前言

Rust 是針對「systems programming（系統程式設計）」設計的語言。

這句話需要解釋一下，因為大多數的程式設計師都不熟悉系統程式設計，然而，它是每一項工作的基礎。

當你闔上筆電時，作業系統會察覺你的動作，暫停正在運行的所有程式，關閉螢幕，讓電腦進入睡眠狀態。當你打開筆電時，螢幕與其他組件會再次通電，每一個程式都從上次離開的地方繼續運行。我們認為這是理所當然的事情，但系統程式設計師需要使用很多程式碼來實現它。

系統程式是為這些系統而寫的：

- 作業系統
- 各種設備的驅動程式
- 檔案系統
- 資料庫
- 在很便宜的設備或必須非常可靠的設備上運行的程式
- 加密
- 媒體編解碼程式（讀取和寫入音訊、視訊和圖像檔的軟體）
- 媒體處理（例如語音辨識，或照片編輯軟體）
- 記憶體管理（例如實作垃圾回收程式（garbage collector））
- 文字算繪（將文字與字型轉換成像素）

- 實作高階語言（例如 JavaScript 與 Python）
- 網路
- 虛擬化與軟體容器
- 科學模擬
- 遊戲

簡言之，系統程式設計就是在資源有限的情況下設計程式，每一個 byte 與每一個 CPU 週期都很重要。

支援基本的 app 所需的系統程式碼多得驚人。

本書教的不是系統程式設計，事實上，這本書涵蓋許多記憶體管理細節，如果你沒有寫過系統程式，在一開始，你可能認為這些細節玄之又玄。但如果你是經驗豐富的系統程式設計師，你將發現 Rust 是一種特別的語言，它是一項新工具，可排除幾十年來困擾業界的重大問題。

誰該閱讀本書？

如果你是系統程式設計師，而且準備接受 C++ 的替代方案，這本書是為你準備的。如果你是任何程式語言的高階開發者，無論你使用 C#、Java、Python、JavaScript 或其他語言，本書也是為你寫的。

但是，光是學習 Rust 還不夠，為了充分利用這個語言，你也要累積一些系統程式設計的經驗。建議你在閱讀本書的同時，使用 Rust 來實作一些業餘系統程式設計專案，你可以寫一些你從未寫過，而且可以充分利用 Rust 的速度、並行功能與安全性的作品，上述的主題清單可提供一些靈感。

著作動機

我們想要寫當初我們學習 Rust 時希望擁有的書。我們的目標是正面迎戰最重要的 Rust 新概念，清楚且深入地介紹它們，盡量避免讀者用試誤法來學習。

導覽本書

本書的前兩章對 Rust 進行簡要介紹，第 3 章介紹基本資料型態，第 4 章與第 5 章說明所有權與參考的核心概念。我們建議你完整地依序看完前五章。

第 6 章到第 10 章介紹這個語言的基本概念：運算式（第 6 章）、錯誤處理（第 7 章）、crate 與模組（第 8 章）、結構（第 9 章），以及 enum 與模式（第 10 章）。你不一定要詳細閱讀這幾章，但相信我們，千萬不要跳過「錯誤處理」一章。

第 11 章介紹 trait 與泛型，它們是最後兩項必須知道的重要概念。trait 就像 Java 或 C# 的介面，它們也是 Rust 將你的型態整合到這種語言本身的主要手段。第 12 章說明 trait 如何支援運算子多載，第 13 章介紹更多公用 trait。

學會 trait 與泛型之後，你就可以理解本書其餘的內容了。closure 與 iterator 是不可錯過的兩項重要工具，它們分別是第 14 章與第 15 章的主題。你可以用任何順序來閱讀其餘的章節，也可以在必要時查閱它們，其餘各章探討這種語言的其他功能，包括集合（第 16 章）、字串與文本（第 17 章）、輸入與輸出（第 18 章）、並行（第 19 章）、非同步程式（第 20 章）、巨集（第 21 章）、unsafe 程式碼（第 22 章），以及呼叫其他語言的函式（第 23 章）。

本書編排方式

本書使用下列的編排方式：

斜體字（*Italic*）

代表新術語、URL、email 地址、檔名，與副檔名。中文用楷體表示。

定寬字（`Constant width`）

代表程式，也在文章中代表程式元素，例如變數或函式名稱、資料庫、資料類型、環境變數、陳述式，與關鍵字。

定寬粗體字（**`Constant width bold`**）

代表應由使用者親自輸入的命令或其他文字。

定寬斜體（*`Constant width italic`*）

這種文字應換成使用者提供的值，或由程式脈絡決定的值。

第一章

系統程式設計師也可以使用好東西

> 在某些情況下（例如，*Rust* 所針對的情況），比競爭對手快 *10* 倍甚至 *2* 倍是決定性因素，它決定了一個系統在市場上的命運，和硬體市場一樣。
>
> —Graydon Hoare（*https://oreil.ly/Akgzc*）

> 現在的電腦都是平行的…
> 設計平行程式等於設計程式。
>
> —Michael McCool 等，*Structured Parallel Programming*

> 民族國家的攻擊者利用 *TrueType* 解析器的缺陷來監視別人，所有軟體都是安全敏感的。
>
> —Andy Wingo（*https://oreil.ly/7dnHr*）

本書開頭列出這三條引言是有原因的，不過，讓我們從一個謎題談起。下面的 C 程式會產生什麼？

```
int main(int argc, char **argv) {
  unsigned long a[1];
  a[3] = 0x7ffff7b36cebUL;
  return 0;
}
```

今天早上，這段程式在 Jim 的筆電中印出：

```
undef: Error: .netrc file is readable by others.
undef: Remove password or make file unreadable by others.
```

然後崩潰了。當你在自己的電腦裡執行它時，你可能會看到不一樣的行為。為何如此？

這段程式有缺陷。陣列 `a` 只有一個元素長，所以根據 C 語言標準，`a[3]` 是一種未定義行為（*undefined behavior*）：

> 它是在使用「不可移植的，或錯誤的程式結構」或「錯誤的資料」時發生的行為，國際標準未規定此時該怎麼辦？

未定義行為不只會造成不可預測的結果，該標準明確地允許程式做任何事情。這個例子將值存入陣列的第四個元素剛好破壞了函式呼叫堆疊（call stack），因此當 `main` 函式 `return` 時，程式不會優雅地退出，而是跳到標準 C 程式庫的程式裡，從用戶的主目錄裡的檔案提取密碼。程式沒有正常地執行。

C 與 C++ 有上百條避免未定義行為的規則，這些規則大多是常識：不要存取不該存取的記憶體、別讓算術運算子溢位、不要除以零…等，但是編譯器並不強制執行這些規則，它甚至沒有義務檢查公然違規的行為。事實上，上述的程式在編譯時不會出現錯誤或警告，避免未定義行為的責任完全落在你這位程式設計師身上。

據經驗，程式設計師不會妥善地記錄它們。研究員 Peng Li 在猶他大學就學時，曾經修改 C 與 C++ 編譯器，讓它們編譯出來的程式可以回報自己是否執行了某種未定義行為。他發現，幾乎所有程式都執行了未定義行為，包括高標準且備受尊敬的專案。如果有人認為他可以在 C 與 C++ 裡避免未定義行為，那就相當於他認為只要了解下棋規則，即可贏得棋局。

偶發的奇怪訊息或崩潰也許是一種品質問題，但無意間寫出來的未定義行為向來是安全缺陷的主因之一，這種安全缺陷最早可追溯到 1988 年出現的莫里斯蠕蟲（Morris Worm），它使用上述技術的變體，透過早期的網際網路從一台電腦感染另一台電腦。

所以 C 與 C++ 讓程式設計師面臨一種尷尬的情境：這些語言是系統程式設計的業界標準，但是它們對程式員的要求，幾乎保證會持續帶來層出不窮的崩潰與資訊安全問題。我們的謎題帶來一個更大的問題：我們真的沒辦法做得更好嗎？

Rust 為你承擔重任

我們的答案可以從本書的三個引言裡找到。第三個引言是指 2010 年有一隻電腦蠕蟲入侵工業控制設備，利用未定義行為和許多其他技術來控制受害者的電腦，那個未定義行為出現在一段解析 TrueType 字體的程式裡面。可以確定的是，那段程式的作者並未想到它會被那樣子使用，這個故事告訴我們，不是只有作業系統和伺服器需要擔心安全問題，只要你的軟體需要處理不可信任的來源送來的資料，它就可能變成入侵目標。

Rust 語言給你一個簡單的承諾：只要編譯器認為你的程式沒有問題，它就沒有未定義行為。懸空指標（dangling pointer）、重複釋出（double-free）、對空指標解參考…等問題都會在編譯期抓到。陣列參考是透過編譯期與執行期檢查來保護的，所以不會出現緩衝區溢位（buffer overrun）：Rust 會產生一條錯誤訊息，並安全地退出，而不會像可悲的 C 程式那樣。

此外，Rust 的目標是用起來既安全且愉快。為了更有力地保證程式的行為，Rust 對你的程式施加了比 C 和 C++ 更多的限制，你必須透過練習與經驗來習慣這些限制。但是整體來說，這個語言比較靈活，而且更有表現力，Rust 程式及其應用領域的廣度可以證明這一點。

根據我們的經驗，「相信這個語言能夠抓到更多錯誤」可以鼓勵我們嘗試更有企圖心的專案。如果記憶體管理和指標有效性等問題可以解決，那麼修改大量、複雜程式的風險就會降低。而且，如果 bug 絕對不會破壞不相關的部分，偵錯將容易許多。

當然，Rust 無法偵測的 bug 仍然很多，但是在實務上，將未定義行為排除可以大幅改善開發品質。

平行程式設計被馴服了

在 C 與 C++ 裡面使用並行（concurrency）是出了名的困難。開發者通常只會在無法用單執行緒程式來實現性能目標時，才會轉而使用並行。但是第二句引言認為，平行化對現代電腦而言太重要了，不能視為最終手段。

事實上，在 Rust 裡面確保記憶體安全的那些限制，也保證了 Rust 程式不會出現資料爭用。你可以在執行緒之間自由地分享資料，只要它不會改變即可。會改變的資料只能使用同步基元（synchronization primitive）來操作。你可以使用所有的傳統並行工具，包括互斥鎖（mutex）、條件變數、通道（channel）、原子（atomic）…等。Rust 會檢查你有沒有正確地使用它們。

所以 Rust 是一種充分利用現代多核心電腦能力的優秀語言。除了一般的並行基元之外，Rust 生態系統也提供許多其他的程式庫，可協助你將複雜的工作負擔平均分給許多處理器、使用 Read-Copy-Update 之類的無鎖同步機制…等。

然而，Rust 仍然很快

最後要討論第一條引言。Rust 的目標與 Bjarne Stroustrup 在其論文「Abstraction and the C++ Machine Model」中敘述的 C++ 目標一致：

> 一般來說，*C++* 的實作應遵守零額外開銷（*zero-overhead*）原則：用不到的東西不需要為它付出代價，甚至更進一步，用最好的程式來編寫你所使用的東西。

系統程式設計的目標通常是將機器的性能推到極限，對遊戲而言，那就是讓整台機器全力為玩家創造最棒的體驗。對網頁瀏覽器而言，瀏覽器的效率就是網頁內容的作者可發揮的上限。在機器固有的限制之內，你必須盡量把記憶體與處理器的工作重點放在內容本身。同樣的原則也適用於作業系統：kernel 必須把機器的資源留給用戶的程式使用，而不是自己耗用它們。

但是 Rust「很快」到底是什麼意思？任何通用的語言都可能寫出緩慢的程式。比較準確的說法是，如果你準備投入資源，設計出充分利用底層機器的程式，Rust 會支持你的付出。這種語言具備高效的預設機制，可讓你控制記憶體的使用情況，以及處理器應專心處理哪裡。

Rust 可促進合作

本章其實還有隱藏版的第四條引言：「系統程式設計師也可以擁有好東西。」意思是 Rust 支援程式共享與重複使用。

Rust 的程式包（package）管理器和組建工具 Cargo，可讓你輕鬆地使用別人在 Rust 的公用程式包版本庫（crates.io 網站）發表的程式庫。你只要將程式庫名稱與版本號碼加入一個檔案，Cargo 就會幫你下載程式庫，並且下載該程式庫所使用的其他程式庫，並將所有程式庫連結起來。你可以將 Cargo 視為 Rust 對 NPM 或 RubyGems 的回應，強調健全的版本管理，以及可重現的組建結果。現在有一些流行的 Rust 程式庫提供了所有東西，包括現成的序列化、HTTP 用戶端與伺服器，以及現代的圖形 API。

更進一步說，這個語言本身也是為了支援合作而設計的：Rust 的 trait 與泛型可讓你建立具備靈活介面的程式庫，讓它們可在許多不同的環境中發揮作用。Rust 的標準程式庫也提供了一組基本的核心型態，可為常見的情況建立共享的規範，讓你更容易同時使用不同的程式庫。

下一章將用幾段 Rust 小程式來具體說明本章提出的廣泛主張，並展示這個語言的優點。

第二章

Rust 導覽

Rust 為本書作者帶來一項挑戰：這種語言的特點並不是可用一頁來展示的神奇功能，其特點在於，它的各個部分在設計上可以順暢地合作，以滿足上一章提出來的目標——安全、高效地設計系統程式。這種語言的各個組件對其他部分而言都是最合理的安排。

所以，我們不會一次討論一種語言功能，而是用幾個完整的小例子來介紹這種語言的多種功能，這些例子的背景包括：

- 先熱身一下，用一個程式以命令列引數來進行簡單的計算，並且寫一些單元測試。這個例子將展示 Rust 的核心型態，並介紹 *trait*。
- 接下來要建構一個 web 伺服器。我們將使用第三方程式庫來處理 HTTP 的細節，並介紹字串處理、closure，以及錯誤處理。
- 第三個程式將繪製一幅漂亮的碎形（fractal）圖，將計算工作分配給多個執行緒，以提升速度。這個程式包含一個泛形函式範例，說明如何處理像素緩衝區之類的東西，並展示 Rust 的並行。
- 最後展示一個可靠的命令列工具，它可以用正規表達式來處理檔案。你將看到 Rust 標準程式庫處理檔案的工具，以及最常用的第三方正規表達式程式庫。

Rust 承諾在防止未定義行為時盡量不影響性能，這個承諾影響了系統的每個部分的設計，從標準資料結構，例如 vector（向量）與字串，到 Rust 程式使用第三方程式庫的方式。本書將詳細討論管理這些事情的細節。但現在，我們想要讓你知道 Rust 是一種能幹的、用起來很愉快的語言。

當然，你要先在電腦上安裝 Rust。

rustup 與 Cargo

安裝 Rust 的最佳做法是使用 `rustup`。前往 *https://rustup.rs* 並按照下面的指示來操作。

或者，你也可以到 Rust 網站（*https://oreil.ly/4Q2FB*）取得 Linux、macOS 與 Windows 的預先組建程式包。有些作業系統版本也內建了 Rust。我們比較喜歡 `rustup`，因為它是管理 Rust 版本的工具，就像 Ruby 的 RVM，或 Node 的 NVM。例如，當新版的 Rust 被發表時，你只要輸入 `rustup update` 就可以升級它。

無論如何，一旦你完成安裝，你就可以在命令列使用三個新命令：

```
$ cargo --version
cargo 1.49.0 (d00d64df9 2020-12-05)
$ rustc --version
rustc 1.49.0 (e1884a8e3 2020-12-29)
$ rustdoc --version
rustdoc 1.49.0 (e1884a8e3 2020-12-29)
```

其中的 `$` 是命令列提示符號，在 Windows 上，它是 `C:\>` 之類的東西。在這個抄本（transcript）中，我們執行安裝好的三條命令，要求它們回報各自的版本。以下依序介紹這三個命令：

- `cargo` 是 Rust 的編譯管理器、程式包管理器，以及通用工具。你可以用 Cargo 來建立新專案、組建和執行程式，以及管理程式所使用的任何外部程式庫。
- `rustc` 是 Rust 編譯器。我們通常讓 Cargo 為我們呼叫編譯器，但有時直接執行它很有幫助。
- `rustdoc` 是 Rust 文件工具。如果你在原始碼裡面以格式正確的註釋來撰寫文件，`rustdoc` 可以用它們來建立格式正確的 HTML。如同 `rustc`，我們通常讓 Cargo 為我們執行 `rustdoc`。

為了方便，Cargo 可以為我們建立一個新 Rust 程式包，使用一些適當安排的標準參考資訊（metadata）：

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

這個命令會建立一個名為 *hello* 的新程式包目錄，可組建一個命令列可執行檔。

我們來看一下這個程式包的最上層目錄：

```
$ cd hello
$ ls -la
total 24
drwxrwxr-x.  4 jimb jimb 4096 Sep 22 21:09 .
drwx------. 62 jimb jimb 4096 Sep 22 21:09 ..
drwxrwxr-x.  6 jimb jimb 4096 Sep 22 21:09 .git
-rw-rw-r--.  1 jimb jimb    7 Sep 22 21:09 .gitignore
-rw-rw-r--.  1 jimb jimb   88 Sep 22 21:09 Cargo.toml
drwxrwxr-x.  2 jimb jimb 4096 Sep 22 21:09 src
```

我們可以看到 Cargo 建立一個 *Cargo.toml* 檔來保存程式包的參考資訊。這個檔案目前還沒有什麼內容：

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

# 其他索引鍵及其定義請參考：
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

如果我們的程式依賴其他的程式庫，我們可以將它們寫在這個檔案裡，Cargo 會負責幫我們下載、組建與更新這些程式庫。我將在第 8 章詳細討論 *Cargo.toml* 檔。

Cargo 為我們的程式包做了一項設定，讓它可以使用 `git` 版本控制系統，並建立了一個 *.git* 參考資訊子目錄，以及一個 *.gitignore* 檔。你可以要求 Cargo 跳過這一步，做法是在命令列執行 `cargo new` 時傳遞 `--vcs none`。

在 *src* 子目錄裡面有實際的 Rust 程式：

```
$ cd src
$ ls -l
total 4
-rw-rw-r--. 1 jimb jimb 45 Sep 22 21:09 main.rs
```

看來，Cargo 已經開始為我們編寫程式了。在 *main.rs* 檔案裡面有這段文字：

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

在 Rust 裡面，你甚至不需要自己撰寫「Hello, World!」程式。以上就是 Rust 程式樣板的規模：2 個檔案，共 13 行。

你可以在程式包內的任何目錄中呼叫 `cargo run` 命令來組建與執行程式：

```
$ cargo run
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.28s
     Running `/home/jimb/rust/hello/target/debug/hello`
Hello, world!
```

Cargo 呼叫 Rust 編譯器（`rustc`），然後執行它產生的可執行檔。Cargo 將可執行檔放在程式包最上層的 *target* 子目錄裡面：

```
$ ls -l ../target/debug
total 580
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 build
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 deps
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 examples
-rwxrwxr-x. 1 jimb jimb 576632 Sep 22 21:37 hello
-rw-rw-r--. 1 jimb jimb    198 Sep 22 21:37 hello.d
drwxrwxr-x. 2 jimb jimb     68 Sep 22 21:37 incremental
$ ../target/debug/hello
Hello, world!
```

完成執行後，Cargo 可以幫忙清除生成的檔案：

```
$ cargo clean
$ ../target/debug/hello
bash: ../target/debug/hello: No such file or directory
```

Rust 函式

Rust 的語法被故意設計成非原創的，如果你熟悉 C、C++、Java 或 JavaScript，你應該看得懂 Rust 程式的一般結構。下面這個函式使用 Euclid 演算法來計算兩個整數的最大公因數（*https://oreil.ly/DFpyb*）。你可以將這段程式加入 *src/main.rs* 的結尾：

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m;
            m = n;
            n = t;
        }
        m = m % n;
    }
    n
}
```

fn 關鍵字（讀成「fun」）代表接下來有一個函式。我們定義一個名為 gcd 的函式，它有兩個參數，n 與 m，兩者的型態皆為 u64，即 unsigned（無正負號）64-bit 整數。在 -> 後面的型態是回傳型態，這個函式回傳一個 u64 值。用 4 個空格來縮排是 Rust 的標準風格。

Rust 的機器整數型態名稱反映了它們的大小與符號：i32 是 signed（帶正負號）32-bit 整數，u8 是 unsigned 8-bit 整數（用於「byte」值），以此類推。isize 與 usize 型態保存指標大小的 signed 與 unsigned 整數，它們在 32-bit 平台上是 32 bits，在 64-bit 平台上是 64 bits。Rust 也有兩個浮點數型態，f32 與 f64，它們是 IEEE 單精度與雙精度浮點型態，就像 C 與 C++ 裡面的 float 與 double。

在預設情況下，一旦變數被初始化，它的值就無法改變，但是在參數 n 與 m 的前面加上 mut 關鍵字（讀成「mute」，*mutable*（可變）的簡寫）可讓函式的本體對它們賦值。在實務上，大多數的變數都不能賦值，在可以賦值的變數前面加上 mut 關鍵字可提醒程式的讀者。

函式的本體始於呼叫 assert! 巨集的地方，這個巨集的目的是確認兩個引數都不是零。! 字元代表它是一個巨集呼叫式，不是函式呼叫式。如同 C 和 C++ 裡面的 assert 巨集，Rust 的 assert! 會確定它的引數是 true，如果不是，它會終止程式，並提供有用的訊息，包括未通過檢查的原始碼位置，這種突然終止的情況稱為 *panic*（恐慌）。C 和 C++ 的斷言（assertion）是可以跳過的，但 Rust 一定會檢查斷言，無論程式如何編譯。Rust 也有一個 debug_assert! 巨集，當程式被編譯成加速版時，它的斷言會被跳過。

這個函式的核心是一個 while 迴圈，裡面有一個 if 陳述式與一個賦值。與 C 和 C++ 不同的是，Rust 的條件運算式不需要使用括號，但它們控制的陳述式需要加上大括號。

我們用 let 陳述式來宣告區域變數，例如函式中的 t。只要 Rust 可以從變數的用法推斷 t 的型態，我們就不需要寫出它的型態。在這個函式中，t 能用的型態只有 u64，與 m 和 n 的型態一樣。Rust 只會在函式主體內推斷型態，你必須寫出函式參數與回傳值的型態，如同這裡的做法。你可以這樣寫出 t 的型態：

```
let t: u64 = m;
```

Rust 有 return 陳述式，但 gcd 函式不需要它。如果函式本體的結尾是一個運算式，而且結尾沒有分號，那個運算式就是函式的回傳值。事實上，被大括號包起來的任何區域都可以當成一個運算式。例如，這是印出一條訊息，然後產生 x.cos() 作為值的運算式：

```
{
    println!("evaluating cos x");
    x.cos()
}
```

當控制流程跑到函式的結尾時，Rust 通常使用這種形式來建立函式的值，只在函式的中間提早返回時，明確地使用 `return` 陳述式。

編寫與執行單元測試

Rust 內建簡單的測試功能，你可以在 *src/main.rs* 的結尾加入這段程式來測試 `gcd` 函式：

```
#[test]
fn test_gcd() {
    assert_eq!(gcd(14, 15), 1);

    assert_eq!(gcd(2 * 3 * 5 * 11 * 17,
                   3 * 7 * 11 * 13 * 19),
               3 * 11);
}
```

我們定義一個名為 `test_gcd` 的函式，它呼叫 `gcd`，並檢查該函式是否回傳正確的值。在函式定義上面的 `#[test]` 代表 `test_gcd` 是一個測試函式，如此一來，當我們進行一般的編譯時會跳過它，但是在使用 `cargo test` 命令來執行程式時，會加入並自動呼叫它。我們可以在原始碼裡面到處撰寫測試函式，將它們寫在被它們檢查的程式旁邊，`cargo test` 會自動找到它們並執行它們全部。

`#[test]` 標記是一種屬性（*attribute*），屬性是一種開放式系統，它用額外的資訊來標記函式和其他宣告式，很像 C++ 與 C# 的屬性，或 Java 的註解（annotation）。它們的用途是控制編譯器的警告訊息和編寫風格檢查、按條件加入程式碼（就像 C 和 C++ 的 `#ifdef`）、告訴 Rust 如何與其他語言的程式互動…等。我們接下來會看到更多屬性的例子。

將 `gcd` 與 `test_gcd` 的定義加入本章開頭建立的 *hello* 程式包之後，即可在程式包的子目錄中執行這個測試：

```
$ cargo test
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished test [unoptimized + debuginfo] target(s) in 0.35s
     Running unittests (/home/jimb/rust/hello/target/debug/deps/hello-2375...)

running 1 test
test test_gcd ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

處理命令列引數

為了讓程式接收一系列的命令列引數，並印出它們的最大公因數，我們將 `main` 函式放入 *src/ main.rs* 如下：

```
use std::str::FromStr;
use std::env;

fn main() {
    let mut numbers = Vec::new();

    for arg in env::args().skip(1) {
        numbers.push(u64::from_str(&arg)
                     .expect("error parsing argument"));
    }

    if numbers.len() == 0 {
        eprintln!("Usage: gcd NUMBER ...");
        std::process::exit(1);
    }

    let mut d = numbers[0];
    for m in &numbers[1..] {
        d = gcd(d, *m);
    }

    println!("The greatest common divisor of {:?} is {}",
             numbers, d);
}
```

這段程式有相當的長度，讓我們來分別講解各個部分：

```
use std::str::FromStr;
use std::env;
```

第一個 `use` 宣告式將標準程式庫 *trait* `FromStr` 加入作用域。trait 就是該型態可以實作的方法。實作了 `FromStr` trait 的任何型態都有 `from_str` 方法，可以嘗試從一個字串解析出該型態的值。`u64` 型態實作了 `FromStr`，我們將呼叫 `u64::from_str` 來解析命令列引數。雖然其他地方都不會使用 `FromStr` 這個名稱，但我們必須將 trait 加入作用域才可以使用它的方法。第 11 章會介紹 trait。

第二個 `use` 宣告式加入 `std::env` 模組，它提供一些實用的函式與型態來讓你和執行環境互動，包括 `args` 函式，可用來讀取程式的命令列引數。

接下來是程式的 main 函式：

```
fn main() {
```

我們的 main 函式不回傳值，所以可以省略參數後面的 -> 與回傳型態。

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

我們宣告一個可變區域變數 numbers，並將它設為空向量。Vec 是 Rust 的可擴展向量型態，相當於 C++ 的 std::vector、Python 的 list、JavaScript 的 array。雖然這種向量在設計上可以動態伸縮，但是為了讓 Rust 允許我們將數字推入它的末端，我們仍然必須將這個變數標成 mut。

numbers 的型態是 Vec<u64>，它是以 u64 值組成的向量，但是與之前一樣，我們不需要寫出這件事，Rust 會幫我們推斷出來，原因是我們推入向量的值是 u64，也因為我們將向量的元素傳給 gcd，它只接收 u64 值。

```
for arg in env::args().skip(1) {
```

我們使用 for 迴圈來處理命令列引數，依序將變數 arg 設為各個引數，並執行迴圈本體。

std::env 模組的 args 函式會回傳一個 *iterator*，iterator 是一個可以按需求產生各個引數的值，它也會在結束提示我們。iterator 在 Rust 裡面很普遍，標準程式庫還有其他的 iterator 可產生向量的元素、檔案的行、從通訊通道接收的訊息，以及可迭代的幾乎所有其他東西。Rust 的 iterator 非常有效率，編譯器通常可以將它們轉換成手寫迴圈般的程式。我們將在第 15 章展示它如何運作，並提供一些範例。

iterator 除了可以和 for 迴圈一起使用之外，也提供了大量的方法可以直接使用。例如，args 回傳的 iterator 產生的第一個值一定是我們執行的程式的名稱，我們想要跳過它，所以呼叫 iterator 的 skip 方法來產生省略了第一個值的新 iterator。

```
numbers.push(u64::from_str(&arg)
             .expect("error parsing argument"));
```

我們呼叫 u64::from_str 來將命令列引數 arg 解析成 unsigned 64-bit 整數。u64::from_st 與對著某些 u64 值呼叫的方法不同，它是與 u64 型態關聯的函式，類似 C++ 和 Java 的靜態（static）方法。from_str 函式不會直接回傳 u64，而是回傳一個 Result 值，它會指出解析成功還是失敗。Result 值有兩個 variant（變式）：

- 寫成 Ok(v) 的值，代表解析成功，v 是產生的值。
- 寫成 Err(e) 的值，代表解析失敗，e 是解釋為何錯誤的錯誤值。

只要函式做的事情可能會失敗，例如進行輸入或輸出，或是做與作業系統互動的其他事情，它都可以回傳 `Result` 型態，用 `Ok` variant 來傳送成功的結果（已轉換的 bytes 數、已打開的檔案…等），用 `Err` variant 來傳送錯誤碼，指出為何出錯。與大多數現代語言不同的是，Rust 沒有例外（exception），它用 `Result` 或 panic 來處理所有錯誤，第 7 章會說明這個主題。

我們使用 `Result` 的 `expect` 方法來檢查解析成功與否，如果結果是 `Err(e)`，`expect` 會印出一條訊息，裡面有 `e` 的敘述，並且立刻退出程式。如果結果是 `Ok(v)`，`expect` 只會回傳 `v` 本身，我們將它推入數字向量的結尾。

```
if numbers.len() == 0 {
    eprintln!("Usage: gcd NUMBER ...");
    std::process::exit(1);
}
```

數字空集合沒有最大公因數，所以我們先確定向量至少有一個元素，如果沒有，就退出程式，並送出一條錯誤訊息。我們使用 `eprintln!` 巨集來將錯誤訊息寫到標準錯誤輸出串流。

```
let mut d = numbers[0];
for m in &numbers[1..] {
    d = gcd(d, *m);
}
```

這個迴圈使用 `d` 作為運行值，迴圈會修改它，並用它來保存已經處理過的數字的最大公因數。同樣的，為了可在迴圈中設定 `d` 的值，我們必須將它標成可變。

`for` 迴圈有兩個奇怪的地方。首先，`for m in &numbers[1..]` 裡面的 `&` 有什麼作用？其次，在 `gcd(d, *m);` 裡，`*m` 的 `*` 有什麼作用？這兩個細節是相輔相成的。

目前我們的程式只處理簡單的值，例如整數，它們都被放在固定大小的記憶體區塊裡面。但接下來，我們要迭代向量了，它可能有任何大小，或許非常大。Rust 在處理這種值的時候很謹慎：它希望讓程式設計師控制記憶體的使用，明確表示每一個值的生命期，同時也要確保記憶體在用不到時立刻釋出。

所以在迭代時，我們要告訴 Rust，向量的*所有權*仍然屬於 `numbers`，我們只是*借用*它的元素來執行迴圈。`&numbers[1..]` 的 `&` 運算子代表借用向量第二個之後的元素的*參考*（*reference*）。`for` 迴圈會迭代參考的元素，讓 `m` 依序借用每一個元素。`*m` 的 `*` 運算子會解*參考*（*dereference*）`m`，產生它參考的值，那個值是接下來要傳給 `gcd` 的 `u64`。最後，因為 `numbers` 擁有向量，所以 Rust 會在 `main` 的結尾，當 `numbers` 離開作用域時自動釋出它。

Rust 的所有權規則與參考規則是 Rust 管理記憶體和執行安全的並行的關鍵，我們將在第 4 章與第 5 章詳細討論它們。你必須習慣這些規則才能習慣 Rust，但是在這趟入門的旅途中，你只要知道，`&x` 借用 `x` 的參考，而 `*r` 是 `r` 參考所引用的值即可。

我們繼續看這段程式：

```
    println!("The greatest common divisor of {:?} is {}",
             numbers, d);
```

迭代 `numbers` 的元素之後，程式將結果印到標準輸出串流。`println!` 巨集接收一個模板字串，將模板字串裡面的 `{...}` 換成其餘的引數，並將結果寫到標準輸出串流。

C 和 C++ 要求 `main` 在程式成功結束時回傳零，在出錯時回傳非零的退出狀態，但是 Rust 假設一旦 `main` 返回，那就代表程式成功完成了。你必須明確地呼叫 `expect` 或 `std::process::exit` 等函式，才能在程式終止時顯示錯誤狀態碼。

`cargo run` 命令可將引數傳給程式，我們來嘗試一下我們的命令列處理程式：

```
$ cargo run 42 56
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.22s
     Running `/home/jimb/rust/hello/target/debug/hello 42 56`
The greatest common divisor of [42, 56] is 14
$ cargo run 799459 28823 27347
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `/home/jimb/rust/hello/target/debug/hello 799459 28823 27347`
The greatest common divisor of [799459, 28823, 27347] is 41
$ cargo run 83
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `/home/jimb/rust/hello/target/debug/hello 83`
The greatest common divisor of [83] is 83
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `/home/jimb/rust/hello/target/debug/hello`
Usage: gcd NUMBER ...
```

我們在這一節使用 Rust 標準程式庫的一些功能。如果你想知道還有什麼功能可用，強烈建議你參考 Rust 的網路文件，它有即時搜尋功能，可讓你輕鬆地探索，甚至有原始碼的超連結。當你安裝 Rust 時，`rustup` 命令會在你的電腦安裝它。你可以在 Rust 網站瀏覽標準程式庫文件（*https://oreil.ly/CGsB5*），或是用下面的命令，在瀏覽器內瀏覽它：

```
$ rustup doc --std
```

提供 web 網頁

Rust 的優勢之一在於它在 crates.io 網站公布了許多免費的程式庫。你可以用 `cargo` 命令來讓程式使用 crates.io 程式包：它會下載正確的程式包版本、組建它，並按需求修改它。Rust 程式包稱為 *crate*，無論它是程式庫還是可執行檔；Cargo 與 crates.io 的名稱都是從這個字衍生的。

為了展示它如何運作，我們將使用 `actix-web` web 框架 crate、`serde` 序列化 crate，以及它們所使用的各種其他 crate 來建立一個簡單的 web 伺服器。如圖 2-1 所示，我們的網站會提示用戶輸入兩個數字，並計算它們的最大公因數。

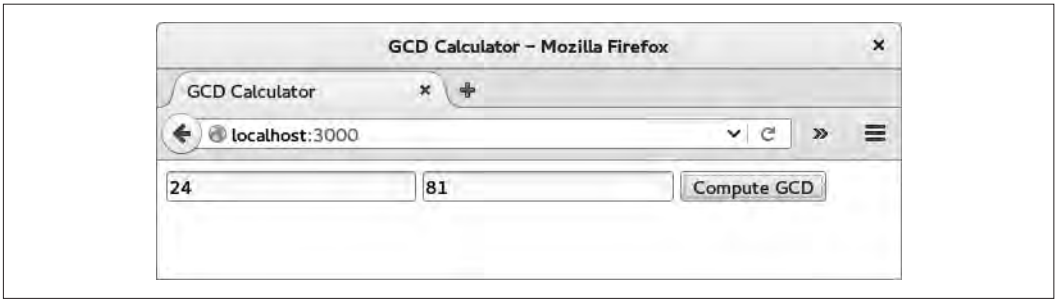

圖 2-1　供用戶計算 GCD 的網頁

首先，我們讓 Cargo 為我們建立一個新的程式包，名為 `actix-gcd`：

```
$ cargo new actix-gcd
     Created binary (application) `actix-gcd` package
$ cd actix-gcd
```

然後編輯新專案的 *Cargo.toml* 檔案，在裡面列出我們想要使用的程式包，其內容為：

```
[package]
name = "actix-gcd"
version = "0.1.0"
edition = "2021"

# 在這個網址有更多索引鍵和它們的定義：
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
actix-web = "1.0.8"
serde = { version = "1.0", features = ["derive"] }
```

在 *Cargo.toml* 的 `[dependencies]` 段落裡面的每一行都是一個 crates.io 的 crate 的名稱，以及我們想要使用的版本。在這個例子裡，我們想要使用 `1.0.8` 版的 `actix-web` crate，以及 `1.0` 版的 `serde` crate。在 crates.io 可能有更新的版本，但列出具體的版本可以確保你的程式在有新程式包問世的情況下也可以成功編譯。我們將在第 8 章詳細討論版本管理。

crate 可能有選用的功能，它們是並非所有用戶都需要的介面或實作，但適合放在該 crate 裡。`serde` crate 提供一種絕妙的 web 表單資料處理工具，但根據 `serde` 的文件，我們必須選擇 crate 的 `derive` 功能才能使用它，這就是為什麼我們在 *Cargo.toml* 檔案裡面那樣寫。

注意，你只要列出打算直接使用的 crate 即可，`cargo` 會加入這些 crate 所使用的其他 crate。

在第一個版本裡，我們先讓 web 伺服器簡單一點：它只提供一個網頁來提示用戶輸入計算用的數字。我們在 *actix-gcd/src/main.rs* 裡面加入下面的程式：

```
use actix_web::{web, App, HttpResponse, HttpServer};

fn main() {
    let server = HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(get_index))
    });

    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000").expect("error binding server to address")
        .run().expect("error running server");
}

fn get_index() -> HttpResponse {
    HttpResponse::Ok()
        .content_type("text/html")
        .body(
            r#"
                <title>GCD Calculator</title>
                <form action="/gcd" method="post">
                <input type="text" name="n"/>
                <input type="text" name="m"/>
                <button type="submit">Compute GCD</button>
                </form>
            "#,
        )
}
```