從零開始學 Rust

Andy Shiue

生成日期：2026 年 6 月 19 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。個人目前建議使用 GPT-5.5 Thinking，自己試過效果還不錯。

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
- 馬具：OpenClaw / Claude Code

目錄

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**，它是 Rust 的專案管理工具。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：

```rust
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** 是 Rust 的專案管理工具，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？

變數就像一個容器，你可以把東西放進去，之後再拿出來用。

來看看怎麼用：

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

這裡 `let x = 5;` 就是在說：「我要建立一個叫 x 的變數，然後把 5 放進去。」

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

變數不只能放數字，也能放文字：

```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，但一定要賦值一次

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);
    println!("{} - {} = {}", a, b, a - b);
    println!("{} * {} = {}", a, b, a * b);
    println!("{} / {} = {}", a, b, a / b);
    println!("{} % {} = {}", a, b, a % b);
}
```

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
- 不確定順序時加括號，既安全又讓程式碼更好讀

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。之後我們會學到迴圈、函數等等，只要看到 `{}`，裡面就是一個作用域。這是 Rust 裡面一個很統一的規則。

1.9.3 重點整理

- 每對大括號 `{}` 都會建立一個作用域（scope）
- 在作用域裡建立的變數，出了 `{}` 就消失、不能再用
- `if` 和其他帶有 `{}` 的語法都會形成作用域

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()` → `.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

```
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("請輸入數字");

    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

```
    let num = input.trim().parse::<i32>().expect("請輸入數字");
```

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("請輸入數字")` → 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

```
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("請輸入數字");

    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("請輸入數字")`

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

```
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("請輸入數字");

    // 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()` → 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，裡面的迴圈就會完整跑完一輪。

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，外層跑一次、內層就完整跑一輪
- 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 跳過迴圈中的某些輪次。

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` 直接跳到外層的下一輪，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 跳過外層迴圈的一輪

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：

```rust
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（float，64 位元），就是帶小數點的數字
- `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：

```rust
    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
`i32`	`u32`	32	約 ±21 億

有號	無號	位元數	範圍（有號）
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

浮點數就是帶小數點的數字，只有兩種：

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——**浮點數除法會保留小數**：

```
fn main() {
    let a = 10.0;
    let b = 3.0;
    println!("{}", a / b); // 3.3333333333333335
    println!("{}", a % b); // 1.0
}
```

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...。

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 不會自動幫你轉換型別。

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
- 浮點數除法保留小數，但有精確度問題（`0.1 + 0.2 ≠ 0.3`）
- 不同型別的數字不能直接混著算，Rust 不會自動轉型

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 x = 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);
}
```

let 只能放在 fn 裡面，但 const 可以放在最外層，讓整個程式都能用到。

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
- 可以放在函數外面，讓整個程式都能用

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("請輸入數字"); // 轉成數字，還是叫 input
    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("請輸入數字");

        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 {:?} 印出來的結果一樣。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、行數和值，超方便：

```
[src/main.rs:3] x = 5
[src/main.rs:4] 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，會印出檔名和行數

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。但函數參數不行——因為函數是你的「對外介面」，Rust 希望介面要清清楚楚的，不要搞得模模糊糊。

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，**要加分號**
- 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（記得加分號）
- 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），不然會無限迴圈
- 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 → 2（包含）
- 索引 2 → 3（包含）
- 索引 3 → 4（包含）
- 索引 4 → 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：

```rust
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

```rust
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**的陣列了！

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：

```rust
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);
}
```

這段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!("（下一集學 match 之後，就能根據方向做不同的事）");
}
```

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。

```
    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期是未知的，所以你可以用不同長度的模式來匹配：

```
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,
}
```

```rust
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

```rust
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：

```rust
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 @ 在比對範圍的同時，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：`1..=5` 可以比對 1 到 5 之間的值。但有個問題——比對成功後，你沒辦法知道「到底是 1、2、3、4 還是 5」，因為你只知道它在這個範圍裡。

@（at 符號）可以解決這個問題。它讓你在比對的同時，把實際的值存到一個變數裡：

```
fn main() {
    let age = 50;
    match age {
        n @ 0..=12 => println!("{}歲，是小孩", n),
        n @ 13..=19 => println!("{}歲，是青少年", n),
        n => println!("{}歲，是大人", n),
    }
}
```

`n @ 0..=12` 的意思是「如果值在 0 到 12 之間，就把這個值叫做 n」。這樣你就能同時做範圍檢查和取值了。

3.19.3 範例程式碼

```
fn main() {
    let age = 15;

    match age {
        n @ 0..=6 => println!("{}歲，學齡前", n),
        n @ 7..=12 => println!("{}歲，國小", n),
        n @ 13..=15 => println!("{}歲，國中", n),
        n @ 16..=18 => println!("{}歲，高中", n),
        n => println!("{}歲，已成年", n),
    }

    // 搭配 char 使用
    let ch = 'k';

    match ch {
        c @ 'a'..='m' => println!("'{}' 在字母表前半段", c),
        c @ 'n'..='z' => println!("'{}' 在字母表後半段", c),
        c => println!("'{}' 不是小寫字母", c),
    }

    // 更實用的例子：HTTP 狀態碼
    let status = 404;

    match status {
        code @ 200..=299 => println!("成功！狀態碼：{}", code),
        code @ 300..=399 => println!("重新導向，狀態碼：{}", code),
        code @ 400..=499 => println!("用戶端錯誤，狀態碼：{}", code),
        code @ 500..=599 => println!("伺服器錯誤，狀態碼：{}", code),
```

```
        code => println!("未知狀態碼：{}", code),
    }
}
```

3.19.4　@ 不只能搭配 range

@ 可以搭配任何模式，不只是 range。比如搭配 | （多個值）：

```
fn main() {
    let day = 6;

    match day {
        d @ (1 | 3 | 5 | 7) => println!("第 {} 天，是休息日", d),
        d @ (2 | 4 | 6) => println!("第 {} 天，是工作日", d),
        d => println!("第 {} 天，不是合法的日子", d),
    }
}
```

`d @ (1 | 3 | 5 | 7)` 的意思是「如果值是 1、3、5 或 7，就把它叫做 d」。注意 | 的部分要用括號包起來。

3.19.5　重點整理

- `n @ 1..=5` 在比對範圍的同時，把值綁定到變數 n
- 語法：變數名 `@` 模式
- @ 可以搭配任何模式，不只是 range：`d @ (1 | 3 | 5 | 7)` 也行
- 如果不用 @，你只知道值符合模式，但不知道具體是多少
- @ 的概念是「把符合這個模式的值，用這個名字存起來」

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

有時候光靠模式匹配還不夠，你還需要加上一些額外的條件。比如說，你想用 `match` 判斷一個數是奇數還是偶數——這沒辦法用 range pattern 或固定值表達，因為它需要做運算（`% 2`）。

Rust 的 **`match` guard** 讓你在模式後面加上 `if` 條件：

```
fn main() {
    let n = 137;
    match n {
        x if x % 2 == 0 => println!("{} 是偶數", x),
        x => println!("{} 是奇數", x),
    }
}
```

`x if x % 2 == 0` 的意思是「先把值綁定到 x，然後額外檢查 `x % 2 == 0` 是否成立」。只有模式匹配**而且** guard 條件為 `true` 的時候，這個分支才會被執行。

注意：guard 不算在「窮舉」的判斷裡。就算你寫了所有可能的 guard，Rust 可能還是會要求你加 _ 預設分支。

3.20.3 範例程式碼

```
fn main() {
    let n = 8;

    match n {
        x if x % 2 == 0 => println!("{} 是偶數", x),
        x => println!("{} 是奇數", x),
    }

    // 搭配 tuple 使用
    let point = (3, 7);

    match point {
        (x, y) if x == y => println!("在對角線上：({}, {})", x, y),
        (x, y) if x > 0 && y > 0 => println!("({}, {}) 在第一象限", x, y),
        (x, y) => println!("其他的點 ({}, {})", x, y),
    }
}
```

3.20.4 重點整理

- match guard：在模式後面加 if 條件 做額外判斷
- 語法：模式 if 條件 => ...
- 只有模式匹配**且**條件為 true 時，分支才會執行
- guard 可以使用模式裡綁定的變數（如 x if x > 0）
- guard 條件不算窮舉，通常最後還是要加 _ 預設分支

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 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：**tuple 和 struct 的解構不會失敗**。一個 (i32, i32) 一定有兩個值，一個 Point 一定有 x 和 y——沒有其他可能。

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失敗的模式（irrefutable pattern），所以 tuple 和 `struct` 可以，`enum` 會有問題

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 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

```
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」
- `Self` 可以用在回傳型別 `-> 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 概念說明

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？」

這就是所謂的「資料競爭（data race）」。Rust 的所有權規則就是為了**從根本上防止這種問題**。

4.1.2.4 `clone` = 買一個新的保險箱

那如果我真的需要兩份一樣的資料怎麼辦？

答案是：**不要複製鑰匙，而是買一個新的保險箱，把裡面的東西複製一份放進去，然後配一把新的鑰匙。**

這樣兩個人各有自己的保險箱、自己的鑰匙，互不干擾。

在 Rust 裡面，這叫做 **`clone`（克隆）**。它會完整複製一份資料，產生一個全新的、獨立的副本。

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，這可能會造成資料競爭
- **`clone`（克隆）**：買新的保險箱 + 複製內容物 + 配新鑰匙，兩份完全獨立
- Rust 在編譯時就強制執行所有權規則，防止資料競爭

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！");
    }
}

// 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` 來複製資料。

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」：

```
    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：完整複製一份

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()` 來複製：

```
#[derive(Debug, Clone)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = p1.clone();  // 複製一份，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 就是「複製一份完整的鑰匙圈和保險箱」。兩個變數各自擁有自己的資料，互不干擾。

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 複製一份，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()` 可以複製一份獨立的副本
- 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：

```
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。

你可以把 Copy 想像成：「這個東西太小了、太簡單了，複製一份根本不費力，所以 Rust 直接幫你複製，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：

```
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 對很多其他 trait 也有同樣的行為——只要所有元素都有實作某個 trait，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)]：

```
#[derive(Debug, Copy, Clone)]
struct Point {
    x: i32,
    y: i32,
}
```

注意：#[derive(Copy)] 一定要同時加 Clone——如果只寫 #[derive(Copy)] 而不寫 Clone，編譯器會報錯。

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「手動複製」。如果一個東西連手動複製都不行，那自動複製當然更不行。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：

```rust
#[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()）
適用場景	小而簡單的資料	任何資料
使用限制	所有欄位都必須是 Copy	沒有特別限制

簡單來說：**copy 是自動的複製，clone 是手動的複製。**

4.4.3 範例程式碼

```rust
#[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：**只有你確定這個型別永遠都會很小、很簡單，而且不會再加非 Copy 的欄位時，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——需要複製的時候手動寫 .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 對很多 trait（Copy、Clone 等）都有同樣的行為：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 一定要搭配 Clone 一起 derive
- **Copy = 自動複製，Clone = 手動複製（.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**：複製一份，但如果資料很大，複製就很浪費

有沒有辦法**不交出去、不複製，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，你隨時可以繼續用 p。

&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**。所以參考和借用是一體兩面：參考是那個「借據」，借用是「借東西來看」這個動作。之後看到「參考」這個詞，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 & 參考是唯讀的

用 & 借用的時候，你**只能讀，不能改**。如果你想借來改，那是下一集的內容。

4.5.3 範例程式碼

```
#[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
- & 參考是**唯讀**的，不能修改借來的資料

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

```
#[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 &，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。

這也是**資料競爭（data race）**，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 懸垂參考（dangling reference）

還有一個重要的規則：**參考不能活得比它的原值還久**。一旦值離開了作用域被丟棄，任何指向它的參考就會變成**懸垂參考**——指向一個已經不存在的地方。Rust 在編譯時就會阻止這種事情發生。

最常見的情況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。

另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：**參考不能活得比它指向的值還久**。

4.7.3 範例程式碼

```rust
#[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 離開作用域，可變借用結束

    // 可變借用結束了，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造成**資料競爭**，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
- **懸垂參考**：參考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就防止資料競爭，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

參數型別	意思
`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;
}
```

```rust
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`——借用自己產生一份新的，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 記憶體的兩個區域

程式執行的時候，資料會被放在記憶體裡。記憶體有兩個主要的區域：

堆疊（stack）：

- 快速
- 大小在編譯時就確定了
- 函數的區域變數、整數、浮點數、布林值、char 都放在這裡
- 函數結束時，這些變數就自動被清掉了

堆積（heap）：

- 比較慢，但更靈活
- 大小可以在程式執行時才決定（例如一段文字，你不知道使用者會輸入多長）
- 需要手動管理（在其他語言裡），或靠所有權系統自動管理（在 Rust 裡）

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：

- **鑰匙圈上的裝飾** = stack 上的資料（小而固定，跟著鑰匙圈走）
- **保險箱** = heap 上的資料（大而靈活，存在別的地方）
- **鑰匙** = 指標（pointer），記錄了保險箱在記憶體中的位置

所以當我們說「move 是把鑰匙圈交出去」：

- 如果鑰匙圈上只有裝飾（stack 資料），交出去很便宜，甚至可以直接複製一份（這就是 Copy！）
- 如果鑰匙圈上有鑰匙（指標），交出去的是指標，保險箱（heap 資料）不會複製

4.9.2.3 為什麼整數是 Copy？

現在你應該懂了：整數（i32 等）就像鑰匙圈上的小裝飾，完全在 stack 上，小小的、複製起來零成本。所以 Rust 讓它們自動 Copy。

而像之後會學到的 String 那樣的型別，它的資料在 heap 上。如果隨便複製，就等於複製了整個保險箱裡的東西，代價很大。所以 Rust 需要用 move，要複製就得明確呼叫 .clone()。

4.9.3 範例程式碼

```rust
#[derive(Debug, Copy, Clone)]
struct StackData {
    x: i32,
    y: i32,
    active: bool,
}

fn main() {
    // 這些都在 stack 上
    let a = 42;    // i32，4 bytes，stack
    let b = 3.14;  // f64，8 bytes，stack
    let c = true;  // bool，1 byte，stack
    let ch = '🦀'; // char，4 bytes，stack
    println!("整數：{}，浮點：{}，布林：{}，字元：{}", a, b, c, ch);
```

```
    // struct 裡面全是 stack 資料，所以整個 struct 也在 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);

    // 陣列也在 stack 上（大小固定）
    let arr = [1, 2, 3, 4, 5];
    println!("陣列：{:?}", arr);

    // tuple 也在 stack 上
    let t = (42, true, 'A');
    println!("tuple：{:?}", t);

    // 之後會學 String 和 Vec，它們的資料在 heap 上
    // 到時候 move 和借用的重要性就更明顯了！
}
```

4.9.4 重點整理

- **stack（堆疊）**：快速、大小固定。整數、浮點、布林、char、小 struct 都在這裡
- **heap（堆積）**：靈活、大小可變。大型或動態大小的資料放在這裡
- 鑰匙圈比喻揭秘：裝飾 = stack 資料、保險箱 = heap 資料、鑰匙 = 指標
- 整數是 Copy，因為它們完全在 stack 上，複製幾乎沒有成本
- heap 資料預設 move（只轉移鑰匙），要完整複製就用 clone（複製整個保險箱的內容）

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 複製一份，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？	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（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); // &Vec 自動轉成 &[i32]
    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);
    }
    println!("v 還在：{:?}", v); // OK！
}
```

`for x in &v` 只是借用，v 不會被消耗。

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：&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;   // &Vec → &[i32]
    println!("Vec: {:?}, slice: {:?}", v, v_ref);
}
```

4.13.4 Vec 的元素只能是 i32 嗎？

讀到這裡你可能有個疑問：Vec 的元素只能是 i32 嗎？當然不是！下一章將會花大量篇幅討論相關內容。

4.13.5 重點整理

- **Vec 和 String 的所有權行為完全對稱**：都在 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
- 大部分情況用 for x in &v，除非你確定不再需要這個 Vec
- Vec 的元素不只能放 i32——下一章會學到怎麼讓它放其他型別

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！

回想一下，Vec<i32> 和 Vec<String> 就是同一個 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("請輸入數字");
}
```

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

第二集我們定義了泛型 `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——只在你**確定**不會是 None 的時候才用。

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 Result 和 Option 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("請輸入數字")` 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

```rust
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——不一致時要另外處理
- `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。

5.12.2.3 預設實作可以呼叫其他方法

注意上面的 description 預設實作裡呼叫了 self.name()。這是允許的——預設實作可以使用同一個 trait 中的其他方法。這讓你可以建立「只要提供幾個基本方法，其他方法就自動有了」的設計。

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 不會引入新功能，它只是讓長路徑變短。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 只是路徑的簡寫，不引入新功能
- 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)
    }
}

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：

```
fn show_clone<T: Clone + std::fmt::Display>(x: &T) {
    let cloned = x.clone();
    println!("原始：{}", x);
    println!("克隆：{}", 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：

```
fn show_clone<T>(x: &T)
where
    T: Clone + std::fmt::Display,
{
    let cloned = x.clone();
    println!("原始：{}", x);
    println!("克隆：{}", 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
// clone 一份，印出原始值，然後回傳複製品
fn clone_and_show<T: Clone + Display>(x: &T) -> T {
    println!("複製了：{}", 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!("拿到的複製品：{}", cloned);

    let cloned2 = clone_and_show(&String::from("hello"));
    println!("拿到的複製品：{}", 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) 完全等價——都是說「x 的型別必須實作 Display」。只是寫法更簡潔。

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
    // 即使我們知道裡面其實是 String，編譯器只認 Display 這個 trait
}
```

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——Clone、Display、Describe，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 trait 定義值離開作用域時的清理行為，以及手動提前釋放資源。

5.20.2 概念說明

到目前為止，我們知道值離開作用域就不能用了。但其實 Rust 在背後還做了一件事——值離開作用域時，Rust 會自動**丟棄**（drop）它，釋放它佔用的資源（包括記憶體）。大部分時候你不需要在意這件事，但有時候你想在值被丟棄的那一刻做一些額外的事情，比如印一條訊息、關閉檔案、清理暫存等。

5.20.2.1 Drop trait

Drop trait 讓你自訂「被丟棄時要做什麼」：

```
impl Drop for MyType {
    fn drop(&mut self) {
        println!("MyType 被丟棄了！");
    }
}
```

Rust 會在值離開作用域時**自動呼叫** drop。你不能手動呼叫 x.drop()——Rust 禁止這樣做，因為值被 drop 之後又被自動 drop 一次會出問題。

5.20.2.2　手動提前釋放

如果你想提前釋放一個值，用 `drop()`：

```
struct MyType { name: String }

impl Drop for MyType {
    fn drop(&mut self) {
        println!("MyType 被丟棄了！");
    }
}

fn main() {
    let x = MyType { name: String::from("小明") };
    drop(x); // 提前丟棄
    // x 不能再用了
}
```

`drop` 是一個函數（不是 method），它會取走值的所有權，然後讓值離開作用域，觸發 `Drop`。

5.20.2.3　有 Drop 的型別不能部分 move

這是一個重要的限制。如果一個 `struct` 實作了 `Drop`，你就不能從它的欄位 move 出值：

```
struct Resource {
    name: String,
    id: i32,
}

impl Drop for Resource {
    fn drop(&mut self) {
        println!("釋放 {}", self.name);
    }
}

fn main() {
    let r = Resource { name: String::from("A"), id: 1 };
    let n = r.name; // 編譯錯誤！不能部分 move
}
```

為什麼？因為 `drop` 需要完整的 `self`。如果你把 `name` move 走了，`drop` 執行時 `self.name` 就不存在了——這不安全。所以 Rust 禁止有 `Drop` 的型別做部分 move。

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);
    }
}

fn main() {
```

```
    let a = Resource { name: String::from("資料庫連線") };
    let b = Resource { name: String::from("檔案處理器") };

    println!("建立了兩個資源");

    // 手動提前釋放 a
    drop(a);
    println!("a 已經被提前釋放了");

    // a 不能再用了
    // println!("{}", a.name); // 編譯錯誤！

    println!("接下來 b 會在 main 結束時自動釋放");

    // 作用域示範
    {
        let c = Resource { name: String::from("暫時的資源") };
        println!("c 在這個作用域裡");
    } // c 在這裡被自動 drop

    println!("c 已經被釋放了，b 還在");
} // b 在這裡被自動 drop
```

5.20.4 重點整理

- Drop trait 讓你自訂值離開作用域時的清理行為
- Rust 在值離開作用域時**自動呼叫** .drop()，不能手動呼叫
- 想提前釋放，用 drop(value)
- **有 Drop 的型別不能部分 move**——因為 drop 需要完整的 self

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——它把資料放在 heap 上，然後在 stack 上留一把鑰匙（指標）。

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 上只留一個指標。

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 是唯一擁有者——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 是唯一擁有者，move 語義和其他非 Copy 的型別一樣

5.22 Rc<T>

5.22.1 本集目標

學會用 Rc<T> 讓多個擁有者共享同一份資料，理解參考計數的原理。

5.22.2 概念說明

上一集學了 Box<T>——一個保險箱只有一把鑰匙，一個擁有者。但有時候你需要**多個擁有者**共享同一份資料。

5.22.2.1 問題：一個值只能有一個擁有者

```
    let a = Box::new(String::from("hello"));
    let b = a; // move！a 不能再用了
```

如果你希望 a 和 b 都能用這個值，怎麼辦？

你可能會想：「那 clone 一份不就好了？」

```
    let a = Box::new(String::from("hello"));
    let b = a.clone(); // 複製了整個 String 的內容
```

這確實能讓 a 和 b 都能用。但問題是——clone 是真的把 heap 上的資料**完整複製**了一份。如果資料很大（比如一個很長的 Vec），每次 clone 都是一筆不小的開銷。而且 a 和 b 指向的是**兩份獨立的資料**，改了 a 不會影響 b。

如果你需要的是「多個人共享**同一份**資料」，Box 的 clone 就不是正確的工具了。

5.22.2.2 Rc：參考計數

還記得第 4 章的保險箱比喻嗎？Rust 預設的規則是「一個保險箱只有一把鑰匙」——這就是所有權。但 Rc<T> 打破了這個預設：它讓你**配好幾把鑰匙，都能打開同一個保險箱**。

Rc<T>（reference counting）用一個「計數器」來追蹤目前有幾把鑰匙：

- 建立 Rc 時，計數 = 1
- .clone() 時，計數 +1（不會複製資料，只是增加計數）
- 某個 Rc 離開作用域時，計數 -1
- 計數歸零時，資料才會被真正釋放

5.22.2.3 Rc 的 .clone() 不是深度複製

對 Rc 呼叫 .clone() 和我們之前學的 .clone() 不一樣——它不會複製裡面的資料，只是增加參考計數。所以速度很快，成本很低。

Rc 的 clone 不是用 #[derive(Clone)] 產生的，而是標準庫自己手動實作的。如果用 derive，它會深度複製裡面的資料；但 Rc 的 clone 只是增加計數器，行為完全不同。

5.22.2.4 Rc 是唯讀的

Rc<T> 只提供不可變的存取——所有擁有者都只能讀，不能改。如果需要可以改，之後會學 RefCell。

5.22.2.5 等等，第 4 章不是說多把鑰匙會有問題嗎？

第 4 章說一個保險箱只有一把鑰匙。那 Rc 怎麼能配好幾把？有兩件事要知道：

1. **計數器的代價**：Rc 內部有一個計數器來追蹤目前有幾把鑰匙，這樣才知道什麼時候該銷毀保險箱。這個計數器在每次 clone 和 drop 時都要更新，是 Box 沒有的額外開銷。Box 保證只有一把鑰匙，離開作用域就銷毀，不需要計數——Rc 並不是像 Box 那樣基本的操作，而是用額外的機制換來「多個擁有者」的能力
2. **使用上的限制**：未來講到多執行緒的時候會提到，Rc 事實上也無法在某些情況下避免資料競爭，因此有使用上的限制。這裡先知道就好

換句話說，Rust 原生的所有權規則就是「一把鑰匙」——這是語言層面的保證，零成本。Rc 的「多把鑰匙」是用計數器在執行時期**模擬**出來的，繞過了原生的限制，但也因此有額外的成本和限制。

而且要注意：Rc 本身還是遵守一般的所有權規則的——每個 Rc 變數都像是一個獨立的值，move 就 move、drop 就 drop。只是它的 .clone() **在邏輯上**像是「打一把新鑰匙」，而不是「複製整個保險箱」。

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 只是增加計數，不複製資料
```

```
    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 都可以讀取
    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：多個 struct 共享同一份資料
    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> 用參考計數讓多個擁有者共享同一份資料
- Rc::new(value) 建立時計數為 1
- .clone() 計數 +1，只增加計數，不複製內部資料——成本很低
- 某個 Rc 被 drop 時計數 -1，歸零時才釋放資料
- Rc 是**唯讀的**——所有擁有者只能讀，不能改
- Rust 原生的「一個擁有者」是零成本的語言保證；Rc 的「多個擁有者」是用計數器在執行時期模擬出來的，有額外開銷和限制
- Rc 本身遵守一般的所有權規則（move、drop），只是 .clone() 在邏輯上是「打一把新鑰匙」而非「複製保險箱」
- 用 Rc::strong_count(&x) 查看目前的參考計數

5.23 Deref 與自動解參考

5.23.1 本集目標

理解 Deref trait 和 Rust 的自動解參考機制，以及智慧指標為什麼能直接呼叫內部型別的方法。

5.23.2 概念說明

5.23.2.1 對智慧指標使用 *

到目前為止，我們只對一般的參考（&T）用過 *。但其實 * 也能用在其他型別上：

```rust
fn main() {
    let b = Box::new(42);
    let val: i32 = *b; // 把值從 Box 裡拿出來
    println!("{}", val); // 42
}
```

*b 得到的是 Box 裡面的 i32。這之所以能成立，是因為 Box<T> 實作了一個叫 Deref 的 trait。

5.23.2.2 Deref trait 與智慧指標

Deref 告訴 Rust：「當你需要解參考我的時候，該怎麼做。」Box<T> 和 Rc<T> 都實作了 Deref。在 Rust 中，我們常常把**實作了 Deref 的型別**叫作**智慧指標（smart pointer）**。

5.23.2.3 *v 背後發生了什麼

當你對一個實作了 Deref 的型別使用 * 時，Rust 實際上會這樣展開：

```rust
*v
// 等同於
*(Deref::deref(&v))
```

Deref::deref 接收 &Self，回傳一個參考（例如 &T），然後外面的 * 再把這個參考解開，得到 T 本身。

以剛才的 Box<i32> 為例：

```rust
let b = Box::new(42);

*b
// 展開為 *(Deref::deref(&b))
// Deref::deref(&b) 回傳 &i32
// 再 * 一次得到 i32
```

所以解參考 Box<T> 最終得到的是 T。Rc<T> 也一樣，解參考 Rc<T> 得到 T。

5.23.2.4 DerefMut

DerefMut 是 Deref 的可變版本。當你對一個可變的智慧指標寫入時，Rust 會用 DerefMut 來展開：

```rust
*v = 新的值
// 等同於
*(DerefMut::deref_mut(&mut v)) = 新的值
```

DerefMut::deref_mut 回傳 &mut T，外面的 * 解開後就能寫入值。例如：

```rust
fn main() {
    let mut b = Box::new(0);
    *b = 42;
    println!("{}", *b); // 42
}
```

Box<T> 同時實作了 Deref 和 DerefMut，所以既能讀也能寫。Rc<T> 則只實作了 Deref，不允許透過 * 修改內容。

5.23.2.5 Deref coercion

Deref coercion 是 Rust 在需要的時候自動透過 Deref 轉換參考型別的機制。這不只發生在 method call，任何需要型別匹配的地方都可能觸發。

例如，一個函數接受 &i32，你可以直接傳 &Box<i32> 進去，Rust 會自動透過 Deref 把 &Box<i32> 轉成 &i32：

```
fn show(val: &i32) {
    println!("{}", val);
}

fn main() {
    let b = Box::new(42);
    show(&b); // deref coercion：&Box<i32> 自動轉成 &i32
}
```

Deref coercion 也可以連鎖。例如 &Box<Box<i32>> 會先 deref 成 &Box<i32>，再 deref 成 &i32。DerefMut 同理。

5.23.2.6 method call 的自動解參考

method call 有另一套獨立的機制。前面學過 (&a).method() 可以簡寫為 a.method()——如果 method 接收的是 &self，Rust 會自動幫你加 &。反過來，如果你有一個 &T 或智慧指標，而方法定義在 T 上，Rust 也會自動幫你加 *。

當你用 . 呼叫方法時，Rust 會嘗試加 &、加 *、或兩者組合，一層一層嘗試，直到找到有對應方法的型別。如果 a 是 &Box<i32>，而你呼叫一個定義在 i32 上、接收 &self 的方法，Rust 會做 (&**a).method()——先 *a 得到 Box<i32>，再 * 得到 i32，再 & 回去得到 &i32 來匹配 &self。

來看一些比較簡單的例子：

```
let boxed = Box::new(String::from("hello"));

// 你寫的：
boxed.len()

// Rust 實際上做的：
(*boxed).len()
// *boxed 得到 String，String 有 len()，找到了
```

如果有多層包裝，Rust 會一層一層剝開：

```
let double_boxed = Box::new(Box::new(String::from("hello")));

// 你寫的：
double_boxed.len()

// Rust 實際上做的：
(**double_boxed).len()
// *double_boxed 得到 Box<String>，沒有 len()
// 再 * 一次得到 String，有 len()，找到了
```

Rc 也一樣：

```
use std::rc::Rc;

fn main() {
    let rc = Rc::new(vec![1, 2, 3]);
    println!("{}", rc.len()); // 自動解參考，呼叫 Vec 的 len()
}
```

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 本身有 clone 方法（增加參考計數），T 可能也有 clone 方法（深度複製資料）。直接呼叫 .clone() 會拿到 Rc 的 clone：

```
use std::rc::Rc;

fn main() {
    let a = Rc::new(String::from("hello"));
    let b = a.clone(); // Rc 的 clone，增加參考計數，不複製 String
}
```

如果你想呼叫內層 String 的 clone，可以明確寫出來：

```
    let c = (*a).clone(); // String 的 clone，真的複製了一份 String
```

5.23.3　範例程式碼

```
use std::rc::Rc;

fn show(val: &i32) {
    println!("值：{}", val);
}

fn main() {
    // *Box<T> 得到 T（Deref）
    let b = Box::new(42);
    let val: i32 = *b;
    println!("解參考 Box：{}", val);

    // DerefMut：透過 * 寫入值
    let mut b = Box::new(0);
    *b = 42;
    println!("寫入後：{}", *b);

    // deref coercion：&Box<i32> 自動轉成 &i32
    let b = Box::new(99);
    show(&b);

    // 自動解參考：Box<String> 直接呼叫 String 的方法
    let boxed = Box::new(String::from("hello"));
    println!("Box 裡的字串長度：{}", boxed.len());
    // 等同於 (*boxed).len()

    // Rc 也一樣
    let rc = Rc::new(vec![10, 20, 30]);
    println!("Rc 裡的 Vec 長度：{}", rc.len());

    // clone 的優先順序
    let a = Rc::new(String::from("shared"));
    let b = a.clone();       // Rc 的 clone（快，只增加計數）
    let c = (*a).clone();    // String 的 clone（慢，複製整個 String）
    println!("a = {}, b = {}, c = {}", a, b, c);
    println!("Rc 計數 = {}", Rc::strong_count(&a)); // 2，不是 3
```

```
}
```

5.23.4 重點整理

- 在 Rust 中，實作了 Deref 的型別常被稱為智慧指標；*v 展開為 *(Deref::deref(&v))，所以解參考 Box<T> 得到 T
- DerefMut 是 Deref 的可變版本；*v = 值 展開為 *(DerefMut::deref_mut(&mut v)) = 值
- Deref coercion：Rust 在型別不匹配時會自動透過 Deref 轉換參考，不限於 method call（如 &Box<i32> → &i32）
- method call 的自動解參考是獨立的機制：用 . 呼叫方法時，Rust 會嘗試加 &、加 * 或兩者組合來找到對應的方法
- 方法同名時外層優先——Rc 的 clone 優先於 String 的 clone

5.24 Cell<T>

5.24.1 本集目標

學會用 Cell<T> 在不可變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 &（不可變參考）的情況下，還是想修改值。

5.24.2.1 Cell 的概念

Cell<T> 提供一種「繞過借用規則」的方式——它用 .get() 取值、.set(v) 設值，**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 Cell 有一個重要的限制：

5.24.2.2 T 必須是 Copy

Cell<T> 的 .get() 會把值**複製一份**出來（不是借用）。所以 T 必須實作 Copy。

你不能用 Cell<String>，因為 String 不是 Copy。只能用 Copy 的型別（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

上一集學的 Rc<T> 需要一個參考計數器——每次 clone 時計數 +1，drop 時計數 -1。但 Rc 對外只提供 &self（不可變參考），計數器卻需要被修改。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（不可變參考），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
- **`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>，但它的限制是 T 必須是 Copy。如果你想修改一個 String 或 Vec 呢？

5.25.2.1 RefCell：執行期的借用檢查

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，T 必須 Copy，**零成本**（編譯後和直接存取沒有差別）
- RefCell<T>：用 .borrow() 和 .borrow_mut() 取得參考，T **不需要** Copy，但**有執行期成本**（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> 可以共享資料，但不能改。RefCell<T> 可以改，但不能共享。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]));
}
```

這樣多個 Rc 可以共享同一份資料，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;
```

```rust
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移到執行時
- `.borrow()` 取得不可變參考，`.borrow_mut()` 取得可變參考
- `T` **不需要** `Copy`（和 `Cell` 的差別）
- `Cell` 是零成本的，但 `RefCell` 每次借用都有執行期檢查的開銷
- 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 是取 s1 和 s2 的交集，s2 已經死了
}
```

`'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）：

```rust
fn first_char(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

```rust
// 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

上一集學了生命週期標注，你可能會擔心：「每個有參考的函數都要寫 `'a` 嗎？好麻煩！」

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
- 多個參考參數且回傳參考 → 要寫

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`
- 有多個 input lifetime 且回傳型別有 lifetime 時，才需要手動標注

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，但因為它和 Word 是同一個 enum，建立 Token::Number(42) 時仍然需要指定 'a——只是這個 'a 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
}
```

注意這裡不能直接寫 Excerpt 不加任何東西——Excerpt 有一個必要的生命週期參數，就像 Vec 有一個必要的型別參數一樣，不能省略。但我們可以用 '_ 讓編譯器推斷。

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` 一定有 `Display` 的功能。

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)]` 要同時寫兩個——只寫 `derive(Copy)` 會報錯，因為 `Copy` 要求 `Clone`。

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
trait Summarize: Display {
    fn summary(&self) -> String {
        // 因為有 Display supertrait，可以用 to_string()
```

```
        let full = self.to_string();
        if full.len() > 10 {
            let mut s = String::new();
            // 取前 10 個字元
            let mut count = 0;
            for c in full.chars() {
                if count >= 10 {
                    break;
                }
                s.push(c);
                count += 1;
            }
            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);

    // 用 Summarize（預設實作會用到 Display）
    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`——Copy 要求 Clone，所以 derive 時必須同時寫兩個
- `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 的預設實作裡可以使用 supertrait 的方法

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 是 PartialEq 的 supertrait（上一集學的），它保證**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 和任何值比較都會回傳 false——包括它自己：

```
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，因為它和誰比結果都是 false，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

```
#[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` 決定了解參考後得到什麼型別。例如 `Box<T>` 的實作是 `type Target = T`——解參考 `Box<i32>` 得到 `i32`。這跟 `Container` 的 `type Item` 是同樣的道理：一個 `Box<i32>` 解參考後只會得到 `i32`，不會得到別的東西，所以用 associated type 而不是泛型參數。

5.32.2.5 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`」。

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()
    }
}
```

```rust
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`，代表解參考後得到 `T`
- 在 `trait` bound 中用 `Container<Item = i32>` 指定 associated type

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**（寫入時才複製）。它定義在 `std::borrow` 模組裡。來看它的定義（省略了一些我們還沒學的部分）：

```
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 的所有方法、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「寫入時才複製」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：寫入時才複製
    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`，`Cow<'a, str>` 不管是 `Borrowed` 還是 `Owned` 都能直接當 `&str` 用——這是它最大的優點
- `to_mut()`：寫入時才複製（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。但它有一個限制——函數體沒辦法使用呼叫處的區域變數。下一集會介紹閉包（closure），它能做到這件事。

6.1.3 範例程式碼

```rust
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
- 函數指標的限制：沒辦法使用呼叫處的區域變數。下一集的閉包能做到這件事

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，但有個限制：它不能使用呼叫處的區域變數。閉包（closure）就是為了解決這個問題而存在的。

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：

```rust
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：

```rust
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：

```rust
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

透過手動把閉包拆解成 `struct` + 方法，理解編譯器在背後做了什麼事。你會看到三種閉包各自對應什麼樣的 `struct`，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`，可變借用 `struct`
- **`Fn`**：`f()` → `f.call()` — 傳 `&self`，唯讀借用 `struct`

看出來了嗎？這就是第 4 章學的三種方法接收者：`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：只讀取捕捉的值，不消耗也不修改，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：**三種閉包的 struct 欄位型別不同**，不是同一個 struct 換三種方法。

6.3.2.3 FnOnce：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：struct 存可變借用，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, // 可變借用 name
}

// 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，所以只能借用整個 struct。

6.3.2.5 Fn：struct 存唯讀借用，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, // 唯讀借用 name
}

// let greet = GreetRef { name: &name };

impl<'a> GreetRef<'a> {
    fn call_ref(&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	struct 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	可變借用（如 &mut String）	修改捕捉的值，可以多次呼叫
&self	Fn	唯讀借用（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式。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

以下的完整程式碼把三種閉包都手動模擬出來。每一個 struct 對應一種閉包，欄位型別和方法接收者都不同：

```
// === 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 存可變借用，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 存唯讀借用，方法接 &self
struct GreetRef<'a> {
    name: &'a String,
}

impl<'a> GreetRef<'a> {
    fn call_ref(&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，借用結束
    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_ref();
    greet_ref.call_ref();
    greet_ref.call_ref();
}
```

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：只讀取，不修改不消耗，可以無限次呼叫
- 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，不會跳出外面的函式——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 概念說明

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. **如果閉包體裡修改了捕捉的變數**（例如 `count += 1;`）→ 這個閉包是 **FnMut**，可以多次呼叫但需要 &mut
3. **如果閉包體只讀取捕捉的變數**（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**——如果只讀取，就給 Fn（因為 Fn 的閉包也能當 FnMut 和 FnOnce 用）。如果有修改，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、修改 → FnMut、只讀 → 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 Fn / FnMut / FnOnce

6.5.1 本集目標

理解 Fn、FnMut、FnOnce 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**。就像第 5 章學的 Clone、Display 一樣，Fn / FnMut / FnOnce 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：可以被呼叫至少一次（可能會消耗自己）
- FnMut(Args) -> Ret：可以被多次呼叫（可能會修改內部狀態）
- Fn(Args) -> Ret：可以被多次呼叫（不會修改任何東西）

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，那給它 &mut self 當然也行（只是多給了它不需要的修改權限）
- **FnMut → FnOnce**：如果一個閉包用 &mut self 就能執行，那給它 self（整個擁有權）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_readonly(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且不允許修改

為什麼？因為 FnOnce 能接受所有閉包（所有閉包都至少是 FnOnce），而 Fn 只能接受不修改狀態的閉包。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）。只有少數場景需要**保證閉包不修改狀態**時才會用 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

```rust
// 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();
}

// 需要多次呼叫且不修改 → 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（因為 Fn: FnMut: 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

- Fn、FnMut、FnOnce 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，Fn 只接受不修改的）
- 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，需要保證不修改才用 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 move 方式捕捉外部變數，理解它為什麼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！
}
```

為什麼錯？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：「不要用借用，把所有捕捉的變數都**搬進**閉包裡。」這樣 name 就歸閉包所有了，不管原本的作用域怎麼結束，閉包都能繼續用 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 的欄位可能是參考（&T 或 &mut T）；加了 move 之後，**所有欄位都變成擁有所有權的值**（T）：

```
    // 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
```

因為所有欄位都是擁有所有權的，這個 struct 不借用任何東西，所以沒有 lifetime 的問題——可以安全地從函數回傳、存進 struct。

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**：

```
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

閉包能不能 clone 或 copy，取決於它捕捉的變數——跟 tuple 類似，如果裡面的東西都能 copy，整體就能 copy：

- 所有捕捉的變數都是 Copy → 閉包也是 Copy
- 所有捕捉的變數都是 Clone → 閉包也是 Clone
- 其他某些 trait 也是同理

```
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` 強制閉包獲得所有捕捉變數的所有權，不依賴外部借用，適合需要長壽命的場景
- 回傳閉包時通常需要 `move`，避免懸垂參考

- move **不影響**閉包是 Fn / FnMut / FnOnce——那取決於閉包**怎麼使用**捕捉的值
- 閉包能否 clone / copy 取決於捕捉的變數是否全為 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().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，理解它背後其實是 IntoIterator + while let 的組合。

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：

```
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 迴圈展開

上面的 for 迴圈，編譯器其實會轉換成這樣：

```
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」：

```
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：

```
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

```
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);
    }

    // 手動展開成 while let（完全等價）
    let fruits = vec!["蘋果", "香蕉", "橘子"];
    println!("\n--- 手動展開 ---");
    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，展開後是 `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 `iter` / `into_iter` / `iter_mut`

6.10.1 本集目標

搞懂三種迭代模式的差別——借用、消耗、可變借用——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。今天來正式介紹 `Vec` 提供的三個方法：

方法	產出型別	語意	`Vec` 之後還能用嗎？
`.iter()`	`&T`	借用每個元素	✓ 可以
`.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）

6.10.2.2 .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。

6.10.2.3 .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()。

6.10.2.4 .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 &mut T，讓你可以原地修改每個元素。

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（共享借用）	.iter()	for x in &v
T（移動所有權）	.into_iter()	for x in v
&mut T（可變借用）	.iter_mut()	for x in &mut 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>、&Vec<T>、&mut 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 Vec<T> {
    type Item = &'a T;
    fn into_iter(self) -> ... { /* 等同於 .iter()，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 */ }
}
```

所以 `for x in &v` 其實是對 `&v`（型別是 `&Vec<T>`）呼叫 `into_iter()`，走到 `&Vec<T>` 的那個 impl，最終拿到 `&T`。

大部分集合型別（`Vec`、`String`、陣列等）都遵循這個模式——為自己、`&self`、`&mut self` 三種各實作一次 `IntoIterator`。

6.10.2.7 選哪一個？

- 只需要讀取 → `.iter()`（最常用）
- 需要拿走元素的所有權 → `.into_iter()`
- 需要原地修改 → `.iter_mut()`

原則就是所有權的原則：不要拿你不需要的權限。

6.10.3 範例程式碼

```
fn main() {
    // .iter() ——只讀借用
    let animals = vec![
        String::from("貓"),
        String::from("狗"),
        String::from("兔子"),
    ];

    println!("--- .iter()（借用） ---");
    for animal in animals.iter() {
        println!("動物：{}", animal);
    }
    println!("animals 還在：{:?}", animals);

    // .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);

    // .into_iter() ——消耗所有權
    let words = vec![
        String::from("hello"),
        String::from("world"),
    ];
```

```
    println!("\n--- .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

    // 簡寫版的對應
    println!("\n--- 簡寫版 ---");
    let nums = vec![1, 2, 3];

    // for x in &nums 等於 for x in nums.iter()
    for x in &nums {
        print!("{} ", x);
    }
    println!("← &nums（借用）");

    // for x in nums 等於 for x in nums.into_iter()
    for x in nums {
        print!("{} ", x);
    }
    println!("← nums（消耗）");
    // nums 已經不能用了
}
```

6.10.4 重點整理

- `.iter()` 產出 `&T`，借用元素，集合不受影響
- `.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
- `for x in &v` = `.iter()`，`for x in v` = `.into_iter()`，`for x in &mut v` = `.iter_mut()`
- 選擇原則：不要拿超過需要的權限——只讀就 `.iter()`，要改就 `.iter_mut()`，要消耗就 `.into_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 evens: Vec<&i32> = vec![1, 2, 3, 4, 5].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 使用社群套件。

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套件。

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 Rust 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套件

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 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 加入外部套件

想用別人寫好的套件？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：

```toml
[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 的官方套件庫。你可以在上面搜尋套件、看下載數、閱讀文件。每個套件頁面都會有：

- 使用說明和版本歷史
- 連結到 docs.rs 的自動產生文件
- 下載數（可以當作套件熱門程度的參考）

7.1.2.5 依賴的版本語意

在 `[dependencies]` 裡指定外部套件的版本時，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

有些套件提供可選功能，用 `features` 開啟：

```toml
[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` 套件產生隨機數：

```rust
// 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]` 記外部套件
- `edition` 是 Rust 語言規格的版本（2015、2018、2021、2024），不同 edition 的 crate 可以互相搭配
- `cargo add <套件名>` 是加入外部套件最快的方式
- crates.io 是 Rust 的官方套件庫，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` 可以開啟套件的可選功能

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。

7.2.2.3 mod 的預設可見性

一個很重要的觀念：**mod 裡的所有項目預設都是私有的**。同一個 mod 內部的程式碼可以互相存取，但外部看不到。這是 Rust 用來保護封裝性的設計。我們第 4 集會深入探討。

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 可以巢狀，路徑就越來越長：`a::b::c::func()`
- mod 內的所有項目**預設是私有的**，要讓外部使用需加 `pub`
- 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
}
```

```rust
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（crate）有兩種類型：

- **binary crate**：有 `src/main.rs`，會編譯成可執行檔
- **library crate**：有 `src/lib.rs`，給別人使用的程式庫

一個專案可以**同時**有 `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;

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
- 一個專案可以同時是 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 概念說明

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：**trait 的方法只有在 trait 被 use 進來之後才能呼叫。**

```
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：

```
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)`：只有父 `mod` 可見
- `pub(in crate::some::path)`：只對指定的 `mod` 路徑可見——最精細的控制

```
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")
    }

    // 注意 queries 本身是 pub——如果這個 mod 不是 pub，
    // 那裡面的東西即使標了 pub(super) 也沒用，
    // 因為外面根本看不到這個 mod，更別說裡面的東西了。
    pub mod queries {
        // 只有 database mod（父 mod）能看到
        pub(super) fn raw_query() -> String {
            String::from("SELECT * FROM users")
        }

        pub fn safe_query() -> String {
            let raw = raw_query(); // 同 mod 內可以呼叫
            format!("SAFE: {}", raw)
        }
    }
```

```rust
}

// pub(in path) 的例子
mod app {
    pub mod api {
        pub mod internal {
            // 只有 app::api 能看到這個函數
            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) 限制了只有 app::api 能存取

// 注意：pub(in path) 的 path 必須是「包含你的」mod（從你往外數的某一層）。
// 如果你寫了一個跟你無關的 mod 路徑，例如：
//     pub(in crate::some_unrelated_mod) fn foo() {}
// 編譯器會直接報錯——你不能對一個「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::queries::safe_query(); // OK，pub
    println!("{}, {}", conn, q);
    database::queries::raw_query();          // 編譯錯誤！pub(super) 只給父 mod
}
```

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
- `pub(crate)`：crate 內部可見，外部不可見
- `pub(super)`：只有父 `mod` 可見
- `pub(in path)`：只對指定的 `mod` 路徑可見

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::Rng;
```

std 是 Rust 的**標準函式庫（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::Rng;  // 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）

```rust
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：

```rust
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：

```rust
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：

```rust
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：

```rust
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 mod 底下所有 pub 的東西全部帶進來：

```rust
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：

```rust
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 是標準函式庫，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：

```rust
#[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`：

```rust
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` 是你程式碼的子 `mod`，而同一個 `mod` 裡的東西互相看得到——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

```rust
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::Rng; // 使用者 use your_crate::Rng 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` 是標準函式庫定義的，`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. 從所有 `///` 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：

```
//! # 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、使用別人的套件。這一集要反過來——把你自己的套件發布出去。

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！你的套件現在在 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

套件發布後，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

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 套件頁面上）
- 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
///
```

```rust
/// ```
/// 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）
- 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，以及模組系統和套件管理。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任意位置插入底線 `_`，編譯器會直接忽略它們：

```
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任意位置，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`）
- 浮點數字面值必須有小數點或科學記號

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 為什麼只有 `loop` 能這樣做？

你可能會問：`while` 和 `for` 為什麼不行？

原因是：`while` 和 `for` 有可能**一次都不執行**。如果迴圈體從未執行，那 `break` 帶出的值根本不存在，編譯器就無法保證一定有回傳值。

但 `loop` 不同——它是無條件迴圈，**一定會進入迴圈體**，所以編譯器可以確定 `break` 一定會被執行到（否則就是無窮迴圈）。這就是為什麼只有 `loop` 能當表達式回傳值。

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 重點整理

- `let x = loop { break value; };` 讓 `loop` 成為表達式，回傳 `break` 帶出的值

- 只有 `loop` 能這樣做，`while` 和 `for` 不行——因為它們可能一次都不執行
- 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"##;
```

你可以加任意多層 `#`，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
- # 的層數可以增加（r##"..."##、r###"..."###），只要前後一致
- 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、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：

```
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）：

```
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：

```
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：

```
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：

```rust
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：

```rust
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 `}}`：

```rust
fn main() {
    println!("這是大括號：{{}}"); // 印出：這是大括號：{}
}
```

8.5.3 範例程式碼

```rust
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` 看得到。其他函數無法使用它。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**。你可以先使用，後定義：

```
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 color = Color::Red;
    describe(color);

    // 定義放在使用之後
    struct Point {
        x: f64,
        y: f64,
    }

    let p = Point { x: 3.0, y: 4.0 };
    println!("座標：({}, {})", p.x, p.y);

    // 這些項目定義的順序完全不重要
    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 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**——可以先使用、後定義
- **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：項目是編譯期的靜態定義，編譯器會先掃描完所有項目再處理執行期程式碼

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` 並且移動了 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 `..Default::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, ..Default::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,
        ..Default::default()
    };
    println!("自訂設定：{:?}", custom);

    // 全部用預設值
    let default_config = Config { ..Default::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

- `..Default::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**。編譯器看到你 `match` 的是一個參考（`&opt`），就會自動幫你在模

式裡加上 ref。

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

認識四種會讓程式立即終止的巨集，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「讓程式直接掛掉」的巨集。它們都會造成 panic（程式中止），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，直接中止」。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! —直接中止
    // panic!("故意 panic！");
    println!("程式正常結束");
}
```

8.9.4 重點整理

- `panic!("msg")` 是最基本的中止方式，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。

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**：

```
// Iterator 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,
}

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
- 型別本身的方法優先於 `trait` 方法
- 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

這些指標是所謂的**胖指標（fat pointer）**——它們不只存一個位址，還多存了一個長度資訊：

```
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 8 集介紹的 `Clone` 嗎？它的方法是 `fn clone(&self) -> Self`——直接回傳 `Self`。由於 `Self` 預設可能不是 `Sized`，而回傳的型別必須有已知大小，所以 `Clone` 實際上的定義是：

```
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。想想 &'static mut str：它指向的是程式檔案裡的唯讀區段，你不可能讓那塊記憶體變大。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`，它們解參考得到的正是 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]`

也就是說解參考 `String` 得到的是 `str`，解參考 `Vec<T>` 得到的是 `[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

```
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 DST 的指標是**胖指標（fat pointer）**：位址 + 長度，在 64 位元電腦上佔 16 bytes
- **`Sized`**：表示型別大小在編譯期已知；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（但必須透過參考使用）
- `trait` 裡的 `Self` 預設是 `?Sized`；如果方法需要回傳 `Self`，要在 `trait` 上加 `: Sized`（如 `Clone: Sized`）
- `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` 分別得到 DST `str` 和 `[T]`，deref coercion 讓 `&String` → `&str`、`&Vec<T>` → `&[T]` 成為可能

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。

附錄一最後一集曾經提過 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。

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
}
```

&T 和 Box<T> 大小一樣——因為它們存的都是位址。&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）

附錄一最後一集介紹過 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，我們的程式都是從頭到尾一行一行執行的。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。

```
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：

```
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：

```
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：

```
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 thread::spawn 不只能在 main 裡呼叫——任何函數都可以 spawn 執行緒。新執行緒的生命週期不確定，它可能活得比呼叫它的函數還久。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（不可變參考）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

你不需要手動實作 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：閉包和回傳值裡的所有東西都不能有會過期

的借用。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就是上一集說的 move——把值搬進閉包之後，閉包不借用任何東西，自然滿足 'static。

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
- 編譯器自動推導 Send / Sync，不用手動標記

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（不可變參考）修改內部的值。在單執行緒的世界裡，RefCell 會在執行期檢查借用規則，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 的意思是「不可分割」，一個 atomic 操作要嘛完整執行完，要嘛完全沒發生，不會被其他執行緒打斷到一半。`RefCell` 的計數器讀取和寫入不是 atomic 的，代表一個執行緒讀到一半，另一個執行緒可能就插進來改了值。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 `RefCell` 是 Send

但 `RefCell` 可以被 **move** 到另一個執行緒。為什麼？因為 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 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 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，要嘛完整執行完，要嘛完全沒發生，不會被其他執行緒打斷到一半
- `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>`
- `RefCell` 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); // 增加計數，不複製資料
    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 T 必須是 Send + Sync

Arc 要求 T: Send + Sync。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 的不可變參考，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

```
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
- `T` 必須是 `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。

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：

```rust
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

```rust
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`：

```rust
// 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。用 `SeqCst` 就不會有這個問題，它保證所有執行緒看到的操作順序一致。

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，所有執行緒看到的操作順序都一致

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};
```

```rust
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，目前值：{}", 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

- 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("取得鎖失敗") 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("取得鎖失敗");
        *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("取得鎖失敗");
            *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("取得鎖失敗")); // 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("取得鎖失敗");
```

```
*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("取得鎖失敗");
    *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("取得鎖失敗");
                *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("取得鎖失敗"));
}
```

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

```
use std::sync::RwLock;

fn main() {
    let lock = RwLock::new(42);

    // 多個讀者可以同時讀
    {
        let r1 = lock.read().expect("取得讀鎖失敗");
        let r2 = lock.read().expect("取得讀鎖失敗"); // OK，可以同時持有多個讀鎖
        println!("r1 = {}, r2 = {}", *r1, *r2);
    }

    // 寫入時獨佔
    {
        let mut w = lock.write().expect("取得寫鎖失敗");
        *w += 1;
    }
```

```
}
```

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()	多個 read() 或一個 write()
檢查方式	執行期，違反會 panic	作業系統的鎖，違反會阻塞等待

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

```rust
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("取得讀鎖失敗");
            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("取得寫鎖失敗");
            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("取得讀鎖失敗"));
}
```

9.9.4 重點整理

- RwLock<T> 區分讀鎖和寫鎖：多個讀者可以同時讀，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
- 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("取得鎖失敗")。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，不管是 lock、read 還是 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("取得鎖失敗");
        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("取得鎖失敗");
}
```

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。

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
            g
        }
    };
}
```

拿到 guard 之後把資料修復成合理的值，然後繼續用。

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("取得鎖失敗");
        *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

- 持有鎖的執行緒 panic 了 → 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

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("取得鎖失敗"); // 拿到鎖 1
        // 假設這裡有一些延遲...
        let _g2 = l2.lock().expect("取得鎖失敗"); // 等待鎖 2
    });

    let l1 = Arc::clone(&lock1);
    let l2 = Arc::clone(&lock2);

    let b = thread::spawn(move || {
        let _g2 = l2.lock().expect("取得鎖失敗"); // 拿到鎖 2
        // 假設這裡有一些延遲...
        let _g1 = l1.lock().expect("取得鎖失敗"); // 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` 就永遠等不到：

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(42);
    let _g1 = m.lock().expect("取得鎖失敗");
    let _g2 = m.lock().expect("取得鎖失敗"); // 可能死鎖！第一個鎖還沒放，第二次 lock 永遠等不到
}
```

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("取得鎖失敗"); // 先鎖 1
        let g2 = l2.lock().expect("取得鎖失敗"); // 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("取得鎖失敗"); // 也是先鎖 1
        let g2 = l2.lock().expect("取得鎖失敗"); // 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
- 同一個執行緒對同一個 Mutex lock 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：

```rust
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：

```rust
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：

```rust
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

```rust
use std::thread;
```

```rust
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 thread 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，編譯器在編譯期不知道會是哪個。所以 dyn Describe 是 DST（附錄一最後一集學過），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，編譯器在編譯期無法決定是哪一個。

10.1.2.3 胖指標：位址 + vtable

附錄一最後一集學過 &[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。

```rust
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。

```rust
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	編譯期	執行期
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：

```rust
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？除了回傳 `Self`，其實 `Clone` 有一個 supertrait 就是 `Sized`：

```
trait Clone: Sized {
    fn clone(&self) -> Self;
}
```

`Clone: Sized` 代表「實作 Clone 的型別必須是 Sized」。但 `dyn Clone` 是 DST，不是 `Sized`。所以 `impl Clone for dyn Clone` 根本不成立，`dyn Clone` 無法存在。

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期執行
```

10.3.2.2 const fn

在函數前面加上 const，它就變成編譯期也能執行的函數：

```
const fn square(x: i32) -> i32 { x * x }
const VALUE: i32 = square(5); // OK！編譯期算出 25
```

const fn 不是「只能在編譯期用」——它在執行期也能正常呼叫，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); // 編譯期：20

fn main() {
    let x = max(3, 7); // 執行期：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、for
- let 綁定（包括 let mut）
- 建立 tuple、struct、enum
- 呼叫其他 const fn
- panic!（編譯期 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，在編譯期就算好了
}
```

這在需要「就地」做編譯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);
```

```rust
fn main() {
    println!("10! = {}", FACT_10);
    println!("clamp(150, 0, 100) = {}", CLAMPED);

    // 執行期也能呼叫
    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` 可以在編譯期執行，也可以在執行期執行
- 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：

```rust
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：

```rust
trait HasLimit {
    const LIMIT: i32;
}

impl HasLimit for u8 {
    const LIMIT: i32 = 255;
}
```

```
impl HasLimit for i8 {
    const LIMIT: i32 = 127;
}
fn main() {
    println!("u8：{}", <u8 as HasLimit>::LIMIT); // 255
    println!("i8：{}", <i8 as HasLimit>::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：

```
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：

```
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

```
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: [i32; { 2 + 3 }] = filled(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

學會為泛型參數和 const generics 設定預設值。

10.6.2 概念說明

10.6.2.1 泛型的預設型別參數

有時候某個泛型參數「大部分情況都是同一個值」。Rust 允許你給預設值——不指定就自動套用。

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

```
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

- 泛型參數可以有預設值：`<T = String>`、`<Rhs = Self>`
- const generics 也可以：`<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` 直接追加內容，`s1 = s1 + &s2` 先消耗 s1 再建新的。效率和語意不同，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 }

impl Add for Vec2 {
    type Output = Vec2;
    fn add(self, rhs: Vec2) -> Vec2 {
        Vec2 { x: self.x + rhs.x, y: self.y + rhs.y }
    }
}
```

```rust
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：

```rust
    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。它們只在轉換保證不會失敗的時候才存在：

```
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！不保證安全
}
```

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

- 能用 From / Into 就用——編譯期保證安全
- 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：只存在於保證安全的轉換
- 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
}
```

```rust
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：

```rust
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：

```rust
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：

```rust
#[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**：

```rust
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

```rust
#[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

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
struct Color(u8, u8, u8);
```

10.10.2.3 警告控制

```
#[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

```
#[cfg(target_os = "windows")]
fn windows_only() { /* ... */ }

#[cfg(test)]
mod tests { /* ... */ }
```

10.10.2.5 測試

```
#[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。

```
#[inline]         // 建議編譯器 inline 這個函數
#[inline(always)] // 強制 inline
#[inline(never)]  // 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!` macro

10.11.1　本集目標

學會用 `cfg!` 在執行期根據條件選擇邏輯，以及跟 `#[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	回傳 `bool`
時機	不符合的程式碼消失，不會被編譯	兩邊都會被編譯，執行時選邊

重要差別：`#[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 `cfg!` 兩邊都會編譯——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!(...) 回傳 bool，兩邊程式碼都會被編譯，執行時選擇
- #[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 只要引入 `my_lib`，就能直接用 `log_msg!("hello")`。`$crate` 會自動替換成正確的 crate 路徑。

10.12.3 範例程式碼

```
macro_rules! max {
    ($a:expr, $b:expr) => {
        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);

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。

```rust
#[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)]
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
```

```rust
}

// 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 i32 的 bytes 當成 f32 來讀）

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——編譯器自動推導的時候沒問題，但如果你手動實作（覆蓋自動推導），你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）
- 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;
```

	const	static
記憶體	沒有固定位址，值嵌進使用的地方	有固定位址，整個程式共用一份
取位址	不能取 &	可以取 &，保證永遠合法

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 的值必須編譯期確定

static 的值必須在編譯期就算出來。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

```
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` 的值必須編譯期確定，但 `Vec` / `String` 等做不到
- `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：

```
unsafe extern "C" {
    fn abs(x: i32) -> i32;
}

fn main() {
    let result = unsafe { abs(-42) };
    println!("abs(-42) = {}", 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`：

```rust
unsafe extern "C" {
    safe fn abs(x: i32) -> i32; // 你保證 abs 一定安全
}

fn main() {
    let result = abs(-42); // 不需要 unsafe 也能呼叫！
    println!("abs(-42) = {}", 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

```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

```rust
unsafe extern "C" {
    static errno: i32; // C 那邊的全域變數
}
```

10.17.3 範例程式碼

```rust
unsafe extern "C" {
    safe fn abs(x: i32) -> i32;
    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!("abs(-10) = {}", abs(-10));

    // 沒標記 safe 的需要 unsafe
    let root = unsafe { sqrt(25.0) };
    println!("sqrt(25) = {}", 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

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 的大小是最大欄位的大小（4 bytes）。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
大小	最大 variant + discriminant	最大欄位（沒有額外開銷）

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！
    }

    // union 的大小 = 最大欄位的大小
    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!("...")` — 程式崩潰
- `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。

你其實一直在用這個特性：

```
    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) -> ! {
    eprintln!("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() {
    print_length("hello");            // &str
    print_length(String::from("hi")); // String
}
```

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：

```rust
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　為什麼 AsRef 用 `impl AsRef<T>` 但 AsMut 用 `&mut impl AsMut<T>`？

AsRef 的 as_ref 只需要 &self，所以把值傳進來完全沒問題——函數內部借用一下就好，呼叫端的值不受影響（如果是 Copy 的話），或者你本來就打算把值 move 進來。

但 AsMut 不一樣。如果你寫 `fn foo(buf: impl AsMut<[u8]>)`，值會被 move 進來——呼叫端用完就沒了。你都特地傳 &mut 了，就是想改完之後繼續用，所以參數寫成 `&mut impl AsMut<[u8]>`，借用它的可變參考就好，不需要 move。

那你可能會想：「`&mut Vec<u8>` 又不是 `Vec<u8>`，為什麼 `&mut Vec<u8>` 能當成 `AsMut<[u8]>` 用？」答案是標準庫有這個 blanket implementation：

```rust
impl<T, U> AsMut<U> for &mut T
where
    T: AsMut<U> + ?Sized,
    U: ?Sized,
{ ... }
```

意思是：如果 T 實作了 `AsMut<U>`，那 `&mut T` 也自動實作 `AsMut<U>`。所以 `&mut Vec<u8>` 能直接當 `AsMut<[u8]>` 用。

11.1.2.5　跟 Deref 的差別

Deref 是自動的——編譯器幫你加 *，你不用寫任何東西。AsRef 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 deref 成 str），但可以實作多個 AsRef（String 同時是 `AsRef<str>` 和 `AsRef<[u8]>`）。AsMut 同理。

11.1.2.6　什麼時候用

寫函數參數想泛化接受多種型別的時候，用 `impl AsRef<T>` 和 `&mut impl AsMut<T>`。標準庫到處都在用。

11.1.3　範例程式碼

```rust
fn describe(s: impl AsRef<str>) {
    let s = s.as_ref();
    println!("「{}」有 {} 個字元", s, s.chars().count());
}

fn count_bytes(data: impl AsRef<[u8]>) {
    println!("共 {} bytes", data.as_ref().len());
}
```

```rust
fn main() {
    // AsRef<str>：接受 &str 和 String
    describe("hello");
    describe(String::from("你好"));

    // AsRef<[u8]>：接受 Vec<u8>、String 等
    count_bytes(vec![1, 2, 3]);
    count_bytes(String::from("hi"));
}
```

11.1.4 重點整理

- AsRef<T>：便宜地借用成 &T，用 .as_ref() 呼叫
- AsMut<T>：便宜地借用成 &mut T，用 .as_mut() 呼叫
- AsRef 參數用 impl AsRef<T>（傳值），AsMut 參數用 &mut impl AsMut<T>（借用，改完呼叫端還能繼續用）
- &mut T 之所以能當 impl AsMut<U> 用，是因為標準庫的 blanket implementation
- 一個型別可以實作多個 AsRef / AsMut（Deref / DerefMut 只能一個目標）
- 標準庫大量使用

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：

```rust
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：

```rust
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 和任何值比較都是 false），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); // 3.0
}
```

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：

```
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：

```
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：

```
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：

```
pub struct Reverse<T>(pub T);
```

它的 `Ord` 實作把比較順序反了過來：

```
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 函數把 key 對應到記憶體位置，不管裡面有多少資料，查一個 key 的速度幾乎是固定的。

11.3.2.2 建立與基本操作

```
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

```
use std::collections::HashMap;
```

```rust
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。它的做法是把 key 丟進一個 **hash 函數**，算出一個數字（hash value），用這個數字決定值放在記憶體的哪個位置。之後要查的時候，再對 key 算一次 hash，就能直接跳到那個位置，不用一個一個找。

所以 key 的型別必須實作 Hash trait——告訴 HashMap 怎麼對這個型別算 hash。

11.3.2.6 Key 的要求：Eq + Hash

Key 除了要 Hash，還要 Eq。因為不同的 key 可能被分到同一個位置，HashMap 需要用 == 來確認找到的確實是你要的 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 Eq（因為 NAN），所以不能當 key。

11.3.2.7 幫自己的型別實作 Hash

Hash 可以 derive：

```rust
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

```
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> 用 key 查 value，不管資料量多大查詢速度幾乎是固定的
- 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().or_insert() 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是固定的：

```
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`：有序的 `HashMap`，key 必須實作 `Ord`
- `BTreeSet`：有序的 `HashSet`，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") {
        println!("PATH 的前 50 個字元：{}", &path[..path.len().min(50)]);
    }
}
```

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

寫程式常常要跟檔案打交道——讀設定檔、寫 log、處理使用者指定的路徑。之後會學怎麼讀寫檔案，但在那之前，我們得先知道怎麼表示「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

```
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`：

```
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

```
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

```
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 輸入輸出：`stdin`、檔案讀寫

11.9.1 本集目標

認識 Rust 的輸入輸出方法，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，標準庫裡很多輸入輸出函數都用它。

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 抽象化輸入輸出：

- 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 是輸入輸出的核心 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(e) => write!(f, "輸入輸出錯誤：{}", e),
            AppError::Parse(e) => write!(f, "解析錯誤：{}", e),
        }
    }
}

impl std::error::Error for AppError {}
```

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>

如果你不需要精確區分錯誤種類，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)
}
```

任何實作了 Error 的型別都能自動轉成 Box<dyn Error>，所以 ? 直接就能用，不需要手動寫 From。

缺點是呼叫者沒辦法用 match 精確處理不同的錯誤種類——它只知道「有個錯誤」，但不知道具體是哪種。

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 → 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>：通用錯誤型別，任何 Error 都能自動轉換，? 直接能用
- 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 macro 自動生成 Display、Error、From：

```rust
use thiserror::Error;

#[derive(Debug, Error)]
enum AppError {
    #[error("輸入輸出錯誤：{0}")]
    Io(#[from] std::io::Error),

    #[error("解析錯誤：{0}")]
    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
- 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>，但更好用
- .context("...") 幫錯誤加上額外說明，方便除錯
- 不用定義任何錯誤型別，任何 Error 都能自動轉換

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**：完全不用定義錯誤型別，所有錯誤統一處理。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 macro 自動生成 Display、Error、From，適合函式庫
- #[error("...")] 生成 Display，#[from] 生成 From
- anyhow：通用錯誤型別，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。

11.12.2 概念說明

11.12.2.1 動機

正常情況下 panic 會讓當前的執行緒直接中止。但在 **FFI 邊界**，panic 不能往上傳到 C 那邊，否則是未定義行為（進階語言功能那章提過 FFI）。catch_unwind 讓你在 Rust 這邊攔住 panic，不讓它跨越語言邊界。

11.12.2.2 基本用法

```rust
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(...)
}
```

catch_unwind 接受一個閉包，如果閉包正常回傳就得到 Ok(值)，如果 panic 了就得到 Err。

11.12.2.3 UnwindSafe

catch_unwind 要求閉包是 UnwindSafe 的。為什麼？因為 panic 的時候，閉包裡的操作可能做到一半，資料處於不一致的狀態——跟多執行緒那章 poisoning 的道理一樣。

&mut T 不是 UnwindSafe：如果你透過 &mut 修改資料修到一半 panic 了，catch 之後那份資料可能是半成品。&T、i32 等不可變的東西是 UnwindSafe 的。

11.12.2.4 AssertUnwindSafe

如果你確定沒問題，可以用 AssertUnwindSafe 包起來繞過檢查：

```
use std::panic::{catch_unwind, AssertUnwindSafe};

fn main() {
    let mut data = vec![1, 2, 3];
    let result = catch_unwind(AssertUnwindSafe(|| {
        data.push(4);
    }));
}
```

這跟 poisoning 或 unsafe 的精神類似——你自己負責保證正確性。

11.12.2.5 panic = "abort"

Cargo.toml 可以設定 panic = "abort"，這樣 panic 會直接終止程式，不會執行任何清理工作（包括 drop）。在這個設定下 catch_unwind 沒有用——panic 就是直接結束，沒有東西可以 catch。

```
[profile.release]
panic = "abort"
```

11.12.2.6 注意

catch_unwind 不是用來做一般的錯誤處理的——那是 Result 的工作。catch_unwind 只用在上面提到的那些特殊場景。

11.12.3 範例程式碼

```
use std::panic;

fn might_fail(x: i32) -> i32 {
    if x == 0 {
        panic!("不能是零！");
    }
    100 / x
}

fn main() {
    let inputs = vec![10, 5, 0, 2];

    for x in inputs {
        let result = panic::catch_unwind(|| might_fail(x));
        match result {
            Ok(val) => println!("100 / {} = {}", x, val),
            Err(_) => println!("處理 {} 時 panic 了，跳過", x),
        }
    }

    println!("程式繼續執行");
}
```

11.12.4 重點整理

- catch_unwind 攔截 panic，回傳 Ok(值) 或 Err

- 主要用途：FFI 邊界，防止 panic 跨越語言邊界
- UnwindSafe：&mut T 不是 UnwindSafe（資料可能是半成品）
- AssertUnwindSafe：手動保證安全，繞過 UnwindSafe 檢查
- panic = "abort" 設定下 catch_unwind 無效
- 不要用 catch_unwind 做一般的錯誤處理——那是 Result 的工作

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！