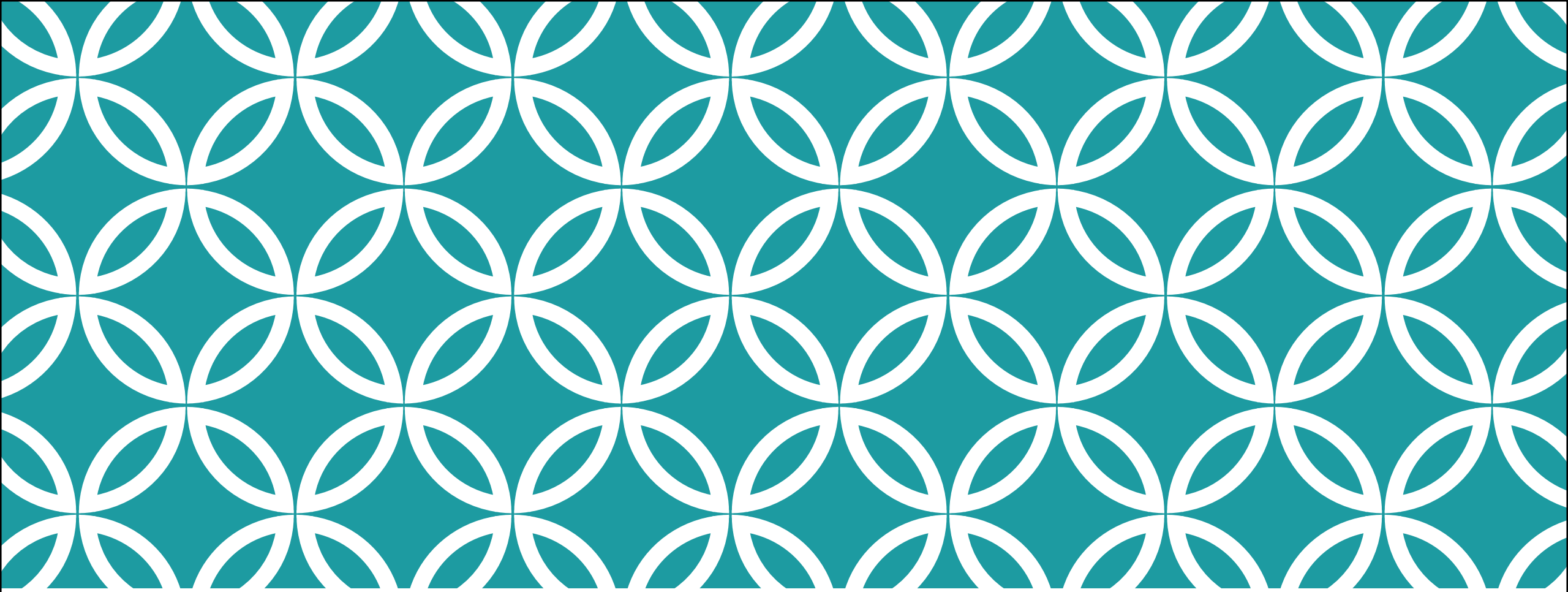


套件與 Crates

套件與 Crates

透過套件、Crate 與模組管理成長中的專案

- 當寫的程式規模更大時，組織程式碼就很重要。因為用腦袋要記住整個程式碼是幾乎不可能的。要是能組織相關功能的程式碼並將它們分成明確功能的話，就能清楚地找到實作特定功能的程式碼，以及該在哪裏修改該功能的行為。之前寫過的程式都只在一個檔案內的一個模組(module)中。
- 隨著專案成長，應該要組織程式碼，拆成數個模組與數個檔案。一個套件(package)可以包含數個執行檔 crate 以及選擇性提供一個函式庫 crate。隨著套件增長，可以取出不同的部分作為獨立的 crate，成為對外的依賴函式庫。
- 此章節將會介紹這些所有概念。對於非常龐大的專案，需要一系列的關聯套件組合在一起的話，Cargo 有提供工作空間(workspaces)，會在之後的「Cargo 工作空間」做介紹。
- 還會討論對實作細節進行封裝，讓程式碼在頂層更好使用。一旦實作了某項功能，其他程式就可以用程式碼的公開介面呼叫該程式碼，而不必去知道它實作如何運作。在寫程式碼時會去定義哪些部分是給其他程式碼公開使用的，以及哪些部分是私底下可以任意修改的實作細節。這能再減少腦袋需要煩惱的細節數量。

套件與 Crates

透過套件、Crate 與模組管理成長中的專案

- 還有一個概念需要再提一次，也就是作用域(scope)：
  - 程式碼需要能被定義在「作用域內」並要能夠指明此作用域。當讀取寫入或編譯程式碼時，程式設計師與編譯器需要知道特定地點的名稱，才能知道其內的變數、函式、結構體、列舉、常數或其他任何有意義的項目。可以建立作用域，並改變其在作用域內與作用域外的名稱。無法在同個作用域內擁有兩個相同名稱的項目。可以使用一些工具來解決名稱衝突的問題。
- Rust 有一系列的功能能讓程式碼組織，包含哪些細節能對外提供、哪些細節是私有的，以及程式每個作用域的名稱為何。這些功能有時會統一稱作模組系統(module system)，其中包含：
  - 套件(Package)：建構、測試並分享 crate 的 Cargo 功能
  - Crates： 產生函式庫或執行檔的模組集合
  - 模組(Modules)與 use：控制組織、作用域與路徑的隱私權
  - 路徑(Paths): 對一個項目的命名方式，像是一個結構體、函式或模組

套件與 Crates

透過套件、Crate 與模組管理成長中的專案

- 首先要介紹的第一個模組系統部分為套件與 crates。
- 一個 crate 是 Rust 編譯器同個時間內視為程式碼的最小單位。就算執行的是 rustc 而非 cargo，然後傳入單一原始碼檔案，編譯器會將該檔案視為一個 crate。Crate 能包含模組，而模組可以在其他檔案中定義然後同時與 crate 一起編譯，會在接下來的段落看到。
- 一個 crate 可以有兩種形式：**執行檔 crate 或函式庫 crate**。執行檔(Binary)crate 是種能編譯成執行檔並執行的程式，像是命令列程式或伺服器。這種 crate 需要有一個函式 main 來定義執行檔執行時該做什麼事。目前建立的所有 crate 都是執行檔 crate。

套件與 Crates

透過套件、Crate 與模組管理成長中的專案

- 函式庫(Library)crate 則不會有 main 函式，而且它們也不會編譯成執行檔。這種 crate 定義的功能用來分享給多重專案使用。舉例來說，在之前用到的 rand crate 就提供了產生隨機數值的功能。當大多數的程式開發者講到「crate」時，其實指的是函式庫 crate，所以講到「crate」時相當於就是在講其他程式語言概念中的「函式庫」。
- crate 的源頭會是一個原始檔案，讓 Rust 的編譯器可以作為起始點並組織 crate 模組的地方(會在「定義模組來控制作用域與隱私權」的段落詳加解釋模組)。

套件與 Crates

透過套件、Crate 與模組管理成長中的專案

- 套件(package)則是提供一系列功能的一或數個 crate。一個套件會包含一個 Cargo.toml 檔案來解釋如何建構那些 crate。Cargo 本身其實就是個套件，包含了已經用來建構程式碼的命令列工具。Cargo 套件還包含執行檔 crate 需要依賴的函式庫 crate。其他專案可以依賴 Cargo 函式庫來使用與 Cargo 命令列工具用到的相同邏輯功能。
- 一個套件能依照喜好擁有數個執行檔 crate，但最多只能有一個函式庫 crate。而一個套件至少要有一個 crate，無論是函式庫或執行檔 crate。看看當建立一個套件時發生了什麼事。首先先輸入 cargo new 命令：

```
$ cargo new my-project
     Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

套件與 Crates

```
$ cargo new my-project
     Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

透過套件、Crate 與模組管理成長中的專案

- 在執行 cargo new 之後，使用ls(dir)來查看 Cargo 建立了什麼。在專案的目錄中會有個 Cargo.toml 檔案，這是套件的設定檔。然後還會有個 src 目錄底下包含了 main.rs。透過文字編輯器打開 Cargo.toml，會發現沒有提到 src/main.rs。
- Cargo 遵循的常規是 src/main.rs 就是與套件同名的執行檔 crate 的 crate 源頭。同樣地，Cargo 也會知道如果套件目錄包含 src/lib.rs的話，則該套件就會包含與套件同名的函式庫 crate。Cargo 會將 crate 源頭檔案傳遞給 rustc 來建構函式庫或執行檔。
- 在此的套件只有包含 src/main.rs 代表它只有一個同名的執行檔 crate 叫做 my-project。**如果套件包含 src/main.rs 與 src/lib.rs 的話，它就有兩個 crate：一個執行檔與一個函式庫，兩者都與套件同名**。
- 一個套件可以有多個執行檔 crate，只要將檔案放在 src/bin 目錄底下就好，每個檔案會被視為獨立的執行檔 crate。

套件與 Crates

定義模組來控制作用域與隱私權

- 在此將討論模組以及其他模組系統的部分，像是路徑(paths)允許來命名項目，而use關鍵字可以將路徑引入作用域，再來pub關鍵字可以讓指定的項目對外公開。還會討論到as關鍵字、外部套件以及全域(glob)運算子。
- 首先先介紹一些規則好讓在之後組織程式碼時能更容易理解初步概念，再詳細解釋每個規則。

套件與 Crates

定義模組來控制作用域與隱私權

- 模組懶人包
  - 這裡先快速帶過模組、路徑、use 關鍵字以及 pub 關鍵字在編譯器中是怎麼運作的，以及多數開發者會怎麼組織程式碼。會在此章節透過範例逐一介紹，不過這裡能快速理解模組是怎麼運作的。
    - **從 crate 源頭開始**：在編譯 crate 時，編譯器會先尋找 crate 源頭檔案(函式庫 crate 的話，通常就是 src/lib.rs；執行檔 crate 的話，通常就是 src/main.rs)來編譯程式碼。
    - **宣告模組**：在 crate 源頭檔案中，可以宣告新的模組，比如說宣告了一個「garden」模組 mod garden;。編譯器會在以下這幾處尋找模組的程式碼：
      1. 同檔案內用 mod garden 加上大括號，寫在括號內的程式碼
      2. src/garden.rs 檔案中
      3. src/garden/mod.rs 檔案中

套件與 Crates

定義模組來控制作用域與隱私權

- 模組懶人包
  - 這裡先快速帶過模組、路徑、use 關鍵字以及 pub 關鍵字在編譯器中是怎麼運作的，以及多數開發者會怎麼組織程式碼。會在此章節透過範例逐一介紹，不過這裡能快速理解模組是怎麼運作的。
    - **模組的路徑：**一旦有個模組成為 crate 的一部分，只要隱私權規則允許，可以在 crate 裡任何地方使用該模組的程式碼。舉例來說，「garden」模組底下的「vegetables」模組的 Asparagus 型別可用 crate::garden::vegetables::Asparagus 來找到。
    - **私有 vs 公開：**模組內的程式碼從上層模組來看預設是私有的。要公開的話，將它宣告為 **pub mod** 而非只是 mod。要讓公開模組內的項目也公開的話，在這些項目前面也加上 pub 即可。
    - **use 關鍵字：**在一個作用域內，use 關鍵字可以建立項目的捷徑，來縮短冗長的路徑名稱。在任何能使用 crate::garden::vegetables::Asparagus 的作用域中，可以透過 use crate::garden::vegetables::Asparagus; 來建立捷徑，接著只需要寫 Asparagus 就能在作用域內使用該型別了。

套件與 Crates

定義模組來控制作用域與隱私權

- 模組懶人包
  - 這裡建立個執行檔 crate 叫做 backyard 來展示這些規則。Crate 的目錄也叫做 backyard，其中包含了這些檔案與目錄：

```
backyard
├── Cargo.lock
├── Cargo.toml
└── src
    ├── garden
    │   └── vegetables.rs
    ├── garden.rs
    └── main.rs
```

  - 此例的 crate 源頭檔案就是 src/main.rs，它包含了：

```rust
use crate::garden::vegetables::Asparagus;

pub mod garden;

fn main() {
    let plant = Asparagus {};
    println!("I'm growing {:?}!", plant);
}
```

套件與 Crates

```rust
use crate::garden::vegetables::Asparagus;

pub mod garden;

fn main() {
    let plant = Asparagus {};
    println!("I'm growing {:?}!", plant);
}
```

定義模組來控制作用域與隱私權

- 模組懶人包
  - pub mod garden; 這行告訴編譯器要包含在 src/garden.rs 中的程式碼，也就是：`pub mod vegetables;`
  - 這裡的 pub mod vegetables; 代表 src/garden/vegetables.rs 的程式碼也包含在內。而這段程式碼就是：

```rust
#[derive(Debug)]
pub struct Asparagus {}
```

套件與 Crates

定義模組來控制作用域與隱私權

- 組織相關程式碼成模組
  - 模組(Modules)能在 crate 內組織程式碼成數個群組以便使用且增加閱讀性。模組也能控制項目的隱私權，因為模組內的程式碼預設是私有的。私有項目是內部的實作細節，並不打算讓外部能使用。能讓模組與其內的項目公開，讓外部程式碼能夠使用並依賴它們。
  - 舉例來說，建立一個提供餐廳功能的函式庫 crate。定義一個函式簽名不過本體會是空的，好專注在程式組織，而非餐廳程式碼的實作。
  - 在餐飲業中，餐廳有些地方會被稱作前台(front of house)而其他部分則是後台(back of house)。前台是消費者的所在區域，這裡是安排顧客座位、點餐並結帳、吧台調酒的地方。而後台則是主廚與廚師工作的廚房、洗碗工洗碗以及經理管理行政工作的地方。

套件與 Crates

定義模組來控制作用域與隱私權

- 組織相關程式碼成模組
  - 要讓 crate 架構長這樣的話，可以組織函式進入模組中。要建立一個新的函式庫叫做 restaurant 的話，請執行 cargo new --lib restaurant。然後將底下範例程式碼放入 src/lib.rs 中，這定義了一些模組與函式簽名。以下是前台的段落：

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

用 mod 關鍵字加上模組的名稱(在此例為 front_of_house)來定義一個模組，並用大括號涵蓋模組的本體。

在模組中，可以再包含其他模組，在此例中包含了 hosting 和 serving。模組還能包含其他項目，像是結構體、列舉、常數、特徵、以及像是範例的函式。

套件與 Crates

定義模組來控制作用域與隱私權

- 組織相關程式碼成模組
  - 使用模組的話，就能將相關的定義組合起來，並用名稱指出會與它們互相關聯。程式設計師在使用此程式碼時只要觀察依據組合起來的模組名稱就好，不必走訪所有的定義。這樣就能快速找到想使用的定義。要對此程式碼增加新功能的開發者也能知道該將程式碼放在哪裡，以維持程式碼的組織。
  - 稍早提到說 src/main.rs 和 src/lib.rs 屬於 crate 的源頭。之所以這樣命名的原因是因為這兩個文件的內容都會在 crate 源頭模組架構中組成一個模組叫做 crate，這樣的結構稱之為模組樹(module tree)。

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

套件與 Crates

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

定義模組來控制作用域與隱私權

- 組織相關程式碼成模組
  - 此樹顯示了有些模組是包含在其他模組內的，比方說 hosting 就在 front_of_house 底下。此樹也顯示了有些模組是其他模組的同輩(siblings)，代表它們是在同模組底下定義的，hosting 和 serving 都在 front_of_house 底下定義。如果模組 A 被包含在模組 B 中，會說模組 A 是模組 B 的下一代(child)，而模組 B 是模組 A 的上一代(parent)。注意到整個模組樹的根是一個隱性模組叫做 crate。
  - 模組樹可能會想到電腦中檔案系統的目錄樹，這是一個非常恰當的比喻！就像檔案系統中的目錄，使用模組來組織程式碼。而且就像目錄中的檔案，需要有方法可以找到模組。

套件與 Crates

參考模組項目的路徑

- 要展示 Rust 如何從模組樹中找到一個項目，要使用和查閱檔案系統時一樣的路徑方法。要呼叫函式的話，需要知道它的路徑，路徑可以有兩種形式：

1. 絕對路徑(absolute path)：
   - **從 crate 的源頭起始的完整路徑**。
     - 如果是外部 crate 的話，絕對路徑起始於該 crate 的名稱；
     - 如果是當前 crate 的話，則是 crate 作為起頭。
2. 相對路徑(relative path)
   - **從本身的模組開始，使用 self、super 或是當前模組的標識符(identifiers)**。

- **無論是絕對或相對路徑其後都會接著一或多個標識符，並使用雙冒號(::)區隔開來。**

套件與 Crates

```rust
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 絕對路徑
    crate::front_of_house::hosting::add_to_waitlist();

    // 相對路徑
    front_of_house::hosting::add_to_waitlist();
}
```

```rust
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

參考模組項目的路徑

- 看看右上範例，假設想呼叫函式add_to_waitlist。這就像在問函式add_to_waitlist的路徑在哪？左上範例移除了一些上面範例的模組與函式來精簡程式碼的呈現方式。
- 展示兩種從 crate 源頭定義的 eat_at_restaurant 函式內呼叫 add_to_waitlist 的方法。這些路徑是正確的，不過目前還有其他問題會導致此範例無法編譯，會在稍後說明。
- eat_at_restaurant 函式是函式庫 crate 公開 API 的一部分，所以會加上 pub 關鍵字。在「使用 pub 關鍵字公開路徑」的段落中，會提到更多 pub 的細節。

套件與 Crates

```rust
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 絕對路徑
    crate::front_of_house::hosting::add_to_waitlist();

    // 相對路徑
    front_of_house::hosting::add_to_waitlist();
}
```

參考模組項目的路徑

- 在 eat_at_restaurant 中第一次呼叫 add_to_waitlist 函式的方式是用絕對路徑。add_to_waitlist 函式和 eat_at_restaurant 都是在同一個 crate 底下，所以可以使用 crate 關鍵字來作為絕對路徑的開頭。接續加上對應的模組直到抵達 add_to_waitlist。
- 可以想像一個有相同架構的檔案系統，然後指定 /front_of_house/hosting/add_to_waitlist 這樣的路徑來執行 add_to_waitlist 程式。使用 crate 這樣的名稱作為 crate 源頭的開始，就像在 shell 使用 / 作為檔案系統的根一樣。
- 而第二次在 eat_at_restaurant 呼叫 add_to_waitlist 的方式是使用相對路徑。路徑的起頭是 front_of_house，因為它和 eat_at_restaurant 都被定義在模組樹的同一層中。
- 這裡相對應的檔案系統路徑就是 front_of_house/hosting/add_to_waitlist。使用一個模組名稱作為開頭通常就是代表相對路徑。

套件與 Crates

```rust
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 絕對路徑
    crate::front_of_house::hosting::add_to_waitlist();

    // 相對路徑
    front_of_house::hosting::add_to_waitlist();
}
```

參考模組項目的路徑

- 何時該用相對或絕對路徑是在專案中要做的選擇，依照想將程式碼的定義連帶與使用它們的程式碼一起移動，或是分開移動到不同地方。
- 舉例來說，如果同時將 front_of_house 模組和 eat_at_restaurant 函式移入另一個模組叫做 customer_experience 的話，就會需要修改 add_to_waitlist 的絕對路徑，但是相對路徑就可以原封不動。而如果只單獨將 eat_at_restaurant 函式移入一個叫做 dining 模組的話，add_to_waitlist 的絕對路徑就不用修改，但相對路徑就需要更新。
- 通常會傾向於指定絕對路徑，因為分別移動程式碼定義與項目呼叫的位置通常是比較常見的。

套件與 Crates

參考模組項目的路徑

```rust
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 絕對路徑
    crate::front_of_house::hosting::add_to_waitlist();

    // 相對路徑
    front_of_house::hosting::add_to_waitlist();
}
```

- 嘗試編譯範例並看看為何不能編譯吧！以下是得到的錯誤資訊：

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: module `hosting` is private
 --> src/lib.rs:9:28
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                            ^^^^^^^ private module
  |
note: the module `hosting` is defined here
 --> src/lib.rs:2:5
  |
2 |     mod hosting {
  |     ^^^^^^^^^^^

error[E0603]: module `hosting` is private
  --> src/lib.rs:12:21
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                     ^^^^^^^ private module
   |
note: the module `hosting` is defined here
  --> src/lib.rs:2:5
   |
2  |     mod hosting {
   |     ^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

錯誤訊息表示 **hosting 模組是私有的**。換句話說，指定 hosting 模組與 add_to_waitlist 函式的路徑是正確的，但是因為它沒有私有部分的存取權，所以 Rust 不讓使用。

在 Rust 中所有項目(函式、方法、結構體、列舉、模組與常數)的隱私權都是私有的。如果想要建立私有的函式或結構體，可以將它們放入模組內。

上層模組的項目無法使用下層模組的私有項目，但下層模組能使用它們上方所有模組的項目。這麼做的原因是因為下層模組用來實現實作細節，而下層模組應該要能夠看到在自己所定義的地方的其他內容。

套件與 Crates

參考模組項目的路徑

```rust
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 絕對路徑
    crate::front_of_house::hosting::add_to_waitlist();

    // 相對路徑
    front_of_house::hosting::add_to_waitlist();
}
```

- 嘗試編譯範例並看看為何不能編譯吧！以下是得到的錯誤資訊：

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: module `hosting` is private
 --> src/lib.rs:9:28
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                            ^^^^^^^ private module
  |
note: the module `hosting` is defined here
 --> src/lib.rs:2:5
  |
2 |     mod hosting {
  |     ^^^^^^^^^^^

error[E0603]: module `hosting` is private
  --> src/lib.rs:12:21
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                     ^^^^^^^ private module
   |
note: the module `hosting` is defined here
  --> src/lib.rs:2:5
   |
2  |     mod hosting {
   |     ^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

繼續用餐廳做比喻的話，可以想像隱私權規則就像是餐廳的後台辦公室。對餐廳顧客來說裡面發生什麼事情都是未知的，但是辦公室經理可以知道經營餐廳時的所有事物。

Rust 選擇這樣的模組系統，讓內部實作細節預設都是隱藏起來的。這樣一來，就能知道內部哪些程式碼需要修改，而不會破壞到外部的程式碼。

不過 Rust 有提供 **pub 關鍵字能讓項目公開**，可以將下層模組內部的一些程式碼公開給上層模組來使用。

套件與 Crates

參考模組項目的路徑

- 使用 pub 關鍵字公開路徑
  - 再執行一次範例的錯誤，它告訴 hosting 模組是私有的。希望上層模組中的 eat_at_restaurant 函式可以呼叫下層模組的 add_to_waitlist 函式，所以**將 hosting 模組加上 pub 關鍵字**，如底下範例所示：

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: module `hosting` is private
 --> src/lib.rs:9:28
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                            ^^^^^^^ private module
  |
note: the module `hosting` is defined here
 --> src/lib.rs:2:5
  |
2 |     mod hosting {
  |     ^^^^^^^^^^^

error[E0603]: module `hosting` is private
  --> src/lib.rs:12:21
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                     ^^^^^^^ private module
   |
note: the module `hosting` is defined here
  --> src/lib.rs:2:5
   |
2  |     mod hosting {
   |     ^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

```rust
mod front_of_house {
    pub mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 絕對路徑
    crate::front_of_house::hosting::add_to_waitlist();

    // 相對路徑
    front_of_house::hosting::add_to_waitlist();
}
```

套件與 Crates

參考模組項目的路徑

- 使用 pub 關鍵字公開路徑

```rust
mod front_of_house {
    pub mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 絕對路徑
    crate::front_of_house::hosting::add_to_waitlist();

    // 相對路徑
    front_of_house::hosting::add_to_waitlist();
}
```

- 不幸的是範例的程式碼仍然回傳了另一個錯誤，如底下範例所示：

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: function `add_to_waitlist` is private
 --> src/lib.rs:9:37
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                                     ^^^^^^^^^^^^^^^ private function
  |
note: the function `add_to_waitlist` is defined here
 --> src/lib.rs:3:9
  |
3 |         fn add_to_waitlist() {}
  |         ^^^^^^^^^^^^^^^^^^^^

error[E0603]: function `add_to_waitlist` is private
  --> src/lib.rs:12:30
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                              ^^^^^^^^^^^^^^^ private function
   |
note: the function `add_to_waitlist` is defined here
  --> src/lib.rs:3:9
   |
3  |         fn add_to_waitlist() {}
   |         ^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

到底發生了什麼事？在 mod hosting 之前加上pub關鍵字確實公開了模組。有了這項修改後，的確可以在取得front_of_house 後，繼續進入 hosting。

但是 hosting 的所有內容仍然是私有的。**模組中的 pub 關鍵字只會讓該模組公開讓上層模組使用而已，而不是存取它所有的內部程式碼。**

因為模組相當於一個容器，如果只公開模組的話，本身並不能做多少事情。需要再進一步選擇公開模組內一些項目才行。

套件與 Crates

參考模組項目的路徑

- 使用 pub 關鍵字公開路徑
  - 錯誤訊息表示 add_to_waitlist 函式是私有的。隱私權規則如同模組一樣適用於結構體、列舉、函式與方法。
  - **在 add_to_waitlist 的函式定義加上 pub 公開**它吧，如底下範例所示：

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: function `add_to_waitlist` is private
 --> src/lib.rs:9:37
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                                     ^^^^^^^^^^^^^^^ private function
  |
note: the function `add_to_waitlist` is defined here
 --> src/lib.rs:3:9
  |
3 |         fn add_to_waitlist() {}
  |         ^^^^^^^^^^^^^^^^^^^^

error[E0603]: function `add_to_waitlist` is private
  --> src/lib.rs:12:30
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                              ^^^^^^^^^^^^^^^ private function
   |
note: the function `add_to_waitlist` is defined here
  --> src/lib.rs:3:9
   |
3  |         fn add_to_waitlist() {}
   |         ^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 絕對路徑
    crate::front_of_house::hosting::add_to_waitlist();

    // 相對路徑
    front_of_house::hosting::add_to_waitlist();
}
```

現在程式碼就能成功編譯了！要理解為何加上 pub 關鍵字使其可以在 add_to_waitlist 取得這些路徑，同時遵守隱私權規則，來看看絕對路徑與相對路徑。

套件與 Crates

參考模組項目的路徑

- 使用 pub 關鍵字公開路徑
  - 錯誤訊息表示 add_to_waitlist 函式是私有的。隱私權規則如同模組一樣適用於結構體、列舉、函式與方法。
  - 在 add_to_waitlist 的函式定義加上 pub 公開它吧，如底下範例所示：

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: function `add_to_waitlist` is private
 --> src/lib.rs:9:37
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                                     ^^^^^^^^^^^^^^^ private function
  |
note: the function `add_to_waitlist` is defined here
 --> src/lib.rs:3:9
  |
3 |         fn add_to_waitlist() {}
  |         ^^^^^^^^^^^^^^^^^^^^

error[E0603]: function `add_to_waitlist` is private
  --> src/lib.rs:12:30
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                              ^^^^^^^^^^^^^^^ private function
   |
note: the function `add_to_waitlist` is defined here
  --> src/lib.rs:3:9
   |
3  |         fn add_to_waitlist() {}
   |         ^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

```rust
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 絕對路徑
    crate::front_of_house::hosting::add_to_waitlist();

    // 相對路徑
    front_of_house::hosting::add_to_waitlist();
}
```

在絕對路徑中，始於 crate，這是 crate 模組樹的根。再來 front_of_house 模組被定義在 crate 源頭中，front_of_house 模組不是公開，但因為 eat_at_restaurant 函式被定義在與 front_of_house 同一層模組中(也就是 eat_at_restaurant 與 front_of_house 同輩(siblings))，可以從 eat_at_restaurant 參考 front_of_house。

接下來是有 pub 標記的 hosting 模組，可以取得 hosting 的上層模組，所以可以取得 hosting。最後 add_to_waitlist 函式也有 pub 標記而可以取得它的上層模組，所以整個程式呼叫就能執行了！

套件與 Crates

參考模組項目的路徑

- 使用 pub 關鍵字公開路徑
  - 錯誤訊息表示 add_to_waitlist 函式是私有的。隱私權規則如同模組一樣適用於結構體、列舉、函式與方法。
  - 在 add_to_waitlist 的函式定義加上 pub 公開它吧，如底下範例所示：

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: function `add_to_waitlist` is private
 --> src/lib.rs:9:37
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                                     ^^^^^^^^^^^^^^^ private function
  |
note: the function `add_to_waitlist` is defined here
 --> src/lib.rs:3:9
  |
3 |         fn add_to_waitlist() {}
  |         ^^^^^^^^^^^^^^^^^^^^

error[E0603]: function `add_to_waitlist` is private
  --> src/lib.rs:12:30
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                              ^^^^^^^^^^^^^^^ private function
   |
note: the function `add_to_waitlist` is defined here
  --> src/lib.rs:3:9
   |
3  |         fn add_to_waitlist() {}
   |         ^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 絕對路徑
    crate::front_of_house::hosting::add_to_waitlist();

    // 相對路徑
    front_of_house::hosting::add_to_waitlist();
}
```

而在相對路徑中，基本邏輯與絕對路徑一樣，不過第一步有點不同。

不是從 crate 源頭開始，路徑是從 front_of_house 開始。front_of_house 與 eat_at_restaurant 被定義在同一層模組中，所以從 eat_at_restaurant 開始定義的相對路徑是有效的。

再來因為 hosting 與 add_to_waitlist 都有 pub 標記，其餘的路徑也都是可以進入的，所以此函式呼叫也是有效的！

套件與 Crates

參考模組項目的路徑

```
fn deliver_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::deliver_order();
    }

    fn cook_order() {}
}
```

- 使用 super 作為相對路徑的開頭
  - 可以在路徑開頭使用 super 來建構從上層模組出發的相對路徑，而不用從 crate 源頭開始。這就像在檔案系統中使用 .. 作為路徑開頭一樣。使用 super 能參考確定位於上層模組的項目。當模組與上層模組有高度關聯，且上層模組可能以後會被移到模組樹的其他地方時，這能讓組織模組樹更加輕鬆。
  - 請考慮右上範例的程式碼，這模擬了一個主廚修正一個錯誤的訂單，並親自提供給顧客的場景。定義在 back_of_house 模組的函式 fix_incorrect_order 呼叫了定義在上層模組的函式 deliver_order，不過這次是使用 super 來指定 deliver_order 的路徑：
  - fix_incorrect_order 函式在 back_of_house 模組中，所以可以使用 super 前往 back_of_house 的上層模組，在此例的話就是源頭 crate。然後在此時能找到 deliver_order。
  - 成功！認定 back_of_house 模組與 deliver_order函式應該會維持這樣相同的關係，在要組織 crate 的模組樹時，它們理當一起被移動。因此使用 super 在未來程式碼被移動到不同模組時，不用更新太多程式路徑。

套件與 Crates

參考模組項目的路徑

- 公開結構體與列舉
  - 也可以使用 pub 來公開結構體與列舉，但是有些額外細節要考慮到。如果在結構體定義之前加上 pub 的話，的確能公開結構體，但是結構體內的欄位仍然會是私有的。可以視情況決定每個欄位要不要公開。
  - 在右邊範例定義了一個公開的結構體 back_of_house::Breakfast 並公開欄位 toast，不過將欄位 seasonal_fruit 維持是私有的。這次範例模擬的情境是，餐廳顧客可以選擇早餐要點什麼類型的麵包，但是由主廚視庫存與當季食材來決定提供何種水果。餐廳提供的水果種類隨季節變化很快，所以顧客無法選擇或預先知道他們會拿到何種水果。

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("桃子"),
            }
        }
    }
}

pub fn eat_at_restaurant() {
    // 點夏季早餐並選擇黑麥麵包
    let mut meal = back_of_house::Breakfast::summer("黑麥");
    // 我們想改成全麥麵包
    meal.toast = String::from("全麥");
    println!("我想要{}麵包，謝謝", meal.toast);

    // 接下來這行取消註解的話，我們就無法編譯通過
    // 我們無法擅自更改餐點搭配的季節水果
    // meal.seasonal_fruit = String::from("藍莓");
}
```

套件與 Crates

參考模組項目的路徑

- 公開結構體與列舉
  - 因為 back_of_house::Breakfast 結構體中的 toast 欄位是公開的，在 eat_at_restaurant 中可以加上句點來對 toast 欄位進行讀寫。
  - 注意不能在 eat_at_restaurant 使用 seasonal_fruit 欄位，因為它是私有的。請嘗試解開修改 seasonal_fruit 欄位數值的那行程式註解，看看會獲得什麼錯誤！
  - 另外因為 back_of_house::Breakfast 擁有私有欄位，該結構體必須提供一個公開的關聯函式(associated function)才有辦法產生Breakfast的實例(在此例命名為 summer)。
  - 如果Breakfast沒有這樣的函式的話，就無法在 eat_at_restaurant 建立 Breakfast 的實例，因為無法在 eat_at_restaurant 設置私有欄位 seasonal_fruit 的數值。

```rust
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("桃子"),
            }
        }
    }
}

pub fn eat_at_restaurant() {
    // 點夏季早餐並選擇黑麥麵包
    let mut meal = back_of_house::Breakfast::summer("黑麥");
    // 我們想改成全麥麵包
    meal.toast = String::from("全麥");
    println!("我想要{}麵包，謝謝", meal.toast);

    // 接下來這行取消註解的話，我們就無法編譯通過
    // 我們無法擅自更改餐點搭配的季節水果
    // meal.seasonal_fruit = String::from("藍莓");
}
```

套件與 Crates

參考模組項目的路徑

- 公開結構體與列舉
  - 接下來，如果公開列舉的話，那它所有的變體也都會公開。只需要在 enum 關鍵字之前加上 pub 就好，如底下範例所示：

```
mod back_of_house {
    pub enum Appetizer {
        Soup,
        Salad,
    }
}

pub fn eat_at_restaurant() {
    let order1 = back_of_house::Appetizer::Soup;
    let order2 = back_of_house::Appetizer::Salad;
}
```

因為公開了Appetizer列舉，可以在eat_at_restaurant使用Soup和 Salad。列舉的變體沒有全部都公開的話，通常會讓列舉很不好用。要用 pub 標註所有的列舉變體都公開的話又很麻煩。所以公開列舉的話，預設就會公開其變體。相反地，結構體不讓它的欄位全部都公開的話，通常反而比較實用。因此結構體欄位的通用原則是預設為私有，除非有 pub 標註。

還有一個pub的使用情境還沒提到，也就是模組系統最後一項功能：use 關鍵字。接下來會先解釋 use，再來研究如何組合 pub 和 use。

套件與 Crates

執行檔與函式庫套件的最佳實踐

- 提到套件能同時包含 src/main.rs 作為執行檔 crate 源頭以及 src/lib.rs 作為函式庫 crate 源頭，兩者預設都是用套件的名稱。通常來說，一個函式庫與一個執行檔 crate 這樣的套件模式，在執行檔中只會留下必要的程式碼，其餘則呼叫函式庫的程式碼。這樣其他專案也能運用到套件提供的多數功能，因為函式庫 crate 的程式碼可以分享。
- 模組要定義在 src/lib.rs。然後在執行檔 crate 中，任何公開項目都能用套件名稱作為開頭找到。執行檔 crate 應視為函式庫 crate 的使用者，就像外部 crate 那樣使用一樣，只能使用公開 API。這有助於設計出良好的 API，不僅是作者，同時還是自己的客戶！
- 在後面會透過寫個命令列程式來介紹這樣的組織練習，該程式會包含一個執行檔 crate 與一個函式庫 crate。

套件與 Crates

透過 use 關鍵字引入路徑

- 要是每次都得寫出呼叫函式的路徑的話是很冗長、重複且不方便的。舉例來說，底下範例在考慮要使用絕對或相對路徑來呼叫 add_to_waitlist 函式時，每次想要呼叫 add_to_waitlist，都得指明 front_of_house 以及 hosting。幸運的是，有簡化過程的辦法：可以用 use 關鍵字建立路徑的捷徑，然後在作用域內透過更短的名稱來使用。
- 在底下範例中，引入了 crate::front_of_house::hosting 模組進 eat_at_restaurant 函式的作用域中，所以要呼叫函式 add_to_waitlist 的話，只需要指明 hosting::add_to_waitlist：

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

套件與 Crates

透過 use 關鍵字引入路徑

- 使用 use 將路徑引入作用域就像是在檔案系統中產生符號連結一樣 (symbolic link)。在 crate 源頭加上 use crate::front_of_house::hosting 後，hosting 在作用域內就是個有效的名稱了。使用 use 的路徑也會檢查隱私權，就像其他路徑一樣。
- 注意到 use 只會在它所位在的特定作用域內建立捷徑。底下範例將eat_at_restaurant移入子模組 customer，這樣就會與 use 陳述式的作用域不同，所以其函式本體將無法編譯：

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

mod customer {
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
    }
}
```

套件與 Crates

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

mod customer {
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
    }
}
```

透過 use 關鍵字引入路徑

- 編譯器錯誤顯示了該捷徑無法用在 customer 模組內：

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0433]: failed to resolve: use of undeclared crate or module `hosting`
  --> src/lib.rs:11:9
   |
11 |         hosting::add_to_waitlist();
   |         ^^^^^^^ use of undeclared crate or module `hosting`

warning: unused import: `crate::front_of_house::hosting`
 --> src/lib.rs:7:5
  |
7 | use crate::front_of_house::hosting;
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_imports)]` on by default
```

- 會發現還有另外一個警告說明 use 在它的作用域中並沒有被用到！要解決此問題的話，可以將 use 也移動到 customer 模組內，或是在customer 子模組透過 super::hosting 參考上層模組的捷徑。

套件與 Crates

透過 use 關鍵字引入路徑

- 建立慣用的 use 路徑
  - 在上面範例可能會好奇為何指明 use crate::front_of_house::hosting 然後在 eat_at_restaurant 呼叫，而不是直接用 use 指明 add_to_waitlist 函式的整個路徑就好。像底下範例這樣寫：

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

mod customer {
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
    }
}
```

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting::add_to_waitlist;

pub fn eat_at_restaurant() {
    add_to_waitlist();
}
```

雖然上面範例與左邊範例都能完成相同的任務，但是上面範例的做法比較符合習慣用法。使用 use 將函式的上層模組引入作用域，必須在呼叫函式時得指明對應模組。在呼叫函式時指定上層模組能清楚地知道該函式並非本地定義的，同時一樣能簡化路徑。

左邊範例的程式碼會不清楚 add_to_waitlist 是在哪定義的。

套件與 Crates

透過 use 關鍵字引入路徑

- 建立慣用的 use 路徑
  - 另一方面，如果是要使用 use 引入結構體、列舉或其他項目的話，直接指明完整路徑反而是符合習慣的方式。底下範例顯示了將標準函式庫的 HashMap 引入執行檔 crate 作用域的習慣用法：

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

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

  - 此習慣沒什麼強硬的理由：就只是大家已經習慣這樣的用法來讀寫 Rust 的程式碼。

套件與 Crates

透過 use 關鍵字引入路徑

- 建立慣用的 use 路徑
  - 這樣的習慣有個例外，那就是如果將兩個相同名稱的項目使用 use 陳述式引入作用域時，因為 Rust 不會允許。底下範例展示了如何引入兩個同名但屬於不同模組的 Result 型別進作用域中並使用的方法：

```
use std::fmt;
use std::io;

fn function1() -> fmt::Result {
    // --省略--
}

fn function2() -> io::Result<()> {
    // --省略--
}
```

如同所見使用對應的模組可以分辨出是在使用哪個 Result 型別。如果直接指明 use std::fmt::Result 和 use std::io::Result 的話，會在同一個作用域中擁有兩個 Result 型別，這樣一來 Rust 就無法知道想用的 Result 是哪一個。

套件與 Crates

透過 use 關鍵字引入路徑

- 使用 as 關鍵字提供新名稱
  - 要在相同作用域中使用use引入兩個同名型別的話，還有另一個辦法。在路徑之後，可以用as指定一個該型別在本地的新名稱，或者說別名(alias)。底下範例展示重寫了之前的範例，將其中一個 Result 型別使用 as 重新命名：

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --省略--
}

fn function2() -> IoResult<()> {
    // --省略--
}
```

在第二個 use 陳述式，選擇了將 std::io::Result 型別重新命名為 IoResult，這樣就不會和同樣引入作用域內 std::fmt 的 Result 有所衝突。

之前的範例與左邊的範例都屬於習慣用法，可以選擇比較喜歡的方式！

套件與 Crates

透過 use 關鍵字引入路徑

- 使用 pub use 重新匯出名稱
  - 當使用 use 關鍵字將名稱引入作用域時，該有效名稱在新的作用域中是私有的。要是希望呼叫這段程式碼時，也可以使用這個名稱的話(就像該名稱是在此作用域內定義的)，可以組合 pub 和 use。這樣的技巧稱之為重新匯出(re-exporting)，因為將項目引入作用域，並同時公開給其他作用域參考。
  - 底下範例將之前範例在源頭模組中原本的 use 改成 pub use：

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

套件與 Crates

透過 use 關鍵字引入路徑

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

- 使用 pub use 重新匯出名稱
  - 在此之前，外部程式碼會需要透過 restaurant::front_of_house::hosting::add_to_waitlist() 這樣的路徑才能呼叫 add_to_waitlist。現在 pub use 從源頭模組重新匯出了 hosting 模組，外部程式碼現在可以使用 restaurant::hosting::add_to_waitlist() 這樣的路徑就好。
  - 當程式碼的內部結構與使用程式的開發者對於該領域所想像的結構不同時，重新匯出會很有用。再次用餐廳做比喻的話就像是，經營餐廳的人可能會想像餐廳是由「前台」與「後台」所組成，但光顧的顧客可能不會用這些術語來描繪餐廳的每個部分。
  - 使用 pub use 的話，可以用某種架構寫出程式碼，再以不同的架構對外公開。這樣的函式庫可以完整的組織起來，且對開發函式庫的開發者與使用函式庫的開發者都提供友善的架構。會在後面的「透過 pub use 匯出理想的公開 API」段落再看看另一個 pub use 的範例並了解它會如何影響 crate 的技術文件。

套件與 Crates

透過 use 關鍵字引入路徑

- 使用外部套件
  - 在之前有用到一個外部套件叫做 rand 來取得隨機數字。要在專案內使用 rand 的話，會在 Cargo.toml 加上此行：`rand = "0.8.5"`
  - 在 Cargo.toml 新增 rand 作為依賴函式庫會告訴 Cargo 要從 crates.io 下載 rand 以及其他相關的依賴，讓專案可以使用 rand。
  - 接下來要將 rand 的定義引入套件的作用域的話，加上一行 use 後面接著 crate 的名稱 rand 然後列出想要引入作用域的項目。回想一下之前將 Rng 特徵引入作用域中，並呼叫函式 rand::thread_rng：

```
use rand::Rng;

fn main() {
    let secret_number = rand::thread_rng().gen_range(1..=100);
}
```

套件與 Crates

透過 use 關鍵字引入路徑

- 使用外部套件
  - Rust 社群成員在crates.io發佈了不少套件可供使用，要將這些套件引入到套件的步驟是一樣的。在套件的 Cargo.toml 檔案列出它們，然後使用 use 將這些 crate 內的項目引入作用域中。
  - 請注意到標準函式庫 std 對於套件來說也是一個外部 crate。由於標準函式庫會跟著 Rust 語言發佈，所以不需要更改 Cargo.toml 來包含 std。但是仍然需使用 use 來將它的項目引入套件的作用域中。舉例來說，要使用 HashMap 可以這樣寫：
    ```
    use std::collections::HashMap;
    ```
  - 這是個用標準函式庫的 crate 名稱 std 起頭的絕對路徑。

套件與 Crates

透過 use 關鍵字引入路徑

- 使用巢狀路徑來清理大量的 use 行數
  - 如果要使用在相同 crate 或是相同模組內定義的數個項目，針對每個項目都單獨寫一行的話，會佔據檔案內很多空間。舉例來說，底下範例用了這兩個 use 陳述式來引入作用域中：

```
// --省略--
use std::cmp::Ordering;
use std::io;
// --省略--
```

  - 可以改使用巢狀路徑(nested paths)來只用一行就能將數個項目引入作用域中。先指明相同路徑的部分，加上雙冒號，然後在大括號內列出各自不同的路徑部分，如底下範例所示：

```
// --省略--
use std::{cmp::Ordering, io};
// --省略--
```

套件與 Crates

透過 use 關鍵字引入路徑

- 使用巢狀路徑來清理大量的 use 行數
  - 在較大的程式中，使用巢狀路徑將相同 crate 或相同模組中的許多項目引入作用域，可以大量減少 use 陳述式的數量！
  - 可以在路徑中的任何部分使用巢狀路徑，這在組合兩個享有相同子路徑的 use 陳述式時非常有用。舉例來說，底下範例顯示了兩個 use 陳述式：一個將 std::io 引入作用域，另一個將 std::io::Write 引入作用域：

```
use std::io;
use std::io::Write;
```

套件與 Crates

透過 use 關鍵字引入路徑

```
use std::io;
use std::io::Write;
```

- 使用巢狀路徑來清理大量的 use 行數
  - 這兩個路徑的相同部分是 std::io，這也是整個第一個路徑。要將這兩個路徑合為一個 use 陳述式的話，可以在巢狀路徑使用 self，如底下範例所示：

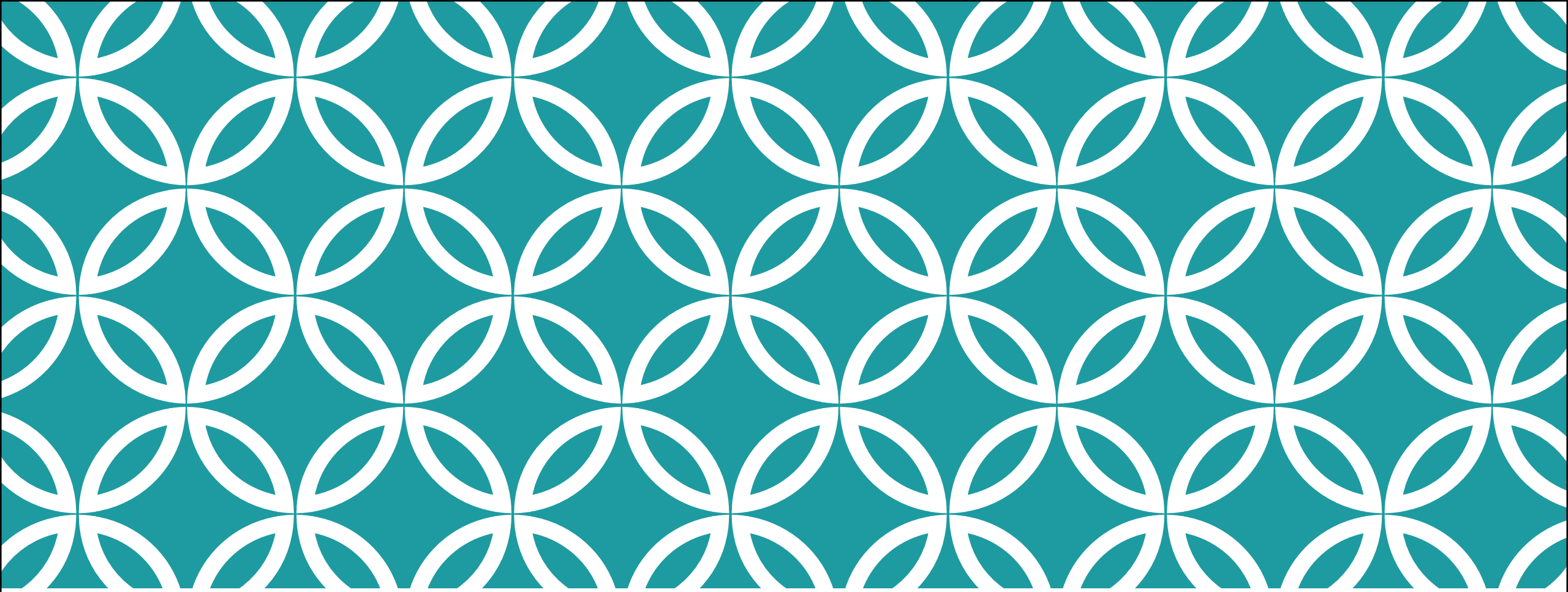
```
use std::io::{self, Write};
```

  - 此行就會將 std::io 和 std::io::Write 引入作用域。

套件與 Crates

透過 use 關鍵字引入路徑

- 全域運算子
  - 如果想要將在一個路徑中所定義的所有公開項目引入作用域的話，可以在指明路徑之後加上全域(glob)運算子 *：
    ```
    use std::collections::*;
    ```
  - 此 use 陳述式會將 std::collections 定義的所有公開項目都引入作用域中。不過請小心使用全域運算子！它容易無法分辨作用域內的名稱，以及程式中使用的名稱是從哪定義來的。
  - 全域運算子很常用在 tests 模組下，將所有東西引入測試中。會在之後的「如何寫測試」段落來討論。全域運算子也常拿來用在 prelude 模式中，可以查閱標準函式庫的技術文件來瞭解此模式的更多資訊。



常見集合

常見集合

Rust 的標準函式庫提供一些非常實用的資料結構稱之為集合(collections)。多數其他資料型別只會呈現一個特定數值，但是集合可以包含數個數值。不像內建的陣列與元組型別，這些集合指向的資料位於**堆積**上，代表資料的數量不必在編譯期就知道，而且可以隨著程式執行增長或縮減。

每種集合都有不同的能力以及消耗，依照情形選擇適當的集合，是一項會隨著開發時間漸漸掌握的技能。在此會介紹三種在 Rust 程式中十分常用的集合：

- 向量(Vector)：
  - 允許接二連三地儲存數量不定的數值。
- 字串(String)：
  - 字元的集合。在之前就提過 String 型別，這邊會深入介紹。
- 雜湊映射(Hash map)：
  - 允許將值(value)與特定的鍵(key)相關聯。這是從一種更通用的資料結構映射(map)衍生出來的特定實作。

常見集合

透過向量儲存列表

- 第一個要來看的集合是 Vec<T> 常稱為向量(vector)。向量允許在一個資料結構儲存不止一個數值，而且該結構的記憶體會接連排列所有數值。它們很適合用來處理手上的項目列表，像是一個檔案中每行的文字或是購物車內每項物品。

建立新的向量

- 要建立一個新的空向量的話，呼叫 Vec::new 函式，如底下範例所示：

```
let v: Vec<i32> = Vec::new();
```

常見集合

建立新的向量

```
let v: Vec<i32> = Vec::new();
```

- 注意在此加了型別詮釋。因為沒有對此向量插入任何數值，Rust 不知道想儲存什麼類型的元素。這是一項重點，向量是用泛型(generics)實作，會在之後說明如何為型別使用泛型。現在只需要知道標準函式庫提供的Vec<T>型別可以持有任意型別，然後當特定向量要持有特定型別時，可以在尖括號內指定該型別。在右上範例，告訴 Rust 在 v 中的 Vec<T> 會持有 i32 型別的元素。
- 不過通常在建立 Vec<T> 時只需要給予初始數值，Rust 就能推導出想儲存的數值型別，所以不太常會需要指明型別詮釋。Rust 還提供了 vec! 巨集能方便地建立一個新的向量並取得提供的數值。在底下範例中，建立了一個新的 Vec<i32> 並擁有數值 1、2 和 3。整數型別為 i32 是因為這是預設整數型別，如同在之前的「資料型別」段落提到的一樣。

```
let v = vec![1, 2, 3];
```

- 因為給予了初始的 i32 數值，Rust 可以推導出 v 的型別為 Vec<i32>，所以型別詮釋就不是必要的了。接下來，看看如何修改向量。

常見集合

更新向量

- 要在建立向量之後新增元素的話，可以使用 push 方法，如底下範例所示：

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

- 與其他變數一樣，如果想要變更其數值的話，需要使用 mut 關鍵字使它成為可變的，如同之前提到的一樣。插入的數值所屬型別均為i32，然後Rust可以從資料推導，所以不必指明Vec<i32>。

常見集合

讀取向量元素

- 要參考向量儲存的數值有兩種方式。為了更加清楚說明此範例，詮釋了函式回傳值的型別。
- 底下範例顯示了取得向量中數值的方法，可以使用索引語法與 get 方法：

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
println!("第三個元素是 {third}");

let third: Option<&i32> = v.get(2);
match third {
    Some(third) => println!("第三個元素是 {third}"),
    None => println!("第三個元素並不存在。"),
}
```

這邊要注意一些地方。使用了索引數值 2 來獲取第三個元素：向量可以用數字來索引，從零開始計算。

使用 & 和 [] 會給索引數值的元素參考，而使用 get 方法加上一個索引作為引數，則會給 Option<&T>，可以用 match 來配對。

常見集合

```rust
let v = vec![1, 2, 3, 4, 5];

let does_not_exist = &v[100];
let does_not_exist = v.get(100);
```

讀取向量元素

- Rust 提供兩種取得元素參考方式，所以當嘗試使用索引數值取得向量範圍外的元素時，可以決定程式的行為。來看看一個範例，有一個向量擁有五個元素，但嘗試用索引 100 來取得對應數值，如右上範例所示：
- 當執行程式時，第一個 [] 方法會讓程式恐慌，因為它參考了不存在的元素。此方法適用於當希望一有無效索引時就讓程式崩潰的狀況。當使用 get 方法來索取向量不存在的索引時，它會回傳 None 而不會恐慌。如果正常情況下，偶而會不小心存取超出向量範圍索引的話，就會想要只用此方法。程式碼就會有個邏輯專門處理 Some(&element) 或 None，如同前面所述。
- 舉例來說，可能會有由使用者輸入的索引。如果不小心輸入太大的數字的話，程式可以回傳 None，可以告訴使用者目前向量有多少項目，並讓使用者可以再輸入一次。這會比直接讓程式崩潰還來的友善，可能只是不小心打錯而已！

常見集合

讀取向量元素

- 當程式有個有效參考時，借用檢查器(borrow checker)會貫徹所有權以及借用規則來確保此參考及其他對向量內容的參考都是有效的。回想一下有個規則是**不能在同個作用域同時擁有可變與不可變參考**。這個規則一樣適用於底下範例，在此有一個向量第一個元素的不可變參考，然後嘗試在向量後方新增元素。如果嘗試在此動作後繼續使用第一個參考的話，程式會無法執行：

```
let mut v = vec![1, 2, 3, 4, 5];

let first = &v[0];

v.push(6);

println!("第一個元素是：{first}");
```

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:5
  |
4 |     let first = &v[0];
  |                  - immutable borrow occurs here
5 |
6 |     v.push(6);
  |     ^^^^^^^^^ mutable borrow occurs here
7 |
8 |     println!("第一個元素為 {first}");
  |                             ----- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `collections` due to previous error
```

常見集合

讀取向量元素

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:5
  |
4 |     let first = &v[0];
  |                  - immutable borrow occurs here
5 |
6 |     v.push(6);
  |     ^^^^^^^^^ mutable borrow occurs here
7 |
8 |     println!("第一個元素為 {first}");
  |                             ----- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `collections` due to previous error
```

- 範例的程式碼看起來好像能執行。為何第一個元素的參考要在意向量的最後端發生了什麼事呢？此錯誤其實跟向量運作的方式有關：**由於向量會將元素放在前一位的記憶體位置後方，在向量後方新增元素時，如果當前向量的空間不夠再塞入另一個值的話，可能會需要配置新的記憶體並複製舊的元素到新的空間中。**
- 這樣一來，**第一個元素的索引可能就會指向已經被釋放的記憶體**，借用規則會防止程式遇到這樣的情形。

常見集合

走訪向量的元素

```rust
let v = vec![100, 32, 57];
for i in &v {
    println!("{i}");
}
```

- 想要依序存取向量中每個元素的話，可以走訪所有元素而不必用索引一個一個取得。上方範例闡釋了如何使用 for 迴圈來取得一個 i32 向量中每個元素的不可變參考並印出他們：
- 還可以走訪可變向量中的每個元素取得可變參考來改變每個元素。底下範例就使用 for 迴圈來為每個元素加上 50：

```rust
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

- 要改變可變參考指向的數值，在使用 += 運算子之前，需要使用 * 解參考運算子來取得 i 的數值。會在之後的「追蹤指標的數值」段落來講解更多解參考運算子的細節。
- 當走訪向量時，無論是不可變或可變地都是安全，因為借用檢查器的規則能確保如此。如果嘗試在上方二個範例的 for 迴圈本體插入或刪除項目，就會獲得和之前範例程式碼類似的編譯錯誤。for 迴圈持有的向量參考能避免同時修改整個向量。

常見集合

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("藍色")),
    SpreadsheetCell::Float(10.12),
];
```

使用列舉來儲存多種型別

- **向量只能儲存同型別的數值**，這在某些情況會很不方便，一定會有場合是要儲存不同型別到一個列表中。幸運的是，列舉的變體是定義在相同的列舉型別，所以**當需要在向量儲存不同型別的元素時，可以用列舉來定義！**
- 舉例來說，假設想從表格中的一行取得數值，但是有些行內的列會包含整數、浮點數以及一些字串。可以定義一個列舉，其變體會持有不同的數值型別，然後**所有的列舉變體都會被視為相同型別：就是它們的列舉**。接著就可以建立一個擁有此列舉型別的向量，最終達成持有不同型別。如右上範例所示：

常見集合

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("藍色")),
    SpreadsheetCell::Float(10.12),
];
```

使用列舉來儲存多種型別

- Rust 需要在編譯時期知道向量的型別以及要在堆積上用到多少記憶體才能儲存每個元素。必須明確知道哪些型別可以放入向量中。
- 如果 Rust 允許向量一次持有任意型別的話，在對向量中每個元素進行處理時，可能就會有一或多種型別會產生錯誤。使用列舉和 match 表達式讓 Rust 可以在編譯期間確保每個可能的情形都已經處理完善了，如同之前提到的一樣。如果無法確切知道執行時程式所處理的所有型別的話，列舉就不管用了。這時使用**特徵物件**會比較好，會在之後再來解釋。
- 現在已經講了一些向量常見的用法，有時間的話記得到向量的 API技術文件瞭解標準函式庫中 Vec<T> 所有實用的方法。舉例來說，除了 push 方法以外，還有個 pop 方法可以移除並回傳最後一個元素。

常見集合

釋放向量的同時也會釋放其元素

- 就像其它 struct 一樣，向量會在作用域結束時被釋放，如底下範例所示：

```
{
    let v = vec![1, 2, 3, 4];

    // 使用 v 做些事情
} // <- v 在此離開作用域並釋放
```

- 當向量被釋放時，其所有內容也都會被釋放，代表它持有的那些整數都會被清除。這雖然聽起來很直觀，但是當開始參考向量中的元素時可能就會變得有點複雜。

常見集合

透過字串儲存 UTF-8 編碼的文字

- 已經在前面提到字串(String)，但現在要更加深入探討。初學者常常會卡在三個環節：Rust 傾向於回報可能的錯誤、字串的資料結構比開發者所熟悉的還要複雜，以及 UTF-8。這些要素讓來自其他程式語言背景的開發者會遇到一些困難。
- 會在集合章節討論字串的原因是，字串本身就是位元組的集合，且位元組作為文字呈現時，它會提供一些實用的方法。在此段落將和其他集合型別一樣討論 String 的操作，像是建立、更新與讀取。還會討論到 String 與其他集合不一樣的地方，像是 String 的索引就比其他集合還複雜，因為它會依據人們對於 String 資料型別的理解而有所不同。

常見集合

什麼是字串？

- 首先要好好定義字串(String)這個術語。Rust 在核心語言中只有一個字串型別，那就是字串切片 str，它通常是以借用的形式存在 &str。在前面章節中提到字串切片是一個針對存在某處的 UTF-8 編碼資料的參考。舉例來說，字串字面值(String literals)就儲存在程式的執行檔中，因此就是字串切片。
- String 型別是 Rust 標準函式庫所提供的型別，並不是核心語言內建的型別，它是可增長的、可變的、可擁有所有權的UTF-8 編碼字串型別。當提及 Rust 中的「字串」時，通常指的是 String 以及字串切片 &str 型別，而不只是其中一種型別。雖然此段落大部分都在討論 String，這兩個型別都時常用在 Rust 的標準函式庫中，且 String 與字串切片都是 UTF-8 編碼的。

常見集合

建立新的字串

- 許多 Vec<T> 可使用的方法在 String 也都能用，因為 String 其實就是一種位元組向量的封裝再加上一些額外的保障、限制與能力。其中一個 Vec<T> 與 String 都有且用途相同的函式就是 new，這用來產生新的實例，如底下範例所示：

```
let mut s = String::new();
```

- 此行會建立新的字串叫做 s，之後可以再寫入資料。不過通常會希望建立字串的同時能夠初始化資料。為此可以使用 to_string 方法，任何有實作 Display 特徵的型別都可以使用此方法，就像字串字面值的使用方式一樣。底下範例就展示了兩種例子：

```
let data = "初始內容";

let s = data.to_string();

// 此方法也能直接用於字面值上
let s = "初始內容".to_string();
```

常見集合

建立新的字串

```
let data = "初始內容";

let s = data.to_string();

// 此方法也能直接用於字面值上
let s = "初始內容".to_string();
```

- 此程式碼建立了一個字串內容為**初始內容**。
- 可以用函式 String::from 從字串字面值建立 String。底下範例的程式碼和使用to_string的上面範例效果一樣：

```
let s = String::from("初始內容");
```

- 因為字串用在許多地方，可以使用許多不同的通用字串 API 供選擇。有些看起來似乎是多餘的，但是它們都有一席之地的！在上面的範例中 String::from 和 to_string 都在做相同的事，所以選擇跟喜好風格與閱讀性比較有關。

常見集合

建立新的字串

- 另外記得字串是 UTF-8 編碼的，所以可以包含任何正確編碼的資料，如底下範例所示：

```
let hello = String::from("السلام عليكم");
let hello = String::from("Dobrý den");
let hello = String::from("Hello");
let hello = String::from("שָׁלוֹם");
let hello = String::from("नमस्ते");
let hello = String::from("こんにちは");
let hello = String::from("안녕하세요");
let hello = String::from("你好");
let hello = String::from("Olá");
let hello = String::from("Здравствуйте");
let hello = String::from("Hola");
```

- 以上全是合理的 String 數值。

常見集合

更新字串

- 就和 Vec<T> 一樣，如果插入更多資料的話，String 可以增長大小並變更其內容。除此之外也可以使用 + 運算子或 format! 巨集來串接 String 數值。

使用 push_str 和 push 追加字串

- 可以使用 push_str 方法來追加一個字串切片使字串增長，如底下範例所示：

```
let mut s = String::from("foo");
s.push_str("bar");
```

- 在這兩行之後，s 會包含 foobar。push_str 方法取得的是字串切片因為並不需要取得參數的所有權。

  舉例來說，在右下範例想在 s2 追加其內容給 s1 之後仍能使用：

  如果 push_str 方法會取得 s2 的所有權，就無法在最後一行印出其數值了。幸好這段程式碼是可以執行的！

```
let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2);
println!("s2 is {s2}");
```

常見集合

更新字串

使用 push_str 和 push 追加字串

- 而 push 方法會取得一個字元作為參數並加到 String 上。底下範例顯示了一個使用 push 方法將字母 "l" 加到 String 的程式碼：

```
let mut s = String::from("lo");
s.push('l');
```

- 結果就是 s 會包含 lol。

常見集合

更新字串

使用 + 運算子或 format! 巨集串接字串

- 通常會想要組合兩個字串在一起，其中一種方式是用 + 運算子。如底下範例所示：

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // 注意到 s1 被移動因此無法再被使用
```

- 程式碼最後的字串 s3 就會獲得 Hello, world!。s1 之所以在相加後不再有效，以及 s2 是使用參考的原因，都和使用 + 運算子時呼叫的方法簽名有關。+ 運算子使用的是 add 方法，其簽名會長得像這樣：

```
fn add(self, s: &str) -> String {
```

常見集合

更新字串

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // 注意到 s1 被移動因此無法再被使用
```

```
fn add(self, s: &str) -> String {
```

使用 + 運算子或 format! 巨集串接字串

- 在標準函式庫中 add 是用泛型(generics)與關聯型別(associated types)定義。在此使用實際型別代替的add簽名。會在之後討論到泛型。此簽名給了一些需要瞭解 + 運算子的一些線索。
- 首先 s2 有 & 代表是將第二個字串的參考與第一個字串相加，因為函式 add 中的參數 s 說明只能將 &str 與 String 相加，無法將兩個 String 數值相加。但等等 &s2 是 &String 才對，並非 add 第二個參數所指定的 &str。為何之前的範例可以編譯呢？
- 可以在 add 的呼叫中使用 &s2 的原因是因為編譯器可以**強制(coerce) &String 引數轉換成 &str**。當呼叫 add 方法時，Rust 強制解參考(deref coercion)讓 &s2 變成 &s2[..]。會在之後探討強制解參考。因為 add 不會取得 s 參數的所有權，s2 在此運算後仍然是個有效的 String。

常見集合

更新字串

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // 注意到 s1 被移動因此無法再被使用
```

使用 + 運算子或 format! 巨集串接字串

- 再來，可以看到 add 的簽名會取得 self 的所有權，因為 self 沒有 &。這代表之前範例的 s1 會移動到 add 的呼叫內，在之後就不再有效。
- 所以雖然 let s3 = s1 + &s2; 看起來像是它拷貝了兩個字串的值並產生了一個新的，但此陳述式實際上是取得 s1 的所有權、追加一份 s2 的複製內容、然後回傳最終結果的所有權。換句話說，雖然它看起來像是產生了很多拷貝，但實際上並不是。此實作反而比較有效率。
- 如果需要串接數個字串的話，+ 運算子的行為看起來就顯得有點笨重了：

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
```

常見集合

更新字串

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
```

使用 + 運算子或 format! 巨集串接字串

- 此時 s 會是 tic-tac-toe。有這麼多的 + 和 " 字元，很難看清楚發生什麼事。如果要完成更複雜的字串組合的話，可以改使用 format! 巨集：

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{s1}-{s2}-{s3}");
```

- 此程式碼一樣能設置 s 為 tic-tac-toe。format! 巨集運作的方式和 println! 類似，但不會將輸出結果顯示在螢幕上，它做的是回傳內容的 String。使用 format! 的程式碼版本看起來比較好讀懂，而且 format! 產生的程式碼使用的是參考，所以此呼叫不會取走任何參數的所有權。

常見集合

索引字串

```
let s1 = String::from("hello");
let h = s1[0];
```

- 在其他程式語言中，使用索引參考字串來取得獨立字元是有效且常見的操作。然而在Rust 中，如果嘗試對 String 使用索引語法的話，會得到錯誤。請看看右上範例這段無效的程式碼：
- 此程式會有以下錯誤結果：

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0277]: the type `String` cannot be indexed by `{integer}`
 --> src/main.rs:3:13
  |
3 |     let h = s1[0];
  |             ^^^^^ `String` cannot be indexed by `{integer}`
  |
  = help: the trait `Index<{integer}>` is not implemented for `String`
  = help: the following other types implement trait `Index<Idx>`:
            <String as Index<RangeFrom<usize>>>
            <String as Index<RangeFull>>
            <String as Index<RangeInclusive<usize>>>
            <String as Index<RangeTo<usize>>>
            <String as Index<RangeToInclusive<usize>>>
            <String as Index<std::ops::Range<usize>>>
            <str as Index<I>>

For more information about this error, try `rustc --explain E0277`.
error: could not compile `collections` due to previous error
```

錯誤訊息與提示告訴了 Rust 字串並不支援索引。但為何不支援呢？要回答此問題，需要先討論 Rust 如何儲存字串進記憶體的。

常見集合

索引字串

- **內部呈現**
  - String 基本上就是 Vec<u8> 的封裝。看看底下中一些正確編碼為 UTF-8 字串的例子，像是這一個：

```
let hello = String::from("Hola");
```

  - 在此例中 len 會是 4，也就是向量儲存的字串「Hola」長度為 4 個位元組。每個字母在用 UTF-8 編碼時長度均為**1個**位元組。但接下來這行可能就會感到驚訝了(請注意字串的開頭是西里爾字母 Ze 的大寫，而不是阿拉伯數字 3)。

```
let hello = String::from("Здравствуйте");
```

  - 可能會以為這字串的長度為 12，事實上 Rust 給的答案卻是 24。這是將「Здравствуйте」用 UTF-8 編碼後的位元組長度，因為該字串的每個Unicode純量都佔據**2個**位元組。**因此字串位元組的索引不會永遠都能對應到有效的Unicode純量數值。**

常見集合

索引字串

- **內部呈現**
  - 用以下無效的 Rust 程式碼進一步說明：

```
let hello = "Здравствуйте";
let answer = &hello[0];
```

  - 已經知道第一個字母 answer 不會是 3。當經過 UTF-8 編碼時，3 的第一個位元組會是 208 然後第二個是 151。所以 answer 實際上會拿到 208，但 208 本身又不是個有效字元。回傳 208 可能不會是使用者想要的，使用者希望的應該是此字串的第一個字母，但這是 Rust 在位元組索引 0 唯一能回傳的資料。就算字串都只包含拉丁字母，使用者通常也不會希望看到位元組數值作為回傳值。如果 &"hello"[0] 是有效程式碼且會回傳位元組數值的話，它會回傳的是 **104** 並非 h。
  - **為了預防回傳意外數值進而導致無法立刻察覺的錯誤，Rust 不會成功編譯這段程式碼，並在開發過程前期就杜絕誤會發生。**

常見集合

索引字串

- **位元組、純量數值與形素群集**
  - UTF-8 還有一個重點是在 Rust 中，實際上可以有三種觀點來理解字串：**位元組、純量數值(scalar values)以及形素群集**(grapheme clusters，最接近人們常說的「字母」)。
  - 如果觀察印度語「नमस्ते」，它存在向量中的 u8 數值就會長這樣：

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164,
224, 165, 135]
```

  - 這 18 個位元組是電腦最終儲存的資料？如果用 Unicode 純量數值觀察的話，也就是 Rust 的 char 型別，這些位元組會組成像這樣：

```
['न', 'म', 'स', '्', 'त', 'े']
```

常見集合

索引字串

- **位元組、純量數值與形素群集**

```
['न', 'म', 'स', '्', 'त', 'े']
```

  - 這邊有六個 char 數值，但第四個和第六個卻不是字母，它們是單獨存在不具任何意義的變音符號。最後如果以形素群集的角度來看的話，就會得到一般人所說的構成此印度語的四個字母：

```
["न", "म", "स्", "ते"]
```

  - Rust 提供多種不同的方式來解釋電腦中儲存的原始字串資料，讓每個程式無論是何種人類語言的資料，都可以選擇它們需要的呈現方式。
  - Rust還有一個不允許索引String來取得字元的原因是因為，**索引運算必須永遠預期是花費常數時間(O(1))**。但在 String 上無法提供這樣的效能保證，因為 Rust 會需要從索引的開頭走訪每個內容才能決定多少有效字元存在。

常見集合

字串切片

- 索引字串通常不是個好點子，因為字串索引要回傳的型別是不明確的，是要一個位元組數值、一個字元、一個形素群集還是一個字串切片呢。因此如果真的想要使用索引建立字串切片的話，Rust 會要更明確些。要明確指定索引與想要的字串切片。
- 與其在 [] 只使用一個數字來索引，可以在 [] 指定一個範圍來建立包含特定位元組**的字串切片：**

```
let hello = "Здравствуйте";

let s = &hello[0..4];
```

- s 在此會是 &str 並包含字串前 4 個位元組。稍早提過這些字元各佔 2 個位元組，所以這裡的 s 就是 3д。

常見集合

```
let hello = "Здравствуйте";

let s = &hello[0..4];
```

字串切片

- **如果嘗試只用 &hello[0..1] 來取得字元部分的位元組的話，Rust 會和在向量中取得無效索引一樣在執行時恐慌：**

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/collections`
thread 'main' panicked at 'byte index 1 is not a char boundary; it is inside 'З' (bytes 0..2)
of `Здравствуйте`', src/main.rs:4:14
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

- 在使用範圍來建立字串切片時要格外小心，因為這樣做有可能會使程式崩潰。

常見集合

走訪字串的方法

- 要對字串的部分進行操作最好的方式是明確表達想要的是**字元還是位元組**。對獨立的 Unicode 純量型別來說的話，就是使用 chars 方法。對「Зд」呼叫 chars 會將兩個擁有 char 型別的數值拆開並回傳，這樣一來就可以走訪每個元素：

```
for c in "Зд".chars() {
    println!("{c}");
}
```

- 此程式碼會顯示以下輸出：

```
З
д
```

- 而 bytes 方法會回傳每個原始位元組，可能會在某些場合適合：

```
for b in "Зд".bytes() {
    println!("{b}");
}
```

- 此程式碼會印出此字串的四個位元組：

```
208
151
208
180
```

請確定已經瞭解有效的 Unicode 純量數值可能不止佔 1 個位元組。

常見集合

透過雜湊映射儲存鍵值配對

- 最後一個常見的集合是雜湊映射(hash map)，HashMap<K, V> 型別會儲存一個鍵(key)型別 K 對應到一個數值(value)型別 V。它透過雜湊函式(hashing function)來決定要將這些鍵與值放在記憶體何處。許多程式語言都有支援這種類型的資料結構，不過通常它們會提供不同的名稱，像是 hash、map、object、hash table、dictionary 或 associative array 等等。
- 雜湊映射適合用於當不想像向量那樣用索引搜尋資料，而是透過一個可以為任意型別的鍵來搜尋的情況。舉例來說，在比賽中可以使用雜湊映射來儲存每隊的分數，每個鍵代表隊伍名稱，而每個值代表隊伍分數。給予一個隊伍名稱，就能取得該隊伍分數。
- 會在此介紹雜湊映射的基本API，還有很多實用的函式定義在標準函式庫的HashMap<K, V> 中，所以別忘了查閱標準函式庫的技術文件來瞭解更多資訊。

常見集合

建立新的雜湊映射

- 其中一種建立空的雜湊映射的方式是使用 new 並透過 insert 加入新元素。在底下範例追蹤兩支隊伍的分數，分別為藍隊與黃隊。藍隊初始分數有 10 分，黃隊則有 50 分：

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

let mut scores = HashMap::new();

scores.insert(String::from("藍隊"), 10);
scores.insert(String::from("黃隊"), 50);
```

- 注意到需要先使用 use 將標準函式庫的 HashMap 集合引入。在介紹的三個常見集合中，此集合是最少被用到的，所以它並沒有包含在 prelude 內能自動參考。雜湊映射也沒有像前者那麼多標準函式庫提供的支援，像是內建建構它們的巨集。
- 和向量一樣，雜湊映射會將它們的資料儲存在堆積上。此 HashMap 的鍵是 String 型別而值是 i32 型別。和向量一樣，雜湊函式宣告後就都得是同類的，所有的鍵都必須是同型別，且所有的值也都必須是同型別。

常見集合

取得雜湊映射的數值

- 可以透過 get 方法並提供鍵來取得其在雜湊映射對應的值，如底下範例所示：

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

let mut scores = HashMap::new();

scores.insert(String::from("藍隊"), 10);
scores.insert(String::from("黃隊"), 50);

let team_name = String::from("藍隊");
let score = scores.get(&team_name).copied().unwrap_or(0);
```

- score 在此將會是對應藍隊的分數，而且結果會是 10。結果是使用 Some 的原因是因為 get 回傳的是 Option<&V>。如果雜湊映射中該鍵沒有對應值的話，get 就會回傳 None。所以程式會需要透過在之前談到的方式處理 Option。

常見集合

取得雜湊映射的數值

- 也可以使用 for 迴圈用類似的方式來走訪雜湊映射中每個鍵值配對：

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

let mut scores = HashMap::new();

scores.insert(String::from("藍隊"), 10);
scores.insert(String::from("黃隊"), 50);

for (key, value) in &scores {
    println!("{key}: {value}");
}
```

- 此程式會以任意順序印出每個配對：

```
黃隊: 50
藍隊: 10
```

常見集合

雜湊映射與所有權

- 像是 i32 這種有實作 Copy 特徵的型別其數值可以被拷貝進雜湊映射之中。但對於像是 String這種擁有所有權的數值則會被**移動**到雜湊映射，並成為該數值新的擁有者，如底下範例所示：

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

let field_name = String::from("Favorite color");
let field_value = String::from("藍隊");

let mut map = HashMap::new();
map.insert(field_name, field_value);
// field_name 和 field_value 在這之後就不能使用了，你可以試著使用它們並看看編譯器回傳什麼錯誤
```

- 之後就無法使用變數 field_name 和 field_value，因為它們的值已經透過呼叫 insert 被移入雜湊映射之中。
- 如果插入雜湊映射的數值是參考的話，該值就不會被移動到雜湊映射之中。不過該值的參考就必須一直有效，至少直到該雜湊映射離開作用域為止。會在之後的「透過生命週期驗證參考」段落討落更多細節。

常見集合

更新雜湊映射

- 雖然鍵值配對的數量可以增加，但每個鍵同一時間就只能有一個對應的值而已。(反之並不成立：比如藍隊黃隊可以同時都在 scores 雜湊映射內儲存 10 分)
- 當想要改變雜湊映射的資料的話，必須決定如何處理當一個鍵已經有一個值的情況。**可以不管舊的值，直接用新值取代；也可以保留舊值、忽略新值，只有在該鍵尚未擁有對應數值時才賦值給它；或者也可以將舊值與新值組合起來。**

常見集合

更新雜湊映射-覆蓋數值

- 如果在雜湊映射插入一個鍵值配對，然後又在相同鍵插入不同的數值的話，該鍵相對應的數值就會被取代。如底下範例雖然呼叫了兩次 insert，但是雜湊映射只會保留一個鍵值配對，因為向藍隊的鍵插入了兩次數值：

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

let mut scores = HashMap::new();

scores.insert(String::from("藍隊"), 10);
scores.insert(String::from("藍隊"), 25);

println!("{:?}", scores);
```

- 此程式碼會印出 {"藍隊": 25}，原本的數值 10 會被覆蓋。

常見集合

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

let mut scores = HashMap::new();

scores.insert(String::from("藍隊"), 10);

scores.entry(String::from("黃隊")).or_insert(50);
scores.entry(String::from("藍隊")).or_insert(50);

println!("{:?}", scores);
```

更新雜湊映射-只在鍵不存在的情況下插入鍵值

- 通常檢查雜湊映射有沒有存在某個特定的鍵值是很常見的。接下來的動作通常就是檢查**如果鍵存在於雜湊映射的話，就不改變其值。但如果鍵不存在的話，就插入數值給它。**
- 雜湊映射提供了一個特別的 API 叫做 **entry** 可以用想要檢查的鍵作為參數。entry 方法的回傳值是一個列舉叫做 Entry，它代表了一個可能存在或不存在的數值。假設想要檢查黃隊的鍵有沒有對應的數值。如果沒有的話，想插入 50。而對藍隊也一樣。使用entry API的話，程式碼會長得像右上範例：
- Entry 中的 **or_insert** 方法定義了如果 Entry 的鍵有對應的數值的話，就回傳該值的可變參考；如果沒有的話，那就插入參數作為新數值，並回傳此值的可變參考。這樣的技巧比親自寫邏輯還清楚，而且更有利於**借用檢查器**的檢查。
- 執行右上範例的程式碼會印出 {"黃隊": 50, "藍隊": 10}。第一次 entry 的呼叫會對黃隊插入數值 50，因為黃隊尚未有任何數值。第二次 entry 的呼叫則不會改變雜湊映射，因為藍隊已經有數值 10。

常見集合

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

let text = "hello world wonderful world";

let mut map = HashMap::new();

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}

println!("{:?}", map);
```

更新雜湊映射-依據舊值更新數值

- 雜湊映射還有另一種常見的用法是，依照鍵的舊數值來更新它。舉例來說，上方範例展示如何計算一些文字內每個單字各出現多少次的程式碼。使用雜湊映射，鍵為單字然後值為每次追蹤計算對應單字出現多少次的次數。如果是第一次看到該單字的話，插入數值 0。
- 此程式碼會印出 {"world": 2, "hello": 1, "wonderful" 1}。可能會看到鍵值配對的順序不太一樣，回想一下「取得雜湊映射的數值」段落中提過走訪雜湊映射的**順序是任意**的。
- split_whitespace 方法會走訪 text 中被空格分開來的切片。or_insert 方法會回傳該鍵對應數值的可變參考(&mut V)。在此將可變參考儲存在 count 變數中，所以要賦值的話，必須先使用 * 來解參考(dereference)count。可變參考會在 for 結束時離開作用域，所以所有的改變都是安全的且符合借用規則。