示「其他所有情況都走這裡」：

```
fn main() {
    let score = 95;
    match score {
        100 => println!("滿分！"),
        _ => println!("不是滿分"),
    }
}
```

2. 忽略某個位置的值

在 tuple 或 enum 的模式裡，用 _ 佔住不需要的位置：

```
fn main() {
    let point = (5, 5);
    match point {
        (0, _) => println!("在 y 軸上"), // 不關心第二個值
        (_, 0) => println!("在 x 軸上"), // 不關心第一個值
        (_, _) => println!("其他位置"),
    }
}
```

3.15.3 範例程式碼

```
enum Direction {
    Up,
    Down,
    Left,
    Right,
}

fn main() {
    // _ 作為預設分支
    let dir = Direction::Left;

    match dir {
        Direction::Up => println!("往上"),
```

```
        _ => println!("不是往上（可能是下、左、右）"),
    }

    // _ 忽略 tuple 裡的某個值
    let record = ("Alice", 95, 'A');

    match record {
        (name, _, _) => println!("名字是 {}", name),
    }

    // 混合使用
    let data = (1, Direction::Up);

    match data {
        (_, Direction::Up) => println!("方向是上（不管編號是多少）"),
        (id, _) => println!("編號 {}（方向不是上）", id),
    }

    // 在 i32 上使用 _ 作為預設
    let score = 87;

    match score {
        100 => println!("滿分！"),
        0 => println!("零分..."),
        _ => println!("得了 {} 分", score),
    }
}
```

3.15.4　重點整理

- _ 是 wildcard，匹配任何值但不綁定變數。
- _ => ... 放在 match 最後面當「預設分支」，處理所有未列出的情況。
- 在模式裡用 _ 忽略不需要的欄位。
- 有了 _，match 就不用每個 variant 都寫出來了。

3.16　.. 忽略多個值

3.16.1　本集目標

學會用 .. 一次忽略 struct 或 tuple 中多個不關心的值。

3.16.2　概念說明

上一集學了用 _ 忽略一個值。但如果一個 struct 有很多欄位，而你只關心其中一兩個呢？每個不要的都寫 _ 太麻煩了。

Rust 提供了 ..（兩個點），意思是「剩下的我都不要了」。

3.16.2.1　在 match struct 時使用

```
struct Player {
    id: i32,
    hp: i32,
```

```
    mp: i32,
    level: i32,
}

fn main() {
    let p = Player { id: 1, hp: 0, mp: 50, level: 10 };

    match p {
        Player { hp: 0, .. } => println!("這個玩家倒下了！"),
        Player { level, .. } => println!("等級 {}", level),
    }
}
```

`Player { hp: 0, .. }` 表示「hp 是 0，其他欄位我不管」。不用每個不要的欄位都寫 `_`。

enum 的 struct variant 也能這樣匹配，用法完全一樣。

3.16.2.2 在 match tuple 時使用

```
fn main() {
    let scores = (90, 85, 78, 92, 88);

    match scores {
        (first, ..) => println!("第一科：{}", first),
    }

    match scores {
        (.., last) => println!("最後一科：{}", last),
    }

    match scores {
        (first, .., last) => println!("第一科 {}，最後一科 {}", first, last),
    }
}
```

`(first, ..)` 只取第一個，`(.., last)` 只取最後一個，`(first, .., last)` 取頭和尾。

tuple struct 和 enum 的 tuple variant 也能用類似的方式匹配，例如 `MyStruct(first, ..)` 或 `MyEnum::Variant(first, ..)`。

3.16.2.3 在陣列和切片裡使用

第 13 集學過 slice pattern，`..` 在陣列和切片裡也一樣好用：

```
fn main() {
    let data: &[i32] = &[10, 20, 30, 40, 50];

    match data {
        [first, .., last] => println!("頭 = {}，尾 = {}", first, last),
        [only] => println!("只有一個：{}", only),
        [] => println!("空的"),
    }
}
```

3.16.2.4 注意：.. 只能出現一次

`..` 在一層的模式裡只能出現一次，因為如果出現兩次，Rust 會不知道中間的值怎麼分配。

3.16.3　範例程式碼

```
struct Player {
    id: i32,
    hp: i32,
    mp: i32,
    level: i32,
}

fn main() {
    // struct 上使用 ..
    let p1 = Player { id: 1, hp: 100, mp: 50, level: 10 };

    match p1 {
        Player { hp, .. } => println!("HP = {}", hp),
    }

    let p2 = Player { id: 2, hp: 0, mp: 30, level: 5 };

    match p2 {
        Player { hp: 0, .. } => println!("這個玩家已經倒下了！"),
        Player { level, .. } => println!("等級 {}", level),
    }

    // tuple 上使用 ..
    let scores = (90, 85, 78, 92, 88);

    match scores {
        (first, ..) => println!("第一科：{}", first),
    }

    match scores {
        (.., last) => println!("最後一科：{}", last),
    }

    match scores {
        (first, .., last) => println!("第一科 {}，最後一科 {}", first, last),
    }

    // 切片上使用 ..
    let data: &[i32] = &[10, 20, 30, 40, 50];

    match data {
        [first, .., last] => println!("頭 = {}，尾 = {}", first, last),
        [only] => println!("只有一個：{}", only),
        [] => println!("空的"),
    }
}
```

3.16.4　重點整理

- `..` 用來一次忽略多個欄位或值。
- match struct 時：`Player { hp, .. }` 只取 hp，其他全部忽略；enum 的 struct variant 也一樣。

- match tuple 時：(first, ..) 只取第一個，(.., last) 只取最後一個；tuple struct 和 enum 的 tuple variant 用類似寫法。
- 陣列和切片裡：[first, ..] 取第一個，[first, .., last] 取頭和尾。
- .. 在一層模式裡只能出現一次。

3.17 range pattern

3.17.1 本集目標

學會在 match 裡用範圍來比對數值。

3.17.2 概念說明

之前學 match 的時候，我們都是一個一個值去比對。但如果想比對「1 到 5 之間的任何數字」呢？總不能寫五個分支吧。

Rust 提供了 **range pattern**，讓你在 match 裡用範圍來比對：

```
fn main() {
    let score = 12;
    match score {
        1..=5 => println!("低分"),
        _ => {}
    }
}
```

1..=5 代表 1、2、3、4、5（包含頭尾）。這個 ..= 和第 1 章學的 for i in 0..=5 是差不多的意思。

除了 ..=（包含結尾），也可以用 ..（不包含結尾）：

```
fn main() {
    let score = 65;
    match score {
        0..50 => println!("不及格"),  // 0 到 49
        50..=100 => println!("及格"), // 50 到 100（包含）
        _ => {}
    }
}
```

3.17.2.1 注意：兩種 .. 不要搞混！

上一集的 .. 和這一集的 .. 長得一模一樣，但意義完全不同：

- **上一集**：Point { x, .. } → 忽略剩餘欄位，.. 代表「其他我不管了」。
- **這一集**：0..50 → 數值範圍，.. 代表「從某個數到某個數」。

Rust 編譯器會根據前後文判斷是哪一種，不會搞混。但初學時要注意分辨。

3.17.2.2 單邊範圍

range pattern 也支援只寫一邊：

```
fn main() {
    let temperature = 25;
    match temperature {
```

```
        ..0 => println!("零下"),    // 小於 0
        0..=30 => println!("普通"), // 0 到 30
        31.. => println!("很熱"),   // 31 以上
    }
}
```

3.17.2.3　char 也能用

range pattern 不只能用在數字，也能用在 char：

```
fn main() {
    let c = '哼';
    match c {
        'a'..='z' => println!("小寫英文字母"),
        'A'..='Z' => println!("大寫英文字母"),
        '0'..='9' => println!("數字"),
        _ => println!("其他字元"),
    }
}
```

3.17.3　範例程式碼

```
fn main() {
    // 用 range pattern 判斷分數等級
    let score = 78;

    match score {
        90..=100 => println!("A"),
        80..90 => println!("B"),
        70..80 => println!("C"),
        60..70 => println!("D"),
        0..60 => println!("F"),
        _ => println!("分數超出範圍"),
    }

    // 單邊範圍
    let temperature = -5;

    match temperature {
        ..0 => println!("零下，很冷！"),
        0..=35 => println!("還可以"),
        36.. => println!("太熱了！"),
    }

    // char 的 range pattern
    let c = 'G';

    match c {
        'a'..='z' => println!("'{}' 是小寫字母", c),
        'A'..='Z' => println!("'{}' 是大寫字母", c),
        '0'..='9' => println!("'{}' 是數字", c),
        _ => println!("'{}' 是其他字元", c),
    }
}
```

3.17.4 重點整理

- `..` 和 `..=` 除了能用在 `for` 迴圈上，還能用來當作 pattern。
- `1..=5` → 包含頭尾（1, 2, 3, 4, 5）。
- `0..50` → 包含頭、不包含尾（0 到 49）。
- `..0` → 小於 0；`31..` → 31 以上（單邊範圍）。
- char 也能用 range pattern：`'a'..='z'`。
- 這裡的 `..` 是「數值範圍」，和上一集忽略欄位的 `..` 是不同的東西，不要搞混。

3.18 多個值 |

3.18.1 本集目標

學會在 `match` 的同一個分支裡比對多個可能的值。

3.18.2 概念說明

有時候你想讓好幾個值都執行同樣的程式碼。比如說，星期六和星期天都是假日，不需要分開寫兩個分支。

Rust 用 |（直線符號）來表示「或」：

```
fn main() {
    let day = 1;
    match day {
        6 | 7 => println!("假日"),
        _ => println!("工作日"),
    }
}
```

6 | 7 的意思是「6 或 7」。你可以用 | 串接任意多個值：

```
fn main() {
    let n = 3;
    match n {
        1 | 2 | 3 => println!("前三名"),
        _ => println!("其他"),
    }
}
```

也可以搭配 enum 使用：

```
enum Color {
    Red,
    Green,
    Blue,
}

fn main() {
    let color = Color::Red;
    match color {
        Color::Red | Color::Blue => println!("冷暖色"),
        Color::Green => println!("綠色"),
    }
```

```
}
```

3.18.3 範例程式碼

```
enum Season {
    Spring,
    Summer,
    Autumn,
    Winter,
}

fn main() {
    // 數字的多值比對
    let month = 7;

    match month {
        3 | 4 | 5 => println!("春天"),
        6 | 7 | 8 => println!("夏天"),
        9 | 10 | 11 => println!("秋天"),
        12 | 1 | 2 => println!("冬天"),
        _ => println!("無效月份"),
    }

    // enum 的多值比對
    let s = Season::Autumn;

    let is_hot = match s {
        Season::Summer => true,
        Season::Spring | Season::Autumn | Season::Winter => false,
    };
    println!("天氣熱嗎？{}", is_hot);

    // 搭配 range pattern 和 |
    let ch = '5';

    match ch {
        'a'..='z' | 'A'..='Z' => println!("字母"),
        '0'..='9' => println!("數字"),
        ' ' | '\t' | '\n' => println!("空白字元"),
        _ => println!("其他"),
    }
}
```

3.18.4 重點整理

- `|` 在 `match` 裡表示「或」，讓同一個分支比對多個值。
- 語法：`模式1 | 模式2 | 模式3 => ...`。
- 可以搭配 enum variant 使用。
- 也可以搭配 range pattern 使用：`'a'..='z' | 'A'..='Z'`。
- 當多個值要做相同處理時，用 `|` 比寫多個分支更簡潔。

3.19 @ 綁定

3.19.1 本集目標

學會用 @ 在比對模式的同時，把符合的值綁定到一個變數上。

3.19.2 概念說明

前面學過 range pattern：`0..=100` 可以比對 0 到 100 之間的值。現在假設一個 `Command` 的 `SetVolume` variant 攜帶了音量，我們想檢查音量是否在範圍內，並在分支裡印出實際的音量：

```
enum Command {
    SetVolume(i32),
    SetBrightness(i32),
    Quit,
}

fn main() {
    let command = Command::SetVolume(72);

    match command {
        Command::SetVolume(level @ 0..=100) => {
            println!("把音量設為 {}", level);
        }
        Command::SetVolume(level) => {
            println!("音量 {} 超出範圍", level);
        }
        Command::SetBrightness(level) => {
            println!("把亮度設為 {}", level);
        }
        Command::Quit => println!("結束程式"),
    }
}
```

`Command::SetVolume(level @ 0..=100)` 裡，右邊的 `0..=100` 負責比對範圍，左邊的 `level` 則綁定實際音量。當值是 `Command::SetVolume(72)` 時，這個 pattern 會匹配成功，而分支裡的 `level` 就是 72。

這就是 @ 的語法：

```
變數名 @ pattern
```

左邊建立綁定，右邊負責比對。比對成功後，就能在分支裡使用左邊的變數。

@ 不只能搭配 range，也能搭配 | 等其他 pattern：

下方範例的第一個分支先用 `('a' | 'e' | 'i' | 'o' | 'u')` 比對小寫母音，再把匹配到的字元綁定成 `key`。當 @ 右邊用了 | 時，這一組 pattern 要用括號包起來。

`MouseClick` 的分支則示範在 `struct` variant 的欄位裡使用 @。`0..=10` 比對 `x` 欄位的範圍，`horizontal` 綁定匹配到的實際座標。

3.19.3 範例程式碼

```
enum Event {
    KeyPress(char),
    MouseClick { x: i32, y: i32 },
    Quit,
}

fn main() {
    let event = Event::MouseClick { x: 6, y: 30 };

    match event {
        Event::KeyPress(key @ ('a' | 'e' | 'i' | 'o' | 'u')) => {
            println!("按下小寫母音 '{}'", key);
        }
        Event::KeyPress(key @ 'a'..='z') => {
            println!("按下其他小寫字母 '{}'", key);
        }
        Event::KeyPress(key) => {
            println!("按下其他按鍵 '{}'", key);
        }
        Event::MouseClick {
            x: horizontal @ 0..=10,
            y,
        } => {
            println!("在左側區域點擊：({}, {})", horizontal, y);
        }
        Event::MouseClick { x, y } => {
            println!("在其他區域點擊：({}, {})", x, y);
        }
        Event::Quit => println!("結束"),
    }
}
```

3.19.4 重點整理

- 變數名 `@ pattern` 會用右邊的 pattern 比對；成功後，實際值會綁定到左邊的變數。
- `Command::SetVolume(level @ 0..=100)` 同時限制音量範圍並取得實際音量。
- `@` 可以用在 enum variant、`struct` 欄位等巢狀資料中。
- `@` 可以搭配 range、`|` 等 pattern；搭配 `|` 時要寫成 `value @ (pattern1 | pattern2)`。

3.20 `match` guard

3.20.1 本集目標

學會在 `match` 分支加上額外的條件判斷（guard）。

3.20.2 概念說明

pattern 很擅長檢查資料的形狀、固定值與範圍。不過 pattern 不會執行 `from == to` 這類欄位間的比較，而且使用 `let` 建立的變數不能當作 range pattern 的邊界。需要這類額外運算時，就可以使用 `match` guard。

Rust 的 **match guard** 可以在 pattern 後面加上 `if` 條件：

```
pattern if 條件 => ...
```

例如，一筆感測器讀數會攜帶房間編號和測量值。我們可以先用 pattern 解構欄位，再用 guard 判斷測量值是否超過警戒線：

```rust
enum Reading {
    Temperature { room: i32, celsius: i32 },
    Humidity { room: i32, percent: i32 },
    Offline { room: i32 },
}

fn main() {
    let reading = Reading::Temperature {
        room: 3,
        celsius: 34,
    };
    let heat_warning = 30;

    match reading {
        Reading::Temperature { room, celsius } if celsius >= heat_warning => {
            println!("{} 號房過熱：{} 度", room, celsius);
        }
        Reading::Temperature { room, celsius } => {
            println!("{} 號房溫度正常：{} 度", room, celsius);
        }
        Reading::Humidity { room, percent } if percent > 70 => {
            println!("{} 號房太潮濕：{}%", room, percent);
        }
        Reading::Humidity { room, percent } => {
            println!("{} 號房濕度正常：{}%", room, percent);
        }
        Reading::Offline { room } => {
            println!("{} 號房的感測器離線", room);
        }
    }
}
```

第一個分支可以分成兩步理解：

1. `Reading::Temperature { room, celsius }` 先確認資料是 `Temperature`，並把兩個欄位分別綁定成 `room` 和 `celsius`。
2. `if celsius >= heat_warning` 再使用剛才綁定的 `celsius` 做額外判斷。

pattern 匹配成功後，`room` 和 `celsius` 就能在 guard 與右邊的程式碼中使用。guard 也可以使用 pattern 以外、早已存在的變數，例如上面的 `heat_warning`。

3.20.3 guard 不成立時會繼續往下

pattern 匹配成功，不代表這個分支一定執行。如果 guard 是 `false`，Rust 會繼續嘗試後面的分支。

以上面的溫度為例：

- 如果是 `Temperature` 而且 `celsius >= heat_warning`，執行第一個分支。
- 如果是 `Temperature` 但沒有達到警戒值，第一個 guard 失敗，接著由第二個 `Temperature` 分

支處理。

- 如果根本不是 `Temperature`，第一、第二個 pattern 都不匹配，繼續尋找其他 variant。

所以有 guard 的分支通常放在較一般的分支前面。

guard 不只可以比較一個欄位和門檻，也能比較同一個 pattern 綁定的多個欄位：

下方範例的第一個 guard 比較 `from` 和 `to`，第二個 guard 比較 `amount` 和外部的 `daily_limit`。這類需要欄位間運算或執行時期變數的條件，正是 guard 比單純 pattern 更合適的地方。

3.20.4　範例程式碼

```
enum Request {
    Transfer {
        from: i32,
        to: i32,
        amount: i32,
    },
    CheckBalance {
        account: i32,
    },
}

fn main() {
    let request = Request::Transfer {
        from: 1001,
        to: 2002,
        amount: 1500,
    };
    let daily_limit = 1000;

    match request {
        Request::Transfer { from, to, amount } if from == to => {
            println!(
                "帳戶 {} 不需要轉帳給自己，金額 {}",
                from, amount
            );
        }
        Request::Transfer { from, to, amount } if amount > daily_limit => {
            println!(
                "從帳戶 {} 轉 {} 到帳戶 {}，需要額外確認",
                from, amount, to
            );
        }
        Request::Transfer { from, to, amount } => {
            println!(
                "從帳戶 {} 轉 {} 到帳戶 {}",
                from, amount, to
            );
        }
        Request::CheckBalance { account } => {
            println!("查詢帳戶 {} 的餘額", account);
        }
    }
}
```

3.20.5 guard 與窮舉檢查

假設我們為 Temperature 寫了兩個分支：

- 一個 guard 是 celsius >= heat_warning。
- 另一個 guard 是 celsius < heat_warning。

我們看得出來，任何溫度一定會符合其中一個條件，邏輯上已經涵蓋所有可能。但 Rust 編譯器做窮舉檢查時，不見得能從 guard 之間的邏輯關係推論出「所有溫度都處理到了」。即使兩個分支都寫了，編譯器還是可能認為 Temperature 沒有被完整處理。

要讓涵蓋範圍對編譯器也很明確，可以保留一個**沒有 guard 的 pattern**。因此，前面的例子會讓第二個 Temperature 分支不加 guard：第一個分支處理達到警戒線的溫度，第二個分支接住其餘所有溫度。Humidity 和 Transfer 的分支也是同樣的安排。

3.20.6 重點整理

- match guard 的語法是 pattern if 條件 => ...。
- Rust 會先匹配 pattern、建立綁定，再檢查 guard。
- guard 可以使用同一個 pattern 綁定的變數，也可以使用外部已存在的變數。
- pattern 匹配但 guard 為 false 時，Rust 會繼續嘗試後面的分支。
- 即使幾個 guard 在邏輯上涵蓋所有可能，編譯器仍可能需要一個沒有 guard 的分支，才能確認 match 已經窮舉。
- guard 特別適合表達欄位之間的比較、計算，或和執行時期門檻的比較。

3.21 let 解構 tuple

3.21.1 本集目標

學會用 let 直接把 tuple 的值拆開，分別賦值給不同的變數。

3.21.2 概念說明

之前我們學了在 match 裡解構 tuple，像是 (x, y) => ...。但其實**不用 match，用 let 就可以直接解構**！

```
    let (x, y) = (1, 2);
```

這一行做了兩件事：

1. 建立一個 tuple (1, 2)。
2. 把第一個值取出來叫 x，第二個值取出來叫 y。

之前在第 2 章學 tuple 時，都是用 t.0、t.1 來取值。現在學了解構，你可以一行就把所有值拆開，每個值都有一個好讀的名字。

之前學的 _ 和 .. 也可以在 let 解構裡使用。

3.21.3 mut 在綁定上

之前在第 1 章學了 let mut x = 5;。其實 mut 不是型別的一部分——它是**綁定（binding）的修飾**。

既然 let 解構就是在做 binding，自然也可以對個別變數加 mut：

```
    let (mut a, b) = (1, 2);
    a += 10; // OK，a 是可變的
    b += 10; // 錯誤，b 是不可變的
```

同一個 pattern 裡，可以有些變數加 `mut`，有些不加——各自獨立。

這個規則不只適用於 `let`，**任何綁定變數的地方都能加 `mut`**：

- match 分支：`Some(mut x) => { x += 1; }`。
- for 迴圈：`for mut x in [1, 2, 3] { ... }`。
- 函數參數：`fn foo(mut x: i32) { x += 1; }`。

之後學到的綁定變數也一樣。都是同一件事——`mut` 修飾的是 binding，不是型別。

3.21.4　範例程式碼

```
fn main() {
    // 基本的 let 解構
    let (x, y) = (10, 20);
    println!("x = {}, y = {}", x, y);

    // 三個值的 tuple 也可以
    let (name, score, grade) = ("小明", 95, 'A');
    println!("{} 得了 {} 分，等級 {}", name, score, grade);

    // 搭配 _ 忽略不需要的值
    let (_, second, _) = (1, 2, 3);
    println!("只要第二個：{}", second);

    // 搭配 .. 忽略多個值
    let (first, ..) = (100, 200, 300, 400);
    println!("只要第一個：{}", first);

    // 個別加 mut
    let (mut a, b) = (1, 2);
    a += 10;
    println!("a = {}, b = {}", a, b);

    // 函數回傳 tuple，直接解構
    let (min, max) = min_max(7, 3);
    println!("最小 {}，最大 {}", min, max);
}

fn min_max(a: i32, b: i32) -> (i32, i32) {
    if a < b {
        (a, b)
    } else {
        (b, a)
    }
}
```

3.21.5　重點整理

- `let (x, y) = (1, 2);` 可以直接把 tuple 拆開。

- 解構 tuple 有時比用 .0、.1 更好讀。
- 可以搭配 _ 忽略單個值，搭配 .. 忽略多個值。
- mut 是 binding 的修飾，不是型別的一部分——任何綁定變數的地方都能加 mut。
- 函數回傳 tuple 時，可以用 let 解構直接取出每個值。

3.22 let 解構 struct

3.22.1 本集目標

學會用 let 直接把 struct 的欄位拆開，分別賦值給變數。

3.22.2 概念說明

上一集學了 let 解構 tuple，現在來解構 struct。概念完全一樣——用 let 把 struct 的欄位一次拆開：

```
    let Point { x, y } = p;
```

這一行會把 p.x 的值放進變數 x，p.y 的值放進變數 y。這裡用的是 field shorthand（第 11 集學的），所以 x 既是欄位名也是變數名。

如果你想要的變數名和欄位名不同，可以用 欄位名: 變數名 的寫法：

```
    let Point { x: px, y: py } = p;
    // 現在變數叫 px 和 py
```

之前學的 .. 也可以用，只取你需要的欄位：

```
    let Point { x, .. } = p;
    // 只取 x，忽略其他欄位
```

tuple struct 也可以解構，用法跟 tuple pattern 幾乎一樣，只是前面要加上型別名稱：

```
struct Pair(i32, i32);

fn main() {
    let p = Pair(1, 2);
    let Pair(a, b) = p;
}
```

3.22.3 範例程式碼

```
struct Point {
    x: i32,
    y: i32,
}

struct Rectangle {
    x: i32,
    y: i32,
    width: i32,
    height: i32,
}
```

```
fn main() {
    let p = Point { x: 5, y: 10 };

    // let 解構 struct（用 field shorthand）
    let Point { x, y } = p;
    println!("x = {}, y = {}", x, y);

    // 用不同的變數名
    let p2 = Point { x: 3, y: 7 };
    let Point { x: px, y: py } = p2;
    println!("px = {}, py = {}", px, py);

    // 搭配 .. 只取部分欄位
    let rect = Rectangle { x: 0, y: 0, width: 100, height: 50 };
    let Rectangle { width, height, .. } = rect;
    println!("寬 {}，高 {}", width, height);
    let area = width * height;
    println!("面積 = {}", area);
}
```

3.22.4 重點整理

- `let Point { x, y } = p;` 把 struct 的欄位拆成個別變數。
- `..` 可以忽略不需要的欄位。
- tuple struct 也能解構：`let Pair(a, b) = p;`。
- `let` 解構在取出 struct 資料時非常方便。

3.23 for 迴圈解構

3.23.1 本集目標

學會在 for 迴圈的變數位置直接解構 tuple 或 struct。

3.23.2 概念說明

我們已經學過 `let` 可以解構 tuple 和 struct。其實 for 迴圈的變數位置也可以——同樣的解構語法直接寫上去就行。

走訪一個裝著 tuple 的陣列：

```
fn main() {
    let pairs = [(1, "one"), (2, "two"), (3, "three")];

    for (num, name) in pairs {
        println!("{} = {}", num, name);
    }
}
```

`(num, name)` 就是模式，陣列裡的每個元素都是 tuple，迴圈會把它拆開分別給 num 和 name。

走訪 struct 也一樣：

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let points = [
        Point { x: 0, y: 0 },
        Point { x: 1, y: 2 },
        Point { x: 3, y: 4 },
    ];

    for Point { x, y } in points {
        println!("({}, {})", x, y);
    }
}
```

把這想成是 let 解構和 for 迴圈的結合：每次迴圈拿出一個元素時，就用 let 解構的語法把它拆開。

3.23.3 範例程式碼

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    // 走訪 tuple 陣列並解構
    let scores = [("Alice", 85), ("Bob", 92), ("Carol", 78)];
    for (name, score) in scores {
        println!("{}: {}", name, score);
    }

    // 走訪 struct 陣列並解構
    let points = [
        Point { x: 0, y: 0 },
        Point { x: 3, y: 4 },
        Point { x: -1, y: 2 },
    ];
    for Point { x, y } in points {
        println!("({}, {})", x, y);
    }

    // 用 .. 忽略不要的欄位
    let more_points = [
        Point { x: 1, y: 10 },
        Point { x: 2, y: 20 },
    ];
    for Point { x, .. } in more_points {
        println!("x = {}", x);
    }
}
```

3.23.4 重點整理

- `for` 迴圈的變數位置可以直接寫解構模式。
- 走訪 tuple 的陣列：`for (a, b) in pairs`。
- 走訪 `struct` 的陣列：`for Point { x, y } in points`。

3.24 函數參數解構

3.24.1 本集目標

學會在函數的參數位置直接解構 tuple 或 `struct`。

3.24.2 概念說明

我們已經學了在 `let`、`match` 和 `for` 裡解構。其實**函數的參數也可以解構**！

假設你有一個函數，接收一個 tuple `(i32, i32)` 代表座標。與其在函數內再拆開，不如直接在參數位置就拆好：

```
fn print_point((x, y): (i32, i32)) {
    println!("({}, {})", x, y);
}
```

注意語法：`(x, y)` 是模式（pattern），`: (i32, i32)` 是型別標註。模式和型別之間用 `:` 分隔。

呼叫的時候和平常一樣，傳一個 tuple 進去：

```
fn print_point((x, y): (i32, i32)) {
    println!("({}, {})", x, y);
}
fn main() {
    print_point((3, 7));
}
```

`struct` 也可以在參數位置解構：

```
fn print_point_struct(Point { x, y }: Point) {
    println!("({}, {})", x, y);
}
```

3.24.3 範例程式碼

```
struct Point {
    x: i32,
    y: i32,
}

// 函數參數解構 tuple
fn add_coordinates((x1, y1): (i32, i32), (x2, y2): (i32, i32)) -> (i32, i32) {
    (x1 + x2, y1 + y2)
}

// 函數參數解構 struct
// 當然這邊你也能選擇用 match
fn describe_point(Point { x, y }: Point) {
```

```
    if x == 0 && y == 0 {
        println!("原點");
    } else if x == 0 {
        println!("在 y 軸上，y = {}", y);
    } else if y == 0 {
        println!("在 x 軸上，x = {}", x);
    } else {
        println!("一般的點 ({}, {})", x, y);
    }
}

fn main() {
    // 傳 tuple 給函數
    let a = (1, 2);
    let b = (3, 4);
    let result = add_coordinates(a, b);
    println!("({}, {}) + ({}, {}) = ({}, {})", a.0, a.1, b.0, b.1, result.0, result.1);

    // 傳 struct 給函數
    let p = Point { x: 0, y: 5 };
    describe_point(p);

    let origin = Point { x: 0, y: 0 };
    describe_point(origin);
}
```

3.24.4 為什麼 tuple 和 struct 能用 let 解構？

你可能會好奇：為什麼 tuple 和 struct 就可以在 let、for 和函數參數裡直接解構？

```
    let (x, y) = (1, 2);    // OK
    let Point { x, y } = p; // OK
    let Shape::Circle { radius } = s; // 不行！
```

答案是：**本集使用的 tuple 和 struct pattern 都不會比對失敗**。任何 (i32, i32) 都能符合 (x, y)，任何 Point 都能符合 Point { x, y }。

但 enum 不一樣。一個 Shape 可能是 Circle 或 Rectangle。如果你寫 let Shape::Circle { radius } = s;，但 s 其實是 Rectangle 呢？這就失敗了。Rust 不允許 let 裡出現可能失敗的模式。

比對時一定會成功的模式被叫做 **irrefutable pattern**(不可反駁的模式)，可能失敗的叫做 **refutable pattern**（可反駁的模式）。let、for 和函數參數只接受 irrefutable pattern。

還有什麼 irrefutable pattern 呢？

```
fn main() {
    let arr = [1, 2, 3];
    let [head, ..] = arr;
    println!("第一個元素是 {}", head);
}
```

arr 的型別是 [i32; 3]，編譯器看一眼就知道它一定有三個元素，所以 [head, ..] 比對一定成功——這也是 irrefutable pattern，所以可以直接用 let 解構。反過來，如果今天是切片 &[i32]，那就不行了：切片有可能是空的，[head, ..] 在切片身上會變成 refutable，let 就接不住了。

想處理 refutable pattern？下一集會教 `if let`。

3.24.5 重點整理

- 函數參數也可以直接用模式解構：`fn foo((x, y): (i32, i32))`。
- tuple 和 `struct` 都可以在參數位置解構。
- 呼叫時和平常一樣傳值，解構是函數內部的事。
- `let`、`for` 和函數參數只接受不會比對失敗的 pattern（irrefutable pattern），所以本集的 `(x, y)` 和 `Point { x, y }` 可以使用，`Shape::Circle { radius }` 則不行。

3.25 `if let`

3.25.1 本集目標

學會用 `if let` 來簡化「只關心一種模式」的 `match`。

3.25.2 概念說明

有時候你只關心 `enum` 的某一個 variant，其他的都不在意。用 `match` 寫的話，必須處理所有情況，就算你只想處理一個：

```
    match c {
        Color::Red => println!("是紅色！"),
        _ => {} // 其他情況什麼都不做
    }
```

那個 `_ => {}` 看起來很多餘。Rust 提供了 `if let` 語法來簡化這種情況：

```
    if let Color::Red = c {
        println!("是紅色！");
    }
```

`if let 模式 = 值` 的意思是「如果這個值符合這個模式，就執行大括號裡的程式碼」。

你也可以加上 `else` 處理不符合的情況：

```
    if let Color::Red = c {
        println!("是紅色！");
    } else {
        println!("不是紅色");
    }
```

注意：`if let` 裡的 `=` 是一個等號，不是兩個。這不是在做比較，而是在做「模式匹配」。

3.25.3 範例程式碼

```
enum Color {
    Red,
    Green,
    Blue,
}

enum Shape {
    Circle(f64),
```

```
    Rectangle(i32, i32),
}

fn main() {
    let c = Color::Red;

    // 用 if let 檢查是不是 Red
    if let Color::Red = c {
        println!("是紅色！");
    }

    // 搭配 else
    let c2 = Color::Blue;

    if let Color::Red = c2 {
        println!("是紅色！");
    } else {
        println!("不是紅色");
    }

    // if let 也可以取出 variant 裡的資料
    let s = Shape::Circle(5.0);

    if let Shape::Circle(r) = s {
        println!("是圓形！半徑 = {}", r);
        let area = r * r * 3.14159;
        println!("面積大約 {}", area);
    }

    // 如果不是 Circle，if let 的大括號裡就不會執行
    let s2 = Shape::Rectangle(10, 20);

    if let Shape::Circle(r) = s2 {
        println!("這行不會被執行，因為 s2 是 Rectangle");
        println!("半徑 {}", r);
    } else {
        println!("不是圓形");
    }
}
```

3.25.4 if let guard

if let 也可以用在 match 的 guard 位置（第 20 集學的 match guard）。語法是 模式 if let 模式2 = 表達式 =>：

```
enum Wrapper {
    Value(i32),
    Empty,
}

fn lookup(key: i32) -> Wrapper {
    if key > 0 { Wrapper::Value(key * 10) } else { Wrapper::Empty }
}
```

```
fn main() {
    let items = [1, -2, 3];

    for item in items {
        match item {
            x if let Wrapper::Value(v) = lookup(x) => {
                println!("{} 查到了：{}", x, v);
            }
            x => println!("{} 查不到", x),
        }
    }
}
```

`x if let Wrapper::Value(v) = lookup(x)` 的意思是：先把值綁定到 `x`，然後用 `lookup(x)` 的結果再做一次模式匹配——只有結果是 `Wrapper::Value(v)` 的時候這個分支才成立。

這個例子其實用一般的 `if let` 也寫得出來。但當程式邏輯更複雜——例如外層的 `match` 已經在比對其他模式，而你又需要在某個分支裡對另一個值做模式匹配——`if let` guard 有時可以讓程式碼更好讀，不用在 `match` 的分支裡面再套一層 `if let`。

3.25.5 重點整理

- `if let 模式 = 值 { ... }` 是 `match` 只有一個分支時的簡寫。
- 只在值符合模式時執行大括號裡的程式碼。
- 可以加 `else` 處理不符合的情況。
- 可以在模式裡取出資料，像 `if let Shape::Circle(r) = s`。
- 比起寫 `match` + `_ => {}`，`if let` 更簡潔。
- `if let` 也能用在 `match` guard：`模式 if let 模式2 = 表達式 => ...`。

3.26 `while let`

3.26.1 本集目標

學會用 `while let` 在迴圈中持續做模式匹配，直到模式不再符合為止。

3.26.2 概念說明

上一集學了 `if let`——「如果符合模式就執行一次」。`while let` 則是「只要符合模式就一直執行」，是 `if let` 的迴圈版本。

語法：

```
while let 模式 = 值 {
    // 迴圈本體
}
```

每次迴圈開始前，Rust 會檢查「值是否符合模式」。符合就繼續跑，不符合就停下來。

為了示範 `while let`，我們用一個自訂 `enum` 來模擬「可能有值、可能結束」的情況：

```
enum Step {
    Value(i32),
    Done,
```

```
}
```

3.26.3 範例程式碼

```
enum Step {
    Value(i32),
    Done,
}

fn get_step(index: i32) -> Step {
    if index < 5 {
        Step::Value(index * 10)
    } else {
        Step::Done
    }
}

fn main() {
    let mut i = 0;

    // while let：只要 get_step 回傳 Value，就繼續
    while let Step::Value(v) = get_step(i) {
        println!("第 {} 步，值 = {}", i, v);
        i += 1;
    }
    println!("結束了！總共跑了 {} 步", i);

    println!();

    // 另一個例子：倒數計時
    let mut count = 5;

    // 利用自訂 enum 模擬倒數
    while let Countdown::Tick(n) = get_countdown(count) {
        println!("倒數 {}...", n);
        count -= 1;
    }
    println!("發射！🚀");
}

enum Countdown {
    Tick(i32),
    Launch,
}

fn get_countdown(n: i32) -> Countdown {
    if n > 0 {
        Countdown::Tick(n)
    } else {
        Countdown::Launch
    }
}
```

3.26.4 重點整理

- `while let 模式 = 值 { ... }` 是 `if let` 的迴圈版本。
- 只要值符合模式，就持續執行迴圈。
- 值不符合模式時，迴圈自動結束。

3.27 `let else`

3.27.1 本集目標

學會用 `let...else...` 在 pattern 不匹配時提前離開，寫出更扁平的程式碼。

3.27.2 概念說明

3.27.2.1 `if let` 的反面

上一集學了 `while let`，再上一集學了 `if let`——「如果匹配成功就做某件事」。但有時候你想要的是反過來：「如果匹配**失敗**就提前離開，成功的話繼續往下走。」

假設我們有這個 `enum`：

```
enum Color {
    Red,
    Green,
    Blue,
    Custom(i32, i32, i32),
}
```

用 `if let` 寫的話：

```
fn describe(color: Color) {
    if let Color::Custom(r, g, b) = color {
        println!("自訂顏色：{} {} {}", r, g, b);
    } else {
        println!("不是自訂顏色，結束");
        return;
    }
    // 這裡想用 r, g, b……但它們已經不在作用域了！
}
```

`r`、`g`、`b` 只活在 `if let` 的 `{}` 裡面，後面的程式碼用不到。

3.27.2.2 `let...else...` 語法

`let...else...` 讓綁定的變數活在後面的程式碼裡，而不是只活在 `{}` 裡面：

```
fn describe(color: Color) {
    let Color::Custom(r, g, b) = color else {
        println!("不是自訂顏色，結束");
        return;
    };
    // r, g, b 在這裡可以直接用！
    println!("紅：{}，綠：{}，藍：{}", r, g, b);
}
```

意思是：

1. 嘗試用 pattern 匹配 color。
2. 如果成功，r、g、b 被綁定，程式繼續往下。
3. 如果失敗，執行 else 裡面的程式碼。

3.27.2.3 else 裡面必須離開

else 區塊不能只是「做點事然後繼續」——它必須讓程式離開當前的流程。合法的寫法包括：

- return —— 離開函數
- break —— 離開迴圈
- continue —— 跳到迴圈下一輪

為什麼？因為如果 pattern 不匹配，變數就沒有被綁定。如果 else 之後程式繼續往下跑，那些變數就是未定義的——Rust 不允許這種事。

3.27.2.4 和 if let 的比較

- if let：匹配成功才進入 {} 區塊，綁定的變數只活在裡面。
- let...else...：匹配失敗就離開，綁定的變數活在後面所有的程式碼裡。

let...else... 讓程式碼更扁平——不用多縮排一層。

3.27.3 範例程式碼

```
enum Shape {
    Circle(f64),
    Rectangle(i32, i32),
}

fn print_circle_info(shape: Shape) {
    let Shape::Circle(radius) = shape else {
        println!("不是圓形，跳過");
        return;
    };
    // radius 在這裡可以直接用
    println!("圓形，半徑 = {}", radius);
}

fn main() {
    print_circle_info(Shape::Circle(3.14));
    print_circle_info(Shape::Rectangle(10, 20));

    // 在迴圈裡搭配 continue
    let shapes = [
        Shape::Rectangle(3, 4),
        Shape::Circle(1.0),
        Shape::Rectangle(5, 6),
        Shape::Circle(2.5),
    ];

    println!("\n只印圓形：");
    for shape in shapes {
        let Shape::Circle(r) = shape else {
            continue;  // 不是圓形，跳過這一輪
        };
```

```
        println!("半徑：{}", r);
    }
}
```

3.27.4　重點整理

- `let pattern = expr else { return / break / continue };` 在匹配失敗時提前離開。
- `else` 裡面必須離開當前流程（`return` / `break` / `continue`）。
- 匹配成功的話，綁定的變數在後續程式碼還能使用。
- 比 `if let` 更適合「失敗就離開，成功繼續」的場景——程式碼更扁平。

3.28　associated function

3.28.1　本集目標

學會用 `impl` 為 `struct` 或 `enum` 定義 associated function（關聯函數），以及用 `::` 呼叫。

3.28.2　概念說明

到目前為止，我們的函數都是「獨立的」——定義在最外層，和任何型別沒有關係。但很多時候，某些函數和特定的型別密切相關。比如說，「建立一個新的 `Point`」這件事，和 `Point` 這個型別有直接關係。

Rust 用 `impl` 區塊讓你把函數「附加」到型別上：

```
impl Point {
    fn new(x: i32, y: i32) -> Point {
        Point { x, y }
    }
}
```

這樣定義的函數叫做 **associated function**（關聯函數），因為它和 `Point` 這個型別「關聯」在一起。呼叫的時候用 `::`：

```
    let p = Point::new(3, 7);
```

`Point::new` 看起來是不是有點眼熟？之前用 `enum` 的時候也是用 `::` 啊！像 `Color::Red`。概念是一樣的——`::` 可以表示「某個型別底下的東西」。

associated function 最常見的用途就是 `new`——作為「建構函數」來建立型別的值。

3.28.3　範例程式碼

```
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    // associated function：建立一個新的 Point
    fn new(x: i32, y: i32) -> Point {
        Point { x, y }
    }
```

```
    // 也可以定義其他 associated function
    fn origin() -> Point {
        Point { x: 0, y: 0 }
    }
}

// enum 也可以有 impl！
enum Color {
    Red,
    Green,
    Blue,
}

impl Color {
    fn from_number(n: i32) -> Color {
        match n {
            0 => Color::Red,
            1 => Color::Green,
            _ => Color::Blue,
        }
    }
}

fn main() {
    // 用 :: 呼叫 associated function
    let p1 = Point::new(3, 7);
    println!("p1 = ({}, {})", p1.x, p1.y);

    let p2 = Point::origin();
    println!("p2 = ({}, {})", p2.x, p2.y);

    // enum 的 associated function
    let c = Color::from_number(1);
    match c {
        Color::Red => println!("紅"),
        Color::Green => println!("綠"),
        Color::Blue => println!("藍"),
    }
}
```

3.28.4 重點整理

- `impl 型別名 { ... }` 為型別定義 associated function。
- associated function 用 `型別名::函數名()` 呼叫。
- 最常見的用途是 `new` 函數，作為建構函數。
- `struct` 和 `enum` 都可以有 `impl` 區塊。

3.29 method

3.29.1 本集目標

學會用 `self` 定義 method（方法），讓函數可以用 `.` 在值上面呼叫。

3.29.2 概念說明

上一集學了 associated function，它是用 `::` 呼叫的，和「型別」相關。但有時候我們想對一個**已經存在的值**做操作，比如「算出這個 `Point` 的 `x + y`」。

這就是 **method**（方法）——參數列表的第一個位置放 `self`，代表「呼叫這個方法的那個值本身」：

```
impl Point {
    fn sum(self) -> i32 {
        self.x + self.y
    }
}
```

呼叫的時候用 `.` 而不是 `::`：

```
    let p = Point::new(3, 7);
    let s = p.sum(); // 用 . 呼叫 method
```

注意：呼叫 `p.sum()` 的時候，**不需要再手動傳入 `self`**。`.` 前面的 `p` 會自動變成方法裡的 `self`。所以雖然定義時寫了 `fn sum(self)`，呼叫時只要寫 `p.sum()` 而不是 `p.sum(p)`。

3.29.2.1 method 可以有其他參數

method 除了 `self` 之外，還可以有一個或更多其他的參數——就跟一般函數一樣：

```
impl Point {
    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}
```

呼叫時，`self` 由 `.` 前面的值自動帶入，你只需要傳其他的參數：

```
    let p1 = Point::new(1, 2);
    let p2 = Point::new(3, 4);
    let p3 = p1.add(p2); // p1 是 self，p2 是 other
```

3.29.2.2 associated function vs method 的差別：

- associated function：沒有 `self`，用 `::` 呼叫 → `Point::new(3, 7)`。
- method：第一個參數是 `self`，用 `.` 呼叫 → `p.sum()`。

3.29.3 範例程式碼

```
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    // associated function（沒有 self）
    fn new(x: i32, y: i32) -> Point {
        Point { x, y }
    }
```

```
    // method（第一個參數是 self）
    fn sum(self) -> i32 {
        self.x + self.y
    }

    // method 可以有 self 之外的參數
    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }

    // 另一個 method
    fn is_origin(self) -> bool {
        self.x == 0 && self.y == 0
    }
}

enum Direction {
    Up,
    Down,
    Left,
    Right,
}

impl Direction {
    // enum 也可以有 method
    fn is_horizontal(self) -> bool {
        match self {
            Direction::Left => true,
            Direction::Right => true,
            Direction::Up => false,
            Direction::Down => false,
        }
    }
}

fn main() {
    let p = Point::new(3, 7); // :: 呼叫 associated function
    let s = p.sum();          // . 呼叫 method
    println!("3 + 7 = {}", s);

    // method 帶其他參數
    let a = Point::new(1, 2);
    let b = Point::new(10, 20);
    let c = a.add(b); // a 是 self，b 是 other
    println!("相加後：({}, {})", c.x, c.y);

    let origin = Point::new(0, 0);
    println!("是原點嗎？{}", origin.is_origin());

    // enum 的 method
    let dir = Direction::Left;
```

```
    let horizontal = dir.is_horizontal();
    println!("是水平方向嗎？{}", horizontal);
}
```

3.29.4　重點整理

- method 的第一個參數是 `self`，代表值本身。
- method 用 `.` 呼叫：`p.sum()`，`.` 前面的值自動成為 `self`，不需要手動傳入。
- method 除了 `self` 還可以有其他參數：`fn add(self, other: Point) -> Point`，呼叫時括號內只寫 `self` 以外的參數。
- `struct` 和 `enum` 都可以有 method。

3.30　大寫 `Self`

3.30.1　本集目標

學會用大寫 `Self` 作為「目前正在 `impl` 的型別」的別名，讓程式碼更簡潔。

3.30.2　概念說明

上一集我們在 `impl` 裡面寫了這樣的程式碼：

```
impl Point {
    fn new(x: i32, y: i32) -> Point {
        Point { x, y }
    }
}
```

注意到 `Point` 這個名字出現了三次：`impl Point`、`-> Point`、`Point { x, y }`。如果型別名很長（例如 `Rectangle`），一直重複寫就很囉嗦。

Rust 提供了大寫 `Self`（注意 S 是大寫的！），它在 `impl` 區塊裡面代表「目前正在 `impl` 的型別」。所以上面的程式碼可以改成：

```
impl Point {
    fn new(x: i32, y: i32) -> Self {
        Self { x, y }
    }
}
```

`Self` 就是 `Point` 的別名。這樣寫有兩個好處：

1. 更簡潔，尤其是型別名很長的時候。
2. 如果之後改了型別名，`impl` 裡面不用每個地方都改。

注意區分：

- 小寫 `self`：代表「這個值本身」（method 的第一個參數）。
- 大寫 `Self`：代表「目前的型別」。

3.30.3 範例程式碼

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    // 用 Self 代替 Point
    fn new(x: i32, y: i32) -> Self {
        Self { x, y }
    }

    fn origin() -> Self {
        Self { x: 0, y: 0 }
    }

    // method 裡也可以用 Self
    fn flip(self) -> Self {
        Self { x: self.y, y: self.x }
    }

    fn sum(self) -> i32 {
        self.x + self.y
    }
}

// enum 也可以用 Self
enum Light {
    Red,
    Yellow,
    Green,
}

impl Light {
    fn next(self) -> Self {
        match self {
            Self::Red => Self::Green,
            Self::Green => Self::Yellow,
            Self::Yellow => Self::Red,
        }
    }

    fn is_stop(self) -> bool {
        match self {
            Self::Red => true,
            Self::Yellow => true,
            Self::Green => false,
        }
    }
}

fn main() {
    // struct 使用 Self
    let p = Point::new(3, 7);
```

```
    println!("原始：({}, {})", p.x, p.y);

    let p2 = Point::new(3, 7);
    let flipped = p2.flip();
    println!("翻轉：({}, {})", flipped.x, flipped.y);

    let p3 = Point::origin();
    println!("原點：({}, {})", p3.x, p3.y);

    // enum 使用 Self
    let light = Light::Red;
    let stop = light.is_stop();
    println!("需要停嗎？{}", stop);

    let light2 = Light::Red;
    let next_light = light2.next();
    let stop2 = next_light.is_stop();
    println!("下一個燈需要停嗎？{}", stop2);
}
```

3.30.4　重點整理

- 大寫 `Self` 在 `impl` 區塊裡代表「目前的型別」，可以用在需要寫型別的地方。
- `Self` 可以用在參數型別、回傳型別 `-> Self`、建構值 `Self { ... }`、以及 enum variant `Self::Red`。
- 小寫 `self` = 值本身，大寫 `Self` = 型別本身。
- `struct` 和 `enum` 的 `impl` 裡都可以用 `Self`。
- 使用 `Self` 讓程式碼更簡潔，也更容易維護。

恭喜你完成了第 3 章！🎉 這一章你學會了 `struct`、`enum`、pattern matching（`match`、`if let`、`while let`、`let...else...`）、解構、associated function 和 method。你現在已經能用 Rust 的型別系統來組織資料和行為了。下一章我們要進入 Rust 最核心也最獨特的概念——所有權（ownership）！

第 4 章

所有權與借用

本章會以一個與前面幾章非常不同的角度切入 Rust 這個程式語言的功能。Rust 除了有作為高階語言建立抽象的能力，也希望能達成較靠近硬體的低階語言的效能，為了滿足這樣的設計理念，Rust 必須在語言內設下諸多限制保證執行時期的效率。

4.1 所有權（鑰匙圈比喻）

4.1.1 本集目標

用生活化的鑰匙圈比喻，理解 Rust 最核心的概念——所有權。

4.1.2 正文

這集我們不寫程式，先來聊一個 Rust 最重要的概念：**所有權（ownership）**。

4.1.2.1 鑰匙圈比喻

想像你有一個鑰匙圈。鑰匙圈上可能掛著一些小裝飾（很輕、隨身帶著），也可能掛著一把鑰匙，這把鑰匙可以打開一個保險箱。規則很簡單：

> **每個鑰匙圈只能在一個人手上。**

這就是 Rust 的所有權規則。你手上拿著鑰匙圈，上面的裝飾和鑰匙都是你的。

4.1.2.2 移轉（move）= 交出去就沒了

如果有人跟你說：「把你的鑰匙圈給我。」你把整個鑰匙圈交給對方之後，你手上就什麼都沒了。你不能再用那把鑰匙去開保險箱，因為鑰匙已經不在你手上了。

在 Rust 裡面，這叫做 **move（移轉）**。當你把一個值交給別人（例如賦值給另一個變數），原本的變數就不能再用了。

4.1.2.3 為什麼不能複製鑰匙？

你可能會想：「那我去複製一把鑰匙不就好了？」

問題在這裡：如果兩個人各拿一把鑰匙，都可以打開同一個保險箱，那就可能出事了——

- A 正在整理保險箱裡的東西。
- B 同時也打開保險箱，把東西拿走了。
- A 回頭一看：「咦？我的東西呢？」

這表示同一份資料出現了互相衝突的存取。這種衝突如果發生在未經同步的多個執行緒之間，就稱為**資料競爭（data race）**。Rust 的所有權規則就是為了**從根本上防止這類問題**。

4.1.2.4 `clone` = 弄一個一樣好用的新鑰匙圈，並確保不會出問題

那如果我真的需要第二串一樣能用的鑰匙圈怎麼辦？

Rust 的答案叫做 **`clone`（克隆）**。它的意思是：**弄一個跟你手上這個一樣好用的新鑰匙圈，並且確保這樣做不會出問題。**

最常見的「確保」方式，就是不去動原本的鑰匙，而是買一個新的保險箱，把裡面的東西也各自 `clone` 一份放進去，配一把新鑰匙掛在新的鑰匙圈上。在最單純的情況下——箱子裡裝的都是普通的資料——兩個人會各有自己的保險箱、自己的東西，互不干擾，兩份完全獨立。

不過並非所有型別都用這一招。之後會遇到真的「只多打一把鑰匙」、改用其他機制確保安全的型別。但現階段你遇到的 `clone` 都可以理解成「買新保險箱」。

4.1.2.5 為什麼 Rust 要這麼嚴格？

大部分的程式語言不管這些，讓你隨便複製、隨便共用，然後等出了 bug 再說。Rust 不一樣——它在你寫程式的時候就幫你把關，確保不會有兩個人同時亂動同一份資料。

這就是 Rust 的核心哲學：**在編譯時期就防止錯誤，而不是等到程式跑起來才出事。**

4.1.3 重點整理

- 每個值都有一個「擁有者」，就像每個鑰匙圈只能在一個人手上。
- **`move`（移轉）**：把鑰匙圈交給別人，你就沒有了。
- 不能簡單地複製鑰匙去開同一個保險箱，這可能會造成互相衝突的存取。
- **`clone`（克隆）**：弄一個跟原本一樣好用的新鑰匙圈，並確保不會出問題——通常靠「買新保險箱 + 內容物各自 `clone` + 配新鑰匙」做到；最單純的情況下，兩份完全獨立。
- Rust 在編譯時期就強制執行所有權規則，防止這類存取衝突。

4.2 `trait` 簡介

4.2.1 本集目標

學會定義 `trait` 和為型別實作 `trait`，並認識 `#[derive]` 這個自動產生實作的捷徑。

4.2.2 概念說明

4.2.2.1 什麼是 `trait`？

在進入所有權的主題之前，我們先來學一個重要的工具：**`trait`**。它和上一集的鑰匙圈比喻沒有直接關係，但之後講 `Clone`、`Copy` 等概念的時候會用到，所以先學起來。

在第 3 章，我們學會了用 `impl` 幫 `struct` 和 `enum` 加上 method。但如果我們想要規定「某些型別都必須有某個功能」呢？

比如說，我想規定：「某些型別都必須能打招呼。」這就是 **`trait`** 的用途——它定義了一組「能力」或「行為」，然後不同的型別可以各自實作這些行為。

`trait` 就像一張「規格表」，上面寫著：「你要符合這個規格，就必須提供這些功能。」

4.2.2.2 定義 `trait`

用 `trait` 關鍵字來定義：

```
trait Greet {
    fn greet(self);
}
```

這段程式碼的意思是：「凡是實作了 Greet 這個 trait 的型別，都必須有一個 greet method。」

4.2.2.3 為型別實作 trait

```
impl Greet for Cat {
    fn greet(self) {
        println!("喵～");
    }
}
```

之前我們寫 impl Cat { ... } 是直接幫 Cat 加 method。現在寫 impl Greet for Cat { ... } 是說「Cat 符合 Greet 這個規格」，然後在裡面提供 Greet 要求的 method。

4.2.2.4 derive：自動產生實作的捷徑

有些 trait 的實作方式很固定，Rust 編譯器可以幫你自動產生。這時候就用 #[derive(...)]：

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}
```

還記得第 2 章我們用 {:?} 印出 tuple 和陣列嗎？其實 {:?} 就是在使用 Debug 這個 trait。tuple 和陣列內建就有 Debug，但我們自己定義的 struct 和 enum 沒有——所以要加 #[derive(Debug)]，讓 Rust 自動幫我們產生 Debug 的實作。

4.2.3 範例程式碼

```
// 定義一個 trait：所有實作者都必須能「打招呼」
trait Greet {
    fn greet(self);
}

// 定義兩種動物
struct Cat;
struct Dog;

// 幫 Cat 實作 Greet
impl Greet for Cat {
    fn greet(self) {
        println!("我是一隻貓咪喵～");
    }
}

// 幫 Dog 實作 Greet
impl Greet for Dog {
    fn greet(self) {
        println!("我是一隻狗狗汪！");
    }
}
```

```rust
// 用 derive 讓 Rust 自動產生 Debug 實作
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let cat = Cat;
    let dog = Dog;

    // 呼叫 trait method
    cat.greet();
    dog.greet();

    // 用 {:?} 印出 struct（因為有 #[derive(Debug)]）
    let p = Point { x: 3, y: 7 };
    println!("{:?}", p);
}
```

4.2.4 重點整理

- **trait** 是一組行為的規格定義，像是一張「能力清單」。
- 用 `impl TraitName for TypeName` 來幫型別實作 trait（例如 `impl Greet for Cat`）。
- `#[derive(Debug)]` 讓 Rust 自動幫你的 struct / enum 實作 Debug trait。
- 加了 `#[derive(Debug)]` 之後，就可以用 `{:?}` 來印出自訂的 struct / enum。

4.3 move 與 Clone

4.3.1 本集目標

理解 Rust 的 move 語意——賦值和傳入函數都會轉移所有權——以及用 Clone 來建立資料副本。

4.3.2 概念說明

4.3.2.1 move：交出去就沒了

上一集我們學了 trait，現在來看所有權在程式碼裡的樣子。

在 Rust 裡，當你把一個 struct 的值賦給另一個變數，原本的變數就**不能再用了**。這就是第 1 集講的「把鑰匙圈交出去」：

```rust
    let p1 = Point { x: 1, y: 2 };
    let p2 = p1; // p1 的所有權移轉給 p2
    // 從這裡開始，p1 不能再用了！
```

這個行為叫做 **move（移轉）**。Rust 編譯器會在編譯時期就檢查這件事，如果你在 move 之後還嘗試使用原本的變數，編譯器會直接報錯。

4.3.2.2 傳進函數也是 move

不只是賦值，把值傳進函數也會發生 move：

```
struct Point {
    x: i32,
    y: i32,
}

fn print_point(p: Point) {
    println!("({}, {})", p.x, p.y);
}

fn main() {
    let p1 = Point { x: 1, y: 2 };
    print_point(p1); // p1 被 move 進函數了
    // p1 不能再用了！
}
```

因為函數的參數就像一個新的變數，值被「交」給了它。

4.3.2.3 Clone

如果你需要保留原本的值，又想要一份副本，就用 **Clone**。

首先，你的型別要加上 #[derive(Clone)]（當然也可以順便加 Debug）：

```
#[derive(Debug, Clone)]
struct Point {
    x: i32,
    y: i32,
}
```

然後用 .clone() 來建立副本：

```
#[derive(Debug, Clone)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = p1.clone();  // 建立一份副本，p1 還在
    println!("{:?}", p1); // OK！p1 還能用
    println!("{:?}", p2); // p2 是獨立的副本
}
```

回想第 1 集的比喻：clone 的意思是「弄一個跟原本一樣好用的新鑰匙圈，並確保不會出問題」。對 Point 來說，新鑰匙圈就是一份完整的副本——#[derive(Clone)] 產生的 clone 正是這個做法：對每個欄位各自呼叫 clone，兩個變數各自擁有自己 clone 出來的結果。

4.3.2.4 整數不會 move？

你可能會注意到，整數的行為不太一樣：

```
fn main() {
    let a = 42;
    let b = a;
    println!("{}", a); // 這居然可以！
}
```

為什麼整數不會 move？這個問題我們下一集再來解答。

4.3.3 範例程式碼

```
#[derive(Debug, Clone)]
struct Point {
    x: i32,
    y: i32,
}

fn print_point(p: Point) {
    println!("函數收到的點：({}, {})", p.x, p.y);
}

fn main() {
    let p1 = Point { x: 10, y: 20 };

    // 用 clone 建立一份副本，這樣 p1 不會被 move 走
    let p2 = p1.clone();
    println!("p1 = {:?}", p1);
    println!("p2 = {:?}", p2);

    // 傳進函數也是 move，所以先 clone
    print_point(p1.clone());
    println!("p1 還在：{:?}", p1);

    // 如果不 clone，直接傳進去，p1 就被 move 走了
    print_point(p1);
    // 下面這行如果取消註解，編譯器會報錯：
    // println!("p1 不見了：{:?}", p1);
}
```

4.3.4 重點整理

- `let p2 = p1;` 會 **move**，之後 p1 不能再用。
- 把值傳進函數也是 move。
- `#[derive(Clone)]` + `.clone()` 會對每個欄位各自呼叫 `clone`——對 `Point` 這種型別來說，就是一份獨立的副本。
- `clone` 之後，原本的變數還可以繼續使用。
- 整數（`i32` 等）不會 move——下一集會解釋為什麼。

4.4 Copy

4.4.1 本集目標

理解 `Copy` `trait`——為什麼整數、浮點數、布林值、字元在賦值時不會 move。

4.4.2 概念說明

4.4.2.1 上一集的問題

上一集我們發現，`struct` 的值在賦值或傳入函數時會被 move，但整數不會：

```rust
fn main() {
    let a = 42;
    let b = a;
    println!("{}", a); // 完全沒問題！
}
```

為什麼？答案就是 **Copy trait**。

4.4.2.2 什麼是 Copy？

Copy 是一個特殊的 trait。如果一個型別實作了 Copy，那麼在賦值或傳入函數時，Rust 會自動複製一份，而不是 move。

Copy 表示 copy 這個值是單純、機械化的操作，所以 Rust 可以在賦值或傳入函數時自動完成，不用特別寫 .clone()。

4.4.2.3 哪些型別自動有 Copy？

以下這些型別天生就有 Copy：

- 整數：i8, i16, i32, i64, i128, u8, u16, u32, u64, u128, isize, usize。
- 浮點數：f32, f64。
- 布林值：bool。
- 字元：char。
- ……還有其他更多型別。

另外，**tuple** 和**陣列**如果裡面每個元素都是 Copy 的，那它們整體也是 Copy 的：

```rust
fn main() {
    let t = (1, true, 'a');  // (i32, bool, char) → 全部 Copy → tuple 也是 Copy
    let t2 = t;
    println!("{:?}", t);     // OK！

    let arr = [1, 2, 3];     // [i32; 3] → i32 是 Copy → 陣列也是 Copy
    let arr2 = arr;
    println!("{:?}", arr);   // OK！
}
```

這就是為什麼你在前面幾章寫的程式碼裡，整數、tuple、陣列可以隨便賦值給多個變數、傳進多個函數，完全不會有問題。

除了 Copy 之外，當 tuple 的每個型別都有實作 Clone 時，tuple 也會自動實作 Clone。事實上 tuple 對一些其他 trait 也有同樣的行為——對這些 trait 而言，只要所有元素都有實作，tuple 整體就會有。這點以後不再贅述。

4.4.2.4 自己的型別也可以加 Copy

如果你的型別裡面所有值都是 Copy 的型別，那你的型別也可以加上 #[derive(Copy, Clone)]：

```rust
#[derive(Debug, Copy, Clone)]
struct Point {
    x: i32,
    y: i32,
}
```

注意：Copy 要求 Clone。使用 derive 時，通常會寫成 #[derive(Copy, Clone)]；如果型別沒有

實作 Clone，只寫 #[derive(Copy)] 就會編譯失敗。

為什麼？因為 Rust 規定：任何可以 copy 的東西，也必須可以 clone。Copy 是「自動複製」，Clone 是「手動呼叫 .clone()」。如果一個東西連手動 clone 都做不到，那自動複製當然更不該發生。所以 Copy 要求你先有 Clone。

加上之後，Point 的行為就跟整數一樣了——賦值不會 move：

```
#[derive(Debug, Copy, Clone)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = p1; // 自動複製，p1 還在！
    println!("{:?}", p1); // OK
}
```

4.4.2.5 copy 和 clone 的差別

	copy	clone
觸發方式	自動（賦值、傳入函數）	手動（.clone()）
能否自訂行為	不能	可以，透過 .clone() 的實作決定
實作前提	內容都必須是 Copy	沒有限制

簡單來說：**copy 是自動的複製；clone 要手動呼叫，做什麼由型別自己決定。**

4.4.3 範例程式碼

```
#[derive(Debug, Copy, Clone)]
struct Point {
    x: i32,
    y: i32,
}

fn print_point(p: Point) {
    println!("函數收到：({}, {})", p.x, p.y);
}

fn double(n: i32) -> i32 {
    n * 2
}

fn main() {
    // 整數自動 Copy
    let a = 42;
    let b = a;
    println!("a = {}, b = {}", a, b); // 兩個都能用

    // bool 也是 Copy
    let flag = true;
```

```
    let flag2 = flag;
    println!("flag = {}, flag2 = {}", flag, flag2);

    // 整數傳進函數不會 move
    let x = 10;
    let result = double(x);
    println!("x = {}, result = {}", x, result);

    // 自訂的 struct 加上 Copy 後也不會 move
    let p1 = Point { x: 3, y: 7 };
    let p2 = p1;     // 自動複製
    print_point(p1); // p1 還能用
    println!("p1 = {:?}", p1); // 還是能用！
    println!("p2 = {:?}", p2);
}
```

4.4.4 不要隨便幫自己的型別加 Copy

看完這一集，你可能會想：「那我以後每個 `struct` 都加 `#[derive(Copy, Clone)]` 不就好了？」

請不要這樣做。原因是：一旦加了 `Copy`，使用你這個型別的程式碼就會依賴「賦值時自動複製」的行為。如果有一天你需要修改這個 `struct`，加了一個不是 `Copy` 的欄位，你就必須拿掉 `Copy`。

問題來了：拿掉 `Copy` 之後，原本寫 `let p2 = p1;` 的地方全部會從「自動複製」變成「move」，`p1` 就不能再用了。所有用到這個型別的程式碼都可能因此壞掉，而且壞的地方可能很多、很分散。

所以好的習慣是：**只有當你確定這個型別永遠不會再加入非 `Copy` 欄位時，才加 `Copy`。**像 `Point { x: i32, y: i32 }` 這種就很適合。如果不確定，只加 `Clone` 就好——需要的時候手動寫 `.clone()`，未來要改也不會影響其他程式碼。

4.4.5 重點整理

- **`Copy`** 是一個 `trait`，讓型別在賦值和傳入函數時自動複製，而不是 move。
- `i32`、`f64`、`bool`、`char` 等基本型別天生就有 `Copy`。
- tuple 和陣列如果所有元素都是 `Copy`，整體也是 `Copy`。
- tuple 對一些 `trait`（`Copy`、`Clone` 等）有同樣的行為：對這些 `trait` 而言，所有元素都有實作 → tuple 就有實作。
- 自訂 `struct` 可以加 `#[derive(Copy, Clone)]`，但所有欄位都必須是 `Copy` 的型別。
- `Copy` 要求 `Clone`；使用 derive 時，通常會把兩者一起寫上。
- **`Copy` = 自動複製，`Clone` = 手動呼叫 `.clone()`**
- 不要隨便加 `Copy`——未來拿掉會讓所有依賴自動複製的程式碼壞掉。不確定就只加 `Clone`。

4.5 借用 &

4.5.1 本集目標

學會用 & 借用值，不需要 move 也不需要 `clone`，就能讓別人讀取你的資料。

4.5.2 概念說明

4.5.2.1 move 和 clone 都有代價

前面我們學了兩種方式來處理所有權：

- **move**：交出去就沒了，原本的變數不能再用。
- **clone**：建立一份副本（至少目前學過的型別都是這樣），但如果資料很大，這樣就很浪費。

有沒有辦法**不交出去、不建立副本，只是借別人看一下**？

有！這就是**借用（borrowing）**，用 & 符號。

4.5.2.2 & 就是「借」

```
    let p = Point { x: 1, y: 2 };
    let r: &Point = &p; // r 是 p 的參考，p 還是擁有者
```

&p 的意思是：「我不要拿走 p 的所有權，我只是借來看看。」p 還在，之後還可以繼續使用；至於借用期間有哪些限制，後面會整理。

&p 產生的這個東西（也就是 r）叫做**參考（reference）**，型別寫成 &Point。而「用參考去看別人的資料、但不拿走所有權」這件事，就叫做**借用**。所以借用和參考是一體兩面：**借用**是「借來看」這個動作，**參考**則是這個動作產生的憑證——拿著它就能去看那份資料。之後看到「參考」這個詞，通常指的就是用 & 借來的這個值。

4.5.2.3 函數參數用 & 就不會 move

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn print_point(p: &Point) {
    println!("({}, {})", p.x, p.y);
}

fn main() {
    let p1 = Point { x: 10, y: 20 };
    print_point(&p1); // 傳 &p1，只是借，不是 move
    println!("{:?}", p1); // p1 還在！
}
```

注意兩個地方：

1. 函數參數的型別寫 &Point（前面加 &）。
2. 呼叫時傳 &p1（也加 &）。

這樣函數只是「借」了 p1 來看，用完就還回去，p1 的所有權完全沒有改變。

4.5.2.4 之前的 & 原來是參考！

還記得之前學的 &[i32]（切片）和 &str（字串切片）嗎？當時我們說還不用深入理解，但現在我們可以說明了——那些 & 就是借用！

- &[i32] 是一段陣列資料的參考，不擁有它。

- `&str` 是一段字串資料的參考，不擁有它。

所以像這樣的函數：

```
fn sum(nums: &[i32]) -> i32 {
    let mut total = 0;
    for x in nums {
        total += x;
    }
    total
}
```

用 `for x in nums` 就能走訪切片裡的每個元素，跟之前走訪陣列的方式一樣。函數只是借用了陣列的一段切片，不會把整個陣列 move 走。

4.5.2.5 * 解參考

& 是「借」，反過來 * 就是「順著參考找到原本的值」，叫做**解參考（dereference）**：

```
fn main() {
    let x = 42;
    let r = &x;
    println!("{}", *r); // 42，和 x 一樣
}
```

不過大部分情況下你不需要手動寫 *——Rust 在用 . 存取欄位、呼叫 method、或 `println!` 的時候都會自動幫你解參考。所以目前知道有這個東西就好，下一集會用到它。

注意：前面學的 `&[T]` 和 `&str` 比較特別，你沒辦法對它們使用 * 取得一個值。原因之後會學到，現在先知道就好。

4.5.2.6 &T 都是 Copy

上一集學了 Copy——有些型別賦值的時候會自動複製，不會 move。不管 T 是什麼，&T 都是 Copy 的。畢竟參考只是借用，複製一個參考不會影響原本的資料，只是多了一個人在看而已：

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let s = Point { x: 0, y: 0 };
    let r1 = &s;
    let r2 = r1; // 複製參考，不是 move
    println!("{:?}, {:?}", r1, r2); // r1 和 r2 都能用
}
```

注意：`Point` 本身不是 Copy（賦值會 move），但 `&Point` 是 Copy。

4.5.2.7 & 參考通常是唯讀的

用 & 借用的時候，你**通常只能讀，不能直接修改**。如果你想借來直接修改，那是下一集的內容。

4.5.3 範例程式碼

```rust
#[derive(Debug, Clone)]
struct Point {
    x: i32,
    y: i32,
}

// 借用，不會 move
fn print_point(p: &Point) {
    println!("({}, {})", p.x, p.y);
}

// 切片參數就是參考
fn sum(nums: &[i32]) -> i32 {
    let mut total = 0;
    for x in nums {
        total += x;
    }
    total
}

fn main() {
    let p1 = Point { x: 10, y: 20 };

    // 借用：傳 &p1，p1 不會被 move
    print_point(&p1);
    print_point(&p1); // 可以借很多次！
    println!("p1 還在：{:?}", p1);

    // 陣列切片也是借用
    let numbers = [1, 2, 3, 4, 5];
    let total = sum(&numbers);
    println!("總和 = {}", total);
    println!("numbers 還在：{:?}", numbers);

    // &str 也是借用
    let greeting: &str = "你好";
    println!("{}", greeting);
    println!("{}", greeting); // 可以用很多次
}
```

4.5.4 重點整理

- & 是借用，**不轉移所有權**，原本的變數還能繼續用。
- 函數參數寫 &Type，呼叫時傳 &value。
- 借用可以多次進行，不像 move 只能一次。
- * 是解參考——順著參考找到原本的值（但大部分情況 Rust 會自動幫你做）。
- &[T] 和 &str 是比較特別的參考，沒辦法對它們用 * 取得值。
- &T 都是 Copy——複製一個參考不影響原本的資料。
- & 參考通常是**唯讀**的，不能直接修改借來的資料。

4.6 可變借用 &mut

4.6.1 本集目標

學會用 &mut 借用值並修改它，不需要 move 就能改變別人的資料。

4.6.2 概念說明

4.6.2.1 上一集的限制

上一集我們學了 & 借用，但拿到的參考是唯讀的——你只能看，不能改。如果我們想借別人的東西來修改呢？

4.6.2.2 &mut 就是「借來改」

```
fn main() {
    let mut x = 10;
    let r: &mut i32 = &mut x; // 可變參考
    *r = 20;                  // 透過 r 修改 x 的值
}
```

幾個重點：

1. 原本的變數必須是 let mut（因為你要改它）。
2. 借用時寫 &mut x。
3. 要透過參考去修改值，要寫 *r（上一集學的解參考——順著參考找到原本的值）。

4.6.2.3 函數參數用 &mut

更常見的用法是在函數裡：

```
fn add_ten(n: &mut i32) {
    *n += 10;
}

fn main() {
    let mut x = 5;
    add_ten(&mut x);
    println!("{}", x); // 15
}
```

函數拿到的是 &mut i32——一個**可變參考**。透過 *n 可以修改原本的值。呼叫時傳 &mut x。

4.6.2.4 struct 的可變借用

對 struct 也一樣：

```
fn move_right(p: &mut Point) {
    p.x += 1; // struct 的欄位不需要寫 *，Rust 會自動處理
}
```

注意：修改 struct 的欄位時，不需要寫 (*p).x += 1，直接寫 p.x += 1 就好——上一集提過 Rust 會自動解參考。

4.6.3 範例程式碼

```rust
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

// 透過可變參考修改整數
fn add_ten(n: &mut i32) {
    *n += 10;
}

// 透過可變參考修改 struct 的欄位
fn move_right(p: &mut Point) {
    p.x += 1;
}

fn move_up(p: &mut Point) {
    p.y += 1;
}

fn main() {
    // 修改整數
    let mut score = 80;
    println!("修改前：{}", score);
    add_ten(&mut score);
    println!("修改後：{}", score);

    // 修改 struct
    let mut pos = Point { x: 0, y: 0 };
    println!("起始位置：{:?}", pos);

    move_right(&mut pos);
    move_right(&mut pos);
    move_up(&mut pos);
    println!("移動後：{:?}", pos);

    // 直接用 &mut 修改
    let mut val = 100;
    let r = &mut val;
    *r += 50;
    println!("val = {}", val);
}
```

4.6.4 重點整理

- &mut 是可變借用，借來之後可以修改原本的值。
- 原本的變數必須是 let mut。
- 函數參數寫 &mut Type，呼叫時傳 &mut value。
- 修改 struct 欄位可以直接寫 r.field（自動解參考）。
- 下一集會學到可變借用的重要限制規則。

4.7 借用規則

4.7.1 本集目標

理解 Rust 的借用規則：同時只能有一個 `&mut` 或多個 `&`、不能 move 被借用的值，以及懸垂參考的問題。

4.7.2 概念說明

4.7.2.1 為什麼需要規則？

上一集我們學了 `&mut` 可變借用。但如果 Rust 允許你同時有多個可變參考，會怎樣？

想像你有一串鑰匙圈。借給很多人**看**（`&`）沒問題——大家都只是看，不會改變鑰匙圈上有什麼。但如果同時借給兩個人**修改**（`&mut`）——A 在加一把新鑰匙，B 同時在拆掉那把——結果就不可預測了。

這是對同一份資料互相衝突的存取，可能會導致各種奇怪的 bug。所以 Rust 制定了嚴格的借用規則。

4.7.2.2 規則一：同時只能有一個 `&mut`

在同一個時間點，一個值最多只能有**一個**可變參考：

```
    let mut x = 10;
    let r1 = &mut x;
    let r2 = &mut x; // 編譯錯誤！已經有一個 &mut 了
    *r1 += 1;
```

4.7.2.3 規則二：`&` 和 `&mut` 不能同時存在

如果有人在讀（`&`），就不能有人在改（`&mut`）；反過來也是：

```
    let mut x = 10;
    let r1 = &x;     // 唯讀借用
    let r2 = &mut x; // 編譯錯誤！已經有 & 了，不能再 &mut
    println!("{}", r1);
```

4.7.2.4 規則三：多個 `&` 可以同時存在

多個人同時讀，沒有任何問題：

```
fn main() {
    let x = 10;
    let r1 = &x;
    let r2 = &x;
    let r3 = &x;
    println!("{} {} {}", r1, r2, r3); // 完全OK
}
```

4.7.2.5 不能 move 被借用的值

前面學過，move 會讓原本的變數不能再用。所以如果一個值還被參考借著，就不能把原本的值 move 走：

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
```

```
}

fn main() {
    let p = Point { x: 1, y: 2 };
    let r = &p;

    let p2 = p; // 編譯錯誤！p 還被 r 借用中
    println!("{:?}", r);
}
```

r 後面還要使用借來的值，但 `let p2 = p;` 會把 p move 走，讓 p 從這行之後不能再用。Rust 不允許一邊保留還會被使用的參考，一邊讓原本的變數失效。

不過借用不是從建立參考開始就一路活到大括號結束。當參考最後一次被使用之後，借用就結束了，這時候再 move 就沒問題：

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 1, y: 2 };
    let r = &p;

    println!("{:?}", r); // r 最後一次使用

    let moved = p; // OK：借用已經結束
    println!("{:?}", moved);
}
```

另一種情況是：你不能從參考後面把非 Copy 的值 move 出來。

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 1, y: 2 };
    let r = &p;

    let moved = *r; // 編譯錯誤！不能從 &Point 後面 move 出 Point
    println!("{:?}", moved);
}
```

*r 是順著參考找到原本的值。Point 沒有實作 Copy，所以把它存進 moved 是 move 而不是複製。

這一條和上面那一條不一樣：上面那個只要等 r 用完就沒事了，這一條不管 r 之後用不用得到都不行。因為 p 才是這個值的擁有者，r 只是借來看的——借來的東西只能看，不能拿走。

當然，如果 *r 後面的值是 i32 這種 Copy 型別，就不一樣了：Rust 會複製一份，而不是 move，因此不違反規則。

4.7.2.6 懸垂參考（dangling reference）

還有一個重要的規則：**參考必須指向還有效的值**。如果參考還會被使用，但它指向的值已經失效，就會變成**懸垂參考**——指向一個已經不存在的地方。Rust 在編譯時期就會阻止這種事情發生。

最常見的懸垂參考，是參考從內層作用域「逃」到外面：

```
    let r;
    {
        let x = 42;
        r = &x; // x 只活在這個大括號裡
    } // x 在這裡被丟棄了
    println!("{}", r); // 編譯錯誤！r 指向的 x 已經不存在了
```

x 在大括號結束時就被丟棄了，但 r 還試圖在外面使用它——Rust 不允許。

還有一個常見的情況是函數試圖回傳區域變數的參考：

```
fn bad() -> &i32 {
    let x = 42;
    &x // x 在函數結束時就被丟棄了，參考會指向一個已經不存在的值
}
```

道理是一樣的：x 在函數結束後就消失了，回傳的參考會指向一個不存在的值。

至於 Rust 是怎麼追蹤「參考還有沒有效」的，那就是之後會學到的**生命週期**（lifetime）概念了。這裡先記住：**參考必須指向還有效的值**。

4.7.3 範例程式碼

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    // 多個不可變借用：OK
    let p = Point { x: 1, y: 2 };
    let r1 = &p;
    let r2 = &p;
    println!("r1 = {:?}, r2 = {:?}", r1, r2);

    // 一個可變參考：OK
    let mut p2 = Point { x: 10, y: 20 };
    let r3 = &mut p2;
    r3.x += 5;
    println!("修改後：{:?}", r3);

    // r3 最後一次使用之後，可變借用就結束了，現在可以用 & 借用
    let r4 = &p2;
    println!("唯讀借用：{:?}", r4);

    // 示範：同時多個唯讀借用
    let nums = [10, 20, 30, 40, 50];
    let slice1 = &nums[0..3];
```

```
    let slice2 = &nums[2..5];
    println!("slice1 = {:?}", slice1);
    println!("slice2 = {:?}", slice2);
}
```

4.7.4 重點整理

- 允許無限制地借用，同樣會讓一份資料出現互相衝突的存取，所以 Rust 制定了借用規則。
- **同一時間只能有一個 `&mut`**，不能同時有兩個可變參考。
- **`&` 和 `&mut` 不能同時存在**——要嘛大家都只讀，要嘛只有一個人在改。
- **多個 `&` 可以同時存在**，多人同時讀沒問題。
- **不能 move 被借用的值**：參考之後還會被使用時，不能把原值 move 走；參考最後一次使用後借用就結束，這時候再 move 就沒問題。
- **也不能從參考後面 move 出非 `Copy` 的值**：借來的東西只能看，不能拿走，這一條和參考之後用不用得到無關。
- **懸垂參考**：參考必須指向還有效的值，不能活得比它指向的值還久——不管是值離開了作用域，還是函數回傳區域變數的參考都不行。
- Rust 靠這些規則在編譯時期就防止存取衝突，之後會學到生命週期來更精確地追蹤參考的有效期。

4.8 `self` vs `&self` vs `&mut self`

4.8.1 本集目標

學會在 method 中選擇 `self`、`&self`、`&mut self`，以及函數參數的 `T` / `&T` / `&mut T` 怎麼選。

4.8.2 概念說明

4.8.2.1 回顧：第 3 章的 `self`

在第 3 章，我們學了 `impl` 和 method，當時所有的 method 都用 `self` 傳值：

```
impl Cat {
    fn meow(self) {
        println!("喵～");
    }
}
```

但 `self` 傳值會**消耗**這個值——呼叫完之後，原本的變數就不能再用了（因為被 move 了）。

現在我們學了借用，就可以用更聰明的方式了！

4.8.2.2 三種 `self`

寫法	意思	效果
`self`	取得所有權	呼叫後原本的變數不能再用（move）
`&self`	唯讀借用自己	呼叫後原本的變數還能用，但不能用借用改
`&mut self`	可變借用自己	呼叫後原本的變數還能用，而且可以用借用改

4.8.2.3 怎麼選？

- **只是要讀取資料** → 用 `&self`（最常用！）
- **要修改自己的欄位** → 用 `&mut self`。
- **要轉移所有權（呼叫後原本的變數不能再用）** → 用 `self`。

大部分的 method 都用 `&self`，因為你通常只是想「看看這個東西的狀態」，不需要消耗它。

4.8.2.4 實際例子：Clone

一個很好的例子是 `Clone trait`。它定義的簡化版長這樣：

```
trait Clone {
    fn clone(&self) -> Self;
}

fn main() {}
```

`clone` 接收 `&self`——只是借用自己，不消耗——然後回傳一個新的 `Self`（大寫 `Self`，第 3 章最後一集教過，代表實作這個 `trait` 的型別）。這解釋為什麼你可以對同一個變數連續呼叫好幾次 `.clone()`——因為 `clone` 只是借用，不會 move 原本的值。

如果 `clone` 的簽名是 `fn clone(self) -> Self`，那每次 `clone` 都會消耗原本的值，那就違反 `clone` 的本意了。

4.8.2.5 函數參數也一樣

不只是 method 的 `self`，一般函數參數也是同樣的邏輯：

參數型別	意思
`p: Point`	拿走所有權（move）
`p: &Point`	唯讀借用
`p: &mut Point`	可變借用

選擇的原則一樣：

- 只讀 → `&T`。
- 要改 → `&mut T`。
- 要消耗 → `T`。

4.8.3 範例程式碼

```
#[derive(Debug)]
struct Counter {
    id: i32,
    count: i32,
}

impl Counter {
    // associated function：建立新的 Counter
    fn new(id: i32) -> Self {
        Counter { id, count: 0 }
    }
```

```
    // &self：只讀
    fn get_count(&self) -> i32 {
        self.count
    }

    // &self：只讀，印出資訊
    fn display(&self) {
        println!("計數器 {}：目前計數 = {}", self.id, self.count);
    }

    // &mut self：可變借用，修改 count
    fn increment(&mut self) {
        self.count += 1;
    }

    // self：取得所有權，回傳最終結果
    fn finish(self) -> i32 {
        println!("計數器 {} 結束！最終計數 = {}", self.id, self.count);
        self.count
    }
}

// 一般函數也一樣的邏輯
fn print_counter(c: &Counter) {
    println!("（函數版）計數器 {}：{}", c.id, c.count);
}

fn reset_counter(c: &mut Counter) {
    c.count = 0;
}

fn main() {
    let mut c = Counter::new(1);

    // &self：只讀
    c.display();
    println!("目前：{}", c.get_count());

    // &mut self：修改
    c.increment();
    c.increment();
    c.increment();
    c.display();

    // 一般函數的 &T 和 &mut T
    print_counter(&c);
    reset_counter(&mut c);
    c.display();
    c.increment();
    c.increment();

    // self：取得所有權
    let final_count = c.finish();
    println!("回傳的最終計數：{}", final_count);
    // c 的所有權已經被 finish 拿走了，下面這行會編譯錯誤：
```

```
    // c.display();
}
```

4.8.4 呼叫時不用手動加 & 或 &mut

你可能注意到了——呼叫的時候我們只寫 c.display()、c.increment()，不用寫 (&c).display() 或 (&mut c).increment()。Rust 會根據 method 的 self 參數自動幫你加上 & 或 &mut。你當然也能寫 (&c).display() 或 (&mut c).increment()，但沒有必要。

4.8.5 重點整理

- &self：唯讀借用，最常用，呼叫後還能繼續用。
- &mut self：可變借用，可以修改欄位，呼叫後還能繼續用。
- self：消耗所有權，呼叫後變數就不能再用了。
- 選擇原則：**只讀** → **&self**，**要改** → **&mut self**，**要消耗** → **self**。
- Clone 裡方法的定義是 fn clone(&self) -> Self——借用自己產生一個新的 Self，所以 clone 不會消耗原本的值。
- 一般函數參數也一樣：**只讀** → **&T**，**要改** → **&mut T**，**要消耗** → **T**。
- 呼叫 method 時直接寫 c.method()。

4.9 stack 與 heap

4.9.1 本集目標

理解 stack 和 heap 的差別，並揭秘第 1 集鑰匙圈比喻的真正含義。

4.9.2 概念說明

4.9.2.1 記憶體中兩個常見的位置

程式執行時，資料會被放在記憶體裡。這一集先來認識兩個常見的存放位置：stack 和 heap。

堆疊（stack）：

- 函數被呼叫時，通常會用 stack 存放大小在編譯時期就已知的區域變數。
- 到目前為止學過的整數、浮點數、布林值、char、長度固定的陣列、tuple，以及只包含這些資料的 struct，在作為區域變數時通常都直接放在 stack 上。
- 大小在編譯時期就已知的資料不一定很小；重點是編譯器事先知道需要保留多少空間。
- 函數結束時，這次呼叫使用的 stack 空間會一起收回。

堆積（heap）：

- 程式執行時，可以視需要另外取得 heap 空間。
- 當資料量在執行時期才知道，或可能隨著程式執行而增長時，資料通常會另外放在 heap 上。例如，如果程式需要存下使用者輸入的所有數字，事前可能不知道總共有幾個。
- 程式會記住之後要到哪裡找到存放在那裡的資料。
- Rust 的所有權系統會決定何時可以歸還這塊空間。

4.9.2.2 鑰匙圈比喻揭秘！

還記得第 1 集的鑰匙圈比喻嗎？現在來揭秘它真正的意思：

- **鑰匙** = 讓程式之後能找到保險箱的資訊。
- **保險箱** = 另外存放在 heap 上的資料。
- **鑰匙圈上的裝飾** = 直接帶在鑰匙圈上的資料。

所以當我們說「move 是把鑰匙圈交出去」：

- 鑰匙和裝飾會一起交給新的擁有者。
- 保險箱本身留在原處，不需要搬動或重新建立。

4.9.2.3 為什麼整數是 Copy？

整數（i32 等）就像鑰匙圈上的裝飾。複製整數是單純、機械化的操作，所以整數實作了 Copy。

有些型別還負責管理另外存放在 heap 上的資料，因此不能用同樣的方式自動複製。這類值在賦值時會 move；需要建立副本時，則要明確呼叫 .clone()。接下來幾集會看到具體的例子。

4.9.3 範例程式碼

```
#[derive(Debug, Copy, Clone)]
struct StackData {
    x: i32,
    y: i32,
    active: bool,
}

fn main() {
    // 這些區域變數的大小在編譯時期就已知，
    // 所以這些值可以直接放在 stack 上
    let a = 42;    // i32，4 bytes
    let b = 3.14;  // f64，8 bytes
    let c = true;  // bool，1 byte
    let ch = '🦀'; // char，4 bytes
    println!("整數：{}，浮點：{}，布林：{}，字元：{}", a, b, c, ch);

    // struct 直接存放所有欄位，所以也可以直接放在 stack 上
    let data = StackData { x: 10, y: 20, active: true };
    let data2 = data; // Copy！data 還能用
    println!("data = {:?}", data);
    println!("data2 = {:?}", data2);

    // 長度固定的陣列也可以直接放在 stack 上
    let arr = [1, 2, 3, 4, 5];
    println!("陣列：{:?}", arr);

    // tuple 也可以直接放在 stack 上
    let t = (42, true, 'A');
    println!("tuple：{:?}", t);
}
```

4.9.4 重點整理

- **stack（堆疊）**：通常直接存放大小在編譯時期就已知的區域變數，例如整數、固定長度的陣列、tuple，以及由這些資料組成的 struct；函數結束時，這次呼叫使用的空間會一起收回。
- **heap（堆積）**：通常用來存放資料量在執行時期才知道或可能增長的資料；程式會記住之後要

到哪裡找到存放在那裡的資料。

- 鑰匙圈比喻揭秘：鑰匙 = 用來找到資料的資訊、保險箱 = 另外存放在 heap 上的資料、裝飾 = 直接帶著的資料。
- 整數的複製是單純、機械化的操作，所以整數實作了 Copy。
- 負責管理另外存放在 heap 上資料的型別，在賦值時會 move；需要建立副本時要明確呼叫 .clone()。

4.10 String

4.10.1 本集目標

認識 Rust 的 String 型別——一個擁有資料、可以修改的字串。

4.10.2 概念說明

4.10.2.1 之前的字串都是借來的

從第 1 章開始，我們一直在用 &str 這個型別：

```
    let greeting: &str = "你好";
```

"你好" 這個字串是直接寫在程式碼裡的，它的資料被編譯進程式本身。&str 是一個參考——你只是在看這段文字，但你**不擁有**它，也**不能修改**它。

4.10.2.2 String：你擁有的字串

String 是一個你可以擁有、可以修改的字串型別。它的資料主要存在 heap 上。

用 String::from() 來建立：

```
    let s = String::from("你好");
```

String::from 是一個 associated function（跟第 3 章學的一樣，用 :: 呼叫），它會把 &str 的內容複製一份到 heap 上，建立一個你擁有的 String。

4.10.2.3 push_str：在後面加上文字

String 可以修改！用 push_str 來接上更多文字：

```
fn main() {
    let mut s = String::from("你好");
    s.push_str("，世界！");
    println!("{}", s); // 你好，世界！
}
```

注意變數要宣告成 let mut，因為我們要修改它。

4.10.2.4 format!：組合多個值成字串

format! 跟 println! 的用法一模一樣，只是它不會印出來，而是回傳一個 String：

```
fn main() {
    let name = "小明";
    let age = 20;
    let msg = format!("我叫{}，今年{}歲", name, age);
    println!("{}", msg);
```

```
}
```

4.10.2.5 String 也適用所有權規則

因為 String 的資料主要在 heap 上，所以它**不是 Copy**。賦值和傳入函數都會 move：

```
    let s1 = String::from("hello");
    let s2 = s1; // move！s1 不能再用了
```

這跟之前學的一樣——想保留 s1 就用 .clone() 或 & 借用。

4.10.3 範例程式碼

```
fn main() {
    // 建立 String
    let mut greeting = String::from("你好");
    println!("{}", greeting);

    // push_str：接上更多文字
    greeting.push_str("，世界");
    greeting.push_str("！");
    println!("{}", greeting);

    // format!：組合多個值
    let name = "小花";
    let score = 95;
    let report = format!("{}同學的成績是{}分", name, score);
    println!("{}", report);

    // String 會 move（不是 Copy）
    let s1 = String::from("Rust");
    // let s2 = s1; // 如果這樣寫，s1 就被 move 走了，不能再用
    let s2 = s1.clone(); // 用 clone 建立一份副本，s1 還在
    println!("s1 = {}", s1);
    println!("s2 = {}", s2);

    // 傳進函數：用借用就不會 move
    let s3 = String::from("哈囉");
    print_string(&s3);
    println!("s3 還在：{}", s3);

    // Debug 格式也能用
    let s4 = String::from("debug 測試");
    println!("{:?}", s4);
}

fn print_string(s: &String) {
    println!("函數收到：{}", s);
}
```

4.10.4 重點整理

- String 是擁有資料的字串型別，資料主要存在 heap 上。
- String::from("...") 建立新的 String。

- push_str 在字串後面接上更多文字（需要 let mut）。
- format! 跟 println! 語法一樣，但回傳 String 而不是印出來。
- String **不是 Copy**，賦值和傳入函數會 move。
- 要保留原本的 String，用 .clone() 或 & 借用。

4.11 String vs &str

4.11.1 本集目標

搞清楚 String 和 &str 的差別，以及函數參數該用哪一個。

4.11.2 概念說明

4.11.2.1 兩種字串，到底差在哪？

	String	&str
擁有資料？	✔ 擁有	✘ 只是借用
資料在哪？	主要在 heap 上	可能在程式碼裡，也可能借用 String 的資料
可以修改？	✔ 可以（push_str 等）	✘ 不行
會 move？	✔ 會	✘ 不會（它就是個參考）

4.11.2.2 &String 會自動轉成 &str

當你有一個 String，想把它的參考傳給接受 &str 的函數時，Rust 會自動幫你轉換：

```
fn greet(name: &str) {
    println!("你好，{}！", name);
}

fn main() {
    let s = String::from("小明");
    greet(&s); // &String 自動轉成 &str，完全OK！
}
```

為什麼可以這樣？之後會學到。現在只要知道：**傳 &String 的地方如果參數型別是 &str，Rust 會自動處理**。

4.11.2.3 函數參數偏好 &str

如果你的函數只需要「讀」一段文字，不需要擁有它，參數型別就寫 &str：

```
fn count_chars(s: &str) -> i32 {
    let mut count = 0;
    for _c in s.chars() {
        count += 1;
    }
    count
}
```

這裡用到的 .chars() 是一個 method——String 和 &str 都有實作。它會把字串拆成一個一個字元讓你走訪。

這樣做的好處是：

1. 傳 &str（字串字面值）可以用。
2. 傳 &String 也可以用（自動轉換）。
3. 不會 move 任何東西。

這就是為什麼 Rust 社群普遍建議：**函數參數用 &str 而不是 &String**。

4.11.2.4 什麼時候用 String？

- 你需要**擁有**這段文字（存在 struct 裡、回傳給呼叫者）。
- 你需要**修改**這段文字（push_str 等）。

4.11.3 範例程式碼

```
// 參數用 &str：既能接 &str，也能接 &String
fn greet(name: &str) {
    println!("你好，{}！", name);
}

fn char_count(s: &str) -> i32 {
    let mut count = 0;
    for _c in s.chars() {
        count += 1;
    }
    count
}

fn main() {
    // &str：字串字面值
    let literal = "世界";
    greet(literal);

    // String：擁有的字串
    let owned = String::from("小花");
    greet(&owned); // &String 自動轉 &str

    // 兩種都能傳給接受 &str 的函數
    println!("「{}」有 {} 個字元", literal, char_count(literal));
    println!("「{}」有 {} 個字元", owned, char_count(&owned));

    // String 可以修改，&str 不行
    let mut s = String::from("Rust");
    s.push_str(" 好好玩");
    println!("{}", s);

    // String 會 move
    let s1 = String::from("hello");
    let s2 = s1; // move
    // println!("{}", s1); // 編譯錯誤！
    println!("{}", s2);

    // &str 不會 move（它本身就是參考）
    let greeting: &str = "哈囉";
    let greeting2 = greeting;  // 這是 Copy！(&str 是 Copy 的)
    println!("{}", greeting);  // OK
    println!("{}", greeting2); // OK
```

```
}
```

4.11.4 重點整理

- **String** 擁有資料（主要在 heap 上），可以修改，會 move。
- **&str** 是參考，不擁有資料，不能修改，不會 move。
- &String 會自動轉成 &str。
- 函數參數偏好用 &str——接受範圍更廣（&str 和 &String 都能傳）。
- 需要擁有或修改字串時才用 String。

4.12 Vec 基礎

4.12.1 本集目標

學會使用 Vec——一個可以動態增長的陣列。

4.12.2 概念說明

4.12.2.1 陣列的限制

我們在第 2 章學了陣列 [i32; 5]，但陣列的大小是固定的——宣告時就決定了，之後不能加東西也不能減東西。

如果我們需要一個**大小可以變化**的集合呢？比如：使用者一筆一筆輸入資料，或者程式在執行過程中不斷累積結果。

這就需要 **Vec**。Vec 就像一個**可以伸縮的陣列**，資料主要存在 heap 上。

4.12.2.2 建立 Vec

最簡單的方式是用 vec! 巨集：

```
    let nums = vec![1, 2, 3, 4, 5];
```

這樣就建立了一個包含 5 個 i32 的 Vec。Rust 會根據你放的值自動推斷型別。

跟陣列的 [0; 5] 類似，vec! 也支援「重複 N 次」的寫法：

```
    let zeros = vec![0; 10]; // 10 個 0
```

你也可以建立空的 Vec，然後一個一個加：

```
    let mut nums = Vec::new();
    nums.push(10);
    nums.push(20);
```

Rust 會在你第一次 push 的時候推斷出型別。

4.12.2.3 索引和走訪

Vec 的索引跟陣列一樣，用 [i]：

```
fn main() {
    let nums = vec![10, 20, 30];
    println!("{}", nums[0]); // 10
    println!("{}", nums[2]); // 30
```

```
}
```

走訪也跟陣列一樣，用 for：

```
fn main() {
    let nums = vec![10, 20, 30];
    for n in &nums {
        println!("{}", n);
    }
}
```

注意：走訪的時候用 &nums（借用），這樣 nums 不會被 move 走。下一集會詳細說明。

4.12.2.4 push：加入新元素

```
fn main() {
    let mut fruits = Vec::new();
    fruits.push("蘋果");
    fruits.push("香蕉");
    fruits.push("櫻桃");
    println!("{:?}", fruits);
}
```

push 會把新元素加到最後面。注意 Vec 必須是 let mut 才能 push。

4.12.2.5 len：取得長度

```
fn main() {
    let nums = vec![1, 2, 3];
    println!("長度：{}", nums.len());
}
```

4.12.3 範例程式碼

```
fn main() {
    // 用 vec! 建立
    let scores = vec![85, 92, 78, 95, 88];
    println!("成績：{:?}", scores);
    println!("第一筆：{}", scores[0]);
    println!("共 {} 筆", scores.len());

    // 空的 Vec，用 push 加入
    let mut names = Vec::new();
    names.push("小明");
    names.push("小花");
    names.push("阿旺");
    println!("名單：{:?}", names);

    // 走訪
    println!("逐一列出：");
    for name in &names {
        println!("  - {}", name);
    }

    // 用 for 走訪並加總
```

```
    let nums = vec![10, 20, 30, 40, 50];
    let mut total = 0;
    for x in &nums {
        total += x;
    }
    println!("總和 = {}", total);

    // Vec 可以一直 push
    let mut growing = Vec::new();
    for i in 0..5 {
        growing.push(i * 10);
    }
    println!("動態建立：{:?}", growing);
}
```

4.12.4 重點整理

- Vec 是可以動態增長的陣列，資料主要存在 heap 上。
- vec![1, 2, 3] 建立有初始值的 Vec；vec![0; 10] 建立 10 個 0（跟陣列的 [0; 10] 類似）。
- Vec::new() 建立空的 Vec。
- push 在最後面加入元素（需要 let mut）。
- 索引用 v[0]、v[1] 等，長度用 v.len()（method，回傳元素個數）。
- 走訪用 for x in &v（借用，不 move）。
- Vec 和陣列的操作方式很像，但 Vec 大小可以變化。

4.13 Vec 與所有權

4.13.1 本集目標

理解 Vec 的所有權行為，以及它和 String / &str 的對稱關係。

4.13.2 概念說明

4.13.2.1 Vec 和 String 是一對

在前面幾集，我們學了 String 和 &str 的關係：

擁有版本	借用版本
String	&str

Vec 也有完全一樣的對應：

擁有版本	借用版本
Vec	&[T]（切片）

String 擁有一段文字，&str 借用一段文字。Vec 擁有一組元素，&[T] 借用一組元素。**概念完全對稱。**

4.13.2.2 Vec 會 move

Vec 的資料主要在 heap 上，所以它不是 Copy。賦值和傳入函數都會 move：

```
    let v1 = vec![1, 2, 3];
    let v2 = v1; // move！v1 不能再用了
```

跟 String 一模一樣。

4.13.2.3 函數參數用切片 &[T]

跟 String / &str 的建議一樣——如果函數只需要讀取 i32 的 Vec 的內容，用切片 &[i32]：

```
fn sum(nums: &[i32]) -> i32 {
    let mut total = 0;
    for x in nums {
        total += x;
    }
    total
}

fn main() {
    let v = vec![1, 2, 3, 4, 5];
    let total = sum(&v); // i32 的 &Vec 自動轉成 &[i32]
    println!("總和：{}", total);
    println!("v 還在：{:?}", v);
}
```

就像 &String 會自動轉成 &str，i32 的 &Vec 也會自動轉成 &[i32]。

4.13.2.4 for 迴圈與所有權

這是很重要的一點：for 迴圈走訪 Vec 時，可以選擇要 move 還是 borrow：

for x in v——move！

```
fn main() {
    let v = vec![1, 2, 3];
    for x in v {
        println!("{}", x);
    }
    // v 被 move 走了，不能再用！
}
```

for x in v 會消耗整個 Vec。迴圈結束後，v 就不存在了。

for x in &v——borrow！

```
fn main() {
    let v = vec![1, 2, 3];
    for x in &v {
        println!("{}", x); // x 的型別是 &i32
    }
    println!("v 還在：{:?}", v); // OK！
}
```

for x in &v 只是借用，v 不會被消耗。

這裡要特別注意：x 不是 i32，而是參考，型別是 &i32。因為迴圈走訪的是借來的 &v，所以每次拿

到的元素也只是借來的，不會把元素移出 Vec。同樣地，前面函數參數的 nums: &[i32] 本身已經是借來的切片，所以 for x in nums 裡的 x 也是 &i32。

大部分情況你應該用 for x in &v，除非你確定不再需要這個 Vec。

4.13.3 範例程式碼

```
// 參數用切片：i32 的 &Vec 自動轉 &[i32]
fn sum(nums: &[i32]) -> i32 {
    let mut total = 0;
    for x in nums {
        total += x;
    }
    total
}

fn print_all(nums: &[i32]) {
    let mut first = true;
    for x in nums {
        if first {
            first = false;
        } else {
            print!(", ");
        }
        print!("{}", x);
    }
    println!();
}

fn main() {
    // Vec 會 move
    let v1 = vec![10, 20, 30];
    let v2 = v1.clone(); // clone 保留 v1
    println!("v1 = {:?}", v1);
    println!("v2 = {:?}", v2);

    // 函數用切片參數（借用）
    let scores = vec![85, 92, 78, 95, 88];
    println!("總分 = {}", sum(&scores));
    print_all(&scores);
    println!("scores 還在：{:?}", scores);

    // 切片操作
    let slice = &scores[1..4]; // 借用一部分
    println!("中間三筆：{:?}", slice);
    println!("中間三筆的總分 = {}", sum(slice));

    // for x in &v：借用走訪
    println!("逐一列出（借用）：");
    for s in &scores {
        println!("  {}", s);
    }
    println!("scores 還在：{:?}", scores);

    // for x in v：move 走訪（用完就沒了）
```

```
    let temp = vec![1, 2, 3];
    println!("消耗走訪：");
    for x in temp {
        println!("  {}", x);
    }
    // temp 已經被 move 了，下面會編譯錯誤：
    // println!("{:?}", temp);

    // 對稱關係整理
    // String  ↔ &str      （擁有 ↔ 借用 文字）
    // Vec     ↔ &[T]      （擁有 ↔ 借用 一組值）
    println!("--- 對稱關係 ---");
    let s = String::from("hello");
    let s_ref: &str = &s; // &String → &str
    println!("String: {}, &str: {}", s, s_ref);

    let v = vec![1, 2, 3];
    let v_ref: &[i32] = &v; // i32 的 &Vec → &[i32]
    println!("Vec: {:?}, slice: {:?}", v, v_ref);
}
```

4.13.4 「裝 i32 的 Vec」的型別怎麼寫？

這一集我們一直說「i32 的 Vec」，但你可能已經注意到：程式碼裡從來沒把這個型別寫出來過——變數的型別都靠 Rust 自己推導，函數參數也只寫了切片 &[i32]。如果哪天你需要親手寫出它（比如當函數的參數或回傳值型別），該怎麼寫？還有上面表格裡 &[T] 的那個 T，到底可以是什麼？這兩個問題的答案是同一個機制——下一章會花大量篇幅討論。

4.13.5 重點整理

- **Vec 和 String 的所有權行為完全對稱**：資料都主要在 heap 上、都會 move、都可以 clone。
- String ↔ &str 就像 Vec ↔ &[T]（擁有 ↔ 借用）。
- &Vec 會自動轉成 &[T]（跟 &String 自動轉 &str 一樣）。
- 函數參數偏好用切片 &[T] 而不是 &Vec。
- for x in v：**move**，消耗整個 Vec。
- for x in &v：**borrow**，只是借用，Vec 還在；這個例子裡的 x 是參考，型別是 &i32。
- 大部分情況用 for x in &v，除非你確定不再需要這個 Vec。
- 我們始終沒把「裝 i32 的 Vec」的型別親手寫出來——怎麼寫，以及 &[T] 的 T 是什麼，下一章揭曉。

恭喜你完成了第 4 章！🎉 這一章你學會了 Rust 最核心的概念——所有權、move、clone、Copy、borrowing，還有 String 和 Vec 這兩個最常用的非 Copy 型別。這些概念是 Rust 和其他語言最大的不同，也是 Rust 能在不需要犧牲效能的情況下保證記憶體安全的關鍵。下一章我們將進入泛型、trait bound 和生命週期——讓你的程式碼能處理任意型別，同時保持型別安全！

第 5 章

泛型、Trait Bound 與生命週期

如果說第 3 章講的是建立高階抽象供人類理解，而第 4 章講的是配合現代電腦硬體制定限制以達成效率，這章便是要延伸並結合前兩章的內容。在建立抽象上我們導入能接收型別作為參數的型別——泛型，然後我們會學習能限制泛型的 `trait`，接著理解生命週期不但可以達成作為低階語言的效能，也如在高階語言中和型別系統完美結合。

5.1 泛型函數

5.1.1 本集目標

學會用 <T> 定義泛型函數，讓同一個函數可以處理不同型別。

5.1.2 概念說明

在第 4 章，我們學了 `Vec`，用它來存一堆 `i32`。但你有沒有注意到，我們總是寫 `vec![1, 2, 3]`，讓 Rust 自己推斷型別？

其實，`Vec` 不是一個完整的型別。它的完整寫法是 `Vec<i32>`、`Vec<String>`、`Vec<bool>`——角括號 <> 裡面放的是「這個 `Vec` 要裝什麼型別的東西」。

第 4 章故意沒提角括號，因為當時我們還沒學過泛型。現在，是時候揭開這個秘密了。

5.1.2.1 什麼是泛型？

假設你想寫一個函數，傳入兩個值，回傳第一個：

```
fn first_i32(a: i32, b: i32) -> i32 {
    a
}
```

如果又需要處理 `f64` 呢？難道要再寫一個 `first_f64`？

泛型就是解決這個問題的。我們用一個「型別參數」`T` 來代替具體的型別：

```
fn first<T>(a: T, b: T) -> T {
    a
}
```

這裡的 <T> 寫在函數名後面，表示「這個函數有一個型別參數叫 `T`」。然後參數 `a` 和 `b` 的型別都是 `T`，回傳值也是 `T`。

當你呼叫 `first(10, 20)` 的時候，Rust 看到 `10` 是 `i32`，就知道 `T = i32`。呼叫 `first(3.14, 2.71)` 的時候，`T = f64`。同一個函數定義，自動適用於不同型別。

5.1.2.2 命名慣例

型別參數通常用單個大寫字母：`T`（Type）、`U`、`V`。如果有語意的話也會用比較長的名字，但目前用 `T` 就好。

5.1.3 範例程式碼

```
// 泛型函數：回傳兩個值中的第一個
fn first<T>(a: T, _b: T) -> T {
    a
}

// 可以有多個型別參數
fn make_pair<T, U>(a: T, b: U) -> (T, U) {
    (a, b)
}

fn main() {
    // T 被推斷為 i32
    let x = first(10, 20);
    println!("{}", x);

    // T 被推斷為 &str
    let y = first("hello", "world");
    println!("{}", y);

    // 但兩個參數的型別必須一樣，因為 first 的兩個參數都是 T
    // let bad = first(1, "a"); // 編譯錯誤！1 是 i32 但 "a" 是 &str

    // T = i32, U = &str
    let pair = make_pair(42, "hello");
    println!("{:?}", pair);
}
```

5.1.4 重點整理

- `Vec` 的完整寫法是 `Vec<T>`，角括號裡放型別參數——第 4 章故意省略，現在正式學習。
- 泛型函數用 `<T>` 宣告型別參數，讓同一個函數可以處理不同型別。
- Rust 會根據傳入的值自動推斷 `T` 是什麼型別。
- 可以有多個型別參數：`<T, U>`。
- 型別參數慣例用大寫字母：`T`、`U`、`V`。

5.2 泛型 `struct`

5.2.1 本集目標

學會定義帶型別參數的 `struct`，讓同一個結構可以存放不同型別的資料。

5.2.2 概念說明

上一集我們學了泛型函數。其實 `struct` 也可以有型別參數！

舉例來說，`Vec<T>` 本身就是一個泛型 `struct`：`Vec<i32>` 和 `Vec<String>` 使用同一個 `struct` 定

義，只是裡面放的型別不同。我們也可以自己定義這樣的泛型 struct。

5.2.2.1 定義泛型 struct

```
struct Pair<T> {
    first: T,
    second: T,
}
```

這裡的 <T> 寫在 struct 名稱後面，表示「Pair 有一個型別參數 T」。first 和 second 的型別都是 T，所以它們必須是同一種型別。

使用的時候：

```
    let p = Pair { first: 1, second: 2 };       // T = i32
    let q = Pair { first: "hi", second: "yo" }; // T = &str
```

5.2.2.2 多個型別參數

如果你希望 first 和 second 可以是不同型別，就用兩個型別參數：

```
struct MixedPair<T, U> {
    first: T,
    second: U,
}
```

這和上一集的 make_pair<T, U> 概念一模一樣。

5.2.3 範例程式碼

```
// 兩個欄位必須同型別
#[derive(Debug)]
struct Pair<T> {
    first: T,
    second: T,
}

// 兩個欄位可以不同型別
#[derive(Debug)]
struct MixedPair<T, U> {
    first: T,
    second: U,
}

fn main() {
    let int_pair = Pair { first: 10, second: 20 };
    println!("{:?}", int_pair);

    let str_pair = Pair { first: "hello", second: "world" };
    println!("{:?}", str_pair);

    // Pair<T> 的兩個欄位必須同型別，以下會編譯錯誤：
    // let bad = Pair { first: 42, second: "oops" };

    let mixed = MixedPair { first: 42, second: "answer" };
    println!("{:?}", mixed);
```

```
}
```

5.2.4 重點整理

- `struct` 可以用 `<T>` 宣告型別參數，讓同一個定義適用於不同型別。
- `Pair<T>` 的兩個欄位都是 `T`，所以必須同型別。
- 需要不同型別時，用多個型別參數：`MixedPair<T, U>`。
- 和泛型函數一樣，Rust 會根據使用方式自動推斷型別參數。

5.3 泛型 enum

5.3.1 本集目標

學會定義帶型別參數的 `enum`。

5.3.2 概念說明

上一集學了泛型 `struct`，這一集來看泛型 `enum`。其實概念完全一樣——在 `enum` 名稱後面加 `<T>`，讓 variant 攜帶的資料可以是任何型別。

5.3.2.1 定義泛型 enum

假設我們想做一個「也許有值」的型別，裡面可能有東西，也可能是空的：

```
enum Maybe<T> {
    Something(T),
    Nothing,
}
```

`Something(T)` 攜帶一個 `T` 型別的值，`Nothing` 什麼都不帶。

泛型 `enum` 也可以有多個型別參數。比如一個「二選一」的型別：

```
enum Either<L, R> {
    Left(L),
    Right(R),
}
```

`Either<L, R>` 要嘛是 `Left(L)`，要嘛是 `Right(R)`——兩個型別完全獨立。

5.3.3 範例程式碼

```
// 自己定義的泛型 enum
#[derive(Debug)]
enum Maybe<T> {
    Something(T),
    Nothing,
}

// 兩個型別參數的泛型 enum
#[derive(Debug)]
enum Either<L, R> {
    Left(L),
    Right(R),
```

```
}

fn main() {
    let a: Maybe<i32> = Maybe::Something(42);
    let b: Maybe<i32> = Maybe::Nothing;

    println!("{:?}", a);
    println!("{:?}", b);

    // 用 match 取出值
    match a {
        Maybe::Something(val) => println!("裡面有：{}", val),
        Maybe::Nothing => println!("空的"),
    }

    // 兩個型別參數
    let x: Either<i32, &str> = Either::Left(100);
    let y: Either<i32, &str> = Either::Right("hello");

    println!("{:?}", x);
    println!("{:?}", y);
}
```

5.3.4 重點整理

- enum 也可以帶型別參數：`enum Maybe<T> { ... }`。
- variant 攜帶的資料型別可以用 T 來泛化。
- 可以有多個型別參數：`enum Either<L, R> { Left(L), Right(R) }`。
- 標準庫裡有很多重要的泛型 enum，之後會陸續認識。

5.4 turbofish 語法

5.4.1 本集目標

學會用 `::<>` turbofish 語法手動指定型別參數，理解它和泛型定義的關係。

5.4.2 概念說明

前幾集我們學了泛型——函數、struct、enum 都可以有型別參數 <T>。大部分時候 Rust 能自動推斷 T 是什麼，但有時候編譯器推不出來，就需要我們手動告訴它。

5.4.2.1 turbofish 是什麼？

還記得第 1 章學 parse 的時候，我們寫過這樣的程式碼嗎？

```
fn main() {
    let input = "1";
    let num = input.trim().parse::<i32>().expect("不是數字");
}
```

當時我們把 `::<i32>` 當黑盒子照抄。現在學了泛型，終於可以理解它了！

`.parse()` 是一個泛型方法，有一個型別參數 T，代表「你想把字串轉成什麼型別」。但光看

`input.trim().parse()` 這段程式碼，編譯器不知道你想轉成 `i32` 還是 `f64` 還是其他東西。

所以我們用 `::<i32>` 手動指定 `T = i32`。這個 `::<>` 語法就叫做 **turbofish**（因為 `::<>` 看起來像一條魚 🐟）。

5.4.2.2 turbofish 的本質

turbofish 就是「手動填入泛型定義裡角括號的型別參數」：

- 泛型定義：`fn parse<T>(...)`——這裡的 `<T>` 是宣告。
- turbofish：`.parse::<i32>()`——這裡的 `::<i32>` 是填入。

函數、方法、型別都可以用 turbofish：

```
// 函數的 turbofish
func::<i32>(arg);

// 型別的 turbofish
Vec::<i32>::new();
```

5.4.2.3 `.parse()` 做了什麼？

順便完整解釋一下 `parse`：它把字串轉換成你指定的型別。轉換可能失敗（比如 `"abc"` 不能轉成數字），所以需要搭配 `.expect()` 處理失敗的情況——這點在第 1 章就用過了。

5.4.3 範例程式碼

```
fn first<T>(a: T, _b: T) -> T {
    a
}

fn main() {
    // 通常 Rust 能自動推斷，不需要 turbofish
    let x = first(10, 20);
    println!("{}", x);

    // 手動用 turbofish 指定型別
    let y = first::<f64>(3.14, 2.71);
    println!("{}", y);

    // Vec 的 turbofish
    let v = Vec::<i32>::new();
    println!("{:?}", v);

    // parse 的 turbofish——呼應第 1 章的黑盒子
    let input = "42";
    let num = input.parse::<i32>().expect("不是數字");
    println!("{}", num);

    let pi = "3.14".parse::<f64>().expect("不是數字");
    println!("{}", pi);
}
```

5.4.4 重點整理

- **turbofish** `::<>` 是手動指定泛型型別參數的語法。

- 大部分時候 Rust 能自動推斷，不需要 turbofish。
- 當編譯器推不出型別時（例如 `.parse()`），就需要用 turbofish 手動指定。
- 第 1 章寫的 `.parse::<i32>()` 其實就是 turbofish——現在我們理解它為什麼這樣寫了。
- `.parse()` 把字串轉成指定型別，轉換可能失敗所以搭配 `.expect()` 使用。

5.5 placeholder type _

5.5.1 本集目標

學會用 _ 在型別標注中讓編譯器推斷部分型別。

5.5.2 概念說明

上一集學了 turbofish，可以手動指定所有型別參數。但有時候你只想指定一部分，剩下的讓 Rust 自己推斷。這時候就用 _ 作為型別層級的萬用字元。

5.5.2.1 _ 當型別佔位符

看這個例子：

```
fn main() {
    let v: Vec<_> = vec![1, 2, 3];
}
```

這裡我們告訴 Rust「這是一個 Vec」，但裡面的元素型別用 _ 表示「你自己推斷吧」。Rust 看到 1, 2, 3 是整數，就推斷 _ = i32。

turbofish 裡也可以用 _：

```
    let v = Vec::<_>::new();
```

不過這樣寫其實和直接寫 Vec::new() 讓 Rust 全部推斷沒什麼差別。_ 更常用在你需要指定外層型別、但內層型別讓 Rust 推斷的情況。

5.5.2.2 什麼時候有用？

當型別有多個參數，你只想標注一部分的時候。_ 的威力在型別越複雜時越明顯——之後學到更多標準庫型別時會自然體會到。

5.5.3 範例程式碼

```
fn main() {
    // 用 _ 讓 Rust 推斷 Vec 的元素型別
    let v: Vec<_> = vec![1, 2, 3];
    println!("{:?}", v);

    // turbofish 裡也能用 _
    let v2 = Vec::<_>::new(); // 和 Vec::new() 一樣，_ 讓 Rust 推斷
    let v2: Vec<i32> = v2;    // 之後透過使用方式確定型別
    println!("{:?}", v2);

    // 比較：完全不標型別 vs 用 _ 部分標注
    let a = vec![true, false]; // Rust 全部推斷：Vec<bool>
    let b: Vec<_> = vec![true, false]; // 告訴 Rust 是 Vec，元素型別自己推斷
```

```
    println!("{:?}", a);
    println!("{:?}", b);
}
```

5.5.4 重點整理

- _ 可以在型別標注中當佔位符，讓 Rust 推斷該位置的型別。
- 適合用在「外層型別我知道，內層讓 Rust 推斷」的情況。
- turbofish 和 `let` 標注都可以使用 _。

5.6 型別別名

5.6.1 本集目標

學會用 `type` 為型別建立別名，讓複雜的泛型型別變得更好讀。

5.6.2 概念說明

隨著我們學了泛型，型別會越來越複雜。比如一個三維的資料結構：

```
Vec<Vec<Vec<i32>>>
```

每次都寫完整型別有點累，而且不好讀。Rust 提供了 `type` 關鍵字來建立**型別別名**：

```
type Grid3D = Vec<Vec<Vec<i32>>>;
```

從此以後，`Grid3D` 和 `Vec<Vec<Vec<i32>>>` 就是同一個型別——只是換了個名字。它不會建立新型別，就只是一個簡寫。

5.6.2.1 簡單的別名

```
type Name = String;
```

`Name` 和 `String` 完全等價，可以互換使用。

5.6.2.2 帶參數的型別別名

型別別名也可以帶泛型參數：

```
type Pair<T> = (T, T);
```

這樣 `Pair<i32>` 就等於 `(i32, i32)`，`Pair<String>` 就等於 `(String, String)`。

5.6.2.3 注意

型別別名只是簡寫，不是新型別。`Name` 和 `String` 完全可以互換使用，編譯器視它們為同一個型別。

5.6.3 範例程式碼

```
// 簡單的型別別名
type Name = String;

// 簡化複雜的巢狀型別
type Grid3D = Vec<Vec<Vec<i32>>>;
```

```
// 帶泛型參數的別名
type Pair<T> = (T, T);

fn main() {
    // Name 就是 String
    let greeting: Name = String::from("你好");
    println!("{}", greeting);

    // 三維 Vec 用別名就很清爽
    let mut grid: Grid3D = vec![vec![vec![0; 3]; 3]; 3];
    grid[1][1][1] = 42;
    println!("grid[1][1][1] = {}", grid[1][1][1]);

    // Pair<i32> 就是 (i32, i32)
    let point: Pair<i32> = (3, 7);
    println!("{:?}", point);

    let coords: Pair<f64> = (1.5, 3.7);
    println!("{:?}", coords);
}
```

5.6.4 重點整理

- `type Name = ExistingType;` 建立型別別名，只是簡寫，不是新型別。
- 型別別名可以帶泛型參數：`type Pair<T> = (T, T);`。
- 常見用途：簡化複雜的巢狀型別（如 `Vec<Vec<Vec<i32>>>`）。
- 別名和原型別完全等價，可以互換使用。

5.7 泛型 impl

5.7.1 本集目標

學會為泛型 struct 實作方法，理解 `impl<T>` 語法中兩個 T 的含義。

5.7.2 概念說明

第 2 集我們定義了泛型 struct `Pair<T>`。這集要幫它 impl。

回想第 3 章，impl struct 是這樣寫的：

```
impl Point {
    fn sum(&self) -> i32 {
        self.x + self.y
    }
}
```

那泛型 struct 呢？

5.7.2.1 impl<T> 的語法

```
impl<T> Pair<T> {
    fn new(first: T, second: T) -> Pair<T> {
        Pair { first, second }
```

```
    }
}
```

注意這裡有**兩個** T 出現在不同位置，它們的角色不一樣：

1. `impl<T>` 的 `<T>`：**宣告**一個型別參數 T。告訴 Rust「接下來我要用一個叫 T 的型別參數」。
2. `Pair<T>` 的 `<T>`：**使用**剛才宣告的 T。告訴 Rust「我要幫的是 `Pair<T>` 這個型別」。

換句話說：`impl<T>` 宣告 T，然後把 T 傳給 `Pair<T>`——「對於任何型別 T，幫 `Pair<T>` 實作以下方法」。

如果你只寫 `impl Pair<T>` 而不加 `impl<T>`，Rust 會以為 T 是一個具體的型別名稱（就像 `i32` 或 `String` 一樣），然後找不到叫 T 的型別就報錯。

反過來，如果你寫 `impl Pair<i32>`（不需要 `impl<T>`），那就是只幫 `Pair<i32>` 這一種加方法，`Pair<String>` 或其他的都不會有。

5.7.2.2 方法裡使用 T

宣告了 T 之後，在整個 `impl` 區塊裡都可以使用它：

```
impl<T> Pair<T> {
    fn new(first: T, second: T) -> Pair<T> {
        Pair { first, second }
    }

    fn first(&self) -> &T {
        &self.first
    }
}
```

5.7.2.3 實作 trait 也是一樣

第 4 章教 `trait` 的時候，我們幫具體型別實作了 `trait`，像 `impl Greet for Cat`。如果你想幫一個泛型型別實作 `trait`，語法也是一樣——在 `impl` 後面加上 `<T>` 來宣告型別參數：

```
impl<T> SomeTrait for Pair<T> {
    // ...
}
```

一樣是「對於任何型別 T，幫 `Pair<T>` 實作這個 `trait`」。

5.7.3 範例程式碼

```
#[derive(Debug)]
struct Pair<T> {
    first: T,
    second: T,
}

impl<T> Pair<T> {
    // associated function
    fn new(first: T, second: T) -> Pair<T> {
        Pair { first, second }
    }
```

```
    // method：回傳 first 的參考
    fn first(&self) -> &T {
        &self.first
    }

    // method：回傳 second 的參考
    fn second(&self) -> &T {
        &self.second
    }
}

fn main() {
    let p = Pair::new(10, 20);
    println!("first = {}", p.first());
    println!("second = {}", p.second());
    println!("{:?}", p);

    let q = Pair::new("hello", "world");
    println!("first = {}", q.first());
    println!("second = {}", q.second());
}
```

5.7.4 重點整理

- 為泛型 struct 實作方法時，寫 `impl<T> Pair<T> { ... }`。
- `impl<T>` 的 `<T>` 是**宣告** T，`Pair<T>` 的 `<T>` 是**使用** T。
- 結論：`impl<T>` 宣告 T，然後把 T 傳給 `Pair<T>`。
- 宣告之後，整個 `impl` 區塊裡的方法都可以使用 T。
- 幫泛型型別實作 trait 語法類似：`impl<T> SomeTrait for Pair<T> { ... }`。

5.8 Option<T>

5.8.1 本集目標

認識 Rust 標準庫最重要的泛型 enum——`Option<T>`，理解它如何取代 null 並防止執行時期錯誤。

5.8.2 概念說明

5.8.2.1 null 的問題

在某些程式語言裡，任何變數都可能是 null（空值）。這導致一個經典問題：你以為變數有值，用了它，結果執行時炸掉——「null pointer exception」。null 的發明者 Tony Hoare 甚至稱它為「十億美金的錯誤」。

Rust 的解法很簡單：**沒有 null。**

取而代之的是一個泛型 enum：`Option<T>`。

5.8.2.2 Option 的定義

`Option<T>` 長這樣（標準庫已經幫你定義好了）：

```
enum Option<T> {
    Some(T),
    None,
}
```

看起來是不是很像第 3 集我們自己寫的 Maybe<T>？沒錯！概念完全一樣：

- Some(T) 表示「有一個 T 型別的值」。
- None 表示「沒有值」。

5.8.2.3 強制處理 None

Option 的厲害之處在於：編譯器**強制**你處理「沒有值」的情況。你不能直接把 Option<i32> 當成 i32 來用，必須先檢查它到底是 Some 還是 None。

這就是用 match 的時候了：

```
    match maybe_value {
        Some(v) => println!("有值：{}", v),
        None => println!("沒有值"),
    }
```

5.8.2.4 Option 不用寫完整路徑

因為 Option、Some、None 實在太常用了，Rust 預設就把它們引入到每個檔案裡。所以你不需要寫 Option::Some(42)，直接寫 Some(42) 就好。

5.8.2.5 零成本的秘密：niche optimization

一個有趣的小知識：Option<&T> 和普通的參考 &T 佔用一樣大的記憶體！

因為參考 &T 不可能是 null，所以 Rust 在記憶體中聰明地用 null 來代表 None，不需要額外的空間。這叫做 **niche optimization**——利用型別中「不可能出現的值」來塞額外的資訊。

5.8.3 範例程式碼

```
// 在切片中找到第一個偶數，找不到就回傳 None
fn find_even(numbers: &[i32]) -> Option<i32> {
    for n in numbers {
        if n % 2 == 0 {
            return Some(*n);
        }
    }
    None
}

fn main() {
    let nums = vec![1, 3, 5, 8, 11];
    let result = find_even(&nums);

    // 用 match 取出 Option 的值
    match result {
        Some(n) => println!("找到偶數：{}", n),
        None => println!("沒有偶數"),
    }
```

```
    let odds = vec![1, 3, 5, 7];
    let result2 = find_even(&odds);

    match result2 {
        Some(n) => println!("找到偶數：{}", n),
        None => println!("沒有偶數"),
    }
}
```

5.8.4 重點整理

- Option<T> 是 Rust 用來表達「可能沒有值」的泛型 enum，取代了其他語言的 null。
- Some(T) 表示有值，None 表示沒有值。
- 編譯器強制你處理 None 的情況，執行時期不會有 null pointer exception。
- Option、Some、None 太常用，Rust 預設就引入了，不需要額外路徑。
- niche optimization：Option<&T> 和 &T 大小相同，零額外成本。

5.9 Option 常用方法

5.9.1 本集目標

學會 Option 的常用方法：unwrap、expect、unwrap_or、flatten，以及用 if let 取值。

5.9.2 概念說明

上一集我們用 match 來處理 Option，這是最安全的方式。但每次都寫 match 有時候太囉嗦了。Rust 提供了一些方便的方法。

5.9.2.1 unwrap：暴力取值

```
    let x: Option<i32> = Some(42);
    let value = x.unwrap(); // 42
```

如果是 Some，直接拿到裡面的值。但如果是 None，程式會 panic（崩潰）！所以 unwrap 要小心用——通常是在你確定不會是 None 的時候使用。

5.9.2.2 expect：帶訊息的 unwrap

```
    let x: Option<i32> = None;
    let value = x.expect("不應該是 None"); // panic，印出你的訊息
```

和 unwrap 一樣，但 panic 時會印出你自訂的訊息，方便除錯。

5.9.2.3 unwrap_or：提供預設值

```
    let x: Option<i32> = None;
    let value = x.unwrap_or(0); // 0
```

如果是 Some 就取出值，如果是 None 就用你給的預設值。不會 panic，很安全。

5.9.2.4 flatten：把巢狀 Option 壓平

有時候你會碰到 Option<Option<T>> 這種巢狀結構：

```
    let nested: Option<Option<i32>> = Some(Some(42));
    let flat: Option<i32> = nested.flatten(); // Some(42)
```

flatten 把兩層 Option 壓成一層。如果外層或內層是 None，結果就是 None。

5.9.3 範例程式碼

```
fn find_even(numbers: &[i32]) -> Option<i32> {
    for n in numbers {
        if n % 2 == 0 {
            return Some(*n);
        }
    }
    None
}

fn main() {
    let nums = [1, 3, 5, 7];
    let has_even = [2, 4, 6];

    // unwrap_or：安全地提供預設值
    let result = find_even(&nums).unwrap_or(0);
    println!("偶數（沒找到就給 0）：{}", result);

    // expect：確定有值時使用
    let result2 = find_even(&has_even).expect("應該要有偶數");
    println!("找到偶數：{}", result2);

    // if let：第 3 章學過的語法
    if let Some(n) = find_even(&has_even) {
        println!("用 if let 取出：{}", n);
    }

    // flatten：壓平巢狀 Option
    let nested: Option<Option<i32>> = Some(Some(42));
    let flat = nested.flatten();
    println!("{:?}", flat);

    let nested_none: Option<Option<i32>> = Some(None);
    let flat_none = nested_none.flatten();
    println!("{:?}", flat_none);

    let outer_none: Option<Option<i32>> = None;
    let flat_outer = outer_none.flatten();
    println!("{:?}", flat_outer);
}
```

5.9.4 重點整理

- unwrap()：取出 Some 的值，None 時 panic——小心使用。
- expect("訊息")：和 unwrap 一樣，但 panic 時印出自訂訊息。
- unwrap_or(預設值)：None 時回傳預設值，不會 panic。
- flatten()：把 Option<Option<T>> 壓成 Option<T>。

- 搭配 `if let Some(x) = ...`（第 3 章學的）也很方便。

5.10 Result<T, E>

5.10.1 本集目標

學會使用 `Result<T, E>` 處理可能失敗的操作，理解它和 `Option` 的對稱關係。

5.10.2 概念說明

上兩集學了 `Option<T>`——「可能有值，可能沒有」。但有時候，「沒有值」不夠——你還需要知道**為什麼**沒有。

比如解析數字，失敗時你想知道是「格式錯誤」還是「數字太大」。這就是 `Result<T, E>` 的用途。

5.10.2.1 Result 的定義

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

- `Ok(T)` 表示成功，裡面包著成功的值。
- `Err(E)` 表示失敗，裡面包著錯誤資訊。

和 `Option` 一樣，`Result`、`Ok`、`Err` 也是 Rust 預設就引入到每個檔案裡的。

5.10.2.2 Option 和 Result 的對稱

Option	Result
`Some(T)`	`Ok(T)`
`None`	`Err(E)`

`Option` 只知道「有或沒有」，`Result` 還知道「為什麼沒有」。

5.10.2.3 回顧第 1 章的黑盒子

還記得第 1 章的 `.expect("讀取失敗")` 和 `.parse::<i32>().expect("不是數字")` 嗎？

`.parse()` 回傳的就是 `Result`。`expect` 的行為和 `Option` 的 `expect` 一模一樣——成功就取出 `Ok` 的值，失敗就 panic 並印出你的訊息。

現在我們終於能完整理解第 1 章的那段「黑盒子」程式碼了。

5.10.2.4 常用方法

和 `Option` 一樣，`Result` 也有：

- `.unwrap()`：成功取出值，失敗 panic。
- `.expect("訊息")`：和 `unwrap` 一樣，但自訂 panic 訊息。
- `.unwrap_or(預設值)`：失敗時用預設值。

5.10.3 範例程式碼

```
fn divide(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err(String::from("除數不能是零"))
    } else {
        Ok(a / b)
    }
}

fn main() {
    // 用 match 處理 Result
    let result = divide(10, 3);
    match result {
        Ok(value) => println!("10 / 3 = {}", value),
        Err(msg) => println!("錯誤：{}", msg),
    }

    // 除以零的情況
    let bad = divide(10, 0);
    match bad {
        Ok(value) => println!("結果：{}", value),
        Err(msg) => println!("錯誤：{}", msg),
    }

    // unwrap_or：失敗時用預設值
    let safe = divide(10, 0).unwrap_or(0);
    println!("安全的結果：{}", safe);

    // 回顧第 1 章：parse 回傳 Result
    let input = "42";
    let num: Result<i32, _> = input.parse();
    match num {
        Ok(n) => println!("解析成功：{}", n),
        Err(e) => println!("解析失敗：{:?}", e),
    }

    // expect：確定不會失敗時使用
    let num2 = "100".parse::<i32>().expect("這不應該失敗");
    println!("{}", num2);
}
```

5.10.4 重點整理

- `Result<T, E>` 表示「成功（`Ok`）或失敗（`Err`）」，比 `Option` 多了錯誤資訊。
- `Ok(T)` 對應成功，`Err(E)` 對應失敗。
- `Result`、`Ok`、`Err` 和 `Option` 一樣，是 Rust 預設就引入每個檔案的。
- `unwrap`、`expect`、`unwrap_or` 的用法和 `Option` 完全對稱。
- 第 1 章的 `.parse().expect(...)` 就是在用 `Result`——現在我們能理解了。

5.11 ? 運算子

5.11.1 本集目標

學會用 ? 運算子簡化錯誤傳播，避免一次又一次的 match。

5.11.2 概念說明

上一集學了 Result，我們用 match 來處理成功和失敗。但如果一個函數裡有好幾個可能失敗的操作呢？

```
fn do_stuff() -> Result<i32, String> {
    let a = match "42".parse::<i32>() {
        Ok(n) => n,
        Err(e) => return Err(format!("{:?}", e)),
    };
    let b = match "10".parse::<i32>() {
        Ok(n) => n,
        Err(e) => return Err(format!("{:?}", e)),
    };
    Ok(a + b)
}
```

每個 parse 都要 match 一次，太囉嗦了。? 運算子就是用來解決這個問題的。

5.11.2.1 ? 的本質

? 放在 Result 後面，做的事情就是：

- 如果是 Ok(v)，把 v 取出來，繼續往下跑。
- 如果是 Err(e)，回傳 Err，提前離開函數。

所以 ? 就是 match + early return 的簡寫。

5.11.2.2 注意：錯誤型別要一致

使用 Result 的時候，Err 裡的型別必須和函數回傳的 Err 型別一致或有某種關聯（之後會講具體是哪種關聯）。如果沒有關聯，就不能直接用 ?——你得先把錯誤轉成對的型別。

比如 .parse() 的錯誤型別是 std::num::ParseIntError，但你的函數回傳 Result<_, String>。這時候你可以用 match 自己轉換，然後再手動 return：

```
    let n = match input.parse::<i32>() {
        Ok(v) => Ok(v),
        Err(e) => return Err(format!("{:?}", e)),
    };
```

或者先包一層把錯誤轉好的輔助函數，在那個函數回傳之後就能直接用 ?——下面的範例程式碼就是這樣做的。

後面我們會教到更方便處理這種狀況的做法，不用每次都自己手動轉換錯誤型別。

5.11.2.3 ? 也能用在 Option

? 不只能用在 Result 上，也能用在 Option 上——如果是 None，就直接 return None。

5.11.2.4 `main` 也能回傳 `Result`

如果 `main` 函數回傳 `Result<(), String>`，你就可以在 `main` 裡使用 `?`。

5.11.3 範例程式碼

```rust
// 手動轉換錯誤型別的輔助函數
fn parse_i32(input: &str) -> Result<i32, String> {
    match input.parse::<i32>() {
        Ok(n) => Ok(n),
        Err(e) => Err(format!("解析 '{}' 失敗：{:?}", input, e)),
    }
}

// 使用 ? 簡化錯誤傳播
fn add_two_strings(a: &str, b: &str) -> Result<i32, String> {
    let x = parse_i32(a)?; // Ok 就取值，Err 就提前回傳
    let y = parse_i32(b)?;
    Ok(x + y)
}

// ? 用在 Option 上：第一個元素是正數嗎？
fn first_is_positive(numbers: &[i32]) -> Option<bool> {
    // 如果切片是空的，.first() 回傳 None，? 直接 return None
    let first = numbers.first()?;
    Some(*first > 0)
}

// main 也能回傳 Result，這樣就能用 ?
fn main() -> Result<(), String> {
    let result = add_two_strings("42", "10")?;
    println!("42 + 10 = {}", result);

    // 錯誤的情況
    let bad = add_two_strings("42", "abc");
    match bad {
        Ok(n) => println!("結果：{}", n),
        Err(e) => println!("錯誤：{}", e),
    }

    let nums = [3, 7, 2];
    match first_is_positive(&nums) {
        Some(true) => println!("第一個元素是正數"),
        Some(false) => println!("第一個元素不是正數"),
        None => println!("空的切片"),
    }

    let empty: &[i32] = &[];
    match first_is_positive(empty) {
        Some(b) => println!("結果：{}", b),
        None => println!("空的切片，沒有第一個元素"),
    }

    Ok(())
}
```

5.11.4 重點整理

- ? 是 match + early return 的簡寫。
- Result 上用 ?：Ok 取值，Err 提前回傳。
- Option 上用 ?：Some 取值，None 提前回傳。
- 使用 ? 時，錯誤型別必須和函數回傳型別一致或有關聯，沒有的話得自己先轉換。
- fn main() -> Result<(), String> 讓 main 也能使用 ?。

5.12 多個方法的 trait 與預設實作

5.12.1 本集目標

學會在 trait 中定義多個方法，以及用預設實作讓實作者只需覆寫需要的部分。

5.12.2 概念說明

第 4 章學 trait 的時候，我們的 trait 都只有一個方法。其實 trait 可以有很多個方法，而且有些方法可以提供**預設實作**——也就是先寫好一個「通用版本」，實作者不喜歡再覆寫。

5.12.2.1 多個方法

```
trait Describe {
    fn name(&self) -> String;
    fn description(&self) -> String;
}
```

實作的時候，所有方法都必須提供：

```
impl Describe for Cat {
    fn name(&self) -> String { ... }
    fn description(&self) -> String { ... }
}
```

5.12.2.2 預設實作

有些方法可以先寫好一個合理的預設版本：

```
trait Describe {
    fn name(&self) -> String;

    fn description(&self) -> String {
        let n = self.name();
        let mut result = String::from("我是 ");
        result.push_str(&n);
        result
    }
}
```

description 有預設實作，它呼叫了 .name() 來組合字串。實作 Describe 的時候，只需要提供 .name() 就好——description() 會自動使用預設版本。

當然，你也可以覆寫預設實作，提供自己的版本。

5.12.3 範例程式碼

```
trait Describe {
    // 必須實作的方法
    fn name(&self) -> String;

    // 預設實作：可以直接用，也可以覆寫
    fn description(&self) -> String {
        let n = self.name();
        let mut result = String::from("我是 ");
        result.push_str(&n);
        result
    }
}

struct Cat {
    nickname: String,
}

struct Dog {
    nickname: String,
}

// Cat 只實作 name，description 用預設的
impl Describe for Cat {
    fn name(&self) -> String {
        self.nickname.clone()
    }
}

// Dog 覆寫 description
impl Describe for Dog {
    fn name(&self) -> String {
        self.nickname.clone()
    }

    fn description(&self) -> String {
        let n = self.name();
        let mut result = String::from("汪汪！我叫 ");
        result.push_str(&n);
        result.push_str("，我是一隻狗！");
        result
    }
}

fn main() {
    let cat = Cat { nickname: String::from("小橘") };
    let dog = Dog { nickname: String::from("阿柴") };

    // Cat 用預設的 description
    println!("{}", cat.name());
    println!("{}", cat.description());

    // Dog 用自訂的 description
    println!("{}", dog.name());
```

```
    println!("{}", dog.description());
}
```

5.12.4 重點整理

- trait 可以定義多個方法。
- 方法可以提供**預設實作**——在方法後面直接寫 { ... } 而不是 ;。
- 預設實作可以呼叫同一個 trait 中的其他方法。
- 實作 trait 時，有預設實作的方法可以不寫（使用預設版本），也可以覆寫。

5.13 trait bound

5.13.1 本集目標

學會用 trait bound 限制泛型參數的能力，以及用條件式 impl 為符合條件的型別加方法。

5.13.2 概念說明

第一集學泛型函數的時候，我們寫了 fn first<T>(a: T, b: T) -> T。但如果你想在泛型函數裡 clone 一個值呢？

```
fn duplicate<T>(x: &T) -> (T, T) {
    (x.clone(), x.clone()) // 編譯錯誤！
}
```

編譯器會報錯：「不是所有 T 都有 .clone() 方法。」

這很合理——T 可以是任何型別，萬一有個型別沒有實作 Clone 呢？

5.13.2.1 trait bound：限制 T 的能力

解法是加上 **trait bound**，告訴 Rust「T 必須實作 Clone」：

```
fn duplicate<T: Clone>(x: &T) -> (T, T) {
    (x.clone(), x.clone())
}

fn main() {}
```

T: Clone 的意思是「T 必須實作 Clone trait」。這樣 Rust 就知道 x.clone() 一定可以呼叫。

5.13.2.2 到處都能加 trait bound

trait bound 不只能用在函數上。幾乎所有有泛型參數的地方都能加——struct、enum、impl 定義裡都可以：

```
struct Wrapper<T: Clone> {
    value: T,
}
```

5.13.2.3 條件式 impl

其中最實用的是在 impl 區塊上加 trait bound。這叫做**條件式 impl**——只有當型別參數符合某些條件時，才提供特定的方法。

```
impl<T: Clone> Pair<T> {
    fn to_tuple(&self) -> (T, T) {
        (self.first.clone(), self.second.clone())
    }
}
```

這段的意思是：只有當 T 實作了 Clone 的時候，Pair<T> 才有 to_tuple 方法。

5.13.2.4 實際效果

```
    let p1 = Pair::new(1, 2); // i32 有 Clone
    let t = p1.to_tuple();    // 可以呼叫 ✓

    let p2 = Pair::new(Pair::new(1, 2), Pair::new(3, 4)); // Pair 沒有 derive Clone
    p2.to_tuple(); // 編譯錯誤！Pair<i32> 沒有實作 Clone
```

Pair<Pair<i32>> 不能呼叫 .to_tuple()，因為 Pair<i32> 沒有實作 Clone(我們沒有幫它 derive Clone)。

5.13.3 範例程式碼

```
#[derive(Debug)]
struct Pair<T> {
    first: T,
    second: T,
}

// 所有 Pair<T> 都有 new
impl<T> Pair<T> {
    fn new(first: T, second: T) -> Pair<T> {
        Pair { first, second }
    }
}

// 只有 T: Clone 的 Pair<T> 才有 to_tuple
impl<T: Clone> Pair<T> {
    fn to_tuple(&self) -> (T, T) {
        (self.first.clone(), self.second.clone())
    }
}

// 泛型函數 + trait bound
fn duplicate<T: Clone>(x: &T) -> (T, T) {
    (x.clone(), x.clone())
}

fn main() {
    // i32 有 Clone，所以 Pair<i32> 有 to_tuple
    let p = Pair::new(10, 20);
    let t = p.to_tuple();
    println!("{:?}", t);

    // 泛型函數也可以用
    let pair = duplicate(&42);
```

```
    println!("{:?}", pair);

    let pair2 = duplicate(&String::from("hello"));
    println!("{:?}", pair2);

    // Pair<Pair<i32>> 不能呼叫 to_tuple
    // 因為 Pair<i32> 沒有 derive Clone
    let nested = Pair::new(Pair::new(1, 2), Pair::new(3, 4));
    println!("{:?}", nested);
    // nested.to_tuple(); // 編譯錯誤！Pair<i32> 沒有實作 Clone
}
```

5.13.4 重點整理

- trait bound `T: Clone` 限制 T 必須實作特定 trait。
- trait bound 可以加在函數、struct、enum、impl 等各種泛型參數上。
- 沒有 trait bound 的話，泛型函數或方法不能假設 T 有任何能力。
- 條件式 impl：`impl<T: Clone> Pair<T> { ... }` 只在 T 符合條件時提供方法。

5.14 use 基礎

5.14.1 本集目標

學會用 use 把長路徑縮短，並理解為什麼之前不用 use 就能用 Option、Vec 等型別。

5.14.2 概念說明

Rust 有非常多內建的函數、型別和 trait，為了組織它們，Rust 的標準庫用模組把東西分門別類。舉例來說，每個型別都有一個完整的**路徑**來說明它位於哪個模組裡，路徑用 :: 分隔，像是 std::string::String（String 位於 std 的 string 模組裡）、std::vec::Vec、std::fmt::Display。平常要用某個型別，就要寫出它的完整路徑。

但奇怪的是，我們前面一直在用 Vec、String、Option、Result 這些型別，從來沒寫過 std::vec::Vec 這種完整路徑也能用，為什麼？

因為 Rust 有一個叫 **prelude** 的機制——Rust 預設就把最常用的函數、型別和 trait 引入到每個檔案裡。Vec、String、Option、Result、Some、None、Ok、Err，還有 Clone、Copy 等常用 trait，都在 prelude 裡面，所以不用寫完整路徑。

但不是所有東西都在 prelude 裡。比如 std::fmt::Display 這個 trait，就不在 prelude 裡。如果你想用它，就要寫完整路徑——或者用 use 把它引入。

5.14.2.1 use 的語法

```
use std::fmt::Display;
```

這行的意思是：「把 std::fmt::Display 引入到當前的作用域，之後直接寫 Display 就好。」

use 會把既有名稱帶入目前的作用域，讓你能使用較短的路徑。沒有 use，你寫 std::fmt::Display；有了 use，你只需要寫 Display。

5.14.3 範例程式碼

```
use std::cmp::max;

fn main() {
    // 沒有 use 的話，要寫完整路徑：
    println!("較小的是：{}", std::cmp::min(3, 7));

    // 有了 use，直接寫 max 就好：
    println!("較大的是：{}", max(3, 7));
    println!("較大的是：{}", max(10, -2));
}
```

`std::cmp::max` 和 `std::cmp::min` 是標準庫提供的函數，回傳兩個值中比較大或比較小的那個。它們不在 prelude 裡，所以要嘛寫完整路徑，要嘛用 `use` 引入。

5.14.4 重點整理

- `use std::fmt::Display;` 把長路徑縮短，之後直接寫 `Display`。
- `use` 會把既有名稱帶入目前的作用域，讓你能使用較短的路徑。
- Rust 的編譯器預設引入 prelude 的常用型別和 `trait`（`Vec`、`String`、`Option`、`Clone` 等）。
- 不在 prelude 裡的東西（如 `Display`）需要寫完整路徑或用 `use` 引入。

5.15 Display trait

5.15.1 本集目標

學會為自訂型別實作 `Display trait`，理解 `Display` 和 `Debug` 的差別，以及 `Display` 和 `ToString` 的關係。

5.15.2 概念說明

第 2 章我們學了 `{:?}` 來印出 tuple、陣列和加了 `#[derive(Debug)]` 的 `struct`。但 `{:?}` 是給開發者看的「`Debug` 格式」。如果你想用 `{}` 來印出自訂型別，就需要實作 `Display trait`。

5.15.2.1 Display vs Debug

- **`Debug`**（`{:?}`）：給開發者看的格式，可以用 `#[derive(Debug)]` 自動產生。
- **`Display`**（`{}`）：給使用者看的格式，**必須手動實作**，不能 `derive`。

為什麼要分開？因為開發者需要看到所有欄位、型別資訊（`Debug` 格式），但使用者只需要看到好讀的文字。兩者的需求不同，所以不能用同一個 `trait` 解決。

5.15.2.2 實作 Display

```
use std::fmt::Display;
use std::fmt::Formatter;
use std::fmt::Result;

impl Display for Point {
    fn fmt(&self, f: &mut Formatter) -> Result {
        write!(f, "({}, {})", self.x, self.y)
    }
```

```
}
```

`fmt` 方法接收一個 `&mut Formatter`，你用 `write!` 巨集把想要的格式寫進去。`write!` 的用法和 `println!` 幾乎一樣，只是第一個參數是 `&mut Formatter`。

5.15.2.3 Display 和 ToString 的關係

Rust 有一個 `ToString` trait，它只有一個方法：

```
fn to_string(&self) -> String;
```

重點來了——**你不需要自己實作 `ToString`**。標準庫裡有這樣一段程式碼：

```
impl<T: Display> ToString for T {
    fn to_string(&self) -> String {
        // 內部用 Display 的 fmt 方法來產生字串
        // ...
    }
}
```

這段的意思是：「對於**所有**實作了 `Display` 的型別 `T`，自動幫它實作 `ToString`。」這叫做 **blanket implementation**（毯子式實作）——像一條毯子，蓋住所有符合條件的型別。

所以只要實作 `Display`，你的型別就自動有 `.to_string()` 方法，不用額外做任何事。

5.15.3 範例程式碼

```
use std::fmt::Display;
use std::fmt::Formatter;

#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

// 手動實作 Display
impl Display for Point {
    fn fmt(&self, f: &mut Formatter) -> std::fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}

#[derive(Debug)]
struct Color {
    r: u8,
    g: u8,
    b: u8,
}

impl Display for Color {
    fn fmt(&self, f: &mut Formatter) -> std::fmt::Result {
        write!(f, "R{}G{}B{}", self.r, self.g, self.b)
    }
}
```

```rust
fn main() {
    let p = Point { x: 3, y: 7 };

    // Debug 格式（給開發者看）
    println!("Debug: {:?}", p);

    // Display 格式（給使用者看）
    println!("Display: {}", p);

    // 因為有 Display，自動獲得 .to_string()
    let s = p.to_string();
    println!("to_string: {}", s);

    let c = Color { r: 255, g: 128, b: 0 };
    println!("Debug: {:?}", c);
    println!("Display: {}", c);
    println!("to_string: {}", c.to_string());
}
```

5.15.4 重點整理

- Display trait 讓你的型別可以用 {} 格式印出。
- Debug（{:?}）給開發者看，可以 derive；Display（{}）給使用者看，必須手動實作。
- 實作方式：impl Display for MyType，在 fmt 方法裡用 write! 寫格式。
- 實作 Display 會自動獲得 .to_string() 方法（blanket implementation）。

5.16 多個 trait bound 與 where

5.16.1 本集目標

學會用 + 組合多個 trait bound，以及用 where 子句讓複雜的 bound 更好讀。

5.16.2 概念說明

第 13 集我們學了 T: Clone，要求 T 必須實作 Clone。但如果你想同時要求 T 實作多個 trait 呢？

5.16.2.1 多個 trait bound

用 + 把多個 trait bound 串起來：

```rust
fn show_clone<T: Clone + std::fmt::Display>(x: &T) {
    let cloned = x.clone();
    println!("原始：{}", x);
    println!("clone 出來的值：{}", cloned);
}
```

T: Clone + Display 表示 T 必須同時實作 Clone 和 Display。

5.16.2.2 where 子句

當 trait bound 很長的時候，寫在 <> 裡面會很擠。Rust 提供 where 子句，放在函數簽名後面：

```rust
fn show_clone<T>(x: &T)
where
```

```
    T: Clone + std::fmt::Display,
{
    let cloned = x.clone();
    println!("原始：{}", x);
    println!("clone 出來的值：{}", cloned);
}
```

兩種寫法完全等價，只是 where 比較好讀。

5.16.2.3 where 比角括號更靈活

where 子句的冒號前面不只能放 T，還能放更複雜的東西。比如一個 tuple 型別：

```
fn clone_pair<T, U>(pair: &(T, U)) -> (T, U)
where
    (T, U): Clone,
{
    pair.clone()
}
```

(T, U): Clone 要求 tuple (T, U) 能被 clone。這種寫法只能出現在 where 子句裡，不能放在 <> 裡——這就是 where 更靈活的地方。

5.16.3 範例程式碼

```
use std::fmt::Display;

// 多個 trait bound：Clone + Display
// 印出原始值，然後回傳 clone 出來的值
fn clone_and_show<T: Clone + Display>(x: &T) -> T {
    println!("準備 clone：{}", x);
    x.clone()
}

// 用 where 子句：有時候比較好讀
fn show_pair<T, U>(a: &T, b: &U)
where
    T: Display,
    U: Display,
{
    println!("a = {}, b = {}", a, b);
}

fn main() {
    // 多個 trait bound
    let cloned = clone_and_show(&42);
    println!("拿到的 clone：{}", cloned);

    let cloned2 = clone_and_show(&String::from("hello"));
    println!("拿到的 clone：{}", cloned2);

    // where 子句
    show_pair(&10, &"world");
}
```

5.16.4 where 還能用在哪裡

where 不只能用在函數上。其他很多會用到泛型的地方也都能用 where，例如 impl 區塊：

```
impl<T> Pair<T>
where
    T: Clone + Display,
{
    // 方法定義
}
```

此外，where 也能出現在 struct、enum 和 trait 的定義上。目前知道就好了，之後需要用到的時候自然會想起來。

5.16.5 重點整理

- 用 + 組合多個 trait bound：T: Clone + Display。
- where 子句是另一種寫 trait bound 的方式，更好讀。
- where 比角括號更靈活，冒號前面可以放 tuple 等複雜型別（如 (T, U): Clone）。
- where 不只能用在函數上，impl、struct、enum、trait 等能用泛型的地方都能用。

5.17 impl Trait 語法

5.17.1 本集目標

學會用 impl Trait 作為 trait bound 的簡寫，理解它在參數和回傳值中的不同含義。

5.17.2 概念說明

我們前面學了 trait bound：fn foo<T: Display>(x: &T)。Rust 還提供了一種更簡潔的寫法：impl Trait。

5.17.2.1 參數位置的 impl Trait

```
fn show(x: &impl Display) {
    println!("{}", x);
}
```

這和 fn show<T: Display>(x: &T) 幾乎完全等價——都是說「x 的型別必須實作 Display」，寫法也更簡潔。

5.17.2.2 每個 impl Trait 是獨立的型別

重要觀念：參數中的每個 impl Trait 代表一個**獨立的**型別。

```
fn show_two(a: &impl Display, b: &impl Display) {
    println!("{} {}", a, b);
}
```

a 和 b 可以是**不同的型別**——只要它們都實作了 Display。比如 a 可以是 i32，b 可以是 String。

如果你要求 a 和 b **必須是同一個型別**，就要用具名的型別參數：

```
fn show_same<T: Display>(a: &T, b: &T) {
    println!("{} {}", a, b);
```

```
}
```

5.17.2.3 回傳位置的 impl Trait

impl Trait 也可以用在回傳值：

```
fn greeting() -> impl Display {
    String::from("你好")
}
```

這表示「我會回傳一個實作了 Display 的型別，但不告訴你具體是什麼型別」。呼叫者只知道回傳值可以用 Display 的方法（像 println!("{}", greeting())），不知道具體是 String 還是其他什麼。

5.17.3 範例程式碼

```
use std::fmt::Display;

// 參數位置的 impl Trait
fn show(x: &impl Display) {
    println!("顯示：{}", x);
}

// 每個 impl Trait 是獨立型別，a 和 b 可以不同型別
fn show_pair(a: &impl Display, b: &impl Display) {
    println!("{} 和 {}", a, b);
}

// 要求同一型別，用泛型
fn show_same<T: Display>(a: &T, b: &T) {
    println!("{} 和 {}", a, b);
}

// 回傳位置的 impl Trait
fn make_greeting(name: &str) -> impl Display {
    let mut s = String::from("你好，");
    s.push_str(name);
    s.push_str("！");
    s
}

fn main() {
    // 參數位置
    show(&42);
    show(&String::from("hello"));

    // 兩個參數可以不同型別
    show_pair(&42, &"hello");

    // 要求同型別
    show_same(&10, &20);
    // show_same(&10, &"hello"); // 編譯錯誤！i32 和 &str 不同型別

    // 回傳 impl Trait
```

```
    let greeting = make_greeting("世界");
    println!("{}", greeting);

    // greeting 的型別是 `impl Display`，不是 `String`
    // 所以你不能把它當 String 用：
    // greeting.push_str("!!!"); // 編譯錯誤！impl Display 沒有 push_str 方法
    // 我們知道裡面其實是 String，但編譯器只看到 Display
}
```

5.17.4 重點整理

- `fn foo(x: &impl Display)` 是 `fn foo<T: Display>(x: &T)` 的簡寫。
- 每個 `impl Trait` 參數代表獨立的型別——兩個 `impl Display` 可以是不同型別。
- 要求同型別，用具名的型別參數 `<T: Display>`。
- 回傳位置的 `-> impl Trait` 隱藏具體型別，呼叫者只知道它實作了什麼 trait。

5.18 多參數 trait

5.18.1 本集目標

學會定義帶其他型別參數的 trait，讓同一個型別可以針對不同目標型別實作同一個 trait。

5.18.2 概念說明

到目前為止，我們的 trait 都比較簡單——Describe、Clone、Display，沒有其他型別參數。但有時候你想定義的行為和**另一個型別**有關。

比如「轉換」這件事：i32 可以轉成 f64，也可以轉成 String。同一個型別，轉換的目標不同，邏輯也不同。

5.18.2.1 帶其他型別參數的 trait

```
trait Convert<T> {
    fn convert(self) -> T;
}
```

`Convert<T>` 的意思是：「可以轉換成 T 型別」。同一個型別可以實作 `Convert<f64>`、`Convert<String>` 等不同版本。

5.18.2.2 實作多參數 trait

```
impl Convert<(i32,)> for i32 {
    fn convert(self) -> (i32,) {
        (self,)
    }
}
```

這裡 i32 實作了 `Convert<(i32,)>`——把自己轉成單元素 tuple。

同一個型別可以實作多次，只要型別參數不同：

```
impl Convert<String> for i32 {
    fn convert(self) -> String {
```

```
        // 用 ToString trait（i32 已經有了）
        self.to_string()
    }
}
```

5.18.2.3 和沒有其他參數的 trait 的差別

- Clone（沒有其他參數）：一個型別只能實作一次 Clone。
- Convert<T>（有其他參數）：一個型別可以實作 Convert<String>、Convert<(i32,)> 等多個版本。

5.18.3 範例程式碼

```
// 定義一個帶型別參數的 trait
trait Convert<T> {
    fn convert(self) -> T;
}

// i32 轉成單元素 tuple
impl Convert<(i32,)> for i32 {
    fn convert(self) -> (i32,) {
        (self,)
    }
}

// i32 轉成 String
impl Convert<String> for i32 {
    fn convert(self) -> String {
        self.to_string()
    }
}

// bool 轉成 i32
impl Convert<i32> for bool {
    fn convert(self) -> i32 {
        if self {
            1
        } else {
            0
        }
    }
}

fn main() {
    // i32 -> (i32,)
    let x: i32 = 42;
    let tuple: (i32,) = x.convert();
    println!("{:?}", tuple);

    // i32 -> String
    let y: i32 = 100;
    let s: String = y.convert();
    println!("{}", s);
```

```
    // bool -> i32
    let b = true;
    let n: i32 = b.convert();
    println!("{}", n);
}
```

5.18.4 重點整理

- trait 可以帶其他型別參數：trait Convert<T> { ... }。
- 同一個型別可以對不同的 T 實作同一個 trait（例如 Convert<String> 和 Convert<(i32,)>）。
- 這和沒有其他參數的 trait 不同——一個型別只能實作一次那些 trait。
- 多參數 trait 讓「和另一個型別相關的行為」可以統一定義。

5.19 From<T> / Into<T>

5.19.1 本集目標

學會使用標準庫的 From 和 Into trait 做型別轉換，理解「實作 From 就自動獲得 Into」的機制。

5.19.2 概念說明

上一集我們自己定義了 Convert<T> trait。其實 Rust 標準庫已經有一套更完整的轉換機制：From 和 Into。

5.19.2.1 From

From<T> 的定義（簡化版）：

```
trait From<T> {
    fn from(value: T) -> Self;
}
```

它的意思是：「我可以從 T 轉換而來。」

你一定見過這個：

```
    let s = String::from("hello");
```

這就是 String 實作了 From<&str>——從 &str 轉換成 String。

5.19.2.2 Into

Into<T> 是 From 的反方向：

```
trait Into<T> {
    fn into(self) -> T;
}
```

重點：**你只需要實作 From，就自動獲得 Into。**不需要自己實作 Into。

這又是一個 blanket implementation——Rust 有一個規則是「如果 Y: From<X>，那 X 自動實作 Into<Y>」。

5.19.2.3 TryFrom / TryInto

有些轉換可能失敗——比如把一個很大的 i64 轉成 i32 可能會溢位。這時候用 TryFrom 和 TryInto，它們回傳 Result 而不是直接回傳值。

和 From / Into 一樣，實作 TryFrom 就自動獲得 TryInto。

5.19.3 範例程式碼

```
use std::fmt::Display;
use std::fmt::Formatter;

struct Celsius {
    value: f64,
}

struct Fahrenheit {
    value: f64,
}

impl Display for Celsius {
    fn fmt(&self, f: &mut Formatter) -> std::fmt::Result {
        write!(f, "{}°C", self.value)
    }
}

impl Display for Fahrenheit {
    fn fmt(&self, f: &mut Formatter) -> std::fmt::Result {
        write!(f, "{}°F", self.value)
    }
}

// 實作 From：從 Celsius 轉成 Fahrenheit
impl From<Celsius> for Fahrenheit {
    fn from(c: Celsius) -> Fahrenheit {
        Fahrenheit {
            value: c.value * 1.8 + 32.0,
        }
    }
}

fn main() {
    // String::from——我們一直在用的
    let s = String::from("hello");
    println!("{}", s);

    // 自訂的 From
    let boiling = Celsius { value: 100.0 };
    println!("攝氏：{}", boiling);
    let f = Fahrenheit::from(Celsius { value: 100.0 });
    println!("華氏：{}", f);

    // 自動獲得 Into（不需要另外實作）
    let body_temp = Celsius { value: 37.0 };
    let f2: Fahrenheit = body_temp.into();
```

```
    println!("體溫：{}", f2);

    // TryFrom 的例子：i32 轉 u8 可能失敗
    let big: i32 = 300;
    let result = u8::try_from(big);
    match result {
        Ok(n) => println!("轉換成功：{}", n),
        Err(e) => println!("轉換失敗：{:?}", e),
    }

    let small: i32 = 42;
    let ok = u8::try_from(small);
    match ok {
        Ok(n) => println!("轉換成功：{}", n),
        Err(e) => println!("轉換失敗：{:?}", e),
    }
}
```

5.19.4 重點整理

- From<T> 定義「從 T 轉換而來」：String::from("hello") 就是這個。
- 實作 From 就自動獲得 Into——不需要另外實作。
- .into() 是 .from() 的反方向：let f: Fahrenheit = celsius.into();。
- TryFrom / TryInto 用於可能失敗的轉換，回傳 Result。
- 實作 TryFrom 也會自動獲得 TryInto。

5.20 Drop

5.20.1 本集目標

學會用 drop(value) 提前丟棄值，理解 Rust 會自動丟棄值所擁有的內容，並用 Drop trait 在內部值被丟棄前執行額外動作。

5.20.2 概念說明

5.20.2.1 用 drop(value) 提前丟棄值

值通常會在離開作用域時被 Rust 自動丟棄。如果不想等到作用域結束，可以呼叫 drop(value) 提前丟棄它：

```
fn main() {
    let message = String::from("hello");
    println!("{}", message);

    drop(message);
    println!("message 已經被丟棄");

    // println!("{}", message); // 編譯錯誤！message 的值已經被 move
}
```

drop 是 prelude 提供的普通函數，不需要額外 use。它會取得傳入值的所有權，因此 drop(message) 之後不能再使用原本的值。

更強調地說，被丟棄的是 message 綁定的**值**，不是變數名稱本身。變數原本的作用域並沒有縮短，只是它的值已經被 move 給 drop，接著被丟棄。

5.20.2.2 內部值也會被自動丟棄

丟棄一個值時，Rust 也會繼續丟棄這個值所擁有的欄位或元素：

```
struct Message {
    title: String,
    body: String,
}

fn main() {
    let message = Message {
        title: String::from("問候"),
        body: String::from("你好！"),
    };

    drop(message);
}
```

這裡只需要丟棄 message，Rust 就會自動丟棄裡面的 title 和 body。同樣的機制也適用於 tuple、enum、陣列和 Vec 等型別所擁有的值，不需要自己逐一清理。

5.20.2.3 用 Drop 在丟棄前執行額外動作

有時候，在 Rust 自動丟棄內部值之前，我們想先做一些事情，例如關閉連線、歸還資源或印出紀錄。這時可以為型別實作 Drop trait：

```
impl Drop for Resource {
    fn drop(&mut self) {
        println!("釋放資源：{}", self.name);
    }
}
```

當 Resource 被丟棄時，Rust 會先執行 Drop 的 .drop() 方法，之後再自動丟棄它的欄位。這個方法是丟棄過程中的額外動作，不會取代 Rust 對內部值的自動處理。

雖然 Drop 的 .drop() 方法和 drop(value) 函數都叫作 drop，但兩者的用法不同：

- drop(value) 是可以主動呼叫的普通函數，用來提前丟棄值。
- Drop 的 .drop() 方法由 Rust 在丟棄值時自動執行，不能寫成 value.drop() 主動呼叫。

為什麼不能手動呼叫 value.drop()？因為這個方法只取得 &mut self，不會取得值的所有權。如果允許手動呼叫，值在呼叫後仍然存在；等它之後真的被丟棄時，Rust 又會執行一次相同的方法，可能重複釋放同一份資源。因此 Rust 直接禁止這種寫法。

drop(value) 不一樣：它會取得值的所有權，呼叫後原本的值不能再使用。這能讓 Rust 提前完成整個丟棄過程，而不會留下稍後還要再次丟棄的值。

5.20.3 範例程式碼

```
struct Resource {
    name: String,
}
```

```
impl Drop for Resource {
    fn drop(&mut self) {
        println!("釋放資源：{}", self.name);
    }
}

struct Worker {
    name: String,
    resource: Resource,
}

impl Drop for Worker {
    fn drop(&mut self) {
        println!(
            "停止工作者：{}（接著會釋放 {}）",
            self.name,
            self.resource.name,
        );
    }
}

fn main() {
    let worker = Worker {
        name: String::from("下載器"),
        resource: Resource {
            name: String::from("網路連線"),
        },
    };

    println!("工作者執行中");
    drop(worker);
    println!("工作者已被提前丟棄");

    {
        let temporary = Resource {
            name: String::from("暫存檔"),
        };
        println!("暫時資源使用中：{}", temporary.name);
    } // temporary 在這裡被自動丟棄
}
```

5.20.4　實作 Drop 的型別不能部分 move

如果一個型別實作了 Drop，就不能從它的欄位 move 出值：

```
struct Resource {
    name: String,
    id: i32,
}

impl Drop for Resource {
    fn drop(&mut self) {
        println!("釋放 {}（編號 {}）", self.name, self.id);
    }
}
```

```
fn main() {
    let resource = Resource {
        name: String::from("資料庫連線"),
        id: 1,
    };

    let name = resource.name; // 編譯錯誤！不能部分 move
}
```

Drop 的 .drop() 方法會取得完整的 &mut self，因此可能存取任何欄位。如果允許先把 name move 出去，之後執行這個方法時，resource 就不再完整，所以 Rust 禁止這種操作。

如果某個欄位本身又是一個 struct，從更內層的欄位 move 出值也一樣會讓外層值不完整，因此同樣不允許。

限制的是從欄位 **move** 出值；以下操作仍然可以：

- move 整個 resource。
- 借用欄位，例如 &resource.name。
- 複製有實作 Copy 的欄位，例如 resource.id。

5.20.5 重點整理

- Rust 會在值離開作用域時自動丟棄它。
- drop(value) 取得值的所有權，讓你在作用域結束前提前丟棄它。
- 外層值被丟棄時，它所擁有的內部值也會被自動丟棄。
- Drop 的 .drop() 方法讓你在內部值被丟棄前執行額外動作；它由 Rust 自動執行，不能直接呼叫。
- 實作 Drop 的型別不能部分 move，但仍然可以 move 整個值、借用欄位或複製 Copy 欄位。

5.21 Box<T>

5.21.1 本集目標

學會用 Box<T> 把資料放在 heap 上，理解它在遞迴型別中的必要性。

5.21.2 概念說明

還記得第 4 章的保險箱比喻嗎？鑰匙圈上掛著鑰匙，鑰匙可以打開保險箱，保險箱裡放著真正的東西。

Box<T> 就是那個保險箱的鑰匙——資料住在 heap 上，而 stack 上的 Box 值讓 Rust 能找到它。

5.21.2.1 為什麼需要 Box？

大部分時候，Rust 把資料直接放在 stack 上就好了。但有兩種情況需要 Box：

1. 資料太大

如果一個 struct 有很多欄位、佔很多空間，放在 stack 上可能不太好（stack 空間有限）。用 Box 把它移到 heap 上，stack 上只留下那把「鑰匙」。這種用來找到存放在其他位置之資料的資訊稱為指標。

2. 遞迴型別

這是更重要的原因。假設你想定義一個連結串列（linked list）：

```
enum List {
    Node(i32, List), // 編譯錯誤！
    Empty,
}
```

Rust 需要在編譯時期知道每個型別的大小。但這裡有個問題：要知道 `List` 的大小，你需要知道 `Node` 有多大。`Node` 包含一個 `i32` 和一個 `List`——所以你需要知道 `List` 有多大。但 `List` 裡面又有 `List` ……

展開來看：`List` 的大小 = `i32` + `List` 的大小 = `i32` + `i32` + `List` 的大小 = ……永遠算不完。編譯器在這裡直接報錯：「recursive type has infinite size（遞迴型別大小無限大）」。

解法就是用 `Box`：

```
enum List {
    Node(i32, Box<List>),
    Empty,
}
```

`Box<List>` 的大小是固定的（就是一個指標的大小），問題就解決了。

5.21.2.2　`Box` 的使用

```
fn main() {
    let x = Box::new(42);
    println!("{}", x); // 可以直接用，Rust 會自動拿裡面的值
}
```

`Box::new(value)` 把值搬到 heap 上。`Box` 擁有裡面的值，離開作用域時會自動釋放（因為 `Box` 實作了 `Drop`）。

5.21.3　範例程式碼

```
// 用 Box 的遞迴型別：連結串列
enum List {
    Node(i32, Box<List>),
    Empty,
}

// 印出串列
fn print_list(list: &List) {
    match list {
        List::Node(value, next) => {
            print!("{} -> ", value);
            print_list(next);
        }
        List::Empty => {
            println!("end");
        }
    }
}
```

```
fn main() {
    // 基本的 Box 使用
    let x = Box::new(42);
    println!("Box 裡的值：{}", x);

    // 一步一步建立連結串列：3 -> 2 -> 1 -> end
    // 從最後面開始建立
    let list = List::Empty;                    // end
    let list = List::Node(1, Box::new(list)); // 1 -> end
    let list = List::Node(2, Box::new(list)); // 2 -> 1 -> end
    let list = List::Node(3, Box::new(list)); // 3 -> 2 -> 1 -> end

    print_list(&list);

    // Box 是單獨的一把鑰匙——鑰匙不是 Copy，所以 let b = a 是 move
    let a = Box::new(String::from("hello"));
    let b = a; // 鑰匙從 a 交給 b，a 就空了
    // println!("{}", a); // 編譯錯誤！a 已經被 move 了
    println!("{}", b);
}
```

5.21.4 重點整理

- Box<T> 把資料放在 heap 上，stack 上只留一個指標（保險箱比喻的「鑰匙」）。
- 最重要的用途：**遞迴型別**（如連結串列）需要 Box 來打破無限大小的問題。
- Box::new(value) 建立 Box，離開作用域時自動釋放。
- Box 是單獨的一把鑰匙；move 它的規則和 move 其他非 Copy 的值一樣。

5.22 Rc<T>

5.22.1 本集目標

學會讓好幾個 Rc<T> 值共享同一份 heap 資料，並理解參考計數如何讓這份資料活著。

5.22.2 概念說明

上一集是 Box<T>：一個保險箱，一把鑰匙。Box 值就是那把鑰匙，heap 資料則是保險箱裡的東西。

但有時候，程式的好幾個地方需要使用**同一份** heap 資料，而不是另外建立一份獨立的 heap 值。

5.22.2.1 問題：move 一個 Box

Box<T> 只有一把鑰匙。把它指定給另一個變數，就是把鑰匙 move 過去：

```
    let a = Box::new(String::from("hello"));
    let b = a; // move！a 不能再用了
```

move 之後，鑰匙在 b 手上。heap 資料沒有被複製，而 a 再也不能用它。

如果你希望 a 和 b 都能繼續使用同一段文字，怎麼辦？

你可能會想：「呼叫 .clone() 不就好了？」

```
    let a = Box::new(String::from("hello"));
    let b = a.clone(); // 建立一個新的 Box，裝著另一個 String
```

這確實能讓 a 和 b 都能用。但在這個 Box<String> 的例子裡，.clone() 建立的是另一個 Box，裝著一個內容相同的新 String。如果內部的值很大，這會是一筆不小的開銷。而且 a 和 b 現在用的是**兩個各自獨立的 String 值**，不是同一個。

如果你需要的是「好幾個值使用**同一份** heap 資料」，Box 的 .clone() 就不是正確的工具。

5.22.2.2 Rc 的 .clone()：真的只多打一把鑰匙

還記得第 4 章對 .clone() 的定義嗎？——**「弄一個跟原本一樣好用的新鑰匙圈，並確保這樣做不會出問題」**。對 String 或 Vec<i32> 這類單純的具所有權型別來說，「確保」的方法是打一把新鑰匙、配一個內容相同的新保險箱。

Rc 就是第 4 章預告過的例外：它的 .clone() 真的只多打一把鑰匙。你得到一個新的 Rc 值，開的是同一個保險箱。資料本身不會被複製，這就是為什麼就算資料很大，對 Rc 呼叫 .clone() 依然便宜。

但如果好幾把鑰匙都能開同一個保險箱，Rust 就需要一個辦法知道保險箱什麼時候可以撤掉。這就輪到計數器登場了。

5.22.2.3 Rc：參考計數

Rc<T> 是 **reference counting**（參考計數）的縮寫。

把 Rc<T> 想成一個允許多把鑰匙的保險箱。Rc 替這個保險箱維護一個計數器。

計數器追蹤的是目前還有幾個 Rc<T> 值能開這個保險箱：

- 建立第一個 Rc：計數 = 1。
- .clone()：計數 +1——你得到的是同一個保險箱的另一把鑰匙，不是另一份獨立的資料。
- 某個 Rc 值離開作用域：計數 -1。
- 計數歸零時，heap 資料才會被釋放。

計數器管的是「保險箱什麼時候可以撤掉」：只要還有人手上有鑰匙，保險箱就不會被撤走，所以不會有人拿著鑰匙、卻發現保險箱早就不在了。

這也表示 Rc 的 Clone 不能用 derive。Rc 的 .clone() 要做兩件事：多打一把鑰匙，還要把保險箱上的計數器加一。derive 只會對每個欄位各自呼叫 .clone()，第一件做得到，第二件做不到——鑰匙多了一把，計數器卻沒有跟著加。所以標準庫替 Rc 手寫了 Clone。

5.22.2.4 要分清楚的兩個層次

這個區分很重要：

```
    let a = Rc::new(String::from("hello"));
    let b = a.clone();
```

這裡有兩個層次：

- a 擁有一個 Rc<String> 值。
- b 擁有另一個 Rc<String> 值。
- 兩個 Rc<String> 值開的是同一份 heap 資料。
- 計數器追蹤的是目前還存在幾個 Rc<String> 值。

所以 Rc 並不代表一般的 Rust 值就不守所有權規則了。每個 Rc 值仍然是一個普通的值：它會 move、

會被 drop、move 之後舊變數就不能再用。特別的是那些 Rc 值背後的 heap 資料：它會一直活到最後一把鑰匙消失為止。

5.22.2.5 Rc 是唯讀的

Rc<T> 本身提供的是共享的讀取。好幾個 Rc 值可以讀同一份 heap 資料，但不能隨意修改它。

如果需要「共享而且可以修改」的資料，之後會學 RefCell<T>。

5.22.2.6 等等，第 4 章不是提醒過不能隨便複製鑰匙嗎？

第 4 章的擔心很具體：兩個人各拿一把鑰匙開同一個保險箱，A 正在整理保險箱裡的東西，B 同時把東西拿走了。那 Rc 憑什麼可以配好幾把鑰匙？

因為 Rc 把那個情境裡的兩種危險分別堵住了：

1. **沒有人能動保險箱裡的東西**：Rc<T> 只給共享的讀取。第 4 章怕的是有人整理、有人拿走；大家都只能看的話，這件事就不會發生。
2. **保險箱不會在還有人拿著鑰匙的時候被撤掉**：這是計數器的工作。要等最後一把鑰匙消失，資料才會被釋放。

第二點不是免費的：計數器在每次 clone 和 drop 時都要更新，這是 Box<T> 沒有的額外開銷。Box<T> 是單純的「一把鑰匙」heap 配置，沒有計數器；Rc<T> 加上了執行時期的記帳，讓好幾個 Rc 值能開同一個保險箱。這種共享很有用，但不是免費的。

另外，講到多執行緒的時候，我們會看到 Rc 在那裡還有別的限制。這裡先知道就好。

5.22.3 範例程式碼

```
use std::rc::Rc;

fn main() {
    // 建立 Rc，計數 = 1
    let a = Rc::new(String::from("共享的資料"));
    println!("建立 a，計數 = {}", Rc::strong_count(&a));

    // .clone() 替同一份 heap 資料多建立一個 Rc
    let b = a.clone();
    println!("clone 給 b，計數 = {}", Rc::strong_count(&a));

    let c = a.clone();
    println!("clone 給 c，計數 = {}", Rc::strong_count(&a));

    // a、b、c 都能讀同一份資料
    println!("a = {}", a);
    println!("b = {}", b);
    println!("c = {}", c);

    {
        let _d = a.clone();
        println!("在作用域裡，計數 = {}", Rc::strong_count(&a));
    } // _d 被 drop，計數 -1
    println!("離開作用域後，計數 = {}", Rc::strong_count(&a));

    // 實際用途：好幾個值共享同一個名字
```

```
    let shared_name = Rc::new(String::from("Rust"));

    let greeting1 = shared_name.clone();
    let greeting2 = shared_name.clone();

    println!("1: {}", greeting1);
    println!("2: {}", greeting2);
}
```

5.22.4 重點整理

- Rc<T> 讓好幾個 Rc 值共享同一份 heap 資料。
- Rc::new(value) 建立時計數為 1。
- .clone() 替同一份 heap 資料多建立一個 Rc 值：計數 +1，不會多出一份獨立的內部值。
- drop 掉一個 Rc 計數 -1；計數歸零時 heap 資料才被釋放。
- Rc<T> 本身提供共享的讀取，不是不受限制的修改。
- Rc 能安全共享靠兩件事：只給讀取，加上計數器保證資料活到最後一個 Rc 消失。
- Rc<T> 有使用上的限制，不是所有共享問題的萬用解。
- 每個 Rc 值仍然遵守一般的所有權規則：它像其他非 Copy 值一樣 move、一樣 drop。
- 用 Rc::strong_count(&x) 查看目前的參考計數。

5.23 Deref

5.23.1 本集目標

理解 Deref trait、DerefMut trait 和 Rust 的 deref coercion，以及智慧指標為什麼常常能當成它們裡面的值來用。

5.23.2 概念說明

5.23.2.1 對 Rc 使用 *

到目前為止，我們的 * 幾乎都用在一般的參考（&T）上。但 * 也能用在某些智慧指標上：

```
use std::rc::Rc;

fn main() {
    let value = Rc::new(42);
    let number: i32 = *value;

    println!("{}", number); // 42
}
```

Rc<i32> 不是 i32，但 Rust 能用這把鑰匙觸及裡面的 i32。這裡 i32 是 Copy，所以把 *value 指定給 number 會產生出另一個 i32 值。

如果內部的值不是 Copy——例如 String——就沒辦法這樣把它搬出來，正如你從第 4 章的借用行為會預期的：

```
use std::rc::Rc;

fn main() {
```

```
    let text = Rc::new(String::from("hello"));
    let moved: String = *text; // 編譯錯誤！
}
```

可能還有其他 Rc 值開著同一份 heap 資料。把裡面的 String 搬走，那些 Rc 值的鑰匙就只能開到一個空保險箱了，所以 Rust 禁止這麼做。

5.23.2.2 Deref trait

這背後的機制是 Deref trait。目前我們還不需要它的精確定義；重要的概念更簡單：

Deref 告訴 Rust 怎麼**穿過**一個值去借用。例如，穿過 Rc<i32> 可以借用到裡面的 i32，產生一個 &i32。

那個參考正是重點。Deref 給 Rust 的是一個指向內部值的**參考**；它本身並不交出內部值的所有權。

Rc<T> 和 Box<T> 都實作了 Deref。像這樣本職就是作為某個內部值的鑰匙——管理它，並讓 Rust 能透過 Deref 觸及它——的型別，我們常稱為**智慧指標（smart pointer）**。其他一些標準庫型別（像 String 和 Vec<T>）雖然也實作了 Deref，但當鑰匙並不是它們的本職；本集聚焦在智慧指標上。

5.23.2.3 *v 背後發生了什麼

對一個實作了 Deref 的型別使用 * 時，好用的心智模型是：

```
*v
// 大致上：穿過 v 借用，然後沿著那個參考走過去
```

以剛才的 Rc<i32> 為例：

```
let value = Rc::new(42);

*value
// 大致上：
// 穿過 value 借用，得到 &i32
// 然後沿著那個 &i32 走過去
```

因為 i32 是 Copy，這能產生出另一個 i32 值。如果內部的值不是 Copy（像 String），一般的 Deref 沒辦法讓你把它搬出來。

5.23.2.4 deref coercion

deref coercion 是 Rust 在需要時自動透過 Deref 轉換參考型別的機制。

例如，這個函數要的是 &i32：

```
use std::rc::Rc;

fn show(n: &i32) {
    println!("{}", n);
}

fn main() {
    let value = Rc::new(42);

    show(&value); // &Rc<i32> 自動變成 &i32
}
```

show 需要 &i32，但 &value 是 &Rc<i32>。因為 Rc<i32> 實作的 Deref 讓 Rust 能借用到內部的

i32，Rust 就能轉換：

```
&Rc<i32> -> &i32
```

這個轉換發生在參考層面，沒有任何所有權被 move 走。

deref coercion 也可以連鎖：

```
use std::rc::Rc;

fn show(n: &i32) {
    println!("{}", n);
}

fn main() {
    let value = Rc::new(Box::new(42));

    show(&value); // &Rc<Box<i32>> -> &Box<i32> -> &i32
}
```

Rust 先穿過 Rc，再穿過 Box，直到參考型別符合函數要的為止。

5.23.2.5 method call 的自動解參考

method call 有它自己的自動解參考行為。當你用 . 呼叫方法時，Rust 會先試外層的型別；在那裡找不到對應的方法，就往內走一層再試一次。

例如：

```
use std::rc::Rc;

fn main() {
    let numbers = Rc::new(vec![10, 20, 30]);

    println!("{}", numbers.len()); // 呼叫 Vec<i32> 的 .len()
}
```

Rc<Vec<i32>> 自己沒有定義 .len()，但 Vec<i32> 有。Rust 能用 Rc 這把鑰匙，借用到內部的 Vec<i32>，再對它呼叫 .len()。

有多層包裝時，Rust 能一次往內走一層：

```
let numbers = Rc::new(Box::new(vec![10, 20, 30]));

numbers.len()
// Rust 能穿過 Rc、再穿過 Box，找到 Vec 的 .len()
```

這就是為什麼智慧指標常常讓人感覺就像它們裡面的值：method call 能自動穿過智慧指標去借用。

5.23.2.6 DerefMut

DerefMut 是 Deref 的可變版本。它告訴 Rust 怎麼**可變地**穿過一個值去借用：從一個可變的智慧指標，借到指向內部值的可變參考。

Rc<T> 沒有實作 DerefMut，因為可能還有其他 Rc 值開著同一份 heap 資料。一般的 Rc<T> 提供的是共享的讀取，不是不受限制的可變存取。Rc<T> 無法證明自己是這份 heap 資料唯一的鑰匙；如果允許 DerefMut，同一份 heap 資料就可能同時出現好幾個 &mut T。

Box<T> 就不同了：一把鑰匙、沒有計數器，所以可變的 Box<T> 能提供對內部值的可變存取：

```rust
fn main() {
    let mut text = Box::new(String::from("hello"));

    text.push_str(" world");
    println!("{}", text);

    *text = String::from("replaced");
    println!("{}", text);
}
```

`.push_str()` 的呼叫可變地借用了內部的 `String`；透過 `*text` 的賦值則把內部的 `String` 整個換掉。兩者都是 `DerefMut` 的正常行為：Rust 穿過 `Box` 拿到一個 `&mut String`。

5.23.2.7 方法同名時的優先順序

Rust 從外往內找方法。外層智慧指標自身的方法，優先於內層型別的方法。

一個常見的例子是 `.clone()`。`Rc` 自己有 `.clone()` 方法：替同一份 heap 資料多建立一個 `Rc`，並增加參考計數。內部的值可能也有自己的 `.clone()` 方法。

直接呼叫 `.clone()` 得到的是另一個 `Rc`：

```rust
use std::rc::Rc;

fn main() {
    let a = Rc::new(String::from("hello"));
    let b = a.clone(); // Rc 的 .clone()：增加計數，不會建立新的 String
}
```

如果要的是內層 `String` 自己的 `.clone()`，就明確寫出來：

```rust
    let c = (*a).clone(); // String 的 .clone()：建立一個新的 String
```

5.23.2.8 Box<T> 的一項特權

上面把 `Deref` 一律當成「穿過智慧指標去借用」，這是正確的通用模型。

`Box<T>` 多了一項額外能力：當你**擁有**這個 `Box<T>` 時，Rust 允許你用 `*box_value` 把裡面的 `T` 搬出來：

```rust
fn main() {
    let boxed = Box::new(String::from("owned"));
    let text: String = *boxed; // OK：把 String 從 Box 裡搬出來

    println!("{}", text);
}
```

這是 `Box<T>` 專屬的特殊待遇，**不是**一般 `Deref` 型別做得到的事：

```rust
use std::rc::Rc;

fn main() {
    let shared = Rc::new(String::from("shared"));
    let text: String = *shared; // 編譯錯誤！
}
```

所以通用規則保持簡單：`Deref` 讓 Rust 穿過一個值去借用；用 `*` 把非 `Copy` 的值搬出來，是 `Box<T>`

專屬的特殊能力。

5.23.3 範例程式碼

```
use std::rc::Rc;

fn show(n: &i32) {
    println!("值：{}", n);
}

fn main() {
    // Rc<i32>：* 觸及那個 i32。因為 i32 是 Copy，這會產生出另一個 i32 值。
    let shared = Rc::new(42);
    let number: i32 = *shared;
    println!("number：{}", number);

    // deref coercion：&Rc<i32> -> &i32
    show(&shared);

    // deref coercion 可以連鎖：&Rc<Box<i32>> -> &Box<i32> -> &i32
    let nested = Rc::new(Box::new(99));
    show(&nested);

    // method call 自動解參考：Rc<Vec<i32>> 能呼叫 Vec<i32> 的方法。
    let numbers = Rc::new(vec![10, 20, 30]);
    println!("長度：{}", numbers.len());

    // DerefMut：Box<String> 能可變地借用內部的 String。
    let mut text = Box::new(String::from("hello"));
    text.push_str(" world");
    println!("{}", text);

    *text = String::from("replaced");
    println!("{}", text);

    // 方法同名的優先順序：Rc 的 .clone() 勝過 String 的 .clone()。
    let a = Rc::new(String::from("shared"));
    let b = a.clone();    // Rc 的 .clone()：增加計數
    let c = (*a).clone(); // String 的 .clone()：建立一個新的 String
    println!("a = {}, b = {}, c = {}", a, b, c);
    println!("Rc 計數 = {}", Rc::strong_count(&a)); // 2，不是 3

    // Box<T> 的特權：擁有 Box 就能把 T 搬出來。
    let boxed = Box::new(String::from("owned"));
    let owned: String = *boxed;
    println!("從 Box 搬出來：{}", owned);
}
```

5.23.4 重點整理

- Deref 的主軸是「穿過一個值去借用」：它讓 Rust 拿到指向內部值的參考。
- 對 Deref 型別使用 *v，就是「先借用，再沿著參考走過去」；之後能複製、改寫還是 move，由內部值的型別和這個運算式的用法決定。
- deref coercion 會自動把 &Rc<i32> 這類參考轉成 &i32，而且可以連鎖穿過多層。

- method call 的自動解參考讓智慧指標能呼叫內部值的方法。
- DerefMut 提供對內部值的可變存取；Box<T> 支援，Rc<T> 不支援。
- 方法同名時外層優先——Rc 的 .clone() 會先於內部值的 .clone() 被選中。
- 用 *box_value 把非 Copy 的值搬出來是 Box<T> 專屬的特殊待遇，不是一般 Deref 的行為。

5.24 Cell<T>

5.24.1 本集目標

學會用 Cell<T> 在共享參考的情況下修改值，理解它的限制。

5.24.2 概念說明

第 4 章學了借用規則：要嘛一個 &mut，要嘛多個 &，不能同時。這很安全，但有時候你在只有 &（共享參考）的情況下，還是想修改值。

5.24.2.1 Cell 的概念

Cell<T> 提供內部可變性——即使只有共享參考，也能用 .get() 複製值、用 .set(v) 替換值，**不需要可變參考**。

```
use std::cell::Cell;

fn main() {
    let x = Cell::new(42);
    x.set(100);              // 不需要 mut！
    println!("{}", x.get()); // 100
}
```

但 .get() 有一個重要限制：

5.24.2.2 要使用 .get()，T 必須是 Copy

Cell<T> 的 .get() 會把值**複製一份**出來（不是借用）。所以要使用 .get()，T 必須實作 Copy。

你不能在 Cell<String> 上呼叫 .get()，因為 String 不是 Copy。只有 Copy 型別能使用 .get()（i32、f64、bool 等）。

5.24.2.3 為什麼不要用 mut？

有些情況下你不方便拿到 &mut。比如一個 struct 被多處共享參考（&self），但你想修改裡面的某個計數器。Cell 就很適合這種場景。

5.24.2.4 Rc 就是用 Cell 實作的

前面學的 Rc<T> 需要一個參考計數器——每次 clone 時計數 +1，drop 時計數 -1。但看看 Clone 的簽名：fn clone(&self) -> Self，它只拿得到 &self（共享參考），卻必須把計數 +1。怎麼辦？答案就是用 Cell！Rc 內部的計數器就是 Cell<usize>，所以即使只有 &self 也能更新計數。

5.24.3 範例程式碼

```
use std::cell::Cell;

struct Counter {
```

```
    count: Cell<i32>,
    name: String,
}

impl Counter {
    fn new(name: String) -> Counter {
        Counter {
            count: Cell::new(0),
            name,
        }
    }

    // 注意：只需要 &self，不需要 &mut self
    fn increment(&self) {
        let current = self.count.get();
        self.count.set(current + 1);
    }

    fn get_count(&self) -> i32 {
        self.count.get()
    }
}

fn main() {
    // 基本用法
    let x = Cell::new(42);
    println!("原始值：{}", x.get());

    x.set(100);
    println!("修改後：{}", x.get());

    // 在 struct 裡使用 Cell
    let counter = Counter::new(String::from("訪問次數"));

    // 只有 &counter（共享參考），但可以修改 count
    counter.increment();
    counter.increment();
    counter.increment();

    println!("{} 的計數：{}", counter.name, counter.get_count());
}
```

5.24.4 重點整理

- `Cell<T>` 讓你在不需要 `&mut` 的情況下修改值。
- `.get()` 複製值出來，`.set(v)` 寫入新值。
- **要使用 `.get()`，`T` 必須是 `Copy`**——因為 `get` 是複製，不是借用。
- 適合用在「只有 `&self` 但想修改某個欄位」的場景。

5.25 RefCell<T>

5.25.1 本集目標

學會用 RefCell<T> 在執行時期檢查借用規則，搭配 Rc 實現可變的共享資料。

5.25.2 概念說明

上一集學了 Cell<T>，但它的 .get() 要求 T 必須是 Copy。如果你想修改 String 或 Vec，並借用裡面的值來讀寫呢？

5.25.2.1 RefCell：執行時期的借用檢查

RefCell<T> 和 Cell 類似——讓你在不需要 &mut 的情況下修改值。差別在於：

- Cell<T>：用 .get() / .set()；其中 .get() 要求 T 必須是 Copy，而且是**零成本**（編譯後和直接存取沒有差別）。
- RefCell<T>：用 .borrow() 和 .borrow_mut() 取得用起來類似共享參考和可變參考的值，T **不需要** Copy，但**有執行時期成本**（每次借用都要檢查有沒有違反規則）。

```
use std::cell::RefCell;

fn main() {
    let x = RefCell::new(String::from("hello"));
    x.borrow_mut().push_str(" world"); // 修改裡面的 String
    println!("{}", x.borrow());        // 借用來讀
}
```

5.25.2.2 執行時期檢查

普通的 & 和 &mut 是在**編譯時期**檢查借用規則。RefCell 把這個檢查移到了**執行時期**。規則一模一樣（一個 &mut 或多個 &），只是違反時不是編譯錯誤，而是 **panic**。

```
    let x = RefCell::new(42);
    let a = x.borrow();     // 不可變借用
    let b = x.borrow_mut(); // panic！已經有不可變借用了
```

所以 RefCell 不是「繞過」借用規則，而是「延後檢查」。

5.25.2.3 Rc + RefCell：可變的共享資料

Rc<T> 提供的是共享的讀取：好幾個 Rc 值可以讀同一份資料，但不能改。RefCell<T> 提供的是內部可變性：只有共享參考也能改，但要讓好幾個地方都拿得到，還是得靠 Rc。把它們組合起來：

```
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let shared = Rc::new(RefCell::new(vec![1, 2, 3]));
}
```

這樣好幾個 Rc 值就共享同一份資料，而且透過 borrow_mut() 可以修改它。

5.25.3 範例程式碼

```
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    // 基本的 RefCell 用法
    let data = RefCell::new(String::from("hello"));

    // 不可變借用
    {
        let borrowed = data.borrow();
        println!("讀取：{}", borrowed);
    } // borrowed 離開作用域，釋放借用

    // 可變借用
    {
        let mut borrowed_mut = data.borrow_mut();
        borrowed_mut.push_str(" world");
    } // borrowed_mut 離開作用域，釋放借用

    println!("修改後：{}", data.borrow());

    // 違反借用規則 → panic！
    // 取消下面的註解就會在執行時 panic
    // {
    //     let r1 = data.borrow();      // 不可變借用
    //     let r2 = data.borrow_mut();  // 同時可變借用 → panic!
    // }

    // Rc + RefCell：可變的共享資料
    let shared = Rc::new(RefCell::new(vec![1, 2, 3]));

    let a = shared.clone();
    let b = shared.clone();

    // 透過 a 修改
    a.borrow_mut().push(4);

    // 透過 b 也能看到修改
    println!("透過 b 讀取：{:?}", b.borrow());

    // 透過 b 修改
    b.borrow_mut().push(5);

    // 透過 a 也能看到
    println!("透過 a 讀取：{:?}", a.borrow());
}
```

5.25.4 重點整理

- `RefCell<T>` 和 `Cell` 類似——讓你在不需要 `&mut` 的情況下修改值。
- `RefCell<T>` 把借用規則的檢查從編譯時期移到執行時期。
- `.borrow()` 取得用起來類似共享參考的值，`.borrow_mut()` 取得用起來類似可變參考的值。

- `.borrow()` 和 `.borrow_mut()` 不要求 T 必須是 Copy，這和 `Cell<T>` 的 `.get()` 不同。
- `Cell` 是零成本的，但 `RefCell` 每次借用都有執行時期檢查的開銷。
- 違反借用規則時會 **panic**（不是編譯錯誤）。
- `Rc<RefCell<T>>` 組合：可變的共享資料。

5.26 生命週期基礎

5.26.1 本集目標

理解為什麼需要生命週期標注 `'a`，學會在函數回傳參考時標注生命週期。

5.26.2 概念說明

第 4 章講借用的時候，我們留了一個伏筆：「不能回傳區域變數的參考。」現在來正式面對這個問題。

5.26.2.1 問題一：回傳區域變數的參考

```
fn make_greeting() -> &str {
    let s = String::from("哈囉");
    &s // 編譯錯誤！
} // s 在這裡被釋放了，回傳的參考指向一塊已經不存在的記憶體
```

這個比較好理解——s 離開函數就沒了，回傳它的參考毫無意義。Rust 直接擋掉。

5.26.2.2 問題二：多個參考，回傳哪一個？

但這個情況就比較複雜了：

```
fn longer(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        a
    } else {
        b
    }
}
```

這段程式碼也會編譯失敗。a 和 b 都是外面傳進來的參考，不會在函數結束時消失，那為什麼不行？

因為 Rust 在檢查**呼叫端**的時候，需要知道回傳值的參考能「活多久」。看這個例子：

```
    let s1 = String::from("hello world");
    let result;
    {
        let s2 = String::from("hi");
        result = longer(&s1, &s2);
    } // s2 在這裡被釋放了
    println!("{}", result); // result 到底還能不能用？
```

如果 `longer` 回傳了 a（也就是 &s1），`result` 是安全的，因為 s1 還活著。但如果回傳了 b（也就是 &s2），`result` 就是懸垂參考——s2 已經被釋放了。

問題是：**編譯器在檢查 `longer` 的呼叫端時，不會去看 `longer` 的函數體**。它只看函數簽名。而簽名上寫 `-> &str`，沒有任何資訊告訴它回傳值和哪個參數的壽命有關。

5.26.2.3 生命週期標注 'a

解法是用生命週期標注，明確描述回傳值和參數的關係：

```
fn longer<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() > b.len() {
        a
    } else {
        b
    }
}
```

'a 是一個**生命週期參數**（和型別參數 T 類似，但用 ' 開頭）。這段簽名告訴 Rust：「a、b 和回傳值都標注了同一個 'a。所以回傳值的壽命不能超過 a 和 b 中**較短**的那個。」注意生命週期參數和型別參數一樣寫在 <> 裡面。如果同時有生命週期和型別參數，**生命週期要寫在前面**：`fn foo<'a, T>(x: &'a T) -> &'a T`。

5.26.2.4 為什麼是較短的？

因為 a 和 b 共用同一個 'a，Rust 會取兩者的**交集**——也就是兩者都還活著的那段時間。

回到剛才的例子：

```
fn longer<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() > b.len() {
        a
    } else {
        b
    }
}

fn main() {
    let s1 = String::from("hello world"); // s1 的壽命比較長
    let result;
    {
        let s2 = String::from("hi");      // s2 的壽命比較短
        result = longer(&s1, &s2);
        println!("{}", result);           // ✅ 這裡 s1 和 s2 都還活著
    } // s2 在這裡被釋放
    // println!("{}", result);            // ❌ 不行！'a 也會隨 s2 結束
}
```

'a 被推斷為 s2 的壽命（較短的那個），所以 result 只能在 s2 還活著的範圍內使用。

5.26.2.5 &'a mut T

可變參考也可以加生命週期標注，寫成 `&'a mut T`——就是把 'a 放在 & 和 mut 之間。'a 一樣描述這個參考能活多久。

```
fn replace<'a>(target: &'a mut String, new_value: &str) {
    target.clear();
    target.push_str(new_value);
}
```

5.26.2.6 生命週期不改變壽命

重要觀念：生命週期標注不會讓任何參考活得更久或更短。它只是**描述**已有的關係，幫助編譯器做檢查。就像型別標注不會改變值的內容一樣。

5.26.2.7 不是所有函數都要標

如果函數只有一個參考參數，Rust 通常能自動推斷（下一集會詳細講）：

```
fn first_byte(s: &str) -> &str {
    &s[..1] // 回傳值的壽命顯然和 s 一樣，不用手動標
}
```

5.26.2.8 'static 生命週期

有一個特殊的生命週期：'static，表示「活到程式結束」。

字串字面值就是 'static——"hello" 的型別是 &'static str，因為字串字面值被寫死在程式碼裡，整個程式執行期間都存在。

5.26.3 範例程式碼

```
// 回傳參考時，需要標注生命週期
fn longer<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() > b.len() {
        a
    } else {
        b
    }
}

// 回傳值只和 a 有關，b 不影響
fn always_first<'a>(a: &'a str, _b: &str) -> &'a str {
    a
}

fn main() {
    // 例子一：兩個參數壽命一樣長
    let s1 = String::from("很長的字串");
    let s2 = String::from("短");
    let result = longer(&s1, &s2);
    println!("比較長的是：{}", result);

    // 例子二：兩個參數壽命不同
    let s3 = String::from("hello world");
    let r;
    {
        let s4 = String::from("hi");
        r = longer(&s3, &s4);
        println!("在作用域內：{}", r); // ✅ s3 和 s4 都還活著
    }
    // println!("{}", r); // ❌ 編譯錯誤！s4 已經被釋放，r 的生命週期不夠長

    // 例子三：回傳值只借用其中一個參數
    let s5 = String::from("我會被回傳");
    let r2;
```

```
    {
        let s6 = String::from("我不會");
        r2 = always_first(&s5, &s6);
    }
    // r2 只借用 s5，所以即使 s6 被釋放也沒關係
    println!("{}", r2); // ✅ s5 還活著，r2 可以用

    // 'static 生命週期
    let s: &'static str = "我是靜態字串，活到程式結束";
    println!("{}", s);
}
```

5.26.4 重點整理

- 當函數回傳參考時，Rust 需要知道回傳值能活多久——這就是生命週期標注的目的。
- `'a` 是生命週期參數，描述參考之間的壽命關係。
- 多個參數共用同一個 `'a` 時，Rust 取交集（較短的那個）。
- 生命週期標注**不改變壽命**，只是描述已有的關係。
- `'static` 表示「活到程式結束」——字串字面值的型別是 `&'static str`。

5.27 生命週期省略規則

5.27.1 本集目標

理解 Rust 的生命週期省略規則，知道為什麼大部分時候不需要手動寫生命週期標注。

5.27.2 概念說明

上一集我們只有在 `longer` 這種「回傳值可能來自不同參考參數」的情況下手動寫了 `'a`。那其他回傳參考的函數呢？是不是也都要寫？

好消息是：大部分時候不用。Rust 有一套**省略規則（elision rules）**，會自動幫你補上生命週期標注。

5.27.2.1 三條省略規則

Rust 編譯器按照這三條規則嘗試推斷生命週期：

規則一：每個參數能放生命週期的位置各自獲得獨立的生命週期

```
fn foo(a: &str, b: &str)
// 編譯器看成：fn foo<'a, 'b>(a: &'a str, b: &'b str)
```

規則二：如果經過規則一之後只有一個 input lifetime，回傳值的生命週期就等於它

```
fn first_word(s: &str) -> &str
// 規則一：fn first_word<'a>(s: &'a str) -> &str
// 規則二：只有一個 input lifetime 'a → fn first_word<'a>(s: &'a str) -> &'a str
```

這就是為什麼上面的 `first_word` 不用寫 `'a`——只有一個 input lifetime，規則二自動搞定。

注意一個參數可能帶有多個 input lifetime——比如 `&'a &'b T`（參考的參考）就有兩個（`'a` 和 `'b`）。如果有兩個以上的 input lifetime，規則二就不適用了。

規則三：如果有 `&self` 或 `&mut self` 參數，回傳值的生命週期就等於 `self` 的

```
impl MyStruct {
    fn name(&self) -> &str { ... }
    // 編譯器看成：fn name<'a>(&'a self) -> &'a str
}
```

5.27.2.2 什麼時候規則不夠用？

當有多個參考參數、但回傳值的生命週期不確定跟哪個綁定時——就是上一集 `longer` 函數的情況。這時候就必須手動標注。

5.27.2.3 總結

- 一個參考參數 → 幾乎不用寫。
- method 回傳 `&self` 的一部分 → 不用寫。

5.27.3 範例程式碼

```
// 規則二：一個 input lifetime，自動推斷
fn trim_hello(s: &str) -> &str {
    if s.len() >= 5 {
        &s[5..]
    } else {
        s
    }
}

struct Article {
    title: String,
    content: String,
}

impl Article {
    fn new(title: String, content: String) -> Article {
        Article { title, content }
    }

    // 規則三：&self 參數，回傳值生命週期跟 self 綁定
    fn title(&self) -> &str {
        &self.title
    }

    fn summary(&self) -> &str {
        &self.content
    }
}

// 多個參考參數 + 回傳參考 → 需要手動標注
fn pick_longer<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() >= b.len() {
        a
    } else {
        b
    }
```

```
}

fn main() {
    // 規則二：不用寫生命週期
    let greeting = String::from("Hello, world!");
    let trimmed = trim_hello(&greeting);
    println!("{}", trimmed);

    // 規則三：method 接收參考不用寫生命週期
    let article = Article::new(
        String::from("Rust 生命週期"),
        String::from("其實沒那麼可怕"),
    );
    println!("標題：{}", article.title());
    println!("摘要：{}", article.summary());

    // 多個參考參數：需要手動標注
    let a = String::from("hello");
    let b = String::from("hi");
    let result = pick_longer(&a, &b);
    println!("比較長的：{}", result);
}
```

5.27.4 重點整理

- Rust 有三條**省略規則**，大部分時候會自動補上生命週期標注。
- 規則一：每個參數能放生命週期的位置各自獲得獨立的生命週期。
- 規則二：只有一個 input lifetime → 回傳值的生命週期自動等於它。
- 規則三：method 有 `&self` 或 `&mut self` → 回傳值的生命週期自動等於 `self`。

5.28 型別上的生命週期

5.28.1 本集目標

學會為包含參考的 `struct` 和 `enum` 標注生命週期，以及用 `'_` 匿名生命週期簡化標注。

5.28.2 概念說明

到目前為止，我們的 `struct` 和 `enum` 都擁有自己的資料（`String`、`i32` 等）。但有時候你想讓它們借用別人的資料——例如存一個 `&str` 而不是 `String`。

5.28.2.1 型別裡放參考

```
struct Excerpt {
    text: &str, // 編譯錯誤！
}
```

這會報錯。因為 Rust 需要知道：「這個 `&str` 能活多久？」如果借來的資料被釋放了，`struct` 裡的參考就變成懸垂參考。

解法是加上生命週期參數：

```
struct Excerpt<'a> {
    text: &'a str,
}
```

'a 告訴 Rust：「這個 struct 的壽命不能超過它借用的資料。」

enum 也一樣——如果 variant 攜帶參考，就需要生命週期：

```
enum Token<'a> {
    Word(&'a str),
    Number(i32),
}
```

Token::Word 借用了一段文字，所以 Token 的壽命不能超過那段文字。Token::Number 本身不包含任何參考，但它和 Word 是同一個 enum，所以 Token::Number(42) 的型別一樣是 Token<'a>——這個 'a 由編譯器自己推斷，你不用寫，而且它對 Number 來說不起實際作用。

5.28.2.2 使用帶生命週期的型別

```
    let novel = String::from("很長的故事...");
    let excerpt = Excerpt { text: &novel };
```

excerpt 借用了 novel 的資料，所以 excerpt 不能活得比 novel 更久。

5.28.2.3 '_ 匿名生命週期

當生命週期可以被推斷的時候，你可以用 '_ 來簡化：

```
fn print_excerpt(e: &Excerpt<'_>) {
    println!("{}", e.text);
}
```

'_ 告訴 Rust「我知道這裡需要一個生命週期，你自己推斷吧」。還記得第 5 集學的型別佔位符 _ 嗎？'_ 就是它的生命週期版本。

5.28.2.4 impl 帶生命週期的 struct

```
impl<'a> Excerpt<'a> {
    fn text(&self) -> &str {
        self.text
    }
}
```

和泛型 struct 的 impl 一樣——impl<'a> 宣告生命週期參數，Excerpt<'a> 使用它。

注意 fn text(&self) -> &str 不需要寫任何生命週期標注——上一集學的省略規則第三條在這裡生效了：method 有 &self 時，回傳值的生命週期自動等於 self。

5.28.2.5 帶 lifetime 的型別作為函數參數

如果函數接收帶 lifetime 的型別，可以搭配 '_ 讓編譯器推斷：

```
fn into_text(e: Excerpt<'_>) -> &str {
    e.text
}
```

完整寫出來是：

```
fn into_text<'a>(e: Excerpt<'a>) -> &'a str {
    e.text
}
```

省略規則看到 Excerpt<'_> 帶有一個 input lifetime，規則二把回傳值的生命週期也設為同一個。

注意這裡 e 本身不是參考，函數結束時 e 會被 drop。但回傳的 &'a str 不是借用 e，而是借用 e 裡面存的那段文字——那段文字的壽命是 'a，跟 e 本身的壽命無關。

5.28.3 範例程式碼

```
// struct 裡放參考，需要生命週期標注
struct Excerpt<'a> {
    text: &'a str,
    page: i32,
}

impl<'a> Excerpt<'a> {
    fn new(text: &'a str, page: i32) -> Excerpt<'a> {
        Excerpt { text, page }
    }

    fn text(&self) -> &str {
        self.text
    }

    fn summary(&self) -> String {
        let mut s = String::from("第 ");
        let page_str = self.page.to_string();
        s.push_str(&page_str);
        s.push_str(" 頁：");
        s.push_str(self.text);
        s
    }
}

// 用 '_ 匿名生命週期
fn print_excerpt(e: &Excerpt<'_>) {
    println!("[p.{}] {}", e.page, e.text);
}

fn main() {
    let novel = String::from("在很久很久以前，有一個程式設計師...");

    // excerpt 借用了 novel 的資料
    let excerpt = Excerpt::new(&novel[..15], 1);
    println!("{}", excerpt.text());
    println!("{}", excerpt.summary());

    // 用匿名生命週期的函數
    print_excerpt(&excerpt);

    // excerpt 不能活得比 novel 更久
    // 如果 novel 被 drop 了，excerpt 就不能用了
```

```
}
```

5.28.4 重點整理

- `struct` 裡放參考時，必須標注生命週期：`struct Excerpt<'a> { text: &'a str }`。
- 生命週期保證 `struct` 不會活得比借用的資料更久。
- `'_` 是匿名生命週期，讓編譯器自己推斷（生命週期版的 `_`）。
- `impl` 帶生命週期的 `struct`：`impl<'a> Excerpt<'a> { ... }`。

5.29 lifetime bound

5.29.1 本集目標

學會 `T: 'a` 這種 lifetime bound，理解為什麼 `&'a T` 需要 `T` 裡面的參考都活得過 `'a`。

5.29.2 概念說明

5.29.2.1 問題：T 裡面可能有參考

到目前為止，我們的泛型函數大多處理 `i32`、`String` 這些擁有自己資料的型別。但 `T` 也可能是 `&str` 或其他包含參考的型別。

看這個 `struct`：

```
struct Ref<'a, T> {
    value: &'a T,
}
```

如果 `T` 是 `&'x str`，那 `value` 就是 `&'a &'x str`——一個參考指向另一個參考。這時候 `'x` 必須活得至少和 `'a` 一樣長，否則外層的 `&'a` 還活著的時候，裡面的 `&'x str` 可能已經失效了。

5.29.2.2 T: 'a 的意思

`T: 'a` 是一個 **lifetime bound**，表示「T 裡面的所有參考都活得過 `'a`」。

如果 `T` 是 `i32`（沒有參考），`T: 'a` 自動滿足。如果 `T` 是 `&'x str`，那 `T: 'a` 就要求 `'x` 至少活得和 `'a` 一樣長。

5.29.2.3 什麼時候要寫？

在很多情況下，編譯器看到 `&'a T` 就知道需要 `T: 'a`，會自動幫你加上。但在某些 `trait` 定義或比較複雜的泛型結構裡，你可能需要手動寫：

```
struct Ref<'a, T: 'a> {
    value: &'a T,
}
```

這裡的 `T: 'a` 其實是多餘的（編譯器能從 `&'a T` 推出來），但手動寫出來也不會錯，而且讓意圖更清楚。

5.29.2.4 參考帶生命週期的型別

同樣的道理推廣到任何帶生命週期的型別。如果你有 `&'b A<'a>`——一個活 `'b` 那麼久的參考，指向一個 `A<'a>`——那 `A<'a>` 整體必須在 `'b` 的期間都是有效的。這意味著 `A` 裡面借用的資料必須活得過 `'b`，也就是 `'a` 必須至少和 `'b` 一樣長。

原因很直覺：你持有一個參考 &'b，透過它可以存取 A 裡面所有借用的資料。如果 A 借用的資料比你持有參考的時間更早失效，你就能存取到已經被回收的記憶體。所以 Rust 要求 'a 至少活得和 'b 一樣長。

5.29.3 範例程式碼

```
struct Excerpt<'a> {
    text: &'a str,
}

// T: 'a 確保 T 裡的參考活得過 'a
struct Ref<'a, T: 'a> {
    value: &'a T,
}

impl<'a, T: 'a> Ref<'a, T> {
    fn new(value: &'a T) -> Ref<'a, T> {
        Ref { value }
    }

    fn get(&self) -> &T {
        self.value
    }
}

fn main() {
    // T = i32（沒有參考，T: 'a 自動滿足）
    let num = 42;
    let r = Ref::new(&num);
    println!("Ref<i32>: {}", r.get());

    // T = &str（T 本身就是參考）
    let text = String::from("hello");
    let slice: &str = &text;
    let r2 = Ref::new(&slice);
    println!("Ref<&str>: {}", r2.get());

    // &'b A<'a> 的例子
    let novel = String::from("很長的故事...");
    let excerpt = Excerpt { text: &novel };
    let r3 = &excerpt; // &'b Excerpt<'a>
    // 這裡 'a 是 novel 的壽命，'b 是 r3 借用 excerpt 的時間
    // novel 至少活得和 r3 一樣久，所以 'a 活得過 'b，條件滿足
    println!("透過參考讀取：{}", r3.text);

    // T = String（擁有資料，沒有參考，T: 'a 自動滿足）
    let s = String::from("world");
    let r3 = Ref::new(&s);
    println!("Ref<String>: {}", r3.get());
}
```

5.29.4 重點整理

- T: 'a 表示 T 裡面的所有參考都活得過 'a。

- 如果 T 沒有參考（如 i32、String），T: 'a 自動滿足。
- &'a T 要合法就需要 T: 'a——大部分情況編譯器能自動推斷。
- 理解 lifetime bound 才能讀懂標準庫裡比較複雜的泛型。

5.30 supertrait

5.30.1 本集目標

學會用 supertrait 定義 trait 之間的依賴關係，理解 Copy: Clone 以及 DerefMut: Deref 的設計原理。

5.30.2 概念說明

有時候一個 trait 需要建立在另一個 trait 的基礎之上。

5.30.2.1 supertrait 語法

```
trait Summarize: std::fmt::Display {
    fn summary(&self) -> String;
}
```

Summarize: Display 的意思是：「要實作 Summarize，你必須先實作 Display。」Display 就是 Summarize 的 **supertrait**，反過來說，Summarize 是 Display 的 **subtrait**。

好處是在 Summarize 的預設實作或使用者程式碼裡，可以確定 self 一定實作了 Display。

注意：**實作 Summarize 不會自動幫你實作 Display**。你必須自己手動實作 Display，然後才能實作 Summarize。supertrait 只是一個「前提條件」，不是「自動贈送」。

5.30.2.2 Copy: Clone

第 4 章學過 Copy 和 Clone。它們之間就是 supertrait 的關係：

```
trait Copy: Clone { }
```

這表示：**要實作 Copy，必須先實作 Clone。**

為什麼？因為 Copy 是一種「自動複製」的能力，而 Clone 是「手動複製」的能力。邏輯上，如果你能自動複製，那你也一定能手動複製。所以 Copy 要求 Clone 作為前提。

這就是為什麼 #[derive(Copy, Clone)] 會同時寫上兩個。在這個例子中，只寫 derive(Copy) 會失敗，因為 Point 沒有用其他方式實作 Clone。如果手動實作了 Clone，就可以單獨 derive Copy。

5.30.2.3 DerefMut: Deref

第 23 集學的 DerefMut 也是一樣的道理——DerefMut 的 supertrait 是 Deref。要能可變解參考，前提是先要能不可變解參考。所以實作了 DerefMut 的型別一定也實作了 Deref。

5.30.3 範例程式碼

```
use std::fmt::Display;
use std::fmt::Formatter;

// 定義一個 supertrait：Summarize 要求 Display
```

```rust
trait Summarize: Display {
    fn summary(&self) -> String {
        // supertrait bound 要求 Display，
        // 所以 Rust 也會透過 ToString 提供 .to_string()
        let full = self.to_string();
        // 把字元收進 Vec，用「字元數」判斷長度
        //  （字串的 .len() 算的是 byte 數）
        let mut chars = Vec::new();
        for c in full.chars() {
            chars.push(c);
        }
        if chars.len() > 10 {
            let mut s = String::new();
            // 取前 10 個字元
            for c in &chars[..10] {
                s.push(*c);
            }
            s.push_str("...");
            s
        } else {
            full
        }
    }
}

struct Article {
    title: String,
    content: String,
}

// 必須先實作 Display（supertrait）
impl Display for Article {
    fn fmt(&self, f: &mut Formatter) -> std::fmt::Result {
        write!(f, "{}: {}", self.title, self.content)
    }
}

// 然後才能實作 Summarize
impl Summarize for Article {}

// Copy: Clone 的示範
#[derive(Debug, Clone, Copy)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let article = Article {
        title: String::from("Rust"),
        content: String::from("一門很棒的程式語言，值得學習"),
    };

    // 用 Display（supertrait）
    println!("完整：{}", article);
```

```
    // 使用 Summarize 的預設實作：
    // .to_string() 來自 ToString，而 Display 讓它可以使用。
    println!("摘要：{}", article.summary());

    // Copy 需要 Clone 的示範
    let p = Point { x: 1, y: 2 };
    let p2 = p; // copy（自動複製）
    let p3 = p.clone(); // clone（手動複製）也可以用
    println!("{:?} {:?} {:?}", p, p2, p3);
}
```

5.30.4 重點整理

- `trait A: B` 表示「要實作 A，必須先實作 B」——B 是 A 的 supertrait，A 是 B 的 subtrait。
- `Copy: Clone`——實作 Copy 的型別也必須實作 `Clone`；兩者通常一起 derive，但 `Clone` 也可以手動實作。
- `DerefMut: Deref`——要能可變解參考，必須先能不可變解參考。
- 實作 subtrait 不會自動實作 supertrait——你必須自己先寫 `impl Supertrait`。
- subtrait 的預設實作裡，可以依賴 supertrait 保證的能力。

5.31 常見的 derive trait

5.31.1 本集目標

學會 `PartialEq`、`Eq`、`PartialOrd`、`Ord`、`Default` 等常見 `derive trait` 的用途和差別。

5.31.2 概念說明

第 4 章我們學了 `Debug`、`Clone`、`Copy`。Rust 標準庫還有其他可以 `derive` 的 `trait`，今天來認識最常用的幾個。

5.31.2.1 PartialEq 和 Eq

`PartialEq` 讓你的型別可以用 `==` 和 `!=` 比較。

```
#[derive(PartialEq)]
struct Point { x: i32, y: i32 }
```

`Eq` 建立在 `PartialEq` 之上——`PartialEq` 是 `Eq` 的 supertrait（上一集學的）。`Eq` 額外保證**自反性**——每個值都等於自己。

「等一下，什麼值不等於自己？」——`f64::NAN`！在浮點數規範裡，`NAN != NAN`。所以 `f64` 只有 `PartialEq`，沒有 `Eq`。

如果你的型別不包含浮點數，通常 `PartialEq` 和 `Eq` 都可以 `derive`。

5.31.2.2 PartialOrd 和 Ord

`PartialOrd` 讓你的型別可以用 `<`、`>`、`<=`、`>=` 比較。

`Ord` 是完整排序——保證任意兩個值都能比大小。`f64` 因為有 `NAN`，所以只有 `PartialOrd`，沒有 `Ord`。

NAN 和任何值做 <、>、<=、>=、== 都是 false——包括和它自己比。只有 != 例外，一律是 true：

```
fn main() {
    let nan = f64::NAN;
    println!("{}", nan < 1.0);  // false
    println!("{}", nan > 1.0);  // false
    println!("{}", nan == nan); // false
    println!("{}", nan <= nan); // false
}
```

這就是為什麼 f64 不能有 Ord——你沒辦法把 NAN 放進一個排序裡，因為它和誰都沒有大小關係，沒有一個合理的位置可以放它。

5.31.2.3 四個 trait 的完整關係

先看它們的定義（簡化版）：

```
pub trait PartialEq { ... }
pub trait Eq: PartialEq { }
pub trait PartialOrd: PartialEq { ... }
pub trait Ord: PartialOrd + Eq { ... }
```

整理成繼承關係：

- Eq: PartialEq — 要有完整等價，先要有部分等價。
- PartialOrd: PartialEq — 要能比大小，先要能比相不相等（因為 <= 包含了 ==）。
- Ord: PartialOrd + Eq — 要有完整排序，先要有部分排序**和**完整等價。

為什麼 PartialOrd 要求 PartialEq？因為「比大小」本身隱含了「能判斷相等」——如果 a <= b 且 b <= a，那 a == b。

為什麼 Ord 要求 Eq？因為完整排序必須能比較任意兩個值，包括相等的情況。而且 Ord 保證所有值都有確定的位置，所以不允許 NAN 這種「和自己不相等」的值。

這就是為什麼 f64 只能走一邊（PartialEq + PartialOrd），無法走到另一邊（Eq + Ord）。

5.31.2.4 Default

Default trait 提供一個「預設值」。數字的預設值是 0，bool 是 false，String 是空字串，Vec 是空 Vec。

如果 struct 的每個欄位都有 Default，你就可以 derive 它：

```
#[derive(Debug, Default)]
struct Config {
    width: i32,
    height: i32,
    title: String,
}

fn main() {
    let config = Config::default();
    // Config { width: 0, height: 0, title: "" }
}
```

5.31.3 範例程式碼

```rust
#[derive(Debug, PartialEq, Eq, PartialOrd, Ord, Clone, Default)]
struct Student {
    grade: i32,
    name: String,
}

fn main() {
    let alice = Student { grade: 90, name: String::from("Alice") };
    let bob = Student { grade: 85, name: String::from("Bob") };
    let alice2 = Student { grade: 90, name: String::from("Alice") };

    // PartialEq：== 和 !=
    println!("alice == alice2: {}", alice == alice2);
    println!("alice == bob: {}", alice == bob);
    println!("alice != bob: {}", alice != bob);

    // PartialOrd：< > <= >=
    // derive 的 Ord 按欄位順序比較（先比 grade，再比 name）
    println!("alice > bob: {}", alice > bob);
    println!("bob < alice: {}", bob < alice);

    // 排序需要 Ord
    let mut students = vec![
        Student { grade: 70, name: String::from("Charlie") },
        Student { grade: 90, name: String::from("Alice") },
        Student { grade: 85, name: String::from("Bob") },
    ];

    students.sort();
    for s in &students {
        println!("{}: {}", s.name, s.grade);
    }

    // f64 的特殊情況：NAN
    let nan = f64::NAN;
    println!("NAN == NAN: {}", nan == nan); // false！
    println!("NAN < 1.0: {}", nan < 1.0);   // false！
    println!("NAN > 1.0: {}", nan > 1.0);   // false！

    // f64 沒有 Ord，所以不能用 .sort()
    // let mut floats = vec![1.0, 2.0, f64::NAN];
    // floats.sort(); // 編譯錯誤！f64 沒有實作 Ord

    // Default
    let default_student = Student::default();
    println!("預設學生：{:?}", default_student);
    // Student { grade: 0, name: "" }
}
```

5.31.4 重點整理

- PartialEq：==、!= 比較；Eq：保證自反性（NAN 是例外）。
- PartialOrd：<、>、<=、>= 比較；Ord：保證完整排序。

- `f64` 因為 `NAN` 的存在，只有 `Partial` 版本，沒有完整版。
- derive 的 `Ord` 按欄位宣告順序逐一比較。
- `Default`：提供預設值（數字 `0`、`bool` `false`、`String` 空字串）。

5.32 associated type

5.32.1 本集目標

學會在 `trait` 中定義 associated type（關聯型別），理解它和泛型參數的差別。

5.32.2 概念說明

第 18 集我們學了多參數 `trait`：`trait Convert<T>`。但有時候，型別參數不是「開放的」——一個型別只會有一種合理的實作。

5.32.2.1 問題：多參數 trait 太自由了

想像一個「容器」的 `trait`。容器裡面裝什麼型別的元素？用多參數 `trait` 的話：

```
trait Container<T> {
    fn first(&self) -> Option<&T>;
}
```

但這意味著同一個型別可以同時實作 `Container<i32>` 和 `Container<String>`——通常容器只會有一種元素型別。

5.32.2.2 associated type：一對一的關係

associated type 解決了這個問題：

```
trait Container {
    type Item;
    // 要使用 Self 的 associated type，用 Self::Type 的語法
    fn first(&self) -> Option<&Self::Item>;
}
```

`type Item;` 宣告了一個 associated type。實作的時候必須指定它是什麼：

```
impl Container for NumberList {
    type Item = i32;
    fn first(&self) -> Option<&i32> {
        self.data.first()
    }
}
```

當 `Self`（`NumberList`）和角括號裡的參數（這裡沒有）都確定了，`Item` 就唯一確定是 `i32`，不會有歧義。

5.32.2.3 和泛型參數的差別

你可以把 `trait` 想像成一個函數，它接受一些「輸入」然後決定一些「輸出」：

- **輸入（input）**：`Self`（誰來實作這個 `trait`）和角括號裡的型別參數（`<T>`）。
- **輸出（output）**：associated type（`type Item`）。

輸入決定了輸出——當你確定了「誰」（`Self`）和「角括號裡的參數」，associated type 就唯一確定

了。

舉例來說，Convert<T> 裡的 T 是輸入，所以同一個 Self 搭配不同的 T 可以有不同的實作：i32 可以同時實作 Convert<String> 和 Convert<(i32,)>。

但 Container 的 Item 是輸出。當你確定了 Self 是 NumberList，Item 就**只能有一個答案**——i32。

用哪個？如果「確定了所有 input 之後，這個型別就只有一個合理的答案」，把它放在 associated type（output）。如果「同一組 input 可以搭配多種不同答案」，把它放在角括號裡（input）。

5.32.2.4 Deref 也有 associated type

第 23 集學的 Deref trait 就用了 associated type：

```
trait Deref {
    type Target;
    fn deref(&self) -> &Self::Target;
}
```

type Target 就是解參考所觸及的型別。

舉例來說，如果 p 是 Box<i32>，而我們寫：

```
    let n = *p;
```

Rust 大致上會把 *p 這部分當成：

```
    let n = *p.deref();
```

deref 接收 &self，所以 p.deref() 會先借用這個智慧指標，再回傳一個指向內部值的參考：&Self::Target。最後，外面的 * 沿著這個參考走過去。

對 Box<i32> 來說，Self::Target 是 i32，所以 .deref() 回傳 &i32。因為 i32 是 Copy，let n = *p; 能產生出另一個 i32。

這跟 Container 的 type Item 是同樣的道理：一旦 Self 定為 Box<i32>，Target 就只有一個答案——i32。這正是 Target 用 associated type 而不是泛型參數的原因。

5.32.2.5 DerefMut 用的是同一個 Target

可變的版本是 DerefMut。簡化之後長這樣：

```
trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

DerefMut 沒有另外宣告一個 associated type，它用的就是 Deref 的 Self::Target。

這件事是有道理的：不可變解參考和可變解參考，必須觸及同一種內部值。如果 Box<String> 的 Target = String，那 .deref() 回傳 &String，.deref_mut() 就回傳 &mut String。

舉例來說：

```
    *p = String::from("world");
```

因為等號左邊是 *p，Rust 需要對內部值的可變存取。它大致上會把這部分當成：

```
    *p.deref_mut() = String::from("world");
```

deref_mut 接收 &mut self，所以 p.deref_mut() 會先可變地借用這個智慧指標，再回傳一個指

向內部值的可變參考：`&mut Self::Target`。最後，外面的 `*` 沿著這個可變參考走過去，賦值就能把內部的 `String` 換掉。

5.32.2.6 在 `trait` bound 中指定 associated type

你可以在 `trait` bound 裡指定 associated type 的具體型別：

```
fn print_first<C: Container<Item = i32>>(c: &C) { ... }
```

`Container<Item = i32>` 表示「實作了 `Container`，而且 `Item` 是 `i32`」。

如果只是要限制 associated type 必須實作某個 `trait`，也可以直接把 `trait` bound 寫在 associated type 後面：

```
fn print_first<C: Container<Item: Display>>(c: &C) { ... }
```

`Item: Display` 表示「`Container` 的 `Item` 必須實作 `Display`」。

5.32.3 範例程式碼

```
use std::fmt::Display;

// 用 associated type 定義容器 trait
trait Container {
    type Item;

    fn first(&self) -> Option<&Self::Item>;
    fn last(&self) -> Option<&Self::Item>;
    fn len(&self) -> usize;
}

struct NumberList {
    data: Vec<i32>,
}

impl Container for NumberList {
    type Item = i32; // 指定 associated type

    fn first(&self) -> Option<&i32> {
        self.data.first()
    }

    fn last(&self) -> Option<&i32> {
        self.data.last()
    }

    fn len(&self) -> usize {
        self.data.len()
    }
}

struct WordList {
    words: Vec<String>,
}

impl Container for WordList {
```

```
    type Item = String; // 不同的型別，不同的 Item

    fn first(&self) -> Option<&String> {
        self.words.first()
    }

    fn last(&self) -> Option<&String> {
        self.words.last()
    }

    fn len(&self) -> usize {
        self.words.len()
    }
}

// 在 trait bound 中用 associated type
fn print_first_item<C>(c: &C)
where
    C: Container,
    C::Item: Display,
{
    match c.first() {
        Some(item) => println!("第一個元素：{}", item),
        None => println!("容器是空的"),
    }
}

fn main() {
    let nums = NumberList { data: vec![10, 20, 30] };
    let words = WordList {
        words: vec![
            String::from("hello"),
            String::from("world"),
        ],
    };

    println!("數字容器長度：{}", nums.len());
    print_first_item(&nums);

    println!("文字容器長度：{}", words.len());
    print_first_item(&words);

    // last
    match nums.last() {
        Some(n) => println!("最後一個數字：{}", n),
        None => println!("空的"),
    }
}
```

5.32.4 重點整理

- `type Item;` 在 `trait` 中定義 associated type。
- 用 `Self::Item` 的語法可以在 `trait` 定義中讀取 `Self` 的 associated type。
- 實作時用 `type Item = i32;` 指定具體型別。

- **input vs output**：`Self` 和角括號參數是 input，associated type 是 output。input 決定 output。
- `Deref` 的 `type Target` 也是 associated type——`Box<T>` 的 `Target = T`，代表解參考觸及的是 `T`。
- `DerefMut` 用的是同一個 `Self::Target`，回傳 `&mut Self::Target`。
- 在 `trait` bound 中用 `Container<Item = i32>` 指定 associated type。
- 在 `trait` bound 中也可以用 `Container<Item: Display>` 限制 associated type 必須實作某個 `trait`。

5.33 `Cow<'a, B>`

5.33.1 本集目標

學會使用 `Cow<'a, str>` 實現「能借就借，需要時才 `clone`」的彈性策略。

5.33.2 概念說明

有些函數有時候可以直接回傳借用的資料，有時候又需要回傳擁有的資料。

5.33.2.1 舉個例子

假設你有一個函數，幫字串加上問候語。如果字串已經有「你好」開頭，直接回傳原字串就好（借用）。如果沒有，就要建一個新的字串（擁有）。

回傳型別是 `&str` 還是 `String`？兩個都不完全對。

5.33.2.2 `Cow` 來拯救

`Cow` 的全名是 **`Clone` on write**（寫入時才 `clone`）。它定義在 `std::borrow` 模組裡。來看一個教學用的簡化定義，先抓住核心結構；它省略了標準庫完整型別需要的部分，所以這段還不是可以直接拿來完整運作的實作：

```
enum Cow<'a, B>
where
    B: 'a + ToOwned,
{
    Borrowed(&'a B),
    Owned(B::Owned), // ToOwned 的 associated type
}
```

一行一行看：

- **`'a`**：生命週期參數，代表借用資料的壽命。
- **`B: 'a`**：lifetime bound（前幾集學的），`B` 裡面的參考必須活得過 `'a`。
- **`B: ToOwned`**：`trait` bound，`B` 必須實作 `ToOwned`。
- **`Borrowed(&'a B)`**：借用的版本，存一個 `&'a B`。
- **`Owned(...)`**：擁有所有權的版本，型別由 `ToOwned` 的 associated type `Owned` 決定。

`ToOwned` 是一個 `trait`，它有一個 associated type `Owned`，代表「擁有所有權版本的型別」。

對 `str` 來說：

- `str` 實作了 `ToOwned`，`type Owned = String`。
- 所以 `Cow<'a, str>` = `Borrowed(&'a str)` 或 `Owned(String)`。

對 [T] 來說：

- [T] 實作了 ToOwned，type Owned = Vec<T>。
- 所以 Cow<'a, [T]> = Borrowed(&'a [T]) 或 Owned(Vec<T>)。

5.33.2.3 Cow 實作了 Deref

這是使用 Cow 時最關鍵的一點：Cow<'a, B> 實作了 Deref<Target = B>。也就是說，不管裡面是 Borrowed(&str) 還是 Owned(String)，你都可以直接把 &Cow<'_, str> 當成 &str 來用——呼叫 &str 的方法或傳給接受 &str 的函數，完全不用管它實際上是借用還是擁有。

```
use std::borrow::Cow;

fn main() {
    let cow: Cow<'_, str> = Cow::Owned(String::from("hello"));
    // 直接當 &str 用，Deref 自動處理
    println!("長度：{}", cow.len());
    println!("大寫：{}", cow.to_uppercase());
}
```

因為有 Deref，呼叫端通常不需要在意裡面到底是借用還是擁有——直接當 &str 用就好。

5.33.2.4 常用方法

- **.to_mut()**：如果是 Borrowed，先 clone 成 Owned，然後回傳可變參考。如果已經是 Owned，直接回傳它的可變參考。這就是「寫入時才 clone」的核心。
- **.into_owned()**：不管是 Borrowed 還是 Owned，都轉成擁有所有權的值。Borrowed 會 clone 一份，Owned 則直接拿走。

5.33.3 範例程式碼

```
use std::borrow::Cow;

// 如果字串已經是「你好」開頭，直接借用回傳
// 否則建立新的 String
fn ensure_greeting(s: &str) -> Cow<'_, str> {
    if s.starts_with("你好") {
        // 不需要修改，直接借用
        Cow::Borrowed(s)
    } else {
        // 需要修改，建立新字串
        let mut greeting = String::from("你好，");
        greeting.push_str(s);
        Cow::Owned(greeting)
    }
}

fn main() {
    // 已經有「你好」開頭 → 借用，不花成本
    let s1 = "你好世界";
    let result1 = ensure_greeting(s1);
    println!("{}", result1);

    // 沒有「你好」開頭 → 建立新字串
    let s2 = "Rust";
```

```
    let result2 = ensure_greeting(s2);
    println!("{}", result2);

    // 可以判斷是借用還是擁有
    match ensure_greeting(s1) {
        Cow::Borrowed(s) => println!("借用的：{}", s),
        Cow::Owned(s) => println!("擁有的：{}", s),
    }

    match ensure_greeting(s2) {
        Cow::Borrowed(s) => println!("借用的：{}", s),
        Cow::Owned(s) => println!("擁有的：{}", s),
    }

    // to_mut：寫入時才 clone
    let mut cow: Cow<'_, str> = Cow::Borrowed("hello");
    // 現在是 Borrowed，呼叫 to_mut 會先 clone 成 Owned
    cow.to_mut().push_str(" world");
    println!("{}", cow); // "hello world"

    // into_owned：轉成擁有的 String
    let cow2: Cow<'_, str> = Cow::Borrowed("bye");
    let owned: String = cow2.into_owned();
    println!("{}", owned);
}
```

5.33.4 重點整理

- `Cow<'a, str>` 可以是借用（`&str`）或擁有（`String`），視情況而定。
- `Cow` 利用 `ToOwned` trait 的 associated type 來決定擁有版本的型別（`str` → `String`、`[T]` → `Vec<T>`）。
- `Cow` 實作了 `Deref`，不管是 `Borrowed` 還是 `Owned`，`&Cow<'_, str>` 都能直接當 `&str` 用——這是它最大的優點。
- `.to_mut()`：寫入時才 clone（`Borrowed` → clone 成 `Owned` → 回傳可變參考）。
- `.into_owned()`：不管哪種都轉成擁有所有權的值。
- 適合用在「大部分時候不修改，偶爾需要修改」的場景。

恭喜你完成了第 5 章！🎉 這一章的內容非常紮實——從泛型、trait bound、生命週期，到 `Box`、`Rc` 等智慧指標與 `Deref` 機制，再到 `Cell`、`RefCell` 的 interior mutability，以及 `Display`、associated type、`Cow`。這些是 Rust 型別系統最強大的武器，也是讀懂標準庫原始碼的基礎。下一章我們將進入閉包與迭代器——Rust 最優雅的函數式程式設計風格！

第 6 章

閉包與迭代器

這一章會教到閉包與迭代器，這些功能結合使用了前面提到的許多語言功能，是比較進階難懂的主題，因此被放在第 5 章之後。儘管如此，他們在現實的 Rust 程式碼中仍然非常常見，因此如果想要讀懂真正的 Rust 程式碼，理解閉包與迭代器是必須的。

6.1 函數指標

6.1.1 本集目標

認識函數指標（function pointer）型別，學會把函數名稱當成值來傳遞和儲存。

6.1.2 概念說明

在 Rust 裡，函數不只能被呼叫——還能像值一樣被傳來傳去、存進變數、放進 `Vec`。要做到這件事，我們需要認識**函數指標**（function pointer）型別。

6.1.2.1 函數指標的寫法

假設你有一個函數：

```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

這個函數的函數指標型別是 `fn(i32) -> i32`。注意這裡的 `fn` 是小寫的——它代表函數指標型別，不是定義函數的關鍵字。

6.1.2.2 把函數存進變數

你可以把函數名稱直接賦值給一個變數：

```
    let f: fn(i32) -> i32 = add_one;
```

之後就能用 `f(10)` 來呼叫它，效果跟直接呼叫 `add_one(10)` 一樣。

6.1.2.3 把函數當參數傳遞

函數指標最常用的場景之一，就是「把一個函數傳給另一個函數」：

```
fn apply(f: fn(i32) -> i32, value: i32) -> i32 {
    f(value)
}
```

這讓 `apply` 可以接受任何簽名為 `fn(i32) -> i32` 的函數，非常靈活。

6.1.2.4　多個參數和不同回傳型別

函數指標的型別由參數和回傳值決定：

- 沒有參數、沒有回傳值：`fn()`。
- 兩個參數：`fn(i32, i32) -> i32`。
- 回傳 `String`：`fn(&str) -> String`。

6.1.2.5　函數指標 vs 下一集的閉包

函數指標 `fn(...) -> ...` 是一個具體的型別，大小固定。但函數沒辦法直接用其他函數裡的區域變數——想用，就得靠參數傳進去。下一集要介紹的閉包（closure）不一樣：閉包**寫在哪裡，旁邊有什麼區域變數就能直接拿來用**，不必一個個當參數傳。

6.1.3　範例程式碼

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn double(x: i32) -> i32 {
    x * 2
}

fn apply(f: fn(i32) -> i32, value: i32) -> i32 {
    f(value)
}

fn pick_function(use_double: bool) -> fn(i32) -> i32 {
    if use_double {
        double
    } else {
        add_one
    }
}

fn main() {
    // 把函數存進變數
    let f: fn(i32) -> i32 = add_one;
    println!("f(5) = {}", f(5));

    // 把函數當參數傳遞
    println!("apply(add_one, 10) = {}", apply(add_one, 10));
    println!("apply(double, 10) = {}", apply(double, 10));

    // 函數也可以當回傳值
    let chosen = pick_function(true);
    println!("chosen(7) = {}", chosen(7));

    let chosen2 = pick_function(false);
    println!("chosen2(7) = {}", chosen2(7));

    // 把函數放進 Vec 裡
    let operations: Vec<fn(i32) -> i32> = vec![add_one, double];
    for op in &operations {
```

```
        println!("op(3) = {}", op(3));
    }
}
```

6.1.4 重點整理

- 函數指標型別寫作 `fn(參數型別) -> 回傳型別`，注意是小寫 `fn`。
- 函數名稱可以直接當成值，賦值給變數或傳遞給其他函數，也可以存進 `Vec` 等容器。
- 函數沒辦法直接用其他函數裡的區域變數，要用只能靠參數傳進去；下一集的閉包則可以直接用它所在位置旁邊的區域變數。

6.2 閉包用法展示

6.2.1 本集目標

學會閉包的基本語法，了解閉包如何捕捉外部變數，並看到標準庫中使用閉包的實際案例。

6.2.2 概念說明

6.2.2.1 閉包的語法

上一集的函數指標很好用，但函數沒辦法直接用其他函數裡的區域變數，要用什麼都得靠參數傳進去。閉包（closure）不一樣：它寫在哪裡，就能直接用那裡的區域變數——這正是它存在的理由。

閉包的基本語法用 `|` 來包參數：

```
    let add_one = |x| x + 1;
```

你也可以加上型別標註，跟函數一樣明確：

```
    let add_one = |x: i32| -> i32 { x + 1 };
```

呼叫閉包的方式和呼叫一般函數一樣，直接用 `add_one(5)` 就好了——不需要任何特殊語法。

6.2.2.2 什麼時候要加大括號？

規則很簡單：

- **只有一個表達式**的時候，可以省略大括號：`|x| x + 1`。
- **有多行程式碼**或**需要 `let` 之類語句**的時候，要用大括號包起來：

```
    let process = |x: i32| {
        let doubled = x * 2;
        println!("計算中：{}", doubled);
        doubled + 1
    };
```

跟函數一樣，大括號裡最後一行不加分號就是回傳值。

另外，如果有加型別標註（`-> i32`），就一定要加大括號：

```
    let add_one = |x: i32| -> i32 { x + 1 }; // 有 -> 就必須有 {}
    let add_one = |x: i32| x + 1;            // 沒有 -> 可以省略 {}
```

6.2.2.3 閉包能捕捉外部變數

這是閉包和函數指標最大的差別：

```
fn main() {
    let offset = 10;
    let add_offset = |x| x + offset; // 捕捉了 offset
    println!("{}", add_offset(5));   // 15
}
```

`add_offset` 這個閉包「記住」了外部的 `offset`，每次呼叫都會用到它。普通函數做不到這件事。

6.2.2.4 閉包不是只有一種

根據閉包**怎麼使用**捕捉到的變數，Rust 會把閉包分成不同的種類——有些閉包只能呼叫一次，有些可以呼叫很多次。這一集先看兩個例子感受一下差別，下幾集再深入解釋。

6.2.2.5 `Result` 的 `map` —— `FnOnce` 的例子

標準庫很多方法都接受閉包。還記得第 5 章的 `Result<T, E>` 嗎？它有一個 `map` 方法，可以把 `Ok` 裡的值做轉換。`map` 只需要呼叫閉包一次，所以它接受 `FnOnce`——「至少能呼叫一次」就夠了。

這意味著你可以傳一個**會消耗捕捉到的變數**的閉包給它：

```
fn main() {
    let prefix = String::from("結果是：");
    let result: Result<i32, String> = Ok(42);
    let message = result.map(|x| {
        // prefix 被 move 進來，這個閉包只能呼叫一次
        let mut s = prefix; // move！
        s.push_str(&x.to_string());
        s
    });
    println!("{:?}", message); // Ok("結果是：42")
}
```

這個閉包把 `prefix` move 進來了，呼叫一次之後 `prefix` 就沒了。但沒關係，`map` 本來就只呼叫接收的函數一次。

6.2.2.6 `Vec` 的 `retain` —— `FnMut` 的例子

`Vec<T>` 的 `retain` 方法會保留符合條件的元素，移除不符合的。它接受一個閉包，這個閉包接收 `&T`（每個元素的參考）、回傳 `bool`（`true` 保留、`false` 移除）。因為 `retain` 要對每個元素都呼叫一次，所以它要求 `FnMut`——「可以多次呼叫」。

你可以傳一個**會修改捕捉到的變數**的閉包：

```
fn main() {
    let mut numbers = vec![1, 2, 3, 4, 5, 6];
    let mut removed_count = 0;
    numbers.retain(|x| {
        if x % 2 == 0 {
            true // 保留偶數
        } else {
            removed_count += 1; // 修改外部變數
            false
        }
    });
```

```
    println!("{:?}，移除了 {} 個", numbers, removed_count);
    // [2, 4, 6]，移除了 3 個
}
```

這個閉包每次被呼叫都會修改 `removed_count`——它是 `FnMut`。注意它沒有 move 任何東西（只是透過 `&mut` 修改外部變數），所以可以被呼叫很多次。

6.2.2.7 如果把 FnOnce 傳給 retain？

上面傳給 `Result` 的 `map` 那種會 move 變數的閉包能傳給 `retain` 嗎？

```
    let mut items = vec![1, 2, 3];
    let header = String::from("剔除：");
    items.retain(|x| {
        if *x <= 1 {
            let mut log = header; // move header
            log.push_str(&x.to_string());
            log.push(' ');
        }
        *x > 1
    }); // 編譯錯誤！
```

這個閉包在第一次剔除元素時就把 `header` move 走了，第二次要剔除時 `header` 已經不存在。它只能呼叫一次（`FnOnce`），但 `retain` 需要多次呼叫（`FnMut`）。所以編譯器會報錯。

6.2.2.8 不捕捉變數的閉包 → 可以轉成函數指標

如果一個閉包沒有捕捉任何外部變數，它就跟普通函數沒什麼差別。Rust 允許它自動轉型成函數指標 `fn`：

```
    let add_one: fn(i32) -> i32 = |x| x + 1; // 沒有捕捉，可以轉成 fn
```

但如果捕捉了外部變數，就不能這樣轉了。

6.2.3 範例程式碼

```
fn apply_fn_pointer(f: fn(i32) -> i32, value: i32) -> i32 {
    f(value)
}

fn main() {
    // 基本閉包語法
    let square = |x: i32| -> i32 { x * x };
    println!("square(4) = {}", square(4));

    // 捕捉外部變數
    let base = 100;
    let add_base = |x| x + base;
    println!("add_base(7) = {}", add_base(7));

    // Result 的 map（FnOnce）
    let result: Result<i32, String> = Ok(21);
    let doubled = result.map(|x| x * 2);
    println!("doubled = {:?}", doubled);
```

```
    let err_result: Result<i32, String> = Err(String::from("oops"));
    let still_err = err_result.map(|x| x * 2);
    println!("still_err = {:?}", still_err);

    // Vec 的 retain（FnMut）
    let mut scores = vec![55, 72, 88, 43, 91, 60];
    scores.retain(|s| *s >= 60);
    println!("及格分數：{:?}", scores);

    // 不捕捉變數的閉包可以轉成函數指標
    let triple: fn(i32) -> i32 = |x| x * 3;
    println!("apply_fn_pointer(triple, 5) = {}", apply_fn_pointer(triple, 5));

    // 捕捉了變數的閉包不能轉成函數指標
    // let offset = 10;
    // let bad: fn(i32) -> i32 = |x| x + offset; // 編譯錯誤！
}
```

6.2.4 重點整理

- 閉包用 |參數| 表達式 語法，可以省略型別標註讓 Rust 推導。
- 閉包最大的特色是能**捕捉外部變數**，這是函數指標做不到的。
- `Result` 的 `map` 接受 `FnOnce` 閉包——只需呼叫一次。
- `Vec` 的 `retain` 接受 `FnMut` 閉包——需要多次呼叫。
- 如果閉包只能呼叫一次（`FnOnce`），就不能傳給需要多次呼叫的方法。
- 不捕捉外部變數的閉包可以自動轉型成函數指標 `fn`。

6.3 手動實作閉包

6.3.1 本集目標

這一集會用 `struct` + 方法來手動模擬閉包，理解編譯器在背後做了什麼事。你會看到三種閉包各自的一個 `struct` 範例，以及為什麼呼叫閉包其實是在呼叫方法。

6.3.2 概念說明

6.3.2.1 閉包 = 匿名 `struct` + 方法

上一集我們看到閉包可以捕捉外部變數。但它是怎麼「記住」這些變數的？

答案很直接——編譯器幫你做了兩件事：

1. **建立一個匿名 `struct`**，把捕捉的變數存成欄位。
2. **在那個 `struct` 上 `impl` 一個方法**，方法的內容就是你寫在 `||` 後面的閉包體。

換句話說，你寫的閉包體（`{ ... }` 裡面的程式碼）就是那個方法的實作。

今天我們就來手動做一次編譯器做的事，把三種閉包分別模擬出來。

6.3.2.2 閉包呼叫 = 方法呼叫

當你寫 `f()` 呼叫一個閉包，編譯器其實把它轉換成 `struct` 上的方法呼叫：

- **`FnOnce`**：`f()` → `f.call_once()` — 傳 `self`，消耗整個 `struct`。

- **FnMut**：`f()` → `f.call_mut()` — 傳 `&mut self`，也就是 struct 的可變參考。
- **Fn**：`f()` → `f.call()` — 傳 `&self`，也就是 struct 的共享參考。

看出來了嗎？這就是第 4 章學過的三種 `self` 參數寫法：`self`、`&mut self`、`&self`。閉包的三種分類，本質上就是方法接收 `self` 的三種方式。

上一集介紹了 FnOnce（消耗捕捉的值，只能呼叫一次）和 FnMut（修改捕捉的值，可多次呼叫）。**Fn 在上一集沒有出現**——它是第三種：呼叫時不會消耗閉包，也不需要閉包值的可變參考，所以可以呼叫任意多次。

接下來我們分別用 struct 手動模擬這三種閉包。注意：**下面三個範例使用了不同的欄位型別**。這些是範例所使用的欄位型別，不是三種閉包的固定要求。

6.3.2.3 FnOnce 範例：struct 存擁有的值，方法接 self

假設我們有這樣的閉包：

```
fn main () {
    let name = String::from("Alice");
    let greet = || {
        let s = name; // 閉包體內把 name 移走了
        println!("Hello, {}!", s);
    };
    greet();
    // greet(); // 編譯錯誤！name 已經被移走，不能再呼叫
}
```

編譯器會產生類似這樣的東西：

```
struct GreetOnce {
    name: String, // 擁有 name
}

// 建立閉包 = 把捕捉的變數塞進 struct
// let greet = GreetOnce { name };

impl GreetOnce {
    // 呼叫閉包 = 呼叫 struct 上的方法
    fn call_once(self) {
        let s = self.name; // 把 name 從 struct 裡移出來
        println!("Hello, {}!", s);
    }
}
```

因為方法接收 `self`，呼叫的時候整個 struct 被消耗掉了，所以只能呼叫一次。這就是 FnOnce。

6.3.2.4 FnMut 範例：struct 存可變參考，方法接 &mut self

假設閉包修改了捕捉的變數：

```
fn main() {
    let mut name = String::from("Alice");
    let mut greet = || {
        name.push_str("!");
        println!("Hello, {}", name);
    };
    greet();
```

```
    greet(); // 可以多次呼叫
}
```

編譯器產生的東西：

```
struct GreetMut<'a> {
    name: &'a mut String, // name 的可變參考
}

// let mut greet = GreetMut { name: &mut name };

impl<'a> GreetMut<'a> {
    fn call_mut(&mut self) {
        self.name.push_str("!");
        println!("Hello, {}", self.name);
    }
}
```

為什麼 struct 存 &mut，方法又接 &mut self？因為一個閉包可能捕捉多個變數。假設閉包同時修改了 a、b、c 三個變數，struct 裡就會有三個欄位：

```
struct SomeClosure<'a> {
    a: &'a mut i32,
    b: &'a mut String,
    c: &'a mut Vec<i32>,
}
```

方法用 &mut self 而不是 self，因為用 self 的話呼叫一次就消耗掉了——那就變成 FnOnce 了。FnMut 需要多次呼叫，所以只能取得整個 struct 的可變參考。

6.3.2.5 Fn 範例：struct 存共享參考，方法接 &self

如果閉包只是讀取捕捉的變數，完全不修改：

```
fn main() {
    let name = String::from("Alice");
    let greet = || {
        println!("Hello, {}!", name);
    };
    greet();
    greet(); // 可以多次呼叫，完全沒問題
}
```

編譯器產生的東西：

```
struct GreetRef<'a> {
    name: &'a String, // name 的共享參考
}

// let greet = GreetRef { name: &name };

impl<'a> GreetRef<'a> {
    fn call(&self) {
        println!("Hello, {}!", self.name);
    }
}
```

因為方法接收 &self，struct 不會被消耗也不會被修改，所以可以呼叫任意多次。這就是 Fn。

6.3.2.6 對照表

self 類型	對應類型	這個範例的欄位存什麼	能做什麼
self	FnOnce	擁有的值（如 String）	消耗捕捉的值，只能呼叫一次
&mut self	FnMut	可變參考（如 &mut String）	修改捕捉的值，可以多次呼叫
&self	Fn	共享參考（如 &String）	只讀取，可以多次呼叫

6.3.2.7 小結：閉包到底是什麼？

把上面的東西串起來：

1. 編譯器幫你建一個匿名 struct，把捕捉的變數存進去。
2. 你寫的閉包體就是那個 struct 上方法的實作。
3. 當你寫 f() 的時候，編譯器根據閉包的種類，呼叫 struct 上的 .call_once() / .call_mut() / .call()。

每次你寫一個閉包，編譯器就在幕後做了「建 struct → impl 方法 → 呼叫方法」這些事。

理解了「閉包體其實只是一個方法的內容」之後，還可以順便想想一件事：在閉包裡寫 return 會發生什麼事？因為閉包體事實上就是某個方法的實作內容，return 跳出的是那個方法，也就是**只會跳出最內層的閉包**，而不會跳出包在外面的函數。這其實和 break 的預設效果很像——break 預設也只會跳出最內層的迴圈，而不是一口氣跳出所有巢狀迴圈。

6.3.3 範例程式碼

以下的完整程式碼整理了上面的三個範例。每個 struct 都使用該範例的欄位型別和 self 參數寫法來模擬一種閉包：

```
// === FnOnce 模擬 ===
// struct 擁有值，方法接 self
struct GreetOnce {
    name: String,
}

impl GreetOnce {
    fn call_once(self) {
        // 閉包體：把 name 移走
        let s = self.name;
        println!("[FnOnce] Hello, {}!", s);
        // self 被消耗了，不能再用
    }
}

// === FnMut 模擬 ===
// struct 存可變參考，方法接 &mut self
struct GreetMut<'a> {
    name: &'a mut String,
}

impl<'a> GreetMut<'a> {
    fn call_mut(&mut self) {
```

```
        // 閉包體：修改捕捉的變數
        self.name.push_str("!");
        println!("[FnMut] Hello, {}", self.name);
    }
}

// === Fn 模擬 ===
// struct 存共享參考，方法接 &self
struct GreetRef<'a> {
    name: &'a String,
}

impl<'a> GreetRef<'a> {
    fn call(&self) {
        // 閉包體：只讀取，不修改
        println!("[Fn] Hello, {}!", self.name);
    }
}

fn main() {
    // --- FnOnce：呼叫一次就消耗 ---
    let name1 = String::from("Alice");
    let greet_once = GreetOnce { name: name1 };
    greet_once.call_once();
    // greet_once.call_once(); // 編譯錯誤！struct 已經被消耗了

    // --- FnMut：可以多次呼叫，每次修改 ---
    let mut name2 = String::from("Bob");
    {
        let mut greet_mut = GreetMut { name: &mut name2 };
        greet_mut.call_mut(); // Bob!
        greet_mut.call_mut(); // Bob!!
        greet_mut.call_mut(); // Bob!!!
    } // greet_mut 離開作用域，它的可變參考不再被使用
    println!("name2 現在是：{}", name2);

    // --- Fn：只讀取，呼叫幾次都行 ---
    let name3 = String::from("Charlie");
    let greet_ref = GreetRef { name: &name3 };
    greet_ref.call();
    greet_ref.call();
    greet_ref.call();
}
```

6.3.4 重點整理

- 閉包背後就是一個匿名 `struct`，捕捉的變數變成 `struct` 的欄位。
- **三種閉包的差別在方法怎麼接收 `self`**：`self`（FnOnce）、`&mut self`（FnMut）、`&self`（Fn）。
- 閉包體就是 `struct` 上方法的實作內容。
- `f()` 會被編譯器轉換成方法呼叫：`f.call_once()` / `f.call_mut()` / `f.call()`。
- Fn：呼叫時只需要閉包值的共享參考，所以可以重複呼叫。
- 因為閉包體只是一個方法的內容，閉包裡的 `return` 只會跳出最內層的閉包，不會跳出外面的函數——就像 `break` 預設只跳出最內層的迴圈。

- 下一集會講編譯器是怎麼**自動判斷**一個閉包該歸類為 FnOnce、FnMut 還是 Fn。

6.4 閉包種類的推斷

6.4.1 本集目標

理解 Rust 如何根據閉包體的內容，自動推斷一個閉包是 FnOnce、FnMut 還是 Fn。

6.4.2 正文

上一集我們手動用 struct 模擬了三種閉包，對應 self、&mut self、&self。但你寫閉包的時候從來不需要告訴 Rust「這是 FnOnce」或「這是 FnMut」——Rust 會自動判斷。

6.4.2.1 推斷規則

Rust 看的是**閉包體裡面對捕捉變數做了什麼**：

1. **如果閉包體裡 move 了捕捉的變數**（例如 `let s = captured_string;`）→ 這個閉包是 **FnOnce**，因為 move 走了就沒了，只能呼叫一次。
2. **如果閉包體需要以可變方式存取捕捉狀態**（例如 `count += 1;`）→ 這個閉包是 **FnMut**，可以多次呼叫但需要 &mut。
3. **如果閉包體只需要以共享方式存取捕捉狀態**（例如 `println!("{}", name);`）→ 這個閉包是 **Fn**，只需要 &self。

Rust 會選**能接受最多種使用方式的那個**——如果只需要共享存取，就給 Fn（因為 Fn 的閉包也能當 FnMut 和 FnOnce 用）。如果需要可變存取，就變成 FnMut。如果有 move，就變成 FnOnce。

6.4.2.2 範例對照

```
let name = String::from("Alice");

// 只讀取 name → Fn
let greet = || println!("Hi, {}!", name);

// 修改 count → FnMut
let mut count = 0;
let mut increment = || { count += 1; };

// move name → FnOnce
let consume = || { let s = name; };
```

你不需要寫任何標記——Rust 看閉包體就知道了。

6.4.2.3 捕捉多個變數時怎麼辦？

一個閉包可能同時捕捉多個變數，而且對每個變數的用法不同：

```
let name = String::from("Alice");
let mut count = 0;
let closure = || {
    count += 1;           // 修改 count → 需要 &mut
    println!("{}", name); // 只讀取 name → 只需要 &
};
```

想像成 struct 的話，這個閉包的匿名 struct 會有兩個欄位：count（需要 &mut）和 name（只需要

&)。但呼叫閉包時只有一個 self——而 &mut self 裡面可以做 & 的操作，反過來不行——所以整個閉包是 FnMut（&mut self）。就像一個 method 接收 &mut self，但裡面不一定每個欄位都要改：

```
struct Data<'a> {
    count: &'a mut i32,
    name: &'a String,
}

impl<'a> Data<'a> {
    fn increment_and_greet(&mut self) {
        *self.count += 1;                  // 修改 count
        println!("Hello, {}!", self.name); // 只讀取 name
    }
}
```

閉包也是同樣的道理。

同理，FnOnce 的 self 裡面的值當然也能取 & 或 &mut——擁有一個值就包含了可以借用它。

6.4.2.4 如果沒有捕捉任何變數呢？

沒有捕捉變數的閉包自動是 Fn，因為它不需要存取任何外部狀態：

```
    let add_one = |x: i32| x + 1; // Fn
```

第 2 集提到的「不捕捉變數的閉包可以轉成函數指標」也是因為這個原因——它連匿名 struct 都不需要。

6.4.3 重點整理

- Rust 根據閉包體的內容自動推斷閉包的種類：move → FnOnce、可變存取 → FnMut、共享存取 → Fn。
- 不需要手動標記，編譯器會自動選擇能接受最多種使用方式的閉包。
- 沒有捕捉變數的閉包是 Fn，也可以轉成函數指標。
- Fn 的閉包可以傳給 FnMut 和 FnOnce；FnMut 可以傳給 FnOnce；反過來不行。

6.5 FnOnce / FnMut / Fn

6.5.1 本集目標

理解 FnOnce、FnMut、Fn 是 trait 而非型別，掌握它們的繼承關係，並學會選擇正確的閉包 trait。

6.5.2 概念說明

6.5.2.1 它們是 trait，不是型別

前幾集我們一直說 FnOnce、FnMut、Fn，但還沒正式說明——它們其實是 **trait**。就像前面學的 Clone、Display 一樣，FnOnce / FnMut / Fn 是定義在標準庫裡的 trait。每個閉包的匿名 struct 會自動 impl 對應的 trait（上一集講的推斷規則決定 impl 哪些）。

那這些 trait 到底長什麼樣？

- FnOnce(Args) -> Ret：可以被呼叫一次（這次呼叫可能會消耗閉包，因此不保證之後還能再

次呼叫）。

- `FnMut(Args) -> Ret`：可以被重複呼叫，並以可變方式存取捕捉的狀態。
- `Fn(Args) -> Ret`：可以被重複呼叫，但不會以可變方式存取捕捉的狀態。

注意！`fn(i32) -> i32`（小寫）是函數指標**型別**，而 `Fn(i32) -> i32`（大寫）是 **trait**。兩個完全不同的東西。

6.5.2.2 繼承關係

這三個 trait 有繼承（supertrait）關係：

```
Fn : FnMut : FnOnce
```

意思是：

- 所有實作 Fn 的東西，自動也實作 FnMut 和 FnOnce。
- 所有實作 FnMut 的東西，自動也實作 FnOnce。
- 但 FnOnce 不一定有 FnMut，FnMut 不一定有 Fn。

為什麼是這個方向？

- **Fn → FnMut**：如果一個閉包只需要 `&self` 就能執行，那改用 `&mut self` 當然也行（只是用可變參考代替原本就足夠的共享參考）。
- **FnMut → FnOnce**：如果一個閉包用 `&mut self` 就能執行，那給它 `self`（整個所有權）當然也行——擁有一個東西就包含了可以修改它。只是呼叫完之後 struct 被消耗了，不能再呼叫第二次。

反過來就不行——一個需要消耗自己（FnOnce）的閉包，不能保證多次呼叫（FnMut）。

6.5.2.3 用 `impl Trait` 接受閉包

還記得第 5 章的 `impl Trait` 嗎？用它來接受閉包參數：

```
fn call_once(f: impl FnOnce() -> String) -> String {
    f()
}

fn call_many_times(mut f: impl FnMut()) {
    f();
    f();
    f();
}

fn call_twice(f: impl Fn() -> i32) -> i32 {
    f() + f()
}
```

注意 FnMut 的參數要加 `mut`——因為呼叫 FnMut 閉包需要 `&mut self`，所以 `f` 本身要是 `mut` 的。

6.5.2.4 程式設計原則：選能接受最多種閉包的 bound

當你設計一個接受閉包的函數時，應該選**能接受最多種閉包**的 trait bound：

1. 先試 FnOnce —— 如果你只需要呼叫一次。
2. 不夠再用 FnMut —— 如果你需要多次呼叫。
3. 最後才用 Fn —— 如果你需要多次呼叫，而且不能取得閉包值的可變參考。

為什麼？因為 FnOnce 能接受所有閉包（所有閉包都會實作 FnOnce），而 Fn 只能接受可以透過共享參考呼叫的閉包。選能接受最多種的 bound，使用者傳入的自由度最高。

實務上 Fn 很少用到——大部分需要多次呼叫閉包的函數用 FnMut 就夠了（FnMut 也能接受 Fn 的閉包）。當函數需要在沒有閉包值可變參考的情況下呼叫它時，才使用 Fn。

6.5.2.5 函數指標也實作了這三個 trait

普通的函數（和函數指標 fn）自動實作了 Fn、FnMut、FnOnce。所以你可以把函數名稱傳給任何接受這三個 trait 的地方。

6.5.3 範例程式碼

```
// 只需要呼叫一次 → 用 FnOnce（能接受最多種閉包）
fn consume_and_print(f: impl FnOnce() -> String) {
    let result = f();
    println!("結果：{}", result);
}

// 需要多次呼叫 → 用 FnMut
fn repeat_three_times(mut f: impl FnMut()) {
    f();
    f();
    f();
}

// 需要多次呼叫，而且不能取得閉包值的可變參考 → 用 Fn
fn sum_two_calls(f: impl Fn(i32) -> i32, x: i32) -> i32 {
    f(x) + f(x)
}

fn main() {
    // FnOnce：閉包消耗了捕捉的值
    let name = String::from("Rust");
    consume_and_print(|| {
        let s = name; // move name
        format!("Hello, {}!", s)
    });

    // FnMut：閉包修改了捕捉的變數
    let mut count = 0;
    repeat_three_times(|| {
        count += 1;
        println!("第 {} 次呼叫", count);
    });
    println!("總共呼叫了 {} 次", count);

    // Fn：閉包只讀取
    let multiplier = 3;
    let result = sum_two_calls(|x| x * multiplier, 5);
    println!("sum_two_calls 結果：{}", result);

    // 普通函數也能傳進去
    fn double(x: i32) -> i32 {
        x * 2
```

```
    }
    let result2 = sum_two_calls(double, 10);
    println!("用普通函數：{}", result2);

    // Fn 的閉包也可以傳給 FnOnce 的參數（所有 Fn 閉包都實作 FnOnce）
    let greeting = String::from("哈囉");
    consume_and_print(|| {
        format!("{}, 世界！", greeting) // 只是讀取 greeting，是 Fn
    });
    // greeting 還活著，因為閉包只是借用了它
    println!("greeting 還在：{}", greeting);
}
```

6.5.4 重點整理

- FnOnce、FnMut、Fn 是 **trait**，不是型別；fn 才是函數指標型別。
- 繼承關係：Fn ⊂ FnMut ⊂ FnOnce（FnOnce 能接受所有閉包）。
- 用 `impl FnOnce()` / `impl FnMut()` / `impl Fn()` 來接受閉包參數。
- FnMut 的參數要加 `mut`。
- 函數接收閉包的設計原則：**先選 FnOnce**，需要多次呼叫再改 FnMut，呼叫時不能取得閉包值的可變參考才用 Fn。
- 函數指標自動實作了 Fn + FnMut + FnOnce。

6.6 move 閉包

6.6.1 本集目標

學會用 move 關鍵字強制閉包以值捕捉外部變數，理解它在什麼情況下能解決生命週期問題。

6.6.2 概念說明

6.6.2.1 預設的捕捉行為

Rust 的閉包很聰明，會自動選擇「最輕量」的捕捉方式：

- 如果只讀取變數 → 用 &T（借用）。
- 如果需要修改 → 用 &mut T（可變借用）。
- 如果需要消耗 → 用 T（move）。

大部分時候這很好用。但有些情況下，借用會造成生命週期的問題。

6.6.2.2 問題場景：回傳閉包

假設你想寫一個函數，回傳一個閉包：

```
fn make_greeter(name: String) -> impl Fn() {
    || println!("Hello, {}!", name) // 編譯錯誤！
}
```

這段程式會編譯失敗，因為閉包預設用借用的方式捕捉 name（&name），但 name 是函數的局部變數，函數結束後就被丟掉了。閉包裡的借用就變成了懸垂參考——第 4 章的老朋友。

6.6.2.3 move 關鍵字

加上 move 就解決了：

```
fn make_greeter(name: String) -> impl Fn() {
    move || println!("Hello, {}!", name)
}

fn main() {}
```

move 告訴 Rust：把所有用到的外部變數都**以值捕捉**進閉包裡。在這裡，閉包捕捉的是 String 本身，所以 name 歸閉包所有；不管原本的作用域怎麼結束，閉包都能繼續用自己的 name。

6.6.2.4 move 閉包的匿名 struct

回想前幾集——閉包是匿名 struct。沒有 move 的時候，struct 的欄位可能是外部變數的參考（&T 或 &mut T）；加了 move 之後，閉包會**以值捕捉**這些變數：

```
    // 沒有 move：閉包借用 name，struct 裡存的是參考
    let name = String::from("Alice");
    let greet = || println!("{}", name);
    // name 還能用，因為閉包只是借用

    // 有 move：name 被搬進 struct，閉包擁有它
    let name = String::from("Alice");
    let greet = move || println!("{}", name);
    // name 不能再用了，已經被搬進閉包裡
```

在這個範例中，被捕捉的變數是 String，所以閉包擁有這個字串，不再借用區域變數 name。因此它可以安全地從函數回傳。

不過，以值捕捉參考，不會讓閉包一併取得該參考所指向資料的所有權。如果被捕捉的變數本身就是參考，閉包存的仍然是原本的參考：

```
fn make_printer<'a>(text: &'a str) -> impl Fn() + 'a {
    // text 本身是 &'a str；move 以值捕捉的就是這個參考
    move || println!("{}", text)
}

fn main() {
    let message = String::from("hello");
    let print = make_printer(&message);
    print();
}
```

這裡的 text 是 &'a str。因為共享參考有 Copy，move 會把這個參考值複製進閉包，並不會讓閉包取得字串資料的所有權。用 struct 類比的話，閉包的欄位仍然是 text: &'a str。回傳型別上的 + 'a 明確表達了這個 lifetime 關係：當 text 指向區域字串時，回傳的閉包不能在該字串被丟棄後繼續使用。換句話說，move 不會自動讓閉包變成 'static。

6.6.2.5 move 不影響閉包是哪種 Fn trait

很多人會搞混：move 閉包不代表它是 FnOnce！

move 只影響**怎麼捕捉**，不影響**怎麼使用**：

```rust
fn main() {
    let name = String::from("Alice");
    let greet = move || println!("Hello, {}!", name);
    // name 被 move 進閉包了，但閉包只是「讀取」name
    // 所以這個閉包是 Fn，可以多次呼叫
    greet();
    greet();
}
```

6.6.2.6 閉包自動實作的 trait

前面幾集大多把閉包當成「可以呼叫的東西」來看，還沒有很適合的時機問另一個所有權問題：閉包這個值本身能不能被 move、copy，或 clone？

這集終於開始談閉包和所有權的關係，所以適合在這裡補上答案。閉包這個值本身可以像其他值一樣被 move；但它能不能 copy 或 clone，取決於它實際捕捉進來的值——跟 tuple 類似，如果閉包裡實際儲存的每個值都能 copy，整體就能 copy：

- 實際捕捉的值全都實作 Copy → 閉包也是 Copy。
- 實際捕捉的值全都實作 Clone → 閉包也是 Clone。
- 其他某些 trait 也是同理。

```rust
fn main() {
    let x = 42;
    let f = move || x + 1; // x 是 i32（Copy），所以 f 也是 Copy
    let g = f; // Copy 了 f
    println!("{}", f()); // f 還能用
    println!("{}", g());
}
```

6.6.3 範例程式碼

```rust
// 回傳閉包時，通常需要 move
fn make_adder(n: i32) -> impl Fn(i32) -> i32 {
    move |x| x + n
}

fn make_counter(start: i32) -> impl FnMut() -> i32 {
    let mut count = start;
    move || {
        count += 1;
        count
    }
}

fn main() {
    // move 讓閉包擁有捕捉的值，可以安全回傳
    let add_five = make_adder(5);
    println!("10 + 5 = {}", add_five(10));
    println!("20 + 5 = {}", add_five(20));

    // move + FnMut：閉包擁有 count，並且每次修改它
    let mut counter = make_counter(0);
    println!("計數：{}", counter());
```

```
    println!("計數：{}", counter());
    println!("計數：{}", counter());

    // move 不代表 FnOnce
    let name = String::from("Bob");
    let greet = move || {
        println!("Hi, {}!", name); // 只是讀取 name，所以是 Fn
    };
    greet();
    greet(); // 可以多次呼叫，不是 FnOnce

    // 捕捉 Copy 型別的閉包可以 Copy
    let factor = 3;
    let multiply = move |x: i32| x * factor;
    let multiply_copy = multiply; // Copy 了
    println!("multiply(4) = {}", multiply(4)); // 原本的還能用
    println!("multiply_copy(4) = {}", multiply_copy(4));

    // 捕捉 String（非 Copy）的 move 閉包不能 Copy
    let label = String::from("result");
    let show = move |x: i32| {
        println!("{}: {}", label, x);
    };
    // let show2 = show; // 這會 move show，不是 Copy
    show(42);
}
```

6.6.4 重點整理

- move 強制閉包以值捕捉所有用到的外部變數。如果被捕捉的變數本身是參考，它仍然是參考，也保留原本的 lifetime。
- 回傳閉包時通常需要 move，讓閉包以值捕捉函數內的區域變數；但以值捕捉的參考仍然必須活得夠久。
- move **不影響**閉包是 FnOnce / FnMut / Fn——那取決於閉包**怎麼使用**捕捉的值。
- 閉包能否 clone / copy，取決於它實際捕捉進來的值是否全為 Clone / Copy。

6.7 Option / Result 的閉包方法

6.7.1 本集目標

認識 Option 和 Result 上接受閉包的常用方法，體會閉包如何讓程式碼更簡潔流暢。

6.7.2 概念說明

第 5 章我們用 match 處理 Option 和 Result，每次都要展開兩個分支。現在學了閉包，很多操作可以一行搞定。

6.7.2.1 Option 的閉包方法

以下方法定義在 Option<T> 上，簽名中的 T 就是 Option<T> 的型別參數。

map —— 轉換 Some 裡的值

```
    // Option<T> 上的方法：
    // fn map<U>(self, f: impl FnOnce(T) -> U) -> Option<U>
    let x: Option<i32> = Some(5);
    let y = x.map(|v| v * 2); // Some(10)
```

如果是 None，map 什麼都不做，直接回傳 None。不用寫 match。

and_then —— 鏈式操作（可能失敗）

map 的閉包回傳普通值，但如果你的轉換本身也可能回傳 None 呢？用 and_then：

```
    // Option<T> 上的方法：
    // fn and_then<U>(self, f: impl FnOnce(T) -> Option<U>) -> Option<U>
    let x: Option<i32> = Some(5);
    let y = x.and_then(|v| if v > 3 { Some(v * 2) } else { None });
```

and_then 的閉包回傳 Option，避免了 Option<Option<T>> 的巢狀問題。其實 and_then 就等於先 map 再 flatten——map 會產生 Option<Option<U>>，flatten 再把它攤平成 Option<U>。and_then 一步到位。

unwrap_or_else —— 給一個計算預設值的閉包

```
fn main() {
    // Option<T> 上的方法：
    // fn unwrap_or_else(self, f: impl FnOnce() -> T) -> T
    let x: Option<i32> = None;
    let y = x.unwrap_or_else(|| {
        println!("沒有值，計算預設值...");
        42
    });
}
```

跟 unwrap_or 不同，unwrap_or_else 的預設值是**惰性計算**的——只有在真的是 None 的時候才會執行閉包。

filter —— 條件過濾

```
    // Option<T> 上的方法：
    // fn filter(self, predicate: impl FnOnce(&T) -> bool) -> Option<T>
    let x: Option<i32> = Some(4);
    let y = x.filter(|v| v % 2 == 0); // Some(4)，因為 4 是偶數
    let z = x.filter(|v| v % 2 != 0); // None，因為 4 不是奇數
```

6.7.2.2 Result 的閉包方法

Result 也有類似的一套方法。以下方法定義在 Result<T, E> 上，T 是 Ok 的型別，E 是 Err 的型別。

map —— 轉換 Ok 的值

```
    // Result<T, E> 上的方法：
    // fn map<U>(self, f: impl FnOnce(T) -> U) -> Result<U, E>
    let r: Result<i32, String> = Ok(10);
    let doubled = r.map(|v| v * 2); // Ok(20)
```

map_err —— 轉換 Err 的值

跟 map 相反——map 對 Ok 做事、Err 不動；map_err 對 Err 做事、Ok 不動。

```
    // Result<T, E> 上的方法：
    // fn map_err<F>(self, f: impl FnOnce(E) -> F) -> Result<T, F>
    let r: Result<i32, String> = Err(String::from("not found"));
    let r2 = r.map_err(|e| format!("錯誤：{}", e));
```

and_then —— 鏈式操作

```
    // Result<T, E> 上的方法：
    // fn and_then<U>(self, f: impl FnOnce(T) -> Result<U, E>) -> Result<U, E>
    let r: Result<i32, String> = Ok(5);
    let r2 = r.and_then(|v| {
        if v > 0 {
            Ok(v * 10)
        } else {
            Err(String::from("必須是正數"))
        }
    });
```

跟 Option 一樣，and_then 就等於 map 再 flatten。

unwrap_or_else —— 從 Err 計算預設值

```
fn main() {
    // Result<T, E> 上的方法：
    // fn unwrap_or_else(self, f: impl FnOnce(E) -> T) -> T
    let r: Result<i32, String> = Err(String::from("oops"));
    let value = r.unwrap_or_else(|e| {
        println!("發生錯誤：{}，使用預設值", e);
        0
    });
}
```

6.7.2.3 跟 match 的比較

用 match：

```
    let result = match opt {
        Some(v) => Some(v * 2),
        None => None,
    };
```

用閉包方法：

```
    let result = opt.map(|v| v * 2);
```

一行搞定，而且意圖更清晰——「對 Some 裡的值做轉換」。

6.7.3 範例程式碼

```
fn parse_and_double(input: &str) -> Result<i32, String> {
    input
        .parse::<i32>()
        .map_err(|e| format!("解析失敗：{}", e))
        .and_then(|n| {
            if n >= 0 {
```

```
                Ok(n * 2)
            } else {
                Err(String::from("不接受負數"))
            }
        })
}

fn find_even(numbers: &[i32]) -> Option<i32> {
    for n in numbers {
        if n % 2 == 0 {
            return Some(*n);
        }
    }
    None
}

fn main() {
    // Option 的 map
    let maybe_num: Option<i32> = Some(21);
    let doubled = maybe_num.map(|n| n * 2);
    println!("map: {:?}", doubled);

    // Option 的 and_then
    let result = maybe_num.and_then(|n| {
        if n > 10 { Some(n - 10) } else { None }
    });
    println!("and_then: {:?}", result);

    // Option 的 filter
    let even = maybe_num.filter(|n| n % 2 == 0);
    println!("filter(偶數): {:?}", even);

    // Option 的 unwrap_or_else
    let none_value: Option<i32> = None;
    let default = none_value.unwrap_or_else(|| {
        println!("計算預設值中...");
        99
    });
    println!("unwrap_or_else: {}", default);

    // Result 鏈式操作
    println!("\n--- Result 鏈式操作 ---");
    let good = parse_and_double("21");
    println!("parse_and_double(\"21\") = {:?}", good);

    let bad_parse = parse_and_double("abc");
    println!("parse_and_double(\"abc\") = {:?}", bad_parse);

    let negative = parse_and_double("-5");
    println!("parse_and_double(\"-5\") = {:?}", negative);

    // Result 的 unwrap_or_else
    let safe_value = parse_and_double("oops").unwrap_or_else(|e| {
        println!("錯誤處理：{}", e);
        0
```

```
    });
    println!("安全取值：{}", safe_value);

    // 組合 Option 方法
    println!("\n--- Option 鏈式操作 ---");
    let numbers = vec![1, 3, 5, 8, 11];
    let result = find_even(&numbers)
        .filter(|n| *n > 5)
        .map(|n| n * 10);
    println!("找第一個偶數，> 5 才乘 10：{:?}", result);
}
```

6.7.4 重點整理

- Option 和 Result 的 map 對內部值做轉換，None / Err 時不執行。
- and_then 用於閉包本身也回傳 Option / Result 的情況，避免巢狀。
- unwrap_or_else 懶惰計算預設值，只在 None / Err 時才執行閉包。
- Option 的 filter 根據條件決定保留 Some 或轉成 None。
- Result 的 map_err 可以轉換錯誤型別，方便錯誤處理鏈。
- 這些方法可以鏈式呼叫，比層層 match 更簡潔易讀。
- 你可能已經注意到：光看型別簽名就能猜出方法在做什麼（Option<T> 的 map 接受 FnOnce(T) -> U，回傳 Option<U>）。這是函數式程式設計的一大特色——型別本身就是文件。

6.8 Iterator trait

6.8.1 本集目標

認識 Iterator trait 的核心——只要實作 next 方法，就能免費獲得數十個好用的方法。

6.8.2 概念說明

6.8.2.1 Iterator 的定義

Iterator trait 的核心簡單到不行：

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

就這樣。只有一個必須實作的方法 next，它每次被呼叫就回傳：

- Some(值) —— 還有下一個元素。
- None —— 迭代結束了。

還記得第 5 章學的 associated type 嗎？type Item 就是一個 associated type，代表「這個迭代器產出的元素型別」。

6.8.2.2 手動呼叫 next

你可以直接手動呼叫 .next() 來逐一取得元素：

```
fn main() {
    let v = vec![10, 20, 30];
    let mut iter = v.iter();

    println!("{:?}", iter.next()); // Some(&10)
    println!("{:?}", iter.next()); // Some(&20)
    println!("{:?}", iter.next()); // Some(&30)
    println!("{:?}", iter.next()); // None
}
```

注意 iter 必須是 mut 的，因為每次呼叫 .next() 都會推進內部狀態。

6.8.2.3 只需實作 next，其他方法免費送

Iterator trait 提供了大量的**預設實作**（還記得第 5 章嗎？）。因為所有的迭代操作本質上都是「不斷呼叫 next 直到 None」，所以只要你實作了 next，像 map、filter、count、sum 等幾十個方法全部自動可用。

6.8.2.4 自訂 Iterator

讓我們自己做一個迭代器。假設我們想要一個「倒數計時器」：

```
struct Countdown {
    value: i32,
}

impl Iterator for Countdown {
    type Item = i32;

    fn next(&mut self) -> Option<i32> {
        if self.value > 0 {
            let current = self.value;
            self.value -= 1;
            Some(current)
        } else {
            None
        }
    }
}
```

只要實作了 next，map、filter、sum、collect 等幾十個方法全部自動可用。這些方法接下來幾集會陸續學到。

6.8.2.5 標準庫的迭代器工廠

標準庫提供了一些方便的函數來建立迭代器：

- std::iter::repeat(value) —— 無限重複同一個值
- std::iter::from_fn(closure) —— 用閉包來決定每次 .next() 回傳什麼。

```
use std::iter;

fn main() {
    // 無限產生 42
    let mut repeater = iter::repeat(42);
    println!("{:?}", repeater.next()); // Some(42)
```

```
    println!("{:?}", repeater.next()); // Some(42)（永遠不會 None）

    // 用閉包產生遞增數字
    let mut n = 0;
    let mut counter = iter::from_fn(move || {
        n += 1;
        Some(n)
    });
    println!("{:?}", counter.next()); // Some(1)
    println!("{:?}", counter.next()); // Some(2)
}
```

注意 repeat 和 from_fn 產生的迭代器可能是**無限的**——永遠不會回傳 None。第 15 集會深入討論這個特性。

6.8.3 範例程式碼

```
use std::iter;

// 自訂迭代器：費氏數列（無限！）
struct Fibonacci {
    a: u64,
    b: u64,
}

impl Fibonacci {
    fn new() -> Fibonacci {
        Fibonacci { a: 0, b: 1 }
    }
}

impl Iterator for Fibonacci {
    type Item = u64;

    fn next(&mut self) -> Option<u64> {
        let current = self.a;
        let new_b = self.a + self.b;
        self.a = self.b;
        self.b = new_b;
        Some(current) // 永遠不回傳 None
    }
}

fn main() {
    // 手動呼叫 Vec 的 .iter() 的 .next()
    let names = vec!["Alice", "Bob", "Charlie"];
    let mut name_iter = names.iter();
    println!("第一個：{:?}", name_iter.next());
    println!("第二個：{:?}", name_iter.next());
    println!("第三個：{:?}", name_iter.next());
    println!("結束了：{:?}", name_iter.next());

    // 自訂 Iterator：費氏數列（手動呼叫 next）
    println!("\n費氏數列：");
```

```
    let mut fib = Fibonacci::new();
    println!("{:?}", fib.next()); // Some(0)
    println!("{:?}", fib.next()); // Some(1)
    println!("{:?}", fib.next()); // Some(1)
    println!("{:?}", fib.next()); // Some(2)
    println!("{:?}", fib.next()); // Some(3)
    println!("{:?}", fib.next()); // Some(5)
    // 永遠不會 None——這是一個無限迭代器

    // std::iter::repeat：無限重複
    let mut threes = iter::repeat(3);
    println!("\nrepeat(3)：");
    println!("{:?}", threes.next()); // Some(3)
    println!("{:?}", threes.next()); // Some(3)
    println!("{:?}", threes.next()); // Some(3)（永遠不會 None）

    // std::iter::from_fn：用閉包控制產出
    let mut n = 0;
    let mut squares = iter::from_fn(|| {
        n += 1;
        if n <= 3 {
            Some(n * n)
        } else {
            None
        }
    });
    println!("\nfrom_fn（前 3 個平方數）：");
    println!("{:?}", squares.next()); // Some(1)
    println!("{:?}", squares.next()); // Some(4)
    println!("{:?}", squares.next()); // Some(9)
    println!("{:?}", squares.next()); // None
}
```

6.8.4 重點整理

- Iterator trait 的核心是 next(&mut self) -> Option<Self::Item>。
- 只需實作 .next()，就能免費獲得數十個預設實作（接下來會陸續學到）。
- 自己幫型別實作 Iterator 很簡單——定義 type Item 和 next 就好。
- std::iter::repeat(value) 建立無限重複的迭代器。
- std::iter::from_fn(closure) 用閉包來控制每次產出的值。
- 迭代器可以是無限的（永不回傳 None）。

6.9 for 迴圈的真面目

6.9.1 本集目標

揭開 for 迴圈的真面目，透過 IntoIterator + while let 理解它的運作方式。

6.9.2 概念說明

6.9.2.1 for 迴圈不是魔法

從第 1 章開始我們就在用 for 迴圈：

```rust
fn main() {
    let v = vec![1, 2, 3];
    for x in v {
        println!("{}", x);
    }
}
```

看起來很簡單對吧？但這背後到底發生了什麼事？

6.9.2.2 概念上的改寫

概念上，可以把上面的 for 迴圈理解成這樣：

```rust
fn main() {
    let v = vec![1, 2, 3];
    let mut iter = v.into_iter();
    while let Some(x) = iter.next() {
        println!("{}", x);
    }
}
```

三個步驟：

1. 呼叫 v.into_iter() 把 v 轉成迭代器。
2. 反覆呼叫 iter.next()。
3. 用 while let Some(x) 解構（還記得第 3 章的 while let 嗎？），直到拿到 None 就結束。

6.9.2.3 IntoIterator trait

IntoIterator 是一個 trait，定義了「如何把自己轉成迭代器」：

```rust
trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item = Self::Item>;
    fn into_iter(self) -> Self::IntoIter;
}
```

任何實作了 IntoIterator 的型別都可以用 for 迴圈。Vec、陣列、字串切片的 .chars() ……背後都是因為實作了這個 trait。

6.9.2.4 Iterator 也實作了 IntoIterator

有個很方便的設計：每個 Iterator 都自動實作了 IntoIterator（into_iter() 直接回傳自己）。所以你可以把迭代器直接丟進 for：

```rust
fn main() {
    let v = vec![1, 2, 3];
    let iter = v.iter(); // 這是一個 Iterator
    for x in iter {      // Iterator 也實作了 IntoIterator
        println!("{}", x);
    }
}
```

6.9.3 範例程式碼

```rust
fn main() {
    // 正常的 for 迴圈
    let fruits = vec!["蘋果", "香蕉", "橘子"];
    println!("--- for 迴圈 ---");
    for fruit in fruits {
        println!("水果：{}", fruit);
    }

    // 手動改寫成 while let（概念上等價）
    let fruits = vec!["蘋果", "香蕉", "橘子"];
    println!("\n--- 手動改寫 ---");
    let mut iter = fruits.into_iter();
    while let Some(fruit) = iter.next() {
        println!("水果：{}", fruit);
    }

    // 自訂迭代器（Iterator 自動實作 IntoIterator，所以能用 for）
    println!("\n--- 自訂 Iterator ---");
    let countdown = Countdown { value: 5 };
    for n in countdown {
        print!("{} ", n);
    }
    println!("發射！");

    // 迭代器本身也可以放進 for
    println!("\n--- Iterator 直接用 for ---");
    let numbers = vec![10, 20, 30, 40, 50];
    for n in numbers.iter() {
        if *n > 20 {
            println!("大於 20 的：{}", n);
        }
    }

    // Range 也實作了 IntoIterator
    println!("\n--- Range ---");
    for i in 1..=5 {
        print!("{} ", i);
    }
    println!();
}

// 自訂迭代器
struct Countdown {
    value: i32,
}

impl Iterator for Countdown {
    type Item = i32;

    fn next(&mut self) -> Option<i32> {
        if self.value > 0 {
            let current = self.value;
            self.value -= 1;
```

```
            Some(current)
        } else {
            None
        }
    }
}
```

6.9.4 重點整理

- `for x in v` 是簡寫；概念上可以理解成 `v.into_iter()` + `while let Some(x) = iter.next()`。
- `IntoIterator` trait 定義了「如何把自己轉成迭代器」。
- 任何實作了 `IntoIterator` 的型別都能用 `for` 迴圈。
- 每個 `Iterator` 自動實作了 `IntoIterator`。
- 之所以能寫 `for i in 1..5` 或 `for i in 1..=5`，就是因為 range 實作了 `IntoIterator`。

6.10 `into_iter` / `iter_mut` / `iter`

6.10.1 本集目標

搞懂三種迭代模式的差別——消耗、可變借用、借用——以及它們和所有權系統的關係。

6.10.2 概念說明

6.10.2.1 三種迭代方式

前面提到過 `for x in v` 和 `for x in &v` 的差別。今天來補完整個概念，正式介紹 Vec 提供的三個迭代方法：

方法	產出型別	語意	Vec 之後還能用嗎？
`.into_iter()`	`T`	消耗整個集合	✗ 不行
`.iter_mut()`	`&mut T`	可變借用每個元素	✓ 可以（已修改）
`.iter()`	`&T`	借用每個元素	✓ 可以

6.10.2.2 `.into_iter()` —— 拿走一切

```
    let names = vec![String::from("Alice"), String::from("Bob")];
    for name in names.into_iter() {
        println!("{}", name); // name 是 String（擁有所有權）
    }
    println!("{:?}", names); // 編譯錯誤！names 被消耗了
```

`.into_iter()` 把每個元素的所有權交出來。集合本身被消耗，之後不能再用。

其實 `for name in names` 就等於 `for name in names.into_iter()`。

6.10.2.3 `.iter_mut()` —— 借來改改

```
fn main() {
    let mut scores = vec![60, 70, 80];
    for score in scores.iter_mut() {
```

```
        *score += 10; // score 是 &mut i32
    }
    println!("{:?}", scores); // [70, 80, 90]
}
```

.iter_mut() 回傳一個產出 &mut T 的迭代器，讓你可以原地修改每個元素。

6.10.2.4 .iter() —— 只是看看

```
fn main() {
    let names = vec![String::from("Alice"), String::from("Bob")];
    for name in names.iter() {
        println!("{}", name); // name 是 &String
    }
    println!("names 還在：{:?}", names); // 沒問題，只是借用
}
```

.iter() 回傳 &T 的迭代器。集合本身不受影響，用完還在。

6.10.2.5 對應關係

這三種方法其實對應第 4 章學的三種所有權操作：

所有權概念	迭代方法	for 簡寫
T（move）	.into_iter()	for x in v
&mut T（可變借用）	.iter_mut()	for x in &mut v
&T（借用）	.iter()	for x in &v

6.10.2.6 背後的 IntoIterator

上一集學到 for x in something 會呼叫 something.into_iter()。那三種 for 迴圈是怎麼運作的？

其實是因為 Vec<T>、&mut Vec<T>、&Vec<T> 分別實作了 IntoIterator：

```
impl<T> IntoIterator for Vec<T> {
    type Item = T;
    fn into_iter(self) -> ... { /* 消耗 Vec，產出 T */ }
}

impl<'a, T> IntoIterator for &'a mut Vec<T> {
    type Item = &'a mut T;
    fn into_iter(self) -> ... { /* 等同於 .iter_mut()，產出 &mut T */ }
}

impl<'a, T> IntoIterator for &'a Vec<T> {
    type Item = &'a T;
    fn into_iter(self) -> ... { /* 等同於 .iter()，產出 &T */ }
}
```

所以 for x in v、for x in &mut v、for x in &v，其實會分別走到 Vec<T>、&mut Vec<T>、&Vec<T> 的 IntoIterator 實作，最終拿到 T、&mut T、&T。

大部分集合型別（Vec、陣列等）都遵循這個模式——為自己、&mut self、&self 三種各實作一次 IntoIterator。

6.10.2.7 選哪一個？

- 需要拿走元素的所有權 → `.into_iter()`。
- 需要原地修改 → `.iter_mut()`。
- 只需要讀取 → `.iter()`（最常用）。

選擇剛好能提供所需存取方式的迭代方法。

6.10.3 範例程式碼

```
fn main() {
    // .into_iter() ——消耗所有權
    let words = vec![
        String::from("hello"),
        String::from("world"),
    ];
    println!("--- .into_iter()（消耗） ---");
    for word in words.into_iter() {
        println!("拿到了：{}", word); // word 是 String（擁有所有權）
    }
    // println!("{:?}", words); // 編譯錯誤！words 被消耗了

    // .iter_mut() ——可變借用，原地修改
    let mut prices = vec![100, 200, 300];
    println!("\n--- .iter_mut()（修改） ---");
    println!("打折前：{:?}", prices);
    for price in prices.iter_mut() {
        *price = *price * 8 / 10; // 打八折
    }
    println!("打折後：{:?}", prices);

    // .iter() ——借用
    let animals = vec![
        String::from("貓"),
        String::from("狗"),
        String::from("兔子"),
    ];

    println!("\n--- .iter()（借用） ---");
    for animal in animals.iter() {
        println!("動物：{}", animal);
    }
    println!("animals 還在：{:?}", animals);

    // 簡寫版的對應
    println!("\n--- 簡寫版 ---");
    let owned = vec![1, 2, 3];

    // for x in owned 等於 for x in owned.into_iter()
    for x in owned {
        print!("{} ", x);
    }
    println!("← owned（消耗）");
    // owned 已經不能用了
```

```
    let mut mutable = vec![1, 2, 3];
    // for x in &mut mutable 等於 for x in mutable.iter_mut()
    for x in &mut mutable {
        *x *= 10;
    }
    println!("{:?} ← &mut mutable（可變借用）", mutable);

    let borrowed = vec![1, 2, 3];
    // for x in &borrowed 等於 for x in borrowed.iter()
    for x in &borrowed {
        print!("{} ", x);
    }
    println!("← &borrowed（借用）");
    println!("borrowed 還在：{:?}", borrowed);
}
```

6.10.4 重點整理

- `.into_iter()` 產出 T，消耗整個集合，拿走所有權。
- `.iter_mut()` 產出 `&mut T`，可以原地修改元素。
- `.iter()` 產出 `&T`，借用元素，集合不受影響。
- `for x in v` = `.into_iter()`，`for x in &mut v` = `.iter_mut()`，`for x in &v` = `.iter()`。
- 選擇剛好能提供所需存取方式的迭代方法——要消耗就 `.into_iter()`，要改就 `.iter_mut()`，只讀就 `.iter()`。

6.11 收集

6.11.1 本集目標

學會用 `.collect()` 把迭代器收集成各種集合型別。

6.11.2 概念說明

平常我們不會花這麼多集數在介紹方法，但迭代器實在太重要了——它是 Rust 日常寫程式碼最常用的工具之一，所以接下來幾集會多花點時間。不過就算介紹了很多方法，一定還是會漏掉不少。有需要的話，請參考官方文件的 `Iterator` trait 頁面。

6.11.2.1 `.collect()` —— 迭代器的終點站

前幾集我們建立了迭代器，但迭代器本身是**惰性的**（第 15 集會詳細講）——它不會真的執行，直到有人「拉動」它。`.collect()` 就是最常用的拉動方式：把迭代器的所有元素收集成一個集合。

```
    let v: Vec<i32> = (1..=5).into_iter().collect();
```

6.11.2.2 收集成 `String`

`.collect()` 不只能收集成 `Vec`。如果迭代器產出的是 `char` 或 `&str`，可以直接收集成 `String`：

```
fn main() {
    let chars = vec!['R', 'u', 's', 't'];
    let word: String = chars.into_iter().collect();
    println!("{}", word); // "Rust"
}
```

6.11.2.3 `.last()` —— 取最後一個元素

`.last()` 會消耗整個迭代器，回傳最後一個元素（`Option<T>`）：

```
fn main() {
    let v = vec![10, 20, 30];
    let last = v.iter().last();
    println!("{:?}", last); // Some(&30)
}
```

注意它需要走完整個迭代器才能知道最後一個是什麼。

6.11.3 範例程式碼

```
fn main() {
    // 基本 collect ── Range 轉 Vec
    let numbers: Vec<i32> = (1..=10).into_iter().collect();
    println!("1 到 10：{:?}", numbers);

    // turbofish 語法
    let numbers2 = (1..=5).into_iter().collect::<Vec<i32>>();
    println!("turbofish：{:?}", numbers2);

    // 收集成 String
    let greeting: String = vec!['你', '好', '世', '界'].into_iter().collect();
    println!("字串：{}", greeting);

    // .last()
    let last_num = (1..=100).into_iter().last();
    println!("\n1..=100 的最後一個：{:?}", last_num);

    let empty: Vec<i32> = vec![];
    let last_empty = empty.iter().last();
    println!("空 Vec 的 last：{:?}", last_empty);
}
```

6.11.4 重點整理

- `.collect()` 把迭代器的元素收集成目標集合型別。
- 用型別標註 `let v: Vec<i32>` 或 turbofish `.collect::<Vec<i32>>()` 告訴 Rust 目標型別。
- 可以收集成 `Vec`、`String` 等多種型別。
- `.last()` 消耗整個迭代器，回傳 `Some` 包裝的最後一個元素。

6.12 聚合

6.12.1 本集目標

學會用迭代器的聚合方法把一整個序列「摺疊」成一個值。

6.12.2 概念說明

6.12.2.1 什麼是聚合？

前幾集我們學了怎麼建立迭代器、怎麼 collect 成集合。但有時候你不需要一個集合，你要的是一個**單一的值**——總和、最大值、個數……這就是聚合（aggregation）。

6.12.2.2 .count() —— 數有幾個

```
    let names = vec!["Alice", "Bob", "Charlie"];
    let count = names.iter().count(); // 3
```

6.12.2.3 .sum() 和 .product()

```
    let total: i32 = (1..=10).into_iter().sum();         // 55
    let factorial: i64 = (1..=10).into_iter().product(); // 3628800
```

跟 .collect() 一樣，.sum() 和 .product() 需要你指定回傳型別，通常用型別標註解決。

6.12.2.4 .min() 和 .max()

```
    let v = vec![3, 1, 4, 1, 5, 9, 2, 6];
    let smallest = v.iter().min(); // Some(&1)
    let largest = v.iter().max();  // Some(&9)
```

回傳 Option，因為迭代器可能是空的（空的就回傳 None）。

6.12.2.5 .fold(init, f) —— 最通用的聚合

fold 是所有聚合方法的「老大」。它的型別：

```
fn fold<B>(self, init: B, f: impl FnMut(B, Self::Item) -> B) -> B;
```

接受一個初始值 init（型別 B）和一個閉包，每一步把「累積值」和「當前元素」組合成新的累積值：

```
    let sum = (1..=5).into_iter().fold(0, |acc, x| acc + x);
    // 步驟：0+1=1, 1+2=3, 3+3=6, 6+4=10, 10+5=15
```

其實本集介紹的其他方法都能用 fold 實作：

```
    // count = fold 從 0 開始，每次 +1
    let count = (1..=5).into_iter().fold(0, |acc, _x| acc + 1);

    // sum = fold 從 0 開始，每次加上元素
    let sum = (1..=5).into_iter().fold(0, |acc, x| acc + x);

    // product = fold 從 1 開始，每次乘上元素
    let product = (1..=5).into_iter().fold(1, |acc, x| acc * x);

    // min / max 的實作留給底下的 reduce 做——用 fold 的話不太自然
```

fold 還能做更靈活的事情。想把數字串成字串？想同時追蹤多個值？都可以：

```
    let text = (1..=5).into_iter().fold(String::new(), |mut acc, x| {
        if !acc.is_empty() {
            acc.push_str(", ");
        }
        acc.push_str(&x.to_string());
```

```
        acc
    });
    // "1, 2, 3, 4, 5"
```

6.12.2.6 `.reduce(f)` —— 沒有初始值的 `fold`

`reduce` 跟 `fold` 很像，但它用第一個元素當初始值：

```
    let product = vec![2, 3, 4].into_iter().reduce(|acc, x| acc * x);
    // Some(24)：2*3=6, 6*4=24
```

因為可能沒有第一個元素（迭代器是空的），所以 `reduce` 回傳 `Option`。

用 `reduce` 實作 `min` 和 `max` 就很自然：

```
    let min = vec![3, 1, 4, 1, 5].into_iter()
        .reduce(|a, b| if a < b { a } else { b });
    // Some(1)

    let max = vec![3, 1, 4, 1, 5].into_iter()
        .reduce(|a, b| if a > b { a } else { b });
    // Some(5)
```

因為 `reduce` 本身就回傳 `Option`，空迭代器自動得到 `None`—— `fold` 需要特別處理空迭代器的情形。

6.12.3 範例程式碼

```
fn main() {
    let scores = vec![85, 92, 78, 95, 88, 76, 91];

    // .count()
    let total = scores.iter().count();
    println!("總共 {} 個分數", total);

    // .sum()
    let sum: i32 = scores.iter().sum();
    println!("總分：{}", sum);

    // .min() / .max()
    let min = scores.iter().min();
    let max = scores.iter().max();
    println!("最低分：{:?}，最高分：{:?}", min, max);

    // .product()
    let factorial: i64 = (1..=10).into_iter().product();
    println!("\n10! = {}", factorial);

    // .fold() ——計算平均分
    let (count2, sum2) = scores.iter().fold((0, 0), |(c, s), &score| {
        (c + 1, s + score)
    });
    println!("\n用 fold 算平均：{} / {} = {}", sum2, count2, sum2 / count2);

    // .fold() ——把數字串成字串
    let nums = vec![1, 2, 3, 4, 5];
    let formatted = nums.iter().fold(String::new(), |mut acc, &n| {
```

```
        if !acc.is_empty() {
            acc.push_str(" → ");
        }
        acc.push_str(&n.to_string());
        acc
    });
    println!("連接：{}", formatted);

    // .reduce() ——找最長的字串
    let words = vec!["cat", "elephant", "dog", "hippopotamus"];
    let longest = words
        .iter()
        .reduce(|a, b| if a.len() >= b.len() { a } else { b });
    println!("\n最長的字：{:?}", longest);

    // .reduce() 回傳 Option（空迭代器的情況）
    let empty: Vec<i32> = vec![];
    let result = empty.into_iter().reduce(|a, b| a + b);
    println!("空 Vec 的 reduce：{:?}", result);
}
```

6.12.4 重點整理

- `.count()` 計算元素個數。
- `.sum()` 和 `.product()` 計算總和與乘積，需要標註回傳型別。
- `.min()` 和 `.max()` 回傳 `Option`，因為迭代器可能是空的。
- `.fold(init, |acc, x| ...)` 是最通用的聚合——用初始值和閉包逐步累積。
- `.reduce(|acc, x| ...)` 類似 `fold` 但用第一個元素當初始值，回傳 `Option`。
- 聚合方法會消耗整個迭代器，產出一個單一的值。

6.13 組合與截取

6.13.1 本集目標

學會用 `zip`、`enumerate`、`chain`、`take`、`skip`、`flatten` 來組合和截取迭代器。

6.13.2 概念說明

6.13.2.1 `.zip(iter)` —— 把兩個迭代器配對

`zip` 把兩個迭代器「拉鍊式」地配對起來，產出 tuple：

```
    let names = vec!["Alice", "Bob", "Charlie"];
    let scores = vec![90, 85, 92];
    let paired: Vec<_> = names.iter().zip(scores.iter()).collect();
    // [("Alice", 90), ("Bob", 85), ("Charlie", 92)]
```

如果兩個迭代器長度不同，`zip` 在較短的那個結束時就停止。

6.13.2.2 `.enumerate()` —— 帶上索引

```
    let names = vec!["Alice", "Bob", "Charlie"];
    for (i, name) in names.iter().enumerate() {
```

```
        println!("第 {} 個：{}", i, name);
    }
```

enumerate 把每個元素包成 (index, element) 的 tuple，索引從 0 開始。

6.13.2.3 .chain(iter) —— 串接兩個迭代器

chain 把兩個迭代器首尾相接：

```
    let first = vec![1, 2, 3];
    let second = vec![4, 5, 6];
    let all: Vec<i32> = first.into_iter().chain(second.into_iter()).collect();
    // [1, 2, 3, 4, 5, 6]
```

6.13.2.4 .take(n) —— 只取前 n 個

```
    let first_three: Vec<i32> = (1..=100).into_iter().take(3).collect();
    // [1, 2, 3]
```

6.13.2.5 .skip(n) —— 跳過前 n 個

```
    let after_skip: Vec<i32> = (1..=10).into_iter().skip(7).collect();
    // [8, 9, 10]
```

6.13.2.6 .flatten() —— 把巢狀結構攤平

如果迭代器的元素本身也是迭代器（或 Option、Vec 等），flatten 可以把它攤平一層：

```
    let nested = vec![vec![1, 2], vec![3, 4], vec![5]];
    let flat: Vec<i32> = nested.into_iter().flatten().collect();
    // [1, 2, 3, 4, 5]
```

Option 也可以 flatten——Some(value) 被取出，None 被忽略：

```
    let options = vec![Some(1), None, Some(3), None, Some(5)];
    let values: Vec<i32> = options.into_iter().flatten().collect();
    // [1, 3, 5]
```

這是因為 Option 也實作了 IntoIterator。

6.13.3 範例程式碼

```
fn main() {
    // zip ——名字和分數配對
    let students = vec!["小明", "小華", "小美"];
    let grades = vec![88, 95, 72];
    println!("--- zip ---");
    for (name, grade) in students.iter().zip(grades.iter()) {
        println!("{}：{} 分", name, grade);
    }

    // enumerate ——帶索引
    println!("\n--- enumerate ---");
    let fruits = vec!["蘋果", "香蕉", "櫻桃"];
    for (i, fruit) in fruits.iter().enumerate() {
        println!("第 {} 個：{}", i + 1, fruit);
```

```
    }

    // chain ——串接兩個 Vec
    let morning = vec!["開會", "寫報告"];
    let afternoon = vec!["寫程式", "code review"];
    let all_tasks: Vec<&&str> = morning.iter().chain(afternoon.iter()).collect();
    println!("\n今日行程：{:?}", all_tasks);

    // take 和 skip
    let numbers: Vec<i32> = (1..=20).into_iter().collect();
    let first_five: Vec<&i32> = numbers.iter().take(5).collect();
    let last_five: Vec<&i32> = numbers.iter().skip(15).collect();
    println!("\n前 5 個：{:?}", first_five);
    println!("跳過 15 個後：{:?}", last_five);

    // take + skip 組合：取中間的
    let middle: Vec<&i32> = numbers.iter().skip(5).take(5).collect();
    println!("第 6~10 個：{:?}", middle);

    // flatten ——攤平巢狀 Vec
    let matrix = vec![
        vec![1, 2, 3],
        vec![4, 5, 6],
        vec![7, 8, 9],
    ];
    let flat: Vec<i32> = matrix.into_iter().flatten().collect();
    println!("\n攤平矩陣：{:?}", flat);

    // flatten ——過濾 Option
    let maybe_values = vec![Some(10), None, Some(30), None, Some(50)];
    let real_values: Vec<i32> = maybe_values.into_iter().flatten().collect();
    println!("有值的：{:?}", real_values);

    // zip + map 組合，下集就會教迭代器的 map
    println!("\n--- zip + map ---");
    let prices = vec![100, 200, 300];
    let quantities = vec![2, 1, 4];
    let grand_total: i32 = prices.iter()
        .zip(quantities.iter())
        .map(|(p, q)| p * q)
        .sum();
    println!("總計：{}", grand_total);
}
```

6.13.4 重點整理

- `.zip(iter)` 把兩個迭代器配對成 tuple，以較短的為準。
- `.enumerate()` 為每個元素加上從 0 開始的索引。
- `.chain(iter)` 把兩個迭代器首尾串接。
- `.take(n)` 只取前 n 個元素，`.skip(n)` 跳過前 n 個。
- `.flatten()` 把巢狀結構攤平一層（`Vec<Vec<T>>` → `Vec<T>`，也適用於 `Option`）。
- 這些方法可以自由組合，打造出強大的資料處理管道。

6.14 轉換與過濾

6.14.1 本集目標

學會迭代器最常用的轉換與過濾方法，以及如何用鏈式呼叫組合出強大的資料管道。

6.14.2 概念說明

6.14.2.1 .map(f) —— 轉換每個元素

map 對每個元素套用閉包，產出轉換後的新元素：

```
let doubled: Vec<i32> = vec![1, 2, 3].iter().map(|x| x * 2).collect();
// [2, 4, 6]
```

注意！.iter() 產出 &T，所以閉包的參數是 &i32。如果不想處理參考，可以搭配 .copied()（等等會講）。

6.14.2.2 .flat_map(f) —— map + flatten

flat_map 等於先 map 再 flatten（上一集學的）。每個元素經過閉包轉換成一個迭代器，然後全部攤平：

```
let words = vec!["abc", "de", "f"];
let chars: Vec<char> = words.iter().flat_map(|s| s.chars()).collect();
// ['a', 'b', 'c', 'd', 'e', 'f']
```

還記得第 7 集 Option 和 Result 的 and_then 嗎？flat_map 在迭代器上做的事情本質上一樣——「轉換，但因為轉換結果本身也是容器，就攤平」。

6.14.2.3 .filter(pred) —— 過濾元素

filter 只保留閉包回傳 true 的元素：

```
let numbers = vec![1, 2, 3, 4, 5];
let evens: Vec<&i32> = numbers.iter().filter(|&&x| x % 2 == 0).collect();
// [&2, &4]
```

filter 的閉包接收 &&T（因為 .iter() 已經是 &T，filter 再借用一次就是 &&T）。這是初學者常被搞混的地方，但寫多了就習慣了。

6.14.2.4 .copied() 和 .cloned()

當迭代器產出參考（&T）但你想要值（T）時，可以用這兩個方法把每個元素逐個複製出來：

- .copied() —— 要求 T: Copy，對每個 &T 做 copy 得到 T。
- .cloned() —— 要求 T: Clone，對每個 &T 呼叫 .clone() 得到 T。

```
let numbers = vec![1, 2, 3];
let owned: Vec<i32> = numbers.iter().copied().collect();
// 從 &i32 變成 i32
```

.copied() 常搭配 .filter() 一起用，可以避免 &&T 的困擾：

```
let evens: Vec<i32> = vec![1, 2, 3, 4, 5]
    .iter()
    .copied()
    .filter(|x| x % 2 == 0)
```

```
        .collect();
    // [2, 4]，乾淨多了！
```

6.14.2.5 .rev() —— 反轉迭代順序

```
    let reversed: Vec<i32> = (1..=5).into_iter().rev().collect();
    // [5, 4, 3, 2, 1]
```

.rev() 需要迭代器實作 DoubleEndedIterator trait——也就是說，它必須能從兩端取元素。Vec、陣列等都支援，但像 from_fn 產出的迭代器就不支援（因為沒有「尾端」的概念）。

6.14.2.6 鏈式呼叫的威力

迭代器的方法可以自由串接，形成資料處理管道：

```
    let result: Vec<String> = names
        .iter()
        .enumerate()
        .filter(|(_, name)| name.len() > 3)
        .map(|(i, name)| format!("#{}: {}", i + 1, name))
        .collect();
```

每一步都做一件小事，串在一起就能做很複雜的操作。而且因為迭代器是惰性的（下一集會講），中間不會產生額外的 Vec。

6.14.3 範例程式碼

```
fn main() {
    let scores = vec![55, 82, 91, 47, 73, 88, 69, 95];

    // map ——每個分數加 5 分（加分調整）
    let adjusted: Vec<i32> = scores.iter().map(|s| s + 5).collect();
    println!("加分後：{:?}", adjusted);

    // flat_map ——每個字拆成字元
    let words = vec!["Rust", "好棒"];
    let all_chars: Vec<char> = words.iter().flat_map(|w| w.chars()).collect();
    println!("所有字元：{:?}", all_chars);

    // flat_map 類似 and_then ——解析成功的留下，失敗的丟掉
    let inputs = vec!["42", "not_a_number", "7"];
    let parsed: Vec<i32> = inputs.iter().flat_map(|s| s.parse::<i32>()).collect();
    println!("成功解析的：{:?}", parsed);

    // filter ——篩出及格的
    let passing: Vec<i32> = scores.iter().copied().filter(|&s| s >= 60).collect();
    println!("及格的：{:?}", passing);

    // copied ——從 &i32 變成 i32
    let max_score: Option<i32> = scores.iter().copied().max();
    println!("\n最高分：{:?}", max_score);

    // cloned ——從 &String 變成 String
    let names = vec![String::from("Alice"), String::from("Bob")];
    let cloned_names: Vec<String> = names.iter().cloned().collect();
```

```
    println!("cloned: {:?}", cloned_names);
    println!("原本還在：{:?}", names);

    // rev ——反轉
    let countdown: Vec<i32> = (1..=5).into_iter().rev().collect();
    println!("\n倒數：{:?}", countdown);

    // 鏈式組合
    println!("\n--- 鏈式組合 ---");
    let long_words: Vec<&str> = vec!["hi", "hello", "hey", "howdy", "greetings"]
        .into_iter()
        .filter(|w| w.len() >= 4)
        .collect();
    println!("4 字以上的：{:?}", long_words);

    // filter + map 組合
    let words = vec!["hello", "hi", "hey", "howdy", "greetings"];
    let long_upper: Vec<String> = words
        .iter()
        .filter(|w| w.len() >= 4)
        .map(|w| w.to_uppercase())
        .collect();
    println!("\n4 字以上轉大寫：{:?}", long_upper);
}
```

6.14.4 重點整理

- `.map(f)` 轉換每個元素，`.filter(pred)` 過濾不符合條件的元素。
- `.flat_map(f)` = `.map(f)` + `.flatten()`，概念上跟 `Option` / `Result` 的 `and_then` 類似。
- `.copied()` 把 `&T` 逐個轉成 `T`（需要 `T: Copy`），`.cloned()` 類似但用 `Clone`。
- `.rev()` 反轉迭代順序，需要 `DoubleEndedIterator`。
- 這些方法可以自由鏈式呼叫，形成清晰的資料處理管道。
- 配合 `.copied()` 可以避免 `filter` 中惱人的 `&&T` 問題。

6.15 惰性求值

6.15.1 本集目標

理解迭代器的惰性（lazy）本質——`.map(f)` 和 `.filter(pred)` 不會立刻執行，而是建立巢狀結構，等 `.collect()` 或 `for` 才逐一拉動。

6.15.2 概念說明

6.15.2.1 迭代器是惰性的

這可能是整個第 6 章最重要的概念：**迭代器的轉換方法不會立刻執行**。

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];
    let iter = v.iter().map(|x| {
        println!("處理 {}", x);
        x * 2
    });
```

```
    // 到這裡為止，什麼都沒有印出來！
}
```

map 並沒有「跑過」每個元素。它只是建立了一個新的迭代器結構，記錄了「等下要做什麼」。直到有人呼叫 collect()、for、sum() 等「消費」方法時，才會一個一個元素地拉動。

6.15.2.2 俄羅斯套娃

每次呼叫 .map(f) 或 .filter(pred)，你其實是在迭代器外面「套一層」。就像俄羅斯套娃：

```
    v.iter()                 // 最內層：原始迭代器
        .filter(|x| **x > 2) // 第二層：Filter 結構，存著 inner + 閉包
        .map(|x| x * 10);    // 第三層：Map 結構，存著 inner + 閉包
```

每一層都是一個 struct，裡面存著內層的迭代器和自己的閉包。標準庫的 Map 和 Filter 大致長這樣：

```
struct Map<I, F> {
    iter: I, // 內層迭代器
    f: F,    // 要套用的閉包
}

struct Filter<I, P> {
    iter: I,      // 內層迭代器
    predicate: P, // 過濾條件的閉包
}
```

它們的 .next() 實作也很直覺：

```
// Map 的 next()：從內層拿一個元素，套用閉包
impl<B, I: Iterator, F: FnMut(I::Item) -> B> Iterator for Map<I, F> {
    type Item = B;
    fn next(&mut self) -> Option<B> {
        let x = self.iter.next()?; // 問內層要一個元素
        Some((self.f)(x))          // 套用閉包回傳
    }
}

// Filter 的 next()：不斷從內層拿，直到找到符合條件的
impl<I: Iterator, P: FnMut(&I::Item) -> bool> Iterator for Filter<I, P> {
    type Item = I::Item;
    fn next(&mut self) -> Option<I::Item> {
        loop {
            let x = self.iter.next()?; // 問內層要一個元素
            if (self.predicate)(&x) {
                return Some(x);        // 符合條件，回傳
            }
            // 不符合，繼續問下一個
        }
    }
}
```

所以整條鏈就是一堆 struct 套在一起——呼叫最外層的 .next()，它去問內層，內層再問更內層，一路拉到最底。

6.15.2.3 pull-based：一次只處理一個元素

當你呼叫 `.collect()` 或 `for` 迴圈時，最外層的迭代器開始「拉」：

1. 最外層（Map）問第二層（Filter）：「給我下一個元素」。
2. Filter 問最內層（原始迭代器）：「給我下一個元素」。
3. 最內層回傳 `Some(&1)`。
4. Filter 檢查條件：`1 > 2`？不通過。再問一次。
5. 最內層回傳 `Some(&2)`。
6. Filter 檢查：`2 > 2`？不通過。再問。
7. 最內層回傳 `Some(&3)`。
8. Filter 檢查：`3 > 2`？通過！回傳給 Map。
9. Map 套用閉包：`3 * 10 = 30`，回傳 `Some(30)`。

每個元素是**一路到底**處理完的——不像先做完所有 `filter`，再做所有 `map`。這意味著**中間不需要任何暫存的 `Vec`**。

6.15.2.4 無限迭代器

因為是惰性的，迭代器可以是**無限的**。`std::iter::repeat` 和 `std::iter::from_fn` 都可以產生永遠不回傳 `None` 的迭代器：

```
use std::iter;

fn main() {
    // 永遠產出 1, 2, 3, 4, 5, ...
    let mut n = 0;
    let naturals = iter::from_fn(move || {
        n += 1;
        Some(n)
    });
}
```

這不會無窮迴圈，因為迭代器是惰性的——沒人呼叫 `.next()` 就什麼都不會發生。

6.15.2.5 `.take(n)` 馴服無限迭代器

用 `.take(n)` 就能從無限迭代器中取出有限個元素：

```
    let first_ten: Vec<i32> = naturals.take(10).collect();
    // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

這就是惰性求值的威力——你可以先描述一個「概念上無限」的計算，最後再決定要取多少。

6.15.2.6 不小心忘記消費？

因為迭代器是惰性的，如果你寫了 `.map(f)` 但忘記 `.collect()` 或 `for`，什麼事都不會發生。Rust 編譯器會發出警告：

```
warning: unused `Map` that must be used
note: iterators are lazy and do nothing unless consumed
```

看到這個警告就知道：你忘了消費迭代器了。

6.15.3 範例程式碼

```rust
use std::iter;

fn main() {
    // 惰性示範：map 不會立刻執行
    println!("--- 惰性示範 ---");
    let v = vec![1, 2, 3];
    let iter = v.iter().map(|x| {
        println!("  處理 {}", x);
        x * 2
    });
    println!("map 建立完了，但還沒執行...");
    println!("現在開始 collect：");
    let result: Vec<i32> = iter.collect();
    println!("結果：{:?}", result);

    // pull-based：filter + map 一次處理一個元素
    println!("\n--- Pull-based 示範 ---");
    let data = vec![1, 2, 3, 4, 5, 6];
    let processed: Vec<i32> = data
        .iter()
        .filter(|&&x| {
            println!("  filter 檢查 {}", x);
            x % 2 == 0
        })
        .map(|&x| {
            println!("  map 處理 {}", x);
            x * 10
        })
        .collect();
    println!("結果：{:?}", processed);
    // 注意印出的順序！filter 和 map 是交替執行的

    // from_fn 建立無限迭代器（取前 10 個質數）
    let mut candidate = 1;
    let primes: Vec<i32> = iter::from_fn(move || {
        loop {
            candidate += 1;
            let is_prime = (2..candidate).into_iter().all(|d| candidate % d != 0);
            if is_prime {
                return Some(candidate);
            }
        }
    })
    .take(10)
    .collect();
    println!("\n前 10 個質數：{:?}", primes);

    // 不需要中間 Vec——全部在一條管道裡
    println!("\n--- 零中間 Vec ---");
    let sum_of_even_squares: i32 = (1..=100)
        .into_iter()
        .filter(|x| x % 2 == 0)
        .map(|x| x * x)
```

```
        .sum();
    println!("1~100 偶數的平方和：{}", sum_of_even_squares);
    // 沒有任何中間的 Vec 被建立，全部是一次一個元素處理完的
}
```

6.15.4 重點整理

- 迭代器的 `.map(f)` / `.filter(pred)` 等方法是**惰性的**，不會立刻執行。
- 每次呼叫轉換方法都是在外面「套一層」`struct`（俄羅斯套娃）。
- 消費（`.collect()`、`for`、`.sum()` 等）才會觸發執行。
- 執行方式是 **pull-based**——一次拉一個元素，完整通過所有層，不需要中間 `Vec`。
- 因為惰性，迭代器可以是**無限的**。
- 用 `.take(n)` 從無限迭代器中取出有限個元素。
- 忘記消費迭代器的話，編譯器會發出警告提醒你。

恭喜你完成了第 6 章！🎉 從函數指標到閉包的三種 `Fn` `trait`，再到迭代器的惰性求值——這一章結合了所有權、`trait`、泛型等前面學過的概念，展現了 Rust 函數式程式設計的威力。你現在已經能寫出簡潔、高效、不需要中間暫存的資料處理管道了。下一章我們將學習 Cargo、`crate` 與 `mod` 系統——讓你的程式碼從單一檔案擴展到真正的專案結構！

第 7 章

Cargo、Crate 與 Mod 系統

本章的主題比較簡單，講的是如何分拆管理程式碼以及限制程式碼的可見範圍，最終建立自己的專案。這些功能並沒有太高的理論難度，但專案等級的模組化卻讓軟體工程師方便許多。

7.1 Cargo 與 crates.io

7.1.1 本集目標

認識 Cargo 的更多功能以及如何透過 crates.io 使用社群 `crate`。

7.1.2 概念說明

我們從第 1 章開始就在用 `cargo new` 和 `cargo run`。其實 `cargo run` 背後做了兩件事：先**編譯**你的程式碼，再**執行**編譯出來的執行檔。如果你只想編譯但不執行，可以用 `cargo build`——它只會產生執行檔，放在 `target/debug/` 資料夾裡。

這一集我們來多認識一些 Cargo 的功能，特別是怎麼引入外部 `crate`。

7.1.2.1 debug build vs release build

`cargo build` 和 `cargo run` 預設跑的是 **debug 模式**——編譯快但執行慢（沒有最佳化）。當你要發布程式的時候，加上 `--release`：

```
cargo build --release
```

這會產生最佳化過的執行檔，放在 `target/release/` 而不是 `target/debug/`。差異可以非常大——有些程式 release 版本跑起來快好幾倍。

7.1.2.2 Cargo.toml

每個 Cargo 專案的根目錄都有 `Cargo.toml`。TOML 是一種設定檔格式，設計得讓人好讀好寫。

一個典型的 `Cargo.toml` 長這樣：

```
[package]
name = "my_project"
version = "0.1.0"
edition = "2024"

[dependencies]
```

- `[package]`：專案的基本資訊（名稱、版本、Rust edition）
- `[dependencies]`：這個專案用到的外部 `crate`

其中 `edition` 是 Rust 的**版本號**——但不是 Rust 編譯器的版本，而是**語言規格的版本**。Rust 每隔幾年會發布一個新的 edition（2015、2018、2021、2024），每次可能會微調一些語法或預設行為。不

同 edition 編寫的 crate 可以互相搭配使用，所以不用擔心相容性問題。cargo new 會自動幫你設成最新的 edition。

7.1.2.3 加入外部 crate

想用別人寫好的 crate？最簡單的方式：

```
cargo add rand
```

這會自動在 Cargo.toml 的 [dependencies] 加上類似這樣的一行：

```
[dependencies]
rand = "0.10"
```

實際加上的版本號取決於你執行 cargo add 時的最新版本，不一定和這裡寫的一樣。

7.1.2.4 crates.io

crates.io 是 Rust 的官方 crate registry。你可以在上面搜尋 crate、看下載數、閱讀文件。每個 crate 頁面都會有：

- 使用說明和版本歷史
- 連結到 docs.rs 的自動產生文件。
- 下載數（可以當作 crate 熱門程度的參考）。

7.1.2.5 依賴的版本語意

在 [dependencies] 裡指定外部 crate 的版本時，有不同的寫法：

- "^1.0"（或直接寫 "1.0"）：相容 1.x.y 的任何版本，但不會升到 2.0。
- "=1.0.0"：鎖定**剛好**這個版本。
- ">=1.2, <1.5"：指定範圍。

大多數時候用預設的 ^ 就好，Cargo 會幫你選合適的版本。更多細節可以參考官方文件。

7.1.2.6 Cargo features

有些 crate 提供可選功能，用 features 開啟：

```
[dependencies]
serde = { version = "1.0", features = ["derive"] }
```

這樣就能用 serde 的 #[derive(Serialize, Deserialize)]，而不需要的功能不會被編譯進來。

7.1.3 範例程式碼

用 rand crate 產生隨機數：

```
// 先執行：cargo add rand
extern crate rand;

use rand::RngExt;

fn main() {
    let mut rng = rand::rng();

    let n: u32 = rng.random_range(1..=100);
    println!("隨機數字：{}", n);
```

```
    let coin: bool = rng.random();
    if coin {
        println!("正面！");
    } else {
        println!("反面！");
    }
}
```

7.1.4 重點整理

- `cargo build --release` 產生最佳化的執行檔，適合發布。
- `Cargo.toml` 用 TOML 格式，`[package]` 記專案資訊，`[dependencies]` 記外部 crate。
- `edition` 是 Rust 語言規格的版本（2015、2018、2021、2024），不同 edition 編寫的 crate 可以互相搭配。
- `cargo add <crate 名稱>` 是加入外部 crate 最快的方式。
- crates.io 是 Rust 的官方 crate registry，docs.rs 提供自動產生的文件。
- 版本號 `"1.0"` 等同 `"^1.0"`，允許相容升級；`"=1.0.0"` 鎖定精確版本。
- `features` 可以開啟 crate 的可選功能。

7.2 mod

7.2.1 本集目標

學會用 mod 將程式碼組織成有層次的結構。

7.2.2 概念說明

當程式越寫越長，全部塞在一個 `main.rs` 裡面會變得很難維護。這時候我們需要把相關的函數、struct、enum 分組——在 Rust 裡，這個分組機制就是 **模組（mod）**。

7.2.2.1 在同一個檔案裡定義 mod

最簡單的用法：直接在檔案裡用 mod 關鍵字建立一個區塊。

```
mod math {
    pub fn add(a: i32, b: i32) -> i32 {
        a + b
    }

    pub fn multiply(a: i32, b: i32) -> i32 {
        a * b
    }
}
```

要呼叫 mod 裡的函數，用 `::` 路徑語法：

```
    let result = math::add(3, 5);
```

注意那個 pub——mod 裡的東西**預設是私有的**。如果不加 pub，外面就看不到、用不了。關於 pub 的完整規則我們在第 4 集會詳細講，這裡先記住：想讓外面用，就加 pub。

7.2.2.2 巢狀 mod

mod 可以一層一層巢狀：

```
mod math {
    pub mod basic {
        pub fn add(a: i32, b: i32) -> i32 {
            a + b
        }
    }

    pub mod advanced {
        pub fn power(base: i32, exp: u32) -> i32 {
            let mut result = 1;
            for _ in 0..exp {
                result *= base;
            }
            result
        }
    }
}
```

呼叫的時候就用完整路徑：

```
    let sum = math::basic::add(2, 3);
    let p = math::advanced::power(2, 10);
```

這就像檔案系統的資料夾結構一樣——math 底下有 basic 和 advanced 兩個子 mod。

7.2.3 範例程式碼

```
mod geometry {
    pub struct Rectangle {
        pub width: f64,
        pub height: f64,
    }

    impl Rectangle {
        pub fn new(width: f64, height: f64) -> Rectangle {
            Rectangle { width, height }
        }

        pub fn area(&self) -> f64 {
            self.width * self.height
        }
    }

    pub mod utils {
        pub fn describe_shape(name: &str, area: f64) {
            println!("{} 的面積是 {}", name, area);
        }
    }
}

fn main() {
    let rect = geometry::Rectangle::new(10.0, 5.0);
```

```
    let area = rect.area();
    geometry::utils::describe_shape("長方形", area);
}
```

7.2.4 重點整理

- `mod name { ... }` 在同一個檔案裡建立 mod。
- mod 裡的東西用 `mod_name::item` 的路徑語法呼叫。
- mod 內的所有項目**預設是私有的**，要讓外部使用需加 `pub`。
- mod 可以巢狀，路徑就越來越長：`a::b::c::func()`。
- mod 是 Rust 組織程式碼的基本單位，就像資料夾組織檔案一樣。

7.3 檔案 mod

7.3.1 本集目標

學會將 mod 拆分到不同檔案，理解 Rust 的檔案與 mod 對應規則。

7.3.2 概念說明

上一集我們把 mod 寫在同一個檔案裡，但實際專案不可能全部塞在一起。Rust 提供了一套規則，讓你把 mod 拆到獨立的檔案中。

7.3.2.1 基本拆分：mod + 獨立檔案

假設你有一個 `math` mod，想把它搬到自己的檔案。做法很簡單：

1. 在 `main.rs`（或 `lib.rs`）裡寫 `mod math;`（注意結尾是分號，不是大括號）。
2. 建立 `math.rs`，把 mod 的內容放進去。

```
src/
├── main.rs
└── math.rs
```

main.rs：

```
mod math;

fn main() {
    let result = math::add(3, 5);
    println!("3 + 5 = {}", result);
}
```

math.rs：

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

pub fn subtract(a: i32, b: i32) -> i32 {
    a - b
}
```

注意 `math.rs` 裡面**不需要再寫 `mod math { ... }`**——檔案本身就代表那個 mod。

7.3.2.2　子 mod 的資料夾結構

如果 math mod 底下還有子 mod，有兩種組織方式：

方式一：用 mod.rs（傳統風格）

```
src/
├── main.rs
└── math/
    ├── mod.rs
    ├── basic.rs
    └── advanced.rs
```

math/mod.rs 就是 math mod 的入口，裡面聲明子 mod：

```rust
// math/mod.rs
pub mod basic;
pub mod advanced;
```

方式二：同名檔案 + 資料夾（推薦）

```
src/
├── main.rs
├── math.rs          ← math mod 的入口
└── math/
    ├── basic.rs
    └── advanced.rs
```

```rust
// math.rs
pub mod basic;
pub mod advanced;
```

兩種方式效果完全一樣，選你喜歡的就好。比較新的專案傾向用方式二，因為不會有一堆檔案都叫 mod.rs，在編輯器裡比較好辨認。

7.3.2.3　lib.rs vs main.rs

一個 Rust 專案可以包含一或多個 crate。crate 分成兩種類型：

- **binary crate**：有 src/main.rs，會編譯成可執行檔。
- **library crate**：有 src/lib.rs，給別人使用的函式庫。

一個專案可以**同時包含** main.rs 和 lib.rs。main.rs 是 binary crate 的根，lib.rs 是 library crate 的根。

```
src/
├── main.rs    ← binary crate root
├── lib.rs     ← library crate root
├── math.rs
└── math/
    ├── basic.rs
    └── advanced.rs
```

在 main.rs 裡可以用 crate 名稱引用 lib.rs 裡的東西：

```rust
// main.rs
// 假設 Cargo.toml 的 [package] name = "my_project"
use my_project::math;
```

```rust
fn main() {
    let result = math::basic::add(1, 2);
    println!("{}", result);
}
```

7.3.3 範例程式碼

由於檔案 mod 涉及多個檔案，無法用單一檔案示範。以下是完整的多檔案範例，建立對應的檔案結構後用 cargo run 執行：

```
src/
├── main.rs
├── math.rs
└── math/
    ├── basic.rs
    └── advanced.rs
```

main.rs：

```rust
mod math;

fn main() {
    let sum = math::basic::add(10, 20);
    println!("10 + 20 = {}", sum);

    let p = math::advanced::power(2, 8);
    println!("2 ^ 8 = {}", p);
}
```

math.rs：

```rust
pub mod basic;
pub mod advanced;
```

math/basic.rs：

```rust
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

math/advanced.rs：

```rust
pub fn power(base: i32, exp: u32) -> i32 {
    let mut result = 1;
    for _ in 0..exp {
        result *= base;
    }
    result
}
```

7.3.4 重點整理

- mod math;（分號結尾）告訴 Rust 去找子 mod。
- 被拆出去的檔案裡**不需要**再寫 mod math { ... }，檔案本身就是 mod。
- 子 mod 可以用 math/mod.rs（傳統）或 math.rs + math/ 資料夾（推薦）。
- main.rs 是 binary crate 的根，lib.rs 是 library crate 的根。

- 一個專案可以同時包含 binary crate 和 library crate。

7.4 pub 可見性

7.4.1 本集目標

完整理解 Rust 的可見性規則，掌握 pub 的各種用法。

7.4.2 正文

第 2 集提到 mod 裡的東西預設是私有的，這一集我們來把可見性規則講清楚。

7.4.2.1 預設私有

Rust 的哲學是**預設封閉**——所有東西一開始都是私有的，你必須明確地用 pub 開放。這跟有些語言預設公開的設計完全相反。

```
mod secrets {
    fn hidden() {
        // 外面看不到我
    }

    pub fn visible() {
        // 外面可以呼叫我
        hidden(); // 同 mod 內可以互相呼叫
    }
}

fn main() {
    secrets::visible(); // OK
    secrets::hidden();  // 編譯錯誤！hidden 是私有的
}
```

你可能會好奇：fn main() 和 mod secrets 都沒加 pub，為什麼 main 能看到 secrets？因為它們都定義在根 mod 裡——同一個 mod 的成員互相看得到，不需要 pub。pub 是用來讓**其他 mod** 看到你的東西的。

7.4.2.2 pub fn

函數加 pub 就對外公開，沒什麼好說的。

7.4.2.3 pub struct —— 欄位要個別標記

struct 加 pub 只是讓這個**型別**公開，欄位還是私有的！每個欄位要**個別**加 pub：

```
mod user {
    pub struct Profile {
        pub name: String,  // 外部可讀寫
        pub email: String, // 外部可讀寫
        age: u32,          // 私有！外部看不到
    }

    impl Profile {
        pub fn new(name: String, email: String, age: u32) -> Profile {
            Profile { name, email, age }
        }
```

```
        pub fn age(&self) -> u32 {
            self.age // 透過方法公開唯讀存取
        }
    }
}

fn main() {
    let p = user::Profile::new(
        String::from("Yaju"),
        String::from("yaju@senpai.com"),
        24,
    );
    println!("名字：{}", p.name);  // OK，name 是 pub
    println!("年齡：{}", p.age()); // OK，透過方法存取
    println!("{}", p.age);         // 編譯錯誤！age 欄位是私有的
}
```

這個設計很重要——它讓你可以控制哪些欄位要暴露、哪些要隱藏。如果 struct 有任何私有欄位，外部就無法直接用 StructName { ... } 建構，必須透過你提供的建構函數。

tuple struct 也一樣——欄位預設是私有的，要個別加 pub：

```
mod geometry {
    pub struct Point(pub f64, pub f64); // 兩個欄位都公開
    pub struct Id(u64);                 // 欄位是私有的！
}

fn main() {
    let p = geometry::Point(1.0, 2.0); // OK，欄位都是 pub
    println!("x = {}", p.0);

    let id = geometry::Id(42); // 編譯錯誤！Id 的欄位是私有的
}
```

7.4.2.4 pub enum —— variants 自動公開

enum 跟 struct 不一樣：只要 enum 本身是 pub，所有 variants 都**自動公開**。

```
mod status {
    pub enum Color {
        Red,
        Green,
        Blue,
    }
}

fn main() {
    let c = status::Color::Red; // 所有 variant 都可用
    match c {
        status::Color::Red => println!("紅色"),
        status::Color::Green => println!("綠色"),
        status::Color::Blue => println!("藍色"),
    }
}
```

這很合理——如果你公開了一個 enum 但藏了某些 variant，別人根本沒辦法正確 match，那還不如

不公開。

7.4.2.5 `pub trait` 和 `impl`

`trait` 加 `pub` 後，裡面的 `fn` **不用也不能**個別加 `pub`——它們的可見性自動跟著 `trait` 走。如果 `trait` 是公開的，裡面的 `fn` 就是公開的；如果 `trait` 是私有的，裡面的 `fn` 就是私有的。這很合理：`trait` 是一個「契約」，如果你公開了這個契約，契約裡的所有條款當然也要公開，不然別人怎麼實作？

```
mod animal {
    pub trait Speak {
        fn speak(&self); // 不用加 pub，跟著 trait 走
    }

    pub struct Dog;

    impl Speak for Dog {
        fn speak(&self) {
            println!("汪！");
        }
    }
}

fn main() {
    use animal::Speak; // trait 要在作用域內才能呼叫它的方法
    let d = animal::Dog;
    d.speak();
}
```

注意 `use animal::Speak;` 這行——即使 `Dog` 已經實作了 `Speak`，你還是要把 `Speak trait` 引入作用域才能呼叫它的方法。如果拿掉這行，`d.speak()` 會編譯錯誤。這是 Rust 的規則：**使用 `.method()` 語法時，提供該方法的 `trait` 必須在作用域內。**

```
mod animal {
    pub trait Speak {
        fn speak(&self); // 不用加 pub，跟著 trait 走
    }

    pub struct Dog;

    impl Speak for Dog {
        fn speak(&self) {
            println!("汪！");
        }
    }
}

fn main() {
    // 沒有 use animal::Speak;
    let d = animal::Dog;
    d.speak(); // 編譯錯誤！Speak 不在作用域內
}
```

`impl` 區塊本身**不需要也不能加 `pub`**。對於 `impl Type`（不是 `impl Trait for Type`），裡面的 `fn` 各自用 `pub` 控制可見性：

```rust
mod shapes {
    pub struct Circle {
        pub radius: f64,
    }

    impl Circle {
        pub fn area(&self) -> f64 {
            std::f64::consts::PI * self.radius * self.radius
        }

        // 這是私有方法，只有 mod 內部能用
        fn internal_check(&self) -> bool {
            self.radius > 0.0
        }
    }
}
```

7.4.2.6 pub(crate)、pub(super)、pub(in path)

有時候你不想完全公開，但又想讓 crate 內部的其他 mod 使用。Rust 提供了精細的控制：

- pub(crate)：整個 crate 內部可見，但外部（別的 crate）看不到。
- pub(super)：在父 mod 的範圍內可見。
- pub(in crate::some::path)：在指定的祖先 mod 範圍內可見——最精細的控制。

```rust
mod database {
    // 整個 crate 內部都能呼叫，但如果這是 library，
    // 使用你 library 的人看不到這個函數
    pub(crate) fn connect() -> String {
        String::from("connected")
    }

    // queries 是私有的：database 可以存取，但 main 不行。
    mod queries {
        // pub(super) 讓父 mod database 可以呼叫這個函數。
        pub(super) fn raw_query() -> String {
            String::from("SELECT * FROM users")
        }
    }

    pub(crate) fn safe_query() -> String {
        let raw = queries::raw_query(); // OK：database 是 queries 的父 mod
        format!("SAFE: {}", raw)
    }
}

// pub(in path) 的例子
mod app {
    pub mod api {
        pub mod internal {
            // 在 app::api 的範圍內可見。
            pub(in crate::app::api) fn secret_key() -> &'static str {
                "super-secret"
            }
        }
```

```
        pub fn get_key() -> &'static str {
            internal::secret_key() // OK，我們在 app::api 裡
        }
    }
}

// app::api::internal::secret_key() 在這裡看不到
// 因為 pub(in crate::app::api) 限制了只能在 app::api 的範圍內存取

// 注意：pub(in path) 必須指定「包含你的」mod
//  （從你往外數的某一層），不能是無關的路徑：
//      pub(in crate::some_unrelated_mod) fn foo() {}
// 編譯器會報錯；你不能對不包含你的 mod 開放可見性。

fn main() {
    let conn = database::connect(); // OK，我們在同一個 crate
    let q = database::safe_query(); // OK，pub(crate)
    println!("{}, {}", conn, q);
    database::queries::raw_query(); // 編譯錯誤！queries 是私有的
}
```

7.4.2.7　你公開的這些東西，合起來就是你的 API

你用 pub 開放出去的那組東西——函數、型別、方法、trait——合起來就是這個 crate 的 **API**。API（application programming interface，應用程式介面）指的就是「**一段程式對外公開、讓別人呼叫的介面**」：別人只看得到、也只該依賴你公開的這一面，至於你藏在 pub 後面那些私有的實作細節，他們碰不到，你之後也可以隨意改。從第 1 章到現在，每次寫 `String::new()`、`vec.push(x)`、`iter.map(...)`，你都是在呼叫標準庫的 API——標準庫把這些函數和方法標成 pub 給你用，內部怎麼實作則完全藏起來。差別只在於：之前你是 API 的**使用者**，而從這一章開始，你也成了 API 的**設計者**。

7.4.3　重點整理

- Rust **預設一切私有**，必須明確加 pub 才公開。
- `pub struct` 只公開型別名稱，每個欄位需要**個別**加 pub（tuple struct 也一樣）。
- 有私有欄位的 struct 無法從外部直接建構，必須提供建構函數。
- `pub enum` 的所有 variants **自動公開**。
- `impl Trait for T` 裡的 fn 可見性跟著 trait 走，不加 pub；`impl T` 裡的 fn 各自用 pub 控制。
- 使用 `.method()` 語法時，提供該方法的 trait 必須在作用域內。
- `pub(crate)`：crate 內部可見，外部不可見。
- `pub(super)`：在父 mod 的範圍內可見。
- `pub(in path)`：在指定的祖先 mod 範圍內可見。
- 你公開的所有 pub 的東西**合起來就是你的 API**，沒公開的是實作細節；標準庫本身就是一套 API，差別只在這章中你從 API 的「使用者」變成了「設計者」。

7.5 use

7.5.1 本集目標

學會用 use 簡化路徑，理解 Rust 的路徑解析規則和各種匯入方式。

7.5.2 概念說明

在前面我們已經初步接觸過 use，這裡要把所有用法和路徑規則講完整。

7.5.2.1 為什麼需要 use

每次呼叫都寫完整路徑很累：

```
    let sum = crate::math::basic::add(1, 2);
    let diff = crate::math::basic::subtract(5, 3);
```

用 use 把路徑帶進來，之後就能直接用短名稱：

```
use crate::math::basic::add;
use crate::math::basic::subtract;

fn main() {
    let sum = add(1, 2);
    let diff = subtract(5, 3);
}
```

7.5.2.2 絕對路徑 vs 相對路徑

Rust 有兩種路徑起點：

絕對路徑——從 crate root 開始：

```
use crate::math::add; // 自己這個 crate 裡的 math mod
```

相對路徑——從當前 mod 的位置開始：

```
use math::add; // 當前 mod 底下的 math 子 mod
```

7.5.2.3 外部 crate 的路徑

在 Cargo.toml 加了外部 crate 後，直接用 crate 名稱作為路徑開頭：

```
use std::collections::HashMap;
use rand::RngExt;
```

std 是 Rust 的**標準庫（standard library）**——Rust 內建的一組工具，包含我們已經用過的 Vec、String、Option、Result、println! 等等，以及更多像是檔案操作、網路、集合等功能。你不需要在 Cargo.toml 加 dependency 就能用它，因為每個 Rust 程式都會自動連結 std。使用時路徑寫法跟外部 crate 一樣——std::collections::HashMap、std::fmt::Display 等。不只 std 會被自動連結，std 中的 **prelude** 更會被自動引入——也就是說，Vec、String、Option、Result、Clone、Copy 等最常用的型別和 trait，不用寫 use 就能直接用。這就是為什麼我們在最前面好幾章沒寫 use 也能用這些東西。

如果你想明確強調「這是外部 crate」，可以用 :: 開頭：

```
use ::rand::RngExt; // 明確表示 rand 是外部 crate，不是本地 mod
```

這在你的 crate 裡也有一個叫 rand 的 mod 時特別有用，可以避免歧義。

7.5.2.4 super:: 和 self::

- super::：往上一層，指向**父 mod**。
- self::：指向**當前 mod**（通常省略，但有時在 use 中有用）。

```
mod outer {
    pub fn greet() -> String {
        String::from("Hello from outer")
    }

    pub mod inner {
        pub fn call_parent() -> String {
            super::greet() // 呼叫父 mod 的 greet
        }
    }
}
```

7.5.2.5 一次 use 多個東西

匯入同一個路徑底下的多個東西，可以用大括號合併：

```
use std::io::{self, Read, Write};
// 等同於：
// use std::io;
// use std::io::Read;
// use std::io::Write;
```

self 在這裡代表 std::io 本身，所以你既匯入了 io 這個 mod，也匯入了裡面的 Read 和 Write。

7.5.2.6 use ... as（別名）

如果兩個不同地方有同名的東西，可以用 as 取別名：

```
use std::fmt::Result as FmtResult;
use std::io::Result as IoResult;

fn format_something() -> FmtResult {
    Ok(())
}

fn read_something() -> IoResult<()> {
    Ok(())
}
```

7.5.2.7 use 的名字衝突

如果你 use 了兩個同名的東西到同一個作用域，Rust 會直接報錯：

```
mod a {
    pub fn hello() -> &'static str { "from a" }
}

mod b {
    pub fn hello() -> &'static str { "from b" }
```

```
}

use a::hello;
use b::hello; // 編譯錯誤！hello 已經被定義了
```

這時候就用 as 取別名來解決。

但如果是**不同作用域**，內層的 use 會遮蔽（shadow）外層的——就像 let 的 shadowing：

```
mod a {
    pub fn hello() -> &'static str { "from a" }
}

mod b {
    pub fn hello() -> &'static str { "from b" }
}

use a::hello;

fn main() {
    println!("{}", hello());     // "from a"

    {
        use b::hello;            // 在這個作用域裡 shadow 了外面的 hello
        println!("{}", hello()); // "from b"
    }

    println!("{}", hello());     // "from a"（回到外層）
}
```

7.5.2.8 glob import（星號匯入）

* 會把目前位置能從該 mod 存取的名稱全部帶進來：

```
use std::collections::*; // HashMap, HashSet, BTreeMap... 全部可用
```

一般不推薦在正式程式碼裡用，因為不清楚到底帶了什麼進來，容易衝突。但在**測試**裡很常見——use super::*; 可以把父 mod 的所有東西帶進測試 mod。下一集我們會教怎麼用 cargo test 寫測試，到時候就會看到這個用法。

7.5.2.9 use enum variant

use 不只能匯入 mod 底下的東西，也能匯入 enum 的 variant：

```
use std::cmp::Ordering::{Less, Equal, Greater};

fn compare(a: i32, b: i32) {
    match a.cmp(&b) {
        Less => println!("小於"),
        Equal => println!("相等"),
        Greater => println!("大於"),
    }
}
```

不用每次都寫 Ordering::Less，直接用 Less 就好。這在 match 很多 variant 的時候特別方便。

7.5.3 範例程式碼

```rust
mod math {
    pub mod basic {
        pub fn add(a: i32, b: i32) -> i32 {
            a + b
        }

        pub fn subtract(a: i32, b: i32) -> i32 {
            a - b
        }
    }

    pub mod advanced {
        pub fn power(base: i32, exp: u32) -> i32 {
            let mut result = 1;
            for _ in 0..exp {
                result *= base;
            }
            result
        }

        pub fn factorial(n: u64) -> u64 {
            let mut result: u64 = 1;
            for i in 1..=n {
                result *= i;
            }
            result
        }
    }
}

// 各種 use 的方式
use math::basic::add;
use math::basic::subtract;
use math::advanced::{power, factorial};

fn main() {
    println!("3 + 5 = {}", add(3, 5));
    println!("10 - 4 = {}", subtract(10, 4));
    println!("2 ^ 10 = {}", power(2, 10));
    println!("10! = {}", factorial(10));
}
```

7.5.4 重點整理

- use 將路徑帶入作用域，讓你不必每次寫完整路徑。
- 絕對路徑用 crate:: 開頭，相對路徑從當前 mod 位置開始。
- 外部 crate 直接用名稱開頭；加 :: 前綴可以明確標記為外部 crate。
- std 是標準庫，不用加 dependency 就能用，prelude 也在裡面。
- super:: 指向父 mod，self:: 指向當前 mod。
- use a::b::{self, X, Y}; 一次 use 多個東西。
- use X as Alias; 取別名，解決名字衝突。

- 同作用域 use 同名會報錯；不同作用域會 shadow（內層遮蔽外層）。
- use something::*; 星號匯入——測試裡常用，正式程式碼少用。
- enum 的 variant 也可以被 use。

7.6 cargo test

7.6.1 本集目標

學會用 #[test] 寫測試、用 assert! 系列巨集驗證結果、用 cargo test 跑測試。

7.6.2 概念說明

7.6.2.1 為什麼要寫測試？

程式碼寫完之後，你怎麼確定它是對的？手動跑一遍？那下次改了程式碼又要再跑一遍。**自動化測試**讓你寫一次，之後隨時都能驗證——一個指令就知道有沒有東西壞掉。

7.6.2.2 最簡單的測試

在函數上面加 #[test]，它就變成測試函數：

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}
```

跑 cargo test，Rust 會自動找出所有標了 #[test] 的函數並執行它們。如果測試函數 panic 了，那個測試就算失敗。

7.6.2.3 assert 系列巨集

- assert!(condition) — 如果 condition 是 false，程式 panic。
- assert_eq!(left, right) — 如果 left != right，程式 panic。
- assert_ne!(left, right) — 如果 left == right，程式 panic。

assert_eq! 和 assert_ne! 在失敗時會印出兩個值的 Debug 格式，方便你看到底哪裡不對。

assert! 系列不只能用在測試裡——你也可以在普通程式碼裡用它們來檢查條件。但要注意：assert! 在 debug 和 release 模式下**都會執行**，即使是正式發布的程式，條件不成立一樣會 panic。如果你只想在開發階段檢查、正式發布時自動移除，可以用 debug_assert!、debug_assert_eq!、debug_assert_ne!——它們在 release 模式下會被編譯器完全忽略。

不過在**測試**裡面，直接用 assert! 系列就好——測試本來就預設用 debug build 跑。

7.6.2.4 測試預期中的 panic

有時候你想反過來確認某段程式碼**會** panic——比如存取超出範圍的索引。這時候用 #[should_panic]：

```
#[test]
#[should_panic]
fn test_out_of_bounds() {
    let v = vec![1, 2, 3];
    let _ = v[10]; // 這裡會 panic
}
```

如果函數 panic 了，測試通過；如果函數**沒有** panic，測試反而失敗。

你還可以用 `expected` 參數指定 panic 訊息必須包含什麼字串，確保 panic 的原因是對的：

```
#[test]
#[should_panic(expected = "index out of bounds")]
fn test_out_of_bounds_message() {
    let v = vec![1, 2, 3];
    let _ = v[10];
}
```

7.6.2.5 測試 mod 的慣用結構

上一集學了 `use super::*;`——測試最常這樣用。慣例是在檔案底部加一個測試 `mod`：

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}

fn multiply(a: i32, b: i32) -> i32 {
    a * b
}

#[cfg(test)]
mod tests {
    use super::*; // 把父 mod 的所有東西引進來

    #[test]
    fn test_add() {
        assert_eq!(add(2, 3), 5);
    }

    #[test]
    fn test_multiply() {
        assert_eq!(multiply(3, 4), 12);
    }
}
```

幾個重點：

- `#[cfg(test)]` 告訴編譯器：這個 `mod` **只在跑測試時才編譯**。正式發布的程式不會包含測試程式碼。
- `mod tests` 是一個普通的 `mod`，只是慣例叫 `tests`。
- `use super::*;` 把父 `mod`（也就是這個檔案的最外層）的所有東西引進來，這樣測試裡就能直接呼叫 `add`、`multiply` 等函數。

7.6.2.6 `cargo test`

```
cargo test
```

這個指令會：

1. 編譯你的程式碼（包含測試）。
2. 執行所有 `#[test]` 函數。
3. 報告哪些通過、哪些失敗。

7.6.2.7 測試私有函數

因為 mod tests 是外層 mod 的子 mod，Rust 的隱私規則允許它存取父 mod 宣告的私有項目，所以測試可以直接測試**私有函數**，不需要 pub。

7.6.3 範例程式碼

```
fn is_even(n: i32) -> bool {
    n % 2 == 0
}

fn abs(n: i32) -> i32 {
    if n >= 0 { n } else { -n }
}

fn clamp(value: i32, min: i32, max: i32) -> i32 {
    if value < min {
        min
    } else if value > max {
        max
    } else {
        value
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_is_even() {
        assert!(is_even(4));
        assert!(!is_even(7));
        assert!(is_even(0));
    }

    #[test]
    fn test_abs() {
        assert_eq!(abs(5), 5);
        assert_eq!(abs(-3), 3);
        assert_eq!(abs(0), 0);
    }

    #[test]
    fn test_clamp() {
        assert_eq!(clamp(5, 0, 10), 5);   // 在範圍內，不變
        assert_eq!(clamp(-3, 0, 10), 0);  // 低於下限，變成 min
        assert_eq!(clamp(15, 0, 10), 10); // 超過上限，變成 max
    }

    #[test]
    fn test_not_equal() {
        assert_ne!(abs(-5), -5); // abs(-5) 應該是 5，不是 -5
    }
```

```
    // 測試預期中的 panic
    #[test]
    #[should_panic(expected = "already borrowed")]
    fn test_refcell_double_borrow() {
        use std::cell::RefCell;
        let cell = RefCell::new(42);
        let _r = cell.borrow();
        let _w = cell.borrow_mut(); // 已經有不可變借用，這裡會 panic
    }
}

fn main() {
    // main 裡可以不用寫什麼——測試透過 cargo test 跑
    println!("用 cargo test 來跑測試！");
}
```

7.6.4 重點整理

- `#[test]` 標記測試函數，`cargo test` 自動找到並執行所有測試。
- `assert!(condition)`、`assert_eq!(a, b)`、`assert_ne!(a, b)` 驗證結果（debug 和 release 都會執行）。
- `debug_assert!`、`debug_assert_eq!`、`debug_assert_ne!` 只在 debug 模式執行，release 時會被忽略。
- `#[should_panic]` 測試預期中的 panic；加上 `expected = "..."` 可以確認 panic 訊息。
- `#[cfg(test)]` 讓測試 `mod` 只在測試時編譯。
- `use super::*;` 引入父 `mod` 的所有東西——測試最常用的寫法。
- 測試可以直接測試私有函數（因為測試 `mod` 是子 `mod`）。

7.7 `pub use`

7.7.1 本集目標

學會用 `pub use` 重新匯出（re-export）內部項目，讓使用者不需要知道你的 `mod` 結構。

7.7.2 概念說明

假設你寫了一個 library，內部結構長這樣：

```
src/
├── lib.rs
├── math.rs
└── math/
    ├── basic.rs
    └── advanced.rs
```

如果不做任何處理，使用你 library 的人得寫：

```
use your_crate::math::basic::add;
use your_crate::math::advanced::power;
```

這很麻煩——使用者根本不在意你內部怎麼分資料夾，他只想用 `add` 和 `power`。

7.7.2.1 pub use 的魔法

pub use 把內部的東西「重新匯出（re-export）」到當前 mod，讓外部可以用更短的路徑存取：

```
// lib.rs
mod math;

// 重新匯出，讓使用者不需要知道 math::basic:: 的路徑
pub use math::basic::add;
pub use math::advanced::power;
```

現在使用你 library 的人只需要：

```
use your_crate::add;
use your_crate::power;
```

乾淨多了。

注意：pub use 只能匯出**本來就是 pub 的東西**。如果你試圖 pub use 一個 private 的 item，編譯器會報錯——你不能把別人藏起來的東西公開出去。

7.7.2.2 re-export 其他 crate 的東西

pub use 不只能匯出自己 mod 的內容，也能匯出**其他 crate** 的東西：

```
// lib.rs
pub use rand::RngExt; // 使用者 use your_crate::RngExt 就好，不用自己加 rand 依賴
```

這在 library 設計裡很常見——你的 library 依賴了某個 crate，但你想讓使用者透過你的 crate 就能用到那些型別，不用自己在 Cargo.toml 加依賴。

7.7.2.3 分層 re-export

你也可以在中間層的 mod 做 re-export，建立更有層次的公開 library：

```
// math.rs
pub mod basic;
pub mod advanced;

// 把常用的函數提升到 math 層級
pub use basic::add;
pub use basic::subtract;
pub use advanced::power;
```

這樣外部可以用 your_crate::math::add，不需要知道 basic 這一層。

7.7.2.4 實際案例

很多知名的 Rust library 都大量使用 re-export。比如你寫 use std::io::Read;，其實 Read 可能定義在更深層的地方，只是被 re-export 到 std::io 了。

7.7.3 範例程式碼

```
mod shapes {
    pub mod circle {
        pub struct Circle {
            pub radius: f64,
        }
```

```
        impl Circle {
            pub fn new(radius: f64) -> Circle {
                Circle { radius }
            }

            pub fn area(&self) -> f64 {
                std::f64::consts::PI * self.radius * self.radius
            }
        }
    }

    pub mod rectangle {
        pub struct Rectangle {
            pub width: f64,
            pub height: f64,
        }

        impl Rectangle {
            pub fn new(width: f64, height: f64) -> Rectangle {
                Rectangle { width, height }
            }

            pub fn area(&self) -> f64 {
                self.width * self.height
            }
        }
    }

    // 重新匯出：使用者不需要知道 circle 和 rectangle 這兩個子 mod
    pub use circle::Circle;
    pub use rectangle::Rectangle;
}

// 直接從 shapes 拿，不需要 shapes::circle::Circle
use shapes::{Circle, Rectangle};

fn main() {
    let c = Circle::new(5.0);
    println!("圓形面積：{}", c.area());

    let r = Rectangle::new(4.0, 6.0);
    println!("長方形面積：{}", r.area());
}
```

7.7.4 重點整理

- `pub use path::Item;` 把內部的東西重新匯出，讓外部用更短的路徑存取。
- 可以匯出自己 `mod` 的內容，也可以匯出其他 `crate` 的東西。
- library 的 `lib.rs` 常用 `pub use` 把重要型別提升到 `crate` 頂層。

7.8 orphan rule

7.8.1 本集目標

理解 Rust 的 orphan rule（孤兒規則），以及當你想為外部型別實作外部 `trait` 時該怎麼辦。

7.8.2 概念說明

在第 5 章我們學過 `trait`——你可以為自己的型別實作任何 `trait`。但你有沒有試過這樣：

```
use std::fmt;

impl fmt::Display for Vec<i32> {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "my vec")
    }
}
```

編譯器會直接拒絕你。為什麼？

7.8.2.1 orphan rule（孤兒規則）

Rust 有一條規則：

> **要 `impl` 一個 `trait`，`trait` 或型別至少有一個必須是你這個 `crate` 定義的。**

換句話說：**`trait` 是你的，或型別是你的**，至少要符合一個。

上面的例子裡，`Display` 是標準庫定義的，`Vec<i32>` 也是——兩個都不是你的，所以不行。

7.8.2.2 為什麼要有這個限制

想像一下如果沒有 orphan rule：

- `crate A` 為 `Vec<i32>` 實作了 `Display`，印出 `[1, 2, 3]`。
- `crate B` 也為 `Vec<i32>` 實作了 `Display`，印出 `1 | 2 | 3`。
- 你的程式同時用了 `A` 和 `B`……編譯器要用哪一個？

這就是衝突。orphan rule 從根本上避免了這個問題。

7.8.2.3 合法的情況

以下這些都是合法的：

```
// 情況 1：你的型別 + 外部 trait
struct MyPoint {
    x: f64,
    y: f64,
}

impl std::fmt::Display for MyPoint {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}

// 情況 2：外部型別 + 你的 trait
trait Describable {
```

```
    fn describe(&self) -> String;
}

impl Describable for Vec<i32> {
    fn describe(&self) -> String {
        format!("一個有 {} 個元素的 Vec", self.len())
    }
}
```

7.8.2.4　newtype pattern（繞過限制的方法）

如果你真的需要為外部型別實作外部 `trait`，可以用 **newtype pattern**——建立一個 tuple `struct` 把外部型別包起來：

```
use std::fmt;

struct MyVec(Vec<i32>);

impl fmt::Display for MyVec {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        let items: Vec<String> = self.0.iter()
            .map(|x| x.to_string())
            .collect();
        write!(f, "[{}]", items.join(", "))
    }
}
```

`MyVec` 是你定義的型別，所以可以為它實作 `Display`。`self.0` 存取內部的 `Vec<i32>`。

7.8.3　範例程式碼

```
use std::fmt;

// newtype pattern：用自己的 struct 包住外部型別
struct Scores(Vec<i32>);

impl Scores {
    fn new() -> Scores {
        Scores(Vec::new())
    }

    fn add(&mut self, score: i32) {
        self.0.push(score);
    }

    fn total(&self) -> i32 {
        self.0.iter().sum()
    }
}

// 現在可以為「你的型別」實作 Display
impl fmt::Display for Scores {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        let items: Vec<String> = self.0.iter()
            .map(|x| x.to_string())
```

```
            .collect();
        write!(f, "成績：[{}]，總分：{}", items.join(", "), self.total())
    }
}

fn main() {
    let mut scores = Scores::new();
    scores.add(85);
    scores.add(92);
    scores.add(78);
    scores.add(95);

    // 因為實作了 Display，可以直接 println
    println!("{}", scores);
}
```

7.8.4 多參數 trait 的情況

上面講的規則是最簡單的版本。對於多參數 trait（像第 5 章學的 From<T>），規則其實更複雜。簡單來說：

```
// OK：你的型別出現在參數裡
impl From<MyType> for String { ... }

// 不行：兩邊都是外部的
impl From<String> for Vec<i32> { ... }
```

完整的規則涉及「covered type parameter」等概念，超出本教學的範圍。有興趣可以參考官方文件。

7.8.5 重點整理

- **orphan rule**：要 impl trait，trait 或型別至少有一個必須是你的 crate 定義的。
- 「你的型別 + 外部 trait」✓ 合法。
- 「外部型別 + 你的 trait」✓ 合法。
- 「外部型別 + 外部 trait」✗ 不合法。
- 這個規則是為了防止不同 crate 之間的 impl 衝突。
- **newtype pattern**：用 struct MyWrapper(OriginalType) 把外部型別包起來，就變成你的型別了。
- 多參數 trait 的 orphan rule 遠比上面講的更複雜，詳見官方文件。

7.9 文件註解

7.9.1 本集目標

學會撰寫文件註解，理解文件範例就是測試（doctest），並用 cargo doc 產生專業的 HTML 文件。

7.9.2 概念說明

Rust 把文件當作語言的一等公民——不是用外部工具硬擠出來的，而是內建在語法裡的。更厲害的是：文件裡的範例程式碼會被 cargo test 當成測試執行，所以 Rust 的文件範例永遠不會悄悄過期。

7.9.2.1 /// 項目文件註解

三個斜線 /// 是用來為**接下來的項目**（函數、struct、enum、trait 等）寫文件：

```rust
/// 計算兩個整數的最大公因數。
///
/// 使用歐幾里得演算法，效率為 O(log(min(a, b)))。
///
/// # Examples
///
/// ```
/// use my_math_lib::gcd;
///
/// let result = gcd(12, 8);
/// assert_eq!(result, 4);
/// ```
pub fn gcd(mut a: u64, mut b: u64) -> u64 {
    while b != 0 {
        let temp = b;
        b = a % b;
        a = temp;
    }
    a
}
```

/// 裡面支援完整的 **Markdown 語法**——標題、粗體、程式碼區塊、列表，全部都能用。

7.9.2.2 //! mod/crate 層級文件

兩個斜線加驚嘆號 //! 是為**包含它的項目**寫文件，通常放在檔案最頂端：

```rust
//! # Math Library
//!
//! 這個 library 提供基本的數學運算函數。
//!
//! ## 功能
//!
//! - 基本算術運算
//! - 最大公因數計算
//! - 次方運算

pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

放在 lib.rs 頂端就是整個 crate 的文件，放在某個 mod 檔案頂端就是那個 mod 的文件。

7.9.2.3 常用的文件段落

Rust 社群有一些約定俗成的文件段落名稱：

- # Examples — 使用範例（最重要的一個！）
- # Panics — 什麼情況下會 panic。
- # Errors — 如果回傳 Result，什麼情況下會是 Err。

7.9.2.4 文件範例就是測試（doctest）

重點來了。`# Examples` 裡的程式碼區塊不只是給人看的——**`cargo test` 會把每一個文件範例抽出來、編譯、執行**，這叫做 **doctest**。第 6 集學的 `cargo test` 其實除了跑 `#[test]` 函數之外，也會跑所有的 doctest。

每個文件範例會被當成一個**獨立的小程式**來編譯——它在你的 library **外面**，就像一個使用你 library 的人寫的程式。所以範例裡必須寫 `use my_math_lib::gcd;`，就像真正的使用者一樣。忘記寫 `use`，doctest 會編譯失敗——而**編譯失敗也算測試失敗**。順帶一提，範例裡不需要寫 `fn main()`，rustdoc 會自動幫你包一層。

這個設計帶來一個很美好的結果：**範例永遠是對的**。如果你改了函數的名字或簽名，卻忘了改文件裡的範例，`cargo test` 立刻就跳錯誤給你看。在很多語言裡，文件範例會隨著程式碼演進而悄悄過期；在 Rust，過期的範例會直接擋住你的測試。

一個要注意的地方：只有 **library crate** 的 doctest 會執行。binary crate 裡的文件註解一樣能產生文件，但裡面的範例**不會**被當成測試跑。

7.9.2.5 `cargo doc`

寫好文件註解後，一行指令就能產生漂亮的 HTML 文件：

```
cargo doc --open
```

這會：

1. 編譯你的 crate（不執行）。
2. 從**公開項目**的 `///` 和 `//!` 產生 HTML 文件。
3. 自動在瀏覽器打開。

生成的文件就跟你在 docs.rs 上看到的一模一樣。

7.9.3 範例程式碼

換一個完整的例子。假設 `Cargo.toml` 裡 `[package]` 的 `name` 是 `temperature`，以下是 `src/lib.rs` 的內容：

```rust
//! # 溫度轉換工具
//!
//! 提供攝氏和華氏之間的轉換函數。

/// 攝氏轉華氏。
///
/// # 公式
///
/// `F = C × 9/5 + 32`
///
/// # Examples
///
/// ```
/// use temperature::celsius_to_fahrenheit;
///
/// let f = celsius_to_fahrenheit(100.0);
/// assert!((f - 212.0).abs() < 0.001);
/// ```
pub fn celsius_to_fahrenheit(c: f64) -> f64 {
```

```
    c * 9.0 / 5.0 + 32.0
}

/// 華氏轉攝氏。
///
/// # 公式
///
/// `C = (F - 32) × 5/9`
///
/// # Examples
///
/// ```
/// use temperature::fahrenheit_to_celsius;
///
/// let c = fahrenheit_to_celsius(32.0);
/// assert!((c - 0.0).abs() < 0.001);
/// ```
pub fn fahrenheit_to_celsius(f: f64) -> f64 {
    (f - 32.0) * 5.0 / 9.0
}

/// 溫度的表示方式。
///
/// 支援攝氏和華氏兩種單位。
pub enum Temperature {
    /// 攝氏溫度
    Celsius(f64),
    /// 華氏溫度
    Fahrenheit(f64),
}

impl Temperature {
    /// 將任何溫度轉換為攝氏。
    ///
    /// # Examples
    ///
    /// ```
    /// use temperature::Temperature;
    ///
    /// let body = Temperature::Fahrenheit(98.6);
    /// assert!((body.to_celsius() - 37.0).abs() < 0.001);
    /// ```
    pub fn to_celsius(&self) -> f64 {
        match self {
            Temperature::Celsius(c) => *c,
            Temperature::Fahrenheit(f) => fahrenheit_to_celsius(*f),
        }
    }

    /// 將任何溫度轉換為華氏。
    pub fn to_fahrenheit(&self) -> f64 {
        match self {
            Temperature::Celsius(c) => celsius_to_fahrenheit(*c),
            Temperature::Fahrenheit(f) => *f,
        }
```

```
    }
}
```

7.9.4 重點整理

- `///` 為接下來的項目（`fn`、`struct`、`enum` 等）撰寫文件。
- `//!` 為包含它的項目（`mod`、`crate`）撰寫文件，通常放在檔案最頂端。
- 文件註解支援完整的 Markdown 語法。
- `# Examples` 是最重要的文件段落——好的範例勝過千言萬語。
- **文件範例就是 doctest**：`cargo test` 會編譯並執行所有文件範例，編譯失敗或 `assert` 失敗都算測試失敗。
- doctest 以「library 使用者」的身分編譯，所以範例裡要寫 `use your_crate::...`。
- doctest 只對 library `crate` 執行。
- `cargo doc --open` 一鍵產生並打開 HTML 文件。
- 你在 docs.rs 上看到的文件，就是用同樣的機制產生的。

7.10 cargo publish

7.10.1 本集目標

學會將你的 library 發布到 crates.io，讓全世界的 Rust 開發者都能使用。

7.10.2 概念說明

到目前為止，我們學會了怎麼組織程式碼、寫文件、使用別人的 `crate`。這一集要反過來——把你自己的專案發布出去。

7.10.2.1 帳號設定

首先，你需要一個 crates.io 的帳號：

1. 到 crates.io 用 GitHub 帳號登入。
2. 到帳號設定頁面，產生一個 **API Token**。
3. 在終端機執行：

```
cargo login
```

按 enter 後，終端機會提示你貼上 token——貼上後再按 enter 就完成了。token 會被存在本機，之後 publish 時自動使用。

7.10.2.2 準備 `Cargo.toml`

發布前，`Cargo.toml` 需要補上一些必要的 metadata：

```
[package]
name = "my-awesome-lib"
version = "0.1.0"
edition = "2024"
description = "一個很棒的數學運算 library"
license = "MIT"
repository = "https://github.com/yourname/my-awesome-lib"
readme = "README.md"
```

```
keywords = ["math", "utility"]
categories = ["mathematics"]
```

根據官方文件，發布前應填寫：

- `license`（或 `license-file`）：開源授權條款（如 `MIT`、`Apache-2.0`、`MIT OR Apache-2.0`）。
- `description`：一行簡短描述
- `homepage`：專案首頁網址
- `repository`：原始碼倉庫網址
- `readme`：README 檔案路徑

另外建議但非必需：

- `keywords`：搜尋用的關鍵字（最多 5 個）
- `categories`：分類（需符合 crates.io 的分類清單）

7.10.2.3　發布前檢查

發布前可以先用 `cargo package` 檢查有沒有問題：

```
cargo package
```

這會模擬打包過程，檢查有沒有缺少必要欄位或其他問題。

7.10.2.4　發布！

一切準備好後：

```
cargo publish
```

完成！你的專案現在在 crates.io 上了，任何人都可以 `cargo add my-awesome-lib` 來使用。

7.10.2.5　版本更新流程

專案發布後，如果要更新：

1. 修改程式碼。
2. 更新 `Cargo.toml` 裡的 `version`，遵循 SemVer（語意化版本號）。
3. 再次 `cargo publish`。

SemVer 的規則：

- **1.0 之前**（`0.x.y`）：整個 API 都被視為不穩定，任何版本都可能有破壞性變更。
- **1.0 之後**：
  - bug 修復：`1.0.0` → `1.0.1`（patch）。
  - 新增功能（向下相容）：`1.0.1` → `1.1.0`（minor）。
  - 破壞性變更：`1.1.0` → `2.0.0`（major）——改第一個數字。

為什麼 SemVer 要這麼講究「破壞性變更」？因為一旦你發布出去，**你的公開 API（尤其是那些 `pub` 的東西）就不再只是你自己的事了**——別人寫的程式會 `use` 你的函數、依賴你的型別與方法宣告。這時你的公開 API 就變成一份**對使用者的承諾**：他們依賴的這一面，你不能說改就改。

這份承諾**不只包含 `pub` 的東西，文件記載的行為也可能是其中一部分**。只要這些承諾仍然成立，私有實作細節仍可自由修改。發布或更新專案前最值得問自己的一句話是：「我真的想長期維護這個 `pub` 嗎？」——公開越多，等於承諾越多，日後想改卻不弄壞別人程式的空間就越小。把不必要的東西留成私有（或 `pub(crate)`），是在替未來的自己保留改動的自由。

注意：已發布的版本**無法刪除或覆蓋**。如果發現某個版本有嚴重問題，可以用 `cargo yank` 標記它為不建議使用，但已經在用的人不會受影響：

```
cargo yank --version 0.1.0
```

7.10.2.6 發布前最好做的事

- 寫好 `README.md`（這會顯示在 crates.io 的 crate 頁面上）。
- 跑過 `cargo test` 確認所有測試通過。
- 用 `///` 寫好文件註解（上一集學的）。
- 確保有範例程式碼。
- 用 `cargo doc --open` 檢查文件看起來沒問題。

7.10.3 範例程式碼

一個準備好發布的小 library 的完整結構：

```
my-math-lib/
├── Cargo.toml
├── README.md
└── src/
    └── lib.rs
```

Cargo.toml：

```toml
[package]
name = "my-math-lib"
version = "0.1.0"
edition = "2024"
description = "Simple math utility functions"
license = "MIT"
homepage = "https://example.com/my-math-lib"
repository = "https://github.com/example/my-math-lib"
readme = "README.md"
keywords = ["math", "utility"]
categories = ["mathematics"]
```

src/lib.rs：

```rust
//! # My Math Lib
//!
//! 提供簡單好用的數學函數。

/// 計算最大公因數。
///
/// # Examples
///
/// ```
/// use my_math_lib::gcd;
///
/// assert_eq!(gcd(12, 8), 4);
/// ```
pub fn gcd(mut a: u64, mut b: u64) -> u64 {
    while b != 0 {
        let temp = b;
        b = a % b;
```

```
        a = temp;
    }
    a
}

/// 計算最小公倍數。
///
/// # Examples
///
/// ```
/// use my_math_lib::lcm;
///
/// assert_eq!(lcm(4, 6), 12);
/// ```
pub fn lcm(a: u64, b: u64) -> u64 {
    if a == 0 || b == 0 {
        return 0;
    }
    a / gcd(a, b) * b
}

/// 判斷一個數是否為質數。
///
/// # Examples
///
/// ```
/// use my_math_lib::is_prime;
///
/// assert!(is_prime(7));
/// assert!(!is_prime(4));
/// ```
pub fn is_prime(n: u64) -> bool {
    if n < 2 {
        return false;
    }
    let mut i: u64 = 2;
    while i * i <= n {
        if n % i == 0 {
            return false;
        }
        i += 1;
    }
    true
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_gcd() {
        assert_eq!(gcd(12, 8), 4);
        assert_eq!(gcd(7, 3), 1);
        assert_eq!(gcd(0, 5), 5);
    }
```

```
    #[test]
    fn test_lcm() {
        assert_eq!(lcm(4, 6), 12);
        assert_eq!(lcm(0, 5), 0);
    }

    #[test]
    fn test_is_prime() {
        assert!(!is_prime(0));
        assert!(!is_prime(1));
        assert!(is_prime(2));
        assert!(is_prime(17));
        assert!(!is_prime(15));
    }
}
```

發布指令流程：

```
cargo test       # 確認測試通過
cargo doc --open # 檢查文件
cargo package    # 模擬打包
cargo publish    # 正式發布！
```

7.10.4 重點整理

- 在 crates.io 用 GitHub 登入，產生 API token 後用 `cargo login` 設定。
- `Cargo.toml` 發布前應填寫 `license`、`description`、`homepage`、`repository`、`readme`。
- `cargo package` 可以在發布前檢查問題。
- `cargo publish` 正式發布到 crates.io。
- 更新版本時修改 `version` 欄位，遵循 SemVer（語意化版本號）。
- 你的公開 API（尤其是那些 `pub` 的東西）是**對使用者的承諾**；移除公開項目或以不相容的方式修改它，屬於破壞性變更（major）。向下相容的新增屬於 minor，但不是所有新增都向下相容。文件記載的行為也可能是承諾的一部分；只要這些承諾仍然成立，私有實作細節就可以修改。
- 已發布的版本無法刪除，只能用 `cargo yank` 標記為不建議使用。
- 發布前寫好 README、文件註解、測試，是對使用者的基本尊重。

恭喜你完成了第 7 章！🎉 到這裡為止，我們已經教完了 Rust 的主要觀念——所有權、借用、泛型、`trait`、生命週期、閉包、迭代器，以及模組系統與 Cargo 專案的建置和發布。你現在已經可以獨當一面了。如果你腦中有什麼點子，現在就是動手實作的好時機！

即便如此，Rust 還有很多獨特而強大的功能。後面的章節會繼續介紹那些還沒介紹的重要主題，希望能帶給你對 Rust 更完整、更全面的認識。

附錄一

提一些比較沒有機會在主要章節中討論到的遺珠之憾。

8.1 數字字面值格式

8.1.1 本集目標

學會用底線分隔、不同進位制、型別後綴，以及浮點數字面值的各種寫法。

> 本集是**第 1 章**的補充。

8.1.2 概念說明

第 1 章我們學了基本的數字寫法，像 `42`、`3.14`。但 Rust 的數字字面值其實有很多寫法，讓你寫出來的數字更好讀、更精確。

8.1.2.1 底線分隔符

當數字很大的時候，`1000000` 和 `1_000_000` 哪個比較好讀？Rust 允許你在數字字面值的某些位置插入底線 `_`；它們不會影響數值：

```
    let million = 1_000_000;
    let weird_but_legal = 1_00_00_00; // 合法，但別這樣寫
```

8.1.2.2 不同進位制

除了十進位，Rust 還支援三種進位制前綴：

- `0x` — 十六進位（例如 `0xff` = 255）
- `0b` — 二進位（例如 `0b1010` = 10）
- `0o` — 八進位（例如 `0o77` = 63）

這在處理位元運算、顏色值等場景特別實用。底線也能搭配進位制使用：`0b1111_0000`。

8.1.2.3 型別後綴

你可以直接在數字後面加上型別：

```
    let byte = 0xFFu8;      // 十六進位 + u8
    let big = 1_000_000i64; // 底線 + i64
    let pi = 3.14f32;       // 浮點數 + f32
```

不加後綴的話，整數預設是 `i32`，浮點數預設是 `f64`。

8.1.2.4 浮點數字面值

浮點數有幾種寫法：

```
    let a = 3.14;         // 一般小數，預設 f64
    let b = 3.14f32;      // 指定 f32
    let c = 1.0e10;       // 科學記號：1.0 × 10^10
    let d = 2.5E-3;       // 科學記號：2.5 × 10^-3 = 0.0025
    let e = 1_234.567_8; // 底線也能用在浮點數裡
```

8.1.3 範例程式碼

```
fn main() {
    // 底線分隔
    let population = 23_000_000;
    println!("台灣人口約 {} 人", population);

    // 十六進位
    let hex_color = 0xFF5733;
    println!("顏色值：{}", hex_color);

    // 二進位
    let bits = 0b1010_1100;
    println!("位元值：{}", bits);

    // 八進位
    let octal = 0o755;
    println!("八進位 0o755 = {}", octal);

    // 型別後綴
    let byte_max = 0xFFu8;
    println!("u8 最大值：{}", byte_max);

    // 浮點數
    let pi = 3.14_159_265f64;
    println!("圓周率約 {}", pi);

    // 科學記號
    let speed_of_light = 3.0e8;
    println!("光速約 {} m/s", speed_of_light);

    let tiny = 1.6e-19;
    println!("電子電荷約 {} C", tiny);
}
```

8.1.4 重點整理

- _ 可以插在數字字面值的某些位置，幫助閱讀，編譯器會忽略。
- 0x 十六進位、0b 二進位、0o 八進位。
- 型別後綴如 u8、i64、f32 可以直接接在數字後面。
- 浮點數支援科學記號（1.0e10、2.5E-3）。

8.2 && 和 || 的短路行為

8.2.1 本集目標

理解 && 和 || 不一定會執行兩邊——有時候執行完左邊就知道結果了。

本集是**第 1 章**的補充。

8.2.2 概念說明

第 1 章學了 &&（而且）和 ||（或者）。但有一個細節當時沒提：它們有**短路行為**（short-circuit evaluation）。

8.2.2.1 && 的短路

&& 的左邊如果是 `false`，右邊就不會被執行——因為不管右邊是什麼，整個結果一定是 `false`：

```
fn main() {
    let x = 0;
    // 左邊 x != 0 是 false，所以右邊 10 / x 不會被執行
    // 如果右邊被執行了，10 / 0 會 panic！
    if x != 0 && 10 / x > 2 {
        println!("大於 2");
    }
}
```

8.2.2.2 || 的短路

|| 的左邊如果是 `true`，右邊就不會被執行——因為不管右邊是什麼，整個結果一定是 `true`：

```
fn check() -> bool {
    println!("check 被呼叫了");
    true
}

fn main() {
    // 左邊已經是 true，check() 不會被呼叫
    if true || check() {
        println!("結果是 true");
    }
    // 只會印出 "結果是 true"，不會印出 "check 被呼叫了"
}
```

8.2.2.3 為什麼要知道這個

大部分時候你不需要特別在意短路行為。但當右邊的表達式有**副作用**（例如印東西、修改變數）或**可能出錯**（例如除以零）的時候，知道右邊不一定會執行就很重要了。

8.2.3 範例程式碼

```
fn is_even(n: i32) -> bool {
    println!("  檢查 {} 是不是偶數", n);
    n % 2 == 0
}

fn is_positive(n: i32) -> bool {
```

```
    println!("  檢查 {} 是不是正數", n);
    n > 0
}

fn main() {
    // &&：左邊 false 就不看右邊
    println!("--- && 短路 ---");
    let n = -3;
    if is_even(n) && is_positive(n) {
        println!("{} 是正偶數", n);
    } else {
        println!("{} 不是正偶數", n);
    }
    // is_even(-3) 回傳 false，is_positive 不會被呼叫

    // ||：左邊 true 就不看右邊
    println!("\n--- || 短路 ---");
    let n = 4;
    if is_even(n) || is_positive(n) {
        println!("{} 是偶數或正數", n);
    }
    // is_even(4) 回傳 true，is_positive 不會被呼叫

    // 實用場景：避免除以零
    println!("\n--- 實用場景 ---");
    let divisor = 0;
    if divisor != 0 && 100 / divisor > 10 {
        println!("商大於 10");
    } else {
        println!("除數是零或商不大於 10");
    }
}
```

8.2.4 重點整理

- &&：左邊是 `false` 就不看右邊，整個結果直接是 `false`。
- ||：左邊是 `true` 就不看右邊，整個結果直接是 `true`。
- 這叫短路行為（short-circuit evaluation）。
- 當右邊有副作用或可能出錯時，短路行為特別重要。

8.3 break 回傳值

8.3.1 本集目標

學會用 `break` 從 `loop` 迴圈中回傳值，把迴圈當作表達式使用。

> 本集是**第 1 章**的補充。

8.3.2 概念說明

還記得 Rust 裡「幾乎所有東西都是表達式」嗎？`loop` 迴圈也不例外——你可以透過 `break` 帶一個值出來，讓整個 `loop` 變成一個表達式。

8.3.2.1 基本語法

```
    let result = loop {
        break 42;
    };
```

這裡 `loop { break 42; }` 的型別是 `i32`，因為 `break` 帶出了 `42`。

8.3.2.2 為什麼 `while` 和 `for` 不能這樣做？

你可能會問：`while` 和 `for` 為什麼不行？

原因是：`while` 和 `for` 可以在條件變成 false 或 iterator 走完時正常結束，完全不必執行到 `break`。在這種情況下，迴圈產生的是 `()`，而不是 `break` 帶出的值。

但 `loop` 沒有讓它正常結束的條件。如果一個 `loop` 會結束，就一定是執行了 `break`；如果沒有執行到 `break`，它就會一直跑下去，不會產生結果。正因如此，`break` 帶出的值可以成為整個 `loop` 的值。

8.3.2.3 實際應用場景

最常見的用法是「在迴圈裡搜尋某個東西，找到就帶出來」：

```
    let found = loop {
        // 做一些搜尋...
        if condition {
            break some_value;
        }
    };
```

這比先宣告一個變數、在迴圈裡賦值、再 `break` 出來要簡潔得多。

8.3.3 範例程式碼

```
fn main() {
    // 基本用法：loop 回傳值
    let lucky_number = loop {
        break 7;
    };
    println!("幸運數字：{}", lucky_number);

    // 實用範例：找到第一個大於 100 的平方數
    let mut n = 1;
    let result = loop {
        let square = n * n;
        if square > 100 {
            break square;
        }
        n += 1;
    };
    println!("第一個大於 100 的平方數：{}", result);
    println!("它是 {} 的平方", n);
}
```

8.3.4 搭配標籤使用 break 回傳值

像 'search: 這樣的標籤可以放在迴圈（loop、while 或 for）或一般的 block 表達式 { ... } 前面。後者會形成 labeled block。這裡的 'search 是標籤，不是 lifetime。

break 'label value 會跳出有標籤的 loop 或 block，並讓該表達式產生 value。跳出最內層的 loop 時，可以省略標籤（寫成 break value）；在 labeled block 中則一定要寫出標籤。

```
fn main() {
    let from_loop = 'search: loop {
        loop {
            break 'search 7;
        }
    };

    let from_block = 'answer: {
        let n = 7;
        if n > 5 {
            break 'answer n * 2;
        }
        0
    };

    println!("來自 loop：{}", from_loop);
    println!("來自 block：{}", from_block);
}
```

這裡的 break 'search 7 會直接跳出兩層 loop，讓有 'search 標籤的外層 loop 產生 7。break 'answer n * 2 會讓 labeled block 產生 14。

8.3.5 重點整理

- let x = loop { break value; }; 讓 loop 成為表達式，回傳 break 帶出的值。
- while 和 for 不能用 break 回傳值，因為它們可以不經過 break 就正常結束。
- break 'label value 可以從有標籤的 loop 或 labeled block 回傳值；labeled block 一定要寫出標籤。
- 常見用途是在迴圈中搜尋，找到後用 break 帶出結果。

8.4 多行字串 & raw string literal

8.4.1 本集目標

學會在 Rust 中撰寫多行字串、行接續符號、以及不需要跳脫字元的 raw string。

> 本集是**第 1 章**的補充。

8.4.2 概念說明

寫程式的時候，我們常常需要處理多行文字、檔案路徑、或是包含特殊字元的字串。Rust 提供了幾種好用的語法來應對這些情況。

8.4.2.1 多行字串

在 Rust 裡，字串字面值可以直接跨行：

```
    let poem = "床前明月光，
疑是地上霜。";
```

換行符號會直接被包含在字串裡。

8.4.2.2 行接續符 \

如果你想把很長的字串分行寫，但**不要**換行符號出現在結果裡，可以在行尾加 \。它會吃掉換行以及下一行開頭的空白：

```
    let long = "這是一段很長的句子，\
                但其實只有一行。";
    // 結果："這是一段很長的句子，但其實只有一行。"
```

8.4.2.3 raw string literal

有時候字串裡有很多反斜線（例如 Windows 路徑），每個都要跳脫很煩。`r"..."` 語法讓你完全不需要跳脫：

```
    let path = r"C:\Users\test\documents";
    // 不需要寫成 "C:\\Users\\test\\documents"
```

8.4.2.4 包含引號的 raw string

如果 raw string 裡面需要有雙引號怎麼辦？用 `r#"..."#` 語法：

```
    let json = r#"{"name": "Andy", "age": 29}"#;
```

如果字串裡面連 `"#` 都有？那就多加幾層 `#`：

```
    let tricky = r##"這裡有 "#" 符號"##;
```

最多可以使用 255 個 `#`，只要開頭和結尾的數量一致就好。

8.4.3 範例程式碼

```
fn main() {
    // 多行字串
    let haiku = "古池や
蛙飛び込む
水の音";
    println!("俳句：\n{}", haiku);
    println!("---");

    // 行接續符：\ 吃掉換行和前導空白
    let sentence = "Rust 是一門注重安全性、\
                    效能和並行的程式語言。";
    println!("{}", sentence);
    println!("---");

    // Raw string：不處理跳脫字元
    let win_path = r"C:\Users\Andy\Desktop\project";
    println!("路徑：{}", win_path);

    // 正則表示式之類的場景也很好用
    let pattern = r"\d+\.\d+";
```

```
    println!("正則：{}", pattern);

    // 包含雙引號的 raw string
    let json = r#"{"name": "小明", "score": 95}"#;
    println!("JSON：{}", json);

    // 多層 # ——當字串裡有 "# 的時候
    let code_sample = r##"
        let s = r#"hello"#;
        println!("{}", s);
    "##;
    println!("程式碼範例：{}", code_sample);

    // raw string 也能多行
    let html = r#"
<html>
    <body>
        <h1>Hello, Rust!</h1>
    </body>
</html>
"#;
    println!("{}", html);
}
```

8.4.4 重點整理

- 字串字面值可以直接跨行，換行符號會被保留。
- 行尾加 \ 可以接續下一行，同時忽略換行和下一行的前導空白。
- `r"..."` 是 raw string，不處理任何跳脫字元（\n、\\ 等都照原樣保留）。
- `r#"..."#` 讓 raw string 裡可以包含雙引號。
- raw string 最多可以使用 255 個 #（`r##"..."##`、`r###"..."###` 等），只要開頭和結尾的數量一致。
- raw string 特別適合 Windows 路徑、正則表示式、JSON、嵌入程式碼等場景。

8.5 格式化字串進階

8.5.1 本集目標

學會 `println!` 的各種格式化技巧，包括變數捕獲簡寫、位置參數、寬度、精度控制、對齊和進位制顯示。

> 本集是**第 2 章**的補充。

8.5.2 概念說明

我們之前一直用 `println!("{}", x)` 來印東西，但其實 Rust 的格式化字串功能強大得多。這集介紹最常用的技巧，但不會涵蓋所有用法——完整的格式化語法請參考官方文件。

8.5.2.1 變數捕獲簡寫

你可以直接在 `{}` 裡寫變數名稱：

```rust
fn main() {
    let name = "Andy";
    println!("{name}"); // 等同於 println!("{}", name)
}
```

這比一直寫 {} 然後在後面對應變數方便多了，尤其是有很多變數的時候。注意只能放變數名，不能放表達式（"{x + 1}" 不行）。

8.5.2.2 位置參數

{} 預設會按照順序對應後面的參數，但你也可以在大括號裡寫數字，明確指定要用第幾個參數（從 0 開始數）：

```rust
fn main() {
    println!("{0} {1}", "哈", "囉"); // 哈 囉
    println!("{1} {0}", "哈", "囉"); // 囉 哈
}
```

同一個參數可以重複使用，不用傳兩次：

```rust
fn main() {
    println!("{0}，{0}！", "等等"); // 等等，等等！
}
```

什麼時候會用到？最常見的情況是同一個值要在字串裡出現很多次，或是想調整輸出順序但不想改參數順序的時候。

8.5.2.3 小數精度

用 :.N 控制小數點後幾位：

```rust
fn main() {
    let pi = 3.14159265;
    println!("{pi:.2}"); // 印出 3.14
}
```

8.5.2.4 寬度

用 :N 指定最小寬度——不夠寬的話會用空白補齊：

```rust
fn main() {
    let x = 42;
    println!("{x:5}"); // "   42"（寬度 5，靠右，空白補齊）
}
```

8.5.2.5 對齊

用 :>N、:<N、:^N 來明確控制靠右、靠左、置中：

```rust
fn main() {
    let name = "Andy";
    println!("[{name:>10}]"); // 靠右對齊，寬度 10
    println!("[{name:<10}]"); // 靠左對齊
    println!("[{name:^10}]"); // 置中
}
```

8.5.2.6 填充字元

預設用空白填充，你也可以指定其他字元：

```
fn main() {
    let id = 42;
    println!("{id:0>5}"); // 印出 00042（用 0 填充）
}
```

8.5.2.7 進位制顯示

用 `:b`、`:x`、`:o` 分別以二進位、十六進位、八進位顯示數字：

```
fn main() {
    let n = 255;
    println!("{n:b}"); // 11111111
    println!("{n:x}"); // ff
    println!("{n:o}"); // 377
}
```

這些格式也可以組合——例如 `{:0>8b}` 是「零填充到 8 位的二進位」。

8.5.2.8 跳脫大括號

如果你想在格式化字串裡印出 `{` 或 `}` 本身，用 `{{` 和 `}}`：

```
fn main() {
    println!("這是大括號：{{}}"); // 印出：這是大括號：{}
}
```

8.5.3 範例程式碼

```
fn main() {
    let name = "小明";
    let score = 87.5678;

    // 變數捕獲簡寫
    println!("學生：{name}");
    println!("分數：{score}");

    // 位置參數：指定順序、重複使用
    println!("{1}的分數是{0}", score, name);
    println!("{0}！{0}！{0}！", "加油");

    // 小數精度
    println!("四捨五入到兩位：{score:.2}");

    // 寬度
    println!("[{name:10}]"); // 字串預設靠左
    let x = 42;
    println!("[{x:10}]");    // 數字預設靠右

    // 對齊
    println!("[{name:>10}]"); // 靠右
    println!("[{name:<10}]"); // 靠左
    println!("[{name:^10}]"); // 置中
```

```
    // 零填充
    let id = 42;
    println!("編號：{id:0>5}");

    // 進位制顯示
    let value = 255;
    println!("十進位：{value}");
    println!("二進位：{value:b}");
    println!("十六進位：{value:x}");
    println!("八進位：{value:o}");

    // 組合技：零填充 + 靠右 + 2 位寬 + 十六進位
    let byte = 10;
    println!("0x{byte:0>2x}"); // 印出 0x0a

    // 組合技：零填充 + 靠右 + 8 位寬 + 二進位
    println!("{byte:0>8b}"); // 印出 00001010

    // 如果要印出大括號本身，用 {{ 和 }}
    println!("這是一個大括號：{{}}"); // 印出：這是一個大括號：{}
}
```

8.5.4 重點整理

- `println!("{x}")` 直接在大括號裡寫變數名，只能放變數不能放表達式。
- `{0}`、`{1}` 用數字指定第幾個參數（從 0 開始），同一個參數可以重複使用。
- `{:.2}` 控制小數點後位數。
- `{:5}` 指定最小寬度。
- `{:>10}`、`{:<10}`、`{:^10}` 分別是靠右、靠左、置中對齊。
- `{:0>5}` 用 `0` 填充到寬度 5。
- `{:b}`、`{:x}`、`{:o}` 分別用二進位、十六進位、八進位顯示。
- 格式化選項可以組合使用，例如 `{:0>8b}` 是零填充 + 靠右 + 8 位寬 + 二進位。
- 要印出 `{` 和 `}` 本身，用 `{{` 和 `}}` 跳脫。

8.6 `struct` / `enum` 放在 `fn` 裡面

8.6.1 本集目標

了解 `fn`、`struct`、`enum` 等「項目」可以定義在函數內部，以及它們與 `let` 綁定在順序上的根本差異。

> 本集是**第 3 章**的補充。

8.6.2 概念說明

你可能習慣了把 `struct` 和 `enum` 定義在 `fn main()` 的外面，但其實把它們放在裡面也完全合法：

```
fn main() {
    struct Point {
        x: i32,
        y: i32,
    }
```

```
    let p = Point { x: 1, y: 2 };
    println!("{}", p.x);
}
```

這段程式碼完全可以編譯。

8.6.2.1 限制：只在該函數內可見

放在函數內的型別定義，只有那個 fn 內看得到。外面的函數無法使用它。所以慣例上，我們還是會把型別定義放在外面——除非你確定這個型別只在一個 fn 裡面用到。

8.6.2.2 重要差異：項目不受順序限制

這裡有一個很多人不知道的重點。在 Rust 裡，**項目（items）**——包括 fn、struct、enum、trait、impl 等——**不受定義順序影響**。即使還沒定義，也可以先使用：

```
fn main() {
    let p = Point { x: 1, y: 2 }; // 先使用
    println!("{}", p.x);

    struct Point {                // 後定義
        x: i32,
        y: i32,
    }
}
```

這和 let 完全不同！let 綁定必須在使用之前出現，否則編譯器會報錯。但項目定義是「全域可見」的（在它所在的作用域內），跟你寫在哪一行無關。

8.6.2.3 為什麼會這樣？

因為項目是在編譯時期就確定的靜態定義，編譯器會先掃描所有項目，建立完整的型別資訊，然後才處理 let 等執行時期的敘述。

8.6.3 範例程式碼

```
fn main() {
    // 先呼叫，後定義——完全合法
    greet();

    // 先使用 struct，後定義
    let p = Point { x: 3.0, y: 4.0 };
    println!("座標：({}, {})", p.x, p.y);

    // 先使用 enum，後定義
    let color = Color::Red;
    describe(color);

    // 這些項目都定義在使用之後
    struct Point {
        x: f64,
        y: f64,
    }

    enum Color {
```

```
        Red,
        Green,
        Blue,
    }

    fn describe(c: Color) {
        match c {
            Color::Red => println!("紅色"),
            Color::Green => println!("綠色"),
            Color::Blue => println!("藍色"),
        }
    }

    fn greet() {
        println!("哈囉！");
    }

    // 但 let 綁定必須在使用之前！
    // 以下如果取消註解會編譯失敗：
    // println!("{}", not_yet);
    let not_yet = 42;
    println!("let 綁定必須先宣告：{}", not_yet);
}
```

8.6.4 重點整理

- `struct`、`enum`、`fn` 等項目可以合法地定義在函數內部。
- 定義在 `fn` 內的項目，只有該 `fn` 內看得到（作用域限制）。
- 一般還是把型別定義放在 `fn` 外面，除非只有單一 `fn` 使用。
- **項目不受定義順序影響**——無論在定義前或定義後都可以使用。
- **`let` 綁定必須在使用之前出現**——這是項目和 `let` 的根本差異。
- 原因：項目是編譯時期的靜態定義，編譯器會先掃描完所有項目再處理執行時期程式碼。

8.7 `struct` update syntax

8.7.1 本集目標

學會用 `..` 語法從既有的 `struct` 實例快速建立新實例，並理解 Copy 與 move 欄位的差異。

> 本集是**第 3 章**的補充。

8.7.2 概念說明

還記得建立 `struct` 的時候，每個欄位都要寫出來嗎？如果你只想改一兩個欄位，其他照舊，每次都全部寫一遍很煩。Rust 提供了 **`struct` update syntax**，用 `..` 來「填入剩下的欄位」。

8.7.2.1 基本語法

```
    let p2 = Point { x: 10, ..p1 };
```

意思是：`p2` 的 `x` 設為 `10`，其餘欄位都從 `p1` 移過來。

`..p1` 必須放在最後面，而且前面要有逗號（如果前面有其他欄位的話）。

8.7.2.2 copy 與 move 的差異

這裡有個重要的細節。`..p1` 並不是「clone 整個 struct」，而是**逐欄位**處理：

- 如果欄位的型別實作了 Copy（像 i32、f64、bool），就是 **copy**。
- 如果欄位的型別**沒有** Copy（像 String），就是 **move**。

也就是說，如果你用 `..p1` 並且 move 走了 p1 的某些非 Copy 欄位，那些欄位之後就不能再透過 p1 存取了。

8.7.2.3 搭配 Default

如果你的 struct 有實作 Default trait，可以用 `..` 搭配它的 default() 來建立「只指定幾個欄位，其他用預設值」的實例：

```
    let config = Config { debug: true, ..Config::default() };
```

這在有很多欄位的 struct 特別好用。

8.7.3 範例程式碼

```
#[derive(Debug)]
struct Config {
    width: u32,
    height: u32,
    fullscreen: bool,
    title: String,
}

impl Default for Config {
    fn default() -> Self {
        Config {
            width: 800,
            height: 600,
            fullscreen: false,
            title: String::from("My App"),
        }
    }
}

#[derive(Debug, Clone, Copy)]
struct Point {
    x: f64,
    y: f64,
}

fn main() {
    // 基本用法：只改一個欄位
    let p1 = Point { x: 1.0, y: 2.0 };
    let p2 = Point { x: 10.0, ..p1 };
    println!("p1 = {:?}", p1); // p1 還能用，因為 f64 是 Copy
    println!("p2 = {:?}", p2);

    // 搭配 Default：只指定想改的欄位
    let custom = Config {
        width: 1920,
```

```
        height: 1080,
        ..Config::default()
    };
    println!("自訂設定：{:?}", custom);

    // 全部用預設值
    let default_config = Config { ..Config::default() };
    println!("預設設定：{:?}", default_config);

    // 注意 move 語義！
    let c1 = Config {
        width: 1024,
        height: 768,
        fullscreen: true,
        title: String::from("Game"),
    };
    let c2 = Config {
        fullscreen: false,
        ..c1 // title (String) 會被 move！
    };
    // println!("{}", c1.title); // 編譯錯誤！title 已經被 move 了
    println!("c1.width = {}", c1.width);  // 但 Copy 欄位還是能用
    println!("c2 = {:?}", c2);
}
```

8.7.4 重點整理

- `let p2 = Point { x: 1, ..p1 };` 用 p1 填入 p2 剩餘的欄位。
- `..source` 必須放在最後面。
- Copy 型別的欄位會被 copy，非 Copy 型別的欄位會被 move。
- 如果所有欄位都是 Copy，原本的 `struct` 還能繼續使用。
- 如果有非 Copy 欄位被 move，原本 `struct` 的那些欄位就不能再存取。
- `..Config::default()` 很適合用在「大部分欄位用預設值，只改幾個」的場景。

8.8 `ref` pattern 與 `match` ergonomics

8.8.1 本集目標

了解 `ref` 關鍵字在模式匹配中的作用，以及為什麼在現代 Rust 中幾乎不需要手動寫 `ref`。

> 本集是**第 3 章**的補充。

8.8.2 概念說明

這集要講一個你可能在舊程式碼裡看過、但在現代 Rust 中幾乎用不到的語法：`ref`。理解它的存在和原理，有助於你讀懂別人的程式碼。

8.8.2.1 `ref` 是什麼？

在模式中，`ref` 會把綁定的變數變成一個參考，而不是取得所有權：

```
    let val = String::from("hello");
    let ref r = val; // r 的型別是 &String
```

```
    // 等同於：let r = &val;
```

你可能會想：那我直接寫 &val 不就好了？沒錯，在 let 綁定中，兩者完全等價。ref 的存在感主要在 match 裡面。

8.8.2.2 在 match 中的 ref

以前（Rust 1.26 之前），如果你想在 match 裡借用而不是 move，必須手動寫 ref：

```
fn main() {
    let opt = Some(String::from("hello"));
    match opt {
        Some(ref s) => println!("{}", s), // 借用，不 move
        None => println!("nothing"),
    }
    // opt 還能用，因為我們只是借用了裡面的值
}
```

如果不寫 ref，s 會拿走 String 的所有權，之後就不能再用 opt 了。

8.8.2.3 match ergonomics（Rust 1.26+）

從 Rust 1.26 開始，編譯器變聰明了。當你 match 一個**參考**的時候，裡面的綁定會自動變成參考：

```
fn main() {
    let opt = Some(String::from("hello"));
    match &opt {     // 注意這裡是 &opt
        Some(s) => { // s 自動是 &String，不需要寫 ref
            println!("{}", s);
        }
        None => println!("nothing"),
    }
    // opt 還能用！
}
```

這就是所謂的 **match ergonomics**：以前得自己寫 ref 才有的效果，現在編譯器看到你 match 的是一個參考（&opt），就自動幫你做到了。

8.8.2.4 所以現在還需要寫 ref 嗎？

幾乎不需要了。99% 的情況你只要 match 參考（match &value），編譯器就會自動處理。但讀舊程式碼的時候，看到 ref 至少要知道它在做什麼。

8.8.3 範例程式碼

```
fn main() {
    // ===== ref 基本用法 =====
    let name = String::from("Rust");
    let ref r = name; // r: &String
    println!("ref 綁定：{}", r);
    println!("原本還能用：{}", name);

    // ===== 舊寫法：match 中用 ref 避免 move =====
    let data = Some(String::from("重要資料"));

    match data {
```

```
        Some(ref s) => println!("舊寫法借用：{}", s),
        None => println!("空的"),
    }
    println!("data 還在：{:?}", data); // 因為用了 ref，沒有 move

    // ===== 新寫法：match ergonomics =====
    let data2 = Some(String::from("新世界"));

    match &data2 {   // match 參考
        Some(s) => { // s 自動是 &String
            println!("新寫法借用：{}", s);
        }
        None => println!("空的"),
    }
    println!("data2 還在：{:?}", data2);

    // ===== 更複雜的例子 =====
    let pairs = vec![
        (String::from("台北"), 25),
        (String::from("東京"), 10),
        (String::from("紐約"), 5),
    ];

    // match ergonomics 讓 for 迴圈中的解構也很自然
    for (city, temp) in &pairs {
        // city: &String, temp: &i32（自動借用）
        println!("{} 氣溫 {} 度", city, temp);
    }
    println!("pairs 還在，共 {} 筆", pairs.len());
}
```

8.8.4 重點整理

- `let ref x = val;` 等同於 `let x = &val;`——在 `let` 中兩者完全一樣。
- 在 `match` 中，`Some(ref x)` 會借用而不是 move 內部的值。
- **match ergonomics（Rust 1.26+）**：`match` 一個參考時，模式中的變數自動變成參考。
- 現代 Rust 幾乎不需要手動寫 `ref`，用 `match &value` 就好。
- `for (k, v) in &collection` 也受 match ergonomics 影響，`k` 和 `v` 自動是參考。
- 認識 `ref` 主要是為了讀懂舊程式碼。

8.9 `panic!` / `todo!` / `unimplemented!` / `unreachable!`

8.9.1 本集目標

認識四種會觸發 panic 的巨集，以及它們各自的使用時機。

> 本集是通用補充，不特定屬於哪一章。

8.9.2 概念說明

你可能注意到 `println!()`、`format!()` 這些名字後面都有 `!`。在 Rust 裡，名字帶 `!` 的東西叫做**巨集（macro）**——它和函數不太一樣，但目前你只需要知道怎麼用就好，巨集的原理以後會教。

這集要介紹四個常用的巨集，程式執行到它們時就會觸發 panic。panic 會中斷正常執行；在目前學過的程式中，未被處理的 panic 會讓程式結束。這四個巨集都會造成 panic，但語義不同，傳達給讀程式碼的人的訊息也不同。

8.9.2.1 panic!("訊息")

最基本的「程式出事了，現在 panic」。當你遇到無法處理的錯誤時使用：

```
    panic!("發生了不該發生的事！");
```

你可以帶格式化訊息：panic!("找不到 id: {}", id);

8.9.2.2 todo!()

「我還沒寫完，先放個佔位符」。開發中最常用，讓你先把程式架構搭好，細節之後再填：

```
fn calculate_tax(income: f64) -> f64 {
    todo!() // 之後再實作
}
```

編譯可以通過，但執行到這裡就會 panic，訊息是「not yet implemented」。

8.9.2.3 unimplemented!()

「這個功能沒有實作」。跟 todo!() 很像，但語義不同——todo!() 明確表示「之後會做」，unimplemented!() 則**不保證之後會做**。可能是不打算做，可能是目前沒需求，也可能是 trait 要求的方法但對這個型別沒意義：

```
trait Foo {
    fn bar(&self) -> u8;
    fn baz(&self);
}

struct MyStruct;

impl Foo for MyStruct {
    fn bar(&self) -> u8 {
        1 + 1
    }

    fn baz(&self) {
        // 對 MyStruct 來說 baz 沒意義，但 trait 要求必須定義
        unimplemented!()
    }
}

fn main() {}
```

8.9.2.4 unreachable!()

「這行程式碼不應該被執行到」。如果你確定某段邏輯不可能走到，用這個來標記：

```
fn main() {
    let direction = "north";
    match direction {
        "north" | "south" | "east" | "west" => println!("有效方向"),
        _ => unreachable!("方向只有四種，不可能走到這裡"),
```

```
    }
}
```

如果真的走到了，表示你的假設有誤，panic 會幫你發現這個 bug。

8.9.2.5 四者比較

- `panic!` — 出事了。用於無法處理的錯誤。
- `todo!` — 還沒寫，之後會實作。開發中的佔位符。
- `unimplemented!` — 沒有實作，不保證之後會做。可能是沒需求、可能是 `trait` 要求但沒意義。
- `unreachable!` — 不該走到這裡。標記邏輯上不可能的分支。

8.9.3 範例程式碼

```
enum Shape {
    Circle(f64),
    Rectangle(f64, f64),
    Triangle(f64, f64, f64),
}

fn area(shape: &Shape) -> f64 {
    match shape {
        Shape::Circle(r) => 3.14159 * r * r,
        Shape::Rectangle(w, h) => w * h,
        Shape::Triangle(_, _, _) => todo!("三角形面積之後再實作"),
    }
}

fn describe_score(score: u32) -> &'static str {
    match score {
        90..=100 => "優秀",
        80..=89 => "良好",
        70..=79 => "普通",
        60..=69 => "及格",
        0..=59 => "不及格",
        _ => unreachable!("分數應該在 0-100 之間"),
    }
}

trait Storage {
    fn save(&mut self, data: &str);
    fn load(&self) -> String;
}

struct LocalStorage;

impl Storage for LocalStorage {
    fn save(&mut self, data: &str) {
        println!("儲存到本地：{}", data);
    }

    fn load(&self) -> String {
        // trait 要求定義，但 LocalStorage 不需要這個功能
```

```
        unimplemented!()
    }
}

fn main() {
    // todo! ─開發中的佔位符
    let circle = Shape::Circle(5.0);
    println!("圓形面積：{}", area(&circle));

    let rect = Shape::Rectangle(3.0, 4.0);
    println!("矩形面積：{}", area(&rect));

    // 如果取消下一行的註解，會 panic 並顯示 todo! 訊息
    // let tri = Shape::Triangle(3.0, 4.0, 5.0);
    // println!("三角形面積：{}", area(&tri));

    // unreachable! ─不該走到的分支
    let grade = describe_score(85);
    println!("85 分的評等：{}", grade);

    // unimplemented! ─沒有實作的功能
    let mut storage = LocalStorage;
    storage.save("hello");
    // storage.load(); // 取消註解會 panic：not implemented

    // panic! ─觸發 panic
    // panic!("故意 panic！");
    println!("程式正常結束");
}
```

8.9.4 重點整理

- `panic!("msg")` 是最基本的 panic 方式，用於無法處理的錯誤。
- `todo!()` 是開發佔位符，明確表示「之後會實作」。
- `unimplemented!()` 表示「沒有實作」，不保證之後會做——可能是沒需求、可能是 `trait` 要求但對該型別沒意義。
- `unreachable!()` 標記邏輯上不可能到達的程式碼路徑。
- 它們都會造成 panic，差別在於傳達的**意圖**不同——選對的那個，讓程式碼更有表達力。

8.10 `let` chains

8.10.1 本集目標

認識 `let` chains——在 `if` 和 `while` 的條件裡用 && 串接多個 `let` 和布林條件。

> 本集是**第 3 章**的補充。

8.10.2 概念說明

8.10.2.1 問題：巢狀的 `if let`

第 3 章學了 `if let`。但如果你需要連續做多次模式匹配，就會變成巢狀的 `if let`：

```
enum Wrapper {
    Value(i32),
    Empty,
}

fn get_a() -> Wrapper { Wrapper::Value(10) }
fn get_b(x: i32) -> Wrapper { Wrapper::Value(x + 1) }

fn main() {
    if let Wrapper::Value(a) = get_a() {
        if a > 0 {
            if let Wrapper::Value(b) = get_b(a) {
                println!("a = {}, b = {}", a, b);
            }
        }
    }
}
```

每多一個條件就多一層縮排，程式碼越來越深。

8.10.2.2 `let` chains 攤平

你可以用 && 把多個 `let` 和布林條件串在同一個 `if` 裡：

```
enum Wrapper {
    Value(i32),
    Empty,
}

fn get_a() -> Wrapper { Wrapper::Value(10) }
fn get_b(x: i32) -> Wrapper { Wrapper::Value(x + 1) }

fn main() {
    if let Wrapper::Value(a) = get_a()
        && a > 0
        && let Wrapper::Value(b) = get_b(a)
    {
        println!("a = {}, b = {}", a, b);
    }
}
```

每個用 && 串起來的條件從左到右依序檢查。前面的 `let` 綁定的變數在後面的條件裡可以使用（像上面的 a）。只要任何一個條件不成立，後面的就不會執行——跟 && 的短路行為一樣。

8.10.2.3 `while` 裡也能用

```
while let Some(item) = next_item()
    && item.value > 0
{
    // ...
}
```

8.10.3 範例程式碼

```
enum Command {
    Run { speed: i32 },
    Stop,
}

fn get_command() -> Command {
    Command::Run { speed: 5 }
}

fn get_boost() -> Command {
    Command::Run { speed: 3 }
}

fn main() {
    // 巢狀寫法
    if let Command::Run { speed: s } = get_command() {
        if s > 0 {
            if let Command::Run { speed: boost } = get_boost() {
                println!("巢狀：速度 {} + 加速 {} = {}", s, boost, s + boost);
            }
        }
    }

    // let chains 寫法——同樣的邏輯，更扁平
    if let Command::Run { speed: s } = get_command()
        && s > 0
        && let Command::Run { speed: boost } = get_boost()
    {
        println!("扁平：速度 {} + 加速 {} = {}", s, boost, s + boost);
    }
}
```

8.10.4 重點整理

- let chains 讓你在 if 和 while 裡用 && 串接多個 let 和布林條件。
- 取代巢狀的 if let，讓程式碼更扁平。
- 前面綁定的變數後面可以使用。
- 跟 && 的短路行為一致：前面不成立就不繼續。

8.11 Rc 迴圈與 Weak

8.11.1 本集目標

理解 Rc 參考迴圈會造成記憶體洩漏，並學會用 Weak 來打破迴圈。

> 本集是**第 5 章**的補充。

8.11.2 概念說明

還記得第 5 章學的 Rc<T> 嗎？它透過參考計數來管理記憶體——每多一個 Rc 指向同一筆資料，計數就加一；每少一個就減一；歸零時釋放記憶體。

聽起來很完美，但有一個致命弱點：**參考迴圈（reference cycle）**。

8.11.2.1 什麼是參考迴圈？

想像兩個節點 A 和 B，A 持有 Rc 指向 B，B 也持有 Rc 指向 A。當外部不再持有它們的時候：

1. A 的外部 Rc 被 drop → A 的計數減一，但 B 還在指向 A → 計數不為零 → A 不釋放。
2. B 的外部 Rc 被 drop → B 的計數減一，但 A 還在指向 B → 計數不為零 → B 不釋放。

結果：A 和 B **永遠不會被釋放**，這就是記憶體洩漏。從外面看不見、卻互相撐著的環——這才是迴圈問題的本質。

洩漏麻煩的地方在於它沒有任何外顯症狀：程式不會 panic、不會編譯失敗，只是那塊記憶體再也回不來。所以我們用 Drop 讓它現形——幫節點加一個會印字的解構子，如果訊息沒印出來，就代表值沒被釋放：

```
use std::rc::Rc;
use std::cell::RefCell;

struct Node {
    name: &'static str,
    other: Option<Rc<RefCell<Node>>>,
}

impl Drop for Node {
    fn drop(&mut self) {
        println!("{} 被釋放了", self.name);
    }
}

fn main() {
    {
        let a = Rc::new(RefCell::new(Node { name: "A", other: None }));
        let b = Rc::new(RefCell::new(Node { name: "B", other: None }));

        a.borrow_mut().other = Some(b.clone());
        b.borrow_mut().other = Some(a.clone());

        println!("離開作用域前，A 的 strong count = {}", Rc::strong_count(&a));
    }

    println!("已經離開作用域了");
}
```

輸出只有兩行：

```
離開作用域前，A 的 strong count = 2
已經離開作用域了
```

`strong count = 2` 正是上面說的「B 還在指向 A」，而兩個「被釋放了」都沒有出現——a 和 b 這兩個變數雖然消失了，它們指向的節點卻還互相撐著。把 `b.borrow_mut().other = ...` 那一行註解掉再跑一次，環斷了，兩則訊息就都會出現。

8.11.2.2 Weak 是什麼

Weak<T> 是一種「弱參考」——它指向同一筆資料，但**不會增加 strong count**。

```
    let strong = Rc::new(42);
    let weak: Weak<i32> = Rc::downgrade(&strong);
```

Rc::downgrade 把 Rc 降級成 Weak。Rc 內部有**兩個**計數器：strong count 和 weak count。.clone() 增加 strong count，Rc::downgrade() 只增加 weak count。Rc 判斷「要不要釋放值」只看 strong count——strong count 歸零就釋放，不管 weak count 是多少。

因為 Weak 指向的資料可能已經被釋放了，你不能直接存取。必須先 .upgrade()：

```
use std::rc::{Rc, Weak};

fn main() {
    let strong = Rc::new(42);
    let weak: Weak<i32> = Rc::downgrade(&strong);

    match weak.upgrade() {
        Some(rc) => println!("還在：{}", rc),
        None => println!("已經被釋放了"),
    }
}
```

upgrade 回傳 Option<Rc<T>>——如果資料還在，給你一個 Rc；如果已經釋放，回傳 None。

8.11.2.3 用 Weak 打破迴圈

回到剛才的例子。關鍵問題是：strong count 構成的圖上有環。只要把其中一個方向改成 Weak，strong count 的圖上就沒有環了——因為 Weak 不貢獻 strong count。

用一個具體的例子來說明。假設我們想建一個**雙向鏈結串列（doubly linked list）**——每個節點同時指向前一個和後一個節點，這樣我們要從頭走到尾還是從尾走到頭都很容易。如果兩個方向都用 Rc，相鄰的兩個節點就形成迴圈。

解法是：next（往後）用 Rc，prev（往前）用 Weak：

```
use std::rc::{Rc, Weak};
use std::cell::RefCell;

struct Node<T> {
    value: T,
    next: Option<Rc<RefCell<Node<T>>>>,
    prev: Option<Weak<RefCell<Node<T>>>>,
}
```

為什麼這樣就不會迴圈？看 strong count 的圖：

```
外部 ──Rc──→ A ──Rc──→ B ──Rc──→ C
             ←─ Weak ─←    ←─ Weak ─←
```

Weak 那些邊不算在 strong count 裡。strong count 的圖只有從左到右的箭頭，是一條鏈，沒有環。

外部放掉 A → A 的 strong count 歸零 → A 被 drop → A 的 next 也跟著 drop → B 的 strong count 歸零 → B 被 drop → ..... 連鎖反應一路到底。中間沒有任何節點被 prev 撐住，因為 prev 是 Weak，不貢獻 strong count。

8.11.2.4 upgrade 出來的 Rc 會造成問題嗎？

你可能會想：「如果我 upgrade 一個 Weak 拿到 Rc 之後一直握著不放，不就多了一個 strong count 嗎？」

沒錯，upgrade 出來的 Rc 確實會讓 strong count +1。但這個 Rc 是一個**獨立的變數**——它的 strong count 貢獻記在「持有那個 Rc 的變數」頭上，不是記在原本的 Weak 欄位上。Weak 欄位本身對 strong count 的貢獻永遠是 0。

迴圈問題在資料結構建好的當下就已經被解決或沒被解決了，跟你之後怎麼 upgrade 完全無關。

8.11.3 範例程式碼

```
use std::rc::{Rc, Weak};
use std::cell::RefCell;

struct Node<T> {
    value: T,
    next: Option<Rc<RefCell<Node<T>>>>,
    prev: Option<Weak<RefCell<Node<T>>>>,
}

impl<T> Node<T> {
    fn new(value: T) -> Rc<RefCell<Node<T>>> {
        Rc::new(RefCell::new(Node { value, next: None, prev: None }))
    }
}

/// 把 b 接在 a 後面
fn link<T>(a: &Rc<RefCell<Node<T>>>, b: &Rc<RefCell<Node<T>>>) {
    a.borrow_mut().next = Some(b.clone());
    b.borrow_mut().prev = Some(Rc::downgrade(a));
}

fn main() {
    let a = Node::new(1);
    let b = Node::new(2);
    let c = Node::new(3);

    link(&a, &b);
    link(&b, &c);

    // 從前往後走（用 Rc）
    print!("往後走：");
    let mut current = Some(a.clone());
    while let Some(node) = current {
        print!("{} ", node.borrow().value);
        // next 是 Option<Rc<...>>，as_ref 變成 Option<&Rc<...>>，再 map clone 出新的 Rc
        current = node.borrow().next.as_ref().map(|rc| rc.clone());
    }
    println!();

    // 從後往前走（用 Weak，需要 upgrade）
    print!("往前走：");
    let mut current = Some(c.clone());
```

```
    while let Some(node) = current {
        print!("{} ", node.borrow().value);
        current = node.borrow().prev.as_ref().and_then(|w| w.upgrade());
    }
    println!();

    // 檢查計數
    // strong count 記在被指向的節點上：
    // a.next 指向 b → b 的 strong +1，b.next 指向 c → c 的 strong +1
    // weak count 也記在被指向的節點上：
    // b.prev 指向 a → a 的 weak +1，c.prev 指向 b → b 的 weak +1
    // a: strong=1（變數 a），weak=1（b.prev）
    // b: strong=2（變數 b + a.next），weak=1（c.prev）
    // c: strong=2（變數 c + b.next），weak=0（沒有節點的 prev 指向 c）
    println!("a strong={}, weak={}", Rc::strong_count(&a), Rc::weak_count(&a));
    println!("b strong={}, weak={}", Rc::strong_count(&b), Rc::weak_count(&b));
    println!("c strong={}, weak={}", Rc::strong_count(&c), Rc::weak_count(&c));
}
```

8.11.4 重點整理

- Rc 的參考迴圈會造成記憶體洩漏——strong count 永遠無法歸零。
- Weak 不增加 strong count，所以不會阻止資料被釋放。
- `Rc::downgrade(&rc)` 建立 `Weak<T>`，`weak.upgrade()` 回傳 `Option<Rc<T>>`。
- 用 Weak 打破迴圈：讓 strong count 構成的圖上不會有環。
- 雙向鏈結串列的做法：`next` 用 `Rc`（擁有後繼），`prev` 用 `Weak`（觀察前驅）。
- Weak 欄位對 strong count 的貢獻永遠是 0，`upgrade` 出來的 `Rc` 是獨立的變數。
- `Rc::strong_count()` 和 `Rc::weak_count()` 可以查看目前的計數。

8.12 fully qualified syntax

8.12.1 本集目標

學會三種不同層級的方法呼叫語法，以及在 `trait` 方法名稱衝突時如何消歧義。

> 本集是**第 5 章**的補充。

8.12.2 概念說明

在 Rust 裡，呼叫一個方法其實有三種寫法，從簡單到完整：

8.12.2.1 第一種：方法語法

```
    dog.speak();
```

最常用的寫法。編譯器會自動找到對應的方法。

8.12.2.2 第二種：指定 `trait` 或型別

```
    Animal::speak(&dog);
```

明確告訴編譯器「我要呼叫 `Animal` `trait` 上的 `speak`」。`&dog` 就是原本的 `&self`。

8.12.2.3 第三種：完全限定語法（fully qualified syntax）

```
    <Dog as Animal>::speak(&dog);
```

最明確的寫法：「在 Dog 實作的 Animal trait 上，呼叫 speak 方法，傳入 &dog」。

8.12.2.4 什麼時候需要用到？

大部分時候第一種就夠了。但當**多個 trait 定義了同名方法**的時候，編譯器不知道你要呼叫哪一個，就需要更明確的語法：

```
trait Animal {
    fn name(&self) -> &str;
}

trait Robot {
    fn name(&self) -> &str;
}
```

如果某個型別同時實作了 Animal 和 Robot，呼叫 .name() 時編譯器會報錯。這時候就需要第二種或第三種的語法來消歧義。

8.12.2.5 associated function 更常需要

如果是沒有 self 參數的 **associated function**，因為沒有接收者可以讓編譯器推斷，更容易需要完全限定語法：

```
    // 如果多個 trait 都有 create() 這個 associated function
    let x = <MyType as TraitA>::create();
```

8.12.2.6 存取 associated type

完全限定語法也可以用來存取某個型別在特定 trait 上的 **associated type**：

```
// IntoIterator trait 有一個 associated type 叫 Item
// 用完全限定語法取得它的具體型別：
type MyItem = <Vec<i32> as IntoIterator>::Item; // i32
```

有些地方可以直接寫 Type::TypeName，但如果有歧義或是編譯器無法推斷，就需要用完全限定語法明確指定。

8.12.3 範例程式碼

```
trait Animal {
    fn speak(&self);
    fn category() -> &'static str;
}

trait Robot {
    fn speak(&self);
    fn category() -> &'static str;
}

struct CyberDog {
    name: String,
}
```

```rust
impl Animal for CyberDog {
    fn speak(&self) {
        println!("{} 汪汪叫！(動物) ", self.name);
    }

    fn category() -> &'static str {
        "哺乳類"
    }
}

impl Robot for CyberDog {
    fn speak(&self) {
        println!("{} 嗶嗶叫！(機器人) ", self.name);
    }

    fn category() -> &'static str {
        "人工智慧"
    }
}

// CyberDog 自己也有 speak
impl CyberDog {
    fn speak(&self) {
        println!("{} 汪嗶汪嗶！(本體) ", self.name);
    }
}

fn main() {
    let dog = CyberDog {
        name: String::from("小白"),
    };

    // 第一層：方法語法 —優先呼叫型別本身的方法
    dog.speak(); // "小白 汪嗶汪嗶！(本體) "

    // 第二層：指定 trait
    Animal::speak(&dog); // "小白 汪汪叫！(動物) "
    Robot::speak(&dog);  // "小白 嗶嗶叫！(機器人) "

    // 第三層：完全限定語法
    <CyberDog as Animal>::speak(&dog); // "小白 汪汪叫！(動物) "
    <CyberDog as Robot>::speak(&dog);  // "小白 嗶嗶叫！(機器人) "

    // associated function（沒有 self）—更需要完全限定語法
    // Animal::category(); // 編譯錯誤！編譯器不知道是哪個型別的實作
    let animal_cat = <CyberDog as Animal>::category();
    let robot_cat = <CyberDog as Robot>::category();
    println!("動物分類：{}", animal_cat);
    println!("機器人分類：{}", robot_cat);

    // 存取 associated type
    // Vec<i32> 實作了 IntoIterator，它的 Item 是 i32
    // 用完全限定語法取得 associated type：
    let _: <Vec<i32> as IntoIterator>::Item = 42; // 型別是 i32
```

```
    println!("Vec<i32> 的 IntoIterator::Item 是 i32");
}
```

8.12.4 重點整理

- 方法呼叫有三種層級：`object.method()` → `Trait::method(&object)` → `<Type as Trait>::method(&object)`。
- 通常用最簡單的就好，有衝突時才升級。
- 當多個 `trait` 定義同名方法時，需要指定要呼叫哪個 `trait` 的版本。
- associated function（沒有 `self`）更常需要完全限定語法。
- 完全限定語法的格式：`<Type as Trait>::function(args)`。
- 也可以用來存取 associated type：`<Type as Trait>::TypeName`。

8.13 DST 簡介

8.13.1 本集目標

理解什麼是動態大小型別（DST），以及 `Sized`、`?Sized` 在泛型中的意義。

> 本集是**第 5 章**的補充。

8.13.2 概念說明

（本集提到的指標大小以 64 位元系統為準——現在絕大多數電腦都是 64 位元。）

在 Rust 的型別系統裡，大部分型別的大小在編譯時期就已知——`i32` 是 4 bytes、`bool` 是 1 byte、`(i32, i32)` 是 8 bytes。但有些型別的大小**在編譯時期是未知的**，這就是 **DST（Dynamically Sized Types）**，動態大小型別。

8.13.2.1 常見的 DST

你其實已經見過它們了：

- **`str`**：字串切片的「內容」型別。`"hello"` 是 5 bytes，`"哈囉"` 是 6 bytes，長度不固定。
- **`[T]`**：陣列切片的「內容」型別。`[i32]` 可能是 3 個元素也可能是 100 個。

因為大小不固定，你**不能**直接把它們當作值使用：

```
fn main() {
    let s: str = "hello";       // 編譯錯誤！
    let arr: [i32] = [1, 2, 3]; // 編譯錯誤！
}
```

8.13.2.2 怎麼用？靠指標！

DST 必須藏在某種指標後面：

- `&str`、`&[T]` — 參考
- `Box<str>`、`Box<[T]>` — 指向 heap 的指標

指向 `str` 和 `[T]` 的指標是**胖指標（fat pointer）**——它們包含一個位址和一個長度：

```
一般指標：[位址]         (8 bytes)
胖指標：  [位址][長度]   (16 bytes)
```

所以 &str 實際上佔 16 bytes：8 bytes 指向字串資料，8 bytes 記錄長度。

8.13.2.3 Sized trait

Rust 有一個特殊的 trait 叫 Sized，表示「這個型別的大小在編譯時期已知」。**絕大多數型別都自動實作了 Sized**。

而且——這是很多人不知道的——**泛型參數預設有 Sized bound**：

```
fn print_it<T>(val: T) { ... }
// 其實等同於
fn print_it<T: Sized>(val: T) { ... }
```

這很合理，因為如果 T 的大小未知，函數根本不知道要在 stack 上分配多少空間。

8.13.2.4 ?Sized：放寬限制

有時候你希望泛型參數可以接受 DST，這時候用 ?Sized 來放寬限制：

```
fn print_it<T: ?Sized>(val: &T) { ... }
//                     ^^^^^^^ 注意：必須透過參考
```

?Sized 的意思是「T 可以是 Sized，也可以不是」。但因為大小可能未知，你通常只能透過參考或智慧指標來使用 T。

8.13.2.5 trait 裡的 Self 預設是 ?Sized

前面說泛型參數 T 預設有 Sized bound。但 trait 裡的 Self 是個例外——它預設是 ?Sized 的，也就是說 Self 不一定是 Sized。

還記得第 4 章第 3 集介紹的 Clone 嗎？它的方法是 fn clone(&self) -> Self——直接回傳 Self。要把這項操作限制在 Sized 型別，可以把 bound 加在整個 trait 上，也可以只在該方法加上 where Self: Sized。Clone 選擇加在整個 trait 上：

```
trait Clone: Sized {
    fn clone(&self) -> Self;
}
```

8.13.2.6 回頭看第 5 章的 Cow

第 5 章最後一集教 Cow 的時候，我們用的也是簡化版的定義：

```
// 第 5 章提供的簡化版
pub enum Cow<'a, B>
where
    B: 'a + ToOwned,
{
    Borrowed(&'a B),
    Owned(B::Owned),
}
```

如果你試過要把 str 或 [T] 放進 Cow——例如寫 Cow<'_, str>——你會發現編譯不過。因為泛型參數 B 預設要求 Sized，而 str 不是 Sized。

加上 ?Sized 就能解決：

```
pub enum Cow<'a, B>
where
```

```
    B: 'a + ToOwned + ?Sized,
{
    Borrowed(&'a B),
    Owned(B::Owned),
}
```

`Borrowed(&'a B)` 裡的 `B` 已經在參考後面，所以即使 `B` 是 DST 也沒問題——胖指標會幫你搞定。

8.13.2.7 `&mut [T]` 與 `&mut str`

DST 也可以拿可變參考。`&mut [T]` 很實用——你可以修改切片裡的元素：

```
    let mut arr = [1, 2, 3, 4, 5];
    let slice: &mut [i32] = &mut arr[1..4];
    slice[0] = 99;  // arr 變成 [1, 99, 3, 4, 5]
```

但 `&mut str` 就很沒用了。雖然語法上合法，但你幾乎做不了什麼。原因是：

首先，`&mut str` 和 `&mut [T]` 一樣不能改長度。 `str` 是 DST，`&mut str` 是一個胖指標（位址 + 長度），長度是參考的一部分。`&mut str` 只是借用——你不擁有那塊記憶體的配置權，沒辦法讓它變大或變小。要改長度只能透過擁有記憶體的 `String`。

再來，連改內容都受限。 UTF-8 編碼裡，一個字元可能佔 1~4 bytes：

- `'a'` → 1 byte
- `'é'` → 2 bytes
- `'哈'` → 3 bytes

假設你有 `"哈囉"`（6 bytes），想把 `'哈'` 改成 `'a'`——`'a'` 只有 1 byte，但 `'哈'` 佔了 3 bytes，你沒辦法就地替換，因為長度不同。如果硬改了第一個 byte 卻沒處理後面的，UTF-8 的多 byte 序列就斷了。而 Rust 的 `str` 保證內容一定是合法的 UTF-8，破壞這個保證會導致未定義行為。

所以標準庫裡 `&mut str` 上的方法少得可憐，基本上只有 `make_ascii_uppercase()` 和 `make_ascii_lowercase()` 這類「不會改變 byte 長度」的操作（ASCII 字母的大小寫轉換剛好是 1 byte 對 1 byte）。要修改字串，還是用 `String` 吧。

8.13.2.8 DST 與 `Deref`

第 5 章也介紹了 `Deref` trait。`String` 和 `Vec<T>` 也實作了 `Deref`，它們的 `Deref` target 正是 DST：

- `String` 實作了 `Deref`，`Deref::deref(&String)` 回傳 `&str`。
- `Vec<T>` 實作了 `Deref`，`Deref::deref(&Vec<T>)` 回傳 `&[T]`。

也就是說 `String` 的 target 是 `str`，`Vec<T>` 的 target 是 `[T]`。雖然 DST 沒辦法直接放在變數裡，但 deref coercion 發生在**參考的層級**：`&String` 轉成 `&str`，`&Vec<T>` 轉成 `&[T]`。轉換的結果就是一個胖指標，帶著位址和長度，不需要知道 DST 的實際大小。

這就是為什麼一個接受 `&str` 的函數可以直接傳 `&String` 進去，接受 `&[T]` 的函數可以直接傳 `&Vec<T>` 進去——背後的機制正是 DST + `Deref` 的組合。

8.13.2.9 看不懂指標？

如果你對「指標」、「胖指標」、「位址」這些概念的理解還是很模糊，別擔心——下一章的第一集會正式介紹指標到底是什麼。

8.13.3 範例程式碼

```rust
use std::fmt::Display;

// 預設：T 必須是 Sized
fn print_sized<T: Display>(val: T) {
    println!("Sized 值：{}", val);
}

// 放寬：T 可以是 DST，但必須透過參考
fn print_unsized<T: Display + ?Sized>(val: &T) {
    println!("可能是 DST：{}", val);
}

// 展示 64 位元電腦上的胖指標大小
fn show_pointer_sizes() {
    use std::mem::size_of;

    println!("--- 指標大小比較 ---");
    println!("&i32     = {} bytes", size_of::<&i32>());     // 8
    println!("&[i32]   = {} bytes", size_of::<&[i32]>());   // 16（胖指標）
    println!("&str     = {} bytes", size_of::<&str>());     // 16（胖指標）
    println!("Box<i32> = {} bytes", size_of::<Box<i32>>()); // 8
    println!("Box<str> = {} bytes", size_of::<Box<str>>()); // 16（胖指標）
}

fn main() {
    // Sized 值：一般型別
    print_sized(42);
    print_sized(String::from("hello"));

    // ?Sized：可以接受 &str（str 是 DST）
    print_unsized("hello");                // T = str（DST）
    print_unsized(&42);                    // T = i32（Sized，也可以）
    print_unsized(&String::from("world")); // T = String（Sized）

    // &str 和 &[T] 是胖指標
    show_pointer_sizes();

    // str 和 [T] 不能直接當值用
    // let s: str = *"hello";    // 編譯錯誤！
    // let a: [i32] = *&[1,2,3]; // 編譯錯誤！

    // 但透過參考就沒問題
    let s: &str = "hello";
    let a: &[i32] = &[1, 2, 3];
    println!("\n&str = {}", s);
    println!("&[i32] 長度 = {}", a.len());

    // Box<str> 也可以
    let boxed: Box<str> = String::from("boxed string").into_boxed_str();
    println!("Box<str> = {}", boxed);
}
```

8.13.4 重點整理

- **DST（Dynamically Sized Types）**：大小在編譯時期未知的型別，如 `str`、`[T]`。
- DST 不能直接當值使用，必須透過指標：`&str`、`&[T]`、`Box<str>` 等。
- 指向 `str` 和 `[T]` 的指標是**胖指標（fat pointer）**，包含位址與長度；在 64 位元電腦上佔 16 bytes。
- **`Sized`**：表示型別大小在編譯時期已知；泛型參數預設有 `T: Sized` bound。
- **`?Sized`**：放寬限制，讓泛型參數可以接受 DST。
- `trait` 裡的 `Self` 預設是 `?Sized`。`Clone` 對整個 `trait` 加上 `Sized`；也可以改在方法加上 `where Self: Sized`，讓限制只套用於該方法。
- `Cow<'a, B>` 中的 `B: ?Sized` 就是為了讓 `B` 可以是 `str` 或 `[T]` 等 DST。
- `String` 和 `Vec<T>` 的 `Deref` target 分別是 DST `str` 和 `[T]`，deref coercion 讓 `&String` → `&str`、`&Vec<T>` → `&[T]` 成為可能。

8.14 `extern crate`

8.14.1 本集目標

了解 `extern crate` 和 `use` 的差別，以及為什麼本教學的部分範例仍然會寫 `extern crate`。

> 本集是**第 7 章**的補充。

8.14.2 正文

第 7 章介紹過如何用 Cargo 加入外部 `crate`。例如要使用 `rand`，先執行：

```
cargo add rand
```

Cargo 會把 `rand` 加進 `Cargo.toml`。在新版 Rust 中，這樣就可以直接從程式裡使用它：

```rust
use rand::RngExt;

fn main() {
    let mut rng = rand::rng();
    let n = rng.random_range(1..=100);
    println!("{}", n);
}
```

一般的 Cargo 專案不需要再寫 `extern crate rand;`。

8.14.2.1 `extern crate` 做了什麼？

`extern crate` 會明確告訴編譯器要載入某個外部 `crate`：

```rust
extern crate rand;
```

不過，它不會下載或安裝 `rand`，也不能取代 `Cargo.toml` 裡的 dependency。外部 `crate` 還是必須先由 Cargo 或其他建置工具準備好。

8.14.2.2 `extern crate` 和 `use` 不一樣

下面兩行的工作不同：

```
extern crate rand;

use rand::RngExt;
```

- `extern crate rand;` 明確載入 `rand` 這個外部 crate。
- `use rand::RngExt;` 把 `rand` 裡的 `RngExt` trait 引入目前的作用域。

也就是說，`extern crate` 處理的是外部 crate 本身，`use` 處理的是程式裡名稱的使用方式。

8.14.2.3 用 as 替外部 crate 取別名

`extern crate` 也可以在載入外部 crate 時，替它建立一個在目前作用域中使用的名稱：

```
extern crate rand as random;
```

這個語法的一般形式是：

```
extern crate a as b;
```

其中：

- `a` 是外部 crate 的名稱。
- `b` 是這項宣告在目前作用域中建立的名稱。

例如，替 `rand` 取名為 `random` 之後，就可以透過 `random` 使用它：

```
extern crate rand as random;

use random::RngExt;

fn main() {
    let mut rng = random::rng();
    let n = rng.random_range(1..=100);
    println!("{}", n);
}
```

不過，在新版 Rust 的一般 Cargo 專案中，如果目的只是替名稱取別名，通常直接使用第 7 章介紹過的 `use ... as ...` 即可：

```
use rand as random;
```

雖然兩種語法都使用 `as`，用途仍然不同：`extern crate rand as random;` 明確載入外部 crate，並建立名稱 `random`；`use rand as random;` 則是替已經可以使用的 `rand` 建立別名。

8.14.2.4 為什麼本教學仍然使用它？

你可能已經在本教學的範例中看過：

```
extern crate rand;
```

這不代表新版 Rust 的一般 Cargo 專案仍然需要這樣寫。**本教學加入 `extern crate`，是為了通過內部測試。**

本教學使用 `mdbook test`，自動編譯並測試書中的 Rust 程式碼。測試前會先把範例需要的外部 crate 編譯好，再使用 `-L` 告訴測試工具要去哪個資料夾尋找這些編譯結果。

但 `-L` 只提供搜尋路徑，不會像 Cargo 一樣替每個 dependency 傳入完整的資訊。因此，範例需要用 `extern crate rand;` 明確告訴編譯器要載入 `rand`，內部測試才能找到並使用它。

這是本教學測試方式的特殊需求，不是新版 Rust 的一般寫法。如果你把範例複製到自己的 Cargo 專案，而且已經用 `cargo add rand` 加入 dependency，通常可以刪除 `extern crate rand;`。

8.14.3 重點整理

- `extern crate name;` 會明確告訴編譯器載入某個外部 `crate`。
- `extern crate` 不會下載套件，也不能取代 `Cargo.toml` 裡的 dependency。
- `extern crate` 和 `use` 的用途不同：前者處理外部 `crate`，後者把名稱引入作用域。
- `extern crate a as b;` 會明確載入外部 `crate a`，並在目前作用域中替它建立名稱 `b`；新版 Rust 若只需要別名，通常使用 `use a as b;`。
- 新版 Rust 的一般 Cargo 專案通常不需要寫 `extern crate`。
- 本教學加入 `extern crate`，是為了讓使用 `mdbook test -L` 的內部測試能找到外部 `crate`。

第二部

第 9 章

多執行緒

本章會提到如何使用多執行緒撰寫可同時運行多個任務的程式。

9.1 指標

9.1.1 本集目標

理解記憶體位址的概念，知道指標在底層是什麼東西。

9.1.2 概念說明

前面幾章我們用 `&T`、`Box<T>`、`Rc<T>` 的時候，關心的是「誰擁有資料」、「誰在借」。這一集要換個角度——這些東西在記憶體裡到底是什麼。

附錄一的〈DST 簡介〉曾經提過 DST 和胖指標的概念。如果當時覺得看不太懂是正常的——因為那時候我們還沒正式介紹過指標。這一集就是要把這個基礎補上。

> 這一集會用比較簡化的方式理解記憶體與位址；實際執行時，編譯器與硬體做的事情會複雜很多，但我們這裡不會深入探討那些細節。

9.1.2.1 記憶體位址

程式執行的時候，每一個變數都會被放在記憶體的某個位置，而每個位置都有一個編號，叫做**位址**。`&x` 拿到的就是 `x` 的位址。用 `{:p}` 格式化可以把它印出來看：

```
fn main() {
    let x: i32 = 42;
    println!("{:p}", &x); // 例如 0x7ffd5e8a3b4c
}
```

這個十六進位的數字就是 `x` 在記憶體中的位址。

9.1.2.2 &T 的真面目

`&x` 產生的值，一般來說就是 `x` 的記憶體位址。`&T` 這個型別存的東西，本質上也就是作為位址的數字。當你把 `&x` 傳進函數的時候，傳的不是 `x` 的內容，是 `x` 的位址。

9.1.2.3 指標大小

在大部分情況下，`&T` 佔 8 bytes，在 64 位元系統上就是一個位址的大小。用 `std::mem::size_of` 來驗證：

```
use std::mem::size_of;

fn main() {
    println!("{}", size_of::<i32>());          // 4
```

```
    println!("{}", size_of::<[i32; 1000]>());  // 4000
    println!("{}", size_of::<&i32>());         // 8
    println!("{}", size_of::<&[i32; 1000]>()); // 8
    println!("{}", size_of::<Box<i32>>());     // 8
}
```

上面的 &T 和 Box<T> 指向的都是 Sized 型別；在這個前提下，它們大小一樣——因為存的都只有位址。&T 指向的資料可能在 stack 上也可能在 heap 上，而擁有所有權的 Box<T> 指向的一定在 heap 上。不管指向哪裡，位址本身的大小是一樣的。所以當 T 本身很大的時候，傳一個位址會比複製整個 T 輕量——但代價是每次存取都要透過位址去查找，多了一層間接。

9.1.2.4 解參考

有了位址，我們能做什麼？用 * 運算子**解參考**（dereference），透過位址取得對應的內容：

```
fn main() {
    let x = 42;
    let r = &x;
    println!("{}", *r); // 透過位址取得值：42
}
```

解參考不是免費的，大部分時候這個成本很小，但知道它的存在是有意義的。

9.1.2.5 胖指標（fat pointer）

附錄一的〈DST 簡介〉介紹過 DST——[T] 和 str 是大小不確定的型別，沒辦法直接放在變數裡，通常要透過 &[T]、&str、Box<[T]> 等方式使用。但 DST 的大小不固定，光有位址不夠。想像一下：你拿到一個位址，知道從這裡開始是一段連續的 i32 資料——但到哪裡結束？記憶體本身不會告訴你，位址只是一個起點。所以除了位址之外，還得額外記錄長度，才知道這段資料有多長。因此 &[T] 和 &str 佔 16 bytes：

```
use std::mem::size_of;

fn main() {
    println!("{}", size_of::<&i32>());   // 8（位址）
    println!("{}", size_of::<&[i32]>()); // 16（位址 + 長度）
    println!("{}", size_of::<&str>());   // 16（位址 + 長度）
}
```

9.1.3 範例程式碼

```
use std::mem::size_of;

fn main() {
    let x: i32 = 42;
    let r: &i32 = &x;

    // 印出位址
    println!("x 的位址：{:p}", &x);
    println!("r 存的值：{:p}", r); // 和上面一樣

    // 解參考
    println!("透過 r 取得 x 的值：{}", *r);
```

```
    // 智慧指標也能解參考
    let b = Box::new(99);
    println!("Box 裡的值：{}", *b);

    // 指標大小
    println!("--- 一般指標 ---");
    println!("i32 大小：{} bytes", size_of::<i32>());
    println!("&i32 大小：{} bytes", size_of::<&i32>());
    println!("[i32; 1000] 大小：{} bytes", size_of::<[i32; 1000]>());
    println!("&[i32; 1000] 大小：{} bytes", size_of::<&[i32; 1000]>());
    println!("Box<i32> 大小：{} bytes", size_of::<Box<i32>>());

    // 胖指標
    println!("--- 胖指標 ---");
    println!("&[i32] 大小：{} bytes", size_of::<&[i32]>());
    println!("&str 大小：{} bytes", size_of::<&str>());
    println!("Box<[i32]> 大小：{} bytes", size_of::<Box<[i32]>>());
}
```

9.1.4 重點整理

- &T 在底層就是一個記憶體位址，本質上是一個數字。
- 在 64 位元系統上的大部分情況下，&T 和 Box<T> 的大小是 8 bytes，就是一個位址的大小。
- 用 * 解參考，透過位址取得對應的內容，這有一層間接存取的成本。
- &[T] 和 &str 是胖指標（fat pointer），佔 16 bytes（位址 + 長度），因為 DST 的大小不固定。

9.2 thread::spawn

9.2.1 本集目標

學會建立執行緒，讓程式同時做好幾件事。

9.2.2 概念說明

到目前為止，我們的程式都只有單一執行流程，同一時間只會做一件事。但有些時候你希望程式能**同時**做好幾件事——比如一邊下載檔案一邊更新進度條。這就是**執行緒**的用途。

9.2.2.1 建立執行緒

std::thread::spawn 接收一個閉包，在新的執行緒上執行它：

```
use std::thread;

fn main() {
    thread::spawn(|| {
        println!("我在另一個執行緒！");
    });
}
```

9.2.2.2 不 join 就會死

有個很重要的事：main 函數結束時，整個程式就結束了——不管其他執行緒有沒有跑完。

```rust
use std::thread;

fn main() {
    thread::spawn(|| {
        for i in 0..10 {
            println!("子執行緒：{}", i);
        }
    });

    println!("main 結束了");
    // 子執行緒可能只印了一部分，甚至什麼都沒印
}
```

9.2.2.3 JoinHandle

thread::spawn 會回傳一個 JoinHandle。呼叫 .join() 可以等待那個執行緒跑完：

```rust
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        for i in 0..5 {
            println!("子執行緒：{}", i);
        }
    });

    handle.join().expect("執行緒發生錯誤"); // 等子執行緒跑完
    println!("全部完成");
}
```

.join() 不只是等待——它還能拿到閉包的回傳值。閉包回傳什麼，.join().expect("執行緒發生錯誤") 就得到什麼：

```rust
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        let answer = 21 * 2;
        answer // 閉包的回傳值
    });

    let result = handle.join().expect("執行緒發生錯誤");
    println!("從另一個執行緒拿到的結果：{}", result); // 42
}
```

這是從另一個執行緒把計算結果傳回來最簡單的方式。

9.2.2.4 move 閉包

如果閉包裡要用到外面的變數，一般需要加 move：

```rust
use std::thread;

fn main() {
    let name = String::from("Rust");

    let handle = thread::spawn(move || {
```

```
        println!("Hello, {}!", name);
    });

    println!("{}", name); // 編譯錯誤！name 已經被 move 進閉包了

    handle.join().expect("執行緒發生錯誤");
}
```

為什麼需要 move？因為新執行緒可能活得比呼叫 spawn 的函數還久。如果閉包只是借用 name，而那個函數先結束、把 name 丟掉了，閉包就拿著一個懸垂參考。加上 move 之後，name 的所有權搬進了閉包裡，不管原本的作用域怎麼結束，閉包都能繼續用它。

9.2.2.5　輸出交錯

多個執行緒同時跑的時候，它們的輸出會交錯——每次執行結果可能不一樣：

```
use std::thread;

fn main() {
    let h1 = thread::spawn(|| {
        for _ in 0..5 {
            println!("AAA");
        }
    });

    let h2 = thread::spawn(|| {
        for _ in 0..5 {
            println!("BBB");
        }
    });

    h1.join().expect("執行緒發生錯誤");
    h2.join().expect("執行緒發生錯誤");
}
```

跑幾次看看，你會發現 AAA 和 BBB 的順序每次都可能不同。這就是多執行緒的特性——執行順序是不確定的。

9.2.3　範例程式碼

```
use std::thread;

fn main() {
    let data = vec![1, 2, 3, 4, 5];

    let handle = thread::spawn(move || {
        let sum: i32 = data.iter().sum();
        println!("子執行緒算出的總和：{}", sum);
        sum
    });

    // data 已經被 move 了，這裡不能再用
    // println!("{:?}", data); // 編譯錯誤

    let result = handle.join().expect("執行緒發生錯誤");
```

```
    println!("主執行緒收到結果：{}", result);
}
```

9.2.4 重點整理

- `thread::spawn(|| { ... })` 建立一個新的執行緒。
- `main` 結束時所有執行緒跟著死，用 `.join()` 等待執行緒完成。
- `.join()` 還能拿到閉包的回傳值，是從另一個執行緒傳回結果最簡單的方式。
- 閉包裡要用外面的變數一般需要 `move`，因為不確定新執行緒的生命週期。
- 多個執行緒的執行順序是不確定的，輸出可能會交錯。

9.3 Send / Sync

9.3.1 本集目標

理解 Rust 如何在編譯時期保證型別能安全地跨執行緒使用。

9.3.2 概念說明

9.3.2.1 為什麼需要額外的保護

還記得第 4 章一開頭的鑰匙圈比喻嗎？現在你應該能理解鑰匙就是指標了。

Rust 有所有權規則以及借用規則——例如同時不能有兩個 `&mut`——的原因之一就是為了防止一個位址的值被同時讀寫，造成前面提到過的**資料競爭（data race）**。舉個具體的例子：假設有一個 `i32` 值是 0，執行緒 A 和執行緒 B 各自透過指標對它加 1。你預期結果是 2，但實際上可能是這樣：

1. 執行緒 A 讀取值：0。
2. 執行緒 B 也讀取值：0。
3. 執行緒 A 把 0 + 1 = 1 寫回去。
4. 執行緒 B 也把 0 + 1 = 1 寫回去。

結果是 1，不是 2。兩次加 1 只生效了一次。

注意這裡的關鍵是兩個執行緒同時對同一份資料做讀寫——事實上，只要有共用的資料而且有人在寫，就可能出事。即使另一方只是在讀，也可能讀到還沒完全完成寫入的資料。

Rust 的所有權和借用規則能防止很多問題——例如同時不能有兩個 `&mut`，`&` 和 `&mut` 也不能同時存在。但在多執行緒下，光靠這些還不夠。比如上面的例子：如果只是把 `i32` 的值傳給另一個執行緒，`i32` 是 `Copy` 的，會直接複製一份過去，兩邊各自操作自己的副本，不會出事。但有些型別不是這麼單純——把它 move 過去之後，原本的執行緒可能還持有共用的資料。哪些型別可以安全地跨執行緒？哪些不行？Rust 用兩個 `trait`——`Send` 和 `Sync`——來回答這個問題。

9.3.2.2 `spawn` 實際上在做什麼

上一集用 `thread::spawn` 建立執行緒的時候，我們傳入了一個閉包。閉包會捕捉外部變數——而 `spawn` 實際上就是**把這些捕捉的變數送到另一個執行緒去**。這才是我們真正要關心的事：哪些東西可以被安全地送過去？

9.3.2.3 Send

一個型別如果實作了 Send，代表它的值可以安全地 move 到另一個執行緒。大部分型別都是 Send——i32、String、Vec<T>（只要 T 是 Send）等等都是。

9.3.2.4 Sync

一個型別如果實作了 Sync，代表它的 &T（共享參考）可以安全地在多個執行緒之間共享。換句話說：

T: Sync 等價於 &T: Send

如果 &T 能安全地送到另一個執行緒，那 T 就是 Sync 的。

9.3.2.5 Sync 通常蘊含 Send

如果一個東西能被多個執行緒同時讀都沒問題（Sync），那把它整個搬到另一個執行緒去——連同時讀的可能性都不存在了——通常只會更安全。所以大部分 Sync 的型別也是 Send，但少數例外存在。

9.3.2.6 auto trait：編譯器自動幫你實作的 trait

通常不需要手動實作 Send 或 Sync。它們是所謂的 **auto trait**——編譯器會自動幫你的型別實作。規則很簡單：如果一個型別裡存的資料都是 Send，那它本身預設就是 Send。Sync 同理。

```
struct MyData {
    x: i32,    // Send + Sync
    s: String, // Send + Sync
}
// MyData 自動就是 Send + Sync
```

9.3.2.7 不用死背

你不需要記住哪些型別是 Send、哪些是 Sync。試著把一個不安全的型別丟進 thread::spawn，編譯器會直接報錯告訴你：

```
use std::rc::Rc;
use std::thread;

fn main() {
    let data = Rc::new(42);
    thread::spawn(move || {
        println!("{}", data);
    });
    // 編譯錯誤！Rc<i32> 不是 Send
}
```

9.3.2.8 回頭看 spawn 的型別簽名

現在我們知道了 Send 和 Sync，可以回頭看看 thread::spawn 的型別簽名：

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static,
```

閉包 F 必須是 Send——閉包捕捉了什麼，它的型別就包含什麼，所以如果捕捉的變數不是 Send，閉包本身也不會是 Send，spawn 就會編譯失敗。回傳值 T 也必須是 Send，因為結果要從子執行緒傳回來。

還有那個 'static——為什麼需要它？因為我們完全不知道 spawn 出來的執行緒會活多久。你可能會 join 它，也可能不 join 讓它自己跑到 main 結束才被強制終止。Rust 的型別系統沒辦法保證你一定會在某個時間點 join，所以它要求最保守的保證：閉包和回傳值都不能持有會過期的參考。第 5 章第 29 集學過 lifetime bound，T: 'a 代表 T 裡面的所有參考都必須活得過 'a。F: 'static 就是這個概念的極端情況——閉包裡面的參考要活得跟整個程式一樣久。實務上最常見的做法，是用 move 把擁有所有權的值搬進閉包。不過如果捕捉的變數本身就是參考，move 只是把那個參考搬進或複製進閉包，閉包裡仍然有那個參考。使用 thread::spawn 時，應該改成把原本擁有所有權的資料搬進閉包，或先 clone 出一份擁有所有權的資料再搬進去。如果真的需要讓執行緒持有區域資料的參考，這一章第 13 集會介紹專門處理這種情況的 thread::scope。

9.3.3 範例程式碼

```rust
use std::thread;

// 這個 struct 的所有欄位都是 Send + Sync，
// 所以它自動就是 Send + Sync
struct Config {
    name: String,
    max_retries: u32,
}

fn main() {
    let config = Config {
        name: String::from("my_app"),
        max_retries: 3,
    };

    // Config 是 Send，可以安全地 move 到另一個執行緒
    let handle = thread::spawn(move || {
        println!("設定名稱：{}", config.name);
        println!("最大重試次數：{}", config.max_retries);
    });

    handle.join().expect("執行緒發生錯誤");
}
```

9.3.4 重點整理

- 資料競爭（data race）：多個執行緒同時存取同一份資料且至少一方在寫，結果不可預期。
- thread::spawn 的閉包會把捕捉的變數送到另一個執行緒，所以這些變數必須是 Send。
- Send = 值可以安全地 move 到另一個執行緒。
- Sync = &T 可以安全地在多個執行緒之間共享（T: Sync 等價於 &T: Send）。
- Sync 通常蘊含 Send——能被多執行緒同時讀，搬過去只會更安全。
- 編譯器通常會自動實作 Send / Sync，一般不需要手動標記。

9.4 RefCell 在多執行緒

9.4.1 本集目標

理解為什麼 interior mutability 在多執行緒下是危險的，以及 RefCell 的 Send / Sync 特性。

9.4.2 概念說明

9.4.2.1 interior mutability 是多執行緒的一大威脅

第 5 章學過，RefCell 能透過 &T（共享參考）修改內部的值。在單執行緒的世界裡，RefCell 會在執行時期檢查借用規則，不會出問題。

但在多執行緒的世界裡，事情就不一樣了。&T 看起來是「只讀」的，而 Sync 的定義就是 &T 能安全地在多個執行緒之間共享。如果一個型別能透過 &T 偷偷修改內容，多個執行緒同時這樣做就可能出事。

9.4.2.2 RefCell 的借用計數不是 atomic

RefCell 用普通的整數來記錄目前的借用狀態（有幾個不可變借用、有沒有可變借用）。這個計數器的操作不是 **atomic** 的。atomic 操作是不可分割的——其他執行緒不可能看到做到一半的狀態。RefCell 的計數器沒有這種保證，所以兩個執行緒可能都在任一方更新前讀到相同的舊值。如果兩個執行緒同時透過 &RefCell<T> 呼叫 borrow_mut()，以下是可能發生的事：

1. 執行緒 A 呼叫 borrow_mut()，讀取計數器，看到值是 0（沒有人在借）。
2. 執行緒 B 也呼叫 borrow_mut()，讀取計數器，也看到值是 0。
3. 執行緒 A 判斷「沒有人在借，可以拿可變借用」，把計數器改成「可變借用中」。
4. 執行緒 B 也判斷「沒有人在借」——因為它在步驟 2 讀到的是舊值——也拿到了可變借用。

結果：兩個執行緒同時拿到了可變借用，RefCell 的執行時期檢查完全被繞過了。

9.4.2.3 RefCell 不是 Sync

因為上面的原因，RefCell 不是 Sync——&RefCell<T> 不能在多個執行緒之間共享。如果你試著這樣做，編譯器會擋住你。

9.4.2.4 當 T: Send 時，RefCell<T> 是 Send

只要 T 本身是 Send，RefCell<T> 就可以被 **move** 到另一個執行緒。move 之後，只有那一個執行緒擁有這個 RefCell，不存在多個執行緒同時操作的問題。

```
use std::cell::RefCell;
use std::thread;

fn main() {
    let data = RefCell::new(vec![1, 2, 3]);

    // OK：RefCell<Vec<i32>> 是 Send，可以 move 到另一個執行緒
    let handle = thread::spawn(move || {
        data.borrow_mut().push(4);
        println!("{:?}", data.borrow());
    });

    handle.join().expect("執行緒發生錯誤");
}
```

9.4.3 範例程式碼

```
use std::cell::RefCell;
use std::thread;
```

```
fn main() {
    // RefCell<String> 可以 move 到另一個執行緒（Send）
    let data = RefCell::new(String::from("hello"));

    let handle = thread::spawn(move || {
        // 在這個執行緒裡，RefCell 運作正常
        data.borrow_mut().push_str(" world");
        println!("子執行緒：{}", data.borrow());
    });

    handle.join().expect("執行緒發生錯誤");

    // 但不能在多個執行緒之間共享 &RefCell（非 Sync）
    // 如果你試著讓兩個執行緒共享同一個 RefCell，編譯器會擋住你。
}
```

9.4.4 重點整理

- interior mutability 讓 &T 能修改內容，但這在多執行緒下很危險。
- atomic 操作 = 不可分割的操作，其他執行緒不可能看到做到一半的狀態。
- RefCell 的 borrow 計數是普通整數，不是 atomic，多執行緒同時操作可能繞過檢查。
- RefCell 不是 Sync——不能在多個執行緒之間共享 &RefCell<T>。
- 當 T: Send 時，RefCell<T> 是 Send——可以 move 到另一個執行緒，因為 move 後只有一個執行緒擁有。

9.5 Rc 在多執行緒

9.5.1 本集目標

理解為什麼 Rc 完全不能跨執行緒——既不是 Send 也不是 Sync。

9.5.2 概念說明

9.5.2.1 Rc 不是 Sync

Rc 的參考計數和 RefCell 的 borrow 計數一樣，是普通整數，不是 atomic 操作。如果多個執行緒同時透過 &Rc<T> 做 clone 或 drop，參考計數的加減可能互相干擾，導致計數錯誤——可能提前釋放資料，或永遠不釋放。

所以 Rc 不是 Sync，理由和 RefCell 相同。

9.5.2.2 Rc 連 Send 都不是

上一集說 RefCell 是 Send，因為 move 過去之後只有一個執行緒擁有。但 Rc 不一樣。

Rc 的設計就是讓多個 Rc 指向同一份資料。你把一個 Rc move 到另一個執行緒，但它的 clone 可能還留在原本的執行緒。兩邊同時操作參考計數，計數器就可能壞掉。

問題不在 move 本身，而在 **move 之後兩個執行緒仍然共享同一個計數器**。

```
use std::rc::Rc;

fn main() {
    let a = Rc::new(42);
```

```
    let b = a.clone(); // a 和 b 共享同一份資料和計數器

    // 如果把 a move 到另一個執行緒，
    // b 還在原本的執行緒——兩邊同時操作計數器就爆了
    std::thread::spawn(move || {
        println!("{}", a);
    });
    // 編譯錯誤！Rc<i32> 不是 Send
}
```

9.5.2.3 Rc 完全不能跨執行緒

Rc 不是 Send 也不是 Sync。不能 move 到其他執行緒，也不能在多個執行緒之間共享參考。如果需要跨執行緒共享資料，得用別的工具。

9.5.3 範例程式碼

```
use std::rc::Rc;
use std::thread;

fn main() {
    // Rc 在單執行緒中正常運作
    let a = Rc::new(String::from("hello"));
    let b = a.clone();
    println!("a = {}, b = {}", a, b);
    println!("計數 = {}", Rc::strong_count(&a));

    // 但不能跨執行緒——以下不會通過編譯：

    // let data = Rc::new(42);
    // thread::spawn(move || {
    //     println!("{}", data);
    // });
    // 編譯錯誤：Rc<i32> 不是 Send

    println!("Rc 只能在單執行緒中使用");
}
```

9.5.4 重點整理

- Rc 的參考計數是普通整數，不是 atomic，所以不是 Sync。
- Rc 連 Send 都不是：move 一個 Rc 到另一個執行緒後，它的 clone 可能還在原執行緒，兩邊同時操作計數器就會出問題。
- 也就是說 Rc 完全不能跨執行緒。

9.6 Arc<T>

9.6.1 本集目標

學會用 Arc<T> 在多個執行緒之間安全地共享資料。

9.6.2 概念說明

9.6.2.1 問題回顧

前面說了 Rc 不能跨執行緒，因為參考計數不是 atomic。但我們確實需要在多個執行緒之間共享資料——怎麼辦？

9.6.2.2 Arc：atomic reference counting

Arc<T> 就是把 Rc 的參考計數換成 **atomic 操作**的版本。atomic 操作保證即使多個執行緒同時修改計數器，也不會互相干擾。

用法跟 Rc 幾乎一樣：

```
use std::sync::Arc;

fn main() {
    let a = Arc::new(String::from("hello"));
    let b = Arc::clone(&a); // 跟 Rc 一樣：複製指標，共享資料
    println!("計數 = {}", Arc::strong_count(&a)); // 2
}
```

9.6.2.3 跨執行緒共享

把 Arc::clone 出來的東西 move 到另一個執行緒：

```
use std::sync::Arc;
use std::thread;

fn main() {
    let data = Arc::new(vec![1, 2, 3]);

    let data_clone = Arc::clone(&data);
    let handle = thread::spawn(move || {
        println!("子執行緒：{:?}", data_clone);
    });

    println!("主執行緒：{:?}", data);
    handle.join().expect("執行緒發生錯誤");
}
```

9.6.2.4 Arc<T>: Send 和 Arc<T>: Sync 要求 T: Send + Sync

Arc<T> 本身不要求 T: Send + Sync。但要像上面的例子一樣，在多個執行緒之間傳遞與共享 Arc<T>，T 就必須同時滿足這兩個 trait。為什麼？

Sync：多個執行緒透過各自的 Arc 同時存取同一份 T。第 5 章學了 Deref——Arc 實作了 Deref，所以你可以透過 Arc 直接存取 T 的內容。這等於多個執行緒同時持有 T 的共享參考，所以 T 必須是 Sync。

Send：最後一個 Arc 被 drop 的時候，T 也會被 drop。而哪個執行緒持有最後一個 Arc 是不確定的，所以 T 的 drop 可能發生在任何執行緒上——T 等於被「送」到那個執行緒去銷毀，所以 T 必須是 Send。

9.6.3 範例程式碼

```rust
use std::sync::Arc;
use std::thread;

fn main() {
    let data = Arc::new(vec![1, 2, 3, 4, 5]);

    let mut handles = vec![];

    for i in 0..3 {
        let data_clone = Arc::clone(&data);
        let handle = thread::spawn(move || {
            let sum: i32 = data_clone.iter().sum();
            println!("執行緒 {} 算出的總和：{}", i, sum);
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().expect("執行緒發生錯誤");
    }

    println!("最終計數 = {}", Arc::strong_count(&data)); // 1
}
```

9.6.4 重點整理

- `Arc<T>` 是 `Rc<T>` 的多執行緒版本，參考計數用 atomic 操作。
- 用法跟 `Rc` 幾乎一樣：`Arc::new()`、`Arc::clone()`。
- `Arc::clone` 後把 clone move 到其他執行緒，就能共享資料。
- `Arc<T>: Send` 和 `Arc<T>: Sync` 都要求 `T: Send + Sync`：Sync 因為多執行緒同時存取，Send 因為 drop 可能發生在任何執行緒。

9.7 atomic 型別

9.7.1 本集目標

學會用 atomic 型別在多個執行緒之間安全地讀寫簡單的值。

9.7.2 概念說明

9.7.2.1 什麼是 atomic 操作

上一集學了 `Arc`，它的參考計數用的是 atomic 操作。到底什麼是 atomic？

假設兩個執行緒同時對一個變數做 `count += 1`。這看起來是一步，但實際上分成三步：讀出目前的值、加 1、寫回去。如果兩個執行緒同時做這三步，可能會發生這樣的事：

1. 執行緒 A 讀出 `count = 0`。
2. 執行緒 B 讀出 `count = 0`。
3. 執行緒 A 寫入 `count = 1`。
4. 執行緒 B 寫入 `count = 1`。

兩邊各加了一次，結果卻是 1 而不是 2。

有些 **atomic 操作**把讀、改、寫合成一個不可分割的動作——其他執行緒不可能看到做到一半的狀態。用 atomic 操作做 `count += 1`，兩個執行緒同時跑，結果一定是 2。

9.7.2.2 `AtomicI32` 和 `AtomicBool`

標準庫在 `std::sync::atomic` 提供了幾種 atomic 型別，最常用的是整數和布林：

```
use std::sync::atomic::{AtomicI32, AtomicBool, Ordering};

fn main() {
    let counter = AtomicI32::new(0);
    let flag = AtomicBool::new(false);
}
```

9.7.2.3 基本操作

```
use std::sync::atomic::{AtomicI32, Ordering};

fn main() {
    let counter = AtomicI32::new(0);

    counter.store(10, Ordering::Relaxed);              // 寫入
    let val = counter.load(Ordering::Relaxed);         // 讀取：10
    let old = counter.fetch_add(5, Ordering::Relaxed); // 加 5，回傳舊值 10
    // 現在 counter 是 15
}
```

每個操作都要傳一個 `Ordering` 參數。為什麼需要這個？

現代處理器為了效能，可能會**重新排列指令的執行順序**。在單執行緒下這不會造成問題——處理器保證結果跟按順序執行一樣。但在多執行緒下，一個執行緒裡的指令重排，可能讓另一個執行緒看到不一致的狀態。

`Ordering` 就是告訴處理器「這個操作前後的指令不能隨便重排」，一般來說，限制越嚴格，效能代價越高。

舉個例子：假設執行緒 A 把資料寫進一個 `Vec`，然後把一個 atomic 旗標設成 `true`；執行緒 B 看到旗標是 `true` 就去讀那個 `Vec`：

```
// 執行緒 A
data.push(42);                          // 第 1 步：寫入資料
ready.store(true, Ordering::Relaxed); // 第 2 步：設旗標

// 執行緒 B
if ready.load(Ordering::Relaxed) {    // 看到 true
    println!("{}", data[0]);          // 但資料可能還沒寫進去！
}
```

用 `Relaxed` 的話，處理器可能把執行緒 A 的第 1 步和第 2 步重排——執行緒 B 看到旗標已經是 `true`，但資料還沒寫進去。處理器之所以敢重排，是因為從執行緒 A 自己的角度來看，先設旗標再寫資料和先寫資料再設旗標結果完全一樣——它不知道還有另一個執行緒在看。`SeqCst` 會保留程式裡的兩個先後順序：執行緒 A 先寫資料再設旗標，執行緒 B 先檢查旗標再讀資料。因此，執行緒 B 看到 `true` 時，也一定看得到執行緒 A 先寫好的資料。

細節很複雜，初學的話可以先記住兩個：

- `Ordering::Relaxed`：只保證這個 atomic 操作本身是正確的，不限制其他指令的順序。適合單純的計數器。
- `Ordering::SeqCst`：最嚴格。在這個例子裡，它會保留「寫資料 → 設旗標」和「檢查旗標 → 讀資料」的順序，所以執行緒 B 看到 `ready == true` 時，也一定看得到執行緒 A 先寫好的資料。

不確定的時候用 `SeqCst` 最安全。

9.7.2.4 interior mutability

注意看上面的程式碼——`store` 和 `fetch_add` 明明在修改值，卻不需要 `&mut self`，只要 `&self` 就行。這跟第 5 章學的 `Cell` 一樣，是 interior mutability。

為什麼一定要這樣設計？因為如果要 `&mut self` 才能修改，那就只有一個執行緒能拿到 `&mut`，其他執行緒根本碰不到這個值——那還跨什麼執行緒？atomic 的重點就是讓多個執行緒透過 `&` 同時存取同一個值，所以必須有 interior mutability。

`Cell` 也有 interior mutability，但 `Cell` 不是 `Sync`（不能跨執行緒共享）。atomic 是 `Sync`——因為底層硬體保證了操作的原子性，多個執行緒同時透過 `&` 修改也不會出問題。

9.7.2.5 搭配 `Arc` 使用

atomic 最常見的用法就是搭配 `Arc`，讓多個執行緒共同修改一個計數器：

```
use std::sync::Arc;
use std::sync::atomic::{AtomicI32, Ordering};
use std::thread;
fn main() {
    let counter = Arc::new(AtomicI32::new(0));

    let mut handles = vec![];

    for _ in 0..10 {
        let counter_clone = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            for _ in 0..1000 {
                counter_clone.fetch_add(1, Ordering::Relaxed);
            }
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().expect("執行緒發生錯誤");
    }

    println!("最終結果：{}", counter.load(Ordering::Relaxed)); // 一定是 10000
}
```

10 個執行緒各加 1000 次，結果一定是 10000——不會少算。

9.7.2.6 atomic 型別 vs 鎖

atomic 操作只能用在簡單的型別——例如整數（AtomicI32、AtomicU64、AtomicUsize 等）和布林（AtomicBool）。如果你要保護一個 Vec、String 或任何複雜的資料結構，atomic 型別做不到，需要用下一集教的鎖。

但對於簡單的計數器或旗標，atomic 操作比鎖快——每個執行緒都能直接操作，不需要排隊等別人用完。

9.7.3 範例程式碼

```
use std::sync::Arc;
use std::sync::atomic::{AtomicI32, Ordering};
use std::thread;

fn main() {
    let counter = Arc::new(AtomicI32::new(0));

    let mut handles = vec![];

    // 三個執行緒，每個加到不同的上限
    for limit in [100, 200, 300] {
        let counter_clone = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            for _ in 0..limit {
                counter_clone.fetch_add(1, Ordering::Relaxed);
            }
            println!("加了 {} 次，目前：{}", limit, counter_clone.load(Ordering::Relaxed));
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().expect("執行緒發生錯誤");
    }

    // 100 + 200 + 300 = 600，不管執行順序如何
    println!("最終結果：{}", counter.load(Ordering::Relaxed));
}
```

9.7.4 重點整理

- 有些 atomic 操作把讀、改、寫合成一個不可分割的動作，多個執行緒同時做也不會出錯。
- 常用型別：AtomicI32、AtomicUsize、AtomicBool。
- 常用方法：load（讀）、store（寫）、fetch_add（加並回傳舊值）。
- Ordering 控制記憶體排序，不確定就用 SeqCst。
- atomic 型別有 interior mutability——用 &self 就能修改，而且是 Sync（可以跨執行緒共享）。
- 只能用在簡單型別，複雜資料需要用鎖。

9.8 Mutex<T>

9.8.1 本集目標

學會用 Mutex<T> 讓多個執行緒安全地修改共享資料。

9.8.2 概念說明

9.8.2.1 想修改複雜的共享資料怎麼辦？

上一集學了 atomic，但它只能用在整數和布林等簡單型別。如果你想讓多個執行緒修改一個 Vec、String 或任何複雜的資料結構呢？

9.8.2.2 Mutex：多執行緒版的 interior mutability

Mutex<T> 和 RefCell 有些像——都提供一種 interior mutability，讓你在不需要 &mut 的情況下修改值。差別在於：

- **RefCell**：單執行緒，用普通整數做借用檢查。
- **Mutex**：多執行緒，用作業系統的鎖（lock）保護資料。

9.8.2.3 lock 和 MutexGuard

用 mutex.lock().expect("鎖失敗") 取得鎖。它會回傳一個 MutexGuard：

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(42);
    {
        let mut guard = m.lock().expect("鎖失敗");
        *guard += 1; // 透過 guard 修改值
        println!("{}", *guard); // 43
    } // guard 被 drop，自動解鎖
}
```

MutexGuard 實作了 Deref 和 DerefMut（第 5 章學的），所以它也是一種智慧指標——你可以直接把它當 &T 或 &mut T 使用。

同一時間只有一個執行緒能 lock 成功。其他執行緒呼叫 .lock() 時會**阻塞**（等待），直到持有鎖的執行緒把 guard drop 掉。

9.8.2.4 Arc + Mutex

實際上多半是這樣搭配使用——Arc 負責讓多個執行緒共享 Mutex，Mutex 負責保護裡面的資料：

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().expect("鎖失敗");
```

```
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().expect("執行緒發生錯誤");
    }

    println!("結果：{}", *counter.lock().expect("鎖失敗")); // 10
}
```

9.8.2.5 MutexGuard 不要活太久

guard 活著的期間，鎖都不會放開，其他執行緒全部在等。所以盡量縮短 guard 的生命週期：

```
// 不好：guard 活到作用域結束，鎖持有太久
let mut guard = mutex.lock().expect("鎖失敗");
*guard += 1;
// ... 做了很多不需要鎖的事情 ...
// guard 到後面才被 drop

// 好：用完就放
{
    let mut guard = mutex.lock().expect("鎖失敗");
    *guard += 1;
} // guard 立刻被 drop，鎖立刻釋放
// ... 做其他事情 ...
```

9.8.2.6 Mutex 把 Send 變成 Sync

第 3 集學了 Send 和 Sync。有些型別是 Send 但不是 Sync——例如第 4 集講的 RefCell<T>，它能安全地被 move 到另一個執行緒（Send），但不能讓多個執行緒同時透過 &RefCell<T> 存取（不是 Sync）。

Mutex 能解決這個問題。Mutex<T> 保證同一時間只有一個執行緒能存取 T——即使多個執行緒共享同一個 &Mutex<T>，也只有拿到鎖的那一個能操作裡面的 T。所以 Mutex<T> 只要求 T: Send，就能讓 Mutex<T> 本身成為 Sync。

換句話說：T 不是 Sync 沒關係，Mutex 的鎖機制已經確保不會有同時存取的問題。T 需要 Send 是因為：當執行緒 A 拿到鎖、操作完 T、放鎖之後，下一個拿到鎖的可能是執行緒 B。從 T 的角度來看，它原本被 A 獨佔使用，現在換成被 B 獨佔使用——效果等同於 T 從 A 被「送」到了 B。所以 T 必須是 Send。

9.8.3 範例程式碼

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for i in 0..5 {
```

```
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            // 縮小 guard 的作用域
            {
                let mut num = counter.lock().expect("鎖失敗");
                *num += 1;
                println!("執行緒 {} 把計數器改成 {}", i, *num);
            } // guard 在這裡就被 drop 了

            // 這裡已經不持有鎖了
            println!("執行緒 {} 做完了", i);
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().expect("執行緒發生錯誤");
    }

    println!("最終結果：{}", *counter.lock().expect("鎖失敗"));
}
```

9.8.4 重點整理

- Mutex<T> 是多執行緒版的 interior mutability，用鎖保護資料。
- .lock().expect(...) 回傳 MutexGuard，透過 DerefMut 直接當 &mut T 用。
- 同一時間只有一個執行緒能持有鎖，其他執行緒會等待。
- guard 被 drop 時自動解鎖。
- 常見搭配：Arc<Mutex<T>>——Arc 負責共享，Mutex 負責安全修改。
- MutexGuard 不要活太久，鎖住的期間其他執行緒全部在等。
- Mutex<T> 只要求 T: Send 就能是 Sync——Mutex 的鎖機制讓不是 Sync 的型別也能安全地被多個執行緒共享。

9.9 RwLock<T>

9.9.1 本集目標

學會用 RwLock<T> 實現讀寫分離的鎖，以及和 Mutex 的比較。

9.9.2 概念說明

9.9.2.1 Mutex 的限制

Mutex 不管你要讀還是要寫，都要鎖住。但很多時候多個執行緒只是要讀資料——讀和讀之間不會衝突，全部鎖住太浪費了。

9.9.2.2 RwLock：讀寫分離

RwLock<T> 區分讀鎖和寫鎖：

- **讀鎖**（.read().expect(...)）：多個執行緒可以**同時**持有讀鎖。
- **寫鎖**（.write().expect(...)）：寫鎖是獨佔的，持有寫鎖時不能有任何讀鎖或其他寫鎖。

「多個讀者同時」這件事一條執行緒示範不出來，本集最後的範例程式碼會用三條執行緒跑給你看。這裡先看兩種鎖怎麼取得：

```
use std::sync::RwLock;

fn main() {
    let lock = RwLock::new(42);

    // 讀鎖：只能看
    {
        let r = lock.read().expect("讀鎖失敗");
        println!("讀到 {}", *r);
    } // r 在這裡 drop，讀鎖放掉

    // 寫鎖：獨佔，可以改
    {
        let mut w = lock.write().expect("寫鎖失敗");
        *w += 1;
    } // w 在這裡 drop，寫鎖放掉

    println!("現在是 {}", *lock.read().expect("讀鎖失敗"));
}
```

9.9.2.3 guard 的行為

讀鎖回傳 RwLockReadGuard，寫鎖回傳 RwLockWriteGuard。跟 MutexGuard 一樣，它們也是智慧指標——可以直接操作內容，drop 時自動放鎖。

一樣要注意 guard 不要活太久。

9.9.2.4 和 RefCell 的對照

	RefCell	RwLock
執行緒	單執行緒	多執行緒
規則	多個 borrow() 或一個 borrow_mut()	多條執行緒的 read() 或一個 write()
檢查方式	執行時期，違反會 panic	作業系統的鎖，違反會阻塞等待

不過有一個 RefCell 沒有的陷阱：RefCell 可以在同一條執行緒裡 borrow() 好幾次，但 RwLock 說的「多個讀者」指的是**多條執行緒**。同一條執行緒重複拿同一把 RwLock 的讀鎖，標準庫明說**可能會 panic**；在某些平台上還可能直接卡住。

9.9.2.5 Mutex vs RwLock

什麼時候用哪個？

- **Mutex**：簡單、開銷小。適合讀寫都頻繁，或鎖持有時間很短的場景。大部分情況下 Mutex 就夠了。
- **RwLock**：在讀遠多於寫的時候有優勢，因為多個讀者可以同時進行。但鎖本身的開銷比 Mutex 大，而且有**寫者飢餓**（writer starvation）的風險——如果讀者一直源源不斷，寫者可能永遠拿不到鎖。

9.9.3 範例程式碼

```
use std::sync::{Arc, RwLock};
use std::thread;

fn main() {
    let data = Arc::new(RwLock::new(vec![1, 2, 3]));

    let mut handles = vec![];

    // 啟動 3 個讀者
    for i in 0..3 {
        let data = Arc::clone(&data);
        let handle = thread::spawn(move || {
            let read_guard = data.read().expect("讀鎖失敗");
            println!("讀者 {}：{:?}", i, *read_guard);
            // 多個讀者可以同時持有讀鎖
        });
        handles.push(handle);
    }

    // 啟動 1 個寫者
    {
        let data = Arc::clone(&data);
        let handle = thread::spawn(move || {
            let mut write_guard = data.write().expect("寫鎖失敗");
            write_guard.push(4);
            println!("寫者：寫入完成，現在是 {:?}", *write_guard);
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().expect("執行緒發生錯誤");
    }

    println!("最終結果：{:?}", *data.read().expect("讀鎖失敗"));
}
```

9.9.4 重點整理

- RwLock<T> 區分讀鎖和寫鎖：多條執行緒可以同時讀，寫者獨佔。
- .read().expect(...) 取得讀鎖，.write().expect(...) 取得寫鎖。
- guard 透過 Deref 操作內容，drop 時自動放鎖。
- 和 RefCell 的對照：RefCell 是單執行緒版本，RwLock 是多執行緒版本。
- 同一條執行緒不要重複拿同一把 RwLock 的讀鎖——那不是 RefCell 的 borrow()，可能 panic 或卡死。
- Mutex 簡單、開銷小，大部分情況夠用；RwLock 適合讀遠多於寫的場景，但開銷較大且有寫者飢餓的風險。

9.10 poisoning

9.10.1 本集目標

理解什麼是鎖的中毒（poisoning），以及該怎麼處理它。

9.10.2 概念說明

9.10.2.1 `.lock()` 為什麼回傳 `Result`

前面幾集學 `Mutex` 和 `RwLock` 的時候，我們都寫 `.lock().expect("鎖失敗")`。但什麼時候取鎖會「失敗」？答案就是 **poisoning**。

9.10.2.2 什麼是 poisoning

如果一個執行緒在持有 `Mutex` 鎖或 `RwLock` 寫鎖的期間 panic 了，鎖會被標記為「中毒」（poisoned）。之後任何執行緒再嘗試取鎖，`Mutex::lock` 或 `RwLock` 的 `read`、`write` 都會收到 `Err(PoisonError)`。

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let data = Arc::new(Mutex::new(vec![1, 2, 3]));
    let data2 = Arc::clone(&data);

    let handle = thread::spawn(move || {
        let mut guard = data2.lock().expect("鎖失敗");
        guard.push(4);
        panic!("哎呀！"); // panic 的時候 guard 還活著 → 鎖中毒
    });

    let _ = handle.join(); // 收集 panic，不讓它傳播

    // 之後再 lock → Err
    match data.lock() {
        Ok(guard) => println!("正常：{:?}", *guard),
        Err(_poisoned) => println!("鎖中毒了！"),
    }
}
```

9.10.2.3 為什麼要有 poisoning

panic 通常代表程式出了預期外的錯誤。如果一個執行緒在修改資料修到一半就 panic 了，資料可能是半成品——比如 `Vec` push 了一半、或某兩個欄位只更新了一個。poisoning 是一個安全機制：讓你知道出事了，由你決定要不要繼續用。

9.10.2.4 三種處理方式

1. 直接 panic（最常見）

```
use std::sync::Mutex;

fn main() {
    let data = Mutex::new(Vec::<i32>::new());
    let guard = data.lock().expect("鎖失敗");
```

```
}
```

如果鎖中毒了，你的執行緒也跟著 panic。大部分情況這樣就好——上一個執行緒 panic 了，通常代表整個程式該結束了。

2. 忽略中毒，繼續用

```
use std::sync::{Mutex, PoisonError};

fn main() {
    let data = Mutex::new(Vec::<i32>::new());
    let guard = data.lock().unwrap_or_else(PoisonError::into_inner);
}
```

`PoisonError::into_inner` 讓你拿回 guard，跳過中毒的警告。如果你確定資料的狀態沒問題，或者你不在意，可以這樣做。

要注意這是一個「**每次取鎖都這樣寫**」的做法：鎖本身從頭到尾都還是中毒狀態，你只是每一次都不理它。只要程式裡有一個地方忘了這樣寫、直接用了 `lock().expect(...)`，那裡就會 panic。

3. 修復資料再繼續

```
use std::sync::{Mutex, PoisonError};

fn main() {
    let data = Mutex::new(Vec::<i32>::new());
    let guard = match data.lock() {
        Ok(g) => g,
        Err(poisoned) => {
            let mut g = poisoned.into_inner();
            *g = vec![];          // 重設成已知的安全狀態
            data.clear_poison(); // 解除中毒，之後的 lock 才會恢復正常
            g
        }
    };
}
```

拿到 guard 之後把資料修復成合理的值，然後繼續用。

第 2 種做法可以一直不理中毒，是因為它每次取鎖都寫成同一個形式。但第 3 種做法的用意不一樣——它想把資料修好、然後回到正常運作。這時候只修資料是不夠的，程式裡其他地方老老實實寫的 `lock()` 還是會拿到 `Err`；修完之後要再呼叫 `clear_poison()`，才算真的解除中毒（`RwLock` 也有同名的方法）。

9.10.2.5　為什麼 `.into_inner()` 是安全的

你可能會好奇：中毒的鎖裡面的資料可能是半成品，存取它真的沒問題嗎？

從記憶體的角度來看是沒問題的。不管鎖有沒有中毒，裡面的資料都是合法的記憶體——不會存取到已經不能使用的記憶體、不會把型別搞混、沒有資料競爭。poisoning 保護的是**邏輯一致性**，不是**記憶體安全**。資料可能邏輯上不對，但從記憶體的角度看完全合法。所以可以安全地呼叫 `.into_inner()`。

9.10.2.6　`RwLock` 的 poisoning

`RwLock` 的 poisoning 只在**寫鎖** panic 的時候觸發。讀鎖 panic 不會中毒——因為讀的時候不會修改

資料，不會留下不一致的狀態。但一旦中毒了，read 和 write 都會回傳 Err。

9.10.3 範例程式碼

```rust
use std::sync::{Arc, Mutex, PoisonError};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));

    // 啟動一個會 panic 的執行緒
    let counter2 = Arc::clone(&counter);
    let handle = thread::spawn(move || {
        let mut guard = counter2.lock().expect("鎖失敗");
        *guard += 1;
        panic!("糟糕，出事了！");
    });

    // 等待那個執行緒結束（那個執行緒會 panic，但我們用 let _ 忽略）
    let _ = handle.join();

    // 嘗試取鎖——會收到 PoisonError
    match counter.lock() {
        Ok(guard) => {
            println!("正常取得鎖，值 = {}", *guard);
        }
        Err(poisoned) => {
            println!("鎖中毒了！");

            // 拿到資料看看
            let guard = poisoned.into_inner();
            println!("裡面的值 = {}", *guard);
        }
    }

    // 或者用一行忽略中毒
    let guard = counter.lock().unwrap_or_else(PoisonError::into_inner);
    println!("忽略中毒，值 = {}", *guard);
}
```

9.10.4 重點整理

- 執行緒在持有 Mutex 鎖或 RwLock 寫鎖期間 panic → 鎖中毒（poisoned）。
- 之後 lock / read / write 都回傳 Err(PoisonError)。
- RwLock 只有寫鎖 panic 才會中毒，讀鎖 panic 不會。
- PoisonError::into_inner 可以拿回 guard——記憶體安全沒問題，只是邏輯一致性的問題。
- 三種處理方式：
  - panic（.unwrap() 或 .expect()）。
  - 忽略（.unwrap_or_else(PoisonError::into_inner)）。
  - 修復資料再繼續（into_inner 不會解除中毒，修好之後要 clear_poison() 才算真的復原）。

9.11 mpsc

9.11.1 本集目標

學會用 channel 讓執行緒之間透過傳訊息溝通，以及和共享記憶體方式的比較。

9.11.2 概念說明

9.11.2.1 另一種思路

前面的 Mutex 和 RwLock 是「共享記憶體」的思路——多個執行緒存取同一份資料，用鎖來避免衝突。

channel 是完全不同的思路：**執行緒之間用傳訊息的方式溝通**。資料直接送過去，不共享。

9.11.2.2 建立 channel

`std::sync::mpsc::channel()` 建立一對發送端（tx）和接收端（rx）：

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel::<i32>();
}
```

mpsc 代表 **multiple producer, single consumer**——可以有多個發送端，但接收端只有一個。

9.11.2.3 發送和接收

`tx.send(value)` 把值送出去（會 move 值），`rx.recv()` 在另一端接收（會阻塞直到收到）：

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        tx.send(String::from("hello")).expect("發送失敗");
    });

    let received = rx.recv().expect("接收失敗");
    println!("收到：{}", received);
}
```

9.11.2.4 多個發送端

用 `tx.clone()` 產生新的發送端：

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    for i in 0..3 {
        let tx = tx.clone();
        thread::spawn(move || {
```

```
            tx.send(format!("來自執行緒 {}", i)).expect("發送失敗");
        });
    }

    drop(tx); // 原始的 tx 也要 drop，不然 rx 永遠不會結束

    for received in rx {
        println!("收到：{}", received);
    }
}
```

9.11.2.5 什麼時候結束

所有 tx 都被 drop 之後，rx.recv() 會先把還沒收的訊息全部收完，收完之後再呼叫 .recv() 才會回傳 Err。用 for msg in rx 迴圈也一樣——先把剩餘的訊息全部跑完，然後才結束。這是判斷「沒有人會再發送了，而且所有訊息都已經處理完了」的方式。

注意上面的例子裡的 drop(tx)——如果你 clone 了 tx 但沒有 drop 原始的 tx，接收端會認為還有發送端存活，永遠不會結束。

9.11.2.6 channel vs 共享記憶體

什麼時候用哪個？

- **多個執行緒需要反覆讀寫同一份資料**（例如共用的計數器、共用的快取）→ Mutex / RwLock 比較直接。
- **一邊產生資料、一邊消費資料**的流水線關係 → channel 更自然。資料的所有權直接轉移，不需要鎖，也不存在忘了放鎖的問題。

9.11.3 範例程式碼

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    // 啟動 3 個 worker，各自做一些計算後把結果送回來
    for i in 0..3 {
        let tx = tx.clone();
        thread::spawn(move || {
            let result = i * i;
            println!("執行緒 {} 計算完成：{}", i, result);
            tx.send((i, result)).expect("發送失敗");
        });
    }

    // drop 原始的 tx，這樣當所有 clone 都完成後，rx 迴圈會結束
    drop(tx);

    // 接收所有結果
    let mut total = 0;
    for (id, result) in rx {
        println!("主執行緒收到：執行緒 {} 的結果是 {}", id, result);
```

```
        total += result;
    }

    println!("所有結果的總和：{}", total);
}
```

9.11.4 重點整理

- channel 讓執行緒之間用傳訊息溝通，資料直接送過去，不共享。
- `mpsc::channel()` 建立發送端 `tx` 和接收端 `rx`。
- `tx.send(value)` 會 move `value`，`rx.recv()` 阻塞直到收到。
- `tx.clone()` 產生多個發送端，但接收端只有一個（mpsc）。
- 所有 `tx` 都 `drop` 之後，`rx` 的迴圈自動結束。
- 流水線關係用 channel，反覆讀寫同一份資料用 `Mutex` / `RwLock`。

9.12 死鎖

9.12.1 本集目標

理解死鎖是什麼、為什麼 Rust 的編譯器擋不住它、以及如何避免。

9.12.2 概念說明

9.12.2.1 什麼是死鎖

死鎖（deadlock）就是兩個或多個執行緒互相等待對方放鎖，結果誰都動不了，程式永遠卡住。

最經典的情況：執行緒 A 拿著鎖 1 等鎖 2，執行緒 B 拿著鎖 2 等鎖 1。兩邊永遠等下去。

9.12.2.2 程式碼示範

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let lock1 = Arc::new(Mutex::new(0));
    let lock2 = Arc::new(Mutex::new(0));

    let l1 = Arc::clone(&lock1);
    let l2 = Arc::clone(&lock2);

    let a = thread::spawn(move || {
        let _g1 = l1.lock().expect("鎖失敗"); // 拿到鎖 1
        // 假設這裡有一些延遲...
        let _g2 = l2.lock().expect("鎖失敗"); // 等待鎖 2
    });

    let l1 = Arc::clone(&lock1);
    let l2 = Arc::clone(&lock2);

    let b = thread::spawn(move || {
        let _g2 = l2.lock().expect("鎖失敗"); // 拿到鎖 2
        // 假設這裡有一些延遲...
```

```
        let _g1 = l1.lock().expect("鎖失敗"); // 等待鎖 1
    });

    // 如果時機剛好，程式會永遠卡在這裡
    a.join().expect("執行緒發生錯誤");
    b.join().expect("執行緒發生錯誤");
}
```

執行緒 A 先拿到鎖 1，然後想拿鎖 2。但鎖 2 被執行緒 B 拿走了，B 又在等鎖 1——結果誰都走不動。

9.12.2.3 編譯器不會擋死鎖

Send 和 Sync 保護的是**資料競爭**（data race）——多個執行緒同時存取資料造成的未定義行為。死鎖是**邏輯問題**，程式不會壞掉或出現未定義行為，只是永遠卡住。Rust 的編譯器無法偵測死鎖。

9.12.2.4 同一個執行緒也會死鎖

就算只有一個執行緒，對同一個 `Mutex` `lock` 兩次也有可能會死鎖——如果第一次 `lock` 還沒放開，第二次 `lock` 可能永遠等不到：

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(42);
    let _g1 = m.lock().expect("鎖失敗");
    let _g2 = m.lock().expect("鎖失敗"); // 可能死鎖：_g1 還握著鎖
}
```

9.12.2.5 如何避免

- **所有執行緒以相同順序取鎖**：如果每個人都先拿鎖 1 再拿鎖 2，就不會互相卡住。
- **減少同時持有多個鎖**：能用一個鎖解決就不要用兩個。
- **guard 不要活太久**：用完趕快 `drop`，縮短持有鎖的時間。

9.12.3 範例程式碼

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let lock1 = Arc::new(Mutex::new(String::from("資源 A")));
    let lock2 = Arc::new(Mutex::new(String::from("資源 B")));

    // 正確的方式：兩個執行緒以相同順序取鎖

    let l1 = Arc::clone(&lock1);
    let l2 = Arc::clone(&lock2);
    let a = thread::spawn(move || {
        let g1 = l1.lock().expect("鎖失敗"); // 先鎖 1
        let g2 = l2.lock().expect("鎖失敗"); // 再鎖 2
        println!("執行緒 A：{} 和 {}", *g1, *g2);
    });

    let l1 = Arc::clone(&lock1);
```

```
    let l2 = Arc::clone(&lock2);
    let b = thread::spawn(move || {
        let g1 = l1.lock().expect("鎖失敗"); // 也是先鎖 1
        let g2 = l2.lock().expect("鎖失敗"); // 再鎖 2
        println!("執行緒 B：{} 和 {}", *g1, *g2);
    });

    a.join().expect("執行緒發生錯誤");
    b.join().expect("執行緒發生錯誤");
    println!("沒有死鎖！");
}
```

9.12.4 重點整理

- 死鎖：多個執行緒互相等待對方放鎖，程式永遠卡住。
- Rust 的編譯器不會擋死鎖——Send / Sync 保護的是資料競爭，死鎖是邏輯問題。
- 同一個執行緒將同一個 Mutex 鎖兩次也有可能會死鎖，因為第一次的鎖還沒放開。
- 避免方法：統一取鎖順序、減少同時持有多個鎖、guard 用完趕快 drop。

9.13 thread::scope 簡介

9.13.1 本集目標

學會用 thread::scope 建立有限生命週期的執行緒，不需要 move 或 Arc 就能借用外部資料。

9.13.2 概念說明

9.13.2.1 thread::spawn 的限制

前面用 thread::spawn 的時候，閉包裡要用外面的變數都得 move 進去，或用 Arc 包起來。這是因為 spawn 出來的執行緒可能活得比呼叫它的函數還久——Rust 沒辦法保證資料在執行緒結束前不會被丟掉。

9.13.2.2 為什麼 spawn 不能借用

第 3 集看過 thread::spawn 的型別簽名，閉包和回傳值都要求 'static——必須活得跟整個程式一樣久。這就是為什麼你不能借用局部變數：局部變數的參考不是 'static 的。

9.13.2.3 thread::scope

thread::scope 解決了這個問題。它保證裡面 spawn 的所有執行緒在 scope 結束前都會被 join：

```
use std::thread;

fn main() {
    let data = vec![1, 2, 3, 4, 5];

    thread::scope(|s| {
        s.spawn(|| {
            println!("子執行緒：{:?}", data); // 直接借用，不需要 move
        });
    }); // 所有 scoped thread 在這裡保證已經結束

    // data 還能用
```

```
    println!("主執行緒：{:?}", data);
}
```

因為 scope 保證所有執行緒在 } 之前都跑完了，所以 data 不可能被提前丟掉——閉包可以安全地借用它，不需要 move 也不需要 Arc。

9.13.2.4 對比 spawn + Arc 的寫法

同一件事，用 thread::spawn 要這樣寫：

```
use std::sync::Arc;
use std::thread;

fn main() {
    let data = Arc::new(vec![1, 2, 3, 4, 5]);
    let data_clone = Arc::clone(&data);

    let handle = thread::spawn(move || {
        println!("{:?}", data_clone);
    });

    handle.join().expect("執行緒發生錯誤");
}
```

用 thread::scope 簡潔很多：

```
use std::thread;

fn main() {
    let data = vec![1, 2, 3, 4, 5];

    thread::scope(|s| {
        s.spawn(|| {
            println!("{:?}", data);
        });
    });
}
```

不需要 Arc、不需要 clone、不需要 move、不需要手動 join。

9.13.3 範例程式碼

```
use std::thread;

fn main() {
    let mut results = vec![];
    let input = vec![1, 2, 3, 4, 5];

    thread::scope(|s| {
        // 多個執行緒同時借用 input（不可變借用）
        let h1 = s.spawn(|| {
            let sum: i32 = input.iter().sum();
            sum
        });

        let h2 = s.spawn(|| {
```

```
            let max = input.iter().max().expect("空的 input");
            *max
        });

        let h3 = s.spawn(|| {
            let min = input.iter().min().expect("空的 input");
            *min
        });

        // scope 裡面也可以 join 拿回傳值
        results.push(h1.join().expect("執行緒發生錯誤"));
        results.push(h2.join().expect("執行緒發生錯誤"));
        results.push(h3.join().expect("執行緒發生錯誤"));
    });

    println!("input 還能用：{:?}", input);
    println!("總和 = {}, 最大 = {}, 最小 = {}", results[0], results[1], results[2]);
}
```

9.13.4 重點整理

- `thread::spawn` 要求 `'static`，所以閉包不能借用局部變數。
- `thread::scope` 保證所有 scoped 執行緒在 `scope` 結束前 `join`，因此可以安全借用外部資料。
- 不需要 `move`、不需要 `Arc`、不需要手動 `join`——程式碼簡潔很多。
- 當你只需要在一個區域內使用多執行緒，`thread::scope` 比 `thread::spawn` 方便。

恭喜你完成了多執行緒這一章！🎉 這一章從指標的底層概念出發，一路學到了執行緒、`Send` / `Sync`、`Arc`、`Mutex`、`RwLock`、poisoning、channel 和 `thread::scope`。多執行緒程式設計在很多語言裡是讓人頭痛的領域，但 Rust 的型別系統在編譯時期就幫你擋住了資料競爭——你不需要靠經驗和直覺來避免 bug，編譯器就是你最好的隊友。下一章我們將進入進階語言功能！

第 10 章

進階語言功能

本章會討論前面章節還沒有機會討論到的進階語言功能。

10.1 dyn Trait 基礎

10.1.1 本集目標

學會用 dyn Trait 在同一個位置存放不同型別的值，理解動態分派的原理。

10.1.2 概念說明

10.1.2.1 問題：不同型別放在同一個地方

第 5 章學了 impl Trait，可以寫 fn print_it(x: &impl Display) 讓函數接受任何實作 Display 的型別。但如果你想把不同型別的值放在同一個 Vec 裡呢？

```
trait Describe {
    fn describe(&self) -> String;
}

struct Cat;
struct Dog;

impl Describe for Cat {
    fn describe(&self) -> String {
        String::from("一隻貓")
    }
}

impl Describe for Dog {
    fn describe(&self) -> String {
        String::from("一隻狗")
    }
}
```

Cat 和 Dog 是不同型別——你沒辦法寫 Vec<impl Describe> 把它們放在一起。impl Trait 在編譯時期就決定了具體型別，而 Vec 裡的每個元素必須是同一個型別。

10.1.2.2 dyn Trait 登場

dyn Describe 代表「某個實作了 Describe 的型別，但具體是什麼我不知道」。

但既然不知道具體是什麼，dyn Describe 的大小就不固定——Cat 可能佔 1 byte，Dog 可能佔 100 bytes，編譯器在編譯時期不知道會是哪個。所以 dyn Describe 是 DST（附錄一的〈DST 簡介〉學過），必須放在指標後面：

- `&dyn Describe` — 借用。
- `Box<dyn Describe>` — 擁有

```
trait Describe {
    fn describe(&self) -> String;
}

struct Cat;
struct Dog;

impl Describe for Cat {
    fn describe(&self) -> String {
        String::from("一隻貓")
    }
}

impl Describe for Dog {
    fn describe(&self) -> String {
        String::from("一隻狗")
    }
}

fn main() {
    let animals: Vec<Box<dyn Describe>> = vec![
        Box::new(Cat),
        Box::new(Dog),
    ];

    for animal in &animals {
        println!("{}", animal.describe());
    }
}
```

同樣的道理，函數回傳也能用 `dyn Trait`：

```
fn make_animal(is_cat: bool) -> Box<dyn Describe> {
    if is_cat {
        Box::new(Cat)
    } else {
        Box::new(Dog)
    }
}
```

`impl Trait` 做不到這件事——因為 `if` 的兩個分支回傳不同型別，編譯器在編譯時期無法決定是哪一個。

10.1.2.3　胖指標：位址 + vtable

附錄一的〈DST 簡介〉學過 `&[T]` 是胖指標（位址 + 長度）。`&dyn Trait` 也是胖指標，但存的東西不同：

```
&[T]         = [資料位址][長度]
&dyn Trait   = [資料位址][vtable 指標]
```

vtable（虛擬方法表）是一張表，裡面存著這個具體型別對這個 `trait` 的所有方法的函數指標。`Cat` 的 vtable 裡有指向 `Cat::describe` 的指標，`Dog` 的 vtable 裡有指向 `Dog::describe` 的指標。

當你呼叫 animal.describe() 的時候，Rust 會去 vtable 裡查「describe 是哪個函數」，然後呼叫它。

```
use std::mem::size_of;

trait Describe {
    fn describe(&self) -> String;
}

fn main() {
    println!("{}", size_of::<&i32>());          // 8
    println!("{}", size_of::<&dyn Describe>()); // 16（位址 + vtable 指標）
    println!("{}", size_of::<&[i32]>());        // 16（位址 + 長度）
}
```

10.1.2.4 動態分派 vs 靜態分派

靜態分派（impl Trait / 泛型）：編譯器知道具體型別，為每個型別各生成一份函數的程式碼。這叫做 **monomorphization**（單態化）。呼叫方法時直接跳到對的函數，速度快，但如果型別很多，程式碼會變大。

```
use std::fmt::Display;

fn print_it(x: &impl Display) {
    println!("{}", x);
}

fn main() {
    print_it(&42);      // 編譯器生成 print_it::<i32>
    print_it(&"hello"); // 編譯器生成 print_it::<&str>
}
```

動態分派（dyn Trait）：編譯器只生成一份程式碼，執行時期透過 vtable 查找要呼叫的函數。程式碼只有一份，但每次呼叫多了一層 vtable 查找。

	靜態分派（impl Trait / 泛型）	動態分派（dyn Trait）
決定時機	編譯時期	執行時期
程式碼量	每個型別一份	只有一份
呼叫速度	快（直接呼叫）	稍慢（查 vtable）
能混合不同型別	不能	能

大部分情況用靜態分派就好。需要把不同型別放在同一個位置的時候才用 dyn Trait。

10.1.2.5 Box<dyn Fn()> vs impl Fn()

第 6 章學了閉包。Box<dyn Fn()> 讓你把不同的閉包統一成同一個型別：

```
fn main() {
    let callbacks: Vec<Box<dyn Fn()>> = vec![
        Box::new(|| println!("hello")),
        Box::new(|| println!("world")),
    ];

    for cb in &callbacks {
```

```
        cb();
    }
}
```

`Vec<impl Fn()>` 做不到，因為每個閉包是不同的匿名型別。

10.1.2.6 `dyn Trait` 的 lifetime bound

`dyn Trait` 後面可以加 lifetime bound，寫成 `dyn Trait + 'a`，讀成 `dyn (Trait + 'a)`——跟泛型裡的 `T: Trait + 'a` 意思一樣，dyn 把這個 bound 變成一個型別。

在某些位置，如果你沒寫 lifetime bound，編譯器會自動補上預設值。`Box<dyn Trait>` 的預設是 `'static`，所以完整寫法是 `Box<dyn Trait + 'static>`。`+ 'static` 代表裡面裝的具體型別不能包含任何非 `'static` 的參考。看看這個例子：

```
struct Foo<'a>(&'a str);

impl<'a> Describe for Foo<'a> {
    fn describe(&self) -> String { String::from(self.0) }
}

// 這個函數不會過編譯！
// Box<dyn Describe> = Box<dyn Describe + 'static>
// 但 Foo 借用了 s，s 不是 'static
fn make_box(s: &str) -> Box<dyn Describe> {
    Box::new(Foo(s))
}
```

如果需要裝有參考的型別，明確寫出 lifetime，覆蓋掉預設的 `'static`：

```
fn make_box<'a>(s: &'a str) -> Box<dyn Describe + 'a> {
    Box::new(Foo(s))
}
```

`&'a dyn Trait` 則預設是 `&'a (dyn Trait + 'a)`——比較不用特別處理。

10.1.2.7 `trait` upcasting

如果 `trait B` 是 `trait A` 的 subtrait（`trait B: A`），那 `dyn B` 可以轉成 `dyn A`：

```
trait Animal {
    fn name(&self) -> &str;
}

trait Pet: Animal {
    fn owner(&self) -> &str;
}

fn print_animal_name(a: &dyn Animal) {
    println!("{}", a.name());
}

fn example(pet: &dyn Pet) {
    print_animal_name(pet); // dyn Pet → dyn Animal，OK
}
```

`Pet` 一定是 `Animal`，所以 `dyn Pet` 當然可以當 `dyn Animal` 用。

10.1.3 範例程式碼

```
trait Describe {
    fn describe(&self) -> String;
}

struct Cat { name: String }
struct Dog { name: String }

impl Describe for Cat {
    fn describe(&self) -> String {
        format!("貓咪 {}", self.name)
    }
}

impl Describe for Dog {
    fn describe(&self) -> String {
        format!("狗狗 {}", self.name)
    }
}

fn make_animal(is_cat: bool, name: &str) -> Box<dyn Describe> {
    if is_cat {
        Box::new(Cat { name: String::from(name) })
    } else {
        Box::new(Dog { name: String::from(name) })
    }
}

fn main() {
    let animals: Vec<Box<dyn Describe>> = vec![
        Box::new(Cat { name: String::from("小花") }),
        Box::new(Dog { name: String::from("小黑") }),
        make_animal(true, "咪咪"),
        make_animal(false, "旺財"),
    ];

    for animal in &animals {
        println!("{}", animal.describe());
    }

    println!(
        "&dyn Describe 大小：{} bytes",
        std::mem::size_of::<&dyn Describe>()
    );
}
```

10.1.4 重點整理

- dyn Trait 代表「某個實作了 Trait 的型別，具體是什麼不知道」。
- dyn Trait 是 DST，必須放在指標後面：&dyn Trait、Box<dyn Trait>。
- &dyn Trait 是胖指標：資料位址 + vtable 指標。
- 動態分派（dyn Trait）透過 vtable 查找方法；靜態分派（impl Trait）編譯時期決定。
- 大部分情況用靜態分派，需要混合不同型別時才用 dyn Trait。

- `Box<dyn Fn()>` 可以把不同閉包統一成同一個型別。
- `Box<dyn Trait>` 在某些地方預設隱含 `+ 'static`；`dyn Trait + 'a` 讀成 `dyn (Trait + 'a)`，dyn 把 trait bound 變成型別。
- `dyn SubTrait` 可以轉成 `dyn SuperTrait`（trait upcasting）。

10.2 dyn compatibility

10.2.1 本集目標

理解哪些 trait 可以用 dyn、哪些不行，以及背後的原因。

10.2.2 概念說明

10.2.2.1 不是所有 trait 都能用 dyn

上一集學了 `dyn Trait`。但如果你嘗試寫 `dyn Clone`，會得到編譯錯誤。這是因為 `Clone` 不是 **dyn compatible** 的。

10.2.2.2 核心概念：`impl Trait for dyn Trait`

要理解 dyn compatibility，先想想 `dyn Trait` 是怎麼運作的。編譯器自動生成了一個：

```
impl Trait for dyn Trait {
    fn method(&self, ...) {
        // 查 vtable，呼叫實際的實作
    }
}
```

在這個自動生成的 `impl` 裡，`Self = dyn Trait`。而 `dyn Trait` 是 DST——大小不固定、不是 `Sized`。

如果 trait 的某些方法在 `Self = dyn Trait` 的情況下沒辦法運作，這個 trait 就不是 dyn compatible 的。具體來說有以下幾種情況：

10.2.2.3 限制一：`Self` 不能出現在 `self` 之外的型別中

```
trait Compare {
    fn compare(&self, other: &Self) -> bool;
}

impl Compare for Cat {
    fn compare(&self, other: &Cat) -> bool { ... }
}

impl Compare for Dog {
    fn compare(&self, other: &Dog) -> bool { ... }
}
```

compare 的第二個參數是 `&Self`。當你用 `dyn Compare` 的時候，具體型別已經被抹掉了——你不知道裡面是 Cat 還是 Dog。但 `Cat::compare` 期望的是 `&Cat`，`Dog::compare` 期望的是 `&Dog`。

如果有人傳了一個 Dog 進來，但 vtable 找到的函數是 `Cat::compare`，函數就會把 Dog 的資料當成 Cat 來用——型別搞混了。

要確保不搞混，編譯器就需要在執行時期檢查「傳進來的 y 的具體型別跟 x 的具體型別一樣」。但 dyn

的重點就是把具體型別抹掉了，編譯器已經不知道原本是什麼型別，沒辦法做這個檢查。所以 Rust 直接禁止這樣做。

10.2.2.4 限制二：方法不能有泛型參數

```
trait Converter {
    fn convert<U>(&self) -> U;
}
```

vtable 是一張固定大小的函數指標表。但泛型方法對每個不同的 `U` 都是一個不同的函數——`convert::<i32>` 和 `convert::<String>` 是兩個不同的函數指標。vtable 沒辦法塞進無限多個版本。

主要是 vtable 必須在編譯 `impl` 的那一方建好——因為只有那邊才知道 `Self` 的具體型別。但編譯 `impl` 的時候，你不知道使用者之後會用哪些 `U`，所以 vtable 不可能提前準備好所有版本。

10.2.2.5 限制三：`trait` 本身不能要求 `Self: Sized`

回頭看開頭的問題——為什麼 `dyn Clone` 不行？附錄一的〈DST 簡介〉已經提過，除了回傳 `Self`，`Clone` 還以 `Sized` 作為 supertrait：

```
trait Clone: Sized {
    fn clone(&self) -> Self;
}
```

`dyn Clone` 若要成立，編譯器就必須能產生前面提過的 `impl Clone for dyn Clone`。但 `Clone: Sized` 要求實作者必須是 `Sized`，而 `dyn Clone` 會是 DST。這個 `impl` 因此不可能成立，所以 `Clone` 不是 dyn compatible，也就不能形成 `dyn Clone`。

10.2.2.6 退出機制：`where Self: Sized`

如果一個 `trait` 只有一部分的方法是 dyn compatible，其他不是，可以在那些其他方法上全部加 `where Self: Sized` 讓它們退出：

```
trait MyTrait {
    fn normal(&self) -> String; // dyn MyTrait 上能呼叫
    fn special(&self) -> Self   // 回傳 Self，不 dyn compatible
        where Self: Sized;      // 加上這個，讓它退出
}
```

`where Self: Sized` 的意思是「只有 `Self` 是 `Sized` 的時候才能呼叫這個方法」。`dyn MyTrait` 不是 `Sized`，所以這個方法在 `dyn MyTrait` 上不能呼叫——但 `trait` 本身還是 dyn compatible 的，其他方法仍然能透過 `dyn MyTrait` 使用。

```
let x: &dyn MyTrait = &something;
x.normal(); // OK
// x.special(); // 編譯錯誤：dyn MyTrait 不是 Sized
```

dyn compatibility 的完整規則其實比這集講的更複雜，但八九不離十就是這些了。

10.2.3 範例程式碼

```
// dyn compatible 的 trait
trait Greet {
    fn greet(&self) -> String;
```

```rust
}

struct Alice;
struct Bob;

impl Greet for Alice {
    fn greet(&self) -> String { String::from("Hi, I'm Alice!") }
}

impl Greet for Bob {
    fn greet(&self) -> String { String::from("Hey, I'm Bob!") }
}

// 混合使用 where Self: Sized 的 trait
trait Animal {
    fn name(&self) -> &str;

    // 這個方法不 dyn compatible（回傳 Self），用 where Self: Sized 退出
    fn duplicate(&self) -> Self
    where
        Self: Sized + Clone;
}

#[derive(Clone)]
struct Cat { name: String }

impl Animal for Cat {
    fn name(&self) -> &str { &self.name }
    fn duplicate(&self) -> Self
    where
        Self: Sized + Clone,
    {
        self.clone()
    }
}

fn main() {
    // dyn Greet：不同型別放在同一個 Vec
    let greeters: Vec<Box<dyn Greet>> = vec![
        Box::new(Alice),
        Box::new(Bob),
    ];

    for g in &greeters {
        println!("{}", g.greet());
    }

    // dyn Animal：name() 能用，duplicate() 不能用
    let cat = Cat { name: String::from("小花") };
    let animal: &dyn Animal = &cat;
    println!("動物：{}", animal.name());  // OK
    // animal.duplicate(); // 編譯錯誤：dyn Animal 不是 Sized

    // 但用具體型別就能呼叫 duplicate
    let cat2 = cat.duplicate();
```

```
    println!("複製：{}", cat2.name());
}
```

10.2.4 重點整理

- 不是所有 trait 都能用 dyn——必須是 dyn compatible 的。
- 核心概念：編譯器自動生成 `impl Trait for dyn Trait`，`Self = dyn Trait`（DST）。
- `Self` 不能出現在 `self` 之外的型別中——具體型別已被抹掉。
- 方法不能有泛型參數——vtable 固定大小，放不下無限多版本。
- trait 不能要求 `Self: Sized`——`dyn Trait` 是 DST，不是 `Sized`。
- 個別方法加 `where Self: Sized` 可以讓它退出 dyn，trait 本身仍然 dyn compatible。

10.3 const fn

10.3.1 本集目標

學會用 `const fn` 定義編譯時期也能執行的函數，以及 `const { }` 區塊。

10.3.2 概念說明

10.3.2.1 問題：想用函數算 const 的值

第 2 章學了 const——編譯時期常數。但 const 的值只能用簡單的表達式：

```
const MAX: i32 = 100;        // OK
const DOUBLE: i32 = MAX * 2; // OK
```

如果你想用一個函數來算呢？

```
fn square(x: i32) -> i32 { x * x }
const VALUE: i32 = square(5); // 編譯錯誤！一般函數不能在編譯時期執行
```

10.3.2.2 const fn

在函數前面加上 const，它就變成編譯時期也能執行的函數：

```
const fn square(x: i32) -> i32 { x * x }
const VALUE: i32 = square(5); // OK！編譯時期算出 25
```

`const fn` 不是「只能在編譯時期用」——它在執行時期也能正常呼叫，就像普通函數一樣。它只是多了一個能力：**可以在編譯時期執行**。

```
const fn max(a: i32, b: i32) -> i32 {
    if a > b { a } else { b }
}

const BIGGER: i32 = max(10, 20); // 編譯時期：20

fn main() {
    let x = max(3, 7); // 執行時期：也能用，就是普通函數
    println!("{}", x);
    println!("{}", BIGGER);
}
```

10.3.2.3　限制

`const fn` 裡面不能做所有事情。基本原則是：**編譯器必須能在自己內部模擬執行這段程式碼**。

能做的：

- 算術、比較、邏輯運算
- `if`、`match`、`loop`、`while`。
- `let` 綁定（包括 `let mut`）。
- 建立 tuple、`struct`、`enum`。
- 呼叫其他 `const fn`。
- `panic!`（編譯時期 panic 會變成編譯錯誤）。

不能做的：

- 呼叫非 `const` 的函數。
- 輸入 / 輸出（`println!` 等）。
- 與作業系統互動
- inline assembly

Rust 每個版本都在放寬限制，能在 `const fn` 裡做的事越來越多。

10.3.2.4　const 區塊

`const { ... }` 可以在任何地方插入一段編譯時期運算，不需要定義 `const` 變數或 `const fn`：

```
fn main() {
    let x = const { 1 + 2 + 3 };
    println!("{}", x); // 6，在編譯時期就算好了
}
```

這在需要「就地」做編譯時期運算的時候很方便，不用另外定義一個 `const`。

10.3.3　範例程式碼

```
const fn factorial(n: u64) -> u64 {
    if n <= 1 {
        1
    } else {
        n * factorial(n - 1)
    }
}

const fn clamp(value: i32, min: i32, max: i32) -> i32 {
    if value < min {
        min
    } else if value > max {
        max
    } else {
        value
    }
}

const FACT_10: u64 = factorial(10);
const CLAMPED: i32 = clamp(150, 0, 100);
```

```
fn main() {
    println!("10! = {}", FACT_10);
    println!("clamp(150, 0, 100) = {}", CLAMPED);

    // 執行時期也能呼叫
    let n = factorial(5);
    println!("5! = {}", n);

    // const 區塊
    let size = const { std::mem::size_of::<[i32; 100]>() };
    println!("100 個 i32 的大小：{} bytes", size);
}
```

10.3.4 重點整理

- const fn 可以在編譯時期執行，也可以在執行時期執行。
- 主要用來初始化 const 的值。
- 限制：不能呼叫非 const fn、不能做輸入輸出，但限制逐版放寬。
- const { ... } 區塊可以在任何地方插入編譯時期運算。

10.4 associated const

10.4.1 本集目標

學會在 trait 和 impl 裡定義常數。

10.4.2 概念說明

10.4.2.1 trait 裡的 associated const

除了方法和 associated type，trait 裡也能定義常數：

```
trait HasLimit {
    const LIMIT: i32;
}

impl HasLimit for u8 {
    const LIMIT: i32 = 255;
}

impl HasLimit for i8 {
    const LIMIT: i32 = 127;
}
```

實作的時候必須指定值。使用時用 Type::CONST 的語法：

```
trait HasLimit {
    const LIMIT: i32;
}

impl HasLimit for u8 {
    const LIMIT: i32 = 255;
}
```

```rust
impl HasLimit for i8 {
    const LIMIT: i32 = 127;
}
fn main() {
    println!("u8：{}", u8::LIMIT); // 255
    println!("i8：{}", i8::LIMIT); // 127
}
```

10.4.2.2　associated const 可以有預設值

跟 trait 的預設方法一樣，associated const 也能有預設值：

```rust
trait Config {
    const TIMEOUT: u64 = 30;
    const RETRIES: u32 = 3;
}

struct MyApp;

impl Config for MyApp {
    const TIMEOUT: u64 = 60; // 覆蓋預設
    // RETRIES 用預設值 3
}

fn main() {}
```

10.4.2.3　impl 裡的 associated const

associated const 不一定要在 trait 裡——你也可以直接在 impl 區塊裡定義跟型別綁定的常數：

```rust
struct Circle;

impl Circle {
    const PI: f64 = 3.14159265358979;
}

fn main() {
    println!("PI = {}", Circle::PI);
}
```

這就像 associated function 一樣，用 :: 存取。

10.4.3　範例程式碼

```rust
trait Bounded {
    const LOWER: i32;
    const UPPER: i32;

    fn is_in_range(&self, value: i32) -> bool {
        value >= Self::LOWER && value <= Self::UPPER
    }
}

struct Percentage;

impl Bounded for Percentage {
```

```
    const LOWER: i32 = 0;
    const UPPER: i32 = 100;
}

struct Temperature;

impl Bounded for Temperature {
    const LOWER: i32 = -273;
    const UPPER: i32 = 1000;
}

// impl 裡的 associated const
struct Grid;

impl Grid {
    const WIDTH: usize = 80;
    const HEIGHT: usize = 24;
    const TOTAL: usize = Self::WIDTH * Self::HEIGHT;
}

fn main() {
    let p = Percentage;
    println!("50 在範圍內？{}", p.is_in_range(50));
    println!("150 在範圍內？{}", p.is_in_range(150));

    println!("溫度範圍：{} ~ {}", Temperature::LOWER, Temperature::UPPER);

    println!("Grid 大小：{}x{} = {}", Grid::WIDTH, Grid::HEIGHT, Grid::TOTAL);
}
```

10.4.4 重點整理

- trait 裡可以定義 const NAME: Type;，impl 時指定值。
- associated const 可以有預設值，impl 時可以覆蓋。
- impl 區塊（不在 trait 裡）也能定義 associated const，用 Type::CONST 存取。

10.5 const generics

10.5.1 本集目標

學會用常數值作為泛型參數，處理任意長度的陣列。

10.5.2 概念說明

10.5.2.1 問題：想寫處理任意長度陣列的函數

[i32; 3] 和 [i32; 5] 是不同型別——長度是型別的一部分。如果你想寫一個函數印出任意長度的陣列，難道每個長度都要寫一個？

10.5.2.2 const generics

泛型參數不只能是型別，也能是**常數值**：

```rust
fn print_array<const N: usize>(arr: [i32; N]) {
    for x in arr {
        println!("{}", x);
    }
}

fn main() {
    print_array([1, 2, 3]);        // N = 3
    print_array([10, 20, 30, 40]); // N = 4
}
```

`<const N: usize>` 宣告一個常數泛型參數 `N`，型別是 `usize`。跟型別參數 `<T>` 一樣，編譯器會為每個不同的 `N` 生成一份程式碼。

10.5.2.3　跟 slice 的差別

你可能會想：傳 `&[i32]` 不就好了？確實，如果只是要讀取一串資料，slice 更靈活。但 const generics 能做到 slice 做不到的事：

回傳固定長度的陣列：

```rust
fn zeros<const N: usize>() -> [i32; N] {
    [0; N]
}

fn main() {
    let a: [i32; 3] = zeros();
    let b: [i32; 10] = zeros();
}
```

slice 沒辦法回傳 `[T]`（DST），但 `[T; N]` 可以。

在型別層面保證長度：

```rust
fn add_arrays<const N: usize>(a: [i32; N], b: [i32; N]) -> [i32; N] {
    let mut result = [0; N];
    for i in 0..N {
        result[i] = a[i] + b[i];
    }
    result
}
```

兩個參數的長度在編譯時期就保證一致。slice 做不到。

10.5.2.4　用在 `struct` 上

```rust
struct Matrix<const ROWS: usize, const COLS: usize> {
    data: [[f64; COLS]; ROWS],
}
```

10.5.2.5　表達式語法

如果 const generic 的位置不是簡單的字面值或路徑，要用 `{}` 包起來：

```rust
fn example<const N: usize>() -> [i32; N] { [0; N] }

fn main() {
```

```
    let a = example::<3>();          // 字面值，不用 {}
    let b = example::<{ 1 + 2 }>(); // 表達式，要用 {}
}
```

10.5.2.6 搭配 const fn

前面學的 const fn 也能當 const generic 的值：

```
const fn double(n: usize) -> usize { n * 2 }

fn zeros<const N: usize>() -> [i32; N] { [0; N] }

fn main() {
    let c = zeros::<{ double(3) }>(); // [i32; 6]，const fn 當值
}
```

10.5.3 範例程式碼

```
fn sum<const N: usize>(arr: [i32; N]) -> i32 {
    let mut total = 0;
    for i in 0..N {
        total += arr[i];
    }
    total
}

fn filled<T: Copy, const N: usize>(value: T) -> [T; N] {
    [value; N]
}

fn main() {
    println!("sum([1, 2, 3]) = {}", sum([1, 2, 3]));
    println!("sum([10, 20]) = {}", sum([10, 20]));

    let ones: [i32; 5] = filled(1);
    println!("{:?}", ones);

    let hellos: [&str; 3] = filled("hello");
    println!("{:?}", hellos);

    // 表達式語法
    let zeros = filled::<i32, { 2 + 3 }>(0);
    println!("{:?}", zeros);
}
```

10.5.4 重點整理

- 泛型參數可以是常數值：<const N: usize>。
- 最常見的用途：處理任意長度的陣列 [T; N]。
- 跟 slice 的差別：const generics 能回傳固定長度陣列、在型別層面保證長度。
- 表達式要用 {} 包：Foo::<{ 1 + 2 }>。
- 可以搭配 const fn 使用。

10.6 預設參數

10.6.1 本集目標

學會型別參數和 const generic 參數可以在哪些地方設定預設值，以及如何定義。

10.6.2 概念說明

10.6.2.1 預設型別參數

在 `struct`、`enum`、`union`、型別別名和 `trait` 的宣告中，如果某個型別參數大部分情況都是同一種型別，就可以為它設定預設值。使用該型別或 `trait` 時若省略這個引數，就會套用預設值。函數和方法的泛型參數則不能有預設值。

以標準庫的 `PartialEq` 為例：

```
trait PartialEq<Rhs = Self> {
    fn eq(&self, other: &Rhs) -> bool;
}
```

`Rhs = Self` 表示：如果你不指定 `Rhs`，預設就是 `Self`。所以 `impl PartialEq for Point` 等同於 `impl PartialEq<Point> for Point`——比較的對象預設是自己。

如果偶爾想比較不同型別，覆蓋就行：

```
impl PartialEq<(i32, i32)> for Point {
    fn eq(&self, other: &(i32, i32)) -> bool {
        self.x == other.0 && self.y == other.1
    }
}
```

10.6.2.2 自己定義

用 `=` 在泛型定義裡給預設值：

```
struct Container<T = String> {
    value: T,
}

fn main() {
    let c: Container = Container { value: String::from("hello") }; // T 預設是 String
    let c2: Container<i32> = Container { value: 42 };              // 手動指定
}
```

10.6.2.3 const generics 的預設值

const generic 參數也能在上述型別與 `trait` 宣告中設定預設值，但同樣不能在函數或方法上設定。

```
struct Buffer<const N: usize = 1024> {
    data: [u8; N],
}

fn main() {
    let buf: Buffer = Buffer { data: [0; 1024] };     // N 預設是 1024
    let small: Buffer<64> = Buffer { data: [0; 64] }; // 手動指定
}
```

10.6.2.4 有預設值的參數必須放後面

```
struct Pair<T, U = T> { // OK：U 有預設值，放在 T 後面
    first: T,
    second: U,
}
```

10.6.3 範例程式碼

```
struct Pair<T, U = T> {
    first: T,
    second: U,
}

impl<T: std::fmt::Debug, U: std::fmt::Debug> Pair<T, U> {
    fn show(&self) {
        println!("({:?}, {:?})", self.first, self.second);
    }
}

fn main() {
    // U 用預設值 (= T = i32)
    let p1: Pair<i32> = Pair { first: 1, second: 2 };
    p1.show();

    // 手動指定 U
    let p2: Pair<i32, &str> = Pair { first: 42, second: "hello" };
    p2.show();
}
```

10.6.4 重點整理

- 型別參數可以在 `struct`、`enum`、`union`、型別別名和 `trait` 宣告中設定預設值。
- `const` generic 參數也能在相同的位置設定預設值。
- 函數和方法的泛型參數不能有預設值。
- 語法為 `<T = String>`、`<Rhs = Self>` 或 `<const N: usize = 1024>`。
- 不指定就套用預設，指定了就覆蓋。
- `PartialEq<Rhs = Self>` 是標準庫最典型的例子。
- 有預設值的參數必須放在沒有預設值的參數後面。

10.7 運算子重載

10.7.1 本集目標

學會幫自己的型別實作 `+`、`-` 等運算子。

10.7.2 概念說明

10.7.2.1 運算子就是 trait 方法

Rust 裡 `a + b` 其實是 `a.add(b)` 的簡寫——`+` 對應 `std::ops::Add` `trait`。幫你的型別實作 `Add`，就能用 `+`。

10.7.2.2 Add trait 的定義

```
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

三個重點：

- `Rhs = Self`：上一集學的預設參數，加法右邊預設和左邊同型別。
- `type Output`：第 5 章學的 associated type，加法的結果不一定跟輸入同型別。
- `self` 不是 `&self`：`add` 會消耗左邊的值（Copy 的型別不受影響）。

10.7.2.3 幫 Point 實作 Add

```
use std::ops::Add;

#[derive(Debug)]
struct Point { x: i32, y: i32 }

impl Add for Point {
    type Output = Point;
    fn add(self, rhs: Point) -> Point {
        Point {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}
```

10.7.2.4 常用運算子

`std::ops` 裡常用的 `trait`：

運算子	trait	方法
`+`	`Add`	`add(self, rhs)`
`-`	`Sub`	`sub(self, rhs)`
`*`	`Mul`	`mul(self, rhs)`
`/`	`Div`	`div(self, rhs)`
`%`	`Rem`	`rem(self, rhs)`
`-x`	`Neg`	`neg(self)`
`!x`	`Not`	`not(self)`
`&`	`BitAnd`	`bitand(self, rhs)`
`|`	`BitOr`	`bitor(self, rhs)`
`^`	`BitXor`	`bitxor(self, rhs)`
`<<`	`Shl`	`shl(self, rhs)`
`>>`	`Shr`	`shr(self, rhs)`
`+=`	`AddAssign`	`add_assign(&mut self, rhs)`
`&=`	`BitAndAssign`	`bitand_assign(&mut self, rhs)`
`[]`	`Index`	`index(&self, idx)`
`[]` 可變	`IndexMut`	`index_mut(&mut self, idx)`

位元運算子（`&`、`|`、`^`、`<<`、`>>`、`!`）在系統程式設計中很常用——處理旗標、遮罩、位元欄位等等。

如果你還不熟悉位元運算，建議自行查閱相關資料。

上面列的所有二元運算子都有對應的 assign 版本（例如 &= 對應 BitAndAssign、<<= 對應 ShlAssign），用法跟前面教過的 += 或 -= 類似。

10.7.2.5 AddAssign vs Add

a += b 和 a = a + b 在 Rust 裡的實作不一定一樣：

- Add::add(self, rhs) 消耗 a，產生新值。
- AddAssign::add_assign(&mut self, rhs) 就地修改 a。

對 i32 感覺差不多，但對非 Copy 型別（如 String），s1 += &s2 只要能可變借用 s1，s1 = s1 + &s2 則要先把 s1 的所有權交出去、拿到結果再指派回來。所有權的要求不同，有些型別的效率也可能不同，所以需要分開的 trait。

Add 和 AddAssign 是完全獨立的——實作了 Add 不代表 += 自動能用，反過來也是。沒實作就是編譯錯誤。

10.7.2.6 Index / IndexMut

Vec 能用 v[i] 就是因為它實作了 Index：

```
use std::ops::Index;

struct MyVec(Vec<i32>);

impl Index<usize> for MyVec {
    type Output = i32;
    fn index(&self, idx: usize) -> &i32 {
        &self.0[idx]
    }
}
```

10.7.2.7 不同型別相加

覆蓋 Rhs 的預設值：

```
use std::ops::Add;

struct Meters(f64);
struct Centimeters(f64);

impl Add<Centimeters> for Meters {
    type Output = Meters;
    fn add(self, rhs: Centimeters) -> Meters {
        Meters(self.0 + rhs.0 / 100.0)
    }
}
```

10.7.3 範例程式碼

```
use std::ops::{Add, Neg};

#[derive(Debug, Clone, Copy)]
struct Vec2 { x: f64, y: f64 }
```

```rust
impl Add for Vec2 {
    type Output = Vec2;
    fn add(self, rhs: Vec2) -> Vec2 {
        Vec2 { x: self.x + rhs.x, y: self.y + rhs.y }
    }
}

impl Neg for Vec2 {
    type Output = Vec2;
    fn neg(self) -> Vec2 {
        Vec2 { x: -self.x, y: -self.y }
    }
}

fn main() {
    let a = Vec2 { x: 1.0, y: 2.0 };
    let b = Vec2 { x: 3.0, y: 4.0 };
    let c = a + b;
    println!("a + b = {:?}", c);
    println!("-a = {:?}", -a);
}
```

10.7.4　重點整理

- `a + b` 是 `Add::add(a, b)` 的簡寫，其他運算子同理。
- `Add` 的簽名用了預設參數（`Rhs = Self`）和 associated type（`Output`）。
- `AddAssign`（`+=`）是就地修改（`&mut self`），`Add`（`+`）是產生新值（`self`）。
- `Index` / `IndexMut` 讓你的型別能用 `[]` 運算子。
- 覆蓋 `Rhs` 可以實現不同型別之間的運算。

10.8　型別轉換 as

10.8.1　本集目標

學會用 as 做數字型別轉換，以及更安全的替代方案。

10.8.2　概念說明

10.8.2.1　基本用法

第 1 章學過 Rust 不會自動轉型。需要轉換時用 as：

```rust
fn main() {
    let x: i32 = 42;
    let y: f64 = x as f64;

    let a: f64 = 3.99;
    let b: i32 = a as i32; // 3，截斷小數，不是四捨五入
}
```

10.8.2.2　整數之間

小轉大——值不會變：

```
    let x: u8 = 200;
    let y: u32 = x as u32; // 200
```

大轉小——**靜默截斷**，不會報錯：

```
    let x: u32 = 300;
    let y: u8 = x as u8; // 44！300 = 256 + 44，只留最低的 8 個位元
```

有號無號之間也可能出意外：

```
    let x: i32 = -1;
    let y: u32 = x as u32; // 4294967295
```

10.8.2.3 From / Into：更安全的選擇

第 5 章學過 From 和 Into。依照慣例，這兩個 trait 應該只為不會失敗的轉換實作：

```
fn main() {
    let x: i32 = 42;
    let y: f64 = f64::from(x); // OK
    let z: i32 = i32::from(3.14_f64); // 編譯錯誤！轉換可能遺失資訊
}
```

10.8.2.4 TryFrom / TryInto

可能失敗的轉換用 TryFrom——回傳 Result：

```
use std::convert::TryFrom;

fn main() {
    let x: u32 = 300;
    let result = u8::try_from(x); // Err
    let y: u32 = 42;
    let result = u8::try_from(y); // Ok(42)
}
```

10.8.2.5 什麼時候用哪個

- 不會失敗的轉換用 From / Into。
- 可能失敗用 TryFrom / TryInto——回傳 Result。
- as 只在你確實知道自己在做什麼的時候用。

10.8.3 範例程式碼

```
use std::convert::TryFrom;

fn main() {
    // as 基本轉換
    let x: i32 = 42;
    let y: f64 = x as f64;
    println!("i32 {} → f64 {}", x, y);

    let a: f64 = 3.99;
    let b: i32 = a as i32;
    println!("f64 {} → i32 {}（截斷，不是四捨五入）", a, b);
```

```
    // 危險的靜默截斷
    let big: u32 = 300;
    let small: u8 = big as u8;
    println!("u32 {} → u8 {}（靜默截斷！）", big, small);

    // From：安全
    let safe: f64 = f64::from(42_i32);
    println!("From：{}", safe);

    // TryFrom：可能失敗
    match u8::try_from(300_u32) {
        Ok(v) => println!("TryFrom 成功：{}", v),
        Err(e) => println!("TryFrom 失敗：{}", e),
    }

    match u8::try_from(42_u32) {
        Ok(v) => println!("TryFrom 成功：{}", v),
        Err(e) => println!("TryFrom 失敗：{}", e),
    }
}
```

10.8.4　重點整理

- as 做數字型別轉換：浮點轉整數截斷小數，大整數轉小整數靜默截斷。
- From / Into：依照慣例，用於不會失敗的轉換。
- TryFrom / TryInto：可能失敗的轉換，回傳 Result。
- 優先用 From，其次 TryFrom，最後才 as。

10.9　enum discriminant

10.9.1　本集目標

了解 enum variant 背後的整數值，以及如何自訂它。

10.9.2　概念說明

10.9.2.1　每個 variant 都有一個整數值

第 3 章學了 C-style enum。每個 variant 背後都有一個整數，叫做 **discriminant**。Rust 用它來區分目前是哪個 variant。

```
enum Color {
    Red,   // 0
    Green, // 1
    Blue,  // 2
}
```

預設從 0 開始，每個 variant 遞增 1。

10.9.2.2　用 as 取得 discriminant

上一集學了 as。C-style enum 可以用 as 轉成整數看到它的 discriminant：

```
enum Color {
    Red,   // 0
    Green, // 1
    Blue,  // 2
}

fn main() {
    println!("{}", Color::Red as i32);   // 0
    println!("{}", Color::Green as i32); // 1
    println!("{}", Color::Blue as i32);  // 2
}
```

10.9.2.3 自訂 discriminant

手動指定值：

```
enum HttpStatus {
    Ok = 200,
    NotFound = 404,
    InternalError = 500,
}

fn main() {
    println!("{}", HttpStatus::NotFound as i32); // 404
}
```

沒指定的 variant 從前一個 +1：

```
enum Level {
    Low = 1,
    Medium,    // 2
    High,      // 3
    Critical = 10,
    Emergency, // 11
}
```

10.9.2.4 #[repr] 控制底層型別

預設的底層型別由編譯器決定。用 #[repr] 明確指定：

```
#[repr(u8)]
enum Direction {
    North, // 0_u8
    South, // 1_u8
    East,  // 2_u8
    West,  // 3_u8
}
```

常見的選擇有 u8、u16、u32、i32 等。

10.9.2.5 帶資料的 enum 也有 discriminant

帶資料的 enum 內部也有 discriminant 來區分是哪個 variant，但你**不能用 as 取得它**：

```
enum Shape {
    Circle(f64),
    Rectangle(f64, f64),
```

```
}

fn main() {
    Shape::Circle(3.0) as i32; // 編譯錯誤！
}
```

10.9.3　範例程式碼

```
#[repr(u8)]
enum Command {
    Quit = 0,
    Move = 1,
    Write = 2,
    ChangeColor = 3,
}

enum Season {
    Spring = 1,
    Summer, // 2
    Autumn, // 3
    Winter, // 4
}

fn main() {
    println!("Quit = {}", Command::Quit as u8);
    println!("Write = {}", Command::Write as u8);

    println!("Spring = {}", Season::Spring as i32);
    println!("Winter = {}", Season::Winter as i32);
}
```

10.9.4　重點整理

- 每個 enum variant 都有一個整數 discriminant，預設從 0 遞增。
- C-style enum 可以用 as 轉成整數看到 discriminant。
- 手動指定值用 = 數字，沒指定的從前一個 +1。
- #[repr(u8)] 等控制底層型別。
- 帶資料的 enum 也有 discriminant，但不能用 as 取得。

10.10　attribute 總覽

10.10.1　本集目標

整理 Rust 常見的 attribute，理解 outer 和 inner 的差別。

10.10.2　概念說明

10.10.2.1　outer vs inner

- **outer attribute** #[...]：放在項目的上面，修飾那個項目。
- **inner attribute** #![...]：放在項目的裡面（通常是檔案開頭），修飾包含它的整個項目。

```rust
#![allow(dead_code)] // inner：修飾整個 mod

#[derive(Debug)]     // outer：修飾下面的 struct
struct Point { x: i32, y: i32 }
```

差一個驚嘆號！。

10.10.2.2 derive

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
struct Color(u8, u8, u8);
```

10.10.2.3 警告控制

```rust
#[allow(dead_code)]        // 不警告未使用的程式碼
#[allow(unused_variables)] // 不警告未使用的變數
#[warn(missing_docs)]      // 開啟「缺少文件」的警告
#[deny(unsafe_code)]       // 把「使用 unsafe」升級成錯誤
```

10.10.2.4 條件編譯

```rust
#[cfg(target_os = "windows")]
fn windows_only() { /* ... */ }

#[cfg(test)]
mod tests { /* ... */ }
```

10.10.2.5 測試

```rust
#[test]
fn test_add() { assert_eq!(1 + 1, 2); }

#[test]
#[should_panic]
fn test_panic() { panic!("故意的"); }

#[test]
#[ignore]
fn slow_test() { /* 暫時跳過 */ }
```

10.10.2.6 效能提示

呼叫函數的時候，程式需要跳到函數的位置去執行，跑完再跳回來。**inline** 是一種最佳化：編譯器把函數的程式碼直接「貼」到呼叫的地方，省掉跳來跳去的開銷。

```rust
#[inline]         // 建議編譯器 inline 這個函數
#[inline(always)] // 建議一律 inline
#[inline(never)]  // 建議禁止 inline
```

大部分時候不需要手動寫——編譯器會自己判斷。只有在跨 `crate` 呼叫的小函數、或效能很關鍵的地方才需要。

10.10.2.7　記憶體佈局

Rust 的編譯器會自由調整 `struct` 欄位在記憶體裡的排列順序和對齊方式來節省空間。但如果你要跟 C 語言互動，C 的 `struct` 有固定的排列規則，`#[repr(C)]` 就是告訴 Rust「用 C 的規則排列」：

```
#[repr(C)]  // 用 C 語言的記憶體佈局
#[repr(u8)] // enum 底層型別（上一集學過）
```

10.10.2.8　其他常用

`#[must_use]` 標記在函數或型別上，如果呼叫者拿到回傳值卻沒有使用，編譯器會警告。`Result` 就有 `#[must_use]`——這就是為什麼你不處理 `Result` 的時候會看到警告。

```
#[must_use]
fn compute() -> i32 { 42 }

fn main() {
    compute(); // 警告：回傳值沒有被使用
    let _ = compute(); // OK：用 let _ 明確忽略
}
```

```
#[non_exhaustive] // 告訴其他 crate 這個 enum / struct 未來可能加新的東西
#[deprecated]     // 標記已棄用
#[deprecated(since = "2.0", note = "請用 new_function")]
```

10.10.2.9　doc comment 是 attribute 的簡寫

```
/// 這是一個函數
fn foo() {}

// 等同於
#[doc = "這是一個函數"]
fn foo() {}
```

`///` 只是 `#[doc = "..."]` 的簡寫。同理，`//!` 是 `#![doc = "..."]` 的簡寫——用在檔案開頭，為整個 mod 或 crate 寫說明文件。

10.10.3　範例程式碼

```
#![allow(dead_code)]

#[derive(Debug, Clone, PartialEq)]
struct Config {
    name: String,
    value: i32,
}

#[must_use]
fn create_config(name: &str, value: i32) -> Config {
    Config { name: String::from(name), value }
}

#[deprecated(note = "請用 create_config")]
fn make_config() -> Config {
    create_config("default", 0)
```

```
}

#[cfg(target_os = "linux")]
fn linux_only() {
    println!("只在 Linux 上執行");
}

fn main() {
    let c = create_config("test", 42);
    println!("{:?}", c);
}
```

10.10.4 重點整理

- `#[...]`（outer）修飾下面的項目，`#![...]`（inner）修飾包含它的項目。
- `#[derive(...)]`：自動實作 `trait`。
- `#[allow/warn/deny(...)]`：控制警告。
- `#[cfg(...)]`：條件編譯。
- `#[test]` / `#[should_panic]` / `#[ignore]`：測試相關
- `#[must_use]`：忽略回傳值時警告。
- `#[deprecated]`：標記已棄用。
- `///` 是 `#[doc = "..."]` 的簡寫，`//!` 是 `#![doc = "..."]` 的簡寫。

10.11 cfg! 巨集

10.11.1 本集目標

學會用 `cfg!` 判斷編譯時期的條件並取得 `bool` 結果，以及它跟 `#[cfg]` 的差別。

10.11.2 概念說明

10.11.2.1 cfg! 回傳 bool

上一集學了 `#[cfg(...)]`——條件編譯，不符合條件的程式碼整塊被移除。但有時候你只是想根據條件走不同分支，不想移除整塊程式碼。`cfg!` 就是做這件事的：

```
fn main() {
    if cfg!(target_os = "windows") {
        println!("你在 Windows 上");
    } else {
        println!("你不在 Windows 上");
    }
}
```

10.11.2.2 跟 #[cfg] 的差別

	`#[cfg(...)]`	`cfg!(...)`
作用	條件編譯：整段程式碼移除或保留	在編譯時期展開成 `true` 或 `false`
程式碼檢查	不符合條件的程式碼會被移除，不接受檢查	兩個分支都會保留並接受檢查

`cfg!` 常放在一般的 `if` 裡，但條件已經在編譯時期決定，不是等到執行時期才判斷。兩個分支都會保留，而且必須通過編譯器的檢查。

這是很重要的差別：`#[cfg]` 不符合的那塊完全不存在，裡面就算有不存在的函數也不會報錯。但使用 `cfg!` 時，如果某一邊有編譯錯誤，不管條件成不成立都會報錯。

```
// #[cfg] 版：Windows 上不會編譯 linux_only()，不會報錯
#[cfg(target_os = "linux")]
fn linux_only() { /* Linux 特有功能 */ }

// cfg! 版：兩邊都會被編譯
if cfg!(target_os = "linux") {
    // linux_only(); // 如果函數不存在，在 Windows 上也會編譯錯誤！
}
```

10.11.2.3 常見條件

`#[cfg]` 和 `cfg!` 能用的條件一樣：

- `target_os = "windows"` / `"linux"` / `"macos"`
- `target_arch = "x86_64"` / `"aarch64"`。
- `debug_assertions` — debug 模式下為 `true`
- `feature = "my_feature"` — Cargo feature
- `test` — 在 `cargo test` 時為 `true`

10.11.3 範例程式碼

```
fn main() {
    if cfg!(debug_assertions) {
        println!("debug 模式");
    } else {
        println!("release 模式");
    }

    let os = if cfg!(target_os = "windows") {
        "Windows"
    } else if cfg!(target_os = "linux") {
        "Linux"
    } else if cfg!(target_os = "macos") {
        "macOS"
    } else {
        "其他"
    };
    println!("作業系統：{}", os);
}
```

10.11.4 重點整理

- `cfg!(...)` 會在編譯時期展開成 `bool` 常數；兩個分支都會保留並接受檢查。
- `#[cfg(...)]` 是條件編譯，不符合的程式碼整塊移除。
- 兩者能用的條件一樣：`target_os`、`debug_assertions`、`feature`、`test` 等。

10.12 macro_rules!

10.12.1 本集目標

學會用 macro_rules! 定義自己的宣告式巨集。

10.12.2 概念說明

10.12.2.1 巨集 vs 函數

我們從第 1 章就一直在用巨集——println!、vec!、format!、assert_eq!。呼叫時有個驚嘆號 !，這就是巨集和函數的區別。

巨集和函數最根本的差異：巨集在**編譯時期展開成程式碼**。你寫的巨集呼叫會在編譯的時候被替換成展開後的程式碼，然後編譯器再去編譯那段展開後的結果。巨集能產生任意的程式碼——定義新的函數、struct、甚至其他巨集呼叫。它也能接受型別名稱、模式等不是值的東西當參數。

但巨集也更難寫、更難讀、錯誤訊息比較差。能用函數就不要用巨集。

10.12.2.2 基本語法

```
macro_rules! say_hello {
    () => {
        println!("Hello!");
    };
}

fn main() {
    say_hello!(); // 印出 Hello!
}
```

結構是 (pattern) => { expansion }——左邊匹配，右邊展開。

10.12.2.3 帶參數

用 $name:kind 捕獲參數：

```
macro_rules! say {
    ($msg:expr) => {
        println!("{}", $msg);
    };
}

fn main() {
    say!("hi");
    say!(1 + 2);
}
```

常見的 kind：

- expr：表達式
- ty：型別
- ident：識別符（如變數名稱）
- tt：token tree（最靈活）

還有其他 kind，如果需要用到的話請自行搜尋。

10.12.2.4　多個分支

```
macro_rules! log {
    ($val:expr) => {
        println!("值：{}", $val);
    };
    ($name:expr, $val:expr) => {
        println!("{} = {}", $name, $val);
    };
}

fn main() {
    log!(42);           // 值：42
    log!("score", 100); // score = 100
}
```

10.12.2.5　重複匹配

`$( ... ),*` 的語法可以匹配重複的項目。拆開來看：

- `$( ... )` 裡面放要重複的模式。
- `,` 是分隔符號——每個重複項之間要有逗號。分隔符號不一定要是逗號，也可以用 `;` 等其他符號，或是省略不用。
- `*` 表示零個或更多個。也可以用 `+` 表示一個或更多個。

```
macro_rules! make_vec {
    ($($element:expr),*) => {
        {
            let mut v = Vec::new();
            $( v.push($element); )*
            v
        }
    };
}

fn main() {
    let v = make_vec![1, 2, 3];
}
```

展開的時候也用 `$( ... )*`——`$( v.push($element); )*` 會對每個捕獲的元素重複展開一次，變成：

```
v.push(1);
v.push(2);
v.push(3);
```

10.12.2.6　三種括號

巨集可以用三種括號呼叫，效果完全一樣：

- `macro!(...)` — 小括號，像函數呼叫。
- `macro![...]` — 中括號，像陣列（`vec![1,2,3]` 用這個）。
- `macro!{...}` — 大括號，像程式碼區塊。

差別只是慣例。

10.12.2.7 巨集的作用域

macro_rules! 定義的巨集在定義之後才能用（跟函數不同——函數不受定義順序限制）。

如果想讓巨集可以被其他 crate 使用，在前面加 #[macro_export]。在巨集內部引用定義巨集的 crate 的東西時，用 $crate 路徑——這樣不管使用者的 crate 怎麼命名你的 crate，路徑都能正確指向：

```
// 在 my_lib crate 裡

pub fn _log_impl(msg: &str) {
    println!("[LOG] {}", msg);
}

#[macro_export]
macro_rules! log_msg {
    ($msg:expr) => {
        $crate::_log_impl($msg);
    };
}
```

別的 crate 引入 my_lib 後，可以寫 my_lib::log_msg!("hello")，或先用 use my_lib::log_msg; 匯入，再寫 log_msg!("hello")。$crate 會自動指向定義這個巨集的 crate。

10.12.3 範例程式碼

```
macro_rules! max {
    ($a:expr, $b:expr) => {{
        let a = $a;
        let b = $b;
        if a > b { a } else { b }
    }};
}

macro_rules! print_all {
    ($($item:expr),*) => {
        $(
            println!("{}", $item);
        )*
    };
}

// stringify! 是內建巨集，把傳入的東西原樣變成字串
// stringify!(hello) 會變成 "hello"
macro_rules! create_fn {
    ($name:ident) => {
        fn $name() {
            println!("呼叫了函數 {}", stringify!($name));
        }
    };
}

create_fn!(hello);
create_fn!(world);
```

```rust
fn main() {
    println!("max(3, 7) = {}", max!(3, 7));

    print_all!["a", "b", "c"];

    hello();
    world();
}
```

10.12.4　重點整理

- 能用函數就不要用巨集。
- `macro_rules!` 定義宣告式巨集：`(pattern) => { expansion }`。
- 用 `$name:expr` 等接收參數，常見 kind：`expr`、`ty`、`ident`、`tt`。
- `$(...),*` 匹配重複項，展開時 `$( ... )*` 對每個重複。
- 三種括號 `()` / `[]` / `{}` 效果相同。
- 巨集定義後才能用（跟函數不同）。
- `#[macro_export]` 讓巨集可被其他 `crate` 使用。

10.13　proc macro

10.13.1　本集目標

認識三種 proc macro，理解它們的運作原理。這集只是大概介紹 proc macro 的概念和骨架，不會帶你實際寫一個完整的 proc macro。如果有需要，請自行搜尋相關教學。

10.13.2　概念說明

10.13.2.1　什麼是 proc macro

上一集的 `macro_rules!` 用模式匹配展開程式碼。但有些事情它做不到——例如讀取 `struct` 的欄位名稱來自動產生程式碼。`#[derive(Debug)]` 是怎麼知道你的 `struct` 有哪些欄位的？答案就是 **proc macro**（procedural macro）。

proc macro 拿到你的程式碼作為輸入（一串 token），然後產生新的程式碼（也是一串 token）。

10.13.2.2　`TokenStream`

proc macro 的輸入和輸出都是 `TokenStream`——Rust 程式碼的 token 序列。`struct Foo { x: i32 }` 進來的時候，proc macro 看到的是一串 token：`struct`、`Foo`、`{`、`x`、`:`、`i32`、`}`。

10.13.2.3　三種 proc macro

1. `derive` macro

搭配 `#[derive(...)]` 使用，最常見。

```rust
#[proc_macro_derive(MyDerive)]
pub fn my_derive(input: TokenStream) -> TokenStream {
    // input：被 #[derive(MyDerive)] 標記的 struct / enum 的程式碼
    // 回傳：要「附加」在旁邊的新程式碼（原始 struct / enum 不會被取代）
    TokenStream::new()
}
```

使用：`#[derive(MyDerive)] struct Foo { x: i32 }`

2. attribute macro

自訂 attribute。

```
#[proc_macro_attribute]
pub fn my_attr(attr: TokenStream, item: TokenStream) -> TokenStream {
    // attr：attribute 的參數
    // item：被標記的整個項目
    // 回傳：「取代」原本的項目
    item
}
```

使用：`#[my_attr(some_arg)] fn my_function() { ... }`

3. function-like macro

看起來像函數呼叫。

```
#[proc_macro]
pub fn my_macro(input: TokenStream) -> TokenStream {
    // input：括號裡的內容
    // 回傳：展開後的程式碼
    input
}
```

使用：`my_macro!(任何 token);`

10.13.2.4 三者的差別

- **derive**：**附加**新程式碼，不取代原本的 `struct` / `enum`。
- **attribute**：**取代**被標記的項目。
- **function-like**：括號裡的內容被**展開**成新的程式碼。

10.13.2.5 獨立 crate

proc macro 必須定義在獨立的 `crate` 裡，`Cargo.toml` 要加：

```
[lib]
proc-macro = true
```

10.13.2.6 syn 和 quote

實務上通常搭配兩個社群 `crate`：

- **syn**：把 `TokenStream` 解析成結構化的資料（例如知道「這是一個 `struct`，有一個欄位叫 `x`」）。
- **quote**：方便地從結構化資料生成 `TokenStream`。

沒有它們你就得自己一個一個 token 處理，非常痛苦。

10.13.3 範例程式碼

以下是三種 proc macro 的最小骨架（需要在獨立的 proc-macro `crate` 裡）：

```
use proc_macro::TokenStream;

// 1. derive macro
#[proc_macro_derive(MyDerive)]
```

```rust
pub fn my_derive(input: TokenStream) -> TokenStream {
    // 用 syn 解析 input，用 quote 生成程式碼
    TokenStream::new() // 什麼都不生成
}

// 2. attribute macro
#[proc_macro_attribute]
pub fn my_attr(_attr: TokenStream, item: TokenStream) -> TokenStream {
    item // 原封不動回傳
}

// 3. function-like macro
#[proc_macro]
pub fn my_macro(input: TokenStream) -> TokenStream {
    input // 原封不動回傳
}
```

以下是使用端的程式碼（在另一個 crate 裡）：

```rust
// 假設 proc-macro crate 叫 my_macros
use my_macros::{MyDerive, my_attr, my_macro};

#[derive(MyDerive)]
struct Foo { x: i32 }

#[my_attr]
fn hello() {
    println!("hello");
}

fn main() {
    hello();
    my_macro!(這裡可以放任何 token);
}
```

10.13.4 重點整理

- proc macro 分三種：derive、attribute、function-like。
- 本質是接收 TokenStream、回傳 TokenStream 的編譯時期函數。
- derive 附加程式碼、attribute 取代項目、function-like 展開內容。
- 必須在獨立 crate 裡定義（proc-macro = true）。
- 常用 syn（解析）和 quote（生成）兩個 crate。

10.14 unsafe

10.14.1 本集目標

理解 unsafe 的意義、能做什麼、以及寫 unsafe 程式碼時該注意什麼。

10.14.2 概念說明

10.14.2.1 為什麼需要 unsafe

Rust 的安全保證建立在一些**假設**上——例如 `&mut T` 一定是獨佔的、參考一定指向有效的資料。編譯器會幫你檢查這些假設是否成立。

但有些操作是編譯器無法驗證的。Rust 不是不讓你做這些事，而是要你明確說「這段我自己負責」——這就是 `unsafe`。

10.14.2.2 Rust 的安全保證

safe Rust 保證以下這些事情**不會發生**，不管你的程式碼怎麼寫：

- 不會存取到已經被釋放的記憶體。
- 不會有資料競爭（多個執行緒同時讀寫且至少一方在寫）。
- 不會有懸垂參考。
- 不會同一個值被 `drop` 兩次。
- 不會讀到未初始化的記憶體。
- 不會把型別搞混（例如把 `isize` 的 bytes 當成指標來讀）。

違反上面任何一條，就叫做**未定義行為**（undefined behavior，簡稱 UB）。它比「結果不確定」嚴重得多：編譯器是**假設這些事永遠不會發生**才做最佳化的，所以一旦發生，壞掉的不只是那一行——整個程式都不再有任何保證，可能算出怪答案、可能當掉，也可能在完全無關的地方出事。

`unsafe` 程式碼的責任就是：即使繞過了編譯器的檢查，也必須確保這些保證**全部成立**。

10.14.2.3 unsafe 區塊

把需要 `unsafe` 操作的程式碼包在 `unsafe { }` 裡。`unsafe` 不是「關掉所有檢查」——借用規則、型別檢查在 `unsafe` 區塊裡照常運作。`unsafe` 只是多開放幾種特定操作。

10.14.2.4 五種 unsafe 操作

1. **解參考原始指標**（`*const T`、`*mut T`）
2. **呼叫 `unsafe` 函數**
3. **手動實作 `unsafe trait`**
4. **存取 `static mut` 變數**
5. **存取 `union` 的欄位**

10.14.2.5 原始指標

原始指標是沒有借用規則保護的指標。**建立**不需要 `unsafe`，**使用**（解參考）才需要：

```
fn main() {
    let x = 42;
    let ptr: *const i32 = &raw const x; // 建立：不需要 unsafe

    let value = unsafe { *ptr }; // 解參考：需要 unsafe
    println!("{}", value); // 42
}
```

你也可以用 `as` 從參考轉成原始指標：

```
    let x = 42;
    let ptr = &x as *const i32; // &i32 轉成 *const i32
```

但 `&raw const x` 和 `&raw mut x` 更好——它們直接從變數拿到原始指標，不經過建立參考。有時候光是建立參考本身就可能違反規則（例如對未初始化的記憶體取 &），用 `&raw` 就沒有這個問題。

10.14.2.6 `unsafe fn`

如果一個函數的安全性需要呼叫者自己保證，標記成 `unsafe fn`：

```
unsafe fn dangerous(ptr: *const i32) -> i32 {
    unsafe { *ptr }
}

fn main() {
    let x = 42;
    let value = unsafe { dangerous(&raw const x) };
}
```

注意：在 Rust 2024 edition 後，即使在 `unsafe fn` 裡面，做 unsafe 操作也要寫 `unsafe { }` 區塊——讓每個 unsafe 操作都被明確標出。

10.14.2.7 `unsafe trait`

有些 `trait` 的正確實作需要滿足編譯器沒辦法自動檢查的條件：

```
unsafe trait MyGuarantee {
    fn check(&self) -> bool;
}

unsafe impl MyGuarantee for i32 {
    fn check(&self) -> bool { *self >= 0 }
}
```

`unsafe trait` 的意思是：「實作這個 `trait` 必須滿足某些編譯器沒辦法檢查的條件。」實作時用 `unsafe impl`，表示你保證那些條件成立。

Send 和 Sync 就是 `unsafe trait`——適合的情況下，編譯器會自動實作；但如果你手動實作，就必須自己保證多執行緒下的安全性。

注意：**呼叫** `unsafe trait` 的方法不需要 `unsafe`——危險的是實作，不是使用。

10.14.2.8 `unsafe` 的邊界

`unsafe` 程式碼必須保證：**不管被什麼 safe code 呼叫，都不會造成未定義行為。**

例如標準庫的 `Vec`：內部用 `unsafe` 管理記憶體，但對外提供 safe 的 API。不管你怎麼用 `Vec` 的 safe API，都不可能觸發未定義行為。

10.14.2.9 寫 `unsafe` 程式碼的注意事項

- **盡量縮小 `unsafe` 區塊**——只包住真正需要 `unsafe` 的那幾行。
- **寫 `// SAFETY:` 註解**——解釋為什麼這段 `unsafe` 操作是正確的。
- **注意借用規則**——即使用原始指標，「`&mut` 必須獨佔」等規則在語意上仍然有效。
- **維護型別的不變量**——例如 `String` 一定是合法 UTF-8、`bool` 一定是 0 或 1。
- **考慮 panic safety**——如果 `unsafe` 區塊裡有可能 panic 的操作，確保 panic 後資料結構仍然合法。
- **用 Miri 測試**——`cargo +nightly miri test` 可以偵測很多 `unsafe` 的問題。

10.14.2.10 常見用途

- 實作資料結構（連結串列、Vec 的內部）
- 跟 C 語言互動
- 效能關鍵部分

10.14.3 範例程式碼

```
fn main() {
    // 原始指標
    let mut x = 42;
    let ptr_const: *const i32 = &raw const x;
    let ptr_mut: *mut i32 = &raw mut x;

    unsafe {
        println!("讀取：{}", *ptr_const);
        *ptr_mut = 100;
        println!("修改後：{}", *ptr_mut);
    }

    // unsafe fn
    unsafe fn add_one(ptr: *mut i32) {
        unsafe { *ptr += 1; }
    }

    let mut val = 10;
    // SAFETY: ptr 指向有效的、已初始化的 i32，且沒有其他參考
    unsafe { add_one(&raw mut val); }
    println!("val = {}", val);
}
```

10.14.4 重點整理

- unsafe 讓你做編譯器無法驗證的操作，但不是關掉所有檢查。
- 五種 unsafe 操作：解參考原始指標、呼叫 unsafe fn、實作 unsafe trait、存取 static mut、存取 union 欄位。
- 原始指標 *const T / *mut T：沒有借用規則保護的指標，不保證指向有效的資料。建立不需要 unsafe，解參考需要。
- &raw const x / &raw mut x：直接拿原始指標，不經過參考。
- unsafe fn 在 2024 edition 後也要寫 unsafe { } 區塊。
- unsafe trait 的危險在實作，不在使用（呼叫方法不需要 unsafe）。
- **未定義行為（UB）**：違反了 safe Rust 那些保證。編譯器是假設它永遠不會發生才做最佳化的，所以一旦發生，整個程式都不再有保證。
- unsafe 程式碼的邊界：不管被什麼 safe code 呼叫都不能造成未定義行為。

10.15 static 變數

10.15.1 本集目標

了解 static 和 const 的差別，以及為什麼幾乎不該用 static mut。

10.15.2 概念說明

10.15.2.1 static vs const

第 2 章學了 const——編譯時期常數，值被直接嵌進使用它的地方。static 看起來很像，但有一個根本差異：**static 變數有固定的記憶體位址**。

```
static GREETING: &str = "Hello, world!";
static MAX_SIZE: usize = 1024;
```

大部分情況 const 就夠了。需要固定記憶體位址（例如傳給 C 函數）的時候才用 static。

10.15.2.2 static mut

Rust 允許可變的 static——但讀寫都需要 unsafe：

```
static mut COUNTER: i32 = 0;

fn increment() {
    unsafe { COUNTER += 1; }
}
```

為什麼需要 unsafe？因為 static 是全域共享的，多個執行緒同時讀寫就是資料競爭。

static mut 幾乎永遠不該用。現代 Rust 有更好的替代：

- 簡單的計數器 → AtomicI32、AtomicBool。
- 複雜的可變全域狀態 → Mutex<T>（搭配 static）。
- 延遲初始化 → LazyLock（下一集教）。

10.15.3 範例程式碼

```
use std::sync::atomic::{AtomicI32, Ordering};

// const：值嵌進使用的地方
const MAX: i32 = 100;

// static：有固定位址
static GREETING: &str = "Hello!";

// 用 atomic 取代 static mut
static COUNTER: AtomicI32 = AtomicI32::new(0);

fn increment() {
    COUNTER.fetch_add(1, Ordering::Relaxed);
}

fn main() {
    println!("{}", GREETING);
    println!("MAX = {}", MAX);

    increment();
    increment();
    increment();
    println!("COUNTER = {}", COUNTER.load(Ordering::Relaxed));
}
```

10.15.4 重點整理

- `static` 有固定記憶體位址，整個程式共用一份。
- `const` 沒有固定位址，值被嵌入使用的地方；大部分情況用 `const` 就好。
- `static mut` 讀寫都需要 `unsafe`，幾乎永遠不該用。
- 替代方案：`AtomicXxx`、`Mutex<T>`、`LazyLock`。

10.16 LazyLock

10.16.1 本集目標

學會用 `LazyLock` 延遲初始化全域變數。

10.16.2 概念說明

`LazyLock` 嚴格來說是標準庫提供的工具，不算語言功能。但因為上一集剛學了 `static`，它又是搭配 `static` 最常用的東西，所以一併在這裡介紹。

10.16.2.1 問題：static 的值必須在編譯時期確定

`static` 的值必須在編譯時期就算出來。空的 `Vec::new()` 可以（因為它是 `const fn`，不需要配置記憶體），但如果你想要一個**已經有內容**的 `Vec` 呢？

```
// vec! 巨集和 String::from 都需要在執行時期配置記憶體
static NAMES: Vec<String> = vec![String::from("Alice"), String::from("Bob")];
```

那怎麼辦？既然沒辦法在編譯時期給值，那就**先不給**——等到程式執行時第一次用到的時候再初始化。這就是延遲初始化。

10.16.2.2 LazyLock

`std::sync::LazyLock` 就是做這件事的——你給它一個閉包，它會在第一次存取時才執行閉包產生值，之後都用快取的結果。`LazyLock` 實作了 `Deref`，所以你可以直接把它當成裡面的值來用，跟 `Box`、`Rc` 等智慧指標一樣：

```
use std::sync::LazyLock;

static NAMES: LazyLock<Vec<String>> = LazyLock::new(|| {
    vec![String::from("Alice"), String::from("Bob")]
});

fn main() {
    println!("{:?}", *NAMES); // 第一次：執行閉包
    println!("{}", NAMES[0]); // 之後：直接用快取
}
```

10.16.2.3 為什麼叫 LazyLock

- **Lazy**：不到用的時候不初始化。
- **Lock**：內部有鎖，多個執行緒同時存取時只會初始化一次（thread-safe）。

10.16.3　範例程式碼

```rust
use std::sync::LazyLock;

static NAMES: LazyLock<Vec<String>> = LazyLock::new(|| {
    println!("初始化 NAMES！");
    vec![String::from("Alice"), String::from("Bob"), String::from("Charlie")]
});

fn print_first() {
    println!("第一個名字：{}", NAMES[0]);
}

fn main() {
    println!("程式開始");
    print_first(); // 第一次存取，這時才會初始化
    print_first(); // 第二次存取，直接用快取
    println!("共 {} 個名字", NAMES.len());
}
```

10.16.4　重點整理

- `static` 的初始值必須能在編譯時期求值；建立有內容的 `Vec`、呼叫 `String::from` 等操作需要在執行時期執行。
- `LazyLock` 延遲到第一次存取才初始化，之後用快取。
- `LazyLock` 是 thread-safe 的，可以安全地用在 `static`。

10.17　`extern` blocks

10.17.1　本集目標

學會呼叫 C 函數和讓 C 呼叫 Rust 函數。這集只是大概介紹 FFI 的功能。如果你想要一個完整的 FFI 例子（從建立 C 函式庫到在 Rust 裡呼叫），請自行搜尋相關教學。

10.17.2　概念說明

10.17.2.1　FFI 是什麼

FFI（Foreign Function Interface）是讓不同程式語言互相呼叫函數的機制。Rust 可以呼叫 C 寫的函數，C 也可以呼叫 Rust 寫的函數。因為幾乎所有語言都能跟 C 互通，所以 Rust 透過 C 這個橋樑，就能跟大部分語言互動。

10.17.2.2　呼叫 C 函數

用 `unsafe extern "C"` 區塊宣告外部的 C 函數：

```rust
unsafe extern "C" {
    fn fabs(x: f64) -> f64;
}

fn main() {
    let result = unsafe { fabs(-42.0) };
    println!("fabs(-42.0) = {}", result);
}
```

呼叫外部函數需要 unsafe——因為 Rust 沒辦法檢查 C 那邊的函數是不是安全的。

在 Rust 2024 edition 後，extern 區塊本身也需要 unsafe——因為你在宣告裡寫的函數簽名（參數型別、回傳型別等）是否正確，Rust 沒辦法驗證。如果簽名跟 C 那邊實際的不一樣，就會導致未定義行為。

10.17.2.3 safe fn

如果你確定某個外部函數是安全的，可以標記 safe：

```
unsafe extern "C" {
    safe fn fabs(x: f64) -> f64; // fabs 對所有 f64 輸入都是安全的
}

fn main() {
    let result = fabs(-42.0); // 不需要 unsafe 也能呼叫！
    println!("fabs(-42.0) = {}", result);
}
```

10.17.2.4 "C" 是什麼

extern "C" 的 "C" 指的是 **ABI**（Application Binary Interface）——函數在二進位層面的呼叫方式。"C" 是最常用的 ABI，幾乎所有語言都能跟 C ABI 互通。

10.17.2.5 讓 C 呼叫 Rust

```
#[unsafe(no_mangle)]
pub extern "C" fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

- extern "C"：用 C ABI。
- #[unsafe(no_mangle)]：不要混淆函數名稱，讓 C 能用 add 找到它。在 2024 edition 中，no_mangle 是 unsafe attribute，因為它改變了函數的連結方式，可能影響安全性。

10.17.2.6 extern 區塊裡也能宣告 static 變數

```
unsafe extern "C" {
    static daylight: i32; // C 那邊的全域變數
}
```

10.17.3 範例程式碼

```
unsafe extern "C" {
    safe fn fabs(x: f64) -> f64;
    fn sqrt(x: f64) -> f64;
}

#[unsafe(no_mangle)]
pub extern "C" fn rust_add(a: i32, b: i32) -> i32 {
    a + b
}
```

```
fn main() {
    // 標記 safe 的函數不需要 unsafe
    println!("fabs(-10.0) = {}", fabs(-10.0));

    // 沒標記 safe 的需要 unsafe
    let root = unsafe { sqrt(25.0) };
    println!("sqrt(25.0) = {}", root);

    // Rust 的 extern "C" 函數也能在 Rust 裡直接呼叫
    println!("rust_add(3, 4) = {}", rust_add(3, 4));
}
```

10.17.4 重點整理

- `unsafe extern "C" { ... }` 宣告外部 C 函數。
- 呼叫外部函數需要 `unsafe`；標記 `safe fn` 的除外。
- `"C"` 是 ABI，指定函數在二進位層面的呼叫方式。
- `#[unsafe(no_mangle)] pub extern "C" fn` 讓 C 可以呼叫 Rust。
- `extern` 區塊裡也能宣告 `static` 變數。

10.18 union

10.18.1 本集目標

認識 union——所有欄位共享同一塊記憶體。

10.18.2 概念說明

10.18.2.1 什麼是 union

`struct` 的每個欄位各佔一塊記憶體。`union` 不一樣——**所有欄位共享同一塊記憶體**：

```
union IntOrBool {
    i: i32,
    b: bool,
}
```

`IntOrBool` 的大小是 4 bytes，足以容納 4 bytes 的 `i32` 或 1 byte 的 `bool`。`i` 和 `b` 佔的是同一塊記憶體——寫入 `i` 會覆蓋 `b` 的內容。

10.18.2.2 寫入不需要 unsafe，讀取需要

```
    let u = IntOrBool { i: 1 };
    let value = unsafe { u.i }; // 讀取需要 unsafe
```

為什麼讀取需要 `unsafe`？因為 Rust 不知道你上次寫入的是哪個欄位。`bool` 在記憶體中**必須是 0 或 1**。如果你用 `i` 寫入 42，再用 `b` 讀出來，那塊記憶體的內容是 42——對 `bool` 來說不是有效的值，這是**未定義行為**。讀取 union 欄位時，你必須自己保證記憶體裡的內容對你要讀的型別是有效的——編譯器檢查不了這件事，所以要 `unsafe`。

10.18.2.3 跟 enum 的差別

	enum	union
知道目前是哪個 variant	有 discriminant	不知道，你自己追蹤
讀取	安全	需要 unsafe

10.18.2.4 用途：FFI

union 在純 Rust 裡幾乎用不到——enum 更安全也更好用。union 存在的主要原因是跟 C 語言互動：C 有 union，你需要 Rust 版的 union 來對應它的記憶體佈局。

10.18.3 範例程式碼

```
union IntOrBool {
    i: i32,
    b: bool,
}

fn main() {
    // 寫入不需要 unsafe
    let u = IntOrBool { b: true };

    // 讀取需要 unsafe
    unsafe {
        // 寫 b 讀 b，沒問題
        println!("b = {}", u.b);
    }

    let v = IntOrBool { i: 42 };
    unsafe {
        println!("i = {}", v.i);
        // 千萬不要這樣做：
        // println!("b = {}", v.b);
        // bool 必須是 0 或 1，但這塊記憶體是 42 → 未定義行為！
    }

    // IntOrBool 的大小是 4 bytes
    println!("size: {} bytes", std::mem::size_of::<IntOrBool>()); // 4
}
```

10.18.4 重點整理

- union 的所有欄位共享同一塊記憶體。
- 寫入不需要 unsafe，讀取需要——因為 Rust 不知道裡面存的是哪個欄位。
- 讀取時你必須保證記憶體內容對該型別有效——bool 必須是 0 或 1，寫入 42 後讀 b 是未定義行為。
- 跟 enum 不同：union 沒有 discriminant，不追蹤目前是哪個 variant。
- 主要用途是 FFI（跟 C 語言的 union 對應）。

10.19 never type !

10.19.1 本集目標

認識 ! 型別——代表永遠不會產生值。

10.19.2 概念說明

10.19.2.1 永不回傳的函數

大部分函數執行完會回傳一個值。但有些函數**永遠不會回傳**：

```
fn forever() -> ! {
    loop {
        // 永遠跑下去
    }
}
```

`-> !` 表示這個函數不可能回傳。

10.19.2.2 哪些東西的型別是!

- `panic!("...")` — 讓目前的執行緒發生 panic，而不是正常回傳。
- `std::process::exit(0)` — 程式結束。
- `loop {}`（沒有 break）— 永遠跑下去。
- `return` 表達式本身
- `break` 表達式本身
- `continue` 表達式本身

10.19.2.3 ! 可以被強制轉換成任何型別

這是 ! 最實用的特性。因為一個永遠不會產生值的表達式，放在任何需要值的地方都不會矛盾——反正它不會真的產生值。

這就是為什麼下面的程式碼能通過編譯：

```
    let x: i32 = match option {
        Some(v) => v,
        None => panic!("不該是 None"),
    };
```

`match` 的每個分支必須回傳同一個型別。`Some(v) => v` 回傳 `i32`，`None => panic!(...)` 回傳 !。因為 ! 可以轉成任何型別，所以被當成 `i32`，`match` 的型別一致。

`return`、`break` 和 `continue` 也一樣：

```
    let x: i32 = match option {
        Some(v) => v,
        None => return, // return 的型別是 !
    };
```

```
    for item in list {
        let value: i32 = match item.parse::<i32>() {
            Ok(n) => n,
            Err(_) => continue, // continue 的型別是 !
        };
        println!("{}", value);
```

```
    }
```

10.19.3 範例程式碼

```
fn exit_with_error(msg: &str) -> ! {
    println!("錯誤：{}", msg);
    std::process::exit(1);
}

fn parse_or_exit(input: &str) -> i32 {
    match input.parse::<i32>() {
        Ok(n) => n,
        Err(_) => exit_with_error("請輸入有效的數字"), // ! 被當成 i32
    }
}

fn main() {
    let value = parse_or_exit("42");
    println!("解析成功：{}", value);

    // let bad = parse_or_exit("abc"); // 這會呼叫 exit_with_error，程式直接結束
}
```

10.19.4 重點整理

- ！是 never type，代表永遠不會產生值。
- -> ！的函數永遠不會回傳。
- panic!、process::exit、return、break、continue 的型別都是 ！。
- ！可以被強制轉換成任何型別——match 裡一條路線回傳值一條路線 panic 就是靠這個。

恭喜你完成了進階語言功能這一章！🎉 這一章涵蓋了 Rust 的進階語言功能——從 dyn Trait、編譯時期運算、型別轉換、attribute、巨集系統，到 unsafe、static、FFI、union 和 never type。這些功能大部分在日常開發中不會天天用到，但知道它們的存在，需要的時候就能派上用場。下一章我們將看看標準庫裡的更多實用工具。

第 11 章

進階標準庫

和前面的章節有點不同：本章把重點從語言功能上拉開，而稍微介紹一些標準庫裡的內容。這樣的教學不可能是全面的，但如果知道一些這方面的知識仍然會方便許多。

11.1 AsRef<T> / AsMut<T>

11.1.1 本集目標

學會用 AsRef 和 AsMut 讓函數接受多種型別。

11.1.2 概念說明

11.1.2.1 動機

假設你寫了一個函數接受 &str：

```
fn print_length(s: &str) {
    println!("長度：{}", s.len());
}
```

呼叫者手上有 String，因為 Deref 的關係，&String 會自動轉成 &str，所以沒問題。但如果你想寫一個函數，讓它同時接受 String、&str、甚至其他型別呢？

11.1.2.2 AsRef

AsRef<T> trait 表示「我能便宜地借用成 &T」：

```
fn print_length(s: impl AsRef<str>) {
    println!("長度：{}", s.as_ref().len());
}

fn main() {
    let text = String::from("hello");

    print_length("嗨");                 // &str
    print_length(&text);
    print_length(String::from("你好"));  // String

    println!("{text}"); // 仍然可以使用
}
```

標準庫已經幫很多型別實作了 AsRef：

- String: AsRef<str>
- String: AsRef<[u8]>
- Vec<T>: AsRef<[T]>

11.1.2.3 AsMut

AsMut<T> 是可變版本，借用成 &mut T：

一般來說，我們使用 AsMut 是為了修改呼叫者原本擁有所有權的值，而不是取得該值的所有權。因此，參數通常會寫成 &mut impl AsMut<T>，而不是 impl AsMut<T>。

```
fn fill_zeros(buf: &mut impl AsMut<[u8]>) {
    for byte in buf.as_mut() {
        *byte = 0;
    }
}

fn main() {
    let mut v = vec![1, 2, 3];
    fill_zeros(&mut v);
    println!("{:?}", v); // [0, 0, 0]
}
```

11.1.2.4 所有權與使用時機

AsRef 和 AsMut 只描述一個型別能提供哪種參考，不會決定函數是否取得參數的所有權。所有權關係取決於參數型別，以及呼叫者實際傳入什麼。

雖然 s: impl AsRef<str> 是傳值參數，但實際傳入的型別也可以是參考。print_length(&text) 只借用 text，所以之後仍能使用；如果直接傳入 text，則會把它 move 走。這種寫法讓呼叫者自行選擇傳入擁有所有權的值或參考。

在 fill_zeros 中，外層的 &mut 表示函數只借用 buffer，AsMut<[u8]> 則表示該 buffer 能提供一個 &mut [u8]。

11.1.2.5 跟 Deref 的差別

Deref 是在 deref coercion 和 method call 這些地方被自動使用的：Rust 幫你穿過值去借用。AsRef 則是手動呼叫 .as_ref()。

更重要的差別：每個型別只能有一個 Deref 目標（String 的 target 是 str），但可以實作多個 AsRef（String 同時是 AsRef<str> 和 AsRef<[u8]>）。AsMut 同理。

當函數需要以同一種方式借用多種輸入型別時，就適合使用 AsRef<T> 或 AsMut<T>。

11.1.3 重點整理

- AsRef<T> 和 AsMut<T> 讓函數將多種輸入型別分別借用成 &T 和 &mut T。
- AsRef / AsMut 不決定所有權；impl AsRef<T> 可以接收擁有所有權的值或參考，而 &mut impl AsMut<T> 會借用並修改原值。
- 轉換時需手動呼叫 .as_ref() 或 .as_mut()；Deref 則可由 Rust 自動使用。
- 一個型別可以實作多個 AsRef / AsMut，但只能有一個 Deref 目標。

11.2 Ordering 與排序

11.2.1 本集目標

認識 Ordering、min/max 系列函數、排序方法，以及 Reverse 的原理。

11.2.2 概念說明

11.2.2.1 Ordering

第 5 章學了 Ord trait，知道實作了 Ord 的型別可以比大小。Ord 的核心方法是 cmp，它比較兩個值，回傳 std::cmp::Ordering——一個只有三個值的 enum：

```
use std::cmp::Ordering;

fn main() {
    match 5.cmp(&3) {
        Ordering::Less => println!("比較小"),
        Ordering::Equal => println!("一樣"),
        Ordering::Greater => println!("比較大"),
    }
}
```

11.2.2.2 min / max

std::cmp::min(a, b) 和 std::cmp::max(a, b) 回傳兩個值中比較小或大的那個，要求型別實作 Ord：

```
use std::cmp;

fn main() {
    println!("{}", cmp::min(3, 7)); // 3
    println!("{}", cmp::max(3, 7)); // 7
}
```

11.2.2.3 浮點數的問題

f64 沒有實作 Ord（第 5 章提過，因為 NAN 和任何值都比不出大小），所以不能直接用 cmp::min。

f64 只有 PartialOrd，它的方法是 partial_cmp，回傳 Option<Ordering> 而不是 Ordering——因為碰到 NAN 的時候沒辦法比大小，只能回傳 None。

這時候可以用 min_by / max_by，自訂比較邏輯：

```
use std::cmp;

fn main() {
    let smaller = cmp::min_by(3.0_f64, 2.5, |a, b| {
        a.partial_cmp(b).unwrap() // 如果確定不會碰到 NAN，用 unwrap 取出 Ordering
    });
    println!("{}", smaller); // 2.5

    let bigger = cmp::max_by(3.0_f64, 2.5, |a, b| {
        a.partial_cmp(b).unwrap()
    });
    println!("{}", bigger); // 3
}
```

min_by / max_by 的閉包回傳 Ordering，你自己決定怎麼比。

11.2.2.4 min_by_key / max_by_key

根據某個 key 來比較：

```rust
use std::cmp;

fn main() {
    let short = cmp::min_by_key("hello", "hi", |s| s.len());
    println!("{}", short); // "hi"
}
```

11.2.2.5 排序

Vec 和切片提供了幾種排序方法：

```rust
fn main() {
    let mut nums = vec![3, 1, 4, 1, 5];

    // sort：由小到大，要求 Ord
    nums.sort();
    println!("{:?}", nums); // [1, 1, 3, 4, 5]

    // sort_by：自訂比較，傳入閉包回傳 Ordering
    nums.sort_by(|a, b| b.cmp(a));
    println!("{:?}", nums); // [5, 4, 3, 1, 1]

    // sort_by_key：根據 key 排序
    let mut words = vec!["banana", "apple", "fig"];
    words.sort_by_key(|w| w.len());
    println!("{:?}", words); // ["fig", "apple", "banana"]
}
```

11.2.2.6 Reverse

`std::cmp::Reverse` 可以把排序順序反過來：

```rust
use std::cmp::Reverse;

fn main() {
    let mut nums = vec![3, 1, 4, 1, 5];
    nums.sort_by_key(|&x| Reverse(x));
    println!("{:?}", nums); // [5, 4, 3, 1, 1]
}
```

這是怎麼做到的？`Reverse` 其實就是一個 newtype：

```rust
pub struct Reverse<T>(pub T);
```

它的 `Ord` 實作把比較順序反了過來：

```rust
impl<T: Ord> Ord for Reverse<T> {
    fn cmp(&self, other: &Reverse<T>) -> Ordering {
        other.0.cmp(&self.0) // 注意：是 other 跟 self 比，反過來了
    }
}
```

正常的 `5.cmp(&3)` 回傳 `Greater`，但 `Reverse(5).cmp(&Reverse(3))` 內部做的是 `3.cmp(&5)`，回傳 `Less`。`sort_by_key` 用 key 的 `cmp` 來決定順序，key 被 `Reverse` 包住之後比較邏輯就自動反過來了。

比起 `sort_by(|a, b| b.cmp(a))`，`Reverse` 的寫法意圖更清楚。

11.2.3 範例程式碼

```
use std::cmp::{self, Reverse};

fn main() {
    // min / max
    println!("min(10, 20) = {}", cmp::min(10, 20));
    println!("max(10, 20) = {}", cmp::max(10, 20));

    // 浮點數用 min_by / max_by
    let smaller = cmp::min_by(1.5_f64, 2.3, |a, b| {
        a.partial_cmp(b).unwrap()
    });
    println!("min_by(1.5, 2.3) = {}", smaller);

    // 排序
    let mut scores = vec![85, 92, 78, 95, 88];
    scores.sort();
    println!("由小到大：{:?}", scores);

    scores.sort_by_key(|&s| Reverse(s));
    println!("由大到小：{:?}", scores);

    // 根據字串長度排序
    let mut names = vec!["Alice", "Bob", "Charlie", "Dave"];
    names.sort_by_key(|n| n.len());
    println!("依長度排：{:?}", names);
}
```

11.2.4 重點整理

- `Ordering` 有三個值：`Less`、`Equal`、`Greater`。
- `cmp::min` / `cmp::max` 取兩者的較小/大值，要求 `Ord`。
- `f64` 沒有 `Ord`，用 `min_by` / `max_by` 自訂比較。
- `min_by_key` / `max_by_key` 根據 key 比較。
- `sort()` 由小到大、`sort_by()` 自訂比較、`sort_by_key()` 根據 key 排序。
- `Reverse` 是一個 tuple `struct`，`Ord` 實作把比較反過來，所以排序結果跟著反轉。

11.3 HashMap<K, V>

11.3.1 本集目標

學會用 HashMap 儲存和查詢 key-value 資料。

11.3.2 概念說明

11.3.2.1 動機

如果你想用名字查分數、用 ID 查使用者，用 `Vec` 當然也做得到——存一堆 (名字, 分數) 的 tuple，要查的時候從頭走訪找到名字相符的那個。但這樣資料越多就越慢。

`HashMap<K, V>` 解決了這個問題。它用 hash 函數縮小 key 可能存放的範圍，通常不用走訪每一筆資料就能找到對應的值。

11.3.2.2 建立與基本操作

```rust
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();
    scores.insert("Alice", 95);
    scores.insert("Bob", 80);

    println!("{:?}", scores.get("Alice")); // Some(&95)
    println!("{:?}", scores.get("Eve"));   // None
}
```

`insert` 放入、`get` 查詢回傳 `Option<&V>`（key 不存在就是 `None`）、`remove` 刪除並回傳 `Option<V>`（key 存在就回傳 `Some(被刪掉的值)`，不存在就回傳 `None`）。

對同一個 key 再 `insert` 會覆蓋舊值。

11.3.2.3 用 `collect` 從迭代器建立

```rust
use std::collections::HashMap;

fn main() {
    let scores: HashMap<&str, i32> = vec![("Alice", 95), ("Bob", 80)]
        .into_iter()
        .collect();
}
```

11.3.2.4 走訪

```rust
use std::collections::HashMap;

fn main() {
    let scores: HashMap<&str, i32> = vec![("Alice", 95), ("Bob", 80)]
        .into_iter()
        .collect();
    for (name, score) in &scores {
        println!("{}: {}", name, score);
    }
}
```

注意走訪順序是**不固定**的——每次跑可能不一樣。如果你需要固定順序，用 `BTreeMap`（之後會介紹）。

11.3.2.5 Hash 是什麼

HashMap 要根據 key 快速找到對應的值。它會把 key 丟進一個 **hash 函數**，算出一個稱為 hash value 的數字，再用這個數字決定要從內部表格的哪裡開始找。如果多個 key 指向同一區域，它可能會檢查數個候選位置，直到找到 key 或確認它不在裡面。這樣就不用走訪每一筆已儲存的資料。

所以 key 的型別必須實作 `Hash trait`——告訴 Rust 如何把這個型別的值交給 hasher 處理。

11.3.2.6 Key 的要求：`Eq + Hash`

Key 除了要 `Hash`，還要 `Eq`。Hash value 只能縮小搜尋範圍，不能唯一識別一個 key。不同的 key 可能指向同一區域，所以 HashMap 會用 `==` 確認候選者是不是你要找的 key。

大部分基本型別（整數、bool、char、&str、String）都已經實作了 Eq + Hash。f64 沒有實作 Eq 或 Hash，所以不能直接當 key（NaN 是其中一個原因）。

11.3.2.7 幫自己的型別實作 Hash

Hash 可以 derive：

```
use std::collections::HashMap;

#[derive(Debug, PartialEq, Eq, Hash)]
struct Student {
    name: String,
    grade: i32,
}

fn main() {
    let mut map = HashMap::new();
    map.insert(Student { name: String::from("Alice"), grade: 90 }, "優等");
}
```

注意你同時需要 PartialEq、Eq 和 Hash——因為 Eq: PartialEq，三個都要。

一般來說，當你 derive PartialEq 和 Eq 的時候，建議也一起 derive Hash。這不會有額外的代價，但讓你的型別以後需要當 HashMap 的 key 的時候不用再回來改。

11.3.2.8 entry API

「有就不動，沒有才插入」是很常見的需求：

```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();
    scores.insert("Alice", 95);

    scores.entry("Alice").or_insert(0); // Alice 已存在，不動
    scores.entry("Eve").or_insert(0);   // Eve 不存在，插入 0
}
```

or_insert 回傳 &mut V，可以直接修改。這在計數的時候特別好用：

```
    let words = vec!["hello", "world", "hello", "rust"];
    let mut counts = HashMap::new();

    for word in words {
        let count = counts.entry(word).or_insert(0);
        *count += 1;
    }
    // {"hello": 2, "world": 1, "rust": 1}
```

11.3.2.9 其他常用方法

HashMap 還有一些常用的方法：

- .contains_key(&key)：檢查 key 是否存在，回傳 bool。
- .len()：回傳有幾組 key-value。
- .is_empty()：是不是空的。

- `.keys()`：所有 key 的迭代器
- `.values()`：所有 value 的迭代器

11.3.3 範例程式碼

```rust
use std::collections::HashMap;

fn main() {
    // 統計每個字元出現幾次
    let text = "hello world";
    let mut char_counts = HashMap::new();

    for c in text.chars() {
        if c == ' ' { continue; }
        let count = char_counts.entry(c).or_insert(0);
        *count += 1;
    }

    // 印出結果（順序不固定）
    for (ch, count) in &char_counts {
        println!("'{}': {} 次", ch, count);
    }

    // 找出出現最多次的字元
    if let Some((ch, count)) = char_counts.iter().max_by_key(|(_, count)| *count) {
        println!("出現最多的是 '{}'，共 {} 次", ch, count);
    }
}
```

11.3.4 重點整理

- `HashMap<K, V>` 利用 hash 按 key 尋找 value，不用走訪每一筆資料。
- `insert` 放入、`get` 查詢（回傳 `Option<&V>`）、`remove` 刪除。
- Key 必須實作 `Eq + Hash`，`Hash` 也可以 `derive`。
- `f64` 不能當 key（沒有 `Eq`）。
- `.entry(k).or_insert(v)` 是「沒有才插入」的慣用寫法，回傳 `&mut V`。
- 走訪順序不固定。

11.4 HashSet<T>

11.4.1 本集目標

學會用 `HashSet` 處理集合運算。

11.4.2 概念說明

11.4.2.1 動機

`HashMap` 存的是 key-value 對，但有時候你只關心「有沒有」而不關心對應的值——例如追蹤哪些使用者已經上線、哪些單字出現過。這時候用 `HashSet`。

11.4.2.2 本質

HashSet 其實就是只有 key 沒有 value 的 HashMap。所以元素一樣要求 Eq + Hash。

11.4.2.3 基本操作

```
use std::collections::HashSet;

fn main() {
    let mut fruits = HashSet::new();
    fruits.insert("apple");
    fruits.insert("banana");
    fruits.insert("apple"); // 重複，不會加進去

    println!("{}", fruits.contains("apple")); // true
    println!("{}", fruits.len());             // 2

    fruits.remove("banana");
}
```

11.4.2.4 從迭代器建立

```
use std::collections::HashSet;

fn main() {
    let nums: HashSet<i32> = vec![1, 2, 3, 2, 1].into_iter().collect();
    println!("{:?}", nums); // {1, 2, 3}，自動去掉重複
}
```

11.4.2.5 集合運算

這是 HashSet 最有用的地方：

```
use std::collections::HashSet;

fn main() {
    let a: HashSet<i32> = [1, 2, 3].into_iter().collect();
    let b: HashSet<i32> = [2, 3, 4].into_iter().collect();

    // 交集：兩邊都有的
    let intersection: HashSet<_> = a.intersection(&b).copied().collect();
    // {2, 3}

    // 聯集：合在一起
    let union_set: HashSet<_> = a.union(&b).copied().collect();
    // {1, 2, 3, 4}

    // 差集：a 有但 b 沒有的
    let diff: HashSet<_> = a.difference(&b).copied().collect();
    // {1}

    // 對稱差集：只在其中一邊的
    let sym_diff: HashSet<_> = a.symmetric_difference(&b).copied().collect();
    // {1, 4}
}
```

11.4.2.6 運算子

進階語言功能那章學了運算子重載——HashSet 就用了這個功能。你可以用 & | - ^ 對兩個 HashSet 的參考做集合運算：

```
    let intersection = &a & &b; // 交集
    let union_set    = &a | &b; // 聯集
    let diff         = &a - &b; // 差集
    let sym_diff     = &a ^ &b; // 對稱差集
```

11.4.2.7 其他關係

```
use std::collections::HashSet;

fn main() {
    let small: HashSet<i32> = [1, 2].into_iter().collect();
    let big: HashSet<i32> = [1, 2, 3, 4].into_iter().collect();

    println!("{}", small.is_subset(&big));   // true
    println!("{}", big.is_superset(&small)); // true
    println!("{}", small.is_disjoint(&big)); // false（有交集）
}
```

11.4.2.8 走訪

跟 HashMap 一樣，走訪順序不固定：

```
    for fruit in &fruits {
        println!("{}", fruit);
    }
```

11.4.3 範例程式碼

```
use std::collections::HashSet;

fn main() {
    let class_a: HashSet<&str> = ["Alice", "Bob", "Charlie", "Dave"].into_iter().collect();
    let class_b: HashSet<&str> = ["Charlie", "Dave", "Eve", "Frank"].into_iter().collect();

    println!("A 班：{:?}", class_a);
    println!("B 班：{:?}", class_b);

    // 兩班都有的人
    let both = &class_a & &class_b;
    println!("都有：{:?}", both);

    // 全部的人
    let all = &class_a | &class_b;
    println!("全部：{:?}", all);

    // 只在 A 班的人
    let only_a = &class_a - &class_b;
    println!("只在 A：{:?}", only_a);

    // 去掉重複
    let words = vec!["hello", "world", "hello", "rust", "world"];
```

```
    let unique: HashSet<_> = words.into_iter().collect();
    println!("不重複的字：{:?}", unique);
}
```

11.4.4 重點整理

- HashSet<T> 是只有 key 的 HashMap，元素不重複。
- 元素必須實作 Eq + Hash。
- insert 加入、contains 檢查、remove 移除。
- 集合運算：intersection(交集)、union(聯集)、difference(差集)、symmetric_difference(對稱差集)。
- 也可以用運算子：&（交集）、|（聯集）、-（差集）、^（對稱差集）。
- is_subset、is_superset、is_disjoint 判斷其他關係。

11.5 其他集合簡介

11.5.1 本集目標

認識 BTreeMap、BTreeSet 和 VecDeque。

11.5.2 概念說明

HashMap 和 HashSet 是最常用的集合，但標準庫還有其他選擇。

11.5.2.1 BTreeMap

跟 HashMap 的差別：key 是**有序**的。走訪的時候會按照 key 的排序順序，不是隨機順序：

```
use std::collections::BTreeMap;

fn main() {
    let mut scores = BTreeMap::new();
    scores.insert("Charlie", 70);
    scores.insert("Alice", 90);
    scores.insert("Bob", 85);

    for (name, score) in &scores {
        println!("{}: {}", name, score);
    }
    // 一定按字母順序：Alice, Bob, Charlie
}
```

代價是 key 必須實作 Ord（而不是 Hash + Eq）。查詢速度方面，HashMap 不管資料量多大幾乎是固定的，BTreeMap 資料越多會稍微慢一點，但還是很快。

11.5.2.2 BTreeSet

BTreeSet 就是只有 key 的 BTreeMap，跟 HashSet 對 HashMap 的關係一樣。元素有序，走訪時按順序輸出：

```
use std::collections::BTreeSet;

fn main() {
```

```
    let mut set = BTreeSet::new();
    set.insert(3);
    set.insert(1);
    set.insert(2);

    for x in &set {
        print!("{} ", x);
    }
    // 1 2 3
}
```

HashSet 的集合運算（交集、聯集等）BTreeSet 也都有。

11.5.2.3 什麼時候用哪個

- 不在乎順序 → HashMap / HashSet（比較快）。
- 需要按某種順序走訪、或需要找最小/最大的 key → BTreeMap / BTreeSet。

11.5.2.4 VecDeque

Vec 只能在尾巴高效地 push / pop。如果在頭 insert 或 remove，要把後面所有元素往後搬一格，資料越多越慢。

VecDeque（雙端佇列）在頭和尾都能高效操作，不管資料量多大，速度都幾乎不變：

```
use std::collections::VecDeque;

fn main() {
    let mut deque = VecDeque::new();
    deque.push_back(1);
    deque.push_back(2);
    deque.push_front(0);

    println!("{:?}", deque); // [0, 1, 2]

    deque.pop_front(); // 拿掉 0
    deque.pop_back();  // 拿掉 2
    println!("{:?}", deque); // [1]
}
```

11.5.2.5 什麼時候用 VecDeque

需要先進先出（FIFO）的佇列，或需要頻繁在頭尾操作的時候。如果只在尾巴操作，Vec 就夠了。

11.5.3 範例程式碼

```
use std::collections::{BTreeMap, VecDeque};

fn main() {
    // BTreeMap：有序的 key-value
    let mut scores = BTreeMap::new();
    scores.insert("Charlie", 70);
    scores.insert("Alice", 90);
    scores.insert("Bob", 85);
    scores.insert("Dave", 60);
```

```
    // 一定按字母順序印出
    for (name, score) in &scores {
        println!("{}: {}", name, score);
    }

    // VecDeque：雙端佇列
    let mut queue = VecDeque::new();
    queue.push_back("第一個");
    queue.push_back("第二個");
    queue.push_back("第三個");

    // 從前面拿，先進先出
    while let Some(item) = queue.pop_front() {
        println!("處理：{}", item);
    }
}
```

11.5.4 重點整理

- BTreeMap：走訪時，資料會按照 key 的排序順序出現；key 必須實作 Ord。
- BTreeSet：走訪時，元素會按照排序順序出現；元素必須實作 Ord。
- 需要排序走訪用 BTree 系列，不需要就用 Hash 系列（比較快）。
- VecDeque：雙端佇列，頭尾操作都很快。
- Vec 只在尾巴操作快，頭部操作慢（要搬移所有元素）。

11.6 std::env / std::process

11.6.1 本集目標

學會讀取命令列參數、環境變數，以及控制程式結束。

11.6.2 概念說明

11.6.2.1 命令列參數

程式執行的時候可以帶參數，例如 cargo run -- hello world。用 std::env::args() 拿到：

```
use std::env;

fn main() {
    for arg in env::args() {
        println!("{}", arg);
    }
}
```

第一個是程式本身的路徑，後面才是你傳的參數。通常會 collect 成 Vec 來用：

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() < 2 {
        println!("請提供參數");
        return;
```

```
    }
    println!("你輸入的是：{}", args[1]);
}
```

11.6.2.2 環境變數

環境變數是作業系統提供的一組 key-value 設定，程式可以讀取它們來取得系統資訊。如果你不熟悉環境變數，請自行搜尋相關資料。

```
use std::env;

fn main() {
    match env::var("HOME") {
        Ok(val) => println!("HOME = {}", val),
        Err(_) => println!("HOME 沒有設定"),
    }
}
```

`env::var` 回傳 `Result<String, VarError>`。環境變數不存在就會回傳 `Err`。

11.6.2.3 process::exit

```
use std::process;

fn main() {
    process::exit(1); // 立刻結束程式，回傳錯誤碼 1
}
```

回傳 0 通常代表成功，非 0 代表失敗。進階語言功能那章學過 `process::exit` 的回傳型別是 `!`（never type）。

11.6.2.4 eprintln!

```
fn main() {
    eprintln!("這是錯誤訊息");
    println!("這是正常輸出");
}
```

`println!` 輸出到 **stdout**（標準輸出），`eprintln!` 輸出到 **stderr**（標準錯誤）。兩者在終端機上看起來一樣，但可以分開導向不同的地方。錯誤訊息應該用 `eprintln!`。

11.6.3 範例程式碼

```
use std::env;
use std::process;

fn main() {
    let args: Vec<String> = env::args().collect();

    if args.len() < 2 {
        eprintln!("用法：{} <名字>", args[0]);
        process::exit(1);
    }

    let name = &args[1];
```

```
    println!("你好，{}！", name);

    // 印出一些環境變數
    if let Ok(home) = env::var("HOME") {
        println!("你的 HOME 目錄：{}", home);
    }

    if let Ok(path) = env::var("PATH") {
        let preview: String = path.chars().take(50).collect();
        println!("PATH 的前 50 個字元：{}", preview);
    }
}
```

11.6.4 重點整理

- `env::args()` 回傳命令列參數的迭代器，第一個是程式路徑。
- `env::var("NAME")` 回傳 `Result`。
- `process::exit(code)` 立刻結束程式，回傳型別是 `!`。
- `eprintln!` 輸出到 `stderr`，錯誤訊息應該用它。

11.7 `std::path`

11.7.1 本集目標

學會用 `Path` 和 `PathBuf` 處理跨平台路徑。

11.7.2 概念說明

11.7.2.1 動機

程式常常需要讀寫檔案，例如讀設定檔或寫 log。我們很快就會學怎麼做。不過在那之前，得先知道怎麼表示「檔案在哪裡」。

不同作業系統的路徑格式不一樣——Windows 用 `\`，Linux / macOS 用 `/`。如果你用字串硬拼路徑，跨平台就有可能出問題。`std::path` 幫你處理這些差異。

11.7.2.2 `Path` 和 `PathBuf`

跟 `str` 和 `String` 的關係一樣：

- `Path` 對應 `str`——是 DST，不能直接持有，通常用 `&Path`。
- `PathBuf` 對應 `String`——是具所有權的版本，可以修改。

```
use std::path::{Path, PathBuf};

fn main() {
    let p = Path::new("/home/user/file.txt");

    let mut buf = PathBuf::from("/home/user");
    buf.push("documents");
    buf.push("file.txt");
    println!("{}", buf.display()); // /home/user/documents/file.txt
}
```

push 會自動加上正確的路徑分隔符號。

11.7.2.3 常用方法

```rust
use std::path::Path;

fn main() {
    let p = Path::new("/home/user/notes.txt");

    println!("{:?}", p.parent());    // Some("/home/user")
    println!("{:?}", p.file_name()); // Some("notes.txt")
    println!("{:?}", p.extension()); // Some("txt")
    println!("{:?}", p.file_stem()); // Some("notes")
    println!("{}", p.exists());      // 檢查路徑是否存在
    println!("{}", p.is_file());     // 是不是檔案
    println!("{}", p.is_dir());      // 是不是目錄
}
```

file_name、extension、file_stem 回傳的是 Option<&OsStr>，不是 Option<&str>——因為檔案名稱在某些作業系統上不一定是合法的 UTF-8。大部分時候可以用 .to_str().unwrap() 轉成 &str。

11.7.2.4 join

join 跟 push 類似，但不改變原本的 Path 或 PathBuf，而是回傳新的 PathBuf：

```rust
use std::path::Path;

fn main() {
    let dir = Path::new("/home/user");
    let file = dir.join("documents").join("file.txt");
    println!("{}", file.display()); // /home/user/documents/file.txt
}
```

11.7.2.5 和字串的轉換

```rust
use std::path::{Path, PathBuf};

fn main() {
    // &str → &Path
    let p = Path::new("hello.txt");

    // &str → PathBuf
    let buf = PathBuf::from("/some/path");

    // PathBuf → String（可能有損，非 UTF-8 字元會被替換）
    let s: String = buf.to_string_lossy().into_owned();
}
```

11.7.3 範例程式碼

```rust
use std::path::{Path, PathBuf};

fn show_info(path: &Path) {
    println!("路徑：{}", path.display());
```

```
    if let Some(parent) = path.parent() {
        println!("  上層：{}", parent.display());
    }
    if let Some(name) = path.file_name() {
        println!("  檔名：{:?}", name);
    }
    if let Some(ext) = path.extension() {
        println!("  副檔名：{:?}", ext);
    }
    println!("  存在：{}", path.exists());
}

fn main() {
    show_info(Path::new("/home/user/notes.txt"));

    // 用 PathBuf 組合路徑
    let mut config_path = PathBuf::from("/home/user");
    config_path.push(".config");
    config_path.push("app");
    config_path.push("settings.toml");
    show_info(&config_path);

    // join 不改變原本的 Path
    let base = Path::new("/var/log");
    let log_file = base.join("app.log");
    show_info(&log_file);
}
```

11.7.4 重點整理

- Path 是 DST（對應 str），PathBuf 是擁有所有權的版本（對應 String）。
- push / join 自動加上正確的路徑分隔符號。
- parent、file_name、extension、file_stem 拆解路徑。
- exists、is_file、is_dir 檢查路徑狀態。

11.8 字串方法

11.8.1 本集目標

認識 &str 和 String 上最常用的方法，以及 Rust 字串與 UTF-8 的關係。

11.8.2 概念說明

讀寫檔案的時候，拿到的內容通常是字串，你會需要各種方法來處理它——搜尋、分割、修剪、替換等等。之前我們用過 .trim()、.parse() 和 .chars()，但 &str 和 String 上其實有非常多實用的方法。這集介紹最常用的幾個。

11.8.2.1 搜尋

```
    let s = "hello, world!";

    s.contains("world");    // true
```

```
    s.starts_with("hello"); // true
    s.ends_with("!");       // true
    s.find("world");        // Some(7)，回傳第一次出現的位置（byte index）
```

11.8.2.2　修剪與替換

```
    "  hello  ".trim();       // "hello"
    "  hello  ".trim_start(); // "hello  "
    "  hello  ".trim_end();   // "  hello"

    "hello world".replace("world", "Rust"); // "hello Rust"
```

11.8.2.3　分割

```
    let parts: Vec<&str> = "a,b,c".split(',').collect();
    // ["a", "b", "c"]

    let words: Vec<&str> = "hello  world".split_whitespace().collect();
    // ["hello", "world"]
```

`split` 回傳迭代器，通常搭配 `collect` 使用。

11.8.2.4　逐字元走訪

```
fn main() {
    for c in "hello".chars() {
        println!("{}", c);
    }
}
```

`.chars()` 回傳 Unicode 字元的迭代器。也有 `.bytes()` 回傳原始 byte。

11.8.2.5　大小寫

```
    "Hello".to_uppercase(); // "HELLO"
    "Hello".to_lowercase(); // "hello"
```

11.8.2.6　`len` 是 byte 數

```
    "hello".len();          // 5
    "hello".is_empty();     // false
    "hello".repeat(3);      // "hellohellohello"

    // 注意：.len() 回傳的是 byte 數，不是字元數
    "你好".len();           // 6（UTF-8 byte 數）
    "你好".chars().count(); // 2（字元數）
```

11.8.3　範例程式碼

```
fn main() {
    let sentence = "  Hello, Rust World!  ";

    // 修剪空白
    let trimmed = sentence.trim();
```

```
    println!("修剪後：'{}'", trimmed);

    // 搜尋
    println!("包含 Rust：{}", trimmed.contains("Rust"));
    println!("Rust 的位置：{:?}", trimmed.find("Rust"));

    // 分割
    let words: Vec<&str> = trimmed.split_whitespace().collect();
    println!("字數：{}", words.len());
    for word in &words {
        println!("  {}", word);
    }

    // 替換
    let replaced = trimmed.replace("Rust", "世界");
    println!("替換後：{}", replaced);

    // UTF-8
    let chinese = "你好世界";
    println!("byte 數：{}", chinese.len());          // 12
    println!("字元數：{}", chinese.chars().count()); // 4

    for (i, c) in chinese.chars().enumerate() {
        println!("第 {} 個字元：{}", i + 1, c);
    }
}
```

11.8.4 重點整理

- `contains`、`starts_with`、`ends_with`、`find`：搜尋
- `trim`、`trim_start`、`trim_end`：修剪空白
- `replace`：替換
- `split`、`split_whitespace`：分割，回傳迭代器。
- `chars`：逐字元走訪；`bytes`：逐 byte 走訪。
- `len` 回傳 byte 數，字元數要用 `.chars().count()`。

11.9 I/O：stdin、檔案讀寫

11.9.1 本集目標

認識 Rust I/O（輸入輸出，input / output）方法，學會讀寫檔案。

11.9.2 概念說明

前面學了怎麼用 Path 表示檔案位置、怎麼處理字串內容，這集來學怎麼實際讀寫檔案。

11.9.2.1 回顧 stdin

第 1 章我們照抄了這三行來讀取使用者輸入：

```
    let mut input = String::new();
    std::io::stdin().read_line(&mut input).expect("讀取失敗");
    let name = input.trim();
```

你已經學過 String、&mut 和 .expect()，這段應該都看得懂了。還沒提過的是 std::io::stdin() 回傳一個 Stdin struct，而 read_line 回傳的是 io::Result<usize>——這是 Result<usize, io::Error> 的型別別名，標準庫裡很多 I/O 函數都用它。

11.9.2.2 最簡單的檔案讀寫

fs::read_to_string 的參數型別是 impl AsRef<Path>——第 1 集學的 AsRef 在這裡派上用場。你可以傳 &str、String、&Path、PathBuf 都行，不用手動轉換。

讀整個檔案成字串：

```
use std::fs;

fn main() {
    let content = fs::read_to_string("hello.txt").expect("讀取失敗");
    println!("{}", content);
}
```

寫入檔案（檔案不存在會建立，存在會覆蓋）：

```
use std::fs;

fn main() {
    fs::write("output.txt", "Hello, file!").expect("寫入失敗");
}
```

這兩個函數夠簡單，但 read_to_string 會把整個檔案一次讀進記憶體——如果檔案很大就不適合。

11.9.2.3 File + BufReader：逐行讀取

```
use std::fs::File;
use std::io::{BufRead, BufReader};

fn main() {
    let file = File::open("data.txt").expect("開啟失敗");
    let reader = BufReader::new(file);

    for line in reader.lines() {
        let line = line.expect("讀取行失敗");
        println!("{}", line);
    }
}
```

File::open 開啟檔案，BufReader 包在外面提供緩衝區，.lines() 逐行讀取，每行是一個 io::Result<String>。

11.9.2.4 寫入檔案

```
use std::fs::File;
use std::io::Write;

fn main() {
    let mut file = File::create("output.txt").expect("建立失敗");
    writeln!(file, "第一行").expect("寫入失敗");
    writeln!(file, "第二行").expect("寫入失敗");
}
```

File::create 建立（或覆蓋）檔案，writeln! 跟 println! 很像，只是輸出目標從螢幕換成檔案。要用 writeln! 需要引入 Write trait。

11.9.2.5 Read、Write、BufRead

標準庫用 trait 抽象化 I/O：

- Read：能讀取 bytes 的東西（File、Stdin、TcpStream 等）。
- Write：能寫入 bytes 的東西（File、Stdout、TcpStream 等）。
- BufRead：帶緩衝的讀取，提供 lines() 等方法。BufReader 可以把任何 Read 變成 BufRead。

這就是為什麼很多函數的參數寫成 impl Read 或 impl Write——不管你傳檔案、stdin 還是網路連線，只要實作了對應的 trait 就能用。

11.9.3 範例程式碼

```
use std::fs::{self, File};
use std::io::{self, BufRead, BufReader, Write};

fn main() -> io::Result<()> {
    // 寫入檔案
    let mut file = File::create("names.txt")?;
    writeln!(file, "Alice")?;
    writeln!(file, "Bob")?;
    writeln!(file, "Charlie")?;

    // 一次讀取整個檔案
    let all = fs::read_to_string("names.txt")?;
    println!("整個檔案：\n{}", all);

    // 逐行讀取
    let file = File::open("names.txt")?;
    let reader = BufReader::new(file);
    for (i, line) in reader.lines().enumerate() {
        println!("第 {} 行：{}", i + 1, line?);
    }

    Ok(())
}
```

11.9.4 重點整理

- io::Result<T> 是 Result<T, io::Error> 的型別別名。
- fs::read_to_string / fs::write：最簡單的一行讀寫
- File::open + BufReader：逐行讀取大檔案
- File::create + writeln!：逐行寫入。
- Read、Write、BufRead 是 I/O 的核心 trait，讓不同來源（檔案、stdin、網路）用同一套介面。

11.10 Error trait

11.10.1 本集目標

學會自訂錯誤型別，以及用 Box<dyn Error> 統一處理不同種類的錯誤。

11.10.2 概念說明

11.10.2.1 回顧：Result 和 ?

第 5 章學了 Result<T, E> 和 ? 運算子。但當時錯誤型別都很單純——一個函數只會產生一種錯誤。實際的程式常常會碰到多種錯誤：讀檔案可能失敗（io::Error），解析數字也可能失敗（ParseIntError）。如果函數裡兩種都會發生，回傳的 Result 的 E 該填什麼？

11.10.2.2 Error trait

標準庫定義了 std::error::Error trait，所有錯誤型別的共同介面：

```
pub trait Error: std::fmt::Display + std::fmt::Debug {
    fn source(&self) -> Option<&(dyn Error + 'static)> { None }
}
```

要實作 Error，你的型別必須先實作 Display 和 Debug。.source() 回傳造成這個錯誤的底層原因，預設是 None。

11.10.2.3 自訂錯誤型別

用一個 enum 把所有可能的錯誤包在一起：

```
use std::fmt;

#[derive(Debug)]
enum AppError {
    Io(std::io::Error),
    Parse(std::num::ParseIntError),
}

impl fmt::Display for AppError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            AppError::Io(_) => write!(f, "輸入輸出操作失敗"),
            AppError::Parse(_) => write!(f, "整數解析失敗"),
        }
    }
}

impl std::error::Error for AppError {
    fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
        match self {
            AppError::Io(e) => Some(e),
            AppError::Parse(e) => Some(e),
        }
    }
}
```

以 AppError::Parse 為例，Display 會顯示「整數解析失敗」，而 .source() 會回傳原本的

ParseIntError，讓呼叫者需要時再查看更詳細的解析錯誤。這樣外層訊息和底層原因各自保留，不會把同一段錯誤文字印兩次。

第 5 章學了 ?，當時說它遇到 Err 就提前回傳。其實 ? 還多做了一件事：它會呼叫 From::from(e) 把錯誤轉換成函數回傳型別裡的 E。所以只要你幫底層錯誤實作了 From，? 就能自動轉換：

```
impl From<std::io::Error> for AppError {
    fn from(e: std::io::Error) -> Self {
        AppError::Io(e)
    }
}

impl From<std::num::ParseIntError> for AppError {
    fn from(e: std::num::ParseIntError) -> Self {
        AppError::Parse(e)
    }
}
```

現在同一個函數裡可以用 ? 處理兩種錯誤：

```
fn read_number(path: &str) -> Result<i32, AppError> {
    let content = std::fs::read_to_string(path)?; // io::Error → AppError
    let num = content.trim().parse::<i32>()?;     // ParseIntError → AppError
    Ok(num)
}
```

11.10.2.4 問題：每次都要寫這麼多？

自訂錯誤型別 + impl Display + impl Error + 每種 From... 很囉嗦。有沒有更簡單的方式？

11.10.2.5 Box<dyn Error>

如果你不需要讓回傳型別列出一組可以窮舉 match 的錯誤種類，可以用 Box<dyn Error> 當通用錯誤型別：

```
use std::error::Error;

fn read_number(path: &str) -> Result<i32, Box<dyn Error>> {
    let content = std::fs::read_to_string(path)?;
    let num = content.trim().parse::<i32>()?;
    Ok(num)
}
```

實作了 Error + 'static 的具體錯誤型別可以自動轉成 Box<dyn Error>，所以這些錯誤可以直接搭配 ? 使用，不需要逐一手寫 From。

Box<dyn Error> 會抹除具體錯誤的靜態型別，因此呼叫者不能像處理自訂錯誤 enum 一樣進行窮舉 match。不過，錯誤資訊並未完全消失：仍可用 .source() 查看底層原因，也能在已知目標型別時使用 .downcast_ref::<T>() 檢查具體型別。

例如，呼叫者可以檢查 Box<dyn Error> 裡裝的是不是 std::io::Error：

```
fn main() {
    if let Err(error) = read_number("missing.txt") {
        if let Some(io_error) = error.downcast_ref::<std::io::Error>() {
            println!("這是 I/O 錯誤，種類是：{:?}", io_error.kind());
        } else {
```

```
            println!("這是其他錯誤：{}", error);
        }
    }
}
```

如果 Box 裡直接裝的是 std::io::Error，.downcast_ref::<std::io::Error>() 會回傳 Some(&std::io::Error)；型別不同就回傳 None。例如，Box 裡裝的是 AppError 時，即使 AppError::Io 內部包著 std::io::Error，error.downcast_ref::<std::io::Error>() 仍會回傳 None，因為直接裝在 Box 裡的型別是 AppError。這時要先呼叫 .source() 取得內部錯誤，再對它呼叫 .downcast_ref::<T>()。

Box<dyn Error> 本身不保證錯誤可以跨執行緒傳遞。如果錯誤需要送到其他執行緒，或 API 要求可跨執行緒的錯誤，常見的型別是 Box<dyn Error + Send + Sync>；放進去的錯誤也必須實作 Send + Sync。

11.10.2.6 什麼時候用哪個

- **快速原型、腳本、main 函數**：Box<dyn Error> 最省事。
- **函式庫、需要讓呼叫者精確處理錯誤**：自訂錯誤 enum + impl Error + impl From。

下一集會介紹社群 crate 怎麼大幅簡化自訂錯誤型別的寫法。

11.10.3 範例程式碼

```
use std::error::Error;
use std::fs;

fn first_line_number(path: &str) -> Result<i32, Box<dyn Error>> {
    let content = fs::read_to_string(path)?;
    let first_line = content.lines().next().ok_or("檔案是空的")?;
    let num = first_line.trim().parse::<i32>()?;
    Ok(num)
}

fn main() {
    match first_line_number("number.txt") {
        Ok(n) => println!("讀到的數字：{}", n),
        Err(e) => println!("錯誤：{}", e),
    }
}
```

11.10.4 重點整理

- Error trait 要求 Display + Debug，是所有錯誤型別的共同介面。
- 自訂錯誤：定義 enum → impl Display → impl Error（用 .source() 保留原因）→ 為每種底層錯誤 impl From。
- 有了 From，? 就能自動把底層錯誤轉成你的自訂錯誤。
- Box<dyn Error>：會抹除具體錯誤的靜態型別，但仍可透過 .source() 或 downcast 檢查錯誤。
- 需要跨執行緒傳遞錯誤時，常用 Box<dyn Error + Send + Sync>。
- Box<dyn Error> 適合快速開發；自訂錯誤 enum 適合函式庫。

11.11 thiserror / anyhow 簡介

11.11.1 本集目標

認識兩個社群最常用的錯誤處理 crate。

11.11.2 概念說明

這集介紹的不是標準庫的內容，而是兩個社群 crate。但它們在 Rust 生態裡幾乎是標配，非常實用，所以放在這裡一起介紹。

使用前要先安裝：

```
cargo add thiserror
cargo add anyhow
```

11.11.2.1 背景

上一集看到自訂錯誤要寫一堆重複的程式碼（enum + Display + Error + 每種 From）。thiserror 和 anyhow 幫你解決這個問題。

11.11.2.2 thiserror：給函式庫用

thiserror 用 derive 巨集自動生成 Display、Error、From：

```rust
use thiserror::Error;

#[derive(Debug, Error)]
enum AppError {
    #[error("輸入輸出操作失敗")]
    Io(#[from] std::io::Error),

    #[error("整數解析失敗")]
    Parse(#[from] std::num::ParseIntError),

    #[error("自訂錯誤：{0}")]
    Custom(String),
}
```

- `#[error("...")]` 自動生成 Display 的實作。
- `#[from]` 自動生成 From 的實作，並把同一個欄位當成 `.source()` 回傳的底層原因。
- 上一集手動寫了幾十行的東西，現在幾行就搞定。

使用方式跟上一集一樣——`?` 會自動轉換：

```rust
fn read_number(path: &str) -> Result<i32, AppError> {
    let content = std::fs::read_to_string(path)?;
    let num = content.trim().parse::<i32>()?;
    Ok(num)
}
```

呼叫端一樣可以 match 精確處理每種錯誤。

11.11.2.3 anyhow：給應用程式用

如果你不需要讓呼叫者區分錯誤種類（例如 main 函數、CLI 工具），anyhow 更簡單：

```rust
use anyhow::{Context, Result};

fn read_number(path: &str) -> Result<i32> {
    let content = std::fs::read_to_string(path)
        .context("讀取檔案失敗")?;
    let num = content.trim().parse::<i32>()
        .context("解析數字失敗")?;
    Ok(num)
}
```

- `anyhow::Result<T>` 就是 `Result<T, anyhow::Error>`。
- `anyhow::Error` 類似 `Box<dyn Error + Send + Sync>`，但使用起來更方便。
- `.context("...")` 幫錯誤加上額外說明，方便除錯。
- 不用定義新的錯誤型別；實作了 `Error + Send + Sync + 'static` 的錯誤可以自動轉換。

11.11.2.4 兩者的關係

- **thiserror**：幫你定義精確的錯誤型別，省去手寫重複的程式碼。適合函式庫——使用者能 match 你的錯誤。
- **anyhow**：不用另外定義錯誤型別，用同一個型別統一處理符合上述條件的錯誤。適合應用程式——只需要報告錯誤，不需要讓別人程式化處理。

兩者可以搭配使用：函式庫用 thiserror 定義錯誤，應用程式用 anyhow 統一接收。

11.11.3 範例程式碼

```rust
// 這個範例展示 anyhow 的用法
extern crate anyhow;

use anyhow::{Context, Result};
use std::fs;

fn read_config(path: &str) -> Result<(String, i32)> {
    let content = fs::read_to_string(path)
        .context("無法讀取設定檔")?;

    let mut lines = content.lines();

    let name = lines.next()
        .context("設定檔是空的")?
        .to_string();

    let value = lines.next()
        .context("缺少第二行")?
        .trim()
        .parse::<i32>()
        .context("第二行不是有效的數字")?;

    Ok((name, value))
}

fn main() -> Result<()> {
    let (name, value) = read_config("config.txt")?;
    println!("名稱：{}，值：{}", name, value);
```

```
    Ok(())
}
```

11.11.4　重點整理

- `thiserror`：用 `derive` 巨集自動生成 `Display`、`Error`、`From`，適合函式庫。
- `#[error("...")]` 生成 `Display`；`#[from]` 生成 `From`，並把欄位標記為底層 `source`。
- `anyhow`：統一處理實作了 `Error + Send + Sync + 'static` 的錯誤，不用定義錯誤 `enum`，適合應用程式。
- `.context("...")` 幫錯誤加上額外說明。
- 函式庫用 `thiserror`，應用程式用 `anyhow`，兩者可以搭配。

11.12　`catch_unwind`

11.12.1　本集目標

學會用 `catch_unwind` 攔截 panic，並理解什麼時候適合使用它。

11.12.2　概念說明

11.12.2.1　基本用法

一般來說，沒有被攔截的 panic 最終會終止目前的執行緒。`catch_unwind` 可以在閉包周圍建立一道邊界，攔住離開閉包的 panic：

```
use std::panic;

fn main() {
    let result = panic::catch_unwind(|| {
        println!("正常執行");
        42
    });
    println!("{:?}", result); // Ok(42)

    let result = panic::catch_unwind(|| {
        panic!("出事了阿北");
    });
    println!("{:?}", result); // Err(...)
}
```

閉包正常回傳時會得到 `Ok(值)`；閉包 panic 時則會得到 `Err`，程式之後可以繼續執行。

即使 panic 已經被攔住，終端機上仍可能顯示 panic 訊息。重點是程式會繼續執行，而且 `catch_unwind` 會回傳 `Err`。

11.12.2.2　為什麼要攔截 panic？

其中一個特殊用途是在提供給 C 呼叫的 Rust 函數裡攔截 panic。如果 `extern "C"` 函數裡的 panic 沒有被攔住，整個程式會在回到 C 之前自動中止。

為了避免中止，你可以在 Rust 函數裡使用 `catch_unwind`，再把 `Err` 轉成錯誤碼。本集最後的範例會示範這種寫法。如果你能接受程式中止，就不需要使用 `catch_unwind`。

11.12.2.3 UnwindSafe

`catch_unwind` 要求閉包是 `UnwindSafe` 的。理由很簡單：panic 可能在修改做到一半時將操作打斷，而程式攔截 panic 後，可能繼續使用只更新到一半的資料。

`&mut T` 無法通過這項檢查。如果閉包透過可變參考修改資料後 panic，閉包外的資料可能會停在只更新一半的狀態。

大部分共享參考都能通過檢查，例如 `&i32` 和 `&String`，但並不是所有 `&T` 都可以。例如 `&Cell<T>` 和 `&RefCell<T>` 就不行，因為 `Cell` 和 `RefCell` 能透過共享參考修改資料。

`UnwindSafe` 只是提醒你思考 panic 後留下的狀態，並不會證明資料在邏輯上一定正確。

11.12.2.4 AssertUnwindSafe

如果你已經考慮過可能留下的狀態，也知道該怎麼處理，`AssertUnwindSafe` 可以讓你明確要求 Rust 接受這個閉包：

```
use std::panic::{catch_unwind, AssertUnwindSafe};

fn main() {
    let mut data = vec![1, 2, 3];
    let original_len = data.len();

    let result = catch_unwind(AssertUnwindSafe(|| {
        data.push(4);
        panic!("修改到一半時停止了");
    }));

    if result.is_err() {
        data.truncate(original_len);
    }

    println!("{:?}", data); // [1, 2, 3]
}
```

`AssertUnwindSafe` 不會自動幫你修復資料。它只是告訴 Rust：你願意負責在 panic 後檢查或恢復資料的狀態。

11.12.2.5 panic = "abort"

`Cargo.toml` 可以設定：

```
[profile.release]
panic = "abort"
```

使用這個設定時，panic 會立刻終止整個程式，`catch_unwind` 無法將它攔下來。

11.12.2.6 不是一般的錯誤處理

`catch_unwind` 不是一般錯誤處理用的 `try／catch`。能預期的失敗應該使用 `Result`。只有在刻意需要限制 panic 影響範圍時才使用 `catch_unwind`，例如讓 FFI 函數回傳錯誤碼，而不是讓整個程式中止。

11.12.3 範例程式碼

```rust
use std::panic;

// 模擬由 FFI 函數呼叫，而且我們無法完全控制的程式碼。
fn library_task(mode: i32) -> i32 {
    if mode == 0 {
        panic!("library task panic 了");
    }
    100 / mode
}

extern "C" fn ffi_entry(mode: i32) -> i32 {
    match panic::catch_unwind(|| library_task(mode)) {
        Ok(value) => value,
        Err(_) => -1, // 把 panic 轉成錯誤碼
    }
}

fn main() {
    println!("成功：{}", ffi_entry(4)); // 25
    println!("失敗：{}", ffi_entry(0)); // -1，程式會繼續執行
}
```

這裡的 panic 會在 `ffi_entry` 裡被攔下來，不會離開 `extern "C"` 函數。這個函數最後會正常回傳 `-1`。

11.12.4 重點整理

- `catch_unwind` 執行一個閉包，回傳 `Ok(值)` 或 `Err`。
- 被攔住的 panic 仍可能印出訊息，但程式可以繼續執行。
- 如果 `extern "C"` Rust 函數裡的 panic 沒有被攔住，整個程式會在回到 C 之前中止。只有想改成回傳錯誤碼等其他結果時，才需要事先攔截。
- `&mut T` 無法通過 `UnwindSafe` 檢查。大部分共享參考可以，但 `&Cell<T>` 和 `&RefCell<T>` 是例外。
- `AssertUnwindSafe` 會要求 Rust 接受你的判斷，但你仍要負責處理只更新到一半的資料。
- `panic = "abort"` 設定下，`catch_unwind` 無法攔截 panic。
- 能預期的失敗應該使用 `Result`，不要使用 `catch_unwind`。

恭喜你完成了進階標準庫這一章！🎉 這一章介紹了標準庫和社群裡的各種實用工具——從 `AsRef`、排序、集合，到輸入輸出、字串方法、錯誤處理，再到 `catch_unwind`。下一章我們將進入非同步的世界！

第 12 章

非同步

有了前面各章節的基礎，現在我們可以進入這份教學最難的一章了：非同步。除了有各式各樣的新概念之外，你還必須在本章中閱讀大量程式碼；這章並不是我特別為了訓練讀者而設計的，但以結果論來說或許的確能如此。在本章的前半段，我們會設計一套類似 API 的完整工具。如果能完成這部分的話，相信讀者會更能了解在 Rust 語言之中如何建構抽象；後半段我們則介紹實務上撰寫非同步程式會用到的工具。

12.1 第一個 async 程式

12.1.1 本集目標

直接用 Tokio 寫一個會回應瀏覽器的小小伺服器，先對 async 程式長什麼樣子有個印象。

12.1.2 正文

歡迎來到非同步的世界！這一章我們會花很多力氣，慢慢把非同步（async）的原理一層一層挖開。不過第一集先不講原理，我們直接寫一個能跑的程式，讓你對 async 的程式碼長相先有個感覺。讀到這裡的你已經學過很多東西了，相信光看程式碼，大概就猜得出它在做什麼。

先從「非同步」這三個字的字面意思開始。**同步**是「大家照同一個步調走」：一件事沒做完，下一件事就先等著。**非同步**則是「不一定要照同一個步調等」：某件事正在等結果時，程式可以先去推進別的事。放到伺服器裡，就是等待某個瀏覽器連進來、或等待某個回應送出去時，不必讓其他連線都卡在原地。

12.1.2.1 Rust 的 async 需要一個 runtime

和很多其他程式語言不一樣，Rust 標準庫本身**沒有**內建非同步的執行引擎（我們之後會叫它 runtime）。標準庫只定義了非同步的「規格」，至於要怎麼真的把這些非同步的工作跑起來，是交給第三方 `crate` 決定的。這聽起來有點奇怪，但這個設計讓 Rust 的 async 可以用在從大型伺服器到小型嵌入式裝置等各種場合。

目前最多人用的 runtime 叫做 **Tokio**。這一章後半，我們會深入了解 Tokio 的功能。要使用它，先在 `Cargo.toml` 加上依賴：

```
[dependencies]
tokio = { version = "1", features = ["full"] }
```

或是用指令：

```
cargo add tokio --features full
```

等一下的程式會在 `main` 裡直接寫 `.await`。這裡先記一個語法規則：**`.await` 只能出現在 `async` 的環境裡**。普通的 `fn main()` 裡不能直接 `.await`，所以我們會把它寫成 `async fn main()`。

不過，`async fn main()` 不能像普通的 `fn main()` 那樣自己直接當程式入口。`#[tokio::main]` 這個 attribute 就是 Tokio 提供的幫手：它會替我們準備 runtime，讓這個 `async fn main()` 可以真的被執行。

12.1.2.2 一個會數數的伺服器

下面這個程式會在你的電腦上開一個小伺服器，每當有人連進來，就回一句「這是第 N 個 request」。所有連線共用同一個計數器，所以你重新整理瀏覽器時，數字會一直往上加：

```
use std::sync::Arc;
use std::sync::atomic::{AtomicU64, Ordering};
use tokio::io::AsyncWriteExt;
use tokio::net::TcpListener;

#[tokio::main]
async fn main() {
    // 所有連線共用的計數器
    let counter = Arc::new(AtomicU64::new(0));

    // 監聽本機的 8080 連接埠
    let listener = TcpListener::bind("127.0.0.1:8080").await.expect("綁定失敗");
    println!("伺服器啟動了，請用瀏覽器打開 http://127.0.0.1:8080");

    loop {
        // 等待下一個連線進來
        let (mut socket, _) = listener.accept().await.expect("accept 失敗");

        // 把計數器的所有權分一份給待會的背景工作
        let counter = Arc::clone(&counter);

        // 把這個連線丟到背景處理，主迴圈馬上回去等下一個連線
        tokio::spawn(async move {
            let n = counter.fetch_add(1, Ordering::SeqCst) + 1;
            let body = format!("這是第 {} 個 request\n", n);
            let response = format!(
                "HTTP/1.1 200 OK\r\nContent-Length: {}\r\n\r\n{}",
                body.len(),
                body,
            );
            socket.write_all(response.as_bytes()).await.expect("回應失敗");
        });
    }
}
```

在你的電腦上把它跑起來之後，打開瀏覽器連到 `http://127.0.0.1:8080`，重新整理幾次，你會看到數字一直增加。

> 用瀏覽器開啟這個網址時，瀏覽器除了請求網頁本身，還可能自動請求 `/favicon.ico`。這個簡化的範例會把每個連進來的 TCP 連線都計數，因此數字有時會一次增加 2。畫面上顯示的是網頁本身的回應；favicon 的回應不會顯示在頁面中。

12.1.2.3 `.await` 是什麼意思

程式裡出現了好幾個 `.await`，這是 async 程式最核心的東西。你可以先這樣理解它：

`.await` 的意思是「這件事可能要等一下才會好；如果要等的話，請在等的這段時間試著找別的事做。」

以 `listener.accept().await` 為例：接受新連線要等到真的有人連進來，這中間可能是幾毫秒，也可能是好幾秒。`.await` 標出這個「可能需要等」的位置；等的期間，這個 async 工作可以先被暫停，把執行機會讓出去。

12.1.2.4 它真的能同時處理很多連線

注意我們用了 `tokio::spawn`，把「處理單一連線」這件事丟到背景。主迴圈只負責接收新連線；每接到一個，就生出一個背景工作去寫回應，自己立刻回去等下一個連線。

所以就算某個連線正在 `socket.write_all(response.as_bytes()).await`，主迴圈也不必等它寫完。其他連線可以繼續被接受、繼續被處理。這種把很多件事拆開、交錯推進的能力，正是 async 的賣點。

這一集先讓你看到 async 程式的長相和效果。下一集我們再把動機講清楚：它適合什麼場景、為什麼不直接開很多 `Thread`，以及「並行」和「平行」到底差在哪裡。

12.1.3 重點整理

- Rust 標準庫只定義 async 的規格，真正執行要靠第三方的 **runtime**，最常用的是 **Tokio**。
- `.await` 只能寫在 async 的環境裡；`#[tokio::main]` 讓 `main` 可以寫成 `async fn`，並幫你把 runtime 準備好，把它驅動起來。
- `.await` 的意思是「等這件事好，期間可以去做別的事」，而不是傻傻卡住。
- 搭配 `tokio::spawn` 把工作丟到背景，async 程式可以同時推進很多連線。

12.2 為什麼需要 async

12.2.1 本集目標

搞懂 async 適合什麼樣的程式，並且分清楚「並行」與「平行」這兩個常被搞混的詞。

12.2.2 正文

12.2.2.1 async 為什麼存在

上一集的伺服器有一個特色：它大部分時間都在**等**。等有人連進來、等資料送達、等回應寫出去。真正用到 CPU 計算的時間少得可憐。

這種「大部分時間都在等」的程式其實非常多：

- 網路伺服器：等客戶端送請求、等資料庫回答。
- 爬蟲：送出一堆網路請求，然後等對方回。
- 資料庫查詢：送出查詢，等結果。
- 聊天室：等每個使用者輸入訊息。
- 各種背景工作：等計時器、等檔案、等別的程式。

這些程式的瓶頸不是「CPU 算不夠快」，而是「花太多時間在等」。async 就是為了這種情境而生的：它讓你的程式在等待某件事的時候，把寶貴的執行緒拿去推進別的工作。

12.2.2.2 為什麼不直接開很多執行緒就好

你可能會想：前幾章不是教過多執行緒嗎？要同時處理一萬個連線，那就開一萬條執行緒不就好了？

問題出在成本。作業系統的執行緒（OS Thread）很**吃記憶體**——每開一條，作業系統就得替它準備一塊 stack 空間，常常是好幾 MB。一萬條執行緒就可能吃掉好幾 GB 的記憶體，這還沒算上作業系統在這麼多執行緒之間切換的負擔。對於一個「大部分時間都在等」的程式來說，開一萬條執行緒、然後讓它們幾乎都在睡覺，實在太奢侈了。

async 的做法不一樣：它可以用**少少幾條**執行緒，輪流推進**成千上萬個**等待中的工作。每個工作不再對應一條笨重的 OS 執行緒，而是一個輕量的東西（我們之後會慢慢看到它的真面目）。這就是 async 能支撐大量連線的原因。

12.2.2.3 並行 vs 平行

這裡要把兩個很容易混淆的詞講清楚，因為它們是理解 async 的關鍵。

並行（concurrency）：同時處理很多件事，但靠的是「交錯切換」。想像一位咖啡師一個人顧好幾桌客人：他幫 A 桌點完餐，趁咖啡機在沖的時候去幫 B 桌點餐，再回頭幫 A 桌收尾。任何一個瞬間他其實只在做一件事，但因為他懂得在「等」的空檔切去做別的，整體看起來好像同時服務了很多桌。一個人、一條執行緒就能做到並行。

平行（parallelism）：同一個瞬間，真的有很多事一起在執行。這需要多個 CPU 核心——就像咖啡店裡站了好幾位咖啡師，每人各顧一桌，真正同一時間一起動手。

這兩件事是**獨立的兩個維度**，可以自由組合：

- 單執行緒的 async：有並行、沒有平行（一位咖啡師輪流服務多桌）。
- 把純計算丟到多個核心：有平行（多位咖啡師），不見得需要 async 的並行。
- 多執行緒的 runtime（Tokio 預設）：兩者都有（多位咖啡師，而且每位都會在空檔切去服務別桌）。

12.2.2.4 async 提供的是「並行」

關鍵結論來了：**async 本身提供的是並行**——它讓你用少少幾條執行緒，交錯推進一大堆正在等待的工作。至於能不能**平行**，那是由 runtime 決定用幾條執行緒來跑這些工作。

所以請記住：async **不會讓你的 CPU 計算變快**。如果你的程式是在做一個很吃資源的數學運算，async 幫不上忙，那是平行（丟到多核心處理器上）才能解決的問題。async 解決的是另一件事——**讓「等待」的時間不被浪費**，在等的空檔切去做別的事。

下一集開始，我們就要把「async 到底是什麼」這件事一層一層拆開來看。

12.2.3 重點整理

- async 適合「大部分時間都在等 I/O」的程式：伺服器、爬蟲、資料庫、聊天室、背景工作。
- OS 執行緒很吃記憶體，用「一個連線一條執行緒」撐不住大量連線；async 用少少幾條執行緒推進大量工作。
- **並行**是交錯切換、同時處理很多事，一條執行緒就能做到（咖啡師比喻）；**平行**是多核心同一瞬間一起執行。
- async 提供的是並行，要不要平行由 runtime 用幾條執行緒決定。
- async 不會讓計算變快，它只是讓等待的時間可以拿去做別的事。

12.3 async fn 回傳的是 Future

12.3.1 本集目標

建立一個關鍵的心智模型：呼叫 async fn 並不會執行它，你只是拿到一個還沒開始跑的 Future。

12.3.2 正文

12.3.2.1 呼叫 async fn 不會執行它

這是新手最常踩的坑，所以我們用實驗來證明。先看一個普通的 async fn：

```
async fn say_hello() {
    println!("hello");
}

#[tokio::main]
async fn main() {
    say_hello(); // 注意：這行不會印出 hello！
}
```

直覺上你會以為呼叫 say_hello() 就會印出 hello，但實際上**什麼都不會發生**。函數體裡的 println! 完全沒有執行。不只如此，編譯器還會給你一個警告：

```
warning: unused implementer of `Future` that must be used
note: futures do nothing unless you `.await` or poll them
```

這個警告（來自 #[must_use]）已經把真相說出來了：呼叫 say_hello() 得到的是一個 **Future**——一個「還沒跑的工作」。你只是把這個工作描述出來，但沒有人去執行它，所以它就被丟掉了。

要真的讓它跑，得加上 .await：

```
async fn say_hello() {
    println!("hello");
}

#[tokio::main]
async fn main() {
    say_hello().await; // 這次才會印出 hello
}
```

12.3.2.2 讓編譯器親口證實它是 Future

如果你還不信，我們可以用另一招逼編譯器說實話：故意把回傳值的型別標錯，看它怎麼罵人。

```
async fn say_hello() {
    println!("hello");
}

#[tokio::main]
async fn main() {
    let x: () = say_hello(); // 編譯錯誤
}
```

say_hello 的函數體沒有回傳值，照理說「應該」回傳 ()，所以我們故意寫 let x: () = ...。但編譯器會報錯：

```
expected `()`, found future
```

它清楚地告訴你：say_hello() 的型別**不是** ()，而是一個 future。這就確認了——呼叫一個 async fn，拿到的是一個 Future，而不是函數體執行後的結果。

12.3.2.3 Future 是惰性的

前面兩段分別看到兩件事：呼叫 say_hello() 不會立刻執行函數體；而且 say_hello() 的回傳值型別不是 ()，是一個 Future。這兩件事合起來，就是這一集最重要的一句話：

> 呼叫 async fn 只是得到一個 Future，這個 Future 是**惰性的**。

「惰性」這個詞你應該不陌生。回想第 6 章的**迭代器**：當你寫 v.iter().map(...).filter(...)，這幾個方法其實一個元素都還沒處理，它們只是把「之後要做的事」描述出來；真正開始跑，是等到你 .collect() 或用 for 走訪的那一刻。

Future 和 Iterator 骨子裡是同一套設計哲學：**先描述，晚執行**。Iterator 描述「一連串值要怎麼算出來」，等你索取才動；Future 描述「一個非同步工作要做什麼」，等你被 runtime 推進才動。

下一集我們換個角度，把 .await 和你早就學過的 ? 放在一起看，你會發現它們其實是同一類東西。

12.3.3 重點整理

- 呼叫 async fn **不會**執行函數體，你只會拿到一個 Future。
- async fn main 裡不做像是 .await 的動作的話，被呼叫的 async 函數一行都不會跑，還會收到 #[must_use] 警告。
- 把回傳值標成 () 會讓編譯器報 expected (), found future，證明它真的是 Future。
- Future 是**惰性的**，和第 6 章的 Iterator 一樣，都是「先描述、晚執行」的設計。

12.4 async 是一種作用

12.4.1 本集目標

換一個角度看 async：把 .await 和你早就會的 ? 放在一起，發現它們其實是同一類東西。

12.4.2 正文

12.4.2.1 兩個「小尾巴」

回想一下第 5 章的 ?。當一個運算式的型別是 Option / Result，你在它後面黏一個 ?，就能把裡面那個「成功的值」拉出來用，而「萬一失敗了怎麼辦」這件事交給編譯器自動處理：

```rust
let x = a.parse::<i32>()?; // ? 把 Result 裡的值拉出來
```

.await 做的事也很像。當一個運算式的型別是 Future，你在它後面黏一個 .await，就能把裡面那個「之後會算好的值」拉出來用，而「萬一還沒好怎麼辦」這件事交給 runtime 自動處理：

```rust
let x = some_async_thing().await; // .await 把 Future 裡的值拉出來
```

看出來了嗎？? 和 .await 都是**黏在運算式後面的小尾巴**，它們把「包在某個特殊世界裡的值」拉到你手上。

12.4.2.2 兩個世界，各有各的規矩

可以把這想成：有些值不是住在「普通世界」，而是住在一個被包起來的特殊世界裡。

- **`Option` / `Result` 世界**：值可能算不出來。這個世界的規矩是「可能失敗」。
- **`Future` 世界**：值可能還沒好，要等。這個世界的規矩是「或許還沒好」。

當你用 `?` 或 `.await` 把值拉出來，你寫的程式看起來就跟普通程式沒兩樣——一行接一行、把值拿來算。但背後編譯器其實在替你做一件事：把這些「包裝過的值」按照各自世界的規矩**串接起來**。每一個 `?` 或 `.await`，就是一道「套用規矩」的接縫：`?` 那道接縫會在出錯時自動提早回傳；`.await` 那道接縫則會在還沒好時自動暫停，把執行緒讓出去。

12.4.2.3 為什麼 `.await` 需要專屬的 `async` 語法

`?` 的規矩比較單純，編譯器只要插入一個「出錯就提早 `return`」的判斷就好。但 `.await` 的規矩複雜得多：在「還沒好」的時候，它得把整個函數**暫停**起來、記住現在跑到哪、把執行緒讓給別人，等好了再從原地接著跑。

要做到這件事，編譯器得把你的 `async` 函數**大幅改寫**成一個叫「狀態機」的東西（本章後面會解釋，現在先記得這個詞就好）。正因為改寫幅度這麼大，Rust 才需要 `async` 這個專屬的關鍵字——它等於是在告訴編譯器：「這一段請幫我改寫成可以暫停、可以恢復的形式」。

12.4.2.4 `async` 會「傳染」

`?` 的限制是你只能在「回傳 `Option` / `Result` 的函數」裡用它。同理，`.await` 也只能在 `async` 的環境裡用。在一個普通函數裡直接 `.await` 會編譯失敗：

```
async fn add(a: i32, b: i32) -> i32 {
    a + b
}

fn normal_function() {
    let sum = add(3, 4).await; // 編譯錯誤：一般函數裡不能 .await
}
```

換句話說，你只能在「世界裡面」拉值。想用 `.await`，你所在的函數自己也得是 `async`——於是 `async` 會一路往上「傳染」。這和 `?` 要求「呼叫端自己也得能處理錯誤」是同一回事。

12.4.2.5 作用總要「落地」

不管哪個世界，總有一刻要回到普通世界——把包裝拆掉、得到一個實實在在的值。先看錯誤處理是怎麼「落地」的，它有兩條路。

第一條：讓 `main` 自己回傳 `Result`，在程式的邊界給編譯器處理。

```
fn parse_and_add(a: &str, b: &str) -> Result<i32, std::num::ParseIntError> {
    let x = a.parse::<i32>()?;
    let y = b.parse::<i32>()?;
    Ok(x + y)
}

fn main() -> Result<(), std::num::ParseIntError> {
    let sum = parse_and_add("3", "4")?;
    println!("結果是 {}", sum);
    Ok(())
}
```

第二條：自己用 match 當場把 Result 拆開，在普通程式裡處理掉。

```
fn parse_and_add(a: &str, b: &str) -> Result<i32, std::num::ParseIntError> {
    let x = a.parse::<i32>()?;
    let y = b.parse::<i32>()?;
    Ok(x + y)
}

fn main() {
    match parse_and_add("3", "4") {
        Ok(sum) => println!("結果是 {}", sum),
        Err(e) => println!("出錯了：{}", e),
    }
}
```

12.4.2.6 async 的落地，完全對應

Future 世界的落地也是這兩條路，而且一一對應：

第一條：#[tokio::main]，對應「回傳 Result 的 main」。你只管把 main 加上 async，用 Tokio 這個框架在程式的邊界處理：

```
extern crate tokio;

async fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[tokio::main]
async fn main() {
    let sum = add(3, 4).await;
    println!("結果是 {}", sum);
}
```

第二條：block_on，對應「自己 match」。你在一個普通的 main 裡，當場叫 runtime 把一個 Future 跑到完成，結算成普通值：

```
extern crate tokio;

async fn add(a: i32, b: i32) -> i32 {
    a + b
}

fn main() {
    let runtime = tokio::runtime::Runtime::new().expect("建立 runtime 失敗");
    let sum = runtime.block_on(add(3, 4)); // 當場把 Future 結算成普通值
    println!("結果是 {}", sum);
}
```

兩兩對照：回傳 Result 的 main ↔ #[tokio::main]（交給框架在邊界結算），match ↔ block_on（自己在同步程式裡當場結算）。把這個對應記在心裡，async 對你來說就不再是全新的東西，而是「你早就會的 ?，但換了一套更複雜的規矩」。

12.4.3 重點整理

- ? 和 .await 都是黏在運算式後的小尾巴，把「特殊世界裡的值」拉出來。

- Result 世界的規矩是「可能失敗」，Future 世界的規矩是「可能還沒好」；編譯器替你把包裝過的值按規矩串接起來。
- .await 的規矩複雜，會把函數改寫成**狀態機**，所以需要 async 專屬語法。
- async 的 .await 像 ? 一樣會「傳染」：要 .await，所在的函數自己也得是 async。
- 落地兩條路一一對應：回傳 Result 的 main ↔ #[tokio::main]，match ↔ block_on。

12.5 async block

12.5.1 本集目標

學會用 async { ... } 在函數裡當場做出一個 Future，並理解它和 async fn 的關係。

12.5.2 正文

12.5.2.1 當場做一個 Future

除了 async fn，Rust 還讓你用 async { ... } 在程式中間**當場**建立一個 Future：

```
#[tokio::main]
async fn main() {
    // 這個 async block 本身就是一個 Future
    let fut = async {
        println!("我在一個 async block 裡");
        42
    };

    // 和 async fn 一樣，要 .await 才會真的跑
    let value = fut.await;
    println!("拿到 {}", value);
}
```

注意：和 async fn 完全一樣，光是寫出 async { ... } 不會執行裡面的程式，你只是做出一個惰性的 Future，要 .await 才會動。

12.5.2.2 async fn 和 async block 的關係

這兩者可以先用一個很粗略的角度來分：

- async fn 是一個**具名的 Future 工廠**——你定義一次，之後可以重複呼叫，每次呼叫產生一個新的 Future。
- async block 是**當場建立的一個匿名 Future**——就在這裡、這一個，沒有名字。

也就是說，如果只看「具名、可重複使用」和「匿名、當場建立」這件事，它們有點像普通函數和閉包的差別。但這只是幫你抓第一印象的比喻，不要對它進行過度理解：async fn 和 async block 都產生 Future，但 async block 本身不是閉包，它不靠 () 呼叫，建立出來後是靠 .await 或 runtime 推進。

12.5.2.3 async move

async block 也可以寫成 async move { ... }。這裡的 move 和閉包的 move 類似：它會把用到的外部變數搬進這個 Future 裡。被搬進去的是變數本身；如果那個變數本來就是參考，那搬進去的也會是那個參考。

```rust
#[tokio::main]
async fn main() {
    let name = String::from("Ferris");

    let fut = async move {
        println!("hello, {}", name);
    };

    fut.await;

    // println!("{}", name); // 編譯錯誤：name 已經被搬進 async block
}
```

如果不用 `move`，async block 通常會盡量借用外面的變數；用了 `async move`，則會把用到的外部變數搬進產生出來的 `Future`。這在你想把 `Future` 存起來、交給 runtime 執行，或讓它離開目前作用域時很常見。

12.5.2.4 在 Result 世界，這件事不需要新語法

這裡有個很有意思的對照。在 `Result` 世界，如果你想要「當場來一段可以用 `?` 的區塊」，其實**不需要任何新語法**——一個立刻呼叫的閉包就辦到了：

```rust
fn main() {
    // 定義一個閉包，然後馬上用 () 呼叫它
    let result: Result<i32, std::num::ParseIntError> = (|| {
        let x = "3".parse::<i32>()?;
        let y = "4".parse::<i32>()?;
        Ok(x + y)
    })();

    println!("{:?}", result);
}
```

這裡的 `(|| { ... })()` 是「定義一個閉包並立刻呼叫」。閉包的函數體可以用 `?`，因為這個閉包自己回傳 `Result`；呼叫完之後，外層的 `main` 只會拿到那個 `Result` 值。

12.5.2.5 為什麼 Future 世界不能照搬

你可能會想：那 `Future` 世界是不是也照抄就好？把 `.await` 塞進一個立刻呼叫的閉包裡？

```rust
async fn get_number() -> i32 {
    42
}

#[tokio::main]
async fn main() {
    let value = (|| {
        get_number().await // 編譯錯誤：普通閉包裡不能 .await
    })();
}
```

不行。原因回到前幾集講的：`.await` 需要把整段程式**改寫成狀態機**，才能做到「暫停以允許並行」。但一個普通閉包只會被編譯成一個普通函數，裡面**沒有**「暫停，之後再恢復」這回事——它表達不了那種改寫。所以 `Result` 世界那招在這裡行不通。

這正是 async block 存在的理由。當你寫 `async { ... }`，等於是明確地叫編譯器：「把這一塊改寫

成一個 Future」。有了這個專屬語法，裡面才能合法地用 .await：

```
extern crate tokio;

async fn get_number() -> i32 {
    42
}

#[tokio::main]
async fn main() {
    let value = async {
        get_number().await // 這次可以了，因為這是 async block
    }.await;
    println!("{}", value);
}
```

到這裡，前五集先把 async fn、.await、async block 這幾個基本語法和心智模型建立好了。下一集開始，我們要捲起袖子，自己動手把 Future 的內部機制拆開來看。

12.5.3 重點整理

- async { ... } 在函數中間當場建立一個匿名的 Future，一樣要 .await 才會跑。
- async fn 是具名、可重複呼叫的 Future 工廠；async block 是當場建立的一個匿名 Future。
- async move { ... } 會把用到的外部變數搬進產生出來的 Future，常用在 Future 需要離開目前作用域或交給 runtime 執行時。
- 如果建立後立刻呼叫的閉包回傳 Result 就可以用 ?；外層只會拿到閉包呼叫後的 Result 值。
- .await 不能照搬這招，因為它只能出現在 async 結構裡，普通閉包做不到暫停和恢復——所以才需要 async block 這個專屬語法。

12.6 Future trait 與最陽春的 executor

12.6.1 本集目標

看懂 Future trait 的正式定義，並親手寫一個最笨，但真的能跑的 executor。

12.6.2 正文

12.6.2.1 Future trait 長什麼樣

前幾集一直講「Future」，現在來看它真正的定義。它是標準庫裡的一個 trait：

```
pub trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

拆開來看：

- type Output 是這個 Future 完成後會給出的值的型別。
- poll 是核心方法。它會問這個 Future：「你好了沒？」
- 回傳值是 Poll，一個只有兩種狀態的 enum：

```
pub enum Poll<T> {
    Ready(T), // 好了，這是結果
    Pending,  // 還沒好，待會再來問
}
```

所以推進一個 Future 的方式就是不斷 poll 它：回 Pending 就代表還沒好，回 Ready(value) 就代表完成，可以把結果拿走。

12.6.2.2 為什麼 poll 的 self 是 Pin<&mut Self>

你大概注意到一個奇怪的地方：poll 的第一個參數不是我們熟悉的 self / &self / &mut self，而是 self: Pin<&mut Self>。

先別緊張，這一章後面會花好幾集慢慢講 Pin 的細節，現在你只要接受一件事：**Pin 是一個很特別的型別**。Rust 規定，除了我們之前認識的 self / &self / &mut self，能放在 self 位置的型別只有一小撮「智慧指標」：

- Box<Self>、Rc<Self>、Arc<Self>。
- 以及 Pin<...>。

一般你自訂的型別**不能**這樣用在 self 位置。poll 之所以能寫成 self: Pin<&mut Self>，正是因為它夠特別。目前你可以先把 Pin<&mut Self> 想成「一個受了限制的 &mut Self」——它讓你能改 Future 的內容，但不准你把它整個搬走。為什麼要有這個限制，之後會講。

12.6.2.3 最陽春的 executor

poll 是 Future 的引擎，但總得有人去發動它——這個「不斷 poll 直到完成」的角色就叫 **executor**。Rust 標準庫**不附**任何 executor，所以我們自己寫一個最笨的版本：

```
use std::future::Future;
use std::task::{Context, Poll, Waker};

fn block_on<F: Future>(future: F) -> F::Output {
    // 把 Future 放到 heap 上並「釘住」，得到 Pin<Box<F>>
    let mut future = Box::pin(future);

    // 做一個 Context，裡面包什麼都不做的 Waker，之後再解釋它是幹嘛的
    let mut cx = Context::from_waker(Waker::noop());

    loop {
        // .as_mut() 把 Pin<Box<F>> 借用成 Pin<&mut F>，正是 poll 要的型別
        match future.as_mut().poll(&mut cx) {
            Poll::Ready(value) => return value, // 好了，把結果回傳
            Poll::Pending => {
                // 還沒好，這個最笨的 executor 就是回去再 poll 一次（空轉）
            }
        }
    }
}

fn main() {
    let value = block_on(async {
        println!("async block 開始跑");
        1 + 2
    });
```

```
    println!("結果是 {}", value);
}
```

12.6.2.4 兩個搬運值的小工具

這個 executor 用了兩個和 Pin 有關的工具，先簡單認識：

Box::pin(x) 的型別是 fn pin(x: T) -> Pin<Box<T>>——把值放到 heap 上，並用 Pin 把它釘住。目前你就把它當成「一個受限的指標」就好。

Pin<Ptr> 上 as_mut 的型別是 fn as_mut(&mut self) -> Pin<&mut <Ptr as Deref>::Target>，對 Pin<Box<T>> 來說就是 -> Pin<&mut T>，剛好就是 poll 需要的 self: Pin<&mut Self>。重點是 as_mut 只是可變借用，沒有把 future 交出去，所以我們的 loop 才能拿同一個 future 反覆 poll。

12.6.2.5 老實說：到目前為止其實「沒在等任何東西」

這裡要誠實交代一件事。從第 3 集到這一集，我們寫的那些 async fn、async block，其實**沒有真的在等什麼**——它們裡面都沒有會卡住的 .await。對於這種 Future，第一次 poll 就會直接回 Ready，我們的 Pending 分支根本不會跑到。

也就是說，前面這些例子純粹是拿來示範 Future 和 executor 的機制，還稱不上是「真正用到 async 功能」的程式。下一集我們要手寫一個 Delay——一個會真的回 Pending、需要等一段時間才完成的 Future，那才是第一個更像樣的非同步工作。

12.6.2.6 executor 有很多種設計

最後提醒一個觀念：Rust 標準庫只定義了 Future trait，**怎麼實作 executor 完全留給 runtime 自由發揮**。我們這集寫的是「回 Pending 就忙著空轉重 poll」的笨版本——超級浪費 CPU 資源。真正的 runtime 會聰明得多：沒事做的時候去睡覺，有事了才被叫醒。

正因為標準庫不規定 executor 怎麼寫，才會有 Tokio、smol 等各有特色的 runtime。接下來幾集，我們會從這個最笨的版本出發，一步一步讓它演進到接近真實 runtime 的樣子。

12.6.3 重點整理

- Future trait 的核心是 poll，回傳 Poll::Ready(value)（好了）或 Poll::Pending（還沒好）。
- poll 的 self 是 Pin<&mut Self>，Pin 是少數能直接放在 self 位置的特別型別；目前先當成「受限的 &mut Self」。
- **executor** 負責不斷 poll 一個 Future 直到 Ready；標準庫不附 executor，要自己做或靠 runtime 提供。
- Box::pin 把值放 heap 並釘住、as_mut 借出 Pin<&mut T>，兩者配合讓 loop 能反覆 poll 同一個 Future。
- 前幾集的 async 其實都沒在等東西，poll 一次就 Ready；下一集的 Delay 才會真的有 Pending 的狀況。
- 標準庫只定義 Future，executor 怎麼寫留給 runtime，這就是 Tokio、smol 等不同 runtime 存在的原因。

12.7 手寫一個 Delay Future

12.7.1 本集目標

親手寫出第一個會真的回 Pending 的 Future——一個計時器 Delay，並用上一集的 executor 跑它。

12.7.2 正文

12.7.2.1 為什麼要做一個 Delay

上一集說好了，這集要寫一個「會真的需要等」的 Future。但真實世界裡需要等的事件——等網路封包、等硬碟、等資料庫——背後都牽扯到作業系統的一堆概念，第一次認識 Pending 就碰這些太複雜了。

所以我們先用最簡單的東西撐著：一個**計時器**。規則很單純：

- 還沒到期 → 回 Pending（事情還沒好）。
- 到期了 → 回 Ready（完成）。

這個 Delay 就是我們接下來好幾集的主角，每當需要一個「要花時間才會好的事件」，我們就拿它來模擬，用它來研究 .await、join、還有 Waker。

12.7.2.2 寫出 Delay

Delay 記住一個「到期時間點」when，每次被 poll 時就比對現在的時間有沒有超過它：

```
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};
use std::time::{Duration, Instant};

struct Delay {
    when: Instant, // 預計完成的時間點
}

impl Delay {
    fn new(duration: Duration) -> Delay {
        Delay {
            when: Instant::now() + duration, // 從現在算起，過 duration 之後到期
        }
    }
}

impl Future for Delay {
    type Output = ();

    fn poll(self: Pin<&mut Self>, _cx: &mut Context<'_>) -> Poll<()> {
        if Instant::now() >= self.when {
            println!("Delay 完成了");
            Poll::Ready(()) // 到期了
        } else {
            Poll::Pending // 還沒到期，待會再來問
        }
    }
}
```

poll 的邏輯就是這麼直接：時間到了回 Ready(())，沒到回 Pending。Output 是 ()，因為這個計時器完成時不需要給出什麼值，純粹是「時間到了」這個事件本身。

12.7.2.3 用我們的 executor 跑跑看

把上一集那個最笨的 block_on 搬過來，就能執行自己寫的 Delay 了：

```
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll, Waker};
use std::time::{Duration, Instant};

struct Delay {
    when: Instant,
}

impl Delay {
    fn new(duration: Duration) -> Delay {
        Delay {
            when: Instant::now() + duration
        }
    }
}

impl Future for Delay {
    type Output = ();

    fn poll(self: Pin<&mut Self>, _cx: &mut Context<'_>) -> Poll<()> {
        if Instant::now() >= self.when {
            println!("Delay 完成了");
            Poll::Ready(())
        } else {
            Poll::Pending
        }
    }
}

fn block_on<F: Future>(future: F) -> F::Output {
    let mut future = Box::pin(future);
    let mut cx = Context::from_waker(Waker::noop());
    loop {
        match future.as_mut().poll(&mut cx) {
            Poll::Ready(value) => return value,
            Poll::Pending => {}
        }
    }
}

fn main() {
    println!("開始");
    block_on(Delay::new(Duration::from_secs(1)));
    println!("一秒過去了");
}
```

跑起來，你會看到「開始」之後停頓一秒，才印出「一秒過去了」。我們第一個會真正回傳 Pending 的 Future 成功了！在這一秒之內，block_on 的迴圈瘋狂地一直 poll 並一直拿到 Pending，直到

時間終於到了才拿到 Ready。

> **注意**：網頁版的程式碼沙盒不一定能清楚顯示出時間延遲。這一章後面還有不少類似的計時、等待範例；如果你想實際感受到「停一秒」「同時等不同工作」這類時間變化，建議把程式複製到自己的電腦上執行。

12.7.2.4 老實說：這個 Delay 被過度簡化了

這個版本能跑，但它其實**偷懶**了——poll 裡的 cx 參數被寫成 _cx，完全沒用到。

cx 裡裝著一個叫 Waker 的東西，正常的 Future 在回 Pending 之前，應該用它通知 executor「等我好了再來叫我」。但我們的 Delay 完全沒這麼做。那為什麼它還能跑？因為我們搭配的 executor 也一樣笨——它根本不睡覺，回 Pending 就馬上再 poll，所以就算沒人通知它也無所謂。

換句話說，這個 Delay 是**綁定**在這個笨 executor 上才能正常運作的。如果把它丟到一個「會睡覺、要被 Waker 叫醒才繼續」的真實 executor 上，它回了 Pending 卻從不通知對方，executor 就會一睡不醒——這個 Delay 等於永遠不會完成。

之後會修掉這個偷懶的部分。但在那之前，我們會先用這個 Delay 把 .await 和並行的一些觀念建立起來。下一集就來看在 async 裡 .await 這個 Delay 會發生什麼事。

12.7.3 重點整理

- 真實 I/O 太複雜，所以先用最簡單的計時器來第一次認識 Pending。
- Delay 用一個計時器模擬「要花時間才會好的事件」：沒到期回 Pending，到期回 Ready，之後幾集都拿它當替代品。
- 自訂 Future 一旦 impl Future、實作 poll，搭配上一集的 block_on 就能跑。
- 這個 Delay 被過度簡化了：poll 沒用到 cx 裡的 Waker，只因為搭配的 executor 也不睡覺才剛好能跑；換到會睡覺的 executor 就會出問題，我們後面會修。

12.8 用 .await 等待 Delay

12.8.1 本集目標

用 .await 等待上一集的 Delay，並透過 println! 親眼看到 Future 是怎麼「暫停又恢復」的。

12.8.2 正文

12.8.2.1 在 .await 前後印訊息

我們已經有了 Delay，也有了 block_on。現在把 Delay 放進一個 async block 裡，用 .await 等它，而且在每個 .await 的前後都加上 println!，看看執行的順序：

```
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll, Waker};
use std::time::{Duration, Instant};

struct Delay {
    when: Instant,
}

impl Delay {
```

```
    fn new(duration: Duration) -> Delay {
        Delay {
            when: Instant::now() + duration
        }
    }
}

impl Future for Delay {
    type Output = ();

    fn poll(self: Pin<&mut Self>, _cx: &mut Context<'_>) -> Poll<()> {
        if Instant::now() >= self.when {
            Poll::Ready(())
        } else {
            Poll::Pending
        }
    }
}

fn block_on<F: Future>(future: F) -> F::Output {
    let mut future = Box::pin(future);
    let mut cx = Context::from_waker(Waker::noop());
    loop {
        match future.as_mut().poll(&mut cx) {
            Poll::Ready(value) => return value,
            Poll::Pending => {}
        }
    }
}

fn main() {
    block_on(async {
        println!("開始");

        println!("等第一個 delay...");
        Delay::new(Duration::from_secs(1)).await;
        println!("第一個 delay 完成，繼續往下");

        println!("等第二個 delay...");
        Delay::new(Duration::from_secs(1)).await;
        println!("第二個 delay 完成，繼續往下");
    });
}
```

跑起來，輸出會像這樣一步一步出現：

```
開始
等第一個 delay...
(停頓一秒)
第一個 delay 完成，繼續往下
等第二個 delay...
(停頓一秒)
第二個 delay 完成，繼續往下
```

12.8.2.2 它是怎麼「暫停又恢復」的

這個輸出順序揭露了 Future 的運作方式。記得整個 async block 本身就是一個 Future，block_on 不斷在 poll 它：

1. 第一次 poll：從頭開始跑，印出「開始」「等第一個 delay...」，然後遇到第一個 .await。這時 Delay 還沒到期，回 Pending——於是整個 async block 也跟著回 Pending，**從這裡暫停**。
2. 接下來 executor 一次又一次 poll，但 Delay 還沒到期，每次都卡在第一個 .await 那裡回 Pending，沒能往下走。
3. 一秒後 Delay 如期回傳 Ready(())，這次 poll 越過第一個 .await，印出「第一個 delay 完成」「等第二個 delay...」，遇到第二個 .await 又回 Pending，**在新的地方暫停**。
4. 再一秒，第二個 Delay 如期回傳 Ready(())，越過第二個 .await，印完最後一句，整個 async block 回 Ready，block_on 結束。

關鍵在於：**每次被 poll，Future 都從上次暫停的地方接著跑**，一路跑到下一個還沒好的 .await 才可能停下。這種「能記住進度、暫停後又從原地恢復」的能力，正是由前面所說的「狀態機」完成——但這集先看現象就好。

12.8.2.3 .await 不會自動幫你並行

注意一個重點：上面兩個 Delay 是**一個接一個**等的，總共花了兩秒。第二個 Delay 是等第一個完成後才開始計時的。

這常讓新手誤會。.await 的意思是「等這件事好」，它**不會**自動把你的程式變成並行。連續寫兩個 .await，就是老老實實地依序等待，不會聰明地「兩個一起等」。

那如果我就是想讓兩個 Delay 同時計時、總共只花一秒，該怎麼辦？這就是下一集的主題——我們要自己動手寫一個能把多個 Future 並行推進的工具。

12.8.3 重點整理

- 在 async 裡用 .await 等待 Delay，搭配 println! 可以看到執行的步進過程。
- 每次被 poll，Future 都從上次暫停處接著跑，直到遇到下一個沒完成的 .await 才回 Pending。
- Future 能記住進度、暫停後從原地恢復，這是背後狀態機的功勞。
- .await **不會**自動並行：連續兩個 .await 會依序等待，想要並行得用別的工具（下一集會講一種）。

12.9 手寫 join

12.9.1 本集目標

自己寫一個 Future，把好幾個 Future 包成一個，讓它們**並行**推進。

12.9.2 正文

12.9.2.1 目標：一起等好幾個 Future

上一集結尾留下一個問題：連續兩個 .await 會依序等待。如果我想讓好幾個工作**同時**進行、一起等它們全部完成，該怎麼辦？

辦法是自己寫一個 Future，我們叫它 JoinAll。它把一整個 Vec 的 Future 收進來，每次被 poll

的時候，就用 for 迴圈把裡面**每一個**還沒完成的 Future 各用 poll 試著推進一次。等到全部都完成了，自己才回 Ready。

12.9.2.2 寫出 JoinAll

```
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll, Waker};
use std::time::{Duration, Instant};

struct Delay {
    when: Instant,
}

impl Delay {
    fn new(duration: Duration) -> Delay {
        Delay {
            when: Instant::now() + duration
        }
    }
}

impl Future for Delay {
    type Output = ();

    fn poll(self: Pin<&mut Self>, _cx: &mut Context<'_>) -> Poll<()> {
        if Instant::now() >= self.when {
            Poll::Ready(())
        } else {
            Poll::Pending
        }
    }
}

fn block_on<F: Future>(future: F) -> F::Output {
    let mut future = Box::pin(future);
    let mut cx = Context::from_waker(Waker::noop());
    loop {
        match future.as_mut().poll(&mut cx) {
            Poll::Ready(value) => return value,
            Poll::Pending => {}
        }
    }
}

type BoxFuture = Pin<Box<dyn Future<Output = ()>>>;

// 把一個 Vec 的 Future 包起來，每個都用 Some 裝著（完成後換成 None）
struct JoinAll {
    futures: Vec<Option<BoxFuture>>,
}

fn boxed<F>(future: F) -> BoxFuture
where
    F: Future<Output = ()> + 'static,
```

```
{
    Box::pin(future)
}

fn join_all(futures: Vec<BoxFuture>) -> JoinAll {
    JoinAll {
        futures: futures.into_iter().map(Some).collect(),
    }
}

impl Future for JoinAll {
    type Output = ();

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        let this = self.get_mut(); // JoinAll 是 Unpin，可以拿回普通的 &mut
        let mut all_done = true;

        for slot in &mut this.futures {
            // 把 Future 暫時取出來（slot 變成 None），poll 一次
            if let Some(mut fut) = slot.take() {
                match fut.as_mut().poll(cx) {
                    Poll::Ready(_) => {
                        // 完成了，就不放回去，slot 維持 None
                    }
                    Poll::Pending => {
                        *slot = Some(fut); // 還沒好，放回去下次再 poll
                        all_done = false;
                    }
                }
            }
        }

        if all_done {
            Poll::Ready(()) // 全部都完成了
        } else {
            Poll::Pending // 還有沒完成的
        }
    }
}

// 一個有「兩個 .await」的工作，所以要 poll 複數次才會完成
async fn worker(id: u32) {
    println!("worker {} 開始", id);
    Delay::new(Duration::from_secs(1)).await;
    println!("worker {} 過了第一秒", id);
    Delay::new(Duration::from_secs(1)).await;
    println!("worker {} 完成", id);
}

fn main() {
    block_on(async {
        let workers = vec![
            boxed(worker(1)),
            boxed(worker(2)),
            boxed(worker(3)),
```

```
        ];
        join_all(workers).await;
        println!("全部 worker 都完成了");
    });
}
```

這裡我們用 `type BoxFuture = Pin<Box<dyn Future<Output = ()>>>` 幫型別取了個短名字；`BoxFuture` 只是型別別名，並沒有多包一層。`dyn Future<Output = ()>` 的意思是：「我不管裡面具體是哪一種 `Future`，只要它完成時回傳 `()` 就好。」`boxed(...)` 呼叫 `Box::pin`，產生 `Pin<Box<F>>`；它宣告的回傳型別再把具體的 `F` 擦除在 `dyn Future<Output = ()>` 後面。這樣每個結果都有相同的 `BoxFuture` 型別，`JoinAll` 裡的 `Vec` 才能裝下它們。

你可能會注意到這行：

```
let this = self.get_mut(); // JoinAll 是 Unpin，可以拿回普通的 &mut
```

`poll` 收到的 `self` 型別是 `Pin<&mut JoinAll>`，不是普通的 `&mut JoinAll`。但有些時候，Rust 允許我們把外面這層 `Pin` 拿掉，還原成裡面原本的可變參考。`get_mut()` 做的就是這件事：把 `Pin<&mut JoinAll>` 變回 `&mut JoinAll`。後面會解釋可以這麼做的正式原因；現在只要知道：拿到普通的 `&mut JoinAll` 之後，我們才能用熟悉的方式修改裡面的 `Vec`。

另一個值得注意的是這行：

```
if let Some(mut fut) = slot.take() { ... }
```

`slot` 的型別是 `&mut Option<BoxFuture>`。`Option::take` 會把 `Option` 裡的值**拿出來**（取得所有權），並且在原本的位置留下 `None`。所以如果 `slot` 原本是 `Some(fut)`，呼叫 `take()` 之後，我們會拿到那個 `Some(fut)`，而 `slot` 會暫時變成 `None`。

這正好符合我們要做的事：先把子 `Future` 拿出來 `poll` 一次。如果它完成了，就不放回去，讓 `slot` 維持 `None`；如果它還沒完成，就用 `*slot = Some(fut)` 放回去，下一輪再繼續 `poll`。

12.9.2.3 它為什麼是並行的

跑起來你會發現：三個 `worker` 幾乎同時開始、同時結束，總共只花**兩秒**，而不是六秒。

原因是 `JoinAll` 的 `poll` 在一輪裡就把三個 `worker` 各推進一次。三個 `Delay` 同時在計時，所以兩秒後三個 `worker` 全部到期。這就是並行——同一段時間裡，三件「都在等」的事一起被推著走。對照上一集，如果你寫成 `worker(1).await; worker(2).await; worker(3).await;`，那會是一個跑完才換下一個，總共六秒。

12.9.2.4 連「要 poll 很多次」的 Future 也照樣推得動

特別注意我們故意挑了 `worker` 這個有**兩個 `.await`** 的工作放進去。這種 `Future` 不是 `poll` 一次就好，得 `poll` 很多很多次（兩個 `Delay` 各要等一秒，期間 executor 會狂 `poll`）才會走完。

而 `JoinAll` 完全不用為這件事操心——它只管「對每個還沒完成的 `Future` 各 `poll` 一次」，至於某個 `Future` 內部卡在第幾個 `.await`、還要 `poll` 幾次才完成，那是那個 `Future` 自己記著的（記得嗎？`Future` 會記住自己的進度）。`JoinAll` 只要重複地一輪一輪 `poll`，每個 `Future` 自然會一步步往前，直到全部回 `Ready`。`poll` 這套設計正有這樣的威力：單純組合 `Future` 的人不必理解被組合者的內部實作細節。

不過，我們的 executor 還是那個瘋狂空轉的笨版本。下一集就來解決這件事——讓 executor 在沒事做的時候去睡覺，等該醒了再被叫醒。

12.9.3 重點整理

- 把多個 Future 並行推進的辦法，是自己寫一個 Future（JoinAll），在 poll 裡用 for 迴圈把每個子 Future 各 poll 一次。
- 完成的子 Future 換成 None，全部都 None（完成）時 JoinAll 才回 Ready。
- JoinAll 不必處理「某個 Future 要 poll 很多次」的情況——子 Future 自己記得進度，只管一輪一輪 poll 即可。

12.10 用 Thread 與 Waker 喚醒 executor

12.10.1 本集目標

讓 executor 學會睡覺：沒事做時 park 起來，等事件完成時用 Waker 把它叫醒。同時搞懂 poll 的兩條重要契約。

12.10.2 正文

12.10.2.1 不要再空轉了

到目前為止，我們的 executor 有個很糟的毛病：拿到 Pending 就馬上再 poll，整條 Thread 被一個其實還在等的工作燒滿。真實的 runtime 不會這樣，它會在沒事做時去**睡覺**，等真的有進展了再被叫醒。

「叫醒」的工具，就是前幾集一直被我們冷落的 Waker。cx.waker() 拿得到一個 Waker，Future 在回 Pending 之前，應該把這個 Waker 交給「負責通知它好了的人」。等事件完成，那個人就呼叫 waker.wake()，把睡著的 executor 叫醒。

這一集我們就讓計時這件事改由**另一條 Thread** 負責：Delay 第一次被 poll 時，spawn 一條 Thread 去 sleep，睡飽了就 wake executor。

12.10.2.2 自己做一個 Waker

先看 Waker 怎麼生出來。標準庫提供一個 Wake trait，你實作它的 wake 方法，描述「被喚醒時該做什麼」，再用 Waker::from 把它轉成 Waker。

我們希望「喚醒」的動作是把 executor 那條 Thread 叫醒，所以做一個記著 executor Thread 的小型別：

```
use std::sync::Arc;
use std::task::Wake;
use std::thread::{self, Thread};

struct ThreadWaker {
    thread: Thread, // executor 那條 Thread
}

impl Wake for ThreadWaker {
    fn wake(self: Arc<Self>) {
        self.thread.unpark(); // 被喚醒 = 把那條 Thread unpark
    }
}
```

注意 wake 的 self 是 Arc<Self>（這也是第 6 集說的、能放在 self 位置的特別型別之一）。Waker::from(Arc::new(...)) 就能把它變成一個 Waker。

12.10.2.3 會睡覺的 executor

有了 ThreadWaker，executor 就能改成「Pending 就 park 睡覺」：

```
use std::sync::Arc;
use std::task::{Context, Poll, Wake, Waker};
use std::thread::{self, Thread};

fn block_on<F: Future>(future: F) -> F::Output {
    let mut future = Box::pin(future);

    // 做一個「會 unpark 目前這條 executor Thread」的 Waker
    let waker = Waker::from(Arc::new(ThreadWaker {
        thread: thread::current(),
    }));
    let mut cx = Context::from_waker(&waker);

    loop {
        match future.as_mut().poll(&mut cx) {
            Poll::Ready(value) => return value,
            Poll::Pending => thread::park(), // 沒事做，睡到被 unpark
        }
    }
}
```

12.10.2.4 會自己叫醒別人的 Delay

最後改寫 Delay：回 Pending 之前，spawn 一條 Thread 去睡，睡醒就 wake：

```
use std::future::Future;
use std::pin::Pin;
use std::sync::Arc;
use std::task::{Context, Poll, Wake, Waker};
use std::thread::{self, Thread};
use std::time::{Duration, Instant};

struct ThreadWaker {
    thread: Thread,
}

impl Wake for ThreadWaker {
    fn wake(self: Arc<Self>) {
        self.thread.unpark();
    }
}

struct Delay {
    when: Instant,
    started: bool, // 計時 Thread 開了沒
}

impl Delay {
    fn new(duration: Duration) -> Delay {
        Delay {
            when: Instant::now() + duration,
            started: false,
        }
```

```
    }
}

impl Future for Delay {
    type Output = ();

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        let this = self.get_mut();
        if Instant::now() >= this.when {
            Poll::Ready(())
        } else {
            if !this.started {
                this.started = true;
                let waker = cx.waker().clone(); // clone 一個 Waker 給計時 Thread
                let when = this.when;
                thread::spawn(move || {
                    let now = Instant::now();
                    if now < when {
                        thread::sleep(when - now);
                    }
                    waker.wake(); // 時間到，叫醒 executor
                });
            }
            Poll::Pending
        }
    }
}

fn block_on<F: Future>(future: F) -> F::Output {
    let mut future = Box::pin(future);
    let waker = Waker::from(Arc::new(ThreadWaker {
        thread: thread::current(),
    }));
    let mut cx = Context::from_waker(&waker);
    loop {
        match future.as_mut().poll(&mut cx) {
            Poll::Ready(value) => return value,
            Poll::Pending => thread::park(),
        }
    }
}

fn main() {
    block_on(async {
        println!("開始");
        Delay::new(Duration::from_secs(1)).await;
        println!("一秒後");
        Delay::new(Duration::from_secs(1)).await;
        println!("兩秒後");
    });
}
```

這次 executor 不再空轉燒 CPU 了——它 `poll` 一次拿到 `Pending` 就 `park` 睡著，整整睡一秒，被計時 `Thread` 的 `wake` 叫醒後才再 `poll`。

12.10.2.5 wake 比 park 早發生會睡死嗎？

這裡有個值得擔心的時序問題。executor 的 poll 回 Pending 之後，到它真的執行 thread::park() 之間，有一個小空檔。萬一計時 Thread 剛好在這個空檔裡 wake → unpark，那 executor 不就「先被叫醒、然後才去睡」，結果這次 unpark 撲了個空、executor 一睡不醒嗎？

不會。unpark 的設計是：如果這條 Thread 還沒在 park，它會**留一張 permit（許可）**。下次這條 Thread 呼叫 park() 時，看到有 permit 就**立刻返回**，根本不睡。所以不管 wake（也就是 unpark）落在 park() 之前還是之後，都不會漏接。正是因為 park / unpark 自帶這個保證，我們才敢直接拿它們來當「睡覺 / 叫醒」的工具。

12.10.2.6 poll 的兩條契約

趁這套 poll / wake 邏輯剛兜好，把標準庫對 Future::poll 的兩條重要契約講清楚：

契約一：只有最近一次 poll 給的 Waker 算數。每次 poll，cx.waker() 拿到的 Waker **可能不一樣**（例如 Task 被搬到別條 Thread 上跑）。所以一個正確的 Future，每次 poll 都該把最新的 Waker 重新存一份，喚醒時用最新的那個。

我們的 Delay 卻偷懶了——靠 started 旗標，它只在第一次 poll 抓一次 Waker 就不管了。這之所以沒出事，是因為我們的 executor 每次交出去的 Waker，**叫醒的都是同一個對象**，所以第一次抓到的那份一直有效。要是叫醒的對象換了人，這個 Delay 就會叫醒失敗——而 Future 不該去賭這件事。第 14 集手寫 I/O 的 Future 時，我們就會老實地每次 poll 都重存一次；Delay 則會一直維持現在這個偷懶的寫法。

契約二：Ready 之後不可以再 poll。一個 Future 一旦回了 Ready，就**不准**再被 poll，否則行為沒有保證（可能 panic、可能卡死）。所以 executor 必須記得：哪個 Future 做完了，就要把它移除、別再碰。我們現在的 block_on 拿到 Ready 就直接 return，自然不會犯規；但等到要同時管很多個 Future 時，這件事就得認真處理了（下一集就會做）。

12.10.2.7 一個 Future 一條 Thread？這不行

最後潑一盆冷水：我們現在是「每個在等的 Delay 都 spawn 一條 Thread」。這顯然不是好辦法——還記得第 2 集說的嗎？Thread 很吃記憶體。如果有一萬個連線在等，就要一萬條 Thread，這**正是 async 一開始想避免的問題**，結果我們又繞回去了。

接下來幾集要把這件事徹底解決。我們會先把每個被 wake 的 Future 包成一個叫 **Task** 的東西，讓它能排回 executor 的一條「ready queue」（待辦佇列）；之後就可以引入 **reactor**，用少少一條或幾條 Thread 盯住大量的 I/O，徹底擺脫「一個工作一條 Thread」。

12.10.3 重點整理

- Future 回 Pending 前該把 cx.waker() 交給「負責通知它的人」，事件完成時呼叫 waker.wake() 叫醒 executor。
- 自製 Waker：實作 Wake trait 的 wake 方法，再用 Waker::from(Arc::new(...)) 轉成 Waker。
- executor 用 thread::park() 睡覺，Waker 用 unpark() 叫醒；unpark 會留 permit，所以 wake 落在 park 前或後都不漏接。
- **契約一**：每次 poll 的 Waker 可能不同，正確的 Future 每次都要重存最新的 Waker（Delay 偷懶只存一次，是過度簡化）。
- **契約二**：Ready 之後不可再 poll，executor 要把完成的 Future 移除。
- 「一個 Future 一條 Thread」太耗資源，下一集起改用 Task + ready queue，再加上 reactor

來解決。

12.11 spawn 與 ready queue

12.11.1 本集目標

導入 Task 這個觀念，讓 executor 能同時養很多個 Future，並用 ready queue（待辦佇列）管理它們。

12.11.2 正文

12.11.2.1 為什麼需要 Task

前幾集的 executor 手上永遠只有**一個** Future，就在迴圈裡反覆 poll 它。但真實的 runtime 要同時養**很多個** Future。

問題來了：當某個 Future 的 Waker 喊「我好了 !」，如果 executor 手上有一堆裸 Future，它怎麼知道是**哪一個**好了、該去 poll 哪一個？光一個 Future 本身，是沒帶這個資訊的。

我們的解法是給每個 Future 配一份「隨身資料」，把它包成一個 **Task**。一個 Task 裝著：

- 它自己的那個 Future
- 它該排回**哪條** ready queue
- 該叫醒**哪條** executor Thread
- 一個避免自己重複排隊的旗標
- 一個標記自己已經完成的旗標

從此 executor 不再直接管 Future，而是管 Task。而所謂 spawn，就是「把一個 Future 包成 Task、交給 executor」。

12.11.2.2 ready queue 與「喚醒」

executor 將會有一條 **ready queue**：裡面排著「現在該被 poll 的 Task」。executor 的工作就是從 queue 裡拿 Task 出來 poll 它的 Future；queue 空了就去睡覺。

當一個 Task 被 wake，它就把**自己**放回 ready queue，然後 unpark 把睡著的 executor 叫醒。注意這個 unpark 只是一個**鬧鈴**——它只說「有事做了，起床 !」，並不指出是哪個 Task 好了。真正「哪些 Task 該被 poll」的資訊，是放在 ready queue 裡的。

12.11.2.3 把它寫出來

這集的程式比較長，但骨架就是上面那幾句話。來看 Task 怎麼把自己排回 queue（這就是它的 Wake 實作），以及 Executor 怎麼顯式提供 spawn 和 block_on：

```
use std::collections::VecDeque;
use std::future::Future;
use std::pin::Pin;
use std::sync::atomic::{AtomicBool, Ordering};
use std::sync::{Arc, Mutex};
use std::task::{Context, Poll, Wake, Waker};
use std::thread::{self, Thread};
use std::time::{Duration, Instant};

struct Delay {
```

```
    when: Instant,
    started: bool,
}

impl Delay {
    fn new(duration: Duration) -> Delay {
        Delay {
            when: Instant::now() + duration,
            started: false,
        }
    }
}

impl Future for Delay {
    type Output = ();

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        let this = self.get_mut();
        if Instant::now() >= this.when {
            Poll::Ready(())
        } else {
            if !this.started {
                this.started = true;
                let waker = cx.waker().clone();
                let when = this.when;
                thread::spawn(move || {
                    let now = Instant::now();
                    if now < when {
                        thread::sleep(when - now);
                    }
                    waker.wake();
                });
            }
            Poll::Pending
        }
    }
}

type Queue = Arc<Mutex<VecDeque<Arc<Task>>>>;

// 一個 Future + 重新排程所需的隨身資料
struct Task {
    future: Mutex<Pin<Box<dyn Future<Output = ()> + Send>>>,
    queue: Queue,
    executor_thread: Thread,
    queued: AtomicBool, // 自己現在排在 queue 裡嗎？
    done: AtomicBool,   // 自己已經完成了嗎？
}

impl Wake for Task {
    fn wake(self: Arc<Self>) {
        // 拿到舊的 queued 值，同時把新的 true 留回去
        if !self.queued.swap(true, Ordering::SeqCst) {
            self.queue.lock().expect("鎖失敗").push_back(self.clone());
            self.executor_thread.unpark(); // 叫醒 executor
```

```
        }
    }
}

struct Executor {
    queue: Queue,
    executor_thread: Thread,
    remaining: usize, // 還沒完成的 Task 數量
}

impl Executor {
    fn new() -> Executor {
        Executor {
            queue: Arc::new(Mutex::new(VecDeque::new())),
            executor_thread: thread::current(),
            remaining: 0,
        }
    }

    // spawn：把一個 Future 包成 Task，排進 executor 的 queue
    fn spawn(&mut self, future: impl Future<Output = ()> + Send + 'static) {
        let task = Arc::new(Task {
            future: Mutex::new(Box::pin(future)),
            queue: self.queue.clone(),
            executor_thread: self.executor_thread.clone(),
            queued: AtomicBool::new(false),
            done: AtomicBool::new(false),
        });

        self.remaining += 1;
        task.wake(); // 新 task 需要第一次排進 ready queue
    }

    fn block_on(&mut self, future: impl Future<Output = ()> + Send + 'static) {
        // 傳進來的 Future 也 spawn 成一個 Task
        self.spawn(future);

        while self.remaining > 0 {
            // 先把 ready queue 清空
            loop {
                let task = self.queue.lock().expect("鎖失敗").pop_front();
                let Some(task) = task else { break };

                if task.done.load(Ordering::SeqCst) {
                    continue; // 完成後才被排進來的過期喚醒，跳過
                }

                task.queued.store(false, Ordering::SeqCst); // poll 前先放掉旗標
                let waker = Waker::from(task.clone());
                let mut cx = Context::from_waker(&waker);
                let mut future = task.future.lock().expect("鎖失敗");

                if future.as_mut().poll(&mut cx).is_ready() {
                    task.done.store(true, Ordering::SeqCst); // 從此所有喚醒都失效
                    self.remaining -= 1; // 完成了
```

```
                }
            }

            // queue 空了。全部 Task 都完成了嗎？
            if self.remaining > 0 {
                // 還有沒完成的，睡覺等人叫醒
                thread::park();
            }
        }
    }
}

fn main() {
    let mut executor = Executor::new();

    executor.spawn(async {
        println!("task A：開始");
        Delay::new(Duration::from_secs(1)).await;
        println!("task A：一秒到");
    });

    executor.block_on(async {
        println!("task B：開始");
        Delay::new(Duration::from_secs(2)).await;
        println!("task B：兩秒到");
    });

    println!("executor 結束");
}
```

跑起來，兩個 Task（A、B）並行推進：A 在第一秒到期、B 在第二秒到期，各自到期時只把**自己**排回 queue 被 `poll` 一次，互不干擾。`block_on` 會等到 executor 裡所有 Task 都完成才回來，所以最後才印出「executor 結束」。

12.11.2.4 `queued` 旗標為什麼用 `swap`

`wake` 裡的 `queued.swap(true, ...)` 和第 9 集的 `Option::take` 很像：它不是單純「讀一個值」，而是**拿到舊值，同時把新值留在原本的位置**。

第 9 集的 `slot.take()` 是「把 `Some(fut)` 拿出來，原本的位置留下 `None`」。這裡的 `queued.swap(true, ...)` 則是「把舊的 `queued` 拿出來，原本的位置留下 `true`」。所以：

- 如果拿到的是 `false`，代表這個 Task 原本**不在** queue 裡，我們就把它 push 進去。
- 如果拿到的是 `true`，代表它已經在 queue 裡了，這次 `wake` 就不用再排一次。

為什麼不能先 `load` 再 `store`？因為 `wake` 可能來自不同 `Thread`。為了保險，`swap` 把「看舊值」和「留下新值」綁成一次 atomic 操作，才不會兩條 `Thread` 都同時看到 `false`、然後把同一個 Task 重複排進 queue。

12.11.2.5 `done` 旗標：兌現契約二

上一集的契約二說：`Future` 回 `Ready` 之後就不准再被 `poll`。當時的 `block_on` 拿到 `Ready` 就直接 `return`，自然不會犯規；但現在 executor 要同時養很多 Task，事情就沒這麼單純了。

威脅來自散落在外面的 Waker 複本。Task 完成後，executor 這邊確實不會再主動排它；可是像 Delay 交給計時執行緒的那份 Waker，executor 收不回來。萬一有人拿著這種過期的 Waker，在 Task **完成之後**才呼叫 wake，這個已完成的 Task 就會被重新排進 ready queue，然後再次被 poll——契約二被打破，而且 remaining -= 1 還會多扣一次。

所以 Task 還需要一個 done 旗標，讓過期的喚醒失效。防線設在 executor 這一側：pop 出 Task 之後先檢查 done，是 true 就直接 continue 跳過。這樣過期的喚醒頂多讓 Task 多排進 queue 一次，絕不會讓它再被 poll。而因為只有 executor 這條 Thread 會 poll Future、也只有它會把 done 設成 true，所以「設完 done 之後絕不再 poll」是嚴格成立的。

老實說，這一集的範例其實觸發不了這個問題——每個計時器只 fire 一次，而且都在 Task 完成之前。但 executor 的正確性不能靠這種巧合。

12.11.2.6 為什麼 Future 欄位要是 Send

你可能還會注意到 Task 的 future 欄位型別寫成 Mutex<Pin<Box<dyn Future<Output = ()> + Send>>>，為什麼要 Send？

順著推一遍就懂了：Future 被收進 Task，而 Task 又 impl Wake 兼任 Waker（理論上不一定要讓 Task 自己當 Waker，但這樣寫最省事）。Waker::from(Arc<Task>) 這個轉換要求 Task: Send + Sync + 'static。一個型別要 Send + Sync，它的**每個欄位**都得是 Send + Sync——包括那個 Future。

於是 dyn Future 得加上 + Send（讓它能被搬到別條 Thread），外面再包一層 Mutex（Mutex<T> 在 T: Send 時自動是 Sync）。上一集的 Waker 因為構造簡單，我們不必煩惱這些 bound；這集 Task 自己當 Waker，就得認真對待了。

下一集我們在這個基礎上，讓 spawn 能回傳結果——加上 JoinHandle。

12.11.3 重點整理

- 把每個 Future 包成 **Task**（Future ＋排程隨身資料），executor 從此管 Task 而非裸 Future。
- **ready queue** 排著該被 poll 的 Task；Task 被 wake 時把自己排回 queue 再 unpark executor。
- unpark 只是「起床」的鬧鈴，不說哪個 Task 好了；那資訊在 ready queue 裡。
- spawn 是 Executor 的方法：把 Future 包成 Task，排進自己的 ready queue。
- queued.swap(true, ...) 像 Option::take：拿到舊值、留下新值，且是一次 atomic 操作，避免同一個 Task 重複入列。
- Task 完成後可能還有收不回來的 Waker 複本帶來**過期的喚醒**；done 旗標守住契約二——executor pop 出 Task 先檢查 done，poll 到 Ready 就把它設起來，之後的喚醒全部失效。
- Task 自己當 Waker，Waker::from(Arc<Task>) 要求 Task: Send + Sync + 'static，所以 Future 欄位要 + Send 並用 Mutex 包起來。

12.12 spawn 與 JoinHandle

12.12.1 本集目標

讓 spawn 出去的 Task 能把結果回傳，方法是加上 JoinHandle——一個可以 .await 的等待端。

12.12.2 正文

12.12.2.1 和上一集只差三樣東西

上一集的 spawn 有個遺憾：它只收 Future<Output = ()>，工作做完就做完了，沒辦法把結果交回來。這集來補上。

好消息是，排程的核心邏輯**完全不動**，我們只在上面加三樣東西：

1. 新增一個共享狀態 Shared<T>，和一個 JoinHandle<T>（後者也是一個 Future）。
2. Executor::spawn 從只收 Future<Output = ()>，升級成收 Future<Output = T> 並回傳 JoinHandle<T>。
3. Executor::block_on 從回傳 ()，升級成回傳「傳進去那個 Future 的值」T。

12.12.2.2 完成的一方，怎麼通知等待的一方

核心問題是：背景 Task 完成時，怎麼把結果交給「正在 .await 它的人」？

答案是**透過一塊共享狀態 Shared<T>**，而不是 Future 直接通知 Future。Shared<T> 裡放兩樣東西：算好的結果，以及正在 poll JoinHandle 的 Task 所擁有的 Waker。

底下的流程是這樣的：

- JoinHandle<T> 本身是一個 Future，但建立它時不會自動把它排成另一個 Task。在這個例子中，executor poll 傳給 block_on 的 async block，跑到 handle.await 時才會 poll 它。如果直接把它傳給 block_on，block_on 也會像處理其他輸入的 Future 一樣，把它包進一個 Task。
- 某個 Task poll JoinHandle 時，如果結果還沒好，JoinHandle 就把 cx.waker()——也就是目前這個 Task 的 Waker——存進 Shared<T>，然後回傳 Pending。
- 背景 Task 完成時，會把結果放進 Shared<T>，再對存放在裡面的 Waker 呼叫 wake。這會把剛才 poll JoinHandle 的 Task 排回 ready queue，並 unpark executor；下次這個 Task 被 poll 時，就能取得結果。

```
use std::collections::VecDeque;
use std::future::Future;
use std::pin::Pin;
use std::sync::atomic::{AtomicBool, Ordering};
use std::sync::{Arc, Mutex};
use std::task::{Context, Poll, Wake, Waker};
use std::thread::{self, Thread};
use std::time::{Duration, Instant};

struct Delay {
    when: Instant,
    started: bool,
}

impl Delay {
    fn new(duration: Duration) -> Delay {
        Delay {
            when: Instant::now() + duration,
            started: false,
        }
    }
}
```

```
impl Future for Delay {
    type Output = ();

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        let this = self.get_mut();
        if Instant::now() >= this.when {
            Poll::Ready(())
        } else {
            if !this.started {
                this.started = true;
                let waker = cx.waker().clone();
                let when = this.when;
                thread::spawn(move || {
                    let now = Instant::now();
                    if now < when {
                        thread::sleep(when - now);
                    }
                    waker.wake();
                });
            }
            Poll::Pending
        }
    }
}

type Queue = Arc<Mutex<VecDeque<Arc<Task>>>>;

struct Task {
    future: Mutex<Pin<Box<dyn Future<Output = ()> + Send>>>,
    queue: Queue,
    executor_thread: Thread,
    queued: AtomicBool,
    done: AtomicBool, // Task 已經完成了嗎？(守住契約二)
}

impl Wake for Task {
    fn wake(self: Arc<Self>) {
        if !self.queued.swap(true, Ordering::SeqCst) {
            self.queue.lock().expect("鎖失敗").push_back(self.clone());
            self.executor_thread.unpark();
        }
    }
}

// 背景 Task 與它的 JoinHandle 共用的狀態
struct Shared<T> {
    state: Mutex<(Option<T>, Option<Waker>)>, // (結果, 目前 Task 的 Waker)
}

struct JoinHandle<T> {
    shared: Arc<Shared<T>>,
}

impl<T> Future for JoinHandle<T> {
```

```
    type Output = T;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<T> {
        let mut state = self.shared.state.lock().expect("鎖失敗");
        if let Some(value) = state.0.take() {
            Poll::Ready(value) // 結果好了
        } else {
            state.1 = Some(cx.waker().clone()); // 還沒好，存目前 Task 的 Waker
            Poll::Pending
        }
    }
}

struct Executor {
    queue: Queue,
    executor_thread: Thread,
    remaining: usize,
}

impl Executor {
    fn new() -> Executor {
        Executor {
            queue: Arc::new(Mutex::new(VecDeque::new())),
            executor_thread: thread::current(),
            remaining: 0,
        }
    }

    // spawn<T>：收 Future<Output = T>，回傳 JoinHandle<T>
    fn spawn<T, F>(&mut self, future: F) -> JoinHandle<T>
    where
        F: Future<Output = T> + Send + 'static,
        T: Send + 'static,
    {
        let shared = Arc::new(Shared { state: Mutex::new((None, None)) });
        let shared_for_task = shared.clone();

        // 把 Future<Output = T> 包成 executor 看得懂的 Future<Output = ()>
        let task_future = async move {
            let value = future.await; // 真正跑那個工作
            let mut state = shared_for_task.state.lock().expect("鎖失敗");
            state.0 = Some(value); // 放進結果
            if let Some(waker) = state.1.take() {
                waker.wake(); // 叫醒在等的人
            }
        };

        let task = Arc::new(Task {
            future: Mutex::new(Box::pin(task_future)),
            queue: self.queue.clone(),
            executor_thread: self.executor_thread.clone(),
            queued: AtomicBool::new(false),
            done: AtomicBool::new(false),
        });
```

```
        self.remaining += 1;
        task.wake();

        JoinHandle { shared }
    }

    fn block_on<T, F>(&mut self, future: F) -> T
    where
        F: Future<Output = T> + Send + 'static,
        T: Send + 'static,
    {
        let handle = self.spawn(future); // 也 spawn 成 Task，保留 JoinHandle

        // 跑到所有 Task 完成（迴圈和上一集一模一樣）
        while self.remaining > 0 {
            loop {
                let task = self.queue.lock().expect("鎖失敗").pop_front();
                let Some(task) = task else { break };

                if task.done.load(Ordering::SeqCst) {
                    continue;
                }

                task.queued.store(false, Ordering::SeqCst);
                let waker = Waker::from(task.clone());
                let mut cx = Context::from_waker(&waker);
                let mut future = task.future.lock().expect("鎖失敗");

                if future.as_mut().poll(&mut cx).is_ready() {
                    task.done.store(true, Ordering::SeqCst);
                    self.remaining -= 1;
                }
            }

            if self.remaining > 0 {
                thread::park();
            }
        }

        // 從 Shared 取出結果回傳
        handle.shared.state.lock().expect("鎖失敗").0.take().expect("結果還沒好")
    }
}

fn main() {
    let mut executor = Executor::new();

    // spawn 一個回傳 i32 的背景 Task
    let handle = executor.spawn(async {
        Delay::new(Duration::from_secs(1)).await;
        println!("背景 task：算好了");
        21 * 2
    });

    let result = executor.block_on(async move {
```

```
        // 在這裡 .await 背景 Task 的 JoinHandle，取得結果
        let value = handle.await;
        println!("main task：拿到背景結果 {}", value);

        value + 100 // 自己再回傳一個值
    });
    println!("block_on 回傳：{}", result);
}
```

12.12.2.3 一步步看它怎麼跑

假設 A 是上面那個背景 Task：它等一秒後算出 42。B 是傳給 block_on 的那個 Task：它 .await A 的 JoinHandle，拿到結果後再回傳 142。

1. executor.spawn(A)：spawn 先建立一層 task_future，負責等待 A、把結果寫進 Shared<T>、喚醒等待者。真正放進 ready queue 的是這層包裝後的 Task；spawn 接著立刻回傳一個 JoinHandle<i32>。
2. executor.block_on(B)：B 也被 spawn 成一個 Task，放進 ready queue；block_on 自己保留 B 的 JoinHandle，最後要從裡面取出 B 的回傳值。
3. executor 先 poll A 這個 Task。實際被 poll 的是外層 task_future；它跑到 let value = future.await，才開始 poll 內層真正的 A。內層 A 跑到 Delay::new(...).await，開始 poll Delay；Delay 還沒完成，所以回 Pending。這個 Pending 一路傳回外層 task_future，A 這次 poll 就結束了。
4. A 回 Pending 後，executor 沒有睡覺，因為 ready queue 裡還有 B。它立刻 poll B 這個 Task。同樣地，先被 poll 的是 B 外面的 task_future；它跑到 let value = future.await，才開始 poll 傳給 block_on 的那個 async block。
5. B 內層的 async block 跑到 handle.await，於是 poll A 的 JoinHandle。這時 A 的結果還沒好，JoinHandle 把 B 的 Waker 存進 Shared<T>，回 Pending。這個 Pending 一路傳回 B 外面的 task_future，B 也先停住。
6. ready queue 空了，executor 用 thread::park() 睡著。
7. 約一秒後，A 的計時 Thread 呼叫 A 的 Waker，A 被排回 ready queue，executor 被 unpark 叫醒。
8. executor 再 poll A。這次還是先 poll A 外面的 task_future，它繼續 poll 內層 A；Delay 已經完成，所以 A 從 .await 後面繼續跑，先印出 背景 task：算好了，再算出 42。
9. A 外面的 task_future 拿到 42，把它放進 Shared<T>，再取出剛剛存著的 B 的 Waker 並 wake 它。這不是直接繼續執行 B，而是把 B 排回 ready queue。
10. executor 接下來 poll B。B 外面的 task_future 繼續 poll 內層 async block；這次 handle.await 從 Shared<T> 取到 42，印出 main task：拿到背景結果 42，然後 B 回傳 142。
11. B 自己外面那層 task_future 把 142 寫進 B 自己的 Shared<T>。所有 Task 都完成了，block_on 從 B 的 JoinHandle 裡取出 142 回傳，最後印出 block_on 回傳：142。

12.12.2.4 cx.waker() 到底是誰的 Waker

走完這一遍，可以補上一個你可能沒注意到、但很重要的一點。executor poll 某個 Task 時，會先用這個 Task 建出一個 Waker，放進 Context 裡；接著這個 Context 會沿著外層 task_future、內層 async block，再到 .await 前面的 Future 一路傳下去。也就是說，一個 Task 裡被一路 poll 到的 Future，共用的都是同一個 Task 的 Waker。

這其實也接近 Task 作為排程單位的意義：被排回 ready queue、被 executor 再次 poll 的是 Task，不是裡面某一個單獨的 Future。所以內層 Future 要登記喚醒方式時，除了目前這個 Task 的 Waker，也沒有其他更合理的 Waker 可以用。

套回上面的流程，就能看清這裡出現的兩個不同的 Waker：A 被 poll 時（步驟 3），Delay 從 cx.waker() 拿到的是 **A 的** Waker，交給計時 Thread，計時到就叫醒「做事的 A」（步驟 7）；B 被 poll 時（步驟 5），JoinHandle 從 cx.waker() 拿到的是 **B 的** Waker，存進 Shared<T>，A 完成後用它叫醒「等結果的 B」（步驟 9）。來源不同，但最後都走同一條路：把對應的 Task 排回 ready queue，再 unpark executor。

12.12.2.5 不是 Future 直接通知 Future

請注意這點：JoinHandle 和背景 Task 之間沒有直接連線，它們只共用一塊 Shared<T>。等待的一方把自己的 Waker 留在共享狀態裡，完成的一方做完後從共享狀態取出這個 Waker、把它 wake。所有的喚醒，最後都還是回到「排回 ready queue + unpark executor」這條老路上。

到這裡，我們手寫的 executor 已經有模有樣了：能 spawn、能睡覺、能被叫醒。但它還缺一塊大拼圖——目前「等待」靠的還是替每個 Delay 開一條 Thread。下一集起，我們要引入 mio 和 reactor，用少少幾條 Thread 盯住真正的 I/O。

12.12.3 重點整理

- JoinHandle<T> 是一個 Future，.await 它就能拿到背景 Task 的回傳值。
- 排程核心不變，只加三樣：Shared<T> + JoinHandle<T>、回傳 JoinHandle<T> 的 Executor::spawn、回傳 T 的 Executor::block_on。
- executor poll 某個 Task 時，Context 會一路傳到內層 Future；所以內層 Future 看到的 cx.waker() 就是目前這個 Task 的 Waker。
- JoinHandle 沒有自己的 Waker；它被 poll 並回傳 Pending 時，會把目前這個 Task 的 Waker 存進 Shared<T>。
- 背景 Task 完成時會把結果放進 Shared<T>，再對那個 Waker 呼叫 wake，讓剛才 poll JoinHandle 的 Task 重新排進 ready queue。
- 喚醒不是 Future 直接通知 Future，而是完成方透過共享狀態喚醒等待方。

12.13 mio

12.13.1 本集目標

認識 mio——讓「一條 Thread 盯住一大堆 I/O 來源」成為可能的工具，它是下一集 reactor 的基礎。

12.13.2 正文

12.13.2.1 reactor 在 runtime 裡的角色

先退一步，看看我們正在打造的 runtime 全貌。一個 runtime 其實有兩個各司其職的角色：

- **executor**：從 ready queue 拿 Task 出來 poll，也就是「跑 Task」。它對外部世界一無所知——它不知道網路封包到了沒、檔案讀好了沒。
- **reactor**：負責盯住所有的 I/O 來源，哪個 ready 了就 wake 對應的 Task，也就是「等事件」。它**不** poll Future、它不是 Task，它只盯著外部事件來源。

前幾集我們的「等待」是靠替每個 Delay 開一條 Thread，這太浪費了。reactor 的任務，就是用**一**

條 Thread 盯住**很多** I/O 來源。要做到這件事，靠的就是這一集的主角：mio。

12.13.2.2 mio 的兩個主角

mio 是 Rust 生態裡負責跨平台 I/O 事件通知的底層 crate（Tokio 內部也是用它）。使用前要先加上依賴：

```
[dependencies]
mio = { version = "1", features = ["os-poll", "net"] }
```

先注意一件事：mio 本身**不是** async。它不認得 async fn，不會幫你建立 Future，也不會替你 .await。它提供的是比較底層的能力：例如把 socket 設成非阻塞，登記到 Poll，然後讓「呼叫 poll.poll(...) 的那條 Thread」睡著等「socket 是不是 ready 了」。

我們這集只要認識它的兩個東西：

- **mio::Poll**：一個可以「睡著等 I/O 事件」的地方。一條 Thread 把很多 I/O 來源登記給它之後，就能用一次 poll.poll(...) 同時盯住全部，哪個有動靜就醒來。
- **Token**：事件來源的「名牌」。登記某個 I/O 來源時，你給它一個 Token；之後 Poll 通知你「有事件」時，會把當初的 Token 還給你，你就知道是哪個來源在叫。

12.13.2.3 看 mio 怎麼盯著一個 TcpListener

下面這個例子把一個 TcpListener（負責接受連線的東西）登記給 Poll，然後另開一條 Thread 過一秒去連它。主 Thread 就睡在 poll.poll() 上，等到 listener 回報 ready 才醒：

```
extern crate mio;

use mio::net::TcpListener;
use mio::{Events, Interest, Poll, Token};
use std::time::Duration;

// 給 listener 的名牌
const SERVER: Token = Token(0);

fn main() {
    let mut poll = Poll::new().expect("建立 Poll 失敗");
    let mut events = Events::with_capacity(128); // 一次最多收 128 個事件

    let addr = "127.0.0.1:8080".parse().expect("位址解析失敗");
    let mut listener = TcpListener::bind(addr).expect("綁定失敗");

    // 把 listener 登記給 Poll：名牌是 SERVER，我們關心「可讀」事件
    //  （有人連進來就算可讀）
    poll.registry()
        .register(&mut listener, SERVER, Interest::READABLE)
        .expect("登記失敗");

    // 另一條 thread 過一秒後連進來
    std::thread::spawn(|| {
        std::thread::sleep(Duration::from_secs(1));
        let _ = std::net::TcpStream::connect("127.0.0.1:8080");
    });

    println!("睡在 poll 上，等 I/O 事件……");
    loop {
```

```
        // poll 會睡在這裡，直到有登記過的來源發生事件
        poll.poll(&mut events, None).expect("poll 失敗");

        for event in events.iter() {
            match event.token() {
                SERVER => {
                    // 名牌對上了，表示 listener 回報可讀，所以試著 accept
                    match listener.accept() {
                        Ok((_stream, addr)) => {
                            println!("有人連進來了：{}", addr);
                            return; // 因為是範例直接收工
                        }
                        // readiness event 可能是假陽性，回去等下一次即可
                        Err(e) if e.kind() == std::io::ErrorKind::WouldBlock => {}
                        Err(e) => panic!("accept 失敗：{}", e),
                    }
                }
                _ => {}
            }
        }
    }
}
```

12.13.2.4 把流程看一遍

1. `Poll::new()` 做出一個 `Poll`。
2. `registry().register(&mut listener, SERVER, Interest::READABLE)` 把 `listener` 登記進去，給它名牌 `SERVER`，並說明我們關心的是「可讀」（`Interest::READABLE`）。如果是要等「可寫」，就用 `Interest::WRITABLE`。
3. `poll.poll(&mut events, None)` 讓這條 Thread **睡著**，直到有登記過的來源發生事件（`None` 代表不設逾時，睡到有事為止）。
4. 醒來後，逐一檢查 `events`。`event.token()` 還給我們當初登記的名牌；對上 `SERVER`，就知道是 `listener` 回報 ready，於是試著呼叫 `accept()`：
   - 成功就代表真的接到一條連線。
   - 回傳 `WouldBlock` 代表目前仍然無法接受連線，所以回去等待下一次 event。readiness 通知可能是**假陽性**，這是正常情況，不是故障。
   - 其他錯誤才是真的出問題，這個簡化範例直接 panic。

`mio` 的 socket 是**非阻塞**的：呼叫 `accept` 或 `read` 時，不會把 Thread 卡在那裡等待連線或資料。如果操作目前還做不了，就會立刻回傳 `WouldBlock`。在這一集，這代表回到 `poll.poll()` 等待；下一集把 I/O 包成 `Future` 後，同樣的情況就會對應到 `Poll::Pending`。

關鍵在於：就算你登記了**一百個** I/O 來源，也只要**一條** Thread 睡在同一個 `poll.poll()` 上。哪個來源有事，`Poll` 就把對應的名牌交給你。這正是 reactor 用少少幾條 Thread 盯住大量 I/O 的祕密武器。

下一集，我們就把 `mio` 接到前面手寫的 runtime 上，做出真正的 reactor，讓我們的 runtime 第一次能處理真實的網路 I/O。

12.13.3 重點整理

- runtime 有兩個角色：**executor** 跑 Task（poll），**reactor** 等事件（盯 I/O、wake 對應 Task）；reactor 不是 Task、不 poll Future。
- mio 本身不是 async runtime：它只做非阻塞 I/O 的事件通知，不建立 Future、不 .await、也不排程 Task。
- mio::Poll 是「睡著等 I/O 事件」的地方，一條 Thread 就能同時盯住很多 I/O 來源。
- Token 是事件來源的名牌：登記時給，事件發生時 Poll 還給你，讓你認出是哪個來源。
- 用 registry().register(&mut source, token, Interest::READABLE) 登記，poll.poll() 睡著等事件，event.token() 認名牌後再嘗試 I/O 操作。如果回傳 WouldBlock，就繼續等下一次 event；放進 Future 後，這會對應到 Poll::Pending。

12.14 手寫 reactor

12.14.1 本集目標

把前幾集的喚醒功能接到**真實的 I/O**——做出一個 reactor，讓我們的 runtime 第一次能處理網路連線。

12.14.2 正文

12.14.2.1 executor 一行不改

這集有件事能讓人安心：**executor 完全沿用第 12 集**。Task、Executor::spawn<T>、JoinHandle<T>、Shared<T>、Executor::block_on 一行都不用改。

我們唯一要換掉的是「誰來 wake」。前面是 Delay 自己開一條計時 Thread 來 wake；現在改成一條 **reactor Thread**，它睡在 mio::Poll 上等真實的 I/O，醒來後找到對應的 Waker 把它 wake()。

要加的東西是一個 Reactor，以及兩個 I/O Future（Accept 和 Read）。

12.14.2.2 Reactor 與 I/O Future

Reactor 跑在自己的 Thread 上，睡在 mio::Poll 上。而那些跑在 executor Thread 上的 Future，要怎麼跟它溝通？答案是**透過共享狀態，而不是傳訊息**。三樣東西用 Arc 共用：

- **Registry**（mio 的）：Future 拿它直接登記 / 取消 socket。
- **AtomicUsize**：reactor 用它替每個來源自分配獨一無二的 Token。
- **Mutex<HashMap<Token, Waker>>**：Future 在執行時把自己的 Waker 寫進去（用 Token 當鑰匙），reactor 收到事件後就照 Token 取出來 wake。

```
use std::collections::{HashMap, VecDeque};
use std::future::Future;
use std::io::Read as _;
use std::pin::Pin;
use std::sync::atomic::{AtomicBool, AtomicUsize, Ordering};
use std::sync::{Arc, Mutex};
use std::task::{Context, Poll, Wake, Waker};
use std::thread::{self, Thread};
use mio::event::Source;
use mio::net::{TcpListener, TcpStream};
use mio::{Events, Interest, Poll as MioPoll, Registry, Token};
```

```
type Queue = Arc<Mutex<VecDeque<Arc<Task>>>>;

struct Task {
    future: Mutex<Pin<Box<dyn Future<Output = ()> + Send>>>,
    queue: Queue,
    executor_thread: Thread,
    queued: AtomicBool,
    done: AtomicBool,
}

impl Wake for Task {
    fn wake(self: Arc<Self>) {
        if !self.queued.swap(true, Ordering::SeqCst) {
            self.queue.lock().expect("鎖失敗").push_back(self.clone());
            self.executor_thread.unpark();
        }
    }
}

struct Shared<T> {
    state: Mutex<(Option<T>, Option<Waker>)>,
}

struct JoinHandle<T> {
    shared: Arc<Shared<T>>,
}

impl<T> Future for JoinHandle<T> {
    type Output = T;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<T> {
        let mut state = self.shared.state.lock().expect("鎖失敗");
        if let Some(value) = state.0.take() {
            Poll::Ready(value)
        } else {
            state.1 = Some(cx.waker().clone());
            Poll::Pending
        }
    }
}

struct Executor {
    queue: Queue,
    executor_thread: Thread,
    remaining: usize,
}

impl Executor {
    fn new() -> Executor {
        Executor {
            queue: Arc::new(Mutex::new(VecDeque::new())),
            executor_thread: thread::current(),
            remaining: 0,
        }
    }
```

```
    fn spawn<T, F>(&mut self, future: F) -> JoinHandle<T>
    where
        F: Future<Output = T> + Send + 'static,
        T: Send + 'static,
    {
        let shared = Arc::new(Shared { state: Mutex::new((None, None)) });
        let shared_for_task = shared.clone();

        let task_future = async move {
            let value = future.await;
            let mut state = shared_for_task.state.lock().expect("鎖失敗");
            state.0 = Some(value);
            if let Some(waker) = state.1.take() {
                waker.wake();
            }
        };

        let task = Arc::new(Task {
            future: Mutex::new(Box::pin(task_future)),
            queue: self.queue.clone(),
            executor_thread: self.executor_thread.clone(),
            queued: AtomicBool::new(false),
            done: AtomicBool::new(false),
        });

        self.remaining += 1;
        task.wake();

        JoinHandle { shared }
    }

    fn block_on<T, F>(&mut self, future: F) -> T
    where
        F: Future<Output = T> + Send + 'static,
        T: Send + 'static,
    {
        let handle = self.spawn(future);

        while self.remaining > 0 {
            loop {
                let task = self.queue.lock().expect("鎖失敗").pop_front();
                let Some(task) = task else { break };

                if task.done.load(Ordering::SeqCst) {
                    continue;
                }

                task.queued.store(false, Ordering::SeqCst);
                let waker = Waker::from(task.clone());
                let mut cx = Context::from_waker(&waker);
                let mut future = task.future.lock().expect("鎖失敗");

                if future.as_mut().poll(&mut cx).is_ready() {
                    task.done.store(true, Ordering::SeqCst);
```

```
                    self.remaining -= 1;
                }
            }

            if self.remaining > 0 {
                thread::park();
            }
        }

        handle.shared.state.lock().expect("鎖失敗").0.take().expect("結果還沒好")
    }
}

struct Reactor {
    registry: Registry, // Future 用它登記 / 取消 socket
    next_token: AtomicUsize, // 自分配 Token
    wakers: Mutex<HashMap<Token, Waker>>, // Token -> 等待中的 Waker
}

impl Reactor {
    fn unique_token(&self) -> Token {
        Token(self.next_token.fetch_add(1, Ordering::Relaxed))
    }

    fn register(&self, source: &mut impl Source, token: Token, interest: Interest) {
        self.registry.register(source, token, interest).expect("登記失敗");
    }

    fn deregister(&self, source: &mut impl Source) {
        self.registry.deregister(source).expect("取消登記失敗");
    }

    fn set_waker(&self, token: Token, waker: Waker) {
        self.wakers.lock().expect("鎖失敗").insert(token, waker);
    }

    fn clear_waker(&self, token: Token) {
        self.wakers.lock().expect("鎖失敗").remove(&token);
    }

    // 跑在自己的 Thread 上：睡在 poll 上，醒來照 Token 找 Waker 來 wake
    fn run(&self, mut poll: MioPoll) {
        let mut events = Events::with_capacity(128);
        loop {
            poll.poll(&mut events, None).expect("poll 失敗");
            for event in events.iter() {
                let waker = self
                    .wakers
                    .lock()
                    .expect("鎖失敗")
                    .remove(&event.token());

                if let Some(waker) = waker {
                    waker.wake();
                }
```

```
            }
        }
    }
}

fn start_reactor() -> Arc<Reactor> {
    let poll = MioPoll::new().expect("建立 Poll 失敗");
    let registry = poll.registry().try_clone().expect("複製 Registry 失敗");
    let reactor = Arc::new(Reactor {
        registry,
        next_token: AtomicUsize::new(0),
        wakers: Mutex::new(HashMap::new()),
    });
    // reactor 跑在自己的 Thread 上
    let reactor_for_thread = reactor.clone();
    std::thread::spawn(move || reactor_for_thread.run(poll));
    reactor
}

// 開始實作新的 Future

struct Accept {
    reactor: Arc<Reactor>,
    listener: TcpListener,
    listener_token: Token,
}

impl Accept {
    fn new(reactor: Arc<Reactor>, mut listener: TcpListener) -> Accept {
        let listener_token = reactor.unique_token();
        reactor.register(&mut listener, listener_token, Interest::READABLE);
        Accept { reactor, listener, listener_token }
    }
}

impl Future for Accept {
    type Output = TcpStream;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<TcpStream> {
        let this = self.get_mut();
        // 順序刻意是「先登記 Waker，再試 accept」。
        // 如果先 accept 得到 WouldBlock，才準備登記 Waker，
        // 連線可能剛好在中間進來；reactor 那時找不到 Waker 可叫醒，
        // executor 就可能睡過頭。
        this.reactor.set_waker(this.listener_token, cx.waker().clone());
        match this.listener.accept() {
            Ok((stream, _addr)) => {
                // 這次 poll 可能「先登記、再立刻成功」。
                // 成功後就不需要再等 I/O 事件，所以要把剛剛存進去的 Waker 清掉。
                this.reactor.clear_waker(this.listener_token);
                this.reactor.deregister(&mut this.listener);
                Poll::Ready(stream)
            }
            Err(e) if e.kind() == std::io::ErrorKind::WouldBlock => Poll::Pending,
            Err(e) => panic!("accept 失敗：{}", e),
```

```
        }
    }
}

struct Read<'a> {
    reactor: Arc<Reactor>,
    stream: &'a mut TcpStream,
    buf: &'a mut [u8],
    stream_token: Token,
}

impl<'a> Future for Read<'a> {
    type Output = usize;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<usize> {
        let this = self.get_mut();
        this.reactor.set_waker(this.stream_token, cx.waker().clone()); // 先登記
        match this.stream.read(this.buf) { // 再試一次
            Ok(n) => {
                // 清掉 Waker
                this.reactor.clear_waker(this.stream_token);
                Poll::Ready(n)
            }
            Err(e) if e.kind() == std::io::ErrorKind::WouldBlock => Poll::Pending,
            Err(e) => panic!("read 失敗：{}", e),
        }
    }
}

// 接一條連線，讀取並印出 request（簡化、不設逾時）
async fn serve(reactor: Arc<Reactor>, listener: TcpListener) {
    let mut stream = Accept::new(reactor.clone(), listener).await;

    let stream_token = reactor.unique_token();
    reactor.register(&mut stream, stream_token, Interest::READABLE);

    for i in 1..=3 {
        let mut buf = vec![0u8; 1024];
        let n = Read {
            reactor: reactor.clone(),
            stream: &mut stream,
            buf: &mut buf,
            stream_token,
        }
        .await;
        if n == 0 {
            println!("連線關閉了");
            break;
        }
        println!("第 {} 個 request：{}", i, String::from_utf8_lossy(&buf[..n]).trim());
    }

    reactor.deregister(&mut stream);
}
```

```rust
fn main() {
    let reactor = start_reactor();
    let addr = "127.0.0.1:8080".parse().expect("位址解析失敗");
    let listener = TcpListener::bind(addr).expect("綁定失敗");

    let mut executor = Executor::new();
    executor.block_on(serve(reactor, listener));
}
```

> 注意：這段程式會在本機監聽 127.0.0.1:8080，需要另外用 nc 之類的工具連進來（例如 nc 127.0.0.1 8080）才看得到效果。網頁版沙盒不適合體驗這種互動式網路程式；如果想體驗完整成果，請在自己的電腦上執行上面的程式碼。

12.14.2.3 Token 跟 I/O 來源綁在一起

Accept 和 Read 沒有共用同一個 Token。Accept 裡的 listener_token 是給 TcpListener 用的；接到連線後，serve 另外建立 stream_token，登記給那條 TcpStream。

後面的三次 Read 會共用同一個 stream_token，這是刻意的：Token 是 I/O 來源的名牌，不是每一次 .await 都要換一張名牌。這個簡化範例同一時間只會等待這條 stream 上的一次 read，所以同一個 stream Token 對應一個等待中的 Waker 就夠了。

等 I/O 成功後，Accept / Read 會呼叫 clear_waker，把這次等待用的 Waker 從 HashMap 裡拿掉。這樣 reactor 裡就不會留下「已經不需要喚醒」的等待者。

12.14.2.4 「先登記再試」為什麼重要

注意 Accept 和 Read 的 poll 都是**先** set_waker、**再**試一次 accept / read。這個順序是刻意的。

這個「再試一次」不代表這一輪一定會成功。如果還是 WouldBlock，這次 poll 就回 Pending；等 reactor 之後收到事件、呼叫剛剛存好的 Waker，executor 下一輪再 poll 這個 Future，才會再試一次 I/O。

想像如果反過來：先試 read 拿到 WouldBlock（還沒資料），然後正要去登記 Waker——就在這個空檔，資料剛好到了，reactor 醒來想 wake，卻發現 HashMap 裡還沒有這個 Token 的 Waker，這個喚醒就**漏掉**了，於是這個 Future 永遠不會再被 poll。

把順序倒過來——先把 Waker 放好，再試一次 I/O——就堵住了這個空檔：萬一資料早就到了，這次的 accept / read 會直接成功回 Ready；萬一真的還沒到，Waker 也已經就位，等 reactor 通知下一輪再試。成功就回 Ready，WouldBlock 就回 Pending。不過也因為我們是「先登記再試」，所以如果這次 accept / read 真的立刻成功，剛剛放進 HashMap 的 Waker 就已經用不到了。這時候 Accept / Read 會在回 Ready 前呼叫 clear_waker，把它清掉。換句話說，set_waker 是為了避免「還沒登記就錯過喚醒」，clear_waker 則是為了避免「已經完成了，卻留下不需要的等待者」。

12.14.2.5 喚醒路徑完全沒變

把這集和第 12 集對照，你會發現喚醒的終點一模一樣。reactor 雖然跑在自己的 Thread 上，但它呼叫的 waker.wake() 仍然是某個 Task 的 Waker——wake 一樣會把那個 Task 排回 ready queue、unpark executor。我們只是把「負責叫醒 Thread 的人」從計時 Thread 換成了 reactor Thread，後面的流程完全沒動。

到這裡，我們從零手寫的 runtime 作為教學專案算是大功告成了！它能 spawn、能睡覺、能被計時器或真實 I/O 喚醒。

這份簡化的實作在細節上仍有許多 bug，也沒有處理不少邊界情況，離能用於正式環境的 runtime 還很遠。不過，它已經足以讓我們看出 async runtime 背後大致在做什麼：安排 `Task`、等待事件，再喚醒 `Future`，讓 executor 重新 `poll`。

有了這幅整體圖像，接下來幾集我們要轉回頭，把 `async fn` 背後那個一直被我們提到卻還沒拆開的「狀態機」看個明白。

12.14.3 重點整理

- reactor 把喚醒接到真實 I/O：**executor 完全沿用第 12 集**，只把「誰來 `wake`」從計時 `Thread` 換成 reactor `Thread`。
- reactor 跑在自己的 `Thread`、睡在 `mio::Poll` 上，醒來照 `Token` 從 `HashMap` 取出 `Waker` 來 `wake`。
- `Future` 與 reactor 透過 `Arc` 共享的 `Registry`、`AtomicUsize`、`Mutex<HashMap<Token, Waker>>` 溝通，不傳訊息。
- `Token` 是 I/O 來源的名牌：listener 有自己的 `listener_token`，stream 有自己的 `stream_token`；在我們的程式碼中同一條 stream 的多次 `Read` 可以共用同一個 stream `Token`。
- `WouldBlock` 是非阻塞 I/O 的正常狀態，意思是「現在還不能做 `accept` / `read` 之類的 I/O 動作，晚點再試」，在 `Future` 裡對應 `Poll::Pending`。
- I/O `Future` 的 `poll` 一律「**先 `set_waker` 再試 I/O**」，避免漏接喚醒；如果立刻成功，回 `Ready` 前要 `clear_waker` 清掉不再需要的等待者。
- 不管喚醒來自計時器還是 I/O，最後都走「排回 ready queue ＋ `unpark`」同一條路。

12.15 `async fn` 背後的狀態機

12.15.1 本集目標

揭開 `async fn` 的真面目：它被編譯器改寫成一個能暫停、能恢復的**狀態機**。

12.15.2 正文

12.15.2.1 `.await` 不是開新 `Thread`

先破除一個可能的誤會。當你看到 `.await`，可能會以為它「在背後偷偷開了一條 `Thread` 去等」。**完全不是**。從第 6 集到現在，我們手寫這套 runtime 的 executor 從頭到尾就是一條 `Thread` 在反覆 `poll`，`.await` 沒有變出任何新 `Thread`。

那 `.await` 到底做了什麼？它把你的函數**切成好幾段**——每個 `.await` 是一個切點。函數可以在切點暫停、把控制權交還給 executor，之後再從同一個切點恢復。

編譯器做到這件事的方法，是把整個 `async fn` 改寫成一個**狀態機**：用「目前在哪個狀態」記住進度，下次被 `poll` 時，再依照那個狀態接著往下跑。

12.15.2.2 一個 `async fn` 會被改寫成什麼

假設有這麼一個 `async fn`，裡面等兩次：

```
async fn two_delays() {
    Delay::new(Duration::from_secs(1)).await;
    println!("一秒到");
```

```
    Delay::new(Duration::from_secs(1)).await;
    println!("兩秒到");
}
```

編譯器看到它之後，會把它改寫成一個 enum——每個「狀態」代表「目前卡在哪一段」：

- Start：還沒開始。
- FirstDelay：正在等第一個 Delay（這個還沒完成的 Delay 本身也得存進來）。
- SecondDelay：正在等第二個 Delay。
- Done：跑完了。

然後它替這個 enum 實作 Future，poll 裡用 match 看現在在哪個狀態、該做什麼。我們把這個改寫**手動**寫出來，你就會看到 async fn 背後長什麼樣：

```
use std::future::Future;
use std::pin::Pin;
use std::sync::Arc;
use std::task::{Context, Poll, Wake, Waker};
use std::thread::{self, Thread};
use std::time::{Duration, Instant};

struct ThreadWaker {
    thread: Thread,
}

impl Wake for ThreadWaker {
    fn wake(self: Arc<Self>) {
        self.thread.unpark();
    }
}

struct Delay {
    when: Instant,
    started: bool,
}

impl Delay {
    fn new(duration: Duration) -> Delay {
        Delay {
            when: Instant::now() + duration,
            started: false,
        }
    }
}

impl Future for Delay {
    type Output = ();

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        let this = self.get_mut();
        if Instant::now() >= this.when {
            Poll::Ready(())
        } else {
            if !this.started {
                this.started = true;
```

```
                let waker = cx.waker().clone();
                let when = this.when;
                thread::spawn(move || {
                    let now = Instant::now();
                    if now < when {
                        thread::sleep(when - now);
                    }
                    waker.wake();
                });
            }
            Poll::Pending
        }
    }
}

fn block_on<F: Future>(future: F) -> F::Output {
    let mut future = Box::pin(future);
    let waker = Waker::from(Arc::new(ThreadWaker {
        thread: thread::current(),
    }));
    let mut cx = Context::from_waker(&waker);
    loop {
        match future.as_mut().poll(&mut cx) {
            Poll::Ready(value) => return value,
            Poll::Pending => thread::park(),
        }
    }
}

// 這就是 two_delays 那個 async fn 背後大概的樣子
enum TwoDelays {
    Start,
    FirstDelay(Delay), // 正在等第一個 Delay，把它存著
    SecondDelay(Delay), // 正在等第二個 Delay
    Done,
}

impl Future for TwoDelays {
    type Output = ();

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        let this = self.get_mut();
        loop {
            match this {
                TwoDelays::Start => {
                    // 進入第一段：建立第一個 Delay，切換狀態
                    *this = TwoDelays::FirstDelay(Delay::new(Duration::from_secs(1)));
                }
                TwoDelays::FirstDelay(delay) => match Pin::new(delay).poll(cx) {
                    Poll::Ready(()) => {
                        println!("一秒到");
                        *this = TwoDelays::SecondDelay(Delay::new(Duration::from_secs(1)));
                    }
                    Poll::Pending => return Poll::Pending, // 卡在這一段，暫停
                },
```

```
                TwoDelays::SecondDelay(delay) => match Pin::new(delay).poll(cx) {
                    Poll::Ready(()) => {
                        println!("兩秒到");
                        *this = TwoDelays::Done;
                        return Poll::Ready(());
                    }
                    Poll::Pending => return Poll::Pending,
                },
                TwoDelays::Done => panic!("不該在 Ready 之後再 poll"),
            }
        }
    }
}

fn main() {
    println!("開始");
    block_on(TwoDelays::Start); // 等同於 block_on(two_delays())
}
```

上面一併附上前面寫過的陽春 Delay 和 block_on，只是為了讓這個手寫狀態機真的跑得起來。這段範例的主角不是 executor，而是 TwoDelays：它示範了 async fn 可能被改寫成什麼樣的狀態機。

12.15.2.3 對照著看

把這個手寫狀態機和原本的 async fn 對照：

- 原本 async fn 裡的**進度**，變成 enum 的**一個 variant**。
- 原本跨 .await 還要用到的**區域變數**（這裡是還沒完成的 Delay），被存進 variant 裡帶著走。
- 每個 .await，變成「poll 子 Future：Ready 就切到下一個狀態繼續，Pending 就 return Poll::Pending 暫停」。
- 下次被 poll，match 直接跳到上次停下的狀態，從那裡接著跑——這就是「從原地恢復」。

這正解釋了前面幾集看到的現象：為什麼 Future 每次被 poll 都能記得自己跑到哪、為什麼暫停後能從同一個地方繼續。因為它根本就是一個記著「目前在哪個狀態」的狀態機。

你平常寫 async fn 時，這一切都是編譯器自動幫你產生的，你完全不必手寫這種 enum。但理解它的真面目之後，後面幾集要談的 Pin 才會有意義——因為這個自動產生的狀態機，藏著一個跟「搬動記憶體」有關的危險。下一集就來看那樣的危險。

12.15.3 重點整理

- .await **不會**開新 Thread，它把函數切成可暫停、可恢復的好幾段。
- 編譯器把 async fn / async block 改寫成一個**狀態機**（概念上是個 enum）：進度變成 variant，跨 .await 的區域變數存進 variant。
- poll 用 match 看目前狀態：子 Future Ready 就切到下一狀態，Pending 就回 Pending 暫停。
- 下次 poll 直接跳回上次的狀態，從原地恢復——這就是 Future 能「記住進度」的原因。
- 這個改寫平常由編譯器自動完成，但理解它是搞懂後面 Pin 的前提。

12.16 自我參照的 Future

12.16.1 本集目標

理解 async 狀態機為什麼可能變成「自己指向自己」的結構，以及為什麼這種結構被 move 會出事。

12.16.2 正文

12.16.2.1 move 一個值，它的位址會變

先看一段**完全沒有** async 的普通程式。我們用 {:p}（印位址的格式）看一個值被 move 前後的位址：

```
fn main() {
    let p1 = String::from("hello");
    println!("p1 的位址：{:p}", &p1);

    let p2 = p1; // move：把 p1 搬到 p2
    println!("p2 的位址：{:p}", &p2);
}
```

兩個位址不一樣。這很合理——p1 和 p2 是兩個不同的區域變數，住在 stack 上不同的地方，move 就是把值從一個地方搬到另一個地方。

對一般的值來說，這完全沒問題：move 之後舊變數 p1 就不能再用了（這是第 4 章的所有權規則），所以「舊位址作廢」根本不影響任何人。

12.16.2.2 但如果值裡存了「指向自己的位址」呢

問題出在一種特殊的值：**它的某個欄位，存著一個指向自己另一個欄位的位址**。

想像這種值被 move 到新位置。它內部那個存起來的位址**不會自動更新**——它還指著**舊**位置。可是舊位置的東西已經搬走了，於是這個指標就變成了**懸垂指標**（指向一塊不再有效的記憶體）。一旦有人順著它去讀，就是未定義行為，程式可能讀到垃圾，也可能直接爆炸。

那這種「自己指向自己」的值，平常會遇到嗎？會——**自我參照的 Future 狀態機正是這種值**。回想上一集：async fn 被改寫成狀態機，跨 .await 還要用到的區域變數會被存進狀態機裡。如果其中一個區域變數是「另一個區域變數的參考」，那狀態機裡就會有一個欄位指向自己的另一個欄位——標準的自我參照結構。

```
async fn borrows() {
    let s = String::from("hello");
    let r = &s; // r 借用 s
    other().await; // 跨過一個 .await，s 和 r 都得被狀態機保存
    println!("{}", r); // .await 之後還用 r
}
```

這個 async fn 的狀態機，在 .await 那個狀態裡同時存著 s 和 r，而 r 指向 s。這就是自我參照。一旦它在這個狀態下被 move，r 就會變成懸垂指標。所以結論是：**Future 被 poll 到可能自我參照的狀態後，再 move 就有風險**。

12.16.2.3 先證明「create → poll → move → poll」做得出來

不過在談怎麼防範之前，先確認一件事：一個 Future 真的可能在「被 poll 過之後又被 move，然後再被 poll」。我們寫一個最小的 Future——Counter，每次 poll 就把計數 +1，並用 {:p} 印出 self 的位址：

```rust
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll, Waker};

struct Counter {
    count: u32,
}

impl Future for Counter {
    type Output = ();

    fn poll(self: Pin<&mut Self>, _cx: &mut Context<'_>) -> Poll<()> {
        let this = self.get_mut();
        this.count += 1;
        println!("第 {} 次 poll，self 在位址 {:p}", this.count, this);
        Poll::Pending
    }
}

fn main() {
    let mut cx = Context::from_waker(Waker::noop());

    let mut counter = Counter { count: 0 };
    let _ = Pin::new(&mut counter).poll(&mut cx); // poll 一次

    let mut moved = counter; // move 到新位置
    let _ = Pin::new(&mut moved).poll(&mut cx); // 再 poll
}
```

跑起來會看到兩次 poll 印出的位址**不一樣**——證明這套「poll → move → 再 poll」的流程真的能發生，而且第二次 poll 時 Future 已經在新位址了。Counter 自己沒有自我參照，所以搬了也沒差；但如果換成上面那種自我參照的狀態機，這一搬就出事了。

12.16.2.4 Rust 的防線：搬了會壞的，連門都不給進

那 Rust 怎麼防止自我參照的 Future 被亂搬？我們把同一套流程，套到剛剛那個「跨 .await 借用」的 async fn 上試試：

```rust
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Waker};

async fn other() {}

async fn borrows() {
    let s = String::from("hello");
    let r = &s;
    other().await;
    println!("{}", r);
}

fn main() {
    let mut cx = Context::from_waker(Waker::noop());
    let mut fut = borrows();
```

```
    // 想做跟 Counter 一樣的事：poll 一次
    let _ = Pin::new(&mut fut).poll(&mut cx); // 編譯錯誤！

    // 然後 move 到新位置
    let mut moved = fut;

    // 再 poll 一次
    let _ = Pin::new(&mut moved).poll(&mut cx); // 這裡也不會被放行
}
```

編譯器直接擋下來：

```
error[E0277]: `{async fn body of borrows()}` cannot be unpinned
```

這段程式碼把「poll 一次、move、再 poll」的動作都寫出來了，但編譯器其實在第一次 `Pin::new(&mut fut)` 就擋下來。

`Pin::new` 要求型別是 `Unpin`（「搬了不會壞」的意思，後面詳談）。`Counter` 是 `Unpin`，所以放行；但這個自我參照的 `async fn` 狀態機**不是** `Unpin`，於是 `Pin::new` 在你**還沒真的 poll、也還沒真的搬它之前**就把路擋死。

對照兩個例子，Rust 的防線就很清楚了：搬了不會壞的（像 `Counter`），給你方便、隨你搬；搬了會壞的（自我參照狀態機），連 `Pin::new` 這道門都不讓你進。至於 `Pin` 是怎麼用型別系統築起這道防線的，就是接下來的主題。

12.16.3 重點整理

- move 一個值，它的位址會變；對一般值無所謂，因為舊變數不能再用。
- 若值裡存了「指向自己的位址」，一 move 那個位址沒人更新，就變成懸垂指標——很危險。
- `async fn` / async block 產生的狀態機可能變成這種值：如果某個借用跨過 `.await`，狀態機就可能同時保存被借用的值和那個參考，形成某個欄位指向自己另一個欄位的結構。
- `Counter` 範例證明「poll → move → 再 poll」真的做得出來（兩次位址不同）。
- Rust 用 `Unpin` 當防線：`Counter` 是 `Unpin` 可被 `Pin::new`，自我參照的 `async` 狀態機不是 `Unpin`，`Pin::new` 直接編譯失敗。

12.17 為什麼 poll 需要 Pin

12.17.1 本集目標

弄懂 `poll` 為什麼把 `self` 寫成 `Pin<&mut Self>`，以及 `Pin` 究竟是用什麼手段把「不准 move」這件事變成事實。

12.17.2 正文

12.17.2.1 poll 想要的事

回顧：處於自我參照狀態的 `Future` 一旦被 move，內部那個指向自己的指標就會懸空。

還記得上一集對 `Counter` 做的那一串動作嗎？

```
let _ = Pin::new(&mut counter).poll(&mut cx); // poll 一次
let mut moved = counter; // 用 let 把它整個搬到新位置
let _ = Pin::new(&mut moved).poll(&mut cx); // 再 poll 一次
```

兩次 poll 印出的位址不一樣——這證明了一個 Future 真的可能「poll 過、被搬走、再 poll」。Counter 沒有自我參照，搬了無所謂；但同一套動作換成自我參照的 Future，第二次 poll 時它已經躺在新位址，內部那根指向自己的指標就懸空了。

所以負責推進 Future 的人（executor）得守住一條規矩：**在一連串 poll 之間，不可以把這個 Future 搬走**。問題是，poll 該收什麼樣的 self，才幫得上這條規矩？站在 poll 的角度，它同時想要兩個性質：

1. **要能動手腳**：每次 poll 都得改 Future 內部的欄位（推進狀態機、把進度往前帶），所以它需要某種「可變」的存取權。
2. **但不准搬家**：它不能讓人趁機把整個 Future 從原本的位址挪走，不然自我參照就毀了。

現成的工具——普通的 &mut Self——只滿足第一個性質。一旦 poll 收的是 &mut Self，那這個 Future 對 executor 來說就只是它手裡一個普通的值，executor 大可在兩次 poll 之間像上面那樣 let moved = ... 把它搬走，沒有東西攔得住。

所以 Rust 需要一種「能動手腳，但擋住搬家」的 &mut。這就是 Pin<&mut T>：你可以把它讀成「**一個被綁在原地、不准搬走的 &mut T**」。

不過要注意，「不准搬走」不是說這個值從建立開始就不能 move。一個值在被釘住以前，完全可以照一般 Rust 規則搬來搬去；Pin 保證的是：當你把需要釘住的值釘住之後，編譯器就不能再讓它被搬走。反正在開始 poll 之前，async 狀態機裡也還不會有自我指涉的參考；真正危險的是 poll 之後狀態機可能建立了自我參照，卻又被搬到別的位置。

12.17.2.2 關鍵：不能留下能搬走內部值的做法

Pin<&mut T> 號稱「不准搬走」，但它**憑什麼**做得到？

真正的關鍵有兩個：

1. 建立 Pin 的時候，不能留下另一條路讓你之後把值搬走。
2. 拿到 Pin 之後，安全 API 不能把能搬走內部值的普通指標交回給你。

先看第一點。

Pin 要守住「不准 move」的承諾，不能只看 Pin 本身裡面有沒有漏洞，還要看建立 Pin 之後，外面是不是仍然留著另一條路能把值搬走。

這就是 Pin::new(&mut value) 可疑的地方。Pin<&mut T> 只是暫時借用：借用結束後，外面原本那個變數還在，仍然可能被搬走。如果 Pin::new(&mut value) 對任何 T 都成立，那前面 Counter 那種「先 poll、再 move、再 poll」的流程，就可以原封不動套到一個自我參照的 Future 上。

所以對「搬了會壞」的型別來說，Pin::new(&mut value) 理應不該隨便成立。沒錯，原則上就是這樣；只是有些型別「搬了也不會壞」，所以 Rust 願意讓它們走這條路。這件事下一集會解釋。

再看第二點。

對 Pin<&mut T> 來說，危險的是普通的 &mut T。因為只要你拿到 &mut T，就能做 Option::take 之類的操作：

```
let old = option.take();
```

它不只把 Some(value) 變成 None，還會把裡面的 value 搬出來，回傳成 Some(value)。所以如果你有一個被 pin 住的 Option<Future>，又讓人從中拿到普通的 &mut Option<Future>，對方就可以 .take()，把那個 Future 從原本位址搬走。

所以對於一個未知的任意 T，Pin<&mut T> 不會給你普通的 &mut T。更一般地說，Pin<P> 會小心保護 P 這層指標。P 可能是 &mut T、Box<T>，或其他智慧指標。如果 Pin<Box<T>> 隨便把裡面的 Box<T> 還給你，你就又拿到能操作 T 的普通擁有者了，接著就可能把 T move 出來。所以對於任意 T，Pin 的安全 API 不把指向 T 的指標直接交回給你，而是只提供幾個不會破壞 pin 保證的操作。

12.17.2.3 Pin 只有幾招能用

也因為它的任務就是「擋 move」，Pin 能讓你做的事不多。一般的用法就是：

唯讀——Pin<P<T>> 永遠能解參考成 T（讀又搬不走值，沒風險），這來自 Deref：

```
impl<Ptr: Deref> Deref for Pin<Ptr> {
    type Target = Ptr::Target;
    fn deref(&self) -> &Ptr::Target { /* ... */ }
}
```

重新借出一根釘住的參考——as_mut 把 &mut Pin<Box<T>> 之類的東西借成 Pin<&mut T>。as_mut 可以一次又一次地被呼叫，因為 as_mut 只是借而已，而且借出來的仍然是 Pin<&mut T>，不是普通的 &mut T。

當然，拿著一個 Pin<&mut F>，你還能做最關鍵的一件事——呼叫它的 poll。對於 async fn 或 async block 產生的 Future，你不需要自己實作 poll；編譯器會替你產生實作。第 6 集 executor 反覆跑的 future.as_mut().poll(...) 就是這樣：as_mut 重新借出一個 Pin<&mut F>，交給 F 自己的 poll——而當這個 F 來自 async fn / async block 時，跑的正是編譯器產生的那個 poll。

12.17.2.4 Pin 釘的是「值」，不是「指標」

接著澄清一個很容易誤會的點：

> Pin<P> 釘住的，是 **P 指向的那個值**——而不是「Pin<P> 這個指標變數自己」。

所以 Pin<Box<T>> 這個東西**本身**是可以到處 move 的。你把它從一個變數搬到另一個、塞進 Vec、再拿出來，都沒問題——因為你搬的只是那根指標，被它指著的值始終待在 heap 上的原位。下面用 {:p} 印出「被指的值」的位址（&* 為 Pin<P<T>> 取得 &T），讓事實說話：

```
use std::pin::Pin;

struct Data {
    value: i32,
}

fn main() {
    let mut queue: Vec<Pin<Box<Data>>> = Vec::new();

    let boxed = Box::pin(Data { value: 7 });
    println!("放進 queue 前，值在 {:p}", &*boxed);

    queue.push(boxed); // Pin<Box<Data>> 這根指標被搬進 Vec
    let popped = queue.pop().unwrap(); // 又被搬出來

    println!("從 queue 拿出後，值在 {:p}", &*popped); // 位址一模一樣
}
```

兩次印出的位址完全相同：指標在 Vec 裡進進出出，但 heap 上那個 Data 從頭到尾沒有被 move。Pin 唯一禁止的，是「**透過它，把被指的值從原位址搬走**」這個動作而已。

12.17.2.5 Pin 通常藏在幕後

最後給你一顆定心丸：使用現成 runtime 撰寫 `async fn` + `.await` 時，`Pin` 通常都藏在幕後。編譯器會產生 `Future` 狀態機，runtime 則負責把它釘住並進行 `poll`。這一章會直接碰到 `Pin`，是因為我們正在親手探索這套底層機制。所以看不太懂這幾集的細節也別焦慮；這些內容是要讓你知道底下發生了什麼，而不是因為日常開發會把它們全部手寫一遍。

真有一天你要手刻底層 `Future`，需要從外層的 `Pin<P<外層>>` 取出某個欄位的 `Pin<P<內層>>`（這動作叫 projection），社群的 `pin-project` crate 可以替你安全地做掉，不必自己寫 `unsafe`。知道有這工具就夠了，這裡先不深入。

而如果你想「把 `Pin` 裡的值拿回成一個普通的 `&mut T`」，下一集也會講「搬了反正不會壞」時常常能用的辦法。

12.17.3 重點整理

- `poll` 想要「能改內部、但不准搬走」兩件事；普通 `&mut Self` 擋不住 move（executor 仍能在兩次 `poll` 之間 `let moved = ...` 把它搬走），所以不能用。
- `Pin<&mut T>` 是「綁在原地、不准搬走的 `&mut`」；`poll` 因此收 `Pin<&mut Self>`。
- 值在被釘住以前仍然可以照一般 Rust 規則 move；`Pin` 管的是「釘住之後」不能再把值搬走。
- `Pin` 擋 move 要守兩件事：建立時不能留下另一條外部路徑讓你之後搬走值；使用時安全 API 也不能把能搬走內部值的內層指標交給你。
- `Pin` 的用法很有限：唯讀靠 `Deref`，重新借出用 `as_mut`，當然還能餵給 `poll`。
- `Pin<P>` 釘的是「被指的值」不是「指標本身」，所以 `Pin<Box<T>>` 自己能隨意 move（連塞進 `Vec` 再拿出來都行），這就是 executor 能到處搬 `Pin<Box<Fut>>` 的原因。
- 使用現成 runtime 撰寫 `async fn` + `.await` 時，`Pin` 通常都藏在幕後：編譯器產生 `Future` 狀態機，runtime 負責把它釘住並進行 `poll`。這一章會直接碰到 `Pin`，是因為我們正在親手探索這套底層機制。

12.18 Unpin

12.18.1 本集目標

認識 `Unpin` 這個「搬了不會壞」的標籤，看懂為什麼有了它，被釘住的 `&mut` 就能變回普通的 `&mut`。

12.18.2 正文

12.18.2.1 先問一個問題：「不准搬」對誰有意義？

上一集 `Pin` 千方百計擋住釘住之後的 move。但先退一步想：這條規矩到底在防誰？

答案是——除了像是**自我參照的 `Future` 狀態機**這種少見的特例，你平常用的型別（`i32`、`String`、`Vec`、自己定義的 `struct`……）搬來搬去根本不會出事，move 對它們就是把幾個 bytes 換個位置存而已。對這些型別硬套上「不准搬」，純粹是多管閒事。

Rust 用一個標籤把這兩群分開，這個標籤就叫 **`Unpin`**：一個型別是 `Unpin`，意思就是「**我搬了不會壞，`Pin` 不必管我**」。

12.18.2.2 `Unpin` 幾乎人人都有

`Unpin` 和多執行緒一章介紹過的 `Send` / `Sync` 一樣是 **`auto trait`**——如果一個型別裡存的資料都是 `Unpin`，那它本身預設就是 `Unpin`。我們直接跳到結果：**絕大多數型別都是 `Unpin`**。

寫個小工具就能驗證。下面的 assert_unpin 只收 Unpin 的型別，把各種常見值丟進去都能過：

```
fn assert_unpin<T: Unpin>(_: T) {}

fn main() {
    assert_unpin(42);
    assert_unpin(String::from("hi"));
    assert_unpin(vec![1, 2, 3]);
    println!("這些都是 Unpin");
}
```

連我們前面手寫的 Delay、Counter、JoinAll、JoinHandle 也都是 Unpin——它們的欄位都是普通可搬的東西。

那誰不是 Unpin？把同一個工具拿去檢查一個「跨 .await 借了區域變數」的 async fn（也就是第 16 集那種自我參照狀態機），就會被擋下來：

```
fn assert_unpin<T: Unpin>(_: T) {}

async fn other() {}

async fn demo() {
    let s = String::from("hi");
    let r = &s;
    other().await; // s 和 r 都跨過了 .await
    println!("{}", r);
}

fn main() {
    assert_unpin(demo()); // 編譯錯誤：demo() 的 Future 不是 Unpin
}
```

編譯器會說 ... cannot be unpinned。這就對了：async fn / async block 的 Future **不能假設是 Unpin**，因為它有可能正是那個搬了會壞的自我參照狀態機。

12.18.2.3 Unpin 的型別，可以把值要回來

知道誰是 Unpin 之後，上一集留的伏筆就能解開了：「把釘住的值變回成普通 &mut T」這件事，對 Unpin 的型別是開放的。

道理很直接：既然這個型別搬了不會壞，Pin 的保護對它本來就是多餘的，那乾脆讓你把普通 &mut T 拿回去。具體來說，當 T: Unpin，Pin<&mut T> 上才有 get_mut 把它變回 &mut T，而 Pin<P<T>> 才會實作 DerefMut：

```
use std::pin::Pin;

fn main() {
    let mut n = 10;
    let mut pinned: Pin<&mut i32> = Pin::new(&mut n);

    // i32 是 Unpin，所以 Pin<&mut i32> 實作 DerefMut
    *pinned = 100;
    println!("{}", pinned);

    // 也可以用 get_mut 拿回普通的 &mut i32
    let back: &mut i32 = pinned.get_mut();
```

```
    *back += 5;
    println!("{}", back);
}
```

因此，我們前面每個自訂 Future 的 poll 開頭都寫 let this = self.get_mut();，也沒有出問題。那些型別全是 Unpin，當然能用 get_mut。要是哪天你的 Future 不是 Unpin，這行就會編譯失敗，逼你改用 Pin 的方法小心處理。

12.18.2.4 兩個動作，都要求 Unpin

上一集說過，Pin 的保證有兩個：

1. 建立 Pin 的時候，不能留下另一條路讓你之後把值搬走。
2. 拿到 Pin 之後，安全 API 不能把能搬走內部值的普通指標交回給你。

現在把 Pin::new 和 get_mut 擺在一起看，就會發現它們分別放寬了上面兩個限制，而且都要求**同一個條件**——Unpin：

```
// 方法一：用既有指標建立 Pin
// 如果被指的值不是 Unpin，不能這樣做
impl<P: Deref> Pin<P> {
    pub fn new(pointer: P) -> Pin<P> where P::Target: Unpin { /* ... */ }
}

// 方法二：把釘住的值變回普通 &mut
// 如果 T 不是 Unpin，也不能這樣做
impl<T: ?Sized> Pin<&mut T> {
    pub fn get_mut(self) -> &mut T where T: Unpin { /* ... */ }
}
```

Pin::new 放寬的是第一個限制。它允許你拿一個既有的指標，例如 &mut T，直接包成 Pin<&mut T>。這件事對「搬了會壞」的型別很危險，因為 Pin<&mut T> 只是暫時借用；借用結束後，外面的原變數 T 還可能被搬走。所以 Pin::new 只允許用在 T: Unpin 的型別上。

get_mut 放寬的是第二個限制。它會把 Pin<&mut T> 變回普通 &mut T。這也只對 T: Unpin 安全，因為普通 &mut T 可以做 Option::take 這類把值搬出原位址的事。DerefMut 同理。

Unpin 就是在說：「這個型別搬了也不會壞，所以能安全進行這些動作。」第 16 集用到的 Counter 是 Unpin，使用 Pin::new(&mut counter)、get_mut、DerefMut 都暢行無阻；自我參照的 async 狀態機不是 Unpin，這兩個動作都不給它用。

所以實務上的判斷很簡單：你手上的 Future 是 Unpin 嗎？是的話，Pin::new、get_mut、DerefMut 隨你用；不是的話（一般就是 async fn/ async block 生出來的 Future），你就得用能維持 Pin 保證的方式把它釘起來——例如 Box::pin 放在 heap 上，或用下一集要登場的 pin! 釘在 stack 上。

12.18.3 重點整理

- 除了像自我參照的 async 狀態機這種少見特例，平常用的大多數型別搬了都不會壞。
- Unpin 就是「搬了不會壞」的標籤，是 auto trait，由編譯器自動實作，**絕大多數型別都是 Unpin**。
- async fn / async block 的 Future 不能假設是 Unpin（可能是自我參照的狀態機）。
- 當 T: Unpin，Pin<&mut T> 才能用 get_mut 變回普通 &mut T，Pin<P<T>> 才實作 DerefMut；

我們手寫 Future 的 self.get_mut() 能用就是這個原因。

- Pin::new 和 get_mut / DerefMut 分別放寬上一集的兩個限制：用既有指標建立 Pin、拿回普通 &mut；它們都只對 Unpin 開放。
- Future 不是 Unpin 時，就得用 Box::pin 或下一集的 pin! 來釘。

12.19 pin!

12.19.1 本集目標

學會用 pin! 在 stack 上釘住一個 Future，並理解它為什麼非得是巨集不可。

12.19.2 正文

12.19.2.1 stack pinning

前面要釘住一個 Future，我們都用 Box::pin——把它放到 heap 上。但有時候你不想為了釘住一個 Future 而特地做一次 heap 配置（因為這麼做是有成本的），尤其當這個 Future 只在目前這個作用域裡用、不需要傳出去的時候。

這種情況可以用 std::pin::pin!。它會把一個值釘在目前作用域裡，給你一個 Pin<&mut T>：

```
use std::future::Future;
use std::pin::pin;
use std::task::{Context, Poll, Waker};

async fn hello() -> i32 {
    42
}

fn main() {
    // 在 stack 上釘住這個 future，拿到 Pin<&mut _>
    let mut future = pin!(hello());

    let mut cx = Context::from_waker(Waker::noop());
    match future.as_mut().poll(&mut cx) {
        Poll::Ready(value) => println!("完成：{}", value),
        Poll::Pending => println!("還沒好"),
    }
}
```

這裡有個容易誤會的點：目前編譯器保守地把 hello() 這種 async fn 產生的 Future 與 async block 一律當成**不是** Unpin——連這個沒有任何 .await、根本不可能自我參照的 hello 也一樣（編譯器不想逐一判斷，乾脆全部不實作 Unpin）。所以上一集的 Pin::new 對它行不通，但 pin! 可以：因為 pin! 做的是 **stack pinning**，而它釘的辦法不要求 Unpin。

12.19.2.2 為什麼 pin! 一定要是巨集

這是一個很有意思的問題。pin! 為什麼是巨集，不是一個普通函數？

先抓住一個關鍵：pin! 交給你的 Pin<&mut T> 是一個**參考**，而參考一定要指著一個還活著的值。只要你還握著這個 Pin<&mut T>，被它借的那個值就不能消失。

所以問題不是「怎麼做出一個 Pin<&mut T>」而已；問題是這個 Pin<&mut T> 借用的值要活得夠久。

如果把它寫成普通函數，會是這樣：

```
fn pin<T>(value: T) -> Pin<&mut T> { /* ??? */ }
```

這行不通。value 是 pin 這個函數自己的區域變數。函數一返回，它的區域變數就被清掉，value 跟著消失——於是回傳的那個 Pin<&mut T> 立刻變成懸垂參考，指向一塊已經作廢的記憶體。事實上編譯器根本不會讓你回傳一個指向「函數自己區域變數」的參考。

pin! 不是普通函數，所以不會有「回傳指向自己 stack 上變數的參考」這個問題。pin! 產生的 Pin<&mut T> 借用的是使用 pin! 的那個作用域裡的值，而不是借用某個普通函數自己的暫時變數。因此，那個借用不會在函數返回時立刻懸垂，你可以安心拿來用。

12.19.2.3 對照 Box::pin

那為什麼 Box::pin 可以是普通函數？因為它走的是完全不同的路：Box::pin 把值放到 **heap** 上，並把那塊 heap 記憶體的所有權包進 Pin<Box<T>> 交給你。heap 上擁有所有權的東西活得比「這次函數呼叫」久，函數返回也不會丟掉它，所以回傳一個擁有它的 Pin<Box<T>> 完全沒問題。

一句話總結兩者的差別：

- pin!：**stack 借用**——在不配置 heap 的情況下，從目前區塊中的值取得 Pin<&mut T>；因為普通函數不能回傳指向自己 stack 上變數的參考，所以這件事必須由巨集在呼叫位置完成。
- Box::pin：**heap 擁有**——東西放 heap，交出所有權，可以到處傳，所以可以是普通函數。代價是一次 heap 配置。

12.19.3 重點整理

- pin! 做 **stack pinning**：取得一個只在目前區塊內有效的 Pin<&mut T>，不需 heap 配置，適合不必把被釘住的值傳出作用域的情況。
- pin! 必須是**巨集**而非函數：普通函數不能回傳指向自己 stack 上變數的 Pin<&mut T>；函數一返回，那個參考就會懸垂（編譯器也不准你這樣寫）。
- Box::pin 交出所有權把值放 heap（活得比這次呼叫久），所以能當普通函數；差別就在「stack 借用 vs heap 擁有」。

12.20 async 遞迴

12.20.1 本集目標

理解為什麼 async fn 不能直接呼叫自己，以及怎麼用 Box::pin 解決。

12.20.2 正文

12.20.2.1 直接遞迴會編譯失敗

來試著寫一個 async 版的階乘——一個 async fn 在裡面 .await 自己：

```
async fn factorial(n: u64) -> u64 {
    if n == 0 {
        1
    } else {
        n * factorial(n - 1).await // 編譯錯誤
    }
}
```

```
fn main() {}
```

編譯器會直接拒絕：

```
error[E0733]: recursion in an async fn requires boxing
```

12.20.2.2 為什麼？因為 Future 型別的大小無法決定

回想第 15 集：async fn 會被改寫成一個狀態機，而跨 .await 用到的東西都要存進這個狀態機。

這裡的關鍵不是遞迴執行時會不會停。n == 0 當然是 base case，真的跑起來時會停；但在程式開始跑之前，編譯器就要先決定 factorial 回傳的 Future 型別有多大。

粗略想像一下，它可能需要長得像這樣：

```
enum FactorialFuture {
    Start { n: u64 },
    Waiting {
        n: u64,
        child: FactorialFuture,
    },
    Done,
}
```

Waiting 狀態要保存正在 .await 的 factorial(n - 1)；而 factorial(n - 1) 回傳的又是同一種 FactorialFuture。於是這個型別直接包含自己，編譯器在算 child 欄位要占多少空間時，永遠算不出固定答案。

這個情境你其實見過。第 5 章講遞迴型別時就碰過一模一樣的問題：一個 struct 直接包含自己，大小會無限大。當時的解法是用 Box 把遞迴的部分放到 heap 上——Box<T> 不管 T 多大，它本身永遠只是一個指標大小。

12.20.2.3 解法：用 Box::pin 包住遞迴呼叫

async 遞迴的解法一樣：把遞迴呼叫產生的 Future 用 Box::pin 包起來。這樣狀態機裡存的就只是一個固定大小的指標，而不是直接存另一個同型別的狀態機：

```
use std::future::Future;
use std::task::{Context, Poll, Waker};

async fn factorial(n: u64) -> u64 {
    if n == 0 {
        1
    } else {
        n * Box::pin(factorial(n - 1)).await
    }
}

fn block_on<F: Future>(future: F) -> F::Output {
    let mut future = Box::pin(future);
    let mut cx = Context::from_waker(Waker::noop());
    loop {
        match future.as_mut().poll(&mut cx) {
            Poll::Ready(v) => return v,
            Poll::Pending => {}
        }
```

```
    }
}

fn main() {
    let result = block_on(factorial(5));
    println!("5! = {}", result);
}
```

上面一併附上前面寫的最陽春的 `block_on`（這個 `factorial` 其實沒有真的需要等的 `.await`，所以用這種版本就夠了）。

這裡的 `Box` 和 `Pin` 各自負責不同的事：`Box` 讓狀態機只需要保存固定大小的指標，`Pin` 則讓裡面的 `Future` 可以被安全地 `poll`。只寫 `Box::new(factorial(n - 1)).await` 不夠，因為 `async fn` 回傳的 `Future` 不保證實作 `Unpin`，而 `Box<F>` 只有在 `F: Unpin` 時才實作 `Future`。所以我們用 `Box::pin` 得到 `Pin<Box<F>>`；只要 `F: Future`，`Pin<Box<F>>` 本身也實作 `Future`，可以直接被 `.await`。

到這裡，我們把 async 底層的機制——`Future`、executor、reactor、狀態機、`Pin`——從頭到尾走過一遍了。下一集起，我們要回到 Tokio，看看一個真正成熟的 runtime 提供了哪些好用的工具方便使用者撰寫 async 程式碼。

12.20.3 重點整理

- `async fn` 直接呼叫自己會編譯失敗，因為編譯器無法決定狀態機型別的大小。
- base case 只能決定執行時期會不會停，不能決定編譯時期的型別大小；問題和第 5 章的遞迴型別一樣：自己包含自己，大小無限。
- 解法是把遞迴呼叫用 `Box::pin` 包起來，讓狀態機裡只存一個固定大小的指標。

12.21 回到 Tokio

12.21.1 本集目標

從手寫 runtime 回到 Tokio，重新看看 `tokio::spawn` 和 `JoinHandle`、比較 Tokio 與手寫 `block_on` 的差別，以及認識 Tokio runtime 的多執行緒 / 單執行緒模式。

12.21.2 正文

12.21.2.1 你已經懂底層了

恭喜你撐過了最硬的幾集！前面我們從零手寫了 executor、reactor、`Task`、`JoinHandle`，還拆開了狀態機和 `Pin`。Tokio 真正的實作當然複雜得多，但現在回頭看它的 API，很多名詞和設計取捨應該不會讓人陌生。

12.21.2.2 `tokio::spawn` 與 `JoinHandle`

`tokio::spawn` 就是我們手寫過的 `spawn`：把一個 `Future` 包成 `Task` 交給 runtime 排程，回傳一個 `JoinHandle`：

```
#[tokio::main]
async fn main() {
    let handle = tokio::spawn(async {
        21 * 2
```

```
    });
    let result = handle.await.expect("背景 task panic 了");
    println!("結果：{}", result);
}
```

（Tokio 的 JoinHandle .await 回傳的是 Result，因為背景 Task 有可能 panic，所以這裡用 expect。）

12.21.2.3 Tokio 的 block_on 和手寫版有何不同

和手寫版不同，tokio::runtime::Runtime::block_on **不需要** Send 或 'static。因為它只是把你給它的那個 Future，在目前這條呼叫的 Thread 上跑到完成，不會搬到別條 Thread，所以沒有 Send 的顧慮。

語意上還有一個重要差異：我們第 11 集後手寫的 block_on 會等 ready queue 裡所有 Task 都完成才回傳，但 Tokio 的 block_on 是「**傳給它的那個 Future 一完成就回傳**」，不會等其他用 tokio::spawn 開出去的背景 Task。那些還沒做完的背景 Task 會留在 runtime 上。

一句話對比：手寫版是「跑完整批才繼續」，Tokio 是「跑完我指定的這一個就繼續」。所以在 Tokio 裡，block_on 回傳時，只代表你傳進去的那個 Future 完成了；你 spawn 出去的背景 Task 可能還沒跑完。如果 runtime 接著結束，這些背景 Task 就沒有機會繼續做完。

12.21.2.4 新手最常見的編譯錯誤：.await 期間持有非 Send 的值

tokio::spawn 要求 Future: Send，而一個 Future 是不是 Send，取決於它**跨 .await 時保存了什麼**。如果在 .await 期間還持有一個像 Rc 這樣非 Send 的值，整個 Future 就不是 Send，於是不能 spawn：

```
use std::rc::Rc;

async fn some_async() {}

#[tokio::main]
async fn main() {
    tokio::spawn(async {
        let rc = Rc::new(5);
        some_async().await; // 跨 .await 還持有 rc，而 Rc 不是 Send
        println!("{}", rc);
    });
}
```

編譯器會說 future cannot be sent between threads safely，並指出 Rc<i32> 在一個 .await 上被跨越使用。

解法有幾種：

用 Send 的替代品。這裡把 Rc<i32> 換成 Arc<i32>，後者是 Send 的：

```
use std::sync::Arc;

async fn some_async() {}

#[tokio::main]
async fn main() {
    tokio::spawn(async {
```

```
        let arc = Arc::new(5);
        some_async().await;
        println!("{}", arc);
    });
}
```

在 .await 之前就把非 Send 的值處理掉。用 {} 縮小它的作用域，讓它在 .await 之前就被 drop，這樣狀態機跨 .await 時根本不持有它：

```
use std::rc::Rc;

async fn some_async() {}

#[tokio::main]
async fn main() {
    tokio::spawn(async {
        let n = {
            let rc = Rc::new(5);
            *rc
        }; // rc 在這個 block 結束就 drop 了，沒有跨 .await
        some_async().await;
        println!("{}", n);
    });
}
```

（明確呼叫 drop(rc) 把它在 .await 前丟掉，也是同樣的效果。）

12.21.2.5 #[tokio::main] 的 flavor

最後，雖然 #[tokio::main] 預設用多執行緒 runtime，但你可以改：

```
extern crate tokio;

// 單執行緒 runtime
#[tokio::main(flavor = "current_thread")]
async fn main() {
    println!("我跑在單一執行緒上");
}
```

或指定 worker Thread 的數量：

```
extern crate tokio;

// 多執行緒，指定 4 條 worker
#[tokio::main(flavor = "multi_thread", worker_threads = 4)]
async fn main() {
    println!("我有 4 條 worker threads");
}
```

單執行緒 runtime 的好處是沒有跨 Thread 的成本，缺點是不能真平行。

12.21.3 重點整理

- tokio::spawn 把 Future 交給 runtime，回傳 JoinHandle（.await 後得到 Result，因為 Task 可能 panic）。
- 和手寫版不同，Tokio 的 block_on 不需要 Send 或 'static，而且是**指定的 Future** 一完成就

回傳，不會等待**所有** Task。
- `.await` 期間持有像 `Rc` 這樣非 `Send` 的值，會讓 `Future` 不是 `Send`，不能 spawn；解法是改用 `Arc`、或用作用域 / `drop` 讓它在 `.await` 前消失。
- `#[tokio::main]` 預設多執行緒，但可用 `flavor = "current_thread"` 或 `worker_threads = N` 調整；無論哪種，`tokio::spawn` 的 `Future` 與輸出仍然必須是 `Send + 'static`。

12.22 spawn_blocking

12.22.1 本集目標

學會「不要 block 住執行緒」這條紀律，以及碰到非得 block 的工作時，用 `spawn_blocking` 把它安置好。

12.22.2 正文

12.22.2.1 一條鐵律：不要 block 住執行緒

async 能用少少幾條執行緒推進大量工作，靠的是大家**輪流**。`.await` 是一個**可能**讓出執行緒的位置：如果等待的 `Future` 還沒準備好、回傳 `Pending`，Task 就會讓出執行緒，換別人跑；如果 `Future` 已經準備好了，程式會立刻繼續，不會讓出執行緒。

這就帶出一條鐵律：**一個 Task 不能長時間不 `.await`**。如果某個 Task 卡著 Thread 不放——可能是在做一個很耗時的計算（幾秒的數學運算），也可能是呼叫了某個**同步**的阻塞函數（`std::thread::sleep`、同步讀檔、慢的同步資料庫呼叫）——那它就會霸佔這條 Thread。在阻塞工作完成前，**這條 Thread 無法 poll 任何其他 Task**。

舉個不好的示範：

```
#[tokio::main]
async fn main() {
    // 在 async task 裡做很重的同步計算——壞示範
    let sum: u64 = (0..2_000_000_000u64).sum(); // 這段期間完全沒有 .await
    println!("總和：{}", sum);
}
```

這段計算從頭到尾沒有 `.await`，所以執行它的 `Future` 在計算完成前都無法讓出控制權。如果這類程式碼在透過 `tokio::spawn` 產生的 Task 中執行，它會在這段期間佔住一條 worker Thread，使該 Thread 無法推進其他 Task；不過，其他 worker Thread 仍可繼續執行。

12.22.2.2 解法：`spawn_blocking`

碰到這種「非 block 不可」的工作，解法是 `tokio::task::spawn_blocking`。它把工作丟到一個**專用的 blocking 執行緒池**去做（那個池子裡的執行緒就是設計來被卡住的），回傳一個可以 `.await` 的 handle：

```
#[tokio::main]
async fn main() {
    let handle = tokio::task::spawn_blocking(|| {
        // 重計算丟到專用的 blocking 池裡做
        (0..2_000_000_000u64).sum::<u64>()
    });

    // 如果工作還沒完成，在這裡 .await 就會讓出執行緒，完成後再被喚醒
```

```
    let sum = handle.await.expect("blocking task 失敗");
    println!("總和：{}", sum);
}
```

關鍵在於：因為你用 .await 等那個 handle，**只要 handle 還沒準備好**，你自己的 Task 就會讓出執行緒，runtime 可以拿去推進別的 Task；等 blocking 池那邊算好了，再把你喚醒。慢的計算被隔離在專用池裡，不會拖累負責 async 工作的 Thread。

（順帶一提：想在 async 裡「睡一下」，別用 std::thread::sleep，那會 block 住執行緒；要用 tokio::time::sleep(...).await，它是 async 的，會乖乖讓出執行緒。）

12.22.2.3 為什麼不乾脆用 std::thread::spawn

你可能會問：要把工作丟到別條 Thread，多執行緒章不是有 std::thread::spawn 嗎？

問題出在「怎麼拿回結果」。std::thread::spawn 給你的 JoinHandle，要拿結果得呼叫 .join()——而 .join() 是**阻塞**的，它不是 async、不能 .await。如果你在 async 裡呼叫 .join()，就又把執行緒卡住了，繞回原來的問題。

spawn_blocking 的價值，是它把「同步工作在 blocking 池做完後，通知正在 .await 的 async Task 繼續跑」這件事包好了。你不用自己拿 std::thread::JoinHandle 去 .join()，也不用自己接 Waker；只要對它回傳的 handle .await，結果還沒準備好時，Task 就會讓出執行緒，等結果好了再被喚醒。

12.22.2.4 但長命的背景執行緒還是該用 thread::spawn

最後值得提的是：spawn_blocking 適合的是**會結束**的一次性工作。如果你要的是一條**長命的獨立背景執行緒**（例如一個跑著無窮迴圈，整個程式生命週期都在的監聽器），那還是該用 std::thread::spawn。

為什麼？因為 blocking 池的空間有限。把一個無窮迴圈丟進 spawn_blocking，它會**永久佔住**池子裡的一個名額再也不還，這屬於誤用——久了池子被佔滿，真正需要它的短工作就排不進去了。

12.22.3 重點整理

- 鐵律：執行緒只在 .await 時能被讓出，所以 Task 不能長時間不 .await，否則會霸佔執行緒、拖住同條執行緒上的其他 Task。
- 昂貴計算、同步阻塞呼叫（std::thread::sleep、同步 I/O、慢的同步 DB）都會 block 住執行緒。
- tokio::task::spawn_blocking 把這種工作丟到專用 blocking 池，回傳可 .await 的 handle；結果還沒準備好時，你的 Task 會讓出執行緒。
- 不用 std::thread::spawn 是因為它的 .join() 是阻塞的、不能 .await；spawn_blocking 幫你接好了「做完 → 喚醒 Task」這座橋。
- 但長命的獨立背景執行緒仍該用 std::thread::spawn；把無窮迴圈丟進 spawn_blocking 會永久佔住池子名額，是誤用。

12.23 join!

12.23.1 本集目標

學會用 join! 在同一個 Task 裡同時等待多個 Future，並理解它為什麼是巨集。

12.23.2 正文

12.23.2.1 在同一個 Task 裡並行

第 9 集我們手寫過 JoinAll，把多個 Future 一起推進。Tokio 提供現成的 join!，做的是同一件事：

```
extern crate tokio;

use tokio::time::{sleep, Duration};

async fn fetch_a() -> i32 {
    sleep(Duration::from_secs(1)).await;
    1
}

async fn fetch_b() -> &'static str {
    sleep(Duration::from_secs(1)).await;
    "hello"
}

#[tokio::main]
async fn main() {
    // 兩個 Future 同時等，總共約一秒，回傳一個 tuple
    let (a, b) = tokio::join!(fetch_a(), fetch_b());
    println!("a = {}, b = {}", a, b);
}
```

join! 會等**所有** branch 都完成才往下走，把每個 branch 的結果包成一個 tuple 還給你。上面兩個 fetch 各要等一秒，但因為是並行，總共只花約一秒，不是兩秒。

12.23.2.2 spawn 和 join! 的差別

spawn 和 join! 兩者都能並行，但作法不同：

- tokio::spawn 把每個工作變成**獨立的 Task** 交給 runtime，可能被分到不同的 Thread 上跑，所以要 Send + 'static。
- join! 是在**同一個 Task** 裡輪流 poll 那幾個 branch，它們**不會**變成獨立 Task。

正因為各個 branch 都保存在目前這個 Task 裡，並由 join! 等到全部完成，它們不會脫離目前作用域成為獨立的 Task。因此，join! 適合「**固定數量**、需要在目前作用域內一併完成」的並行 I/O——例如同時呼叫三個 API、同時讀兩個檔案。

12.23.2.3 join! 的並行不是 CPU 平行

這裡要澄清一個重要的限制。join! 的各個 branch 是在**同一個 Task 上輪流被 poll** 的，這代表它的並行是「交錯切換」那種，**不可能是** CPU 平行。

後果很實際：如果某個 branch 長時間不 .await（在裡面做比較耗時的計算，或呼叫同步阻塞函數），它就會霸佔住執行緒——而且因為大家在同一個 Task 上輪流，**連同一個 join! 裡其他 branch 都得不到 poll**。並行的假象當場破功。

這正是上一集「不要 block 住執行緒」那條鐵律在 join! 上的具體案例。如果某個 branch 真的要幹大事，記得用 spawn_blocking，別讓它卡在 join! 裡。

12.23.2.4 為什麼 join! 是巨集

你大概會注意到 join! 也是一個巨集，不是函數。這次又為什麼非得是巨集？

因為它要吃任意數量、各自不同型別的 Future，再回傳一個形狀剛好對應的 tuple。join!(a, b)、join!(a, b, c, d) 都行，而且每個 branch 的 Future 型別可以完全不一樣；回傳值也會跟著變成 (A::Output, B::Output) 或 (A::Output, B::Output, C::Output, D::Output)。

一般 Rust 函數的參數數量是固定的。我們可以分別為兩個、三個或四個 Future 撰寫泛型函數，讓每個函數回傳元素型別與這些 Future 輸出相符的 tuple；但單一函數無法涵蓋所有可能的參數數量。巨集則能在**編譯時期**根據每次呼叫，產生 tuple 形狀完全吻合的程式碼。

對照第 9 集的 JoinAll 就更清楚了：JoinAll 處理的是「**同回傳型別、動態數量**」——數量要到執行時期才定，至於裡面具體是哪一種 Future，我們用 dyn Future<Output = ()> 把它抹掉了，只要求輸出一律是 ()。join! 反過來，是「**異回傳型別、固定數量**」——數量和每個 Future 的輸出型別在你寫程式碼時就定死了，所以能用巨集在編譯時期攤開成一個剛好對應的 tuple。

12.23.3 重點整理

- join! 在**同一個 Task** 裡同時等多個 Future，等全部完成後把結果包成 tuple 回傳。
- 和 spawn 不同：join! 的 branch 不變成獨立 Task，適合固定數量、需要在目前作用域內一併完成的並行 I/O。
- join! 的並行不是 CPU 平行：branch 在同一個 Task 上輪流 poll，某個 branch 卡住會害其他 branch 都得不到 poll。
- join! 是巨集，因為每次呼叫都可以傳入不同數量、型別各異的 Future，並產生形狀對應的輸出 tuple；一般函數則必須為每種參數數量各寫一個版本。
- 對照我們自己寫的 JoinAll（同回傳型別、動態數量），join! 是異回傳型別、固定數量。

12.24 Semaphore 與 backpressure

12.24.1 本集目標

學會用 Semaphore 限制「同時進行的數量」，並理解 backpressure 這個觀念。

12.24.2 正文

12.24.2.1 限制同時進行的數量

有些事情你不希望「無限制地一起做」。例如：同時下載的檔案數別太多（不然頻寬爆掉）、同時開啟的檔案數有上限、同時對某個 API 的請求數要節制（不然對方會擋你）。

tokio::sync::Semaphore 就是管這個的。它的核心是一些固定數量的 **permit（許可證）**：你設定總共有幾張 permit，誰要做事就得先拿一張，做完還回去。permit 被拿光時，後來的人就得等，直到有人還回一張。

```
extern crate tokio;

use std::sync::Arc;
use tokio::sync::Semaphore;
use tokio::time::{sleep, Duration};

#[tokio::main]
```

```rust
async fn main() {
    // 總共只有 3 張 permit，所以最多 3 個 task 能同時工作
    let semaphore = Arc::new(Semaphore::new(3));

    let mut handles = vec![];
    for i in 0..10 {
        let semaphore = Arc::clone(&semaphore);
        handles.push(tokio::spawn(async move {
            // 拿一張 permit，拿不到就 .await 等著
            let _permit = semaphore.acquire().await.expect("semaphore 已關閉");
            println!("task {} 拿到 permit，開始工作", i);
            sleep(Duration::from_millis(100)).await;
            // _permit 在這裡離開作用域，自動把名額還回去
        }));
    }

    for h in handles {
        h.await.expect("task 失敗");
    }
}
```

雖然我們 spawn 了 10 個 Task，但因為只有 3 張 permit，任何時刻最多只有 3 個在工作，其餘的乖乖排隊等 permit。

12.24.2.2 permit 靠 Drop 自動歸還

注意上面我們拿到 permit 後，**完全沒有手動把它還回去**——它怎麼自己回來的？

因為 permit 實作了 Drop。當 _permit 離開作用域時，它的 Drop 實作就自動把名額還給 Semaphore。所以你只要讓 permit 在「該結束」的時候離開作用域，歸還就自動發生，不會忘記。這也是為什麼我們用 `let _permit = ...` 把它綁成一個變數——是為了讓它**活到工作結束**才被 drop；如果寫成 `let _ = ...`，它會立刻被 drop，permit 馬上就還回去了，等於沒限制到名額。

12.24.2.3 backpressure

Semaphore 帶出一個更一般的觀念：**backpressure**。

想像一條生產線：上游一直送東西進來，下游慢慢處理。如果上游送得比下游處理得快，東西就會越積越多——記憶體被塞爆只是時間問題。backpressure 的意思就是：**當下游忙不過來時，要有辦法讓上游「慢下來、等一等」**，而不是讓它無限制地塞。

Semaphore 可以用來做出這種 backpressure：permit 代表「容量」，容量滿了，想進來的人就被擋在 `acquire().await` 等待，自然就慢了下來。

下一集要講的 **bounded channel** 也是同一個道理——它的容量有限，滿了的時候 `send().await` 就會等待，逼上游放慢腳步。所以你可以用「**容量有限，滿了就等**」這個角度，去理解各種帶 backpressure 的工具。

12.24.3 重點整理

- `tokio::sync::Semaphore` 用固定數量的 **permit** 表示容量，限制「同時進行的數量」：同時下載數、同時開檔數、同時進入某段流程的 Task 數等。
- `acquire().await` 拿一張 permit，拿不到就等；permit 實作 Drop，離開作用域時自動把名額還回去。

- 用 `let _permit = ...` 讓 permit 活到工作結束才歸還；別寫成 `let _ = ...`（會馬上 drop）。
- **backpressure**：下游忙不過來時讓上游等一等，避免無限堆積；Semaphore（和下一集的 bounded channel）都可以用「容量有限、滿了就等」來理解。

12.25 mpsc

12.25.1 本集目標

學會用 async 版的 mpsc channel 在 Task 之間傳遞工作，並理解 bounded channel 的 backpressure。

12.25.2 正文

12.25.2.1 Task 之間的工作佇列

多執行緒一章我們用過 `std::sync::mpsc` 讓 Thread 之間互傳訊息。async 世界有對應的 `tokio::sync::mpsc`，是 Task 之間最常見的 queue：一邊（生產者）把工作 send 進去，另一邊（消費者）recv 出來處理。一樣是 **multi-producer single-consumer**——可以有很多發送端，但只有一個接收端。

```
extern crate tokio;

use tokio::sync::mpsc;

#[tokio::main]
async fn main() {
    // 建立一個容量 32 的 bounded channel
    let (tx, mut rx) = mpsc::channel::<i32>(32);

    // 生產者：spawn 出去送 5 個工作
    tokio::spawn(async move {
        for i in 0..5 {
            tx.send(i).await.expect("接收端已關閉");
            println!("送出 {}", i);
        }
        // tx 在這裡 drop；剩下的訊息都收完後，recv 會收到 None
    });

    // 消費者：一直收到 channel 關閉
    while let Some(value) = rx.recv().await {
        println!("收到 {}", value);
    }
    println!("channel 關閉了，結束");
}
```

`rx.recv().await` 回傳 `Option`：有訊息就是 `Some(value)`；所有發送端都 drop，而且 channel 裡剩下的訊息都收完後，就會回 `None`，於是 `while let` 自然結束。

12.25.2.2 bounded channel 與 backpressure

注意我們建立 channel 時給了一個容量 32——這是 **bounded（有容量上限的）** channel。容量上限正是上一集所提到的 backpressure：當 channel 裡累積的訊息**塞滿** 32 個（代表消費者來不及處理），生產者的 `tx.send(value).await` 就會**等待**，直到消費者收走一些、騰出空位才繼續。

這也解釋了為什麼 send 要 .await——因為它**可能要等**（等空位）。對照多執行緒一章介紹過的同步版 send，它不用等（那是無上限的）；這裡的 .await 正是 backpressure 的體現。Tokio 也有 unbounded_channel，它的 send 不用 .await，但就沒有 backpressure，要小心用。

12.25.3 重點整理

- tokio::sync::mpsc 是 async Task 之間最常見的工作佇列：多發送端、單接收端。
- rx.recv().await 回傳 Option；有訊息時回 Some，所有發送端 drop 且剩餘訊息收完後回 None。
- **bounded channel** 有容量上限，塞滿時 send().await 會等待——這就是 backpressure，逼生產者配合消費者的速度。
- send 要 .await 正是因為它可能要等空位；unbounded_channel 不用等但就沒有 backpressure。

12.26 oneshot、watch 與 broadcast

12.26.1 本集目標

認識另外三種 channel，並學會判斷該用哪一個。

12.26.2 正文

上一集的 mpsc 是「多發送、單接收」。Tokio 還有三種 channel，各自適合不同的狀況。最基本的不同之處是**發送端**和**接收端**各有幾個。

12.26.2.1 oneshot：一個值，一次

oneshot 是「**一個發送端、一個接收端、只送一個值**」。最適合「背景算一個結果，算好送回來」這種一次性的回傳。

```
extern crate tokio;

use tokio::sync::oneshot;

#[tokio::main]
async fn main() {
    let (tx, rx) = oneshot::channel::<i32>();

    tokio::spawn(async move {
        // 算好一個結果，送回去（send 只能用一次，而且不用 .await）
        tx.send(42).expect("接收端不見了");
    });

    // rx 本身就是一個 Future，.await 它就拿到那個值
    let result = rx.await.expect("發送端不見了");
    println!("拿到結果：{}", result);
}
```

注意 oneshot 的接收端 rx 本身就是一個 Future，直接 rx.await 即可。

12.26.2.2 watch：只關心「最新狀態」

watch 是「**多個發送端、多個接收端，但 channel 只保留最新的值**」。Sender 可以 clone，所以多

個 task 都能更新同一個 channel。它不是排隊收每一則訊息，而是像一個「公告欄」：各個發送端都能隨時更新上面的內容，接收端只關心「現在公告欄上寫什麼」。中間錯過的舊值不會補給你。

這最適合用來廣播如「目前設定是什麼」的狀態。

```
extern crate tokio;

use tokio::sync::watch;

#[tokio::main]
async fn main() {
    let (tx, mut rx) = watch::channel("啟動中");

    tokio::spawn(async move {
        tx.send("執行中").expect("沒有接收端");
        tx.send("完成").expect("沒有接收端");
    });

    // changed().await 等到值有更新，borrow() 讀目前最新的值
    while rx.changed().await.is_ok() {
        println!("最新狀態：{}", *rx.borrow());
    }
}
```

12.26.2.3 broadcast：把事件送給所有訂閱者

broadcast 是「**多發送、多接收，而且每個接收端都有自己的接收進度**」。和 watch 不同，它不是只給最新值，而是會把每則訊息送給所有目前訂閱的接收端。適合「一則事件要通知所有訂閱者」的場景。

```
extern crate tokio;

use tokio::sync::broadcast;

#[tokio::main]
async fn main() {
    let (tx, mut rx1) = broadcast::channel::<i32>(16);
    let mut rx2 = tx.subscribe(); // 多開一個接收端

    tx.send(1).expect("沒有接收端");
    tx.send(2).expect("沒有接收端");

    // rx1 和 rx2 都會收到 1 和 2
    println!("rx1 收到：{}", rx1.recv().await.expect("接收失敗"));
    println!("rx1 收到：{}", rx1.recv().await.expect("接收失敗"));
    println!("rx2 收到：{}", rx2.recv().await.expect("接收失敗"));
    println!("rx2 收到：{}", rx2.recv().await.expect("接收失敗"));
}
```

不過 broadcast 不是無限容量的歷史紀錄。建立 channel 時給的 16 是容量；如果某個接收端太久沒收，落後超過容量，舊訊息會被丟掉。這時 recv().await 會回傳 Lagged(n)，告訴你漏掉了幾則：

```
match rx.recv().await {
    Ok(value) => println!("收到：{}", value),
    Err(broadcast::error::RecvError::Lagged(n)) => {
```

```
        println!("太慢了，漏掉 {} 則訊息", n);
    }
    Err(broadcast::error::RecvError::Closed) => {
        println!("所有發送端都關閉了");
    }
}
```

所以比較精準地說：broadcast 會把訊息廣播給所有接收端，但每個接收端要自己跟上；如果跟不上，就會收到 Lagged，而不是保證永遠拿得到每一則舊訊息。

12.26.3 重點整理

- 不同的 channel「發送端 / 接收端數量」也不一樣。
- oneshot：單送單收、只送一個值一次，接收端本身是 Future（rx.await），適合回傳結果。
- watch：多送多收、只保留最新值，適合廣播目前狀態；用 .changed().await + .borrow()。
- broadcast：多送多收、把事件通知所有訂閱者；每個接收端有自己的進度，但落後超過容量時會收到 Lagged。
- 對照上一集的 mpsc（多送單收、收每一則、queue）。

12.27 AsyncRead 與 AsyncWrite

12.27.1 本集目標

認識 async 版的 I/O 動作，並第一次接觸 async 特有的「取消（cancellation）」概念。

12.27.2 正文

12.27.2.1 async 版的讀寫

我們在進階標準庫一章用過同步的 Read / Write trait。async 世界有對應的 AsyncRead / AsyncWrite，概念一樣，只是讀寫的動作變成可以 .await。

有個重要性質要先講：AsyncRead / AsyncWrite trait 底層真正的核心方法是 poll_read / poll_write。它們只承諾「**嘗試推進一次**」；poll_read 會把讀到的資料填進 buffer，poll_write 會回報這次實際寫入了幾個 bytes。兩者都**不保證**一次就讀滿你的 buffer，也不保證一次就把資料全部寫完。比方說你想讀 100 個 bytes，某次 poll_read 可能只填進 30 個——剩下的得之後再讀。

12.27.2.2 用 AsyncReadExt / AsyncWriteExt 的便利 helper

每次都自己處理「沒讀滿、沒寫完」很煩。所以 Tokio 在 AsyncReadExt / AsyncWriteExt 這兩個擴充 trait 裡，提供了很多幫你包好迴圈的 helper。這些 helper 內部也是反覆推進底層的 poll_read / poll_write，以下為其中兩個 helper：

- .read_exact(&mut buf)：一直讀，直到把 buf **填滿**為止。
- .write_all(buf)：一直寫，直到把 buf **整個寫完**為止。

```
use tokio::io::{AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpStream;

#[tokio::main]
async fn main() {
    let mut stream = TcpStream::connect("127.0.0.1:8080").await.expect("連線失敗");
```

```
    // write_all：寫完整個 buffer（可能呼叫 poll_write 不只一次）
    stream.write_all(b"GET / HTTP/1.0\r\n\r\n").await.expect("寫入失敗");

    // read_exact：讀滿 16 個 bytes（可能呼叫 poll_read 不只一次）
    let mut buf = [0u8; 16];
    stream.read_exact(&mut buf).await.expect("讀取失敗");
    println!("讀到 16 個 byte：{:?}", buf);
}
```

12.27.2.3 第一次認識「取消」

`read_exact` 這類 helper，剛好帶我們碰到一個 async 非常重要、卻容易忽略的概念：**取消（cancellation）**。

還記得 `Future` 是惰性的嗎？它只有被 `poll` 才會動。反過來說——如果你**不再 `poll` 它**、直接把它 `drop` 掉，那這個 async 工作就等於被**喊停**了，它後面的程式碼再也不會執行。這就是 async 的取消：**`drop` 一個 `Future` 就是取消它**。

這是 async 特有的能力。普通的 `Thread` 做不到這種乾淨的喊停——你沒辦法從外面安全地把一條正在跑的 `Thread` 中途叫停。但 async 工作只是一個還沒跑完的 `Future`，你不理它、把它丟掉，它就停了。

12.27.2.4 `read_exact` 不是 cancellation safe

取消雖然方便，卻有個陷阱。像 `read_exact` 這種「**跨好幾次推進、中途累積狀態**」的操作要小心。

想像你準備了一個 100 bytes 的 buffer，交給 `read_exact(&mut buf)`。它的目標是把整個 `buf` 填滿才回傳。可是底層 `poll_read` 可能第一次只讀到 30 bytes，所以 `read_exact` 會記住「目前已經讀到 30，還差 70」，然後繼續 `.await`。

問題在這裡：如果這個 `read_exact` 在中途被取消（被 `drop`），它記住的進度也會一起消失。沿用剛才「第一次只讀到 30 bytes」的情境，那 30 bytes 已經從 socket 取出、寫進 `buf` 的前面了；但 `read_exact` 還沒成功回傳，所以它也沒有把「目前已經讀了 30 bytes」這件事交還給你。換句話說，這次「讀滿 100 bytes」的動作停在半路，剩下 70 bytes 不會自動補完。

取消也可能發生在別的時間點：如果發生得更早，可能其實 0 bytes 都還沒讀到；如果發生得更晚，可能已經讀到 80 bytes；如果剛好讀滿 100 bytes，`read_exact` 就有可能正常完成。麻煩的是：只要它還沒正常完成就被丟掉，你就失去那個「目前到底讀到哪裡」的進度。對需要照順序解析的 I/O 來說，已經被讀走的 bytes 不能當作沒讀過重來一次；如果你沒有自己另外保存進度，後續就很難安全地接著讀。

我們說 `read_exact` 和 `write_all` **不是 cancellation safe**：它被中途取消會留下爛攤子（可能已經消費了一部分資料，但整個「填滿 buffer」的動作沒有完成）。所以你**不該**把 `read_exact` 或 `write_all` 這類操作，放進「可能會被中途丟掉」的地方。

那「可能會被中途丟掉的地方」是哪裡？最典型的就是下一集要講的 `select!`——它天生就會在某個 branch 完成時，把其他還沒完成的 branch `drop` 掉（也就是取消）。所以下一集我們會再回到 cancellation safety 的話題，看看在 `select!` 裡怎麼避免踩到這個坑。

12.27.3 重點整理

- `AsyncRead` / `AsyncWrite` 是 async 版的 `Read` / `Write`；底層核心是 `poll_read` / `poll_write`，每次只嘗試推進一次：`poll_read` 把資料填進 buffer，`poll_write` 回報

本次寫入的 bytes 數，兩者都不保證讀滿或寫完。

- `AsyncReadExt` / `AsyncWriteExt` 提供 `read_exact`、`write_all` 等 helper，幫你包好「讀滿 / 寫完」的迴圈。
- **取消**：`Future` 是惰性的，`drop` 掉一個 `Future`（不再 `poll`）就等於取消這個 `async` 工作——這是 `async` 特有、`Thread` 做不到的。
- `read_exact` 這類「跨多次推進、累積中間狀態」的操作**不是 cancellation safe**：中途被取消時，可能已經消費了一部分資料，但整個「讀滿 buffer」的動作沒有完成，不該放進會被中途 `drop` 的地方。

12.28 `select!`

12.28.1 本集目標

學會用 `select!` 等待多個 branch，直到其中一個完成且輸出符合 pattern，並理解它和取消（cancellation）的密切關係。

12.28.2 正文

12.28.2.1 等「誰先到」

`join!` 等的是「**全部**都完成」。`select!` 可說是它的一種相反：它同時等多個 branch，只要其中一個完成，而且輸出符合左邊的 pattern，就執行那個 branch 對應的 handler，然後整個 `select!` 就結束——**其他還沒完成的 branch 會被 `drop` 掉**。

12.28.2.2 基本語法

`select!` 的每個 branch 大致長這樣：

```
tokio::select! {
    pattern = future => {
        // future 完成後，輸出會被 pattern 接住
    }
    _ = other_future => {
        // 不關心 other_future 的輸出
    }
}
```

等號前的 pattern 用來接住等號後那個 `Future` 的輸出；如果 pattern 裡綁定了變數，那個變數會被帶進右邊的大括號裡使用。等號後寫的是要等待的 `Future` 而已，這裡**不要**自己加 `.await`。`select!` 會負責同時 `poll` 這些 `Future`，等其中一個完成且輸出符合 pattern。

如果你不需要某個 `Future` 的輸出，就像一般 `match` pattern 一樣用 `_` 忽略它：

```
tokio::select! {
    value = compute() => {
        println!("算出來了：{}", value);
    }
    _ = shutdown.recv() => {
        println!("收到 shutdown 訊號");
    }
}
```

如果輸出本身是 `Option<T>` 或 `Result<T, E>`，最直覺的寫法是先把整個值接住，再在 handler 裡

自己 match：

```
tokio::select! {
    message = receiver.recv() => {
        match message {
            Some(message) => println!("收到訊息：{}", message),
            None => println!("channel 關閉"),
        }
    }
    _ = shutdown.recv() => {
        println!("準備關閉");
    }
}
```

select! 本身也可以有回傳值，回傳的是勝出 branch 大括號裡最後一個表達式。這點跟 match 很像：每個 branch 都要回傳同一種型別。

```
let status = tokio::select! {
    value = compute() => {
        println!("算出來了：{}", value);
        "done"
    }
    _ = shutdown.recv() => {
        println!("收到 shutdown 訊號");
        "shutdown"
    }
};

println!("狀態：{}", status);
```

select! 最經典的用途是 **timeout**：把「真正的工作」和「一個計時器」一起 select!，看誰先到。

```
extern crate tokio;

use tokio::time::{sleep, Duration};

async fn do_work() {
    sleep(Duration::from_secs(5)).await; // 假設工作要五秒
    println!("工作完成");
}

#[tokio::main]
async fn main() {
    tokio::select! {
        _ = do_work() => {
            println!("工作順利做完了");
        }
        _ = sleep(Duration::from_secs(1)) => {
            println!("逾時！工作太久了，不等了");
        }
    }
}
```

這裡計時器一秒就到，比五秒的工作快，所以 select! 走計時器那個 branch、印出「逾時」，然後 **把 do_work() 那個 Future drop 掉**——工作就此被取消。

`select!` 很適合這些情境：

- **timeout**（上面的例子）。
- **同時接收多個 channel**：哪個 channel 先有訊息就處理哪個。
- **等待 shutdown signal**：一邊做正常工作，一邊聽「該收工了」的訊號，誰先到就反應誰。

12.28.2.3 在迴圈裡用 `select!` 要小心 cancellation safety

剛剛提到了 `drop`：這正是上一集講的 **cancellation**：`drop` 一個 `Future` 就是取消它。而 `select!` 天生就會在某個 branch 勝出時，把其他 branch 全部 `drop`。理解這一點，後面用 `select!` 才不會踩雷。

`select!` 常常被放在 `loop` 裡反覆使用（例如一個伺服器迴圈：每輪 `select!` 等「新工作」或「shutdown 訊號」）。這種寫法要特別小心上一集的 **cancellation safety**。

回想上一集：`read_exact` 這類「跨多次推進、累積中間狀態」的操作**不是 cancellation safe**，中途被取消時，可能已經消費了一部分資料，但整個「讀滿 buffer」的動作沒有完成。而 `select!` 每一輪都可能因為**別的** branch 先完成，而把這一個 branch 的 `Future` `drop` 掉（取消）。如果你把 `read_exact` 放進 `select!` 的某個 branch，又在 `loop` 裡反覆跑，那它就很可能在讀到一半時被取消，留下不好接續的半成品。

輸掉的 branch 不會執行自己的大括號，這本來就是 `select!` 的正常行為，問題不在這裡。真正要小心的是：那個輸掉的 `Future` 在被丟掉以前，可能已經造成一部分外部效果，例如從 socket 讀走一些 bytes，或把一部分資料寫出去。

所以風險不是「handler 沒跑」，而是「`Future` 被取消時，已經做了一半的事沒有被完整收尾」。如果這個操作需要跨多步累積進度，就應該把進度放在 `select!` 外面，branch 裡只等待一次可以安全取消的小步驟。本章後面我們會示範如何遵守這樣的設計原則。

12.28.2.4 幾個實用補充

`select!` 還有幾個常用功能：

branch precondition：在 branch 後面加 `, if 條件`。這個條件可以使用進入 `select!` 前就已經存在的變數，例如下面的 `accepting_jobs`；但不能使用左邊 pattern 才會綁定的變數。`job` 要等 `jobs.recv()` 完成且 `Some(job)` 匹配成功後，才能在 handler 裡使用。

```
tokio::select! {
    Some(job) = jobs.recv(), if accepting_jobs => {
        handle(job).await;
    }
    _ = shutdown.recv() => {
        accepting_jobs = false;
    }
}
```

處理一次 `select!` 時，相關步驟依序是：

1. 先求值所有 branch 的 `if` precondition；條件為假的 branch 會在這次 `select!` 中被停用。
2. 求值所有等號右邊的 `async` 表達式，包括已停用 branch 的表達式。條件為假時，表達式仍會被求值以建立 `Future`，但產生的 `Future` 不會被 `poll`。這裡的「求值」是建立 `Future`，不是執行其中的 `async` 工作；不過，建立 `Future` 前的參數運算等一般表達式仍會執行。
3. `poll` 其餘 branch 的 `Future`。
4. 某個 `Future` 完成後，嘗試用它的輸出匹配左邊的 pattern。匹配成功就執行 handler 並結束

select!；匹配失敗則停用這個 branch，繼續等待其他 branch。因此最先完成的 branch 不一定會勝出；勝出的是完成且 pattern 匹配成功的 branch。

5. 所有 branch 都被停用時，執行 else；如果沒有 else，select! 會 panic。

else branch：例如 Some(job) = jobs.recv() 遇到 channel 關閉、.recv() 回 None 時，Some(job) 會匹配失敗，這個 branch 便會被停用。如果其他 branch 也全部被停用，就會執行 else。

```
tokio::select! {
    Some(job) = jobs.recv(), if accepting_jobs => {
        handle(job).await;
    }
    Some(msg) = messages.recv(), if accepting_messages => {
        handle_message(msg).await;
    }
    else => {
        break; // 這一輪沒有任何 branch 能跑
    }
}
```

公平性與 biased;：select! 預設會隨機選擇從哪個 branch 開始 poll。這主要在 select! 被反覆執行（通常是在 loop 裡），而且多個 branch 持續 ready 時有影響。每次改變起點，能降低某個 branch 只因排在前面而每輪都勝出的風險。

加上 biased; 後，select! 會固定由上到下 poll。各個 branch 仍是一個一個被 poll，但只要其中一個回傳 Ready 且輸出符合 pattern，整個 select! 就會立刻結束。因此，排在前面且一直 ready 的 branch，可能讓後面的 branch 永遠得不到 poll：

```
loop {
    tokio::select! {
        biased;

        // 如果一直有訊息在等，這個 branch 就會一直是 Ready
        Some(message) = messages.recv() => {
            handle_message(message).await;
        }
        // 即使 shutdown 已經到達，這個 branch 也可能永遠得不到 poll
        _ = shutdown.recv() => {
            break;
        }
    }
}
```

如果 messages.recv() 每一輪都立刻 ready，它就會在程式檢查到 shutdown.recv() 之前勝出，這就是 starvation。使用 biased; 時，不能延遲的 branch 應該排在前面：

```
loop {
    tokio::select! {
        biased;

        _ = shutdown.recv() => {
            break;
        }
        Some(message) = messages.recv() => {
            handle_message(message).await;
```

```
        }
    }
}
```

12.28.3 重點整理

- `select!` 同時等多個 branch，第一個完成且輸出符合 pattern 的 branch 會執行對應 handler，其他 branch 被 `drop`（取消）。
- 基本語法是 `pattern = future => { ... }`；`=>` 左邊不用寫 `.await`，不需要輸出時用 `_ = future`；`select!` 本身也能回傳勝出 branch 的值。
- 所以 `select!` 是程式裡**最常製造取消**的地方；適合 timeout、多 channel 接收、等 shutdown 訊號。
- 在 `loop` 裡用 `select!` 要注意 cancellation safety：別把 `read_exact` 這類不可安全取消的 `Future` 放進會被 `drop` 的 branch。
- 補充功能：branch `if`（precondition）、pattern 不匹配時停用該 branch、`else`（所有 branch 都被停用時），以及用 `biased;` 固定由上到下 `poll`。

12.29 async 的 Mutex、RwLock 與 Notify

12.29.1 本集目標

搞懂為什麼有時候得用 Tokio 版的鎖、什麼時候用標準庫的就好，並認識喚醒工具 `Notify`。

12.29.2 正文

12.29.2.1 從 Send / Sync 的一個例外講起

先回到多執行緒一章的 Send / Sync。日常的型別有個規律：一個型別只要是 Sync（能被多條 Thread 同時借用），通常它也是 Send（能搬到別條 Thread）。

但有些少數的例外：`std::sync::Mutex` 和 `RwLock` 的 **guard**（`.lock()` 回傳的那個 `MutexGuard` / `RwLockReadGuard` / `RwLockWriteGuard`）是 **Sync 但不是 Send**。為什麼？因為在某些作業系統上，一把鎖**必須由當初上鎖的那條 Thread 來解鎖**；如果把 guard 搬到別條 Thread 才 `drop`（解鎖），就會出錯。所以標準庫乾脆禁止這些 guard 被 Send。

12.29.2.2 這些例外會影響到 async

這樣不實作 Send 的特性，在 async 裡會變成一個讓人困惑的編譯錯誤。回想第 21 集：一個 `Future` 跨 `.await` 時持有非 Send 的東西，整個 `Future` 就不是 Send，於是不能 `tokio::spawn`。而標準庫內的 guard 正是非 Send 的——所以**抓著標準庫的 guard 跨 `.await`** 就會中招：

```
use std::sync::{Arc, Mutex};

async fn do_io() {}

#[tokio::main]
async fn main() {
    let data = Arc::new(Mutex::new(0));
    tokio::spawn(async move {
        let mut guard = data.lock().expect("鎖失敗"); // 標準庫的 MutexGuard 不是 Send
        do_io().await; // 抓著 guard 跨 .await
```

```
        *guard += 1;
    }); // 編譯錯誤：future 不是 Send，不能 spawn
}
```

這個錯誤其實是個**有益的警告**——它正好提醒你違反了一條重要紀律：**Mutex 保護的是共享的可變狀態，lock 的作用域應該越短越好，盡量不要抓著 lock 去等 I/O**。拿著鎖等 I/O 的話，其他人在這段時間全被擋在鎖外面，並行就可能會崩了。

所以最好的解法通常不是「想辦法跨 .await 持有鎖」，而是**縮短 lock 作用域**：在 .await 之前就把該改的改完、讓 guard 離開作用域：

```
use std::sync::{Arc, Mutex};

async fn do_io() {}

#[tokio::main]
async fn main() {
    let data = Arc::new(Mutex::new(0));
    tokio::spawn(async move {
        {
            let mut guard = data.lock().expect("鎖失敗");
            *guard += 1;
        } // guard 在這裡就 drop 了，沒有跨 .await
        do_io().await; // 等 I/O 時手上沒抓著鎖
    });
}
```

12.29.2.3 必要時才用 Tokio 的鎖

但有時候你真的**需要**抓著鎖跨 .await（例如要在持有鎖的狀態下做一個 async 操作，且邏輯上不能拆開）。這種時候才改用 tokio::sync::Mutex——它的 guard 是 Send 的，可以安全地跨 .await：

```
use std::sync::Arc;
use tokio::sync::Mutex; // 注意是 tokio 的 Mutex

#[tokio::main]
async fn main() {
    let data = Arc::new(Mutex::new(0));
    let d = data.clone();
    tokio::spawn(async move {
        let mut guard = d.lock().await; // 注意 .lock() 要 .await
        *guard += 1; // 這個 guard 是 Send，可以跨 .await
    });
}
```

但請記得：**標準庫的鎖比 Tokio 的鎖快**（Tokio 的鎖為了能跨 .await 付出額外成本）。所以預設用 std 的鎖、縮短作用域；只有「非抓著鎖跨 .await 不可」時，才動用 Tokio 的 Mutex。

和標準庫一樣，Tokio 也有 RwLock，把讀寫分開：.read().await 允許多個讀者同時進入，.write().await 則獨佔給單一寫者。使用原則也與 Tokio 的 Mutex 一樣。

12.29.2.4 Notify：不帶資料的喚醒

最後介紹 tokio::sync::Notify。它是一個**不帶 payload（不帶資料）的喚醒工具**——它能讓一個 Task 睡著等（.notified().await），讓另一個 Task 戳它一下叫它起來（.notify_one()），但**不**

傳任何值。

```
use std::sync::Arc;
use tokio::sync::Notify;

#[tokio::main]
async fn main() {
    let notify = Arc::new(Notify::new());
    let n = notify.clone();

    let handle = tokio::spawn(async move {
        n.notified().await; // 睡著等通知
        println!("被通知了，醒來做事");
    });

    notify.notify_one(); // 戳醒一個等待者
    handle.await.expect("Task 發生 panic");
}
```

`Notify` 通常搭配一塊**自己用 `Mutex` 管理的共享狀態**使用：你改完共享狀態後，`notify` 一下，被叫醒的 Task 自己去看狀態變成什麼。它**不是 queue**——多次呼叫 `.notify_one()` 可能合併成一次。如果當下沒有人在等，`Notify` 最多只會保存一個 permit。

12.29.2.5 `Notify` 和 `watch` 的差別

`Notify` 容易和第 26 集的 `watch` 搞混，但兩者定位不同：

- **`Notify`：不帶資料，最多保存一個 permit**。它只負責「戳人起床」，至於起床要看什麼，得你自己用 `Mutex` 之類的東西管著。
- **`watch`：帶有「最新值」**。它本身就存著一份最新的狀態，接收端醒來直接讀得到。

12.29.3 重點整理

- 標準庫的 `Mutex` / `RwLock` guard 是 `Sync` 但非 `Send`（某些作業系統規定上鎖的 `Thread` 上才能解鎖），抓著它跨 `.await` 會讓 `Future` 非 `Send`、不能 `spawn`。
- 這個編譯錯誤是有益的警告：`Mutex` 的 lock 作用域要短，別抓著鎖等 I/O；通常縮短作用域（`.await` 前就 `drop` guard）即可。
- 一定要抓著鎖跨 `.await` 時才用 `tokio::sync::Mutex`（guard 是 `Send`，`.lock().await`），但優先用標準庫更快的鎖。
- Tokio `RwLock` 把讀寫分開：`.read().await` 多讀、`.write().await` 一寫。
- `Notify` 是不帶資料的喚醒工具，最多保存一個 permit；多次呼叫 `.notify_one()` 可能合併，而 `watch` 會保存最新值。

12.30 `Stream`

12.30.1 本集目標

認識 `Stream`——`async` 版的 `Iterator`，以及怎麼走訪它。

12.30.2 正文

12.30.2.1 Stream 是 async 版的 Iterator

第 6 章的 Iterator 是「一連串值，要一個一個取」。但它的 .next() 是**同步**的——呼叫就馬上給你下一個值（或 None）。

Stream 是它的 async 版本：一樣是一連串值要一個一個取，但下一個值**可能要等**（例如等網路送來下一筆資料、等計時器、等使用者輸入）。所以 Stream 的 .next() 回傳的是一個 Future，你要 .next().await 才拿得到下一個值。

對照記就很好懂：

- iterator.next() → 回傳 Option<Item>（同步、馬上給）。
- stream.next().await → 回傳 Option<Item>（要 .await、可能等一下）。

兩者都用「None 代表結束」。

本集的範例會用到 tokio-stream 這個 crate（它不在 Tokio 本體裡），使用前要加上依賴：

```
[dependencies]
tokio-stream = "0.1"
```

一個小地方要注意：crate 名稱在 Cargo.toml 裡寫 tokio-stream（連字號），但在程式碼裡要寫成 tokio_stream（底線）——crate 名稱裡的 - 到了程式碼裡一律變成 _。

12.30.2.2 走訪一個 Stream

Iterator 可以用 for 走訪，但 Stream 不行（for 沒辦法 .await）。Stream 的標準走訪寫法是 **while let Some(x) = stream.next().await**——一個一個取，取到 None 就停：

```
extern crate tokio;
extern crate tokio_stream;

use tokio_stream::StreamExt;

#[tokio::main]
async fn main() {
    // 從一個 Vec 做出最簡單的 stream
    let mut stream = tokio_stream::iter(vec![1, 2, 3]);

    // 一個一個取值，取到 None 為止
    while let Some(value) = stream.next().await {
        println!("收到 {}", value);
    }
}
```

12.30.2.3 Stream 不在標準庫裡

有件事要特別說明：和 Future 不同，Stream **目前不在標準庫裡**。Stream trait 定義在 futures-core crate 中；tokio-stream 會重新匯出它，並提供自己的 StreamExt。要用本集的 next、map、filter 等方法，得引入 tokio_stream::StreamExt：

```
extern crate tokio;
extern crate tokio_stream;

use tokio_stream::StreamExt;
```

```rust
#[tokio::main]
async fn main() {
    // 和 Iterator 一樣可以串接 map / filter 這些工具
    let mut stream = tokio_stream::iter(1..=5)
        .map(|x| x * 2)
        .filter(|x| x % 3 == 0);

    while let Some(value) = stream.next().await {
        println!("{}", value);
    }
}
```

你會發現 map、filter 這些方法跟第 6 章的 Iterator 幾乎一模一樣——因為 Stream 本來就是 Iterator 的 async 翻版。學過 Iterator，Stream 對你來說只是多了 .await。

實務上 Stream 很適合表達「源源不絕、會陸續到來的資料」——例如一個一個進來的網路連線、資料庫查詢的逐筆結果或定時觸發的事件。tokio_stream 提供了一整套處理它們的工具。

12.30.3 重點整理

- Stream 是 async 版的 Iterator：一連串值一個一個取，但下一個值可能要等，所以是 .next().await。
- 對照：iterator.next() 同步回 Option；stream.next().await 要 .await 才回 Option；都用 None 表示結束。
- 走訪用 **while let Some(x) = stream.next().await**（Stream 不能用 for）。
- Stream 不在標準庫，定義在 futures；用 tokio_stream::StreamExt 取得 next、map、filter 等方法（用法和 Iterator 幾乎一樣）。

12.31 JoinSet 與 FuturesUnordered

12.31.1 本集目標

學會處理「大量、動態產生、誰先好先處理」的並行工作，並理解 JoinSet 和 FuturesUnordered 的取捨。

12.31.2 正文

12.31.2.1 join! 的不足

join! 很好用，但它有兩個限制：數量**固定**（你寫程式時就得列出所有 branch），而且它要等**全部**完成。

可是很多時候你的工作是「**大量、動態產生、而且誰先好就先處理誰**」——例如爬一千個網頁。這種需求 join! 應付不來，得換工具。有兩條路，差別在於「要不要變成獨立 Task」。

12.31.2.2 路線一：JoinSet（spawn 的動態版）

tokio::task::JoinSet 可以想成「**spawn 的動態版**」。你往裡面 spawn 任意多個工作，每一個都是**獨立的 Task**。在多執行緒 runtime 上，Tokio 可以把它們排到不同的 worker Thread，因此能夠平行執行（而且和 spawn 一樣需要 Send + 'static）。然後用 join_next().await 把完成的結果一個一個收回來——**誰先完成就先拿到誰**：

```rust
use tokio::task::JoinSet;
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    let mut set = JoinSet::new();

    // 動態 spawn 五個工作，故意讓延遲長短不同
    for i in 0..5 {
        set.spawn(async move {
            sleep(Duration::from_millis(100 * (5 - i))).await;
            i
        });
    }

    // 誰先做完就先收到誰（不是按 spawn 的順序）
    while let Some(result) = set.join_next().await {
        let value = result.expect("task panic 或被取消");
        println!("完成：{}", value);
    }
}
```

`join_next()` 回傳 `Option<Result<T, JoinError>>`：

- `None`：已經沒有 Task 了，收完了。
- `Some(Ok(value))`：一個 Task 順利完成。
- `Some(Err(...))`：那個 Task panic 或被取消（所以要處理這個 `Err`）。

`JoinSet` 還支援 `.abort_all()` 把所有工作一次取消，而且 `JoinSet` 被 `drop` 時會**自動取消**裡面所有還沒完成的 Task——這在做 graceful shutdown 時很方便（下一集會用到）。

12.31.2.3 路線二：FuturesUnordered（join! 的動態版）

`futures::stream::FuturesUnordered` 則是「**join! 的動態版**」。它在**同一個 Task 內**輪流推進一堆 Future，**不**把它們變成獨立 Task，也**不一定**跨 Thread。代價和好處都從這裡來：

- 因為它不會把這些 Future spawn 成獨立 Task，所以 FuturesUnordered 本身**不要求這些 Future 是 Send + 'static**——它可以放借用了區域變數的 Future（JoinSet 因為要 spawn 就做不到）。
- 但因為大家在同一個 Task 上輪流，如果一個 Future 的 poll 阻塞或執行太久，就會延後其他 Future 被 poll（又是「不要 block 住執行緒」那條鐵律）。

`FuturesUnordered` 定義在 `futures` 這個 crate 裡，使用前要加上依賴（上一集加過的 `tokio-stream` 這裡也會用到）：

```toml
[dependencies]
futures = "0.3"
tokio-stream = "0.1"
```

`FuturesUnordered` 本身其實就是一個 `Stream`——它會追蹤喚醒通知，只 `poll` 剛加入或已被喚醒的 Future，不會把它們 spawn 成 Task。所以它**不依賴特定 runtime**，這是它相對於 `JoinSet` 的一大優點（`JoinSet` 的 spawn 就綁死 Tokio runtime）。用 `Stream` 的方式走訪它：

```rust
extern crate futures;
extern crate tokio;
extern crate tokio_stream;

use futures::stream::FuturesUnordered;
use tokio_stream::StreamExt;

#[tokio::main]
async fn main() {
    let mut futures = FuturesUnordered::new();

    // 動態塞進一堆 Future（不會變成獨立 Task）
    for i in 0..5 {
        futures.push(async move { i * 2 });
    }

    // 它是個 Stream，誰先完成就先冒出來
    while let Some(value) = futures.next().await {
        println!("完成：{}", value);
    }
}
```

12.31.2.4 怎麼選

兩者都是「誰先完成就先產生結果」，都很適合爬蟲、批次請求這類工作。差別在：

- 想讓每個工作成為**能夠平行執行的獨立 Task，並有較高的排程隔離** → 用 **JoinSet**（但要 Send + 'static、綁 Tokio）。
- 想**就地借用區域變數、工作輕量、不想依賴特定 runtime** → 用 **FuturesUnordered**（同一個 Task 內多工，不需 Send，但一個 Future 的 poll 阻塞或執行太久，就會延後其他 Future 被 poll）。

下一集，我們把目前學的這些工具——select!、channel、JoinSet——兜成一個完整的 graceful shutdown 流程。

12.31.3 重點整理

- 處理「大量、動態、誰先好先處理」的工作，join! 不夠用，改用 JoinSet 或 FuturesUnordered。
- **JoinSet**（spawn 的動態版）：每個工作是獨立 Task、在多執行緒 runtime 上能夠平行執行、需 Send + 'static、綁 Tokio；join_next() 回 Option<Result<T, JoinError>>，支援 .abort_all() 與 drop 時自動取消。
- **FuturesUnordered**（join! 的動態版）：在同一個 Task 內多工，不會 spawn 成獨立 Task，而且本身不要求其中的 Future 是 Send + 'static（因此在周圍情境允許時可以借用區域變數）；但一個 Future 的 poll 阻塞或執行太久，就會延後其他 Future 被 poll。它本身是個不綁 runtime 的 Stream。
- 要獨立 Task、可以平行執行並有較高的排程隔離，用 JoinSet；要就地借用、工作輕量、不綁 runtime，用 FuturesUnordered。

12.32 graceful shutdown

12.32.1 本集目標

把前面學的工具兜成一個完整的 graceful shutdown（優雅關閉）流程。

12.32.2 正文

12.32.2.1 什麼是 graceful shutdown

伺服器要關閉時，最粗暴的做法是直接砍掉——但這樣進行到一半的工作就斷在那裡，可能留下壞掉的資料、沒回應完的請求。**graceful shutdown** 是更有禮貌的關法：收到停止要求時不硬切，而是「**通知大家收工** → **等手邊的工作做完（或到期限）** → **乾淨退出**」。

把它拆成三個要素：

1. **訊號來源**：怎麼知道「該關了」。
2. **廣播 shutdown**：怎麼把「要收工了」告訴所有 worker。
3. **等待 drain**：怎麼等所有 worker 收尾完畢。

我們用前面學過的工具，一個一個對上。

12.32.2.2 三要素組起來

- **訊號來源**用 `tokio::signal::ctrl_c()`——它是個 `Future`，`.await` 它會等到使用者按下 Ctrl-C（實務上還會再加上監聽 SIGTERM）。
- **廣播 shutdown** 用第 26 集的 `watch` 當一個 shutdown flag：一對多、而且晚訂閱的 worker 也讀得到當前狀態。
- **等待 drain** 用第 31 集的 `JoinSet`，`join_next()` 一直收到全空為止。

每個 worker 內部用 `select!`，**同時**等「下一份工作」和「shutdown 訊號」。如果先等到工作，就離開 `select!` 處理它；如果先等到 shutdown，就不再拿新工作：

```rust
use std::time::Duration;
use tokio::sync::watch;
use tokio::task::JoinSet;
use tokio::time::{sleep, timeout};

async fn wait_next_job(id: u32, next_job: &mut u32) -> u32 {
    // 這裡用 sleep 假裝「等待下一份工作送進來」。
    // 這個等待可以被 shutdown 取消；真正處理工作會放在 select! 外面。
    sleep(Duration::from_millis(500)).await;
    let job = *next_job;
    *next_job += 1;
    println!("worker {} 拿到工作 {}", id, job);
    job
}

async fn process_job(id: u32, job: u32) {
    // 這裡才是假裝「真正處理工作」。
    // 它刻意放在 select! 外面，所以 shutdown 不會直接把它取消在半路。
    sleep(Duration::from_millis(300)).await;
    println!("worker {} 處理完工作 {}", id, job);
}
```

```rust
async fn worker(id: u32, mut shutdown: watch::Receiver<bool>) {
    let mut next_job = 0;

    loop {
        let job = tokio::select! {
            // 等下一份工作：這件事可以被 shutdown 取消
            job = wait_next_job(id, &mut next_job) => job,
            // 收工訊號
            _ = shutdown.changed() => {
                println!("worker {} 收到收工訊號，退出", id);
                break;
            }
        };

        // 在 select! 外面處理，避免被 shutdown 直接 drop 在半路
        process_job(id, job).await;
    }
}

#[tokio::main]
async fn main() {
    // 廣播 shutdown 用的 watch flag
    let (shutdown_tx, shutdown_rx) = watch::channel(false);

    // 用 JoinSet 管理所有 worker
    let mut workers = JoinSet::new();
    for id in 0..3 {
        workers.spawn(worker(id, shutdown_rx.clone()));
    }

    // 1. 等訊號
    tokio::signal::ctrl_c().await.expect("監聽 Ctrl-C 失敗");
    println!("收到 Ctrl-C，開始 graceful shutdown");

    // 2. 廣播收工
    shutdown_tx.send(true).expect("沒有 worker 在聽");

    // 3. 等所有 worker drain，但給 5 秒期限
    match timeout(Duration::from_secs(5), async {
        while workers.join_next().await.is_some() {}
    })
    .await
    {
        Ok(()) => println!("所有 worker 都乾淨退出了"),
        Err(_) => {
            println!("逾時！強制取消剩下的 worker");
            workers.abort_all();
        }
    }
}
```

這裡的 `timeout(Duration::from_secs(5), future)` 可以讀成：「最多等這個 future 五秒」。

它自己也是一個 Future。如果裡面的 future 在五秒內完成，`.await` 會得到 `Ok(裡面的輸出)`；如果五秒到了還沒完成，`.await` 會得到 `Err(_)`。在這個例子裡，裡面的 future 是：

```
async {
    while workers.join_next().await.is_some() {}
}
```

也就是「一直等 worker 結束，直到 JoinSet 空掉」。所以整段 timeout 的意思是：**最多等五秒讓所有 worker 自己收尾；五秒內都退完就印成功，超過五秒就進入 Err(_)，把剩下的 worker 強制取消**。

12.32.2.3 cancellation safety 的設計重點

這裡有個呼應第 27、28 集的關鍵設計：要**刻意安排 select! 的位置**。在上面的 worker 裡，select! 等的是「下一份工作」和「shutdown」；一旦真的拿到工作，就離開 select!，再呼叫 process_job。因此 shutdown 勝出時，被 drop（取消）的是「等下一份工作」這種可以重來的等待，而不是已經開始處理的工作。

如果反過來，把真正的處理流程直接放進會輸給 shutdown 的 branch，像 read_exact 這類不可安全取消的操作就可能被砍在半路，資料也跟著掉了。這就是前面強調過的 cancellation safety 在 shutdown 上的具體應用。

12.32.2.4 一定要給期限

graceful 不代表**無限期**等。萬一某個 worker 卡死了，你不能讓整個程式陪它一直耗下去。所以 drain 一定要**給期限**：上面用 tokio::time::timeout 把整個 drain 包起來，逾時就 abort_all()（或直接 drop 掉 JoinSet，它會自動取消剩下的 Task）強制收掉。

一句話總結這個原則：**先禮貌地等，等不到就動手**。

12.32.2.5 更匹配的工具：CancellationToken

用 watch 當 shutdown flag 可行，但有點像「借」一個狀態廣播工具來當開關。tokio-util 提供了一個從頭就為「取消」設計的工具——CancellationToken，語意更貼切。tokio-util 不在 Tokio 本體裡，使用前要加上依賴：

```
[dependencies]
tokio-util = "0.7"
```

（和第 30 集的 tokio-stream 一樣，crate 名稱裡的 - 在程式碼中會變成 _：use tokio_util::...。）

把上面的 watch 換成它：

```
use std::time::Duration;
use tokio::task::JoinSet;
use tokio::time::{sleep, timeout};
use tokio_util::sync::CancellationToken;

async fn wait_next_job(id: u32, next_job: &mut u32) -> u32 {
    sleep(Duration::from_millis(500)).await;
    let job = *next_job;
    *next_job += 1;
    println!("worker {} 拿到工作 {}", id, job);
    job
}

async fn process_job(id: u32, job: u32) {
    sleep(Duration::from_millis(300)).await;
```

```
    println!("worker {} 處理完工作 {}", id, job);
}

async fn worker(id: u32, token: CancellationToken) {
    let mut next_job = 0;

    loop {
        let job = tokio::select! {
            job = wait_next_job(id, &mut next_job) => job,
            _ = token.cancelled() => { // 直接等「被取消」
                println!("worker {} 收到取消，退出", id);
                break;
            }
        };

        process_job(id, job).await;
    }
}

#[tokio::main]
async fn main() {
    let token = CancellationToken::new();

    let mut workers = JoinSet::new();
    for id in 0..3 {
        workers.spawn(worker(id, token.clone())); // 每個 worker 拿一份 clone
    }

    tokio::signal::ctrl_c().await.expect("監聽 Ctrl-C 失敗");
    token.cancel(); // 一聲令下，全部取消

    match timeout(Duration::from_secs(5), async {
        while workers.join_next().await.is_some() {}
    })
    .await
    {
        Ok(()) => println!("全部退出"),
        Err(_) => {
            println!("逾時！強制取消剩下的 worker");
            workers.abort_all();
        }
    }
}
```

`token.cancelled()` 是一個等「被取消」的 `Future`，`token.cancel()` 一呼叫，所有持有 `clone` 的 worker 都會醒來。它讀起來就是「取消」的意思，比借 `watch` 當開關更貼合需求。

12.32.3 重點整理

- graceful shutdown：不硬切，而是「通知收工 → 等做完（或到期限）→ 乾淨退出」。
- 三要素：訊號來源（`tokio::signal::ctrl_c()`）、廣播 shutdown（`watch` flag）、等待 drain（`JoinSet` 的 `join_next()` 到全空）。
- `select!` 適合用來同時等「下一份工作」與「shutdown」；若真正的工作不能安全取消，就先用 `select!` 拿到工作，再離開 `select!` 處理，避免 shutdown 把處理中的工作 `drop` 在半路

（cancellation safety）。
- drain 一定要給期限：用 `tokio::time::timeout` 包住，逾時就 `abort_all()` 或 drop `JoinSet`——先禮貌地等，等不到就動手。
- 更匹配的工具是 `tokio_util` 的 `CancellationToken`：`token.cancel()` 一聲令下，所有 `token.cancelled()` 都醒來，語意比借 `watch` 更貼切。

12.33 測試 async 程式

12.33.1 本集目標

學會用 `#[tokio::test]` 寫 async 測試，以及用虛擬時間讓牽涉延遲的測試跑得又快又穩。

12.33.2 正文

12.33.2.1 `#[tokio::test]`

第 7 章學過用 `#[test]` 加 `cargo test` 寫測試。但 `#[test]` 標記的是一個普通函數，沒辦法 `.await`。要測 async 程式，Tokio 提供 `#[tokio::test]`——它會自動幫你的測試函數套上一個 runtime，你不用自己 `block_on`：

```
async fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[tokio::test]
async fn test_add() {
    let result = add(2, 3).await;
    assert_eq!(result, 5);
}
```

就這麼簡單。`#[tokio::test]` 等於「`#[test]` + 自動準備 runtime + 允許 async」。其他和第 7 章一樣：放進 `#[cfg(test)] mod tests`、用 `cargo test` 執行、用 `assert_eq!` 之類的巨集檢查結果。

12.33.2.2 牽涉時間的測試怎麼辦

async 程式常常牽涉時間——timeout、延遲、定時重試。如果照實測，一個「5 秒後逾時」的邏輯，測試就得真的等 5 秒，又慢又煩。

Tokio 的解法是**虛擬時間**：讓測試裡的時間由你**手動推進**，不必真的空等。兩個關鍵函數：

- `tokio::time::pause()`：把時間「暫停」，從此時間不會自己流動。
- `tokio::time::advance(duration)`：手動把時間往前快轉一段。

```
use tokio::time::{self, Duration};

#[tokio::test]
async fn test_with_virtual_time() {
    time::pause(); // 暫停時間

    let start = time::Instant::now();

    // 把虛擬時間往前推 10 秒——瞬間完成，不必真的等
    time::advance(Duration::from_secs(10)).await;
```

```
    assert_eq!(start.elapsed(), Duration::from_secs(10));
}
```

這個測試**瞬間**就跑完了，即使邏輯上「過了 10 秒」。因為時間是虛擬的，advance 一下就跳過去了。如果你想讓測試從一開始就暫停時間，也可以直接寫 `#[tokio::test(start_paused = true)]`，省掉手動呼叫 `pause()`。

有了虛擬時間，凡是牽涉 timeout、延遲、重試間隔的測試，都能變得 **deterministic（每次結果一致）**又快——你完全掌控時間怎麼走，不必看真實時鐘的臉色。

（小提醒：pause / advance 這些虛擬時間工具需要開啟 Tokio 的 `test-util` 功能，在 `Cargo.toml` 把 Tokio 的 features 加上 `"test-util"` 即可。）

12.33.3 重點整理

- `#[tokio::test]` 自動幫測試函數套上 runtime、允許 `.await`，等於「`#[test]` + runtime + async」；其餘用法和第 7 章的 `cargo test` 一樣。
- 牽涉時間的測試別用真實時間（又慢又不穩），改用 Tokio 的虛擬時間。
- `tokio::time::pause()` 暫停時間、`tokio::time::advance(duration)` 手動快轉，讓 timeout / 延遲的測試瞬間完成且結果一致。
- 也可用 `#[tokio::test(start_paused = true)]` 從頭暫停時間；虛擬時間工具需要 Tokio 的 `test-util` 功能。

12.34 Tokio 以外的 runtime

12.34.1 本集目標

認識 Tokio 以外的 async runtime，並學會分辨自己寫的程式哪些綁定特定 runtime、哪些不綁。

12.34.2 正文

12.34.2.1 標準庫只定義語言層的抽象

這一章我們花了很大力氣，從零手寫了一個 runtime。現在你應該會很清楚，標準庫只定義了**語言層的抽象**——`Future trait`、`Poll`、`Context`、`Waker`、`Pin` 這些。但「怎麼真正把 `Future` 跑起來」——executor 怎麼排程、reactor 怎麼盯 I/O、計時器怎麼實作——標準庫一概不管，全部留給第三方 runtime 自由發揮。我們前面手寫的東西（executor、reactor、timer、`Task` 的設計），正是一個 runtime 該包含的零件。

12.34.2.2 不只有 Tokio

Tokio 是目前最主流的 runtime，但不是唯一的選擇。因為標準庫不規定 runtime 怎麼寫，社群就長出了好幾種各有特色的 runtime：

- **Tokio**：功能最完整、生態最大的通用 runtime，多執行緒、什麼都有。本章後半用的就是它。
- **smol**：走輕量、精簡路線的 runtime，核心很小、容易理解。
- **monoio / glommio**：走 **thread-per-core** 路線的特化 runtime，常搭配 Linux 的 io_uring，為極致 I/O 效能而生。
- **Embassy**：給**嵌入式**裝置用的 runtime，能在沒有作業系統、沒有標準庫的微控制器上跑 async。

這些 runtime 在各方面可能都不一樣：用幾條執行緒、怎麼排程、I/O 怎麼做、計時器怎麼實作、spawn 的細節與限制。選哪個，取決於你的場景——寫一般網路服務用 Tokio 最省事；做嵌入式就得用 Embassy。

12.34.2.3 runtime-agnostic vs runtime-specific

既然有這麼多 runtime，你寫程式時可以有個意識：你寫的這段程式碼，到底**綁不綁**特定 runtime？

- **runtime-agnostic（不綁 runtime）的部分**：純粹的 `Future` 組合邏輯。例如你自己 `impl Future`、用 async / `.await` 串接、用 `join!` / `select!` 組合、用 `FuturesUnordered`——這些只依賴標準庫的 `Future` 抽象，搬到別的 runtime 上通常照樣能用。
- **runtime-specific（綁 runtime）的部分**：真正碰到外部世界或排程的東西。例如 `tokio::net::TcpStream`（I/O）、`tokio::time::sleep`（timer）、`tokio::spawn`（排程）——這些是 Tokio 提供的，換一個 runtime 就得換成它對應的版本。

實務上不必為了「runtime 中立」而綁手綁腳——大部分專案選定 Tokio 就一路用到底。但知道這個界線，能幫你在「想換 runtime」或「寫一個給別人用、不想綁死 runtime 的函式庫」時，清楚哪些程式碼可以原封不動、哪些得抽換。

12.34.3 重點整理

- Rust 標準庫只定義 `Future` 等**語言層抽象**，不內建 runtime；executor、reactor、timer、I/O、Task 設計都由 runtime 提供（正是我們手寫過的那些零件）。
- Tokio 是主流通用 runtime；此外還有輕量的 smol、thread-per-core 的 monoio / glommio、嵌入式用的 Embassy 等，各方面設計可能不同。
- 寫程式時可留意：純 `Future` 組合邏輯（自訂 `Future`、`join!`、`select!`、`FuturesUnordered`）大多 **runtime-agnostic**；I/O、timer、spawn 則是 **runtime-specific**，換 runtime 要抽換。

恭喜你完成了非同步這一章！🎉 這一章從第一個 `async fn` 開始，一路拆到 `Future`、`poll`、`Waker`、executor、reactor、`Pin`，再回到 Tokio 的 spawn、I/O、channel、`select!`、graceful shutdown 和測試。走到這裡，你已經看過 async 背後那套「惰性的 `Future` 如何被 runtime 推進」的完整骨架。之後在使用 Tokio 或其他 runtime 時，你看到的不只是非同步的 API，而也知道那些 API 大概在替你安排哪些事情。