# 前言

我已經知道結局

這是讓你們臉垮下來的部分

——明日巨星合唱團〈Experimental Film〉（2004）

我記得 1995 年是新語言「JavaScript」的問世之時。過了幾年後，我決定學習這個語言，所以買了一本厚重的參考書，從頭到尾把它讀完。這本書寫得很好，非常詳細地說明該語言，從字串、串列到物件皆有論述。但是讀完那本書之後，我依然不知如何編寫 JavaScript 程式幫助我的生活。若沒有藉由編寫程式以套用書中知識，則學到的東西不多。從那之後，我對於學習程式語言的方式有所變革，這也許是身為程式設計師的你能夠發展的最有價值技能。對我來說，如此表示可重構已經知悉的程式，譬如：井字遊戲。

Rust 是當今這個領域的新成員，也許你已經拿起本書看看與其相關的一切。本書不是該語言的參考書。因為已有這方面的書籍，而且寫得不錯。我轉而寫了本書，要求你挑戰編寫可能已知悉的多個小程式。Rust 被普遍認為其學習曲線相當陡峭，但相信本書這種做法能協助你快速提高這個語言的運用生產力。

具體來說，你要編寫 Rust 版的 Unix 核心指令列工具，如 `head`、`cal`。這將為你帶來這些工具的詳加論述，以及它們相當實用的原因，同時還提供 Rust 概念（如字串、向量、filehandle）應用情況。若你不熟悉 Unix 或指令列程式設計，則將學到某些概念，如程式結束碼、指令列引數，輸出重導向，用管線將某程式的輸出（`STDOUT` 或標準輸出）連接到另一個程式的輸入（`STDIN` 或標準輸入），以及使用 `STDERR`（標準錯誤）將錯誤訊息與其他

輸出分開顯示。你編寫這些程式所顯現的模式（如驗證參數、讀寫檔案、剖析文字、使用正規表達式），可在建立自己的 Rust 程式時使用。

這些工具與概念有不少的部分是 Windows 沒有支援的，因此該平台的使用者對於數個 Unix 核心程式，僅會建立出功能受限的版本。

## 何謂 Rust（為何人人皆在談論 Rust）？

Rust（*https://www.rust-lang.org*）是「讓人人都能建置高效可靠軟體的程式語言」。Rust 是 Graydon Hoare 約莫於 2006 年創造的（還有其他人參與此專案計畫），當時 Hoare 任職於 Mozilla Research。該語言在 2010 年因累積足夠的關注與使用者，而讓 Mozilla 贊助相關的開發工作。 2021 年的 Stack Overflow 開發者大調查（*https://oreil.ly/3rumR*）中，有近 80,000 名開發人員將 Rust 列為「最愛的」程式語言（也已蟬聯六年的最愛語言）。

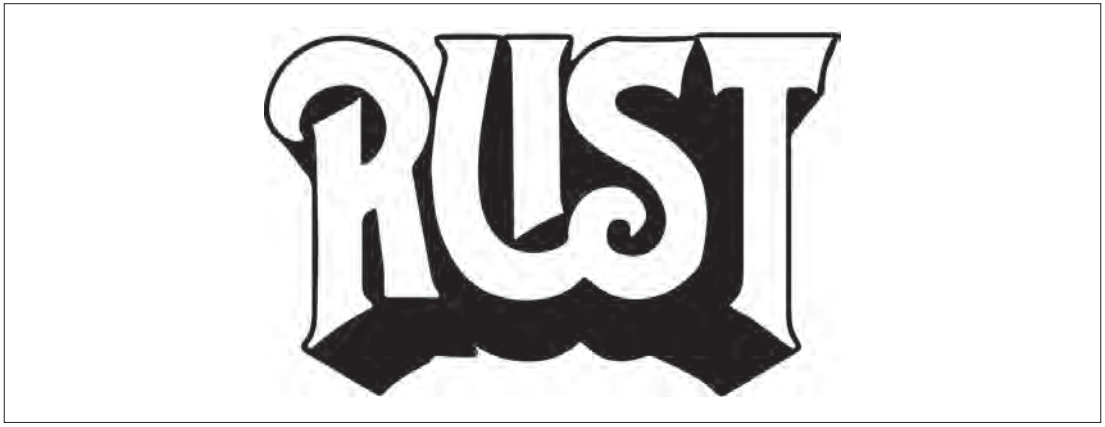


**圖 P-1　這是我參考舊的 Rush 樂團標誌製作的 Rust 標誌。1980 年代，當我還是個打鼓小子時，接觸到 Rush 不少作品。無論如何，Rust 很酷，我以此標誌表示之。**

該語言的語法與 C 語言類似，因此你會發現諸如 `for` 迴圈、以分號結尾的陳述句、表示區塊結構的大括號之類的內容。重點是，Rust 能以借用檢查器（*borrow checker*）確保記憶體的安全，該檢查器可追蹤程式的某部分是否安全存取記憶體的個別位置。然而，這種安全性並不會以犧牲效能為代價。Rust 程式會被編譯為本機的二進位可執行檔，而且編譯的速度往往等同或超越 C 或 C++ 的效能。基於這個原因，Rust 通常被視為一種專為效能與安全而設計的系統程式語言。

Rust 是一種如 C/C++、Java 這類的**靜態型別**（*statically typed*）語言。如此表示變數始終不能更改其型別，譬如把數值改成字串。因為編譯器往往可以從程式碼的上下文中找出變數所屬的型別，所以你不必在 Rust 中刻意宣告變數的型別。上述的特性與像 Perl、Perl、JavaScript、Python 這類的**動態型別**（*dynamically typed*）語言形成強烈對比，動態型別語言的變數可以在程式的任何位置更改其型別，例如把字串改成 filhandle。

Rust **並非**物件導向（OO）語言（如 C++、Java 這樣的物件導向語言），即 Rust 沒有類別或繼承。Rust 主要採用 `struct`（結構）表示複雜的資料型別以及使用 *trait* 描述型別的行為模式。這些結構可能含有方法（函式），可以改變資料的內部狀態，甚至會在說明文件中被稱為**物件**（*object*），不過這些並非正式文義所指的物件。

Rust 從其他程式語言與程式設計法中（包括像 Haskell 這樣的純函式語言）借用許多振奮人心的概念。例如，Rust 的變數預設是**不可變的**（*immutable*），即不能變更其初始值；你必須特別告知編譯器讓變數是可變的。函式也是**一級**（*first-class*）值，即可以將其作為引數傳給其他**較高級函式**。最振奮我心的是 Rust 具備**列舉**（*enum*）型別與 *sum* 型別，又被稱為**代數資料型別**（ADT），例如用於表示函式回傳一個 `Result`，該 `Result` 可以是內含某值的 `Ok` 或含其他種值的 `Err`。處理這些值的程式碼都必須處理所有的可能情況，因此你永遠不會有忘記處理錯誤的風險（這些錯誤可能讓程式意外的停擺）。

## 適合閱讀本書的讀者

若你想藉由編寫實用的指令列程式（處理常見的程式設計作業）學習 Rust 語言的基礎知識，則應該閱讀本書。我想大多數讀者至少已有另一個程式語言的一些編程基礎知識。例如，你可能知道如何建立變數、使用迴圈重複進行一個作業、建立函式等等。我覺得 Rust 可能是不易入門的程式語言，理由是 Rust 廣泛使用型別並且需要了解電腦記憶體的某些細節。我還認為你至少要對如何使用指令列有所了解，並知道一些基本的 Unix 指令，譬如建立、刪除、更改目錄。本書將聚焦於實用面的內容，說明完成工作所需了解的知識。至於 Rust 的細節則讓更全面的書籍來表述，例如由 Jim Blandy、Jason Orendorff、Leonora F. S. Tindall 所著的《Programming Rust》第二版（O'Reilly）和 Steve Klabnik、Carol Nichols 所寫的《The Rust Programming Language》（No Starch Press）。強烈建議你搭配本書一起閱讀其中一本或兩本，進而更深入了解 Rust 語言本身。

若你想知道如何編寫與執行測試，用於檢查 Rust 程式，則也應該閱讀本書。我主張測試不僅要用於驗證程式是否能正常運作，還要協助將一個問題分成小而可理解可測試的部分。我將說明如何使用我提供的測試，以及如何使用**測試驅動開發**（TDD），就這種開發方式而言，首先編寫測試，然後編寫能通過這些測試的程式碼。我希望本書呈現的是，Rust 編譯器的嚴格特性與測試互相結合，可以造就更好的程式（更易於維護和修改的程式）。

## 應該學習 Rust 的理由

學習 Rust 的理由不少。第一，我發現 Rust 的型別檢查避免我犯了許多基本錯誤。我的程式語言經驗主要是動態型別語言，如 Perl、Python、JavaScript，這些語言幾乎沒有型別檢查。使用像 Rust 這樣的靜態型別語言越頻繁，我就越意識到動態型別語言強行為我做不少的工作，而如此也要求我驗證程式以及編寫更多的測試。漸漸覺得 Rust 編譯器雖然很嚴格，但卻是我的舞伴（而非我的反對者）。當然，身為一個舞伴，每次你與其碰撞或錯過提示時，它都會告訴你，而最終會讓你成為一個更好的舞者，這畢竟是我們追求的目標。一般來說，當我編譯一個 Rust 程式時，它通常會按照我的意圖運作。

第二，與不懂 Rust 或根本不是開發者的人共用 Rust 程式很容易。若我為同事編寫 Python 程式，必須為他們提供 Python 原始碼方能執行，並要確保他們有正確版本的 Python 和執行我的程式碼所需的所有模組。相較之下，Rust 程式直接編譯成機器可執行檔。我可以在我的機器上編寫與測試一個程式，針對要執行的架構建立一個對應的可執行檔，並向我的同事提供該程式的副本。假設他們有一樣的執行架構，他們不需要安裝 Rust，即可直接執行該程式。

第三，我經常使用 Docker 或 Singularity 建置容器封裝工作流程。我發現 Rust 程式的容器往往比 Python 程式的容器小幾個數量級。例如，內有 Python 執行環境的 Docker 容器可能需要幾百 MB。相較之下，我可以建置一個陽春的 Linux 虛擬機，內有一個 Rust 二進位檔，可能只有數十 MB 大小。除非我真的需要 Python 的一些特定功能，譬如機器學習或自然語言處理模組，否則我更喜歡用 Rust 編寫程式，造就更小更精簡的容器。

最後，我發現使用 Rust 相當有生產力，原因是有豐富可用的模組生態系統。我在 *crates.io* 上發現許多有用的 Rust crate（此為 Rust 函式庫的稱呼），以及 *Docs.rs* 的說明內容非常全面而易於瀏覽。

## 編程挑戰

本書將藉由建立完整程式說明如何編寫與測試 Rust 程式碼。每一章都將示範如何從頭開始撰寫一個程式，增加功能，處理錯誤訊息以及測試程式的邏輯。我不希望你在通勤的公車上被動地閱讀本書，再隨手把它闔上。藉由編寫自己的解決方案，你會得到最多的收穫，但我相信，即使只是輸入我提供的原始碼，對於學習結果也會有所助益。

我為本書選擇的問題源自 Unix 指令列工具 `coreutils`（*https://oreil.ly/fYV82*），原因是我認為這些問題對許多讀者來說應該相當熟捻。例如，我假設你已用過 `head`、`tail` 查看檔案的前幾行或後幾行，但是你是否曾經自行編寫過這些程式呢？其他的 Rustacean（Rustacean 泛指 Rust 使用者）也有同樣的想法（*https://www.rustaceans.org*），所以你可以在網際網路上找到這些程式的諸多 Rust 實作版本（*https://oreil.ly/RmiBN*）。此外，這些都是相當小的程式，每個程式皆可傳達某些特定技能。我已經對這些專案排序，讓它們得以循序建置，因此最好能夠依序閱讀各個章節。

我選擇這些程式的一個原因是它們提供了某種基準真相（ground truth）。雖然 Unix 有許多版本以及這些程式有多種實作，不過它們的運作通常都雷同，也會產生相同的結果。我使用 macOS 進行開發，如此表示我主要執行這些程式的 BSD（Berkeley Standard Distribution）或 GNU（GNU's Not Unix）版本（*https://www.gnu.org*），兩者為 Unix 的變體。一般來說，BSD 版早於 GNU 版，而且選項較少。對於每個挑戰程式，我使用 shell script 將原版程式的輸出重導向輸出檔中。目標是讓 Rust 程式可為相同的輸入建立相同的輸出。我有特別引入 Windows 編碼的檔案以及與 Unicode 字元混合的簡單 ASCII 文字，刻意讓我的程式如同原版程式的做法，處理行尾和字元的各種概念。

對於大多數的挑戰程式，我僅試圖實作原版程式的功能子集，不然挑戰程式可能會變得非常複雜。為了方便說明，我也決定對某些程式的輸出做些微變更。將此比擬成播放錄音一同演練的樂器演奏學習。你不必跟著奏出原版的每個音符。重點是要學習常用的模式，如處理引數與讀取輸入，這樣你就可以繼續編寫你的素材。就附加的挑戰來說，嘗試用其他語言編寫這些程式，如此你可以看到這些解決方案與 Rust 版有何不同。

第一章

# 真心話大挑戰

事實上

我們什麼都不知道

——明日巨星合唱團〈Ana Ng〉（1988）

本章將說明如何組成、執行、測試 Rust 程式。我會使用 Unix 平台（macOS）解釋有關指令列程式的一些基本概念。其中只有某些概念能套用到 Windows 作業系統，但無論你用哪個平台皆能正常執行這些 Rust 程式。

你將學習如何：

- 把 Rust 程式碼編譯成可執行檔
- 利用 Cargo 建新專案
- 使用 `$PATH` 環境變數
- 引入源自 *crates.io* 的外部 Rust crate
- 解讀程式執行結束狀態
- 執行常見的系統指令與選項
- 編寫 Rust 版的 `true`、`false` 程式
- 組成、編寫、執行測試碼（測試程式碼）

## 以「Hello, world!」入門

在螢幕上顯示「Hello, world!」，似乎是初學程式語言時普遍被認同的做法。執行 **`cd /tmp`** 進入暫存目錄，編寫第一個程式。至此僅為牛刀小試，所以我們尚不需要實際存放的目錄。

開啟文字編輯器，輸入下列程式碼，並儲存於 *hello.rs* 檔案中：

```
fn main() { ❶
    println!("Hello, world!"); ❷
} ❸
```

❶ 用 `fn` 定義函式（function）。該函式的名稱是 `main`。

❷ `println!`（*print line*）是**顯示行文字**的巨集（macro），能在 `STDOUT`（讀作 *standard out*，即標準輸出）顯示文字。分號表明該陳述句（statement）的結尾。

❸ 此函式的本體以大括號括起來。

Rust 會自動從 `main` 函式開始執行。函式引數（argument）列於函式名稱之後的括號內。因為 `main()` 沒有列出任何引數，所以此函式不帶任何引數。我在此要指出的最後一點是 `println!`（*https://oreil.ly/GGmNx*）看似函式，實際上卻是**巨集**（*https://oreil.ly/RFXMp*）——基本上是用於編寫程式碼的程式碼。本書用到的其他巨集——`assert!`（*https://oreil.ly/SQHyp*）、`vec!`（*https://oreil.ly/KACU4*）——結尾也是驚嘆號。

若要執行此一程式，首先你必須使用 Rust 編譯器（compiler）—— `rustc`，將該程式碼**編譯**（*compile*）成電腦能夠執行的格式：

```
$ rustc hello.rs
```

Windows 中改用下列指令：

```
> rustc.exe .\hello.rs
```

若一切順利的話，上述的指令不會顯示任何輸出內容，不過此時在 macOS、Linux 上應該有個 *hello* 新檔案（Windows 上則為 *hello.exe*）。這是二進位編碼檔案，可在作業系統上直接執行，因此通常稱之為**可執行檔**（*executable*）或**二進位檔**（*binary*）。以下是在 macOS 上用 `file` 指令查看此檔的檔案類型：

```
$ file hello
hello: Mach-O 64-bit executable x86_64
```

你應該可以成功執行該程式，而看到既歡心又真切的訊息：

```
$ ./hello ❶
Hello, world!
```

❶ 點（.）表示目前目錄。

本章稍後討論的 $PATH 環境變數（environment variable），內容是搜尋待執行程式可能所在的目錄。為了避免惡意程式碼偷偷被執行，千萬不要把目前工作目錄（current working directory 或現行工作目錄）列在此變數中。例如，某駭客建立名為 ls 的程式，內容卻是執行 rm -rf /，試圖刪除整個檔案系統。若你恰巧以 root 使用者身分執行該程式，那麼就會毀了你的一整天。

Windows 上則要以下列的方式執行：

```
> .\hello.exe
Hello, world!
```

如果這是你實作的第一個 Rust 程式，那麼就此恭喜你。接著我要說明如何讓該程式碼的組成更好。

## Rust 專案目錄的組成

你可能會在自己的 Rust 專案中撰寫許多原始碼檔，也會用到他人的程式碼（源自 *crates.io* 等處）。最好針對每個專案建置一個目錄，為所有 Rust 原始碼檔建一個 *src* 子目錄。在 Unix 系統上，首先你要用 rm hello 指令移除 *hello* 二進位檔（執行檔），原因是待會要以此名稱建立目錄。接著可以使用下列指令建置目錄結構：

```
$ mkdir -p hello/src ❶
```

❶ mkdir 指令用於建置目錄，-p 選項表示建立子目錄之前先建置父目錄，微軟平台的 PowerShell 則不需要此選項。

用 mv 指令將 *hello.rs* 原始碼檔移到 *hello/src* 目錄中：

```
$ mv hello.rs hello/src
```

使用 `cd` 指令更換到該目錄，再度編譯此一程式：

```
$ cd hello
$ rustc src/hello.rs
```

此時這個目錄應該會有個可執行檔 `hello`。以下是我使用 `tree` 指令呈現該目錄的內容（你可能需要手動安裝此指令才能執行）：

```
$ tree
.
├── hello
└── src
    └── hello.rs
```

這是簡單 Rust 專案的基本結構。

## 以 Cargo 建立與執行專案

建 Rust 新專案有更簡單的方式——利用 Cargo 工具。你可以刪除 *hello* 暫存目錄：

```
$ cd .. ❶
$ rm -rf hello ❷
```

❶ 換到父目錄，其中用兩點（`..`）表示父目錄。

❷ 遞迴（*recursive*）選項 `-r` 可移除目錄裡的所有內容，而強制（*force*）選項 `-f` 會忽略指令執行過程中的任何錯誤。

若你要保存隨後的程式，請換到專案的解決方案目錄。 接著使用 Cargo 重新建立專案：

```
$ cargo new hello

     Created binary (application) `hello` package
```

如此應該會有新建的 *hello* 目錄，這是你可以存取的目錄。以下我再度使用 `tree` 呈現該目錄的內容：

```
$ cd hello
$ tree
.
├── Cargo.toml ❶
└── src ❷
    └── main.rs ❸
```

❶ *Cargo.toml* 為專案設定檔（configuration file）。副檔名 *toml* 為 Tom's Obvious, Minimal Language（湯姆的淺顯極簡語言）的縮寫。

❷ *src* 目錄用來放置 Rust 原始碼檔。

❸ *main.rs* 為 Rust 程式的預設起始點。

你可以使用 `cat`（*concatenate*）指令查看 Cargo 所建的原始碼檔（第三章會編寫 Rust 版的 `cat` 指令程式）：

```
$ cat src/main.rs
fn main() {
    println!("Hello, world!");
}
```

這次使用 **`cargo run`** 編譯原始碼（而非以 `rustc` 編譯該程式）以及執行程式（用一個指令完成所有動作）：

```
$ cargo run
   Compiling hello v0.1.0 (/private/tmp/hello) ❶
    Finished dev [unoptimized + debuginfo] target(s) in 1.26s
     Running `target/debug/hello`
Hello, world! ❷
```

❶ 前三行是 Cargo 運作的相關資訊。

❷ 此為程式的輸出內容。

若你不想要 Cargo 顯示程式碼編譯與執行的狀態訊息，可以加上 `-q` 或 `--quiet` 選項：

```
$ cargo run --quiet
Hello, world!
```

### Cargo 指令

我怎麼會知道要用 `-q|--quiet` 選項呢？執行 **`cargo`** 時若不加任何引數，會列出一段不短的說明。友善的指令列工具會描述其用法，就像《愛麗絲夢遊仙境》的餅乾寫著「吃掉我」一樣。注意，*USAGE*（**用法**）是這種說明文件開頭會出現的字詞。通常將這樣的輔助訊息稱為**用法**說明。本書的程式也會顯示其用法。你可以用 **`cargo help`** ***`command`*** 取得任何 Cargo 指令的說明（在此 ***`command`*** 表示待查指令的名稱）。

以 Cargo 執行程式後，用 `ls` 指令列出目前工作目錄內容（第十四章將編寫 Rust 版的 `ls` 程式）。此時應該有個 *target* 新目錄。Cargo 會預設 *debug* 目標（*https://oreil.ly/1Fs8Q*），因而可以看到內含建置成品的 *target/debug* 目錄：

```
$ ls
Cargo.lock  Cargo.toml  src/        target/
```

你可以用前述的 `tree` 指令或 `find` 指令（第七章將編寫 Rust 版的 `find` 程式）查看 Cargo 與 Rust 建立的所有檔案。Cargo 執行的可執行檔應為 *target/debug/hello*。你可以直接執行它：

```
$ ./target/debug/hello
Hello, world!
```

總之，Cargo 把原始碼放在 *src/main.rs* 中，使用其中的 `main` 函式建置 *target/debug/hello* 二進位檔（執行檔），接著執行該檔。不過，為何二進位檔被稱為 *hello* 而不是 *main* 呢？我們可以檢視 *Cargo.toml* 之後，得到答案：

```
$ cat Cargo.toml
[package]
name = "hello" ❶
version = "0.1.0" ❷
edition = "2021" ❸

# 在 https://doc.rust-lang.org/cargo/reference/manifest.html
# 可得知這類檔案的諸多重點及其定義 ❹

[dependencies] ❺
```

❶ 此為 Cargo 所建的專案名稱，而它也會被作為可執行檔的名稱。

❷ 此為程式的版本。

❸ 此為編譯程式所用的 Rust 版號（*https://oreil.ly/4fgvX*）。版號的運作方式讓 Rust 社群能納入無法向下相容（backward compatible 或回溯相容）的變更，本書所有程式採用 2021 版號的 Rust 撰寫。

❹ 這一行是註解，本書只有此例有如此敘述。若你想要將此行註解從該檔案中移除也無妨。

❺ 此處放置專案會用到的外部 crate。該專案目前沒有用到其他 crate，所以這個段落（section）內容空白。

Rust 的函式庫（library）被稱為 *crate*，並以語意化版本號碼（*semantic version number*）——主版本（*major*）.次版本（*minor*）.修訂版（*patch*）形式表示，例如 `1.2.4` 表示主版本號碼是 1、次版本號碼是 2、修訂版號碼是 4。crate 的主版本更新代表該 crate 的公開應用程式介面（application programming interface 或 API）有重大的變更。

## 編寫與執行整合測試

> 相較於「測試」這件事，「設計測試」還是目前防止 bug 的最佳方式之一。建立有效測試而必須完成的思維能夠在寫程式之前發現 bug、排除 bug——實際上，測試設計的思維在軟體建置的每個階段（概念、規格、設計、編程等階段）都能對錯誤有所發覺與消除。
>
> —— Boris Beizer《*Software Testing Techniques*》（Van Nostrand Reinhold）

雖然「Hello, world!」很簡單，但仍須經過測試。本書將呈現兩大類的測試。為應用程式裡的函式撰寫測試碼屬於 *inside-out*（由內而外）的單元測試（*unit testing*），第四章將介紹單元測試。針對使用者執行應用程式的可能情況撰寫測試碼則是 *outside-in*（由外向內）的整合測試（*integration testing*），這是我們將為此範例程式執行的測試。按 Rust 專案的慣例，會於 *src* 目錄所在之處（即同一父目錄）建立 *tests* 目錄，用於存放對應的測試碼，你可以執行 **`mkdir tests`** 指令即可達成所求。

目的是在指令列上測試 `hello` 程式的執行（如同一般使用者所做的執行動作）。針對指令列介面（*command-line interface* 或 CLI）建立 *tests/cli.rs*，並將下列程式碼輸入該檔案中。注意，此函式的用意是呈現出最簡單合理的 Rust 測試碼，不過它尚無任何有用的功能：

```
#[test] ❶
fn works() {
    assert!(true); ❷
}
```

❶ `#[test]` 屬性（attribute）表示 Rust 測試時會執行此函式。

❷ `assert!` 巨集（*https://oreil.ly/SQHyp*）可判斷某布林表達式（Boolean expression）是否為 `true`。

目前的專案內容應該如下所示：

```
$ tree -L 2
.
├── Cargo.lock ❶
├── Cargo.toml
├── src ❷
│   └── main.rs
├── target ❸
│   ├── CACHEDIR.TAG
│   ├── debug
│   └── tmp
└── tests ❹
    └── cli.rs
```

❶ *Cargo.lock* 檔案（*https://oreil.ly/81q3a*）記錄應用程式建置所需的依賴套件（dependency），其中包含實際依賴的套件版本資訊。你應該不用變更該檔案的內容。

❷ *src* 目錄擺放應用程式建置所需的 Rust 原始碼檔。

❸ *target* 目錄存放建置成品。

❹ *tests* 目錄放置針對應用程式測試之用的程式碼（以 Rust 原始碼檔存放）。

本書的測試碼會使用 `assert!` 驗證某預期是否為 `true`，以及 `assert_eq!`（*https://oreil.ly/P6Bfw*）驗證某項是否為預期值。因為上述測試碼估算字面常數（literal value）`true`，所以始終都會通過測試。執行 **`cargo test`** 就能了解此測試碼的實際運作。你應該會在測試的輸出文字中看到下列的結果：

```
running 1 test
test works ... ok
```

將 *tests/cli.rs* 中的 `true` 改成 `false` 則會產生測試失敗的測試碼：

```
#[test]
fn works() {
    assert!(false);
}
```

在輸出內容中，你應該會看到下列測試失敗的結果：

```
running 1 test
test works ... FAILED
```

測試用的函式中可依你的需求加入許多 `assert!`、`asser_eq!` 的呼叫。而只要其中先有個呼叫發生問題，則整個測試即宣告失敗。

此刻我們要建立比較有用處的測試碼，執行指令以及檢查執行結果。Unix、Windows PowerShell 皆有 `ls` 指令可用，因此我們就以此指令著手應用。將 *tests/cli.rs* 的內容換成下列的程式碼：

```
use std::process::Command; ❶

#[test]
fn runs() {
    let mut cmd = Command::new("ls"); ❷
    let res = cmd.output(); ❸
    assert!(res.is_ok()); ❹
}
```

❶ 匯入 `std::process::Command`（*https://oreil.ly/ErqAX*）。`std` 表示此為標準（*standard*）函式庫的內容，是相當普及實用的 Rust 程式碼，而被包含在 Rust 語言中。

❷ 建立新 `Command`（執行 `ls`）。`let` 關鍵字（*https://oreil.ly/cYjVT*）將某變數以某值綁定（bind）`mut` 關鍵字（*https://oreil.ly/SH6Qr*）令變數為可變的（*mutable*），即其內容可以變更。

❸ 執行此指令並獲取輸出結果，其為 `Result`（*https://oreil.ly/EYxds*）。

❹ 驗證結果是否為 `Ok` 變體（variant）

Rust 變數預設為不可變的，即其值不能變更。

執行 **`cargo test`**，確認是否可在輸出內容中看到通過測試的結果：

```
running 1 test
test runs ... ok
```

用下列的程式碼替換 *tests/cli.rs* 的內容，即 `run` 函式執行的指令從 `ls` 改為 `hello`：

```
use std::process::Command;

#[test]
fn runs() {
    let mut cmd = Command::new("hello");
    let res = cmd.output();
    assert!(res.is_ok());
}
```

重新執行該測試碼，要注意的是測試沒有通過，失敗的原因是找不到 `hello` 指令：

```
running 1 test
test runs ... FAILED
```

回想一下，此二進位檔（執行檔）的存放路徑是 *target/debug/hello*。若你試圖在指令列上執行該指令，則會遇到找不到該指令的情況：

```
$ hello
-bash: hello: command not found
```

當你執行某指令時，作業系統會在預定的一組目錄中尋找對應名稱的檔案。[1] 在 Unix 類型的作業系統上，你可以檢視 shell 的 `PATH` 環境變數，得知該組目錄串列，目錄彼此以冒號分隔。（在 Windows 上，該環境變數則為 `$env:Path`。我可以使用**轉換字元**（*translate character*）指令 `tr` 將 `PATH` 的冒號（：）改成換行（newline）符號（`\n`）妥善呈現這些目錄：

1 shell 的指令別名（alias）與函式的執行方式如同指令，在此我僅論述待執行程式的搜尋。

```
$ echo $PATH | tr : '\n' ❶
/opt/homebrew/bin
/Users/kyclark/.cargo/bin
/Users/kyclark/.local/bin
/usr/local/bin
/usr/bin
/bin
/usr/sbin
/sbin
```

❶ 告知 `bash` 插入 `$PATH` 變數，使用 `|` 管線（pipe）將此變數內容送入 `tr`。

就算切換到 *target/debug* 目錄，還是找不到 *hello* 指令，原因是之前提到的安全限制——`PATH` 會將目前工作目錄排除在外：

```
$ cd target/debug/
$ hello
-bash: hello: command not found
```

必須直接表示要執行目前工作目錄的程式：

```
$ ./hello
Hello, world!
```

接著我需要用某種方式執行僅存於目前 crate 裡的二進位檔。

## 新增專案依賴套件

目前 `hello` 程式僅存於 *target/debug* 目錄中。若我將它複製到 `PATH` 所列的任何目錄中（注意，其中包含存放使用者私有程式的 *$HOME/.local/bin* 目錄），就可以執行它並成功通過測試。不過在此要直接測試儲存於目前 crate 的程式，而非複製過來測試。我可以使用 `assert_cmd`（*https://oreil.ly/hyuZZ*）找尋位於目前 crate 目錄中的程式。首先需要將 `assert_cmd` 視為開發依賴套件（*https://oreil.ly/pezix*）加入 *Cargo.toml* 中。向 Cargo 表明，針對一般測試與效能測試（benchmarking）僅需此 crate：

```
[package]
name = "hello"
version = "0.1.0"
edition = "2021"

[dependencies]
```

```
[dev-dependencies]
assert_cmd = "1"
```

然後，我可以利用此 crate 建立 `Command`（在 Cargo 存放二進位檔的目錄中可見的 `Command`）。下列測試並無驗證程式能否產生正確輸出，僅判斷程式的執行是否成功。用以下程式碼更新 *tests/cli.rs* 的內容，讓 `run` 函式改用 `assert_cmd::Command`（原為 `std::process::Command`）：

```
use assert_cmd::Command; ❶

#[test]
fn runs() {
    let mut cmd = Command::cargo_bin("hello").unwrap(); ❷
    cmd.assert().success(); ❸
}
```

❶ 匯入 `assert_cmd::Command`。

❷ 為執行目前 crate 的 `hello` 而建立 `Command`。其回傳 `Result`，由於應該能夠找到此二進位檔（執行檔），所以隨後的程式碼會呼叫 `Result::unwrap`（*https://oreil.ly/SV6w1*）。倘若不是如此（找不到這個二進位檔），則 `unwrap` 會導致 panic（錯誤），使得測試失敗，如此並不是件壞事。

❸ 使用 `Assert::success`（*https://oreil.ly/4VWet*）確保該指令能成功執行。

後續章節對於 `Result` 型別會有更多的論述。目前只要知道這是一種建模方式，為作業成敗建立的物件，其中有兩種可能的變體：`Ok`、`Err`。

執行 **`cargo test`**，以確認是否能看到下列的測試通過訊息：

```
running 1 test
test runs ... ok
```

## 理解程式結束碼

程式執行成功的意義為何？指令列程式應向作業系回報最後的結束狀態（exit status），表示成功或有問題。可移植作業系統介面（Portable Operating System Interface 或 POSIX）標準規定：標準結束碼（exit code）0 表示成功（即零錯誤之意），其他情況則以 1 到 255 之間的

任意數表示。我可以使用 bashshell 執行 `true` 指令呈現上述的情形。以下內容源自 macOS 作業系統上 **`man true`** 的使用手冊（manual page）：

```
TRUE(1)                    BSD 一般指令使用手冊                    TRUE(1)

名稱
     true -- 回傳 true 值。

概述
     true

描述
     該 true 工具程式始終以結束碼零回傳。

另參閱
     csh(1), sh(1), false(1)

標準
     該 true 工具程式符合 IEEE Std 1003.2-1992 (''POSIX.2'')。

BSD                              June 27, 1991                              BSD
```

如此說明文件所示，該程式的功能僅有回傳結束碼零。執行 **`true`**，並不會顯示任何輸出內容，但可以檢視 `bash` 的 **`$?`** 變數，得知最近一個指令執行的結束狀態：

```
$ true
$ echo $?
0
```

依此類推，`false` 指令執行結束時必然會回傳非零的結束碼：

```
$ false
$ echo $?
1
```

你在試寫本書的程式時，於正常結束處都應回傳零，在錯誤之處則回傳非零值。針對上述的 `true`、`false` 程式，你可以試著編寫自己的版本。首先使用 **`mkdir src/bin`** 建置 *src/bin* 目錄，接著使用下列內容編寫 *src/bin/true.rs* 檔案：

```
fn main() {
    std::process::exit(0); ❶
}
```

❶ 使用 `std::process::exit` 函式（*https://oreil.ly/hrM3X*），以零值結束程式。

此時 *src* 目錄的結構應該如下所示：

```
$ tree src/
src/
├── bin
│   └── true.rs
└── main.rs
```

執行此程式，然後自行確認程式執行結束時回傳的結束碼：

```
$ cargo run --quiet --bin true ❶
$ echo $?
0
```

❶ `--bin` 選項指明待執行的二進位目標檔（填寫檔名）。

將下列的測試函式加入 *tests/cli.rs* 檔案中，確保 `true` 程式可正確執行。這段程式碼擺在現有的 `runs` 函式之前或之後都無妨：

```
#[test]
fn true_ok() {
    let mut cmd = Command::cargo_bin("true").unwrap();
    cmd.assert().success();
}
```

此時執行 **`cargo test`**，應該會看到下列兩個測試結果：

```
running 2 tests
test true_ok ... ok
test runs ... ok
```

這些測試碼的執行順序不見得與上述程式碼的宣告順序相同。原因是 Rust 為一種安全的程式語言，能編寫並行（*concurrent*）程式碼，即程式碼的執行可跨多個執行緒（thread）。該測試運用此並行特性平行（parallel）執行多項測試碼，因此每次執行這組測試，其結果可能會以不同的順序呈現。這是語言特徵，而非 bug。若要依序執行測試碼，可以 **`cargo test test-threads=1`** 指定單執行緒執行它們。

Rust 程式結束時預設回傳零值。回想一下，*src/main.rs* 沒有直接呼叫 `std::process::exit`。即此 `true` 程式可能什麼也沒做。確定是這樣嗎？請將 *src/bin/true.rs* 的程式碼改成：

```
fn main() {}
```

執行此測試套件（test suite），驗證是否依然過關。接著用下列的原始碼（位於 *src/bin/false.rs* 檔案中）編寫我們的 `false` 程式：

```
fn main() {
    std::process::exit(1); ❶
}
```

❶ 結束執行時就 1 到 255 之間擇一數值回傳表示錯誤。

自行驗證該程式的結束碼是否非零：

```
$ cargo run --quiet --bin false
$ echo $?
1
```

接著將下列測試碼加入 *tests/cli.rs* 中，驗證程式執行時是否會回報錯誤訊息：

```
#[test]
fn false_not_ok() {
    let mut cmd = Command::cargo_bin("false").unwrap();
    cmd.assert().failure(); ❶
}
```

❶ 使用 `Assert::failure`（*https://oreil.ly/QHgoR*）確保該指令執行會以失敗呈現。

執行 **`cargo test`**，驗證所有程式皆按預期執行：

```
running 3 tests
test runs ... ok
test true_ok ... ok
test false_not_ok ... ok
```

`false` 程式的另一種做法是使用 `std::process::abort`（*https://oreil.ly/HPsKS*）。將 *src/bin/false.rs* 的內容改成：

```
fn main() {
    std::process::abort();
}
```

再次執行此測試套件，確保程式依然如預期執行。