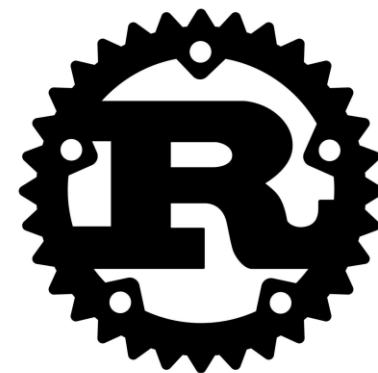


函式語言功能：閉包與疊代器



函式語言功能：閉包與疊代器

Rust 的設計靈感啟發自許多現有的語言與技術，其中一個影響十分顯著的就是函式程式設計(functional programming)。以函式風格的程式設計通常包含將函式視為數值並作為引數傳遞、將它們從其他函式回傳、將它們賦值給變數以便之後使用，以及更多。

在這邊不會討論哪些才是屬於函式程式設計或哪些不是，而是介紹一些 Rust 中類似於許多語言常視為函式語言特色的功能。更明確來說，會涵蓋：

- 閉包(Closures)：類似函式的結構並可以賦值給變數
- 疊代器(Iterators)：走訪一系列元素的方法
- 閉包與疊代器的效能

已經在其他章節提到的功能像是模式配對與列舉也都有被函式風格所影響。因為掌握閉包與疊代器是寫出符合語言風格與高效 Rust 程式碼中重要的一環。

函式語言功能：閉包與疊代器

閉包：獲取其環境的匿名函式

- Rust 的閉包(closures)是個能賦值給變數或作為其他函式引數的匿名函式。可以在某處建立閉包，然後在不同的地方呼叫閉包並執行它。而且不像函式，閉包可以從它們所定義的作用域中獲取數值。將會解釋這些閉包功能如何允許程式碼重用以及自訂行為。



函式語言功能：閉包與疊代器

透過閉包獲取環境

- 首先會來研究如何用閉包來獲取定義在環境的數值並在之後使用。
- 考慮以下假設情境：
 - 每隔一段時間，襯衫公司會送出獨家限量版襯衫給郵寄清單的某位顧客來作為宣傳手段。郵寄清單的顧客可以在設定中加入最愛顏色。如果被選中的人有設定最愛顏色的話，就會獲得該顏色的襯衫。如果沒有指定任何最愛顏色的話，公司就會選擇目前顏色最多的選項。
- 要實作的方式有很多種。舉例來說，可以使用一個列舉叫做 `ShirtColor` 然後其變體有 `Red` 和 `Blue`(為了簡潔限制顏色的種類)。用 `Inventory` 來代表公司的庫存，然後用 `shirts` 欄位來包含 `Vec<ShirtColor>` 來代表目前庫存有的襯衫顏色。
- 定義在 `Inventory` 的 `giveaway` 方法會取得免費襯衫得主的選擇性襯衫顏色偏好，然後回傳會拿到的襯衫顏色。如右邊範例所示：

```
#[derive(Debug, PartialEq, Copy, Clone)]
enum ShirtColor {
    Red,
    Blue,
}

struct Inventory {
    shirts: Vec<ShirtColor>,
}

impl Inventory {
    fn giveaway(&self, user_preference: Option<ShirtColor>) -> ShirtColor {
        user_preference.unwrap_or_else(|| self.most_stocked())
    }

    fn most_stocked(&self) -> ShirtColor {
        let mut num_red = 0;
        let mut num_blue = 0;

        for color in &self.shirts {
            match color {
                ShirtColor::Red => num_red += 1,
                ShirtColor::Blue => num_blue += 1,
            }
        }
        if num_red > num_blue {
            ShirtColor::Red
        } else {
            ShirtColor::Blue
        }
    }
}

fn main() {
    let store = Inventory {
        shirts: vec![ShirtColor::Blue, ShirtColor::Red, ShirtColor::Blue],
    };

    let user_pref1 = Some(ShirtColor::Red);
    let giveaway1 = store.giveaway(user_pref1);
    println!(
        "偏好 {:?} 的使用者獲得 {:?}",
        user_pref1, giveaway1
    );

    let user_pref2 = None;
    let giveaway2 = store.giveaway(user_pref2);
    println!(
        "偏好 {:?} 的使用者獲得 {:?}",
        user_pref2, giveaway2
    );
}
```

函式語言功能：閉包與疊代器

透過閉包獲取環境

- 定義在 main 中的 store 在這次的限量版宣傳中的庫存有兩件藍色襯衫與一件紅色襯衫。呼叫了 giveaway 方法兩次，一次是給偏好紅色襯衫的使用者，另一次則是給無任何偏好的使用者。
- 再次強調這可以用各種方式實作，只是在此想專注在閉包，所以除了用到已經學過的概念以外，**giveaway 方法中還使用了閉包**。
- 在 giveaway 方法中，從參數型別 Option<ShirtColor> 取得使用者偏好然後對 user_preference 呼叫 unwrap_or_else 方法。Option<T> 的 unwrap_or_else 方法定義在標準函式庫中。
 - 它接收一個引數：一個沒有任何引數的閉包然後會回傳數值 T (該型別為 Option<T> 的 Some 儲存的型別，在此例中就是 ShirtColor)。
 - 如果 Option<T> 是 Some 變體，unwrap_or_else 就會回傳 Some 裡的數值。如果 Option<T> 是 None 變體，unwrap_or_else 會呼叫閉包並回傳閉包回傳的數值。

```
#[derive(Debug, PartialEq, Copy, Clone)]
enum ShirtColor {
    Red,
    Blue,
}

struct Inventory {
    shirts: Vec<ShirtColor>,
}

impl Inventory {
    fn giveaway(&self, user_preference: Option<ShirtColor>) -> ShirtColor {
        user_preference.unwrap_or_else(|| self.most_stocked())
    }

    fn most_stocked(&self) -> ShirtColor {
        let mut num_red = 0;
        let mut num_blue = 0;

        for color in &self.shirts {
            match color {
                ShirtColor::Red => num_red += 1,
                ShirtColor::Blue => num_blue += 1,
            }
        }
        if num_red > num_blue {
            ShirtColor::Red
        } else {
            ShirtColor::Blue
        }
    }
}

fn main() {
    let store = Inventory {
        shirts: vec![ShirtColor::Blue, ShirtColor::Red, ShirtColor::Blue],
    };

    let user_pref1 = Some(ShirtColor::Red);
    let giveaway1 = store.giveaway(user_pref1);
    println!(
        "偏好 {:?} 的使用者獲得 {:?}",
        user_pref1, giveaway1
    );

    let user_pref2 = None;
    let giveaway2 = store.giveaway(user_pref2);
    println!(
        "偏好 {:?} 的使用者獲得 {:?}",
        user_pref2, giveaway2
    );
}
```

函式語言功能：閉包與疊代器

透過閉包獲取環境

- 寫上閉包表達式 `|| self.most_stocked()` 作為 `unwrap_or_else` 的引數。這是個沒有任何參數的閉包(如果閉包有參數的話，它們會出現在兩條直線中間)。閉包本體會呼叫 `self.most_stocked()`。
- 直接在此定義閉包，然後 `unwrap_or_else` 的實作就會在需要結果時執行閉包。執行此程式的話就會印出：

```
$ cargo r
Compiling shirt-company v0.1.0 (file:///projects/shirt-company)
Finished dev [unoptimized + debuginfo] target(s) in 0.00s
Running `target/debug/shirt-company`
偏好 Some(Red) 的使用者獲得 Red
偏好 None 的使用者獲得 Blue
```

```
#[derive(Debug, PartialEq, Copy, Clone)]
enum ShirtColor {
    Red,
    Blue,
}

struct Inventory {
    shirts: Vec<ShirtColor>,
}

impl Inventory {
    fn giveaway(&self, user_preference: Option<ShirtColor>) -> ShirtColor {
        user_preference.unwrap_or_else(|| self.most_stocked())
    }

    fn most_stocked(&self) -> ShirtColor {
        let mut num_red = 0;
        let mut num_blue = 0;

        for color in &self.shirts {
            match color {
                ShirtColor::Red => num_red += 1,
                ShirtColor::Blue => num_blue += 1,
            }
        }
        if num_red > num_blue {
            ShirtColor::Red
        } else {
            ShirtColor::Blue
        }
    }
}

fn main() {
    let store = Inventory {
        shirts: vec![ShirtColor::Blue, ShirtColor::Red, ShirtColor::Blue],
    };

    let user_pref1 = Some(ShirtColor::Red);
    let giveaway1 = store.giveaway(user_pref1);
    println!(
        "偏好 {:?} 的使用者獲得 {:?}",
        user_pref1, giveaway1
    );

    let user_pref2 = None;
    let giveaway2 = store.giveaway(user_pref2);
    println!(
        "偏好 {:?} 的使用者獲得 {:?}",
        user_pref2, giveaway2
    );
}
```

函式語言功能：閉包與疊代器

透過閉包獲取環境

- 這裡值得注意的是對當前 `Inventory` 實例傳入的是一個呼叫 `self.most_stocked()` 的閉包。
- 標準函式庫不需要知道定義的任何型別像 `Inventory` 與 `ShirtColor` 或是在此情境中需要使用的任何邏輯，閉包就會獲取 `Inventory` 實例的不可變參考 `self`，然後傳給 `unwrap_or_else` 方法中指定的程式碼。反之，函式就無法像這樣獲取它們周圍的環境。

```
$ cargo r
Compiling shirt-company v0.1.0 (file:///projects/shirt-company)
Finished dev [unoptimized + debuginfo] target(s) in 0.00s
Running `target/debug/shirt-company`
偏好 Some(Red) 的使用者獲得 Red
偏好 None 的使用者獲得 Blue
```

```
#[derive(Debug, PartialEq, Copy, Clone)]
enum ShirtColor {
    Red,
    Blue,
}

struct Inventory {
    shirts: Vec<ShirtColor>,
}

impl Inventory {
    fn giveaway(&self, user_preference: Option<ShirtColor>) -> ShirtColor {
        user_preference.unwrap_or_else(|| self.most_stocked())
    }

    fn most_stocked(&self) -> ShirtColor {
        let mut num_red = 0;
        let mut num_blue = 0;

        for color in &self.shirts {
            match color {
                ShirtColor::Red => num_red += 1,
                ShirtColor::Blue => num_blue += 1,
            }
        }
        if num_red > num_blue {
            ShirtColor::Red
        } else {
            ShirtColor::Blue
        }
    }
}

fn main() {
    let store = Inventory {
        shirts: vec![ShirtColor::Blue, ShirtColor::Red, ShirtColor::Blue],
    };

    let user_pref1 = Some(ShirtColor::Red);
    let giveaway1 = store.giveaway(user_pref1);
    println!(
        "偏好 {:?} 的使用者獲得 {:?}",
        user_pref1, giveaway1
    );

    let user_pref2 = None;
    let giveaway2 = store.giveaway(user_pref2);
    println!(
        "偏好 {:?} 的使用者獲得 {:?}",
        user_pref2, giveaway2
    );
}
```


函式語言功能：閉包與疊代器

閉包型別推導與詮釋

- 函式與閉包還有更多不同的地方。閉包通常不必像 `fn` 函式那樣要求詮釋參數或回傳值的型別。函式需要型別詮釋是因為它們是顯式公開給使用者的介面。嚴格定義此介面是很重要的，這能確保每個人同意函式使用或回傳的數值型別為何。但是閉包並不是為了對外公開使用，它們儲存在變數且沒有名稱能公開給函式庫的使用者。
- 閉包通常很短，而且只與小範圍內的程式碼有關，而非適用於任何場合。有了這樣限制的環境，編譯器能可靠地推導出參數與回傳值的型別，如同其如何推導出大部分的變數型別一樣。(但在有些例外情形下編譯器還是需要閉包的型別詮釋)

函式語言功能：閉包與疊代器

閉包型別推導與詮釋

```
let expensive_closure = |num: u32| -> u32 {  
    println!("緩慢計算中...");  
    thread::sleep(Duration::from_secs(2));  
    num  
};
```

- 至於變數的話，雖然不是必要的，但如果希望能夠增加閱讀性與清楚程度，還是可以加上型別詮釋。要在閉包詮釋型別的話，就會如右上範例的定義所示。在此範例中，定義一個閉包並儲存至一個變數中，而非像之前範例將閉包作為引數傳入：
- 加上型別詮釋後，閉包的語法看起來就更像函式的語法了。在此定義了一個對參數加 1 的函式，以及一個有相同行為的閉包做為比較。加了一些空格來對齊相對應的部分。這顯示了閉包語法和函式語法有多類似，只是改用直線以及有些語法是選擇性的。

```
fn add_one_v1    (x: u32) -> u32 { x + 1 }  
let add_one_v2 = |x: u32| -> u32 { x + 1 };  
let add_one_v3 = |x|           { x + 1 };  
let add_one_v4 = |x|           x + 1 ;
```

函式語言功能：閉包與疊代器

閉包型別推導與詮釋

```
fn add_one_v1 (x: u32) -> u32 { x + 1 }  
let add_one_v2 = |x: u32| -> u32 { x + 1 };  
let add_one_v3 = |x|           { x + 1 };  
let add_one_v4 = |x|           x + 1 ;
```

- 第一行顯示的是函式定義，而第二行則顯示有完成型別詮釋的閉包定義。在第三行移除了閉包定義的型別詮釋，然後在第四行移除大括號，因為閉包本體只有一個表達式，所以是選擇性的。
- 這些都是有效的定義，並會在被呼叫時產生相同行為。而 `add_one_v3` 和 `add_one_v4` 一定要被呼叫，編譯器才能從它們的使用方式中推導出型別。這就像 `let v = Vec::new();` 需要型別詮釋，或是有某種型別的數值插入 `Vec` 中，`Rust` 才能推導出型別。

函式語言功能：閉包與疊代器

閉包型別推導與詮釋

- 對於閉包定義，編譯器會對每個參數與它們的回傳值推導出一個實際型別。
- 舉例來說，右上範例展示只會將收到的參數作為回傳值的閉包定義。此閉包並沒有什麼意義，純粹作為範例解釋。注意到沒有在定義中加上任何型別詮釋。
- 由於沒有型別詮釋，可以用任何型別來呼叫閉包，像第一次呼叫就用 `String`。如果接著嘗試用整數呼叫 `example_closure`，就會得到錯誤。

當第一次使用 `String` 數值呼叫 `example_closure` 時，編譯器會推導 `x` 與閉包回傳值的型別為 `String`。這樣 `example_closure` 閉包內的型別就會鎖定，然後如果對同樣的閉包嘗試使用不同的型別的話，就會得到型別錯誤。

```
let example_closure = |x| x;  
  
let s = example_closure(String::from("哈囉"));  
let n = example_closure(5);
```

```
$ cargo run  
Compiling closure-example v0.1.0 (file:///projects/closure-example)  
error[E0308]: mismatched types  
--> src/main.rs:5:29  
5 |         let n = example_closure(5);  
   |                        ^----- help: try using a conversion method: `.to_string()`  
   |                        |               |  
   |                        |               expected struct `String`, found integer  
   |                        arguments to this function are incorrect  
  
note: closure parameter defined here  
--> src/main.rs:3:28  
2 |         let example_closure = |x| x;  
   |                                ^
```

函式語言功能：閉包與疊代器

獲取參考或移動所有權

- 閉包要從它們周圍環境取得數值有三種方式，這能直接對應於函式取得參數的三種方式：**不可變借用**、**可變借用與取得所有權**。閉包會依照函式本體如何使用獲取的數值，來決定要用哪種方式。
- 在右上範例中，定義一個閉包來獲取list向量的不可變參考，因為它只需要不可變參考就能印出數值：
- 此範例還示範了變數能綁定閉包的定義，然後之後就可以用變數名稱加上括號來呼叫閉包，這樣變數名稱就像函式名稱一樣。由於可以同時擁有 list 的多重不可變參考，list 在閉包定義前、在閉包定義後閉包呼叫前以及閉包呼叫時的程式碼中都是能使用的。此程式碼就會編譯、執行並印出：

```
fn main() {  
    let list = vec![1, 2, 3];  
    println!("定義閉包前：{:?}", list);  
  
    let only_borrows = || println!("來自閉包：{:?}", list);  
  
    println!("呼叫閉包前：{:?}", list);  
    only_borrows();  
    println!("呼叫閉包後：{:?}", list);  
}
```

```
$ cargo run  
Compiling closure-example v0.1.0 (file:///projects/closure-example)  
Finished dev [unoptimized + debuginfo] target(s) in 0.43s  
Running `target/debug/closure-example`  
定義閉包前：[1, 2, 3]  
呼叫閉包前：[1, 2, 3]  
來自閉包：[1, 2, 3]  
呼叫閉包後：[1, 2, 3]
```

函式語言功能：閉包與疊代器

獲取參考或移動所有權

```
fn main() {  
    let mut list = vec![1, 2, 3];  
    println!("呼叫閉包前{:?}", list);  
  
    let mut borrows_mutably = || list.push(7);  
  
    borrows_mutably();  
    println!("呼叫閉包後：{:?}", list);  
}
```

- 接著在右上範例中改變閉包本體，對 list 向量加上一個元素。這樣閉包現在就會獲取**可變參考**：
- 此程式碼會編譯、執行並印出：

```
$ cargo run  
Compiling closure-example v0.1.0 (file:///projects/closure-example)  
Finished dev [unoptimized + debuginfo] target(s) in 0.43s  
Running `target/debug/closure-example`  
呼叫閉包前[1, 2, 3]  
呼叫閉包後：[1, 2, 3, 7]
```

- 注意到在 borrows_mutably 閉包的定義與呼叫之間的 println! 不見了：當 borrows_mutably 定義時它會獲取 list 的可變參考。在閉包呼叫之後沒有再使用閉包，所以可變參考就結束。
- 在閉包定義與呼叫之間，利用不可變參考印出輸出是不允許的，因為在可變參考期間不能再有其他參考。可以試試看在那加上 println! 然後看看會收到什麼錯誤訊息！
- 如果想要強迫閉包取得周圍環境數值的所有權的話，可以在參數列表前使用 **move** 關鍵字。

函式語言功能：閉包與疊代器

獲取參考或移動所有權

```
use std::thread;

fn main() {
    let list = vec![1, 2, 3];
    println!("呼叫閉包前：{:?}", list);

    thread::spawn(move || println!("來自執行緒：{:?}", list))
        .join()
        .unwrap();
}
```

- 此技巧適用於將閉包傳給新執行緒來移動資料，讓新執行緒能擁有該資料。會在之後討論並行時，介紹為何會想使用它們。但現在簡單探索怎麼在閉包使用 `move` 關鍵字開個新的執行緒就好。
- 右上範例更改了之前範例讓向量在新的執行緒印出而非原本的主執行緒：
- 開了一個新的執行緒，將閉包作為引數傳入，閉包本體會印出 `list`。在之前範例中，**閉包只用不可變參考獲取 `list`，因為要印出 `list` 的需求只要這樣就好**。而在此例中，儘管閉包本體仍然只需要不可變參考就好，在閉包定義時想要指定 `list` 應該要透過 `move` 關鍵字移入閉包。
- 新的執行緒可能會在主執行緒之前結束或者主執行緒也有可能先結束。如果主執行緒持有 `list` 的所有權卻在新執行緒之前結束並釋放 `list` 的話，執行緒拿到的不可變參考就會無效了。因此編譯器會要求 `list` 移入新執行緒的閉包中，這樣參考才會有效。
- 嘗試將`move`關鍵字刪掉或是在主執行緒的閉包定義之後使用`list`，看看會收到什麼編譯器錯誤訊息！

函式語言功能：閉包與疊代器

Fn 特徵以及將獲取的數值移出閉包

- 一旦閉包從其定義的周圍環境獲取數值的參考或所有權(也就是說被移入閉包中)，閉包本體的程式碼會定義閉包在執行結束後要對參考或數值做什麼事情(也就是說被移出閉包)。閉包本體可以做以下的事情：將獲取的數值移出閉包、改變獲取的數值、不改變且不移動數值，或是一開始就不從環境獲取任何值。
- 閉包從周圍環境獲取並處理數值的方式會影響閉包會實作哪種特徵，而這些特徵能讓函式與結構體決定它們想使用哪種閉包。**閉包會依照閉包本體處理數值的方式，自動實作一種或多種 Fn 特徵：**
 - FnOnce 適用於**可以呼叫一次的閉包**。所有閉包至少都會有此特徵，因為所有閉包都能被呼叫。**會將獲取的數值移出本體的閉包只會實作 FnOnce 而不會再實作其他 Fn 特徵**，因為這樣它只能被呼叫一次。
 - FnMut 適用於**不會將獲取數值移出本體**，而且**可能會改變獲取數值的閉包**。這種閉包**可以被呼叫多次**。
 - Fn 適用於**不會將獲取數值移出本體**，而且**不會改變獲取數值或是甚至不從環境獲取數值的閉包**。這種閉包**可以被呼叫多次**，而且不會改變周圍環境，這對於並行呼叫閉包多次來說非常重要。

函式語言功能：閉包與疊代器

Fn 特徵以及將獲取的數值移出閉包

- 來觀察右邊範例中Option<T>用到的unwrap_or_else 方法定義：
- 回想一下 T 是一個泛型型別，代表著 Option 的 Some 變體內的數值型別。型別 T 同時也是函式 unwrap_or_else 的回傳型別：比如說對 Option<String> 呼叫 unwrap_or_else 的話就會取得 String。
- 接著注意到函式 unwrap_or_else 有個額外的泛型型別參數 F。型別 F 是參數 f 的型別，也正是當呼叫 unwrap_or_else 時的閉包。
- 泛型型別 F 指定的特徵界限是 FnOnce() -> T，也就是說F必須要能夠呼叫一次、不帶任何引數然後回傳T。**特徵界限中使用FnOnce限制了unwrap_or_else只能呼叫f最多一次**。在unwrap_or_else本體中，如果 Option 是 Some 的話，f 就不會被呼叫。如果 Option 是 None 的話，f 就會被呼叫一次。由於所有閉包都有實作 FnOnce，unwrap_or_else 能接受大多數各種不同的閉包，讓它的用途非常彈性。
- 注意：函式也可以實作這三種 Fn 特徵。如果不必獲取環境數值，在需要有實作其中一種 Fn 特徵的項目時，可以使用函式名稱而不必用到閉包。舉例來說，對於 Option<Vec<T>> 的數值，可以呼叫 unwrap_or_else(Vec::new) 在數值為 None 時取得新的空向量。

```
impl<T> Option<T> {  
    pub fn unwrap_or_else<F>(self, f: F) -> T  
    where  
        F: FnOnce() -> T  
    {  
        match self {  
            Some(x) => x,  
            None => f(),  
        }  
    }  
}
```

函式語言功能：閉包與疊代器

Fn 特徵以及將獲取的數值移出閉包

- 現在來看看標準函式庫中切片定義的 `sort_by_key` 方法，來觀察它和 `unwrap_or_else` 有什麼不同以及為何 `sort_by_key` 的特徵界限使用的是 `FnMut` 而不是 `FnOnce`。
- 閉包會取得一個引數，這會是該切片當下項目的參考，然後回傳型別 `K` 的數值以供排序。當想透過切片項目的特定屬性做排序時，此函式會很實用。
- 右上範例有個 `Rectangle` 實例的列表，然後使用 `sort_by_key` 透過 `width` 屬性由低至高排序它們：

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    list.sort_by_key(|r| r.width);
    println!("{:?}", list);
}
```

函式語言功能：閉包與疊代器

Fn 特徵以及將獲取的數值移出閉包

- 此程式碼會印出：

```
$ cargo run
Compiling rectangles v0.1.0 (file:///projects/rectangles)
Finished dev [unoptimized + debuginfo] target(s) in 0.41s
Running `target/debug/rectangles`
[
  Rectangle {
    width: 3,
    height: 5,
  },
  Rectangle {
    width: 7,
    height: 12,
  },
  Rectangle {
    width: 10,
    height: 1,
  },
]
```

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    list.sort_by_key(|r| r.width);
    println!("{:?}", list);
}
```

- `sort_by_key` 的定義會需要 **FnMut** 閉包的原因是因為它得呼叫閉包好幾次，對切片的每個項目都要呼叫一次。
- 閉包 `|r| r.width` 沒有獲取、改變或移動周圍環境的任何值，所以它符合特徵界限的要求。

函式語言功能：閉包與疊代器

Fn 特徵以及將獲取的數值移出閉包

- 此反之，底下範例示範了一個只實作 FnOnce 特徵的閉包，**因為它有將數值移出環境**。
- 編譯器不會允許將此閉包用在 sort_by_key：

這裡嘗試用很糟糕且令人費解的方式計算 list 在排序時 sort_by_key 被呼叫了幾次。此程式碼嘗試計數的方式是把閉包周圍環境中型別為 String 的 value 變數放入 sort_operations 向量中。

閉包會獲取 value，然後將 value 移出閉包，也就是將 value 的所有權轉移到 sort_operations 向量裡。此向量只能呼叫一次，嘗試呼叫第二次是無法成功的，因為 value 已經不存在於環境中了，無法再次放入 sort_operations！因此，**此閉包僅實作了 FnOnce**。

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut sort_operations = vec![];
    let value = String::from("by key called");

    list.sort_by_key(|r| {
        sort_operations.push(value);
        r.width
    });
    println!("{:?}", list);
}
```

函式語言功能：閉包與疊代器

Fn 特徵以及將獲取的數值移出閉包

- 當嘗試編譯此程式碼時，會收到錯誤訊息說明 `value` 無法移出閉包，因為閉包必須實作 `FnMut`：

```
$ cargo run
  Compiling rectangles v0.1.0 (/Users/wuwayne/Desktop/book-tw/listings/ch13-functional-features/
  listing-13-08)
error[E0507]: cannot move out of `value`, a captured variable in an `FnMut` closure
  --> src/main.rs:18:30

15 |         let value = String::from("by key called");
    |         ----- captured outer variable
16 |
17 |         list.sort_by_key(|r| {
    |         --- captured by this `FnMut` closure
18 |             sort_operations.push(value);
    |                               ^^^^^ move occurs because `value` has type `String`, which does not
    |                               implement the `Copy` trait
```

錯誤訊息指出閉包本體將 `value` 移出環境的地方。要修正此問題的話，需要改變閉包本體，讓它不再將數值移出環境。

要計算 `sort_by_key` 呼叫次數的話，在環境中放置一個計數器，然後在閉包本體增加其值是更直觀的計算方法。

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut sort_operations = vec![];
    let value = String::from("by key called");

    list.sort_by_key(|r| {
        sort_operations.push(value);
        r.width
    });
    println!("{:?}", list);
}
```

函式語言功能：閉包與疊代器

Fn 特徵以及將獲取的數值移出閉包

- 底下範例的閉包就能用在 `sort_by_key`，因為它只獲取了 `num_sort_operations` 計數器的可變參考，因此可以被呼叫不只一次：

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut num_sort_operations = 0;
    list.sort_by_key(|r| {
        num_sort_operations += 1;
        r.width
    });
    println!("{::#?} 的排序經過 {num_sort_operations} 次運算", list);
}
```

- 當要在函式或型別中定義與使用閉包時，**Fn** 特徵是很重要的。在下個段落中，將討論疊代器。疊代器有許多方法都需要閉包引數。

函式語言功能：閉包與疊代器

使用疊代器來處理一系列的項目

- 疊代器(Iterator)模式可以對一個項目序列依序進行某些任務。疊代器的功用是走訪序列中每個項目，並決定該序列何時結束。當使用疊代器，不需要自己實作這些邏輯。
- 在 Rust 中疊代器是惰性(lazy)的，代表除非呼叫方法來使用疊代器，不然它們不會有任何效果。舉例來說，底下範例的程式碼會透過 Vec<T> 定義的方法 iter 從向量v1 建立一個疊代器來走訪它的項目。此程式碼本身沒有啥實用之處：

```
let v1 = vec![1, 2, 3];  
let v1_iter = v1.iter();
```

- 疊代器儲存在變數 v1_iter 中。一旦建立了疊代器，可以有很多使用它的方式。在之前範例中，在 for 迴圈中使用疊代器來對每個項目執行一些程式碼。在過程中這就隱性建立並使用了一個疊代器，雖然當時沒有詳細解釋細節。

函式語言功能：閉包與疊代器

使用疊代器來處理一系列的項目

```
let v1 = vec![1, 2, 3];  
let v1_iter = v1.iter();  
for val in v1_iter {  
    println!("取得: {}", val);  
}
```

- 在右上範例中，區隔了疊代器的建立與使用疊代器 for 迴圈。當使用 v1_iter 疊代器的 for 迴圈被呼叫時，疊代器中的每個元素才會在迴圈中每次疊代中使用，以此印出每個數值：
- 在標準函式庫沒有提供疊代器的語言中，可能會用別種方式寫這個相同的函式，像是先從一個變數 0 作為索引開始、使用該變數索引向量來獲取數值，然後在迴圈中增加變數的值直到它抵達向量的總長。
- 疊代器會處理這些所有邏輯，減少重複且可能會搞砸的程式碼。疊代器還能靈活地將相同的邏輯用於不同的序列，而不只是像向量這種能進行索引的資料結構。研究看看疊代器怎麼辦到的。

函式語言功能：閉包與疊代器

Iterator 特徵與 next 方法

```
pub trait Iterator {  
    type Item;  
  
    fn next(&mut self) -> Option<Self::Item>;  
  
    // 以下省略預設實作  
}
```

- 所有的疊代器都會實作定義在標準函式庫的 `Iterator` 特徵。特徵的定義如右上所示：
- 注意到此定義使用了一些新的語法：`type Item` 與 `Self::Item`，這是此特徵定義的關聯型別 (associated type)。會在之後探討關聯型別。現在只需要知道此程式碼表示要實作 **Iterator** 特徵的話，還需要定義 **Item** 型別，而此 **Item** 型別會用在方法 `next` 的回傳型別中。換句話說，`Item` 型別會是從疊代器回傳的型別。
- `Iterator` 型別只要求實作者定義一個方法：`next` 方法會用 `Some` 依序回傳疊代器中的每個項目，並在疊代器結束時回傳 `None`。

函式語言功能：閉包與疊代器

Iterator 特徵與 next 方法

```
#[test]
fn iterator_demonstration() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

- 可以直接在疊代器呼叫 `next` 方法。右上範例展示從向量建立的疊代器重複呼叫 `next` 每次會得到什麼數值：
- 注意到 `v1_iter` 需要是可變的：在疊代器上呼叫 `next` 方法會改變疊代器內部用來紀錄序列位置的狀態。換句話說，此程式碼消耗或者說使用了疊代器。每次 `next` 的呼叫會從疊代器消耗一個項目。而不必在 `for` 迴圈指定 `v1_iter` 為可變是因為迴圈會取得 `v1_iter` 的所有權並在內部將其改為可變。
- 另外還要注意的是從 `next` 呼叫取得的是向量中數值的不可變參考：
 - **`iter` 方法會從疊代器中產生不可變參考。**
 - 如果想要一個取得 `v1` 所有權的疊代器，可以呼叫 **`into_iter`** 而非 `iter`。
 - 同樣地，如果想要走訪可變參考，可以呼叫 **`iter_mut`** 而非 `iter`。

函式語言功能：閉包與疊代器

消耗疊代器的方法

```
#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

- 標準函式庫提供的 `Iterator` 特徵有一些不同的預設實作方法，可以查閱標準函式庫的 `Iterator` 特徵 API 技術文件來找到這些方法。其中有些方法就是在它們的定義呼叫 `next` 方法，這就是為何當實作 `Iterator` 特徵時需要提供 `next` 方法的實作。
- 會呼叫 `next` 的方法被稱之為消耗配接器(consuming adaptors)，因為呼叫它們會使用掉疊代器。其中一個例子就是方法 `sum`，這會取得疊代器的所有權並重複呼叫 `next` 來走訪所有項目，因而消耗掉疊代器。隨著走訪的過程中，會將每個項目加到總計中，並在疊代完成時回傳總計數值。右上範例展示了一個使用 `sum` 方法的測試：
- 呼叫 `sum` 之後就不再被允許使用 `v1_iter` 了，因為 `sum` 取得了疊代器的所有權。

函式語言功能：閉包與疊代器

產生其他疊代器的方法

```
let v1: Vec<i32> = vec![1, 2, 3];  
v1.iter().map(|x| x + 1);
```

- 疊代配接器(iterator adaptors)是定義在 `Iterator` 特徵的方法，它們不會消耗掉疊代器。它們會改變原本疊代器的一些屬性來產生不同的疊代器。
- 右上範例呼叫了疊代器的疊代配接器方法 `map`，它會取得一個閉包在進行疊代時對每個項目進行呼叫。`map` 方法會回傳個項目被改變過的新疊代器。這裡的閉包會將向量中的每個項目加 1 來產生新的疊代器：
- 不過此程式碼會產生個警告：

右上範例的程式碼不會做任何事情，指定的閉包沒有被呼叫到半次。警告提醒了原因：疊代配接器是惰性的，必須在此消耗疊代器才行。要修正並消耗此疊代器，將使用 `collect` 方法，此方法會消耗疊代器並收集結果數值至一個資料型別集合。

```
$ cargo run  
  Compiling iterators v0.1.0 (file:///projects/iterators)  
warning: unused `Map` that must be used  
--> src/main.rs:4:5  
4 |         v1.iter().map(|x| x + 1);  
  |         ^^^^^^^^^^^^^^^^^^^^^^^^^  
  
= note: `#[warn(unused_must_use)]` on by default  
= note: iterators are lazy and do nothing unless consumed  
  
warning: `iterators` (bin "iterators") generated 1 warning  
Finished dev [unoptimized + debuginfo] target(s) in 0.47s  
Running `target/debug/iterators`
```

函式語言功能：閉包與疊代器

產生其他疊代器的方法

```
let v1: Vec<i32> = vec![1, 2, 3];  
let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();  
assert_eq!(v2, vec![2, 3, 4]);
```

- 在右上範例中，將走訪 `map` 呼叫所產生的疊代器結果數值收集到一個向量中。此向量最後會包含原本向量每個項目都加 1 的數值：
- 因為 `map` 接受一個閉包，可以對每個項目指定任何想做的動作。這是一個展示如何使用閉包來自訂行為，同時又能重複使用 `Iterator` 特徵提供的走訪行為的絕佳例子。
- 可以透過疊代配接器串連多重呼叫，在進行一連串複雜運算的同時，仍保持良好的閱讀性。但因為所有的疊代器都是惰性的，必須呼叫能消耗配接器的方法來取得疊代配接器的結果。

函式語言功能：閉包與疊代器

使用閉包獲取它們的環境

- 許多疊代配接器都會拿閉包作為引數，而通常向疊代配接器指定的閉包引數都能獲取它們周圍的環境。
- 右邊例子使用 `filter` 方法來取得閉包。閉包會取得疊代器的每個項目並回傳布林值。如果閉包回傳 `true`，該數值就會被包含在 `filter` 產生的疊代器中；如果閉包回傳 `false`，該數值就不會被包含在結果疊代器中。
- 在右邊範例中使用 `filter` 與一個從它的環境獲取變數 `shoe_size` 的閉包來走訪一個有 `Shoe` 結構體實例的集合。它會回傳只有符合指定大小的鞋子：

```
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn filters_by_size() {
        let shoes = vec![
            Shoe {
                size: 10,
                style: String::from("運動鞋"),
            },
            Shoe {
                size: 13,
                style: String::from("涼鞋"),
            },
            Shoe {
                size: 10,
                style: String::from("靴子"),
            },
        ];

        let in_my_size = shoes_in_size(shoes, 10);

        assert_eq!(
            in_my_size,
            vec![
                Shoe {
                    size: 10,
                    style: String::from("運動鞋")
                },
                Shoe {
                    size: 10,
                    style: String::from("靴子")
                },
            ]
        );
    }
}
```


函式語言功能：閉包與疊代器

使用閉包獲取它們的環境

- 函式 `shoes_in_size` 會取得鞋子向量的所有權以及一個鞋子大小作為參數。它會回傳只有符合指定大小的鞋子向量。
- 在 `shoes_in_size` 的本體中，呼叫 `into_iter` 來建立一個會取得向量所有權的疊代器。然後呼叫 `filter` 來將該疊代器轉換成只包含閉包回傳為 `true` 的元素的新疊代器。
- 閉包會從環境獲取 `shoe_size` 參數並比較每個鞋子數值的大小，讓只有符合大小的鞋子保留下來。最後呼叫 `collect` 來收集疊代器回傳的數值進一個函式會回傳的向量。
- 此測試顯示了當呼叫 `shoes_in_size` 時，會得到指定相同大小的鞋子。

```
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn filters_by_size() {
        let shoes = vec![
            Shoe {
                size: 10,
                style: String::from("運動鞋"),
            },
            Shoe {
                size: 13,
                style: String::from("涼鞋"),
            },
            Shoe {
                size: 10,
                style: String::from("靴子"),
            },
        ];

        let in_my_size = shoes_in_size(shoes, 10);

        assert_eq!(
            in_my_size,
            vec![
                Shoe {
                    size: 10,
                    style: String::from("運動鞋")
                },
                Shoe {
                    size: 10,
                    style: String::from("靴子")
                },
            ]
        );
    }
}
```

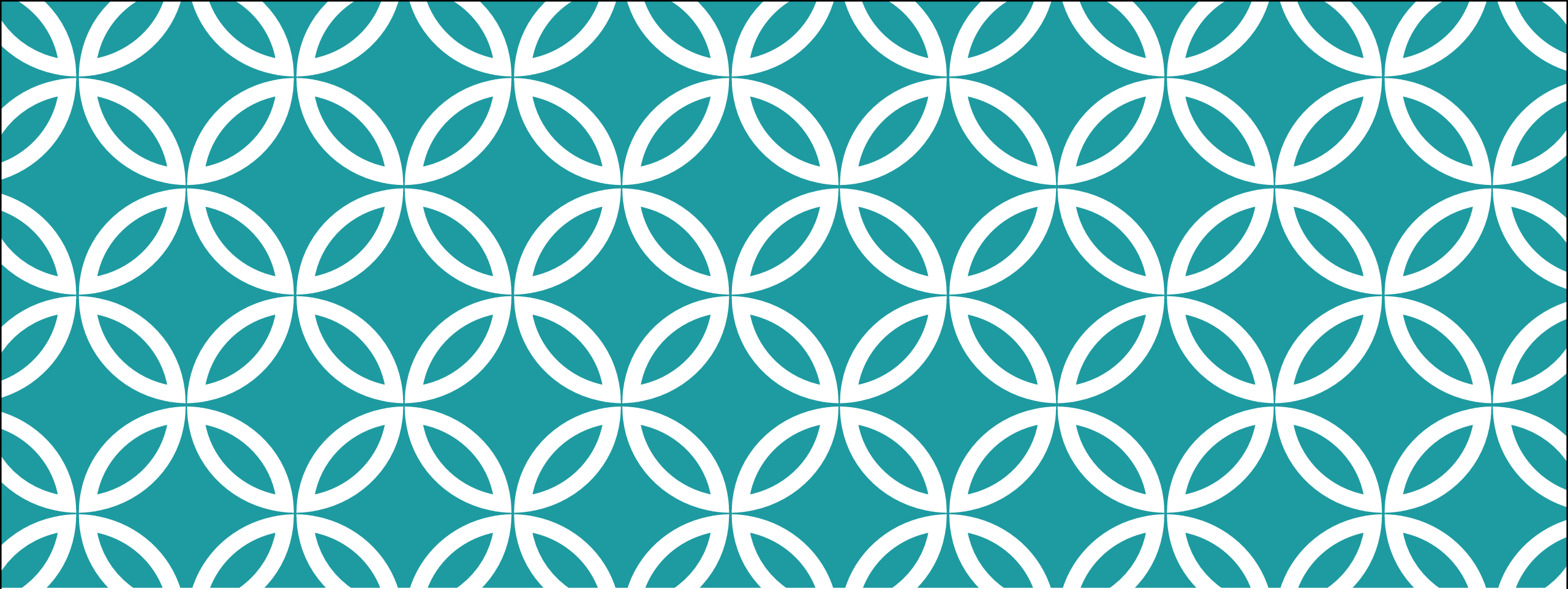
函式語言功能：閉包與疊代器

比較效能：迴圈 vs. 疊代器

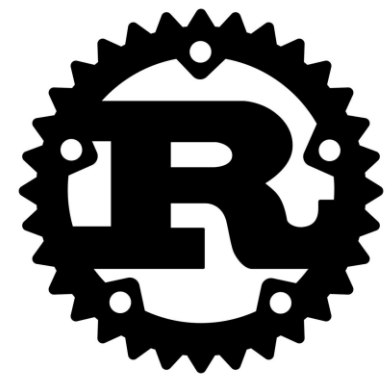
- 為了決定該使用迴圈還是疊代器，需要知道哪個實作比較快：是顯式 for 迴圈的版本，還是疊代器的版本。可以透過讀取整本 Sir Arthur Conan Doyle 寫的 The Adventures of Sherlock Holmes 到 String 中並搜尋內容中的 the 來進行評測。以下為針對 search 函式使用 for 迴圈與使用疊代器的版本評測(benchmark)：

```
test bench_search_for ... bench: 19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench: 19,234,900 ns/iter (+/- 657,200)
```

- 疊代器版本竟然比較快一些！要做更全面的評測，應該要檢查使用不同大小的不同文字來作為 contents、不同單字與不同長度來作為 query，以及所有各式各樣的可能性。
- 這邊的重點在於：疊代器雖然是高階抽象，但其編譯出來的程式碼與親自寫出低階的程式碼幾乎相同。
- 疊代器是 Rust 其中一種零成本抽象(zero-cost abstractions)**，這指的是使用的抽象不會在執行時有額外開銷(零開銷(zero-overhead))。零開銷的原則：沒有使用到的話，就不必買單。而且有使用到的話，不可能再寫出更好的程式碼。



Cargo 與 Crates.io



Cargo 與 Crates.io

更多關於 Cargo 與 Crates.io 的內容

- 目前只使用了 Cargo 最基本的功能來建構、執行與測試程式碼，但它還能做更多事。在此將討論這些其他的進階功能，將瞭解如何做到以下動作：
 - 透過發佈設定檔來自訂建構
 - 發佈函式庫到 crates.io
 - 透過工作空間組織大型專案
 - 從 crates.io 安裝執行檔
 - 使用自訂命令擴展 Cargo 的功能

Cargo 與 Crates.io

透過發佈設定檔來自訂建構

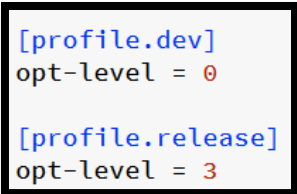
- 在 Rust 中發佈設定檔(release profiles)是個預先定義好並可用不同配置選項來自訂的設定檔，能讓程式設計師掌控更多選項來編譯程式碼。每個設定檔的配置彼此互相獨立。
- Cargo 有兩個主要的設定檔：
 - dev 設定檔會在當對 Cargo 執行 `cargo build` 時所使用；
 - release 設定檔會在當對 Cargo 執行 `cargo build --release` 時所使用。
- dev 設定檔預設定義為適用於開發時使用，而release 設定檔預設定義為適用於發佈時使用。
- 可能會覺得這些設定檔名稱很眼熟，因為它們就已經顯示在輸出結果過：

dev 與 release 是編譯器會使用到的不同設定檔。

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
$ cargo build --release
    Finished release [optimized] target(s) in 0.0s
```

Cargo 與 Crates.io

透過發佈設定檔來自訂建構

- 當專案的 Cargo.toml 沒有顯式加上任何 [profile.*] 段落的話，Cargo 就會使用每個設定檔的預設設置。透過想要自訂的任何設定檔加上 [profile.*] 段落，可以覆寫任何預設設定的子集。
- 舉例來說，以下是 dev 與 release 設定檔中 opt-level 設定的預設數值：

```
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```
- opt-level 設定控制了 Rust 對程式碼進行優化的程度，範圍從 0 到 3。提高優化程度會增加編譯時間，所以如果在開發過程中得時常編譯程式碼的話，傾向於編譯快一點而不管優化的多寡，就算結果程式碼會執行的比較慢。
- 這就是 dev 的 opt-level 預設為 0 的原因。當準備好要發佈程式碼時，則最好花多點時間來編譯。只需要在發佈模式編譯一次，但編譯程式則會被執行很多次，所以發佈模式選擇花費多點編譯時間來讓程式跑得比較快。這就是 release 的 opt-level 預設為 3 的原因。

Cargo 與 Crates.io

透過發佈設定檔來自訂建構

- 可以在 Cargo.toml 加上不同的數值來覆蓋預設設定。舉例來說，如果希望在開發設定檔使用優化等級 1 的話，可以在專案的 Cargo.toml 檔案中加上這兩行：

```
[profile.dev]  
opt-level = 1
```

- 這樣就會覆蓋預設設定 0。現在當執行 `cargo build`，Cargo 就會使用 dev 設定檔的預設值以及自訂的 `opt-level`。因為將 `opt-level` 設為 1，Cargo 會比原本的預設進行更多優化，但沒有發佈建構那麼多。
- 對於完整的設置選項與每個設定檔的預設列表，請查閱 [Cargo 的技術文件](#)。

Cargo 與 Crates.io

發佈 Crate 到 Crates.io

- 已經使用過 crates.io 的套件來作為專案的依賴函式庫，但是也可以發佈自己的套件來將程式碼提供給其他人使用。crates.io 會發行套件的原始碼，所以它主要用來託管開源程式碼。
- Rust 與 Cargo 有許多功能可以幫助其他人更容易找到並使用發佈的套件。會介紹其中一些功能並解釋如何發佈套件。

寫上有幫助的技術文件註解

- 準確地加上套件的技術文件有助於其他使用者知道如何及何時使用它們，所以投資時間在寫技術文件上是值得的。在前面提過如何使用兩條斜線 // 來加上 Rust 程式碼註解。
- Rust 還有個特別的註解用來作為技術文件，俗稱為技術文件註解 (documentation comment)，這能用來產生 HTML 技術文件。這些 HTML 顯示公開 API 項目中技術文件註解的內容，讓對此函式庫有興趣的開發者知道如何使用 crate，而不需知道 crate 是如何實作的。

Cargo 與 Crates.io

寫上有幫助的技術文件註解

- 技術文件註解使用**三條斜線 ///** 而不是兩條，並支援 Markdown 符號來格式化文字。技術文件註解位於它們對應項目的上方。底下範例顯示了 `my_crate` crate 中 `add_one` 的技術文件註解：

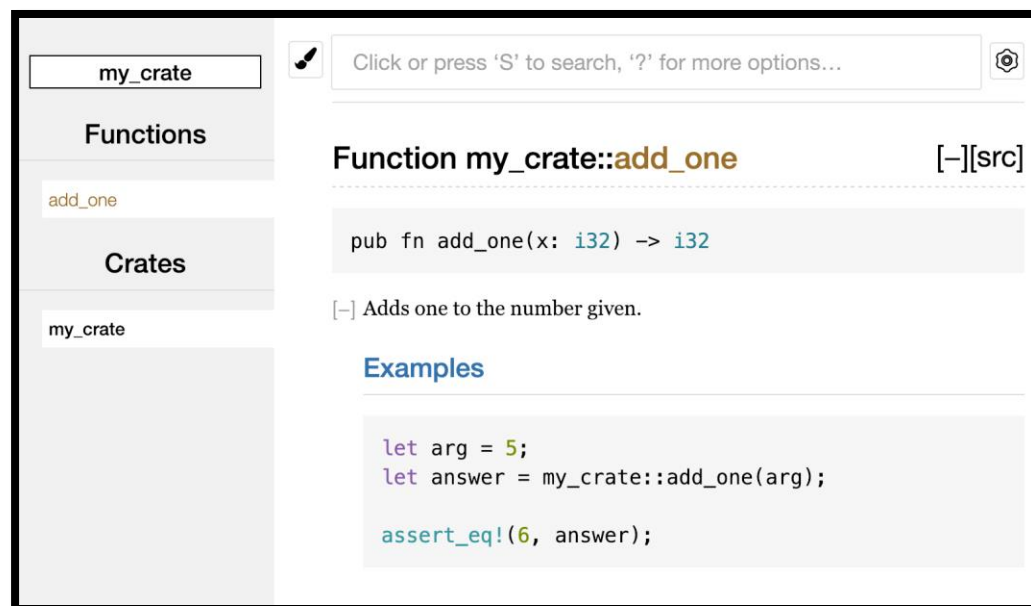
```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let arg = 5;
/// let answer = my_crate::add_one(arg);
///
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

在這裡加上了解釋函式 `add_one` 行為的描述、加上一個標題為 **Examples** 的段落並附上展示如何使用 `add_one` 函式的程式碼。可以透過執行 `cargo doc` 來從技術文件註解產生 HTML 技術文件。此命令會執行隨著 Rust 一起發佈的工具 `rustdoc`，並在 `target/doc` 目錄下產生 HTML 技術文件。

Cargo 與 Crates.io

寫上有幫助的技術文件註解

- 為了方便起見，可以執行 `cargo doc --open` 來建構當前 crate 的 HTML 技術文件(以及 crate 所有依賴的技術文件)並在網頁瀏覽器中開啟結果。
- 導向到函式 `add_one` 就能看到技術文件註解是如何呈現的，如底下圖示所示：



Cargo 與 Crates.io

寫上有幫助的技術文件註解

- 常見技術文件段落

- 在上面範例使用 `# Examples` Markdown 標題來在 HTML 中建立一個標題為「Examples」的段落。以下是 crate 技術文件中常見的段落標題：
- **Panics**：該函式可能會導致恐慌的可能場合。函式的呼叫者不希望他們的程式恐慌的話，就要確保沒有在這些情況下呼叫該函式。
- **Errors**：如果函式回傳 `Result`，解釋發生錯誤的可能種類以及在何種條件下可能會回傳這些錯誤有助於呼叫者，讓他們可以用不同方式來寫出處理不同種錯誤的程式碼。
- **Safety**：如果呼叫的函式是 `unsafe` 的話(會在之後討論不安全的議題)，就必須要有個段落解釋為何該函式是不安全的，並提及函式預期呼叫者要確保哪些不變條件(invariants)。
- 大多數的技術文件註解不全都需要這些段落，但這些可能是使用者有興趣瞭解的內容，可以作為提醒的檢查列表。

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let arg = 5;
/// let answer = my_crate::add_one(arg);
///
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

Cargo 與 Crates.io

寫上有幫助的技術文件註解

- 將技術文件註解作為測試
 - 在技術文件註解加上範例程式碼區塊有助於解釋如何使用函式庫，而且這麼做還有個額外好處：執行 `cargo test` 也會將技術文件視為測試來執行！在技術文件加上範例的確是最佳示範，但是如果程式碼在技術文件寫完之後變更的話，該範例可能就會無法執行了。
 - 如果對之前範例中有附上技術文件的函式 `add_one` 執行 `cargo test` 的話，會看見測試結果有以下這樣的段落：

```
Doc-tests my_crate

running 1 test
test src/lib.rs - add_one (line 5) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.27s
```

- 現在如果變更函式或範例使其內的 `assert_eq!` 會恐慌並再次執行 `cargo test` 的話，會看到技術文件測試能互相獲取錯誤，告訴範例與程式碼已經不同步了！

Cargo 與 Crates.io

寫上有幫助的技術文件註解

- 包含項目結構的註解
 - 風格為 `//!` 技術文件註解會對其包含該註解的項目加上的技術文件，而不是對註解後的項目加上技術文件。通常將此技術文件註解用於 `crate` 源頭檔(通常為 `src/lib.rs`)或模組來對整個 `crate` 或模組加上技術文件。
 - 舉例來說，如果希望能加上技術文件來描述包含 `add_one` 函式的 `my_crate` 目的，可以用 `//!` 在 `src/lib.rs` 檔案開頭加上技術文件註解，如底下範例所示：

```
//! # My Crate
//!
//! `my_crate` is a collection of utilities to make performing certain
//! calculations more convenient.

/// Adds one to the number given.
// --省略--
```

Cargo 與 Crates.io

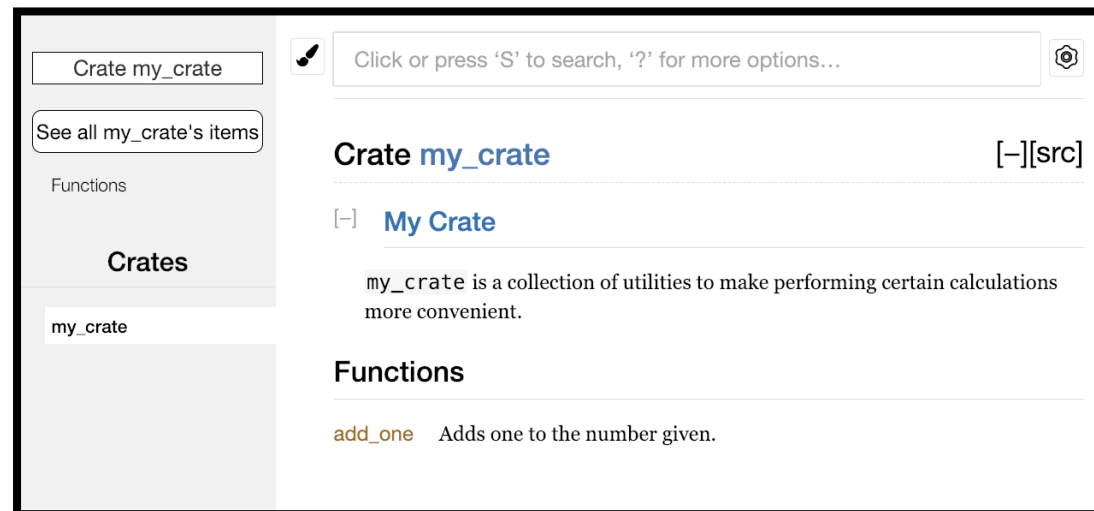
寫上有幫助的技術文件註解

- 包含項目結構的註解
- 注意到 `///` 最後一行之後並沒有緊貼任何程式碼，因為是用 `///` 而非 `///` 來下註解，是對包含此註解的整個項目加上技術文件，而不是此註解之後的項目。在此例中，該項目就是 `src/lib.rs` 檔案，也就是 `crate` 的源頭。這些註解會描述整個 `crate`。當執行 `cargo doc --open`，這些註解會顯示在 `my_crate` 技術文件的首頁，位於 `crate` 公開項目列表的上方，如下面圖示所示：

```
///! # My Crate
///!
///! `my_crate` is a collection of utilities to make performing certain
///! calculations more convenient.

/// Adds one to the number given.
// --省略--
```

項目中的技術文件註解可以用來分別描述 `crate` 和模組。用它們來將解釋容器整體的目的有助於使用者瞭解該 `crate` 的程式碼組織架構。



Cargo 與 Crates.io

透過 `pub use` 匯出理想的公開 API

- 公開 API 的架構是發佈 crate 時要考量到的一大重點。使用 crate 的人可能並沒有那麼熟悉其中的架構，而且如果 crate 模組分層越深的話，可能就難以找到想使用的部分。在前面章節中，介紹了如何使用 `mod` 關鍵字來組織程式碼成模組、如何使用 `pub` 關鍵字來公開項目，以及如何使用 `use` 關鍵字來將項目引入作用域。
- 然而在開發 crate 時的架構雖然來說是合理的，但對使用者來說可能就不是那麼合適了。可能會希望用有數個層級的分層架構來組織程式碼，但是要是有人想使用定義在分層架構裡的型別時，它們可能就很難發現這些型別的存在。
- 而且輸入 `use my_crate::some_module::another_module::UsefulType;` 是非常惱人的，會希望輸入 `use my_crate::UsefulType;` 就好。

Cargo 與 Crates.io

透過 pub use 匯出理想的公開 API

- 好消息是如果架構不便於其他函式庫所使用的話，不必重新組織內部架構：可以透過使用 `pub use` 選擇重新匯出(re-export)項目來建立一個不同於內部私有架構的公開架構。重新匯出會先取得某處公開項目，再從其他地方使其公開，讓它像是被定義在其他地方一樣。
- 舉例來說，建立了一個函式庫叫做 `art` 來模擬藝術概念。在函式庫中有兩個模組：`kinds` 模組包含兩個列舉 `PrimaryColor` 和 `SecondaryColor`；而 `utils` 模組包含一個函式 `mix`，如底下範例所示：
(src/lib.rs)

```
///! # Art
///!
///! A library for modeling artistic concepts.

pub mod kinds {
    /// The primary colors according to the RYB color model.
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// The secondary colors according to the RYB color model.
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use crate::kinds::*;

    /// Combines two primary colors in equal amounts to create
    /// a secondary color.
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --省略--
    }
}
```

Cargo 與 Crates.io

透過 pub use 匯出理想的公開 API

- 下圖顯示了此 crate 透過 cargo doc 產生的技術文件首頁：

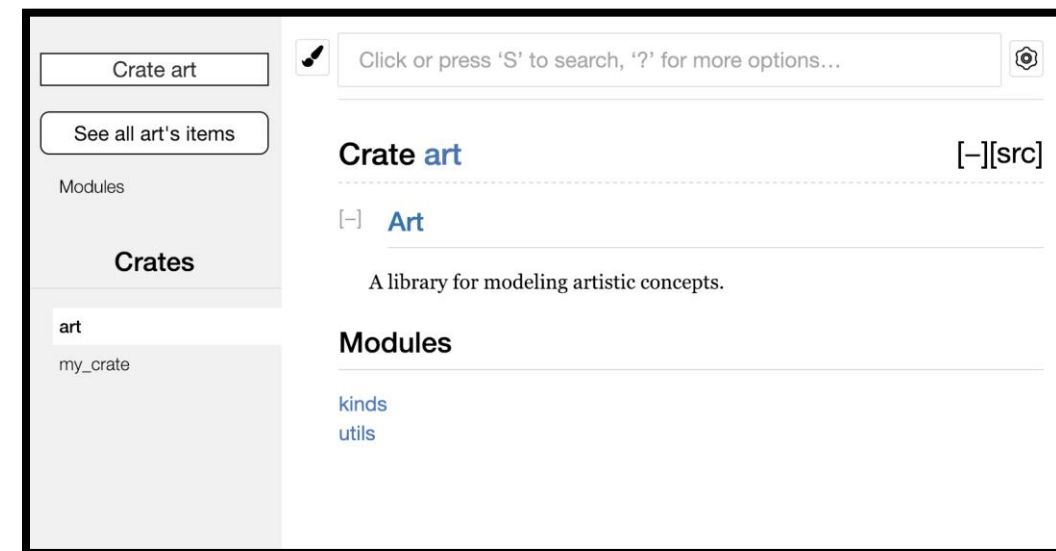
```
#![ # Art
//:
//: A library for modeling artistic concepts.

pub mod kinds {
    /// The primary colors according to the RYB color model.
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// The secondary colors according to the RYB color model.
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use crate::kinds::*;

    /// Combines two primary colors in equal amounts to create
    /// a secondary color.
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --省略--
    }
}
```



- 注意到 PrimaryColor 與 SecondaryColor 型別沒有列在首頁，而函式 mix 也沒有。必須點擊 kinds 與 utils 才能看到它們。

Cargo 與 Crates.io

透過 `pub use` 匯出理想的公開 API

```
#!/ # Art
//:
//: A library for modeling artistic concepts.

pub mod kinds {
    /// The primary colors according to the RYB color model.
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// The secondary colors according to the RYB color model.
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use crate::kinds::*;

    /// Combines two primary colors in equal amounts to create
    /// a secondary color.
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --省略--
    }
}
```

- 其他依賴此函式庫的 `crate` 需要使用 `use` 陳述式來將 `art` 的項目引入作用域中，並指定當前模組定義的架構。下圖範例顯示了從 `art crate` 使用 `PrimaryColor` 和 `mix` 項目的 `crate` 範例：`(src/main.rs)`

```
use art::kinds::PrimaryColor;
use art::utils::mix;

fn main() {
    let red = PrimaryColor::Red;
    let yellow = PrimaryColor::Yellow;
    mix(red, yellow);
}
```

- 範例中使用 `art crate` 的程式碼作者必須搞清楚 `PrimaryColor` 位於 `kinds` 模組中而 `mix` 位於 `utils` 模組中。
`art crate` 的模組架構對開發 `art crate` 的開發者才比較有意義，對使用者來說就沒那麼重要。
- 內部架構沒有提供什麼有用的資訊給想要知道如何使用 `art crate` 的人，還容易造成混淆，因為開發者得自己搞清楚要從何處找起，而且必須在 `use` 陳述式中指定每個模組名稱。

Cargo 與 Crates.io

透過 `pub use` 匯出理想的公開 API

- 要從公開 API 移除內部架構，可以修改範例中 `art crate` 的程式碼，並加上 `pub use` 陳述式來在頂層重新匯出項目，如底下範例所示：(src/lib.rs)

```
#![ # Art
#![
#![ A library for modeling artistic concepts.

pub use self::kinds::PrimaryColor;
pub use self::kinds::SecondaryColor;
pub use self::utils::mix;

pub mod kinds {
    // --省略--
}

pub mod utils {
    // --省略--
}
```

```
#![ # Art
#![
#![ A library for modeling artistic concepts.

pub mod kinds {
    /// The primary colors according to the RGB color model.
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// The secondary colors according to the RGB color model.
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use crate::kinds::*;

    /// Combines two primary colors in equal amounts to create
    /// a secondary color.
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --省略--
    }
}
```

Cargo 與 Crates.io

透過 pub use 匯出理想的公開 API

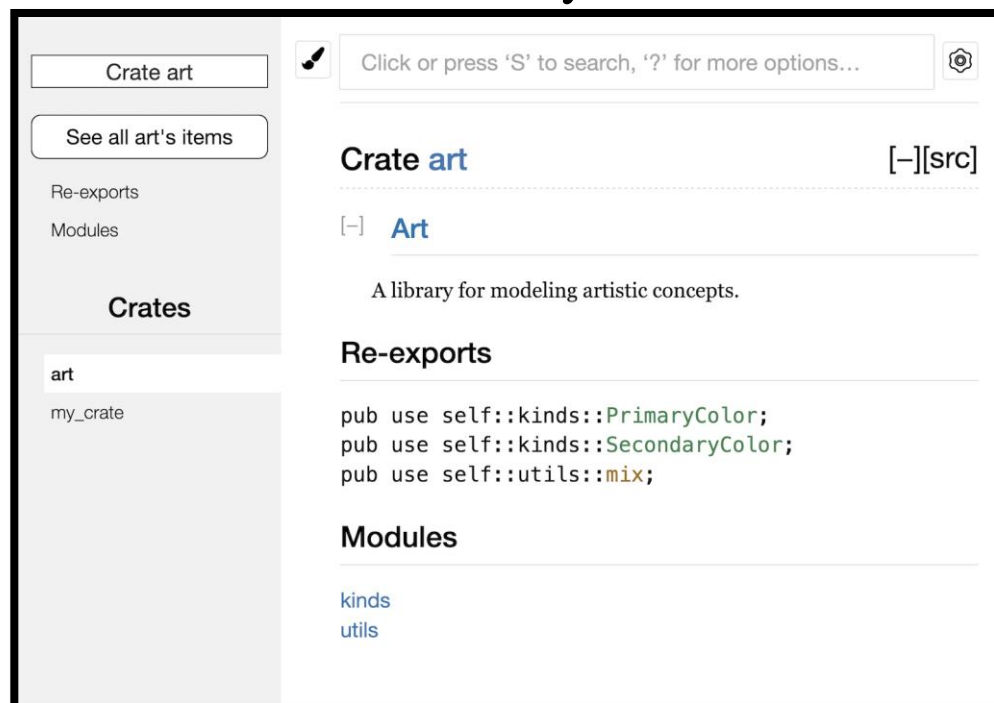
- cargo doc 對此 crate 產生的 API 技術文件現在就會顯示與連結重新匯出的項目到首頁中，如下圖所示。讓 PrimaryColor 與 SecondaryColor 型別以及函式 mix 更容易被找到：

```
//! # Art
//!
//! A library for modeling artistic concepts.

pub use self::kinds::PrimaryColor;
pub use self::kinds::SecondaryColor;
pub use self::utils::mix;

pub mod kinds {
    // --省略--
}

pub mod utils {
    // --省略--
}
```



Cargo 與 Crates.io

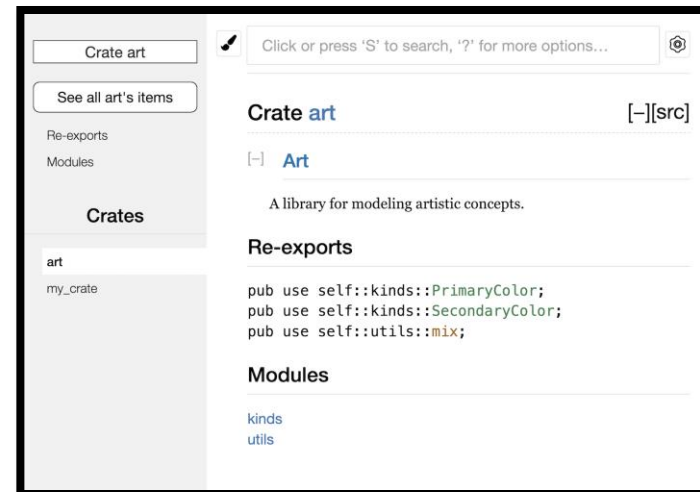
透過 `pub use` 匯出理想的公開 API

- `art` crate 使用者仍可以看到並使用之前範例的內部架構，如之前範例所展示的方式或者它們可以使用像範例這樣更方便的架構，如底下範例所示：`(src/main.rs)`

```
use art::mix;
use art::PrimaryColor;

fn main() {
    // --省略--
}
```

如果有許多巢狀模組(`nested modules`)的話，在頂層透過 `pub use` 重新匯出型別可以大大提升使用 `crate` 的體驗。另一項 `pub use` 的常見用途是重新匯出目前 `crate` 依賴的定義，讓那些 `crate` 定義成 `crate` 公開 API 的一部分。提供實用的公開 API 架構更像是一門藝術而不只是科學，而可以一步步來尋找最適合使用者的 API 架構。使用 `pub use` 可以給更多組織 `crate` 內部架構的彈性，並將內部架構與要呈現給使用者的介面互相解偶(`decouple`)。可以觀察一些安裝過的程式碼，看看它們的內部架構是不是不同於它們的公開 API。



Cargo 與 Crates.io

設定 Crates.io 帳號

- 在可以發佈任何 crate 之前，需要建立一個 crates.io 的帳號並取得一個 API token。請前往 crates.io 的首頁並透過 GitHub 帳號來登入(GitHub 目前是必要的，但未來可能會支援其他建立帳號的方式)，一旦登入好了之後，到帳號設定 <https://crates.io/me/> 並索取 API key，然後用這個 API key 來執行 cargo login 命令，如以下所示：

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

- 此命令會傳遞 API token 給 Cargo 並儲存在本地的 ~/.cargo/credentials。注意此 token 是個祕密(secret)，千萬不要分享給其他人。如果因為任何原因分享給任何人的話，最好撤銷掉並回到 crates.io 產生新的 token。

Cargo 與 Crates.io

新增詮釋資料到新的 Crate

- 假設有個 crate 想要發佈。在發佈之前，需要加上一些詮釋資料(metadata)，也就是在 crate 的 Cargo.toml 檔案中 [package] 的段落內加上更多資料。
- crate 必須要有個獨特的名稱。雖然在本地端開發 crate 時，crate 可以是任何想要的名稱。但是 crates.io 上的 crate 名稱採先搶先贏制。一旦有 crate 名稱被取走了，其他人就不能再使用該名稱來發佈 crate。在嘗試發佈 crate 前，最好先搜尋想使用的名稱。如果該名稱已被使用了，就需要想另一個名稱，並在 Cargo.toml 檔案中 [package] 段落的 name 欄位使用新的名稱來發佈，如以下所示：

```
[package]  
name = "guessing_game"
```


Cargo 與 Crates.io

新增詮釋資料到新的 Crate

- 當選好獨特名稱後，此時執行 `cargo publish` 來發佈 crate 的話，會得到以下警告與錯誤：

```
$ cargo publish
  Updating crates.io index
warning: manifest has no description, license, license-file, documentation, homepage or
repository.
See https://doc.rust-lang.org/cargo/reference/manifest.html#package-metadata for more info.
--省略--
error: failed to publish to registry at https://crates.io

Caused by:
  the remote server responded with an error: missing or empty metadata fields: description,
  license. Please see https://doc.rust-lang.org/cargo/reference/manifest.html for how to
  upload metadata
```

- 這是因為還缺少一些關鍵資訊：描述與授權條款是必須的，所以人們才能知道 crate 在做什麼以及在何種情況下允許使用。在 `Cargo.toml` 檔案中加上一兩句描述，它就會顯示在 crate 的搜尋結果中。

Cargo 與 Crates.io

新增詮釋資料到新的 Crate

- 至於 license 欄位，需要給予 license identifier value。Linux Foundation's Software Package Data Exchange(SPDX) 有列出可以使用的標識符數值。舉例來說，要指定 crate 使用 MIT 授權條款的話，就加上 MIT 標識符：

```
[package]
name = "guessing_game"
license = "MIT"
```

- 如果想使用沒有出現在 SPDX 的授權條款，需要將該授權條款的文字儲存在一個檔案中，將該檔案加入專案中並使用 license-file 來指定該檔案名稱，而不使用 license。
- 社群中許多人都會用 MIT OR Apache-2.0 **雙授權**條款作為它們專案的授權方式，這和 Rust 的授權條款一樣。這也剛好展示也可以用 OR 指定數個授權條款，讓專案擁有數個不同的授權方式。

Cargo 與 Crates.io

新增詮釋資料到新的 Crate

- 有了獨特名稱、版本、描述與授權條款，已經準備好發佈的 Cargo.toml 檔案會如以下所示：

```
[package]
name = "guessing_game"
version = "0.1.0"
edition = "2021"
description = "A fun game where you guess what number the computer has chosen."
license = "MIT OR Apache-2.0"

[dependencies]
```

- Cargo 技術文件還介紹了其他可以指定的詮釋資料，讓 crate 更容易被其他人發掘並使用。

Cargo 與 Crates.io

發佈至 Crates.io

```
$ cargo publish
  Updating crates.io index
  Packaging guessing_game v0.1.0 (file:///projects/guessing_game)
  Verifying guessing_game v0.1.0 (file:///projects/guessing_game)
  Compiling guessing_game v0.1.0
(file:///projects/guessing_game/target/package/guessing_game-0.1.0)
  Finished dev [unoptimized + debuginfo] target(s) in 0.19s
  Uploading guessing_game v0.1.0 (file:///projects/guessing_game)
```

- 現在已經建立了帳號、儲存了 API token、選擇了 crate 的獨特名稱並指定了所需的詮釋資料。現在已經準備好發佈了！發佈 crate 會上傳一個指定版本到 crates.io 供其他人使用。
- 發佈 crate 時請格外小心，因為發佈是會永遠存在的。該版本無法被覆寫，而且程式碼無法被刪除。crates.io 其中一個主要目標就是要作為儲存程式碼的永久伺服器，讓所有依賴 crates.io 的 crate 的專案可以持續正常運作。允許刪除版本會讓此目標幾乎無法達成。
- 不過能發佈的 crate 版本不會有數量限制。再執行 cargo publish 命令，這次就應該會成功了。
- 恭喜！現在將程式碼分享給 Rust 社群了，任何人現在都可以輕鬆將 crate 加到專案中作為依賴了。

Cargo 與 Crates.io

發佈至 Crates.io

- 對現有 Crate 發佈新版本
 - 當對 crate 做了一些改變並準備好發佈新版本時，可以變更 Cargo.toml 中的 version 數值，並再發佈一次。請使用語意化版本規則依據作出的改變來決定下一個妥當的版本數字。接著執行 cargo publish 來上傳新版。

Cargo 與 Crates.io

發佈至 Crates.io

- 透過 `cargo yank` 棄用 Crates.io 的版本
 - 雖然無法刪除 crate 之前的版本，還是可以防止任何未來的專案加入它們作為依賴。這在 crate 版本因某些原因而被破壞時會很有用。在這樣的情況下，Cargo 支援撤回(yanking)crate 版本。
 - 撤回一個版本能防止新專案用該版本作為依賴，同時允許現存依賴它的專案能夠繼續依賴該版本。實際上，撤回代表所有專案的 `Cargo.lock` 都不會被破壞，且任何未來產生的 `Cargo.lock` 檔案不會使用被撤回的版本。
 - 要撤回一個 crate 的版本，在先前發布的 crate 目錄底下執行 `cargo yank` 並指定想撤回的版本。舉例來說，如果發布了一個 `guessing_game` crate 的版本 1.0.1，然讓想撤回的話，可以在 `guessing_game` 專案目錄底下執行：

```
$ cargo yank --vers 1.0.1
Updating crates.io index
Yank guessing_game@1.0.1
```

Cargo 與 Crates.io

發佈至 Crates.io

- 透過 `cargo yank` 棄用 Crates.io 的版本
 - 而對命令加上 `--undo` 的話，還可以在復原撤回的動作，允許其他專案可以再次依賴該版本：

```
$ cargo yank --vers 1.0.1 --undo
Updating crates.io index
Yank guessing_game@1.0.1
```

- 撤回並不會刪除任何程式碼。舉例來說，它並不會刪除任何不小心上傳的祕密訊息。如果真的出現這種情形，必須立即重設那些資訊。

Cargo 與 Crates.io

Cargo 工作空間

- 在前面章節中，建立的套件包含一個執行檔 `crate` 與一個函式庫 `crate`。隨著專案開發，可能會發現函式庫 `crate` 變得越來越大，而可能會想要將套件拆成數個函式庫 `crate`。
- Cargo 提供了一個功能叫做工作空間(workspaces)能來幫助管理並開發數個相關的套件。
- 建立工作空間
 - 工作空間是一系列的共享相同 `Cargo.lock` 與輸出目錄的套件。建立個使用工作空間的專案，會使用簡單的程式碼，好能專注在工作空間的架構上。組織工作空間的架構有很多種方式，會介紹其中一種常見的方式。
 - 工作空間將會包含一個執行檔與兩個函式庫。執行檔會提供主要功能，並依賴其他兩個函式庫。其中一個函式庫會提供函式 `add_one`，而另一個函式庫會提供函式 `add_two`。這三個 `crate` 會包含在相同的工作空間中，先從建立工作空間的目錄開始：

```
$ mkdir add  
$ cd add
```


Cargo 與 Crates.io

Cargo 工作空間

- 建立工作空間
 - 接著在 `add` 目錄中，建立會設置整個工作空間的 `Cargo.toml` 檔案。此檔案不會有 `[package]` 段落。反之，會使用一個 `[workspace]` 段落作為起始，可以透過指定執行檔 `crate` 的套件路徑來將它加到工作空間的成員中。在此例中，路徑是 `adder`：

```
[workspace]
members = [
    "adder",
]
```

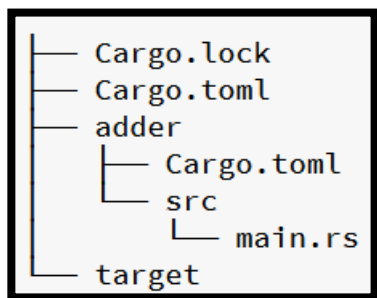
- 接下來會在 `add` 目錄下執行 `cargo new` 來建立 `adder` 執行檔 `crate`：

```
$ cargo new adder
Created binary (application) `adder` package
```

Cargo 與 Crates.io

Cargo 工作空間

- 建立工作空間
 - 在這個階段，已經可以執行 `cargo build` 來建構工作空間。目錄 `add` 底下的檔案應該會看起來像這樣：



- 工作空間在頂層有一個 `target` 目錄用來儲存編譯結果。 `adder` 套件不會有自己的 `target` 目錄。就算在 `adder` 目錄底下執行 `cargo build`，編譯結果仍然會在 `add/target` 底下而非 `add/adder/target`。Cargo 之所以這樣組織工作空間的 `target` 目錄是因為工作空間的 `crate` 是會彼此互相依賴的。如果每個 `crate` 都有自己的 `target` 目錄，每個 `crate` 就得重新編譯工作空間中的其他每個 `crate` 才能將編譯結果放入它們自己的 `target` 目錄。共享 `target` 目錄的話，`crate` 可以避免不必要的重新建構。

Cargo 與 Crates.io

Cargo 工作空間

- 在工作空間中建立第二個套件
 - 接下來在工作空間中建立另一個套件成員 `add_one`。請修改頂層 `Cargo.toml` 來指定 `add_one` 的路徑到 `members` 列表中：

```
[workspace]

members = [
    "adder",
    "add_one",
]
```

- 然後產生新的函式庫 `crate add_one`：
- `add` 目錄現在應該要擁有這些目錄與檔案：

```
$ cargo new add_one --lib
Created library `add_one` package
```

```
├── Cargo.lock
├── Cargo.toml
├── add_one
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

Cargo 與 Crates.io

Cargo 工作空間

- 在工作空間中建立第二個套件
 - 在 `add_one/src/lib.rs` 檔案中，加上一個函式 `add_one`：`(add_one/src/lib.rs)`

```
pub fn add_one(x: i32) -> i32 {  
    x + 1  
}
```

- 現在可以讓 `adder` 套件的執行檔依賴擁有函式庫的 `add_one` 套件。首先，需要將 `add_one` 的路徑依賴加到 `adder/Cargo.toml`。

```
[dependencies]  
add_one = { path = "../add_one" }
```

Cargo 與 Crates.io

Cargo 工作空間

- 在工作空間中建立第二個套件
 - Cargo 不會假設工作空間下的 crate 會彼此依賴，要指定彼此之間依賴的關係。
 - 接著在 adder 內使用 add_one crate 的 add_one 函式。開啟 adder/src/main.rs 檔案並在最上方加上 use 來將 add_one 函式庫引入作用域。然後變更 main 函式來呼叫 add_one 函式，如底下範例所示：

```
use add_one;

fn main() {
    let num = 10;
    println!(
        "你好，世界！{num} 加一會是 {}！", add_one::add_one(num);
    );
}
```

Cargo 與 Crates.io

Cargo 工作空間

- 在工作空間中建立第二個套件
 - 在頂層的 add 目錄執行 cargo build 來建構工作空間吧！

```
$ cargo build
Compiling add_one v0.1.0 (file:///projects/add/add_one)
Compiling adder v0.1.0 (file:///projects/add/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.68s
```

- 要執行 add 目錄的執行檔 crate，可以透過 -p 加上套件名稱使用 cargo run 來執行想要在工作空間中指定的套件：

```
$ cargo run -p adder
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/adder`
你好，世界！10 加一會是 11！
```

- 這就會執行 adder/src/main.rs 的程式碼，其依賴於 add_one crate。

Cargo 與 Crates.io

Cargo 工作空間

- 在工作空間中依賴外部套件
 - 注意到工作空間只有在頂層有一個 `Cargo.lock` 檔案，而不是在每個 `crate` 目錄都有一個 `Cargo.lock`。這確保所有的 `crate` 都對所有的依賴使用相同的版本。如果加了 `rand` 套件到 `adder/Cargo.toml` 與 `add_one/Cargo.toml` 檔案中，Cargo 會將兩者的版本解析為同一個 `rand` 版本並記錄到同個 `Cargo.lock` 中。
 - 確保工作空間所有 `crate` 都會使用相同依賴代表工作空間中的 `crate` 永遠都彼此相容。將 `rand` `crate` 加到 `add_one/Cargo.toml` 檔案的 `[dependencies]` 段落中，使 `add_one` `crate` 可以使用 `rand` `crate`：`rand = "0.8.5"`

Cargo 與 Crates.io

Cargo 工作空間

- 在工作空間中依賴外部套件
 - 現在就可以將 `use rand;` 加到 `add_one/src/lib.rs` 檔案中，接著在 `add` 目錄下執行 `cargo build` 來建構整個工作空間就會引入並編譯 `rand` crate。會得到一個警告，因為還沒有開始使用引入作用域的 `rand`：

```
$ cargo build
  Updating crates.io index
  Downloaded rand v0.8.5
  --省略--
  Compiling rand v0.8.5
  Compiling add_one v0.1.0 (file:///projects/add/add_one)
warning: unused import: `rand`
--> add_one/src/lib.rs:1:5
1 | use rand;
  |     ^^^^
   = note: `[warn(unused_imports)]` on by default
warning: `add_one` (lib) generated 1 warning
  Compiling adder v0.1.0 (file:///projects/add/adder)
  Finished dev [unoptimized + debuginfo] target(s) in 10.18s
```

頂層的 `Cargo.lock` 現在就包含 `add_one` 有 `rand` 作為依賴的資訊。不過就算能在工作空間的某處使用 `rand`，並不代表可以在工作空間的其他 `crate` 中使用它，除非它們的 `Cargo.toml` 也加上了 `rand`。

Cargo 與 Crates.io

Cargo 工作空間

- 在工作空間中依賴外部套件
 - 舉例來說，如果將 `use rand;` 加到 `adder/src/main.rs` 檔案中想讓 `adder` 套件也使用的話，就會得到錯誤：

```
$ cargo build
--省略--
Compiling adder v0.1.0 (file:///projects/add/adder)
error[E0432]: unresolved import `rand`
--> adder/src/main.rs:2:5
  |
2 | use rand;
  |     ^^^^ no external crate `rand`
```

- 要修正此問題，只要修改 `adder` 套件的 `Cargo.toml` 檔案，指示它也加入 `rand` 作為依賴就好了。這樣建構 `adder` 套件就會將在 `Cargo.lock` 中將 `rand` 加入 `adder` 的依賴，但是沒有額外的 `rand` 會被下載。
- Cargo 會確保工作空間中每個套件的每個 `crate` 都會使用相同的 `rand` 套件版本。這可以節省空間，並能確保工作空間中的 `crate` 彼此可以互相兼容。

Cargo 與 Crates.io

Cargo 工作空間

- 在工作空間中新增測試
 - 再進一步加入一個測試函式 `add_one::add_one` 到 `add_one` crate 之中：`(add_one/src/lib.rs)`

```
pub fn add_one(x: i32) -> i32 {  
    x + 1  
}  
  
#[cfg(test)]  
mod tests {  
    use super::*;  
  
    #[test]  
    fn it_works() {  
        assert_eq!(3, add_one(2));  
    }  
}
```

Cargo 與 Crates.io

Cargo 工作空間

- 在工作空間中新增測試
 - 現在在頂層的 `add` 目錄執行 `cargo test`。像這樣在工作空間的架構下執行 `cargo test` 會執行工作空間下所有 crate 的測試：

```
$ cargo test
  Compiling add_one v0.1.0 (file:///projects/add/add_one)
  Compiling adder v0.1.0 (file:///projects/add/adder)
  Finished test [unoptimized + debuginfo] target(s) in 0.27s
   Running unittests src/lib.rs (target/debug/deps/add_one-f0253159197f7841)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
   Running unittests src/main.rs (target/debug/deps/adder-49979ff40686fa8e)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

   Doc-tests add_one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Cargo 與 Crates.io

Cargo 工作空間

- 在工作空間中新增測試
 - 輸出的第一個段落顯示了 `add_one` crate 中的 `it_works` 測試通過。下一個段落顯示 `adder` crate 沒有任何測試。然後最後一個段落顯示 `add_one` 中沒有任何技術文件測試。
 - 也可以在頂層目錄使用 `-p` 並指定想測試的 crate 名稱來測試工作空間中特定的 crate：

```
$ cargo test
Compiling add_one v0.1.0 (file:///projects/add/add_one)
Compiling adder v0.1.0 (file:///projects/add/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.27s
Running unittests src/lib.rs (target/debug/deps/add_one-f0253159197f7841)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Running unittests src/main.rs (target/debug/deps/adder-49979ff40686fa8e)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Doc-tests add_one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

```
$ cargo test -p add_one
Finished test [unoptimized + debuginfo] target(s) in 0.00s
Running unittests src/lib.rs (target/debug/deps/add_one-b3235fea9a156f74)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Doc-tests add_one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Cargo 與 Crates.io

Cargo 工作空間

- 在工作空間中新增測試
 - 此輸出顯示 `cargo test` 只執行了 `add_one` crate 的測試並沒有執行 `adder` crate 的測試。
 - 如果想要發佈工作空間的 crate 到 `crates.io`，工作空間中的每個 crate 必須分別獨自發佈。和 `cargo test` 一樣可以用 `-p` 的選項來指定想要的 crate 名稱，來發布工作空間內的特定 crate。
 - 之後想嘗試練習的話，可以在工作空間中在加上 `add_two` crate，方式和 `add_one` crate 類似！
 - 隨著專案成長，可以考慮使用工作空間：拆成各個小部分比一整塊大程式還更容易閱讀。再者，如果需要經常同時修改的話，將 crate 放在同個工作空間中更易於彼此的協作。

```
$ cargo test -p add_one
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
    Running unittests src/lib.rs (target/debug/deps/add_one-b3235fea9a156f74)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

    Doc-tests add_one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Cargo 與 Crates.io

透過 cargo install 安裝執行檔

- `cargo install` 命令能本地安裝並使用執行檔 crates。這並不是打算要取代系統套件，這是為了方便讓 Rust 開發者可以安裝 crates.io 上分享的工具。注意只能安裝有執行檔目標的套件。
- 執行檔目標(binary target)是在 crate 有 `src/main.rs` 檔案或其他指定的執行檔時，所建立的可執行程式。而相反地，函式庫目標就無法單獨執行，因為它提供給其他程式使用的函式庫。通常 crate 都會提供 README 檔案說明此 crate 是函式庫還是執行檔目標，或者兩者都是。
- 所有透過 `cargo install` 安裝的執行檔都儲存在安裝根目錄的 `bin` 資料夾中。如果是用 `rustup.rs` 安裝 Rust 且沒有任何自訂設置的話，此目錄會是 `$HOME/.cargo/bin`。請確定該目錄有在 `$PATH` 中，這樣才能夠執行 `cargo install` 安裝的程式。

Cargo 與 Crates.io

透過 cargo install 安裝執行檔

- 舉例來說，之有個 Rust 版本的 grep 工具叫做 ripgrep 能用來搜尋檔案。要安裝 ripgrep 的話，可以執行以下命令：

```
$ cargo install ripgrep
  Updating crates.io index
  Downloaded ripgrep v13.0.0
  Downloaded 1 crate (243.3 KB) in 0.88s
  Installing ripgrep v13.0.0
  --省略--
  Compiling ripgrep v13.0.0
  Finished release [optimized + debuginfo] target(s) in 3m 10s
  Installing ~/.cargo/bin/rg
  Installed package `ripgrep v13.0.0` (executable `rg`)
```

- 輸出的最後兩行顯示了執行檔的安裝位置與名稱，在 ripgrep 此例中就是 rg。如之前提到的只要 \$PATH 有包含安裝目錄，就可以執行 rg --help 並開始使用更快速的搜尋檔案工具！

Cargo 與 Crates.io

透過自訂命令來擴展 Cargo 的功能

- Cargo 的設計能在不用修改 Cargo 的情況下擴展新的子命令。如果 `$PATH` 中有任何叫做 `cargo-something` 的執行檔，就可以用像是執行 Cargo 子命令的方式 `cargo something` 來執行它
- 像這樣的自訂命令在執行 `cargo --list` 時也會顯示出來。能夠透過 `cargo install` 來安裝擴展外掛並有如內建 Cargo 工具般來執行使用是 Cargo 設計上的一大方便優勢！