

系統程式期末報告

資工二_110710538_李宗翰

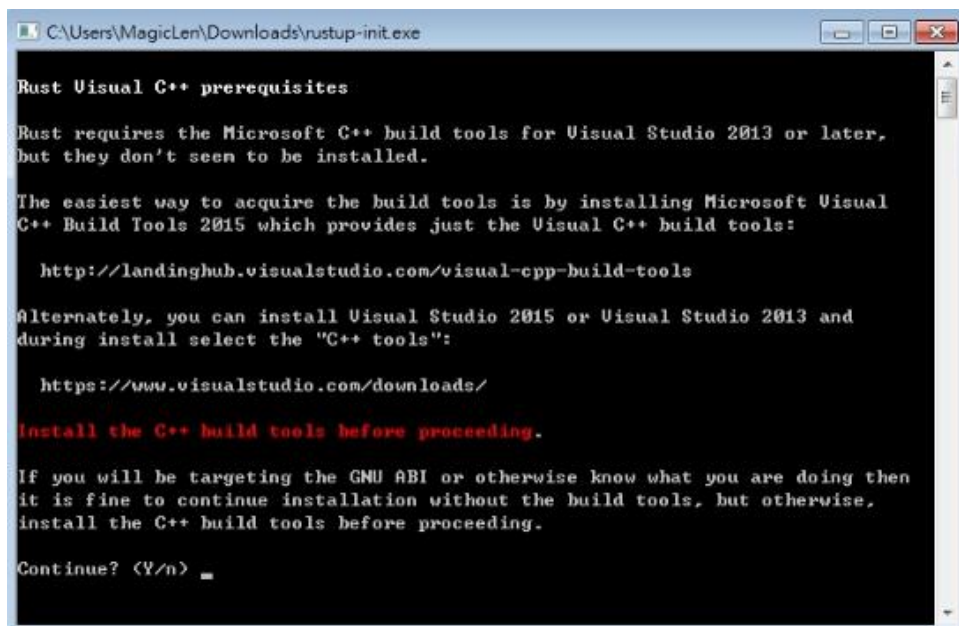
主題:安裝及學習 rust

安裝 rustup(windows):

透過 <https://www.rust-lang.org/zh-TW/tools/install> 下載

rustup-init.exe 安裝檔

接著執行 rustup-init.exe 安裝檔。安裝檔可能會提示說需要微軟的 Visual C++ Build Tools。



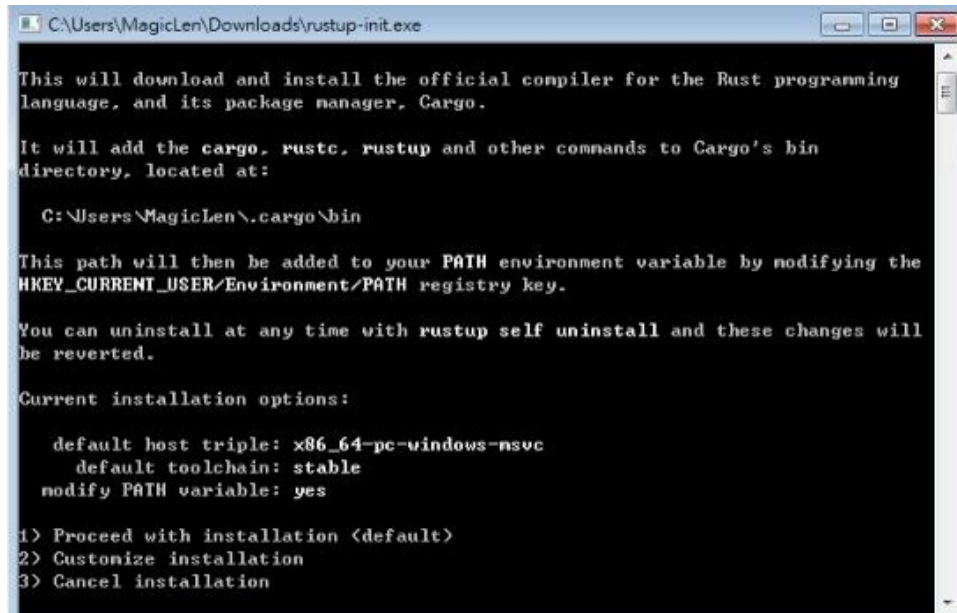
可以利用以下連結下載到「Visual C++ Build Tools」的安裝程式—「Build Tools for Visual Studio 2017」。

<https://www.visualstudio.com/downloads/>

確定.NET Framework 4.6 以上的版本安裝好之後，就可以執行「Build Tools for Visual Studio 2017」的安裝程式，來安裝「Visual C++ Build Tools」(Windows SDK 也必須安裝)。



如果「Visual C++ Build Tools」有安裝成功，再次執行 `rustup-init.exe`，此時它就不會再出現需要安裝「Visual C++ Build Tools」的訊息了。



```
C:\Users\MagicLen\Downloads>rustup-init.exe

This will download and install the official compiler for the Rust programming
language, and its package manager, Cargo.

It will add the cargo, rustc, rustup and other commands to Cargo's bin
directory, located at:

    C:\Users\MagicLen\.cargo\bin

This path will then be added to your PATH environment variable by modifying the
HKEY_CURRENT_USER\Environment\PATH registry key.

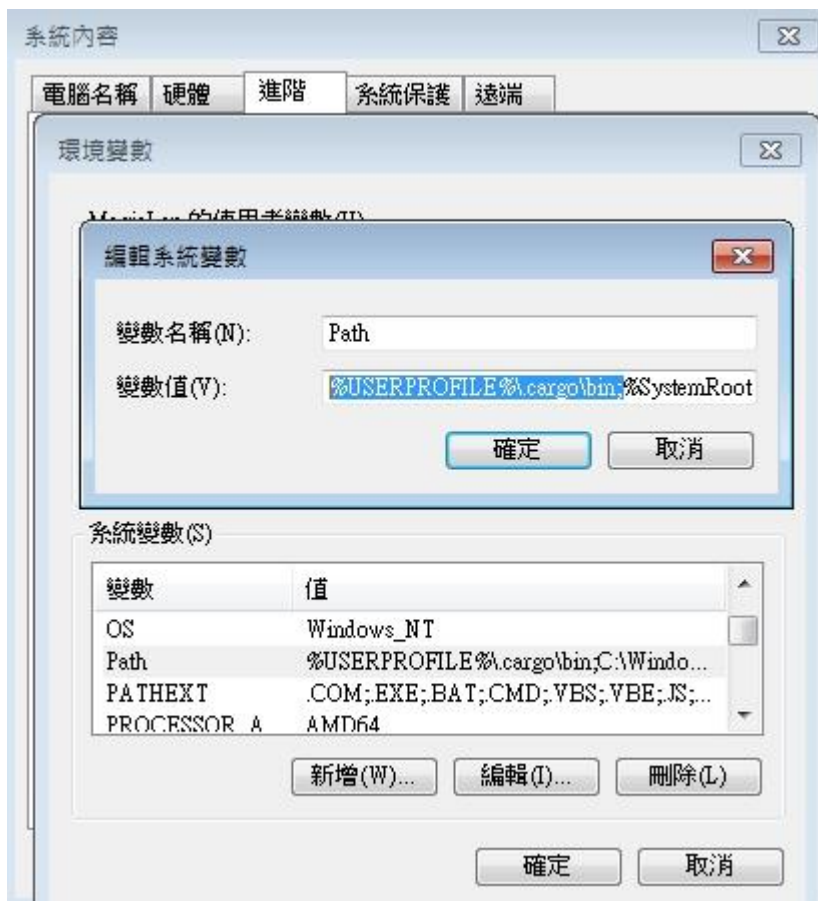
You can uninstall at any time with rustup self uninstall and these changes will
be reverted.

Current installation options:

    default host triple: x86_64-pc-windows-msvc
    default toolchain: stable
    modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
```

選擇「Proceed with installation」就能開始安裝。將 Rust 和 Rust 包含的相關工具安裝好之後，還需要將 `%USERPROFILE%\.cargo\bin` 路徑加至 PATH 環境變數。



參考資料 1:<https://magiclens.org/rust-introduction/>

參考資料 2:<https://www.rust-lang.org/zh-TW/tools/install>

創建項目目錄

首先創建一個存放 Rust 代碼的目錄。Rust 並不關心代碼的存放位置，不過對於本書的練習和項目來說，我們建議你在 home 目錄中創建 *projects* 目錄，並將你的所有項目存放在這裡。

打開終端並輸入如下命令創建 *projects* 目錄，並在 *projects* 目錄中為“Hello, world!”項目創建一個目錄。

對於 Linux、macOS 和 Windows PowerShell，輸入：

```
$ mkdir ~/projects$ cd ~/projects$ mkdir  
hello_world$ cd hello_world
```

對於 Windows CMD，輸入：

```
> mkdir "%USERPROFILE%\projects"> cd /d  
"%USERPROFILE%\projects"> mkdir hello_world>  
cd hello_world
```

編寫並運行 Rust 程序

接下來，新建一個源文件，命名為 *main.rs*。Rust 源文件總是以 *.rs* 擴展名結尾。如果文件名包含多個單詞，使用下劃線分隔它們。例如命名為 *hello_world.rs*，而不是 *helloworld.rs*。

現在打開剛創建的 *main.rs* 文件，輸入示例 1-1 中的代碼。

文件名: main.rs

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

示例 1-1:一個打印 Hello, world!的程序

保存文件，並回到終端窗口。在 Linux 或 macOS 上，輸入如下命令，編譯並運行文件：

```
$ rustc main.rs$ ./mainHello, world!
```

在 Windows 上，輸入命令 `.\main.exe`，而不是 `./main`：

```
> rustc main.rs> .\main.exeHello, world!
```

不管使用何種操作系統，終端應該打印字符串 `Hello, world!`。如果沒有看到這些輸出，回到安裝部分的“故障排除”小節查找有幫助的方法。

如果 `Hello, world!` 出現了，恭喜你！你已經正式編寫了一個 Rust 程序。現在你成為一名 Rust 程序員，歡迎！

分析這個 Rust 程序

現在，讓我們回過頭來仔細看看“`Hello, world!`”程序中到底發生了什麼。這是第一塊拼圖：

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

這幾行定義了一個 Rust 函數。`main` 函數是一個特殊的函數：在可執行的 Rust 程序中，它總是最先運行的代碼。第一行代碼聲明了一個叫做 `main` 的函數，它沒有參數也沒有返回值。如果有參數的話，它們的名稱應該出現在小括號中，`()`。

還須注意，函數體被包裹在花括號中，`{}`。Rust 要求所有函數體都要用花括號包裹起來。一般來說，將左花括號與函數聲明置於同一行並以空格分隔，是良好的代碼風格。

在編寫本書的時候，一個叫做 `rustfmt` 的自動格式化工具正在開發中。如果你希望在 `Rust` 項目中保持一種標準風格，`rustfmt` 會將代碼格式化為特定的風格。`Rust` 團隊計劃最終將該工具包含在標準 `Rust` 發行版中，就像 `rustc`。所以根據你閱讀本書的時間，它可能已經安裝到你的電腦中了！檢查在線文檔以了解更多細節。

在 `main()` 函數中是如下代碼：

```
println!("Hello, world!");
```

這行代碼完成這個簡單程序的所有工作：在屏幕上打印文本。這裡有四個重要的細節需要注意。首先 `Rust` 的縮進風格使用 4 個空格，而不是 1 個製表符（`tab`）。

第二，`println!` 調用了一個 `Rust` 宏（`macro`）。如果是調用函數，則應輸入 `println`（沒有`!`）。我們將在第十九章詳細討論宏。現在你只需記住，當看到符號`!`的時候，就意味著調用的是宏而不是普通函數。

第三，`"Hello, world!"` 是一個字符串。我們把這個字符串作為一個參數傳遞給 `println!`，字符串將被打印到屏幕上。

第四，該行以分號結尾（`;`），這代表一個表達式的結束和

下一個表達式的開始。大部分 Rust 代碼行以分號結尾。

編譯和運行是彼此獨立的步驟

你剛剛運行了一個新創建的程序，那麼讓我們檢查此過程中的每一個步驟。

在運行 Rust 程序之前，必須先使用 Rust 編譯器編譯它，即輸入 `rustc` 命令並傳入源文件名稱，如下：

```
$ rustc main.rs
```

如果你有 C 或 C++ 背景，就會發現這與 `gcc` 和 `clang` 類似。編譯成功後，Rust 會輸出一個二進制的可執行文件。

在 Linux、macOS 或 Windows 的 PowerShell 上，在 shell 中輸入 `ls` 命令可以看見這個可執行文件。在 Linux 和 macOS，你會看到兩個文件。在 Windows PowerShell 中，你會看到同使用 CMD 相同的三個文件。

```
$ lsmain main.rs
```

在 Windows 的 CMD 上，則輸入如下內容：

```
> dir /B %= the /B option says to only show the  
file names =%main.exemain.pdbmain.rs
```

這展示了擴展名為 `.rs` 的源文件、可執行文件（在 Windows 下是 `main.exe`，其它平台是 `main`），以及當使用 CMD 時會

有一個包含調試信息、擴展名為`.pdb`的文件。從這裡開始運行 `main` 或 `main.exe` 文件，如下：

```
$ ./main # Windows 是 .\main.exe
```

如果 `main.rs` 是上文所述的“Hello, world!”程序，它將會在終端上打印 `Hello, world!`。

如果你更熟悉動態語言，如 Ruby、Python 或 JavaScript，則可能不習慣將編譯和運行分為兩個單獨的步驟。Rust 是一種**預編譯靜態類型**（*ahead-of-time compiled*）語言，這意味著你可以編譯程序，並將可執行文件送給其他人，他們甚至不需要安裝 Rust 就可以運行。如果你給他人一個`.rb`、`.py` 或`.js` 文件，他們需要先分別安裝 Ruby，Python，JavaScript 實現（運行時環境，VM）。不過在這些語言中，只需要一句命令就可以編譯和運程序。這一切都是語言設計上的權衡取捨。

僅僅使用 `rustc` 編譯簡單程序是沒問題的，不過隨著項目的增長，你可能需要管理你項目的方方面面，並讓代碼易於分享。接下來，我們要介紹一個叫做 **Cargo** 的工具，它會幫助你編寫真實世界中的 Rust 程序。

參考資料:<https://www.rust-lang.org/zh-TW/tools/install>

使用 Cargo 創建項目

我們使用 Cargo 創建一個新項目，然後看看與上面的 Hello, world! 項目有什麼不同。回到 *projects* 目錄（或者你存放代碼的目錄）。接著，可在任何操作系統下運行以下命令：

```
$ cargo new hello_cargo$ cd hello_cargo
```

第一行命令新建了名為 *hello_cargo* 的目錄。我們將項目命名為 *hello_cargo*，同時 Cargo 在一個同名目錄中創建項目文件。

進入 *hello_cargo* 目錄並列出文件。將會看到 Cargo 生成了兩個文件和一個目錄：一個 *Cargo.toml* 文件，一個 *src* 目錄，以及位於 *src* 目錄中的 *main.rs* 文件。它也在 *hello_cargo* 目錄初始化了一個 git 倉庫，以及一個 *.gitignore* 文件。

注意：Git 是一個常用的版本控制系統（version control system，VCS）。可以通過 `--vcs` 參數使 `cargo new` 切換到其它版本控制系統（VCS），或者不使用 VCS。運行 `cargo new --help` 參看可用的選項。

請自行選用文本編輯器打開 *Cargo.toml* 文件。它應該看起

來如示例 1-2 所示：

文件名: Cargo.toml

```
[package]name      =      "hello_cargo"version      =  
"0.1.0"authors      =      ["Your      Name  
<you@example.com>"]edition      =  
"2018"[dependencies]
```

示例 1-2: *cargo new* 命令生成的 *Cargo.toml* 的內容

這個文件使用 TOML (*Tom's Obvious, Minimal Language*) 格式，這是 Cargo 配置文件的格式。

第一行，**[package]**，是一個片段 (**section**) 標題，表明下面的語句用來配置一個包。隨著我們在這個文件增加更多的信息，還將增加其他片段 (**section**)。

接下來的四行設置了 Cargo 編譯程序所需的配置：項目的名稱、版本、作者以及要使用的 Rust 版本。Cargo 從環境中獲取你的名字和 email 信息，所以如果這些信息不正確，請修改並保存此文件。附錄 E 會介紹 **edition** 的值。

最後一行，**[dependencies]**，是羅列項目依賴的片段的開始。在 Rust 中，代碼包被稱為 *crates*。這個項目並不需要其他的 crate，不過在第二章的第一個項目會用到依賴，那

時會用得上這個片段。

現在打開 *src/main.rs* 看看：

文件名: *src/main.rs*

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

Cargo 為你生成了一個“Hello, world!”程序，正如我們之前編寫的示例 1-1！目前為止，之前項目與 Cargo 生成項目的區別是 Cargo 將代碼放在 *src* 目錄，同時項目根目錄包含一個 *Cargo.toml* 配置文件。

Cargo 期望源文件存放在 *src* 目錄中。項目根目錄只存放 README、license 信息、配置文件和其他跟代碼無關的文件。使用 Cargo 幫助你保持項目乾淨整潔，一切井井有條。

如果沒有使用 Cargo 開始項目，比如我們創建的 Hello,world! 項目，可以將其轉化為一個 Cargo 項目。將代碼放入 *src* 目錄，並創建一個合適的 *Cargo.toml* 文件。

構建並運行 Cargo 項目

現在讓我們看看通過 Cargo 構建和運行“Hello, world!”程序有什麼不同！在 *hello_cargo* 目錄下，輸入下面的命令來構建項目：

```
$ cargo build    Compiling hello_cargo v0.1.0
```

```
(file:///projects/hello_cargo)           Finished
dev [unoptimized + debuginfo] target(s) in 2.85
secs
```

這個命令會創建一個可執行文件 `target/debug/hello_cargo`（在 Windows 上是 `target\debug\hello_cargo.exe`），而不是放在目前目錄下。可以通過這個命令運行可執行文件：

```
$ ./target/debug/hello_cargo # 或者在 Windows
下 为      .\target\debug\hello_cargo.exeHello,
world!
```

如果一切順利，終端上應該會打印出 `Hello, world!`。首次運行 `cargo build` 時，也會使 Cargo 在項目根目錄創建一個新文件：`Cargo.lock`。這個文件記錄項目依賴的實際版本。這個項目並沒有依賴，所以其內容比較少。你自己永遠也不需要碰這個文件，讓 Cargo 處理它就行了。

我們剛剛使用 `cargo build` 構建了項目，並使用 `./target/debug/hello_cargo` 運行了程序，也可以使用 `cargo run` 在一個命令中同時編譯並運行生成的可執行文件：

```
$ cargo run           Finished dev [unoptimized +
```

```
debuginfo] target(s) in 0.0 secs      Running  
`target/debug/hello_cargo`Hello, world!
```

注意這一次並沒有出現表明 Cargo 正在編譯 hello_cargo 的輸出。Cargo 發現文件並沒有被改變，就直接運行了二進製文件。如果修改了源文件的話，Cargo 會在運行之前重新構建項目，並會出現像這樣的輸出：

```
$ cargo run      Compiling hello_cargo v0.1.0  
(file:///projects/hello_cargo)      Finished  
dev [unoptimized + debuginfo] target(s) in 0.33  
secs      Running  
`target/debug/hello_cargo`Hello, world!
```

Cargo 還提供了一個叫 cargo check 的命令。該命令快速檢查代碼確保其可以編譯，但並不產生可執行文件：

```
$ cargo check    Checking hello_cargo v0.1.0  
(file:///projects/hello_cargo)      Finished  
dev [unoptimized + debuginfo] target(s) in 0.32  
secs
```

為什麼你會不需要可執行文件呢？通常 cargo check 要比 cargo build 快得多，因為它省略了生成可執行文件

的步驟。如果你在編寫代碼時持續的進行檢查，`cargo check` 會加速開發！為此很多 Rustaceans 編寫代碼時定期運行 `cargo check` 確保它們可以編譯。當準備好使用可執行文件時才運行 `cargo build`。

參考資料：<https://kaisery.gitbooks.io/trpl-zh-cn/content/ch01-03-hello-cargo.html>